内容提要

线程的概念：多任务的实现

- 多进程
  - 一个独立程序的每一次运行称为一个进程，例如
    - 用字处理软件编辑文稿时，同时打开mp3播放程序听音乐，这两个独立的程序在同时运行，称为两个进程
  - 进程要占用相当一部分处理器时间和内存资源
  - 进程具有独立的内存空间
    - 通信很不方便，编程模型比较复杂

线程的概念：多任务的实现

- 多线程
  - 一个程序中多段代码同时并发执行，称为多线程
  - 线程比进程开销小，协作和数据交换容易
  - Java是第一个支持内置线程操作的主流编程语言，多数程序设计语言支持多线程要借助于操作系统“原语(primitives)”

Thread类

- 直接继承了Object类，并实现了Runnable接口。位于java.lang包中
- 封装了线程对象需要的属性和方法
- 继承Thread类——创建多线程的方法之一
  - 从Thread类派生一个子类，并创建子类的对象
  - 子类应该重写Thread类的run方法，写入需要在新线程中执行的语句段。
  - 调用start方法来启动新线程，线程进入就绪状态，获得执行权限后自动执行run方法。

Thread类

【例10-1】在一个新线程中计算阶乘

```
public class FactorialThreadTester {
    public static void main( String [] args ) {
        System.out.println("main thread starts");
        FactorialThread thread
            =new FactorialThread(10);
        thread.start(); //启动新线程，使之进入就绪状态，
                        //获得执行权限后将自动执行run方法
        System.out.println("main thread ends" );
    }
}
```

```
class FactorialThread extends Thread {
    private int num;
    public FactorialThread( int num ) { this.num=num; }
    // 覆盖run方法
    public void run() {
        int i=num;
        int result=1;
        System.out.println("new thread started" );
        while(i>0){
            result=result*i;
            i=i-1;
         }
        System.out.println(num+"！ = "+result);
        System.out.println("new thread ends");
    }
}
```

- 运行结果

  main thread starts

  main thread ends

  new thread started

  10！= 3628800

  new thread ends

- 结果说明
  - **main**线程已经执行完后，新线程才执行完
  - **main**函数调用**thread.start()**方法启动新线程后并不等待其**run()**方法返回就继续运行，**thread.run()**方法在一边独自运行，不影响原来的**main**函数的运行

Thread类

【例10-1'】修改上例，延长主线程。

如果启动新线程后希望主线程多持续一会再结束，可在start语句后加上让当前线程（这里当然是main）休息20毫秒的语句：

```
try { Thread.sleep(20); } catch(Exception e){};
```

- 修改后运行结果

```
main thread starts
new thread started
10！= 3628800
new thread ends
main thread ends
```

- 结果说明
  - 新线程结束后main线程才结束

Thread类：常用方法

名称	说明
Thread()	构造一个新的线程对象，默认名为Thread-n，n是从0开始递增的整数
Thread(Runnable target)	构造一个新的线程对象，以一个实现Runnable接口的类的对象为参数。默认名为Thread-n，n是从0开始递增的整数
Thread(String name)	构造一个新的线程对象，并同时指定线程名
static Thread currentThread()	返回当前正在运行的线程对象
static void yield()	使当前线程对象暂停，允许别的线程开始运行
static void sleep(long millis)	使当前线程暂停运行指定毫秒数，但此线程并不失去已获得的锁旗标。

Thread类：常用方法

名称	说明
void start()	启动线程，JVM将调用此线程的run方法，结果是将“同时”运行两个线程：当前线程和执行run方法的线程。
void run()	Thread的子类应该重写此方法，内容应为该线程应执行的任务。
final void stop()	停止线程运行，释放该线程占用的对象锁旗标。
void interrupt()	中断此线程
final void join()	如果此前启动了线程A，调用join方法将等待线程A死亡才能继续执行当前线程
final void join(long millis)	如果此前启动了线程A，调用join方法将等待指定毫秒数或线程A死亡才能继续执行当前线程

Thread类：常用方法

名称	说明
final void setPriority(int newPriority)	设置线程优先级
final void setDaemon(Boolean on)	设置是否为后台线程，如果当前运行线程均为后台线程则JVM停止运行。这个方法必须在start()方法前使用
final void checkAccess()	判断当前线程是否有权力修改调用此方法的线程
void setName(String name)	更改本线程的名称为指定参数
final String getName()	返回该线程的名称
final boolean isAlive()	测试线程是否处于活动状态，如果线程被启动并且没有死亡则返回true

Thread类

【例10-2】创建3个新线程，每个线程睡眠一段时间（0～6秒），然后结束。

```
public class ThreadSleepTester {
   public static void main(String[] args) {
        //创建并命名每个线程
        TestThread  thread1 = new TestThread( "thread1" );
        TestThread  thread2 = new TestThread( "thread2" );
        TestThread  thread3 = new TestThread( "thread3" );
        System.out.println( "Starting threads" );
        thread1.start(); //启动线程1
        thread2.start(); // 启动线程2
        thread3.start(); // 启动线程3
        System.out.println( "Threads started, main ends\n" );
   }
```

```
class TestThread extends Thread {
   private int sleepTime;
   public TestThread( String name ) {
         super( name );
         sleepTime = ( int ) ( Math.random() * 6000 );
   }
   public void run(){
         System.out.println(this.getName()+
                              " going to sleep for " + sleepTime );
         try {
            Thread.sleep(sleepTime); //线程休眠
          } catch (InterruptedException e){
              e.printStackTrace();
         }
         System.out.println( getName() + " finished" );
   }
}
```

- 某次运行结果

```
Starting threads
thread1 going to sleep for 1087
Threads started, main ends

thread2 going to sleep for 4127
thread3 going to sleep for 3548
thread1 finished
thread3 finished
thread2 finished
```

- 结果说明
  - 由于线程2休眠时间最长，所以最后结束，线程1休眠时间最短，所以最先结束
  - 每次运行，都会产生不同的随机休眠时间，所以结果都不相同

Runnable接口

- Thread类实现了该接口
- 只有一个run()方法
- 更便于多个线程共享资源
- Java不支持多继承，如果已经继承了某个基类，便需要实现Runnable接口来生成多线程
- 以实现runnable的对象为参数建立新的线程
- start方法启动线程后，线程处于就绪状态，获得执行全选后便会运行run()方法

Runnable接口

【例10-3】使用Runnable接口实现例10-1功能。

```
public class FactorialThreadTester2 {
    public static void main(String[] args) {
            System.out.println("main thread starts");
            FactorialThread2  t=new FactorialThread2(10);  //创建实现了Runnable接口的对象
            Thread thread=new Thread(t);  // 创建新线程
            thread.start();  //启动新线程，将自动进入run()方法
            System.out.println("main thread ends " );
    }
}
```

```
class FactorialThread2 implements Runnable{
   private int num;
    public FactorialThread2( int num ) { this.num=num; }
    public void run(){
          int i=num;
       int result=1;
       System.out.println("new thread started" );
       while(i>0){
           result=result*i;
           i=i-1;
        }
       System.out.println(num+"！ = "+result);
       System.out.println("new thread ends");
   }
}
```

Runnable接口

【例10-4】使用Runnable接口实现例10-2功能。

```
public class ThreadSleepTester2 {
   public static void main(String[] args) {
          TestThread2  t1 = new TestThread2();
          TestThread2  t2 = new TestThread2();
          TestThread2  t3 = new TestThread2();
          System.out.println( "Starting threads" );
          new Thread( t1,"thread1" ).start();  //创建、命名、启动线程1
          new Thread( t2,"thread2" ).start(); //创建、命名、启动线程2
          new Thread( t3,"thread3" ).start(); //创建、命名、启动线程3
          System.out.println( "Threads started, main ends\n" );
   }
}
```

```
class TestThread2 implements Runnable{
   private int sleepTime;
   public TestThread2() {
         sleepTime = ( int ) ( Math.random() * 6000 );
   }
   public void run(){
         System.out.println(Thread.currentThread().getName()
                                        + " going to sleep for " + sleepTime );
         try {
              Thread.sleep(sleepTime); //线程休眠
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
         System.out.println( Thread.currentThread().getName()
                                                  + " finished" );
   }
}
```

线程间的数据共享

- 用同一个实现了Runnable接口的类的实例作为参数创建多个线程，因此它们共享同一对象中的数据。

【例10-5】修改例10-4，只用一个Runnable类型的对象为参数创建3个新线程。

```
public class ThreadSleepTester {
    public static void main(String[] args) {
        TestThread  t = new TestThread2();
        System.out.println( "Starting threads" );
        new Thread( t,"thread1" ).start(); //创建、命名、启动线程1
        new Thread( t,"thread2" ).start(); //创建、命名、启动线程2
        new Thread( t,"thread3" ).start(); //创建、命名、启动线程3
        System.out.println( "Threads started, main ends\n" );
    }
```

线程间的数据共享

- 某次运行结果

```
Starting threads
Threads started, main ends

thread2 going to sleep for 3240
thread3 going to sleep for 3240
thread1 going to sleep for 3240
thread3 finished
thread2 finished
thread1 finished
```

当多线程对同一个对象操作时，线程类必须实现Runnable接口，而不能继承Thread类。因为继承Thread类实现多线程时，操作的是不同的对象。

- 结果说明
  - 因为是用一个Runnable类型对象创建的3个新线程，这3个线程就共享了这个对象的私有成员sleepTime，在本次运行中，3个线程都休眠了3240毫秒

线程间的数据共享

【例10-6】用三个线程模拟三个售票口，总共出售200张票

- 用3个线程模仿3个售票口的售票行为
- 这3个线程应该共享200张票的数据

```
public class SellTicketsTester {
    public static void main(String[] args) {
        SellTickets t = new SellTickets(); // 线程目标对象
        // 创建3个线程
        Thread threads[] = {
        new Thread(t, "窗口1"),
        new Thread(t, "窗口2"),
        new Thread(t, "窗口3") };
        // 启动这些线程
        for (Thread thread : threads){ thread.start(); }
    }
}
```

```
public class SellTickets implements Runnable {
    private int tickets = 200;

    public void run() {
        String name = Thread.currentThread().getName();
        while (tickets > 0) {
            for(int i=0;i<10000;i++){}  // 模拟其他处理耗时
            System.out.println(name + "正在出售票号：" +
            tickets);
            tickets--;
        }
    }
}
```

- 某次运行结果最后几行

```
窗口3正在出售票号： 4
窗口3正在出售票号： 3
窗口3正在出售票号： 2
窗口3正在出售票号： 1
窗口1正在出售票号： 23
窗口2正在出售票号： 21
```

- 结果说明
  - 在这个例子中，创建了3个线程，每个线程调用的是同一个SellTickets对象中的run()方法，访问的是同一个对象中的变量（tickets）
  - 如果是通过创建Thread类的子类来模拟售票过程，再创建3个新线程，则每个线程都会有各自的方法和变量，虽然方法是相同的，但变量却是各有200张票，因而结果将会是各卖出200张票，与原意就不符了

- 可能存在的问题
  - 一张票被重复售出。例如，某次运行结果中间几行

```
窗口1正在出售票号：188
窗口1正在出售票号：187
窗口1正在出售票号：186
窗口2正在出售票号：185
窗口2正在出售票号：186
窗口2正在出售票号：184
```

  - 售出不存在的票。例如，某次运行结果最后几行

```
窗口3正在出售票号：4
窗口2正在出售票号：168
窗口2正在出售票号：2
窗口2正在出售票号：1
窗口3正在出售票号：0
窗口1正在出售票号：179
```

```java
public class SellTickets implements Runnable {
    private int tickets = 10;

    public void run() {
        String name = Thread.currentThread().getName();
        while (tickets > 0) {
            for(int i=0;i<10000;i++){} // 模拟其他处理耗时
            System.out.println(name + "正在出售票号：" + tickets);
            tickets--;
        }
```

控制台

<已终止> SellTicketsTester [Java 应用程序] C:\Program Files\Java\jre1.8.0_261\bin\javaw.exe (2023年11月23日 下午3:05:49)

```
窗口1正在出售票号：10
窗口3正在出售票号：10
窗口2正在出售票号：10
窗口1正在出售票号：8
窗口3正在出售票号：6
窗口2正在出售票号：5
窗口1正在出售票号：5
窗口3正在出售票号：3
窗口2正在出售票号：2
窗口1正在出售票号：2
窗口3正在出售票号：1
窗口2正在出售票号：-1
```

多线程的同步控制

- 有时线程之间彼此不独立、需要同步
- 线程同步
  - 互斥：许多线程在同一个共享数据上操作而互不干扰，同一时刻只能有一个线程访问该共享数据。因此有些方法或程序段在同一时刻只能被一个线程执行，称之为监视区
  - 协作：多个线程可以有条件地同时操作共享数据。执行监视区代码的线程在条件满足的情况下可以允许其它线程进入监视区

多线程的同步控制

- synchronized ——线程同步关键字，实现互斥
  - 用于指定需要同步的代码段或方法，也就是监视区
  - 可实现与一个锁旗标的交互。例如：

    synchronized（对象）{需要互斥的代码段}
  - synchronized的功能是：首先判断对象的锁旗标是否在，如果在就获得锁旗标，然后就可以执行紧随其后的代码段；如果对象的锁旗标不在（已被其他线程拿走），就进入等待状态，直到获得锁旗标
  - 当被synchronized限定的代码段执行完，就释放锁旗标
  - synchronized提高了安全性，但降低了性能。

多线程的同步控制

- Java 使用监视区机制
  - 每个对象只有一个“锁旗标”，利用多线程对“锁旗标”的争夺实现线程间的互斥
  - 当线程A获得了一个对象的锁旗标后，线程B必须等待线程A完成规定的操作、并释放出锁旗标后，才能获得该对象的锁旗标，并执行线程B中的操作

多线程的同步控制

【例10-7】修改例10-6，实现线程同步。

```
public class SellTickets implements Runnable {
    private int tickets = 200;

    public void run() {
        String name = Thread.currentThread().getName();
        while (tickets > 0) {
            synchronized(this) {  //申请对象this的锁旗标
                if(tickets>0){
                    for(int i=0;i<10000;i++){}  // 模拟其他处理耗时
                    System.out.println(name + "正在出售票号：" +
                    tickets);
                    tickets--;
                }
            } // 释放对象this的锁旗标
        }
    }
}
```

- 某次运行结果最后几行

  窗口2正在出售票号： 5
  窗口2正在出售票号： 4
  窗口2正在出售票号： 3
  窗口2正在出售票号： 2
  窗口2正在出售票号： 1

- 说明
  - 当线程执行到synchronized的时候，检查传入的实参对象，并申请得到该对象的锁旗标。如果得不到，那么线程就被放到一个与该对象锁旗标相对应的等待线程池中。直到该对象的锁旗标被归还，池中的等待线程才能重新去获得锁旗标，然后继续执行下去
  - 除了可以对指定的代码段进行同步控制之外，还可以定义整个方法在同步控制下执行，只要在方法定义前加上synchronized关键字即可

```java
public class SellTickets implements Runnable {
  private int tickets = 10;
  public void run() {
    String name = Thread.currentThread().getName();
    while (tickets > 0) {
      synchronized(this) { //申请对象this的锁旗标
       if(tickets>0){
         for(int i=0;i<10000;i++){} // 模拟其他处理耗时
          System.out.println(name + "正在出售票号： " + tickets);
          tickets--;
       }
      } // 释放对象this的锁旗标
    }
```

控制台

<已终止> SellTicketsTester [Java 应用程序] C:\Program Files\Java\jre1.8.0_261\bin\javaw.exe (2023年11月23日 下午3:12:20)

```
窗口1正在出售票号： 10
窗口1正在出售票号： 9
窗口1正在出售票号： 8
窗口1正在出售票号： 7
窗口1正在出售票号： 6
窗口1正在出售票号： 5
窗口1正在出售票号： 4
窗口1正在出售票号： 3
窗口1正在出售票号： 2
窗口1正在出售票号： 1
```

多线程的同步控制

【例10-8】用两个线程模拟存票、售票过程

- 假定开始售票处并没有票，一个线程往里存票，另外一个线程则往出卖票
- 新建一个票类对象，让存票和售票线程都访问它。本例采用两个线程共享同一个数据对象来实现对同一份数据的操作
- 控制两个线程同步

```
public class ProducerAndConsumer {
  public static void main(String[] args) {
        Tickets t=new Tickets(200); //票对象，票总数200
        new Consumer(t).start();  //开始卖票线程
        new Producer(t).start();   //开始存票线程
  }
}
```

```
class Tickets {
   int number=0;          //票号
   int size;              //总票数
   boolean available=false; //表示目前是否有票可售
   public Tickets(int size) { //构造函数，传入总票数参数
       this.size=size;
   }
}
class Producer extends Thread{   //存票线程
  Tickets t=null;
  public Producer(Tickets t){     this.t=t;        }
  public void run() {
        while( t.number < t.size){
            synchronized(t){  //申请对象t的锁旗标
                t.number++;
                System.out.println("Producer puts ticket " + t.number);
                t.available=true;
            } //释放对象t的锁旗标
        }
  }
}
```

```
class Consumer extends Thread{  //售票线程
   Tickets t=null;
   int i=0;
   public Consumer(Tickets t){      this.t=t;      }
   public void run(){
         while(i<t.size){
            synchronized(t){  //申请对象t的锁旗标
               if(t.available==true && i<=t.number)
                  System.out.println("Consumer buys ticket " + (++i));
               if(i==t.number) //现有的票号卖完了
                  t.available=false;
            } //释放对象t的锁旗标
         }
   }
}
```

- 某次运行结果中间几行

Producer puts ticket 38
Producer puts ticket 39
Producer puts ticket 40
Consumer buys ticket 1
Consumer buys ticket 2
Consumer buys ticket 3

- 说明：如果没有控制线程同步，可能出现错误
  - 假如售票线程运行到**t.available=false**之前：

    CPU切换到存票线程，将available置为true，直到整个存票线程结束。
  - 再次切换到售票线程：

    执行t.available=false。此时售票号小于存票数，且存票线程已经结束不再能将t.available置为true，则售票线程陷入了死循环
  - 出错测试：去除线程同步，然后在**t.available=false**之前加上**sleep**语句，让售票线程多停留一会

多线程的同步控制

【例10-8'】将互斥方法放在共享的资源类Tickets中

```
class Tickets2 {
    int size;  //票总数
    int number=0;  //存票序号
    int i=0;   //售票序号
    boolean available=false;  //是否有待售的票
    public Tickets2(int size) {  this.size=size;  }
    public synchronized void put() {  //同步方法，实现存票的功能
        System.out.println("Producer puts ticket "+(++number));
        available=true;
    }
    public synchronized void sell() {  //同步方法，实现售票的功能
        if(available==true && i<=number)
            System.out.println("Consumer buys ticket "+(++i));
        if(i==number)   available=false;
    }
}
```

```
class Producer2 extends Thread {
   Tickets2 t=null;
   public Producer2(Tickets2 t) { this.t=t; }
   public void run() {
      //如果存票数小于限定总量，则不断存入票
     while(t.number<t.size)   t.put();
   }
}


class Consumer2 extends Thread {
   Tickets2 t=null;
   public Consumer2(Tickets2 t) { this.t=t; }
   public void run() {
      //如果售票数小于限定总量，则不断售票
     while(t.i<t.size) t.sell();
   }
}
```

```
public class ProducerAndConsumer2 {
   public static void main(String[] args) {
         Tickets2 t=new Tickets2(200); //票对象，票总数200
         new Consumer2(t).start();  //开始卖票线程
         new Producer2(t).start();   //开始存票线程
   }
}
```

- 某次运行结果的前若干行

```
Producer puts ticket 1
…
Producer puts ticket 29
Consumer buys ticket 1
…
Consumer buys ticket 29
Producer puts ticket 30
…
Producer puts ticket 53
Consumer buys ticket 30
Producer puts ticket 54
…
Producer puts ticket 79
Consumer buys ticket 31
…
Consumer buys ticket 79
Producer puts ticket 80
…
Producer puts ticket 118
Consumer buys ticket 80
Producer puts ticket 119
Consumer buys ticket 81
…
Consumer buys ticket 119
Producer puts ticket 120
…
```

线程之间的通信

- 为了更有效地协调不同线程的工作，需要在线程间建立沟通渠道，通过线程间的“对话”来解决线程间的同步问题
- Object 类的一些方法为线程间的通信提供了有效手段
  - wait() 如果当前状态不适合本线程执行，正在执行同步代码（synchronized）的某个线程A调用该方法（在对象x上），该线程暂停执行而进入对象x的等待池，并释放已获得的对象x的锁旗标。线程A要一直等到其他线程在对象x上调用notify或notifyAll方法，才能够在重新获得对象x的锁旗标后继续执行（从wait语句后继续执行）

线程之间的通信

- **notify()** 随机唤醒一个等待的线程，本线程继续执行
  - 线程被唤醒以后，还要等发出唤醒消息者释放监视区，这期间关键数据仍可能被改变
  - 被唤醒的线程开始执行时，一定要判断当前状态是否适合自己运行
- **notifyAll()** 唤醒所有等待的线程，本线程继续执行

线程之间的通信

【例10-9】修改例10-8’，使每存入一张票，就售一张票，售出后，再存入

```
class Tickets3 {
    ……
    public synchronized void put() { //同步方法，实现存票的功能
        if(available)  //如果还有存票待售，则存票线程等待
            try{ this.wait();} catch(Exception e){}
        System.out.println("Producer puts ticket "+(++number));
        available=true;
        this.notify();  //存票后唤醒售票线程开始售票
    }
    public synchronized void sell() { //同步方法，实现售票的功能
        if(!available)  //如果没有存票，则售票线程等待
            try{ this.wait();} catch(Exception e){}
        System.out.println("Consumer buys ticket "+ number);
        available=false;
        this.notify();              //售票后唤醒存票线程开始存票
        if (number==size) number=size+1 ; //在售完最后一张票后，
                        //设置一个结束标志，number>size表示售票结束
    }
}
```

```
class Producer3 extends Thread {
   Tickets3 t=null;
   public Producer3(Tickets3 t) { this.t=t; }
   public void run() {
      //如果存票数小于限定总量，则存入票
     while(t.number<t.size)   t.put();
   }
}


class Consumer3 extends Thread {
   Tickets3 t=null;
   public Consumer3(Tickets3 t) { this.t=t; }
   public void run() {
      //如果售票数小于限定总量，则售票
     while(t.number<t.size) t.sell();
   }
}
```

```
public class ProducerAndConsumer3 {
  public static void main(String[] args) {
        Tickets3 t=new Tickets3(200); //票对象，票总数200
        new Consumer3(t).start();  //开始卖票线程
        new Producer3(t).start();   //开始存票线程
   }
}
```

- 某次运行结果

Producer puts ticket 1
Consumer buys ticket 1
Producer puts ticket 2
Consumer buys ticket 2
Producer puts ticket 3
Consumer buys ticket 3
……
Producer puts ticket 199
Consumer buys ticket 199
Producer puts ticket 200
Consumer buys ticket 200

- 说明：
  - 当Consumer线程售出票后，available值变为false，当Producer线程存入票后，available值变为true
  - 只有available为true时，Consumer线程才能售票，否则就必须等待Producer线程放入新的票后的通知
  - 只有available为false时，Producer线程才能存票，否则必须等待Consumer线程售出票后的通知

后台线程

- 也叫守护线程，通常是为了辅助其它线程而运行的线程
- 后台线程不会妨碍程序终止
- 一个进程中只要还有一个前台线程在运行，这个进程就不会结束；如果一个进程中的所有前台线程都已经结束，那么无论是否还有未结束的后台线程，这个进程都会结束
- “垃圾回收”便是一个后台线程
- 如果对某个线程对象在启动（调用start方法）之前调用了setDaemon(true)方法，这个线程就变成了后台线程

多线程的同步控制

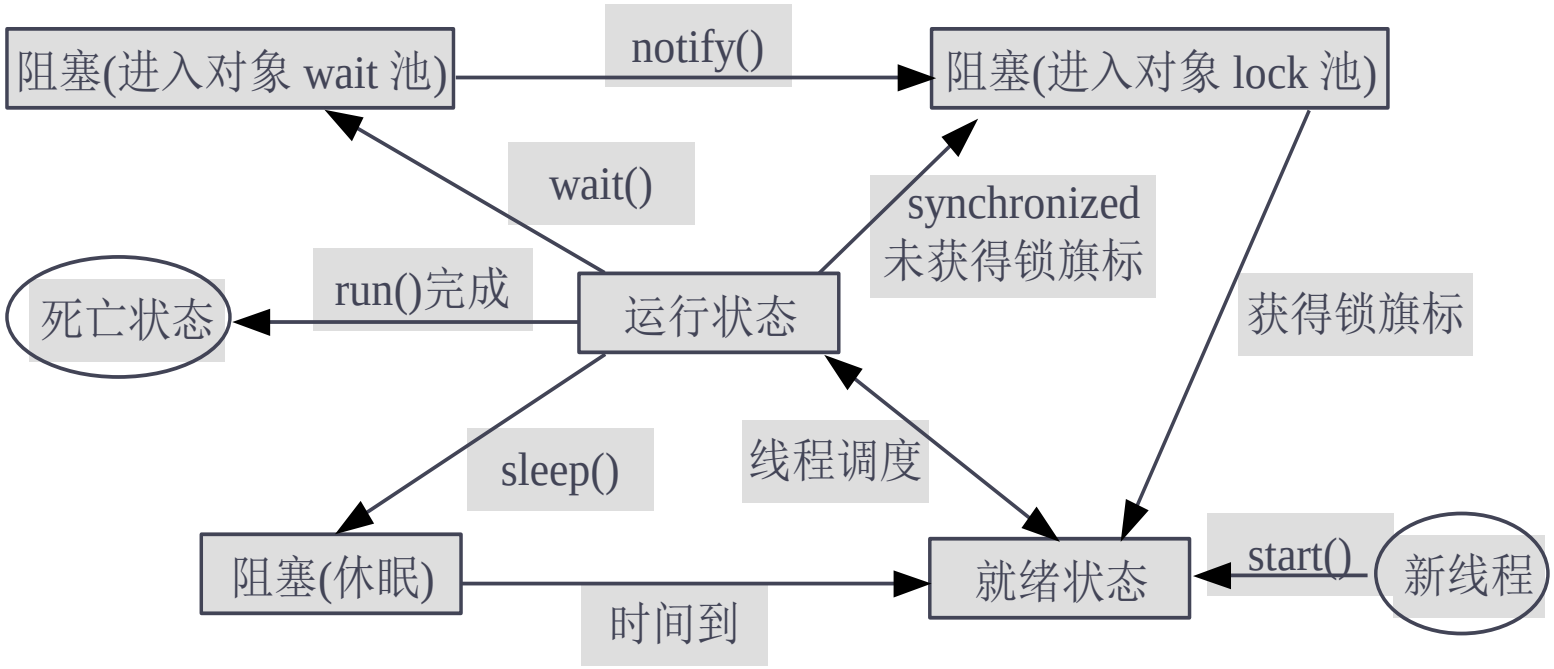
【例10-10】创建一个无限循环的后台线程，验证主线程结束后，程序即结束

```
public class DaemonTest {
    public static void main(String[] args) {
        ThreadTest t=new ThreadTest();
        t.setDaemon(true);  // 线程t设置为后台线程
        t.start();
    }
}
class ThreadTest extends Thread {
    public void run()
    { while(true)   System.out.println((int)(Math.random()*9));  }
}
```

- 说明：
  - 运行程序，则发现整个程序在主线程结束时就随之中止运行了
  - 如果注释掉t.setDaemon(true)语句，则程序永远不会结束

线程的生命周期

- 线程从产生到消亡的过程
- 一个线程在任何时刻都处于某种线程状态（thread state）



线程生命周期状态图

线程的生命周期

- 诞生状态
  - 线程刚刚被创建
- 就绪状态
  - 线程的 start 方法已被执行
  - 线程已准备好运行
- 运行状态
  - 处理机分配给了线程，线程正在运行
- 阻塞状态（Blocked）
  - 在线程发出输入/输出请求且必须等待其返回
  - 遇到用synchronized标记的方法而未获得其监视区暂时不能进入执行时
- 休眠状态（Sleeping）
  - 执行sleep方法而进入休眠
- 死亡状态
  - 线程已完成或退出

线程的生命周期：死锁问题

- 线程在运行过程中，其中某个步骤往往需要满足一些条件才能继续进行下去，如果这个条件不能满足，线程将在这个步骤上出现阻塞
- 线程A可能会陷于对线程B的等待，而线程B同样陷于对线程C的等待，依次类推，整个等待链最后又可能回到线程A。如此一来便陷入一个彼此等待的轮回中，任何线程都动弹不得，此即所谓死锁（deadlock）
- 对于死锁问题，关键不在于出现问题后调试，而是在于预防

小　结

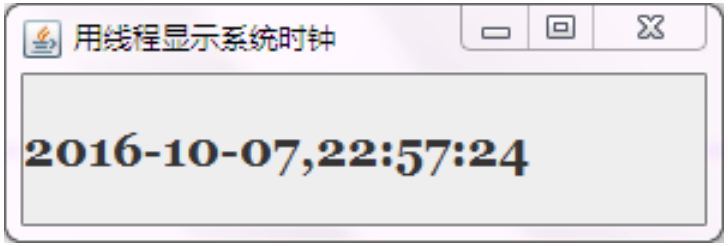
1. 线程的创建方法
   a) 继承Thread类，重写Thread类的run方法，写入需要在新线程中执行的语句段
   b) 直接实现Runnable接口，在接口的run方法中写入需要在新线程中执行的语句段
2. 用同一个实现了Runnable接口的对象作为参数创建多个线程，这些线程将共享该对象中的数据
3. synchronized 关键字用于指定需要同步的代码段或方法，以控制线程间的同步（实现代码互斥）
4. 线程间的通信：wait()、notify()、notifyAll()方法
5. 线程的生命周期：诞生、就绪、阻塞、休眠、死亡

习 题

a) 用线程实现一个如图所示的电子表。 提示：

① 将系统时间显示在JLabel控件上

② 用一个线程循环读取系统时间并显示到这个JLabel上，然后睡眠1秒



b) （选做）用线程实现一个简单动画：一个球在窗体上做直线运动，遇到窗体边界就弹起。 提示：

① 定义一个球类，具有坐标(X, Y)、位移量(offsetX, offsetY)、半径(radius)属性和必要的构造方法、set/get方法

② 重写窗体父类的paint方法，在其中绘制球

③ 用一个线程循环改变球的位置和方向，并让窗体重绘，然后睡眠一段时间，例如0.4秒

注意，应在线程的构造函数中传入必要的参数！

谢谢大家！