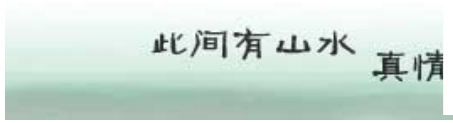
多线程

面向对象程序设计

第 12 讲 OOP 编程 - 多线程

刘进

2230652597@qq.com

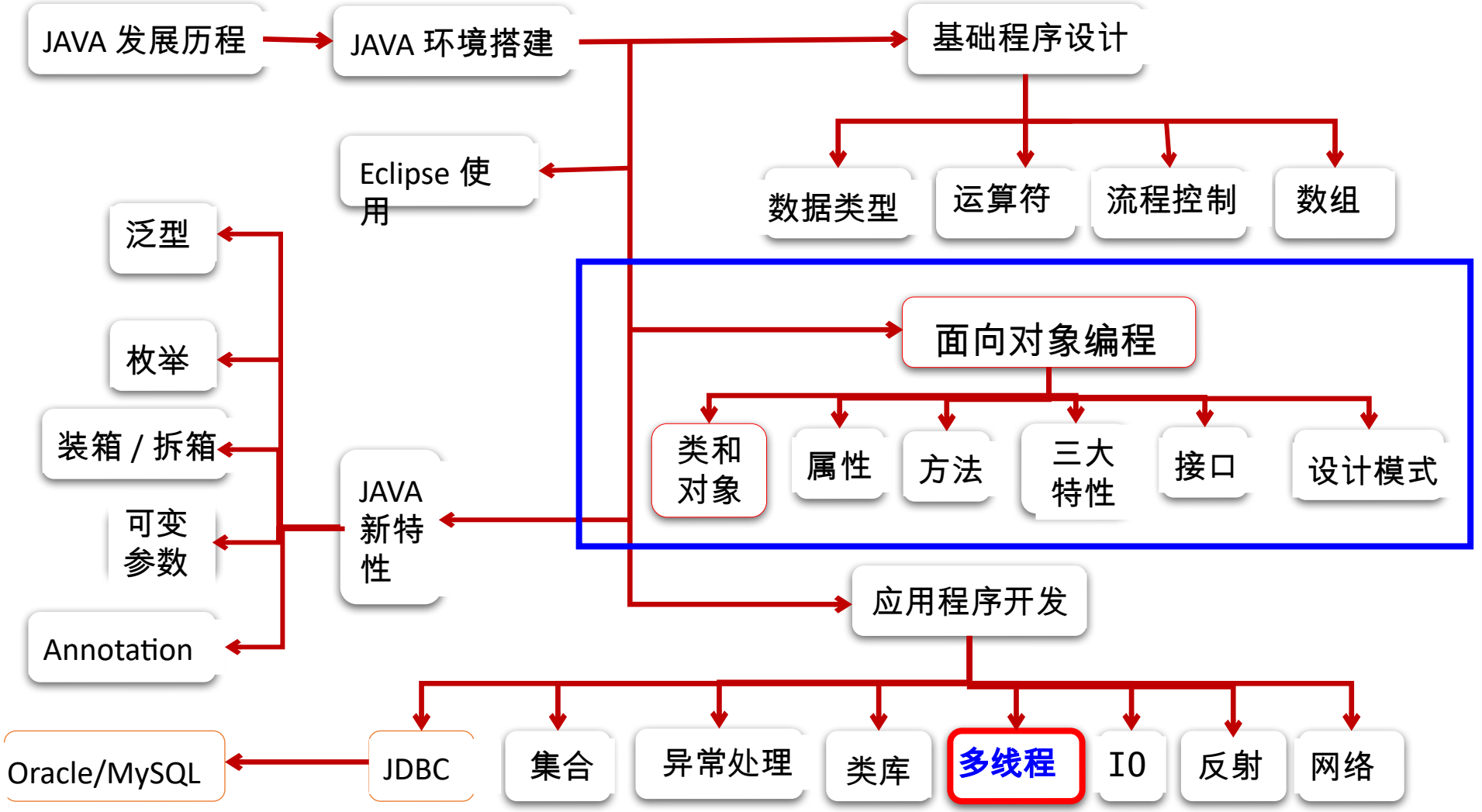


OOP 教辅 2022 秋季 QQ 群：

305915615

群名称：OOP教辅2022秋季
群 号：305915615

Java 基础知识图解



路线图

课程内容

- 程序、进程、线程的概念
- **Java 中多线程的创建和使用**
  - 继承 Thread 类与实现 Runnable 接口
  - Thread 类的主要方法
  - 线程的调度与设置优先级
- 线程的生命周期
- **线程的同步**
- 线程的通信

内容

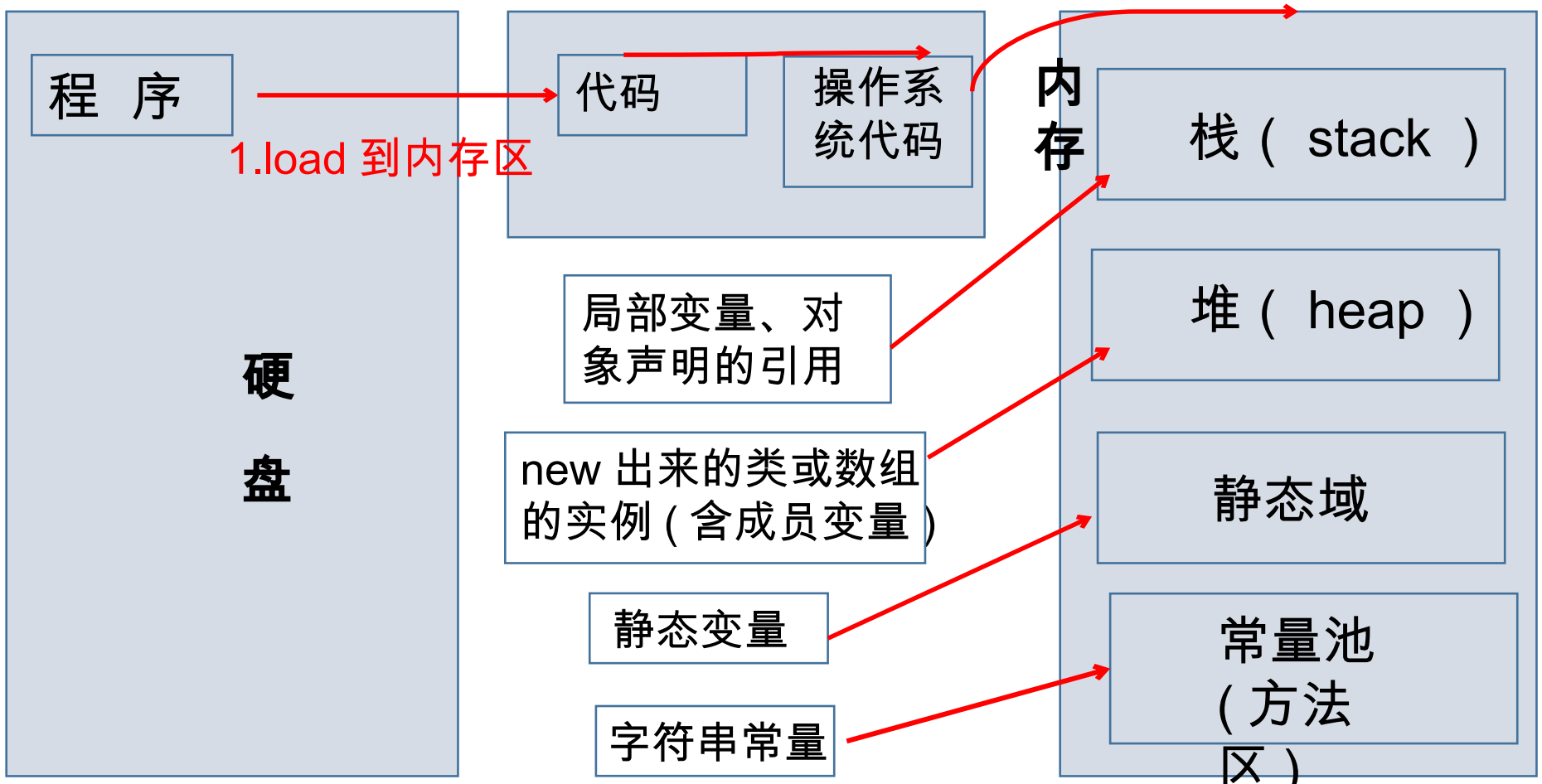
一、基本概念：程序 - 进程 - 线程

背景

（补充）程序的执行过程

2. 找到 main 方法开始执行

3. 执行过程中的内存管理



变量

内存中大致的存储方式

一、基本概念：程序 - 进程 - 线程

概念

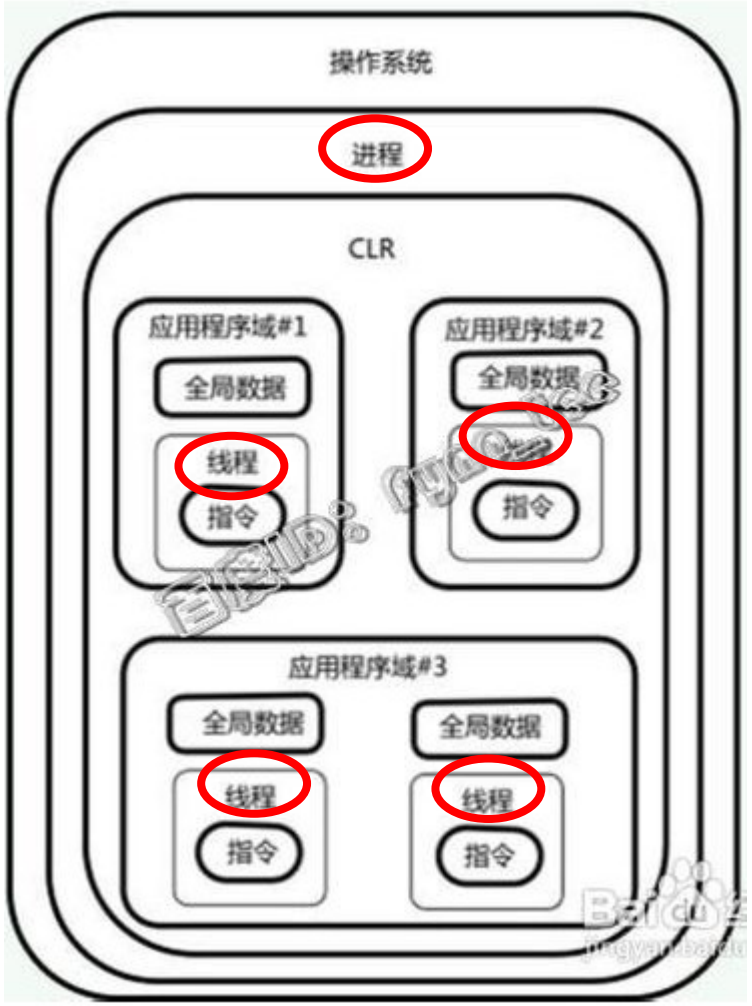
- **程序 (program)** 是为完成特定任务、用某种语言编写的一组指令的集合。即指一段**静态的代码**，静态对象。
- **进程 (process)** 是一个应用**程序在处理机上的一次执行过程，它是一个动态过程**：有它自身的产生、存在和消亡的过程。进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，**进程是系统进行资源分配和调度的一个独立单位**.
  - 如：运行中的 QQ ，运行中的 MP3 播放器
  - 程序是静态的，进程是动态的
- **线程 (thread)** ，**线程是进程中的一部分，进程包含多个线程在运行**。进程可进一步细化为线程，是一个程序内部的一条执行路径。
  - 若一个程序可同一时间执行多个线程，就是支持多线程的

一个程序至少有一个进程，一个进程至少有一个线程

进程与线程

概念

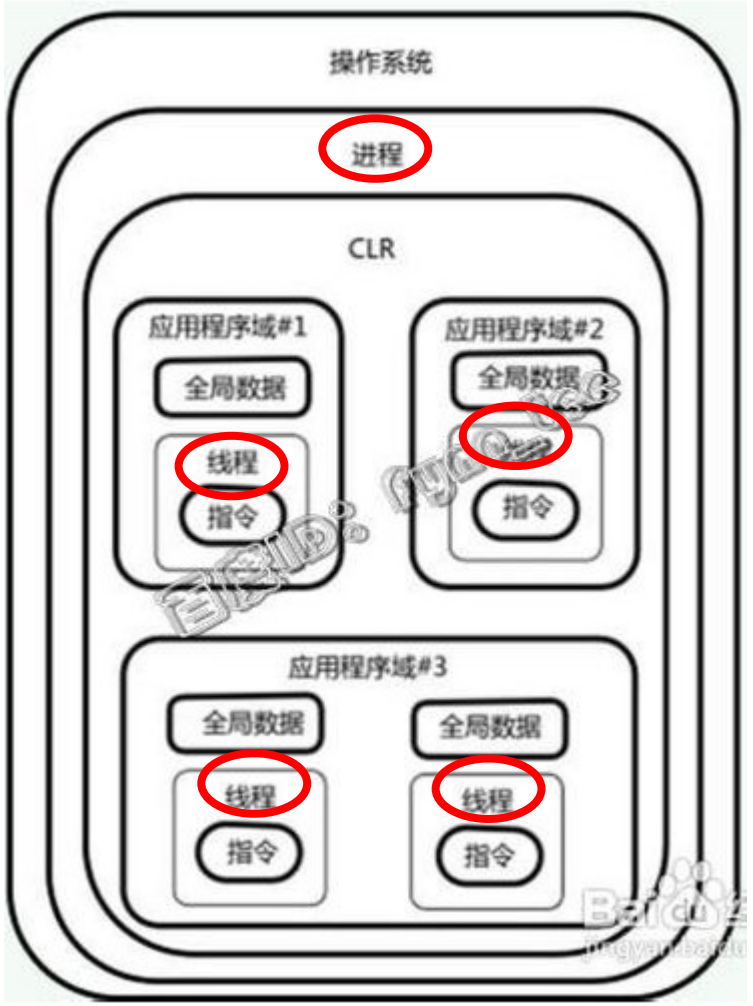
- 进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，是系统进行资源分配和调度的一个独立单位。
- 线程是进程的一个实体，是 CPU 调度和分派的基本单位，它是比进程更小的能独立运行的基本单位，自己基本上不拥有系统资源。相对进程而言，线程是一个更加接近于执行体的概念，它可以与同进程中的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。



进程与线程

区别
1、概念体系中的位置不一样
2、资源管理方式不一样

- 进程和线程的主要差别在于它们是不同的操作系统资源管理方式。**进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。**



进程与线程

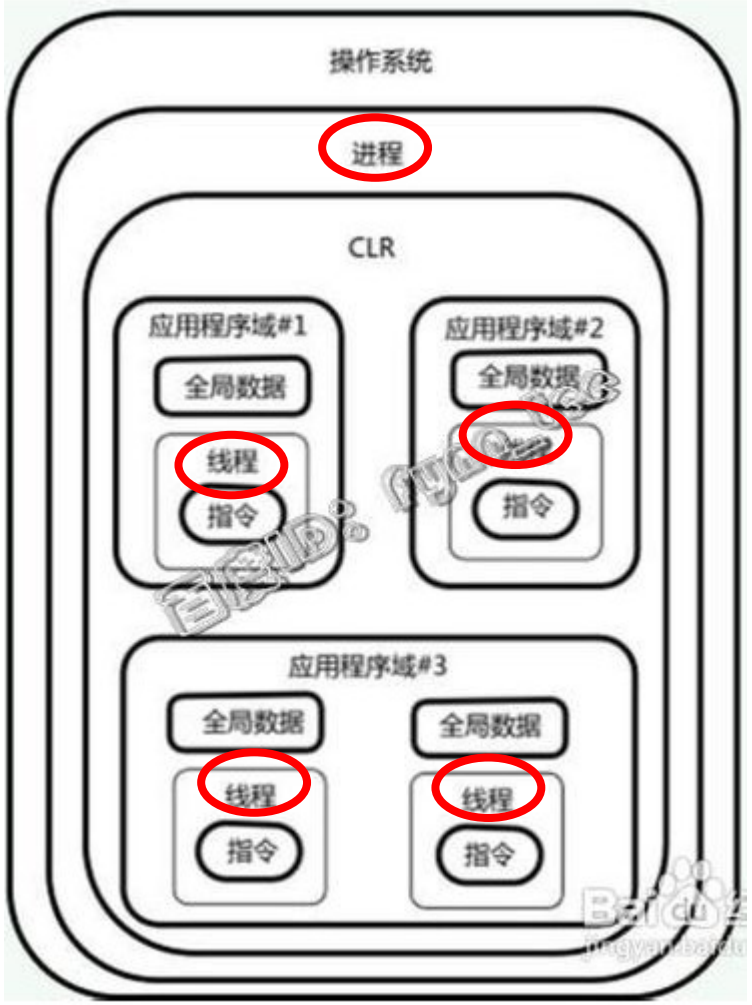
区别
2、资源管理方式不一样

线程与进程的区别归纳：

- **a. 地址空间和其它资源**：进程间相互独立，同一进程的各线程间共享。某进程内的线程在其它进程不可见。

- **b. 通信**：进程间通信 IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。

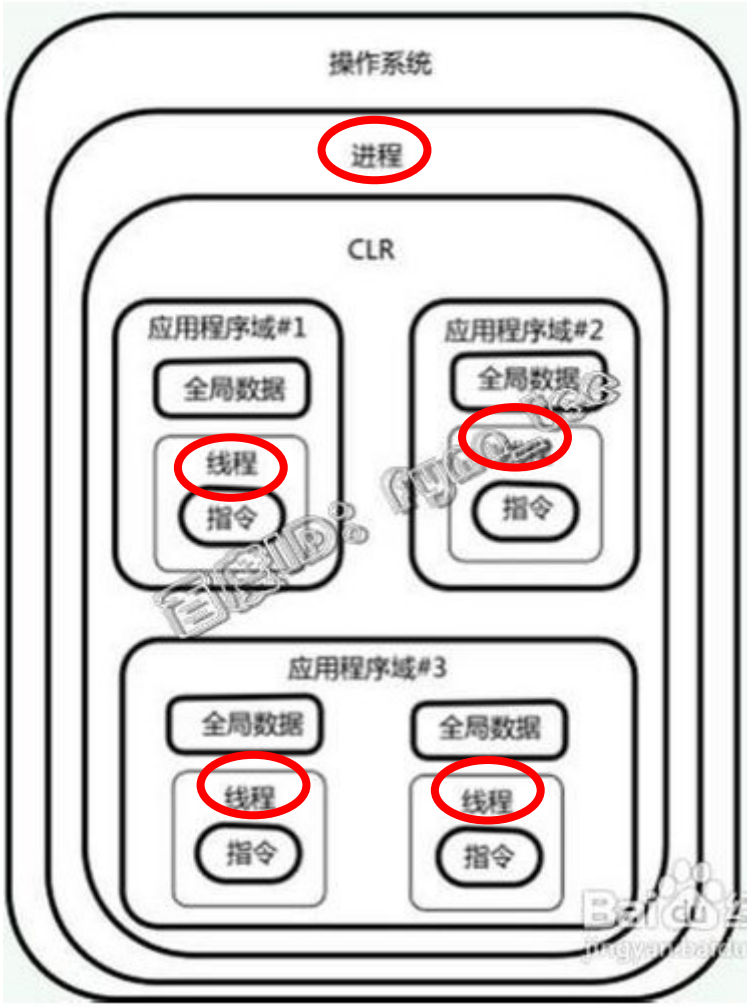
- **c. 调度和切换**：线程上下文切换比进程上下文切换要快得多。



进程与线程

区别
2、资源管理方式不一样

- 线程的划分尺度小于进程，使得多线程程序的并发性高。
- 进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。
- 线程在执行过程中与进程有区别：每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。
- 从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这是进程和线程的重要区别。

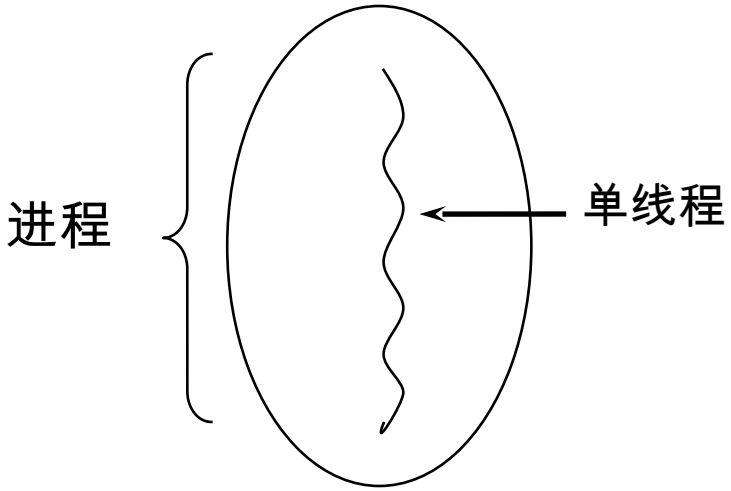


线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源的管理和保护；而进程正相反。同时，线程适合于在 SMP 机器上运行，而进程则可以跨机器迁移。

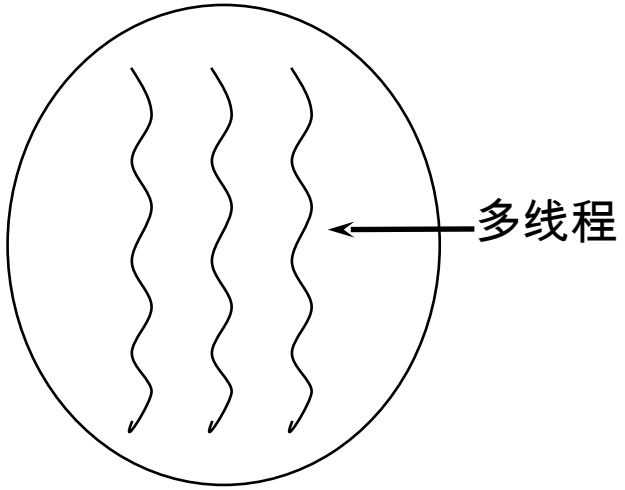
进程与多线程

传统与现实

传统进程



多线程进程



- 多线程：指的是这个程序（一个进程）运行时产生了不止一个线程

每个 Java 程序都有一个隐含的主线程： main 方法

Parallel 与 Concurrency

英文也博大精深
思维切进的维度不同
in time=timely 和 in real time
的故事

- Parallel 并行：多个 cpu 实例或者多台机器同时执行一段处理逻辑，是**真正的同时**。
- Concurrency 并发：通过 cpu 调度算法，让用户看上去同时执行，实际上从 cpu 操作层面**不是真正的同时**。并发往往在场景中有公用的资源，那么针对这个公用的资源往往产生瓶颈，我们会用 TPS 或者 QPS 来反映这个系统的处理能力。

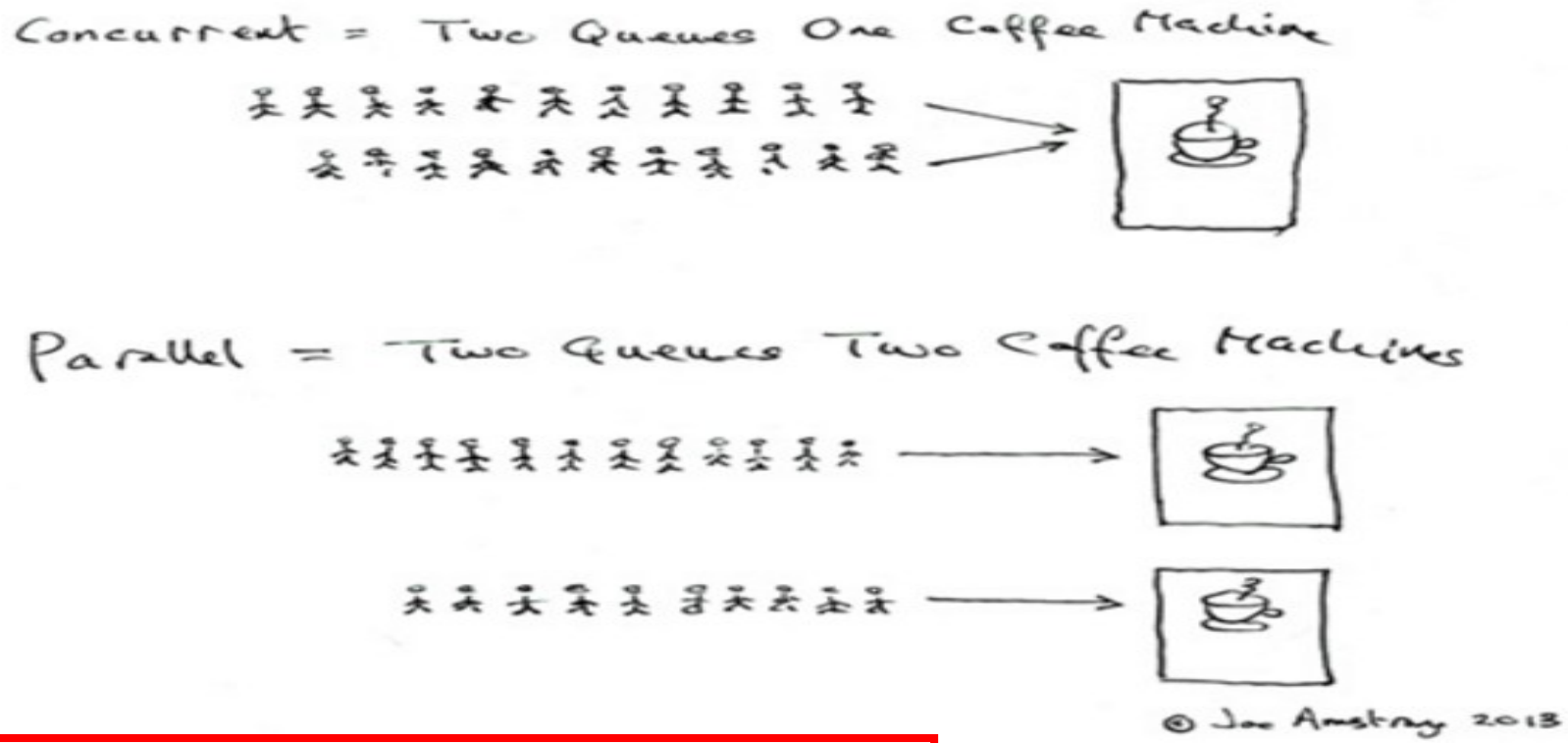
并行与并发

Concurrent and Parallel Programming

05 Apr 2013

What's the difference between concurrency and parallelism?

Explain it to a five year old.



Concurrent = Two queues and one coffee machine.

Parallel = Two queues and two coffee machines.

concurrency

parallel

线程安全与同步

并发与并行

- 线程安全：经常用来描绘一段代码。指在并发的情况之下，该代码经过多线程使用，线程的调度顺序不影响任何结果。这个时候使用多线程，我们只需要关注系统的内存，cpu是不是够用即可。反过来，线程不安全就意味着线程的调度顺序会影响最终结果，如不加事务的转账代码：

线程安全
结果唯一

```
void transferMoney(User from, User to, float amount){
  to.setMoney(to.getBalance() + amount);
  from.setMoney(from.getBalance() - amount);
}
```

- 同步：Java中的同步指的是通过人为的控制和调度，保证共享资源的多线程访问成为线程安全，来保证结果的准确。如上面的代码简单加入@synchronized关键字。在保证结果准确的同时，提高性能，才是优秀的程序。线程安全的优先级高于性能。

同步 ~ 线程安全

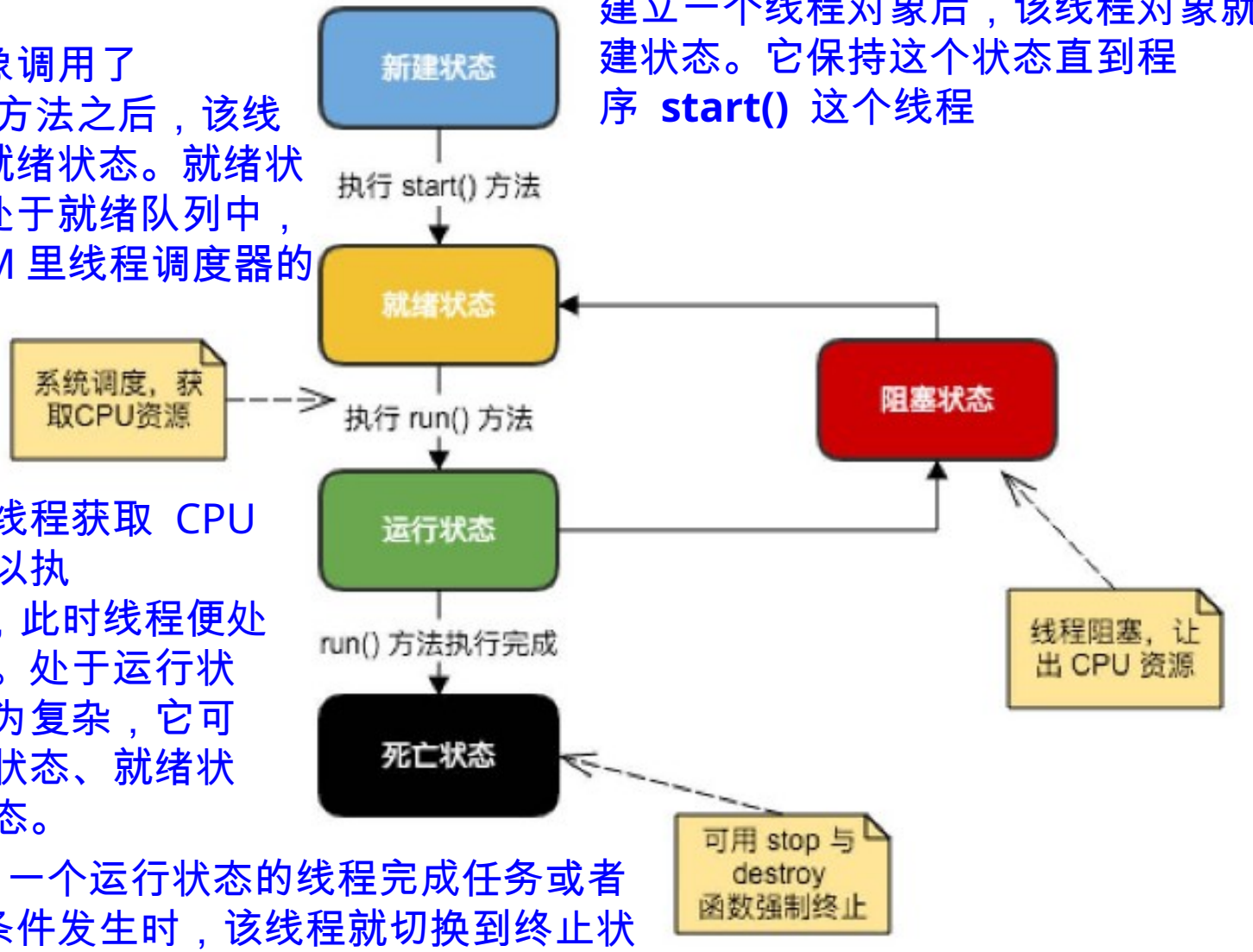
一个线程的生命周期

线程是一个动态执行的过程，它也有一个从产生到死亡的过程。

下图显示了一个线程完整的生命周期。

使用 **new** 关键字和 **Thread** 类或其子类建立一个线程对象后，该线程对象就处于新建状态。它保持这个状态直到程序 **start()** 这个线程

当线程对象调用了 **start()** 方法之后，该线程就进入就绪状态。就绪状态的线程处于就绪队列中，要等待 JVM 里线程调度器的调度。



就绪状态的线程获取 CPU 资源，就可以执行 **run()**，此时线程便处于运行状态。处于运行状态的线程最为复杂，它可以变为阻塞状态、就绪状态和死亡状态。

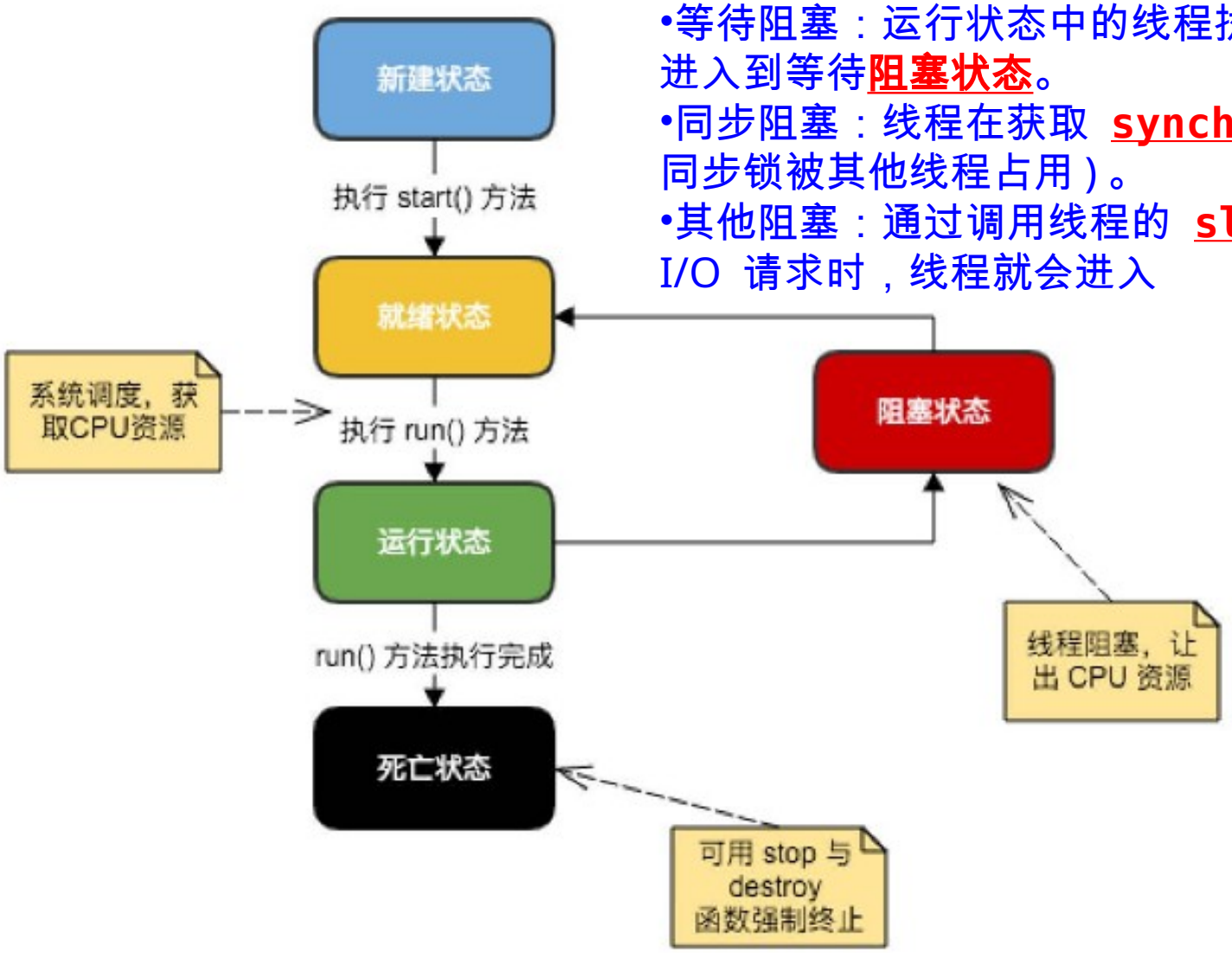
死亡状态：一个运行状态的线程完成任务或者其他终止条件发生时，该线程就切换到终止状态。

生命周期
用状态图描述
start()
run()
stop()

一个线程的生命周期

线程是一个动态执行的过程，它也有一个从产生到死亡的过程。

下图显示了一个线程完整的生命周期。



阻塞状态：如果一个线程执行了 **sleep（睡眠）**、**suspend（挂起）**等方法，失去所占用资源之后，该线程就从运行状态进入**阻塞状态**。在睡眠时间已到或获得设备资源后可以重新进入就绪状态。可以分为三种：

- 等待阻塞：运行状态中的线程执行 **wait()** 方法，使线程进入到等待**阻塞状态**。
- 同步阻塞：线程在获取 **synchronized** 同步锁失败（因为同步锁被其他线程占用）。
- 其他阻塞：通过调用线程的 **sleep()** 或 **join()** 发出了 I/O 请求时，线程就会进入

生命周期用状态图描述

何时需要多线程

- 程序需要同时执行两个或多个任务。
- 程序需要实现一些需要等待的任务时，如用户输入、文件读写操作、网络操作、搜索等。
- 需要一些后台运行的程序时。

什么场景下需要

Java 多线程编程

- Java 给**多线程编程提供了内置的支持**。一个多线程程序包含两个或多个能并发运行的部分。程序的每一部分都称作一个线程，并且每个线程定义了一个独立的执行路径。
- 多线程是多任务的一种特别的形式，但多线程使用了更小的资源开销。
- 这里定义和线程相关的另一个术语 - 进程：一个进程包括由操作系统分配的内存空间，包含一个或多个线程。一个线程不能独立的存在，它必须是进程的一部分。一个进程一直运行，直到所有的非守候线程都结束运行后才能结束。
- 多线程能满足程序员编写高效率的程序来达到充分利用 CPU 的目的。

Java 内置支持
随控布局

线程的优先级

每一个 Java 线程都有一个优先级，这样有助于操作系统确定线程的调度顺序。

Java 线程的优先级是一个整数，其取值范围是 1（Thread.MIN_PRIORITY）- 10（Thread.MAX_PRIORITY）。

默认情况下，每一个线程都会分配一个优先级 NORM_PRIORITY（5）。

具有较高优先级的线程对程序更重要，并且应该在低优先级的线程之前分配处理器资源。但是，线程优先级不能保证线程执行的顺序，而且非常依赖于平台。

优先级
1-10

创建一个线程

Java 提供了三种创建线程的方法：

- 通过实现 Runnable 接口；
- 通过继承 Thread 类本身；
- 通过 Callable 和 Future 创建线程。

创建线程 3 种方式
主要用前 2 种

多线程的创建和启动

- Java 语言的 JVM 允许程序运行多个线程，它通过 **java.lang.Thread** 类来实现。

- Thread 类的特性
  - 每个线程都是通过某个特定 Thread 对象的 **run()** 方法来完成操作的，经常把 **run()** 方法的主体称为**线程体**
  - 通过该 Thread 对象的 **start()** 方法来调用这个线程

创建多线程
通过 Thread 类
也是个画板和框架
run（）放线程主体
start（）启动线程对象

二、线程的创建和启动

MyThread.java

```
//MyThread.java
public class MyThread extends Thread {
    public void run() {
        System.out.println("MyThread running!");
    }
}
```

TestMyThread.java

```
//TestMyThread.java
public class TestMyThread {
    public static void main(String args[]) {
        MyThread myThread = new MyThread();
        myThread.start();
    }
}
```

Problems Console

```
<terminated> TestMyThread [Java Application] C:\Program File
MyThread running!
```

Java线程类也是一个object类,它的实例都继承自java.lang.Thread或其子类。 可以用如下方式用java中创建一个线程：

```
1 Tread thread = new Thread();
```

执行该线程可以调用该线程的start()方法:

```
1 thread.start();
```

在上面的例子中，我们并没有为线程编写运行代码，因此调用该方法后线程就终止了。

举例：
通过 Thread 类创建多线程
也是个画板和框架
run（）放线程主体

start（）启动线程对象

mt 子线程的创建和启动过程

```
class MyThread extends Thread{
    public MyThread(){
        super();
    }
    public void run(){
        for(int i = 0;i<100;i++){
            System.out.println("子线程："+i);
        }
    }
}
public class TestThread {
    public static void main(String[] args) {
        //1.创建线程
        MyThread mt = new MyThread();
        //2.启动线程；并调用当前线程的run()方法。
        mt.start();
```

3

1

2

举例：
通过 Thread 类创建多线程
也是个画板和框架
run（）放线程主体
start（）启动线程对象

Thread 类

- **构造方法**
  - **Thread()** ：创建新的 Thread 对象
  - **Thread(String threadname)** ：创建线程并指定线程实例名
  - **Thread(Runnable target)** ：指定创建线程的目标对象，它实现了 Runnable 接口中的 run 方法
  - **Thread(Runnable target, String name)** ：创建新的 Thread 对象

Thread 的构造器

创建线程的两种方式

1. 继承 Thread 类

1) 定义子类继承 Thread 类。

2) 子类中重写 Thread 类中的 run 方法。

3) 创建 Thread 子类对象，即创建了线程对象。

4) 调用线程对象 start 方法：启动线程，调用 run 方法。

通过 Thread 类创建多线程
也是个画板和框架
run（）放线程主体
start（）启动线程对象

*MyThread001.java | TestMyThread01.java

```java
//MyThread001.java
public class MyThread001 extends Thread {
    public MyThread001() {
        super();
    }
    public void run() {
        System.out.println("子线程进入运行状态！");
        for(int i=0;i<100;i++) {
            System.out.println("子线程-处于运行状态时的持续计数：  "+i);
        }
    }
}
```

举例：
通过 Thread 类创建多线程
也是个画板和框架
run（）放线程主体
start（）启动线程对象

*MyThread001.java | TestMyThread01.java

```java
//TestMyThread01.java
public class TestMyThread01 {
    public static void main(String[] args) {
        //创建线程
        MyThread001 mt = new MyThread001();
        //启动线程并调用当前线程的run()方法
        mt.start();
    }
}
```

Problems | Console

```
<terminated> TestMyThread01 [Java Application] C:\Prograr
子线程开始进入运行状态!
子线程-处于运行状态时的持续计数：  0
子线程-处于运行状态时的持续计数：  1
子线程-处于运行状态时的持续计数：  2
子线程-处于运行状态时的持续计数：  3
子线程-处于运行状态时的持续计数：  4
子线程-处于运行状态时的持续计数：  5
```

创建线程的两种方式

2. 实现 Runnable 接口

1 ）定义子类，实现 Runnable 接口。

2 ）子类中重写 Runnable 接口中的 run 方法。

3 ）**通过 Thread 类含参构造器创建线程对象**。

4 ）将 Runnable 接口的子类对象作为实际参数传递给 Thread 类的构造方法中。

- **Thread(Runnable target)** ：指定创建线程的目标对象，它实现了 Runnable 接口中的 run 方法
- **Thread(Runnable target, String name)** ：创建新的 Thread 对象

5 ）调用 Thread 类的 start 方法：开启线程，调用 Runnable 子类接口的 run 方法。

通过 Runable 接口创建多线程
也是个画板和框架
重写 Runable 接口的 run（）,放线程主体
将 Runable 接口的子类对象（实现类）作为实参传递给 Thread 的构造器

RunnableDemo.java | TestThread.java

```java
//RunnableDemo.java
class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;

   RunnableDemo( String name) {
      threadName = name;
      System.out.println("Creating " +  threadName );
   }

   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // 让线程睡眠一会
            Thread.sleep(50);
         }
      }catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
      }
      System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start () {
      System.out.println("Starting " +  threadName );
      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}
```

定义子类，实现 Runnable 接口

子类中重写 Runnable 接口中的 run 方法

举例：
通过 Runable 接口创建多线程
也是个画板和框架
重写 Runable 接口的 run（），放线程主体（装药）
将 Runable 接口的子类对象（实现类）作为实参传递给 Thread 的构造器

通过 Thread 类含参构造器创建线程对象

将 Runnable 接口的子类对象作为实际参数传递给 Thread 类的构造方法中。

- **Thread(Runnable target)**：指定创建线程的目标对象，它实现了 Runnable 接口中的 run 方法
- **Thread(Runnable tar… name)**：创建新的 Th…

调用 Thread 类的 start 方法：开启线程，调用 Runnable 子类接口的 run 方法

RunnableDemo.java | TestThread.java

```java
//RunnableDemo.java

class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;

   RunnableDemo( String name) {
      threadName = name;
      System.out.println("Creating " + threadName );
   }

   public void run() {
      System.out.println("Running " + threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + "," + i);
            // 让线程睡眠一会
            Thread.sleep(50);
         }
      }catch (InterruptedException e) {
         System.out.println("Thread " + threadName + " interrupted.");
      }
      System.out.println("Thread " + threadName + " exiting.");
   }

   public void start () {
      System.out.println("Starting " + threadName );
      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}
```

RunnableDemo.java | TestThread.java

```java
//TestThread.java

public class TestThread {
    public static void main(String args[]) {
        System.out.println("-----Show Thread-1-------");
        RunnableDemo R1 = new RunnableDemo( "Thread-1");
        R1.start();

        System.out.println("\n-----Show Thread-2-------");
        RunnableDemo R2 = new RunnableDemo( "Thread-2");
        R2.start();
    }
}
```

Problems | Console

```
<terminated> TestThread [Java
-----Show Thread-1-------
Creating Thread-1
Starting Thread-1

-----Show Thread-2-
Creating Thread-2
Starting Thread-2
Running Thread-1
Running Thread-2
Thread: Thread-1, 4
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-2 exiting.
Thread Thread-1 exiting.
```

Problems | Console

```
<terminated> TestThread [Java
-----Show Thread-1-------
Creating Thread-1
Starting Thread-1

-----Show Thread-2---
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-2, 2
Thread: Thread-1, 2
Thread: Thread-2, 1
Thread: Thread-1, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

继承方式和实现方式的联系与区别

public class **Thread** extends Object implements Runnable

【区别】

继承 Thread:　　线程代码存放 Thread 子类 run 方法中。

实现 Runnable ：线程代码存在接口的子类的 run 方法。

什么场景下需要

【实现方法的好处】

1 ）避免了单继承的局限性

2 ）多个线程可以共享同一个接口实现类的对象，非常适合多个相同线程来处理同一份资源。

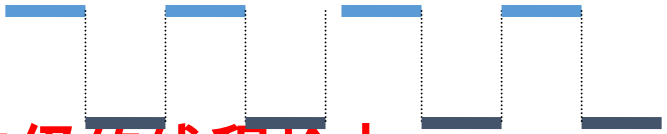
Thread 类的有关方法 (1)

- **void start():** 启动线程，并执行对象的 run() 方法
- **run():** 线程在被调度时执行的操作
- **String getName():** 返回线程的名称
- **void setName(String name)**: 设置该线程名称
- **static currentThread():** 返回当前线程

Thread 类的常用方法

线程的调度

- **调度策略**
  - 时间片



线程调度

  - **抢占式：高优先级的线程抢占 CPU**
- **Java 的调度方法**
  - 同优先级线程组成先进先出队列（先到先服务），使用时间片策略
  - 对高优先级，使用优先调度的抢占式策略

线程的优先级

- **线程的优先级控制**
  - **MAX_PRIORITY （ 10 ）；**
  - **MIN _PRIORITY （ 1 ）；**
  - **NORM_PRIORITY （ 5 ）；**
  - **◆涉及的方法：**
  - **getPriority() ：返回线程优先值**
  - **setPriority(int newPriority) ：改变线程的优先级**
  - 线程创建时继承父线程的优先级

线程优先级

Thread 类的有关方法 (2)

- **static void yield()**：线程让步
  - 暂停当前正在执行的线程，把执行机会让给优先级相同或更高的线程
  - 若队列中没有同优先级的线程，忽略此方法
- **join()** ：当某个程序执行流中调用其他线程的 join() 方法时，调用线程将被阻塞，直到 join() 方法加入的 join 线程执行完为止
  - 低优先级的线程也可以获得执行
- **static void sleep(long millis)**：（指定时间：毫秒）
  - 令当前活动线程在指定时间段内放弃对 CPU 控制，使其他线程有机会被执行，时间到后重排队。
  - **抛出 InterruptedException 异常**
- **stop()**: 强制线程生命期结束
- **boolean isAlive()** ：返回 boolean ，判断线程是否还活着

Thread 类的常用方法

使用多线程的优点

背景：只使用单个线程完成多个任务（调用多个方法），肯定比用多个线程来完成用的时间更短，为何仍需多线程呢？

多线程程序的优点：

1. 提高应用程序的响应。对图形化界面更有意义，可增强用户体验。
2. 提高计算机系统 CPU 的利用率
3. 改善程序结构。将既长又复杂的进程分为多个线程，独立运行，利于理解和修改

什么场景下需要

补充：线程的分类

Java 中的线程分为两类：一种是**守护线程**，一种是**用户线程**。

- 它们在几乎每个方面都是相同的，唯一的区别是判断 JVM 何时离开。
- 守护线程是用来服务用户线程的，通过在 start() 方法前调用 **thread.setDaemon(true)** 可以把一个用户线程变成一个守护线程。
- **Java 垃圾回收就是一个典型的守护线程。**
- 若 JVM 中都是守护线程，当前 JVM 将退出。

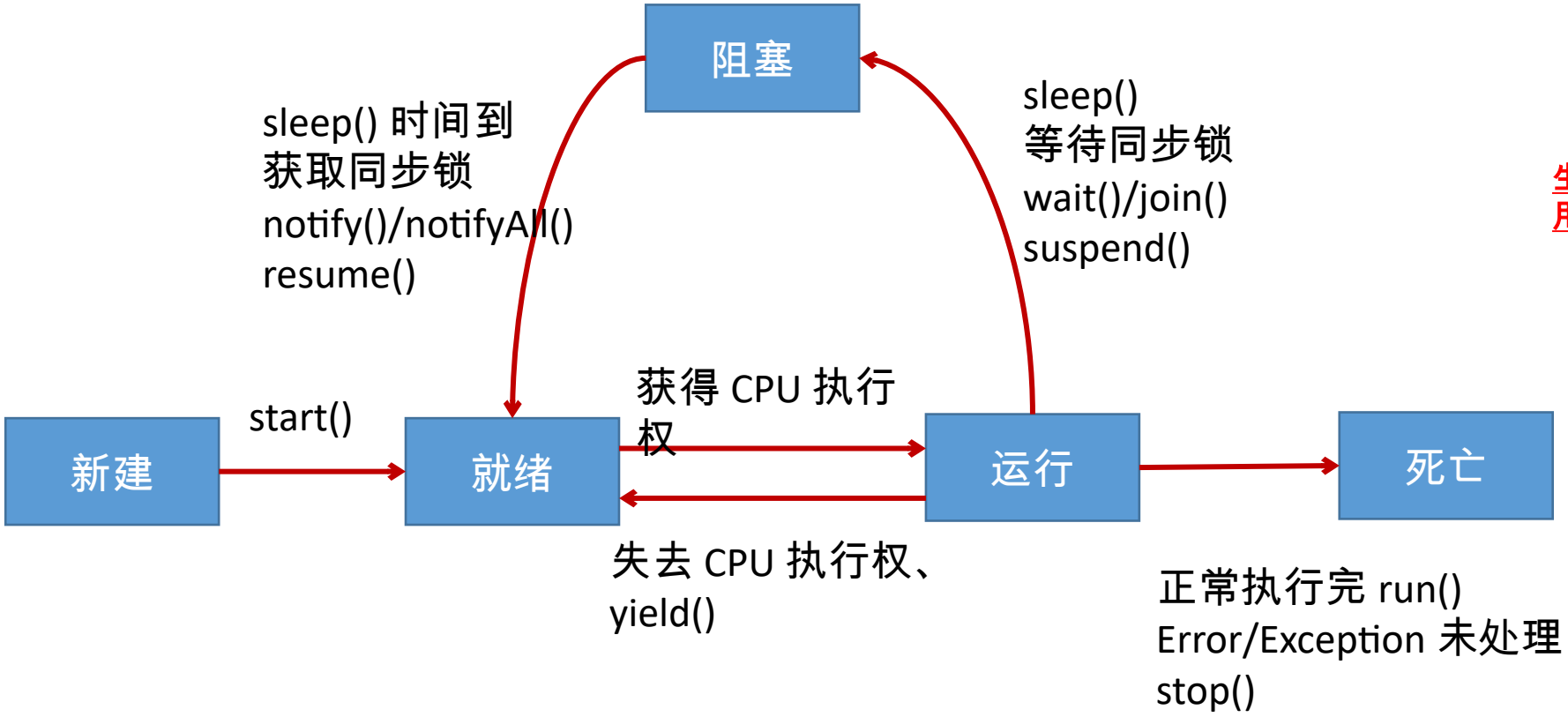
守护线程

三、线程的生命周期

Thread.State

JDK 中用 Thread.State 枚举表示了线程的几种状态

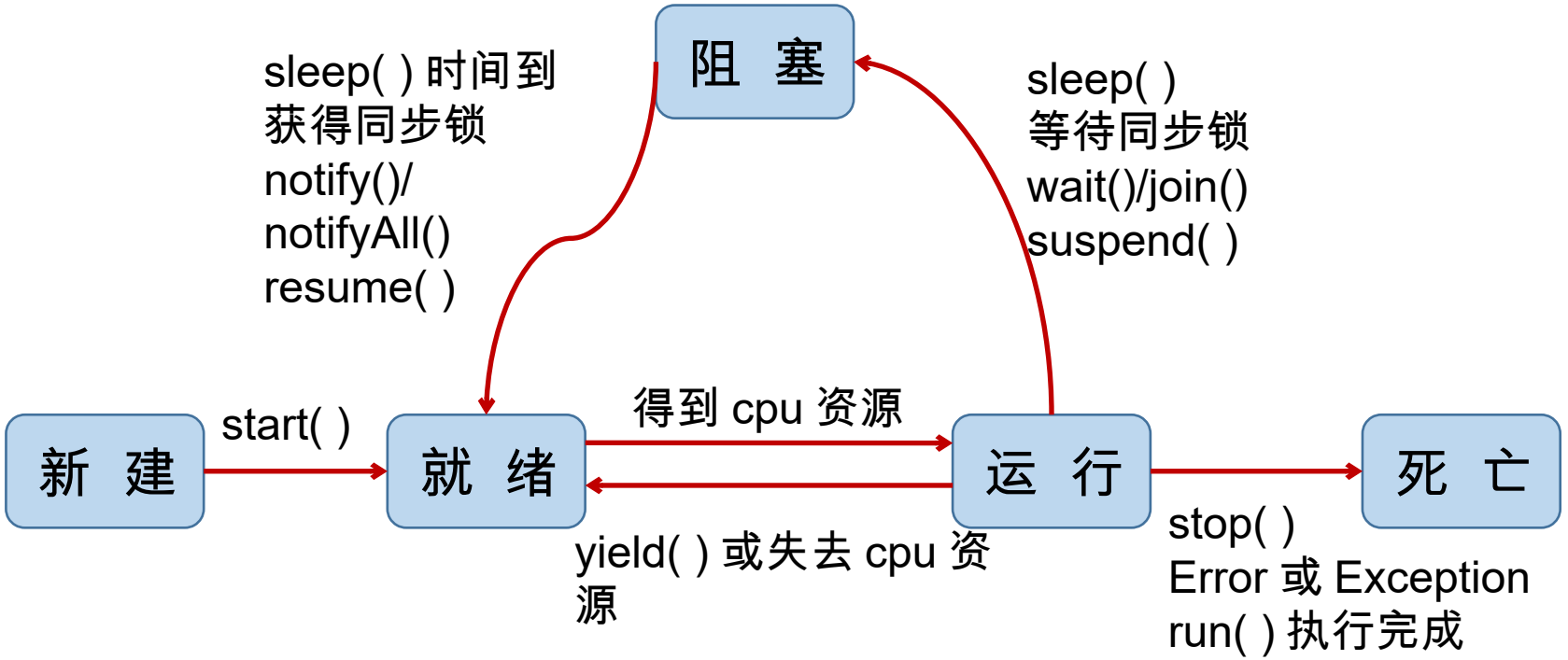
- 要想实现多线程，必须在主线程中创建新的线程对象。 Java 语言使用 Thread 类及其子类的对象来表示线程，在它的一个完整的生命周期中通常要经历如下的**五种状态**：
  - **新建：** 当一个 Thread 类或其子类的对象被声明并创建时，新生的线程对象处于新建状态
  - **就绪：**处于新建状态的线程被 start() 后，将进入线程队列等待 CPU 时间片，此时它已具备了运行的条件
  - **运行：**当就绪的线程被调度并获得处理器资源时，便进入运行状态，run() 方法定义了线程的操作和功能
  - **阻塞：**在某种特殊情况下，被人为挂起或执行输入输出操作时，让出 CPU 并临时中止自己的执行，进入阻塞状态
  - **死亡：**线程完成了它的全部工作或线程被提前强制性地中止



生命周期
用状态图描述

线程的生命周期

三、线程的生命周期



线程状态转换图

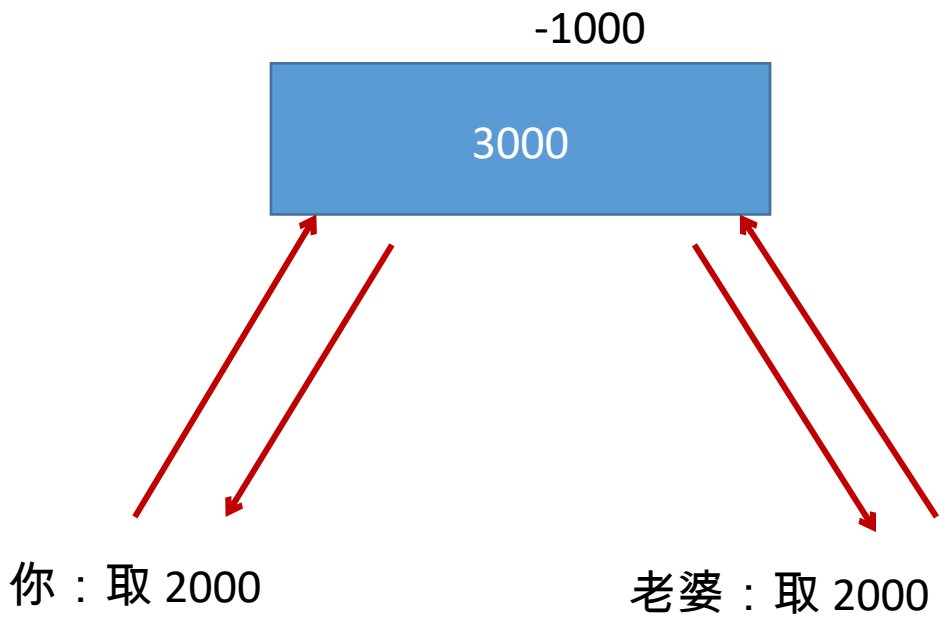
生命周期
用状态图描述

四、线程的同步

●问题的提出

线程安全问题

- ➢多个线程执行的不确定性引起执行结果的不稳定
- ➢多个线程对账本的共享，会造成操作的不完整性，会破坏数据。



Problems Console

<terminated> TestThread [Java

```
-----Show Thread-1-------
Creating Thread-1
Starting Thread-1

-----Show Thread-2-------
Creating Thread-2
Starting Thread-2
Running Thread-1
Running Thread-2
Thread: Thread-1, 4
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-2 exiting.
Thread Thread-1 exiting.
```

Problems Console

<terminated> TestThread [Java

```
-----Show Thread-1-------
Creating Thread-1
Starting Thread-1

-----Show Thread-2-------
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-2, 2
Thread: Thread-1, 2
Thread: Thread-2, 1
Thread: Thread-1, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

例 题

线程安全举例

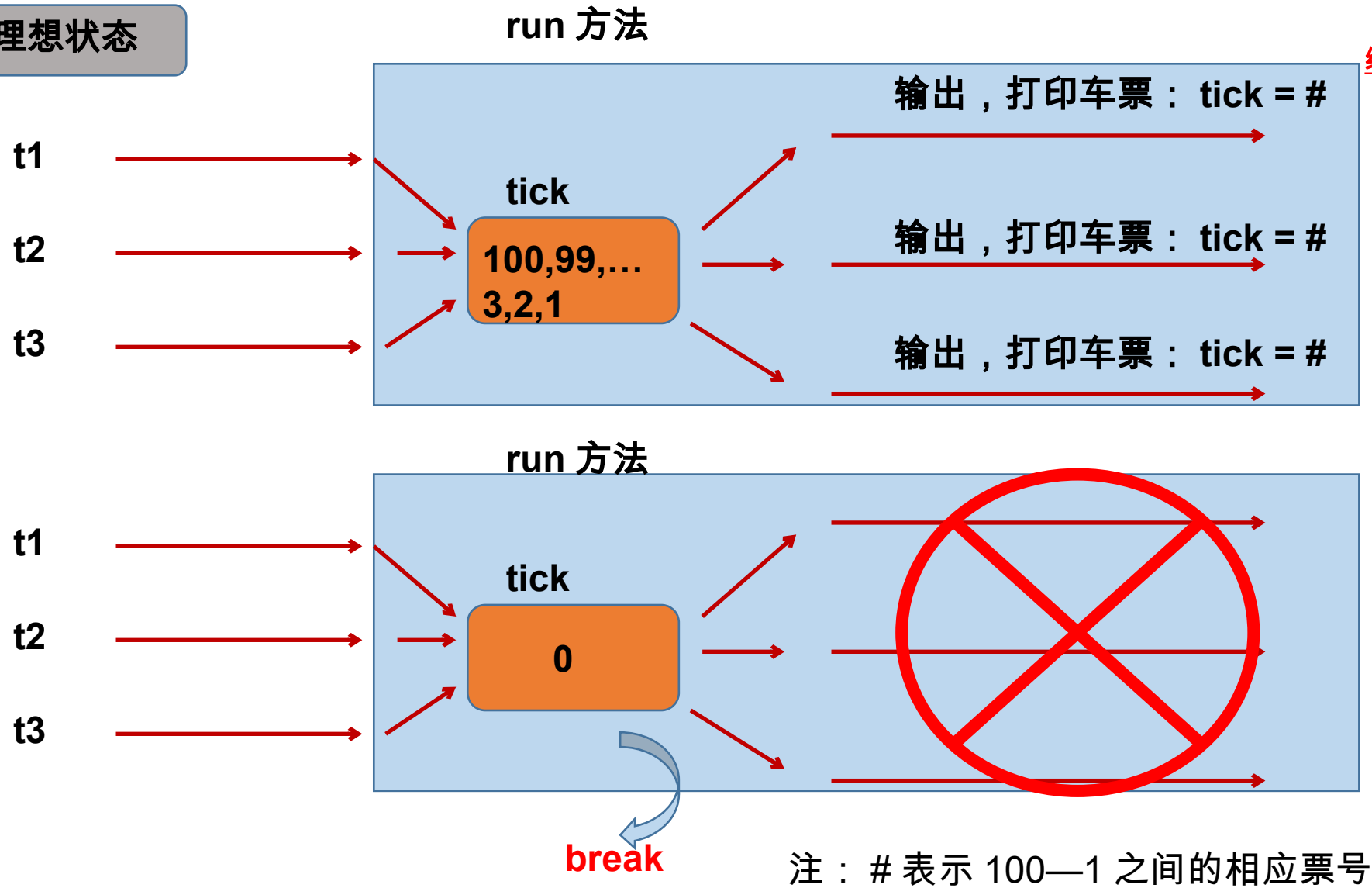
模拟火车站售票程序，开启三个窗口售票。

```
class Ticket implements Runnable{                                    }
   private  int tick = 100;
   public void run(){
   while(true){
      if(tick>0){

System.out.println(Thread.currentThread().getName()+ "售出车票， tick 号为： "+ tick--);
       }
      else
          break;
    }
  }
}

class  TicketDemo{
   public static void main(String[] args) {

   Ticket t = new Ticket();

   Thread t1 = new Thread(t);
   Thread t2 = new Thread(t);
   Thread t3 = new Thread(t);
    t1.setName("t1 窗口 ");
    t2.setName("t2 窗口 ");
    t3.setName("t3 窗口 ");
    t1.start();
    t2.start();
    t3.start();
   }
```

线程安全举例

线程安全举例

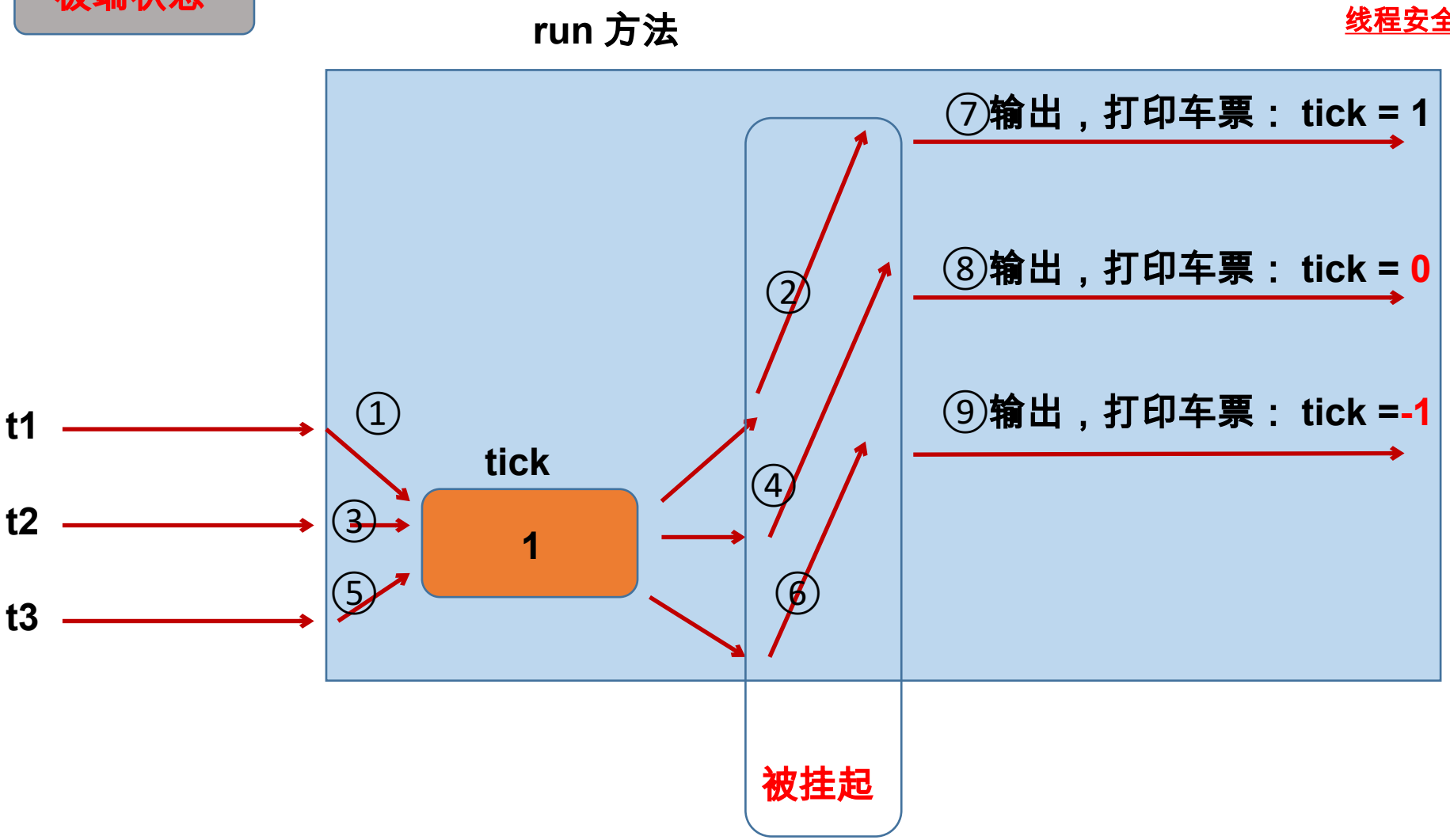
理想状态



注： # 表示 100—1 之间的相应票号

线程安全举例

极端状态


run 方法
t1
t2
t3
①
②
③
④
⑤
⑥
tick
1
⑦输出，打印车票： tick = 1
⑧输出，打印车票： tick = 0
⑨输出，打印车票： tick =-1
被挂起

```
private int tick = 100;
public synchronized void run(){
    while(true){
      if(tick>0){
        try{
          Thread.sleep(10);
        }catch(InterruptedException e){ e.printStackTrace();}
        System.out.println(Thread.currentThread().getName()+ “售出车票， tick 号为：

        "+tick--);
} } }
```

线程安全举例

1. 多线程出现了安全问题

2. **问题的原因**:当多条语句在操作同一个线程共享数据时，一个线程对多条语句只执行了一部分，还没有执行完，另一个线程参与进来执行。导致共享数据的错误。

3. **解决办法**：**对多条操作共享数据的语句，只能让一个线程都执行完，在执行过程中，其他线程不可以参与执行**。

线程安全语法

Synchronized 的使用方法

线程安全
语法

- Java 对于多线程的安全问题提供了专业的解决方式：
  同步机制

1. synchronized (对象) {
   // 需要被同步的代码；
   }

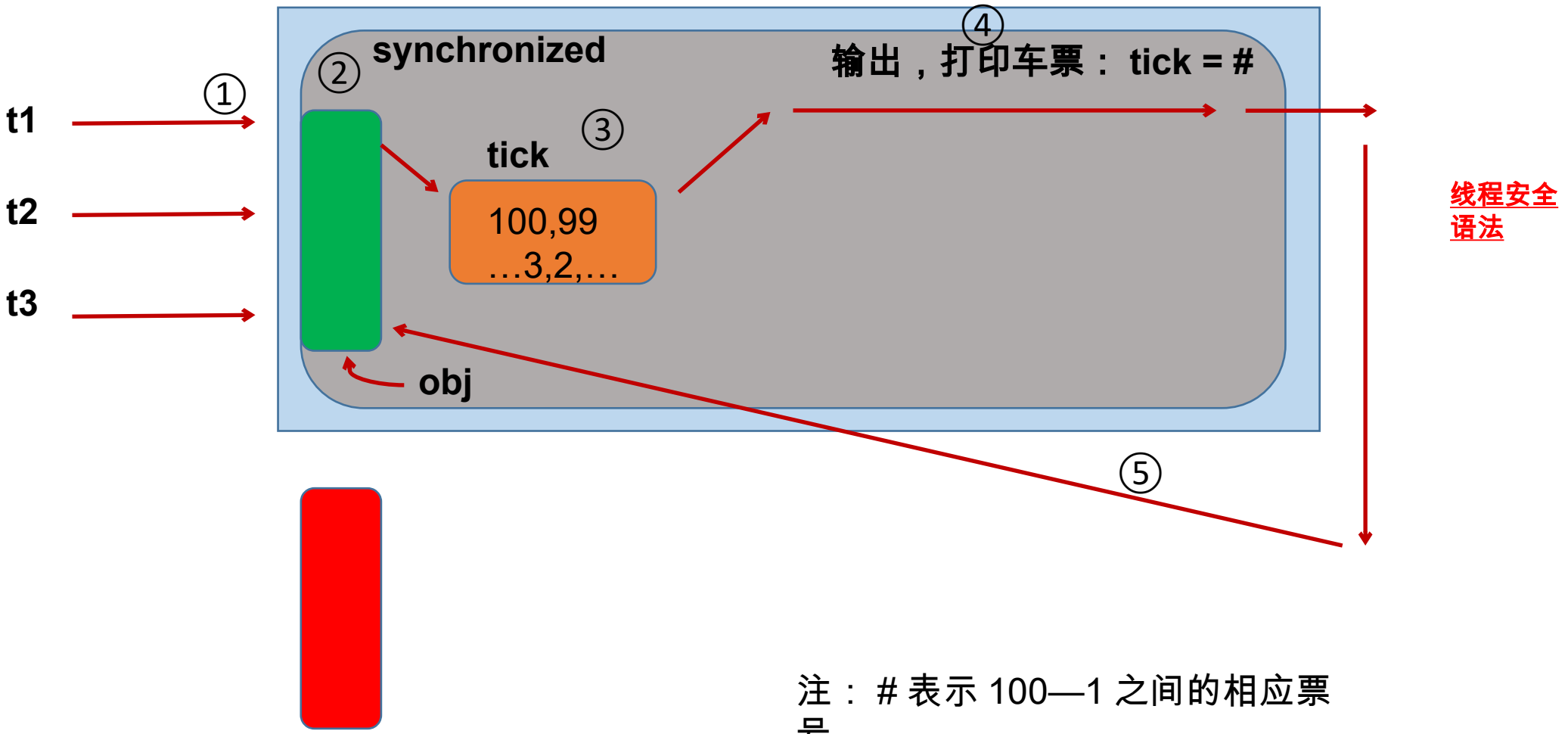
2. synchronized 还可以放在方法声明中，表示整个方法为同步方法。

例如：

```
public synchronized void show (String name){
        ....
}
```

分析同步原理

run 方法



注： # 表示 100—1 之间的相应票号

互斥锁

- 在 Java 语言中，引入了对象互斥锁的概念，来保证共享数据操作的完整性。
  - 每个对象都对应于一个可称为“互斥锁”的标记，这个标记用来保证在任一时刻，**只能有一个线程访问该对象**。
  - **关键字 synchronized 来与对象的互斥锁联系。当某个对象用 synchronized 修饰时，表明该对象在任一时刻只能由一个线程访问。**
  - **同步的局限性：导致程序的执行效率要降低**
  - **同步方法（非静态的）的锁为 this 。**
  - **同步方法（静态的）的锁为当前类本身。**

线程安全
互斥锁（操作系统常见）

synchronized, wait, notify 是任何对象都具有的同步工具。让我们先来了解他们

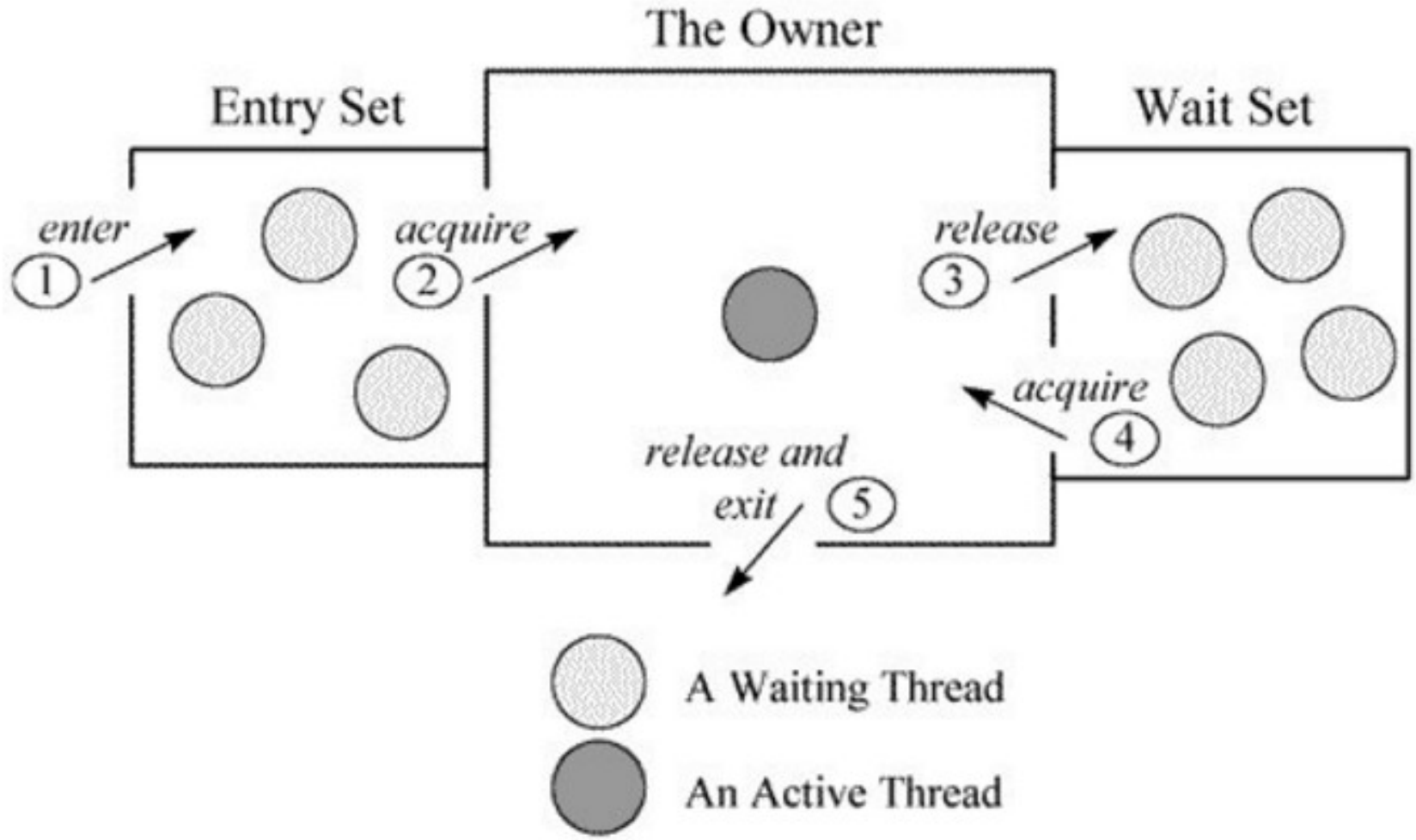


Figure 20-1. A Java monitor.

monitor

线程安全
互斥锁（操作系统常见）

他们是应用于同步问题的人工线程调度工具。讲其本质，首先就要明确monitor的概念，Java中的每个对象都有一个监视器，来监测并发代码的重入。在非多线程编码时该监视器不发挥作用，反之如果在synchronized 范围内，监视器发挥作用。

线程安全
互斥锁（操作系统常见）

wait/notify必须存在于synchronized块中。并且，这三个关键字针对的是同一个监视器（某对象的监视器）。这意味着wait之后，其他线程可以进入同步块执行。

```
private int tick = 100;
public synchronized void run(){
    while(true){
      if(tick>0){
        try{
          Thread.sleep(10);
        }catch(InterruptedException e){ e.printStackTrace();}
       System.out.println(Thread.currentThread().getName()+“售出车票，tick号为：
        "+tick--);
} } }
```

- synchronized单独使用：
  - 代码块：如下，在多线程环境下，synchronized块中的方法获取了lock实例的monitor，如果实例相同，那么只有一个线程能执行该块内容

```
public class Thread1 implements Runnable {
    Object lock;
    public void run() {
        synchronized(lock){
          ..do something
        }
    }
}
```

线程安全语法

  - 直接用于方法： 相当于上面代码中用lock来锁定的效果，实际获取的是Thread1类的monitor。更进一步，如果修饰的是static方法，则锁定该类所有实例。

```
public class Thread1 implements Runnable {
    public synchronized void run() {
         ..do something
    }
}
```

- 直接用于方法：相当于上面代码中用lock来锁定的效果，实际获取的是Thread1类的monitor。更进一步，如果修饰的是static方法，则锁定该类所有实例。

```
public class Thread1 implements Runnable {
    public synchronized void run() {
            ..do something
    }
}
```

线程安全
语法

```
private int tick = 100;
public synchronized void run(){
    while(true){
      if(tick>0){
        try{
          Thread.sleep(10);
        }catch(InterruptedException e){ e.printStackTrace();}
        System.out.println(Thread.currentThread().getName()+“售出车票，tick号为：
        "+tick--);
} } }
```

单例设计模式之懒汉式

```
class Singleton {
        private static Singleton instance = null;
        private Singleton(){}
        public static Singleton getInstance(){
                if(instance==null){
                        synchronized(Singleton.class){
                                if(instance == null){
                                        instance=new Singleton();
                                }       }       }
                return instance;
        }       }
public class TestSingleton{
        public static void main(String[] args){
                Singleton s1=Singleton.getInstance();
                Singleton s2=Singleton.getInstance();
                System.out.println(s1==s2);
        }       }
```

线程安全
懒汉模式
举例

小结：释放锁的操作

线程安全
锁的释放

- 当前线程的同步方法、同步代码块执行结束
- 当前线程在同步代码块、同步方法中遇到 break 、 return 终止了该代码块、该方法的继续执行。
- 当前线程在同步代码块、同步方法中出现了未处理的 Error 或 Exception ，导致异常结束
- 当前线程在同步代码块、同步方法中执行了线程对象的 **wait()** 方法，当前线程暂停，并释放锁。

小结：不会释放锁的操作

线程安全
不释放锁的操作
小心使用

- 线程执行同步代码块或同步方法时，程序调用 Thread.sleep() 、Thread.yield() 方法暂停当前线程的执行

- **线程执行同步代码块时，其他线程调用了该线程的 suspend() 方法将该线程挂起，该线程不会释放锁（同步监视器）。**
  - **应尽量避免使用 suspend() 和 resume() 来控制线程**

线程的死锁问题

线程安全
死锁

- 死锁
  - 不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，就形成了线程的死锁
- 解决方法
  - 专门的算法、原则
  - 尽量减少同步资源的定义

DeadLock.java

五、线程通信

- **wait() 与 notify() 和 notifyAll()**
  - **wait()** ：令当前线程挂起并放弃 CPU 、同步资源，使别的线程可访问并修改共享资源，而当前线程排队等候再次对资源的访问
  - **notify()** ：唤醒正在排队等待同步资源的线程中优先级最高者结束等待
  - **notifyAll ()** ：唤醒正在排队等待资源的所有线程结束等待 .
- **Java.lang.Object 提供的这三个方法只有在 synchronized 方法或 synchronized 代码块中才能使用**，否则会报 java.lang.IllegalMonitorStateException 异常

wait() 方法

- 在当前线程中调用方法： 对象名 .wait()
- 使当前线程进入等待（某对象）状态 ，直到另一线程对该对象发出 notify（或 notifyAll) 为止。
- 调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）
- **调用此方法后，当前线程将释放对象监控权 ，然后进入等待**
- 在当前线程被 notify 后，要重新获得监控权，然后从断点处继续代码的执行。

notify()/ notifyAll()

- 在当前线程中调用方法： 对象名 .notify()
- 功能：唤醒等待该对象监控权的一个线程。
- 调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）

例题：使用两个线程打印 1-100. 线程 1，线程 2 交替打印

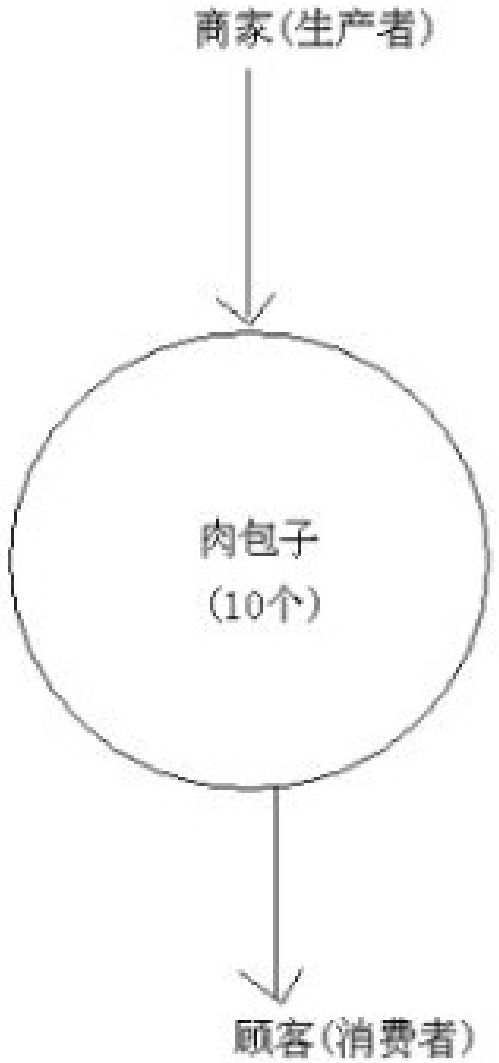
线程安全举例

```
class Communication implements Runnable{
static int i = 1;
public void run() {
while (true) {
 synchronized (this) {
 notify();
 if (i <= 100) {
  System.out.println(Thread.currentThread().getName() + ":"+ i++);
 }
 else   break;
 try {
 wait();}
 catch (InterruptedException e) {
}}}}}
```

经典例题：生产者 / 消费者问题

线程协同

生产消费者问题

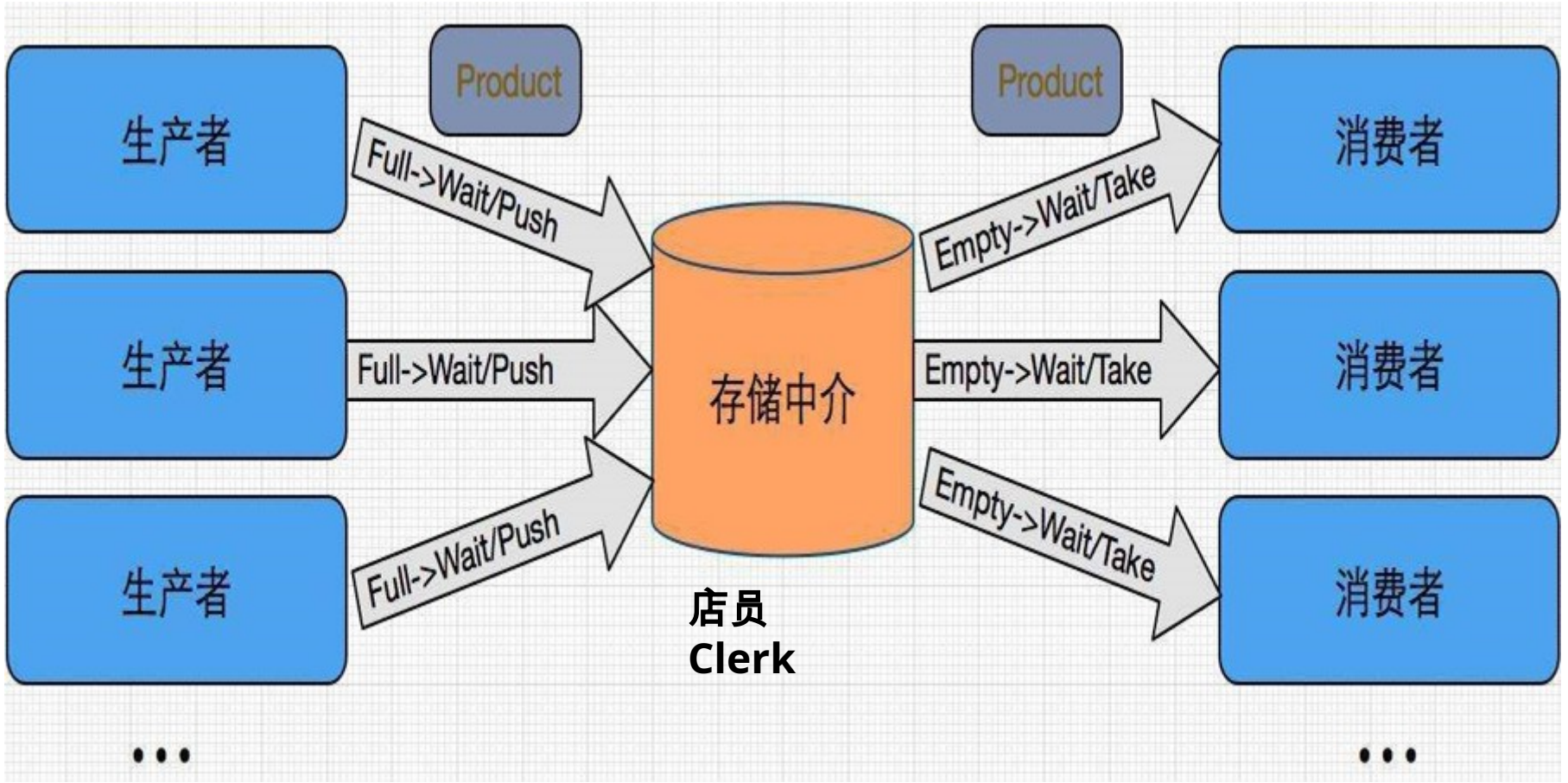


店员

线程间通信问题：
不同种类的线程间针对同一个资源的操作。

线程协同

生产消费者问题



经典例题：生产者 / 消费者问题

- 生产者 (Productor) 将产品交给店员 (Clerk) ，而消费者 (Customer) 从店员处取走产品，店员一次只能持有固定数量的产品 (比如 :20) ，如果生产者试图生产更多的产品，店员会叫生产者停一下，如果店中有空位放产品了再通知生产者继续生产；如果店中没有产品了，店员会告诉消费者等一下，如果店中有产品了再通知消费者来取走产品。
- 这里可能出现两个问题：
  - 生产者比消费者快时，消费者会漏掉一些数据没有取到。
  - 消费者比生产者快时，消费者会取相同的数据。

RunnableDemo.java | TestThread.java | Clerk.java | Produtor.java | Consumer.java | TestProduceCon

```java
//Clerk.java
public class Clerk {
    //商品容量
    int capacity = 20;
    //当前商品数量
    int product = 0;

    public synchronized void addProduct() {//生产产品
        if (product >= 20) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } else {
            product++;
            System.out.println(Thread.currentThread().getName() + ":生产了" + product + "个产品");
            notifyAll();
        }
    }

    public synchronized void consumeProduct() {//消费产品
        if (product <= 0) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } else {
            System.out.println(Thread.currentThread().getName() + ":消费了" + product + "个产品");
            product--;
            notifyAll();
        }
    }
}
```

线程协同

生产消费者问题

举例

Produtor.java | RunnableDemo.java | TestThread.java

```java
//Produtor.java
public class Produtor implements Runnable {
    Clerk clerk;

    public Produtor(Clerk clerk) {
        this.clerk = clerk;
    }

    public void run() {
        System.out.println("生产者生产产品");
        while (true) {
            try {//sleep一下，让效果更明显
                Thread.currentThread().sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            clerk.addProduct();
        }
    }
}
```

Produtor.java | Consumer.java | RunnableDemo.java

```java
//Consumer.java
public class Consumer implements Runnable {
    Clerk clerk;
    
    public Consumer(Clerk clerk) {
        this.clerk = clerk;
    }
    
    public void run() {
        System.out.println("消费者消费产品");
        while (true) {
            try {
                Thread.currentThread().sleep(20);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            clerk.consumeProduct();
        }
    }
}
```

通过 Runable 接口创建多线程
也是个画板和框架
重写 Runable 接口的 run（ ），放线程主体（装药）
将 Runable 接口的子类对象（实现类）作为实参传递给 Thread 的构造器

TestProduceConsume.java | Produtor.java | Consumer.j

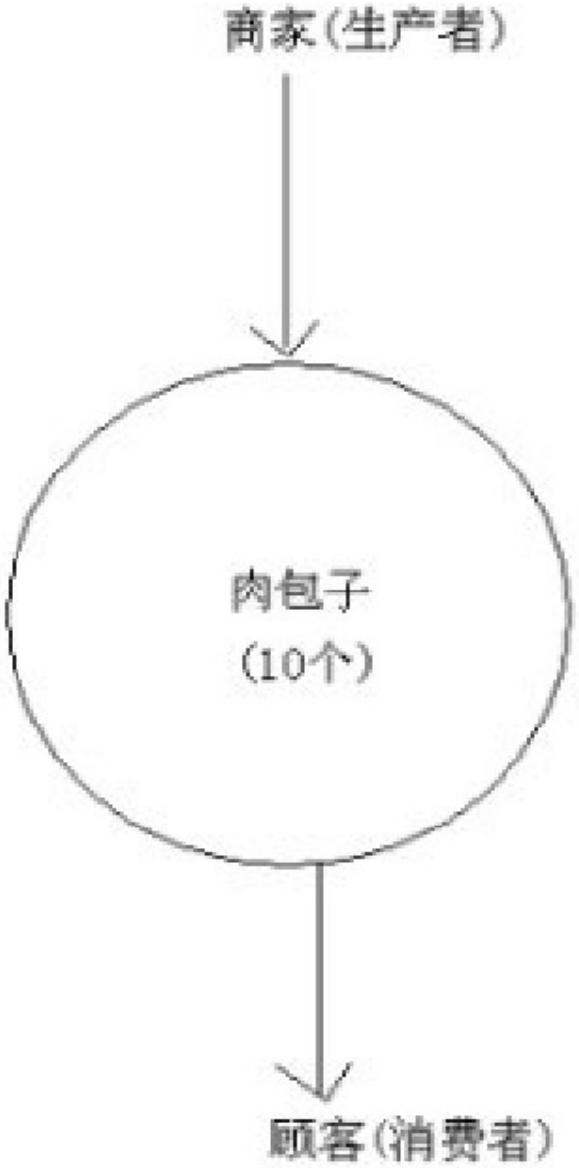
```java
//TestProduceConsume.java
public class TestProduceConsume {
    public static void main(String[] args) {
        Clerk clerk = new Clerk();
        Produtor p1 = new Produtor(clerk);
        Consumer c1 = new Consumer(clerk);
        Thread t1 = new Thread(p1);
        Thread t2 = new Thread(c1);
        t1.setName("生产者1");
        t2.setName("消费者1");
        t1.start();
        t2.start();
    }
}
```

举例：
通过 Runable 接口创建多线程
也是个画板和框架
重写 Runable 接口的 run（）,放线程主体（装药）
将 Runable 接口的子类对象（实现类）作为实参传递给 Thread 的构造器

Problems Console

TestProduceConsume [Java Application] C:\P

```
消费者消费产品
生产者生产产品
生产者1:生产了1个产品
生产者1:生产了2个产品
消费者1:消费了2个产品
生产者1:生产了2个产品
消费者1:消费了2个产品
生产者1:生产了2个产品
生产者1:生产了3个产品
消费者1:消费了3个产品
生产者1:生产了3个产品
生产者1:生产了4个产品
消费者1:消费了4个产品
生产者1:生产了4个产品
生产者1:生产了5个产品
消费者1:消费了5个产品
生产者1:生产了5个产品
生产者1:生产了6个产品
消费者1:消费了6个产品
生产者1:生产了6个产品
生产者1:生产了7个产品
消费者1:消费了7个产品
生产者1:生产了7个产品
生产者1:生产了8个产品
消费者1:消费了8个产品
生产者1:生产了8个产品
```



练习

模拟银行取钱的问题

1. 定义一个 Account 类

1 ）该 Account 类封装了账户编号（ String ）和余额（ double ）两个属性

2 ）设置相应属性的 getter 和 setter 方法

3 ）提供无参和有两个参数的构造器

4 ）系统根据账号判断与用户是否匹配，需提供 hashCode() 和 equals() 方法的重写

2. 提供一个取钱的线程类

1 ）提供了 Account 类的 account 属性和 double 类的取款额的属性

2 ）提供带线程名的构造方法

3 ） run() 方法中提供取钱的操作

3. 在主类中创建线程进行测试。考虑线程安全问题。

```
public class Account {
private String accountId;
private double balance;
public Account(){
}
public Account(String accountId,double
balance){
this.accountId = accountId;
this.balance = balance;}
public String getAccountId() {
return accountId;}
public void setAccountId(String
accountId) {
this.accountId = accountId;}
public double getBalance() {
return balance;}
public void setBalance(double balance) {
this.balance = balance;}
public String toString() {
return "Account [accountId=" + accountId
+ ", balance=" + balance + "]";
}
```

```
public int hashCode() {
final int prime = 31;
int result = 1;
result = prime * result
+ ((accountId == null) ? 0 : accountId.hashCode());
long temp;
temp = Double.doubleToLongBits(balance);
result = prime * result + (int) (temp ^ (temp >>>
32)); return result;}
public boolean equals(Object obj) {
if (this == obj) return true;
if (obj == null)
return false;
if (getClass() != obj.getClass())
return false;
Account other = (Account) obj;
if (accountId == null) {
if (other.accountId != null) return false;
} else if (!accountId.equals(other.accountId))
return false;
if (Double.doubleToLongBits(balance) != Double
.doubleToLongBits(other.balance))
return false;
return true;}
```

```
public class WithDrawThread extends Thread{
Account account;
// 要取款的额度
double withDraw;
public WithDrawThread(String name,Account account,double amt){
super(name);
this.account = account;
this.withDraw = amt;}
public void run(){
synchronized (account) {
if (account.getBalance() > withDraw) {
System.out.println(Thread.currentThread().getName()
+ ": 取款成功，取现的金额为 ： " + withDraw);
try {Thread.sleep(50);
} catch (InterruptedException e) {
e.printStackTrace();}
account.setBalance(account.getBalance() - withDraw);
} else {
System.out.println(" 取现额度超过账户余额，取款失败 ");}
System.out.println(" 现在账户的余额为 ： " + account.getBalance());
}}}
```

```
public class TestWithDrawThread {
    public static void main(String[] args) {
        Account account = new Account("1234567",10000);
        Thread t1 = new WithDrawThread(" 小明 ",account,8000);
        Thread t2 = new WithDrawThread(" 小明 's  wife",account,2800);
        t1.start();
        t2.start();
    }
}
```

课堂练习

三个形式一个题
“线程协同”

1、三个线程，一个线程打印 a ，一个线程打印 b ，一个线程打印 c ，三个线程同时执行，要求打印出 20 个连着的 abc 。
abcabcabc…
通过一个锁和一个状态变量来实现（推荐）

2、三个 Java 多线程循环打印递增的数字，每个线程打印 5 个数值，打印周期 1-75.（依次输出 1 到 75 ）

3、创建两个子线程，让其中一个输出 1-100 之间的偶数，另一个输出 1-100 之间的奇数。（依次输出 1 到 100 ）

课堂练习

三个形式一个题
“线程协同”

1、三个线程，一个线程打印 a ，一个线程打印 b ，一个线程打印 c ，三个线程同时执行，要求打印出 20 个连着的 abc 。
abcabcabc…
通过一个锁和一个状态变量来实现（推荐）

2、三个 Java 多线程循环打印递增的数字，每个线程打印 5 个数值，打印周期 1-75.（依次输出 1 到 75 ）

3、创建两个子线程，让其中一个输出 1-100 之间的偶数，另一个输出 1-100 之间的奇数。（依次输出 1 到 100 ）

课堂练习

1、三个线程，一个线程打印 a ，一个线程打印 b ，一个线程打印 c ，三个线程同时执行，要求打印出 20 个连着的 abc 。
abcabcabc…
通过一个锁和一个状态变量来实现（推荐）

先直接说结论：这个题考的是**线程协同（按顺序执行）**，附件里的代码正好给了一个典型解法：

用**一个共享锁 + 一个状态变量** `state`，再配合 `wait()/notifyAll()`，让三个线程"轮班上班"，于是就只能打印出 `abcabcabc…` 这样的序列。 123

1. 问题本质：让线程“按顺序轮流执行”

题目要求：

- 线程A：只打印 `a`
- 线程B：只打印 `b`
- 线程C：只打印 `c`
- 三个线程**同时启动**，最终输出：`abcabcabc...` 共 20 组

也就是说，本质约束是：

> **任何时刻，只允许“轮到的那个线程”打印一个字符，其他线程必须乖乖等着。**

所以我们需要两样东西：

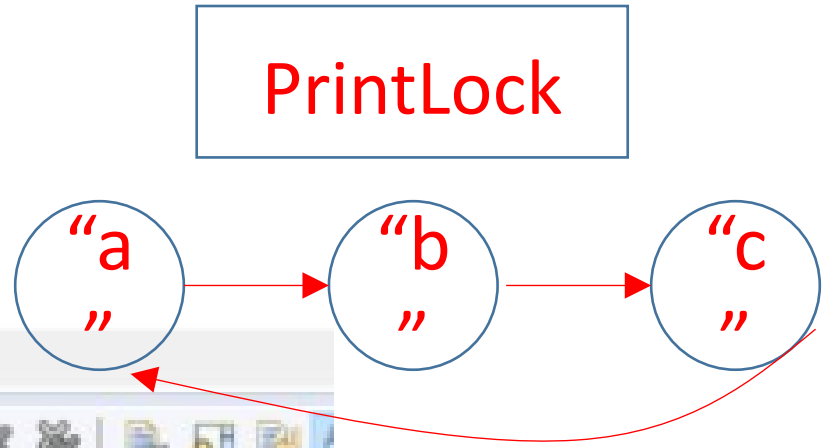
1. **谁能打印** —— 用一个**状态变量**表示，现在轮到谁：
   - 比如 `state = 0` 轮到 `a`
   - `state = 1` 轮到 `b`
   - `state = 2` 轮到 `c`
2. **怎么排队等** —— 用一个**共享锁对象** `printLock`。
   - 所有线程进入临界区时都 `synchronized (printLock)`
   - 不该自己打印的时候就 `printLock.wait()`
   - 打完后修改 `state`，再 `printLock.notifyAll()` 叫醒其他线程来抢锁

“线程协同”

1、三个线程，一个线程打印 a ，一个线程打印 b ，一个线程打印 c ，三个线程同时执行，要求打印出 20 个连着的 abc 。
abcabcabc…
通过一个锁和一个状态变量来实现（推荐）

StateLockPrinter.java

```
package ch006;
//基于一个锁和一个状态变量实现打印10个连续的abc
public class StateLockPrinter {
    //状态变量
    private volatile int state=0;

    // 打印线程
    private class Printer implements Runnable {

    public void test() throws InterruptedException{

    public static void main(String[] args) throws InterruptedException {

        StateLockPrinter print = new StateLockPrinter();
        print.test();
    }
}
```



Console

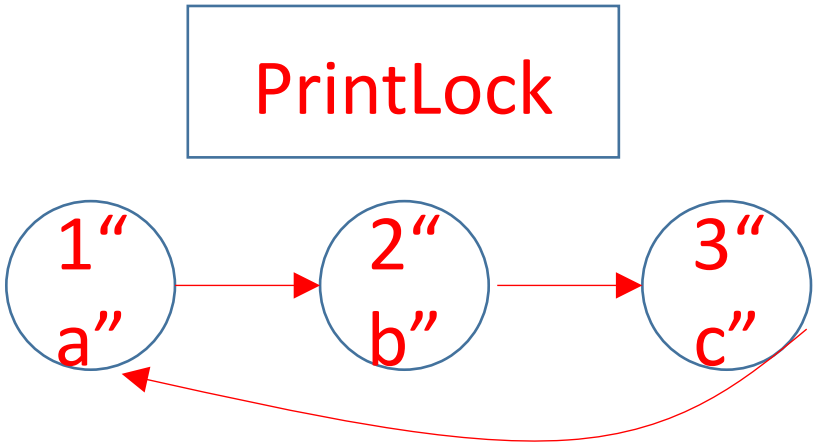
```
<terminated> StateLockPrinter [Java Application] C:\Program Files\Java\jdk1.8.0_144\bin\java
abcabcabcabcabcabcabcabcabcabc
```

“线程协同”

// 打印线程

```
private class Printer implements Runnable {
    //打印次数
    private static final int PRINT_COUNT=10;
    //打印锁
    private final Object printLock;
    //打印标志位 和state变量相关
    private final int printFlag;
    //后继线程的线程的打印标志位，state变量相关
    private final int nextPrintFlag;
    //该线程的打印字符
    private final char printChar;
    public Printer(Object printLock, int printFlag,int nextPrintFlag, char printChar) {

    public void run() {
    }
}
```

PrintLock, 1, 2, a



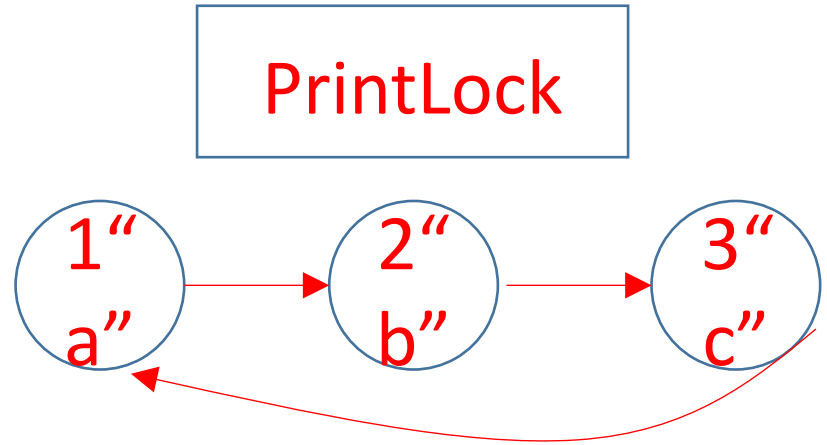
可以把 `Printer` 理解成“**一个线程角色的配置**”：

- A 线程 = 打印 `a` 的角色：`printFlag = 0`，`nextPrintFlag = 1`，`printChar = 'a'`
- B 线程 = 打印 `b` 的角色：`printFlag = 1`，`nextPrintFlag = 2`，`printChar = 'b'`
- C 线程 = 打印 `c` 的角色：`printFlag = 2`，`nextPrintFlag = 0`，`printChar = 'c'`

这样 `state` 就在 `0 -> 1 -> 2 -> 0 -> 1 -> 2 ...` 之间循环，形成 `abcabc...`。

```
private class Printer implements Runnable {
    //打印次数
    private static final int PRINT_COUNT=10;
    //打印锁
    private final Object printLock;
    //打印标志位 和state变量相关
    private final int printFlag;
    //后继线程的线程的打印标志位. state变量相关
    private final int nextPrintFlag;
    //该线程的打印字符
    private final char printChar;
    public Printer(Object printLock, int printFlag,int nextPrintFlag, char printChar) {
        super();
        this.printLock = printLock;
        this.printFlag=printFlag;
        this.nextPrintFlag=nextPrintFlag;
        this.printChar = printChar;
    }

    public void run() {

    }
}
```

“线程协同”

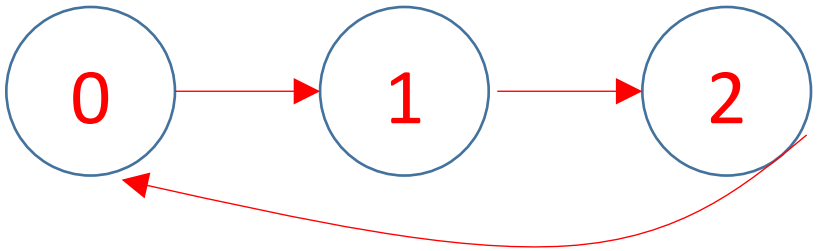


PrintLock, 1, 2, a

run() 方法里的“轮班机制”

```
    @Override
    public void run() {
        //获取打印锁 进入临界区
        synchronized (printLock) {
            //连续打印PRINT_COUNT次
            for(int i=0;i<PRINT_COUNT;i++){
                //循环检验标志位 每次都阻塞然后等待唤醒
                while (state!=printFlag) {
                    try {
                        printLock.wait();
                    } catch (InterruptedException e) {
                        return;
                    }
                }
                //打印字符
                System.out.print(printChar);
                //设置状态变量为下一个线程的标志位
                state=nextPrintFlag;
                //注意要notifyall，不然会死锁，因为notify只通知一个，
                //但是同时等待的是两个,如果唤醒的不是正确那个就会没人唤醒，死锁了
                printLock.notifyAll();
            }
        }
    }

}
```

PrintLock



“线程协同”

```
@Override
public void run() {
    synchronized (printLock) {                    // 进入同一把锁
        for (int i = 0; i < PRINT_COUNT; i++) {
            // 1. 不该我打印，就一直等
            while (state != printFlag) {
                printLock.wait();
            }

            // 2. 轮到我了，打印一个字符
            System.out.print(printChar);

            // 3. 改状态，让下一位同学上
            state = nextPrintFlag;

            // 4. 叫醒所有在这把锁上等待的线程
            printLock.notifyAll();
        }
    }
}
```

关键点：

1. **为什么用 `while` 而不是 `if`？**

   因为 `wait()` 被唤醒后，不保证一定是因为“该你打印了”。

   唤醒后要**重新检查条件**，不满足就继续 wait，这就是典型的**条件等待模式**。

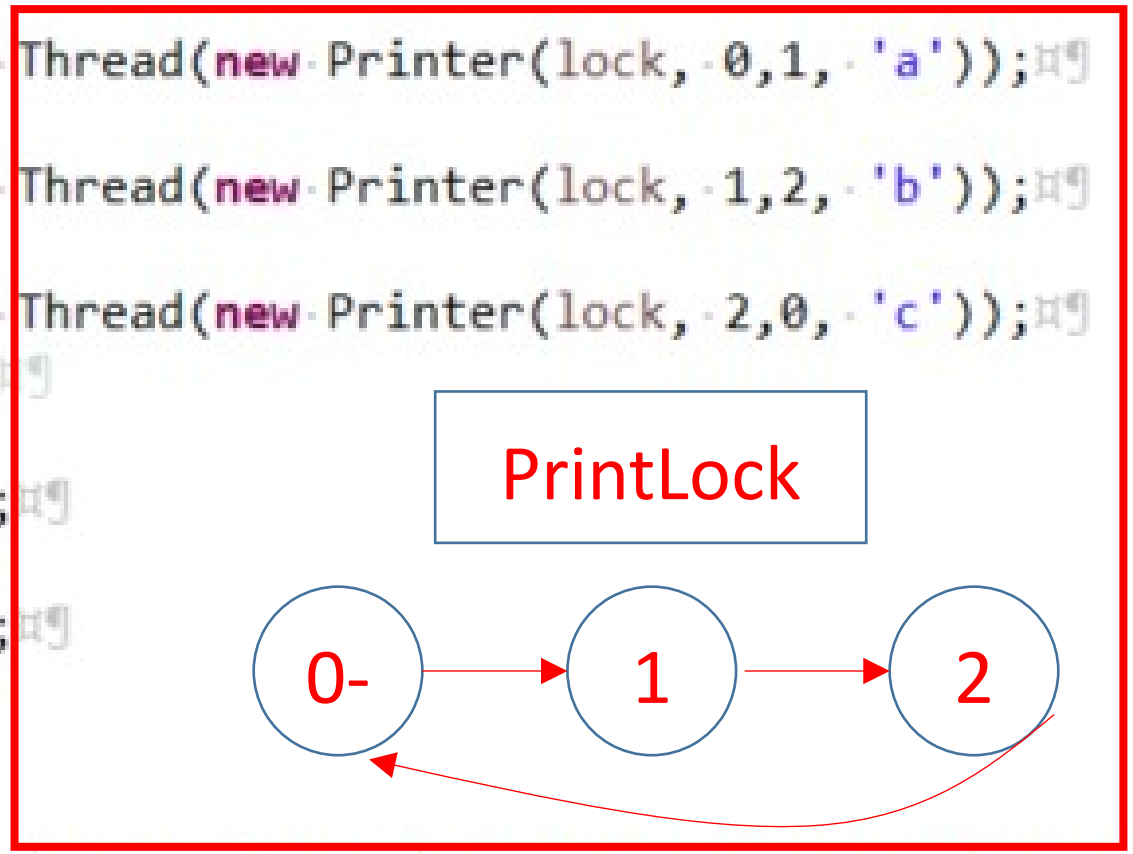
2. **为什么用 `notifyAll()` 而不是 `notify()`？**

   - 如果只 `notify()`，随机唤醒一个线程；
   - 有可能唤醒的是“当前还没轮到的线程”，它发现 `state != printFlag` 又继续 `wait()`；
   - 结果是大家都在 `wait()` 没人能推进，容易死锁。
   - `notifyAll()` 叫醒所有线程，再靠 `while (state != printFlag)` 这道门来筛选谁能打印。

3. **`state` 为啥要 `volatile`？**

   在这个例子里，`state` 的读写都是在 `synchronized(printLock)` 内完成，其实光靠锁就已经保证了可见性，`volatile` 更像是一个“显式提醒”，让你知道它是跨线程共享状态。

```
public void test() throws InterruptedException{
    //锁
    Object lock=new Object();
    //打印A的线程
    Thread threadA=new Thread(new Printer(lock, 0,1, 'a'));
    //打印B的线程
    Thread threadB=new Thread(new Printer(lock, 1,2, 'b'));
    //打印C的线程
    Thread threadC=new Thread(new Printer(lock, 2,0, 'c'));
    //一次启动A B C线程
    threadA.start();
    Thread.sleep(1000);
    threadB.start();
    Thread.sleep(1000);
    threadC.start();
}

public static void main(String[] args) throws InterruptedException {

    StateLockPrinter print = new StateLockPrinter();
    print.test();
}
```



“线程协同”

这里做了几件事：

1. 创建了**同一个** `lock` **对象**，三个 `Printer` 都用它当锁；
2. 给每个 `Printer` 配好“身份”：
   - A：`0 → 1, 'a'`
   - B：`1 → 2, 'b'`
   - C：`2 → 0, 'c'`
3. 依次启动线程（中间 `sleep` 只是为了演示时更容易看清顺序，其实也可以同时启动）。

只要把 `PRINT_COUNT` 改成 `20`，就能打印 20 轮 `abc`。

这道题在面试里想考你什么？

综合一下，其实它同时考：

1. **线程协同思维**：

   会不会把"顺序打印"抽象成：**共享状态 + 条件等待 + 轮换更新。**

2. **wait/notify 用法**：

   - 必须在 `synchronized` 中使用；
   - 等待条件要用 `while` 包起来；
   - 多线程场景下用 `notifyAll()` 更安全。

3. **对锁和状态的抽象能力**：

   用一个 Lock + 一个 int，就能协调三个线程，而不是搞三把锁、三个条件，非常简洁。

你能试着用你自己的话说一说：

printFlag 和 nextPrintFlag 这两个变量，各自承担什么角色？

课堂练习

三个形式一个题
“线程协同”

2 、三个 Java 多线程循环打印递增的数字 ，每个线程打印 5 个数值 ，打印周期 1-75. （ 依次输出 1 到 75 ）

这个题，其实就是把你刚才看的“abc 轮流打印”题，**升级了一下版本**：

从“每次轮流打印 1 个固定字符”，变成“每次轮流打印 5 个**递增的数字**”。核心机制完全一样，还是：

> 一把共享锁 `printLock` + 一个状态变量 `state` + 条件等待 `wait/notifyAll`。 123

1. 题目要求 → 代码里的几个关键常量

题目：

> 3 个 Java 线程循环打印递增数字，每个线程打印 5 个数值，最终依次输出 1–75。

附件里在 `PrintNum` 里，几个关键字段是这样的： 123

- `private static int count = 0;`

  👉 全局计数器，从 0 开始，每打印一个数字就 `++count`，所以全局是 1,2,3,...,75。
- `private final int TOTAL_TIMES = 5;`

  👉 **每个线程轮到"自己上场"的次数是 5 轮。**
- `private final int PER_TIMES = 5;`

  👉 每轮上场时，这个线程要连续打印 **5 个数字**。

再结合"3 个线程"，总打印次数就是：

> `TOTAL_TIMES * 3 * PER_TIMES = 5 * 3 * 5 = 75`（PPT 右上角就写了这个式子）。
> 123

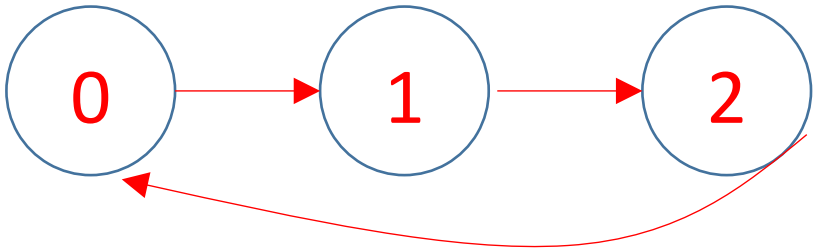
```java
package ch006;

public class PrintNum implements Runnable {

    private static int count=0;
    private final int TOTAL_TIMES=5;
    private final int PER_TIMES=5;

    private static volatile int state=0;
    private Object printLock;
    private int printFlag;
    private int nextPrintFlag;

    PrintNum(Object printLock,int printFlag,int nextPrintFlag){
        this.printLock=printLock;
        this.printFlag=printFlag;
        this.nextPrintFlag=nextPrintFlag;
    }
    @Override
    public void run() {
        synchronized (printLock) {
            for(int i=0;i<TOTAL_TIMES;i++) {
                while(state!=printFlag) {
                    try {
                        printLock.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                for(int j=0;j<PER_TIMES;j++) {
                    System.out.print(++count+" "); //先加后打
                }
                System.out.println();
                state=nextPrintFlag;
                printLock.notifyAll();
            }
        }
    }
}
```

5*3*5=75

2、三个 Java 多线程循环打印递增的数字，每个线程打印 5 个数值，打印周期 1-75.（依次输出 1 到 75 ）

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
26 27 28 29 30
31 32 33 34 35
36 37 38 39 40
41 42 43 44 45
46 47 48 49 50
51 52 53 54 55
56 57 58 59 60
61 62 63 64 65
66 67 68 69 70
71 72 73 74 75
```

PrintLock



2. 状态变量 + 锁：谁轮到谁上

和 “abc 题”一模一样： 123

- `private static volatile int state = 0;`
  - `state = 0` → 轮到线程 A
  - `state = 1` → 轮到线程 B
  - `state = 2` → 轮到线程 C
- `private final Object printLock;`

  👉 所有线程都用这一**把锁**做同步。

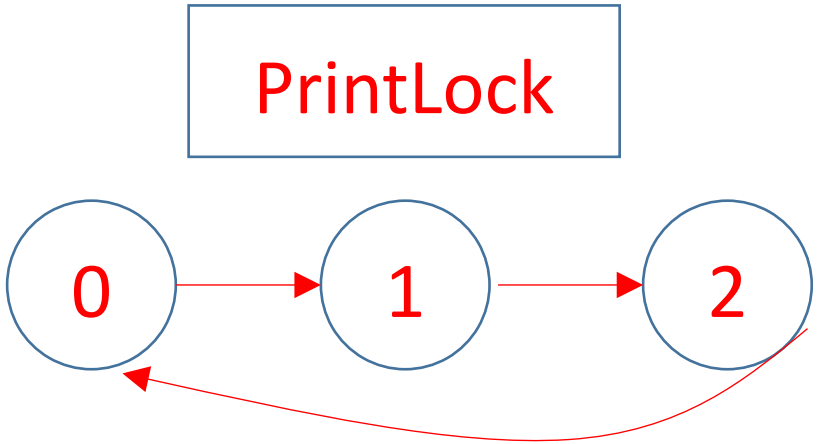
每个 `PrintNum` 保存了：

- `printFlag`：**当前线程被允许打印时，state 应该等于几**（0/1/2）
- `nextPrintFlag`：**自己打印完后，下一位应该把 state 改成几**

所以三个线程就形成一个环：

> 0 → 1 → 2 → 0 → 1 → 2 → …（就像 abc 轮流打印时那张小图）。 123

```java
@Override
public void run() {
    synchronized (printLock) {
        for(int i=0;i<TOTAL_TIMES;i++) {
            while(state!=printFlag) {
                try {
                    printLock.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            for(int j=0;j<PER_TIMES;j++) {
                System.out.print(++count+" "); //先加后打
            }
            System.out.println();
            state=nextPrintFlag;
            printLock.notifyAll();
        }
    }
}
```



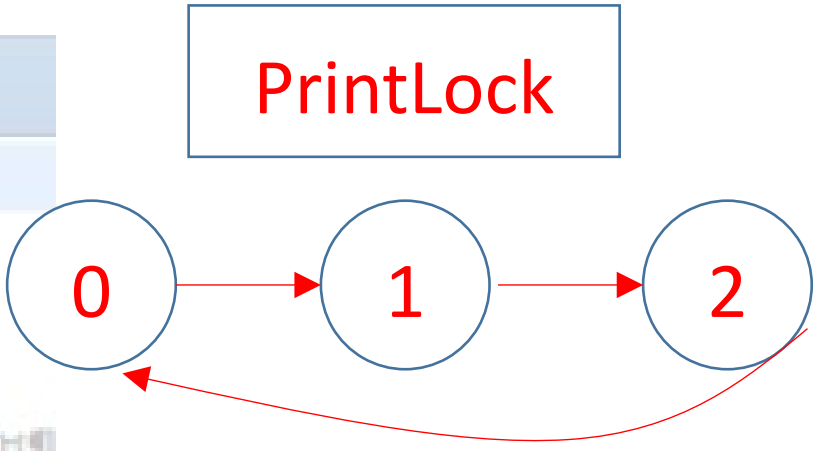

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
26 27 28 29 30
31 32 33 34 35
36 37 38 39 40
41 42 43 44 45
46 47 48 49 50
51 52 53 54 55
56 57 58 59 60
61 62 63 64 65
66 67 68 69 70
71 72 73 74 75
```

逐步理解一下：

1. **外层** `for (i < TOTAL_TIMES)`
   - 控制"**我一共要轮流上场几次**"，题目就是 5 次。
2. `while (state != printFlag) wait()`
   - 不轮到我时，就一直挂起等待；
   - 必须用 `while` 而不是 `if`，因为被唤醒后要重新检查条件。
3. **内层** `for (j < PER_TIMES)`
   - 真正打印数字的部分；
   - 这里每次 `++count`，于是三个线程共享同一个递增序列。
4. **更新** `state` + `notifyAll()`
   - 把接力棒交给下一位线程；
   - `notifyAll()` 保证三个等待线程都被唤醒，再由 `while` 条件来筛选谁能继续跑。

PrintNum.java TestPrintNum.java

```
package ch006;

public class TestPrintNum {

    public static void main(String[] args) {
        Object lock=new Object();
        PrintNum ta=new PrintNum(lock,0,1);
        PrintNum tb=new PrintNum(lock,1,2);
        PrintNum tc=new PrintNum(lock,2,0);
        Thread t1=new Thread(ta);
        Thread t2=new Thread(tb);
        Thread t3=new Thread(tc);
        t1.start();
        t2.start();
        t3.start();
    }
}
```




```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
26 27 28 29 30
31 32 33 34 35
36 37 38 39 40
41 42 43 44 45
46 47 48 49 50
51 52 53 54 55
56 57 58 59 60
61 62 63 64 65
66 67 68 69 70
71 72 73 74 75
```

4. 按题目要求走一轮你看看

假设：

- 初始：`count = 0`，`state = 0`
- 线程 A：`printFlag = 0, nextPrintFlag = 1`
- 线程 B：`1 → 2`
- 线程 C：`2 → 0`

第一大轮（i = 0）：

1. `state = 0`，A 先醒：
   - 打印：1 2 3 4 5
   - `state = 1`，`notifyAll()`
2. B 醒，`state = 1`：
   - 打印：6 7 8 9 10
   - `state = 2`，`notifyAll()`
3. C 醒，`state = 2`：
   - 打印：11 12 13 14 15
   - `state = 0`，`notifyAll()`

这一轮结束，`count = 15`。

第二大轮（i = 1）：

- 又是 A → B → C，分别打印：
  - 16..20
  - 21..25
  - 26..30

5. 这个题在“考”你什么？

跟前一个 abc 题一样，本质是考你：

1. **线程协同模型**：
   - 一把锁 + 一个状态变量，控制“轮流执行”；
   - 通过 `while (state != printFlag) wait()` 实现条件同步。
2. **共享递增状态**：
   - 抽象出一个 `static count`，所有线程都在这一个变量上 `++`；
   - 让“打印顺序”=“线程轮次”×“每次打印个数”。
3. **设计循环结构**：
   - 外层控制轮次；
   - 内层控制每次打印多少个。

你觉得：为什么 count 和 state 要写成 static（共享静态变量），如果改成普通成员变量会发生什么情况？

课堂练习

三个形式一个题
“线程协同”

3、创建两个子线程，让其中一个输出 1-100 之间的偶数，另一个输出 1-100 之间的奇数。（依次输出 1 到 100 ）

这个题本质上还是跟前两个题一样：

用一个共享的计数器 + 一把锁 + 一个状态变量，把两个线程“排队轮流上场”。

只不过这次不是 a/b/c，而是“奇数线程”和“偶数线程”。附件里的 `Printer001.java` 正好就是这个题的参考实现。 123

1. 题目真正想要什么?

> 创建两个子线程:
>
> - “线程1 输出 1–100 之间的奇数”
> - “线程2 输出 1–100 之间的偶数
>
> 最终整体输出是：1,2,3,...,100 依次递增（不能乱序）”

注意两层含义:

1. **每个线程打印的数值集合不同**：一个只负责奇数，一个只负责偶数;
2. **最终输出顺序必须是 1,2,3,...,100**：也就是说两个线程要严格交替打印。

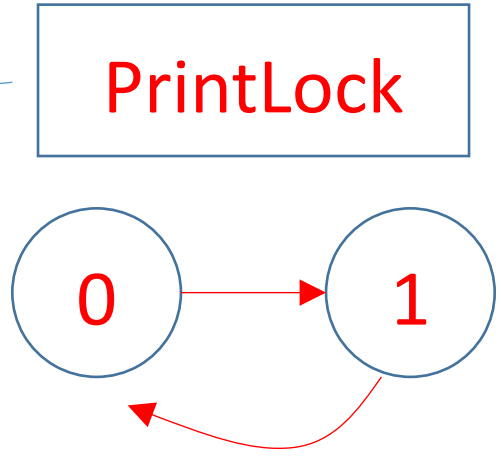
所以问题可以换句话说:

> 让两个线程**轮流执行**:
> 线程 A 打印“下一个数”（第一次是 1），
> 线程 B 打印“下一个数”（第一次是 2），
> 如此往复，一共打印 100 次。

TestPrinter.java | Printer001.java

```java
package ch006;

public class Printer001 implements Runnable {

    private static int count=0;
    private final int TOTAL_TIMES=50;

    private static volatile int state=0;
    private Object printLock;
    private int printFlag;
    private int nextPrintFlag;

    Printer001(Object printLock,int printFlag,int nextPrintFlag){
        this.printLock=printLock;
        this.printFlag=printFlag;
        this.nextPrintFlag=nextPrintFlag;
    }

    @Override
    public void run() {
        synchronized (printLock) {
            for(int i=0;i<TOTAL_TIMES;i++) {
                while(state!=printFlag) {
                    try {
                        printLock.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println(++count+" ");
                state=nextPrintFlag;
                printLock.notifyAll();
            }
        }
    }
}
```

3、创建两个子线程，让其中一个输出 1-100 之间的偶数，另一个输出 1-100 之间的奇数。（依次输出 1 到 100 ）



Console

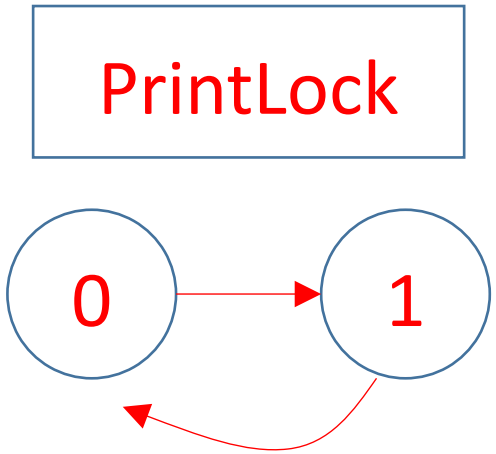
<terminated>

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39

- `count`：全局递增的数字，每打印一次就 `++count`；
- `TOTAL_TIMES = 50`：

  每个线程打印 50 次，一共 2 个线程 → `2 * 50 = 100` 个数字；
- `state`：
  - `0` 表示轮到"奇数线程"（线程 A）；
  - `1` 表示轮到"偶数线程"（线程 B）；
- `printFlag / nextPrintFlag`：
  - A: `printFlag = 0, nextPrintFlag = 1`
  - B: `printFlag = 1, nextPrintFlag = 0`

    → 形成 `0 → 1 → 0 → 1 …` 的轮流执行。



你可以把它类比成"两个人轮流在同一本账本上记数字"：

- 共享账本上的数字是 `count`；
- 谁能拿到账本，由 `state` 决定（0：A，1：B）；
- 拿到账本的人做的事只有两步：
  1. `count = count + 1`
  2. 把新的 `count` 写出来（打印）
  3. 把"下一位轮到谁"的标志改成对方（切换 `state`）。

TestPrinter.java | Printer001.java

```java
package ch006;

public class TestPrinter {

    public static void main(String[] args) {
        Object lock=new Object();
        Printer001 ta=new Printer001(lock,0,1);
        Printer001 tb=new Printer001(lock,1,0);
        Thread t1=new Thread(ta);
        Thread t2=new Thread(tb);
        t1.start();
        t2.start();
    }
}
```

Console

<terminated>

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
```

4. 为什么一个线程只会打印奇数、另一个只会打印偶数?

这是这个题最有意思的点。

初始条件:

- `count = 0`（还没打印任何数）；
- `state = 0`（先轮到线程 A）。

按轮流过程推演几步，你自己感受一下规律:

1. **第一次**:
   - `state = 0` → A 线程被允许执行;
   - A 做 `++count`: `count` 从 0 变成 1，打印 1（奇数）；
   - A 把 `state` 改成 1（轮到 B）。
2. **第二次**:
   - `state = 1` → B 线程被允许执行;
   - B 做 `++count`: `count` 从 1 变成 2，打印 2（偶数）；
   - B 把 `state` 改成 0（轮到 A）。
3. **第三次**:
   - 又轮到 A，`count` 从 2 变成 3（奇数）.....

这样循环下去:

- A 每次执行时，`count` 一定是偶数 → `++count` 后变奇数 → A 只打印奇数;
- B 每次执行时，`count` 一定是奇数 → `++count` 后变偶数 → B 只打印偶数。

并且，因为两个线程严格轮流，一共轮流 100 次，所以输出顺序必然是:

> 1 (A), 2 (B), 3 (A), 4 (B), ..., 99 (A), 100 (B)

刚好满足"奇偶分工 + 顺序递增"。

5. 这道题想考你哪几个点?

结合前三道题，其实整个系列在强化同一种模式：

1. 线程协同的通用套路

- 共享锁 `printLock`；
- 状态变量 `state` 决定"轮到谁"；
- 每个线程用 `printFlag` 判定自己是不是当前"该上场的人"；
- 不轮到自己就 `wait()`，打印完交班用 `notifyAll()`。

2. 如何把业务要求"翻译成状态机"

- abc 题：`state` 在 0/1/2 间循环；
- 1–75 题：多了一个共享计数器、每轮打印多个数；
- 奇偶题：`state` 在 0/1 间切换，结合 `count` 的初值和递增规则，控制"奇偶分工"。

现在假设我们把 count 的初始值改成 1 （ private static int count = 1; ），其他都不变。

你觉得这时： A 线程会打印奇数还是偶数？ B 线程呢？

本章课后作业

1.第 12 讲 ppt 上的示例程序代码编写（与调试）一遍，代码发给

2230652597@qq.com