

Java 程序设计

第 12 章 Java 多线程机制





导读

主要内容

- Java 中的线程
- Thread 类与线程的创建
- 线程的常用方法
- 线程同步
- 协调同步的线程
- 线程联合
- GUI 线程
- 计时器线程

重点和难点

- 重点：多线程的概念；如何创建多线程
- 难点：理解多线程机制



12.1 进程与线程

12.1.1 操作系统与进程

- **程序**是一段静态的代码，它是应用软件执行的蓝本。
- **进程**是程序的一次动态执行过程，它对应了从程序加载、执行至执行完毕的一个完整从产生、发展至消亡的过程。
- 现代操作系统可以同时管理多个进程，即可以让计算机系统中的多个进程同时执行。

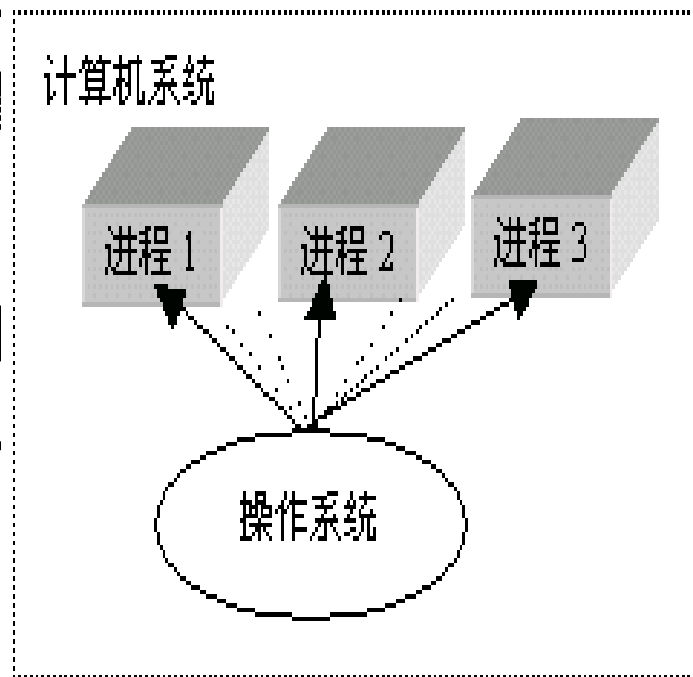


图 12.1 操作系统让进程轮流执行



12.1.2 进程与线程

- 线程是比进程更小的执行单位，一个进程在其执行过程中，可以产生多个线程，形成多条执行线索，每条线索，即每个线程也有它自身的产生、存在和消亡的过程。
- 线程间可以共享进程中的某些内存单元（包括代码与数据），线程的中断与恢复可以更加节省系统的开销

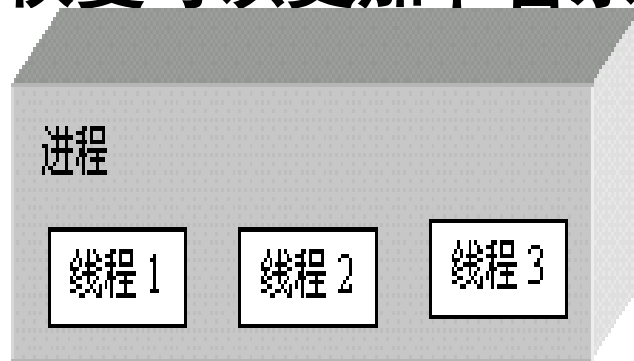


图 12.2 进程中的线程





12.2 Java 中的线程

12.2.1 Java 的多线程机制

- Java 语言的一大特性点就是内置对**多线程**的支持。
- Java 虚拟机快速地把控制从一个线程切换到另一个线程。这些线程将被轮流执行，使得每个线程都有机会使用 CPU 资源



12.2.2 主线程（main 线程）

- 每个 Java 应用程序都有一个缺省的主线程。
- 当 JVM（Java Virtual Machine 虚拟机）加载代码，发现 main 方法之后，就会启动一个线程，这个线程称为“主线程”（main 线程），该线程负责执行 main 方法。
- JVM 一直要等到 Java 应用程序中的所有线程都结束之后，才结束 Java 应用程序。

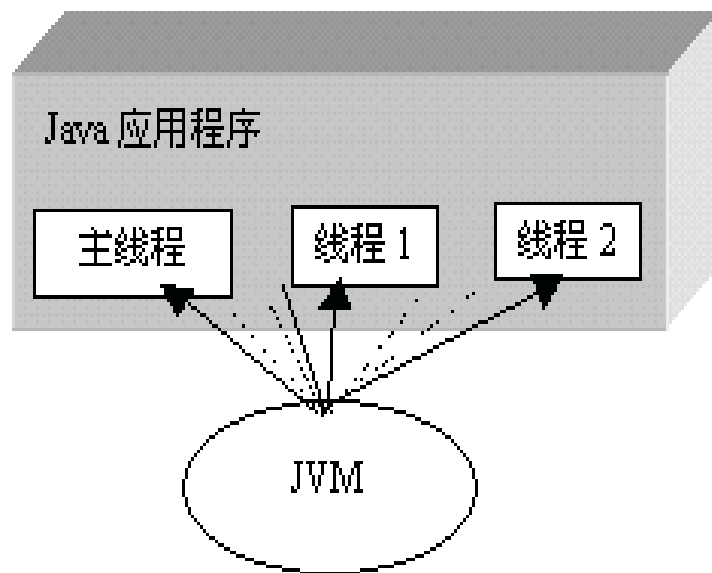


图 12.2 JVM 多线程的执行

12.2.3

线程的状态与生命周期

- 建的线程在它的一个完整的生命周期中通常要经历如下的四种状态：
 1. **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。
 2. **运行**：线程必须调用 **start()** 方法（从父类继承的方法）通知 JVM，这样 JVM 就会知道又有一个新一个线程排队等候切换了。一旦轮到它来享用 CPU 资源时，此线程的就可以脱离创建它的主线程独立开始自己的生命周期了。
 3. **中断**：暂时停止运行
 4. **死亡**：处于死亡状态的线程不具有继续运行的能力。线程释放了实体。



12.2.3

线程的状态与生命周期

3. 中断：有 4 种原因的中断：

- JVM 将 CPU 资源从当前线程**切换**给其他线程，使本线程让出 CPU 的使用权处于中断状态。
- 线程使用 CPU 资源期间，执行了 **sleep(int millisecond)** 方法，使当前线程进入休眠状。
- 线程使用 CPU 资源期间，执行了 **wait()** 方法。
- 线程使用 CPU 资源期间，执行某个操作进入**阻塞**状态。



12.2.3

线程的状态与生命周期

例子1([Example12_1.java](#))通过分析运行结果阐述线程的 4 种状态。例子 1 在主线程中用 Thread 的子类创建了两个线程 ([SpeakElephant.java](#), [SpeakCar.java](#))，这两个线程分别在命令行窗口输出 20 句“大象”和“轿车”；主线程在命令行窗口输出 15 句“主人”。

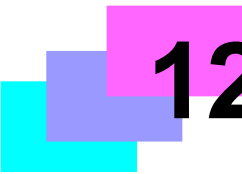
例子1的运行效果如图 12.4。

例子1在不同的计算机运行或在一台计算机反复运行的结果不尽相同，输出结果依赖当前 CPU 资源的使用情况。

```
C:\ch12>java Example12_1
主人1 轿车1 大象1 轿车2 主人2 轿车3 大象2 轿车4 主人3 轿车5 大象3 轿车6
主人4 轿车7 大象4 轿车8 主人5 轿车9 大象5 轿车10 主人6 轿车11 大象6 轿车1
2 轿车13 主人7 轿车14 大象7 轿车15 主人8 轿车16 大象8 轿车17 主人9 轿车18
轿车19 大象9 轿车20 主人10 大象10 主人11 主人12 主人13 主人14 主人15 大
象11 大象12 大象13 大象14 大象15 大象16 大象17 大象18 大象19 大象20
```

图 12.4 轮流执行线程





12.2.3

线程的状态与生命周期

```
for (int i=1; i <= 10; i++) {  
    System.out.println(" 大象 " + i + " ");  
}
```



12.2.4 线程调度与优先级

- 处于就绪状态的线程首先进入就绪队列排队等候 CPU 资源，同一时刻在就绪队列中的线程可能有多个。Java 虚拟机（JVM）中的线程调度器负责管理线程，调度器把线程的**优先级分为 10 个级别**，分别用 Thread 类中的类常量表示。
- Java 调度器的任务是使**高优先级**的线程能始终运行，一旦时间片有空闲，则使具有同等优先级的线程以**轮流的方式**顺序使用时间片。





12.3 Thread 类与线程的创建

12.3.1 使用 Thread 的子类

- 在 Java 语言中，用 **Thread** 类或子类创建线程对象。
- 在编写 Thread 类的子类时，需要**重写父类的 run()** 方法，其目的是规定线程的具体操作，否则线程就什么也不做，因为父类的 run() 方法中没有任何操作语句。





12.3.2 使用 Thread 类

- 创建线程的另一个途径就是用 Thread 类直接创建线程对象。使用 Thread 创建线程通常使用的构造方法是：

Thread(Runnable target)

该构造方法中的参数是一个 **Runnable 类型**的接口。

- 在创建线程对象时必须向构造方法的参数传递一个实现 Runnable 接口类的实例，该实例对象称作所创**线程的目标对象**，当线程调用 start() 方法后，一旦轮到它来享用 CPU 资源，目标对象就会自动调用接口中的 run() 方法（接口回调）。
- **例子2** ([Example12_2.java](#), [ElephantTarget.java](#), [CarTarget.java](#)) 和前面的例子 1 不同，不使用 Thread 类的子类创建线程，而是使用 Thread 类创建 speakElephant 和 speakCar 线程，请读者注意比较例子 1 和例子 2 的细微差别。



12.3.2 使用 Thread 类例题

- 线程间可以**共享相同的内存单元**（包括代码与数据），并利用这些共享单元实现数据交换、实时通信与必要的同步操作
- 例子 3([Example12_3.java](#), [House.java](#)) 中使用 Thread 类创建两个模拟猫和狗的线程，**猫和狗共享房屋中的一桶水**，即**房屋是线程的目标对象**，房屋中的一桶水被猫和狗共享。猫和狗轮流喝水（狗喝的多，猫喝的少），当水被喝尽时，猫和狗进入死亡状态。猫或狗在轮流喝水的过程中，主动休息片刻（让 Thread 类调用 `sleep(int n)` 进入中断状态），而不是等到被强制中断喝水





12.3.2

使用 Thread 类例题

```
public class House implements Runnable{
    int waterAmount;
    public void setWaterAmount(int w) {
        this.waterAmount = w; }
    public void run() {
        while (true) {
            String name = Thread.currentThread().getName();
            if (name.equals(" 狗 ")) {
                System.out.println(name + " 喝水 "); waterAmount -= 2; }
            else if (name.equals(" 猫 ")) {
                System.out.println(name + " 喝水 "); waterAmount -= 1; }
            System.out.println(" 剩 " + waterAmount);
            try {
                Thread.sleep(2000); }
            catch (InterruptedException e) { }
            if (waterAmount <= 0)
                return; } } }
```



12.3.3

目标对象与线程的关系

□ 从对象和对象之间的关系角度上看，目标对象和线程的关系有以下两种情景。

1. 目标对象和线程完全解耦

- ◆ 目标对象没有组合线程对象，目标对象经常需要通过获得**线程的名字**（因为无法获得线程对象的引用）以便确定是哪个线程正在占用 CPU 资源，即被 JVM 正在执行的线程。

2. 目标对象组合线程（弱耦合）

- ◆ 目标对象可以组合线程。目标对象类组合线程对象时，目标对象可以通过获得**线程对象**的引用。

例子 4 中 ([Example12_4.java](#), [House.java](#))，线程 cat 和 dog 在 House 中，请注意例子 4 与例子 3 的区别。



12.3.4 关于 run() 方法启动的次数

- 对于具有相同目标对象的线程，当其中一个线程享用 CPU 资源时，目标对象**自动调用接口中的 run 方法**，这时，run 方法中的局部变量被分配内存空间，当轮到另一个线程享用 CPU 资源时，目标对象会再次调用接口中的 run 方法，那么，run() 方法中的局部变量会再次分配内存空间。也就是说 run() 方法已经启动运行了两次，分别运行在不同的线程中，即运行在不同的时间片内





12.4 线程的常用方法

1. **start()** : 线程调用该方法将启动线程, 使之从新建状态进入就绪队列排队, 一旦轮到它来享用 CPU 资源时, 就可以脱离创建它的线程独立开始自己的生命周期了。
2. **run()** : Thread 类的 run() 方法与 Runnable 接口中的 run() 方法的功能和作用相同, 都用来定义线程对象被调度之后所执行的操作, 都是系统自动调用而用户程序不得引用的方法。
3. **sleep(int millisecond)** : 优先级高的线程可以在它的 run() 方法中调用 sleep 方法来使自己放弃 CPU 资源, 休眠一段时间





12.4 线程的常用方法

- 4 . **isAlive()** : 线程处于“新建”状态时, 线程调用 **isAlive()** 方法返回 **false** 。在线程的 **run()** 方法结束之前, 即没有进入死亡状态之前, 线程调用 **isAlive()** 方法返回 **true** 。
- 5 . **currentThread()** : 该方法是 **Thread** 类中的类方法, 可以用类名调用, 该方法返回当前正在使用 **CPU** 资源的线程。
- 6 . **interrupt()** : 一个占有 **CPU** 资源的线程可以让休眠的线程调用 **interrupt()** 方法“吵醒”自己, 即导致休眠的线程发生 **InterruptedException** 异常, 从而结束休眠, 重新排队等待 **CPU** 资源。



12.4 线程的常用方法

例子 5([Example12_5.java](#), [Home.java](#)) 中一个线程每隔 1 秒钟在命令行窗口输出本地机器的时间，在 3 秒钟后，该线程又被分配了实体，新实体又开始运行。因为垃圾实体仍然在工作，因此，在命令行每秒钟能看见两行同样的本地机器时间。运行效果如图 12.7。

```
C:\ch12>java Example12_3
11:35:23
11:35:24
11:35:25
11:35:26
11:35:26
11:35:27
11:35:27
11:35:28
11:35:28
```

图 12.7 分配了 2 次实体的线程





12.4 线程的常用方法

```
public class Home implements Runnable {  
    int time = 0;  
    SimpleDateFormat m = new SimpleDateFormat("hh:mm:ss");  
    Date date;  
    public void run() {  
        while (true) {  
            date = new Date();  
            System.out.println(m.format(date));  
            time++;  
            try { Thread.sleep(1000); }  
            catch (InterruptedException e) {}  
            if (time == 3){  
                Thread th = Thread.currentThread();  
                th = new Thread(this); th.start(); } } } }
```



12.4 线程的常用方法

例子 6([Example12_6.java](#), [ClassRoom.java](#)) 中，有两个线程：student 和 teacher，其中 student 准备睡一小时后再开始上课，teacher 在输出 3 句“上课”后，吵醒休眠的线程 student。运行效果如图 12.8。

```
张三正在睡觉，不听课  
上课！  
上课！  
上课！  
张三被老师叫醒了  
张三开始听课
```

图 12.8 吵醒休眠的线程



```
public class Classroom implements Runnable {
```

```
    Thread std, tea;
```

```
    Classroom() {
```

```
        std = new Thread(this); tea = new Thread(this);
```

```
        std.setName(" 张三 "); tea.setName(" 李四 "); }
```

```
public void run() {
```

```
    if (Thread.currentThread() == std) {
```

```
        try {
```

```
            System.out.println(std.getName() + " 正在睡觉， 没听课 ");
```

```
            Thread.sleep(1000*60*60);}
```

```
        catch (InterruptedException e) {
```

```
            System.out.println(std.getName() + " 被老师叫醒了 "); }
```

```
            System.out.println(std.getName() + " 开始听课 "); }
```

```
    else if (Thread.currentThread() == tea) {
```

```
        for (int i=1; i<=3; i++) {
```

```
            System.out.println(" 上课 ");
```

```
            try { Thread.sleep(500); } catch (InterruptedException e) { } }
```

```
            std.interrupt(); } } }
```





12.5 线程同步

- 在处理多线程问题时，我们必须注意这样一个问题：当两个或多个线程同时访问同一个变量，并且一个线程需要修改这个变量。我们应对这样的问题作出处理。
- 在处理线程同步时，要做的第一件事就是要把修改数据的方法用关键字 **synchronized** 来修饰。
- 所谓线程同步就是若干个线程都需要使用一个 **synchronized** 修饰的方法。



12.5 线程同步

例子 7([Example12_7.java](#), [Bank.java](#)) 中有两个线程：会计和出纳，他俩共同拥有一个帐本。程序要保证其中一人使用 `saveOrTake(int amount)` 时，另一个人将必须等待，即 `saveOrTake(int amount)` 方法应当是一个 **synchronized 方法**。程序运行效果如图 12.9。

```
会计存入100, 帐上有300万, 休息一会再存  
会计存入100, 帐上有400万, 休息一会再存  
会计存入100, 帐上有500万, 休息一会再存  
出纳取出50, 帐上有450万, 休息一会再取  
出纳取出50, 帐上有400万, 休息一会再取  
出纳取出50, 帐上有350万, 休息一会再取
```

图 12.9 线程同步



```

public class Bank implements Runnable {
    int money = 200;
    public void setMoney(int m) { money = m; }
    public void run() {
        if (Thread.currentThread().getName().equals(" 会计 ")) { saveOrTake(300); }
        else if (Thread.currentThread().getName().equals(" 出 纳 "))
        { saveOrTake(300); } }
    public void saveOrTake(int m) {
        if (Thread.currentThread().getName().equals(" 会计 ")) {
            for (int i=1; i<=3; i++) { money = money + m / 3;
                System.out.println(" 存入 " + m/3 + ", 账上余额为: " + money + ", 休息一会
                再存 ");
                try { Thread.sleep(1000); }
                catch (InterruptedException e) {} } }
            else if (Thread.currentThread().getName().equals(" 出纳 ")) {
                for (int i=1; i<=3; i++) { money = money - m / 3;
                    System.out.println(" 取出 " + m/3 + ", 账上余额为: " + money + ", 休息一会
                    再取 ");
                    try { Thread.sleep(1000); }
                    catch (InterruptedException e) {} } } } }

```





12.6 协调同步的线程

- **wait() 方法**可以中断方法的执行，使本线程等待，暂时让出 CPU 的使用权，并允许其它线程使用这个同步方法
- **notifyAll() 方法**通知所有的由于使用这个同步方法而处于等待的线程结束等待。曾中断的线程就会从刚才的中断处继续执行这个同步方法，并遵循“先中断先继续”的原则。
- **notify() 方法**只是通知处于等待中的线程的某一个结束等待



12.6 协调同步的线程

例子 8([Example12_8.java](#), [TicketHouse.java](#)) 模拟两个人，张飞和李逵买电影票。售票员只有两张五元的钱，电影票 5 元钱一张。张飞拿二十元一张的人民币排在李逵的前面买票，李逵拿一张 5 元的人民币买票。因此张飞必须等待（李逵比张飞先买了票）。程序运行效果如图 12.10。

```
张飞靠边等...  
给李逵入场卷, 李逵的钱正好  
  
张飞继续买票  
给张飞入场卷, 张飞给20, 找赎15元
```

图 12.10 .wait 与 notifyAll+



```

public class TickHouse implements Runnable {
    int five = 2, ten = 0, twenty = 0;
    public void run() {
        if (Thread.currentThread().getName().equals("张飞")) { saleTicket(20); }
        else if (Thread.currentThread().getName().equals("李逵"))
        { saleTicket(5); } }
    private synchronized void saleTicket(int money) {
        if (money == 5) { five += 1;
            System.out.println("给" + Thread.currentThread().getName() + "入场券，" +
                Thread.currentThread().getName() + "的钱正好"); }
        else if (money == 20) {
            while (five < 3) {
                try {
                    System.out.println("\n" + Thread.currentThread().getName() + "靠边等...");
                    wait(); System.out.println("继续买票"); }
                catch (InterruptedException e) {} }
            five -= 3; twenty += 1;
            System.out.println("给" + Thread.currentThread().getName() + "入场券，" +
                Thread.currentThread().getName() + "给钱20，找零15元"); }
        notifyAll(); } }

```





12.7 线程联合

- 一个线程 A 在占有 CPU 资源期间，可以让其它线程调用 `join()` 和本线程联合，如：

B.join();

- 称 A 在运行期间联合了 B。如果线程 A 在占有 CPU 资源期间一旦联合 B 线程，那么 A 线程将立刻中断执行，一直等到它联合的线程 B 执行完毕，A 线程再重新排队等待 CPU 资源，以便恢复执行。如果 A 准备联合的 B 线程已经结束，那么 `B.join()` 不会产生任何效果。





12.7 线程联合

- 例子 9 ([Example12_9.java](#), [ThreadJoin.java](#)) 使用线程联合模拟顾客等待蛋糕师制作蛋糕，程序运行效果如图 12.11。

```
顾客等待蛋糕师制作生日蛋糕  
蛋糕师开始制作生日蛋糕, 请等...  
蛋糕师制作完毕  
顾客买了生日蛋糕 价钱:158
```

图 12.11 线程联合



```
public class TJoin implements Runnable {
```

```
    Thread joinThread;
```

```
    public void setJoinThread(Thread t) { this.joinThread = t; }
```

```
    public void run() {
```

```
        if (Thread.currentThread().getName().equals(" 线程 1")) {
```

```
            for (int i = 1; i <= 5; i++) {
```

```
                System.out.println(" 线程 1_" + i);
```

```
                if (i == 3)
```

```
                    try { this.joinThread.start(); this.joinThread.join(); }
```

```
                    catch (InterruptedException e1) { e1.printStackTrace(); }
```

```
                    try { Thread.sleep(1000); }
```

```
                    catch (InterruptedException e) { e.printStackTrace(); } } }
```

```
            else if (Thread.currentThread() == this.joinThread) {
```

```
                for (int i = 1; i <= 5; i++) {
```

```
                    System.out.println(" 线程 2_" + i);
```

```
                    try { Thread.sleep(1000); }
```

```
                    catch (InterruptedException e) { e.printStackTrace(); } } } }
```





12.8 GUI 线程

- 当 Java 程序包含图形用户界面（GUI）时，Java 虚拟机在运行应用程序时会自动启动更多的线程，其中有两个重要的线程：
 - ◆ AWT-EventQueue 线程：负责处理 GUI 事件
 - ◆ AWT-Window 线程：负责将窗体或组件绘制到桌面





12.8 GUI 线程

- JVM 要保证各个线程都有使用 CPU 资源的机会，比如，程序中发生 GUI 界面事件时，JVM 就会将 CPU 资源切换给 AWT-EventQueue 线程，AWT-EventQueue 线程就会来处理这个事件，比如，你单击了程序中的按钮，触发 ActionEvent 事件，AWT-EventQueue 线程就立刻排队等候执行处理事件的代码



12.8 GUI 线程

例子 10([Example12_10.java](#), [WindowTyped.java](#)) 是训练用户寻找键盘上的字母的快速能力。一个线程 `giveLetter` 负责每隔 3 秒给出一个英文字母, 用户需要在文本框中输入这个英文字母, 按回车确认。当用户按回车键时, 将触发 `ActionEvent` 事件, 那么 JVM 就会中断 `giveLetter` 线程, 把 CUP 的使用权切换给 `WT-EventQueue` 线程, 以便处理 `ActionEvent` 事件。程序运行效果如图 12.12。

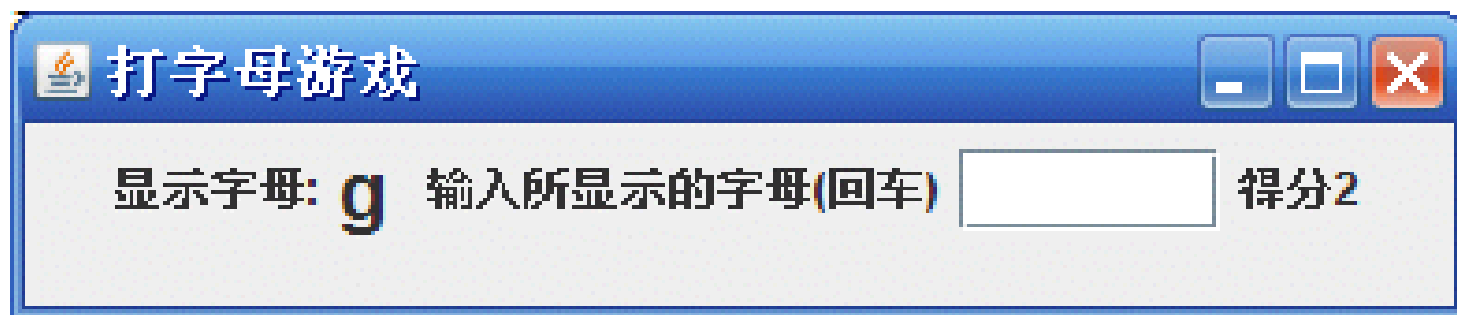


图 12.12 打字母游戏



```
public class WindowTyped extends JFrame implements ActionListener, Runnable {  
    JTextField inputLetter; Thread giveLetter; JLabel showLetter, showScore;  
    int sleepTime, score; Color c;  
    WindowTyped() {  
        setLayout(new FlowLayout());  
        giveLetter = new Thread(this);  
        inputLetter = new JTextField(6);  
        showLetter = new JLabel(" ", JLabel.CENTER);  
        showScore = new JLabel(" 分数 ");  
        showLetter.setFont(new Font("Arial", Font.BOLD, 22));  
        this.add(new JLabel(" 显示字母: "));  
        this.add(showLetter);  
        this.add(new JLabel(" 输入 所显示的字母 ( 回车 )"));  
        this.add(inputLetter);  
        this.add(showScore);  
        this.inputLetter.addActionListener(this);  
        this.setBounds(100,100,400,280);  
        this.setVisible(true);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        giveLetter.start(); }  
}
```



```

public void run() {
    char c = 'a';
    while (true) {
        showLetter.setText("" + c + " "); validate(); c = (char)(c+1);
        if (c > 'z') c = 'a';
        try { Thread.sleep(sleepTime); }
        catch (InterruptedException e) {
            System.out.println(" 发生 InterruptedException"); } } }

```

```

public void actionPerformed(ActionEvent e) {
    String s = showLetter.getText().trim();
    String letter = inputLetter.getText().trim();
    if (s.equals(letter)) {
        score++; showScore.setText(" 得分: " + score);
        inputLetter.setText(null); validate(); giveLetter.interrupt(); } }

```

```

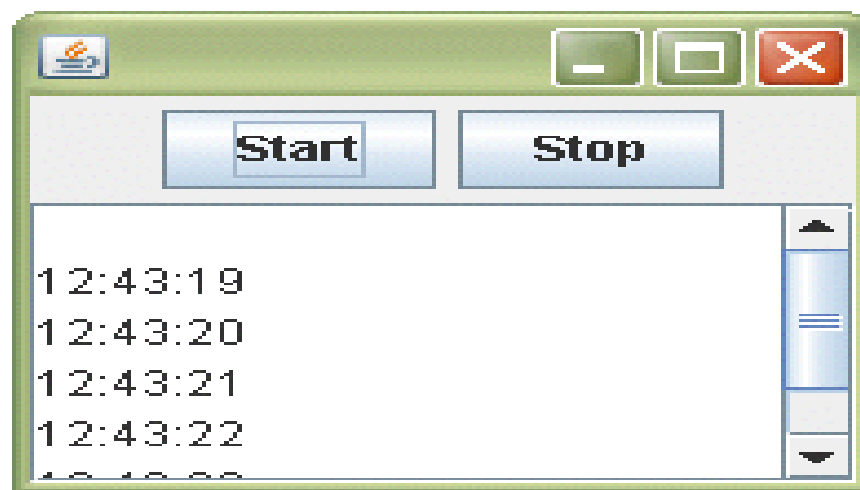
public void setSleepTime(int n) { this.sleepTime = n; } }

```



12.8 GUI 线程 – 例题

例子 11(Example12_11.java , Win.java) 中单击 start 按钮线程开始工作：每隔一秒钟显示一次当前时间；单击 stop 按钮后，线程就结束了生命，释放了实体，即释放线程对象的内存。把一个线程委派给一个组件事件时要格外小心，比如单击一个按钮让线程开始运行，那么当这个线程在执行完 run() 方法之前，客户可能会随时再次单击该按钮，这时就会发生 `IllegalThreadStateException` 异常。程序运行效果如图 12.13。





12.9 计时器线程

- 计时器每隔 a 毫秒“震铃”一次，参数 b 是计时器的监视器。计时器发生的震铃事件是 `ActionEvent` 类型事件。当震铃事件发生时，监视器就会监视到这个事件，监视器就回调 `ActionListener` 接口中的 `actionPerformed(ActionEvent e)` 方法。
 - ◆ 使用 `Timer` 类的方法 `start()` 启动计时器，即启动线程
 - ◆ 使用 `Timer` 类的方法 `stop()` 停止计时器，即挂起线程
 - ◆ 使用 `restart()` 重新启动计时器，即恢复线程



12.9 计时器线程

例子 12(Example12_12.java, WindowTime.java) 中，单击“开始”按钮启动计时器，并将时间显示在文本框中，同时移动文本框在容器中的位置；单击“暂停”按钮暂停计时器；单击“继续”按钮重新启动计时器。程序运行效果如图 12.14 。



图 12.14 计时器线程



12.10 守护线程

- 一个线程调用 `void setDaemon(boolean on)` 方法可以将自己设置成一个守护（Daemon）线程，例如：
`thread.setDaemon(true);`
- 当程序中的所有用户线程都已结束运行时，即使守护线程的 `run` 方法中还有需要执行的语句，**守护线程也立刻结束运行**
- 例子 13(Example12_13.java , Daemon.java) 中有一个守护线程





```
public class Daemon implements Runnable {  
    Thread thA, thB;
```

```
    Daemon() { thA = new Thread(this); thB = new Thread(this); }
```

```
    public void run() {  
        if (Thread.currentThread() == thA) {  
            for (int i = 0; i < 8; i++) {  
                System.out.println("i = " + i);  
                try { Thread.sleep(1000); }  
                catch (InterruptedException e) {} } }  
            else if (Thread.currentThread() == thB) {  
                while (true) {  
                    System.out.println(" 线程 B 是守护线程 ");  
                    try { Thread.sleep(1000); }  
                    catch (InterruptedException e) {} } } } } }
```



12.11 应用举例

- 在电视节目中经常看见主持人提出的问题，并要求考试者在限定时间内回答问题。这里由程序提出问题，用户回答问题。问题保存在 **test.txt** 中，**test.txt** 的格式如下：
 - (1) 每个问题提供 **A、B、C、D** 四个选择（单项选择）。
 - (2) 两个问题之间是用减号（-）尾加前一问题的答案分隔（例如：----D----）。
- 例题 14(Example12_14.java , StandardExamInTime.java) 运行效果如图 12.15 。

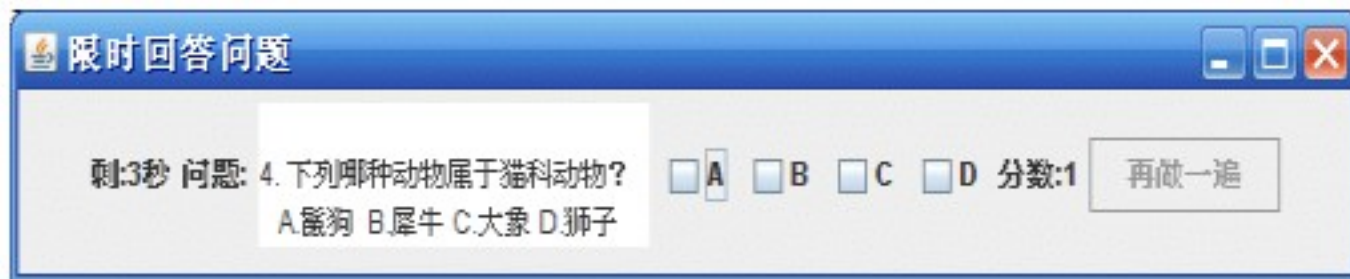


图 12.15 限时回答问题

