Java技术

第七章 多线程程序设计

路 强

luqiang@hfut.edu.cn

合肥工业大学计算机与信息学院

本章内容与难点

- 本章学习Java多线程的基本思想、方法。
- 学习重点
  - 线程概念的理解
  - 线程的状态
  - Java程序中创建线程的方法
  - 线程常用方法
  - 线程的同步与互斥

目　录

1 线　程　概　述

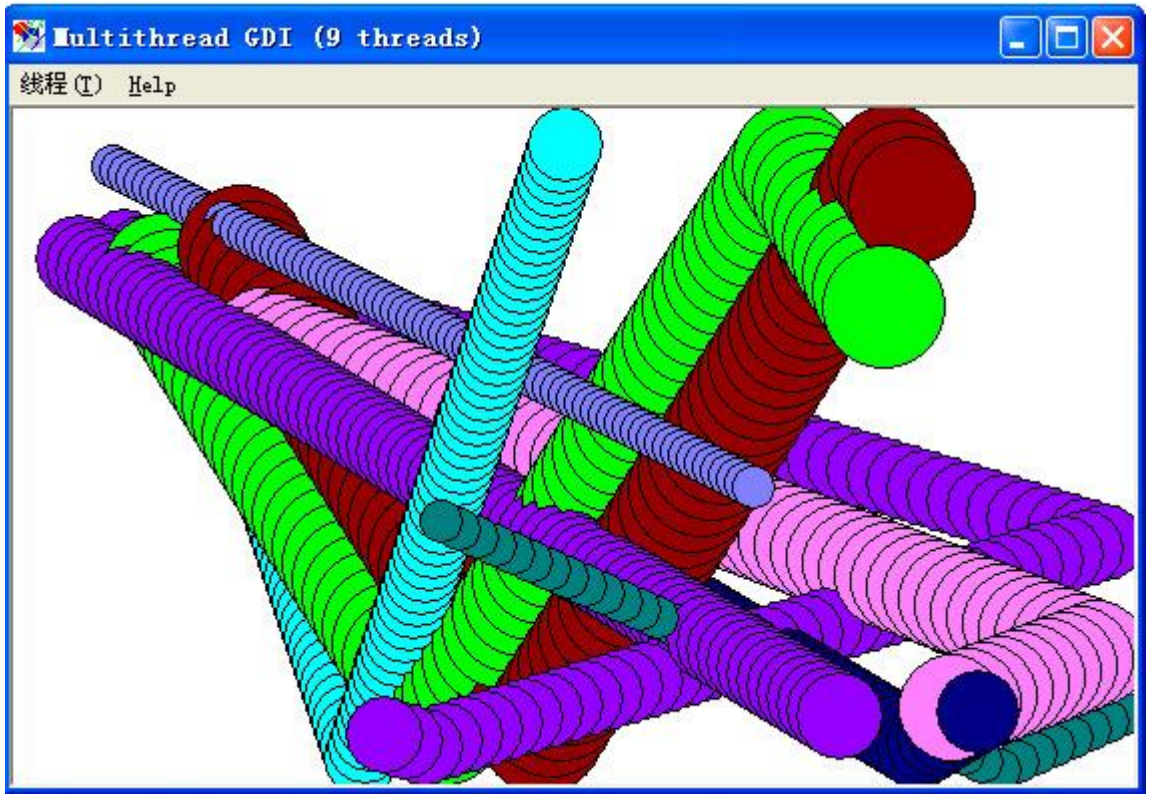
2 Java线程的状态

3 Java线程的创建

4 多线程的互斥和同步

一个例程

- 在窗体中绘制一个“会运动的、碰壁后会反弹的”小球



程序主体框架

```
// 定义带有“球”的窗体类
class MyFrame
        extends Frame {
    …   // 窗体的属性设置
    inner class Ball {
        // 定义球类
        … // 球的属性设置
        … // 球的运动方法
    }
}
```

定义”带有球”的窗体类

```
// 程序主类
class DrawSingleBall {
    public static void main(..)
    {
        // 创建窗口
        new MyFrame
        …
        // 球开始运动
        …
    }
}
```

开始执行

执 行 体

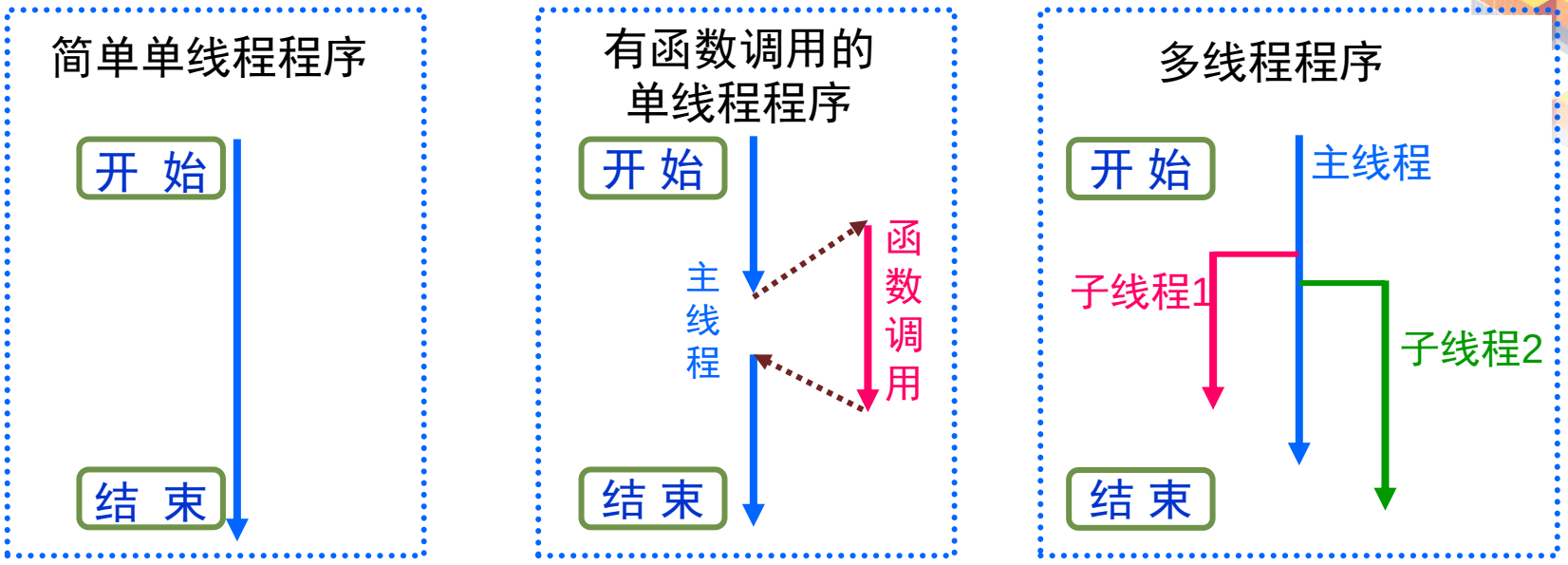
执行完毕

常用软件分析

- 目前大多数大型软件都是基于**多线程**开发的。

- 举　例
  - Windows Media Player
  - 暴风影音
  - Winamp
  - Powerpoint
  - QQ、MSN
  - 游戏（CS、SC）
  - …

程序执行的线索



- 线　程：是一个程序内部的顺序控制流
- 多线程：
  - 在同一应用程序中有多个顺序控制流 “同时” 执行
  - 并发 与 并行

多线程的动机

采用多线程技术的 动 机

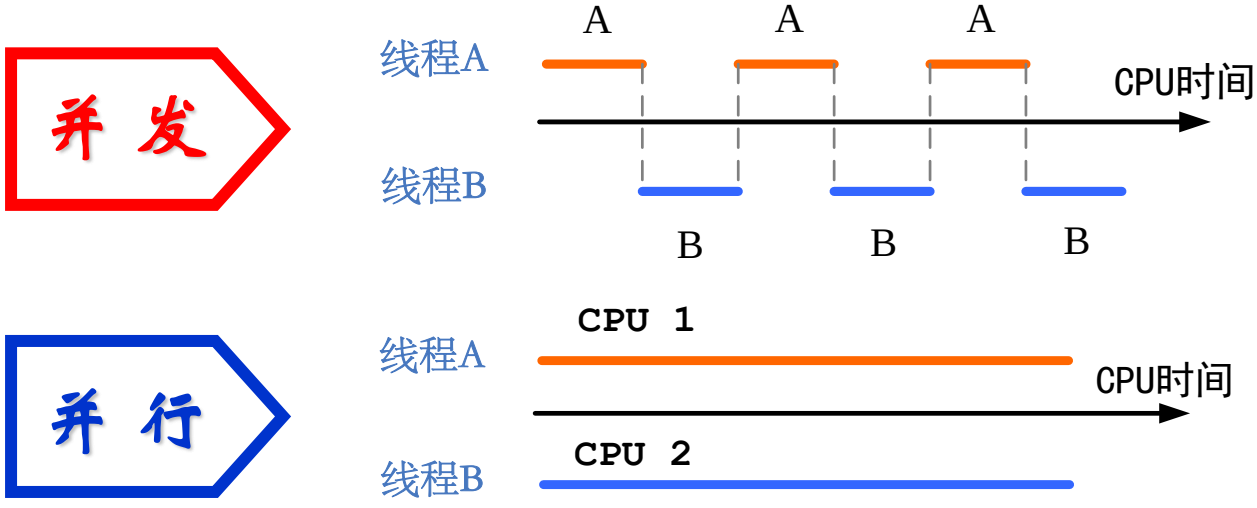
- 问　题：程序的某部分与特定的事件或资源联系在一起，而设计者不想让这种联系阻碍程序其余部分运行
- 解　决：创建一个与事件或资源关联的线程，并且让此线程独立于主线程运行。

- 一个程序在其执行过程中，可以产生多个线程，形成多条执行线索。每条线程，有产生、存在和消亡的过程。
- 程序中多个线程，按照自己的执行路线并发工作，独立完成各自的功能，互不干扰。

Java多线程的执行

单CPU如何实现多线程的并行执行

- 划分极短的CPU时间片
- 各线程轮流占用一个时间片



Java的多线程

Java虚拟机获得的总CPU时间内，在若干个独立的可控制的线程之间切换。

目　录

1 线　程　概　述

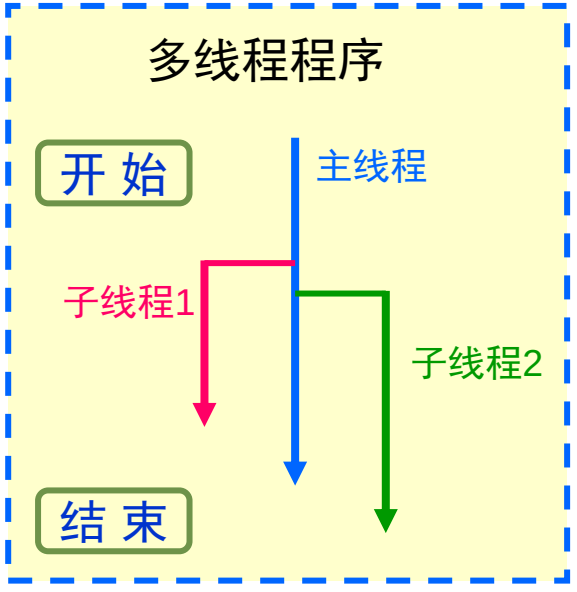
2 Java线程的状态

3 Java线程的创建

4 多线程的互斥和同步

Java程序的执行

- 每个Java程序都有一个缺省的主线程
  - Java应用程序总是从主类的main()方法开始执行
  - main()方法执行过程中可以创建其
- 程序执行情形
  - 如果main()方法中没有创建其它的时，结束的Java应用程序
  - 如果main()方法中创建了其它线程之间轮流切换执行，保证每个线程
  - 直到程序中所有线程都结束，Java应用程序才结束

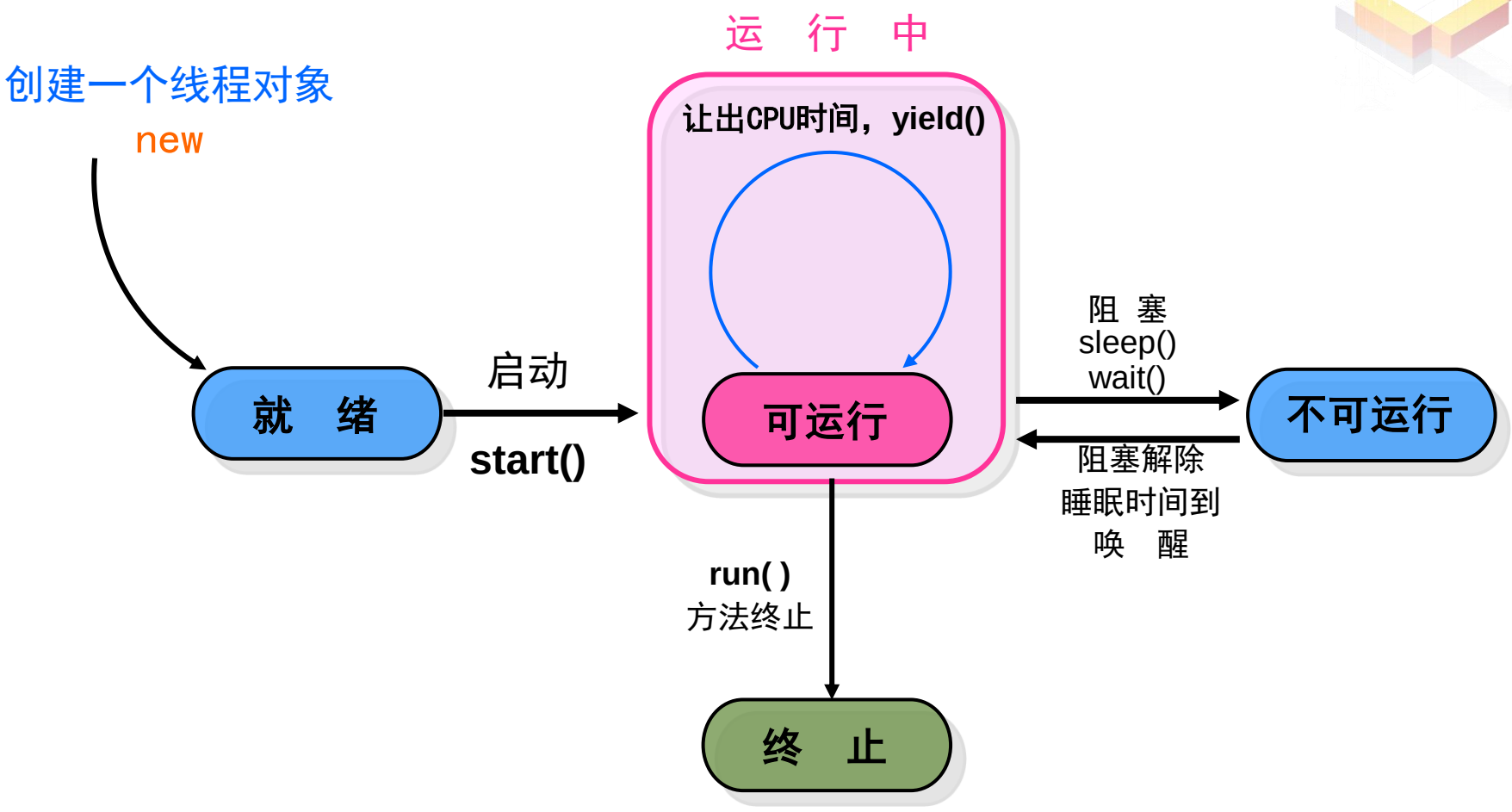


线程的状态

线程可以处于以下四种状态之一：

- **就　绪**：线程对象已建立，但尚未启动，还不能运行
- **可运行**：
  - 线程已经启动，等待获得**CPU时间片**执行
  - 获得**CPU时间片**的线程开始执行
  - 获得**CPU时间片**的线程可以放弃当前执行机会，等待下次执行
- **不可运行**：
  - 不分配给线程CPU时间，直到线程重新进入就绪状态
  - 原　因：阻塞、线程睡眠(sleep)、线程挂起(wait)
- **终　止**：线程结束的正常方式是从run( )方法返回

线程状态的转换

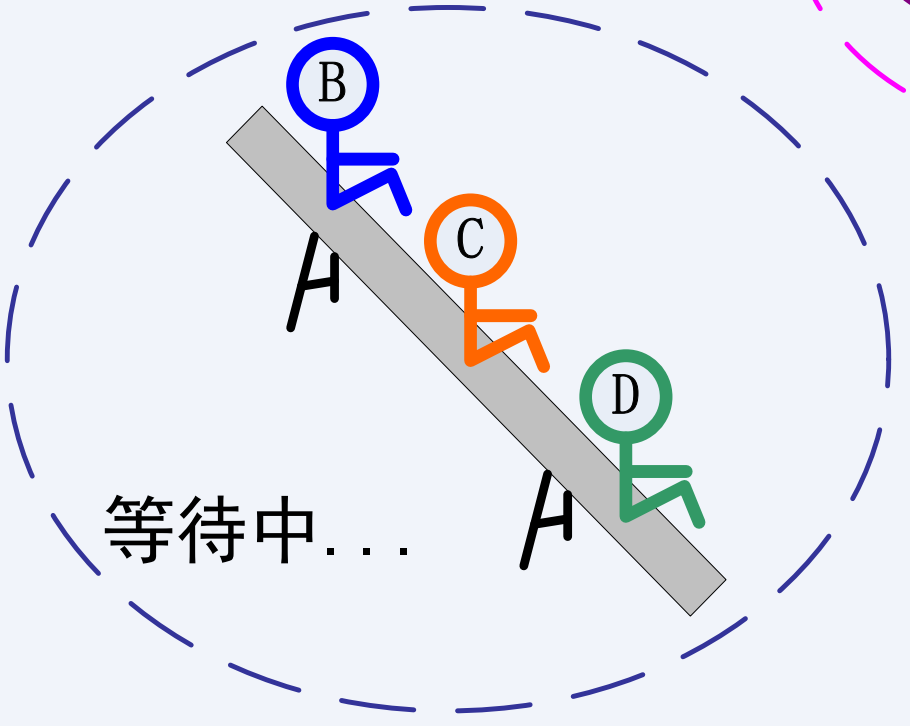

运 行 中
创建一个线程对象
new
让出CPU时间，yield()
就 绪
启动
start()
可运行
阻 塞
sleep()
wait()
不可运行
阻塞解除
睡眠时间到
唤 醒
run( )
方法终止
终 止

多线程竞争CPU

某寝室……

大家凑钱买了一台

电脑，约定轮流使用



如何“轮流”使用?

1. 等某人“良心发现”，主动放弃

2. 约定好每人每次用时(时间片?)

3. “大股东”，优先使用

4. 某人有急事，临时”抢用”

目　录

1 线　程　概　述

2 Java线程的状态

3 Java线程的创建

4 多线程的互斥和同步

创建线程

java.lang.Thread类

- 专门用来创建线程和对线程进行操作的类
- 定义了许多对线程进行操作的方法
- 特殊的 run( )方法

创建线程有两种方法

- 继承Thread类
- 通过定义实现Runnable接口

详细内容见软件包 java.lang.thread 的Thread类

创建线程方法一

- 编写自己的类
  - 该类是java.lang.Thread类的子类
  - 该子类应 重写 Thread 类的 run() 方法

    准备在线程中完成的工作放在run() 方法实现
- 写法之例

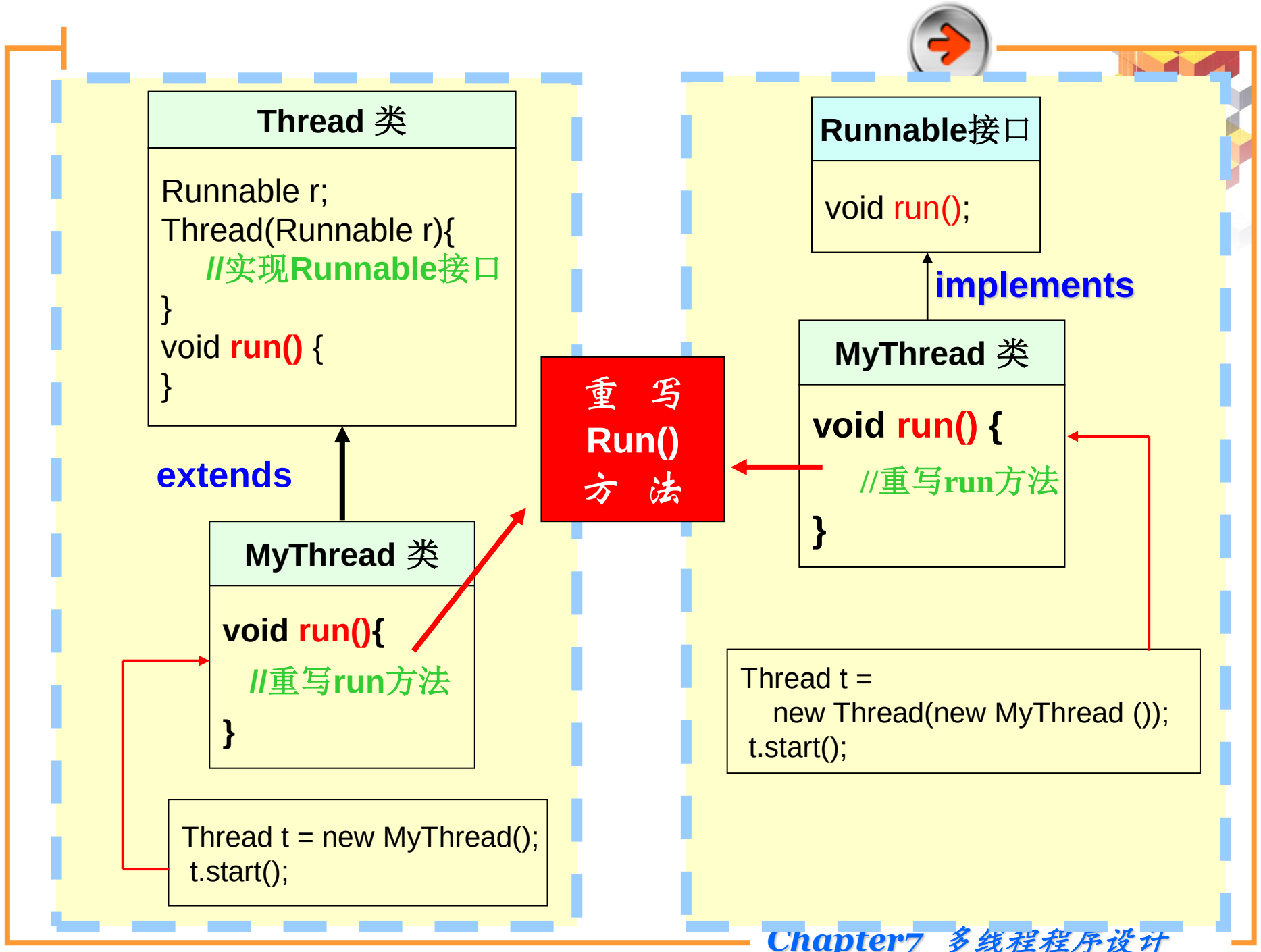
```
1.  //---------------------------------------
2.  class  MyThread  extends  Thread {  // 定义MyThread类
3.      //定义成员属性……
4.      MyThread ( ) {  // 构造方法
5.          ……
6.      }
7.      public void run( ) {
8.          // 完成要完成的工作
9.              ……
10.     }
11. }
12. //---------------------------------------
```

创建线程方法二

- 编写自己的类
  - 实现 Runnable 接口
  - 该子类应重写 Runnable 接口的 run() 方法

    准备在线程中完成的工作放在run() 方法实现
- 写法之例：

```
1.  //----------------------------------------
2.  class MyThread  implements  Runnable {
3.      //定义成员属性
4.          ……
5.      MyThread ( ) { // 构造方法
6.          ……
7.      }
8.      public void run( ) {
9.          // 完成要完成的工作
10.         ……
11.     }
12. }
13. //----------------------------------------
```


Thread 类
Runnable r;
Thread(Runnable r){
//实现Runnable接口
}
void run() {
}
extends
MyThread 类
void run(){
//重写run方法
}
Thread t = new MyThread();
t.start();
重 写
Run()
方 法
Runnable接口
void run();
implements
MyThread 类
void run() {
//重写run方法
}
Thread t =
new Thread(new MyThread ());
t.start();

程序设计.例一

程序功能需求

- 程序完成两个子任务
- 每个子任务各自在命令行打印信息
- 打印内容为：
  - “自己的名称和遍数”
  - “打印结束提示”

解决方法

- 设计自己的线程类
  (派生自Thread类 或 实现Runnable接口)
- 在主线程(main方法)中创建两个线程

Code

通过Thread类实现多线程.例

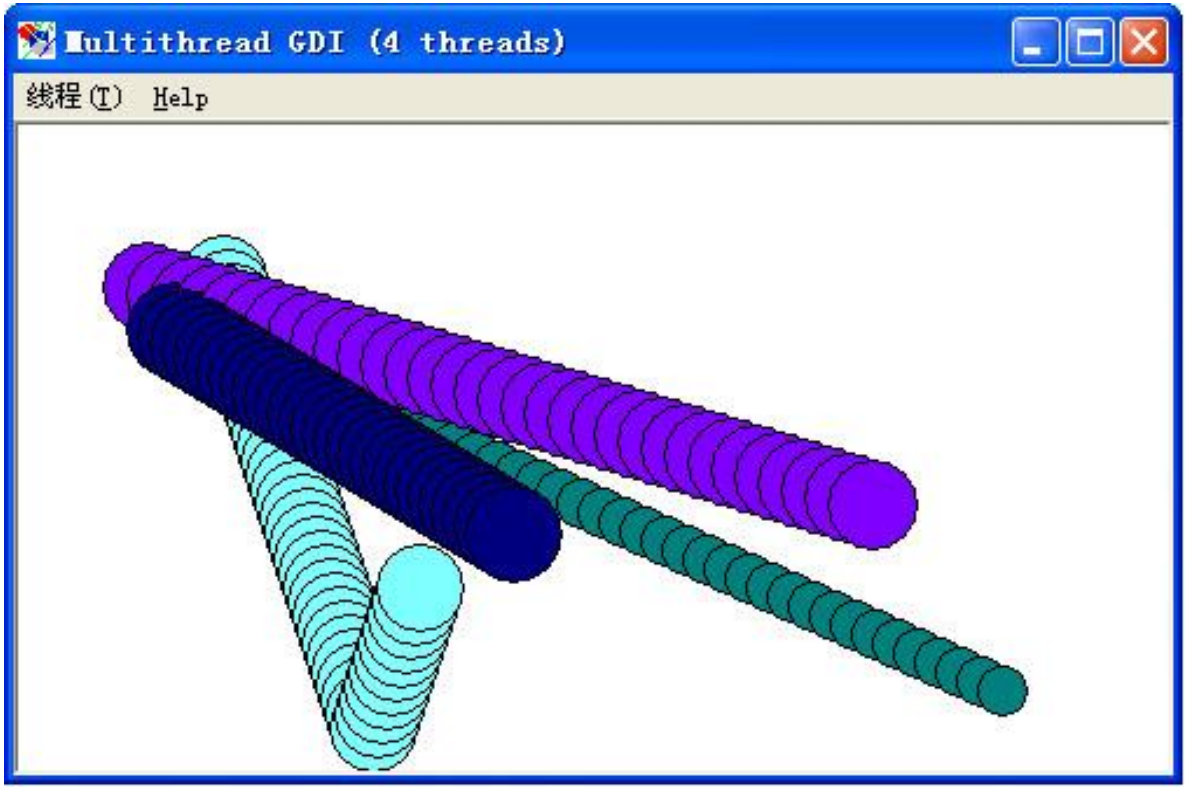

```
1.  //通过继承Thread类实现多线程
2.  class myThread extends Thread {
3.      private String tname;   //线程名
4.      private int count;      //循环次数
5.      //构造方法
6.      myThread (String s, int c){
7.          tname = s;
8.          count = c;
9.      }
```

```
19.  //主类
20. public class Thread_1{
21.     public static void main(String args[]){
22.       //创建线程A
23.         myThread ta =
24.                 new myThread("A", 10);
25.       //创建线程B
26.         myThread tb =
27.                 new myThread("B", 10);
        // 调用start方法
        ta.start();
        tb.start();
    }
```

1. 先创建myThread的两个对象ta和tb
2. 分别调用其start方法启动两个线程

程序设计之二

- “升级”版画球的程序
- 要 求：在窗体中绘制 “多个、独立运动的”小球



课后选做练习

目　录

1 线　程　概　述

2 Java线程的状态

3 Java线程的创建

4 多线程的互斥和同步

多线程间的关系

- 在多线程环境中，可能会有两个甚至更多的线程**试图同时访问**一个有限的资源。必须对这种潜在资源冲突进行预防。
  - **互斥**
  - **同步**
- **解决方法：**
  - 在线程使用一个资源时为其**加锁**即可。
  - 访问资源的第一个线程为其加上锁以后，其他线程便不能再使用那个资源，除非被解锁。

线程并发引起的不确定性

```
1.  class Cbank{   //银行类
2.      private static int s=1000;
3.      public static void sub(int m){  //取款
4.          int temp=s;
5.          temp=temp-m;
6.          try{
7.             Thread.sleep((int)(1000*Math.random()));
8.          }
9.          catch(InterruptedException e){ }
10.         s=temp;
11.         System.out.println("s="+s);
12.     }
13. }
14. class Customer extends Thread{  //储户类
15.     public void run(){
16.         for(int i=1; i<=5; i++)
17.         Cbank.sub(100);
18.     }
19. }
20. class Thread_3  {
21.     public static void main(String args[]){
22.         Customer Customer1=new Customer( );
23.         Customer Customer2=new Customer( );
24.         Customer1.start();
25.         Customer2.start();
26.     }
27. }
```

想要实现，两个储户从一个存款帐户中取钱，保证余额正确。

出现乱子？

多线程互斥的实现

- 对于访问某个关键共享资源的所有方法，都必须把它们设为synchronized，例如：

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

- 当一个线程A使用一个synchronized修饰的方法时，其它线程想使用这个方法时就必须等待，直到线程A 使用完该方法（除非线程A主动让出CPU资源）
- 如果想保护某些资源不被多个线程同时访问，可以强制通过synchronized方法访问那些资源。
  - 调用synchronized方法时，对象就会**被锁定**
  - 当synchronized方法执行完或发生异常时，会自动释放锁
  - 被synchronized保护的数据应该是**私有（private）**

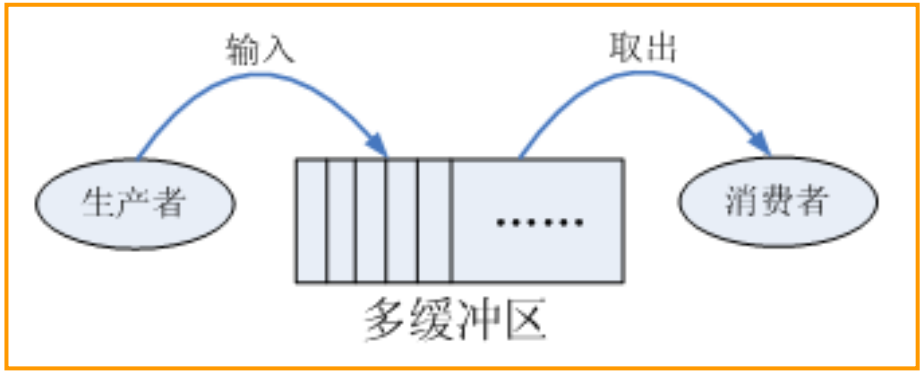
线程互斥之例

```
class Cbank  {
   private static int s=1000;
   public synchronized static void sub(int m){
     int temp=s;
     temp=temp-m;
     try  {
      Thread.sleep((int)(1000*Math.random()));
     }
    catch(InterruptedException e)
       { }
    s=temp;
    System.out.println("s="+s);
   }
}
```

```
class Customer extends Thread   {
    public void run()  {
        for(int i=1; i<=5; i++)
        Cbank.sub(100);
    }
 }
//main class
class Thread_4 {
   public static void main(String args[]){
   Customer Customer1=
                          new Customer( );
   Customer Customer2=
                          new Customer( );
   Customer1.start();
   Customer2.start();
  }
}
```

经典同步 - 生产者消费者问题

- 设生产者和消费者共用n个缓冲区。生产者负责生产产品送到缓冲区，消费者取走缓冲区中的产品。
  - 设每缓冲区的大小只能装入一个产品的信息，
  - 生产者在完成将产品送入缓冲区后，必须等消费者将产品取出后才能放入下一个产品
  - 只有在生产者将产品送入缓冲区后消费者才能读取产品
- 生产者和消费者存在直接制约关系，必须协调同步。



线程的阻塞状态

- 一个线程进入阻塞状态(暂停执行)，可能有如下原因：
  1. 通过调用sleep(milliseconds)使线程进入休眠状态，在这种情况下，线程在指定的时间内不会运行。
  2. 通过调用wait( )使线程挂起。直到线程得到了notify( )或notifyAll( )消息，线程才会进入就绪状态。我们将在下一节验证这一点。
  3. 线程在等待某个输入/输出完成。
  4. 线程试图在某个对象上调用其同步控制方法，但是对象锁不可用。

wait()、notify 和notifyAll()方法

- **挂起** 就是让线程暂时让出CPU的使用权限，暂停执行
  - 挂起一个线程需使用wait方法，即让准备挂起的线程调用wait 方法，主动让出CPU的使用权
  - Java 2中已经废止了以前的suspend( )方法
- **通知** 恢复处于挂起状态线程的运行
  - 其它线程在占有CPU资源期间，让挂起的线程的目标对象执行notify()方法，使得挂起的线程继续执行
  - 如果线程没有目标对象，为了恢复该线程，其它线程在占有CPU资源期间，让挂起的线程调用notifyAll()方法，使挂起的线程继续执行。

生产者消费者问题之例

- ProducerConsumer
  - ProducerConsumerTest.java
  - Buffer.java

    缓冲区类
  - Producer.java

    生产者类
  - Consumer.java

    消费者类

Code

yield(), sleep(), wait()

- 都是中止当前线程线程的执行
- 区别
  - yield(): “让步”
    线程放弃当前的CPU使用权，但将CPU资源让出来后马上重新参加CPU资源竞争；线程会自动回到执行状态。
  - sleep(): “休眠”
    使线程停止执行一段时间，该时间由你给定的毫秒数决定；
    线程会自动回到执行状态
  - wait()：“挂起”
    线程将释放所有资源，等得到通知后在参加资源竞争；
    启动办法是notify() 和notifyAll()方法

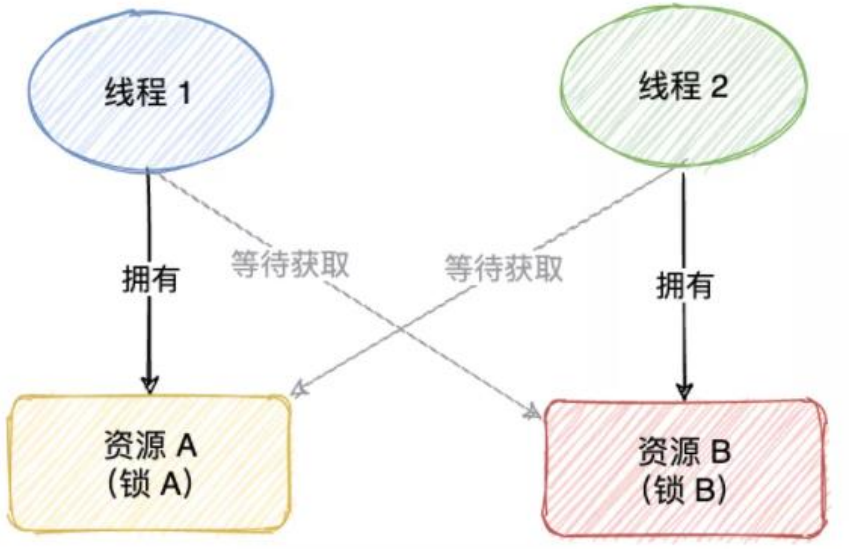
本章总结

- 本章学习Java多线程的基本思想、方法。
  - 线程概念的理解
  - 线程的状态
  - Java程序中创建线程的方法
  - 线程常用方法
  - 线程的同步与互斥

Thank You !

死锁

- 死锁（Dead Lock）指的是两个或两个以上的运算单元（进程、线程），都在等待对方停止执行，以取得系统资源，但是没有一方提前退出，就称为死锁。



例程：DeadLockExample.java
线程 1 和线程 2 都在等待对方释放锁，这样就造成了死锁问。

```
/Library/Java/JavaVirtualMachines
线程 1:获取到锁 A!
线程 2:获取到锁 B!
线程 2:等待获取 A...
线程 1:等待获取 B...
```

多线程互斥的实现

- 对于访问某个关键共享资源的所有方法，都必须把它们设为synchronized，例如：

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

- 当一个线程A使用一个synchronized修饰的方法时，其它线程想使用这个方法时就必须等待，直到线程A 使用完该方法（除非线程A主动让出CPU资源）
- 如果想保护某些资源不被多个线程同时访问，可以强制通过synchronized方法访问那些资源。
  - 调用synchronized方法时，对象就会**被锁定**
  - 当synchronized方法执行完或发生异常时，会自动释放锁
  - 被synchronized保护的数据应该是**私有（private）**