



疯狂

Java讲义

(第2版)

李刚 编著

疯狂源自梦想

技术成就辉煌

疯狂源自梦想

技术成就辉煌



疯狂Java讲义 (第 2 版)

李 刚 编著

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书是《疯狂 Java 讲义》的第 2 版，第 2 版保持了第 1 版系统、全面、讲解浅显、细致的特性，全面介绍了新增的 Java 7 的新特性。

本书深入介绍了 Java 编程的相关方面，全书内容覆盖了 Java 的基本语法结构、Java 的面向对象特征、Java 集合框架体系、Java 泛型、异常处理、Java GUI 编程、JDBC 数据库编程、Java 注释、Java 的 IO 流体系、Java 多线程编程、Java 网络通信编程和 Java 反射机制；覆盖了 java.lang、java.util、java.text、java.io 和 java.nio、java.sql、java.awt、javax.swing 包下绝大部分类和接口。本书全面介绍了 Java 7 的二进制整数、菱形语法、增强 switch 语句、多异常捕获、自动关闭资源的 try 语句、JDBC 4.1 新特性、NIO.2、AIO 等新特性。

与第 1 版类似，本书并不单纯从知识角度来讲解 Java，而是从解决问题的角度来介绍 Java 语言，所以本书中涉及大量的实用案例开发：五子棋游戏、梭哈游戏、仿 QQ 的游戏大厅、MySQL 企业管理器、仿 EditPlus 的文本编辑器、多线程、断点下载工具、Spring 框架的 IoC 容器……这些案例既能让读者巩固每章的知识，又可以让读者学以致用，激发编程自豪感，进而引爆内心的编程激情。本书光盘里包含书中所有示例的代码和《疯狂 Java 实战演义》的所有项目代码，这些项目可以作为本书课后练习的“非标准答案”。如果读者需要获取关于课后习题的解决方法、编程思路，可以登录 <http://www.crazyit.org> 站点与笔者及本书庞大的读者群相互交流。

本书为所有打算深入掌握 Java 编程的读者而编写，适合各种层次的 Java 学习者和工作者阅读，也适合作为大学教育、培训机构的 Java 教材。但如果只是想简单涉猎 Java，则本书过于庞大，不适合阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

疯狂 Java 讲义 / 李刚编著. —2 版. —北京：电子工业出版社，2012.1
ISBN 978-7-121-15578-9

I. ①疯… II. ①李… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字（2011）第 270649 号

策划编辑：张月萍

责任编辑：葛 娜

技术审核：白 涛

印 刷：北京京科印刷有限公司

装 订：三河市皇庄路通装订厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：850×1168 1/16 印张：54 字数：1 747 千字 彩插：4

印 次：2012 年 1 月第 1 次印刷

印 数：5000 册 定价：109.00 元（含光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话（010）88254888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。



如何学习 Java

——谨以此文献给打算以编程为职业、并愿意为之疯狂的人

经常看到有些学生、求职者捧着一本类似 JBuilder 入门、Eclipse 指南之类的图书学习 Java，当他们学会了在这些工具中拖出窗体、安装按钮之后，就觉得自己掌握、甚至精通了 Java；又或是找来一本类似 JSP 动态网站编程之类的图书，学会使用 JSP 脚本编写一些页面后，就自我感觉掌握了 Java 开发。

还有一些学生、求职者听说 J2EE、Spring 或 EJB 很有前途，于是立即跑到书店或图书馆找来一本相关图书。希望立即学会它们，然后进入软件开发业、大显身手。

还有一些学生、求职者非常希望找到一本既速成、又大而全的图书，比如突击 J2EE 开发、一本书精通 J2EE 之类的图书（包括笔者曾出版的《轻量级 J2EE 企业应用实战》一书，据说销量不错），希望这样一本图书就可以打通自己的“任督二脉”，一跃成为 J2EE 开发高手。

也有些学生、求职者非常喜欢 J2EE 项目实战、项目大全之类的图书，他们的想法很单纯：我按照书上介绍，按图索骥、依葫芦画瓢，应该很快就可学会 J2EE，很快就能成为一个受人羡慕的 J2EE 程序员了。

.....

凡此种种，不一而足。但最后的结果往往是失败，因为这种学习没有积累、没有根基，学习过程中困难重重，每天都被一些相同、类似的问题所困扰，起初热情十足，经常上论坛询问，按别人的说法解决问题之后很高兴，既不知道为什么错？也不知道为什么对？只是盲目地抄袭别人的说法。最后的结果有两种：

① 久而久之，热情丧失，最后放弃学习。

② 大部分常见问题都问遍了，最后也可以从事一些重复性开发，但一旦遇到新问题，又将束手无策。

第二种情形在普通程序员中占了极大的比例，笔者多次听到、看到（在网络上）有些程序员抱怨：我做了 2 年多 Java 程序员了，工资还是 3000 多点。偶尔笔者会与他们聊聊工作相关内容，他们会告诉笔者：我也用 Spring 了啊，我也用 EJB 了啊……他们感到非常不平衡，为什么我的工资这么低？其实笔者很想告诉他们：你们太浮躁了！你们确实是用了 Spring、Hibernate 又或是 EJB，但你们未想过为什么要用这些技术？用这些技术有什么好处？如果不用这些技术行不行？

很多时候，我们的程序员把 Java 当成一种脚本，而不是一门面向对象的语言。他们习惯了在 JSP 脚本中使用 Java，但从不去想 JSP 如何运行，Web 服务器里的网络通信、多层机制，为何一个 JSP 页面能同时向多个请求者提供服务？更不会想如何开发 Web 服务器；他们像代码机器一样编写 Spring Bean 代码，但从不去理解 Spring 容器的作用，更不会想如何开发 Spring 容器。

有时候，笔者的学生在编写五子棋、梭哈等作业感到困难时，会向他们的大学师兄、朋友求救，这些程序员告诉他：不用写了，网上有下载的！听到这样回答，笔者不禁感到哑然：网上还有 Windows 下载呢！网上下载和自己编写是两码事。偶尔，笔者会怀念以前黑色屏幕、绿荧荧字符时代，那时候程序员很单纯：当我们想偷懒时，习惯思维是写一个小工具；现在程序员很聪明：当他们想偷懒时，习惯思维是从网上下一个小工具。但是，谁更幸福？

当笔者的学生把他们完成的小作业放上互联网之后，然后就有许多人称他们为“高手”！这个称呼却让他们万分惭愧；惭愧之余，他们也感到万分欣喜，非常有成就感，这就是编程的快乐。编程的过程，与寻宝的过程完全一样：历经辛苦，终于找到心中的梦想，这是何等的快乐？

如果真的打算将编程当成职业，那就不应该如此浮躁，而是应该扎扎实实先学好 Java 语言，然后按 Java 本身的学习规律，踏踏实实一步一个脚印地学习，把基本功练扎实了才可获得更大的成功。

实际情况是，有多少程序员真正掌握了 Java 的面向对象？真正掌握了 Java 的多线程、网络通信、反射等内容？有多少 Java 程序员真正理解了类初始化时内存运行过程？又有多少程序员理解 Java 对象从创建到消失的全部细节？有几个程序员真正独立地编写过五子棋、梭哈、桌面弹球这种小游戏？又有几个 Java 程序员敢说：我可以开发 Struts？我可以开发 Spring？我可以开发 Tomcat？很多人又会说：这些都是许多人开发出来的！实际情况是：许多开源框架的核心最初完全是由一个人开发的。现在这些优秀程序已经出来了！你，是否深入研究过它们，是否深入掌握了它们？

如果要真正掌握 Java，包括后期的 Java EE 相关技术（例如 Struts、Spring、Hibernate 和 EJB 等），一定要记住笔者的话：绝不要从 IDE（如 JBuilder、Eclipse 和 NetBeans）工具开始学习！IDE 工具的功能很强大，初学者学起来也很容易上手，但也非常危险：因为 IDE 工具已经为我们做了许多事情，而软件开发者要全部了解软件开发的全部步骤。

A stylized handwritten signature in black ink, consisting of several loops and a long horizontal stroke at the end.

2011 年 12 月 17 日



光盘说明

一、光盘内容



本光盘是《疯狂 Java 讲义（第 2 版）》一书的配书光盘，书中的代码按章、按节存放，即第 3 章第 1 节所使用的代码放在 codes 文件夹的 03\3.1 文件夹下，依此类推。

另：书中每份源代码也给出与光盘源文件的对应关系，方便读者查找。



本光盘 codes 目录下有 18 个文件夹，其内容和含义说明如下：

(1) 01~18 个文件夹名对应于《疯狂 Java 讲义（第 2 版）》中的章名，即第 3 章所使用的代码放在 codes 文件夹的 03 文件夹下，依此类推。

(2) 本书所有代码都是 IDE 工具无关的程序，读者既可以在命令行窗口直接编译、运行这些代码，也可以导入 Eclipse、NetBeans 等 IDE 工具来运行它们。

(3) 本书第 12 章第 11 节的 TestTableModel.java 程序，以及第 13 章的绝大部分程序都需要连接数据库，所以读者需要先导入 *.sql 文件中的数据库脚本，并修改 mysql.ini 文件中的数据库连接信息。连接数据库时所用的驱动程序 JAR 文件为 mysql-connector-java-3.1.10-bin.jar 文件。这些需要连接数据库的程序里还提供了一个 *.cmd 文件，该文件是一个批处理文件，运行该文件可以运行相应的 Java 程序，例如 DatabaseMetaDataTest.java 对应的 *.cmd 文件为 runDatabaseMetaDataTest.cmd。

(4) 光盘根目录下提供了一个“Java 设计模式（疯狂 Java 联盟版）.chm”文件，这是一份关于设计模式的电子教材，由疯狂 Java 联盟的杨恩雄亲自编写、制作，他同意广大读者阅读、传播这份开源文档。

(5) 因为本书第 1 版有些读者提出关于《疯狂 Java 讲义》课后习题标准答案的问题，因此本书光盘根目录下包含一个 project_codes 文件夹，该文件夹里包含了疯狂 Java 联盟的杨恩雄编写的《疯狂 Java 实战演义》一书的光盘内容，该光盘中包含了大量实战性很强的项目，这些项目基本覆盖了《疯狂 Java 讲义（第 2 版）》课后习题的要求，读者可以参考相关案例来完成《疯狂 Java 讲义（第 2 版）》的课后习题。

二、运行环境



本书中的程序在以下环境中调试通过：

(1) 安装 jdk-7-windows-i586.exe，安装完成后，添加 CLASSPATH 环境变量，该环境变量的值为.;%JAVA_HOME%\lib\tools.jar;%JAVA_HOME%\lib\dt.jar。如果为了可以编译和运行 Java 程序，还应该该在 PATH 环境变量中增加%JAVA_HOME%\bin。其中 JAVA_HOME 代表 JDK（不是 JRE）的安装路径。如何安装上面工具，请参考本书的第 1 章。

(2) 安装 MySQL 5.5 或更高版本，安装 MySQL 时选择 GBK 编码方式（按第 13 章所介绍的方式安装）。

三、注意事项

(1) 书中有大量代码需要连接数据库，读者应修改数据库 URL 以及用户名、密码，让这些代码与

读者运行环境一致。如果项目下有 SQL 脚本，则导入 SQL 脚本即可；如果没有 SQL 脚本，系统将在运行时自动建表，读者只需创建对应的数据库即可。

（2）在使用本光盘的程序时，请将程序拷贝到硬盘上，并去除文件的只读属性。

四、技术支持

如果您在使用本光盘中遇到不懂的技术问题，则可以登录如下网站与作者联系：

<http://www.crazyit.org>



前 言

2011 年 7 月 28 日, Oracle 如约发布了 Java 7 正式版。从 Java 6 到 Java 7, 广大开发者经过了漫长的等待, 新发布的 Java 7 基本满足了广大开发者的渴望: Java 7 加入了不少新特性, 这些新特性进一步增强了 Java 语言的功能。

为了向广大工作者、学习者介绍最新、最前沿的 Java 知识, 在 Java 7 正式发布之前, 笔者已经深入研究过 Java 7 绝大部分可能新增的功能; 当 Java 7 正式发布之后, 笔者在第一时间开始了《疯狂 Java 讲义》的升级: 使用 Java 7 改写了全书所有程序, 全面介绍了 Java 7 的各种新特性。

在以“疯狂 Java 体系”图书为教材的疯狂软件教育中心 (www.fkjava.org) 里, 经常有学生询问: 为什么叫疯狂 Java 这个名字? 也有一些读者通过网络、邮件来询问这个问题。其实这个问题的答案可以在本书第 1 版的前言中找到。疯狂的本质是一种“享受编程”的状态。在一些不了解编程的人看来: 编程的人总面对着电脑、在键盘上敲打, 这种生活实在太枯燥了, 但实际上是因为他们并未真正了解编程, 并未真正走进编程。在外人眼中: 程序员不过是在敲打键盘; 但在程序员心中: 程序员敲出的每个字符, 都是程序的一部分。

程序是什么呢? 程序是对现实世界的数字化模拟。开发一个程序, 实际是创造一个或大或小的“模拟世界”。在这个过程中, 程序员享受着“创造”的乐趣, 程序员沉醉在他所创造的“模拟世界”里: 疯狂地设计、疯狂地编码实现。实现过程不断地遇到问题, 然后解决它; 不断地发现程序的缺陷, 然后重新设计、修复它——这个过程本身就是一种享受。一旦完全沉浸到编程世界里, 程序员是“物我两忘”的, 眼中看到的、心中想到的, 只有他正在创造的“模拟世界”。

在学会享受编程之前, 编程学习者都应该采用“案例驱动”的方式, 学习者需要明白程序的作用是: 解决问题——如果你的程序不能解决你自己的问题, 如何期望你的程序去解决别人的问题呢? 那你的程序的价值何在? ——知道一个知识点能解决什么问题, 才去学这个知识点, 而不是盲目学习! 因此本书强调编程实战, 强调以项目激发编程兴趣。

仅仅是看完这本书, 你不会成为高手! 在编程领域里, 没有所谓的“武林秘籍”, 再好的书一定要配合大量练习, 否则书里的知识依然属于作者, 而读者则仿佛身入宝山而一无所获的笨汉。本书配合了大量高强度的练习, 希望读者强迫自己去完成这些项目。这些习题的答案可以参考本书所附光盘中《疯狂 Java 实战演义》的配套代码。如果需要获得编程思路和交流, 可以登录 <http://www.crazyit.org> 与广大读者和笔者交流。

本书第 1 版面市的近 3 年时间里, 几万读者已经通过本书步入了 Java 编程世界, 这些读者的肯定、赞誉让笔者感到十分欣慰。也有不少读者给予本书一些很好的建议, 因此笔者在创作第 2 版时对部分章节进行了一些小调整, 希望这种调整能让本书日臻完善。

笔者非常欢迎所有热爱编程、愿意推动中国软件业的学习者、工作者对本书提出宝贵的意见, 非常乐意与大家交流。中国软件业还处于发展阶段, 所有热爱编程、愿意推动中国软件业的人应该联合起来, 共同为中国软件行业贡献自己的绵薄之力。

本书有什么特点



本书并不是一本简单的 Java 入门教材, 也不是一门“闭门造车”式的 Java 读物。本书来自于笔者 6 年多的 Java 培训经历, 凝结了笔者近 6000 小时的授课经验, 总结了上千个 Java 学员学习过程中的典型错误。

因此, 本书具有如下三个特点:

1. 案例驱动，引爆编程激情

本书不再是知识点的铺陈，而是致力于将知识点融入实际项目的开发中，所以本书中涉及了大量 Java 案例：仿 QQ 的游戏大厅、MySQL 企业管理器、仿 EditPlus 的文本编辑器、多线程、断点下载工具……希望读者通过编写这些程序找到编程的乐趣。

2. 再现李刚老师课堂氛围

本书的内容是笔者 6 年多授课经历的总结，知识体系取自疯狂 Java 实战的课程体系。

本书力求再现笔者的课堂氛围：以浅显比喻代替乏味的讲解，以疯狂实战代替空洞的理论。

书中包含了大量“注意”、“学生提问”部分，这些正是上千个 Java 学员所犯错误的汇总。

3. 注释详细，轻松上手

为了降低读者阅读的难度，书中代码的注释非常详细，几乎每两行代码就有一行注释。不仅如此，本书甚至还把一些简单理论作为注释穿插到代码中，力求让读者能轻松上手。

本书所有程序中关键代码以粗体字标出，也是为了帮助读者能迅速找到这些程序的关键点。

本书写给谁看








如果你仅仅想对 Java 有所涉猎，那么本书并不适合你；如果你想全面掌握 Java 语言，并使用 Java 来解决问题、开发项目，或者希望以 Java 编程作为你的职业，那么本书将非常适合你。希望本书能引爆你内心潜在的编程激情，如果本书能让你产生废寝忘食的感觉，那笔者就非常欣慰了。










2011-12-17

目 录


CONTENTS

第 1 章 Java 语言概述.....1	第 2 章 理解面向对象.....21
1.1 Java 语言的发展简史.....2	2.1 面向对象.....22
1.2 Java 的竞争对手及各自优势.....4	2.1.1 结构化程序设计简介.....22
1.2.1 C#简介和优势.....4	2.1.2 程序的三种基本结构.....23
1.2.2 Ruby 简介和优势.....5	2.1.3 面向对象程序设计简介.....25
1.2.3 Python 简介和优势.....5	2.1.4 面向对象的基本特征.....26
1.3 Java 程序运行机制.....6	2.2 UML (统一建模语言) 介绍.....27
1.3.1 高级语言的运行机制.....6	2.2.1 用例图.....29
1.3.2 Java 程序的运行机制和 JVM.....6	2.2.2 类图.....29
1.4 开发 Java 的准备.....7	2.2.3 组件图.....31
1.4.1 下载和安装 Java 7 的 JDK.....8	2.2.4 部署图.....32
 不是说JVM是运行Java程序的虚拟机吗? 那JRE和JVM的关系是怎样的呢?.....8	2.2.5 顺序图.....32
 为什么不安装公共JRE呢?.....9	2.2.6 活动图.....33
1.4.2 设置 PATH 环境变量.....10	2.2.7 状态机图.....34
 为什么选择用户变量? 用户变量与系统变量有什么区别?.....11	2.3 Java 的面向对象特征.....35
1.5 第一个 Java 程序.....11	2.3.1 一切都是对象.....35
1.5.1 编辑 Java 源代码.....11	2.3.2 类和对象.....35
1.5.2 编译 Java 程序.....12	2.4 本章小结.....36
 当我们编译C程序时, 不仅需要指定存放目标文件的位置, 也需要指定目标文件的文件名, 这里使用 javac 编译Java程序时怎么不需要指定目标文件的文件名呢?.....12	第 3 章 数据类型和运算符.....37
1.5.3 运行 Java 程序.....13	3.1 注释.....38
1.5.4 根据 CLASSPATH 环境变量定位类.....14	3.1.1 单行注释和多行注释.....38
1.6 Java 程序的基本规则.....15	3.1.2 文档注释.....39
1.6.1 Java 程序的组织形式.....15	 API文档是什么?.....39
1.6.2 Java 源文件的命名规则.....15	 为什么要掌握查看API文档的方法?.....41
1.6.3 初学者容易犯的错误.....16	3.2 标识符和关键字.....45
1.7 垃圾回收机制.....18	3.2.1 分隔符.....45
1.8 何时开始使用 IDE 工具.....19	3.2.2 标识符规则.....46
 我想学习Java编程, 到底是学习 Eclipse好呢, 还是学习NetBeans好呢?.....20	3.2.3 Java 关键字.....46
1.9 本章小结.....20	3.3 数据类型分类.....47
本章练习.....20	3.4 基本数据类型.....47
	3.4.1 整型.....48
	3.4.2 Java 7 新增的二进制整数.....49
	3.4.3 字符型.....50
	3.4.4 浮点型.....51
	3.4.5 Java 7 新增的数值中使用下画线分隔.....52
	3.4.6 布尔型.....52
	3.5 基本类型的类型转换.....53
	3.5.1 自动类型转换.....53

3.5.2	强制类型转换	54	4.6.2	基本类型数组的初始化	93
3.5.3	表达式类型的自动提升	56	4.6.3	引用类型数组的初始化	94
3.6	直接量	57	4.6.4	没有多维数组	96
3.6.1	直接量的类型	57	学生提问	我是否可以让图 4.13 中灰色覆盖的数组元素再次指向另一个数组？这样不就可以扩展成三维数组吗？甚至扩展成更多维的数组？	98
3.6.2	直接量的赋值	57	4.6.5	操作数组的工具类	99
3.7	运算符	58	4.6.6	数组的应用举例	101
3.7.1	算术运算符	58	4.7	本章小结	104
3.7.2	赋值运算符	61	本章练习	104	
3.7.3	位运算符	61	第 5 章	面向对象（上）	105
3.7.4	扩展后的赋值运算符	64	5.1	类和对象	106
3.7.5	比较运算符	65	5.1.1	定义类	106
3.7.6	逻辑运算符	66	学生提问	构造器不是没有返回值吗？为什么不能用 void 修饰呢？	108
3.7.7	三目运算符	66	5.1.2	对象的产生和使用	108
3.7.8	运算符的结合性和优先级	67	5.1.3	对象、引用和指针	109
3.8	本章小结	68	5.1.4	对象的 this 引用	110
本章练习	68	5.2	方法详解	114	
第 4 章	流程控制与数组	70	5.2.1	方法的所属性	114
4.1	顺序结构	71	5.2.2	方法的参数传递机制	115
4.2	分支结构	71	5.2.3	形参个数可变的方法	118
4.2.1	if 条件语句	71	5.2.4	递归方法	119
4.2.2	Java 7 的 switch 分支语句	75	5.2.5	方法重载	121
4.3	循环结构	77	学生提问	为什么方法的返回值类型不能用于区分重载的方法？	121
4.3.1	while 循环语句	77	5.3	成员变量和局部变量	122
4.3.2	do while 循环语句	78	5.3.1	成员变量和局部变量	122
4.3.3	for 循环	79	5.3.2	成员变量的初始化和内存中的运行机制	125
4.3.4	嵌套循环	81	5.3.3	局部变量的初始化和内存中的运行机制	127
4.4	控制循环结构	83	5.3.4	变量的使用规则	128
4.4.1	使用 break 结束循环	83	5.4	隐藏和封装	129
4.4.2	使用 continue 结束本次循环	84	5.4.1	理解封装	129
4.4.3	使用 return 结束方法	85	5.4.2	使用访问控制符	129
4.5	数组类型	86	5.4.3	package、import 和 import static	132
4.5.1	理解数组：数组也是一种类型	86	5.4.4	Java 的常用包	137
学生提问	int[] 是一种类型吗？怎么使用这种类型呢？	86	5.5	深入构造器	137
4.5.2	定义数组	86	5.5.1	使用构造器执行初始化	138
4.5.3	数组的初始化	87	学生提问	构造器是创建 Java 对象的途径，是不是说构造器完全负责创建 Java 对象？	138
学生提问	能不能只分配内存空间，不赋初始值呢？	87			
4.5.4	使用数组	88			
学生提问	为什么要我记住这些异常信息？	89			
4.5.5	foreach 循环	89			
4.6	深入数组	91			
4.6.1	内存中的数组	91			
学生提问	为什么有栈内存和堆内存之分？	91			


5.5.2	构造器重载	139	6.4.1	final 成员变量	174
	为什么要用this来调用另一个重载的构造器？我把另一个构造器里的代码复制、粘贴到这个构造器里不就可以了么？	140	6.4.2	final 局部变量	176
5.6	类的继承	140	6.4.3	final 修饰基本类型变量和引用类型变量的区别	177
5.6.1	继承的特点	141	6.4.4	可执行“宏替换”的 final 变量	177
5.6.2	重写父类的方法	142	6.4.5	final 方法	179
5.6.3	super 限定	143	6.4.6	final 类	180
5.6.4	调用父类构造器	146	6.4.7	不可变类	180
	为什么我创建Java对象时从未感觉到java.lang. Object类的构造器被调用过？	148	6.4.8	缓存实例的不可变类	183
5.7	多态	148	6.5	抽象类	186
5.7.1	多态性	148	6.5.1	抽象方法和抽象类	186
5.7.2	引用变量的强制类型转换	149	6.5.2	抽象类的作用	189
5.7.3	instanceof 运算符	151	6.6	更彻底的抽象：接口	190
5.8	继承与组合	151	6.6.1	接口的概念	190
5.8.1	使用继承的注意点	151	6.6.2	接口的定义	191
5.8.2	利用组合实现复用	153	6.6.3	接口的继承	192
	使用组合关系来实现复用时，需要创建两个Animal对象，是不是意味着使用组合关系时系统开销更大？	155	6.6.4	使用接口	193
5.9	初始化块	156	6.6.5	接口和抽象类	194
5.9.1	使用初始化块	156	6.6.6	面向接口编程	195
5.9.2	初始化块和构造器	157	6.7	内部类	199
5.9.3	静态初始化块	158	6.7.1	非静态内部类	199
5.10	本章小结	160		非静态内部类对象和外部类对象的关系是怎样的？	202
	本章练习	161	6.7.2	静态内部类	203
第 6 章	面向对象（下）	162		为什么静态内部类的实例方法也不能访问外部类的实例属性呢？	204
6.1	Java 7 增强的包装类	163		接口里是否能定义内部接口？	205
	Java为什么要对这些数据进行缓存呢？	166	6.7.3	使用内部类	205
6.2	处理对象	167		既然内部类是外部类的成员，那么是否可以为外部类定义子类，在子类中再定义一个内部类来重写其父类中的内部类？	207
6.2.1	打印对象和 toString 方法	167	6.7.4	局部内部类	207
6.2.2	==和 equals 方法	168	6.7.5	匿名内部类	208
	判断obj是否为Person类的实例时，为何不用obj instanceof Person来判断呢？	172	6.7.6	闭包（Closure）和回调	211
6.3	类成员	172	6.8	枚举类	213
6.3.1	理解类成员	172	6.8.1	手动实现枚举类	213
6.3.2	单例（Singleton）类	173	6.8.2	枚举类入门	215
6.4	final 修饰符	174	6.8.3	枚举类的 Field、方法和构造器	216
			6.8.4	实现接口的枚举类	218
				枚举类不是用final修饰了吗？怎么还能派生子类呢？	207
			6.8.5	包含抽象方法的枚举类	220
			6.9	对象与垃圾回收	221
			6.9.1	对象在内存中的状态	221

6.9.2	强制垃圾回收	222
6.9.3	finalize 方法	223
6.9.4	对象的软、弱和虚引用	225
6.10	修饰符的适用范围	228
6.11	使用 JAR 文件	229
6.11.1	jar 命令详解	230
6.11.2	创建可执行的 JAR 包	232
6.11.3	关于 JAR 包的技巧	232
6.12	本章小结	233
	本章练习	233
第 7 章	与运行环境交互	234
7.1	与用户互动	235
7.1.1	运行 Java 程序的参数	235
7.1.2	使用 Scanner 获取键盘输入	236
7.1.3	使用 BufferedReader 获取 键盘输入	238
7.2	系统相关	238
7.2.1	System 类	239
7.2.2	Runtime 类	240
7.3	常用类	241
7.3.1	Object 类	241
7.3.2	Java 7 新增的 Objects 类	243
7.3.3	String、StringBuffer 和 StringBuilder 类	244
7.3.4	Math 类	247
7.3.5	Java 7 的 ThreadLocalRandom 与 Random	248
7.3.6	BigDecimal 类	250
7.4	处理日期的类	253
7.4.1	Date 类	253
7.4.2	Calendar 类	254
7.4.3	TimeZone 类	257
7.5	正则表达式	258
7.5.1	创建正则表达式	258
7.5.2	使用正则表达式	261
7.6	国际化与格式化	264
7.6.1	Java 国际化的思路	265
7.6.2	Java 支持的国家 and 语言	265
7.6.3	完成程序国际化	266
7.6.4	使用 MessageFormat 处理 包含占位符的字符串	267
7.6.5	使用类文件代替资源文件	268
7.6.6	使用 NumberFormat 格式化数字	269
7.6.7	使用 DateFormat 格式化日期、 时间	270

7.6.8	使用 SimpleDateFormat 格式化 日期	272
7.7	本章小结	273
	本章练习	273
第 8 章	Java 集合	274
8.1	Java 集合概述	275
8.2	Collection 和 Iterator 接口	276
8.2.1	使用 Iterator 接口遍历集合元素	278
8.2.2	使用 foreach 循环遍历集合元素	280
8.3	Set 集合	280
8.3.1	HashSet 类	281
	 hashCode()方法对于HashSet是不 是十分重要?	283
8.3.2	LinkedHashSet 类	285
8.3.3	TreeSet 类	286
8.3.4	EnumSet 类	292
8.3.5	各 Set 实现类的性能分析	293
8.4	List 集合	294
8.4.1	List 接口和 ListIterator 接口	294
8.4.2	ArrayList 和 Vector 实现类	297
8.4.3	固定长度的 List	298
8.5	Queue 集合	298
8.5.1	PriorityQueue 实现类	299
8.5.2	Deque 接口与 ArrayDeque 实现类	300
8.5.3	LinkedList 实现类	301
8.5.4	各种线性表的性能分析	302
8.6	Map	303
8.6.1	HashMap 和 Hashtable 实现类	304
8.6.2	LinkedHashMap 实现类	308
8.6.3	使用 Properties 读写属性文件	308
8.6.4	SortedMap 接口和 TreeMap 实现类	309
8.6.5	WeakHashMap 实现类	311
8.6.6	IdentityHashMap 实现类	312
8.6.7	EnumMap 实现类	313
8.6.8	各 Map 实现类的性能分析	314
8.7	HashSet 和 HashMap 的性能选项	314
8.8	操作集合的工具类: Collections	315
8.8.1	排序操作	315
8.8.2	查找、替换操作	318
8.8.3	同步控制	319
8.8.4	设置不可变集合	320
8.9	烦琐的接口: Enumeration	320
8.10	本章小结	321

本章练习	321
第 9 章 泛型	322
9.1 泛型入门	323
9.1.1 编译时不检查类型的异常	323
9.1.2 手动实现编译时检查类型	324
9.1.3 使用泛型	324
9.1.4 Java 7 泛型的“菱形”语法	325
9.2 深入泛型	326
9.2.1 定义泛型接口、类	326
9.2.2 从泛型类派生子类	328
9.2.3 并不存在泛型类	329
9.3 类型通配符	329
9.3.1 使用类型通配符	331
9.3.2 设定类型通配符的上限	332
9.3.3 设定类型形参的上限	333
9.4 泛型方法	334
9.4.1 定义泛型方法	334
9.4.2 泛型方法和类型通配符的区别	337
9.4.3 Java 7 的“菱形”语法与泛型构造器	338
9.4.4 设定通配符下限	339
9.4.5 泛型方法与方法重载	341
9.5 擦除和转换	341
9.6 泛型与数组	343
9.7 本章小结	344
第 10 章 异常处理	345
10.1 异常概述	346
10.2 异常处理机制	347
10.2.1 使用 try...catch 捕获异常	347
10.2.2 异常类的继承体系	349
10.2.3 Java 7 提供的多异常捕获	351
10.2.4 访问异常信息	352
10.2.5 使用 finally 回收资源	353
10.2.6 异常处理的嵌套	355
10.2.7 Java 7 的自动关闭资源的 try 语句	355
10.3 Checked 异常和 Runtime 异常体系	357
10.3.1 使用 throws 声明抛出异常	357
10.4 使用 throw 抛出异常	359
10.4.1 抛出异常	359
10.4.2 自定义异常类	360
10.4.3 catch 和 throw 同时使用	361
10.4.4 Java 7 增强的 throw 语句	362

10.4.5 异常链	363
10.5 Java 的异常跟踪栈	365
10.6 异常处理规则	366
10.6.1 不要过度使用异常	367
10.6.2 不要使用过于庞大的 try 块	368
10.6.3 避免使用 Catch All 语句	368
10.6.4 不要忽略捕获到的异常	368
10.7 本章小结	369
本章练习	369
第 11 章 AWT 编程	370
11.1 GUI (图形用户界面) 和 AWT	371
11.2 AWT 容器	372
11.3 布局管理器	375
11.3.1 FlowLayout 布局管理器	375
11.3.2 BorderLayout 布局管理器	376
学生提问 BorderLayout 最多只能放置 5 个组件吗？那它也太不实用了吧？	377
11.3.3 GridLayout 布局管理器	378
11.3.4 GridBagLayout 布局管理器	379
11.3.5 CardLayout 布局管理器	381
11.3.6 绝对定位	383
11.3.7 BoxLayout 布局管理器	384
学生提问 图 11.15 和图 11.16 显示的所有按钮都紧挨在一起，如果希望像 FlowLayout、GridLayout 等布局管理器那样指定组件的间距应该怎么办？	385
11.4 AWT 常用组件	386
11.4.1 基本组件	386
11.4.2 对话框 (Dialog)	388
11.5 事件处理	390
11.5.1 Java 事件模型的流程	390
11.5.2 事件和事件监听器	392
11.5.3 事件适配器	396
11.5.4 使用内部类实现监听器	397
11.5.5 使用外部类实现监听器	397
11.5.6 类本身作为事件监听器类	398
11.5.7 匿名内部类实现监听器	399
11.6 AWT 菜单	399
11.6.1 菜单条、菜单和菜单项	399
11.6.2 右键菜单	401
学生提问 为什么即使我没有给多行文本域编写右键菜单，但当我在多行文本域上单击右键时也一样会弹出右键菜单？	403

11.7	在 AWT 中绘图	403	12.6.1	创建进度条	482
11.7.1	画图的实现原理	403	12.6.2	创建进度对话框	485
11.7.2	使用 Graphics 类	404	12.7	使用 JSlider 和 BoundedRangeModel 创建滑动条	486
11.8	处理位图	408	12.8	使用 JSpinner 和 SpinnerModel 创建微调控制器	489
11.8.1	Image 抽象类和 BufferedImage 实现类	408	12.9	使用 JList、JcomboBox 创建列表框	492
11.8.2	使用 ImageIO 输入/输出位图	411	12.9.1	简单列表框	493
11.9	剪贴板	414	12.9.2	不强制存储列表项的 ListModel 和 ComboBoxModel	496
11.9.1	数据传递的类和接口	415	12.9.3	强制存储列表项的 DefaultListModel 和 DefaultComboBoxModel	499
11.9.2	传递文本	415	 为什么JComboBox提供了添加、删 除列表项的方法？而JList没有提供 添加、删除列表项的方法呢？	510	
11.9.3	使用系统剪贴板传递图像	417	12.9.4	使用 ListCellRenderer 改变列表 项外观	501
11.9.4	使用本地剪贴板传递对象引用	421	12.10	使用 JTree 和 TreeModel 创建树	503
11.9.5	通过系统剪贴板传递 Java 对象	423	12.10.1	创建树	504
11.10	拖放功能	426	12.10.2	拖动、编辑树节点	507
11.10.1	拖放目标	427	12.10.3	监听节点事件	511
11.10.2	拖放源	429	12.10.4	使用 DefaultTreeCellRenderer 改变节点外观	512
11.11	本章小结	431	12.10.5	扩展 DefaultTreeCellRenderer 改变节点外观	514
	本章练习	431	12.10.6	实现 TreeCellRenderer 改变 节点外观	516
第 12 章	Swing 编程	432	12.11	使用 JTable 和 TableModel 创建表格	518
12.1	Swing 概述	433	12.11.1	创建表格	518
12.2	Swing 基本组件的用法	434	 我们指定的表格数据、表格列标题 都是Object类型的数组，JTable如何 显示这些Object对象？	519	
12.2.1	Java 7 的 Swing 组件层次	434	12.11.2	TableModel 和监听器	524
12.2.2	AWT 组件的 Swing 实现	435	12.11.3	TableColumnModel 和监听器	528
 为什么单击Swing多行文本域时不 是弹出像AWT多行文本域中的右 键菜单？	441	12.11.4	实现排序	531	
12.2.3	为组件设置边框	441	12.11.5	绘制单元格内容	534
12.2.4	Swing 组件的双缓冲和键盘驱动	443	12.11.6	编辑单元格内容	537
12.2.5	使用 JToolBar 创建工具条	444	12.12	使用 JFormattedTextField 和 JTextPane 创建格式文本	541
12.2.6	使用 JFileChooser 和 Java 7 增强的 JColorChooser	446	12.12.1	监听 Document 的变化	541
12.2.7	使用 JOptionPane	454	12.12.2	使用 JPasswordField	543
12.3	Swing 中的特殊容器	459	12.12.3	使用 JFormattedTextField	544
12.3.1	使用 JSplitPane	459	12.12.4	使用 JEditorPane	552
12.3.2	使用 JTabbedPane	461			
12.3.3	使用 JLayeredPane、JDesktopPane 和 JInternalFrame	464			
12.4	Swing 简化的拖放功能	472			
12.5	Java 7 新增的 Swing 功能	473			
12.5.1	使用 JLayer 装饰组件	473			
12.5.2	创建透明、不规则形状窗口	479			
12.6	使用 JProgressBar、ProgressMonitor 和 BoundedRangeModel 创建进度条	481			

12.12.5 使用 JTextPane	552
12.13 本章小结	559
本章练习	559

第 13 章 MySQL 数据库与 JDBC 编程

13.1 JDBC 基础	561
13.1.1 JDBC 简介	561
13.1.2 JDBC 驱动程序	562
13.2 SQL 语法	563
13.2.1 安装数据库	563
13.2.2 关系数据库基本概念和 MySQL 基本命令	565
13.2.3 SQL 语句基础	567
13.2.4 DDL 语句	568
13.2.5 数据库约束	572
13.2.6 索引	579
13.2.7 视图	579
13.2.8 DML 语句语法	580
13.2.9 单表查询	583
13.2.10 数据库函数	587
13.2.11 分组和组函数	589
13.2.12 多表连接查询	591
13.2.13 子查询	595
13.2.14 集合运算	596
13.3 JDBC 的典型用法	597
13.3.1 JDBC 常用接口和类简介	598
13.3.2 JDBC 编程步骤	599



前面给出的仅仅是 MySQL 和 Oracle 两种数据库的驱动,我看不出驱动类字符串有什么规律啊。如果我希望使用其他数据库,那怎么找到其他数据库的驱动类呢?

13.4 执行 SQL 语句的方式	602
13.4.1 使用 executeUpdate 方法执行 DDL 和 DML 语句	602
13.4.2 使用 execute 方法执行 SQL 语句	604
13.4.3 使用 PreparedStatement 执行 SQL 语句	605
13.4.4 使用 CallableStatement 调用 存储过程	609
13.5 管理结果集	611
13.5.1 可滚动、可更新的结果集	611
13.5.2 处理 Blob 类型数据	613
13.5.3 使用 ResultSetMetaData 分析结果集	617

13.6 Java 7 的 RowSet 1.1	620
13.6.1 Java 7 新增的 RowSetFactory 与 RowSet	620
13.6.2 离线 RowSet	623
13.6.3 离线 RowSet 的查询分页	624
13.7 事务处理	625
13.7.1 事务的概念和 MySQL 事务支持	626
13.7.2 JDBC 的事务支持	627
13.7.3 批量更新	629
13.8 分析数据库信息	630
13.8.1 使用 DatabaseMetaData 分析数据库信息	630
13.8.2 使用系统表分析数据库信息	632
13.8.3 选择合适的分析方式	633
13.9 使用连接池管理连接	633
13.9.1 DBCP 数据源	634
13.9.2 C3P0 数据源	634
13.10 本章小结	635
本章练习	635


第 14 章 Annotation (注释)

14.1 基本 Annotation	637
14.1.1 限定重写父类方法: @Override	637
14.1.2 标示已过时: @Deprecated	638
14.1.3 抑制编译器警告: @SuppressWarnings	639
14.1.4 Java 7 的“堆污染”警告与 @SafeVarargs	639
14.2 JDK 的元 Annotation	640
14.2.1 使用 @Retention	640
14.2.2 使用 @Target	641
14.2.3 使用 @Documented	641
14.2.4 使用 @Inherited	642
14.3 自定义 Annotation	643
14.3.1 定义 Annotation	643
14.3.2 提取 Annotation 信息	645
14.3.3 使用 Annotation 的示例	646
14.4 编译时处理 Annotation	650
14.5 本章小结	654

第 15 章 输入/输出

15.1 File 类	656
15.1.1 访问文件和目录	656
15.1.2 文件过滤器	658
15.2 理解 Java 的 IO 流	659

15.2.1	流的分类	659	16.2.1	继承 Thread 类创建线程类	710
15.2.2	流的概念模型	660	16.2.2	实现 Runnable 接口创建线程类	711
15.3	字节流和字符流	661	16.2.3	使用 Callable 和 Future 创建线程	713
15.3.1	InputStream 和 Reader	661	16.2.4	创建线程的三种方式对比	714
15.3.2	OutputStream 和 Writer	663	16.3	线程的生命周期	715
15.4	输入/输出流体系	665	16.3.1	新建和就绪状态	715
15.4.1	处理流的用法	665	16.3.2	运行和阻塞状态	716
15.4.2	输入/输出流体系	666	16.3.3	线程死亡	717
15.4.3	转换流	668	16.4	控制线程	718
	怎么没有把字符流转换成字节流的转换流呢?	668	16.4.1	join 线程	719
15.4.4	推回输入流	669	16.4.2	后台线程	720
15.5	重定向标准输入/输出	671	16.4.3	线程睡眠: sleep	721
15.6	Java 虚拟机读写其他 进程的数据	672	16.4.4	线程让步: yield	721
15.7	RandomAccessFile	674	16.4.5	改变线程优先级	722
15.8	对象序列化	677	16.5	线程同步	724
15.8.1	序列化的含义和意义	677	16.5.1	线程安全问题	724
15.8.2	使用对象流实现序列化	678	16.5.2	同步代码块	726
15.8.3	对象引用的序列化	680	16.5.3	同步方法	727
15.8.4	自定义序列化	683	16.5.4	释放同步监视器的锁定	729
15.8.5	另一种自定义序列化机制	688	16.5.5	同步锁 (Lock)	730
15.8.6	版本	690	16.5.6	死锁	732
15.9	NIO	691	16.6	线程通信	733
15.9.1	Java 新 IO 概述	691	16.6.1	传统的线程通信	734
15.9.2	使用 Buffer	691	16.6.2	使用 Condition 控制线程通信	737
15.9.3	使用 Channel	694	16.6.3	使用阻塞队列 (BlockingQueue) 控制线程通信	739
15.9.4	字符集和 Charset	697	16.7	线程组和未处理的异常	742
	二进制序列与字符之间如何对应 呢?	698	16.8	线程池	744
15.9.5	文件锁	699	16.8.1	Java 5 实现的线程池	745
15.10	Java 7 的 NIO.2	701	16.8.2	Java 7 新增的 ForkJoinPool	746
15.10.1	Path、Paths 和 Files 核心 API	701	16.9	线程相关类	750
15.10.2	使用 FileVisitor 遍历 文件和目录	702	16.9.1	ThreadLocal 类	750
15.10.3	使用 WatchService 监控文件 变化	704	16.9.2	包装线程不安全的集合	752
15.10.4	访问文件属性	705	16.9.3	线程安全的集合类	752
15.11	本章小结	706	16.10	本章小结	753
	本章练习	706	第 17 章	网络编程	754
第 16 章	多线程	707	17.1	网络编程的基础知识	755
16.1	线程概述	708	17.1.1	网络基础知识	755
16.1.1	线程和进程	708	17.1.2	IP 地址和端口号	756
16.1.2	多线程的优势	709	17.2	Java 的基本网络支持	757
16.2	线程的创建和启动	710	17.2.1	使用 InetAddress	757
			17.2.2	使用 URLDecoder 和 URLEncoder	758
			17.2.3	使用 URL 和 URLConnection	759

17.3	基于 TCP 协议的网络编程	765	18.1.1	JVM 和类	813
17.3.1	TCP 协议基础	765	18.1.2	类的加载	814
17.3.2	使用 ServerSocket 创建 TCP 服务器端	766	18.1.3	类的连接	815
17.3.3	使用 Socket 进行通信	767	18.1.4	类的初始化	815
17.3.4	加入多线程	769	18.1.5	类初始化的时机	816
17.3.5	记录用户信息	772	18.2	类加载器	817
17.3.6	半关闭的 Socket	779	18.2.1	类加载器简介	817
17.3.7	使用 NIO 实现非阻塞 Socket 通信	780	18.2.2	类加载机制	818
17.3.8	使用 Java 7 的 AIO 实现 非阻塞通信	786	18.2.3	创建并使用自定义的类 加载器	820
	 上面程序中好像没用到④⑥号代 码的get()方法的返回值，这两个地 方不调用get()方法行吗？	789	18.2.4	URLClassLoader 类	823
17.4	基于 UDP 协议的网络编程	793	18.3	通过反射查看类信息	824
17.4.1	UDP 协议基础	793	18.3.1	获得 Class 对象	825
17.4.2	使用 DatagramSocket 发送、 接收数据	793	18.3.2	从 Class 中获取信息	825
17.4.3	使用 MulticastSocket 实现 多点广播	797	18.4	使用反射生成并操作对象	829
17.5	使用代理服务器	807	18.4.1	创建对象	829
17.5.1	直接使用 Proxy 创建连接	808	18.4.2	调用方法	831
17.5.2	使用 ProxySelector 自动选择 代理服务器	809	18.4.3	访问属性值	833
17.6	本章小结	811	18.4.4	操作数组	834
	本章练习	811	18.5	使用反射生成 JDK 动态代理	835
第 18 章	类加载机制与反射	812	18.5.1	使用 Proxy 和 InvocationHandler 创建动态代理	835
18.1	类的加载、连接和初始化	813	18.5.2	动态代理和 AOP	837
			18.6	反射和泛型	841
			18.6.1	泛型和 Class 类	841
			18.6.2	使用反射来获取泛型信息	843
			18.7	本章小结	844
				本章练习	844

4.5 数组类型

数组是编程语言中最常见的一种数据结构，可用于存储多个数据，每个数组元素存放一个数据，通常可通过数组元素的索引来访问数组元素，包括为数组元素赋值和取出数组元素的值。Java 语言的数组则具有其特有的特征，下面将详细介绍 Java 语言的数组。

4.5.1 理解数组：数组也是一种类型

Java 的数组要求所有的数组元素具有相同的数据类型。因此，在一个数组中，数组元素的类型是唯一的，即一个数组里只能存储一种数据类型的数据，而不能存储多种数据类型的数据。

因为 Java 语言是面向对象的语言，而类与类之间可以支持继承关系，这样可能产生一个数组里可以存放多种数据类型的假象。例如有一个水果数组，要求每个数组元素都是水果，实际上数组元素既可以是苹果，也可以是香蕉，但这个数组的数组元素的类型还是唯一的，只能是水果类型。



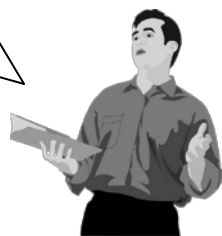
一旦数组的初始化完成，数组在内存中所占的空间将被固定下来，因此数组的长度将不可改变。即使把某个数组元素的数据清空，但它所占的空间依然被保留，依然属于该数组，数组的长度依然不变。

Java 的数组既可以存储基本类型的数据，也可以存储引用类型的数据，只要所有的数组元素具有相同的类型即可。

值得指出的是，数组也是一种数据类型，它本身是一种引用类型。例如 `int` 是一个基本类型，但 `int[]`（这是定义数组的一种方式）就是一种引用类型了。

学生提问：`int[]` 是一种类型吗？怎么使用这种类型呢？

答：没错，`int[]` 就是一种数据类型，与 `int` 类型、`String` 类型类似，一样可以使用该类型来定义变量，也可以使用该类型进行类型转换等。使用 `int[]` 类型来定义变量、进行类型转换时与使用其他普通类型没有任何区别。`int[]` 类型是一种引用类型，创建 `int[]` 类型的对象也就是创建数组，需要使用创建数组的语法。



4.5.2 定义数组

Java 语言支持两种语法格式来定义数组：

```
type[] arrayName;  
type arrayName[];
```

对这两种语法格式而言，通常推荐使用第一种格式。因为第一种格式不仅具有更好的语意，而且具有更好的可读性。对于 `type[] arrayName;` 方式，很容易理解这是定义一个变量，其中变量名是 `arrayName`，而变量类型是 `type[]`。前面已经指出：`type[]` 确实是一种新类型，与 `type` 类型完全不同（例如 `int` 类型是基本类型，但 `int[]` 是引用类型）。因此，这种方式既容易理解，也符合定义变量的语法。但第二种格式 `type arrayName[]` 的可读性就差了，看起来好像定义了一个类型为 `type` 的变量，而变量名是 `arrayName[]`，这与真实的含义相去甚远。

可能有些读者非常喜欢 `type arrayName[]`；这种定义数组的方式，这可能是因为早期某些计算机读物的误导，从现在开始就不要再使用这种糟糕的方式了。



提示：

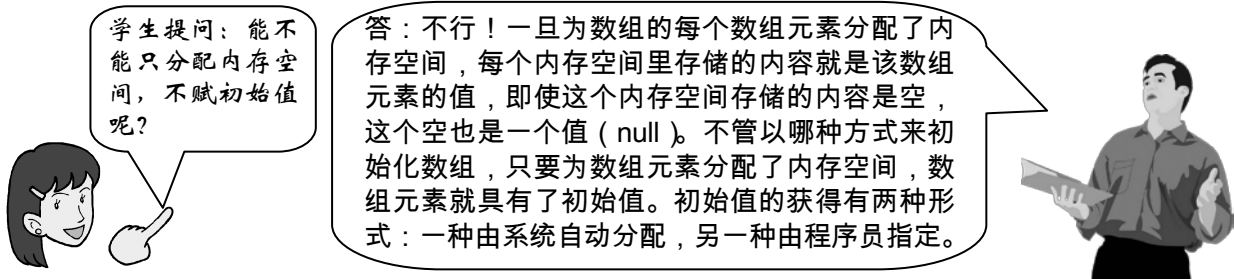
Java 的模仿者 C# 就不再支持 `type arrayName[]` 这种语法，它只支持第一种定义数组的语法。越来越多的语言不再支持 `type arrayName[]` 这种数组定义语法。

数组是一种引用类型的变量，因此使用它定义一个变量时，仅仅表示定义了一个引用变量（也就是定义了一个指针），这个引用变量还未指向任何有效的内存，因此定义数组时不能指定数组的长度。而且由于定义数组只是定义了一个引用变量，并未指向任何有效的内存空间，所以还没有内存空间来存储数组元素，因此这个数组也不能使用，只有对数组进行初始化后才可以使

定义数组时不能指定数组的长度。

4.5.3 数组的初始化

Java 语言中数组必须先初始化，然后才可以使用。所谓初始化，就是为数组的数组元素分配内存空间，并为每个数组元素赋初始值。



数组的初始化有如下两种方式。

- 静态初始化：初始化时由程序员显式指定每个数组元素的初始值，由系统决定数组长度。
- 动态初始化：初始化时程序员只指定数组长度，由系统为数组元素分配初始值。

1. 静态初始化

静态初始化的语法格式如下：

```
arrayName = new type[]{element1, element2, element3, element4 ...}
```

在上面的语法格式中，前面的 `type` 就是数组元素的数据类型，此处的 `type` 必须与定义数组变量时所使用的 `type` 相同，也可以是定义数组时所指定的 `type` 的子类，并使用花括号把所有的数组元素括起来，多个数组元素之间以英文逗号（,）隔开，定义初始化值的花括号紧跟 `[]` 之后。值得指出的是，执行静态初始化时，显式指定的数组元素值的类型必须与 `new` 关键字后的 `type` 类型相同，或者是其子类的实例。下面代码定义使用了这三种形式来进行静态初始化。

程序清单：codes\04\4.5\ArrayTest.java

```
//定义一个 int 数组类型的变量，变量名为 intArr
int[] intArr;
//使用静态初始化，初始化数组时只指定数组元素的初始值，不指定数组长度
intArr = new int[]{5, 6, 8, 20};
//定义一个 Object 数组类型的变量，变量名为 objArr
Object[] objArr;
//使用静态初始化，初始化数组时数组元素的类型是
//定义数组时数组元素类型的子类
objArr = new String[] {"Java", "李刚"};
Object[] objArr2;
//使用静态初始化
```

```
objArr2 = new Object[] {"Java" , "李刚"};
```

因为 Java 语言是面向对象的编程语言，能很好地支持子类和父类的继承关系：子类实例是一种特殊的父类实例。在上面程序中，String 类型是 Object 类型的子类，即字符串是一种特殊的 Object 实例。关于继承更详细的介绍，请参考本书第 5 章。

除此之外，静态初始化还有如下简化的语法格式：

```
arrayName = {element1, element2, element3, element4 ...}
```

在这种语法格式中，直接使用花括号来定义一个数组，花括号把所有的数组元素括起来形成一个数组。在实际开发过程中，可能更习惯将数组定义和数组初始化同时完成，代码如下（程序清单同上）：

```
//数组的定义和初始化同时完成，使用简化的静态初始化写法  
int[] a = {5, 6, 7, 9};
```

2. 动态初始化

动态初始化只指定数组的长度，由系统为每个数组元素指定初始值。动态初始化的语法格式如下：

```
arrayName = new type[length];
```

在上面语法中，需要指定一个 int 类型的 length 参数，这个参数指定了数组的长度，也就是可以容纳数组元素的个数。与静态初始化相似的是，此处的 type 必须与定义数组时使用的 type 类型相同，或者是定义数组时使用的 type 类型的子类。下面代码示范了如何进行动态初始化（程序清单同上）。

```
//数组的定义和初始化同时完成，使用动态初始化语法  
int[] prices = new int[5];  
//数组的定义和初始化同时完成，初始化数组时元素的类型是定义数组时元素类型的子类  
Object[] books = new String[4];
```

执行动态初始化时，程序员只需指定数组的长度，即为每个数组元素指定所需的内存空间，系统将负责为这些数组元素分配初始值。指定初始值时，系统按如下规则分配初始值。

- 数组元素的类型是基本类型中的整数类型（byte、short、int 和 long），则数组元素的值是 0。
- 数组元素的类型是基本类型中的浮点类型（float、double），则数组元素的值是 0.0。
- 数组元素的类型是基本类型中的字符类型（char），则数组元素的值是 '\u0000'。
- 数组元素的类型是基本类型中的布尔类型（boolean），则数组元素的值是 false。
- 数组元素的类型是引用类型（类、接口和数组），则数组元素的值是 null。

不要同时使用静态初始化和动态初始化，也就是说，不要在进行数组初始化时，既指定数组的长度，也为每个数组元素分配初始值。



数组初始化完成后，就可以使用数组了，包括为数组元素赋值、访问数组元素值和获得数组长度等。

4.5.4 使用数组

数组最常用的用法就是访问数组元素，包括对数组元素进行赋值和取出数组元素的值。访问数组元素都是通过数组引用变量后紧跟一个方括号（[]），方括号里是数组元素的索引值，这样就可以访问数组元素了。访问到数组元素后，就可以把一个数组元素当成一个普通变量使用了，包括为该变量赋值和取出该变量的值，这个变量的类型就是定义数组时使用的类型。

值得指出的是，Java 语言的数组索引是从 0 开始的，也就是说，第一个数组元素的索引值为 0，最后一个数组元素的索引值为数组长度减 1。下面代码示范了输出数组元素的值，以及为指定数组元素赋值（程序清单同上）。

```
//输出 objArr 数组的第二个元素，将输出字符串"李刚"  
System.out.println(objArr[1]);  
//为 objArr2 的第一个数组元素赋值  
objArr2[0] = "Spring";
```

如果访问数组元素时指定的索引值小于 0, 或者大于等于数组的长度, 编译程序不会出现任何错误, 但运行时出现异常: `java.lang.ArrayIndexOutOfBoundsException: N` (数组索引越界异常), 异常信息后的 `N` 就是程序员试图访问的数组索引。

下面代码试图访问的数组元素索引值等于数组长度, 将引发数组索引越界异常 (程序清单同上)。

```
//访问数组元素指定的索引值等于数组长度, 所以下面代码将在运行时出现异常
System.out.println(objArr2[2]) ;
```



学生提问: 为什么要我记住这些异常信息?

答: 编写一个程序, 并不是单单指在电脑里敲出这些代码, 还包括调试这个程序, 使之可以正常运行。没有任何人可以保证自己写的程序总是正确的, 因此调试程序是写程序的重要组成部分, 调试程序的工作量往往超过编写代码的工作量。如何根据错误提示信息, 准确定位错误位置, 并排除程序错误是程序员的基本功。培养这些基本功需要记住常见的异常信息, 以及对应的出错原因。



所有的数组都提供了一个 `length` 属性, 通过这个属性可以访问到数组的长度, 一旦获得了数组的长度, 就可以通过循环来遍历该数组的每个数组元素。下面代码示范了输出 `prices` 数组 (动态初始化的 `int[]` 数组) 的每个数组元素的值 (程序清单同上)。

```
//使用循环输出 prices 数组的每个数组元素的值
for (int i = 0; i < prices.length; i ++ )
{
    System.out.println(prices[i]);
}
```

执行上面代码将输出 5 个 0, 因为 `prices` 数组执行的是默认初始化, 数组元素是 `int` 类型, 系统为 `int` 类型的数组元素赋值为 0。

下面代码示范了为动态初始化的数组元素进行赋值, 并通过循环方式输出每个数组元素 (程序清单同上)。

```
//对动态初始化后的数组元素进行赋值
books[0] = "疯狂 Java 讲义";
books[1] = "轻量级 Java EE 企业应用实战";
//使用循环输出 books 数组的每个数组元素的值
for (int i = 0 ; i < books.length ; i++ )
{
    System.out.println(books[i]);
}
```

上面代码将先输出字符串“疯狂 Java 讲义”和“轻量级 Java EE 企业应用实战”, 然后输出两个 `null`, 因为 `books` 使用了动态初始化, 系统为所有数组元素都分配一个 `null` 作为初始值, 后来程序又为前两个元素赋值, 所以看到了这样的程序输出结果。

从上面代码中不难看出, 初始化一个数组后, 相当于同时初始化了多个相同类型的变量, 通过数组元素的索引就可以自由访问这些变量 (实际上都是数组元素)。使用数组元素与使用普通变量并没有什么不同, 一样可以对数组元素进行赋值, 或者取出数组元素的值。

4.5.5 foreach 循环

从 Java 5 之后, Java 提供了一种更简单的循环: `foreach` 循环, 这种循环遍历数组和集合 (关于集合的介绍请参考本书第 8 章) 更加简洁。使用 `foreach` 循环遍历数组和集合元素时, 无须获得数组和集合长度, 无须根据索引来访问数组元素和集合元素, `foreach` 循环自动遍历数组和集合的每个元素。

`foreach` 循环的语法格式如下:

```
for(type variableName : array | collection)
{
```

```
    //variableName 自动迭代访问每个元素...  
}
```

在上面语法格式在中, **type** 是数组元素或集合元素的类型, **variableName** 是一个形参名, **foreach** 循环将自动将数组元素、集合元素依次赋给该变量。下面程序示范了如何使用 **foreach** 循环来遍历数组元素。

程序清单: codes\04\4.5\ForeachTest.java

```
public class ForEachTest  
{  
    public static void main(String[] args)  
    {  
        String[] books = {"轻量级 Java EE 企业应用实战" ,  
                           "疯狂 Java 讲义",  
                           "疯狂 Android 讲义"};  
        //使用 foreach 循环来遍历数组元素  
        //其中 book 将会自动迭代每个数组元素  
        for (String book : books)  
        {  
            System.out.println(book);  
        }  
    }  
}
```

从上面程序可以看出, 使用 **foreach** 循环遍历数组元素时无须获得数组长度, 也无须根据索引来访问数组元素。**foreach** 循环和普通循环不同的是, 它无须循环条件, 无须循环迭代语句, 这些部分都由系统来完成, **foreach** 循环自动迭代数组的每个元素, 当每个元素都被迭代一次后, **foreach** 循环自动结束。

当使用 **foreach** 循环来迭代输出数组元素或集合元素时, 通常不要对循环变量进行赋值, 虽然这种赋值在语法上是允许的, 但没有太大的实际意义, 而且极易引起错误。例如下面程序:

程序清单: codes\04\4.5\ForeachErrorTest.java

```
public class ForEachErrorTest  
{  
    public static void main(String[] args)  
    {  
        String[] books = {"轻量级 Java EE 企业应用实战" ,  
                           "疯狂 Java 讲义",  
                           "疯狂 Android 讲义"};  
        //使用 foreach 循环来遍历数组元素, 其中 book 将会自动迭代每个数组元素  
        for (String book : books)  
        {  
            book = "疯狂 Ajax 讲义";  
            System.out.println(book);  
        }  
        System.out.println(books[0]);  
    }  
}
```

运行上面程序, 将看到如下运行结果:

```
疯狂 Ajax 讲义  
疯狂 Ajax 讲义  
疯狂 Ajax 讲义  
轻量级 Java EE 企业应用实战
```

从上面运行结果来看, 由于我们在 **foreach** 循环中对数组元素进行赋值, 结果导致不能正确遍历数组元素, 不能正确地取出每个数组元素的值。而且当再次访问第一个数组元素时, 发现数组元素的值依然没有改变。不难看出, 当使用 **foreach** 来迭代访问数组元素时, **foreach** 中的循环变量相当于一个临时变量, 系统会把数组元素依次赋给这个临时变量, 而这个临时变量并不是数组元素, 它只是保存了数组元素的值。因此, 如果希望改变数组元素的值, 则不能使用这种 **foreach** 循环。

使用 **foreach** 循环迭代数组元素时, 并不能改变数组元素的值, 因此不要对 **foreach**



的循环变量进行赋值。

4.6 深入数组

数组是一种引用数据类型，数组引用变量只是一个引用，数组元素和数组变量在内存里是分开存放的。下面将深入介绍数组在内存中的运行机制。

4.6.1 内存中的数组

数组引用变量只是一个引用，这个引用变量可以指向任何有效的内存，只有当该引用指向有效内存后，才可通过该数组变量来访问数组元素。

与所有引用变量相同的是，引用变量是访问真实对象的根本方式。也就是说，如果我们在程序中访问数组对象本身，则只能通过这个数组的引用变量来访问它。

实际的数组对象被存储在堆(heap)内存中；如果引用该数组对象的数组引用变量是一个局部变量，那么它被存储在栈(stack)内存中。数组在内存中的存储示意图如图4.2所示。

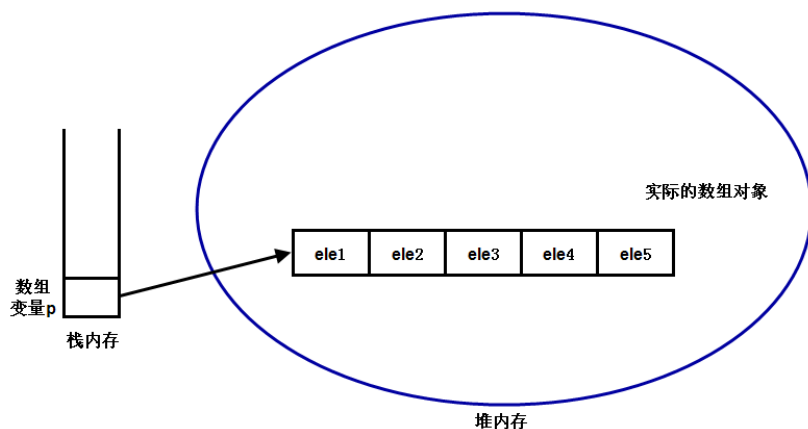


图 4.2 数组在内存中的存储示意图

如果需要访问图4.2所示堆内存中的数组元素，则程序中只能通过 `p[index]` 的形式实现。也就是说，数组引用变量是访问堆内存中数组元素的根本方式。

学生提问：为什么有栈内存和堆内存之分？

答：当一个方法执行时，每个方法都会建立自己的内存栈，在这个方法内定义的变量将会逐个放入这块栈内存里，随着方法的执行结束，这个方法的内存栈也将自然销毁。因此，所有在方法中定义的局部变量都是放在栈内存中的；当我们在程序中创建一个对象时，这个对象将被保存到运行时数据区中，以便反复利用（因为对象的创建成本通常较大），这个运行时数据区就是堆内存。堆内存中的对象不会随方法的结束而销毁，即使方法结束后，这个对象还可能被另一个引用变量所引用（在方法的参数传递时很常见），则这个对象依然不会被销毁。只有当一个对象没有任何引用变量引用它时，系统的垃圾回收器才会在合适的时候回收它。

如果堆内存中数组不再有任何引用变量指向自己，则这个数组将成为垃圾，该数组所占的内存将会被系统的垃圾回收机制回收。因此，为了让垃圾回收机制回收一个数组所占的内存空间，可以将该数组变量赋为 `null`，也就切断了数组引用变量和实际数组之间的引用关系，实际的数组也就成了垃圾。

只要类型相互兼容，就可以让一个数组变量指向另一个实际的数组，这种操作会让人产生数组的长

度可变的错觉。如下代码所示:

程序清单: codes\04\4.6\ArrayInRam.java

```
public class ArrayInRam
{
    public static void main(String[] args)
    {
        //定义并初始化数组,使用静态初始化
        int[] a = {5, 7, 20};
        //定义并初始化数组,使用动态初始化
        int[] b = new int[4];
        //输出b数组的长度
        System.out.println("b数组的长度为: " + b.length);
        //循环输出a数组的元素
        for (int i = 0, len = a.length; i < len; i++)
        {
            System.out.println(a[i]);
        }
        //循环输出b数组的元素
        for (int i = 0, len = b.length; i < len; i++)
        {
            System.out.println(b[i]);
        }
        //因为a是int[]类型,b也是int[]类型,所以可以将a的值赋给b。
        //也就是让b引用指向a引用指向的数组
        b = a;
        //再次输出b数组的长度
        System.out.println("b数组的长度为: " + b.length);
    }
}
```

运行上面代码后,将可以看到先输出b数组的长度为4,然后依次输出a数组和b数组的每个数组元素,接着会输出b数组的长度为3。看起来似乎数组的长度是可变的,但这只是一个假象。必须牢记:定义并初始化一个数组后,在内存中分配了两个空间,一个用于存放数组的引用变量,另一个用于存放数组本身。下面将结合示意图来说明上面程序的运行过程。

当程序定义并初始化了a、b两个数组后,系统内存中实际上产生了4块内存区,其中栈内存中有两个引用变量:a和b;堆内存中也有两块内存区,分别用于存储a和b引用所指向的数组本身。此时计算机内存的存储示意图如图4.3所示。

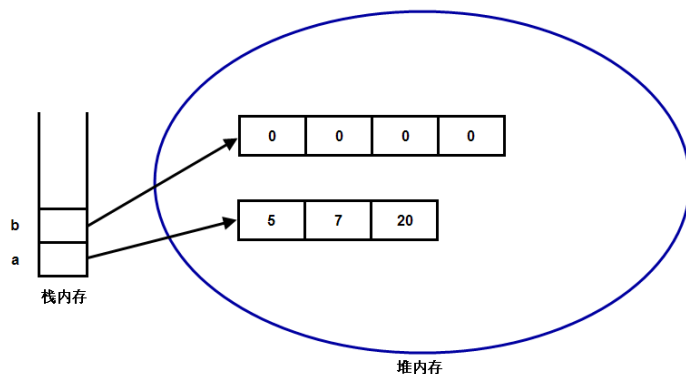


图 4.3 定义并初始化 a、b 两个数组后的存储示意图

从图4.3中可以非常清楚地看出a引用和b引用各自所引用的数组对象,并可以很清楚地看出a变量所引用的数组长度是3,b变量所引用的数组长度是4。

当执行上面的粗体字标识代码**b = a**时,系统将会把a的值赋给b,a和b都是引用类型变量,存储的是地址。因此把a的值赋给b后,就是让b指向a所指向的地址。此时计算机内存的存储示意图如图4.4所示。

从图4.4中可以看出,当执行了**b = a**之后,堆内存中的第一个数组具有了两个引用:a变量和b

变量都引用了第一个数组。此时第二个数组失去了引用，变成垃圾，只有等待垃圾回收机制来回收它——但它的长度依然不会改变，直到它彻底消失。



提示：

程序员进行程序开发时，不要仅仅停留在代码表面，而要深入底层的运行机制，才可以对程序的运行机制有更准确的把握。当我们看一个数组时，一定要把数组看成两个部分：一部分是数组引用，也就是在代码中定义的数组引用变量，还有一部分是实际的数组对象，这部分是在堆内存里运行的，通常无法直接访问它，只能通过数组引用变量来访问。

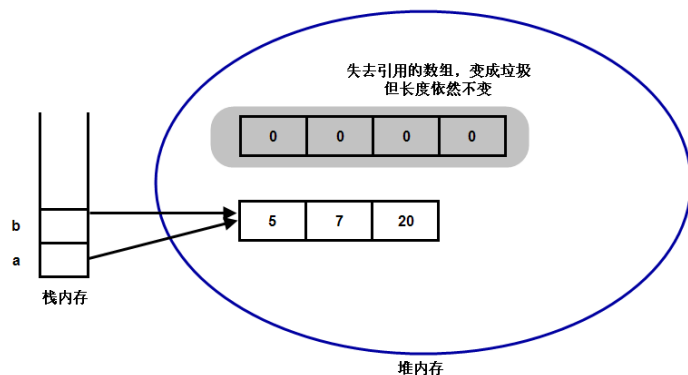


图 4.4 让 b 引用指向 a 引用所指向数组后的存储示意图

4.6.2 基本类型数组的初始化

对于基本类型数组而言，数组元素的值直接存储在对应的数组元素中，因此，初始化数组时，先为该数组分配内存空间，然后将数组元素的值存入对应数组元素中。

下面程序定义了一个 `int[]` 类型的数组变量，采用动态初始化的方式初始化了该数组，并显式为每个数组元素赋值。

程序清单：codes\04\4.6\PrimitiveArrayTest.java

```
public class PrimitiveArrayTest
{
    public static void main(String[] args)
    {
        // 定义一个 int[] 类型的数组变量
        int[] iArr;
        // 动态初始化数组，数组长度为 5
        iArr = new int[5];
        // 采用循环方式为每个数组元素赋值
        for (int i = 0; i < iArr.length; i++)
        {
            iArr[i] = i + 10;
        }
    }
}
```

上面代码的执行过程代表了基本类型数组初始化的典型过程。下面将结合示意图详细介绍这段代码的执行过程。

执行第一行代码 `int[] iArr;` 时，仅定义一个数组变量，此时内存中的存储示意图如图 4.5 所示。

执行了 `int[] iArr;` 代码后，仅在栈内存中定义了一个空引用（就是 `iArr` 数组变量），这个引用并未指向任何有效的内存，当然无法指定数组的长度。

当执行 `iArr = new int[5];` 动态初始化后，系统将负责为该数组分配内存空间，并分配默认的初始值：所有数组元素都被赋值为 0，此时内存中的存储示意图如图 4.6 所示。

此时 `iArr` 数组的每个数组元素的值都是 0，当循环为该数组的每个数组元素依次赋值后，此时每个

数组元素的值都变成程序显式指定的值。显式指定每个数组元素值后的存储示意图如图 4.7 所示。

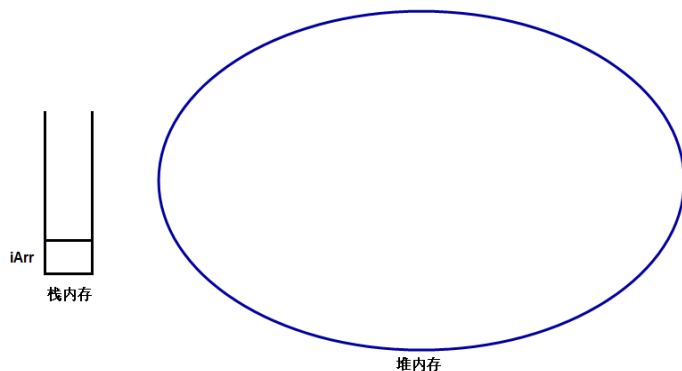


图 4.5 定义 iArr 数组变量后的存储示意图

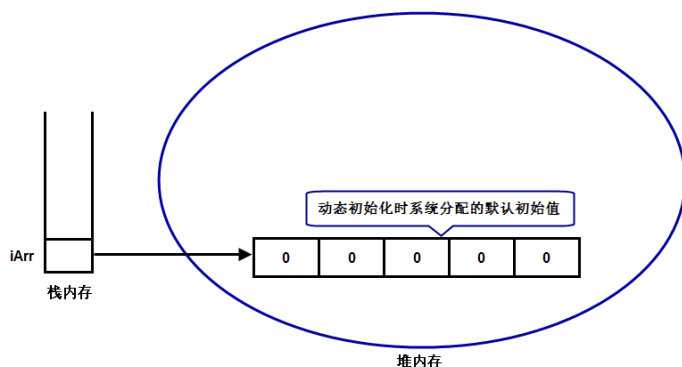


图 4.6 动态初始化 iArr 数组后的存储示意图

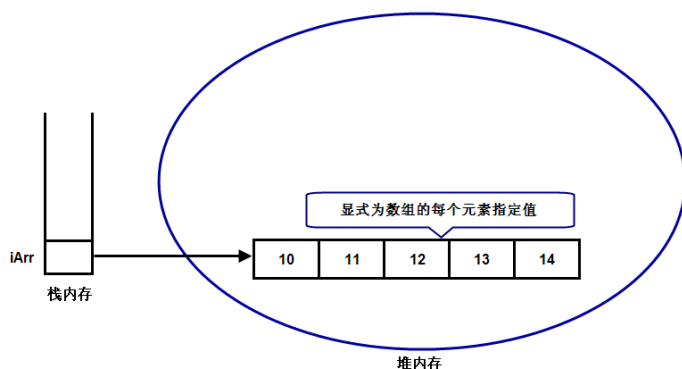


图 4.7 显式指定每个数组元素值后的存储示意图

从图 4.7 中可以看到基本类型数组的存储示意图，每个数组元素的值直接存储在对应的内存中。操作基本类型数组的数组元素时，实际上就是操作基本类型的变量。

4.6.3 引用类型数组的初始化

引用类型数组的数组元素是引用，因此情况变得更加复杂。每个数组元素里存储的还是引用，它指向另一块内存，这块内存里存储了有效数据。

为了更好地说明引用类型数组的运行过程，下面先定义一个 `Person` 类（所有类都是引用类型）。关于定义类、对象和引用的详细介绍请参考第 5 章。`Person` 类的代码如下：

程序清单：codes\04\4.6\ReferenceArrayTest.java

```
class Person
{
    //年龄
    public int age;
    //身高
```

```
public double height;
//定义一个 info 方法
public void info()
{
    System.out.println("我的年龄是: " + age
        + ", 我的身高是: " + height);
}
}
```

下面程序将定义一个 `Person[]` 数组，接着动态初始化这个 `Person[]` 数组，并为这个数组的每个数组元素指定值。程序代码如下（程序清单同上）：

```
public class ReferenceArrayTest
{
    public static void main(String[] args)
    {
        //定义一个 students 数组变量，其类型是 Person[]
        Person[] students;
        //执行动态初始化
        students = new Person[2];
        //创建一个 Person 实例，并将这个 Person 实例赋给 zhang 变量
        Person zhang = new Person();
        //为 zhang 所引用的 Person 对象的 age、height 赋值
        zhang.age = 15;
        zhang.height = 158;
        //创建一个 Person 实例，并将这个 Person 实例赋给 lee 变量
        Person lee = new Person();
        //为 lee 所引用的 Person 对象的 age、height 赋值
        lee.age = 16;
        lee.height = 161;
        //将 zhang 变量的值赋给第一个数组元素
        students[0] = zhang;
        //将 lee 变量的值赋给第二个数组元素
        students[1] = lee;
        //下面两行代码的结果完全一样，因为 lee
        //和 students[1] 指向的是同一个 Person 实例
        lee.info();
        students[1].info();
    }
}
```

上面代码的执行过程代表了引用类型数组初始化的典型过程。下面将结合示意图详细介绍这段代码的执行过程。

执行 `Person[] students;` 代码时，这行代码仅仅在栈内存中定义了一个引用变量，也就是一个指针，这个指针并未指向任何有效的内存区。此时内存中存储示意图如图 4.8 所示。

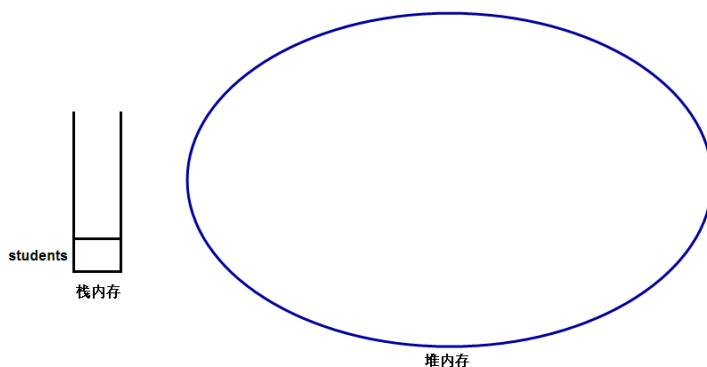


图 4.8 定义一个 `students` 数组变量后的存储示意图

在图 4.8 所示的栈内存中定义了一个 `students` 变量，它仅仅是一个引用，并未指向任何有效的内存。直到执行初始化，本程序对 `students` 数组执行动态初始化，动态初始化由系统为数组元素分配默认的初始值：`null`，即每个数组元素的值都是 `null`。执行动态初始化后的存储示意图如图 4.9 所示。

从图 4.9 中可以看出，`students` 数组的两个数组元素都是引用，而且这个引用并未指向任何有效的

内存，因此每个数组元素的值都是 `null`。这意味着依然不能直接使用 `students` 数组元素，因为每个数组元素都是 `null`，这相当于定义了两个连续的 `Person` 变量，但这个变量还未指向任何有效的内存区，所以这两个连续的 `Person` 变量（`students` 数组的数组元素）还不能使用。

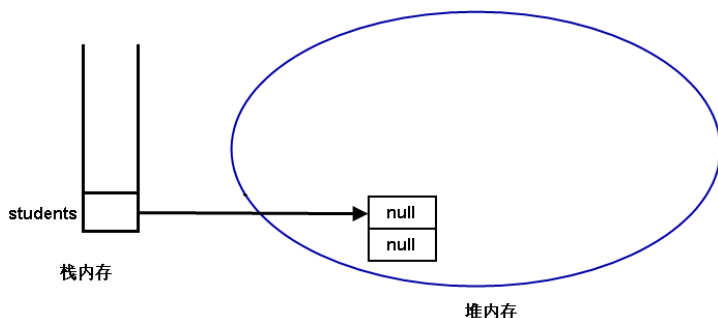


图 4.9 动态初始化 `students` 数组后的存储示意图

接着的代码定义了 `zhang` 和 `lee` 两个 `Person` 实例，定义这两个实例实际上分配了 4 块内存，在栈内存中存储了 `zhang` 和 `lee` 两个引用变量，还在堆内存中存储了两个 `Person` 实例。此时的内存存储示意图如图 4.10 所示。

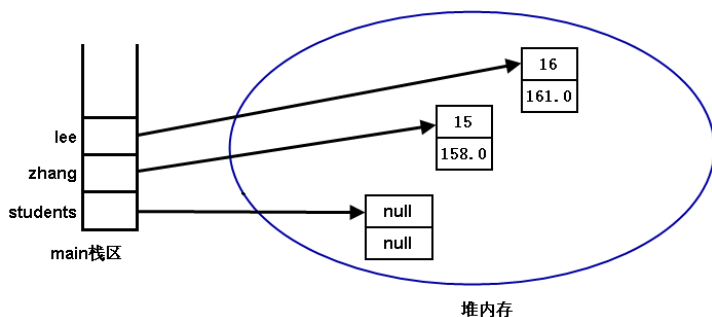


图 4.10 创建两个 `Person` 实例后的存储示意图

此时 `students` 数组的两个数组元素依然是 `null`，直到程序依次将 `zhang` 赋给 `students` 数组的第一个元素，把 `lee` 赋给 `students` 数组的第二个元素，`students` 数组的两个数组元素将会指向有效的内存区。此时的内存存储示意图如图 4.11 所示。

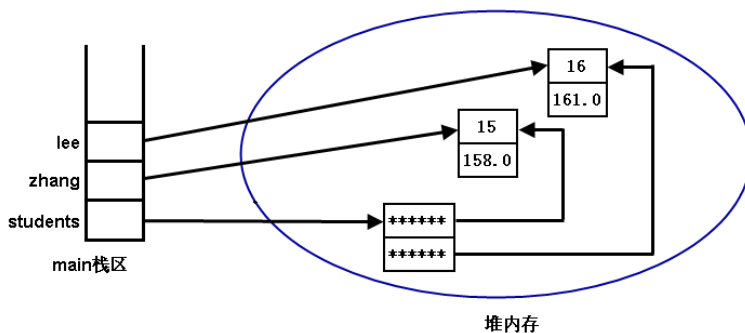


图 4.11 为数组元素赋值后的存储示意图

从图 4.11 中可以看出，此时 `zhang` 和 `students[0]` 指向同一个内存区，而且它们都是引用类型变量，因此通过 `zhang` 和 `students[0]` 来访问 `Person` 实例的 `Field` 和方法的效果完全一样，不论修改 `students[0]` 所指向的 `Person` 实例的 `Field`，还是修改 `zhang` 变量所指向的 `Person` 实例的 `Field`，所修改的其实是同一个内存区，所以必然互相影响。同理，`lee` 和 `students[1]` 也是引用同一个 `Person` 对象，也具有相同的效果。

4.6.4 没有多维数组

Java 语言里提供了支持多维数组的语法。但笔者还是想说，没有多维数组——如果从数组底层的运

行机制上来看。

Java 语言里的数组类型是引用类型，因此，数组变量其实是一个引用，这个引用指向真实的数组内存。数组元素的类型也可以是引用，如果数组元素的引用再次指向真实的数组内存，这种情形看上去很像多维数组。

回到前面定义数组类型的语法：`type[] arrName;`，这是典型的一维数组的定义语法，其中 `type` 是数组元素的类型。如果希望数组元素也是一个引用，而且是指向 `int` 数组的引用，则可以把 `type` 具体成 `int[]`（前面已经指出，`int[]`就是一种类型，`int[]`类型的用法与普通类型并无任何区别），那么上面定义数组的语法就是 `int[][] arrName`。

如果把 `int` 这个类型扩大到 Java 的所有类型（不包括数组类型），则出现了定义二维数组的语法：

```
type[][] arrName;
```

Java 语言采用上面的语法格式来定义二维数组，但它的实质还是一维数组，只是其数组元素也是引用，数组元素里保存的引用指向一维数组。

接着对这个“二维数组”执行初始化，同样可以把这个数组当成一维数组来初始化，把这个“二维数组”当成一个一维数组，其元素的类型是 `type[]` 类型，则可以采用如下语法进行初始化：

```
arrName = new type[length][]
```

上面的初始化语法相当于初始化了一个一维数组，这个一维数组的长度是 `length`。同样，因为这个一维数组的数组元素是引用类型（数组类型）的，所以系统为每个数组元素都分配初始值：`null`。

这个二维数组实际上完全可以当成一维数组使用：使用 `new type[length]` 初始化一维数组后，相当于定义了 `length` 个 `type` 类型的变量；类似的，使用 `new type[length][]` 初始化这个数组后，相当于定义了 `length` 个 `type[]` 类型的变量，当然 这些 `type[]` 类型的变量都是数组类型，因此必须再次初始化这些数组。

下面程序示范了如何把二维数组当成一维数组处理。

程序清单：codes\04\4.6\TwoDimensionTest.java

```
public class TwoDimensionTest
{
    public static void main(String[] args)
    {
        //定义一个二维数组
        int[][] a;
        //把 a 当成一维数组进行初始化，初始化 a 是一个长度为 4 的数组
        //a 数组的数组元素又是引用类型
        a = new int[4][];
        //把 a 数组当成一维数组，遍历 a 数组的每个数组元素
        for (int i = 0 , len = a.length; i < len ; i++ )
        {
            System.out.println(a[i]);
        }
        //初始化 a 数组的第一个元素
        a[0] = new int[2];
        //访问 a 数组的第一个元素所指数组的第二个元素
        a[0][1] = 6;
        //a 数组的第一个元素是一个一维数组，遍历这个一维数组
        for (int i = 0 , len = a[0].length ; i < len ; i ++ )
        {
            System.out.println(a[0][i]);
        }
    }
}
```

上面程序中粗体字标识部分把 `a` 这个二维数组当成一维数组处理，只是每个数组元素都是 `null`，所以我们看到输出结果都是 `null`。下面结合示意图来说明这个程序的执行过程。

程序的第一行 `int[][] a;`，将在栈内存中定义一个引用变量，这个变量并未指向任何有效的内存空间，此时的堆内存中还未为这行代码分配任何存储区。

程序对 `a` 数组执行初始化：`a = new int[4][];`，这行代码让 `a` 变量指向一块长度为 4 的数组内存，这

个长度为4的数组里每个数组元素都是引用类型（数组类型），系统为这些数组元素分配默认的初始值：`null`。此时 `a` 数组在内存中的存储示意图如图 4.12 所示。

从图 4.12 来看，虽然声明 `a` 是一个二维数组，但这里丝毫看不出它是一个二维数组的样子，完全是一维数组的样子。这个一维数组的长度是4，只是这4个数组元素都是引用类型，它们的默认值是 `null`。所以程序中可以把 `a` 数组当成一维数组处理，依次遍历 `a` 数组的每个元素，将看到每个数组元素的值都是 `null`。

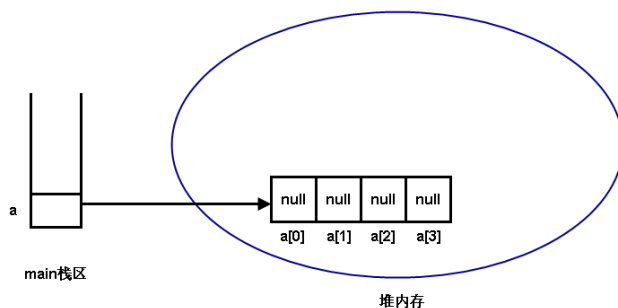


图 4.12 将二维数组当成一维数组初始化的存储示意图

因为 `a` 数组的元素必须是 `int[]` 数组，所以接下来的程序对 `a[0]` 元素执行初始化，也就是让图 4.12 右边堆内存中的第一个数组元素指向一个有效的数组内存，指向一个长度为2的 `int` 数组。因为程序采用动态初始化 `a[0]` 数组，因此系统将为 `a[0]` 所引用数组的每个元素分配默认的初始值：0，然后程序显式为 `a[0]` 数组的第二个元素赋值为6。此时在内存中的存储示意图如图 4.13 所示。

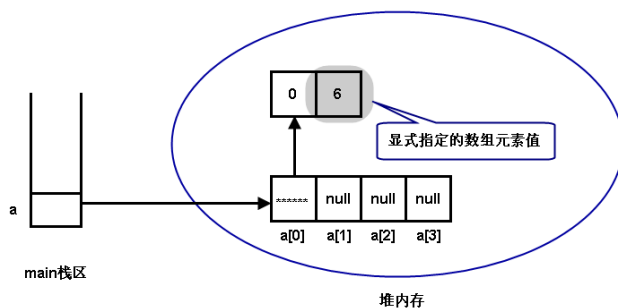


图 4.13 初始化 `a[0]` 后的存储示意图

图 4.13 中灰色覆盖的数组元素就是程序显式指定的数组元素值。`TwoDimensionTest.java` 接着迭代输出 `a[0]` 数组的每个数组元素，将看到输出 0 和 6。

学生提问：我是否可以图 4.13 中灰色覆盖的数组元素再次指向另一个数组？这样不就可以扩展成三维数组吗？甚至扩展成更多维的数组？

答：不能！至少在这个程序中不能。因为 Java 是强类型语言，当我们定义 `a` 数组时，已经确定了 `a` 数组的数组元素是 `int[]` 类型，则 `a[0]` 数组的数组元素只能是 `int` 类型，所以灰色覆盖的数组元素只能存储 `int` 类型的变量。对于其他弱类型语言，例如 JavaScript 和 Ruby 等，确实可以把一维数组无限扩展，扩展成二维数组、三维数组……如果想在 Java 语言中实现这种可无限扩展的数组，则可以定义一个 `Object[]` 类型的数组，这个数组的元素是 `Object` 类型，因此可以再次指向一个 `Object[]` 类型的数组，这样就可以从一维数组扩展到二维数组、三维数组……



从上面程序中可以看出，初始化多维数组时，可以只指定最左边维的大小；当然，也可以一次指定每一维的大小。例如下面代码（程序清单同上）：

```
//同时初始化二维数组的两个维数
int[][] b = new int[3][4];
```

上面代码将定义一个 **b** 数组变量，这个数组变量指向一个长度为 3 的数组，这个数组的每个数组元素又是一个数组类型，它们各指向对应的长度为 4 的 `int[]` 数组，每个数组元素的值为 0。这行代码执行后在内存中的存储示意图如图 4.14 所示。

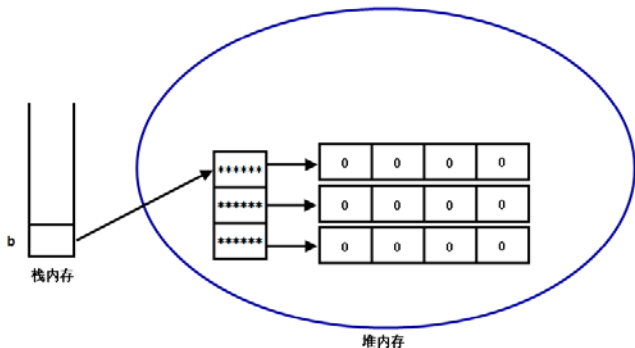


图 4.14 同时初始化二维数组的两个维数后的存储示意图

还可以使用静态初始化方式来初始化二维数组。使用静态初始化方式来初始化二维数组时，二维数组的每个数组元素都是一维数组，因此必须指定多个一维数组作为二维数组的初始化值。如下代码所示（程序清单同上）：

```
//使用静态初始化语法来初始化一个二维数组
String[][] str1 = new String[][]{new String[3]
    , new String[]{"hello"}};
//使用简化的静态初始化语法来初始化二维数组
String[][] str2 = {new String[3]
    , new String[]{"hello"}};
```

上面代码执行后内存中的存储示意图如图 4.15 所示。

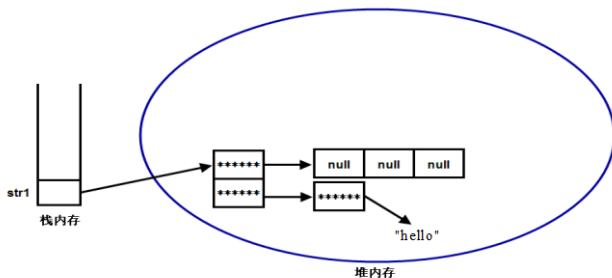


图 4.15 采用静态初始化语法初始化二维数组的存储示意图

通过上面讲解，我们可以得到一个结论：二维数组是一维数组，其数组元素是一维数组；三维数组也是一维数组，其数组元素是二维数组；四维数组还是一维数组，其数组元素是三维数组……从这个角度来看，Java 语言里没有多维数组。

4.6.5 操作数组的工具类

Java 提供的 `Arrays` 类里包含的一些 `static` 修饰的方法可以直接操作数组，这个 `Arrays` 类里包含了如下几个 `static` 修饰的方法（`static` 修饰的方法可以直接通过类名调用）。

- `int binarySearch(type[] a, type key)`: 使用二分法查询 `key` 元素值在 `a` 数组中出现的索引；如果 `a` 数组不包含 `key` 元素值，则返回负数。调用该方法时要求数组中元素已经按升序排列，这样才能得到正确结果。
- `int binarySearch(type[] a, int fromIndex, int toIndex, type key)`: 这个方法与前一个方法类似，但它

只搜索 a 数组中 fromIndex 到 toIndex 索引的元素。调用该方法时要求数组中元素已经按升序排列，这样才能得到正确结果。

- `type[] copyOf(type[] original, int newLength)`: 这个方法将会把 original 数组复制成一个新数组，其中 length 是新数组的长度。如果 length 小于 original 数组的长度，则新数组就是原数组的前面 length 个元素；如果 length 大于 original 数组的长度，则新数组的前面元素就是原数组的所有元素，后面补充 0（数值类型）、false（布尔类型）或者 null（引用类型）。
- `type[] copyOfRange(type[] original, int from, int to)`: 这个方法与前面方法相似，但这个方法只复制 original 数组的 from 索引到 to 索引的元素。
- `boolean equals(type[] a, type[] a2)`: 如果 a 数组和 a2 数组的长度相等，而且 a 数组和 a2 数组的数组元素也一一相同，该方法将返回 true。
- `void fill(type[] a, type val)`: 该方法将会把 a 数组的所有元素都赋值为 val。
- `void fill(type[] a, int fromIndex, int toIndex, type val)`: 该方法与前一个方法的作用相同，区别只是该方法仅仅将 a 数组的 fromIndex 到 toIndex 索引的数组元素赋值为 val。
- `void sort(type[] a)`: 该方法对 a 数组的数组元素进行排序。
- `void sort(type[] a, int fromIndex, int toIndex)`: 该方法与前一个方法相似，区别是该方法仅仅对 fromIndex 到 toIndex 索引的元素进行排序。
- `String toString(type[] a)`: 该方法将一个数组转换成一个字符串。该方法按顺序把多个数组元素连缀在一起，多个数组元素使用英文逗号 (,) 和空格隔开。

下面程序示范了 Arrays 类的用法。

程序清单: codes\04\4.6\ArraysTest.java

```
public class ArraysTest
{
    public static void main(String[] args)
    {
        //定义一个 a 数组
        int[] a = new int[]{3, 4, 5, 6};
        //定义一个 a2 数组
        int[] a2 = new int[]{3, 4, 5, 6};
        //a 数组和 a2 数组的长度相等，每个元素依次相等，将输出 true
        System.out.println("a 数组和 a2 数组是否相等: "
            + Arrays.equals(a, a2));
        //通过复制 a 数组，生成一个新的 b 数组
        int[] b = Arrays.copyOf(a, 6);
        System.out.println("a 数组和 b 数组是否相等: "
            + Arrays.equals(a, b));
        //输出 b 数组的元素，将输出[3, 4, 5, 6, 0, 0]
        System.out.println("b 数组的元素为: "
            + Arrays.toString(b));
        //将 b 数组的第 3 个元素（包括）到第 5 个元素（不包括）赋值为 1
        Arrays.fill(b, 2, 4, 1);
        //输出 b 数组的元素，将输出[3, 4, 1, 1, 0, 0]
        System.out.println("b 数组的元素为: "
            + Arrays.toString(b));
        //对 b 数组进行排序
        Arrays.sort(b);
        //输出 b 数组的元素，将输出[0, 0, 1, 1, 3, 4]
        System.out.println("b 数组的元素为: "
            + Arrays.toString(b));
    }
}
```

Arrays 类处于 java.util 包下，为了在程序中使用 Arrays 类，必须在程序中导入 java.util.Arrays 类。关于如何导入指定包下的类，请参考本书第 5 章。为了篇幅考虑，本



书中的程序代码都没有包含 import 语句，读者可参考光盘里对应程序来阅读书中代码。

除此之外，在 System 类里也包含了一个 static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)方法，该方法可以将 src 数组里的元素值赋给 dest 数组的元素，其中 srcPos 指定从 src 数组的第几个元素开始赋值，length 参数指定将 src 数组的多少个元素值赋给 dest 数组的元素。

4.6.6 数组的应用举例

数组的用途是很广泛的，如果程序中有多个类型相同的变量，而且它们具有逻辑的整体性，则可以把它们定义成一个数组。

例如，在实际开发中的一个常用工具函数：需要将一个浮点数转换成人民币读法字符串，这个程序就需要使用数组。实现这个函数的思路是，首先把这个浮点数分成整数部分和小数部分。提取整数部分很容易，直接将这个浮点数强制类型转换成一个整数即可，这个整数就是浮点数的整数部分；再使用浮点数减去整数将可以得到这个浮点数的小数部分。

然后分开处理整数部分和小数部分，其中小数部分的处理比较简单，直接截断到保留 2 位数字，转换成几角几分的字符串。整数部分的处理则稍微复杂一点，但只要我们认真分析不难发现，中国的数字习惯是 4 位一节的，一个 4 位的数字可被转成几千几百几十几，至于后面添加什么单位则不确定，如果这节 4 位数字出现在 1~4 位，则后面添加单位元；如果这节 4 位数字出现在 5~8 位，则后面添加单位万；如果这节 4 位数字出现在 9~12 位，则后面添加单位亿；多于 12 位就暂不考虑了。

因此实现这个程序的关键就是把一个 4 位数字字符串转换成一个中文读法。下面程序把这个需求实现了一部分。

程序清单：codes\04\4.6\Num2Rmb.java

```
public class Num2Rmb
{
    private String[] hanArr = {"零", "壹", "贰", "叁", "肆",
        "伍", "陆", "柒", "捌", "玖"};
    private String[] unitArr = {"十", "百", "千"};
    /**
     * 把一个浮点数分解成整数部分和小数部分字符串
     * @param num 需要被分解的浮点数
     * @return 分解出来的整数部分和小数部分。第一个数组元素是整数部分，第二个数组元素是小数部分
     */
    private String[] divide(double num)
    {
        // 将一个浮点数强制类型转换为 long 型，即得到它的整数部分
        long zheng = (long)num;
        // 浮点数减去整数部分，得到小数部分，小数部分乘以 100 后再取整得到 2 位小数
        long xiao = Math.round((num - zheng) * 100);
        // 下面用了 2 种方法把整数转换为字符串
        return new String[]{zheng + "", String.valueOf(xiao)};
    }
    /**
     * 把一个四位的数字字符串变成汉字字符串
     * @param numStr 需要被转换的四位的数字字符串
     * @return 四位的数字字符串被转换成汉字字符串
     */
    private String toHanStr(String numStr)
    {
        String result = "";
        int numLen = numStr.length();
        // 依次遍历数字字符串的每一位数字
        for (int i = 0 ; i < numLen ; i++)
        {
            // 把 char 型数字转换成 int 型数字，因为它们的 ASCII 码值恰好相差 48
            // 因此把 char 型数字减去 48 得到 int 型数字，例如 '4' 被转换成 4
            int num = numStr.charAt(i) - 48;
            // 如果不是最后一位数字，而且数字不是零，则需要添加单位（千、百、十）
            if ( i != numLen - 1 && num != 0 )
            {
                result = result + hanArr[num] + unitArr[numLen - i - 1];
            }
            else
            {
                result = result + hanArr[num];
            }
        }
        return result;
    }
}
```

```

        {
            result += hanArr[num] + unitArr[numLen - 2 - i];
        }
        // 否则不要添加单位
        else
        {
            result += hanArr[num];
        }
    }
    return result;
}
}
public static void main(String[] args)
{
    Num2Rmb nr = new Num2Rmb();
    // 测试把一个浮点数分解成整数部分和小数部分
    System.out.println(Arrays.toString(nr.divide(236711125.123)));
    // 测试把一个四位的数字字符串变成汉字字符串
    System.out.println(nr.toHanStr("6109"));
}
}

```

运行上面程序，看到如下运行结果：

```

[236711125, 12]
陆仟壹佰零玖

```

从上面程序的运行结果来看，初步实现了所需功能，但这个程序并不是这么简单，对零的处理比较复杂。例如，有两个零连在一起时该如何处理呢？如果最高位是零如何处理呢？最低位是零又如何处理呢？因此这个程序还需要继续完善，希望读者能把这个程序写完。

除此之外，我们还可以利用二维数组来完成五子棋、连连看、俄罗斯方块、扫雷等常见小游戏。下面简单介绍利用二维数组实现五子棋。我们需要定义一个二维数组作为下棋的棋盘，每当一个棋手下一步棋后，也就是为二维数组的一个数组元素赋值。下面程序完成了这个程序的初步功能。

程序清单：codes\04\4.6\Gobang.java

```

public class Gobang
{
    // 定义一个二维数组来充当棋盘
    private String[][] board;
    // 定义棋盘的大小
    private static int BOARD_SIZE = 15;
    public void initBoard()
    {
        // 初始化棋盘数组
        board = new String[BOARD_SIZE][BOARD_SIZE];
        // 把每个元素赋为"十"，用于在控制台画出棋盘
        for (int i = 0 ; i < BOARD_SIZE ; i++)
        {
            for (int j = 0 ; j < BOARD_SIZE ; j++)
            {
                board[i][j] = "十";
            }
        }
    }
    // 在控制台输出棋盘的方法
    public void printBoard()
    {
        // 打印每个数组元素
        for (int i = 0 ; i < BOARD_SIZE ; i++)
        {
            for (int j = 0 ; j < BOARD_SIZE ; j++)
            {
                // 打印数组元素后不换行
                System.out.print(board[i][j]);
            }
            // 每打印完一行数组元素后输出一个换行符
            System.out.print("\n");
        }
    }
}

```

```

    }
    public static void main(String[] args) throws Exception
    {
        Gobang gb = new Gobang();
        gb.initBoard();
        gb.printBoard();
        //这是用于获取键盘输入的方法
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String inputStr = null;
        //br.readLine(): 每当在键盘上输入一行内容后按回车键, 刚输入的内容将被 br 读取到
        while ((inputStr = br.readLine()) != null)
        {
            //将用户输入的字符串以逗号 (,) 作为分隔符, 分隔成 2 个字符串
            String[] posStrArr = inputStr.split(",");
            //将 2 个字符串转换为用户下棋的坐标
            int xPos = Integer.parseInt(posStrArr[0]);
            int yPos = Integer.parseInt(posStrArr[1]);
            //把对应的数组元素赋为"●"。
            gb.board[yPos - 1][xPos - 1] = "●";
            /*
            电脑随机生成 2 个整数, 作为电脑下棋的坐标, 赋给 board 数组
            还涉及
            1. 坐标的有效性, 只能是数字, 不能超出棋盘范围
            2. 下的棋的点, 不能重复下棋
            3. 每次下棋后, 需要扫描谁赢了
            */
            gb.printBoard();
            System.out.println("请输入您下棋的坐标, 应以 x,y 的格式: ");
        }
    }
}

```

运行上面程序, 将看到如图 4.16 所示的界面。

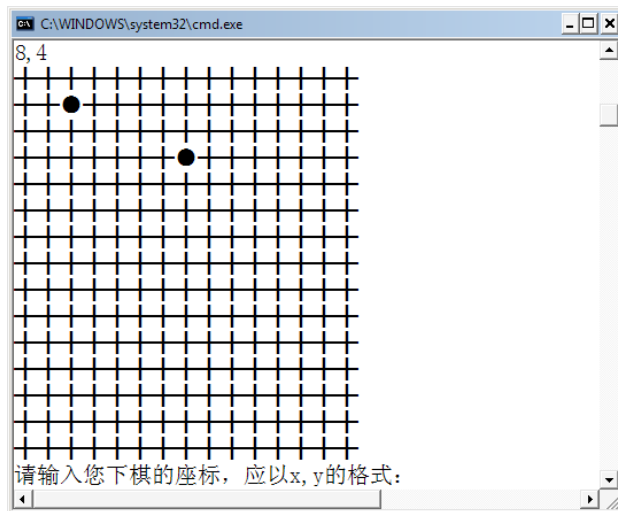


图 4.16 控制台五子棋的运行界面

从图 4.16 来看, 程序上面显示的黑点一直是棋手下的棋, 电脑还没有下棋, 电脑下棋可以使用随机生成两个坐标值来控制, 当然也可以增加人工智能 (但这已经超出了本书的范围, 实际上也很简单) 来控制下棋。



提示:

上面程序涉及读取用户键盘输入, 读者可以参考本书 7.1 节的介绍来阅读本程序。除此之外, 本程序中的 main 方法还包含了 throws Exception 声明, 表明该程序的 main 方法不处理任何异常。本书第 10 章才会介绍异常处理的知识, 所以此处不处理任何异常。

除此之外，读者还需要在这个程序的基础上进行完善，保证用户和电脑下的棋的坐标上不能已经有棋子（通过判断对应数组元素只能是“十”来确定），还需要进行 4 次循环扫描，判断横、竖、左斜、右斜是否有 5 个棋连在一起，从而判定胜负。

17.4 基于 UDP 协议的网络编程

UDP 协议是一种不可靠的网络协议，它在通信实例的两端各建立一个 Socket，但这两个 Socket 之间并没有虚拟链路，这两个 Socket 只是发送、接收数据报的对象。Java 提供了 DatagramSocket 对象作为基于 UDP 协议的 Socket，使用 DatagramPacket 代表 DatagramSocket 发送、接收的数据报。

17.4.1 UDP 协议基础

UDP 协议是英文 User Datagram Protocol 的缩写，即用户数据报协议，主要用来支持那些需要在计算机之间传输数据的网络连接。UDP 协议从问世至今已经被使用了很多年，虽然 UDP 协议目前应用不如 TCP 协议广泛，但 UDP 协议依然是一个非常实用和可行的网络传输层协议。尤其是在一些实时性很强的应用场景中，比如网络游戏、视频会议等，UDP 协议的快速更具有独特的魅力。

UDP 协议是一种面向非连接的协议，面向非连接指的是在正式通信前不必与对方先建立连接，不管对方状态就直接发送。至于对方是否可以接收到这些数据内容，UDP 协议无法控制，因此说 UDP 协议是一种不可靠的协议。UDP 协议适用于一次只传送少量数据、对可靠性要求不高的应用环境。

与前面介绍的 TCP 协议一样，UDP 协议直接位于 IP 协议之上。实际上，IP 协议属于 OSI 参考模型的网络层协议，而 UDP 协议和 TCP 协议都属于传输层协议。

因为 UDP 协议是面向非连接的协议，没有建立连接的过程，因此它的通信效率很高；但也正因为如此，它的可靠性不如 TCP 协议。

UDP 协议的主要作用是完成网络数据流和数据报之间的转换——在信息的发送端，UDP 协议将网络数据流封装成数据报，然后将数据报发送出去；在信息的接收端，UDP 协议将数据报转换成实际数据内容。



提示：

可以认为 UDP 协议的 Socket 类似于码头，数据报则类似于集装箱；码头的作用就是负责发送、接收集装箱，而 DatagramSocket 的作用则是发送、接收数据报。因此对于基于 UDP 协议的通信双方而言，没有所谓的客户端和服务端的概念。

UDP 协议和 TCP 协议简单对比如下。

- TCP 协议：可靠，传输大小无限制，但是需要连接建立时间，差错控制开销大。
- UDP 协议：不可靠，差错控制开销较小，传输大小限制在 64KB 以下，不需要建立连接。

17.4.2 使用 DatagramSocket 发送、接收数据

Java 使用 DatagramSocket 代表 UDP 协议的 Socket，DatagramSocket 本身只是码头，不维护状态，不能产生 IO 流，它的唯一作用就是接收和发送数据报，Java 使用 DatagramPacket 来代表数据报，DatagramSocket 接收和发送的数据都是通过 DatagramPacket 对象完成的。

先看一下 DatagramSocket 的构造器。

- DatagramSocket(): 创建一个 DatagramSocket 实例，并将该对象绑定到本机默认 IP 地址、本机所有可用端口中随机选择的某个端口。
- DatagramSocket(int port): 创建一个 DatagramSocket 实例，并将该对象绑定到本机默认 IP 地址、指定端口。
- DatagramSocket(int port, InetAddress laddr): 创建一个 DatagramSocket 实例，并将该对象绑定到指定 IP 地址、指定端口。

通过上面三个构造器中的任意一个构造器即可创建一个 DatagramSocket 实例，通常在创建服务器时，创建指定端口的 DatagramSocket 实例——这样保证其他客户端可以将数据发送到该服务器。一旦

得到了 DatagramSocket 实例之后, 就可以通过如下两个方法来接收和发送数据。

- receive(DatagramPacket p): 从该 DatagramSocket 中接收数据报。
- send(DatagramPacket p): 以该 DatagramSocket 对象向外发送数据报。

从上面两个方法可以看出, 使用 DatagramSocket 发送数据报时, DatagramSocket 并不知道将该数据报发送到哪里, 而是由 DatagramPacket 自身决定数据报的目的地。就像码头并不知道每个集装箱的目的地, 码头只是将这些集装箱发送出去, 而集装箱本身包含了该集装箱的目的地。

下面看一下 DatagramPacket 的构造器。

- DatagramPacket(byte[] buf, int length): 以一个空数组来创建 DatagramPacket 对象, 该对象的作用是接收 DatagramSocket 中的数据。
- DatagramPacket(byte[] buf, int length, InetAddress addr, int port): 以一个包含数据的数组来创建 DatagramPacket 对象, 创建该 DatagramPacket 对象时还指定了 IP 地址和端口——这就决定了该数据报的目的地。
- DatagramPacket(byte[] buf, int offset, int length): 以一个空数组来创建 DatagramPacket 对象, 并指定接收到的数据放入 buf 数组中时从 offset 开始, 最多放 length 个字节。
- DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port): 创建一个用于发送的 DatagramPacket 对象, 指定发送 buf 数组中从 offset 开始, 总共 length 个字节。



提示：

当 Client/Server 程序使用 UDP 协议时, 实际上并没有明显的服务器端和客户端, 因为两方都需要先建立一个 DatagramSocket 对象, 用来接收或发送数据报, 然后使用 DatagramPacket 对象作为传输数据的载体。通常固定 IP 地址、固定端口的 DatagramSocket 对象所在的程序被称为服务器, 因为该 DatagramSocket 可以主动接收客户端数据。

在接收数据之前, 应该采用上面的第一个或第三个构造器生成一个 DatagramPacket 对象, 给出接收数据的字节数组及其长度。然后调用 DatagramSocket 的 receive() 方法等待数据报的到来, receive() 将一直等待 (该方法会阻塞调用该方法的线程), 直到收到一个数据报为止。如下代码所示:

```
// 创建一个接收数据的 DatagramPacket 对象
DatagramPacket packet = new DatagramPacket(buf, 256);
// 接收数据报
socket.receive(packet);
```

在发送数据之前, 调用第二个或第四个构造器创建 DatagramPacket 对象, 此时的字节数组里存放了想发送的数据。除此之外, 还要给出完整的地址, 包括 IP 地址和端口号。发送数据是通过 DatagramSocket 的 send() 方法实现的, send() 方法根据数据报的地址来寻径以传送数据报。如下代码所示:

```
// 创建一个发送数据的 DatagramPacket 对象
DatagramPacket packet = new DatagramPacket(buf, length, address, port);
// 发送数据报
socket.send(packet);
```



提示：

使用 DatagramPacket 接收数据时, 会感觉 DatagramPacket 设计得过于烦琐。开发者只关心该 DatagramPacket 能放多少数据, 而 DatagramPacket 是否采用字节数组来存储数据完全不想关心。但 Java 要求创建接收数据用的 DatagramPacket 时, 必须传入一个空的字节数组, 该数组的长度决定了该 DatagramPacket 能放多少数据, 这实际上暴露了 DatagramPacket 的实现细节。接着 DatagramPacket 又提供了一个 getData() 方法, 该方法又可以返回 Datagram Packet 对象里封装的字节数组, 该方法更显得有些多余——如果程序需要获取

DatagramPacket 里封装的字节数组，直接访问传给 DatagramPacket 构造器的字节数组实参即可，无须调用该方法。

当服务器端（也可以是客户端）接收到一个 DatagramPacket 对象后，如果想向该数据报的发送者“反馈”一些信息，但由于 UDP 协议是面向非连接的，所以接收者并不知道每个数据报由谁发送过来，但程序可以调用 DatagramPacket 的如下 3 个方法来获取发送者的 IP 地址和端口。

- InetAddress getAddress(): 当程序准备发送此数据报时，该方法返回此数据报的目标机器的 IP 地址；当程序刚接收到一个数据报时，该方法返回该数据报的发送主机的 IP 地址。
- int getPort(): 当程序准备发送此数据报时，该方法返回此数据报的目标机器的端口；当程序刚接收到一个数据报时，该方法返回该数据报的发送主机的端口。
- SocketAddress getSocketAddress(): 当程序准备发送此数据报时，该方法返回此数据报的目标 SocketAddress；当程序刚接收到一个数据报时，该方法返回该数据报的发送主机的 SocketAddress。



提示：

getSocketAddress()方法的返回值是一个 SocketAddress 对象，该对象实际上就是一个 IP 地址和一个端口号。也就是说，SocketAddress 对象封装了一个 InetAddress 对象和一个代表端口的整数，所以使用 SocketAddress 对象可以同时代表 IP 地址和端口。

下面程序使用 DatagramSocket 实现了 Server/Client 结构的网络通信。本程序的服务器端使用循环 1000 次来读取 DatagramSocket 中的数据报，每当读取到内容之后便向该数据报的发送者送回一条信息。服务器端程序代码如下。

程序清单：codes\17\17.4\UdpServer.java

```
public class UdpServer
{
    public static final int PORT = 30000;
    // 定义每个数据报的最大大小为 4KB
    private static final int DATA_LEN = 4096;
    // 定义接收网络数据的字节数组
    byte[] inBuff = new byte[DATA_LEN];
    // 以指定字节数组创建准备接收数据的 DatagramPacket 对象
    private DatagramPacket inPacket =
        new DatagramPacket(inBuff, inBuff.length);
    // 定义一个用于发送的 DatagramPacket 对象
    private DatagramPacket outPacket;
    // 定义一个字符串数组，服务器端发送该数组的元素
    String[] books = new String[]
    {
        "疯狂 Java 讲义",
        "轻量级 Java EE 企业应用实战",
        "疯狂 Android 讲义",
        "疯狂 Ajax 讲义"
    };
    public void init()throws IOException
    {
        try(
            // 创建 DatagramSocket 对象
            DatagramSocket socket = new DatagramSocket(PORT))
        {
            // 采用循环接收数据
            for (int i = 0; i < 1000 ; i++ )
            {
                // 读取 Socket 中的数据，读到的数据放入 inPacket 封装的数组里
                socket.receive(inPacket);
                // 判断 inPacket.getData()和 inBuff 是否是同一个数组
                System.out.println(inBuff == inPacket.getData());
                // 将接收到的内容转换成字符串后输出
```

```

        System.out.println(new String(inBuff
            , 0 , inPacket.getLength()));
        // 从字符串数组中取出一个元素作为发送数据
        byte[] sendData = books[i % 4].getBytes();
        // 以指定的字节数组作为发送数据, 以刚接收到的 DatagramPacket 的
        // 源 SocketAddress 作为目标 SocketAddress 创建 DatagramPacket
        outPacket = new DatagramPacket(sendData
            , sendData.length , inPacket.getSocketAddress());
        // 发送数据
        socket.send(outPacket);
    }
}

public static void main(String[] args)
    throws IOException
{
    new UdpServer().init();
}
}

```

上面程序中的粗体字代码就是使用 DatagramSocket 发送、接收 DatagramPacket 的关键代码, 该程序可以接收 1000 个客户端发送过来的数据。

客户端程序代码也与此类似, 客户端采用循环不断地读取用户键盘输入, 每当读取到用户输入的内容后就将该内容封装成 DatagramPacket 数据报, 再将该数据报发送出去; 接着把 DatagramSocket 中的数据读入接收用的 DatagramPacket 中 (实际上是读入该 DatagramPacket 所封装的字节数组中)。客户端程序代码如下。

程序清单: codes\17\17.4\UdpClient.java

```

public class UdpClient
{
    // 定义发送数据报的目的地
    public static final int DEST_PORT = 30000;
    public static final String DEST_IP = "127.0.0.1";
    // 定义每个数据报的最大大小为 4KB
    private static final int DATA_LEN = 4096;
    // 定义接收网络数据的字节数组
    byte[] inBuff = new byte[DATA_LEN];
    // 以指定的字节数组创建准备接收数据的 DatagramPacket 对象
    private DatagramPacket inPacket =
        new DatagramPacket(inBuff , inBuff.length);
    // 定义一个用于发送的 DatagramPacket 对象
    private DatagramPacket outPacket = null;
    public void init()throws IOException
    {
        try(
            // 创建一个客户端 DatagramSocket, 使用随机端口
            DatagramSocket socket = new DatagramSocket())
        {
            // 初始化发送用的 DatagramSocket, 它包含一个长度为 0 的字节数组
            outPacket = new DatagramPacket(new byte[0] , 0
                , InetAddress.getByName(DEST_IP) , DEST_PORT);
            // 创建键盘输入流
            Scanner scan = new Scanner(System.in);
            // 不断地读取键盘输入
            while(scan.hasNextLine())
            {
                // 将键盘输入的一行字符串转换成字节数组
                byte[] buff = scan.nextLine().getBytes();
                // 设置发送用的 DatagramPacket 中的字节数据
                outPacket.setData(buff);
                // 发送数据报
                socket.send(outPacket);
                // 读取 Socket 中的数据, 读到的数据放在 inPacket 所封装的字节数组中
                socket.receive(inPacket);
                System.out.println(new String(inBuff , 0
                    , inPacket.getLength()));
            }
        }
    }
}

```

```
    }  
    }  
    }  
    public static void main(String[] args)  
        throws IOException  
    {  
        new UdpClient().init();  
    }  
}
```

上面程序中的粗体字代码同样也是使用 `DatagramSocket` 发送、接收 `DatagramPacket` 的关键代码，这些代码与服务器端代码基本相似。而客户端与服务器端的唯一区别在于：服务器端的 IP 地址、端口是固定的，所以客户端可以直接将该数据报发送给服务器端，而服务器端则需要根据接收到的数据报来决定“反馈”数据报的目的地。

读者可能会发现，使用 `DatagramSocket` 进行网络通信时，服务器端无须也无法保存每个客户端的状态，客户端把数据报发送到服务器端后，完全有可能立即退出。但不管客户端是否退出，服务器端都无法知道客户端的状态。

当使用 UDP 协议时，如果想让一个客户端发送的聊天信息被转发到其他所有的客户端则比较困难，可以考虑在服务器端使用 `Set` 集合来保存所有的客户端信息，每当接收到一个客户端的数据报之后，程序检查该数据报的源 `SocketAddress` 是否在 `Set` 集合中，如果不在就将该 `SocketAddress` 添加到该 `Set` 集合中。这样又涉及一个问题：可能有些客户端发送一个数据报之后永久性地退出了程序，但服务器端还将该客户端的 `SocketAddress` 保存在 `Set` 集合中……总之，这种方式需要处理的问题比较多，编程比较烦琐。幸好 Java 为 UDP 协议提供了 `MulticastSocket` 类，通过该类可以轻松地实现多点广播。

►► 17.4.3 使用 `MulticastSocket` 实现多点广播

`DatagramSocket` 只允许数据报发送给指定的目标地址，而 `MulticastSocket` 可以将数据报以广播方式发送到多个客户端。

若要使用多点广播，则需要让一个数据报标有一组目标主机地址，当数据报发出后，整个组的所有主机都能收到该数据报。IP 多点广播（或多点发送）实现了将单一信息发送到多个接收者的广播，其思想是设置一组特殊网络地址作为多点广播地址，每一个多点广播地址都被看做一个组，当客户端需要发送、接收广播信息时，加入到该组即可。

IP 协议为多点广播提供了这批特殊的 IP 地址，这些 IP 地址的范围是 224.0.0.0 至 239.255.255.255。多点广播示意图如图 17.8 所示。

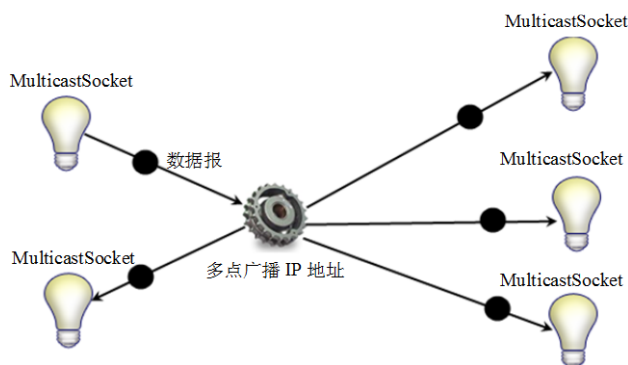


图 17.8 多点广播示意图

从图 17.8 中可以看出，`MulticastSocket` 类是实现多点广播的关键，当 `MulticastSocket` 把一个 `DatagramPacket` 发送到多点广播 IP 地址时，该数据报将被自动广播到加入该地址的所有 `MulticastSocket`。`MulticastSocket` 既可以将数据报发送到多点广播地址，也可以接收其他主机的广播信息。

`MulticastSocket` 有点像 `DatagramSocket`，事实上 `MulticastSocket` 是 `DatagramSocket` 的一个子类，也

就是说, `MulticastSocket` 是特殊的 `DatagramSocket`。当要发送一个数据报时, 可以使用随机端口创建 `MulticastSocket`, 也可以在指定端口创建 `MulticastSocket`。 `MulticastSocket` 提供了如下 3 个构造器。

- `public MulticastSocket()`: 使用本机默认地址、随机端口来创建 `MulticastSocket` 对象。
- `public MulticastSocket(int portNumber)`: 使用本机默认地址、指定端口来创建 `MulticastSocket` 对象。
- `public MulticastSocket(SocketAddress bindaddr)`: 使用本机指定 IP 地址、指定端口来创建 `MulticastSocket` 对象。

创建 `MulticastSocket` 对象后, 还需要将该 `MulticastSocket` 加入到指定的多点广播地址, `MulticastSocket` 使用 `joinGroup()` 方法加入指定组; 使用 `leaveGroup()` 方法脱离一个组。

- `joinGroup(InetAddress multicastAddr)`: 将该 `MulticastSocket` 加入指定的多点广播地址。
- `leaveGroup(InetAddress multicastAddr)`: 让该 `MulticastSocket` 离开指定的多点广播地址。

在某些系统中, 可能有多个网络接口。这可能会给多点广播带来问题, 这时候程序需要在一个指定的网络接口上监听, 通过调用 `setInterface()` 方法可以强制 `MulticastSocket` 使用指定的网络接口; 也可以使用 `getInterface()` 方法查询 `MulticastSocket` 监听的网络接口。



提示: 如果创建仅用于发送数据报的 `MulticastSocket` 对象, 则使用默认地址、随机端口即可。
但如果创建接收用的 `MulticastSocket` 对象, 则该 `MulticastSocket` 对象必须具有指定端口, 否则发送方无法确定发送数据报的目标端口。

`MulticastSocket` 用于发送、接收数据报的方法与 `DatagramSocket` 完全一样。但 `MulticastSocket` 比 `DatagramSocket` 多了一个 `setTimeToLive(int ttl)` 方法, 该 `ttl` 参数用于设置数据报最多可以跨过多少个网络, 当 `ttl` 的值为 0 时, 指定数据报应停留在本地主机; 当 `ttl` 的值为 1 时, 指定数据报发送到本地局域网; 当 `ttl` 的值为 32 时, 意味着只能发送到本站点的网络上; 当 `ttl` 的值为 64 时, 意味着数据报应保留在本地区; 当 `ttl` 的值为 128 时, 意味着数据报应保留在本大洲; 当 `ttl` 的值为 255 时, 意味着数据报可发送到所有地方; 在默认情况下, 该 `ttl` 的值为 1。

从图 17.8 中可以看出, 使用 `MulticastSocket` 进行多点广播时所有的通信实体都是平等的, 它们都将自己的数据报发送到多点广播 IP 地址, 并使用 `MulticastSocket` 接收其他人发送的广播数据报。下面程序使用 `MulticastSocket` 实现了一个基于广播的多人聊天室。程序只需要一个 `MulticastSocket`, 两个线程, 其中 `MulticastSocket` 既用于发送, 也用于接收; 一个线程负责接收用户键盘输入, 并向 `MulticastSocket` 发送数据, 另一个线程则负责从 `MulticastSocket` 中读取数据。

程序清单: codes\17\17.4\MulticastSocketTest.java

```
// 让该类实现 Runnable 接口, 该类的实例可作为线程的 target
public class MulticastSocketTest implements Runnable
{
    // 使用常量作为本程序的多点广播 IP 地址
    private static final String BROADCAST_IP
        = "230.0.0.1";
    // 使用常量作为本程序的多点广播目的地端口
    public static final int BROADCAST_PORT = 30000;
    // 定义每个数据报的最大大小为 4KB
    private static final int DATA_LEN = 4096;
    // 定义本程序的 MulticastSocket 实例
    private MulticastSocket socket = null;
    private InetAddress broadcastAddress = null;
    private Scanner scan = null;
    // 定义接收网络数据的字节数组
    byte[] inBuff = new byte[DATA_LEN];
    // 以指定字节数组创建准备接收数据的 DatagramPacket 对象
    private DatagramPacket inPacket
        = new DatagramPacket(inBuff, inBuff.length);
    // 定义一个用于发送的 DatagramPacket 对象
    private DatagramPacket outPacket = null;
```

```

public void init()throws IOException
{
    try(
        // 创建键盘输入流
        Scanner scan = new Scanner(System.in))
    {
        // 创建用于发送、接收数据的 MulticastSocket 对象
        // 由于该 MulticastSocket 对象需要接收数据，所以有指定端口
        socket = new MulticastSocket(BROADCAST_PORT);
        broadcastAddress = InetAddress.getByName(BROADCAST_IP);
        // 将该 socket 加入指定的多点广播地址
        socket.joinGroup(broadcastAddress);
        // 设置本 MulticastSocket 发送的数据报会被回送到自身
        socket.setLoopbackMode(false);
        // 初始化发送用的 DatagramSocket，它包含一个长度为 0 的字节数组
        outPacket = new DatagramPacket(new byte[0]
            , 0 , broadcastAddress , BROADCAST_PORT);
        // 启动以本实例的 run()方法作为线程执行体的线程
        new Thread(this).start();
        // 不断地读取键盘输入
        while(scan.hasNextLine())
        {
            // 将键盘输入的一行字符串转换成字节数组
            byte[] buff = scan.nextLine().getBytes();
            // 设置发送用的 DatagramPacket 里的字节数据
            outPacket.setData(buff);
            // 发送数据报
            socket.send(outPacket);
        }
    }
    finally
    {
        socket.close();
    }
}

public void run()
{
    try
    {
        while(true)
        {
            // 读取 Socket 中的数据，读到的数据放在 inPacket 所封装的字节数组里
            socket.receive(inPacket);
            // 打印输出从 socket 中读取的内容
            System.out.println("聊天信息: " + new String(inBuff
                , 0 , inPacket.getLength()));
        }
    }
    // 捕获异常
    catch (IOException ex)
    {
        ex.printStackTrace();
        try
        {
            {
                if (socket != null)
                {
                    // 让该 Socket 离开该多点 IP 广播地址
                    socket.leaveGroup(broadcastAddress);
                    // 关闭该 Socket 对象
                    socket.close();
                }
                System.exit(1);
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}

public static void main(String[] args)

```

```

        throws IOException
    {
        new MulticastSocketTest().init();
    }
}

```

上面程序中 `init()` 方法里的第一行粗体字代码先创建了一个 `MulticastSocket` 对象，由于需要使用该对象接收数据报，所以为该 `Socket` 对象设置使用固定端口；第二行粗体字代码将该 `Socket` 对象添加到指定的多点广播 IP 地址；第三行粗体字代码设置该 `Socket` 发送的数据报会被回送到自身（即该 `Socket` 可以接收到自己发送的数据报）。至于程序中使用 `MulticastSocket` 发送、接收数据报的代码，与使用 `DatagramSocket` 并没有区别，故此处不再赘述。

下面将结合 `MulticastSocket` 和 `DatagramSocket` 开发一个简单的局域网即时通信工具，局域网内每个用户启动该工具后，就可以看到该局域网内所有的在线用户，该用户也会被其他用户看到，即看到如图 17.9 所示的窗口。

在图 17.9 所示的用户列表中双击任意一个用户，即可启动一个如图 17.10 所示的交谈窗口。

如果双击图 17.10 所示用户列表窗口中的“所有人”列表项，即可启动一个与图 17.10 相似的交谈窗口，不同的是通过该窗口发送的消息将会被所有人看到。



图 17.9 局域网聊天工具

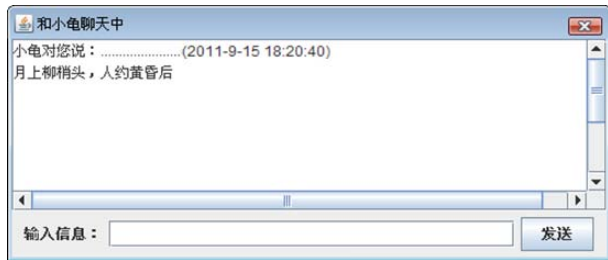


图 17.10 与特定用户交谈

该程序的实现思路是，每个用户都启动两个 `Socket`，即一个 `MulticastSocket`，一个 `DatagramSocket`。其中 `MulticastSocket` 会周期性地向 230.0.0.1 发送在线信息，且所有用户的 `MulticastSocket` 都会加入到 230.0.0.1 这个多点广播 IP 地址中，这样每个用户都可以收到其他用户广播的在线信息，如果系统经过一段时间没有收到某个用户广播的在线信息，则从用户列表中删除该用户。除此之外，该 `MulticastSocket` 还用于向所有用户发送广播信息。

`DatagramSocket` 主要用于发送私聊信息，当用户收到其他用户广播来的 `DatagramPacket` 时，即可获取该用户 `MulticastSocket` 对应的 `SocketAddress`，这个 `SocketAddress` 将作为发送私聊信息的重要依据——本程序让 `MulticastSocket` 在 30000 端口监听，而 `DatagramSocket` 在 30001 端口监听，这样程序就可以根据其他用户广播来的 `DatagramPacket` 得到他的 `DatagramSocket` 所在的地址。

本系统提供了一个 `UserInfo` 类，该类封装了用户名、图标、对应的 `SocketAddress` 以及该用户对应的交谈窗口、失去联系的次数等信息。该类的代码片段如下。

程序清单：codes\17\17.4\LanTalk\UserInfo.java

```

public class UserInfo
{
    // 该用户的图标
    private String icon;
    // 该用户的名字
    private String name;
    // 该用户的 MulticastSocket 所在的 IP 地址和端口
    private SocketAddress address;
}

```

```

// 该用户失去联系的次数
private int lost;
// 该用户对应的交谈窗口
private ChatFrame chatFrame;
public UserInfo(){}
// 有参数的构造器
public UserInfo(String icon , String name
    , SocketAddress address , int lost)
{
    this.icon = icon;
    this.name = name;
    this.address = address;
    this.lost = lost;
}
// 省略所有 field 的 setter 和 getter 方法
...
// 使用 address 作为该用户的标识, 所以根据 address
// 重写 hashCode() 和 equals() 方法
public int hashCode()
{
    return address.hashCode();
}
public boolean equals(Object obj)
{
    if (obj != null && obj.getClass() == UserInfo.class)
    {
        UserInfo target = (UserInfo)obj;
        if (address != null)
        {
            return address.equals(target.getAddress());
        }
    }
    return false;
}
}

```

通过 `UserInfo` 类的封装, 所有客户端只需要维护该 `UserInfo` 类的列表, 程序就可以实现广播、发送私聊信息等功能。本程序底层通信的工具类则需要一个 `MulticastSocket` 和一个 `DatagramSocket`, 该工具类的代码如下。

程序清单: codes\17\17.4\LanTalk\ComUtil.java

```

// 聊天交换信息的工具类
public class ComUtil
{
    // 使用常量作为本程序的多点广播 IP 地址
    private static final String BROADCAST_IP
        = "230.0.0.1";
    // 使用常量作为本程序的多点广播目的地端口
    // DatagramSocket 所用的端口为该端口号-1
    public static final int BROADCAST_PORT = 30000;
    // 定义每个数据报的最大大小为 4KB
    private static final int DATA_LEN = 4096;
    // 定义本程序的 MulticastSocket 实例
    private MulticastSocket socket = null;
    // 定义本程序私聊的 Socket 实例
    private DatagramSocket singleSocket = null;
    // 定义广播的 IP 地址
    private InetAddress broadcastAddress = null;
    // 定义接收网络数据的字节数组
    byte[] inBuff = new byte[DATA_LEN];
    // 以指定字节数组创建准备接收数据的 DatagramPacket 对象
    private DatagramPacket inPacket =
        new DatagramPacket(inBuff , inBuff.length);
    // 定义一个用于发送的 DatagramPacket 对象
    private DatagramPacket outPacket = null;
    // 聊天的主界面程序
    private LanTalk lanTalk;
    // 构造器, 初始化资源
}

```



```

public ComUtil(LanTalk lanTalk) throws Exception
{
    this.lanTalk = lanTalk;
    // 创建用于发送、接收数据的 MulticastSocket 对象
    // 因为该 MulticastSocket 对象需要接收数据，所以有指定端口
    socket = new MulticastSocket(BROADCAST_PORT);
    // 创建私聊用的 DatagramSocket 对象
    singleSocket = new DatagramSocket(BROADCAST_PORT + 1);
    broadcastAddress = InetAddress.getByName(BROADCAST_IP);
    // 将该 socket 加入指定的多点广播地址
    socket.joinGroup(broadcastAddress);
    // 设置本 MulticastSocket 发送的数据报被回送到自身
    socket.setLoopbackMode(false);
    // 初始化发送用的 DatagramSocket，它包含一个长度为 0 的字节数组
    outPacket = new DatagramPacket(new byte[0]
        , 0 , broadcastAddress , BROADCAST_PORT);
    // 启动两个读取网络数据的线程
    new ReadBroad().start();
    Thread.sleep(1);
    new ReadSingle().start();
}
// 广播消息的工具方法
public void broadCast(String msg)
{
    try
    {
        // 将 msg 字符串转换成字节数组
        byte[] buff = msg.getBytes();
        // 设置发送用的 DatagramPacket 里的字节数据
        outPacket.setData(buff);
        // 发送数据报
        socket.send(outPacket);
    }
    // 捕获异常
    catch (IOException ex)
    {
        ex.printStackTrace();
        if (socket != null)
        {
            // 关闭该 Socket 对象
            socket.close();
        }
        JOptionPane.showMessageDialog(null
            , "发送信息异常，请确认 30000 端口空闲，且网络连接正常！"
            , "网络异常" , JOptionPane.ERROR_MESSAGE);
        System.exit(1);
    }
}
// 定义向单独用户发送消息的方法
public void sendSingle(String msg , SocketAddress dest)
{
    try
    {
        // 将 msg 字符串转换成字节数组
        byte[] buff = msg.getBytes();
        DatagramPacket packet = new DatagramPacket(buff
            , buff.length , dest);
        singleSocket.send(packet);
    }
    // 捕获异常
    catch (IOException ex)
    {
        ex.printStackTrace();
        if (singleSocket != null)
        {
            // 关闭该 Socket 对象
            singleSocket.close();
        }
        JOptionPane.showMessageDialog(null
            , "发送信息异常，请确认 30001 端口空闲，且网络连接正常！"

```

```

        , "网络异常", JOptionPane.ERROR_MESSAGE);
        System.exit(1);
    }
}
// 不断地从 DatagramSocket 中读取数据的线程
class ReadSingle extends Thread
{
    // 定义接收网络数据的字节数组
    byte[] singleBuff = new byte[DATA_LEN];
    private DatagramPacket singlePacket =
        new DatagramPacket(singleBuff , singleBuff.length);
    public void run()
    {
        while (true)
        {
            try
            {
                // 读取 Socket 中的数据
                singleSocket.receive(singlePacket);
                // 处理读到的信息
                lanTalk.processMsg(singlePacket , true);
            }
            // 捕获异常
            catch (IOException ex)
            {
                ex.printStackTrace();
                if (singleSocket != null)
                {
                    // 关闭该 Socket 对象
                    singleSocket.close();
                }
                JOptionPane.showMessageDialog(null
                    , "接收信息异常, 请确认 30001 端口空闲, 且网络连接正常!"
                    , "网络异常", JOptionPane.ERROR_MESSAGE);
                System.exit(1);
            }
        }
    }
}
// 持续读取 MulticastSocket 的线程
class ReadBroad extends Thread
{
    public void run()
    {
        while (true)
        {
            try
            {
                // 读取 Socket 中的数据
                socket.receive(inPacket);
                // 打印输出从 Socket 中读取的内容
                String msg = new String(inBuff , 0
                    , inPacket.getLength());
                // 读到的内容是在线信息
                if (msg.startsWith(YeekuProtocol.PRESENCE)
                    && msg.endsWith(YeekuProtocol.PRESENCE))
                {
                    String userMsg = msg.substring(2
                        , msg.length() - 2);
                    String[] userInfo = userMsg.split(YeekuProtocol
                        .SPLITTER);
                    UserInfo user = new UserInfo(userInfo[1]
                        , userInfo[0] , inPacket.getSocketAddress(), 0);
                    // 控制是否需要添加该用户的旗标
                    boolean addFlag = true;
                    ArrayList<Integer> delList = new ArrayList<>();
                    // 遍历系统中已有的所有用户, 该循环必须循环完成
                    for (int i = 1 ; i < lanTalk.getUserNum() ; i++ )
                    {
                        UserInfo current = lanTalk.getUser(i);

```



```

private DefaultListModel<UserInfo> listModel
    = new DefaultListModel<>();
// 定义一个 JList 对象
private JList<UserInfo> friendsList = new JList<>(listModel);
// 定义一个用于格式化日期的格式器
private DateFormat formatter = DateFormat.getDateInstance();
public LanTalk()
{
    super("局域网聊天");
    // 设置该 JList 使用 ImageCellRenderer 作为单元格绘制器
    friendsList.setCellRenderer(new ImageCellRenderer());
    listModel.addElement(new UserInfo("all" , "所有人"
        , null , -2000));
    friendsList.addMouseListener(new ChangeMusicListener());
    add(new JScrollPane(friendsList));
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(2, 2, 160 , 600);
}
// 根据地址来查询用户
public UserInfo getUserBySocketAddress(SocketAddress address)
{
    for (int i = 1 ; i < getUserNum() ; i++)
    {
        UserInfo user = getUser(i);
        if (user.getAddress() != null
            && user.getAddress().equals(address))
        {
            return user;
        }
    }
    return null;
}
// -----下面四个方法是对 ListModel 的包装-----
// 向用户列表中添加用户
public void addUser(UserInfo user)
{
    listModel.addElement(user);
}
// 从用户列表中删除用户
public void removeUser(int pos)
{
    listModel.removeElementAt(pos);
}
// 获取该聊天窗口的用户数量
public int getUserNum()
{
    return listModel.size();
}
// 获取指定位置的用户
public UserInfo getUser(int pos)
{
    return listModel.elementAt(pos);
}
// 实现 JList 上的鼠标双击事件监听器
class ChangeMusicListener extends MouseAdapter
{
    public void mouseClicked(MouseEvent e)
    {
        // 如果鼠标的击键次数大于 2
        if (e.getClickCount() >= 2)
        {
            // 取出鼠标双击时选中的列表项
            UserInfo user = (UserInfo)friendsList.getSelectedValue();
            // 如果该列表项对应用户的交谈窗口为 null
            if (user.getChatFrame() == null)
            {
                // 为该用户创建一个交谈窗口, 并让用户引用该窗口
                user.setChatFrame(new ChatFrame(null , user));
            }
            // 如果该用户的窗口没有显示, 则让该用户的窗口显示出来

```

```

        if (!user.getChatFrame().isShowing())
        {
            user.getChatFrame().setVisible(true);
        }
    }
}
/**
 * 处理网络数据报, 该方法将根据聊天信息得到聊天者
 * 并将信息显示在聊天对话框中
 * @param packet 需要处理的数据报
 * @param single 该信息是否为私聊信息
 */
public void processMsg(DatagramPacket packet , boolean single)
{
    // 获取发送该数据报的 SocketAddress
    InetSocketAddress srcAddress = (InetSocketAddress)
        packet.getSocketAddress();
    // 如果是私聊信息, 则该 Packet 获取的是 DatagramSocket 的地址
    // 将端口号减 1 才是对应的 MulticastSocket 的地址
    if (single)
    {
        srcAddress = new InetSocketAddress(srcAddress.getHostName()
            , srcAddress.getPort() - 1);
    }
    UserInfo srcUser = getUserBySocketAddress(srcAddress);
    if (srcUser != null)
    {
        // 确定消息将要显示到哪个用户对应的窗口中
        UserInfo alertUser = single ? srcUser : getUser(0);
        // 如果该用户对应的窗口为空, 则显示该窗口
        if (alertUser.getChatFrame() == null)
        {
            alertUser.setChatFrame(new ChatFrame(null , alertUser));
        }
        // 定义添加的提示信息
        String tipMsg = single ? "对您说: " : "对大家说: ";
        // 显示提示信息
        alertUser.getChatFrame().addString(srcUser.getName()
            + tipMsg + ".....("
            + formatter.format(new Date()) + ")\n"
            + new String(packet.getData() , 0 , packet.getLength())
            + "\n");
        if (!alertUser.getChatFrame().isShowing())
        {
            alertUser.getChatFrame().setVisible(true);
        }
    }
}
// 主方法, 程序的入口
public static void main(String[] args)
{
    LanTalk lanTalk = new LanTalk();
    new LoginFrame(lanTalk , "请输入用户名、头像后登录");
}
// 定义用于改变 JList 列表项外观的类
class ImageCellRenderer extends JPanel
    implements ListCellRenderer<UserInfo>
{
    private ImageIcon icon;
    private String name;
    // 定义绘制单元格时的背景色
    private Color background;
    // 定义绘制单元格时的前景色
    private Color foreground;
    @Override
    public Component getListCellRendererComponent(JList list
        , UserInfo userInfo , int index
        , boolean isSelected , boolean cellHasFocus)

```

```

    {
        // 设置图标
        icon = new ImageIcon("ico/" + userInfo.getIcon() + ".gif");
        name = userInfo.getName();
        // 设置背景色、前景色
        background = isSelected ? list.getSelectionBackground()
            : list.getBackground();
        foreground = isSelected ? list.getSelectionForeground()
            : list.getForeground();
        // 返回该 JPanel 对象作为单元格绘制器
        return this;
    }
    // 重写 paintComponent 方法, 改变 JPanel 的外观
    public void paintComponent(Graphics g)
    {
        int imageWidth = icon.getImage().getWidth(null);
        int imageHeight = icon.getImage().getHeight(null);
        g.setColor(background);
        g.fillRect(0, 0, getWidth(), getHeight());
        g.setColor(foreground);
        // 绘制好友图标
        g.drawImage(icon.getImage(), getWidth() / 2 - imageWidth / 2,
            10, null);
        g.setFont(new Font("SansSerif", Font.BOLD, 18));
        // 绘制好友用户名
        g.drawString(name, getWidth() / 2 - name.length() * 10,
            imageHeight + 30);
    }
    // 通过该方法来设置该 ImageCellRenderer 的最佳大小
    public Dimension getPreferredSize()
    {
        return new Dimension(60, 80);
    }
}

```

上面类中提供的 `addUser()` 和 `removeUser()` 方法暴露给通信类 `ComUtil` 使用, 用于向用户列表中添加、删除用户。除此之外, 该类还提供了一个 `processMsg()` 方法, 该方法用于处理网络中读取的数据报, 将数据报中的内容取出, 并显示在特定的窗口中。



提示：

上面讲解的只是本程序的关键类, 本程序还涉及 `YeekuProtocol`、`ChatFrame`、`LoginFrame` 等类, 由于篇幅关系, 此处不再给出这些类的源代码, 读者可以参考 `.codes\17\17.4\LanTalk` 路径下的源代码。