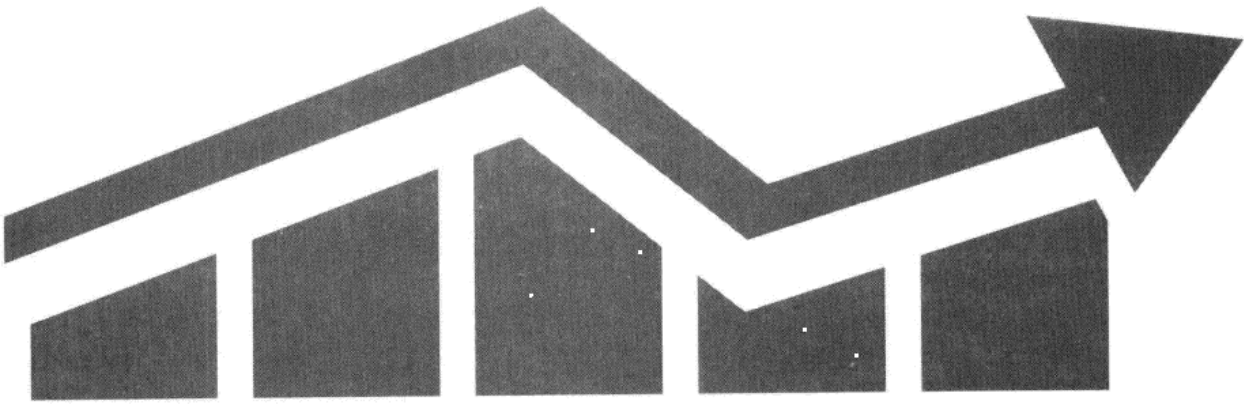
TURING 图灵程序设计丛书



算法 Algorithms Fourth Edition （第4版）

[美] Robert Sedgewick
Kevin Wayne 著

谢路云 译

人民邮电出版社
北京

图书在版编目（CIP）数据

算法 : 第4版 / (美) 塞奇威克 (Sedgewick, R.) , (美) 韦恩 (Wayne, K.) 著 ; 谢路云译. -- 北京 : 人民邮电出版社, 2012.10
(图灵程序设计丛书)
书名原文: Algorithms, Fourth Edition
ISBN 978-7-115-29380-0

Ⅰ. ①算… Ⅱ. ①塞… ②韦… ③谢… Ⅲ. ①电子计算机－算法理论 Ⅳ. ①TP301.6

中国版本图书馆CIP数据核字(2012)第220659号

内 容 提 要

本书作为算法领域经典的参考书，全面介绍了关于算法和数据结构的必备知识，并特别针对排序、搜索、图处理和字符串处理进行了论述。第4版具体给出了每位程序员应知应会的50个算法，提供了实际代码，而且这些Java代码实现采用了模块化的编程风格，读者可以方便地加以改造。配套网站提供了本书内容的摘要及更多的代码实现、测试数据、练习、教学课件等资源。

本书适合用做大学教材或从业者的参考书。

图灵程序设计丛书
算法（第4版）

◆ 著 [美] Robert Sedgewick Kevin Wayne
译 谢路云
责任编辑 朱 巍
执行编辑 丁晓昀 刘美英

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：787×1092 1/16
印张：40.5 彩插 12
字数：1115千字 2012年10月第1版
印数：1－5 000册 2012年10月北京第1次印刷
著作权合同登记号 图字：01-2011-5236号
ISBN 978-7-115-29380-0

定价：99.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版权声明

Authorized translation from the English language edition, entitled *Algorithms, Fourth Edition*, 978-0-321-57351-3 by Robert Sedgewick, Kevin Wayne, published by Pearson Education, Inc., publishing as Addison Wesley, Copyright © 2011 by Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD. and POSTS & TELECOM PRESS Copyright © 2012.

本书中文简体字版由Pearson Education Asia Ltd.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

谨以此书献给Adam、Andrew、Brett、Robbie，并特别感谢Linda。

——Robert Sedgewick

献给Jackie和Alex。

——Kevin Wayne

译 者 序

在计算机领域，算法是一个永恒的主题。即使仅把算法入门方面的书都摆出来，国内外的加起来怕是能铺满整个天安门广场。在这些书中，有几本尤其与众不同，本书就是其中之一。

本书是学生的良师。在翻译的过程中我曾无数次感叹："要是当年我能拥有这本书那该多好！"应该说本书是为在校学生量身打造的。没有数学基础？没关系，只要你在高中学过了数学归纳法，那么书中95%以上的数学内容你都可以看得懂，更何况书中还辅以大量图例。没学过编程？没关系，第1章会给大家介绍足够多的Java知识，即使你不是计算机专业的学生，也不会遇到困难。整本书的内容编排循序渐进，由易到难，前后呼应，足见作者的良苦用心。没有比本书更专业的算法教科书了。

本书是老师的好帮手。如果老师们还只能照本宣科，只能停留在算法本身一二三四的阶段，那就已经大大落后于这个时代了。算法并不仅仅是计算的方法，探究算法的过程反映出的是我们对这个世界的认知方法：是唯唯诺诺地将课本当做圣经，还是通过"实验—失败—再实验"循环的锤炼？数学是保证，数据是验证。本书通过各种算法，从各个角度，多次说明了这个道理，这也正是第1章是全书内容最多的一章的原因。希望每一位读者都不要错过第1章。无论你有没有编程基础，都会从中得到有益的启示。

本书是程序员的益友。在工作了多年之后，快速排序、霍夫曼编码、KMP等曾经熟悉的概念在你脑中是不是已经凋零成了一个个没有内涵的名词？是时候重新拾起它们了。无论是为手头的工作寻找线索，还是为下一份工作努力准备，这些算法基础知识都是你不能跳过的。本书强调软件工程中的最佳实践，特别适合已有工作经验的程序员朋友。所有的算法都是先有API，再有实现，之后是证明，最后是数据。这种先接口后实现、强调测试的做法，无疑是在工作中摸爬滚打多年的程序员最熟悉的。

本书也有一些遗憾，比如没有介绍动态规划这样重要的思想。但是瑕不掩玉，它仍然是最好的入门级算法书。我强烈地希望能够把本书翻译成中文，但同时也诚惶诚恐，如履薄冰，担心自己的水平不足以准确传达原文的意思。翻译的过程虽然辛苦，但我觉得非常值得。感谢人民邮电出版社图灵公司给了我这个机会，感谢编辑和审稿专家的细心检查。同时感谢我的妻子朱天的全力支持。译者水平有限，bug在所难免，还请读者批评指正。

谢路云

2012.9.17

前　言

本书力图研究当今最重要的计算机算法并将一些最基础的技能传授给广大求知者。它适合用做计算机科学进阶教材，面向已经熟悉了计算机系统并掌握了基本编程技能的学生。本书也可用于自学，或是作为开发人员的参考手册，因为书中实现了许多实用算法并详尽分析了它们的性能特点和用途。这本书取材广泛，很适合作为该领域的入门教材。

算法和数据结构的学习是所有计算机科学教学计划的基础，但它并不只是对程序员和计算机系的学生有用。任何计算机使用者都希望计算机能运行得更快一些或是能解决更大规模的问题。本书中的算法代表了近50年来的大量优秀研究成果，是人们工作中必备的知识。从物理中的*N*体模拟问题到分子生物学中的基因序列问题，我们描述的基本方法对科学研究而言已经必不可少；从建筑建模系统到模拟飞行器，这些算法已经成为工程领域极其重要的工具；从数据库系统到互联网搜索引擎，算法已成为现代软件系统中不可或缺的一部分。这仅是几个例子而已，随着计算机应用领域的不断扩张，这些基础方法的影响也会不断扩大。

在开始学习这些基础算法之前，我们先要熟悉全书中都将会用到的栈、队列等低级抽象的数据类型。然后依次研究排序、搜索、图和字符串方面的基础算法。最后一章将会从宏观角度总结全书的内容。

独特之处

本书致力于研究有实用价值的算法。书中讲解了多种算法和数据结构，并提供了大量相关的信息，读者应该能有信心在各种计算环境下实现、调试并应用它们。本书的特点涉及以下几个方面。

算法　书中均有算法的完整实现，并讨论了程序在多个样例上的运行状况。书中的代码都是可以运行的程序而非伪代码，因此非常便于投入使用。书中程序是用Java语言编写的，但其编程风格方便读者使用其他现代编程语言重用其中的大部分代码来实现相同算法。

数据类型　我们在数据抽象上采用了现代编程风格，将数据结构和算法封装在了一起。

应用　每一章都会给出所述算法起到关键作用的应用场景。这些场景多种多样，包括物理模拟与分子生物学、计算机与系统工程学，以及我们熟悉的数据压缩和网络搜索等。

学术性　我们非常重视使用数学模型来描述算法的性能。我们用模型预测算法的性能，然后在真实的环境中运行程序来验证预测。

广度　本书讨论了基本的抽象数据类型、排序算法、搜索算法、图及字符串处理。我们在算法

的讨论中研究数据结构、算法设计范式、归纳法和解题模型。这将涵盖20世纪60年代以来的经典方法以及近年来产生的新方法。

我们的主要目标是将今天最重要的实用算法介绍给尽可能广泛的群体。这些算法一般都十分巧妙奇特，20行左右的代码就足以表达。它们展现出的问题解决能力令人叹为观止。没有它们，创造计算智能、解决科学问题、开发商业软件都是不可能的。

本书网站

本书的一个亮点是它的配套网站algs4.cs.princeton.edu。这一网站面向教师、学生和专业人士，免费提供关于算法和数据结构的丰富资料。

一份在线大纲　包含了本书内容的结构并提供了链接，浏览起来十分方便。

全部实现代码　书中所有的代码均可以在这里找到，且其形式适合用于程序开发。此外，还包括算法的其他实现，例如高级的实现、书中提及的改进的实现、部分习题的答案以及多个应用场景的客户端代码。我们的重点是用真实的应用环境来测试算法。

习题与答案　网站还提供了一些附加的选择题（只需要一次单击便可获取答案）、很多算法应用的例子、编程练习和答案以及一些有挑战性的难题。

动态可视化　书是死的，但网站是活的，在这里我们充分利用图形类演示了算法的应用效果。

课程资料　网站包含和本书及网上内容对应的一整套幻灯片，以及一系列编程作业、核对表、测试数据和备课手册。

相关资料链接　网站包含大量的链接，提供算法应用的更多背景知识以及学习算法的其他资源。

我们希望这个站点和本书互为补充。一般来说，建议读者在第一次学习某种算法或是希望获得整体概念时看书，并把网站作为编程时的参考或是在线查找更多信息的起点。

作为教材

本书为计算机科学专业进阶的教材，涵盖了这门学科的核心内容，并能让学生充分锻炼编程、定量推理和解决问题等方面的能力。一般来说，此前学过一门计算机方面的先导课程就足矣，只要熟悉一门现代编程语言并熟知现代计算机系统，就都能够阅读本书。

虽然本书使用Java实现算法和数据结构，但其代码风格使得熟悉其他现代编程语言的人也能看懂。我们充分利用了Java的抽象性（包括泛型），但不会依赖这门语言的独门特性。

书中涉及的多数数学知识都有完整的讲解（少数会有延伸阅读），因此阅读本书并不需要准备太多数学知识，不过有一定的数学基础当然更好。应用场景都来自其他学科的基础内容，同样也在书中有完整介绍。

本书涉及的内容是任何准备主修计算机科学、电气工程、运筹学等专业的学生应了解的基础知识，并且对所有对科学、数学或工程学感兴趣的学生也十分有价值。

背景介绍

这本书意在接续我们的一本基础教材《Java程序设计：一种跨学科的方法》，那本书对计算机领域做了概括性介绍。这两本书合起来可用做两到三个学期的计算机科学入门课程教材，为所有学生在自然科学、工程学和社会科学中解决计算问题提供必备的基础知识。

本书大部分内容来自Sedgewick的算法系列图书。本质上，本书和该系列的第1版和第2版最接近，但还包含了作者多年教学和学习的经验。Sedgewick的《C算法（第3版）》、《C++算法（第3版）》、《Java算法（第3版）》更适合用做参考书或是高级课程的教材，而本书则是专门为大学一、二年级学生设计的一学期教材，也是最新的基础入门书或从业者的参考书。

致谢

本书的编写花了近40年时间，因此想要一一列出所有参与人是不可能的。本书的前几版一共列出了好几十人，其中包括（按字母顺序）Andrew Appel、Trina Avery、Marc Brown、Lyn Dupré、Philippe Flajolet、Tom Freeman、Dave Hanson、Janet Incerpi、Mike Schidlowsky、Steve Summit和Chris Van Wyk。我要感谢他们所有人，尽管其中有些人的贡献要追溯到几十年前。至于第4版，我们要感谢试用了本书样稿的普林斯顿及其他院校的数百名学生，以及通过本书网站发表意见和指出错误的世界各地的读者。

我们还要感谢普林斯顿大学对于高质量教学的坚定支持，这是本书得以面世的基础。

Peter Gordon几乎从本书写作之初就提出了很多有用的建议，这一版奉行的“归本溯源”的指导思想也是他最早提出的。关于第4版，我们要感谢Barbara Wood认真又专业的编辑工作，Julie Nahil对生产过程的管理，以及Pearson出版公司中为本书的付梓和营销辛勤工作的朋友。所有人都在积极地追赶进度，而本书的质量并没有受到丝毫影响。

目　录

第1章　基础……1

1.1　基础编程模型……4

1.1.1　Java程序的基本结构……4

1.1.2　原始数据类型与表达式……6

1.1.3　语句……8

1.1.4　简便记法……9

1.1.5　数组……10

1.1.6　静态方法……12

1.1.7　API……16

1.1.8　字符串……20

1.1.9　输入输出……21

1.1.10　二分查找……28

1.1.11　展望……30

1.2　数据抽象……38

1.2.1　使用抽象数据类型……38

1.2.2　抽象数据类型举例……45

1.2.3　抽象数据类型的实现……52

1.2.4　更多抽象数据类型的实现……55

1.2.5　数据类型的设计……60

1.3　背包、队列和栈……74

1.3.1　API……74

1.3.2　集合类数据类型的实现……81

1.3.3　链表……89

1.3.4　综述……98

1.4　算法分析……108

1.4.1　科学方法……108

1.4.2　观察……108

1.4.3　数学模型……112

1.4.4　增长数量级的分类……117

1.4.5　设计更快的算法……118

1.4.6　倍率实验……121

1.4.7　注意事项……123

1.4.8　处理对于输入的依赖……124

1.4.9　内存……126

1.4.10　展望……129

1.5　案例研究：union-find算法……136

1.5.1　动态连通性……136

1.5.2　实现……140

1.5.3　展望……148

第2章　排序……152

2.1　初级排序算法……153

2.1.1　游戏规则……153

2.1.2　选择排序……155

2.1.3　插入排序……157

2.1.4　排序算法的可视化……159

2.1.5　比较两种排序算法……159

2.1.6　希尔排序……162

2.2　归并排序……170

2.2.1　原地归并的抽象方法……170

2.2.2　自顶向下的归并排序……171

2.2.3　自底向上的归并排序……175

2.2.4　排序算法的复杂度……177

2.3　快速排序……182

2.3.1　基本算法……182

2.3.2　性能特点……185

2.3.3　算法改进……187

2.4　优先队列……195

2.4.1　API……195

2.4.2　初级实现……197

2.4.3　堆的定义……198

2.4.4　堆的算法……199

2.4.5　堆排序……205

2.5　应用……214

2.5.1　将各种数据排序……214

2.5.2　我应该使用哪种排序算法……218

2.5.3　问题的归约……219

2.5.4　排序应用一览……221

第3章 查找 ……227
3.1 符号表 ……228
3.1.1 API ……228
3.1.2 有序符号表 ……230
3.1.3 用例举例 ……233
3.1.4 无序链表中的顺序查找 ……235
3.1.5 有序数组中的二分查找 ……238
3.1.6 对二分查找的分析 ……242
3.1.7 预览 ……244
3.2 二叉查找树 ……250
3.2.1 基本实现 ……250
3.2.2 分析 ……255
3.2.3 有序性相关的方法与删除操作 ……257
3.3 平衡查找树 ……269
3.3.1 2-3查找树 ……269
3.3.2 红黑二叉查找树 ……275
3.3.3 实现 ……280
3.3.4 删除操作 ……282
3.3.5 红黑树的性质 ……284
3.4 散列表 ……293
3.4.1 散列函数 ……293
3.4.2 基于拉链法的散列表 ……297
3.4.3 基于线性探测法的散列表 ……300
3.4.4 调整数组大小 ……304
3.4.5 内存使用 ……306
3.5 应用 ……312
3.5.1 我应该使用符号表的哪种实现 ……312
3.5.2 集合的API ……313
3.5.3 字典类用例 ……315
3.5.4 索引类用例 ……318
3.5.5 稀疏向量 ……322

第4章 图 ……329
4.1 无向图 ……331
4.1.1 术语表 ……331
4.1.2 表示无向图的数据类型 ……333
4.1.3 深度优先搜索 ……338
4.1.4 寻找路径 ……342
4.1.5 广度优先搜索 ……344
4.1.6 连通分量 ……349
4.1.7 符号图 ……352
4.1.8 总结 ……358
4.2 有向图 ……364
4.2.1 术语 ……364
4.2.2 有向图的数据类型 ……365
4.2.3 有向图中的可达性 ……367
4.2.4 环和有向无环图 ……369
4.2.5 有向图中的强连通性 ……378
4.2.6 总结 ……385
4.3 最小生成树 ……390
4.3.1 原理 ……391
4.3.2 加权无向图的数据类型 ……393
4.3.3 最小生成树的API和测试用例 ……396
4.3.4 Prim算法 ……398
4.3.5 Prim算法的即时实现 ……401
4.3.6 Kruskal算法 ……404
4.3.7 展望 ……407
4.4 最短路径 ……412
4.4.1 最短路径的性质 ……413
4.4.2 加权有向图的数据结构 ……414
4.4.3 最短路径算法的理论基础 ……420
4.4.4 Dijkstra算法 ……421
4.4.5 无环加权有向图中的最短路径算法 ……425
4.4.6 一般加权有向图中的最短路径问题 ……433
4.4.7 展望 ……445

第5章 字符串 ……451
5.1 字符串排序 ……455
5.1.1 键索引计数法 ……455
5.1.2 低位优先的字符串排序 ……458
5.1.3 高位优先的字符串排序 ……461
5.1.4 三向字符串快速排序 ……467
5.1.5 字符串排序算法的选择 ……470
5.2 单词查找树 ……474
5.2.1 单词查找树 ……475
5.2.2 单词查找树的性质 ……483
5.2.3 三向单词查找树 ……485
5.2.4 三向单词查找树的性质 ……487
5.2.5 应该使用字符串符号表的哪种实现 ……489
5.3 子字符串查找 ……493
5.3.1 历史简介 ……493
5.3.2 暴力子字符串查找算法 ……494

5.3.3 Knuth-Morris-Pratt子字符串查找算法……496
5.3.4 Boyer-Moore字符串查找算法……502
5.3.5 Rabin-Karp指纹字符串查找算法……505
5.3.6 总结……509
5.4 正则表达式……514
5.4.1 使用正则表达式描述模式……514
5.4.2 缩略写法……516
5.4.3 正则表达式的实际应用……517
5.4.4 非确定有限状态自动机……518
5.4.5 模拟NFA的运行……520
5.4.6 构造与正则表达式对应的NFA……522
5.5 数据压缩……529
5.5.1 游戏规则……529
5.5.2 读写二进制数据……530
5.5.3 局限……533
5.5.4 热身运动：基因组……534
5.5.5 游程编码……537
5.5.6 霍夫曼压缩……540
第6章 背景……558
索引……611

第1章 基 础

本书的目的是研究多种重要而实用的*算法*，即适合用计算机实现的解决问题的方法。和算法关系最紧密的是*数据结构*，即便于算法操作的组织数据的方法。本章介绍的就是学习算法和数据结构所需要的基本工具。

首先要介绍的是我们的*基础编程模型*。本书中的程序只用到了 Java 语言的一小部分，以及我们自己编写的用于封装输入输出以及统计的一些库。1.1 节总结了相关的语法、语言特性和书中将会用到的库。

接下来我们的重点是*数据抽象*并定义*抽象数据类型*（ADT）以进行模块化编程。在 1.2 节中我们介绍了用 Java 实现抽象数据类型的过程，包括定义它的*应用程序编程接口*（API）然后通过 Java 的类机制来实现它以供各种用例使用。

之后，作为重要而实用的例子，我们将学习三种基础的抽象数据类型：背包、队列和栈。1.3 节用数组、变长数组和链表实现了背包、队列和栈的 API，它们是全书算法实现的起点和样板。

性能是算法研究的一个核心问题。1.4 节描述了分析算法性能的方法。我们的基本做法是科学式的，即先对性能提出假设，建立数学模型，然后用多种实验验证它们，必要时重复这个过程。

我们用一个*连通性*问题作为例子结束本章，它的解法所用到的算法和数据结构可以实现经典的 union-find 抽象数据结构。

算法

编写一段计算机程序一般都是实现一种已有的*方法*来解决某个问题。这种方法大多和使用的编程语言无关——它适用于各种计算机以及编程语言。是这种方法而非计算机程序本身描述了解决问题的步骤。在计算机科学领域，我们用*算法*这个词来描述一种有限、确定、有效的并适合用计算机程序来实现的解决问题的方法。算法是计算机科学的基础，是这个领域研究的核心。

要定义一个算法，我们可以用自然语言描述解决某个问题的过程或是编写一段程序来实现这个过程。如发明于 2300 多年前的*欧几里德算法*所示，其目的是找到两个数的最大公约数：

自然语言描述

计算两个非负整数 p 和 q 的最大公约数：若 q 是 0，则最大公约数为 p。否则，将 p 除以 q 得到余数 r，p 和 q 的最大公约数即为 q 和 r 的最大公约数。

Java 语言描述

```
Public static int gcd(int p, int q)
{
    if (q == 0) return p;
    int r = p % q;
    return gcd(q, r);
}
```

欧几里德算法

如果你不熟悉欧几里德算法，那么你应该在学习了 1.1 节之后完成练习 1.1.24 和练习 1.1.25。在本书中，我们将用计算机程序来描述算法。这样做的重要原因之一是可以更容易地验证它们是否如所要求的那样有限、确定和有效。但你还应该意识到用某种特定语言写出一段程序只是表达一个算法的一种方法。数十年来本书中许多算法都曾被表达为多种编程语言的程序，这正说明每种算法都是适合于在任何计算机上用任何编程语言实现的方法。

我们关注的大多数算法都需要适当地组织数据，而为了组织数据就产生了*数据结构*，数据结构也是计算机科学研究的核心对象，它和算法的关系非常密切。在本书中，我们的观点是数据结构是算法的副产品或是结果，因此要理解算法必须学习数据结构。简单的算法也会产生复杂的数据结构，相应地，复杂的算法也许只需要简单的数据结构。本书中我们将会研讨许多数据结构的性质，也许
4 本书就应该叫《*算法与数据结构*》。

当用计算机解决一个问题时，一般都存在多种不同的方法。对于小型问题，只要管用，方法的不同并没有什么关系。但是对于大型问题（或者是需要解决大量小型问题的应用），我们就需要设计能够有效利用时间和空间的方法了。

学习算法的主要原因是它们能节约非常多的资源，甚至能够让我们完成一些本不可能完成的任务。在某些需要处理上百万个对象的应用程序中，设计优良的算法甚至可以将程序运行的速度提高数百万倍。在本书中我们将在多个场景中看到这样的例子。与此相反，花费金钱和时间去购置新的硬件可能只能将速度提高十倍或是百倍。无论在任何应用领域，精心设计的算法都是解决大型问题最有效的方法。

在编写庞大或者复杂的程序时，理解和定义问题、控制问题的复杂度和将其分解为更容易解决的子问题需要大量的工作。很多时候，分解后的子问题所需的算法实现起来都比较简单。但是在大多数情况下，某些算法的选择是非常关键的，因为大多数系统资源都会消耗在它们身上。本书的焦点就是这类算法。我们所研究的基础算法在许多应用领域都是解决困难问题的有效方法。

计算机程序的共享已经变得越来越广泛，尽管书中涉及了许多算法，我们也只实现了其中的一小部分。例如，Java 库包含了许多重要算法的实现。但是，实现这些基础算法的简化版本有助于我们更好地理解、使用和优化它们在库中的高级版本。更重要的是，我们经常需要重新实现这些基础算法，因为在全新的环境中（无论是硬件的还是软件的），原有的实现无法将新环境的优势完全发挥出来。在本书中，我们的重点是用最简洁的方式实现优秀的算法。我们会仔细地实现算法的关键部分，并尽最大努力揭示如何进行有效的底层优化工作。

5 为一项任务选择最合适的算法是困难的，这可能会需要复杂的数学分析。计算机科学中研究这种问题的分支叫做*算法分析*。通过分析，我们将要学习的许多算法都有着优秀的理论性能；而另一些我们则只是根据经验知道它们是可用的。我们的主要目标是学习典型问题的各种有效算法，但也会注意比较不同算法之间的性能差异。不应该使用资源消耗情况未知的算法，因此我们会时刻关注算法的期望性能。

本书框架

接下来概述一下全书的主要内容，给出涉及的主题以及本书大致的组织结构。这组主题触及了尽可能多的基础算法，其中的某些领域是计算机科学的核心内容，通过对这些领域的深入研究，我们找出了应用广泛的基本算法，而另一些算法则来自计算机科学和相关领域比较前沿的研究成果。总之，本书讨论的算法都是数十年来研发的重要成果，它们将继续在快速发展的计算机应用中扮演重要角色。

第 1 章　基础

它讲解了在随后的章节中用来实现、分析和比较算法的基本原则和方法，包括 Java 编程模型、数据抽象、基本数据结构、集合类的抽象数据类型、算法性能分析的方法和一个案例分析。

第 2 章　排序

有序地重新排列数组中的元素是非常重要的基础算法。我们会深入研究各种排序算法，包括插入排序、选择排序、希尔排序、快速排序、归并排序和堆排序。同时我们还会讨论另外一些算法，它们用于解决几个与排序相关的问题，例如优先队列、选举以及归并。其中许多算法会成为后续章节中其他算法的基础。

第 3 章　查找

从庞大的数据集中找到指定的条目也是非常重要的。我们将会讨论基本的和高级的查找算法，包括二叉查找树、平衡查找树和散列表。我们会梳理这些方法之间的关系并比较它们的性能。

第 4 章　图

图的主要内容是对象和它们的连接，连接可能有权重和方向。利用图可以为大量重要而困难的问题建模，因此图算法的设计也是本书的一个主要研究领域。我们会研究深度优先搜索、广度优先搜索、连通性问题以及若干其他算法和应用，包括 Kruskal 和 Prim 的最小生成树算法、Dijkstra 和 Bellman-Ford 的最短路径算法。

6

第 5 章　字符串

字符串是现代应用程序中的重要数据类型。我们将会研究一系列处理字符串的算法，首先是对字符串键的排序和查找的快速算法，然后是子字符串查找、正则表达式模式匹配和数据压缩算法。此外，在分析一些本身就十分重要的基础问题之后，这一章对相关领域的前沿话题也作了介绍。

第 6 章　背景

这一章将讨论与本书内容有关的若干其他前沿研究领域，包括科学计算、运筹学和计算理论。我们会介绍性地讲一下基于事件的模拟、B 树、后缀数组、最大流量问题以及其他高级主题，以帮助读者理解算法在许多有趣的前沿研究领域中所起到的巨大作用。最后，我们会讲一讲搜索问题、问题转化和 NP 完全性等算法研究的支柱理论，以及它们和本书内容的联系。

学习算法是非常有趣和令人激动的，因为这是一个历久弥新的领域（我们学习的绝大多数算法都还不到“五十岁”，有些还是最近才发明的，但也有一些算法已经有数百年的历史）。这个领域不断有新的发现，但研究透彻的算法仍然是少数。本书中既有精巧、复杂和高难度的算法，也有优雅、朴素和简单的算法。在科学和商业应用中，我们的目标是理解前者并熟悉后者，这样才能掌握这些有用的工具并学会算法式思考，以迎接未来计算任务的挑战。

7

1.1 基础编程模型

我们学习算法的方法是用 Java 编程语言编写的程序来实现算法。这样做是出于以下原因：

- 程序是对算法精确、优雅和完全的描述；
- 可以通过运行程序来学习算法的各种性质；
- 可以在应用程序中直接使用这些算法。

相比用自然语言描述算法，这些是重要而巨大的优势。

这样做的一个缺点是我们要使用特定的编程语言，这会使分离算法的思想和实现细节变得困难。我们在实现算法时考虑到了这一点，只使用了大多数现代编程语言都具有且能够充分描述算法所必需的语法。

我们仅使用了 Java 的一个子集。尽管我们没有明确地说明这个子集的范围，但你也会看到我们只使用了很少的 Java 特性，而且会优先使用大多数现代编程语言所共有的语法。我们的代码是完整的，因此希望你能下载这些代码并用我们的测试数据或是你自己的来运行它们。

我们把描述和实现算法所用到的语言特性、软件库和操作系统特性总称为*基础编程模型*。本节以及 1.2 节会详细说明这个模型，相关内容自成一体，主要是作为文档供读者查阅，以便理解本书的代码。我们的另一本入门级的书籍 *An Introduction to Programming in Java: An Interdisciplinary Approach* 也使用了这个模型。

作为参考，图 1.1.1 所示的是一个完整的 Java 程序。它说明了我们的基础编程模型的许多基本特点。在讨论语言特性时我们会用这段代码作为例子，但可以先不用考虑代码的实际意义（它实现了经典的二分查找算法，并在白名单过滤应用中对算法进行了检验，请见 1.1.10 节）。我们假设你具备某种主流语言编程的经验，因此你应该知道这段代码中的大多数要点。图中的注释应该能够解答你的任何疑问。因为图中的代码某种程度上反映了本书代码的风格，而且对各种 Java 编程惯例和语言构造，在用法上我们都力求一致，所以即使是经验丰富的 Java 程序员也应该看一看。

8

1.1.1 Java 程序的基本结构

一段 Java 程序（类）或者是一个*静态方法*（函数）库，或者*定义了一个数据类型*。要创建静态方法库和定义数据类型，会用到下面五种语法，它们是 Java 语言的基础，也是大多数现代语言所共有的。

- *原始数据类型*：它们在计算机程序中精确地定义整数、*浮点数*和*布尔值*等。它们的定义包括取值范围和能够对相应的值进行的操作，它们能够被组合为类似于数学公式定义的*表达式*。
- *语句*：语句通过创建*变量*并对其赋值、控制运行流程或者引发副作用来进行计算。我们会使用六种语句：*声明*、*赋值*、*条件*、*循环*、*调用*和*返回*。
- *数组*：数组是多个同种数据类型的值的集合。
- *静态方法*：静态方法可以封装并重用代码，使我们可以用独立的模块开发程序。
- *字符串*：字符串是一连串的字符，Java 内置了对它们的一些操作。
- *标准输入 / 输出*：标准输入输出是程序与外界联系的桥梁。
- *数据抽象*：数据抽象封装和重用代码，使我们可以定义非原始数据类型，进而支持面向对象编程。

我们将在本节学习前五种语法，数据抽象是下一节的主题。

运行 Java 程序需要和操作系统或开发环境打交道。为了清晰和简洁，我们把这种输入命令执行程序的环境称为*虚拟终端*。请登录本书的网站去了解如何使用虚拟终端，或是现代系统中许多其他高级的编程开发环境的使用方法。

在例子中，BinarySearch 类有两个静态方法 `rank()` 和 `main()`。第一个方法 `rank()` 含有四条语句：两条声明语句，一条循环语句（该语句中又有一条赋值语句和两条条件语句）和一条返回语句。

第二个方法 main() 包含三条语句：一条声明语句、一条调用语句和一个循环语句（该语句中又包含一条赋值语句和一条条件语句）。

要执行一个 Java 程序，首先需要用 javac 命令编译它，然后再用 java 命令运行它。例如，要运行 BinarySearch，首先要输入 javac BinarySearch.java（这将生成一个叫 BinarySearch.class 的文件，其中含有这个程序的 Java 字节码）；然后再输入 java BinarySearch（接着是一个白名单文件名）把控制权移交给这段字节码程序。为了理解这段程序，我们接下来要详细介绍原始数据类型和表达式，各种 Java 语句、数组、静态方法、字符串和输入输出。

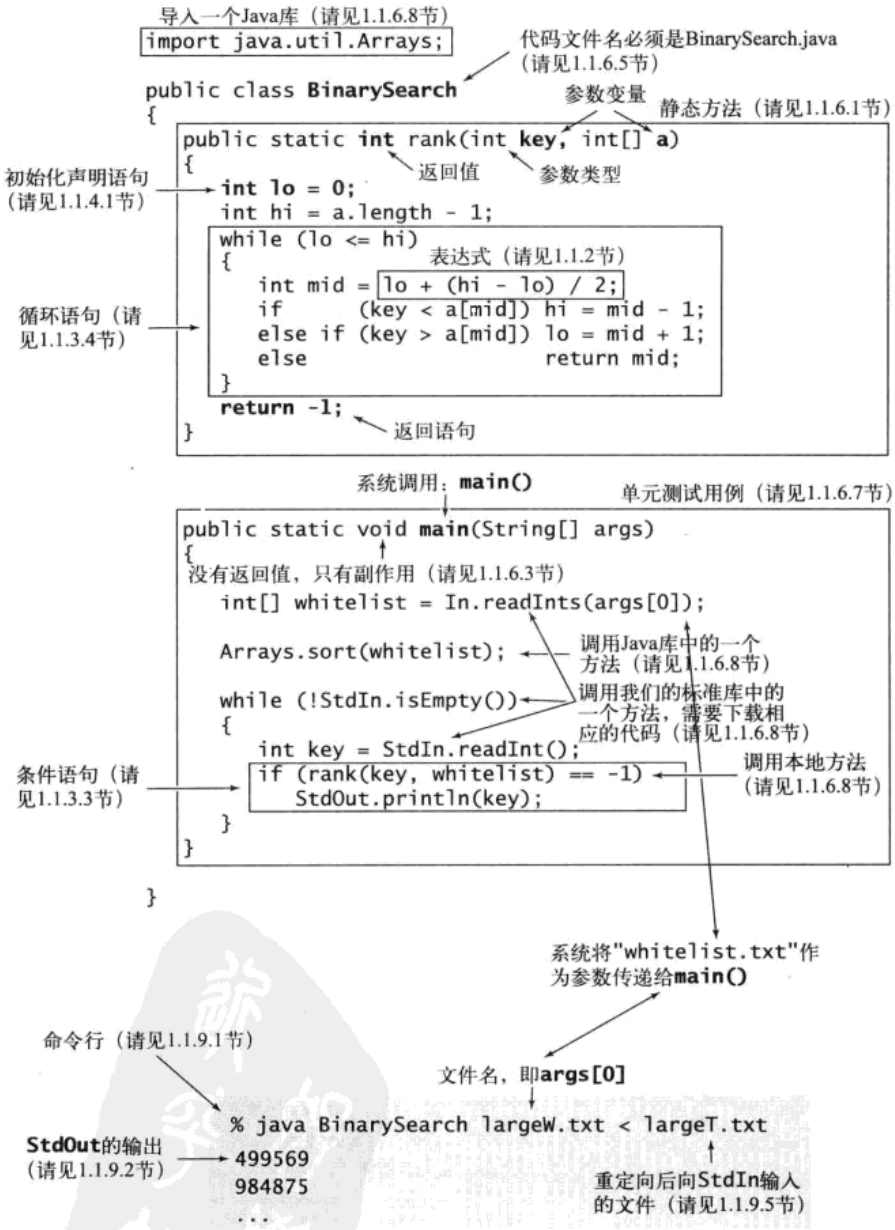


图 1.1.1 Java 程序及其命令行的调用

1.1.2 原始数据类型与表达式

数据类型就是一组数据和对其所能进行的操作的集合。首先考虑以下 4 种 Java 语言最基本的原始数据类型：

- 整型，及其算术运算符（int）；
- 浮点型，及其算术运算符（double）；
- 布尔型，它的值 {true, false} 及其逻辑操作（boolean）；
- 字符型，它的值是你能够输入的英文字母数字字符和符号（char）。

接下来我们看看如何指明这些类型的值和对这些类型的操作。

Java 程序控制的是用标识符命名的变量。每个变量都有自己的类型并存储了一个合法的值。在 Java 代码中，我们用类似数学表达式的表达式来实现对各种类型的操作。对于原始类型来说，我们用标识符来引用变量，用 +、-、*、/ 等运算符来指定操作，用字面量，例如 1 或者 3.14 来表示值，用形如 (x+2.236)/2 的表达式来表示对值的操作。表达式的目的就是计算某种数据类型的值。表 1.1.1 对这些基本内容进行了说明。

表 1.1.1 Java 程序的基本组成

术 语	例 子	定 义
原始数据类型	int double boolean char	一组数据和对其所能进行的操作的集合（Java 语言内置）
标识符	a abc Ab$ a_b ab123 lo hi	由字母、数字、下划线和 $ 组成的字符串，首字符不能是数字
变量	[任意标识符]	表示某种数据类型的值
运算符	+ - * /	表示某种数据类型的运算
字面量	int 1 0 -42 double 2.0 1.0e-15 3.14 boolean true false char 'a' '+' '9' '\n'	值在源代码中的表示
表达式	int lo + (hi - lo) / 2 double 1.0e-15 * t boolean lo <= hi	字面量、变量或是能够计算出结果的一串字面量、变量和运算符的组合

只要能够指定值域和在此值域上的操作，就能定义一个数据类型。表 1.1.2 总结了 Java 的 int、double、boolean 和 char 类型的相关信息。许多现代编程语言中的基本数据类型和它们都很相似。对于 int 和 double 来说，这些操作是我们熟悉的算数运算；对于 boolean 来说则是逻辑运算。需要注意的重要一点是，+、-、*、/ 都是被重载过的——根据上下文，同样的运算符对不同类型会执行不同的操作。这些初级运算的关键性质是运算产生的数据的数据类型和参与运算的数据的数据类型是相同的。这也意味着我们经常需要处理近似值，因为很多情况下由表达式定义的准确值并非参与表达式运算的值。例如，5/3 的值是 1 而 5.0/3.0 的值是 1.66666666666667，两者都很接近但并不准确地等于 5/3。下表并不完整，我们会在本节最后的答疑部分中讨论更多运算符和偶尔需要考虑到的各种异常情况。

表 1.1.2 Java 中的原始数据类型

类 型	值 域	运 算 符	典型表达式 表 达 式	值
int	-2^{31} 至 $+2^{31}-1$ 之间的整数（32 位，二进制补码）	+（加）	5 + 3	8
		-（减）	5 - 3	2
		*（乘）	5 * 3	15
		/（除）	5 / 3	1
		%（求余）	5 % 3	2

（续）

类　型	值　域	运 算 符	典型表达式 表 达 式	值
double	双精度实数（64 位，IEEE 754 标准）	+（加） -（减） *（乘） /（除）	3.141 - 0.03 2.0 - 2.0e-7 100 * 0.015 6.02e23 / 2.0	3.111 1.9999998 1.5 3.01e23
boolean	true 或 false	&&（与） ||（或） !（非） ^（异或）	true && false false || true !false true ^ true	false true true false
char	字符（16 位）	（算术运算符，但很少使用）		

12

1.1.2.1 表达式

如表 1.1.2 所示，Java 使用的是中缀表达式：一个字面量（或是一个表达式），紧接着是一个运算符，再接着是另一个字面量（或者另一个表达式）。当一个表达式包含一个以上的运算符时，运算符的作用顺序非常重要，因此 Java 语言规范约定了如下的运算符优先级：运算符 * 和 /（以及 %）的优先级高于 + 和 -（优先级越高，越早运算）；在逻辑运算符中，! 拥有最高优先级，之后是 &&，接下来是 ||。一般来说，相同优先级的运算符的运算顺序是从左至右。与在正常的算数表达式中一样，使用括号能够改变这些规则。因为不同语言中的优先级规则会有些许不同，我们在代码中会使用括号并用各种方法努力消除对优先级规则的依赖。

1.1.2.2 类型转换

如果不会损失信息，数值会被自动提升为高级的数据类型。例如，在表达式 1+2.5 中，1 会被转换为浮点数 1.0，表达式的值也为 double 值 3.5。转换指的是在表达式中把类型名放在括号里将其后的值转换为括号中的类型。例如，(int)3.7 的值是 3 而 (double)3 的值是 3.0。需要注意的是将浮点型转换为整型将会截断小数部分而非四舍五入，在复杂的表达式中的类型转换可能会很复杂，应该小心并尽量少使用类型转换，最好是在表达式中只使用同一类型的字面量和变量。

1.1.2.3 比较

下列运算符能够比较相同数据类型的两个值并产生一个布尔值：相等（==）、不等（!=）、小于（<）、小于等于（<=）、大于（>）和大于等于（>=）。这些运算符被称为混合类型运算符，因为它们的结果是布尔型，而不是参与比较的数据类型。结果是布尔型的表达式被称为布尔表达式。我们将会看到这种表达式是条件语句和循环语句的重要组成部分。

1.1.2.4 其他原始类型

Java 的整型能够表示 2^{32} 个不同的值，用一个 32 位二进制即可表示（虽然现在的许多计算机有 64 位二进制，但整型仍然是 32 位）。与此相似，浮点型的标准规定为 64 位。这些大小对于一般应用程序中使用的整数和实数已经足够了。为了提供更大的灵活性，Java 还提供了其他五种原始数据类型：

- 64 位整数，及其算术运算符 (long)；
- 16 位整数，及其算术运算符 (short)；
- 16 位字符，及其算术运算符 (char)；
- 8 位整数，及其算术运算符 (byte)；
- 32 位单精度实数，及其算术运算符 (float)。

13 在本书中我们大多使用 int 和 double 进行算术运算，因此我们在此不会再详细讨论其他类似的数据类型。

1.1.3 语句

Java 程序是由语句组成的。语句能够通过创建和操作变量、对变量赋值并控制这些操作的执行流程来描述运算。语句通常会被组织成代码段，即花括号中的一系列语句。

- 声明语句：创建某种类型的变量并用标识符为其命名。
- 赋值语句：将（由表达式产生的）某种类型的数值赋予一个变量。Java 还有一些隐式赋值的语法可以使某个变量的值相对于当前值发生变化，例如将一个整型值加 1。
- 条件语句：能够简单地改变执行流程——根据指定的条件执行两个代码段之一。
- 循环语句：更彻底地改变执行流程——只要条件为真就不断地反复执行代码段中的语句。
- 调用和返回语句：和静态方法有关（见 1.1.6 节），是改变执行流程和代码组织的另一种方式。

程序就是由一系列声明、赋值、条件、循环、调用和返回语句组成的。一般来说代码的结构都是嵌套的：一个条件语句或循环语句的代码段中也能包含条件语句或是循环语句。例如，rank() 中的 while 循环就包含一个 if 语句。接下来，我们逐个说明各种类型的语句。

1.1.3.1 声明语句

声明语句将一个变量名和一个类型在编译时关联起来。Java 需要我们用声明语句指定变量的名称和类型。这样，我们就清楚地指明了能够对其进行的操作。Java 是一种强类型的语言，因为 Java 编译器会检查类型的一致性（例如，它不会允许将布尔类型和浮点类型的变量相乘）。变量可以声明在第一次使用之前的任何地方——一般我们都在首次使用该变量的时候声明它。变量的作用域就是定义它的地方，一般由相同代码段中声明之后的所有语句组成。

1.1.3.2 赋值语句

赋值语句将（由一个表达式定义的）某个数据类型的值和一个变量关联起来。在 Java 中，当我们写下 c=a+b 时，我们表达的不是数学等式，而是一个操作，即令变量 c 的值等于变量 a 的值与变量 b 的值之和。当然，在赋值语句执行后，从数学上来说 c 的值必然会等于 a+b，但语句的目的是改变 c 的值（如果需要的话）。赋值语句的左侧必须是单个变量，右侧可以是能够得到相应类型的
14 值的任意表达式。

1.1.3.3 条件语句

大多数运算都需要用不同的操作来处理不同的输入。在 Java 中表达这种差异的一种方法是 if 语句：

```
if (<boolean expression>) { <block statements> }
```

这种描述方式是一种叫做模板的形式记法，我们偶尔会使用这种格式来表示 Java 的语法。尖括号（<>）中的是我们已经定义过的语法，这表示我们可以在指定的位置使用该语法的任意实例。在这里，<boolean expression> 表示一个布尔表达式，例如一个比较操作。<block statements> 表示一段 Java 语句。我们也可以给出 <boolean expression> 和 <block statements> 的形式定义，不过我们不想深入这些细节。if 语句的意义不言自明：当且仅当布尔表达式的值为真 (true) 时代码段中的语句才会被执行。以下 if-else 语句能够在两个代码段之间作出选择：

```
if (<boolean expression>) { <block statements> }
else                      { <block statements> }
```

1.1.3.4 循环语句

许多运算都需要重复。Java 语言中处理这种计算的基本语句的格式是：

```
while (<boolean expression>) { <block statements> }
```

while 语句和 if 语句的形式相似（只是用 while 代替了 if），但意义大有不同。当布尔表达式的值为假（false）时，代码什么也不做；当布尔表达式的值为真（true）时，执行代码段（和 if 一样），然后再次检查布尔表达式的值，如果仍然为真，再次执行代码段。如此这般，只要布尔表达式的值为真，就继续执行代码段。我们将循环语句中的代码段称为*循环体*。

1.1.3.5 break 与 continue 语句

有些情况下我们也会需要比基本的 if 和 while 语句更加复杂的流程控制。相应地，Java 支持在 while 循环中使用另外两条语句：

- break 语句，立即从循环中退出；
- continue 语句，立即开始下一轮循环。

本书很少在代码中使用它们（许多程序员从来都不用），但在某些情况下它们的确能够大大简化代码。

15

1.1.4 简便记法

程序有很多种写法，我们追求清晰、优雅和高效的代码。这样的代码经常会使用以下这些广为流传的简便写法（不仅仅是 Java，许多语言都支持它们）。

1.1.4.1 声明并初始化

可以将声明语句和赋值语句结合起来，在声明（创建）一个变量的同时将它初始化。例如，int i = 1; 创建了名为 i 的变量并赋予其初始值 1。最好在接近首次使用变量的地方声明它并将其初始化（为了限制它的作用域）。

1.1.4.2 隐式赋值

当希望一个变量的值相对于其当前值变化时，可以使用一些简便的写法。

- 递增 / 递减运算符，++i; 等价于 i=i+1; 且表达式为 i+1;。类似地，--i; 等价于 i=i-1;i++;。和 i--; 的意思相同，只是表达式的值为 i 的值。
- 其他复合运算符，在赋值语句中将一个二元运算符写在等号之前，等价于将左边的变量放在等号右边并作为第一个操作数。例如，i/=2; 等价于 i=i/2;。注意，i += 1; 等价于 i = i + 1;（以及 ++i;）。

1.1.4.3 单语句代码段

如果条件或循环语句的代码段只有一条语句，代码段的花括号可以省略。

1.1.4.4 for 语句

很多循环的模式都是这样的：初始化一个索引变量，然后使用 while 循环并将包含索引变量的表达式作为循环的条件，while 循环的最后一条语句会将索引变量加 1。使用 Java 的 for 语句可以更紧凑地表达这种循环：

```
for (<initialize>; <boolean expression>; <increment>)
{
   <block statements>
}
```

除了几种特殊情况之外，这段代码都等价于：

```
<initialize>;
while (<boolean expression>)
{
   <block statements>
   <increment>;
}
```

16 我们将使用 for 语句来表示对这种*初始化—递增*循环用法的支持。

表 1.1.3 总结了各种 Java 语句及其示例与定义。

表 1.1.3 Java 语句

语 句	示 例	定 义
声明语句	`int i;` `double c;`	创建一个指定类型的变量并用标识符命名
赋值语句	`a = b + 3;` `discriminant = b * b - 4.0 * c;`	将某一数据类型的值赋予一个变量
声明并初始化	`int i = 1;` `double c = 3.14159265;`	在声明时赋予变量初始值
隐式赋值	`++i;` `i += 1;`	`i = i + 1;`
条件语句（if）	`if (x < 0) x = -x;`	根据布尔表达式的值执行一条语句
条件语句（if-else）	`if (x > y) max = x;` `else max = y;`	根据布尔表达式的值执行两条语句中的一条
循环语句（while）	`int v = 0;` `while(v <= N)` `  v = 2 * v;` `double t = c;` `while (Math.abs(t - c/t) > 1e-15*t)` `t = (c/t + t) / 2.0;`	执行语句，直至布尔表达式的值变为假（false）
循环语句（for）	`for (int i = 1; i <= N; i++)` `  sum += 1.0/i;` `for (int i = 0; i <= N; i++)` `  StdOut.println(2*Math.PI*i/N);`	while 语句的简化版
调用语句	`int key = StdIn.readInt();`	调用另一方法（请见 1.1.6.2 节）
17 返回语句	`return false;`	从方法中返回（请见 1.1.6.3 节）

1.1.5 数组

*数组*能够顺序存储相同类型的多个数据。除了存储数据，我们也希望能够访问数据。访问数组中的某个元素的方法是将其编号然后*索引*。如果我们有 N 个值，它们的编号则为 0 至 N–1。这样对于 0 到 N–1 之间任意的 i，我们就能够在 Java 代码中用 a[i] 唯一地表示第 i 个元素的值。在 Java 中这种数组被称为*一维数组*。

1.1.5.1 创建并初始化数组

在 Java 程序中创建一个数组需要三步：

- 声明数组的名字和类型；
- 创建数组；
- 初始化数组元素。

在声明数组时，需要指定数组的名称和它含有的数据的类型。在创建数组时，需要指定数组的长度（元素的个数）。例如，在以下代码中，“完整模式”部分创建了一个有 N 个元素的 double 数组，

所有的元素的初始值都是 0.0。第一条语句是数组的声明，它和声明一个相应类型的原始数据类型变量十分相似，只有类型名之后的方括号说明我们声明的是一个数组。第二条语句中的关键字 new 使 Java 创建了这个数组。我们需要在运行时明确地创建数组的原因是 Java 编译器在编译时无法知道应该为数组预留多少空间（对于原始类型则可以）。for 语句初始化了数组的 N 个元素，将它们的值置为 0.0。在代码中使用数组时，一定要依次声明、创建并初始化数组。忽略了其中的任何一步都是很常见的编程错误。

完整模式

```
double[] a;
a = new double[N];
for (int i = 0; i < N; i++)
   a[i] = 0.0;
```

声明数组

创建数组

初始化数组

简化写法

```
double[] a = new double[N];
```

声明初始化

```
int[] a = { 1, 1, 2, 3, 5, 8 };
```

声明、创建并初始化一个数组

1.1.5.2 简化写法

为了精简代码，我们常常会利用 Java 对数组默认的初始化来将三个步骤合为一条语句，即上例中的简化写法。等号的左侧声明了数组，等号的右侧创建了数组。这种写法不需要 for 循环，因为在一个 Java 数组中 double 类型的变量的默认初始值都是 0.0，但如果你想使用不同的初始值，那么就需要使用 for 循环了。数值类型的默认初始值是 0，布尔型的默认初始值是 false。例子中的第三种方式用花括号将一列由逗号分隔的值在编译时将数组初始化。

18

1.1.5.3 使用数组

典型的数组处理代码请见表 1.1.4。在声明并创建数组之后，在代码的任何地方都能通过数组名之后的方括号中的索引来访问其中的元素。数组一经创建，它的大小就是固定的。程序能够通过 a.length 获取数组 a[] 的长度，而它的最后一个元素总是 a[a.length - 1]。Java 会自动进行边界检查——如果你创建了一个大小为 N 的数组，但使用了一个小于 0 或者大于 N-1 的索引访问它，程序会因为运行时抛出 ArrayOutOfBoundsException 异常而终止。

表 1.1.4 典型的数组处理代码

任　务	实现（代码片段）
找出数组中最大的元素	`double max = a[0];` `for (int i = 1; i < a.length; i++)` `  if (a[i] > max) max = a[i];`
计算数组元素的平均值	`int N = a.length;` `double sum = 0.0;` `for (int i = 0; i < N; i++)` `  sum += a[i];` `double average = sum / N;`
复制数组	`int N = a.length;` `double[] b = new double[N];` `for (int i = 0; i < N; i++)` `  b[i] = a[i];`
颠倒数组元素的顺序	`int N = a.length;` `for (int i = 0; i < N/2; i++)` `{` `  double temp = a[i];` `  a[i] = a[N-1-i];` `  a[N-i-1] = temp;` `}`

（续）

任 务	实现（代码片段）
矩阵相乘（方阵） a[][] * b[][] = c[][]	int N = a.length; double[][] c = new double[N][N]; for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { // 计算行 i 和列 j 的点乘 for (int k = 0; k < N; k++) c[i][j] += a[i][k]*b[k][j]; }

1.1.5.4 起别名

请注意，*数组名表示的是整个数组*——如果我们将一个数组变量赋予另一个变量，那么两个变量将会指向同一个数组。例如以下这段代码：

```
int[] a = new int[N];
...
a[i] = 1234;
...
int[] b = a;
...
b[i] = 5678;  // a[i] 的值也会变成 5678
```

这种情况叫做*起别名*，有时可能会导致难以察觉的问题。如果你是想将数组复制一份，那么应该声明、创建并初始化一个新的数组，然后将原数组中的元素值挨个复制到新数组，如表 1.1.4 的第三个例子所示。

1.1.5.5 二维数组

在 Java 中*二维数组*就是一维数组的数组。二维数组可以是参差不齐的（元素数组的长度可以不一致），但大多数情况下（根据合适的参数 M 和 N）我们都会使用 $M \times N$，即 M 行长度为 N 的数组的二维数组（也可以称数组含有 N 列）。在 Java 中访问二维数组也很简单。二维数组 a[][] 的第 i 行第 j 列的元素可以写作 a[i][j]。声明二维数组需要两对方括号。创建二维数组时要在类型名之后分别在方括号中指定行数以及列数，例如：

```
double[][] a = new double[M][N];
```

我们将这样的数组称为 $M \times N$ 的数组。我们约定，第一维是行数，第二维是列数。和一维数组一样，Java 会将数值类型的数组元素初始化为 0，将布尔型的数组元素初始化为 false。默认的初始化对二维数组更有用，因为可以节约更多的代码。下面这段代码和刚才只用一行就完成创建和初始化的语句是等价的：

```
double[][] a;
a = new double[M][N];
for (int i = 0; i < M; i++)
  for (int j = 0; j < N; j++)
    a[i][j] = 0.0;
```

19 ~ 21

在将二维数组初始化为 0 时这段代码是多余的，但是如果想要初始化为其他值，我们就需要嵌套的 for 循环了。

1.1.6 静态方法

本书中的所有 Java 程序要么是*数据类型的定义*（详见 1.2 节），要么是一个*静态方法库*。在许多语言中，静态方法被称为*函数*，因为它们和数学函数的性质类似。静态方法是一组在被调用时会

被顺序执行的语句。修饰符 static 将这类方法和 1.2 节的*实例方法*区别开来。当讨论两类方法共有的属性时我们会使用不加定语的*方法*一词。

1.1.6.1 静态方法

*方法*封装了由一系列语句所描述的运算。方法需要*参数*（某种数据类型的值）并根据参数计算出某种数据类型的*返回值*（例如数学函数的结果）或者产生某种*副作用*（例如打印一个值）。BinarySearch 中的静态函数 rank() 是前者的一个例子；main() 则是后者的一个例子。每个静态方法都是由*签名*（关键字 public static 以及函数的返回值，方法名以及一串各种类型的参数）和*函数体*（即包含在花括号中的代码）组成的，如图 1.1.2 所示。静态函数的例子请见表 1.1.5。

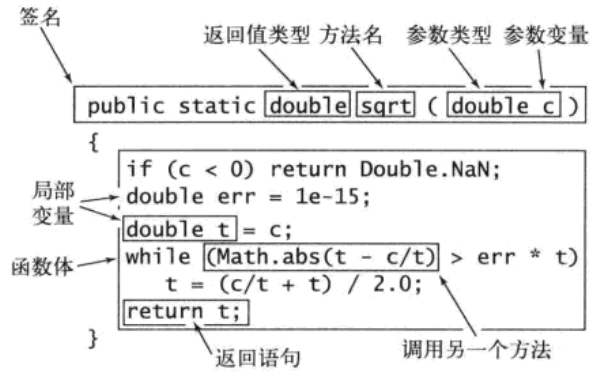


图 1.1.2 静态方法解析

表 1.1.5 典型静态方法的实现

任务	实现
计算一个整数的绝对值	`public static int abs(int x)` `{` `  if (x < 0) return -x;` `  else  return x;` `}`
计算一个浮点数的绝对值	`public static double abs(double x)` `{` `  if (x < 0.0) return -x;` `  else  return x;` `}`
判定一个数是否是素数	`public static boolean isPrime(int N)` `{` `  if (N < 2) return false;` `  for (int i = 2; i*i <= N; i++)` `  if (N % i == 0) return false;` `  return true;` `}`
计算平方根（牛顿迭代法）	`public static double sqrt(double c)` `{` `  if (c < 0) return Double.NaN;` `  double err = 1e-15;` `  double t = c;` `  while (Math.abs(t - c/t) > err * t)` `  t = (c/t + t) / 2.0;` `  return t;` `}`
计算直角三角形的斜边	`public static double hypotenuse(double a, double b)` `{ return Math.sqrt(a*a + b*b); }`
计算调和级数（请见表 1.4.5）	`public static double H(int N)` `{` `  double sum = 0.0;` `  for (int i = 1; i <= N; i++)` `  sum += 1.0 / i;` `  return sum;` `}`

1.1.6.2 调用静态方法

调用静态方法的方法是写出方法名并在后面的括号中列出参数值，用逗号分隔。当调用是表达式的一部分时，方法的返回值将会替代表达式中的方法调用。例如，BinarySearch 中调用 rank() 返回了一个 int 值。仅由一个方法调用和一个分号组成的语句一般用于产生副作用。例如，BinarySearch 的 main() 函数中对系统方法 Arrays.sort() 的调用产生的副作用，是将数组中的所有条目有序地排列。调用方法时，它的参数变量将被初始化为调用时所给出的相应表达式的值。返回语句将结束静态方法并将控制权交还给调用者。如果静态方法的目的是计算某个值，返回语句应该指定这个值（如果这样的静态方法在执行完所有的语句之后都没有返回语句，编译器会报错）。

22 ~ 23

1.1.6.3 方法的性质

对方法所有性质的完整描述超出了本书的范畴，但以下几点值得一提。

- 方法的参数按值传递：在方法中参数变量的使用方法和局部变量相同，唯一不同的是参数变量的初始值是由调用方提供的。方法处理的是参数的值，而非参数本身。这种方式产生的结果是在静态方法中改变一个参数变量的值对调用者没有影响。本书中我们一般不会修改参数变量。值传递也意味着数组参数将会是原数组的别名（见 1.1.5.4 节）——方法中使用的参数变量能够引用调用者的数组并改变其内容（只是不能改变原数组变量本身）。例如，Arrays.sort() 将能够改变通过参数传递的数组的内容，将其排序。
- 方法名可以被重载：例如，Java 的 Math 包使用这种方法为所有的原始数值类型实现了 Math.abs()、Math.min() 和 Math.max() 函数。重载的另一种常见用法是为函数定义两个版本，其中一个需要一个参数而另一个则为该参数提供一个默认值。
- 方法只能返回一个值，但可以包含多个返回语句：一个 Java 方法只能返回一个值，它的类型是方法签名中声明的类型。静态方法第一次执行到一条返回语句时控制权将会回到调用代码中。尽管可能存在多条返回语句，任何静态方法每次都只会返回一个值，即被执行的第一条返回语句的参数。
- 方法可以产生副作用：方法的返回值可以是 void，这表示该方法没有返回值。返回值为 void 的静态函数不需要明确的返回语句，方法的最后一条语句执行完毕后控制权将会返回给调用方。我们称 void 类型的静态方法会产生副作用（接受输入、产生输出、修改数组或者改变系统状态）。例如，我们的程序中的静态方法 main() 的返回值就是 void，因为它的作用是向外输出。技术上来说，数学方法的返回值都不会是 void（Math.random() 虽然不接受参数但也有返回值）。

24

2.1 节所述的实例方法也拥有这些性质，尽管两者在副作用方面大为不同。

1.1.6.4 递归

方法可以调用自己（如果你对递归概念感到奇怪，请完成练习 1.1.16 到练习 1.1.22）。例如，下面给出了 BinarySearch 的 rank() 方法的另一种实现。我们会经常使用递归，因为递归代码比相应的非递归代码更加简洁优雅、易懂。下面这种实现中的注释就言简意赅地说明了代码的作用。我们可以用数学归纳法证明这段注释所解释的算法的正确性。我们会在 3.1 节中展开这个话题并为二分查找提供一个这样的证明。

编写递归代码时最重要的有以下三点。

- 递归总有一个最简单的情况——方法的第一条语句总是一个包含 return 的条件语句。
- 递归调用总是去尝试解决一个规模更小的子问题，这样递归才能收敛到最简单的情况。在下面的代码中，第四个参数和第三个参数的差值一直在缩小。

- 递归调用的父问题和尝试解决的子问题之间不应该有交集。在下面的代码中，两个子问题各自操作的数组部分是不同的。

```
public static int rank(int key, int[] a)
{  return rank(key, a, 0, a.length - 1);  }

public static int rank(int key, int[] a, int lo, int hi)
{  //如果key存在于a[]中，它的索引不会小于lo且不会大于hi

   if (lo > hi) return -1;
   int mid = lo + (hi - lo) / 2;
   if      (key < a[mid]) return rank(key, a, lo, mid - 1);
   else if (key > a[mid]) return rank(key, a, mid + 1, hi);
   else                   return mid;
}
```

二分查找的递归实现

25

违背其中任意一条都可能得到错误的结果或是低效的代码（见练习 1.1.19 和练习 1.1.27），而坚持这些原则能写出清晰、正确且容易评估性能的程序。使用递归的另一个原因是我们可以使用数学模型的来估计程序的性能。我们会在 3.2 节的二分查找以及其他几个地方分析这个问题。

1.1.6.5 基础编程模型

*静态方法库*是定义在一个 Java 类中的一组静态方法。类的声明是 `public class` 加上类名，以及用花括号包含的静态方法。存放类的文件的文件名和类名相同，扩展名是 .java。Java 开发的基本模式是编写一个静态方法库（包含一个 `main()` 方法）来完成一个任务。输入 java 和类名以及一系列字符串就能调用类中的 `main()` 方法，其参数为由输入的字符串组成的一个数组。`main()` 的最后一条语句执行完毕之后程序终止。在本书中，当我们提到用于执行一项任务的 Java 程序时，我们指的是用这种模式开发的代码（可能还包括对数据类型的定义，如 1.2 节所示）。例如，BinarySearch 就是一个由两个静态方法 `rank()` 和 `main()` 组成的 Java 程序，它的作用是将输入中所有不在通过命令行指定的白名单中的数字打印出来。

1.1.6.6 模块化编程

这个模型的最重要之处在于通过静态方法库实现了模块化编程。我们可以构造许多个静态方法库（模块），一个库中的静态方法也能够调用另一个库中定义的静态方法。这能够带来许多好处：

- 程序整体的代码量很大时，每次处理的模块大小仍然适中；
- 可以共享和重用代码而无需重新实现；
- 很容易用改进的实现替换老的实现；
- 可以为解决编程问题建立合适的抽象模型；
- 缩小调试范围（请见 1.1.6.7 节关于单元测试的讨论）。

例如，BinarySearch 用到了三个独立的库，即我们的 StdOut 和 StdIn 以及 Java 的 Arrays，而这三个库又分别用到了其他的库。

1.1.6.7 单元测试

Java 编程的最佳实践之一就是每个静态方法库中都包含一个 `main()` 函数来测试库中的所有方法（有些编程语言不支持多个 `main()` 方法，因此不支持这种方式）。恰当的单元测试本身也是很有挑战性的编程任务。每个模块的 `main()` 方法至少应该调用模块中的其他代码并在某种程度上保

证它的正确性。随着模块的成熟，我们可以将 main() 方法作为一个开发用例，在开发过程中用它来测试更多的细节；也可以把它编成一个测试用例来对所有代码进行全面的测试。当用例越来越复杂时，我们可能会将它独立成一个模块。在本书中，我们用 main() 来说明模块的功能并将测试用
26 例留做练习。

1.1.6.8 外部库

我们会使用来自 4 个不同类型的库中的静态方法，重用每种库代码的方式都稍有不同。它们大多都是静态方法库，但也有部分是数据类型的定义并包含了一些静态方法。

- 系统标准库 java.lang.*：这其中包括 Math 库，实现了常用的数学函数；Integer 和 Double 库，能够将字符串转化为 int 和 double 值；String 和 StringBuilder 库，我们稍后会在本节和第 5 章中详细讨论；以及其他一些我们没有用到的库。
- 导入的系统库，例如 java.utils.Arrays：每个标准的 Java 版本中都含有上千个这种类型的库，不过本书中我们用到的并不多。要在程序的开头使用 import 语句导入才能使用这些库（我们也是这样做的）。
- 本书中的其他库：例如，其他程序也可以使用 BinarySearch 的 rank() 方法。要使用这些库，请在本书的网站上下载它们的源代码并放入你的工作目录中。
- 我们为本书（以及我们的另一本入门教材 *An Introduction to programming in Java: An Interdisciplinary Approach*）开发的标准库 Std*：我们会在下面简要地介绍这些库，它们的源代码和使用方法都能够在本书的网站上找到。

要调用另一个库中的方法（存放在相同或者指定的目录中，或是一个系统标准库，或是在类定义前用 import 语句导入的库），我们需要在方法前指定库的名称。例如，BinarySearch 的 main() 方法调用了系统库 java.utils.Arrays 的 sort() 方法，我们的库 StdIn 中的 readInts() 方法和 StdOut 库中的 println() 方法。

```
系统标准库
   Math
   Integer†
   Double†
   String†
   StringBuilder
   System
导入的系统库
   java.util.Arrays
我们的标准库
   StdIn
   StdOut
   StdDraw
   StdRandom
   StdStats
   In†
   Out†
† 含有静态方法的数据类型的定义
```

本书使用的含有静态方法的库

我们自己及他人使用模块化方式编写的方法库能够极大地扩展我们的编程模型。除了在 Java 的标准版本中可用的所有库之外，网上还有成千上万各种用途的代码库。为了将我们的编程模型限
27 制在一个可控范围之内，以将精力集中在算法上，我们只会使用以下所示的方法库，并在 1.1.7 节中列出了其中的部分方法。

1.1.7 API

模块化编程的一个重要组成部分就是记录库方法的用法并供其他人参考的文档。我们会统一使用应用程序编程接口（API）的方式列出本书中使用的每个库方法名称、签名和简短的描述。我们用用例来指代调用另一个库中的方法的程序，用实现描述实现了某个 API 方法的 Java 代码。

1.1.7.1 举例

在表 1.1.6 的例子中，我们用 java.lang 中 Math 库常用的静态方法说明 API 的文档格式。

这些方法实现了各种数学函数——它们通过参数计算得到某种类型的值（random() 除外，它没有对应的数学函数，因为它不接受参数）。它们的参数都是 double 类型且返回值也都是 double 类型，因此可以将它们看做 double 数据类型的扩展——这种扩展的能力正是现代编程语言的特性

之一。API 中的每一行描述了一个方法，提供了使用该方法所需要知道的所有信息。Math 库也定义了常数 PI（圆周率 π）和 E（自然对数 e），你可以在自己的程序中通过这些变量名引用它们。例如，Math.sin(Math.PI/2) 的结果是 1.0，Math.log(Math.E) 的结果也是 1.0（因为 Math.sin() 的参数是弧度而 Math.log() 使用的是自然对数函数）。

表 1.1.6 Java 的数学函数库的 API（节选）

public class **Math**	
static double abs(double a)	*a* 的绝对值
static double max(double a, double b)	*a* 和 *b* 中的较大者
static double min(double a, double b)	*a* 和 *b* 中的较小者
注 1：abs()、max() 和 min() 也定义了 int、long 和 float 的版本。	
static double sin(double theta)	正弦函数
static double cos(double theta)	余弦函数
static double tan(double theta)	正切函数
注 2：角用弧度表示，可以使用 toDegrees() 和 toRadians() 转换角度和弧度。 注 3：它们的反函数分别为 asin()、acos() 和 atan()。	
static double exp(double a)	指数函数（e^a）
static double log(double a)	自然对数函数（$\log_e a$，即 $\ln a$）
static double pow(double a, double b)	求 *a* 的 *b* 次方（a^b）
static double random()	[0, 1) 之间的随机数
static double sqrt(double a)	*a* 的平方根
static double E	常数 e（常数）
static double PI	常数 π（常数）

其他函数请见本书的网站。

28

1.1.7.2 Java 库

成千上万个库的在线文档是 Java 发布版本的一部分。为了更好地描述我们的编程模型，我们只是从中节选了本书所用到的若干方法。例如，BinarySearch 中用到了 Java 的 Arrays 库中的 sort() 方法，我们对它的记录如表 1.1.7 所示。

表 1.1.7 Java 的 Arrays 库节选（java.util.Arrays）

public class **Arrays**	
static void sort(int[] a)	将数组按升序排序

注：其他原始类型和 Object 对象也有对应版本的方法。

Arrays 库不在 java.lang 中，因此我们需要用 import 语句导入后才能使用它，与 BinarySearch 中一样。事实上，本书的第 2 章讲的正是数组的各种 sort() 方法的实现，包括 Arrays.sort() 中实现的归并排序和快速排序算法。Java 和很多其他编程语言都实现了本书讲解的许多基础算法。例如，Arrays 库还包含了二分查找的实现。为避免混淆，我们一般会使用自己的实现，但对于你已经掌握的算法使用高度优化的库实现当然也没有任何问题。

29

1.1.7.3 我们的标准库

为了介绍 Java 编程、为了科学计算以及算法的开发、学习和应用，我们也开发了若干库来提供一些实用的功能。这些库大多用于处理输入输出。我们也会使用以下两个库来测试和分析我们的实

现。第一个库扩展了 Math.random() 方法（见表 1.1.8），以根据不同的概率密度函数得到随机值；第二个库则支持各种统计计算（见表 1.1.9）。

表 1.1.8 我们的随机数静态方法库的 API

public class **StdRandom**	
static void initialize(long seed)	初始化
static double random()	0 到 1 之间的实数
static int uniform(int N)	0 到 N-1 之间的整数
static int uniform(int lo, int hi)	lo 到 hi-1 之间的整数
static double uniform(double lo, double hi)	lo 到 hi 之间的实数
static boolean bernoulli(double p)	返回真的概率为 p
static double gaussian()	正态分布，期望值为 0，标准差为 1
static double gaussian(double m, double s)	正态分布，期望值为 m，标准差为 s
static int discrete(double[] a)	返回 i 的概率为 a[i]
static void shuffle(double[] a)	将数组 a 随机排序

注：库中也包含为其他原始类型和 Object 对象重载的 shuffle() 函数。

表 1.1.9 我们的数据分析静态方法库的 API

public class **StdStats**	
static double max(double[] a)	最大值
static double min(double[] a)	最小值
static double mean(double[] a)	平均值
static double var(double[] a)	采样方差
static double stddev(double[] a)	采样标准差
static double median(double[] a)	中位数

StdRandom 的 initialize() 方法为随机数生成器提供种子。这样我们就可以重复和随机数有关的实验。以上一些方法的实现请参考表 1.1.10。有些方法的实现非常简单，为什么还要在方法库中实现它们？设计良好的方法库对这个问题的标准回答如下。

- 这些方法所实现的抽象层有助于我们将精力集中在实现和测试本书中的算法，而非生成随机数或是统计计算。每次都自己写完成相同计算的代码，不如直接在用例中调用它们要更简洁易懂。
- 方法库会经过大量测试，覆盖极端和罕见的情况，是我们可以信任的。这样的实现需要大量的代码。例如，我们经常需要使用的各种数据类型的实现，又比如 Java 的 Arrays 库针对不同数据类型对 sort() 进行了多次重载。

这些是Java模块化编程的基础，不过在这里可能有些夸张。但这些方法库的方法名称简单、实现容易，其中一些仍然能作为有趣的算法练习。因此，我们建议你到本书的网站上去学习一下 StdRandom.java 和 StdStats.java 的源代码并好好利用这些经过验证了的实现。使用这些库（以及检验它们）最简单的方法就是从网站上下载它们的源代码并放入你的工作目录。网站上讲解了在各种系统上使用它们的配置目录的方法。

表 1.1.10 StdRandom 库中的静态方法的实现

期望的结果	实 现
随机返回 [a,b) 之间的一个 double 值	`public static double uniform(double a, double b)` `{ return a + StdRandom.random() * (b-a); }`
随机返回 [0..N) 之间的一个 int 值	`public static int uniform(int N)` `{ return (int) (StdRandom.random() * N); }`
随机返回 [lo,hi) 之间的一个 int 值	`public static int uniform(int lo, int hi)` `{ return lo + StdRandom.uniform(hi - lo); }`
根据离散概率随机返回的 int 值（出现 i 的概率为 a[i]）	`public static int discrete(double[] a)` `{ // a[] 中各元素之和必须等于 1` `double r = StdRandom.random();` `double sum = 0.0;` `for (int i = 0; i < a.length; i++)` `{` `sum = sum + a[i];` `if (sum >= r) return i;` `}` `return -1;` `}`
随机将 double 数组中的元素排序（请见练习 1.1.36）	`public static void shuffle(double[] a)` `{` `int N = a.length;` `for (int i = 0; i < N; i++)` `{ // 将 a[i] 和 a[i..N-1] 中任意一个元素交换` `int r = i + StdRandom.uniform(N-i);` `double temp = a[i];` `a[i] = a[r];` `a[r] = temp;` `}` `}`

1.1.7.4 你自己编写的库

你应该将自己编写的每一个程序都当做一个日后可以重用的库。

- 编写用例，在实现中将计算过程分解成可控的部分。
- 明确静态方法库和与之对应的 API（或者多个库的多个 API）。
- 实现 API 和一个能够对方法进行独立测试的 main() 函数。

这种方法不仅能帮助你实现可重用的代码，而且能够教会你如何运用模块化编程来解决一个复杂的问题。 31 ~ 32

API 的目的是将调用和实现分离：除了 API 中给出的信息，调用者不需要知道实现的其他细节，而实现也不应考虑特殊的应用场景。API 使我们能够广泛地重用那些为各种目的独立开发的代码。没有任何一个 Java 库能够包含我们在程序中可能用到的所有方法，因此这种能力对于编写复杂的应用程序特别重要。相应地，程序员也可以将 API 看做调用和实现之间的一份契约，它详细说明了每个方法的作用。实现的目标就是能够遵守这份契约。一般来说，做到这一点有很多种方法，而且将调用者的代码和实现的代码分离使我们可以将老算法替换为更新更好的实现。在学习算法的过程中，这也使我们能够感受到算法的改进所带来的影响。 33

1.1.8 字符串

字符串是由一串字符（char 类型的值）组成的。一个 String 类型的字面量包括一对双引号和其中的字符，比如 "Hello, World"。String 类型是 Java 的一个数据类型，但并不是原始数据类型。我们现在就讨论 String 类型是因为它非常基础，几乎所有 Java 程序都会用到它。

1.1.8.1 字符串拼接

和各种原始数据类型一样，Java 内置了一个串联 String 类型字符串的运算符（+）。表 1.1.11 是对表 1.1.2 的补充。拼接两个 String 类型的字符串将得到一个新的 String 值，其中第一个字符串在前，第二个字符串在后。

表 1.1.11 Java 的 String 数据类型

类 型	值 域	举 例	运 算 符	表达式举例	
				表 达 式	值
String	一串字符	"AB" "Hello" "2.5"	+（拼接）	"Hi," + "Bob" "12" + "34" "1" + "+" + "2"	"Hi, Bob" "1234" "1+2"

1.1.8.2 类型转换

字符串的两个主要用途分别是将用户从键盘输入的内容转换成相应数据类型的值以及将各种数据类型的值转化成能够在屏幕上显示的内容。Java 的 String 类型为这些操作内置了相应的方法，而且 Integer 和 Double 库还包含了分别和 String 类型相互转化的静态方法（见表 1.1.12）。

表 1.1.12 String 值和数字之间相互转换的 API

public class **Integer**		
static int	parseInt(String s)	将字符串 s 转换为整数
static String	toString(int i)	将整数 i 转换为字符串
public class **Double**		
static double	parseDouble(String s)	将字符串 s 转换为浮点数
static String	toString(double x)	将浮点数 x 转换为字符串

1.1.8.3 自动转换

我们很少明确使用刚才提到的 toString() 方法，因为 Java 在连接字符串的时候会自动将任意数据类型的值转换为字符串：如果加号（+）的一个参数是字符串，那么 Java 会自动将其他参数都转换为字符串（如果它们不是的话）。除了像 "The square root of 2.0 is " + Math.sqrt(2.0) 这样的使用方式之外，这种机制也使我们能够通过一个空字符串 "" 将任意数据类型的值转换为字符串值。

1.1.8.4 命令行参数

在 Java 中字符串的一个重要的用途就是使程序能够接收到从命令行传递来的信息。这种机制很简单。当你输入命令 java 和一个库名以及一系列字符串之后，Java 系统会调用库的 main() 方法并将那一系列字符串变成一个数组作为参数传递给它。例如，BinarySearch 的 main() 方法需要一个命令行参数，因此系统会创建一个大小为 1 的数组。程序用这个值，也就是 args[0]，来获取白

名单文件的文件名并将其作为 StdIn.readInts() 的参数。另一种在我们的代码中常见的用法是当命令行参数表示的是数字时，我们会用 parseInt() 和 parseDouble() 方法将其分别转换为整数和浮点数。

字符串的用法是现代程序中的重要部分。现在我们还只是用 String 在外部表示为字符串的数字和内部表示为数字类数据类型的值进行转换。在 1.2 节中我们会看到 Java 为我们提供了非常丰富的字符串操作；在 1.4 节中我们会分析 String 类型在 Java 内部的表示方法；在第 5 章我们会深入学习处理字符串的各种算法。这些算法是本书中最有趣、最复杂也是影响力最大的一部分算法。 35

1.1.9 输入输出

我们的标准输入、输出和绘图库的作用是建立一个 Java 程序和外界交流的简易模型。这些库的基础是强大的 Java 标准库，但它们一般更加复杂，学习和使用起来都更加困难。我们先来简单地了解一下这个模型。

在我们的模型中，Java 程序可以从*命令行参数*或者一个名为*标准输入流*的抽象字符流中获得输入，并将输出写入另一个名为*标准输出流*的字符流中。

我们需要考虑 Java 和操作系统之间的接口，因此我们要简要地讨论一下大多数操作系统和程序开发环境所提供的相应机制。本书网站上列出了关于你所使用的系统的更多信息。默认情况下，命令行参数、标准输入和标准输出是和应用程序绑定的，而应用程序是由能够接受命令输入的操作系统或是开发环境所支持。我们笼统地用*终端*来指代这个应用程序提供的供输入和显示的窗口。20 世纪 70 年代早期的 Unix 系统已经证明我们可以用这个模型方便直接地和程序以及数据进行交互。我们在经典的模型中加入了一个*标准绘图*模块用来可视化表示对数据的分析，如图 1.1.3 所示。

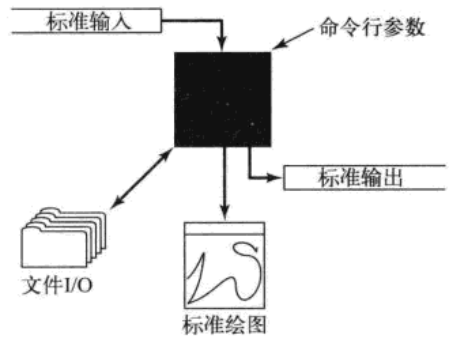


图 1.1.3 Java 程序整体结构

1.1.9.1 命令和参数

终端窗口包含一个提示符，通过它我们能够向操作系统输入命令和参数。本书中我们只会用到几个命令，如表 1.1.13 所示。我们会经常使用 java 命令来运行我们的程序。我们在 1.1.8.4 节中提到过，Java 类都会包含一个静态方法 main()，它有一个 String 数组类型的参数 args[]。这个数组的内容就是我们输入的命令行参数，操作系统会将它传递给 Java。Java 和操作系统都默认参数为字符串。如果我们需要的某个参数是数字，我们会使用类似 Integer.parseInt() 的方法将其转换为适当的数据类型的值。图 1.1.4 是对命令的分析。

表 1.1.13 操作系统常用命令

命　令	参　数	作　用
javac	.java 文件名	编译 Java 程序
java	.class 文件名（不需要扩展名）和命令行参数	运行 Java 程序
more	任意文本文件名	打印文件内容

36

1.1.9.2 标准输出

我们的 StdOut 库的作用是支持标准输出。一般来说，系统会将标准输出打印到终端窗口。print() 方法会将它的参数放到标准输出中；println() 方法会附加一个换行符；printf() 方法能够格式化输出（见 1.1.9.3 节）。Java 在其 System.out 库中提供了类似的方法，但我们会用 StdOut 库来统一处理标准输入和输出（并进行了一些技术上的改进），见表 1.1.4。

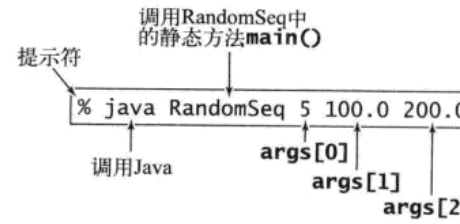


图 1.1.4 命令详解

表 1.1.14 我们的标准输出库的静态方法的 API

public class StdOut	
static void print(String s)	打印 s
static void println(String s)	打印 s 并接一个换行符
static void println()	打印一个换行符
static void printf(String f, ...)	格式化输出

注：其他原始类型和 Object 对象也有对应版本的方法。

要使用这些方法，请从本书的网站上将 StdOut.java 下载到你的工作目录，并像 StdOut.println("Hello, World"); 这样在代码中调用它们。左下方的程序就是一个例子。

1.1.9.3 格式化输出

在最简单的情况下 printf() 方法接受两个参数。第一个参数是一个格式字符串，描述了第二个参数应该如何在输出中被转换为一个字符串。最简单的格式字符串的第一个字符是 % 并紧跟一个字符表示的转换代码。我们最常使用的转换代码包括 d（用于 Java 整型的十进制数）、f（浮点型）和 s（字符串）。在 % 和转换代码之间可以插入一个整数来表示转换之后的值的宽度，即输出字符串的长度。默认情况下，转换后会在字符串的左边添加空格以达到需要的宽度，如果我们想在右边加入空格则应该使用负宽度（如果转换得到的字符串比设定宽度要长，宽度会被忽略）。在宽度之后我们还可以插入一个小数点以及一个数值来指定转换后的 double 值保留的小数位数（精度）或是 String 字符串所截取的长度。使用 printf() 方法时需要记住的最重要的一点就是，格式字符串中的转换代码和对应参数的数据类型必须匹配。也就是说，Java 要求参数的数据类型和转换代码表示的数据类型必须相同。printf() 的第一个 String 字符串参数也可以包含其他字符。所有非格式字符串的

```
public class RandomSeq
{
   public static void main(String[] args)
   {  // 打印N个(lo, hi)之间的随机值
      int N = Integer.parseInt(args[0]);
      double lo = Double.parseDouble(args[1]);
      double hi = Double.parseDouble(args[2]);
      for (int i = 0; i < N; i++)
      {
         double x = StdRandom.uniform(lo, hi);
         StdOut.printf("%.2f\n", x);
      }
   }
}
```

StdOut 的用例示例

```
% java RandomSeq 5 100.0 200.0
123.43
153.13
144.38
155.18
104.02
```

字符都会被传递到输出之中，而格式字符串则会被参数的值所替代（按照指定的方式转换为字符串）。例如，这条语句：

```
StdOut.printf("PI is approximately %.2f\n", Math.PI);
```

会打印出：

```
PI is approximately 3.14
```

可以看到，在printf()中我们需要明确地在第一个参数的末尾加上 \n 来换行。printf() 函数能够接受两个或者更多的参数。在这种情况下，在格式化字符串中每个参数都会有对应的转换代码，这些代码之间可能隔着其他会被直接传递到输出中的字符。也可以直接使用静态方法 String.format() 来用和 printf() 相同的参数得到一个格式化字符串而无需打印它。我们可以用格式化打印方便地将实验数据输出为表格形式（这是它们在本书中的主要用途），如表 1.1.15 所示。

表 1.1.15 printf() 的格式化方式（更多选项请见本书网站）

数据类型	转换代码	举 例	格式化字符串举例	转换后输出的字符串
int	d	512	"%14d"	" 512"
			"%-14d"	"512 "
double	f	1595.1680010754388	"%14.2f"	" 1595.17"
			"%.7f"	"1595.1680011"
	e		"%14.4e"	" 1.5952e+03"
String	s	"Hello, World"	"%14s"	" Hello, World"
			"%-14s"	"Hello, World "
			"%14.5s"	"Hello "

38

1.1.9.4 标准输入

我们的 StdIn 库从标准输入流中获取数据，这些数据可能为空也可能是一系列由空白字符分隔的值（空格、制表符、换行符等）。默认状态下系统会将标准输出定向到终端窗口——你输入的内容就是输入流（由 <ctrl-d> 或 <ctrl-z> 结束，取决于你使用的终端应用程序）。这些值可能是 String 或是 Java 的某种原始类型的数据。标准输入流最重要的特点是这些值会在你的程序读取它们之后消失。只要程序读取了一个值，它就不能回退并再次读取它。这个特点产生了一些限制，但它反映了一些输入设备的物理特性并简化了对这些设备的抽象。有了输入流模型，这个库中的静态方法大都是自文档化的（它们的签名即说明了它们的用途）。右侧列出了 StdIn 的一个用例。

```
public class Average
{
   public static void main(String[] args)
   {  // 取StdIn中所有数的平均值
      double sum = 0.0;
      int cnt = 0;
      while (!StdIn.isEmpty())
      {  // 读取一个数并计算累计之和
         sum += StdIn.readDouble();
         cnt++;
      }
      double avg = sum / cnt;
      StdOut.printf("Average is %.5f\n", avg);
   }
}
```

StdIn 的用例举例

```
% java Average
1.23456
2.34567
3.45678
4.56789
<ctrl-d>
Average is 2.90123
```

表 1.1.16 详细说明了标准输入库中的静态方法的 API。

表 1.1.16 标准输入库中的静态方法的 API

Public class StdIn		
static boolean	isEmpty()	如果输入流中没有剩余的值则返回 true，否则返回 false
static int	readInt()	读取一个 int 类型的值
static double	readDouble()	读取一个 double 类型的值
static float	readFloat()	读取一个 float 类型的值
static long	readLong()	读取一个 long 类型的值
static boolean	readBoolean()	读取一个 boolean 类型的值
static char	readChar()	读取一个 char 类型的值
static byte	readByte()	读取一个 byte 类型的值
static String	readString()	读取一个 String 类型的值
static boolean	hasNextLine()	输入流中是否还有下一行
static String	readLine()	读取该行的其余内容
static String	readAll()	读取输入流中的其余内容

39

1.1.9.5 重定向与管道

标准输入输出使我们能够利用许多操作系统都支持的命令行的扩展功能。只需要向启动程序的命令中加入一个简单的提示符，就可以将它的标准输出重定向至一个文件。文件的内容既可以永久保存也可以在之后作为另一个程序的输入：

```
% java RandomSeq 1000 100.0 200.0 > data.txt
```

这条命令指明标准输出流不是被打印至终端窗口，而是被写入一个叫做 data.txt 的文件。每次调用 StdOut.print() 或是 StdOt.println() 都会向该文件追加一段文本。在这个例子中，我们最后会得到一个含有 1000 个随机数的文件。终端窗口中不会出现任何输出：它们都被直接写入了“>”号之后的文件中。这样我们就能将信息存储以备下次使用。请注意不需要改变 RandomSeq 的任何内容——它使用的是标准输出的抽象，因此它不会因为我们使用了该抽象的另一种不同的实现而受到影响。类似，我们可以重定向标准输入以使 StdIn 从文件而不是终端应用程序中读取数据：

```
% java Average < data.txt
```

这条命令会从文件 data.txt 中读取一系列数值并计算它们的平均值。具体来说，“<”号是一个提示符，它告诉操作系统读取文本文件 data.txt 作为输入流而不是在终端窗口中等待用户的输入。当程序调用 StdIn.readDouble() 时，操作系统读取的是文件中的值。将这些结合起来，将一个程序的输出重定向为另一个程序的输入叫做管道：

40

```
% java RandomSeq 1000 100.0 200.0 | java Average
```

这条命令将 RandomSeq 的标准输出和 Average 的标准输入指定为同一个流。它的效果是好像在 Average 运行时 RandomSeq 将它生成的数字输入了终端窗口。这种差别影响非常深远，因为它突破了我们能够处理的输入输出流的长度限制。例如，即使计算机没有足够的空间来存储十亿个数，我们仍然可以将例子中的 1000 换成 1 000 000 000（当然我们还是需要一些时间来处理它们）。当 RandomSeq 调用 StdOut.println() 时，它就向输出流的末尾添加了一个字符串；当 Average 调用 StdIn.readInt() 时，它就从输入流的开头删除了一个字符串。这些动作发生的实际顺序取决于操作系统：它可能会先运行 RandomSeq 并产生一些输出，然后再运行 Average，来消耗这些输出，或者它也可以先运行 Average，直到它需要一些输入然后再运行 RandomSeq 来产生一些输出。虽然

最后的结果都一样，但我们的程序就不再需要担心这些细节，因为它们只会和标准输入和标准输出的抽象打交道。

图 1.1.5 总结了重定向与管道的过程。

1.1.9.6 基于文件的输入输出

我们的 In 和 Out 库提供了一些静态方法，来实现向文件中写入或从文件中读取一个原始数据类型（或 String 类型）的数组的抽象。我们会使用 In 库中的 readInts()、readDoubles() 和 readStrings() 以及 Out 库中的 writeInts()、writeDoubles() 和 writeStrings() 方法，参数可以是文件或网页如表 1.1.17 所示。例如，借此我们可以在同一个程序中分别使用文件和标准输入达到两种不同的目的，例如 BinarySearch。In 和 Out 两个库也实现了一些数据类型和它们的实例方法，这使我们能够将多个文件作为输入输出流并将网页作为输入流，我们还会在 1.2 节中再次考察它们。

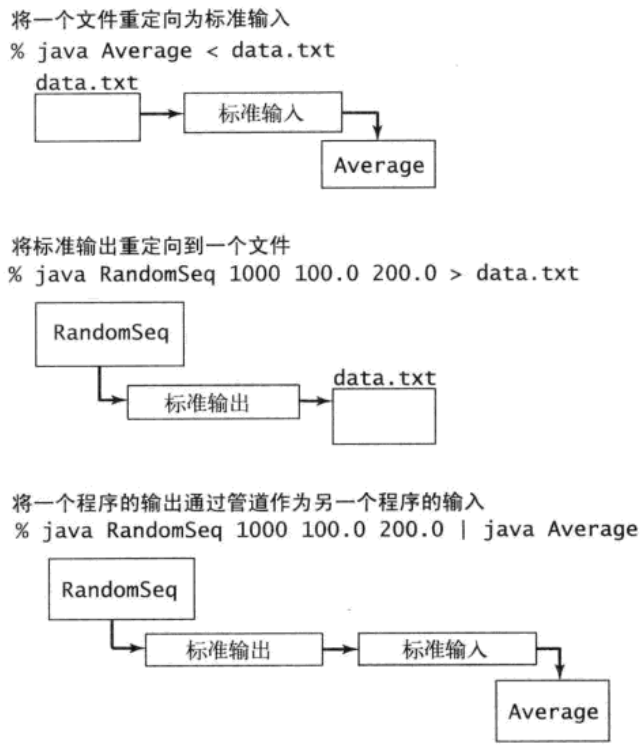


图 1.1.5 命令行的重定向与管道

表 1.1.17 我们用于读取和写入数组的静态方法的 API

public class **In**		
static int[]	readInts(String name)	读取多个 int 值
static double[]	readDoubles(String name)	读取多个 double 值
static String[]	readStrings(String name)	读取多个 String 值
public class **Out**		
static void	write(int[], String name)	写入多个 int 值
static void	write(doule[] a, String name)	写入多个 double 值
static void	write(String[] a, String name)	写入多个 String 值

注 1：库也支持其他原始数据类型。
注 2：库也支持 StdIn 和 StdOut（忽略 name 参数）。

41

1.1.9.7 标准绘图库（基本方法）

目前为止，我们的输入输出抽象层的重点只有文本字符串。现在我们要介绍一个产生图像输出的抽象层。这个库的使用非常简单并且允许我们利用可视化的方式处理比文字丰富得多的信息。和我们的标准输入输出一样，标准绘图抽象层实现在库 StdDraw 中，可以从本书的网站上下载 StdDraw.java 到你的工作目录来使用它。标准绘图库很简单：我们可以将它想象为一个抽象的能够在二维画布上画出点和直线的绘图设备。这个设备能够根据程序调用的 StdDraw 中的静态方法画出

一些基本的几何图形，这些方法包括画出点、直线、文本字符串、圆、长方形和多边形等。和标准输入输出中的方法一样，这些方法几乎也都是自文档化的：StdDraw.line() 能够根据参数的坐标画出一条连接点 (x_0, y_0) 和点 (x_1, y_1) 的线段，StdDraw.point() 能够根据参数坐标画出一个以 (x, y) 为中心的点，等等，如图 1.1.6 所示。几何图形可以被填充（默认为黑色）。默认的比例尺为单位正方形（所有的坐标均在 0 和 1 之间）。标准的实现会将画布显示为屏幕上的一个窗口，点和线为黑色，背景为白色。

42

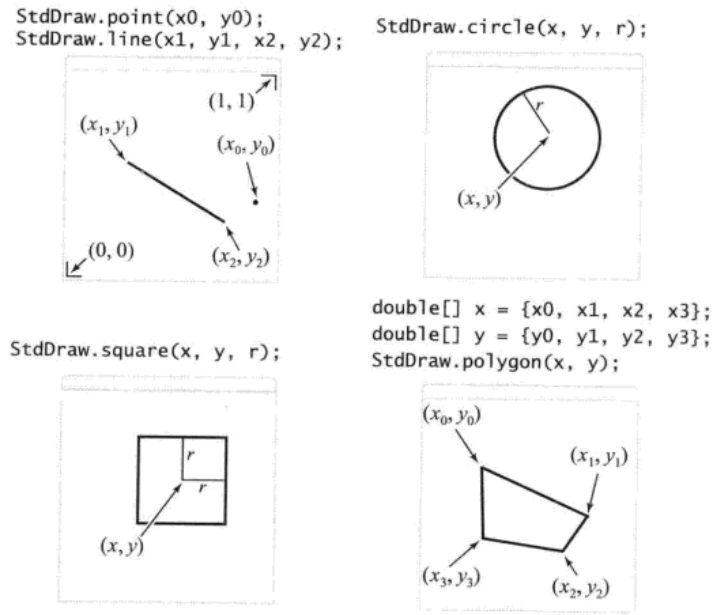


图 1.1.6 StdDraw 的用法举例

表 1.1.18 是对标准绘图库中静态方法 API 的汇总。

表 1.1.18 标准绘图库的静态（绘图）方法的 API

public class **StdDraw**
static void line(double x0, double y0, double x1, double y1)
static void point(double x, double y)
static void text(double x, double y, String s)
static void circle(double x, double y, double r)
static void filledCircle(double x, double y, double r)
static void ellipse(double x, double y, double rw, double rh)
static void filledEllipse(double x, double y, double rw, double rh)
static void square(double x, double y, double r)
static void filledSquare(double x, double y, double r)
static void rectangle(double x, double y, double rw, double rh)
static void filledRectangle(double x, double y, double rw, double rh)
static void polygon(double[] x, double[] y)
static void filledPolygon(double[] x, double[] y)

1.1.9.8 标准绘图库（控制方法）

标准绘图库中还包含一些方法来改变画布的大小和比例、直线的颜色和宽度、文本字体、绘图时间（用于动画）等。可以使用在 StdDraw 中预定义的 BLACK、BLUE、CYAN、DARK_GRAY、GRAY、GREEN、LIGHT_GRAY、MAGENTA、ORANGE、PINK、RED、BOOK_RED、WHITE 和 YELLOW 等颜色常数作为 setPenColor() 方法的参数（可以用 StdDraw.RED 这样的方式调用它们）。画布窗口的菜单还包含一个选项用于将图像保存为适于在网上传播的文件格式。表 1.1.19 总结了 StdDraw 中静态控制方法的 API。

表 1.1.19 标准绘图库的静态（控制）方法的 API

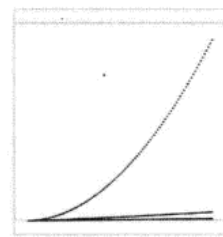
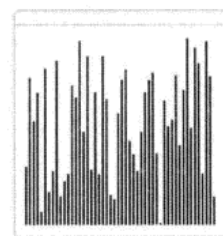
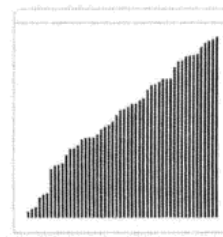
public class **StdDraw**	
static void setXscale(double x0, double x1)	将 x 的范围设为 (x_0, x_1)
static void setYscale(double y0, double y1)	将 y 的范围设为 (y_0, y_1)
static void setPenRadius(double r)	将画笔的粗细半径设为 r
static void setPenColor(Color c)	将画笔的颜色设为 c
static void setFont(Font f)	将文本字体设为 f
static void setCanvasSize(int w, int h)	将画布窗口的宽和高分别设为 w 和 h
static void clear(Color c)	清空画布并用颜色 c 将其填充
static void show(int dt)	显示所有图像并暂停 dt 毫秒

43

在本书中，我们会在数据分析和算法的可视化中使用 StdDraw。表 1.1.20 是一些例子，我们在本书的其他章节和练习中还会遇到更多的例子。绘图库也支持动画——当然，这个话题只能在本书的网站上展开了。

44

表 1.1.20 StdDraw 绘图举例

数 据	绘图的实现（代码片段）	结 果
函数值	`int N = 100;` `StdDraw.setXscale(0, N);` `StdDraw.setYscale(0, N*N);` `StdDraw.setPenRadius(.01);` `for (int i = 1; i <= N; i++)` `{` `  StdDraw.point(i, i);` `  StdDraw.point(i, i*i);` `  StdDraw.point(i, i*Math.log(i));` `}`	
随机数组	`int N = 50;` `double[] a = new double[N];` `for (int i = 0; i < N; i++)` `  a[i] = StdRandom.random();` `for (int i = 0; i < N; i++)` `{` `  double x = 1.0*i/N;` `  double y = a[i]/2.0;` `  double rw = 0.5/N;` `  double rh = a[i]/2.0;` `  StdDraw.filledRectangle(x, y, rw, rh);` `}`	
已排序的随机数组	`int N = 50;` `double[] a = new double[N];` `for (int i = 0; i < N; i++)` `  a[i] = StdRandom.random();` `Arrays.sort(a);` `for (int i = 0; i < N; i++)` `{` `  double x = 1.0*i/N;` `  double y = a[i]/2.0;` `  double rw = 0.5/N;` `  double rh = a[i]/2.0;` `  StdDraw.filledRectangle(x, y, rw, rh);` `}`	

45

1.1.10 二分查找

我们要学习的第一个 Java 程序的示例程序就是著名、高效并且应用广泛的二分查找算法，如下所示。这个例子将会展示本书中学习新算法的基本方法。和我们将要学习的所有程序一样，它既是算法的准确定义，又是算法的一个完整的 Java 实现，而且你还能够从本书的网站上下载它。

二分查找

```
import java.util.Arrays;
public class BinarySearch
{
   public static int rank(int key, int[] a)
   {  // 数组必须是有序的
      int lo  = 0;
      int hi = a.length - 1;
      while (lo <= hi)
      {  // 被查找的键要么不存在，要么必然存在于 a[lo..hi] 之中
         int mid = lo + (hi - lo) / 2;
         if      (key < a[mid]) hi = mid - 1;
         else if (key > a[mid]) lo = mid + 1;
         else                   return mid;
      }
      return -1;
   }
   public static void main(String[] args)
   {
      int[] whitelist = In.readInts(args[0]);
      Arrays.sort(whitelist);
      while (!StdIn.isEmpty())
      {  // 读取键值，如果不存在于白名单中则将其打印
         int key = StdIn.readInt();
         if (rank(key, whitelist) < 0)
            StdOut.println(key);
      }
   }
}
```

这段程序接受一个白名单文件（一列整数）作为参数，并会过滤掉标准输入中的所有存在于白名单中的条目，仅将不在白名单上的整数打印到标准输出中。它在 rank() 静态方法中实现了二分查找算法并高效地完成了这个任务。

```
% java BinarySearch tinyW.txt < tinyT.txt
50
99
13
```

关于二分查找算法的完整讨论，包括它的正确性、性能分析及其应用，请见 3.1 节。

1.1.10.1 二分查找

我们会在 3.2 节中详细学习二分查找算法，但此处先简单地描述一下。算法是由静态方法 rank() 实现的，它接受一个整数键和一个已经有序的 int 数组作为参数。如果该键存在于数组中则返回它的索引，否则返回 -1。算法使用两个变量 lo 和 hi，并保证如果键在数组中则它一定在 a[lo..hi] 中，然后方法进入一个循环，不断将数组的中间键（索引为 mid）和被查找的键比较。如果被查找的键等于 a[mid]，返回 mid；否则算法就将查找范围缩小一半，如果被查找的键小于 a[mid] 就继续在左半边查找，如果被查找的键大于 a[mid] 就继续在右半边查找。算法找到被查找的键或是查找范围为空时该过程结束。二分查找之所以快是因为它只需检查很少几个条目（相对于数组的大小）就能够找到目标元素（或者确认目标元素不存在）。在有序数组中进行二分查找的示

例如图 1.1.7 所示。

1.1.10.2 开发用例

对于每个算法的实现，我们都会开发一个用例 main() 函数，并在书中或是本书的网站上提供一个示例输入文件来帮助读者学习该算法并检测它的性能。在这个例子中，这个用例会从命令行指定的文件中读取多个整数，并会打印出标准输入中所有不存在于该文件中的整数。我们使用了图 1.1.8 所示的几个较小的测试文件来展示它的行为，这些文件也是图 1.1.7 中的跟踪和例子的基础。我们会使用较大的测试文件来模拟真实应用并测试算法的性能（请见 1.1.10.3 节）。

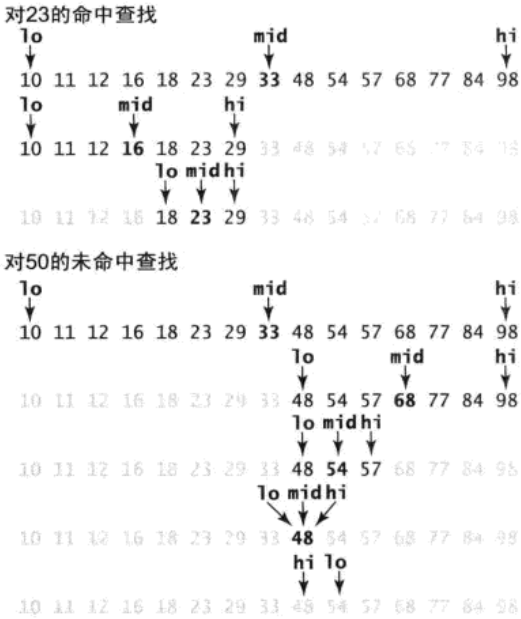


图 1.1.7 有序数组中的二分查找

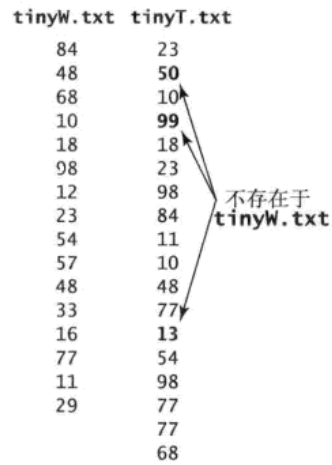


图 1.1.8 为 BinarySearch 的测试用例准备的小型测试文件

46
|
47

1.1.10.3 白名单过滤

如果可能，我们的测试用例都会通过模拟实际情况来展示当前算法的必要性。这里该过程被称为*白名单过滤*。具体来说，可以想象一家信用卡公司，它需要检查客户的交易账号是否有效。为此，它需要：

- 将客户的账号保存在一个文件中，我们称它为*白名单*；
- 从标准输入中得到每笔交易的账号；
- 使用这个测试用例在标准输出中打印所有与任何客户无关的账号，公司很可能拒绝此类交易。

在一家有上百万客户的大公司中，需要处理数百万甚至更多的交易都是很正常的。为了模拟这种情况，我们在本书的网站上提供了文件 largeW.txt（100 万个整数）和 largeT.txt（1000 万个整数）其基本情况如图 1.1.9 所示。

1.1.10.4 性能

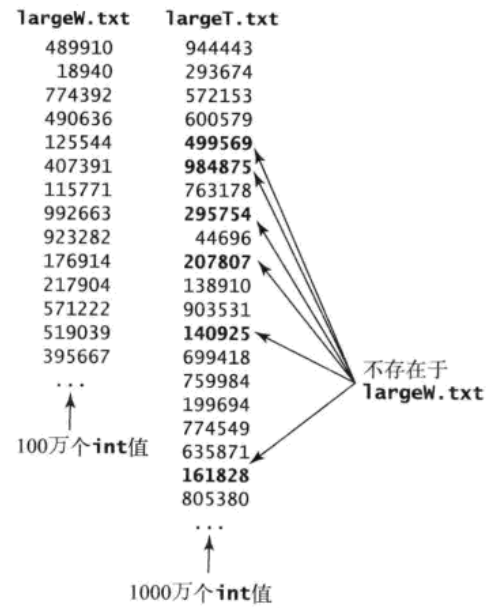
一个程序只是可用往往是不够的。例如，以下 rank() 的实现也可以很简单，它会检查数组的每个元素，甚至都不需要数组是有序的：

```
public static int rank(int key, int[] a)
{
   for (int i = 0; i < a.length; i++)
      if (a[i] == key) return i;
   return -1;
}
```

有了这个简单易懂的解决方案，我们为什么还需要归并排序和二分查找呢？你在完成练习 1.1.38 时会看到，计算机用 rank() 方法的暴力实现处理大量输入（比如含有 100 万个条目的白名单和 1000 万条交易）非常慢。*没有如二分查找或者归并排序这样的高效算法，解决大规模的白名单问题是不可能的。*良好的性能常常是极为重要的，因此我们会在 1.4 节中为性能研究做一些铺垫，并会分析我们学习的所有算法的性能特点（包括 2.2 节的归并排序和 3.1 节中的二分查找）。

目前，我们在这里粗略地勾勒出我们的编程模型的目标是，确保你能够在计算机上运行类似于 BinarySearch 的代码，使用它处理我们的测试数据并为适应各种情况修改它（比如本节练习中所描述的一些情况）以完全理解它的可应用性。我们的编程模型就是设计用来简化这些活动的，这对各种算法的学习至关重要。

48
~
49




```
% java BinarySearch largeW.txt < largeT.txt
499569
984875
295754
207807
140925
161828
...
```

3 675 966个int值

图 1.1.9 为 BinarySearch 测试用例准备的大型文件

1.1.11 展望

在本节中，我们描述了一个精巧而完整的编程模型，数十年来它一直在（并且现在仍在）为广大程序员服务。但现代编程技术已经更进一步。前进的这一步被称为*数据抽象*，有时也被称为*面向对象编程*，它是我们下一节的主题。简单地说，数据抽象的主要思想是鼓励程序定义自己的*数据类型*（一系列值和对这些值的操作），而不仅仅是那些操作预定义的数据类型的静态方法。

面向对象编程在最近几十年得到了广泛的应用，数据抽象已经成为现代程序开发的核心。我们在本书中“拥抱”数据抽象的原因主要有三。

- 它允许我们通过模块化编程复用代码。例如，第 2 章中的排序算法和第 3 章中的二分查找以及其他算法，都允许调用者用同一段代码处理任意类型的数据（而不仅限于整数），包括调用者自定义的数据类型。
- 它使我们可以轻易构造多种所谓的*链式数据结构*，它们比数组更灵活，在许多情况下都是高效算法的基础。
- 借助它我们可以准确地定义所面对的算法问题。比如 1.5 节中的 union-find 算法、2.4 节中的优先队列算法和第 3 章中的符号表算法，它们解决问题的方式都是定义数据结构并高效地实现它们的一组操作。这些问题都能够用数据抽象很好地解决。

尽管如此，但我们的重点仍然是对算法的研究。在了解了这些知识以后，我们将学习面向对象编程中和我们的使命相关的另一个重要特性。 50

答疑

问 什么是 Java 的字节码？

答 它是程序的一种低级表示，可以运行于 Java 的虚拟机。将程序抽象为字节码可以保证 Java 程序员的代码能够运行在各种设备之上。

问 Java 允许整型溢出并返回错误值的做法是错误的。难道 Java 不应该自动检查溢出吗？

答 这个问题在程序员中一直是有争议的。简单的回答是它们之所以被称为原始数据类型就是因为缺乏此类检查。避免此类问题并不需要很高深的知识。我们会使用 `int` 类型表示较小的数（小于 10 个十进制位）而使用 `long` 表示 10 亿以上的数。

问 `Math.abs(-2147483648)` 的返回值是什么？

答 `-2147483648`。这个奇怪的结果（但的确是真的）就是整数溢出的典型例子。

问 如何才能将一个 `double` 变量初始化为无穷大？

答 可以使用 Java 的内置常数：`Double.POSITIVE_INFINITY` 和 `Double.NEGATIVE_INFINITY`。

问 能够将 `double` 类型的值和 `int` 类型的值相互比较吗？

答 不通过类型转换是不行的，但请记住 Java 一般会自动进行所需的类型转换。例如，如果 `x` 的类型是 `int` 且值为 `3`，那么表达式 `(x<3.1)` 的值为 `true`——Java 会在比较前将 `x` 转换为 `double` 类型（因为 `3.1` 是一个 `double` 类型的字面量）。

问 如果使用一个变量前没有将它初始化，会发生什么？

答 如果代码中存在任何可能导致使用未经初始化的变量的执行路径，Java 都会抛出一个编译异常。

问 Java 表达式 `1/0` 和 `1.0/0.0` 的值是什么？

答 第一个表达式会产生一个运行时除零异常（它会终止程序，因为这个值是未定义的）；第二个表达式的值是 `Infinity`（无穷大）。 51

问 能够使用 < 和 > 比较 `String` 变量吗？

答 不行，只有原始数据类型定义了这些运算符。请见 1.1.2.3 节。

问 负数的除法和余数的结果是什么？

答 表达式 `a/b` 的商会向 0 取整；`a % b` 的余数的定义是 `(a/b)*b + a % b` 恒等于 `a`。例如 `-14/3` 和 `14/-3` 的商都是 `-4`，但 `-14 % 3` 是 `-2`，而 `14 % -3` 是 `2`。

问 为什么使用 `(a && b)` 而非 `(a & b)`？

答 运算符 `&`、`|` 和 `^` 分别表示整数的位逻辑操作与、或和异或。因此，`10|6` 的值为 `14`，`10^6` 的值为 `12`。在本书中我们很少（偶尔）会用到这些运算符。`&&` 和 `||` 运算符仅在独立的布尔表达式中有效，原因是短路求值法则：表达式从左向右求值，一旦整个表达式的值已知则停止求值。

问 嵌套 `if` 语句中的二义性有问题吗？

答 是的。在 Java 中，以下语句：

```
if <expr1> if <expr2> <stmntA> else <stmntB>
```

等价于：

```
if <expr1> { if <expr2> <stmntA> else <stmntB> }
```

即使你想表达的是：

```
if <expr1> { if <expr2> <stmntA> } else <stmntB>
```

避免这种“无主的”else陷阱的最好办法是显式地写明所有大括号。

问 一个for循环和它的while形式有什么区别？

答 for循环头部的代码和for循环的主体代码在同一个代码段之中。在一个典型的for循环中，递增变量一般在循环结束之后都是不可用的；但在和它等价的while循环中，递增变量在循环结束之后仍然是可用的。这个区别常常是使用while而非for循环的主要原因。

52

问 有些Java程序员用int a[]而不是int[] a来声明一个数组。这两者有什么不同？

答 在Java中，两者等价且都是合法的。前一种是C语言中数组的声明方式。后者是Java提倡的方式，因为变量的类型int[]能更清楚地说明这是一个整型的数组。

问 为什么数组的起始索引是0而不是1？

答 这个习惯来源于机器语言，那时要计算一个数组元素的地址需要将数组的起始地址加上该元素的索引。将起始索引设为1要么会浪费数组的第一个元素的空间，要么会花费额外的时间来将索引减1。

问 如果a[]是一个数组，为什么StdOut.println(a)打印出的是一个十六进制的整数，比如@f62373，而不是数组中的元素呢？

答 问得好。该方法打印出的是这个数组的地址，不幸的是你一般都不需要它。

问 我们为什么不使用标准的Java库来处理输入和图形？

答 我们的确用到了它们，但我们希望使用更简单的抽象模型。StdIn和StdDraw背后的Java标准库是为实际生产设计的，这些库和它们的API都有些笨重。要想知道它们真正的模样，请查看StdIn.java和StdDraw.java的代码。

问 我的程序能够重新读取标准输入中的值吗？

答 不行，你只有一次机会，就好像你不能撤销println()的结果一样。

问 如果我的程序在标准输入为空之后仍然尝试读取，会发生什么？

答 会得到一个错误。StdIn.isEmpty()能够帮助你检查是否还有可用的输入以避免这种错误。

问 这条出错信息是什么意思？

```
Exception in thread "main" java.lang.NoClassDefFoundError: StdIn
```

答 你可能忘记把StdIn.java文件放到工作目录中去了。

问 在Java中，一个静态方法能够将另一个静态方法作为参数吗？

答 不行，但问得好，因为有很多语言都能够这么做。

53

练习

1.1.1 给出以下表达式的值：

a. (0 + 15) / 2

b. 2.0e-6 * 100000000.1

c. true && false || true && true

1.1.2 给出以下表达式的类型和值：

a. (1 + 2.236)/2

b. 1 + 2 + 3 + 4.0



c. `4.1 >= 4`

d. `1 + 2 + "3"`

1.1.3 编写一个程序，从命令行得到三个整数参数。如果它们都相等则打印 `equal`，否则打印 `not equal`。

1.1.4 下列语句各有什么问题（如果有的话）？

a. `if (a > b) then c = 0;`

b. `if a > b { c = 0; }`

c. `if (a > b) c = 0;`

d. `if (a > b) c = 0 else b = 0;`

1.1.5 编写一段程序，如果 `double` 类型的变量 `x` 和 `y` 都严格位于 0 和 1 之间则打印 `true`，否则打印 `false`。

1.1.6 下面这段程序会打印出什么？

```
int f = 0;
int g = 1;
for (int i = 0; i <= 15; i++)
{
   StdOut.println(f);
   f = f + g;
   g = f - g;
}
```

54

1.1.7 分别给出以下代码段打印出的值：

a.
```
double t = 9.0;
while (Math.abs(t - 9.0/t) > .001)
   t = (9.0/t + t) / 2.0;
StdOut.printf("%.5f\n", t);
```

b.
```
int sum = 0;
for (int i = 1; i < 1000; i++)
   for (int j = 0; j < i; j++)
      sum++;
StdOut.println(sum);
```

c.
```
int sum = 0;
for (int i = 1; i < 1000; i *= 2)
   for (int j = 0; j < 1000; j++)
      sum++;
StdOut.println(sum);
```

1.1.8 下列语句会打印出什么结果？给出解释。

a. `System.out.println('b');`

b. `System.out.println('b' + 'c');`

c. `System.out.println((char) ('a' + 4));`

1.1.9 编写一段代码，将一个正整数 `N` 用二进制表示并转换为一个 `String` 类型的值 `s`。

解答：Java 有一个内置方法 `Integer.toBinaryString(N)` 专门完成这个任务，但该题的目的就是给出这个方法的其他实现方法。下面就是一个特别简洁的答案：

```
String s = "";
for (int n = N; n > 0; n /= 2)
   s = (n % 2) + s;
```

55

1.1.10 下面这段代码有什么问题？

```
int[] a;
for (int i = 0; i < 10; i++)
   a[i] = i * i;
```

解答：它没有用 new 为 a[] 分配内存。这段代码会产生一个 variable a might not have been initialized 的编译错误。

1.1.11 编写一段代码，打印出一个二维布尔数组的内容。其中，使用 * 表示真，空格表示假。打印出行号和列号。

1.1.12 以下代码段会打印出什么结果？

```
int[] a = new int[10];
for (int i = 0; i < 10; i++)
   a[i] = 9 - i;
for (int i = 0; i < 10; i++)
   a[i] = a[a[i]];
for (int i = 0; i < 10; i++)
   System.out.println(i);
```

1.1.13 编写一段代码，打印出一个 M 行 N 列的二维数组的转置（交换行和列）。

1.1.14 编写一个静态方法 lg()，接受一个整型参数 N，返回不大于 $\log_2 N$ 的最大整数。不要使用 Math 库。

1.1.15 编写一个静态方法 histogram()，接受一个整型数组 a[] 和一个整数 M 为参数并返回一个大小为 M 的数组，其中第 i 个元素的值为整数 i 在参数数组中出现的次数。如果 a[] 中的值均在 0 到 M-1 之间，返回数组中所有元素之和应该和 a.length 相等。

1.1.16 给出 exR1(6) 的返回值：

```
public static String exR1(int n)
{
   if (n <= 0) return "";
   return exR1(n-3) + n + exR1(n-2) + n;
}
```

56

1.1.17 找出以下递归函数的问题：

```
public static String exR2(int n)
{
   String s = exR2(n-3) + n + exR2(n-2) + n;
   if (n <= 0) return "";
   return s;
}
```

答：这段代码中的基础情况永远不会被访问。调用 exR2(3) 会产生调用 exR2(0)、exR2(-3) 和 exR2(-6)，循环往复直到发生 StackOverflowError。

1.1.18 请看以下递归函数：

```
public static int mystery(int a, int b)
{
   if (b == 0)      return 0;
   if (b % 2 == 0) return mystery(a+a, b/2);
   return mystery(a+a, b/2) + a;
}
```

mystery(2, 25) 和 mystery(3, 11) 的返回值是多少？给定正整数 a 和 b，mystery(a,b) 计算的结果是什么？将代码中的 + 替换为 * 并将 return 0 改为 return 1，然后回答相同的问题。

1.1.19 在计算机上运行以下程序：

```
public class Fibonacci
{
   public static long F(int N)
   {
      if (N == 0) return 0;
      if (N == 1) return 1;
      return F(N-1) + F(N-2);
   }
   public static void main(String[] args)
   {
      for (int N = 0; N < 100; N++)
         StdOut.println(N + " " + F(N));
   }
}
```

57

计算机用这段程序在一个小时之内能够得到 F(N) 结果的最大 N 值是多少？开发 F(N) 的一个更好的实现，用数组保存已经计算过的值。

1.1.20 编写一个递归的静态方法计算 ln(*N*!) 的值。

1.1.21 编写一段程序，从标准输入按行读取数据，其中每行都包含一个名字和两个整数。然后用 printf() 打印一张表格，每行的若干列数据包括名字、两个整数和第一个整数除以第二个整数的结果，精确到小数点后三位。可以用这种程序将棒球球手的击球命中率或者学生的考试分数制成表格。

1.1.22 使用 1.1.6.4 节中的 rank() 递归方法重新实现 BinarySearch 并跟踪该方法的调用。每当该方法被调用时，打印出它的参数 lo 和 hi 并按照递归的深度缩进。*提示*：为递归方法添加一个参数来保存递归的深度。

1.1.23 为 BinarySearch 的测试用例添加一个参数：+ 打印出标准输入中不在白名单上的值；-，则打印出标准输入中在白名单上的值。

1.1.24 给出使用欧几里德算法计算 105 和 24 的最大公约数的过程中得到的一系列 p 和 q 的值。扩展该算法中的代码得到一个程序 Euclid，从命令行接受两个参数，计算它们的最大公约数并打印出每次调用递归方法时的两个参数。使用你的程序计算 1 111 111 和 1 234 567 的最大公约数。

1.1.25 使用数学归纳法证明欧几里德算法能够计算任意一对非负整数 p 和 q 的最大公约数。

58

提高题

1.1.26 将三个数字排序。假设 a、b、c 和 t 都是同一种原始数字类型的变量。证明以下代码能够将 a、b、c 按照升序排列：

```
if (a > b) { t = a; a = b; b = t; }
if (a > c) { t = a; a = c; c = t; }
if (b > c) { t = b; b = c; c = t; }
```

1.1.27 二项分布。估计用以下代码计算 binomial(100, 50) 将会产生的递归调用次数：

```
public static double binomial(int N, int k, double p)
{
   if (N == 0 && k == 0) return 1.0; and if (N < 0 || k < 0) return 0.0;
   return (1.0 - p)*binomial(N-1, k, p) + p*binomial(N-1, k-1);
}
```

将已经计算过的值保存在数组中并给出一个更好的实现。

1.1.28 删除重复元素。修改 BinarySearch 类中的测试用例来删去排序之后白名单中的所有重复元素。

1.1.29 等值键。为 BinarySearch 类添加一个静态方法 rank()，它接受一个键和一个整型有序数组（可能存在重复键）作为参数并返回数组中小于该键的元素数量，以及一个类似的方法 count() 来返回数组中等于该键的元素的数量。注意：如果 i 和 j 分别是 rank(key,a) 和 count(key,a) 的返回值，那么 a[i..i+j-1] 就是数组中所有和 key 相等的元素。

1.1.30 数组练习。编写一段程序，创建一个 $N \times N$ 的布尔数组 a[][]。其中当 i 和 j 互质时（没有相同因子），a[i][j] 为 true，否则为 false。

1.1.31 随机连接。编写一段程序，从命令行接受一个整数 N 和 double 值 p（0 到 1 之间）作为参数，
59 在一个圆上画出大小为 0.05 且间距相等的 N 个点，然后将每对点按照概率 p 用灰线连接。

1.1.32 直方图。假设标准输入流中含有一系列 double 值。编写一段程序，从命令行接受一个整数 N 和两个 double 值 l 和 r。将 (l, r) 分为 N 段并使用 StdDraw 画出输入流中的值落入每段的数量的直方图。

1.1.33 矩阵库。编写一个 Matrix 库并实现以下 API：

public class **Matrix**		
static double	dot(double[] x, double[] y)	向量点乘
static double[][]	mult(double[][] a, double[][] b)	矩阵和矩阵之积
static double[][]	transpose(double[][] a)	转置矩阵
static double[]	mult(double[][] a, double[] x)	矩阵和向量之积
static double[]	mult(double[] y, double[][] a)	向量和矩阵之积

编写一个测试用例，从标准输入读取矩阵并测试所有方法。

1.1.34 过滤。以下哪些任务需要（在数组中，比如）保存标准输入中的所有值？哪些可以被实现为一个过滤器且仅使用固定数量的变量和固定大小的数组（和 N 无关）？在每个问题中，输入都来自于标准输入且含有 N 个 0 到 1 的实数。

- ❑ 打印出最大和最小的数
- ❑ 打印出所有数的中位数
- ❑ 打印出第 k 小的数，k 小于 100
- ❑ 打印出所有数的平方和
- ❑ 打印出 N 个数的平均值
- ❑ 打印出大于平均值的数的百分比
- ❑ 将 N 个数按照升序打印
- 60 ❑ 将 N 个数按照随机顺序打印

实验题

1.1.35 模拟掷骰子。以下代码能够计算每种两个骰子之和的准确概率分布：

```
int SIDES = 6;
double[] dist = new double[2*SIDES+1];
for (int i = 1; i <= SIDES; i++)
   for (int j = 1; j <= SIDES; j++)
      dist[i+j] += 1.0;

for (int k = 2; k <= 2*SIDES; k++)
   dist[k] /= 36.0;
```

dist[i] 的值就是两个骰子之和为 i 的概率。用实验模拟 N 次掷骰子，并在计算两个 1 到 6 之间的随机整数之和时记录每个值的出现频率以验证它们的概率。N 要多大才能够保证你的经验数据和准确数据的吻合程度达到小数点后三位？

1.1.36 *乱序检查*。通过实验检查表 1.1.10 中的乱序代码是否能够产生预期的效果。编写一个程序 ShuffleTest，接受命令行参数 M 和 N，将大小为 M 的数组打乱 N 次且在每次打乱之前都将数组重新初始化为 a[i] = i。打印一个 $M \times M$ 的表格，对于所有的列 j，行 i 表示的是 i 在打乱后落到 j 的位置的次数。数组中的所有元素的值都应该接近于 N/M。

1.1.37 *糟糕的打乱*。假设在我们的乱序代码中你选择的是一个 0 到 N-1 而非 i 到 N-1 之间的随机整数。证明得到的结果并非均匀地分布在 $N!$ 种可能性之间。用上一题中的测试检验这个版本。

1.1.38 *二分查找与暴力查找*。根据 1.1.10.4 节给出的暴力查找法编写一个程序 BruteForceSearch，在你的计算机上比较它和 BinarySearch 处理 largeW.txt 和 largeT.txt 所需的时间。 61

1.1.39 *随机匹配*。编写一个使用 BinarySearch 的程序，它从命令行接受一个整型参数 T，并会分别针对 $N=10^3$、10^4、10^5 和 10^6 将以下实验运行 T 遍：生成两个大小为 N 的随机 6 位正整数数组并找出同时存在于两个数组中的整数的数量。打印一个表格，对于每个 N，给出 T 次实验中该数量的平均值。 62

1.2 数据抽象

数据类型指的是一组值和一组对这些值的操作的集合。目前，我们已经详细讨论过 Java 的*原始数据类型*：例如，原始数据类型 `int` 的取值范围是 -2^{31} 到 $2^{31}-1$ 之间的整数，`int` 的操作包括 +、*、-、/、%、< 和 >。原则上所有程序都只需要使用原始数据类型即可，但在更高层次的抽象上编写程序会更加方便。在本节中，我们将重点学习定义和使用数据类型，这个过程也被称为*数据抽象*（它是对 1.1 节所述的*函数抽象*风格的补充）。

Java 编程的基础主要是使用 `class` 关键字构造被称为*引用类型*的数据类型。这种编程风格也称为*面向对象编程*，因为它的核心概念是对象，即保存了某个数据类型的值的实体。如果只有 Java 的原始数据类型，我们的程序会在很大程度上被限制在算术计算上，但有了引用类型，我们就能编写操作字符串、图像、声音以及 Java 的标准库中或者本书的网站上的数百种抽象类型的程序。比各种库中预定义的数据类型更重要的是 Java 编程中的数据类型的种类是无限的，因为你能够定义自己的*数据类型*来抽象任意对象。

抽象数据类型（ADT）是一种能够对使用者隐藏数据表示的数据类型。用 Java 类来实现抽象数据类型和用一组静态方法实现一个函数库并没有什么不同。抽象数据类型的主要不同之处在于它将*数据*和函数的实现关联，并将数据的表示方式隐藏起来。在*使用*抽象数据类型时，我们的注意力集中在 API 描述的操作上而不会去关心数据的表示；在*实现*抽象数据类型时，我们的注意力集中在*数据*本身并将实现对该数据的各种操作。

抽象数据类型之所以重要是因为在程序设计上它们支持封装。在本书中，我们将通过它们：

- 以适用于各种用途的 API 形式准确地定义问题；
- 用 API 的实现描述算法和数据结构。

我们研究同一个问题的不同算法的主要原因在于它们的性能特点不同。抽象数据类型正适合于对算法的这种研究，因为它确保我们可以随时将算法性能的知识应用于实践中：可以在不修改任何用例代码的情况下用一种算法替换另一种算法并改进所有用例的性能。

63
↓
64

1.2.1 使用抽象数据类型

要使用*一种数据类型并不一定非得知道它是如何实现的*，所以我们首先来编写一个使用一种名为 `Counter`（计数器）的简单数据类型的程序。它的值是一个名称和一个非负整数，它的操作有*创建对象并初始化为 0*、*当前值加 1* 和*获取当前值*。这个抽象对象在许多场景中都会用到。例如，这样一个数据类型可以用于电子记票软件，它能够保证投票者所能进行的唯一操作就是将他选择的候选人的计数器加一。我们也可以在分析算法性能时使用 `Counter` 来记录基本操作的调用次数。要使用 `Counter` 对象，首先需要了解应该如何定义数据类型的操作，以及在 Java 语言中应该如何创建和使用某个数据类型的对象。这些机制在现代编程中都非常重要，我们在全书中都会用到它们，因此请仔细学习我们的第一个例子。

1.2.1.1 抽象数据类型的 API

我们使用应用程序编程接口（API）来说明抽象数据类型的行为。它将列出所有*构造函数*和*实例方法*（即操作）并简要描述它们的功用，如表 1.2.1 中 `Counter` 的 API 所示。

尽管数据类型定义的基础是一组值的集合，但在 API 可见的仅是对它们的操作，而非它们的意义。因此，抽象数据类型的定义和静态方法库（请见 1.1.6.3 节）之间有许多共同之处：

- 两者的实现均为 Java 类；

- 实例方法可能接受 0 个或多个指定类型的参数，由括号表示并由逗号分隔；
- 它们可能会返回一个指定类型的值，也可能不会（用 void 表示）。

当然，它们也有三个显著的不同。

- API 中可能会出现若干个名称和类名相同且没有返回值的函数。这些特殊的函数被称为*构造函数*。在本例中，Counter 对象有一个接受一个 String 参数的构造函数。 65
- 实例方法不需要 static 关键字。它们不是静态方法——它们的目的就是操作该数据类型中的值。
- 某些实例方法的存在是为了尊重 Java 的习惯——我们将此类方法称为*继承的方法*并在 API 中将它们显示为灰色。

表 1.2.1 计数器的 API

public class	Counter	
	Counter(String id)	创建一个名为 id 的计数器
void	increment()	将计数器的值加 1
int	tally()	该对象创建之后计数器被加 1 的次数
String	toString()	对象的字符串表示

和静态方法库的 API 一样，抽象数据类型的 API 也是和用例之间的一份契约，因此它是开发任何用例代码以及实现任意数据类型的起点。在本例中，这份 API 告诉我们可以通过构造函数 Counter()、实例方法 increment() 和 tally()，以及继承的 toString() 方法使用 Counter 类型的对象。

1.2.1.2 继承的方法

根据 Java 的约定，任意数据类型都能通过在 API 中包含特定的方法从 Java 的内在机制中获益。例如，Java 中的所有数据类型都会继承 toString() 方法来返回用 String 表示的该类型的值。Java 会在用 + 运算符将任意数据类型的值和 String 值连接时调用该方法。该方法的默认实现并不实用（它会返回用字符串表示的该数据类型值的内存地址），因此我们常常会提供实现来重载默认实现，并在此时在 API 中加上 toString() 方法。此类方法的例子还包括 equals()、compareTo() 和 hashCode()（请见 1.2.5.5 节）。

1.2.1.3 用例代码

和基于静态方法的模块化编程一样，API 允许我们在不知道实现细节的情况下编写调用它的代码（以及在不知道任何用例代码的情况下编写实现代码）。1.1.7 节介绍的将程序组织为独立模块的机制可以应用于所有的 Java 类，因此它对基于抽象数据类型的模块化编程与对静态函数库一样有效。这样，只要抽象数据类型的源代码 .java 文件和我们的程序文件在同一个目录下，或是在标准 Java 库中，或是可以通过 import 语句访问，或是可以通过本书网站上介绍的 classpath 机制之一访问，该程序就能够使用这个抽象数据类型，模块化编程的所有优势就都能够继续发挥。通过将实现某种数据类型的全部代码封装在一个 Java 类中，我们可以将用例代码推向更高的抽象层次。在用例代码中，你需要*声明变量、创建对象*来保存数据类型的值并*允许*通过实例方法来操作它们。尽管你也会注意到它们的一些相似之处，但这种方式和原始数据类型的使用方式非常不同。 66

1.2.1.4 对象

一般来说，可以声明一个变量 heads 并将它通过以下代码和 Counter 类型的数据关联起来：

```
Counter heads;
```

但如何为它赋值或是对它进行操作呢？这个问题的答案涉及数据抽象中的一个基础概念：对象是能够承载数据类型的值的实体。所有对象都有三大重要特性：状态、标识和行为。对象的状态即数据类型中的值。对象的标识份能够将一个对象区别于另一个对象。可以认为对象的标识就是它在内存中的位置。对象的行为就是数据类型的操作。数据类型的实现的唯一职责就是维护一个对象的身份，这样用例代码在使用数据类型时只需遵守描述对象行为的 API 即可，而无需关注对象状态的表示方法。对象的状态可以为用例代码提供信息，或是产生某种副作用，或是被数据类型的操作所改变。但数据类型的值的表示细节和用例代码是无关的。引用是访问对象的一种方式。Java 使用术语引用类型以示和原始数据类型（变量和值相关联）的区别。不同的 Java 实现中引用的实现细节也各不相同，但可以认为引用就是内存地址，如图 1.2.1 所示（简洁起见，图中的内存地址为三位数）。

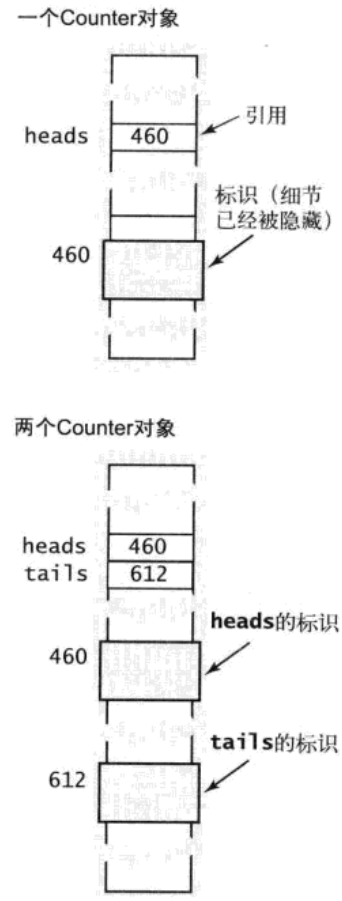


图 1.2.1 对象的表示

1.2.1.5 创建对象

每种数据类型中的值都存储于一个对象中。要创建（或实例化）一个对象，我们用关键字 new 并紧跟类名以及 ()（或在括号中指定一系列的参数，如果构造函数需要的话）来触发它的构造函数。构造函数没有返回值，因为它总是返回它的数据类型的对象的引用。每当用例调用了 new()，系统都会：

- ❑ 为新的对象分配内存空间；
- ❑ 调用构造函数初始化对象中的值；
- ❑ 返回该对象的一个引用。

在用例代码中，我们一般都会在一条声明语句中创建一个对象并通过将它和一个变量关联来初始化该变量，和使用原始数据类型时一样。和原始数据类型不同的是，变量关联的是指向对象的引用而并非数据类型的值本身。我们可以用同一个类创建无数对象——每个对象都有自己的标识，且所存储的值和另一个相同类型的对象可以相同也可以不同。例如，以下代码创建了两个不同的 Counter 对象：

67

```
Counter heads = new Counter("heads");
Counter tails = new Counter("tails");
```

抽象数据类型向用例隐藏了值的表示细节。可以假定每个 Counter 对象中的值是一个 String 类型的名称和一个 int 计数器，但不能编写依赖于任何特定表示方法的代码（即使知道假定是否正确——也许计数器是一个 long 值呢）对象的创建过程如图 1.2.2 所示。

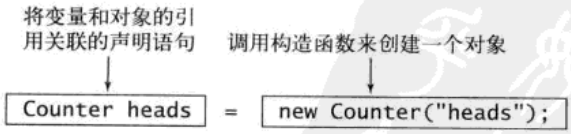


图 1.2.2 创建对象

1.2.1.6 调用实例方法

实例方法的意义在于操作数据类型中的值，因此 Java 语言提供了一种特别的机制来

触发实例方法，它突出了实例方法和对象之间的联系。具体来说，我们调用一个实例方法的方式是先写出对象的变量名，紧接着是一个句点，然后是实例方法的名称，之后是 0 个或多个在括号中并由逗号分隔的参数。实例方法可能会改变数据类型中的值，也可能只是访问数据类型中的值。实例方法拥有我们在 1.1.6.3 节讨论过的静态方法的所有性质——参数按值传递，方法名可以被重载，方法可以有返回值，它们也许还会产生一些副作用。但它们还有一个特别的性质：*方法的每次触发都是和一个对象相关的*。例如，以下代码调用了实例方法 increment() 来操作 Counter 对象 heads（在这里该操作会将计数器的值加 1）：

```
heads.increment();
```

而以下代码会调用实例方法 tally() 两次，第一次操作的是 Counter 对象 heads，第二次是 Counter 对象 tails（这里该操作会返回计数器的 int 值）：

```
heads.tally() - tails.tally();
```

以上示例的调用过程见图 1.2.3。



图 1.2.3 触发实例方法的各种方式

正如这些例子所示，在用例中实例方法和静态方法的调用方式完全相同——可以通过语句（void 方法）也可以通过表达式（有返回值的方法）。静态方法的主要作用是实现函数；非静态（实例）方法的主要作用是实现数据类型的操作。两者都可能出现在用例代码中，但很容易就可以区分它们，因为静态方法调用的开头是类名（按习惯为大写），而非静态方法调用的开头总是对象名（按习惯为小写）。表 1.2.2 总结了这些不同之处。 68

表 1.2.2 实例方法与静态方法

	实例方法	静态方法
举例	heads.increment()	Math.sqrt(2.0)
调用方式	对象名	类名
参量	对象的引用和方法的参数	方法的参数
主要作用	访问或改变对象的值	计算返回值

1.2.1.7 使用对象

通过声明语句可以将变量名赋给对象，在代码中，我们不仅可以用该变量创建对象和调用实例方法，也可以像使用整数、浮点数和其他原始数据类型的变量一样使用它。要开发某种给定数据类型的用例，我们需要：

- 声明该类型的变量，以用来引用对象；
- 使用关键字 new 触发能够创建该类型的对象的一个构造函数；
- 使用变量名在语句或表达式中调用实例方法。

例如，下面用例代码中的 Flips 类就使用了 Counter 类。它接受一个命令行参数 T 并模拟 T 次掷硬币（它还调用了 StdRandom 类）。除了这些直接用法外，我们可以和使用原始数据类型的

变量一样使用和对象关联的变量：

- ❑ 赋值语句；
- ❑ 向方法传递对象或是从方法中返回对象；
- ❑ 创建并使用对象的数组。

```
public class Flips
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Counter heads = new Counter("heads");
      Counter tails = new Counter("tails");
      for (int t = 0; t < T; t++)
         if (StdRandom.bernoulli(0.5))
              heads.increment();
         else tails.increment();
      StdOut.println(heads);
      StdOut.println(tails);
      int d = heads.tally() - tails.tally();
      StdOut.println("delta: " + Math.abs(d));
   }
}
```

```
% java Flips 10
5 heads
5 tails
delta: 0

% java Flips 10
8 heads
2 tails
delta: 6

% java Flips 1000000
499710 heads
500290 tails
delta: 580
```

Counter 类的用例，模拟 T 次掷硬币

接下来将逐个分析它们。你会发现，你需要从引用而非值的角度去考虑问题才能理解这些用法的行为。

1.2.1.8 赋值语句

使用引用类型的赋值语句将会创建该引用的一个副本。赋值语句不会创建新的对象，而只是创建另一个指向某个已经存在的对象的引用。这种情况被称为*别名*：两个变量同时指向同一个对象。别名的效果可能会出乎你的意料，因为对于原始数据类型的变量，情况不同，你必须理解其中的差异。如果 x 和 y 是原始数据类型的变量，那么赋值语句 x = y 会将 y 的值复制到 x 中。对于引用类型，复制的是引用（而非实际的值）。在 Java 中，别名是 bug 的常见原因，如下例所示（图 1.2.4）：

```
Counter c1 = new Counter("ones");
c1.increment();
Counter c2 = c1;
c2.increment();
StdOut.println(c1);
```

对于一般的 toString() 实现，这段代码将会打印出 "2 ones"。这可能并不是我们想要的，而且乍一看有些奇怪。这种问题经常出现在使用对象经验不足的人所编写的程序之中（可能就是你，所以请集中注意力！）。改变一个对象的状态将会影响到所有和该对象的别名有关的代码。我们习惯于认为两个不同的

```
Counter c1;
c1 = new Counter("ones");
c1.increment();
Counter c2 = c1;
c2.increment();
```

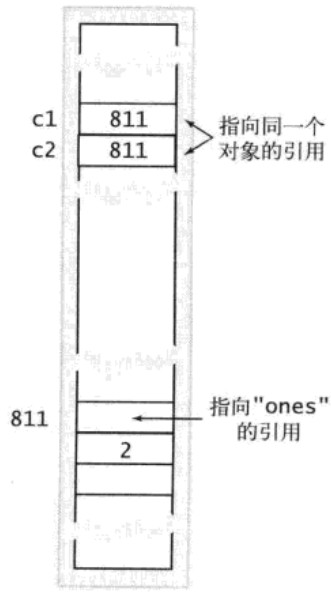


图 1.2.4 别名

原始数据类型的变量是相互独立的，但这种感觉对于引用类型的变量并不适用。

69 ≀ 70

1.2.1.9 将对象作为参数

可以将对象作为参数传递给方法，这一般都能简化用例代码。例如，当我们使用 Counter 对象作为参数时，本质上我们传递的是一个名称和一个计数器，但我们只需要指定一个变量。当我们调用一个需要参数的方法时，该动作在 Java 中的效果相当于每个参数值都出现在了一个赋值语句的右侧，而参数名则在该赋值语句的左侧。也就是说，Java 将参数值的一个副本从调用端传递给了方法，这种方式称为*按值传递*（请见 1.1.6.3 节）。这种方式的一个重要后果是方法无法改变调用端变量的值。对于原始数据类型来说，这种策略正是我们所期望的（两个变量互相独立），但每当使用引用类型作为参数时我们创建的都是别名，所以就必须小心。换句话说，这种约定将会传递引用的值（复制引用），也就是传递对象的引用。例如，如果我们传递了一个指向 Counter 类型的对象的引用，那么方法虽然无法改变原始的引用（比如将它指向另一个 Counter 对象），但它能够改变该对象的值，比如通过该引用调用 increment() 方法。

1.2.1.10 将对象作为返回值

当然也能够将对象作为方法的返回值。方法可以将它的参数对象返回，如下面的例子所示，也可以创建一个对象并返回它的引用。这种能力非常重要，因为 Java 中的方法只能有一个返回值——有了对象我们的代码实际上就能返回多个值。

```
public class FlipsMax
{
   public static Counter max(Counter x, Counter y)
   {
      if (x.tally() > y.tally()) return x;
      else                       return y;
   }

   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Counter heads = new Counter("heads");
      Counter tails = new Counter("tails");
      for (int t = 0; t < T; t++)
         if (StdRandom.bernoulli(0.5))
            heads.increment();
         else tails.increment();

      if (heads.tally() == tails.tally())
         StdOut.println("Tie");
      else StdOut.println(max(heads, tails) + " wins");
   }
}
```

```
% java FlipsMax 1000000
500281 tails wins
```

一个接受对象作为参数并将对象作为返回值的静态方法的例子

71

1.2.1.11 数组也是对象

在 Java 中，所有非原始数据类型的值都是对象。也就是说，数组也是对象。和字符串一样，Java 语言对于数组的某些操作有特殊的支持：声明、初始化和索引。和其他对象一样，当我们将数组传递给一个方法或是将一个数组变量放在赋值语句的右侧时，我们都是在创建该数组引用的一个副本，而非数组的副本。对于一般情况，这种效果正合适，因为我们期望方法能够重新安排数组的

条目并修改数组的内容，如 java.utils.Array.sort() 或表 1.1.10 讨论的 shuffle() 方法。

1.2.1.12 对象的数组

我们已经看到，数组元素可以是任意类型的数据：我们实现的 main() 方法的 args[] 参数就是一个 String 对象的数组。创建一个对象的数组需要以下两个步骤：

- 使用方括号语法调用数组的构造函数创建数组；
- 对于每个数组元素调用它的构造函数创建相应的对象。

例如，下面这段代码模拟的是掷骰子。它使用了一个 Counter 对象的数组来记录每种可能的值的出现次数。在 Java 中，对象数组即是一个由对象的引用组成的数组，而非所有对象本身组成的数组。如果对象非常大，那么在移动它们时由于只需要操作引用而非对象本身，这就会大大提高效率；如果对象很小，每次获取信息时都需要通过引用反而会降低效率。

```
public class Rolls
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      int SIDES = 6;
      Counter[] rolls = new Counter[SIDES+1];
      for (int i = 1; i <= SIDES; i++)
         rolls[i] = new Counter(i + "'s");

      for (int t = 0; t < T; t++)
      {
         int result = StdRandom.uniform(1, SIDES+1);
         rolls[result].increment();
      }
      for (int i = 1; i <= SIDES; i++)
         StdOut.println(rolls[i]);
   }
}
```

```
% java Rolls 1000000
167308 1's
166540 2's
166087 3's
167051 4's
166422 5's
166592 6's
```

模拟 T 次掷骰子的 Counter 对象的用例

72

有了这些对象的知识，运用数据抽象的思想编写代码（定义和使用数据类型，将数据类型的值封装在对象中）的方式称为*面向对象编程*。刚才学习的基本概念是我们面向对象编程的起点，因此有必要对它们进行简单的总结。*数据类型*指的是一组值和一组对值的操作的集合。我们会将数据类型实现在独立的 Java 类模块中并编写它们的用例。*对象*是能够存储任意该数据类型的值的实体，或数据类型的*实例*。对象有三大关键性质：*状态*、*标识*和*行为*。一个数据类型的实现所支持的操作如下。

- *创建对象*（创造它的标识）：使用 new 关键字触发构造函数并创建对象，初始化对象中的值并返回对它的引用。
- *操作对象中的值*（控制对象的行为，可能会改变对象的状态）：使用和对象关联的变量调用实例方法来对对象中的值进行操作。
- *操作多个对象*：创建对象的数组，像原始数据类型的值一样将它们传递给方法或是从方法中返回，只是变量关联的是对象的引用而非对象本身。

73

这些能力是这种灵活且应用广泛的现代编程方式的基础，也是我们在本书中对算法研究的基础。

1.2.2 抽象数据类型举例

Java 语言内置了上千种抽象数据类型，我们也会为了辅助算法研究创建许多其他抽象数据类型。实际上，我们编写的每一个 Java 程序实现的都是某种数据类型（或是一个静态方法库）。为了控制复杂度，我们会明确地说明在本书中用到的所有抽象数据类型的 API（实际上并不多）。

在本节中，我们会举一些抽象数据类型的例子，以及它们的一些用例。在某些情况下，我们会节选一些含有数十个方法的 API 的一部分。我们将会用这些 API 展示一些实例以及在本书中会用到的一些方法，并用它们说明要使用一个抽象数据类型并不需要了解其实现细节。

作为参考，下页显示了我们在本书中将会用到或开发的所有数据类型。它们可以被分为以下几类。

- `java.lang.*` 中的标准系统抽象数据类型，可以被任意 Java 程序调用。
- Java 标准库中的抽象数据类型，如 java.swt、java.net 和 java.io，它们也可以被任意 Java 程序调用，但需要 `import` 语句。
- I/O 处理类抽象数据类型，和 StdIn 和 StdOut 类似，允许我们处理多个输入输出流。
- 面向数据类抽象数据类型，它们的主要作用是通过封装数据的表示简化数据的组织和处理。稍后在本节中我们将介绍在计算几何和信息处理中的几个实际应用的例子，并会在以后将它们作为抽象数据类型用例的范例 。
- 集合类抽象数据类型，它们的主要用途是简化对同一类型的一组数据的操作。我们将会在 1.3 节中介绍基本的 `Bag`、`Stack` 和 `Queue` 类，在第 2 章中介绍优先队列（PQ）及其相关的类，在第 3 章和第 5 章中分别介绍符号表（ST）和集合（SET）以及相关的类。
- 面向操作的抽象数据类型，我们用它们分析各种算法，如 1.4 节和 1.5 节所述。
- 图算法相关的抽象数据类型，它们包括一些用来封装各种图的表示的面向数据的抽象数据类型，和一些提供图的处理算法的面向操作的抽象数据类型。

这个列表中并没有包含我们将在练习中遇到的某些抽象数据类型，读者可以在本书的索引中找到它们。另外，如 1.2.4.1 节所述，我们常常通过描述性的前缀来区分各种抽象数据类型的多种实现。从整体上来说，我们使用的抽象数据类型说明组织并理解你所使用的数据结构是现代编程中的重要因素。

一般的应用程序可能只会使用这些抽象数据类型中的 5 ~ 10 个。在本书中，开发和组织抽象数据类型的主要目标是使程序员们在编写用例时能够轻易地利用它们的一小部分。 74

1.2.2.1 几何对象

面向对象编程的一个典型例子是为几何对象设计数据类型。例如，表 1.2.3 至表 1.2.5 中的 API 为三种常见的几何对象定义了相应的抽象数据类型：`Point2D`（平面上的点）、`Interval1D`（直线上的间隔）、`Interval2D`（平面上的二维间隔，即和数轴对齐的长方形）。和以前一样，这些 API 都是自文档化的，它们的用例十分容易理解，列在了表 1.2.5 的后面。这段代码从命令行读取一个 `Interval2D` 的边界和一个整数 T，在单位正方形内随机生成 T 个点并统计落在间隔之内的点数（用来估计该长方形的面积）。为了表现效果，用例还画出了间隔和落在间隔之外的所有点。这种计算方法是一个模型，它将计算几何图形的面积和体积的问题转化为了判定一个点是否落在该图形中（稍稍简单，但仍然不那么容易）。我们当然也能为其他几何对象定义 API，比如线段、三角形、多边形、圆等，不过实现它们的相关操作可能十分有挑战性。本节末尾的练习会考察其中几个例子。

java.lang 中的标准 Java 系统类型	
Integer	int 的封装类
Double	double 的封装类
String	可由索引访问的 char 值序列
StringBuilder	字符串构造类
其他 Java 数据类型	
java.awt.Color	颜色
java.awt.Font	字体
java.net.URL	URL
java.io.File	文件
我们的标准 I/O 类型	
In	输入流
Out	输出流
Draw	绘图类
用于用例的面向数据的数据类型	
Point2D	平面上的点
Interval1D	一维间隔
Interval2D	二维间隔
Date	日期
Transaction	换位
用于算法分析的数据类型	
Counter	计数器
Accumulator	累加器
VisualAccumulator	可视累加器
Stopwatch	计时器

集合类数据类型	
Stack	下压栈
Queue	先进先出（FIFO）队列
Bag	包
MinPQ, MaxPQ	优先队列
IndexMinPQ IndexMaxPQ	索引优先队列
ST	符号表
SET	集合
StringST	符号表（字符串键）
面向数据的图数据类型	
Graph	无向图
Digraph	有向图
Edge	边（加权）
EdgeWeightedGraph	无向图（加权）
DirectedEdge	边（有向，加权）
EdgeWeightedDigraph	图（有向，加权）
面向操作的图数据类型	
UF	动态连通性
DepthFirstPaths	路径的深度优先搜索
CC	连通分量
BreadthFirstPaths	路径的广度优先搜索
DirectedDFS	有向图路径的深度优先搜索
DirectedBFS	有向图路径的广度优先搜索
TransitiveClosure	所有路径
Topological	拓扑排序
DepthFirstOrder	深度优先搜索顶点被访问的顺序
DirectedCycle	环的搜索
SCC	强连通分量
MST	最小生成树
SP	最短路径

75

本书中使用的部分抽象数据类型

表 1.2.3 平面上的点的 API

public class **Point2D**		
	Point2D(double x, double y)	创建一个点
double	x()	*x* 坐标
double	y()	*y* 坐标
double	r()	极径（极坐标）
double	theta()	极角（极坐标）
double	distTo(Point2D that)	从该点到 that 的欧几里德距离
void	draw()	用 StdDraw 绘出该点

表 1.2.4 直线上间隔的 API

public class **Interval1D**		
	Interval1D(double lo, double hi)	创建一个间隔
double	length()	间隔长度
boolean	contains(double x)	x 是否在间隔中
boolean	intersect(Interval1D that)	该间隔是否和间隔 that 相交
void	draw()	用 StdDraw 绘出该间隔

表 1.2.5 平面上的二维间隔的 API

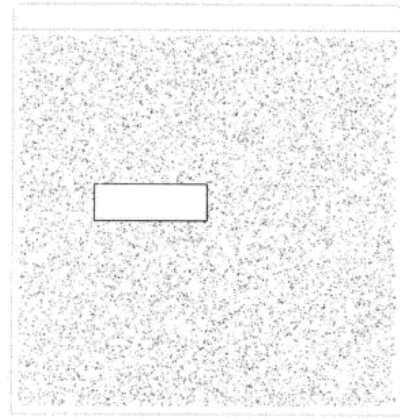
public class **Interval2D**		
	Interval2D(Interval1D x, Interval1D y)	创建一个二维间隔
double	area()	二维间隔的面积
boolean	contains(Point2D p)	p 是否在二维间隔中
boolean	intersect(Interval2D that)	该间隔是否和二维间隔 that 相交
void	draw()	用 StdDraw 绘出该二维间隔

```
public static void main(String[] args)
{
   double xlo = Double.parseDouble(args[0]);
   double xhi = Double.parseDouble(args[1]);
   double ylo = Double.parseDouble(args[2]);
   double yhi = Double.parseDouble(args[3]);
   int T = Integer.parseInt(args[4]);

   Interval1D xinterval = new Interval1D(xlo, xhi);
   Interval1D yinterval = new Interval1D(ylo, yhi);
   Interval2D box = new Interval2D(x, y);
   box.draw();

   Counter c = new Counter("hits");
   for (int t = 0; t < T; t++)
   {
      double x = Math.random();
      double y = Math.random();
      Point2D p = new Point(x, y);
      if (box.contains(p)) c.increment();
      else                 p.draw();
   }

   StdOut.println(c);
   StdOut.println(box.area());
}
```



```
% java Interval2D .2 .5 .5 .6 10000
297 hits
.03
```

Interval2D的测试用例

处理几何对象的程序在自然世界模型、科学计算、电子游戏、电影等许多应用的计算中有着广泛的应用。此类程序的研发已经发展成了计算几何学的这门影响深远的研究学科。在贯穿全书的众多例子中你会看到，我们在本书中学习的许多算法在这个领域都有应用。在这里我们要说明的是直接表示几何对象的抽象数据类型的定义并不困难且在用例中的应用也十分简洁。本书网站和本节末尾的若干练习都证明了这一点。

76
ι
77

1.2.2.2 信息处理

无论是需要处理数百万信用卡交易的银行，还是需要处理数十亿点击的网络分析公司，或是需

要处理数百万实验观察结果的科学研究小组，无数应用的核心都是组织和处理信息。抽象数据类型是组织信息的一种自然方式。虽然没有给出细节，表 1.2.6 中的两份 API 也展示了商业应用程序中的一种典型做法。这里的主要思想是定义和真实世界中的物体相对应的对象。一个日期就是一个日、月和年的集合，一笔交易就是一个客户、日期和金额的集合。这只是两个例子，我们也可以为客户、时间、地点、商品、服务和其他任何东西定义对象以保存相关的信息。每种数据类型都包含能够创建对象的构造函数和用于访问其中数据的方法。为了简化用例的代码，我们为每个类型都提供了两个构造函数，一个接受适当类型的数据，另一个则能够解析字符串中的数据（细节请见练习 1.2.19）。和以前一样，用例并不需要知道数据的表示方法。用这种方式组织数据最常见的理由是将一个对象和它相关的数据变成一个整体：我们可以维护一个 Transaction 对象的数组，将 Date 值作为参数或是某个方法的返回值等。这些数据类型的重点在于封装数据，同时它们也可以确保用例的代码不依赖于数据的表示方法。我们不会深究这种组织信息的方式，需要注意的只是这种做法，以及实现继承的方法 toString()、compareTo()、equals() 和 hashCode() 可以使我们的算法处理任意类型的数据。我们会在 1.2.5.4 节中详细讨论继承的方法。例如，我们已经注意到，根据 Java 的习惯，在数据结构中包含一个 toString() 的实现可以帮助用例打印出由对象中的值组成的一个字符串。我们会在 1.3 节、2.5 节、3.4 节和 3.5 节中用 Date 类和 Transaction 类作为例子考察其他继承的方法所对应的习惯用法。1.3 节给出了有关数据类型和 Java 语言的类型参数（泛型）机制的几个经典例子，它们都遵循了这些习惯用法。第 2 章和第 3 章也都利用了泛型和继承的方法来实现可以处理任意数据类型的高效排序和查找算法。

表 1.2.6 商业应用程序中的示例 API（日期和交易）

public class	**Date** implements Comparable<Date>	
	Date(int month, int day, int year)	创建一个日期
	Date(String date)	创建一个日期（解析字符串的构造函数）
int	month()	月
int	day()	日
int	year()	年
String	toString()	对象的字符串表示
boolean	equals(Object that)	该日期和 that 是否相同
int	compareTo(Date that)	将该日期和 that 比较
int	hashCode()	散列值
public class	**Transaction** implements Comparable<Transaction>	
	Transaction(String who, Date when, double amount)	
	Transaction(String transaction)	创建一笔交易（解析字符串的构造函数）
String	who()	客户名
Date	when()	交易日期
double	amount()	交易金额
String	toString()	对象的字符串表示
boolean	equals(Object that)	该笔交易和 that 是否相同
int	compareTo(Date that)	将该笔交易和 that 比较
int	hashCode()	散列值

每当遇到逻辑上相关的不同类型的数据时，你都应该考虑像刚才的例子那样定义一个抽象数据类型。这么做能够帮助我们组织数据并在一般应用程序中极大地简化使用者的代码。它是我们在通向数据抽象之路上迈出的重要一步。

78 ~ 79

1.2.2.3 字符串

Java 的 String 是一种重要而实用的抽象数据类型。一个 String 值是一串可以由索引访问的 char 值。String 对象拥有许多实例方法，如表 1.2.7 所示。

表 1.2.7 Java 的字符串 API（部分）

Public class **String**		
	String()	创建一个空字符串
int	length()	字符串长度
int	charAt(int i)	第 i 个字符
int	indexOf(String p)	p 第一次出现的位置（如果没有则返回 -1）
int	indexOf(String p, int i)	p 在 i 个字符后第一次出现的位置（如果没有则返回 -1）
String	concat(String t)	将 t 附在该字符串末尾
String	substring(int i, int j)	该字符串的子字符串（第 i 个字符到第 j-1 个字符）
String[]	split(String delim)	使用 delim 分隔符切分字符串
int	compareTo(String t)	比较字符串
boolean	equals(String t)	该字符串的值和 t 的值是否相同
int	hashCode()	散列值

String 值和字符数组类似，但两者是不同的。数组能够通过 Java 语言的内置语法访问每个字符，String 则为索引访问、字符串长度以及其他许多操作准备了实例方法。另一方面，Java 语言为 String 的初始化和连接提供了特别的支持：我们可以直接使用字符串字面量而非构造函数来创建并初始化一个字符串，还可以直接使用 + 运算符代替 concat() 方法。我们不需要了解实现的细节，但是在第 5 章中你会看到，了解某些方法的性能特点在开发字符串处理算法时是非常重要的。为什么不直接使用字符数组代替 String 值？对于任何抽象数据类型，这个问题的答案都是一样的：*为了使代码更加简洁清晰*。有了 String 类型，我们可以写出清晰干净的用例代码而无需关心字符串的表示方式。先看一下右侧这段短小的列表，其中甚至含有一些需要我们在第 5 章才会学到的高级算法才能实现的强大操作。例如，split() 方法的参数可以是*正则表达式*（请见 5.4 节），“典型的字符串处理代码”（显示在下页）中 split() 的参数是 "\\s+"，它表示“一个或多个制表符、空格、换行符或回车”。

```
String a = "now is ";
String b = "the time ";
String c = "to"
```

方法	返回值
a.length()	7
a.charAt(4)	i
a.concat(c)	"now is to"
a.indexOf("is")	4
a.substring(2, 5)	"w i"
a.split(" ")[0]	"now"
a.split(" ")[1]	"is"
b.equals(c)	false

字符串操作举例

80

任 务	实 现
判断字符串是否是一条回文	`public static boolean isPalindrome(String s)` `{` `  int N = s.length();` `  for (int i = 0; i < N/2; i++)` `  if (s.charAt(i) != s.charAt(N-1-i))` `  return false;` `  return true;` `}`
从一个命令行参数中提取文件名和扩展名	`String s = args[0];` `int dot = s.indexOF(".");` `String base = s.substring(0, dot);` `String extension = s.substring(dot + 1, s.length());`
打印出标准输入中所有含有通过命令行指定的字符串的行	`String query = args[0];` `while (!StdIn.isEmpty())` `{` `  String s = StdIn.readLine();` `  if (s.contains(query)) StdOut.println(s);` `}`
以空白字符为分隔符从 StdIn 中创建一个字符串数组	`String input = StdIn.readAll();` `String[] words = input.split("\\s+");`
检查一个字符串数组中的元素是否已按照字母表顺序排列	`public boolean isSorted(String[] a)` `{` `  for (int i = 1; i < a.length; i++)` `  {` `  if (a[i-1].compareTo(a[i]) > 0)` `  return false;` `  }` `  return true;` `}`

典型的字符串处理代码

81

1.2.2.4 再谈输入输出

1.1 节中的 StdIn、StdOut 和 StdDraw 标准库的一个缺点是对于任意程序，我们只能接受一个输入文件、向一个文件输出或是产生一幅图像。有了面向对象编程，我们就能定义类似的机制来在一个程序中同时处理多个输入流、输出流和图像。具体来说，我们的标准库定义了数据类型 In、Out 和 Draw，它们的 API 如表 1.2.8 至表 1.2.10 所示。当使用一个 String 类型的参数调用它们的构造函数时，In 和 Out 会首先尝试在当前目录下查找指定的文件。如果找不到，它会假设该参数是一个网站的名称并尝试连接到那个网站（如果该网站不存在，它会抛出一个运行时异常）。无论哪种情况，指定的文件或网站都会成为被创建的输入或输出流对象的来源或目标，所有 read*() 和 print*() 方法都会指向那个文件或网站（如果你使用的是无参数的构造函数，对象将会使用标准的输入输出流）。这种机制使得单个程序能够处理多个文件和图像；你也能将这些对象赋给变量，将它们当做方法的参数、作为方法的返回值或是创建它们的数组，可以像操作任何类型的对象那样操作它们。下页所示的程序 Cat 就是一个 In 和 Out 的用例，它使用了多个输入流来将多个输入文件归并到同一个输出文件中。In 和 Out 类也包括将仅含 int、double 或 String 类型值的文件读取为一个数组的静态方法（请见 1.3.1.5 节和练习 1.2.15）。

```
public class Cat
{
   public static void main(String[] args)
   {  // 将所有输入文件复制到输出流（最后一个参数）中
      Out out = new Out(args[args.length-1]);
      for (int i = 0; i < args.length - 1; i++)
      {  // 将第i个输入文件复制到输出流中
         In in = new In(args[i]);
         String s = in.readAll();
         out.println(s);
         in.close();
      }
      out.close();
   }
}
```

```
% more in1.txt
This is

% more in2.txt
a tiny
test.

% java Cat in1.txt in2.txt out.txt

% more out.txt
This is
a tiny
test.
```

82

In和Out的用例示例

表 1.2.8 我们的输入流数据类型的 API

public class **In**		
	In()	从标准输入创建输入流
	In(String name)	从文件或网站创建输入流
boolean	isEmpty()	如果输入流为空则返回 true，否则返回 false
int	readInt()	读取一个 int 类型的值
double	readDouble()	读取一个 double 类型的值
	...	
void	close()	关闭输入流

注：In 对象也支持 StdIn 所支持的所有操作。

表 1.2.9 我们的输出流数据类型的 API

public class **Out**		
	Out()	从标准输出创建输出流
	Out(String name)	从文件创建输出流
void	print(String s)	将 s 添加到输出流中
void	println(String s)	将 s 和一个换行符添加到输出流中
void	println()	将一个换行符添加到输出流中
void	printf(String f, ...)	格式化并打印到输出流中
void	close()	关闭输出流

注：Out 对象也支持 StdOut 所支持的所有操作。

表 1.2.10 我们的绘图数据类型的 API

public class **Draw**		
	Draw()	
void	line(double x0, double y0, double x1, double y1)	
void	point(double x, double y)	
	...	

注：Draw 对象也支持 StdDraw 所支持的所有操作。

83

1.2.3 抽象数据类型的实现

和静态方法库一样，我们也需要使用 Java 的类（class）实现抽象数据类型并将所有代码放入一个和类名相同并带有 .java 扩展名的文件中。文件的第一部分语句会定义表示数据类型的值的实例变量。它们之后是实现对数据类型的值的操作的构造函数和实例方法。实例方法可以是公共的（在 API 中说明）或是私有的（用于辅助计算，用例无法使用）。一个数据类型的定义中可能含有多个构造函数，而且也可能含有静态方法，特别是单元测试用例 main()，它通常在调试和测试中很实用。作为第一个例子，我们来学习 1.2.1.1 节定义的 Counter 抽象数据类型的实现。它的完整实现（带有注释）如图 1.2.5 所示，在对它的各个部分的讨论中，我们还将该图作为参考。本书后面开发的每个抽象数据类型的实现都会含有和这个简单例子相同的元素。

```
public class Counter
{
   private final String name;
   private int count;
   ...
}
```

实例变量的声明

抽象数据类型中的实例变量是私有的

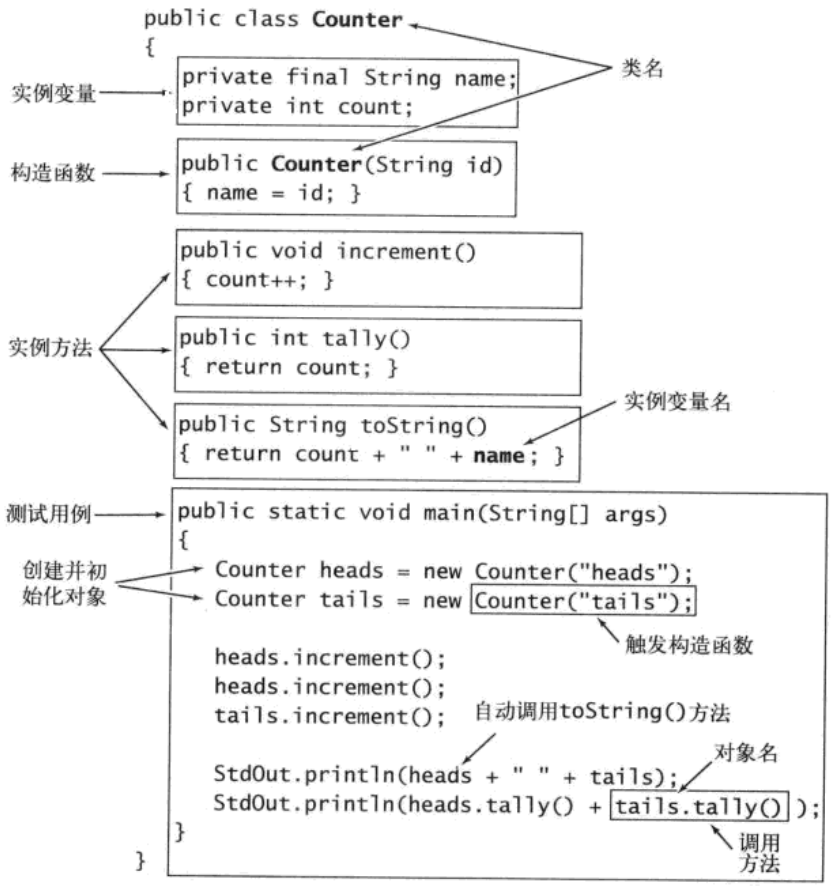


图 1.2.5 详解数据类型的定义类

1.2.3.1 实例变量

要定义数据类型的值（即每个对象的状态），我们需要声明实例变量，声明的方式和局部变量差不多。实例变量和你所熟悉的静态方法或是某个代码段中的局部变量最关键的区别在于：每一时刻每个局部变量只会有一个值，但每个实例变量则对应着无数值（数据类型的每个实例对象都会有一个）。

这并不会产生二义性，因为我们在访问实例变量时都需要通过一个对象——我们访问的是这个对象的值。同样，每个实例变量的声明都需要一个可见性修饰符。在抽象数据类型的实现中，我们会使用 private，也就是使用 Java 语言的机制来保证向使用者隐藏抽象数据类型中的数据表示，如下面的示例所示。如果该值在初始化之后不应该再被改变，我们也会使用 final。Counter 类型含有两个实例变量，一个 String 类型的值 name 和一个 int 类型的值 count。如果我们使用 public 修饰这些实例变量（在 Java 中是允许的），那么根据定义，这种数据类型就不再是抽象的了，因此我们不会这么做。

1.2.3.2 构造函数

每个 Java 类都至少含有一个构造函数以创建一个对象的标识。构造函数类似于一个静态方法，但它能够直接访问实例变量且没有返回值。一般来说，构造函数的作用是初始化实例变量。每个构造函数都将创建一个对象并向调用者返回一个该对象的引用。构造函数的名称总是和类名相同。我们可以和重载方法一样重载这个名称并定义签名不同的多个构造函数。如果没有定义构造函数，类将会隐式定义一个默认情况下不接受任何参数的构造函数并将所有实例变量初始化为默认值。原始数字类型的实例变量默认值为 0，布尔类型变量为 false，引用类型变量为 null。我们可以在声明语句中初始化这些实例变量并改变这些默认值。当用例使用关键字 new 时，Java 会自动触发一个构造函数。重载构造函数一般用于将实例变量由默认值初始化为用例提供的值。例如，Counter 类型有个接受一个参数的构造函数，它将实例变量 name 初始化为由参数给定的值（实例变量 count 仍将被初始化为默认值 0）。构造函数解析如图 1.2.6 所示。

84 ~ 85

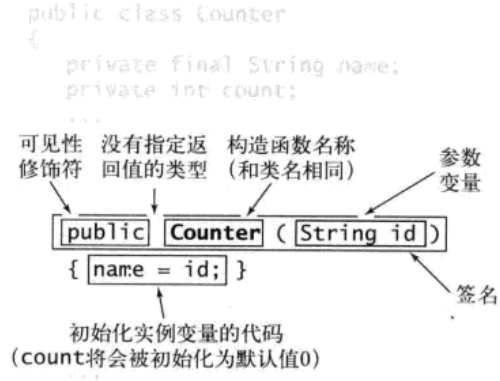


图 1.2.6 详解构造函数

1.2.3.3 实例方法

实现数据类型的实例方法（即每个对象的行为）的代码和 1.1 节中实现静态方法（函数）的代码完全相同。每个实例方法都有一个返回值类型、一个签名（它指定了方法名、返回值类型和所有参数变量的名称）和一个主体（它由一系列语句组成，包括一个返回语句来将一个返回类型的值传递给调用者）。当调用者触发了一个方法时，方法的参数（如果有）均会被初始化为调用者所提供的值，方法的语句会被执行，直到得到一个返回值并且将该值返回给调用者。它的效果就好像调用者代码中的函数调用被替换为了这个返回值。实例方法的所有这些行为都和静态方法相同，只有一点关键的不同：它们可以访问并操作实例变量。如何指定我们希望使用的对象的实例变量？只要稍加思考，就能够得到合理的答案：在一个实例方法中对变量的引用指的是该方法的变量中的值。当我们调用 heads.increment() 时，increment() 方法中的代码访问的是 heads 中的实例变量。换句话说，面向对象编程为 Java 程序增加了另一种使用变量的重要方式。

86

- 通过触发一个实例方法来操作该对象的值。

这与调用静态方法仅仅是语法上的区别（请见答疑），但在许多情况下它颠覆了现代程序员对程序开发的思维方式。你会看到，这种方式与算法和数据结构的研究非常契合。实例方法解析如图 1.2.7 所示。

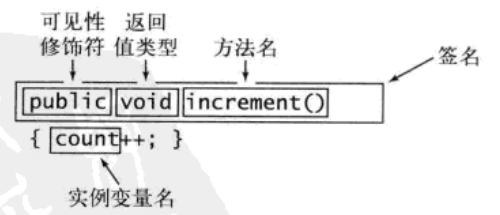


图 1.2.7 详解实例方法

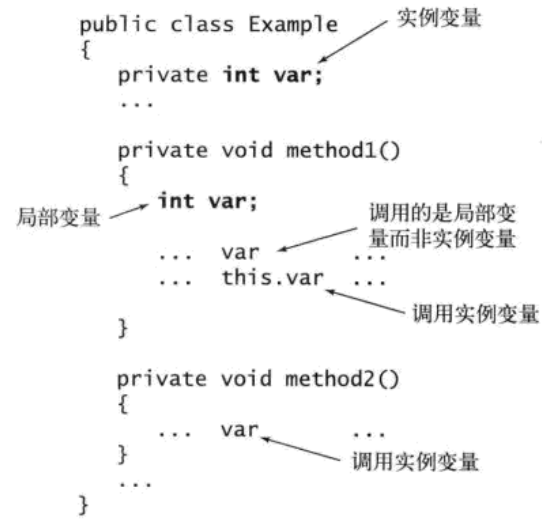
1.2.3.4 作用域

总的来说，我们在实现实例方法的 Java 代码中使用了三种变量：

- 参数变量；
- 局部变量；
- 实例变量。

在静态方法中前两者的用法没有变化：方法的签名定义了参数变量，在方法被调用时参数方法会被初始化为调用者提供的值；局部变量的声明和初始化都在方法的主体中。参数变量的作用域是整个方法；局部变量的作用域是当前代码段中它的定义之后的所有语句。实例变量则完全不同（如右侧示例所示）：它们为该类的对象保存了数据类型的值，它们的作用域是整个类（如果出现二义性，可以使用 this 前缀来区别实例变量）。理解实例方法中这三种变量的区别是理解面向对象编程的关键。

87



实例方法中的实例变量和局部变量的作用范围

1.2.3.5 API、用例与实现

这些都是你要在 Java 中构造并使用抽象数据类型所需要理解的基本组件。我们将要学习的每个抽象数据类型的实现都会是一个含有若干私有实例变量、构造函数、实例方法和一个测试用例的 Java 类。要完全理解一个数据类型，我们需要它的 API、典型的用例和它的实现。Counter 类型的总结请见表 1.2.11。为了强调用例和实现的分离，我们一般会将用例独立成为含有一个静态方法 main() 的类，并将数据类型定义中的 main() 方法预留为一个用于开发和最小单元测试的测试用例（至少调用每个实例方法一次）。我们开发的每种数据类型都会遵循相同的步骤。我们思考的不是应该采取什么行动来达成某个计算性的目的（如同我们第一次学习编程时那样），而是用例的需求。我们会按照下面三步走的方式用抽象数据类型满足它们。

- 定义一份 API：API 的作用是将使用和实现分离，以实现模块化编程。我们制定一份 API 的目标有二：第一，我们希望用例的代码清晰而正确，事实上，在最终确定 API 之前就编写一些用例代码来确保所设计的数据类型操作正是用例所需要的是很好的主意；第二，我们希望能够实现这些操作，定义一些无法实现的操作是没有意义的。
- 用一个 Java 类实现 API 的定义：首先我们选择适当的实例变量，然后再编写构造函数和实例方法。
- 实现多个测试用例来验证前两步做出的设计决定。

表 1.2.11 一个简单计数器的抽象数据类型

API	public class **Counter**	
	Counter(String id)	创建一个名为 id 的计数器
void	increment()	将计数器的值加 1
int	tally()	计数器的值
String	toString()	对象的字符串表示

（续）

典型的用例

```
public class Flips
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Counter heads = new Counter("heads");
      Counter tails = new Counter("tails");
      for (int t = 0; t < T; t++)
         if (StdRandom.bernoulli(0.5))
            heads.increment();
         else tails.increment();
      StdOut.println(heads);
      StdOut.println(tails);
      int d = heads.tally() - tails.tally();
      StdOut.println("delta: " + Math.abs(d));
   }
}
```

数据类型的实现

```
public class Counter
{
   private final String name;
   private int count;
   public Counter(String id)
   { name = id; }
   public void increment()
   { count++; }
   public int tally()
   { return count; }
   public String toString()
   { return count + " " + name; }
}
```

使用方法

```
% java Flips 1000000
500172 heads
499828 tails
delta: 344
```

用例一般需要什么操作？数据类型的值应该是什么才能最好地支持这些操作？这些基本的判断是我们开发的每种实现的核心内容。

88
~
89

1.2.4 更多抽象数据类型的实现

和任何编程概念一样，理解抽象数据类型的威力和用法的最好办法就是仔细研究更多的例子和实现。本书中大量代码是通过抽象数据类型实现的，因此你的机会很多，但是一些更简单的例子能够帮助我们为研究抽象数据类型打好基础。

1.2.4.1 日期

表 1.2.12 是我们在表 1.2.6 中定义的 Date 抽象数据类型的两种实现。简单起见，我们省略了解析字符串的构造函数（请见练习 1.2.19）和继承的方法 equals()（请见 1.2.5.8 节）、compareTo()（请见 2.1.1.4 节）和 hashCode()（请见练习 3.4.22）。表 1.2.12 中左侧的简单实现将日、月和年设为实例变量，这样实例方法就可以直接返回适当的值；右侧的实现更加节省空间，仅使用了一个 int 变量来表示一个日期。它将 d 日、m 月和 y 年的一个日期表示为一个混合进制的整数 512y+32m+d。用例分辨这两种实现的区别的一种方法可能是打破我们对日期的隐式假设：第

二种实现的正确性基于日的值在 0 到 31 之间，月的值在 0 到 15 之间，年的值为正（在实际应用中，两种实现都应该检查月份的值是否在 1 到 12 之间，日的值是否在 1 到 31 之间，以及例如 2009 年 6 月 31 日和 2 月 29 日这样的非法日期，尽管这么做要费些工夫）。这个例子的主要意思是说明我们在 API 中极少完整地指定对实现的要求（一般来说我们都会尽力而为，这里还可以做得更好）。用例要分辨出这两种实现的区别的另一种方法是性能：右侧的实现中保存数据类型的值所需的空间较少，代价是在向用例按照约定的格式提供这些值时花费的时间更多（需要进行一两次算数运算）。这种交换是很常见的：某些用例可能偏爱其中一种实现，而另一些用例可能更喜欢另一种，因此我们两者都要满足。事实上，本书中反复出现的一个主题就是我们需要理解各种实现对空间和时间的需求以及它们对各种用例的适用性。在实现中使用数据抽象的一个关键优势是我们可以将一种实现替换为另一种而无需改变用例的任何代码。

表 1.2.12 一种封装日期的抽象数据类型以及它的两种实现

API	public class **Date**		
		Date(int month, int day, int year)	创建一个日期
	int	day()	日
	int	month()	月
	int	year()	年
	String	toString()	对象的字符串表示

测试用例

```
public static void main(String[] args)
{
   int m = Integer.parseInt(args[0]);
   int d = Integer.parseInt(args[1]);
   int y = Integer.parseInt(args[2]);
   Date date = new Date(m, d, y);
   StdOut.println(date);
}
```

使用方法

```
% java Date 12 31 1999
12/31/1999
```

数据类型的实现

```
public class Date
{
   private final int month;
   private final int day;
   private final int year;
   public Date(int m, int d, int y)
   {  month = m; day = d; year = y; }
   public int month()
   {  return month;  }
   public int day()
   {  return day;  }
   public int year()
   {  return year;  }
   public String toString()
   {  return month() + "/" + day()
                     + "/" + year();  }
}
```

数据类型的另一种实现

```
public class Date
{
   private final int value;
   public Date(int m, int d, int y)
   { value = y*512 + m*32 + d; }
   public int month()
   { return (value / 32) % 16; }
   public int day()
   { return value % 32; }
   public int year()
   { return value / 512; }

   public String toString()
   {  return month() + "/" + day()
                     + "/" + year();  }
}
```

1.2.4.2 维护多个实现

同一份 API 的多个实现可能会产生维护和命名问题。在某些情况下，我们可能只是想将较老的实现替换为改进的实现。而在另一些情况下，我们可能需要维护两种实现，一种适用于某些用例，另一种适用于另一些用例。实际上，本书的一个主要目标就是深入讨论若干种基本抽象数据结构的实现并衡量它们的性能的不同。在本书中，我们经常会比较同一份 API 的两种不同实现在同一个用例中的性能表现。为此，我们通常采用一种非正式的命名约定。

90 / 91

- 通过前缀的描述性修饰符区别同一份 API 的不同实现。例如，我们可以将表 1.2.12 中的 `Date` 实现命名为 `BasicDate` 和 `SmallDate`，我们可能还希望实现一种能够验证日期是否合法的 `SmartDate`。
- 维护一个没有前缀的参考实现，它应该适合于大多数用例的需求。在这里，大多数用例应该直接会使用 `Date`。

在一个庞大的系统中，这种解决方案并不理想，因为它可能会需要修改用例的代码。例如，如果需要开发一个新的实现 `ExtraSmallDate`，那么我们只能修改用例的代码或是让它成为所有用例的参考实现。Java 有许多高级语言特性来保证在无需修改用例代码的情况下维护多个实现，但我们很少会使用它们，因为即使 Java 专家使用起它们来也十分困难（有时甚至是有争议的），尤其是同我们极为需要的其他高级语言特性（泛型和迭代器）一起使用时。这些问题很重要（例如，忽略它们会导致千禧年著名的 Y2K 问题，因为许多程序使用的都是它们自己对日期的抽象实现，且并没有考虑到年份的头两位数字），但是深究它们会使我们大大偏离对算法的研究。

1.2.4.3 累加器

表 1.2.13 中的累加器 API 定义了一种能够为用例计算一组数据的实时平均值的抽象数据类型。例如，本书中经常会使用该数据类型来处理实验结果（请见 1.4 节）。它的实现很简单：它维护一个 `int` 类型的实例变量来记录已经处理过的数据值的数量，以及一个 `double` 类型的实例变量来记录所有数据值之和，将和除以数据数量即可得到平均值。请注意该实现并没有保存数据的值——它可以用于处理大规模的数据（甚至是在一个无法全部保存它们的设备上），而一个大型系统也可以大量使用

表 1.2.13 一种能够累加数据的抽象数据类型

API	public class **Accumulator**	
	Accumulator()	创建一个累加器
void	addDataValue(double val)	添加一个新的数据值
double	mean()	所有数据值的平均值
String	toString()	对象的字符串表示

典型的用例

```
public class TestAccumulator
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Accumulator a = new Accumulator();
      for (int t = 0; t < T; t++)
         a.addDataValue(StdRandom.random());
      StdOut.println(a);
   }
}
```

使用方法

```
% java TestAccumulator 1000
Mean (1000 values): 0.51829

% java TestAccumulator 1000000
Mean (1000000 values): 0.49948

% java TestAccumulator 1000000
Mean (1000000 values): 0.50014
```

（续）

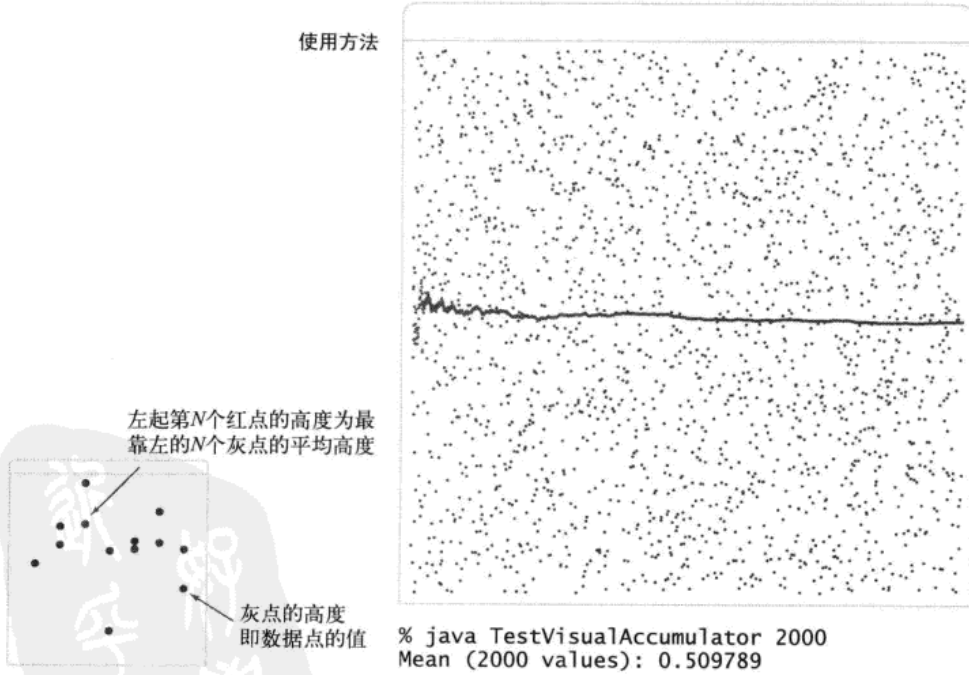
数据类型的实现

```
public class Accumulator
{
   private double total;
   private int N;
   public void addDataValue(double val)
   {
       N++;
       total += val;
   }
   public double mean()
   {  return total/N;  }
   public String toString()
   { return "Mean (" + N + " values): "
               + String.format("%7.5f", mean()); }
}
```

92～93 累加器。这种性能特点很容易被忽视，所以也许应该在 API 中注明，因为一种存储所有数据值的实现可能会使调用它的应用程序用光所有内存。

1.2.4.4 可视化的累加器

表 1.2.14 所示的可视化累加器的实现继承了 Accumulator 类并展示了一种实用的副作用：它用 StdDraw 画出了所有数据（灰色）和实时的平均值（红色），见图 1.2.8。完成这项任务最简单的办法是添加一个构造函数来指定需要绘出的点数和它们的最大值（用于调整图像的比例）。严格说来，VisualAccumulator 并不是 Accumulator 的 API 的实现（它的构造函数的签名不同且产生



94 图 1.2.8 可视化累加器图像（另见彩插）

了一种不同的副作用）。一般来说，我们会仔细而完整地设计 API，并且一旦定型就不愿再对它做任何改动，因为这有可能会涉及修改无数用例（和实现）的代码。但添加一个构造函数来取得某些功能有时能够获得通过，因为它对用例的影响和改变类名所产生的变化相同。在本例中，如果已经开发了一个使用 Accumulator 的用例并大量调用了 addDataValue() 和 mean()，只需改变用例的一行代码就能享受到 VisualAccumulator 的优势。

表 1.2.14 一种能够累加数据的抽象数据类型（可视版本，另见彩插）

API	public class **VisualAccumulator**		
		VisualAccumulator(int trials, double max)	
	void	addDataValue(double val)	添加一个新的数据值
	double	mean()	所有数据的平均值
	String	toString()	对象的字符串表示

典型的用例

```
public class TestVisualAccumulator
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      VisualAccumulator a = new VisualAccumulator(T,1.0);
      for (int t = 0; t < T; t++)
         a.addDataValue(StdRandom.random());
      StdOut.println(a);
   }
}
```

数据类型的实现

```
public class VisualAccumulator
{
   private double total;
   private int N;
   public VisualAccumulator(int trials, double max)
   {
      StdDraw.setXscale(0, trials);
      StdDraw.setYscale(0, max);
      StdDraw.setPenRadius(.005);
   }
   public void addDataValue(double val)
   {
      N++;
      total += val;
      StdDraw.setPenColor(StdDraw.DARK_GRAY);
      StdDraw.point(N, val);
      StdDraw.setPenColor(StdDraw.RED);
      StdDraw.point(N, total/N);
   }
   public double mean()
   public String toString()
   // 和 Accumulator 相同
}
```

1.2.5 数据类型的设计

*抽象数据类型是一种向用例隐藏内部表示的数据类型。*这种思想强有力地影响了现代编程。我们遇到过的众多例子为我们研究抽象数据类型的高级特性和它们的Java实现打下了基础。简单看来，下面的许多话题和算法的学习关系不大，因此你可以跳过本节，在今后实现抽象数据类型中遇到特定问题时再回过头来参考它。我们的目的是将关于设计数据类型的重要知识集中起来以供参考，并为本书中的所有抽象数据类型的实现做铺垫。

1.2.5.1 封装

面向对象编程的特征之一就是使用数据类型的实现封装数据，以简化实现和隔离用例开发。封装实现了模块化编程，它允许我们：

- 独立开发用例和实现的代码；
- 切换至改进的实现而不会影响用例的代码；
- 支持尚未编写的程序（对于后续用例，API能够起到指南的作用）。

封装同时也隔离了数据类型的操作，这使我们可以：

- 限制潜在的错误；
- 在实现中添加一致性检查等调试工具；
- 确保用例代码更明晰。

一个封装的数据类型可以被任意用例使用，因此它扩展了Java语言。我们所提倡的编程风格是将大型程序分解为能够独立开发和调试的小型模块。这种方式将修改代码的影响限制在局部区域，改进了我们的软件质量。它也促进了代码复用，因为我们可以用某种数据类型的新实现代替老的实现来改进它的性能、准确度或是内存消耗。同样的思想也适用于许多其他领域。我们在使用系统库时常常从封装中受益。Java系统的新实现往往更新了多种数据类型或静态方法库的实现，但它们的API并没有变化。在算法和数据结构的学习中，我们总是希望开发出更好的算法，因为只需用抽象数据类型的改进实现替换老的实现即可在不改变任何用例代码的情况下改进所有用例的性能。模块化编程成功的关键在于保持模块之间的独立性。我们坚持将API作为用例和实现之间唯一的依赖点来做到这一点。*并不需要知道一个数据类型是如何实现的才能使用它，实现数据类型时也应该假设*
96 *使用者除了API什么也不知道。*封装是获得所有这些优势的关键。

1.2.5.2 设计API

构建现代软件最重要也最有挑战的一项任务就是设计API。它需要经验、思考和反复的修改，但设计一份优秀的API所付出的所有时间都能从调试和代码复用所节省的时间中获得回报。为一个小程序给出一份API似乎有些多余，但你应该按照能够复用的方式编写每个程序。理想情况下，一份API应该能够清楚地说明所有可能的输入和副作用，然后我们应该先写出检查实现是否与API相符的程序。但不幸的是，计算机科学理论中一个叫做*说明书问题*（specification problem）的基础结论说明这个目标是不可能实现的。简单地说，这样一份说明书应该用一种类似于编程语言的形式语言编写。而从数学上可以证明，判定这样两个程序进行的计算是否相同是不可能的。因此，我们的API将是与抽象数据类型相关联的值以及一系列构造函数和实例方法的目的和副作用的自然语言描述。为了验证我们的设计，我们会在API附近的正文中给出一些用例代码。但是，这些宏观概述之中也隐藏着每一份API设计都可能落入的无数陷阱。

- API可能会难以实现：实现的开发非常困难，甚至不可能。

- ❑ API 可能会难以使用：用例代码甚至比没有 API 时更复杂。
- ❑ API 的范围可能太窄：缺少用例所需的方法。
- ❑ API 的范围可能太宽：包含许多不会被任何用例调用的方法。这种缺陷可能是最常见的，并且也是最难以避免的。API 的大小一般会随着时间而增长，因为向已有的 API 中添加新方法很简单，但在不破坏已有用例程序的前提下从中删除方法却很困难。
- ❑ API 可能会太粗略：无法提供有效的抽象。
- ❑ API 可能会太详细：抽象过于细致或是发散而无法使用。
- ❑ API 可能会过于依赖某种特定的数据表示：用例代码可能会因此无法从数据表示的细节中解脱出来。要避免这种缺陷也是很困难的，因为数据表示显然是抽象数据类型实现的核心。 97

这些考虑有时又被总结为另一句格言：只为用例提供它们所需要的，仅此而已。

1.2.5.3 算法与抽象数据类型

数据抽象天生适合算法研究，因为它能够为我们提供一个框架，在其中能够准确地说明一个算法的目的以及其他程序应该如何使用该算法。在本书中，算法一般都是某个抽象数据类型的一个实例方法的实现。例如，本章开头的白名单例子就很自然地被实现为一个抽象数据类型的用例。它进行了以下操作：

- ❑ 由一组给定的值构造了一个 SET（集合）对象；
- ❑ 判定一个给定的值是否存在于该集合中。

这些操作封装在 StaticSETofInts 抽象数据类型中，和 Whitelist 用例一起显示在表 1.2.15 中。StaticSETofInts 是更一般也更有用的符号表抽象数据类型的一种特殊情况，符号表抽象数据类型将是第 3 章的重点。在我们研究过的所有算法中，二分查找是较为适合用于实现这些抽象数据类型的一种。和 1.1.10 节中的 BinarySearch 实现比较起来，这里的实现所产生的用例代码更加清晰和高效。例如，StaticSETofInts 强制要求数组在 rank() 方法被调用之前排序。有了抽象数据类型，我们可以将抽象数据类型的调用和实现区分开来，并确保任意遵守 API 的用例程序都能受益于二分查找算法（使用 BinarySearch 的程序在调用 rank() 之前必须能够将数组排序）。白名单应用是众多二分查找算法的用例之一。

每个 Java 程序都是一组静态方法和（或）一种数据类型的实现的集合。在本书中我们主要关注的是抽象数据类型的实现中的操作和向用例隐藏其中的数据表示，例如 StaticSETofInts。正如这个例子所示，数据抽象使我们能够：

- ❑ 准确定义算法能为用例提供什么；
- ❑ 隔离算法的实现和用例的代码；
- ❑ 实现多层抽象，用已知算法实现其他算法。

应用

```
% java Whitelist largeW.txt <
largeT.txt
499569
984875
295754
207807
140925
161828
...
```

表 1.2.15 将二分查找重写为一段面向对象的程序（用于在整数集合中进行查找的一种抽象数据类型）

API	public class	StaticSETofInts	
		StaticSETofInts(int[] a)	根据 a[] 中的所有值创建一个集合
	boolean	contains(int key)	key 是否存在于集合中

（续）

典型的用例

```
public class Whitelist
{
   public static void main(String[] args)
   {
      int[] w = In.readInts(args[0]);
      StaticSETofInts set = new StaticSETofInts(w);
      while (!StdIn.isEmpty())
      {  // 读取键，如果不在白名单中则打印它
         int key = StdIn.readInt();
         if (!set.contains(key))
            StdOut.println(key);
      }
   }
}
```

数据类型的实现

```
import java.util.Arrays;
public class StaticSETofInts
{
   private int[] a;
   public StaticSETofInts(int[] keys)
   {
      a = new int[keys.length];
      for (int i = 0; i < keys.length; i++)
         a[i] = keys[i]; // 保护性复制
      Arrays.sort(a);
   }
   public boolean contains(int key)
   {  return rank(key) != -1;  }
   private int rank(int key)
   {  // 二分查找
      int lo  = 0;
      int hi = a.length - 1;
      while (lo <= hi)
      {  // 键要么存在于a[lo..hi]中，要么不存在
         int mid = lo + (hi - lo) / 2;
         if      (key < a[mid]) hi = mid - 1;
         else if (key > a[mid]) lo = mid + 1;
         else                   return mid;
      }
      return -1;
   }
}
```

无论是使用自然语言还是伪代码描述算法，这些都是我们所希望拥有的性质。使用 Java 的类机制来支持数据的抽象将使我们收获良多：我们编写的代码将能够测试算法并比较各种用例程序的性能。

98 ~ 99

1.2.5.4 接口继承

Java 语言为定义对象之间的关系提供了支持，称为接口。程序员广泛使用这些机制，如果上过软件工程的课程那么你可以详细地研究一下它们。我们学习的第一种继承机制叫做子类型。它允许我们通过指定一个含有一组公共方法的接口为两个本来并没有关系的类建立一种联系，这两个类都必须实现这些方法。例如，如果不使用我们的非正式 API，也可以为 Date 声明一个接口：

```
public interface Datable
{
   int month();
   int day();
   int year();
}
```

并在我们的实现中引用该接口：

```
public class Date implements Datable
{
   // 实现代码（和以前一样）
}
```

这样，Java 编译器就会检查该实现是否和接口相符。为任意实现了 month()、day() 和 year() 的类添加 implements Datable 保证了所有用例都能用该类的对象调用这些方法。这种方式称为*接口继承*——实现类继承的是接口。接口继承使得我们的程序能够通过调用接口中的方法操作实现该接口的任意类型的对象（甚至是还未被创建的类型）。我们可以在更多非正式的 API 中使用接口继承，但为了避免代码依赖于和理解算法无关的高级语言特性以及额外的接口文件，我们并 100
没有这么做。在某些情况下 Java 的习惯用法鼓励我们使用接口：我们用它们进行比较和迭代，如表 1.2.16 所示。我们会在接触那些概念时再详细研究它们。

表 1.2.16 本书中所用到的 Java 接口

	接 口	方 法	章 节
比较	java.lang.Comparable	compareTo()	2.1
	java.util.Comparator	compare()	2.5
迭代	java.lang.Iterable	iterator()	1.3
	java.util.Iterator	hasNext() next() remove()	1.3

1.2.5.5 实现继承

Java 还支持另一种继承机制，被称为*子类*。这种非常强大的技术使程序员不需要重写整个类就能改变它的行为或者为它添加新的功能。它的主要思想是定义一个新类（*子类*，或称为*派生类*）来继承另一个类（*父类*，或称为*基类*）的所有实例方法和实例变量。子类包含的方法比父类更多。另外，子类可以重新定义或者*重写*父类的方法。子类继承被系统程序员广泛用于编写所谓*可扩展的库*——任何一个程序员（包括你）都能为另一个程序员（或者也许是一个系统程序员团队）创建的库添加方法。这种方法能够有效地重用潜在的十分庞大的库中的代码。例如，这种方法被广泛用于图形用户界面的开发，因此实现用户所需要的各种控件（下拉菜单，剪切—粘贴，文件访问等）的大量代码都能够被重用。子类继承的使用在系统程序员和应用程序员之间是有争议的（它和接口继承之间的优劣还没有定论）。在本书中我们会避免使用它，因为它会破坏封装。但这种机制是 Java 的一部分，因此它的残余是无法避免的：具体来说，每个类都是 Java 的 Object 类的子类。这种结构意味着每个类都含有 getClass()、toString()、equals()、hashCode()（见表 1.2.17）和另外几个我们不会在本书中用到的方法的实现。实际上，每个类都通过子类继承从 Object 类中继承了这些方法，因此任何用例都可以在任意对象中调用这些方法。我们通常会重载新类的 toString()、equals() 和 hashCode() 方法，因为 Object 类的默认实现一般无法提供所需的行为。接下来我们将讨论 toString() 和 equals()，在 3.4 节中讨论 hashCode()。

表 1.2.17 本书中所使用的由 Object 类继承得到的方法

方 法	作 用	章 节
Class getClass()	该对象的类是什么	1.2
String toString()	该对象的字符串表示	1.1
boolean equals(Object that)	该对象是否和 that 相等	1.2
101 int hashCode()	该对象的散列值	3.4

1.2.5.6 字符串表示的习惯

按照习惯，每个 Java 类型都会从 Object 继承 toString() 方法，因此任何用例都能够调用任意对象的 toString() 方法。当连接运算符的一个操作数是字符串时，Java 会自动将另一个操作数也转换为字符串，这个约定是这种自动转换的基础。如果一个对象的数据类型没有实现 toString() 方法，那么转换会调用 Obejct 的默认实现。默认实现一般都没有多大实用价值，因为它只会返回一个含有该对象内存地址的字符串。因此我们通常会为我们的每个类实现并重写默认的 toString() 方法，如下面代码框的 Date 类中加粗的部分所示。由代码可以看到，toString() 方法的实现通常很简单，只需隐式调用（通过 +）每个实例变量的 toString() 方法即可。

1.2.5.7 封装类型

Java 提供了一些内置的引用类型，称为封装类型。每种原始数据类型都有一个对应的封装类型：Boolean、Byte、Character、Double、Float、Integer、Long 和 Short 分别对应着 boolean、byte、char、double、float、int、long 和 short。这些类主要由类似于 parseInt() 这样的静态方法组成，但它们也含有继承得到的实例方法 toString()、compareTo()、equals() 和 hashCode()。在需要的时候 Java 会自动将原始数据类型转换为封装类型，如 1.3.1.1 节所述。例如，当一个 int 值需要和一个 String 连接时，它的类型会被转换为 Integer 并触发 toString() 方法。

1.2.5.8 等价性

两个对象相等意味着什么？如果我们用相同类型的两个引用变量 a 和 b 进行等价性测试（a == b），我们检测的是它们的标识是否相同，即引用是否相同。一般用例希望能够检查数据类型的值（对象的状态）是否相同或者实现某种针对该类型的规则。Java 为我们开了个头，为 Integer、Double 和 String 等标准数据类型以及一些如 File 和 URL 的复杂数据类型提供了实现。在处理这些类型的数据时，可以直接使用内置的实现。例如，如果 x 和 y 均为 String 类型的值，那么当且仅当 x 和 y 的长度相同且每个位置的字符均相同时 x.equals(y) 的返回值为 true。当我们在定义自己的数据类型时，比如 Date 或 Transaction，需要重载 equals() 方法。Java 约定 equals() 必须是一种等价性关系。它必须具有：

- 自反性，x.equals(x) 为 true；
- 对称性，当且仅当 y.equals(x) 为 true 时，x.equals(y) 返回 true；
- 传递性，如果 x.equals(y) 和 y.equals(z) 均为 true，x.equals(z) 也将为 true。

另外，它必须接受一个 Object 为参数并满足以下性质：

- 一致性，当两个对象均未被修改时，反复调用 x.equals(y) 总是会返回相同的值；
- 非空性，x.equals(null) 总是返回 false。

102 这些定义都是自然合理的，但确保这些性质成立并遵守 Java 的约定，同时又避免在实现时做无用功却并不容易，如 Date 所示。它通过以下步骤做到了这一点。

- 如果该对象的引用和参数对象的引用相同，返回 true。这项测试在成立时能够免去其他所有测试工作。
- 如果参数为空（null），根据约定返回 false（还可以避免在下面的代码中使用空引用）。
- 如果两个对象的类不同，返回 false。要得到一个对象的类，可以使用 getClass() 方法。请注意我们会使用 == 来判断 Class 类型的对象是否相等，因为同一种类型的所有对象的 getClass() 方法一定能够返回相同的引用。
- 将参数对象的类型从 Object 转换到 Date（因为前一项测试已经通过，这种转换必然成功）。
- 如果任意实例变量的值不相同，返回 false。对于其他类，等价性测试方法的定义可能不同。例如，我们只有在两个 Counter 对象的 count 变量相等时才会认为它们相等。

```
public class Date
{
   private final int month;
   private final int day;
   private final int year;

   public Date(int m, int d, int y)
   {  month = m; day = d; year = y; }

   public int month()
   {  return month;  }

   public int day()
   {  return day;  }

   public int year()
   {  return year;  }

   public String toString()
   {  return month() + "/" + day() + "/" + year();  }

   public boolean equals(Object x)
   {
      if (this == x) return true;
      if (x == null) return false;
      if (this.getClass() != x.getClass()) return false;
      Date that = (Date) x;
      if (this.day != that.day)           return false;
      if (this.month != that.month)       return false;
      if (this.year != that.year)         return false;
      return true;
   }

}
```

在数据类型的定义中重写toString()和equals()方法

你可以使用上面的实现作为实现任意数据类型的 equals() 方法的模板。只要实现一次 equals() 方法，下一次就不会那么困难了。

103

1.2.5.9 内存管理

我们可以为一个引用变量赋予一个新的值，因此一段程序可能会产生一个无法被引用的对象。例如，请看图 1.2.9 中所示的三行赋值语句。在第三行赋值语句之后，不仅 a 和 b 会指向同一个 Date 对象（1/1/2011），而且不存在能够引用初始化变量 b 的那个 Date 对象的引用了。本来该对

象的唯一引用就是变量 b，但是该引用被赋值语句覆盖了，这样的对象被称为孤儿。对象在离开作用域之后也会变成孤儿。Java 程序经常会创建大量对象（以及许多保存原始数据类型值的变量），但在某个时刻程序只会需要它们之中的一小部分。因此，编程语言和系统需要某种机制来在必要时为数据类型的值分配内存，而在不需要时释放它们的内存（对于一个对象来说，有时是在它变成孤儿之后）。内存管理对于原始数据类型更容易，因为内存分配所需要的所有信息在编译阶段就能够获取。Java（以及大多数其他系统）会在声明变量时为它们预留内存空间，并会在它们离开作用域后释放这些空间。对象的内存管理更加复杂：系统会在创建一个对象时为它分配内存，但是程序在执行时的动态性决定了一个对象何时才会变为孤儿，系统并不能准确地知道应该何时释放一个对象的内存。在许多语言中（例如 C 和 C++），分配和释放内存是程序员的责任。众所周知，这种操作既繁琐又容易出错。Java 最重要的一个特性就是自动内存管理。它通过记录孤儿对象并将它们的内存释放到内存池中将程序员从管理内存的责任中解放出来。这种回收内存的方式叫做垃圾回收。Java 的一个特点就是它不允许修改引用的策略。这种策略使 Java 能够高效自动地回收垃圾。程序员们至今仍在争论，为获得无需为内存管理操心的方便而付出的使用自动垃圾回收的代价是否值得。

```
Date a = new Date(12, 31, 1999);
Date b = new Date(1, 1, 2011);
a = b;
```

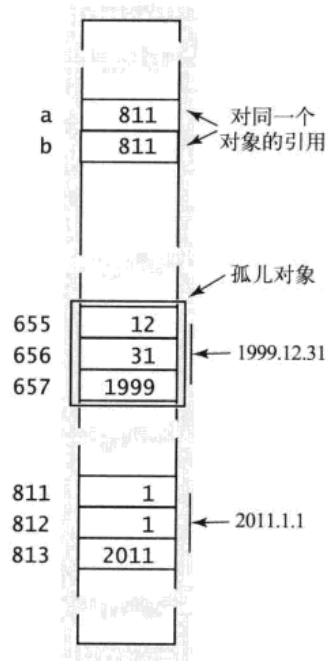


图 1.2.9 孤儿对象

104

1.2.5.10 不可变性

不可变数据类型，例如 Date，指的是该类型的对象中的值在创建之后就无法再被改变。与此相反，可变数据类型，例如 Counter 或 Accumulator，能够操作并改变对象中的值。Java 语言通过 final 修饰符来强制保证不可变性。当你将一个变量声明为 final 时，也就保证了只会对它赋值一次，可以用赋值语句，也可以用构造函数。试图改变 final 变量的值的代码将会产生一个编译时错误。在我们的代码中，我们用 final 修饰值不会改变的实例变量。这种策略就像文档一样，说明了这个变量的值不会再发生改变，它能够预防意外修改，也能使程序的调试更加简单。像 Date 这样实例变量均为原始数据类型且被 final 修饰的数据类型（按照约定，在不使用子类继承的代码中）是不可变的。数据类型是否可变是一个重要的设计决策，它取决于当前的应用场景。对于类似于 Date 的数据类型，抽象的目的是封装不变的值，以便将其和原始数据类型一样用于赋值语句、作为函数的参数或返回值（而不必担心它们的值会被改变）。程序员在使用 Date 时可能会写出操作两个 Date 类型的变量的代码 d = d0，就像操作 double 或者 int 值一样。但如果 Date 类型是可变的且 d 的值在 d = d0 之后可以被改变，那么 d0 的值也会被改变（它们都是指向同一个对象的引用）！从另一方面来说，对于类似于 Counter 和 Accumulator 的数据类型，抽象的目的是封装变化中的值。作为用例程序员，你在使用 Java 数组（可变）和 Java 的 String 类型（不可变）时就已经遇到了这种区别。将一个 String 传递给一个方法时，你不会担心该方法会改变字符串中的字符顺序，但当你把一个数组传递给一个方法时，方法可以自由改变数组的内容。String 对象是

不可变的，因为我们一般都不希望 String 的值改变，而 Java 数组是可变的，因为我们一般的确希望改变数组中的值。但也存在我们希望使用可变字符串（这就是 Java 的 StringBuilder 类存在的目的）和不可变数组（这就是稍后讨论的 Vector 类存在的目的）的情况。一般来说，不可变的数据类型比可变的数据类型使用更容易，误用更困难，因为能够改变它们的值的方式要少得多。调试使用不可变类型的代码更简单，因为我们更容易确保用例代码中使用它们的变量的状态前后一致。在使用可变数据类型时，必须时刻关注它们的值会在何时何地发生变化。而不可变性的缺点在于我们需要为每个值创建一个新对象。这种开销一般是可以接受的，因为 Java 的垃圾回收器通常都为此进行了优化。不可变性的另一个缺点在于，final 非常不幸地只能用来保证原始数据类型的实例变量的不可变性，而无法用于引用类型的变量。如果一个应用类型的实例变量含有修饰符 final，该实例变量的值（某个对象的引用）就永远无法改变了——它将永远指向同一个对象，但对象的值本身仍然是可变的。例如，这段代码并没有实现一个不可变的数据类型：

```
public class Vector
{
   private final double[] coords;

   public Vector(double[] a)
   {  coords = a; }
   ...
}
```

用例程序可以通过给定的数组创建一个 Vector 对象，并在构造函数执行之后（绕过 API）改变 Vector 中的元素的值：

```
double[] a = { 3.0, 4.0 };
Vector vector = new Vector(a);
a[0] = 0.0;  // 绕过了公有 API
```

实例变量 coords[] 是 private 和 final 的，但 Vector 是可变的，因为用例拥有指向数据的一个引用。任何数据类型的设计都需要考虑到不可变性，而且数据类型是否是不可变的则应该在 API 中说明，这样使用者才能知道该对象中的值是无法改变的。在本书中，我们对不可变性的主要兴趣在于用它保证我们的算法的正确性。例如，如果一个二分查找算法所使用的数据的类型是可变的，那么算法的用例就可能破坏我们对二分查找中的数组已经有序的假设。可变数据与不可变数据的示例见表 1.2.18。

表 1.2.18 可变与不可变数据类型举例

可变数据类型	不可变数据类型
Counter	Date
Java 数组	String

105
~
106

1.2.5.11 契约式设计

在最后，我们将简要讨论 Java 语言中能够在程序运行时检验程序状态的一些机制。为此我们将使用两种 Java 的语言特性：

- 异常（Exception），一般用于处理不受我们控制的不可预见的错误；
- 断言（Assertion），验证我们在代码中做出的一些假设。

大量使用异常和断言是很好的编程实践。为了节约版面我们在本书中极少使用它们，但你在本书网站上的所有代码中都会找到它们。这些代码中的每个和异常条件以及断言恒等式有关的算法周围都有大量的注释。

1.2.5.12 异常与错误

异常和错误都是在程序运行中出现的破坏性事件。Java 采取的行动称为抛出异常或是抛出错误。我们已经在学习 Java 的基本特性的过程中遇到过 Java 系统方法抛出的异常：

StackOverflowError、ArithmeticException、ArrayIndexOutOfBoundsException、OutOfMemoryError 和 NullPointerException 都是典型的例子。你也可以创建自己的异常，最简单的一种是 RuntimeException，它会中断程序的执行并打印出一条出错信息：

```
throw new RuntimeException("Error message here.");
```

一种叫做快速出错的常规编程实践提倡，一旦出错就立刻抛出异常，使定位出错位置更容易（这和忽略错误并将异常推迟到以后处理的方式相反）。

1.2.5.13 断言

断言是一条需要在程序的某处确认为 true 的布尔表达式。如果表达式的值为 false，程序将会终止并报告一条出错信息。我们使用断言来确定程序的正确性并记录我们的意图。例如，假设你计算得到一个值并可以将它作为索引访问一个数组。如果该值为负数，稍后它将会产生一条 ArrayIndexOutOfBoundsException 异常。但如果代码中有一句 assert index >= 0;，你就能找到出错的位置。还可以选择性地加上一条详细的消息来辅助定位 bug，例如：

```
assert index >= 0 : "Negative index in method X";
```

默认设置没有启用断言，可以在命令行下使用–enableassertions 标志（简写为–ea）启用断言。断言的作用是调试：程序在正常操作中不应该依赖断言，因为它们可能会被禁用。系统编程课程会
107 学习使用断言来保证代码永远不会被系统错误终止或是进入死循环。一种叫做契约式设计的编程模型采用的就是这种思想。数据类型的设计者需要说明前提条件（用例在调用某个方法前必须满足的条件）、后置条件（实现在方法返回时必须达到的要求）和副作用（方法可能对对象状态产生的任何其他变更）。在开发过程中，这些条件可以用断言进行测试。

1.2.5.14 小结

本节所讨论的语言机制说明实用数据类型的设计中所遇到的问题并不容易解决。专家们仍然在讨论支持某些我们已经学习过的设计理念的最佳方法。为什么 Java 不允许将函数作为参数？为什么 Matlab 会复制作为参数传递给函数的数组？正如本章前文所述，如果你总是抱怨编程语言的特性，那么你只能自己设计编程语言了。如果你不希望这样，最好的策略就是使用应用最广泛的编程语言。大多数系统都含有大量的库，在适当的时候你应该能用到它们，但通常你都能够通过构造易于移植到其他编程语言的抽象层来简化用例代码并进行自我保护。设计数据类型是你的主要目标，从而使
108 大多数工作都能在抽象层次完成，且和手头的问题匹配。

表 1.2.19 总结了我们讨论过的各种 Java 类。

表 1.2.19 Java 类（数据类型的实现）

类的类别	举 例	特 点
静态方法	Math StdIn StdOut	没有实例变量
不可变的抽象数据类型	Date Transaction String Integer	实例变量均为 private 实例变量均为 final 保护性复制引用类型数据 注意：这些都是必要但不充分条件
可变的抽象数据类型	Counter Accumulator	实例变量均为 private 并非所有实例变量均为 final
具有 I/O 副作用的抽象数据类型	VisualAccumulator In Out Draw	实例变量均为 private 实例方法会处理 I/O

109

答疑

问 为什么要使用数据抽象？

答 它能够帮助我们编写可靠而正确的代码。例如，在 2000 年的美国总统竞选中，Al Gore 在弗罗里达州的 Volusia 县的一个电子计票机上得到了 –16022 张选票——显然电子计票机软件中的选票计数器的封装不正确！

问 为什么要区别原始数据类型和引用类型？为什么不只用引用类型？

答 因为性能。Java 提供了 `Integer`、`Double` 等和原始数据类型对应的引用类型，以供希望忽略这些类型的区别的程序员使用。原始数据类型更接近计算机硬件所支持的数据类型，因此使用它们的程序比使用引用类型的程序运行得更快。

问 数据类型必须是抽象的吗？

答 不。Java 也支持 `public` 和 `protected` 来帮助用例直接访问实例变量。如正文所述，允许用例代码直接访问数据所带来的好处比不上对数据的特定表示方式的依赖所带来的坏处，因此我们代码中所有的实例变量都是私有的（`private`），有时也会使用私有实例方法在公有方法之间共享代码。

问 如果我在创建一个对象时忘记使用 new 关键字会发生什么？

答 对于 Java，这种代码看起来就好像你希望调用一个静态方法，却得到一个对象类型的返回值。因为并没有定义这样一个方法，你得到的错误信息和引用一个未定义的符号是一样的。如果编译这段代码：

```
Counter c = Counter("test");
```

会得到这条错误信息：

```
cannot find symbol
symbol  : method Counter(String)
```

如果你提供给构造函数的参数数量不对，也会得到相同的出错信息。 110

问 如果我在创建一个对象数组时忘记使用 new 关键字会发生什么？

答 创建每个对象都需要使用 new，所以要创建一个含有 N 个对象的数组，需要使用 N+1 次 new 关键字：创建数组需要一次，创建每个对象各需要一次。如果忘了创建数组：

```
Counter[] a;
a[0] = new Counter("test");
```

你得到的错误信息和尝试为一个未初始化的变量赋值是一样的：

```
variable a might not have been initialized
      a[0] = new Counter("test");
      ^
```

但如果在创建数组中的一个对象时忘了使用 new，然后又尝试调用它的方法，会得到一个 `NullPointerException`：

```
Counter[] a = new Counter[2];
a[0].increment();
```

问 为什么不用 `StdOut.println(x.toString())` 来打印对象？

答 这条语句也可以，但 Java 能够自动调用任意对象的 `toString()` 方法来帮我们省去这些麻烦，因为 `println()` 接受的参数是一个 `Object` 对象。

问 指针是什么？

答 问得好。或许上面那个异常应该叫做 `NullReferenceException`。和 Java 的引用一样，可以把指针看做机器地址。在许多编程语言中，指针是一种原始数据类型，程序员可以用各种方法操作它。

但众所周知，指针的编程非常容易出错，因此需要精心设计指针类的操作以帮助程序员避免错误。Java 将这种观点发挥到了极致（许多主流编程语言的设计者也赞同这种做法）。在 Java 中，创建引
111 用的方法只有一种（new），且改变引用的方法也只有一种（赋值语句）。也就是说，程序员能对引用进行的操作只有创建和复制。在编程语言的行话里，Java 的引用被称为*安全指针*，因为 Java 能够保证每个引用都会指向某种类型的对象（而且它能找出无用的对象并将其回收）。习惯于编写直接操作指针的程序员认为 Java 完全没有指针，但人们仍在为是否真的需要不安全的指针而争论。

问 我在哪里能够找到 Java 如何实现引用和进行垃圾收集的细节？

答 Java 系统的实现各有不同。例如，实现引用的一种自然方式是使用指针（机器地址）；而另一种使用的则可能是*句柄*（指针的指针）。前者访问数据的速度更快，而后者则能够更好地实现垃圾回收。

问 导入（import）一个对象名意味着什么？

答 没什么，只是可以少打一些字。如果不想使用 import 语句，你也可以在代码中用 java.util.Arrays 代替所有的 Arrays。

问 实现继承有什么问题？

答 子类继承阻碍模块化编程的原因有两点。第一，父类的任何改动都会影响它的所有子类。子类的开发不可能和父类*无关*。事实上，子类是*完全依赖于*父类的。这种问题被称为*脆弱的基类*问题。第二，子类代码可以访问所有实例变量，因此它们可能会扭曲父类代码的意图。例如，用于选票统计系统的 Counter 类的设计者可能会尽最大努力保证 Counter 每次只能将计数器加一（还记得 Al Gore 的问题吗）。但它的子类可以完全访问这个实例变量，因此可以将它改变为任意值。

问 怎样才能使一个类不可变？

答 要保证含有一个可变类型的实例变量的数据类型的不可变性，需要得到一个本地副本，这被称为*保护性复制*，但这也不一定能够达到目的。得到副本是一个方面，保证没有任何实例方法能够改变数据的值是另一方面。

112 问 什么是空（null）？

答 它是一个不指向任何对象的字面量。引用 null 调用一个方法是没有意义的，并且会产生 NullPointerException。如果你得到了这条错误信息，请检查并确认构造函数是否正确地初始化了类的所有实例变量。

问 实现某种数据类型的类中能否存在静态方法？

答 当然可以。例如，我们实现的所有类中都含有一个 main() 方法。另外，对于涉及多个对象的操作，如果它们都不是触发该方法的合适对象，那么就应该考虑添加一个静态方法。例如，我们可以在 Point 类中定义如下静态方法：

```
public static double distance(Point a, Point b)
{
   return a.distTo(b);
}
```

这种方法常常能够简化用例代码。

问 除了参数变量、局部变量和实例变量外还有其他种类的变量吗？

答 如果你在类的声明中包含了关键字 static（在其他类型之前），就创建了一种称为*静态变量*的完全不同的变量。和实例变量一样，类中的所有方法都可以访问静态变量，但静态变量却并不和任何具体的对象相关联。在较老的编程语言中，这种变量被称为*全局变量*，因为它们的作用域是全局的。在现代编程中，我们希望限制变量的作用域，因此很少使用这种变量。在使用它们时会非常小心。

问 什么是弃用（deprecated）的方法？

答 不再被支持但为了保持兼容性而留在 API 中的方法叫做弃用的方法。例如，Java 曾经包含了一个 `Character.isSpace()` 的方法，程序员也使用这个方法编写了一些程序。当 Java 的设计者们后来希望支持 Unicode 空白字符时，他们无法既改变 `isSpace()` 的行为又不损害用例程序。因此他们添加了一个新方法 `Character.isWhiteSpace()` 并放弃了老的方法。随着时间的推移，这种方式显然会使 API 更复杂。有时候甚至整个类都会被弃用。例如，Java 为了更好地支持国际化就将它的 `java.util.Date` 标记为弃用。 113

练习

1.2.1 编写一个 Point2D 的用例，从命令行接受一个整数 N。在单位正方形中生成 N 个随机点，然后计算两点之间的最近距离。

1.2.2 编写一个 Interval1D 的用例，从命令行接受一个整数 N。从标准输入中读取 N 个间隔（每个间隔由一对 `double` 值定义）并打印出所有相交的间隔对。

1.2.3 编写一个 Interval2D 的用例，从命令行接受参数 `N`、`min` 和 `max`。生成 `N` 个随机的 2D 间隔，其宽和高均匀地分布在单位正方形中的 `min` 和 `max` 之间。用 `StdDraw` 画出它们并打印出相交的间隔对的数量以及有包含关系的间隔对数量。

1.2.4 以下这段代码会打印出什么？

```
String string1 = "hello";
String string2 = string1;
string1 = "world";
StdOut.println(string1);
StdOut.println(string2);
```

1.2.5 以下这段代码会打印出什么？

```
String s = "Hello World";
s.toUpperCase();
s.substring(6, 11);
StdOut.println(s);
```

答：`"hello World"`。`String` 对象是不可变的——所有字符串方法都会返回一个新的 `String` 对象（但它们不会改变参数对象的值）。这段代码忽略了返回的对象并直接打印了原字符串。要打印出 `"WORLD"`，请用 `s = s.toUpperCase()` 和 `s = s.substring(6, 11)`。

1.2.6 如果字符串 `s` 中的字符循环移动任意位置之后能够得到另一个字符串 `t`，那么 `s` 就被称为 `t` 的回环变位（circular rotation）。例如，ACTGACG 就是 TGACGAC 的一个回环变位，反之亦然。判定这个条件在基因组序列的研究中是很重要的。编写一个程序检查两个给定的字符串 `s` 和 `t` 是否互为回环变位。提示：答案只需要一行用到 `indexOf()`、`length()` 和字符串连接的代码。 114

1.2.7 以下递归函数的返回值是什么？

```
public static String mystery(String s)
{
   int N = s.length();
   if (N <= 1) return s;
   String a = s.substring(0, N/2);
   String b = s.substring(N/2, N);
   return mystery(b) + mystery(a);
}
```

1.2.8 设 a[] 和 b[] 均为长数百万的整形数组。以下代码的作用是什么？有效吗？

```
int[] t = a; a = b; b = t;
```

答：这段代码会将它们交换。它的效率不可能再高了，因为它复制的是引用而不需要复制数百万个元素。

1.2.9 修改 BinarySearch（请见 1.1.10.1 节中的二分查找代码），使用 Counter 统计在有查找中被检查的键的总数并在查找全部结束后打印该值。提示：在 main() 中创建一个 Counter 对象并将它作为参数传递给 rank()。

1.2.10 编写一个类 VisualCounter，支持加一和减一操作。它的构造函数接受两个参数 N 和 max，其中 N 指定了操作的最大次数，max 指定了计数器的最大绝对值。作为副作用，用图像显示每次计数器变化后的值。

1.2.11 根据 Date 的 API 实现一个 SmartDate 类型，在日期非法时抛出一个异常。

1.2.12 为 SmartDate 添加一个方法 dayOfTheWeek()，为日期中每周的日返回 Monday、Tuesday、Wednesday、Thursday、Friday、Saturday 或 Sunday 中的适当值。你可以假定时间是 21 世纪。

1.2.13 用我们对 Date 的实现（请见表 1.2.12）作为模板实现 Transaction 类型。

115 ~ 116

1.2.14 用我们对 Date 中的 equals() 方法的实现（请见 1.2.5.8 节中的 Date 类代码框）作为模板，实现 Transaction 中的 equals() 方法。

提高题

1.2.15 文件输入。基于 String 的 split() 方法实现 In 中的静态方法 readInts()。

解答：

```
public static int[] readInts(String name)
{
   In in = new In(name);
   String input = StdIn.readAll();
    String[] words = input.split("\\s+");
   int[] ints = new int[words.length;
   for(int i = 0; i < word.length; i++)
      ints[i] = Integer.parseInt(words[i]);
   return ints;
}
```

我们会在 1.3 节中学习另一个不同的实现（请见 1.3.1.5 节）。

1.2.16 有理数。为有理数实现一个不可变数据类型 Rational，支持加减乘除操作。

public class **Rational**		
	Rational(int numerator, int denominator)	
Rational	plus(Rational b)	该数与 *b* 之和
Rational	minus(Rational b)	该数与 *b* 之差
Rational	times(Rational b)	该数与 *b* 之积
Rational	divides(Rational b)	该数与 *b* 之商
boolean	equals(Rational that)	该数与 that 相等吗
String	toString()	对象的字符串表示

无需测试溢出（请见练习 1.2.17），只需使用两个 long 型实例变量表示分子和分母来控制溢出

的可能性。使用欧几里德算法来保证分子和分母没有公因子。编写一个测试用例检测你实现的所有方法。 117

1.2.17 *有理数实现的健壮性*。在 Rational（请见练习 1.2.16）的开发中使用断言来防止溢出。

1.2.18 *累加器的方差*。以下代码为 Accumulator 类添加了 var() 和 stddev() 方法，它们计算了 addDataValue() 方法的参数的方差和标准差，验证这段代码。

```
public class Accumulator
{
   private double m;
   private double s;
   private int N;
   public void addDataValue(double x)
   {
      N++;
      s = s + 1.0 * (N-1) / N * (x - m) * (x - m);
      m = m + (x - m) / N;
   }
   public double mean()
   {  return m;  }
   public double var()
   {  return s/(N - 1);  }
   public double stddev()
   {  return Math.sqrt(this.var());  }
}
```

与直接对所有数据的平方求和的方法相比较，这种实现能够更好地避免四舍五入产生的误差。 118

1.2.19 *字符串解析*。为你在练习 1.2.13 中实现的 Date 和 Transaction 类型编写能够解析字符串数据的构造函数。它接受一个 String 参数指定的初始值，格式如表 1.2.20 所示：

表 1.2.20 被解析的字符串的格式

类 型	格 式	举 例
Date	由斜杠分隔的整数	5/22/1939
Transaction	客户、日期和金额，由空白字符分隔	Turing 5/22/1939 11.99

部分解答：

```
public Date(String date)
{
   String[] fields = date.split("/");
   month = Integer.parseInt(fields[0]);
   day   = Integer.parseInt(fields[1]);
   year  = Integer.parseInt(fields[2]);
}
```

119

1.3 背包、队列和栈

许多基础数据类型都和对象的集合有关。具体来说，数据类型的值就是一组对象的集合，所有操作都是关于添加、删除或是访问集合中的对象。在本节中，我们将学习三种这样的数据类型，分别是背包（Bag）、队列（Queue）和栈（Stack）。它们的不同之处在于删除或者访问对象的顺序不同。

背包、队列和栈数据类型都非常基础并且应用广泛。我们在本书的各种实现中也会不断用到它们。除了这些应用以外，本节中的实现和用例代码也展示了我们开发数据结构和算法的一般方式。

本节的第一个目标是说明我们对集合中的对象的表示方式将直接影响各种操作的效率。对于集合来说，我们将会设计适于表示一组对象的数据结构并高效地实现所需的方法。

本节的第二个目标是介绍*泛型*和*迭代*。它们都是简单的 Java 概念，但能极大地简化用例代码。它们是高级的编程语言机制，虽然对于算法的理解并不是必需的，但有了它们我们能够写出更加清晰、简洁和优美的用例（以及算法的实现）代码。

本节的第三个目标是介绍并说明*链式*数据结构的重要性，特别是经典数据结构*链表*，有了它我们才能高效地实现背包、队列和栈。理解链表是学习各种算法和数据结构中最关键的第一步。

对于这三种数据结构，我们都会学习其 API 和用例，然后再讨论数据类型的值的所有可能的表示方法以及各种操作的实现。这种模式会在全书中反复出现（且数据结构会越来越复杂）。这里的
120 实现是下文所有实现的模板，值得仔细研究。

1.3.1 API

照例，我们对集合型的抽象数据类型的讨论从定义它们的 API 开始，如表 1.3.1 所示。每份 API 都含有一个无参数的构造函数、一个向集合中添加单个元素的方法、一个测试集合是否为空的方法和一个返回集合大小的方法。Stack 和 Queue 都含有一个能够删除集合中的特定元素的方法。除了这些基本内容之外，我们将在以下几节中解释这几份 API 反映出的两种 Java 特性：*泛型*与*迭代*。

表 1.3.1 泛型可迭代的基础集合数据类型的 API

背包		
public class **Bag<Item>** implements Iterable<Item>		
	Bag()	创建一个空背包
void	add(Item item)	添加一个元素
boolean	isEmpty()	背包是否为空
int	size()	背包中的元素数量
先进先出（FIFO）队列		
public class **Queue<Item>** implements Iterable<Item>		
	Queue()	创建空队列
void	enqueue(Item item)	添加一个元素
Item	dequeue()	删除最近添加的元素
boolean	isEmpty()	队列是否为空
int	size()	队列中的元素数量

（续）

下压（后进先出，LIFO）栈		
public class **Stack<Item>** implements Iterable<Item>		
	Stack()	创建一个空栈
void	push(Item item)	添加一个元素
Item	pop()	删除最近添加的元素
boolean	isEmpty()	栈是否为空
int	size()	栈中的元素数量

121

1.3.1.1 泛型

集合类的抽象数据类型的一个关键特性是我们应该可以用它们存储任意类型的数据。一种特别的 Java 机制能够做到这一点，它被称为*泛型*，也叫做*参数化类型*。泛型对编程语言的影响非常深刻，许多语言并没有这种机制（包括早期版本的 Java）。在这里我们对泛型的使用仅限于一点额外的 Java 语法，非常容易理解。在每份 API 中，类名后的 <Item> 记号将 Item 定义为一个类型参数，它一个象征性的占位符，表示的是用例将会使用的某种具体数据类型。可以将 Stack<Item> 理解为*某种元素的栈*。在实现 Stack 时，我们并不知道 Item 的具体类型，但用例可以用我们的栈处理任意类型的数据，甚至是在我们的实现之后才出现的数据类型。在创建栈时，用例会提供一种具体的数据类型：我们可以将 Item 替换为任意引用数据类型（Item 出现的每个地方都是如此）。这种能力正是我们所需要的。例如，可以编写如下代码来用栈处理 String 对象：

```
Stack<String> stack = new Stack<String>();
stack.push("Test");
...
String next = stack.pop();
```

并在以下代码中使用队列处理 Date 对象：

```
Queue<Date> queue = new Queue<Date>();
queue.enqueue(new Date(12, 31, 1999));
...
Date next = queue.dequeue();
```

如果你尝试向 stack 变量中添加一个 Date 对象（或是任何其他非 String 类型的数据）或者向 queue 变量中添加一个 String 对象（或是任何其他非 Date 类型的数据），你会得到一个编译时错误。如果没有泛型，我们必须为需要收集的每种数据类型定义（并实现）不同的 API。有了泛型，我们只需要一份 API（和一次实现）就能够处理所有类型的数据，甚至是在未来定义的数据类型。你很快将会看到，使用泛型的用例代码很容易理解和调试，因此全书中我们都会用到它。

1.3.1.2 自动装箱

类型参数必须被实例化为引用类型，因此 Java 有一种特殊机制来使泛型代码能够处理原始数据类型。我们还记得 Java 的封装类型都是原始数据类型所对应的引用类型：Boolean、Byte、Character、Double、Float、Integer、Long 和 Short 分别对应着 boolean、byte、char、double、float、int、long 和 short。在处理赋值语句、方法的参数和算术或逻辑表达式时，Java 会自动在引用类型和对应的原始数据类型之间进行转换。在这里，这种转换有助于我们同时使用泛型和原始数据类型。例如：

122

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);       // 自动装箱 (int -> Integer)
int i = stack.pop(); // 自动拆箱 (Integer -> int)
```

自动将一个原始数据类型转换为一个封装类型被称为*自动装箱*，自动将一个封装类型转换为一个原始数据类型被称为*自动拆箱*。在这个例子中，当我们将一个原始类型的值 17 传递给 push() 方法时，Java 将它的类型自动转换（自动装箱）为 Integer。pop() 方法返回了一个 Integer 类型的值，Java 在将它赋予变量 i 之前将它的类型自动转换（自动拆箱）为了 int。

1.3.1.3 可迭代的集合类型

对于许多应用场景，用例的要求只是用某种方式处理集合中的每个元素，或者叫做*迭代访问*集合中的所有元素。这种模式非常重要，在 Java 和其他许多语言中它都是一级语言特性（不只是库，编程语言本身就含有特殊的机制来支持它）。有了它，我们能够写出清晰简洁的代码且不依赖于集合类型的具体实现。例如，假设用例在 Queue 中维护一个交易集合，如下：

```
Queue<Transaction> collection = new Queue<Transaction>();
```

如果集合是可迭代的，用例用一行语句即可打印出交易的列表：

```
for (Transaction t : collection)
{  StdOut.println(t);  }
```

这种语法叫做 foreach 语句：可以将 for 语句看做*对于集合中的每个交易 t(foreach)，执行以下代码段*。这段用例代码不需要知道集合的表示或实现的任何细节，它只想逐个处理集合中的元素。相同的 for 语句也可以处理交易的 Bag 对象或是任何可迭代的集合。很难想象还有比这更加清晰和简洁的代码。你将会看到，支持这种迭代需要在实现中添加额外的代码，但这些工作是值得的。

有趣的是，Stack 和 Queue 的 API 的唯一不同之处只是它们的名称和方法名。这让我们认识到无法简单地通过一列方法的签名说明一个数据类型的所有特点。在这里，只有自然语言的描述才能说明选择被删除元素（或是在 foreach 语句中下一个被处理的元素）的规则。这些规则的差异是 API 的重要组成部分，而且显然对用例代码的开发十分重要。

123

1.3.1.4 背包

背包是一种不支持从中删除元素的集合数据类型——它的目的就是帮助用例收集元素并迭代遍历所有收集到的元素（用例也可以检查背包是否为空或者获取背包中元素的数量）。迭代的顺序不确定且与用例无关。要理解背包的概念，可以想象一个非常喜欢收集弹子球的人。他将所有的弹子球都放在一个背包里，一次一个，并且会不时在所有的弹子球中寻找某一颗拥有某种特点的弹子球。使用 Bag 的 API，用例可以将元素添加进背包并根据需要随时使用 foreach 语句访问所有的元素。用例也可以使用栈或是队列，但使用 Bag 可以说明元素的处理顺序不重要。图 1.3.1 所示的 Stats 类是 Bag 的一个典型用例。

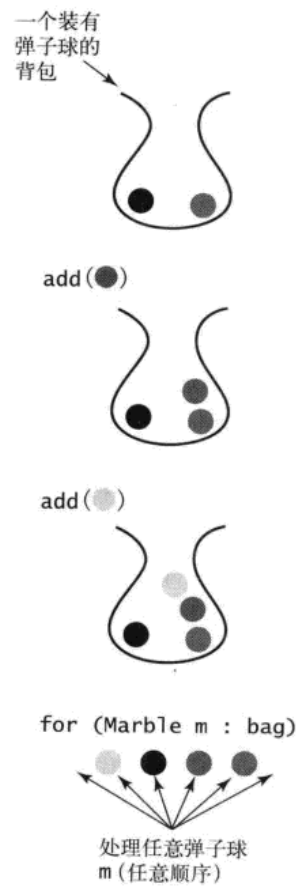


图 1.3.1 背包的操作(另见彩插)

它的任务是简单地计算标准输入中的所有 double 值的平均值和样本标准差。如果标准输入中有 N 个数字，那么平均值为它们的和除以 N，样本标准差为每个值和平均值之差的平方之和除以 $N-1$ 之后的平方根。在这些计算中，数的计算顺序和结果无关，因此我们将它们保存在一个 Bag 对象中并使用 foreach 语法来计算每个和。注意：不需要保存所有的数也可以计算标准差（就像我们在 Accumulator 中计算平均值那样——请见练习 1.2.18）。用 Bag 对象保存所有数字是更复杂的统计计算所必需的。 124

以下代码框列出的是常用的背包用例。

背包的典型用例

```
public class Stats
{
   public static void main(String[] args)
   {
      Bag<Double> numbers = new Bag<Double>();

      while (!StdIn.isEmpty())
         numbers.add(StdIn.readDouble());
      int N = numbers.size();

      double sum = 0.0;
      for (double x : numbers)
         sum += x;
      double mean = sum/N;

      sum = 0.0;
      for (double x : numbers)
         sum += (x - mean)*(x - mean);
      double std = Math.sqrt(sum/(N-1));

      StdOut.printf("Mean: %.2f\n", mean);
      StdOut.printf("Std dev: %.2f\n", std);

   }
}
```

使用方法

```
% java Stats
100
99
101
120
98
107
109
81
101
90

Mean: 100.60
Std dev: 10.51
```

125

1.3.1.5 先进先出队列

先进先出队列（或简称队列）是一种基于先进先出（FIFO）策略的集合类型，如图 1.3.2 所示。按照任务产生的顺序完成它们的策略我们每天都会遇到：在剧院门前排队的人们、在收费站前排队的汽车或是计算机上某种软件中等待处理的任务。任何服务性策略的基本原则都是公平。在提到公平时大多数人的第一个想法就是应该优先服务等待最久的人，这正是先进先出策略的准则。队列是许多日常现象的自然模型，它也是无数应用程序的核心。当用例使用 foreach 语句迭代访问队列中的元素时，元素的处理顺序就是它们被添加到队列中的顺序。在应用程序中使用队列的主要原因是在用集合保存元素的同时保存它们的相对顺序：使它们入列顺序和出列顺序相同。例如，下页的用例是我们的 In 类的静态方法 readInts() 的一种实现。这个方法为用例解决的问题是用例无需预先知道文件的大小即可将文件中的所有整数读入一个数组中。我们首先将所有的整数读入队列中，然后使用 Queue 的 size() 方法得到所需数组的大小，创建数组并将队列中的所有整数移动到数组中。队列之所以合适是因为它能够将整数按照文件中的顺序放入数组中（如果该顺序并不重要，也可以使用 Bag 对象）。这段代码使用了自动装箱和拆箱来转换用例中的 int 原始数据类型和队列的 Integer 封装类型。

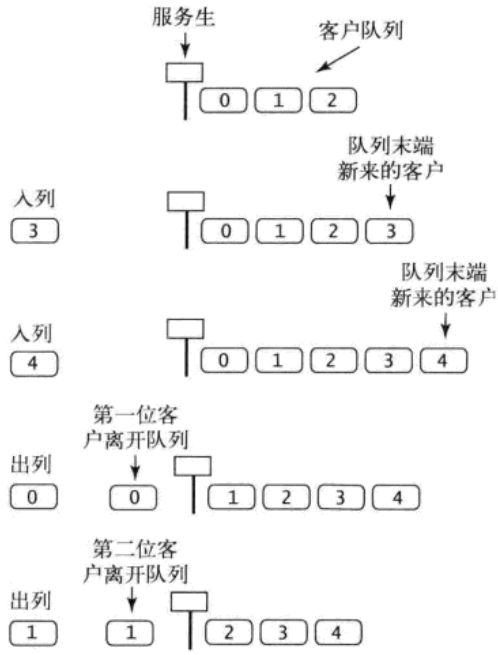


图 1.3.2 一个典型的先进先出队列

```
public static int[] readInts(String
name)
{
   In in = new In(name);
   Queue<Integer> q = new
Queue<Integer>();
   while (!in.isEmpty())
      q.enqueue(in.readInt());

   int N = q.size();
   int[] a = new int[N];
   for (int i = 0; i < N; i++)
      a[i] = q.dequeue();
   return a;
}
```

Queue 的用例

126

1.3.1.6 下压栈

下压栈（或简称栈）是一种基于后进先出（LIFO）策略的集合类型，如图 1.3.3 所示。当你的邮件在桌上放成一叠时，使用的就是栈。新邮件来到时你将它们放在最上面，当你有空时你会一封一封地从上到下阅读它们。现在人们应付的纸质品比以前少得多，但计算机上的许多常用程序遵循相同的组织原则。例如，许多人仍然用栈的方式存放电子邮件——在收信时将邮件压入（push）最顶端，在取信时从最顶端将它们弹出（pop），且第一封一定是最新的邮件（后进，先出）。这种策略的好处是我们能够及时看到感兴趣的邮件，坏处是如果你不把栈清空，某些较早的邮件可能永远也不会被阅读。你在网上冲浪时很可能会遇到栈的另一个例子。点击一个超链接，浏览器会显示一个新的页面（并将它压入一个栈）。你可以不断点击超链接并访问新页面，但总是可以通过点击“回退”按钮重新访问以前的页面（从栈中弹出）。栈的后进先出策略正好能够提供你所需要的行为。当用例使用 `foreach` 语句迭代遍历栈中的元素时，元素的处理顺序和它们被压入

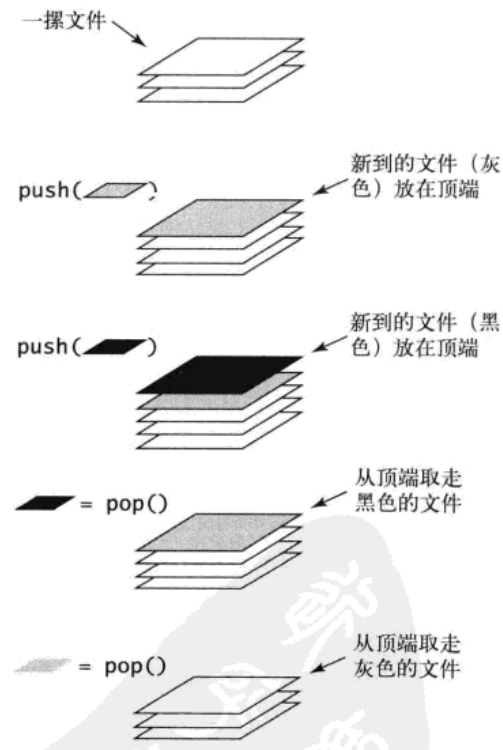


图 1.3.3 下压栈的操作

的顺序正好相反。在应用程序中使用栈迭代器的一个典型原因是在用集合保存元素的同时颠倒它们的相对顺序。例如，右侧的用例 Reverse 将会把标准输入中的所有整数逆序排列，同样它也无需预先知道整数的多少。在计算机领域，栈具有基础而深远的影响，下一节我们会仔细研究一个例子，以说明栈的重要性。

```
public class Reverse
{
   public static void main(String[] args)
   {
      Stack<Integer> stack;
      stack = new Stack<Integer>();
      while (!StdIn.isEmpty())
         stack.push(StdIn.readInt());

      for (int i : stack)
         StdOut.println(i);
   }
}
```

Stack 的用例

127

1.3.1.7 算术表达式求值

我们要学习的另一个栈用例同时也是展示泛型的应用的一个经典例子。我们在 1.1 节中最初学习的几个程序之一就是用来计算算术表达式的值的，例如：

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

如果将 4 乘以 5，把 3 加上 2，取它们的积然后加 1，就得到了 101。但 Java 系统是如何完成这些运算的呢？不需要研究 Java 系统的构造细节，我们也可以编写一个 Java 程序来解决这个问题。它接受一个输入字符串（表达式）并输出表达式的值。为了简化问题，首先来看一下这份明确的递归定义：*算术表达式*可能是一个数，或者是由一个左括号、一个算术表达式、一个运算符、另一个算术表达式和一个右括号组成的表达式。简单起见，这里定义的是*未省略括号*的算术表达式，它明确地说明了所有运算符的操作数——你可能更熟悉形如 1 + 2 * 3 的表达式，省略了括号，而采用优先级规则。我们将要学习的简单机制也能处理优先级规则，但在这里我们不想把问题复杂化。为了突出重点，我们支持最常见的二元运算符 *、+、– 和 /，以及只接受一个参数的平方根运算符 sqrt。我们也可以轻易支持更多数量和种类的运算符来计算多种大家熟悉的数学表达式，包括三角函数、指数和对数函数。我们的重点在于如何解析由括号、运算符和数字组成的字符串，并按照正确的顺序完成各种初级算术运算操作。如何才能够得到一个（由字符串表示的）算术表达式的值呢？E.W.Dijkstra 在 20 世纪 60 年代发明了一个非常简单的算法，用两个栈（一个用于保存运算符，一个用于保存操作数）完成了这个任务，其实现过程见下页，求值算法的轨迹如图 1.3.4 所示。

表达式由括号、运算符和操作数（数字）组成。我们根据以下 4 种情况从左到右逐个将这些实体送入栈处理：

- 将*操作数*压入操作数栈；
- 将*运算符*压入运算符栈；
- 忽略*左括号*；
- 在遇到*右括号*时，弹出一个运算符，弹出所需数量的操作数，并将运算符和操作数的运算结果压入操作数栈。

在处理完最后一个右括号之后，操作数栈上只会有一个值，它就是表达式的值。这种方法乍一看有些难以理解，但要证明它能够计算得到正确的值很简单：每当算法遇到一个被括号包围并由一个运算符和两个操作数组成的子表达式时，它都将运算符和操作数的计算结果压入操作数栈。这样的结果就好像在输入中用这个值代替了该子表达式，因此用这个值代替子表达式得到的结果和原表达式相同。我们可以反复应用这个规律并得到一个最终值。例如，用该算法计算以下表达式得到的结果都是相同的：

128

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
( 1 + ( 5 * ( 4 * 5 ) ) )
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

上一页中的 Evaluate 类是该算法的一个实现。这段代码是一个简单的“解释器”：一个能够解释给定字符串所表达的运算并计算得到结果的程序。

Dijkstra 的双栈算术表达式求值算法

```
public class Evaluate
{
   public static void main(String[] args)
   {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty())
      {  // 读取字符，如果是运算符则压入栈
         String s = StdIn.readString();
         if      (s.equals("("))               ;
         else if (s.equals("+"))    ops.push(s);
         else if (s.equals("-"))    ops.push(s);
         else if (s.equals("*"))    ops.push(s);
         else if (s.equals("/"))    ops.push(s);
         else if (s.equals("sqrt")) ops.push(s);
         else if (s.equals(")"))
         {  // 如果字符为 "}"，弹出运算符和操作数，计算结果并压入栈
            String op = ops.pop();
            double v = vals.pop();
            if      (op.equals("+"))    v = vals.pop() + v;
            else if (op.equals("-"))    v = vals.pop() - v;
            else if (op.equals("*"))    v = vals.pop() * v;
            else if (op.equals("/"))    v = vals.pop() / v;
            else if (op.equals("sqrt")) v = Math.sqrt(v);
            vals.push(v);
         }  // 如果字符既非运算符也不是括号，将它作为 double 值压入栈
         else vals.push(Double.parseDouble(s));
      }
      StdOut.println(vals.pop());
   }
}
```

这段 Stack 的用例使用了两个栈来计算表达式的值。它展示了一种重要的计算模型：将一个字符串解释为一段程序并执行该程序得到结果。有了泛型，我们只需实现 Stack 一次即可使用 String 值的栈和 Double 值的栈。简单起见，这段代码假设表达式没有省略任何括号，数字和字符均以空白字符相隔。

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0

% java Evaluate
( ( 1 + sqrt ( 5.0 ) ) / 2.0 )
1.618033988749895
```

129
~
130

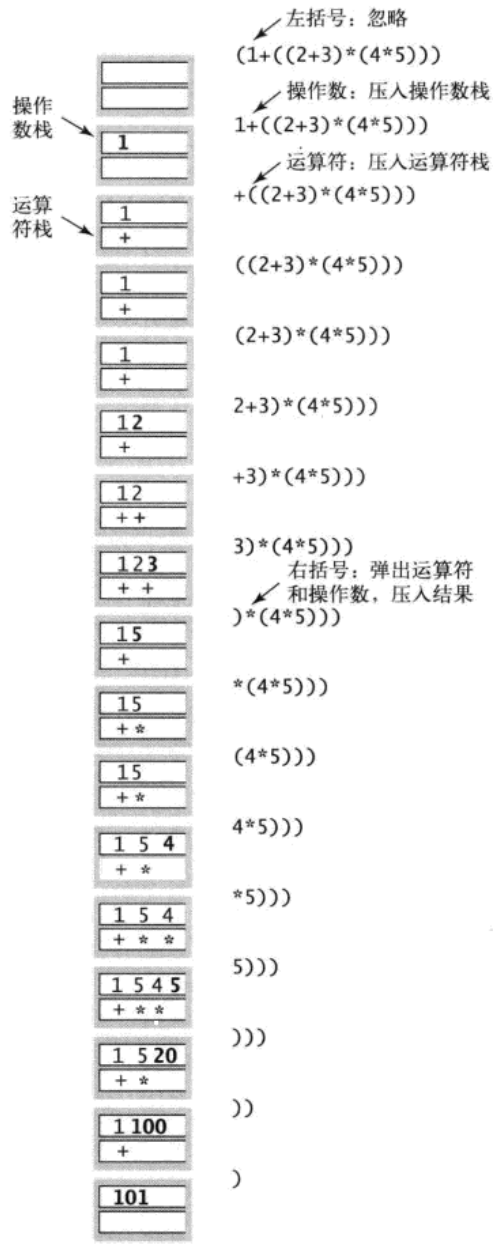


图 1.3.4 Dijkstra 的双栈算术表达式求值算法的轨迹 131

1.3.2 集合类数据类型的实现

在讨论 Bag、Stack 和 Queue 的实现之前，我们会先给出一个简单而经典的实现，然后讨论它的改进并得到表 1.3.1 中的 API 的所有实现。

1.3.2.1 定容栈

作为热身，我们先来看一种表示容量固定的字符串栈的抽象数据类型，如表 1.3.2 所示。它的 API 和 Stack 的 API 有所不同：它只能处理 String 值，它要求用例指定一个容量且不支持迭代。

实现一份 API 的第一步就是*选择数据的表示方式*。对于 FixedCapacityStackOfStrings，我们显然可以选择 String 数组。由此我们可以得到表 1.3.2 中底部的实现，它已经是简单得不能再简单了（每个方法都只有一行）。它的实例变量为一个用于保存栈中的元素的数组 a[]，和一个用于保存栈中的元素数量的整数 N。要删除一个元素，我们将 N 减 1 并返回 a[N]。要添加一个元素，我们将 a[N] 设为新元素并将 N 加 1。这些操作能够保证以下性质：

表 1.3.2 一种表示定容字符串栈的抽象数据类型

API	public class **FixedCapacityStackOfStrings**	
	FixedCapacityStackOfStrings(int cap)	创建一个容量为 cap 的空栈
void	push(String item)	添加一个字符串
String	pop()	删除最近添加的字符串
boolean	isEmpty()	栈是否为空
int	size()	栈中的字符串数量

测试用例

```
public static void main(String[] args)
{
   FixedCapacityStackOfStrings s;
   s = new FixedCapacityStackOfStrings(100);
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (!item.equals("-"))
           s.push(item);
      else if (!s.isEmpty()) StdOut.print(s.pop() + " ");
   }
   StdOut.println("(" + s.size() + " left on stack)");
}
```

使用方法

```
% more tobe.txt
to be or not to - be - - that - - - is
% java FixedCapacityStackOfStrings < tobe.txt
to be not that or be (2 left on stack)
```

数据类型的实现

```
public class FixedCapacityStackOfStrings
{
   private String[] a; // stack entries
   private int N;      // size
   public FixedCapacityStackOfStrings(int cap)
   {  a = new String[cap];  }
   public boolean isEmpty() {  return N == 0; }
   public int size()        {  return N; }
   public void push(String item)
   {  a[N++] = item; }
   public String pop()
   {  return a[--N]; }
}
```

- 数组中的元素顺序和它们被插入的顺序相同；
- 当 N 为 0 时栈为空；
- 栈的顶部位于 a[N-1]（如果栈非空）。

和以前一样，用恒等式的方式思考这些条件是检验实现正常工作的最简单的方式。请你务必完全理解这个实现。做到这一点的最好方法是检验一系列操作中栈内容的轨迹，如表 1.3.3 所示。测试用例会从标准输入读取多个字符串并将它们压入一个栈，当遇到 - 时它会将栈的内容弹出并打印结果。这种实现的主要性能特点是 push 和 pop 操作所需的时间独立于栈的长度。许多应用会因为这种简洁性而选择它。但几个缺点限制了它作为通用工具的潜力，我们要改进的也是这一点。经过一些修改（以及 Java 语言机制的一些帮助），我们就能给出一个适用性更加广泛的实现。这些努力是值得的，因为这个实现是本书中其他许多更强大的抽象数据类型的模板。

表 1.3.3 FixedCapacityStackOfStrings 的测试用例的轨迹

StdIn (*push*)	StdOut (*pop*)	N	a[]				
			0	1	2	3	4
		0					
to		1	to				
be		2	to	be			
or		3	to	be	or		
not		4	to	be	or	not	
to		5	to	be	or	not	to
-	to	4	to	be	or	not	to
be		5	to	be	or	not	be
-	be	4	to	be	or	not	be
-	not	3	to	be	or	not	be
that		4	to	be	or	that	be
-	that	3	to	be	or	that	be
-	or	2	to	be	or	that	be
-	be	1	to	be	or	that	be
is		2	to	is	or	not	to

132 ~ 133

1.3.2.2 泛型

FixedCapacityStackOfStrings 的第一个缺点是它只能处理 String 对象。如果需要一个 double 值的栈，你就需要用类似的代码实现另一个类，也就是把所有的 String 都替换为 double。这还算简单，但如果我们需要 Transaction 类型的栈或者 Date 类型的队列等，情况就很棘手了。如 1.3.1.1 节的讨论所示，Java 的参数类型（泛型）就是专门用来解决这个问题的，而且我们也看过了几个用例的代码（请见 1.3.1.4 节、1.3.1.5 节、1.3.1.6 节和 1.3.1.7 节）。但如何才能实现一个泛型的栈呢？表 1.3.4 中的代码展示了实现的细节。它实现了一个 FixedCapacityStack 类，该类和 FixedCapacityStackOfStrings 类的区别仅在于加粗部分的代码——我们把所有的 String 都替换为 Item（一个地方除外，会在稍后讨论）并用下面这行代码声明了该类：

```
public class FixedCapacityStack<Item>
```

Item 是一个类型参数，用于表示用例将会使用的某种具体类型的象征性的占位符。可以将 FixedCapacityStack<Item> 理解为某种元素的栈，这正是我们想要的。在实现 FixedCapacityStack 时，我们并不知道 Item 的实际类型，但用例只要能在创建栈时提供具体的数据类型，它就能用栈处理任意数据类型。实际的类型必须是引用类型，但用例可以依靠自动装箱将原始数据类型转换为相应的封装类型。Java 会使用类型参数 Item 来检查类型不匹配的错误——尽管具体的数据类型还不知道，赋予 Item 类型变量的值也必须是 Item 类型的，等等。在这里有一个细节非常重要：我们希望用以下代码在 FixedCapacityStack 的构造函数的实现中创建一个泛型的数组：

```
a = new Item[cap];
```

由于某些历史和技术原因（不在本书讲解范围之内），创建泛型数组在 Java 中是不允许的。我

们需要使用类型转换：

```
a = (Item[]) new Object[cap];
```

这段代码才能够达到我们所期望的效果（但 Java 编译器会给出一条警告，不过可以忽略它），
134 我们在本书中会一直使用这种方式（Java 系统库中类似抽象数据类型的实现中也使用了相同的方式）。

表 1.3.4 一种表示泛型定容栈的抽象数据类型

API	public class **FixedCapacityStack**<Item>	
	FixedCapacityStack(int cap)	创建一个容量为 cap 的空栈
void	push(**Item** item)	添加一个字符串
Item	pop()	删除最近添加的字符串
boolean	isEmpty()	栈是否为空
int	size()	栈中的元素数量

测试用例

```
public static void main(String[] args)
{
   FixedCapacityStack<String> s;
   s = new FixedCapacityStack<String>(100);
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (!item.equals("-"))
           s.push(item);
      else if (!s.isEmpty()) StdOut.print(s.pop() + " ");
   }
   StdOut.println("(" + s.size() + " left on stack)");
}
```

使用方法

```
% more tobe.txt
to be or not to - be - - that - - - is
% java FixedCapacityStack < tobe.txt
to be not that or be (2 left on stack)
```

数据类型的实现

```
public class FixedCapacityStack<Item>
{
   private Item[] a;   // stack entries
   private int N;      // size
   public FixedCapacityStack(int cap)
   {  a = (Item[]) new Object[cap];  }
   public boolean isEmpty() {  return N == 0; }
   public int size()        {  return N; }
   public void push(Item item)
   {  a[N++] = item; }
   public Item pop()
   {  return a[--N]; }
}
```

135

1.3.2.3 调整数组大小

选择用数组表示栈内容意味着用例必须预先估计栈的最大容量。在 Java 中，数组一旦创建，其大小是无法改变的，因此栈使用的空间只能是这个最大容量的一部分。选择大容量的用例在栈为空或几乎为空时会浪费大量的内存。例如，一个交易系统可能会涉及数十亿笔交易和数千个交易的集合。即使这种系统一般都会限制每笔交易只能出现在一个集合中，但用例必须保证所有集合都有能力保存所有的交易。另一方面，如果集合变得比数组更大那么用例有可能溢出。为此，push() 方法需要在代码中检测栈是否已满，我们的 API 中也应该含有一个 isFull() 方法来允许用例检测栈是否已满。我们在此省略了它的实现代码，因为我们希望用例从处理栈已满的问题中解脱出来，如我们的原始 Stack API 所示。因此，我们修改了数组的实现，动态调整数组 a[] 的大小，使得它既足以保存所有元素，又不至于浪费过多的空间。实际上，完成这些目标非常简单。首先，实现一个方法将栈移动到另一个大小不同的数组中：

```
private void resize(int max)
{  // 将大小为 N <= max 的栈移动到一个新的大小为 max 的数组中
   Item[] temp = (Item[]) new Object[max];
   for (int i = 0; i < N; i++)
      temp[i] = a[i];
   a = temp;
}
```

现在，在 push() 中，检查数组是否太小。具体来说，我们会通过检查栈大小 N 和数组大小 a.length 是否相等来检查数组是否能够容纳新的元素。如果没有多余的空间，我们会将数组的长度加倍。然后就可以和从前一样用 a[N++] = item 插入新元素了：

```
public void push(String item)
{  // 将元素压入栈顶
   if (N == a.length) resize(2*a.length);
   a[N++] = item;
}
```

类似，在 pop() 中，首先删除栈顶的元素，然后如果数组太大我们就将它的长度减半。只要稍加思考，你就明白正确的检测条件是栈大小是否小于数组的四分之一。在数组长度被减半之后，它的状态约为半满，在下次需要改变数组大小之前仍然能够进行多次 push() 和 pop() 操作。 136

```
public String pop()
{  // 从栈顶删除元素
   String item = a[--N];
   a[N] = null;  // 避免对象游离（请见下节）
   if (N > 0 && N == a.length/4) resize(a.length/2);
   return item;
}
```

在这个实现中，栈永远不会溢出，使用率也永远不会低于四分之一（除非栈为空，那种情况下数组的大小为 1）。我们会在 1.4 节中详细分析这种实现方法的性能特点。

push() 和 pop() 操作中数组大小调整的轨迹见表 1.3.5。

1.3.2.4 对象游离

Java 的垃圾收集策略是回收所有无法被访问的对象的内存。在我们对 pop() 的实现中，被弹出的元素的引用仍然存在于数组中。这个元素实际上已经是一个孤儿了——它永远也不会再被访问了，但 Java 的垃圾收集器没法知道这一点，除非该引用被覆盖。即使用例已经不再需要这个元素了，数组中的引用仍然可以让它继续存在。这种情况（保存一个不需要的对象的引用）

称为游离。在这里，避免对象游离很容易，只需将被弹出的数组元素的值设为 null 即可，这将覆盖无用的引用并使系统可以在用例使用完被弹出的元素后回收它的内存。

表 1.3.5 一系列 push() 和 pop() 操作中数组大小调整的轨迹

push()	pop()	N	a.length	a[]							
				0	1	2	3	4	5	6	7
		0	1	null							
to		1	1	to							
be		2	2	to	be						
or		3	4	to	be	or	null				
not		4	4	to	be	or	not				
to		5	8	to	be	or	not	to	null	null	null
-	to	4	8	to	be	or	not	null	null	null	null
be		5	8	to	be	or	not	be	null	null	null
-	be	4	8	to	be	or	not	null	null	null	null
-	not	3	8	to	be	or	null	null	null	null	null
that		4	8	to	be	or	that	null	null	null	null
-	that	3	8	to	be	or	null	null	null	null	null
-	or	2	4	to	be	null	null				
-	be	1	2	to	null						
is		2	2	to	is						

137

1.3.2.5 迭代

本节开头已经提过，集合类数据类型的基本操作之一就是，能够使用 Java 的 foreach 语句通过迭代遍历并处理集合中的每个元素。这种方式的代码既清晰又简洁，且不依赖于集合数据类型的具体实现。在讨论迭代的实现之前，我们先看一段能够打印出一个字符串集合中的所有元素的用例代码：

```
Stack<String> collection = new Stack<String>();
...
for (String s : collection)
   StdOut.println(s);
...
```

这里，foreach 语句只是 while 语句的一种简写方式（就好像 for 语句一样）。它本质上和以下 while 语句是等价的：

```
Iterator<String> i = collection.iterator();
while (i.hasNext())
{
   String s = i.next();
   StdOut.println(s);
}
```

这段代码展示了一些在任意可迭代的集合数据类型中我们都需要实现的东西：

- 集合数据类型必须实现一个 iterator() 方法并返回一个 Iterator 对象；
- Iterator 类必须包含两个方法：hasNext()（返回一个布尔值）和 next()（返回集合中的一个泛型元素）。

在 Java 中，我们使用接口机制来指定一个类所必须实现的方法（请见 1.2.5.4 节）。对于可迭代的集合数据类型，Java 已经为我们定义了所需的接口。要使一个类可迭代，第一步就是在它的声

明中加入 implements Iterable<Item>，对应的接口（即 java.lang.Iterable）为：

```
public interface Iterable<Item>
{
   Iterator<Item> iterator();
}
```

然后在类中添加一个方法 iterator() 并返回一个迭代器 Iterator<Item>。迭代器都是泛型的，因此我们可以使用参数类型 Item 来帮助用例遍历它们指定的任意类型的对象。对于一直使用的数组表示法，我们需要逆序迭代遍历这个数组，因此我们将迭代器命名为 ReverseArrayIterator，并添加了以下方法： 138

```
public Iterator<Item> iterator()
{  return new ReverseArrayIterator();  }
```

迭代器是什么？它是一个实现了 hasNext() 和 next() 方法的类的对象，由以下接口所定义（即 java.util.Iterator）：

```
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
    void remove();
}
```

尽管接口指定了一个 remove() 方法，但在本书中 remove() 方法总为空，因为我们希望避免在迭代中穿插能够修改数据结构的操作。对于 ReverseArrayIterator，这些方法都只需要一行代码，它们实现在栈类的一个嵌套类中：

```
private class ReverseArrayIterator implements Iterator<Item>
{
   private int i = N;

   public boolean hasNext() {  return i > 0;   }
   public Item next()       {  return a[--i];  }
   public void remove()     {                  }
}
```

请注意，嵌套类可以访问包含它的类的实例变量，在这里就是 a[] 和 N（这也是我们使用嵌套类实现迭代器的主要原因）。从技术角度来说，为了和 Iterator 的结构保持一致，我们应该在两种情况下抛出异常：如果用例调用了 remove() 则抛出 UnsupportedOperationException，如果用例在调用 next() 时 i 为 0 则抛出 NoSuchElementException 。因为我们只会在 foreach 语法中使用迭代器，这些情况都不会出现，所以我们省略了这部分代码。还剩下一个非常重要的细节，我们需要在程序的开头加上下面这条语句：

```
import java.util.Iterator;
```

因为（某些历史原因）Iterator 不在 java.lang 中（尽管 Iterable 是 java.lang 的一部分）。现在，使用 foreach 处理该类的用例能够得到的行为和使用普通的 for 循环访问数组一样，但它无知道数据的表示方法是数组（即实现细节）。对于我们在本书中学习的和 Java 库中所包含 139
的所有类似于集合的基础数据类型的实现，这一点非常重要。例如，我们无需改变任何用例代码就可以随意切换不同的表示方法。更重要的是，从用例的角度来来说，无需知晓类的实现细节用例也能使用迭代。

算法 1.1 是 Stack API 的一种能够动态改变数组大小的实现。用例能够创建任意类型数据的栈，并支持用例用 foreach 语句按照后进先出的顺序迭代访问所有栈元素。这个实现的基础

是 Java 的语言特性，包括 Iterable 和 Iterator，但我们没有必要深究这些特性的细节，因为代码本身并不复杂，并且可以用做其他集合数据类型的实现的模板。

例如，我们在实现 Queue 的 API 时，可以使用两个实例变量作为索引，一个变量 head 指向队列的开头，一个变量 tail 指向队列的结尾，如表 1.3.6 所示。在删除一个元素时，使用 head 访问它并将 head 加 1；在插入一个元素时，使用 tail 保存它并将 tail 加 1。如果某个索引在增加之后越过了数组的边界则将它重置为 0。实现检查队列是否为空、是否充满并需要调整数组大小的细节是一项有趣而又实用的编程练习（请见练习 1.3.14）。

表 1.3.6 ResizingArrayQueue 的测试用例的轨迹

StdIn	StdOut	N	head	tail	a[]							
（入列）	（出列）				0	1	2	3	4	5	6	7
		5	0	5	to	be	or	not	to			
-	to	4	1	5	to	be	or	not	to			
be		5	1	6	to	be	or	not	to	be		
-	be	4	2	6	to	be	or	not	to	be		
-	or	3	3	6	to	be	or	not	to	be		

在算法的学习中，算法 1.1 十分重要，因为它几乎（但还没有）达到了任意集合类数据类型的实现的最佳性能：

- 每项操作的用时都与集合大小无关；
- 空间需求总是不超过集合大小乘以一个常数。

ResizingArrayQueue 的缺点在于某些 push() 和 pop() 操作会调整数组的大小：这项操作的耗时和栈大小成正比。下面，我们将学习一种克服该缺陷的方法，使用一种完全不同的方式来组织数据。

140

算法 1.1 下压（LIFO）栈（能够动态调整数组大小的实现）

```
import java.util.Iterator;
public class ResizingArrayStack<Item> implements Iterable<Item>
{
   private Item[] a = (Item[]) new Object[1];  // 栈元素
   private int N = 0;                          // 元素数量
   public boolean isEmpty()  { return N == 0; }
   public int size()         { return N;      }
   private void resize(int max)
   {  // 将栈移动到一个大小为 max 的新数组
      Item[] temp = (Item[]) new Object[max];
      for (int i = 0; i < N; i++)
         temp[i] = a[i];
      a = temp;
   }
   public void push(Item item)
   {  // 将元素添加到栈顶
      if (N == a.length) resize(2*a.length);
      a[N++] = item;
   }
   public Item pop()
   {  // 从栈顶删除元素
      Item item = a[--N];
```

```
        a[N] = null;  // 避免对象游离（请见 1.3.2.4 节）
        if (N > 0 && N == a.length/4) resize(a.length/2);
        return item;
    }
    public Iterator<Item> iterator()
     {  return new ReverseArrayIterator();  }
    private class ReverseArrayIterator implements Iterator<Item>
    {  // 支持后进先出的迭代
        private int i = N;
        public boolean hasNext() {  return i > 0;   }
        public    Item next()    {  return a[--i];  }
        public    void remove()  {                  }
    }
}
```

这份泛型的可迭代的 Stack API 的实现是所有集合类抽象数据类型实现的模板。它将所有元素保存在数组中，并动态调整数组的大小以保持数组大小和栈大小之比小于一个常数。 141

1.3.3 链表

现在我们来学习一种基础数据结构的使用，它是在集合类的抽象数据类型实现中表示数据的合适选择。这是我们构造非 Java 直接支持的数据结构的第一个例子。我们的实现将成为本书中其他更加复杂的数据结构的构造代码的模板。所以请仔细阅读本节，即使你已经使用过链表。

定义。链表是一种递归的数据结构，它或者为空（null），或者是指向一个结点（node）的引用，该结点含有一个泛型的元素和一个指向另一条链表的引用。

在这个定义中，结点是一个可能含有任意类型数据的抽象实体，它所包含的指向结点的应用显示了它在构造链表之中的作用。和递归程序一样，递归数据结构的概念一开始也令人费解，但其实它的简洁性赋予了它巨大的价值。

1.3.3.1 结点记录

在面向对象编程中，实现链表并不困难。我们首先用一个嵌套类来定义结点的抽象数据类型：

```
private class Node
{
   Item item;
   Node next;
}
```

一个 Node 对象含有两个实例变量，类型分别为 Item（参数类型）和 Node。我们会在需要使用 Node 类的类中定义它并将它标记为 private，因为它不是为用例准备的。和任意数据类型一样，我们通过 new Node() 触发（无参数的）构造函数来创建一个 Node 类型的对象。调用的结果是一个指向 Node 对象的引用，它的实例变量均被初始化为 null。Item 是一个占位符，表示我们希望用链表处理的任意数据类型（我们将会使用 Java 的泛型使之表示任意引用类型）；Node 类型的实例变量显示了这种数据结构的链式本质。为了强调我们在组织数据时只使用了 Node 类，我们没有定义任何方法且会在代码中直接引用实例变量：如果 first 是一个指向某个 Node 对象的变量，我们可以使用 first.item 和 first.node 访问它的实例变量。这种类型的类有时也被称为记录。它们实现的不是抽象数据类型因为我们会直接使用其实例变量。但是在我们的实现中，Node 和它的用例代码都会被封装在相同的类中且无法被该类的用例访问，所以

142 我们仍然能够享受数据抽象的好处。

1.3.3.2 构造链表

现在，根据递归定义，我们只需要一个 Node 类型的变量就能表示一条链表，只要保证它的值是 null 或者指向另一个 Node 对象且该对象的 next 域指向了另一条链表即可。例如，要构造一条含有元素 to、be 和 or 的链表，我们首先为每个元素创造一个结点：

```
Node first  = new Node();
Node second = new Node();
Node third  = new Node();
```

并将每个结点的 item 域设为所需的值（简单起见，我们假设在这些例子中 Item 为 String）：

```
first.item  = "to";
second.item = "be";
third.item  = "or";
```

然后设置 next 域来构造链表：

```
first.next  = second;
second.next = third;
```

（注意：third.next 仍然是 null，即对象创建时它被初始化的值。）结果是，third 是一条链表（它是一个结点的引用，该结点指向 null，即一个空链表），second 也是一条链表（它是一个结点的引用，且该结点含有一个指向 third 的引用，而 third 是一条链表），first 也是一条链表（它是一个结点的引用，且该结点含有一个指向 second 的引用，而 second 是一条链表）。图 1.3.5 所示的代码以不同的顺序完成了这些赋值语句。

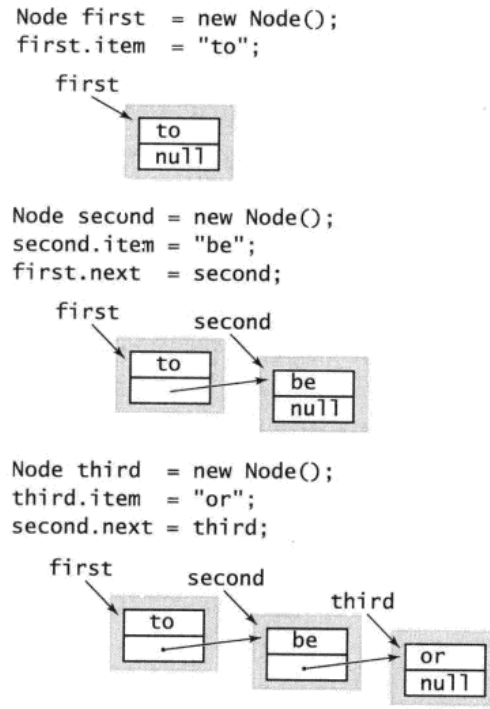


图 1.3.5 用链接构造一条链表

链表表示的是一列元素。在我们刚刚考察过的例子中，first 表示的序列是 to、be、or。我们也可以用一个数组来表示一列元素。例如，可以用以下数组表示同一列字符串：

```
String[] s = { "to", "be", "or" };
```

不同之处在于，在链表中向序列插入元素或是从序列中删除元素都更方便。下面，我们来学习完
143 成这些任务的代码。

在追踪使用链表和其他链式结构的代码时，我们会使用可视化表示方法：

- 用长方形表示对象；
- 将实例变量的值写在长方形中；
- 用指向被引用对象的箭头表示引用关系。

这种表示方式抓住了链表的关键特性。方便起见，我们用术语链接表示对结点的引用。简单起见，当元素的值为字符串时（如我们的例子所示），我们会将字符串写在长方形之内，而非使用 1.2 节中所讨论的更准确的方式表示字符串对象和字符数组。这种可视化的表示方式使我们能够将注意力集中在链表上。

1.3.3.3 在表头插入结点

首先，假设你希望向一条链表中插入一个新的结点。最容易做到这一点的地方就是链表的开头。例如，要在首结点为 first 的给定链表开头插入字符串 not，我们先将 first 保存在 oldfirst 中，然后将一个新结点赋予 first，并将它的 item 域设为 not，next 域设为 oldfirst。以上过程如图 1.3.6 所示。这段在链表开头插入一个结点的代码只需要几行赋值语句，所以它所需的时间和链表的长度无关。

1.3.3.4 从表头删除结点

接下来，假设你希望删除一条链表的首结点。这个操作更简单：只需将 first 指向 first.next 即可。一般来说你可能会希望在赋值之前得到该元素的值，因为一旦改变了 first 的值，就再也无法访问它曾经指向的结点了。曾经的结点对象变成了一个孤儿，Java 的内存管理系统最终将回收它所占用的内存。和以前一样，这个操作只含有一条赋值语句，因此它的运行时间和链表的长度无关。此过程如图 1.3.7 所示。

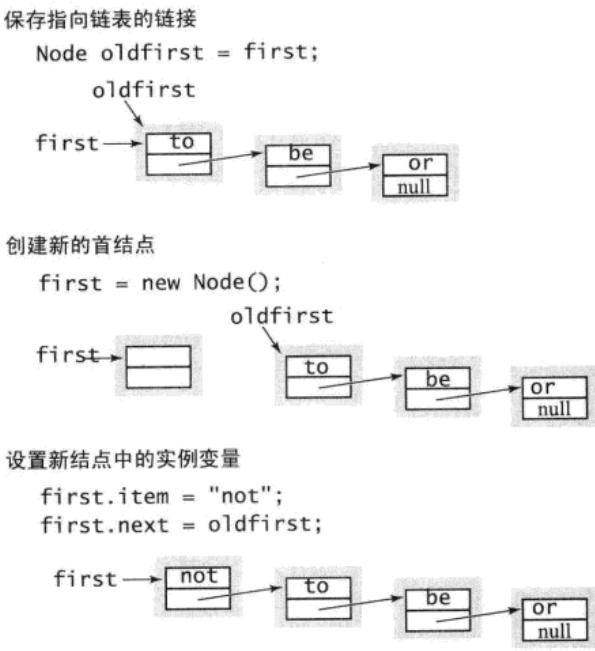


图 1.3.6 在链表的开头插入一个新结点

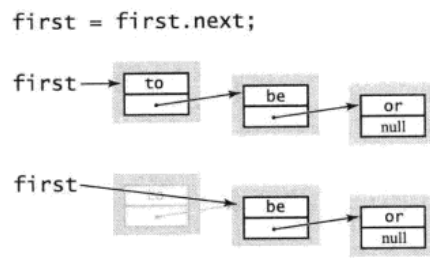


图 1.3.7 删除链表的首结点

144

1.3.3.5 在表尾插入结点

如何才能在链表的尾部添加一个新结点？要完成这个任务，我们需要一个指向链表最后一个结点的链接，因为该结点的链接必须被修改并指向一个含有新元素的新结点。我们不能在链表代码中草率地决定维护一个额外的链接，因为每个修改链表的操作都需要添加检查是否要修改该变量（以及作出相应修改）的代码。例如，我们刚刚讨论过的删除链表首结点的代码就可能改变指向链表的尾结点的引用，因为当链表中只有一个结点时，它既是首结点又是尾结点！另外，这段代码也无法处理链表为空的情况（它会使用空链接）。类似这些情况的细节使链表代码特别难以调试。在链表结尾插入新结点的过程如图 1.3.8 所示。

1.3.3.6 其他位置的插入和删除操作

总的来说，我们已经展示了在链表中如何通过若干指令实现以下操作，其中我们可以通过 first 链接访问链表的首结点并通过 last 链接访问链表的尾结点：

- 在表头插入结点；
- 从表头删除结点；
- 在表尾插入结点。

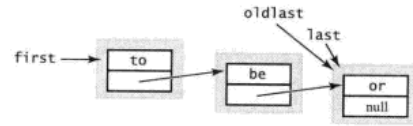
其他操作，例如以下几种，就不那么容易实现了：

- 删除指定的结点；
- 在指定结点前插入一个新结点。

例如，我们怎样才能删除链表的尾结点呢？ last 链接帮不上忙，因为我们需要将链表尾结点的前一个结点中的链接（它指向的正是 last）值改为 null。在缺少其他信息的情况下，唯一的解决办法
145 就是遍历整条链表并找出指向 last 的结点（请见下文以及练习 1.3.19）。这种解决方案并不是我们想要的，因为它所需的时间和链表的长度成正比。实现任意插入和删除操作的标准解决方案是使用*双向链表*，其中每个结点都含有两个链接，分别指向不同的方向。我们将实现这些操作的代码留做练习（请见练习 1.3.31）。我们的所有实现都不需要双向链表。

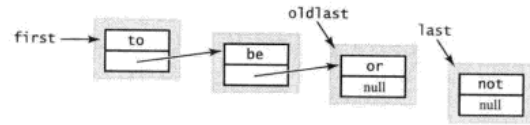
保存指向尾结点的链接

```
Node oldlast = last;
```



创建新的尾结点

```
last = new Node();
last.item = "not";
```



将尾链接指向新结点

```
oldlast.next = last;
```

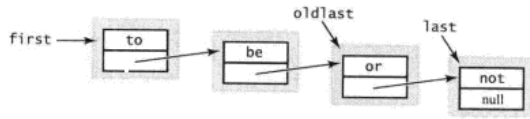


图 1.3.8 在链表的结尾插入一个新结点

1.3.3.7 遍历

要访问一个数组中的所有元素，我们会使用如下代码来循环处理 a[] 中的所有元素：

```
for (int i = 0; i < N; i++)
{
   // 处理 a[i]
}
```

访问链表中的所有元素也有一个对应的方式：将循环的索引变量 x 初始化为链表的首结点，然后通过 x.item 访问和 x 相关联的元素，并将 x 设为 x.next 来访问链表中的下一个结点，如此反复直到 x 为 null 为止（这说明我们已经到达了链表的结尾）。这个过程被称为链表的*遍历*，可以用以下循环处理链表的每个结点的代码简洁表达，其中 first 指向链表的首结点：

```
for (Node x = first; x != null; x = x.next)
{
   // 处理 x.item
}
```

这种方式和迭代遍历一个数组中的所有元素的标准方式一样自然。在我们的实现中，它是迭代
146 器使用的基本方式，它使用例能够迭代访问链表的所有元素而无需知道链表的实现细节。

1.3.3.8 栈的实现

有了这些预备知识，给出我们的 Stack API 的实现就很简单了，如 94 页的算法 1.2 所示。它将栈保存为一条链表，栈的顶部即为表头，实例变量 first 指向栈顶。这样，当使用 push() 压入一

个元素时，我们会按照 1.3.3.3 节所讨论的代码将该元素添加在表头；当使用 pop() 删除一个元素时，我们会按照 1.3.3.4 节讨论的代码将该元素从表头删除。要实现 size() 方法，我们用实例变量 N 保存元素的个数，在压入元素时将 N 加 1，在弹出元素时将 N 减 1。要实现 isEmpty() 方法，只需检查 first 是否为 null（或者可以检查 N 是否为 0）。该实现使用了泛型的 Item——你可以认为类名后的 <Item> 表示的是实现中所出现的所有 Item 都会替换为用例所提供的任意数据类型的名称（请见 1.3.2.2 节）。我们暂时省略了关于迭代的代码并将它们留到算法 1.4 中继续讨论。图 1.3.9 显示了我们所常用的测试用例的轨迹（测试用例代码放在了图后面）。链表的使用达到了我们的最优设计目标：

- 它可以处理任意类型的数据；
- 所需的空间总是和集合的大小成正比；
- 操作所需的时间总是和集合的大小无关。

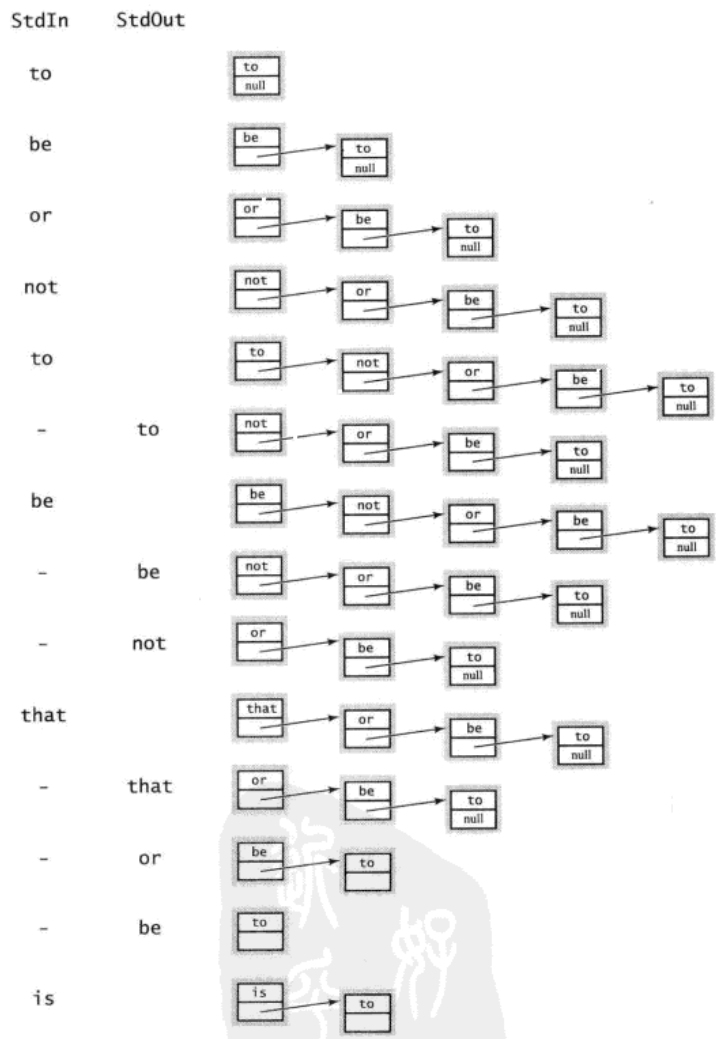


图 1.3.9 stack 的开发用例的轨迹

```
public static void main(String[] args)
{  // 创建一个栈并根据StdIn中的指示压入或弹出字符串
   Stack<String> s = new Stack<String>();
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (!item.equals("-"))
         s.push(item);
      else if (!s.isEmpty()) StdOut.print(s.pop() + " ");
   }
   StdOut.println("(" + s.size() + " left on stack)");
}
```

Stack的测试用例

这份实现是我们对许多算法的实现的原型。它定义了链表数据结构并实现了供用例使用的方法push()和pop()，仅用了少量代码就取得了所期望的效果。算法和数据结构是相辅相成的。在本例中，算法的实现代码很简单，但数据结构的性质却并不简单，我们用了好几页纸来说明这些性质。这种数据结构的定义和算法的实现的相互作用很常见，也是本书中我们对抽象数据类型的实现重点。

147 ~ 148

算法 1.2 下压堆栈（链表实现）

```
public class Stack<Item> implements Iterable<Item>
{
   private Node first; // 栈顶（最近添加的元素）
   private int N;      // 元素数量
   private class Node
   {  // 定义了结点的嵌套类
      Item item;
      Node next;
   }
   public boolean isEmpty() {  return first == null; }  // 或：N == 0
   public int size()        {  return N; }
   public void push(Item item)
   {  // 向栈顶添加元素
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
      N++;
   }
   public Item pop()
   {  // 从栈顶删除元素
      Item item = first.item;
      first = first.next;
      N--;
      return item;
   }
   // iterator()的实现请见算法1.4
   // 测试用例main()的实现请见本节前面部分
}
```

```
% more tobe.txt
to be or not to - be - - that - - - is

% java Stack < tobe.txt
to be not that or be (2 left on stack)
```

这份泛型的Stack实现的基础是链表数据结构。它可以用于创建任意数据类型的栈。要支持迭代，请添加算法1.4中为Bag数据类型给出的加粗部分的代码。

149

1.3.3.9 队列的实现

基于链表数据结构实现 Queue API 也很简单，如算法 1.3 所示。它将队列表示为一条从最早插入的元素到最近插入的元素的链表，实例变量 first 指向队列的开头，实例变量 last 指向队列的结尾。这样，要将一个元素入列（enqueue()），我们就将它添加到表尾（请见图 1.3.8 中讨论的代码，但是在链表为空时需要将 first 和 last 都指向新结点）；要将一个元素出列（dequeue()），我们就删除表头的结点（代码和 Stack 的 pop() 方法相同，只是当链表为空时需要更新 last 的值）。size() 和 isEmpty() 方法的实现和 Stack 相同。和 Stack 一样，Queue 的实现也使用了泛型参数 Item。这里我们省略了支持迭代的代码并将它们留到算法 1.4 中继续讨论。下面所示的是一个开发用例，它和我们在 Stack 中使用的用例很相似，它的轨迹如算法 1.3 所示。Queue 的实现使用的数据结构和 Stack 相同——链表，但它实现了不同的添加和删除元素的算法，这也是用例所看到的后进先出和先进后出的区别所在。和刚才一样，我们用链表达到了最优设计目标：它可以处理任意类型的数据，所需的空间总是和集合的大小成正比，操作所需的时间总是和集合的大小无关。[①]

```
public static void main(String[] args)
{  // 创建一个队列并操作字符串入列或出列

   Queue<String> q = new Queue<String>();

   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (!item.equals("-"))
           q.enqueue(item);
      else if (!q.isEmpty()) StdOut.print(q.dequeue() + " ");
   }

   StdOut.println("(" + q.size() + " left on queue)");
}
```

Queue 的测试用例

```
% more tobe.txt
to be or not to - be - - that - - - is

% java Queue < tobe.txt
to be or not to be (2 left on queue)
```

150

算法 1.3 先进先出队列

```
public class Queue<Item> implements Iterable<Item>
{
   private Node first; // 指向最早添加的结点的链接
   private Node last;  // 指向最近添加的结点的链接
   private int N;      // 队列中的元素数量
   private class Node
   {  // 定义了结点的嵌套类
```

① 这里原书应该是因为版面原因没有使用列表，如果版面允许可以使用和 Stack 部分相同的列表显示这三个目标。——译者注

```
      Item item;
      Node next;
   }
   public boolean isEmpty() {  return first == null;  }  // 或:  N == 0.
   public int size()        {  return N;  }
   public void enqueue(Item item)
   {  // 向表尾添加元素
      Node oldlast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else           oldlast.next = last;
      N++;
   }
   public Item dequeue()
   {  // 从表头删除元素
      Item item = first.item;
      first = first.next;
      if (isEmpty()) last = null;
      N--;
      return item;
   }
   // iterator() 的实现请见算法 1.4
   // 测试用例 main() 的实现请见前面
}
```

这份泛型的 Queue 实现的基础是链表数据结构。它可以用于创建任意数据类型的队列。要支持迭代，
151 请添加算法 1.4 中为 Bag 数据类型给出的加粗部分的代码。

Queue 的开发用例的轨迹如图 1.3.10 所示。

在结构化存储数据集时，**链表是数组的一种重要的替代方式**。这种替代方案已经有数十年的历史。事实上，编程语言历史上的一块里程碑就是 McCathy 在 20 世纪 50 年代发明的 LISP 语言，而链表则是这种语言组织程序和数据的主要结构。在练习中你会发现，链表编程也会遇到各种问题，且调试十分困难。在现代编程语言中，安全指针、自动垃圾回收（请见 1.2 节答疑部分）和抽象数据类型的使用使我们能够将链表处理的代码封装在若干个类中，正如本文所述。

1.3.3.10 背包的实现

用链表数据结构实现我们的 Bag API 只需要将 Stack 中的 push() 改名为 add()，并去掉 pop() 的实现即可，如算法 1.4 所示（也可以用相同的方法实现 Queue，但需要的代码更多）。在这份实现中，加粗部分的代码可以通过遍历链表使 Stack、Queue 和 Bag 变为可迭代的。对于 Stack，链表的访问顺序是后进先出；对于 Queue，链表的访问顺序是先进先出；对于 Bag，它正好也是后进先出的顺序，但顺序在这里并不重要。如算法 1.4 中加粗部分的代码所示，要在集合数据类型中实现迭代，第一步就是要添加下面这行代码，这样我们的代码才能引用 Java 的 Iterator 接口：

```
import java.util.Iterator;
```

第二步是在类的声明中添加这行代码，它保证了类必然会提供一个 iterator() 方法：

```
implements Iterable<Item>
```

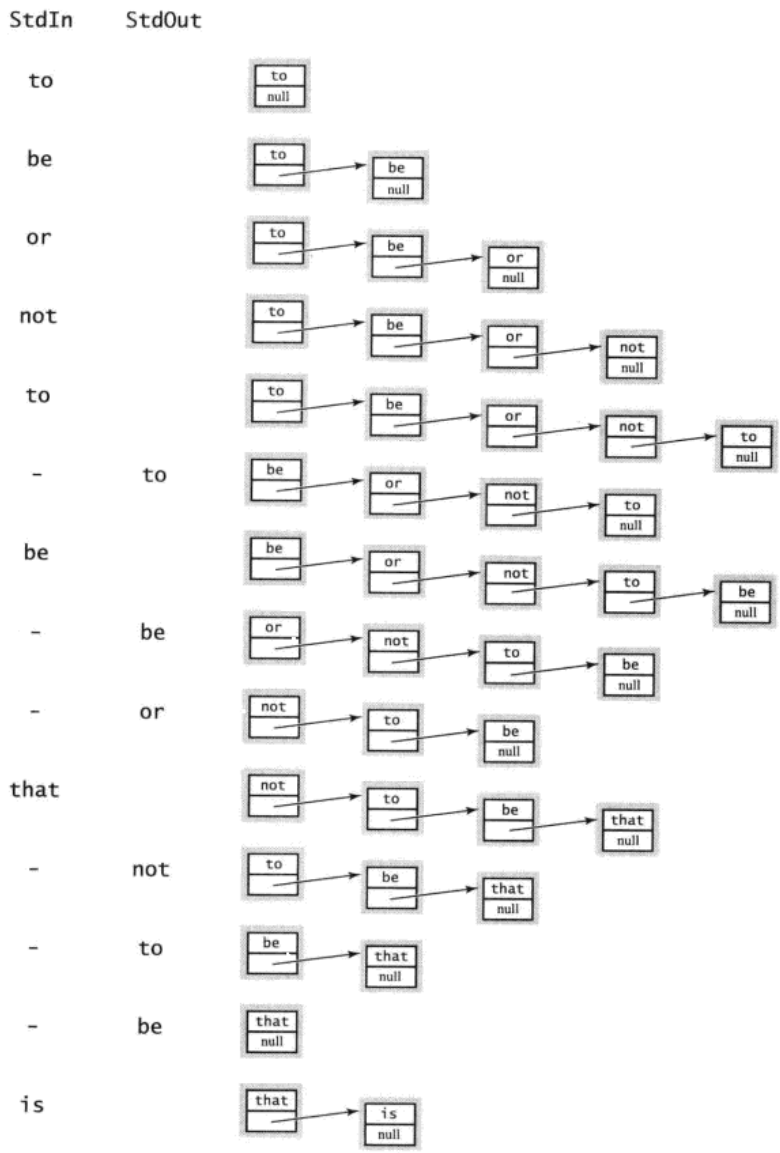


图 1.3.10 Queue 的开发用例的轨迹

iterator() 方法本身只是简单地从实现了 Iterator 接口的类中返回一个对象：

```
public Iterator<Item> iterator()
{  return new ListIterator(); }
```

这段代码保证了类必然会实现方法 hasNext()、next() 和 remove() 供用例的 foreach 语法使用。要实现这些方法，算法 1.4 中的嵌套类 ListIterator 维护了一个实例变量 current 来记录链表的当前结点。hasNext() 方法会检测 current 是否为 null，next() 方法会保存当前元素的引用，将 current 变量指向链表中的下个结点并返回所保存的引用。

152
~
154

算法 1.4 背包

```
import java.util.Iterator;
public class Bag<Item> implements Iterable<Item>
{
   private Node first;  //链表的首结点
   private class Node
   {
       Item item;
       Node next;
   }
   public void add(Item item)
   {  // 和 Stack 的 push() 方法完全相同
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }
   public Iterator<Item> iterator()
   {  return new ListIterator();  }
   private class ListIterator implements Iterator<Item>
   {
       private Node current = first;
       public boolean hasNext()
       {  return current != null;  }
       public void remove() { }
       public Item next()
       {
           Item item = current.item;
           current = current.next;
           return item;
       }
   }
}
```

这份 Bag 的实现维护了一条链表，用于保存所有通过 add() 添加的元素。size() 和 isEmpty() 方法的代码和 Stack 中的完全相同，因此在此处省略。迭代器会遍历链表并将当前结点保存在 current 变量中。我们可以将加粗的代码添加到算法 1.1 和算法 1.2 中使 Stack 和 Queue 变为可迭代的，因为它们
155 背后的数据结构是相同的，只是 Stack 和 Queue 的链表访问顺序分别是后进先出和先进先出而已。

1.3.4 综述

在本节中，我们所学习的支持泛型和迭代的背包、队列和栈的实现所提供的抽象使我们能够编写简洁的用例程序来操作对象的集合。深入理解这些抽象数据类型非常重要，这是我们研究算法和数据结构的开始。原因有三：第一，我们将以这些数据类型为基石构造本书中的其他更高级的数据结构；第二，它们展示了数据结构和算法的关系以及同时满足多个有可能相互冲突的性能目标时所要面对的挑战；第三，我们将要学习的若干算法的实现重点就是需要其中的抽象数据类型能够支持对对象集合的强大操作，这些实现正是我们的起点。

数据结构

我们现在拥有两种表示对象集合的方式，即数组和链表（如表 1.3.7 所示）。Java 内置了数组，链表也很容易使用 Java 的标准方法实现。两者都非常基础，常常被称为*顺序存储*和*链式存储*。在本书后面部分，我们会在各种抽象数据类型的实现中将多种方式结归并扩展这些基本的数据结构。其中一种重要的扩展就是各种含有多个链接的数据结构。例如，3.2 节和 3.3 节的重点就是被称为二叉

树的数据结构，它由含有两个链接的结点组成。另一个重要的扩展是复合型的数据结构：我们可以使用背包存储栈，用队列存储数组，等等。例如，第4章的主题是图，我们可以用数组的背包表示它。用这种方式很容易定义任意复杂的数据结构，而我们重点研究抽象数据类型的一个重要原因就是试图控制这种复杂度。

表 1.3.7 基础数据结构

数据结构	优 点	缺 点
数组	通过索引可以直接访问任意元素	在初始化时就需要知道元素的数量
链表	使用的空间大小和元素数量成正比	需要通过引用访问任意元素

156

我们在本节中研究**背包**、**队列**和**栈**时描述数据结构和算法的方式是全书的原型（本书中的数据结构示例见表 1.3.8）。在研究一个新的应用领域时，我们将会按照以下步骤识别目标并使用数据抽象解决问题：

- 定义 API；
- 根据特定的应用场景开发用例代码；
- 描述一种数据结构（一组值的表示），并在 API 所对应的抽象数据类型的实现中根据它定义类的*实例变量*；
- 描述算法（实现一组操作的方式），并根据它实现类中的*实例方法*；
- 分析算法的性能特点。

在下一节中，我们会详细研究最后一步，因为它常常能够决定哪种算法和实现才是解决现实应用问题的最佳选择。

表 1.3.8 本书所给出的数据结构举例

数据结构	章 节	抽象数据类型	数据表示
父链接树	1.5	UnionFind	整型数组
二分查找树	3.2、3.3	BST	含有两个链接的结点
字符串	5.1	String	数组、偏移量和长度
二叉堆	2.4	PQ	对象数组
散列表（拉链法）	3.4	SeparateChainingHashST	链表数组
散列表（线性探测法）	3.4	LinearProbingHashST	两个对象数组
图的邻接链表	4.1、4.2	Graph	Bag 对象的数组
单词查找树	5.2	TrieST	含有链接数组的结点
三向单词查找树	5.3	TST	含有三个链接的结点

157

答疑

问 并不是所有编程语言都支持泛型，甚至 Java 的早期版本也不支持。有其他替代方案吗？

答 如正文所述，一种替代方法是为每种类型的数据都实现一个不同的集合数据类型。另一种方法是构造一个 Object 对象的栈，并在用例中使用 pop() 时将得到的对象转换为所需的数据类型。这种方式的问题在于类型不匹配错误只能在运行时发现。而在泛型中，如果你的代码将错误类型的对象压入栈中，比如这样：

```
Stack<Apple> stack = new Stack<Apple>();
Apple  a = new Apple();
...
Orange b = new Orange();
...
stack.push(a);
...
stack.push(b);      // 编译时错误
```

会得到一个编译时错误：

```
push(Apple) in Stack<Apple> cannot be applied to (Orange)
```

能够在编译时发现错误足以说服我们使用泛型。

问 为什么 Java 不允许泛型数组？

答 专家们仍然在争论这一点。你可能也需要成为专家才能理解它！对于初学者，请先了解*共变数组*（covariant array）和*类型擦除*（type erasure）。

问 如何才能创建一个字符串栈的数组？

答 使用类型转换，比如：

```
Stack<String>[] a = (Stack<String>[]) new Stack[N];
```

警告：这段类型转换的用例代码和 1.3.2.2 节所示的有所不同。你可能会以为需要使用 `Object` 而非 `Stack`。在使用泛型时，Java 会在编译时检查类型的安全性，但会在运行时抛弃所有这些信息。因此在运行时语句右侧就变成了 `Stack<Object>[]` 或者只剩下了 `Stack[]`，因此我们必须将它们转化为 `Stack<String>[]`。

158 **问** 在栈为空时调用 `pop()` 会发生什么？

答 这取决于实现。对于我们在算法 1.2 中给出的实现，你会得到一个 `NullPointerException` 异常。对于我们在本书的网站上给出的实现，我们会抛出一个运行时异常以帮助用户定位错误。一般来说，在应用广泛的代码中这类检查越多越好。

问 既然有了链表，为什么还要学习如何调整数组的大小？

答 我们还将会学习若干抽象数据类型的示例实现，它们需要使用数组来实现一些链表难以实现的操作。`ResizingArrayStack` 是控制它们的内存使用的样板。

问 为什么将 `Node` 声明为嵌套类？为什么使用 `private`？

答 将 `Node` 声明为私有的嵌套类之后，我们可以将 `Node` 的方法和实例变量的访问范围限制在包含它的类中。私有嵌套类的一个特点是只有包含它的类能够直接访问它的实例变量，因此无需将它的实例变量声明为 `public` 或是 `private`。专业背景较强的读者注意：非静态的嵌套类也被称为*内部类*，因此从技术上来说我们的 `Node` 类也是内部类，尽管非泛型的类也可以是静态的。

问 当我输入 `javac Stack.java` 运行算法 1.2 和其他程序时，我发现了 Stack.class 和 Stack$Node.class 两个文件。第二个文件是做什么用的？

答 第二个文件是为内部类 `Node` 创建的。Java 的命名规则会使用 `$` 分隔外部类和内部类。

问 Java 标准库中有栈和队列吗？

答 有，也没有。Java 有一个内置的库，叫做 java.util.Stack，但你需要栈的时候请不要使用它。它新增了几个一般不属于栈的方法，例如获取第 i 个元素。它还允许从栈底添加元素（而非栈顶），所以它可以被当做队列使用！尽管拥有这些额外的操作看起来可能很有用，但它们其实是累赘。我们使
159 用某种数据类型不仅仅是为了获得我们能够想象的各种操作，也是为了准确地指定我们所需要的操

作。这么做的主要好处在于系统能够防止我们执行一些意外的操作。java.util.Stack 的 API 是宽接口的一个典型例子，我们通常会极力避免出现这种情况。

问 是否允许用例向栈或队列中添加空（null）元素？

答 在 Java 中实现集合类数据类型时这个问题是很常见的。我们的实现（以及 Java 的栈和队列库）允许插入 null 值。

问 如果用例在迭代中调用 push() 或者 pop()，Stack 的迭代器应该怎么办？

答 作为一个快速出错的迭代器，它应该立即抛出一个 java.util.ConcurrentModifi-cationException 异常。请见练习 1.3.50。

问 我们能够用 foreach 循环访问数组吗？

答 可以（尽管数组没有实现 Iterable 接口）。以下代码将会打印所有命令行参数：

```
public static void main(String[] args)
{  for (String s : args) StdOut.println(s);  }
```

问 我们能够用 foreach 循环访问字符串吗？

答 不行，String 没有实现 Iterable 接口。

问 为什么不实现一个单独的 Collection 数据类型并实现添加元素、删除最近插入的元素、删除最早插入的元素、删除随机元素、迭代、返回集合元素数量和其他我们可能需要的方法？这样我们就能在一个类中实现所有这些方法并可以应用于各种用例。

答 再次强调一遍，这又是一个宽接口的例子。Java 在它的 java.util.ArrayList 和 java.util.LinkedList 类的实现中犯过这样的错误。避免使用它们的一个原因是这样无法保证高效实现所有这些方法。在本书中，我们总是以 API 作为设计高效算法和数据结构的起点，而设计只含有几个操作的接口显然比设计含有许多操作的接口更简单。我们坚持窄接口的另一个原因是它们能够限制用例的行为，这将使用例代码更加易懂。如果一段用例代码使用 Stack<String>，而另一段用例代码使用 Queue<Transaction>，我们就可以知道后进先出的访问顺序对于前者很重要，而先进先出的访问顺序对于后者很重要。 160

练习

1.3.1 为 FixedCapacityStackOfStrings 添加一个方法 isFull()。

1.3.2 给定以下输入，java Stack 的输出是什么？

```
it was - the best - of times - - - it was - the - -
```

1.3.3 假设某个用例程序会进行一系列入栈和出栈的混合栈操作。入栈操作会将整数 0 到 9 按顺序压入栈；出栈操作会打印出返回值。下面哪种序列是不可能产生的？

a. 4 3 2 1 0 9 8 7 6 5
b. 4 6 8 7 5 3 2 9 0 1
c. 2 5 6 7 4 8 9 3 1 0
d. 4 3 2 1 0 5 6 7 8 9
e. 1 2 3 4 5 6 9 8 7 0
f. 0 4 6 5 3 8 1 7 2 9
g. 1 4 7 9 8 6 5 3 0 2
h. 2 1 4 3 6 5 8 7 9 0

1.3.4 编写一个 Stack 的用例 Parentheses，从标准输入中读取一个文本流并使用栈判定其中的括号是否配对完整。例如，对于 [()]{}{[()()]()} 程序应该打印 true，对于 [(]) 则打印 false。

1.3.5 当 N 为 50 时下面这段代码会打印什么？从较高的抽象层次描述给定正整数 N 时这段代码的行为。

```
Stack<Integer> stack = new Stack<Integer>();
while (N > 0)
{
   stack.push(N % 2);
   N = N / 2;
}
for (int d : stack) StdOut.print(d);
StdOut.println();
```

161

答：打印 N 的二进制表示（当 N 为 50 时打印 110010）。

1.3.6 下面这段代码对队列 q 进行了什么操作？

```
Stack<String> stack = new Stack<String>();
while (!q.isEmpty())
   stack.push(q.dequeue());
while (!stack.isEmpty())
   q.enqueue(stack.pop());
```

1.3.7 为 Stack 添加一个方法 peek()，返回栈中最近添加的元素（而不弹出它）。

1.3.8 给定以下输入，给出 DoublingStackOfStrings 的数组的内容和大小。

```
it was - the best - of times - - - it was - the - -
```

1.3.9 编写一段程序，从标准输入得到一个缺少左括号的表达式并打印出补全括号之后的中序表达式。例如，给定输入：

```
1 + 2 ) * 3 - 4 ) * 5 - 6 ) ) )
```

你的程序应该输出：

```
( ( 1 + 2 ) * ( ( 3 - 4 ) * ( 5 - 6 ) ) )
```

1.3.10 编写一个过滤器 InfixToPostfix，将算术表达式由中序表达式转为后序表达式。

1.3.11 编写一段程序 EvaluatePostfix，从标准输入中得到一个后序表达式，求值并打印结果（将上一题的程序中得到的输出用管道传递给这一段程序可以得到和 Evaluate 相同的行为）。

1.3.12 编写一个可迭代的 Stack 用例，它含有一个静态的 copy() 方法，接受一个字符串的栈作为参数并返回该栈的一个副本。注意：这种能力是迭代器价值的一个重要体现，因为有了它我们无需改变基本 API 就能够实现这种功能。

162 1.3.13 假设某个用例程序会进行一系列入列和出列的混合队列操作。入列操作会将整数 0 到 9 按顺序插入队列；出列操作会打印出返回值。下面哪种序列是不可能产生的？

a. 0 1 2 3 4 5 6 7 8 9

b. 4 6 8 7 5 3 2 9 0 1

c. 2 5 6 7 4 8 9 3 1 0

d. 4 3 2 1 0 5 6 7 8 9

1.3.14 编写一个类 ResizingArrayQueueOfStrings，使用定长数组实现队列的抽象，然后扩展实现，使用调整数组的方法突破大小的限制。

1.3.15 编写一个 Queue 的用例，接受一个命令行参数 k 并打印出标准输入中的倒数第 k 个字符串（假设标准输入中至少有 k 个字符串）。

1.3.16 使用 1.3.1.5 节中的 readInts() 作为模板为 Date 编写一个静态方法 readDates()，从标准输入中读取由练习 1.2.19 的表格所指定的格式的多个日期并返回一个它们的数组。

1.3.17 为 Transaction 类完成练习 1.3.16。

163

链表练习

这部分练习是专门针对链表的。建议：使用正文中所述的可视化表达方式画图。

1.3.18 假设 x 是一条链表的某个结点且不是尾结点。下面这条语句的效果是什么？

```
x.next = x.next.next;
```

答：删除 x 的后续结点。

1.3.19 给出一段代码，删除链表的尾结点，其中链表的首结点为 first。

1.3.20 编写一个方法 delete()，接受一个 int 参数 k，删除链表的第 k 个元素（如果它存在的话）。

1.3.21 编写一个方法 find()，接受一条链表和一个字符串 key 作为参数。如果链表中的某个结点的 item 域的值为 key，则方法返回 true，否则返回 false。

1.3.22 假设 x 是一条链表中的某个结点，下面这段代码做了什么？

```
t.next = x.next;
x.next = t;
```

答：插入结点 t 并使它成为 x 的后续结点。

1.3.23 为什么下面这段代码和上一道题中的代码效果不同？

```
x.next = t;
t.next = x.next;
```

答：在更新 t.next 时，x.next 已经不再指向 x 的后续结点，而是指向 t 本身！

1.3.24 编写一个方法 removeAfter()，接受一个链表结点作为参数并删除该结点的后续结点（如果参数结点或参数结点的后续结点为空则什么也不做）。

1.3.25 编写一个方法 insertAfter()，接受两个链表结点作为参数，将第二个结点插入链表并使之成为第一个结点的后续结点（如果两个参数为空则什么也不做）。

164

1.3.26 编写一个方法 remove()，接受一条链表和一个字符串 key 作为参数，删除链表中所有 item 域为 key 的结点。

1.3.27 编写一个方法 max()，接受一条链表的首结点作为参数，返回链表中键最大的节点的值。假设所有键均为正整数，如果链表为空则返回 0。

1.3.28 用递归的方法解答上一道练习。

1.3.29 用环形链表实现 Queue。环形链表也是一条链表，只是没有任何结点的链接为空，且只要链表非空则 last.next 的值为 first。只能使用一个 Node 类型的实例变量（last）。

1.3.30 编写一个函数，接受一条链表的首结点作为参数，（破坏性地）将链表反转并返回结果链表的首结点。

迭代方式的解答：为了完成这个任务，我们需要记录链表中三个连续的结点：reverse、first 和 second。在每轮迭代中，我们从原链表中提取结点 first 并将它插入到逆链表的开头。我们需要一直保持 first 指向原链表中所有剩余结点的首结点，second 指向原链表中所有剩余结点

的第二个结点，reverse 指向结果链表中的首结点。

```
public Node reverse(Node x)
{
   Node first   = x;
   Node reverse = null;
   while (first != null)
   {
      Node second = first.next;
      first.next  = reverse;
      reverse     = first;
      first       = second;
   }
   return reverse;
}
```

165 在编写和链表相关的代码时，我们必须小心处理异常情况（链表为空或是只有一个或两个结点）和边界情况（处理首尾结点）。它们通常比处理正常情况要困难得多。

递归解答：假设链表含有 N 个结点，我们先递归颠倒最后 N–1 个结点，然后小心地将原链表中的首结点插入到结果链表的末端。

```
public Node reverse(Node first)
{
   if (first == null) return null;
   if (first.next == null) return first;
   Node second = first.next;
   Node rest = reverse(second);
   second.next = first;
   first.next  = null;
   return rest;
}
```

1.3.31 实现一个嵌套类 DoubleNode 用来构造双向链表，其中每个结点都含有一个指向前驱元素的引用和一项指向后续元素的引用（如果不存在则为 null）。为以下任务实现若干静态方法：在表头插入结点、在表尾插入结点、从表头删除结点、从表尾删除结点、在指定结点之前插入新结点、
166 在指定结点之后插入新结点、删除指定结点。

提高题

1.3.32 Steque。一个以栈为目标的队列（或称 steque），是一种支持 push、pop 和 enqueue 操作的数据类型。为这种抽象数据类型定义一份 API 并给出一份基于链表的实现。[①]

1.3.33 Deque。一个双向队列（或者称为 deque）和栈或队列类似，但它同时支持在两端添加或删除元素。Deque 能够存储一组元素并支持表 1.3.9 中的 API：

表 1.3.9 泛型双向队列的 API

public class **Deque**<Item> implements Iterable<Item>		
	Deque()	创建空双向队列
boolean	isEmpty()	双向队列是否为空

① push、pop 都是对队列同一端的操作，enqueue 和 push 对应，但操作的是队列的另一端。——译者注

（续）

public class **Deque**<Item> implements Iterable<Item>		
int	size()	双向队列中的元素数量
void	pushLeft(Item item)	向左端添加一个新元素
void	pushRight(Item item)	向右端添加一个新元素
Item	popLeft()	从左端删除一个元素
Item	popRight()	从右端删除一个元素

编写一个使用双向链表实现这份 API 的 Deque 类，以及一个使用动态数组调整实现这份 API 的 ResizingArrayDeque 类。

1.3.34 随机背包。随机背包能够存储一组元素并支持表 1.3.10 中的 API：

表 1.3.10 泛型随机背包的 API

public class **RandomBag**<Item> implements Iterable<Item>		
	RandomBag()	创建一个空随机背包
boolean	isEmpty()	背包是否为空
int	size()	背包中的元素数量
void	add(Item item)	添加一个元素

编写一个 RandomBag 类来实现这份 API。请注意，除了形容词随机之外，这份 API 和 Bag 的 API 是相同的，这意味着迭代应该随机访问背包中的所有元素（对于每次迭代，所有的 $N!$ 种排列出现的可能性均相同）。提示：用数组保存所有元素并在迭代器的构造函数中随机打乱它们的顺序。 167

1.3.35 随机队列。随机队列能够存储一组元素并支持表 1.3.11 中的 API：

表 1.3.11 泛型随机队列的 API

public class **RandomQueue**<Item>		
	RandomQueue()	创建一条空的随机队列
boolean	isEmpty()	队列是否为空
void	enqueue(Item item)	添加一个元素
Item	dequeue()	删除并随机返回一个元素（取样且不放回）
Item	sample()	随机返回一个元素但不删除它（取样且放回）

编写一个 RandomQueue 类来实现这份 API。提示：使用（能够动态调整大小的）数组表示数据。删除一个元素时，随机交换某个元素（索引在 0 和 N-1 之间）和末位元素（索引为 N-1）的位置，然后像 ResizingArrayStack 一样删除并返回末位元素。编写一个用例，使用 RandomQueue<Card> 在桥牌中发牌（每人 13 张）。

1.3.36 随机迭代器。为上一题中的 RandomQueue<Item> 编写一个迭代器，随机返回队列中的所有元素。

1.3.37 Josephus 问题。在这个古老的问题中，N 个身陷绝境的人一致同意通过以下方式减少生存人数。他们围坐成一圈（位置记为 0 到 $N-1$）并从第一个人开始报数，报到 M 的人会被杀死，直到最后一个人留下来。传说中 Josephus 找到了不会被杀死的位置。编写一个 Queue 的用例 Josephus，从命令行接受 N 和 M 并打印出人们被杀死的顺序（这也将显示 Josephus 在圈中的位置）。

```
% java Josephus 7 2
1 3 5 0 4 2 6
```

168

1.3.38 删除第 k 个元素。实现一个类并支持表 1.3.12 中的 API：

表 1.3.12 泛型一般队列的 API

public class **GeneralizedQueue**<Item>		
	GeneralizedQueue()	创建一条空队列
boolean	isEmpty()	队列是否为空
void	insert(Item x)	添加一个元素
Item	delete(int k)	删除并返回最早插入的第 k 个元素

首先用数组实现该数据类型，然后用链表实现该数据类型。注意：我们在第 3 章中介绍的算法和数据结构可以保证 insert() 和 delete() 的实现所需的运行时间和和队列中的元素数量成对数关系——请见练习 3.5.27。

1.3.39 环形缓冲区。环形缓冲区，又称为环形队列，是一种定长为 N 的先进先出的数据结构。它在进程间的异步数据传输或记录日志文件时十分有用。当缓冲区为空时，消费者会在数据存入缓冲区前等待；当缓冲区满时，生产者会等待将数据存入缓冲区。为 RingBuffer 设计一份 API 并用（回环）数组将其实现。

1.3.40 前移编码。从标准输入读取一串字符，使用链表保存这些字符并清除重复字符。当你读取了一个从未见过的字符时，将它插入表头。当你读取了一个重复的字符时，将它从链表中删去并再次插入表头。将你的程序命名为 MoveToFront：它实现了著名的前移编码策略，这种策略假设最近访问过的元素很可能会再次访问，因此可以用于缓存、数据压缩等许多场景。

1.3.41 复制队列。编写一个新的构造函数，使以下代码

```
Queue<Item> r = new Queue<Item>(q);
```

169 得到的 r 指向队列 q 的一个新的独立的副本。可以对 q 或 r 进行任意入列或出列操作但它们不会相互影响。提示：从 q 中取出所有元素再将它们插入 q 和 r。

1.3.42 复制栈。为基于链表实现的栈编写一个新的构造函数，使以下代码

```
Stack<Item> t = new Stack<Item>(s);
```

得到的 t 指向栈 s 的一个新的独立的副本。

1.3.43 文件列表。文件夹就是一列文件和文件夹的列表。编写一个程序，从命令行接受一个文件夹名作为参数，打印出该文件夹下的所有文件并用递归的方式在所有子文件夹的名下（缩进）列出其下的所有文件。提示：使用队列，并参考 java.io.File。

1.3.44 文本编辑器的缓冲区。为文本编辑器的缓冲区设计一种数据类型并实现表 1.3.13 中的 API。

表 1.3.13 文本缓冲区的 API

Public class **Buffer**		
	Buffer()	创建一块空缓冲区
void	insert(char c)	在光标位置插入字符 c
char	delete()	删除并返回光标位置的字符
void	left(int k)	将光标向左移动 k 个位置
void	right(int k)	将光标向右移动 k 个位置
int	size()	缓冲区中的字符数量

提示：使用两个栈。

1.3.45 栈的可生成性。假设我们的栈测试用例将会进行一系列混合的入栈和出栈操作，序列中的整数0,1,…,N-1（按此先后顺序排列）表示入栈操作，N 个减号表示出栈操作。设计一个算法，判定给定的混合序列是否会使数组向下溢出（你所使用的空间量与 N 无关，即不能用某种数据结构存储所有整数）。设计一个线性时间的算法判定我们的测试用例能否产生某个给定的排列（这取决于出栈操作指令的出现位置）。 170

解答：除非对于某个整数 k，前 k 次出栈操作会在前 k 次入栈操作前完成，否则栈不会向下溢出。如果某个排列可以产生，那么它产生的方式一定是唯一的：如果输出排列中的下一个整数在栈顶，则将它弹出，否则将它压入栈之中。

1.3.46 栈可生成性问题中禁止出现的排列。若三元组 (a,b,c) 中 a<b<c 且 c 最先被弹出，a 第二，b 第三（c 和 a 以及 a 和 b 之间可以间隔其他整数），那么当且仅当排列中不含这样的三元组时（如上题所述的）栈才可能生成它。

部分解答：设有一个这样的三元组 (a,b,c)。c 会在 a 和 b 之前被弹出，但 a 和 b 会在 c 之前被压入。因此，当 c 被压入时，a 和 b 都已经在栈之中了。所以，a 不可能在 b 之前被弹出。

1.3.47 可连接的队列、栈或 steque。为队列、栈或 steque（请见练习 1.3.32）添加一个能够（破坏性地）连接两个同类对象的额外操作 catenation。

1.3.48 双向队列与栈。用一个双向队列实现两个栈，保证每个栈操作只需要常数次的双向队列操作（请见练习 1.3.33）。

1.3.49 栈与队列。用三个栈实现一个队列，保证每个队列操作（在最坏情况下）都只需要常数次的栈操作。警告：非常难！

1.3.50 快速出错的迭代器。修改 Stack 的迭代器代码，确保一旦用例在迭代器中（通过 push() 或 pop() 操作）修改集合数据就抛出一个 java.util.ConcurrentModificationException 异常。

解答：用一个计数器记录 push() 和 pop() 操作的次数。在创建迭代器时，将该值记录到 Iterator 的一个实例变量中。在每次调用 hasNext() 和 next() 之前，检查该值是否发生了变化，如果变化则抛出异常。 171

1.4 算法分析

随着使用计算机的经验的增长，人们在使用计算机解决困难问题或是处理大量数据时不可避免的将会产生这样的疑问：

我的程序会运行多长时间？

为什么我的程序耗尽了所有内存？

在重建某个音乐或照片库、安装某个新应用程序、编辑某个大型文档或是处理一大批实验数据时，你肯定也问过自己这些问题。这些问题太模糊了，我们无法准确回答——答案取决于许多因素，比如你所使用的计算机的性能、被处理的数据的性质和完成任务所使用的程序（实现了某种算法）。这些因素都会产生大量需要分析的信息。

尽管有这些困难，你在本节中将会看到，为这些基础问题给出实质性的答案有时其实非常简单。这个过程的基础是*科学方法*，它是科学家们为获取自然界知识所使用的一系列为大家所认同的方法。我们将会使用*数学分析*为算法成本建立简洁的模型并使用*实验数据*验证这些模型。

1.4.1 科学方法

科学家用来理解自然世界的方法对于研究计算机程序的运行时间同样有效：

- 细致地观察真实世界的特点，通常还要有精确的测量；
- 根据观察结果提出假设模型；
- 根据模型预测未来的事件；
- 继续观察并核实预测的准确性；
- 如此反复直到确认预测和观察一致。

科学方法的一条关键原则是我们所设计的实验必须是可重现的，这样他人也可以自己验证假设的真实性。所有的假设也必须是可证伪的，这样我们才能确认某个假设是错误的（并需要修正）。正如爱因斯坦的一句名言所说：“*再多的实验也不一定能够证明我是对的，但只需要一个实验就能证明我是错的。*”我们永远也没法知道某个假设是否绝对正确，我们只能验证它和我们的观察的一致性。

172

1.4.2 观察

我们的第一个挑战是决定如何定量测量程序的运行时间。在这里这个任务比自然科学中的要简单得多。我们不需要向火星发射火箭或者牺牲一些实验室的小动物或是分裂某个原子——只需要运行程序即可。事实上，每次运行程序都是在进行一次科学实验，将这个程序和自然世界联系起来并回答我们的一个核心问题：我的程序会运行多长时间？

我们对大多数程序的第一个定量观察就是计算性任务的困难程度可以用*问题的规模*来衡量。一般来说，问题的规模可以是输入的大小或是某个命令行参数的值。根据直觉，程序的运行时间应该随着问题规模的增长而变长，但我们每次在开发和运行一个程序时想问的问题都是运行时间的增长有多快。

从许多程序中得到的另一个定量观察是运行时间和输入本身相对无关，它主要取决于问题规模。如果这个关系不成立，我们就需要进行一些实验来更好地理解并更好地控制运行时间对输入的敏感度。但这个关系常常是成立的，因此我们现在来重点研究如何更好地将问题规模和运行时间的关系量化。

1.4.2.1 举例

右侧的 ThreeSum 程序是一个可运行的示例。它会统计一个文件中所有和为 0 的三整数元组的数量（假设整数不会溢出）。这种计算可能看起来有些不自然，但其实它和许多基础计算性任务都有着深刻的联系（例如，请见练习 1.4.26）。作为测试输入，我们使用的是本书网站上的 1Mints.txt 文件。它含有 100 万个随机生成的 int 值。1Mints.txt 中的第二个、第八个和第十个元组的和均为 0。文件中还有多少组这样的数据？ ThreeSum 能够告诉我们答案，但它所需的时间可以接受吗？问题的规模 N 和 ThreeSum 的运行时间有什么关系？我们的第一个实验就是在计算机上运行 ThreeSum 并处理本书网站上的 1Kints.txt、2Kints.txt、4Kints.txt 和 8Kints.txt 文件，它们分别含有 1Mints.txt 中的 1000、2000、4000 和 8000 个整数。你可以很快得到这样的整数元组在 1Kints.txt 中共有 70 组，在 2Kints.txt 中共有 528 组，如图 1.4.1 所示。这个程序需要用比之前长得多的时间得到在 4Kints.txt 中共有 4039 组和为 0 的整数。在等待它处理 8Kints.txt 的时候，你会发现你在问自己：“我的程序还要运行多久？”你会看到，对于这个程序，回答这个问题很简单。实际上，你常常能在程序运行的时候就给出一个较为准确的预测。

1.4.2.2 计时器

准确测量给定程序的确切运行时间是很困难的。不过幸运的是我们一般只需要近似值就可以了。我们希望能够把需要几秒钟或者几分钟就能完成的程序和需要几天、几个月甚至更长时间才能完成的程序区别开来，而且我们希望知道对

```
public class ThreeSum
{
   public static int count(int[] a)
   {  // 统计和为0的元组的数量
      int N = a.length;
      int cnt = 0;
      for (int i = 0; i < N; i++)
         for (int j = i+1; j < N; j++)
            for (int k = j+1; k < N; k++)
               if (a[i] + a[j] + a[k] == 0)
                  cnt++;
      return cnt;
   }

   public static void main(String[] args)
   {
      int[] a = In.readInts(args[0]);
      StdOut.println(count(a));
   }
}
```

对于给定的 N，这段程序需要运行多长时间 173

```
% more 1Mints.txt
 324110
-442472
 626686
-157678
 508681
 123414
 -77867
 155091
 129801
 287381
 604242
 686904
-247109
  77867
 982455
-210707
-922943
-738817
  85168
 855430
 ...
```

```
% java ThreeSum 1Kints.txt
```




```
70
% java ThreeSum 2Kints.txt
```




```
528
% java ThreeSum 4Kints.txt
```


滴答滴答滴答滴答滴答滴答滴答滴答滴答滴答

于同一个任务某个程序是不是比另一个程序快一倍。因此，我们仍然需要准确的测量手段来生成实验数据，并根据它们得出并验证关于程序的运行时间和问题规模的假设。为此，我们使用了如表1.4.1所示的Stopwatch数据类型。它的elapsedTime()方法能够返回自它创建以来所经过的时间，以秒为单位。它的实现基于Java系统的currentTimeMillis()方法，该方法能够返回以毫秒记数的当前时间。它在构造函数中保存了当前时间，并在elapsedTime()方法被调用时再次调用该方法
174 来计算得到对象创建以来经过的时间。

表1.4.1 一种表示计时器的抽象数据类型

API	public class **Stopwatch**	
	Stopwatch()	创建一个计时器
double	elapseTime()	返回对象创建以来所经过的时间

典型用例

```
public static void main(String[] args)
{
   int N = Integer.parseInt(args[0]);
   int[] a = new int[N];
   for (int i = 0; i < N; i++)
      a[i] = StdRandom.uniform(-1000000, 1000000);
   Stopwatch timer = new Stopwatch();
   int cnt = ThreeSum.count(a);
   double time = timer.elapsedTime();
   StdOut.println(cnt + "triples" + time + "seconds");
}
```

使用方法

```
% java Stopwatch 1000
51 triples 0.488 seconds
% java Stopwatch 2000
516 triples 3.855 seconds
```

数据类型的实现

```
public class Stopwatch
{
   private final long start;
   public Stopwatch()
   {  start = System.currentTimeMillis();  }
   public double elapsedTime()
   {
      long now = System.currentTimeMillis();
      return (now - start) / 1000.0;
   }
}
```

175

1.4.2.3 实验数据的分析

DoublingTest是Stopwatch的一个更加复杂的用例，并能够为ThreeSum产生实验数据。它会生成一系列随机输入数组，在每一步中将数组长度加倍，并打印出ThreeSum.count()处理每种输入规模所需的运行时间。这些实验显然是可重现的——你也可以在自己的计算机上运行它们，多少次都行。在运行DoublingTest时，你会发现自己进入了一个“预测—验证”的循环：它会快速打印出几行数据，

但随即慢了下来。每当它打印出一行结果时，你都会开始琢磨它还需要多久才能打出下一行。当然，因为大家使用的计算机不同，你得到的实际运行时间很可能和我们的计算机得到的不一样。事实上，如果你的计算机比我们的快一倍，你所得到的运行时间应该大致是我们所得到的一半。由此我们马上可以得出一条有说服力的猜想：程序在不同的计算机上的运行时间之比通常是一个常数。尽管如此，你还是会提出更详细的问题：作为问题规模的一个函数，我的程序的运行时间是多久？为了帮助你回答这个问题，我们来将数据绘制成图表。图 1.4.2 就是产生结果，使用的分别是标准比例尺和对数比例尺。其中 x 轴表示 N，y 轴表示程序的运行时间 $T(N)$。由对数的图像我们立即可以得到一个关于运行时间的猜想——因为数据和斜率为 3 的直线完全吻合。该直线的公式为（其中 a 为常数）：

$$\lg(T(N)) = 3\lg N + \lg a$$

它等价于：

$$T(N) = aN^3$$

这就是我们想要的运行时间关于输入规模 N 的函数。我们可以用其中一个数据点来解出 a 的值——例如，$T(8000) = a8000^3$，可得 $a = 9.98 \times 10^{-11}$——因此我们就可以用以下公式预测 N 值较大时程序的运行时间：

$$T(N) = 9.98 \times 10^{-11} N^3$$

我们可以根据对数图像中的数据点距离这条直线的远近来不严格地检验这条假设。一些统计学方法可以帮助我们更加仔细地分析出 a 和指数 b 的近似值，但我们的快速计算已经足以在大多数情况下估计出程序的运行时间。例如，我们预计，在我们的计算机上，当 N=16000 时程序的运行时间约为 $9.98 \times 10^{-11} \times 16000^3$=408.8 秒，也就是约 6.8 分钟（实际时间为 409.3 秒）。在等待计算机得出 DoublingTest 在 N=16000 的实验数据时，也可以用这个方法来预测它何时将会结束，然后等待并验证你的结果是否正确。

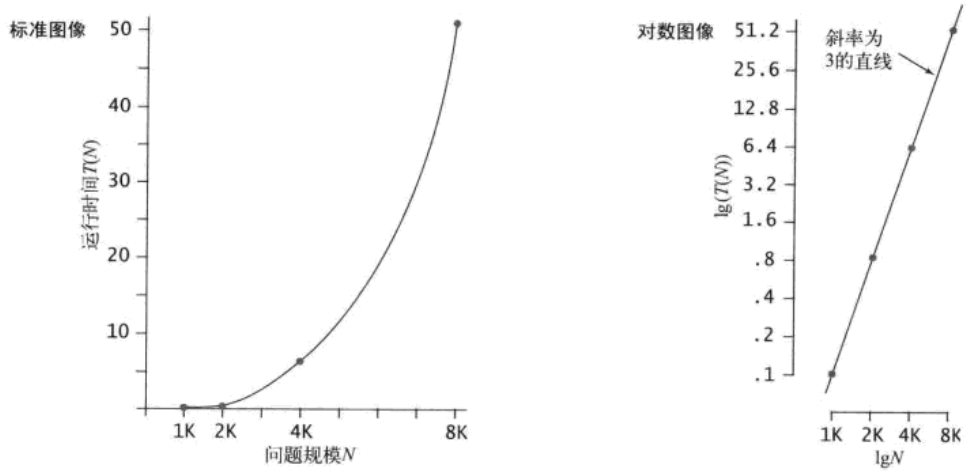
176

实验程序

```
public class DoublingTest
{
   public static double timeTrial(int N)
   {  // 为处理 N 个随机的六位整数的 ThreeSum.count() 计时
      int MAX = 1000000;
      int[] a = new int[N];
      for (int i = 0; i < N; i++)
         a[i] = StdRandom.uniform(-MAX, MAX);
      Stopwatch timer = new Stopwatch();
      int cnt = ThreeSum.count(a);
      return timer.elapsedTime();
   }
   public static void main(String[] args)
   {  // 打印运行时间的表格
      for (int N = 250; true; N += N)
      {  // 打印问题规模为 N 时程序的用时
         double time = timeTrial(N);
         StdOut.printf("%7d %5.1f\n", N, time);
      }
   }
}
```

实验结果

```
% java DoublingTest
    250   0.0
    500   0.0
   1000   0.1
   2000   0.8
   4000   6.4
   8000  51.1
...
```



177

图 1.4.2 实验数据（ThreeSum.count() 的运行时间）的分析

到现在为止，这个过程和科学家们在尝试理解真实世界的奥秘时进行的过程完全相同。对数图像中的直线等价于我们对数据符合公式 $T(N)=aN^b$ 的猜想。这种公式被称为*幂次法则*。许多自然和人工的现象都符合幂次法则，因此假设程序的运行时间符合幂次法则也是合情合理的。事实上，对于算法的分析，我们有许多数学模型强烈支持这种函数和其他类似的假设，我们现在就来学习它们。

1.4.3 数学模型

在计算机科学的早期，D. E. Knuth 认为，尽管有许多复杂的因素影响着我们对程序的运行时间的理解，原则上我们仍然可能构造出一个数学模型来描述任意程序的运行时间。Knuth 的基本见地很简单——一个程序运行的总时间主要和两点有关：

- 执行每条语句的耗时；
- 执行每条语句的频率。

前者取决于计算机、Java 编译器和操作系统，后者取决于程序本身和输入。如果对于程序的所有部分我们都知道了这些性质，可以将它们相乘并将程序中所有指令的成本相加得到总运行时间。

第一个挑战是判定语句的执行频率。有些语句的分析很容易：例如，ThreeSum.count() 中将 cnt 的值设为 0 的语句只会执行一次。有些则需要深入分析：例如，ThreeSum.count() 中的 if 语句会执行 $N(N-1)(N-2)/6$ 次（从输入数组中能够取得的三个不同整数的数量——请见练习 1.4.1）。其他则取决于输入数据，例如，ThreeSum.count() 中的指令 cnt++ 执行的次数为输入中和为 0 的整数三元组的数量，这可能是 0 也可能是任意值。对于 DoublingTest 的情况，输入值是随机产生的，我们可以用概率分析得到该值的期望（请见练习 1.4.40）。

1.4.3.1 近似

这种频率分析可能会产生复杂冗长的数学表达式。例如，刚才我们所讨论的 ThreeSum 中的 if 语句的执行次数为：

178

$$N(N-1)(N-2)/6=N^3/6-N^2/2+N/3$$

一般在这种表达式中，首项之后的其他项都相对较小（例如，当 $N=1000$ 时，$-N^2/2+N/3 \approx 499\,667$，相对于 $N^3/6 \approx 166\,666\,667$ 就小得多了），如图 1.4.3 所示。我们常常使用约等于号（~）来忽略较小的项，从而大大简化我们所处理的数学公式。该符号使我们能够用近似的方式忽略公式中那

些非常复杂但幂次较低，且对最终结果的贡献无关紧要的项：

定义。我们用 ~$f(N)$ 表示所有随着 N 的增大除以 $f(N)$ 的结果趋近于 1 的函数。我们用 $g(N) \sim f(N)$ 表示 $g(N)/f(N)$ 随着 N 的增大趋近于 1。

例如，我们用 ~$N^3/6$ 表示 ThreeSum 中的 `if` 语句的执行次数，因为 $N^3/6-N^2/2+N/3$ 除以 $N^3/6$ 的结果随着 N 的增大趋向于 1。一般我们用到的近似方式都是 $g(N) \sim af(N)$，其中 $f(N)=N^b(\log N)^c$，其中 a、b 和 c 均为常数。我们将 $f(N)$ 称为 $g(N)$ 的增长的数量级（如表 1.4.2 所示）。我们一般不会指定底数，因为常数 a 能够弥补这些细节。这种形式的函数覆盖了我们在对程序运行时间的研究中经常遇到的几种函数，如表 1.4.3 所示（指数级别是一个例外，我们会在第 6 章中讲到）。我们会详细说明这几种函数并在处理完 ThreeSum 之后简要讨论为什么它们会出现在算法分析领域之中。

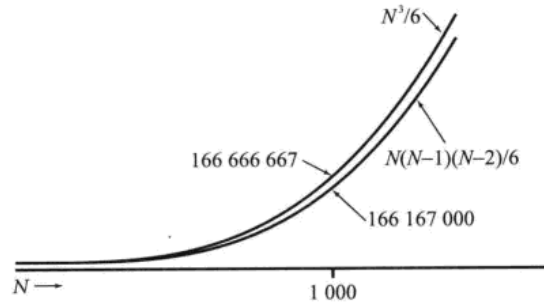


图 1.4.3 首项近似

表 1.4.2 典型的近似

函数	近似	增长的数量级
$N^3/6-N^2/2+N/3$	~$N^3/6$	N^3
$N^2/2-N/2$	~$N^2/2$	N^2
$\lg N+1$	~$\lg N$	$\lg N$
3	~3	1

表 1.4.3 常见的增长数量级函数

增长的数量级	
描述	函数
常数级别	1
对数级别	$\log N$
线性级别	N
线性对数级别	$N\log N$
平方级别	N^2
立方级别	N^3
指数级别	2^N

1.4.3.2 近似运行时间

按照 Knuth 的方法，要得到一个 Java 程序的总运行时间的数学表达式，（原则上）我们需要研究我们的 Java 编译器来找出每条 Java 指令所对应的机器指令数，并根据我们的计算机的指令规范得到每条机器指令的运行时间，然后才能得到一个总运行时间。对于 ThreeSum，这个时间的大致总结如图 1.4.4 所示。我们根据执行的频率将 Java 的语句分块，计算出每种频率的首项近似，判定每条指令的执行成本并计算出总和。请注意，某些执行频率可能会依赖于输入。在本例中，`cnt++` 的执行次数显然就是依赖于输入的——它就是和为 0 的整数三元组的数量，范围在 0 到 ~$N^3/6$ 之间。通过用常数 t_0、t_1、t_2…表示各个代码块的执行时间，我们假设每个 Java 代码块所对应的机器指令集所需的执行时间都是固定的。除此之外，我们基本不会涉及任何特定系统的细节（这些常数的值）。从这里我们观察到的一个关键现象是执行最频繁的指令决定了程序执行的总时间——我们将这些指令称为程序的*内循环*。对于 ThreeSum 来说，它的内循环是将 `k` 加 1、判断它是否小于 `N` 以及判断给定的三个整数之和是否为 0 的语句（也许还包括记数的语句，不过这取决于输入）。这种情况是很典型的：许多程序的运行时间都只取决于其中的一小部分指令。

179

1.4.3.3 对增长数量级的猜想

总之，1.4.2.3 节中的实验和表 1.4.4 中的数学模型都支持以下猜想：

性质 A。 ThreeSum（在 N 个数中找出三个和为 0 的整数元组的数量）的运行时间的增长数量级为 N^3。

例证。 设 $T(N)$ 为 ThreeSum 处理 N 个整数的运行时间。根据前文所述的数学模型有 $T(N) \sim aN^3$，其中常数 a 取决于计算机的具体型号。在许多计算机上完成的实验（包括你我的计算机）都验证了这个近似。

在本书中，我们使用性质表示需要用实验验证的猜想。数学分析的最终结果和我们的实验分析的最终结果完全相同——ThreeSum 的运行时间是 $\sim aN^3$，其中常数 a 取决于计算机的具体型号。这
180 次吻合既验证了实验结果和数学模型，也揭示了该程序的更多性质，因为我们不需要实验就能确定 N 的指数。稍加努力，我们就能确定某个特定系统上的 a 的值，不过这一般都只在有性能压力的情形下才需要由专家来完成。

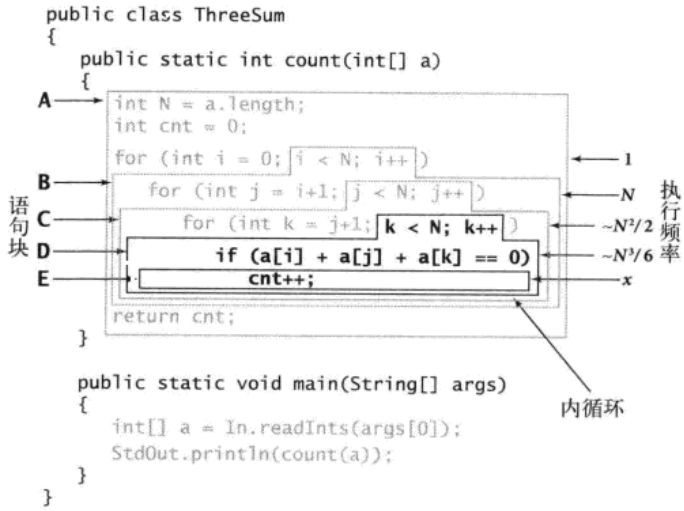


图 1.4.4 程序语句执行频率的分析

表 1.4.4 程序运行时间的分析（示例）

语句块	运行时间（以秒记）	频率	总时间
E	t_0	x（取决于输入）	$t_0 x$
D	t_1	$N^3/6 - N^2/2 + N/3$	$t_1 (N^3/6 - N^2/2 + N/3)$
C	t_2	$N^2/2 - N/2$	$t_2 (N^2/2 - N/2)$
B	t_3	N	$t_3 N$
A	t_4	1	t_4
		总时间	$(t_1/6)\,N^3 + (t_2/2 - t_1/2)\,N^2 + (t_1/3 - t_2/2 + t_3)\,N + t_4 + t_0 x$
		近似	$\sim (t_1/6)\,N^3$（假设 x 很小）
		增长的数量级	N^3

181

1.4.3.4 算法的分析

类似于性质 A 的猜想的意义很重要，因为它们将抽象世界中的一个 Java 程序和真实世界中运行它的一台计算机联系了起来。增长数量级概念的应用使我们能够继续向前迈进一步：将程序和它实现的算法隔离开来。ThreeSum 的运行时间的增长数量级是 N^3，这与它是由 Java 实现或是它运行在你的笔记本电脑上或是某人的手机上或是一台超级计算机上无关。决定这一点的主要因素是它需要检查输入中任意三个整数的所有可能组合。你所使用的算法（有时还要算上输入模型）决定了增长的数量级。将算法和某台计算机上的具体实现分离开来是一个强大的概念，因为这使我们对算法性能的知识可以应用于任何计算机。例如，我们可以说 ThreeSum 是暴力算法“计算所有不同的整数三元组的和，统计和为 0 的组数”的一种实现，可以预料的是在任何计算机上使用任何语言对该算法的实现所需的运行时间都是和 N^3 成正比的。实际上，经典算法的性能理论大部分都发表于数十年前，但它们仍然适用于今天的计算机。

1.4.3.5 成本模型

我们使用了一个成本模型来评估算法的性质。这个模型定义了我们所研究的算法中的基本操作。例如，适合于右侧所示的 3-sum 问题的成本模型是我们访问数组元素的次数。

3-sum 的成本模型。在研究解决 3-sum 问题的算法时，我们记录的是数组的访问次数（访问数组元素的次数，无论读写）。

在这个成本模型之下，我们可以用精确的数学语言说明算法而非某个特定实现的性质，如下：

命题 B。3-sum 的暴力算法使用了 ~$N^3/2$ 次数组访问来计算 N 个整数中和为 0 的整数三元组的数量。

证明。该算法访问了 ~$N^3/6$ 个整数三元组中的所有 3 个整数。

我们使用术语命题来表示在某个成本模型下算法的数学性质。在全书中我们都会使用某个确定的成本模型研究所讨论的算法。我们希望通过明确成本模型使给定实现所需的运行时间的增长数量级和它背后的算法的成本的增长数量级相同（换句话说，成本模型应该和内循环中的操作相关）。我们会研究算法准确的数学性质（命题）并对实现的性能作出猜想（性质），可以通过实验验证这些猜想。在本例中，命题 B 的数学结论支持了性质 A 中由科学方法得到并由实验验证过的猜想。

182 ~ 183

1.4.3.6 总结

对于大多数程序，得到其运行时间的数学模型所需的步骤如下：

- 确定输入模型，定义问题的规模；
- 识别内循环；
- 根据内循环中的操作确定成本模型；
- 对于给定的输入，判断这些操作的执行频率。这可能需要进行数学分析——我们在本书中会在学习具体的算法时给出一些例子。

如果一个程序含有多个方法，我们一般会分别讨论它们，例如我们在 1.1 节中见过的示例程序 BinarySearch。

二分查找。它的输入模型是大小为 N 的数组 `a[]`，内循环是一个 `while` 循环中的所有语句，

成本模型是比较操作（比较两个数组元素的值）。3.1 节中详细完整地给出了 1.1 节中所讨论过的命题 B 的证明，该命题说明它所需的比较次数最多为 $\lg N+1$。

白名单。它的输入模型是白名单的大小 N 和由标准输入得到的 M 个整数，且我们假设 $M>>N$，内循环是一个 `while` 循环中的所有语句，成本模型是比较操作（承自二分查找）。由二分查找的分析我们可以立即得到对白名单问题的分析——比较次数最多为 $M(\lg N+1)$。

根据以下因素我们可以知道，白名单问题计算所需时间的增长数量级最多为 $M\lg N$：

- 如果 N 很小，输入—输出可能会成为主要成本。
- 比较的次数取决于输入——在 ~ M 和 ~ $M\lg N$ 之间，取决于标准输入中有多少个整数在白名单中以及二分查找需要多久才能找出它们（一般来说为 ~ $M\lg N$）。
- 我们假设 `Arrays.sort()` 的成本远小于 $M\lg N$。`Arrays.sort()` 使用的是 2.2 节中的*归并排序*算法。我们会看到归并排序的运行时间的增长数量级为 $N\log N$（请见第 2 章的命题 G），因此这个假设是合理的。

因此，该模型支持了我们在 1.1 节中作出的假设，即当 M 和 N 很大时二分查找算法也能够完成计算。如果我们将标准输入流的长度加倍，可以预计的是运行时间也将加倍；如果我们将白名单的大小加倍，可以预计的是运行时间只会稍有增加。

在算法分析中**进行数学建模**是一个多产的研究领域，但它多少超出了本书的范畴。通过二分查
184 找、归并排序和其他许多算法你仍会看到，理解特定的数学模型对于理解基础算法的运行效率是很关键的，因此我们常常会详细地证明它们或是引用经典研究中的结论。在其中，我们会遇到各种数学分析中广泛使用的函数和近似函数。作为参考，我们分别在表 1.4.5 和表 1.4.6 中对它们的部分信息进行了总结。

表 1.4.5 算法分析中的常见函数

描 述	记 号	定 义
向下取整（floor）	$\lfloor x \rfloor$	不大于 x 的最大整数
向上取整（ceiling）	$\lfloor x \rfloor$	不小于 x 的最小整数
自然对数	$\ln N$	$\log_e N(e^x=N)$
以 2 为底的对数	$\lg N$	$\log_2 N(2^x=N)$
以 2 为底的整型对数	$\lfloor \lg N \rfloor$	不大于 $\lg N$ 的最大整数 （N 的二进制表示的位数）− 1
调和级数	H_N	$1+1/2+1/3+1/4+\cdots+1/N$
阶乘	$N!$	$1\times 2\times 3\times 4\times \cdots \times N$

表 1.4.6 算法分析中常用的近似函数

描 述	近似函数
调和级数求和	$H_N=1+1/2+1/3+1/4+\cdots+1/N \sim \ln N$
等差数列求和	$1+2+3+4+\cdots+N \sim N^2/2$
等比数列求和	$1+2+4+8+\cdots+N=2N-1 \sim 2N$，其中 $N=2^n$
斯特灵公式	$\lg N!=\lg 1+\lg 2+\lg 3+\lg 4+\cdots+\lg N \sim N\lg N$
二项式系数	$\binom{N}{k} \sim N^k/k!$，其中 k 为小常数
185 指数函数	$(1-1/x)^x \sim 1/e$

1.4.4 增长数量级的分类

我们在实现算法时使用了几种结构性的原语（普通语句、条件语句、循环、嵌套语句和方法调用），所以成本增长的数量级一般都是问题规模 N 的若干函数之一。表 1.4.7 总结了这些函数以及它们的称谓、与之对应的典型代码以及一些例子。

表 1.4.7 对增长数量级的常见假设的总结

描述	增长的数量级	典型的代码	说明	举例
常数级别	1	`a = b + c;`	普通语句	将两个数相加
对数级别	$\log N$	（请见 1.1.10.2 节，二分查找）	二分策略	二分查找
线性级别	N	`double max = a[0];` `for (int i = 1; i < N; i++)` `  if (a[i] > max) max = a[i];`	循环	找出最大元素
线性对数级别	$N \log N$	[请见算法 2.4]	分治	归并排序
平方级别	N^2	`for (int i = 0; i < N; i++)` `  for (int j = i+1; j < N; j++)` `  if (a[i] + a[j] == 0)` `  cnt++;`	双层循环	检查所有元素对
立方级别	N^3	`for (int i = 0; i < N; i++)` `  for (int j = i+1; j < N; j++)` `  for (int k = j+1; k < N; k++)` `  if (a[i] + a[j] + a[k] == 0)` `  cnt++;`	三层循环	检查所有三元组
指数级别	2^N	（请见第 6 章）	穷举查找	检查所有子集

1.4.4.1 常数级别

运行时间的增长数量级为常数的程序完成它的任务所需的操作次数一定，因此它的运行时间不依赖于 N。大多数的 Java 操作所需的时间均为常数。

1.4.4.2 对数级别

运行时间的增长数量级为对数的程序仅比常数时间的程序稍慢。运行时间和问题规模成对数关系的程序的经典例子就是二分查找（请见 1.1.10.2 节的 BinarySearch）。对数的底数和增长的数量级无关（因为不同的底数仅相当于一个常数因子），所以我们在说明对数级别时一般使用 log$N$。

1.4.4.3 线性级别

使用常数时间处理输入数据中的所有元素或是基于单个 `for` 循环的程序是十分常见的。此类程序的增长数量级是线性的——它的运行时间和 N 成正比。

1.4.4.4 线性对数级别

我们用线性对数描述运行时间和问题规模 N 的关系为 Nlog$N$ 的程序。和之前一样，对数的底数和增长的数量级无关。线性对数算法的典型例子是 `Merge.sort`（请见算法 2.4）和 `Quick.sort()`（请见算法 2.5）。

1.4.4.5 平方级别

一个运行时间的增长数量级为 N^2 的程序一般都含有两个嵌套的 `for` 循环，对由 N 个元素得到的所有元素对进行计算。初级排序算法 `Selection.sort()`（请见算法 2.1）和 `Insertion.sort()`（请见算法 2.2）都是这种类型的典型程序。

1.4.4.6 立方级别

一个运行时间的增长数量级为 N^3 的程序一般都含有三个嵌套的 for 循环，对由 N 个元素得到的所有三元组进行计算。本节中的 ThreeSum 就是一个典型的例子。

1.4.4.7 指数级别

在第 6 章中（也只会在第 6 章）我们将会遇到运行时间和 2^N 或者更高级别的函数成正比的程序。一般我们会使用*指数级别*来描述增长数量级为 b^N 的算法，其中 b>1 且为常数，尽管不同的 b 值得到的运行时间可能完全不同。指数级别的算法非常慢——不可能用它们解决大规模的问题。但指数级别的算法仍然在算法理论中有着重要的地位，因为它们看起来仍然是解决许多问题的最佳方案。

186 ~ 187

以上是最常见分类，但肯定不是最全面的。算法的增长数量级可能是 $N^2\log N$ 或者 $N^{3/2}$ 或者是其他类似的函数。实际上，详细的算法分析可能会用到若干个世纪以来发明的各种数学工具。

我们所学习的一大部分算法的性能特点都很简单，可以使用我们所讨论过的某种增长数量级函数精确地描述。因此，我们可以在某个成本模型下提出十分准确的命题。例如，*归并排序所需的比较次数在 $1/2N\lg N$ 到 $N\lg N$ 之间，由此我们立即可知归并排序所需的运行时间的增长数量级是线性对数的。*简单起见，我们将这句话简写为*归并排序是线性对数的。*

图 1.4.5 显示了增长数量级函数在实际应用中的重要性。其中 x 轴为问题规模，y 轴为运行时间。这些图表清晰的说明了平方级别和立方级别的算法对于大规模的问题是不可用的。许多重要的问题的直观解法是平方级别的，但我们也发现了它们的线性对数级别的算法。此类算法（包括归并排序）在实践中非常重要，因为它们能够解决的问题规模远大于平方级别的解法能够处理的规模。因此，在本书中我们自然希望为各种基础问题找到对数级别、线性级别或是线性对数级别的算法。

188

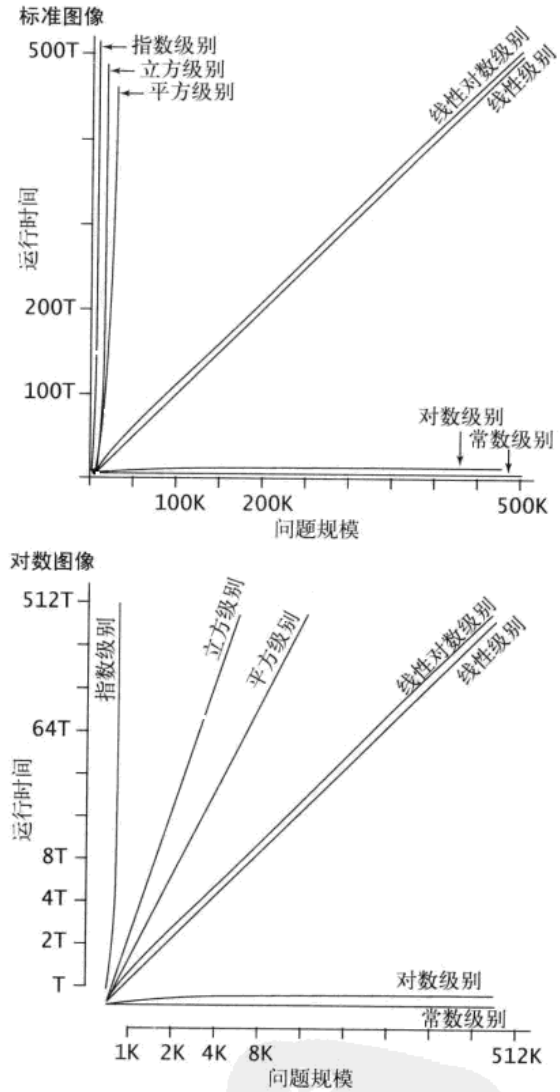


图 1.4.5 典型的增长数量级函数

1.4.5 设计更快的算法

学习程序的增长数量级的一个重要动力是为了帮助我们为同一个问题设计更快的算法。为了说明这一点，我们下面来讨论一个解决 3-sum 问题的更快的算法。我们甚至还没有开始学习

算法，怎么知道如何设计一个更快的算法呢？这个问题的答案是，我们已经讨论并使用过两个经典的算法，即*归并排序*和*二分查找*。也知道归并排序是线性对数级别的，二分查找是对数级别的。如何利用它们解决 3-sum 问题呢？

1.4.5.1 热身运动 2-sum

我们先来考虑这个问题的简化版本，即找出一个输入文件中所有和为 0 的整数对的数量。简单起见，我们还假设所有整数均各不相同。这个问题很容易在平方级别解决，只需将 `ThreeSum.count()` 中关于 `k` 的循环和 `a[k]` 去掉即可得到一个双层循环来检查所有的整数对，如表 1.4.7 中的*平方级别*条目所示（我们将这个实现称为 TwoSum）。下面这个实现显示了归并排序和二分查找是如何在*线性对数级别*解决 2-sum 问题的。改进后的算法的思想是当且仅当 `-a[i]` 存在于数组中（且 `a[i]` 非零）时，`a[i]` 存在于某个和为 0 的整数对之中。要解决这个问题，我们首先将数组排序（为二分查找做准备），然后对于数组中的每个 `a[i]`，使用 BinarySearch 的 `rank()` 方法对 `-a[i]` 进行二分查找。如果结果为 `j` 且 `j>i`，我们就将计数器加 1。这个简单的条件测试覆盖了三种情况：

- 如果二分查找不成功则会返回 -1，因此我们不会增加计数器的值；
- 如果二分查找返回的 `j>i`，我们就有 `a[i] + a[j] = 0`，增加计数器的值；
- 如果二分查找返回的 `j` 在 0 和 `i` 之间，我们也有 `a[i] + a[j] = 0`，但不能增加计数器的值，以避免重复计数。

这样得到的结果和平方级别的算法得到的结果完全相同，但它所需的时间要少得多。归并排序所需的时间和 $N\log N$ 成正比，二分查找所需的时间和 $\log N$ 成正比，因此整个算法的运行时间和 $N\log N$ 成正比。像这样设计一个更快的算法并不仅仅是一种学院派的练习——更快的算法使我们能够解决更庞大的问题。例如，你现在可以在可接受的时间范围内在计算机上解决 100 万个整数（1Mints.txt）的 2-sum 问题了，但如果用平方级别的算法你肯定需要等上很长很长的时间（请见练习 1.4.41）。

```
import java.util.Arrays;

public class TwoSumFast
{
   public static int count(int[] a)
   {  // 计算和为0的整数对的数目
      Arrays.sort(a);
      int N = a.length;
      int cnt = 0;
      for (int i = 0; i < N; i++)
         if (BinarySearch.rank(-a[i], a) > i)
            cnt++;
      return cnt;
   }

   public static void main(String[] args)
   {
      int[] a = In.readInts(args[0]);
      StdOut.println(count(a));
   }
}
```

2-sum 问题的线性对数级别的解法

1.4.5.2 3-sum 问题的快速算法

这种方式对 3-sum 问题同样有效。和刚才一样，我们假设所有整数均各不相同。当且仅当 `-(a[i] + a[j])` 在数组中（不是 `a[i]` 也不是 `a[j]`）时，整数对 (`a[i]` 和 `a[j]`) 为某个和为 0 的三元组的一部分。下面代码框中的代码会将数组排序并进行 $N(N-1)/2$ 次二分查找，每次查找所需的时间都和 $\log N$ 成正比。因此总运行时间和 $N^2\log N$ 成正比。可以注意到，在这种情况下排序的成本是次要因素。这个解法也使我们能够解决更大规模的问题（请见练习 1.4.42）。图 1.4.6 显示了用这 4 种算法解决我们提到过的几种问题规模时的成本的悬殊差距。这样的差距显然是我们追求更快的算法的动力。

189

1.4.5.3 下界

表 1.4.8 总结了本节所讨论的内容。我们立即产生了一个有趣的疑问：我们还能找到比 2-sum 问题的 TwoSumFast 和 3-sum 问题的 ThreeSumFast 快得多的算法吗？是否存在解决 2-sum 问题的线性级别的算法，3-sum 问题的线性对数级别的算法？对于 2-sum，这个问题的回答是没有（成本模型仅允许使用并计算这些整数的线性或是平方级别的函数中的比较操作）；对于 3-sum，回答是不知道，不过专家们相信 3-sum 可能的最优算法是平方级别的。为算法在最坏情况下的运行时间给出一个下界的思想是非常有意义的，我们会在 2.2 节中学习排序时再次讨论它。复杂的下界是很难找到的，但它非常有助于指引我们追求更加有效的算法。

190

```
import java.util.Arrays;

public class ThreeSumFast
{
   public static int count(int[] a)
   {  // 计算和为 0 的三元组的数目
      Arrays.sort(a);
      int N = a.length;
      int cnt = 0;
      for (int i = 0; i < N; i++)
         for (int j = i+1; j < N; j++)
            if (BinarySearch.rank(-a[i]-a[j], a) > j)
               cnt++;
      return cnt;
   }

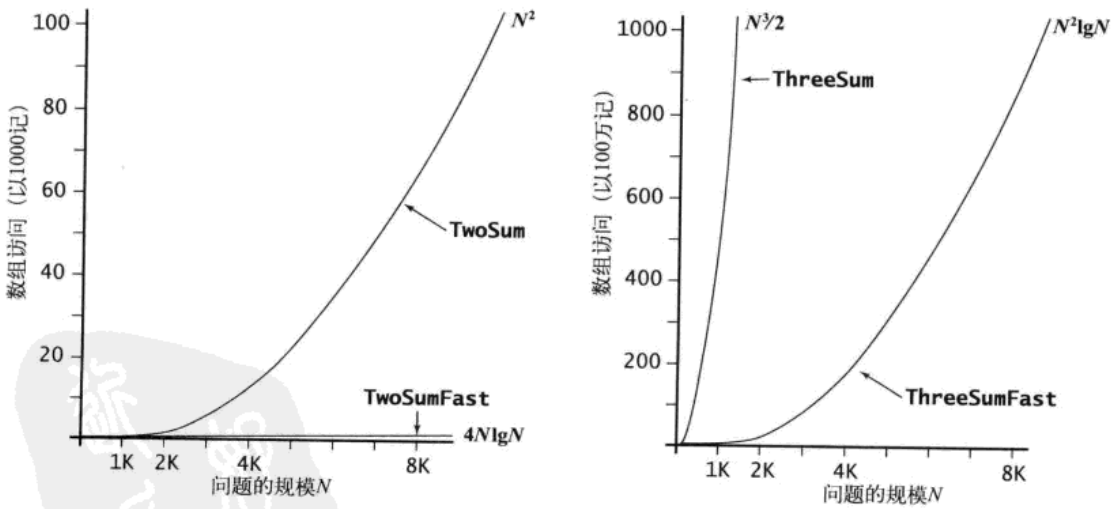
   public static void main(String[] args)
   {
      int[] a = In.readInts(args[0]);
      StdOut.println(count(a));
   }
}
```

3-sum问题的$N^2 \lg N$解法

本节中所讨论的例子为我们学习本书中的其他算法打下了基础。在本书中，我们会按照以下方式解决各种新的问题。

表 1.4.8 运行时间的总结

算 法	运行时间的增长数量级
TwoSum	N^2
TwoSumFast	$N \log N$
ThreeSum	N^3
ThreeSumFast	$N^2 \log N$



191

图 1.4.6 解决 2-sum 和 3-sum 问题的各种算法的成本

- 实现并分析该问题的一种简单的解法。我们通常将它们称为暴力算法，例如 ThreeSum 和 TwoSum。
- 考查算法的各种改进，它们通常都能降低算法所需的运行时间的增长数量级，例如 TwoSumFast 和 ThreeSumFast。
- 用实验证明新的算法更快。

在许多情况下，我们会学习解决同一个问题的多种算法，因为对于实际问题来说运行时间只是选择算法时所要考虑的各种因素之一。在本书中我们会在解决各种基础问题时逐渐理解这一点。

1.4.6 倍率实验

下面这种方法可以简单有效地预测任意程序的性能并判断它们的运行时间大致的增长数量级。

- 开发一个输入生成器来产生实际情况下的各种可能的输入（例如 DoublingTest 中的 `timeTrial()` 方法能够生成随机整数）。
- 运行下方的 DoublingRatio 程序，它是 DoublingTest 的修改版本，能够计算每次实验和上一次的运行时间的比值。
- 反复运行直到该比值趋近于极限 2^b。

这个实验对于比值没有极限的算法无效，但它仍然适用于许多程序，我们可以得出以下结论。

- 它们的运行时间的增长数量级约为 N^b。
- 要预测一个程序的运行时间，将上次观察得到的运行时间乘以 2^b 并将 N 加倍，如此反复。

如果你希望预测的输入规模不是 N 乘以 2 的幂，可以相应地调整这个比例（请见练习 1.4.9）。

如下所示，ThreeSum 的比例约为 8，因此我们可以预测程序对于 N=16 000、32 000 和 64 000 的运行时间将分别为 408.8、3270.4 和 26 163.2 秒，也就是处理 8000 个整数所需的时间（51.1 秒）连续乘以 8 即可。

实验程序

```
public class DoublingRatio
{
   public static double timeTrial(int N)
   // 参见 DoublingTest（请见 1.4.2.3 节实验程序）
   public static void main(String[] args)
   {
      double prev = timeTrial(125);
      for (int N = 250; true; N += N)
      {
         double time = timeTrial(N);
         StdOut.printf("%6d %7.1f ", N, time);
         StdOut.printf("%5.1f\n", time/prev);
         prev = time;
      }
   }
}
```

试验结果

```
% java DoublingRatio
   250     0.0   2.7
   500     0.0   4.8
  1000     0.1   6.9
  2000     0.8   7.7
  4000     6.4   8.0
  8000    51.1   8.0
```

预测

```
 16000   408.8   8.0
 32000  3270.4   8.0
 64000 26163.2   8.0
```

192

该测试基本类似于 1.4.2.3 节所描述的过程（运行实验，绘出对数图像得到运行时间为 aN^b 的猜想，从直线的斜率得到 b 的值，然后算出 a），但它更容易使用。事实上，可以手工通过 DoublingRatio 准确地预测程序的性能。在比例趋近于极限时，只需要不断乘以该比例即可得到更大规模的问题的运行时间。这里，增长数量级的近似模型是一个幂次法则，指数为该比例的以 2 为底的对数。

为什么这个比例会趋向于一个常数？简单的数学计算显示我们讨论过的所有常见的增长数量级函数（指数级别除外）均会出现这种情况：

命题 C。（倍率定理） 如果 $T(N) \sim aN^b \lg N$，那么 $T(2N)/T(N) \sim 2^b$。

证明。
$$\begin{aligned} T(2N)/T(N) &= a(2N)^b \lg(2N)/aN^b \lg N \\ &= 2^b(1+\lg 2/\lg N) \\ &\sim 2^b \end{aligned}$$

一般来说，数学模型中的对数项是不能忽略的，但在倍率假设中它在预测性能的公式中的作用并不那么重要。

在有性能压力的情况下**应该考虑**对编写过的所有程序进行倍率实验——这是一种估计运行时间的增长数量级的简单方法，或许它能够发现一些性能问题，比如你的程序并没有想象的那样高效。一般来说，我们可以用以下方式对程序的运行时间的增长数量级作出假设并预测它的性能。

1.4.6.1 评估它解决大型问题的可行性

对于编写的每个程序，你都需要能够回答这个基本问题：“该程序能在可接受的时间内处理这些数据吗？”对于大量数据，要回答这个问题我们需要一个比乘以 2 更大的系数（比如 10）来进行推断，如表 1.4.9 所示。无论是投资银行家处理每日的金融数据还是工程师对设计进行模拟测试，
193 定期运行需要若干个小时才能完成的程序是很常见的，表 1.4.9 的重点也就是这些情况。了解程序的运行时间的增长数量级能够为你提供精确的信息，从而理解你能够解决的问题规模的上限。理解诸如此类的问题，是研究性能的首要原因。没有这些知识，你将对一个程序所需的时间一无所知；而如果你有了它们，一张信封的背面就足够你计算出运行所需的时间并采取相应的行动。

1.4.6.2 评估使用更快的计算机所产生的价值

你可能会面对的另一个基本问题是：“如果我能够得到一台更快的计算机，解决问题的速度能够加快多少？”一般来说，如果新计算机比老的快 x 倍，运行时间也将变为原来的 x 分之一。但你一般都会用新计算机来处理更大规模的问题，这将会如何影响所需的运行时间呢？同样，增长的数量级信息也正是你回答这个问题所需要的。

著名的摩尔定律告诉我们，18 个月后计算机的速度和内存容量都会翻一番，5 年后计算机的速度和内存容量都会变为现在的 10 倍。表 1.4.9 说明如果你使用的是平方或者立方级别的算法，摩尔定律就不适用了。进行倍率测试并检查随着输入规模的倍增前后运行时间之比是趋向于 2 而非 4 或者 8 即可验证这种情况。

表 1.4.9 根据增长的数量级函数作出的预测

运行时间的增长数量级		系数为 2	系数为 10	处理输入规模为 N 的数据需要若干小时的某个程序	
描述	函数			处理 $10N$ 的预计时间	在快 10 倍的计算机上处理 $10N$ 的预计时间
线性级别	N	2	10	一天	几个小时
线性对数级别	$N\log N$	2	10	一天	几个小时
平方级别	N^2	4	100	几个星期	一天
立方级别	N^3	8	1000	几个月	几个星期
194 指数级别	2^N	2^N	2^{9N}	永远	永远

1.4.7 注意事项

在对程序的性能进行仔细分析时，得到不一致或是有误导性的结果的原因可能有许多种。它们都是由于我们的猜想基于的一个或多个假设并不完全正确所造成的。我们可以根据新的假设得出新的猜想，但我们考虑的细节越多，在分析中需要注意的方面也就越多。

1.4.7.1 大常数

在首项近似中，我们一般会忽略低级项中的常数系数，但这可能是错的。例如，当我们取函数 $2N^2+cN$ 的近似为 ~ $2N^2$ 时，我们的假设是 c 很小。如果事实不是这样（比如 c 可能是 10^3 或是 10^6），该近似就是错误的。因此，我们要对可能的大常数保持敏感。

1.4.7.2 非决定性的内循环

内循环是决定性因素的假设并不总是正确的。错误的成本模型可能无法得到真正的内循环，问题的规模 N 也许没有大到对指令的执行频率的数学描述中的首项大大超过其他低级项并可以忽略它们的程度。有些程序在内循环之外也有大量指令需要考虑。换句话说，成本模型可能还需要改进。

1.4.7.3 指令时间

每条指令执行所需的时间总是相同的假设并不总是正确的。例如，大多数现代计算机系统都会使用*缓存*技术来组织内存，在这种情况下访问大数组中的若干个并不相邻的元素所需的时间可能很长。如果让 DoublingRatio 运行的时间长一些，你可能可以观察到缓存对 ThreeSum 所产生的效果。在运行时间的比例看似收敛到 8 以后，由于缓存，对于大数组该比例也可能突然变为很大的值。

1.4.7.4 系统因素

一般来说，你的计算机总是同时运行着许多程序。Java 只是争夺资源的众多应用程序之一，而且 Java 本身也有许多能够大大影响程序性能的选项和设置。某种垃圾收集器或是 JIT 编译器或是正在从因特网中进行的下载都可能极大地影响实验的结果。这些因素可能会干扰到*实验必须是可重现的*这条科学研究的基本原则，因为此时此刻计算机中所发生的一切是无法再次重现的。原则上来说此时系统中运行的其他程序应该是可以忽略或可以控制的。

1.4.7.5 不分伯仲

在我们比较执行相同任务的两个程序时，常常出现的情况是其中一个在某些场景中更快而在另一些场景中更慢。我们已经提到过的一些因素可能会造成这种差异。有些程序员（以及一些学生）特别喜欢投入大量精力进行比赛并找出“最佳”的实现，但此类工作最好还是留给专家。 195

1.4.7.6 对输入的强烈依赖

在研究程序的运行时间的增长数量级时，我们首先作出的几个假设之一就是运行时间应该和输入相对无关。当这个条件无法满足时，我们很可能无法得到一致的结果或是验证我们的猜想。例如，假设我们为回答：“*输入中是否存在和为 0 的三个整数？*”而修改 ThreeSum 并返回 `boolean` 值，将 `cnt++` 替换为 `return true` 并在最后加上 `return false` 作为结尾，那么如果输入中的头三个整数的和为 0，该程序的运行时间的增长数量级为*常数级别*；如果输入不含有这样的三个整数，程序的运行时间的增长数量级则为*立方级别*。

1.4.7.7 多个问题参量

我们过去的重点一直是使用仅需要一个参量的函数来衡量程序的性能，参量一般是命令行参数或是输入的规模。但是，多个参量也是可能的。典型的例子是需要构造一个数据结构并使用该数据结构进行一系列操作的算法。在这种应用程序中数据结构的大小和操作的次数都是问题的参量。我们已经见过一个这样的例子，即对使用二分查找的白名单问题的分析，其中白名单中有 N 个整数而输入中有 M 个整数，运行时间一般和 $M\log N$ 成正比。

尽管需要注意的问题很多，对于每个程序员来说，对程序的运行时间的增长数量级的理解都是非常有价值的，而且我们这里所描述的方法也都十分强大并且应用范围广泛。Knuth 证明了原则上我们只要正确并完整地使用了这些方法就能够对程序作出详细准确的预测。计算机系统一般都非常复杂，完整精确的分析最好留给专家们，但相同的方法也可以有效地近似估计出任何程序所需的运行时间。火箭科学家需要大致知道一枚试验火箭的着陆地点是在大海里还是在城市中；医学研究者需要知道一次药物测试是会杀死还是治愈实验对象；任何使用计算机程序的科学家或是工程师也应
196 该能够预计它是会运行一秒钟还是一年。

1.4.8 处理对于输入的依赖

对于许多问题，刚才所提到的注意事项中最突出的一个就是对于输入的依赖，因为在这种情况下程序的运行时间的变化范围可能非常大。1.4.7.6 节中 ThreeSum 的修改版本的运行时间的范围根据输入的不同可能在常数级别到立方级别之间，因此如果我们想要预测它的性能，就需要对它进行更加细致的分析。在这里我们会简略讨论一些有效的方法，我们会在学习本书中的其他算法时用到它们。

1.4.8.1 输入模型

一种方法是更加小心地对我们所要解决的问题所处理的输入建模。例如，我们可能会假设 ThreeSum 的所有输入均为随机 `int` 值。使用这种方法的困难主要有两点：

- 输入模型可能是不切实际的；
- 对输入的分析可能极端困难，所需的数学技巧远非一般的学生或者程序员所能掌握。

其中前者更为重要，因为计算的目的就是发现输入的性质。例如，如果我们编写了一个程序来处理基因组，我们怎样才能估计出它在处理不同的基因组时的性能呢？描述自然界中的基因组的优秀模型正是科学家们所寻找的，因此预计我们的程序在处理自然界中得到的数据时所需的运行时间实际上也是在为寻找这个模型做出贡献！第二个困难只和最重要的几个算法的数学结果有关，我们将会看到几个用简单可靠的输入模型加上经典的数学分析帮助我们预测程序性能的例子。

1.4.8.2 对最坏情况下的性能的保证

有些应用程序要求程序对于任意输入的运行时间均小于某个指定的上限。为了提供这种性能保证，理论研究者们要从极度悲观的角度来估计算法的性能：在最坏情况下程序的运行时间是多少？例如，这种保守的做法对于运行在核反应堆、心脏起搏器或者刹车控制器之中的软件可能是十分必要的。我们希望保证此类软件能够在某个指定的时间范围内完成任务，否则结果会非常糟糕。科学家们在研究自然界时一般不会去考虑最坏的情况：在生物学中，最坏的情况也许是人类的灭绝；在物理学中，最坏的情况也许是宇宙的结束。但是在计算机系统中最坏情况是非常现实的忧虑，因为程序的输入可能来自另外一个（可能是恶意的）用户而非自然界。例如，没有使用提供性能保证算
197 法的网站无法抵御拒绝服务攻击，这是一种黑客用大量请求淹没服务器的攻击，会使网站的运行速度相比正常状态大幅下降。因此，我们的许多算法的设计已经考虑了为性能提供保证，例如：

命题 D。在 Bag（请见算法 1.4）、Stack（请见算法 1.2）和 Queue（请见算法 1.3）的链表实现中所有的操作在最坏情况下所需的时间都是常数级别的。

证明。由代码可知，每个操作所执行的指令数量均小于一个很小的常数。注意：该论证依赖于一个（合理的）假设，即 Java 系统能够在常数时间内创建一个新的 Node 对象。

1.4.8.3 随机化算法

为性能提供保证的一种重要方法是引入随机性。例如，我们将在2.3节中学习的快速排序算法（可能是使用最广泛的排序算法）在最坏情况下的性能是平方级别的，但通过随机打乱输入，根据概率我们能够保证它的性能是线性对数的。每次运行该算法，它所需的时间均不相同，但它的运行时间超过线性对数级别的可能性小到可以忽略。与此类似，我们将在3.4节中学习的用于符号表的散列算法（同样也可能是使用最广泛的同类算法）在最坏情况下的性能是线性级别的，但根据概率我们可以保证它的运行时间是常数级别的。这些保证并不是绝对的，但它们失效的可能性甚至小于你的电脑被闪电击中的可能性。因此，这种保证在实际中也可以用来作为最坏情况下的性能保证。

1.4.8.4 操作序列

对于许多应用来说，算法的“输入”可能并不只是数据，还包括用例所进行的一系列操作的顺序。例如，对于一个下压栈来说，用例先压入 N 个值然后再将它们全部弹出的所得到的性能，和 N 次压入弹出的混合操作序列所得到的性能可能大不相同。我们的分析要将这些情况都考虑进去（或者包含一个操作序列的合理模型）。

1.4.8.5 均摊分析

相应地，提供性能保证的另一种方法是通过记录所有操作的总成本并除以操作总数来将成本均摊。在这里，我们可以允许执行一些昂贵的操作，但保持所有操作的平均成本较低。这种类型分析的典型例子是我们在1.3节中对基于动态调整数组大小的Stack数据结构（请见1.3.2.5节的算法1.1）的研究。简单起见，假设 N 是2的幂。如果数据结构初始为空，N 次连续的 push() 调用需要访问数组元素多少次？计算这个答案很简单，数组访问的次数为 198

$$N+4+8+16+\cdots+2N=5N-4$$

其中，首项表示 N 次 push() 调用，其余的项表示每次数组长度加倍时初始化数据结构所访问数组的次数。因此，每次操作访问数组的平均次数为常数，但最后一次操作所需的时间是线性的。这种计算被称为均摊分析，因为我们将少量昂贵操作的成本通过各种大量廉价的操作摊平了。VisualAccumulator能够很容易地展示这个过程，如图1.4.7所示。

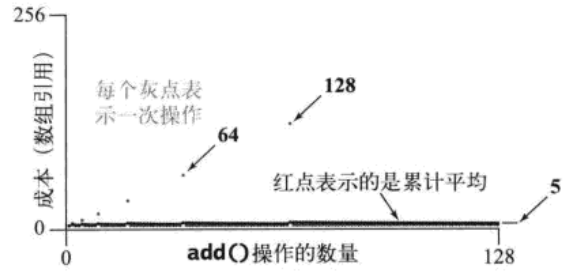


图 1.4.7 向一个 RandomBag 对象中添加元素时的均摊成本（另见彩插）

命题 E。在基于可调整大小的数组实现的Stack数据结构中（请见算法1.1），对空数据结构所进行的任意操作序列对数组的平均访问次数在最坏情况下均为常数。

简略证明。对于每次使数组大小增加（假设大小从 N 变为 $2N$）的 push() 操作，对于 $N/2+2$ 到 N 之间的任意 k，考虑使栈大小增长到 k 的最近 $N/2-1$ 次 push() 操作。将使数组长度加倍所需的 $4N$ 次访问和所有 push() 操作所需的 $N/2$ 次数组访问（每次 push() 操作均需访问一次数组）取平均，我们可以得到每次操作的平均成本为9次数组访问。要证明长度为 M 的任意操作序列所需的数组访问次数和 M 成正比则更加复杂（请见练习1.4.32）。

这种分析应用范围很广，我们会使用可动态调整大小的数组作为数据结构实现本书中的若干算法。

算法分析者的任务就是尽可能地揭示关于某个算法的更多信息，而程序员的任务则是利用这些信息开发有效解决现实问题的程序。在理想状态下，我们希望根据算法能够得到清晰简洁的代码并能够为我们感兴趣的输入提供良好的保证和性能。我们在本章中讨论的许多经典算法之所以对众多应用都十分重要就是因为它们具备这些性质。以它们作为样板，在编程中遇到典型问题时你也能独
199 立给出很好的解决方法。

1.4.9 内存

和运行时间一样，一个程序对内存的使用也和物理世界直接相关：计算机中的电路很大一部分的作用就是帮助程序保存一些值并在稍后取出它们。在任意时刻需要保存的值越多，需要的电路也就越多。你可能知道计算机能够使用的内存上限（知道这一点的人应该比知道运行时间限制的人要多）因为你很可能已经在内存上花了不少额外的支出。

计算机上的 Java 对内存的使用经过了精心的设计（程序的每个值在每次运行时所需的内存量都是一样的），但实现了 Java 的设备非常多，而内存的使用是和实现相关的。简单起见，我们用*典型*这个词暗示和机器相关的值。

Java 最重要的特性之一就是它的内存分配系统。它的任务是把你从对内存的操作之中解脱出来。显然，你肯定已经知道应该在适当的时候利用这个功能，但是你也应该（至少是大概）知道程序对内存的需求在何时会成为解决问题的障碍。

分析内存的使用比分析程序所需的运行时间要简单得多，主要原因是它所涉及的程序语句较少（只有声明语句）且在分析中我们会将复杂的对象简化为原始数据类型，而原始数据类型的内存使用是预先定义好的，而且非常容易理解：只需将变量的数量和它们的类型所对应的字节数分别相乘并汇总即可。例如，因为 Java 的 `int` 数据类型是 –2 147 483 648 到 2 147 483 647 之间的整数值的集合，即总数为 2^{32} 个不同的值，典型的 Java 实现使用 32 位来表示 `int` 值。其他原始数据类型的内存使用也是基于类似的考虑：典型的 Java 实现使用 8 位表示字节，用 2 字节（16 位）表示一个 `char` 值，用 4 字节（32 位）表示一个 `int` 值，用 8 字节（64 位）表示一个 `double` 或者 `long` 值，用 1 字节表示一个 `boolean` 值（因为计算机访问内存的方式都是一次 1 字节），见表 1.4.10。根据可用内存的总量就能够计算出保存这些值的极限数量。例如，如果计算机有 1 GB 内存（10 亿字节），那么同一时间最多能在内存中保存 3200 万个 `int` 值或是 1600 万个 `double` 值。

从另一方面来说，对内存使用的分析和硬件以及 Java 的不同实现中的各种差异有关，因此我们举出的这个特定的例子并不是一成不变的，你应该以它为参考来学习在条件允许的情况下如何分析内存的使用。例如，许多数据结果都涉及对机器地址的表示，而在
200 各种计算机中一个机器地址所需的内存又各有不同。为了保持一致，我们假设表示机器地址需要 8 字节，这是现在广泛使用的 64 位构架中的典型表示方式，许多老式的 32 位构架只使用 4 字节表示机器地址。

表 1.4.10 原始数据类型的常见内存需求

类 型	字 节
`boolean`	1
`byte`	1
`char`	2
`int`	4
`float`	4
`long`	8
`double`	8

1.4.9.1 对象

要知道一个对象所使用的内存量，需要将所有实例变量使用的内存与对象本身的开销（一般是

16 字节）相加。这些开销包括一个指向对象的类的引用、垃圾收集信息以及同步信息。另外，一般内存的使用都会被填充为 8 字节（64 位计算机中的机器字）的倍数。例如，一个 Integer 对象会使用 24 字节（16 字节的对象开销，4 字节用于保存它的 int 值以及 4 个填充字节）。类似地，一个 Date 对象（请见表 1.2.12）需要使用 32 字节：16 字节的对象开销，3 个 int 实例变量各需 4 字节，以及 4 个填充字节。对象的引用一般都是一个内存地址，因此会使用 8 字节。例如，一个 Counter 对象（请见表 1.2.11）需要使用 32 字节：16 字节的对象开销，8 字节用于它的 String 型实例变量（一个引用），4 字节用于 int 实例变量，以及 4 个填充字节。当我们说明一个引用所占的内存时，我们会单独说明它所指向的对象所占用的内存，因此这个内存使用总量并没有包含 String 值所使用的内存。常见对象的内存需求列在了图 1.4.8 中。

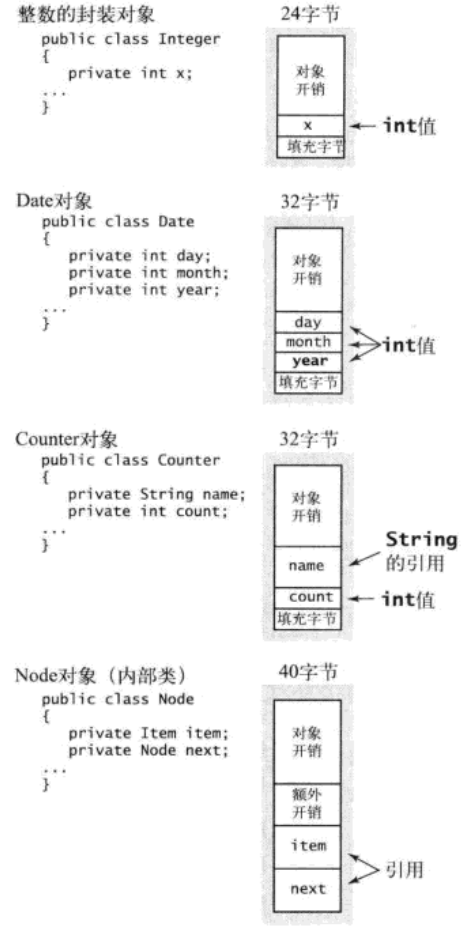


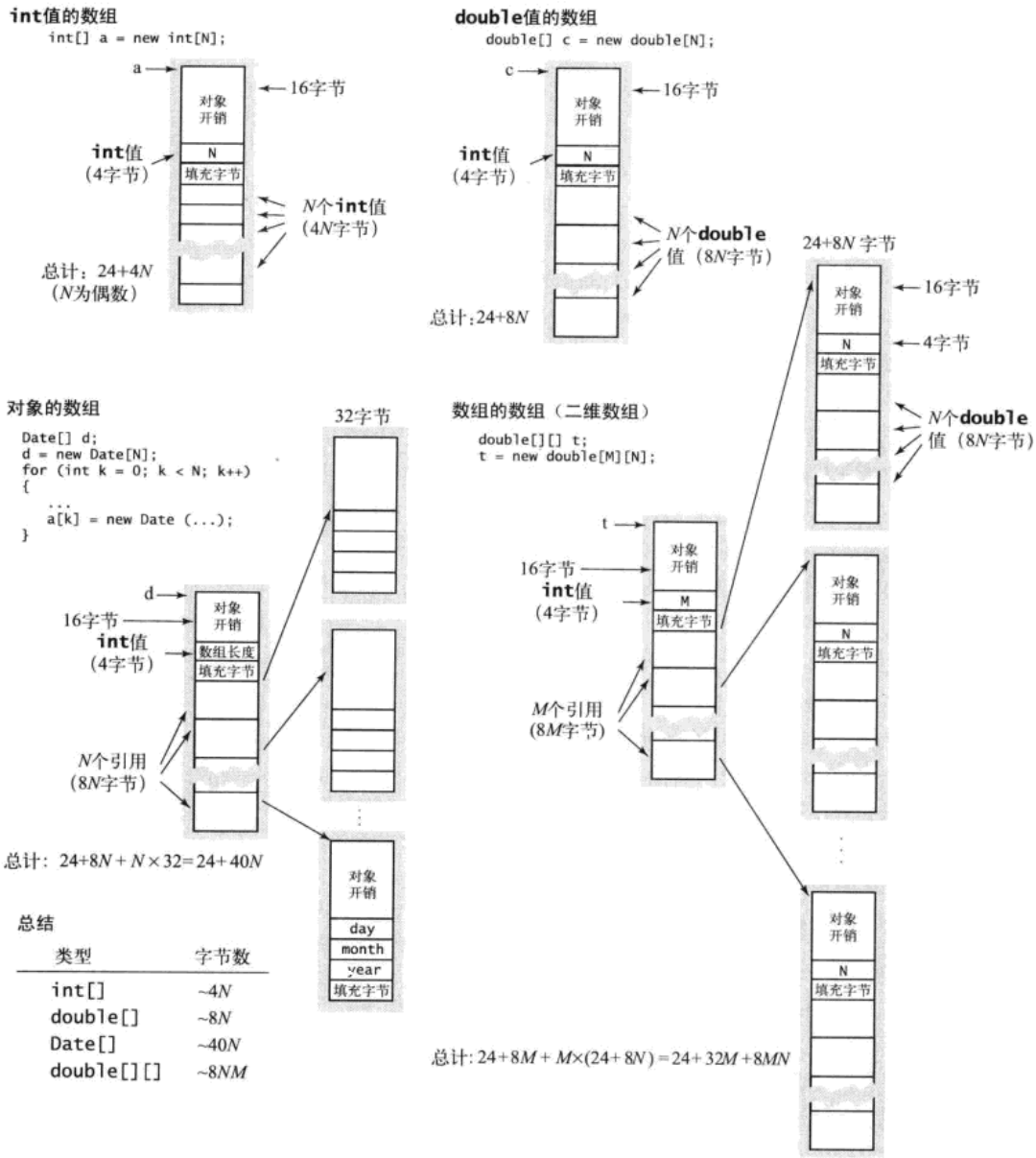
图 1.4.8 典型对象的内存需求

1.4.9.2 链表

嵌套的非静态（内部）类，例如我们的 Node 类（请见 1.3.3.1 节），还需要额外的 8 字节（用于一个指向外部类的引用）。因此，一个 Node 对象需要使用 40 字节（16 字节的对象开销，指向 Item 和 Node 对象的引用各需 8 字节，另外还有 8 字节的额外开销）。因为 Integer 对象需要使用 24 字节，一个含有 N 个整数的基于链表的栈（请见算法 1.2）需要使用（32+64N）字节，包括 Stack 对象的 16 字节的开销，引用类型实例变量 8 字节，int 型实例变量 4 字节，4 个填充字节，每个元素需要 64 字节，一个 Node 对象的 40 字节和一个 Integer 对象的 24 字节。 201

1.4.9.3 数组

图 1.4.9 总结了 Java 中的各种类型的数组对内存的典型需求。Java 中数组被实现为对象，它们一般都会因为记录长度而需要额外的内存。一个原始数据类型的数组一般需要 24 字节的头信息（16 字节的对象开销，4 字节用于保存长度以及 4 填充字节）再加上保存值所需的内存。例如，一个含有 N 个 int 值的数组需要使用（24 + 4N）字节（会被填充为 8 的倍数），一个含有 N 个 double 值的数组需要使用（24 + 8N）字节。一个对象的数组就是一个对象的引用的数组，所以我们应该在对象所需的内存之外加上引用所需的内存。例如，一个含有 N 个 Date 对象（请见表 1.2.12）的数组需要使用 24 字节（数组开销）加上 8N 字节（所有引用）加上每个对象的 32 字节，总共（24 + 40N）字节。二维数组是一个数组的数组（每个数组都是一个对象）。例如，一个 $M \times N$ 的 double 类型的二维数组需要使用 24 字节（数组的数组的开销）加上 8M 字节（所有元素数组的引用）加上 24M 字节（所有元素数组的开销）加上 8MN 字节（M 个长度为 N 的 double 类型的数组），总共（8MN+32M+24）~ 8MN 字节；当数组元素是对象时计算方法类似，结果相同，用来保存充满指向数组对象的引用的数组以及所有这些对象本身。



总结

类型	字节数
int[]	~4N
double[]	~8N
Date[]	~40N
double[][]	~8NM

图 1.4.9 int 值、double 值、对象和数组的数组对内存的典型需求

1.4.9.4 字符串对象

我们可以用相同的方式说明 Java 的 String 类型对象所需的内存，只是对于字符串来说别名是非常常见的。String 的标准实现含有 4 个实例变量：一个指向字符数组的引用（8 字节）和三个 int 值（各 4 字节）。第一个 int 值描述的是字符数组中的偏移量，第二个 int 值是一个计数器（字符串的长度）。按照图 1.4.9 中所示的实例变量名，对象所表示的字符串由 value[offset] 到 value[offset + count - 1] 中的字符组成。String 对象中的第三个 int 值是一个散列值，它在某些情况下可以节省一些计算，我们现在可以忽略它。因此，每个 String 对象总共会使用 40 字

节（16 字节表示对象，三个 int 实例变量各需 4 字节，加上数组引用的 8 字节和 4 个填充字节）。这是除字符数组之外字符串所需的内存空间，所有字符所需的内存需要另记，因为 String 的 char 数组常常是在多个字符串之间共享的。因为 String 对象是不可变的，这种设计使 String 的实现在能够在多个对象都含有相同的 value[] 数组时节省内存。

1.4.9.5 字符串的值和子字符串

一个长度为 N 的 String 对象一般需要使用 40 字节（String 对象本身）加上（24+2N）字节（字符数组），总共（64+2N）字节。但字符串处理经常会和子字符串打交道，所以 Java 对字符串的表示希望能够避免复制字符串中的字符。当你调用 substring() 方法时，就创建了一个新的 String 对象（40 字节），但它仍然重用了相同的 value[] 数组，因此该字符串的子字符串只会使用 40 字节的内存。含有原始字符串的字符数组的别名存在于子字符串中，子字符串对象的偏移量和长度域标记了子字符串的位置。换句话说，*一个子字符串所需的额外内存是一个常数，构造一个子字符串所需的时间也是常数*，即使字符串和子字符串的长度极大也是这样。某些简陋的字符串表示方法在创建子字符串时需要复制其中的字符，这将需要线性的时间和空间。确保子字符串的创建所需的空间（以及时间）和其长度无关是许多基础字符串处理算法的效率的关键所在。字符串的值与子字符串示例如图 1.4.10 所示。

202 ~ 203

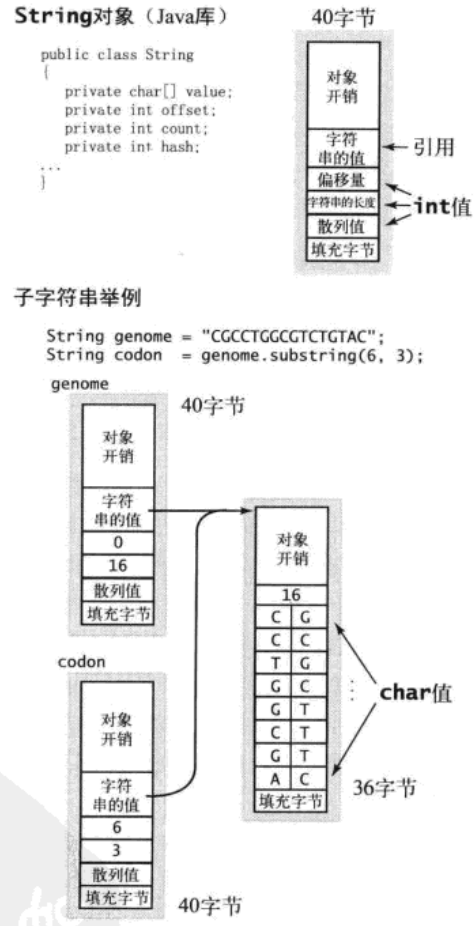


图 1.4.10 一个 String 对象和一个子字符串

204

这些基础机制能够有效帮助我们估计大量程序对内存的使用情况，但许多复杂的因素仍然会使这个任务变得更加困难。我们已经提到了别名可能产生的潜在影响。另外，当涉及函数调用时，内存的消耗就变成了一个复杂的动态过程，因为 Java 系统的内存分配机制扮演一个重要的角色，而这套机制又和 Java 的实现有关。例如，当你的程序调用一个方法时，系统会从内存中的一个特定区域为方法分配所需要的内存（用于保存局部变量），这个区域叫做栈（Java 系统的下压栈）。当方法返回时，它所占用的内存也被返回给了系统栈。因此，在递归程序中创建数组或是其他大型对象是很危险的，因为这意味着每一次递归调用都会使用大量的内存。当通过 new 创建对象时，系统会从堆内存的另一块特定区域为该对象分配所需的内存（这里的堆和我们将在 2.4 节学习的二叉堆数据结构不同）。而且，你要记住所有对象都会一直存在，直到对它的引用消失为止。此时系统的垃圾回收进程会将它所占用的内存收回到堆中。这种动态过程使准确估计一个程序的内存使用变得极为困难。

1.4.10 展望

良好的性能是非常重要的。速度极慢的程序和不正确的程序一样无用，因此显然有必要在一开

始就关注程序的运行成本，这能够让你大致估计出所要解决的问题的规模，而聪明的做法是时刻关注程序中的内循环代码的组成。

但在编程领域中，最常见的错误或许就是过于关注程序的性能。你的首要任务应该是写出清晰正确的代码。仅仅为了提高运行速度而修改程序的事最好留给专家们来做。事实上，这么做常常会降低生产效率，因为它会产生复杂而难以理解的代码。C.A.R. Hoare（快速排序的发明人，也是一位推动编写清晰而正确的代码的领军人物）曾将这种想法总结为："不成熟的优化是所有罪恶之源。"Knuth 为这句话加上了一个定语"在编程领域中（或者至少是大部分罪恶）"。另外，如果降低成本带来的效益并不明显，那么对运行时间的改进就不值得了。例如，如果一个程序所需的运行时间只是一瞬间而已，那么即使是将它的速度提高十倍也是无关紧要的。即使程序的运行需要好几分钟，实现并调试一个新算法所需要的时间也可能会大大超过直接运行一个稍微慢一点的算法——这种时候就应该让计算机代劳。更糟糕的情况是你可能花了大量的时间和心血去实现一个理论上能够改进程序的想法，但实际上什么也没发生。

在编程领域中，第二常见的错误或许是完全忽略了程序的性能。较快的算法一般都比暴力算法更复杂，所以很多人宁可使用较慢的算法也不愿应付复杂的代码。但是，几行优秀的代码有时能够给你带来巨大的收益。许多人在使用平方级别的暴力算法去解决问题的盲目等待中浪费了大量的时间，但实际上线性级别或是线性对数级别的算法能够在几分之一的时间内完成任务。当我们需要处理大规模问题时，通常，除了寻找更好的算法之外我们别无选择。

我们将使用本节所述的各种方法来评估算法对内存的使用，并在多个成本模型下对算法进行数学分析从而得到相应的近似函数，然后根据近似函数提出对算法所需的运行时间的增长数量级的猜想并通过实验验证它们。改进程序，使之更加清晰、高效和优雅应该是我们一贯的目标。如果你在
205 开发一个程序的全过程中都能关注它的运行成本，那么你都会从该程序的每次执行中受益。

答疑

问 为什么不用 StdRandom 生成随机数来代替 1Mints.txt？

答 在开发中，这样做能够使调试代码和重复实验更简单。每次调用 StdRandom 都会产生不同的值，所以修正一个 bug 之后并再次运行程序可能并不能测试这次修正！可以使用 StdRandom 中的 initialize() 方法来解决这个问题，但 1Mints.txt 类参考文件能够使添加测试用例变得更容易。另外，不同的程序员还能够比较程序在不同计算机上的性能而不必担心输入模型的不同。只要你的程序已经调试完毕且你已经大致了解了它的性能，当然有必要用随机数据测试它。例如，DoublingTest 和 DoublingRatio 使用的就是这种方式。

问 我在计算机上运行了 DoublingRatio，但我得到的结果和书上的不一致。有些比例的收敛值并不是 8，为什么？

答 这就是为什么我们在 1.4.7 节中讨论了注意事项。最可能的情况是你计算机上的操作系统在实验进行中还开小差去干了点儿别的活儿。消除这种问题的一种方式是花更多时间做更多次实验。比如，可以修改 DoublingTest，让它对于每个 N 都进行 1000 次实验，这样对于每个 N 它都能给出对运行时间更加精确的估计值。

问 在近似函数的定义中，"随着 N 的增大"确切的意思是什么？

答 $f(N)\sim g(N)$ 的正式定义为 $\lim_{N\to\infty} f(N)/g(N)=1$。

问 我还见到过其他表示增长的数量级的符号，它们都表示什么意思？

答 使用最广泛的记法是“大 O”：对于 $f(N)$ 和 $g(N)$，如果存在常数 c 和 N_0 使得对于所有 $N>N_0$ 都有 $|f(N)| < cg(N)$，则我们称 $f(N)$ 为 $O(g(N))$。这种记法在描述算法性能的渐进上限时十分有用，这在算法理论领域是十分重要的，但它在预测算法性能或是比较算法时并没有什么作用。

问 上题中，为什么说没有作用呢？

答 主要原因是它描述的仅仅是运行时间的上限，而算法的实际性能可能要好得多。一个算法的运行时间可能既是 $O(N^2)$ 也是 $\sim aN\log N$ 的。因此，它不能解释类似倍率实验等测试（请见 1.4.6 节命题 C）。 206

问 那为什么“大 O”符号的应用非常广泛呢？

答 因为它简化了对增长数量级的上限的研究，甚至也适用于一些无法进行精确分析的复杂算法。另外，它还可以和计算理论中用于将算法按照它们在最坏情况下的性能分类的“大 Omega”和“大 Theta”符号一起使用。如果存在常数 c 和 N_0 使得对于 $N>N_0$ 都有 $|f(N)| > cg(N)$，则我们称 $f(N)$ 为 $\Omega(g(N))$。如果 $f(N)$ 既是 $O(g(N))$ 也是 $\Omega(g(N))$，则我们称 $f(N)$ 为 $\Theta(g(N))$。“大 Omega”记法通常用来表示最坏情况下的性能下限，而“大 Theta”记法则通常用于描述算法的最优性能，即不存在有更好的最坏情况下的渐进增长数量级的算法。算法的最优性显然是实际应用中值得考虑的一点，但你会看到，还有其他许多因素需要考虑。

问 渐进性能的上限难道不重要吗？

答 重要，但我们希望讨论的是给定成本模型下所有语句执行的准确频率，因为它们能够提供更多关于算法性能的信息，而且从我们所讨论的算法中获取这些频率是可能的。例如，我们可以说“ThreeSum 访问数组的次数为 $\sim N^3/2$”，以及“在最坏情况下 `cnt++` 执行的次数为 $\sim N^3/6$”，它们虽然有些冗长但给出的信息比“ThreeSum 的运行时间为 $O(N^3)$”要多得多。

问 当一个算法的运行时间的增长数量级为 $N\log N$ 时，根据双倍测试会得到它的运行时间为 $\sim aN$ 的猜想（其中 a 为常数）。这有问题吗？

答 需要注意的是，我们不能根据实验数据推测它们所符合的某个特定的数学模型。但如果我们只是在预测性能，这并不是什么问题。例如，当 N 在 16 000 到 32 000 之间时，$14N$ 和 $N\lg N$ 的图像非常接近。这些数据同时与两条曲线吻合。随着 N 的增大，两条曲线更为接近。想要用实验来检验一个算法的运行时间是线性对数级别而非线性级别是要费一番工夫的。

问 `int[] a = new int[N]` 表示 N 次数组访问吗（所有数组元素均会被初始化为 0）？

答 大多数情况下是的，我们在本书中也是这样假设的，不过复杂编译器的实现会在遇到大型稀疏数组时尽力避免这种开销。 207

练习

1.4.1 证明从 N 个数中取三个整数的不同组合的总数为 $N(N-1)(N-2)/6$。提示：使用数学归纳法。

1.4.2 修改 ThreeSum，正确处理两个较大的 `int` 值相加可能溢出的情况。

1.4.3 修改 DoublingTest，使用 `StdDraw` 产生类似于正文中的标准图像和对数图像，根据需要调整比例使图像总能够充满窗口的大部分区域。

1.4.4 参照表 1.4.4 为 TwoSum 建立一张类似的表格。

1.4.5 给出下面这些量的近似：

a. $N+1$

b. $1+1/N$

c. $(1+1/N)(1+2/N)$

d. $2N^3-15N^2+N$

e. $\lg(2N)/\lg N$

f. $\lg(N^2+1)/\lg N$

g. $N^{100}/2^N$

1.4.6 给出以下代码段的运行时间的增长数量级（作为 N 的函数）：

a.
```
int sum = 0;
for (int n = N; n > 0; n /= 2)
  for(int i = 0; i < n; i++)
    sum++;
```

b.
```
int sum = 0;
for (int i = 1; i < N; i *= 2)
    for (int j = 0; j < i; j++)
        sum++;
```

208

c.
```
int sum = 0;
for (int i = 1; i < N; i *= 2)
   for (int j = 0; j < N; j++)
       sum++;
```

1.4.7 以统计涉及输入数字的算术操作（和比较）的成本模型分析 ThreeSum。

1.4.8 编写一个程序，计算输入文件中相等的整数对的数量。如果你的第一个程序是平方级别的，请继续思考并用 `Array.sort()` 给出一个线性对数级别的解答。

1.4.9 已知由倍率实验可得某个程序的时间倍率为 2^b 且问题规模为 N_0 时程序的运行时间为 T，给出一个公式预测该程序在处理规模为 N 的问题时所需的运行时间。

1.4.10 修改二分查找算法，使之总是返回和被查找的键匹配的索引最小的元素（且仍然能够保证对数级别的运行时间）。

1.4.11 为 `StaticSETofInts`（请见表 1.2.15）添加一个实例方法 `howMany()`，找出给定键的出现次数且在最坏情况下所需的运行时间和 $\log N$ 成正比。

1.4.12 编写一个程序，有序打印给定的两个有序数组（含有 N 个 `int` 值）中的所有公共元素，程序在最坏情况下所需的运行时间应该和 N 成正比。

1.4.13 根据正文中的假设分别给出表示以下数据类型的一个对象所需的内存量：

a. `Accumulator`

b. `Transaction`

c. `FixedCapacityStackOfStrings`，其容量为 C 且含有 N 个元素

d. `Point2D`

e. `Interval1D`

f. `Interval2D`

209

g. `Double`

提高题

1.4.14 4-sum。为 4-sum 设计一个算法。

1.4.15 快速 3-sum。作为热身，使用一个线性级别的算法（而非基于二分查找的线性对数级别的算法）实现 TwoSumFaster 来计算已排序的数组中和为 0 的整数对的数量。用相同的思想为 3-sum 问题给出一个平方级别的算法。

1.4.16 最接近的一对（一维）。编写一个程序，给定一个含有 N 个 double 值的数组 a[]，在其中找到一对最接近的值：两者之差（绝对值）最小的两个数。程序在最坏情况下所需的运行时间应该是线性对数级别的。

1.4.17 最遥远的一对（一维）。编写一个程序，给定一个含有 N 个 double 值的数组 a[]，在其中找到一对最遥远的值：两者之差（绝对值）最大的两个数。程序在最坏情况下所需的运行时间应该是线性级别的。

1.4.18 数组的局部最小元素。编写一个程序，给定一个含有 N 个不同整数的数组，找到一个局部最小元素：满足 a[i]<a[i - 1]，且 a[i]<a[i+1] 的索引 i。程序在最坏情况下所需的比较次数为 ~ 2lg$N$。

答：检查数组的中间值 a[N/2] 以及和它相邻的元素 a[N/2-1] 和 a[N/2+1]。如果 a[N/2] 是一个局部最小值则算法终止；否则则在较小的相邻元素的半边中继续查找。

1.4.19 矩阵的局部最小元素。给定一个含有 N^2 个不同整数的 $N \times N$ 数组 a[]。设计一个运行时间和 N 成正比的算法来找出一个局部最小元素：满足 a[i][j]<a[i+1][j]、a[i][j]<a[i][j+1]、a[i][j]<a[i-1][j] 以及 a[i][j]<a[i][j-1] 的索引 i 和 j。程序的运行时间在最坏情况下应该和 N 成正比。

1.4.20 双调查找。如果一个数组中的所有元素是先递增后递减的，则称这个数组为双调的。编写一个程序，给定一个含有 N 个不同 int 值的双调数组，判断它是否含有给定的整数。程序在最坏情况下所需的比较次数为 ~ 3lg$N$。

1.4.21 无重复值之中的二分查找。用二分查找实现 StaticSETofInts（请见表 1.2.15），保证 contains() 的运行时间为 ~ lg$R$，其中 R 为参数数组中不同整数的数量。 210

1.4.22 仅用加减实现的二分查找（Mihai Patrascu）。编写一个程序，给定一个含有 N 个不同 int 值的按照升序排列的数组，判断它是否含有给定的整数。只能使用加法和减法以及常数的额外内存空间。程序的运行时间在最坏情况下应该和 log$N$ 成正比。

答：用斐波纳契数代替 2 的幂（二分法）进行查找。用两个变量保存 F_k 和 F_{k-1} 并在 $[i, i+F_k]$ 之间查找。在每一步中，使用减法计算 F_{k-2}，检查 $i+F_{k-2}$ 处的元素，并根据结果将搜索范围变为 $[i, i+F_{k-2}]$ 或是 $[i+F_{k-2}, i+F_{k-2}+F_{k-1}]$。

1.4.23 分数的二分查找。设计一个算法，使用对数级别的比较次数找出有理数 p/q，其中 $0<p<q<N$，比较形式为给定的数是否小于 x？提示：两个分母均小于 N 的有理数之差不小于 $1/N^2$。

1.4.24 扔鸡蛋。假设你面前有一栋 N 层的大楼和许多鸡蛋，假设将鸡蛋从 F 层或者更高的地方扔下鸡蛋才会摔碎，否则则不会。首先，设计一种策略来确定 F 的值，其中扔 ~lg$N$ 次鸡蛋后摔碎的鸡蛋数量为 ~lg$N$，然后想办法将成本降低到 ~2lg$F$。

1.4.25 扔两个鸡蛋。和上一题相同的问题，但现在假设你只有两个鸡蛋，而你的成本模型则是扔鸡蛋的次数。设计一种策略，最多扔 $2\sqrt{N}$ 次鸡蛋即可判断出 F 的值，然后想办法把这个成本降低到 ~ c$\sqrt{F}$ 次。这和查找命中（鸡蛋完好无损）比未命中（鸡蛋被摔碎）的成本小得多的情形类似。

1.4.26 三点共线。假设有一个算法，接受平面上的 N 个点并能够返回在同一条直线上的三个点的组数。证明你能够用这个算法解决 3-sum 问题。强烈提示：使用代数证明当且仅当 a+b+c=0 时 (a, a^3)、(b, b^3) 和 (c, c^3) 在同一条直线上。 211

1.4.27 两个栈实现的队列。用两个栈实现一个队列，使得每个队列操作所需的堆栈操作均摊后为一个常数。提示：如果将所有元素压入栈再弹出，它们的顺序就被颠倒了。如果再次重复这个过程，它们的顺序则会复原。

1.4.28 一个队列实现的栈。使用一个队列实现一个栈，使得每个栈操作所需的队列操作数量为线性级别。提示：要删除一个元素，将队列中的所有元素一一出列再入列，除了最后一个元素，应该将它删除并返回（这种方法的确非常低效）。

1.4.29 两个栈实现的 steque。用两个栈实现一个 steque（请见练习 1.3.32），使得每个 steque 操作所需的栈操均摊后为一个常数。

1.4.30 一个栈和一个 steque 实现的双向队列。使用一个栈和 steque 实现一个双向队列（请见练习 1.3.32），使得双向队列的每个操作所需的栈和 steque 操作均摊后为一个常数。

1.4.31 三个栈实现的双向队列。使用三个栈实现一个双向队列，使得双向队列的每个操作所需的栈操作均摊后为一个常数。

1.4.32 均摊分析。请证明，对一个基于大小可变的数组实现的空栈的 M 次操作访问数组的次数和 M 成正比。

1.4.33 32 位计算机中的内存需求。给出 32 位计算机中 `Integer`、`Date`、`Counter`、`int[]`、`double[]`、`double[][]`、`String`、`Node` 和 `Stack`（链表表示）对象所需的内存，设引用需要 4 字节，表示对象开销为 8 字节，所需内存均会被填充为 4 字节的倍数。

1.4.34 热还是冷。你的目标是猜出 1 到 N 之间的一个秘密的整数。每次猜完一个整数后，你会知道你的猜测和这个秘密整数是否相等（如果是则游戏结束）。如果不相等，你会知道你的猜测相比上一次猜测距离该秘密整数是比较热（接近）还是比较冷（远离）。设计一个算法在 ~2lg$N$ 之内找到
212 这个秘密整数，然后再设计一个算法在 ~1lg$N$ 之内找到这个秘密整数。

1.4.35 下压栈的时间成本。解释下表中的数据，它显示了各种下压栈的实现的一般时间成本，其中成本模型会同时记录数据引用的数量（指向被压入栈之中的数据的引用，指向的可能是数组，也可能是某个对象的实例变量）和被创建的对象数量。

下压栈（的各种实现）的时间成本

数据结构	元素类型	压入 N 个 int 值的成本	
		数据的引用	创建的对象
基于链表	`int`	$2N$	N
	`Integer`	$3N$	$2N$
基于大小可变的数组	`int`	~ $5N$	lg$N$
	`Integer`	~ $5N$	~ N

1.4.36 下压栈的空间成本。解释下表中的数据，它显示了各种下压栈的实现的一般空间成本，其中链表的节点为一个静态的嵌套类，从而避免非静态嵌套类的开销。

下压栈（的各种实现）的空间成本

数据结构	元素类型	N 个 int 值所需的空间（字节）
基于链表	`int`	~ $32N$
	`Integer`	~ $56N$
基于大小可变的数组	`int`	~ $4N$ 到~ $16N$ 之间
213	`Integer`	~ $32N$ 到~ $56N$ 之间

实验题

1.4.37 *自动装箱的性能代价。*通过实验在你的计算机上计算使用自动装箱和自动拆箱所付出的性能代价。实现一个 FixedCapacityStackOfInts，并使用类似 DoublingRatio 的用例比较它和泛型 FixedCapacityStack<Integer> 在进行大量 push() 和 pop() 操作时的性能。

1.4.38 *3-sum 的初级算法的实现。*通过实验评估以下 ThreeSum 内循环的实现性能：

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        for (int k = 0; k < N; k++)
            if (i < j && j < k)
                if (a[i] + a[j] + a[k] == 0)
                    cnt++;
```

为此实现另一个版本的 DoublingTest，计算该程序和 ThreeSum 的运行时间的比例。

1.4.39 *改进倍率测试的精度。*修改 DoublingRatio，使它接受另一个命令行参数来指定对于每个 N 值调用 timeTrial() 方法的次数。用程序对每个 N 执行 10、100 和 1000 遍实验并评估结果的准确程度。

1.4.40 *随机输入下的 3-sum 问题。*猜测找出 N 个随机 int 值中和为 0 的整数三元组的数量所需的时间并验证你的猜想。如果你擅长数学分析，请为此问题给出一个合适的数学模型，其中所有值均匀地分布在 $-M$ 到 M 之间，且 M 不能是一个小整数。

1.4.41 *运行时间。*使用 DoublingRatio 估计在你的计算机上用 TwoSumFast、TwoSum、ThreeSumFast 以及 ThreeSum 处理一个含有 100 万个整数的文件所需的时间。

1.4.42 *问题规模。*设在你的计算机上用 TwoSumFast、TwoSum、ThreeSumFast 以及 ThreeSum 能够处理的问题的规模为 $2^P \times 10^3$ 个整数。使用 Doub lingRatio 估计 P 的最大值。

1.4.43 *大小可变的数组与链表。*通过实验验证对于栈来说基于大小可变的数组的实现快于基于链表的实现的猜想（请见练习 1.4.35 和练习 1.4.36）。为此实现另一个版本的 DoublingRatio，计算两个程序的运行时间的比例。 214

1.4.44 *生日问题。*编写一个程序，从命令行接受一个整数 N 作为参数并使用 StdRandom.uniform() 生成一系列 0 到 N–1 之间的随机整数。通过实验验证产生第一个重复的随机数之前生成的整数数量为 $\sim \sqrt{\pi N / 2}$。

1.4.45 *优惠券收集问题。*用和上一题相同的方式生成随机整数。通过实验验证生成所有可能的整数值所需生成的随机数总量为 $\sim N H_N$。 215

1.5 案例研究：union-find 算法

为了说明我们设计和分析算法的基本方法，我们现在来学习一个具体的例子。我们的目的是强调以下几点：

- 优秀的算法因为能够解决实际问题而变得更为重要；
- 高效算法的代码也可以很简单；
- 理解某个实现的性能特点是一项有趣而令人满足的挑战；
- 在解决同一个问题的多种算法之间进行选择时，科学方法是一种重要的工具；
- 迭代式改进能够让算法的效率越来越高。

我们会在本书中不断巩固这些主题思想。本节中的例子是一个原型，它将会为我们用相同的方法解决许多其他问题打下坚实的基础。

我们将要讨论的问题并非无足轻重，它是一个非常基础的计算性问题，而我们开发的解决方案将会用于多种实际应用之中，从物理化学中的渗流到通信网络中的连通性等。我们首先会给出一个简单的方案，然后对它的性能进行研究并由此得出应该如何继续改进我们的算法。

1.5.1 动态连通性

首先我们详细地说明一下问题：问题的输入是一列整数对，其中每个整数都表示一个某种类型的对象，一对整数 p q 可以被理解为“p 和 q 是相连的”。我们假设“相连”是一种对等的关系，这也就意味着它具有：

- 自反性：p 和 p 是相连的；
- 对称性：如果 p 和 q 是相连的，那么 q 和 p 也是相连的；
- 传递性：如果 p 和 q 是相连的且 q 和 r 是相连的，那么 p 和 r 也是相连的。

对等关系能够将对象分为多个等价类。在这里，当且仅当两个对象相连时它们才属于同一个等价类。我们的目标是编写一个程序来过滤掉序列中所有无意义的整数对（两个整数均来自于同一个等价类中）。换句话说，当程序从输入中读取了整数对 p q 时，如果已知的所有整数对都不能说明 p 和 q 是相连的，那么则将这一对整数写入到输出中。如果已知的数据可以说明 p 和 q 是相连的，那么程序应该忽略 p q 这对整数并继续处理输入中的下一对整数。图 1.5.1 用一个例子说明了这个过程。为了达到所期望的效果，我们需要设计一个数据结构来保存程序已知的所有整数对的足够多的信息，并用它们来判断一对新对象是否是相连的。我们将这个问题通俗地叫做动态连通性问题。这个问题可能有以下应用。

216

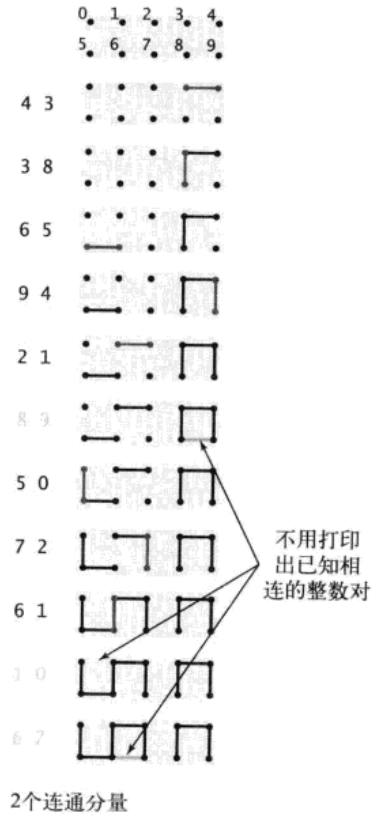


图 1.5.1 动态连通性问题（另见彩插）

1.5.1.1 网络

输入中的整数表示的可能是一个大型计算机网络中的计算机，而整数对则表示网络中的连接。这个程序能够判定我们是否需要在 p 和 q 之间架设一条新的连接才能进行通信，或是我们可以通过已有的连接在两者之间建立通信线路；或者这些整数表示的可能是电子电路中的触点，而整数对表

示的是连接触点之间的电路；或者这些整数表示的可能是社交网络中的人，而整数对表示的是朋友关系。在此类应用中，我们可能需要处理数百万的对象和数十亿的连接。

1.5.1.2 变量名等价性

某些编程环境允许声明两个等价的变量名（指向同一个对象的多个引用）。在一系列这样的声明之后，系统需要能够判别两个给定的变量名是否等价。这种较早出现的应用（如 FORTRAN 语言）推动了我们即将讨论的算法的发展。

1.5.1.3 数学集合

在更高的抽象层次上，可以将输入的所有整数看做属于不同的数学集合。在处理一个整数对 p q 时，我们是在判断它们是否属于相同的集合。如果不是，我们会将 p 所属的集合和 q 所属的集合归并，最终所有的整数属于同一个集合。

为了进一步限定话题，我们会在本节以下内容中使用网络方面的术语，将对象称为*触点*，将整数对称为*连接*，将等价类称为*连通分量*或是简称*分量*。简单起见，假设我们有用 0 到 N–1 的整数所表示的 N 个触点。这样做并不会降低算法的通用性，因为我们在第 3 章中将会学习一组高效的算法，将整数标识符和任意名称关联起来。

图 1.5.2 是一个较大的例子，意在说明连通性问题的难度。你很快就可以找到图左侧中部一个只含有一个触点的分量，以及左下方一个含有 5 个触点的分量，但让你验证其他所有触点是否都是 217
相互连通的可能就有些困难了。对于程序来说，这个任务更加困难，因为它所处理的只有触点的名字和连接而并不知道触点在图像中的几何位置。我们如何才能快速知道这种网络中任意给定的两个触点是否相连呢？

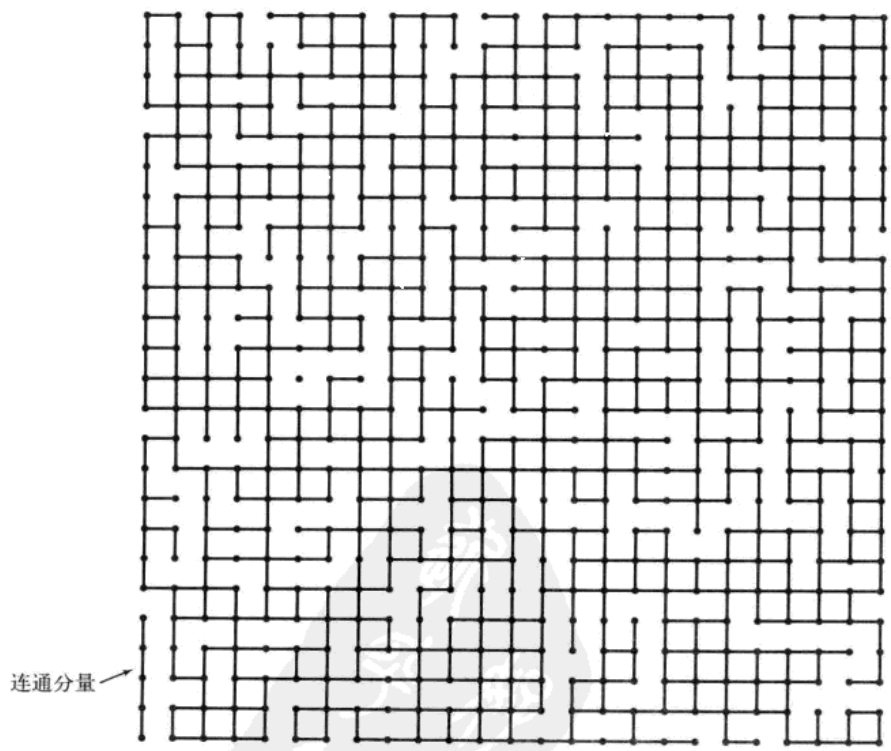


图 1.5.2 中等规模的连通性问题举例（625 个触点，900 条边，3 个连通分量）

我们在设计算法时面对的第一个任务就是精确地定义问题。我们希望算法解决的问题越大，它完成任务所需的时间和空间可能就越多。我们不可能预先知道这其间的量化关系，而且我们通
218 常只会在发现解决问题很困难，或是代价巨大，或是在幸运地发现算法所提供的信息比原问题所需要的更加有用时修改问题。例如，连通性问题只要求我们的程序能够判别给定的整数对 p q 是否相连，但并没有要求给出两者之间的通路上的所有连接。这样的要求会使问题更加困难，并得到另一组不同的算法，我们会在 4.1 节中学习它们。

为了说明问题，我们设计了一份 API 来封装所需的基本操作：初始化、连接两个触点、判断包含某个触点的分量、判断两个触点是否存在于同一个分量之中以及返回所有分量的数量。详细的 API 如表 1.5.1 所示。

表 1.5.1 union-find 算法的 API

public class UF		
	UF(int N)	以整数标识（0 到 $N-1$）初始化 N 个触点
void	union(int p, int q)	在 p 和 q 之间添加一条连接
int	find(int p)	p 所在的分量的标识符（0 到 $N-1$）
boolean	connected(int p, int q)	如果 p 和 q 存在于同一个分量中则返回 true
int	count()	连通分量的数量

如果两个触点在不同的分量中，union() 操作会将两个分量归并。find() 操作会返回给定触点所在的连通分量的标识符。connected() 操作能够判断两个触点是否存在于同一个分量之中。count() 方法会返回所有连通分量的数量。一开始我们有 N 个分量，将两个分量归并的每次 union() 操作都会使分量总数减一。

我们马上就将看到，为解决动态连通性问题设计算法的任务转化为了实现这份 API。所有的实现都应该：

- 定义一种数据结构表示已知的连接；
- 基于此数据结构实现高效的 union()、find()、connected() 和 count() 方法。

众所周知，数据结构的性质将直接影响到算法的效率，因此数据结构和算法的设计是紧密相关的。API 已经说明触点和分量都会用 int 值表示，所以我们可以用一个以触点为索引的数组 id[]
219 作为基本数据结构来表示所有分量。我们将使用分量中的某个触点的名称作为分量的标识符，因此你可以认为每个分量都是由它的触点之一所表示的。一开始，我们有 N 个分量，每个触点都构成了一个只含有它自己的分量，因此我们将 id[i] 的值初始化为 i，其中 i 在 0 到 N–1 之间。对于每个触点 i，我们将 find() 方法用来判定它所在的分量所需的信息保存在 id[i] 之中。connected() 方法的实现只用一条语句 find(p) == find(q)，它返回一个布尔值，我们在所有方法的实现中都会用到 connected() 方法。

总之，我们的起点就是算法 1.5。我们维护了两个实例变量，一个是连通分量的个数，一个是数组 id[]。find() 和 union() 的实现是本节剩余内容将要讨论的主题。

算法 1.5 union-find 的实现

```
public class UF
{
   private int[] id;     // 分量id（以触点作为索引）
   private int count;    // 分量数量
```

```
public UF(int N)
{  // 初始化分量id数组
   count = N;
   id = new int[N];
   for (int i = 0; i < N; i++)
      id[i] = i;
}
public int count()
{  return count;  }
public boolean connected(int p, int q)
{  return find(p) == find(q);  }
public int  find(int p)
public void union(int p, int q)
// 请见1.5.2.1节用例（quick-find）、1.5.2.3节用例（quick-union）和算法1.5（加权quick-union）
public static void main(String[] args)
{  // 解决由StdIn得到的动态连通性问题
   int N = StdIn.readInt();                  // 读取触点数量
   UF uf = new UF(N);                        // 初始化N个分量
   while (!StdIn.isEmpty())
   {
      int p = StdIn.readInt();
      int q = StdIn.readInt();               // 读取整数对
      if (uf.connected(p, q)) continue;      // 如果已经连通则忽略
      uf.union(p, q);                        // 归并分量
      StdOut.println(p + " " + q);          // 打印连接
   }
   StdOut.println(uf.count() + "components");
}
}
```

```
% java UF < tinyUF.txt
4 3
3 8
6 5
9 4
2 1
5 0
7 2
6 1
2 components
```

这份代码是我们对 UF 的实现。它维护了一个整型数组 `id[]`，使得 `find()` 对于处在同一个连通分量中的触点均返回相同的整数值。`union()` 方法必须保证这一点。

为了测试 API 的可用性并方便开发，我们在 `main()` 方法中包含了一个用例用于解决动态连通性问题。它会从输入中读取 N 值以及一系列整数对，并对每一对整数调用 `find()` 方法：如果某一对整数中的两个触点已经连通，程序会继续处理下一对数据；如果不连通，程序会调用 `union()` 方法并打印这对整数。在讨论实现之前，我们也准备了一些测试数据（如右侧的代码框所示）：文件 tinyUF.txt 含有 10 个触点和 11 条连接，图 1.5.1 使用的就是它；文件 mediumUF.txt 含有 625 个触点和 900 条连接，如图 1.5.2 所示；例子文件 largeUF.txt 含有 100 万个触点和 200 万条连接。我们的目标是在可以接受的时间范围内处理和 largeUF.txt 规模类似的输入。

为了分析算法，我们将重点放在不同算法访问任意数组元素的总次数上。我们这样做相当于隐式地猜测各种算法在一台特定的计算机上的运行时间在这个量乘以某个常数的范围之内。这个猜想基于代码，用实验验证它并不困难。我们将会看到，这个猜想是算法比较的一个很好的开始。

```
% more tinyUF.txt
10
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7

% more mediumUF.txt
625
528 503
548 523
...
900条连接

% more largeUF.txt
1000000
786321 134521
696834 98245
...
200万条连接
```