HZ BOOKS
华章IT

资深Java专家10年经验总结，全程案例式讲解，首本全面介绍Java多线程编程技术的专著

结合大量实例，全面讲解Java多线程编程中的并发访问、线程间通信、锁等最难突破的核心技术与应用实践

Java
核心技术
系列

Java
多线程编程核心技术

Java Multi-thread Programming

高洪岩 著

机械工业出版社
China Machine Press

提供各种书籍pdf下载，如有需要，请联系 QQ: 461573687

PDF制作说明：

本人可以提供各种PDF电子书资料，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 QQ: 461573687, 或者 QQ: 2404062482。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ。因PDF电子书都有版权，请不要随意传播，最近pdf也越来越难做了，希望大家尊重下个人劳动，谢谢！

备用QQ:2404062482

Java
核心技术
系列

Java
多线程编程
核心技术

Java Multi-thread Programming

高洪岩 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Java 多线程编程核心技术 / 高洪岩著 . —北京：机械工业出版社，2015.6
（Java 核心技术系列）

ISBN 978-7-111-50206-7

I. J… II. 高… III. JAVA 语言 – 程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2015）第 098874 号

Java 多线程编程核心技术

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：高婧雅 责任校对：董纪丽
印 刷：三河市宏图印务有限公司 版 次：2015 年 6 月第 1 版第 1 次印刷
开 本：186mm × 240mm 1/16 印 张：19.75
书 号：ISBN 978-7-111-50206-7 定 价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzit@hzbook.com

版权所有 • 侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

华章IT
HZBOOKS | Information Technology

Preface 前　言

为什么要写这本书

早在几年前笔者就曾想过整理一份与 Java 多线程有关的稿件，因为市面上所有的 Java 书籍都是以一章或两章的篇幅介绍多线程技术，并没有完整地覆盖该技术的知识点，但可惜，苦于当时的时间及精力有限，一直没有达成所愿。

也许是注定的安排，我目前所在的单位是集技术与教育为一体的软件类企业。我在工作中发现很多学员在学习完 JavaSE/JavaEE 之后想对更深入的技术进行探索，比如在对大数据、分布式、高并发类的专题进行攻克时，立即遇到针对 java.lang 包中 Thread 类的学习，但 Thread 类的学习并不像 JDBC 那样简单，学习多线程会遇到太多的问题、弯路以及我们所谓的“坑”，为了带领学员在技术层面上进行更高的追求，我将多线程的技术点以教案的方式进行整理，在课堂上与同学们一起学习、交流，同学们反响也非常热烈。此至，若干年前的心愿终于了却，学员们也很期待这本书能出版发行，因为这样他们就有了真正的纸质参考资料，其他爱好 Java 多线程的朋友们也在期盼本书的出版。本书能促进他们相互交流与学习，这就是我最大的心愿。

本书秉承大道至简的主导思想，只介绍 Java 多线程开发中最值得关注的内容，希望能抛砖引玉，以个人的一些想法和见解，为读者拓展出更深入、更全面的思路。

本书特色

在本书写作的过程中，我尽量减少“啰嗦”的文字语言，全部用案例来讲解技术点的实现，使读者看到代码及运行结果后就可以知道此项目要解决的是什么问题，类似于网络中的博客风格，可让读者用最短的时间学完相关知识点，明白这些知识点是如何应用的，以及在

使用时要避免什么。本书就像“瑞士军刀”一样，精短小，但却非常锋利，可帮读者快速学习知识并解决问题。

读者对象

本书适合所有 Java 程序员阅读，尤其适合以下读者：

- Java 多线程开发者
- Java 并发开发者
- 系统架构师
- 大数据开发者
- 其他对多线程技术感兴趣的人员

如何阅读本书

在整理本书时，我一直本着实用、易懂的原则，最终整理出 7 章：

第 1 章讲解了 Java 多线程的基础，包括 Thread 类的核心 API 的使用。

第 2 章讲解了在多线程中对并发访问的控制，主要就是 synchronized 的使用，由于此关键字在使用上非常灵活，所以书中用了很多案例来介绍此关键字的使用，为读者学习同步相关内容打好坚实的基础。

第 3 章介绍线程并不是孤独的，它们之间要通信，要交互。本章主要介绍 wait()、notifyAll() 和 notify() 方法的使用，使线程间能互相通信，合作完成任务。本章还介绍了 ThreadLocal 类的使用。学习完本章，读者就能在 Thread 多线程中进行数据的传递了。

第 4 章讲解了 synchronized 关键字，它使用起来比较麻烦，所以在 Java 5 中提供了 Lock 对象，以求能更好地实现并发访问时的同步处理，包括读写锁等相关技术点。

第 5 章讲解了 Timer 定时器类，其内部实现就是使用的多线程技术。定时器的计划任务执行是很重要的技术点，包括在 Android 开发时都会有深入的使用，所以会为读者详细讲解。

第 6 章讲解的单例模式虽然很简单，但如果遇到多线程将会变得非常麻烦，如何在多线程中解决这么棘手的问题呢？本章将全面介绍解决方案。

第 7 章，在整理稿件的过程中肯定会出现一些技术知识点的空缺，前面被遗漏的技术案例将在本章进行补充，以帮助读者形成完整的多线程的知识体系。编写本章的目的就是尽量使本书不存在技术空白点。

勘误和支持

由于我的水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正，让我与大家一起，在技术之路上互勉共进。我的邮箱是 279377921@qq.com，期待能够得到你们的真挚反馈。本书的源代码可以在华章网站（www.hzbook.com）下载。

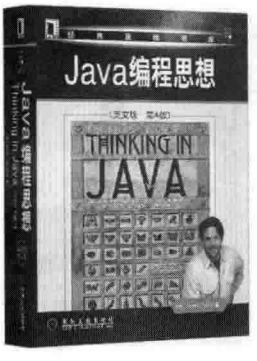
致谢

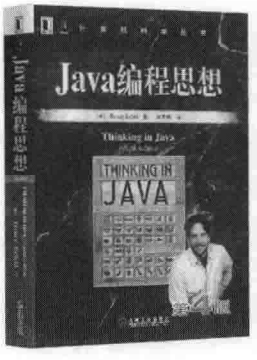
感谢所在单位领导的支持与厚爱，使我在技术道路上更有信心。

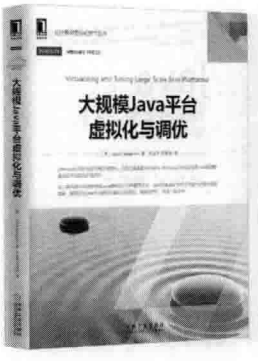
感谢机械工业出版社华章公司的高婧雅和杨福川，因为有了你们的鼓励、帮助和引导，我才能顺利完成本书。

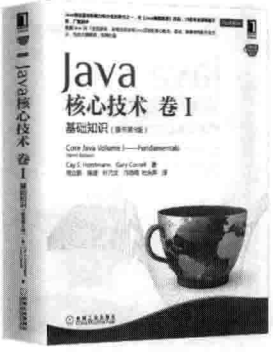
高洪岩

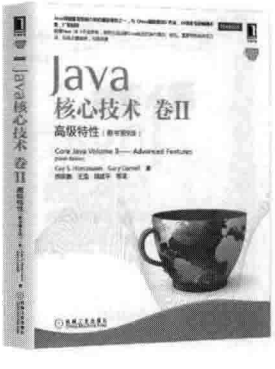
推荐阅读


Java编程思想
THINKING IN JAVA

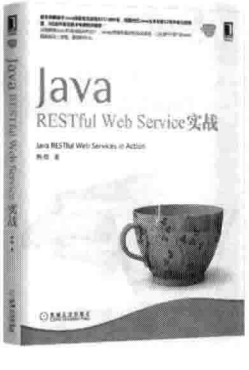

Java编程思想
Thinking in Java
THINKING IN JAVA

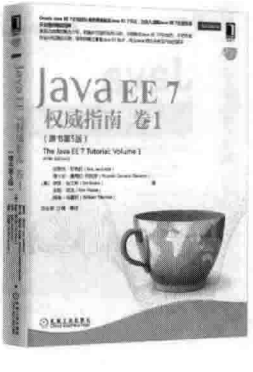

大规模Java平台
虚拟化与调优

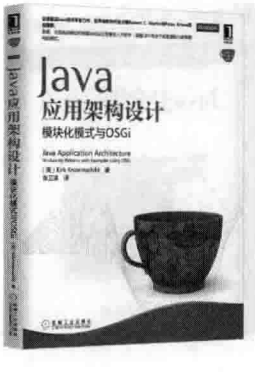

Java
核心技术 卷I
基础知识


Java
核心技术 卷II
高级特性

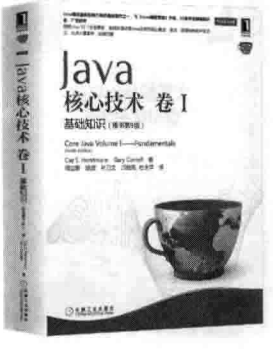

Java
虚拟机规范
(Java SE 7版)

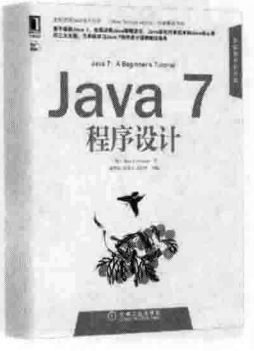

Java
RESTful Web Service实战

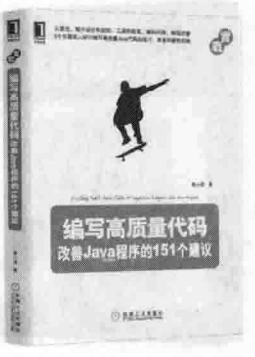

Java EE 7
权威指南 卷1

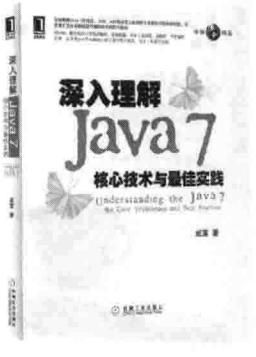

Java
应用架构设计
模块化模式与OSGi

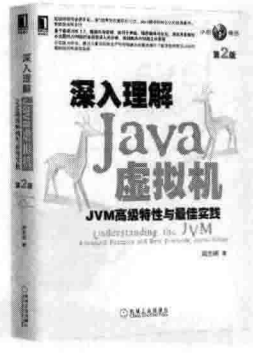
推荐阅读

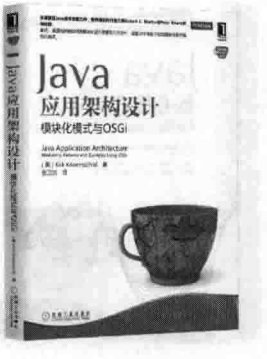

Java
核心技术 卷I
基础知识

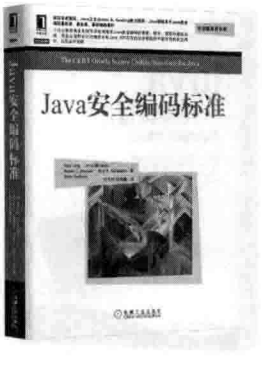

Java 7
程序设计

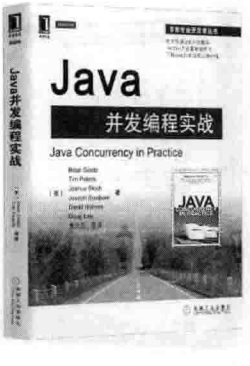

编写高质量代码
改善Java程序的151个建议

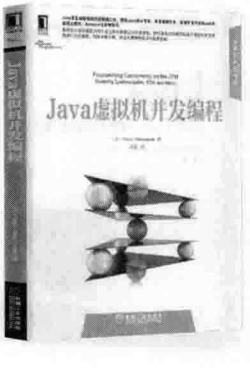

深入理解
Java 7
核心技术与最佳实践


深入理解
Java
虚拟机
JVM高级特性与最佳实践


Java
应用架构设计
模块化模式与OSGi


Java安全编码标准


Java
并发编程实战
Java Concurrency in Practice


Java虚拟机并发编程

目　录 *Contents*

第 1 章 Chapter 1

Java 多线程技能

作为本书的第 1 章，一定要引导读者快速进入 Java 多线程的学习，所以本章中主要介绍 Thread 类中的核心方法。Thread 类的核心方法较多，读者应该着重掌握如下关键技术点：

- 线程的启动
- 如何使线程暂停
- 如何使线程停止
- 线程的优先级
- 线程安全相关的问题

上面的 5 点也是本章学习的重点与思路，掌握这些内容是学习 Java 多线程的必经之路。

1.1 进程和多线程的概念及线程的优点

本节主要介绍在 Java 语言中使用多线程技术。但讲到多线程这个技术时不得不提及“进程”这个概念，“百度百科”里对“进程”的解释如图 1-1 所示。

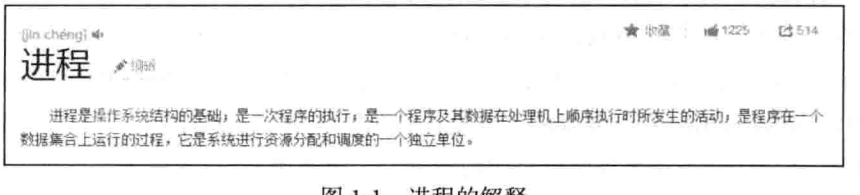


图 1-1 进程的解释

初看这段文字会觉得十分的抽象，难以理解，但如果你看到图 1-2 所示的内容，那么你对进程还不能理解吗？

图 1-2 Windows 7 系统中的进程列表

难道可以将一个正在操作系统中运行的 exe 程序理解成一个“进程”吗？没错！

通过查看“Windows 任务管理器”中的列表，完全可以将运行在内存中的 exe 文件理解成进程，进程是受操作系统管理的基本运行单元。

那什么是线程呢？线程可以理解成是在进程中独立运行的子任务。比如，QQ.exe 运行时就有很多的子任务在同时运行。再如，好友视频线程、下载文件线程、传输数据线程、发送表情线程等，这些不同的任务或者说功能都可以同时运行，其中每一项任务完全可以理解成是“线程”在工作，传文件、听音乐、发送图片表情等功能都有对应的线程在后台默默地运行。

这样做有什么优点呢？更具体来讲，使用多线程有什么优点呢？其实如果读者有使用“多任务操作系统”的经验，比如 Windows 系列，那么它的方便性大家应该都有体会：使用多任务操作系统 Windows 后，可以最大限度地利用 CPU 的空闲时间来处理其他的任务，比如一边让操作系统处理正在由打印机打印的数据，一边使用 Word 编辑文档。而 CPU 在这些任务之间不停地切换，由于切换的速度非常快，给使用者的感受就是这些任务似乎在同时运

行。所以使用多线程技术后，可以在同一时间内运行更多不同种类的任务。

为了更加有效地理解多线程的优势，看一下如图 1-3 所示的单任务的模型图，理解一下单任务的缺点。

在图 1-3 中，任务 1 和任务 2 是两个完全独立、互不相关的任务，任务 1 是在等待远程服务器返回数据，以便进行后期的处理，这时 CPU 一直处于等待状态，一直在“空运行”。如果任务 2 是在 10 秒之后被运行，虽然执行任务 2 用的时间非常短，仅仅是 1 秒，但也必须在任务 1 运行结束后才可以运行任务 2。本程序是运行在单任务环境中，所以任务 2 有非常长的等待时间，系统运行效率大幅降低。单任务的特点就是排队执行，也就是同步，就像在 cmd 中输入一条命令后，必须等待这条命令执行完才可以执行下一条命令一样。这就是单任务环境的缺点，即 CPU 利用率大幅降低。

而多任务的环境如图 1-4 所示。

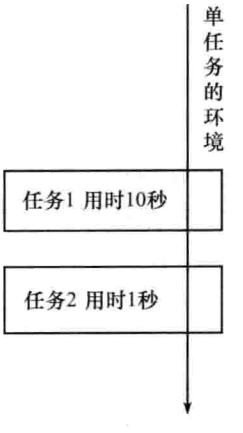


图 1-3　单任务运行环境

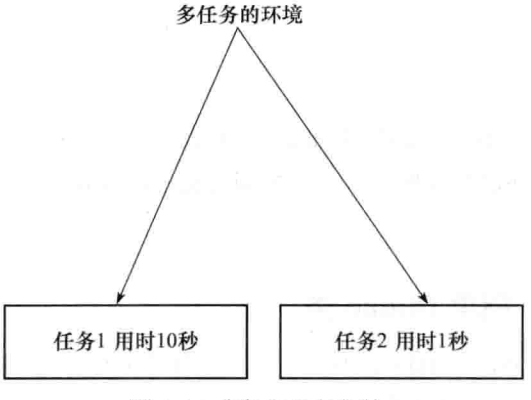


图 1-4　多任务运行环境

在图 1-4 中可以发现，CPU 完全可以在任务 1 和任务 2 之间来回切换，使任务 2 不必等到 10 秒再运行，系统的运行效率大大得到提升。这就是要使用多线程技术、要学习多线程的原因。这是多线程技术的优点，使用多线程也就是在使用异步。

注意　多线程是异步的，所以千万不要把 Eclipse 里代码的顺序当成线程执行的顺序，线程被调用的时机是随机的。

1.2　使用多线程

想学习一个技术就要“接近”它，所以在本节，首先用一个示例来接触一下线程。

一个进程正在运行时至少会有 1 个线程在运行，这种情况在 Java 中也是存在的。这些线程在后台默默地执行，比如调用 public static void main() 方法的线程就是这样的，而且它是由 JVM 创建的。

创建示例项目 callMainMethodMainThread，创建 Test.java 类。代码如下：

```
package test;
public class Test {
public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName());
    }
}
```

程序运行后的效果如图 1-5 所示。

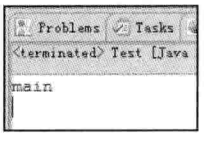


图 1-5　主线程 main

在控制台中输出的 main 其实就是一个名称叫作 main 的线程在执行 main() 方法中的代码。另外需要说明一下，在控制台输出的 main 和 main 方法没有任何的关系，仅仅是名字相同而已。

1.2.1　继承 Thread 类

在 Java 的 JDK 开发包中，已经自带了对多线程技术的支持，可以很方便地进行多线程编程。实现多线程编程的方式主要有两种，一种是继承 Thread 类，另一种是实现 Runnable 接口。

但在学习如何创建新的线程前，先来看看 Thread 类的结构，如下：

```
public class Thread implements Runnable
```

从上面的源代码中可以发现，Thread 类实现了 Runnable 接口，它们之间具有多态关系。

其实，使用继承 Thread 类的方式创建新线程时，最大的局限就是不支持多继承，因为 Java 语言的特点就是单根继承，所以为了支持多继承，完全可以实现 Runnable 接口的方式，一边实现一边继承。但用这两种方式创建的线程在工作时的性质是一样的，没有本质的区别。

本节来看一下第一种方法。创建名称为 t1 的 Java 项目，创建一个自定义的线程类 MyThread.java，此类继承自 Thread，并且重写 run 方法。在 run 方法中，写线程要执行的任务的代码如下：

```
package com.mythread.www;
```

```
public class MyThread extends Thread {
    @Override
    public void run() {
        super.run();
        System.out.println("MyThread");
    }
}
```

运行类代码如下：

```
package test;
import com.mythread.www.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread mythread = new MyThread();
        mythread.start();
        System.out.println("运行结束！");
    }
}
```

运行结果如图 1-6 所示。

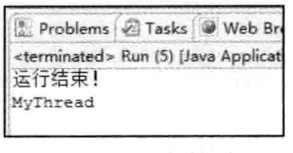


图 1-6　运行结果

从图 1-6 中的运行结果来看，MyThread.java 类中的 run 方法执行的时间比较晚，这也说明在使用多线程技术时，代码的运行结果与代码执行顺序或调用顺序是无关的。

线程是一个子任务，CPU 以不确定的方式，或者说是以随机的时间来调用线程中的 run 方法，所以就会出现先打印“运行结束！”后输出“MyThread”这样的结果了。

注意　如果多次调用 start() 方法，则会出现异常 Exception in thread "main" java.lang.IllegalThreadStateException。

上面介绍了线程的调用的随机性，下面将在名称为 randomThread 的 Java 项目中演示线程的随机性。

创建自定义线程类 MyThread.java，代码如下：

```
package mythread;
public class MyThread extends Thread {
    @Override
    public void run() {
```

```
        try {
            for (int i = 0; i < 10; i++) {
                int time = (int) (Math.random() * 1000);
                Thread.sleep(time);
                System.out.println("run=" + Thread.currentThread().getName());
            }
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

再创建运行类 Test.java，代码如下：

```
package test;
import mythread.MyThread;
public class Test {
    public static void main(String[] args) {
        try {
            MyThread thread = new MyThread();
            thread.setName("myThread");
            thread.start();
            for (int i = 0; i < 10; i++) {
                int time = (int) (Math.random() * 1000);
                Thread.sleep(time);
                System.out.println("main=" + Thread.currentThread().getName());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

在代码中，为了展现出线程具有随机特性，所以使用随机数的形式来使线程得到挂起的效果，从而表现出 CPU 执行哪个线程具有不确定性。

Thread.java 类中的 start() 方法通知“线程规划器”此线程已经准备就绪，等待调用线程对象的 run() 方法。这个过程其实就是让系统安排一个时间来调用 Thread 中的 run() 方法，也就是使线程得到运行，启动线程，具有异步执行的效果。如果调用代码 thread.run() 就不是异步执行了，而是同步，那么此线程对象并不交给“线程规划器”来进行处理，而是由 main 主线程来调用 run() 方法，也就是必须等 run() 方法中的代码执行完后才可以执行后面的代码。

以异步的方式运行的效果如图 1-7 所示。

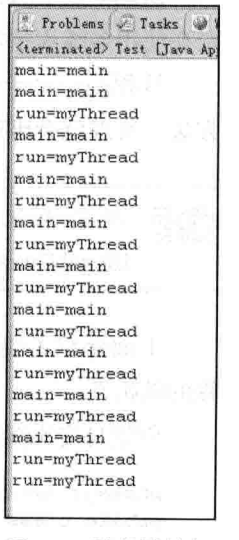


图 1-7 随机被执行的线程

另外还需要注意一下，执行 start() 方法的顺序不代表线程启动的顺序。创建测试用的项目名称为 z，类 MyThread.java 代码如下：

```
package extthread;
public class MyThread extends Thread {
    private int i;
    public MyThread(int i) {
        super();
        this.i = i;
    }
    @Override
    public void run() {
        System.out.println(i);
    }
}
```

运行类 Test.java 代码如下：

```
package test;
import extthread.MyThread;
public class Test {
    public static void main(String[] args) {
        MyThread t11 = new MyThread(1);
        MyThread t12 = new MyThread(2);
        MyThread t13 = new MyThread(3);
        MyThread t14 = new MyThread(4);
        MyThread t15 = new MyThread(5);
        MyThread t16 = new MyThread(6);
        MyThread t17 = new MyThread(7);
        MyThread t18 = new MyThread(8);
        MyThread t19 = new MyThread(9);
        MyThread t110 = new MyThread(10);
        MyThread t111 = new MyThread(11);
        MyThread t112 = new MyThread(12);
        MyThread t113 = new MyThread(13);
        t11.start();
        t12.start();
        t13.start();
        t14.start();
        t15.start();
        t16.start();
        t17.start();
        t18.start();
        t19.start();
        t110.start();
        t111.start();
        t112.start();
        t113.start();
    }
}
```

程序运行后的结果如图 1-8 所示。

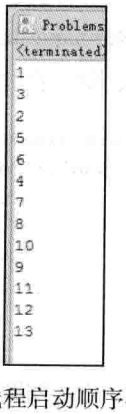


图 1-8　线程启动顺序与 start() 执行顺序无关

1.2.2 实现 Runnable 接口

如果欲创建的线程类已经有一个父类了，这时就不能再继承自 Thread 类了，因为 Java 不支持多继承，所以就需要实现 Runnable 接口来应对这样的情况。

创建项目 t2，继续创建一个实现 Runnable 接口的类 MyRunnable，代码如下：

```
package myrunnable;
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("运行中!");
    }
}
```

如何使用这个 MyRunnable.java 类呢？这就要看一下 Thread.java 的构造函数了，如图 1-9 所示。

构造方法摘要

Thread()
分配新的 Thread 对象。

Thread(Runnable target)
分配新的 Thread 对象。

Thread(Runnable target, String name)
分配新的 Thread 对象。

Thread(String name)
分配新的 Thread 对象。

Thread(ThreadGroup group, Runnable target)
分配新的 Thread 对象。

Thread(ThreadGroup group, Runnable target, String name)
分配新的 Thread 对象，以便将 target 作为其运行对象，将指定的 name 作为其名称，并作为 group 所引用的线程组的一员。

Thread(ThreadGroup group, Runnable target, String name, long stackSize)
分配新的 Thread 对象，以便将 target 作为其运行对象，将指定的 name 作为其名称，作为 group 所引用的线程组的一员，并具有指定的*堆栈大小*。

Thread(ThreadGroup group, String name)
分配新的 Thread 对象。

图 1-9 Thread 构造函数

在 Thread.java 类的 8 个构造函数中，有两个构造函数 Thread(Runnable target) 和 Thread(Runnable target,String name) 可以传递 Runnable 接口，说明构造函数支持传入一个 Runnable 接口的对象。运行类代码如下：

```
public class Run {
    public static void main(String[] args) {
        Runnable runnable=new MyRunnable();
        Thread thread=new Thread(runnable);
        thread.start();
        System.out.println("运行结束!");
    }
}
```

运行结果如图 1-10 所示。

图 1-10 所示的打印结果没有什么特殊之处。

使用继承 Thread 类的方式来开发多线程应用程序在设计上是有局限性的，因为 Java 是单根继承，不支持多继承，所以为了改变这种限制，可以使用实现 Runnable 接口的方式来实现多线程技术。这也是上面的示例介绍的知识点。

另外需要说明的是，Thread.java 类也实现了 Runnable 接口，如图 1-11 所示。

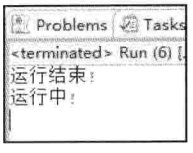


图 1-10　运行结果

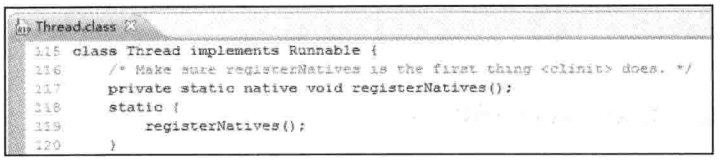


图 1-11　类 Thread 实现 Runnable 接口

那也就意味着构造函数 Thread(Runnable target) 不光可以传入 Runnable 接口的对象，还可以传入一个 Thread 类的对象，这样做完全可以将一个 Thread 对象中的 run() 方法交由其他的线程进行调用。

1.2.3　实例变量与线程安全

自定义线程类中的实例变量针对其他线程可以有共享与不共享之分，这在多个线程之间进行交互时是很重要的一个技术点。

（1）不共享数据的情况

不共享数据的情况如图 1-12 所示。

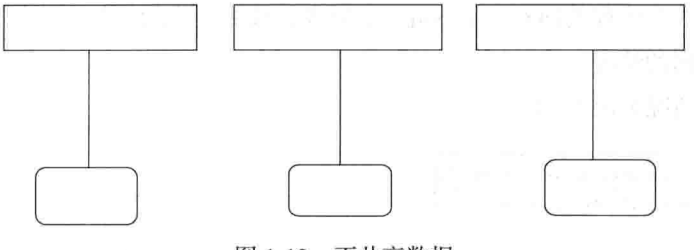


图 1-12　不共享数据

下面通过一个示例来看下数据不共享情况。

创建实验用的 Java 项目，名称为 t3，MyThread.java 类代码如下：

```
public class MyThread extends Thread {
    private int count = 5;
    public MyThread(String name) {
        super();
        this.setName(name);// 设置线程名称
    }
    @Override
```

```
    public void run() {
        super.run();
        while (count > 0) {
            count--;
            System.out.println("由 " + this.currentThread().getName()
                    + " 计算, count=" + count);
        }
    }
}
```

运行类 Run.java 代码如下：

```
public class Run {
    public static void main(String[] args) {
        MyThread a=new MyThread("A");
        MyThread b=new MyThread("B");
        MyThread c=new MyThread("C");
        a.start();
        b.start();
        c.start();
    }
}
```

不共享数据运行结果如图 1-13 所示。

由图 1-13 可以看到，一共创建了 3 个线程，每个线程都有各自的 count 变量，自己减少自己的 count 变量的值。这样的情况就是变量不共享，此示例并不存在多个线程访问同一个实例变量的情况。

如果想实现 3 个线程共同对一个 count 变量进行减法操作的目的，该如何设计代码呢？

（2）共享数据的情况

共享数据的情况如图 1-14 所示。

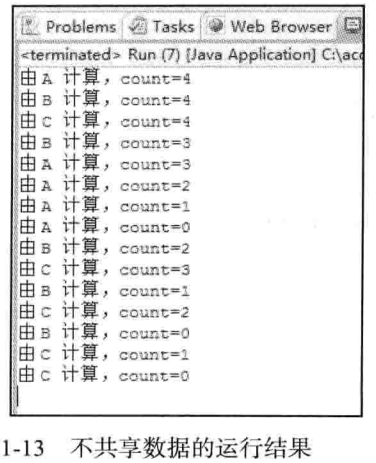


图 1-13　不共享数据的运行结果

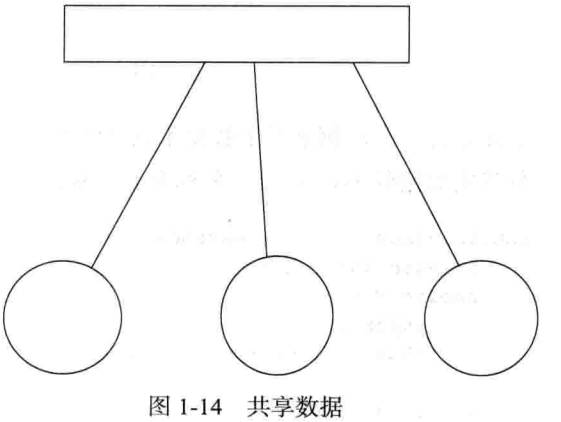


图 1-14　共享数据

共享数据的情况就是多个线程可以访问同一个变量，比如在实现投票功能的软件时，多个线程可以同时处理同一个人的票数。

下面通过一个示例来看下数据共享情况。

创建 t4 测试项目，MyThread.java 类代码如下：

```
public class MyThread extends Thread {
    private int count=5;
    @Override
     public void run() {
        super.run();
            count--;
    // 此示例不要用 for 语句，因为使用同步后其他线程就得不到运行的机会了，
    // 一直由一个线程进行减法运算
                System.out.println(" 由 "+this.currentThread().getName()+" 计算，
count="+count);
        }
}
```

运行类 Run.java 代码如下：

```
public class Run {
    public static void main(String[] args) {
        MyThread mythread=new MyThread();
        Thread a=new Thread(mythread,"A");
        Thread b=new Thread(mythread,"B");
        Thread c=new Thread(mythread,"C");
        Thread d=new Thread(mythread,"D");
        Thread e=new Thread(mythread,"E");
        a.start();
        b.start();
        c.start();
        d.start();
        e.start();
    }
}
```

运行结果如图 1-15 所示。

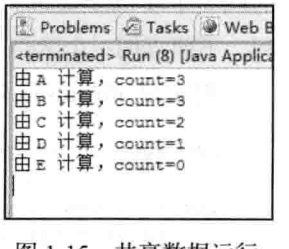


图 1-15　共享数据运行结果

从图 1-15 中可以看到，线程 A 和 B 打印出的 count 值都是 3，说明 A 和 B 同时对 count 进行处理，产生了“非线程安全”问题。而我们想要得到的打印结果却不是重复的，而是依次递减的。

在某些 JVM 中，i-- 的操作要分成如下 3 步：

1）取得原有 i 值。

2）计算 i-1。

3）对 i 进行赋值。

在这 3 个步骤中，如果有多个线程同时访问，那么一定会出现非线程安全问题。

其实这个示例就是典型的销售场景：5 个销售员，每个销售员卖出一个货品后不可以得出相同的剩余数量，必须在每一个销售员卖完一个货品后其他销售员才可以在新的剩余物品数上继续减 1 操作。这时就需要使多个线程之间进行同步，也就是用按顺序排队的方式进行减 1 操作。更改代码如下：

```
public class MyThread extends Thread {
    private int count=5;
    @Override
    synchronized public void run() {
        super.run();
            count--;
                System.out.println("由 "+this.currentThread().getName()+" 计算，
count="+count);
        }
}
```

重新运行程序，就不会出现值一样的情况了，如图 1-16 所示。

通过在 run 方法前加入 synchronized 关键字，使多个线程在执行 run 方法时，以排队的方式进行处理。当一个线程调用 run 前，先判断 run 方法有没有被上锁，如果上锁，说明有其他线程正在调用 run 方法，必须等其他线程对 run 方法调用结束后才可以执行 run 方法。这样也就实现了排队调用 run 方法的目的，也就达到了按顺序对 count 变量减 1 的效果了。synchronized 可以在任意对象及方法上加锁，而加锁的这段代码称为“互斥区”或“临界区”。

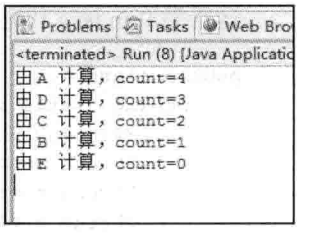


图 1-16 方法调用被同步

当一个线程想要执行同步方法里面的代码时，线程首先尝试去拿这把锁，如果能够拿到这把锁，那么这个线程就可以执行 synchronize 里面的代码。如果不能拿到这把锁，那么这个线程就会不断地尝试拿这把锁，直到能够拿到为止，而且是有多个线程同时去争抢这把锁。

本节中出现了一个术语“非线程安全”。非线程安全主要是指多个线程对同一个对象中的同一个实例变量进行操作时会出现值被更改、值不同步的情况，进而影响程序的执行流程。下面再用一个示例来学习一下如何解决“非线程安全”问题。

创建 t4_threadsafe 项目，来实现一下非线程安全的环境。LoginServlet.java 代码如下：

```
package controller;
//本类模拟成一个 Servlet 组件
public class LoginServlet {
    private static String usernameRef;
    private static String passwordRef;
    public static void doPost(String username, String password) {
        try {
            usernameRef = username;
```

```
            if (username.equals("a")) {
                Thread.sleep(5000);
            }
            passwordRef = password;
            System.out.println("username=" + usernameRef + " password="
                    + password);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

线程 ALogin.java 代码如下：

```
package extthread;
import controller.LoginServlet;
public class ALogin extends Thread {
    @Override
    public void run() {
        LoginServlet.doPost("a", "aa");
    }
}
```

线程 BLogin.java 代码如下：

```
package extthread;
import controller.LoginServlet;
public class BLogin extends Thread {
    @Override
    public void run() {
        LoginServlet.doPost("b", "bb");
    }
}
```

运行类 Run.java 代码如下：

```
public class Run {
    public static void main(String[] args) {
        ALogin a = new ALogin();
        a.start();
        BLogin b = new BLogin();
        b.start();
    }
}
```

程序运行后的效果如图 1-17 所示。

解决这个“非线程安全”的方法也是使用 synchronized 关键字。更改代码如下：

```
    synchronized public static void doPost(String username, String password) {
```

```
        try {
            usernameRef = username;
            if (username.equals("a")) {
                Thread.sleep(5000);
            }
            passwordRef = password;
            System.out.println("username=" + usernameRef + " password="
                    + password);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
```

程序运行后效果如图 1-18 所示。

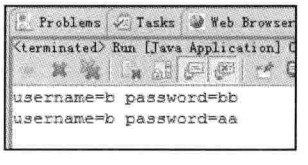


图 1-17 非线程安全

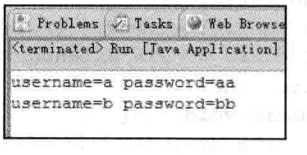


图 1-18 排队进入方法

1.2.4 留意 i-- 与 System.out.println() 的异常

在前面章节中，解决非线程安全问题使用的是 synchronized 关键字，本节将通过程序案例细化一下 println() 方法与 i++ 联合使用时“有可能”出现的另外一种异常情况，并说明其中的原因。

创建名称为 sameNum 的项目，自定义线程 MyThread.java 代码如下：

```
package extthread;
public class MyThread extends Thread {
    private int i = 5;
    @Override
    public void run() {
        System.out.println("i=" + (i--) + " threadName="
                + Thread.currentThread().getName());
    //注意：代码 i-- 由前面项目中单独一行运行改成在当前项目中在 println() 方法中直接进行打印
    }
}
```

运行类 Run.java 代码如下：

```
package test;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread run = new MyThread();
        Thread t1 = new Thread(run);
```

```
        Thread t2 = new Thread(run);
        Thread t3 = new Thread(run);
        Thread t4 = new Thread(run);
        Thread t5 = new Thread(run);
        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();
    }
}
```

程序运行后根据概率还是会出现非线程安全问题，如图 1-19 所示。

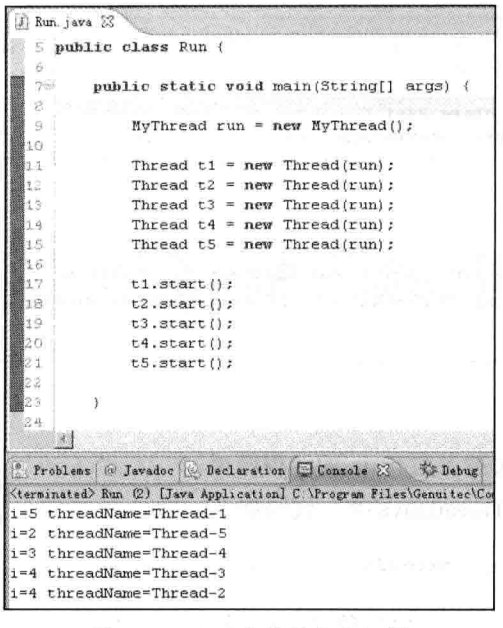


图 1-19　出现非线程安全问题

本实验的测试目的是：虽然 println() 方法在内部是同步的，但 i-- 的操作却是在进入 println() 之前发生的，所以有发生非线程安全问题的概率，如图 1-20 所示。

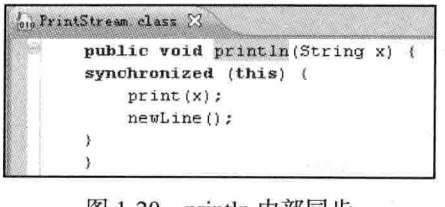


图 1-20　println 内部同步

所以，为了防止发生非线程安全问题，还是应继续使用同步方法。

1.3 currentThread() 方法

currentThread() 方法可返回代码段正在被哪个线程调用的信息。下面通过一个示例进行说明。

创建 t6 项目，创建 Run1.java 类代码如下：

```
public class Run1 {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName());
    }
}
```

程序运行结果如图 1-21 所示。

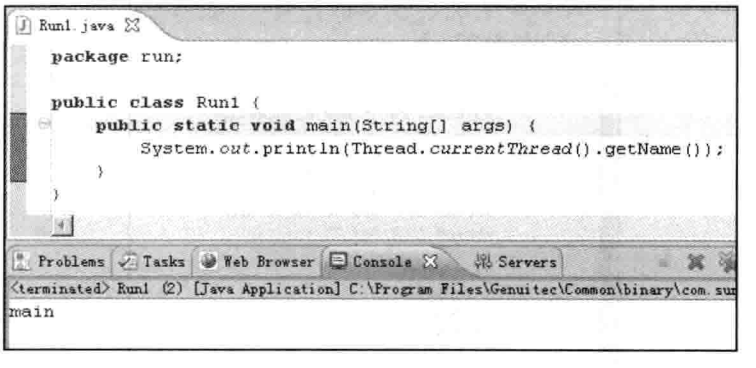


图 1-21 运行结果

结果说明，main 方法被名为 main 的线程调用。

继续实验，创建 MyThread.java 类。代码如下：

```
public class MyThread extends Thread {
    public MyThread() {
        System.out.println("构造方法的打印: " + Thread.currentThread().getName());
    }
    @Override
    public void run() {
        System.out.println("run 方法的打印: " + Thread.currentThread().getName());
    }
}
```

运行类 Run2.java 代码如下：

```
public class Run2 {
    public static void main(String[] args) {
        MyThread mythread = new MyThread();
        mythread.start();
        //mythread.run();
    }
}
```

程序运行结果如图 1-22 所示。

从图 1-22 中的运行结果可以发现，MyThread.java 类的构造函数是被 main 线程调用的，而 run 方法是被名称为 Thread-0 的线程调用的，run 方法是自动调用的方法。

文件 Run2.java 代码更改如下：

```
public class Run2 {
    public static void main(String[] args) {
        MyThread mythread = new MyThread();
        //mythread.start();
        mythread.run();
    }
}
```

运行结果如图 1-23 所示。

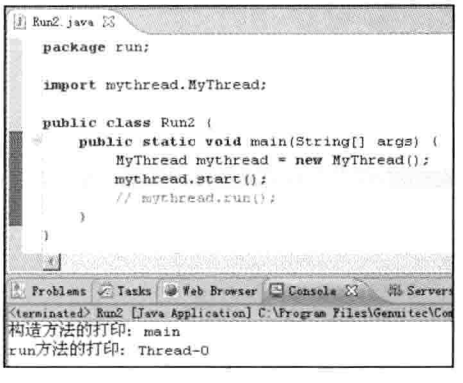


图 1-22　运行结果

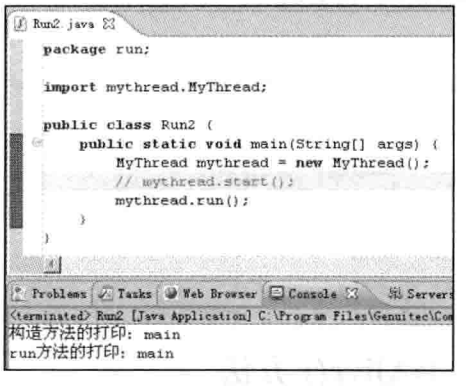


图 1-23　均被 main 主线程所调用

再来测试一个比较复杂的情况，创建测试用的项目 currentThreadExt，创建 Java 文件 CountOperate.java。代码如下：

```
package mythread;
public class CountOperate extends Thread {
    public CountOperate() {
        System.out.println("CountOperate---begin");
        System.out.println("Thread.currentThread().getName()="
                + Thread.currentThread().getName());
        System.out.println("this.getName()=" + this.getName());
        System.out.println("CountOperate---end");
    }
    @Override
    public void run() {
        System.out.println("run---begin");
        System.out.println("Thread.currentThread().getName()="
                + Thread.currentThread().getName());
        System.out.println("this.getName()=" + this.getName());
        System.out.println("run---end");
```

```
    }
}
```

创建 Run.java 文件，代码如下：

```
package test;
import mythread.CountOperate;
public class Run {
    public static void main(String[] args) {
        CountOperate c = new CountOperate();
        Thread t1 = new Thread(c);
        t1.setName("A");
        t1.start();
    }
}
```

程序运行结果如下：

```
CountOperate---begin
Thread.currentThread().getName()=main
this.getName()=Thread-0
CountOperate---end
run---begin
Thread.currentThread().getName()=A
this.getName()=Thread-0
run---end
```

1.4 isAlive() 方法

方法 isAlive() 的功能是判断当前的线程是否处于活动状态。

新建项目 t7，类文件 MyThread.java 代码如下：

```
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("run=" + this.isAlive());
    }
}
```

运行 Run.java 代码如下：

```
public class Run {
    public static void main(String[] args) {
        MyThread mythread = new MyThread();
        System.out.println("begin ==" + mythread.isAlive());
        mythread.start();
        System.out.println("end ==" + mythread.isAlive());
    }
}
```

程序运行结果如图 1-24 所示。

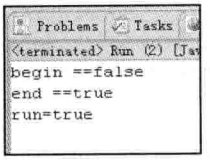


图 1-24　运行结果

方法 isAlive() 的作用是测试线程是否处于活动状态。什么是活动状态呢？活动状态就是线程已经启动且尚未终止。线程处于正在运行或准备开始运行的状态，就认为线程是“存活”的。

需要说明一下，如以下代码：

```
System.out.println("end ==" + mythread.isAlive());
```

虽然在上面的示例中打印的值是 true，但此值是不确定的。打印 true 值是因为 mythread 线程还未执行完毕，所以输出 true。如果代码更改如下：

```
public static void main(String[] args) throws InterruptedException {
    MyThread mythread = new MyThread();
    System.out.println("begin ==" + mythread.isAlive());
    mythread.start();
    Thread.sleep(1000);
    System.out.println("end ==" + mythread.isAlive());
}
```

则上述代码运行的结果输出为 false，因为 mythread 对象已经在 1 秒之内执行完毕。

另外，在使用 isAlive() 方法时，如果将线程对象以构造参数的方式传递给 Thread 对象进行 start() 启动时，运行的结果和前面示例是有差异的。造成这样的差异的原因还是来自于 Thread.currentThread() 和 this 的差异。下面测试一下这个实验。

创建测试用的 isaliveOtherTest 项目，创建 CountOperate.java 文件，代码如下：

```
package mythread;
public class CountOperate extends Thread {
    public CountOperate() {
        System.out.println("CountOperate---begin");
        System.out.println("Thread.currentThread().getName()="
                + Thread.currentThread().getName());
        System.out.println("Thread.currentThread().isAlive()="
                + Thread.currentThread().isAlive());
        System.out.println("this.getName()=" + this.getName());
        System.out.println("this.isAlive()=" + this.isAlive());
        System.out.println("CountOperate---end");
    }
    @Override
```

```
    public void run() {
        System.out.println("run---begin");
        System.out.println("Thread.currentThread().getName()="
                + Thread.currentThread().getName());
        System.out.println("Thread.currentThread().isAlive()="
                + Thread.currentThread().isAlive());
        System.out.println("this.getName()=" + this.getName());
        System.out.println("this.isAlive()=" + this.isAlive());
        System.out.println("run---end");
    }
}
```

创建 Run.java 文件，代码如下：

```
package test;
import mythread.CountOperate;
public class Run {
    public static void main(String[] args) {
        CountOperate c = new CountOperate();
        Thread t1 = new Thread(c);
        System.out.println("main begin t1 isAlive=" + t1.isAlive());
        t1.setName("A");
        t1.start();
        System.out.println("main end t1 isAlive=" + t1.isAlive());
    }
}
```

程序运行结果如下：

```
CountOperate---begin
Thread.currentThread().getName()=main
Thread.currentThread().isAlive()=true
this.getName()=Thread-0
this.isAlive()=false
CountOperate---end
main begin t1 isAlive=false
main end t1 isAlive=true
run---begin
Thread.currentThread().getName()=A
Thread.currentThread().isAlive()=true
this.getName()=Thread-0
this.isAlive()=false
run---end
```

1.5 sleep() 方法

方法 sleep() 的作用是在指定的毫秒数内让当前“正在执行的线程”休眠（暂停执行）。这个“正在执行的线程”是指 this.currentThread() 返回的线程。

通过一个示例进行说明。创建项目 t8，类 MyThread1.java 代码如下：

```
public class MyThread1 extends Thread {
    @Override
    public void run() {
        try {
            System.out.println("run threadName="
                    + this.currentThread().getName() + " begin");
            Thread.sleep(2000);
            System.out.println("run threadName="
                    + this.currentThread().getName() + " end");
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

运行类 Run1.java 代码如下：

```
public class Run1 {
    public static void main(String[] args) {
        MyThread1 mythread = new MyThread1();
        System.out.println("begin =" + System.currentTimeMillis());
        mythread.run();
        System.out.println("end   =" + System.currentTimeMillis());
    }
}
```

直接调用 run() 方法，程序运行结果如图 1-25 所示。

继续实验，创建 MyThread2.java 代码如下：

```
public class MyThread2 extends Thread {
    @Override
    public void run() {
        try {
            System.out.println("run threadName="
                    + this.currentThread().getName() + " begin ="
                    + System.currentTimeMillis());
            Thread.sleep(2000);
            System.out.println("run threadName="
                    + this.currentThread().getName() + " end   ="
                    + System.currentTimeMillis());
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

创建 Run2.java 代码如下：

```
public class Run2 {
    public static void main(String[] args) {
        MyThread2 mythread = new MyThread2();
        System.out.println("begin =" + System.currentTimeMillis());
        mythread.start();
        System.out.println("end   =" + System.currentTimeMillis());
    }
}
```

使用 start() 方法启动线程，程序运行结果如图 1-26 所示。

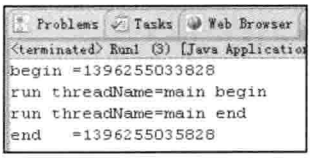


图 1-25 将 main 线程暂停了 2 秒

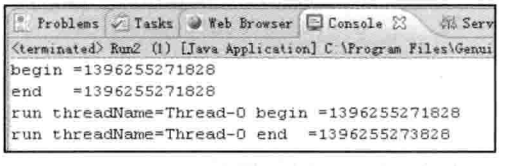


图 1-26 运行结果

由于 main 线程与 MyThread2 线程是异步执行的，所以首先打印的信息为 begin 和 end。而 MyThread2 线程是随后运行的，在最后两行打印 run begin 和 run end 相关的信息。

1.6 getId() 方法

getId() 方法的作用是取得线程的唯一标识。

创建测试用的项目 runThread，创建 Test.java 类，代码如下：

```
package test;
public class Test {
    public static void main(String[] args) {
        Thread runThread = Thread.currentThread();
        System.out.println(runThread.getName() + " " + runThread.getId());
    }
}
```

程序运行后的效果如图 1-27 所示。

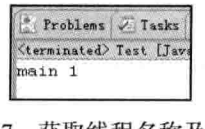


图 1-27 获取线程名称及 id 值

从打印的运行结果来看，当前执行代码的线程名称为 main，线程 id 值为 1。

1.7　停止线程

停止线程是在多线程开发时很重要的技术点，掌握此技术可以对线程的停止进行有效的处理。停止线程在 Java 语言中并不像 break 语句那样干脆，需要一些技巧性的处理。

使用 Java 内置支持多线程的类设计多线程应用是很常见的事情，然而，多线程给开发人员带来了一些新的挑战，如果处理不好就会导致超出预期的行为并且难以定位错误。

本节将讨论如何更好地停止一个线程。停止一个线程意味着在线程处理完任务之前停掉正在做的操作，也就是放弃当前的操作。虽然这看起来非常简单，但是必须做好防范措施，以便达到预期的效果。停止一个线程可以使用 Thread.stop() 方法，但最好不用它。虽然它确实可以停止一个正在运行的线程，但是这个方法是不安全的（unsafe），而且是已被弃用作废的（deprecated），在将来的 Java 版本中，这个方法将不可用或不被支持。

大多数停止一个线程的操作使用 Thread.interrupt() 方法，尽管方法的名称是“停止，中止”的意思，但这个方法不会终止一个正在运行的线程，还需要加入一个判断才可以完成线程的停止。关于此知识点在后面有专门的章节进行介绍。

在 Java 中有以下 3 种方法可以终止正在运行的线程：

1）使用退出标志，使线程正常退出，也就是当 run 方法完成后线程终止。

2）使用 stop 方法强行终止线程，但是不推荐使用这个方法，因为 stop 和 suspend 及 resume 一样，都是作废过期的方法，使用它们可能产生不可预料的结果。

3）使用 interrupt 方法中断线程。

这 3 种方法都会在后面的章节进行介绍。

1.7.1　停止不了的线程

本示例将调用 interrupt() 方法来停止线程，但 interrupt() 方法的使用效果并不像 for+break 语句那样，马上就停止循环。调用 interrupt() 方法仅仅是在当前线程中打了一个停止的标记，并不是真的停止线程。

创建名称为 t11 的项目，文件 MyThread.java 代码如下：

```
public class MyThread extends Thread {
    @Override
    public void run() {
        super.run();
        for (int i = 0; i < 500000; i++) {
            System.out.println("i=" + (i + 1));
        }
    }
}
```

运行类 Run.java 代码如下：

```
public class Run {
    public static void main(String[] args) {
        try {
            MyThread thread = new MyThread();
            thread.start();
            Thread.sleep(2000);
            thread.interrupt();
        } catch (InterruptedException e) {
            System.out.println("main catch");
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 1-28 所示。

把 Eclipse 软件中的控制台的日志复制到 EditPlus 软件中，确认一下日志是否是 50 万行，效果如图 1-29 所示。

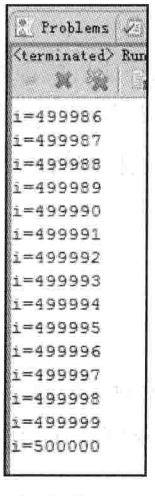


图 1-28 在控制台输出 50 万行的日志

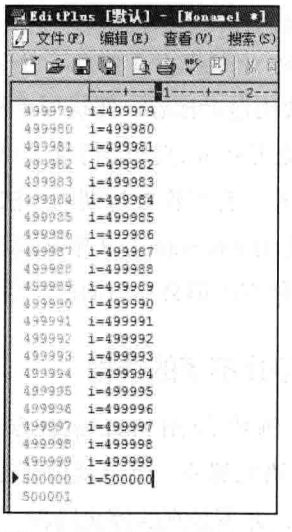


图 1-29 复制 50 万行日志

从运行的结果来看，调用 interrupt 方法并没有停止线程。如何停止线程呢？

1.7.2 判断线程是否是停止状态

在介绍如何停止线程的知识点前，先来看一下如何判断线程的状态是不是停止的。在 Java 的 SDK 中，Thread.java 类里提供了两种方法。

1）this.interrupted()：测试当前线程是否已经中断。

2）this.isInterrupted()：测试线程是否已经中断。

interrupted() 方法的声明如图 1-30 所示。

isInterrupted () 方法的声明如图 1-31 所示。

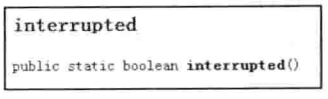


图 1-30　interrupted 方法的声明

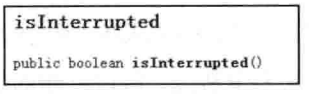


图 1-31　isInterrupted 方法的声明

那么这两个方法有什么区别呢？先来看看 this.interrupted() 方法的解释：测试当前线程是否已经中断，当前线程是指运行 this.interrupted() 方法的线程。为了对此方法有更深入的了解，创建项目，名称为 t12，类 MyThread.java 代码如下：

```
public class MyThread extends Thread {
    @Override
    public void run() {
        super.run();
        for (int i = 0; i < 500000; i++) {
            System.out.println("i=" + (i + 1));
        }
    }
}
```

类 Run.java 代码如下：

```
public class Run {
    public static void main(String[] args) {
        try {
            MyThread thread = new MyThread();
            thread.start();
            Thread.sleep(1000);
            thread.interrupt();
            //Thread.currentThread().interrupt();
            System.out.println("是否停止 1？ ="+thread.interrupted());
            System.out.println("是否停止 2？ ="+thread.interrupted());
        } catch (InterruptedException e) {
            System.out.println("main catch");
            e.printStackTrace();
        }
        System.out.println("end!");
    }
}
```

程序运行后的结果如图 1-32 所示。

类 Run.java 中虽然是在 thread 对象上调用以下代码：

```
thread.interrupt();
```

来停止 thread 对象所代表的线程，在后面又使用以下代码：

```
System.out.println("是否停止1？ ="+thread.interrupted());
System.out.println("是否停止2？ ="+thread.interrupted());
```

来判断 thread 对象所代表的线程是否停止，但从控制台打印的结果来看，线程并未停止，这也就证明了 interrupted() 方法的解释：测试当前线程是否已经中断。这个“当前线程”是 main，它从未中断过，所以打印的结果是两个 false。

如何使 main 线程产生中断效果呢？创建 Run2.java 代码如下：

```
public class Run2 {
    public static void main(String[] args) {
        Thread.currentThread().interrupt();
        System.out.println("是否停止1？ =" + Thread.interrupted());
        System.out.println("是否停止2？ =" + Thread.interrupted());
        System.out.println("end!");
    }
}
```

程序运行后的效果如图 1-33 所示。

图 1-32 运行结果

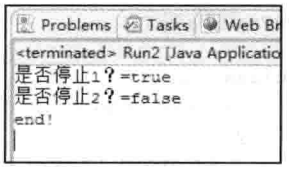


图 1-33 主线程 main 已是停止状态

从上述的结果来看，方法 interrupted() 的确判断出当前线程是否是停止状态。但为什么第 2 个布尔值是 false 呢？查看一下官方帮助文档中对 interrupted 方法的解释：

测试当前线程是否已经中断。线程的中断状态由该方法清除。换句话说，如果连续两次调用该方法，则第二次调用将返回 false（在第一次调用已清除了其中断状态之后，且第二次调用检验完中断状态前，当前线程再次中断的情况除外）。

文档已经解释得很详细，interrupted() 方法具有清除状态的功能，所以第 2 次调用 interrupted() 方法返回的值是 false。

介绍完 interrupted() 方法后再来看一下 isInterrupted() 方法，声明如下：

```
public boolean isInterrupted()
```

从声明中可以看出 isInterrupted() 方法不是 static 的。

继续创建 Run3.java 类，代码如下：

```
public class Run3 {
    public static void main(String[] args) {
        try {
            MyThread thread = new MyThread();
            thread.start();
            Thread.sleep(1000);
            thread.interrupt();
            System.out.println("是否停止 1？ ="+thread.isInterrupted());
            System.out.println("是否停止 2？ ="+thread.isInterrupted());
        } catch (InterruptedException e) {
            System.out.println("main catch");
            e.printStackTrace();
        }
        System.out.println("end!");
    }
}
```

程序运行结果如图 1-34 所示。

从结果中可以看到，方法 isInterrupted() 并未清除状态标志，所以打印了两个 true。

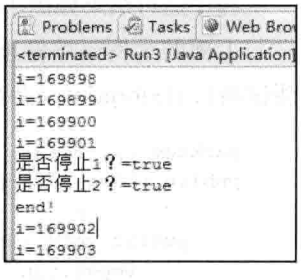


图 1-34　已经是停止状态

最后，再来看一下这两个方法的解释。

1）this.interrupted()：测试当前线程是否已经是中断状态，执行后具有将状态标志置清除为 false 的功能。

2）this.isInterrupted()：测试线程 Thread 对象是否已经是中断状态，但不清除状态标志。

1.7.3　能停止的线程——异常法

有了前面学习过的知识点，就可在线程中用 for 语句来判断一下线程是否是停止状态，如果是停止状态，则后面的代码不再运行即可。

创建实验用的项目 t13，类 MyThread.java 代码如下：

```
public class MyThread extends Thread {
    @Override
    public void run() {
        super.run();
        for (int i = 0; i < 500000; i++) {
            if (this.interrupted()) {
                System.out.println("已经是停止状态了！我要退出了！");
                break;
            }
            System.out.println("i=" + (i + 1));
        }
    }
}
```

类 Run.java 代码如下：

```
public class Run {
    public static void main(String[] args) {
        try {
            MyThread thread = new MyThread();
            thread.start();
            Thread.sleep(2000);
            thread.interrupt();
        } catch (InterruptedException e) {
            System.out.println("main catch");
            e.printStackTrace();
        }
        System.out.println("end!");
    }
}
```

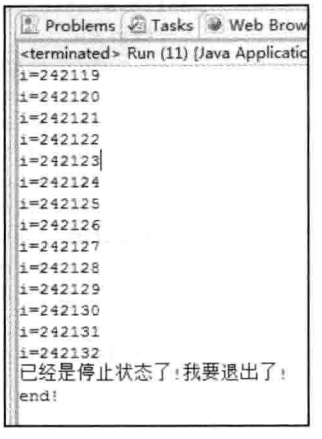


图 1-35 线程已经是停止状态

程序运行结果如图 1-35 所示。

上面的示例虽然停止了线程，但如果 for 语句下面还有语句，还是会继续运行的。创建测试项目 t13forprint，类 MyThread.java 代码如下：

```
package exthread;
public class MyThread extends Thread {
    @Override
    public void run() {
        super.run();
        for (int i = 0; i < 500000; i++) {
            if (this.interrupted()) {
                System.out.println("已经是停止状态了！我要退出了！");
                break;
            }
            System.out.println("i=" + (i + 1));
        }
        System.out.println("我被输出，如果此代码是 for 又继续运行，线程并未停止！ ");
    }
}
```

文件 Run.java 代码如下：

```
package test;
import exthread.MyThread;
import exthread.MyThread;
public class Run {
    public static void main(String[] args) {
        try {
            MyThread thread = new MyThread();
            thread.start();
            Thread.sleep(2000);
            thread.interrupt();
```

```
            } catch (InterruptedException e) {
                System.out.println("main catch");
                e.printStackTrace();
            }
            System.out.println("end!");
        }
    }
```

程序运行后输出结果如图 1-36 所示。

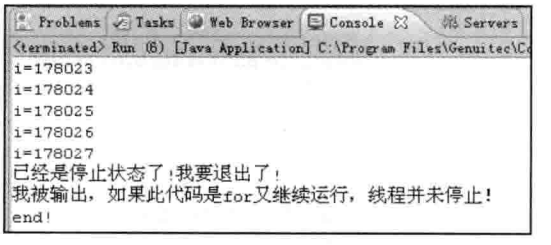


图 1-36　for 后面的语句继续运行

该如何解决语句继续运行的问题呢？看一下更新后的代码。

创建 t13_1 项目，类 MyThread.java 代码如下：

```
package exthread;
public class MyThread extends Thread {
    @Override
    public void run() {
        super.run();
        try {
            for (int i = 0; i < 500000; i++) {
                if (this.interrupted()) {
                    System.out.println("已经是停止状态了！我要退出了！");
                    throw new InterruptedException();
                }
                System.out.println("i=" + (i + 1));
            }
            System.out.println("我在 for 下面");
        } catch (InterruptedException e) {
            System.out.println("进 MyThread.java 类 run 方法中的 catch 了！");
            e.printStackTrace();
        }
    }
}
```

类 Run.java 代码如下：

```
package test;
import exthread.MyThread;
public class Run {
    public static void main(String[] args) {
        try {
            MyThread thread = new MyThread();
            thread.start();
            Thread.sleep(2000);
            thread.interrupt();
        } catch (InterruptedException e) {
            System.out.println("main catch");
            e.printStackTrace();
```

```
        }
        System.out.println("end!");
    }
}
```

运行结果如图 1-37 所示。

图 1-37 运行结果

1.7.4 在沉睡中停止

如果线程在 sleep() 状态下停止线程，会是什么效果呢？

新建项目 t14，类 MyThread.java 代码如下：

```
public class MyThread extends Thread {
    @Override
    public void run() {
        super.run();
        try {
            System.out.println("run begin");
            Thread.sleep(200000);
            System.out.println("run end");
        } catch (InterruptedException e) {
            System.out.println("在沉睡中被停止！进入catch!"+this.isInterrupted());
            e.printStackTrace();
        }
    }
}
```

文件 Run.java 代码如下：

```
public class Run {
    public static void main(String[] args) {
        try {
            MyThread thread = new MyThread();
            thread.start();
            Thread.sleep(200);
            thread.interrupt();
```

```
        } catch (InterruptedException e) {
            System.out.println("main catch");
            e.printStackTrace();
        }
        System.out.println("end!");
    }
}
```

程序运行后的效果如图 1-38 所示。

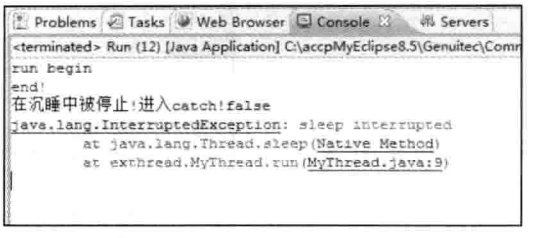


图 1-38　运行结果

从打印的结果来看，如果在 sleep 状态下停止某一线程，会进入 catch 语句，并且清除停止状态值，使之变成 false。

前一个实验是先 sleep 然后再用 interrupt() 停止，与之相反的操作在学习线程时也要注意。

新建项目 t15，类 MyThread.java 代码如下：

```
public class MyThread extends Thread {
    @Override
    public void run() {
        super.run();
        try {
            for(int i=0;i<100000;i++){
                System.out.println("i="+(i+1));
            }
            System.out.println("run begin");
            Thread.sleep(200000);
            System.out.println("run end");
        } catch (InterruptedException e) {
            System.out.println(" 先停止，再遇到了 sleep! 进入 catch!");
            e.printStackTrace();
        }
    }
}
```

类 Run.java 代码如下：

```
public class Run {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
        thread.interrupt();
        System.out.println("end!");
    }
}
```

运行结果如图 1-39 所示。

图 1-39　先执行 interrupt 停止线程

控制台最下面的输出如图 1-40 所示。

图 1-40 遇到 sleep 提示异常

1.7.5 能停止的线程——暴力停止

使用 stop() 方法停止线程则是非常暴力的。

示例项目名称 useStopMethodThreadTest，文件 MyThread.java 代码如下：

```
package testpackage;
public class MyThread extends Thread {
    private int i = 0;
    @Override
    public void run() {
        try {
            while (true) {
                i++;
                System.out.println("i=" + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

文件 Run.java 代码如下：

```
package test.run;
import testpackage.MyThread;
public class Run {
    public static void main(String[] args) {
        try {
            MyThread thread = new MyThread();
            thread.start();
            Thread.sleep(8000);
            thread.stop();
        } catch (InterruptedException e) {
```

```
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

程序运行后的效果如图 1-41 所示。

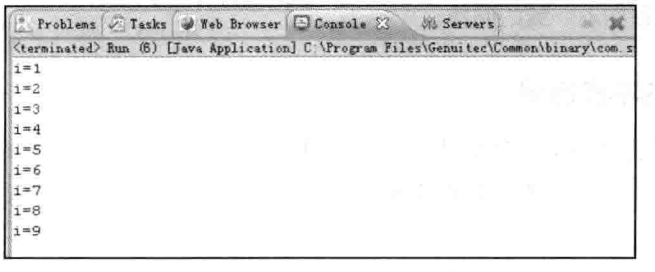


图 1-41　线程被暴力停止（stop）运行后图标呈灰色

1.7.6　方法 stop() 与 java.lang.ThreadDeath 异常

调用 stop() 方法时会抛出 java.lang.ThreadDeath 异常，但在通常的情况下，此异常不需要显式地捕捉。

创建测试用的项目 runMethodUseStopMethod，文件 MyThread.java 代码如下：

```
package testpackage;
public class MyThread extends Thread {
    @Override
    public void run() {
        try {
            this.stop();
        } catch (ThreadDeath e) {
            System.out.println("进入了 catch() 方法! ");
            e.printStackTrace();
        }
    }
}
```

文件 Run.java 代码如下：

```
package test.run;
import testpackage.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
    }
}
```

程序运行后的效果如图 1-42 所示。

方法 stop() 已经被作废，因为如果强制让线程停止则有可能使一些清理性的工作得不到完成。另外一个情况就是对锁定的对象进行了“解锁”，导致数据得不到同步的处理，出现数据不一致的问题。

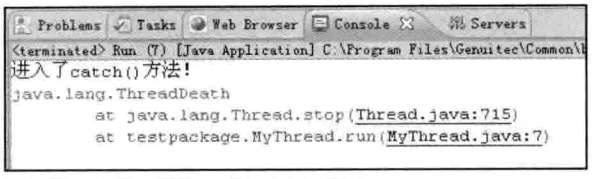


图 1-42 进入 catch 异常

1.7.7 释放锁的不良后果

使用 stop() 释放锁将会给数据造成不一致性的结果。如果出现这样的情况，程序处理的数据就有可能遭到破坏，最终导致程序执行的流程错误，一定要特别注意。

来看一个示例。

创建项目 stopThrowLock，文件 SynchronizedObject.java 代码如下：

```
package testpackage;
public class SynchronizedObject {
    private String username = "a";
    private String password = "aa";
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    synchronized public void printString(String username, String password) {
        try {
            this.username = username;
            Thread.sleep(100000);
            this.password = password;
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

文件 MyThread.java 代码如下：

```
package testpackage;
public class MyThread extends Thread {
    private SynchronizedObject object;
```

```
    public MyThread(SynchronizedObject object) {
        super();
        this.object = object;
    }
    @Override
    public void run() {
        object.printString("b", "bb");
    }
}
```

文件 Run.java 代码如下：

```
package test.run;
import testpackage.MyThread;
import testpackage.SynchronizedObject;
public class Run {
    public static void main(String[] args) {
        try {
            SynchronizedObject object = new SynchronizedObject();
            MyThread thread = new MyThread(object);
            thread.start();
            Thread.sleep(500);
            thread.stop();
            System.out.println(object.getUsername() + " "
                    + object.getPassword());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 1-43 所示。

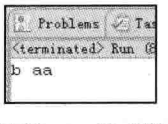


图 1-43　强制 stop 造成数据不一致

由于 stop() 方法已经在 JDK 中被标明是“作废 / 过期”的方法，显然它在功能上具有缺陷，所以不建议在程序中使用 stop() 方法。

1.7.8　使用 return 停止线程

将方法 interrupt() 与 return 结合使用也能实现停止线程的效果。

创建测试用的项目 useReturnInterrupt，线程类 MyThread.java 代码如下：

```
package extthread;
public class MyThread extends Thread {
```

```
    @Override
    public void run() {
            while (true) {
                if (this.isInterrupted()) {
                    System.out.println("停止了!");
                    return;
                }
                System.out.println("timer=" + System.currentTimeMillis());
            }
    }
}
```

运行类 Run.java 代码如下：

```
package test.run;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) throws InterruptedException {
        MyThread t=new MyThread();
        t.start();
        Thread.sleep(2000);
        t.interrupt();
    }
}
```

程序运行结果如图 1-44 所示。

不过还是建议使用“抛异常”的方法来实现线程的停止，因为在 catch 块中还可以将异常向上抛，使线程停止的事件得以传播。

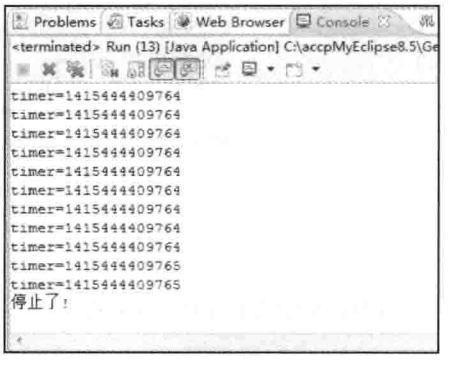


图 1-44 成功停止运行

1.8 暂停线程

暂停线程意味着此线程还可以恢复运行。在 Java 多线程中，可以使用 suspend() 方法暂停线程，使用 resume() 方法恢复线程的执行。

1.8.1 suspend 与 resume 方法的使用

本节将讲述如何使用 suspend 与 resume 方法。

创建测试用的项目 suspend_resume_test，文件 MyThread.java 代码如下：

```
package mythread;
public class MyThread extends Thread {
    private long i = 0;
    public long getI() {
```

```
        return i;
    }
    public void setI(long i) {
        this.i = i;
    }
    @Override
    public void run() {
        while (true) {
            i++;
        }
    }
}
```

文件 Run.java 代码如下：

```
package test.run;
import mythread.MyThread;
public class Run {
    public static void main(String[] args) {
        try {
            MyThread thread = new MyThread();
            thread.start();
            Thread.sleep(5000);
            //A 段
            thread.suspend();
            System.out.println("A= " + System.currentTimeMillis() + " i="
                    + thread.getI());
            Thread.sleep(5000);
            System.out.println("A= " + System.currentTimeMillis() + " i="
                    + thread.getI());
            //B 段
            thread.resume();
            Thread.sleep(5000);
            //C 段
            thread.suspend();
            System.out.println("B= " + System.currentTimeMillis() + " i="
                    + thread.getI());
            Thread.sleep(5000);
            System.out.println("B= " + System.currentTimeMillis() + " i="
                    + thread.getI());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行后的结果如图 1-45 所示。

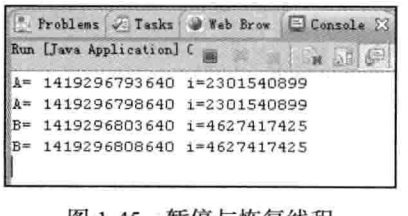


图 1-45　暂停与恢复线程

从控制台打印的时间上来看，线程的确被暂停了，而且还可以恢复成运行的状态。

1.8.2 suspend 与 resume 方法的缺点——独占

在使用 suspend 与 resume 方法时，如果使用不当，极易造成公共的同步对象的独占，使得其他线程无法访问公共同步对象。

创建 suspend_resume_deal_lock 项目，文件 SynchronizedObject.java 代码如下：

```
package testpackage;
public class SynchronizedObject {
    synchronized public void printString() {
        System.out.println("begin");
        if (Thread.currentThread().getName().equals("a")) {
            System.out.println("a 线程永远 suspend 了！ ");
            Thread.currentThread().suspend();
        }
        System.out.println("end");
    }
}
```

文件 Run.java 代码如下：

```
package test.run;
import testpackage.SynchronizedObject;
public class Run {
    public static void main(String[] args) {
        try {
            final SynchronizedObject object = new SynchronizedObject();
            Thread thread1 = new Thread() {
                @Override
                public void run() {
                    object.printString();
                }
            };
            thread1.setName("a");
            thread1.start();
            Thread.sleep(1000);
            Thread thread2 = new Thread() {
                @Override
                public void run() {
                    System.out
                            .println("thread2 启动了，但进入不了 printString() 方法！
只打印 1 个 begin");
                    System.out
                            .println(" 因为 printString() 方法被 a 线程锁定并且永远
suspend 暂停了！ ");
                    object.printString();
                }
            };
            thread2.start();
        } catch (InterruptedException e) {
```

```
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

程序运行后的结果如图 1-46 所示。

还有另外一种独占锁的情况也要格外注意，稍有不慎，就会掉进“坑”里。创建测试用的项目 suspend_resume_LockStop，类 MyThread.java 代码如下：

```
package mythread;
public class MyThread extends Thread {
    private long i = 0;
    @Override
    public void run() {
        while (true) {
            i++;
        }
    }
}
```

类 Run.java 代码如下：

```
package test.run;
import mythread.MyThread;
public class Run {
    public static void main(String[] args) {
        try {
            MyThread thread = new MyThread();
            thread.start();
            Thread.sleep(1000);
            thread.suspend();
            System.out.println("main end!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行后顺序打印的信息如图 1-47 所示。

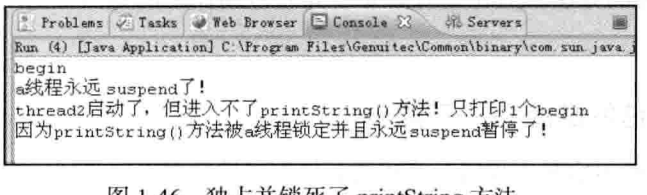


图 1-46　独占并锁死了 printString 方法

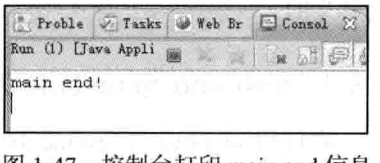


图 1-47　控制台打印 main end 信息

但如果将线程类 MyThread.java 更改如下：

```
package mythread;
public class MyThread extends Thread {
    private long i = 0;
    @Override
    public void run() {
        while (true) {
            i++;
            System.out.println(i);
        }
    }
}
```

再次运行程序，控制台将不打印 main end，运行结果如图 1-48 所示。

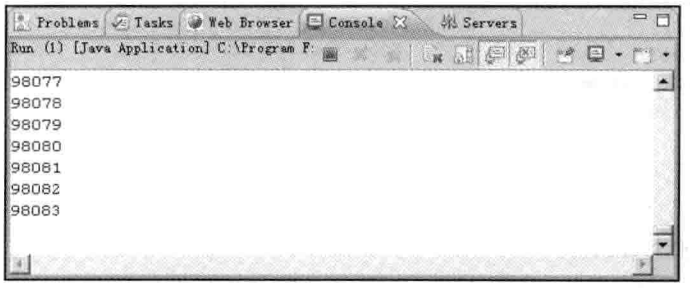


图 1-48 不打印 main end 信息

出现这样情况的原因是，当程序运行到 println() 方法内部停止时，同步锁未被释放。方法 println() 源代码如图 1-49 所示。

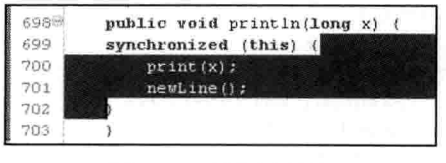


图 1-49 锁不被释放

这导致当前 PrintStream 对象的 println() 方法一直呈“暂停”状态，并且“锁未释放”，而 main() 方法中的代码 System.out.println("main end!"); 迟迟不能执行打印。

虽然 suspend() 方法是过期作废的方法，但还是有必要研究它过期作废的原因，这是很有意义的。

1.8.3 suspend 与 resume 方法的缺点——不同步

在使用 suspend 与 resume 方法时也容易出现因为线程的暂停而导致数据不同步的情况。

创建项目 suspend_resume_nosameValue，文件 MyObject.java 代码如下：

```
package myobject;
```

```
public class MyObject {
    private String username = "1";
    private String password = "11";
    public void setValue(String u, String p) {
        this.username = u;
        if (Thread.currentThread().getName().equals("a")) {
            System.out.println("停止 a 线程! ");
            Thread.currentThread().suspend();
        }
        this.password = p;
    }
    public void printUsernamePassword() {
        System.out.println(username + " " + password);
    }
}
```

文件 Run.java 代码如下：

```
package test;
import myobject.MyObject;
public class Run {
    public static void main(String[] args) throws InterruptedException {
        final MyObject myobject = new MyObject();
        Thread thread1 = new Thread() {
            public void run() {
                myobject.setValue("a", "aa");
            };
        };
        thread1.setName("a");
        thread1.start();
        Thread.sleep(500);
        Thread thread2 = new Thread() {
            public void run() {
                myobject.printUsernamePassword();
            };
        };
        thread2.start();
    }
}
```

程序运行结果如图 1-50 所示。

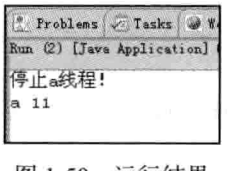


图 1-50　运行结果

程序运行的结果出现值不同步的情况，所以在程序中使用 suspend() 方法要格外注意。关于如何解决这些问题，请看后面的章节。

1.9 yield 方法

yield() 方法的作用是放弃当前的 CPU 资源，将它让给其他的任务去占用 CPU 执行时间。但放弃的时间不确定，有可能刚刚放弃，马上又获得 CPU 时间片。

在本示例中，可以取得运行的时间，作为比较结果，测试 yield 方法的使用效果。

创建 t17 项目，MyThread.java 文件代码如下：

```
package extthread;
public class MyThread extends Thread {
    @Override
    public void run() {
        long beginTime = System.currentTimeMillis();
        int count = 0;
        for (int i = 0; i < 50000000; i++) {
            //Thread.yield();
            count = count + (i + 1);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("用时: " + (endTime - beginTime) + "毫秒! ");
    }
}
```

文件 Run.java 代码如下：

```
package test;
import extthread.MyThread;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
    }
}
```

程序运行后的结果如图 1-51 所示。

将代码：

```
//Thread.yield();
```

去掉注释符号，再次运行，运行时间如图 1-52 所示。

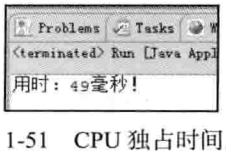


图 1-51 CPU 独占时间片

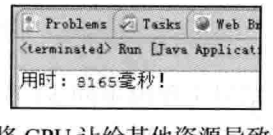


图 1-52 将 CPU 让给其他资源导致速度变慢

1.10　线程的优先级

在操作系统中，线程可以划分优先级，优先级较高的线程得到的 CPU 资源较多，也就是 CPU 优先执行优先级较高的线程对象中的任务。

设置线程优先级有助于帮“线程规划器”确定在下一次选择哪一个线程来优先执行。

设置线程的优先级使用 setPriority() 方法，此方法在 JDK 的源代码如下：

```
public final void setPriority(int newPriority) {
    ThreadGroup g;
    checkAccess();
    if (newPriority > MAX_PRIORITY || newPriority < MIN_PRIORITY) {
        throw new IllegalArgumentException();
    }
    if ((g = getThreadGroup()) != null) {
        if (newPriority > g.getMaxPriority()) {
            newPriority = g.getMaxPriority();
        }
        setPriority0(priority = newPriority);
    }
}
```

在 Java 中，线程的优先级分为 1 ~ 10 这 10 个等级，如果小于 1 或大于 10，则 JDK 抛出异常 throw new IllegalArgumentException()。

JDK 中使用 3 个常量来预置定义优先级的值，代码如下：

```
public final static int MIN_PRIORITY = 1;
public final static int NORM_PRIORITY = 5;
public final static int MAX_PRIORITY = 10;
```

1.10.1　线程优先级的继承特性

在 Java 中，线程的优先级具有继承性，比如 A 线程启动 B 线程，则 B 线程的优先级与 A 是一样的。

创建 t18 项目，创建 MyThread1.java 文件，代码如下：

```
package extthread;
public class MyThread1 extends Thread {
    @Override
    public void run() {
        System.out.println("MyThread1 run priority=" + this.getPriority());
        MyThread2 thread2 = new MyThread2();
        thread2.start();
    }
}
```

创建 MyThread2.java 文件，代码如下：

```
package extthread;
public class MyThread2 extends Thread {
    @Override
    public void run() {
        System.out.println("MyThread2 run priority=" + this.getPriority());
    }
}
```

文件 Run.java 代码如下：

```
package test;
import extthread.MyThread1;
public class Run {
    public static void main(String[] args) {
        System.out.println("main thread begin priority="
                + Thread.currentThread().getPriority());
        // Thread.currentThread().setPriority(6);
        System.out.println("main thread end   priority="
                + Thread.currentThread().getPriority());
        MyThread1 thread1 = new MyThread1();
        thread1.start();
    }
}
```

程序运行后的效果如图 1-53 所示。

将代码：

```
// Thread.currentThread().setPriority(6);
```

前的注释符号去掉，再次运行 Run.java 文件，显示结果如图 1-54 所示。

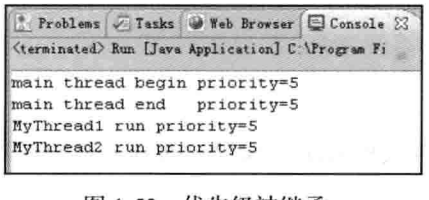


图 1-53 优先级被继承

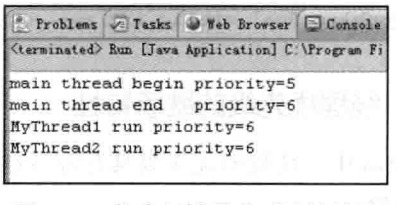


图 1-54 优先级被更改再继续继承

1.10.2 优先级具有规则性

虽然使用 setPriority() 方法可以设置线程的优先级，但还没有看到设置优先级所带来的效果。

创建名称为 t19 的项目，文件 MyThread1.java 代码如下：

```
package extthread;
import java.util.Random;
public class MyThread1 extends Thread {
    @Override
    public void run() {
        long beginTime = System.currentTimeMillis();
```

```
            long addResult = 0;
            for (int j = 0; j < 10; j++) {
                for (int i = 0; i < 50000; i++) {
                    Random random = new Random();
                    random.nextInt();
                    addResult = addResult + i;
                }
            }
            long endTime = System.currentTimeMillis();
              System.out.println("★ ★ ★ ★ ★thread 1 use time=" + (endTime -
beginTime));
        }
    }
```

文件 MyThread2.java 代码如下：

```
    package extthread;
    import java.util.Random;
    public class MyThread2 extends Thread {
        @Override
        public void run() {
            long beginTime = System.currentTimeMillis();
            long addResult = 0;
            for (int j = 0; j < 10; j++) {
                for (int i = 0; i < 50000; i++) {
                    Random random = new Random();
                    random.nextInt();
                    addResult = addResult + i;
                }
            }
            long endTime = System.currentTimeMillis();
            System.out.println("☆ ☆ ☆ ☆ ☆thread 2 use time=" + (endTime -
beginTime));
        }
    }
```

文件 Run.java 代码如下。

```
    package test;
    import extthread.MyThread1;
    import extthread.MyThread2;
    public class Run {
        public static void main(String[] args) {
            for (int i = 0; i < 5; i++) {
                MyThread1 thread1 = new MyThread1();
                thread1.setPriority(10);
                thread1.start();
                MyThread2 thread2 = new MyThread2();
                thread2.setPriority(1);
                thread2.start();
            }
        }
    }
```

文件 Run.java 在运行 3 次后的打印结果如图 1-55 所示。

```
★★★★★thread 1 use time=486    ★★★★★thread 1 use time=468    ★★★★★thread 1 use time=477
★★★★★thread 1 use time=515    ★★★★★thread 1 use time=500    ★★★★★thread 1 use time=500
★★★★★thread 1 use time=511    ★★★★★thread 1 use time=511    ★★★★★thread 1 use time=513
☆☆☆☆☆thread 2 use time=528    ★★★★★thread 1 use time=514    ☆☆☆☆☆thread 2 use time=516
★★★★★thread 1 use time=533    ★★★★★thread 1 use time=516    ★★★★★thread 1 use time=533
★★★★★thread 1 use time=550    ☆☆☆☆☆thread 2 use time=565    ★★★★★thread 1 use time=548
☆☆☆☆☆thread 2 use time=562    ☆☆☆☆☆thread 2 use time=571    ☆☆☆☆☆thread 2 use time=569
☆☆☆☆☆thread 2 use time=562    ☆☆☆☆☆thread 2 use time=584    ☆☆☆☆☆thread 2 use time=583
☆☆☆☆☆thread 2 use time=571    ☆☆☆☆☆thread 2 use time=575    ☆☆☆☆☆thread 2 use time=587
☆☆☆☆☆thread 2 use time=573    ☆☆☆☆☆thread 2 use time=584    ☆☆☆☆☆thread 2 use time=590
```

图 1-55 高优先级的线程总是先执行完

从图 1-55 中可以发现，高优先级的线程总是大部分先执行完，但不代表高优先级的线程全部先执行完。另外，不要以为 MyThread1 线程先被 main 线程所调用就会先执行完，出现这样的结果全是因为 MyThread1 线程的优先级是最高值为 10 造成的。当线程优先级的等级差距很大时，谁先执行完和代码的调用顺序无关。为了验证这个结论，继续实验，改变 Run.java 代码如下：

```
public class Run {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            MyThread1 thread1 = new MyThread1();
            thread1.setPriority(1);
            thread1.start();
            MyThread2 thread2 = new MyThread2();
            thread2.setPriority(10);
            thread2.start();
        }
    }
}
```

文件 Run.java 在运行 3 次后的打印结果如图 1-56 所示。

```
☆☆☆☆☆thread 2 use time=483    ☆☆☆☆☆thread 2 use time=448    ☆☆☆☆☆thread 2 use time=472
☆☆☆☆☆thread 2 use time=490    ☆☆☆☆☆thread 2 use time=503    ☆☆☆☆☆thread 2 use time=485
☆☆☆☆☆thread 2 use time=525    ☆☆☆☆☆thread 2 use time=505    ☆☆☆☆☆thread 2 use time=508
★★★★★thread 1 use time=527    ☆☆☆☆☆thread 2 use time=499    ☆☆☆☆☆thread 2 use time=479
★★★★★thread 1 use time=535    ☆☆☆☆☆thread 2 use time=532    ☆☆☆☆☆thread 2 use time=515
☆☆☆☆☆thread 2 use time=523    ★★★★★thread 1 use time=528    ★★★★★thread 1 use time=552
☆☆☆☆☆thread 2 use time=561    ★★★★★thread 1 use time=550    ★★★★★thread 1 use time=551
★★★★★thread 1 use time=564    ★★★★★thread 1 use time=572    ★★★★★thread 1 use time=571
★★★★★thread 1 use time=576    ★★★★★thread 1 use time=580    ★★★★★thread 1 use time=574
★★★★★thread 1 use time=579    ★★★★★thread 1 use time=580    ★★★★★thread 1 use time=585
```

图 1-56 大部分 thread2 先执行完

从图 1-56 中可以发现，大部分的 thread2 先执行完，也就验证了线程的优先级与代码执行顺序无关，出现这样的结果是因为 MyThread2 的优先级是最高的，说明线程的优先级具有一定的规则性，也就是 CPU 尽量将执行资源让给优先级比较高的线程。

1.10.3　优先级具有随机性

前面案例介绍了线程的优先级较高则优先执行完 run() 方法中的任务，但这个结果不能说的太肯定，因为线程的优先级还具有“随机性”，也就是优先级较高的线程不一定每一次都先执行完。

创建名称为 t20 的项目，文件 MyThread1.java 代码如下：

```
package extthread;
import java.util.Random;
public class MyThread1 extends Thread {
    @Override
    public void run() {
        long beginTime = System.currentTimeMillis();
        for (int i = 0; i < 1000; i++) {
            Random random = new Random();
            random.nextInt();
        }
        long endTime = System.currentTimeMillis();
          System.out.println("★ ★ ★ ★ ★thread 1 use time=" + (endTime -
beginTime));
    }
}
```

文件 MyThread2.java 代码如下：

```
package extthread;
import java.util.Random;
public class MyThread2 extends Thread {
    @Override
    public void run() {
        long beginTime = System.currentTimeMillis();
        for (int i = 0; i < 1000; i++) {
            Random random = new Random();
            random.nextInt();
        }
        long endTime = System.currentTimeMillis();
          System.out.println("☆ ☆ ☆ ☆ ☆thread 2 use time=" + (endTime -
beginTime));
    }
}
```

文件 Run.java 代码如下：

```
package test;
import extthread.MyThread1;
import extthread.MyThread2;
public class Run {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            MyThread1 thread1 = new MyThread1();
            thread1.setPriority(5);
            thread1.start();
            MyThread2 thread2 = new MyThread2();
            thread2.setPriority(6);
            thread2.start();
        }
    }
}
```

为了让结果体现“随机性”，所以两个线程的优先级一个设置为 5，另一个设置为 6，让优先级接近一些。

文件 Run.java 在运行 6 次后的打印结果如图 1-57 所示。

```
★★★★★thread 1 use time=3     ★★★★★thread 1 use time=3     ★★★★★thread 1 use time=3
☆☆☆☆☆thread 2 use time=1     ★★★★★thread 1 use time=1     ☆☆☆☆☆thread 2 use time=3
★★★★★thread 1 use time=4     ☆☆☆☆☆thread 2 use time=1     ★★★★★thread 1 use time=2
★★★★★thread 1 use time=2     ★★★★★thread 1 use time=1     ★★★★★thread 1 use time=0
★★★★★thread 1 use time=2     ☆☆☆☆☆thread 2 use time=2     ★★★★★thread 1 use time=3
☆☆☆☆☆thread 2 use time=3     ☆☆☆☆☆thread 2 use time=0     ☆☆☆☆☆thread 2 use time=3
☆☆☆☆☆thread 2 use time=3     ★★★★★thread 1 use time=0     ☆☆☆☆☆thread 2 use time=1
☆☆☆☆☆thread 2 use time=3     ☆☆☆☆☆thread 2 use time=2     ★★★★★thread 1 use time=0
★★★★★thread 1 use time=1     ★★★★★thread 1 use time=3     ☆☆☆☆☆thread 2 use time=0
☆☆☆☆☆thread 2 use time=0     ☆☆☆☆☆thread 2 use time=1     ☆☆☆☆☆thread 2 use time=2

★★★★★thread 1 use time=2     ★★★★★thread 1 use time=2     ★★★★★thread 1 use time=2
☆☆☆☆☆thread 2 use time=0     ★★★★★thread 1 use time=0     ★★★★★thread 1 use time=2
☆☆☆☆☆thread 2 use time=3     ★★★★★thread 1 use time=3     ★★★★★thread 1 use time=0
★★★★★thread 1 use time=2     ☆☆☆☆☆thread 2 use time=0     ☆☆☆☆☆thread 2 use time=2
★★★★★thread 1 use time=0     ★★★★★thread 1 use time=1     ☆☆☆☆☆thread 2 use time=3
☆☆☆☆☆thread 2 use time=0     ☆☆☆☆☆thread 2 use time=2     ★★★★★thread 1 use time=1
★★★★★thread 1 use time=0     ☆☆☆☆☆thread 2 use time=2     ★★★★★thread 1 use time=0
★★★★★thread 1 use time=3     ★★★★★thread 1 use time=1     ☆☆☆☆☆thread 2 use time=1
☆☆☆☆☆thread 2 use time=1     ☆☆☆☆☆thread 2 use time=0     ☆☆☆☆☆thread 2 use time=1
☆☆☆☆☆thread 2 use time=1     ☆☆☆☆☆thread 2 use time=0     ☆☆☆☆☆thread 2 use time=1
```

图 1-57　运行 6 次后的结果

那么，根据此实验可以得出一个结论，不要把线程的优先级与运行结果的顺序作为衡量的标准，优先级较高的线程并不一定每一次都先执行完 run() 方法中的任务，也就是说，线程优先级与打印顺序无关，不要将这两者的关系相关联，它们的关系具有不确定性和随机性。

1.10.4　看谁运行得快

创建实验用的项目 countPriority，创建两个线程类，代码如图 1-58 所示。

ThreadA.java

```
1  package extthread;
2
3  public class ThreadA extends Thread {
4
5      private int count = 0;
6
7      public int getCount() {
8          return count;
9      }
10
11     @Override
12     public void run() {
13         while (true) {
14             count++;
15         }
16     }
17
18 }
19
```

ThreadB.java

```
1  package extthread;
2
3  public class ThreadB extends Thread {
4
5      private int count = 0;
6
7      public int getCount() {
8          return count;
9      }
10
11     @Override
12     public void run() {
13         while (true) {
14             count++;
15         }
16     }
17
18 }
19
```

图 1-58　两个线程类代码

创建类 Run.java，代码如下：

```
package test;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        try {
            ThreadA a = new ThreadA();
            a.setPriority(Thread.NORM_PRIORITY - 3);
            a.start();
            ThreadB b = new ThreadB();
            b.setPriority(Thread.NORM_PRIORITY + 3);
            b.start();
            Thread.sleep(20000);
            a.stop();
            b.stop();
            System.out.println("a=" + a.getCount());
            System.out.println("b=" + b.getCount());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 1-59 所示。

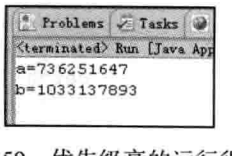


图 1-59　优先级高的运行得快

1.11 守护线程

在 Java 线程中有两种线程，一种是用户线程，另一种是守护线程。

守护线程是一种特殊的线程，它的特性有“陪伴”的含义，当进程中不存在非守护线程了，则守护线程自动销毁。典型的守护线程就是垃圾回收线程，当进程中没有非守护线程了，则垃圾回收线程也就没有存在的必要了，自动销毁。用个比较通俗的比喻来解释一下“守护线程”：任何一个守护线程都是整个 JVM 中所有非守护线程的“保姆”，只要当前 JVM 实例中存在任何一个非守护线程没有结束，守护线程就在工作，只有当最后一个非守护线程结束时，守护线程才随着 JVM 一同结束工作。Daemon 的作用是为其他线程的运行提供便利服务，守护线程最典型的应用就是 GC（垃圾回收器），它就是一个很称职的守护者。

创建项目 daemonThread，文件 MyThread.java 代码如下：

```
package testpackage;
public class MyThread extends Thread {
    private int i = 0;
    @Override
    public void run() {
        try {
            while (true) {
                i++;
                System.out.println("i=" + (i));
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

文件 Run.java 代码如下：

```
package test.run;
import testpackage.MyThread;
public class Run {
    public static void main(String[] args) {
        try {
            MyThread thread = new MyThread();
            thread.setDaemon(true);
            thread.start();
            Thread.sleep(5000);
            System.out.println("我离开 thread 对象也不再打印了，也就是停止了！ ");
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
```

```
        }
    }
}
```

程序运行后的效果如图 1-60 所示。

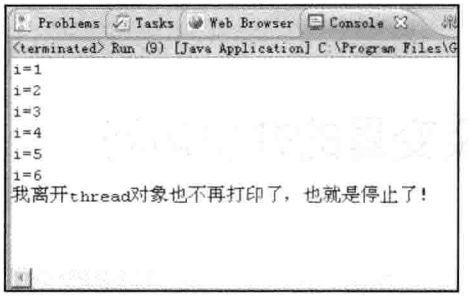


图 1-60　守护线程也退出了

1.12　本章小结

本章介绍了 Thread 类的 API，在使用这些 API 的过程中，会出现一些意想不到的情况，其实这也是多线程具有不可预知性的一个体现。学习和掌握这些常用情况，也就掌握了多线程开发的命脉与习性，是学习多线程更深层知识的基础。

Chapter 2 第 2 章

对象及变量的并发访问

本章主要介绍 Java 多线程中的同步，也就是如何在 Java 语言中写出线程安全的程序，如何在 Java 语言中解决非线程安全的相关问题。多线程中的同步问题是学习多线程的重中之重，这个技术在其他的编程语言中也涉及，如 C++ 或 C#。

本章应该着重掌握如下技术点：

- synchronized 对象监视器为 Object 时的使用。
- synchronized 对象监视器为 Class 时的使用。
- 非线程安全是如何出现的。
- 关键字 volatile 的主要作用。
- 关键字 volatile 与 synchronized 的区别及使用情况。

2.1 synchronized 同步方法

在第 1 章中已经接触“线程安全”与“非线程安全”相关的技术点，它们是学习多线程技术时一定会遇到的经典问题。“非线程安全”其实会在多个线程对同一个对象中的实例变量进行并发访问时发生，产生的后果就是“脏读”，也就是取到的数据其实是被更改过的。而“线程安全”就是以获得的实例变量的值是经过同步处理的，不会出现脏读的现象。此知识点在第 1 章也介绍，但本章将细化线程并发访问的内容，在细节上更多接触在并发时变量值的处理方法。

2.1.1　方法内的变量为线程安全

“非线程安全”问题存在于“实例变量”中，如果是方法内部的私有变量，则不存在“非线程安全”问题，所得结果也就是“线程安全”的了。

下面的示例项目就是实现方法内部声明一个变量时，是不存在“非线程安全”问题的。

创建 t1 项目，HasSelfPrivateNum.java 文件代码如下：

```
package service;
public class HasSelfPrivateNum {
    public void addI(String username) {
        try {
            int num = 0;
            if (username.equals("a")) {
                num = 100;
                System.out.println("a set over!");
                Thread.sleep(2000);
            } else {
                num = 200;
                System.out.println("b set over!");
            }
            System.out.println(username + " num=" + num);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

}
```

文件 ThreadA.java 代码如下：

```
package extthread;
import service.HasSelfPrivateNum;
public class ThreadA extends Thread {
    private HasSelfPrivateNum numRef;
    public ThreadA(HasSelfPrivateNum numRef) {
        super();
        this.numRef = numRef;
    }
    @Override
    public void run() {
        super.run();
        numRef.addI("a");
    }
}
```

文件 ThreadB.java 代码如下：

```
package extthread;
import service.HasSelfPrivateNum;
public class ThreadB extends Thread {
    private HasSelfPrivateNum numRef;
    public ThreadB(HasSelfPrivateNum numRef) {
        super();
        this.numRef = numRef;
    }
    @Override
    public void run() {
        super.run();
        numRef.addI("b");
    }
}
```

文件 Run.java 代码如下：

```
package test;
import service.HasSelfPrivateNum;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        HasSelfPrivateNum numRef = new HasSelfPrivateNum();
        ThreadA athread = new ThreadA(numRef);
        athread.start();
        ThreadB bthread = new ThreadB(numRef);
        bthread.start();
    }
}
```

程序运行后的效果如图 2-1 所示。

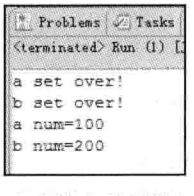


图 2-1 方法中的变量呈线程安全状态

可见，方法中的变量不存在非线程安全问题，永远都是线程安全的。这是方法内部的变量是私有的特性造成的。

2.1.2 实例变量非线程安全

如果多个线程共同访问 1 个对象中的实例变量，则有可能出现“非线程安全”问题。

用线程访问的对象中如果有多个实例变量，则运行的结果有可能出现交叉的情况。此情况在第 1 章中非线程安全的案例演示过。

如果对象仅有 1 个实例变量，则有可能出现覆盖的情况。

创建 t2 项目，HasSelfPrivateNum.java 文件代码如下：

```
package service;
public class HasSelfPrivateNum {
    private int num = 0;
    public void addI(String username) {
        try {
            if (username.equals("a")) {
                num = 100;
                System.out.println("a set over!");
                Thread.sleep(2000);
            } else {
                num = 200;
                System.out.println("b set over!");
            }
            System.out.println(username + " num=" + num);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

}
```

文件 ThreadA.java 代码如下：

```
package extthread;
import service.HasSelfPrivateNum;
public class ThreadA extends Thread {
    private HasSelfPrivateNum numRef;
    public ThreadA(HasSelfPrivateNum numRef) {
        super();
        this.numRef = numRef;
    }
    @Override
    public void run() {
        super.run();
        numRef.addI("a");
    }
}
```

文件 ThreadB.java 代码如下：

```
package extthread;
import service.HasSelfPrivateNum;
public class ThreadB extends Thread {
    private HasSelfPrivateNum numRef;
    public ThreadB(HasSelfPrivateNum numRef) {
        super();
```

```
        this.numRef = numRef;
    }
    @Override
    public void run() {
        super.run();
        numRef.addI("b");
    }
}
```

文件 Run.java 代码如下：

```
package test;
import service.HasSelfPrivateNum;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        HasSelfPrivateNum numRef = new HasSelfPrivateNum();
        ThreadA athread = new ThreadA(numRef);
        athread.start();
        ThreadB bthread = new ThreadB(numRef);
        bthread.start();
    }
}
```

程序运行后的结果如图 2-2 所示。

图 2-2 单例模式中的实例变量呈非线程安全状态

本实验是两个线程同时访问一个没有同步的方法，如果两个线程同时操作业务对象中的实例变量，则有可能会出现“非线程安全”问题。此示例的知识点在前面已经介绍过，只需要在 public void addI(String username) 方法前加关键字 synchronized 即可。更改后的代码如下：

```
package service;
public class HasSelfPrivateNum {
    private int num = 0;
    synchronized public void addI(String username) {
        try {
            if (username.equals("a")) {
                num = 100;
                System.out.println("a set over!");
                Thread.sleep(2000);
            } else {
```

```
                num = 200;
                System.out.println("b set over!");
            }
            System.out.println(username + " num=" + num);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

程序再次运行结果如图 2-3 所示。

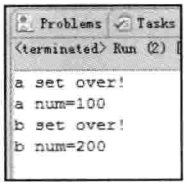


图 2-3　同步后线程安全了

实验结论：在两个线程访问同一个对象中的同步方法时一定是线程安全的。本实验由于是同步访问，所以先打印出 a，然后打印出 b。

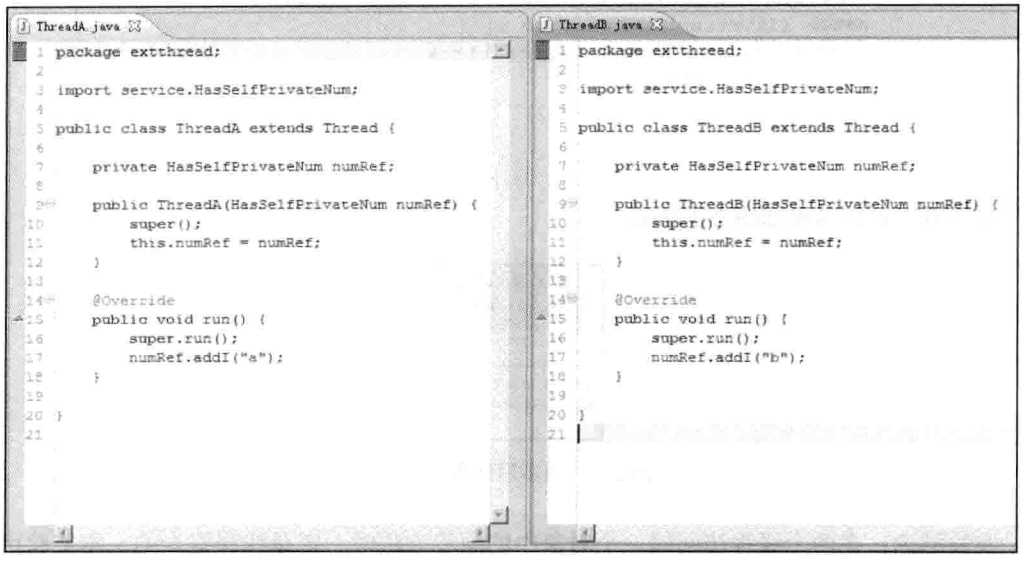
2.1.3　多个对象多个锁

再来看一个实验，创建项目名称为 twoObjectTwoLock，创建 HasSelfPrivateNum.java 类，代码如下：

```
package service;
public class HasSelfPrivateNum {
    private int num = 0;
    synchronized public void addI(String username) {
        try {
            if (username.equals("a")) {
                num = 100;
                System.out.println("a set over!");
                Thread.sleep(2000);
            } else {
                num = 200;
                System.out.println("b set over!");
            }
            System.out.println(username + " num=" + num);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

上面的代码中有同步方法 addI，说明此方法应该被顺序调用。

创建线程 ThreadA.java 和 ThreadB.java 代码，如图 2-4 所示。



ThreadA.java

```
1  package extthread;
2
3  import service.HasSelfPrivateNum;
4
5  public class ThreadA extends Thread {
6
7      private HasSelfPrivateNum numRef;
8
9      public ThreadA(HasSelfPrivateNum numRef) {
10         super();
11         this.numRef = numRef;
12     }
13
14     @Override
15     public void run() {
16         super.run();
17         numRef.addI("a");
18     }
19
20 }
21
```

ThreadB.java

```
1  package extthread;
2
3  import service.HasSelfPrivateNum;
4
5  public class ThreadB extends Thread {
6
7      private HasSelfPrivateNum numRef;
8
9      public ThreadB(HasSelfPrivateNum numRef) {
10         super();
11         this.numRef = numRef;
12     }
13
14     @Override
15     public void run() {
16         super.run();
17         numRef.addI("b");
18     }
19
20 }
21
```

图 2-4　两个线程类代码

类 Run.java 代码如下：

```
package test;
import service.HasSelfPrivateNum;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        HasSelfPrivateNum numRef1 = new HasSelfPrivateNum();
        HasSelfPrivateNum numRef2 = new HasSelfPrivateNum();
        ThreadA athread = new ThreadA(numRef1);
        athread.start();
        ThreadB bthread = new ThreadB(numRef2);
        bthread.start();
    }
}
```

创建了 2 个 HasSelfPrivateNum.java 类的对象，程序运行的结果如图 2-5 所示。

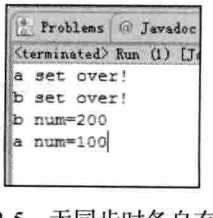


图 2-5　无同步时各自有锁

上面示例是两个线程分别访问同一个类的两个不同实例的相同名称的同步方法，效果却是以异步的方式运行的。本示例由于创建了 2 个业务对象，在系统中产生出 2 个锁，所以运行结果是

异步的，打印的效果就是先打印 b，然后打印 a。

从上面程序运行结果来看，虽然在 HasSelfPrivateNum.java 中使用了 synchronized 关键字，但打印的顺序却不是同步的，是交叉的。为什么是这样的结果呢？

关键字 synchronized 取得的锁都是对象锁，而不是把一段代码或方法（函数）当作锁，所以在上面的示例中，哪个线程先执行带 synchronized 关键字的方法，哪个线程就持有该方法所属对象的锁 Lock，那么其他线程只能呈等待状态，前提是多个线程访问的是同一个对象。

但如果多个线程访问多个对象，则 JVM 会创建多个锁。上面的示例就是创建了 2 个 HasSelfPrivateNum.java 类的对象，所以就会产生出 2 个锁。

同步的单词为 synchronized，异步的单词为 asynchronized。

2.1.4　synchronized 方法与锁对象

为了证明前面讲述线程锁的是对象，创建实验用的项目 synchronizedMethodLockObject，类 MyObject.java 文件代码如下：

```
package extobject;
public class MyObject {
    public void methodA() {
        try {
            System.out.println("begin methodA threadName="
                    + Thread.currentThread().getName());
            Thread.sleep(5000);
            System.out.println("end");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

自定义线程类 ThreadA.java 代码如下：

```
package extthread;
import extobject.MyObject;
public class ThreadA extends Thread {
    private MyObject object;
    public ThreadA(MyObject object) {
        super();
        this.object = object;
    }
    @Override
    public void run() {
        super.run();
```

```
        object.methodA();
    }
}
```

自定义线程类 ThreadB.java 代码如下：

```
package extthread;
import extobject.MyObject;
public class ThreadB extends Thread {
    private MyObject object;
    public ThreadB(MyObject object) {
        super();
        this.object = object;
    }
    @Override
    public void run() {
        super.run();
        object.methodA();
    }
}
```

运行类 Run.java 代码如下：

```
package test.run;
import extobject.MyObject;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        MyObject object = new MyObject();
        ThreadA a = new ThreadA(object);
        a.setName("A");
        ThreadB b = new ThreadB(object);
        b.setName("B");
        a.start();
        b.start();
    }
}
```

程序运行后的效果如图 2-6 所示。

更改 MyObject.java 代码如下：

```
package extobject;
public class MyObject {
    synchronized public void methodA() {
        try {
            System.out.println("begin methodA threadName="
                    + Thread.currentThread().getName());
            Thread.sleep(5000);
            System.out.println("end");
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

如上面代码所示，在 methodA 方法前加入了关键字 synchronized 进行同步处理。程序再次运行效果如图 2-7 所示。

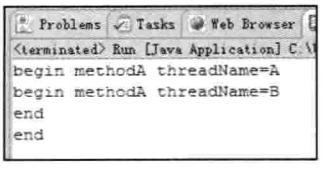


图 2-6　两个线程可一同进入 methodA 方法

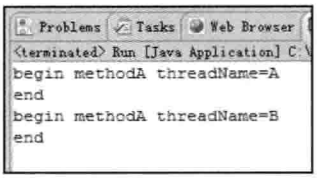


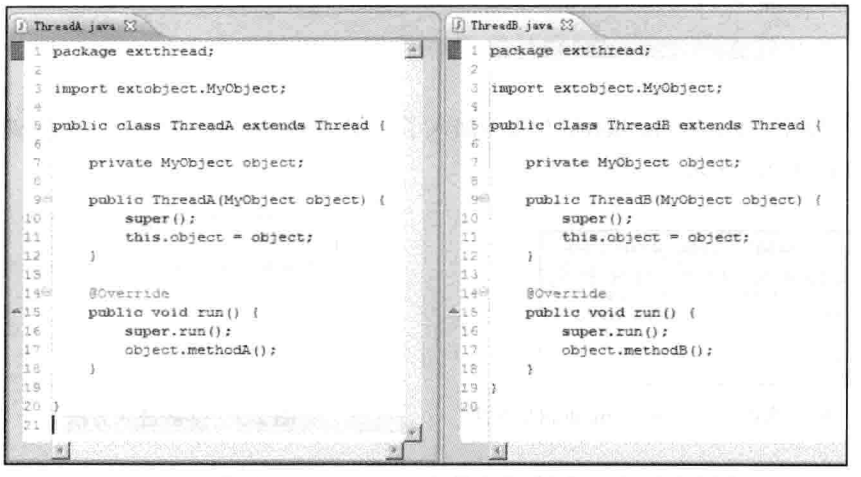
图 2-7　排队进入方法

通过上面的实验得到结论，调用用关键字 synchronized 声明的方法一定是排队运行的。另外需要牢牢记住“共享”这两个字，只有共享资源的读写访问才需要同步化，如果不是共享资源，那么根本就没有同步的必要。

那其他的方法在被调用时会是什么效果呢？如何查看到 Lock 锁对象的效果呢？继续新建实验用的项目 synchronizedMethodLockObject2，类文件 MyObject.java 代码如下：

```
package extobject;
public class MyObject {
    synchronized public void methodA() {
        try {
            System.out.println("begin methodA threadName="
                    + Thread.currentThread().getName());
            Thread.sleep(5000);
            System.out.println("end endTime=" + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public void methodB() {
        try {
            System.out.println("begin methodB threadName="
                    + Thread.currentThread().getName() + " begin time="
                    + System.currentTimeMillis());
            Thread.sleep(5000);
            System.out.println("end");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

两个自定义线程类分别调用不同的方法，代码如图 2-8 所示。



ThreadA.java

```
1  package extthread;
2
3  import extobject.MyObject;
4
5  public class ThreadA extends Thread {
6
7      private MyObject object;
8
9      public ThreadA(MyObject object) {
10         super();
11         this.object = object;
12     }
13
14     @Override
15     public void run() {
16         super.run();
17         object.methodA();
18     }
19
20 }
21
```

ThreadB.java

```
1  package extthread;
2
3  import extobject.MyObject;
4
5  public class ThreadB extends Thread {
6
7      private MyObject object;
8
9      public ThreadB(MyObject object) {
10         super();
11         this.object = object;
12     }
13
14     @Override
15     public void run() {
16         super.run();
17         object.methodB();
18     }
19 }
20
```

图 2-8　调用不同方法的线程类

文件 Run.java 代码如下：

```
package test.run;
import extobject.MyObject;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        MyObject object = new MyObject();
        ThreadA a = new ThreadA(object);
        a.setName("A");
        ThreadB b = new ThreadB(object);
        b.setName("B");
        a.start();
        b.start();
    }
}
```

程序运行结果如图 2-9 所示。

通过上面的实验可以得知，虽然线程 A 先持有了 object 对象的锁，但线程 B 完全可以异步调用非 synchronized 类型的方法。

继续实验，将 MyObject.java 文件中的 methodB() 方法前加上 synchronized 关键字，代码如下：

```
    synchronized public void methodB() {
        try {
            System.out.println("begin methodB threadName="
                    + Thread.currentThread().getName() + " begin time="
                    + System.currentTimeMillis());
            Thread.sleep(5000);
            System.out.println("end");
```

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

本示例是两个线程访问同一个对象的两个同步的方法，运行结果如图 2-10 所示。

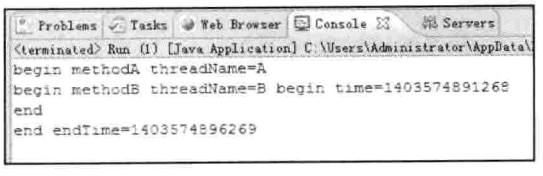


图 2-9　线程 B 异步调用非同步方法

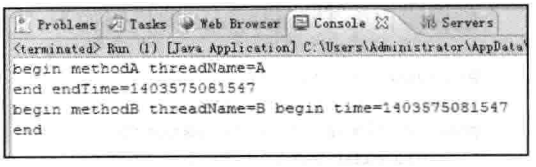


图 2-10　同步运行

此实验的结论是：

1）A 线程先持有 object 对象的 Lock 锁，B 线程可以以异步的方式调用 object 对象中的非 synchronized 类型的方法。

2）A 线程先持有 object 对象的 Lock 锁，B 线程如果在这时调用 object 对象中的 synchronized 类型的方法则需等待，也就是同步。

2.1.5　脏读

在 2.1.4 节示例中已经实现多个线程调用同一个方法时，为了避免数据出现交叉的情况，使用 synchronized 关键字来进行同步。

虽然在赋值时进行了同步，但在取值时有可能出现一些意想不到的意外，这种情况就是脏读（dirtyRead）。发生脏读的情况是在读取实例变量时，此值已经被其他线程更改过了。

创建 t3 项目，PublicVar.java 文件代码如下：

```
package entity;
public class PublicVar {
    public String username = "A";
    public String password = "AA";
    synchronized public void setValue(String username, String password) {
        try {
            this.username = username;
            Thread.sleep(5000);
            this.password = password;
            System.out.println("setValue method thread name="
                    + Thread.currentThread().getName() + " username="
                    + username + " password=" + password);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public void getValue() {
```

```
        System.out.println("getValue method thread name="
                + Thread.currentThread().getName() + " username=" + username
                + " password=" + password);
    }
}
```

同步方法 setValue() 的锁属于类 PublicVar 的实例。

创建线程类 ThreadA.java 的代码如下：

```
package extthread;
import entity.PublicVar;
public class ThreadA extends Thread {
    private PublicVar publicVar;
    public ThreadA(PublicVar publicVar) {
        super();
        this.publicVar = publicVar;
    }
    @Override
    public void run() {
        super.run();
        publicVar.setValue("B", "BB");
    }
}
```

文件 Test.java 代码如下：

```
package test;
import entity.PublicVar;
import extthread.ThreadA;
public class Test {
    public static void main(String[] args) {
        try {
            PublicVar publicVarRef = new PublicVar();
            ThreadA thread = new ThreadA(publicVarRef);
            thread.start();
            Thread.sleep(200);//打印结果受此值大小影响
            publicVarRef.getValue();
        } catch (InterruptedException e) {
            //TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

程序运行后的结果如图 2-11 所示。

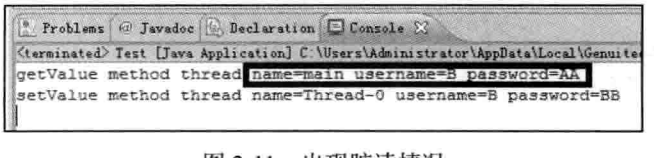


图 2-11 出现脏读情况

出现脏读是因为 public void getValue() 方法并不是同步的，所以可以在任意时候进行调用。解决办法当然就是加上同步 synchronized 关键字，代码如下：

```
synchronized public void getValue() {
    System.out.println("getValue method thread name="
            + Thread.currentThread().getName() + " username=" + username
            + " password=" + password);
}
```

程序运行后的结果如图 2-12 所示。

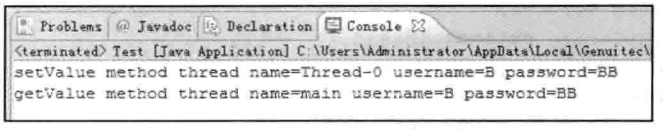


图 2-12　不再出现脏读了

可见，方法 setValue() 和 getValue() 被依次执行。通过这个案例不仅要知道脏读是通过 synchronized 关键字解决的，还要知道如下内容：

当 A 线程调用 anyObject 对象加入 synchronized 关键字的 X 方法时，A 线程就获得了 X 方法锁，更准确地讲，是获得了对象的锁，所以其他线程必须等 A 线程执行完毕才可以调用 X 方法，但 B 线程可以随意调用其他的非 synchronized 同步方法。

当 A 线程调用 anyObject 对象加入 synchronized 关键字的 X 方法时，A 线程就获得了 X 方法所在对象的锁，所以其他线程必须等 A 线程执行完毕才可以调用 X 方法，而 B 线程如果调用声明了 synchronized 关键字的非 X 方法时，必须等 A 线程将 X 方法执行完，也就是释放对象锁后才可以调用。这时 A 线程已经执行了一个完整的任务，也就是说 username 和 password 这两个实例变量已经同时被赋值，不存在脏读的基本环境。

脏读一定会出现操作实例变量的情况下，这就是不同线程“争抢”实例变量的结果。

2.1.6　synchronized 锁重入

关键字 synchronized 拥有锁重入的功能，也就是在使用 synchronized 时，当一个线程得到一个对象锁后，再次请求此对象锁时是可以再次得到该对象的锁的。这也证明在一个 synchronized 方法 / 块的内部调用本类的其他 synchronized 方法 / 块时，是永远可以得到锁的。

创建实验用的项目 synLockIn_1，类 Service.java 代码如下：

```
package myservice;
public class Service {
    synchronized public void service1() {
        System.out.println("service1");
        service2();
    }
    synchronized public void service2() {
```

```
        System.out.println("service2");
        service3();
    }
    synchronized public void service3() {
        System.out.println("service3");
    }
}
```

线程类 MyThread.java 代码如下：

```
package extthread;
import myservice.Service;
public class MyThread extends Thread {
    @Override
    public void run() {
        Service service = new Service();
        service.service1();
    }
}
```

运行类 Run.java 代码如下：

```
package test;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
    }
}
```

程序运行结果如图 2-13 所示。

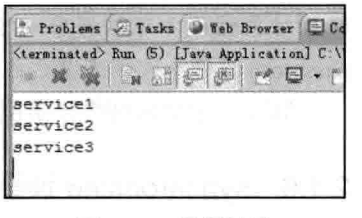


图 2-13 运行结果

“可重入锁”的概念是：自己可以再次获取自己的内部锁。比如有 1 条线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。

可重入锁也支持在父子类继承的环境中。

创建实验用的项目 synLockIn_2，类 Main.java 代码如下：

```
package myservice;
public class Main {
    public int i = 10;
    synchronized public void operateIMainMethod() {
        try {
            i--;
            System.out.println("main print i=" + i);
            Thread.sleep(100);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
```

```
            e.printStackTrace();
        }
    }
}
```

子类 Sub.java 代码如下：

```
package myservice;
public class Sub extends Main {
    synchronized public void operateISubMethod() {
        try {
            while (i > 0) {
                i--;
                System.out.println("sub print i=" + i);
                Thread.sleep(100);
                this.operateIMainMethod();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

自定义线程类 MyThread.java 代码如下：

```
package extthread;
import myservice.Main;
import myservice.Sub;
public class MyThread extends Thread {
    @Override
    public void run() {
        Sub sub = new Sub();
        sub.operateISubMethod();
    }
}
```

运行类 Run.java 代码如下：

```
package test;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
    }
}
```

程序运行后的效果如图 2-14 所示。

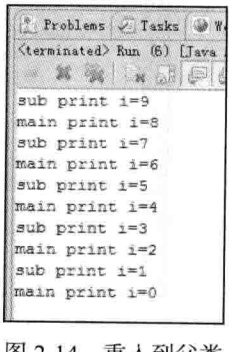


图 2-14　重入到父类中的锁

此实验说明，当存在父子类继承关系时，子类是完全可以通过“可重入锁”调用父类的同步方法的。

2.1.7 出现异常，锁自动释放

当一个线程执行的代码出现异常时，其所持有的锁会自动释放。

创建实验用的项目 throwExceptionNoLock，类 Service.java 代码如下：

```
package service;
public class Service {
    synchronized public void testMethod() {
        if (Thread.currentThread().getName().equals("a")) {
            System.out.println("ThreadName=" + Thread.currentThread().getName()
                    + " run beginTime=" + System.currentTimeMillis());
            int i = 1;
            while (i == 1) {
                if (("" + Math.random()).substring(0, 8).equals("0.123456")) {
                    System.out.println("ThreadName="
                            + Thread.currentThread().getName()
                            + " run   exceptionTime="
                            + System.currentTimeMillis());
                    Integer.parseInt("a");
                }
            }
        } else {
            System.out.println("Thread B run Time="
                    + System.currentTimeMillis());
        }
    }
}
```

两个自定义线程代码如图 2-15 所示。

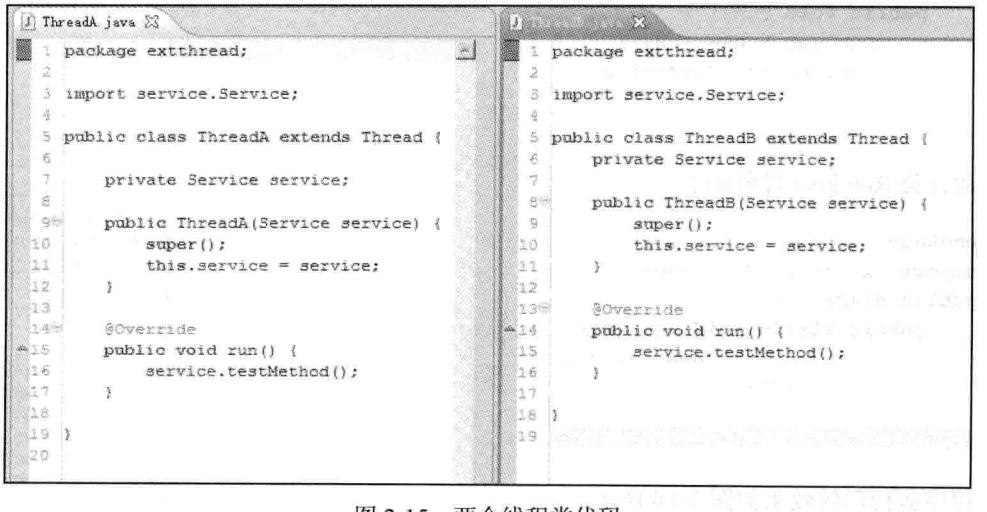


图 2-15 两个线程类代码

运行类 Run.java 代码如下：

```
package controller;
import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;
public class Test {
    public static void main(String[] args) {
        try {
            Service service = new Service();
            ThreadA a = new ThreadA(service);
            a.setName("a");
            a.start();
            Thread.sleep(500);
            ThreadB b = new ThreadB(service);
            b.setName("b");
            b.start();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行后的效果如图 2-16 所示。

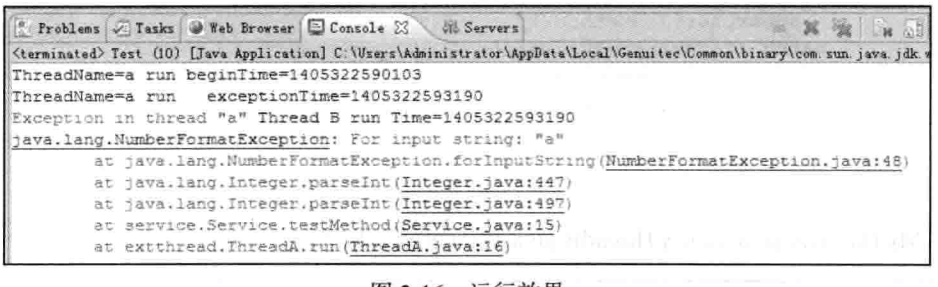


图 2-16　运行效果

线程 a 出现异常并释放锁，线程 b 进入方法正常打印，实验的结论就是出现异常的锁被自动释放了。

2.1.8　同步不具有继承性

同步不可以继承。

创建测试用的项目 synNotExtends，类 Main.java 代码如下：

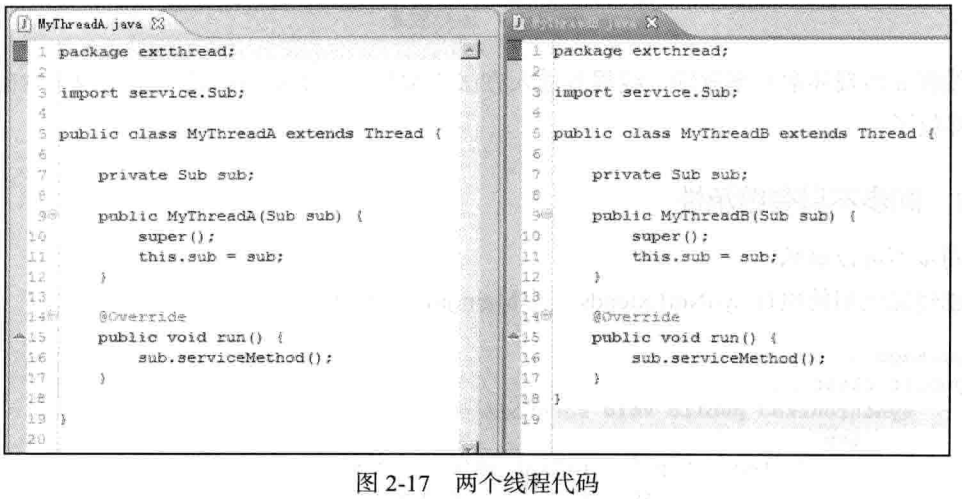
```
package service;
public class Main {
    synchronized public void serviceMethod() {
        try {
            System.out.println("int main 下一步 sleep begin threadName="
                    + Thread.currentThread().getName() + " time="
                    + System.currentTimeMillis());
            Thread.sleep(5000);
```

```
            System.out.println("int main 下一步 sleep    end threadName="
                    + Thread.currentThread().getName() + " time="
                    + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

类 Sub.java 代码如下：

```
package service;
public class Sub extends Main {
    @Override
    public void serviceMethod() {
        try {
            System.out.println("int sub 下一步 sleep begin threadName="
                    + Thread.currentThread().getName() + " time="
                    + System.currentTimeMillis());
            Thread.sleep(5000);
            System.out.println("int sub 下一步 sleep    end threadName="
                    + Thread.currentThread().getName() + " time="
                    + System.currentTimeMillis());
            super.serviceMethod();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

类 MyThreadA.java 和 MyThreadB.java 代码如图 2-17 所示。



```
MyThreadA.java
1  package extthread;
2
3  import service.Sub;
4
5  public class MyThreadA extends Thread {
6
7      private Sub sub;
8
9      public MyThreadA(Sub sub) {
10         super();
11         this.sub = sub;
12     }
13
14     @Override
15     public void run() {
16         sub.serviceMethod();
17     }
18
19 }
20
```

```
1  package extthread;
2
3  import service.Sub;
4
5  public class MyThreadB extends Thread {
6
7      private Sub sub;
8
9      public MyThreadB(Sub sub) {
10         super();
11         this.sub = sub;
12     }
13
14     @Override
15     public void run() {
16         sub.serviceMethod();
17     }
18 }
19
```

图 2-17　两个线程代码

类 Test.java 代码如下：

```
package controller;
import service.Sub;
import extthread.MyThreadA;
import extthread.MyThreadB;
public class Test {
    public static void main(String[] args) {
        Sub subRef = new Sub();
        MyThreadA a = new MyThreadA(subRef);
        a.setName("A");
        a.start();
        MyThreadB b = new MyThreadB(subRef);
        b.setName("B");
        b.start();
    }
}
```

程序运行后的效果如图 2-18 所示。

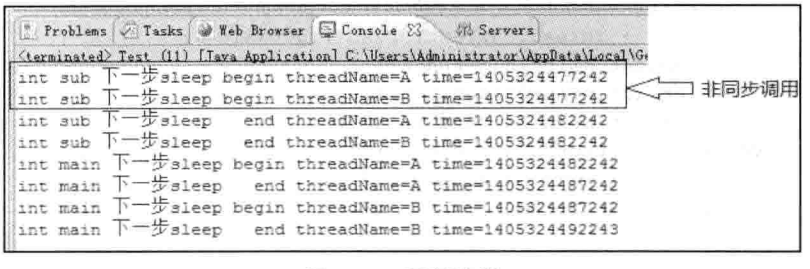


图 2-18　运行效果

从此示例可以看到，同步不能继承，所以还得在子类的方法中添加 synchronized 关键字，添加后的运行效果如图 2-19 所示。

图 2-19　同步了

2.2　synchronized 同步语句块

用关键字 synchronized 声明方法在某些情况下是有弊端的，比如 A 线程调用同步方法执行一个长时间的任务，那么 B 线程则必须等待比较长时间。在这样的情况下可以使用 synchronized 同步语句块来解决。

2.2.1 synchronized 方法的弊端

为了证明用 synchronized 关键字声明方法是有弊端的，创建 t5 项目，来进行测试。

文件 Task.java 代码如下：

```
package mytask;
import commonutils.CommonUtils;
public class Task {
    private String getData1;
    private String getData2;
    public synchronized void doLongTimeTask() {
        try {
            System.out.println("begin task");
            Thread.sleep(3000);
            getData1 = "长时间处理任务后从远程返回的值 1 threadName="
                    + Thread.currentThread().getName();
            getData2 = "长时间处理任务后从远程返回的值 2 threadName="
                    + Thread.currentThread().getName();
            System.out.println(getData1);
            System.out.println(getData2);
            System.out.println("end task");
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

文件 CommonUtils.java 代码如下：

```
package commonutils;
public class CommonUtils {
    public static long beginTime1;
    public static long endTime1;
    public static long beginTime2;
    public static long endTime2;
}
```

文件 MyThread1.java 代码如下：

```
package mythread;
import commonutils.CommonUtils;
import mytask.Task;
public class MyThread1 extends Thread {
    private Task task;
    public MyThread1(Task task) {
        super();
        this.task = task;
    }
    @Override
```

```
    public void run() {
        super.run();
        CommonUtils.beginTime1 = System.currentTimeMillis();
        task.doLongTimeTask();
        CommonUtils.endTime1 = System.currentTimeMillis();
    }
}
```

文件 MyThread2.java 代码如下：

```
package mythread;
import commonutils.CommonUtils;
import mytask.Task;
public class MyThread2 extends Thread {
    private Task task;
    public MyThread2(Task task) {
        super();
        this.task = task;
    }
    @Override
    public void run() {
        super.run();
        CommonUtils.beginTime2 = System.currentTimeMillis();
        task.doLongTimeTask();
        CommonUtils.endTime2 = System.currentTimeMillis();
    }
}
```

文件 Run.java 代码如下：

```
package test;
import mytask.Task;
import mythread.MyThread1;
import mythread.MyThread2;
import commonutils.CommonUtils;
public class Run {
    public static void main(String[] args) {
        Task task = new Task();
        MyThread1 thread1 = new MyThread1(task);
        thread1.start();
        MyThread2 thread2 = new MyThread2(task);
        thread2.start();
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        long beginTime = CommonUtils.beginTime1;
        if (CommonUtils.beginTime2 < CommonUtils.beginTime1) {
            beginTime = CommonUtils.beginTime2;
```

```
        }
        long endTime = CommonUtils.endTime1;
        if (CommonUtils.endTime2 > CommonUtils.endTime1) {
            endTime = CommonUtils.endTime2;
        }
        System.out.println("耗时: " + ((endTime - beginTime) / 1000));
    }
}
```

程序运行后的结果如图 2-20 所示。大约 6 秒钟后程序结束运行。

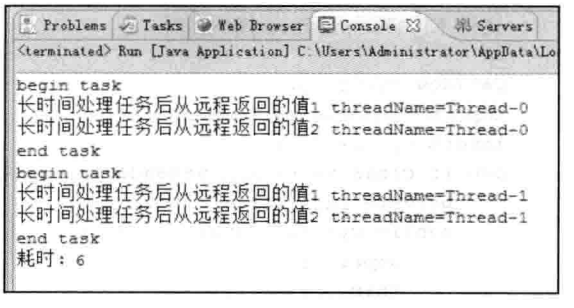


图 2-20 运行结果

在使用 synchronized 关键字来声明方法 public synchronized void doLongTimeTask() 时从运行的时间上来看，弊端很明显，解决这样的问题可以使用 synchronized 同步块。

2.2.2 synchronized 同步代码块的使用

当两个并发线程访问同一个对象 object 中的 synchronized(this) 同步代码块时，一段时间内只能有一个线程被执行，另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。

创建测试用的项目 synchronizedOneThreadIn，类文件 ObjectService.java 代码如下：

```
package service;
public class ObjectService {
    public void serviceMethod() {
        try {
            synchronized (this) {
                System.out.println("begin time=" + System.currentTimeMillis());
                Thread.sleep(2000);
                System.out.println("end    end=" + System.currentTimeMillis());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

自定义线程 ThreadA.java 代码如下：

```
package extthread;
import service.ObjectService;
public class ThreadA extends Thread {
    private ObjectService service;
    public ThreadA(ObjectService service) {
        super();
```

```
        this.service = service;
    }
    @Override
    public void run() {
        super.run();
        service.serviceMethod();
    }
}
```

自定义线程 ThreadB.java 代码如下：

```
package extthread;
import service.ObjectService;
public class ThreadB extends Thread {
    private ObjectService service;
    public ThreadB(ObjectService service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        super.run();
        service.serviceMethod();
    }
}
```

运行类 Run.java 代码如下：

```
package test.run;
import service.ObjectService;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        ObjectService service = new ObjectService();
        ThreadA a = new ThreadA(service);
        a.setName("a");
        a.start();
        ThreadB b = new ThreadB(service);
        b.setName("b");
        b.start();
    }
}
```

运行结果如图 2-21 所示。

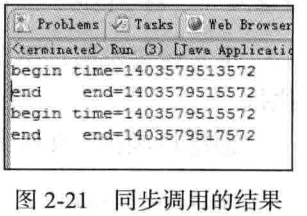


图 2-21　同步调用的结果

上面的实验中虽然使用了 synchronized 同步代码块，但执行的效率还是没有提高，执行的效果还是同步运行的。

如何用 synchronized 同步代码块解决程序执行效率低的问题呢?

2.2.3 用同步代码块解决同步方法的弊端

创建 t6 项目，将 t5 项目中的所有文件复制到 t6 项目中，并更改文件 Task.java 代码如下：

```
package mytask;
public class Task {
    private String getData1;
    private String getData2;
    public void doLongTimeTask() {
        try {
            System.out.println("begin task");
            Thread.sleep(3000);
            String privateGetData1 = "长时间处理任务后从远程返回的值 1 threadName="
                    + Thread.currentThread().getName();
            String privateGetData2 = "长时间处理任务后从远程返回的值 2 threadName="
                    + Thread.currentThread().getName();
            synchronized (this) {
                getData1 = privateGetData1;
                getData2 = privateGetData2;
            }
            System.out.println(getData1);
            System.out.println(getData2);
            System.out.println("end task");
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

程序运行后的效果如图 2-22 所示。

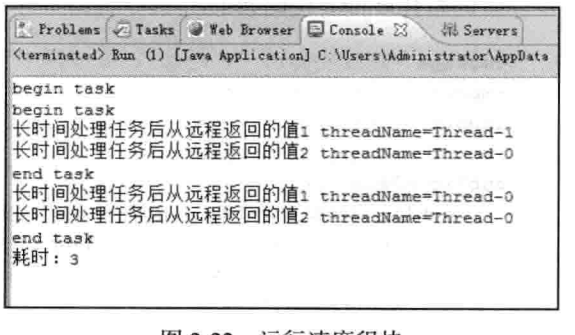


图 2-22 运行速度很快

通过上面的实验可以得知，当一个线程访问 object 的一个 synchronized 同步代码块时，另一个线程仍然可以访问该 object 对象中的非 synchronized(this) 同步代码块。

实验进行到这里，虽然时间缩短，运行效率加快，但同步 synchronized 代码块真的是同步的吗？真的持有当前调用对象的锁吗？答案为是，但必须用代码的方式来进行验证。

2.2.4 一半异步，一半同步

本实验要说明：不在 synchronized 块中就是异步执行，在 synchronized 块中就是同步执行。

创建 t7 项目，文件 Task.java 代码如下：

```
package mytask;
```

```
public class Task {
    public void doLongTimeTask() {
        for (int i = 0; i < 100; i++) {
            System.out.println("nosynchronized threadName="
                    + Thread.currentThread().getName() + " i=" + (i + 1));
        }
        System.out.println("");
        synchronized (this) {
            for (int i = 0; i < 100; i++) {
                System.out.println("synchronized threadName="
                   + Thread.currentThread().getName() + " i=" + (i + 1));
            }
        }
    }
}
```

两个线程类代码如图 2-23 所示。

MyThread1.java

```
package mythread;

import mytask.Task;

public class MyThread1 extends Thread {

    private Task task;

    public MyThread1(Task task) {
        super();
        this.task = task;
    }

    @Override
    public void run() {
        super.run();
        task.doLongTimeTask();
    }

}
```

MyThread2.java

```
package mythread;

import mytask.Task;

public class MyThread2 extends Thread {

    private Task task;

    public MyThread2(Task task) {
        super();
        this.task = task;
    }

    @Override
    public void run() {
        super.run();
        task.doLongTimeTask();
    }

}
```

图 2-23　两个线程代码

文件 Run.java 代码如下：

```
package test;
import mytask.Task;
import mythread.MyThread1;
import mythread.MyThread2;
public class Run {
    public static void main(String[] args) {
        Task task = new Task();
        MyThread1 thread1 = new MyThread1(task);
        thread1.start();
        MyThread2 thread2 = new MyThread2(task);
        thread2.start();
    }
}
```

程序运行后的结果如图 2-24 所示。

进入 synchronized 代码块后则排队执行，结果如图 2-25 所示。

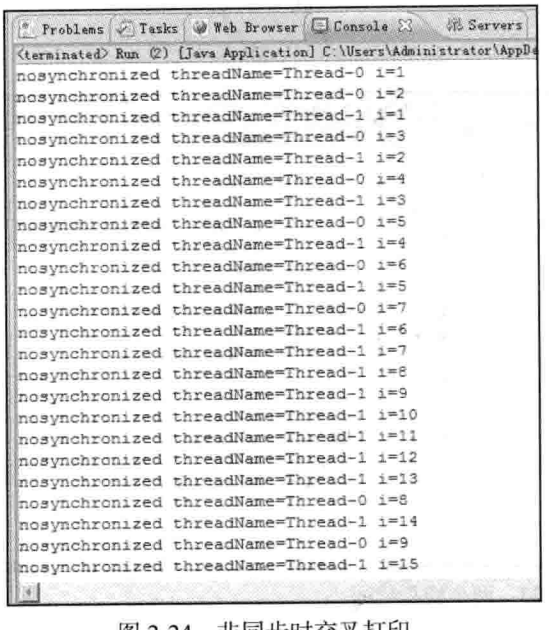


图 2-24 非同步时交叉打印

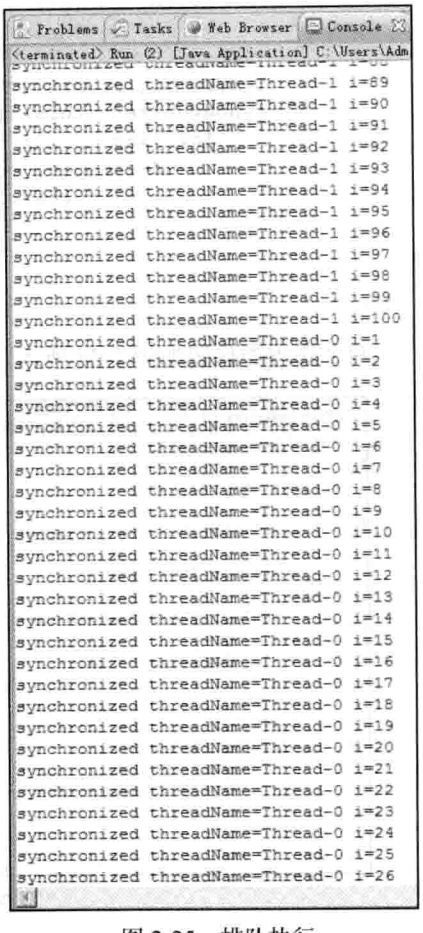


图 2-25 排队执行

2.2.5 synchronized 代码块间的同步性

在使用同步 synchronized(this) 代码块时需要注意的是，当一个线程访问 object 的一个 synchronized(this) 同步代码块时，其他线程对同一个 object 中所有其他 synchronized(this) 同步代码块的访问将被阻塞，这说明 synchronized 使用的“对象监视器”是一个。

创建用于验证的项目 doubleSynBlockOneTwo，类文件 ObjectService.java 代码如下：

```
package service;
public class ObjectService {
    public void serviceMethodA() {
        try {
```

```
            synchronized (this) {
                System.out.println("A begin time=" + System.currentTimeMillis());
                Thread.sleep(2000);
                System.out.println("A end    end=" + System.currentTimeMillis());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public void serviceMethodB() {
        synchronized (this) {
            System.out.println("B begin time=" + System.currentTimeMillis());
            System.out.println("B end    end=" + System.currentTimeMillis());
        }
    }
}
```

自定义线程类 ThreadA.java 代码如下：

```
package extthread;
import service.ObjectService;
public class ThreadA extends Thread {
    private ObjectService service;
    public ThreadA(ObjectService service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        super.run();
        service.serviceMethodA();
    }
}
```

自定义线程类 ThreadB.java 代码如下：

```
package extthread;
import service.ObjectService;
public class ThreadB extends Thread {
    private ObjectService service;
    public ThreadB(ObjectService service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        super.run();
        service.serviceMethodB();
    }
}
```

运行类 Run.java 代码如下：

```
package test.run;
```

```
import service.ObjectService;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        ObjectService service = new ObjectService();
        ThreadA a = new ThreadA(service);
        a.setName("a");
        a.start();
        ThreadB b = new ThreadB(service);
        b.setName("b");
        b.start();
    }
}
```

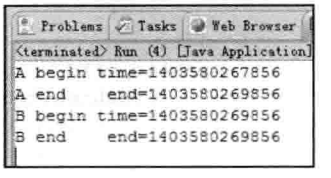


图 2-26 两个同步代码块按顺序执行

程序运行结果如图 2-26 所示。

2.2.6 验证同步 synchronized(this) 代码块是锁定当前对象的

和 synchronized 方法一样，synchronized(this) 代码块也是锁定当前对象的。

创建 t8 项目，文件 Task.java 代码如下：

```
package mytask;
public class Task {
    public void otherMethod() {
    System.out.println("------------------------run--otherMethod");
    }
    public void doLongTimeTask() {
        synchronized (this) {
            for (int i = 0; i < 10000; i++) {
                System.out.println("synchronized threadName="
                  + Thread.currentThread().getName() + " i=" + (i + 1));
            }
        }
    }
}
```

文件 MyThread1.java 代码如下：

```
package mythread;
import mytask.Task;
public class MyThread1 extends Thread {
    private Task task;
    public MyThread1(Task task) {
        super();
        this.task = task;
    }
    @Override
    public void run() {
        super.run();
        task.doLongTimeTask();
```

```
    }
}
```

文件 MyThread2.java 代码如下：

```
package mythread;
import mytask.Task;
public class MyThread2 extends Thread {
    private Task task;
    public MyThread2(Task task) {
        super();
        this.task = task;
    }
    @Override
    public void run() {
        super.run();
        task.otherMethod();
    }
}
```

文件 Run.java 代码如下：

```
package test;
import mytask.Task;
import mythread.MyThread1;
import mythread.MyThread2;
public class Run {
    public static void main(String[] args) throws InterruptedException {
        Task task = new Task();
        MyThread1 thread1 = new MyThread1(task);
        thread1.start();
        Thread.sleep(100);
        MyThread2 thread2 = new MyThread2(task);
        thread2.start();
    }
}
```

程序运行后的效果如图 2-27 所示。

更改 Task.java 代码如下：

```
package mytask;
public class Task {
    synchronized public void otherMethod() {
    System.out.println("-----------------------run--otherMethod");
    }
    public void doLongTimeTask() {
        synchronized (this) {
            for (int i = 0; i < 10000; i++) {
                System.out.println("synchronized threadName="
                        + Thread.currentThread().getName() + " i=" + (i + 1));
            }
```

```
        }
    }
}
```

程序运行效果如图 2-28 所示。

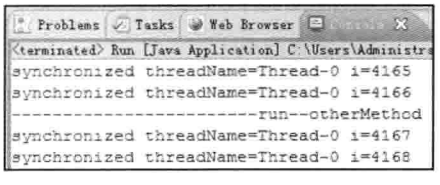


图 2-27 异步打印

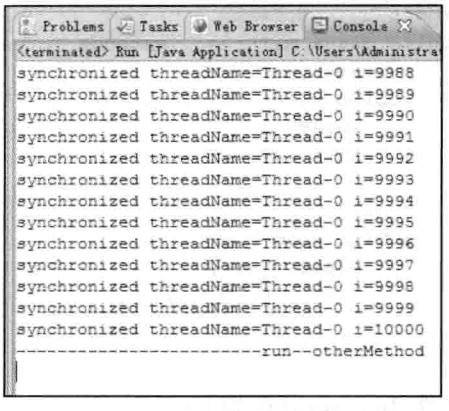


图 2-28 同步打印

2.2.7 将任意对象作为对象监视器

多个线程调用同一个对象中的不同名称的 synchronized 同步方法或 synchronized(this) 同步代码块时，调用的效果就是按顺序执行，也就是同步的，阻塞的。

这说明 synchronized 同步方法或 synchronized(this) 同步代码块分别有两种作用。

（1）synchronized 同步方法

1）对其他 synchronized 同步方法或 synchronized(this) 同步代码块调用呈阻塞状态。

2）同一时间只有一个线程可以执行 synchronized 同步方法中的代码。

（2）synchronized(this) 同步代码块

1）对其他 synchronized 同步方法或 synchronized(this) 同步代码块调用呈阻塞状态。

2）同一时间只有一个线程可以执行 synchronized(this) 同步代码块中的代码。

在前面的学习中，使用 synchronized(this) 格式来同步代码块，其实 Java 还支持对“任意对象”作为“对象监视器”来实现同步的功能。这个“任意对象”大多数是实例变量及方法的参数，使用格式为 synchronized(非 this 对象)。

根据前面对 synchronized(this) 同步代码块的作用总结可知，synchronized(非 this 对象) 格式的作用只有 1 种：synchronized(非 this 对象 x) 同步代码块。

1）在多个线程持有“对象监视器”为同一个对象的前提下，同一时间只有一个线程可以执行 synchronized(非 this 对象 x) 同步代码块中的代码。

2）当持有“对象监视器”为同一个对象的前提下，同一时间只有一个线程可以执行

synchronized(非 this 对象 x) 同步代码块中的代码。

在下面的示例中验证一下第 1 点。

创建测试用的项目 synBlockString，类文件 Service.java 代码如下：

```
package service;
public class Service {
    private String usernameParam;
    private String passwordParam;
    private String anyString = new String();
    public void setUsernamePassword(String username, String password) {
        try {
            synchronized (anyString) {
                System.out.println("线程名称为: " + Thread.currentThread().getName()
                        + "在" + System.currentTimeMillis() + "进入同步块");
                usernameParam = username;
                Thread.sleep(3000);
                passwordParam = password;
                System.out.println("线程名称为: " + Thread.currentThread().getName()
                        + "在" + System.currentTimeMillis() + "离开同步块");
            }
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

自定义线程 ThreadA.java 代码如下：

```
package extthread;
import service.Service;
public class ThreadA extends Thread {
    private Service service;
    public ThreadA(Service service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        service.setUsernamePassword("a", "aa");
    }
}
```

自定义线程 ThreadB.java 代码如下：

```
package extthread;
import service.Service;
public class ThreadB extends Thread {
```

```
    private Service service;
    public ThreadB(Service service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        service.setUsernamePassword("b", "bb");
    }
}
```

运行类 Run.java 代码如下：

```
package test;
import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        Service service = new Service();
        ThreadA a = new ThreadA(service);
        a.setName("A");
        a.start();
        ThreadB b = new ThreadB(service);
        b.setName("B");
        b.start();
    }
}
```

锁非 this 对象具有一定的优点：如果在一个类中有很多个 synchronized 方法，这时虽然能实现同步，但会受到阻塞，所以影响运行效率；但如果使用同步代码块锁非 this 对象，则 synchronized(非 this) 代码块中的程序与同步方法是异步的，不与其他锁 this 同步方法争抢 this 锁，则可大大提高运行效率。

程序运行后的效果如图 2-29 所示。

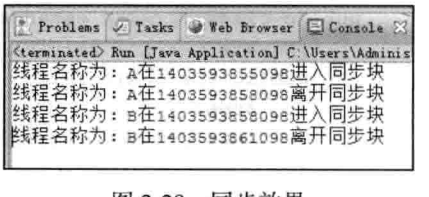


图 2-29 同步效果

Service.java 文件代码更改如下：

```
package service;
public class Service {
    private String usernameParam;
    private String passwordParam;
    public void setUsernamePassword(String username, String password) {
        try {
            String anyString = new String();
            synchronized (anyString) {
                System.out.println("线程名称为：" + Thread.currentThread().getName()
                        + "在" + System.currentTimeMillis() + "进入同步块");
                usernameParam = username;
```

```
                Thread.sleep(3000);
                passwordParam = password;
                System.out.println("线程名称为: " + Thread.currentThread().getName()
                        + "在" + System.currentTimeMillis() + "离开同步块");
            }
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

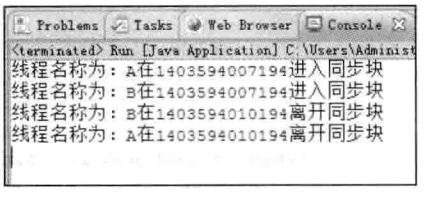


图 2-30　不是同步的而是异步（因为不是同一个锁）

程序运行结果如图 2-30 所示。

可见，使用“synchronized(非 this 对象 x) 同步代码块”格式进行同步操作时，对象监视器必须是同一个对象。如果不是同一个对象监视器，运行的结果就是异步调用了，就会交叉运行。

下面再用另外一个项目来验证一下使用“synchronized(非 this 对象 x) 同步代码块”格式时，持有不同的对象监视器是异步的效果。

创建新的项目，名称为 synBlockString2，验证 synchronized(非 this 对象) 与同步 synchronized 方法是异步调用的效果。

两个自定义线程类代码，如图 2-31 所示。

ThreadA.java

```
package extthread;

import service.Service;

public class ThreadA extends Thread {
    private Service service;

    public ThreadA(Service service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.a();

    }

}
```

```
package extthread;

import service.Service;

public class ThreadB extends Thread {

    private Service service;

    public ThreadB(Service service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.b();

    }

}
```

图 2-31　线程类代码

类 Service.java 代码如下：

```
package service;
public class Service {
    private String anyString = new String();
    public void a() {
        try {
            synchronized (anyString) {
                System.out.println("a begin");
                Thread.sleep(3000);
                System.out.println("a   end");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    synchronized public void b() {
        System.out.println("b begin");
        System.out.println("b   end");
    }
}
```

类 Run.java 代码如下：

```
package test;
import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        Service service = new Service();
        ThreadA a = new ThreadA(service);
        a.setName("A");
        a.start();
        ThreadB b = new ThreadB(service);
        b.setName("B");
        b.start();
    }
}
```

程序运行效果如图 2-32 所示。

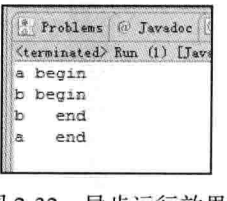


图 2-32 异步运行效果

由于对象监视器不同，所以运行结果就是异步的。

同步代码块放在非同步 synchronized 方法中进行声明，并不能保证调用方法的线程的执

行同步 / 顺序性，也就是线程调用方法的顺序是无序的，虽然在同步块中执行的顺序是同步的，这样极易出现“脏读”问题。

使用“ synchronized(非 this 对象 x) 同步代码块”格式也可以解决“脏读”问题。但在解决脏读问题之前，先做一个实验，实验的目标是验证多个线程调用同一个方法是随机的。

创建测试用的项目，项目名称为 syn_Out_asyn，类 MyList.java 代码如下：

```
package mylist;
import java.util.ArrayList;
import java.util.List;
public class MyList {
    private List list = new ArrayList();
    synchronized public void add(String username) {
        System.out.println("ThreadName=" + Thread.currentThread().getName()
                + "执行了 add 方法！ ");
        list.add(username);
        System.out.println("ThreadName=" + Thread.currentThread().getName()
                + "退出了 add 方法！ ");
    }
    synchronized public int getSize() {
        System.out.println("ThreadName=" + Thread.currentThread().getName()
                + "执行了 getSize 方法！ ");
        int sizeValue = list.size();
        System.out.println("ThreadName=" + Thread.currentThread().getName()
                + "退出了 getSize 方法！ ");
        return sizeValue;
    }
}
```

两个线程对象代码如图 2-33 所示。

MyThreadA.java

```
package extthread;

import mylist.MyList;

public class MyThreadA extends Thread {

    private MyList list;

    public MyThreadA(MyList list) {
        super();
        this.list = list;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100000; i++) {
            list.add("threadA" + (i + 1));
        }
    }

}
```

MyThreadB.java

```
package extthread;

import mylist.MyList;

public class MyThreadB extends Thread {

    private MyList list;

    public MyThreadB(MyList list) {
        super();
        this.list = list;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100000; i++) {
            list.add("threadB" + (i + 1));
        }
    }
}
```

图 2-33　两个线程对象代码

类 Test.java 代码如下：

```
package test;
import mylist.MyList;
import extthread.MyThreadA;
import extthread.MyThreadB;
public class Test {
    public static void main(String[] args) {
        MyList mylist = new MyList();
        MyThreadA a = new MyThreadA(mylist);
        a.setName("A");
        a.start();
        MyThreadB b = new MyThreadB(mylist);
        b.setName("B");
        b.start();
    }
}
```

程序运行后的结果如图 2-34 所示。

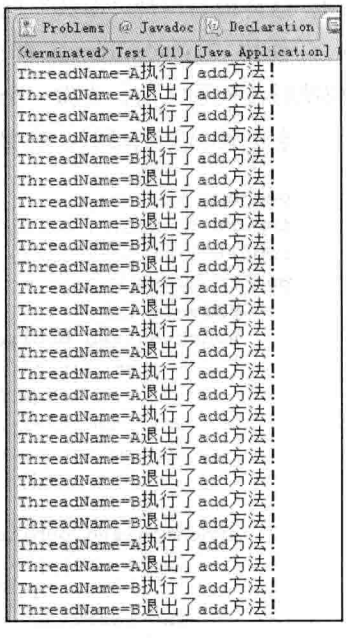


图 2-34 运行结果

从运行结果来看，同步块中的代码是同步打印的，当前线程的“执行”与“退出”是成对出现的。但线程 A 和线程 B 的执行却是异步的，这就有可能出现脏读的环境。由于线程执行方法的顺序不确定，所以当 A 和 B 两个线程执行带有分支判断的方法时，就会出现逻辑上的错误，有可能出现脏读。关于这个错误将要在下面的案例中重现。

创建 t9 项目，创建 1 个只能放 1 个元素的自定义集合工具类 MyOneList.java，代码如下：

```
package mylist;
import java.util.ArrayList;
import java.util.List;
public class MyOneList {
    private List list = new ArrayList();
    synchronized public void add(String data) {
        list.add(data);
    };
    synchronized public int getSize() {
        return list.size();
    };
}
```

创建业务类 MyService.java，代码如下：

```
package service;
import mylist.MyOneList;
public class MyService {
    public MyOneList addServiceMethod(MyOneList list, String data) {
        try {
```

```
            if (list.getSize() < 1) {
                Thread.sleep(2000);//模拟从远程花费 2 秒取回数据
                list.add(data);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return list;
    }
}
```

创建线程类 MyThread1.java，代码如下：

```
package mythread;
import mylist.MyOneList;
import service.MyService;
public class MyThread1 extends Thread {
    private MyOneList list;
    public MyThread1(MyOneList list) {
        super();
        this.list = list;
    }
    @Override
    public void run() {
        MyService msRef = new MyService();
        msRef.addServiceMethod(list, "A");
    }
}
```

创建线程类 MyThread2.java，代码如下：

```
package mythread;
import service.MyService;
import mylist.MyOneList;
public class MyThread2 extends Thread {
    private MyOneList list;
    public MyThread2(MyOneList list) {
        super();
        this.list = list;
    }
    @Override
    public void run() {
        MyService msRef = new MyService();
        msRef.addServiceMethod(list, "B");
    }
}
```

创建 Run.java，代码如下：

```
package test;
import mylist.MyOneList;
```

```
import mythread.MyThread1;
import mythread.MyThread2;
public class Run {
    public static void main(String[] args) throws InterruptedException {
        MyOneList list = new MyOneList();
        MyThread1 thread1 = new MyThread1(list);
        thread1.setName("A");
        thread1.start();
        MyThread2 thread2 = new MyThread2(list);
        thread2.setName("B");
        thread2.start();
        Thread.sleep(6000);
        System.out.println("listSize=" + list.getSize());
    }
}
```

程序运行后的结果如图 2-35 所示。

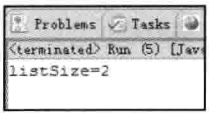


图 2-35 无序性带来的错误结果

“脏读”出现了。出现的原因是两个线程以异步的方式返回 list 参数的 size() 大小。解决办法就是“同步化”。

更改 MyService.java 类文件代码如下：

```
package service;
import mylist.MyOneList;
public class MyService {
    public MyOneList addServiceMethod(MyOneList list, String data) {
        try {
            synchronized (list) {
                if (list.getSize() < 1) {
                    Thread.sleep(2000);
                    list.add(data);
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return list;
    }
}
```

由于 list 参数对象在项目中是一份实例，是单例的，而且也正需要对 list 参数的 getSize() 方法做同步的调用，所以就对 list 参数进行同步处理。

程序运行后的结果如图 2-36 所示。

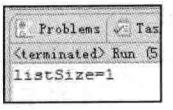


图 2-36　正确的运行结果

2.2.8　细化验证 3 个结论

“synchronized(非 this 对象 x)”格式的写法是将 x 对象本身作为“对象监视器”，这样就可以得出以下 3 个结论：

1）当多个线程同时执行 synchronized(x){} 同步代码块时呈同步效果。

2）当其他线程执行 x 对象中 synchronized 同步方法时呈同步效果。

3）当其他线程执行 x 对象方法里面的 synchronized(this) 代码块时也呈现同步效果。

但需要注意：如果其他线程调用不加 synchronized 关键字的方法时，还是异步调用。

为了验证上面的 3 个结论，创建实验用的 synchronizedBlockLockAll 项目。

（1）验证第 1 个结论

当多个线程同时执行 synchronized(x){} 同步代码块时呈同步效果。

创建名称为 test1 的包。

类 MyObject.java 代码如下：

```
package test1.extobject;
public class MyObject {
}
```

类 Service.java 代码如下：

```
package test1.service;
import test1.extobject.MyObject;
public class Service {
    public void testMethod1(MyObject object) {
        synchronized (object) {
            try {
                System.out.println("testMethod1 ____getLock time="
                        + System.currentTimeMillis() + " run ThreadName="
                        + Thread.currentThread().getName());
                Thread.sleep(2000);
                System.out.println("testMethod1 releaseLock time="
                        + System.currentTimeMillis() + " run ThreadName="
                        + Thread.currentThread().getName());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

两个自定义线程代码如图 2-37 所示。

ThreadA.java

```
package test1.extthread;

import test1.extobject.MyObject;

public class ThreadA extends Thread {

    private Service service;
    private MyObject object;

    public ThreadA(Service service, MyObject object) {
        super();
        this.service = service;
        this.object = object;
    }

    @Override
    public void run() {
        super.run();
        service.testMethod1(object);
    }

}
```

ThreadB.java

```
package test1.extthread;

import test1.extobject.MyObject;

public class ThreadB extends Thread {
    private Service service;
    private MyObject object;

    public ThreadB(Service service, MyObject object) {
        super();
        this.service = service;
        this.object = object;
    }

    @Override
    public void run() {
        super.run();
        service.testMethod1(object);
    }
}
```

图 2-37 两个线程代码

类文件 Run1_1.java 代码如下：

```
package test1.run;
import test1.extobject.MyObject;
import test1.extthread.ThreadA;
import test1.extthread.ThreadB;
import test1.service.Service;
public class Run1_1 {
    public static void main(String[] args) {
        Service service = new Service();
        MyObject object = new MyObject();
        ThreadA a = new ThreadA(service, object);
        a.setName("a");
        a.start();
        ThreadB b = new ThreadB(service, object);
        b.setName("b");
        b.start();
    }
}
```

程序运行后的结果如图 2-38 所示。

Problems | Javadoc | Declaration

```
<terminated> Run1_1 [Java Application] C:\Users\Administrator\AppData\Local\Genuite
testMethod1 ____getLock time=1405501358604 run ThreadName=a
testMethod1 releaseLock time=1405501360605 run ThreadName=a
testMethod1 ____getLock time=1405501360605 run ThreadName=b
testMethod1 releaseLock time=1405501362605 run ThreadName=b
```

图 2-38 同步调用

同步的原因是使用了同一个“对象监视器”。如果使用不同的“对象监视器”会出现什

么效果呢?

创建类文件 Run1_2.java，代码如下：

```
package test1.run;
import test1.extobject.MyObject;
import test1.extthread.ThreadA;
import test1.extthread.ThreadB;
import test1.service.Service;
public class Run1_2 {
    public static void main(String[] args) {
        Service service = new Service();
        MyObject object1 = new MyObject();
        MyObject object2 = new MyObject();
        ThreadA a = new ThreadA(service, object1);
        a.setName("a");
        a.start();
        ThreadB b = new ThreadB(service, object2);
        b.setName("b");
        b.start();
    }
}
```

程序执行效果如图 2-39 所示。

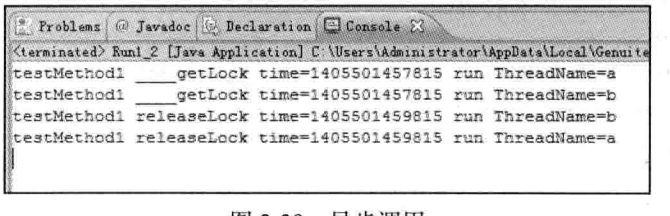


图 2-39　异步调用

（2）验证第 2 个结论

当其他线程执行 x 对象中 synchronized 同步方法时呈同步效果。

创建名称为 test2 的包。

类 MyObject.java 代码如下：

```
package test2.extobject;
public class MyObject {
    synchronized public void speedPrintString() {
        System.out.println("speedPrintString ____getLock time="
                + System.currentTimeMillis() + " run ThreadName="
                + Thread.currentThread().getName());
        System.out.println("-----------------");
        System.out.println("speedPrintString releaseLock time="
                + System.currentTimeMillis() + " run ThreadName="
                + Thread.currentThread().getName());
    }
}
```

类 Service.java 代码如下：

```
package test2.service;
import test2.extobject.MyObject;
public class Service {
    public void testMethod1(MyObject object) {
        synchronized (object) {
            try {
                System.out.println("testMethod1 ____getLock time="
                        + System.currentTimeMillis() + " run ThreadName="
                        + Thread.currentThread().getName());
                Thread.sleep(5000);
                System.out.println("testMethod1 releaseLock time="
                        + System.currentTimeMillis() + " run ThreadName="
                        + Thread.currentThread().getName());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

两个自定义线程代码如图 2-40 所示。

ThreadA.java

```
package test2.extthread;

import test2.extobject.MyObject;

public class ThreadA extends Thread {

    private Service service;
    private MyObject object;

    public ThreadA(Service service, MyObject object) {
        super();
        this.service = service;
        this.object = object;
    }

    @Override
    public void run() {
        super.run();
        service.testMethod1(object);
    }
}
```

ThreadB.java

```
package test2.extthread;

import test2.extobject.MyObject;

public class ThreadB extends Thread {
    private MyObject object;

    public ThreadB(MyObject object) {
        super();
        this.object = object;
    }

    @Override
    public void run() {
        super.run();
        object.speedPrintString();
    }
}
```

图 2-40　自定义两个线程代码

类 Run.java 代码如下：

```
package test2.run;
import test2.extobject.MyObject;
import test2.extthread.ThreadA;
import test2.extthread.ThreadB;
import test2.service.Service;
public class Run {
    public static void main(String[] args) throws InterruptedException {
        Service service = new Service();
```

```
        MyObject object = new MyObject();
        ThreadA a = new ThreadA(service, object);
        a.setName("a");
        a.start();
        Thread.sleep(100);
        ThreadB b = new ThreadB(object);
        b.setName("b");
        b.start();
    }
}
```

程序运行后的效果如图 2-41 所示。

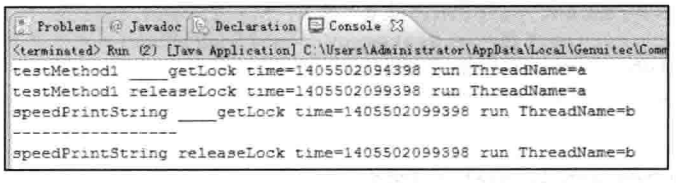


图 2-41　同步效果

（3）验证第 3 个结论

当其他线程执行 x 对象方法里面的 synchronized(this) 代码块时也呈现同步效果。

创建名称为 test3 的包。

创建名称为 MyObject.java 的类，代码如下：

```
package test3.extobject;
public class MyObject {
    public void speedPrintString() {
        synchronized (this) {
            System.out.println("speedPrintString ____getLock time="
                    + System.currentTimeMillis() + " run ThreadName="
                    + Thread.currentThread().getName());
            System.out.println("----------");
            System.out.println("speedPrintString releaseLock time="
                    + System.currentTimeMillis() + " run ThreadName="
                    + Thread.currentThread().getName());
        }
    }
}
```

其他代码与 test2 包中 Java 类的代码一样。程序运行后的效果如图 2-42 所示。

图 2-42　同样是同步效果

2.2.9 静态同步 synchronized 方法与 synchronized(class) 代码块

关键字 synchronized 还可以应用在 static 静态方法上，如果这样写，那是对当前的 *.java 文件对应的 Class 类进行持锁，测试项目在 synStaticMethod 中，类文件 Service.java 代码如下：

```
package service;
public class Service {
    synchronized public static void printA() {
        try {
            System.out.println("线程名称为: " + Thread.currentThread().getName()
                    + "在" + System.currentTimeMillis() + "进入printA");
            Thread.sleep(3000);
            System.out.println("线程名称为: " + Thread.currentThread().getName()
                    + "在" + System.currentTimeMillis() + "离开printA");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    synchronized public static void printB() {
            System.out.println("线程名称为: " + Thread.currentThread().getName()
                    + "在" + System.currentTimeMillis() + "进入printB");
            System.out.println("线程名称为: " + Thread.currentThread().getName()
                    + "在" + System.currentTimeMillis() + "离开printB");
    }
}
```

自定义线程类 ThreadA.java 代码如下：

```
package extthread;
import service.Service;
public class ThreadA extends Thread {
    @Override
    public void run() {
        Service.printA();
    }
}
```

自定义线程类 ThreadB.java 代码如下：

```
package extthread;
import service.Service;
public class ThreadB extends Thread {
    @Override
    public void run() {
        Service.printB();
    }
}
```

运行类 Run.java 代码如下：

```
package test;
import service.Service;
```

```
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        ThreadA a = new ThreadA();
        a.setName("A");
        a.start();
        ThreadB b = new ThreadB();
        b.setName("B");
        b.start();
    }
}
```

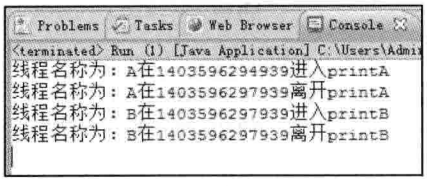


图 2-43　运行结果是同步效果

程序运行后的效果如图 2-43 所示。

从运行结果来看，并没有什么特别之处，都是同步的效果，和将 synchronized 关键字加到非 static 方法上使用的效果是一样的。其实还是有本质上的不同的，synchronized 关键字加到 static 静态方法上是给 Class 类上锁，而 synchronized 关键字加到非 static 静态方法上是给对象上锁。

为了验证不是同一个锁，创建新的项目 synTwoLock，文件 Service.java 代码如下：

```
package service;
public class Service {
    synchronized public static void printA() {
        try {
            System.out.println("线程名称为：" + Thread.currentThread().getName()
                    + "在" + System.currentTimeMillis() + "进入 printA");
            Thread.sleep(3000);
            System.out.println("线程名称为：" + Thread.currentThread().getName()
                    + "在" + System.currentTimeMillis() + "离开 printA");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    synchronized public static void printB() {
        System.out.println("线程名称为：" + Thread.currentThread().getName() + "在"
                + System.currentTimeMillis() + "进入 printB");
        System.out.println("线程名称为：" + Thread.currentThread().getName() + "在"
                + System.currentTimeMillis() + "离开 printB");
    }
    synchronized public void printC() {
        System.out.println("线程名称为：" + Thread.currentThread().getName() + "在"
                + System.currentTimeMillis() + "进入 printC");
        System.out.println("线程名称为：" + Thread.currentThread().getName() + "在"
                + System.currentTimeMillis() + "离开 printC");
    }
}
```

自定义线程类 ThreadA.java 代码如下：

```
package extthread;
```

```
import service.Service;
public class ThreadA extends Thread {
    private Service service;
    public ThreadA(Service service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        service.printA();
    }
}
```

自定义线程类 ThreadB.java 代码如下：

```
package extthread;
import service.Service;
public class ThreadB extends Thread {
    private Service service;
    public ThreadB(Service service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        service.printB();
    }
}
```

自定义线程类 ThreadC.java 代码如下：

```
package extthread;
import service.Service;
public class ThreadC extends Thread {
    private Service service;
    public ThreadC(Service service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        service.printC();
    }
}
```

运行类 Run.java 代码如下：

```
package test;
import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;
```

```
import extthread.ThreadC;
public class Run {
    public static void main(String[] args) {
        Service service = new Service();
        ThreadA a = new ThreadA(service);
        a.setName("A");
        a.start();
        ThreadB b = new ThreadB(service);
        b.setName("B");
        b.start();
        ThreadC c = new ThreadC(service);
        c.setName("C");
        c.start();
    }
}
```

程序运行结果如图 2-44 所示。

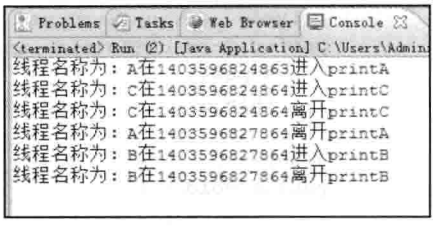


图 2-44　方法 printC() 为异步运行

异步的原因是持有不同的锁，一个是对象锁，另外一个是 Class 锁，而 Class 锁可以对类的所有对象实例起作用。用项目 synMoreObjectStaticOneLock 来验证，类文件 Service.java 代码如下：

```
package service;
public class Service {
    synchronized public static void printA() {
        try {
            System.out.println("线程名称为：" + Thread.currentThread().getName()
                    + "在" + System.currentTimeMillis() + "进入printA");
            Thread.sleep(3000);
            System.out.println("线程名称为：" + Thread.currentThread().getName()
                    + "在" + System.currentTimeMillis() + "离开printA");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    synchronized public static void printB() {
        System.out.println("线程名称为：" + Thread.currentThread().getName() + "在"
                + System.currentTimeMillis() + "进入printB");
        System.out.println("线程名称为：" + Thread.currentThread().getName() + "在"
                + System.currentTimeMillis() + "离开printB");
    }
}
```

自定义线程 ThreadA.java 代码如下：

```
package extthread;
import service.Service;
public class ThreadA extends Thread {
```

```
    private Service service;
    public ThreadA(Service service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        service.printA();
    }
}
```

自定义线程 ThreadB.java 代码如下：

```
package extthread;
import service.Service;
public class ThreadB extends Thread {
    private Service service;
    public ThreadB(Service service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        service.printB();
    }
}
```

运行类 Run.java 代码如下：

```
package test;
import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        Service service1 = new Service();
        Service service2 = new Service();
        ThreadA a = new ThreadA(service1);
        a.setName("A");
        a.start();
        ThreadB b = new ThreadB(service2);
        b.setName("B");
        b.start();
    }
}
```

程序运行结果如图 2-45 所示。

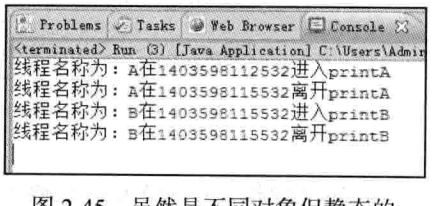


图 2-45 虽然是不同对象但静态的同步方法还是同步运行

同步 synchronized (class) 代码块的作用其实和 synchronized static 方法的作用一样。创建测试用的项目 synBlockMoreObjectOneLock，类文件 Service.java 代码如下：

```
package service;
public class Service {
    public static void printA() {
        synchronized (Service.class) {
            try {
                System.out.println("线程名称为: " + Thread.currentThread().getName()
                        + "在" + System.currentTimeMillis() + "进入printA");
                Thread.sleep(3000);
                System.out.println("线程名称为: " + Thread.currentThread().getName()
                        + "在" + System.currentTimeMillis() + "离开printA");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public static void printB() {
        synchronized (Service.class) {
            System.out.println("线程名称为: " + Thread.currentThread().getName()
                    + "在" + System.currentTimeMillis() + "进入printB");
            System.out.println("线程名称为: " + Thread.currentThread().getName()
                    + "在" + System.currentTimeMillis() + "离开printB");
        }
    }
}
```

两个自定义线程代码如图 2-46 所示。

ThreadA.java

```
package extthread;

import service.Service;

public class ThreadA extends Thread {
    private Service service;

    public ThreadA(Service service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.printA();
    }
}
```

ThreadB.java

```
package extthread;

import service.Service;

public class ThreadB extends Thread {
    private Service service;

    public ThreadB(Service service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.printB();
    }
}
```

图 2-46　两个自定义线程代码

运行类 Run.java 代码如下：

```
package test;
import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
```

```
        Service service1 = new Service();
        Service service2 = new Service();
        ThreadA a = new ThreadA(service1);
        a.setName("A");
        a.start();
        ThreadB b = new ThreadB(service2);
        b.setName("B");
        b.start();
    }
}
```

程序运行后的结果如图 2-47 所示。

图 2-47 同步运行

2.2.10 数据类型 String 的常量池特性

在 JVM 中具有 String 常量池缓存的功能，所以如图 2-48 所示的结果为 true。

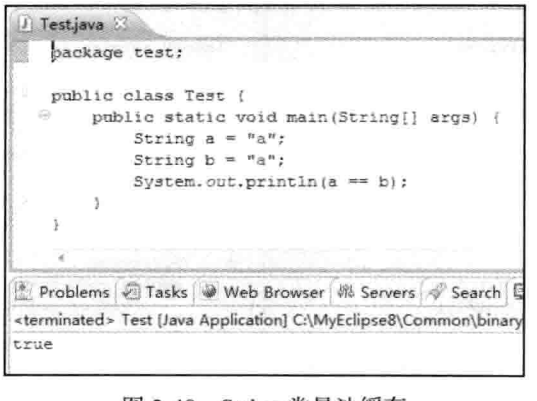


图 2-48 String 常量池缓存

将 synchronized(string) 同步块与 String 联合使用时，要注意常量池以带来的一些例外。新建名称为 StringAndSyn 的项目，类文件 Service.java 代码如下：

```
package service;
public class Service {
    public static void print(String stringParam) {
        try {
```

```
            synchronized (stringParam) {
                while (true) {
                    System.out.println(Thread.currentThread().getName());
                    Thread.sleep(1000);
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

两个自定义线程代码如图 2-49 所示。

ThreadA.java

```
1  package extthread;
2
3  import service.Service;
4
5  public class ThreadA extends Thread {
6      private Service service;
7      public ThreadA(Service service) {
8          super();
9          this.service = service;
10     }
11
12     @Override
13     public void run() {
14         service.print("AA");
15     }
16 }
17
```

ThreadB.java

```
1  package extthread;
2
3  import service.Service;
4
5  public class ThreadB extends Thread {
6      private Service service;
7      public ThreadB(Service service) {
8          super();
9          this.service = service;
10     }
11
12     @Override
13     public void run() {
14         service.print("AA");
15     }
16 }
17
```

图 2-49　两个线程类代码

运行类 Run.java 代码如下：

```
package test;
import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        Service service = new Service();
        ThreadA a = new ThreadA(service);
        a.setName("A");
        a.start();
        ThreadB b = new ThreadB(service);
        b.setName("B");
        b.start();
    }
}
```

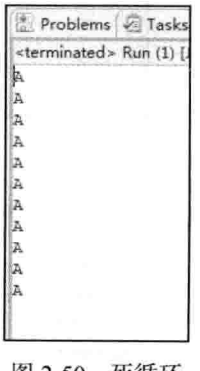


图 2-50　死循环

程序运行结果如图 2-50 所示。

出现这样的情况就是因为 String 的两个值都是 AA，两个线程持有相同的锁，所以造

成线程 B 不能执行。这就是 String 常量池所带来的问题。因此在大多数的情况下，同步 synchronized 代码块都不使用 String 作为锁对象，而改用其他，比如 new Object() 实例化一个 Object 对象，但它并不放入缓存中。

继续实验，创建名称为 StringAndSyn2 的项目，类文件 Service.java 代码如下：

```
package service;
public class Service {
    public static void print(Object object) {
        try {
            synchronized (object) {
                while (true) {
                System.out.println(Thread.currentThread().getName());
                    Thread.sleep(1000);
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

两个自定义线程如图 2-51 所示。

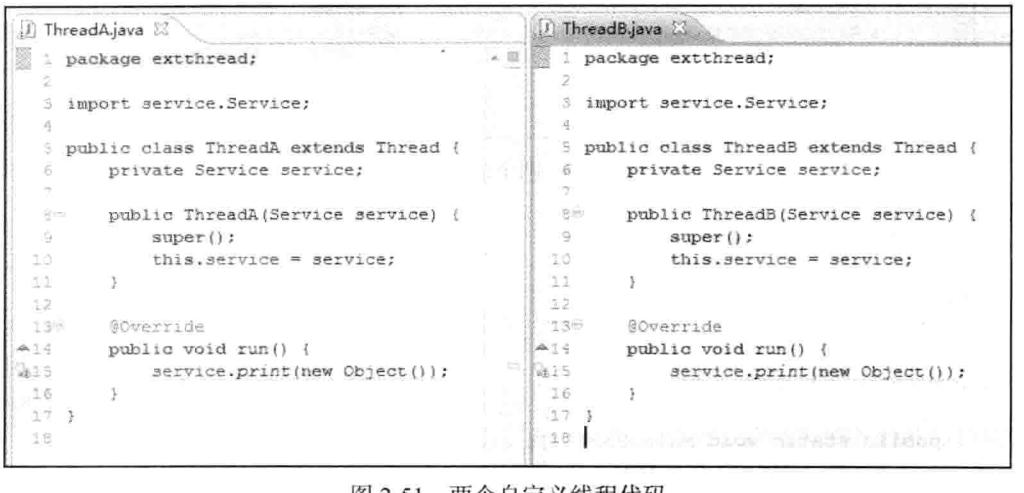


图 2-51 两个自定义线程代码

运行类 Run.java 代码如下：

```
package test;
import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        Service service = new Service();
```

```
        ThreadA a = new ThreadA(service);
        a.setName("A");
        a.start();
        ThreadB b = new ThreadB(service);
        b.setName("B");
        b.start();
    }
}
```

图 2-52　交替打印

程序运行后的效果如图 2-52 所示。

交替打印的原因是持有的锁不是一个。

2.2.11　同步 synchronized 方法无限等待与解决

同步方法容易造成死循环。示例项目 twoStop，类 Service.java 代码如下：

```
package service;
public class Service {
    synchronized public void methodA() {
        System.out.println("methodA begin");
        boolean isContinueRun = true;
        while (isContinueRun) {
        }
        System.out.println("methodA end");
    }
    synchronized public void methodB() {
        System.out.println("methodB begin");
        System.out.println("methodB end");
    }
}
```

自定义线程类代码如图 2-53 所示。

ThreadA.java

```
package extthread;

import service.Service;

public class ThreadA extends Thread {

    private Service service;

    public ThreadA(Service service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.methodA();
    }

}
```

ThreadB.java

```
package extthread;

import service.Service;

public class ThreadB extends Thread {

    private Service service;

    public ThreadB(Service service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.methodB();
    }

}
```

图 2-53　自定义线程类代码

运行类 Run.java 代码如下：

```
package test.run;
import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        Service service = new Service();
        ThreadA athread = new ThreadA(service);
        athread.start();
        ThreadB bthread = new ThreadB(service);
        bthread.start();
    }
}
```

程序运行后的效果如图 2-54 所示。

线程 B 永远得不到运行的机会，锁死了。

这时就可以使用同步块来解决这样的问题。更改后的 Service.java 文件代码如下：

```
package service;
public class Service {
    Object object1 = new Object();
    public void methodA() {
        synchronized (object1) {
            System.out.println("methodA begin");
            boolean isContinueRun = true;
            while (isContinueRun) {
            }
            System.out.println("methodA end");
        }
    }
    Object object2 = new Object();
    public void methodB() {
        synchronized (object2) {
            System.out.println("methodB begin");
            System.out.println("methodB end");
        }
    }
}
```

程序运行结果如图 2-55 所示。

图 2-54 运行结果是死循环

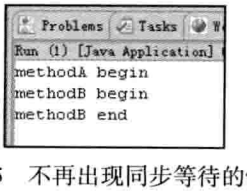


图 2-55 不再出现同步等待的情况

本示例代码在项目 twoNoStop 中。

2.2.12　多线程的死锁

Java 线程死锁是一个经典的多线程问题，因为不同的线程都在等待根本不可能被释放的锁，从而导致所有的任务都无法继续完成。在多线程技术中，“死锁”是必须避免的，因为这会造成线程的“假死”。

创建名称为 deadLockTest 的项目，DealThread.java 类代码如下：

```
package test;
public class DealThread implements Runnable {
    public String username;
    public Object lock1 = new Object();
    public Object lock2 = new Object();
    public void setFlag(String username) {
        this.username = username;
    }
    @Override
    public void run() {
        if (username.equals("a")) {
            synchronized (lock1) {
                try {
                    System.out.println("username = " + username);
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                synchronized (lock2) {
                    System.out.println(" 按 lock1->lock2 代码顺序执行了 ");
                }
            }
        }
        if (username.equals("b")) {
            synchronized (lock2) {
                try {
                    System.out.println("username = " + username);
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                synchronized (lock1) {
                    System.out.println(" 按 lock2->lock1 代码顺序执行了 ");
                }
            }
        }
    }
}
```

运行类 Run.java 代码如下：

```
package test;
public class Run {
    public static void main(String[] args) {
        try {
            DealThread t1 = new DealThread();
            t1.setFlag("a");
            Thread thread1 = new Thread(t1);
            thread1.start();
            Thread.sleep(100);
            t1.setFlag("b");
            Thread thread2 = new Thread(t1);
            thread2.start();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 2-56 所示。

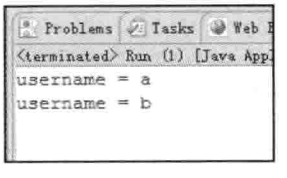


图 2-56 出现死锁

可以使用 JDK 自带的工具来监测是否有死锁的现象。首先进入 CMD 工具，再进入 JDK 的安装文件夹中的 bin 目录，执行 jps 命令，如图 2-57 所示。

图 2-57 执行 jps 命令

得到运行的线程 Run 的 id 值是 3244。再执行 jstack 命令，查看结果，如图 2-58 所示。

```
C:\Users\Administrator\AppData\Local\Genuitec\Common\binary\com.sun.java.jdk.win32.x86_1.6.0.013\bin>jstack -l 3244
```

图 2-58 执行 jstack 命令

监测出有死锁现象，如图 2-59 所示。

死锁是程序设计的 Bug，在设计程序时就要避免双方互相持有对方的锁的情况。需要说明的是，本实验使用 synchronized 嵌套的代码结构来实现死锁，其实不使用嵌套的 synchronized 代码结构也会出现死锁，与嵌套不嵌套无任何的关系，不要被代码结构所误导。只要互相等待对方释放锁就有可能出现死锁。

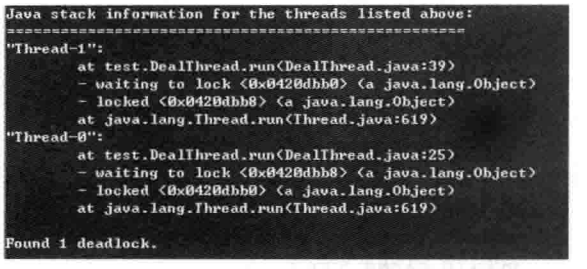


图 2-59　监测出死锁

2.2.13　内置类与静态内置类

关键字 synchronized 的知识点还涉及内置类的使用。先来看一下简单的内置类的测试。创建 innerClass 项目，类 PublicClass.java 代码如下：

```
package test;
public class PublicClass {
    private String username;
    private String password;
    class PrivateClass {
        private String age;
        private String address;
        public String getAge() {
            return age;
        }
        public void setAge(String age) {
            this.age = age;
        }
        public String getAddress() {
            return address;
        }
        public void setAddress(String address) {
            this.address = address;
        }
        public void printPublicProperty() {
            System.out.println(username + " " + password);
        }
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
```

```
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

创建运行类 Run.java，代码如下：

```
package test;
import test.PublicClass.PrivateClass;
public class Run {
    public static void main(String[] args) {
        PublicClass publicClass = new PublicClass();
        publicClass.setUsername("usernameValue");
        publicClass.setPassword("passwordValue");
        System.out.println(publicClass.getUsername() + " "
                + publicClass.getPassword());
        PrivateClass privateClass = publicClass.new PrivateClass();
        privateClass.setAge("ageValue");
        privateClass.setAddress("addressValue");
        System.out.println(privateClass.getAge() + " "
                + privateClass.getAddress());
    }
}
```

如果 PublicClass.java 类和 Run.java 类不在同一个包中，则需要将 PrivateClass 内置声明成 public 公开的。

程序运行后的结果如图 2-60 所示。

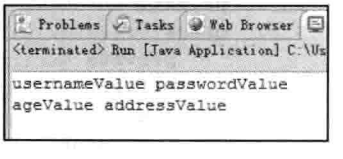


图 2-60 运行结果

想要实例化内置类必须使用如下代码：

```
PrivateClass privateClass = publicClass.new PrivateClass();
```

内置类还有一种叫作静态内置类。

创建实验用的项目 innerStaticClass，类 PublicClass.java 代码如下：

```
package test;
public class PublicClass {
    static private String username;
    static private String password;
    static class PrivateClass {
        private String age;
        private String address;
        public String getAge() {
            return age;
        }
        public void setAge(String age) {
            this.age = age;
        }
        public String getAddress() {
```

```
            return address;
        }
        public void setAddress(String address) {
            this.address = address;
        }
        public void printPublicProperty() {
            System.out.println(username + " " + password);
        }
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

运行类 Run.java 代码如下：

```
package test;
import test.PublicClass.PrivateClass;
public class Run {
    public static void main(String[] args) {
        PublicClass publicClass = new PublicClass();
        publicClass.setUsername("usernameValue");
        publicClass.setPassword("passwordValue");
        System.out.println(publicClass.getUsername() + " "
                + publicClass.getPassword());
        PrivateClass privateClass = new PrivateClass();
        privateClass.setAge("ageValue");
        privateClass.setAddress("addressValue");
        System.out.println(privateClass.getAge() + " "
                + privateClass.getAddress());
    }
}
```

程序运行后的结果如图 2-61 所示。

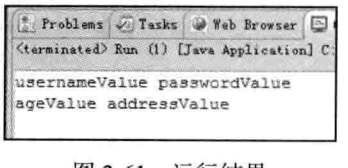


图 2-61　运行结果

2.2.14　内置类与同步：实验 1

本实验测试的案例是在内置类中有两个同步方法，但使用的却是不同的锁，打印的结果也是异步的。

实验用的项目 innerTest1，类 OutClass.java 代码如下：

```
package test;
public class OutClass {
    static class Inner {
        public void method1() {
            synchronized ("其他的锁") {
                for (int i = 1; i <= 10; i++) {
                    System.out.println(Thread.currentThread().getName() + " i="
                            + i);
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                    }
                }
            }
        }
        public synchronized void method2() {
            for (int i = 11; i <= 20; i++) {
                System.out
                   .println(Thread.currentThread().getName() + " i=" + i);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                }
            }
        }
    }
}
```

运行类 Run.java 代码如下：

```
package test;
import test.OutClass.Inner;
public class Run {
    public static void main(String[] args) {
        final Inner inner = new Inner();
        Thread t1 = new Thread(new Runnable() {
            public void run() {
                inner.method1();
            }
        }, "A");
        Thread t2 = new Thread(new Runnable() {
            public void run() {
                inner.method2();
            }
        }, "B");
        t1.start();
        t2.start();
    }
}
```

程序运行结果如图 2-62 所示。

由于持有不同的“对象监视器”，所以打印结果就是乱序的。

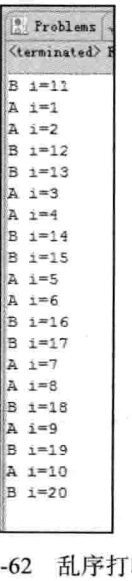


图 2-62 乱序打印

2.2.15　内置类与同步：实验 2

本实验测试同步代码块 synchronized(class2) 对 class2 上锁后，其他线程只能以同步的方式调用 class2 中的静态同步方法。

创建测试用的项目 innerTest2，类 OutClass.java 代码如下：

```
package test;
public class OutClass {
    static class InnerClass1 {
        public void method1(InnerClass2 class2) {
            String threadName = Thread.currentThread().getName();
            synchronized (class2) {
                System.out.println(threadName + " 进入 InnerClass1 类中的 method1 方法 ");
                for (int i = 0; i < 10; i++) {
                    System.out.println("i=" + i);
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                    }
                }
                System.out.println(threadName + " 离开 InnerClass1 类中的 method1 方法 ");
            }
        }
        public synchronized void method2() {
            String threadName = Thread.currentThread().getName();
            System.out.println(threadName + " 进入 InnerClass1 类中的 method2 方法 ");
            for (int j = 0; j < 10; j++) {
                System.out.println("j=" + j);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                }
            }
            System.out.println(threadName + " 离开 InnerClass1 类中的 method2 方法 ");
        }
    }
    static class InnerClass2 {
        public synchronized void method1() {
            String threadName = Thread.currentThread().getName();
            System.out.println(threadName + " 进入 InnerClass2 类中的 method1 方法 ");
            for (int k = 0; k < 10; k++) {
                System.out.println("k=" + k);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                }
            }
            System.out.println(threadName + " 离开 InnerClass2 类中的 method1 方法 ");
        }
    }
}
```

运行类 Run.java 代码如下：

```
package test;
import test.OutClass.InnerClass1;
import test.OutClass.InnerClass2;
public class Run {
    public static void main(String[] args) {
        final InnerClass1 in1 = new InnerClass1();
        final InnerClass2 in2 = new InnerClass2();
        Thread t1 = new Thread(new Runnable() {
            public void run() {
                in1.method1(in2);
            }
        }, "T1");
        Thread t2 = new Thread(new Runnable() {
            public void run() {
                in1.method2();
            }
        }, "T2");
        ///
        ///
        Thread t3 = new Thread(new Runnable() {
            public void run() {
                in2.method1();
            }
        }, "T3");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

程序运行结果如图 2-63 所示。

2.2.16 锁对象的改变

在将任何数据类型作为同步锁时，需要注意的是，是否有多个线程同时持有锁对象，如果同时持有相同的锁对象，则这些线程之间就是同步的；如果分别获得锁对象，这些线程之间就是异步的。

创建测试用的项目 setNewStringTwoLock，MyService.java 类代码如下：

```
package myservice;
public class MyService {
    private String lock = "123";
    public void testMethod() {
        try {
            synchronized (lock) {
                System.out.println(Thread.currentThread().getName() + " begin "
                        + System.currentTimeMillis());
```

```
                lock = "456";
                Thread.sleep(2000);
                System.out.println(Thread.currentThread().getName() + "    end "
                        + System.currentTimeMillis());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

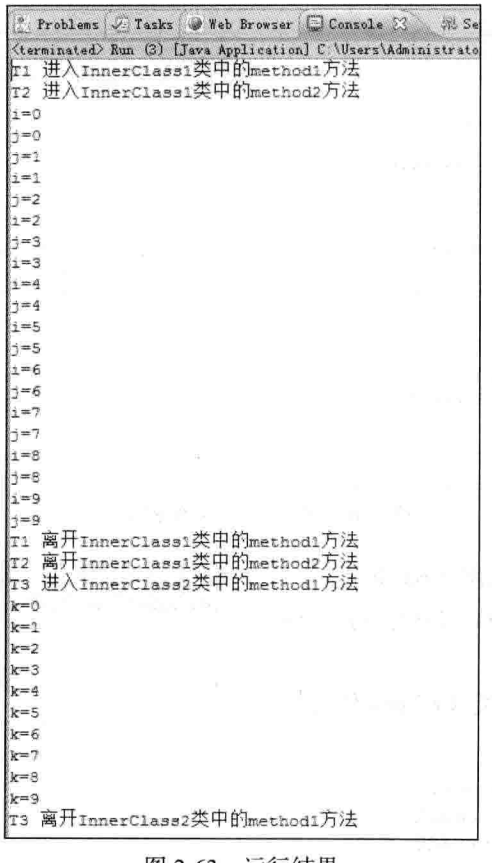


图 2-63　运行结果

. 两个自定义线程类代码如图 2-64 所示。

运行类 Run1.java 代码如下：

```
package test.run;
import myservice.MyService;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run1 {
```

```
    public static void main(String[] args) throws InterruptedException {
        MyService service = new MyService();
        ThreadA a = new ThreadA(service);
        a.setName("A");
        ThreadB b = new ThreadB(service);
        b.setName("B");
        a.start();
        Thread.sleep(50);//存在 50 毫秒
        b.start();
    }
}
```

ThreadA.java

```
1  package extthread;
2
3  import myservice.MyService;
4
5  public class ThreadA extends Thread {
6
7      private MyService service;
8
9      public ThreadA(MyService service) {
10         super();
11         this.service = service;
12     }
13
14     @Override
15     public void run() {
16         service.testMethod();
17     }
18 }
19
```

ThreadB.java

```
1  package extthread;
2
3  import myservice.MyService;
4
5  public class ThreadB extends Thread {
6
7      private MyService service;
8
9      public ThreadB(MyService service) {
10         super();
11         this.service = service;
12     }
13
14     @Override
15     public void run() {
16         service.testMethod();
17     }
18 }
19
```

图 2-64 两个自定义线程类代码

程序运行后的效果如图 2-65 所示。

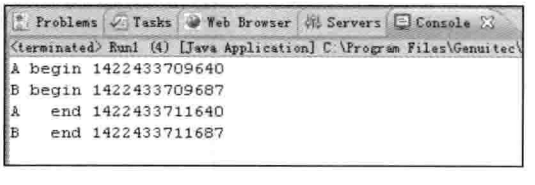


图 2-65 异步输出

因为 50 毫秒过后线程 B 取得的锁是“456”。

继续实验，创建运行类 Run2.java 代码如下：

```
package test.run;
import myservice.MyService;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run2 {
    public static void main(String[] args) throws InterruptedException {
        MyService service = new MyService();
        ThreadA a = new ThreadA(service);
        a.setName("A");
        ThreadB b = new ThreadB(service);
        b.setName("B");
        a.start();
        b.start();
    }
}
```

去掉代码 Thread.sleep(50)，程序运行后的效果如图 2-66 所示。

线程 A 和 B 持有的锁都是“123”，虽然将锁改成了“456”，但结果还是同步的，因为 A 和 B 共同争抢的锁是“123”。

还需要提示一下，只要对象不变，既使对象的属性被改变，运行的结果还是同步。此结论的实验代码在 setNewProperties LockOne 项目里进行演示，创建类 Userinfo.java，结构如图 2-67 所示。

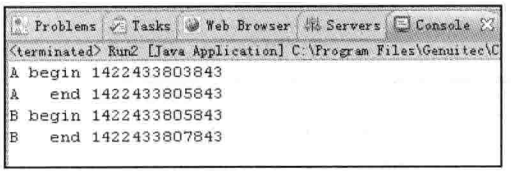


图 2-66　同步输出

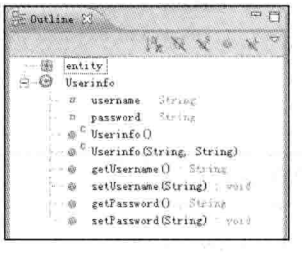


图 2-67　类结构

类 Service.java 代码如下：

```
package service;
import entity.Userinfo;
public class Service {
    public void serviceMethodA(Userinfo userinfo) {
        synchronized (userinfo) {
            try {
                System.out.println(Thread.currentThread().getName());
                userinfo.setUsername("abcabcabc");
                Thread.sleep(3000);
                System.out.println("end! time=" + System.currentTimeMillis());
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

两个线程类代码如图 2-68 所示。

运行类 Run.java 代码如下：

```
package test.run;
import service.Service;
import entity.Userinfo;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        try {
            Service service = new Service();
```

```
            Userinfo userinfo = new Userinfo();
            ThreadA a = new ThreadA(service, userinfo);
            a.setName("a");
            a.start();
            Thread.sleep(50);
            ThreadB b = new ThreadB(service, userinfo);
            b.setName("b");
            b.start();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

ThreadA.java

```
package extthread;

import service.Service;

public class ThreadA extends Thread {

    private Service service;
    private Userinfo userinfo;

    public ThreadA(Service service,
            Userinfo userinfo) {
        super();
        this.service = service;
        this.userinfo = userinfo;
    }

    @Override
    public void run() {
        service.serviceMethodA(userinfo);
    }

}
```

ThreadB.java

```
package extthread;

import service.Service;

public class ThreadB extends Thread {

    private Service service;
    private Userinfo userinfo;

    public ThreadB(Service service,
            Userinfo userinfo) {
        super();
        this.service = service;
        this.userinfo = userinfo;
    }

    @Override
    public void run() {
        service.serviceMethodA(userinfo);
    }

}
```

图 2-68 两个线程类代码

程序运行的效果如图 2-69 所示。

Problems Tasks Web Browser

<terminated> Run (24) [Java Applicatio

```
a
end! time=1413511508687
b
end! time=1413511511687
```

图 2-69 只要对象不变就是同步效果

2.3 volatile 关键字

关键字 volatile 的主要作用是使变量在多个线程间可见。

2.3.1 关键字 volatile 与死循环

如果不是在多继承的情况下，使用继承 Thread 类和实现 Runnable 接口在取得程序运行的结果上并没有什么太大的区别。如果一旦出现“多继承”的情况，则用实现 Runnable 接口的方式来处理多线程的问题就是很有必要的。

本节将用实现 Runnable 接口的方式来继续理解多线程技术的使用，并且使用关键字

volatile 来实验在并发情况下的一些特性。此案例也同样适用于继承自 Thread 类。

创建测试用的项目 t99，创建 PrintString.java 类，代码如下：

```
public class PrintString {
    private boolean isContinuePrint = true;
    public boolean isContinuePrint() {
        return isContinuePrint;
    }
    public void setContinuePrint(boolean isContinuePrint) {
        this.isContinuePrint = isContinuePrint;
    }
    public void printStringMethod() {
        try {
            while (isContinuePrint == true) {
                System.out.println("run printStringMethod threadName="
                        + Thread.currentThread().getName());
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

运行类 Run.java 代码如下：

```
public class Run {
    public static void main(String[] args) {
        PrintString printStringService = new PrintString();
        printStringService.printStringMethod();
        System.out.println("我要停止它！ stopThread="
                + Thread.currentThread().getName());
        printStringService.setContinuePrint(false);
    }
}
```

程序开始运行后，根本停不下来，结果如图 2-70 所示。

```
el [Java Application] C:\accpMyEclipse8.5\Genuitec\Common\
rintStringMethod threadName=main
rintStringMethod threadName=main
rintStringMethod threadName=main
rintStringMethod threadName=main
rintStringMethod threadName=main
rintStringMethod threadName=main
rintStringMethod threadName=main
```

图 2-70　停不下来的程序

停不下来的原因主要就是 main 线程一直在处理 while() 循环，导致程序不能继续执行后面的代码。解决的办法当然是用多线程技术。

2.3.2　解决同步死循环

继续创建新的项目 t10，更改 PrintString.java 类，代码如下：

```
public class PrintString implements Runnable {
```

```
    private boolean isContinuePrint = true;
    public boolean isContinuePrint() {
        return isContinuePrint;
    }
    public void setContinuePrint(boolean isContinuePrint) {
        this.isContinuePrint = isContinuePrint;
    }
    public void printStringMethod() {
        try {
            while (isContinuePrint == true) {
                System.out.println("run printStringMethod threadName="
                        + Thread.currentThread().getName());
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    @Override
    public void run() {
        printStringMethod();
    }
}
```

运行 Run.java 类代码如下：

```
public class Run {
    public static void main(String[] args) {
        PrintString printStringService = new PrintString();
        new Thread(printStringService).start();
        System.out.println("我要停止它! stopThread="
                + Thread.currentThread().getName());
        printStringService.setContinuePrint(false);
    }
}
```

程序运行结果如图 2-71 所示。

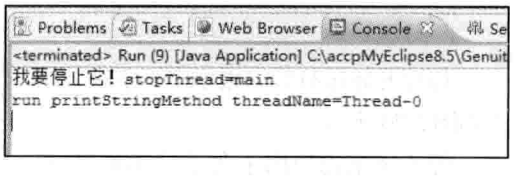


图 2-71 程序被停止了

但当上面的示例代码的格式运行在 -server 服务器模式中 64bit 的 JVM 上时，会出现死循环。解决的办法是使用 volatile 关键字。

关键字 volatile 的作用是强制从公共堆栈中取得变量的值，而不是从线程私有数据栈中取得变量的值。

2.3.3 解决异步死循环

在研究 volatile 关键字之前先来做一个实验，创建 t16 的项目，RunThread.java 类代码如下：

```
package extthread;
```

```
public class RunThread extends Thread {
    private boolean isRunning = true;
    public boolean isRunning() {
        return isRunning;
    }
    public void setRunning(boolean isRunning) {
        this.isRunning = isRunning;
    }
    @Override
    public void run() {
        System.out.println("进入 run 了");
        while (isRunning == true) {
        }
        System.out.println("线程被停止了！");
    }
}
```

类 Run.java 代码如下：

```
package test;
import extthread.RunThread;
public class Run {
    public static void main(String[] args) {
        try {
            RunThread thread = new RunThread();
            thread.start();
            Thread.sleep(1000);
            thread.setRunning(false);
            System.out.println("已经赋值为 false");
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

在 Win7 结合 JDK64bit 的环境中，使用 Eclipse 开发环境运行后的结果如图 2-72 所示。

但是如果使用同样的代码，让它们运行在 JVM 设置为 Server 服务器的环境中，会是什么样的结果呢？配置 Eclipse 中 JVM 的运行参数为 -server，设置界面如图 2-73 所示。

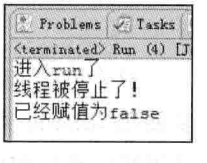


图 2-72　打印结果

单击 Run 按钮后出现了死循环的效果，如图 2-74 所示。

代码“System.out.println("线程被停止了！")”从未被执行。

是什么样的原因造成将 JVM 设置为 -server 时就出现死循环呢？在启动 RunThread.java 线程时，变量 private boolean isRunning = true; 存在于公共堆栈及线程的私有堆栈中。在 JVM 被设置为 -server 模式时为了线程运行的效率，线程一直在私有堆栈中取得 isRunning 的值是 true。而代码 thread.setRunning(false); 虽然被执行，更新的却是公共堆栈中的 isRunning 变量值 false，所以一直就是死循环的状态。内存结构如图 2-75 所示。

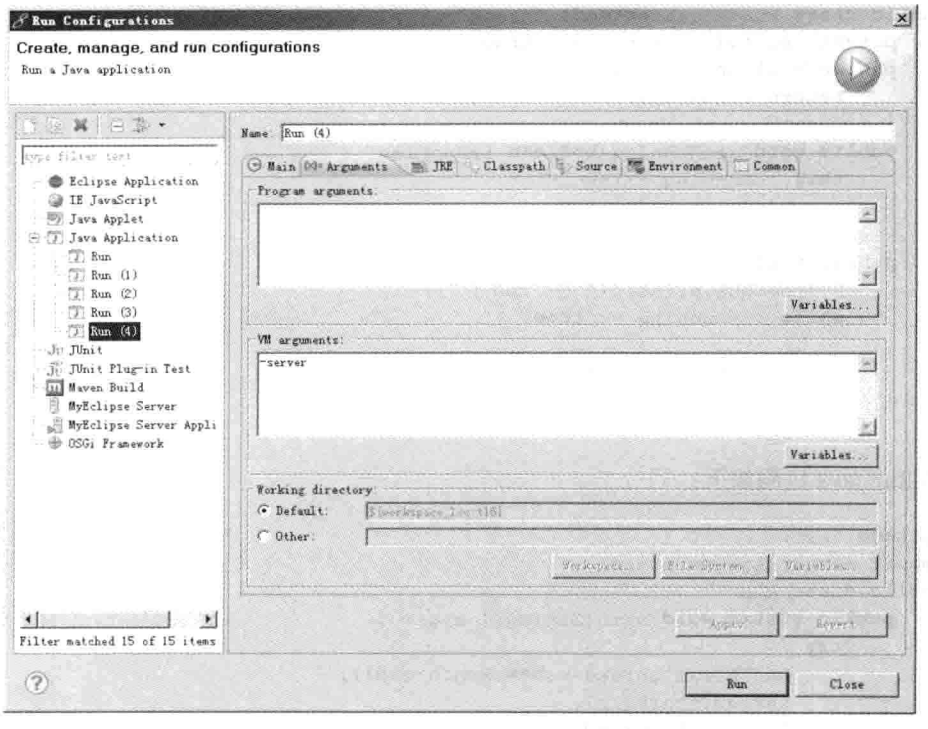


图 2-73 配置 JVM 为 -server 模式

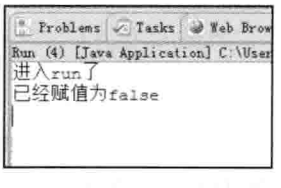


图 2-74 出现了死循环

图 2-75 线程的私有堆栈

这个问题其实就是私有堆栈中的值和公共堆栈中的值不同步造成的。解决这样的问题就要使用 volatile 关键字了，它主要的作用就是当线程访问 isRunning 这个变量时，强制性从公共堆栈中进行取值。

将 RunThread.java 代码更改如下：

```
package extthread;
public class RunThread extends Thread {
    volatile private boolean isRunning = true;
    public boolean isRunning() {
        return isRunning;
    }
```

```
    public void setRunning(boolean isRunning) {
        this.isRunning = isRunning;
    }
    @Override
    public void run() {
        System.out.println("进入 run 了 ");
        while (isRunning == true) {
        }
        System.out.println("线程被停止了！ ");
    }
}
```

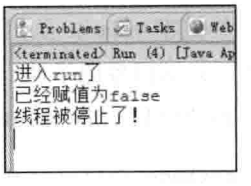


图 2-76　在 –server 服务器模式不再出现死循环

程序运行后的效果如图 2-76 所示。

通过使用 volatile 关键字，强制的从公共内存中读取变量的值，内存结构如图 2-77 所示。

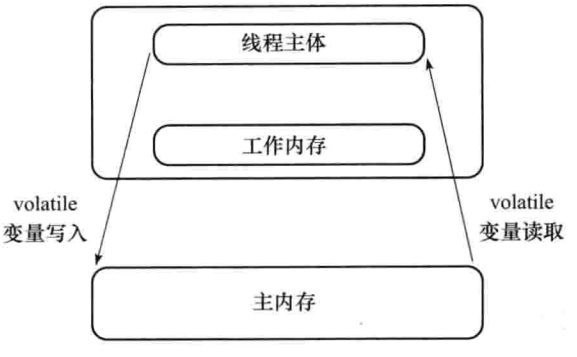


图 2-77　读取公共内存

使用 volatile 关键字增加了实例变量在多个线程之间的可见性。但 volatile 关键字最致命的缺点是不支持原子性。

下面将关键字 synchronized 和 volatile 进行一下比较：

1）关键字 volatile 是线程同步的轻量级实现，所以 volatile 性能肯定比 synchronized 要好，并且 volatile 只能修饰于变量，而 synchronized 可以修饰方法，以及代码块。随着 JDK 新版本的发布，synchronized 关键字在执行效率上得到很大提升，在开发中使用 synchronized 关键字的比率还是比较大的。

2）多线程访问 volatile 不会发生阻塞，而 synchronized 会出现阻塞。

3）volatile 能保证数据的可见性，但不能保证原子性；而 synchronized 可以保证原子性，也可以间接保证可见性，因为它会将私有内存和公共内存中的数据做同步。此知识点在后面有实验做论证。

4）再次重申一下，关键字 volatile 解决的是变量在多个线程之间的可见性；而 synchronized

关键字解决的是多个线程之间访问资源的同步性。

线程安全包含原子性和可见性两个方面，Java 的同步机制都是围绕这两个方面来确保线程安全的。

2.3.4 volatile 非原子的特性

关键字 volatile 虽然增加了实例变量在多个线程之间的可见性，但它却不具备同步性，那么也就不具备原子性。下面用项目来进行测试。

创建项目 volatileTestThread，文件 MyThread.java 代码如下：

```
package extthread;
public class MyThread extends Thread {
    volatile public static int count;
    private static void addCount() {
        for (int i = 0; i < 100; i++) {
            count++;
        }
        System.out.println("count=" + count);
    }
    @Override
    public void run() {
        addCount();
    }
}
```

文件 Run.java 代码如下：

```
package test.run;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread[] mythreadArray = new MyThread[100];
        for (int i = 0; i < 100; i++) {
            mythreadArray[i] = new MyThread();
        }
        for (int i = 0; i < 100; i++) {
            mythreadArray[i].start();
        }
    }
}
```

程序运行结果如图 2-78 所示。

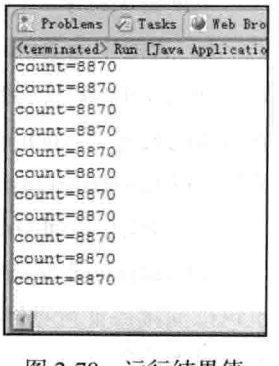


图 2-78 运行结果值不是 10000

更改自定义线程类 MyThread.java 文件代码如下：

```
package extthread;
public class MyThread extends Thread {
    volatile public static int count;
```

```
        //注意一定要添加 static 关键字
        //这样 synchronized 与 static 锁的内容就是 MyThread.class 类了
        //也就达到同步的效果了
        synchronized private static void addCount() {
                for (int i = 0; i < 100; i++) {
                    count++;
                }
                System.out.println("count=" + count);
        }
        @Override
        public void run() {
            addCount();
        }
}
```

程序运行后的结果如图 2-79 所示。

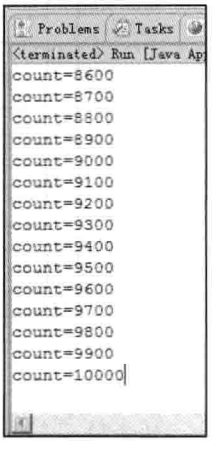


图 2-79　正确的运行结果

在本示例的意图中，如果在方法 private static void addCount() 前加入 synchronized 同步关键字，也就没有必要再使用 volatile 关键字来声明 count 变量了。

关键字 volatile 主要使用的场合是在多个线程中可以感知实例变量被更改了，并且可以获得最新的值使用，也就是用多线程读取共享变量时可以获得最新值使用。

关键字 volatile 提示线程每次从共享内存中读取变量，而不是从私有内存中读取，这样就保证了同步数据的可见性。但在这里需要注意的是：如果修改实例变量中的数据，比如 i++，也就是 i=i+1，则这样的操作其实并不是一个原子操作，也就是非线程安全的。表达式 i++ 的操作步骤分解如下：

1）从内存中取出 i 的值；

2）计算 i 的值；

3）将 i 的值写到内存中。

假如在第 2 步计算值的时候，另外一个线程也修改 i 的值，那么这个时候就会出现脏数据。解决的办法其实就是使用 synchronized 关键字，这个知识点在前面的案例中已经介绍过了。所以说 volatile 本身并不处理数据的原子性，而是强制对数据的读写及时影响到主内存的。

用图来演示一下使用关键字 volatile 时出现非线程安全的原因。变量在内存中工作的过程如图 2-80 所示。

由上，我们可以得出以下结论。

1）read 和 load 阶段：从主存复制变量到当前线程工作内存；

2）use 和 assign 阶段：执行代码，改变共享变量值；

3）store 和 write 阶段：用工作内存数据刷新主存对应变量的值。

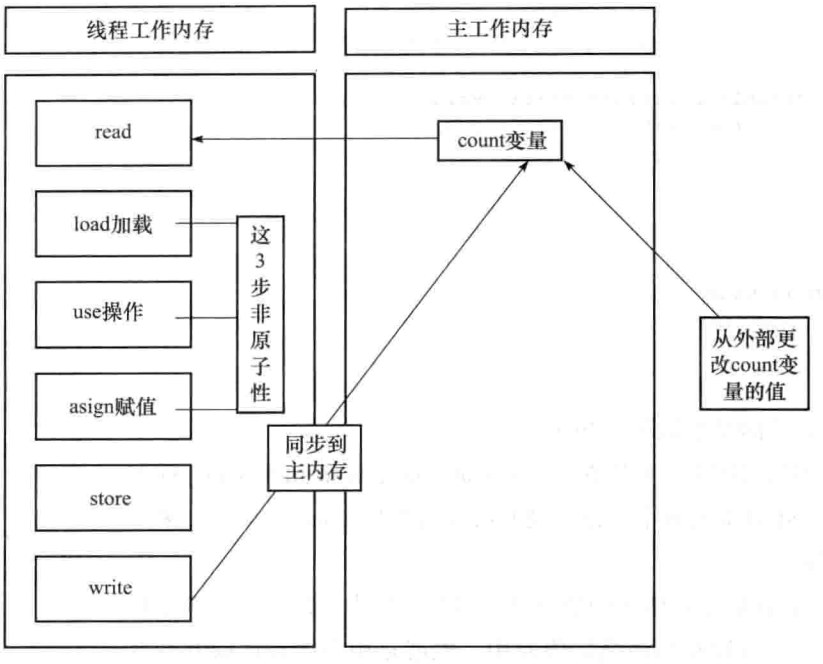


图 2-80 变量在内存中的工作过程

在多线程环境中，use 和 assign 是多次出现的，但这一操作并不是原子性，也就是在 read 和 load 之后，如果主内存 count 变量发生修改之后，线程工作内存中的值由于已经加载，不会产生对应的变化，也就是私有内存和公共内存中的变量不同步，所以计算出来的结果会和预期不一样，也就出现了非线程安全问题。

对于用 volatile 修饰的变量，JVM 虚拟机只是保证从主内存加载到线程工作内存的值是最新的，例如线程 1 和线程 2 在进行 read 和 load 的操作中，发现主内存中 count 的值都是 5，那么都会加载这个最新的值。也就是说，volatile 关键字解决的是变量读时的可见性问题，但无法保证原子性，对于多个线程访问同一个实例变量还是需要加锁同步。

2.3.5 使用原子类进行 i++ 操作

除了在 i++ 操作时使用 synchronized 关键字实现同步外，还可以使用 AtomicInteger 原子类进行实现。

原子操作是不能分割的整体，没有其他线程能够中断或检查正在原子操作中的变量。一个原子（atomic）类型就是一个原子操作可用的类型，它可以在没有锁的情况下做到线程安全（thread-safe）。

创建测试项目 AtomicIntegerTest，文件 AddCountThread.java 代码如下：

```
package extthread;
import java.util.concurrent.atomic.AtomicInteger;
public class AddCountThread extends Thread {
    private AtomicInteger count = new AtomicInteger(0);
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            System.out.println(count.incrementAndGet());
        }
    }
}
```

文件 Run.java 代码如下：

```
package test;
import extthread.AddCountThread;
public class Run {
    public static void main(String[] args) {
        AddCountThread countService = new AddCountThread();
        Thread t1 = new Thread(countService);
        t1.start();
        Thread t2 = new Thread(countService);
        t2.start();
        Thread t3 = new Thread(countService);
        t3.start();
        Thread t4 = new Thread(countService);
        t4.start();
        Thread t5 = new Thread(countService);
        t5.start();
    }
}
```

程序运行后的结果如图 2-81 所示。

成功地累加到 50000。

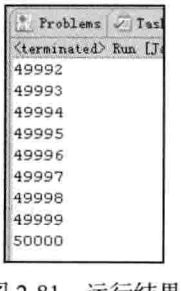


图 2-81　运行结果

2.3.6　原子类也并不完全安全

原子类在具有有逻辑性的情况下输出结果也具有随机性。

创建测试用的项目 atomicIntergerNoSafe，类 MyService.java 代码如下：

```
package service;
import java.util.concurrent.atomic.AtomicLong;
public class MyService {
    public static AtomicLong aiRef = new AtomicLong();
    public void addNum() {
        System.out.println(Thread.currentThread().getName() + "加了 100 之后的值是 :"
                + aiRef.addAndGet(100));
```

```
            aiRef.addAndGet(1);
        }
    }
```

类 MyThread.java 代码如下：

```
package extthread;
import service.MyService;
public class MyThread extends Thread {
    private MyService mySerivce;
    public MyThread(MyService mySerivce) {
        super();
        this.mySerivce = mySerivce;
    }
    @Override
    public void run() {
        mySerivce.addNum();
    }
}
```

类 Run.java 代码如下：

```
package test.run;
import service.MyService;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        try {
            MyService service = new MyService();
            MyThread[] array = new MyThread[5];
            for (int i = 0; i < array.length; i++) {
                array[i] = new MyThread(service);
            }
            for (int i = 0; i < array.length; i++) {
                array[i].start();
            }
            Thread.sleep(1000);
            System.out.println(service.aiRef.get());
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

取消勾选控制台输出的信息数量（Limit console output），如图 2-82 所示。

程序运行后的结果如图 2-83 所示。

打印顺序出错了，应该每加 1 次 100 再加 1 次 1。出现这样的情况是因为 addAndGet()

方法是原子的，但方法和方法之间的调用却不是原子的。解决这样的问题必须要用同步。

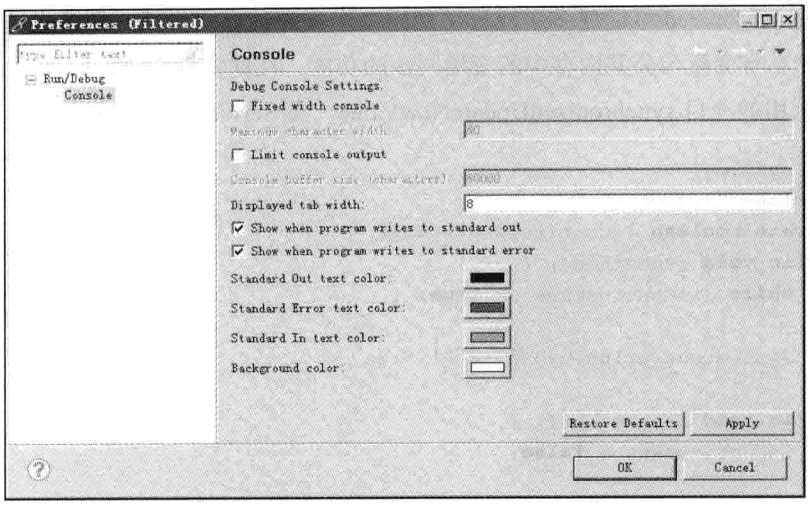


图 2-82　取消勾选 Limit console output

更改 MyService.java 文件代码如下：

```
package service;
import java.util.concurrent.atomic.AtomicLong;
public class MyService {
    public static AtomicLong aiRef = new AtomicLong();
    synchronized public void addNum() {
        System.out.println(Thread.currentThread().getName() + "加了 100 之后的值是 :"
                + aiRef.addAndGet(100));
        aiRef.addAndGet(1);
    }
}
```

程序运行后的结果如图 2-84 所示。

图 2-83　结果值是正确的但顺序是错误

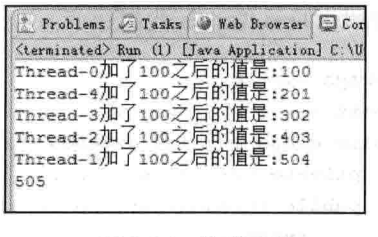


图 2-84　结果正确

从运行结果可以看到，是每次加 100 再加 1，这就是我们想要得到的过程，结果是 505 的同时还保证在过程中累加的顺序也是正确的。

2.3.7 synchronized 代码块有 volatile 同步的功能

关键字 synchronized 可以使多个线程访问同一个资源具有同步性，而且它还具有将线程工作内存中的私有变量与公共内存中的变量同步的功能，在此实验中进行验证。

创建测试用的项目 synchronizedUpdateNewValue，类 Service.java 代码如下：

```
package service;
public class Service {
    private boolean isContinueRun = true;
    public void runMethod() {
        while (isContinueRun == true) {
        }
        System.out.println("停下来了！ ");
    }
    public void stopMethod() {
        isContinueRun = false;
    }
}
```

线程类 ThreadA.java 代码如下：

```
package extthread;
import service.Service;
public class ThreadA extends Thread {
    private Service service;
    public ThreadA(Service service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        service.runMethod();
    }
}
```

线程类 ThreadB.java 代码如下：

```
package extthread;
import service.Service;
public class ThreadB extends Thread {
    private Service service;
    public ThreadB(Service service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
```

```
        service.stopMethod();
    }
}
```

运行类 Run.java 代码如下：

```
package test;
import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        try {
            Service service = new Service();
            ThreadA a = new ThreadA(service);
            a.start();
            Thread.sleep(1000);
            ThreadB b = new ThreadB(service);
            b.start();
            System.out.println("已经发起停止的命令了！");
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

以 –server 服务器模式运行此项目，出现死循环，如图 2-85 所示。

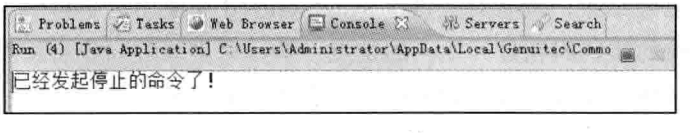


图 2-85　出现死循环

得到这个结果是各线程间的数据值没有可视性造成的，而关键字 synchronized 可以具有可视性。更改 Service.java 代码如下：

```
package service;
public class Service {
    private boolean isContinueRun = true;
    public void runMethod() {
        String anyString = new String();
        while (isContinueRun == true) {
            synchronized (anyString) {
            }
        }
        System.out.println("停下来了！");
    }
```

```
    public void stopMethod() {
        isContinueRun = false;
    }
}
```

再以 -server 服务器模式运行程序后可以正常退出。效果如图 2-86 所示。

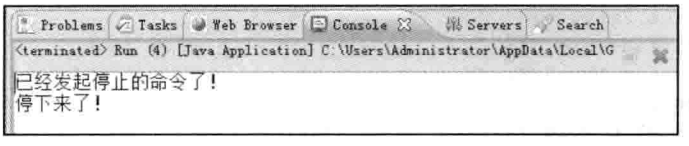


图 2-86 可以正常退出

关键字 synchronized 可以保证在同一时刻，只有一个线程可以执行某一个方法或某一个代码块。它包含两个特征：互斥性和可见性。同步 synchronized 不仅可以解决一个线程看到对象处于不一致的状态，还可以保证进入同步方法或者同步代码块的每个线程，都看到由同一个锁保护之前所有的修改效果。

学习多线程并发，要着重“外练互斥，内修可见”，这是掌握多线程、学习多线程并发的重要技术点。

2.4 本章总结

本章的学习已经结束，读者对关键字 synchronized 在使用上不再陌生，知道什么时候使用它，它所解决的哪些问题是开发上的重点。学习完多线程同步后就可以有效控制线程间处理数据的顺序性，及对处理后的数据进行有效值的保证，更好地对线程执行结果有正确的预期。

第 3 章 Chapter 3

线程间通信

线程是操作系统中独立的个体，但这些个体如果不经过特殊的处理就不能成为一个整体。线程间的通信就是成为整体的必用方案之一，可以说，使线程间进行通信后，系统之间的交互性会更强大，在大大提高 CPU 利用率的同时还会使程序员对各线程任务在处理的过程中进行有效的把控与监督。在本章中需要着重掌握的技术点如下：

- 使用 wait/notify 实现线程间的通信。
- 生产者 / 消费者模式的实现。
- 方法 join 的使用。
- ThreadLocal 类的使用。

3.1 等待 / 通知机制

前面两章介绍了在 Java 语言中多线程的使用，以及对方法及变量在同步情况下的处理方式，本节将介绍多个线程之间进行通信，通过本节的学习可以了解到，线程与线程之间不是独立的个体，它们彼此之间可以互相通信和协作。

3.1.1 不使用等待 / 通知机制实现线程间通信

创建新的项目，名称为 TwoThreadTransData。在实验中使用 sleep() 结合 while(true) 死循环法来实现多个线程间通信。

类 MyList.java 代码如下：

```
package mylist;
import java.util.ArrayList;
import java.util.List;
public class MyList {
private List list = new ArrayList();
public void add() {
    list.add("高洪岩");
}
public int size() {
    return list.size();
}
}
```

线程类 ThreadA.java 代码如下：

```
package extthread;
import mylist.MyList;
public class ThreadA extends Thread {
private MyList list;
public ThreadA(MyList list) {
    super();
    this.list = list;
}
@Override
public void run() {
    try {
        for (int i = 0; i < 10; i++) {
            list.add();
            System.out.println("添加了" + (i + 1) + "个元素");
            Thread.sleep(1000);
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

线程类 ThreadB.java 代码如下：

```
package extthread;
import mylist.MyList;
public class ThreadB extends Thread {
private MyList list;
public ThreadB(MyList list) {
    super();
    this.list = list;
}
@Override
public void run() {
    try {
        while (true) {
```

```
                if (list.size() == 5) {
                    System.out.println("==5了，线程b要退出了！ ");
                    throw new InterruptedException();
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

运行类 Test.java 代码如下：

```
package test;
import mylist.MyList;
import extthread.ThreadA;
import extthread.ThreadB;
public class Test {
public static void main(String[] args) {
    MyList service = new MyList();
    ThreadA a = new ThreadA(service);
    a.setName("A");
    a.start();
    ThreadB b = new ThreadB(service);
    b.setName("B");
    b.start();
}
}
```

程序运行后的效果如图 3-1 所示。

图 3-1　两个线程互相通信成功

虽然两个线程间实现了通信，但有一个弊端就是，线程 ThreadB.java 不停地通过 while 语句轮询机制来检测某一个条件，这样会浪费 CPU 资源。

如果轮询的时间间隔很小，更浪费 CPU 资源；如果轮询的时间间隔很大，有可能会取不到想要得到的数据。所以就需要有一种机制来实现减少 CPU 的资源浪费，而且还可以实现在多个线程间通信，它就是“wait/notify”机制。

3.1.2　什么是等待 / 通知机制

等待 / 通知机制在生活中比比皆是，比如在就餐时就会出现，如图 3-2 所示。

厨师和服务员之间的交互要在“菜品传递台”上，在这期间会有几个问题：

1）厨师做完一道菜的时间不确定，所以厨师将菜品放到“菜品传递台”上的时间也不确定。

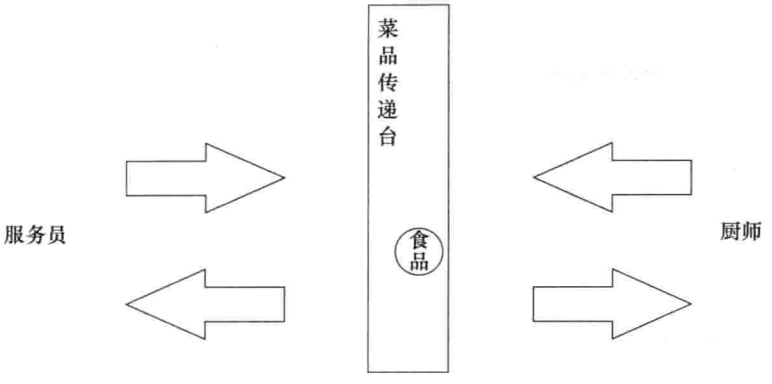


图 3-2 就餐时出现等待通知

2）服务员取到菜的时间取决于厨师，所以服务员就有“等待”(wait）的状态。

3）服务员如何能取到菜呢？这又得取决于厨师，厨师将菜放在“菜品传递台”上，其实就相当于一种通知（notify），这时服务员才可以拿到菜并交给就餐者。

4）在这个过程中出现了“等待 / 通知”机制。

需要说明的是，前面章节中多个线程之间也可以实现通信，原因就是多个线程共同访问同一个变量，但那种通信机制不是“等待 / 通知”，两个线程完全是主动式地读取一个共享变量，在花费读取时间的基础上，读到的值是不是想要的，并不能完全确定。所以现在迫切需要一种“等待 / 通知”机制来满足上面的需求。

3.1.3 等待 / 通知机制的实现

方法 wait() 的作用是使当前执行代码的线程进行等待，wait() 方法是 Object 类的方法，该方法用来将当前线程置入“预执行队列”中，并且在 wait() 所在的代码行处停止执行，直到接到通知或被中断为止。在调用 wait() 之前，线程必须获得该对象的对象级别锁，即只能在同步方法或同步块中调用 wait() 方法。在执行 wait() 方法后，当前线程释放锁。在从 wait() 返回前，线程与其他线程竞争重新获得锁。如果调用 wait() 时没有持有适当的锁，则抛出 IllegalMonitorStateException，它是 RuntimeException 的一个子类，因此，不需要 try-catch 语句进行捕捉异常。

方法 notify() 也要在同步方法或同步块中调用，即在调用前，线程也必须获得该对象的对象级别锁。如果调用 notify() 时没有持有适当的锁，也会抛出 IllegalMonitorStateException。该方法用来通知那些可能等待该对象的对象锁的其他线程，如果有多个线程等待，则由线程规划器随机挑选出其中一个呈 wait 状态的线程，对其发出通知 notify，并使它等待获取该对象的对象锁。需要说明的是，在执行 notify() 方法后，当前线程不会马上释放该对象锁，呈 wait 状态的线程也并不能马上获取该对象锁，要等到执行 notify() 方法的线程将程序执行完，也就是退

出 synchronized 代码块后，当前线程才会释放锁，而呈 wait 状态所在的线程才可以获取该对象锁。当第一个获得了该对象锁的 wait 线程运行完毕以后，它会释放掉该对象锁，此时如果该对象没有再次使用 notify 语句，则即便该对象已经空闲，其他 wait 状态等待的线程由于没有得到该对象的通知，还会继续阻塞在 wait 状态，直到这个对象发出一个 notify 或 notifyAll。

用一句话来总结一下 wait 和 notify：wait 使线程停止运行，而 notify 使停止的线程继续运行。

创建测试用的 Java 项目，名称为 test1。类 Test1.java 代码如下：

```
package test;
public class Test1 {
public static void main(String[] args) {
    try {
        String newString = new String("");
        newString.wait();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

程序运行后的效果如图 3-3 所示。

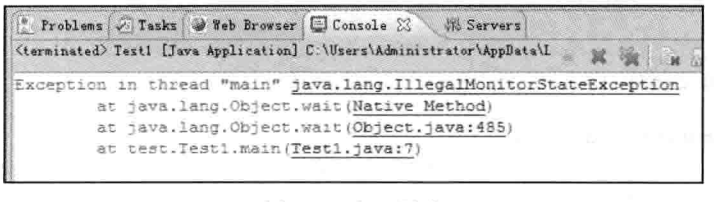


图 3-3　出现异常

出现异常的原因是没有“对象监视器”，也就是没有同步加锁。

继续创建 Test2.java 文件，代码如下：

```
package test;
public class Test2 {
public static void main(String[] args) {
    try {
        String lock = new String();
        System.out.println("syn 上面 ");
        synchronized (lock) {
            System.out.println("syn 第一行 ");
            lock.wait();
            System.out.println("wait 下的代码 !");
        }
        System.out.println("syn 下面的代码 ");
    } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

程序运行后效果如图 3-4 所示。

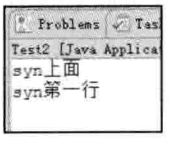


图 3-4 方法 wait 下面的代码不执行了

但线程不能永远等待下去，那样程序就停止不前，不继续向下运行了。如何使呈等待 wait 状态的线程继续运行呢？答案就是使用 notify() 方法。

继续创建实验用的项目，名称为 test2。类 MyThread1.java 代码如下：

```
package extthread;
public class MyThread1 extends Thread {
private Object lock;
public MyThread1(Object lock) {
    super();
    this.lock = lock;
}
@Override
public void run() {
    try {
        synchronized (lock) {
            System.out.println("开始      wait time=" + System.currentTimeMillis());
            lock.wait();
            System.out.println("结束      wait time=" + System.currentTimeMillis());
        }
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
}
```

类 MyThread2.java 代码如下：

```
package extthread;
public class MyThread2 extends Thread {
private Object lock;
public MyThread2(Object lock) {
    super();
    this.lock = lock;
}
@Override
```

```
public void run() {
    synchronized (lock) {
        System.out.println("开始 notify time=" + System.currentTimeMillis());
        lock.notify();
        System.out.println("结束 notify time=" + System.currentTimeMillis());
    }
}
}
```

类 Test.java 代码如下：

```
package test;
import extthread.MyThread1;
import extthread.MyThread2;
public class Test {
public static void main(String[] args) {
    try {
        Object lock = new Object();
        MyThread1 t1 = new MyThread1(lock);
        t1.start();
        Thread.sleep(3000);
        MyThread2 t2 = new MyThread2(lock);
        t2.start();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

程序运行的效果如图 3-5 所示。

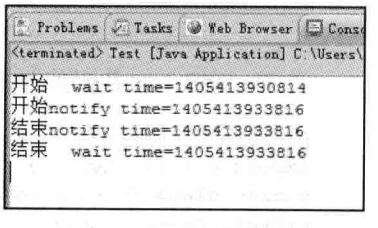


图 3-5　使用等待 / 通知方法的示例

从控制台打印的结果来看，3 秒后线程被 notify 通知唤醒。

如何使用 wait() 与 notify() 来实现前面 size() 值等于 5 的实验呢？创建新的项目 wait_notify_size5。类 MyList.java 代码如下：

```
package extlist;
import java.util.ArrayList;
import java.util.List;
public class MyList {
private static List list = new ArrayList();
public static void add() {
    list.add("anyString");
}
public static int size() {
    return list.size();
}
}
```

类 ThreadA.java 代码如下：

```
package extthread;
import extlist.MyList;
public class ThreadA extends Thread {
private Object lock;
public ThreadA(Object lock) {
    super();
    this.lock = lock;
}
@Override
public void run() {
    try {
        synchronized (lock) {
            if (MyList.size() != 5) {
                System.out.println("wait begin "
                        + System.currentTimeMillis());
                lock.wait();
                System.out.println("wait end  "
                        + System.currentTimeMillis());
            }
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

类 ThreadB.java 代码如下：

```
package extthread;
import extlist.MyList;
public class ThreadB extends Thread {
private Object lock;
public ThreadB(Object lock) {
    super();
    this.lock = lock;
}
@Override
public void run() {
    try {
        synchronized (lock) {
            for (int i = 0; i < 10; i++) {
                MyList.add();
                if (MyList.size() == 5) {
                    lock.notify();
                    System.out.println("已发出通知！");
                }
                System.out.println("添加了 " + (i + 1) + " 个元素！");
                Thread.sleep(1000);
            }
        }
    } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

类 Run.java 代码如下：

```
package test;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
public static void main(String[] args) {
    try {
        Object lock = new Object();
        ThreadA a = new ThreadA(lock);
        a.start();
        Thread.sleep(50);
        ThreadB b = new ThreadB(lock);
        b.start();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

程序运行结果如图 3-6 所示。

图 3-6 运行结果

日志信息 wait end 在最后输出，这也说明 notify() 方法执行后并不立即释放锁。这个知识点在后面进行补充介绍。

关键字 synchronized 可以将任何一个 Object 对象作为同步对象来看待，而 Java 为每个 Object 都实现了 wait() 和 notify() 方法，它们必须用在被 synchronized 同步的 Object 的临界区内。通过调用 wait() 方法可以使处于临界区内的线程进入等待状态，同时释放被同步对象的锁。而 notify 操作可以唤醒一个因调用了 wait 操作而处于阻塞状态中的线程，使其进入就绪状态。被重新唤醒的线程会试图重新获得临界区的控制权，也就是锁，并继续执行临界区内 wait 之后的代码。如果发出 notify 操作时没有处于阻塞状态中的线程，那么该命令会被忽略。

wait() 方法可以使调用该方法的线程释放共享资源的锁，然后从运行状态退出，进入等待队列，直到被再次唤醒。

notify() 方法可以随机唤醒等待队列中等待同一共享资源的“一个”线程，并使该线程退出等待队列，进入可运行状态，也就是 notify() 方法仅通知“一个”线程。

notifyAll() 方法可以使所有正在等待队列中等待同一共享资源的“全部”线程从等待状态退出，进入可运行状态。此时，优先级最高的那个线程最先执行，但也有可能是随机执

行，因为这要取决于 JVM 虚拟机的实现。

在前面的章节中已经介绍了与 Thread 有关的大部分 API，这些 API 可以改变线程对象的状态，如图 3-7 所示。

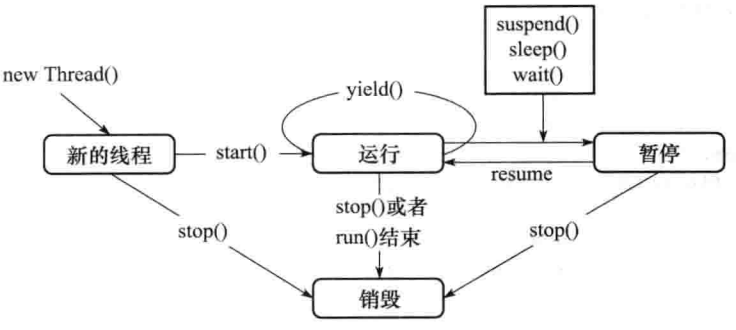


图 3-7 线程状态切换示意图

1）新创建一个新的线程对象后，再调用它的 start() 方法，系统会为此线程分配 CPU 资源，使其处于 Runnable（可运行）状态，这是一个准备运行的阶段。如果线程抢占到 CPU 资源，此线程就处于 Running（运行）状态。

2）Runnable 状态和 Running 状态可相互切换，因为有可能线程运行一段时间后，有其他高优先级的线程抢占了 CPU 资源，这时此线程就从 Running 状态变成 Runnable 状态。

线程进入 Runnable 状态大体分为如下 5 种情况：

- 调用 sleep() 方法后经过的时间超过了指定的休眠时间。
- 线程调用的阻塞 IO 已经返回，阻塞方法执行完毕。
- 线程成功地获得了试图同步的监视器。
- 线程正在等待某个通知，其他线程发出了通知。
- 处于挂起状态的线程调用了 resume 恢复方法。

3）Blocked 是阻塞的意思，例如遇到了一个 IO 操作，此时 CPU 处于空闲状态，可能会转而把 CPU 时间片分配给其他线程，这时也可以称为“暂停”状态。Blocked 状态结束后，进入 Runnable 状态，等待系统重新分配资源。

出现阻塞的情况大体分为如下 5 种：

- 线程调用 sleep 方法，主动放弃占用的处理器资源。
- 线程调用了阻塞式 IO 方法，在该方法返回前，该线程被阻塞。
- 线程试图获得一个同步监视器，但该同步监视器正被其他线程所持有。
- 线程等待某个通知。
- 程序调用了 suspend 方法将该线程挂起。此方法容易导致死锁，尽量避免使用该方法。

4）run() 方法运行结束后进入销毁阶段，整个线程执行完毕。

每个锁对象都有两个队列，一个是就绪队列，一个是阻塞队列。就绪队列存储了将要获得锁的线程，阻塞队列存储了被阻塞的线程。一个线程被唤醒后，才会进入就绪队列，等待 CPU 的调度；反之，一个线程被 wait 后，就会进入阻塞队列，等待下一次被唤醒。

3.1.4　方法 wait() 锁释放与 notify() 锁不释放

当方法 wait() 被执行后，锁被自动释放，但执行完 notify() 方法，锁却不自动释放。

创建实验用的项目，名称为 waitReleaseLock，类 Service.java 代码如下：

```
package service;
public class Service {
public void testMethod(Object lock) {
    try {
        synchronized (lock) {
            System.out.println("begin wait()");
            lock.wait();
            System.out.println("  end wait()");
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

两个自定义线程类如图 3-8 所示。

ThreadA.java

```
1  package extthread;
2
3  import service.Service;
4
5  public class ThreadA extends Thread {
6
7      private Object lock;
8
9      public ThreadA(Object lock) {
10         super();
11         this.lock = lock;
12     }
13
14     @Override
15     public void run() {
16         Service service = new Service();
17         service.testMethod(lock);
18     }
19
20 }
21
```

ThreadB.java

```
1  package extthread;
2
3  import service.Service;
4
5  public class ThreadB extends Thread {
6      private Object lock;
7
8      public ThreadB(Object lock) {
9          super();
10         this.lock = lock;
11     }
12
13     @Override
14     public void run() {
15         Service service = new Service();
16         service.testMethod(lock);
17     }
18
19 }
20
```

图 3-8　两个自定义线程类

运行类 Test.java 代码如下：

```
package test;
import extthread.ThreadA;
```

```
import extthread.ThreadB;
public class Test {
public static void main(String[] args) {
    Object lock = new Object();
    ThreadA a = new ThreadA(lock);
    a.start();
    ThreadB b = new ThreadB(lock);
    b.start();
}
}
```

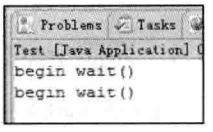


图 3-9 方法 wait() 自动释放锁

程序运行的效果如图 3-9 所示。

如果将 wait() 方法改成 sleep() 方法，就成了同步的效果，如图 3-10 所示。

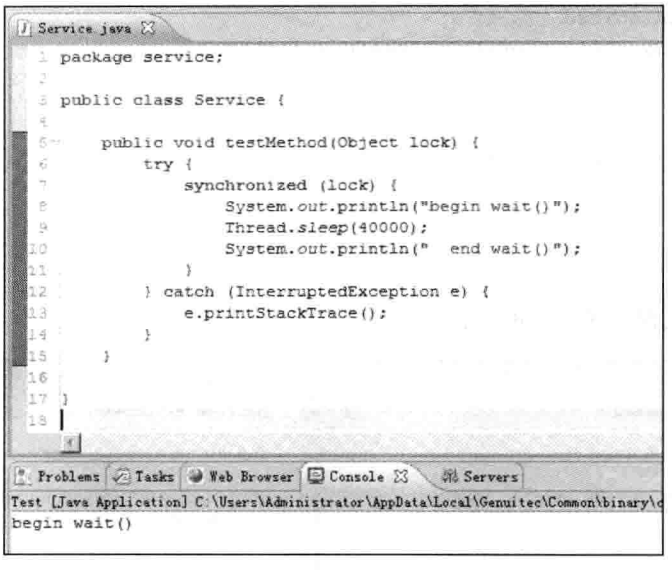


图 3-10 sleep 方法不释放锁

还有一个结论要进行实验：方法 notify() 被执行后，不释放锁。

验证这个结论，创建新的项目 notifyHoldLock，其中类 Service.java 代码如下：

```
package service;
public class Service {
public void testMethod(Object lock) {
    try {
        synchronized (lock) {
            System.out.println("begin wait() ThreadName="
                    + Thread.currentThread().getName());
            lock.wait();
            System.out.println("  end wait() ThreadName="
                    + Thread.currentThread().getName());
```

```
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
public void synNotifyMethod(Object lock) {
    try {
        synchronized (lock) {
            System.out.println("begin notify() ThreadName="
                    + Thread.currentThread().getName() + " time="
                    + System.currentTimeMillis());
            lock.notify();
            Thread.sleep(5000);
            System.out.println("  end notify() ThreadName="
                    + Thread.currentThread().getName() + " time="
                    + System.currentTimeMillis());
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

类 ThreadA.java 代码如下：

```
package extthread;
import service.Service;
public class ThreadA extends Thread {
private Object lock;
public ThreadA(Object lock) {
    super();
    this.lock = lock;
}
@Override
public void run() {
    Service service = new Service();
    service.testMethod(lock);
}
}
```

类 NotifyThread.java 代码如下：

```
package extthread;
import service.Service;
public class NotifyThread extends Thread {
private Object lock;
public NotifyThread(Object lock) {
    super();
    this.lock = lock;
}
@Override
```

```
public void run() {
    Service service = new Service();
    service.synNotifyMethod(lock);
}
}
```

类 synNotifyMethodThread.java 代码如下：

```
package extthread;
import service.Service;
public class synNotifyMethodThread extends Thread {
private Object lock;
public synNotifyMethodThread(Object lock) {
    super();
    this.lock = lock;
}
@Override
public void run() {
    Service service = new Service();
    service.synNotifyMethod(lock);
}
}
```

类 Test.java 代码如下：

```
package test;
import extthread.NotifyThread;
import extthread.ThreadA;
import extthread.synNotifyMethodThread;
public class Test {
public static void main(String[] args) throws InterruptedException {
    Object lock = new Object();
    ThreadA a = new ThreadA(lock);
    a.start();
    NotifyThread notifyThread = new NotifyThread(lock);
    notifyThread.start();
    synNotifyMethodThread c = new synNotifyMethodThread(lock);
    c.start();
}
}
```

程序运行结果如图 3-11 所示。

此实验说明，必须执行完 notify() 方法所在的同步 synchronized 代码块后才释放锁。

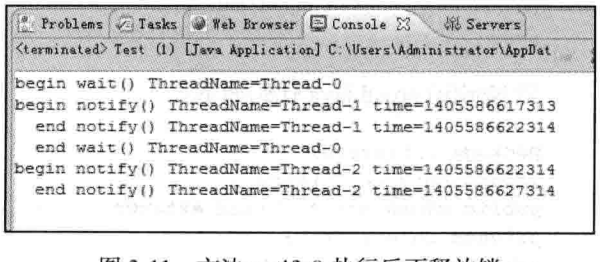


图 3-11 方法 notify() 执行后不释放锁

3.1.5 当 interrupt 方法遇到 wait 方法

当线程呈 wait() 状态时，调用线程对象的 interrupt() 方法会出现 InterruptedException 异常。

创建测试用的项目 waitInterruptException，类 Service.java 代码如下：

```
package service;
public class Service {
public void testMethod(Object lock) {
    try {
        synchronized (lock) {
            System.out.println("begin wait()");
            lock.wait();
            System.out.println("  end wait()");
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
        System.out.println("出现异常了，因为呈 wait 状态的线程被 interrupt 了！");
    }
}
}
```

类 ThreadA.java 代码如下：

```
package extthread;
import service.Service;
public class ThreadA extends Thread {
private Object lock;
public ThreadA(Object lock) {
    super();
    this.lock = lock;
}
@Override
public void run() {
    Service service = new Service();
    service.testMethod(lock);
}
}
```

运行类 Test.java 代码如下：

```
package test;
import extthread.ThreadA;
public class Test {
public static void main(String[] args) {
    try {
        Object lock = new Object();
        ThreadA a = new ThreadA(lock);
        a.start();
        Thread.sleep(5000);
        a.interrupt();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

程序运行的效果如图 3-12 所示。

通过上面的几个实验可以总结如下 3 点：

1）执行完同步代码块就会释放对象的锁。

2）在执行同步代码块的过程中，遇到异常而导致线程终止，锁也会被释放。

3）在执行同步代码块的过程中，执行了锁所属对象的 wait() 方法，这个线程会释放对象锁，而此线程对象会进入线程等待池中，等待被唤醒。

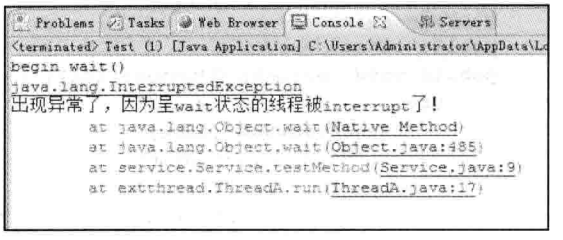


图 3-12 停止 wait 状态下的线程出现异常

3.1.6 只通知一个线程

调用方法 notify() 一次只随机通知一个线程进行唤醒。

创建测试用的项目 notifyOne，其中类 Service.java 代码如下：

```
package service;
public class Service {
public void testMethod(Object lock) {
    try {
        synchronized (lock) {
            System.out.println("begin wait() ThreadName="
                    + Thread.currentThread().getName());
            lock.wait();
            System.out.println("  end wait() ThreadName="
                    + Thread.currentThread().getName());
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

3 个自定义线程，每个线程都呈 wait 状态，代码如图 3-13 所示。

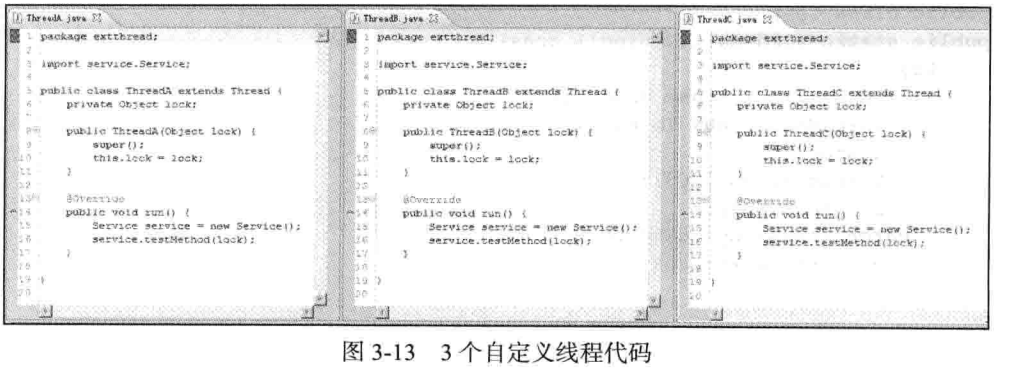


图 3-13 3 个自定义线程代码

创建唤醒线程 NotifyThread.java，代码如下：

```
package extthread;
import service.Service;
public class NotifyThread extends Thread {
private Object lock;
public NotifyThread(Object lock) {
    super();
    this.lock = lock;
}
@Override
public void run() {
    synchronized (lock) {
        lock.notify();
    }
}
}
```

运行类 Test.java 代码如下：

```
package test;
import extthread.NotifyThread;
import extthread.ThreadA;
import extthread.ThreadB;
import extthread.ThreadC;
public class Test {
public static void main(String[] args) throws InterruptedException {
    Object lock = new Object();
    ThreadA a = new ThreadA(lock);
    a.start();
    ThreadB b = new ThreadB(lock);
    b.start();
    ThreadC c = new ThreadC(lock);
    c.start();
    Thread.sleep(1000);
    NotifyThread notifyThread = new NotifyThread(lock);
    notifyThread.start();
}
}
```

程序运行的效果如图 3-14 所示。

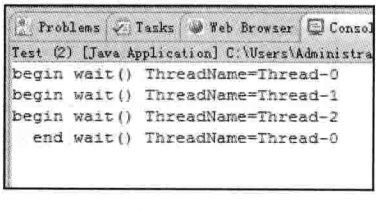


图 3-14　仅有一个线程被唤醒

方法 notify() 仅随机唤醒一个线程。

当多次调用 notify() 方法时，会随机将等待 wait 状态的线程进行唤醒。更改 NotifyThread.java 类代码如下：

```
package extthread;
import service.Service;
public class NotifyThread extends Thread {
private Object lock;
```

```
public NotifyThread(Object lock) {
    super();
    this.lock = lock;
}
@Override
public void run() {
    synchronized (lock) {
        lock.notify();
        lock.notify();
        lock.notify();
        lock.notify();
        lock.notify();
        lock.notify();
        lock.notify();
        lock.notify();
        lock.notify();
    }
}
}
```

程序运行的效果如图 3-15 所示。

多次调用 notify() 方法唤醒了全部 WAITING 中的线程。

3.1.7 唤醒所有线程

前面示例中通过多次调用 notify() 方法来实现唤醒 3 个线程，但并不能保证系统中仅有 3 个线程，也就是若 notify() 方法的调用次数小于线程对象的数量，会出现有部分线程对象无法被唤醒的情况。为了唤醒全部线程，可以使用 notifyAll() 方法。

创建测试用的项目 notifyAll，将 notifyOne 项目中的所有文件复制到 notifyAll 项目中，只需要更改 NotifyThread.java 类使用的方法为 notifyAll() 即可。程序运行后的效果如图 3-16 所示。

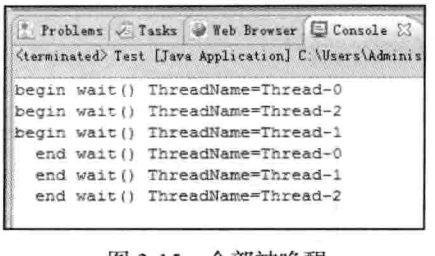


图 3-15 全部被唤醒

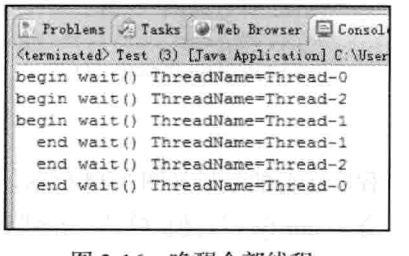


图 3-16 唤醒全部线程

3.1.8 方法 wait(long) 的使用

带一个参数的 wait(long) 方法的功能是等待某一时间内是否有线程对锁进行唤醒，如果超过这个时间则自动唤醒。

创建测试用的项目 waitHasParamMethod，其中 MyRunnable.java 类代码如下：

```
package myrunnable;
public class MyRunnable {
static private Object lock = new Object();
static private Runnable runnable1 = new Runnable() {
    @Override
    public void run() {
        try {
            synchronized (lock) {
                System.out.println("wait begin timer="
                        + System.currentTimeMillis());
                lock.wait(5000);
                System.out.println("wait   end timer="
                        + System.currentTimeMillis());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
};
public static void main(String[] args) {
    Thread t = new Thread(runnable1);
    t.start();
}
}
```

程序运行的效果如图 3-17 所示。

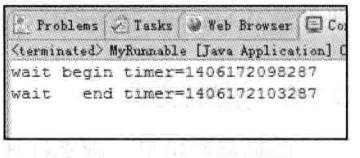


图 3-17 5 秒后自动被唤醒

当然也可以在 5 秒内由其他线程进行唤醒。

代码更改如下：

```
package myrunnable;
public class MyRunnable {
static private Object lock = new Object();
static private Runnable runnable1 = new Runnable() {
    @Override
    public void run() {
        try {
            synchronized (lock) {
                System.out.println("wait begin timer="
                        + System.currentTimeMillis());
                lock.wait(5000);
                System.out.println("wait   end timer="
                        + System.currentTimeMillis());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

```
};
static private Runnable runnable2 = new Runnable() {
    @Override
    public void run() {
        synchronized (lock) {
            System.out.println("notify begin timer="
                    + System.currentTimeMillis());
            lock.notify();
            System.out.println("notify   end timer="
                    + System.currentTimeMillis());
        }
    }
};
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(runnable1);
    t1.start();
    Thread.sleep(3000);
    Thread t2 = new Thread(runnable2);
    t2.start();
}
}
```

程序运行结果如图 3-18 所示。

打印日志中 wait begin 的时间尾数为 3879，在 3000 毫秒后，notify begin 6879 被执行，也就是在此时间点准备对呈 WAITING 状态的线程进行唤醒。

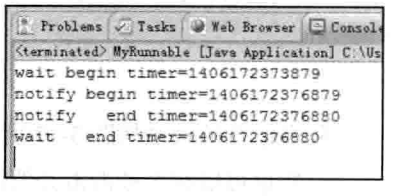


图 3-18　3 秒后由其他线程唤醒

3.1.9　通知过早

如果通知过早，则会打乱程序正常的运行逻辑。

创建测试用的项目 firstNotify，其中类 MyRun.java 代码如下：

```
package test;
public class MyRun {
private String lock = new String("");
private Runnable runnableA = new Runnable() {
    @Override
    public void run() {
        try {
            synchronized (lock) {
                System.out.println("begin wait");
                lock.wait();
                System.out.println("end wait");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

```
};
private Runnable runnableB = new Runnable() {
    @Override
    public void run() {
        synchronized (lock) {
            System.out.println("begin notify");
            lock.notify();
            System.out.println("end notify");
        }
    }
};
public static void main(String[] args) {
    MyRun run = new MyRun();
    Thread a = new Thread(run.runnableA);
    a.start();
    Thread b = new Thread(run.runnableB);
    b.start();
}
}
```

程序运行结果如图 3-19 所示。

如果将 main 方法中的代码改成如下：

```
public static void main(String[] args) throws InterruptedException {
    MyRun run = new MyRun();
    Thread b = new Thread(run.runnableB);
    b.start();
    Thread.sleep(100);
    Thread a = new Thread(run.runnableA);
    a.start();
}
```

程序运行结果如图 3-20 所示。

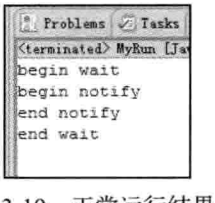


图 3-19 正常运行结果

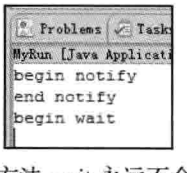


图 3-20 方法 wait 永远不会被通知

如果先通知了，则 wait 方法也就没有必要执行了。更改 MyRun.java 代码如下：

```
package test;
public class MyRun {
private String lock = new String("");
private boolean isFirstRunB = false;
private Runnable runnableA = new Runnable() {
    @Override
    public void run() {
```

```
            try {
                synchronized (lock) {
                    while (isFirstRunB == false) {
                        System.out.println("begin wait");
                        lock.wait();
                        System.out.println("end wait");
                    }
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    };
    private Runnable runnableB = new Runnable() {
        @Override
        public void run() {
            synchronized (lock) {
                System.out.println("begin notify");
                lock.notify();
                System.out.println("end notify");
                isFirstRunB = true;
            }
        }
    };
    public static void main(String[] args) throws InterruptedException {
        MyRun run = new MyRun();
        Thread b = new Thread(run.runnableB);
        b.start();
        Thread.sleep(100);
        Thread a = new Thread(run.runnableA);
        a.start();
    }
}
```

程序运行结果如图 3-21 所示。

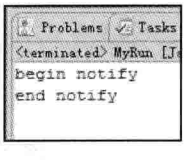


图 3-21 仅仅执行了 notify 方法

继续将上面程序中的 main 方法代码更改如下：

```
public static void main(String[] args) throws InterruptedException {
    MyRun run = new MyRun();
    Thread a = new Thread(run.runnableA);
    a.start();
    Thread.sleep(100);
    Thread b = new Thread(run.runnableB);
```

```
        b.start();
    }
```

更改后运行结果就是正确的了，如图 3-22 所示。

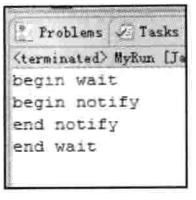


图 3-22　正确的结果

3.1.10　等待 wait 的条件发生变化

在使用 wait/notify 模式时，还需要注意另外一种情况，也就是 wait 等待的条件发生了变化，也容易造成程序逻辑的混乱。

创建测试用的项目，名称为 waitOld，创建类 Add.java，代码如下：

```
package entity;
//加法
public class Add {
private String lock;
public Add(String lock) {
    super();
    this.lock = lock;
}
public void add() {
    synchronized (lock) {
        ValueObject.list.add("anyString");
        lock.notifyAll();
    }
}
}
```

创建类 Subtract.java，代码如下：

```
package entity;
//减法
public class Subtract {
private String lock;
public Subtract(String lock) {
    super();
    this.lock = lock;
}
public void subtract() {
    try {
        synchronized (lock) {
```

```
            if (ValueObject.list.size() == 0) {
                System.out.println("wait begin ThreadName="
                        + Thread.currentThread().getName());
                lock.wait();
                System.out.println("wait   end ThreadName="
                        + Thread.currentThread().getName());
            }
            ValueObject.list.remove(0);
            System.out.println("list size=" + ValueObject.list.size());
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

类 ValueObject.java 代码如下：

```
package entity;
import java.util.ArrayList;
import java.util.List;
public class ValueObject {
public static List list = new ArrayList();
}
```

两个线程类代码如图 3-23 所示。

ThreadAdd.java

```
package extthread;

import entity.Add;

public class ThreadAdd extends Thread {

    private Add p;

    public ThreadAdd(Add p) {
        super();
        this.p = p;
    }

    @Override
    public void run() {
        p.add();
    }

}
```

ThreadSubtract.java

```
package extthread;

import entity.Subtract;

public class ThreadSubtract extends Thread {

    private Subtract r;

    public ThreadSubtract(Subtract r) {
        super();
        this.r = r;
    }

    @Override
    public void run() {
        r.subtract();
    }

}
```

图 3-23　两个线程类代码

类 Run.java 代码如下：

```
package test;
import entity.Add;
import entity.Subtract;
import extthread.ThreadAdd;
```

```
import extthread.ThreadSubtract;
public class Run {
public static void main(String[] args) throws InterruptedException {
    String lock = new String("");
    Add add = new Add(lock);
    Subtract subtract = new Subtract(lock);
    ThreadSubtract subtract1Thread = new ThreadSubtract(subtract);
    subtract1Thread.setName("subtract1Thread");
    subtract1Thread.start();
    ThreadSubtract subtract2Thread = new ThreadSubtract(subtract);
    subtract2Thread.setName("subtract2Thread");
    subtract2Thread.start();
    Thread.sleep(1000);
    ThreadAdd addThread = new ThreadAdd(add);
    addThread.setName("addThread");
    addThread.start();
}
}
```

程序运行结果如图 3-24 所示。

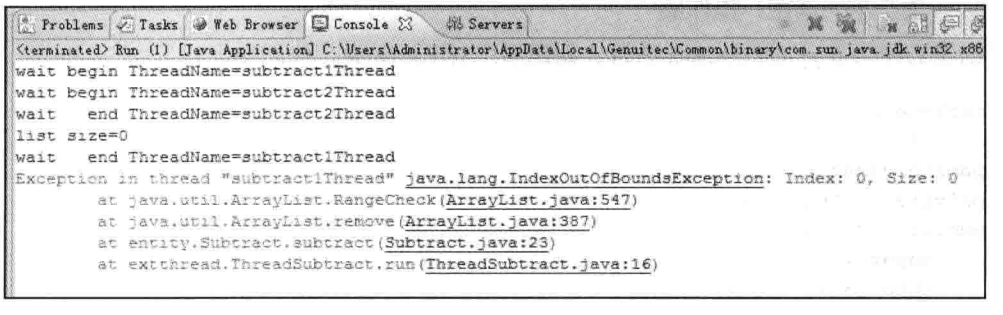


图 3-24 运行结果异常

出现这样异常的原因是因为有两个实现删除 remove() 操作的线程，它们在 Thread.sleep(1000)；之前都执行了 wait() 方法，呈等待状态，当加操作的线程在 1 秒之后被运行时，通知了所有呈 wait 等待状态的减操作的线程，那么第一个实现减操作的线程能正确地删除 list 中索引为 0 的数据，但第二个实现减操作的线程则出现索引溢出的异常，因为 list 中仅仅添加了一个数据，也只能删除一个数据，所以没有第二个数据可供删除。如何解决这样的问题呢?

更改 Subtract.java 中的 subtract 方法，代码如下：

```
public void subtract() {
    try {
        synchronized (lock) {
            while (ValueObject.list.size() == 0) {
                System.out.println("wait begin ThreadName="
                        + Thread.currentThread().getName());
                lock.wait();
                System.out.println("wait   end ThreadName="
```

```
                    + Thread.currentThread().getName());
            }
            ValueObject.list.remove(0);
            System.out.println("list size=" + ValueObject.list.size());
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

程序运行结果如图 3-25 所示。

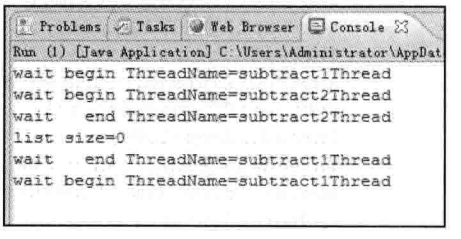


图 3-25 问题解决了

3.1.11 生产者 / 消费者模式实现

等待 / 通知模式最经典的案例就是“生产者 / 消费者”模式。但此模式在使用上有几种“变形”，还有一些小的注意事项，但原理都是基于 wait/notify 的。

1. 一生产与一消费：操作值

创建名称为 p_r_test 的 Java 项目，创建生产者 P.java 类，代码如下：

```
package entity;
// 生产者
public class P {
private String lock;
public P(String lock) {
    super();
    this.lock = lock;
}
public void setValue() {
    try {
        synchronized (lock) {
            if (!ValueObject.value.equals("")) {
                lock.wait();
            }
            String value = System.currentTimeMillis() + "_"
                    + System.nanoTime();
            System.out.println("set 的值是 " + value);
            ValueObject.value = value;
            lock.notify();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

创建消费者类 C.java，代码如下：

```
package entity;
//消费者
public class C {
private String lock;
public C(String lock) {
    super();
    this.lock = lock;
}
public void getValue() {
    try {
        synchronized (lock) {
            if (ValueObject.value.equals("")) {
                lock.wait();
            }
            System.out.println("get 的值是 " + ValueObject.value);
            ValueObject.value = "";
            lock.notify();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

创建存储值的对象 ValueObject.java，代码如下：

```
package entity;
public class ValueObject {
public static String value = "";
}
```

创建两个线程对象，一个是生产者线程，另外一个是消费者线程，代码如图 3-26 所示。

ThreadP.java

```
package extthread;

import entity.P;

public class ThreadP extends Thread {

    private P p;

    public ThreadP(P p) {
        super();
        this.p = p;
    }

    @Override
    public void run() {
        while (true) {
            p.setValue();
        }
    }

}
```

ThreadC.java

```
package extthread;

import entity.C;

public class ThreadC extends Thread {

    private C r;

    public ThreadC(C r) {
        super();
        this.r = r;
    }

    @Override
    public void run() {
        while (true) {
            r.getValue();
        }
    }

}
```

图 3-26　两个线程代码

运行类 Run.java 代码如下：

```
package test;
import entity.P;
import entity.C;
import extthread.ThreadP;
import extthread.ThreadC;
public class Run {
public static void main(String[] args) {
    String lock = new String("");
    P p = new P(lock);
    C r = new C(lock);
    ThreadP pThread = new ThreadP(p);
    ThreadC rThread = new ThreadC(r);
    pThread.start();
    rThread.start();
}
}
```

程序运行结果如图 3-27 所示。

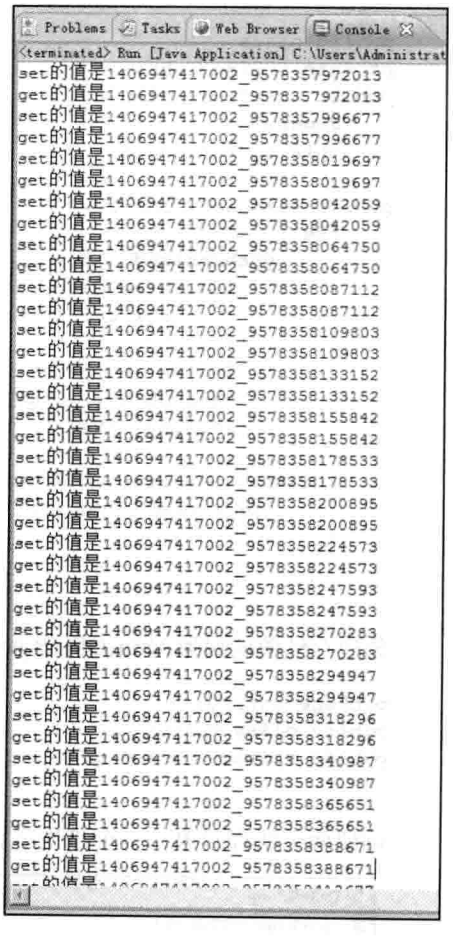


图 3-27 部分打印结果

本示例是 1 个生产者和 1 个消费者进行数据的交互，在控制台中打印的日志 get 和 set 是交替运行的。

但如果在此实验的基础上，设计出多个生产者和多个消费者，那么在运行的过程中极有可能出现“假死”的情况，也就是所有的线程都呈 WAITING 等待状态。

2. 多生产与多消费：操作值 - 假死

“假死”的现象其实就是线程进入 WAITING 等待状态。如果全部线程都进入 WAITING 状态，则程序就不再执行任何业务功能了，整个项目呈停止状态。这在使用生产者与消费者模式时经常遇到。

创建 Java 项目 p_c_allWait，类 P.java 代码如下：

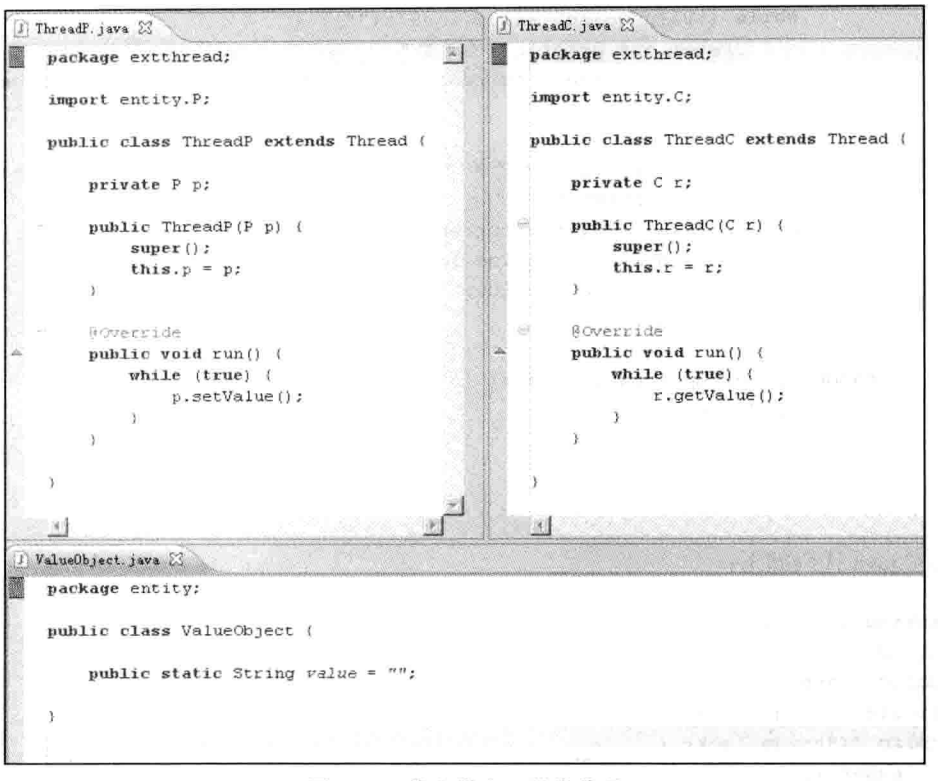
```
package entity;
//生产者
public class P {
private String lock;
public P(String lock) {
    super();
    this.lock = lock;
}
public void setValue() {
    try {
        synchronized (lock) {
```

```
            while (!ValueObject.value.equals("")) {
                System.out.println("生产者 "
                        + Thread.currentThread().getName() + " WAITING了★ ");
                lock.wait();
            }
            System.out.println("生产者 " + Thread.currentThread().getName()
                    + " RUNNABLE了 ");
            String value = System.currentTimeMillis() + "_"
                    + System.nanoTime();
            ValueObject.value = value;
            lock.notify();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

类 C.java 代码如下：

```
package entity;
//消费者
public class C {
private String lock;
public C(String lock) {
    super();
    this.lock = lock;
}
public void getValue() {
    try {
        synchronized (lock) {
            while (ValueObject.value.equals("")) {
                System.out.println("消费者 "
                        + Thread.currentThread().getName() + " WAITING了☆ ");
                lock.wait();
            }
            System.out.println("消费者 " + Thread.currentThread().getName()
                    + " RUNNABLE了 ");
            ValueObject.value = "";
            lock.notify();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

线程类和工具类代码如图 3-28 所示。



ThreadP.java

```
package extthread;

import entity.P;

public class ThreadP extends Thread {

    private P p;

    public ThreadP(P p) {
        super();
        this.p = p;
    }

    @Override
    public void run() {
        while (true) {
            p.setValue();
        }
    }

}
```

ThreadC.java

```
package extthread;

import entity.C;

public class ThreadC extends Thread {

    private C r;

    public ThreadC(C r) {
        super();
        this.r = r;
    }

    @Override
    public void run() {
        while (true) {
            r.getValue();
        }
    }

}
```

ValueObject.java

```
package entity;

public class ValueObject {

    public static String value = "";

}
```

图 3-28 线程类和工具类代码

类 Run.java 代码如下：

```
package test;
import entity.P;
import entity.C;
import extthread.ThreadP;
import extthread.ThreadC;
public class Run {
public static void main(String[] args) throws InterruptedException {
    String lock = new String("");
    P p = new P(lock);
    C r = new C(lock);
    ThreadP[] pThread = new ThreadP[2];
    ThreadC[] rThread = new ThreadC[2];
    for (int i = 0; i < 2; i++) {
        pThread[i] = new ThreadP(p);
        pThread[i].setName("生产者" + (i + 1));
        rThread[i] = new ThreadC(r);
        rThread[i].setName("消费者" + (i + 1));
        pThread[i].start();
        rThread[i].start();
    }
```

```
        Thread.sleep(5000);
        Thread[] threadArray = new Thread[Thread.currentThread()
                .getThreadGroup().activeCount()];
        Thread.currentThread().getThreadGroup().enumerate(threadArray);
        for (int i = 0; i < threadArray.length; i++) {
            System.out.println(threadArray[i].getName() + " "
                    + threadArray[i].getState());
        }
    }
}
```

程序运行后很有可能出现假死状态，如图 3-29 所示。

从打印的信息来看，呈假死状态的进程中所有的线程都呈 WAITING 状态。为什么会出现这样的情况呢？在代码中已经用了 wait/notify 啊？

在代码中确实已经通过 wait/notify 进行通信了，但不保证 notify 唤醒的是异类，也许是同类，比如“生产者”唤醒“生产者”，或“消费者”唤醒“消费者”这样的情况。如果按这样情况运行的比率积少成多，就会导致所有的线程都不能继续运行下去，大家都在等待，都呈 WAITING 状态，程序最后也就呈“假死”状态，不能继续运行下去了。

那么假死出现的具体过程是怎么样的呢？将控制台中的日志复制到 EditPlus 中以显示行号，如图 3-30 所示。

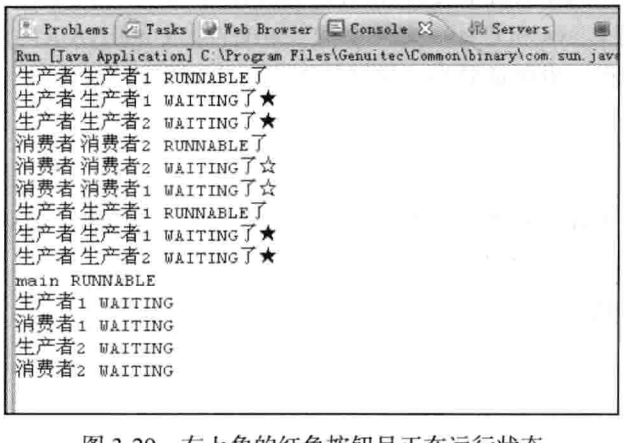


图 3-29　右上角的红色按钮呈正在运行状态

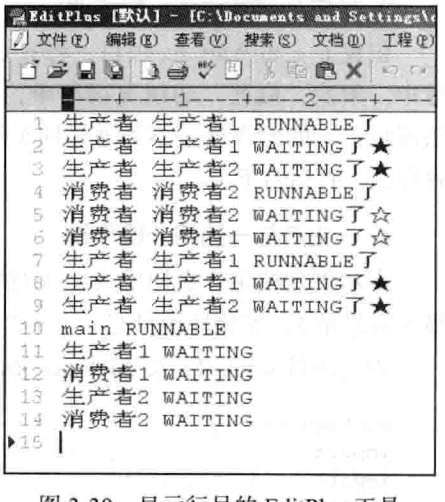


图 3-30　显示行号的 EditPlus 工具

分析 EditPlus 工具中的日志执行过程就明白了。以下分析的步骤与图 3-30 中行号对应。

1）生产者 1 进行生产，生产完毕后发出通知（但此通知属于“通知过早”），并释放锁，准备进入下一次的 while 循环。

2）生产者 1 进入了下一次 while 循环，迅速再次持有锁，发现产品并没有被消费，所以生产者 1 呈等待状态★。

3）生产者 2 被 start() 启动，生产者 2 发现产品还没有被消费，所以生产者 2 也呈等待状态★。

4）消费者 2 被 start() 启动，消费者 2 持有锁，将产品消费并发出通知（发出的通知唤醒了第 7 行生产者 1），运行结束后释放锁，等待消费者 2 进入下次循环。

5）消费者 2 进入了下一次的 while 循环，并持有锁，发现产品并未生产，所以释放锁并呈等待状态★。

6）消费者 1 被 start() 启动，快速持有锁，发现产品并未生产，所以释放锁并呈等待状态★。

7）由于消费者 2 在第 4 行已经将产品进行消费，唤醒了第 7 行的生产者 1 进行顺利生产后释放锁，并发出通知（此通知唤醒了第 9 行的生产者 2），生产者 1 准备进入下一次的 while 循环。

8）这时生产者 1 进入下一次的 while 循环再次持有锁，发现产品还并未消费，所以生产者 1 也呈等待状态★。

9）由于第 7 行的生产者 1 唤醒了生产者 2，生产者 2 发现产品还并未被消费，所以生产者 2 也呈等待状态★。

出现★符号就代表本线程进入等待状态，需要额外注意这样的执行结果。

假死出现的主要原因是有可能连续唤醒同类。怎么能解决这样的问题呢？不光唤醒同类，将异类也一同唤醒就解决了。

3. 多生产与多消费：操作值

创建 p_c_allWait_fix 项目，将 p_c_allWait 项目中的所有源代码复制到 p_c_allWait_fix 项目中。解决“假死”的情况很简单，将 P.java 和 C.java 文件中的 notify() 改成 notifyAll() 方法即可，它的原理就是不光通知同类线程，也包括异类。这样就不至于出现假死的状态了，程序会一直运行下去。

4. 一生产与一消费：操作栈

本示例是使生产者向堆栈 List 对象中放入数据，使消费者从 List 堆栈中取出数据。List 最大容量是 1，实验环境只有一个生产者与一个消费者。

创建项目 stack_1，类 MyStack.java 代码如下：

```
package entity;
import java.util.ArrayList;
import java.util.List;
public class MyStack {
private List list = new ArrayList();
synchronized public void push() {
    try {
        if (list.size() == 1) {
            this.wait();
        }
        list.add("anyString=" + Math.random());
        this.notify();
        System.out.println("push=" + list.size());
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    synchronized public String pop() {
        String returnValue = "";
        try {
            if (list.size() == 0) {
                System.out.println("pop操作中的: "
                        + Thread.currentThread().getName() + " 线程呈wait状态");
                this.wait();
            }
            returnValue = "" + list.get(0);
            list.remove(0);
            this.notify();
            System.out.println("pop=" + list.size());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return returnValue;
    }
}
```

生产者和消费者线程代码如图 3-31 所示。

P_Thread.java

```
package extthread;

import service.P;

public class P_Thread extends Thread {

    private P p;

    public P_Thread(P p) {
        super();
        this.p = p;
    }

    @Override
    public void run() {
        while (true) {
            p.pushService();
        }
    }

}
```

C_Thread.java

```
package extthread;

import service.C;

public class C_Thread extends Thread {

    private C r;

    public C_Thread(C r) {
        super();
        this.r = r;
    }

    @Override
    public void run() {
        while (true) {
            r.popService();
        }
    }

}
```

图 3-31 两个对象的代码

生产者 P.java 服务代码如下：

```
package service;
import entity.MyStack;
public class P {
private MyStack myStack;
public P(MyStack myStack) {
```

```
        super();
        this.myStack = myStack;
    }
    public void pushService() {
        myStack.push();
    }
}
```

消费者 C.java 服务代码如下：

```
package service;
import entity.MyStack;
public class C {
private MyStack myStack;
public C(MyStack myStack) {
    super();
    this.myStack = myStack;
}
public void popService() {
    System.out.println("pop=" + myStack.pop());
}
}
```

运行类 Run.java 代码如下：

```
package test.run;
import service.P;
import service.C;
import entity.MyStack;
import extthread.P_Thread;
import extthread.C_Thread;
public class Run {
public static void main(String[] args) {
    MyStack myStack = new MyStack();
    P p = new P(myStack);
    C r = new C(myStack);
    P_Thread pThread = new P_Thread(p);
    C_Thread rThread = new C_Thread(r);
    pThread.start();
    rThread.start();
}
}
```

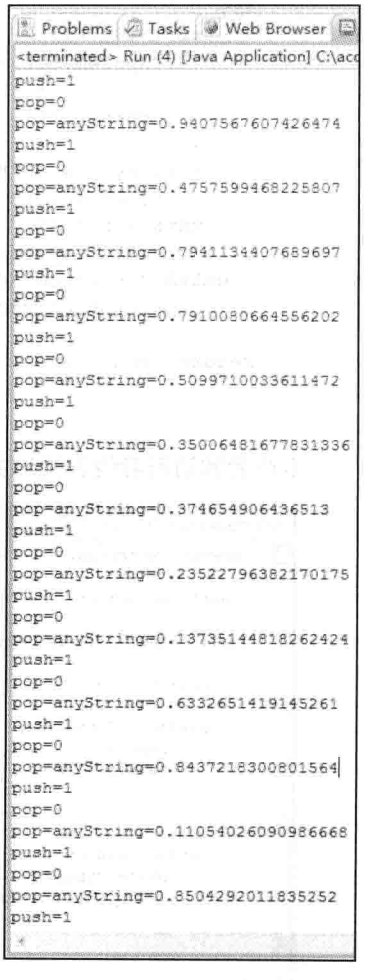


图 3-32 正常打印

程序运行结果是 size() 不会大于 1，如图 3-32 所示。

通过使用生产者 / 消费者模式，容器 size() 的值不会大于 1，这也是本示例想要实现的效果，值在 0 和 1 之间进行交替，也就是生产和消费这两个过程在交替执行。

5. 一生产与多消费——操作栈：解决 wait 条件改变与假死

本示例是使用一个生产者向堆栈 List 对象中放入数据，而多个消费者从 List 堆栈中取出数据。List 最大容量还是 1。

创建新的项目 stack_2_old，将项目 stack_1 中的所有代码内容复制到 stack_2_old 项目

中。更改 Run.java 代码如下：

```
package test.run;
import service.C;
import service.P;
import entity.MyStack;
import extthread.C_Thread;
import extthread.P_Thread;
public class Run {
public static void main(String[] args) throws InterruptedException {
    MyStack myStack = new MyStack();
    P p = new P(myStack);
    C r1 = new C(myStack);
    C r2 = new C(myStack);
    C r3 = new C(myStack);
    C r4 = new C(myStack);
    C r5 = new C(myStack);
    P_Thread pThread = new P_Thread(p);
    pThread.start();
    C_Thread cThread1 = new C_Thread(r1);
    C_Thread cThread2 = new C_Thread(r2);
    C_Thread cThread3 = new C_Thread(r3);
    C_Thread cThread4 = new C_Thread(r4);
    C_Thread cThread5 = new C_Thread(r5);
    cThread1.start();
    cThread2.start();
    cThread3.start();
    cThread4.start();
    cThread5.start();
}
}
```

程序运行后在某些几率下出现异常，如图 3-33 所示。

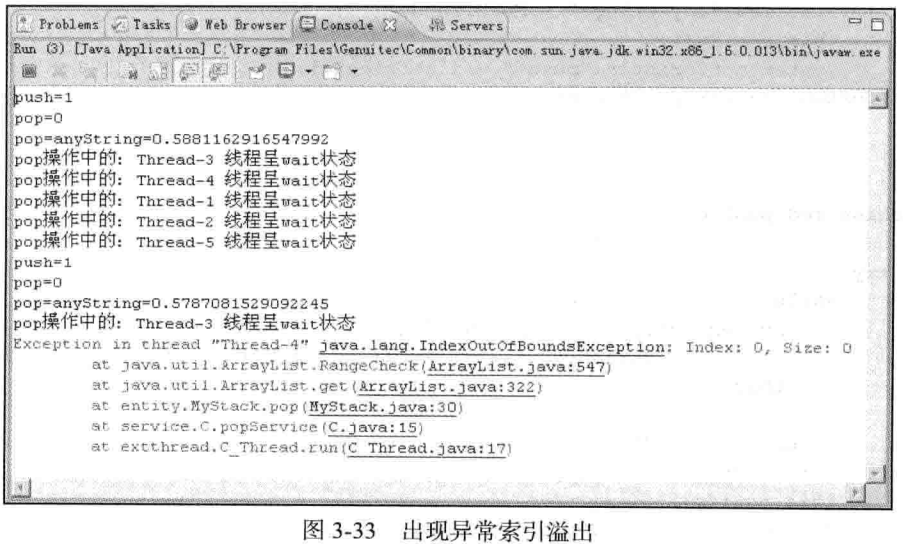


图 3-33 出现异常索引溢出

此问题的出现就是因为在 MyStack.java 类中使用了 if 语句作为条件判断，代码如下：

```
synchronized public void push() {
    try {
        if (list.size() == 1) {
            this.wait();
        }
        list.add("anyString=" + Math.random());
        this.notify();
        System.out.println("push=" + list.size());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

因为条件发生改变时并没有得到及时的响应，所以多个呈 wait 状态的线程被唤醒，继而执行 list.remove(0) 代码而出现异常。解决这个办法是，将 if 改成 while 语句即可。

新建项目 stack_2_new，将 stack_2_old 中的全部代码复制到 stack_2_new 项目中，并且更改 MyStack.java 类代码如下：

```
package entity;
import java.util.ArrayList;
import java.util.List;
public class MyStack {
private List list = new ArrayList();
synchronized public void push() {
    try {
        while (list.size() == 1) {
            this.wait();
        }
        list.add("anyString=" + Math.random());
        this.notify();
        System.out.println("push=" + list.size());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
synchronized public String pop() {
    String returnValue = "";
    try {
        while (list.size() == 0) {
            System.out.println("pop 操作中的: "
                    + Thread.currentThread().getName() + " 线程呈 wait 状态 ");
            this.wait();
        }
        returnValue = "" + list.get(0);
        list.remove(0);
        this.notify();
        System.out.println("pop=" + list.size());
```

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

运行项目没有出现执行异常，却出现了“假死”情况，如图 3-34 所示。

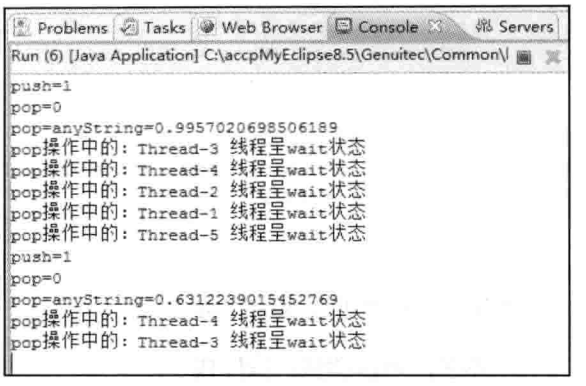


图 3-34 出现假死

解决的办法当然还是使用 notifyAll() 方法了。

创建全新的项目 stack_2_new_final，将 stack_2_new 项目中的所有源代码复制到 stack_2_new_final 项目中，将 MyStack.java 类中两处调用 notify() 方法改成调用 notifyAll() 方法，程序运行后不再出现假死，并且可以正常地运行下去了。

6. 多生产与一消费：操作栈

本示例是使用生产者向堆栈 List 对象中放入数据，使用消费者从 List 堆栈中取出数据。List 最大容量还是 1，实验环境是多个生产者与一个消费者。

创建项目 stack_3，将 stack_2_new_final 项目中 src 下的所有包及类复制到 stack_3 项目中。只需要如下更改 Run.java 代码：

```
package test.run;
import service.C;
import service.P;
import entity.MyStack;
import extthread.C_Thread;
import extthread.P_Thread;
public class Run {
public static void main(String[] args) throws InterruptedException {
    MyStack myStack = new MyStack();
    P p1 = new P(myStack);
    P p2 = new P(myStack);
    P p3 = new P(myStack);
    P p4 = new P(myStack);
    P p5 = new P(myStack);
    P p6 = new P(myStack);
    P_Thread pThread1 = new P_Thread(p1);
    P_Thread pThread2 = new P_Thread(p2);
    P_Thread pThread3 = new P_Thread(p3);
    P_Thread pThread4 = new P_Thread(p4);
    P_Thread pThread5 = new P_Thread(p5);
    P_Thread pThread6 = new P_Thread(p6);
```

```
        pThread1.start();
        pThread2.start();
        pThread3.start();
        pThread4.start();
        pThread5.start();
        pThread6.start();
        C c1 = new C(myStack);
        C_Thread cThread = new C_Thread(c1);
        cThread.start();
    }
}
```

程序运行结果如图 3-35 所示。

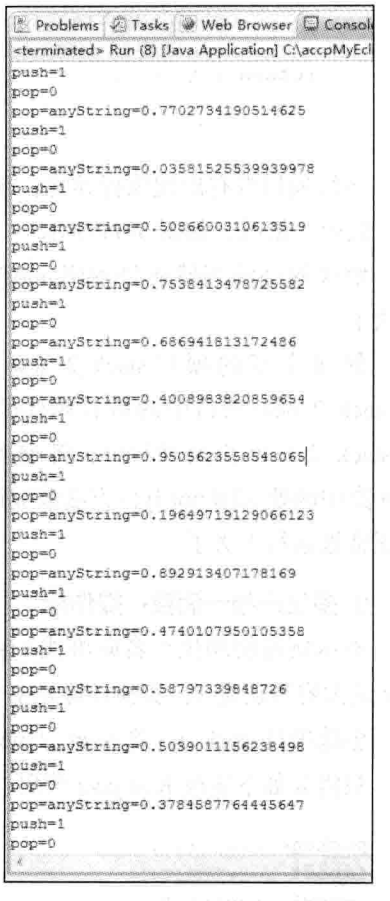


图 3-35 多生产与一消费正确运行结果片段

7. 多生产与多消费：操作栈

本示例是使用生产者向栈 List 对象中放入数据，使用消费者从 List 栈中取出数据。List 最大容量是 1，实验环境是多个生产者与多个消费者。

创建项目 stack_4，将 stack_3 项目中 src 下的所有包及类复制到 stack_4 项目中。

只需要对 Run.java 代码进行如下更改：

```
package test.run;
import service.C;
import service.P;
import entity.MyStack;
import extthread.C_Thread;
import extthread.P_Thread;
public class Run {
public static void main(String[] args) throws InterruptedException {
    MyStack myStack = new MyStack();
    P p1 = new P(myStack);
    P p2 = new P(myStack);
    P p3 = new P(myStack);
    P p4 = new P(myStack);
    P p5 = new P(myStack);
    P p6 = new P(myStack);
    P_Thread pThread1 = new P_Thread(p1);
    P_Thread pThread2 = new P_Thread(p2);
    P_Thread pThread3 = new P_Thread(p3);
    P_Thread pThread4 = new P_Thread(p4);
    P_Thread pThread5 = new P_Thread(p5);
    P_Thread pThread6 = new P_Thread(p6);
    pThread1.start();
    pThread2.start();
    pThread3.start();
```

```
        pThread4.start();
        pThread5.start();
        pThread6.start();
        C r1 = new C(myStack);
        C r2 = new C(myStack);
        C r3 = new C(myStack);
        C r4 = new C(myStack);
        C r5 = new C(myStack);
        C r6 = new C(myStack);
        C r7 = new C(myStack);
        C r8 = new C(myStack);
        C_Thread cThread1 = new C_Thread(r1);
        C_Thread cThread2 = new C_Thread(r2);
        C_Thread cThread3 = new C_Thread(r3);
        C_Thread cThread4 = new C_Thread(r4);
        C_Thread cThread5 = new C_Thread(r5);
        C_Thread cThread6 = new C_Thread(r6);
        C_Thread cThread7 = new C_Thread(r7);
        C_Thread cThread8 = new C_Thread(r8);
        cThread1.start();
        cThread2.start();
        cThread3.start();
        cThread4.start();
        cThread5.start();
        cThread6.start();
        cThread7.start();
        cThread8.start();
    }
}
```

程序运行的结果如图 3-36 所示。

从程序的运行结果来看，list 对象的 size() 并没有超过 1 。

图 3-36　多个生产与多个消费正确运行结果片段

3.1.12　通过管道进行线程间通信：字节流

在 Java 语言中提供了各种各样的输入 / 输出流 Stream，使我们能够很方便地对数据进行操作，其中管道流（pipeStream）是一种特殊的流，用于在不同线程间直接传送数据。一个线程发送数据到输出管道，另一个线程从输入管道中读数据。通过使用管道，实现不同线程间的通信，而无须借助于类似临时文件之类的东西。

在 Java 的 JDK 中提供了 4 个类来使线程间可以进行通信：

1）PipedInputStream 和 PipedOutputStream

2）PipedReader 和 PipedWriter

创建测试用的项目 pipeInputOutput。

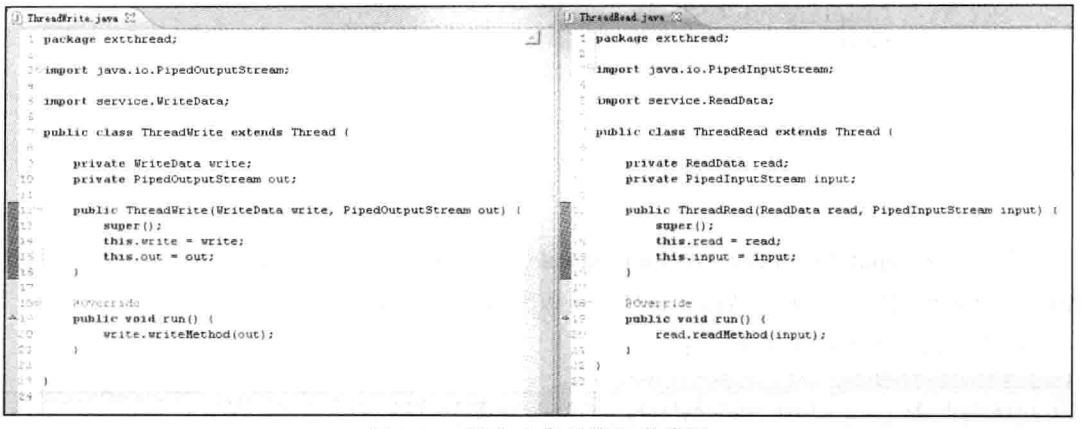
类 WriteData.java 代码如下：

```
package service;

import java.io.IOException;
import java.io.PipedOutputStream;

public class WriteData {

    public void writeMethod(PipedOutputStream out) {
        try {
            System.out.println("write :");
            for (int i = 0; i < 300; i++) {
                String outData = "" + (i + 1);
                out.write(outData.getBytes());
                System.out.print(outData);
            }
            System.out.println();
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

类 ReadData.java 代码如下：

```
package service;

import java.io.IOException;
import java.io.PipedInputStream;

public class ReadData {

    public void readMethod(PipedInputStream input) {
        try {
            System.out.println("read  :");
            byte[] byteArray = new byte[20];
            int readLength = input.read(byteArray);
            while (readLength != -1) {
                String newData = new String(byteArray, 0, readLength);
                System.out.print(newData);
                readLength = input.read(byteArray);
            }
            System.out.println();
            input.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

两个自定义线程代码如图 3-37 所示：



```
package extthread;

import java.io.PipedOutputStream;

import service.WriteData;

public class ThreadWrite extends Thread {

    private WriteData write;
    private PipedOutputStream out;

    public ThreadWrite(WriteData write, PipedOutputStream out) {
        super();
        this.write = write;
        this.out = out;
    }

    @Override
    public void run() {
        write.writeMethod(out);
    }

}
```

```
package extthread;

import java.io.PipedInputStream;

import service.ReadData;

public class ThreadRead extends Thread {

    private ReadData read;
    private PipedInputStream input;

    public ThreadRead(ReadData read, PipedInputStream input) {
        super();
        this.read = read;
        this.input = input;
    }

    @Override
    public void run() {
        read.readMethod(input);
    }
}
```

图 3-37 两个自定义线程类代码

类 Run.java 代码如下：

```
package test;

import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;

import service.ReadData;
import service.WriteData;
import extthread.ThreadRead;
import extthread.ThreadWrite;

public class Run {

    public static void main(String[] args) {

        try {
            WriteData writeData = new WriteData();
            ReadData readData = new ReadData();

            PipedInputStream inputStream = new PipedInputStream();
            PipedOutputStream outputStream = new PipedOutputStream();

            // inputStream.connect(outputStream);
            outputStream.connect(inputStream);

            ThreadRead threadRead = new ThreadRead(readData, inputStream);
            threadRead.start();

            Thread.sleep(2000);

            ThreadWrite threadWrite = new ThreadWrite(writeData, outputStream);
            threadWrite.start();

        } catch (IOException e) {
```

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

    }

}
```

使用代码 inputStream.connect(outputStream) 或 outputStream.connect(inputStream) 的作用使两个 Stream 之间产生通信链接，这样才可以将数据进行输出与输入。

程序运行结果如图 3-38 所示：

```
<terminated> Run (2) [Java Application] C:\Program Files\Genuitec\Common\binary\com.sun.java.
read  :
write :
123456789101112131415161718192021222324252627282930313233343536373839
123456789101112131415161718192021222324252627282930313233343536373839
```

图 3-38 从 1 开始

从程序打印的结果来看，两个线程通过管道流成功进行数据的传输。

但在此实验中，首先是读取线程 new ThreadRead(inputStream) 启动，由于当时没有数据被写入，所以线程阻塞在 int readLength = in.read(byteArray); 代码中，直到有数据被写入，才继续向下运行。

3.1.13 通过管道进行线程间通信：字符流

当然，在管道中还可以传递字符流。

创建测试用的项目 pipeReaderWriter。

类 WriteData.java 代码如下：

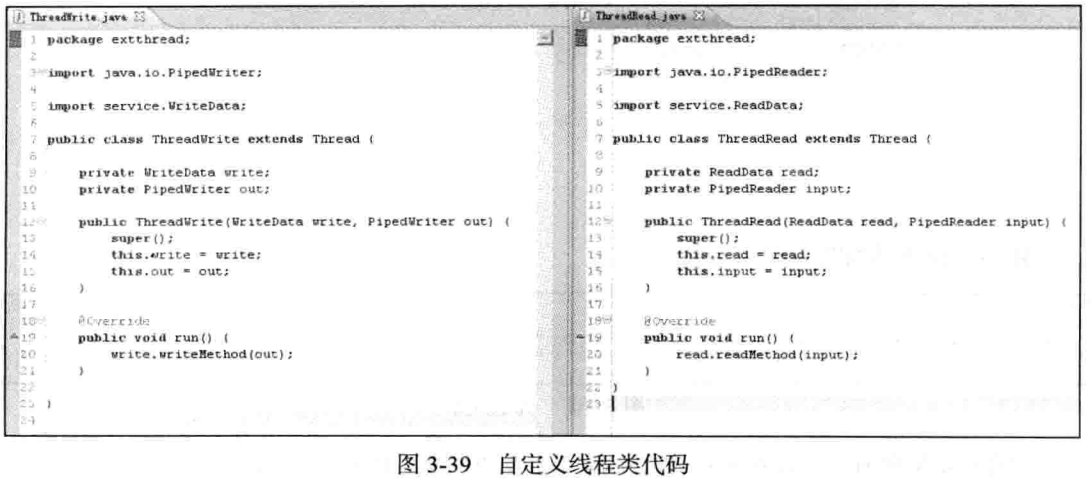
```
package service;

import java.io.IOException;
import java.io.PipedWriter;

public class WriteData {

    public void writeMethod(PipedWriter out) {
        try {
            System.out.println("write :");
            for (int i = 0; i < 300; i++) {
                String outData = "" + (i + 1);
                out.write(outData);
                System.out.print(outData);
            }
            System.out.println();
```

```
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

类 ReadData.java 代码如下：

```
package service;

import java.io.IOException;
import java.io.PipedReader;

public class ReadData {

    public void readMethod(PipedReader input) {
        try {
            System.out.println("read  :");
            char[] byteArray = new char[20];
            int readLength = input.read(byteArray);
            while (readLength != -1) {
                String newData = new String(byteArray, 0, readLength);
                System.out.print(newData);
                readLength = input.read(byteArray);
            }
            System.out.println();
            input.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

两个自定义线程代码如图 3-39 所示：



```
ThreadWrite.java
package extthread;

import java.io.PipedWriter;

import service.WriteData;

public class ThreadWrite extends Thread {

    private WriteData write;
    private PipedWriter out;

    public ThreadWrite(WriteData write, PipedWriter out) {
        super();
        this.write = write;
        this.out = out;
    }

    @Override
    public void run() {
        write.writeMethod(out);
    }

}
```

```
ThreadRead.java
package extthread;

import java.io.PipedReader;

import service.ReadData;

public class ThreadRead extends Thread {

    private ReadData read;
    private PipedReader input;

    public ThreadRead(ReadData read, PipedReader input) {
        super();
        this.read = read;
        this.input = input;
    }

    @Override
    public void run() {
        read.readMethod(input);
    }
}
```

图 3-39 自定义线程类代码

类 Run.java 代码如下：

```
package test;

import java.io.IOException;
import java.io.PipedReader;
import java.io.PipedWriter;

import service.ReadData;
import service.WriteData;
import extthread.ThreadRead;
import extthread.ThreadWrite;

public class Run {

    public static void main(String[] args) {

        try {
            WriteData writeData = new WriteData();
            ReadData readData = new ReadData();

            PipedReader inputStream = new PipedReader();
            PipedWriter outputStream = new PipedWriter();

            // inputStream.connect(outputStream);
            outputStream.connect(inputStream);

            ThreadRead threadRead = new ThreadRead(readData, inputStream);
            threadRead.start();

            Thread.sleep(2000);

            ThreadWrite threadWrite = new ThreadWrite(writeData, outputStream);
            threadWrite.start();

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

    }

}
```

程序运行结果如图 3-40 所示：

```
read  :
write :
12345678910111213141516171819202122232425262728293031323334353637383940414243444546474849505152535455565758596061626364656667
12345678910111213141516171819202122232425262728293031323334353637383940414243444546474849505152535455565758596061626364656667
```

图 3-40　从 1 开始

打印的结果和前一个示例基本一样，此实验是在两个线程中通过管道流进行字符数据的传输。

3.1.14　实战：等待 / 通知之交叉备份

本节目的是要继续学习等待 / 通知相关的知识点，创建 20 个线程，其中 10 个线程是将数据备份到 A 数据库中，另外 10 个线程将数据备份到 B 数据库中，并且备份 A 数据库和 B 数据库是交叉进行的。

首先创建出 20 个线程，效果如图 3-41 所示。

通过一些手段将这 20 个线程的运行效果变成有序的，如图 3-42 所示。

使用的技术还是等待 / 通知。

创建测试用的项目，名称为 wait_notify_insert_test。创建 DBTools.java 类代码如下：

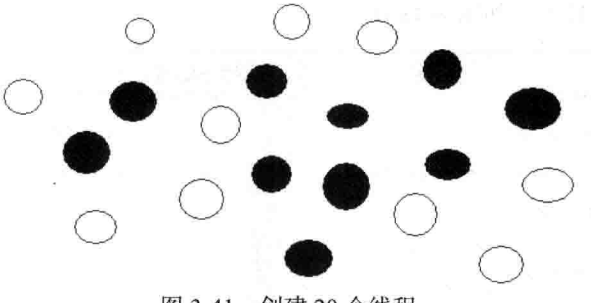


图 3-41　创建 20 个线程

图 3-42　具有交叉的效果

```
package service;
public class DBTools {
volatile private boolean prevIsA = false;
synchronized public void backupA() {
    try {
        while (prevIsA == true) {
            wait();
        }
        for (int i = 0; i < 5; i++) {
            System.out.println("★★★★★");
        }
        prevIsA = true;
        notifyAll();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
synchronized public void backupB() {
    try {
        while (prevIsA == false) {
            wait();
        }
```

```
            for (int i = 0; i < 5; i++) {
                System.out.println("☆☆☆☆☆");
            }
            prevIsA = false;
            notifyAll();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

变量 prevIsA 的主要作用就是确保备份“★★★★★”数据库 A 首先执行，然后与“☆☆☆☆☆”数据库 B 交替进行备份。

创建两个线程工具类，如图 3-43 所示。

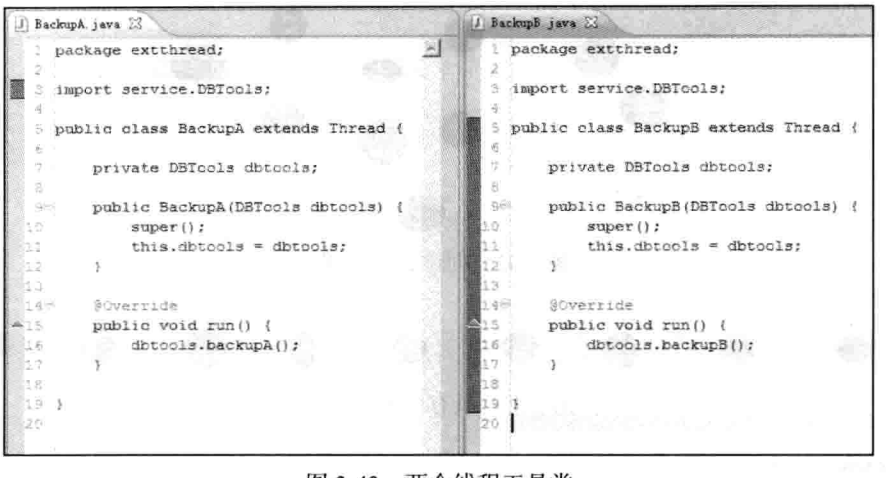


图 3-43 两个线程工具类

运行类 Run.java 代码如下：

```
package test.run;
import service.DBTools;
import extthread.BackupA;
import extthread.BackupB;
public class Run {
public static void main(String[] args) {
    DBTools dbtools = new DBTools();
    for (int i = 0; i < 20; i++) {
        BackupB output = new BackupB(dbtools);
        output.start();
        BackupA input = new BackupA(dbtools);
        input.start();
    }
}
}
```

程序运行后部分的打印效果如下：

★★★★★
★★★★★
★★★★★
★★★★★
★★★★★
☆☆☆☆☆
☆☆☆☆☆
☆☆☆☆☆
☆☆☆☆☆
☆☆☆☆☆
★★★★★
★★★★★
★★★★★
★★★★★
★★★★★
☆☆☆☆☆
☆☆☆☆☆
☆☆☆☆☆
☆☆☆☆☆
☆☆☆☆☆
★★★★★
★★★★★
★★★★★
★★★★★
★★★★★
☆☆☆☆☆
☆☆☆☆☆
☆☆☆☆☆
☆☆☆☆☆
☆☆☆☆☆

打印的效果是交替运行的。

交替打印的原理就是使用如下代码作为标记：

```
volatile private boolean prevIsA = false;
```

实现了 A 和 B 线程交替备份的效果。

3.2　方法 join 的使用

在很多情况下，主线程创建并启动子线程，如果子线程中要进行大量的耗时运算，主线程往往将早于子线程结束之前结束。这时，如果主线程想等待子线程执行完成之后再结束，比如子线程处理一个数据，主线程要取得这个数据中的值，就要用到 join() 方法了。方法 join() 的作用是等待线程对象销毁。

3.2.1　学习方法 join 前的铺垫

在介绍 join 方法之前，先来看一个实验。

创建测试用的 java 项目，名称为 joinTest1，类 MyThread.java 代码如下：

```
package extthread;
public class MyThread extends Thread {
@Override
public void run() {
    try {
        int secondValue = (int) (Math.random() * 10000);
        System.out.println(secondValue);
        Thread.sleep(secondValue);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
}
```

类 Test.java 代码如下：

```
package test;
import extthread.MyThread;
public class Test {
public static void main(String[] args) {
    MyThread threadTest = new MyThread();
    threadTest.start();
    // Thread.sleep(?)
    System.out.println("我想当 threadTest 对象执行完毕后我再执行 ");
    System.out.println("但上面代码中的 sleep() 中的值应该写多少呢？ ");
    System.out.println("答案是：根据不能确定 :)");
}
}
```

程序运行结果如图 3-44 所示。

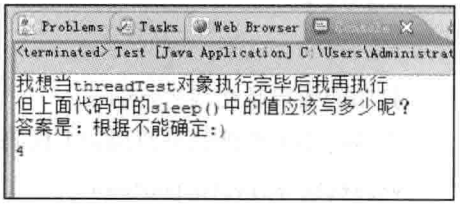


图 3-44 方法 sleep() 中的值不能确定

3.2.2 用 join() 方法来解决

方法 join 可以解决这个问题。新建 java 项目 joinTest2，类 MyThread.java 代码如下：

```
package extthread;
public class MyThread extends Thread {
@Override
public void run() {
    try {
        int secondValue = (int) (Math.random() * 10000);
        System.out.println(secondValue);
        Thread.sleep(secondValue);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
}
```

类 Test.java 代码如下：

```
package test;
import extthread.MyThread;
public class Test {
public static void main(String[] args) {
    try {
        MyThread threadTest = new MyThread();
        threadTest.start();
        threadTest.join();
        System.out.println("我想当 threadTest 对象执行完毕后我再执行，我做到了 ");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

程序运行后的结果如图 3-45 所示。

图 3-45　运行结果

方法 join 的作用是使所属的线程对象 x 正常执行 run() 方法中的任务，而使当前线程 z 进行无限期的阻塞，等待线程 x 销毁后再继续执行线程 z 后面的代码。

方法 join 具有使线程排队运行的作用，有些类似同步的运行效果。join 与 synchronized 的区别是：join 在内部使用 wait() 方法进行等待，而 sychronized 关键字使用的是“对象监视器”原理做为同步。

3.2.3　方法 join 与异常

在 join 过程中，如果当前线程对象被中断，则当前线程出现异常。

创建测试用的项目 joinException，类 ThreadA.java 代码如下：

```
package extthread;
public class ThreadA extends Thread {
@Override
public void run() {
    for (int i = 0; i < Integer.MAX_VALUE; i++) {
        String newString = new String();
        Math.random();
    }
}
}
```

类 ThreadB.java 代码如下：

```
package extthread;
public class ThreadB extends Thread {
@Override
public void run() {
    try {
```

```
            ThreadA a = new ThreadA();
            a.start();
            a.join();
            System.out.println("线程B在run end处打印了");
        } catch (InterruptedException e) {
            System.out.println("线程B在catch处打印了");
            e.printStackTrace();
        }
    }
}
```

类 ThreadC.java 代码如下：

```
package extthread;
public class ThreadC extends Thread {
private ThreadB threadB;
public ThreadC(ThreadB threadB) {
    super();
    this.threadB = threadB;
}
@Override
public void run() {
    threadB.interrupt();
}
}
```

类 Run.java 代码如下：

```
package test.run;
import extthread.ThreadB;
import extthread.ThreadC;
public class Run {
public static void main(String[] args) {
    try {
        ThreadB b = new ThreadB();
        b.start();
        Thread.sleep(500);
        ThreadC c = new ThreadC(b);
        c.start();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

程序运行后的效果如图 3-46 所示。

图 3-46 出现异常

说明方法 join() 与 interrupt() 方法如果彼此遇到，则会出现异常。但进程按钮还呈“红色”，原因是线程 ThreadA 还在继续运行，线程 ThreadA 并未出现异常，是正常执行的状态。

3.2.4　方法 join(long) 的使用

方法 join(long) 中的参数是设定等待的时间。

创建测试用的项目 joinLong，类 MyThread.java 代码如下：

```
package extthread;
public class MyThread extends Thread {
@Override
public void run() {
    try {
        System.out.println("begin Timer=" + System.currentTimeMillis());
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

类 Test.java 代码如下：

```
package test;
import extthread.MyThread;
public class Test {
public static void main(String[] args) {
    try {
        MyThread threadTest = new MyThread();
        threadTest.start();
        threadTest.join(2000);//只等 2 秒
        // Thread.sleep(2000);
        System.out.println("  end timer=" + System.currentTimeMillis());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

程序运行结果如图 3-47 所示。

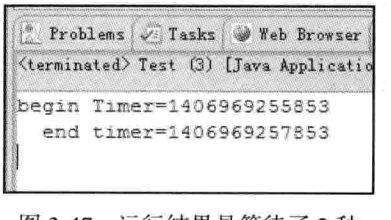


图 3-47　运行结果是等待了 2 秒

但将 main 方法中的代码改成使用 sleep(2000) 方法时，运行的效果还是等待了 2 秒，如图 3-48 所示。

那使用 join(2000) 和使用 sleep(2000) 有什么区别呢？上面的示例中在运行效果上并没有区别，其实区别主要还是来自于这 2 个方法对同步的处理上。

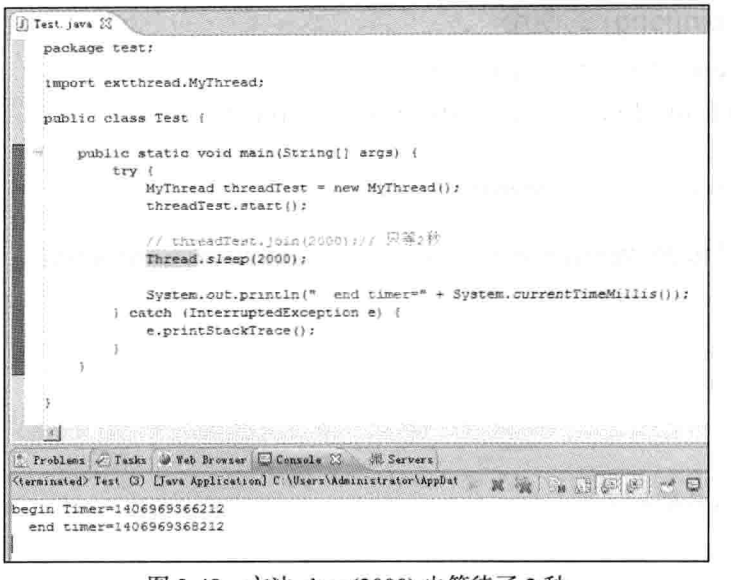


图 3-48 方法 sleep(2000) 也等待了 2 秒

3.2.5 方法 join(long) 与 sleep(long) 的区别

方法 join(long) 的功能在内部是使用 wait(long) 方法来实现的，所以 join(long) 方法具有释放锁的特点。

方法 join(long) 源代码如下：

```
public final synchronized void join(long millis)
        throws InterruptedException {
    long base = System.currentTimeMillis();
    long now = 0;
    if (millis < 0) {
        throw new IllegalArgumentException("timeout value is negative");
    }
    if (millis == 0) {
        while (isAlive()) {
            wait(0);
        }
    } else {
        while (isAlive()) {
            long delay = millis - now;
            if (delay <= 0) {
                break;
            }
            wait(delay);
            now = System.currentTimeMillis() - base;
        }
    }
}
```

从源代码中可以了解到，当执行 wait(long) 方法后，当前线程的锁被释放，那么其他线程就可以调用此线程中的同步方法了。

而 Thread.sleep(long) 方法却不释放锁。在下面的示例中将实验 Thread.sleep(long) 方法具有不释放锁的特点。

创建测试用的项目 join_sleep_1，类 ThreadA.java 代码如下：

```
package extthread;
public class ThreadA extends Thread {
private ThreadB b;
public ThreadA(ThreadB b) {
    super();
    this.b = b;
}
@Override
public void run() {
    try {
        synchronized (b) {
            b.start();
            Thread.sleep(6000);
            //Thread.sleep() 不释放锁!
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

类 ThreadB.java 代码如下：

```
package extthread;
public class ThreadB extends Thread {
@Override
public void run() {
    try {
        System.out.println("  b run begin timer="
                + System.currentTimeMillis());
        Thread.sleep(5000);
        System.out.println("  b run   end timer="
                + System.currentTimeMillis());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
synchronized public void bService() {
    System.out.println("打印了 bService timer=" + System.currentTimeMillis());
}
}
```

类 ThreadC.java 代码如下：

```
package extthread;
public class ThreadC extends Thread {
private ThreadB threadB;
public ThreadC(ThreadB threadB) {
    super();
    this.threadB = threadB;
}
@Override
public void run() {
    threadB.bService();
}
}
```

类 Run.java 代码如下：

```
package test.run;
import extthread.ThreadA;
import extthread.ThreadB;
import extthread.ThreadC;
public class Run {
public static void main(String[] args) {
    try {
        ThreadB b = new ThreadB();
        ThreadA a = new ThreadA(b);
        a.start();
        Thread.sleep(1000);
        ThreadC c = new ThreadC(b);
        c.start();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

程序运行后的效果如图 3-49 所示。

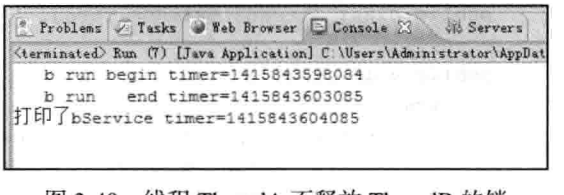


图 3-49 线程 ThreadA 不释放 ThreadB 的锁

由于线程 ThreadA 使用 Thread.sleep(long) 方法一直持有 ThreadB 对象的锁，时间达到 6 秒，所以线程 ThreadC 只有在 ThreadA 时间到达 6 秒后释放 ThreadB 的锁时，才可以调用 ThreadB 中的同步方法 synchronized public void bService()。

上面实验证明 Thread.sleep(long) 方法不释放锁。

下面继续实验，验证 join() 方法释放锁的特点。

创建实验用的项目 join_sleep_2，将 join_sleep_1 中的所有代码复制到 join_sleep_2 项目中，更改 ThreadA.java 类代码如下：

```
package extthread;
public class ThreadA extends Thread {
private ThreadB b;
public ThreadA(ThreadB b) {
    super();
    this.b = b;
}
@Override
public void run() {
    try {
        synchronized (b) {
            b.start();
            b.join();//说明 join 释放锁了!
            for (int i = 0; i < Integer.MAX_VALUE; i++) {
                String newString = new String();
                Math.random();
            }
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

程序运行后的效果如图 3-50 所示。

由于线程 ThreadA 释放了 ThreadB 的锁，所以线程 ThreadC 可以调用 ThreadB 中的同步方法 synchronized public void bService()。

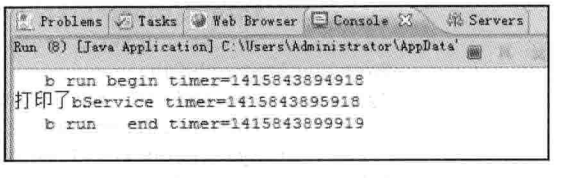


图 3-50　方法 join() 释放锁

此实验也再次说明 join(long) 方法具有释放锁的特点。

3.2.6　方法 join() 后面的代码提前运行：出现意外

针对前面章节中的代码进行测试的过程中，还可以延伸出“陷阱式”的结果，如果稍加不注意，就会掉进“陷阱”里。

创建测试用的项目 joinMoreTest，类 ThreadA.java 代码如下：

```
package extthread;
public class ThreadA extends Thread {
private ThreadB b;
public ThreadA(ThreadB b) {
    super();
    this.b = b;
}
@Override
public void run() {
```

```
        try {
            synchronized (b) {
                System.out.println("begin A ThreadName="
                        + Thread.currentThread().getName() + "  "
                        + System.currentTimeMillis());
                Thread.sleep(5000);
                System.out.println("  end A ThreadName="
                        + Thread.currentThread().getName() + "  "
                        + System.currentTimeMillis());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

类 ThreadB.java 代码如下：

```
package extthread;
public class ThreadB extends Thread {
@Override
synchronized public void run() {
    try {
        System.out.println("begin B ThreadName="
                + Thread.currentThread().getName() + "  "
                + System.currentTimeMillis());
        Thread.sleep(5000);
        System.out.println("  end B ThreadName="
                + Thread.currentThread().getName() + "  "
                + System.currentTimeMillis());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

创建 Run1.java 代码如下：

```
package test.run;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run1 {
public static void main(String[] args) {
    try {
        ThreadB b = new ThreadB();
        ThreadA a = new ThreadA(b);
        a.start();
        b.start();
        b.join(2000);
        System.out.println("                    main end "
                + System.currentTimeMillis());
```

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

程序运行后，在控制台有时候打印结果如图 3-51 所示。

在控制台有时候打印的结果如图 3-52 所示。

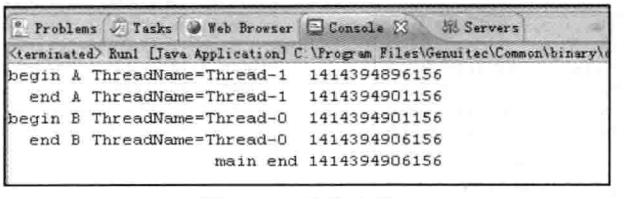


图 3-51　运行结果 1

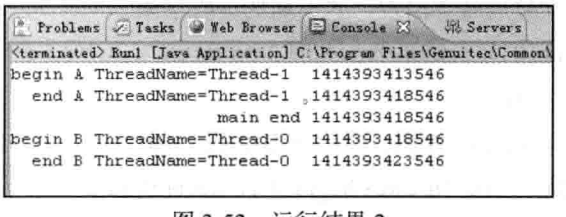


图 3-52　运行结果 2

为什么出现截然不同的运行结果呢?

3.2.7　方法 join() 后面的代码提前运行：解释意外

为了查看 join() 方法在 Run1.java 类中执行的时机，创建 RunFirst.java 类文件，代码如下：

```
package test.run;

import extthread.ThreadA;
import extthread.ThreadB;

public class RunFirst {

    public static void main(String[] args) {
        ThreadB b = new ThreadB();
        ThreadA a = new ThreadA(b);
        a.start();
        b.start();
        System.out.println("   main end=" + System.currentTimeMillis());
    }

}
```

程序第 1 次运行结果如图 3-53 所示。

程序第 2 次运行结果如图 3-54 所示。

通过多次运行 RunFirst.java 文件后，可以发现一个规律：main end 往往都是第一个打印的。所以可以完全确定地得出一个结论：方法 join(2000) 大部分是先运行的，也就是先抢到 ThreadB 的锁，然后快速进行释放。

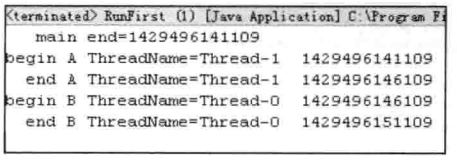


图 3-53 第 1 次运行结果

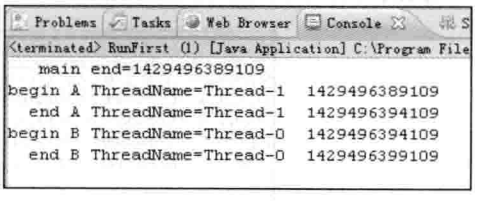


图 3-54 第 2 次运行结果

而执行 Run1.java 文件后就会出现一些不同的运行结果。

先来看看有可能出现的运行结果 A，如图 3-55 所示。

看一下运行结果的解释：

1）b.join(2000) 方法先抢到 B 锁，然后将 B 锁进行释放；

2）ThreadA 抢到锁，打印 ThreadA begin 并且 sleep(5000)；

3）ThreadA 打印 ThreadA end，并释放锁；

4）这时 join(2000) 和 ThreadB 争抢锁，而 join(2000) 再次抢到锁，发现时间已过，释放锁后打印 main end；

5）ThreadB 抢到锁打印 ThreadB begin；

6）5 秒之后再打印 ThreadB end。

再来看看有可能出现的运行结果 B，如图 3-56 所示。

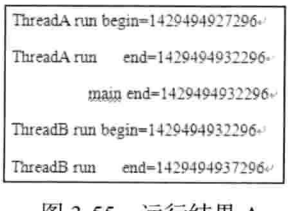


图 3-55 运行结果 A

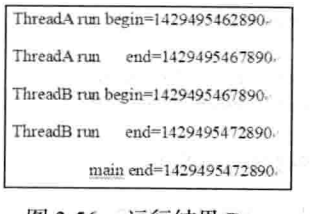


图 3-56 运行结果 B

看一下运行结果的解释：

1）b.join(2000) 方法先抢到 B 锁，然后将 B 锁进行释放；

2）ThreadA 抢到锁，打印 ThreadA begin 并且 sleep(5000)；

3）ThreadA 打印 ThreadA end，并释放锁；

4）这时 join(2000) 和 ThreadB 争抢锁，而 ThreadB 抢到锁后执行 sleep(5000) 后释放锁；

5）main end 在最后输出。

再来看看有可能出现的运行结果 C，如图 3-57 所示。

```
ThreadA run begin=1429495462890
ThreadA run    end=1429495467890
ThreadB run begin=1429495467890
          main end=1429495467890
ThreadB run    end=1429495472890
```

图 3-57　运行结果 C

看一下运行结果的解释：

1）b.join(2000) 方法先抢到 B 锁，然后将 B 锁进行释放；

2）ThreadA 抢到锁，打印 ThreadA begin 并且 sleep(5000)；

3）ThreadA 打印 ThreadA end，并释放锁；

4）这时 join(2000) 和 ThreadB 争抢锁，而 join(2000) 再次抢到锁发现时间已过，释放锁；

5）B 抢到锁打印 ThreadB begin；

6）这时 main end 也异步输出；

7）打印 ThreadB end。

3.3　类 ThreadLocal 的使用

变量值的共享可以使用 public static 变量的形式，所有的线程都使用同一个 public static 变量。如果想实现每一个线程都有自己的共享变量该如何解决呢？ JDK 中提供的类 ThreadLocal 正是为了解决这样的问题。

类 ThreadLocal 主要解决的就是每个线程绑定自己的值，可以将 ThreadLocal 类比喻成全局存放数据的盒子，盒子中可以存储每个线程的私有数据。

3.3.1　方法 get() 与 null

创建名称为 ThreadLocal11 的项目，类 Run.java 代码如下：

```
package test;
public class Run {
public static ThreadLocal tl = new ThreadLocal();
public static void main(String[] args) {
    if (tl.get() == null) {
        System.out.println("从未放过值");
        tl.set("我的值");
    }
    System.out.println(tl.get());
    System.out.println(tl.get());
}
}
```

程序运行的结果如图 3-58 所示。

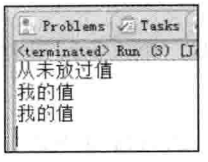


图 3-58 运行结果

从图中的运行结果来看，第一次调用 tl 对象的 get() 方法时返回的值是 null，通过调用 set() 方法赋值后顺利取出值并打印到控制台上。类 Threadlocal 解决的是变量在不同线程间的隔离性，也就是不同线程拥有自己的值，不同线程中的值是可以放入 Threadlocal 类中进行保存的。

3.3.2 验证线程变量的隔离性

创建测试用的项目 ThreadLocalTest，类 Tools.java 代码如下：

```
package tools;
public class Tools {
public static ThreadLocal tl = new ThreadLocal();
}
```

两个自定义线程类代码如图 3-59 所示。

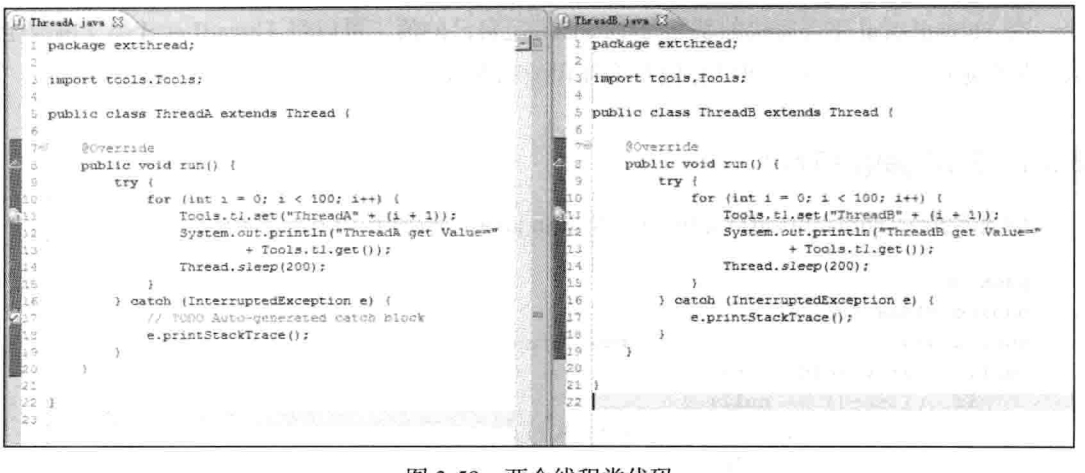


图 3-59 两个线程类代码

类 Run.java 代码如下：

```
package test;
```

```
import tools.Tools;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
public static void main(String[] args) {
    try {
        ThreadA a = new ThreadA();
        ThreadB b = new ThreadB();
        a.start();
        b.start();
        for (int i = 0; i < 100; i++) {
            Tools.tl.set("Main" + (i + 1));
             System.out.println("Main get Value=" +
Tools.tl.get());
            Thread.sleep(200);
        }
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
}
```

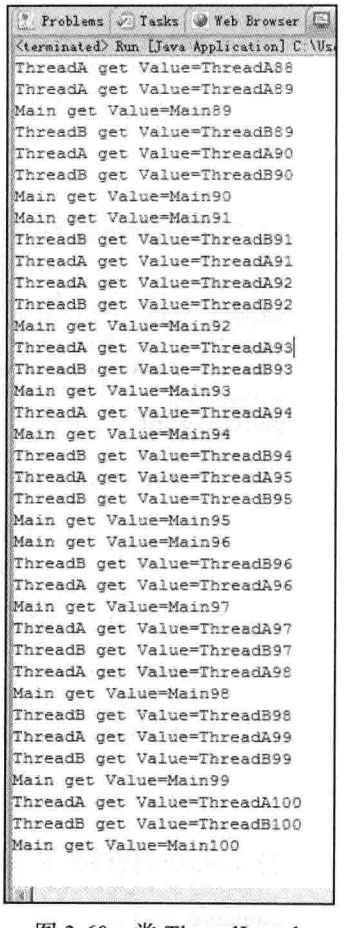


图 3-60 类 ThreadLocal 存储每一个线程的私有数据

程序运行后的效果如图 3-60 所示。

虽然 3 个线程都向 tl 对象中 set() 数据值，但每个线程还是能取出自己的数据。

创建新的项目 s5 来再次验证数据的隔离性。类 Tools.java 代码如下：

```
package tools;
import java.util.Date;
public class Tools {
public static ThreadLocal<Date> tl = new
    ThreadLocal<Date>();
}
```

类 ThreadA.java 代码如下：

```
package extthread;
import java.util.Date;
import tools.Tools;
public class ThreadA extends Thread {
@Override
public void run() {
    try {
        for (int i = 0; i < 20; i++) {
            if (Tools.tl.get() == null) {
```

```
                Tools.tl.set(new Date());
            }
            System.out.println("A " + Tools.tl.get().getTime());
            Thread.sleep(100);
        }
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
}
```

类 ThreadB.java 代码如下：

```
package extthread;
import java.util.Date;
import tools.Tools;
public class ThreadB extends Thread {
@Override
public void run() {
    try {
        for (int i = 0; i < 20; i++) {
            if (Tools.tl.get() == null) {
                Tools.tl.set(new Date());
            }
            System.out.println("B " + Tools.tl.get().getTime());
            Thread.sleep(100);
        }
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
}
```

类 Run.java 代码如下：

```
package test;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
public static void main(String[] args) {
    try {
        ThreadA a = new ThreadA();
        a.start();
        Thread.sleep(1000);
        ThreadB b = new ThreadB();
        b.start();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

程序运行结果如图 3-61 所示。

在第一次调用 Threadlocal 类的 get() 方法返回值是 null，怎么样实现第一次调用 get() 不返回 null 呢？也就是具有默认值的效果。

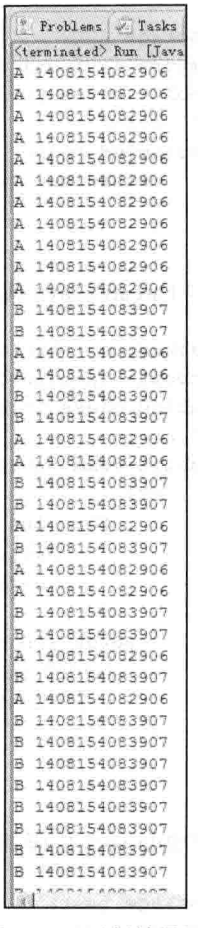


图 3-61　运行结果只有两种时间

3.3.3　解决 get() 返回 null 问题

创建名称为 ThreadLocal22 的项目，继承 ThreadLocal 类产生 ThreadLocalExt.java 类，代码如下：

```
package ext;
public class ThreadLocalExt extends ThreadLocal {
@Override
protected Object initialValue() {
    return "我是默认值 第一次get不再为null";
}
}
```

覆盖 initialValue() 方法具有初始值。

运行类 Run.java 代码如下：

```
package test;
import ext.ThreadLocalExt;
public class Run {
public static ThreadLocalExt tl = new
    ThreadLocalExt();
public static void main(String[] args) {
    if (tl.get() == null) {
        System.out.println("从未放过值");
        tl.set("我的值");
    }
    System.out.println(tl.get());
    System.out.println(tl.get());
}
}
```

程序运行结果如图 3-62 所示。

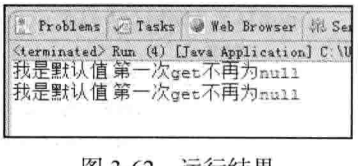


图 3-62　运行结果

此案例仅仅证明 main 线程有自己的值，那其他线程是否会有自己的初始值呢？

3.3.4　再次验证线程变量的隔离性

创建名称为 ThreadLocal33 的项目，类 Tools.java 代码如下：

```
package tools;
import ext.ThreadLocalExt;
public class Tools {
public static ThreadLocalExt tl = new ThreadLocalExt();
}
```

类 ThreadLocalExt.java 代码如下：

```
package ext;
import java.util.Date;
public class ThreadLocalExt extends ThreadLocal {
@Override
protected Object initialValue() {
    return new Date().getTime();
}
}
```

类 ThreadA.java 代码如下：

```
package extthread;
import tools.Tools;
public class ThreadA extends Thread {
@Override
public void run() {
    try {
        for (int i = 0; i < 10; i++) {
            System.out.println("在 ThreadA 线程中取值 =" + Tools.tl.get());
            Thread.sleep(100);
        }
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
}
```

类 Run.java 代码如下：

```
package test;
import tools.Tools;
import extthread.ThreadA;
public class Run {
public static void main(String[] args) {
    try {
        for (int i = 0; i < 10; i++) {
            System.out.println("        在 Main 线程中取值 =" + Tools.tl.get());
            Thread.sleep(100);
        }
        Thread.sleep(5000);
        ThreadA a = new ThreadA();
        a.start();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

程序运行结果如图 3-63 所示。

子线程和父线程各有各自所拥有的值。

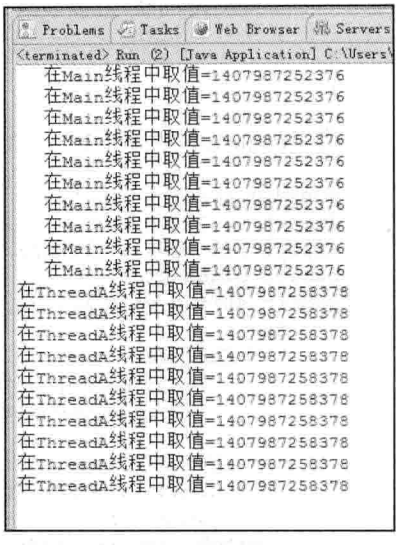


图 3-63　运行结果各有各的值

3.4　类 InheritableThreadLocal 的使用

使用类 InheritableThreadLocal 可以在子线程中取得父线程继承下来的值。

3.4.1　值继承

使用 InheritableThreadLocal 类可以让子线程从父线程中取得值。

创建测试用的项目 InheritableThreadLocal1，类 InheritableThreadLocalExt.java 代码如下：

```
package ext;
import java.util.Date;
public class InheritableThreadLocalExt extends InheritableThreadLocal {
@Override
protected Object initialValue() {
    return new Date().getTime();
}
}
```

类 Tools.java 代码如下：

```
package tools;
import ext.InheritableThreadLocalExt;
public class Tools {
public static InheritableThreadLocalExt tl = new InheritableThreadLocalExt();
}
```

类 ThreadA.java 代码如下：

```
package extthread;
import tools.Tools;
public class ThreadA extends Thread {
@Override
public void run() {
    try {
        for (int i = 0; i < 10; i++) {
            System.out.println("在 ThreadA 线程中取值 =" + Tools.tl.get());
            Thread.sleep(100);
        }
    } catch (InterruptedException e) {
```

```
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

类 Run.java 代码如下：

```
package test;
import tools.Tools;
import extthread.ThreadA;
public class Run {
public static void main(String[] args) {
    try {
        for (int i = 0; i < 10; i++) {
            System.out.println("        在 Main 线程中取值 =" + Tools.tl.get());
            Thread.sleep(100);
        }
        Thread.sleep(5000);
        ThreadA a = new ThreadA();
        a.start();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

程序运行结果如图 3-64 所示。

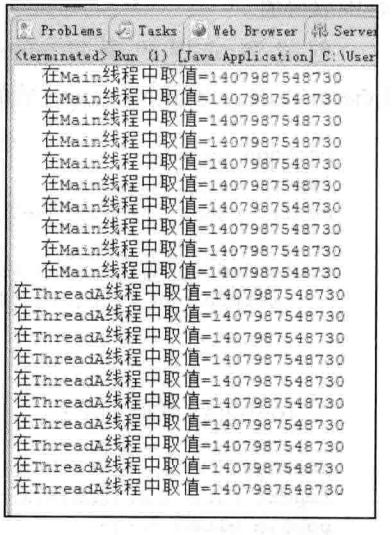


图 3-64 值成功地从父线程继承下来

3.4.2 值继承再修改

如果在继承的同时还可以对值进行进一步的处理那就更好了。

创建测试用的项目 InheritableThreadLocal2，将 InheritableThreadLocal1 项目中的所有类复制到 InheritableThreadLocal2 项目中。

更改类 InheritableThreadLocalExt.java 代码如下：

```
package ext;
import java.util.Date;
public class InheritableThreadLocalExt extends InheritableThreadLocal {
@Override
protected Object initialValue() {
    return new Date().getTime();
}
@Override
protected Object childValue(Object parentValue) {
    return parentValue + " 我在子线程加的 ~!";
}
}
```

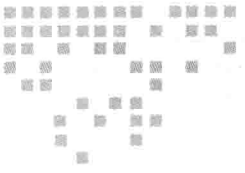
程序运行后的效果如图 3-65 所示。

图 3-65　成功继承并修改

但在使用 InheritableThreadLocal 类需要注意一点的是，如果子线程在取得值的同时，主线程将 InheritableThreadLocal 中的值进行更改，那么子线程取到的值还是旧值。

3.5　本章总结

本章的内容已经学习完毕。经过本章的学习，可以将以前分散的线程对象进行彼此的通信与协作，线程任务不再是单打独斗，更具有团结性，因为它们之间可以互相通信，就像命令官与执行者一样。对任务的计划规划更加合理，不再是随机的和盲目的了。



Chapter 4 第 4 章

Lock 的使用

本章将要介绍使用 Java5 中 Lock 对象也能实现同步的效果，而且在使用上更加方便。本章着重掌握如下 2 个知识点：

- ReentrantLock 类的使用。
- ReentrantReadWriteLock 类的使用。

4.1 使用 ReentrantLock 类

在 Java 多线程中，可以使用 synchronized 关键字来实现线程之间同步互斥，但在 JDK1.5 中新增加了 ReentrantLock 类也能达到同样的效果，并且在扩展功能上也更加强大，比如具有嗅探锁定、多路分支通知等功能，而且在使用上也比 synchronized 更加的灵活。

4.1.1 使用 ReentrantLock 实现同步：测试 1

既然 ReentrantLock 类在功能上相比 synchronized 更多，那么就以一个初步的程序示例来介绍一下 ReentrantLock 类的使用。

创建测试用的项目 ReentrantLockTest，创建类 MyService.java，代码如下：

```
package service;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
```

```
public class MyService {
private Lock lock = new ReentrantLock();
public void testMethod() {
    lock.lock();
    for (int i = 0; i < 5; i++) {
        System.out.println("ThreadName=" + Thread.currentThread().getName()
                + (" " + (i + 1)));
    }
    lock.unlock();
}
}
```

调用 ReentrantLock 对象的 lock() 方法获取锁，调用 unlock() 方法释放锁。

创建类 MyThread.java 代码如下：

```
package extthread;
import service.MyService;
public class MyThread extends Thread {
private MyService service;
public MyThread(MyService service) {
    super();
    this.service = service;
}
@Override
public void run() {
    service.testMethod();
}
}
```

运行类 Run.java 代码如下：

```
package test;
import service.MyService;
import extthread.MyThread;
public class Run {
public static void main(String[] args) {
    MyService service = new MyService();
    MyThread a1 = new MyThread(service);
    MyThread a2 = new MyThread(service);
    MyThread a3 = new MyThread(service);
    MyThread a4 = new MyThread(service);
    MyThread a5 = new MyThread(service);
    a1.start();
    a2.start();
    a3.start();
    a4.start();
    a5.start();
}
}
```

程序运行结果如图 4-1 所示。

从运行的结果来看，当前线程打印完毕之后将锁进行释放，其他线程才可以继续打印。线程打印的数据是分组打印，因为当前线程已经持有锁，但线程之间打印的顺序是随机的。

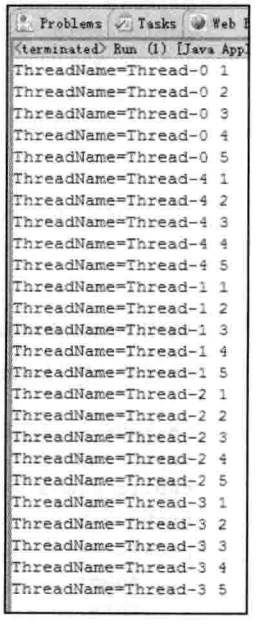


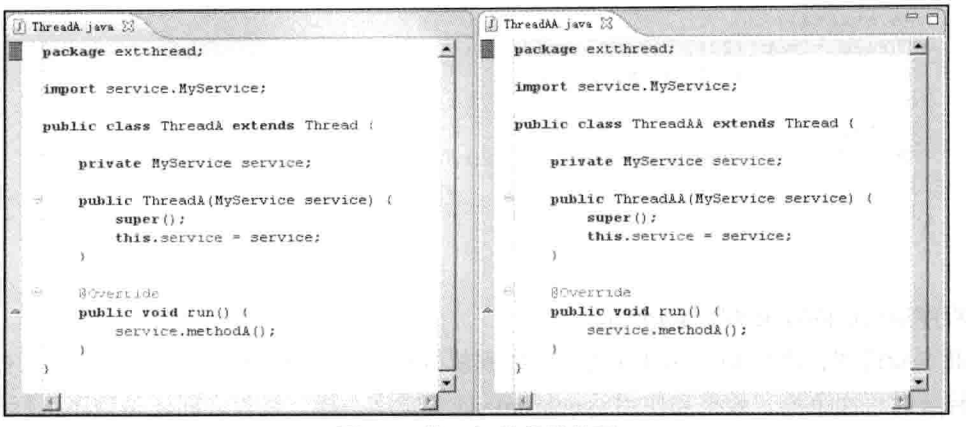
图 4-1 同步运行

4.1.2 使用 ReentrantLock 实现同步：测试 2

创建测试用的项目 ConditionTestMoreMethod，类 MyService.java 代码如下：

```
package service;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class MyService {
private Lock lock = new ReentrantLock();
public void methodA() {
    try {
        lock.lock();
        System.out.println("methodA begin ThreadName="
                + Thread.currentThread().getName() + " time="
                + System.currentTimeMillis());
        Thread.sleep(5000);
        System.out.println("methodA  end ThreadName="
                + Thread.currentThread().getName() + " time="
                + System.currentTimeMillis());
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
public void methodB() {
    try {
        lock.lock();
        System.out.println("methodB begin ThreadName="
                + Thread.currentThread().getName() + " time="
                + System.currentTimeMillis());
        Thread.sleep(5000);
        System.out.println("methodB  end ThreadName="
                + Thread.currentThread().getName() + " time="
                + System.currentTimeMillis());
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
}
```

第一组线程类代码如图 4-2 所示。



ThreadA.java

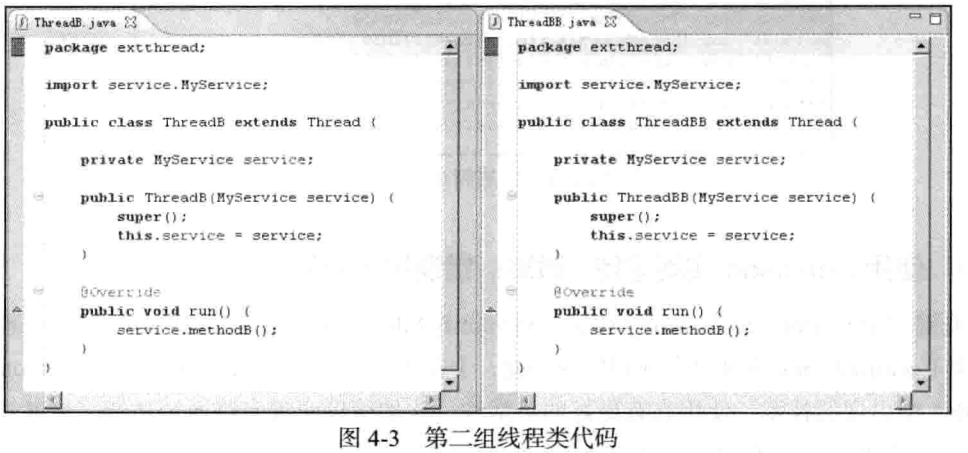
```
package extthread;

import service.MyService;

public class ThreadA extends Thread {

    private MyService service;

    public ThreadA(MyService service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.methodA();
    }
}
```

ThreadAA.java

```
package extthread;

import service.MyService;

public class ThreadAA extends Thread {

    private MyService service;

    public ThreadAA(MyService service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.methodA();
    }
}
```

图 4-2　第一组线程类代码

第二组线程类代码如图 4-3 所示。



ThreadB.java

```
package extthread;

import service.MyService;

public class ThreadB extends Thread {

    private MyService service;

    public ThreadB(MyService service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.methodB();
    }
}
```

ThreadBB.java

```
package extthread;

import service.MyService;

public class ThreadBB extends Thread {

    private MyService service;

    public ThreadBB(MyService service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.methodB();
    }
}
```

图 4-3　第二组线程类代码

类 Run.java 代码如下。

```
package test;
import service.MyService;
import extthread.ThreadA;
import extthread.ThreadAA;
import extthread.ThreadB;
import extthread.ThreadBB;
public class Run {
public static void main(String[] args) throws InterruptedException {
    MyService service = new MyService();
    ThreadA a = new ThreadA(service);
    a.setName("A");
    a.start();
    ThreadAA aa = new ThreadAA(service);
    aa.setName("AA");
```

```
        aa.start();
        Thread.sleep(100);
        ThreadB b = new ThreadB(service);
        b.setName("B");
        b.start();
        ThreadBB bb = new ThreadBB(service);
        bb.setName("BB");
        bb.start();
    }
}
```

程序运行后的效果如图 4-4 所示。

此实验说明，调用 lock.lock() 代码的线程就持有了“对象监视器”，其他线程只有等待锁被释放时再次争抢。效果和使用 synchronized 关键字一样，线程之间还是顺序执行的。

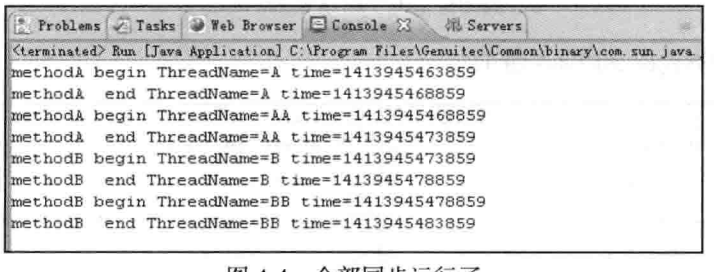


图 4-4 全部同步运行了

4.1.3 使用 Condition 实现等待 / 通知：错误用法与解决

关键字 synchronized 与 wait() 和 notify()/notifyAll() 方法相结合可以实现等待 / 通知模式，类 ReentrantLock 也可以实现同样的功能，但需要借助于 Condition 对象。Condition 类是在 JDK5 中出现的技术，使用它有更好的灵活性，比如可以实现多路通知功能，也就是在一个 Lock 对象里面可以创建多个 Condition（即对象监视器）实例，线程对象可以注册在指定的 Condition 中，从而可以有选择性地进行线程通知，在调度线程上更加灵活。

在使用 notify()/notifyAll() 方法进行通知时，被通知的线程却是由 JVM 随机选择的。但使用 ReentrantLock 结合 Condition 类是可以实现前面介绍过的“选择性通知”，这个功能是非常重要的，而且在 Condition 类中是默认提供的。

而 synchronized 就相当于整个 Lock 对象中只有一个单一的 Condition 对象，所有的线程都注册在它一个对象的身上。线程开始 notifyAll() 时，需要通知所有的 WAITING 线程，没有选择权，会出现相当大的效率问题。

创建 Java 项目 UseConditionWaitNotifyError，类 MyService.java 代码如下：

```
package service;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
```

```
import java.util.concurrent.locks.ReentrantLock;
public class MyService {
private Lock lock = new ReentrantLock();
private Condition condition = lock.newCondition();
public void await() {
    try {
        condition.await();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

类 ThreadA.java 代码如下：

```
package extthread;
import service.MyService;
public class ThreadA extends Thread {
private MyService service;
public ThreadA(MyService service) {
    super();
    this.service = service;
}
@Override
public void run() {
    service.await();
}
}
```

类 Run.java 代码如下：

```
package test;
import service.MyService;
import extthread.ThreadA;
public class Run {
public static void main(String[] args) {
    MyService service = new MyService();
    ThreadA a = new ThreadA(service);
    a.start();
}
}
```

程序运行结果如图 4-5 所示。

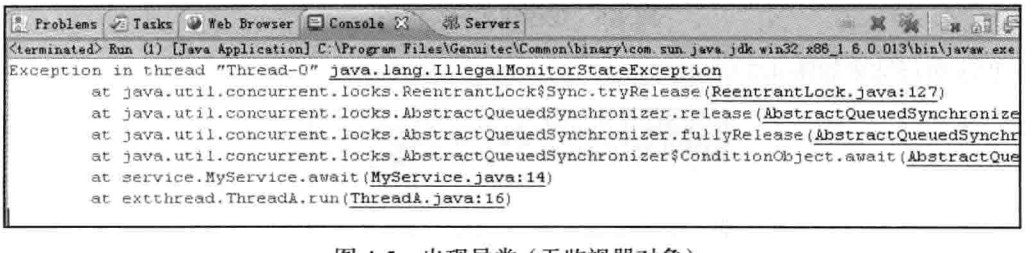


图 4-5　出现异常（无监视器对象）

报错的异常信息是监视器出错，解决的办法是必须在 condition.await() 方法调用之前调用 lock.lock() 代码获得同步监视器。

创建名称为 z3_ok 的 Java 项目，类 MyService.java 代码如下：

```
package service;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
public class MyService {
private ReentrantLock lock = new ReentrantLock();
private Condition condition = lock.newCondition();
public void waitMethod() {
    try {
        lock.lock();
        System.out.println("A");
        condition.await();
        System.out.println("B");
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
        System.out.println("锁释放了！ ");
    }
}
}
```

线程类和运行类代码如图 4-6 所示。

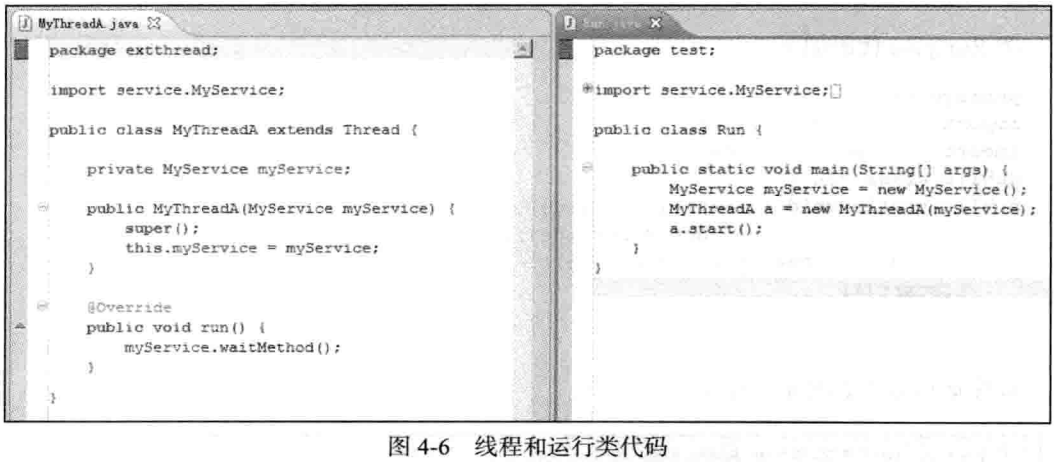


图 4-6 线程和运行类代码

程序运行结果如图 4-7 所示。

图 4-7 只打印字母 A

在控制台中只打印一个字母 A，原因是调用了 Condition 对象的 await() 方法，使当前执行任务的线程进入了等待 WAITING 状态。

4.1.4　正确使用 Condition 实现等待 / 通知

创建项目 UseConditionWaitNotifyOK，类 MyService.java 代码如下：

```
package service;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class MyService {
private Lock lock = new ReentrantLock();
public Condition condition = lock.newCondition();
public void await() {
    try {
        lock.lock();
        System.out.println(" await 时间为 " + System.currentTimeMillis());
        condition.await();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
public void signal() {
    try {
        lock.lock();
        System.out.println("signal 时间为 " + System.currentTimeMillis());
        condition.signal();
    } finally {
        lock.unlock();
    }
}
}
```

类 ThreadA.java 代码如下：

```
package extthread;
import service.MyService;
public class ThreadA extends Thread {
private MyService service;
public ThreadA(MyService service) {
    super();
    this.service = service;
}
@Override
public void run() {
    service.await();
}
}
```

类 Run.java 代码如下：

```
package test;
import service.MyService;
import extthread.ThreadA;
public class Run {
public static void main(String[] args) throws InterruptedException {
    MyService service = new MyService();
    ThreadA a = new ThreadA(service);
    a.start();
    Thread.sleep(3000);
    service.signal();
}
}
```

程序运行结果如图 4-8 所示。

成功实现等待 / 通知模式。

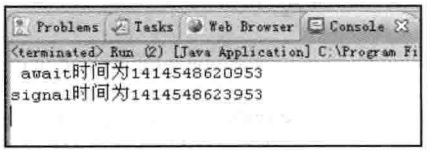


图 4-8 正常运行

Object 类中的 wait() 方法相当于 Condition 类中的 await() 方法。

Object 类中的 wait(long timeout) 方法相当于 Condition 类中的 await(long time, TimeUnit unit) 方法。

Object 类中的 notify() 方法相当于 Condition 类中的 signal() 方法。

Object 类中的 notifyAll() 方法相当于 Condition 类中的 signalAll() 方法。

4.1.5 使用多个 Condition 实现通知部分线程：错误用法

前面章节使用一个 Condition 对象来实现等待 / 通知模式，其实 Condition 对象也可以创建多个。那么一个 Condition 对象和多个 Condition 对象在使用上有什么区别呢？

创建 Java 项目 MustUseMoreCondition_Error，类 MyService.java 代码如下：

```
package service;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class MyService {
private Lock lock = new ReentrantLock();
public Condition condition = lock.newCondition();
public void awaitA() {
    try {
        lock.lock();
        System.out.println("begin awaitA 时间为 " + System.currentTimeMillis()
                + " ThreadName=" + Thread.currentThread().getName());
        condition.await();
        System.out.println("  end awaitA 时间为 " + System.currentTimeMillis()
                + " ThreadName=" + Thread.currentThread().getName());
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
    public void awaitB() {
        try {
            lock.lock();
            System.out.println("begin awaitB 时间为 " + System.currentTimeMillis()
                    + " ThreadName=" + Thread.currentThread().getName());
            condition.await();
            System.out.println("  end awaitB 时间为 " + System.currentTimeMillis()
                    + " ThreadName=" + Thread.currentThread().getName());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
    public void signalAll() {
        try {
            lock.lock();
            System.out.println("  signalAll 时间为 " + System.currentTimeMillis()
                    + " ThreadName=" + Thread.currentThread().getName());
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
```

ThreadA.java 和 ThreadB.java 代码如图 4-9 所示。

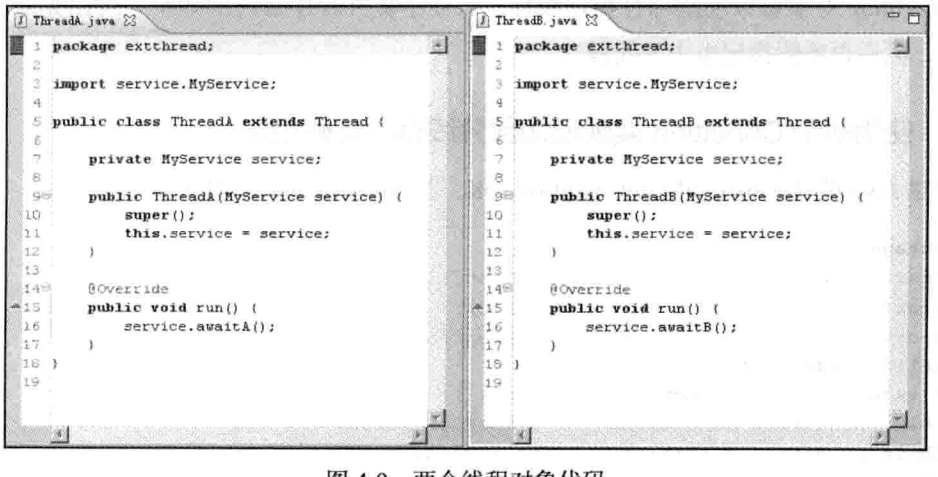


图 4-9　两个线程对象代码

类 Run.java 代码如下：

```
package test;
import service.MyService;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
public static void main(String[] args) throws InterruptedException {
    MyService service = new MyService();
    ThreadA a = new ThreadA(service);
    a.setName("A");
    a.start();
    ThreadB b = new ThreadB(service);
    b.setName("B");
    b.start();
    Thread.sleep(3000);
    service.signalAll();
}
}
```

程序运行后，线程 A 和 B 都被唤醒了，控制台输出如图 4-10 所示。

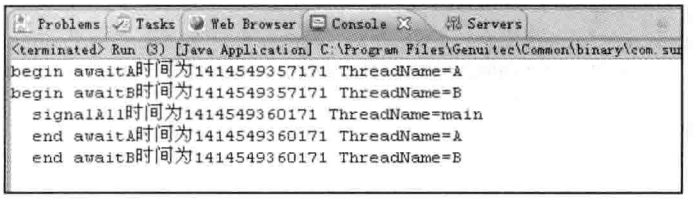


图 4-10 线性 A 和 B 都被唤醒

如果想单独唤醒部分线程该怎么处理呢？这时就有必要使用多个 Condition 对象了，也就是 Condition 对象可以唤醒部分指定线程，有助于提升程序运行的效率。可以先对线程进行分组，然后再唤醒指定组中的线程。

4.1.6 使用多个 Condition 实现通知部分线程：正确用法

创建 Java 项目 MustUseMoreCondition_OK，类 MyService.java 代码如下：

```
package service;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class MyService {
private Lock lock = new ReentrantLock();
public Condition conditionA = lock.newCondition();
public Condition conditionB = lock.newCondition();
public void awaitA() {
```

```
        try {
            lock.lock();
            System.out.println("begin awaitA 时间为 " + System.currentTimeMillis()
                    + " ThreadName=" + Thread.currentThread().getName());
            conditionA.await();
            System.out.println("  end awaitA 时间为 " + System.currentTimeMillis()
                    + " ThreadName=" + Thread.currentThread().getName());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
    public void awaitB() {
        try {
            lock.lock();
            System.out.println("begin awaitB 时间为 " + System.currentTimeMillis()
                    + " ThreadName=" + Thread.currentThread().getName());
            conditionB.await();
            System.out.println("  end awaitB 时间为 " + System.currentTimeMillis()
                    + " ThreadName=" + Thread.currentThread().getName());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
    public void signalAll_A() {
        try {
            lock.lock();
            System.out.println("  signalAll_A 时间为 " + System.currentTimeMillis()
                    + " ThreadName=" + Thread.currentThread().getName());
            conditionA.signalAll();
        } finally {
            lock.unlock();
        }
    }
    public void signalAll_B() {
        try {
            lock.lock();
            System.out.println("  signalAll_B 时间为 " + System.currentTimeMillis()
                    + " ThreadName=" + Thread.currentThread().getName());
            conditionB.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
```

类 ThreadA.java 和 ThreadB.java 代码如图 4-11 所示。

ThreadA.java

```
package extthread;

import service.MyService;

public class ThreadA extends Thread {

    private MyService service;

    public ThreadA(MyService service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.awaitA();
    }
}
```

ThreadB.java

```
package extthread;

import service.MyService;

public class ThreadB extends Thread {

    private MyService service;

    public ThreadB(MyService service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.awaitB();
    }
}
```

图 4-11 线程代码

类 Run.java 代码如下：

```
package test;
import service.MyService;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
public static void main(String[] args) throws InterruptedException {
    MyService service = new MyService();
    ThreadA a = new ThreadA(service);
    a.setName("A");
    a.start();
    ThreadB b = new ThreadB(service);
    b.setName("B");
    b.start();
    Thread.sleep(3000);
    service.signalAll_A();
}
}
```

程序运行后，只有线程 A 被唤醒了，控制台输出如图 4-12 所示。

Problems Tasks Web Browser Console Servers

```
Run (4) [Java Application] C:\Program Files\Genuitec\Common\binary\com.sun.java.jdk.wi
begin awaitA时间为1414549477531 ThreadName=A
begin awaitB时间为1414549477531 ThreadName=B
  signalAll_A时间为1414549480531 ThreadName=main
  end awaitA时间为1414549480531 ThreadName=A
```

图 4-12 线程 B 没有被唤醒

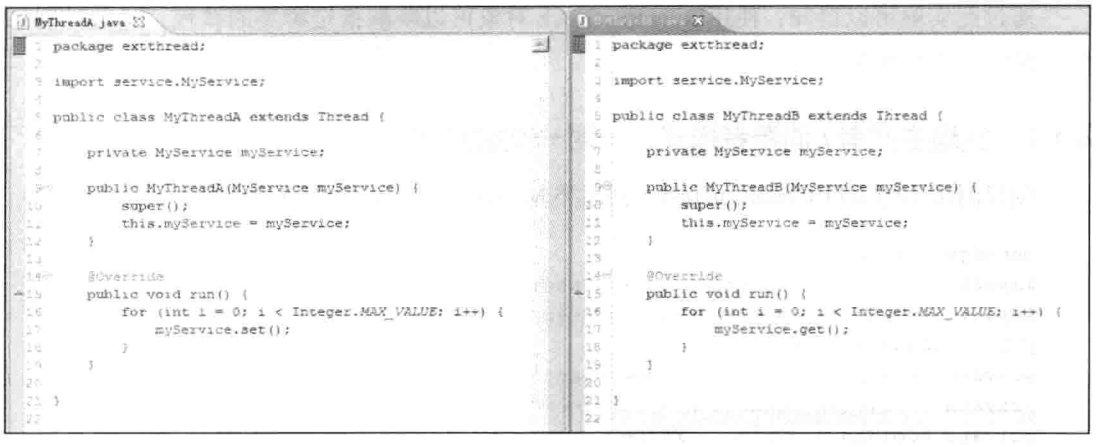
通过此实验可以得知，使用 ReentrantLock 对象可以唤醒指定种类的线程，这是控制部分线程行为的方便方式。

4.1.7　实现生产者 / 消费者模式：一对一交替打印

创建测试用的项目 ConditionTest，创建类 MyService.java，代码如下：

```
package service;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
public class MyService {
private ReentrantLock lock = new ReentrantLock();
private Condition condition = lock.newCondition();
private boolean hasValue = false;
public void set() {
    try {
        lock.lock();
        while (hasValue == true) {
            condition.await();
        }
        System.out.println("打印★ ");
        hasValue = true;
        condition.signal();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
public void get() {
    try {
        lock.lock();
        while (hasValue == false) {
            condition.await();
        }
        System.out.println("打印☆ ");
        hasValue = false;
        condition.signal();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
}
```

创建两个线程类代码，如图 4-13 所示。



```
MyThreadA.java
package extthread;

import service.MyService;

public class MyThreadA extends Thread {

    private MyService myService;

    public MyThreadA(MyService myService) {
        super();
        this.myService = myService;
    }

    @Override
    public void run() {
        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            myService.set();
        }
    }

}
```

```
package extthread;

import service.MyService;

public class MyThreadB extends Thread {

    private MyService myService;

    public MyThreadB(MyService myService) {
        super();
        this.myService = myService;
    }

    @Override
    public void run() {
        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            myService.get();
        }
    }

}
```

图 4-13 两个线程类

运行类 Run.java 代码如下：

```
package test;
import service.MyService;
import extthread.MyThreadA;
import extthread.MyThreadB;
public class Run {
public static void main(String[] args) throws InterruptedException {
    MyService myService = new MyService();
    MyThreadA a = new MyThreadA(myService);
    a.start();
    MyThreadB b = new MyThreadB(myService);
    b.start();
}
}
```

程序运行结果如图 4-14 所示。

通过使用 Condition 对象，成功实现交替打印的效果。

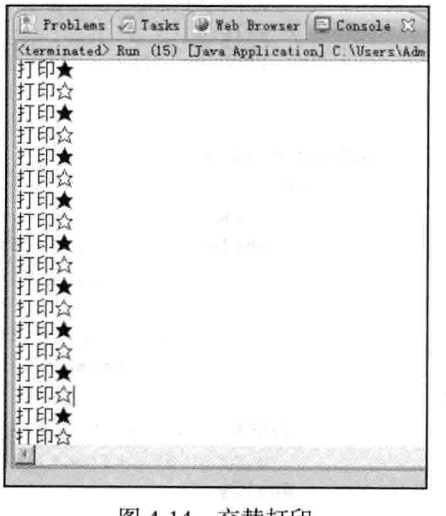


图 4-14 交替打印

4.1.8 实现生产者 / 消费者模式：多对多交替打印

创建新的项目 ConditionTestManyToMany，将 ConditionTest 项目中的所有源代码复制到新项目 ConditionTestManyToMany 中。

更改 MyService.java 类代码如下：

```
package service;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
public class MyService {
private ReentrantLock lock = new ReentrantLock();
```

```
private Condition condition = lock.newCondition();
private boolean hasValue = false;
public void set() {
    try {
        lock.lock();
        while (hasValue == true) {
            System.out.println("有可能★★连续");
            condition.await();
        }
        System.out.println("打印★");
        hasValue = true;
        condition.signal();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
public void get() {
    try {
        lock.lock();
        while (hasValue == false) {
            System.out.println("有可能☆☆连续");
            condition.await();
        }
        System.out.println("打印☆");
        hasValue = false;
        condition.signal();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
}
```

更改Run.java代码如下：

```
package test;
import service.MyService;
import extthread.MyThreadA;
import extthread.MyThreadB;
public class Run {
public static void main(String[] args) throws InterruptedException {
    MyService service = new MyService();
    MyThreadA[] threadA = new MyThreadA[10];
    MyThreadB[] threadB = new MyThreadB[10];
    for (int i = 0; i < 10; i++) {
        threadA[i] = new MyThreadA(service);
        threadB[i] = new MyThreadB(service);
        threadA[i].start();
        threadB[i].start();
```

程序运行后又出现假死，效果如图 4-15 所示。

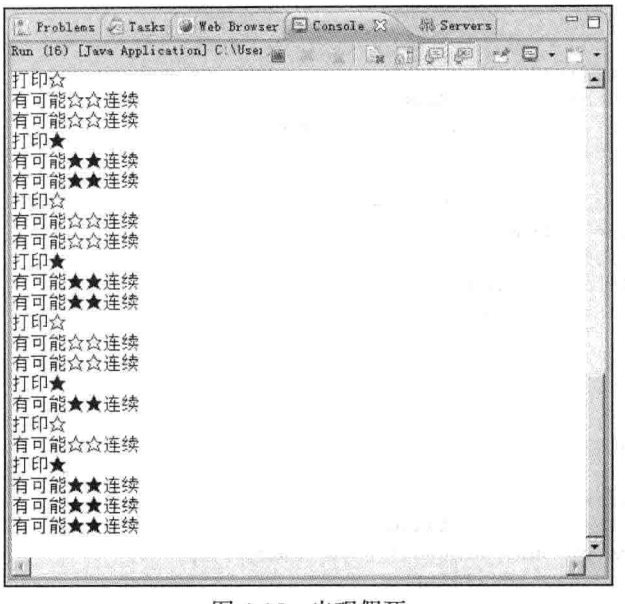


图 4-15 出现假死

根据第 3 章中的 notifyAll() 解决方案，可以使用 signalAll() 方法来解决。将 MyService.java 类中两处 signal() 代码改成 signalAll() 后，程序得到正确运行，效果如图 4-16 所示。

从控制台打印的日志可以发现，运行后不再出现假死状态，假死问题被解决了。

控制台中“打印★”和“打印☆”是交替输出的，但是“有可能★★连续”和“有可能☆☆连续”却不是交替输出的，有时候出现连续打印的情况。原因是程序中使用了一个 Condition 对象，再结合 signalAll() 方法来唤醒所有的线程，那么唤醒的线程就有可能是同类，所以就出现连续打印“有可能★★连续”或“有可能☆☆连续”的情况了。

4.1.9 公平锁与非公平锁

公平与非公平锁：锁 Lock 分为“公平锁”和“非公平锁”，公平锁表示线程获取锁的顺序是按照线程加锁的顺序来分配的，即先来先得的 FIFO 先进先出顺序。而非公平锁就是一种获取锁的抢占机制，是随机获得锁的，和公平锁不一样的就是先来的不一定先得到锁，这个方式可能造成某些线程一直拿不到锁，结果也就是不公平的了。

创建 Java 项目 Fair_noFair_test，创建 Service.java 类，代码如下：

```
package service;
import java.util.concurrent.locks.ReentrantLock;
public class Service {
private ReentrantLock lock;
public Service(boolean isFair) {
    super();
```

```
        lock = new ReentrantLock(isFair);
    }
    public void serviceMethod() {
        try {
            lock.lock();
            System.out.println("ThreadName=" + Thread.currentThread().getName()
                    + "获得锁定");
        } finally {
            lock.unlock();
        }
    }
}
```

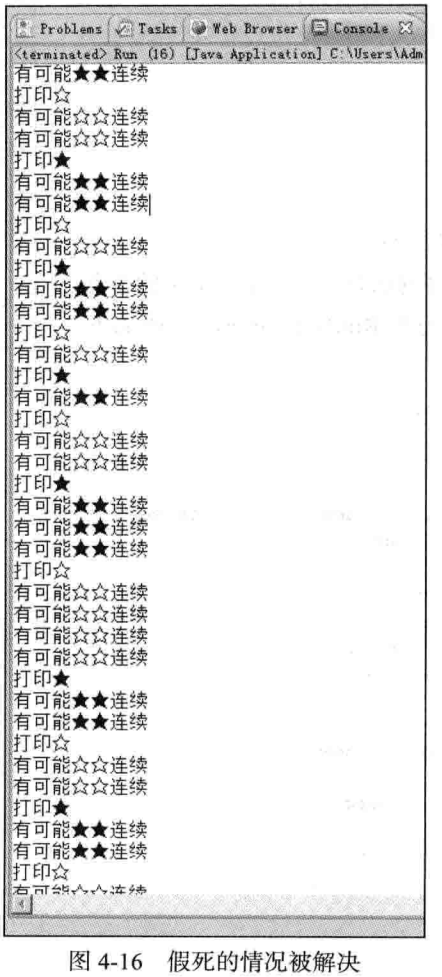


图 4-16　假死的情况被解决

创建公平锁测试的运行类 RunFair.java 代码如下：

```
package test.run;
import service.Service;
```

```
public class RunFair {
public static void main(String[] args) throws InterruptedException {
    final Service service = new Service(true);
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            System.out.println(" ★线程 " + Thread.currentThread().getName()
                    + " 运行了 ");
            service.serviceMethod();
        }
    };
    Thread[] threadArray = new Thread[10];
    for (int i = 0; i < 10; i++) {
        threadArray[i] = new Thread(runnable);
    }
    for (int i = 0; i < 10; i++) {
        threadArray[i].start();
    }
}
}
```

程序运行结果如图 4-17 所示。

图 4-17 运行结果

打印的结果基本是呈有序的状态，这就是公平锁的特点。

创建非公平锁测试的运行类 RunNotFair.java 代码如下：

```
package test.run;
import service.Service;
public class RunNotFair {
public static void main(String[] args) throws
InterruptedException {
    final Service service = new Service(false);
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            System.out.println(" ★线程 " + Thread.currentThread().getName()
                    + " 运行了 ");
            service.serviceMethod();
        }
    };
    Thread[] threadArray = new Thread[10];
    for (int i = 0; i < 10; i++) {
        threadArray[i] = new Thread(runnable);
    }
    for (int i = 0; i < 10; i++) {
        threadArray[i].start();
    }
}
}
```

程序运行结果如图 4-18 所示。

图 4-18 运行结果

非公平锁的运行结果基本上是乱序的，说明先 start() 启动的线程不代表先获得锁。

4.1.10　方法 getHoldCount()、getQueueLength() 和 getWaitQueueLength() 的测试

创建测试用的项目 lockMethodTest1。

1）方法 int getHoldCount() 的作用是查询当前线程保持此锁定的个数，也就是调用 lock() 方法的次数。

创建名称为 test1 的 package 包，创建类 Service.java，代码如下：

```
package test1;
import java.util.concurrent.locks.ReentrantLock;
public class Service {
private ReentrantLock lock = new ReentrantLock();
public void serviceMethod1() {
    try {
        lock.lock();
        System.out.println("serviceMethod1 getHoldCount="
                + lock.getHoldCount());
        serviceMethod2();
    } finally {
        lock.unlock();
    }
}
public void serviceMethod2() {
    try {
        lock.lock();
        System.out.println("serviceMethod2 getHoldCount="
                + lock.getHoldCount());
    } finally {
        lock.unlock();
    }
}
}
```

创建类 Run.java 代码如下：

```
package test1;
public class Run {
public static void main(String[] args) {
    Service service = new Service();
    service.serviceMethod1();
}
}
```

程序运行结果如图 4-19 所示。

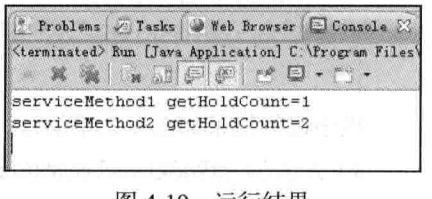


图 4-19　运行结果

2）方法 int getQueueLength() 的作用是返回正等待获取此锁定的线程估计数，比如有 5 个线程，1 个线程首先执行 await() 方法，那么在调用 getQueueLength() 方法后返回值是 4，说明有 4 个线程同时在等待 lock 的释放。

创建名称为 test2 的 package 包，创建类 Service.java，代码如下：

```
package test2;
import java.util.concurrent.locks.ReentrantLock;
public class Service {
public ReentrantLock lock = new ReentrantLock();
public void serviceMethod1() {
    try {
        lock.lock();
        System.out.println("ThreadName=" + Thread.currentThread().getName()
                + "进入方法！ ");
        Thread.sleep(Integer.MAX_VALUE);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
}
```

创建类 Run.java 代码如下：

```
package test2;
public class Run {
public static void main(String[] args) throws InterruptedException {
    final Service service = new Service();
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            service.serviceMethod1();
        }
    };
    Thread[] threadArray = new Thread[10];
    for (int i = 0; i < 10; i++) {
        threadArray[i] = new Thread(runnable);
    }
    for (int i = 0; i < 10; i++) {
        threadArray[i].start();
    }
    Thread.sleep(2000);
    System.out.println("有线程数:" + service.lock.getQueueLength() + "在等待获取锁！ ");
}
}
```

程序运行结果如图 4-20 所示。

3）方法 int getWaitQueueLength(Condition condition) 的作用是返回等待与此锁定相关的给定条件 Condition 的线程估计数，比如有 5 个线程，每个线程都执行了同一个 condition 对象的 await() 方法，则调用 getWaitQueueLength(Condition condition) 方法时返回的 int 值是 5。

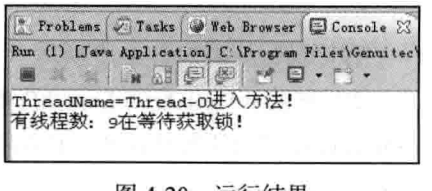


图 4-20 运行结果

创建名称为 test3 的 package 包，创建类 Service.java，代码如下：

```
package test3;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
public class Service {
private ReentrantLock lock = new ReentrantLock();
private Condition newCondition = lock.newCondition();
public void waitMethod() {
    try {
        lock.lock();
        newCondition.await();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
public void notityMethod() {
    try {
        lock.lock();
        System.out.println("有 " + lock.getWaitQueueLength(newCondition)
                + "个线程正在等待 newCondition");
        newCondition.signal();
    } finally {
        lock.unlock();
    }
}
}
```

创建类 Run.java 代码如下：

```
package test3;
public class Run {
public static void main(String[] args) throws InterruptedException {
    final Service service = new Service();
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            service.waitMethod();
        }
    };
    Thread[] threadArray = new Thread[10];
    for (int i = 0; i < 10; i++) {
        threadArray[i] = new Thread(runnable);
    }
    for (int i = 0; i < 10; i++) {
        threadArray[i].start();
    }
    Thread.sleep(2000);
    service.notityMethod();
}
}
```

程序运行结果如图 4-21 所示。

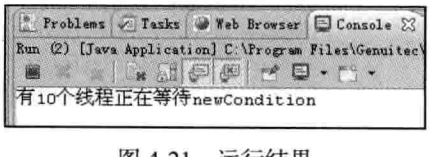


图 4-21　运行结果

4.1.11 方法 hasQueuedThread()、hasQueuedThreads() 和 hasWaiters() 的测试

创建测试用的项目 lockMethodTest2。

1）方法 boolean hasQueuedThread(Thread thread) 的作用是查询指定的线程是否正在等待获取此锁定。

方法 boolean hasQueuedThreads() 的作用是查询是否有线程正在等待获取此锁定。

创建名称为 test1 的 package 包，创建类 Service.java，代码如下：

```
package test1;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
public class Service {
public ReentrantLock lock = new ReentrantLock();
public Condition newCondition = lock.newCondition();
public void waitMethod() {
    try {
        lock.lock();
        Thread.sleep(Integer.MAX_VALUE);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
}
```

创建类 Run.java 代码如下：

```
package test1;
public class Run {
public static void main(String[] args) throws InterruptedException {
    final Service service = new Service();
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            service.waitMethod();
        }
    };
    Thread threadA = new Thread(runnable);
    threadA.start();
    Thread.sleep(500);
    Thread threadB = new Thread(runnable);
    threadB.start();
    Thread.sleep(500);
    System.out.println(service.lock.hasQueuedThread(threadA));
    System.out.println(service.lock.hasQueuedThread(threadB));
    System.out.println(service.lock.hasQueuedThreads());
}
}
```

程序运行结果如图 4-22 所示。

图 4-22　运行结果

2）方法 boolean hasWaiters(Condition condition) 的作用是查询是否有线程正在等待与此锁定有关的 condition 条件。

创建名称为 test2 的 package 包，创建类 Service.java，代码如下：

```
package test2;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
public class Service {
private ReentrantLock lock = new ReentrantLock();
private Condition newCondition = lock.newCondition();
public void waitMethod() {
    try {
        lock.lock();
        newCondition.await();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
public void notityMethod() {
    try {
        lock.lock();
        System.out.println("有没有线程正在等待 newCondition？ "
                + lock.hasWaiters(newCondition) + " 线程数是多少？ "
                + lock.getWaitQueueLength(newCondition));
        newCondition.signal();
    } finally {
        lock.unlock();
    }
}
}
```

创建类 Run.java 代码如下：

```
package test2;
public class Run {
public static void main(String[] args) throws InterruptedException {
    final Service service = new Service();
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            service.waitMethod();
        }
    };
    Thread[] threadArray = new Thread[10];
    for (int i = 0; i < 10; i++) {
        threadArray[i] = new Thread(runnable);
    }
```

```
        for (int i = 0; i < 10; i++) {
            threadArray[i].start();
        }
        Thread.sleep(2000);
        service.notityMethod();
    }
}
```

程序运行结果如图 4-23 所示。

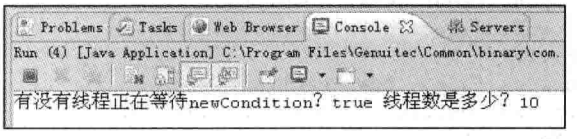


图 4-23 运行结果

4.1.12 方法 isFair()、isHeldByCurrentThread() 和 isLocked() 的测试

创建测试用的项目 lockMethodTest3。

1）方法 boolean isFair() 的作用是判断是不是公平锁。

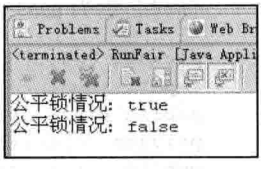
创建名称为 test1 的 package 包，创建类 Service.java，代码如下：

```
package test1;
import java.util.concurrent.locks.ReentrantLock;
public class Service {
private ReentrantLock lock;
public Service(boolean isFair) {
    super();
    lock = new ReentrantLock(isFair);
}
public void serviceMethod() {
    try {
        lock.lock();
        System.out.println("公平锁情况: " + lock.isFair());
    } finally {
        lock.unlock();
    }
}
}
```

创建类 Run.java 代码如下：

```
package test1;
public class Run {
public static void main(String[] args) throws InterruptedException {
    final Service service1 = new Service(true);
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            service1.serviceMethod();
        }
```

```
        };
        Thread thread = new Thread(runnable);
        thread.start();
        final Service service2 = new Service(false);
        runnable = new Runnable() {
            @Override
            public void run() {
                service2.serviceMethod();
            }
        };
        thread = new Thread(runnable);
        thread.start();
    }
}
```



程序运行结果如图 4-24 所示。

图 4-24　运行结果

在默认的情况下，ReentrantLock 类使用的是非公平锁。

2）方法 boolean isHeldByCurrentThread() 的作用是查询当前线程是否保持此锁定。

创建名称为 test2 的 package 包，创建类 Service.java，代码如下：

```
package test2;
import java.util.concurrent.locks.ReentrantLock;
public class Service {
private ReentrantLock lock;
public Service(boolean isFair) {
    super();
    lock = new ReentrantLock(isFair);
}
public void serviceMethod() {
    try {
        System.out.println(lock.isHeldByCurrentThread());
        lock.lock();
        System.out.println(lock.isHeldByCurrentThread());
    } finally {
        lock.unlock();
    }
}
}
```

创建类 Run.java 代码如下：

```
package test2;
public class Run {
public static void main(String[] args) throws InterruptedException {
    final Service service1 = new Service(true);
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            service1.serviceMethod();
        }
    };
    Thread thread = new Thread(runnable);
```

```
        thread.start();
    }
}
```

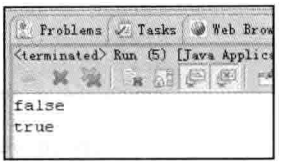


图 4-25 运行结果

程序运行结果如图 4-25 所示。

3）方法 boolean isLocked() 的作用是查询此锁定是否由任意线程保持。

创建名称为 test3 的 package 包，创建类 Service.java，代码如下：

```
package test3;
import java.util.concurrent.locks.ReentrantLock;
public class Service {
private ReentrantLock lock;
public Service(boolean isFair) {
    super();
    lock = new ReentrantLock(isFair);
}
public void serviceMethod() {
    try {
        System.out.println(lock.isLocked());
        lock.lock();
        System.out.println(lock.isLocked());
    } finally {
        lock.unlock();
    }
}
}
```

创建类 Run.java 代码如下：

```
package test3;
public class Run {
public static void main(String[] args) throws InterruptedException {
    final Service service1 = new Service(true);
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            service1.serviceMethod();
        }
    };
    Thread thread = new Thread(runnable);
    thread.start();
}
}
```

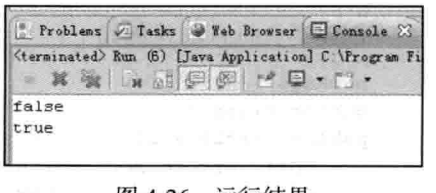


图 4-26 运行结果

程序运行结果如图 4-26 所示。

4.1.13 方法 lockInterruptibly()、tryLock() 和 tryLock(long timeout,TimeUnit unit) 的测试

1）方法 void lockInterruptibly() 的作用是：如果当前线程未被中断，则获取锁定，如果

已经被中断则出现异常。

创建测试用的项目 lockInterruptiblyTest1，类 MyService.java 代码如下：

```
package service;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
public class MyService {
public ReentrantLock lock = new ReentrantLock();
private Condition condition = lock.newCondition();
public void waitMethod() {
    try {
        lock.lock();
        System.out
                .println("lock begin " + Thread.currentThread().getName());
        for (int i = 0; i < Integer.MAX_VALUE / 10; i++) {
            String newString = new String();
            Math.random();
        }
        System.out
                .println("lock   end " + Thread.currentThread().getName());
    } finally {
        if (lock.isHeldByCurrentThread()) {
            lock.unlock();
        }
    }
}
}
```

类 Run.java 代码如下：

```
package test;
import service.MyService;
public class Run {
public static void main(String[] args) throws InterruptedException {
    final MyService service = new MyService();
    Runnable runnableRef = new Runnable() {
        @Override
        public void run() {
            service.waitMethod();
        }
    };
    Thread threadA = new Thread(runnableRef);
    threadA.setName("A");
    threadA.start();
    Thread.sleep(500);
    Thread threadB = new Thread(runnableRef);
    threadB.setName("B");
    threadB.start();
    threadB.interrupt();//打标记
    System.out.println("main end!");
}
}
```

没有出现异常，A、B 线程正常结束，按钮变灰。程序运行结果如图 4-27 所示。

前面实验使用的是 Lock() 方法，说明线程 B 被 interrupt 中断了，那么执行 lock() 则不出现异常，正常执行。

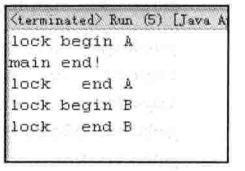


图 4-27 运行结果

如果使用 lockInterruptibly 方法会是什么结果呢？

创建测试用的项目 lockInterruptiblyTest2，将项目 lockInterruptiblyTest1 中的所有源代码复制到 lockInterruptiblyTest2 项目中，将类 MyService.java 中原有代码“lock.lock()”；变成“lock.lockInterruptibly();”。

程序运行结果如图 4-28 所示。

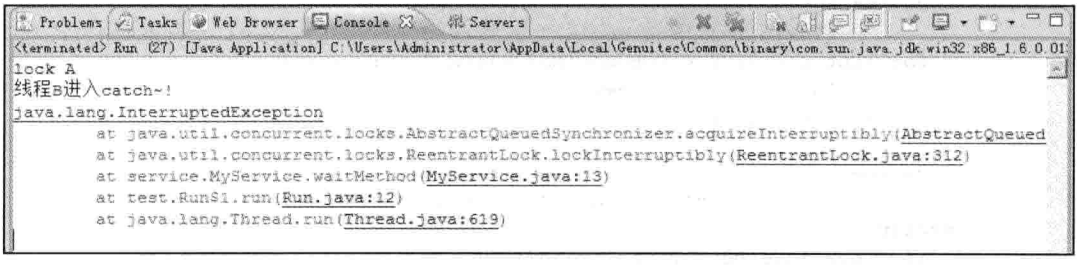


图 4-28 线程 B 被中断后调用 lockInterruptibly 方法报异常

2）方法 boolean tryLock() 的作用是，仅在调用时锁定未被另一个线程保持的情况下，才获取该锁定。

创建测试用的项目 tryLockTest，类 MyService.java 代码如下：

```
package service;
import java.util.concurrent.locks.ReentrantLock;
public class MyService {
public ReentrantLock lock = new ReentrantLock();
public void waitMethod() {
    if (lock.tryLock()) {
        System.out.println(Thread.currentThread().getName() + "获得锁");
    } else {
        System.out.println(Thread.currentThread().getName() + "没有获得锁");
    }
}
}
```

运行类代码如下：

```
package test;
import service.MyService;
public class Run {
public static void main(String[] args) throws InterruptedException {
    final MyService service = new MyService();
    Runnable runnableRef = new Runnable() {
        @Override
        public void run() {
```

```
            service.waitMethod();
        }
    };
    Thread threadA = new Thread(runnableRef);
    threadA.setName("A");
    threadA.start();
    Thread threadB = new Thread(runnableRef);
    threadB.setName("B");
    threadB.start();
}
}
```

程序运行结果如图 4-29 所示。

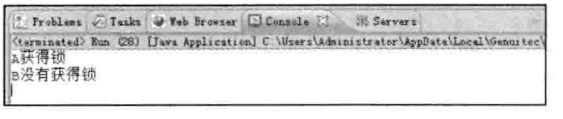


图 4-29　运行结果

3）方法 boolean tryLock(long timeout, TimeUnit unit) 的作用是，如果锁定在给定等待时间内没有被另一个线程保持，且当前线程未被中断，则获取该锁定。

创建测试用的项目 tryLock_param，类 MyService.java 代码如下：

```
package service;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;
public class MyService {
public ReentrantLock lock = new ReentrantLock();
public void waitMethod() {
    try {
        if (lock.tryLock(3, TimeUnit.SECONDS)) {
            System.out.println("       " + Thread.currentThread().getName()
                    + "获得锁的时间: " + System.currentTimeMillis());
            Thread.sleep(10000);
        } else {
            System.out.println("       " + Thread.currentThread().getName()
                    + "没有获得锁");
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        if (lock.isHeldByCurrentThread()) {
            lock.unlock();
        }
    }
}
}
```

运行类 Run2.java 代码如下：

```
package test;
```

```
import service.MyService;
public class Run {
public static void main(String[] args) throws InterruptedException {
    final MyService service = new MyService();
    Runnable runnableRef = new Runnable() {
        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName()
                    + "调用 waitMethod 时间: " + System.currentTimeMillis());
            service.waitMethod();
        }
    };
    Thread threadA = new Thread(runnableRef);
    threadA.setName("A");
    threadA.start();
    Thread threadB = new Thread(runnableRef);
    threadB.setName("B");
    threadB.start();
}
}
```

程序运行结果如图 4-30 所示。

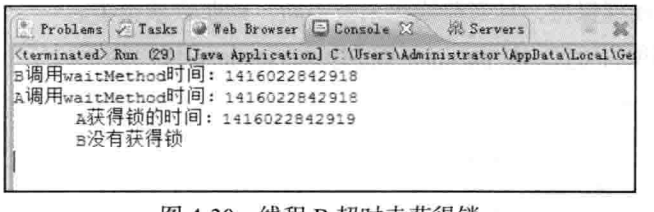


图 4-30 线程 B 超时未获得锁

4.1.14 方法 awaitUninterruptibly() 的使用

创建名称为 awaitUninterruptiblyTest_1 的项目，类 Service.java 代码如下：

```
package service;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
public class Service {
private ReentrantLock lock = new ReentrantLock();
private Condition condition = lock.newCondition();
public void testMethod() {
    try {
        lock.lock();
        System.out.println("wait begin");
        condition.await();
        System.out.println("wait   end");
    } catch (InterruptedException e) {
        e.printStackTrace();
        System.out.println("catch");
    } finally {
```

```
            lock.unlock();
        }
    }
}
```

线程类 MyThread.java 代码如下：

```
package extthread;
import service.Service;
public class MyThread extends Thread {
private Service service;
public MyThread(Service service) {
    super();
    this.service = service;
}
@Override
public void run() {
    service.testMethod();
}
}
```

运行类 Run.java 代码如下：

```
package test;
import service.Service;
import extthread.MyThread;
public class Run {
public static void main(String[] args) {
    try {
        Service service = new Service();
        MyThread myThread = new MyThread(service);
        myThread.start();
        Thread.sleep(3000);
        myThread.interrupt();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

程序运行后出现异常，这是正常现象。效果如图 4-31 所示。

图 4-31　程序运行出现异常

创建测试用的项目 awaitUninterruptiblyTest_2，将 awaitUninterruptiblyTest_1 项目中的所有 java 类复制到 awaitUninterruptiblyTest_2 中。

更改 Service.java 类代码如下：

```
package service;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
public class Service {
private ReentrantLock lock = new ReentrantLock();
private Condition condition = lock.newCondition();
public void testMethod() {
    try {
        lock.lock();
        System.out.println("wait begin");
        condition.awaitUninterruptibly();
        System.out.println("wait   end");
    } finally {
        lock.unlock();
    }
}
}
```

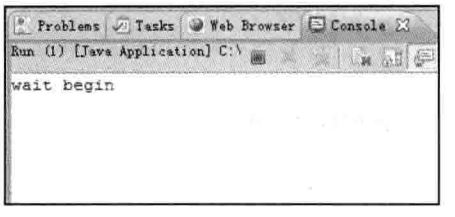


图 4-32 正常运行并没有异常发生

程序运行后的效果如图 4-32 所示。

4.1.15 方法 awaitUntil() 的使用

创建测试用的项目 awaitUntilTest，两个线程类代码如图 4-33 所示。

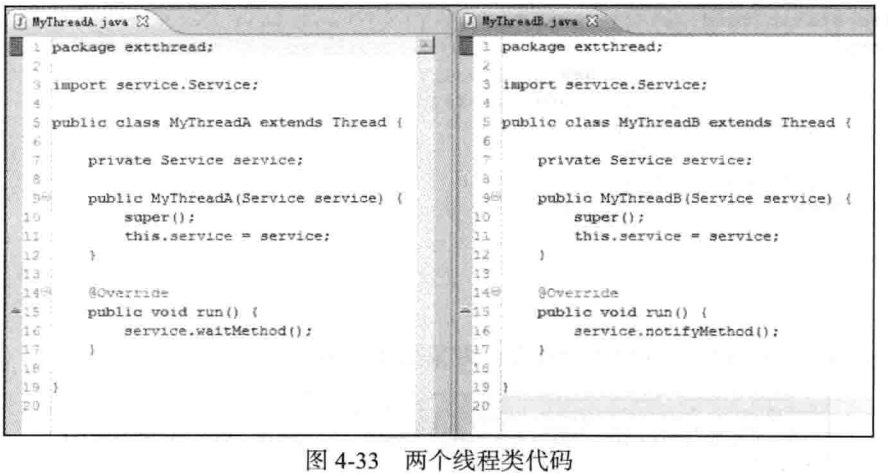


图 4-33 两个线程类代码

类 Service.java 代码如下：

```
package service;
import java.util.Calendar;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
public class Service {
private ReentrantLock lock = new ReentrantLock();
private Condition condition = lock.newCondition();
public void waitMethod() {
```

```
        try {
            Calendar calendarRef = Calendar.getInstance();
            calendarRef.add(Calendar.SECOND, 10);
            lock.lock();
            System.out
                    .println("wait begin timer=" + System.currentTimeMillis());
            condition.awaitUntil(calendarRef.getTime());
            System.out
                    .println("wait   end timer=" + System.currentTimeMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
    public void notifyMethod() {
        try {
            Calendar calendarRef = Calendar.getInstance();
            calendarRef.add(Calendar.SECOND, 10);
            lock.lock();
            System.out
                    .println("notify begin timer=" + System.currentTimeMillis());
            condition.signalAll();
            System.out
                    .println("notify   end timer=" + System.currentTimeMillis());
        } finally {
            lock.unlock();
        }
    }
}
```

创建运行类 Run1.java 代码如下：

```
package test;
import service.Service;
import extthread.MyThreadA;
import extthread.MyThreadB;
public class Run1 {
public static void main(String[] args) throws InterruptedException {
    Service service = new Service();
    MyThreadA myThreadA = new MyThreadA(service);
    myThreadA.start();
}
}
```

程序运行后的效果如图 4-34 所示。

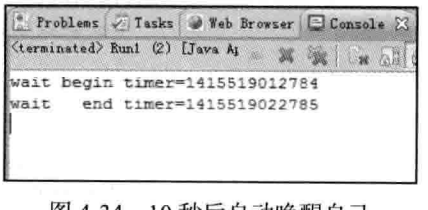


图 4-34　10 秒后自动唤醒自己

创建运行类 Run2.java 代码如下：

```
package test;
import service.Service;
import extthread.MyThreadA;
import extthread.MyThreadB;
public class Run2 {
public static void main(String[] args) throws InterruptedException {
    Service service = new Service();
```

```
        MyThreadA myThreadA = new MyThreadA(service);
        myThreadA.start();
        Thread.sleep(2000);
        MyThreadB myThreadB = new MyThreadB(service);
        myThreadB.start();
    }
}
```

程序运行后的效果如图 4-35 所示。

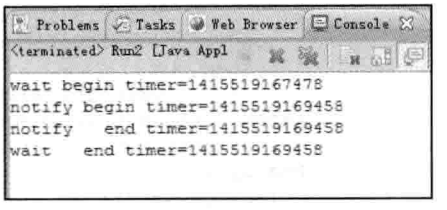


图 4-35 2 秒后被其他线程所唤醒

说明线程在等待时间到达前，可以被其他线程提前唤醒。

4.1.16 使用 Condition 实现顺序执行

使用 Condition 对象可以对线程执行的业务进行排序规划。

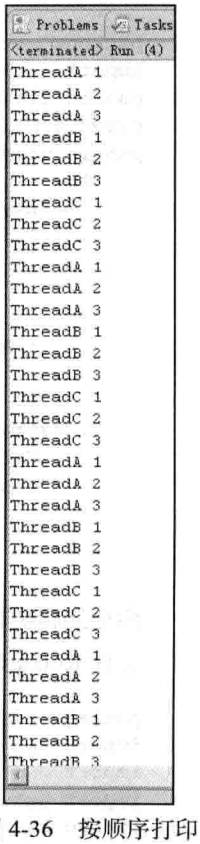
创建实验用的项目 condition123，类 F.java 代码如下：

```
package finaltools;
public class F {
volatile public static int nextPrintWho = 1;
}
```

创建运行类 Run.java 代码如下：

```
package test.run;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
public class Run {
volatile private static int nextPrintWho = 1;
private static ReentrantLock lock = new ReentrantLock();
final private static Condition conditionA = lock.newCondition();
final private static Condition conditionB = lock.newCondition();
final private static Condition conditionC = lock.newCondition();
public static void main(String[] args) {
    Thread threadA = new Thread() {
        public void run() {
            try {
                lock.lock();
                while (nextPrintWho != 1) {
                    conditionA.await();
                }
                for (int i = 0; i < 3; i++) {
                    System.out.println("ThreadA " + (i + 1));
                }
                nextPrintWho = 2;
                conditionB.signalAll();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }
```

```
        };
        Thread threadB = new Thread() {
            public void run() {
                try {
                    lock.lock();
                    while (nextPrintWho != 2) {
                        conditionB.await();
                    }
                    for (int i = 0; i < 3; i++) {
                        System.out.println("ThreadB " + (i + 1));
                    }
                    nextPrintWho = 3;
                    conditionC.signalAll();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    lock.unlock();
                }
            }
        };
        Thread threadC = new Thread() {
            public void run() {
                try {
                    lock.lock();
                    while (nextPrintWho != 3) {
                        conditionC.await();
                    }
                    for (int i = 0; i < 3; i++) {
                        System.out.println("ThreadC " + (i + 1));
                    }
                    nextPrintWho = 1;
                    conditionA.signalAll();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    lock.unlock();
                }
            }
        };
        Thread[] aArray = new Thread[5];
        Thread[] bArray = new Thread[5];
        Thread[] cArray = new Thread[5];
        for (int i = 0; i < 5; i++) {
            aArray[i] = new Thread(threadA);
            bArray[i] = new Thread(threadB);
            cArray[i] = new Thread(threadC);
            aArray[i].start();
            bArray[i].start();
            cArray[i].start();
        }
    }
}
```



程序运行结果如图 4-36 所示。

图 4-36　按顺序打印

4.2 使用 ReentrantReadWriteLock 类

类 ReentrantLock 具有完全互斥排他的效果，即同一时间只有一个线程在执行 ReentrantLock.lock() 方法后面的任务。这样做虽然保证了实例变量的线程安全性，但效率却是非常低下的。所以在 JDK 中提供了一种读写锁 ReentrantReadWriteLock 类，使用它可以加快运行效率，在某些不需要操作实例变量的方法中，完全可以使用读写锁 ReentrantReadWriteLock 来提升该方法的代码运行速度。

读写锁表示也有两个锁，一个是读操作相关的锁，也称为共享锁；另一个是写操作相关的锁，也叫排他锁。也就是多个读锁之间不互斥，读锁与写锁互斥，写锁与写锁互斥。在没有线程 Thread 进行写入操作时，进行读取操作的多个 Thread 都可以获取读锁，而进行写入操作的 Thread 只有在获取写锁后才能进行写入操作。即多个 Thread 可以同时进行读取操作，但是同一时刻只允许一个 Thread 进行写入操作。

4.2.1 类 ReentrantReadWriteLock 的使用：读读共享

创建 Java 项目 ReadWriteLockBegin1，类 Service.java 代码如下：

```
package service;
import java.util.concurrent.locks.ReentrantReadWriteLock;
public class Service {
private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
public void read() {
    try {
        try {
            lock.readLock().lock();
            System.out.println("获得读锁 " + Thread.currentThread().getName()
                    + " " + System.currentTimeMillis());
            Thread.sleep(10000);
        } finally {
            lock.readLock().unlock();
        }
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
}
```

两个线程类代码如图 4-37 所示。

文件 Run.java 代码如下：

```
package test;
import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
```

```
public static void main(String[] args) {
    Service service = new Service();
    ThreadA a = new ThreadA(service);
    a.setName("A");
    ThreadB b = new ThreadB(service);
    b.setName("B");
    a.start();
    b.start();
}
}
```

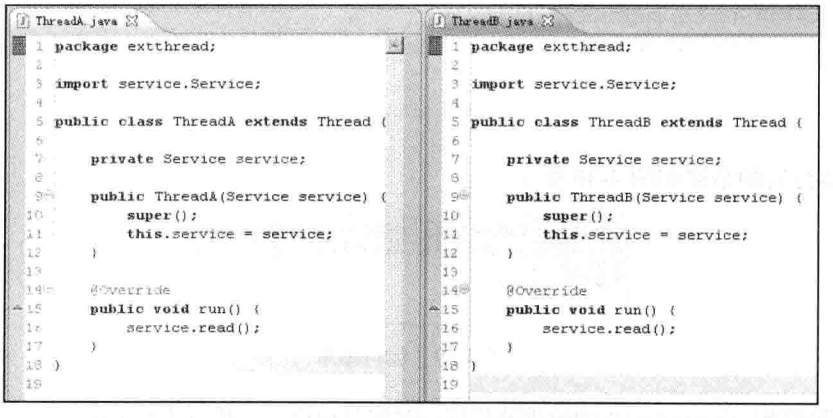


图 4-37　两个线程类代码

程序运行后的结果如图 4-38 所示。

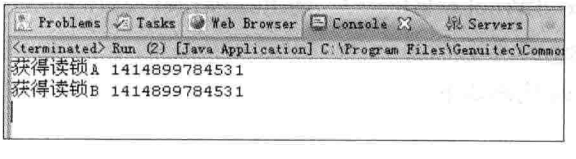


图 4-38　运行结果

从控制台中打印的时间来看，两个线程几乎同时进入 lock() 方法后面的代码。说明在此使用了 lock.readLock() 读锁可以提高程序运行效率，允许多个线程同时执行 lock() 方法后面的代码。

4.2.2　类 ReentrantReadWriteLock 的使用：写写互斥

创建 Java 项目 ReadWriteLockBegin2，将 ReadWriteLockBegin1 中的所有源代码复制到项目 ReadWriteLockBegin2 中。

更改类 Service.java 代码如下：

```
package service;
import java.util.concurrent.locks.ReentrantReadWriteLock;
public class Service {
```

```
private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
public void write() {
    try {
        try {
            lock.writeLock().lock();
            System.out.println("获得写锁 " + Thread.currentThread().getName()
                    + " " + System.currentTimeMillis());
            Thread.sleep(10000);
        } finally {
            lock.writeLock().unlock();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

程序运行后的结果如图 4-39 所示。

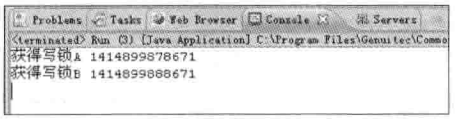


图 4-39 运行结果

使用写锁代码 lock.writeLock() 的效果就是同一时间只允许一个线程执行 lock() 方法后面的代码。

4.2.3 类 ReentrantReadWriteLock 的使用：读写互斥

创建 Java 项目 ReadWriteLockBegin3，将 ReadWriteLockBegin2 中的所有源代码复制到项目 ReadWriteLockBegin3 中。

更改类 Service.java 代码如下：

```
package service;
import java.util.concurrent.locks.ReentrantReadWriteLock;
public class Service {
private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
public void read() {
    try {
        try {
            lock.readLock().lock();
            System.out.println("获得读锁 " + Thread.currentThread().getName()
                    + " " + System.currentTimeMillis());
            Thread.sleep(10000);
        } finally {
            lock.readLock().unlock();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

```
public void write() {
    try {
        try {
            lock.writeLock().lock();
            System.out.println("获得写锁 " + Thread.currentThread().getName()
                    + " " + System.currentTimeMillis());
            Thread.sleep(10000);
        } finally {
            lock.writeLock().unlock();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

运行类 Run.java 代码更改如下：

```
package test;
import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
public static void main(String[] args) throws InterruptedException {
    Service service = new Service();
    ThreadA a = new ThreadA(service);
    a.setName("A");
    a.start();
    Thread.sleep(1000);
    ThreadB b = new ThreadB(service);
    b.setName("B");
    b.start();
}
}
```

程序运行后的结果如图 4-40 所示。

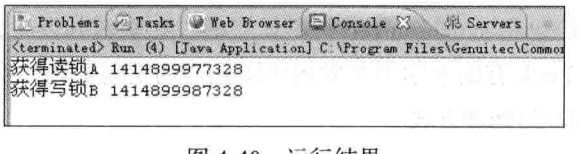


图 4-40　运行结果

此实验说明“读写”操作是互斥的，而且下一个示例说明“写读”操作也是互斥的。即只要出现“写操作”的过程，就是互斥的。

4.2.4　类 ReentrantReadWriteLock 的使用：写读互斥

创建 Java 项目 ReadWriteLockBegin4，将 ReadWriteLockBegin3 中的所有源代码复制到项目 ReadWriteLockBegin4 中。

更改运行类 Run.java 代码更改如下：

```
package test;
import service.Service;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
public static void main(String[] args) throws InterruptedException {
    Service service = new Service();
    ThreadB b = new ThreadB(service);
    b.setName("B");
    b.start();
    Thread.sleep(1000);
    ThreadA a = new ThreadA(service);
    a.setName("A");
    a.start();
}
}
```

程序运行后的结果如图 4-41 所示。

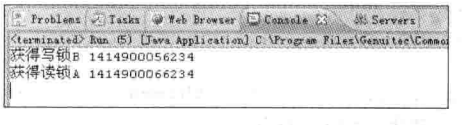


图 4-41 运行结果

从控制台中打印的结果来看，“写读”操作也是互斥的。

“读写”、“写读”和“写写”都是互斥的；而“读读”是异步的，非互斥的。

4.3 本章总结

本章的学习已经完毕，在本章中完全可以使用 Lock 对象将 synchronized 关键字替换掉，而且其具有的独特功能也是 synchronized 所不具有的。在学习并发时，Lock 是 synchronized 关键字的进阶，掌握 Lock 有助于学习并发包中源代码的实现原理，在并发包中大量的类使用了 Lock 接口作为同步的处理方式。

第 5 章 Chapter 5

定时器 Timer

定时 / 计划功能在移动开发领域使用较多，比如 Android 技术。定时计划任务功能在 Java 中主要使用的就是 Timer 对象，它在内部使用多线程的方式进行处理，所以它和线程技术还是有非常大的关联的。在本章节着重掌握如下技术点：

- 如何实现指定时间执行任务。
- 如何实现按指定周期执行任务。

这两点在后面的章节都有详细的技术介绍。

5.1 定时器 Timer 的使用

在 JDK 库中 Timer 类主要负责计划任务的功能，也就是在指定的时间开始执行某一个任务。Timer 类的方法列表如图 5-1 所示。

Timer 类的主要作用就是设置计划任务，但封装任务的类却是 TimerTask 类，类结构如图 5-2 所示。

执行计划任务的代码要放入 TimerTask 的子类中，因为 TimerTask 是一个抽象类。

在下面的章节中将介绍全部与计划任务有关的方法。

5.1.1 方法 schedule（TimerTask task, Date time）的测试

该方法的作用是在指定的日期执行一次某一任务。

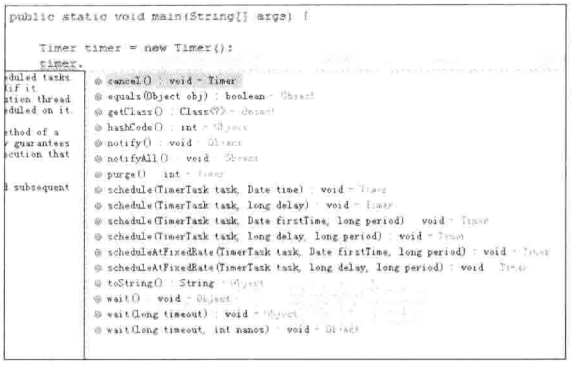


图 5-1 类 Timer 的方法列表

图 5-2 类 TimerTask 类相关的信息

1. 执行任务的时间晚于当前时间：在未来执行的效果

创建测试用的项目 timerTest1，创建类 Run1.java 代码如下：

```
package test;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run1 {
    private static Timer timer = new Timer();
    static public class MyTask extends TimerTask {
        @Override
        public void run() {
            System.out.println("运行了！时间为：" + new Date());
        }
    }
    public static void main(String[] args) {
        try {
            MyTask task = new MyTask();
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String dateString = "2014-10-12 11:55:00";
            Date dateRef = sdf.parse(dateString);
            System.out.println("字符串时间：" + dateRef.toLocaleString() + " 当前时间："
                    + new Date().toLocaleString());
            timer.schedule(task, dateRef);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行后的结果如图 5-3 所示。

任务虽然执行完了，但进程还未销毁，呈红色状态■，为什么会出现这样的情况？

在创建 Timer 对象时，JDK 源代码如下：

图 5-3　运行结果

```
public Timer() {
    this("Timer-" + serialNumber());
}
```

此构造方法调用的是如下构造方法：

```
public Timer(String name) {
    thread.setName(name);
    thread.start();
}
```

查看构造方法可以得知，创建一个 Timer 就是启动一个新的线程，这个新启动的线程并不是守护线程，它一直在运行。

下一步将新创建的 Timer 改成守护线程。新建 Java 类 Run1TimerIsDaemon.java，代码如下：

```
package test;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run1TimerIsDaemon {
    private static Timer timer = new Timer(true);
    static public class MyTask extends TimerTask {
        @Override
        public void run() {
            System.out.println("运行了! 时间为: " + new Date());
        }
    }
    public static void main(String[] args) {
        try {
            MyTask task = new MyTask();
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String dateString = "2014-10-12 12:05:00";
            Date dateRef = sdf.parse(dateString);
            System.out.println("字符串时间: " + dateRef.toLocaleString() + " 当前时间: "
                + new Date().toLocaleString());
            timer.schedule(task, dateRef);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行后的结果如图 5-4 所示。

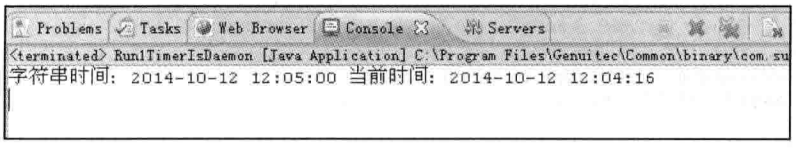


图 5-4 守护线程创建成功进程退出

程序运行后迅速结束当前的进程，并且 TimerTask 中的任务不再被运行，因为进程已经结束了。

2. 计划时间早于当前时间：提前运行的效果

如果执行任务的时间早于当前时间，则立即执行 task 任务。

示例代码如下：

```
public static void main(String[] args) {
    try {
        MyTask task = new MyTask();
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String dateString = "2014-10-12 09:08:00";
        Timer timer = new Timer();
        Date dateRef = sdf.parse(dateString);
        System.out.println("字符串时间: " + dateRef.toLocaleString() + " 当前时间: "
            + new Date().toLocaleString());
        timer.schedule(task, dateRef);
    } catch (ParseException e) {
        e.printStackTrace();
    }
}
```

运行结果如图 5-5 所示。

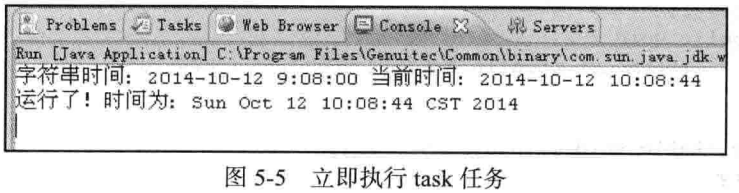


图 5-5 立即执行 task 任务

3. 多个 TimerTask 任务及延时的测试

Timer 中允许有多个 TimerTask 任务。

创建类 Run2.java，代码如下：

```
package test;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run2 {
```

```
    private static Timer timer = new Timer();
    static public class MyTask1 extends TimerTask {
        @Override
        public void run() {
            System.out.println("运行了！时间为：" + new Date());
        }
    }
    static public class MyTask2 extends TimerTask {
        @Override
        public void run() {
            System.out.println("运行了！时间为：" + new Date());
        }
    }
        public static void main(String[] args) {
        try {
            MyTask1 task1 = new MyTask1();
            MyTask2 task2 = new MyTask2();
            SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String dateString1 = "2014-10-12 10:39:00";
            String dateString2 = "2014-10-12 10:40:00";
            Date dateRef1 = sdf1.parse(dateString1);
            Date dateRef2 = sdf2.parse(dateString2);
            System.out.println("字符串 1 时间：" + dateRef1.toLocaleString() + " 当前时间："
                + new Date().toLocaleString());
            System.out.println("字符串 2 时间：" + dateRef2.toLocaleString() + " 当前时间："
                + new Date().toLocaleString());
            timer.schedule(task1, dateRef1);
            timer.schedule(task2, dateRef2);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 5-6 所示。

图 5-6　一个 Timer 中可以运行多个 TimerTask 任务

TimerTask 是以队列的方式一个一个被顺序执行的，所以执行的时间有可能和预期的时间不一致，因为前面的任务有可能消耗的时间较长，则后面的任务运行的时间也会被延迟。请看下面的示例。

创建测试用的 Java 类 Run2Later.java，代码如下：

```
package test;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run2Later {
    private static Timer timer = new Timer();
    static public class MyTask1 extends TimerTask {
        @Override
        public void run() {
            try {
                System.out.println("1 begin 运行了! 时间为: " + new Date());
                Thread.sleep(20000);
                System.out.println("1   end 运行了! 时间为: " + new Date());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    static public class MyTask2 extends TimerTask {
        @Override
        public void run() {
            System.out.println("2 begin 运行了! 时间为: " + new Date());
            System.out.println("运行了! 时间为: " + new Date());
            System.out.println("2   end 运行了! 时间为: " + new Date());
        }
    }
    public static void main(String[] args) {
        try {
            MyTask1 task1 = new MyTask1();
            MyTask2 task2 = new MyTask2();
            SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String dateString1 = "2014-10-12 11:33:00";
            String dateString2 = "2014-10-12 11:33:10";
            Date dateRef1 = sdf1.parse(dateString1);
            Date dateRef2 = sdf2.parse(dateString2);
            System.out.println("字符串1时间: " + dateRef1.toLocaleString() + " 当前时间: "
                    + new Date().toLocaleString());
            System.out.println("字符串2时间: " + dateRef2.toLocaleString() + " 当前时间: "
                    + new Date().toLocaleString());
            timer.schedule(task1, dateRef1);
            timer.schedule(task2, dateRef2);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 5-7 所示。

由于 task1 需要用时 20 秒执行完任务，task1 开始的时间是 11 ： 33 ： 00，那么将要影响

task2 的计划任务执行的时间，task2 以此时间为基准，向后延迟 20 秒，task2 在 11 ： 33 ： 20 执行任务。因为 Task 是被放入队列中的，得一个一个顺序运行。

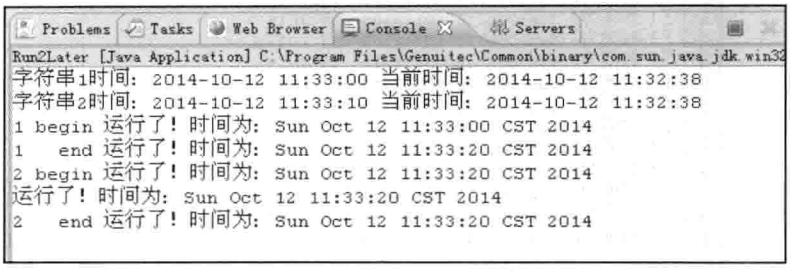


图 5-7　任务 2 的运行时间被延迟

5.1.2　方法 schedule(TimerTask task, Date firstTime, long period) 的测试

该方法的作用是在指定的日期之后，按指定的间隔周期性地无限循环地执行某一任务。

1. 计划时间晚于当前时间：在未来执行的效果

创建测试用的项目 timerTest2，创建类 Run.java 代码如下：

```
package test;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run {
    static public class MyTask extends TimerTask {
        @Override
        public void run() {
            System.out.println("运行了! 时间为: " + new Date());
        }
    }
    public static void main(String[] args) {
        try {
            MyTask task = new MyTask();
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String dateString = "2014-10-12 10:12:00";
            Timer timer = new Timer();
            Date dateRef = sdf.parse(dateString);
            System.out.println("字符串时间: " + dateRef.toLocaleString() + " 当前时间: "
                + new Date().toLocaleString());
            timer.schedule(task, dateRef, 4000);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行后的结果如图 5-8 所示。

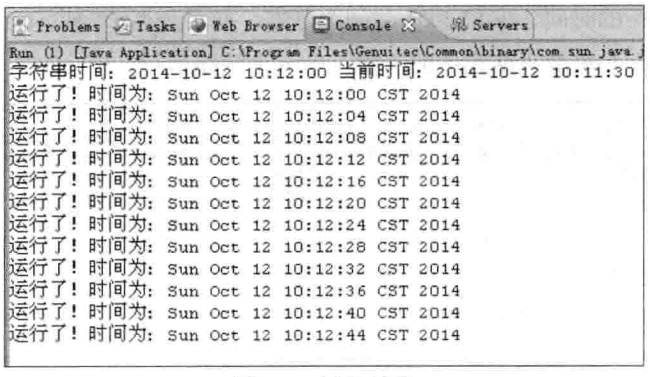


图 5-8 运行结果

从运行结果来看，每隔 4 秒运行一次 TimerTask 任务，并且是无限期地重复执行。

2. 计划时间早于当前时间：提前运行的效果

如果计划时间早于当前时间，则立即执行 task 任务。

示例代码如下：

```
public static void main(String[] args) {
    try {
        MyTask task = new MyTask();
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String dateString = "2014-10-12 09:12:00";
        Timer timer = new Timer();
        Date dateRef = sdf.parse(dateString);
        System.out.println("字符串时间: " + dateRef.toLocaleString() + " 当前时间: "
                + new Date().toLocaleString());
        timer.schedule(task, dateRef, 4000);
    } catch (ParseException e) {
        e.printStackTrace();
    }
}
```

运行结果如图 5-9 所示。

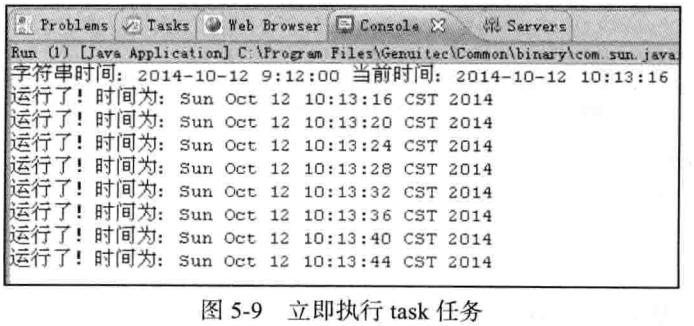


图 5-9 立即执行 task 任务

控制台打印的结果是，程序运行后立即执行 task 任务。

3. 任务执行时间被延时

创建类 Run2_1.java 代码如下：

```
package test;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run2_1 {
    static public class MyTaskA extends TimerTask {
        @Override
        public void run() {
            try {
                System.out.println("A 运行了！时间为：" + new Date());
                Thread.sleep(5000);
                System.out.println("A 结束了！时间为：" + new Date());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public static void main(String[] args) {
        try {
            MyTaskA taskA = new MyTaskA();
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String dateString = "2015-3-19 14:14:00";
            Timer timer = new Timer();
            Date dateRef = sdf.parse(dateString);
            System.out.println("字符串时间：" + dateRef.toLocaleString() + " 当前时间："
                    + new Date().toLocaleString());
            timer.schedule(taskA, dateRef, 4000);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 5-10 所示。

```
字符串时间：2015-3-19 14:14:00 当前时间：2015-3-19 14:13:26
A运行了！时间为：Thu Mar 19 14:14:00 CST 2015
A结束了！时间为：Thu Mar 19 14:14:05 CST 2015
A运行了！时间为：Thu Mar 19 14:14:05 CST 2015
A结束了！时间为：Thu Mar 19 14:14:10 CST 2015
A运行了！时间为：Thu Mar 19 14:14:10 CST 2015
A结束了！时间为：Thu Mar 19 14:14:15 CST 2015
A运行了！时间为：Thu Mar 19 14:14:15 CST 2015
A结束了！时间为：Thu Mar 19 14:14:20 CST 2015
A运行了！时间为：Thu Mar 19 14:14:20 CST 2015
A结束了！时间为：Thu Mar 19 14:14:25 CST 2015
```

图 5-10　任务被延时但还是一个一个顺序运行

4. TimerTask 类的 cancel() 方法

TimerTask 类中的 cancel() 方法的作用是将自身从任务队列中清除。

创建 Run2.java 文件，代码如下：

```
package test;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run2 {
    static public class MyTaskA extends TimerTask {
        @Override
        public void run() {
            System.out.println("A运行了！时间为：" + new Date());
            this.cancel();
        }
    }
    static public class MyTaskB extends TimerTask {
        @Override
        public void run() {
            System.out.println("B运行了！时间为：" + new Date());
        }
    }
    public static void main(String[] args) {
        try {
            MyTaskA taskA = new MyTaskA();
            MyTaskB taskB = new MyTaskB();
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String dateString = "2014-10-12 09:12:00";
            Timer timer = new Timer();
            Date dateRef = sdf.parse(dateString);
            System.out.println("字符串时间：" + dateRef.toLocaleString() + " 当前时间："
                    + new Date().toLocaleString());
            timer.schedule(taskA, dateRef, 4000);
            timer.schedule(taskB, dateRef, 4000);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行后的效果如图 5-11 所示。

图 5-11 TimerTaskA 仅运行一次后被清除了

TimerTask 类的 cancel() 方法是将自身从任务队列中被移除，其他任务不受影响。

5. Timer 类的 cancel() 方法

和 TimerTask 类中的 cancel() 方法清除自身不同，Timer 类中的 cancel() 方法的作用是将任务队列中的全部任务清空。

创建类 Run3.java，代码如下：

```
package test;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run3 {
    private static Timer timer = new Timer();
    static public class MyTaskA extends TimerTask {
        @Override
        public void run() {
            System.out.println("A运行了！时间为：" + new Date());
            timer.cancel();
        }
    }
    static public class MyTaskB extends TimerTask {
        @Override
        public void run() {
            System.out.println("B运行了！时间为：" + new Date());
        }
    }
    public static void main(String[] args) {
        try {
            MyTaskA taskA = new MyTaskA();
            MyTaskB taskB = new MyTaskB();
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String dateString = "2014-10-12 09:12:00";
            Date dateRef = sdf.parse(dateString);
            System.out.println("字符串时间：" + dateRef.toLocaleString() + " 当前时间："
                    + new Date().toLocaleString());
            timer.schedule(taskA, dateRef, 4000);
            timer.schedule(taskB, dateRef, 4000);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 5-12 所示。

全部任务都被清除，并且进程被销毁，按钮由红色变成灰色。

6. Timer 的 cancel() 方法注意事项

Timer 类中的 cancel() 方法有时并不一定会停止执行计划任务，而是正常执行。

创建 Run4.java 类，代码如下：

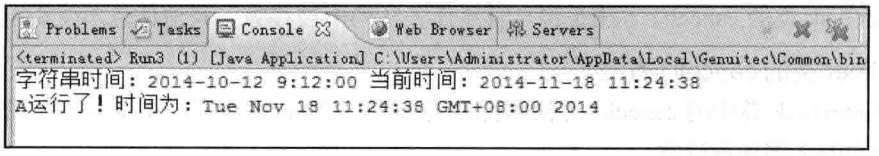


图 5-12 进程被清空

```
package test;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run4 {
    static int i = 0;
    static public class MyTask extends TimerTask {
        @Override
        public void run() {
            System.out.println("正常执行了" + i);
        }
    }
    public static void main(String[] args) {
        while (true) {
            try {
                i++;
                Timer timer = new Timer();
                MyTask task = new MyTask();
                SimpleDateFormat sdf = new SimpleDateFormat(
                        "yyyy-MM-dd HH:mm:ss");
                String dateString = "2014-10-12 09:08:00";
                Date dateRef = sdf.parse(dateString);
                timer.schedule(task, dateRef);
                timer.cancel();
            } catch (ParseException e) {
                e.printStackTrace();
            }
        }
    }
}
```

程序运行后的部分结果如图 5-13 所示。

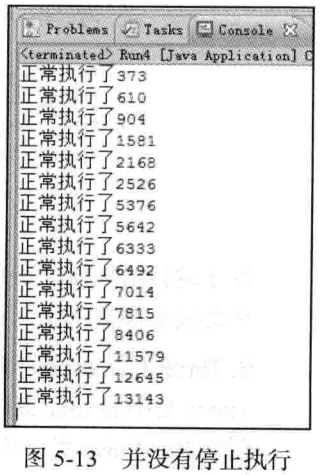


图 5-13 并没有停止执行

这是因为 Timer 类中的 cancel() 方法有时并没有争抢到 queue 锁，所以 TimerTask 类中的任务继续正常执行。

5.1.3 方法 schedule(TimerTask task, long delay) 的测试

该方法的作用是以执行 schedule（TimerTask task, long delay）方法当前的时间为参考时间，在此时间基础上延迟指定的毫秒数后执行一次 TimerTask 任务。

创建测试用的项目 timerTest3，创建类 Run.java 代码如下：

```
package test;
import java.text.ParseException;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run {
    static public class MyTask extends TimerTask {
        @Override
        public void run() {
            System.out.println("运行了！时间为：" + new Date());
        }
    }
    public static void main(String[] args) {
            MyTask task = new MyTask();
            Timer timer = new Timer();
            System.out.println("当前时间："+new Date().toLocaleString());
            timer.schedule(task, 7000);
    }
}
```

任务 task 被延迟 7 秒执行。程序运行后的结果如图 5-14 所示。

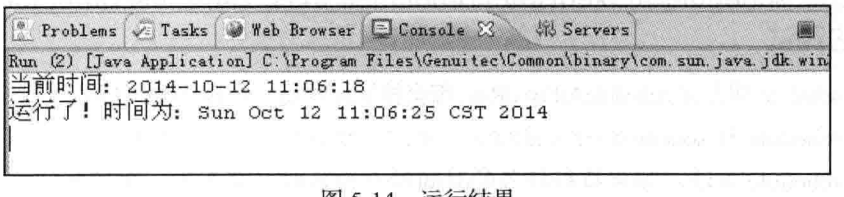


图 5-14　运行结果

5.1.4　方法 schedule(TimerTask task, long delay, long period) 的测试

该方法的作用是以执行 schedule（TimerTask task, long delay, long period）方法当前的时间为参考时间，在此时间基础上延迟指定的毫秒数，再以某一间隔时间无限次数地执行某一任务。

创建测试用的项目 timerTest4，创建类 Run.java 代码如下：

```
package test;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run {
    static public class MyTask extends TimerTask {
        @Override
        public void run() {
            System.out.println("运行了！时间为：" + new Date());
        }
    }
    public static void main(String[] args) {
            MyTask task = new MyTask();
```

```
        Timer timer = new Timer();
        System.out.println("当前时间: "+new Date().toLocaleString());
        timer.schedule(task, 3000,5000);
    }
}
```

程序运行后的结果如图 5-15 所示。

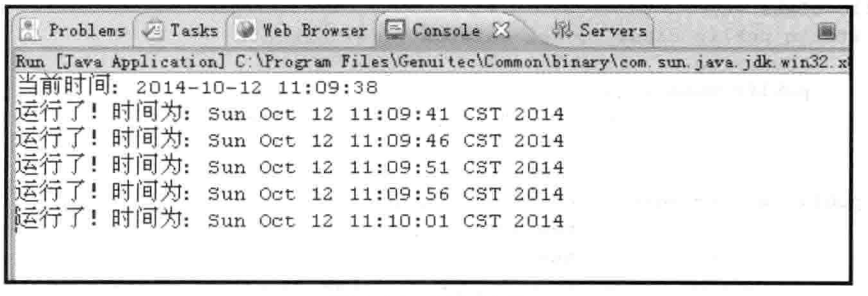


图 5-15 运行结果

凡是使用方法中带有 period 参数的，都是无限循环执行 TimerTask 中的任务。

5.1.5 方法 scheduleAtFixedRate(TimerTask task, Date firstTime, long period) 的测试

方法 schedule 和方法 scheduleAtFixedRate 都会按顺序执行，所以不要考虑非线程安全的情况。

方法 schedule 和 scheduleAtFixedRate 主要的区别只在于不延时的情况。

使用 schedule 方法：如果执行任务的时间没有被延时，那么下一次任务的执行时间参考的是上一次任务的“开始”时的时间来计算。

使用 scheduleAtFixedRate 方法：如果执行任务的时间没有被延时，那么下一次任务的执行时间参考的是上一次任务的“结束”时的时间来计算。

延时的情况则没有区别，也就是使用 schedule 或 scheduleAtFixedRate 方法都是如果执行任务的时间被延时，那么下一次任务的执行时间参考的是上一次任务“结束”时的时间来计算。

1. 测试 schedule 方法任务不延时

创建项目 timerTest5，创建 Java 类 Run1.java，代码如下：

```
package test.run;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run1 {
    private static Timer timer = new Timer();
    private static int runCount = 0;
    static public class MyTask1 extends TimerTask {
```

```
        @Override
        public void run() {
            try {
                System.out.println("1 begin 运行了! 时间为: " + new Date());
                Thread.sleep(1000);
                System.out.println("1   end 运行了! 时间为: " + new Date());
                runCount++;
                if (runCount == 5) {
                    timer.cancel();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public static void main(String[] args) {
        try {
            MyTask1 task1 = new MyTask1();
            SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String dateString1 = "2014-10-12 15:11:00";
            Date dateRef1 = sdf1.parse(dateString1);
            System.out.println("字符串1时间: " + dateRef1.toLocaleString() + " 当前时间: "
                    + new Date().toLocaleString());
            timer.schedule(task1, dateRef1, 3000);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行后结果如图5-16所示。

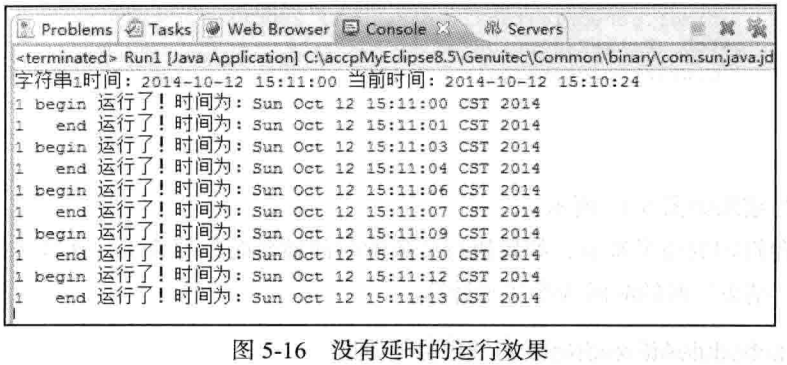


图5-16 没有延时的运行效果

控制台打印的结果证明，在不延时的情况下，如果执行任务的时间没有被延时，则下一次执行任务的时间是上一次任务的开始时间加上delay时间。

2. 测试schedule方法任务延时

创建Run2.java类，代码如下：

```
package test.run;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run2 {
    private static Timer timer = new Timer();
    private static int runCount = 0;
    static public class MyTask1 extends TimerTask {
        @Override
        public void run() {
            try {
                System.out.println("1 begin 运行了! 时间为: " + new Date());
                Thread.sleep(5000);
                System.out.println("1   end 运行了! 时间为: " + new Date());
                runCount++;
                if (runCount == 5) {
                    timer.cancel();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public static void main(String[] args) {
        try {
            MyTask1 task1 = new MyTask1();
            SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String dateString1 = "2014-10-12 15:16:00";
            Date dateRef1 = sdf1.parse(dateString1);
            System.out.println("字符串1时间: " + dateRef1.toLocaleString() + " 当前时间: "
                    + new Date().toLocaleString());
            timer.schedule(task1, dateRef1, 2000);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 5-17 所示。

从控制台打印的结果来看，如果执行任务的时间被延时，那么下一次任务的执行时间以上一次任务“结束”时的时间为参考来计算。

3. 测试 scheduleAtFixedRate 方法任务不延时

创建 Run3.java 文件，本示例使用方法 scheduleAtFixedRate 作为测试。

完整代码如下：

```
package test.run;
import java.text.ParseException;
import java.text.SimpleDateFormat;
```

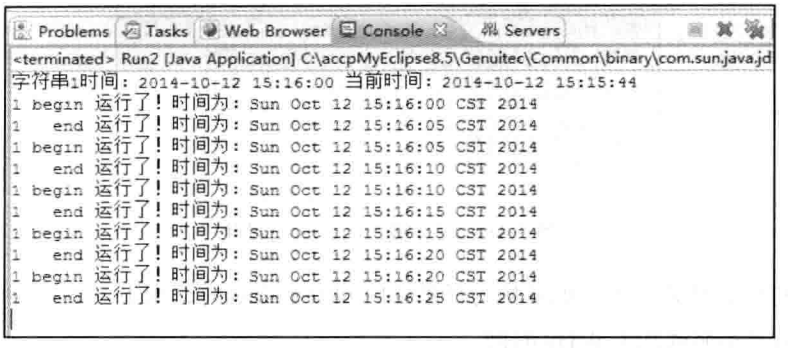


图 5-17　任务延时的效果

```
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run3 {
    private static Timer timer = new Timer();
    private static int runCount = 0;
    static public class MyTask1 extends TimerTask {
        @Override
        public void run() {
            try {
                System.out.println("1 begin 运行了！时间为: " + new Date());
                Thread.sleep(2000);
                System.out.println("1   end 运行了！时间为: " + new Date());
                runCount++;
                if (runCount == 5) {
                    timer.cancel();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public static void main(String[] args) {
        try {
            MyTask1 task1 = new MyTask1();
            SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String dateString1 = "2014-10-12 15:28:00";
            Date dateRef1 = sdf1.parse(dateString1);
            System.out.println("字符串 1 时间: " + dateRef1.toLocaleString() + " 当前时间: "
                    + new Date().toLocaleString());
            timer.scheduleAtFixedRate(task1, dateRef1, 3000);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行后的结果如图 5-18 所示。

```
<terminated> Run3 [Java Application] C:\Program Files\Genuitec\Common\binary\com.
字符串1时间: 2014-10-12 15:28:00 当前时间: 2015-4-23 9:54:46
1 begin 运行了! 时间为: Thu Apr 23 09:54:46 CST 2015
1   end 运行了! 时间为: Thu Apr 23 09:54:48 CST 2015
1 begin 运行了! 时间为: Thu Apr 23 09:54:48 CST 2015
1   end 运行了! 时间为: Thu Apr 23 09:54:50 CST 2015
1 begin 运行了! 时间为: Thu Apr 23 09:54:50 CST 2015
1   end 运行了! 时间为: Thu Apr 23 09:54:52 CST 2015
1 begin 运行了! 时间为: Thu Apr 23 09:54:52 CST 2015
```

图 5-18 没有被延时的运行结果

控制台打印的结果证明，如果执行任务的时间没有被延时，则下一次执行任务的时间是上一次任务的开始时间加上 delay 时间。

4. 测试 scheduleAtFixedRate 方法任务延时

创建 Run4.java 文件，代码如下：

```
package test.run;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run4 {
    private static Timer timer = new Timer();
    private static int runCount = 0;
    static public class MyTask1 extends TimerTask {
        @Override
        public void run() {
            try {
                System.out.println("1 begin 运行了! 时间为: " + new Date());
                Thread.sleep(5000);
                System.out.println("1   end 运行了! 时间为: " + new Date());
                runCount++;
                if (runCount == 5) {
                    timer.cancel();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public static void main(String[] args) {
        try {
            MyTask1 task1 = new MyTask1();
            SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String dateString1 = "2014-10-12 15:33:00";
            Date dateRef1 = sdf1.parse(dateString1);
            System.out.println("字符串1时间: " + dateRef1.toLocaleString() + " 当前时间: "
                    + new Date().toLocaleString());
            timer.scheduleAtFixedRate(task1, dateRef1, 2000);
```

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

程序运行结果如图 5-19 所示。

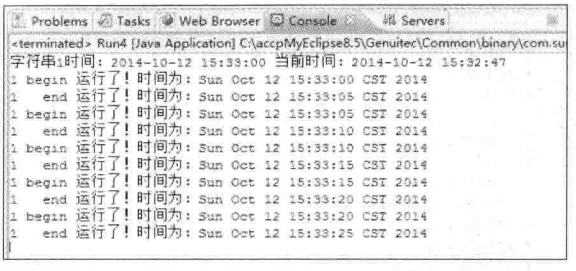


图 5-19　任务延时的运行结果

从控制台打印的结果来看，如果执行任务的时间被延时，那么下一次任务的执行时间以上一次任务“结束”时的时间为参考来计算。

5. 验证 schedule 方法不具有追赶执行性

创建 Java 类 Run5.java，代码如下：

```
package test.run;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run5 {
    private static Timer timer = new Timer();
    static public class MyTask1 extends TimerTask {
        @Override
        public void run() {
                System.out.println("1 begin 运行了！时间为：" + new Date());
                System.out.println("1   end 运行了！时间为：" + new Date());
        }
    }
    public static void main(String[] args) {
        try {
            MyTask1 task1 = new MyTask1();
            SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String dateString1 = "2014-10-12 15:37:00";
            Date dateRef1 = sdf1.parse(dateString1);
            System.out.println("字符串 1 时间：" + dateRef1.toLocaleString() + " 当前时间："
                    + new Date().toLocaleString());
            timer.schedule(task1, dateRef1, 5000);
        } catch (ParseException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 5-20 所示。

图 5-20 不追赶的情况

时间“2014-10-12 15:37:00”到“2014-10-12 15:43:53”之间的时间所对应的 Task 任务被取消了，不执行了。这就是 Task 任务不追赶的情况。

6. 验证 scheduleAtFixedRate 方法具有追赶执行性

创建 Java 类 Run6.java，代码如下：

```
package test.run;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Run6 {
    private static Timer timer = new Timer();
    static public class MyTask1 extends TimerTask {
        @Override
        public void run() {
                System.out.println("1 begin 运行了！时间为：" + new Date());
                System.out.println("1   end 运行了！时间为：" + new Date());
        }
    }
    public static void main(String[] args) {
        try {
            MyTask1 task1 = new MyTask1();
            SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String dateString1 = "2014-10-12 15:45:50";
            Date dateRef1 = sdf1.parse(dateString1);
            System.out.println("字符串1时间：" + dateRef1.toLocaleString() + " 当前时间："
                    + new Date().toLocaleString());
```

```
                timer.scheduleAtFixedRate(task1, dateRef1, 5000);
            } catch (ParseException e) {
                e.printStackTrace();
            }
        }
    }
```

程序运行结果如图 5-21 所示。

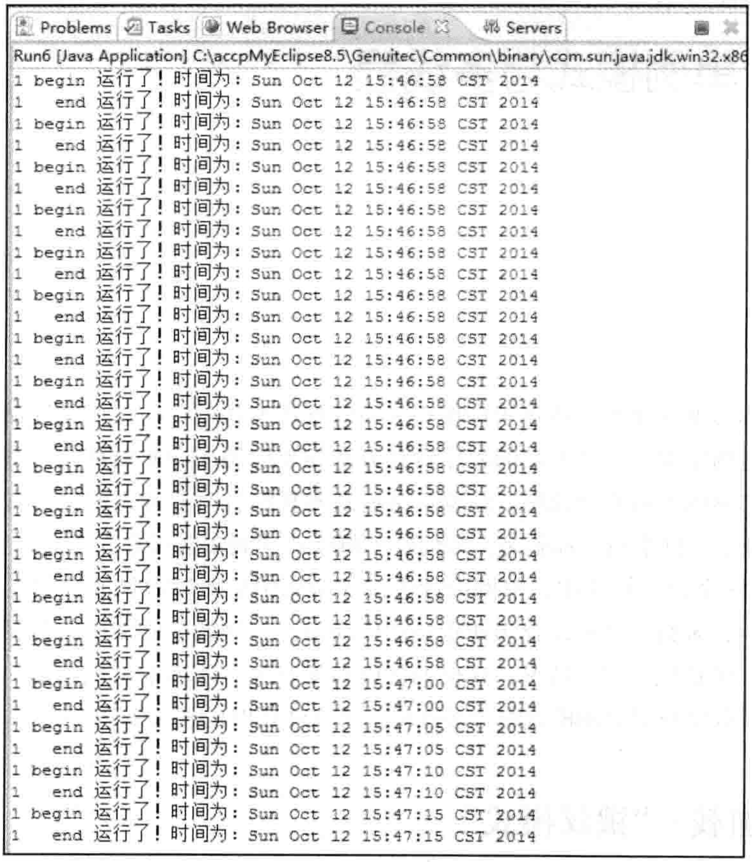


图 5-21　追赶的情况

两个时间段内所对应的 Task 任务被“补充性”执行了，这就是 Task 任务追赶执行的特性。

5.2　本章总结

通过本章的学习，应该掌握如何在 Java 中使用定时任务的功能，并且可以对这些定时任务使用指定的 API 进行处理。这些示例代码完全可以应用在 Android 技术中，实现类似于轮询动画等常见的功能。

Chapter 6 第 6 章

单例模式与多线程

本章的知识点非常重要，通过单例模式与多线程技术相结合，在这个过程中能发现很多以前从未考虑过的情况，一些不良的程序设计方法如果应用在商业项目中，将会遇到非常大的麻烦。本章的案例也将充分说明，线程与某些技术相结合时要考虑的事情有很多。在学习本章时只需要考虑一件事情，那就是：如何使单例模式遇到多线程是安全的、正确的。

在标准的 23 个设计模式中，单例设计模式在应用中是比较常见的。但在常规的该模式教学资料介绍中，多数并没有结合多线程技术作为参考，这就造成在使用多线程技术的单例模式时会出现一些意想不到的情况，这样的代码如果在生产环境中出现异常，有可能造成灾难性的后果。本章将介绍单例模式结合多线程技术在使用时的相关知识。

6.1 立即加载 / “饿汉模式”

什么是立即加载？立即加载就是使用类的时候已经将对象创建完毕，常见的实现办法就是直接 new 实例化。而立即加载从中文的语境来看，有“着急”、“急迫”的含义，所以也称为“饿汉模式”。

立即加载 / “饿汉模式”是在调用方法前，实例已经被创建了，来看一下实现代码。

创建测试用的项目，名称为 singleton_0，创建类 MyObject.java 代码如下：

```
package test;
public class MyObject {
    //立即加载方式 == 饿汉模式
    private static MyObject myObject = new MyObject();
```

```
    private MyObject() {
    }
    public static MyObject getInstance() {
        //此代码版本为立即加载
        //此版本代码的缺点是不能有其他实例变量
        //因为 getInstance() 方法没有同步
        //所以有可能出现非线程安全问题
        return myObject;
    }
}
```

创建线程类 MyThread.java 代码如下：

```
package extthread;
import test.MyObject;
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println(MyObject.getInstance().hashCode());
    }
}
```

创建运行类 Run.java 代码如下：

```
package test.run;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

程序运行后的结果如图 6-1 所示。

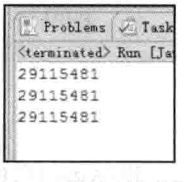


图 6-1　“饿汉模式”的运行结果

控制台打印的 hashCode 是同一个值，说明对象是同一个，也就实现了立即加载型单例设计模式。

6.2　延迟加载 / “懒汉模式”

什么是延迟加载？延迟加载就是在调用 get() 方法时实例才被创建，常见的实现办法就是在 get() 方法中进行 new 实例化。而延迟加载从中文的语境来看，是“缓慢”、“不急迫”的含义，所以也称为“懒汉模式”。

1. 延迟加载 / “懒汉模式”解析

延迟加载 / “懒汉模式”是在调用方法时实例才被创建。一起来看一下实现代码。

创建测试用的项目 singleton_1，创建类 MyObject.java 代码如下：

```
package test;
public class MyObject {
    private static MyObject myObject;
    private MyObject() {
    }
    public static MyObject getInstance() {
        //延迟加载
        if (myObject != null) {
        } else {
            myObject = new MyObject();
        }
        return myObject;
    }
}
```

创建线程类 MyThread.java 代码如下：

```
package extthread;
import test.MyObject;
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println(MyObject.getInstance().hashCode());
    }
}
```

创建运行类 Run.java 代码如下：

```
package test.run;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

程序运行后的效果如图 6-2 所示。

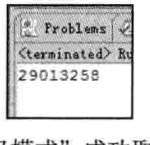


图 6-2 “懒汉模式”成功取出一个实例

此实验虽然取得一个对象的实例，但如果是在多线程的环境中，就会出现取出多个实例的情况，与单例模式的初衷是相背离的。

2. 延迟加载 / “懒汉模式”的缺点

前面两个实验虽然使用“立即加载”和“延迟加载”实现了单例设计模式，但在多线程

的环境中，前面“延迟加载”示例中的代码完全就是错误的，根本不能实现保持单例的状态。来看一下如何在多线程环境中结合“错误的单例模式”创建出“多例”。

创建测试用的项目，名称为 singleton_2，创建类 MyObject.java 代码如下：

```
package test;
public class MyObject {
    private static MyObject myObject;
    private MyObject() {
    }
    public static MyObject getInstance() {
        try {
            if (myObject != null) {
            } else {
                //模拟在创建对象之前做一些准备性的工作
                Thread.sleep(3000);
                myObject = new MyObject();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return myObject;
    }
}
```

创建线程类 MyThread.java 代码如下：

```
package extthread;
import test.MyObject;
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println(MyObject.getInstance().hashCode());
    }
}
```

创建运行类 Run.java 代码如下：

```
package test.run;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

程序运行后的效果如图 6-3 所示。

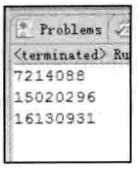


图 6-3 非单例设计模式

控制台打印出了 3 种 hashCode，说明创建出了 3 个对象，并不是单例的，这就是“错误的单例模式”。如何解决呢？先看一下解决方案。

3. 延迟加载 /“懒汉模式”的解决方案

（1）声明 synchronized 关键字

既然多个线程可以同时进入 getInstance() 方法，那么只需要对 getInstance() 方法声明 synchronized 关键字即可。

创建测试用的项目 singleton_2_1，创建类 MyObject.java 代码如下：

```
package test;
public class MyObject {
    private static MyObject myObject;
    private MyObject() {
    }
    //设置同步方法效率太低了
    //整个方法被上锁
    synchronized public static MyObject getInstance() {
        try {
            if (myObject != null) {
            } else {
                //模拟在创建对象之前做一些准备性的工作
                Thread.sleep(3000);
                myObject = new MyObject();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return myObject;
    }
}
```

创建线程类 MyThread.java 代码如下：

```
package extthread;
import test.MyObject;
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println(MyObject.getInstance().hashCode());
    }
}
```

创建运行类 Run.java 代码如下：

```
package test.run;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

程序运行后的结果如图 6-4 所示。

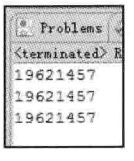


图 6-4　运行结果

此方法加入同步 synchronized 关键字得到相同实例的对象，但此种方法的运行效率非常低下，是同步运行的，下一个线程想要取得对象，则必须等上一个线程释放锁之后，才可以继续执行。

（2）尝试同步代码块

同步方法是对方法的整体进行持锁，这对运行效率来讲是不利的。改成同步代码块能解决吗？

创建测试用的项目 singleton_2_2，创建类 MyObject.java 代码如下：

```
package test;
public class MyObject {
    private static MyObject myObject;
    private MyObject() {
    }
    public static MyObject getInstance() {
        try {
            //此种写法等同于：
            //synchronized public static MyObject getInstance()
            //的写法，效率一样很低，全部代码被上锁
            synchronized (MyObject.class) {
                if (myObject != null) {
                } else {
                    //模拟在创建对象之前做一些准备性的工作
                    Thread.sleep(3000);
                    myObject = new MyObject();
                }
            }
```

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

创建线程类 MyThread.java 代码如下：

```
package extthread;
import test.MyObject;
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println(MyObject.getInstance().hashCode());
    }
}
```

创建运行类 Run.java 代码如下：

```
package test.run;
import test.MyObject;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();
        t1.start();
        t2.start();
        t3.start();
        // 此版本代码虽然是正确的
        // 但 public static MyObject getInstance() 方法
        // 中的全部代码都是同步的了，这样做也会降低运行效率
    }
}
```

程序运行后的结果如图 6-5 所示。

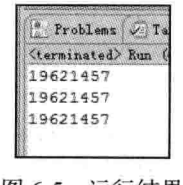


图 6-5 运行结果

此方法加入同步 synchronized 语句块得到相同实例的对象，但此种方法的运行效率也是非常低的，和 synchronized 同步方法一样是同步运行的。继续更改代码尝试解决这个缺点。

（3）针对某些重要的代码进行单独的同步

同步代码块可以针对某些重要的代码进行单独的同步，而其他的代码则不需要同步。这

样在运行时，效率完全可以得到大幅提升。

创建测试用的项目，名称为 singleton_3，创建类 MyObject.java 代码如下：

```
package test;
public class MyObject {
    private static MyObject myObject;
    private MyObject() {
    }
    public static MyObject getInstance() {
        try {
            if (myObject != null) {
            } else {
                //模拟在创建对象之前做一些准备性的工作
                Thread.sleep(3000);
                //使用 synchronized (MyObject.class)
                //虽然部分代码被上锁
                //但还是有非线程安全问题
                synchronized (MyObject.class) {
                    myObject = new MyObject();
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return myObject;
    }
}
```

创建线程类 MyThread.java 代码如下：

```
package extthread;
import test.MyObject;
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println(MyObject.getInstance().hashCode());
    }
}
```

创建运行类 Run.java 代码如下：

```
package test.run;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();
        t1.start();
```

```
        t2.start();
        t3.start();
    }
}
```

程序运行后的结果如图 6-6 所示。

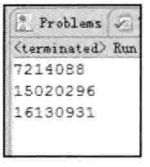


图 6-6 运行结果

此方法使同步 synchronized 语句块，只对实例化对象的关键代码进行同步，从语句的结构上来讲，运行的效率的确得到了提升。但如果是遇到多线程的情况下还是无法解决得到同一个实例对象的结果。到底如何解决“懒汉模式”遇到多线程的情况呢?

（4）使用 DCL 双检查锁机制

在最后的步骤中，使用的是 DCL 双检查锁机制来实现多线程环境中的延迟加载单例设计模式。

创建测试用的项目 singleton_5，创建类 MyObject.java 代码如下：

```
package test;
public class MyObject {
    private volatile static MyObject myObject;
    private MyObject() {
    }
    //使用双检测机制来解决问题，既保证了不需要同步代码的异步执行性
    //又保证了单例的效果


    public static MyObject getInstance() {
        try {
            if (myObject != null) {
            } else {
                //模拟在创建对象之前做一些准备性的工作
                Thread.sleep(3000);
                synchronized (MyObject.class) {
                    if (myObject == null) {
                        myObject = new MyObject();
                    }
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return myObject;
    }
```

```
    //此版本的代码称为双重检查 Double-Check Locking
}
```

创建线程类 MyThread.java 代码如下：

```
package extthread;
import test.MyObject;
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println(MyObject.getInstance().hashCode());
    }
}
```

创建运行类 Run.java 代码如下：

```
package test.run;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

程序运行后的结果如图 6-7 所示。

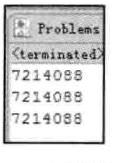


图 6-7　运行结果

使用双重检查锁功能，成功地解决了“懒汉模式”遇到多线程的问题。DCL 也是大多数多线程结合单例模式使用的解决方案。

6.3　使用静态内置类实现单例模式

DCL 可以解决多线程单例模式的非线程安全问题。当然，使用其他的办法也能达到同样的效果。

创建测试用的项目 singleton_7，创建类 MyObject.java 代码如下：

```
package test;
```

```
public class MyObject {
    //内部类方式
    private static class MyObjectHandler {
        private static MyObject myObject = new MyObject();
    }
    private MyObject() {
    }
    public static MyObject getInstance() {
        return MyObjectHandler.myObject;
    }
}
```

创建线程类 MyThread.java 代码如下：

```
package extthread;
import test.MyObject;
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println(MyObject.getInstance().hashCode());
    }
}
```

创建运行类 Run.java 代码如下：

```
package test.run;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

程序运行后的结果如图 6-8 所示。

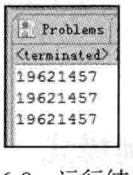


图 6-8 运行结果

6.4 序列化与反序列化的单例模式实现

静态内置类可以达到线程安全问题，但如果遇到序列化对象时，使用默认的方式运行得

到的结果还是多例的。

创建测试用的项目 singleton_7_1，创建类 MyObject.java 代码如下：

```
package test;
import java.io.ObjectStreamException;
import java.io.Serializable;
public class MyObject implements Serializable {
    private static final long serialVersionUID = 888L;
    //内部类方式
    private static class MyObjectHandler {
        private static final MyObject myObject = new MyObject();
    }
    private MyObject() {
    }
    public static MyObject getInstance() {
        return MyObjectHandler.myObject;
    }
    //protected Object readResolve() throws ObjectStreamException {
    //System.out.println("调用了 readResolve 方法！ ");
    //return MyObjectHandler.myObject;
    //}
}
```

创建业务类 SaveAndRead.java 代码如下：

```
package test.run;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import test.MyObject;
public class SaveAndRead {
    public static void main(String[] args) {
        try {
            MyObject myObject = MyObject.getInstance();
            FileOutputStream fosRef = new FileOutputStream(new File(
                    "myObjectFile.txt"));
            ObjectOutputStream oosRef = new ObjectOutputStream(fosRef);
            oosRef.writeObject(myObject);
            oosRef.close();
            fosRef.close();
            System.out.println(myObject.hashCode());
        } catch (FileNotFoundException e) {
            //TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            //TODO Auto-generated catch block
```

```
            e.printStackTrace();
        }
        try {
            FileInputStream fisRef = new FileInputStream(new File(
                    "myObjectFile.txt"));
            ObjectInputStream iosRef = new ObjectInputStream(fisRef);
            MyObject myObject = (MyObject) iosRef.readObject();
            iosRef.close();
            fisRef.close();
            System.out.println(myObject.hashCode());
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行后的效果如图 6-9 所示。

解决办法就是在反序列化中使用 readResolve() 方法。

去掉如下代码的注释：

```
protected Object readResolve() throws ObjectStreamException {
    System.out.println("调用了 readResolve 方法！ ");
    return MyObjectHandler.myObject;
}
```

程序运行后的结果如图 6-10 所示。

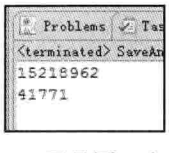


图 6-9 不是同一个对象

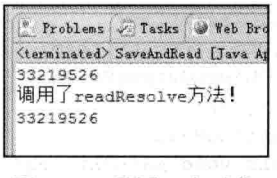


图 6-10 是同一个对象

6.5 使用 static 代码块实现单例模式

静态代码块中的代码在使用类的时候就已经执行了，所以可以应用静态代码块的这个特性来实现单例设计模式。

创建测试用的项目 singleton_8，创建类 MyObject.java 代码如下：

```
package test;
public class MyObject {
    private static MyObject instance = null;
```

```
    private MyObject() {
    }
    static {
        instance = new MyObject();
    }
    public static MyObject getInstance() {
        return instance;
    }
}
```

创建线程类 MyThread.java 代码如下：

```
package extthread;
import test.MyObject;
public class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(MyObject.getInstance().hashCode());
        }
    }
}
```

创建运行类 Run.java 代码如下：

```
package test.run;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

程序运行后的结果如图 6-11 所示。

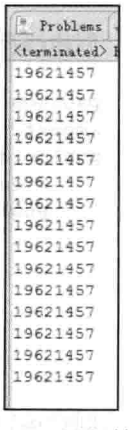


图 6-11　运行结果

6.6　使用 enum 枚举数据类型实现单例模式

枚举 enum 和静态代码块的特性相似，在使用枚举类时，构造方法会被自动调用，也可以应用其这个特性实现单例设计模式。

创建测试用的项目 singleton_9，创建类 MyObject.java 代码如下：

```
package test;
import java.sql.Connection;
import java.sql.DriverManager;
```

```
import java.sql.SQLException;
public enum MyObject {
    connectionFactory;
    private Connection connection;
    private MyObject() {
        try {
            System.out.println("调用了MyObject的构造");
            String url = "jdbc:sqlserver://localhost:1079;databaseName=ghydb";
            String username = "sa";
            String password = "";
            String driverName = "com.microsoft.sqlserver.jdbc.SQLServerDriver";
            Class.forName(driverName);
            connection = DriverManager.getConnection(url, username, password);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    public Connection getConnection() {
        return connection;
    }
}
```

创建线程类 MyThread.java 代码如下：

```
package extthread;
import test.MyObject;
public class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(MyObject.connectionFactory.getConnection()
                    .hashCode());
        }
    }
}
```

创建运行类 Run.java 代码如下：

```
package test.run;
import test.MyObject;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

程序运行后的结果如图 6-12 所示。

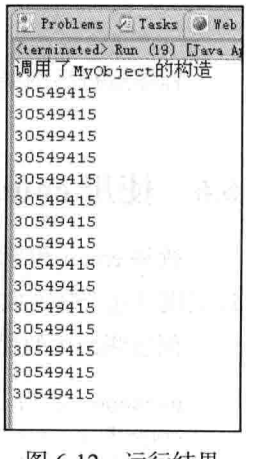


图 6-12 运行结果

6.7　完善使用 enum 枚举实现单例模式

前面一节将枚举类进行曝露，违反了“职责单一原则”，在项目 singleton_10 中进行完善。

将项目 singleton_9 中的所有源代码复制到 singleton_10 项目中，更改类 MyObject.java 代码如下：

```
package test;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class MyObject {
    public enum MyEnumSingleton {
        connectionFactory;
        private Connection connection;
        private MyEnumSingleton() {
            try {
                System.out.println("创建 MyObject 对象 ");
                String url = "jdbc:sqlserver://localhost:1079;databaseName=y2";
                String username = "sa";
                String password = "";
                String driverName = "com.microsoft.sqlserver.jdbc.SQLServerDriver";
                Class.forName(driverName);
                connection = DriverManager.getConnection(url, username,
                        password);
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        public Connection getConnection() {
            return connection;
        }
    }
    public static Connection getConnection() {
        return MyEnumSingleton.connectionFactory.getConnection();
    }
}
```

更改 MyThread.java 类代码如下：

```
package extthread;
import test.MyObject;
public class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
```

```
                System.out.println(MyObject.getConnection().hashCode());
            }
        }
    }
```

程序运行的结果如图 6-13 所示。

图 6-13　成功实现单例模式

6.8　本章总结

本章使用若干案例来阐述单例模式与多线程结合时遇到的情况与解决方法。本章也复习了不同单例模式的使用，使得以后再遇到单例模式时，就能从容面对多线程环境的情况了。

第 7 章 Chapter 7

拾遗增补

本章是本书的最后一章，在本章中将对前面几章遗漏的知识点进行补充，丰富多线程案例的完整性。在开发此类应用中，这些案例能起到优化性能的作用，至少在遇到某些情况时会回想起这些案例的初衷与解决办法。

本章应该掌握如下知识点：

- ❑ 线程组的使用。
- ❑ 如何切换线程状态。
- ❑ SimpleDataFormat 类与多线程的解决办法。
- ❑ 如何处理线程的异常。

7.1 线程的状态

线程对象在不同的运行时期有不同的状态，状态信息就存在于 State 枚举类中，如图 7-1 所示。每个枚举类型的解释如图 7-2 所示。

图 7-1 枚举类型 State 信息

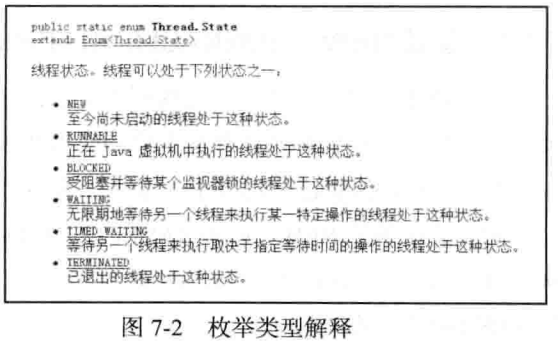


图 7-2 枚举类型解释

调用与线程有关的方法是造成线程状态改变的主要原因，其因果关系如图 7-3 所示。

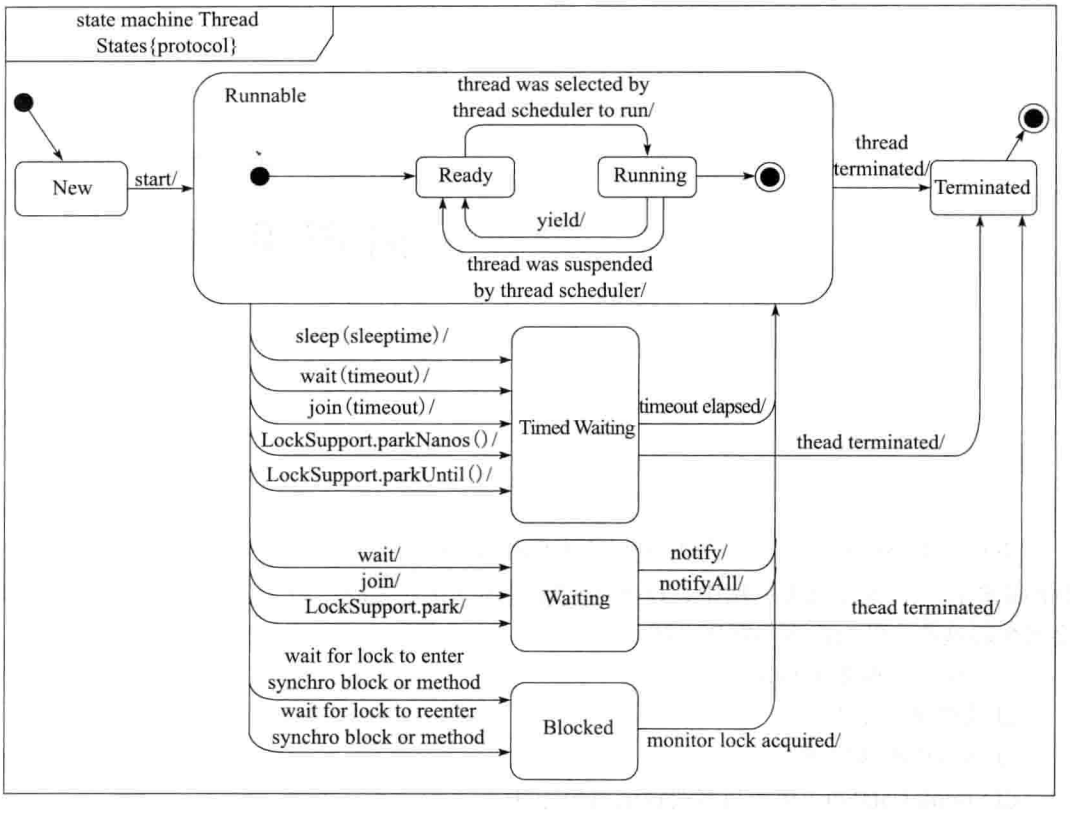


图 7-3 方法与状态关系示意图

从图 7-3 中可以得知，在调用与线程有关的方法后，会进入不同的线程状态，这些状态之间某些是可双向切换的，比如 WAITING 和 RUNNING 状态之间可以循环地进行切换；而有些是单向切换的，比如线程销毁后并不能自动进入 RUNNING 状态。

在下面的小节中将对这 6 种线程状态用程序代码的方式进行验证。

7.1.1 验证 NEW、RUNNABLE 和 TERMINATED

下面使用代码的方式验证线程所有的状态值，了解线程的状态有助于程序员监控线程对象所处的情况，比如哪些线程从未启动，哪些线程正在执行，哪些线程正在阻塞，哪些线程正在等待，哪些线程已经销毁了，等等。这些是与线程生命周期相关的信息。

首先验证的是 NEW、RUNNABLE 及 TERMINATED 状态，NEW 状态是线程实例化后还从未执行 start() 方法时的状态，而 RUNNABLE 状态是线程进入运行的状态，TERMINATED 是线程被销毁时的状态。

创建名称为 stateTest1 的项目，类 MyThread.java 代码如下：

```
package extthread;
public class MyThread extends Thread {
    public MyThread() {
        System.out.println(" 构造方法中的状态: " + Thread.currentThread().getState());
    }
    @Override
    public void run() {
        System.out.println("run 方法中的状态: " + Thread.currentThread().getState());
    }
}
```

类 Run.java 代码如下：

```
package test;
import extthread.MyThread;
public class Run {
    // NEW,
    // RUNNABLE,
    // TERMINATED,
    // BLOCKED,
    // WAITING,
    // TIMED_WAITING,
    public static void main(String[] args) {
        try {
            MyThread t = new MyThread();
            System.out.println("main 方法中的状态 1: " + t.getState());
            Thread.sleep(1000);
            t.start();
            Thread.sleep(1000);
            System.out.println("main 方法中的状态 2: " + t.getState());
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

程序运行结果如图 7-4 所示。

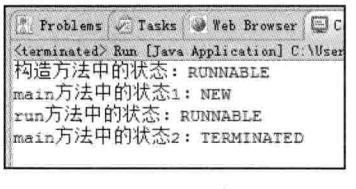


图 7-4 运行结果

注意 构造方法中打印出的日志其实是显示 main 主线程的状态为 RUNNABLE。

7.1.2 验证 TIMED_WAITING

线程状态 TIMED_WAITING 代表线程执行了 Thread.sleep() 方法，呈等待状态，等待时间到达，继续向下运行。

创建名称为 stateTest2 的项目，类 MyThread.java 代码如下：

```
package extthread;
public class MyThread extends Thread {
    @Override
    public void run() {
        try {
            System.out.println("begin sleep");
            Thread.sleep(10000);
            System.out.println("  end sleep");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

类 Run.java 代码如下：

```
package test;
import extthread.MyThread;
public class Run {
    //NEW,
    //RUNNABLE,
    //TERMINATED,
    //BLOCKED,
    //WAITING,
    //TIMED_WAITING,
    public static void main(String[] args) {
        try {
            MyThread t = new MyThread();
            t.start();
            Thread.sleep(1000);
            System.out.println("main 方法中的状态: " + t.getState());
        } catch (InterruptedException e) {
            //TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

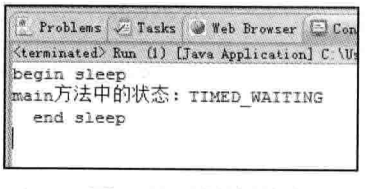


图 7-5 运行结果

程序运行结果如图 7-5 所示。

执行 sleep() 方法后线程的状态枚举值就是 TIMED_WAITING。

7.1.3 验证 BLOCKED

BLOCKED 状态出现在某一个线程在等待锁的时候。

创建名称为 stateTest3 的项目，创建业务对象 MyService.java，代码如下：

```
package service;
public class MyService {
    synchronized static public void serviceMethod() {
        try {
```

```
            System.out.println(Thread.currentThread().getName() + "进入了业务方法!");
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

类 MyThread1.java 代码如下：

```
package extthread;
import service.MyService;
public class MyThread1 extends Thread {
    @Override
    public void run() {
        MyService.serviceMethod();
    }
}
```

类 MyThread2.java 代码如下：

```
package extthread;
import service.MyService;
public class MyThread2 extends Thread {
    @Override
    public void run() {
        MyService.serviceMethod();
    }
}
```

类 Run.java 代码如下：

```
package test;
import extthread.MyThread1;
import extthread.MyThread2;
public class Run {
    //NEW,
    //RUNNABLE,
    //TERMINATED,
    //BLOCKED,
    //WAITING,
    //TIMED_WAITING,
    public static void main(String[] args) {
        MyThread1 t1 = new MyThread1();
        t1.setName("a");
        t1.start();
        MyThread2 t2 = new MyThread2();
        t2.setName("b");
        t2.start();
        System.out.println("main方法中的t2状态: " + t2.getState());
    }
}
```

程序运行结果如图 7-6 所示。

从控制台打印的结果来看，t2 线程一直在等待 t1 释放锁，所以 t2 当时的状态就是 BLOCKED。

图 7-6 运行结果

7.1.4 验证 WAITING

状态 WAITING 是线程执行了 Object.wait() 方法后所处的状态。

创建名称为 stateTest4 的项目，类 Lock.java 代码如下：

```
package service;
public class Lock {
    public static final Byte lock = new Byte("0");
}
```

类 MyThread.java 代码如下：

```
package extthread;
import service.Lock;
public class MyThread extends Thread {
    @Override
    public void run() {
        try {
            synchronized (Lock.lock) {
                Lock.lock.wait();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

类 Run.java 代码如下：

```
package test;
import extthread.MyThread;
public class Run {
    //NEW,
    //RUNNABLE,
    //TERMINATED,
    //BLOCKED,
    //WAITING,
    //TIMED_WAITING,
    public static void main(String[] args) {
        try {
            MyThread t = new MyThread();
            t.start();
            Thread.sleep(1000);
```

```
            System.out.println("main 方法中的 t 状态: " + t.getState());
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

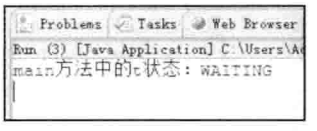


图 7-7 运行结果

程序运行结果如图 7-7 所示。

执行 wait() 方法后线程的状态枚举值就是 WAITING。

7.2 线程组

可以把线程归属到某一个线程组中，线程组中可以有线程对象，也可以有线程组，组中还可以有线程。这样的组织结构有些类似于树的形式，如图 7-8 所示。

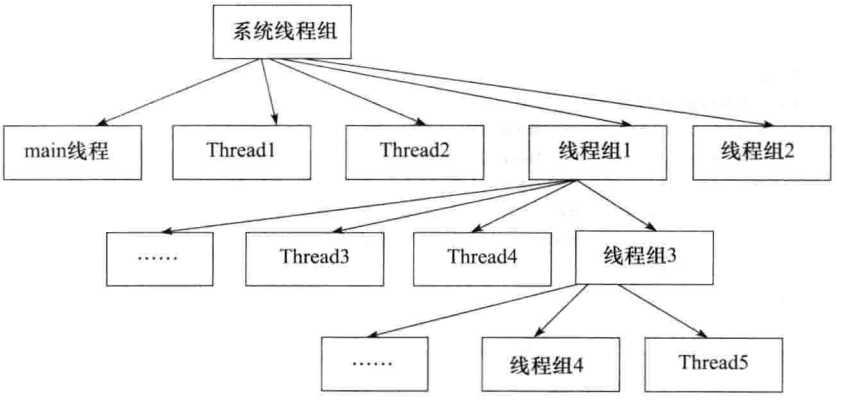


图 7-8 线程关系树结构

线程组的作用是，可以批量的管理线程或线程组对象，有效地对线程或线程组对象进行组织。

7.2.1 线程对象关联线程组：1 级关联

所谓的 1 级关联就是父对象中有子对象，但并不创建子孙对象。这种情况经常出现在开发中，比如创建一些线程时，为了有效地对这些线程进行组织管理，通常的情况下是创建一个线程组，然后再将部分线程归属到该组中。这样的处理可以对零散的线程对象进行有效的组织与规划。

创建名称为 groupAddThread 的项目。

创建两个线程类，如图 7-9 所示。

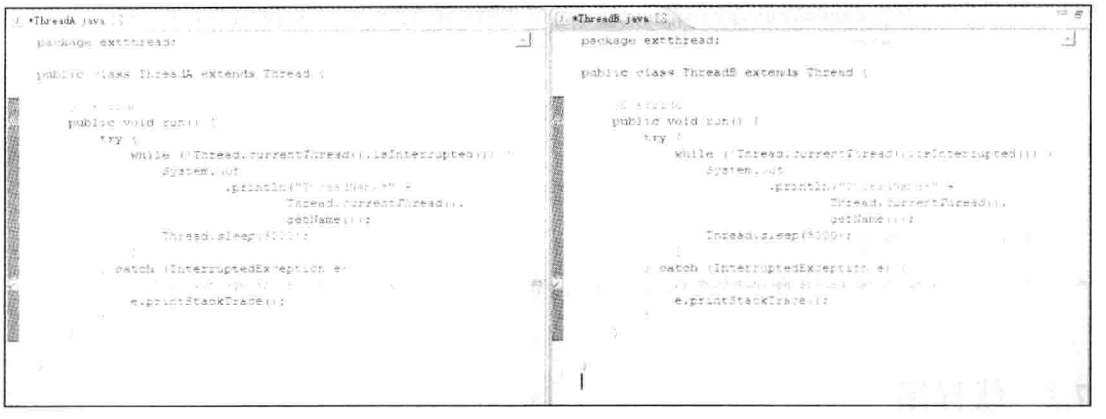


图 7-9 两个线程类代码

创建运行类 Run.java 代码如下：

```
package test;
import extthread.ThreadA;
import extthread.ThreadB;
public class Run {
    public static void main(String[] args) {
        ThreadA aRunnable = new ThreadA();
        ThreadB bRunnable = new ThreadB();
        ThreadGroup group = new ThreadGroup("高洪岩的线程组");
        Thread aThread = new Thread(group, aRunnable);
        Thread bThread = new Thread(group, bRunnable);
        aThread.start();
        bThread.start();
        System.out.println("活动的线程数为: " + group.activeCount());
        System.out.println("线程组的名称为: " + group.getName());
    }
}
```

程序运行后的结果如图 7-10 所示。

图 7-10 运行结果

控制台中打印的信息表示线程组中有两个线程，并且打印出了线程组的名称。另外，两个线程一直无限地并且每隔 3 秒打印日志。

7.2.2 线程对象关联线程组：多级关联

所谓的多级关联就是父对象中有子对象，子对象中再创建子对象，也就是出现子孙对象的效果了。但是此种写法在开发中不太常见，如果线程树结构设计得非常复杂反而不利于线程对象的管理，但 JDK 却提供了支持多级关联的线程树结构。

创建名称为 groupAddThreadMoreLevel 的项目。

创建运行类 Run.java 代码如下：

```
package test.run;
public class Run {
    public static void main(String[] args) {
        //在 main 组中添加一个线程组 A，然后在这个 A 组中添加线程对象 Z
        //方法 activeGroupCount() 和 activeCount() 的值不是固定的
        //是系统中环境的一个快照
        ThreadGroup mainGroup = Thread.currentThread().getThreadGroup();
        ThreadGroup group = new ThreadGroup(mainGroup, "A");
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println("runMethod!");
                    Thread.sleep(10000);//线程必须在运行状态才可以受组管理
                } catch (InterruptedException e) {
                    //TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        };
        Thread newThread = new Thread(group, runnable);
        newThread.setName("Z");
        newThread.start();//线程必须启动然后才归到组 A 中
        ///
        ThreadGroup[] listGroup = new ThreadGroup[Thread.currentThread()
                .getThreadGroup().activeGroupCount()];
        Thread.currentThread().getThreadGroup().enumerate(listGroup);
        System.out.println("main 线程中有多少个子线程组: " + listGroup.length + " 名字为: "
                + listGroup[0].getName());
        Thread[] listThread = new Thread[listGroup[0].activeCount()];
        listGroup[0].enumerate(listThread);
        System.out.println(listThread[0].getName());
    }
}
```

程序运行后的结果如图 7-11 所示。

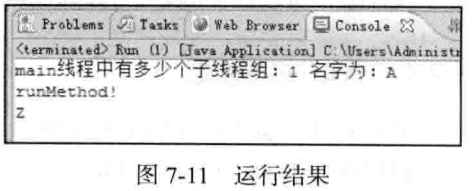


图 7-11 运行结果

本程序代码的结构就是 main 组创建一个新组，然后又在该新组中添加了线程。

7.2.3 线程组自动归属特性

自动归属就是自动归到当前线程组中。

创建名称为 autoAddGroup 的项目。

创建运行类 Run.java 代码如下：

```
package test.run;
public class Run {
    public static void main(String[] args) {
        //方法 activeGroupCount() 取得当前线程组对象中的子线程组数量
        //方法 enumerate() 的作用是将线程组中的子线程组以复制的形式
        //拷贝到 ThreadGroup[] 数组对象中
        System.out.println("A 处线程: " + Thread.currentThread().getName()
                + " 所属的线程组名为: "
                + Thread.currentThread().getThreadGroup().getName() + " "
                + " 中有线程组数量: "
                + Thread.currentThread().getThreadGroup().activeGroupCount());
        ThreadGroup group = new ThreadGroup(" 新的组 ");//自动加到 main 组中
        System.out.println("B 处线程: " + Thread.currentThread().getName()
                + " 所属的线程组名为: "
                + Thread.currentThread().getThreadGroup().getName() + " "
                + " 中有线程组数量: "
                + Thread.currentThread().getThreadGroup().activeGroupCount());
        ThreadGroup[] threadGroup = new ThreadGroup[Thread.currentThread()
                .getThreadGroup().activeGroupCount()];
        Thread.currentThread().getThreadGroup().enumerate(threadGroup);
        for (int i = 0; i < threadGroup.length; i++) {
            System.out.println(" 第一个线程组名称为: " + threadGroup[i].getName());
        }
    }
}
```

程序运行后的结果如图 7-12 所示。

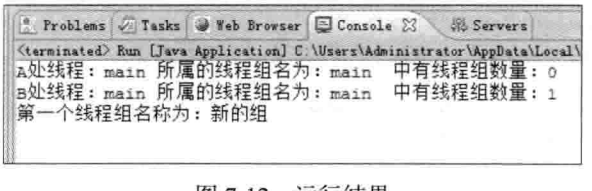


图 7-12 运行结果

本实验要证明的是，在实例化一个 ThreadGroup 线程组 x 时如果不指定所属的线程组，则 x 线程组自动归到当前线程对象所属的线程组中，也就是隐式地在一个线程组中添加了一个子线程组，所以在控制台中打印的线程组数量值由 0 变成了 1。

7.2.4 获取根线程组

创建名称为 getGroupParent 的项目。

创建运行类 Run.java 代码如下：

```
package test.run;
public class Run {
    public static void main(String[] args) {
        System.out.println("线程: " + Thread.currentThread().getName()
                + " 所在的线程组名为: "
                + Thread.currentThread().getThreadGroup().getName());
        System.out
                .println("main 线程所在的线程组的父线程组的名称是: "
                        + Thread.currentThread().getThreadGroup().getParent()
                                .getName());
        System.out.println("main 线程所在的线程组的父线程组的父线程组的名称是: "
                + Thread.currentThread().getThreadGroup().getParent()
                        .getParent().getName());
    }
}
```

程序运行后的结果如图 7-13 所示。

运行结果说明 JVM 的根线程组就是 system，再取其父线程组则出现空异常。

7.2.5 线程组里加线程组

创建名称为 mainGroup 的项目，创建类 Run.java 代码如下：

```
package test.run;
public class Run {
    public static void main(String[] args) {
        System.out.println("线程组名称: "
                + Thread.currentThread().getThreadGroup().getName());
        System.out.println("线程组中活动的线程数量: "
                + Thread.currentThread().getThreadGroup().activeCount());
        System.out.println("线程组中线程组的数量-加之前: "
                + Thread.currentThread().getThreadGroup().activeGroupCount());
        ThreadGroup newGroup = new ThreadGroup(Thread.currentThread()
                .getThreadGroup(), "newGroup");
        System.out.println("线程组中线程组的数量-加之后: "
                + Thread.currentThread().getThreadGroup().activeGroupCount());
        System.out
                .println("父线程组名称: "
                        + Thread.currentThread().getThreadGroup().getParent()
                                .getName());
    }
}
```

程序运行后的结果如图 7-14 所示。

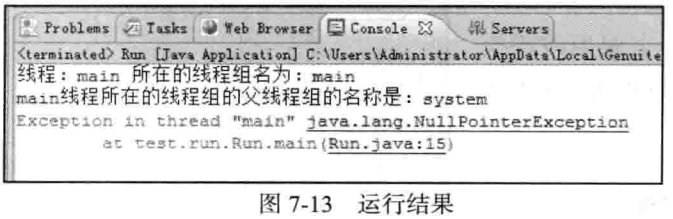


图 7-13 运行结果

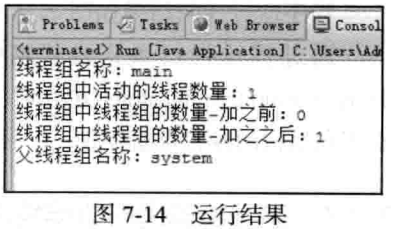


图 7-14 运行结果

本实验用显式的方式在一个线程组中添加了一个子线程组。

7.2.6 组内的线程批量停止

创建名称为 groupInnerStop 的项目。

创建 MyThread.java 类代码如下：

```
package mythread;
public class MyThread extends Thread {
    public MyThread(ThreadGroup group, String name) {
        super(group, name);
    }
    @Override
    public void run() {
        System.out.println("ThreadName=" + Thread.currentThread().getName()
                + "准备开始死循环了：)");
        while (!this.isInterrupted()) {
        }
        System.out.println("ThreadName=" + Thread.currentThread().getName()
                + "结束了：)");
    }
}
```

创建类 Run.java 代码如下：

```
package test.run;
import mythread.MyThread;
public class Run {
    public static void main(String[] args) {
        try {
            ThreadGroup group = new ThreadGroup("我的线程组");
            for (int i = 0; i < 5; i++) {
                MyThread thread = new MyThread(group, "线程" + (i + 1));
                thread.start();
            }
            Thread.sleep(5000);
            group.interrupt();
            System.out.println("调用了 interrupt() 方法");
        } catch (InterruptedException e) {
            System.out.println("停了停了！");
            e.printStackTrace();
        }
    }
}
```

程序运行后的结果如图 7-15 所示。

通过将线程归属到线程组中，当调用线程组 ThreadGroup 的 interrupt() 方法时，可以将该组中的所有正在运行的线程批量停止。

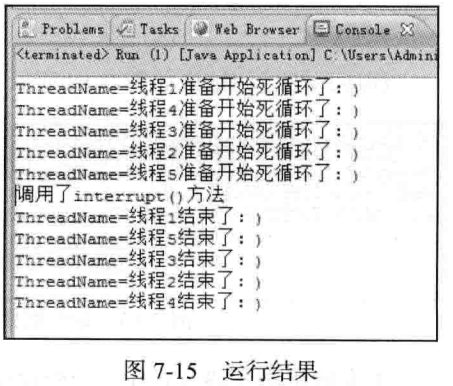


图 7-15 运行结果

7.2.7 递归与非递归取得组内对象

创建名称为 groupRecurseTest 的项目。

创建类 Run.java 代码如下：

```
package test.run;
public class Run {
    public static void main(String[] args) {
        ThreadGroup mainGroup = Thread.currentThread().getThreadGroup();
        ThreadGroup groupA = new ThreadGroup(mainGroup, "A");
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println("runMethod!");
                    Thread.sleep(10000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        ThreadGroup groupB = new ThreadGroup(groupA, "B");
        //分配空间，但不一定全部用完
        ThreadGroup[] listGroup1 = new ThreadGroup[Thread.currentThread()
                .getThreadGroup().activeGroupCount()];
        //传入 true 是递归取得子组及子孙组
        Thread.currentThread().getThreadGroup().enumerate(listGroup1, true);
        for (int i = 0; i < listGroup1.length; i++) {
            if (listGroup1[i] != null) {
                System.out.println(listGroup1[i].getName());
            }
        }
        ThreadGroup[] listGroup2 = new ThreadGroup[Thread.currentThread()
                .getThreadGroup().activeGroupCount()];
        Thread.currentThread().getThreadGroup().enumerate(listGroup2, false);
        for (int i = 0; i <listGroup2.length; i++) {
            if (listGroup2[i] != null) {
                System.out.println(listGroup2[i].getName());
            }
        }
    }
}
```

程序运行后的结果如图 7-16 所示。

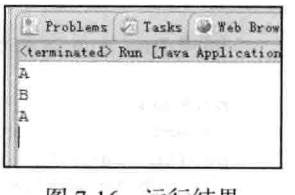


图 7-16 运行结果

7.3 使线程具有有序性

正常的情况下，线程在运行时多个线程之间执行任务的时机是无序的。可以通过改造代码的方式使它们运行具有有序性。

创建名称为 threadRunSyn 的 Java 项目。

创建 MyThread.java 线程类，代码如下：

```
package mythread;
public class MyThread extends Thread {
```

```
    private Object lock;
    private String showChar;
    private int showNumPosition;
    private int printCount = 0;//统计打印了几个字母
    volatile private static int addNumber = 1;
    public MyThread(Object lock, String showChar, int showNumPosition) {
        super();
        this.lock = lock;
        this.showChar = showChar;
        this.showNumPosition = showNumPosition;
    }
    @Override
    public void run() {
        try {
            synchronized (lock) {
                while (true) {
                    if (addNumber % 3 == showNumPosition) {
                        System.out.println("ThreadName="
                                + Thread.currentThread().getName()
                                + " runCount=" + addNumber + " " + showChar);
                        lock.notifyAll();
                        addNumber++;
                        printCount++;
                        if (printCount == 3) {
                            break;
                        }
                    } else {
                        lock.wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

创建 Run.java 运行类代码如下：

```
package test.run;
import mythread.MyThread;
public class Run {
    public static void main(String[] args) {
        Object lock = new Object();
        MyThread a = new MyThread(lock, "A", 1);
        MyThread b = new MyThread(lock, "B", 2);
        MyThread c = new MyThread(lock, "C", 0);
        a.start();
        b.start();
        c.start();
    }
}
```

程序运行后的结果如图 7-17 所示。

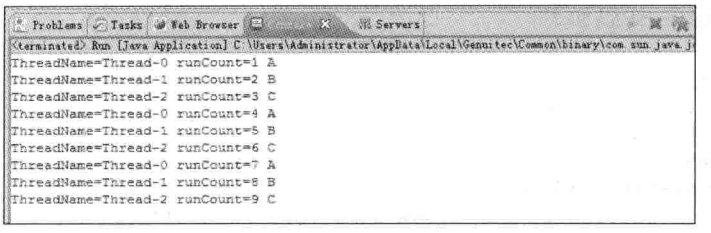


图 7-17 打印 3 批 ABC 字母

7.4 SimpleDateFormat 非线程安全

类 SimpleDateFormat 主要负责日期的转换与格式化，但在多线程的环境中，使用此类容易造成数据转换及处理的不准确，因为 SimpleDateFormat 类并不是线程安全的。

7.4.1 出现异常

本示例将实现使用类 SimpleDateFormat 在多线程环境下处理日期但得出的结果却是错误的情况，这也是在多线程环境开发中容易遇到的问题。

创建项目 formatError，类 MyThread.java 代码如下：

```
package extthread;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
public class MyThread extends Thread {
    private SimpleDateFormat sdf;
    private String dateString;
    public MyThread(SimpleDateFormat sdf, String dateString) {
        super();
        this.sdf = sdf;
        this.dateString = dateString;
    }
    @Override
    public void run() {
        try {
            Date dateRef = sdf.parse(dateString);
            String newDateString = sdf.format(dateRef).toString();
            if (!newDateString.equals(dateString)) {
                System.out.println("ThreadName=" + this.getName()
                        + "报错了 日期字符串: " + dateString + " 转换成的日期为: "
                        + newDateString);
            }
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

运行类 Test.java 代码如下：

```
package test.run;
import java.text.SimpleDateFormat;
import extthread.MyThread;
public class Test {
    public static void main(String[] args) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        String[] dateStringArray = new String[] { "2000-01-01", "2000-01-02",
                "2000-01-03", "2000-01-04", "2000-01-05", "2000-01-06",
                "2000-01-07", "2000-01-08", "2000-01-09", "2000-01-10" };
        MyThread[] threadArray = new MyThread[10];
        for (int i = 0; i < 10; i++) {
            threadArray[i] = new MyThread(sdf, dateStringArray[i]);
        }
        for (int i = 0; i < 10; i++) {
            threadArray[i].start();
        }
    }
}
```

程序运行后的结果如图 7-18 所示。

图 7-18 运行结果

从控制台中打印的结果来看，使用单例的 SimpleDateFormat 类在多线程的环境中处理日期，极易出现日期转换错误的情况。

7.4.2 解决异常方法 1

创建项目，名称为 formatOK1，类 MyThread.java 代码如下：

```
package extthread;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import tools.DateTools;
public class MyThread extends Thread {
    private SimpleDateFormat sdf;
    private String dateString;
    public MyThread(SimpleDateFormat sdf, String dateString) {
        super();
        this.sdf = sdf;
```

```
        this.dateString = dateString;
    }
    @Override
    public void run() {
        try {
            Date dateRef = DateTools.parse("yyyy-MM-dd", dateString);
            String newDateString = DateTools.format("yyyy-MM-dd", dateRef)
                    .toString();
            if (!newDateString.equals(dateString)) {
                System.out.println("ThreadName=" + this.getName()
                        + "报错了 日期字符串: " + dateString + " 转换成的日期为: "
                        + newDateString);
            }
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

类 DateTools.java 代码如下：

```
package tools;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
public class DateTools {
    public static Date parse(String formatPattern, String dateString)
            throws ParseException {
        return new SimpleDateFormat(formatPattern).parse(dateString);
    }
    public static String format(String formatPattern, Date date) {
        return new SimpleDateFormat(formatPattern).format(date).toString();
    }
}
```

运行类 Test.java 代码与前面一节是一样的。

程序运行后的结果如图 7-19 所示。

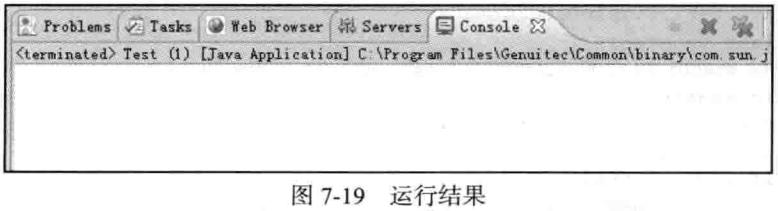


图 7-19 运行结果

控制台中没有输出任何异常，解决处理错误的原理其实就是创建了多个 SimpleDateFormat 类的实例。

7.4.3 解决异常方法 2

前面介绍过，ThreadLocal 类能使线程绑定到指定的对象。使用该类也可以解决多线程

环境下 SimpleDateFormat 类处理错误的情况。

创建项目，名称为 formatOK2，类 MyThread.java 代码如下：

```
package extthread;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import tools.DateTools;
public class MyThread extends Thread {
    private SimpleDateFormat sdf;
    private String dateString;
    public MyThread(SimpleDateFormat sdf, String dateString) {
        super();
        this.sdf = sdf;
        this.dateString = dateString;
    }
    @Override
    public void run() {
        try {
            Date dateRef = DateTools.getSimpleDateFormat("yyyy-MM-dd").parse(
                    dateString);
            String newDateString = DateTools.getSimpleDateFormat("yyyy-MM-dd")
                    .format(dateRef).toString();
            if (!newDateString.equals(dateString)) {
                System.out.println("ThreadName=" + this.getName()
                        + "报错了 日期字符串: " + dateString + " 转换成的日期为: "
                        + newDateString);
            }
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

类 DateTools.java 代码如下：

```
package tools;
import java.text.SimpleDateFormat;
public class DateTools {
    private static ThreadLocal<SimpleDateFormat> tl = new ThreadLocal<SimpleDateFormat>();
    public static SimpleDateFormat getSimpleDateFormat(String datePattern) {
        SimpleDateFormat sdf = null;
        sdf = tl.get();
        if (sdf == null) {
            sdf = new SimpleDateFormat(datePattern);
            tl.set(sdf);
        }
        return sdf;
    }
}
```

运行类 Test.java 代码与前面小节是一样的。

程序运行后的结果如图 7-20 所示。

控制台没有信息被输出，看来运行结果是正确的。

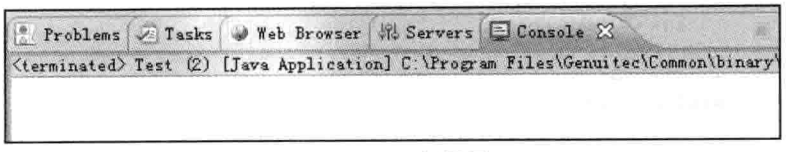


图 7-20 运行结果

7.5 线程中出现异常的处理

创建项目 threadCreateException，创建线程类 MyThread.java，代码如下：

```
package extthread;
public class MyThread extends Thread {
    @Override
    public void run() {
        String username = null;
        System.out.println(username.hashCode());
    }
}
```

创建 Main1.java 文件，代码如下：

```
package controller;
import extthread.MyThread;
public class Main1 {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
    }
}
```

运行结果如图 7-21 所示。

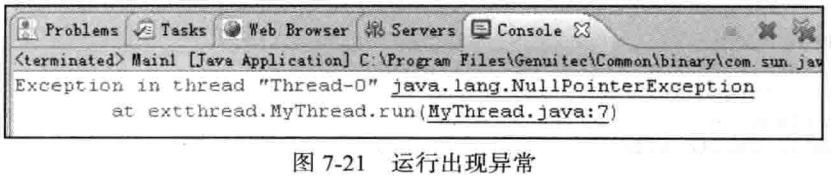


图 7-21 运行出现异常

程序运行后在控制台输出空指针异常。在 Java 的多线程技术中，可以对多线程中的异常进行“捕捉”，使用的是 UncaughtExceptionHandler 类，从而可以对发生的异常进行有效的处理。

创建 Main2.java 文件代码如下：

```
package controller;
import java.lang.Thread.UncaughtExceptionHandler;
import extthread.MyThread;
public class Main2 {
```

```
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.setName("线程 t1");
        t1.setUncaughtExceptionHandler(new UncaughtExceptionHandler() {
            @Override
            public void uncaughtException(Thread t, Throwable e) {
                System.out.println("线程 :" + t.getName() + " 出现了异常: ");
                e.printStackTrace();
            }
        });
        t1.start();
        MyThread t2 = new MyThread();
        t2.setName("线程 t2");
        t2.start();
    }
}
```

方法 setUncaughtExceptionHandler() 的作用是对指定的线程对象设置默认的异常处理器。运行结果如图 7-22 所示。

图 7-22 出现异常

方法 setUncaughtExceptionHandler() 是给指定线程对象设置的异常处理器。在 Thread 类中还可以使用 setDefaultUncaughtExceptionHandler() 方法对所有线程对象设置异常处理器，示例代码如下。

创建 Main3.java 文件代码如下：

```
package controller;
import java.lang.Thread.UncaughtExceptionHandler;
import extthread.MyThread;
public class Main3 {
    public static void main(String[] args) {
        MyThread
            .setDefaultUncaughtExceptionHandler(new UncaughtExceptionHandler() {
                @Override
                public void uncaughtException(Thread t, Throwable e) {
                    System.out.println("线程 :" + t.getName() + " 出现了异常: ");
                    e.printStackTrace();
                }
            });
        MyThread t1 = new MyThread();
        t1.setName("线程 t1");
        t1.start();
```

```
        MyThread t2 = new MyThread();
        t2.setName("线程t2");
        t2.start();
    }
}
```

方法 setDefaultUncaughtExceptionHandler() 的作用是为指定线程类的所有线程对象设置默认的异常处理器。

运行结果如图 7-23 所示。

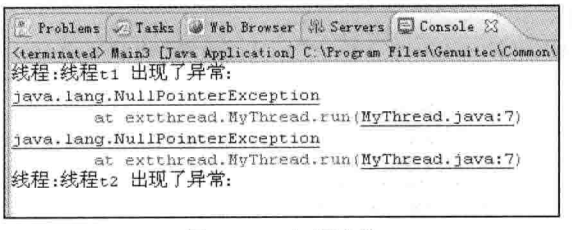


图 7-23 出现异常

7.6 线程组内处理异常

创建项目 threadGroup_1，类 MyThread.java 代码如下：

```
package extthread;
public class MyThread extends Thread {
    private String num;
    public MyThread(ThreadGroup group, String name, String num) {
        super(group, name);
        this.num = num;
    }
    @Override
    public void run() {
        int numInt = Integer.parseInt(num);
        while (true) {
            System.out.println("死循环中: " + Thread.currentThread().getName());
        }
    }
}
```

类 Run.java 代码如下：

```
package test.run;
import extthread.MyThread;
public class Run {
    public static void main(String[] args) {
        ThreadGroup group = new ThreadGroup("我的线程组");
        MyThread[] myThread = new MyThread[10];
        for (int i = 0; i < myThread.length; i++) {
            myThread[i] = new MyThread(group, "线程" + (i + 1), "1");
```

```
            myThread[i].start();
        }
        MyThread newT = new MyThread(group, "报错线程", "a");
        newT.start();
    }
}
```

程序运行后，其中一个线程出现异常，但其他线程却一直以死循环的方式持续打印的结果，如图 7-24 所示。

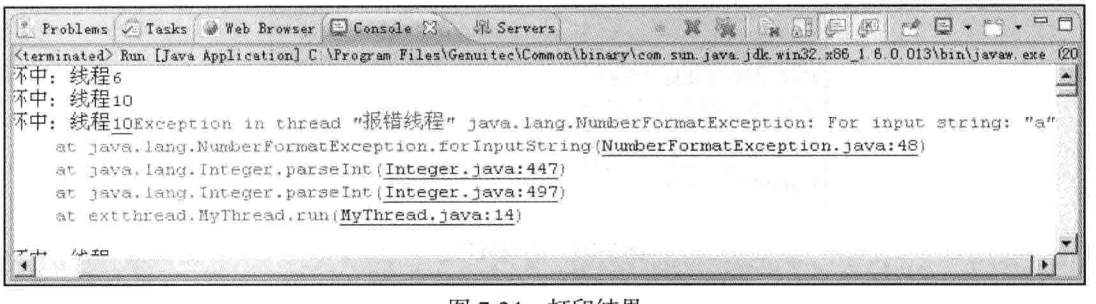


图 7-24 打印结果

红色按钮变成灰色是因为被用鼠标强制停止，而 while（true）死循环是无限输出的。

从运行的结果来看，在默认的情况下，线程组中的一个线程出现异常不会影响其他线程的运行。

如果想实现线程组内一个线程出现异常后全部线程都停止运行该如何实现呢？

创建项目 threadGroup_2，创建新的线程组 MyThreadGroup.java 类，代码如下：

```
package extthreadgroup;
public class MyThreadGroup extends ThreadGroup {
    public MyThreadGroup(String name) {
        super(name);
    }
    @Override
    public void uncaughtException(Thread t, Throwable e) {
        super.uncaughtException(t, e);
        this.interrupt();
    }
}
```

public void uncaughtException(Thread t, Throwable e) 方法中的 t 参数是出现异常的线程对象。

类 MyThread.java 代码如下：

```
package extthread;
public class MyThread extends Thread {
    private String num;
    public MyThread(ThreadGroup group, String name, String num) {
        super(group, name);
        this.num = num;
    }
    @Override
```

```
    public void run() {
        int numInt = Integer.parseInt(num);
        while (this.isInterrupted() == false) {
            System.out.println("死循环中: " + Thread.currentThread().getName());
        }
    }
}
```

需要注意的是，使用自定义 java.lang.ThreadGroup 线程组，并且重写 uncaughtException 方法处理组内线程中断行为时，每个线程对象中的 run() 方法内部不要有异常 catch 语句，如果有 catch 语句，则 public void uncaughtException（Thread t, Throwable e）方法不执行。

类 Run.java 代码如下：

```
package test.run;
import extthread.MyThread;
import extthreadgroup.MyThreadGroup;
public class Run {
    public static void main(String[] args) {
        MyThreadGroup group = new MyThreadGroup("我的线程组");
        MyThread[] myThread = new MyThread[10];
        for (int i = 0; i < myThread.length; i++) {
            myThread[i] = new MyThread(group, "线程" + (i + 1), "1");
            myThread[i].start();
        }
        MyThread newT = new MyThread(group, "报错线程", "a");
        newT.start();
    }
}
```

程序运行后，其中一个线程出现异常，其他线程全部停止了，结果如图 7-25 所示。

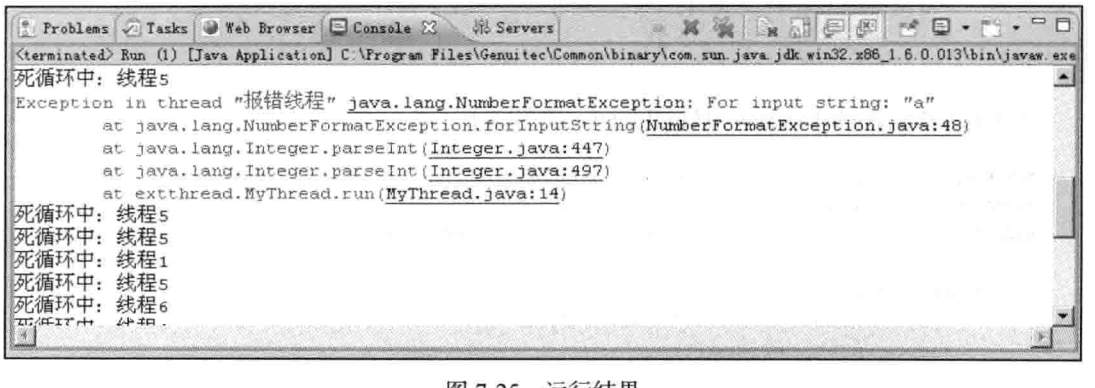


图 7-25 运行结果

7.7 线程异常处理的传递

前面介绍了若干个线程异常处理的方式，那么这些处理的方式如果放在一起运行，会出

现什么样的运行结果呢?

创建测试用的项目 threadExceptionMove，类 MyThread.java 代码如下：

```
package extthread;
public class MyThread extends Thread {
    private String num = "a";
    public MyThread() {
        super();
    }
    public MyThread(ThreadGroup group, String name) {
        super(group, name);
    }
    @Override
    public void run() {
        int numInt = Integer.parseInt(num);
        System.out.println("在线程中打印: " + (numInt + 1));
    }
}
```

类 MyThreadGroup.java 代码如下：

```
package extthreadgroup;
public class MyThreadGroup extends ThreadGroup {
    public MyThreadGroup(String name) {
        super(name);
    }
    @Override
    public void uncaughtException(Thread t, Throwable e) {
        super.uncaughtException(t, e);
        System.out.println("线程组的异常处理");
        e.printStackTrace();
    }
}
```

类 ObjectUncaughtExceptionHandler.java 代码如下：

```
package test.extUncaughtExceptionHandler;
import java.lang.Thread.UncaughtExceptionHandler;
public class ObjectUncaughtExceptionHandler implements UncaughtExceptionHandler {
    @Override
    public void uncaughtException(Thread t, Throwable e) {
        System.out.println("对象的异常处理");
        e.printStackTrace();
    }
}
```

类 StateUncaughtExceptionHandler.java 代码如下：

```
package test.extUncaughtExceptionHandler;
import java.lang.Thread.UncaughtExceptionHandler;
public class StateUncaughtExceptionHandler implements UncaughtExceptionHandler {
```

```
    @Override
    public void uncaughtException(Thread t, Throwable e) {
        System.out.println("静态的异常处理");
        e.printStackTrace();
    }
}
```

创建运行类 Ru java，代码如下：

```
package test;
import test.extUncaughtExceptionHandler.ObjectUncaughtExceptionHandler;
import test.extUncaughtExceptionHandler.StateUncaughtExceptionHandler;
import extthread.MyThread;
public class Run1 {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        //对象
        myThread
            .setUncaughtExceptionHandler(new ObjectUncaughtExceptionHandler());
        //类
        MyThread
            .setDefaultUncaughtExceptionHandler(new StateUncaughtExceptionHandler());
        myThread.start();
    }
}
```

程序运行后的效果如图 7-26 所示。

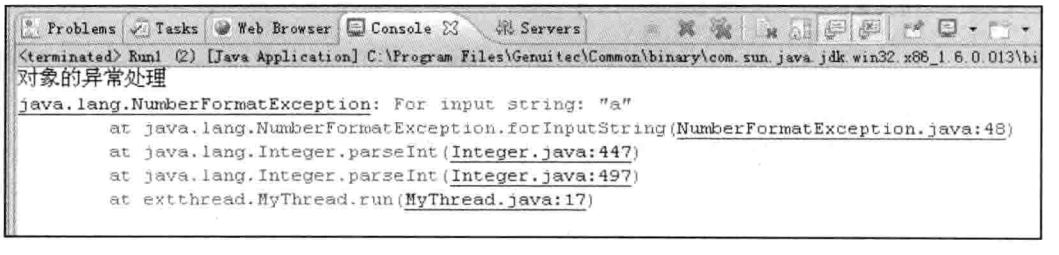


图 7-26 对象异常处理被运行

更改 Run1.java 代码如下：

```
public class Run1 {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        //对象
        //smyThread
        //.setUncaughtExceptionHandler(new ObjectUncaughtExceptionHandler());
        //类
        MyThread
            .setDefaultUncaughtExceptionHandler(new StateUncaughtExceptionHandler());
        myThread.start();
    }
}
```

运行结果如图 7-27 所示。

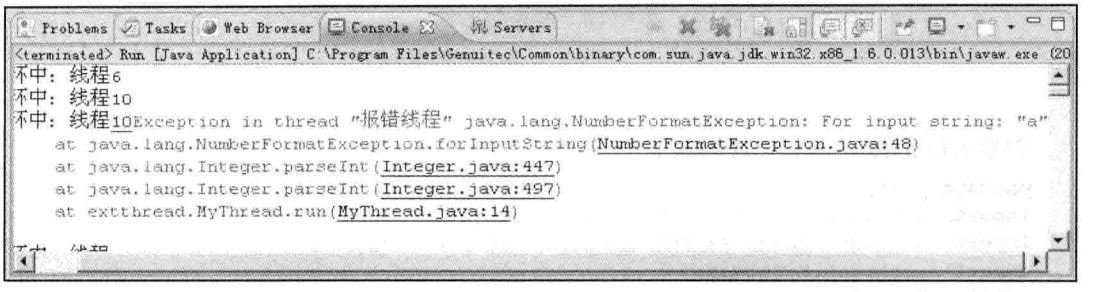


图 7-27 运行结果

继续实验。创建类 Run2.java，代码如下：

```
package test;
import test.extUncaughtExceptionHandler.ObjectUncaughtExceptionHandler;
import test.extUncaughtExceptionHandler.StateUncaughtExceptionHandler;
import extthread.MyThread;
import extthreadgroup.MyThreadGroup;
public class Run2 {
    public static void main(String[] args) {
        MyThreadGroup group = new MyThreadGroup("我的线程组");
        MyThread myThread = new MyThread(group, "我的线程");
        //对象
        myThread
            .setUncaughtExceptionHandler(new ObjectUncaughtExceptionHandler());
        //类
        MyThread
            .setDefaultUncaughtExceptionHandler(new StateUncaughtExceptionHandler());
        myThread.start();
    }
}
```

程序运行后的结果如图 7-28 所示。

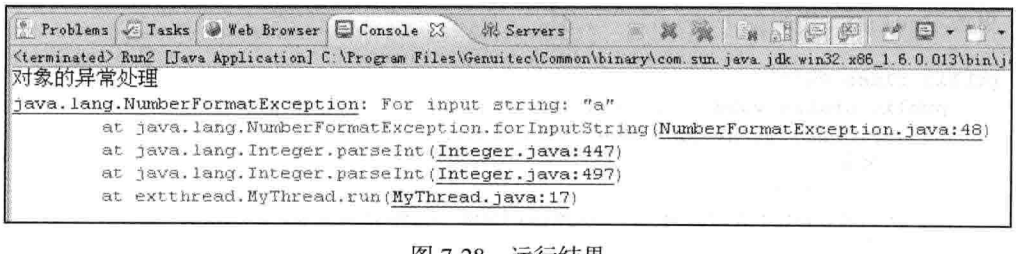


图 7-28 运行结果

更改 Run2.java 代码如下：

```
package test;
```

```
import test.extUncaughtExceptionHandler.ObjectUncaughtExceptionHandler;
import test.extUncaughtExceptionHandler.StateUncaughtExceptionHandler;
import extthread.MyThread;
import extthreadgroup.MyThreadGroup;
public class Run2 {
    public static void main(String[] args) {
        MyThreadGroup group = new MyThreadGroup("我的线程组");
        MyThread myThread = new MyThread(group, "我的线程");
        //对象
        //myThread
        //.setUncaughtExceptionHandler(new ObjectUncaughtExceptionHandler());
        //类
        MyThread
            .setDefaultUncaughtExceptionHandler(new StateUncaughtExceptionHandler());
        myThread.start();
    }
}
```

程序运行后的结果如图 7-29 所示。

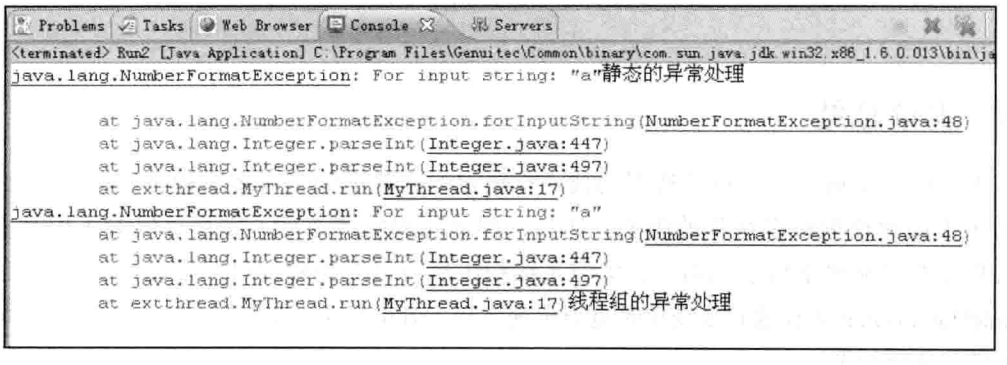


图 7-29 运行结果

本示例想要打印“静态的异常处理”的信息，则必须在 public void uncaughtException（Thread t, Throwable e）方法中加上 super.uncaughtException（t, e）；代码。

继续更改 Run2.java 代码如下：

```
public class Run2 {
    public static void main(String[] args) {
        MyThreadGroup group = new MyThreadGroup("我的线程组");
        MyThread myThread = new MyThread(group, "我的线程");
        //对象
        //myThread
        //.setUncaughtExceptionHandler(new ObjectUncaughtExceptionHandler());
        //类
        //MyThread
        //.setDefaultUncaughtExceptionHandler(new
        //StateUncaughtExceptionHandler());
```

```
            myThread.start();
        }
    }
```

程序运行后的结果如图 7-30 所示。

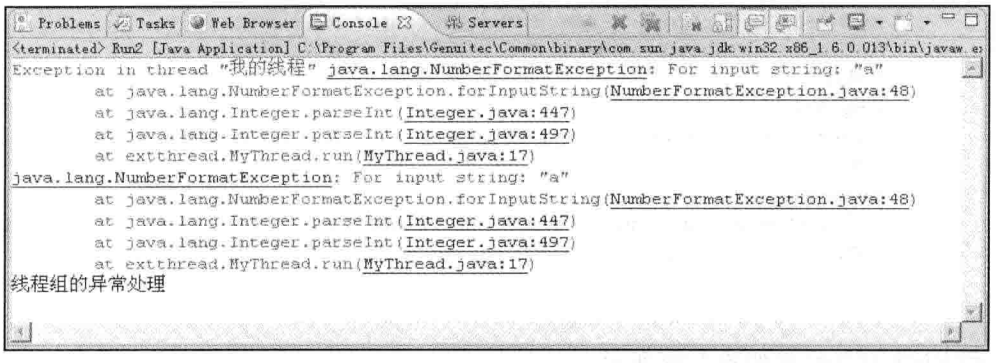


图 7-30 运行结果

7.8 本章总结

本章弥补了前面几个章节遗漏的技术空白点，这些示例是对多线程技术学习的补充，有助于更细化地理解多线程的细节。比如，理解线程的状态后，完全可以对不同状态下的线程正在做哪些事情了如指掌；学习了线程组后可以对线程的组织实施更有效的规划；SimpleDateFormat 类在遇到多线程时也会出现意想不到的异常。最后学习了线程在出现异常时的常用处理方式。

Java 多线程编程核心技术

Java多线程无处不在，如服务器、数据库、应用。多线程可以有效提升计算和处理效率，大大提升吞吐量和可伸缩性，深得广大程序员和公司的青睐。很多人学习完JavaSE/JavaEE之后想往更深入的技术进行探索，比如对大数据、分布式、高并发类的专题进行攻克时，立即遇到针对java.lang包中线程类的学习，但线程类的学习并不像JDBC一样简单，学习曲线陡峭，多弯路与“坑”。要学习这些热点技术，Java多线程技术避无可避。而本书将引领读者拿下该“技术高地”。

本书有以下特点：

- 不留遗漏——全面覆盖Java语言多线程知识点；
- 直击要害——实战化案例精准定位技术细节；
- 学以至用——精要式演示确保开发/学习不脱节；
- 潜移默化——研磨式知识讲解参透技术要点；
- 提升效率——垂直式技术精解不绕弯路；
- 循序提升——渐进式知识点统排确保连贯。

上架指导：计[illegible]设计/Java
ISBN 978-7-111-50206-7
9 787111 502067 >
定价：69.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn