Concepts of Programming Languages, Tenth Edition

PEARSON

编程语言原理

(第10版)

PEARSON

[美] Robert W. Sebesta 著
马跃 王敏 王国栋 译

清华大学出版社

013027846

TP312
934

编程语言原理

(第 10 版)

[美] Robert W. Sebesta 著

马 跃 王 敏 王国栋 译

北京航空航天大学
藏书
图书馆

TP312
934

清华大学出版社

北 京

北航 C1637048

Authorized translation from the English language edition, entitled Concepts of Programming Languages, Tenth Edition, 978-0-13-139531-2 by Robert W. Sebesta, published by Pearson Education, Inc, publishing as Addison-Wesley, Copyright © 2012.

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 TSINGHUA UNIVERSITY PRESS Copyright © 2012.

北京市版权局著作权合同登记号 图字：01-2012-8974

本书封面贴有 Pearson Education(培生教育出版集团)公司防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

编程语言原理(第 10 版)/(美)塞巴斯塔(Sebesta, R. W.) 著；马跃，王敏，王国栋 译. —北京：清华大学出版社，2013.3

书名原文：Concepts of Programming Languages, Tenth Edition

ISBN 978-7-302-31112-6

Ⅰ.①编… Ⅱ.①塞… ②马… ③王… ④王… Ⅲ.①程序语言 Ⅳ.①TP312

中国版本图书馆 CIP 数据核字(2012)第 308695 号

责任编辑：王　军　刘伟琴
装帧设计：牛静敏
责任校对：成凤进
责任印制：沈　露

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：三河市新茂装订有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：41　　**字　　数**：998 千字
版　　次：2013 年 3 月第 1 版　　**印　　次**：2013 年 3 月第1 次印刷
印　　数：1～3000
定　　价：98.00 元

产品编号：048510-01

前　言

第 10 版的变化

本书沿用了前 9 版的写作目标、总体结构和论述手法。本书的首要目标是介绍当代编程语言的主要构成，为读者提供必要的工具来评价已有的和未来的编程语言。另一个目标是，深入讨论编程语言的结构，展示语法描述的形式化方法，介绍词法和语法分析的手段，让读者做好学习编译器设计的准备。

第 10 版与第 9 版相比，有几种不同的变化。为了保持内容的新颖性，删除了对一些过时的编程语言的讨论。例如，第 6 章删除了 COBOL 的记录操作，第 8 章删除了 Fortran 的 Do 语句，第 9 章删除了 Ada 的泛型子程序，第 13 章删除了 Ada 的异步消息传送。

另一方面，第 9 章增加了闭包、间接调用子程序和 F#中的泛型函数等内容，第 11 章和第 12 章都增加了 Objective-C 一节，第 13 章增加了函数式编程语言中的并发，第 14 章添加了 C#事件处理一节，第 15 章增加了 F#、基本命令式语言对函数式编程的支持这两节。

有些内容还进行了移动。例如，函数式编程语言中的几个不同结构从第 15 章移到了前面的章节中。其中，函数式编程语言中的控制语句移到了第 8 章，Scheme 和 ML 的列表及列表操作移到了第 6 章。这些移动表示本书理念的重大变化——在某种程度上，是函数式编程语言的某些结构回归主流。在以前的版本中，函数式编程语言的全部结构都放在第 15 章中。

第 11、12 和 15 章也做了大幅修订，第 12 章还增加了 5 个新图。

最后，本书的许多章节都有很多小改动，主要是为了使论述更加清晰。

为了清晰起见，下面列出第 10 版的新增内容：

- 第 5 章：增加了函数式编程语言中的 let 结构。
- 第 6 章：删除了 COBOL 的记录操作，增加了 F#中的列表、元组和联合等章节。
- 第 8 章：删除了 Fortran 的 Do 语句和 Ada 的 case 语句，函数式编程语言中的控制语句从第 15 章移至该章。
- 第 9 章：增加了闭包、间接调用子程序和 F#中的泛型函数等节，删除了 Ada 的泛型子程序。
- 第 11 章：增加了 Objective-C 一节，该章的内容做了大幅修订。
- 第 12 章：增加了 Objective-C 一节和 5 个新图。
- 第 13 章：增加了函数式编程语言中的并发，删除了 Ada 的异步消息传送。
- 第 14 章：增加了 C#事件处理一节。

- 第 15 章：增加了 F#、基本命令式语言对函数式编程的支持这两节，函数式编程语言中的几个不同结构从该章移至前面的章节中。

写作宗旨

本书描述了编程语言的基本原理，讨论了各种语言构成的设计问题，分析了一些常见语言对这些结构所做的设计选择，并细致地比较了各种备选设计方案。

为了严肃认真地研究学习编程语言，需要探讨一些相关的主题，如编程语言语法和语义的形式化描述方法(参见第 3 章)。同时，不同语言构成的实现技术也是需要考虑的：词法和语法分析参见第 4 章，子程序链接的实现参见第 10 章。其他一些语言构成的实现参见本书的其他章节。

下面将概述第 10 版的内容。

各章概要

第 1 章首先介绍了研究程序设计语言的根本原因，接着讨论了用于评价编程语言和语言构成的准则，也探讨了语言设计的主要影响、常见的设计折中以及基本的实现手段。

第 2 章概述了本书讨论的大多数重要语言的发展过程。尽管每种语言都没有叙述得很完整，但讨论了每种语言的起源、目标和贡献。这种历史回顾是有价值的，因为它为理解当代语言设计的实践基础和理论基础提供了必要的背景知识。它也激发了学生深入学习语言设计与评估的兴趣。另外，由于本书的其他章节都不依赖第 2 章，所以它自成体系，可以独立于其他章节进行阅读。

第 3 章介绍了描述编程语言语法的主要形式化方法 BNF。接着讨论了属性文法(用来描述语言的语法和静态语义)。随后探讨了语义描述的难点任务，包括对 3 种最常见方法的简要介绍：操作语义、指称语义以及公理语义。

第 4 章介绍了词法和语法分析。这一章面向课程表中不需要编译器设计课程的大学生。和第 2 章一样，这一章是独立的，可与本书其他部分分开阅读。

第 5 章～第 14 章详细叙述了命令式编程语言的基本构成的设计问题。对每个问题，都以几种语言为例介绍了设计选择，并进行了评估。具体而言，第 5 章介绍了变量的许多特征，第 6 章阐述了数据类型，第 7 章解释了表达式和赋值语句，第 8 章叙述了控制语句，第 9 章和第 10 章讨论了子程序及其实现方式，第 11 章研讨了数据抽象机制，第 12 章深入讨论了支持面向对象编程的语言特征，包括继承和动态方法绑定等，第 13 章讨论了并发程序单元，第 14 章介绍了异常处理，还简要讨论了事件处理。

最后两章(第 15 章和第 16 章)描述了两种最重要的非主流程序设计范型：函数式编程与逻辑编程。但函数式编程语言的一些数据结构和控制语句放在第 6 章和第 8 章讨论。第

15 章介绍了 Scheme，包括一些基本函数、特殊结构、函数形式以及用 Scheme 写的简单函数示例。对 ML、Haskell 和 F#的简要介绍展示了一些不同类型的函数式语言设计。第 16 章介绍了逻辑编程和逻辑编程语言 Prolog。

寄语教师

在科罗拉多大学科罗拉多斯普林分校低年级的编程语言课程中，本书是这样使用的：我们会详细讲解第 1 章和第 3 章，而对第 2 章，尽管学生发现它很有趣，并从阅读中受益，但我们在第 2 章上花费的教学时间较少，因为它没有什么难懂的技术内容。就像前面讲过的，因为后续各章的内容都不依赖第 2 章，所以这一章可以完全跳过去。另外，因为我们有一门编译器设计课程，所以第 4 章也不在教学安排中。

对于在 C++、Java 或 C#方面有大量编程经验的学生，第 5 章～第 9 章的内容是比较容易的。第 10 章～第 14 章则具有一些挑战性，需要更细致地加以讲解。

第 15 章和第 16 章对大多数低年级学生来说完全是新的。理想情况下，对于要求学习这些章节内容的学生，应该给学生提供 Scheme 和 Prolog 的语言处理器。这两章包含了充足的资料，可以让学生拿一些简单的程序来练习。

本科生课程或许并不能涵盖最后两章的所有内容。不过，研究生课程应该跳过最初几章中对命令式语言的讨论，以便完整地讨论这两章的内容。

补充材料

在网站 www.pearsonhighered.com/cssupport 上，为本书所有读者提供了以下补充材料：

- 一系列教学幻灯片，包含本书每一章的 PowerPoint 幻灯片。
- 包含本书所有图片的 PowerPoint 幻灯片。

本书的配套网站是 www.pearsonhighered.com/sebesta。这个站点包含关于几种语言的微型手册(约 100 页的教程)。这些手册假定学生掌握了如何使用其他一些语言编程。这些手册为学生提供了足够的信息，各章内容都有每种语言的完整版本。当前该站点包含 C++、C、Java 和 Smalltalk 的手册。

有资格的教师可以在教师资源中心 www.pearsonhighered.com/irc 上找到许多习题的答案。请联系您所在学校的 Pearson Education 代表，或访问 www.pearsonhighered.com/irc 进行注册。

获取语言处理器的途径

对于本书讨论的一些编程语言，可以从下面的站点找到它们的处理器和相关信息：

C、C++、Fortran 和 Ada	gcc.gnu.org
C#和 F#	microsoft.com
Java	java.sun.com
Haskell	haskell.org
Lua	www.lua.org
Scheme	www.plt-scheme.org/software/drscheme
Perl	www.perl.com
Python	www.python.org
Ruby	www.ruby-lang.org

几乎所有浏览器都包含了 JavaScript；几乎所有 Web 服务器都包含了 PHP。

上述所有信息也都可以在本书的相关网站上找到。

致　　谢

对本书目前的形式，出色的审阅人员提供了许多很好的建议。他们是(按字母排序)：

Matthew Michael Burke	
I-ping Chu	德宝大学
Teresa Cole	博伊西州立大学
Pamela Cutter	卡拉马祖学院
Amer Diwan	科罗拉多大学
Stephen Edwards	弗吉尼亚科技大学
David E. Goldschmidt	
Nigel Gwee	南部大学巴顿鲁治分校
Timothy Henry	罗德岛大学
Paul M. Jackowitz	斯克兰顿大学
Duane J. Jarc	马里兰大学
K. N. King	佐治亚州立大学
Donald Kraft	路易斯安那州立大学
Simon H. Lin	加州州立大学北岭分校
Mark Llewellyn	中佛罗里达大学
Bruce R. Maxim	密歇根大学迪尔本分校
Robert McCloskey	斯克兰顿大学
Curtis Meadow	缅因州立大学
Gloria Melara	加州州立大学北岭分校
Frank J. Mitropoulos	诺瓦东南大学
Euripides Montagne	中佛罗里达大学
Serita Nelesen	加尔文学院
Bob Neufeld	卫奇塔州立大学
Charles Nicholas	马里兰大学-巴尔的摩分校
Tim R. Norton	科罗拉多大学斯普林斯分校
Richard M. Osborne	科罗拉多大学丹佛分校
Saverio Perugini	代顿大学
Walter Pharr	查尔斯顿学院
Michael Prentice	纽约州立大学布法罗分校
Amar Raheja	加州理工大学波莫纳分校

Hossein Saiedian	堪萨斯大学
Stuart C. Shapiro	纽约州立大学布法罗分校
Neelam Soundarajan	俄亥俄州立大学
Ryan Stansifer	佛罗里达理工学院
Nancy Tinkham	罗恩大学
Paul Tymann	罗彻斯特理工学院
Cristian Videira Lopes	加州大学欧文分校
Sumanth Yenduri	南密西西比大学
Salih Yurttas	德克萨斯州 A&M 大学

在本书以前版本的不同发展阶段，很多人都提供了写作素材。他们所有的评论对我都很有用，我非常感谢。他们是(按字母排序)：Vicki Allan、 Henry Bauer、Carter Bays、Manuel E. Bermudez、Peter Brouwer、Margaret Burnett、Paosheng Chang、Liang Cheng、John Crenshaw、Charles Dana、Barbara Ann Griem、Mary Lou Haag、John V. Harrison、Eileen Head、Ralph C. Hilzer、Eric Joanis、Leon Jololian、Hikyoo Koh、Jiang B. Liu、Meiliu Lu、Jon Mauney、Robert McCoard、Dennis L. Mumaugh、Michael G. Murphy、Andrew Oldroyd、Young Park、Rebecca Parsons、 Steve J. Phelps、 Jeffery Popyack、 Raghvinder Sangwan、 Steven Rapkin、 Hamilton Richard、Tom Sager、Joseph Schell、Sibylle Schupp、Mary Louise Soffa、Neelam Soundarajan、Ryan Stansifer、 Steve Stevenson、 Virginia Teller、 Yang Wang、 John M. Weiss、 Franck Xia 和 Salih Yurnas.

还要感谢编辑 Matt Goldstein、编辑助理 Chelsea Kharakozova、Addison-Wesley 的高级产品主管 Marilyn Lloyd 和 Aardvark 出版服务组的 Gillian Hall，他们为快速而又严谨地出版第 10 版付出了极大的努力。

目　录

第 1 章　预备知识 ····· 1
1.1　学习程序设计语言原理的原因 ····· 2
1.2　程序设计领域 ····· 4
1.2.1　科学应用 ····· 4
1.2.2　商务应用 ····· 4
1.2.3　人工智能 ····· 5
1.2.4　系统程序设计 ····· 5
1.2.5　网络软件 ····· 5
1.3　语言评价标准 ····· 6
1.3.1　可读性 ····· 6
1.3.2　可写性 ····· 10
1.3.3　可靠性 ····· 11
1.3.4　成本 ····· 13
1.4　影响语言设计的因素 ····· 14
1.4.1　计算机体系结构 ····· 14
1.4.2　程序设计方法学 ····· 15
1.5　程序设计语言的分类 ····· 16
1.6　语言设计中的权衡 ····· 17
1.7　实现方法 ····· 18
1.7.1　编译 ····· 19
1.7.2　完全解释 ····· 21
1.7.3　混合实现系统 ····· 22
1.7.4　预处理器 ····· 23
1.8　编程环境 ····· 23

第 2 章　主要程序设计语言的发展 ····· 27
2.1　Zuse 的 Plankalkül 语言 ····· 29
2.1.1　历史背景 ····· 29
2.1.2　语言概述 ····· 29
2.2　伪代码 ····· 30
2.2.1　Short Code 语言 ····· 31
2.2.2　Speedcoding 系统 ····· 31
2.2.3　UNIVAC “编译” 系统 ····· 32
2.2.4　相关工作 ····· 32
2.3　IBM 704 计算机与 Fortran 语言 ····· 32
2.3.1　历史背景 ····· 32
2.3.2　设计过程 ····· 33
2.3.3　Fortran I 概述 ····· 33
2.3.4　Fortran II ····· 34
2.3.5　Fortran IV、77、90、95、2003 和 2008 ····· 34
2.3.6　评价 ····· 35
2.4　函数式程序设计：LISP 语言 ····· 36
2.4.1　人工智能的起源和表处理 ····· 36
2.4.2　LISP 语言的设计过程 ····· 37
2.4.3　语言概述 ····· 37
2.4.4　评价 ····· 39
2.4.5　LISP 的两种后代语言 ····· 39
2.4.6　相关语言 ····· 40
2.5　迈向成熟的第一步：ALGOL 60 ····· 41
2.5.1　历史背景 ····· 41
2.5.2　早期设计过程 ····· 41
2.5.3　ALGOL 58 概述 ····· 42
2.5.4　对 ALGOL 58 报告的响应 ····· 42
2.5.5　ALGOL 60 的设计过程 ····· 43
2.5.6　ALGOL 60 概述 ····· 43
2.5.7　评价 ····· 44
2.6　商务记录的计算机化：COBOL 语言 ····· 45
2.6.1　历史背景 ····· 46

2.6.2 FLOW-MATIC 语言 46
2.6.3 COBOL 语言的设计过程 46
2.6.4 评价 47
2.7 分时处理的开始：BASIC 语言 49
2.7.1 设计过程 49
2.7.2 语言概述 50
2.7.3 评价 50
2.8 满足所有人的需要：PL/I 51
2.8.1 历史背景 51
2.8.2 设计过程 52
2.8.3 语言概述 53
2.8.4 评价 53
2.9 两种早期的动态语言：APL 和 SNOBOL 54
2.9.1 APL 语言的起源与特点 54
2.9.2 SNOBOL 语言的起源与特点 55
2.10 数据抽象的开始：SIMULA 67 55
2.10.1 设计过程 55
2.10.2 语言概述 55
2.11 正交设计：ALGOL 68 56
2.11.1 设计过程 56
2.11.2 语言概述 56
2.11.3 评价 56
2.12 ALGOL 系列语言的早期后代语言 57
2.12.1 为简单性而设计：Pascal 语言 57
2.12.2 可移植的系统语言：C 语言 59
2.13 基于逻辑的程序设计：Prolog 语言 60
2.13.1 设计过程 61
2.13.2 语言概述 61
2.13.3 评价 61
2.14 历史上规模最大的设计工作：Ada 语言 62
2.14.1 历史背景 62
2.14.2 设计过程 62
2.14.3 语言概述 63
2.14.4 评价 64
2.14.5 Ada 95 和 Ada 2005 65
2.15 面向对象的程序设计：Smalltalk 65
2.15.1 设计过程 65
2.15.2 语言概述 66
2.15.3 评价 67
2.16 结合命令式和面向对象的特性：C++ 67
2.16.1 设计过程 68
2.16.2 语言概述 68
2.16.3 评价 69
2.16.4 一种相关语言：Objective-C 69
2.16.5 另一种相关语言：Delphi 69
2.16.6 一种关系不大的语言：Go 70
2.17 基于命令式的面向对象语言：Java 70
2.17.1 设计过程 70
2.17.2 语言概述 71
2.17.3 评价 72
2.18 脚本语言 73
2.18.1 Perl 的起源与特点 73
2.18.2 JavaScript 的起源与特点 75
2.18.3 PHP 的起源与特点 76
2.18.4 Python 的起源与特点 77
2.18.5 Ruby 的起源与特点 77
2.18.6 Lua 的起源与特点 78
2.19 一流的.NET 语言：C# 79
2.19.1 设计过程 79
2.19.2 语言概述 79

2.19.3 评价 80
2.20 标记与程序设计混合的语言 81
2.20.1 XSLT 81
2.20.2 JSP 82

第 3 章 描述语法和语义 89
3.1 概述 90
3.2 描述语法的普遍问题 91
3.2.1 语言识别器 91
3.2.2 语言生成器 92
3.3 描述语法的形式化方法 92
3.3.1 巴科斯-诺尔范式和上下文无关文法 92
3.3.2 扩展的 BNF 102
3.3.3 文法与识别器 104
3.4 属性文法 104
3.4.1 静态语义 105
3.4.2 基本概念 105
3.4.3 属性文法定义 105
3.4.4 本质属性 106
3.4.5 属性文法的例子 106
3.4.6 计算属性值 108
3.4.7 评价 109
3.5 描述程序的意义：动态语义 109
3.5.1 操作语义 110
3.5.2 指称语义 112
3.5.3 公理语义 116

第 4 章 词法分析和语法分析 133
4.1 概述 134
4.2 词法分析 135
4.3 语法分析问题 142
4.3.1 语法分析概述 142
4.3.2 自顶向下的语法分析器 142
4.3.3 自底向上的语法分析器 143
4.3.4 语法分析的复杂度 144
4.4 递归下降的语法分析 144
4.4.1 递归下降的语法分析过程 144
4.4.2 LL 文法类 149
4.5 自下而上的语法分析 152
4.5.1 自下而上的语法分析器的分析问题 152
4.5.2 移进-归约算法 154
4.5.3 LR 语法分析器 155

第 5 章 名字、绑定和作用域 163
5.1 概述 164
5.2 名字 164
5.2.1 设计问题 165
5.2.2 名字形式 165
5.2.3 特殊字 166
5.3 变量 166
5.3.1 名字 167
5.3.2 地址 167
5.3.3 类型 168
5.3.4 数值 168
5.4 绑定的概念 168
5.4.1 属性与变量绑定 169
5.4.2 绑定类型 169
5.4.3 存储绑定和生存期 172
5.5 作用域 175
5.5.1 静态作用域 175
5.5.2 块 177
5.5.3 声明的次序 179
5.5.4 全局作用域 180
5.5.5 静态作用域的评价 182
5.5.6 动态作用域 183
5.5.7 动态作用域的评价 183
5.6 作用域和生存期 184
5.7 引用环境 185
5.8 命名常量 186

第 6 章 数据类型 195
6.1 概述 196

6.2 基本数据类型 197
6.2.1 数值类型 197
6.2.2 布尔类型 200
6.2.3 字符类型 200
6.3 字符串类型 200
6.3.1 设计问题 201
6.3.2 字符串及其操作 201
6.3.3 字符串长度的设计选项 203
6.3.4 评价 203
6.3.5 字符串类型的实现 203
6.4 用户定义的序数类型 204
6.4.1 枚举类型 205
6.4.2 子界类型 207
6.4.3 实现用户定义的有序类型 208
6.5 数组类型 208
6.5.1 设计问题 208
6.5.2 数组和索引 209
6.5.3 下标的绑定和数组的种类 210
6.5.4 数组的初始化 212
6.5.5 数组操作 213
6.5.6 矩形数组和不规则数组 215
6.5.7 切片 215
6.5.8 评价 216
6.5.9 数组类型的实现 216
6.6 关联数组 218
6.6.1 结构和操作 219
6.6.2 关联数组的实现 220
6.7 记录类型 220
6.7.1 记录的定义 221
6.7.2 记录域的引用 222
6.7.3 评价 222
6.7.4 记录类型的实现 223
6.8 元组类型 223
6.9 列表类型 224
6.10 联合类型 227
6.10.1 设计问题 227
6.10.2 判别式联合与自由联合 227
6.10.3 Ada 的联合类型 228
6.10.4 F#的联合类型 229
6.10.5 评价 230
6.10.6 联合类型的实现 230
6.11 指针和引用类型 231
6.11.1 设计问题 232
6.11.2 指针操作 232
6.11.3 指针的相关问题 233
6.11.4 Ada 中的指针 234
6.11.5 C 和 C++中的指针 235
6.11.6 引用类型 236
6.11.7 评价 237
6.11.8 指针和引用类型的实现 237
6.12 类型检查 242
6.13 强类型化 242
6.14 类型等价 243
6.15 理论和数据类型 246

第 7 章 表达式与赋值语句 257
7.1 概述 258
7.2 算术表达式 258
7.2.1 运算符的运算顺序 259
7.2.2 操作数的运算顺序 264
7.3 运算符重载 266
7.4 类型转换 267
7.4.1 表达式中的强制类型转换 267
7.4.2 显式类型转换 269
7.4.3 表达式中的错误 269
7.5 关系表达式和布尔表达式 270
7.5.1 关系表达式 270
7.5.2 布尔表达式 270
7.6 短路求值 271
7.7 赋值语句 273
7.7.1 简单赋值 273
7.7.2 条件赋值 273
7.7.3 混合赋值运算符 273

7.7.4 一元赋值运算符 ····· 274
7.7.5 赋值表达式 ····· 275
7.7.6 多重赋值 ····· 276
7.7.7 函数式编程语言中的赋值 ····· 276
7.8 混合模式赋值 ····· 277

第 8 章 语句级控制结构 ····· 283
8.1 概述 ····· 284
8.2 选择语句 ····· 285
8.2.1 双路选择语句 ····· 285
8.2.2 多重选择结构 ····· 289
8.3 迭代语句 ····· 295
8.3.1 计数控制循环 ····· 296
8.3.2 逻辑控制循环 ····· 300
8.3.3 用户自定义的循环控制机制 ····· 302
8.3.4 基于数据结构的迭代 ····· 303
8.4 无条件分支 ····· 306
8.5 防护命令 ····· 307
8.6 结论 ····· 309

第 9 章 子程序 ····· 315
9.1 概述 ····· 316
9.2 子程序的基本原理 ····· 316
9.2.1 子程序的一般性质 ····· 316
9.2.2 子程序的基本定义 ····· 316
9.2.3 参数 ····· 318
9.2.4 过程与函数 ····· 321
9.3 子程序的设计问题 ····· 322
9.4 局部引用环境 ····· 323
9.4.1 局部变量 ····· 323
9.4.2 嵌套子程序 ····· 324
9.5 参数传递方式 ····· 325
9.5.1 参数传递的语义模型 ····· 325
9.5.2 参数传递的实现模型 ····· 326
9.5.3 参数传递方法的实现 ····· 329
9.5.4 常见语言的参数传递方法 ····· 330
9.5.5 参数的类型检查 ····· 332
9.5.6 多维数组作为参数 ····· 333
9.5.7 设计考虑 ····· 336
9.5.8 参数传递的例子 ····· 336
9.6 子程序作为参数 ····· 339
9.7 间接调用子程序 ····· 340
9.8 重载子程序 ····· 342
9.9 泛型子程序 ····· 343
9.9.1 C++中的泛型函数 ····· 343
9.9.2 Java 5.0 中的泛型方法 ····· 345
9.9.3 C# 2005 中的泛型方法 ····· 346
9.9.4 F#中的泛型函数 ····· 347
9.10 函数的设计问题 ····· 348
9.10.1 函数的副作用 ····· 348
9.10.2 返回值的类型 ····· 348
9.10.3 返回值的个数 ····· 348
9.11 用户定义重载运算符 ····· 349
9.12 闭包 ····· 349
9.13 协同程序 ····· 351

第 10 章 实现子程序 ····· 359
10.1 调用和返回的一般语义 ····· 360
10.2 实现“简单”的子程序 ····· 360
10.3 通过栈动态局部变量实现子程序 ····· 362
10.3.1 更复杂的活动记录 ····· 363
10.3.2 一个不含递归调用的例子 ····· 365
10.3.3 递归调用 ····· 367
10.4 嵌套子程序 ····· 368
10.4.1 基础知识 ····· 368
10.4.2 静态链 ····· 369
10.5 块 ····· 373
10.6 动态作用域的实现 ····· 375
10.6.1 深层访问 ····· 375
10.6.2 浅层访问 ····· 376

第 11 章 抽象数据类型与封装结构 …… 383
11.1 抽象的概念 …… 384
11.2 数据抽象简介 …… 385
11.2.1 抽象数据类型之浮点型 …… 385
11.2.2 用户自定义的抽象数据类型 …… 386
11.2.3 示例 …… 387
11.3 抽象数据类型的设计问题 …… 387
11.4 语言示例 …… 388
11.4.1 Ada 中的抽象数据类型 …… 388
11.4.2 C++中的抽象数据类型 …… 391
11.4.3 Objective-C 中的抽象数据类型 …… 395
11.4.4 Java 中的抽象数据类型 …… 400
11.4.5 C#中的抽象数据类型 …… 401
11.4.6 Ruby 中的抽象数据类型 …… 403
11.5 参数化的抽象数据类型 …… 406
11.5.1 Ada …… 406
11.5.2 C++ …… 407
11.5.3 Java 5.0 …… 409
11.5.4 C# 2005 …… 411
11.6 封装结构 …… 411
11.6.1 引言 …… 411
11.6.2 C 中的封装 …… 412
11.6.3 C++中的封装 …… 412
11.6.4 Ada 包 …… 413
11.6.5 C#程序集 …… 413
11.7 命名封装 …… 414
11.7.1 C++命名空间 …… 414
11.7.2 Java 包 …… 415
11.7.3 Ada 包 …… 416
11.7.4 Ruby 模块 …… 417

第 12 章 面向对象程序设计的支持 …… 425
12.1 概述 …… 426
12.2 面向对象编程 …… 426
12.2.1 引言 …… 426
12.2.2 继承 …… 426
12.2.3 动态绑定 …… 428
12.3 面向对象语言的设计问题 …… 430
12.3.1 对象的排他性 …… 430
12.3.2 子类是子类型吗 …… 430
12.3.3 单继承与多继承 …… 431
12.3.4 对象的分配和释放 …… 432
12.3.5 动态绑定与静态绑定 …… 433
12.3.6 嵌套类 …… 433
12.3.7 对象的初始化 …… 433
12.4 Smalltalk 对面向对象编程的支持 …… 434
12.4.1 一般特征 …… 434
12.4.2 继承 …… 434
12.4.3 动态绑定 …… 435
12.4.4 Smalltalk 的评价 …… 435
12.5 C++对面向对象编程的支持 …… 436
12.5.1 一般特征 …… 436
12.5.2 继承 …… 436
12.5.3 动态绑定 …… 441
12.5.4 评价 …… 443
12.6 Objective-C 对面向对象编程的支持 …… 444
12.6.1 一般特征 …… 444
12.6.2 继承 …… 445
12.6.3 动态绑定 …… 446
12.6.4 评价 …… 447
12.7 Java 对面向对象编程的支持 …… 447
12.7.1 一般特征 …… 447
12.7.2 继承 …… 447
12.7.3 动态绑定 …… 449

12.7.4 被嵌套的类……449
12.7.5 评价……450
12.8 C#对面向对象编程的支持……450
12.8.1 一般特征……450
12.8.2 继承……450
12.8.3 动态绑定……450
12.8.4 被嵌套的类……451
12.8.5 评价……451
12.9 Ada 95 对面向对象编程的支持……452
12.9.1 一般特征……452
12.9.2 继承……452
12.9.3 动态绑定……453
12.9.4 子包……454
12.9.5 评价……455
12.10 Ruby 对面向对象编程的支持……455
12.10.1 一般特征……455
12.10.2 继承……457
12.10.3 动态绑定……457
12.10.4 评价……457
12.11 面向对象构造的实现……457
12.11.1 存储实例数据……457
12.11.2 方法调用到方法的动态绑定……458

第 13 章 并发……467
13.1 概述……468
13.1.1 多处理器体系结构……469
13.1.2 并发的种类……470
13.1.3 使用并发的目的……471
13.2 子程序级并发概述……471
13.2.1 基本概念……471
13.2.2 为并发而设计的语言……475
13.2.3 设计问题……475
13.3 信号量……475
13.3.1 概述……475
13.3.2 合作同步……476
13.3.3 竞争同步……478
13.3.4 评价……479
13.4 管程……479
13.4.1 概述……479
13.4.2 竞争同步……480
13.4.3 合作同步……480
13.4.4 评价……480
13.5 消息传递……481
13.5.1 概述……481
13.5.2 同步消息传递的原理……481
13.6 Ada 对并发的支持……482
13.6.1 基本原理……482
13.6.2 合作同步……485
13.6.3 竞争同步……486
13.6.4 任务终止……487
13.6.5 优先级……487
13.6.6 受保护对象……488
13.6.7 评价……489
13.7 Java 线程……489
13.7.1 Thread 类……490
13.7.2 优先级……491
13.7.3 信号量……492
13.7.4 竞争同步……492
13.7.5 合作同步……493
13.7.6 非阻塞同步……496
13.7.7 显式锁定……496
13.7.8 评价……497
13.8 C#线程……497
13.8.1 基本线程操作……497
13.8.2 同步线程……500
13.8.3 评价……500
13.9 函数式语言中的并发……501
13.9.1 Multilisp……501
13.9.2 并发 ML……501
13.9.3 F#……502
13.10 语句级并发……503

第 14 章 异常处理和事件处理 …… 511
14.1 异常处理概述 …… 512
14.1.1 基本概念 …… 513
14.1.2 设计问题 …… 514
14.2 Ada 中的异常处理 …… 517
14.2.1 异常处理程序 …… 517
14.2.2 将异常绑定到处理程序 …… 518
14.2.3 继续 …… 518
14.2.4 其他设计选择 …… 519
14.2.5 例子 …… 520
14.2.6 评价 …… 522
14.3 C++中的异常处理 …… 522
14.3.1 异常处理程序 …… 522
14.3.2 异常与处理程序的绑定 …… 523
14.3.3 继续 …… 524
14.3.4 其他设计选择 …… 524
14.3.5 例子 …… 524
14.3.6 评价 …… 526
14.4 Java 中的异常处理 …… 526
14.4.1 异常类 …… 526
14.4.2 异常处理程序 …… 527
14.4.3 异常与处理程序的绑定 …… 527
14.4.4 其他设计选择 …… 528
14.4.5 例子 …… 529
14.4.6 finally 子句 …… 530
14.4.7 断言 …… 531
14.4.8 评价 …… 531
14.5 事件处理概述 …… 532
14.6 Java 的事件处理 …… 533
14.6.1 Java Swing 的 GUI 组件 …… 533
14.6.2 Java 事件模型 …… 534
14.7 C#中的事件处理 …… 537

第 15 章 函数式程序设计语言 …… 545
15.1 概述 …… 546
15.2 数学函数 …… 547
15.2.1 简单函数 …… 547
15.2.2 函数形式 …… 548
15.3 函数式程序设计语言基础 …… 549
15.4 第一种函数式程序设计语言 LISP …… 550
15.4.1 数据类型和结构 …… 550
15.4.2 第一个 LISP 解释器 …… 551
15.5 Scheme 概述 …… 553
15.5.1 Scheme 的起源 …… 553
15.5.2 Scheme 解释器 …… 553
15.5.3 基本数值函数 …… 553
15.5.4 定义函数 …… 554
15.5.5 输出函数 …… 556
15.5.6 数值谓词函数 …… 556
15.5.7 控制流 …… 557
15.5.8 表函数 …… 557
15.5.9 用于符号原子和表的谓词函数 …… 560
15.5.10 Scheme 函数示例 …… 561
15.5.11 LET …… 564
15.5.12 Scheme 中的尾递归 …… 565
15.5.13 函数形式 …… 566
15.5.14 构建代码的函数 …… 567
15.6 Common LISP …… 568
15.7 ML …… 570
15.8 Haskell …… 574
15.9 F# …… 578
15.10 基本命令式语言对函数式编程的支持 …… 581
15.11 函数式语言和命令式语言的比较 …… 582

第 16 章 逻辑程序设计语言 …… 589
16.1 概述 …… 590

16.2　谓词演算简介……590
16.2.1　命题……591
16.2.2　子句形式……592
16.3　谓词演算与定理证明……593
16.4　逻辑程序设计概述……595
16.5　Prolog 的起源……596
16.6　Prolog 的基本元素……597
16.6.1　项……597
16.6.2　事实语句……598
16.6.3　规则语句……598
16.6.4　目标语句……599
16.6.5　Prolog 的推理过程……600
16.6.6　简单算术……602
16.6.7　表结构……604
16.7　Prolog 存在的缺陷……609
16.7.1　归结的顺序控制……609
16.7.2　封闭世界假设……610
16.7.3　否定问题……611
16.7.4　固有的限制……612
16.8　逻辑程序设计的应用……613
16.8.1　关系数据库管理系统……613
16.8.2　专家系统……613
16.8.3　自然语言处理……614
参考文献……617

第1章

预备知识

1.1 学习程序设计语言原理的原因
1.2 程序设计领域
1.3 语言评价标准
1.4 影响语言设计的因素
1.5 程序设计语言的分类
1.6 语言设计中的权衡
1.7 实现方法
1.8 编程环境

在开始介绍程序设计语言原理之前，必须了解一些预备知识。首先本章将解释为什么计算机科学专业的学生与专业的软件开发人员，应该学习语言设计与评价的一般性原理，这对于那些认为计算机科学家只需要一两种程序设计语言的应用知识就足够的人来说，很有意义；第二，简要介绍主要的程序设计领域；第三，由于本书将评价语言构成和特性，所以会列出一些标准作为评价的基础；第四，介绍影响语言设计的两大要素：计算机体系结构和程序设计方法学；第五，介绍各类程序设计语言；第六，给出几种语言设计必须考虑的主要的取舍方法。

由于本书也涉及程序设计语言的实现，因此本章概述了最常用的实现方法。最后简要介绍一些编程环境的实例，讨论它们对软件开发的影响。

1.1 学习程序设计语言原理的原因

学生自然想知道，学习程序设计语言原理有什么好处，毕竟计算机科学中有许多其他课题需要认真研习。我们认为，学习程序设计语言原理具有以下这些潜在的优点：

- **增强表达思想的能力**。一般认为，人们思考问题的深度受到他们交流其思想时所用语言的表达能力的影响。那些对自然语言理解肤浅的人，思维的复杂度也有限，特别是在抽象的深度上。换言之，人们难以将口头或书面无法表达的东西概念化。

  程序员在开发软件的过程中同样受到这一限制。他们开发软件所用的语言限制了可用的控制结构、数据结构和抽象层次，从而限制了他们能够构造的算法形式。了解更多的程序设计语言特性，能够在软件开发时减少这些限制。程序员学会新的语言结构后，能够提升软件开发时思维过程的层次。

  可能有人认为，了解其他语言的功能对一个被要求使用不具有这一功能的语言进行开发的程序员没有帮助。然而这种看法并不成立，因为通常来说，语言的结构能够被不直接支持这些结构的其他语言模拟出来。例如，一个了解 Perl 语言(Wall 等，2000)中关联数组的结构和用法的 C 语言程序员，可能会用 C 语言设计出模拟关联数组的结构。换句话说，对程序设计语言概念的学习能够使程序员对重要的语言特性与结构有充分的理解，鼓励程序员去使用它们，甚至在所用的语言不直接支持这种特性或结构的情况下也同样如此。

- **扩充选择合适语言的背景知识**。许多专业的程序员没有受过多少正规的计算机科学教育，而是通过自学或单位内部培训获得了程序设计技能。这类培训通常只教授与公司当前项目直接相关的一两种语言。其他许多程序员在很久以前受过正规的培训，他们那时所学的语言已经不再使用，现在程序设计语言中许多特性当时知道的人并不多。结果，许多程序员在为新项目选择程序设计语言时，仍然使用他们最熟悉的语言，即使该语言并不适合当前的项目。如果这些程序员熟悉更多的语言和语言结构，就能更好地选择具有适合处理手头问题的特性的语言。

  一种语言的某些特性经常可以用另一种语言来模拟，但是使用语言中专门设

计加入的特性总是优于模拟这一特性，用不支持该特性的语言来模拟通常不够简练，更加笨重，也不安全。

- **增强学习新语言的能力**。计算机程序设计仍然是一门相对年轻的学科，设计方法学、软件开发工具和程序设计语言仍然在不断地演进中。因此软件开发是一种激动人心的职业，但也意味着从业人员必须不断学习。学习新程序设计语言的过程可能艰难又漫长，对于只习惯于一两种语言、从来没有研究过程序设计语言基本原理的人来说尤其如此。一旦完全理解了语言的基础原理，就能够非常容易地理解这些原理如何融入所学语言的设计中。例如，理解面向对象程序设计原理的程序员比没有接触过这些原理的程序员更容易学习 Java 语言(Arnold 等，2006)。

  同样的现象也出现在自然语言方面。对母语语法理解得越好，学习另一种自然语言就越容易。此外，学习另一种语言也有助于更好地了解母语。

  TIOBE 程序设计社区发布了一个程序设计语言使用广泛程度的列表(http://www.tiobe.com/tiobe_index/index.htm)。例如，根据 2011 年 8 月的列表，Java、C 和 C++是使用最多的语言，还有几十种其他语言也得到广泛使用[1]。列表数据也表明，程序设计语言使用情况的分布一直在变化。正在使用的语言数量，以及统计数据的变化情况，都暗示每个软件开发人员必须准备着学习不同的语言。

  最后一点也很重要，在职的程序员只有了解程序设计语言的术语和基本概念，才能阅读和理解对程序设计语言的说明和评价，以及关于语言和编译器的其他文献。这些都是选择和学习语言的信息来源。

- **更深刻地理解实现的重要性**。在学习程序设计语言的原理时，学习影响这些原理的实现问题也是很有趣、很必要的。有时，程序员理解了实现问题，也就理解了为什么语言会设计成现在的样子，能够更聪明地按语言设计的方式去使用语言。只有理解了程序设计语言结构上的不同选择，以及不同选择的后果，才能成为更出色的程序员。

  只有了解相关实现细节的程序员才能发现和修正某些程序缺陷。理解实现问题的另一个好处是使我们能够想象出计算机如何执行不同语言的结构。有时，了解实现问题能够提示为程序选择的候选结构的不同效率。例如，如果程序员不知道实现子程序调用的复杂性，通常就意识不到，频繁调用一个小子程序是极其低效的设计。

  由于本书只涉及一部分实现问题，上述两段内容也解释了为什么要学习编译器设计。

- **更好地使用已学的语言**。许多现有的程序设计语言庞大且复杂，因此程序员通常不会熟悉和使用所用语言的所有特性。通过学习程序设计语言的原理，程序员能够了解所用语言中以前不知道、没用过的部分，并开始使用这些特性。

---

1. 注意，这个列表只考虑了程序设计语言的流行性，其准确性并未得到普遍的认可。

- **促进计算机信息处理技术的整体发展**。最后，从计算机信息处理技术的整体来看，也有必要学习程序设计语言的原理。虽然通常能够确定为什么某种程序设计语言会非常流行，但许多人认为，至少回顾过去，最流行的语言并不一定是当时最好的语言。有时一种语言得到广泛的使用，部分原因是选择语言的人对程序设计语言的原理不够熟悉。

  例如，许多人认为，如果 ALGOL 60 语言(Backus 等，1963)在 20 世纪 60 年代初期能替代 Fortran 语言(Metcalf 等，2004)，情况会好得多，因为它比 Fortran 更优雅，控制语句也好得多。事实并非如此，这部分归因于当时的程序员和软件开发经理，他们大多没有清楚地理解 ALGOL 60 的设计原理。他们认为该语言的说明难以阅读(确实是)，更难以理解。他们领悟不到模块结构、递归和结构良好的控制语句的好处，因此看不到 ALGOL 60 相对于 Fortran 的优越性。

  当然，如第 2 章所述，还有其他许多因素导致 ALGOL 60 语言没有得到认可。但是，计算机使用者没有普遍认识到该语言的好处，是一个重要的原因。

  一般来讲，如果选择语言的人能得到更多的信息，可能较好的语言最终会替代较差的语言。

1.2 程序设计领域

计算机已应用于大量不同领域，从控制核电站到在手机上提供视频游戏。由于计算机应用的巨大差异，人们开发了用于不同目的的程序设计语言。本节简要介绍一些计算机应用领域及其相关的语言。

1.2.1 科学应用

第一台数字计算机产生于 20 世纪 40 年代晚期、50 年代早期，是为了科学应用而发明和使用的。那时的科学应用一般使用较为简单的数据结构，但需要大量的浮点算术运算。最常用的数据结构是数组和矩阵，最常用的控制结构是计数循环和选择。为了科学应用而发明的早期高级程序设计语言，就是应这些需要而设计的，其竞争对手是汇编语言，因此效率是主要的考虑因素。第一种用于科学应用的语言是 Fortran。ALGOL 60 以及大多数的衍生语言也都用于这一领域，但它们也能用于其他相关领域。对于那些将效率视为关键因素的科学应用(例如 20 世纪 50 年代和 60 年代常见的应用)来说，后来的任何语言都没有明显好于 Fortran，这就是为什么 Fortran 至今仍在使用的原因。

1.2.2 商务应用

计算机用于商务应用始于 20 世纪 50 年代。人们为这一目的开发了专用的计算机和专用的语言。第一个成功用于商务的高级语言是 COBOL(ISO/ IEC，2002)，它的最初版本出现于 1960 年，至今仍是这些应用中用得最多的语言。商务语言的特点是：有生成详尽报告

的工具，能够精确说明和存储十进制数和字符数据，能够指定十进制运算操作。

除了 COBOL 语言的发展和演进之外，商务应用语言的开发很少，因此本书只对 COBOL 语言的结构进行有限的介绍。

1.2.3　人工智能

人工智能(AI)是计算机应用中一个广泛的领域，其特点是使用符号计算而不是数值计算。符号计算意味着处理由名字而非数字组成的符号，而且符号计算用数据链表比用数组更方便。这类程序设计有时比其他程序设计领域需要更大的灵活性。例如，在一些人工智能应用中，具有在运行时创建和执行代码段的能力会很方便。

第一种为人工智能应用而开发、并广泛使用的程序设计语言是函数式语言 LISP (McCarthy 等，1965)，它产生于 1959 年。1990 年之前开发的大多数人工智能应用程序，都是用 LISP 语言或它的近似语言写成的。但在 20 世纪 70 年代初期，一些人工智能应用有了另一种解决方案——使用 Prolog 语言(Clocksin 和 Mellish，2003)的逻辑程序设计。最近，一些人工智能应用程序也用系统语言，如 C 语言来实现。第 15 章和第 16 章将分别介绍 Scheme 语言(LISP 语言的一种方言；Dybvig，2003)和 Prolog 语言。

1.2.4　系统程序设计

计算机系统中的操作系统和程序设计支持工具统称为系统软件。系统软件几乎总是不停地运行着，因此它必须高效，而且它必须有一些低层次的特性，以允许软件与要执行写操作的外部设备联系。

在 20 世纪 60 年代和 70 年代，一些计算机制造商，例如 IBM、Digital 和 Burroughs(现在的 UNISYS)，为它们计算机上的系统软件开发了专用的面向计算机的高级语言。用于 IBM 大型机的语言是 PL/I 的一种方言 PL/S；用于 Digital 计算机的是 BLISS 语言，它是一种只比汇编语言高一级的语言；用于 Burroughs 计算机的是 Extended ALGOL 语言。但是现在，大多数系统软件都用更常用的程序设计语言来编写，如 C 和 C++。

UNIX 操作系统几乎完全是用 C 语言(1SO，1999)编写的，因此该系统更容易移植到不同的计算机上。C 语言的一些特性使其成为系统程序设计的一个不错的选择，它是低级语言，执行效率高，对使用者没有很多安全方面的限制。系统程序员通常非常优秀，他们认为自己不需要这些限制。然而一些非系统程序员认为，使用 C 语言来开发大规模的重要的软件系统太危险了。

1.2.5　网络软件

万维网(World Wide Web)由一些相互兼容的语言支持，包括非程序设计语言的标记语言(如 HTML 语言)和通用的程序设计语言(如 Java 语言)。由于网页普遍需要动态的网络内容，在内容展示的技术中常常需要一定的计算能力，这一功能可以通过在 HTML 文档中嵌入程序设计代码来实现。这类代码通常采用脚本语言的形式，例如 JavaScript 或 PHP。也

有一些类似标记语言的语言进行了扩展，包括控制文档处理的结构，参见 1.5 节和第 2 章。

1.3 语言评价标准

如前所述，本书旨在仔细研究各种程序设计语言的结构和功能原理，并评价这些特性，将焦点放在它们对软件开发与维护过程的影响上。为此，需要一套评价标准。这套标准不可避免会存在争议，因为即使只有两名计算机科学家，也几乎不可能在某种语言特性相对于其他语言特性的打分上取得一致。尽管存在这些差异，但大多数人都赞同：下面小节中介绍的这些标准很重要。

一些语言特性会影响最重要的 4 项标准中的 3 项标准，如表 1-1 所示。这些标准在后面的小节中介绍[2]。请注意，表中只列出了最重要的语言特性，对应于后续几小节的介绍。如果考虑其他不太重要的语言特性，则几乎所有的表项都打上了点。

注意，这些标准有的是广义的，定义得不够明确(例如可写性)，有的是特定的语言结构(如异常处理)。虽然下面的介绍可能暗示这些标准同等重要，但是我们并不是这个意思，而且显然并非如此。

表 1-1 语言评价标准和影响它们的语言特性

	标 准		
特性	可 读 性	可 写 性	可 靠 性
简单性	•	•	•
正交性	•	•	•
数据类型	•	•	•
语法设计	•	•	•
对抽象的支持		•	•
表达		•	•
类型检查			•
异常处理			•
有限地使用别名			•

1.3.1 可读性

判断程序设计语言的最重要的标准之一就是写出的程序是否便于阅读和理解。在 1970 年之前，人们认为软件开发主要就是编写代码，程序设计语言最重要的优点是高效。语言结构的设计更多是从计算机的角度，而不是计算机用户的角度来考虑。然而在 20 世纪 70

2. 第 4 项重要标准是成本，它没有包含在表格中，因为它与其他 3 项标准以及影响标准的语言特性的关联度较低。

年代，出现了软件生命周期(Booch，1987)的概念，认为写代码只涉及很少的部分，维护才是周期中最主要的部分，特别是从成本角度来看。因为易维护性主要由程序的可读性决定，所以可读性成为衡量程序质量和程序设计语言的一个重要指标。这是程序设计语言演进过程中的一个重要转折点，关注焦点明显地从面向机器转变为面向人。

可读性必须放在问题领域中考虑。例如，如果一段说明计算过程的程序，不是用为说明计算而设计的语言写成的，程序表达就会不自然地绕弯子，读起来特别困难。

下面几小节介绍影响程序设计语言可读性的特性。

1.3.1.1　整体简单性

程序设计语言的整体简单性极大地影响着它的可读性。有大量基本结构的语言比结构少的语言学起来更难。必须使用大规模语言的程序员，通常只学该语言的一个子集，而忽略语言的其他特性。这种学习模式有时用来解释为什么设计那么多语言结构，但这种说法没有道理，只要编写程序的人所学的子集与读程序的人所熟悉的子集不一样，就会出现可读性问题。

程序设计语言的第二个复杂性是特性多样性，即能够用不止一种方法来实现特定操作。例如，在 Java 中，程序员能够用 4 种方法给一个简单的整数变量加上 1。

```
count = count + 1
count += 1
count++
++count
```

虽然在某些上下文中，最后两个语句之间、它们与其他语句之间的意义有微小的区别，但用作单独的表达式时，4 条语句的意义完全一样。这些差异将在第 7 章介绍。

第三个潜在的问题是运算符重载，即一个运算符有多种意义。虽然这很有用，但是如果允许使用者创建自己的重载，并且不是按常规重载，会降低可读性。例如，重载“+”用于整数和浮点数的加法显然是可取的。事实上，这一重载减少了运算符的数目，从而简化了语言。但是假设程序员将“+”定义为计算两个一维数组中所有元素的和，由于它不同于通常的向量求和，因此对这个程序，编写程序的人与读程序的人都难以理解。使程序难以理解的一个更极端的例子是将“+”定义为计算两个向量的第一个元素之差。第 7 章将深入介绍运算符重载。

当然，语言的简单性也不能过分。例如，如下一节所示，大多数汇编语言语句的形式与意义都是简单性的典型体现。但是这种简单性使汇编语言程序的可读性较差。由于汇编语言程序缺乏更复杂的控制语句，所以程序结构不清晰；由于语句简单，所以与高级语言相比，同样的程序需要的语句就多得多。如果高级语言的控制结构和数据结构不适当，虽然没有上面的例子那样极端，但结论是一样的。

1.3.1.2　正交性

程序设计语言的正交性意味着相对较小的基本结构集合，能够以较少的组合方式来构

成语言的控制结构和数据结构。而且，基本结构的任一种可能组合都是合法且有意义的。例如数据类型，假设某语言有 4 种基本的数据类型(整型、单精度型、双精度型和字符型)和 2 种运算符(数组和指针)。如果 2 种运算符都能够作用于自身和 4 种基本的数据类型，就能够定义大量的数据结构。

一个语言特性是正交的，意味着它独立于程序中出现位置的上下文(“正交”来自于正交向量这一数学概念，正交向量相互独立)。正交性来源于基本结构间的对称关系。语言缺乏正交性将导致语言规则的特例。例如，在支持指针的程序设计语言中，应允许定义指针指向该语言定义的任何特定类型。但是，如果指针不允许指向数组，就无法定义许多潜在有用的用户自定义数据结构。

下面比较 IBM 大型机与 VAX 系列微型计算机的汇编语言的一个方面，来说明正交性在设计原理上的应用。考虑一个简单的情形：将两个内存或寄存器中的 32 位整数值相加，所得的和替换其中的一个值。IBM 大型机用两条指令来实现这一操作，如

```
A  Reg1, memory_cell
AR Reg1, Reg2
```

其中 Regl 和 Reg2 代表寄存器。指令的语义是

```
Reg1 ← contents(Reg1) + contents(memory_cell)
Reg1 ← contents(Reg1) + contents(Reg2)
```

在 VAX 中，32 位整数值的加法指令是

```
ADDL operand_1, operand_2
```

其语义是

```
operand_2 ← contents(operand_1) + contents(operand_2)
```

其中，任一操作数都可以是寄存器或存储单元。

VAX 的指令设计是正交的，因为一条指令既可以使用寄存器，也可以使用存储单元作为操作数。有两种方法指定操作数，而且能够以任意形式组合。IBM 的设计不是正交的，在 4 种可能的操作数组合中，只有两种组合是合法的，而且这两种组合需要不同的指令 A 和 AR。IBM 的设计限制较多，因此可写性较差，例如，无法将两个值相加，并将结果存储在内存中。此外，由于上述限制以及额外的指令，IBM 的设计学习起来也更难。

正交性与简单性直接相关：语言设计越正交，语言规则需要的特例就越少，特例越少意味着设计的规范程度越高，语言就越容易学习、阅读和理解。任何学过大量英语的人可以证实，英语中的许多规则特例太难学了(例如，i 总是在 e 前，除非在 c 后)。

作为高级语言中缺乏正交性的一个例子，考虑 C 语言中的以下规则和特例。虽然 C 语言有两种结构化的数据类型——数组与结构(struct)，但结构可从函数返回，而数组不行。结构的成员可以是除 void 和同样结构以外的任意数据类型，而数组元素可以是除 void 和函数以外的任意数据类型。函数参数是按值传递的，除非参数是数组，这时实际上是按引

用传递的(因为在 C 程序中，不带下标的数组名会解释为数组第一个元素的地址)。

作为上下文相关的一个例子，考虑以下 C 语言的表达式：

```
a + b
```

这个表达式通常表示取出 a 和 b 的值并相加，但是如果 a 恰好是一个指针，则会影响 b 的值。例如，如果 a 指向一个占 4 字节的浮点值，那么 b 在与 a 相加之前，必须放大它的值——在这个例子中要乘以 4。因此，a 的类型会影响对 b 值的处理，即 b 的上下文会影响 b 的意义。

过多的正交也会产生问题。ALGOL 68 语言(van Wijngaarden 等，1969)可能是最具有正交性的程序设计语言了。ALGOL 68 中的每一语言结构都有一个类型，这些类型没有任何限制，而且大多数结构都有值。这种自由组合可以产生极其复杂的结构，例如，只要结果是一个地址，就可以把条件语句、声明语句及其他各类语句一起放在赋值运算符的左边。这种极端的正交形式导致了不必要的复杂性。而且，由于语言需要大量的基本结构，高度的正交性将产生爆炸性的组合方式。因此，即使组合方式很简单，它们的整体数量也会导致语言的复杂性。

可见，语言的简单性至少部分归因于相对少量的基本结构的组合，以及正交原理的有限应用。

有人认为，函数式语言同时具有良好的简单性和正交性。函数式语言，如 LISP，主要通过将函数作用于给定参数来执行计算。相反，命令式语言，如 C、C++和 Java，通常用变量和赋值语句来指定如何计算。函数式语言提供了最佳的整体简单性，因为它们能够用一种结构，即函数调用(函数调用能够以简单的方式与其他函数调用组合起来)来完成任何计算。正是这种简单优美使一些语言研究者将函数式语言作为复杂的非函数式语言(如 C++)的主要替代语言。但其他因素，如效率，限制了函数式语言的更广泛应用。

1.3.1.3　数据类型

在语言中是否有适当的方式来定义数据类型和数据结构，是另一个有助于可读性的重要因素。例如，假设由于语言中没有布尔类型，只能用数值类型来作为指示标志，那么在这一语言中，我们会使用下面的赋值语句

```
timeOut = 1
```

这一语句的意义不明确。然而在有布尔类型的语言中，我们会用下面的语句

```
timeOut = true
```

这一语句的意义相当明确。

1.3.1.4　语法设计

语言元素的语法(或形式)对程序的可读性有重要的影响。以下是不同的语法设计影响

可读性的两个例子。

- **特殊字**。程序的外观和可读性很大程度上受语言特殊字(如 while、class 和 for)的影响，特别是在控制结构中构成复合语句(即语句组)的方式。有些语言采用配对的特殊字或符号来构成语句组。C 语言和它的后继语言采用括号来表示复合语句。所有这些语言都很麻烦，因为语句组都是以同样的方式结束，所以当出现 end 或}时，很难确定结束的是哪一个语句组。Fortran 95 和 Ada 语言对每一种语句组都采用不同的结束语法，就比较清楚。例如，Ada 语言用 end if 来结束选择结构，用 end loop 来结束循环结构。这是用更少的保留字保证简单性(如 C++语言)与采用更多的保留字保证可读性(如 Ada 语言)相矛盾的一个例子。

  另一个重要的问题是语言的特殊字是否可以用做程序变量的名称。如果可以，写出来的程序会令人费解。例如在 Fortran 95 语言中，特殊字如 Do 和 End 是合法的变量名称，因此在程序中出现这些字可能有特殊的含意，也可能没有。

- **形式与意义**。设计语句时，使其在表现形式上至少能够部分表明语句的功能，这无疑有助于提高可读性。语法或形式应该直接体现语义或意义，有时两种语言结构看起来相同或相似，但可能由于不同的上下文，具有不同的意义，就违背了这一原则。例如，在 C 语言中，保留字 static 的意义取决于所出现的位置。如果用于在函数中定义变量，则其意义是变量在编译时创建；如果用于在所有函数之外定义变量，则其意义是变量只在定义它的文件内可见，即它不会从这个文件中输出。

  对 UNIX(Raymond，2004)外壳命令的一个主要抱怨是：通过它们的形式并不总能看出它们的功能。例如，只有具备先验知识(或者够聪明)，并且很熟悉 UNIX 的编辑器 ed，才能猜出 UNIX 命令 grep 的意义。grep 的形式对于 UNIX 初学者来说看不出什么东西(在 ed 中，命令/*regular_expression*/能够搜索与正则表达式匹配的子串。在命令前加 g 使它成为一个全局命令，表明搜索的范围是所编辑的整个文件。在命令后加 p 表明要打印匹配子串所在的行。*g/regular_expression/p* 显然可以缩写为 grep，因此会打印文件中所有包含了匹配正则表达式的子串的行)。

1.3.2 可写性

可写性是衡量在指定问题领域内用语言创建程序有多容易的一个标准。大部分影响可读性的语言特性也影响可写性，这从以下事实可以直接得出结论：写程序的过程需要程序员经常回头阅读已经写好的程序部分。

与可读性一样，可写性也必须在语言针对的问题领域中考虑。在特定的应用领域中比较两种语言的可写性，其中一个是为该应用设计的，另一个不是，这显然不合理。例如，对于创建拥有图形用户界面的程序来说，Visual BASIC(VB)语言与 C 语言的可写性差别巨大，对于这一任务，VB 语言比较理想。而编写系统程序，例如操作系统时，它们的可写性同样差别巨大，C 语言是为这一目的设计的。

下面几小节介绍了影响语言可写性的一些非常重要的特性。

1.3.2.1　简单性和正交性

如果语言有大量不同的结构，有些程序员就可能不熟悉所有的结构。这会导致程序员误用某些特性，而不使用其他更优美更高效的特性。如 Hoare(1973)所言，偶尔使用未知的特性很可能会导致意外的结果。因此，较少量的基本结构和一套组合基本结构的一致规则(即正交性)，比简单地提供大量基本结构好得多。程序员只需学习一些简单的基本结构，就能够设计出解决复杂问题的方案。

另一方面，过多的正交有损于可写性。当几乎任何基本结构的组合都合理时，程序中的错误会难以检测，从而导致编译器无法发现代码中的谬误。

1.3.2.2　对抽象的支持

简而言之，抽象就是以允许忽略许多细节的方式来定义和使用复杂结构和操作的能力。抽象是当代程序设计语言设计的关键概念，这是抽象在现代程序设计方法学中扮演中心角色的反映。因此，程序设计语言所允许的抽象程度以及表达抽象的自然程度，对语言的可写性非常重要。程序设计语言支持两类不同的抽象：过程抽象和数据抽象。

过程抽象的一个简单例子，是使用子程序来实现程序中多处需要的排序算法。如果没有子程序，排序的代码将不得不重复出现在所有需要它的地方，这会使程序写起来冗长而乏味。也许更重要的是，如果不使用子程序，使用排序子程序的代码会与排序算法的细节混在一起，严重地掩盖代码流程和整体意图。

作为数据抽象的一个例子，请考虑一棵在其结点上存储整数的二叉树。用不支持指针并且用堆进行动态内存管理的语言(例如 Fortran 77 语言)来实现这样的二叉树，通常采用 3 个平行整数数组，其中 2 个整数用于指定子结点的下标。用 C++和 Java 的话，可以把 2 个指针(或引用)和 1 个整数组成一个简单的类，作为树结点的抽象来实现这棵树。后一种表示形式比较自然，用这两种语言编写使用二叉树的程序，要比用 Fortran 77 来写容易得多，这仅仅因为语言解决问题的方式更接近于问题的领域。

对抽象的整体支持程度显然是语言可写性的一个重要因素。

1.3.2.3　表达性

语言的表达性涉及若干不同特性。在类似 APL(Gilman 和 Rose，1976)的语言中，表达性意味着有若干功能强大的运算符，能够用很短的程序实现大量的运算。表达性更普遍的意义是，语言具有较为简捷而不笨重的方式来指定运算过程。例如，在 C 语言中，count++的形式比 count=count+1 更简短更方便。还有，Ada 语言中的布尔运算符 and then 方便地表明对布尔表达式进行短路求值，Java 语言中的 for 语句在写计数循环时比使用 while 语句更方便，虽然后者也能实现。以上这些都提高了语言的可写性。

1.3.3　可靠性

如果程序在任何情况下都能按设计的那样执行，就称该程序是可靠的。下面几小节介绍对特定语言程序的可靠性有重大影响的若干语言特性。

1.3.3.1 类型检查

类型检查是指对所给程序中的类型错误进行简单的检测，可以由编译器来做，也可以在程序运行时做。类型检查是语言可靠性的一个重要因素。由于运行时的类型检查代价很高，编译时的类型检查就更可取，而且程序中的错误越早发现，修正错误的代价也越小。Java 语言的设计要求在编译时对几乎所有的变量和表达式进行类型检查，这基本上杜绝了 Java 程序在运行时的类型错误。第 6 章将深入介绍类型和类型检查。

下面是最初版本 C 语言(Kernighan 和 Ritchie，1978)的一个例子，由于无法在编译期间或运行期间进行类型检查，使用子程序参数导致了无数的程序错误。在这种语言中，没有检查函数调用中实参的类型，来确定该类型是否匹配函数中对应形参的类型。对于将 float 作为形参类型的函数，可以把 int 类型的变量用作其实参，编译器和运行时系统检测不出这个不一致性。例如，因为表示整数 23 的位串实际上与表示浮点数 23 的位串不相关，如果把整数 23 发送给需要浮点数参数的函数，则该函数使用参数的任何操作都不会生成有意义的结果。而且，这种程序常常很难诊断[3]。C 的当前版本要求检查所有参数的类型，从而避免了这个问题。第 9 章将讨论子程序和参数传递技术。

1.3.3.2 异常处理

程序拦截运行时错误(以及程序可检测出来的其他不寻常的条件)、采取更正措施，再继续运行的能力显然有助于提高程序的可靠性。这个语言特性称为异常处理。Ada、C++、Java 和 C#都包含异常处理的强大功能，但这个特性在许多广泛使用的语言中并不存在，包括 C 和 Fortran。异常处理在第 14 章讨论。

1.3.3.3 别名

别名的定义不太严格，它表示使用两个或多个不同的名称都可以访问同一个内存单元。人们普遍认为，别名是程序设计语言中的一个危险特性。大多数程序设计语言都允许执行某种别名操作，例如，在大多数语言中都可以设置两个指向同一个变量的指针。在这种程序中，程序员必须总是记得，改变一个指针指向的值，另一个指针引用的值也会改变。语言的设计可以禁用第 5 章和第 9 章介绍的某些别名功能。在一些语言中，别名用于克服该语言中数据抽象方面的某些缺陷，其他语言则大大限制别名的使用，以提高其可靠性。

1.3.3.4 可读性和可写性

可读性和可写性都会影响可靠性。如果用一种语言编写的程序不支持表达所需算法的自然方式，就必须使用不自然的方式来表达。不自然的方式在所有可能的情形中都表达正确的可能性不大。程序越容易编写，其出错的可能性就越小。

可读性还在程序生命周期的编写和维护阶段影响可靠性。难以阅读的程序，也难以编写和修改。

---

3. 为了解决这个问题和其他类似的问题，UNIX 系统包含一个实用程序 lint，来检查 C 程序中的此类问题。

1.3.4　成本

程序设计语言的总成本是其许多语言特性的函数。

第一，培训程序员使用语言是有成本的，它是语言的简单性和正交性，以及程序员的经验的函数。语言越强大，学习起来常常就越困难，尽管并不总是很困难。

第二，用语言编写程序是有成本的，它是语言的可写性的函数，部分取决于它与具体应用的接近程度。设计和实现高级语言的初始动力来源于降低软件开发成本的愿望。

培训程序员的成本和用语言编写程序的成本会在优秀的程序设计环境中大大降低。程序设计环境参见 1.8 节。

第三，编译程序是有成本的。早期使用 Ada 的一个主要障碍是运行第一代 Ada 编译器的成本过高。Ada 编译器改进后，这个问题就消失了。

第四，执行用语言编写的程序的成本受到该语言设计的显著影响。需要在运行期间进行许多类型检查的语言，无论编译器的质量如何，执行代码的速度都不快。尽管运行效率是设计早期语言时要考虑的最重要的因素，但现在它已经没有那么重要了。

编译成本和已编译代码的执行速度可以进行一个简单的折中。编译器可用于降低代码的规模、提高已编译代码的执行速度的技术集称为优化。与耗费大量精力来生成优化的代码相比，完全没有或几乎没有进行优化，编译器会执行得快得多。是否进行优化的选择受到使用编译器的环境的影响。在实验室中，开始学习程序设计的学生在开发过程中，常常编译程序许多次，但在运行期间使用很少的代码(他们的程序很小，且代码只需要正确运行一次)，此时就不应进行什么优化。而在生产环境中，编译好的程序在开发完成后会运行许多次，最好花额外的精力优化代码。

影响语言成本的第五个因素是语言实现系统的成本。Java 很快得到认可的一个因素是，在发布其设计后不久，其编译器/解释器系统就可以免费获得。如果一种语言的实现系统很昂贵，或者只能运行在昂贵的硬件上，那么这种语言获得广泛使用的可能性就会小很多。例如，第一代 Ada 编译器的高成本就阻碍了早期 Ada 的流行。

第六，可靠性很差也是有代价的。如果软件在重要的系统(例如核电站或医用 X 光机)出现故障，其代价就会非常高。软件在不那么重要的系统上出现故障，其代价也会很高，可能失去未来的生意，或因为有缺陷的软件系统而被提起诉讼。

最后一个考虑因素是维护程序的成本，包括更正错误和修改程序，以增加新功能。软件维护成本取决于许多语言特性，主要是可靠性。因为维护工作常常不是由软件最初的作者来完成，所以糟糕的可靠性可能使该任务极难完成。

软件维护的重要性无论怎样强调都不过分。据估计，生命周期相对较长的大型软件系统，其维护成本可能是开发成本的 2～4 倍(Sommerville，2005)。

在影响语言成本的所有因素中，3 个因素最重要：程序开发、维护和可靠性。因为它们都是可写性和可读性的函数，所以这两个评价标准是最重要的。

当然，还可以使用许多其他标准来评价程序设计语言。一个标准是可移植性，即程序可以从一种实现方案转为另一种实现方案的便利性。可移植性主要受到语言标准化程度的影响。一些语言，例如 BASIC，根本就没有标准化，用这些语言编写的程序很难从一种实

现方案转为另一种实现方案。标准化是一个耗时而困难的过程。委员会从 1989 年开始开发 C++的标准版本，在 1998 年得到批准。

普遍性(可应用于许多不同的应用程序)和定义良好性(语言的官方定义文档非常完整、准确)是另外两个标准。

大多数标准，尤其是可读性、可写性和可靠性，没有准确的定义，也无法精确地衡量。尽管如此，它们仍是有效的概念，为程序设计语言的设计和评价提供了有价值的参考。

评价标准最后要注意的是，从不同的角度来看，语言设计标准有不同的加权。语言实现人员主要关心的是实现语言结构和特性的难度。语言的使用者则首先考虑可写性，然后考虑可读性。语言设计人员可能强调语言的优雅和获得广泛应用的能力。这些语言特性通常是相互矛盾的。

1.4　影响语言设计的因素

除了 1.3 节介绍的那些因素外，其他一些因素也会影响程序设计语言的基本设计，其中最重要的是计算机体系结构和程序设计方法学。

1.4.1　计算机体系结构

计算机基本的体系结构对语言设计有着深远的影响。过去 50 年中大多数的流行语言都是围绕常见的计算机体系结构来设计的，这一结构用它的创始人 John von Neumann(发音为 von Noyman)的名字命名为冯・诺依曼结构。这些语言称为命令式语言。在冯・诺依曼计算机中，数据和程序都存储在同一内存中，执行指令的中央处理器(CPU)与内存是分开的。因此，指令和数据必须从内存传输到 CPU，CPU 的运算结果必须传回到内存。自 20 世纪 40 年代以来，几乎所有的数字计算机都基于冯・诺依曼结构。冯・诺依曼计算机的整体结构如图 1-1 所示。

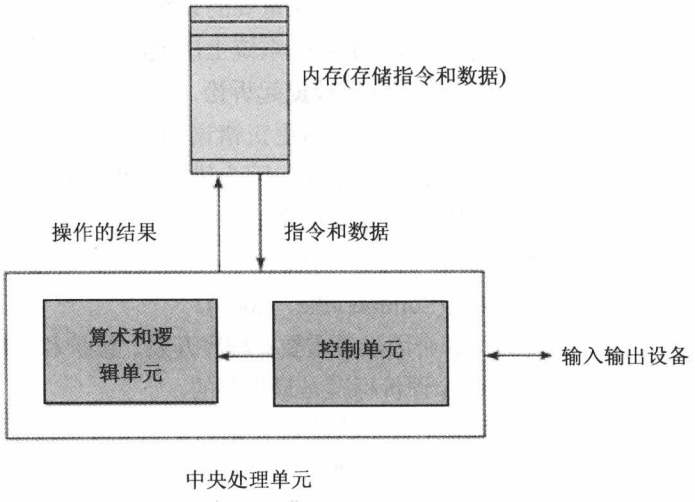


图 1-1　冯・诺依曼计算机体系结构

由于冯·诺依曼结构，命令式语言的核心特征就是变量(内存单元的模型)、赋值语句(基于传输操作)和迭代形式的循环运算(这是在该体系结构上实现循环的最高效形式)。表达式中的操作数由内存传输到 CPU，表达式的计算结果传回到赋值语句左边所代表的内存单元。迭代在冯·诺依曼计算机上运算很快，因为指令存储在内存中相邻的单元内，重复执行一段代码只需一个简单的分支语句。由于迭代的高效性，不鼓励使用递归来实现循环，虽然有时用递归实现更自然。

在冯·诺依曼结构的计算机上执行机器码程序，是一个被称为读取-执行周期的过程。如前所述，程序存储在内存中，但是在 CPU 中执行，每一条要执行的指令必须从内存传输到处理器中。要执行的下一条指令的地址保存在一个叫做程序计数器的寄存器中。读取-执行周期可以简单地用下面的算法来表示：

```
初始化程序计数器
repeat 永远
    获取程序计数器指向的指令
    程序计数器增 1，指向下一条指令
    指令解码
    执行指令
end repeat
```

算法中的“指令解码”步骤通过检查指令来确定指令所指的操作。当碰到停止指令时，程序的执行就中止了，尽管在实际的计算机中极少会执行停止指令。相反，程序的控制由操作系统传给用户程序来使其执行，当用户程序执行结束后再传回操作系统。在计算机系统中，如果有多个用户程序同时在内存中，这一过程就要复杂得多。

如前所述，函数式(或应用式)语言基本的运算方式是将函数作用于给定参数。用函数式语言进行程序设计可以不使用命令式语言中所用到的那种变量、赋值语言以及迭代。虽然许多计算机科学家已经阐述了函数式语言(如 Scheme 语言)的巨大优势，但是在设计出非冯·诺依曼计算机并使函数式语言的程序高效地执行之前，它们仍不可能替代命令式语言。在为这一事实感到遗憾的人中，最具有说服力的是 Fortran 语言最初版本的主要设计者 John Backus(1978)。

尽管事实上，命令式程序设计语言的结构是以计算机体系结构为模型的，而不是出于程序设计语言使用者的能力和意愿，有人仍然认为使用命令式语言比使用函数式语言在某种程度上更自然，因此许多人认为，即使函数式程序与命令式程序运行效率一样高，命令式程序设计语言的使用仍然会占据主导地位。

1.4.2　程序设计方法学

在 20 世纪 60 年代晚期和 70 年代早期，人们主要从结构化程序设计运动开始，对软件开发过程和程序设计语言的设计进行了深入的研究。

进行这一研究的一个重要原因是计算的成本从硬件变成了软件，这是因为硬件成本下

降而程序员人力成本提高，程序员生产力的提高却相对较小。此外，计算机正用于解决规模越来越大、越来越复杂的问题，不再像 20 世纪 60 年代初期那样只是简单地求解一组方程式来模拟卫星轨迹，人们开发程序来处理大型的复杂任务，例如控制大型炼油设备和提供全球航班订票系统。

20 世纪 70 年代的研究产生了新的软件开发方法学，称为自顶向下设计和逐步求精。人们发现程序设计语言的主要缺点是类型检查不完备，控制语句不恰当(需要大量使用 goto 语句)。

在 20 世纪 70 年代晚期，程序设计方法学开始由面向过程转变为面向数据。简单来说，面向数据的方法强调数据的设计，关注如何用抽象数据类型来解决问题。

为了在软件系统设计中有效地使用数据抽象，用来实现程序的语言必须支持数据抽象。最早对数据抽象提供有限支持的语言是 SIMULA 67(Birtwistle 等，1973)，虽然这一语言并没有因此而普及开来。数据抽象的好处直到 20 世纪 70 年代早期才得到广泛认可，不过自 20 世纪 70 年代晚期以来所设计的大多数语言都支持数据抽象。数据抽象将在第 11 章详细介绍。

面向数据的软件开发的最终演进开始于 20 世纪 80 年代早期，演变为面向对象的设计。面向对象的设计方法基于数据抽象，将数据处理和数据对象封装在一起，控制数据访问，加入了继承和动态方法绑定。继承是个强大的概念，大大提高了已有软件的重用可能性，从而有可能大大提高软件开发的生产力。这是面向对象的语言被广泛使用的一个重要原因。动态(运行时)方法绑定使得继承的使用更加灵活。

面向对象的程序设计是与支持这一概念的 Smalltalk 语言(Goldberg 和 Robson，1989)同时发展起来的。虽然 Smalltalk 语言从未像许多其他语言那样被广泛使用，但是支持面向对象的程序设计是现在大多数流行的命令式语言的一部分，如 Ada 95(ARM，1995)、Java、C++和 C#语言。面向对象的概念在 CLOS(Bobrow 等，1988)和 F# (Syme 等，2010)的函数式程序设计和 Prolog++语言(Moss，1994)的逻辑式程序设计中也得到了应用。语言对面向对象的程序设计的支持将在第 12 章中详细介绍。

面向过程的程序设计在某种意义上与面向数据的程序设计是相反的，虽然现在面向数据的方法在软件开发中占主导地位，但是面向过程的方法并没有被遗弃。相反，近年来对面向过程的程序设计有大量的研究，特别是在并发的领域。这些研究工作需要语言机制支持创建和控制并发的程序单元。Ada、Java 和 C#语言有这样的功能。并发将在第 13 章中详细介绍。

软件开发方法学的每一步演进都催生了新的语言结构来支持它们。

1.5 程序设计语言的分类

程序设计语言通常可以分为 4 类：命令式语言、函数式语言、逻辑语言和面向对象的语言。然而，我们并不把支持面向对象程序设计的语言单独归为一类。我们已经介绍了应用最广泛的支持面向对象程序设计的语言如何从命令式语言发展而来。虽然面向对象的软

件开发模式与命令式语言通常采用的面向过程的模式有很大区别，但是命令式语言为了支持面向对象的程序设计而扩展的部分并不多。例如，C 和 Java 语言中的表达式、赋值语句和控制语句几乎相同(另一方面，Java 语言的数组、子程序和语义与 C 语言中的大不一样)。对于支持面向对象程序设计的函数式语言来说，也有类似的结论。

另一类语言是可视化语言，是命令式语言的一个子类。应用最广泛的可视化语言是.NET。这些语言(或是它们的实现)能够以拖放的方式生成代码段。这些语言曾称为第四代语言，现在已经不再使用这一说法了。可视化语言为程序提供了生成图形用户界面的简单方式，例如，在 Visual Studio 中使用.NET 语言开发软件时，只需要敲一下键，就能够生成一段代码，显示一个表单控件，如按钮或文本框。现在所有的.NET 语言都具有这一功能。

有些人将脚本语言归为单独的一类程序设计语言。但是，把这一类语言归在一起更多的是出于它们的实现方式(部分或全部都是解释执行的)，而不是出于通常意义上的语言设计。这些语言通常称为脚本语言，其中 Perl、JavaScript 和 Ruby 语言在任何意义上都属于命令式语言。

逻辑程序设计语言是一种基于规则的语言。在命令式语言中，算法要有详细的说明，必须包含指令或语句的特定执行顺序。但是在基于规则的语言中，规则的定义没有固定的顺序，必须由语言实现系统来确定规则的使用顺序，产生期望的结果。这种软件开发的方法与用其他三类语言开发的方法完全不同，显然需要一种完全不同的语言。应用最广泛的逻辑程序设计语言 Prolog 和逻辑程序设计将在第 16 章中介绍。

近年来出现了一类新的语言——标记与程序设计混合语言。标记语言不是程序设计语言，例如使用最广泛的标记语言 HTML，用于定义网页文档中信息的布局。然而，在某些 HTML 语言和 XML 语言的扩展中，也加入了一些程序设计的功能，其中包括 JSP 标准标签库(Java Server Pages Standard Tag Library，JSTL)和可扩展样式表语言转换(eXtensible Stylesheet Language Transformations，XSLT)。这两种语言将在第 2 章中简要介绍。这类语言无法与任何一种计算机程序设计语言相比，因此在第 2 章后不再涉及它们。

在过去 50 年中，出现了一些特殊用途的语言，例如用于生成商务报表的报表程序生成器(Report Program Generator，RPG)，用于指示可编程机器工具的自动程控工具(Automatically Programmed Tools，APT)，用于系统模拟的通用模拟系统(General Purpose Simulation System，GPSS)。本书不讨论特殊用途的语言，主要是因为它们应用领域狭窄，难以与其他语言进行比较。

1.6　语言设计中的权衡

1.3 节介绍的程序设计语言的评价标准为语言设计提供了一个框架。可惜这个框架是自相矛盾的。Hoare 在其关于语言设计的见解深刻的论文(Hoare，1973)中指出，有太多重要

但相互矛盾的标准，协调并满足这些标准是一个重要的工程任务。

两个相矛盾的标准是可靠性和执行成本。例如，Java 语言的定义要求对数组元素的所有引用都进行检查，以确保数组下标在合法的范围内。这一步骤对于有大量数组元素引用的 Java 程序来说，极大地增加了执行成本。C 语言不需要检查下标的范围，所以 C 程序比语义上相同的 Java 程序执行得快，虽然 Java 程序更可靠。Java 的设计者以执行效率为代价换取了可靠性。

标准相互矛盾直接导致语言设计权衡的另一个例子是 APL 语言。APL 语言包含一组强大的数组运算符。由于运算符数目众多，APL 语言不得不引入大量新的符号来表示这些运算符，许多 APL 运算符也可以用于单个又长又复杂的表达式。高度复杂性的一个结果就是，对于涉及许多数组操作的应用程序来说，APL 语言的可写性非常好，大量的计算实际只需很小一段程序就可说明。而另一个结果就是，APL 程序的可读性很差，紧凑而简练的表达式具有一定的数学美感，但是对于编程者以外的人来说难以理解。著名程序员 Daniel McCracken 曾提到，他花了 4 个小时来阅读和理解一个 4 行的 APL 程序(Daniel McCracken，1970)。APL 语言的设计者以可读性为代价换取了可写性。

可写性和可读性间的矛盾在语言设计中是很常见的。C++语言中的指针可以以多种方式来操作，支持极其灵活的数据寻址。由于指针潜在的可靠性问题，Java 语言中没有引入指针。

语言设计(和评价)标准之间的矛盾大量存在，有些很小，有些则很明显。显而易见，在设计程序设计语言时，选择结构和特性需要许多折中和权衡。

1.7 实现方法

如 1.4.1 节所述，内存和处理器是计算机的两个基本组成部件，内存用于存储程序和数据，处理器是一组电路，提供一系列基本运算或机器指令的实现，例如，算术运算和逻辑运算。在大多数计算机中，有些指令(有时也称为宏指令)实际上是由一些称为微指令的指令实现的，微指令在一个更低的层次上定义，软件觉察不到，所以此处不做深入介绍。

计算机的机器语言是一组指令，在没有其他支持软件的情况下，机器语言是唯一能被大多数硬件计算机“理解”的语言。理论上，能够用某种高级语言作为机器语言来设计和建造计算机，但非常复杂和昂贵，而且极不灵活，因为很难使用其他高级语言(虽然不是不可能)来操作。计算机设计中比较现实的选择是在硬件中实现一种非常低级的语言，提供最普遍需要的基本操作，由系统软件来建立与高级语言程序的接口。

语言实现系统不是计算机中唯一的软件，还需要称为“操作系统”的一个大程序集合来提供比机器语言高级的基本操作。这些基本操作提供系统资源管理、输入和输出操作、

文件管理系统、文本和程序编辑器，以及其他各种常用的功能。由于语言实现系统需要许多操作系统的功能，它们与操作系统进行交互，而不是直接与处理器用机器语言进行交互。

操作系统和语言实现系统在计算机的机器语言接口之上是分层的，这些层可以想象为虚拟的计算机，在更高的层次上提供用户接口。例如，操作系统加上 C 语言编译器就是一台虚拟的 C 语言计算机，用其他的编译器，就能变成其他类别的虚拟计算机。大多数计算机系统都提供几个不同的虚拟计算机。用户程序在虚拟计算机的顶层上形成另一层次。计算机的层次图如图 1-2 所示。

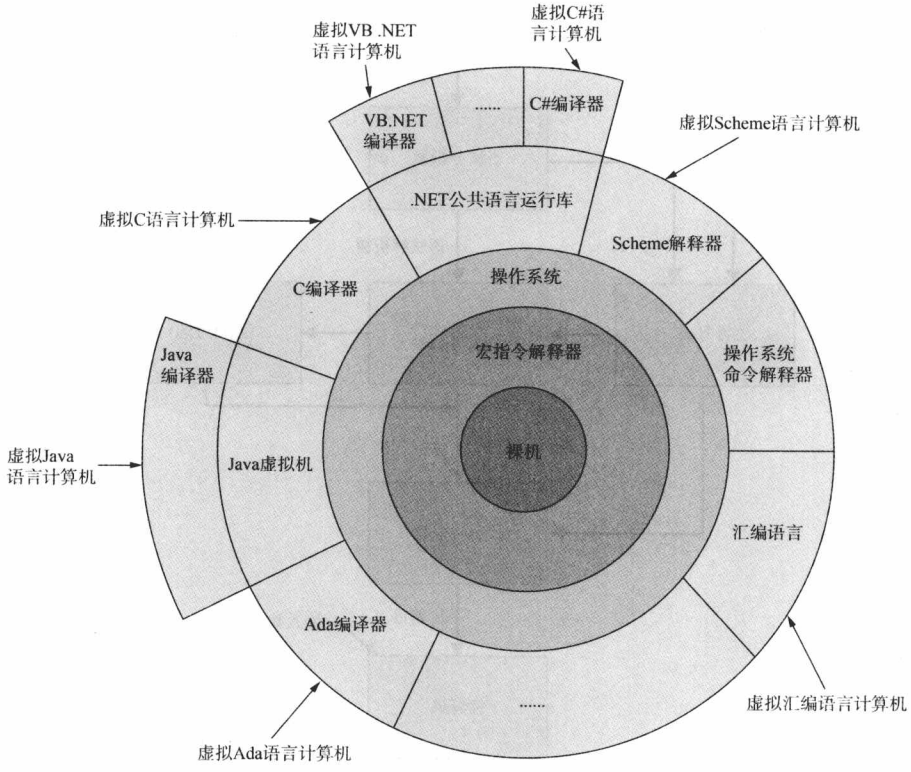


图 1-2　典型计算机系统提供的虚拟计算机分层接口

第一个高级程序设计语言的实现系统于 20 世纪 50 年代晚期出现，它是当时最复杂的软件系统。20 世纪 60 年代，人们进行了大量研究工作，来理解构造这些高级语言实现系统的过程，并将过程形式化。在这些工作中最大的成功是在语法分析领域，主要由于这一部分的实现过程是当时被透彻地理解的自动机理论和形式语言理论的部分应用。

1.7.1　编译

程序设计语言可以用三种常见方法中的任一种实现。作为一个极端方法，程序可以翻译为机器语言，由计算机直接执行，这种方法称为编译器实现，其优点是一旦翻译过程完成，程序的执行速度就非常快。大多数语言如 C、COBOL、C++和 Ada，就是由编译器实

现的。

编译器翻译的语言称为源语言。编译和程序执行的过程跨越了多个阶段，图 1-3 显示了其中最重要的几个阶段。

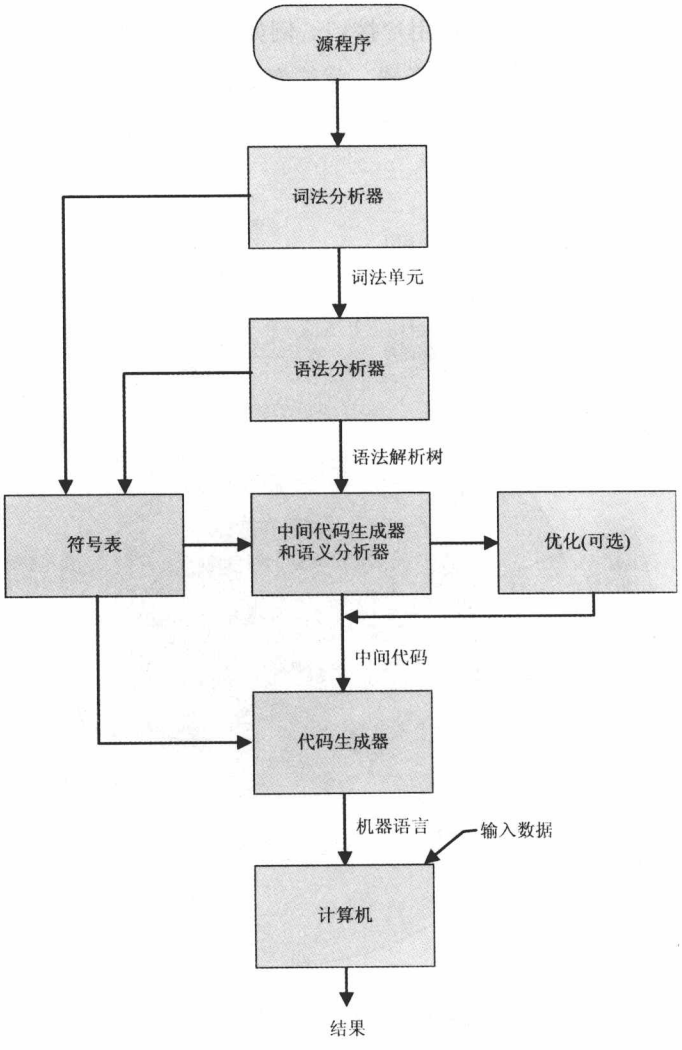


图 1-3 编译过程

词法分析器将源程序中的字符组合为词法单元。程序的词法单元为标识符、特殊字、运算符和标点符号。由于注释对编译器没有用，所以词法分析器忽略程序中的注释。

语法分析器从词法分析器中获取词法单元，用它们构成称为语法分析树的层次结构，语法分析树表示程序的语法结构。许多情况下，并不会构建实际的语法分析树，而是直接生成和使用构建树所需的信息。词法单元和语法分析树将在第 3 章中深入介绍，词法分析和语法分析将在第 4 章中介绍。

中间代码生成器用一种不同的语言生成一个程序，介于源程序和最终编译器输出的机

器语言程序之间[4]。中间语言有时看起来很像汇编语言，有时实际就是汇编语言，有时中间代码比汇编语言的层次高一些。语义分析器是中间代码生成器的一个组成部分，对语法分析中很难发现的错误进行检查，例如类型错误。

优化通常是编译过程的一个可选部分，通过将程序(通常是中间代码的程序)变得更小或更快或又小又快来改进程序。实际上，有些编译器无法做显著的优化，这类编译器可用在目标程序的运行速度远不如编译速度重要的场合。这种场合的一个例子就是程序设计初学者的计算实验室。在大多数商用和工业用场合下，运行速度比编译速度更重要，因此通常需要优化。由于许多类型的优化很难在机器语言上进行，所以大多数优化是在中间代码上做的。

代码生成器将优化过的中间代码程序翻译为相应的机器语言程序。

符号表是编译过程的数据库，其主要内容就是程序中每个用户自定义名称的类型和属性信息。这些信息由词法和语法分析器存储在符号表中，由语义分析器和代码生成器使用。

如前所述，虽然编译器生成的机器语言可以直接在硬件中运行，但几乎总要和其他一些代码一起运行。大多数用户程序的运行也需要操作系统的程序，其中最常见的是输入和输出程序。当用户程序需要时，编译器生成对所需系统程序的调用。在编译器生成的机器语言程序运行之前，必须先找到所需的操作系统的程序，并连接到用户程序。链接操作把系统程序的入口地址放到用户程序对系统程序的调用处，从而将用户程序与系统程序连接起来。有时，用户代码和系统代码合在一起，称为载入模块或可执行镜像。找到系统程序并连接到用户程序的过程，称为链接和载入，有时简称为链接，由通常称为链接器的系统程序完成。

除系统程序外，用户程序经常要链接到程序库中先前已编译过的用户程序。因此链接器不仅将指定程序链接到系统程序，也将它链接到其他用户程序。

计算机内存和处理器连接的速度通常决定了计算机的速度，因为通常指令执行比指令传输到处理器更快。这一连接称为冯·诺依曼瓶颈，它是冯·诺依曼体系结构计算机主要的速度限制因素。冯·诺依曼瓶颈是研究和开发并行计算机的一个主要动机。

1.7.2　完全解释

完全解释是实现方法中与编译相反的一个方法。在这个方法中，程序由另一个称为解释器的程序进行解释执行，而没有任何翻译过程。解释器程序的行为像是机器的软件仿真，其读取-执行周期处理的是高级语言程序语句而不是机器指令。显然，这种软件仿真为该语言提供了一个虚拟机。

完全解释的优点是易于实现许多源程序级的调试操作，因为所有运行时的错误消息都可以查找源程序级的单元。例如，如果发现数组下标越界，错误消息很容易指明源程序行和数组名称。另一方面，这一方法的严重缺陷是执行速度比编译过的系统慢 10～100 倍。执行速度慢的最根本原因是对高级语言语句的解码，比机器语言指令要复杂得多(虽然语句数比相应的机器码指令要少)。而且，不管语句执行多少次，每次都必须解码。因此，完全

4. 注意，程序和代码两个词经常可以互换使用。

解释方法的瓶颈是语句解码，而不是处理器和内存间的连接。

完全解释的另一个缺点是需要更多的存储空间。除源程序外，在解释过程中必须给出符号表，而且源程序的存储形式是为了便于存取和修改而设计的，而不是为了占用最小的存储空间。

虽然 20 世纪 60 年代一些简单的早期语言(APL、SNOBOL 和 LISP 语言)是完全解释的，但是到 20 世纪 80 年代，这种方法已经很少用于高级语言。但是近年来，随着一些网页脚本语言如 JavaScript 和 PHP 的广泛使用，完全解释的方法又以很大的声势东山再起。完全解释的过程如图 1-4 所示。

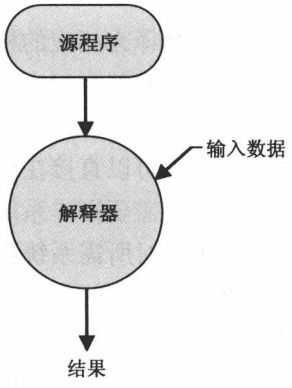


图 1-4　完全解释

1.7.3　混合实现系统

有些语言的实现系统是编译和完全解释的折中，系统将高级语言程序翻译为专为方便解释而设计的中间语言。这一方法比完全解释快，因为源语言语句只解码一次。这种实现方法称为混合实现系统。

混合实现系统的处理过程如图 1-5 所示，系统只是解释执行中间语言代码，而不是将它翻译为机器代码。

Perl 语言是以混合实现系统的方式实现的。Perl 程序在解释前先部分地编译，以检测错误，并简化解释器。

Java 语言的最初实现是混合方式的，它的中间形式称为字节码，可移植到装有字节码解释器和相应运行时系统的任何机器上。字节码解释器和相应的运行时系统统称为 Java 虚拟机。现在出现了将 Java 字节码翻译为机器码的系统，可以提高执行速度。

即时(JIT)实现系统先将程序翻译为中间语言，然后在执行过程调用方法时，将中间语言的方法编译为机器代码，机器代码会保留，用于之后的调用。即时系统现在广泛用于 Java 程

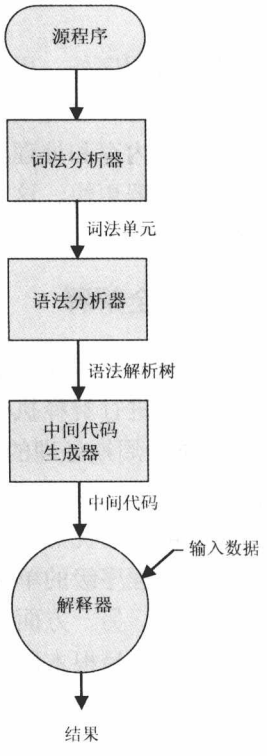


图 1-5　混合实现系统

序。.NET 语言也都是用即时系统实现的。

有时实现程序可为一种语言提供编译和解释两种实现方法。这种情况下，解释器用于开发和调试程序，在程序达到没有缺陷的状态(相对而言)后，再编译程序，以提高执行速度。

1.7.4　预处理器

预处理器是一个程序，在源程序编译前对它进行处理。预处理器指令嵌在源程序中。预处理器本质上是一个宏扩展器，预处理器指令经常用来指定包含在另一个文件中的代码。例如，C 语言预处理器指令

```
#include "myLib.h"
```

使预处理器将文件 myLib.h 的内容复制到程序中#include 的位置处。

其他预处理器指令用于定义代表表达式的符号。例如，可以用

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

定义两个指定表达式的最大值。例如，表达式

```
x = max(2 * y, z / 1.73);
```

可以被预处理器展开为

```
x = ((2 * y) > (z / 1.73) ? (2 * y) : (z / 1.73);
```

请注意，这是表达式的副作用会引发问题的一个例子。例如，如果输入到 max 宏的两个表达式中的任意一个有副作用，如 z++，就会产生问题。由于两个表达式参数中的一个会计算两次，导致宏扩展所生成的代码将 z 递增了两次。

1.8　编程环境

编程环境是软件开发中使用的工具集。这个工具集可以只包含一个文件系统、一个文本编辑器、一个链接器和一个编译器，也可以包含很多集成的工具，其中每个工具都可通过统一的用户界面来使用。在后一种情况下，软件的开发和维护效率大大提高。因此程序设计语言的特性不是系统的软件开发能力的唯一衡量标准。下面简要介绍几个编程环境。

UNIX 是一个很老的编程环境，最早发布于 20 世纪 70 年代中期，是围绕一个可移植的多程序设计操作系统而构建的，它为多种语言的软件开发和维护提供了广泛的功能强大的支持工具。过去，UNIX 缺乏的最重要的特性是工具的统一界面，这使它难以学习和使用。然而，现在 UNIX 常常通过运行于 UNIX 顶层的图形用户界面(GUI)来使用。UNIX 的图形用户界面包括 Solaris 通用桌面环境(CDE)、GNOME 和 KDE。这些图形用户界面使 UNIX 的界面显示与 Windows 和 Macintosh 系统的界面相似。

Borland JBuilder 是一个编程环境，为 Java 开发提供集成的编译器、编辑器、调试器和

文件系统，这 4 个工具都通过图形界面来使用。JBuilder 是用于创建 Java 软件的一个复杂而功能强大的系统。

Microsoft Visual Studio .NET 是软件开发环境演化过程的新进展，它是大量精巧的软件开发工具的集合，这些工具都可通过视窗界面来使用。这一系统可用 5 种.NET 语言中的任意一种来开发软件，这 5 种语言是 C#、Visual BASIC .NET、JScript(Microsoft 版本的 JavaScript)、F#(函数式语言)和 C++/CLI。

NetBeans 是一个主要用于 Java 网页应用程序开发的编程环境，但也支持 JavaScript、Ruby 和 PHP。Visual Studio 和 NetBeans 都不仅仅是开发环境，它们同时也是框架，这意味着它们实际上提供了应用程序代码的通用部分。

小　结

学习程序设计语言很有价值，有以下一些重要原因：能提高编写程序时使用不同语言结构的能力，能更明智地为项目选择程序设计语言，学习新的语言变得更容易。

计算机可广泛用于各种各样解决问题的领域。特定程序设计语言的设计和评价，很大程度取决于它所应用的领域。

评价语言的最重要的标准包括可读性、可写性、可靠性和总成本。这些是本书后面对各种语言特性进行分析和判断的基础。

对语言设计最主要的影响来自于计算机体系结构和软件设计方法学。

设计一种程序设计语言主要是一种工程技巧，必须在特性、结构和功能方面进行一系列的权衡。

实现程序设计语言的主要方法是编译、完全解释和混合实现。

编程环境已经成为软件开发系统的重要组成部分，程序设计语言只是其中的一部分。

复 习 题

1. 对程序员来说，为什么说即使他永远不会真的去设计程序设计语言，了解一些语言设计的背景知识也很有用？
2. 程序设计语言特性的知识对于整个计算机界有何益处？
3. 在过去 50 年间，哪种程序设计语言是用于科学计算的主要语言？
4. 在过去 50 年间，哪种程序设计语言是用于商务应用的主要语言？
5. 在过去 50 年间，哪种程序设计语言是用于人工智能的主要语言？
6. UNIX 是用哪一种语言写的？
7. 一种语言的特性太多有何坏处？
8. 用户自定义的运算符重载如何降低程序的可读性？
9. 列举一个在 C 语言的设计中缺少正交性的例子。
10. 哪种语言将正交性作为基本的设计标准？
11. 在缺乏更复杂的控制语句的语言中，使用哪种基本的控制语句来实现复杂控制？

12. 程序设计语言的哪一种结构提供了过程抽象？
13. 程序可靠是什么意思？
14. 为什么对子程序参数进行类型检查很重要？
15. 什么是别名？
16. 什么是异常处理？
17. 为什么可读性对于可写性很重要？
18. 为什么特定语言的编译成本与该语言的设计相关？
19. 在过去的 50 年中，影响程序设计语言设计的最重要因素是什么？
20. 有一类程序设计语言的结构以冯・诺依曼计算机体系结构来命名，其名称是什么？
21. 20 世纪 70 年代软件开发研究的结果，发现了哪两个程序设计语言的缺点？
22. 面向对象程序设计语言的三个基本特性是什么？
23. 第一种支持面向对象程序设计三个基本特性的程序设计语言是什么？
24. 举出两个语言设计标准相互矛盾的一个例子。
25. 实现程序设计语言的三种常用方法是什么？
26. 编译和完全解释哪一个生成的程序执行速度快？
27. 符号表在编译器中有什么用？
28. 链接器是做什么的？
29. 为什么冯・诺依曼瓶颈很重要？
30. 用完全解释的方法实现一种语言有什么优点？

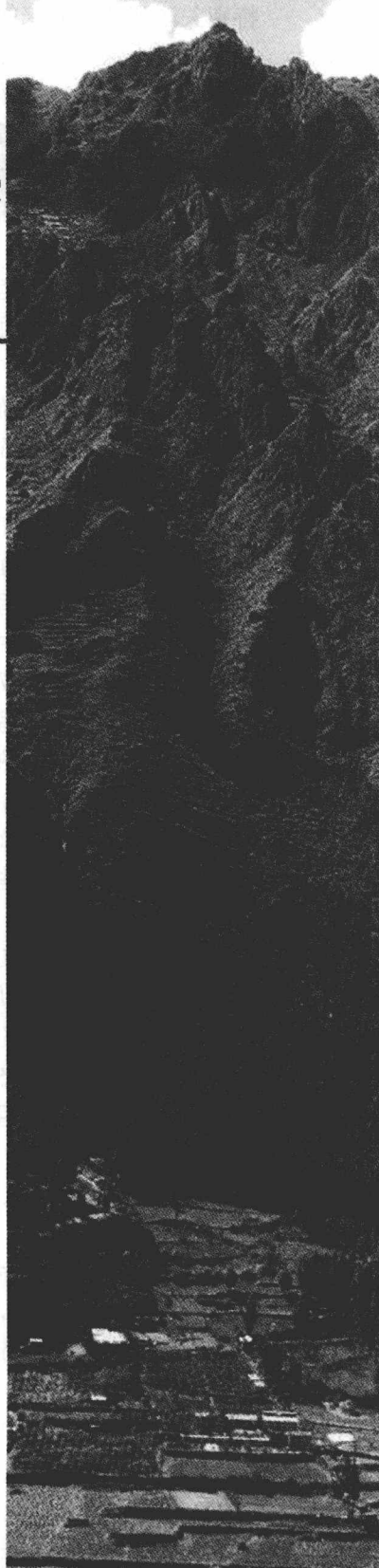
习　　题

1. 你认为抽象思维的能力受语言技能的影响吗？证明你的观点。
2. 你了解的特定程序设计语言中，哪些特性的合理性你不清楚？
3. 你对于用一种程序设计语言处理所有程序设计领域的问题，有何论据支持？
4. 你对于用一种程序设计语言处理所有程序设计领域的问题，有何论据反对？
5. 除了本章介绍的评价程序设计语言的标准之外，再说出一种标准并解释它。
6. 在你看来，哪一条常见的程序设计语言语句对可读性的危害最大？
7. Java 语言使用右括号来标志所有复合语句的结束，对这一设计有哪些支持和反对的意见？
8. 许多程序设计语言对用户自定义名称中的字母区分大小写，对这一设计决策有哪些支持和反对的意见？
9. 从不同的角度说明程序设计语言的成本。
10. 即使硬件相对便宜也要编写高效的程序，对此你有什么意见？
11. 说出你知道的程序设计语言中，一些在效率与安全性方面的设计权衡。
12. 在你看来，完美的程序设计语言应包含哪些主要特性？
13. 你所学的第一种高级程序设计语言是以哪一种方式实现的，完全解释、混合实现，还是编译？

14. 说出你用过的一些编程环境的优缺点。
15. 简单变量的类型声明语句是如何影响语言的可读性的(因为一些语言不需要类型声明语句)?
16. 用本章介绍的标准，针对你所了解的某种程序设计语言写一篇评论。
17. 有些程序设计语言，例如 Pascal，使用分号来分隔语句，然而 Java 用分号作为语句的结束。在你看来，这两种情况哪一种更自然，更不容易导致语法错误？给出理由。
18. 许多现代的程序设计语言允许使用两类注释，一种在两端都用分隔符(多行注释)，另一种只在注释开始处用分隔符(单行注释)。根据我们的标准，说明每类注释的优缺点。

第2章

主要程序设计语言的发展



2.1 Zuse 的 Plankalkül 语言
2.2 伪代码
2.3 IBM 704 计算机与 Fortran 语言
2.4 函数式程序设计：LISP 语言
2.5 迈向成熟的第一步：ALGOL 60
2.6 商务记录的计算机化：COBOL 语言
2.7 分时处理的开始：BASIC 语言
2.8 满足所有人的需要：PL/I
2.9 两种早期的动态语言：APL 和 SNOBOL
2.10 数据抽象的开始：SIMULA 67
2.11 正交设计：ALGOL 68
2.12 ALGOL 系列语言的早期后代语言
2.13 基于逻辑的程序设计：Prolog 语言
2.14 历史上规模最大的设计工作：Ada 语言
2.15 面向对象的程序设计：Smalltalk
2.16 结合命令式和面向对象的特性：C++
2.17 基于命令式的面向对象语言：Java
2.18 脚本语言
2.19 一流的.NET 语言：C#
2.20 标记与程序设计混合的语言

本章介绍一些程序设计语言的发展过程，探讨每一种语言的设计环境，重点介绍各种语言的贡献和开发动机。我们不介绍语言的整体说明，只介绍每种语言引入的一些新特性，特别关注那些对后来的语言和计算机科学领域有重大影响的特性。

本章不深入介绍任何语言特性或概念，这些留给后面的章节。在历数这些语言的发展过程时，只对其特性做简要的非正式的解释就足够了。

本章介绍许多读者并不熟悉的各种各样的语言和语言原理。这些主题在后面的章节中会详细讨论，觉得看不懂的读者，可以在研读过本书的其余部分之后再来看本章。

在这里选择哪种语言来讨论是由作者主观选定的，有些读者会不高兴他们喜欢的一种或多种语言没有涉及。但是，为了将发展历史的介绍限制在适当的篇幅内，必须舍弃一些被某些人高度关注的语言。我们根据每种语言对语言发展和整个计算机领域的重要性的估计来选择语言，对本书后面将提到的其他一些语言也做简要介绍。

本章的编排如下：基本上按时间顺序介绍语言的最初版本，但是语言的后续版本与最初版本一起介绍，而不是放在后面的章节中。例如，Fortran 2003 在 Fortran I(1956)的章节中介绍。在某些情况下，在独立的章节中介绍某语言时，也会介绍一些不太重要的相关语言。

本章包含 14 个完整的示例程序，每个程序都是用不同语言写的。本章并不介绍其中任何一个程序，它们仅用于展示这些语言写的程序是什么样子。熟悉任意一种常见命令式语言的读者，都应该能够阅读和理解这些程序中的大部分代码，除了用 LISP、COBOL 和 Smalltalk 写的程序以外(类似于 LISP 示例的 Scheme 函数将在第 15 章中介绍)。我们用 Fortran、ALGOL60、PL/I、BASIC、Pascal、C、Perl、Ada、Java、JavaScript 和 C#的程序解决同一问题。请注意，这些现代语言大都支持动态数组，但是由于示例问题非常简单，示例

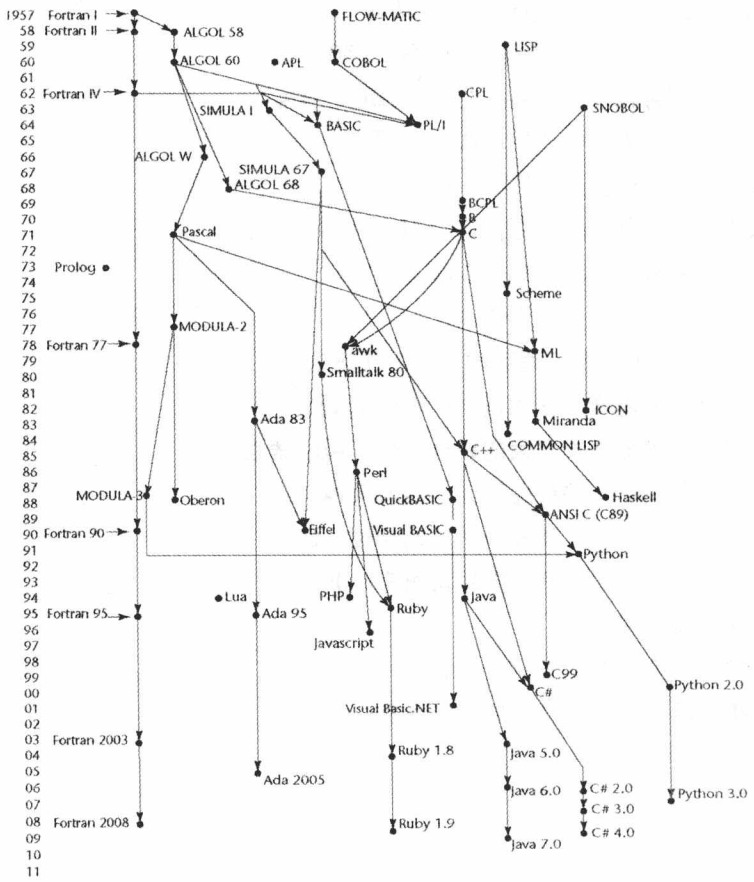


图 2-1 常用高级程序设计语言之间的演进关系

程序没有使用动态数组。在 Fortran 95 的程序中，我们也没有使用相关特性来避免对循环的使用，部分原因是为了使程序简单、可读，部分原因是为了展示该语言基本的循环结构。

图 2-1 是本章介绍的高级语言的演进关系图。

2.1　Zuse 的 Plankalkül 语言

本章介绍的第一种程序设计语言从很多方面来说都不同寻常。首先，它没有实现，其次，虽然它在 1945 年就开发出来了，但它的说明直到 1972 年才发表。由于极少有人熟悉这一语言，所以直到它诞生了 15 年之后，它的一些功能才在其他语言中出现。

2.1.1　历史背景

在 1936 年至 1945 年间，德国科学家 Konrad Zuse(发音是“Tsoozuh”)用机电继电器搭建了一系列复杂而精密的计算机。到了 1945 年初，盟军轰炸摧毁了除最新模型 Z4 之外的所有模型，于是他搬到偏远的巴伐利亚村庄 Hinterstein，他的研究小组成员也各奔东西。

Zuse 独自一人着手开发一种在 Z4 上表示计算过程的语言，这是他自 1943 年开始作为博士论文内容的一个项目。他将这种语言命名为 Plankalkül，意思是程序演算。在一份直到 1972 年才发表的注明日期为 1945 年的长篇手稿(Zuse，1972)中，Zuse 定义了 Plankalkül 语言，并用该语言为各种不同的问题编写了算法。

2.1.2　语言概述

Plankalkül 语言相当完整，在数据结构方面具有一些最先进的特性。Plankalkül 中最简单的数据类型是单个比特位，整型和浮点数类型都是用单个比特建立的。浮点型采用两两互补的标记方法和隐藏位方案，现在该方案用来避免存储浮点值规格化后的小数部分的最高位。

除了常用的标量类型之外，Plankalkül 还包含了数组和记录(在基于 C 的语言中称为结构)，记录可以包含嵌套的记录。

虽然这种语言没有显式的 goto 语句，但是它包含了与 Ada 语言中 for 语句相似的迭代语句。它还有一个带有上标的 Fin 命令，上标指示跳出一定数目的嵌套迭代循环或跳到一个新的迭代循环的起始处。Plankalkül 包含一条选择语句，但是没有 else 子句。

Zuse 程序最有趣的特点之一是用数学表达式来表示程序变量之间的关系。这些表达式声明了程序执行到它们所在的代码处时什么为真，这与 Java 语言和公理化语义中的断言非常相似，公理化语义将在第 3 章中介绍。

Zuse 的手稿中包含的程序比 1945 年前写的任何程序都复杂得多，这些程序功能包括对数组排序、测试图的连通性、执行包括求平方根在内的整数和浮点数的运算，以及对含有括号和 6 个不同优先级运算符的逻辑公式进行语法分析。其中最令人吃惊的也许是他长达 49 页的下国际象棋的算法，而他并不是国际象棋的专家。

如果计算机科学家在 20 世纪 50 年代早期发现了 Zuse 对 Plankalkül 语言的说明，那么唯一妨碍该语言按定义来实现的就只是它的标记方法了。Plankalkül 语言的每一条语句包含两行或三行代码，第一行与现在程序设计语言的语句非常相似。第二行是可有可无的，包含了第一行中所用数组的下标。19 世纪中叶，Charles Babbage 在为他的分析机所写的程序中采用了相同的下标表示方法。每条 Plankalkül 语句的最后一行包含第一行中变量的类型名称。第一次见到这种标记方法时，会很令人吃惊。

下面例子中的赋值语句将表达式 A[4]+1 的值赋给 A[5]，演示了这种标记方法。标号为 V 的行给出下标，标号为 S 的行给出数据类型。在这个例子中，1.n 表示 n 位的整数。

```
  | A + 1 => A
V | 4       5
S | 1.n     1.n
```

我们只能推测，如果 Zuse 的工作在 1945 年甚至 1950 年得到广泛的了解，那么程序设计语言会向什么方向发展。考虑以下情况也很有趣：如果他是在和平环境下与其他科学家一起工作，而不是在 1945 年与世隔绝的德国工作，他的工作将会有怎样不同的结果。

2.2 伪代码

首先请注意，这里所用的伪代码一词与现在的意义是不同的。将本节介绍的语言称为伪代码，是因为它们在开发和使用时(20 世纪 40 年代晚期和 50 年代早期)就是这样命名的。但是，它们显然不是现在意义上的伪代码。

20 世纪 40 年代晚期和 50 年代早期可用的计算机远没有现在的计算机好用。除了速度慢、不可靠、价格昂贵和内存极其小之外，那时的机器由于缺乏支持软件，很难编程。

由于没有高级程序设计语言，甚至没有汇编语言，程序设计采用机器语言，既单调乏味又易于出错。问题之一是用数值编码来表示指令，例如 ADD 指令用代码 14 来表示，而不是用有意义的文本名称来表示，即使只用一个字母也好。这使程序很难阅读。更严重的问题是使用绝对寻址，使程序的修改既单调乏味又易于出错。例如，假设在内存中有一个机器语言的程序，程序中的许多指令都指向程序中的其他位置，通常是引用数据或表明分支指令的目标。在程序结尾以外的任何位置插入指令，会使指向插入点之后地址的所有指令都不再正确，因为这些地址都必须增加，以便为新的指令留出空间。为了使指令正确插入，需要找到并修改所有指向插入指令之后的地址的指令。删除一条指令时，也会产生同样的问题。不过在这种情况下，机器语言通常使用一条“无操作”指令来替换删除掉的指令，从而避免上述问题。

以上是所有机器语言的典型问题，是发明汇编器和汇编语言的原始动机。此外，那时大多数程序设计所解决的问题都是数值问题，需要浮点算术运算和便于使用数组的下标索引方法。但是在 20 世纪 40 年代晚期和 50 年代早期，计算机体系结构不支持这两种功能，这种缺陷自然导致了较高级语言的产生。

2.2.1 Short Code 语言

第一种新语言称为 Short Code，是 1949 年由 John Mauchly 开发的，用于 BINAC 计算机，这是最早的能成功存储程序的电子计算机之一。Short Code 语言后来移植到 UNIVAC I 计算机(在美国销售的最早的商业电子计算机)上，有好些年都一直是该机器上编程的主要语言之一。虽然人们对最初的 Short Code 语言知之甚少，因为它的完整说明从未发表过，但是 UNIVAC I 版本的一份编程手册(Remington-Rand，1952)保存了下来。这两个版本应该非常相似。

UNIVAC I 内存中的一个字有 72 位，分为 12 个 6 位的字节。Short Code 语言由已编码的用于计算的数学表达式组成。代码是字节对形式的值，许多等式可以用字来编码。一些运算的编码如下：

```
01  -    06  abs value    1n (n+2)nd power
02  )    07  +            2n (n+2)nd root
03  =    08  pause        4n if <= n
04  /    09  (            58 print and tab
```

变量用字节对编码来命名，用做常量的地址也是一样。例如，X0 和 Y0 都可以是变量。语句

```
X0 = SQRT(ABS(Y0))
```

可用一个字编码为 00 X0 03 20 06 Y0。开头的 00 用于填充补足整个字。有趣的是没有乘法的编码，如同代数中一样，简单地将两个操作数放在一起就表示乘法。

Short Code 语言并不翻译为机器语言，相反，它由完全解释器来实现。在当时，这一过程称为自动程序设计，这显然简化了程序设计过程，但它是以执行时间为代价的。Short Code 的解释执行约比机器代码的执行慢 50 倍。

2.2.2 Speedcoding 系统

在其他地方，人们正在开发解释系统(它扩展了机器语言)来包含浮点运算。John Backus 为 IBM 701 开发的 Speedcoding 系统就是这样的一个系统(Backus，1954)。Speedcoding 解释器能够有效地将 701 机器转换为一个虚拟的三地址浮点运算器。该系统包含了在浮点数据上进行 4 种算术运算的伪指令，以及求平方根、正弦、反正切、指数和对数的运算。条件和无条件分支，以及输入输出转换也是该虚拟体系结构的组成部分。该系统的局限是，在载入解释器后，剩余的可用内存只有 700 个字，而加法指令的执行要花 4.2 毫秒。另一方面，Speedcoding 系统包含了地址寄存器自动增量的新机制，这一机制直到 1962 年才在 UNIVAC 1107 计算机的硬件中出现。由于有了这一机制，只需 12 条 Speedcoding 指令就能够完成矩阵乘法。Backus 声称，用机器码要花 2 周才能编写出来的程序，用 Speedcoding 来编写只需几个小时。

2.2.3 UNIVAC“编译”系统

在 1951 年至 1953 年间，UNIVAC 公司中由 Grace Hopper 领导的一个小组开发了一系列“编译”系统，称为 A-0、A-1 和 A-2，将伪代码扩展为机器码子程序，其方法与将宏扩展为汇编语言相同。这是机器码的巨大进步，因为所写出的源程序短了很多，但这些“编译器”源程序的伪代码仍然非常原始。Wilkes(1952)独立地提出了类似的方案。

2.2.4 相关工作

大约在同一时期，人们也在研究简化程序设计工作的其他方法。剑桥大学的 David J. Wheeler 提出了一种方法，使用可重新分配地址的块结构，部分地解决了绝对寻址的问题(Wheeler，1950)。后来剑桥大学的 Maurice V. Wilkes 扩展了这一想法，设计了一个汇编程序，将指定的子程序组合在一起，并分配存储空间(Wilkes 等，1951，1957)。这确实是一个重要且根本性的进步。

我们还应该提到汇编语言。汇编语言产生于 20 世纪 50 年代早期，与上述伪代码差别很大，但是对高级语言的设计没有多大影响。

2.3 IBM 704 计算机与 Fortran 语言

计算机历史上最伟大的进步当属 1954 年 IBM 704 计算机的诞生，其主要原因是它功能强大，催生了 Fortran 语言。可以认为：如果不是 IBM 开发了 704 计算机和 Fortran 语言，不久也会有其他公司开发出类似的计算机和相应的高级语言。然而，IBM 是第一个兼具远见和资源来从事这一开发工作的组织。

2.3.1 历史背景

从 20 世纪 40 年代晚期到 50 年代中期，人们还能忍受解释系统速度缓慢的主要原因之一，就是在可用的计算机中没有能完成浮点运算的硬件。所有的浮点运算都只能用软件模拟，这是一个相当耗时的过程。由于软件模拟的浮点运算消耗了大量的处理器时间，解释和模拟变址寻址的开销就显得不那么多了。只要浮点运算只能由软件来做，解释系统的开销就可以接受。然而，当时的许多程序员从不使用解释系统，而是喜欢更高效的手工编码的机器语言或汇编语言。IBM 704 系统在硬件中同时支持变址寻址和浮点运算，它的发布宣布了解释时代的终结，至少是在科学计算领域的终结。由于包含了浮点运算硬件，所以避免了解释系统潜在的开销。

虽然 Fortran 语言常常作为第一种可编译的高级语言而被人称道，但开发出第一个这类语言的荣誉归属何人却一直悬而未决。Knuth 和 Pardo(1977)将这一荣誉归于 Alick E. Glennie，因为他为 Manchester Mark I 计算机开发了 Autocode 编译器。Glennie 在英国圣哈斯德(Fort Halstead)的皇家军备研究院(Royal Armaments Research Establishment)开发了这一编译器，完成于 1952 年 9 月。但是 John Backus(Wexelblat，1981，第 26 页)认为 Glennie

的 Autocode 编译器是很底层的、面向机器的，不能认为是一个编译系统。Backus 将上述荣誉归于麻省理工学院(MIT)的 Laning 和 Zierler。

Laning 和 Zierler 的系统(Laning 和 Zierler，1954)是第一个被实现的代数翻译系统，“代数”的意义是指它能够翻译算术表达式，用独立编码的子程序来计算超越函数(如正弦和对数)，并且支持数组。该系统于 1952 年夏在 MIT 的 Whirlwind 计算机上实现了试验样机，并于 1953 年 5 月在同一计算机上实现了更实用的版本。这个翻译系统在程序中为每个公式或表达式生成一个子程序调用。语言源程序很容易看懂，唯一包含的机器指令是分支指令。虽然这项工作是在 Fortran 相关工作之前，但是它从未流传到 MIT 以外。

尽管有这些前期的工作，但第一种被广泛接受的可编译的高级语言是 Fortran。下面的章节按年代来介绍这一重要的发展过程。

2.3.2　设计过程

在 1954 年 5 月 IBM 704 计算机发布之前，Fortran 语言的开发计划就已经开始了。1954 年 11 月，John Backus 和他在 IBM 的小组已经完成了名为“IBM 数学公式翻译系统：FORTRAN”的报告(IBM，1954)。这一文件在实现 Fortran 语言之前，说明了 Fortran 语言的最初版本，现在称为 Fortran 0。这一文件大胆地宣称，Fortran 语言将提高手工编码程序的效率，且便于编写解释式伪代码系统程序，还乐观地宣称，Fortran 语言将消除编码错误，无需程序调试过程。基于这一假设，最初的 Fortran 编译器只包含很少的语法错误检查。

Fortran 的开发环境如下：(1)计算机的内存很小，速度慢，相对不可靠；(2)计算机的主要用途是科学计算；(3)当时没有高效的计算机程序设计方法；(4)由于与程序员的成本相比，计算机的成本更高，第一代 Fortran 语言编译器的主要目标是生成高速度的目标代码。这样的环境直接决定了 Fortran 早期版本的特性。

2.3.3　Fortran I 概述

从 1955 年 1 月开始一直到 1957 年 4 月发布编译器为止，Fortran 0 在实现的过程中一直在修改，最终实现的语言称为 Fortran I，1956 年 10 月出版的第一本 Fortran 语言的程序员参考手册对它做了说明(IBM，1956)。Fortran I 包含输入/输出格式化、至多 6 个字符的变量名(在 Fortran 0 中至多 2 个字符)、用户自定义子程序(虽然不能单独编译)、If 选择语句和 Do 循环语句。

Fortran I 所有的控制语句都是基于 704 计算机指令的，也不清楚是 704 计算机的设计者决定了 Fortran I 中控制语句的设计，还是 Fortran I 的设计者向 704 计算机的设计者建议了这些指令。

在 Fortran I 语言中没有数据类型的说明语句。以 I、J、K、L、M 和 N 开头的变量都默认为整数类型，所有其他变量都默认为浮点类型。按这样的约定来选择字母是因为当时的科学家和工程师通常使用 i、j 和 k 来做变量的下标。Fortran 语言的设计者大方地加入了三个额外的字母。

Fortran 开发小组在设计语言时最大胆的宣称是，编译器生成的机器代码将能达到手工

编码约一半的效率[1]。这一点相对于其他任何因素，都更令潜在的用户怀疑，Fortran 语言在真正发布前，许多人就失去了对它的兴趣。然而令几乎所有人惊讶的是，Fortran 开发小组几乎实现了这一效率目标。构建第一个编译器的 18 人年的工作，大部分都花在了优化上，成效显著。

从 1958 年 4 月的一份调查结果可以看到 Fortran 语言的初步成功。那时，为 704 计算机所写的代码约有一半是用 Fortran 语言写的，尽管仅仅一年之前程序设计界的大部分人还持怀疑态度。

2.3.4 Fortran II

Fortran II 编译器发布于 1958 年春天，它修正了 Fortran I 编译系统中的许多缺陷，为语言增加了一些重要的特性，其中最重要的就是子程序的单独编译。如果没有单独编译，程序的任何改变都需要重新编译整个程序。Fortran I 没有单独编译的功能，加上 704 计算机的可靠性差，实际上把程序的长度限制为 300～400 行(Wexelblat，1981，第 68 页)。如果程序太长，就很难在死机之前全部编译完成。Fortran II 能够预编译子程序的机器语言版本，大大地缩短了编译过程，使更大程序的开发成为现实。

2.3.5 Fortran IV、77、90、95、2003 和 2008

虽然开发出了 Fortran III，但是没有广泛使用，而 Fortran IV 成为了当时最广泛使用的程序设计语言之一。经过 1960—1962 年的演变，Fortran IV 标准化为 Fortran 66(ANSI，1966)，虽然 Fortran 66 这一名称很少使用。Fortran IV 在 Fortran II 的很多方面做了改进，其中最重要的改进包括变量的显式类型声明、逻辑判断的 If 语句结构，以及将子程序作为参数传递给其他子程序的能力。

1978 年，Fortran IV 由新标准 Fortran 77 替代(ANSI，1978a)。Fortran 77 保留了 Fortran IV 中的大部分特性，增加了字符串处理、逻辑判断的循环控制语句，以及有可选 Else 子句的 If 语句。

Fortran 90(ANSI，1992)与 Fortran 77 差别很大，最重要的新特性是动态数组、记录、指针、多选语句和模块。而且，Fortran 90 的子程序可以递归调用。

在 Fortran 90 的定义中引入了一个新概念：去掉早期版本中的一些语言特性。虽然 Fortran 90 包含了 Fortran 77 所有的特性，但是语言定义中也包含了一系列语言结构，建议在语言的下一版本中去除。

Fortran 90 有两项简单的语法变化，改变了程序和语言的说明文字的外观。首先，去掉了代码所需的固定形式，即语句的特定部分需要位于特定的字符位置，例如语句标号只能出现在前 5 个位置，语句不能从第 7 个位置开始。这种死板的代码格式是为了使用打孔卡片而设计的。第二种改变是语言名称正式的拼写由 FORTRAN 变成 Fortran。这一改变使

1. 实际上，Fortran 小组相信，编译器生成的代码不会比手写的机器代码的一半速度还慢，否则用户不会接受这种语言。

Fortran 程序中关键字和标识符使用全大写字母的约定也发生了变化。新的约定是：只有关键字和标识符的第一个字母才大写。

Fortran 95(INCITS/ISO/IEC，1997)继续了语言的发展，但只做了少量修改。此外，还增加了一种新的迭代结构 Forall，以简化 Fortran 程序的并行化。

Fortran 2003(Metcalf 等，2004)增加了对面向对象程序设计、参数化的派生类型、过程指针和与 C 程序设计语言互通的支持。

Fortran 语言的最新版本是 Fortran 2008(ISO/IEC 1539-1, 2010)，增加了对定义局部作用域的块、联合数组(它提供了平行的执行模型)和 DO CONCURRENT 结构(指定循环时不需要相互依赖)的支持。

2.3.6　评价

最初的 Fortran 语言设计小组认为，语言设计只是设计语言翻译器这一关键任务必需的前期工作。而且，他们从未想到 Fortran 语言可以用在非 IBM 生产的计算机上。实际上，他们考虑为其他 IBM 计算机构建 Fortran 语言编译器，只是因为在 704 计算机上的 Fortran 语言编译器推出之前，IBM 709 计算机作为 704 计算机的后续产品就已经发布了。Fortran 语言对计算机使用的影响，以及所有后来的程序设计语言都得益于 Fortran 语言的事实，相对于 Fortran 语言设计者的朴素目标，实在令人感叹。

Fortran I 及其 Fortran 90 版本之前的所有后续版本，能够使编译器高度优化的特性之一，就是所有变量的类型和存储位置在运行前都固定下来，在运行时不再分配新的变量或存储空间。这是用灵活性换取了简单性和高效性。这一特性排除了递归子程序的可能，很难实现数据结构的动态增长或变化。当然，在开发 Fortran 早期版本的时代，所开发的程序在本质上主要用于数值计算，与现在的软件项目相比很简单，因此这种牺牲不算大。

无论如何强调 Fortran 语言的成功都不为过：它显著地改变了计算机的使用方式。当然，这主要归因于它是第一种广泛使用的高级语言。与后来发展起来的语言和语言原理相比，Fortran 的早期版本在很多方面都很差，这也是可以理解的。毕竟，将 1910 年 T 型福特汽车的性能和舒适度与 2013 款福特野马汽车相比，是不公平的。不过，尽管 Fortran 语言也有不足，但是对 Fortran 软件的巨大投资势头，以及其他一些因素，使 Fortran 语言的使用超过了半个世纪。

ALGOL 60 语言的设计者之一 Alan Perlis 在 1978 年这样评价 Fortran 语言："Fortran 是计算机界的通用语言，它是商业界的语言，它已经生存下来并且还将存在下去，因为它已经成为充满活力的商业贸易极其有用的一部分(Wexelblat，1981，第 161 页)。"

以下是 Fortran 95 程序的一个例子：

```
! Fortran 95 Example program
!  Input:  An integer, List_Len, where List_Len is less
!          than 100, followed by List_Len-Integer values
!  Output: The number of input values that are greater
!          than the average of all input values
```

```
Implicit none
Integer Dimension(99) :: Int_List
Integer :: List_Len, Counter, Sum, Average, Result
Result= 0
Sum = 0
Read *, List_Len
If ((List_Len > 0) .AND. (List_Len < 100)) Then
! Read input data into an array and compute its sum
   Do Counter = 1, List_Len
      Read *, Int_List(Counter)
      Sum = Sum + Int_List(Counter)
   End Do
! Compute the average
   Average = Sum / List_Len
! Count the values that are greater than the average
   Do Counter = 1, List_Len
      If (Int_List(Counter) > Average) Then
         Result = Result + 1
      End If
   End Do
! Print the result
   Print *, 'Number of values > Average is:', Result
Else
   Print *, 'Error - list length value is not legal'
End If
End Program Example
```

2.4 函数式程序设计：LISP 语言

第一个函数式程序设计语言的发明是为了提供表处理这一语言特性，人工智能(AI)领域的最初一些应用需要这个特性。

2.4.1 人工智能的起源和表处理

20 世纪 50 年代中期，人们在一些领域对人工智能产生了兴趣，包括语言学、心理学和数学领域。语言学家关心自然语言的处理，心理学家对人类的信息存入和取出，以及大脑的其他一些基本过程感兴趣，数学家的兴趣在于某些智能过程的机器化，例如定理证明。对这些问题的研究都得到了同样的结论：必须实现某种方法，使计算机能够处理链表中的符号数据。当时，大多数计算处理的都是数组中的数值数据。

表处理的概念是 RAND 公司的 Allen Newell、J. C. Shaw 和 Herbert Simon 提出的，最早发表于一篇经典的论文，该论文介绍了最早的人工智能程序之一“逻辑理论家”(Logical Theorist)[2]，以及可实现这一程序的语言(Newell 和 Simon，1956)。这一语言是 IPL-I

2. “逻辑理论家”通过命题演算得到定理的证明。

(Information Processing Language I，信息处理语言 I)，并没有实现。它的下一版本 IPL-II 在 RAND 公司的 Johnniac 计算机上实现了。对 IPL 的开发持续到 1960 年，当时出版了对 IPL-V 的说明(Newell 和 Tonge，1960)。IPL 语言很低级，所以没有广泛使用。它们实际上是为一种假想的计算机开发的汇编语言，采用一个包含了表处理指令的解释器来实现。IPL 语言没有广泛使用的另一个原因是，它们是在鲜为人知的 Johnniac 计算机上实现的。

IPL 语言的贡献在于表的设计，并证明了表处理是可行的、有用的。

IBM 在 20 世纪 50 年代中期对人工智能产生了兴趣，将定理证明作为一个示范领域。那时 Fortran 语言项目还在进行，开发 Fortran I 编译器的高昂代价让 IBM 决定将表处理功能作为 Fortran 语言的一部分，而不是一种新的语言。因此，Fortran 表处理语言(Fortran List Processing Language，FLPL)设计并实现为 Fortran 语言的扩展，还用 FLPL 编写了一个平面几何的定理证明程序，当时认为这是定理的机器证明中最简单的领域。

2.4.2　LISP 语言的设计过程

1958 年，MIT 的 John McCarthy 在 IBM 的信息研究部获得了一个夏季职位，这个夏天他的目标是研究符号运算，并找到一些符号运算的应用需求。作为一个导向性的示例问题领域，他选择了代数表达式的微分，对这一问题的研究产生了一系列语言需求，包括数学函数控制流程的方法：递归和条件表达式。当时唯一可用的高级语言 Fortran I 没有这些方法。

对符号微分的研究产生了另一个需求，就是链表的动态分配和废弃链表的隐式回收。McCarthy 只是不希望优美的微分算法被显式的回收语句弄乱。

由于 FLPL 不支持递归、条件表达式、动态存储分配和隐式回收，McCarthy 显然需要一种新的语言。

McCarthy 于 1958 年秋季回到 MIT 后，和 Marvin Minsky 从电子研究实验室获得经费，组成了 MIT 人工智能项目组。该项目组一开始的重要工作就是建立一个表处理软件系统，并用该系统实现 McCarthy 提出的“建议采纳者”(Advice Taker)[3]程序。这一应用推动了表处理语言 LISP 的产生。LISP 语言最初的版本有时称为“纯 LISP”，因为它是一种纯粹的函数式语言。下面的章节介绍纯 LISP 的发展过程。

2.4.3　语言概述

2.4.3.1　数据结构

纯 LISP 只有两种数据结构：原子和表。原子是标识符形式的符号或数字的字面值。在链表中存储符号信息的概念是自然的，已经在 IPL-II 中应用。这一结构允许在任一点插入和删除，当时认为这两项操作是表处理所必需的，不过最终人们确定 LISP 程序基本不需要这两项操作。

---

3. “建议采纳者”使用形式语言的句子来表示信息，通过逻辑推理来决定采取的行动。

在指定表结构时，要把表的元素放在圆括号中。简单表的元素仅限原子，形式是：

```
(A B C D)
```

嵌套的表结构也用圆括号来表示。例如，表

```
(A  (B C)  D  (E  (F G)))
```

由 4 个元素组成。第 1 个元素是原子 A，第 2 个是子表(B C)，第 3 个是原子 D，第 4 个是子表(E(F G))，它的第 2 个元素是子表(F G)。

表在内部存储为单向链表结构，其中每一结点有两个指针，每一结点表示一个表元素。包含原子的结点的第一个指针指向该原子的表示，例如它的符号或数值或者子表的指针。以子表为元素的结点，第一个指针指向子表的第一个结点。上述两类结点中，第二个指针都指向表的下一个元素。表由指向它第一个元素的指针来引用。

前面给出的两个表的内部表示如图 2-2 所示。请注意，图中表的元素是水平表示的，最末尾的表元素没有后继，因此它的链接是 NIL，在图 2-2 中表示为该元素内的一条对角线。子表也是用同样的结构来表示的。

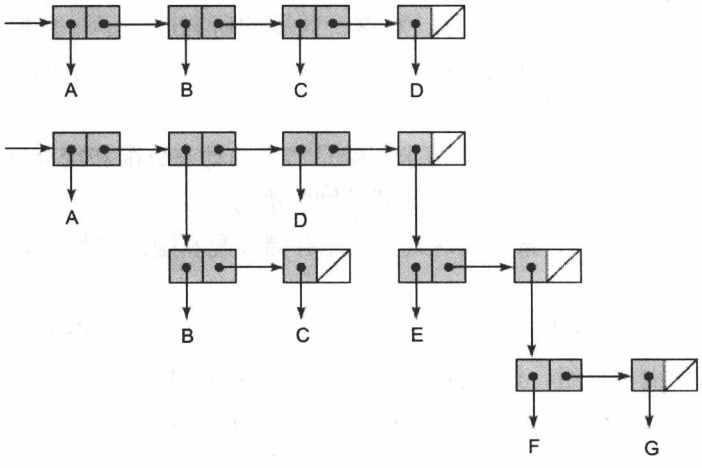


图 2-2 两个 LISP 表的内部表示

2.4.3.2 函数式程序设计中的过程

LISP 设计为函数式程序设计语言。在纯函数式程序中，所有的运算都是以将函数作用于参数的方式来实现的。在函数式语言的程序中，没有命令式语言程序中随处可见的赋值语句和变量，而且重复的过程可以通过递归的函数调用来表示，不需要迭代(循环)。这些函数式程序设计的基本原理，使它与命令式语言的程序设计极不相同。

2.4.3.3 LISP 的语法

LISP 与命令式语言有很大不同，因为它是函数式程序设计语言，LISP 程序看起来与

Java 或 C++等语言的程序有很大差别。例如，Java 的语法是英语和代数的复杂混合，而 LISP 的语法则是简洁的典型，程序代码和数据有完全相同的形式：以圆括号为边界的表。考虑下面的表：

```
(A B C D)
```

按数据来解释时，它是一个有 4 个元素的表，按代码来解释时，它是将名为 A 的函数作用于 3 个参数 B、C 和 D。

2.4.4　评价

LISP 完全垄断人工智能领域的应用长达四分之一个世纪。导致 LISP 的效率极低的许多因素已经去除。目前 LISP 的许多实现是编译执行的，执行编译后的代码比通过解释器执行源代码要快得多。除了在人工智能领域的成功之外，LISP 还是函数式程序设计的先锋，函数式程序设计已被证明是程序设计语言研究的一个活跃领域。如第 1 章所述，许多程序设计语言研究者认为，与使用命令式语言的过程式程序设计相比，函数式程序设计在软件开发方式上要好得多。

下面是 LISP 程序的一个例子：

```
;  LISP Example function
;  The following code defines a LISP predicate function
;  that takes two lists as arguments and returns True
;  if the two lists are equal, and NIL (false) otherwise
   (DEFUN equal_lists (lis1 lis2)
    (COND
      ((ATOM lis1) (EQ lis1 lis2))
      ((ATOM lis2) NIL)
      ((equal_lists (CAR lis1) (CAR lis2))
                  (equal_lists (CDR lis1) (CDR lis2)))
      (T NIL)
    )
)
```

2.4.5　LISP 的两种后代语言

目前有两种 LISP 语言得到广泛使用：Scheme 和 Common LISP。下面的小节对它们做简要介绍。

2.4.5.1　Scheme

Scheme 语言在 20 世纪 70 年代中期诞生于 MIT(Dybvig，2003)，其特点是小巧，只使用静态作用域(在第 5 章中介绍)，将函数作为第一类实体来处理。作为第一类实体，Scheme 的函数可以赋值给变量，可以作为参数传递，可以作为函数调用的值返回，也可以作为表的元素。LISP 的早期版本并不支持所有这些功能，也不使用静态作用域。

作为一种语法语义简单的小型语言，Scheme 很适于教学应用，例如函数式程序设计的

课程和程序设计的一般性介绍。第 15 章对 Scheme 有一些细致的介绍。

2.4.5.2 Common LISP

从 20 世纪 70 年代到 80 年代早期，人们开发并使用了大量不同的 LISP 语言，这导致了众所周知的移植性问题。Common LISP(Graham，1996)正是为了解决这一问题而开发的。Common LISP 的设计将 20 世纪 80 年代早期开发的多种 LISP 语言(其中包括 Scheme)的特性组合到一种语言中。作为这样一种混合产品，Common LISP 又大又复杂。然而，它的基础仍然是纯 LISP，因此它的语法、基本功能和基本性质都来自于纯 LISP。

Common LISP 认识到动态作用域的灵活性和静态作用域的简单性，支持这两类作用域。变量默认的作用域是静态的，但是将变量声明为 special，就可以将变量的作用域变为动态的。

Common LISP 有大量数据类型和结构，包括记录、数组、复数和字符串。它还有一种将模块化的函数集和数据打包的形式，提供访问控制。

Common LISP 将在第 15 章中深入介绍。

2.4.6 相关语言

ML(MetaLanguage，元语言；Ullman，1998)最初由爱丁堡大学的 Robin Milner 于 20 世纪 80 年代设计，是一个名为可计算函数逻辑(Logic for Computable Functions，LCF；Milner 等，1990)的程序验证系统的元语言。ML 本质上是一种函数式语言，但它也支持命令式程序设计。与 LISP 和 Scheme 不同的是，ML 中每个变量和表达式的类型都可以在编译时确定。类型是与对象相关，而不是与名称相关，表达式的类型由表达式的上下文来推断。

与 LISP 和 Scheme 不同，ML 并不使用由 lambda 表达式发展而来的使用圆括号的函数式语法。相反，ML 的语法与命令式语言(如 Java 和 C++)比较像。

Miranda 语言是由英国坎特伯雷市肯特大学的 David Turner 于 20 世纪 80 年代早期开发的(Turner，1986)。Miranda 语言部分基于 ML、SASL 和 KRC 语言。Haskell 语言(Hudak 和 Fasel，1992)主要基于 Miranda 语言，与后者一样，它也是纯函数式语言，没有变量和赋值语句。Haskell 的另一独特属性是惰性求值，这意味着表达式在需要其值时才计算，这导致该语言拥有一些令人吃惊的功能。

Caml (Cousineau 等，1998)及其支持面向对象程序设计的语言版本 OCaml(Smith, 2006)派生于 ML 和 Haskell。最后，F#是直接基于 OCaml 的一种相当新的类型化语言。F#(Syme 等，2010)是一种.NET 语言，可以直接访问整个.NET 库，也可以与任何其他.NET 语言平滑地交互操作。F#支持函数式程序设计和过程式程序设计，还完全支持面向对象的程序设计。

ML、Haskell 和 F#将在第 15 章中介绍。

2.5　迈向成熟的第一步：ALGOL 60

ALGOL 60 对后来的程序设计语言有重大的影响，因此总是作为语言历史学习的重中之重。

2.5.1　历史背景

ALGOL 60 是在人们为科学应用设计通用语言的工作中产生的。1954 年末，Laning 和 Zierler 的代数系统已经运行了一年多，Fortran 语言的第一份报告也已经发表。Fortran 语言在 1957 年成为现实，其他一些高级语言也正在开发中。其中最有名的是由卡内基工学院的 Alan Perlis 设计的 IT 语言，以及 UNIVAC 计算机上的 MATH-MATIC 语言和 UNICODE 语言。大量的新增语言使用户间的交流变得困难，而且新语言都是围绕单个体系结构开发的，有些语言用于 UNIVAC 计算机，有些语言用于 IBM 700 系列计算机。针对这种计算机专用语言大量增多的情况，美国一些主要的计算机用户组，包括 SHARE(IBM 科学用户组)和 USE(UNIVAC 科学交流，大规模的 UNIVAC 科学用户组)，在 1957 年 5 月 10 日向计算机协会(Association for Computing Machinery，ACM)提交了一份申请，建议组成一个委员会来研究和开发一种与计算机无关的科学用程序设计语言。虽然 Fortran 语言是一个选择，但是由于当时 Fortran 语言由 IBM 独有，因此不可能成为一种通用的语言。

在此前的 1955 年，GAMM(“应用数学和力学社团”的德文缩写)成立了一个委员会，来设计一种通用的与计算机无关的算法语言。开发这一新语言的愿望部分源于欧洲人害怕 IBM 占据主导地位。然而到了 1957 年后期，美国出现了多个高级语言，GAMM 小组委员会认为应该扩大工作组，把美国人也包括进来，就给 ACM 发了一封邀请信。1958 年 4 月，在 GAMM 的 Fritz Bauer 向 ACM 做了正式申请之后，两个小组正式同意加入一个联合的语言设计项目。

2.5.2　早期设计过程

GAMM 和 ACM 各派了 4 人参加第一次设计会议。这个会议于 1958 年 5 月 27 日至 6 月 1 日在苏黎世举行，为新语言定了如下目标：

- 语言的语法应该与标准的数学记法尽可能接近，写出来的程序应该不需要更多的解释就可阅读。
- 应该能够在出版物中用这种语言描述算法。
- 用新语言写出的程序必须可以由机器自动翻译为机器语言。

第一个目标表明新的语言将用于科学应用的程序设计，这是当时主要的计算机应用领域。第二个目标对计算机行业来说是全新的。第三个目标显然是任何程序设计语言所必需的。

苏黎世会议成功地催生了一种实现所声明目标的语言，但是语言的设计过程需要在个人之间和大西洋两岸之间进行无数的折中妥协。从某种意义上说，重大问题上的折中没有地域分歧方面的折中那么多，采用逗号(欧洲方式)还是点号(美国方式)来表示小数点的问题

就是一个例子。

2.5.3 ALGOL 58 概述

在苏黎世会议上设计的语言命名为国际算法语言(International Algorithmic Language，IAL)。在设计时有人提出将该语言命名为 ALGOL，即算法语言，但这个名字被拒绝了，因为它体现不出委员会的国际性。但是在此后的第二年，该语言的名字又改为 ALGOL，后来该语言称为 ALGOL 58。

ALGOL 58 在很多方面都是 Fortran 语言的后代，这是很自然的。它对许多 Fortran 语言的特性进行了推广，并添加了一些新的结构和概念。有些特性的推广是为了实现“使语言不限于任何特定的计算机”这一目标，有些则试图使语言更灵活更强大。在这些努力之下，简单和优雅罕见地融合在了一起。

ALGOL 58 形式化了数据类型的概念，虽然只有非浮点型的变量需要显式声明。它加入了复合语句的概念，后来的大多数语言都包含了复合语句。所推广的一些 Fortran 语言特性包括：标识符可以是任意长度，而 Fortran I 限制标识符长度为 6 个或更少字符；数组维数可以是任意多，而 Fortran I 限制数组维数不超过 3；可以由程序员指定数组下界，而 Fortran 语言把数组下界隐式规定为 1；允许嵌套选择语句，这在 Fortran I 中是不可以的。

ALGOL 58 以一种相当不寻常的方式确定了它的赋值运算符。Zuse 在 Plankalkül 语言中采用如下形式来表示赋值：

```
表达式=>变量
```

虽然 Plankalkül 语言没有发布，但 ALGOL 58 委员会中的一些欧洲成员对该语言很熟悉。委员会尝试用 Plankalkül 语言的赋值形式，但是由于字符集是有限制的[4]，大于号改成了冒号。后来主要因为美国人的坚持，整个语句变成了 Fortran 语言的形式：

```
变量:=表达式
```

欧洲人更喜欢相反的形式，但这是颠倒的 Fortran 形式。

2.5.4 对 ALGOL 58 报告的响应

1958 年 12 月，ALGOL 58 报告(Perlis 和 Samelson，1958)的发表引起了极其热烈的反响。在美国，这种新的语言更被看作是一些程序设计语言思想的集成，而不仅仅是一种通用的标准语言。实际上，ALGOL 58 报告并不表示语言已经开发完成，而只是一个用于国际化讨论的草案。尽管如此，有三个主要的设计和实现团队以这份报告作为工作的基础。在密歇根大学诞生了 MAD 语言(Arden 等，1961)，美国海军电子集团(U.S. Naval Electronics Group)开发了 NELIAC 语言(Huskey 等，1963)，系统开发公司(System Development Corporation)设计并实现了 JOVIAL 语言(Shaw，1963)。JOVIAL 是“国际代数语言的 Jules

4. 那时的打孔卡片不包含大于号。

版本(Jules' Own Version of the In-ternational Algebraic Language)”的缩写，是唯一基于 ALGOL 58 并得到广泛应用的语言(Jules 是 JOVIAL 设计者之一 Jules I. Schwartz)。JOVIAL 能够得到广泛应用，是因为它作为美国空军官方的科学计算语言长达四分之一个世纪。

美国的其他计算机社团对新语言没那么友好。起初，IBM 和它主要的科学用户组 SHARE 似乎很欢迎 ALGOL 58。IBM 在报告发表后很快就开始实现该语言，SHARE 成立了一个小组委员会 SHARE IAL 来研究该语言。这个小组委员会后来建议 ACM 将 ALGOL 58 标准化，建议 IBM 为所有 700 系列的计算机实现 ALGOL 58。然而这种热情没有持续多久，到了 1959 年春天，IBM 和 SHARE 因为拥有 Fortran 语言的经验，觉得开发和使用新一代的编译器、训练并说服用户使用新语言等推广新语言的工作非常痛苦，花费巨大。到了 1959 年中期，由于 IBM 和 SHARE 在 Fortran 语言方面既得的利益，他们决定保留 Fortran 语言作为 IBM 700 系列计算机上的科学计算语言，从而抛弃了 ALGOL 58。

2.5.5　ALGOL 60 的设计过程

在 1959 年间，欧洲和美国对 ALGOL 58 展开了激烈的辩论。在欧洲 *ALGOL Bulletin* 和 *Communications of the ACM* 杂志上刊登了大量增改建议。1959 年最重要的事件之一是在国际信息处理大会(International Conference on Information Processing)上苏黎世委员会做了工作报告，而 Backus 介绍了描述程序设计语言语法的新方法，这一方法后来称为 BNF(巴科斯-诺尔范式，Backus-Naur form)。BNF 将在第 3 章中详细介绍。

1960 年 1 月在巴黎召开了第二次 ALGOL 会议。这次会议的工作是讨论已正式提交的 80 项提案。丹麦的 Peter Naur 虽然不是苏黎世委员会的成员，但他积极参与了 ALGOL 的开发，是他创立并出版了 *ALGOL Bulletin* 杂志。他花了大量时间研究了 Backus 介绍 BNF 的论文，认定应该使用 BNF 来正式描述 1960 年会议的成果。在对 BNF 做了一些小改动后，他用 BNF 描述了这一提议的新语言，并在会议开始时将它分发给 1960 年会议小组的成员。

2.5.6　ALGOL 60 概述

虽然 1960 年的会议只有 6 天，但是对 ALGOL 58 做了重大修改。其中最重要的一些新发展是：

- 引入了块结构的概念，这将允许程序员引入新的数据环境或作用域，将程序的某些部分局部化。
- 允许用两种不同的方法向子程序传递参数：按值传递和按名称传递。
- 允许递归过程。ALGOL 58 的说明在这一点上并不明确。值得注意的是，虽然对于命令式语言来说递归是新的特性，但是 LISP 在 1959 年就提供递归函数了。
- 允许栈动态数组。栈动态数组是指数组的下标范围可用变量来指定，从而在程序执行到数组声明处时，数组的大小在分配数组的存储空间时确定。栈动态数组将在第 6 章中详细介绍。

人们提出了一些对语言成功或失败可能造成巨大影响的特性，但被否决了。其中非常重

要的特性包括格式化的输入输出语句，否决它是因为该语言特性被认为是与计算机相关的。

ALGOL 60 的报告于 1960 年 5 月发表(Naur，1960)。在语言描述中仍然存在一些有歧义的地方，计划于 1962 年 4 月在罗马召开第三次会议，讨论这些问题。在这个会议上，委员会只解决问题，不允许向语言添加新的特性。这个会议的成果以“算法语言 ALGOL 60 的修订报告”发表(Backus 等，1963)。

2.5.7 评价

从某些方面来说，ALGOL 60 获得了巨大的成功；从另外的方面来说，它遭受了痛苦的失败。它的成功在于几乎立刻成为计算机文献中表达算法唯一被接受的正式方式，并且持续了 20 多年。自 1960 年以来设计的每一种命令式语言都得益于 ALGOL 60。实际上，大多数语言都是它直接或间接的后代，例如 PL/I、SIMULA 67、ALGOL 68、C、Pascal、Ada、C++、Java 和 C#。

ALGOL 58/ALGOL 60 的设计工作包含了一系列第一。这是第一次由国际组织来尝试设计程序设计语言；它是第一种以计算机无关为目标设计的语言；它也是第一种正式描述语法的语言。BNF 形式体系的成功应用开拓了计算机科学的一些重要领域：形式语言、语法分析理论，以及基于 BNF 的编译器设计。最后，ALGOL 60 的结构影响了计算机体系结构。最引人注目的一个例子是，它的一种扩展语言被用作一系列大规模计算机的系统语言：Burroughs B5000、B6000 和 B7000 计算机，它们设计有硬件栈，从而高效地实现语言中的块结构和递归子程序。

另一方面，ALGOL 60 在美国从来没有得到广泛应用。即使在欧洲，虽然比在美国应用广泛，但也从未占据过统治地位。它不被接受有一些原因，其中一点是 ALGOL 60 的部分特性太灵活了，理解起来很困难，实现效率较低。一个最好的例子就是第 9 章介绍的向子程序传递参数时按名称传递的方法。Rutishauser 在 1967 年的话可以证明 ALGOL 60 实现困难：“即使有也只有极少数完整地实现了 ALGOL 60 语言(Rutishauser，1967，第 8 页)。”

这个语言没有输入和输出语句，是它没有得到广泛接受的另一个重要原因。与实现相关的输入/输出使程序难以移植到其他计算机上。

与 ALGOL 60 相关的对计算机科学作出重要贡献的 BNF，也是该语言没有广泛应用的一个原因。虽然现在认为 BNF 是一种简单而优雅的语法描述方法，但是在 1960 年，它看起来又奇怪又复杂。

最后，虽然还有许多其他问题，但 Fortran 语言使用者的固守和缺少 IBM 的支持，可能是 ALGOL 60 没有得到广泛使用的最重要因素。

ALGOL 60 语言的说明一直存在歧义和模糊之处，从这个意义上说，ALGOL 60 的工作从未真正完成过(Knuth，1967)。

下面是 ALGOL 60 程序的一个例子。

```
Comment ALGOL 60 Example Program
 Input:  An integer, listlen, where listlen is less than
         100, followed by listlen-integer values
```

```
 Output: The number of input values that are greater than
         the average of all the input values ;
begin
  integer array intlist [1:99];
  integer listlen, counter, sum, average, result;
  sum := 0;
  result := 0;
  readint (listlen);
  if (listlen > 0) ∧ (listlen < 100) then
    begin
comment Read input into an array and compute the average;
    for counter := 1 step 1 until listlen do
      begin
      readint (intlist[counter]);
      sum := sum + intlist[counter]
      end;
comment Compute the average;
    average := sum / listlen;
comment Count the input values that are > average;
    for counter := 1 step 1 until listlen do
      if intlist[counter] > average
        then result := result + 1;
comment Print result;
    printstring("The number of values > average is:");
    printint (result)
    end
  else
    printstring ("Error—input list length is not legal";
end
```

2.6　商务记录的计算机化：COBOL 语言

COBOL 语言的故事从某种意义来说正好与 ALGOL 60 相反。虽然 COBOL 语言的应用比任何其他语言都更为广泛，但是它对后来的语言没有什么影响，除了 PL/I 之外。它可能仍然是应用最广泛的语言[5]，虽然很难用什么办法来确定这一点。COBOL 语言的影响这么小，最重要的原因可能是自从 COBOL 语言出现后，没有什么人试图为商务应用设计一种新的语言，这部分归因于 COBOL 语言的功能很好地满足了应用领域的需求。另一个原因是过去 30 年间，大量增长的商务计算只是在小型商务领域。在这些商务应用中几乎没什么软件开发，所使用的大部分软件都是作为现成的套装软件购买的，用于各种常见的商务应用。

5. 20 世纪 90 年代晚期，在关于 2000 年问题的研究中，人们估计在曼哈顿 22 平方英里的范围内约有 8 亿行 COBOL 程序在使用之中。

2.6.1 历史背景

COBOL 语言的起源在某种意义上与 ALGOL 60 有些相似，即这种语言是由人们组成委员会，并在相对较短的时间内开会讨论设计出来的。1959 年，商务计算的状态与几年前设计 Fortran 语言时科学计算的状态相似。一种面向商务应用的编译语言 FLOW-MATIC 在 1957 年开发出来，但是它属于制造商 UNIVAC，而且是为该公司的计算机而设计的。美国空军使用另一种语言 AIMACO，但它只是 FLOW-MATIC 的一个差别不大的变种。IBM 为商务应用设计了一种程序设计语言 COMTRAN(商务翻译语言)，但还没有实现。另有一些其他的语言设计项目还在计划中。

2.6.2 FLOW-MATIC 语言

FLOW-MATIC 的起源至少值得简要介绍一下，因为它是 COBOL 语言的主要祖先。在 1953 年 12 月，Remington-Rand UNIVAC 公司的 Grace Hopper 写了一份实际上是预言的提案。提案建议："数学程序应该用数学记法来写，数据处理程序应该用英语语句来写(Wexelblat，1981，第 16 页)。"遗憾的是，在 1953 年不可能使非程序员相信计算机可以理解英语单词。直到 1955 年，一份相似的提案才有了获得 UNIVAC 管理层资助的希望，即使到那时，也是用原型系统来做最后的说服工作。这个说服过程包括编译并运行一个小程序，先是用英语关键字，再用法语关键字，然后用德语关键字。这个演示令 UNIVAC 管理层印象深刻，成为他们同意 Hopper 提案的主要原因。

2.6.3 COBOL 语言的设计过程

第一次关于商务应用通用语言的正式会议由美国国防部资助，于 1959 年 5 月 28 日和 29 日在五角大楼举行(苏黎世 ALGOL 会议之后整一年)。会议小组一致同意该语言命名为 CBL(通用商务语言)，它应该具有以下总体性特点：大部分人同意该语言应尽可能使用英语，但少数人提出应更多地使用数学记法。该语言必须易于使用，甚至可以牺牲一些性能为代价，从而扩大能够在计算机上编程的人员基数。除了使语言易于使用外，人们还相信，管理人员只要说英语就能够阅读程序。最后，语言的设计不应由于实现问题而受到过分的限制。

会议主要关心的问题之一是这一通用语言的开发应该尽快开始，因为有许多开发新的商务语言的工作已经在进行中。除了已有的一些语言外，美国无线电公司和 Sylvania 公司都在开发他们自己的商务应用语言。很显然，通用语言开发的时间越长，就越难以得到广泛使用。因此，会议决定应快速对已有语言进行研究。针对这一任务，成立了短期委员会(Short Range Committee)。

会议很早就决定将语言的语句分为两类：数据说明语句和可执行运算语句，这两类语句在程序中位于不同位置。短期委员会的争论之一在于语言是否要包含下标。许多委员会成员认为，下标对于处理数据人员来说太复杂了，这些人应该不喜欢数学记法。类似的争论还包括是否应包含算术表达式。短期委员会最终的报告于 1959 年 12 月完成，所描述的

语言后来称为 COBOL 60。

COBOL 60 的语言说明在 1960 年 4 月由政府印务局(Government Printing Office)出版(Department of Defense，1960)，称为“初版”。1961 年和 1962 年都出版了修改版(Department of Defense，1961，1962)。该语言在 1968 年由美国国家标准学会(American National Standards Institute，ANSI)实现了标准化，后来的三个修订版分别于 1974 年、1985 年和 2002 年由 ANSI 实现了标准化。该语言持续发展到今天。

2.6.4　评价

COBOL 语言创造了一些新的概念，有些概念最终出现在其他语言中。例如，COBOL 60 的 DEFINE 动词是第一个高级语言的宏结构。更重要的是，最早出现于 Plankalkül 语言的分层数据结构(记录)在 COBOL 语言中第一次实现。自此之后，大多数命令式语言的设计中都包含记录。由于 COBOL 语言允许使用长名(最多 30 个字符)和连字符，它也是第一个允许名称有真正含义的语言。

总体来说，数据部分是 COBOL 语言设计的强项，而过程部分相对较弱。每一变量是在数据部分详细定义的，包括十进制位数和默认的小数点位置。文件记录和输出到打印机的行也进行了这样的详细描述，这使 COBOL 语言非常适用于打印会计报表。也许最初的过程部分最重要的缺点是没有函数，1974 年标准版的 COBOL 之前的版本也不支持带参数的子程序。

我们对 COBOL 语言最终的评价是，它是第一种由美国国防部(Department of Defense，DoD)强制使用的程序设计语言。这一强制要求从 COBOL 语言最初开发时就有，因为 COBOL 语言并不是专为美国国防部设计的。尽管 COBOL 语言有许多优点，但是如果没有强制要求，可能也无法留存至今。COBOL 语言早期的编译器性能很差，使用代价很高，当然最终，编译器的效率提高了，计算机也变得更快更便宜，内存也大得多了。这些因素的共同作用，使 COBOL 语言在美国国防部内外都取得了成功。COBOL 语言的出现导致了会计电算化，从任何角度来看这都是一项重要的革命。

以下是 COBOL 程序的一个例子。这个程序从文件 BAL-FWD-FILE 中读取数据，该文件包含一些物品的库存信息，其中每一项物品的记录都包括现有数量(BAL-ON-HAND)和物品的再订货点(BAL-REORDER-POINT)。再订货点是现有物品数量的临界值，数量低于这一数值后，必须再次订货。这个程序生成一个必须再次订货的物品清单，存在 REORDER-LISTING 文件中。

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PRODUCE-REORDER-LISTING.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. DEC-VAX.
OBJECT-COMPUTER. DEC-VAX.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
```

```
    SELECT BAL-FWD-FILE ASSIGN TO READER.
    SELECT REORDER-LISTING ASSIGN TO LOCAL-PRINTER.

DATA DIVISION.
FILE SECTION.
FD  BAL-FWD-FILE
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 80 CHARACTERS.

01  BAL-FWD-CARD.
02  BAL-ITEM-NO            PICTURE IS 9(5).
02  BAL-ITEM-DESC          PICTURE IS X(20).
02  FILLER                 PICTURE IS X(5).
02  BAL-UNIT-PRICE         PICTURE IS 999V99.
02  BAL-REORDER-POINT      PICTURE IS 9(5).
02  BAL-ON-HAND            PICTURE IS 9(5).
02  BAL-ON-ORDER           PICTURE IS 9(5).
02  FILLER                 PICTURE IS X(30).

FD  REORDER-LISTING
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 132 CHARACTERS.

01  REORDER-LINE.
02  RL-ITEM-NO             PICTURE IS Z(5).
02  FILLER                 PICTURE IS X(5).
02  RL-ITEM-DESC           PICTURE IS X(20).
02  FILLER                 PICTURE IS X(5).
02  RL-UNIT-PRICE          PICTURE IS ZZZ.99.
02  FILLER                 PICTURE IS X(5).
02  RL-AVAILABLE-STOCK     PICTURE IS Z(5).
02  FILLER                 PICTURE IS X(5).
02  RL-REORDER-POINT       PICTURE IS Z(5).
02  FILLER                 PICTURE IS X(71).

WORKING-STORAGE SECTION.
01  SWITCHES.
    02CARD-EOF-SWITCH   PICTURE IS X.
01  WORK-FIELDS.
    02AVAILABLE-STOCK   PICTURE IS 9(5).

PROCEDURE DIVISION.
000-PRODUCE-REORDER-LISTING.
    OPEN INPUT BAL-FWD-FILE.
    OPEN OUTPUT REORDER-LISTING.
    MOVE "N" TO CARD-EOF-SWITCH.
   PERFORM 100-PRODUCE-REORDER-LINE
       UNTIL CARD-EOF-SWITCH IS EQUAL TO "Y".
   CLOSE BAL-FWD-FILE.
   CLOSE REORDER-LISTING.
   STOP RUN.

100-PRODUCE-REORDER-LINE.
    PERFORM 110-READ-INVENTORY-RECORD.
```

```
    IF CARD-EOF-SWITCH IS NOT EQUAL TO "Y"]
        PERFORM 120-CALCULATE-AVAILABLE-STOCK
        IF AVAILABLE-STOCK IS LESS THAN BAL-REORDER-POINT
            PERFORM 130-PRINT-REORDER-LINE.

110-READ-INVENTORY-RECORD.
    READ BAL-FWD-FILE RECORD
        ATEND
            MOVE "Y" TO CARD-EOF-SWITCH.

120-CALCULATE-AVAILABLE-STOCK.
ADD BAL-ON-HAND BAL-ON-ORDER
    GIVING AVAILABLE-STOCK.

130-PRINT-REORDER-LINE.
    MOVE SPACE                TO REORDER-LINE.
    MOVE BAL-ITEM-NO          TO RL-ITEM-NO.
    MOVE BAL-ITEM-DESC        TO RL-ITEM-DESC.
    MOVE BAL-UNIT-PRICE       TO RL-UNIT-PRICE.
    MOVE AVAILABLE-STOCK      TO RL-AVAILABLE-STOCK.
    MOVE BAL-REORDER-POINT    TO RL-REORDER-POINT.
    WRITE REORDER-LINE.
```

2.7　分时处理的开始：BASIC 语言

BASIC 语言(Mather 和 Waite，1971)是又一种广泛使用但没有得到多少重视的程序设计语言。与 COBOL 语言一样，它大大地被计算机科学家忽略了，也像 COBOL 语言那样，它最早的版本一点儿也不优美，只包含很少的控制语句。

在20世纪70年代晚期和80年代早期，BASIC语言在微机上非常流行，这源自于BASIC语言的两个主要特点。它易于初学者学习，特别是那些不是为了做科学研究的人，它的小型方言可以在内存很少的计算机上实现[6]。随着微机的性能提升和其他语言的实现，BASIC语言没落了下去。随着 20 世纪 90 年代早期 Visual Basic(Microsoft，1991)的出现，BASIC语言又再次快速流行起来。

2.7.1　设计过程

BASIC(初学者通用符号指令码)语言最初是由新罕布什尔州 Dartmouth 学院(现在为Dartmouth 大学)的两个数学家 John Kemeny 和 Thomas Kurtz 设计的，他们在 20 世纪 60 年代早期开发了 Fortran 和 ALGOL 60 多个方言的编译器。他们的理科学生在学习中使用这些语言基本上没有什么困难，但是 Dartmouth 基本上是一所文科学院，理科和工科学生大约只占学生总体的 25%。1963 年春，他们决定为文科学生特别设计一种新的语言，这一新语言将使用终端来访问计算机。该系统的目标是：

(1) 必须易于非理科学生的学习和使用。

6. 一些早期的微机采用 BASIC 语言解释器，只占用 4096 字节的 ROM。

(2) 必须令人愉快且友好。

(3) 必须使学生能很快完成家庭作业。

(4) 必须允许自由和私人的访问。

(5) 必须更重视用户的时间而不是计算机时间。

最后一项目标确实是一个革命性的概念，其部分原因是，他们相信随着时间的推移，计算机将会便宜很多很多，事实确实如此。

第二、第三、第四项目标加在一起，导致了 BASIC 语言的分时性。在 20 世纪 60 年代早期，大量同时使用计算机的用户只有通过终端进行个人的访问操作，才能实现这些目标。

1963 年夏天，Kemeny 以远程访问 GE 225 计算机的方式，着手开发 BASIC 语言第一版的编译器。1963 年秋天，BASIC 语言操作系统的设计和编码也开始了。1964 年 5 月 1 日凌晨 4 时，用分时 BASIC 语言编写的第一个程序被输入计算机并运行起来。6 月，系统的终端数目增加到 11 个，到了秋天，这一数目上涨到了 20。

2.7.2 语言概述

BASIC 语言最初的版本非常小，而且奇怪的是，它是非交互的，正在运行的程序无法从用户那里获取输入数据。程序输入后被编译，以一种面向批处理的方式运行。最初的 BASIC 语言只有 14 种不同的语句类型和 1 种数据类型——浮点类型。由于设计者认为，目标用户中没什么人能理解整数类型和浮点类型的差别，因此把这一类型称为“数”。总的来说，它是一种功能非常有限的语言，但学起来非常简单。

2.7.3 评价

最初的 BASIC 语言最重要的一点是，在通过连接到远程计算机的终端来使用的语言中，它是第一个得到广泛使用的语言[7]。那时才刚刚出现计算机终端，之前大多数程序都是通过穿孔的卡片或纸带输入计算机的。

BASIC 语言的很多设计都来自于 Fortran 语言，也受到 ALGOL 60 语法的些微影响。后来它在很多方面有所发展，但很少或没有标准化。美国国家标准学会(American National Standards Institute，ANSI)发布了一个最小 BASIC 语言标准(Minimal BASIC standard；ANSI，1978b)，但这只体现了语言特性中最最基本的部分。实际上，最初版本的 BASIC 语言与最小 BASIC 语言非常相似。

虽然看起来很令人吃惊，但数字设备公司(Digital Equipment Corporation)在 20 世纪 70 年代，真的使用一个非常精细的 BASIC 语言版本 BASIC-PLUS，为 PDP-11 微机编写了最大的操作系统 RSTS 的许多内容。

对 BASIC 语言的批评之一是，所写出来的程序结构太差。根据第 1 章介绍的评价标准，特别是可读性和可靠性，这种语言确实相当糟糕，显然它的早期版本并不打算也不应该用于编写任何大规模的重要程序。后来的版本则更适用于这样的任务。

---

7. LISP 最初也是通过终端使用的，但它在 20 世纪 60 年代早期没有得到广泛的使用。

20 世纪 90 年代 BASIC 语言的复苏是因为 Visual BASIC(VB)的产生。VB 的广泛使用主要因为它提供了简单的方式来构建图形用户界面(GUI)，因此命名为 VisualBASIC。Visual Basic.NET(简称为 VB.NET)是 Microsoft .NET 语言中的一种，虽然它从 VB 发展而来，但迅速取代了 VB。也许 VB 和 VB.NET 之间最重要的差异是 VB.NET 全面支持面向对象的程序设计。

以下是 BASIC 语言程序的一个例子。

```
REM  BASIC Example Program
REM  Input:   An integer, listlen, where listlen is less
REM           than 100, followed by listlen-integer values
REM  Output:  The number of input values that are greater
REM           than the average of all input values
  DIM intlist(99)
  result = 0
  sum = 0
  INPUT listlen
  IF listlen > 0 AND listlen < 100 THEN
REM  Read input into an array and compute the sum
    FOR counter = 1 TO listlen
      INPUT intlist(counter)
      sum = sum + intlist(counter)
    NEXT counter
REM  Compute the average
    average = sum / listlen
REM  Count the number of input values that are > average
    FOR counter = 1 TO listlen
      IFintlist(counter) > average
        THEN result = result + 1
    NEXT counter
REM  Print the result
    PRINT "The number of values that are > average is:";
          result
  ELSE
    PRINT "Error—input list length is not legal"
  END IF
END
```

2.8　满足所有人的需要：PL/I

PL/I 体现了设计一种可用于许多应用领域的语言的第一次大规模尝试。所有之前的语言和后来的大部分语言都专注于一个特定的应用领域，例如科学计算、人工智能或商务领域。

2.8.1　历史背景

像 Fortran 语言一样，PL/I 是由 IBM 开发的产品。到 20 世纪 60 年代早期为止，工业界的计算机用户已经分为两个相互独立且有较大差异的阵营：科学计算的用户和商务应用

的用户。在 IBM 看来，科学计算的程序员使用大型的 IBM 7090 计算机，或者是小型的 IBM 1620 计算机，他们大量地使用浮点数据和数组。Fortran 是他们主要使用的语言，虽然也用一些汇编语言。他们拥有自己的用户组 SHARE，很少接触商务应用的人。

在商务应用领域，人们使用大型的 IBM 7080 计算机或小型的 IBM 1401 计算机。他们需要十进制和字符串数据类型，以及精细而高效的输入输出功能。他们使用 COBOL 语言，虽然在 1963 年初 PL/I 的故事才刚刚开始，从汇编语言到 COBOL 语言的转变还远没有完成。这一类用户也有自己的用户组 GUIDE，并且很少接触科学计算的用户。

1963 年初，IBM 的规划人员意识到这种状况正在开始改变，这两个相隔很远的用户组以人们认为一定会产生问题的方式，正在向彼此靠近。科学家开始集中大量数据来处理，这些数据需要更复杂更高效的输入和输出功能。商务应用的人们开始使用回归分析来建立管理信息系统，这需要浮点数据和数组。已经开始出现这样的情况：计算机的安装需要两组不同的计算机和技术人员，支持两种不同的程序设计语言[8]。

这一认识很自然地产生了一个想法：设计一种通用计算机，既能进行浮点运算又能进行十进制算术运算，因此既能做科学计算又能做商务应用，这就产生了 IBM System/360 系列计算机的设想，同时也有了一种程序设计语言既能用于科学计算又能用于商务应用的想法。另外，支持系统编程和表处理的特性也加入进来。从而，这一新的语言将代替 Fortran、COBOL、LISP，以及汇编语言的系统应用。

2.8.2 设计过程

设计工作开始于 1963 年 10 月，当时 IBM 和 SHARE 成立了“SHARE Fortran 项目的高级语言开发委员会”。这个新的委员会迅速碰头，并成立了一个称为 3×3 委员会的子委员会，叫这个名字是因为子委员会有 3 名成员来自 IBM，有 3 名成员来自 SHARE。3×3 委员会每隔一周就集中 3 天或 4 天进行语言的设计。

如同 COBOL 语言的短期委员会，最初的设计计划在一个极其短的时间内完成。显然，在 20 世纪 60 年代早期人们普遍认为，不管语言设计工作量有多少，都可以在 3 个月内完成。PL/I 的第一个版本当时名为 Fortran VI，计划在 12 月之前完成，即在委员会成立之后不到 3 个月的时间内完成。委员会成功地以两个不同的理由为借口延长了期限，先是将期限延长到 1964 年 1 月，后来又延长到 1964 年 2 月。

最初的设计想法是新语言将成为 Fortran IV 的扩展，保持兼容性，但是这一目标很快就与 Fortran VI 这一名称一起放弃了。到了 1965 年，这种语言称为新程序设计语言(New Programming Language，NPL)。第一份 NPL 的报告是在 1964 年 3 月 SHARE 的会议上发表的，一份更完整的说明接着发表于 4 月，而实际的实现版本的说明在 1964 年 12 月(IBM，1964)由英国 IBM Hursley 实验室的编译器小组发表，这个小组被选定来实现语言。1965 年，语言的名称改为 PL/I，以避免与英国国家物理实验室(National Physical Laboratory，NPL)的名称相混淆。如果编译器是在英国以外的地方开发的，语言的名字就可能仍然是 NPL。

8. 当时，大型计算机的安装需要专门的硬件和系统软件维护人员。

2.8.3　语言概述

描述 PL/I 最贴切的一句话可能是，它包含了以下几种语言当时认为最优秀的部分：ALGOL 60(递归和块结构)、Fortran IV(单独编译并通过全局数据进行通信)和 COBOL 60(数据结构、输入/输出和报表生成功能)，还有一些扩展的新结构，都设法拼凑在了一起。由于 PL/I 现在不再流行，我们不会试图哪怕只是简要地介绍该语言的所有特性，甚至是最有争议的语言结构。相反，我们将简要介绍该语言对程序设计语言知识宝库的一些贡献。

PL/I 是第一种具有以下功能的程序设计语言：

- 程序可以创建并发执行的子程序。虽然这是一个很好的想法，但是在 PL/I 中没有很好地实现。
- 它可以检测并处理 23 类不同的异常(运行时错误)。
- 子程序可以递归使用，也能禁止这一功能，使非递归子程序的链接更快。
- 指针作为一种数据类型。
- 可以引用数组的横行，例如可以像一维数组那样引用矩阵的第三行。

2.8.4　评价

对 PL/I 的任何评价必须先了解设计工作的雄心壮志。回顾历史，当时认为这么多语言结构可以成功地组合在一起，显得很天真。但是，必须承认当时语言设计的经验很少。总的来说，PL/I 的设计是以这一前提为基础的：任何有用的、能够实现的语言结构都应该包含进来，没有充分考虑程序员如何理解和有效使用这么多的语言结构和特性。Edsger Dijkstra 在他的图灵奖演讲(Dijkstra，1972)中，对 PL/I 的复杂性做了最激烈的批评："我完全无法预见，当程序设计语言——请注意，这是我们基本的工具——本身的复杂度超出了我们的智力掌握，我们还如何能够将不断增长的程序牢牢置于智力掌控之下。"

除了规模庞大导致的复杂性问题之外，PL/I 还有一些现在认为设计得很差的语言结构，其中包括指针、异常处理和并发，但必须指出，这些语言结构在任何之前的语言中都没有出现过。

在用途方面，必须承认 PL/I 至少是部分成功的。在 20 世纪 70 年代，它大量地用于商务和科学应用中。在此期间，它也作为高校的教学工具广泛使用，主要是它的几个子集，如 PL/C(Cornell，1977)和 PL/CS(Conway 和 Constable，1976)。

下面是 PL/I 程序的一个例子：

```
/* PL/I PROGRAM EXAMPLE
 INPUT:  AN INTEGER, LISTLEN, WHERE LISTLEN IS LESS THAN
         100, FOLLOWED BY LISTLEN-INTEGER VALUES
OUTPUT:  THE NUMBER OF INPUT VALUES THAT ARE GREATER THAN
         THE AVERAGE OF ALL INPUT VALUES */
PLIEX: PROCEDURE OPTIONS (MAIN);
  DECLARE INTLIST (1:99) FIXED.
  DECLARE (LISTLEN, COUNTER, SUM, AVERAGE, RESULT) FIXED;
  SUM = 0;
```

```
  RESULT = 0;
  GET LIST (LISTLEN);
  IF (LISTLEN > 0) & (LISTLEN < 100) THEN
    DO;
/* READ INPUT DATA INTO AN ARRAY AND COMPUTE THE SUM */
    DO COUNTER = 1 TO LISTLEN;
      GET LIST (INTLIST (COUNTER));
      SUM = SUM + INTLIST (COUNTER);
    END;
/* COMPUTE THE AVERAGE */
    AVERAGE = SUM / LISTLEN;
/* COUNT THE NUMBER OF VALUES THAT ARE > AVERAGE */
    DO COUNTER = 1 TO LISTLEN;
      IF INTLIST (COUNTER) > AVERAGE THEN
        RESULT = RESULT + 1;
    END;
/* PRINT RESULT */
    PUT SKIP LIST ('THE NUMBER OF VALUES > AVERAGE IS:');
    PUT LIST (RESULT);
    END;
  ELSE
    PUT SKIP LIST ('ERROR-INPUT LIST LENGTH IS ILLEGAL');
  END PLIEX;
```

2.9 两种早期的动态语言：APL 和 SNOBOL

这一节的结构与本章其他小节不同，因为本节介绍的语言很不一样。APL 和 SNOBOL 对后来的主流语言都没有什么影响[9]。本书后面还会介绍 APL 语言的一些有趣特性。

从表现形式和用途上看，APL 和 SNOBOL 都很不一样，但是它们都有两个基本特征：动态类型和动态存储分配。在这两种语言中，变量本质上是没有类型的，变量在赋值时才具有类型，就是被赋的值的类型。变量的存储空间只在赋值时才分配，因为在此之前没有办法知道需要的存储空间的大小。

2.9.1 APL 语言的起源与特点

APL 语言(Brown 等，1988)是在 1960 年左右由 IBM 的 Kenneth E. Iverson 设计的，起初并没有想实现这种程序设计语言，只是用它作为描述计算机体系结构的工具。APL 最早是在名为 *A Programming Language* 的书(Iverson，1962)中介绍的，这也是它名字的由来。在 20 世纪 60 年代中期，IBM 开发出了 APL 语言最早的实现。

APL 语言具有大量功能强大的运算符，它们用许多符号来指定，这对实现者造成了困难。最初，使用 APL 语言的方法是通过 IBM 的打印终端，这些终端有特殊的打印球，提

9. 但是，它们对一些非主流语言有影响(J 语言基于 APL 语言，ICON 语言基于 SNOBOL 语言，AWK 语言部分基于 SNOBOL 语言)。

供语言所需的特殊的字符集。APL 语言具有这么多运算符的原因之一是因为它为数组提供了大量的单元运算。例如，任意矩阵的转置只需一个运算符即可实现。大量的运算符提供了极强的表达能力，但也使 APL 程序很难阅读，使人们认为 APL 是一种最适于用来“一次性”程序设计的语言：虽然用 APL 语言写起程序来很快，但是程序在使用过后就应该扔掉，因为程序难以维护。

APL 语言已经存在了将近 50 年，虽然并不广泛但今天仍在使用，而且在此期间变化不大。

2.9.2　SNOBOL 语言的起源与特点

SNOBOL(发音为“snowball”；Griswold 等，1971)语言是 20 世纪 60 年代早期由贝尔实验室的三个人 D. J. Farber、R. E. Griswold 和 I. P. Polonsky 设计的(Farber 等，1964)。它是专为文本处理而设计的。SNOBOL 语言的核心是一些用于字符串模式匹配的强大运算符。SNOBOL 语言早期的应用之一是编写文本编辑器。由于 SNOBOL 语言的动态特征，它比其他的语言慢，因此不再用于这样的程序了。不过 SNOBOL 语言仍然是一种有生命力且得到支持的语言，在一些不同的应用领域中用于各种文本处理任务。

2.10　数据抽象的开始：SIMULA 67

虽然 SIMULA 67 从未广泛使用，对当时的程序员和计算机界影响也很小，但它引入的一些语言结构使它具有重要的历史地位。

2.10.1　设计过程

在 1962 年到 1964 年间，两个挪威人 Kristen Nygaard 和 Ole-Johan Dahl 在挪威计算中心(Norwegian Computing Center，NCC)开发了 SIMULA I 语言。他们的专业兴趣主要是使用计算机进行仿真，但也做一些运筹学的研究。设计 SIMULA I 只是用于系统仿真，最早于 1964 年末在 UNIVAC 1107 计算机上实现。

SIMULA I 的实现一完成，Nygaard 和 Dahl 就开始致力于扩展这一语言，加入新的特性，修改一些现有的结构，使语言可用于非特定的应用。扩展的结果就是 SIMULA 67，其设计在 1967 年 3 月首次公开发布(Dahl 和 Nygaard，1967)。虽然 SIMULA 67 中的一些特性在 SIMULA I 中也有，但我们只介绍 SIMULA 67。

2.10.2　语言概述

SIMULA 67 是 ALGOL 60 的一个扩展，继承了它的块结构和控制语句。在系统仿真的应用上，ALGOL 60(以及当时的其他语言)主要的缺点在于子程序的设计。仿真需要子程序能够在先前停止的位置重新开始，支持这类控制的子程序称为协同程序，因为调用程序和被调用的子程序相互间是一定程度的对等关系，而不是大多数命令式语言中严格的主从关系。

为了在 SIMULA 67 中提供协同程序的支持，类结构产生了。这是一项重要的进展，

因为从它开始有了数据抽象的概念。而且，数据抽象提供了面向对象程序设计的基础。

有趣的是，直到 1972 年 Hoare 意识到类和数据抽象的关联(Hoare，1972)，数据抽象这一重要概念才发展起来，并被认为是类结构的特点。

2.11 正交设计：ALGOL 68

ALGOL 68 是语言设计中一些新思想的源头，其中一些思想后来被其他语言所采纳，因此我们在这里介绍 ALGOL 68，但它在欧洲和美国从未得到广泛的使用。

2.11.1 设计过程

在 1962 年的 ALGOL 60 修订报告发布之后，ALGOL 系列语言的发展并没有停止，虽然直到 6 年后才发布了下一个设计版本。最后开发的语言 ALGOL 68(van Wijngaarden 等，1969)与它的前身有很大的不同。

ALGOL 68 最有趣的创新之一是它的一个基本设计准则：正交性。请回顾第 1 章对正交性的介绍。正交性的应用产生了 ALGOL 68 中一些创新的特性，其中之一将在下一节介绍。

2.11.2 语言概述

ALGOL 68 应用正交性的一个重要结果是包含了用户自定义数据类型。先前的语言如 Fortran，只包含一些基本数据结构。PL/I 包含大量数据结构，因此难以学习和实现，但显然无法为任意的需求提供适合的数据结构。

ALGOL 68 数据结构的实现方案是提供一些基本类型和结构，并且允许用户将这些基本元素组合成大量不同的结构。对用户自定义数据类型的支持，在之后设计的所有主要的命令式语言中，都在某种程度上得到了延续。用户自定义数据类型很有用，因为它允许用户来设计极其适合特定问题的数据抽象。第 6 章将全方位地介绍数据类型。

ALGOL 68 在数据类型方面的另一个第一是，它引入了动态数组，第 5 章将之称为隐式堆动态。动态数组的声明并不指定下标范围，给动态数组赋值才会分配需要的存储空间。在 ALGOL 68 中，动态数组称为 flex 数组。

2.11.3 评价

ALGOL 68 包含了大量以前从未使用过的特性，它对正交性的应用虽然被一些人认为有些过度，但仍然是革命性的。

不过，ALGOL 68 犯了 ALGOL 60 的一个错误，这是它应用有限的一个重要因素：该语言是用一种优美、简单但不为人知的元语言来说明的。在人们阅读语言说明文档(van Wijngaarden 等，1969)之前，必须学习这一新的元语言，它称为 van Wijngaarden 文法，比 BNF 复杂得多。更糟糕的是，设计者发明了一些词来解释文法和语言。例如，将“关键字(keyword)”称为“指示符(indicant)”，将“子串提取(substring extraction)”称为“修剪

(trimming)”，将“子程序执行过程(process of a subprogram execution)”称为“非程序化强制(coercion of deproceduring)”，这种“强制”还可能是“温和的(meek)”或“固定的(firm)”或者其他说法。

将 PL/I 的设计和 ALGOL 68 的设计进行比较是很自然的事，因为它们都只存在了几年。ALGOL 68 的可写性在于正交性原则：一些基本概念和一些组合机制的无限使用；PL/I 的可写性在于包含了大量的固定结构。ALGOL 68 延续了 ALGOL 60 的优美、简单，而 PL/I 只是简单地将一些语言特性堆积在一起，以达到语言设计的目标。当然，我们必须考虑到 PL/I 的目标是为广泛的应用问题提供统一的工具，而 ALGOL 68 的目标只有一个：科学应用。

PL/I 的接受程度远高于 ALGOL 68，这主要归因于 IBM 的宣传工作，以及理解和实现 ALGOL 68 的困难。这两种语言的实现都很困难，但是 PL/I 有 IBM 提供开发编译器的资源，而 ALGOL 68 没有这样的赞助者。

2.12　ALGOL 系列语言的早期后代语言

所有命令式语言都有一些设计来自于 ALGOL 60 和/或 ALGOL 68。本节介绍 ALGOL 语言一些早期的后代语言。

2.12.1　为简单性而设计：Pascal 语言

2.12.1.1 历史背景

Niklaus Wirth(Wirth 的发音为“Virt”)是国际信息处理联合会(International Federation of Information Processing，IFIP)2.1 工作组的成员，该工作组成立于 20 世纪 60 年代中期，目的是继续 ALGOL 语言的开发。1965 年 8 月，Wirth 和 C. A. R.(“Tony”) Hoare 为这一工作作出了贡献，在 ALGOL 60(Wirth 和 Hoare，1966)的添加和修改上向工作组提交了一份相当保守的提案。工作组的大部分成员否决了这一提案，认为该提案对 ALGOL 60 的提高太少了，并提出了一个复杂得多的修订版本取而代之，该版本后来成了 ALGOL 68。基于该语言和说明该语言的元语言的复杂性，Wirth 和其他一些工作组成员认为不应该发布 ALGOL 68 的报告。这一看法后来证明是有一定道理的，因为计算机界发现 ALGOL 68 的文档以及该语言本身，确实很具挑战性。

Wirth 和 Hoare 的 ALGOL 60 版本叫做 ALGOL-W，是在斯坦福大学实现的，但主要只在一些大学中用作教学工具。ALGOL-W 的主要贡献是引入了传递参数的值-结果方法，以及用于多选的 case 语句。值-结果方法是 ALGOL 60 按名称传递方法的一种替代方法。第 9 章将介绍这两种方法。case 语句将在第 8 章中介绍。

Wirth 后来的主要设计工作仍然是基于 ALGOL 60 的，也是他最成功的工作：Pascal 语言[10]。Pascal 语言的定义最初发表于 1971 年(Wirth，1971)，该版本在实现过程中有一些

10. Pascal 语言是以 17 世纪法国哲学家、数学家 Blaise Pascal 的名字命名的，他在 1642 年发明了第一台机械加法机，此外他还有其他一些发明。

修改，在(Wirth，1973)中对此做了说明。现在通常认为，Pascal 语言的特性实际上来自于一些更早的语言，例如用户自定义数据类型是在 ALGOL 68 中引入的，case 语句是在 ALGOL-W 中引入的，Pascal 语言的记录与 COBOL 和 PL/I 中的记录相似。

2.12.1.2 评价

Pascal 语言最重大的影响在于程序设计的教学。1970 年，计算机科学、计算机工程和理学专业的大部分学生都以 Fortran 语言作为程序设计的入门，有些大学使用 PL/I、基于 PL/I 的语言和 ALGOL-W。到 20 世纪 70 年代中期，Pascal 语言已经成为最广泛的教学语言。这是很自然的，因为 Pascal 语言是专门为程序设计教学而设计的。直到 20 世纪 90 年代末，Pascal 语言才不再是大专院校中最普遍使用的程序设计教学语言。

由于 Pascal 设计为教学语言，所以它缺少一些对很多应用领域很重要的特性。最好的例子是无法写一个以不定长的数组作为参数的子程序，另一个例子是没有单独的编译能力。这些缺陷很自然地导致了许多非标准的方言的产生，例如 Turbo Pascal。

Pascal 语言在程序设计教学和其他应用上的广泛普及，主要在于简单性和表达性的出色结合。虽然 Pascal 语言中有一些不安全的因素，但它仍然是一种相对安全的语言，特别是与 Fortran 语言或 C 语言相比。到 20 世纪 90 年代中期，Pascal 语言在工业界和高校的普及度都开始下滑，这主要是由于 Modula-2、Ada 和 C++的兴起，这些语言包含 Pascal 语言中没有的特性。

下面是 Pascal 语言程序的一个例子。

```
{Pascal Example Program
 Input:  An integer, listlen, where listlen is less than
         100, followed by listlen-integer values
 Output: The number of input values that are greater than
         the average of all input values }
program pasex (input, output);
  type intlisttype = array [1..99] of integer;
  var
    intlist : intlisttype;
    listlen, counter, sum, average, result : integer;
  begin
  result := 0;
  sum := 0;
  readln (listlen);
  if((listlen > 0) and (listlen < 100)) then
    begin
{ Read input into an array and compute the sum }
    for counter := 1 to listlen do
      begin
      readln (intlist[counter]);
      sum := sum + intlist[counter]
      end;
{ Compute the average }
    average := sum / listlen;
```

```
{ Count the number of input values that are > average }
    for counter := 1 to listlen do
      if(intlist[counter] > average) then
        result := result + 1;
{ Print the result }
    Writeln ('The number of values > average is:',
             result)
    end { of the then clause of if (( listlen > 0 ... }
  else
    writeln ('Error-input list length is not legal')
end.
```

2.12.2　可移植的系统语言：C 语言

像 Pascal 语言一样，C 语言对于前面介绍的语言特性没什么贡献，但它的广泛使用已经有很长一段时间了。虽然最初设计是为了用于系统程序设计，但是 C 语言适用于很多不同的应用。

2.12.2.1　历史背景

C 语言的祖先包括 CPL、BCPL、B 和 ALGOL 68 语言。CPL 语言是 20 世纪 60 年代早期在剑桥大学开发的，BCPL 语言是一种简单的系统语言，也是在剑桥大学由 Martin Richards 在 1967 年开发的(Richards，1969)。

20 世纪 60 年代晚期，Ken Thompson 在贝尔实验室从事 UNLX 操作系统最早的相关工作，最初的版本是用汇编语言写的。在 UNIX 系统上实现的第一种高级语言是 B 语言，它是基于 BCPL 语言的，由 Thompson 在 1970 年设计和实现。

BCPL 和 B 都不是类型化的语言，是高级语言中的另类，虽然它们与 Java 这样的语言相比低级很多。非类型化意味着所有的数据都是机器字，这样虽然简单，但会导致许多复杂性和安全性问题，例如如何在表达式中指定是浮点类型而不是整数类型的算术运算的问题。在 BCPL 语言的一种实现中，在浮点运算的变量操作数前面放一个句号，前面没有句号的变量操作数就认为是整数。另一种方法是用别的符号来表示浮点运算。

由于存在这个问题和其他一些问题，人们又开发了一种基于 B 语言的新的类型化语言。这种语言最初命名为 NB，后来命名为 C，是 Dennis Ritchie 于 1972 年在贝尔实验室设计和开发的(Kernighan 和 Ritchie，1978)。C 语言部分通过 BCPL 语言、部分直接地受到 ALGOL 68 的影响，这从它的 for 和 switch 语句、赋值运算符，以及对指针的处理中可以看出来。

在 C 语言最初的 15 年中，唯一的“标准”是 Kernighan 和 Ritchie 的书(Kernighan 和 Ritchie，1978)[11]。在这段时间内，C 语言缓慢地发展着，不同的实现者加入了不同的特性。1989 年，ANSI 提出了 C 语言的正式版本(ANSI，1989)，包含了许多实现者已经加入到语言中的特性。1999 年更新了这个标准(ISO，1999)，新版本包含一些重大的变化，例如复合数据类型、布尔数据类型和 C++形式的注释(//)。1989 年的版本长期以来称为 ANSI C，

11. 这一语言常称为“K&R C”。

现在应该称为C89。1999年的版本称为C99。

2.12.2.2 评价

C语言有适当的控制语句和数据结构建立工具，使它可用于许多应用领域。它还有丰富的运算符，提供了极高的表达能力。

人们喜欢和不喜欢C语言最重要的原因之一是它缺少完整的类型检查。例如在C99之前的版本中，函数的参数不检查类型。喜欢C语言的人欣赏它的灵活性，不喜欢它的人认为它太不安全。20世纪80年代C语言的普及程度飞速发展，主要原因是它的编译器是广泛使用的UNIX操作系统的一部分。在UNIX系统中包含编译器，使程序员能够在许多不同类别的计算机上用到实际上免费且相当出色的编译器。

以下是C语言程序的一个例子。

```
/* C Example Program
 Input:  An integer, listlen, where listlen is less than
         100, followed by listlen-integer values
 Output: The number of input values that are greater than
         the average of all input values */
int main (){
  int intlist[99], listlen, counter, sum, average, result;
  sum = 0;
  result = 0;
  scanf("%d", &listlen);
  if ((listlen > 0) && (listlen < 100)) {
/* Read input into an array and compute the sum */
    for (counter = 0; counter < listlen; counter++) {
      scanf("%d", &intlist[counter]);
      sum += intlist[counter];
     }
/* Compute the average */
    average = sum / listlen;
/* Count the input values that are > average */
    for (counter = 0; counter < listlen; counter++)
      if (intlist[counter] > average) result++;
/* Print result */
    printf("Number of values > average is:%d\n", result);
   }
  else
    printf("Error-input list length is not legal\n");
 }
```

2.13 基于逻辑的程序设计：Prolog语言

简单地说，逻辑程序设计就是用形式化的逻辑表示方法将计算过程传达给计算机。谓词演算是当前逻辑程序设计语言采用的表示方法。

用逻辑程序设计语言来编程是非过程的。这种语言的程序不明确说明如何计算得到结果，而是说明结果所需的形式和/或特征。在逻辑程序设计语言中提供这种功能，需要有一种简明的方法，为计算机提供相关信息和计算所需结果的推理过程。谓词演算提供了与计算机通信的基本形式，最早由 Robinson(1965)提出的称为归结的证明方法提供了推理技术。

2.13.1　设计过程

在 20 世纪 70 代的早期，艾克斯-马赛大学人工智能小组的 Alain Colmerauer 和 Phillippe Roussel，以及爱丁堡大学人工智能系的 Robert Kowalski，详述了 Prolog 语言的基本设计。Prolog 语言的主要组成部分是一种表示谓词演算命题的方法和一种限定形式的归结的实现。谓词演算和归结将在第 16 章介绍。第一个 Prolog 语言解释器于 1972 年在马赛开发出来，Roussel(1975)说明了所实现的语言版本。Prolog 这一名称来自于“程序设计逻辑(programming logic)”。

2.13.2　语言概述

Prolog 语言程序由一系列语句组成。Prolog 语言只有几类语句，但是语句可以很复杂。

Prolog 语言通常的用途之一是作为一种智能的数据库。这一应用为介绍 Prolog 语言提供了一个简单的框架。

Prolog 语言程序的数据库由两类语句组成：事实语句和规则语句。事实语句的例子如下：

```
mother(joanne, jake).
father(vern, joanne).
```

这两条语句说明 joanne 是 jake 的母亲(mother)，vern 是 joanne 的父亲(father)。

规则语句的例子如下：

```
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

这一语句说明，对变量 X、Y 和 Z 的一些特定值，如果 X 是 Y 的父母(parent)且 Y 是 Z 的父母(parent)，就可以推出 X 是 Z 的祖父母(grandparent)。

Prolog 数据库可用目标语句进行交互查询，目标语句如：

```
father(bob, darcie).
```

这一语句问 bob 是否是 darcie 的父亲(father)。给 Prolog 系统提供了这样的查询或目标后，它就使用归结过程尝试确定语句的真实性。如果能够推理出目标为真，就显示 true，如果不能证明，就显示 false。

2.13.3　评价

在 20 世纪 80 年代，有很少的一些计算机科学家认为，逻辑程序设计最有希望摆脱命令式语言的复杂性，摆脱开发大量所需的可靠软件的无数问题。然而，逻辑程序设计至今为止没有得到更广泛的应用有两个主要原因。第一，与其他一些非命令式方法一样，逻辑语言的

程序与相应的命令式语言程序相比，到目前为止是极其低效的。第二，逻辑程序设计只在一些相对较小的应用领域中才是有效的方法，包括：特定类别的数据库管理系统和一些人工智能领域。

有一种 Prolog 语言的方言支持面向对象的程序设计——Prolog++ (Moss，1994)。逻辑程序设计和 Prolog 语言在第 16 章详细介绍。

2.14 历史上规模最大的设计工作：Ada 语言

Ada 语言是至今涉及范围最广、耗资最多的语言设计工作的成果。下面简要介绍 Ada 语言的发展过程。

2.14.1 历史背景

Ada 语言是为美国国防部(Department of Defense，DoD)开发的，因此国防部的计算环境就决定了该语言的形式。在 1974 年之前，国防部一半以上的计算机应用程序都是嵌入式系统。在嵌入式系统中，计算机硬件就嵌入在系统所控制或提供服务的设备中。软件的成本上升很快，主要因为系统的复杂程度在上升。那时共有 450 多种不同的程序设计语言用于国防部的项目，但是没有一种语言是由国防部标准化过的，每一个防卫项目承包商都可以为每一个合同定义一种新的不同的语言[12]。由于语言数目大量增加，应用软件很少能复用，而且也没有创建软件开发工具(因为软件开发工具通常是与语言相关的)。有极其大量的语言在使用，但没有一种真正适用于嵌入式系统应用，因此，1974 年陆军、海军和空军各自独立地提出要为嵌入式系统开发一种高级语言。

2.14.2 设计过程

注意到这一广泛需求，1975 年 1 月国防研究与工程办公室(Defense Research and Engineering)主任 Malcolm Currie 成立了高级语言工作组(High-Order Language Working Group，HOLWG)，由空军的 Lt. Col. William Whitaker 担任第一任领导。HOLWG 有来自所有军种的代表，以及英国、法国、西德的联络人员，其最初的章程是：

- 确定新的国防部高级语言的需求。
- 对现有语言进行评价，确定是否有可行的候选语言。
- 建设采用或实现程序设计语言的一个最小集合。

1975 年 4 月，HOLWG 给出了新语言的“稻草人版(Strawman)”需求文档(Department of Defense，1975a)，分发到各个军事部门、联邦政府单位、挑选出的工业和大学代表，以及对此感兴趣的欧洲团体。

“稻草人版”文档在 1975 年 8 月改为“木头人版(Woodenman)”(Department of Defense，

12. 这一结果主要归因于汇编语言在嵌入式系统上的广泛应用，同时大多数嵌入式系统采用专用的处理器。

1975b)，1976 年 1 月改为“铁皮人版(Tinman)”(Department of Defense，1976)，1977 年 1 月改为“铁人版(Ironman)”(Department of Defense，1977)，1978 年 6 月最终改为“钢铁人版(Steelman)”(Department of Defense，1978)。

经过冗长的过程，提交的许多语言提案最终缩减到四个，这四个提案都是基于 Pascal 语言的。1979 年 5 月，从这四个提案中选出了 Cii Honeywell/Bull 语言设计提案，作为要使用的方案。法国的这个 Cii Honeywell/Bull 设计小组是最后四个提案中唯一的国外竞争者，由 Jean Ichbiah 领导。

1979 年春，海军装备司令部的 Jack Cooper 建议新语言的名称为 Ada，被采纳了。这一名字是为了纪念 Lovelace 伯爵夫人、数学家、诗人拜伦(Lord Byron)的女儿 Augusta Ada Byron(1815－1851)。人们普遍认为她是世界上第一个程序员。她与 Charles Babbage 一起在机械计算机上从事差分机和分析机的工作，编写了一些数值处理程序。

Ada 语言的设计和基本原理由 ACM 在 *SIGPLAN Notices* 杂志上发表(ACM，1979)，发行量超过一万份。1979 年 10 月，在波士顿举行了公开的测试和评估会议，会议代表来自美国和欧洲的 100 多个组织。截至 11 月份，收到了来自 15 个不同国家的 500 多份语言报告，大多数报告都建议一些小的修改，而不是极大的改变或完全的否定。基于这些语言报告，需求说明的下一版本“石头人版(Stoneman)”文档(Department of Defense，1980a)在 1980 年 2 月发布。

1980 年 7 月完成了语言设计的修订版，并被接受和命名为 MIL-STD 1815，即标准的 *Ada Language Reference Manual*(《Ada 语言参考手册》)。选择数字 1815 是因为这是 Augusta Ada Byron 的出生年份。1982 年 7 月发布了 *Ada Language Reference Manual* 的另一个修订版本。1983 年，美国国家标准学会(ANSI)对 Ada 语言进行了标准化，Goos 和 Hartmanis(1983)对这一“最终”的正式版本进行了说明。此后至少有 5 年，Ada 语言的设计没有变化。

2.14.3　语言概述

这一小节简要介绍 Ada 语言的 4 个主要贡献。

Ada 语言的包提供了封装数据对象、数据类型说明和过程的方法，从而在程序设计上提供数据抽象的使用支持，这些将在第 11 章中介绍。

Ada 语言包含大量异常处理机制，这样，检测出各种各样的异常或运行时错误后，程序员都能够获取控制。第 14 章介绍异常处理。

Ada 语言中的程序单元具有通用性。例如，可以写一个排序过程，为待排序数据使用非特定的类型。这样的通用过程在使用前必须先实例化为特定的类型，这是通过一条语句来实现的，让编译器生成使用该指定类型的过程版本。使用这样的通用单元，提高了复用程序单元的范围，而不是由程序员来复制。第 9 章和第 11 章介绍通用性。

Ada 语言还使用会合机制，提供了特殊程序单元(称为“任务”)的并发执行功能。会合是任务间通讯和同步方法的名称。并发在第 13 章中介绍。

2.14.4 评价

在 Ada 语言的设计中，应考虑的最重要的一些方面可能是以下几项：

- 由于采用竞争式的设计，对参与者并没有限制。
- Ada 语言采用了 20 世纪 70 年代晚期大部分的软件工程和语言设计概念。虽然可以质疑实际上用了什么方法将这些特性混合在一起，质疑将如此大量的特性包含在一种语言中是否明智，但大多数人认为这些特性是有价值的。
- 虽然大多数人没有预计到这一点，但是 Ada 语言编译器的开发是一项困难的任务。直到 1985 年，在语言设计完成差不多 4 年之后，才出现了真正可用的 Ada 语言编译器。

在 Ada 语言的最初几年，最严厉的批评是它太大、太复杂了。特别是 Hoare(1981)声称，不应该在重视可靠性的应用中使用它，而它恰恰是为这类应用而设计的。另一方面，其他人称赞它是当时语言设计的典范。实际上，即使是 Hoare，最终对这一语言的意见也有所缓和。

以下是 Ada 语言程序的一个例子。

```
-- Ada Example Program
-- Input:  An integer, List_Len, where List_Len is less
--         than 100, followed by List_Len-integer values
-- Output: The number of input values that are greater
--         than the average of all input values
with Ada.Text_IO, Ada.Integer.Text_IO;
use Ada.Text_IO, Ada.Integer.Text_IO;
procedure Ada_Ex is
  type Int_List_Type is array (1..99) of Integer;
  Int_List : Int_List_Type;
  List_Len, Sum, Average, Result : Integer;
begin
  Result:= 0;
  Sum := 0;
  Get (List_Len);
  if (List_Len > 0) and (List_Len < 100) then
-- Read input data into an array and compute the sum
    for Counter := 1 .. List_Len loop
      Get (Int_List(Counter));
      Sum := Sum + Int_List(Counter);
    end loop;
-- Compute the average
    Average := Sum / List_Len;
-- Count the number of values that are > average
    for Counter := 1 .. List_Len loop
      ifInt_List(Counter) > Average then
        Result:= Result+ 1;
      end if;
    end loop;
-- Print result
```

```
      Put ("The number of values > average is:");
      Put (Result);
      New_Line;
    else
      Put_Line ("Error—input list length is not legal");
    end if;
end Ada_Ex;
```

2.14.5　Ada 95 和 Ada 2005

下面简要介绍 Ada 95 最重要的两个新特性。在本书后面需要区别两个不同版本时，用 Ada 83 表示语言的最初版本，用 Ada 95(它真正的名字)表示后来的版本。在介绍两个版本都有的特性时，就使用 Ada 这个名字。Ada 95 标准语言在 ARM(1995)中定义。

Ada 83 的类型派生机制在 Ada 95 中扩展为允许在基类的继承类中加入新的成员，这提供了继承这一面向对象程序设计语言的关键元素。将子程序调用动态绑定到子程序定义上，是通过子程序的分派实现的，分派操作基于这一类类型中派生类型的标志值。这一特性提供了多态这种面向对象程序设计的又一重要特性。Ada 95 的这些特性将在第 12 章中介绍。

Ada 83 的会合机制只是提供了在并发过程间共享数据的一种笨重而低效的方法，需要引入一个新的任务来控制对共享数据的访问。Ada 95 的受保护对象提供了一个很有吸引力的替代方法：将共享数据封装在一个语法结构中，该结构控制所有的数据访问，不管是会合还是子程序调用。Ada 95 关于并发和共享数据的新特性将在第 13 章中介绍。

人们普遍相信，Ada 95 没有广泛应用是因为国防部不再要求在军队的软件系统中使用它。当然还有其他因素阻碍了它的更广泛应用，其中很重要的就是 C++语言被广泛接受用于面向对象程序设计，在 Ada 95 发布之前就已经这样了。

在 Ada 95 中添加一些新特性，就得到了 Ada 2005，其中包括接口(类似于 Java 的接口)、对调度算法的更多控制和同步的接口。

Ada 广泛应用于商务和国防航空、空中交通控制、铁路运输等领域。

2.15　面向对象的程序设计：Smalltalk

Smalltalk 是第一种完全支持面向对象程序设计的语言，因此在介绍程序设计语言的发展过程时是一个重要的部分。

2.15.1　设计过程

开发 Smalltalk 的想法，来源于 20 世纪 60 年代晚期 Alan Kay 在犹他大学的博士论文(Kay，1969)。Kay 以非凡的远见，预言将来会出现功能强大的台式计算机。回想一下，最早的微机系统直到 20 世纪 70 年代中期才上市，它们与 Kay 预见到的计算机有很大的差距，Kay 预见的计算机每秒执行百万条或更多条指令，并且包含几兆字节内存。这样的计算机

作为工作站，在 20 世纪 80 年代早期才广泛普及。

Kay 相信台式计算机能够由非程序员使用，因而需要非常强大的人机界面功能。20 世纪 60 年代晚期的计算机主要是面向批处理的，只有专业的程序员和科学家才用。Kay 认为，为了能让非程序员使用，计算机必须高度可交互，使用复杂的图形用户界面。一些图形概念来自于 Seymour Papert 系统中 LOGO 语言的经验，这一语言使用图形来帮助孩子们使用计算机(Papert，1980)。

Kay 将他最初预见的系统称为 Dynabook，希望成为通用的信息处理器。这个系统部分基于他曾辅助设计过的 Flex 语言。Flex 语言主要基于 SIMULA 67。Dynabook 采用典型的桌面模式，上面有一些纸，其中一些被部分地盖住了。最上面的纸通常是关注的焦点，其他纸暂时不被关注。Dynabook 的显示就是以这样的场景为模型，采用屏幕窗口来表示桌面上的各种纸张，用户通过键盘或用手指碰触屏幕来与显示场景交互。在凭借 Dynabook 的初步设计获得博士学位之后，Kay 的目标变为亲眼见到这种计算机的实现。

Kay 找到 Xerox Palo Alto 研究中心(Xerox PARC)，介绍了他关于 Dynabook 的想法，因此受雇于 Xerox，随后成立了学习研究小组(Learning Research Group)。这个小组的第一个任务是设计一种语言，来支持 Kay 的程序设计构想，并且在当时最好的个人计算机上实现。小组的工作产生了一个 Dynabook 的过渡产品，由 Xerox Alto 工作站和 Smalltalk-72 软件组成，这两者结合形成一个探索进一步开发的工具。在这个系统上进行的一些研究项目包括一些教孩子进行程序设计的实验。在进行这些实验的同时，也进行进一步的开发，生成了一系列以 Smalltalk-80 为最终产品的语言。随着语言的进步，语言所使用的硬件也在进步。到 1980 年，这种语言和 Xerox 硬件差不多达到了 Alan Kay 早期想法的要求。

2.15.2 语言概述

在 Smalltalk 的世界中没有其他东西，只有对象，从整数常量到大型的复杂软件系统都是对象。Smalltalk 中所有的计算都以一致的技术完成：向一个对象发送一条消息，调用该对象的一个方法。消息的响应也是一个对象，这个对象返回所请求的信息，或者只是简单地告诉消息发送者，所请求的处理已经完成。消息与子程序调用的基本区别是：消息是发送给数据对象的，特别是发送给为此对象定义的一种方法，然后执行所调用的方法，通常会修改消息发送目标对象的数据；子程序调用是发送给子程序代码的消息，子程序要处理的数据通常作为参数传送给它[13]。

在 Smalltalk 中，对象的抽象是类，与 SIMULA 67 中的类非常相似。可以创建类的实例，创建的实例就是程序中的对象。

Smalltalk 语言的语法与其他任何程序设计语言都不同，主要因为使用了消息，而不是算术和逻辑表达式以及传统的控制语句。在下一小节的示例程序中，将举例说明 Smalltalk 语言的一种控制结构。

---

13. 当然，方法的调用也可以把要处理的数据传递给调用的方法。

2.15.3　评价

Smalltalk 语言大大提升了两个不同方面的计算：图形用户界面和面向对象程序设计。现在在软件系统中处于主导地位的用户界面方法是视窗系统，就是从 Smalltalk 发展而来的。当今最重要的软件设计方法学和程序设计语言是面向对象的，虽然面向对象语言的一些思想起源于 SIMULA 67，但它们在 Smalltalk 中才发展成熟。毫无疑问，Smalltalk 语言对计算机世界的影响是广泛而长久的。

以下是 Smalltalk 类定义的一个例子。

```
"Smalltalk Example Program"
"The following is a class definition, instantiations
of which can draw equilateral polygons of any number of
sides"
class name                  Polygon
superclass                  Object
instance variable names     ourPen
numSides
sideLength
"Class methods"
  "Create an instance"
  new
     ^ super new getPen

"Get a pen for drawing polygons"
getPen
   ourPen <- Pen new defaultNib: 2

"Instance methods"
"Draw a polygon"
draw
   numSides timesRepeat: [ourPen go: sideLength;
                          turn: 360 // numSides]

"Set length of sides"
length: len
   sideLength <- len

"Set number of sides"
sides: num
   numSides <- num
```

2.16　结合命令式和面向对象的特性：C++

C 语言的起源已在 2.12 节中介绍，Simula 67 的起源已在 2.10 节中介绍，Smalltalk 的起源已在 2.15 节中介绍。C++模仿 Simula 67，在 C 语言的基础上建立了语言机制，支持 Smalltalk 许多开创性的特性。C++是在 C 语言的基础上通过一系列修改发展而来的，改进了命令式的语言特性，增加了支持面向对象程序设计的语言结构。

2.16.1 设计过程

从 C 语言到 C++的第一步是 1980 年由 Bjarne Stroustrup 在贝尔实验室完成的。对 C 语言的修改包括增加了函数参数的类型检查和类型转换，更重要的是增加了与 SIMULA 67 和 Smalltalk 中的类相关的类结构，增加的还有派生类、对继承部分的公有/私有访问控制、构造函数和析构函数，以及友元类。在 1981 年又增加了内联函数、默认参数，以及赋值运算符重载。通过修改得到的语言称为“带类的 C”，在 Stroustrup(1983)中做了说明。

探讨“带类的 C”的目标是有益的。它的首要目标是提供一种语言，可以像 SIMULA 67 那样用类和实例来组织程序，另一个重要目标是相对于 C 语言来说，应该几乎或完全没有性能损失。例如，根本没有考虑对数据下标范围进行检查，因为相对于 C 语言来说，这会导致严重的性能下降。“带类的 C”还有一个目标是可用于 C 语言能适用的任何应用领域，因此 C 语言中的几乎所有特性都不能去掉，即使是那些不安全的特性。

到 1984 年，这种语言已经扩展为包含虚方法、方法名和运算符重载，以及引用类型，虚方法提供了方法调用到特定方法定义的动态绑定。这一版本的语言称为 C++，在(Stroustrup，1984)中做了说明。

1985 年出现了第一个可用的实现：Cfront 系统，可以将 C++程序翻译为 C 程序。这个版本的 Cfront 和所实现的 C++版本称为 1.0 正式版，在 Stroustrup(1986)中做了说明。

1985 年到 1989 年间，C++继续发展，主要是基于用户对发布的第一版实现的反馈。这个新版本命名为 2.0 版，它的 Cfront 实现在 1989 年 6 月发布。在 C++ 2.0 版中增加的最重要特性是对多继承(类具有一个以上的父类)和抽象类的支持，还有一些其他性能增强。抽象类将在第 12 章中介绍。

C++的 3.0 版是在 1989 年至 1990 年间发展成型的，加入了模板和异常处理，模板提供了参数化的类型。当前 C++的版本是在 1998 年标准化的，在 ISO(1998)中进行了说明。

2002 年，Microsoft 公司发布了.NET 计算平台，包含一个新的 C++版本，称为托管 C++(Managed C++，MC++)。MC++扩展了 C++，提供对.NET 框架功能的访问，增加的部分包括属性、委托、接口和垃圾收集对象的引用类型。属性将在第 11 章中介绍。委托将在 2.19 节介绍 C#时简要介绍。由于.NET 不支持多继承，因此 MC++也不支持。

2.16.2 语言概述

由于 C++同时具有函数和方法，因此它同时支持过程式程序设计和面向对象的程序设计。

C++中的运算符可以重载，就是说用户可以为已有的运算符创建作用于用户自定义类型的运算符。C++的方法也可以重载，就是说用户可以用相同的名称定义一个以上的方法，只要它们的参数数量或类型不相同。

C++中的动态绑定是由虚方法提供的。这些虚方法在一系列通过继承相互关联的类中，使用重载方法定义基于类型的操作。一个指向 A 类型对象的指针也可以指向以类 A 作为祖先类的类对象。当这个指针指向一个重载的虚方法时，动态选择当前类型的方法。

方法和类都可以是模板，也就是说它们都可以参数化。例如，一个方法可以写成模板

方法，允许它有各种参数类型的版本。类也可以有这样的灵活性。

C++支持多继承。C++包含的异常处理与 Ada 中的异常处理差别迥异。差别之一是 C++不能处理硬件可检测的异常。Ada 和 C++中的异常处理结构将在第 14 章中介绍。

2.16.3　评价

C++迅速成为一种非常受欢迎的语言，并一直保持如此。它受欢迎的因素之一是具有优秀且廉价的编译器，另一个因素是几乎完全与 C 语言向后兼容(就是说 C 语言程序可以只做很少修改就作为 C++程序来编译)，而且在大部分实现中可以将 C++代码与 C 语言代码链接在一起，因此对于许多 C 语言程序员来说，学习 C++相对简单。最后，在 C++刚出现时，面向对象的程序设计开始得到广泛的关注，C++是当时唯一适用于大型商业软件项目的语言。

至于 C++的缺点，由于它是非常庞大和复杂的语言，显然与 PL/I 的缺点相似。它继承了 C 语言大部分的非安全因素，因此与 Ada 和 Java 这些语言相比，它不够安全。

2.16.4　一种相关语言：Objective-C

Objective-C(Kochan，2009)是另一种具备命令式和面向对象特性的混合语言。Objective-C 是由 Brad Cox 和 Tom Love 在 20 世纪 80 年代早期设计的。最初，它包含 C 以及 Smalltalk 的类和消息传递功能。在已创建的程序设计语言中，都是给命令式语言添加对面向对象程序设计的支持，而 Objective-C 是唯一一个使用 Smalltalk 语法提供该支持的语言。

Steve Jobs 离开苹果公司，创立了 NeXT 后，就获得了 Objective-C 的许可，并将它用于编写 NeXT 计算机系统软件。NeXT 还发布了它的 Objective-C 编译器、NeXTstep 开发环境和一个工具库。NeXT 项目失败后，苹果公司购买了 NeXT，并使用 Objective-C 编写 MAC OS X。Objective-C 是所有 iPhone 软件的编写语言，所以在 iPhone 面世后，就迅速流行起来。

Objective-C 从 Smalltalk 继承的一个特点是将消息动态绑定到对象上。这意味着没有对消息进行静态检查。如果消息发送给对象，而对象不能响应该消息，则只有在运行期间才知道这个情况，此时会抛出一个异常。

2006 年，苹果公司发布了 Objective-C 2.0，它添加了一种垃圾收集形式和声明属性的新语法。可惜，iPhone 运行库系统不支持垃圾收集。

Objective-C 是 C 的一个严格超集，所以 C 语言的所有非安全特性都出现在 Objective-C 中。

2.16.5　另一种相关语言：Delphi

Delphi(Lischner，2000)是一种混合语言，与 C++和 Objective-C 相似，它是在已有的命令式语言 Pascal 中增加了面向对象支持及其他一些特性后得到的。C++和 Delphi 的许多区别来自于其祖先语言，以及产生它们的程序设计文化背景。由于 C 语言是一种强大但具有潜在不安全因素的语言，所以 C++也是这样，至少在数组下标范围检查、指针运算和大量

强制类型转换等方面是这样。同样，由于 Pascal 比 C 语言更优雅和安全，Delphi 也比 C++更优雅和安全，Delphi 也不如 C++复杂。例如，Delphi 不允许用户自定义运算符重载、通用子程序和参数化的类，而这些都是 C++的一部分。

Delphi 与 Visual C++一样，为开发人员提供了图形用户界面(GUI)和为用 Delphi 编写的应用程序创建 GUI 界面的简单方法。Delphi 是由 Anders Hejlsberg 设计的，他之前开发过 Turbo Pascal 系统。这两种语言都是由 Borland 公司销售和发行的。Hejlsberg 也是 C#的主要设计人员。

2.16.6 一种关系不大的语言：Go

GO 程序设计语言与 C++不直接相关，但它是基于 C 的。本节介绍它的部分原因是，它不需要单独用一节的篇幅介绍，也不适合放在其他地方讨论。

Go 语言是由 Google 的 Rob Pike、Ken Thompson 和 Robert Griesemer 设计的。Thompson 是 C 语言的祖先 B 语言的设计者，还与 Dennis Ritchie 一起设计了 UNIX。他和 Pike 以前都受雇于贝尔实验室。Go 语言的最初设计开始于 2007 年，第一个实现在 2009 年后期发布。开发 Go 最初的动机之一是大型 C++程序在 Google 上编译得很慢。Go 最初的编译器的一个特点是它非常快。Go 语言借用了 C 中的一些语法和结构，它的新特性包括(1)数据声明在语法上与其他基于 C 的语言相反；(2) 变量在类型名称的前面；(3) 如果给变量提供了初始值，变量声明就可以按引用指定类型；(4)函数可以返回多个值。Go 语言不支持传统的面向对象程序设计，因为它没有继承形式。但是，可以给任意类型定义方法，它还没有泛型。Go 语言的控制语句类似于其他基于 C 的语言，但 switch 语句不包含执行下一个语句段的隐式执行机制。Go 语言包含 goto 语句、指针、关联数组和接口(但它们不同于 Java 和 C#中的接口)，并使用其 go 例程支持并发。

2.17 基于命令式的面向对象语言：Java

Java 的设计人员从 C++开始，去掉了一些语言结构，改变了一些结构，又添加了一些其他结构，所得到的语言具有 C++的很多强大功能和灵活性，但更小巧、简单且安全。

2.17.1 设计过程

Java 和许多程序设计语言一样，是因为某个应用领域没有令人满意的已有语言而设计的。1990 年 Sun 公司(Sun Microsystems)认为需要一种程序设计语言用于嵌入式消费类电子产品，例如电烤箱、微波炉和交互式电视系统。对于这样一种语言，可靠性是一个主要目标。在微波炉的软件中，可靠性看起来可能不是重要的因素：如果微波炉出现软件故障，不太可能对人产生严重的危害，不太可能引起重大的法律纠纷。但是如果某一型号微波炉在已经生产和销售百万台后，发现软件有错误，那么召回产品将带来巨大损失。因此，可靠性确实是消费类电子产品中软件的重要特征。

在考虑过 C 语言和 C++之后，Sun 公司认为这两种语言都不能满足为消费类电子产品开发软件的需要。虽然 C 语言相对小巧，但它不提供面向对象程序设计的支持，而他们认为这是必须要有的。C++支持面向对象的程序设计，但是它太庞大、太复杂了，部分原因是它同时也支持面向过程的程序设计。Sun 公司还认为 C 语言和 C++都无法提供所需的高度可靠性，因此设计了一种新语言，后来命名为 Java。Java 设计的基本目标为：要具有比 C++更好的简单性和可靠性。

虽然 Java 最初的动力是消费类电子产品，但在 Java 刚出现的几年中，使用 Java 的产品都没有上市销售。这主要由于出现了新的图形化浏览器，万维网(World Wide Web)自 1993 年开始得到广泛应用，Java 成为网页程序设计的有效工具。特别是相对小巧的 Java 程序——Java applet，在网络浏览器中解释，其输出结果可包含在要显示的网页文档中，在 20 世纪 90 年代中后期迅速得到广泛应用。在 Java 流行的最初几年中，网络是 Java 最常见的应用领域。

Java 设计小组由 James Gosling 领导，他以前设计过 UNIX 系统上的 emacs 编辑器和 NeWS 视窗系统。

2.17.2　语言概述

如前所述，Java 基于 C++，但设计得更小巧、更简单、更可靠。与 C++一样，Java 既有类也有基本类型。Java 数组是预定义类的实例，而数组在 C++中不是这样，许多 C++使用者为数组建立了封装的类，来添加诸如下标范围检查等特性，而这些在 Java 中是隐含的。

Java 没有指针，但是其引用类型提供指针的一些功能，这些引用用于指向类的实例。所有的对象都在堆上分配空间。引用在必要时总是隐式地解引用，因此行为更像普通的标量变量。

Java 有基本的布尔类型，称为 boolean，主要用于控制语句(例如 if 和 while)的控制表达式。与 C 和 C++语言不同，Java 中的算术表达式不能用作控制表达式。

Java 和包括 C++在内的许多支持面向对象程序设计的前辈语言有一个重要的不同：在 Java 中不能编写独立的子程序。所有的 Java 子程序都是方法，定义在类中，而且方法只能通过类或对象来调用。结果就是，C++同时支持过程式和面向对象的程序设计，Java 只支持面向对象的程序设计。

C++和 Java 的另一个重要区别是 C++在类定义中直接支持多继承。Java 只支持类的单继承，但多继承的一些好处可以通过使用接口结构来获得。

struct 和 union 在 C++的语言结构中有，但没有复制到 Java 中。

Java 通过 synchronized 修饰符提供了一种并发控制的相对简单的形式，这个修饰符可作用于方法和语句块。在这两种情形下，都会附加一个锁，锁保证访问和执行的互斥。在 Java 中创建并发进程相对容易，并发进程在 Java 中称为线程(thread)。

Java 隐式地解除对对象的存储分配，通常称为垃圾收集。这样，不再需要对象时，程序员不必显式删除对象。用没有垃圾收集功能的语言所写的程序，经常会有所谓的内存泄

漏问题，即分配了存储空间但没有解除分配，显然这会最终导致耗尽所有可用的内存。解除对象的存储分配将在第 6 章详细介绍。

与 C 语言和 C++不同，Java 只有在赋值类型扩展(从“小”类型赋值为“大”类型)时，才有强制类型转换(隐式的类型转换)。因此 int 到 float 的强制类型转换可通过赋值运算符实现，但 float 到 int 的强制类型转换则不行。

2.17.3 评价

Java 的设计人员很好地去除了 C++中多余的和不安全的特性。例如，去除 C++一半的赋值强制转换，显然是迈向更高可靠性的一步。数组访问的下标范围检查也使语言更加安全。并发的加入扩展了语言所能适用的应用领域，加入图形用户界面、数据库访问、网络访问的类库也起到了这样的作用。

Java 的可移植性，至少以中间代码形式的移植，常常被人们归功于语言的设计，但实际上并不是。任何语言都可以翻译为一种中间代码，并且运行在具有该中间代码虚拟机的任何平台上。这种可移植性的代价就是解释执行的代价，通常比机器代码的执行慢一个数量级。Java 解释器的初始版本称为 Java 虚拟机(Java Virtual Machine，JVM)，实际上至少要比相应的编译过的 C 程序慢 10 倍。但是，人们现在使用即时(Just-in-Time，JIT)编译器，在执行前将许多 Java 程序翻译为机器代码，使 Java 程序的效率与 C++这类传统的编译语言的程序效率相当。

Java 的使用比任何其他语言的使用增长得都快，最初是由于它在动态网页文档程序设计方面的价值。Java 快速崛起的原因之一显然是编程人员喜欢它的设计，一些开发人员认为 C++太庞大、太复杂，不实用、不安全，Java 给他们提供了另一种选择，具有 C++的很多功能，但更简单、更安全。另一个原因是 Java 的编译器/解释器系统是免费的，很容易在网络上获取。Java 现在广泛用于各种不同的应用领域。

Java 最新的版本 Java 7 出现在 2011 年，它一开始就添加了许多内容，包括枚举类、泛型，以及一种新的迭代结构。

以下是 Java 程序的一个例子。

```
// Java Example Program
//  Input: An integer, listlen, where listlen is less
//         than 100, followed by length-integer values
// Output: The number of input data that are greater than
//         the average of all input values
import java.io.*;
class IntSort {
public static void main(String args[]) throws IOException {
  DataInputStream in = new DataInputStream(System.in);
  Int listlen,
      counter,
      sum = 0,
      average,
      result = 0;
```

```
  int[] intlist = new int[99];
  listlen = Integer.parseInt(in.readLine());
  if ((listlen > 0) && (listlen < 100)) {
/* Read input into an array and compute the sum */
    for (counter = 0; counter < listlen; counter++) {
      intlist[counter] =
             Integer.valueOf(in.readLine()).intValue();
      sum += intlist[counter];
    }
/* Compute the average */
    average = sum / listlen;
/* Count the input values that are > average */
    for (counter = 0; counter < listlen; counter++)
      if (intlist[counter] > average) result++;
/* Print result */
      System.out.println(
          "\nNumber of values > average is:" + result);
    }//**  end of then clause of if ((listlen > 0) ...
    else System.out.println(
            "Error-input list length is not legal\n");
  }//** end of method main
} //** end of class IntSort
```

2.18　脚本语言

脚本语言是在最近 25 年中发展起来的。早期的脚本语言用于在文件中输入一系列待解释执行的命令，称为脚本。这类语言的第一种称为 sh(shell 的简称)，最初是少量命令的集合，这些命令经解释后调用系统子程序，执行诸如文件管理、简单的文件过滤这类工具函数。在这一基础上增加变量、控制流语句、函数，以及各种其他功能，就得到了一种完整的程序设计语言。这类语言中功能最强大最广为人知的一种是 ksh(Bolsky 和 Korn，1995)，是由贝尔实验室的 David Korn 开发的。

还有一种脚本语言名为 awk，是贝尔实验室的 Al Aho、Brian Kernighan 和 Peter Weinberger 开发的(Aho 等，1988)。awk 最初是一种报表生成语言，但是后来成为一种更通用的语言。

2.18.1　Perl 的起源与特点

Perl 语言是由 Larry Wall 开发的，最初是 sh 和 awk 的组合。Perl 自诞生以来有着巨大的发展，现在已经是一种功能强大的程序设计语言，但仍有些原始。虽然 Perl 仍然称为脚本语言，但它实际上更像一种典型的命令式语言，因为它总是编译执行的，至少是在执行前先编译为一种中间语言。而且，它具有所有需要的语言结构，可适用于各种有计算问题的领域。

Perl 具有一些有趣的特性，本章和本书的其余部分只介绍其中的几项。

Perl 中的变量是静态类型的，是隐式声明的。变量有三种不同的名称空间，由变量名称的第一个字符来指定。所有标量变量名都以美元符号($)开头，所有数组名都以@符号开头，所有哈希名(哈希在下文简要介绍)都以百分号(%)开头。这一规定使程序中的变量名比其他任何程序设计语言的变量名更易读。

Perl 包含大量隐式变量，其中一些用于存储 Perl 参数，例如换行符或语言实现所用符号的特殊形式。隐式变量通常用作内置函数的默认参数，以及一些运算符的默认操作数。隐式变量具有各不相同的名称，虽然意义不明显，例如$!和@_。隐式变量的名称与用户自定义变量名一样，采用三类名称空间，因此$!是一个标量。

Perl 数组的两个特点使之与常见命令式语言的数组不同。首先，Perl 数组的长度可变，就是说它们可以在执行时根据需要变长或缩短，其次它们是稀疏的，就是说数组元素间可以有间隙。这些间隙不占内存空间，用于数组的迭代语句 foreach 在迭代时会跳过没有的元素。

Perl 具有关联数组，称为哈希。这一数据结构以字符串作为索引，是隐式管理的哈希表。Perl 系统提供哈希函数，在必要时可增大这一数据结构的大小。

Perl 是一种功能强大但有些危险的语言。它的标量类型可存储字符串和数值，数值通常是以双精度浮点形式存储的。根据上下文，数组可以强制转换为字符串，反之亦然。如果字符串用于数值上下文，但是这个字符串无法转换为数值，就会使用 0，而且没有警告或错误信息发给用户。这样的结果可能会导致编译器和运行时系统无法检测的错误。Perl 语言不检查数组下标，因为数组没有固定的下标范围。引用不存在的元素将返回 undef，在数值上下文中会解释为 0，因此在访问数组元素时也没有错误检测机制。

Perl 最初的用途是作为 UNIX 系统的工具来处理文本文件，它曾经而且至今仍作为 UNIX 系统的管理工具而被广泛使用。在万维网出现后，Perl 作为一种通用网关接口(Common Gateway Interface CGI)语言与万维网一起广泛使用，但现在它很少用于这一用途了。Perl 作为一种通用语言可用于各种应用领域，例如计算生物学和人工智能。

下面是 Perl 程序的一个例子。

```
# Perl Example Program
# Input:  An integer, $listlen, where $listlen is less
#         than 100, followed by $listlen-integer values.
# Output: The number of input values that are greater than
#         the average of all input values.
($sum, $result) = (0, 0);
$listlen = <STDIN>;
if (($listlen > 0) && ($listlen < 100)) {
# Read input into an array and compute the sum
  for ($counter = 0; $counter < $listlen; $counter++) {
    $intlist[$counter] = <STDIN>;
  } #- end of for (counter ...
# Compute the average
  $average = $sum / $listlen;
# Count the input values that are > average
  foreach $num (@intlist) {
```

```
    if ($num > $average) { $result++; }
  } #- end of foreach $num ...
# Print result
  print "Number of values > average is: $result \n";
} #- end of if (($listlen ...
else {
  print "Error--input list length is not legal \n";
}
```

2.18.2　JavaScript 的起源与特点

20 世纪 90 年代中期，在第一批图形浏览器出现后，万维网的使用呈爆炸式增长。HTML 文档本身是完全静态的，与 HTML 文档相关的计算需求很快变得非常重要。通用网关接口(CGI)使服务器端的计算成为可能，它允许 HTML 文档请求在服务器上执行程序，并将这种计算的结果以 HTML 文档的形式返回给浏览器。在浏览器端的计算随着 Java applet 的出现也成为可能。这两种解决方案现在大都被新的技术替代，主要就是脚本语言。

JavaScript(Flanagan，2002)最初由 Netscape 公司的 Brendan Eich 开发，其最初的名称是 Mocha，后来命名为 LiveScript。1995 年后期，Netscape 和 Sun 公司联合投资 LiveScript，并将其更名为 JavaScript。JavaScript 从 1.0 版变为 1.5 版有很大的发展，增加了许多新的特性和功能。20 世纪 90 年代末，欧洲计算机制造商协会(European Computer Manufacturers Association，ECMA)在 ECMA-262 文件中提出了 JavaScript 语言标准。这一标准也得到了国际标准化组织(International Standards Organization，ISO)的认可，并形成 ISO-16262 文件。Microsoft 公司的 JavaScript 版本名为 JScript.NET。

虽然 JavaScript 解释器可以嵌入到许多不同的应用程序中，但它最常见的用途是嵌入在 Web 浏览器中。JavaScript 代码嵌入在 HTML 文档中，在文档显示时由浏览器进行解释。在网页程序设计中，JavaScript 的主要用途是验证表单输入数据的正确性，并生成动态的 HTML 文档。JavaScript 现在也与 Rails 网页开发框架一起使用。

虽然名为 JavaScript，但它与 Java 的关联只是采用了相似的语法。Java 是强类型化的，但 JavaScript 是动态类型的(见第 5 章)。JavaScript 的字符串和数组具有可变的长度，因此并不检查数组下标的有效性，而这在 Java 中是要求检查的。Java 完全支持面向对象的程序设计，但 JavaScript 既不支持继承，也不支持将方法调用动态绑定到方法上。

JavaScript 最重要的用途之一是动态创建和修改 HTML 文档。JavaScript 定义了一个对象层次结构，与由文档对象模型(Document Object Model)定义的 HTML 文档的层次模型相吻合。HTML 文档的元素可通过这些对象来访问，提供了动态控制文档元素的基础。

以下是一段 JavaScript 脚本，用于本章前面已用多种语言解决的问题。请注意，假设这段脚本将从一个 HTML 文档中调用，并用 Web 浏览器解释。

```
// example.js
//   Input:  An integer, listLen, where listLen is less
//           than 100, followed by listLen-numeric values
//  Output: The number of input values that are greater
```

```
//            than the average of all input values

var intList = new Array(99);
var listLen, counter, sum = 0, result = 0;

listLen = prompt (
        "Please type the length of the input list", "");
if ((listLen > 0) && (listLen < 100)) {

// Get the input and compute its sum
   for(counter = 0; counter < listLen; counter++) {
      intList[counter] = prompt (
                    "Please type the next number", "");
      sum += parseInt(intList[counter]);
}

// Compute the average
   average = sum / listLen;
// Count the input values that are > average
   for(counter = 0; counter < listLen; counter++)
      if(intList[counter] > average) result++;

// Display the results
   document.write("Number of values > average is: ",
                  result, "<br />");
} else
   document.write(
       "Error - input list length is not legal <br />");
```

2.18.3 PHP 的起源与特点

1994 年，Apache 小组的成员 Rasmus Lerdorf 开发了 PHP 语言(Converse 和 Park，2000)。他的初衷是提供一个工具来跟踪他个人网站的访问者。1995 年，他开发了名为个人主页工具(Personal Home Page Tools)的软件包，这成为 PHP 语言第一个公开发布的版本。PHP 最初是个人主页(Personal Home Page)的简称，后来它的用户群开始使用递归的名称“PHP：超文本预处理器(PHP：Hypertext Preprocessor)”，后来原来的名称不再使用了。PHP 语言现在作为一个开源产品来开发、发布和提供支持。PHP 处理器在大多数网页服务器中都有。

PHP 是专门为 Web 应用设计的内嵌于 HTML 的服务器端脚本语言。当浏览器请求内嵌 PHP 代码的 HTML 文档时，PHP 代码就在 Web 服务器上解释执行。PHP 代码通常生成 HTML 代码作为输出，来替换 HTML 文档中的 PHP 代码，因此 Web 浏览器不会看到 PHP 代码。

PHP 语言与 JavaScript 的语法形式很像，它的字符串和数组本质上都是动态的，并采用动态类型。PHP 的数组结合了 JavaScript 的数组和 Perl 的哈希。

PHP 最初的版本不支持面向对象的程序设计，但是在第二版中添加了支持。不过 PHP 不支持抽象类或接口，不支持析构函数和类成员的访问控制。

PHP 很容易访问 HTML 的表单数据，因此用 PHP 进行表单处理很方便。PHP 对许多不

同的数据库管理系统都有支持，因此是构建需要通过 Web 访问数据库的程序的优秀语言。

2.18.4　Python 的起源与特点

Python(Lutz 和 Ascher，2004)是一种相对较新的面向对象的解释式脚本语言。它是在 20 世纪 90 年代初由 Guido van Rossum 在荷兰的 Stichting Mathematisch Centrum 设计的。它的开发工作现在由 Python 软件基金会(Python Software Foundation)承担。Python 用于与 Perl 相同的领域：系统管理、CGI 程序设计和其他相对较小的计算任务。Python 是一个开源系统，可用于大多数常见的计算平台。PHP 的实现可以在 www.python.org 找到，那里还有与 Python 相关的大量信息。

Python 的语法并不是直接基于任何常用语言的。它有类型检查，但仍是动态类型的。Python 没有使用数组，而是提供了三类数据结构：列表(list)、称为元组(tuple)的不变列表，以及称为字典(dictionary)的哈希结构。Python 有一系列列表方法，例如 append、insert、remove 和 sort，还有一系列字典方法，例如 keys、values、copy 和 has_key。Python 也支持源自于 Haskell 语言的列表推导(list comprehension)，列表推导将在 15.8 节介绍。

Python 是面向对象的，具有 Perl 的模式匹配功能，有异常处理。垃圾收集机制会将不再需要的对象收回。

Python 通过 cgi 模块支持 CGI 程序设计，特别是表单处理，还有支持 cookie、联网(networking)和数据库访问的模块。

Python 支持并发性及线程，也支持网络编程及套接字。Python 对函数式编程的支持多于其他非函数式编程语言。

Python 更有趣的一个特性是，用户可以很容易地扩展它。支持扩展的模块可以用任何可编译的语言写成，扩展可以添加函数、变量和对象类型。这些扩展是作为 Python 解释器的附加部分来实现的。

2.18.5　Ruby 的起源与特点

Ruby(Thomas 等，2005)是 20 世纪 90 年代早期由 Yukihiro Matsumoto(aka Matz)设计的，在 1996 年发布，并在此之后不断地发展。开发 Ruby 的动机是设计者对 Perl 和 Python 都不满意。虽然 Perl 和 Python 都支持面向对象的程序设计[14]，但都不是纯粹的面向对象语言，至少两者都有基本(非对象)类型且都支持函数。

Ruby 的基本特性在于它与 Smalltalk 一样，是一种纯粹的面向对象语言，每一数据值都是对象，所有的操作都通过方法调用来完成。Ruby 中的运算符只是一种语法机制，为相应运算指明具体的方法调用。由于这些运算实际上是方法，因此可以重新定义。所有类，不管是预先定义的还是用户自定义的，都可以有子类。

在 Ruby 中，类和对象都是动态的，因为可以动态地为它们添加方法。这意味着在执

14. 实际上，Python 只部分支持面向对象的程序设计。

行过程中，类和对象在不同时刻可以有不同的方法集。因此同一类的不同实例可以呈现不同的行为。在类的定义中，可以包含方法、数据和常量。

Ruby 的语法与 Eiffel 和 Ada 的语法有关联，由于使用动态类型，因此不需要声明变量。变量的作用域是通过名称来指定的：名称以字母开头的变量具有局部作用域；名称以@开头的变量是实例变量；名称以$开头的变量具有全局作用域。Perl 中的一些特性在 Ruby 中也有，包括名称没有意义的隐式变量如$。

和 Python 一样，Ruby 的用户也可以扩展和修改它。Ruby 在文化上也很有意义，因为它是第一种在日本设计并且在美国得到了相对广泛应用的程序设计语言。

2.18.6 Lua 的起源与特点

Lua[15]是 20 世纪 90 年代早期由 Roberto Ierusalimschy、Waldemar Celes 和 Luis Henrique de Figueiredo 在巴西里约热内卢神学院设计的，是一种支持过程与函数式程序设计的脚本语言，可扩展性是它的一个基本目标。Scheme、Icon 和 Python 对它的设计有影响。

Lua 与 JavaScript 很相似：它不支持面向对象的程序设计，但是显然受到了面向对象程序设计的影响。Lua 和 JavaScript 的对象都可以作为类或对象，都有原型继承而不是类继承。不过，Lua 语言可以扩展为支持面向对象的程序设计。

同 Scheme 一样，Lua 的函数是第一类值(first-class value)的，Lua 还支持闭包，这两个功能使 Lua 可用于函数式程序设计。也如 Scheme 一样，Lua 只有一种数据结构，不过在 Lua 中这一数据结构是表。Lua 的表扩展了 PHP 的关联数组，包含了传统命令式语言的数组，对表元素的引用可以是对传统数组、关联数组或记录的引用。由于函数是第一类值的，因此可以存储在表中，这样的表可起到名称空间的作用。

Lua 使用垃圾收集机制来处理对象，对象都是分配在堆上的。和大多数其他的脚本语言一样，Lua 使用动态类型。

Lua 是一种相对小型且简单的语言，只有 21 个保留字。它的设计理念是提供最基本和相对简单的方式来扩展语言，使它适用于各种应用领域。它的许多扩展都衍生自表这一数据结构，表可以使用 Lua 的元表原理来定制。

Lua 可以方便地用作其他语言的一种脚本语言扩展。就像 Java 早期的实现一样，Lua 是翻译成中间代码后解释执行的。Lua 很容易完全嵌入到其他系统中，部分原因是它的解释器很小，只有约 150KB。

在 2006 年和 2007 年间，Lua 的应用迅速增长，部分是因为它在游戏产业中的应用。在过去 20 多年内出现的一系列脚本语言中，已经有好几种得到了广泛应用。Lua 是最晚出现的一种脚本语言，但迅速成为得到广泛应用的脚本语言之一。

---

15. 名称 Lua 来自于“月亮”的葡萄牙语。

2.19　一流的.NET 语言：C#

Microsoft 公司在 2000 年发布了 C#和新的开发平台.NET[16]，2002 年 1 月发布了两个产品的正式版。

2.19.1　设计过程

C#基于 C++和 Java，但是包含了一些 Delphi 和 Visual BASIC 的思想。它的主要设计者 Anders Hejlsberg 还设计过 Turbo Pascal 和 Delphi，这可以解释 C#中存在 Delphi 成分的原因。

C#的目标是提供一种语言用于基于组件的软件开发，特别是在.NET 框架上进行这样的开发。在这一环境下，各种语言的组件可以很容易地组合成系统。所有的.NET 语言，包括 C#、Visual Basic .NET、托管 C++、F#和 Jscript .NET[17]，都使用通用类型系统(Common Type System，CTS)。CTS 提供了一个通用的类库。所有 5 种.NET 语言的所有类型都继承自一个根类 System.Object。符合 CTS 规范的编译器所生成的对象可以组合到软件系统中。所有 5 种.NET 语言都编译为同一种中间形式——中间语言(Intermediate Language，IL)[18]。不过与 Java 不同，IL 不是解释执行的。在执行前，一个即时编译器会将 IL 翻译为机器代码。

2.19.2　语言概述

很多人认为，与 C++相比，Java 最重要的一个进步在于它去除了 C++中的一些特性。例如，C++支持多继承、指针、struct、enum 类型、运算符重载和 goto 语句，但在 Java 中这些都没有。C#的设计者显然不同意这样大批量地去除 C++的特性，因为这些特性中除了多继承以外，都包含在这一新的语言中。

但是 C#设计者的贡献是，一些 C++的特性在 C#中得到了改进。例如，C#的 enum 类型比 C++中的安全，因为它们不会隐式地转换为整数，这使得它们更加类型安全。struct 类型有巨大的改变，变成一种确实非常有用的结构，然而在 C++中它实际上没什么用。C#的 struct 将在第 12 章中介绍。C#试图改进 C、C++和 Java 中的 switch 语句。C#的 switch 将在第 8 章中介绍。

虽然 C++有函数指针，但是与 C++中变量的指针一样存在固有的不安全性。C#包含一种新的类型——委托(delegate)，是一种面向对象的类型安全的方法引用。委托用于执行事件处理程序、控制线程的执行和回调[19]。回调在 Java 中是通过接口实现的，在 C++中采用

16. .NET 开发系统在第 1 章中有简要介绍。

17. 许多其他语言都已修订为.NET 语言。

18. IL 最初称为 MSIL(意为 Microsoft 中间语言)，但显然很多人认为这个名字太长了。

19. 当一个对象调用另一个对象的方法，并且需要通知调用对象这一方法何时完成任务时，被调用的方法将反过来通知调用对象，这称为回调。

方法的指针。

在 C#中，方法可以有数目不定的参数，只要它们是同一类型的，为此，需要使用数组类型的形参，并在前面加上 params 保留字。

C++和 Java 都有两套不同的类型系统，一套用于基本类型，一套用于对象。除了容易混淆之外，还经常需要在两套系统之间进行值的转换，例如将一个基本类型的数值放入一个存储对象的集合中。C#通过隐式的装箱(boxing)和拆箱(unboxing)操作，在这两套系统之间进行值的半隐式转换。装箱和拆箱操作将在第 12 章中详细介绍[20]。

C#的其他特性包括大部分程序设计语言都不支持的矩形数组(rectangular array)、遍历数组和集合对象的 foreach 语句。Perl、PHP 和 Java 5.0 中也有类似的 foreach 语句。C#也有属性(property)，是公有数据成员的替代品。属性是具有 get 和 set 方法的数据成员，当相关联的数据成员被引用和赋值时，隐式地调用这两个方法。

C#自从 2002 年首次发布后，一直在快速发展。其最新版本是 C# 2010，它添加了一种动态类型化形式、隐式类型化和匿名类型(参见第 6 章)。

2.19.3 评价

C#想成为一种比 C++和 Java 更先进的通用程序设计语言。虽然有人认为 C#的一些特性是一种倒退，但 C#的一些语言结构显然使它超越了前辈语言，在不远的将来，它的一些语言特性一定会被其他程序设计语言所采纳。

以下是 C#程序的一个例子。

```
// C# Example Program
// Input:  An integer, listlen, where listlen is less than
//         100, followed by listlen-integer values.
// Output: The number of input values that are greater
//         than the average of all input values.
using System;
public class Ch2example {
  static void Main() {
    int[] intlist;
    intlistlen,
        counter,
        sum = 0,
        average,
        result = 0;
    intList = new int[99];
    listlen = Int32.Parse(Console.readLine());
    if ((listlen > 0) && (listlen < 100)) {
// Read input into an array and compute the sum
      for(counter = 0; counter < listlen; counter++) {
        intList[counter] =
                        Int32.Parse(Console.readLine());
```

20. Java 5.0 中加入了这一特性。

```
        sum += intList[counter];
      }//- end of for (counter ...
// Compute the average
      average = sum / listlen;
// Count the input values that are > average
      foreach (int num in intList)
        if (num > average) result++;
// Print result
      Console.WriteLine(
         "Number of values > average is:" + result);
    } //- end of if ((listlen ...
    else
      Console.WriteLine(
         "Error--input list length is not legal");
  }//- end of method Main
} //- end of class Ch2example
```

2.20　标记与程序设计混合的语言

标记与程序设计混合的语言是一种标记语言，其中一些语言元素可以指定程序设计动作，例如控制流程和计算。以下几小节介绍两种这样的混合语言，XSLT 和 JSP。

2.20.1　XSLT

可扩展标记语言(eXtensible Markup Language，XML)是一种元标记语言，这种语言是用于定义标记语言的。XML 衍生的标记语言用于定义数据文档，称为 XML 文档。虽然人可以阅读 XML 文档，但它们是由计算机处理的，这种处理有时只是形式上的转换，以便于有效地显示或打印。在许多情况下转换为 HTML，可以由 Web 浏览器来显示。在其他情况下，可以处理文档中的数据，就像处理其他形式的数据文件一样。

从 XML 文档到 HTML 文档的转换是由另一种标记语言——可扩展样式表语言转换(eXtensible Stylesheet Language Transformations，XSLT)——指定的(www.w3.org/TR/XSLT)。XSLT 可以指定类似程序设计的操作。因此，XSLT 是一种标记与程序设计混合的语言。XSLT 是由万维网联盟(World Wide Web Consortium，W3C)于 20 世纪 90 年代晚期定义的。

XSLT 处理器是一个程序，以 XML 数据文档和 XSLT 文档(也是以 XML 文档的形式)为输入，在处理过程中，使用 XSLT 文档中说明的转换规则，将 XML 数据文档转换成另一种 XML 文档[21]。XSLT 文档通过定义模板来指定转换规则，模板是 XSLT 处理器可以在 XML 输入文件中找到的数据模式。与 XSLT 文档中每一个模板相关联的是它的转换指令，指明了在放入输出文档之前如何转换匹配的数据。因此模板(和相关联的处理过程)就像是子程序，当 XSLT 处理器在 XML 文档的数据中找到一个模式匹配时，就执行这个子程序。

21. XSLT 处理器的输出文档也可以是 HTML 文档或者纯文本。

XSLT 也有低级的程序设计结构，例如有一个循环结构，允许选择 XML 文档中重复的部分。它还有一个排序过程。这些低级的语言结构是用 XSLT 标签来指定的，例如<for-each>。

2.20.2 JSP

Java 服务端网页标准标签库(Java Server Pages Standard Tag Library，JSTL)的核心部分是另一种标记与程序设计混合的语言，虽然它的形式和用途与 XSLT 不同。在介绍 JSTL 之前，有必要介绍小服务程序(servlet)和 Java 服务端网页(Java Server Pages，JSP)的概念。小服务程序是一个 Java 类的实例，在 Web 服务器系统中驻存并执行。小服务程序的执行是应 Web 浏览器当前显示的标记文档的请求。小服务程序的输出以 HTML 文档的形式返回给发出请求的浏览器。在 Web 服务器进程中执行的程序，称为小服务程序容器(servlet container)，控制小服务程序的执行。小服务程序通常用于表单处理和数据库访问。

JSP 是一些技术的集合，用于支持动态网页文档，提供 Web 文档的其他一些处理需求。当浏览器请求 JSP 文档(通常是 HTML 和 Java 的混合)时，驻存在 Web 服务器系统中的 JSP 处理器程序将文档转换为小服务程序。文档中嵌入的 Java 代码也复制到小服务程序，纯 HTML 复制到 Java 打印语句中并原样输出。JSP 文档中的 JSTL 标记像下一段落中介绍的那样处理。JSP 处理器生成的小服务程序由小服务程序容器来运行。

JSTL 定义了一组 XML 操作元素，控制 JSP 文档在 Web 服务器上的处理。这些元素与 HTML 和 XML 的其他元素有着相同的形式。JSTL 的控制操作元素中最常用的一个是 if，它将布尔表达式指定为属性[22]。if 元素的内容(开始标签<if>和结束标签</if>之间的文本)是 HTML 代码，只有当布尔表达式取值为真时，才包含在输出文档中。if 元素与 C/C++中的 #if 预处理器指令相似。JSP 容器处理 JSP 文档中的 JSTL 标记部分，其方式与 C/C++预处理器处理 C 和 C++程序的方式类似。预处理器指令指挥预处理器如何根据输入文件组织输出文件。同样，JSTL 的控制动作元素指挥 JSP 处理器如何根据 XML 输入文件构建 XML 输出文件。

if 元素的一个常见用途是验证浏览器用户所提交的表单数据的有效性。JSP 处理器可以访问表单数据，用 if 元素来测试，确保表单元素是有意义的。否则，if 元素可以在输出文档中为用户插入一条错误信息。

在多选控制上，JSTL 有 choose、when 和 otherwise 元素。JSTL 还有 forEach 元素，可以在集合上迭代，集合通常是客户端的表单值。forEach 元素可以有 begin、end 和 step 属性来控制它的迭代。

小　结

我们系统地介绍了一些程序设计语言的开发过程和开发背景。本章较好地展现了当前

22. HTML 中的属性嵌入在元素的开始标签中，提供这一元素的额外信息。

语言设计的问题，为深入讨论当代语言的重要特性打下了基础。

文献注释

早期程序设计语言发展的历史资料最重要的来源也许是由 Richard Wexelblat 编辑的 *History of Programming Languages* 一书(1981)。这本书包含了 13 种重要的程序设计语言的开发背景和环境，都是由语言设计者们自己讲述的。一本相似的著作来自于又一次“历史”会议，这次出版了 *ACM SIGPLAN Notices* 特刊(ACM，1993a)。这本书介绍了另外 13 种程序设计语言的历史和发展。

文章《程序设计语言的早期发展(Early Development of Programming Languages)》(Knuth 和 Pardo，1977)是 *Encyclopedia of Computer Science and Technology* 的一部分，是一篇优秀的 85 页长的文章，详细介绍了直到 Fortran 为止的语言的发展过程。这篇文章包含了一些例程，展示了其中许多语言的特性。

另一本很有意义的书是 Jean Sammet 所著的 *Programming Languages: History and Fundamentals*(Sammet，1969)。这本 785 页的书记载了 20 世纪 50 年代到 60 年代的 80 种程序设计语言的细节。Sammet 还出版了这本书的一些更新内容，例如 *Roster of Programming Languages for 1974-1975* (1976)。

复 习 题

1. Plankalkül 是哪一年设计的？在哪一年发布？
2. Plankalkül 中有哪两种常用的数据结构？
3. 20 世纪 50 年代早期，伪代码是如何实现的？
4. Speedcoding 的开发是为了克服 20 世纪 50 年代早期计算机硬件的两个重大缺点，是哪两个缺点？
5. 为什么在 20 世纪 50 年代早期，程序解释执行的速度很慢但人们仍可以接受？
6. 在 IBM 704 计算机中首次出现的哪一种硬件功能，对程序设计语言的发展产生了巨大的影响？为什么？
7. Fortran 语言的设计项目是哪一年开始的？
8. 设计 Fortran 语言的时代，计算机主要的应用领域是什么？
9. Fortran I 的流程控制语句的来源是什么？
10. 从 Fortran I 变为 Fortran II，增加的最重要的特性是什么？
11. 从 Fortran IV 变为 Fortran 77，增加了哪些流程控制语句？
12. 哪一个 Fortran 语言版本最先有动态变量？
13. 哪一个 Fortran 语言版本最先有字符串处理？
14. 为什么 20 世纪 50 年代晚期，语言学家对人工智能感兴趣？
15. LISP 是在哪里开发的？是由谁开发的？
16. Scheme 和 Common LISP 在哪一方面正好相反？
17. LISP 的哪一种方言在一些大学用于程序设计的概述课程？

18. ALGOL 60 是由哪两个专业组织共同设计的？
19. 在 ALGOL 的哪一个版本中出现了块结构？
20. AIGOL 60 缺少哪一种语言元素，使它没有得到广泛应用？
21. 人们设计了哪种语言来描述 ALGOL 60 的语法？
22. COBOL 是基于哪种语言的？
23. COBOL 的设计过程始于哪一年？
24. 最早来自于 Plankalktül，后来出现在 COBOL 中的数据结构是什么？
25. COBOL 早期的成功(从广泛应用的程度来看)主要归因于哪一个组织？
26. BASIC 最初版本的目标用户群体是什么？
27. 为什么 BASIC 是 20 世纪 80 年代早期的一种重要语言？
28. PL/I 是为了替代哪两种语言而设计的？
29. PL/I 是为了哪一种新型的系列计算机而设计的？
30. SIMULA 67 的哪些特性是现在一些面向对象语言的重要部分？
31. 哪一种创新的数据结构是在 ALGOL 68 中引入的，但常常归功于 Pascal？
32. 在 ALGOL 68 中，大量使用了哪个设计原则？
33. 哪一种语言引入了 case 语句？
34. C 语言的哪些运算符模仿了 ALGOL 68 中类似的运算符？
35. C 语言的哪两个特点使它不如 Pascal 安全？
36. 什么是非过程式语言？
37. Prolog 数据库的两种语句是什么？
38. Ada 主要是为哪一种应用领域而设计的？
39. Ada 中并发的程序单元叫什么？
40. Ada 的哪种语言结构提供了对抽象数据类型的支持？
41. Smalltalk 的世界是由什么构成的？
42. 面向对象程序设计的基础是哪三个概念？
43. 为什么 C++包含了 C 语言中被认为不安全的特性？
44. Objective-C 从哪种语言中借用了方法调用的语法？
45. 几乎所有目前设计的语言都支持哪个程序设计范型，而 Go 语言不支持？
46. Objective-C 的主要应用领域是什么？
47. 同时参与设计了 C 和 Go 的设计者是谁？
48. Ada 和 COBOL 语言的共同点是什么？
49. Java 的最初应用是什么？
50. Java 的什么特点在 JavaScript 中表现最明显？
51. PHP 和 JavaScript 的类型系统与 Java 的类型系统有何区别？
52. 在 C#中有，但在 C、C++和 Java 中没有的数组结构是什么？
53. Perl 的最初版本是为了替代哪两种语言？
54. JavaScript 应用最广泛的领域是什么？
55. JavaScript 和 PHP 在使用上有什么关系？
56. PHP 的主要数据结构是其他语言中哪两种数据结构的组合？

57. Python 代替数组所使用的数据结构是什么？
58. Ruby 与 Smalltalk 相同的特点是什么？
59. Ruby 算术运算符的什么特点使它们与其他语言中的算术运算符不一样？
60. Lua 中内置的数据结构是什么？
61. Lua 通常是编译的、完全解释的，还是不完全解释的？
62. Delphi 类的什么特性也包含在 C#中？
63. C#对 switch 语句的改变反映了 C 语言中 switch 语句的什么缺陷？
64. C#使用的主要平台是什么？
65. XSLT 处理器的输入是什么？
66. XSLT 处理器的输出是什么？
67. JSTL 的哪一种元素与子程序相关？
68. JSP 处理器将 JSP 文档转换成什么？
69. servlet 是在哪里执行的？

习　题

1. 如果 Fortran 语言的设计者熟悉 Plankalkül，你认为 Plankalkül 的什么特性对 Fortran 0 产生的影响最大？
2. Backus 的 701 Speedcoding 系统有哪些功能？将这些功能与现在的可编程手持计算器的功能做比较。
3. 写出由 Grace Hopper 和她的助手所设计的 A-0、A-1 和 A-2 系统的简史。
4. 作为一个研究项目，比较 Fortran 0 和 Laning 与 Zierler 系统的机制。
5. 在你看来，在 ALGOL 设计委员会的三个最初目标中，哪一个在当时最难以实现？
6. 理性地猜测一下，在 LISP 程序中最常见的语法错误是什么？
7. LISP 起初是纯函数式的语言，但是逐渐有了越来越多的命令式特性，为什么？
8. ALGOL 60 没有成为一种应用非常广泛的语言，详细描述在你看来最重要的三个原因。
9. 在你看来，为什么 COBOL 允许长标识符，而 Fortran 和 ALGOL 不允许？
10. 简述 IBM 开发 PL/I 的主要动机。
11. 根据 1964 年以来计算机和语言发展的历史，IBM 决定开发 PL/I 所依据的假设成立吗？
12. 用你自己的话来说明编程语言设计中的正交性概念。
13. PL/I 比 ALGOL 68 应用更为广泛的主要原因是什么？
14. 支持和反对无类型语言的理由各是什么？
15. 除了 Prolog 以外，还有逻辑程序设计语言吗？
16. 太复杂的语言使用起来太危险，因此我们应该使所有语言都小巧简单，你对这种观点的看法是什么？
17. 你认为由委员会来设计语言是个好主意吗？给出你的理由。
18. 语言在不断发展。对于程序设计语言的修改，你认为应该有什么样的限制？将你的答案与 Fortran 的发展做比较。
19. 建一个表格，标记出所有主要的语言发展过程，同时记录什么时候发生、最初出现在

哪一种语言中，以及开发人员的身份。

20. 对于 Microsoft J++和 C#以及 Sun 公司的 Java 的设计，Microsoft 公司和 Sun 公司有过一些公开的交流。阅读一些这样的文档(可以在相关网站上找到)，对两个公司关于委托的不同意见写出你的分析。
21. 脚本语言在近几年中发展的数据结构代替了传统的数组。按年代顺序说明这一发展过程。
22. 给出两个理由，说明为什么完全解释对于一些最新的脚本语言来说是可接受的实现方法。
23. Perl 6 推出时，很可能是一种规模非常大的语言。请理性地猜测一下，像 Lua 这样的语言是否也会不断地扩大规模。给出你的理由。
24. 在你看来，为什么新的脚本语言比新的编译语言更频繁地出现？
25. 简要描述一种标记与程序设计混合的语言。

编程练习

1. 为了理解记录在程序设计语言中的价值，用基于 C 的语言写一段小程序，使用 struct 数组存储学生信息，包括姓名、年龄、GPA(浮点数)和年级(字符串，例如“新生”等)。用同样的语言写出同样功能的程序，但不使用 struct。
2. 为了理解递归在程序设计语言中的价值，写一个实现快速排序的程序，先用递归，然后不用递归。
3. 为了理解计数循环的价值，写一个程序，用计数循环结构实现矩阵乘法。然后只用逻辑循环(例如 while 循环)写出同样功能的程序。

访　谈

用户设计与语言设计

Alan cooper

畅销书 *About Face:The Essentials of User Interface Design* 的作者 Alan Cooper 还深度参与了号称最关心用户界面设计的语言 Visual Basic 的设计工作。对他来说，这一切都出于人性化技术的梦想。

一些背景信息

你是如何开始这些工作的？

我曾从高中辍学，后来在加利福尼亚社区学校的程序设计专业读了两年。我的第一份工作是在旧金山为美国总统轮船公司(American President Lines，美国最老的海洋运输公司之一)做程序员。除了零星地工作过几个月外，我一直是个“个体户”。

你目前的工作是什么？

Cooper 公司(www. cooper. com)的创始人和主席，这个公司将技术人性化。

你最喜欢的工作是什么？

交互操作设计顾问。

你在语言设计和用户界面设计方面非常有名。你对设计语言、设计软件和设计其他东西有什么想法吗？

在软件的世界中，这个想法基本相同：了解你的用户。

关于早期的 Windows 系统

20 世纪 80 年代，你开始使用 Windows 系统，并谈到过被它的特点所吸引：对图形用户界面的支持，以及能够创建自我配置工具的动态链接库。你最终帮助构建了 Windows 系统的哪些部分？

我对 Microsoft 公司在 Windows 系统中加入实际的多任务支持印象深刻。这包括了动态再定位和进程间通讯。MSDOS.exe 是 Windows 系统最初几个版本的外壳程序。这是一个很糟糕的程序，我相信它可以得到大大的改进，我就是那个改进它的人。我立刻开始利用业余时间，写一个比 Windows 系统随带的更好的外壳程序，我将它称为 Tripod。Microsoft 公司最初的外壳程序 MSDOS.exe 是 Windows 系统走向成功的主要绊脚石之一。Tripod 试图以更易使用和配置的方式来解决这个问题。

你的灵感何时涌现？

1987 年底，我在与一个公司客户面谈时，Tripod 的关键设计策略突然跳进我的头脑中。当这个 IS 经理向我解释，他需要给不同的用户创建和发布不同的外壳程序解决方案时，我意识到，其实根本就没有完美的外壳程序。每一个用户都需要自己的外壳程序，根据自己的需求和技能水平来配置。我立刻意识到外壳程序设计问题的解决方案是：一个外壳程序的结构集，每个用户都能使用这个工具构建需要的外壳程序，满足不同的应用和训练需求。

个性化外壳程序的想法最吸引人的地方是什么？

用户可以设计自己的个性化外壳程序，而不是由我来告诉他们理想的外壳程序是什么。通过可定制的外壳程序，程序员可以创建一个功能强大、应用广泛、同时有一些危险的外壳程序，而 IT 经理可以创建一个给接待人员用的外壳程序，只显示接待人员要用的少数专用工具。

你是如何从编写外壳程序到与 Microsoft 公司合作的？

Tripod 和 Ruby 是同一个东西。在我与 Bill Gates 签了一份协议之后，我把原型系统的名称由 Tripod 改为 Ruby。然后我以应该使用原型系统的方式来使用 Ruby 原型系统：用于构建正式版质量代码的可丢弃的模型。我当时就是这么做的。Microsoft 公司取得了 Ruby 的正式版，添加了 QuickBASIC，创建了 VB。所有原始创新都在 Tripod/Ruby 中。

Ruby 是 Visual Basic 的孵化器

请谈谈你对早期 Windows 系统和 DLL 特性的兴趣。

DLL 不是独立的，而是操作系统的一个机制，允许程序员创建可在运行时而不是只在编译时链接的代码对象。就是这一机制使我发明了 VB 的动态可扩展部分，第三方厂商可在其中添加控件。

Ruby 产品在软件设计方面有许多重大的进步，但其中的两项格外成功。我前面提到，我一直对 Windows 系统的动态链接功能很感兴趣，但是拥有工具和知道如何使用工具是两码事。通过 Ruby，我最终发现了动态链接的两个实际用途，都用在了最初的程序中。首先，语言可以安装也可以动态扩展；其次，新发明的小模块可以动态添加。

Ruby 中的语言是第一种具有动态链接库并且链接到可视化前端的语言吗？

据我所知是的。

请用一个简单的例子来说明，这能够让程序员对程序做些什么呢？

从第三方厂商处购买一个控件，例如网格控件，安装到计算机上，就可以使这个网格控件像是语言的一个组成部分那样显示出来，包括可视化的程序设计前端。

为什么人们称你为“Visual Basic 之父”？

Ruby 带有一种小型语言，只适用于执行外壳程序所需的十几条简单命令。但这种语言实现为一个 DLL 链，可以在运行时装载任意数量的 DLL。内置的分析器可以辨识出一个动作，并沿着 DLL 链传递，直到某一个 DLL 确认知道如何处理这个动作。如果所有的 DLL 都不知道如何处理，则存在一个语法错误。根据我们最早的讨论，Microsoft 公司和我都接受扩展这种语言的想法，甚至可能用一种“真正的”语言来代替它。C 语言是最经常提到的候选，但最终 Microsoft 公司利用这种动态接口去掉了小型外壳程序语言，完全替换为 Quick-BASIC。这种崭新的语言和可视化前端的结合是静态的、永久的，虽然最初的动态接口使这样的结合成为可能，但在结合过程中被丢弃了。

关于新思想的最后几句话

在程序设计和程序设计工具的世界中，包括语言和开发环境，你最感兴趣的是哪些项目？

我感兴趣的是创建程序设计工具，这些工具是为帮助用户而不是程序员而设计的。

你认为最重要的原则、著名语录或设计理念是什么？

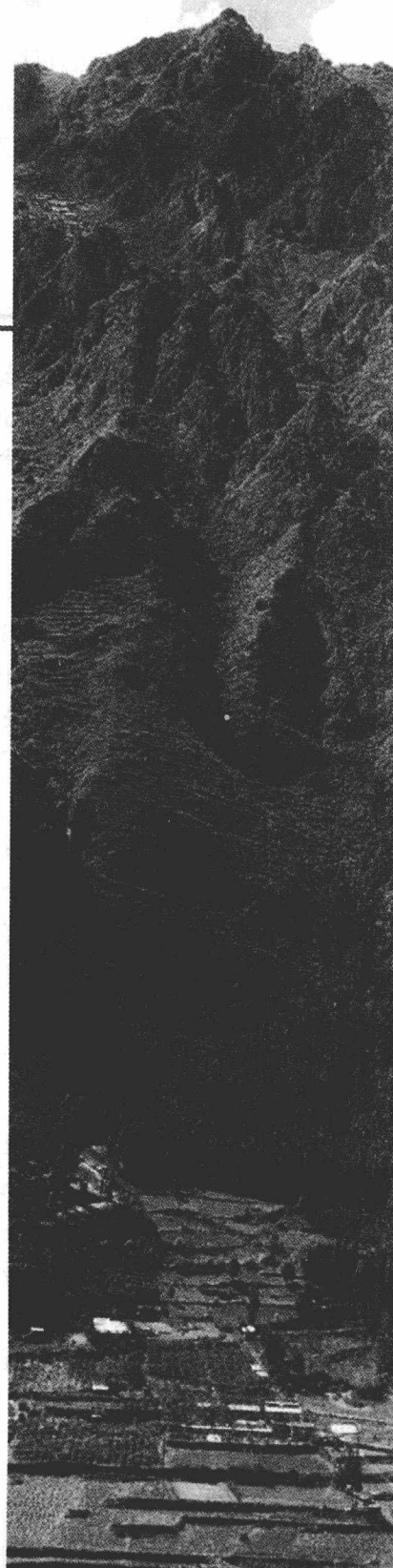
桥不是由工程师建造的，是由钢铁工人建造的。

同样，软件程序也不是由工程师编写的，是由程序员编写的。

第3章

描述语法和语义

3.1 概述
3.2 描述语法的普遍问题
3.3 描述语法的形式化方法
3.4 属性文法
3.5 描述程序的意义：动态语义



本章包括以下主题：首先定义语法和语义这两个术语；然后详细介绍描述语法最常用的方法——上下文无关文法(也称为巴科斯-诺尔范式，Backus-Naur Form)，介绍的内容包括推导、语法分析树、歧义、运算符优先级和结合性，以及扩展的巴科斯-诺尔范式；随后介绍属性文法，它可用于描述程序设计语言的语法和静态语义；最后介绍描述语义的三种形式化方法——操作语义、公理语义和指称语义。由于语义描述方法内在的复杂性，我们只能简要地介绍。三种方法中的任何一种都需要一本书的篇幅来介绍(如同一些作者已经做的那样)。

3.1 概述

为程序设计语言提供一种简明扼要又易于理解的描述是很困难的，但这对语言的成功又是必不可少的。ALGOL60 和 ALGOL68 都先给出了简明扼要的形式化描述，但是这两种语言的描述都不易理解，部分原因是它们都用了一种新的标记方法，结果这两种语言的认可程度都不高。另一方面，一些语言有许多差异甚小的方言，因为其定义很简单、非形式化、不严谨。

描述语言的难题之一是各种各样的人都必须能理解这种描述，其中包括最初的评估人员、实现者和使用者。大多数新的程序设计语言在设计完成前，都会由潜在用户(通常在雇用语言设计者的机构内部)进行仔细的审查。这些人就是最初的评估人员。这个反馈周期是否成功，很大程度取决于语言描述是否清晰。

显然，程序设计语言的实现者必须能够确定语言的表达式、语句和程序单元如何构成，以及执行的预期效果。实现者的任务难度，部分取决于语言描述的完整性和准确性。

最后，语言的使用者必须能够查阅语言参考手册，确定如何编写软件解决方案。教科书和课程已介绍了这一过程，但语言手册通常是该语言唯一权威的、已出版的信息来源。

学习程序设计语言与学习自然语言一样，可以分为语法的学习和语义的学习。程序设计语言的语法是它的表达式、语句和程序单元的形式；它的语义是这些表达式、语句和程序单元的意义。例如，Java 中 while 语句的语法是

while (布尔表达式) 语句

这一语句形式的语义是：当布尔表达式的当前值为真时，执行嵌入的语句，否则继续执行 while 结构后面的语句。然后程序控制隐式地返回到布尔表达式，以重复这一过程。

虽然为了讨论方便，语法和语义通常分开介绍，其实它们是紧密相关的。在设计良好的程序设计语言中，语法应该直接体现语义，也就是说，语句的表现形式应该强烈暗示语句要完成的功能。

描述语法比描述语义简单，部分原因是语法描述有简明和普遍接受的标记方法，但是语义描述还没有。

3.2　描述语法的普遍问题

语言，不管是自然语言(如英语)还是人造语言(如 Java)，是一个由字母表中的字符组成的串的集合。语言的串称为句子或语句。语言的语法规则表明语言字母表中的哪些字符串是属于该语言的，例如英语有庞大、复杂的规则集，来表示英语句子的语法。相比较而言，即使是规模最大最复杂的程序设计语言，其语法也是非常简单的。

为了简单起见，程序设计语言的语法的形式化描述通常不包括最低级语法单元的描述。这些小单元称为词素。词素的描述由词法说明给出，通常与语言的语法描述是分开的。程序设计语言的词素包括数字的字面值、运算符和特殊字等。可以将程序看作是词素串，而不是字符串。

词素可分为几组，例如，程序设计语言中变量、方法、类等的名称，组成了标识符组。每一个词素组是用名称或标记(token)表示的，因此语言的标记是一类词素。例如标识符是一个标记，它可以有词素或实例，例如 sum 和 total。在某些情况下，标记只有一个可能的词素，例如表示算术运算符+的标记只有一个可能的词素。考虑以下 Java 语句：

```
index = 2 * count + 17;
```

这一语句的词素和标记分别是：

词素	标记
index	标识符
=	等于符号
2	整数字面值
*	乘法运算符
Count	标识符
+	加法运算符
17	整数字面值
;	分号

本章中的语言描述示例都非常简单，大部分都包括词素的描述。

3.2.1　语言识别器

通常，语言可以用两种不同的方法来形式化地定义：识别的方法和生成的方法(尽管对于想学习甚至使用程序设计语言的人来说，这两种方法都无法给出语言的有效定义)。假设有一种语言 L，所用的字符集为Σ，为了使用识别方法来形式化地定义 L，需要构造一个称为识别设备的装置 R，来读入字符集Σ中字符组成的串。R 可以指出，给定的输入串属于或不属于 L。实际上，R 或者接受或者拒绝输入串。这样的设备就像过滤器一样，将合法的句子与错误的句子分开。如果输入任何Σ上的字符串，R 只在它属于 L 时接受它，则 R 是 L 的一种描述。出于实用的目的，大多数有用的语言都是无限的，因此这种识别过程似乎冗长而低效。不过，识别设备不是用来枚举语言的所有句子的，它们有别的用途。

编译器的语法分析部分是编译器所翻译的语言的识别器。在这一任务中，识别器不必测试某个字符集上所有可能的字符串，确定每个串是否属于这种语言。相反，它只需要确定给定的程序是否属于这种语言。因此实际上，语法分析器确定给定的程序在语法上是否正确。语法分析器(也称为语法分析程序)的组成参见第 4 章。

3.2.2 语言生成器

语言生成器是用来生成语言句子的一种装置，可以将生成器想象为有一个按钮，每次按下都会生成该语言的一个句子。由于按下按钮时，生成器生成的句子是不可预测的，因此生成器作为语言描述设备似乎用处有限。但是比起识别器，人们更喜欢某些形式的生成器，因为它们更容易阅读和理解。相反，编译器的语法检查部分(语言识别器)对于程序员来说不是很有用的语言描述，因为它只能在试错模式下使用。例如，为了用编译器来确定某语句的正确语法，程序员只能提交一个猜测的版本，并观察编译器是否接受它。另一方面，将它与生成器的结构相比较，通常可以确定某语句的语法是否正确。

对于同一种语言来说，形式化的生成器和识别器有密切的联系，这是计算机科学的重大发现之一，它在很大程度上促成了目前的形式语言和编译器设计理论。下一节将讨论生成器和识别器的关系。

3.3 描述语法的形式化方法

本节介绍形式化的语言生成机制，通常称为文法，常用来描述程序设计语言的语法。

3.3.1 巴科斯-诺尔范式和上下文无关文法

20 世纪 50 年代中期至晚期，Noam Chomsky 和 John Backus 两人在各自独立的研究工作中，提出了相同的语法描述形式化理论，后来成为描述程序设计语言语法使用最广泛的方法。

3.3.1.1 上下文无关文法

20 世纪 50 年代中期，著名语言学家 Chomsky 描述了四种生成方法(或文法)，定义了四种语言(Chomsky，1956，1959)。其中两种文法称为上下文无关文法和正则文法，可用于描述程序设计语言的语法。程序设计语言的标记形式可以用正则文法来描述。而整个程序设计语言的语法，除了极少数例外，都可以用上下文无关文法来描述。由于 Chomsky 是一个语言学家，他主要的兴趣在于自然语言的理论本质，当时他对与计算机通信的人造语言没有兴趣。因此直到后来，他的研究成果才应用到程序设计语言中。

3.3.1.2 巴科斯-诺尔范式的起源

在 Chomsky 取得关于语言分类的研究成果之后不久，ACM-GAMM 工作组开始设计

ALGOL 58。ACM-GAMM 工作组的杰出成员 John Backus 在 1959 年的国际会议上，发表了一篇描述 ALGOL 58 的具有里程碑意义的论文(Backus，1959)。这篇论文介绍了表示程序设计语言语法的一种新的形式化标记方法。这一新的标记方法后来由 Peter Naur 稍作修改，用于描述 ALGOL 60(Naur，1960)。修订后的语法描述方法称为巴科斯-诺尔范式，简称 BNF。

BNF 是一种描述语法的自然标记方法。实际上，公元前几百年 Panini 就曾使用类似于 BNF 的方法来描述梵语的语法(Ingerman，1967)。

虽然计算机使用者没有马上接受 ALGOL 60 报告中使用的 BNF，但是它很快就成为并且一直是简要描述程序设计语言语法的最普遍的方法。

注意，BNF 与 Chomsky 的上下文无关语言的生成方法(称为上下文无关文法)几乎一样。本章的后面将上下文无关文法简称为文法，而且，术语 BNF 和文法将互换使用。

3.3.1.3　基本原理

元语言是描述其他语言的语言。BNF 是描述程序设计语言的元语言。

BNF 对语法结构进行抽象。例如一个简单的 Java 赋值语句可以抽象地表示为<赋值>(尖括号通常用于界定抽象的名称)。<赋值>的实际定义是：

<赋值>→<变量>=<表达式>

箭头左边的文本称为左手边(1eft-hand side，LHS)，是所定义的抽象。箭头右边的文本是 LHS 的定义，称为右手边(right-hand side，RHS)，由标记、词素以及其他抽象的引用所组成(实际上，标记也是抽象)。整个定义称为规则或产生式。在上述规则例子中，为了使<赋值>的定义有意义，抽象<变量>和<表达式>显然也必须定义。

以上规则表明，抽象<赋值>定义为抽象<变量>的一个实例，后跟词素=和抽象<表达式>的一个实例。由这条规则描述语法结构的一个例句是

```
total = subtotal1 + subtotal2
```

在 BNF 描述(或文法)中，抽象通常称为非终结符，规则的词素和标记称为终结符。BNF 描述(或文法)是一个规则集。

非终结符可以有两种或多种不同的定义，表示语言的两种或多种可能的语法形式。使用符号|(表示逻辑或)来分隔不同的定义，就可以将多个定义写成一条规则。例如，Java 的 if 语句可以用两条规则来描述

```
<if 语句> → if (<逻辑表达式>) <语句>
<if 语句> → if (<逻辑表达式>) <语句> else <语句>
```

或者用一条规则来描述

```
<if 语句> → if (<逻辑表达式>) <语句>
          | if (<逻辑表达式>) <语句> else <语句>
```

在这些规则中，<语句>表示单条语句或一条复合语句。

虽然 BNF 很简单，但是它足以描述程序设计语言几乎所有的语法。特别是，它可以描述相似语言结构的列表、不同语言结构出现的顺序，以及任意深度的嵌套结构，甚至可以表示运算符的优先级和结合性。

3.3.1.4 描述列表

变长的列表在数学中通常用省略号(…)来表示，例如 1, 2,…。BNF 不包含省略号，因此需要一种替代方法来描述程序设计语言中的语法元素列表(例如在数据声明语句中出现的标识符列表)。对 BNF 来说，这种替代方法就是递归。如果规则的 LHS 出现在 RHS 中，这条规则就是递归的。以下规则展示了如何使用递归来描述列表：

```
<标识符列表> → 标识符
             | 标识符，<标识符列表>
```

这条规则把<标识符列表>定义为单个标记(标识符)，或者一个标识符后跟一个逗号和另一个<标识符列表>的实例。本章余下的部分将使用递归来描述许多示例文法中的列表。

3.3.1.5 文法和推导

文法是一种定义语言的生成工具。通过对规则的一系列应用，可以生成语言的句子，该句子的开头是文法的一个特定非终结符，称为起始符号。所应用的这一系列规则称为推导。在完整的程序设计语言文法中，起始符号表示一个完整的程序，通常名为<程序>。例 3.1 所示的简单文法说明了什么是推导。

例 3.1 一种简单语言的文法

```
<程序>      → begin <语句列表> end
<语句列表> → <语句>
               | <语句>; <语句列表>
<语句> → <变量> = <表达式>
<变量> → A | B | C
<表达式> → <变量> + <变量>
           | <变量> - <变量>
           | <变量>
```

例 3.1 中文法描述的语言只有一种语句形式：赋值。程序包括特殊字 begin，后跟由分号分隔的一系列语句，最后是特殊字 end。表达式是单个变量，或者用+或-运算符分隔的两个变量。在这种语言中，变量名只有 A、B 和 C。

这种语言的程序推导如下：

```
<程序> => begin <语句列表> end
       => begin <语句> ; <语句列表> end
       => begin <变量> = <表达式> ; <语句列表> end
       => begin A = <表达式> ; <语句列表> end
```

```
=> begin A = <变量> + <变量> ; <语句列表> end
=> begin A = B + <变量> ; <语句列表> end
=> begin A = B + C ; <语句列表> end
=> begin A = B + C ; <语句> end
=> begin A = B + C ; <变量> = <表达式> end
=> begin A = B + C ; B = <表达式> end
=> begin A = B + C ; B = <变量> end
=> begin A = B + C ; B = C end
```

同所有推导一样，这个推导从起始符号(在这个例子中是<程序>)开始。符号=>读作“推出”。推导序列中，每一个后面的字符串都是用非终结符的一个定义来替换前面字符串中的一个非终结符而推出的。推导过程中的每个字符串，包括<程序>，都称为句型。

在上面的推导中，被替换的非终结符总是前面一个句型中最左边的非终结符。以这种替换顺序进行的推导称为最左推导。推导过程一直进行到句型不包含非终结符为止。这个只包含终结符(或词素)的句型是生成的句子。

除了最左推导外，还有最右推导，以及既不是最左也不是最右的推导。推导顺序对文法生成的语言没有影响。

选择不同的规则 RHS 来替换推导中的非终结符，可以生成语言的不同句子。穷举所有选择的组合，就可以生成完整的语言。这种语言与其他大多数语言一样，是无限的，因此无法在有限的时间内生成语言中的所有句子。

例 3.2 是一种典型的程序设计语言的部分文法。

例 3.2　简单赋值语句的文法

```
<赋值> →<标识符> = <表达式>
<标识符> →A | B | C
<表达式> →<标识符> + <表达式>
         | <标识符> * <表达式>
         | (<表达式>)
         | <标识符>
```

例 3.2 的文法描述了赋值语句，它的右边是具有乘法、加法运算符以及圆括号的算术表达式。例如，语句

```
A = B * ( A + C )
```

由以下最左推导生成：

```
<赋值> => <标识符> = <表达式>
       => A = <表达式>
       => A = <标识符> * <表达式>
       => A = B * <表达式>
       => A = B * ( <表达式> )
       => A = B * ( <标识符> + <表达式> )
       => A = B * ( A + <表达式> )
```

```
=> A = B * ( A + <标识符> )
=> A = B * ( A + C )
```

3.3.1.6 语法分析树

文法最吸引人的特性，是它们自然地描述了所定义语言句子的分层的语法结构。这种分层结构称为语法分析树。例如，图 3-1 中的语法分析树展示了前面推导出的赋值语句的结构。

语法分析树的每个中间结点都标记着非终结符，每个叶子结点都标记着终结符。语法分析树的每一棵子树都描述了句子中一个抽象的实例。

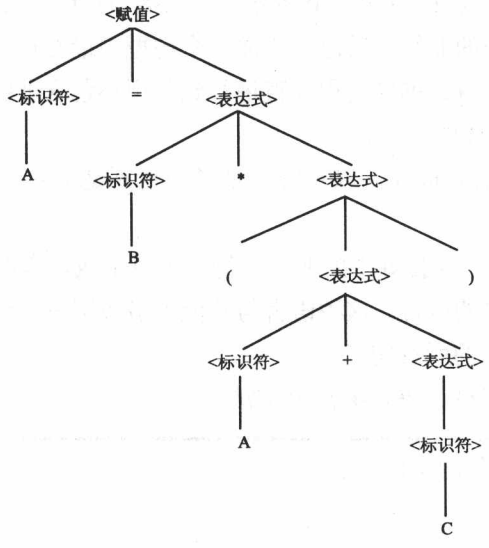


图 3-1 简单语句 A=B*(A+C)的语法分析树

3.3.1.7 歧义

文法生成的句型有两种或多种不同的语法分析树，称为有歧义。考虑例 3.3 所示的文法，这与例 3.2 中的文法只有极小的区别。

例 3.3 简单赋值语句的歧义文法

```
<赋值> → <标识符> = <表达式>
<标识符> → A | B | C
<表达式> → <表达式> + <表达式>
          | <表达式> * <表达式>
          |  (<表达式>)
          | <标识符>
```

例 3.3 的文法是有歧义的，因为句子

```
A = B + C * A
```

有两种不同的语法分析树，如图 3-2 所示。歧义的产生是因为与例 3.2 相比，这个文法说明的语法结构稍少了一些，表达式的语法分析树并不只允许向右生长，而是允许同时向左和向右生长。

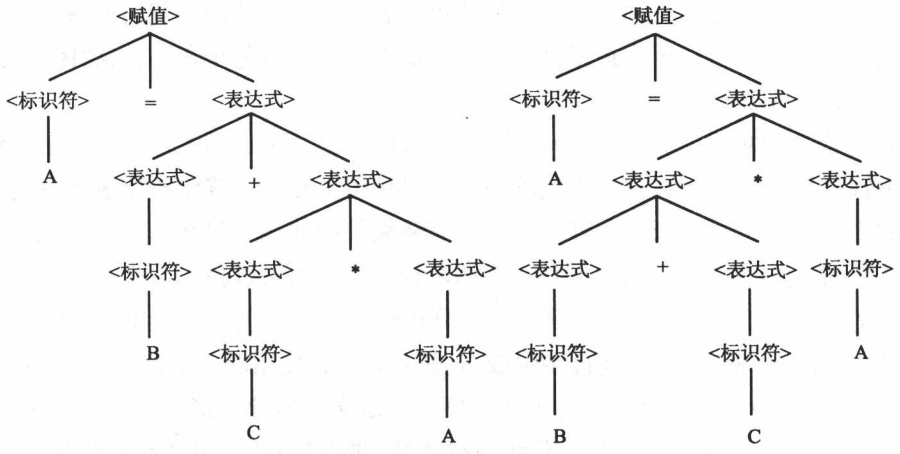


图 3-2　同一个语句 A=B+C*A 的两种不同的语法分析树

语言结构的语法歧义是一个难题，因为编译器通常将这些结构的语法形式作为语义的基础。具体而言，编译器要根据语句的语法分析树来决定为该语句生成什么样的代码。如果一种语言结构有一棵以上的语法分析树，就无法唯一地确定该结构的意义。这个问题将在下面小节的两个具体例子中讨论。

有时，文法还有其他一些特点来帮助确定文法是否有歧义[1]，其中包括：(1)文法可以用一种以上的最左推导生成一个句子；(2)文法可以用一种以上的最右推导生成一个句子。

有些语法分析算法可以基于有歧义的文法。当这样的语法分析程序碰到有歧义的语言结构时，它使用语言设计者提供的非文法信息来构造正确的语法分析树。在很多情况下，有歧义的文法可以重写为无歧义的文法，并且仍然生成需要的语言。

3.3.1.8 运算符优先级

当表达式有两个不同的运算符时，例如 x+y*z，一个显然的语义问题是两个运算符的求值顺序(在上面的表达式中是先加再乘，还是先乘再加)。要回答这个语义问题，可以给运算符设置不同的优先级。例如，如果语言设计者给*设置了比+更高的优先级，就会先做乘法，而不管两个运算符在表达式中的出现顺序如何。

如前所述，文法可以描述特定的语法结构，使语言结构的部分意义可以根据它的语法分析树来确定。特别是，如果算术表达式中的某个运算符在语法分析树的较低处生成(因此应该先求值)，就表示它比在树中更高处生成的运算符优先。例如在图 3-2 的第一棵语法分析树中，乘法运算符在树的较低处生成，表明它比表达式中的加法运算符优先。但是第二

1. 注意，在数学上，无法确定某一文法是否有歧义。

棵树表示的情况正好相反。因此可以看出，两棵语法分析树表示了相矛盾的优先级信息。

请注意，虽然例 3.2 的文法没有歧义，但是其运算符的优先级不是通常的顺序。在这个文法中，具有多个运算符的句子的语法分析树，不管涉及到哪些具体的运算符，表达式中最右边的运算符总是在语法分析树的最低点，表达式中左边的运算符则逐渐上移。例如在表达式 A+B*C 中，*在树的最低处，表示它最先计算。但是在表达式 A*B+C 中，+在最低处，表示它最先计算。

前面讨论的简单表达式的文法可以写成无歧义的，并为+和*运算符指定一致的优先级，而不管运算符在表达式中的出现顺序。可以使用单独的非终结符来表示优先级不同的运算符的操作数，指定正确的运算顺序。这需要额外的非终结符和一些新规则。具体方法是：不是用<表达式>来表示+和*运算符的两个操作数，而可以使用三个非终结符来表示操作数，让文法强制不同的运算符出现在语法分析树的不同高度上。如果<表达式>是表达式的根符号，可以让<表达式>只能直接生成+运算符，并且使用新的非终结符<项>作为+的右操作数，强制+出现在语法分析树的顶端。然后，定义<项>来生成*运算符，用<项>作为左操作数，用一个新的非终结符<因子>作为右操作数。现在*将总是出现在语法分析树的低处，因为在每一个推导中它都比+距离起始符号更远。例 3.4 就是这样一种文法。

例 3.4 表达式的无歧义文法

```
<赋值> → <标识符> = <表达式>
<标识符> → A | B | C
<表达式> → <表达式> + <项>
          | <项>
<项> → <项> * <因子>
       | <因子>
<因子> → ( <表达式>)
         | <标识符>
```

例 3.4 中的文法与例 3.2 和例 3.3 中的文法生成了相同的语言，但它是无歧义的，指定了乘法和加法运算符通常的优先级。以下对句子 A=B+C*A 的推导使用了例 3.4 中的语法：

```
<赋值> => <标识符> = <表达式>
       => A = <表达式>
       => A = <表达式> + <项>
       => A = <项> + <项>
       => A = <因子> + <项>
       => A = <标识符> + <项>
       => A = B + <项>
       => A = B + <项> * <因子>
       => A = B + <因子> * <因子>
       => A = B + <标识符> * <因子>
       => A = B + C * <因子>
       => A = B + C * <标识符>
       => A = B + C * A
```

使用例 3.4 的文法得到该句子唯一的语法分析树，如图 3-3 所示。

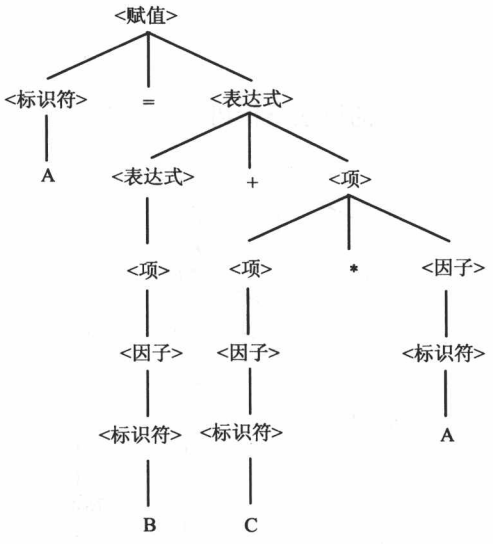


图 3-3　使用无歧义的文法为 A = B + C * A 生成唯一的语法分析树

语法分析树和推导间的联系十分密切：两者中的任何一个都很容易根据另一个来构造。每一个使用无歧义文法的推导都有唯一的语法分析树，但这棵树可以由不同的推导来表示。例如，句子 A=B+C*A 的以下推导与前面同一句子的推导不同。这是一个最右推导，而前面的推导是最左推导。但是这两种推导都用相同的语法分析树表示。

```
<赋值> => <标识符> = <表达式>
       => <标识符> = <表达式> + <项>
       => <标识符> = <表达式> + <项> * <因子>
       => <标识符> = <表达式> + <项> * <标识符>
       => <标识符> = <表达式> + <项> * A
       => <标识符> = <表达式> + <因子> * A
       => <标识符> = <表达式> + <标识符> * A
       => <标识符> = <表达式> + C * A
       => <标识符> = <项> + C * A
       => <标识符> = <因子> + C * A
       => <标识符> = <标识符> + C * A
       => <标识符> = B + C * A
       => A = B + C * A
```

3.3.1.9　运算符的结合性

当表达式包含两个优先级相同的运算符时(例如*和/通常具有相同的优先级)，例如 A / B*C，需要一条语义规则来指定哪一个应该优先[2]。这条规则称为结合性。

2. 出现两个相同运算符的表达式也有同样的问题，例如 A / B / C。

与优先级的情况一样，表达式的文法可以正确地蕴含运算符的结合性。考虑以下赋值语句的例子：

```
A = B + C + A
```

用例 3.4 的文法定义这一句子的语法分析树，如图 3-4 所示。

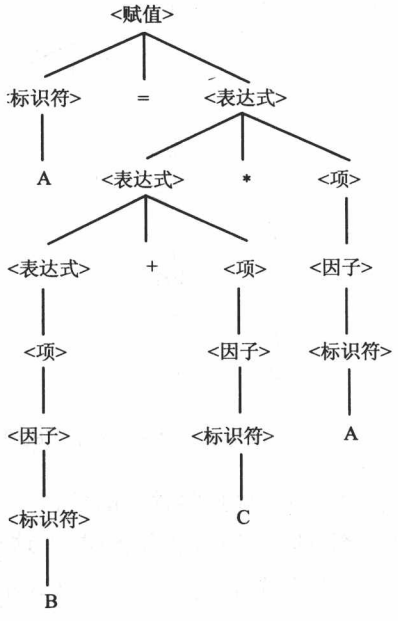


图 3-4 A=B+C+A 的语法分析树说明了加法的结合性

图 3-4 的语法分析树显示，左边的加法运算符低于右边的加法运算符。如果加法是左结合，这就是正确的运算顺序。在大多数情况下，计算机中加法的结合性无关紧要。在数学中，加法是结合的，意味着求值的左结合顺序和右结合顺序是一样的，即(A+B)+C=A+(B+C)。然而计算机中的浮点加法不一定是结合的。例如，假设浮点值要保留 7 位数字的精度，考虑将 11 个数相加的问题，其中一个数是 10^7，其余 10 个数是 1。如果小的数(1)一个一个地加到大数上，对这个数没有影响，因为小的数出现在大数的第 8 位。但是，如果小的数先加在一起，再加到大数上，结果用 7 位数字精度表示就是 $1.000001*10^7$。不管是在数学中还是在计算机中，减法和除法都不是结合的。因此正确的结合性对包含减法或除法的表达式是必不可少的。

当文法规则的 LHS 还出现在 RHS 的开头，这样的规则称为左递归。左递归表示左结合性。例如，在例 3.4 的文法规则中，左递归使加法和乘法都是左结合的。可惜左递归不允许使用一些重要的语法分析算法，当使用这样的算法时，必须修改文法，去除左递归，进而使文法不能准确地表示某些运算符是左结合的。幸运的是，左结合性可以由编译器强制实现，即使文法没有规定这一点。

在大多数提供了指数运算符的语言中，指数运算符是右结合的，可以用右递归来表示

右结合性。如果文法规则的 LHS 出现在 RHS 的右端，它就是右递归的，例如规则

```
<因子> → <表达式> ** <因子>
       | <表达式>
<表达式> → (<表达式>)
         | <标识符>
```

可以用于描述右结合的指数运算符。

3.3.1.10　if-then-else 的无歧义文法

在 Ada 语言中，if-then-else 语句的 BNF 规则如下：

```
<if 语句> → if <逻辑表达式> then <语句>
          | if <逻辑表达式> then <语句> else <语句>
```

如果还有<语句>→<if 语句>，这个文法就是有歧义的。说明这种歧义的最简单句型是

```
If <逻辑表达式> then if <逻辑表达式> then <语句> else <语句>
```

图 3-5 中的两棵语法分析树展示了这个句型的歧义。

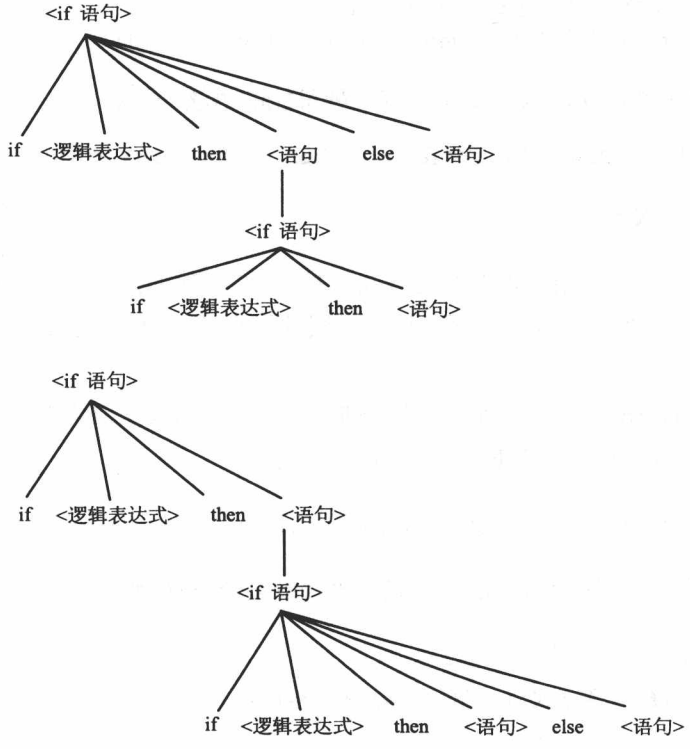


图 3-5　同一个句型的两棵不同的语法分析树

考虑如下语法结构示例：

```
if done == true
```

```
then if denom == 0
  then quotient = 0;
  else quotient = num / denom;
```

问题是，如果用图 3-5 顶部的语法分析树作为翻译的基础，那么 else 子句在 done 非真时执行，这很可能不是这一语言结构的作者的本意。第 8 章将研究与这种 else 结合问题有关的实际问题。

现在来设计描述这个 if 语句的无歧义文法。在许多语言中，if 结构的规则是：如果有 else 子句，它就与前面最近的一个没有匹配的 then 相匹配。因此在 then 和匹配的 else 中间不可能存在没有 else 的 if 语句。因此对于这种情况，必须区别匹配的和不匹配的 if 语句，没有 else 的 if 语句就是不匹配的，所有其他语句则是匹配的。前述文法的问题在于它认为所有语句都有相同的语法重要性，也就是说所有语句都是匹配的。

为了表示不同类型的语句，必须使用不同的抽象(或非终结符)。基于这一想法的无歧义文法如下所示：

```
<语句>→<匹配的> | <不匹配的>
<匹配的>→if <逻辑表达式> then <匹配的> else <匹配的>
        |任何非 if 语句
<不匹配的>→ if <逻辑表达式> then <语句>
         |if <逻辑表达式> then <匹配的> else <不匹配的>
```

对于下面的句型，使用这一文法只有一种可能的语法分析树。

```
if <逻辑表达式> then if <逻辑表达式> then <语句> else <语句>
```

3.3.2 扩展的 BNF

由于 BNF 略有不便，因此人们对它有几种扩展。大多数的扩展版本都称为扩展的 BNF，简称 EBNF，即使它们并不完全一样。这些扩展并没有增强 BNF 的描述能力，只是增加了它的可读性和可写性。

在不同版本的 EBNF 中，通常包含三种扩展。第一种是表示 RHS 的可选部分，用方括号括起来。例如，C 语言的 if-else 语句可以描述为：

```
<if 语句>→if (<表达式>) <语句> [else <语句>]
```

如果不用方括号，这个语句的语法描述就需要以下两条规则：

```
<if 语句>→if (<表达式>) <语句>
         | if (<表达式>) <语句> else <语句>
```

第二种扩展是在 RHS 中使用花括号，表示括起来的部分可以重复无数次或整体省去。这一扩展使列表可以用一条规则来构建，而不必使用递归的两条规则。例如，由逗号分隔的标识符列表可以用以下规则描述：

```
<标识符列表>→<标识符>{, <标识符>}
```

这替代了隐式迭代形式的递归，花括号括起来的部分可以迭代任意次。

第三种常用扩展可处理多个选项。当必须从一组元素中选出一个时，选项可以放在圆括号中，并用 OR 运算符“|”分隔开。例如：

```
<项>→<项>(* | / | %)<因子>
```

在 BNF 中，<项>的描述需要下面三条规则：

```
<项>→<项>*<因子>
     |<项> / <因子>
     |<项>%<因子>
```

EBNF 扩展中的方括号、花括号和圆括号都是元符号，即它们是标记工具，而不是所描述的语法实体中的终结符。如果这些元符号在所描述的语言中也是终结符，则终结符实例可以加下划线或使用引号。例 3.5 描述了如何在 EBNF 文法中使用方括号和多个选项。

例 3.5　表达式文法的 BNF 和 EBNF 版本

BNF：

```
<表达式>→<表达式>+<项>
          |<表达式>-<项>
          |<项>
<项>→<项>*<因子>
          |<项> / <因子>
          |<因子>
<因子>→<表达式>**<因子>
          <表达式>
<表达式>→(<表达式>)
          |<标识符>
```

EBNF：

```
<表达式>→<项>{(+ | -)<项>)<项>}
<因子>→<因子>{(* | / )<因子>}
<因子>→<表达式>{**<表达式>}
<表达式>→(<表达式>)
          | <标识符>
```

BNF 规则

```
<表达式>→<表达式>+<项>
```

明确表示(实际上是强制)+运算符是左结合的。但是，在 EBNF 版本中，

```
<表达式>→<项>{+<项>}
```

并没有表示出结合的方向。基于表达式的 EBNF 文法，语法分析器将语法分析过程设计为强制实现正确的结合性，从而解决这一问题。这将在第 4 章中深入讨论。

有些 EBNF 版本允许在右花括号上附加一个数字上标，表示括起来的部分可以重复的

次数上限。另外，有些版本使用加号(+)上标表示重复一次或多次重复。例如：

<复合语句>→ **begin** <语句> {<语句>} **end**

和

<复合语句>→**begin** {<语句>}$^{+}$ **end**

是等价的。

近年来出现了 BNF 和 EBNF 的一些变体，包括：

- 使用冒号来代替箭头，并把 RHS 放在下一行。
- 可选的 RHS 不是用竖线分隔，而是简单地放在不同的行上。
- 不使用方括号来表示可选的内容，而是使用下标 opt。例如

```
构造函数声明→简单名称 (形式参数列表 opt)
```

- 不是在圆括号括起来的元素列表中使用“|”符号表示选项，而是使用词“one of”。例如

```
赋值运算符→one of = *= /= %= += -= <<= >>= &= ^= |=
```

EBNF 有个标准，即 ISO/IEC 14977:1996(1996)，但很少用。该标准使用等号(=)代替规则中的箭头，用分号结束每一个 RHS，要求所有终结符使用引号，它还指定了其他一些标记规则。

3.3.3 文法与识别器

如本章前面所述，对于给定的语言，生成器和识别器之间有紧密的关系。实际上，给定一个上下文无关文法，由文法生成的语言的识别器可以用算法构造。人们已经开发了一些软件系统来完成这一构造任务。这样的系统可以为一种新语言快速地构建编译器的语法分析部分，因此非常有用。最早的这类语法分析器的构造系统之一称为 yacc[3](Johnson，1975)。现在有许多这样的系统可用。

3.4 属性文法

与上下文无关文法相比，属性文法可以更详细地描述程序设计语言的结构，是上下文无关文法的扩展。这种扩展可以方便地描述某些语言规则，例如类型兼容性。在正式定义属性文法的形式时，必须先弄清静态语义的概念。

3. 术语 yacc 是 yet another compiler compiler 的缩写，表示另一种编译器的编译器。

历史注释

属性文法的应用范围很广，可完整地描述程序设计语言的语法和静态语义(Watt, 1979)；正式地定义可以输入编译器生成系统的语言(Farrow，1982)；作为一些面向语法的编辑系统的基础(Teitelbaum 和 Reps，1981；Fischer 等，1984)；另外，属性文法还用于自然语言的处理系统(Correa，1992)。

3.4.1　静态语义

有些程序设计语言的结构特点很难用 BNF 描述，有些根本无法描述。类型兼容性规则就是用 BNF 很难表示的语法规则的一个例子。例如在 Java 中，浮点值不能赋值给整型变量，而反过来却是可以的。虽然这个限制可以用 BNF 表示，但是需要额外的非终结符和规则。如果 Java 所有的类型规则都用 BNF 来表示，文法将变得太大而无法使用，因为文法规模的大小决定了语法分析器的大小。

作为用 BNF 无法表示的语法规则的一个例子，考虑所有变量必须在引用前先声明这一常见规则。人们已经证明，这条规则无法用 BNF 来表示。

上述问题示例的这类语言规则称为静态语义规则。语言的静态语义与程序执行过程中程序的意义只有间接的关系，但它与正确的程序形式(语法而不是语义)有关。语言的许多静态语义规则规定了类型限制。之所以称为静态语义，是因为检查这些规定所需的分析可以在编译时进行。

由于用 BNF 描述静态语义有问题，人们发明了各种功能更强大的机制来描述静态语义。其中一种机制就是由 Knuth 提出的属性文法(Knuth，1968a)，它既可以描述程序的语法，也可以描述程序的静态语义。

属性文法是描述和检查程序静态语义规则的正式方法。虽然在编译器设计中并不总是正式使用它们，但在每个编译器中都至少非正式地使用了属性文法的基本概念(见 Aho 等，1986)。

动态语义，即表达式、语句和程序单元的意义，将在 3.5 节中介绍。

3.4.2　基本概念

属性文法是添加了属性、属性计算函数和谓词函数的上下文无关文法。属性与文法符号(终结符和非终结符)相关，从可以赋值的意义上说，它与变量类似。属性计算函数有时称为语义函数，与文法规则相关，用于说明如何计算属性值。谓词函数规定了语言的静态语义规则，与文法规则相关。

在正式定义属性文法，并给出例子后，这些概念会更加清楚。

3.4.3　属性文法定义

属性文法是具有以下附加特性的文法：

- 每个文法符号 X 都有一个属性集 A(X)与之相关联。集合 A(X)由两个不相交的集合 S(X)和 I(X)构成，分别称为综合属性和继承属性。综合属性用于沿语法分析树向上传递语义信息，继承属性用于沿语法分析树向下及横向传递语义信息。
- 每条文法规则有一个语义函数集和一个可能为空的谓词函数集与之相关联，它们作用在文法规则的符号属性上。对于规则 $X_0 \rightarrow X_1 \cdots X_n$，$X_0$ 的综合属性用形式为 $S(X_0)=f(A(X_1),\cdots,A(X_n))$的语义函数来计算，因此语法分析树结点上的综合属性值只取决于该结点中子结点的属性值。在上述规则中，符号 $X_j(1 \leqslant j \leqslant n)$的继承属性用形式为 $I(X_j)=f(A(X_0),\cdots,A(X_n))$的语义函数来计算，因此语法分析树结点上的继承属性值取决于该结点的父结点和兄弟结点的属性值。注意，为了避免出现循环定义，继承属性通常限制为形式为 $I(X_j)=f(A(X_0), \cdots, A(X_{(j-1)}))$的函数。这一形式避免继承属性取决于自身或取决于语法分析树右边结点的属性。
- 谓词函数的形式是作用在属性集$\{A(X_0),\cdots,A(X_n)\}$和一个字面属性值集合上的布尔表达式。属性文法唯一允许的推导条件是，每一个非终结符所关联的每一个谓词都为真。谓词函数值为假，意味着违反了语言的语法或静态语义规则。

属性文法的语法分析树是基于 BNF 文法的语法分析树，并且每一结点都附有一个可能为空的属性值集合。如果语法分析树上的所有属性值都已计算，就称这棵树是完全属性的。尽管实际上并不一定这样做，但可以方便地认为，属性值是在编译器构建好完全无属性的语法分析树后再计算的。

3.4.4 本质属性

本质属性是叶子结点的综合属性，其值在语法分析树之外确定。例如，程序中变量实例的类型可来自于符号表，符号表用于存储变量的名称和类型，其内容是基于先前的声明语句而设置的。开始时，假设无属性的语法分析树已经构建好，并需要计算属性值，唯一有值的属性就是叶子结点的本质属性。在语法分析树中，有了本质属性值后，就可以用语义函数来计算其余的属性值了。

3.4.5 属性文法的例子

以下属性文法的片断是如何使用属性文法来描述静态语义的一个非常简单的例子，它描述了 Ada 过程中 end 处的名称必须与过程名匹配这一规则(这条规则无法用 BNF 定义)。<过程名>的串属性用<过程名>.串表示，是编译器在保留字 procedure 后找到的字符串。请注意，在属性文法中，当一个非终结符在一条语法规则内出现一次以上时，用带方括号的下标来区分。下标和方括号都不是所描述语言的一部分。

语法规则：<过程定义>→**procedure** <过程名>[1] <过程体> **end** <过程名>[2];

谓词：　<过程名>[1].string == <过程名>[2].string

在这个例子中，谓词规则声明：在子程序开头的<过程名>非终结符中，名字串属性必须与子程序 end 后的<过程名>非终结符的名字串属性相匹配。

下面考虑属性文法的一个较大的例子。这个例子展示了如何用属性文法来检查简单赋值语句的类型规则。这个赋值语句的语法和静态语义如下：变量名只有 A、B 和 C；赋值的右边可以是一个变量，或形为一个变量加另一个变量的表达式；变量可以是两种类型之一：整数或实数；当赋值右边有两个变量时，它们不必具有相同的类型；当操作数的类型不同时，表达式的类型总是实数，当两者相同时，表达式的类型就是操作数的类型；赋值左边的类型必须与右边相匹配，因此右边操作数的类型可以是混合的，但只有当目标和右边计算结果值的类型相同时，赋值才合法。属性文法指明了这些静态语义规则。

示例的属性文法的语法部分是：

```
<赋值>→<变量> = <表达式>
<表达式>→<变量> + <变量>
        |<变量>
<变量>→A | B | C
```

示例属性文法中非终结符的属性如下所述：

- **实际类型**——与非终结符<变量>和<表达式>相关联的综合属性，用于存储变量或表达式的实际类型(整型或实型)。对于变量来说，实际类型是本质的。对于表达式来说，实际类型由子结点的实际类型或<表达式>非终结符的众子结点的实际类型来确定。
- **期望类型**——与非终结符<表达式>相关联的继承属性，用于存储表达式期望的类型(整型或实型)，由赋值语句左边变量的类型决定。

完整的属性文法见例 3.6。

例 3.6　简单赋值语句的属性文法

1. 语法规则：<赋值>→<变量> = <表达式>
   语义规则：<表达式>.期望类型←<变量>.实际类型
2. 语法规则：<表达式>→<变量>[2] + <变量>[3]
   语义规则：<表达式>.实际类型←
   If (<变量>[2].实际类型=整型) and
   (<变量>[3].实际类型=整型)
   then 整型
   else 实型
   end if
   谓词：　<表达式>.实际类型 == <表达式>.期望类型
3. 语法规则：<表达式>→<变量>
   语义规则：<表达式>.实际类型←<变量>.实际类型
   谓词：　<表达式>.实际类型 == <表达式>.期望类型
4. 语法规则：<变量>→A | B| C
   语义规则：<变量>.实际类型←查找(<变量>.串)

查找函数在符号表中查找给定的变量名，返回变量的类型。

由例 3.6 中文法生成的句子 A=A+B 的语法分析树如图 3-6 所示。如同在文法中一样，树中重复的结点标签后加上了带方括号的数字，以便无歧义地引用它们。

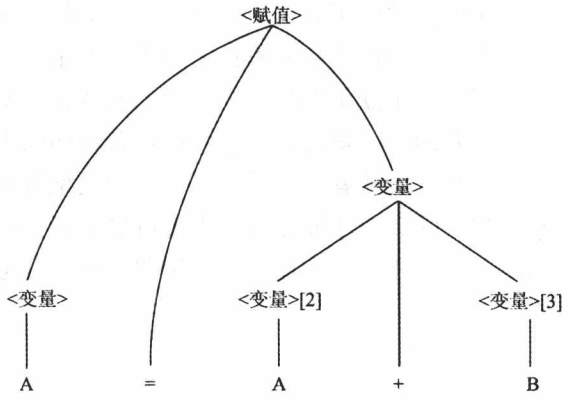


图 3-6 A=A+B 的语法分析树

3.4.6 计算属性值

现在，考虑计算语法分析树中属性值的过程，有时这一过程称为装饰语法分析树。如果所有属性都是继承属性，就可以完全按自顶向下的顺序由根到叶子来处理。或者，如果所有属性都是综合属性，就可以完全按自底向上的顺序由叶子到根来处理。由于文法中包含综合属性和继承属性，所以计算过程不能是单方向的。以下是属性的计算过程，是以可能的计算顺序进行的。

1. <变量>.实际类型←查找(A) (规则 4)
2. <表达式>.期望类型←<变量>.实际类型 (规则 1)
3. <变量>[2].实际类型←查找(A) (规则 4)
   <变量>[3].实际类型←查找(B) (规则 4)
4. <表达式>.实际类型←整型或实型 (规则 2)
5. <表达式>.期望类型 == <表达式>.实际类型是真或假 (规则 2)

图 3-7 中的树显示了图 3-6 中属性值的计算方向。实线表示语法分析树，虚线表示树中属性值的计算方向。

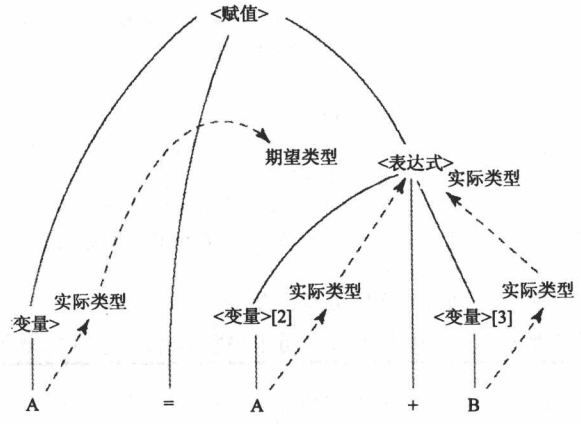


图 3-7 语法分析树中属性的计算方向

图 3-8 中的树显示了结点上最终的属性值。在这个例子中，A 定义为实型，B 定义为整型。

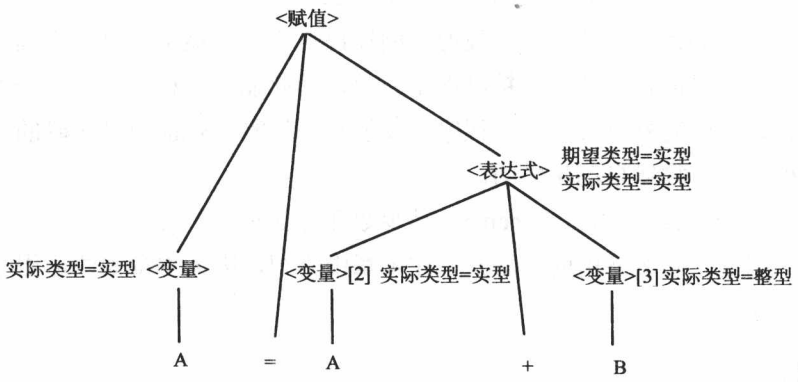


图 3-8　完全属性的语法分析树

确定属性文法一般情况下的属性求值顺序，是一个复杂的问题，需要构造一个依存关系图，来显示所有属性的依存关系。

3.4.7　评价

检查语言的静态语义规则是所有编译器的关键部分。即使编译器的实现者从未听说过属性文法，也需要使用属性文法的基本思想，为编译器设计对静态语义规则的检查。

对于真正的当代程序设计语言，用属性文法描述所有的语法和静态语义，主要困难之一是属性文法规模的大小和复杂程度。完整的程序设计语言需要大量属性和语义规则，使文法难以书写和阅读。而且，大型语法分析树的属性值的计算成本很高。另一方面，不那么正式的属性文法是编译器实现者的功能强大且常用的工具，他们对生成编译器的过程比对形式化理论更感兴趣。

3.5　描述程序的意义：动态语义

现在转到描述程序设计语言中表达式、语句和程序单元的动态语义(或意义)这一困难的任务上来。由于现有的标记方法强大而且自然，描述语法就相对简单。另一方面，目前还没有被普遍接受的其他标记方法或方案可用来描述动态语义。本节简要介绍一些已开发出的方法。本节后面使用术语“语义”时，指的是动态语义。

人们需要描述语义的方法学和标记方法，这有几个不同的原因。程序员在程序中有效使用语言的语句之前，显然需要准确地知道它们的作用。编译器实现者必须确切知道语言结构的意义，才能正确地设计编译器的实现。如果程序设计语言有精确的语义定义，那么这种语言的程序可能不必测试就可证明是正确的。而且，编译器可以产生与语言定义的行

为完全一致的程序，也就是说，可以验证它们的正确性。可以用一个工具，根据程序设计语言语法和语义的完整定义，自动生成语言的编译器。最后，语言设计者在设计语言的语义描述的过程中，也可以发现设计中的歧义和不一致。

软件开发者和编译器设计者，一般通过阅读语言手册的英文解释，来确定程序设计语言的语义。由于这样的解释通常不精确也不完整，因此显然这种方法并不令人满意。由于缺乏程序设计语言的完整语义定义，程序很少能够不经测试就证明是正确的，商业编译器从来就不是根据语言描述自动生成的。

第 15 章介绍的函数式语言 Scheme，是少数几种其定义包含了形式语义描述的程序设计语言之一。但是，它所使用的方法并不在本章中介绍，因为本章关注适用于命令式语言的语义描述方法。

3.5.1 操作语义

操作语义的思想是，通过指明语句或程序运行在机器上的效果来描述它们的意义。运行在机器上的效果表示为机器状态的一连串变化，机器状态是所存储的值的集合。因此，在计算机上执行一个编译好的程序，显然是一种操作语义的描述。大多数程序员在学习程序设计语言时，通常至少写过一次短小的测试程序，来确定某些程序设计语言结构的意义。这样的做法在本质上就是使用操作语义来确定语言结构的意义。

用这种方法来获得完整的形式化语义描述有一些问题。首先，机器语言执行的每一步和由此而导致机器状态的改变都太微小，而数量太庞大了。其次，实际计算机的存储规模太大、太复杂，通常有好几级内存设备，还通过网络与数量可观的其他计算机和内存设备连接起来。因此机器语言和真实的计算机都不能用来获得形式化的操作语义。人们为获得形式化的操作语义专门设计了中间级语言和理想计算机的解释器。

使用操作语义有不同的层次。在最高层次上，关注的是执行完整程序的最终结果，有时称为自然操作语义。在最低层次上，通过检查在程序执行时完整的状态变化序列，操作语义可用于确定程序的确切意义，有时称为结构操作语义。

3.5.1.1 基本过程

要创建语言的操作语义描述，第一步是设计一种适当的中间语言，这种语言的基本特征是清晰明确的。中间语言的每种结构都必须有明显的无歧义的意义。这种语言处于中间级，因为机器语言太低级，不易理解，另一种高级语言显然也不合适。如果这种语义描述用作自然操作语义，那么必须为这种中间语言构造一个虚拟机(解释器)。虚拟机可用于执行单条语句、代码段或整个程序。如果只需要单条语句的意义，就可以使用语义描述，而无需虚拟机。这种用途就是结构操作语义，可以可视化地检查中间代码。

操作语义的基本过程并不少见。实际上，这一概念经常在程序设计教科书和程序设计语言参考手册中用到。例如，C 语言中 for 结构的语义可以用更简单的语句来描述，如下：

C 语句	意义

For(表达式 1；表达式 2；表达式 3){
　　...
}
	表达式 1；
	loop: **if** 表达式 2 == 0 **goto** out
	　　...
	　　表达式 3；
	　　goto loop
	out:　...

阅读这一描述的人就是虚拟机，假设它能够正确“执行”定义中的指令，能够识别出“执行”的效果。

中间语言和用于形式化操作语义描述的相关虚拟机，通常都是高度抽象的。中间语言是为了虚拟机的方便，而不是供人阅读的。不过，为了更好地介绍中间语言，可以使用一种更便于人阅读的中间语言。例如考虑下列语句，它们比较适合描述典型程序设计语言中简单控制语句的语义。

```
ident = var
ident = ident + 1
ident = ident - 1
goto label
if var relop var goto label
```

在这些语句中，relop 是集合{=，<>，>，<，>=，<=}中的一个关系运算符，ident 是一个标识符，var 是一个标识符或常量。这些语句都很简单，因此易于理解和实现。

将这三个赋值语句稍微一般化，就可以描述更一般的算术表达式和赋值语句。新的语句是：

```
ident = var bin_op var
ident = un_op var
```

其中 bin_op 是一个二元算术运算符，un_op 是一个一元运算符。当然，多个算术数据类型和自动类型转换会使这样的一般化更复杂。只需要再加上一些简单的指令，就可以描述数组、记录、指针和子程序的语义。

第 8 章将用这种中间语言描述多种控制语句的语义。

3.5.1.2　评价

形式化操作语义的第一个也是最重要的应用是描述 PL/I(Wegner，1972)的语义。那台特殊的抽象机器和 PL/I 的翻译规则一起命名为维也纳定义语言(Vienna Definition Language，VDL)，这一名称来自于 IBM 设计这种语言所在的城市。

操作语义为语言使用者和实现者提供了一种有效的描述语义的方法，只要这种描述能保持简单且非形式化。不过 PL/I 的 VDL 描述太复杂了，没有达到实用的目的。

操作语义是基于较低级程序设计语言的，而不是数学。一种程序设计语言的语句用较低级程序设计语言的语句来描述，这种方法会导致循环定义，即概念间接地被自身所定义。以下两节所介绍的文法更加形式化，因为它们是基于数学和逻辑的，而不是程序设计语言。

3.5.2　指称语义

指称语义是描述程序意义的最严格、最广为人知的正式方法，牢固地建立在递归函数理论的基础之上。对使用指称语义来描述程序设计语言语义的完整介绍，肯定又长又复杂。我们的目的是介绍指称语义的核心概念，以及一些与程序设计语言规范相关的简单例子。

为程序设计语言构建指称语义说明的过程，需要为每一个语言实体定义一个数学对象，和一个将语言实体的实例映射到该数学对象实例上的函数。由于对象是严格定义的，它们代表了相应实体的确切意义。这种思想基于这一事实：处理数学对象有严格的方法，但处理程序设计语言结构却没有。这种方法的难度在于创建对象和映射函数，之所以命名为“指称”，是因为数学对象指称了相应语法实体的意义。

在程序设计语言规范的指称语义中，映射函数与数学中的所有函数一样，具有定义域和值域，其定义域是函数参数的有效值集合，值域是参数所映射对象的集合。在指称语义中，定义域称为语法域，因为所映射的是语法结构；值域称为语义域。

指称语义与操作语义相关。在操作语义中，程序设计语言结构被译为更简单的程序设计语言结构，后者成为原语言结构意义的基础。在指称语义中，程序设计语言结构映射为数学对象，或者是集合，或者(更经常)是函数。然而，与操作语义不同，指称语义不能代表一步步的程序计算过程。

3.5.2.1　两个简单例子

下面用一个非常简单的语言结构——二进制数的字符串表示，来介绍指称语义的方法。可以用下面的文法规则来描述二进制数的语法：

```
<二进制数>→'0'
          |'1'
          |<二进制数> '0'
          |<二进制数> '1'
```

二进制数 110 的语法分析树如图 3-9 所示。请注意，在语法意义的数字两边加单引号，表示它们并非数学意义上的数字，这跟 ASCII 编码的数字与数学意义的数字之间的关系是类似的。当程序以字符串的形式读入一个数时，必须把它变成数学意义上的数，才能将它用作程序中的值。

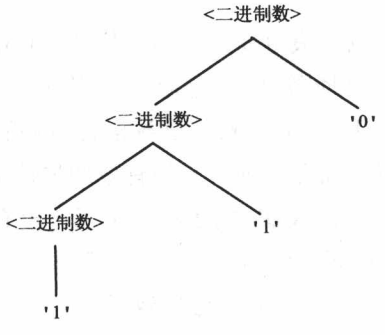


图 3-9　二进制数 110 的语法分析树

二进制的映射函数的语法域是所有二进制数的字符串表示的集合，语义域是非负的十进制数 N 的集合。

为了使用指称语义描述二进制数的意义，可将实际意义(十进制数)关联到每一条以单个终结符作为 RHS 的规则上。

在上例中，十进制数必须关联到前面两条文法规则。另两条文法规则从某种意义上说是计算规则，因为它们将一个终结符与一个非终结符组合在一起，该终结符可以关联一个对象，该非终结符可以表示某种语言结构。假设沿语法分析树向上进行求值，右边的非终结符已经附带了它的意义。然后，把非终结符作为 RHS 的语法规则需要一个函数，来计算 LHS 的意义，它表示完整的 RHS 的意义。

语义函数 M_{bin} 将前面文法规则所描述的语法对象映射为非负十进制数集合 N 中的对象。函数 M_{bin} 的定义如下：

```
Mbin('0')=0
Mbin('1')=1
Mbin(<二进制数>'0') = 2 * Mbin (<二进制数>)
Mbin(<二进制数>'1') = 2 * Mbin (<二进制数>)+1
```

意义，或指称的对象(在本例中是十进制数)，可以附加在上述语法分析树的结点上，得到如图 3-10 所示的语法分析树。这是语法指导的语义。语法实体映射为具有具体意义的数学对象。

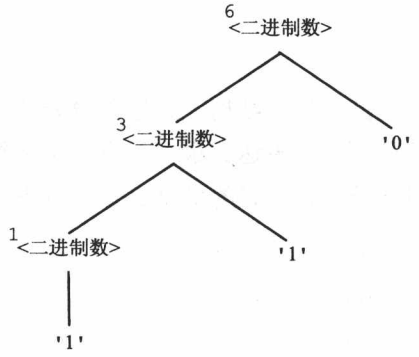


图 3-10　二进制数 110 的带指称对象的语法分析树

由于将来还会用到，所以现在给出描述语法的十进制字面值意义的一个类似例子。在这一例子中，语法域是十进制数的字符串表示的集合，语义域仍是集合 N。

```
<十进制数>→'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
      | <十进制数>('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')
```

这些语法规则的指称映射是：

```
Mdec('0')=0, Mdec('1')=1, Mdec('2')=2, ..., Mdec('9')=9
Mdec(<十进制数>'0') = 10 * Mdec(<十进制数>)
Mdec(<十进制数>'1') = 10 * Mdec(<十进制数>)+1
```

...

M_{dec}(<十进制数>'9') = 10 * M_{dec}(<十进制数>)+9

下一节将给出一些简单语言结构的指称语义描述。这里最重要的简化假设是，语言结构的语法和静态语义都是正确的。此外，假设只包含两种标量类型：整型和布尔型。

3.5.2.2 程序的状态

程序的指称语义可以用理想计算机的状态变化来定义。操作语义是用这种方法定义的，指称语义用几乎一样的方法来定义。不过在进一步简化之后，指称语义只用程序中所有变量的值来定义。因此，指称语义使用程序的状态来描述意义，而操作语义使用机器的状态来描述。操作语义和指称语义的关键区别在于，在操作语义中，状态变化是用某种程序设计语言编码的算法来定义的，而在指称语义中，状态变化用数学函数来定义。

令程序的状态 s 表示为如下有序对的集合：

$s=\{<i_1, v_1>, <i_2, v_2>, \cdots, <i_n, v_n>\}$

每个 i 是一个变量的名称，相关联的 v 是这些变量当前的值。任何一个 v 都有特殊值 undef，表示相关联的变量当前没有定义。令 VARMAP 是一个有两个参数的函数，参数是一个变量的名称和程序的状态，VARMAP(i_j, s)的值是 v_j(在状态 s 中与 i_j 成对的值)。程序和程序结构的大多数语义映射函数将状态映射到状态，这些状态变化用于定义程序和程序结构的意义。一些语言结构，例如表达式，映射到值，而不是状态。

3.5.2.3 表达式

表达式是大多数程序设计语言的基本结构。这里假设表达式没有副作用，而且只处理非常简单的表达式：运算符只有+和*，表达式最多只有一个运算符，操作数只有标量的整型变量和整型字面值，没有圆括号，表达式的值是整数。以下是这种表达式的 BNF 描述：

<表达式>→<十进制数> | <变量> | <二元表达式>
<二元表达式>→<左表达式><运算符><右表达式>
<左表达式>→<十进制数> | <变量>
<右表达式>→<十进制数> | <变量>
<运算符>→+ | *

在表达式中我们唯一考虑的错误是变量具有未定义的值。显然，其他错误也会出现，但大多数是与机器有关的。令 Z 表示整数集合，error 表示错误值，那么 Z∪{error}是表达式指称语义的语义域。

给定表达式 E 和状态 s，映射函数如下：为了区别数学函数定义和程序设计语言的赋值语句，使用符号△=来定义数学函数。下面定义中的蕴涵符=>将操作数的形式与相关联的 case(或 switch)结构相连接。点号标记用于表示一个结点的子结点，例如<二元表达式>.<左表达式>表示<二元表达式>的左子结点。

```
Me(<表达式>, s) △=case <表达式> of
                    <十进制数>=>Mdec(<十进制数>, s)
                    <变量>=>if VARMAP(<变量>, s) == undef
                               then error
                               else VARMAP(<变量>, s)
                    <二元表达式>=>
                     if(Me(<二元表达式>.<左表达式>, s) == undef OR
                        Me(<二元表达式>.<右表达式>, s) == undef)
                      Then error
                      Else if (<二元表达式>.<运算符> == '+')
                                then Me(<二元表达式>.<左表达式>, s) +
                                     Me(<二元表达式>.<右表达式>, s)
                                else Me(<二元表达式>-<左表达式>, s) *
                                     Me(<二元表达式>.<右表达式>, s)
```

3.5.2.4　赋值语句

赋值语句是一个求值的表达式，并且将目标变量设置为表达式的值。在这个例子中，意义函数将状态映射为状态。这个函数可以描述如下：

```
Ma(x=E, s) △ =if Me(E, s) == error
               then error
               else s'={<i1, v1'>, <i2, v2'>, ..., <in, vn'>), 其中
                      对于 j = 1, 2, ..., n
                       if ij == x
                          then vj' = Me(E, s)
                          else vj' = VARMAP(ij, s)
```

请注意，上面倒数第三行的比较 $i_j == x$ 是名称的比较，不是值的比较。

3.5.2.5　逻辑前测试循环

逻辑前测试循环的指称语义看起来简单，但实际上并非如此。为了便于介绍，假设有另外两个已有的映射函数 M_{sl} 和 M_b，前者将语句列表和状态映射为状态，后者将布尔表达式映射为布尔值(或 error)。这个函数是：

```
Ml(while B do L, s)  △  = if Mb(B, s) == undef
                            Then error
                            else if Mb(B, s) == false
                                    then s
                                    else if Msl(L, s) == error
                                            then error
                                            else Ml(while B do L, Msl(L, s))
```

这个循环的意义是，假设没有错误发生，循环中的语句在执行指定的次数之后，得到程序变量的值。实质上，这个循环已经从迭代转换为递归，其中递归控制是由其他递归状态映射函数以数学方式定义的。递归用严谨的数学方式比用迭代方式更容易描述。

现在有一个重要的发现是，这一定义与实际的程序循环一样，可能因为不会终止，而什么都没有计算。

3.5.2.6 评价

对象和函数，例如在前面语言结构中用到的，可以用来定义程序设计语言的其他语法实体。为指定语言定义了完整的系统后，就可以用这个系统来确定该语言完整程序的意义。这为以高度严谨的方式来看待程序设计提供了一个框架。

如前所述，指称语义可辅助语言设计。例如，指称语义描述复杂且困难的语句，可以提示设计者，该语句也可能会让语言使用者难以理解，可能需要换一种设计。

由于指称描述的复杂性，它们对语言使用者没什么用。另一方面，它们为精确描述语言提供了一种优异的方法。

虽然指称语义的使用通常归功于 Scott 和 Strachey(1971)，但是语言描述的一般性指称方法可以追溯到 19 世纪(Frege，1892)。

3.5.3 公理语义

公理语义之所以这样命名是因为它基于数理逻辑，它是本章介绍的说明语义的最抽象方法。公理语义不是直接说明程序的意义，而是说明关于程序可证明什么。如前所述，语义说明可能的用途之一就是证明程序的正确性。

在公理语义中，没有机器或程序状态的模型，没有程序执行时所发生的状态变化的模型。程序的意义基于程序变量和常量的关系，这种关系对程序的每次执行来说都是一样的。

公理语义有两个不同的应用：验证程序的正确性和说明程序语义。本节关注用公理语义描述来验证程序的正确性。

公理语义的定义是与一种证明程序正确性的方法同时提出的。如果能够构建程序正确性的证明，就表示程序可执行它所描述的计算过程。在证明中，程序的每一条语句前后都有一个逻辑表达式，说明程序变量的约束条件。这些条件不是抽象机器的完整状态(像操作语义中那样)，而是用于说明语句的意义。描述约束条件的标记方法—实际上是公理语义的语言——是谓词演算。虽然简单的布尔表达式通常适于表达约束条件，但在一些情况下它们并不适用。

当公理语义用于形式化地说明语句的意义时，意义是以语句对断言(关于语句所影响的数据)的影响来定义的。

3.5.3.1 断言

公理语义中使用的逻辑表达式称为谓词或断言。紧靠在程序语句前面的断言描述了程序在那一点上程序变量的约束条件，紧跟在语句后的断言描述了在语句执行后这些变量(可能还有其他变量)的新约束条件，这些断言分别称为语句的前置条件和后置条件。对于两条相邻的语句来说，第一条语句的后置条件就是第二条语句的前置条件。为已知程序给出公理描述或证明，需要程序中的每一条语句都具有前置条件和后置条件。

下面的小节从以下角度来研究断言：根据已知的后置条件来计算语句的前置条件，虽然反过来考虑也是可以的。假设所有变量都是整数类型的。作为一个简单的例子，考虑以下赋值语句和后置条件：

```
sum= 2 * x + 1 { sum > 1 }
```

前置条件断言和后置条件断言都放在花括号中，以便与程序的语句部分相区别。这一语句可能的一个前置条件是{x>10}。

在公理语义中，特定语句的意义由它的前置条件和后置条件来定义。实际上，这两个断言精确地表示了语句执行的效果。

下面的小节关注语句和程序的正确性证明，这是公理语义常见的应用。公理语义更一般性的思想是用逻辑表达式来精确陈述语句和程序的意义。验证程序的正确性是语言的公理语义描述的一个应用。

3.5.3.2　最弱前置条件

最弱前置条件是保证相关后置条件有效的限制最小的前置条件。例如，对于 3.5.3.1 节给出的语句和后置条件，{x>10}、{x>50}和{x>1000}都是有效的前置条件，其中最弱的是{x>0}。

如果最弱前置条件可以根据语言每一种语句类型的最一般的后置条件计算得到，那么计算这些前置条件的过程就给出语言语义的一种精确描述，而且可以构建这种语言程序的证明。开始证明程序的正确性时，其特点是以程序执行的结果作为程序最后一条语句的后置条件。后置条件和最后一条语句，用来计算最后一条语句的最弱前置条件，然后用这一前置条件作为倒数第二条语句的后置条件。这一过程一直进行，直到到达程序的开始。那时，第一条语句的前置条件表明了程序计算所需结果的条件。如果这些条件隐含在程序的输入说明中，就证明程序是正确的。

推理规则是一种基于其他断言取值来推断一条断言真实性的方法。推理规则的一般形式是：

$$\frac{S1, S2, \cdots, Sn}{S}$$

这个规则表示，如果 S1，S2，…，Sn 为真，那么可以推出 S 为真。推理规则的上面部分称为前提(antecedent)，下面部分称为结论(consequent)。

公理是假设为真的逻辑陈述，因此公理是没有前提的推理规则。

对于一些程序语句，根据语句和后置条件来计算最弱前置条件很简单，可以用公理来说明。然而在大多数情况下，最弱前置条件只能由推理规则来确定。

为了使用已知程序设计语言的公理语义，不管是用于正确性证明还是形式化语义说明，语言中每一种语句都必须有公理或推理规则。下面的小节给出赋值语句的一条公理，以及语句序列、选择语句和逻辑前测试循环语句的推理规则。请注意，这里假设算术表达式和布尔表达式都没有副作用。

3.5.3.3 赋值语句

赋值语句的前置条件和后置条件，共同精确地定义了赋值语句的意义。为了定义赋值语句的意义，必须有一种方法根据它的后置条件来计算它的前置条件。

令 x=E 是一条一般性的赋值语句，Q 是它的后置条件，那么它的前置条件 P 可以用以下公理来定义：

$$P=Q_{x\to E}$$

其意义是：P 的计算结果为 Q，并且所有 x 的实例都替换成 E。例如，赋值语句和后置条件如下：

```
a = b / 2 - 1 {a < 10}
```

计算最弱前置条件的方法是将后置条件{a<10}中的 a 替换为 b / 2-1，如下：

```
b / 2 - 1 < 10
b < 22
```

因此根据已知的赋值语句和后置条件，最弱前置条件是{b<22}。赋值公理只在没有副作用时才能保证正确。如果赋值语句改变了目标变量之外的其他变量，就会有副作用。

表示已知语句形式的公理语义所用的标记方法是：

{P}S{Q}

其中 P 是前置条件，Q 是后置条件，S 是语句形式。对于赋值语句，标记方法是：

$\{Q_{x\to E}\}x = E\{Q\}$

作为计算赋值语句的前置条件的另一个例子，考虑以下赋值语句和后置条件：

```
x = 2 * y - 3 {x > 25}
```

前置条件的计算如下：

```
2 * y - 3 > 25
y > 14
```

因此{y>14}是这一赋值语句和后置条件的最弱前置条件。

请注意，赋值语句左边的符号出现在等号的右边，不影响计算最弱前置条件的过程。例如

```
x = x + y - 3 {x > 10}
```

的最弱前置条件是：

```
x + y - 3 > 10
y > 13 - x
```

回想一下，提出公理语义是为了证明程序的正确性。根据这一点，这时很自然会问：

如何用赋值语句的公理来证明呢？答案是：已知的赋值语句、其前置条件和后置条件，可以看作一条逻辑陈述或定理。如果赋值公理在应用到后置条件和赋值语句上，得到了已知的前置条件，就证明了这条定理。例如下面的逻辑陈述

```
{x > 3} x = x - 3 {x > 0}
```

对以下语句和后置条件使用赋值公理

```
x = x - 3 {x > 0}
```

得到{x>3}，就是已知的前置条件。因此，我们就证明了例子中逻辑陈述的正确性。

接下来，考虑逻辑陈述

```
{x > 5} x = x - 3 {x > 0}
```

在这个例子中，已知的前置条件{x>5}与公理生成的断言不同。但是显然，{x>5}蕴涵了{x>3}。为了在证明中用到这一点，需要一条名为推论规则的推理规则。该推论规则的形式是

$$\frac{\{P\}\ S\{Q\},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}S\{Q'\}}$$

符号=>表示蕴涵，S 可以是任何程序语句。这条规则可以叙述为：如果逻辑陈述{P}S{Q}为真，断言 P'蕴涵了断言 P，断言 Q 蕴涵了断言 Q'，就可推理得到{P'}S{Q'}。换句话说，推论规则的内容是：后置条件总是可以弱化，前置条件总是可以强化。这在程序证明中非常有用，例如它可以完成上述逻辑陈述例子的证明。令 P 为{x>3}，Q 和 Q'为{x>0}，P'为{x>5}，就有

$$\frac{\{x>3\}x = x-3\{x>0\},\ (x>5) \Rightarrow \{x>3\},\ (x>0) \Rightarrow (x>0)}{\{x>5\}x = x-3\{x>0\}}$$

前提的第一项({x>3}x=x-3{x>0})是用赋值公理证明的，第二项和第三项显然正确，因此根据推论规则，结论为真。

3.5.3.4　语句序列

语句序列的最弱前置条件无法用公理描述，因为前置条件是基于序列中特定类别的语句。在这种情况下，前置条件只能用推理规则来描述。令 S1 和 S2 是相邻的程序语句，如果 S1 和 S2 具有如下前置条件和后置条件

{P1}S1{P2}
{P2}S2{P3}

那么这种两条语句序列的推理规则是

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\{P3\}}{\{P1\}S1,\ S2\ \{P3\}}$$

因此，对于本例，{P1}S1; S2{P3}描述了序列 S1; S2 的公理语义。推理规则表明，为了得到序列的前置条件，要计算第二条语句的前置条件。这一新的断言随后用作第一条语句的后置条件，可用于计算第一条语句的前置条件，也是整个序列的前置条件。如果 S1 和 S2 是赋值语句

```
x1=E1
```

和

```
x2=E2
```

那么就有

$\{P3_{x2\rightarrow E2}\}$ x2=E2{P3}
$\{(P3_{x2\rightarrow E2})_{x1\rightarrow E1}\}$x1=E1$\{P3_{x2\rightarrow E2}\}$

因此，序列 x1=E1; x2=E2 和后置条件 P3 的最弱前置条件是$\{(P3_{x2\rightarrow E2})_{x1\rightarrow E1}\}$。

例如，考虑以下序列和后置条件：

```
y = 3 * x + 1;
x = y + 3;
{x < 10}
```

第二条赋值语句的前置条件是

```
y < 7
```

它用作第一条语句的后置条件。现在可以计算第一条赋值语句的前置条件：

```
3 * x + 1 < 7
x < 2
```

因此，{x<2}是第一条语句也是这个两条语句序列的前置条件。

3.5.3.5 选择语句

接下来考虑选择语句的推理规则。选择语句的一般形式是：

if B **then** S1 **else** S2

只考虑包含 else 子句的选择语句。推理规则是

$$\frac{\{B \text{ and } P\}\ S1\ \{Q\},\ \{(\text{not } B) \text{ and } P\}\ S2\{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q\}}$$

这条规则表明，对于布尔控制表达式为真和为假两种情况，必须都要证明选择语句的正确性。横线上的第一条逻辑陈述表示 then 子句，第二条逻辑陈述表示 else 子句。根据推理规则，需要一个可同时用于 then 和 else 子句的前置条件 P。

考虑以下使用选择推理规则计算前置条件的例子。示例的选择语句是：

```
if x > 0 then
  y = y - 1
else
  y = y + 1
```

假设这个选择语句的后置条件 Q 是{y> 0 }，可以在 then 子句上应用赋值公理：

```
y = y - 1 {y > 0}
```

这会得到{y-1>0}或{y>1}，可用作 then 子句的前置条件的 P 部分。现在，在 else 子句上应用同一公理：

```
y = y + 1 {y > 0}
```

得到前置条件{y+1>0}或{y>-l}。由于{y>1}=>{y>-1}，所以推论规则允许用{y>1}作为整个选择语句的前置条件。

历史注释

人们做了大量的工作，研究使用指称语言描述来自动生成编译器的可能性(Jones，1980; Milos 等，1984; Bodwin 等，1982)。这些工作表明，该方法是可行的，但从来没有进展到生成有效的编译器这一步。

3.5.3.6　逻辑前测试循环

命令式程序设计语言的另一个基本结构是逻辑前测试循环，或 while 循环。计算 while 循环的最弱前置条件天生就比语句序列更难，因为迭代的次数并不总是能预先确定。在迭代次数已知的情况下，循环可以展开，像语句序列那样处理。

计算循环的最弱前置条件与证明有关所有正整数的定理类似，后者通常用归纳法来证明，同样的归纳方法可以用于某些循环。归纳法首要的步骤是找到归纳假设。在 while 循环的公理语义中，相应的步骤是找到称为循环不变式的断言，这对于找到最弱前置条件非常关键。

计算 while 循环的前置条件的推理规则是

$$\frac{\{I \text{ and } B\}\ S\ \{I\}}{\{I\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{end}\ \{I \text{ and (not } B)\}}$$

其中 I 是循环不变式。这看起来很简单，但实际上不是，其复杂性在于找到适当的循环不变式。

while 循环的公理语义描述写作：

{P} **while** B **do** S **end** {Q}

循环不变式必须满足一些要求才有用。首先，while 的最弱前置条件必须保证循环不变式为真。然后，循环不变式必须保证在循环终止时后置条件为真。这些约束条件可根据推理规则得到公理语义的描述。在执行循环时，循环不变式的真值不能受到循环控制的布尔表达式求值的影响，也不能受到循环体语句的影响，因此称为不变式。

while 循环的另一个复杂因素是循环终止问题。不会终止的循环不可能正确，实际上什么都没有计算。如果 Q 是循环退出后紧跟着的后置条件，那么循环的前置条件 P 保证在循环退出时满足 Q，并且保证循环会终止。

while 结构完整的公理语义描述需要以下所有条件为真，其中 I 是循环不变式：

P => I
{I and B} S {I}
(I and (not B)) => Q
循环会终止

如果循环计算了一系列数值，那么可能用一种方法来找到循环不变式，这种方法在使用数学归纳法证明关于数列的陈述时，用来确定归纳假设。对几种情况计算迭代次数和循环体前置条件的关系，希望出现一种可适用于一般情况的模式。将得到最弱前置条件的过程看作一个函数 wp 是有帮助的。一般来说，

wp(语句，后置条件)=前置条件

wp 函数通常称为谓词转换函数，因为它将一个谓词或断言作为参数，返回另一个谓词。

为了找到 I，从零开始对循环体的一些不同的迭代次数，用循环后置条件 Q 来计算前置条件。如果循环体包含单条赋值语句，可以用赋值语句公理来计算。考虑以下循环的例子：

```
while y <> x do y = y + 1 end {y = x}
```

请记住，这里的等号用于两种不同的用途。在断言中，它表示数学意义的相等，在断言之外，它表示赋值运算符。

对于零次迭代，最弱前置条件显然是

```
{y = x}
```

对于一次迭代，最弱前置条件是

```
wp(y = y + 1, {y = x}) = {y + 1 = x}, 或 {y = x - 1}
```

对于两次迭代，最弱前置条件是

```
wp(y = y + 1, {y = x - 1})={y + 1 = x - 1},或{y = x - 2}
```

对于三次迭代，最弱前置条件是

```
wp(y = y + 1, {y = x - 2})={y + 1 = x - 2}, 或{y = x - 3}
```

现在很显然，{y<x}满足一次或多次迭代的情况。将它与零次迭代的{y=x}组合起来，得到{y<=x}，这可以用作循环不变式。while 语句的前置条件可以通过循环不变式来确定，实际上，I 可以用作前置条件 P。

对于循环例子，必须保证我们的选择满足 I 的四个要求。首先，由于 P=I，所以 P=>I。第二个要求是，下列逻辑陈述必须为真

{I and B} S{I}

在本例中，有

```
{y <= x and y <> x} y = y + 1 {y <= x}
```

对下列赋值语句和后置条件应用赋值公理

```
y = y + 1 {y <= x}
```

得到{y+1<=x}，这与{y<x}等价，{y<=x and y<>x}蕴涵了这一条件，因此证明了前面的逻辑陈述。

接下来，必须要有

{I and (not B)} => Q

在本例中，有

```
{(y <= x) and not (y <> x)} => {y = x}
{(y <= x) and (y = x)} => {y = x}
{y = x} => {y = x}
```

因此这显然为真。接下来必须考虑循环终止。在本例中，问题是循环

```
{y <= x} while y <> x do y = y + 1 end {y = x}
```

是否会终止。x 和 y 都假设是整数变量，容易看出这个循环确实会终止。前置条件保证 y 初始时不大于 x，循环体每次迭代使 y 增 1，直到 y 等于 x。不管 y 初始时比 x 小多少，它最终会等于 x，因此循环将终止。由于选择的 I 满足所有四个要求，所以它是适当的循环不变式和循环前置条件。

前面用于计算循环不变式的过程，并不总能得到最弱前置条件的断言(尽管在例子中得到了)。

以下循环语句是用数学归纳法中的方法找到循环不变式的另一个例子：

```
while s > 1 do s = s / 2 end {s = 1}
```

与前面一样，我们使用赋值公理来尝试找到这个循环的循环不变式和前置条件。对于零次迭代，最弱前置条件是{s=l}。对于一次迭代，最弱前置条件是

```
wp(s = s / 2, {s = 1}) = {s / 2 = 1},或{s = 2}
```

对于二次迭代，最弱前置条件是

```
wp(s = s / 2, {s = 2}) = {s / 2 = 2}, 或{s = 4}
```

对于三次迭代，最弱前置条件是

```
wp(s = s / 2, {s = 4}) = {s / 2 = 4}, 或{s = 8}
```

从这些迭代中可以清楚地看到，不变式是

```
{s 是 2 的非负次幂}
```

再一次，计算得到的 I 可用作 P，I 满足那四个要求。与前面找到循环前置条件的例子不同，这个条件显然不是最弱前置条件。考虑使用前置条件{s>1}，很容易证明逻辑陈述

```
{s > 1} while s > 1 do s = s / 2 end {s = 1}
```

这个前置条件比前面计算出来的条件宽松得多。循环和前置条件对 s 的任意正值都满足，而不仅仅是上述过程给出的 2 的幂。根据推论规则，采用一个比最弱前置条件更强的前置条件，不会使证明无效。

找到循环不变式并不总是很容易，理解这些不变式的本质是有帮助的。首先，循环不变式是循环后置条件的弱化，同时也是循环的前置条件，因此它必须足够弱，以便在循环开始执行前就已满足，但是当与循环退出条件组合在一起时，它必须足够强，从而保证后置条件为真。

由于证明循环终止有难度，这个要求经常被忽略。如果可以表明循环会终止，就称循环的公理语义描述具有完全正确性，如果除了终止性以外的其他条件可以满足，就称具有部分正确性。

在更复杂的循环中找到适当的循环不变式，即使只要求部分正确性，也需要很多技巧。由于计算 while 循环的前置条件取决于找到循环不变式，因此用公理语义证明有 while 循环的程序的正确性会很难。

3.5.3.7 程序证明

本节给出两个简单程序的证明。证明正确性的第一个例子是一个非常短的程序，它用三条赋值语句交换两个变量的值。

```
{x = A AND y = B}
t = x;
x = y;
y = t;
{x = B AND y = A}
```

由于程序的语句序列中只包含赋值语句，可以用赋值公理和语句序列的推理规则来证明它的正确性。第一步是在整个程序的最后一条语句和后置条件上应用赋值公理，得到前置条件

```
{x = B AND t = A}
```

接下来，使用新的前置条件作为中间语句的后置条件，计算它的前置条件为

```
{y = B AND t = A}
```

接下来，使用新的断言作为第一条语句的后置条件，应用赋值公理，得到

```
{y = B AND x = A}
```

除了 AND 运算符的操作数顺序之外，这与程序的前置条件相同。由于 AND 是一个对称运算符，因此证明就完成了。

下面的例子是证明计算阶乘函数的伪代码程序的正确性。

```
{n >= 0}
count = n;
fact = 1;
while count <> 0 do
    fact = fact * count;
    count = count - 1;
end
{fact = n!}
```

前面介绍的找到循环不变式的方法，对这个例子中的循环不起作用。这里需要一些技巧，简单研究一下代码会有帮助。该循环计算阶乘的顺序是先算最后的乘法，即先做 (n-1)*n(假设 n 大于 1)。因此部分的不变式是

```
fact = (count + 1) * (count + 2) * ... * (n - 1) * n
```

但是还必须确保 count 总是非负的，这可以通过在上述断言上添加这一条件来实现，得到

```
I = (fact = (count + 1) * ... * n) AND (count >= 0)
```

接下来，必须确定 I 满足不变式的要求。再次将 I 作为 P 来使用，因此 P 显然蕴涵 I。下一个问题是

{I and B} S {I}

I and B 是

```
((fact = (count + 1) * ... * n) AND (count >= 0)) AND
   (count <> 0)
```

它可简化为

```
(fact = (count + 1) * ... * n) AND (count > 0)
```

在本例中，必须使用后置条件的不变式，来计算循环体的前置条件。对于

```
{P} count = count - 1 {I}
```

计算 P 为

```
{(fact = count * (count + 1) * ... * n) AND
    (count >= 1)}
```

将它用作循环体中第一条赋值语句的后置条件

```
{P} fact = fact * count {(fact = count * (count + 1)
                          * ... * n) AND (count >= 1)}
```

其中 P 是

```
{(fact = (count + 1) * ... * n) AND (count >= 1)}
```

显然 I and B 蕴涵这个 P，因此根据推论规则，

```
{I AND B} S {I}
```

为真。最后，对 I 的最终测试是

```
I AND (NOT B) => Q
```

对于本例，这是

```
((fact = (count + 1) * ... * n) AND (count >= 0)) AND
   (count = 0)) => fact = n!
```

这显然为真，因为当 count= 0 时，第一部分恰好是阶乘的定义。因此我们选择的 I 满足循环不变式的要求。现在能用从 while 语句中得到的 P(与 I 相同)作为程序第二条赋值语句的后置条件

```
{P} fact = 1 {(fact = (count + 1) * ... * n) AND
    (count >= 0)}
```

这可得到 P

```
(1 = (count + 1) * ... * n) AND (count >= 0))
```

将它用作代码中第一条赋值语句的后置条件

```
{P} count = n {(1 = (count + 1) * ... * n) AND
    (count >= 0))}
```

生成 P

```
{(n + 1) * ... * n = 1) AND (n >= 0)}
```

AND 运算符的左操作数为真(因为 1=1)，右操作数就是整个代码段的前置条件{n>= 0 }。因此，程序被证明是正确的。

3.5.3.8　评价

如前所述，为了使用公理语义的方法来定义完整程序设计语言的语义，语言中的每一个语句类型都必须有公理或推理规则。为程序设计语言的一些语句定义公理或推理规则，已被证明是项困难的任务。这个问题的一个显然的解决方案是，在设计语言时头脑中想着公理语义的方法，使语言只包含可写出公理或推理规则的语句。遗憾的是，这样的语言必须剔除一些有用、强大的部分。

公理语义是研究程序正确性证明的有力工具，它提供了一个优秀的框架，在程序构建过程中和构建之后对程序进行推理。不过对于语言使用者或编译器实现者来说，它在描述程序设计语言意义方面的用处非常有限。

小　结

巴科斯-诺尔范式和上下文无关文法是等价的元语言，很适合于描述程序设计语言的语法。它们不仅是简练的描述工具，而且与它们的生成动作相关联的语法分析树可给出隐含的语法结构的图形化表示。此外，它们很自然地与所生成语言的识别器相关，这就易于构建这些语言的编译器的语法分析部分。

属性文法是一种能够描述语言的语法和静态语义的描述形式，是上下文无关文法的一种扩展。属性文法由文法、属性集、属性计算函数集和谓词集构成，它们一起描述了静态语义规则。

本章简要介绍了三种语义描述方法：操作语义、指称语义和公理语义。操作语义是用语言结构对理想机器的作用来描述其意义的方法。在指称语义中，用数学对象来表示语言结构的意义，语言实体通过递归函数转换为数学对象。公理语义基于形式逻辑，是作为证明程序正确性的工具来设计的。

文献注释

用上下文无关文法和 BNF 描述语法在文献(Cleaveland 和 Uzgalis，1976)中有详尽的介绍。

对公理语义的研究由 Floyd(1967)开始，由 Hoare(1969)进一步发展。Pascal 语言大部分的语义就是 Hoare 和 Wirth 使用这种方法描述的(Hoare 和 Wirth，1973)。他们未完成的部分涉及函数的副作用和 goto 语句，这是最难描述的部分。

在程序开发中，使用前置条件和后置条件的技术是 Dijkstra 描述并提倡的(Dijkstra，1976)，(Griles，1981)也详细介绍了这种技术。

(Gordon，1979)和(Stoy，1977)对指称语义有不错的介绍。本章对所有三种语言描述方法的介绍可以在(Marcotty 等，1976)中找到。(Pagan，1981)也是本章很多资料的优秀参考书。本章中指称语义函数的形式与(Meyer，1990)中的相似。

复 习 题

1. 定义语法和语义。

2. 语言描述是为什么人服务的？
3. 描述一般的语言生成器的作用。
4. 描述一般的语言识别器的作用。
5. 句子和句型的区别是什么？
6. 定义一条左递归文法规则。
7. 大多数 EBNF 的三个常用扩展是什么？
8. 辨别静态和动态语义的差异。
9. 在属性文法中，谓词的作用是什么？
10. 综合属性和继承属性的区别是什么？
11. 对于给定的属性文法的树，属性求值的顺序是如何确定的？
12. 属性文法的主要用途是什么？
13. 请说明描述程序设计语言语义的方法学和标记方法的主要用途。
14. 为什么在操作语义中机器语言不能用来定义语句？
15. 描述使用操作语义的两个层次。
16. 在指称语义中，语法域和语义域是什么？
17. 对于指称语义，程序状态中存储了哪些内容？
18. 最广为人知的语义方法是什么？
19. 为了构建语言的指称语义描述，必须为每一语言实体定义哪两样东西？
20. 推理规则中的哪一部分是前提？
21. 谓词转换函数是什么？
22. 对循环结构来说，部分正确性是什么意思？
23. 公理语义基于数学的哪一个分支？
24. 指称语义基于数学的哪一个分支？
25. 用软件纯解释器来描述操作语义有什么问题？
26. 请说明在公理语义中，给定语句的前置条件和后置条件的意义是什么。
27. 描述用公理语义来证明已知程序正确性的方法。
28. 描述指称语义的基本概念。
29. 操作语义和指称语义根本性的不同是什么？

习　题

1. 语言描述的两个数学模型是生成和识别。描述它们如何定义程序设计语言的语法。
2. 为下列语句写出 EBNF 描述：
   a. Java 语言类定义的头语句
   b. Java 语言的方法调用语句
   c. C 语言的 **`switch`** 语句
   d. C 语言的 **`union`** 定义
   e. C 语言的 **`float`** 字面值

3. 重写例 3.4 中的 BNF，使+的优先级高于*，并且强制+为右结合。
4. 重写例 3.4 中的 BNF，加上 Java 语言的++和--一元运算符。
5. 写出 Java 布尔表达式的 BNF 描述，应包含三个运算符 & &、‖和!，以及关系表达式。
6. 使用例 3.2 的文法，为以下语句给出语法分析树和最左推导：
   a. `A=A*(B+(B+(C*A)))`
   b. `B=C*(A*C+B)`
   c. `A=A*(B+(C))`
7. 使用例 3.4 的文法，为以下语句给出语法分析树和最左推导：
   a. `A=(A+B)*C`
   b. `A=B+C+A`
   c. `A=A*(B+C).`
   d. `A=B*(C*(A+B))`
8. 证明以下文法有歧义：

   <S>→<A>

   <A>→<A>+<A> | <id>

   <id>→ a | b | C
9. 修改例 3.4 的文法，添加比+和*的优先级都高的一元减法运算符。
10. 用英语描述以下文法定义的语言：

    <S>→<A><B><C>

    <A>→a<A> | a

    <B>→b<B> | b

    <C>→c<C> | c
11. 考虑以下文法：

    <S>→<A> a <B> b

    <A>→<A> b | b

    <B>→a <B> | a

    以下语句子中哪些属于该文法生成的语言？

    a. baab

    b. bbbab

    c. bbaaaaa

    d. bbaab
12. 考虑以下文法：

    <S>→a <S> c <B> | <A> | b

    <A>→c <A> | c

    <B>→d | <A>

    以下语句中哪些属于该文法生成的语言？

    a. abcd

    b. acccbd

c. acccbcc

d. acd

e. accc

13. 写出由字符串构成的语言的文法，该字符串有 n 个字母 a，后跟相同数目的字母 b，其中 n>0。例如，字符串 ab、aaaabbbb 和 aaaaaaaabbbbbbbb 属于该语言，但 a、abb、ba 和 aaabb 不属于。
14. 为第 13 题中的文法推导出的句子 aabb 和 aaaabbbb 画出语法分析树。
15. 将例 3.1 中的 BNF 转换为 EBNF。
16. 将例 3.3 中的 BNF 转换为 EBNF。
17. 将以下 EBNF 转换为 BNF：

    S→A{bA}

    A→a[b]A
18. 本质属性与非本质综合属性的区别是什么？
19. 写出属性文法，其基础是 3.4.5 节的例 3.6 中的 BNF，但语言规则如下：在表达式中，数据类型不能混合，但在赋值语句中，赋值运算符的两边不必是相同类型。
20. 写出属性文法，其基础是例 3.2 中的 BNF，类型规则与 3.4.5 节中赋值语句的例子相同。
21. 使用 3.5.1.1 节中的虚拟机器指令，给出下列语句的操作语义定义：

    a. Java 语言的 **do-while**

    b. Ada 语言的 **for**

    c. C++语言的 **if-then-else**

    d. C 语言的 **for**

    e. C 语言的 **switch**
22. 为下列语句写出指称语义映射函数：

    a. Ada 语言的 **for**

    b. Java 语言的 **do-while**

    c. Java 语言的布尔表达式

    d. Java 语言的 **for**

    e. C 语言的 **switch**
23. 计算下列赋值语句和后置条件的最弱前置条件：

    a. `a = 2 * (b - 1) - 1 {a > 0}`

    b. `b = (c + 10) / 3 {b > 6}`

    c. `a = a + 2 * b - 1 {a > 1}`

    d. `x = 2 * y + x - 1 {x > 11}`
24. 计算下列赋值语句序列和后置条件的最弱前置条件：

    a.
    ```
    a = 2 * b + 1;
    b = a - 3
    {b < 0}
    ```

    b.
    ```
    a = 3 * (2 * b + a);
    b = 2 * a - 1
    {b > 5}
    ```

25. 计算下列选择结构和后置条件的最弱前置条件：

a.
```
if (a == b)
    b = 2 * a + 1
else
    b = 2 * a;
{b > 1}
```

b.
```
if (x < y)
    x = x + 1
else
    x = 3 * x
{x < 0}
```

c.
```
if (x > y)
    y = 2 * x + 1
else
    y = 3 * x - 1;
{y > 3}
```

26. 请说明，为了证明形式为 while B do S end 的逻辑前测试循环结构的正确性，有哪四个要求。

27. 证明(n+1)* … *n=1

28. 证明以下程序是正确的：

```
{n > 0}
count = n;
sum = 0;
while count <> 0 do
  sum = sum + count;
  count = count - 1;
end
{sum = 1 + 2 + ... + n}
```

第4章

词法分析和语法分析

4.1 概述
4.2 词法分析
4.3 语法分析问题
4.4 递归下降的语法分析
4.5 自下而上的语法分析

要认真研究编译器设计，至少需要一学期的强化学习，包括为一种小型但实际的程序设计语言设计和实现编译器。这类课程的第一部分就是词法和语法分析。语法分析是编译器的核心，因为其他一些重要组成部分，包括语义分析器和中间代码生成器，都是由语法分析的动作所驱动的。

一些读者可能会觉得奇怪，为什么在一本关于程序设计语言的书中会有一章介绍编译器的某个部分。本书涉及词法和语法分析至少有两个原因：首先，语法分析直接基于第 3 章介绍的文法，因此作为文法的应用来讨论语法分析是很自然的；其次，在编译器设计以外的很多情况下，都需要词法分析器和语法分析器。许多应用程序，其中包括程序列表格式化器、计算程序复杂度的程序，以及必须分析并根据配置文件内容起作用的程序，都需要进行词法分析和语法分析。因此，词法分析和语法分析对软件开发人员来说是重要的内容，即使他们不需要写编译器。而且，一些计算机科学专业不再需要学生上编译器设计的课，学生就没有学过词法或语法分析。在上述情况下，可将本章纳入程序设计语言的课程中。对于有编译器设计课程的专业，可以跳过本章。

本章先用一个简单的例子介绍词法分析；然后介绍语法分析的一般问题，包括语法分析的两种主要方法和语法分析的复杂性；接着介绍自顶向下分析器的递归下降实现技术，包括一个递归下降分析器的某些部分和用分析器进行语法分析的过程；最后一节介绍自底向上的语法分析和 LR 语法分析算法，这一节包括一个小型 LR 语法分析表的例子，以及用 LR 语法分析过程对一个字符串进行分析。

4.1 概述

第 1 章介绍了实现程序设计语言的三种方法：编译、完全解释和混合实现。编译方法用一个称为编译器的程序，将高级程序设计语言所写的程序翻译成机器代码。编译通常用于实现用于大型应用程序的程序设计语言，这些应用程序经常是用 C++和 COBOL 等语言写的。完全解释系统不执行翻译，而是用软件解释器对程序的源代码进行解释。完全解释通常用于其执行效率不重要的小型系统，例如 HTML 文档中内嵌的、用诸如 JavaScript 语言所写的脚本。混合实现系统将高级语言所写的程序翻译为供解释的中间形式。现在这些系统比以前有更广泛的应用，主要原因是脚本语言的广受欢迎。以前混合系统的程序执行比编译系统要慢得多，不过近年来即时(Just-in-Time，JIT)编译器已经得到了广泛应用，特别是 Java 程序和为 Microsoft .NET 系统所写的程序，当方法第一次调用时，就使用 JIT 编译器将中间代码翻译为机器代码。从实际效果看，JIT 编译器将混合系统转变为一个延迟的编译系统。

刚才讨论的这三种实现方法都使用了词法和语法分析器。

语法分析几乎总是基于程序语法的形式化描述。最常用的语法描述机制是第 3 章介绍的上下文无关文法(或 BNF)。与一些非正式的语法描述相比，使用 BNF 至少有三个引人注目的优点：第一，程序语法的 BNF 描述对于使用者和软件系统来说简明清晰；第二，BNF 描

述可用作语法分析器的直接基础；第三，由于模块化特点，基于 BNF 的实现相对易于维护。

几乎所有的编译器都将分析语法的任务分成两个不同的部分，称为词法分析和语法分析，虽然这样的术语让人混淆。词法分析器处理小型的语言结构，例如名字和数值字面值。语法分析器处理大型语言结构，例如表达式、语句和程序单元。4.2 节介绍词法分析，4.3 节、4.4 节和 4.5 节介绍语法分析。

词法分析与语法分析分开有三个原因：

1. **简单性**——词法分析技术不如语法分析所需的技术复杂，因此分开后词法分析过程更简单，而且将词法分析的低层次细节从语法分析器中移除，可使语法分析器更小更简洁。
2. **效率**——虽然很值得对词法分析器进行优化，因为词法分析需要占用整个编译时间的一大部分，但是对语法分析器进行优化却没有什么成效。将两者分开有助于选择性的优化。
3. **可移植性**——由于词法分析器读入输入程序文件，并且通常包含输入的缓存，因此在一定程度上是平台相关的。然而语法分析器可以是平台无关的。将软件系统中机器相关的部分独立出来通常是较好的做法。

4.2　词法分析

词法分析器本质上是一个模式匹配器。模式匹配器试图在给定的字符串中找到与给定的字符模式相匹配的子串。模式匹配是计算科学中一个传统的部分，模式匹配最早应用于文本编辑器，例如在 UNIX 早期版本中引入的 ed 行编辑器。从那时开始，模式匹配进入到一些程序设计语言中，例如 Perl 和 JavaScript，在 Java、C++和 C#的标准类库中也有。

词法分析器是语法分析器的前端。从技术上说，词法分析是语法分析的一部分，词法分析器在程序结构的最低层次上进行语法分析。输入程序对于编译器来说是一个字符串，词法分析器将字符聚集成逻辑的分组，并根据分组的结构赋予内部编码。在第 3 章中，这些逻辑的分组称为词素(lexeme)，分组类别的内部编码称为标记(token)。将输入字符串与字符串模式进行匹配，就识别出词素。虽然标记通常用整数值来表示，但是为了提高词法分析器和语法分析器的可读性，它们通常以命名常量来引用。

考虑以下赋值语句的例子：

```
result = oldsum - value / 100;
```

这一语句的标记和词素如下：

标记	词素
标识符	result
赋值运算符	=
标识符	oldsum
减法运算符	-

```
标识符          value
除法运算符      /
整数字面值      100
分号            ;
```

词法分析器从给定输入串中提取词素，并生成相应的标记。在编译器的早期，词法分析器通常处理整个源程序文件，生成标记和词素的文件。不过现在大多数词法分析器只是子程序，它定位输入中的下一个词素，确定与它相关联的标记编码，并将它们返回给调用程序，即语法分析器。所以，对词法分析器的每次调用都返回一个词素及其标记。语法分析器看到的输入程序的唯一形式，就是词法分析器的输出，一次一个标记。

词法分析过程包括跳过注释和词素外的空白，因为它们与程序的意义无关。词法分析器还将用户自定义名字的词素插入到符号表，符号表在编译器的后面阶段要用到。最后，词法分析器检测标记中的语法错误，例如错误的浮点字面值，并向用户报告这些错误。

建立词法分析器有三种方法：

1. 用正则表达式相关的描述语言，写出语言标记模式的形式化描述[1]。这些描述用作自动生成词法分析器的软件工具的输入。有许多这样的工具可用，最古老的工具称为 lex，通常是 UNIX 系统的一部分。
2. 设计描述语言标记模式的状态转移图，并写出实现该图的程序。
3. 设计描述语言标记模式的状态转移图，并手工构建状态图的表驱动实现。

状态转移图简称为状态图，是一个有向图。状态图的结点标记了状态名称，弧标记了导致状态转移的输入字符。弧也可以包含状态转移时词法分析器应执行的动作。

用于词法分析器的状态图表示成一类数理机器，称为有限自动机。有限自动机可设计为识别正则语言的成员。正则文法是正则语言的生成设备。程序设计语言的标记是正则语言，词法分析器是有限自动机。

现在用一个状态图和实现它的代码来说明词法分析器的构建。状态图可以简单地包含状态和每一个标记模式的状态转移，但是这种方法会得到一个非常大而复杂的图，因为在状态图中，每一个结点对于所分析语言的字符集中的每一个字符都需要一个状态转移。因此我们考虑简化的方法。

假设需要一个只识别算术表达式的词法分析器，包括作为操作数的变量名和整数字面值。变量名由大写字母、小写字母和数字组成，但必须以字母开头，没有长度限制。首先注意，有 52 个不同的字符(大写或小写字母)可作为名字的开始，这需要从状态转移图的初始状态出发的 52 个状态转移。但是，词法分析器只对名字的确定感兴趣，不关心是哪一个特定名字，因此为所有 52 个字符定义一个字符类 LETTER，在名字的第一个字符处只使用一个状态转移。

另一个简化状态转移图的机会是整数字面值标记，有 10 个不同的字符可以作为整数字面值词素的开头，这需要从状态图的起始状态出发的 10 个状态转移。由于词法分析器不关

1. 正则表达式是现在许多程序设计语言中模式匹配工具的基础，不管是直接还是通过类库间接提供。

心特定的数字，因此如果为数字定义一个字符类 DIGIT，对这一字符类中的任一字符使用指向收集整数字面值状态的单个状态转移，就可以构建一个压缩的状态图。

因为名字可以包含数字，所以对于从名字第一个字符后面的结点出发的状态转移，可以用 LETTER 或 DIGIT 的单个状态转移来继续收集名字的字符。

接下来，定义词法分析器内部一些通用任务的功能子程序。首先，需要一个有多项职责的子程序，可以称它为 getChar。调用时，getChar 从输入程序中获取下一个输入字符，将它放在全局变量 nextChar 中。getChar 还必须确定输入字符所属的字符类，将它放到全局变量 charClass 中。词法分析器当前构建的词素可以用字符串或字符数组来实现，称为 lexeme。

在一个称为 addChar 的子程序中，实现把 nextChar 中的字符放到字符串数组 lexeme 中的过程。这个子程序必须被显式调用，因为程序包含一些不必放在 lexeme 中的字符，例如词素间的空白字符。在更实际的词法分析中，注释也不放在 lexeme 中。

当调用词法分析器时，如果输入的下一字符是下一个词素的起始字符会很方便。因此，用一个名为 getNonBlank 的函数来跳过空格。

最后，需要一个名为 lookup 的子程序来计算单个字符标记的标记编码。在本例中，是圆括号和算术运算符。标记编码是由编译器作者随意赋给标记的数。

图 4-1 的状态图描述了标记模式，包括状态图每一状态转移所需要的动作。

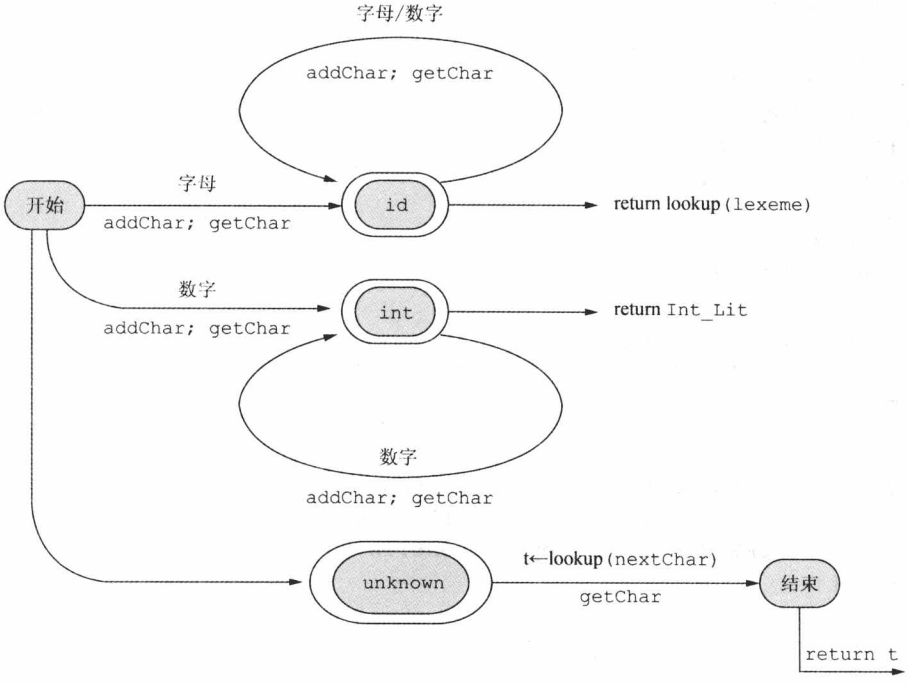


图 4-1　识别名字、圆括号和算术运算符的状态图

以下是图 4-1 的状态图所表示的词法分析器的 C 语言实现，包括用于测试的一个主驱动程序。

```
/* front.c - a lexical analyzer system for simple
             arithmetic expressions */

#include <stdio.h>
#include <ctype.h>

/* Global declarations */
/* Variables */
int charClass;
char lexeme [100];
char nextChar;
int lexLen;
int token;
int nextToken;
FILE *in_fp, *fopen();

/* Function declarations */
void addChar();
void getChar();
void getNonBlank();
int lex();

/* Character classes */
#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99

/* Token codes */
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26
/*****************************************************/
/* main driver */
main() {

/* Open the input data file and process its contents */
  if ((in_fp = fopen("front.in", "r")) == NULL)
    printf("ERROR - cannot open front.in \n");
  else {
    getChar();
    do {
      lex();
    } while (nextToken != EOF);
  }
}
```

```
/*********************************************************/
/* lookup - a function to lookup operators and parentheses
            and return the token */
int lookup(char ch) {
  switch (ch) {
    case '(':
      addChar();
      nextToken = LEFT_PAREN;
      break;

    case ')':
      addChar();
      nextToken = RIGHT_PAREN;
      break;

    case '+':
      addChar();
      nextToken = ADD_OP;
      break;

    case '-':
      addChar();
      nextToken = SUB_OP;
      break;

    case '*':
      addChar();
      nextToken = MULT_OP;
      break;

    case '/':
      addChar();
      nextToken = DIV_OP;
      break;

    default:
      addChar();
      nextToken = EOF;
      break;
  }
  return nextToken;
}

/*********************************************************/
/* addChar - a function to add nextChar to lexeme */
void addChar() {
  if (lexLen <= 98) {
    lexeme[lexLen++] = nextChar;
    lexeme[lexLen] = 0;
  }
  else
    printf("Error - lexeme is too long \n");
}
```

```
/*****************************************************/
/* getChar - a function to get the next character of
             input and determine its character class */
void getChar() {
  if((nextChar = getc(in_fp)) != EOF) {
    if(isalpha(nextChar))
      charClass = LETTER;
    else if (isdigit(nextChar))
          charClass = DIGIT;
        else charClass = UNKNOWN;
    }
    else
      charClass = EOF;
}

/*****************************************************/
/* getNonBlank - a function to call getChar until it
                 returns a non-whitespace character */
void getNonBlank() {
  while (isspace(nextChar))
    getChar();
}
/
*****************************************************/
/* lex - a simple lexical analyzer for arithmetic
        expressions */
int lex() {
  lexLen = 0;
  getNonBlank();
  switch (charClass) {

/* Parse identifiers */
    case LETTER:
      addChar();
      getChar();
      while (charClass == LETTER || charClass == DIGIT) {
        addChar();
        getChar();
      }
    nextToken = IDENT;
    break;

/* Parse integer literals */
    case DIGIT:
      addChar();
      getChar();
      while (charClass == DIGIT) {
        addChar();
        getChar();
      }
      nextToken = INT_LIT;
```

```
      break;

/* Parentheses and operators */
    case UNKNOWN:
      lookup(nextChar);
      getChar();
      break;

/* EOF */
    case EOF:
      nextToken = EOF;
      lexeme[0] = 'E';
      lexeme[1] = 'O';
      lexeme[2] = 'F';
      lexeme[3] = 0;
      break;
} /* End of switch */
 printf("Next token is: %d, Next lexeme is %s\n",
        nextToken, lexeme);
 return nextToken;
} /* End of function lex */
```

这段代码说明词法分析器相对简单。当然，我们省去了输入缓存和其他一些重要细节，而且我们处理的是一种非常小且简单的输入语言。

考虑以下表达式：

```
(sum + 47) / total
```

以下是 front.c 词法分析器用于该表达式的输出：

```
Next token is: 25 Next lexeme is (
Next token is: 11 Next lexeme is sum
Next token is: 21 Next lexeme is +
Next token is: 10 Next lexeme is 47
Next token is: 26 Next lexeme is )
Next token is: 24 Next lexeme is /
Next token is: 11 Next lexeme is total
Next token is: -1 Next lexeme is EOF
```

程序中的名字和保留字有类似的模式。虽然有可能构建一个状态图来识别程序设计语言中的每个保留字，但这会得到一个大得惊人的状态图。让词法分析器用相同的模式来识别名字和保留字，并查询一个保留字表，以确定哪些名字是保留字，这样简单且快速得多。用这种方法就是将保留字看作名字标记类中的例外。

词法分析器通常负责符号表的初始构建，符号表是编译器的名字数据库。符号表中的记录存储了用户自定义名字的信息，以及名字的属性。例如，如果是变量名，那么变量的类型是它存储在符号表中的一个属性。名字通常由词法分析器放到符号表中，名字的属性通常在词法分析器处理之后由编译器的某个部分放到符号表中。

4.3 语法分析问题

分析语法的过程称为语法分析(syntax analysis 或 parsing)。

本节介绍语法分析的一般问题，并介绍两类主要的语法分析算法：自顶向下的算法和自底向上的算法，还包括语法分析过程的复杂性。

4.3.1 语法分析概述

程序设计语言的语法分析器为给定的程序构建语法分析树。在某些情况下，语法分析树仅是隐式构建的，即可能只是生成语法分析树的遍历。但是对所有情况来说，在语法分析过程中都生成了构建语法分析树所需的信息。语法分析树和推导都包含了语言处理器所需的所有语法信息。

语法分析有两个不同的目的：第一，语法分析器必须检查输入程序，确定它在语法上是否正确，当发现错误时，分析器必须生成诊断信息并恢复编译。这里的恢复编译是指分析器必须返回正常的状态，并继续对输入程序进行分析。有了这一步骤，分析器就可以在对输入程序的一次分析过程中发现尽可能多的错误。如果做得不好，则错误恢复会引起更多的错误，至少是更多的错误信息。语法分析的第二个目的是对于语法正确的输入，生成完整的语法分析树，至少是跟踪整个语法分析树的结构。语法分析树(或它的跟踪)用作翻译的基础。

语法分析器根据构建语法分析树的方向来分类。语法分析树的两大类别是自顶向下的(从根开始向下直到叶子来构建分析树)和自底向上的(从叶子向上直到根来构建分析树)。

本章使用一个小型的标记约定来表示文法符号和字符串，使介绍不那么混乱。对于形式语言来说，它们是：

1. 终结符——字母表开头的小写字母(a, b, …)
2. 非终结符——字母表开头的大写字母(A, B, …)
3. 终结符或非终结符——字母表末尾的大写字母(W, X, Y, Z)
4. 终结符串——字母表末尾的小写字母(w, x, y, z)
5. 终结符和/或非终结符的混合串——小写希腊字母(α, β, δ, γ)

对于程序设计语言来说，终结符是语言的小型语法结构，称为词素。程序设计语言的非终结符通常是用尖括号括起来的有意义的名字或缩写，例如<while 语句>、<表达式>和<函数定义>。语言的句子(对于程序设计语言来说就是程序)是终结符串。混合串描述了文法规则的右手边(RHS)，在语法分析算法中使用。

4.3.2 自顶向下的语法分析器

自顶向下的语法分析器以前序方式跟踪或构建语法分析树。语法分析树的前序遍历从根开始，先访问结点再访问它的分支，特定结点的分支以从左到右的顺序访问，这与最左推导相对应。

自顶向下的语法分析器可以用推导描述如下：给定最左推导的一个句型，语法分析器的任务是找到最左推导的下一个句型。最左推导句型的一般形式是 xAα，根据标记约定，x 是终结符串，A 是非终结符串，α 是混合串。由于 x 只包含终结符，A 是句型的最左非终结符，因此必须扩展 A，以得到最左推导的下一个句型。确定下一句型就是选择以 A 为 LHS 的正确文法规则。例如，如果当前句型为

xAα

并且以 A 为 LHS 的规则是 A→bB，A→cBb 和 A→a，则自顶向下的语法分析器必须从这三条规则中选择，来获得下一句型，可以是 xbBα、xcBbα 或 xaα。这是自顶向下的语法分析器的分析决策问题。

不同的自顶向下的语法分析算法采用不同的信息来进行分析决策。最常用的自顶向下的语法分析器将下一个输入标记与当前句型中最左非终结符规则的 RHS 可生成的第一个符号相比较，从而选择正确的 RHS。任何一个以该标记为所生成串最左端的 RHS 都是正确的。因此，在句型 xAα 中，语法分析器将使用 A 所生成的第一个标记来决定应该用哪一条 A 规则来得到下一句型。在上面的例子中，A 规则的 3 个 RHS 都以不同的终结符开始，语法分析器很容易根据下一个输入标记来选择正确的 RHS，在这个例子中输入必须是 a、b 或 c。通常，选择正确的 RHS 没有这样简单，因为当前句型最左非终结符的一些 RHS 可能以非终结符开始。

最常用的自顶向下的语法分析算法都是密切相关的。递归下降语法分析器是直接基于语言语法 BNF 描述的语法分析器的编码版本。除递归下降之外的最常用方法，是使用语法分析表而不是编码，来实现 BNF 规则。这两种方法称为 LL 算法，功能是一样的，即它们用于所有上下文无关文法的同一子集。LL 中的第一个 L 表示对输入进行从左向右的扫描，第二个 L 表示生成最左推导。4.4 节介绍实现 LL 语法分析器的递归下降方法。

4.3.3　自底向上的语法分析器

自底向上的语法分析器从叶子开始向根方向进行语法分析树的构建，这一语法分析顺序对应于反向的最右推导。即推导的句型是以从后向前的顺序生成的。自底向上的语法分析器可以用推导描述如下：给定右句型 α[2]，语法分析器必须确定 α 中的哪一个子串是文法中规则的 RHS，该子串应归约为规则的 LHS，来生成最右推导的前一个句型。例如，自底向上的语法分析器的第一步是确定原始输入句子的哪一个子串是 RHS，且可归约为相应的 LHS，来得到最右推导的倒数第二个句型。找到正确的 RHS 来归约是一个复杂的过程，因为给定的右句型可能包含所分析语言的文法中的多个 RHS。正确的 RHS 称为句柄。

考虑以下文法和推导：

S→aAc
A→aA | b

2. 右句型是在最右推导中出现的句型。

S => aAc => aaAc => aabc

句子 aabc 的自底向上的语法分析器从该句子开始，必须找到其中的句柄。在本例中，这是一个简单的任务，因为串只包含一个 RHS——b。当语法分析器将 b 替换为它的 LHS——A 时，就得到了推导的倒数第二个句型 aaAc。如前所述，通常情况下，找到句柄要困难得多，因为一个句型可能包含多个不同的 RHS。

自底向上的语法分析器检查可能的句柄的一边或两边的符号，来找到指定右句型的句柄。可能的句柄右边的符号通常是还没有分析过的输入标记。

最常用的自底向上的语法分析算法都是 LR 算法，其中 L 表示对输入进行从左向右的扫描，R 表示生成最右推导。

4.3.4 语法分析的复杂度

可用于任意无歧义文法的语法分析算法都很复杂，且效率低下。实际上，这类算法的复杂度是 $O(n^3)$，意思是说算法所需的时间是所分析串长度的立方。需要如此相对大量的时间是因为，这些算法必须经常回退，并重新分析句子的某些部分。当语法分析器在分析过程中出错时，就需要重新分析。语法分析器的回退还需要将正在构建(或跟踪)的语法分析树部分拆散重建。$O(n^3)$的算法通常无法用于实用的过程，例如编译器的语法分析，因为它们太慢了。在这种情况下，计算机科学家经常要寻找虽然不那么通用、但是更快的算法，即以通用性换取高效率。在语法分析方面，已经找到只适用于所有可能文法集合的一个子集的更快算法。只要该子集包含描述程序设计语言的文法，这些算法就可接受(实际上，如第 3 章所述，所有的上下文无关文法都不足以描述大多数程序设计语言的所有语法)。

对于所有商用编译器的语法分析器，所用的算法复杂度是 O(n)，意味着它们所需的时间与被分析串的长度线性相关，这比 $O(n^3)$的算法要高效得多。

4.4 递归下降的语法分析

本节介绍自顶向下的语法分析器的实现过程——递归下降。

4.4.1 递归下降的语法分析过程

递归下降语法分析器之所以这样命名，是因为它由一些子程序组成，许多子程序都是递归的，并且按自顶向下的顺序生成语法分析树。这种递归是程序设计语言本质的体现，包括一些不同类别的嵌套结构。例如，语句经常嵌套于其他语句中，表达式中的圆括号必须正确地嵌套。这些结构的语法很自然地是用递归的文法规则描述的。

EBNF 对于递归下降语法分析器是很理想的。回顾第 3 章，主要的 EBNF 扩展是花括号(表示所包含的部分可以出现零次或多次)和方括号(表示所包含的部分可出现一次或不出现)。请注意，在这两种情况下，所包含的符号都是可选的。考虑以下两个例子：

<if 语句>→**if** <逻辑表达式><语句> [**else** <语句>]

<标识符列表>→标识符{，标识符}

在第一条规则中，if 语句的 else 子句是可选的。在第二条规则中，<标识符列表>是一个标识符，后面跟着逗号与标识符的零次或多次重复。

递归下降语法分析器对于文法中的每一个非终结符都有一个子程序，与特定非终结符相关联的子程序的职责如下：当给定一个输入串时，跟踪以该非终结符为根且叶子与输入串相匹配的语法分析树。实际上，递归下降的语法分析子程序是相关联的非终结符所生成语言(串的集合)的语法分析器。

考虑简单算术表达式的以下 EBNF 描述：

<表达式>→<项>{(+ | -)<项>}
<项>→<因子>{(* | /)<因子>}
<因子>→标识符|整数常量 | (<表达式>)

回顾第 3 章，算术表达式的 EBNF 文法，例如以上的文法，并不强制结合性规则。因此，当使用这样的文法作为编译器的基础时，必须注意确保通常由语法分析驱动的代码生成过程，能够生成与语言结合性规则一致的代码。在使用递归下降的语法分析时，这很容易做到。

在以下的递归下降函数 expr 中，词法分析器是 4.2 节实现的函数，它获取下一个词素，并将它的标记编码放入全局变量 nextToken 中。标记编码以命名常量来定义，与 4.2 节中的一样。

对于只有单个 RHS 的规则来说，递归下降的子程序相对简单。对于 RHS 中的每一个终结符，该终结符与 nextToken 相比较，如果不匹配，则有语法错误；如果匹配，则调用词法分析器，获取下一个输入标记；对于每一个非终结符，调用该非终结符的语法分析子程序。

对于上述示例文法的第一条规则，用 C 语言写的递归下降子程序是：

```
/* expr
   Parses strings in the language generated by the rule:
   <expr> -> <term> {(+ | -) <term>}
   */
void expr() {
  printf("Enter <expr>\n");

/* Parse the first term */
  term();

/* As long as the next token is + or -, get
   the next token and parse the next term */
 while (nextToken == ADD_OP || nextToken == SUB_OP) {
   lex();
   term();
 }
 printf("Exit <expr>\n");
} /* End of function expr */
```

请注意，expr 函数包含了跟踪输出语句，生成本节后面要展示的示例输出。

编写递归下降的语法分析子程序要遵循以下约定：每一子程序将下一个输入标记放入 nextToken。因此，只要开始一个语法分析函数，都保证 nextToken 包含语法分析过程还未用过的输入的最左标记编码。

expr 函数分析的语言部分由一个或多个项组成，由加法或减法运算符分隔开。这是由非终结符<表达式>生成的语言，因此它首先调用分析项的函数 term，然后只要找到 ADD_OP 标记或 SUB_OP 标记(通过调用 lex 来跳过)，就继续调用该函数。这个递归下降函数比大多数函数都简单，因为它的规则只有一个 RHS。而且，它不包含任何语法错误检测或恢复的代码，因为没有可检测的错误与文法规则相关联。

对于规则有一个以上 RHS 的非终结符，递归下降的语法分析子程序开始的代码要确定分析哪一个 RHS。在构建编译器时，检查每一个 RHS，来确定可作为所生成句子开头的终结符的集合。将这些集合与输入的下一标记相比较，语法分析器就可选择正确的 RHS。

<项>的语法分析子程序与<表达式>的类似：

```
/* term
   Parses strings in the language generated by the rule:
   <term> -> <factor> {(* | /) <factor>)
   */
void term() {
  printf("Enter <term>\n");

/* Parse the first factor */
  factor();

/* As long as the next token is * or /, get the
   next token and parse the next factor */
  while (nextToken == MULT_OP || nextToken == DIV_OP) {
    lex();
    factor();
  }
  printf("Exit <term>\n");
} /* End of function term */
```

在算术表达式文法中，<因子>非终结符的函数必须在两个 RHS 中选择，它还包括错误检测。在<因子>的函数中，检测到语法错误的反应只是调用 error 函数。在真正的语法分析器中，当检测到错误时，必须生成诊断信息，而且语法分析器必须从错误处恢复，使语法分析过程继续下去。

```
/* factor
   Parses strings in the language generated by the rule:
   <factor> -> id | int_constant | ( <expr )
   */
void factor() {
  printf("Enter <factor>\n");

/* Determine which RHS */
```

```
  if (nextToken == IDENT || nextToken == INT_LIT)

/* Get the next token */
    lex();

/* If the RHS is ( <expr>), call lex to pass over the
   left parenthesis, call expr, and check for the right
   parenthesis */
  else {
    if (nextToken == LEFT_PAREN) {
      lex();
      expr();
      if (nextToken == RIGHT_PAREN)
        lex();
      else
        error();
    } /* End of if (nextToken == ... */

/* It was not an id, an integer literal, or a left
   parenthesis */
    else
      error();
   } /* End of else */

   printf("Exit <factor>\n");;
} /* End of function factor */
```

以下是对示例表达式(sum+47)/total 的语法分析的跟踪，使用了语法分析函数 expr、term 和 factor，以及 4.2 节的函数 lex。请注意，语法分析器在开始时调用 lex 和起始符号的子程序，在本例中是 expr。

```
Next token is: 25 Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11 Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21 Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10 Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26 Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24 Next lexeme is /
Exit <factor>
```

```
Next token is: 11 Next lexeme is total
Enter <factor>
Next token is: -1 Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

语法分析器对于上述表达式所跟踪的语法分析树如图 4-2 所示。

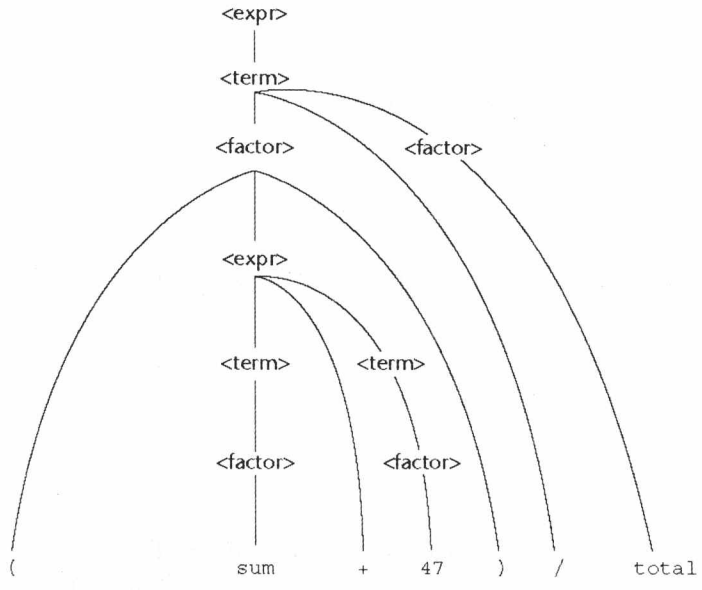


图 4-2 (sum+47)/total 的语法分析树

再来一个文法规则和语法分析函数的例子，应该有助于巩固读者对递归下降语法分析的理解。下面是 Java 中 if 语句的文法描述：

<if 语句>→**if**(<布尔表达式>)<语句>[**else** <语句>]

这一规则的递归下降子程序如下：

```
/* Function ifstmt
    Parses strings in the language generated by the rule:
    <ifstmt> -> if (<boolexpr>) <statement>
                  [else <statement>]
    */
void ifstmt() {
/* Be sure the first token is 'if' */
  if (nextToken != IF_CODE)
    error();
  else {
/* Call lex to get to the next token */
    lex();
/* Check for the left parenthesis */
```

```
    if (nextToken != LEFT_PAREN)
      error();
    else {
/* Call boolexpr to parse the Boolean expression */
      boolexpr();
/* Check for the right parenthesis */
      if (nextToken != RIGHT_PAREN)
        error();
      else {
/* Call statement to parse the then clause */
        statement();
/* If an else is next, parse the else clause */
        if (nextToken == ELSE_CODE) {
/* Call lex to get over the else */
          lex();
          statement();
        } /* end of if (nextToken == ELSE_CODE ... */
      } /* end of else of if (nextToken != RIGHT ... */
    } /* end of else of if (nextToken != LEFT ... */
  } /* end of else of if (nextToken != IF_CODE ... */
} /* end of ifstmt */
```

请注意，这个函数使用了本节并未给出的语句和布尔表达式的语法分析器函数。

这两个例子的目的是说明，如果有了程序设计语言的适当文法，很容易写出递归下降的语法分析器。下一小节讨论构建递归下降的语法分析器的文法特征。

4.4.2　LL 文法类

在选择使用递归下降方法作为编译器或其他程序分析工具的语法分析策略之前，必须考虑该方法在文法限制方面的局限性。本节讨论这些局限性，以及可能的解决方案。

对于 LL 语法分析器来说，导致灾难问题的一个简单的文法特征是左递归。例如以下规则：

A→A+B

A 的递归下降的语法分析器子程序会立即调用自己，分析 RHS 的第一个符号，而激活 A 的分析器子程序会立即再次调用自身，一次又一次，一直调用下去，显然没有止境(除非出现堆栈溢出)。

规则 A→A+B 中的左递归称为直接左递归，因为它出现在一条规则中。直接左递归可以通过以下过程从文法中去除：

对于每一个非终结符 A

1. 将 A 的规则集合到一起，得到 $A \rightarrow A\alpha_1 | \ldots | A\alpha_m | \beta_1 | \beta_2 | \ldots 1\ \beta_n$，其中 β 都不是以 A 开始的。
2. 将原始的 A 规则替换为

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

注意，ε 表示空串。以 ε 作为 RHS 的规则称为删除规则，因为在推导中使用该规则，实际上是将它的 LHS 从句型中删除。

考虑以下示例文法以及上述过程的应用：

E→E + T | T
T→T * F | F
F→(E) | id

对于 E 规则来说，有 α_1=+T 和 β=T，因此将 E 规则替换为

E→T E'
E，→+T E' | ε

对于 T 规则，有 α_1=* F 和 β=F，因此将 T 的规则替换为

T→ FT'
T'→*F T' | ε

由于在 F 规则中没有左递归，所以保持原样，因此替换后的完整文法是

E→T E'
E'→+T E' | ε
T→F T'
T'→*F T' | ε
F→(E) | id

这一文法保证与原始文法生成相同的语言，但不是左递归的。

与 4.4.1 节中用 EBNF 写的表达式文法一样，该文法没有表明运算符的左结合性。但是，基于这一文法可以相对容易地设计出代码生成方法，使加法和乘法运算符具有左结合性。

间接左递归具有与直接左递归同样的问题。例如，假设有

A→B a A
B→A b

这两条规则的递归下降的语法分析器会使 A 子程序立即调用 B 的子程序，B 的子程序立即调用 A 的子程序。因此，问题与直接左递归是一样的。左递归问题并不限于递归下降方法构建自顶向下的语法分析树，它是所有自顶向下的语法分析算法的难题。幸运的是，左递归对于自底向上的语法分析算法来说不是问题。

可以使用算法修改给定文法，去除间接左递归(Aho 等，2006)，但是这里并不涉及。为

程序设计语言编写文法时，通常可以避免包含直接或间接左递归。

左递归并不是妨碍自顶向下语法分析的唯一文法特征。另一个特征是，只使用当前句型中最左非终结符生成的第一个标记，语法分析器是否总能够基于输入的下一标记来选择正确的RHS。有一个对非左递归文法的简单测试可表明是否可以做到这一点，称为成对不相交测试(pairwise disjointness test)。该测试需要能够基于文法中给定非终结符的 RHS 计算一个集合，该集合称为 FIRST，定义如下：

FIRST(α) = {a | α=>* aβ}　(如果 α=>*ε，那么 ε 在 FIRST(α)中)

其中，=>*表示 0 或多个推导步骤。

对于任意混合串 α 计算 FIRST 的算法可以在(Aho 等，2006)中找到。这里，FIRST 通常可以通过检查文法来计算得出。

成对不相交测试是：

对于文法中每个有一个以上 RHS 的非终结符 A，对于每一对规则 A→αi 和 A→αj，FIRST(αi)∩FIRST(αj)= ϕ 必须为真。

(两个集合 FIRST(αi)和 FIRST(αj)的交集必须为空)。

换句话说，如果非终结符 A 有一个以上 RHS，那么每个 RHS 的推导所生成的第一个终结符必须是该 RHS 唯一的。考虑以下规则：

A→aB | bAb | Bb
B→cB | d

A 规则 RHS 的 FIRST 集合是{a}、{b}和{c，d}，显然是不相交的，因此这两条规则通过了成对不相交测试，如果用递归下降的语法分析器，就意味着分析非终结符 A 的子程序的代码，可以只根据非终结符生成的输入的第一个终结符(标记)，来选择要处理的 RHS。现在考虑规则

A→aB | BAb
B→aB | b

A 规则 RHS 的 FIRST 集合是{a}和{a，b}，显然是相交的，因此这两条规则不能通过成对不相交测试。如果用递归下降的语法分析器，A 的子程序不能根据下一个输入符号，来确定要分析的是哪一个 RHS，因为如果下一个符号是 a，它就可以是两个 RHS 的任何一个。如果一个或多个 RHS 以非终结符开头，问题显然会更复杂。

在许多情况下，不能通过成对不相交测试的文法可以修改，以通过测试。例如，考虑规则

<变量>→标识符 | 标识符[<表达式>]

这表明<变量>或者是标识符，或者是标识符后跟一个放在方括号内的表达式(下标)。

上述规则显然不能通过成对相交测试，因为两个 RHS 都以相同的终结符(标识符)开始。这个问题可以通过一个称为提取左因子的过程来缓解。

现在非正式地描述提取左因子。考虑<变量>的规则，两个 RHS 都以标识符开始，在两个 RHS 中，跟在标识符后面的部分是 ε(空串)和[<表达式>]。两条规则可以替换为

<变量>→标识符<新符号>
<新符号>→ε | [<表达式>]

不难发现，这两条规则与开始的两条规则生成相同的语言。但是，这两条规则可通过成对不相交测试。

如果要将文法用作递归下降的语法分析器的基础，就可以采用提取左因子的一种替代方法。使用一种 EBNF 扩展，就可以一种非常类似于提取左因子的方式解决这一问题。考虑前面<变量>的原始规则，可将下标放在方括号内，使它成为可选的，如下

<变量>→标识符 [[<表达式>]]

在这条规则中，外层的方括号表示内部可选的元符号，内层的方括号是所描述的程序设计语言的终结符。要点是，将两条规则替换为一条可生成相同语言的规则，并且通过了成对不相交测试。

提取左因子的形式化算法可以在(Aho 等，2006)中找到。提取左因子不能解决所有文法的成对不相交问题。在某些情况下，必须以另外的方式重写规则才能消除这一问题。

4.5 自下而上的语法分析

本节介绍自底向上的语法分析的一般过程，包括对 LR 语法分析算法的描述。

4.5.1 自下而上的语法分析器的分析问题

考虑以下算术表达式的文法：

E→E + T | T
T→T * F | F
F→(E) | id

注意，该文法与 4.4 节的示例文法生成相同的算术表达式，区别在于该文法是左递归的，这对于自下而上的语法分析器是可接受的。而且，自底向上的语法分析器的文法通常不含诸如用于 BNF 特定扩展的那些元符号。以下的最右推导展示了该文法。

E=>E +T
=>E + T * F
=>E + T * id

=>E+ F * id
=>E + id * id
=>T + id * id
=> F + id * id
=> id + id * id

在这个推导的每个句型中，下划线的部分都是 RHS，用相应的 LHS 重写可得到前一个句型。自下而上的语法分析过程，生成最右推导的逆向推导。因此在示例的推导中，自下而上的语法分析器以最后一个句型(输入句子)开始，生成句型的序列，直到只剩下起始符号——在该文法中是 E。在语法分析的每一步，自下而上的语法分析器的任务是找到句型中必须重写以得到下一个(前一个)句型的特定 RHS(句柄)。如前所述，正确的句型可能包含一个以上 RHS，例如正确的句型

E + T * id

包含三个 RHS：E+T、T 和 id，其中只有一个是句柄。例如，如果在该句型中选择 E+T 来重写，得到的句型就是 E*id，但是 E*id 不是给定文法中一个合法的右句型。

右句型的句柄是唯一的。自底向上的语法分析器的任务是找到由相关联文法生成的任意给定右句型的句柄。句柄形式化地定义如下：

定义：当且仅当 $S=>^*_{rm}\alpha Aw=>_{rm}\alpha\beta w$，$\beta$ 是右句型 $\gamma=\alpha\beta w$ 的句柄

在这个定义中，$=>_{rm}$ 表示一次最右推导，$=>^*_{rm}$ 表示零次或多次最右推导。虽然句柄的定义在数学上是精确的，但对如何找到给定右句型的句柄没什么帮助。下面给出与句柄相关的一些句型子串的定义，目的是给出一些与句柄有关的直观感受。

定义：当且仅当 $S=>^*\gamma=\alpha_1A\alpha_2=>^+\alpha_1\beta\alpha_2$，$\beta$ 是右句型 γ 的短语

在这个定义中，=>+表示一次或多次推导。

定义：当且仅当 $S=>^*\gamma=\alpha_1A\alpha_2=>\alpha_1\beta\alpha_2$，$\beta$ 是右句型 γ 的简单短语

仔细比较这两个定义，显然它们只在最后的推导说明处有不同。短语的定义使用一次或多次推导，而简单短语的定义只使用一次推导。

短语与简单短语的定义看起来可能与句柄的定义一样缺乏实用性，但事实并非如此。考虑短语与语法分析树的关系，在整个语法分析树中，以一个特定的内部结点为根的部分语法分析树中，短语是所有叶子的串。简单短语是以非终结符结点为根的一次推导的短语。对语法分析树来说，短语可以从一个非终结符经过一个或多个树的层次推导出来，而简单短语的推导只经过一个层次。考虑图 4-3 所示的语法分析树。

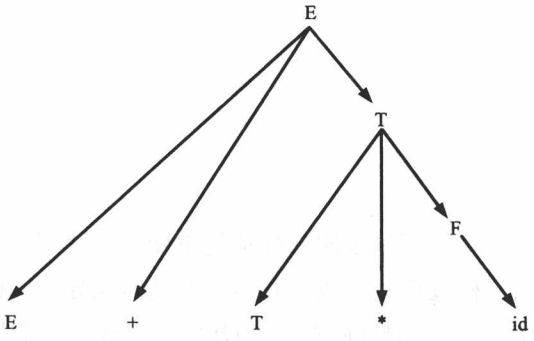


图 4-3 E+T * id 的语法分析树

图 4-3 中语法分析树的叶子组成句型 E+T*id。由于有三个内部结点，因此有三个短语。每一个内部结点是一棵子树的根，子树的叶子是短语。整棵语法分析树的根结点 E 生成完整的结果句型 E+T*id，这是一个短语。内部结点 T 生成叶子 T* id，这是另一个短语。最后，内部结点 F 生成 id，这也是一个短语。因此句型 E+T* id 的短语是 E+T*id、T*id 和 id。注意，短语不一定是底层文法的 RHS。

简单短语是短语的子集。在上面的例子中，唯一的简单短语是 id。简单短语总是文法的 RHS。

介绍短语和简单短语的原因是：任意最右句型的句柄都是它的最左简单短语。因此，假设我们有文法并且可画出一棵语法分析树，现在就有一种高度直观的方法来找到任意右句型的句柄。这种找到句柄的方法显然不适用于语法分析器(如果已经有了语法分析树，为什么还需要语法分析器呢?)其唯一目的是给读者一些直观感受，相对于语法分析树来说句柄是什么，这比根据句型来考虑句柄要容易一些。

现在可以用语法分析树来考虑自下而上的语法分析，虽然语法分析器的目的是生成语法分析树。给出整个句子的语法分析树，很容易找到句柄，它是句子中第一个可重写、以得到前一个句型的串。然后可以从语法分析树中剪去该句柄，重复这一过程直到语法分析树的根，就可构造出完整的最右推导。

4.5.2 移进-归约算法

自底向上的语法分析器经常称作移进-归约算法，因为移进和归约是最常用的两个操作。每个自底向上的语法分析器都有一个栈。与其他语法分析器一样，自底向上的语法分析器的输入是程序的标记流，输出是一系列文法规则。移进操作将下一个输入标记移到语法分析器的栈内，归约操作将语法分析器栈顶的 RHS(句柄)替换为相应的 LHS。每一个程序设计语言的语法分析器都是一个下推自动机(pushdown automaton，PDA)，因为 PDA 是上下文无关语言的识别器。不必熟悉 PDA 就可理解自底向上的语法分析器的工作方式，虽然熟悉 PDA 对理解有帮助。PDA 是一种非常简单的、从左向右扫描符号串的数学机器。PDA 之所以这样命名，是因为它用一个下推栈作为存储器。PDA 可用做上下文无关语言的

识别器。给定上下文无关语言字符集上的一个符号串，为识别而设计的 PDA 能够确定串是不是语言中的句子。在识别过程中，PDA 能够为句子生成构建语法分析树所需的信息。

用 PDA 来检查输入串，从左到右，一次一个符号。对输入的处理好像输入是存储在另一个栈中那样，因为 PDA 无法看到除输入的最左符号之外的更多东西。

请注意，递归下降的语法分析器也是一个 PDA。这时，栈就是运行时系统的栈，记录了子程序的调用(及其他信息)，子程序与文法的非终结符相对应。

4.5.3　LR 语法分析器

人们设计了许多不同的自底向上语法分析算法，大都是一种称为 LR 的过程的变体。LR 语法分析器使用一个相对小巧的程序和一张为特定程序设计语言建立的语法分析表。最初的 LR 算法是由 Donald Knuth 设计的(Knuth，1965)，这个算法有时也称为规范 LR(canonical LR)，在算法发表后的数年中并没有立即得到使用，因为生成所需的语法分析表需要大量的计算机时间和存储空间。后来，人们设计了规范 LR 建表过程的一些变体(DeRemer，1971；DeRemer 和 Pennello，1982)，这些变体具有两个特点：(1)它们生成语法分析表所需要的计算机资源比规范 LR 算法要少得多；(2)它们用于比规范 LR 算法规模更小的文法。

LR 语法分析器有三个优点：

1. 所有程序设计语言都可构建。
2. 在从左到右的扫描中，可尽早发现语法错误。
3. LR 类文法是 LL 语法分析器可分析的文法类的超集(例如，许多左递归文法都是 LR，但都不是 LL)。

LR 语法分析的唯一缺点是对于完整的程序设计语言，难以为给定的文法手工生成语法分析表。不过这不是一个严重的缺点，因为有好些程序能以文法为输入，来生成语法分析表，在本节稍后会介绍。

在出现 LR 语法分析算法之前，有一些语法分析算法能够通过查看句型中可能是句柄的子串的左右两边，找到右句型的句柄。Knuth 的看法是，可以高效地查看可能的句柄的左边，一直到语法分析栈的底部，从而确定它是否是句柄。语法分析栈中所有与语法分析过程相关的信息，都可以用一个状态来表示，可以存储在栈的顶部。换句话说，Knuth 发现不管输入串的长度、句型的长度或语法分析栈的深度是多少，语法分析过程所涉及的只有相对少数的不同情况。每一种情况都可以用一个状态来表示，存储在语法分析栈中，栈中每一个文法符号都会有一个状态符号。栈顶总是一个状态符号，表示到目前为止整个语法分析历史的相关信息。我们将带下标的大写 S 来表示语法分析器的状态。

图 4-4 显示了一个 LR 语法分析器的结构。LR 语法分析器的语法分析栈的内容具有以下形式：

$S_0X_1S_1X_2 \ldots X_mS_m$(栈顶)

其中 S 是状态符号，X 是文法符号。LR 语法分析器的格局是一对字符串(栈，输入)，具体形式是

$(S_0X_1S_1X_2S_2 \ldots X_mS_m，\ a_ia_{i+1} \ldots a_n\$)$

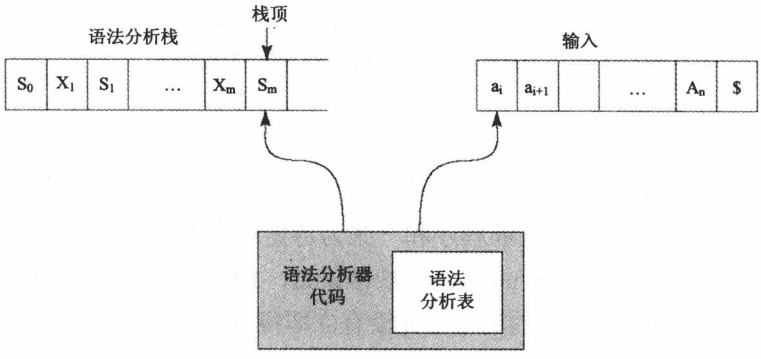


图 4-4　LR 语法分析器的结构

请注意，输入串在右端有一个美元符号，这个符号是在初始化语法分析器时放在那里的，用于正常终止语法分析器。使用这种语法分析器的格局，能够形式化地定义 LR 语法分析过程，这一过程是基于语法分析表的。

LR 语法分析表有两个部分，称为 ACTION 和 GOTO。表的 ACTION 部分说明语法分析器大部分的操作，它以状态符号作为行的标号，以文法终结符作为列的标号。给定当前的语法分析器状态(由语法分析栈顶的状态符号表示)和下一个输入符号(标记)，语法分析表说明语法分析器应进行的操作。两个主要的语法分析器操作是移进和归约，语法分析器或者将下一个输入符号移进到语法分析栈的顶部，或者在栈顶端已经有了句柄，将它归约为 RHS 与句柄相同的规则的 LHS。另两个可能的操作是接受和错误，接受意味着语法分析器成功完成了对输入的语法分析，错误意味着语法分析器检测到语法错误。

LR 语法分析表的 GOTO 部分以状态符号作为行的标号，以非终结符作为列的标号。表的 GOTO 部分的值表示在完成归约后，应将哪一个状态符号压入到语法分析栈中，这表示句柄已从语法分析栈中移除，新的非终结符已经压入语法分析栈中。这一特定的符号可以在行中找到，行的标号是句柄和相关状态符号已经移除后语法分析栈顶部的状态符号。所用的 GOTO 表的列，其标号是归约所用规则的 LHS。

考虑以下算术表达式的传统文法：

1. E→E+T
2. E→T
3. T→T*F
4. T→F
5. F→(E)
6. F→id

该文法的规则编了号，以便于在语法分析表中引用它们。

图 4-5 显示了该文法的 LR 语法分析表，操作采用了缩写：R 表示归约，S 表示移进，R4 表示用规则 4 归约，S6 表示将下一个输入符号移进栈内，并将状态 S6 压入栈中。ACTION 表的空白位置表示语法错误。在完整的语法分析器中，可以调用错误处理程序。

LR 语法分析表很容易用软件工具来构造，例如 yacc(Johnson，1975)，它以文法作为输入。虽然 LR 语法分析表可以手工生成，但是对于真正的程序设计语言的文法来说，这个任务太耗时、乏味且容易出错。对于真正的编译器来说，LR 语法分析表总是用软件工具来生成的。

	Action						Goto		
状态	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

图 4-5　算术表达式文法的 LR 语法分析表

LR 语法分析器的初始格局是

$(S_0, a_1 \ldots a_n\$)$

语法分析器的操作的非正式定义如下：

(1) 移进过程很简单：将下一个输入符号与 ACTION 表中移进操作标明的状态符号一起压入栈中。

(2) 对于归约操作来说，必须将句柄从栈中移除。因为栈中的每一个文法符号都有一个状态符号，所以从栈中移除的符号数目是句柄中符号数目的两倍。移除句柄和相关联的状态符号后，将规则的 LHS 压入到栈中。最后使用 GOTO 表，行的标号是从栈中移除句柄和状态符号后露出的符号，列的标号是归约所用规则的 LHS 那个非终结符。

(3) 如果动作是 Accept，那么语法分析完成，并且没有发现错误。

4. 如果动作是 Error，那么语法分析器就会调用一个错误处理程序。

虽然有许多基于 LR 原理的语法分析算法，但是它们只在语法分析表的构造上有所区别。所有的 LR 语法分析器都采用相同的语法分析算法。

熟悉 LR 语法分析过程最好的方法也许就是使用实际的例子。初始时，语法分析栈只有一个符号 0，表示语法分析器的 0 状态。输入包含输入串以及结尾标志(在这里是美元符号)，结尾标志在输入的右端。在语法分析的每一步，根据语法分析栈顶部的(图 4-4 中最右的)符号和下一个(图 4-4 中最左的)输入符号，确定语法分析器的操作。根据语法分析表中 ACTION 部分的相应表格单元选择正确的操作，在归约后使用语法分析表的 GOTO 部分。GOTO 表用于确定在归约后将哪一个状态符号放入语法分析栈。

下面是对串 id+id*id 的语法分析过程的跟踪，采用了 LR 语法分析算法和图 4-5 所示的语法分析表。

语法分析栈	输入	操作
0	id + id * id $	Shift 5
0id5	+ id * id $	Reduce 6(使用 GOTO[0, F])
0F3	+ id * id $	Reduce 4(使用 GOTO[0, T])
0T2	+ id * id $	Reduce 2(使用 GOTO[0, E])
0E1	+ id * id $	Shift 6
0E1+6	id * id $	Shift 5
0E1+6id5	* id $	Reduce 6(使用 GOTO[6, F])
0E1+6F3	* id $	Reduce 4(使用 GOTO[6, T])
0E1+6T9	* id $	Shift 7
0E1+6T9*7	id $	Shift 5
0E1+6T9*7id5	$	Reduce 6(使用 GOTO[7, F])
0E1+6T9*7F10	$	Reduce 3(使用 GOTO[6, T])
0E1+6T9	$	Reduce 1(使用 GOTO[0, E])
0E1	$	Accept

根据给定文法生成 LR 语法分析表的算法在(Aho 等，2006)中有介绍，虽然不很复杂但超出了程序设计语言类书籍的范围。如前所述，有一些软件系统可用于生成 LR 语法分析表。

小　结

不管程序设计语言的实现方法是什么，语法分析是语言实现中共有的部分。语法分析通常基于所实现语言的形式化语法描述。上下文无关文法，通常也称为 BNF，是描述语法最常用的方法。语法分析的任务通常分为两个部分：词法分析和语法分析。将词法分析分离出来有几个原因：简单性、高效率和可移植性。

词法分析器是一个模式匹配器，将程序的小规模部分(称为词素)分离出来。词素具有类别，例如整数文字值和名字，这些类别称为标记。每个标记都被赋予一个数值编码，与词素一起由词法分析器生成。有三种构造词法分析器的方法：采用软件工具来生成表驱动词法分析器的表；用手工建表；编写代码，来实现所实现语言中标记的状态图描述。如果状态转移表示为字符类，而不是每一可能字符都从每个状态结点进行状态转移，标记的状态图就会小得多。此外，使用表格查找来识别保留字可以简化状态图。

语法分析器有两个目的：检测给定程序的语法错误和生成给定程序的语法分析树，或者只是生成构建语法分析树所需的信息。语法分析器是自顶向下的(意味着构建最左推导和按自顶向下的顺序构建语法分析树)或自底向上的(构建逆向的最右推导和按自底向上的顺序构建语法分析树)。可用于所有无歧义文法的语法分析器的复杂度是 $O(n^3)$，不过为程序设计语言实现的语法分析器用于无歧义方法的子集，复杂度是 O(n)。

递归下降的语法分析器是一种 LL 语法分析器，直接根据源程序文法编写代码来实现。EBNF 作为递归下降的语法分析器的基础很理想。递归下降的语法分析器对文法中的每一个非终结符都有一个子程序，如果给定文法的规则只有一个 RHS，那么代码就比较简单，从左向右检查 RHS。对每一个非终结符来说，代码调用该非终结符关联的子程序，对非终结符生成的任何内容进行语法分析。对每一个终结符来说，代码比较终结符和下一个输入标记。如果匹配，代码只需要调用词法分析器，获得下一个标记；如果不匹配，子程序就报告语法错误。如果规则有一个以上 RHS，子程序必须首先确定应该分析哪一个 RHS，必须能够基于下一个输入标记来做出决定。

有两个不同的文法特征会妨碍基于文法构建递归下降的语法分析器。一是左递归，从文法中消除直接左递归的过程相对简单，虽然没有介绍，但有一种算法可以从文法中消除直接和间接左递归。另一个问题可以通过成对不相交测试来检测，这个测试检查语法分析子程序是否能够基于下一个输入标记来确定正在分析哪一个 RHS。一些不能通过成对不相交测试的文法，通常可以采用提取左因子的方法进行修改来通过测试。

自底向上语法分析器的语法分析问题是找到当前句型的子串，必须将这个子串归约为相应的 LHS，以便得到最右推导中的下一个(前一个)句型。这一子串称为句型的句柄。语法分析树可以为识别句型提供直观的感受。自底向上的语法分析器是一种移进-归约算法，因为在大部分情况下，它将下一个输入词素移进到语法分析栈中，或者归约栈顶的句柄。

移进-归约语法分析器的 LR 家族，是程序设计语言最常用的自底向上的语法分析方法，因为这一类语法分析器与其他分析器相比有一些优点。LR 语法分析器使用语法分析栈，栈包含文法符号和状态符号，以维护语法分析器的状态。语法分析栈顶部的符号总是状态符号，表示语法分析栈中与语法分析过程相关的所有信息。LR 语法分析器使用两张语法分析表：ACTION 表和 GOTO 表。ACTION 表说明在给定语法分析栈顶部的状态符号和下一个输入符号时，语法分析器应采取的操作。GOTO 表用于确定归约完成后，应将哪一个状态符号放入语法分析栈。

复 习 题

1. 语法分析器以文法为基础的三个理由是什么？
2. 说明词法分析从语法分析中分离出来的三个理由。
3. 给出词素和标记的定义。
4. 词法分析器的主要任务是什么？
5. 简要描述构建词法分析器的三种方法。
6. 状态转移图是什么？
7. 在词法分析器的状态图中，字母和数字的状态转移采用字符类而不是单个字符是什么原因？
8. 语法分析的两个目的是什么？
9. 说出自顶向下的语法分析器和自底向上的语法分析器之间的区别。
10. 说出自顶向下语法分析器的语法分析问题是什么。
11. 说出自底向上语法分析器的语法分析问题是什么。
12. 为什么编译器采用的语法分析算法只适用于所有文法的一个子集？
13. 对于标记的编码来说，为什么要使用命名常量，而不是数字？
14. 说明如何为只有一个 RHS 的规则编写递归下降的语法分析子程序。
15. 说明妨碍文法作为自顶向下语法分析器基础的两个文法特征是什么。
16. 对于给定文法和句型，FIRST 集合是什么？
17. 描述一下成对不相交测试。
18. 什么是提取左因子？
19. 什么是句型的短语？
20. 什么是句型的简单短语？
21. 什么是句型的句柄？
22. 自顶向下的语法分析器和自底向上的语法分析器是以什么数学机器为基础的？
23. 说出 LR 语法分析器的三个优点。
24. 对于 LR 语法分析技术的实现，Knuth 的看法是什么？
25. 说明 LR 语法分析器的 ACTION 表的作用。
26. 说明 LR 语法分析器的 GOTO 表的作用。
27. 对于 LR 语法分析器来说，左递归是一个问题吗？

习 题

1. 对以下文法规则执行成对不相交测试。

   a. A→aB | b | cBB

   b. B→aB | bA | aBb

   c. C→aaA | b | caB

2. 对以下文法规则执行成对不相交测试。

   a. S→aSb | bAA

   b. A→b{aB} | a

   c. B→aB | a

3. 对于串 a+b*c，给出 4.4.1 节中递归下降的语法分析器的跟踪。
4. 对于串 a*(b+c)，给出 4.4.1 节中递归下降的语法分析器的跟踪。
5. 给定以下文法和右句型，画出语法分析树，指出短语、简单短语和句柄。

   S→aAb | bBA　　A→ab | aAB　　B→aB | b

   a. aaAbb

   b. bBab

   c. aaAbBb

6. 给定以下文法和右句型，画出语法分析树，指出短语、简单短语和句柄。

   S→AbB | bAc　　A→Ab | aBB　　B→Ac | cBb | c

   a. aAcccbbc

   b. AbcaBccb

   c. baBcBbbc

7. 对于串 id*(id+id)，使用 4.5.3 节中的文法和语法分析表，给出完整的语法分析过程，包括语法分析栈的内容、输入串和操作。
8. 对于串(id+id)*id，使用 4.5.3 节中的文法和语法分析表，给出完整的语法分析过程，包括语法分析栈的内容、输入串和操作。
9. 写一条 EBNF 规则描述 Java 或 C++语言中的 while 语句。用 Java 或 C++语言写出这条规则的递归下降子程序。
10. 写一条 EBNF 规则描述 Java 或 C++语言中的 for 语句。用 Java 或 C++语言写出这条规则的递归下降子程序。
11. 从(Aho 等，2006)中了解消除文法中间接左递归的算法。用这个算法消除以下文法中所有的左递归：

    S→Aa | Bb A→Aa| Abc | c | Sb B→bb

编程练习

1. 设计一个状态图，识别基于 C 语言的程序设计语言的一种注释形式，这种注释以/*开始，以*/结束。
2. 设计一个状态图，识别你喜欢的程序设计语言中的浮点字面值。
3. 编写并测试第 1 题中状态图的实现代码。
4. 编写并测试第 2 题中状态图的实现代码。
5. 修改 4.2 节中的词法分析程序，识别以下的保留字列表，返回它们各自的标记编码：

```
for(FOR_CODE, 30), if(IF_CODE, 31), else(ELSE_CODE, 32),
```

```
while(WHILE_CODE, 33), do(DO_CODE, 34), int(INT_CODE, 35),
float(FLOAT_CODE, 36), switch(SWITCH_CODE, 37)
```

6. 将 4.2 节中给出的词法分析程序(用 C 语言写的)转换成 Java 程序。
7. 将 4.4.1 节中给出的用于<表达式>、<项>和<因子>的递归下降语法分析器的程序转换成 Java 程序。
8. 对于第 1 题中通过测试的规则，编写递归下降的语法分析子程序，来分析规则生成的语言。假设有一个词法分析程序 lex 和一个错误处理子程序 error，检测到语法错误时，都可以调用 error。
9. 对于第 2 题中通过测试的规则，编写递归下降的语法分析子程序，来分析规则生成的语言。假设有一个词法分析程序 lex 和一个错误处理子程序 error，检测到语法错误时，都可以调用 error。
10. 实现并测试 4.5.3 节中给出的 LR 语法分析算法。

第5章

名字、绑定和作用域

5.1 概述
5.2 名字
5.3 变量
5.4 绑定的概念
5.5 作用域
5.6 作用域和生存期
5.7 引用环境
5.8 命名常量

本章介绍变量的基本语义问题。首先讨论程序设计语言中名字及特殊字的性质。然后讨论变量的属性，即变量类型、变量地址和变量值，包括别名的问题。接着介绍绑定和绑定时间等重要概念。因为变量属性的绑定时间可能不同，所以定义了四种不同的变量类型。之后，描述两种完全不同的名字作用域规则，即静态作用域和动态作用域，并介绍语句引用环境的概念。最后，还将讨论命名常量和变量初始化。

5.1 概述

命令式程序设计语言是冯·诺依曼计算机体系结构在不同程度上的抽象。在计算机体系结构中，有两个核心的组成部分：存储器(存储命令和数据)和处理器(修改存储器的内容)。变量是语言中机器存储单元的抽象。在某些情况下，这种抽象的特征和存储单元的特征非常相似，整数变量就是这样，整数变量一般完全由单个或多个硬件存储字节来代表。在其他情况下，这种抽象又和硬件存储单元的差异很大，例如三维数组的抽象就需要一个软件映射函数来支持。

变量可以由一组性质或属性来概括，其中最重要的是类型，它是程序设计语言的一项基本概念。设计一种语言的数据类型要考虑到各种因素(数据类型将在第 6 章中讨论)。其中最重要的因素是变量的作用域和生存期。

函数式程序设计语言允许给表达式指定名字。这些命名的表达式看起来像是命令式语言中给变量指定名字，但两者完全不同，因为这些命名的表达式不能改变，所以它们类似于命令式语言中的命名常量。纯函数式语言没有像命令式语言中的变量。但是，许多函数式语言也包含这种变量。

本书的余下部分常常以引用一种语言的方式来引用这个语言的家族。比如，引用 Fortran 时，就是指 Fortran 语言的所有版本。引用 Ada 的时候也是如此。当引述 C 语言时，会包括 C 的原始版本，以及 C89 和 C99 版本。指明语言的某个特定版本，是因为在要讨论的主题中，该版本与同一语言家族的其他成员有所不同。如果在某语言版本名上加注“+”号，就表示该语言从该版本以来的所有版本。例如，Fortran 95+表示自从 Fortran 95 开始的所有 Fortran 版本。“基于 C 的语言”表示 C、Objective-C、C++、Java 和 C#[1]语言。

5.2 名字

在讨论变量之前，必须先讨论变量的一个基本属性——名字。名字还和子程序、形参以及其他的程序结构相关联。标识符常常与名字一词互换使用。

1. 我们本想包括脚本语言 JavaScript 和 PHP 作为基于 C 的语言，但是它们与其祖先的区别太大了。

5.2.1　设计问题

以下是名字的主要设计问题：

- 名字是否区分大小写？
- 语言中的特殊字是保留字还是关键字？

这些问题将在以下两个小节中讨论，其中包括了一些设计选择的例子。

5.2.2　名字形式

名字是用来标识程序中某些实体的字符串。

Fortran 95+允许在一个名字中有多达 31 个字符。C99 对它的内部名称则没有长度上的限制，但只有前 63 个字符有意义。C99 中的外部名字(必须使用语言连接器来处理的、定义于函数之外的名字)至多只能有 31 个字符。在 Java、C#和 Ada 中，名字没有长度限制，名字中的所有字符都是有意义的。C++不限定名字的长度，但实现者有时会加上这个限制。

历史注释

最早的程序设计语言使用单字符名字。这种记法是很自然的，因为早期的程序设计主要用于数学，数学家在正式记法中给未知量长期使用单字符名字。

Fortran I 打破了单字符名字的传统，允许名字至多有 6 个字符。

在大多数程序设计语言中，名字的形式是相同的：一个字母后跟一个包含字母、数字以及下划线(_)的字符串。虽然在 20 世纪 70 年代和 80 年代时曾广泛使用包括下划线的名字，但这种用法现在已经很少了。在基于 C 的语言中，下划线已经在很大程度上被“骆驼”形式的标记所取代，在“骆驼”形式中，多单词的名字除第一个单词之外的其他单词都取首字母大写，例如 myStack[2]。注意，在名字中使用下划线和混合大小写是程序设计风格的问题，而并非语言设计的问题。

PHP 中的所有变量名都必须从一个美元符号开始。在 Perl 语言里，变量名开头的特殊字符，如$、@或%，指定了它的类型(尽管在其他语言中这些字符有不同的含义)。在 Ruby 中，变量名开头的特殊字符，如@或@@，表明该变量是一个实例或类变量。

在许多语言中，尤其是在基于 C 的语言中，名字中的大小写字母是有区别的；也就是说，这些语言中，名字是区分大小写的。例如，rose、ROSE 和 Rose 这三个名字在 C++中是不同的。对某些人而言，这严重地降低了语言的可读性，因为这些看上去十分相像的名字实际上却表示不同的实体。从这种意义上来说，区分大小写违反了语言构造的基本设计原则：看似一样的事物应该具有相同的意义。在变量名区分大小写的程序设计语言中，虽然 Rose 和 rose 看起来很接近，但它们之间一点联系也没有。

显然，并不是每个人都认为名字区分大小写不好。在 C 语言中，区分大小写问题可以通过在名字中不包含大写字母来避免。然而在 Java 和 C#中，这个问题就无法避免，因为许多预定义的名字包括了大写字母和小写字母。例如，Java 中将字符串转换成整数值的方

---

2. 之所以称为“骆驼”，是因为它里面的文字往往会嵌入大写字母，看起来像一个骆驼的驼峰。

法是 parseInt，如果将其拼写为 ParseInt 或者 parseint，就不能被识别。这是可写性的问题，而不是可读性的问题，因为记住特殊情况下的用法，这种需求使得写出正确的程序更为困难。编译器的这种强制要求对语言设计人员而言是不能容忍的。

5.2.3 特殊字

为了提高程序的可读性，程序设计语言中的特殊字用来对要进行的操作进行命名。它们也用来分离程序和语句中的语法部分。大多数语言将特殊字归为保留字，程序员不能重新定义它们，但在某些语言中它们仅仅是关键字，是可以重新定义的。

关键字是程序设计语言中的一种特殊字，仅用于特定的上下文。还在广泛使用的 Fortran 是唯一把特殊字用作关键字的语言。在 Fortran 语言中，单词 Integer 出现在语句的开头，后跟一个名字时，就表示它是一个关键字，说明这条语句是一条声明语句。但如果 Integer 后跟一个赋值操作符，则表示它是一个变量名。这两种不同的用法如下所示：

```
Integer Apple
Integer = 4
```

Fortran 语言的编译器和 Fortran 程序的读者必须通过上下文区分名字与特殊字。

保留字是程序设计语言中的特殊字，它不能用作名字。作为语言设计的选择，保留字比关键字更好，因为允许重新定义关键字会带来混淆。例如，在 Fortran 语言中可以有下面的语句：

```
Integer Real
Real Integer
```

它们声明了程序变量 Real 为整数(Integer)类型，变量 Integer 为实数(Real)类型[3]。这些声明语句不仅外表很奇怪：Integer 和 Real 作为变量名出现在程序中的其他部分，还可能会给程序读者带来误导。

保留字有一个潜在的问题：如果语言中包含了大量的保留字，用户就很难想出不是保留字的名字。关于这个问题最好的例子就是 COBOL，它有 300 个保留字。遗憾的是，程序员最常用的一些名字出现在保留字列表中，例如：LENGTH、BOTTOM、DESTINATION 和 COUNT。

在大多数语言中，有些名字定义在另外的程序单元中，如 Java 语言包、C 和 C++语言库，这些名字能够设置成对某个程序是可见的。它们都是些预定义的名字，必须在显式导入之后才可见。而一旦导入，这些名字就不能重新定义了。

5.3 变量

程序变量是对计算机中一个或一系列存储单元的抽象。程序员常常认为变量是存储地

3. 当然，任何专业的程序员若编写出这样的程序，就应当担心自己的饭碗不保了。

址的名字，但变量比名字具有更多的意义。

从机器语言到汇编语言，主要是用名字来替代数据存储地址的绝对数值，以使程序的可读性更好，因而也就更方便编写和维护。这一步也避免了手工绝对寻址问题，因为将名字转换成实际地址的翻译器也是选择这些存储地址。

变量可以用一个属性六元组来刻画：(名字，地址，数值，类型，生存期，作用域)。尽管这可能使这个看似简单的概念显得过于复杂，但正是这 6 种属性才能最清晰地解释变量的各个方面。

要讨论变量的属性，就要研究许多重要的相关概念，包括别名、绑定、绑定时间、声明、作用域规则以及引用环境。

变量的名字、地址、类型和数值将在下面几节讨论。生存期与作用域属性将分别在 5.4.3 节和 5.5 节讨论。

5.3.1　名字

变量名在程序中是最常见的名字，已在第 5.2 节中讨论过。大多数变量都有名字，没有名字的变量会在 5.4.3.3 节中讨论。

5.3.2　地址

变量的地址是与这个变量相关联的存储地址。然而这个关联关系却不是第一眼看上去那么简单。在很多语言里，程序中相同的变量可以在不同的时间与不同的地址相关联。例如，如果一个子程序有一个局部变量，它是在调用子程序时从运行时堆栈里分配的，对子程序的不同调用会导致该变量有不同的地址。从某种意义上说，它们是相同变量的不同实例。

变量和地址相关联的过程将在 5.4.3 节中进一步讨论。有关子程序以及子程序激活的实现模型将在第 10 章中讨论。

变量地址有时称为变量的左值(l-value)，因为变量名出现在赋值语句的左边时，常常需要变量的地址。

多个变量可以有同一个地址。当用多个变量名来访问相同的存储地址时，这些变量名就称为别名。使用别名有损于可读性，因为它允许通过给一个变量赋值来改变另一个变量的值。例如，变量 total 和 sum 为别名，total 值的任何变化也能改变 sum 的值，反之亦然。程序的读者必须时刻记住，total 和 sum 是同一个存储单元的不同名字。因为在程序中可以有任何数量的别名，这在实际使用时非常困难。别名也使程序的验证更为困难。

在程序中可以用多种不同的方式创建别名。在 C 和 C++语言中，创建别名的一种常用方式是使用联合(union)类型。联合将在第 6 章深入讨论。

两个指向同一存储地址的指针变量也是别名。引用变量也是这样。这种类型的别名是指针与引用特性的一种副作用。设定 C++指针指向一个命名变量后，当指针被解引用时，就是那个命名变量的别名。

在许多语言中，通过子程序的参数也可以创建别名。这种类型的别名将在第 9 章中讨论。

对于理解程序设计语言，变量与地址关联的时间是非常重要的。这个主题将在第 5.4.3 节讨论。

5.3.3 类型

变量的类型决定变量的取值范围，并决定为这种类型的值所定义的操作集合。例如，在 Java 中，int 类型说明以-2147483648～2147483647 的取值范围，以及一组包括加法、减法、乘法、除法和模数的算术运算。

5.3.4 数值

变量的值是与这个变量相关联的存储单元的内容。将计算机存储单元想成抽象的单元，比想成物理的单元要容易得多。无论是存储单元或单个地址单元，在大多数现代的计算机中是以字节为单位的，一个字节通常有八个位。这个数量对于大多数程序变量还是太小了，抽象的存储单元应具有相关变量所需要的大小。例如，尽管在某种语言的特定实现中，浮点数值可能占据四个物理字节，但可以假设一个浮点数值只占有一个抽象存储单元，任何简单的非结构类型的值都只占有一个抽象单元。自此往后，术语“存储单元”是指抽象存储单元。

变量的值有时称为右值(r-value)，因为变量名位于赋值语句的右边时，通常需要变量的值。要想访问右值，必须先确定左值。但这种确定性并不总是简单的。例如，作用域规则会将问题极大地复杂化，详见 5.5 节。

5.4 绑定的概念

绑定是属性和实体之间的关联，例如变量与其类型或值之间的关联，或者操作和符号之间的关联。绑定发生的时间称为绑定时间。在程序设计语言的语义中，绑定和绑定时间是非常重要的概念。绑定可以发生在语言设计阶段、语言实现阶段、编译阶段、载入阶段，链接阶段或运行阶段。例如，星号(*)在语言设计阶段就常常与乘法操作相绑定。数据类型，如 C 语言中的 int，在语言实现阶段与可能的取值范围相绑定的。在编译期间，Java 程序中的变量绑定于某种数据类型。在程序载入内存中时，变量就与存储单元相绑定了。但在某些情况下，这种类型的绑定直到运行期间才发生，例如在 Java 方法中声明的变量。对子程序的调用在链接时才与该子程序相绑定。

考虑下列 Java 赋值语句：

```
count = count + 5;
```

对这条赋值语句来说，涉及的绑定与绑定时间如下所示：

- count 的类型在编译时绑定。

- count 的可能取值集合在设计编译器时绑定。
- 操作符+的含义在编译时绑定，在确定操作数的类型之后进行。
- 字面量 5 的内部表示在设计编译器时绑定。
- count 的值在这条语句执行时绑定。

全面理解程序实体属性的绑定时间，是理解程序设计语言语义的前提。例如，要理解子程序的作用，就必须先理解子程序调用中的实参是怎样与子程序定义中的形参相绑定的。要决定变量的当前值，就必须知道这个变量是在什么时候通过什么语句与存储单元绑定的。

5.4.1　属性与变量绑定

如果绑定是在运行之前第一次出现，且在整个程序的执行过程中保持不变，这种绑定就是静态的。如果绑定是在运行期间第一次出现，或者能在程序执行期间改变，这种绑定就是动态的。在虚拟的存储器环境中，变量与存储单元的物理绑定是十分复杂的，因为在程序执行期间，存储单元所在的地址空间段或地址空间页可能会多次移进或移出存储器。在某种意义上，这些变量是被重复地绑定与释放。不过，这是由计算机硬件来维护的，并且对程序和用户都是不可见的。由于这些和这里讨论的话题关系不大，所以不考虑硬件绑定。最关键的是要区分静态绑定与动态绑定。

5.4.2　绑定类型

变量能在程序中引用之前，必须被绑定到一个数据类型上。这种绑定具有两个重要方面：一是指定类型的方式，二是绑定发生的时间。变量的类型可以通过显式或者隐式的声明来静态地指定。

5.4.2.1　静态的绑定类型

显式声明是程序中的一条说明语句，它列出一批变量名并指明这些变量的特定类型。隐式声明则是通过默认约定而不是声明语句将变量与类型相关联的方式。在这种情况下，变量名在程序中的第一次出现即构成了它的隐式声明。显式声明与隐式声明都产生对类型的静态绑定。

大多数广泛使用的程序设计语言都只使用静态的类型绑定，都是 20 世纪 60 年代中期以后设计的，都要求对所有的变量予以显式声明(Perl、JavaScript、Ruby 和 ML 是这些语言中的例外)。

隐式的变量类型绑定是由语言处理器(编译器或解释器)完成的，隐式的变量类型绑定有几个不同的基础。其中最简单的是命名约定。在这种情况下，编译器或解释器根据变量名的语法形式将变量与类型相绑定。例如在 Fortran 语言中，程序中出现的标识符在没有显式声明时，是根据下面的约定隐式声明的：如果标识符的开头字母是 I、J、K、L、M、N 或其小写字母，它就隐式地声明为 Integer 类型；否则，它就声明为 Real 类型。

尽管隐式声明能给程序员带来一点方便，但降低了程序的可靠性，因为它妨碍了在编

译过程中发现输入错误和程序员的编程错误。Fortran 的程序员不经意漏掉声明的变量会自动指定默认类型和一些未曾预料的属性，这将导致难以诊断的错误。现在很多 Fortran 程序员都在程序中包含了声明 Implicit none。这种声明使编译器不会隐式地声明任何变量，因此避免了偶然漏掉声明的变量所引发的潜在问题。

要求特殊类型的名字以某些特殊的字符开始，也可以避免隐式声明带来的一些问题。例如，在 Perl 语言中，任何以字符$开头的名字都是标量，它能够存放字符串或者数字值；如果名字的开头字符是@，那么它就是一个数组；如果是以字符%开头，那么它就是一个散列结构[4]。这可以为不同类型的变量创建不同的名称空间。在这种情况下，名字@apple 和%apple 是不相关的，因为它们来自不同的名称空间。此外，程序的读者只要读到变量名，就必定知道它的类型。注意，这个规定不同于 Fortran 语言，因为 Fortran 语言不仅包括隐式声明，还包括显式声明，所以变量名的拼法不一定能够确定变量的类型。

另一种隐式类型声明使用上下文。这有时称为类型推理。在较简单的情形中，上下文是在声明语句中赋值给变量的值的类型。例如，在 C#中，变量的 var 声明必须包含初始值，其类型就是变量的类型。考虑下面的声明：

```
var sum = 0;
var total = 0.0;
var name = "Fred";
```

sum、total 和 name 的类型分别是 int、float 和 string。注意，这些都是静态类型的变量，在声明它们的程序单元的生存期中，它们的类型是固定的。

Visual BASIC 9.0+、Go 和函数式语言 ML、Haskell、OCaml 和 F#也使用类型推理。在这些函数式语言中，变量名所在的上下文是确定变量类型的基础。这种类型推理在第 15 章讨论。

5.4.2.2 动态的绑定类型

在使用动态类型绑定时，变量的类型既不是由声明语句来说明的，也不能通过名字的拼法来确定。而是在赋值语句给变量赋值时，变量才与类型相绑定。当执行赋值语句时，被赋值的变量与赋值语句右边表达式值的类型相绑定。这个赋值语句也可以把变量与地址和存储单元相绑定，因为不同类型的值可能需要不同的存储空间。任何变量都可以赋予任何类型的值。而且，变量的类型在程序执行期间可以改变任意多次。注意，变量的类型如果是动态绑定的，就可能是临时的。

变量的类型是静态绑定的时，变量名就可以看作与类型相绑定，此时，变量的名字和类型就同时绑定了。但是，变量的类型是动态绑定的时，其名字可以看作仅临时绑定到类型上。实际上，变量的名字从来不绑定到类型上。名字可以绑定到变量上，变量可以绑定到类型上。

---

4. 数组和散列都是要考虑的类型，因为它们都可以在其元素中存储任意标量。

动态类型绑定的语言与静态类型绑定的语言有很大区别。变量与类型动态绑定的主要优势是程序设计的灵活性更大。例如，在使用动态类型绑定的语言中，可以将处理数值数据的程序写成通用程序。这就意味着它能够处理任何数字类型的数据，因为把输入数据赋给变量时，用来存储这些数据的变量能够与正确的类型相绑定，所以无论输入什么类型的数据，它都可以接受。反之，如果只使用静态类型绑定，在不知道数据的类型之前，不可能编写一个 C 程序来处理这组数据。

在 20 世纪 90 年代中期以前，应用最广泛的程序设计语言都使用静态类型绑定，只有一些函数式语言(如 LISP)例外。但 90 年代中期以后，使用动态类型绑定的语言有了很大的变化。在 Python、Ruby、JavaScript 和 PHP 中，类型的绑定是动态的。例如，JavaScript 脚本可以包含以下语句：

```
list = [10.2, 3.5];
```

不管变量 list 以前是什么类型，上面的赋值都使 list 成为一个长度为 2 的一维数组。如果之后的赋值语句为：

```
list = 47;
```

在赋值之后，list 就成为一个标量变量。

动态类型绑定在 C# 2010 中引入。在声明中包含 dynamic 保留字，变量就可以声明为使用动态类型绑定，如下所示：

```
dynamic any;
```

这是类似的，但不同于将 any 声明为 object 类型。类似的原因是，any 可以被赋予任意类型的值，就好像把它声明为 object 一样。不同的原因是，它不能用于几个不同的交互操作环境；例如不能用于动态类型化的语言 IronPython 和 IronRuby 等(分别是 Python 和 Ruby 的.NET 版本)。但是，程序从外部源中导入未知类型的数据时，就可以使用它。类成员、属性、方法参数、方法返回值和局部变量都可以声明为 dynamic。

在纯面向对象的语言如 Ruby 中，所有的变量都是引用，没有类型；所有的数据都是对象，任何变量都可以引用其他变量。在某种程度上，这种语言中的变量都具有相同的类型——它们都是引用。但是，Java 中的引用只能引用特定类型的值，而 Ruby 中的变量可以引用任意对象。

动态类型绑定有两个缺点。首先，它降低了程序的可靠性，因为相对于具有静态类型绑定语言的编译器而言，使用动态类型绑定的编译器发现错误的功能较弱。动态类型绑定允许将任何变量赋予任何类型的值，在赋值语句右边的错误类型就不能检测为错误，反之，它会将左边的类型改成错误的类型。例如，假设在某个 JavaScript 程序中，i 和 x 是目前的数值标量变量名，而 y 目前是一个数组名。该程序需要赋值语句：

```
i = x;
```

但由于键入错误，写成了赋值语句

```
i = y;
```

在 JavaScript(或任何一种动态类型绑定的语言)中，解释器都检查不出这个语句中的错误，而只是将 i 的类型改为数组。但在后来使用 i 时，i 会被当成一个标量来用，想得到正确的结果是不可能的。在具有静态类型绑定的语言中，比如 Java，编译器就会发现赋值 i=y 的这个错误，因此不会执行程序。

注意，这个缺点也不同程度地出现在一些使用静态类型绑定的语言中，例如 Fortran、C 以及 C++，在很多情况下，编译器会自动将赋值语句中等号右边的类型转换成等号左边的类型。

也许动态类型绑定最大的缺点就是它的成本。实现动态属性绑定的成本是相当高的，特别是类型检测必须在运行时进行。另外，任何变量必须有一个相关的运行时描述符来维护当前类型。变量值的存储空间必须大小可变，因为不同类型的数值需要不同大小的存储空间。

最后，对变量进行动态类型绑定的语言通常用纯解释器而不是编译器来实现。在编译时，操作数的类型如果是未知的，计算机就无法给出指令。如果编译时不清楚公式 A+B 中 A 和 B 的类型，则编译器就不能为表达式 A+B 生成机器指令。相对于执行等价的机器码，纯粹的解释执行要花费至少十倍的时间。当然，如果一种语言是使用单纯的解释器来实现的，它用于动态类型绑定的时间就隐藏在总的解释时间中，因此这个环境中，类型绑定的成本似乎并不高。另一方面，使用静态类型绑定的语言很少通过纯解释器来实现，因为将这些语言编写的程序翻译成高效的机器码版本十分容易。

5.4.3 存储绑定和生存期

命令式程序设计语言的根本特征在很大程度上取决于这种语言中变量存储绑定的设计方式，因此清晰地理解变量存储绑定是非常重要的。

变量所绑定的存储单元必须取自一个可用的存储单元池。这个过程就称为存储空间分配。存储空间解除分配是将已经与变量解除绑定的存储单元重新放回这个池中的过程。

变量的生存期是变量绑定在某个存储地址的时间。因此变量的生存期开始于变量绑定到某个存储单元时，结束于该变量从存储单元上解除绑定时。为了分析变量的存储绑定，把标量(非结构性的)变量的生存期分成四类会更方便些，即分为静态变量、栈动态变量、显性堆动态变量以及隐性堆动态变量。下面几节将讨论这四种类型的定义、它们的作用和优缺点。

5.4.3.1 静态变量

静态变量在程序运行开始之前，就绑定到存储单元上，且在程序运行结束之前，一直绑定在相同的存储单元上。静态绑定变量在程序设计中有多种用途。显然，全局可存取的变量常常用于程序执行的整个过程中，因此必须在程序运行的整个期间都将它们绑定在相同的存储空间上。有时让子程序中声明的变量成为历史敏感较为方便。这样的子程序必须

有局部的静态变量。

静态变量的另一个优点是高效率。所有的静态变量都可以直接寻址[5]；而其他类型的变量常常需要间接寻址，间接寻址的存取速度比较慢。另外，静态变量在运行时没有进行分配与解除分配所需要的开销，尽管这个开销常常可以忽略不计。

静态绑定存储空间的一个缺点是降低了灵活性；特别是，只有静态变量的语言不支持递归子程序。另一个缺点是这些变量不能够共享存储空间。比如，一个程序有两个子程序，这两个子程序都需要大型数组，并且从不同时调用这两个子程序。如果数组都为静态的，这两个子程序的数组就不能够共享存储空间。

C 和 C++允许程序员在函数内变量的定义中包括 static 修饰符，使所定义的变量成为静态的。注意，当 static 修饰符出现于 C++、Java 以及 C#的类定义中的变量声明时，表示所声明的变量是一个类变量，而不是一个实例变量。有时，在类第一次实例化之前，就创建了类变量。

5.4.3.2　栈动态变量

栈动态变量在声明语句确立之后才产生存储绑定，而它们的类型是静态绑定的。声明的确立过程是指由声明所指示的存储空间分配以及绑定过程，它是在程序执行到声明所在的代码时发生的。因此这种确立发生在运行期间。例如，在 Java 方法头部声明的变量，是在调用方法时确立，而当方法执行完成时，将声明定义的变量解除分配。

顾名思义，栈动态变量的存储空间是从运行时栈中分配的。

有些语言(如 C++和 Java)允许变量声明在语句的任何地方出现。在这些语言的一些实现中，所有在函数或方法中声明的栈动态变量(不包括在嵌套块中声明的变量)，都可以在这个函数或方法开始执行时，绑定到存储空间，即使其中某些变量不是一开始就声明了。这种情况下，在声明之后变量就是可见的了，但存储空间的绑定(以及初始化，如果在声明中指定了初始化)是在函数或方法开始执行时才发生的。在变量可见之前进行变量的存储空间绑定，这对语言的语义没有影响。

栈动态变量的优点如下：至少在大部分情况下，能够实际运用的递归子程序需要有某种形式的动态局部存储，这样，递归子程序的任何一个活动副本都有自己的局部变量版本，而使用栈动态变量，很容易满足这些要求。甚至在没有递归的情况下，子程序的栈动态局部存储也有着优越性，因为子程序中的所有局部变量都可以共享相同的存储空间。

相对于静态变量，栈动态变量的缺点有：运行时的分配和解除分配需要额外的开销，由于间接寻址导致访问较慢，以及子程序不是历史敏感的等。栈动态变量的分配和解除分配并不需要耗费大量的时间，因为在子程序头部声明的所有栈动态变量都同时进行分配与解除分配，而不是分别操作的。

Fortran 95+允许实现人员局部地使用栈动态变量，但需要包括这样一条语句：

---

5. 在一些实现中，静态变量通过基址寄存器来寻址，访问它们的开销与访问栈分配的变量差不多。

```
Save list
```

这个声明允许程序员将该子程序中的部分或者全部变量(即 list 中的那些变量)指明为静态的。

在 Java、C++和 C#中，在方法中定义的变量默认是栈动态的。在 Ada 中，子程序中定义的所有不在堆上的变量都是栈动态的。

除存储之外的所有属性都静态地绑定到栈动态标量变量上。有些结构化类型并不是这样，如第 6 章所述。对于栈动态变量的分配和解除分配过程的实现，参见第 10 章。

5.4.3.3 显式堆动态变量

显式堆动态变量是无名(抽象)的存储单元，由程序员编写的显式运行时指令来进行分配与解除分配。这些变量在堆上分配和解除分配，只能通过指针变量或引用变量来引用。堆是一组存储单元的集合，由于堆的使用不可预知，所以它在组织上是高度松散的。访问显式堆动态变量的指针和引用变量的创建方法与其他标量变量一样。通过操作符(如在C++中)或调用为此提供的系统子程序(如在 C 中)，可以创建显式堆动态变量。

在 C++语言中，分配操作符 new 使用类型名作为它的操作数。在执行分配操作时，会生成一个操作数类型的显式堆动态变量，并返回它的地址。由于显式堆动态变量在编译时和类型相绑定，因而这种绑定是静态的。然而，这种变量与存储空间的绑定发生在它们被创建的时候，即在运行时。

除了用于产生显式堆动态变量的子程序或操作符以外，一些语言还包括用于显式删除它们的子程序或操作符。

下面这段 C++程序是显式堆动态变量的一个例子：

```
int *intnode;          // Create a pointer
intnode = new int;     // Create the heap-dynamic variable
...
delete intnode;        // Deallocate the heap-dynamic variable
                               // to which intnode points
```

在这个例子中，int 类型的显式堆动态变量是通过操作符 new 创建的，接着通过指针 intnode 来引用这个变量。然后通过 delete 操作符将此变量解除分配。C++语言要求有显式解除分配操作符 delete，因为它不使用隐式存储空间的回收，如垃圾收集。

在 Java 中，除基本标量外，所有数据都是对象。Java 的对象都是显式堆动态变量，通过引用变量来访问。Java 不能显式销毁堆动态变量，相反，它使用的是隐式垃圾收集。垃圾收集参见第 6 章。

C#语言不仅有显式堆动态对象，还有栈动态对象，这两者都是隐式地解除分配。不仅如此，C#还支持 C++风格的指针。这种指针可以引用堆、栈，甚至是静态的变量和对象。这些指针与 C++中的指针一样具有危险性，而且由这些指针引用的堆对象并不是隐式地解除分配。C#语言之所以包括这些指针，是希望 C#组件能够与 C 及 C++组件一起工作。为了表明不支持使用指针，且清楚地告诉程序读者，代码使用了指针，所有定义了指针的方

法头部都必须包括保留字 unsafe。

显式堆动态变量常常用于构建动态结构，比如链表和树结构，这些结构在运行期间需要生长或收缩。通过指针或引用和显式堆动态变量，很容易构造出这类结构。

显式堆动态变量的缺点有：正确使用指针和引用变量比较困难，引用变量有额外的开销，存储管理的实现比较复杂。这实际上就是堆的管理问题，开销大而且复杂。显式堆动态变量的实现方法，参见第 6 章。

5.4.3.4　隐式堆动态变量

隐式堆动态变量只有赋值时，才绑定到堆存储空间。实际上，每次赋值时，都会绑定它们所有的属性。例如下面的 JavaScript 赋值语句：

```
highs = [74, 84, 86, 90, 71];
```

不管变量 highs 在程序中是否早已用过，也不管它用在什么方面，现在，它是一个具有 5 个数值的数组。

这种变量的优点是它们具有高度的灵活性，允许编写极为通用的程序，而缺点是在运行时维护所有动态属性的额外开销，其中包括数组下标的类型和范围。另一个缺点是编译器会遗漏对某些错误的检查，参见 5.4.2.2 节。JavaScript 中隐式堆动态变量的例子参见 5.4.2.2 节。

5.5　作用域

作用域是理解变量的重要因素之一。变量的作用域是语句的一个范围，这个范围之内的变量都是可见的。如果一个变量可以在某条语句中引用，那么这个变量在该语句中就是可见的。

语言的作用域规则决定如何将一个名字的特定出现与一个变量相关联。特别地，作用域规则决定了对于在当前执行的子程序或程序块之外声明的变量，如何将它们的引用与它们的声明以及属性关联起来(程序块将在第 5.5.2 节讨论)。清晰地理解语言中的这些规则，对运用这种语言编写程序或阅读代码的能力是非常关键的。

如果变量声明于程序单元或程序块之内，它就是这个程序单元或程序块内的局部变量。而程序单元或程序块的非局部变量是不在这个程序单元或程序块中声明、但对其可见的变量。全局变量是非局部变量的一种特殊类型，参见 5.5.4 节。

关于类、包和名称空间的作用域问题将在第 11 章中讨论。

5.5.1　静态作用域

ALGOL 60 引入了将名字与非局部变量相绑定的方法，称为静态作用域[6]。后来的许多

6. 静态作用域有时称为词法作用域。

命令式语言和很多非命令式语言也采用了该方法。之所以称为静态作用域，是因为可以静态地决定变量的作用域，即在执行之前就决定变量的作用域。这就允许程序阅读人员(以及编译器)仅查看源代码，就可以决定程序中每个变量的类型。

有两类静态作用域语言：一类是语言中的子程序可以嵌套，并产生嵌套的静态作用域；另一类是语言中的子程序不能嵌套。在后一类语言中，还是由子程序来产生静态作用域，但是只有嵌套的类定义和嵌套块能够产生嵌套的作用域。

Ada、JavaScript、Common LISP、Scheme、Fortran 2003+、F#和 Python 允许嵌套子程序，然而基于 C 的语言却不允许。

本节对静态作用域的讨论将集中在允许嵌套子程序的语言上。一开始我们假设所有作用域都与程序单元相关联，所有引用的非局部变量都在其他程序单元中声明[7]。而本章假设在所讨论的语言中，作用域是访问非局部变量的唯一方法。这个假设并非对所有语言都成立。甚至并非对所有使用静态作用域的语言都成立。这种假设仅仅是为了简化这里的讨论。

假设一个程序是由使用静态作用域的语言来编写的，当这个程序的阅读者发现对一个变量的引用时，要想确定该变量的属性，就必须找到该变量的声明语句(显式或隐式声明)。在具有嵌套子程序的静态作用域语言中，这个过程可以这样进行：假设子程序 sub1 中对变量 x 进行引用。首先通过搜索子程序 sub1 中的声明来寻找 x 的正确声明。如果在子程序 sub1 中没有发现这种声明，将继续在声明子程序 sub1 的子程序声明中进行搜索，该子程序称为 sub1 的静态父辈。如果还没有找到变量 x 的声明，则继续到下一个更大的包含单位(声明子程序 sub1 的父辈的程序单元)中进行搜索，以此类推，直到发现变量 x 的声明为止，或者已经搜索了最大程序单元中的声明，但没有找到变量 x 的声明为止——在这种情况下，就发现了一个变量未声明的错误。子程序 subl 的静态父辈和静态父辈的静态父辈，由此往上直到最大的包含子程序，称为 sub1 的静态祖先。静态作用域的实际实现(将在第 10 章讨论)通常比以上描述的过程更有效率。

思考下面的 JavaScript 程序 big，这里面嵌套了两个函数 sub 1 和 sub2：

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

在静态作用域下，对 sub2 中变量 x 的引用也就是对程序 big 中声明的变量 x 的引用。因

7. 未在其他程序单元中定义的非局部变量将在 5.5.4 节中讨论。

为对 x 的搜索开始于引用发生的程序 sub2，然而没有在那里发现 x 的声明。因此继续搜索 sub2 的静态父辈 big，终于在那里找到了 x 的声明。因为 x 不在 sub2 的静态祖先中，所以忽略了 sub1 中声明的 x。

在使用静态作用域的语言中，不管这种语言是否允许嵌套的子程序，一些变量的声明对其他代码段都可以隐藏起来。例如，再次考虑 JavaScript 过程 big。变量 x 在 big 和 sub1 中都有声明，而 sub1 嵌套在 big 中。在 sub1 内，对 x 的任何引用都是在引用局部的 x。因此，外部的 x 隐藏在 sub1 中。

在 Ada 中，祖先作用域中的隐藏变量可以使用选择引用来访问，这种选择引用包括了祖先作用域的名字。例如前面的 big 函数用 Ada 编写，则通过引用 big.x 就可以在 sub1 中访问在 big 中声明的 x。

5.5.2　块

许多语言都允许在可执行代码中定义新的静态作用域。这个由 ALGOL 60 引入的十分有用的概念，允许一段代码拥有自己的局部变量，而这些局部变量的作用域就是这段代码。这种变量是典型的栈动态变量，因而当执行到代码段时，就给变量分配存储空间，而代码段退出执行时，便进行存储空间的解除分配。这段代码称为块，它是块结构语言这个说法的依据。

基于 C 的语言允许任何复合语句(由配对的花括号界定的语句系列)声明变量，并由此定义新的作用域。这样的复合语句就是块。例如，如果 list 为一个整数数组，就可以编写如下程序：

```
if (list[i] < list[j]) {
  int temp;
  temp = list[i];
  list[i] = list[j];
  list[j] = temp;
}
```

处理这些块产生的作用域时(这些块可能嵌套在更大的块中)，完全可以和处理由子程序产生的作用域一样。对那些声明不在块中的变量的引用，可以按照从小到大的顺序来搜索包含它的作用域(块或子程序)，从而找到变量声明。

考虑下面的 C 函数框架：

```
void sub() {
  int count;
  ...
  while (...) {
    int count;
    count++;
    ...
  }
  ...
}
```

在 while 循环中，对 count 的引用是对这个循环中局部 count 的引用。在这种情况下，函数 sub 所声明的 count 对于 while 循环中的代码段是隐藏的。一般而言，一个变量声明事实上隐藏了在包含它的更大作用域中任何同名变量的声明[8]。注意，这种代码在 C 和 C++中是合法的，然而在 Java 和 C#中是不合法的。Java 和 C#的设计人员认为，在嵌套块中重用名字太容易出错，所以不允许这样使用。

尽管 JavaScript 为嵌套的函数使用静态作用域，但不能在这种语言中定义非函数块。

大多数函数式程序设计语言都包含一个结构，它与命令式语言中的块相关，通常称为 let。这个结构包含两个部分，第一部分用于把名字绑定到值上，通常指定为表达式；第二部分是一个表达式，它使用第一部分定义的名字。函数式语言中的程序由表达式组成，而不是由语句组成。因此，let 结构的后一部分是一个表达式，而不是语句。在 Scheme 中，let 结构是对 LET 函数的调用，其形式如下：

```
(LET (
   (名字 1 表达式 1)
   ...
   (名字 n 表达式 n))
   表达式
)
```

LET 调用的语义如下：计算前 n 个表达式，将其值赋予相关的名字。接着计算最后一个表达式，其值就是 LET 的返回值。这不同于命令式语言中的块，因为名字就是值，而不是命令式语言中的变量。名字设置好后，就不能改变。但是，它们类似于命令式语言中块的局部变量，因为它们的作用域是 LET 调用的一部分。例如下面的 LET 调用：

```
(LET (
   (top (+ a b))
   (bottom (- c d)))
   (/ top bottom)
)
```

这个调用计算表达式(a + b) / (c – d)，并返回其值。

在 ML 中，let 结构的形式如下：

```
let
 val 名字 1 = 表达式 1
 ...
 val 名字 n = 表达式 n
in
 表达式
end;
```

每个 val 语句都把名字绑定到表达式上。与 Scheme 相同，第一部分的名字类似于命令

8. 如 5.5.4 节所述，在 C++中，可以使用域操作符(::)来访问内部作用域中隐藏的全局变量。

式语言中的命名常量，一旦设置好，就不能改变[9]。考虑下面的 let 结构：

```
let
  val top = a + b
  val bottom = c - d
in
  top / bottom
end;
```

在 F#中，let 结构的一般形式如下：

```
let left_side = 表达式
```

let 的 left_side 可以是名字或元组模式(一系列用逗号隔开的名字)。

在函数定义中用 let 定义的名字的作用域是定义表达式的结尾到函数的结尾。let 的作用域可以通过缩进如下代码来界定，这段代码创建了一个新的局部作用域。尽管任何缩进都是有效的，但约定是，缩进应有 4 个空格。考虑下面的代码：

```
let n1 =
    let n2 = 7
    let n3 = n2 + 3
    n3;;
let n4 = n3 + n1;;
```

n1 的作用域包括整个代码段，而 n2 和 n3 的作用域在缩进尾部结束。所以在最后一个 let 中使用 n3 会导致错误。let n1 作用域的最后一行是将值绑定到 n1 上，它可以是任意表达式。

第 15 章详细介绍了 Scheme、ML、Haskell 和 F#中的 let 结构。

5.5.3　声明的次序

在 C89 以及其他一些语言中，所有函数中的数据声明，除了嵌套块中的以外，都必须放在函数的开头。但是，有一些语言，比如，C99、C++、Java、JavaScript 和 C#，允许变量声明出现在程序单元语句中的任何地方。变量声明可以生成与复合语句或子程序无关的作用域。比如，在 C99、C++和 Java 中，所有本地变量的作用域都是从声明处开始，到包含这些声明的块的末尾结束。但是，C#的某个块中声明的任何变量，不管声明出现在块中哪个位置，只要它不在嵌套块中，其作用域都是整个块。这个规则也适用于方法。注意，C#仍然要求所有变量在使用前声明。因此，尽管变量的作用域能从声明处扩展到包含声明的块或子程序的顶端，变量仍然不能在声明之前使用。

在 JavaScript 中，局部变量可以在函数的任意位置声明，但这种变量的作用域总是整个函数。如果在函数中声明一个这样的变量之前使用它，该变量的值就是 undefined。

C++、Java 以及 C#中的 for 语句允许在它们的初始化表达式中放置变量定义。在 C++的

9. 如第 15 章所述，名字是可以重置的，但其过程实际上会创建一个新的名字。

早期版本中，这种变量的作用域是从变量定义的位置到包含它的最小块的末尾。然而在 C++的标准版本中，其作用域限制于 for 结构之内，这和 Java 以及 C#语言中的情形一样。考虑下面方法的代码框架：

```
void fun() {
   ...
   for (int count = 0; count < 10; count++){
      ...
   }
   ...
}
```

在 C++的后续版本、Java 和 C#中，count 作用域是从 for 语句到循环体结束。

5.5.4 全局作用域

一些语言，包括 C、C++、PHP、JavaScript 和 Python 语言，允许程序由函数定义的序列构成，其中变量定义可以在函数的外面。在源文件中，定义在函数外面的变量是全局变量，它们对文件中的函数是可见的。

C 和 C++都支持全局数据的声明和定义。声明指定了数据的类型和其他属性，但不会分配存储空间。定义则指定属性，并分配存储空间。对某个特定的全局名字，C 程序可以有任何次数的兼容性声明，但只能有一次定义。

在函数定义外面的变量声明，指明了该变量是在另一个文件中定义的。C 的全局变量对文件中所有后续函数都是隐含可见的，那些包含同名局部变量声明的函数除外。在函数后面定义的全局变量，可以使它在函数中是可见的，办法是把它声明为外部的，格式如下：

```
extern int sum;
```

在 C99 中，全局变量的定义总是有初始值。全局变量的声明则从来没有初始值。如果声明是在函数定义的外面，则它不需要包含 extern 修饰符。

声明和定义的这种思路，也可以推广到 C 和 C++的函数上，即函数原型声明了函数的名字和接口，但不提供代码；而函数的定义则带有代码，是完整的。

在 C++中，可以使用域操作符(::)来访问被局部同名变量隐藏的全局变量。例如，如果全局变量 x 通过一个局部变量 x 而隐藏于一个函数中，则这个全局变量可以用::x 来引用。

PHP 语句能散布在函数定义之间。PHP 变量出现在语句中时，是被隐式地声明的。在函数外面隐式声明的任何变量都是全局变量；而在函数中隐式声明的变量都是局部变量。全局变量的作用域从它们的声明点开始，到程序结束，但不包括任何后续的函数定义。所以，全局变量并不是在任意函数中都是隐式可见的。有两种方法使全局变量对作用域中的函数是可见的：(1)如果函数包含一个与全局变量同名的局部变量，则全局变量可以通过 $GLOBALS 数组来访问，以全局变量名作为数组的字符串常量下标。(2)如果在函数中没有与全局变量同名的局部变量，则在 global 声明语句中包含全局变量，全局变量就可以成为可见的。考虑下面的例子：

```
$day = "Monday";
$month = "January";

function calendar() {
  $day = "Tuesday";
  global $month;
  print "local day is $day <br />";
  $gday = $GLOBALS['day'];
  print "global day is $gday <br \>";
  print "global month is $month <br />";
}

calendar();
```

上述代码的解释执行将产生下面的输出：

```
local day is Tuesday
global day is Monday
global month is January
```

JavaScript 的全局变量非常类似于 PHP 全局变量，但无法访问与函数中局部变量同名的全局变量。

Python 中全局变量的可见性规则有些特殊。变量通常没有声明，这一点跟 PHP 一样。它们作为赋值语句的目标时，会隐式地声明。全局变量可以在函数中引用，但它只有在函数中声明为全局变量，才能在函数中赋值。考虑下面的例子：

```
day = "Monday"

def tester():
  print "The global day is:", day

tester()
```

因为全局变量能够在函数中直接引用，所以这个脚本程序的输出如下：

```
The global day is: Monday
```

下面的脚本程序试图向全局变量 day 赋予一个新值：

```
day = "Monday"

def tester():
  print "The global day is:", day
  day = "Tuesday"
  print "The new value of day is:", day

tester()
```

这个脚本产生了一个 UnboundLocalError 错误信息，因为函数体第 2 行中的 day 赋值语句使 day 成为一个局部变量，于是，函数体第 1 行对 day 的引用就变成对局部变量的不合法的前向引用。

如果在函数开头把 day 声明为全局变量，则对 day 的赋值就可以是对全局变量赋值。这就防止了对 day 的赋值生成一个局部变量。如下面的脚本代码所示：

```
day = "Monday"

def tester():
  global day
  print "The global day is:", day
  day = "Tuesday"
  print "The new value of day is:", day

tester()
```

这个脚本的输出如下：

```
The global day is: Monday
The new value of day is: Tuesday
```

在 Python 中可以嵌套函数，在嵌套函数中定义的变量可以通过静态作用域在嵌套函数中访问，但这些变量必须在嵌套函数中声明为 nonlocal[10]。5.7 节中的框架程序示例说明了如何访问非局部变量。

在 F#中，在函数定义外部定义的所有变量都是全局变量。它们的作用域从其声明点开始到文件尾。

在面向对象语言的类与成员声明中，声明的顺序和全局变量也是问题。这些将在第 12 章中讨论。

5.5.5 静态作用域的评价

静态作用域提供的非局部访问方法在许多情况下都工作得很好。但是这种方法也存在一些问题。首先，在大多数情况下，对变量和子程序而言，它允许的访问权限比需要的更多。这对于精确指定访问限制而言是一个过于简单的粗糙工具。其次，也可能是更重要的，是一个与程序进化有关的问题。软件是高度动态的——我们用的程序要定期持续地变化。这些改变常常导致代码重构，从而摧毁了限制变量和子程序访问的初始结构。为了避免维护这些访问权限的复杂性，当开发者碰到障碍时，就会抛弃这些结构。于是，避开静态作用域限制将导致设计的程序与原先的预想有很大差异，甚至程序中没有变化的部分也是如此。于是，设计人员只能使用比他们需要的多得多的全局变量。最终，所有子程序都在同一层次(即主程序)中嵌套，使用全局变量而不是较深层次的嵌套[11]。而且，最终的设计会显得笨拙、勉强，不能反映基础概念的设计。静态作用域存在的以上问题和其他缺点，在(Clarke、Wileden 和 Wolf，1980)中有详细的讨论。使用静态作用域控制变量和子程序访问权限的一种替代办法是封装结构，许多新的语言都包含它。封装结构将在第 11 章中讨论。

10. nonlocal 保留字在 Python 3 中引入。
11. 类似于 C 语言的结构。

5.5.6　动态作用域

在 APL、SNOBOL4 和 LISP 的早期版本中，变量的作用域是动态的。Perl 和 Common LISP 也允许变量具有动态作用域，即便在这些语言中，默认的作用域机制是静态的。动态作用域基于子程序的调用序列，而不是基于作用域相互之间的空间关系。因此，这种作用域只能在运行期间确定。

再次考虑 5.5.1 节中的 big 函数，重写它时，去掉了函数调用，如下：

```
function big() {
  function sub1() {
    var x = 7;
  }
  function sub2() {
    var y = x;
    var z = 3;
  }
  var x = 3;
}
```

假设将动态作用域规则应用于非局部引用。sub2 中引用的标识符 x 的含义是动态的，即它不能在编译期间确定。它可能从 x 的两个声明中引用任何一个变量，这取决于子程序的调用顺序。

在运行期间确定 x 的正确含义的一种方法是从局部声明开始搜索。这也是处理过程从静态作用域开始的方式，但除此之外，这两种技术之间没有别的相似之处。当局部声明搜索不成功时，就搜索其动态父辈(或调用函数)的声明。如果在那里也没有发现 x 的声明，就继续搜索到该函数的动态父辈，依此类推，直到找到 x 的声明为止。如果在任何动态祖先中都没有发现 x 的声明，就报告一个运行时错误。

考虑上面例子中对 sub2 的两种不同的调用序列。第一种情况，big 调用 sub1，sub1 再调用 sub2。在这种情况下，搜索过程从局部过程 sub2 到它的调用者 sub1，而在 sub1 中找到了 x 的声明，所以，在这种情况下，在 sub2 中引用的 x 是在 sub1 中声明的 x。第二种情况是 big 直接调用 sub2。在这种情况下，sub2 的动态父辈是 big，因而引用的是在 big 中声明的 x。

注意，如果使用了静态作用域，则不管使用上述哪种调用序列，对 sub2 中 x 的引用都是引用 big 中的 x。

Perl 语言中的动态作用域与众不同，虽然它在语义上是传统的动态作用域(参见编程练习题 1)，但事实上却和本节讨论的不完全一样。

5.5.7　动态作用域的评价

动态作用域对于程序设计的影响是非常深刻的。对程序语句可见的非局部变量的正确属性不能静态地确定。而且，对这种变量名字的引用并不总是指向同一个变量。包含有非局部变量引用的子程序中的语句能在子程序的不同执行期间引用不同的非局部变量。有些

程序设计问题就是直接由动态作用域导致的。

首先，从子程序的执行开始到执行结束的期间，子程序中的局部变量对任何其他正在执行的子程序都是可见的，不管子程序代码之间的接近程度如何，也不管执行流程是如何执行到当前子过程的。对于这种可访问性，目前还没有任何办法可以保护局部变量。子程序总是在所有先前已调用且尚未结束执行的子程序环境中执行。结果，与静态作用域相比，动态作用域导致程序的可靠性更低。

动态作用域的第二个问题是不能进行非局部变量引用的静态类型检测。这个问题起源于不能静态地找到引用为非局部变量的声明。

动态作用域也使程序读起来更困难，因为必须知道子程序的调用顺序，才能确定对非局部变量引用的含义，这个任务对于人类读者而言实际上是不可能的。

最后，在动态作用域语言中，对非局部变量的存取比使用静态作用域时对非局部变量的存取所耗的时间多得多。第 10 章将给出原因。

另一方面，动态作用域也不是没有优点。在某些情况下，从一个子程序传递到另一个子程序的参数是在调用程序中定义的变量。在动态作用域语言中，这些并不需要传递，因为它们对于被调用的子程序是隐式可见的。

不难明白为什么动态作用域不如静态作用域使用广泛。使用静态作用域语言编写的程序不仅比使用动态作用域语言编写的等同程序更易读、更可靠，运行速度也更快。正是由于这样的原因，在 LISP 的最新方言版本中，用静态作用域取代了动态作用域。静态作用域及动态作用域的实现方法，将在第 10 章中讨论。

5.6 作用域和生存期

有时，变量的作用域与生存期似乎是有关联的。例如，在不包含方法调用的 Java 方法中声明的变量，它的作用域是从它的声明到方法的末尾，它的生存期则开始于方法开始执行时，终止于方法执行结束时。虽然变量的作用域与生存期很明显不是同一件事(静态作用域是文本或空间的概念，而生存期是时间的概念)，但它们至少在这种情形下看起来是有关联的。

作用域与生存期之间这种表面上的关系在其他情况下并不存在。例如，在 C 和 C++中，在函数中使用 static 修饰符声明的变量静态地绑定到该函数的作用域上，也静态地绑定到存储空间中。所以这个变量的作用域是静态的，是这个函数的局部变量，但它的生存期却扩展到了整个程序的执行期间。

调用子程序时，作用域和生存期也是不相关的。考虑下面的 C++函数：

```
void printheader() {
  ...
 } /* end of printheader */
void compute() {
  int sum;
```

```
    ...
    printheader();
} /* end of compute */
```

变量 sum 的作用域完全包含在函数 compute 中。尽管 printheader 在函数 compute 的运行中间执行，但 sum 的作用域却没有扩展到 printheader 函数体中。然而，sum 的生存期扩展到 printheader 的整个执行期间。在 printheader 调用之前，无论变量 sum 绑定到什么存储空间，这种绑定在 printheader 的执行期间及执行之后都将继续。

5.7　引用环境

语句的引用环境是指这条语句中所有可见变量的集合。在静态作用域语言中，语句的引用环境是在它的局部作用域中声明的变量，和在它的祖先作用域中声明的所有可见变量的集合。在这种语言中编译语句时，需要该语句的引用环境，才能产生代码和数据结构，以便在运行时引用来自其他作用域的变量。如何在静态作用域语言和动态作用域语言中实现非局部变量引用的技术，将在第 10 章中讨论。

在 Python 语言中，作用域可以通过函数定义来产生。语句的引用环境包括局部变量和包含这条语句的函数中声明的全部变量(不包括被较近的函数声明隐藏的非局部作用域中的变量)。每个函数定义产生一个新的作用域和由此而来的新环境。考虑下面的 Python 代码框架：

```
g = 3; # A global

def sub1():
    a = 5; # Creates a local
    b = 7; # Creates another local
    ... <-------------------------------1
  def sub2():
    global g; # Global g is now assignable here
    c = 9; # Creates a new local
    ... <-------------------------------2
    def sub3():
      nonlocal c: # Makes nonlocal c visible here
      g = 11; # Creates a new local
      ... <-------------------------------3
```

程序中各个标记点的引用环境如下：

标记点	引用环境
1	sub1 中的局部变量 a 和 b，全局变量 g 用于引用，而不是赋值
2	sub2 中的局部变量 c，全局变量 g 用于引用和赋值
3	sub2 中的非局部变量 c，sub3 中的局部变量 g

现在考虑这个代码框架中的变量声明。首先注意，虽然 sub1 的作用域层次(它的嵌套

比较浅)高于 sub3 的作用域层次，但是 sub1 的作用域却不是 sub3 的作用域的静态祖先，因此 sub3 不能访问 sub1 中声明的变量。其原因是，在 sub1 中声明的变量是栈动态的，因此这些变量在 sub1 没有执行之前，不会与存储空间绑定。又因为 sub1 没有执行之前 sub3 可能已经执行，所以 sub3 不能访问 sub1 中的变量，这些变量不一定在 sub3 的执行期间与存储空间绑定。

如果子程序开始执行，且该执行还没有终止，就称这个子程序是活跃的。在动态作用域语言中，语句的引用环境是局部声明的变量和当前活跃的其他子程序中的所有变量。活跃子程序中的一些变量可能对引用环境是不可见的。最近激活的子程序的变量声明会隐藏以前激活的子程序中的同名变量。

考虑下面的程序例子。假设其中仅有的函数调用是：main 调用 sub2，sub2 再调用 sub1。

```
void sub1() {
  int a, b;
  ... <-------------------- 1
} /* end of sub1 */
void sub2() {
  int b, c;
  ... <-------------------- 2
  sub1();
} /* end of sub2 */
void main() {
 int c, d;
 ... <-------------------- 3
 sub2();
} /* end of main */
```

程序中各个标记点的引用环境如下：

标记点	引用环境
1	sub1 中的 a 和 b，sub2 中的 c，main 中的 d(main 中的 c 和 sub2 中的 b 是隐藏的)
2	sub2 中的 b 和 c，main 中的 d(main 中的 c 是隐藏的)
3	main 中的 c 和 d

5.8 命名常量

命名常量只与值绑定一次，它有助于提高程序的可读性与可靠性。例如，使用名字 pi 来替代常量 3.14159265，就可以提高可读性。

命名常量的另一个重要用途是将程序参数化。例如，假定某程序要处理固定数目的数据值，比如 100。该程序通常会在多个位置使用常量 100，如声明数组下标的范围或循环控制次数。考虑下面的 Java 代码框架：

```
void example() {
  int[] intList = new int[100];
```

```
String[] strList = new String[100];
...
for (index = 0; index < 100; index++) {
...
}
...
for (index = 0; index < 100; index++) {
...
}
...
average = sum / 100;
...
}
```

若需要修改这个程序，以处理不同数目的数据，就必须找到所有的 100，并修改它。在大程序中，这个任务会很乏味、很容易出错。更容易、更可靠的方法是使用一个命名常量作为程序参数，如下：

```
void example() {
  final int len = 100;
  int[] intList = new int[len];
  String[] strList = new String[len];
  ...
  for (index = 0; index < len; index++) {
    ...
  }
  ...
  for (index = 0; index < len; index++) {
    ...
  }
  ...
  average = sum / len;
  ...
}
```

现在，不管这个数字在程序中使用多少次，当需要修改长度时，只需要修改一行代码(变量 len)。这是抽象所具有的优点的另一个例子。名字 len 是数组元素的数目和循环重复数目的抽象。这个例子说明了命名常量如何有助于程序的修改。

Ada 和 C++允许动态地将值绑定于命名常量。这样就允许含有变量的表达式对声明中的命名常量赋值。例如下面的 C++语句：

```
const int result = 2 * width + 1;
```

将 result 声明为整数类型的命名常量，它的值设为表达式 2 * width + 1 的值，给 result 分配存储空间并与其值绑定时，变量 width 的值必须是可见的。

Java 也允许将值动态地绑定于命名常量。在 Java 中，使用保留字 final 来定义命名常量(如前面的例子)。可以在声明语句中或在后面的赋值语句中将初始值赋给命名常量。可

以用任意的表达式来指明被赋的值。

C#有两种命名常量：用 const 定义的命名常量和用 readonly 定义的命名常量。const 命名常量是隐式静态的，它与值的绑定是静态的。也就是说，它们与值的绑定发生在编译时，这意味着绑定值只能通过字面量或其他 const 成员来指定。而 readonly 命名常量是动态与值绑定的，它们能在常量的声明或静态构造函数中赋值[12]。因此，如果程序需要一个常量值对象，这个对象在每一次使用程序时都有相同的值，就会用到 const 常量。然而，如果程序需要一个常量值对象，而该对象的值只有在创建对象时才能确定，而且在程序的不同执行情况下是相互不同的，此时就要用 readonly 常量。

Ada 允许使用枚举类型与结构类型的命名常量，参见第 6 章。

将值与命名常量绑定的讨论，很自然地引出了变量初始化这个主题，因为将值绑定到命名常量的过程和变量的初始化过程是相同的，只是前者是永久的。

在许多例子中，如果变量在声明它们的程序或子程序的代码开始执行前就有值，将是很方便的。变量在与存储空间绑定的同时也与值相绑定，就称为初始化。如果变量与存储空间的绑定是静态的，则绑定与初始化都发生在运行时之前。这种情况下，初始值必须指定为字面量或表达式，该表达式的非字面量操作数是已定义的命名常量。如果对存储空间的绑定是动态的，则初始化过程也是动态的，初始值可以是任意表达式。

在大多数语言中，初始化是在生成变量的声明中指定的。比如，在 C++中，有以下代码：

```
int sum = 0;
int* ptrSum = &sum;
char name[] = "George Washington Carver";
```

小　结

区分大小写以及名字与特殊字(保留字或关键字)的关系，是名字的设计问题。

变量能够用六种属性来刻画：名字、地址、值、类型、生存期和作用域。

别名是绑定于同一个存储地址的两个或多个名字。它们有损语言的可靠性，又很难从语言中完全消除。

绑定是属性和程序实体的关联。属性与实体的绑定时间对于理解程序设计语言的语义十分关键。绑定可以是静态的或动态的。显式声明或者隐式声明提供一种说明变量与类型的静态绑定的方式。一般而言，动态绑定有较大的灵活性，却牺牲了可读性、效率和可靠性。

根据标量变量的生存期，可以将它们分为四类：静态类、栈动态类、显式堆动态类、隐式堆动态类。

静态作用域是 ALGOL 60 及其大部分后继语言的一个主要特性。它提供了一种简单、可靠、有效的方法来提供非局部变量在子程序中的可见性。动态作用域比静态作用域提供了更多的灵活性，但以可读性、可靠性和效率为代价。

大多数函数式语言都允许用户用 let 结构创建局部作用域，限制了它们定义的名称的作

12. C#的静态构造函数在类实例化之前的某个不确定的时刻运行。

用域。

语句的引用环境是对该语句可见的所有变量的集合。

命名常量就是和值仅绑定一次的变量。

复 习 题

1. 名字的设计问题是什么?
2. 区分大小写的名字有什么潜在的危险?
3. 保留字在哪些方面比关键字好?
4. 什么是别名?
5. C++的哪一类引用变量总是别名?
6. 什么是变量的左值? 什么是变量的右值?
7. 定义绑定和绑定时间。
8. 在设计、实现语言后，在程序的哪四个时间能发生绑定?
9. 定义静态绑定和动态绑定。
10. 隐式声明的优点和缺点分别是什么?
11. 动态类型绑定的优点和缺点分别是什么?
12. 说明静态、栈动态、显式堆动态和隐式动态的变量的定义，并指出它们的优点和缺点。
13. 说明生存期、作用域、静态作用域和动态作用域的定义。
14. 在使用静态作用域的程序中，对非局部变量的引用是如何与其定义相关联的?
15. 静态作用域中的一般性问题是什么?
16. 什么是语句的引用环境?
17. 什么是子程序的静态祖先? 什么是子程序的动态祖先?
18. 程序块是什么?
19. 在函数式语言中，let 结构的作用是什么?
20. 用 ML 的 let 结构定义的名字和在 C 块中声明的变量有什么区别?
21. 描述如何把 F#的 let 封装在一个函数中，以及如何放在所有函数的外部。
22. 动态作用域的优点和缺点分别是什么?
23. 命名常量的优点是什么?

习　　题

1. 下列哪个标识符最易读? 给出理由。

```
SumOfSales
Sum_of_Sales
SUMOFSALES
```

2. 有些程序设计语言是无类型的。这种没有类型的语言有哪些明显的优点和缺点?

3. 用你知道的一种语言来编写一条包含一个算术操作符的简单赋值语句。对这条语句中的每个组成成分，列出在执行语句时用来确定语义的各种绑定。对每种绑定，指出相应的绑定时间。
4. 动态类型绑定和隐式堆动态变量的关系非常紧密。解释这种关系。
5. 描述一种情况，在这种情况下子程序中历史敏感变量是有用的。
6. 思考下面的 JavaScript 代码框架：

```
// The main program
var x;
function sub1() {
  var x;
  function sub2() {
    ...
  }
}
function sub3() {
   ...
}
```

假设这个程序的执行过程遵循下面的单元调用顺序：

```
main 调用 sub1
sub1 调用 sub2
sub2 调用 sub3
```

a. 假设使用静态作用域，其中哪一个 x 的声明对于下列 x 的引用是正确的?

```
i. sub 1
ii. sub2
iii. sub3
```

b. 重复 a 部分，但假设使用动态作用域。

7. 假设下面的 JavaScript 程序使用静态作用域规则进行解释。在函数 sub1 中将输出什么样的 x 值? 若使用动态作用域规则，在函数 sub1 中又将输出什么样的 x 值?

```
var x;
function sub1() {
  document.write("x = " + x + "<br />");
}
function sub2() {
  var x;
  x = 10;
  sub1();
}
x = 5;
sub2();
```

8. 考虑下面的 JavaScript 程序：

```
var x, y, z;
function sub1() {
  var a, y, z;
  function sub2() {
    var a, b, z;
    ...
  }
  ...
}
function sub3() {
  var a, x, w;
  ...
}
```

假设使用的是静态作用域，列出在 sub1、sub2 和 sub3 的函数体中可见的所有变量，以及声明这些变量的程序单元。

9. 考虑下面的 Python 程序：

```
x = 1;
y = 3;
z = 5;
def sub1():
  a = 7;
  y = 9;
  z = 11;
  ...
def sub2():
  global x;
  a = 13;
  x = 15;
  w = 17;
  ...
  def sub3():
    nonlocal a;
    a = 19;
    b = 21;
    z = 23;
    ...
...
```

假设使用的是静态作用域，列出在 sub1、sub2 和 sub3 的函数体中可见的所有变量，以及声明这些变量的程序单元。

10. 考虑下面的 C 程序：

```
void fun(void) {
  int a, b, c; /* definition 1 */
  ...
  while (...) {
    int b, c, d; /*definition 2 */
```

```
    ... <-------------1
    while (...) {
      int c, d, e; /* definition 3 */
      ... <-------------2
      }
    ... <-------------3
    }
  ... <-------------4
}
```

对这个函数中的每一个标记点，请列出每个可见的变量，以及定义这个变量的语句数目。

11. 考虑下面的 C 程序框架：

```
void fun1(void); /* prototype */
void fun2(void); /* prototype */
void fun3(void); /* prototype */
void main() {
  int a, b, c;
  ...
 }
void fun1(void) {
  int b, c, d;
  ...
 }
void fun2(void) {
  int c, d, e;
  ...
 }
void fun3(void) {
  int d, e, f;
  ...
 }
```

给定下面的调用序列，并假设使用的是动态作用域，在最后一个被调用的函数的执行期间，哪些变量是可见的?对于每一个可见变量，给出定义这些变量的函数名称。

a. main 调用 fun1；fun1 调用 fun2；fun2 调用 fun3。
b. main 调用 fun1；fun1 调用 fun3。
c. main 调用 fun2；fun2 调用 fun3；fun3 调用 fun1。
d. main 调用 fun3；fun3 调用 fun1。
e. main 调用 fun1；fun1 调用 fun3；fun3 调用 fun2。
f. main 调用 fun3；fun3 调用 fun2；fun2 调用 fun1。

12. 考虑下面用类似 JavaScript 语法编写的程序：

```
// main program
var x, y, z;

function sub1() {
  var a, y, z;
  ...
```

```
}
function sub2() {
  var a, b, z;
  ...
}
function sub3() {
  var a, x, w;
  ...
}
```

给定下面的调用序列，并假设使用的是动态作用域，在最后一个调用的子程序的执行期间，哪些变量是可见的?对于每一个可见变量，给出定义这些变量的单元名称。

a. main 调用 sub1；sub1 调用 sub2；sub2 调用 sub3。
b. main 调用 sub1；sub1 调用 sub3。
c. main 调用 sub2；sub2 调用 sub3；sub3 调用 sub1。
d. main 调用 sub3；sub3 调用 sub1。
e. main 调用 sub1；sub1 调用 sub3；sub3 调用 sub2。
f. main 调用 sub3；sub3 调用 sub2；sub2 调用 sub1。

编程练习

1. Perl 既允许静态作用域，也允许某种动态作用域。写一个使用这两种作用域的 Perl 程序，清晰展现它们在效果上的差别。清楚地解释本章描述的动态作用域和 Perl 语言实现的动态作用域的差别。
2. 写一个 Common LISP 程序，要求清晰地说明静态和动态作用域之间的不同点。
3. 写一个 JavaScript 脚本程序，它有三层嵌套深度的子程序，每个嵌套的子程序要引用所有嵌套它的子程序中的变量。
4. 用 Python 完成第 3 题。
5. 写一个包含以下语句序列的 C 函数：

```
x=21;
int x;
x=42;
```

运行这个程序并解释它的结果。以 C++和 Java 重写相同的代码，并比较它们的结果。

6. 用 C++、Java 和 C#写一个测试程序，确定 for 语句中声明的变量的作用域。特别地，代码必须确定在 for 语句循环体之后该变量是否可见。
7. 用 C 或 C++写 3 个函数：一个声明静态的大数组，一个在栈上声明同样的大数组，还有一个从堆中生成相同的大数组。对每个子程序调用很多次(最少 100 000 次)，输出每个子程序所需的时间。解释结果。

第6章

数据类型

6.1 概述
6.2 基本数据类型
6.3 字符串类型
6.4 用户定义的序数类型
6.5 数组类型
6.6 关联数组
6.7 记录类型
6.8 元组类型
6.9 列表类型
6.10 联合类型
6.11 指针和引用类型
6.12 类型检查
6.13 强类型化
6.14 类型等价
6.15 理论和数据类型

本章首先介绍数据类型的概念以及常用基本数据类型的特征，然后讨论枚举和子范围类型，接着将学习结构化数据类型的细节，特别是数组、关联数组、记录、元组、列表和联合。之后是对指针和引用的深入研究。

对于每种数据类型，我们都将阐述它的设计问题，描述某些常用语言的设计人员所做出的设计选择，并评价这些设计。

接下来三节全面探讨类型检查、强类型化和类型等价规则。本章最后一节简要介绍数据类型理论的基础知识。

数据类型的实现方式有时极大地影响着它们的设计。因此，各种数据类型的实现(特别是数组的实现)，是本章的另一个重要部分。

6.1 概述

数据类型定义了一组数据值，以及在这些数据值上预先定义的一组操作。计算机程序通过操作数据产生结果。决定程序完成任务的难易程度的一个重要因素，是正在使用的语言中已有的数据类型与要解决的现实问题的匹配程度。因此，语言支持合适种类的数据类型与结构就成为关键。

数据类型化的现代概念是历经过去 55 年的演化形成的。在最早期的语言中，所有问题空间中的数据结构，都不得不用语言支持的少数几种基本的数据结构来建模。例如，在 Fortran 90 之前，链表和二叉树是用数组来实现的。

COBOL 的数据结构迈出了脱离 Fortran I 模型的第一步，它允许程序员指定小数的精度，还为信息记录提供了一种结构化的数据类型。PL/I 将指定精度的功能扩展到整数和浮点数。这种功能从此便引入到 Ada 和 Fortran 中。PL/I 的设计者引入大量的数据类型，目的是支持大范围的应用。ALGOL 68 引入了一种更好的方式，就是仅提供少数的基本类型和少量灵活的结构定义操作符，让程序员为每一种需求设计数据结构。显然，这是数据类型设计的发展进程中最重要的进步之一。用户定义的类型允许给类型取有意义的名字，也增强了程序的可读性。它们允许对有特殊用途的变量进行类型检查，这一点如果没有用户定义的类型就是不可能的。用户定义的类型也有利于可修改性：程序员只需要改变类型的定义语句，就能改变程序中一种变量的类型。

从用户定义的类型这个概念再进一步，就到达了抽象数据类型，大多数 20 世纪 80 年代中期以后设计的编程语言都支持抽象数据类型。抽象数据类型的基本思想是分离类型的接口(它对用户是可见的)与该类型的数据表示和操作集(这两者对用户是隐藏的)。高级编程语言提供的所有类型都是抽象数据类型。用户自定义的抽象数据类型在第 11 章详细讨论。

编程语言的类型系统有很多用途，最实用的就是错误检查。在语言类型系统的指导下，类型检查的过程与价值将在 6.12 节中讨论。第二项用途是类型系统能为程序模块化提供帮助。这种帮助来自于跨模块的类型检查，该检查保证了模块之间接口的一致性。类型系统的另一个用途是文档化。程序文档信息中关于数据的类型声明提供了关于程序

行为的线索。

编程语言的类型系统定义了类型如何与语言中的各种表达式建立联系，包括类型等价和类型兼容的规则。当然，理解编程语言的语义最重要的部分之一就是理解它的类型系统。

尽管最近几年关联数组的流行程度已经显著上升，数组和记录仍然是命令式语言中两种最常见的结构化(非标量)数据类型。自从第一种函数式编程语言(LISP)第一次在 1959 年推出后，列表就是这类语言的一个主要部分。在过去 10 年中，由于函数式编程的日益流行，使列表也添加到主要命令式语言中，例如 Python 和 C#。

结构化数据类型由类型操作符或构造符来定义，用于形成类型表达式。例如，C 语言用方括号和星号作为类型操作符，来指明数组和指针。

用描述符这个术语来描述变量，无论是从逻辑角度看还是从实用角度看，都是很方便的。描述符是变量的一组属性集合。在实现中，描述符是存储变量属性的一个内存区域。如果这些属性都是静态的，那么这些描述符只在编译期间才需要。它们由编译器创建，通常是作为符号表的一部分，并用于编译期间。然而，对于动态属性，描述符的一部分或全部则必须在执行期间维护。在这种情况下，描述符由运行时系统使用。在所有情况下，描述符都用于类型检查，以及生成内存分配和释放操作的代码。

使用术语“变量”时要小心。只使用传统的命令式语言的人可能把标识符看作变量，但在考虑数据类型时，这可能带来混淆。在一些编程语言中，标识符没有数据类型，所以最好记住，标识符只是变量的一个属性。

对象这个词常和变量的值以及变量占的空间联系在一起。不过，在本书中，对象只是用户定义的抽象数据类型的实例，而不是预定义类型的变量值。在面向对象语言中，每个类(无论是预定义的还是用户自定义的)的实例，都称为对象。第 11 章和第 12 章将详细讨论对象。

本章后续各节将讨论许多常见的数据类型。对大多数数据类型，我们将阐明该类型特有的设计问题。对所有数据类型，我们都将列举一个或多个设计示例。有一个设计问题对所有数据类型都是最基本的：应该为类型的变量提供哪些操作？如何说明这些操作？

6.2　基本数据类型

不用其他数据类型来定义的数据类型称为基本数据类型。几乎所有编程语言都会提供一组基本数据类型。有些基本数据类型仅仅是硬件的反映，例如大多数的整数类型。其他数据类型则要求在实现上有少量的非硬件支持。

编程语言中的基本数据类型与一个或多个类型构造符一起，用于提供结构化类型。

6.2.1　数值类型

很多早期的编程语言只有基本的数值类型。在现代编程语言支持的类型中，数值类型仍扮演着中心角色。

6.2.1.1 整数

最常见的基本数值数据类型是整型。现在很多计算机都支持好几种大小的整数类型。有些编程语言支持不同大小的整数类型。例如，Java 包含 4 种有符号的整数：byte、short、int 和 long。有些语言，如 C++和 C#，则包括了无符号的整数类型，它们只不过是没有正负号的整数值类型。无符号类型通常用于二进制数据。

在计算机中，有符号的整数值是由一串比特位表示的，其中一位(通常是最左边的一位)表示符号。大多数整数类型是硬件直接支持的。硬件不直接支持的一种整数类型是 Python 中的 long integer 类型(F#也提供这种整数)。这个类型的值没有长度限制。long integer 的值能定义字面量，如下面的例子：

```
243725839182756281923L
```

Python 中的整数算术运算会产生过大的整数值，不能用 int 类型表示，这些值会存储为 long integer 类型。

可用符号-数值记法来储存负整数，在这种记法中，符号位用于表示负值，其他位表示数值的绝对值。不过，符号-数值记法并没有用于计算机的算术运算中。现在大多数计算机使用二进制补码记法来储存负整数，该方法在进行加减时很方便。在二进制补码记法中，负整数的表示是由它对应的正整数求逻辑反，再加 1 来得到的。但有些计算机仍使用二进制反码记法来表示负整数。在二进制反码记法中，负整数储存为其绝对值的逻辑反码。二进制反码记法有一个缺点，整数 0 有两种表示形式。关于整数表示方法的细节信息，请参阅任何一本关于汇编语言程序设计的书籍。

6.2.1.2 浮点数

浮点数据类型模拟实数，但浮点数表示的只是大多数实数值的近似值。例如，基本数 π 或 e(自然对数的底)都不能用浮点表示法准确地表示。当然，这两个数都不能在任何有限的空间中精确地表示出来。在大多数计算机中，浮点数是以二进制存储的，这使问题变得更糟。例如，即便是十进制的 0.1 也不能用有限的二进制位来表示[1]。浮点类型的另一个问题是算术运算中的精度损失。关于浮点表示法的详细信息，请参阅任何一本关于数值分析的书。

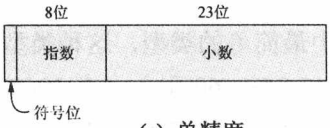
浮点数表示成小数和指数，这是借鉴科学记数法的形式。早期的计算机使用了多种不同的浮点数表示方式。不过，大多数最新的机器使用 IEEE 浮点标准 754 格式。语言的实现人员采用硬件支持的表示形式。大多数语言都包括两种浮点类型，通常称为 float 和 double。float 类型是标准大小，通常存储在四个内存字节中。当需要更大的小数部分和/或更大范围的指数时，就使用 double 类型。double 变量占用的存储空间通常是 float 变量的两倍，并提供至少两倍的小数位数。

精度和范围定义了能用浮点类型表示的数值集合。精度是数值小数部分的准确性，用小数位数来衡量。范围则是指小数范围和指数范围的组合，在这里，指数范围更重要。

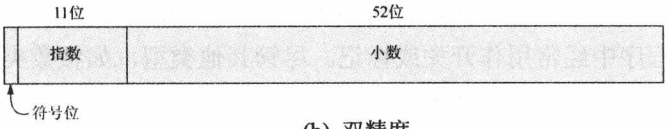
---

1. 十进制的 0.1 等于二进制的 0.0001100110011 …。

图 6-1 显示了 IEEE 浮点标准 754 中单精度和双精度的表示格式(IEEE，1985)。IEEE 格式的详细资料可在参考文献 Tanenbaum(2005)中找到。



(a) 单精度



(b) 双精度

图 6-1　IEEE 浮点数格式

6.2.1.3　复数

有些编程语言支持复数数据类型，例如，Fortran 和 Python。复数值用浮点值的有序对来表示。在 Python 中，复数字面量的虚部用后接的 j 或 J 来指定——例如

```
(7 +3j)
```

支持复数类型的语言包含复数值的算术运算。

6.2.1.4　小数

大多数为支持商业系统应用而设计的大型计算机都为小数数据类型提供硬件支持。小数数据类型存储固定数目的小数位，小数点位于数值中的一个固定位置上。这些类型是商业数据处理中的基本数据类型，因而成为 COBOL 的重要组成部分。C#和 F#也有小数数据类型。

小数类型的优势在于它至少能在有限的范围内精确存储小数数值，这是浮点数做不到的。例如，数值 0.1(十进制)能用小数类型精确地表示，但用浮点数类型就不行，如 6.2.1.2 节所述。小数类型的缺点是：由于不允许使用指数，所以它们表示的数值范围是受限的；它们在内存中的表示很浪费空间，其原因在接下来的段落中加以解释。

小数类型的存储非常像字符串，使用二进制码来表示十进制数字。这些表示方法称为二进制编码的十进制数(binary coded decimal，BCD)。在某些情况下，每个字节存储一位数字，但在另外一些情况下，每个字节存储两位数字。无论哪种方式，它们都比二进制表示占用更多的存储空间。至少需要 4 个二进制位来编码一个十进制数字。因此，要存储一个 6 位的十进制数，就需要 24 个二进制位的内存空间。而以二进制方式存储相同的数只需要 20 个二进制位[2]。在具有这种功能的计算机上，小数值的操作是由硬件来完成的，而在没

2. 当然，除非程序需要维护大量的较大的十进制小数值，否则，上述这些存储差别并不明显。

有该功能的计算机上，是用软件来模拟的。

6.2.2 布尔类型

布尔类型或许是所有类型中最简单的类型，这种类型的取值范围只有两个元素：一个为真，一个为假。布尔类型由 ALGOL 60 引入，在 1960 年以后设计的大多数通用语言都包含了布尔类型。一个众所周知的例外是 C89，它的数值表达式能用作条件。在这些表达式中，所有非零操作数都为真，而零就是假。尽管 C99 和 C++都有布尔类型，但它们也允许把数值表达式用作布尔类型。在随后的 Java 和 C#语言中是不允许这样的。

布尔类型在程序中经常用作开关或标记。尽管其他类型，如整数类型，也可达到同样的目的，但使用布尔类型的可读性更好。

布尔值可以用单个二进制位来表示，但因为在很多计算机上不能有效地访问内存的单个二进制位，所以它们经常存储在能有效寻址的最小内存单元中，通常是一个字节。

6.2.3 字符类型

字符数据以数值编码的形式存储在计算机中。传统上，最广泛应用的编码是 8 位编码的 ASCII(American Standard Code for Information Interchange)，它使用 0～127 的数值来编码 128 个不同的字符。ISO 8859-1 是另一种 8 位字符编码，但它允许编码 256 个不同的字符。Ada 95+就应用了 ISO 8859-1。

由于经济全球化，以及世界范围内计算机之间交流的需要，ASCII 字符集变得不够用了。为此，1991 年统一字符标准协会(Unicode Consortium)发布了 UCS-2 标准，这是一个 16 位的字符集。该字符编码通常称作 Unicode。Unicode 包括了世界上大多数自然语言中的字符。例如，Unicode 包括了塞尔维亚语的 Cyrillic 字母，以及泰语的数字。Unicode 的前 128 个字符与 ASCII 相同。Java 是第一个使用 Unicode 字符集的广泛应用的语言。此后，Unicode 也进入到 JavaScript、Python、Perl、C#和 F#中。

1991 年以后，统一字符标准协会与国际标准组织(ISO)合作，开发了一个 4 字节的字符编码，称为 UCS-4 或 UTF-32，在 2000 年发布的 ISO/IEC 10646 标准中描述了该标准。

为提供处理单个字符编码的方法，多数编程语言都包括了一个基本数据类型。然而，Python 支持的单个字符只是长度为 1 的字符串。

6.3 字符串类型

字符串类型的值由字符序列组成。字符串常量用于标识输出，各类数据的输入和输出常以字符串的形式来完成。当然，对于所有进行字符处理的程序来说，字符串也是一种基本数据类型。

6.3.1　设计问题

字符串类型特有的两个最重要设计问题是：

- 字符串应该是一种特殊的字符数组，还是一种基本类型？
- 字符串应该具有静态长度，还是动态长度？

6.3.2　字符串及其操作

最常用的字符串操作包括赋值、拼接、子串引用、比较以及模式匹配。

子串引用是对给定字符串的一个子串的引用。在讨论子串引用时，通常是以数组为上下文的，此时子串引用称为切片(slice)。

字符串的赋值操作和比较操作通常比较复杂，因为两个操作数可能具有不同的长度。比如说，把较长的字符串赋给较短的字符串时，会发生什么呢？反之，又会怎样呢？通常，针对这些情况，选择的依据是简单、合理，但程序员们常常很难记住它们。

模式匹配是另一种基本的字符串操作。在有些语言中，模式匹配由语言直接支持。而在其他语言中，模式匹配是由函数或者类库来提供的。

如果字符串没有定义为基本类型，则字符串数据通常储存在字符数组中，并以字符数组的形式引用。这就是 C 和 C++采用的方法。

C 和 C++采用 char 数组来存储字符串。这些语言通过标准库提供了一组字符串操作。大多数字符串的应用和部分库函数都遵循一个约定，即字符串由一个特殊的字符 null 来终止，null 用 0 表示。这是维护字符串变量长度的一种替代方法。库函数会一直执行它们的操作，直至 null 字符出现在被处理的字符串中。生成字符串的库函数常常在结果中提供了 null 字符。由编译器生成的字符串文字量也包含 null 字符。例如下面的声明：

```
char str[] = "apples";
```

在这个例子中，str 是一个元素为 char 的数组，其内容是 apples0，其中 0 代表 null 字符。

在 C 和 C++中，最常用的字符串库函数有：strcpy，用于移动字符串；strcat，用于把一个字符串连接到另一个字符串上；strcmp，用于按字母表顺序(按字符代码的顺序)比较两个指定的字符串；以及 strlen，用于返回给定字符串的字符数目(不算 null 字符)。大多数字符串操作函数的参数和返回值都是 char 指针，指向 char 数组。参数也可以是字符串字面量。

C 标准库中的字符串操作函数本质上是不安全的，并已造成了无数编程错误。C++的标准库也提供了它们。其问题是，该库中的函数移动了字符串数据，却不防止数据超出目标字符串的范围。例如，考虑下面的 strcpy 调用：

```
strcpy(dest, src);
```

如果 dest 的长度是 20，src 的长度是 50，strcpy 将在 dest 后面多写出 30 个字节。问题是 strcpy 并不知道 dest 的长度，所以不能保证 dest 后面的内存不会改写。C 字符串库中的其他一些函数也会发生同样的问题。除 C 风格的字符串外，C++还通过标准类库支持字符

串操作，C++的标准类库也与 Java 中的相似。由于 C 字符串库的不安全性，C++程序员应该用标准库中的 string 类，来替代 char 数组以及 C 的字符串库。

历史注释

SNOBOL 4 是第一个广为人知的支持模式匹配的语言。

在 Java 中，字符串由 String 类(值是常量字符串)和 StringBuffer 类(值是可变的，更像字符数组)来支持。字符串的值通过 StringBuffer 类的方法来指定。C#和 Ruby 包含的字符串类与 Java 相似。

Python 也把字符串作为基本类型，并具有子串引用、拼接、单个字符索引访问等操作，以及进行查找和替换操作的方法。字符串也有字符成员操作。所以，虽然 Python 的字符串是基本类型，但对于字符和子串的引用，它们非常像字符数组。然而，Python 字符串是不可变的，这类似于 Java 中的 String 类对象。

在 F#中，字符串是一个类。单个字符用 Unicode UTF-16 表示，可以访问，但不能改变。字符串可用+操作符来拼接。在 ML 中，字符串是一个不可变的基本类型。它使用^作为拼接操作符，拥有子串引用和确定字符串长度的函数。

Perl、JavaScript、Ruby 和 PHP 包含内置的模式匹配操作。在这些语言中，模式匹配表达式在一定程度上近似基于数学表达式。事实上，它们常常称为正则表达式。它们从早期的 UNIX 行编辑器 ed 发展而来，已成为了 UNIX shell 语言的一部分。最终，它们演变为目前的复杂形式。要讲解这种模式匹配表达式，至少需要一整本书(Friedl，2006)。而本节只想通过列举两个相对简单的例子，来简要介绍这些表达式的风格。

考虑下面的模式表达式：

```
/[A-Za-z][A-Za-z\d]+/
```

这个模式匹配(或描述)程序语言中的典型名字形式。方括号括起来的是字符类。第一个字符类代表所有字母；第二个代表所有字母和数字(缩写\d 表示一个数字)。如果只包含第二个字符类，则不能阻止名字以数字开头。跟在第二个类别后面的加法操作符说明必须有一个或者更多该类别中的元素。所以，与整个模式匹配的字符串以一个字母开头，后跟一个或多个字母或数字。

考虑以下模式表达式：

```
/\d+\.? \d*|\.\d+/
```

这个模式匹配数值字面量。\.代表字面常量的小数点[3]。问号表示它后面的内容出现零次或一次。竖线(|)分开了上述模式的两种可能选择。第一种选择匹配的字符串有一个或多个数字，可能后面接一个小数点，然后是零个或多个数字；第二种选择匹配的字符串以小数点开头，后面有一个或多个数字。

---

3. 句点必须用反斜杠来转义，因为句点在正则表达式中具有特殊的含义。

在 C++、Java、Python、C#和 F#的类库中包含了用正则表达式匹配模式的功能。

6.3.3　字符串长度的设计选项

关于字符串的长度，有几种设计选择。第一种选择，长度是静态的，在创建字符串时设置。这种字符串称作静态长度字符串。选择这种字符串的有 Python 中的字符串、Java 语言中 String 类的不可变对象、C++标准类库中相似的类、Ruby 的内置 String 类，以及 C# 和 F#中的.NET 类库。

第二种选择是允许字符串具有可变的长度，其上限是由变量定义所设定的一个固定的最大值，例如 C 中的字符串和 C++中 C 风格的字符串。这种字符串称为动态限长字符串。这种字符串变量能存储从 0 到上限值之间任意数目的字符。C 中的字符串用一个特殊字符来表示字符串的结尾，而不是维护字符串的长度。

第三种选择，是允许字符串具有可变的长度，且不设上限，JavaScript、Perl 和 C++标准库就是这样设计的。这种字符串称为动态长度字符串。这种选择需要一些额外开销来进行动态存储分配和回收，但提供了最大限度的灵活性。

Ada 95+支持上述三种字符串长度的设计选择。

6.3.4　评价

字符串类型对语言的可写性很重要。把字符串当作数组来处理，比作为基本的字符串类型来处理更加不便。例如，假设某种语言将字符串当作字符数组，而且没有预定义的函数来实现 C 中 strcpy 的功能，那么，字符串的简单赋值操作就需要一个循环来完成。将字符串作为基本类型加到语言中，无论是从语言本身还是编译器的复杂度看，代价都不高，所以，当代一些语言没有把字符串作为基本类型，实在说不上有什么道理。当然，通过标准库来提供字符串类型和作为基本类型来提供，便利性是差不多的。

字符串的操作(例如简单模式匹配和字符串连接)非常重要，应该包括在语言中，用来处理字符串类型的值。虽然动态长度的字符串明显是最灵活的，但是需要在这种灵活性和实现它的开销之间进行权衡。

6.3.5　字符串类型的实现

字符串类型可以直接由硬件提供支持，但很多情况下是用软件来实现对字符串的存储、检索和处理。当某种语言用字符数组来实现字符串类型时，它提供的字符串操作通常比较少。

静态字符串类型的描述符有三个字段，它只在编译过程中需要用到。每个描述符的第一个字段是类型的名称。对于静态字符串而言，描述符的第二个字段是它的长度(字符数目)。第三个字段是首字符的地址。这种描述符如图 6-2 所示。动态限长字符串需要一个运行时描述符来存储长度的上限值及当前值，如图 6-3 所示。动态长度字符串需要的运行时描述符更简单，因为只需要存储字符串的当前长度。虽然这里把描述符说成是一块独立的内存，但在大多数情况下它们是存储在符号表中的。

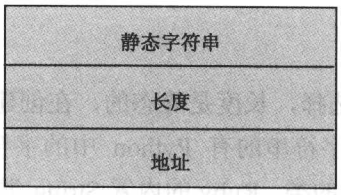


图 6-2 静态字符串的编译时描述符

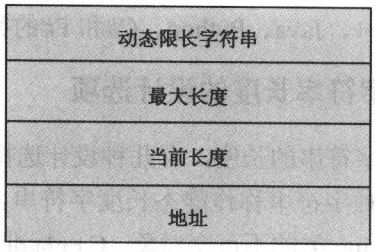


图 6-3 动态限长字符串的运行时描述符

C 和 C++中的动态限长字符串不需要运行时描述符，因为这种字符串用 null 字符来标记结尾。它们也不需要最大长度信息，因为在这两种语言中，没有对数组下标的值进行范围检查。

静态长度和动态限长字符串不要求特殊的动态存储分配。对于动态限长字符串，当字符串变量与存储空间绑定时，就已经为最大长度分配了足够的存储空间，所以，只需要调用一次内存分配过程。

动态长度字符串则需要更复杂的存储管理。字符串的长度以及字符串绑定的存储空间，都必须动态地增长和缩小。

有三种方法支持动态长度字符串所需的动态分配和释放。第一种，字符串存储在链表里，这样，当字符串增长时，需要的新存储单元能从堆中获得。这种方法的缺点是：实现链表时，链接指针项要占额外的存储空间，以及字符串操作具有不可避免的复杂性。

第二种方法是，将字符串以指针数组的形式存储，数组元素指向在堆中分配的单个字符。这种方法虽然会占用额外空间，但字符串的处理要比链表方法更快。

第三种方法是，将完整的字符串存在相邻的存储空间中。当字符增长时，这种方法的问题是：如何把与已有单元相邻的存储空间继续分配给字符串变量呢？很多时候，这些存储空间是不可用的。替代办法是：在内存中找一个能容纳完整新串的新区域，把旧字符串移到新区域，然后释放旧字符串占用的内存单元。后面这种方法比较常用。6.11.8.3 节将讨论可变大小存储段的分配与释放的一般问题。

尽管链表方法需要更多的存储空间，但相关的分配和释放过程却很简单。然而，一些字符串操作因为需要指针追踪而变慢。另一方面，将完整的字符串存储在相邻的存储空间中，字符串操作会较快，需要的存储空间也显著减少，但内存的分配和释放过程会慢一些。

6.4 用户定义的序数类型

序数类型可能的取值范围很容易与一组正整数相关联。例如，在 Java 中，基本序数类型有 integer、char 和 boolean。编程语言支持两种用户定义的序数类型：枚举类型和子范围类型。

6.4.1　枚举类型

枚举类型的所有可能值都是在类型定义时提供或枚举出来的命名常量。枚举类型提供了定义和组合命名常量的一种方法，这些命名常量称为枚举常量。下面的 C#示例是枚举类型的一种典型定义方式：

```
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

通常，枚举常量隐式地赋予整型值 0，1，…，也能在类型的定义中显式地赋予任何整型字面常量。

枚举类型的设计问题有：

- 是否允许枚举常量出现在多个类型定义中？如果允许，程序中出现该常量时，如何对它进行类型检查？
- 枚举值会自动强制转换成整数吗？
- 其他类型会自动强制转换成枚举类型吗？

所有这些设计问题都与类型检查有关。如果枚举变量会自动强制转换成数值类型，就难以控制其合法操作范围或取值范围。如果 int 类型的值会自动强制转换成枚举类型，就可以将任何整数值赋给枚举类型的变量，而不论该数值是否表示枚举常量。

6.4.1.1　设计

在没有枚举类型的语言中，程序员常常用整数值来模拟枚举类型。例如，假设需要在 C 语言程序中表示颜色，但 C 语言没有枚举类型，就可以用 0 表示蓝色，1 表示红色，以此类推。这些值的定义如下所示：

```
int red = 0, blue = 1;
```

现在，可在程序中把 red 和 blue 当做颜色类型来使用。这个方法存在的问题是：因为没有为颜色定义一个类型，所以使用它们时就不会有类型检查。例如，把两个颜色加起来也是合法的，尽管这不是我们期望的操作。也可用算术操作符将颜色类型与其他数值类型结合在一起，尽管这没什么用。此外，由于 red 和 blue 实质上是整型变量，所以可以赋给它们任意的整数值，这样就破坏了它们与颜色之间的指代关系。把 red 和 blue 定义为命名常量，就可以解决最后这个问题。

C 和 Pascal 是最早含有枚举类型的广泛使用的语言。C++包括了 C 中的枚举类型，在 C++中，可以写如下代码：

```
enum colors {red, blue, green, yellow, black};
colors myColor = blue, yourColor = red;
```

尽管枚举常量能赋予任意整型文字量(或常量值的表达式)，但上面的 colors 类型使用了默认的内部值(即 0，1，…)作为枚举常量值。枚举值置于整数上下文中时，会自动强制转换成 int 类型。这就允许在数值表达式中使用枚举值。例如，如果 myColor 当前的值是 blue，

下面的表达式：

```
myColor++
```

会使 myColor 的值变为 green。

C++也允许将枚举常量赋给任意数值类型的变量，不过，这可能是一个错误。但在 C++中，其他类型的值都不会自动强制转换成枚举类型。例如：

```
myColor=4;
```

在 C++中是非法的。如果将等号右边显式地转换成 colors 类型，这个赋值就是合法的。这种规定能防止一些潜在的错误。

在同一个引用环境中，C++枚举常量只能出现在一种枚举类型中。

Ada 中的枚举字面值可以出现在同一个引用环境的多个声明中。这些枚举字面值称作重载字面值。重载的解析规则(即判定这种字面值是什么类型)是：类型必须能从字面值所在的上下文中推断出来。例如，如果比较重载字面值和枚举变量，那么这个字面值的类型就解析为枚举变量的类型。某些情况下，程序员必须为重载字面值给出类型说明，以避免编译错误。

在 Ada 中，由于枚举字面值和枚举变量都不会自动强制转换为整数值，所以枚举类型的操作范围和取值范围都是受限的，这样编译器就能检测出程序员的许多错误。

2004 年，枚举类型加入到 Java 语言的 Java 5.0 版本中。Java 中所有的枚举类型都隐式地作为预定义 Enum 类的子类。因为枚举类型都是类，所以它们可以有实例数据域、构造函数和方法。从语法上讲，除了能够包含数据域、构造函数和方法外，Java 枚举类型的定义类似于 C++。枚举变量可能的取值就是这个类可能的实例。所有枚举类型都继承了 toString 方法以及其他一些方法。使用静态方法 values，可得到由枚举类型的实例构成的数组；使用 ordinal 方法能得到枚举变量的内部数字值。任何其他类型的表达式都不能赋给枚举变量。同样，枚举变量也永远不会自动强制转换成任何其他类型。

除了从不自动强制转换为整数，C#的枚举类型和 C++的很类似。因此，对枚举类型的操作都限定在有意义的操作上。同样，取值范围也限定在特定枚举类型的取值范围上。

在 ML 中，使用 datatype 声明将枚举类型定义为新类型。例如，可以编写下面的代码：

```
datatype weekdays = Monday | Tuesday | Wednesday |
Thursday | Friday
```

weekdays 元素的类型是整数。

F#的枚举类型类似于 ML，但使用保留字 type 替代 datatype，且第一个值放在 OR 操作符(|)的后面。

有趣的是，没有哪个较新的脚本类语言包含枚举类型。这些语言包括 Perl、JavaScript、PHP、Python、Ruby 和 Lua。甚至 Java 也是在推出十年后才加入枚举类型的。

6.4.1.2　评价

枚举类型能提高可读性和可靠性。可读性是以一种很直接的方式增强的：命名的值很容易辨认，而编码的值则不是。

在可靠性方面，Ada、C#、F#和 Java 5.0 的枚举类型有两个优点。第一，对于枚举类型，所有的算术操作都是不合法的。比如，把一个星期中的若干天加起来的操作就是不允许的。第二，枚举变量不能赋值为超出定义范围的值[4]。如果 colors 枚举类型有 10 个枚举常量，并使用 0 到 9 作为它的内部值，就不能将大于 9 的数值赋给 colors 类型的变量。

由于 C 把枚举类型变量当做整型变量，所以它不具备上述两个优点。

比起 C 来，C++要好一些。在 C++中，要把数值赋给枚举类型的变量，必须能将枚举类型的变量转换成所赋值变量的类型。把数值赋给枚举类型变量时需要进行检查，以确定该数值是否在枚举类型的内部值范围内。可惜，如果用户以显式赋值方式使用了一个较宽的取值范围，这种检查就会失效。例如：

```
enum colors {red = 1, blue = 1000, green = 100000}
```

这个例子给 colors 类型的变量赋值时，只会检查该数值是否在 1 到 100 000 的范围内。

6.4.2　子界类型

子界类型是有序类型的连续序列。例如，12..14 是整数类型的子界类型。子界类型由 Pascal 引入，Ada 也包含子界类型。子界类型没有相关的设计问题。

6.4.2.1　Ada 语言的设计

在 Ada 中，子界属于子类型。第 5 章提到，子类型并不是新类型，而是对已有类型进行限定或约束后的新名称。例如，考虑下面的声明：

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
subtype Weekdays is Days range Mon..Fri;
subtype Index is Integer range 1..100;
```

在这些例子中，对已有类型的限定是限制了可能取值的范围。所有为父类型定义的操作都适用于子类型，但不能用超出规定范围的值进行赋值操作。例如：

```
Day1 : Days;
Day2 : Weekdays;
...
Day2 := Day1;
```

只要 Day1 的值不是 Sat 或 Sun，上面的赋值操作就是合法的。

对子界变量的每一个赋值操作，编译器都必须生成范围检查代码。类型的兼容性检查

---

4. 在 C#和 F#中，整数值可以显式强制转换为枚举类型，并赋予枚举变量名。这种值必须用 Enum.IsDefined 方法检测，之后才能把它们赋予枚举变量名。

在编译时进行，而子界类型的范围检查则在运行时进行。

对于用户定义的有序类型，一个最常见的应用是作为数组索引，详见第 6.5 节。它们也可以用作循环变量。实际上，有序类型的子界类型是 Ada 语言中为 for 循环变量指定范围的唯一途径。

6.4.2.2 评价

子类型变量只能存储特定范围的值，所以子界类型增强了程序的可读性。使用子界类型还能提高程序的可靠性，因为如果赋给子界变量的值超出了指定的范围，编译器(当赋给变量的值是一个文字常量时)或运行时系统(当赋给变量的是一个变量或表达式时)就会把这个操作检测为错误。奇怪的是，除 Ada 外，现代语言都没有子界类型。

6.4.3 实现用户定义的有序类型

如前所述，枚举类型通常实现为整数。因为没有对取值范围和操作进行限制，所以这不能提高程序的可靠性。

子界类型的实现方式与父类型是完全相同的，但是在将变量或表达式赋给子界变量时，编译器必须隐式地包含范围检查，这个步骤增加了代码的大小和执行时间，但通常是值得的。此外，优化良好的编译器能够优化掉一些范围检查。

6.5 数组类型

数组是数据元素的一种聚合形式，数组中的元素通过它在聚合体中相对于第一个元素的位置来标识的。数组的各个数据元素都是同一类型。对数组元素的引用是通过下标表达式来指定的。如果引用数组元素的下标表达式中包含了变量，则需要在运行期间进行额外的计算，来确定被引用的数组元素的内存地址。

在许多语言中，如 C、C++、Java、Ada 和 C#，数组的所有元素都必须是同一类型。在这些语言中，指针和引用只能指向或引用一种类型，因而被指向或引用的对象或数据值也是单一类型的。在其他一些语言中，如 JavaScript、Python 和 Ruby，变量是对对象或者数据值的无类型引用。在这些情形下，数组仍然由单一类型的元素组成，但是元素可以引用不同类型的对象或数据值。这种数组仍然是同质的，因为它的元素都是同类型的。

C#和 Java 5.0 提供了泛型数组，这种数组的元素通过其类库来引用对象，参见 6.5.3 节。

6.5.1 设计问题

数组的主要设计问题有：

- 哪些类型对于下标来说是合法的？
- 要对元素引用中的下标表达式检查其范围吗？
- 何时绑定下标范围？

- 何时进行数组存储空间的分配？
- 支持不规则数组或多维矩阵数组吗？或者两个都支持？
- 给数组分配了存储空间后，能对数组初始化吗？
- 如果支持数组片，是哪种类型的数组片呢？

后续各小节将讨论最常用的编程语言中数组的设计选择例子。

6.5.2　数组和索引

数组中的特定元素是通过一个双层语法机制来引用的，其中第一部分是集合名称，第二部分是一个可能是动态的选择器，包含一项或者多项下标或索引。如果引用中的所有下标都是常量，则选择器就是静态的；否则，它就是动态的。选择操作可以看作是从数组名和下标值集合到集合中某个元素的映射。事实上，数组有时也称为有限映射。如果用符号来表达，这种映射可以表示为：

数组名称 (下标值链表) →元素

历史注释

Fortran 90 之前的 Fortran 语言和 PL/I 的设计者为数组下标选择了圆括号，因为那时没有其他合适的字符可用。穿孔卡并不包括方括号字符。

数组引用的语法是相当通用的：数组名称后跟下标列表，列表用圆括号或者方括号括起来。一些语言把多维数组作为数组的数组，每个下标放在独立的方括号内。用圆括号来包括下标表达式有一个问题：圆括号也常常用来包括子程序调用中的参数，这种用法使数组的引用非常像函数调用。例如，考虑下面的 Ada 赋值语句：

```
Sum := Sum + B(I);
```

在 Ada 中，因为子程序参数和数组下标都使用了圆括号，所以读程序的人和编译器只能使用其他信息来判断，这个赋值语句中的 B(I)到底是一个函数调用，还是一个对数组元素的引用。这就降低了代码的可读性。

历史注释

Fortran I 将数组下标的个数限制为 3，因为在设计语言时，主要考虑的因素是运行效率。Fortran I 的设计者利用 IBM 704 的 3 个索引寄存器，开发出了一种非常快捷的方式来访问数组(最多三维)中的元素。Fortran IV 率先在有 7 个索引寄存器的 IBM 7094 上实现，这就允许 Fortran IV 的设计者将数组的下标数增加到 7 个。而同时代的其他语言没有类似的限制。

尽管有潜在的可读性问题，但为了让数组引用和函数调用在形式上保持一致，Ada 的设计者故意选择了用圆括号来包括数组的下标。他们选择圆括号的部分原因是，数组元素引用和函数调用都是映射：数组元素引用把下标映射到数组的特定元素；而函数调用将实

参映射到函数定义以及最终的函数值。

除 Fortran 和 Ada 以外的大部分语言都使用方括号来限定数组下标。

在数组类型中涉及两个不同的类型：元素类型和下标类型。下标的类型通常是整数类型的子界，然而 Ada 允许把任何有序类型作为下标类型，例如布尔类型、字符类型和枚举类型等。例如，在 Ada 中，可编写下面的代码：

```
type Week Day Type is (Monday, Tuesday, Wednesday,
                       Thursday, Friday);
type Sales is array (Week_Day_Type) of Float;
```

Ada 的 for 循环可以使用任何有序类型变量作为它的计数器，参见第 8 章。这样，就能够很方便地处理以有序类型作下标的数组了。

早期的编程语言并未强制规定：必须隐式地检查下标的范围。在程序中，下标范围错误是很常见的，所以要求范围检查是语言可靠性中的一个重要因素。多数现代语言没有规定对下标进行范围检查，但是 Java、ML 和 C#有这样的规定。默认设置时，Ada 会检查所有下标的范围，但是这个功能可由程序员关闭。

Perl 中的下标操作有些特殊。Perl 中所有数组的名字都以@符号开头，因为 Perl 的数组元素总是标量，而标量的名字总是以$符号开头，因此，对数组元素的引用是在变量名中使用$符号，而不是@符号。例如，对于数组@list，用$list[1]来引用它的第 2 个元素。

在 Perl 中可以用负下标来引用数组的元素，在这种情况下，下标值表示从数组末尾开始的偏移量。例如，如果数组@1ist 有 5 个元素，下标为 0..4，则$list[-2]就是对下标为 3 的元素的引用。在 Perl 中引用不存在的元素，会产生 undef，而不是错误报告。

6.5.3 下标的绑定和数组的种类

数组变量与下标类型的绑定通常是静态的，但是下标的取值范围有时是动态限定的。

在有些语言中，下标范围的下限是隐含的。例如，在以 C 为基础的语言中，所有下标的下限都固定为 0；在 Fortran 95+中，则默认为 1，但可以设置为其他整型字面量。在其他一些语言中，下标的下限则必须由程序员指定。

根据下标范围的绑定、存储空间的绑定以及从哪里分配空间等方面的不同，数组可以分为五类。数组类别的名称表达了对上述三个方面的设计选择。在前四类数组中，一旦设定了数组下标的范围，分配了存储空间，则在数组变量的整个生存期内，下标范围与存储空间就会固定不变。记住，一旦下标范围是固定的，数组就不能改变大小。

静态数组的下标范围是静态限定的，存储空间的分配也是静态的(在运行前分配)。静态数组的优势在于效率高：不必动态分配和释放空间。缺点在于，数组的存储空间在程序的整个运行阶段都是固定的。

固定栈动态数组的下标范围是静态限定的，但存储空间是在程序执行到数组声明语句时才分配的。与静态数组相比，固定栈动态数组的优点是它在存储空间上的分配效率。只要两个子程序不是同时运行，一个子程序中的大数组就能与另一个子程序中的大数组使用

相同的存储空间。如果两个数组位于不同时运行的不同语句块中，也能共享存储空间。此类数组的缺点是分配和释放存储空间需要时间开销。

栈动态数组的下标范围和存储空间都是在程序运行到数组声明语句时才动态限定的。但是，一旦限定了下标范围，并分配了存储空间，它们就会在变量的整个生存期内保持不变。相对静态数组和固定栈动态数组，栈动态数组的好处在于它的灵活性。在用到数组之前，不需要知道数组的大小。

固定堆动态数组类似于固定栈动态数组，它的下标范围和存储区域在分配了存储空间后就固定下来。不同之处在于，下标范围和存储区域是在用户程序执行期间需要创建数组时才绑定的，而且存储空间是从堆中而不是从栈中分配的。固定堆动态数组的优势在于灵活性，即数组的大小总是与问题需求相适应。缺点在于从堆中分配空间的耗时会比从栈中分配空间的耗时长。

堆动态数组的下标范围和空间分配都是动态的，在数组生存期内能够改变任意次。相对于其他数组，堆动态数组的优势在于灵活性：数组能够在程序运行过程中根据空间需求的变化增长或收缩。其缺点是分配和释放空间的耗时较长，而且可能会在程序运行中执行许多次。下面给出了以上五种数组的例子。

在 C 和 C++的函数中声明的、有 static 限定符的数组是静态的。

在 C 和 C++的函数中声明的、没有 static 限定符的数组是固定栈动态数组。

Ada 语言中的数组是栈动态数组，如下例所示：

```
Get(List_Len);
declare
  List : array (1..List_Len) of Integer;
  begin
  ...
  end;
```

在这个例子中，用户输入数组 List 的元素个数。当程序运行到 declare 块时，为数组元素动态分配内存。当程序运行到该语句块的结束点时，会释放 List 数组的内存。

C 和 C++语言也提供了固定堆动态数组。对于 C 的数组，标准库函数 malloc 和 free 提供了常用的堆内存分配和释放操作。C++则使用 new 和 delete 操作符来管理堆空间。数组是一个指向存储单元集合的指针，该指针能与数组下标相结合，用于访问数组元素，参见第 6.11.5 节。

在 Java 中，所有非泛型数组都是固定堆动态数组。这些数组一旦创建，它们的下标范围和存储空间就不会再改变。C#也提供了这类数组。

C#还引入了泛型的堆动态数组，它是 List 类的对象。这些数组对象在创建时没有任何元素，如下：

```
List<String> stringList = new List<String>();
```

使用 Add 方法给这个对象添加元素，如下：

```
stringList.Add("Michael");
```

这些数组的元素是通过下标来访问的。

Java 引入了一个与 C#的 List 类似的类，即 Arraylist 类但该结构不支持下标操作，必须使用 get 和 set 方法来访问数组元素。

在 Perl 中，使用 push 函数(在数组末端添加一个或多个元素)和 unshift 函数(在数组头部添加一个或多个元素)，或将超过数组当前最大下标的下标赋给数组，就可以增大数组。而将数组赋予空列表()，就可以把数组缩减成没有元素的空数组。数组的长度定义为最大下标加 1。

像 Perl 一样，JavaScript 也允许数组使用 push 和 unshift 方法来增大，将数组设为空列表来缩小数组。但 JavaScript 不支持负数下标。

JavaScript 的数组可以是稀疏的，即数组的下标值不必是连续的。例如，假设 list 数组有 10 个元素，下标为 0..9。[5]考虑下面的赋值语句：

```
list[50] = 42;
```

现在，list 有 11 个元素，长度为 51。下标为 11..49 的元素都没有定义，所以不需要存储空间。在 JavaScript 数组中，引用不存在的元素会产生 undefined 错误。

在 Python、Ruby 和 Lua 中，要扩展数组的大小，只能添加元素或串接其他数组。Ruby 和 Lua 支持负数下标，但 Python 不支持。在 Python、Ruby 和 Lua 中，可以对单个元素或数组切片执行删除操作。在 Python 中，引用不存在的元素会导致运行时错误，而相似的引用操作在 Ruby 和 Lua 中会返回 nil，而不报错。

尽管 ML 定义不包含数组，但它广泛使用的实现版本 SML/NJ 包含数组。

在 F#中，唯一预定义的集合类型是数组(其他集合类型通过.NET 框架库来提供)。这些数组类似于 C#数组。该语言中的 foreach 语句可用于处理数组。

6.5.4 数组的初始化

有些语言提供了在数组分配内存的同时进行初始化的方法。在 Fortran 95+中，数组可以通过在声明中赋给它一个数列集合来初始化。一维数组的数列集合是一个夹在括号和斜线中间的字面量列表。例如：

```
Integer, Dimension (3) :: List = (/0, 5, 5/)
```

C、C++、Java 和 C#也允许对数组进行初始化，但有一个新变化。对于 C 语言的声明

```
int list [] = {4, 5, 7, 83};
```

编译器会设置数组的长度。这很方便，但并非没有代价。它降低了系统检测某些程序员错误的可能性，例如，不小心遗漏一个数值，没有将它包含在列表中。

---

5. 下标范围也可设置为 1000..1009。

如 6.3.2 节所述，C 和 C++中的字符串是用 char 数组来实现的。这些数组可以用字符串常量来初始化，例如：

```
char name [] = "freddie";
```

数组 name 有 8 个元素，因为所有的字符串都是以空字符(0)结束，这由系统隐式地提供给字符串常量。

C 和 C++中的字符串数组也可以用字符串字面量来初始化。这种情况下，数组是指向字符的指针。例如：

```
char *names [] = {"Bob", "Jake", "Darcie"};
```

这个例子说明了 C 和 C++中字符字面量的性质。在前面的例子中，字符串字面量用来初始化 char 数组 name，字面量处理为一个 char 数组。但是在后来的 names 例子中，字符量处理为指向字符的指针，所以数组是指针(指向字符)数组。例如，names[0]是一个指针，指向字符数组中的字母“B”，该数组包含字母“B”、“o”、“b”和一个空字符。

Java 使用相似的语法来定义和初始化引用 String 对象的数组。例如：

```
String[] names = ["Bob", "Jake", "Darcie"];
```

Ada 提供了两种机制来初始化声明语句中的数组：按存储顺序列出数组元素，或者使用 Ada 中称为箭头的操作符=>，直接给它们赋予索引位置。例如，考虑下面的语句：

```
List : array (1..5) of Integer := (1, 3, 5, 7, 9);
Bunch : array (1..5) of Integer := (1 => 17, 3 => 34,
                                    others => 0);
```

在第一个语句中，数组 List 的所有元素都有初始化值，以出现的先后次序赋给相应位置的数组元素。在第二个语句中，第一个和第三个数组元素使用直接赋值来初始化，others 子句用来初始化剩余的元素。与 Fortran 一样，这些由括号括起来的数组列表称为聚集值。

6.5.5　数组操作

数组操作是将整个数组当作一个整体进行处理的操作。最常见的数组操作就是赋值、连接、相等和不等比较以及切片。6.5.7 节会单独讨论切片的问题。

基于 C 的语言没有提供任何数组操作，只能通过 Java、C++和 C#语言的方法来处理数组。Perl 支持数组赋值，但不支持数组比较。

Ada 支持数组赋值操作，包括数组等号右边是一个聚集值而不是数组名的赋值。Ada 还支持用&符号进行连接操作。这种连接可以在两个一维数组之间，以及一维数组与一个标量之间进行。几乎所有的 Ada 类型都内置了判断相等或不相等的关系操作符。

Python 的数组称为列表，不过它拥有动态数组的所有特点。由于对象可以是任意类型，所以这些数组是混杂型的。Python 提供了数组赋值操作，不过它只是改变引用关系。Python 还有数组连接操作(+)和判断元素是否在数组中的操作(in)。它有两种不同的比较运算符：

一种是判断两个变量是否引用了同一个对象(is)，另一种是判断是否相等(==)，它会比较在被引用对象中的所有对象，无论它们嵌套的层次有多深。

与 Python 类似，Ruby 数组的元素也是对象的引用。而且，与 Python 相似，当 Ruby 的==运算符用来比较两个数组时，只有当两个数组有相同的长度，并且对应元素也相等时，结果才是真。Ruby 的数组可以用 Array 方法来连接。

Fortran 95+包含一些面向元素的数组操作，因为这些操作是在数组元素对之间进行。例如，两个数组的加法操作符(+)会产生一个数组，它的元素是两个数组的元素对之和。赋值操作符、算术操作符、关系操作符以及逻辑操作符都是重载的，能适用于任何大小或形态的数组。Fortran 95+也包括矩阵相乘、矩阵转置和向量点积的内部函数或库函数。

F#在其 Array 模块中包含许多数组操作符，其中包括 Array.append、Array.copy 和 Array.length。

数组及其操作是 APL 语言的核心，它是最强大的数组处理语言。然而由于它相对晦涩难懂，缺乏对后续语言的影响力，这里只简单介绍它的数组操作。

在 APL 中，有四个基本的算术操作是为向量(一维数组)、矩阵和标量操作数定义的。例如，

```
A + B
```

就是一个合法的表达式，不论 A 和 B 是标量变量、向量还是矩阵。

APL 包括了一组用于向量和矩阵的一元操作符，比如下面的一些示例(其中 V 代表向量，M 代表矩阵)：

```
⌽V 反转向量中的元素
⌽M 反转矩阵中的列向量
⊖M 反转矩阵中的行向量
⍉M 转置矩阵(行变成列)
⌹M 反置矩阵
```

APL 还包括几个特殊的操作符，它们把其他操作符作为操作数。其中一个就是内积操作符，用句点(.)表示。它有两个操作数，都是二元操作符。比如：

```
+.×
```

就是一个新操作符，可以带两个参数，参数可以是向量或者矩阵。它首先把两个参数的对应部分相乘，然后把结果相加。例如，如果 A 和 B 都是向量，那么，

```
A×B
```

就是 A 和 B 的数学内积(由 A 和 B 对应元素的积组成的向量)。语句

```
A+.×B
```

是 A 和 B 内积的和。如果 A 和 B 是矩阵，这个表达式就表示 A 和 B 的矩阵乘积。

APL 语言的特殊操作符实际上是函数形式，参见第 15 章。

6.5.6　矩形数组和不规则数组

矩形数组是一种多维数组，它的所有行都有相同数目的元素，所有列都有相同数目的元素。矩形数组精确模拟了矩形表格。

不规则数组中每行的长度不一定相同。例如，不规则矩阵可能有三行，每行分别有 5 个、7 个和 12 个元素。数组的列和更高维也是这样。因此，如果数组有第三维(或层)，每层可以有不同数目的元素。多维数组实际上是数组的数组时，才可能存在不规则数组。例如，矩阵可以显示为一维数组的数组。

C、C++和 Java 都支持不规则数组，但不支持矩形数组。在这些语言中，对多维数组中某个元素的引用使用的是各维分开的括号对，例如：

```
myArray[3][7]
```

Fortran、Ada、C#和 F#支持矩形数组(C#和 F#也支持不规则数组)。在这些语言中，引用数组元素的所有下标表达式都放在一对括号里，如：

```
myArray[3, 7]
```

6.5.7　切片

数组切片是数组的某个子结构。例如，A 是一个矩阵，A 的第一行就是一个可能的切片，最后一行和第一列也是可能的切片。切片并不是一种新的数据类型，这非常重要。应该说，这是一种把数组的一部分作为一个整体来引用的机制。如果语言不能把数组当作单元来操作，则切片在该语言中就没有什么用处。

考虑下面的 Python 声明：

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
```

Python 数组的默认下界是 0。Python 切片引用的语法是一对用冒号分开的数值表达式。第一个数字是切片的首下标，第二个数字是切片最末下标之后的第一个下标。因此，vector[3:6]是一个有三个元素的数组，包含 vector 的第 4 至第 6 个元素(也就是下标分别为 3、4、5 的元素)。只用一个下标就可以指定矩阵的一行。例如，mat[1]代表 mat 的第 2 行。指定一行的一部分时，可以使用与一维数组相同的语法。例如，mat[0][0:2]表示 mat 第 1 行的第 1 个和第 2 个元素，即[1, 2]。

Python 还支持更复杂的数组切片。例如，vector[0:7:2]代表 vector 中下标从 0 到 7(不含 7 自身)、且为偶数的元素，即[2, 6, 10, 14]。

Perl 支持两种形式的切片—指定下标或者下标范围组成的列表。例如：

```
@list[1..5] = @list2[3, 5, 7, 9, 13];
```

注意：切片引用使用了数组名，而不是标量名，这是因为切片是数组，而不是标量。

Ruby 通过 Array 对象的 slice 方法来支持切片，方法的参数有三种形式。单个整型表

达式参数是下标，这时 slice 方法返回给定下标的元素。如果 slice 方法有两个整型表达式参数，则第一个参数是起始下标，第二个为片中的元素个数。例如，假定 list 的定义如下：

```
list = [2, 4, 6, 8, 10]
```

list.slice(2，2)返回[6，8]。slice 方法的第三种参数形式是一个范围，其格式为一个整型表达式，两个句点，第二个整型表达式。slice 方法使用范围参数，返回由数组中给定下标范围的元素所组成的数组。例如 list.slice(1..3)返回[4, 6, 8]。

6.5.8 评价

事实上，所有的编程语言都包含数组。自从 Fortran I 引进数组以来，主要的进展有：把所有的序数类型都包含进来作为可能的下标类型、切片，当然还有动态数组。如 6.6 节所述，联合数组是数组最近取得的进展。

6.5.9 数组类型的实现

实现数组类型比实现基本类型需要更多的编译时间。访问数组元素的代码必须在编译阶段生成。在运行阶段，必须执行这些代码以生成数组元素的地址。没有办法预先计算出引用(如 list[k])要访问的地址。

一维数组是用一组相邻的内存单元来实现的。假设数组 list 下标范围的下界是 0。list 的访问函数通常是这样的格式：

```
list[k]的地址 = list[0]的地址 + k * 元素大小
```

其中，加法的第一个操作数是访问函数的常量项，第二个操作数则是变量项。

如果元素的类型是静态绑定的，数组到存储空间也是静态绑定的，则常量部分的值就能在运行之前计算出来。不过，加法和乘法操作必须在运行时完成。

对下标下界为任意值的数组，该访问函数的一般形式为：

```
list[k]的地址 = list[下界]的地址 + ((k-下界)*元素大小)
```

一维数组在编译期间的描述符形式如图 6-4 所示。该描述符包含了生成访问函数所需要的信息。如果运行时不做索引范围校验，并且属性都是静态的，则运行期间只需要访问函数，不需要描述符。如果在运行时执行下标范围校验，则这些下标范围可能需要存储在一个运行时的描述符中。如果某个数组的下标范围是静态的，则该范围可能合并入做校验的代码中，这样就不需要运行时描述符了。如果描述符的某一项是动态绑定的，则描述符的这些部分必须在运行时予以维护。

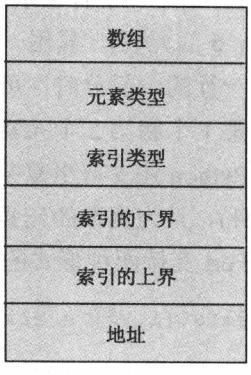


图 6-4 一维数组的编译时描述符

真正的多维数组并不是数组的数组，它们实现起来要

比一维数组复杂得多，尽管从一维向多维数组的扩展是很直接的。硬件存储单元是线性的，通常就是一个简单的字节序列。所以，二维或者多维的数据类型的值必须映射到一维的存储单元上。有两种方法能把多维数组映射到一维上：按行存放和按列存放。在按行存放中，第一个下标是下标下界值的数组元素先存放，然后是第一个下标为第二个值的元素，依此类推。如果数组是一个矩阵，则按行存储它。例如，如果矩阵如下：

```
3    4    7
6    2    5
1    3    8
```

按行存放的存储顺序为：

```
3, 4, 7, 6, 2, 5, 1, 3, 8
```

在按列存放中，最后一个下标是下标下界值的数组元素先存放，然后是最后一个下标为第二值的元素，以此类推。如果数组是一个矩阵，则按列存储它。如果上例中的矩阵按列存放，则它在内存中的顺序如下：

```
3, 6, 1, 4, 2, 3, 7, 5, 8
```

按列存放常用在 Fortran 语言中，而其他具有真正的多维数组的语言则使用按行存放。

多维数组的访问函数将数组基址与一组索引值映射到由下标值指定的元素的内存地址。按行存放的二维数组的访问函数可以按如下方法来生成。通常，某个元素的地址等于结构基址加上元素大小乘以结构中在它之前的元素数。对于按行存放的矩阵，某个元素之前的元素数目等于元素之前的行数乘以行的大小，再加上元素左边的元素个数。如图 6-5 所示，其中假定下标下界都是 0。

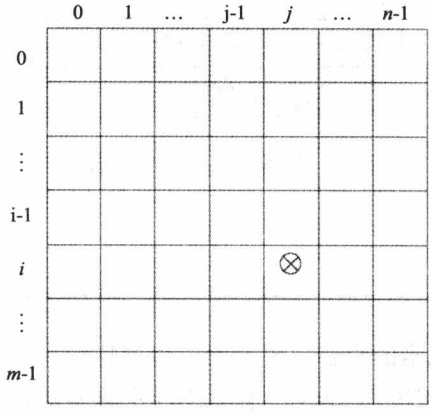


图 6-5　元素[i, j]在矩阵中的位置

为了得到实际的地址值，目标元素前的元素个数必须乘上元素大小。现在，访问函数可以写成

```
a[i, j]的位置 = a[0, 0]的地址+(((第 i 行之前的行数*行的大小)
                              +第 j 列左边的元素数)*元素大小)
```

因为第 i 行之前的行数是 i，且第 j 列左边的元素数是 j，所以

```
a[i, j]的位置=a[0, 0]的地址+(((i*n)+j)*元素大小)
```

其中，n 是每行的元素个数。第一项是常量项，第二项是可变项。

推广到以任意值作为下界，则访问函数的形式如下：

```
a[i, j]的位置=a[row_lb, col_lb]的地址
            +(((i-row_lb)*n)+(j-col_lb))*元素大小
```

其中，row_lb 为行的下界，col_lb 为列的下界。这个式子变形为：

```
a[i, j]的位置=a[row_lb, col_lb]的地址

              -(((row_lb*n)+col_lb)*元素大小)
              +(((i*n)+j)*元素大小)
```

式中的前两项是常量，最后一项是变量。这可以比较简单地推广到任意维的数组。

对数组的各维，访问函数需要一个加法指令和一个乘法指令。因此，访问多下标的数组元素是有代价的。多维数组编译时的描述符如图 6-6 所示。

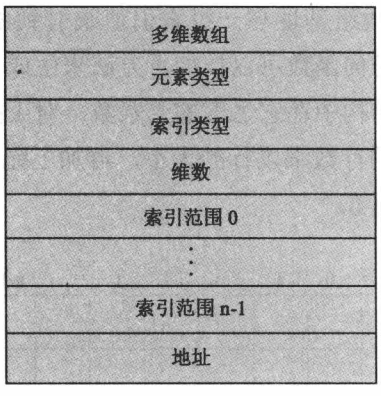


图 6-6　多维数组编译时的描述符

6.6　关联数组

关联数组是无序数据元素的集合，集合元素以同样数目的键值为索引。对于非关联数组，不必储存索引 (因为它们的规律性)。然而对于关联数组，用户定义的键值必须保存在结构中。因此，关联数组的每个元素实际上是成对的两项，一个是键值，一个是数据。我们用 Perl 语言的关联数组设计来描述这种数据结构。关联数组还得到 Python、Ruby 和 Lua 语言直接支持，Java、C++、C#和 F#的标准类库也支持它。

对于关联数组，其唯一的设计问题是数组元素的引用形式。

6.6.1　结构和操作

在 Perl 中，关联数组称为散列，因为在实现时数组元素是通过散列函数来存储和检索的。Perl 中不同散列的名字空间是不同的：每一个散列变量名必须以百分号(%)开头。散列的每个元素包括两个部分：一个是键，它是字符串类型；一个是值，它是标量类型(即数字、字符串或引用)。散列可以用赋值语句设为字面量值，如：

```
%salaries = ("Gary" => 75000, "Perry" => 57000,
             "Mary" => 55750, "Cedric" => 47850);
```

Perl 使用与数组相似的标记方法来引用各个元素的值。键值放在花括号内，散列名换成了标量名，这个名字除了第一个字符外都与散列名相同。尽管散列不是标量，但散列元素的数值部分是标量，所以用标量名字来引用散列元素的数值部分。标量变量的名字以$开头。例如：

```
$salaries{"Perry"} = 58850;
```

使用同样的赋值语句形式可以在散列中添加新元素。元素可以使用 delete 操作符从散列中删除掉，例如：

```
delete $salaries{"Gary");
```

给散列赋予一个空的字面量，可以将散列清空，例如：

```
@salaries=();
```

Perl 散列的大小是动态的：当增加新的元素时，它会变大；当删除元素以及赋予空的字面量而被清空时，它会变小。操作符 exists 返回为真或假，这取决于它的操作数键码是否是散列中的一个元素。例如：

```
if (exists $salaries{"Shelly"}) ...
```

当操作符 keys 应用于一个散列时，它返回散列的键值数组。操作符 values 返回散列的值的数组。操作符 each 对散列的每个元素对进行迭代。

Python 的关联数组(称为字典)和 Perl 的相似，但是它的值都是对象的引用。Ruby 中支持的关联数组和 Python 的相似，但是它的键可以是任意对象[6]，而不仅仅是字符串。因此，这是一个逐步发展的过程：从 Perl 的散列(键必须是字符串)，到 PHP 的数组(键可以是整数或者字符串)，再到 Ruby 的散列(键可以是任意的对象)。

PHP 的数组既是常规数组又是关联数组，可以相互转换。该语言提供的函数允许用下标和散列来访问数组的元素。数组的元素可用简单的数值索引创建，也可用字符串散列键值来创建。

---

6. 会发生变化的对象不合适作为键值，因为对象变化了，会改变散列函数的值。因此，数组和散列从不用作键值。

在 Lua 中，表格类型是唯一的数据结构。Lua 的表格是关联数组，它们的键和值都可以是任何类型。表格可以用作传统数组、关联数组或记录(结构)。当用作传统数组或关联数组时，用方括号把键括起来。当用作记录时，键就是记录的域名，并且可以用点标记法来引用记录的域，即记录名.域名。

关于 Lua 的关联数组作为记录来使用，将在 6.7 节讨论。

C#和 F#通过一个.NET 类来支持关联数组。

如果需要搜索元素，关联数组比数组好得多，因为使用隐式散列操作来访问元素是非常有效的。而且，当数据成对存储时(如雇员的姓名和薪水)，关联数组将是很理想的。另一方面，如果必须处理列表中的每个元素，则使用数组更高效。

6.6.2 关联数组的实现

Perl 关联数组的实现为快速查找进行了优化，当需要增长数组大小时，它也能提供相当快速的重组性能。虽然关联数组在刚开始时只使用了散列值的一小部分，但还是为每个数据项计算出一个 32 位的散列值，并和数据项存储在一起。当关联数组必须扩大，超过它的初始大小时，并不需要改变散列函数，而是使用更多的散列值二进制位。这时，只有一半的数据项需要移动。因此，虽然关联数组的扩大并不是没有开销，但并没有预想的那样高。

在 PHP 中，数组元素通过散列函数来存入内存。然而，所有元素都按创建顺序链接在一起。通过 current 和 next 函数，可以用这些链接来对元素进行迭代访问。

6.7 记录类型

记录是指各种数据元素的集合。在集合中，单个元素通过名称来标识，并通过从结构开始的偏移量来访问。

在程序中，经常需要为非同一类型或大小的数据集合建模。例如，在校大学生的信息通常包括姓名、学号、年级平均成绩等等。针对这一集合的数据类型应当使用字符串来记录姓名，使用整数来记录学号，使用浮点数来记录年级平均成绩等等。记录就是为这些需要而设计的。

记录和含有混合数据的数组似乎是一样的，但是事实并非如此。混合数组的元素都是对分散在不同地方的数据对象的引用，这些元素通常存放在堆中。记录的元素可能有不同的大小，存放在相邻的存储单元内。

记录从 1960 年代初期由 COBOL 引入，之后记录就成为所有最流行的编程语言(Fortran 90 以前的版本除外)中的一部分。在一些支持面向对象的编程语言中，记录用数据来模拟。

在 C、C++和 C#中，记录由 struct 数据类型来支持。在 C++中，结构与类的差异很小。在 C#中，结构同样与类有关，但很不相同。C#结构是栈上分配的值类型，而类的对象与此相反，是堆上分配的引用类型。结构在 C++和 C#中通常用作封装结构，而不只是数据结构。结构的这种用途将在第 11 章深入讨论。ML 和 F#也包含结构。

在 Python 和 Ruby 中，记录可以实现为散列，记录自身能作为数组的元素。

下面的各节将介绍如何声明和定义记录、如何引用记录中的域，以及常见的记录操作。

记录的设计问题是：

- 引用记录域的语法形式是什么？
- 是否允许省略引用？

6.7.1　记录的定义

记录与数组的根本区别在于记录的元素(或域)不是用下标来引用的。相反，在记录中，域是用标识符来命名的，并且用这些标识符来引用。记录与数组的另一个不同之处是：在一些编程语言中，记录可以包含联合，参见 6.10 节。

下面的示例说明了 COBOL 语言的记录声明形式，这是 COBOL 程序的数据段部分：

```
01  EMPLOYEE-RECORD.
    02 EMPLOYEE-NAME.
        05 FIRST    PICTURE IS X(20).
        05 MIDDLE   PICTURE IS X(10).
        05 LAST     PICTURE IS X(20).
    02 HOURLY-RATE PICTURE IS 99V99.
```

EMPLOYEE-RECORD 记录由 EMPLOYEE-NAME 记录和 HOURLY-RATE 域组成。记录声明各行前面的数字 01、02 和 05 是级别数字，它们的相对值表明了记录结构的层级。任何一行，如果跟着它的行有更高的级别数字，则这一行本身就是一个记录。PICTURE 子句表明记录域储存位置的格式，X(20)指定 20 个字母或数字，99V99 指定小数点在中间的 4 位数字。

Ada 的记录使用了一种不同的语法形式，它没有使用 COBOL 的级别数字，而是简单地将记录声明嵌套到记录声明中，以正交的方式来指定其记录结构。在 Ada 中，记录是不能匿名的，而必须是有名字的类型。考虑如下的 Ada 声明：

```
type Employee_Name_Type is record
   First : String (1..20);
   Middle : String (1..10);
   Last : String (1..20);
end record;
type Employee_Record_Type is record
   Employee_Name: Employee_Name_Type;
   Hourly_Rate: Float;
end record;
Employee_Record: Employee_Record_Type;
```

在 Java 和 C#中，记录可以定义为数据类，嵌套记录定义为嵌套类。这些类的数据成员作为记录的域。

如前所述，Lua 的关联数组可以方便地用作记录。例如，考虑如下的声明：

```
employee.name = "Freddie"
```

```
employee.hourlyRate = 13.20
```

这些赋值语句创建了表格(记录)employee，其中包含两个初始化的元素(域)name 和 hourlyRate。

6.7.2 记录域的引用

对记录中单个域的引用，在语法上可用几种不同的方法来说明，其中的两种需要命名所需的域和包含它的记录。COBOL 的域引用具有如下形式：

```
域名 OF 记录名 1  OF ... OF 记录名 n
```

其中，第一个命名的记录是指包含特定域的最小或最内层的记录。序列中的下一个记录名是包含前一个记录的记录名，以此类推。例如，在上面的 COBOL 记录示例中，可按以下方式来引用 MIDDLE 域：

```
MIDDLE OF EMPLOYEE-NANE OF EMPLOYEE-RECORD
```

大多数其他语言采用点标记来引用域，引用中的各个部分由句点连接起来。点标记法中记录名的次序与 COBOL 引用中的相反：先是最大包含记录的名字，最后是域的名字。例如，下面是对前述 Ada 记录示例中的 Middle 域的引用：

```
Employee_Record.Employee_Name.Middle
```

C 和 C++使用相同的语法对结构成员进行引用。

对 Lua 表格元素的引用采用记录域引用的语法，如 6.7.1 节中的赋值语句所示。这种引用也能采用常规表格元素的形式，例如，employee["name"]。

记录域的完全限定引用，是指所有的中间记录名，从最大包含记录到特定的域，都在引用中指定。前面 COBOL 和 Ada 示例中的域引用都是完全限定引用。作为对完全限定引用的一种替代形式，COBOL 允许对记录域进行省略引用。在省略引用中，必须有域名，但能省略部分或全部的包含记录名，只要生成的引用在引用环境中没有歧义即可。例如，FIRST、FIRST OF EMPLOYEE-NAME 和 FIRST OF EMPLOYEE-RECORD，都是对前面 COBOL 记录声明中的雇员名的省略引用。虽然省略引用能给程序员带来方便，但它要求编译器具有精巧的数据结构和处理过程，才能正确地识别出被引用的域。此外，它们还会在某种程度上影响程序的可读性。

6.7.3 评价

在编程语言中，记录通常是很有用的数据类型。记录类型的设计非常简单，使用起来也很安全。

记录和数组是紧密相关的结构形式，因此比较它们是很有趣的。当所有数据值有相同的类型，并且(或者)以相同方式来处理时，就选择使用数组。如果有系统化的方法来遍历结构，这种处理过程就很容易完成。使用动态下标作为寻值的方法，可以很好地支持这种

处理过程。

当一组数据的值是不同的类型，而且不同的域不是以同一方式来处理时，就采用记录。另外，记录的各个域通常也不需要按某种特定的次序来处理。域名类似于字面量、常量或者下标。因为域名是静态的，所以提供了对域的高效访问。动态下标可以用来访问记录的各个域，但是它不允许类型检查，而且速度慢。

记录和数组提供了考虑周全且有效率的方法，满足了对数据结构的两种不同但相关的应用需求。

6.7.4　记录类型的实现

记录中的各个域存储在连续的内存空间中，但因为各个域的大小未必相同，所以数组的访问方式不能用于记录。记录使用偏移地址，即相对记录头的偏移量与各个域相关联。对域的访问是通过这些偏移量来完成的。记录在编译时的描述符的通用形式如图 6-7 所示，记录不需要运行时描述符。

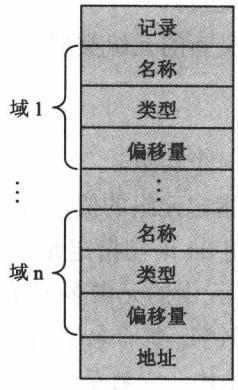


图 6-7　编译时的记录描述符

6.8　元组类型

元组这种数据类型类似于记录，但其元素没有命名。

Python 包含一个不可变的元组类型。如果需要改变元组，就可以使用 list 函数把它转换为一个数组。之后，再用 tuple 函数把它转换回元组。元组的一个用途是当必须把数组编写为受保护的类型，例如把数组发送为外部函数的一个参数，且用户不希望函数能修改参数时，就可以使用元组。

Python 的元组与其列表紧密相关，但元组是不可变的。指定元组字面量，就可以创建元素，如下所示：

```
myTuple = (3, 5.8, 'apple')
```

注意元组的元素不一定需要具有相同的类型。

元组的元素在引用时，可以将索引放在方括号中，如下所示：

```
myTuple[1]
```

这个语句引用了元组的第一个元素，因为元组的索引从 1 开始。

元组可以用加号(+)连接起来，用 del 语句删除它们。元组还可以用其他操作符和函数来操作。

ML 包含一个元组数据类型。ML 元组至少要有两个元素，而 Python 中的元组可以为空，或只包含一个元组。与 Python 一样，ML 元组也可以包含混合类型的元素。下面的语句就创建了一个元组：

```
val myTuple = (3, 5.8, 'apple');
```

访问元组元素的语法如下：

```
#1(myTuple);
```

这个语句引用了元组的第一个元素。

在 ML 中，可以用类型声明定义新的元组类型，如下所示：

```
type intReal = int * real;
```

这个类型的值包括一个整数和一个实数。

F#也有元组。创建元组时，应指定元组的值，该值是一个表达式列表，表达式用逗号分隔开，整个列表放在括号中，并用 let 语句指定元组的名称。如果元组有两个元素，就分别用函数 fst 和 snd 来引用它们。若元组的元素多于 2 个，则常常用左边是 let 语句的元组模式来引用。元组模式只是一系列名字，每个名字用于元组中的一个元素(包含或不包含界定的括号)。元组模式的左边是 let 结构，它就是一个多重赋值语句。例如，考虑下面的 let 结构：

```
let tup = (3, 5, 7);;
let a, b, c = tup;;
```

上述语句把 3 赋予 a，把 5 赋予 b，把 7 赋予 c。

在 Python、ML 和 F#中使用元组，函数就可以返回多个值。

6.9 列表类型

列表是第一种函数式编程语言 LISP 率先支持的一种类型，它总是函数式语言中的一部分，近年来，一些命令式语言也包含它们。

Scheme 和 Common LISP 中的列表用圆括号来界定，其中的元素不用任何标点符号分隔。例如：

```
(A B C D)
```

嵌套列表的形式相同，例如：

```
(A (B C) D)
```

在这个列表中，(B C)是嵌套在外层列表中的列表。

在 LISP 及其衍生的语言中，数据和代码的语法相同。如果列表(A B C)解释为代码，就表示调用带参数 B 和 C 的函数 A。

在 Scheme 中，列表的基本操作是把列表分隔的两个函数和建立列表的两个函数。CAR 函数返回其列表参数的第一个元素。例如下面的例子：

```
(CAR '(A B C))
```

参数列表前的引号用于防止解释器把该列表看作调用带参数 B 和 C 的函数 A(若列表前面没有引号，解释器就会将该列表看作调用带参数 B 和 C 的函数 A)。这个对 CAR 的调用返回 A。

CDR 函数返回其参数列表去掉第一个元素的结果。例如下面的例子：

```
(CDR '(A B C))
```

这个函数调用返回列表(B C)。

Common LISP 还有函数 FIRST (与 CAR 的作用相同)、SECOND、…、 TENTH，它们分别从其列表参数中返回函数名称指定的元素。

在 Scheme 和 Common LISP 中，用 CONS 和 LIST 函数构建新列表。CONS 函数带两个参数，返回一个新列表，函数的第一个参数作为新列表的第一个元素，第二个参数作为新列表的其余元素。例如下面的例子：

```
(CONS 'A '(B C))
```

这个调用返回新列表(A B C)。

LIST 函数带任意数量的参数，它返回一个新列表，该函数的参数就是新列表的元素。例如，考虑下面的 LIST 调用：

```
(LIST 'A 'B '(C D))
```

这个调用返回新列表(A B (C D))。

ML 也有列表和列表操作，但其外观与 Scheme 不同。ML 中的列表在方括号中指定，元素用逗号分隔开，如下面的整数列表：

```
[5, 7, 9]
```

[]是空列表，也可用 nil 来指定它。

Scheme 的 CONS 函数在 ML 中实现为一个二元插入操作符，表示为::。例如：

```
3 :: [5, 7, 9]
```

返回新列表[3, 5, 7, 9]。

列表中元素的类型必须相同，所以下面的列表是非法的：

```
[5, 7.3, 9]
```

对于 Scheme 的 CAR 和 CDR 函数，ML 也有对应的函数 hd(head)和 tl (tail)。例如：

```
hd [5, 7, 9] is 5
tl [5, 7, 9] is [7, 9]
```

Scheme 和 ML 中的列表及列表操作详见第 15 章。

F#与 ML 中的列表有一定的关系，但有几个显著的区别。F#中列表的元素用分号分隔开，而 ML 用逗号分隔开。hd 和 tl 的操作相同，但在 ML 中它们称为 List 类的方法。例如，List.hd [1; 3; 5; 7]返回 1。F#的 CONS 操作使用两个冒号指定，与 ML 相同。

Python 包含列表数据类型，它也用作 Python 的数组。与 Scheme、Common LISP、ML 和 F#的列表不同，Python 的列表是可变的。它可以包含任意数据值或对象。给列表值赋予一个名称，就创建了 Python 列表。列表值是一系列表达式，它们用逗号分隔开，放在方括号中。例如以下语句：

```
myList = [3, 5.8, "grape"]
```

引用列表的元素时，应把下标放在方括号中，如下所示：

```
x = myList[1]
```

这个语句把 5.8 赋予 x。列表的元素索引从 0 开始。列表元素也可以用赋值语句来更新，用 del 删除，如下所示：

```
del myList[1]
```

这个语句删除了 myList 的第二个元素。

Python 包含创建数组的一种强大机制，称为列表包含(list comprehensions)。列表包含派生于 set 记号。它首次出现在函数式编程语言 Haskell 中(参见第 15 章)。列表包含的机制是：把函数应用于给定数组的每个元素，新数组从结果中构建出来。Python 列表包含的语法如下：

```
[表达式 for 迭代变量 in 数组 if 条件]
```

下面是一个例子：

```
[x * x for x in range(12) if x % 3 == 0]
```

range 函数创建了数组[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]。条件语句过滤出数组中所有不能被 3 整除的数字。接着表达式计算剩余数字的平方，将平方的结果放在一个数组中并返回。列表包含会返回以下数组：

```
[0, 9, 36, 81]
```

Python 还支持列表的切片。

Haskell 的列表包含如下形式：

```
[body | qualifiers]
```

例如，下面是一个列表的定义：

```
[n * n | n <- [1..10]]
```

这个语句定义了一个列表，其中包含从 1 到 10 的数字的平方。

F#也有列表包含功能，在该语言中，列表包含功能也可以用于创建数组，例如以下语句：

```
let myArray = [|for i in 1 .. 5 -> (i * i) |];;
```

这个语句创建了数组[1; 4; 9; 16; 25]，并命名为 myArray。

如 6.5 节所述，C#和 Java 分别支持泛型堆动态集合类 List 和 ArrayList。这些结构实际上就是列表。

6.10　联合类型

联合是在程序执行的不同时段可以存储不同类型值的一种类型。编译器的常量表就是一个需要联合的例子，它用来存储从正在编译的程序中发现的常量。每个表项都是常量值。假设正在编译某个语言，其中，常量的类型是整型、浮点型和布尔型。就表格管理而言，如果在同一位置(即表格中的域)上能够存储这三种类型的值，那将是很方便的。于是，所有常量值都能以同样的方式寻址。在某种意义上，该位置上的类型是三种值类型的联合。

6.10.1　设计问题

6.12 节讨论的联合类型的类型检查是一个重要的设计问题。另一个基本问题是在语法上如何表示联合。在某些设计中，联合仅是记录结构的一部分，而在其他的设计中没有这个限制。因此，联合类型的主要设计问题如下：

- 需要进行类型检查吗？注意，这样的类型检查必须是动态的。
- 联合应嵌入到记录中吗？

6.10.2　判别式联合与自由联合

C 和 C++都提供了联合结构，但它们都不支持类型检查。在 C 和 C++中，使用 union 结构来说明联合。这些语言中的联合称为自由联合，因为程序员在使用它们的过程中完全不需要进行类型检查。看看下面的 C 联合：

```
union flexType {
   int intEl;
   float floatEl;
};
```

```
union flexType el1;
float x;
...
el1.intEl = 27;
x = el1.floatEl;
```

最后一个赋值语句没有执行类型检查，因为系统不能确定 el1 当前值的当前类型，所以它把 27 的二进制位串表示赋给 float 变量 x，当然，这是没有意义的。

对联合的类型检查要求每个联合在构造时包含一个类型指示器。这个指示器称为标签，或者判别式，有判别式的联合称为判别式联合。第一种提供判别式联合的语言是 ALGOL 68。现在，Ada、ML、Haskell 和 F#都支持判别式联合。

6.10.3 Ada 的联合类型

Ada 在其前辈语言 Pascal 的基础上，对判别式联合所做的设计允许用户指定变体记录类型的变量，它在变体中只存储一个可能类型的值。这样，用户就可以告诉系统什么时候类型检查可以是静态的。这种受限制的变量称为受限变体变量。

受限变体变量的标签可作为一个命名常量。Ada 中的非受限变体记录允许变体的值在执行过程中改变类型。然而，变体的类型只能通过赋值包括判别式在内的整个记录来改变。这就不允许出现不一致的记录，因为如果新赋值的记录是一个常量数据的集合，则标签的值和变体的类型就会进行静态的一致性检查[7]。如果所赋的值是一个变量，那么在它赋值时就保证了一致性，所以赋予变量的新值可以保证是一致的。

下例展示了一个 Ada 的变体记录：

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form : Shape) is
  record
    Filled : Boolean;
    Color : Colors;
    case Form is
      when Circle =>
        Diameter : Float;
      when Triangle =>
        Left_Side : Integer;
        Right_Side : Integer;
        Angle : Float;
      when Rectangle =>
        Side_1 : Integer;
        Side_2 : Integer;
    end case;
    end record;
```

7. 在这里，一致性的意思是，如果标签规定联合的当前类型是 Integer，则联合的当前值事实上也是 Integer。

这个变体记录的结构如图 6-8 所示。下面两条语句声明了 Figure 类型的变量：

```
Figure_1 : Figure;
Figure_2 : Figure(Form => Triangle);
```

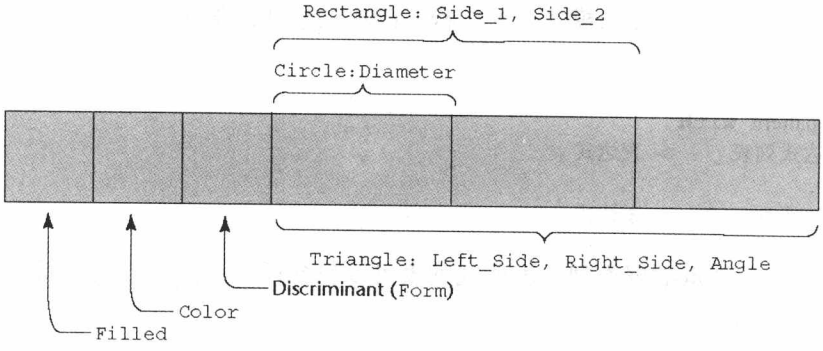


图 6-8　三种形状变量(假定所有变量的大小相同)的判别式联合

Figure_1 声明为一个无初始值的非受限变体记录。它的类型可以通过整个记录的赋值来改变，包括判别式，如下所示：

```
Figure_1 := (Filled => True,
             Color => Blue,
             Form => Rectangle,
             Side_1 => 12,
                       Side_2 => 3);
```

这个赋值语句的右边是一个数据集合。

声明的变量 Figure_2 限定为 triangle，不能改为其他变体。

这种形式的判别式联合是安全的，因为它总是允许做类型检查，但必须动态检查非受限变体中域的引用。例如以下语句：

```
if(Figure_1.Diameter > 3.0) ...
```

运行时系统需要检查 Figure-1，以判断它的 Form 标签是否为 Circle。如果不是，则引用它的 Diameter 将是一个类型错误。

6.10.4　F#的联合类型

在 F#中，联合是用 type 语句来声明的，在此语句中，用 OR 操作符(|)来定义其元素。例如下面的例子：

```
Type intReal =
   |IntValue of int
   |RealValue of float;;
```

在这个例子中，intReal 是联合类型。IntValue 和 RealValue 是构造函数。intReal 类型的

值可以用构造函数创建，就好像它们是函数一样，如下所示[8]：

```
let ir1 = IntValue 17;;
let ir2 = RealValue 3.4;;
```

访问联合的值时，要使用模式匹配结构。F#中的模式匹配用保留字 match 指定。该结构的一般形式如下：

```
match pattern with
  | 表达式列表1 - > 表达式1
  | ...
  | 表达式列表n - > 表达式n
```

pattern 可以是任意数据类型。表达式列表可以包含通配符(_)，也可以只是一个通配符。例如下面的 match 结构：

```
let a = 7;;
let b = "grape";;
let x = match (a, b) with
    | 4, "apple" -> apple
    | _, "grape" -> grape
    | _ -> fruit;;
```

为了显示 intReal 联合的类型，可以使用下面的函数：

```
let printType value =
match value with
    |IntValue value -> printfn "It is an integer"
    |RealValue value -> printfn "It is a float";;
```

下面列出了对这个函数的调用，并显示了结果：

```
printType ir1;;
It is an integer
printType ir2;;
It is a float
```

6.10.5 评价

在某些语言中，联合是具有潜在不安全性的结构。联合是导致 C 和 C++不是强类型语言的原因之一：这些语言不允许对联合类型的引用做类型检查。另一方面，联合能安全地使用，正如 Ada、ML、Haskell 和 F#语言中设计的联合那样。

Java 和 C#都不包含联合，这反映了编程语言对安全性的关注在不断增长。

6.10.6 联合类型的实现

简单地对每个可能的变体使用相同的地址，就可以实现联合。为最大的变体分配足够

8. let 语句用于给名称赋值，并创建一个静态作用域。在使用 F#交互式解释器时，用两个分号结束语句。

的存储空间。判别式联合的标签会与变体一起保存到一个与记录类似的结构中。

在编译时，每个变体的完整描述必须记录下来。为此，可在描述符中把标签项与实例表相关联。实例表为每个变体提供一项，指向相应变体的描述符。为说明这个过程，考虑以下 Ada 例子：

```
type Node (Tag : Boolean) is
  record
  case Tag is
      when True => Count : Integer;
      when False => Sum : Float;
  end case;
 end record;
```

此类型的描述符的形式如图 6-9 所示。

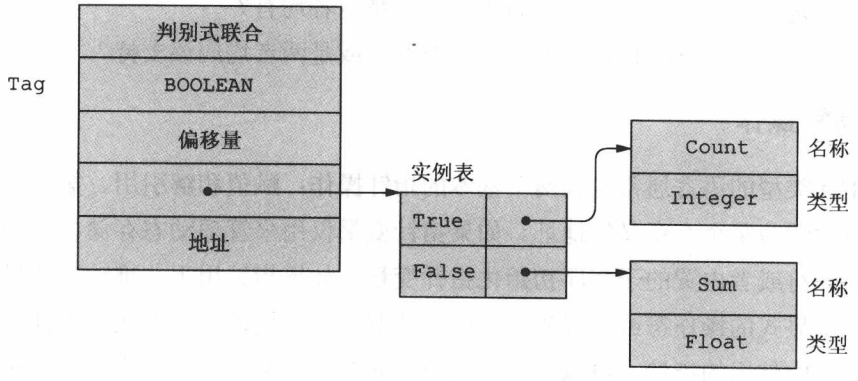


图 6-9　判别式联合的编译时描述符

6.11　指针和引用类型

指针类型变量的取值范围是由内存地址的值和一个特殊的值 nil 构成的。nil 并不是一个有效的地址，它只表明该指针目前无法用来引用内存地址。

指针有两种不同的用途。首先，指针提供了间接寻址的功能，这在汇编语言编程中经常用到。其次，指针提供了一种管理动态存储空间的方式。指针可用来存取名为堆的动态分配区域。

动态分配在堆上的变量称为堆动态变量。它们往往没有与其相关联的标识符，因此只能用指针或引用类型的变量来引用。没有名字的变量称为匿名变量。在第二个指针应用领域中，出现了最重要的设计问题。

指针不像数组和记录，尽管它使用类型操作符(在 C 和 C++中为*，在 Ada 中为 access)定义，但它并非结构化类型。此外，它也不同于标量变量，因为它用来引用其他变量，而不是存储数据。这两种变量分别称为引用类型和值类型。

指针的两种用途为语言增加了可写性。例如，假设需要用一种没有指针的语言(如 Fortran 77)来实现动态数据结构(如二叉树)。这要求程序员提供并维护大量可用的树节点，这些节点可能是由平行数组来实现的。同时，由于 Fortran 77 缺少动态存储机制，程序员就必须猜测所需节点数的最大值。显然，这是一种笨拙、容易出错的处理二叉树的方式。

6.11.6 节讨论的引用变量与指针密切相关。

6.11.1 设计问题

指针的主要设计问题如下：

- 指针变量的作用域和生存期是什么？
- 堆动态变量(指针引用的值)的生存期是什么？
- 是否限制指针所指向的值的类型？
- 指针是用于动态存储管理，间接寻址或是两者均有？
- 语言是否应该支持指针类型，引用类型，或是两者均应该支持？

6.11.2 指针操作

提供指针类型的语言通常包括两个基本的指针操作：赋值和解引用。第一个操作将指针变量的值设置为某个有意义的地址。如果指针变量仅用来管理动态存储，空间分配机制就通过由操作符或者内置的子程序初始化指针变量。如果指针用于非堆动态变量的间接寻址，就必须用显式的操作符或内置的子程序来获取变量的地址，接着赋予指针变量。

对表达式中使用的指针，可以用两种完全不同的方式来理解。首先，可理解为对它绑定的内存单元内容的引用，这种情况下，指针就是一个地址。这正是表达式中的非指针变量的解释方式，虽然在这种情况下，非指针变量的值可能并不是一个地址。然而，指针也可以理解为它绑定的内存单元所指向的另一个内存单元的值的引用。这种情况下，指针是间接引用。前一种情况是正常的指针引用，后者是解引用指针的结果。解引用经过一级间接层次来引用，是第二种基本的指针操作。

指针解引用可以是明确的或隐含的。在 Fortran 95+中它是隐含的，但在许多其他流行的语言中，它必须是明确指定的。在 C++中，它用前置的一元运算符*来明确地指定。考虑下面解引用的例子：如果 ptr 是一个指针变量，值是 7080，而地址为 7080 的单元存储的数据是 206，那么表达式

```
j = *ptr
```

会将 j 的值设定为 206。这个过程如图 6-10 所示。

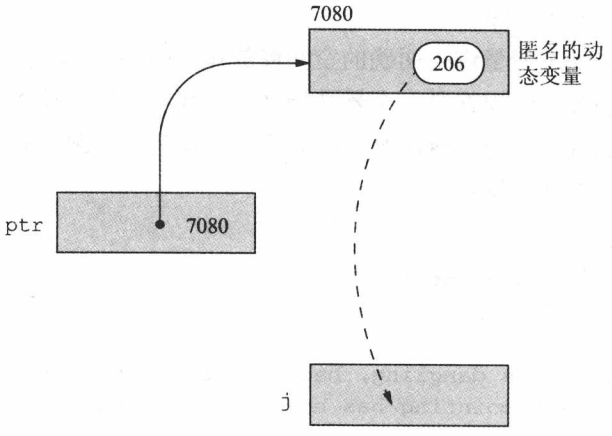


图 6-10　赋值操作 j=*ptr

当指针指向记录时，各种编程语言中引用记录中数据域的语法各有不同。在 C 和 C++中，指向记录的指针引用该记录中的数据域有两种方法。如果指针变量 p 指向含有 age 数据域的记录，(*p).age 可以用来指向 age 数据域。->运算符在指向记录的指针与该记录的某数据域之间使用，首先对指针解引用，然后引用记录的数据域。例如，表达式 p->age 与(*p).age 的效果是相同的。在 Ada 中，也可以使用 p.age，因为在这种情况下，指针会隐式地解引用。

为管理堆而提供指针的语言必须包含一个显式空间分配操作。空间分配有时用子程序(如 C 中的 malloc)来指定。在支持面向对象的编程语言中，堆对象的空间分配常常用 new 操作符来指定。C++没有提供隐式的解引用操作，所以使用 delete 作为其解分配操作符。

6.11.3　指针的相关问题

最先引入指针变量的高级语言是 PL/I。在 PL/I 中，指针可指向堆动态对象和其他程序变量。PL/I 的指针是非常灵活的，但它们的使用可能导致一些编程错误。这些错误中的一部分，在后来的编程语言中仍然存在。一些较新的编程语言，例如 Java，用引用完全取代了指针，并利用其隐式内存释放解决了指针的主要问题。引用类型其实只是操作有限的指针。

6.11.3.1　悬挂指针

悬挂指针(dangling pointer)或悬挂引用是指向内存空间已释放的堆动态变量的指针。悬挂指针是危险的，这有几个原因：首先，它指向的内存空间可能已重新分配给新的堆动态变量。如果这个新变量的类型与原变量不同，悬挂指针的类型检查将失效；即使新动态变量仍然是相同的类型，它的值也与原变量的值毫无关系。其次，若悬挂指针用于更改堆动态变量，新堆动态变量的值就会删除。最后，它指向的位置现在可能由内存管理系统临时使用，如用作指向一串可用的内存空间。此时修改它的值将导致内存管理出错。

在许多语言中，执行如下操作可以创建一个悬挂指针：

(1) 创建一个堆动态变量，然后令指针 p1 指向它。

(2) 将 p1 的值赋给指针 p2。

(3) p1 指向的堆动态变量进行明确的空间释放操作(p1 可能设置为 nil)，但是这个操作并没有改变 p2，p2 现在是一个悬挂指针。如果这个释放空间的操作没有改变 p1，那么 p1 和 p2 都是悬挂指针(当然，这是一个别名的问题，p1 和 p2 都是别名)。

例如，在 C++中有如下操作：

```
int * arrayPtr1;
int * arrayPtr2 = new int[100];
arrayPtr1 = arrayPtr2;
delete [] arrayPtr2;
// Now, arrayPtr1 is dangling, because the heap storage
// to which it was pointing has been deallocated.
```

在 C++中，arrayPtrl 和 arrayPtr2 现在都是悬挂指针，因为 C++的 delete 操作符对于其操作数指针的值没有影响。在 C++中，通常在 delete 操作符的后面执行赋 0 操作，即把 0(代表“空”)赋予释放了空间的指针，这将比较安全。

注意，动态变量的显式释放空间操作是产生悬挂指针的原因。

历史注释

Pascal 包含一个显式的解除分配操作符 dispose。由于 dispose 会产生悬挂指针，因此程序包含 dispose 时，一些 Pascal 实现方案就简单地忽略它。尽管这有效地防止了悬挂指针，但它也不允许重用程序不再需要的堆存储空间。前面讲过，Pascal 最初是设计用于教学的语言，而不是一种行业工具。

6.11.3.2 丢失的堆动态变量

丢失的堆动态变量是已分配空间的堆动态变量，但用户程序不再能访问它。这样的变量常称为垃圾变量(garbage)，因为它们不能用于最初的目的，也不能重新分配空间以在程序中用于新的用途。丢失的堆动态变量大都由以下操作创建：

1 .使指针 p1 指向一个新创建的堆动态变量。

2. 使指针 p1 指向另一个新建的堆动态变量。

第一个堆动态变量现在是不可访问的，或丢失的。这有时叫做内存泄漏。无论语言使用显式还是隐式的解除分配操作，内存泄漏都是个问题。后面的各节将研究语言的设计者如何解决悬挂指针和堆动态变量的丢失问题。

6.11.4 Ada 中的指针

Ada 中的指针称作访问类型。至少在理论上，Ada 的设计部分解决了悬挂指针的问题。在指针类型的作用域结束时，堆动态变量会隐式地解除分配空间(由实现者决定的)，因此显著减少了对显式解除分配空间的要求。但几乎没有 Ada 编译器实现了这一垃圾收集方式，因此，它的优点就几乎只有在理论上成立了。由于堆动态变量只能由一种类型的变量来访问，所以当该类型声明的作用域结束时，就没有指针指向动态变量了。由于不正确的显式解除分

配空间是悬挂指针的主要起因，所以这样悬挂指针的问题就减轻了许多。但是，Ada 语言也有一种显式的解除分配符 Unchecked_Deallocation。它的名字是为了减少它的使用，或者至少提醒使用者，使用它会出现一些潜在的问题。Unchecked_Deallocation 会产生悬挂指针。

堆动态变量的丢失问题在 Ada 语言的指针设计中没有得到解决。

6.11.5　C 和 C++中的指针

C 和 C++中的指针可以像汇编语言中的地址一样使用。这就意味着指针有相当的灵活性，但使用时必须格外小心。这种设计没有提供悬挂指针和堆动态变量丢失问题的解决办法。然而，C 和 C++中能够进行指针运算，使指针比其他编程语言更有吸引力。

C 和 C++中的指针可以指向存储在任何地方的任何变量。实际上，指针可以指向内存中的任意地方，无论那个地方是否有变量，这也是这类指针很危险的原因之一。

在 C 和 C++中，星号(*)表示解引用操作，而&运算符表示取一个变量的地址。例如下面的代码：

```
int *ptr;
int count, init;
...
ptr = &init;
count = *ptr;
```

变量 ptr 的赋值语句把 ptr 设置成 init 的地址。count 的赋值语句对 ptr 解引用，得到 init 变量中的值，然后把它赋给 count。所以这两个赋值的作用是将 init 的值赋给 count。注意指针的声明指定了它的作用域类型。

注意上面的两个赋值语句对 count 的作用效果和下面的赋值语句是一样的：

```
count = init;
```

可将一个正确类型的地址赋给指针，或者也可以赋零值，这用于 nil。

在某些限制的形式里也可以用指针算术。例如，如果 ptr 是一个指针变量，它指向某个数据类型的变量，那么

```
ptr + index
```

是一个合法的表达式，这个表达式的语义如下：不是简单地给 ptr 加一个 index 值，而是先将 index 的值乘以 ptr 指向的数据类型所占用的内存大小。例如，如果 ptr 指向一个占 4 个内存单元大小的类型，index 就乘以 4，然后给 ptr 加上这个乘积。这种地址算术的主要作用是数组操作。下面的讨论只和一维数组相关。

在 C 和 C++中，所有的数组都用 0 作为数组下标的下界，不带下标的数组名总是表示第一个元素的地址。考虑下面的声明：

```
int list [10];
int *ptr;
```

考虑这个赋值：

```
ptr = list;
```

把 list[0]的地址赋给 ptr，因为不带下标的数组名会解释为数组的首地址。给出这个赋值后，下面的表述就是正确的：

- *(ptr+1)和 list[1]是等价的；
- *(ptr+index)和 list[index]是等价的；
- ptr[index]和 list[index]是等价的。

从这些表述可以看出，指针操作和下标操作具有同样的效果。而且，指向数组的指针可以通过下标索引来访问，就好像它们是数组名一样。

C 和 C++中的指针可以指向函数。这个特性用于将函数作为参数传递给另一个函数。指针还可以用于参数传递，参见第 9 章。

C 和 C++包含了类型为 void*的指针，它可以指向任何类型的值。它们实际上是通用的指针。然而，类型检查不是 void*型参数的问题，因为这种语言不允许解引用。void*指针的一个一般用法是作为在内存中起作用的函数的参数类型。例如，假设函数需要将字节序列从内存中的一个位置移到另一个位置，则传递两个任意类型的指针是最常用的。如果函数中相应的形参是 void*类型，这将是合法的。无论将什么类型的指针作为实参传递，都可以将它们转换为 char*类型。

6.11.6 引用类型

引用类型的变量和指针一样，只有一个基本的重要区别：指针指向内存中的地址，而引用指向内存中的对象或值。因此，对地址执行算术操作是很自然的，但不能对引用执行这种操作。

C++包含了一个特殊的引用类型，它主要用于函数定义中的形参。C++引用类型的变量是一个常量指针，它总是隐式地解引用。因为 C++引用类型的变量是常量，所以必须在其定义中用某个变量的地址进行初始化，初始化之后，引用类型的变量也不能设置为引用任何其他变量。隐式的解引用禁止给引用变量赋予地址值。

在定义中，引用类型的变量是用名字前面的&符号指定的。例如：

```
int result = 0;
int &ref_result = result;
...
ref_result = 100;
```

在这个代码段中，result 和 ref_result 都是别名。

C++引用类型在函数定义中作为形参时，为调用函数和被调用函数提供了双向沟通。这在非指针的基本参数类型中是不可能的，因为 C++参数是按值传递的。将指针作为参数传递就可以进行双向沟通，但指针形参需要明确的解引用，降低了代码的可读性和安全性。在被调用的函数中引用的引用参数与其他参数是完全一样的。调用函数不需要说

明对应形参是引用类型的实参有什么特别的地方。编译器传给引用参数的是地址，而不是值。

为了满足比 C++更高的安全要求，Java 的设计者去掉了 C++风格的指针。与 C++引用变量不同，Java 引用变量可以指向不同的类实例，它们不是常量。所有的 Java 类实例都由引用变量来引用，事实上，这也是 Java 引用变量的唯一用途。这些问题详见第 12 章。

请看如下代码，String 是一个 Java 标准类：

```
String str1;
...
str1 = "This is a Java literal string";
```

在这段代码中，str1 定义为一个指向 String 类实例或对象的引用。它最初设置为 null。随后的赋值语句将 str1 指向一个包含 This is a Java literal string 的 String 对象。

由于 Java 类对象是隐式释放的(Java 中没有显式释放的运算符)，所以 Java 没有悬挂引用。

C#包含了 Java 中的引用和 C++的指针，但是它强烈推荐不要使用指针。实际上，任何使用指针的子程序都必须加上 unsafe 标签。因为尽管引用指向的对象是隐式释放的，但是指针指向的对象就不是这样了。C#包含指针只是为了允许 C#程序与 C 或 C++程序合作。

在 Smalltalk、Python、Ruby 和 Lua 等面向对象语言中，所有变量都是引用。它们总是隐式地释放，而且，这些变量的值都是不能访问的。

6.11.7　评价

悬挂指针与垃圾变量的问题已经讨论很久了。堆管理的内容参见 6.11.8.3 节。

指针可以与 goto 语句相比：goto 语句拓宽了下一步可以执行的语句范围；而指针变量拓宽了变量可以指向的内存空间的范围。也许对指针最严厉的批评就是 Hoare(1973)所说的，“将指针引入高级语言是我们无法挽回的退步”。

另一方面，在一些应用程序中，指针是必不可少的。例如编写设备驱动程序时，需要使用指针访问特定的绝对地址。

Java 与 C#中的引用提供了指针的某些灵活性与能力，同时避免了一些容易出错的方面。程序员是否愿意牺牲 C 和 C++指针的强大功能，以换取引用的较强的安全性能，这还有待观察。C#程序使用指针的范围就是这样一项措施。

6.11.8　指针和引用类型的实现

在众多编程语言中，指针用于堆管理。同样，Java 和 C#中的引用，Smalltalk 和 Ruby 中的变量也用于堆管理，所以不应将指针和引用分开讨论。下面首先简单介绍指针和引用在内部是怎样表示的，然后讨论解决悬空指针的两种可能的方法，最后描述堆管理技术的主要问题。

6.11.8.1 指针和引用的表示

在大多数大型计算机中，指针和引用是存储在内存单元中的单个值。然而，以 Intel 微处理机为基础的早期微型计算机中，地址由两部分组成：片段和偏移。所以指针和引用在这些系统中实现为以十六比特为单元的对，每一项用于地址的一个部分。

6.11.8.2 悬挂指针问题的解决方法

目前有一些悬挂指针的推荐解决办法。其中含有墓碑法(tombstone)(Lomet，1975)：在每个堆动态变量中加入一个叫做墓碑的特殊单元，墓碑本身是一个指向堆动态变量的指针。实际的指针变量只指向墓碑，而不指向堆动态变量。当释放堆动态变量时，墓碑留下但是赋值为 nil，表示堆动态变量不再存在。这种方法阻止了指针指向已释放的变量。若一个指针指向 nil 墓碑，则引用它就是一个错误。

墓碑法在时间和空间上的花销很大，因为墓碑从不释放，它们所占的内存不会回收。每次通过墓碑访问堆动态变量，都额外需要一个间接层次，这增加了大多数计算机的运行量。显然，流行语言的设计者认为，这种方法带来的安全性抵不过它增加的花销，因为广泛应用的语言都不使用墓碑法。

在 UW-Pascal(Fischer 和 LeBlanc，1977，1980)的实现中，使用锁键法(locks - and - keys approach)替代了墓碑法。在这个编译器中，指针值表示为有序对(键，地址)，其中键是整数值。堆动态变量表示为变量的存储空间加上一个储存整数锁值的头单元。给堆动态变量分配空间时，就创建锁值，并放在堆动态变量的锁单元中和 new 调用指定的指针的键单元中。每次访问解引用的指针，都要比较指针的键值与堆动态变量的锁值。如果它们相配，访问就是合法的，否则这个访问就当作一个运行时错误。将一个指针值复制给另一个指针时，必须复制键值。因此，给定的堆动态变量可以由任意个指针引用。用 dispose 释放堆动态变量时，会清理它的锁值，把它变为一个非法锁值。因此，如果释放不是 dispose 指定的指针，它的地址值仍然是完好的，但是它的键值将不再与锁值相配，所以不允许进行该访问。

当然，悬挂指针问题的最好解决方法，是不让程序员释放堆动态变量。如果程序不能显式地释放堆动态变量，就不会存在悬挂指针的问题了。为此，运行时系统必须在堆动态变量不再有用时，隐式地释放它们。LISP 系统总是这么做。Java 和 C#也都用这个方法处理引用变量。C#指针并不包括隐式释放操作。

6.11.8.3 堆管理

堆管理是个十分复杂的运行时过程。我们分两种情况探讨这个过程：一种是所有堆存储单元都以同一大小为单位进行分配和释放，另一种是以大小可变的片段来分配和释放。注意，对于存储空间的释放过程，我们只讨论隐式方法。这个讨论很简略，很不全面，因为对这些过程及其相关问题的透彻分析，和它的实现问题一样，并不是一个语言设计问题。

1. 大小固定的存储单元

最简单的情况是所有的分配和释放工作是对单一大小的存储单元进行。如果每个存储单

元包含一个指针，情况就会简单得多。这就是 LISP 的许多实现采用的方案，在 LISP 语言中，存储空间的动态分配问题首次大面积出现。所有 LISP 程序和大多数 LISP 数据都包含链表形式的存储单元。

在大小固定的分配堆空间中，所有可用的存储单元都使用单元中的指针连接起来，形成一个可用空间的链表。分配工作很简单：当需要存储单元时，就从这个表中取得所需数目的存储单元。释放过程则复杂得多。多个指针可以指向同一个堆动态变量，因此很难判断程序何时不再使用这个变量。仅因为一个指针与一个存储单元分离，显然并不会把该存储单元变成垃圾；可能存在其他仍然指向该存储单元的指针。

在 LISP 中，程序中几个最常用的操作会创建一批该程序不再可访问的存储单元，因此应释放这些存储单元(退回可用空间的链表)。LISP 的一个基本设计目标是：保证回收不再使用的存储单元不是程序员的工作，而是运行时系统的工作。这个目标给 LISP 的实现人员带来了一个基础性的设计问题：应在何时执行释放操作？

可以采用几种不同的方法来回收垃圾。两种比较传统的方法在某种程度上是截然相反的。其一称作引用计数器，在创建无法访问的内存时，就增加其回收计数；其二称为标记-清除(mark-sweep)算法，仅在可用空间的列表为空时，才回收内存。这两种方法有时分别称作“积极方法”和“消极方法”。这两种方法还衍生出许多变体。但是，本节只讨论两种基本算法。

回收内存的引用计数法，是在每个存储单元中维护一个计数器，其中存储了当前指向该存储单元的指针数。当指针和存储单元断开连接时，就在递减操作中嵌入一个检查操作，检查引用计数器是否为 0。如果引用计数器是 0，就表示没有程序指针指向这个存储单元，它因此成为垃圾，可以返回可用空间的列表。

引用计数器方法有三个不同的问题。第一，如果存储单元很少，计数器所需的空间就很大。第二，维护计数器的值显然需要占用一些运行时间。每次指针的值改变时，指针以前指向的存储单元中的计数器都需要递减 1，而指针现在指向的存储单元中的计数器都需要递增 1。在像 LISP 这样的语言中，几乎每个动作都需要改变指针，维护计数器的值所需的运行时间就占程序总执行时间的很大一部分。当然，如果指针改变得不十分频繁，就不存在这样的问题。使用延迟引用计数器可以部分去除引用计数器的低效问题，这种方法避免了将引用计数器用于某些指针。第三，在一组存储单元环形连接时，会产生问题。在环形列表中，每个存储单元的引用计数器值都至少是 1，所以无法回收到可用空间的列表中。这个问题的一种解决方法可以在(Friedman 和 Wise，1979)中找到。

引用计数器的好处是，它在本质上是递增的。它的行为与应用程序的行为交织在一起，所以在应用程序运行的过程中不会造成太长的延迟。

垃圾回收的初始标记-清除过程如下：运行系统根据需求分配内存，并在需要时断开指针与存储单元，不考虑内存的重用(允许垃圾堆积)，直到分配了所有可用的存储单元。此时，标记-清除过程开始收集残留在堆上的垃圾。为便于完成这个过程，每个堆存储单元都有一个额外的指示字节或指示域，供回收算法使用。

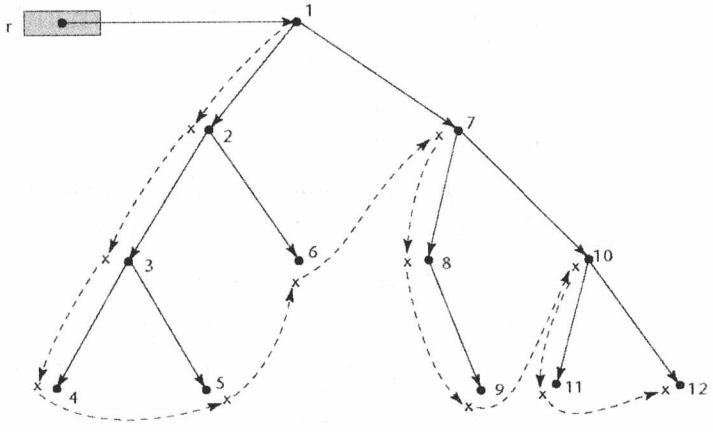
标记-清除过程有三个不同的阶段。首先，堆中的每个存储单元都设置了指示器，指示哪些是垃圾。当然，这个假设只对一些存储单元是正确的。第二步称为标记阶段，是最困难的一步。程序中的每个指针都跟踪到堆中，将所有能访问的单元都标记为可用的。之后，执行第三阶段，它称为清除阶段：堆中没有标记为仍在使用的存储单元都返回到可用空间的列表中。

为了说明算法如何标记当前还在使用的存储单元，下面列出标记算法的一个简单版本。假定所有堆动态变量或堆单元都含有信息部分；标记部分 marker；两个指针 llink 和 rlink。这些存储单元用来建立有向图，其中，任意一个节点至多引出两条边。标记算法将遍历图中的所有跨越树，标记所有找到的单元。像其他的图遍历那样，标记算法也使用了递归。

```
for every pointer r do
    mark(r)

void mark(void * ptr) {
    if(ptr != 0)
      if(*ptr.marker is not marked) {
        set *ptr.marker
        mark(*ptr.llink)
        mark(*ptr.rlink)
      }
}
```

图 6-11 是这个过程在给定图中的运行示例。这个简单的标记算法需要占用很多存储空间(因为需要栈来支持递归)。Schorr 和 Waite(1967)开发了一种不需要增加栈空间的标记过程。这个方法跟踪所链接的结构时，颠倒了指针的顺序。然后，当到达列表的尾部时，该过程可以跟着指针倒回到结构中。



虚线表示标记节点的顺序

图 6-11 标记算法的运行实例

初始版本的标记-清除过程最严重的问题是它们执行的次数太少了——只有当程序使用了全部或几乎所有的堆存储单元时，才执行该过程。在这种情况下，标记-清除过程将耗费

更长时间，因为必须跟踪绝大多数单元，并标记为当前正在使用。这在应用程序进程中将造成明显的迟缓。另外，这个过程可能只产生少量可放在可用空间列表的存储单元。为此，人们进行了很多改进，解决了这个问题。例如，增量标记-清除过程在内存远远没有消耗完之前，会更频繁地回收垃圾，重新声明的存储空间较多，所以该过程比较高效。另外，每次运行该过程所需的时间都明显缩短，减少了应用程序执行过程中的延迟。另一种替代方法是，在不同时间对部分内存执行标记-清除过程，而不是对与应用程序相关的全部内存执行该过程，其改进程度与增量标记-清除过程相同。

标记-清除方法的标记算法和引用计数器方法所需的过程，都可以使用 Suzuki(1982)描述的指针旋转和滑动操作来提高效率。

2. 可变大小的存储单元

管理一个分配可变大小的存储单元[9]的堆，不仅具有管理固定大小存储单元的所有困难，还有额外的问题。遗憾的是，大部分编程语言都需要可变大小的存储单元。可变大小的存储单元所引起的额外问题取决于使用的方法。如果使用标记-清除方法，就会出现下列额外的问题：

- 堆中所有存储单元的指示器的初始设定，很难指示这些单元是垃圾。因为单元的大小是不同的，扫描它们就是一个问题。一种解决方法是要求每个单元的第一个域存放单元的大小。这样就可以进行扫描，虽然它比固定大小单元的同样操作需要略多的空间和时间。
- 标记过程是很重要的。如果没有在存储单元中预先定义指向它的指针的位置，怎么能跟随这个指针生成链呢？完全不包括指针的存储单元也是一个问题。在每个存储单元中加入一个内部指针，这个指针在后台由运行库系统来维护，就可以解决这个问题。然而，这个后台维护的过程增加了程序运行的空间和时间开销。
- 维护可用空间的列表，也是有开销的。列表最初可以只有一个存储单元，其中包含所有的可用空间。请求内存空间段，仅仅减小这块内存的大小。收回的存储单元会添加到这个列表中。问题是不久后，这个表就会变成一个包含许多不同大小的块的长列表。这会降低内存分配的速度，因为请求内存空间时，这个列表需要寻找足够大的内存块。最终，这个列表会由很多非常小的块组成，它们对大多数请求来说不够大。这时，相邻的内存块就需要重新组合为更大的内存块。另一种方法是使用列表中第一个足够大的内存块，这样可以缩短搜索时间，但是列表需要按照内存块的大小排序。在这两种情况中，维护这个表都有额外的开销。

如果使用引用计数器，可以避免前两个问题，但是维护可用空间列表的问题依然存在。要深入地学习内存管理问题，请参阅 Wilson(2005)。

---

9. 这些存储单元有不同的大小，因为它们是抽象的存储单元，存储了变量的值，而不关心它们的类型。此外，变量可以是结构化的类型。

6.12 类型检查

为了讨论类型检查，需要一般化操作数和运算符的概念，以包括子程序和赋值语句。子程序可以看成运算符，它们的参数为操作数。赋值符号可以看成二元运算符，目标变量和表达式是操作数。

类型检查是保证运算符的操作数具有兼容的类型的过程。兼容的类型是指，它对运算符合法，或者在语言规则中允许由编译器生成的代码或解释器隐式转换为合法的类型。这个自动转换称为强制转换。例如，如果在 Java 中对一个 int 变量和一个 float 变量执行相加操作，int 变量的值会强制转换成 float 型，执行浮点数的加法运算。

类型错误是运算符用来操作类型不合适的操作数的情况。例如，在 C 的初始版本中，如果把 int 类型的值传递给接收 float 参数的函数，就会发生类型错误(因为该语言的编译器不检查参数的类型)。

如果在一种语言里，变量对类型的绑定都是静态的，类型检查就几乎总能够静态地进行。动态类型绑定要求在程序运行期间进行类型检查，这称为“动态类型检查”。

有些语言(如 JavaScript 和 PHP)只允许进行动态类型检查，因为它们是动态类型绑定。在编译期间进行类型检查比在运行期间要好，因为错误改正得越早，所花的代价也就越少。静态类型检查的代价是减小了程序的灵活性，可能改进的方法或技巧会少一些。不过目前一般认为，这类小技巧容易出错，降低了可读性。

当一种语言允许在程序运行的不同时刻在同一个内存单元中存储不同类型的值时，类型检查就变得复杂了。这样的内存单元可以由 Ada 语言的变体记录结构、C 及 C++中的联合和 ML、Haskell 和 F#中的辨别联合来创建。在这些情况下，如果要做类型检查，就必须是动态类型检查，而且要求运行库系统维护这些内存单元的当前值的类型。因此，在像 C++这样的语言里，即使所有的变量都是静态类型绑定，在静态类型检查时也不一定能发现所有类型错误。

6.13 强类型化

在 20 世纪 70 年代进行的所谓“结构化程序设计改革”中，一个很著名的语言设计思想是强类型化。强类型化作为价值很高的语言特点被广泛认可。但遗憾的是，它的定义经常是不严格的，而且经常是根本不定义就在计算机文献中使用。

如果总能检查出类型错误，那么这种语言就是强类型化的。这要求所有操作数的类型都可以确定，不论是在编译期间还是在运行期间。强类型化的重要性体现在，它能够检查所有能够导致类型错误的变量滥用。强类型化的语言也允许在运行期间检查那些可以存放不同类型的变量的类型错误。

Ada 几乎是强类型化的。说它“几乎”是强类型化的，是因为它允许程序员抑制对某

种类型转换进行类型检查，而破坏了类型检查规则。只有实例化泛型函数 Unchecked_Conversion 的一个调用时，才能进行这种类型检查的临时抑制。这类函数可以为任何子类型实例化。它取出其参数类型的值，并返回这个参数当前值的位串，不进行实际的转换；它仅用来提取一种类型的变量值，并被作为另一个不同的类型。这种转换有时称为非变换的类型转换。未检查的转换可以用于用户定义的存储空间的分配和解除分配操作，在这些操作中，地址处理为整型，但是必须用作指针。因为在 Unchecked_Conversion 中不进行类型检查，所以程序员必须确保从中得到的值是有意义的。

C 和 C++不是强类型化的语言，因为它们包含了没有进行类型检查的联合类型。

ML 是强类型化的，尽管一些函数参数的类型可能在编译期间不能确定。F#是强类型化的。

Java 和 C#尽管是基于 C++的，但它们与 Ada 一样也是强类型化的。类型可以显式转换，这种行为可能导致类型错误。然而，没有隐式转换方法，所以类型错误总是能检测出来。

语言的强制转换规则对类型检查的值有很重要的影响。例如，Java 中的表达式是强类型化的。然而，算术运算符的操作数分别是浮点类型和整型时，是合法的。此时先把整型操作数的值强制转换为浮点型，然后执行浮点运算，这通常是程序员的意图。可是，这种强制转换也会失去强类型化的好处之一——错误检测。例如，假设程序设置了整型变量 a 和 b，以及浮点型变量 d。现在如果程序员想打印出 a+b，但错打成了 a+d，编译器就检测不出这个错误。a 的值会简单地强制转换为浮点型。所以，强类型化的价值被强制转换削弱了。强制转换较多的语言，像 C 和 C++，不如强制转换较少的语言(比如 Ada)可靠，也不如没有强制转换的语言(如 ML 和 F#)可靠。Java 和 C#的赋值类型强制转换数量是 C++的一半，所以它们的错误检测比 C++好，但是仍然没有达到 ML 和 F#的效率。强制转换的问题详见第 7 章。

6.14　类型等价

类型兼容性的概念是在引入类型检查问题时定义的。兼容性规则规定了对每个运算符都可接受的操作数类型，并因此指明了语言可能的类型错误[10]。这些规则之所以称为兼容性，是因为在有些情况下，操作数的类型能由编译器或者运行库系统隐式地转换为运算符可接受的类型。

对于预定义的标量类型，类型兼容性规则非常简单、严格。但是，对于结构化类型，例如数组、记录和一些用户自定义的类型，兼容性规则要复杂得多。这些类型的强制转换是很少见的，所以其问题不是类型兼容，而是类型等价。就是说，如果在表达式中，不使用强制类型转换，某一操作数能由一种类型替换成其他一种类型，这两种类型就是等价的。类型等价是类型兼容性的一种严格形式——不使用强制类型转换的兼容性。这里的中心问

10. 类型兼容性也是子程序调用中的实参和子程序定义中的形参之间的关系问题。这个问题详见第 9 章。

题在于怎样定义类型等价。

类型等价规则的设计对于语言是很重要的，因为它影响了数据类型及其运算的设计。在这里讨论的类型中，预定义的运算非常少。也许两个变量是等价类型的最重要的结果是两者可以交换它们的值。

有两种方法定义类型等价：名字类型等价和结构类型等价。名字类型等价是说两个变量如果在同一个声明语句中定义，或者用同一个类型名声明，它们就是等价类型。结构类型等价是说两个变量的类型如果有相同的结构，他们就是等价类型。这两种方法有很多的变体，很多编程语言也组合使用它们。

名字类型等价很容易实现，但限制较大。在严格的解释系统下，一个属于整数子区间类型的变量和一个整数类型的变量并不是等同的。例如，假设 Ada 使用严格的名字类型等价，考虑以下的 Ada 程序：

```
type Indextype is 1..100;
count : Integer;
index : Indextype;
```

变量 count 的类型和 index 的类型是不等价的，这两个变量不能互相赋值。

结构化或者用户自定义的类型通过参数在子程序之间传递时，会出现名字类型等价的另一个问题。这种类型必须全局性地定义，且只定义一次。子程序不能在本地语境里声明这种类型的形参。这也是 Pascal 早期版本的一个问题。

注意，要使用名字类型等价，所有的类型必须有名字。多数编程语言允许用户定义匿名类型——它们没有名字。如果一种语言使用了名字类型等价，那么编译器必须隐式地给这些类型指定内部名字。

结构类型等价比名字类型等价更灵活，但更难实现。对于名字类型等价，判断等价只要比较两个类型名。但对于结构类型等价，必须比较两个类型的整个结构。这种比较并不总是很简单(比如有的数据结构指向它自己的类型，例如链表)。同时也会出现其他问题。例如，两个记录(或 struct)的结构相同，而字段名不同时，它们是否等价？在 Fortran 或 Ada 程序中，两个一维数组的元素类型相同，但下标范围不一样，一个是 0..10，另一个是 1..11，它们是类型等价的吗？两个枚举类型的成员数目相同，而字面量的拼写不同，它们是等价的吗？

结构类型等价的另一个困境在于，它不允许区分具有相同结构的各个类型。例如，考虑下面这个类似 Ada 语言的声明：

```
type Celsius = Float;
     Fahrenheit = Float;
```

从结构类型等价的角度看，这两个类型的变量是类型等价的，因而在表达式中可以混用。但在上面的例子中，这肯定不是我们想看到的，因为类型名的不同，表示它们是不同的。一般而言，名字不同的类型可能是不同类别的数据抽象，不应视为等价的。

Ada 语言使用了一种限定形式的名字类型等价，但提供了两种类型结构——子类型和

派生类型——来避免名字类型等价的相关问题。派生类型是一种新类型，它基于以前定义的类型，但与之不等价，尽管它们拥有完全相同的结构。派生类型从它的父类型中继承了所有的属性。看看下面这个例子：

```
type Celsius is new Float;
type Fahrenheit is new Float;
```

尽管这两个派生类型的变量的结构完全相同，但它们不是等价的。同时，它们与其他浮点类型也不是类型等价的。字面量不遵循这一规则。字面量 3.0 一般是实数类型，与任何浮点类型都等价。派生类型也可以包含父类型的范围约束，并继承了父类型的所有操作。

Ada 的子类型是已存在的可能具有范围约束的类型。子类型与父类型等价。例如下面的声明：

```
subtype Small_type is Integer range 0..99;
```

Small_type 类型与 Integer 类型是等价的。

注意，Ada 的派生类型和它的子界类型有很大的不同。例如下面的类型声明：

```
type Derived_Small_Int is new Integer range 1..100;
subtype Subrange_Small_Int is Integer range 1..100;
```

这两种类型 Derived_Small_Int 和 Subrange_Small_Int 的变量的取值范围相同，都继承了 Integer 的操作。但是，Derived_Small_Int 类型的变量和 Integer 类型不兼容。另一方面，Subrange_ Small_Int 类型的变量和 Integer 类型的常量和变量，以及 Integer 类型的子类型兼容。

Ada 非受限数组类型的变量使用了结构类型等价。考虑下面的类型声明和两个对象声明：

```
type Vector is array (Integer range <>) of Integer;
Vector_1: Vector (1..10);
Vector_2: Vector (11..20);
```

虽然这两个对象有不同的名称和下标范围，但它们的类型是等价的，因为对于非受限数组类型的对象，使用的是结构类型等价，而不是名字类型等价。因为这两种类型都有 10 个元素，元素的类型都是 Integer，所以它们是类型等价的。

对于受限匿名类型，Ada 使用了名字类型等价的严格形式。考虑下面受限匿名类型的声明：

```
A : array (1..10) of Integer;
```

在这个例子中，编译器给 A 赋予了一个匿名且唯一的类型，但这个类型不能由程序访问。如果还声明了

```
B : array (1..10) of Integer;
```

则 A 和 B 都是匿名的，但是它们有区别，不是类型等价的，尽管它们在结构上是相同的。多重声明

```
C, D : array (1..10) of Integer;
```

生成两个匿名类型，分别用于 C 和 D，它们是不等价的。这个声明实际上应看作下面的两个声明：

```
C : array (1..10) of Integer;
D : array (1..10) of Integer;
```

注意 Ada 的名字类型等价比本节开始定义的名字类型等价要更加严格。如果写成：

```
type List_10 is array (1..10) of Integer;
C, D : List_10;
```

那么 C 和 D 的类型是等价的。

名字类型等价在 Ada 中工作得很好，部分原因是除了匿名数组以外，所有的类型都要求有类型名(匿名数组由编译器指定内部名字)。

Ada 的类型等价规则比其他有许多类型强制转换的语言更加严格。例如，Java 中加法运算符的两个操作数可以是该语言中几乎任何数值类型的结合，只需要把一个操作数转换为另一个操作数的类型。但是在 Ada 中，算术运算符的操作数没有强制类型转换。

C 使用名字和结构类型等价。每一个 struct、enum 和 union 的声明都创建了一个不与其他任何类型等价的新类型。所以，名字类型等价用于结构体、枚举和联合类型。其他非标量类型使用结构类型等价。如果数组类型的元素类型相同，它们就是等价的。另外，如果数组的大小是固定的，它就与其他有相同大小的数组等价，也与没有固定大小的数组等价。注意 C 和 C++中的 typedef 没有引入新的类型；它只是为已有的类型定义新名字。所以，任何用 typedef 定义的类型都与它的父类型等价。C 中结构体、枚举和联合使用的名字类型等价的一个例外是，如果两个结构体、枚举或联合在不同的文件中定义，就使用结构类型等价。允许在不同文件中定义的结构体、枚举和联合等价，是名字类型等价规则的一个漏洞。

C++与 C 相似，只是定义在不同文件中的结构体和联合不是特例。

在不允许用户定义和命名类型的语言中，比如 Fortran 和 COBOL，显然不能使用名字类型等价。

面向对象的语言(例如 Java 和 C++)带来了另一种类型的兼容问题。这个问题是对象兼容性和它与继承结构的关系，这将在第 12 章中讨论。

表达式中的类型兼容性在第 7 章中讨论，子程序参数中的类型兼容性在第 9 章中讨论。

6.15 理论和数据类型

类型理论是数学、逻辑学、计算机科学和哲学中广泛研究的领域。它在 20 世纪早期始于数学，后来成为逻辑学中的一个标准工具。对类型理论的一般性讨论都非常复杂、冗长、高度抽象。即使仅限于计算机科学，类型理论都包括很多复杂的主题，像类型化的 lambda

计算、连接符、有界量词的元理论，已有的类型以及高阶多态性。所有这些主题都远远超出了本书的讨论范围。

在计算机科学中，类型理论有两个分支：实用的和理论的。实用分支主要研究商业编程语言中的数据类型；理论分支主要研究类型化的 lambda 计算，过去的半个世纪中，理论计算机科学家对它进行了广泛研究。本节简要介绍编程语言里数据类型的数学形式体系。

数据类型定义了一个值集合以及针对这些值的一个操作集合。类型系统是一个类型集以及它们在程序中的使用规则。显然，每种类型化的编程语言都定义了一个类型系统。每种语言的类型系统的形式模型包括一个类型集和一个函数集，其中函数集定义了该语言的类型规则，它用来决定表达式的类型。描述类型系统规则(属性文法)的形式系统参见第 3 章。

属性文法的另一个模式是使用一个类型图和一个函数集，它们与文法规则无关，但指定了类型规则。类型图类似于在标志语义下使用的程序状态，它包含一个有序对集，每个有序对的第一个元素是一个变量名，第二个元素是它的类型。类型图使用程序中的类型声明来创建。在静态类型化的语言中，类型图只需要在编译过程中维护，但在编译器分析程序时，它会改变。如果动态地进行类型检查，类型图就必须在运行期间维持。在编译系统中，类型图的具体版本是主要由词汇和语法分析器构建的符号表。动态类型有时也用与值或者对象相关的标签来维护。

如前所述，数据类型是一个值的集合，但在数据类型中，元素通常是排好序的。例如，所有基数类型的元素都是排好序的。尽管有这个差异，集合运算还是能够用于数据类型，来描述新的数据类型。编程语言的结构化数据类型通过类型标识符定义，或者通过对应集合操作的构造函数来定义。后面将简要介绍这些集合操作和类型构造函数。

有限映射是从一个数值有限集合(域集)到范围值集合的一个函数。有限映射在编程语言中生成了两个不同的类型：函数类型和数组类型，但在某些语言中，函数不是类型。所有语言都有数组的概念，其定义是数组的下标通过映射函数映射到数组中的元素。对于传统的数组，这个映射是很简单的——整型数据映射到数组元素的地址。对于管理数组，该映射通过一个描述散列操作的函数来定义。这个散列函数把关联数组的键值(通常是字符串[11])映射到数组元素的地址上。

n 个集合 S_1、S_2、…、S_n 的笛卡儿乘积或者叉积，是集合 $S_1 \times S_2 \times \ldots \times S_n$。笛卡儿乘积所得集合中的每一个元素是由 S_1、S_2、…、S_n 每一个集合的一个元素组成的集合。所以，$S_1 \times S_2 = \{(x,y) | x$ 来自集合 S_1，y 来自于集合 $S_2\}$。例如，如果 $S_1 = \{1, 2\}$ 而 $S_2 = \{a, b\}$，那么 $S_1 \times S_2 = \{(1, a), (1, b), (2, a), (2, b)\}$。笛卡儿乘积定义了数学上的元组，而元组是 Python、ML 和 F#中的一种数据类型(见 6.5 节)。笛卡儿乘积还可以为记录或结构建模，但不准确。笛卡儿乘积的元素没有名字，但记录需要这些名字。例如下面的 C 结构：

```
struct intFloat {
  int myInt;
  float myFloat;
};
```

11. 在 Ruby 和 Lua 中，关联数组的键值未必是字符串，它可以是任意类型。

这个结构定义了 int×float 的笛卡儿乘积类型。元素的名字是 myInt 和 myFloat。

两个集合 S_1 和 S_2 的并集定义为 $S_1 \cup S_2 = \{x \mid x \text{ 是 } S_1 \text{ 中的元素，或者 } x \text{ 是 } S_2 \text{ 中的元素}\}$。如 6.10 节所述，集合的并集是联合数据类型的模型。

在数学中，子集是通过提供一个元素必须遵守的规则来定义的。它们是 Ada 中子类型的模型，但不完全一致，因为子类型必须包含父类中的连续元素。而数学集合中的元素是无序的，所以这个模型并不完美。

注意，用类型操作符定义的指针，比如 C 中的*，并不是用集合操作来定义的。

这就结束了数据类型的形式体系的讨论，也结束了数据类型的全部讨论。

小　结

语言的数据类型在很大程度上决定了这种语言的风格和用途。数据类型和控制结构构成了语言的核心。

大多数命令式语言的基本数据类型包括数值、字符和布尔型。数值类型通常由硬件直接支持。

用户自定义的枚举类型和子范围类型是十分方便的，提高了程序的可读性和可靠性。

数组是大多数编程语言的一部分。对数组元素的引用和该元素的地址之间的关系在访问函数中给出，这是映射的一种实现方式。数组可以是静态的，如 C++数组定义可以包含 static 说明符；可以储存在固定动态堆栈中，像在 C 语言的函数中(没有 static 说明符)；可以储存在动态堆栈中，像是 Ada 的块；可以是固定动态堆，像是 Java 里的对象；还可以是动态堆，像是 Perl 的数组。大部分语言都只支持对完整数组的一部分操作。

现在大部分语言都有记录类型。记录的域有很多种方式来说明。以 COBOL 为例，可在不命名所有包含记录的情况下引用记录的域，但这会给实现带来混乱，并降低了代码的可读性。在一些支持面向对象编程的语言里，记录由对象来支持。

元组类似于记录，但其元素必须有名称。Python、ML 和 F#支持元组。

列表是函数式编程语言的重要类型，但现在也包含在 Python 和 C#中。

联合能在同一个位置在不同时间里存储多种不同类型的值。判别式联合包括一个记录当前类型值的标签。自由联合不包含这样的标签。除了 Ada、ML 和 F#外，大部分拥有联合的语言都没有为联合提供安全设计。

使用指针可以灵活地寻址，控制动态空间的管理。指针有一些固有的危险性，例如很难避免指针悬挂以及内存泄漏。

Java 和 C#里的引用类型提供了对堆的管理，而不会有指针的危险性。

实现数据类型的难度与该类型是否包含在语言里密切相关。枚举类型、子范围类型和记录类型都相当容易实现。数组也很简单，但当数组拥有几个下标时，对数组元素的访问就比较昂贵。对每个下标的访问，访问函数需要一次加法和一次乘法。

如果不考虑堆的管理，指针相当容易实现。如果所有的内存单元拥有相同的大小，堆的管理也是很容易实现的，但如果内存单元的大小是多样的，存储单元的分配与重分配则

是相当复杂的。

强类型化是指必须检测出所有的类型错误。强类型化的价值在于增加了程序的可靠性。

语言的类型等价规则决定了，在该语言的结构化类型中，什么操作是非法的。名字类型等价和结构类型等价是定义类型等价的两种基本途径。类型理论在许多方面都有了发展。在计算机科学领域，类型理论的实用分支定义了编程语言里的类型和类型规则。集合理论能用来为编程语言里的大部分结构化数据类型建模。

文献注释

在数据类型的设计、使用以及实现方面，有着极为丰富的计算机科学文献。Hoare 在(Dahl 等，1972)中为结构化类型给出了最早的系统定义。Cleaveland(1986)给出了对各种数据类型的一般性讨论。

Fischer 和 LeBlanc(1980)讨论了对 Pascal 数据类型可能具有的非安全性进行运行时检测。大多数关于编译器设计的书，如 Fischer 和 LeBlanc (1991)，以及 Aho 等(1986)都描述了数据类型的实现方法，其他编程语言教材，如 Pratt 和 Zelkowitz(2001)以及 Scott(2000)也都给出了数据类型的实现方法。关于堆管理问题的详细讨论，可在 Tenenbaum 等(1990)中找到。垃圾回收方法由 Schorr 和 Waite(1967)以及 Deutsch 和 Bobrow(1976)开发。关于垃圾回收算法的全面讨论，可在 Cohen(1981)以及 Wilson(2005)文献中找到。

复 习 题

1. 什么是描述符？
2. 小数数据类型具有什么优点和缺点？
3. 字符串类型的设计问题是什么？
4. 描述串长度的三种选择。
5. 定义序数类型、枚举类型和子范围类型。
6. 用户定义的枚举类型有什么优点？
7. C#中用户定义的枚举类型在哪些方面比 C++的枚举类型更可靠？
8. 数组有哪些设计问题？
9. 定义静态数组、固定栈动态数组、栈动态数组、固定堆动态数组以及堆动态数组。每种数组各有什么优点？
10. 在 Perl 中引用一个不存在的数组元素时，会发生什么？
11. JavaScript 是如何支持稀疏数组的？
12. 哪些语言支持负下标？
13. 哪些语言支持带步长的数组切片？
14. Ada 提供了哪些数组初始化特性是其他常见命令式语言中不存在的？
15. 什么是聚集常量？
16. 为什么在 Ada 中要专门为一维数组提供数组操作？

17. 定义按行存放和按列存放。
18. 数组的访问函数是什么？
19. 在 Java 的数组描述符中，哪些是必不可少的？在什么时候(编译时或运行时)必须存储它们？
20. 关联数组的结构是什么？
21. COBOL 记录中级数的作用是什么？
22. 定义对记录中域的完全限定引用和省略引用。
23. 记录和元组的主要区别是什么？
24. Python 元组是可变的吗？
25. F#元组模式的作用是什么？
26. 在哪些命令式语言中，列表用作数组？
27. Scheme 函数 CAR 的操作是什么？
28. F#函数 tl 的操作是什么？
29. Scheme 函数 CAR 用什么方式修改其参数？
30. Python 的列表包含以什么为基础？
31. 定义联合、自由联合和判别式联合。
32. 联合的设计问题有哪些？
33. Ada 中的联合一定会做类型检查吗？
34. F#中的联合是判别式的吗？
35. 指针类型的设计问题有哪些？
36. 指针的两个常见问题是什么？
37. 为什么在多数语言中指针必须严格指向单一类型的变量？
38. 什么是 C++的引用类型，它一般做什么用？
39. 在 C++中作为形参，为什么引用变量比指针更好？
40. 相比于其他语言中的指针，Java 和 C#中引用类型的变量有什么优点？
41. 描述垃圾回收机制中的消极方法和积极方法。
42. 为什么在 Java 和 C#的引用上执行四则运算没有意义？
43. 什么是兼容类型？
44. 定义类型错误。
45. 定义强类型化。
46. 为什么 Java 语言不是强类型化的？
47. 非变换的类型转换是什么？
48. 哪些语言没有类型的强制转换？
49. 为什么 C 和 C++语言不是强类型化的？
50. 什么是名字类型等价？
51. 什么是结构类型等价？

52. 名字类型等价的主要优势是什么？
53. 结构类型等价的主要不足是什么？
54. C 中的什么类型使用结构类型等价？
55. 什么集合操作建模了 C 的 struct 数据类型？

习　题

1. 赞成和反对将布尔值表示为内存中单个二进制位的理由分别是什么？
2. 十进制值是如何浪费内存空间的？
3. VAX 微型计算机的浮点数使用了一种与 IEEE 标准不一样的格式。这种格式是什么？为什么 VAX 计算机的设计者选择了它？Sebesta(1991)是 VAX 浮点表示法的一篇参考文献。
4. 从安全性和实现代价的观点出发，比较用于避免悬挂指针的墓碑法与锁-键法。
5. 指针隐式解引用只存在于某些语境中的缺点是什么？例如，考虑在 Ada 中对一个指向记录的指针隐式解引用，该指针用来引用记录的一个域。
6. 说明 Ada 的子类型和派生类型之间的所有区别。
7. C 和 C++中的->运算符有什么重要的合理性？
8. C++和 Java 中的枚举类型有哪些不同？
9. C 和 C++中的联合数据类型与记录类型是分开的，而不是像 Ada 那样是连在一起的。这两种选择的优缺点分别是什么？
10. 多维数组能够按行存放(如在 C++中)，或者按列存放(如在 Fortran 中)。对这两种储存方式下的三维数组编写相应的访问函数。
11. 在 Burroughs Extended ALGOL 语言中，矩阵是以指针的一维数组来存储的，指针指向矩阵的各行，而矩阵行视为值的一维数组。这种方案的优缺点是什么？
12. 分析并写一份对 C 中 malloc 和 free 函数与 C++中 new 和 delete 运算符的比较报告。在比较中，将安全性作为主要考虑事项。
13. 分析并写一份对使用 C++指针变量和 Java 引用变量访问固定堆动态变量的比较报告。在比较中，将安全性和便利性作为主要考虑内容。
14. 写一份简短报告，讨论 Java 设计者决定不包含 C++指针的得失。
15. 与 C++需要的显式堆存储空间回收相比，赞成和反对 Java 的隐式堆存储空间回收的理由有哪些？请考虑实时系统。
16. 尽管在 Java 早期的几个版本中没有，但 C#仍要包含枚举类型的理由有哪些？
17. 对于 C#中的指针，你期待的使用程度是什么？如果指针并不是完全必要的，那么指针的使用频率将会如何？
18. 列出两个关于矩阵应用的清单，其中一个是关于不规则矩阵的，另一个则是关于矩形矩阵的。然后讨论一下，在编程语言中，是否应该只包含不规则矩阵，或只包含矩形矩阵，亦或两种矩阵都包含进来？

19. 比较一下 C++、Java 和 C#的类库处理字符串的能力。
20. 查阅 Gehani(1983)中给出的强类型化定义，并将其与本章的定义比较。两者有什么不同？
21. 在什么情况下，静态类型检查优于动态类型检查？
22. 解释强制类型转换规则是如何削弱强类型所带来的好处的。

编程练习

1. 设计一组简单的测试程序，判断你用过的某个 C 编译器的类型兼容规则。将你的发现写成一份报告。
2. 判断你用过的某个 C 编译器是否实现了 free 函数。
3. 用一种可以执行下标范围检查的语言编写一个处理矩阵的程序，并从编译器中得到其汇编代码或机器指令的版本。查找出用于下标范围检查的指令数目，并与整个矩阵处理程序的指令总数对比。
4. 假设你有一个可以决定是否使用下标范围检查的编译器，写一个处理大型矩阵存取的程序并记录其运行时间。在使用和不使用下标范围检查两种情况下运行程序，并比较两次运行的时间。
5. 用 C++写一个简单程序，检查枚举类型的安全性。至少包含 10 种对枚举类型的不同操作，判断哪些不正确或可笑的操作是合法的。然后用 C#编写这个程序，运行它，判断有多少不正确或可笑的操作是合法的。比较两者的结果。
6. 用 C++或 C#写一个程序，包含两个不同的枚举类型以及相当数量的使用枚举类型的操作。同时，写一个只使用整型变量的相同功能的程序。比较两者的可读性，并预测两者可靠性的差异。
7. 写一个 C 程序，只使用下标对二维数组元素进行大量引用。再写一个程序，执行相同的操作，但在存储-映射函数中使用指针和指针算术来对数组进行引用。比较这两个程序的效率(时间复杂度)。哪个程序更可靠？为什么？
8. 编写一个 Perl 程序，使用散列和大量对散列的操作。例如，散列可以存储人员姓名和年龄。用一个随机数生成器产生含有三个字符的姓名和年龄，添加到散列中。当产生一个重复的名字时，需要访问一次散列，但不向散列中添加新元素。重新写一个不使用散列的相同程序。比较这两个程序的执行效率。比较它们的编程难易度和程序的可靠性。
9. 自己选择一种语言写一个程序，要求当该语言使用名字等价或结构等价时，程序的表现有差异。
10. 对于简单的赋值语句 A=B，哪些类型的 A 与 B 在 C++中是合法的，但是在 Java 中不合法？
11. 对于简单的赋值语句 A=B，哪些类型的 A 与 B 在 Java 中是合法的，但在 Ada 中不合法？

访　谈

Lua

Roberto Ierusalimschy

Roberto Ierusalimschy 是 Lua 脚本语言的创始人之一，这种语言广泛应用在游戏开发和嵌入系统上。Roberto Ierusalimschy 目前是巴西里约热内卢天主教神学大学计算机系的助理教授(Lua 的更多信息请访问 www.lua.org)。

你第一次接触计算科学是在什么地方？过程是怎样的？

在我 1978 年进入大学之前，对计算科学完全不了解。我记得我曾经读过一本 Fortran 编程语言的书，不过读到关于变量和常量的定义部分时就读不下去了。

大学第一年，我选了一门 Fortran 语言的编程 101 课程。那时我们在 IBM 370 大型机上运行编程作业。我们给卡片穿孔来写代码，用固定的 JCL 卡片围住代码卡片，然后把它交给操作员。过一段时间后(通常是数个小时)得到了结果清单，不过里面经常全是编译错误。

之后我的一个朋友从国外带来一台微型计算机，它的 CPU 是 Z80，有 4KB 内存。我们开始用汇编语言给这台机器编写各种类型的程序——准确地说是用机器语言，因为那台机器上没有汇编器。我们先用汇编语言编写代码，然后手动把它们转化成十六进制，以便输入到计算机中去运行。

从那开始我就被计算科学吸引住了。

过去 25 年，在学术环境中设计的编程语言很少有成功的。尽管你是一位学者，但是 Lua 语言却设计为用于非常实际的应用。你认为 Lua 是一个学术语言，还是一个应用语言？

Lua 当然是一个应用语言，但也具有学术性质。Lua 是为两方面的产业应用而开发的，它也一直应用在产业上。我们在它的设计上做得很务实。不过除了它的第一个版本之外，我们从未感受到来自产业界的压力。我们一直拥有决定新版本发布时间和是否接受用户要求的特权。这给了我们一些其他语言没有的自由。

最近，我们完成了一些关于 Lua 的学术研究。但要把这些学术成果用在正式版本上还要一个漫长的过程，这种成果往往只会对 Lua 有很小的直接影响。不过也有一些令人高兴的例外，例如基于寄存器的虚拟机和“瞬时表格”(将出现在 Lua 5.2 中)。

你曾经说过，Lua 是提升出来的，而不是设计出来的。你能解释一下你这样说的意思吗？你认为这种方法的好处是什么？

我们的意思是，Lua 最重要的部分没有出现在它的第一版中。这个语言开始只是一个很小很简单的语言，然后在它成长的过程中逐渐获得了相关的特性。

在讲这种方法的好处(以及缺点)之前，我先澄清一下：这种方法不是我们选择的。我们从没想过要“培育一种新的语言”，只不过它的产生和发展过程碰巧如此。

我认为设计一种语言最大的困难是：对于平时使用时不同机制之间是如何相互影响的，

要有所预见。通过对语言的不断提升，或者说一步一步地创造它——你能避免大部分相互影响的问题，因为对每个新特性，可以在语言其他部分都各就各位，并在实际应用中由实际用户测试过之后，再考虑它。

当然，这种方法也有一个主要的缺点，某个最需要的特性与已经实现的功能之间存在矛盾，不能相容。

Lua 自从 1994 年第一次发布以来，一直以不同的方式变化。你曾经说过，你有好几次后悔没有在 Lua 中加入布尔类型。为什么你不简单地加一个进去？

虽然听起来很可笑，但我们的确漏掉了“false”值，因为我们并没有使用“true”值。

与最初的 LISP 语言一样，Lua 把 nil 视为 false，而把其余值都当作 true。问题在于 nil 也表示未初始化的变量。这样就无法将未初始化的变量与取 false 值的变量区别开来。所以，我们需要一个 false 值，以便能区分这两种情况。不过，true 值没有什么用处，因为使用 1 或其他任何一个常量就足够了。

我想，这是一个典型的“工业”思想与“学术”思想相冲突的例子。真正实用主义的想法就会毫不犹豫地将布尔类型添加进来。但我们的学术想法会对这种不优雅的做法感到不舒服。最后，实用主义一方获胜，不过这还需要一段时间。

除了预处理器之外，后来被认为是不好的、并从语言中去掉的最重要的 Lua 特性有哪些？

我不记得其他不大好的特性。我们是从 Lua 中去掉了一些特性，但主要是因为它们被新的、常常是某种意义下“更好的”特性替代了。例如：标记方法被元方法替代；C API 中的弱引用被弱表格替代；以及上位值(upvalue)被适当的词法作用域替代。

在考虑一个会破坏向后兼容性的新特性时，如何做出决策？

要做这些决策总是很困难的。首先，我们试着寻找其他某种方式，来避免或者至少是减少不兼容性。如果做不到，我们就试着提供一些简单方法来解决不兼容(例如，如果从核心库中去掉了某个函数，则可能会提供一个单独的实现，让程序员加入到其代码中)。我们还会试着评估检测和更正不兼容性的难度。如果新特性引入了语法错误(如新的保留字)，这并不算糟糕，我们甚至可以提供一个自动化工具来订正旧代码。然而，如果新特性产生了一些难以捉摸的错误(如已有的函数返回一个不同的结果)，则新特性就是不可接受的。

考虑过将 Ruby 那样的迭代方法，而不是以前加入的 for 语句，加入到 Lua 中吗？作出这样的选择，要考虑哪些内容呢？

这些不只是考虑，实际上已经实现了！从 3.1 版(1998)开始，Lua 就有了 foreach 函数，它可以将一个给定的函数应用到表格的所有元素对。同样，使用 gsub 函数，很容易将一个给定的函数应用到字符串中的每个字符上。

Lua 循环体没有使用特殊的“块”机制，而使用了一级函数来完成这个任务。看看下面这个例子：

```
--'t' is a table from names to values
--the next "loop" prints all keys with
values greater than 10
foreach(t, function(key, value)
  if value > 10 then print(key) end
end)
```

不过，我们最初实现迭代器时，Lua 的函数还没有完整的词法作用域。而且，语法也略嫌笨拙(宏会帮助改善这种状况)。此外，退出语句(break 和 return)在迭代器中使用时总是令人迷惑。所以，最终我们决定使用 for 语句。

但是，即便现在函数有了严格的词法作用域，“真正的迭代器”在 Lua 中仍是一个有用的设计。在我关于 Lua 的书中，介绍 for 语句的章节就是以讨论真正的迭代器来结束的。

你称 Lua 是一种可扩展的扩展语言，请简要地叙述一下是什么意思？

Lua 是一个“可扩展的语言”，因为很容易将其他语言定义的函数和类型添加到 Lua 中。所以要扩展它是很简单的。更具体地说，在 Lua 中调用 C 是很简单的。

Lua 是一个“扩展语言”，因为很容易通过 Lua 来扩展应用程序、把 Lua 改造成应用程序的一种宏语言(实际上就是语言的“脚本化”)。更具体地讲，在 C 中调用 Lua 是很方便的。

数据结构已从数组、记录和散列发展到它们的组合。你能评估一下 Lua 的表格在编程语言数据结构的演化进程中的重要性吗？

我不认为 Lua 的表格在其他语言的演化进程中有重要的地位。或许未来会有变化，但我现在还不确定。我的观点是，Lua 表格提供的主要便利是它的简单性，即“多合一”的解决方案。但这种简单性是有代价的，例如，Lua 程序的静态分析是非常困难的，部分原因是它的表格非常一般化，而又无处不在。所以，每种语言都有它各自的优势。

第7章

表达式与赋值语句

7.1 概述
7.2 算术表达式
7.3 运算符重载
7.4 类型转换
7.5 关系表达式和布尔表达式
7.6 短路求值
7.7 赋值语句
7.8 混合模式赋值

如题所示，本章的主题是表达式与赋值语句。我们将首先讨论表达式的语义规则，它们决定了表达式中运算符的运算顺序。接着将解释函数有副作用时，操作数运算顺序的一个潜在问题。然后将讨论重载的运算符，包括预定义的和用户定义的，以及它们在程序中对表达式的作用。接下来描述和评价混合表达式。这将涉及隐式与显式的扩展类型转换与收缩类型转换的定义和评价。随后将讨论关系表达式与布尔表达式，其中包括短路求值的过程。最后，讨论赋值语句最简单的形式和它的所有变式，包括作为表达式的赋值与混合赋值。

字符串的模式匹配表达式在第 6 章作为字符串相关材料的一部分介绍过，故本章不再涉及。

7.1 概述

表达式是编程语言中描述计算过程的基本工具。对程序员来说，谙熟所使用语言的表达式语法和语义是极其重要的。第 3 章介绍了一套形式机制(BNF)来描述表达式的语法。本章将讨论表达式的语义。

为了理解什么是表达式的求值，有必要先熟悉一下运算符和操作数的运算顺序。表达式中运算符的运算顺序，是由语言的结合性与优先级规则规定的。语言的设计者往往没有声明表达式中操作数的运算顺序，尽管一个表达式的值有时取决于该顺序。故允许实现者选择该顺序，从而导致程序在不同的执行下可能得出不同的结果。其他与表达式语义相关的问题还有类型不匹配、强制转换和短路求值等。

命令式编程语言的本质在于赋值语句的主导地位。赋值语句的目的是改变变量的值或程序的状态。所以，这种变量值随着程序的执行而改变的概念，是所有命令式语言中不可或缺的一部分。

函数式语言使用另一种类型的变量，例如函数的参数。这些语言还用声明语句把值绑定到名字上。这些声明类似于赋值语句，但没有副作用。

7.2 算术表达式

自动计算形如数学、自然科学、工程学领域中的算术表达式，是最初高级编程语言的基本目标之一。在编程语言中，算术表达式的多数特点都来源于在数学中形成的惯例。在编程语言中，算术表达式由运算符、操作数、括号和函数调用组成。运算符可以是一元的，表示它只有一个操作数；可以是二元的，表示它有两个操作数；可以是三元的，表示它有三个操作数。

在大多数编程语言中，二元运算符是中缀(infix)，即运算符出现在两个操作数之间。但 Perl 语言是个例外，它有一些运算符是前缀(prefix)，即运算符出现在操作数之前。

算术表达式的目的在于说明一个算术计算。执行一个这样的计算必将引起两个操作：取出操作数(通常是从存储器中取出)；对这些操作数进行算术运算。下面各节将研究算术表达式的一些常见设计细节。

本节将讨论以下算术表达式的基本设计问题：

- 运算符的优先级规则是什么？
- 运算符的结合性规则是什么？
- 操作数的运算顺序是什么？
- 对操作数求值的副作用是否存在约束？
- 语言是否允许用户自定义重载运算符？
- 在表达式中，可以怎样混合类型？

7.2.1　运算符的运算顺序

语言的优先级规则与结合性规则决定了其运算符的运算顺序。

7.2.1.1　优先级

表达式的值至少在一定程度上取决于表达式中运算符的运算顺序。考虑下面的表达式：

```
a + b * c
```

假设变量 a、b、c 的值分别是 3、4、5。如果从左向右运算(先算加法，后算乘法)，结果是 35。若从右向左运算，结果是 23。

很早以前，数学家就建立了这样的概念：将运算符置于一套运算优先级的等级制度下，并在一定程度上基于这一等级制度来决定表达式的运算顺序，而不是简单地按从左向右或从右向左的顺序对运算符进行运算。例如，在数学中，乘法的优先级比加法高，也许是因为乘法比加法复杂吧。如果将这一约定运用在上面的示例表达式中，乘法就先执行，这与大多数编程语言的约定是一致的。

表达式求值中，运算符的优先级规则定义了优先级不同的运算符的运算顺序。在语言的设计者眼中，表达式中运算符的优先级规则是基于运算符的优先级等级制度的。在常见的命令式语言中，运算符的优先级规则几乎完全一样，因为它们都源于数学上的那些规则。在这些语言中，幂运算的优先级是最高的(如果该语言提供这种运算)，次之是优先级相同的乘、除运算，再次之是优先级相同的二元加法和二元减法运算。

许多语言还包含了一元的加法和减法运算符。一元加法称为恒等算子，因为它一般没有关联的操作，故对其操作数没有任何作用。Ellis 和 Stroustrup(1990，p.56)在谈起 C++时，称它是一个历史上的巧合，歪打正着，正确地将它置成了一个无作用的运算符。当然，一元减法会改变操作数的符号。在 Java 和 C#中，一元减法运算符还会将 short 和 byte 类型的操作数隐式地转换为 int 类型。

在所有常见的命令式语言中，一元减法运算符可以出现在表达式的开头或表达式中的任何位置，只要用括号将它和相邻的运算符分开。例如：

```
a + (-b) * c
```

是合法的，但是

```
a + - b * c
```

通常是不合法的。

接下来，考虑以下表达式：

```
- a / b
- a * b
- a ** b
```

前两种情况，一元减法运算符与二元运算符的优先级孰先孰后无关紧要——因为这两个运算符的运算顺序并不影响表达式的值。但是最后一种情况下，运算符的运算顺序就会影响表达式的值了。

在常见的编程语言中，只有 Fortran、Ruby、Visual Basic 和 Ada 有幂运算符。在这四种语言中，幂运算较之一元减法都有更高的优先级，故

```
- A ** B
```

等价于

```
-(A ** B)
```

在 Ruby 和基于 C 的语言中，算术运算符的优先级如下：

	Ruby	基于 C 的语言
最高优先级	**	后缀++、--
	一元运算符+、-	前缀++、--、一元运算符+、-
	*、/、%	*、/、%
最低优先级	二元运算符+、-	二元运算符+、-

运算符**表示幂运算。运算符%有两个整数操作数，返回第一个操作数被第二个操作数除所得的余数[1]。基于 C 的语言中的运算符++和运算符--将在 7.7.4 节中介绍。

APL 在语言中显得比较独特，因为它只有一个优先级，详见下一节。

优先级只给出了运算符运算顺序的一部分规则；而结合性规则也将对其有所影响。

7.2.1.2 结合性

考虑以下表达式：

```
a - b + c - d
```

如果加法与减法运算符有相同的优先级，就像编程语言中那样，那么优先级对该表达

1. 在早于 C99 版本的 C 中，运算符%在某些情形下是有执行依赖性的，因为除法也具有执行依赖性。

式的运算顺序起不了任何作用。

当表达式中两个优先级相同的运算符相邻[2]时，先运算哪个运算符的问题就由语言的结合性规则来决定。运算符可以有左结合性，也可以有右结合性，分别表示先计算左边的运算符还是先计算右边的运算符。

常见语言的结合性是从左到右的，但幂运算符(如果有的话)的结合性有时是从右向左。Java 表达式

```
a - b + c
```

先计算左边的运算符。

在 Fortran 和 Ruby 中，幂运算符是右结合的，所以表达式

```
A ** B ** C
```

先计算右边的运算符。

在 Ada 中，幂运算符是无结合性的，这意味着表达式

```
A ** B ** C
```

是非法的。这个表达式必须加上括号来说明期望的运算顺序，例如

```
(A ** B) ** C
```

或

```
A ** (B ** C)
```

在 Visual Basic 中，幂运算符^是左结合的。

下面是一些常见语言中的结合性规则：

语言	结合规则
Ruby	左结合：*、/、+、-
	右结合：**
基于 C 的语言	左结合：*、/、%、二元运算符+、二元运算符-
	右结合：++、--、一元运算符-、一元运算符+
Ada	左结合：除**之外的所有运算符
	无结合：**

如 7.2.1.1 节所述，在 APL 中，所有运算符都有相同的优先级。因此，在 APL 表达式中，运算符的运算顺序完全由结合性规则决定，即所有运算符都是从右到左计算的。例如，表达式

```
A × B + C
```

---

2. 若两个运算符仅由一个操作数隔开，则称这两个运算符相邻。

先运算加法运算符，然后计算乘法运算符(×是 APL 中的乘法运算符)。如果 A 是 3，B 是 4，C 是 5，则这个 APL 表达式的值为 27。

许多常用语言的编译器利用了一些算术运算符的数学结合性，即结合性规则对仅含有这些运算符的表达式之值没有影响。例如，加法具有数学结合性，故在数学中，表达式

```
A + B + C
```

其值与运算符的运算顺序无关。如果浮点运算对于具有数学结合性的运算符也具有结合性，编译器就可以利用这一点来进行一些简单的优化。具体而言，若允许编译器对运算符的运算重新排序，就可能产生一些能稍快地对表达式求值的代码。编译器通常就做此类优化。

遗憾的是，在计算机里，浮点的表示法和浮点的算术运算都只是其数学上的近似(由于空间限制)。数学运算符具有结合性，并不意味着对应的浮点运算也具有结合性。事实上，只有所有的操作数和中间结果都可以用浮点计数法准确地表示，这个过程才具有精确的结合性。例如，在某些情况下，计算机里的整数加法就不具有结合性。若程序要计算以下表达式

```
A + B + C + D
```

其中 A 和 C 是很大的正数，B 和 D 是绝对值很大的负数。在这种情况下，将 B 和 A 相加不会导致溢出，但将 C 和 A 相加就可能溢出。同样，将 C 和 B 相加不会导致溢出，但将 D 和 B 相加就可能溢出。由于计算机中的算术限制，加法在这种情况下体现出灾难性的非结合性。因此，如果编译器对这些加法运算符重新排序，就会影响表达式的值。当然，如果知道变量的估计值，程序员就可以避免这一问题。程序员可以将表达式定义成两部分(在两个赋值语句中)，以保证不会发生溢出。然而，这种对顺序的依赖性往往会以更加微妙的方式出现，程序员很难注意到。

7.2.1.3 括号

程序员可以在表达式中加入括号，来调整优先级和结合性规则。在表达式中，括号中的部分的优先级较其相邻的非括号中的部分要高。例如，尽管乘法的优先级比加法高，但是在以下表达式中

```
(A + B) * C
```

先运算加法，在数学上这很自然。在这个表达式中，只有计算了括号内子表达式的加法，才能得到乘法运算符的第一个操作数。同理，7.2.1.2 节中的表达式可以写成

```
(A + B) + (C + D)
```

以避免溢出。

若语言允许在算术表达式中使用括号，就可以抛弃所有的优先级规则，简单地将所有运算符按从左到右或从右到左的顺序结合。程序员可以利用括号来指定期望的运算顺序。

这种方法十分简单，因为程序的开发者和阅读者都不必记忆优先级或者结合性规则。这种方案的缺点在于，表达式的书写比较乏味，严重降低了代码的可读性。不过 APL 的设计者 Ken Iverson 就选择了这种方案。

7.2.1.4　Ruby 表达式

Ruby 是一种纯粹的面向对象语言，也就是说，每个数据值，包括字面量，都是对象。Ruby 支持基于 C 的语言中所有的算术运算和逻辑运算。在表达式领域，Ruby 与基于 C 的语言的不同之处在于，所有的算术、关系和赋值运算，以及数组索引、移位和逐位逻辑运算，都实现为方法。例如，表达式 a+b 表示调用 a 引用的对象之方法+，传递的参数是 b 所引用的对象。

把运算符实现为方法的一个有趣结果是，应用程序可以重写它们。因此，这些运算符可以重定义。虽然重写一些预定义类型的运算符常常没有什么用处，但如 7.3 节所述，为用户定义的类型重写一些预定义的运算符是非常有用的，这在一些语言中可以通过运算符重载来实现。

7.2.1.5　LISP 中的表达式

与 Ruby 相同，LISP 中的所有算术运算和逻辑运算都由子程序完成。但在 LISP 中，子程序必须显式调用。例如，要在 LISP 中指定 C 表达式 a+b*c，就必须编写如下表达式[3]：

```
(+ a (* b c))
```

在这个表达式中，+和*都是函数名。

7.2.1.6　条件表达式

if-then-else 语句可以用来执行条件表达式。例如：

```
if (count == 0)
  average = 0;
else
  average = sum / count;
```

在基于 C 的语言中，这段代码可以用包含条件表达式的赋值语句更方便地表示，形式如下：

```
表达式 1 ？ 表达式 2 ： 表达式 3
```

其中，表达式 1 解释为布尔表达式。如果表达式 1 的值为真，整个表达式的值就是表达式 2 的值；否则，整个表达式的值就是表达式 3 的值。例如，上述的 if-then-else 程序段可以用如下包含条件表达式的赋值语句来代替：

---

3. 在 LISP 中，把列表解释为代码时，第一个元素是函数名，其他元素是函数的参数。

```
average = (count == 0) ? 0 : sum / count;
```

实际上，问号表示 then 子句的开始，冒号表示 else 子句的开始。这两个子句都是必须有的。注意，条件表达式中的问号用作三元运算符。

条件表达式可以用在程序(在基于 C 的语言中)中任何可以使用其他表达式的地方。除了基于 C 的语言之外，条件表达式还可以用在 Perl、JavaScript 和 Ruby 中。

7.2.2 操作数的运算顺序

表达式有一个不大讨论的设计特性，即操作数的运算顺序。在计算表达式中的变量时，需要读取它们在内存中的值。有时，常量也是这样计算的。但是在其他情况下，常量可能是机器语言指令的一部分，不需要从内存中读取。如果操作数是带有括号的表达式，就必须先计算它包含的所有运算符，之后才能将它的值用作操作数。

如果运算符的两个操作数都没有副作用，操作数的运算顺序就无关紧要。因此，只有操作数的计算有副作用时，才会发生有趣的事。

7.2.2.1 副作用

当函数改变其参数或者全局变量时，我们就说函数是有副作用的(全局变量在函数外声明，但是可以由函数访问)。

考虑如下的表达式：

```
a + fun(a)
```

如果函数 fun 没有副作用(即不改变 a 的值)，那么 a 和 fun(a)这两个操作数的运算顺序就不影响表达式的值。但是，如果函数 fun 会改变 a，操作数的运算顺序就影响表达式的值。考虑这种情况：函数 fun 返回 10，并将其参数的值改成 20。假设有如下代码：

```
a = 10;
b = a + fun(a);
```

那么，如果先读取 a 的值 (在表达式的计算过程中)，它的值就是 10，整个表达式的值是 20。但是如果先计算第二个操作数，则第一个操作数会变为 20，整个表达式的值是 30。

如下的 C 程序也遇到了同样的问题，即函数改变了表达式中的某个全局变量：

```
int a = 5;
int fun1() {
  a = 17;
  return 3;
} /* end of fun1 */
void main() {
  a = a + fun1();
} /* end of main */
```

函数 main 算出的 a 值与表达式 a+fun1()的运算顺序有关。a 的值要么是 8(如果先计算

a)，要么是 20(如果先调用函数)。

注意，数学里的函数没有副作用，因为数学中没有变量的概念。函数式编程语言也没有这个问题。在数学和函数式编程语言中，函数要比命令式语言中更容易理解，因为其上下文和它们没有关系。

对于操作数的运算顺序以及副作用问题，有两种解决办法。其一是语言的设计者可以禁止函数带有副作用，来防止函数的计算影响到表达式的值；其二是在语言的定义中规定表达式中操作数的运算顺序，并要求实现者按该顺序实现。

在命令式语言中，禁止函数带有副作用是比较困难的，这会降低程序员编程的灵活性。考虑 C 和 C++语言，这两种语言只有函数，这意味着所有的子程序都只返回一个值。而为了去除双向参数的副作用，同时仍让子程序返回多个值，就需要将这些值放在一个 struct 里，然后返回这个 struct。在函数中访问全局变量也是应该禁止的。但是，当对效率的要求很高时，使用全局变量来避免传递参数是提高执行速度的一种重要方法。例如，在编译器里，对符号表等数据进行全局访问是很常见的。

严格限制运算顺序的问题是，编译器使用的一些代码优化技巧会改变操作数的运算顺序。在调用函数时，规定运算顺序将禁止这些优化方法的使用。因此，在实际的语言设计中，还没有完美的解决办法。

Java 语言的定义规定，操作数是从左向右计算的，从而消除了本节讨论的问题。

7.2.2.2　引用透明和副作用

引用透明的概念与函数的副作用有关并受其影响。在程序中，如果任意两个值相同的表达式能够在该程序的任何地方互换，并且不影响程序的运行，这个程序就是引用透明的。引用透明的函数值完全依赖于它的参数[4]。下面的例子表明了引用透明和函数副作用的联系：

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

如果函数 fun 没有副作用，result1 和 result2 就是相等的，因为赋予它们的表达式是等价的。但是，假设函数 fun 有副作用，比如它给 b 或 c 加上 1，那么 result1 将不等于 result2。所以，该副作用去除了这个程序的引用透明性。

引用透明的程序有几个优点。其中最重要的一点是，这种程序的语义比非引用透明的程序更容易理解。从易于理解的角度讲，引用透明使程序中的函数等价于一个数学函数。

用纯函数式语言编写的程序没有变量，所以是引用透明的。在纯函数式语言中，函数存储在局部变量里，所以没有状态。如果某个函数用到了函数外的值，这个值就必须是常量，因为在这种语言中没有变量。因此，函数的值依赖于它参数的值。

引用透明将在第 15 章中进一步讨论。

---

4. 而且，函数的值还不能依赖其参数的计算顺序。

7.3 运算符重载

算术运算符往往有多种用途。例如，+通常用来表示整数加法和浮点数加法。在有些语言中，例如 Java 中，也用它来拼接字符串。运算符的多种用法叫做运算符重载，通常情况下，只要不降低可读性和可靠性，运算符重载是可以接受的。

运算符重载可能带来危险，以 C++中的&符号为例：作为一个二元运算符，&用来表示两个二进制数的按位逻辑 AND 运算。然而，作为一个一元运算符，它的意义完全不同。此时，它的操作数是一个变量，表达式的值是该变量的地址。这种情况下，&称作取址符。例如，执行如下语句：

```
x == &y;
```

结果是 y 的地址放进了 x 里。&的多种用途存在两个问题。其一，两个完全没关系的运算却使用相同的符号，这降低了程序的可读性。其二，如果输入代码时，漏掉了按位 AND 运算的第一个操作数，编译器不会发现这个错误，因为它会解释为一个取址符。这种错误很难发现。

事实上，减号运算符的重载使几乎所有的编程语言都有一个相似却并不严重的问题：就是编译器不能判断这个运算符是一元的还是二元的[5]。所以如果该运算符用于二元运算时，漏掉了第一个操作数，同样不会被编译器认定为错误。然而，两种运算的意义，不论是二元的还是一元的，都是很相近的，所以可读性并未受什么影响。

有些支持抽象数据类型的语言(参见第 11 章)，例如 C++、C#和 F#，允许程序员进一步重载运算符。比如，设想用户要在一个整型标量和一个整型数组之间定义一个*运算，表示数组中的每个元素都乘以该标量。要定义这个运算符，可以编写一个函数子程序*，来执行这个新操作。调用重载的运算符时，编译器会根据操作数的类型，在该语言定义的重载运算符中选择正确的含义。例如，如果 C#程序定义了一个这样的*运算符，那么只要*的左操作数是一个整数，右操作数是一个整型数组时，C#的编译器就会使用这个新的定义。

只要使用合理，用户定义的运算符可以提高程序的可读性。例如，如果为矩阵抽象数据类型重载了+和*运算符，A、B、C、D 是这种数据类型的变量，那么

```
A * B + C * D
```

可以代替

```
MatrixAdd(MatrixMult(A, B), MatrixMult(C, D))
```

另一方面，用户重载运算符也能降低可读性。一方面，无法禁止用户定义一个+来表示乘法。而且，在程序中看到*运算符时，读者需要确定其操作数的类型和这个运算符的定

5. ML 为一元和二元减号运算符使用了不同的符号，减缓了这个问题。~表示一元减号运算符，-表示二元减号运算符。

义，才能确定它的意思。这些定义的部分或全部可能在其他文件里。

另一个要考虑的是，根据其他小组创建的模块来构建软件系统的过程。如果其他小组以不同的方式重载了相同的运算符，显然必须去除这些差异，才能把系统整合在一起。

C++有几个运算符不能重载，类或结构成员运算符(.)和域解析运算符(::)就是其中的两个。有趣的是，运算符重载是几个没有复制到 Java 中的 C++特征之一，但是它出现在 C#中。

用户定义的运算符重载的实现将在第 9 章讨论。

7.4　类型转换

类型转换分为收缩转换和扩展转换。收缩转换将一个类型的值转换为另一个类型，新的类型甚至不能存下原类型数据的几乎所有值。例如，在 Java 中将 double 类型的值转换为 float 类型，就是一个收缩转换，因为 double 的取值范围要比 float 大得多。而扩展转换将一个类型的值转换为另一个类型，新的类型至少能存下原类型的几乎所有值。例如，在 Java 中将一个 int 类型的值转换为 float 类型，就是一个扩展转换。扩展转换几乎总是安全的，这就是说转换后的值的量级是不变的。收缩转换并不总是安全的，有时转换后的值的量级会在该过程中改变。例如，在 Java 程序中将浮点值 1.3E25 转换为整数，结果将与原来的值相去甚远。

虽然扩展转换通常是安全的，但是它可能降低精度。在很多语言的实现中，虽然整数到浮点数是一个扩展转换，但是精度可能会降低。例如，很多情况下，整数是用 32 个 bit 来存储的，其精度至少是 9 位数字。浮点数同样用 32 个 bit 来存，但其精度只有 7 位数字(因为要有存储指数的空间)。因此，整数到浮点数的扩展转换会损失 2 位精度。

当然，非基本类型的强制转换更加复杂。第 5 章讨论了数组和记录类型赋值兼容的复杂性。方法的何种参数类型和返回值类型允许它重载超类中的方法同样是个问题——究竟是仅当类型相同时才能重载，还是其他情况也可以重载。这个问题，以及基于子类型的子类的概念将在第 12 章中讨论。

类型转换既可以是显式的，也可以是隐式的。下面分别讨论这两种类型转换。

7.4.1　表达式中的强制类型转换

算术表达式的一个设计决策是，运算符可否有不同类型的操作数。可以有不同类型的操作数的表达式称为混合模式表达式，支持这种表达式的语言必须定义用于操作数隐式转换类型的协议，因为计算机中没有使用不同类型操作数的二元运算符。第 5 章提到，强制转换定义为一种由编译器执行的隐式转换。程序员显式要求的类型转换称为显式转换，而非强制转换。

虽然一些运算符可能重载，但是我们假定在计算机系统中，无论是在硬件还是在某个

软件层面，语言都为每种操作数类型和运算符定义了一种操作[6]。在使用静态类型绑定的语言中，编译器会根据操作数的类型为重载的运算符选择正确的操作。当运算符的两个操作数有不同的类型，且该语言支持这种设计时，编译器必须选择一个操作数，进行强制类型转换，并且提供强制转换的代码。在接下来的讨论中，将介绍几种常用语言中强制转换的设计方案。

语言设计者在算术表达式的强制转换问题上有分歧。反对大范围强制转换的人们担心这会带来可靠性的问题，因为这样会减少类型检查的好处。而支持大范围强制转换的人认为，这种限制会降低灵活性。问题是究竟是由程序员还是编译器来检测这类错误。

举一个简单的例子，考虑如下的 Java 代码：

```
int a;
float b, c, d;
...
d = b * a;
```

假设乘法运算符的第二个操作数应该是 c，由于键入错误它变成了 a。因为混合模式的表达式在 Java 语言中是合法的，所以编译器不会把它当作一个错误，它只会插入代码，把 int 操作数 a 的值强制转换为 float 类型。但是如果混合模式的表达式在 Java 中是非法的，编译器就会把这个键入错误检测为类型错误。

由于支持混合模式的表达式会减少这种错误检测，Ada 语言几乎不允许表达式中有不同类型的操作数。它不允许在一个表达式中同时出现整型和浮点型的操作数，但有一个例外：幂运算符**可以使用整型或浮点型的数作为第一个操作数，但第二个操作数只能是整数。Ada 允许其他一些不同类型操作数的混用，通常和子界类型有关。如果上述 Java 的代码用 Ada 语言改写，就是：

```
A : Integer;
B, C, D : Float;
...
C := B * A;
```

Ada 的编译器会指出这个表达式错了，因为 Float 和 Integer 的操作数不能在*运算符中混用。

ML 和 F#不强制转换表达式中操作数的类型。类型转换必须是显式的。这样其表达式就获得了与 Ada 相同级别的可靠性。

在其他的大多数常用语言中，算术表达式没有这种混合类型的限制。

在基于 C 的语言中，有一些比 int 类型还要小的整数类型。在 Java 中，它们是 byte 和 short。所有这些类型的操作数在使用任何运算符时，几乎都会强制转换为 int 类型。所以尽管数据可以存储在这些类型的变量中，但必须先转换为一个更大的类型，之后才能进行处理。比如如下的 Java 代码：

---

6. 这个假定不是对每种语言都成立。本节后面将给出一个例子。

```
byte a, b, c;
...
a = b + c;
```

b 和 c 的值都强制转换为 int，并执行整型加法。然后把和转换为 byte，并存储在 a 中。除非有大量的数据需要存储，否则由于现代计算机的存储量巨大，没有必要使用 byte 和 short。

7.4.2　显式类型转换

绝大多数语言都提供一些功能来执行显式类型转换，包括扩展转换和收缩转换。在一些情况下，显式的收缩转换使对象原有的值产生巨大的变化时，程序就会发布警告信息。

历史注释

强制转换过多会带来危险，并付出代价。下面是一个非常极端的例子：PL/I 为了实现表达式的灵活性，允许字符串变量与表达式中的整数相结合。在运行时，扫描字符串可以得到一个数值。如果该值含有一个小数点，就假定它为浮点型，并将另一个操作数强制转换为浮点型，结果也是浮点数。这种强制转换策略的代价很高，因为必须在运行时进行类型检查和转换。同时也不可能检测程序员在表达式中犯的错误，因为二元运算符可以将两个任意类型的操作数相结合。

在基于 C 的语言中，显式类型转换叫做强制类型转换。要指定强制类型转换，应将转换后的类型放在要转换的表达式前的括号中，如下：

```
(int)angle
```

在这些转换中，使用圆括号把类型括起来的一个原因是，这些语言的始祖(C 语言)存在一些双单词的类型名，比如 long int。

在 ML 和 F#中，强制类型转换的语法和调用函数一样。比如在 F#中有：

```
float(sum)
```

7.4.3　表达式中的错误

在表达式的求值过程中，可能发生很多错误。如果语言要求类型检测(无论是静态的还是动态的)，操作数的类型错误就不会发生。这种错误之所以会发生，是因为表达式中操作数的强制转换，这在前文讨论过。出现其他种类的错误，是因为计算机算术的限制和算术自身固有的限制。最常见的错误是由操作的结果无法在内存单元中存储而引起的。根据结果过大或过小，这种错误叫做上溢或下溢。算术的一个限制是不允许除以 0。当然，数学上非法的事并不能阻止程序尝试这样做。

浮点数上溢、下溢和除以 0 是运行时错误的例子。这类错误有时叫浮点溢出。第 14 章将讨论程序检测和处理浮点溢出的语言特性。

7.5 关系表达式和布尔表达式

除了算术表达式之外，编程语言还支持关系表达式和布尔表达式。

7.5.1 关系表达式

关系运算符用于比较两个操作数的值。关系表达式包含两个操作数和一个关系运算符。关系表达式的值是布尔型的(当语言包含布尔类型时)。关系运算符经常要为各种各样的类型重载。操作数的类型决定这个操作的返回值是真还是假。关系表达式的操作数为整型时比较简单，操作数为字符串时比较复杂。通常，可以进行关系运算的操作数是数值型、字符串或序数。

用于相等和不等的关系运算符的语法在不同的编程语言中有不同的形式。比如，在基于 C 的语言中用!=表示不等于，Ada 用/=表示，Lua 用~=表示，Fortran 95+用.NE.或<>表示，ML 和 F#用<>表示。

历史注释

Fortran I 的设计者使用英文缩写来表示关系运算符，因为在设计 Fortran I 的时代(20 世纪 50 年代中期)，穿孔卡还没有包含符号>和<。

JavaScript 和 PHP 有两个额外的关系运算符===和!==，它们类似于==和!=，但是禁止强制转换操作数的类型。比如，表达式

```
"7"==7
```

在 JavaScript 中返回真，因为比较字符串与数字时，会把字符串转换为数值。然而，表达式

```
"7"===7
```

返回假，因为此运算符的操作数不会强制转换类型。

Ruby 用==表示要强制转换类型的相等比较运算符，用 eql? 表示不强制转换类型的相等比较运算符。如第 8 章所述，Ruby 只在 case 语句的 when 子句中使用===。

关系运算符的优先级总是比算术运算符低，因此表达式

```
a + 1 > 2 * b
```

先计算算术表达式。

7.5.2 布尔表达式

布尔表达式包含布尔变量、布尔常量、关系表达式和布尔运算符。布尔运算符一般包含 AND(与)、OR(或)和 NOT(非)三种，有时候还有异或和等价。布尔运算符经常只有布尔型操作数(布尔变量、布尔字面量和关系表达式)，产生布尔型的值。

在布尔代数中，OR 和 AND 运算符必须有相同的优先级。因此，Ada 的 AND 和 OR

运算符有相等的优先级。然而，在基于 C 的语言中，AND 的优先级比 OR 高。也许这是因为 AND 代表乘法，OR 代表加法，乘法的优先级高于加法——但这种联系是无根据的。

因为算术表达式可以是关系表达式的操作数，关系表达式可以是布尔表达式的操作数，所以这三大类运算符必须有不同的优先级。

在基于 C 的语言中，算术、关系、布尔运算符的优先级如下：

最高级	后缀++、--
	一元+、-、前缀++、--、!
	*、/、%
	二元+、-
	<、>、<=、>=
	=、!=
	&&
最低级	||

在流行的命令式语言中，C99 之前的 C 版本有些特别，因为它们没有布尔类型，所以没有布尔值，而是用数值来表示布尔值。即用标量变量(数值或字符)和常量来表示布尔操作数，0 表示假，非 0 表示真。布尔运算的结果是整数，用 0 表示假，1 表示真。在 C99 和 C++中，算术表达式也能用做布尔表达式。

C 设计关系表达式的一个奇怪结果是，下面的表达式是合法的：

```
a > b > c
```

因为 C 语言的关系运算符是左结合的，所以先计算最左边的关系运算符，得到 0 或 1，然后这个值和变量 c 进行比较。在这个表达式中，绝对不会比较 b 和 c。

在包括 Perl 和 Ruby 在内的一些语言中，提供了两套二元逻辑运算符：用&&和 and 表示 AND，用||和 or 表示 OR。&&和 and(||和 or)之间的区别是拼写版本的优先级更低。另外，and 和 or 的优先级相同，但是&&的优先级比||高。

把基于 C 的语言中的非算术运算符包括进来，就会得到 40 多个运算符和至少 14 种不同的优先级，这清晰地反映了在这些语言中运算符的丰富性和表达式的复杂度。

如第 6 章所述，可读性要求一种语言应该包括布尔类型，而不是在布尔表达式中使用简单的数字类型。在使用数字类型作为布尔操作数时，一些错误检查不出来，因为任何数字表达式，无论是有意还是无意，都是布尔运算符的合法操作数。在其他命令式语言中，将非布尔表达式用作布尔运算符的一个操作数，会检测为一个错误。

7.6　短路求值

如果表达式的结果不是在计算了所有的操作数和运算符之后得出的，就叫做表达式的

短路求值。例如算术表达式

```
(13 * a) * (b / 13 - 1)
```

如果 a=0，那么该表达式的值与(b/13-1)的值是无关的，因为对于任意的 x，0*x 的值恒为 0，所以当 a=0 时，就没有必要计算表达式(b/13－1)或者是执行第二个乘法。然而，在程序的执行过程中，算术表达式中的短路求值是不容易检测到的，因此这样的短路求值从来不会发生。

如果 a<0，则布尔表达式

```
(a >= 0) && (b < 10)
```

的值与第二个关系表达式无关，因为对于 b 的所有取值，表达式(False && (b<10))的值永远为 False。因此，当 a<0 时，就没有必要计算 b、常量 10、第二个关系表达式和&&操作。和算术表达式的情况不一样，这种短路求值在执行过程中很容易检测到。

为了阐明布尔表达式的非短路求值可能存在的问题，假设在 Java 中不使用短路求值。使用 while 语句可以编写查表循环。下面是用于这种查询的一段简单的 Java 代码，假定 list 有 listlen 个元素，它是要查询的数组，而 key 是所搜索的值。

```
index = 0;
while ((index < listlen) && (list[index] != key))
  index = index + 1;
```

如果这种求值不是短路的，则无论第一个表达式的值是什么，都要计算 while 语句中布尔表达式中的两个关系表达式。因此如果 key 不在 list 中，程序将会因下标出界错误而终止。index==listlen 的循环会引用 list[listlen]，因为 list 的上界为 listlen-1，所以会导致下标错误。

如果语言支持并使用布尔表达式的短路求值，这就不是一个问题。在前面的例子中，短路求值方法会计算 AND 运算符的第一个操作数，如果第一个操作数为假，则会跳过第二个操作数。

若语言支持布尔表达式的短路求值，但对表达式有副作用，就会发生难以捉摸的错误。假设对表达式进行短路求值，但对该表达式的某部分有副作用，就计算不出该表达式的结果。只有完整地计算整个表达式，才会出现副作用。如果程序的正确性取决于这个副作用，那么短路求值会导致严重的错误。例如下面的 Java 表达式

```
(a > b) || ((b++) / 3)
```

在这个表达式中，只有 a<=b 时才会改变 b 值(在第二个算术表达式中)。如果程序员假设执行期间每一次计算表达式时，都会改变 b(程序的正确性取决于这种改变)，这个程序就会失败。

Ada 使用双字运算符 and then 和 or else 来指定布尔运算符 AND 和 OR 的短路求值。Ada 也有非短路求值的运算符 and 和 or。

在基于 C 的语言中，通常的 AND 和 OR 运算符，分别标记为&&和||，这两个符号都进行短路求值。然而，这些语言也具有按位运算符 AND 和 OR，分别标记为&和|，这两个符号可用于结果为布尔值的操作数，是不进行短路求值的。当然，只有所有操作数都严格限定为 0(假)或 1(真)，位运算符才和通常的布尔运算符等价。

Ruby、Perl、ML、F#和 Python 中的所有逻辑运算符都进行短路求值。

Ada 将短路求值和普通操作同时包含进去的设计无疑是最好的，因为程序员可以对任何适合的布尔表达式使用短路求值。

7.7　赋值语句

如前所述，赋值语句是命令式语言的核心概念之一。它提供了用户动态改变变量与其值的绑定方法。下一节讨论最简单的赋值操作。之后的几节将描述多种可供选择的功能。

7.7.1　简单赋值

目前流行的几乎所有编程语言都使用等号作为赋值运算符。所有这些语言都必须使用与等号不同的符号来表示相等的关系运算符，以避免与赋值运算符相混淆。

ALGOL 60 语言使用符号:=作为赋值运算符，来避免赋值运算符和相等关系运算符混淆。Ada 也使用了这个赋值运算符。

各种语言中赋值操作的用法设计有很大的不同。在某些语言里，比如 Fortran 和 Ada，赋值操作只能表示为独立的语句，赋值操作的结果只能是单个的变量。然而，还存在很多不同的做法。

7.7.2　条件赋值

Perl 语言允许条件赋值。比如

```
($flag ? $count1 : $count2) = 0;
```

这等价于

```
if ($flag) {
  $count1 = 0;
} else {
  $count2 = 0;
}
```

7.7.3　混合赋值运算符

混合赋值运算符是一种快捷方法，可以指定常常需要的赋值形式。通过这个技术可以缩简赋值形式，让目标变量也作为出现在表达式右侧的第一个操作数中，如：

```
a = a + b
```

混合赋值运算符由 ALGOL 68 提出，后面被 C 采用，但形式略有区别，成为其他基于 C 的语言、Perl、JavaScript、Python 和 Ruby 的一部分。这些赋值运算符的语法是将所期望的二元运算符和=运算符连接起来。比如

```
sum += value;
```

等价于

```
sum = sum + value;
```

支持混合赋值运算符的语言对大多数二元运算符都有其特定的形式。

7.7.4 一元赋值运算符

基于 C 的语言、Perl 和 JavaScript 包含两种独特的、非常精简的一元算术运算符。它们将递增、递减操作和赋值操作联系起来。运算符++用于递增，--用于递减，它们可以用于表达式，也可以是独立的单运算符赋值语句。它们既能作为前缀运算符(放在操作数前)，也能作为后缀运算符(放在操作数后)。在赋值语句

```
sum = ++ count;
```

中，count 的值先递增 1，再赋给 sum。这个操作也可以写成

```
count = count + 1;
sum = count;
```

如果将这个运算符用作后缀运算符，如下：

```
sum = count ++;
```

则先把 count 的值赋给 sum，再递增 count。其效果等同于下面的语句：

```
sum = count;
count = count + 1;
```

下面的例子使用一元递增运算符构成一个完整的赋值语句：

```
count ++;
```

这条语句会让 count 增加 1。它看起来不像是赋值，但它确实是一个赋值语句。它与下面的语句等价：

```
count = count + 1;
```

把两个一元运算符作用于同一个操作数时，计算顺序是从右向左。比如：

```
- count ++;
```

首先递增 count，然后取负值。因此它等价于

```
-(count  ++)
```

而不是

```
(-count)  ++
```

7.7.5　赋值表达式

在基于 C 的语言、Perl 和 JavaScript 中，赋值语句会产生一个返回值，这个值与赋给目标变量的值相同。因此赋值语句可以用作表达式或其他表达式中的操作数。这个设计使赋值运算符与其他二元运算符的处理非常相似，但它会改变运算符的左操作数。比如，在 C 中，常常有这样的语句：

```
while ((ch = getchar()) != EOF) { ... }
```

在这个语句中，getchar 获得从标准输入文件(通常是键盘)中输入的下一个字符，并将它赋给变量 ch。接下来这个结果，也就是赋给 ch 的值，会与常量 EOF 进行比较。如果 ch 不等于 EOF，则执行复合语句{…}。注意，赋值语句必须放在括号中。因为在支持将赋值看作表达式的语言中，赋值运算符的优先级低于比较运算符。如果不加上括号，就先比较新读入的字符与 EOF。于是，赋给 ch 的将是比较的结果，0 或者 1。

让赋值语句看作表达式中一个操作数的缺点是，它使语言又多了一种表达式副作用。这类副作用可能会导致表达式很难阅读和理解。有任何一种副作用的表达式都有这个缺点。这样的表达式不能看作表达式(在数学中是指数值的表示)，只能看作一系列有着奇怪的执行顺序的指令。例如，表达式

```
a = b + (c = d / b) - 1
```

表示如下指令：

```
将 d/b 赋给 c
将 b+c 赋给 temp
将 temp-1 赋给 a
```

注意，将赋值运算符当作一般的二元运算符，就可以对多个目标变量赋值。例如：

```
sum = count = 0;
```

其中，count 首先赋为 0，然后把 count 的值赋给 sum。这样的多目标赋值在 Python 中也是允许的。

在 C 的赋值表达式设计中，牺牲了一部分常常导致程序错误的错误检查。例如，如果输入

```
if (x = y) ...
```

而不是

```
if (x == y) ...
```

这是一个常见的错误，但编译器检查不出来。编译器原本应当测试一个关系表达式，

却测试了赋予 x 的值(在这个例子中，真正测试的是这个语句赋给 y 的值)。这其实是两个设计决策的结果：允许赋值运算符像其他二元运算符一样处理，使用两个非常相似的运算符=和==，表达的是完全不同的意思。这是 C 和 C++程序缺乏安全性的又一个例子。注意，Java 和 C#只允许 boolean 表达式出现在 if 语句中，解决了这个问题。

7.7.6 多重赋值

几种新近的编程语言，包括 Perl、Ruby 和 Lua，提供了一种多目标、多来源的赋值语句。比如说，在 Perl 中，可以有如下语句：

```
($first, $second, $third) = (20, 40, 60);
```

它的语义是：把 20 赋给$first，40 赋给$second，60 赋给$third。如果需要交换两个变量的值，仅用如下一条语句就可以实现：

```
($first, $second) = ($second, $first);
```

这个语句正确地交换了$first 和$second 的值，没有使用临时变量(至少不需要程序员来创建和管理临时变量)。

Ruby 中多重赋值的最简单形式与 Perl 类似，区别在于，Perl 中等号两边的元素不需要加上括号。Ruby 中还有其他一些更灵活的多重赋值方式，这里不讨论。

历史注释

第一个实现 C 的 PDP-11 计算机有自动递增和自动递减寻址模式，它们是 C 的递增和递减运算符在用作数组下标时的硬件版本。由此而来的一个可能的猜测是，这些 C 运算符的设计基于 PDP-11 体系结构的设计。但这个猜测是错误的，因为 C 运算符继承自 B 语言，而 B 语言在第一个 PDP-11 计算机推出之前就设计出来了。

7.7.7 函数式编程语言中的赋值

在纯函数式语言中使用的所有标识符和其他函数式语言中使用的部分标识符只是数值的名字。另外，它们的值从不改变，例如，在 ML 中，名字用 val 声明绑定到数值上，其形式如下所示：

```
val cost = quantity * price;
```

如果 cost 在后续 val 声明的左边，该声明就创建了 cost 的一个新版本，它与原来的 cost 没有任何关系，而原来的 cost 隐藏起来了。

F#有一个类似的声明，它使用 let 保留字。F#的 let 和 ML 的 val 之间的区别是，let 会创建新的作用域，而 val 不会。实际上，在 ML 中，val 声明常常嵌套在 let 结构中。let 和 val 详见第 15 章。

7.8　混合模式赋值

混合模式表达式在 7.4.1 节中讨论过。赋值语句也常常是混合类型的。但有一个设计问题：表达式的类型必须与被赋值的变量的类型相同吗？换言之，隐式类型转换是否可以用于一部分类型不匹配的情况？

Fortran、C、C++和 Perl 用于混合模式赋值的隐式类型转换规则，与混合类型表达式的隐式类型转换规则相似。也就是说，很多情况下，混合类型赋值是允许的，会自由引用隐式类型转换[7]。而 Ada 不允许混合类型转换。

和 C++明显不同，Java 和 C#只有在需要使用的隐式类型转换是扩展转换时，才允许使用混合类型赋值[8]。因此，int 值可以赋给 float 变量，反之则不行。与 C、C++相比，Java 和 C#中禁止大约一半的混合类型赋值，是提高可靠性的一种简单、但非常有效的方式。

当然，在函数式语言中，赋值语句仅用于名字值，没有混合类型赋值。

小　结

表达式包括常量、变量、括号、函数调用和运算符。赋值语句包括目标变量、赋值运算符和表达式。

表达式的语义在很大程度上取决于各运算符的运算顺序。语言中用于表达式中运算符的结合性和优先级规则，决定了表达式中各运算符的运算顺序。当函数存在副作用时，操作数的计算顺序也非常重要。类型转换有扩展和收缩两种。有些收缩类型转换会产生不正确的值。在表达式中经常会出现隐式类型转换，但这有碍于类型检查中的错误检测，因而降低语言的可靠性。

赋值语句有各种各样的形式，包括条件目标、赋值运算符和列表赋值。

复 习 题

1. 定义运算符的优先级和运算符的结合性。
2. 什么是三元运算符？
3. 什么是前缀运算符？
4. 什么运算符通常有右结合性？
5. 什么是不具有结合性的运算符？

---

7. 注意，在 Python 和 Ruby 中，类型是关联在对象上的，而不是变量上。因此，在这些语言中，并没有混合类型赋值。

8. 也并不完全是这样。如果把一个整型字面值(编译器默认指定其类型为 int)赋给 char、byte 或者 short 变量，并且这个字面值在相应类型的范围内，那么 int 值在收缩类型转换中转换为对应变量的类型，这个收缩转换不会出错。

6. APL 对结合性是怎样规定的？
7. 在 C++和 Ruby 中，运算符的实现方式有什么区别？
8. 什么是函数的副作用？
9. 什么是隐式类型转换？
10. 什么是条件表达式？
11. 什么是重载的运算符？
12. 什么是收缩类型转换和扩展类型转换？
13. 在 JavaScript 中，==和===有什么区别？
14. 什么是混合模式表达式？
15. 什么是引用透明？
16. 引用透明有什么好处？
17. 操作数的运算顺序与函数副作用是怎么互相影响的？
18. 什么是短路求值？
19. 说出一种对布尔表达式使用短路求值的语言。说出一种从不进行短路求值的语言。再说出一种程序员可以决定是否进行短路求值的语言。
20. C 是怎么支持关系表达式和布尔表达式的？
21. 复合赋值运算符的作用是什么？
22. C 中一元运算符的结合性是怎么规定的？
23. 把赋值运算符看作算术运算符的一个可能的缺点是什么？
24. 哪两种语言允许多重赋值？
25. Ada 允许怎样的混合赋值？
26. Java 允许怎样的混合赋值？
27. ML 允许怎样的混合赋值？
28. 什么是强制类型转换？

习　题

1. 你何时希望编译器忽略表达式中类型的不同？
2. 请阐述你赞成和反对混合模式表达式的论点。
3. 你认为在你最喜欢的语言中取消运算符重载有好处吗？为什么？
4. 你认为取消所有的运算符优先级规定，而要求用括号来指明优先级，好不好？为什么？
5. C 的赋值运算符(例如，+=)应包括到其他(没有包括它们的)语言中吗？为什么？
6. C 的一元赋值运算符(例如，++count)应包括到其他(没有包括它们的)语言中吗？为什么？
7. 指出编程语言中相加运算符不具有交换率的情况。
8. 指出编程语言中相加运算符不具有结合性的情况。
9. 假设表达式遵循如下的结合性和优先级规则：

优先级	最高	*, /, **not**
		+, -, &, **mod**
		-(一元)
		=, /=, <, <=, >=, >
		and
	最低	**or**, **xor**
结合性	从左到右	

在每个子表达式两侧加上括号，为每个右括号加上上标，表明表达式的运算顺序。比如，对于表达式

```
a + b * c + d
```

运算顺序可以表示为

```
((a + (b * c)1)2 + d)3
```

a. `a * b - 1 + c`
b. `a * (b - 1) / c mod d`
c. `(a - b) / c & (d * e / a - 3)`
d. `-a or c = d and e`
e. `a > b xor c or d <= 17`
f. `-a + b`

10. 假设对于所有运算符，没有优先级，结合性为从右到左，请指出第 9 题中各表达式的运算顺序。
11. 对于第 9 题定义的运算符优先级和相关性规定，写一个 BNF 描述。假设仅有的操作数名称是 a、b、c、d 和 e。
12. 利用第 11 题的语法，画出第 9 题中表达式的语法解析树。
13. 函数 fun 的定义如下：

```
int fun(int *k) {
 *k += 4;
return 3 * (*k) - 1;
}
```

假设 fun 用在下面的程序中：

```
void main() {
  int i = 10, j = 10, sum1, sum2;
  sum1 = (i / 2) + fun(&i);
  sum2 = fun(&j) + (j / 2);
 }
```

在如下条件下，sum1 和 sum2 的值是多少？

a. 假如各操作数的运算顺序是从左到右
b. 假如各操作数的运算顺序是从右到左

14. 你反对(或赞成)APL 的运算符优先级规则的主要论点是什么？
15. 为何难以消除 C 语言中的函数副作用？
16. 对于你选择的某种语言，列出所有的运算符，它能消除一切运算符重载。
17. 对于你知道的两种语言，指出当收缩类型转换使被转换的值失去意义时，它们是否会给出错误信息。
18. 用于 C 或 C++的优化编译器是否应当允许改变布尔表达式中各子表达式的顺序？为什么？
19. 对于 Ada 语言，请再次回答问题 17。
20. 观察下面的 C 程序：

```
int fun(int *i) {
  *i += 5;
  return 4;
 }
void main() {
  int x = 3;
  x = x + fun(&x);
 }
```

在执行了 main 函数的赋值语句之后，在如下条件下，x 的值是什么？

a. 操作数按从左到右的顺序计算

b. 操作数按从右到左的顺序计算

21. 为什么 Java 指定表达式中的操作数都按从左到右的顺序计算？
22. 解释一种语言的隐式类型转换规则怎样影响其错误检测。

编程练习

1. 在某个支持 C 语言的系统中，执行习题中第 13 题的代码，得到 sum1 和 sum2 的值。解释得到的结果是怎样产生的。
2. 分别用 C++、Java 和 C#重写练习 1 中的程序，并运行它们，比较得到的结果。
3. 用你最喜欢的语言写一个检验程序，确定它的算术和布尔运算符的优先级和结合性，并输出它们。
4. 编写一个 Java 程序，以揭示在操作数之一是函数调用的情况下，操作数的 Java 求值的规则。
5. 用 C++重复练习 4。
6. 用 C#重复练习 4。
7. 用 C++、Java 或 C#编写一个程序，以揭示当表达式用作方法的实参时，这些表达式的求值顺序。
8. 编写一个有如下语句的 C 程序：

```
int a, b;
```

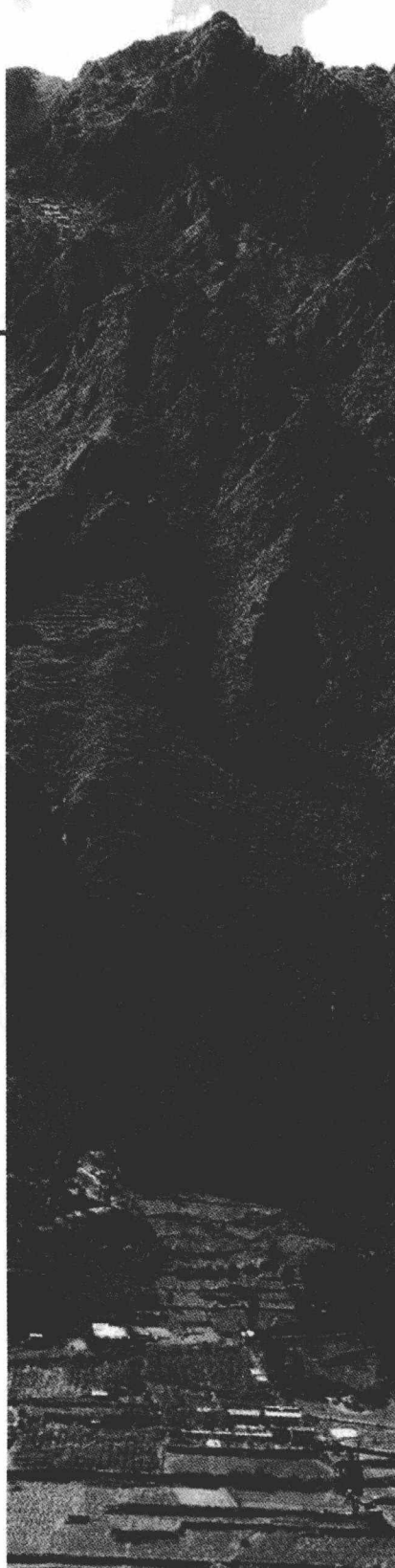
```
a = 10;
b = a + fun();
printf("With the function call on the right, ");
printf(" b is: %d\n", b);
a = 10;
b = fun() + a;
printf("With the function call on the left, ");
printf(" b is: %d\n", b);
```

定义 fun 函数，它的作用是让 a 增加 10。解释所得的结果。

9. 用 Java、C++或 C#编写一个程序，进行大量的浮点数运算和相同数量的整数运算，比较两种运算花费的时间。

第8章

语句级控制结构



8.1 概述
8.2 选择语句
8.3 迭代语句
8.4 无条件分支
8.5 防护命令
8.6 结论

程序的控制流程，或者说执行序列，可以从多个层次来考察。第 7 章讨论了表达式的控制流程，它是由运算符的结合性与优先级规则决定的。最高层次的控制流程，即程序单元之间的控制流程，在第 9 章和第 13 章中讨论。介于上述二者之间的，是语句的控制流程，本章将讨论这个非常重要的内容。

首先概述控制语句的演变过程，之后深入地讨论包括双路选择与多重选择在内的选择语句。然后讨论在编程语言中大量开发使用的循环结构。接着简要介绍无条件转移语句的问题。最后描述防护命令控制结构。

8.1 概述

在命令式语言编写的程序中，计算是通过计算表达式的值，并将结果赋值给变量来实现的。但是，有效的程序极少完全由赋值语句组成。为了让程序更加灵活、强大，两类语言结构是必不可少的：一类是能够在(语句执行中的)多个控制流程路径间进行选择的结构，另一类是能够重复执行某条语句或语句组的结构。提供了这样功能的语句称为控制语句。

函数式编程语言中的计算是通过计算表达式和函数来实现的。而且，表达式和函数中的执行流程由其他表达式和函数来控制，但其中一些类似于命令式语言中的控制语句。

第一个成功的编程语言 Fortran 的控制语句，事实上是由 IBM 704 的设计师设计的。它所有的控制语句都直接与机器指令相关联，所以它们的功能更多的是指令设计而非语言设计的结果。当时，人们对于编程的困难程度知之甚少，所以，在 20 世纪 50 年代中叶产生的 Fortran 语言的控制语句被当时的人们所接受。按照今天的标准，它是完全不够格的。

20 世纪 60 年代中期到 70 年代中期的这 10 年中，人们对控制语句进行了大量的研究和讨论。其主要结论之一是：虽然最少只需一条控制语句(一个可选择的 goto 语句)就能满足要求，但是，不包含 goto 语句的语言只需少量其他的控制语句就能达到同样的效果。实际上，任何能够用流程图来表示的算法，都能用只包含两种控制语句的编程语言实现。其一是能在两种控制流程路径中作出选择的控制语句，其二是带有逻辑控制的循环语句(Böhm 和 Jacopini，1966)。其重要结论是：无条件分支语句是多余的——有潜在价值但绝非必要。这个结论与在使用无条件分支语句或 goto 时遇到的一些实际问题，引发了关于 goto 语句的激烈争论，参见 8.4 节。

程序员不大关心控制语句的理论研究结果，而更关注程序的可写性和可读性。所有广泛应用的语言除那两种必需的控制语句外，还包含更多的控制语句。因为控制语句的数量和种类越多，程序的可写性就越好。例如，在构建本来就用计数器来控制的循环时，不使用带逻辑判断的循环语句，而采用计数器控制的循环语句，显然更加容易编程。语言中控制语句数量的主要限制因素是程序的可读性，因为语句形式越多，程序的阅读者要学习的语言就越庞大。很少有人能学习某种庞大语言的所有语句，相反，他们只学习需要用到的部分。他们学习的部分往往不同于程序员在编写程序时用到的部分。另一方面，控制语句太少，就需要使用像 goto 这样的低级语句，这会降低程序的可读性。

怎样找到一个最佳的控制语句集合，它既能满足功能的要求，又能满足可写性的要求，

是一个广受争议的问题。它的本质是：语言应该扩展到什么程度，来提高程序的可写性，但需要以程序的简明性、长度、可读性为代价。

控制结构包含控制语句和被它控制执行的语句集合两部分。

只有一个设计问题是和所有的选择与循环控制语句有关的：控制结构应该有多个入口吗？所有的选择和循环结构控制着代码片段的执行，问题就在于这些代码片段的执行是否总是从第一条语句开始呢？目前一般认为，多个入口对控制结构的灵活性帮助甚微，反而由于复杂性的增加，而降低了可读性。注意，多个入口只能在包含 goto 语句和标签语句的语言中实现。

那么，为什么控制结构中的多重出口不是一个设计问题呢？原因是所有编程语言都允许控制结构有某些形式的多重出口。基本原理如下：如果控制结构的所有出口都仅允许跳转到控制结构之后的第一条语句，这条语句也是在控制结构没有显式的出口时应执行的下一条语句，那么就不会影响可读性，更没有危险。然而，如果出口跳转的目标不受限制，因而很可能会跳转到包含控制结构的某个程序单元中，那么，这对可读性的影响与程序中任意使用 goto 语句相同。有 goto 语句的语言允许 goto 语句在程序的任意地方出现，包括控制结构。因此，问题就变成应不应该使用 goto，而不是应不应该在控制表达式中使用多重出口。

8.2　选择语句

选择语句提供了从 2 个或多个执行路径中选择的方法。它非常基础，经 Böhm 和 Jacopini 证明，它是所有编程语言的必要成分。

选择语句分为两个大类：双路选择与 n 路选择(多重选择)。第 8.2.1 节讨论双路选择语句，而多重选择语句在 8.2.2 节详述。

8.2.1　双路选择语句

虽然当今命令式语言中的双路选择语句十分相似，但也有一些设计上的变化。双路选择器的一般形式如下：

```
if 控制表达式
   then 子句
   else 子句
```

8.2.1.1　设计问题

双路选择器的设计问题可以总结为：

- 控制选择方向的表达式的形式与类型是怎样的？
- then 和 else 子句如何确定？
- 嵌套选择器的含义如何确定？

8.2.1.2 控制表达式

如果不使用 then 保留字(或其他的语法记号)来引导 then 子句，控制表达式就放在圆括号中。而在使用 then 保留字引导 then 子句时，就不需要使用圆括号，就像 Ruby 语言中的一样，经常省略圆括号。

C89 语言没有布尔数据类型，所以把算术表达式用作控制表达式。Python、C99 与 C++ 也兼容这样的语法。在这些语言中，算术表达式和布尔表达式都可以作为控制表达式。然而在其他的现代语言中，只能将布尔表达式作为控制表达式。

8.2.1.3 子句的形式

在许多当代语言中，then 和 else 子句表示为单条语句或复合语句。一个与此有关的变化是在 Perl 语言中，所有的 then 和 else 子句必须是复合语句，即使它们只包含一条语句。许多语言用大括号构成复合语句，来作为 then 和 else 子句的内容。在 Fortran 95、Ada、Python 和 Ruby 语言中，then 和 else 子句是语句序列，而不是复合语句。在这些语言中，使用一个保留字来结束完整的选择结构[1]。

Python 语言用缩进来指明复合语句。例如：

```
if x > y :
  x = y
  print "case 1"
```

所有等量缩进的语句都包括在复合语句中[2]。注意在 Python 语言中，使用冒号来引导 then 子句，而不是用 then。

子句形式的变化会影响嵌套选择器的含义，参见下一小节。

8.2.1.4 嵌套选择器

第 3 章讨论了双路选择器结构的简单语法引起的语法歧义问题。语法如下：

```
<if 语句>→if <逻辑表达式> then <语句>
        | if <逻辑表达式> then <语句> else <语句>
```

问题在于，当选择语句嵌套在另一个选择结构的 then 子句中时，无法知道 else 子句与哪一个 if 相关联。这个问题反映在选择语句的语义上。考察下列类似 Java 的代码：

```
if (sum == 0)
  if (count == 0)
    result = 0;
else
    result = 1;
```

1. 事实上，Ada 与 Fortran 使用两个保留字，分别为 end if(Ada)和 End If(Fortran)。

2. 在复合语句之后的语句必须有与 if 相同的缩进。

这段代码可以用两种不同的方式解释，关键在于 else 子句与前一个还是后一个 then 子句相匹配。程序的缩进似乎说明 else 子句属于第一个 then 子句。但是，在当代的语言中，除了 Python 和 F#之外，缩进对于语法没有影响，因此被编译器忽略。

在这个例子中，问题的关键是 else 子句跟在两个 then 子句之后，而它们之间没有另外的 else 子句。也没有语法指示器来确定 else 子句与哪个 then 子句相匹配。Java 语言与许多命令式语言一样，语言的静态语义规定，else 子句总是和在它之前离它最近的那个 then 子句相配对。这条静态语义规则，而不是语义实体，消除了歧义。所以在上例中，else 子句与第二个 then 子句匹配。使用规则而不是语义实体的缺点是：程序员想让 else 子句与第一个 then 子句匹配时，编译器认为结构的语法是正确的，但它的语义却相反。在 Java 中，为了强制使用另一种语义，内层的 if 要包含在一个复合语句中，如下：

```
if (sum == 0) {
  if (count == 0)
    result = 0;
}
else
  result = 1;
```

C、C++和 C#语言也存在和 Java 语言相同的选择语句嵌套问题。因为 Perl 规定，所有的 then 和 else 子句必须是复合语句，所以这个问题不存在。在 Perl 语言中，上述代码可写成：

```
if (sum == 0) {
  if (count == 0) {
    result = 0;
  }
} else {
  result = 1;
}
```

如果需要另一种语义，代码为：

```
if (sum == 0) {
  if (count == 0) {
    result = 0;
  }
  else {
    result = 1;
  }
}
```

另一种避免选择语句嵌套问题的途径是使用另一种方法构成复合语句。考察 Java 中 if 语句的语法结构。then 子句跟在控制表达式后面，而 else 子句跟在 else 保留字后面。当 then 子句只有一条语句，并且出现了 else 子句时，虽然不需要标记结束点，但事实上 else 保留字标志了 then 子句的结束。当 then 子句是复合语句时，它用右大括号来结束。然而，如果 if 语句中的最后一个子句，无论是 then 子句还是 else 子句，不是复合语句，那么就没有语法实

体来标志整个选择结构的结束。为此使用一个特殊的保留字，来解决嵌套选择器的语义问题，同时增强了语句的可读性。这是 Fortran 95+、Ada、Ruby 和 Lua 语言中选择结构的设计方法。例如下面的 Ruby 语句：

```
if a > b then
  sum = sum + a
  acount = acount + 1
else
  sum = sum + b
  bcount = bcount + 1
end
```

这种选择结构的设计比基于 C 的语言更加规范，因为无论 then 和 else 子句由多少条语句构成，选择结构的形式都是相同的(Perl 也有这样的特点)。在 Ruby 语言中，then 和 else 子句由语句序列而不是复合语句组成。对本小节开始的选择器例子的第一种理解，是 else 子句与内层嵌套的 if 相匹配，在 Ruby 语言中，这可以写为：

```
if sum == 0 then
  if count == 0 then
    result = 0
  else
    result = 1
  end
end
```

end 保留字标志着嵌套 if 的结束，所以显然，else 子句与内层的 then 子句匹配。

对本小节开头的选择结构的第二种理解，是 else 子句与外层的 if 相匹配，在 Ruby 语言中，这可以写为：

```
if sum == 0 then
  if count == 0 then
    result = 0
  end
else
  result = 1
end
```

下面的 Python 语句与上面的 Ruby 语句是语义等价的：

```
if sum == 0 :
  if count == 0 :
    result = 0
else:
  result = 1
```

如果 else:一行与嵌套的 if 有同样的缩进，则 else 子句就与内层的 if 相匹配。

ML 不存在选择器的嵌套问题，因为它不支持没有 else 的 if 语句。

8.2.1.5　选择器表达式

在函数式语言 ML、F#和 LISP 中，选择器不是一个语句，而是一个结果为一个值的表达式。因此，它可以出现在其他表达式能出现的任何位置。考虑下面的 F#示例选择器：

```
let y =
    if x > 0 then x
    else 2 * x;;
```

该语句会创建名字 y，并根据 x 是否大于 0，将 y 设置为 x 或 2*x。

F#的 if 不需要返回值，例如，如果其子句有副作用，就可能带有输出语句。但如果 if 表达式有返回值，如上例所示，它就必须有 else 子句。

8.2.2　多重选择结构

多重选择结构允许在任意数量的语句或语句组中选择。所以多重选择结构是一个通用的选择器。事实上，双路选择器可以用多重选择器来实现。

在程序中，经常需要在多于两个的控制路径中进行选择。虽然多重选择器可以由双路选择器和 goto 语句组成，但这样的结构是冗长的、不可靠的、难以读写的。所以，显然需要有一个专门的结构。

8.2.2.1　设计问题

多重选择器的一些设计问题与双路选择器是相似的，例如选择器中表达式的类型问题。对于这个问题，选择的范围更大，部分原因是可能的选择更多了。双路选择器只需要一个表达式来选择两个可能的值。另一个问题是选择执行的是单条语句、复合语句，还是语句序列？紧接着的另一个问题是执行选择结构时，是否只执行一个可选段？双路选择器没有这个问题，因为在执行过程中，它们仅允许在一个控制路径上有一个子句。解决多重选择器的这个问题，需要可靠性和灵活性之间的协调。还有一个问题是 case 值的规范形式。最后一个问题是，若没有根据选择器表达式的值选择一个可选语句段，会导致什么结果？(表达式的值没有出现在可选语句段中)。这里的选择是简单地禁止这种情形的产生，或者当它产生时，什么也不做。

下面是这些设计问题的小结：

- 控制这个选择的表达式是什么形式和类型？
- 可选择语句段是怎样指定的？
- 通过结构的执行流是否限定为只包含一个可选段？
- case 值是怎样指定的？
- 如果选择器表达式的值根本就无法表述，那应该如何处理它呢？

8.2.2.2　多重选择器的例子

C 中多重选择器的结构 switch，在 C++、Java 和 JavaScript 中也有，它的设计相对简

单。它的语法形式是：

```
switch(控制表达式){
 case 常量表达式1: 语句1;
 ...
 case 常量表达式n: 语句n;
 [default: 语句n+1]
}
```

这里控制表达式和常量表达式都是离散类型。它包括整数类型、字符类型和枚举类型。可选语句可以是语句序列、复合语句或块。可选的 default 语句段用来处理控制表达式没有的值。假如控制表达式没有提供某个值，也没有提供 default 语句段，则这个结构什么也不执行。

switch 结构在代码的最后没有提供隐式的分支。这使控制流在一次执行过程中，能处理多个可选代码段。考虑下面这个例子：

```
switch (index) {
  case 1:
  case 3: odd += 1;
          sumodd += index;
  case 2:
  case 4: even += 1;
          sumeven += index;
  default: printf("Error in switch, index = %d\n", index);
}
```

这段代码会输出每次执行时的错误信息。同样，每次执行常量 1 或常量 3 后的代码段时，也会执行常量 2 和常量 4 后的代码。要从逻辑上将这些片段分开，必须包含一个显式分支。break 语句(实际上是个受限制的 goto 语句)通常用来退出 switch 结构。

下面的 switch 结构使用 break 语句，将每次执行限制为只执行一个可选语句段：

```
Switch (index) {
  case 1:
  case 3: odd += 1;
          sumodd += index;
          break;
  case 2:
  case 4: even += 1;
          sumeven += index;
          break;
  default: printf("Error in switch, index = %d\n", index);
}
```

有时，允许控制流从一个可选代码段流入另一个可选代码段是很方便的。例如，在上面的例子中，case 1 和 case 2 后的代码段是空的，这就允许控制流分别从 1 和 2 流入 3 和 4。这也是在 switch 结构里没有隐式分支的原因。在一个代码段中，当该出现而未出现的 break 语句允许控制流错误地流入下一代码段时，这个设计的可靠性就会出现问题。C 语言 switch

结构的设计者降低了可靠性，以提高灵活性。然而，研究表明，实际上很少使用令控制流从一个可选代码段流入另一代码段的功能。C 语言的 switch 结构是以 ALGOL 68 中的多重选择语句为原型，在原型的可选语句段里是没有隐式分支的。

C 语言的 switch 结构几乎没有限制 case 表达式的位置，就好像它们是普通的语句标号。这种“懒惰”可能会导致 switch 结构体里出现高度复杂的结构。下面的例子摘自 Harbison 和 Steele(2002)：

```
switch (x)
  default:
  if (prime(x))
    case 2: case 3: case 5: case 7:
      process_prime(x);
  else
    case 4: case 6: case 8: case 9: case 10:
      process_composite(x);
```

这段代码看起来非常复杂，但它是为解决一个实际问题而设计的，并能正确有效地执行，并解决了这个问题[3]。

而 Java 的 switch 语句只允许 case 表达式出现在 switch 语句体的顶层，以防止出现上面那样的复杂情况。

C#的 switch 语句与其他基于 C 的语言有两点不同。首先，C#有一条静态语义规则，它不允许隐式执行多于一段的代码。这个规则是：每个可选语句段都必须以一条显式的无条件分支语句来结束，这个无条件分支语句要么是一个 break，将控制移出 switch 结构；要么是一个 goto 语句，将控制移到另一个可选语句段(或任何其他地方)。例如：

```
switch (value) {
   case -1:
      Negatives++;
      break;
   case 0:
      Zeros++;
      goto case 1;
   case 1:
      Positives++;
   default:
      Console.WriteLine("Error in switch \n");
}
```

注意 Console.WriteLine 是 C#中显示字符串的方法。

C#的 switch 语句不同于其他基于 C 的语言的另一方面是，C#中的控制表达式与 case 结构可以是字符串。

---

3. 这个问题是：当 x 是素数时，调用 process_prime，否则调用 process_composite。这个 switch 主体的设计是针对 x 大都在 1 到 10 范围内这个信息而进行的优化。

PHP 的 switch 结构使用了 C 中 switch 结构的语法，但是表达式的类型更加灵活。case 表达式的值可以是 PHP 的任何标量类型——字符串、整型或双精度型。和 C 一样，如果在被选中语句段的末尾没有 break，就会继续执行下一段。

Ruby 有两种形式的多重选择结构，都称作 case 表达式，都生成最后一个被计算的 case 表达式的值。其中一种形式在语义上类似于一串嵌套的 if 语句：

```
case
when 布尔表达式 then 表达式
...
when 布尔表达式 then 表达式
[else 表达式]
end
```

它的语义是：自顶向下，一次只求一个布尔表达式的值。case 表达式的值是第一个取值为真的布尔表达式所对应的 then 后面的表达式的值。else 语句在这个结构中始终为真，它是可选的。例如[4]：

```
Leap = case
       when year % 400 == 0 then true
       when year % 100 == 0 then false
       else year % 4 == 0
       end
```

如果 year 是一个闰年，这个 case 表达式就为真。

另一个 case 表达式的形式类似于 Java 的 switch 语句。Perl、Python 和 Lua 都没有多重选择语句。

8.2.2.3 多重选择结构的实现

从本质上说，多重选择结构是一个有 n 路分支的代码段，其中 n 是可选择语句段的数目。这个结构需要通过多个条件分支指令实现。C 中带 break 分支的 switch 结构的一般形式如下：

```
switch (表达式) {
 case 常量表达式 1: 语句 1;
  break;
 ...
 case 常量表达式 n: 语句 n
  break;
 [default: 语句 n+1]
}
```

对上述结构的一个简单解释如下：

将求值表达式的代码存入 t 中

4. 这个例子取自 Thomas 等(2005)。

```
goto branches
label1：语句 1 的代码
        goto out
...
labeln：语句 n 的代码
        goto out
default：语句 n+1 的代码
        goto out
branches：if t=常量表达式 1 goto label1
          ...
          if t=常量表达式 n goto labeln
          goto default
out：
```

可选语句段的代码在分支的前面，因此分支的目标在代码进入分支时就是已知的。另一种实现编码条件分支的方式是把 case 值和标号存放在一个表中，然后用一个循环，线性查找到正确的标号。这比编码条件分支需要更少的空间。

无论是使用条件分支，还是在 case 值和标号的表中进行线性查找，都属于简单、低效的方法，仅在 case 数量比较少(比如说小于 10)时可行。在每次找到正确的 case 之前，要进行的平均测试数约为 case 总数的一半。如果最后选择的是 default case，就必须测试所有其他的 case。当 case 数超过 10 时，这种方法的低效带来的坏处就超过了它的简单所带来的好处。

当 case 的数量大于等于 10 时，编译器会建立一个可选语句段标号的散列表，于是选择任意可选语句段的时间都大致相等。如果编程语言允许用数值范围作为 case 表达式，就像 Ada 和 Ruby 一样，这种散列表的方法就不适用了。对于这些情况，case 值和可选语句段地址的二分搜索表会更适用。

如果 case 值的范围相对较小，一半以上的值都可以表述，则可以建立一个数组，数组下标表示 case 值，元素值表示语句段的标号。而没有在 case 值之内的下标，对应的元素值便是 default 段的标号。所以，可以通过数组的索引找到正确的代码段标号，这样就非常快。

当然，从这些方法中选择是编译器的一个额外负担。在很多编译器中，只有两种方法是可行的。在其他的编译器中，决定并运用最高效的方法反而花费了编译器更多的时间。

8.2.2.4　用 if 实现多重选择

在很多情况下，为多重选择使用 switch 或者 case 结构是不够的(Ruby 的 case 是一个例外)。例如，必须根据布尔表达式而不是某种有序类型来选择时，嵌套的双路选择器可以用来模拟一个多重选择器。为了弥补深度嵌套的双路选择器可读性太差的缺陷，一些语言，例如 Perl 和 Python，特意为此做了一些扩充。这个扩充会允许省去一些特殊字。具体而言，else-if 语句序列用单个特殊字代替，并去掉嵌套 if 的结束特殊字。这个嵌套选择器叫做 else-if 子句。下面是一段 Python 选择器的代码(注意在 Python 中 else-if 拼写为 elif)：

```
if count < 10 :
  bag1 = True
```

```
elif count < 100 :
  bag2 = True
elif count < 1000 :
  bag3 = True
```

它等价于：

```
if count < 10 :
  bag1 = True
else :
  if count < 100 :
    bag2 = True
  else :
    if count < 1000 :
      bag3 = True
    else :
      bag4 = True
```

在这两个例子中，else-if 语句(第一个)的可读性较高。注意，这个例子用 switch 语句是不容易模拟的，因为每个可选语句是依据布尔表达式来选择的。因此，else-if 结构不是 switch 的冗余形式。事实上，在当代语言中，没有一个多重选择器是和 if-then-else-if 结构一样普遍的。对于带有 else-if 子句的一般选择语句，其操作语义的描述如下，其中 E 是逻辑表达式，S 是语句：

```
   if E1 goto 1
   if E2 goto 2
   ...
1: S1
   goto out
2: S2
   goto out
...
out: ...
```

从这个描述中可以看到多重选择结构和 else-if 结构之间的不同之处：在多重选择结构里，所有的 E 都限制为单个表达式的值与其他一些值之间的比较。

不含有 else-if 结构的语言也可以使用相同的控制结构，不过要输入更多的代码。

之前 if-then-else-if 结构的 Python 例子可以用 Ruby 的 case 语句写为：

```
case
when count < 10 then bag1 = true
when count < 100 then bag2 = true
when count < 1000 then bag3 = true
end
```

else-if 结构基于常见的数学结构，即条件表达式。

Scheme 的多重选择器基于数学条件表达式，是一种特殊形式的函数，称为 COND。COND 是数学条件表达式的一个略微扩展的版本，它允许多个谓词同时为真。因为不同的

数学条件表达式有不同数目的参数，所以 COND 不需要固定数目的实参。COND 的每个参数都是一对表达式，其中第一个表达式是一个谓词(它计算为#T 或#F)。

COND 的一般形式如下：

```
(COND
 (谓词 1 表达式 1)
 (谓词 2 表达式 2)
 ...
 (谓词 n 表达式 n)
 [(ELSE 表达式 n+1)]
)
```

其中 ELSE 子句是可选的。

COND 的语义如下：参数中的谓词一次计算一个，其顺序是从第一个谓词开始，直到所计算的谓词等于#T 为止。接着计算跟在第一个结果为#T 的谓词后面的表达式，把其值返回为 COND 的值。如果所有的谓词都不是真，且有 ELSE 子句，就计算 ELSE 子句的表达式，并返回它的值。如果所有的谓词都不是真，且没有 ELSE 子句，就不指定 COND 的值。因此所有的 COND 都应包含 ELSE 子句。

下面的例子调用了 COND：

```
(COND
  ((> x y) "x is greater than y")
  ((< x y) "y is greater than x")
  (ELSE "x and y are equal")
)
```

注意，字符串字面量计算后的结果是其本身，所以调用这个 COND 时，会生成一个字符串结果。

F#引入了匹配(match)表达式，它使用模式匹配作为选择器，来提供多重选择结构。

8.3　迭代语句

迭代语句能使一条语句或一块语句执行零次、一次或者多次。迭代语句一般称为循环。从 Plankalkül 开始的每种编程语言都包含能使一段代码重复执行的方法。迭代是计算机能力的精华。如果不能采用某种方式重复执行一条语句或一块语句，程序员就必须按顺序声明所有动作。有用的程序将变得很大、不灵活，还会花费大量的时间编写，用大量内存空间来存储。

编程语言中，最初的迭代语句与数组直接相关。这是因为，在计算机时代的最初几年，计算基本上是数值运算，经常使用循环来处理数组中的数据。

人们开发了几种迭代控制语句。主要的类别是由设计者回答两个基本设计问题的方式来决定的：

- 怎样控制语句的迭代？
- 控制机制在循环语句中的哪里体现？

主要的迭代控制方法是用逻辑条件、计数或者二者的结合。控制机制的位置主要是在循环的开始处或结束处。这里的开始和结束是逻辑上的，而不是字面意义上的。问题不是控制机制的物理位置，而是控制机制是否执行，是在循环体之前还是之后影响循环。第三个方法允许用户选择将控制机制放在哪里，详见 8.3.3 节。

迭代语句体是迭代语句控制执行的一段语句。我们用“前判断”来表示在循环体执行之前进行循环结束测试，用“后判断”来表示在循环体执行之后进行循环结束测试。迭代语句和与它关联的循环体一起构成了迭代结构。

除了主要的迭代语句之外，我们还要讨论一种替代形式：用户定义的迭代控制。

8.3.1 计数控制循环

计数控制语句包含一个变量，叫做循环变量，它记录着目前的计数。这个控制语句也包含循环变量的起始和终止方式，以及循环变量前后的差值，即步长。循环的起始、终止和步长称为循环参数。

虽然逻辑控制循环比计数控制循环更一般，但它不一定更常用，因为计数控制循环更复杂，设计要求更高。

计数控制循环有时会得到机器指令的支持。但是，机器体系结构的普及有时会比设计它时流行的编程语言滞后。例如，VAX 计算机有一个非常方便的指令，来实现后置计数控制循环，这个功能当时 Fortran 语言也有(20 世纪 70 年代中期)。但是当 VAX 计算机广泛使用时，Fortran 语言就不再支持这个循环了(被条件语句前置循环所替代)。

8.3.1.1 设计问题

迭代计数控制循环语句在设计上有很多问题。循环变量和循环参数本身就有一系列设计问题。循环变量与循环参数的类型显然是相同的，至少是兼容的，但是应该允许什么样的类型呢？一个很明显的选择是整数类型，那枚举类型、字符和浮点数怎么样呢？另一个问题是，循环变量是应该与普通变量有相同的作用域，还是有特殊的作用域。允许用户修改循环体内的循环变量或循环参数，代码就很难读懂，所以另一个问题是，允许用户进行这个修改所带来的灵活性，是否抵得上因此而增加的复杂性。类似的问题还有循环变量的计算次数和计算时刻。如果循环变量只计算一次，循环是简单了，但失去了一些灵活性。

以下是对这些设计问题的总结：

- 循环变量的类型和作用域是什么？
- 在循环体中修改循环变量和循环参数是否合法？如果合法，这样的修改会影响循环控制吗？
- 循环参数应该只计算一次，还是每次迭代都计算？

8.3.1.2　Ada 语言中的 for 语句

Ada 的 for 语句是下面这种形式：

```
for 变量 in[reverse]离散范围 loop
  ...
end loop;
```

离散范围是整数或枚举类型的子范围，比如 1..10 或周一..周五。保留字 reverse 出现时，表示离散范围中的值是按相反的顺序分配给循环变量的。

Ada 中 for 语句的最有意思的新功能是循环变量的作用域，也就是循环的范围。变量在 for 语句中隐式地定义，在循环停止时隐式地删除。例如：

```
Count : Float := 1.35;
for Count in 1..10 loop
  Sum := Sum + Count;
end loop;
```

浮点变量 Count 不受 for 循环的影响。当循环停止时，变量 Count 仍然是浮点类型，值是 1.35。同时，浮点变量 Count 在循环体中是隐藏的，被循环计数器 Count 覆盖了，这个 Count 隐式地声明为整数的离散区域。

Ada 的循环变量不能在循环体内赋值。用于指定离散区域的变量可以在循环中改变，但是因为范围只能指定一次，所以这些改变不影响循环控制。通过分支进入 Ada 的 for 循环体是不合法的。下面是 Ada 中 for 循环的操作语义描述：

```
    [定义 for 变量(类型是离散范围)]
    [离散范围求值]
loop:
    if [离散范围中没有剩余元素] goto out
    for 变量=[离散范围中的下一个元素]
    [循环体]
    goto loop
out:
    [取消 for 变量的定义]
```

由于循环变量的作用域是循环体，且循环变量在循环结束后没有定义，所以它们的值在循环结束后就没有意义了。

8.3.1.3　基于 C 的语言的 for 语句

C 中 for 语句的一般形式为：

```
for(表达式 1；表达式 2；表达式 3)
 循环体
```

循环体可以是一条单个的语句、复合语句或空语句。

由于 C 中的赋值语句能产生结果，所以可以认为是表达式，for 语句中的表达式经常

是赋值语句。第一个表达式用于初始化，只在 for 语句开始执行时计算一次。第二个表达式用于循环控制，在每次执行循环体之前都要计算。在 C 中，0 意味着假，所有非零值意味着真。所以当第二个表达式的值为 0 时，for 循环终止；否则执行循环体。在 C99 中，这个表达式也可以是布尔类型。C99 的布尔类型只存储 0 或 1。for 语句中的最后一个表达式在每次循环体执行后计算。它常常用于递增循环计数器。C 中 for 语句的操作语义描述将在下文中给出。由于 C 中的表达式可以用作语句，表达式的计算将显示为语句。

```
      表达式 1
loop:
      if 表达式 2 = 0 goto out
      [循环体]
      表达式 3
      goto loop
out: ...
```

下面是 C 中一个简单的 for 结构例子：

```
for(count=1; count <= 10; count++){
  ...
}
```

C 的 for 语句中的三个表达式都是可选的。当省略第二个表达式时，编译器会将它的值设置为 true，所以没有第二个表达式的 for 语句就是一个死循环。如果省略第一个和/或第三个表达式，编译器就不会做出任何假设。例如，省略第一个表达式时，仅仅说明没有初始化。

注意，C 的 for 语句不需要计数。它很容易构建计数与逻辑循环结构，如下一节所示。

C 中 for 语句的设计方案是：没有显式的循环变量或循环参数；所有相关的变量都能在循环体中修改；表达式将按照之前定义的顺序执行；分支进入 for 的循环体也是合法的，尽管这样会造成破坏。

C 的 for 语句比 Ada 的计数循环语句更灵活，因为它的三个表达式都可以包含多个子表达式，因此可以使用多个任何类型的循环变量。for 语句中的一个表达式使用多个子表达式时，它们之间用逗号分隔。C 的所有语句都有值，这种包含多个表达式的语句也不例外。其值就是最后一个表达式的值。

考虑下面的 for 语句：

```
For (count1 = 0, count2 = 1.0;
     count1 <= 10 && count2 <= 100.0;
     sum = ++count1 + count2, count2 *= 2.5);
```

这段代码的操作语义描述如下：

```
    count1 = 0
    count2 = 1.0
loop:
    if count1 > 10 goto out
```

```
        if count2 > 100.0 goto out
        count1 = count1 + 1
        sum = count1 + count2
        count2 = count2 * 2.5
        goto loop
out: ...
```

在这个例子中，for 语句不需要也没有循环体。所有需要的操作都在 for 语句中，而不在循环体中。这个 for 语句的第一个和第三个表达式有多条语句，且执行了这两个表达式中的所有语句，但是在循环控制中并没有使用它们的值。

C99 和 C++的 for 语句与早期版本的 C 中的 for 语句在两方面有不同之处。第一，除了算术表达式之外，它们还能用布尔表达式来控制循环；第二，第一个表达式可以包含变量的定义，比如：

```
for (int count = 0; count < len; count++) { . . . }
```

在 for 语句中定义的变量的作用域是从变量定义开始到循环体结束。

Java 和 C#中的 for 语句与 C++类似，除了循环控制表达式只能是布尔类型。

在所有基于 C 的语言中，for 语句的后两个循环参数在每次循环时都计算一次。而且，循环参数表达式中的变量可以在循环体中修改。因此，这些循环可以非常复杂，常常比 Ada 的计数循环更不可靠。

8.3.1.4　Python 中的 for 语句

Python 中 for 语句的一般形式是：

```
for 循环变量 in object:
 -循环体
[else:
 -else 子句]
```

循环变量在每次循环时，都指定为 object 中的一个值，而 object 一般是一个值域。如果 else 子句存在，就在循环正常终止后执行。

考虑下面的例子：

```
for count in[2, 4, 6]:
  print count
```

输出

```
2
4
6
```

在 Python 的大多数简单计数循环中，都使用了 range 函数。range 函数带有一个、两个或三个参数。下面的例子说明了 range 函数的用法：

```
range(5)返回[0, 1, 2, 3, 4]
range(2, 7) 返回 [2, 3, 4, 5, 6]
range(0, 8, 2) 返回 [0, 2, 4, 6]
```

注意，range 函数从不返回所给参数范围内的最大值。

8.3.1.5 函数式语言中的计数控制循环

命令式语言中的计数控制循环使用计数变量，但在纯函数式语言中不存在这种变量。函数式语言不使用迭代来控制语句的重复执行，而是使用递归来控制。函数式语言不使用语句，而使用递归函数。在函数式语言中，可以模拟计数循环，具体如下：计数器可以是一个函数的参数，该函数重复执行循环体，而循环体在第二个函数中指定，第二个函数作为参数发送给循环函数。所以，这个循环函数的参数是循环体函数和重复次数。

模拟计数循环的 F#函数 forLoop 的一般形式如下：

```
let rec forLoop loopBody reps =
    if reps <= 0 then
        ()
    else
        loopBody()
        forLoop loopBody, (reps - 1);;
```

在这个函数中，loopBody 参数是包含循环体的函数，reps 参数是重复次数。保留字 rec 位于函数名之前，表示它是递归函数。空括号表示什么也不做，包含空括号是因为在 F# 中，空语句是不合法的，每个 if 语句都必须有 else 子句。

8.3.2 逻辑控制循环

在很多情况下，都必须重复执行大量语句，而对这种重复执行的控制基于一个布尔表达式，而不是计数器。这时，使用逻辑控制循环就很方便。事实上，逻辑控制循环比计数控制循环更具有普适性。任何一个计数循环都可以用逻辑控制循环来构建，但反之不一定成立。而且，只有选择语句和逻辑循环才是流程图中控制结构的基本组成要素。

8.3.2.1 设计问题

由于逻辑控制循环比计数控制循环要简单很多，所以它的设计问题也要少一些。

- 控制应当是前测试型还是后测试型？
- 逻辑控制循环应当是一种特殊形式的计数循环还是其他形式的语句？

8.3.2.2 例子

基于 C 语言的编程语言包括前测试型循环和后测试型循环，但它们不是计数控制循环的特殊形式。以下分别是前测试型与后测试型循环的一般形式：

```
while(控制表达式)
  循环体
```

和

```
do
  循环体
while(控制表达式);
```

以下的 C#代码就用到了这两种语句形式：

```
sum = 0;
indat = Int32.Parse(Console.ReadLine());
while (indat >= 0) {
  sum += indat;
  indat = Int32.Parse(Console.ReadLine());
}

value = Int32.Parse(Console.ReadLine());
do {
  value /= 10;
  digits ++;
} while (value > 0);
```

注意在这个例子中，所有的变量均为整型。Console 对象的 ReadLine 方法从键盘中读入了一行文本，Int32.Parse 在字符串参数中找出数字，将之转化为 int 类型，并返回它。

在前测试型循环(while)中，只要表达式的值为真，就执行循环体。而在后测试型循环语句(do)中，循环体会反复执行，直到表达式的值为假为止。do 语句与 while 语句的唯一差别是，do 语句使循环体至少执行一次。在这两种循环中，循环体均可以是复合语句。这两种语句在操作上的语义描述如下：

```
while

loop:
 if 控制表达式为假 goto out
 [循环体]
 goto loop
out: ...
```

```
do-while

loop:
 [循环体]
  if 控制表达式为真 goto loop
```

在 C 和 C++语言中，都可以通过分支进入 while 与 do 循环体。C89 版本使用算术表达式来控制循环，而在 C99 和 C++中，算术和布尔表达式均可用于控制循环。

Java 中的 while 与 do 语句与 C 和 C++中的类似，不过控制表达式必须是 boolean 类型。而且，由于 Java 中没有 goto 语句，所以循环体只能从最开始处进入。

后测试型循环并不常用，而且在某种程度上比较危险，因为有时程序员会忘记，此时循环体至少会执行一次。在语法上将后测试型循环控制语句放在循环体之后，是因为在这

里它起到了语义上的作用，从而避免了逻辑上不清晰等问题。

前测试型循环可以在纯函数式语言中用递归函数来模拟，该递归函数类似于 8.3.1.5 节中模拟计数循环语句的函数。在这两种情况下，循环体写作一个函数。下面是 F#所模拟的前测试型循环的一般形式：

```
let rec whileLoop test body =
    if test() then
        body()
        whileLoop test body
    else
        ();;
```

8.3.3 用户自定义的循环控制机制

在某些情况下，程序员不将循环控制语句放在循环体头部或末尾会更加方便。于是一些语言提供了这一功能。用户自定义的循环控制的语法机制相对简单，所以它的设计并不困难。这种循环具有死循环的结构，但包括了用户定义的循环退出语句。也许最有趣的问题是只能退出单个循环还是可以退出几个嵌套的循环。这种机制的设计问题是：

- 条件语句是退出机制中不可或缺的一部分吗？
- 只能从一个循环体中退出，还是能从封闭的多个嵌套循环中退出？

C、C++、Python、Ruby 与 C#语言都有无条件、无标志的退出语句(break)。Java 与 Perl 语言有无条件、带标志的退出语句(Java 中的 break 与 Perl 中的 last)。

下面是 Java 中一个嵌套循环的例子，其中有一条 break 语句，用于从嵌套循环中跳到外层循环：

```
outerLoop:
  for (row = 0; row < numRows; row++)
    for (col = 0; col < numCols; col++) {
      sum += mat[row][col];
      if (sum > 1000.0)
        break outerLoop;
   }
```

C、C++和 Python 包括一种无标号的控制语句 continue，它可以将控制转到包含它的最内层循环的控制机制上。这并不是退出循环，而仅是在不中止循环结构的前提下，跳过当前循环中的剩余部分。例如，考虑如下代码：

```
while (sum < 1000) {
  getnext(value);
  if (value < 0) continue;
  sum += value;
}
```

输入一个负值，会跳过赋值语句，于是语句回到循环顶部的条件语句，重新开始执行。另一方面：

```
while (sum < 1000) {
  getnext(value);
  if (value < 0) break;
  sum += value;
}
```

输入一个负值，会终止循环。

last 与 break 都为循环提供了多个出口，这似乎也略微降低了程序的可读性。然而，由于经常需要在一些特殊情况下终止循环，所以这种结构设计是合理的。而且，可读性并没有受到太大破坏，因为这些循环退出语句仅仅是跳转到循环结束的第一条语句(或者是相邻外层循环中)，而不是程序代码中的任意位置。最后，如果采用其他方法代替使用多重 break 来退出多重循环，会显著降低程序的可读性。

使用用户自定义的循环退出语句的动机是简单的：它们满足了从严格受限的分支语句跳转到其他语句中的需要。goto 语句的跳转目标可以是程序中的任意位置，既可以在 goto 语句之前，也可以在 goto 语句之后。但是，用户自定义的循环退出机制的跳转目标语句只能在循环结束之后，且紧跟在复合语句的结尾。

8.3.4　基于数据结构的迭代

Fortran 中的 Do 语句在整数值时使用一种简单的迭代器。例如下面的语句：

```
Do Count = 1, 9, 2
```

在这个语句中，1 是 Count 的初始值，9 是其最后一个值，步长是 2。整数函数，即迭代器，必须在每次迭代时调用，以计算出 Count 的下一个值(对于本例，给 Count 的上一个值加上 2)，并测试迭代是否应继续进行。

在 Python 中，这个循环可以写为：

```
for count in range [0, 9, 2]:
```

在这个语句中，迭代器命名为 range。这些迭代语句通常用于迭代数组，但迭代器和数组之间并没有关联。

Ada 允许把循环迭代器的取值范围和数组的下标范围通过子范围联系起来。例如，可以定义一个子范围，如下声明所示：

```
subtype MyRange is Integer range 0..99;
MyArray: array (MyRange) of Integer;
for Index in MyRange loop
  ...
end loop;
```

子类型 MyRange 用于声明数组，并迭代数组。以这种方式使用子范围时，就不可能出现下标范围溢出错误。

基于数据的通用迭代语句使用用户定义的数据结构与用户定义的函数(迭代器)，来遍

历数据结构中的元素。迭代器在每次迭代的开头调用，每次调用时，迭代器都以特定的顺序返回数据结构中的一个元素。例如，假设程序使用一棵用户定义的二叉树来存储数据结点，各个结点中的数据必须以特定的顺序处理。每次迭代时，用户自定义的迭代语句都使循环变量指向树中的下一个结点。第一次执行用户自定义的迭代语句时，需要对迭代器进行一次特殊的调用，以获得该树中的第一个元素。为了确保每个结点仅访问一次，迭代器必须记录下它上次访问的结点。因此，迭代器必须具有历史敏感性。用户自定义的迭代语句找不到未访问元素时，就会停止执行。

在基于 C 的语言中，for 语句具有极强的灵活性，可用来模拟用户自定义的迭代语句。再次假设需要处理一棵二叉树中的结点。用 root 变量指向树根，令 traverse 函数设置它的参数，按需要的顺序指向树中的下一个元素。代码如下：

```
for (ptr = root; ptr == null; ptr = traverse(ptr)) {
  ...
}
```

在这段代码中，traverse 函数即为迭代器。

预定义的迭代器用来迭代访问 PHP 中的特殊数组。current 指针指向上次迭代得到的元素。next 迭代器将 current 移到数组中的下一个元素，prev 迭代器将 current 移到数组中的前一个元素。reset 运算符可以将 current 重置为数组中的第一个元素。以下代码显示了$list 数组中的所有元素：

```
reset $list;
print ("First number: " + current($list) + "<br />");
while ($current_value = next($list))
  print ("Next number: " + $current_value + "<br \>");
```

用户自定义的迭代语句在面向对象编程中的作用比它们在之前的软件开发模式中更重要，因为面向对象编程的用户经常使用抽象的数据类型来构建数据结构，特别是集合。这时，数据抽象的作者必须提供用户自定义的迭代语句和它的迭代器，因为用户并不知道这种对象的表示方式。

Java 5.0 添加了 for 语句的一个加强版。若数组中的值或集合中的对象实现了 Iterable 接口，则这个语句可以简化迭代过程(Java 中所有预定义的泛型集合都实现了 Iterable)。例如，如果一个 ArrayList[5]集合 myList 包含字符串，下面的语句就迭代它的所有元素，每次的返回值存入 myElement：

```
for (String myElement : myList) { ... }
```

尽管代码中的保留字是 for，它却常称为 foreach 语句。

C#和 F#(以及其他.NET 语言)也为集合提高了泛型库类。例如，列表有泛型集合类，包括长度可变的数组、栈、队列和字典(散列表)。所有这些预定义的泛型集合都有内置的

5. ArrayList 是一个预定义的泛型集合，实质上是元素为一个任意类型的对象的动态数组。

迭代器，由 foreach 语句隐式地使用。而且，用户还可以定义子句的集合，编写自己的迭代器，这些迭代器也可以实现 IEnumerator 接口，以便在这些集合上使用 foreach。

例如下面的 C#代码：

```
List<String> names = new List<String>();
names.Add("Bob");
names.Add("Carol");
names.Add("Alice");
...
foreach (String name in names)
  Console.WriteLine(name);
```

在 Ruby 中，块是一系列代码，用花括号或 do 和 end 保留字来界定。块可由专门编写的函数用来创建许多有效的结构，包括数据结构的迭代器。这个结构包含一个方法调用后跟一个块。块实际上是一个匿名方法，它作为一个参数发送给在它前面调用的方法。被调用的方法就可以调用该块，输出结果或对象。

Ruby 预定义了几个迭代器方法，例如用于计数控制循环的 times 和 upto、用于数组和散列的简单迭代的 each。例如下面的例子使用了 times：

```
>> 4.times {puts "Hey!"}
Hey!
Hey!
Hey!
Hey!
=> 4
```

注意>>是交互式 Ruby 解释器的提示符号，=>用于表示表达式的返回值。Ruby 的 puts 语句会显示其参数。在这个例子中，把 times 方法发送给对象 4，块也作为参数一起发送。times 方法调用该块 4 次，生成 4 行结果。目标对象 4 是 times 的返回值。

最常见的 Ruby 迭代器是 each，它常常用于遍历数组，并给每个元素应用一个块[6]。为此，允许块带有参数是比较方便的。如果有参数，参数就位于块的开头，用竖杠(|)分隔开。下面的例子使用了一个块参数，演示了 each 的用法：

```
>> list = [2, 4, 6, 8]
=> [2, 4, 6, 8]
>> list.each {|value| puts value}
2
4
6
8
=> [2, 4, 6, 8]
```

在这个例子中，把数组发送给 each 方法，并为数组的每个元素调用块。块生成了输出，

6. 这类似于第 15 章讨论的映射函数。

即数组元素的列表。each 的返回值是发送给它的数组。

Ruby 没有计数循环，但有 upto 方法。例如，可以编写如下代码：

```
1.upto(5) {|x| print x, " "}
```

它输出如下结果：

```
1 2 3 4 5
```

也可以模拟其他语言中的 for 循环语法，如下：

```
for x in 1..5
  print x, " "
end
```

Ruby 实际上没有 for 语句——上述结构会由 Ruby 转换为 upto 方法调用。

下面介绍块的工作原理。yield 语句类似于方法调用，但其中没有接收对象，调用是请求执行与方法调用相关的块，而不是调用方法。只有通过块调用的方法中，才调用 yield 语句。如果块有参数，它们就在 yield 语句的圆括号中指定。块返回的值是块中计算的最后一个表达式的值，这个过程也用于实现内置迭代器，例如 times。

8.4 无条件分支

无条件分支语句可以将执行控制权转移到程序中一个特定的位置。20 世纪 60 年代末，语言设计领域最激烈的一场争论是无条件分支语句是否应该成为高级语言的一部分，如果是，它的使用是否应该有所限制。无条件分支语句 goto，是控制程序语句执行流程的最有力的语句。然而，随意使用 goto 语句会导致严重的问题。goto 语句具有极强的力量与极高的灵活性(所有其他的控制结构都可以通过 goto 语句和选择器来实现)，但正是这种力量使 goto 语句变得十分危险。如果语言设计或编程标准没有限制 goto 语句的使用，goto 语句就会使程序变得极其晦涩难懂，导致高度的不可靠性与高昂的维护成本。

这些问题的直接原因是，goto 语句使任何程序语句都能跟在任何执行序列之后，而不管这条语句在已执行语句的前面还是后面。语句的执行顺序与它们出现的顺序——从上到下这种我们比较熟悉的顺序——基本一致时，程序的可读性是最好的。于是，限制 goto 语句，使其只能在程序中向下跳转，就可以缓解这个问题。它允许 goto 语句跳转到代码块附近，以应对错误或特殊情况，但不允许其构成循环结构。

一些语言没有 goto 语句，比如 Java、Python 和 Ruby。然而，现在流行的大多数语言是包括 goto 语句的。Kernighan 和 Ritchie(1978)认为，goto 语句极容易滥用，但是在 Ritchie 设计的 C 语言中包括 goto 语句。已经淘汰掉 goto 的语言提供了额外的控制语句，它们通常采用循环退出语句的形式，来代替在一些场合下合理应用的 goto 语句。

一种相对较新的语言 C#包括了 goto 语句——尽管它基于的语言之一 Java 并不包含

goto。如 8.2.2.2 节所述，C#可以在 switch 语句中合法地使用 goto 语句。

8.3.3 节讨论的所有循环退出语句实际上都是伪 goto 语句。然而它们是严格受限的 goto 语句，并没有削弱可读性。事实上，可以说它们提高了可读性，因为避免使用它们，会使代码变得难以理解、复杂、不自然。

历史注释

尽管一些有见地的人士早就提出了建议，但还是 Edsger Dijkstra 在给计算界第一份广为传颂的材料中揭示了 goto 语句的危险性。他在信中说，“goto 语句的形式太原始，它给制造程序混乱提供了太多机会”(Dijkstra，1968a)。在 Dijkstra 发表 goto 观点后的最初几年，很多人公开争论是否应禁止或者至少限制 goto 的使用。在一些不支持完全删除 goto 语句的人中，Donald Knuth(1974)是其中一员。他争辩说，有时 goto 语句带来的高效胜过了它对可读性的损害。

8.5　防护命令

Dijkstra(1975 年)提出了一些新颖且形式非常不同的选择结构和循环结构。他的主要目标是提供支持正确的程序设计方式的控制语句，在开发过程中确保程序的正确性，而不是依靠检查或测试已完成的程序，来确保它们的正确性。这种方法在 Dijkstra(1976 年)中也有描述。开发防护命令(Gua-rded Commands)的另一个目标是，在并行程序中有时需要不确定性，参见第 13 章。防护命令还可以增强推理的清晰性，这也是开发的目标之一。简单地说，在防护命令结构中，选择结构的可选代码段可以独立处理，不需要考虑结构的其他部分，常见编程语言的选择结构可不是这样。

之所以在本章中提到防护命令，是因为它们是后来为 CSP(Hoare，1978)和 Ada 的并行编程而开发的两个语言机制的基础。Ada 的并发在第 13 章讨论。防护命令也用来定义 Haskell 中的函数，详见第 15 章。

Dijkstra 选择结构的形式如下：

```
if  <布尔表达式>  ->  <语句>
[]  <布尔表达式>  ->  <语句>
[]   ...
[]  <布尔表达式>  ->  <语句>
fi
```

结束保留字 fi 是起始保留字的反向拼写。这种形式的结束保留字取自 ALGOL 68。称为 fatbars 的小块用于分开受保护的子句，并允许以语句序列作为子句。选择结构中的每一行由一个布尔表达式(哨兵)和一条语句或语句序列组成，称为防护命令。

这个选择结构可以出现多重选择，但是其语义有所不同。在执行过程中，每次执行到该结构时，都要计算每个布尔表达式。如果多个表达式为 true，那么选择执行哪个语句就是不确定的。某个实现可能总是选择与取值为真的第一个布尔表达式相关联的语句。但是，

它也有可能选择与其值为真的布尔表达式相关的任何语句。所以，程序的正确性并不取决于(从所有与其值为真的布尔表达式相关的语句中)选择了哪个语句。如果没有一个布尔表达式的值为真，就报告运行时错误，而终止程序。这就迫使程序员考虑并列举出所有的可能性。考虑下面的例子：

```
if i = 0 -> sum := sum + i
[] i > j -> sum := sum + j
[] j > i -> sum := sum + i
fi
```

如果 i=0 且 j>i，那么这个结构是选择第一个还是第三个赋值表达式就是不确定的；如果 i 和 j 相等且不为 0，就会报告运行时错误，因为没有一个条件是 true。

这个结构允许程序员简明扼要地说明，执行顺序有时是不重要的。例如，为了找到两个数中的最大值，可以采用如下代码：

```
if x >= y -> max := x
[] y >= x -> max := y
fi
```

这段代码计算出想要的结果，而不必过于严谨地指出其解决方法。具体而言，当 x 和 y 相等时，哪个赋值给 max 就不重要了。这是一种由语义不确定的语句提供的抽象化形式。

现在考虑用传统的编程语言选择器来编码这个过程：

```
if (x >= y)
  max = x;
else
  max = y;
```

这也可以写成：

```
if (x > y)
  max = x;
else
  max = y;
```

这两个结构实质上没有区别。第一个结构在 x、y 相等时，将 x 赋给 max；第二个结构在同样的情况下，将 y 赋给 max。这两个结构间的选择使代码的正式分析和程序正确性的验证复杂化，这也是 Dijkstra 开发防护命令的原因之一。

由 Dijkstra 提出的循环结构形式如下：

```
do  <布尔表达式>  ->  <语句>
[]  <布尔表达式>  ->  <语句>
[]  ...
[]  <布尔表达式>  ->  <语句>
od
```

这个结构的语义是：所有的布尔表达式在每次迭代中都要计算。如果有多个布尔表达

式的值是 true，则选择执行哪个相关语句就是不确定的(也许是随机的)，之后再次计算表达式。当所有的表达式同时为 false 时，循环终止。

考虑下面的问题：给定四个整型变量 q1、q2、q3 和 q4，重新排列这四个变量，使 q1≤q2≤q3≤q4。如果不使用防护命令，一种简单的解决方案是把这四个变量的值放到一个数组中，接着为这个数组排序，然后把值从数组中取出，并返回给标量变量 q1、q2、q3 和 q4。虽然这个方案不是很困难，但它需要大量代码，尤其是需要包含排序过程时。

现在，考虑下面的代码，其使用防护命令，以更简洁明了的方式解决了这个问题[7]：

```
do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
[] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;
[] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
od
```

Dijkstra 的防护命令控制结构很有趣，部分原因是它们揭示了语句的语法和语义与程序验证是如何互相影响的。使用 goto 语句时，程序验证是几乎不可能的。如果只使用来逻辑循环和选择结构，或者只使用了防护命令，验证就会大大简化。说明防护命令的公理化语义非常方便(Gries，1981)。然而，显然，用防护命令处理比较确定的执行过程时，复杂度会显著增加。

8.6　结论

本章描述并讨论了各种语句级别的控制结构。下面按顺序简单地评价一下。

首先，理论成果表明，只有顺序、选择和前测试逻辑循环结构才绝对需要计算表达式(Böhm 和 Jacopini，1966)。希望禁用无条件分支语句的人使用了这个成果。当然，即使没有这个理论成果，goto 语句也已经有足够多的实际问题来证明它的不足。需要 goto 语句的一个合法场合是提前退出循环，这可以用 break 语句这样限制严格的分支语句来满足。

Böhm 和 Jacopini 的研究成果的一个明显的误用是，除了选择结构和前测试逻辑循环之外，反对在任何其他控制结构中包含 goto 语句。没有一种广泛使用的语言采取了这一措施，而且，由于 goto 语句对可读性和可写性的负面影响，将来也不会有哪种语言会这样做。只用选择结构和前测试逻辑循环写出来的程序，一般在结构上不自然、更复杂、更难读写。例如，C#的多重选择结构对 C#的可写性没有明显的负面影响，而是起到了很好的促进作用。很多语言(例如 Ada)的循环计数结构也是一个例子，特别是当语句很简单时。

许多其他控制结构是否应包含到语言中并不是很清楚(Ledgard 和 Marcotty，1975)。这个问题在很大程度上成为语言的规模是否必须最小化这个基本问题。Wirth(1975)和 Hoare (1973)都强烈地赞同语言设计的简单化。对于控制结构，简单化意味着只有几个控制结构应包含在语言中，它们都应该很简单。

---

7. 这些代码与 Dijkstra(1975)的稍有不同。

已开发的各种语句级别的控制结构体现出语言设计者观念的差异。在开发、讨论、评价之后，哪些控制结构应包含在语言中仍然没有一致的意见。当然，绝大部分当代的语言有相似的控制语句，但它们的语法和语义仍有细节上的差异。此外，在语言是否应该包含 goto 语句的问题上仍然存在争议，C++和 C#有，但是 Java 和 Ruby 没有。

小 结

控制语句分为几类：选择、多重选择、迭代和无条件分支。

在基于 C 的语言中，switch 语句是多重选择语句的代表。C#禁止从一个选中的代码段隐式地继续执行下一个可选代码段，消除了以前版本的可靠性问题。

人们为高级语言开发出了大量不同的循环语句。Ada 中 for 语句的复杂程度很低。它完美地实现了最常用的计数循环形式。C 的 for 语句是最灵活的迭代结构，但这种灵活性带来了一些可靠性问题。

大多数语言都为循环提供了退出语句，而退出循环原来是 goto 语句最常见的用途之一。

基于数据的迭代器是处理数据结构的循环结构，比如链表、散列表和树状结构。在基于 C 的语言中，for 语句允许用户为自定义的数据创建迭代器。Perl 和 C#语言的 foreach 语句就是处理标准数据结构的预定义迭代器。在当今面向对象的语言中，用于集合的迭代器用标准接口来指定，而标准接口是由集合的设计者实现的。

Ruby 中的迭代器是一种发送给各种对象的特殊形式的方法。该语言预定义了常用的迭代器，还允许用户自定义迭代器。

无条件分支 goto 语句是大多数命令式语言的一个组成部分。它的问题也得到了广泛的讨论与争论。目前达成的共识是：它应该保留在大多数语言中，但应通过编程规则最小化它的危险性。

Dijkstra 的防护命令是具有积极理论特色的另一种控制结构。虽然它不是语言的控制结构，但其部分语义出现在 CSP 和 Ada 的并发机制和 Haskell 的功能定义中。

复 习 题

1. 控制结构的定义是什么？
2. Böhm 和 Jocopini 证明了流程图的什么方面？
3. 块的定义是什么？
4. 对于所有的选择和迭代控制语句，其设计问题是什么？
5. 选择结构的设计问题是什么？
6. Python 设计的复合语句有何不寻常？
7. F#选择器在什么情况下有 else 子句？
8. 双路选择器的嵌套问题有什么常见的解决办法？
9. 多重选择语句的设计问题是什么？
10. 在多重选择结构的一次执行过程中，当决定是否执行多个可选代码段时，应该在哪两种语言特性中权衡？

11. C 的多重选择语句有什么特殊之处？
12. C 的 switch 语句来源于以前的哪种语言？
13. 解释一下 C#的 switch 语句为什么比 C 的 switch 语句更安全。
14. 对于所有的迭代控制语句，其设计问题是什么？
15. 计数控制循环语句的设计问题是什么？
16. 什么是前测试循环语句？什么是后测试循环语句？
17. C++和 Java 的 for 语句有什么区别？
18. C 中的 for 语句在哪些方面比在其他许多语言更灵活？
19. Python 中 range 函数的作用是什么？
20. 现在的哪些语言不包含 goto 语句？
21. 逻辑控制循环语句的设计问题是什么？
22. 开发用户自定义的循环控制语句的主要原因是什么？
23. 用户自定义的循环控制机制的设计问题是什么？
24. 与 C 的 break 语句相比，Java 中 break 语句的优势有哪些？
25. C++和 Java 的 break 语句有什么区别？
26. 什么是用户自定义的迭代控制机制？
27. 哪个 Scheme 函数实现了多重选择语句？
28. 函数式语言如何实现重复执行语句？
29. Ruby 如何实现迭代器？
30. 哪种语言预定义的迭代器可以显式调用，以遍历预定义的数据结构？
31. 哪种常见的编程语言从 Dijkstra 的防护命令中借鉴了一些设计？

习　题

1. 描述三种需要结合计数和逻辑循环结构的情况。
2. 学习在 Liskov 等(1981 年)中 CLU 迭代器的特点，并说出它的优点和缺点。
3. 比较 Ada 和 C#中控制语句的设置，确定哪个更优，并说明原因。
4. 在复合语句中使用特殊的结束保留字，有什么有利与不利因素？
5. 在 Python 中使用缩进来指定控制结构中的复合语句，说明其理由和优缺点。
6. 分析在控制语句中，使用与开始保留字倒序的结束保留字(如 ALGOL 68 的 case-esac 保留字)的可读性问题，例如两个相邻字符会打错的常见问题。
7. 请用 *Science Citation Index* 找到一篇引用 Knuth(1974)的文章。在阅读这篇文章和 Knuth 的论文之后，写一篇论文，简述关于 goto 问题的争论。
8. 在 Knuth(1974)关于 goto 问题的论文中，他提出了使用多个出口的循环结构。请阅读这篇论文，并写出这个结构的操作语义描述。
9. 在 Java 中，控制语句仅能使用布尔表达式(C++还允许使用算术表达式)，其支持和反对理由有哪些？

10. 在 Ada 中，case 结构的选择列表必须是完备集，这样在控制表达式中，肯定不存在未表述的值。在 C++中，未表述的值可以在运行期间由 default 选择器捕获。如果没有 default 选择器，未表述的值将使整个结构被忽略。这两种设计(Ada 和 C++)的优缺点是什么？
11. 请对比并解释 Java 和 Ada 中 for 语句的优缺点。
12. 请描述在 Python 的 for 语句中使用 else 子句更简便的一个编程示例。
13. 描述三个需要后测试循环的编程实例。
14. 推测控制结构能传送至 C 循环结构的原因。

编程练习

1. 用指定的语言重写下列循环结构的伪代码段：

```
  k = (j + 13) / 27
loop:
  if k > 10 then goto out
  k = k + 1
  i = 3 * k - 1
  goto loop
out: ...
```

a. Fortran 95

b. Ada

c. C、C++、Java、或 C#

d. Python

e. Ruby

假定所有变量都是整型。讨论这段代码使用哪种语言有最好的可写性、可读性或者在两者之间取得最佳平衡。

2. 重新做第 1 题，但是这次，所有变量及常数均为浮点数，并把

```
k=k+1
```

改为

```
k=k+1.2
```

3. 用指定的语言，通过多重选择结构重写以下代码

```
if ((k == 1) || (k == 2)) j = 2 * k - 1
if ((k == 3) || (k == 5)) j = 3 * k + 1
if (k == 4) j = 4 * k - 1
if ((k == 6) || (k == 7) || (k == 8)) j = k - 2
```

a. Fortran 95(可能需要查阅一些资料)

b. Ada

c. C、C++、Java 或者 C#

d. Python

e. Ruby

假定所有变量都是整型。讨论针对这段代码使用这些语言相对而言各有什么优点。

4. 观察以下 C 程序代码，重写这段代码，但不使用 goto 或者 break 语句。

```
j = -3;
for (i = 0; i < 3; i++) {
 switch (j + 2) {
   case 3:
   case 2: j--; break;
   case 0: j += 2; break;
   default: j = 0;
  }
 if (j > 0) break;
  j = 3 - i
}
```

5. Rubin(1987)在一封致 CACM 编辑的信中，使用如下代码段证明，一些含有 goto 语句的程序段的可读性要优于不含 goto 语句的等价程序段。这段代码在一个 n*n 整数矩阵 x 中找出第一个只包含 0 的行。

```
for (i = 1; i <= n; i++) {
  for (j = 1; j <= n; j++)
    if (x[i][j] != 0)
     goto reject;
  println ('First all-zero row is:', i);
  break;
reject:
 }
```

选用 C、C++、Java、C#或 Ada 中的一种语言重写这段程序，但比较你的代码与示例代码的可读性。

6. 考虑如下编程问题：给定三个整型变量 first、second 和 third，把三者的最大值、中值和最小值分别赋给 max、mid 和 min 三个变量，但不能使用数组、用户定义的子程序以及预定义的子程序。写出问题的两个解决方案：一个使用嵌套选择结构，一个不使用。比较两个答案的复杂度与可靠性。

7. 用 Ada 语言写出下述 Java 程序的 for 结构。

```
int i, j, n = 100;
for (i = 0, j = 17; i < n; i++, j--)
  sum += i * j + 3;
```

8. 利用 C 语言中的 if 与 goto 语句改写练习 4 中的 C 程序。

9. 利用 Java 语言改写练习 4 中的 C 程序，但不使用 switch 结构。

10. 将下面对 Scheme 中 COND 函数的调用改写为 C 程序，将得到的值设置给 y：

```
(COND
 ((> x 10) x)
 ((< x 5) (* 2 x))
 ((= x 7) (+ x 10))
)
```

第9章

子 程 序

9.1 概述
9.2 子程序的基本原理
9.3 子程序的设计问题
9.4 局部引用环境
9.5 参数传递方式
9.6 子程序作为参数
9.7 间接调用子程序
9.8 重载子程序
9.9 泛型子程序
9.10 函数的设计问题
9.11 用户定义重载运算符
9.12 闭包
9.13 协同程序

子程序是程序的基本构成部分，因此，在编程语言设计领域中，子程序是最重要的概念之一。本章探究子程序的设计，内容包括参数传递方法、局部引用环境、子程序重载、泛型子程序，以及与子程序相关的别名和副作用问题。我们还会讨论间接调用子程序、闭包和协同程序。

第 10 章将讨论子程序的实现方法。

9.1 概述

编程语言含有两个基本的抽象过程——过程抽象和数据抽象。早期的高级编程语言只有过程抽象。过程抽象采用子程序的形式，在所有的编程语言中都是一个核心概念。不过，在 20 世纪 80 年代，人们开始认为数据抽象也同样重要。第 11 章将详细讨论数据抽象。

第一台可编程计算机建造于 19 世纪 40 年代的 Babbage 分析引擎，它可以复用位于程序不同位置上的指令卡片集合。在现代编程语言中，这样的指令集写作子程序。这种复用过程的好处是节约了内存空间与编码时间。这种复用过程也是一种抽象，因为子程序的计算细节被程序中一条调用子过程的语句取代了。不是在程序中解释某些计算如何完成，而是通过一条调用语句来描述(即子程序中的语句集合)，这样就有效地抽取出了各种细节。这样，通过在展现逻辑结构的同时隐藏底层细节，从而提高了程序的可读性。

面向对象语言的编程思想与本章讨论的子程序紧密相关。这种思想与子程序的主要区别是它们的调用方式，以及子程序与类和对象的联系。第 12 章将讨论面向对象语言的特殊性质。本章讨论面向对象语言和子程序的共同之处，包括参数和局部变量等。

9.2 子程序的基本原理

9.2.1 子程序的一般性质

本章讨论的所有子程序(不包括 9.13 节描述的协同程序)都有以下特性：

- 每个子程序都只有一个入口。
- 执行被调用的子程序时，调用程序单元暂停。这表示每次都只执行一个子程序。
- 子程序结束时，程序的控制权交还给调用程序单元。

对上述特性进行取舍，就产生了协同程序和并发单元(第 13 章)。

大多数子程序都有名字，但一些子程序是匿名的。9.12 节包含 C#中匿名子程序的例子。

9.2.2 子程序的基本定义

子程序定义描述了子程序的接口与其行为。子程序调用是调用子程序的显式要求。调用一个子程序，子程序就开始执行，在这个子程序尚未终止前，就称这个子程序是活动的。过程与函数是子程序的两种基本形式，它们将在 9.2.4 节中定义及讨论。

子程序头是子程序定义的第一部分，它有如下功能。首先，它指定了以下语法单元是某种子程序定义[1]。在包含多种子程序的语言中，子程序的类型通常由一个特殊字表示。其次，如果子程序不是匿名的，子程序头就提供了子程序名。第三，它可以指定一列参数。

观察这个子程序头的例子：

```
def adder (parameters):
```

这是 Python 子程序 adder 的头部。Ruby 子程序头也以 def 开头。JavaScript 子程序头以 function 开头。

在 C 语言中，函数 adder 的头如下：

```
void adder (parameters)
```

在这个子程序头中，保留字 void 表示子程序没有返回值。

子程序体定义了其操作。在基于 C 的语言(和其他一些语言，例如 JavaScript)中，子程序体用花括号界定。在 Ruby 中，用 end 语句结束子程序体。与复合语句一样，Python 函数体中的语句也必须缩进，函数体的结尾用一行未缩进的语句表示。

Python 函数与其他常见编程语言中的函数的主要区别是，它的 def 语句是可执行的。执行 def 语句时，它将其后的函数名分配给指定的函数体。只有执行了函数的 def 语句，才能调用该函数。观察下例：

```
if ...
    def fun(...):
        ...
else
    def fun(...):
        ...
```

如果执行这个选择结构的 then 子句，就可以调用其中的 fun 函数，而不是 else 子句中的 fun 函数。同样，如果执行 else 子句，就可以调用 else 中的 fun 函数，而不是 then 子句中的 fun 函数。

Ruby 的方法与其他编程语言中的子程序有几点有趣的不同。Ruby 方法通常定义在某个类中，也可以定义在类定义之外，这时它们是根对象 Object 的方法。这类方法可以不经过对象接收器而直接调用(实际上是默认指定了 self 接收器)，就像 C 和 C++中的函数一样。如果在当前类找不到某个方法，程序就会搜索其嵌套类，如果有必要，会一直搜索到 Object 对象。

尽管所有的 Lua 函数都可以用某个名称来定义，可它们实际上都是匿名的。例如下面两个对函数 cube 的定义没有任何区别。

```
function cube(x) return x * x * x end
```

---

1. 有些编程语言包括两种子程序：过程和函数。

```
cube = function (x) return x * x * x end
```

第一例使用了传统的语法，而第二例更精确地展示了函数的匿名性。

子程序的参数描述包括了其形参的个数、顺序和类型。子程序的协议包括了其参数类型和返回值的类型。如果某种语言的子程序有自己的类型，这些类型就由子程序的协议来规定。

子程序既可以有定义，也可以有声明。这种形式同时提供了 C 语言中的变量声明与定义，其中声明过程可以仅给出类型信息，而不定义变量。子程序声明仅提供了它的协议，但是不提供函数体。对于不允许循环引用的语言来说，定义与声明都是必要的。对于变量与子程序来说，编译器需要声明来进行静态类型检查。对于子程序，编译器还需要检查参数的类型。函数声明在 C 和 C++中很常见，称为原型。它们一般出现在头文件中。

除了 C 和 C++之外，大部分其他语言中的子程序都不需要声明，因为不需要在调用子程序之前定义它。

9.2.3 参数

子程序通常描述一个计算过程。非方法的子程序有两种途径访问它们处理的数据。第一种途径是直接访问非本地变量(不在子程序中定义，但在子程序中可见的变量)；第二种则是参数传递。通过参数传递的数据在子程序中可通过本地变量访问。参数传递比直接访问非本地变量更灵活。基本上，通过参数访问数据的子程序进行的是参数化计算。子程序可以通过其参数对接收到的任意数据执行计算(假定参数的类型与子程序所指定的一致)。如果子程序通过非本地变量访问数据，那么，在两次子程序调用之间，要对不同的数据执行计算，就必须为这些非本地变量分配不同的值。对非本地变量进行外部访问会降低程序的可靠性，因为对子程序可见的变量经常在不需要访问它们的场合中可见。这个问题参见第 5 章。

尽管方法也通过非本地引用或参数来访问外部数据，可它们处理的主要数据是调用方法的对象中的数据。但是，当方法访问非本地数据时，其可靠性问题就与非方法的子程序一样。并且，在面向对象的语言中，访问类变量的方法(直接与类而不是与对象相关联)是与非本地数据的概念相关联的，应该尽量避免。在这种情况以及 C 函数访问非本地变量的情况下，方法会产生副作用，即改变除其参数或本地数据之外的变量。这些改写使方法的语义更加复杂，并降低了其可靠性。

纯函数式编程语言，例如 Haskell，没有可变的数据，所以用这种语言编写的函数不能以任何方式改变内存——它们只是执行计算，返回得到的值(或函数，因为函数就是值)。

在某些情况下，把计算过程(而不是数据)当做参数传递给子程序是很方便的。在这种情况下，执行计算的子程序名可以用作参数。这种形式的参数将在 9.6 节讨论。数据参数在 9.5 节讨论。

在子程序头中定义的参数叫做形参。他们有时被视为虚变量，因为它们不是通常意义下的变量。在大多数情况下，它们只在调用子程序时绑定到存储空间上，这种绑定常常通

过程序的其他变量来完成。

子程序调用语句必须包含子程序名和一系列与形参对应的参数。这些参数称为实参[2]。它们必须与形参相区分，因为这两者通常在形式上有着不同的限制，当然，它们的用法也大不相同。

在几乎所有的编程语言中，实参与形参的对应关系——实参到形参的绑定——是由位置决定的。第一个实参绑定到第一个形参，以此类推。这种参数称为位置参数。在参数个数较少时，这种将实参与对应的形参关联起来的方式既安全又高效。

但是，当参数较多时，程序员可能会搞错实参的顺序。这类问题的一种解决方法是提供关键字参数，在调用时，把绑定到实参的形参名用实参来指定。关键字参数的好处是它们可以在实参列表中以任意顺序出现。Python 函数就可以使用这种方式来调用：

```
sumer(length = my_length,
      list = my_array,
      sum = my_sum)
```

sumer 函数在定义中包含 length、list 和 sum 三个形参。

关键字参数的缺点是，子程序的用户必须知道形参名。

除了关键字参数之外，Ada、Fortran 95+和 Python 还支持位置参数。关键字和位置参数可以在子程序调用中混用：

```
sumer(my_length,
      sum = my_sum,
      list = my_array)
```

这种方式的唯一限制是，当参数列表中出现关键字参数后，其后的参数必须带有关键字。这种限制是有必要的，因为在关键字参数出现后，形参的位置就不再符合之前的定义了。

在 Python、Ruby、C++、Fortran 95+、Ada 和 PHP 中，形参可以有默认值。如果没有把实参传递给子程序头部中的形参，这些形参就使用默认值。考虑下面的 Python 函数头：

```
def compute_pay(income, exemptions = 1, tax_rate)
```

在函数 compute_pay 的调用中，形参 exemptions 可以省略，此时，其值取 1。在 Python 的函数调用中，被省略的实参不需要保留逗号，因为这个逗号的唯一价值是指示下一个参数的位置。然而在这种情况下并不必要，因为在被省略的实参之后，所有的实参都必须加上关键字。例如下面的调用：

```
pay = compute_pay(20000.0, tax_rate = 0.15)
```

C++不支持关键字参数，所以默认参数的规则是有区别的：默认参数必须放在最后，因为参数间是有前后位置关系的。一旦在调用中省略了一个默认参数，剩余的形参就必须

2. 一些作者把实参称为因变量，形参称为自变量。

取默认值。C++函数 compute_pay 的头部可以写为：

```
float compute_pay(float income, float tax_rate,
                  int exemptions = 1)
```

请注意，形参已重新排列：使用默认值的形参出现在最后。下面调用了 compute_pay 函数：

```
pay = compute_pay(20000.0, 0.15);
```

在形参不使用默认值的大多数语言中，调用子程序时，实参数目必须匹配子程序头定义的形参数目。但是，在 C、C++、Perl、JavaScript 和 Lua 中，没有这种要求。当实参的数目少于函数定义的形参数目时，程序员有责任确保参数在位置上的相互对应，子程序的执行是正常有效的。

尽管这种允许参数数量可变的设计显然更容易出错，但有时十分方便。例如，C 语言中的 printf 函数能够打印任意数量的项(数据值和/或字面量)。

C#允许方法接受数目可变但类型相同的参数。方法通过 params 修饰符来规定其形参。调用方可以传入一个数组或一组表达式，它们的值由编译器放在一个数组中，并提供给被调用的方法。例如下面的方法：

```
public void DisplayList(params int[] list) {
    foreach (int next in list) {
       Console.WriteLine("Next value {0}", next);
    }
}
```

如果 DisplayList 是为 Myclass 类定义的方法，且有如下声明：

```
Myclass myObject = new Myclass;
int[] myList = new int[6] {2, 4, 6, 8, 10, 12};
```

那么 DisplayList 就能以下面两种方式调用：

```
myObject.DisplayList(myList);
myObject.DisplayList(2, 4, 3 * x - 1, 17);
```

Ruby 语言支持一种复杂但非常灵活的实参结构。最开始的参数是表达式，其值对象传递给对应的形参。这些参数可以后跟一串“键=>值”对，它们放在一个匿名的散列中，对这个散列的引用传递给下一个形参。这用作关键字参数的一种替代方法(Ruby 语言不支持关键字参数)。散列项可以后跟一个以星号(*)开头的参数。这个参数称为“数组形参”。当调用方法时，数组形参设置为引用新的 Array 对象。剩余的实参就都赋予新 Array 对象的元素。如果数组形参所对应的实参是一个数组，则这个实参也要在开头加上星号(*)，且它必须是最后一个实参[3]。所以，Ruby 语言支持可变数量的参数，其方式和 C#语言很相似。

3. 并不完全正确，因为数组形参后能接一个以&开始的方法或函数。

由于Ruby数组可以存储不同类型的值，传递给数组的实参类型就不需要相同。

下面例子中的框架函数定义和调用，展示了Ruby的参数结构：

```
list = [2, 4, 6, 8]
def tester(p1, p2, p3, *p4)
  ...
end
...
tester('first', mon => 72, tue => 68, wed => 59, *list)
```

在tester内部，它的形参值是：

```
p1 is 'first'
p2 is {mon => 72, tue => 68, wed => 59}
p3 is 2
p4 is [4, 6, 8]
```

Python语言支持的参数与Ruby相似。

Lua使用一种简单的机制来支持可变数目的参数，这些参数用省略号(...)表示。省略号可以看作一个数组，或者看作可分配给一组变量的一组值。请看下面的两个函数例子：

```
function multiply (...)
  local product = 1
  for i, next in ipairs{...} do
    product = product * next
  end
  return sum
end
```

ipairs是数组的迭代器(它返回数组中元素的索引和值，一次一个元素)。{...}是一个实参值的数组。

```
function DoIt (...)
  local a, b, c = ...
  ...
end
```

假设DoIt以如下方式调用：

```
doit(4, 7, 3)
```

在这个例子中，a、b、c在函数中分别初始化为4、7、3。

省略号参数不一定是唯一的参数，它可以出现在命名的形参列表的最后。

9.2.4 过程与函数

有两种不同类别的子程序——过程和函数，但它们都是扩展语言的方式。所有的子程序都是定义参数化计算的语句集合。函数有返回值，而过程没有。在大多数没有把过程包含为一种独立形式的子程序的语言中，函数可以定义为没有返回值，它们可用作过程。过

程的计算是由单一的调用语句执行的。实际上，过程定义了新的语句。例如，如果某种语言没有排序命令，用户就可以建立一个过程，对数组进行排序，并调用这个过程来代替不存在的排序命令。在 Ada 语言里，过程就称为“过程”。而在 Fortran 语言里，它们称作子例程。大多数其他语言不支持过程。

过程能够通过两种方法在调用程序单元中产生结果。首先，如果存在一些变量，它们不是形参，但仍在过程和调用程序单元里可见，过程就可以改变它们。其次，如果过程有形参，可以把数据传给调用方，那些形参就可以改变。

函数与过程在结构上十分类似，但在语义上模拟了数学函数。如果函数是一个可靠的模型，就不会产生副作用，即它不会修改其参数或定义在函数外的变量。这种纯粹的函数返回一个值——即我们需要的结果。事实上，大多数编程语言中的函数都有副作用。

调用函数时，函数名和需要的实参出现在表达式中。执行函数所产生的结果会返回给调用代码，并替换掉调用它的代码。例如，表达式 f(x)的值就是用 x 调用函数 f 时得出的值。对于不产生副作用的函数，返回值是它对外界唯一的影响。

函数定义了新的用户自定义操作符。例如，如果某种语言没有幂操作符，就可以编写一个函数，返回以一个参数为底、另一个参数为指数的幂。它在 C++中的声明如下：

```
float power(float base, float exp)
```

它可以这样调用：

```
result = 3.4 * power(10.0, x)
```

标准 C++库已经提供了一个类似的函数 pow。相比之下，在 Perl 语言中，幂操作符是内置的：

```
result = 3.4 * 10.0 ** x
```

在一些编程语言中，用户可以通过为操作符定义新的函数来重载操作符。用户自定义的重载操作符将在第 9.11 节中详细讨论。

9.3 子程序的设计问题

子程序在编程语言中是一个复杂的结构，因而在设计中会出现很多问题。其中一个明显的问题是怎样选择参数的传递方式。在各种语言中使用很多不同的方法，反映出在这个问题上，人们有很多种观点。另一个与此紧密相关的问题是，实参是否应当根据对应的形参类型进行类型检查。

子程序的局部环境在某种程度上支配着子程序的类型。这里最重要的问题是，局部变量是动态分配空间还是静态分配空间的。

其次，子程序定义是否可以嵌套也是一个问题。另一个问题是子程序名是否可以传递为参数。如果子程序名可以传递为参数，且语言允许子程序嵌套，就会出现另一个问题：

对传递为参数的子程序，应当怎样正确地引用它。

最后，还有子程序能否重载或泛型的问题。重载子程序是指，在同一个引用环境中，一个子程序与另一个子程序具有相同的名称。泛型子程序是指，能够在不同的调用中，对不同类型的数据进行操作的子程序。闭包是嵌套的子程序及其引用环境，它们允许子程序从程序的任何位置调用。

下面是子程序设计问题的汇总。更多与函数相关的问题在9.10节中详细讨论。

- 局部变量是动态分配空间还是静态分配空间的？
- 一个子程序的定义可以出现在另一个子程序的定义中吗？
- 应当选择哪个或哪些参数传递方式呢？
- 实参是否应当根据对应的形参类型进行类型检查？
- 如果子程序可以传递为参数，并且可以嵌套，那么应当怎样引用传递过来的子程序？
- 子程序能重载吗？
- 子程序可以泛型化吗？
- 如果语言允许嵌套的子程序，那么会支持闭包吗？

这些问题和设计范例将在接下来的各节中讨论。

9.4 局部引用环境

本节将讨论在子程序内部定义的变量相关问题，也简要涉及子程序嵌套定义的问题。

9.4.1 局部变量

子程序可以定义自己的变量，从而确定局部引用环境。在子程序内部定义的变量称为局部变量，因为它们的作用域通常就是所在的子程序体。

按照第5章的术语，局部变量既可以是静态变量，也可以是栈动态变量。如果局部变量是栈动态变量，它们就在子程序开始运行时，与存储空间绑定，在该子程序终止运行时，解除该绑定。栈动态的局部变量有几个优点，主要的一点是它们给子程序提供了灵活性。递归子程序有栈动态局部变量是很重要的。栈动态局部变量的另一个优点是，在活动的子程序中分配给局部变量的存储空间，可以与所有未活动的子程序中的局部变量共享。这仅在计算机内存较小时，才是一个较大的优势。

栈动态局部变量的主要劣势如下：首先，在每个子程序调用中，分配、初始化(在需要时)以及释放变量需要时间。其次，栈动态局部变量的访问必须是间接的，而静态变量的访问可以是直接的[4]。这种间接访问是必须的，因为某个局部变量在栈中的位置只能在执行期间确定(见第10章)。最后，如果所有的局部变量都是栈动态变量，子程序就不会是历史敏

4. 在一些实现方案中，静态变量也是间接访问的，因此没有这个缺点。

感的，也就是，它们无法在几次调用之间保留局部变量的数据值。有时能写出历史敏感的子程序是很方便的。需要历史灵敏的子程序的一个常见例子是生成伪随机数。每次调用该子程序计算一个伪随机数，都需要用到上一个计算出的伪随机数。因此，必须将上一个伪随机数存放在静态局部变量中。协同程序和迭代循环结构(参见第 8 章)所用的子程序也是需要用到历史敏感的子程序的例子。

与栈动态局部变量相比，静态局部变量的主要优势是它们稍微高效一些——它们不需要在运行期间分配与释放存储空间。例如，如果直接访问静态局部变量，这些访问显然更高效。并且，理所当然地，它们允许子程序是历史敏感的。静态局部变量的最大缺点是它们不支持递归。同时，它们的存储空间不能和其他不活动的子程序中的局部变量共享。

在大多数当代语言中，子程序中的局部变量默认为栈动态变量。在 C 和 C++函数中，除非特别声明为静态，否则局部变量都是栈动态变量。例如，在下面的 C(或 C++)函数中，变量 sum 是静态的，count 是栈动态变量。

```
int adder(int list[], int listlen) {
  static int sum = 0;
  int count;
  for (count = 0; count < listlen; count ++)
    sum += list [count];
  return sum;
}
```

C++、Java 和 C#的方法只有栈动态局部变量。

在 Python 中，方法定义中唯一用到的声明是全局变量的声明。方法中任何声明为全局的变量都必须在方法外部定义。方法外定义的变量可以不声明为全局变量，就在方法中引用，但是这个变量不能在方法中赋值。如果全局变量在某个方法里赋值了，它就隐含地声明为局部变量，这个赋值不会影响全局变量。Python 方法中的所有局部变量都是栈动态变量。

在 Lua 中，只声明作用域有限制的变量。任何语句块，包括函数体，都可以用 local 声明局部变量，例如：

```
local sum
```

Lua 中所有未声明的变量都是全局变量。根据 Ierusalimschy(2006)，Lua 中对局部变量的访问比对全局变量的访问更快。

9.4.2 嵌套子程序

嵌套子程序概念源于 Algol 60。其目标是创建逻辑层次结构和作用域层次结构。如果某个子程序只在另一个子程序中用到，为什么不把它放在那个子程序中，让程序的其他部分看不到它？因为静态作用域通常只在支持嵌套子程序的语言中使用，这也提供了一个高度结构化的方法，在外层子程序中访问非局部变量。第 5 章讨论了由此引出的问题。很长一段时间里，只有支持嵌套子程序的语言直接从 Algol 60 衍生而来，比如 Algol 68、Pascal 和 Ada。很多其他语言，包括 C 的所有直接衍生语言，并不支持子程序的嵌套。近来，一

些新语言也开始支持它了。这些语言有 JavaScript、Python、Ruby 和 Lua 等。另外，大多数函数式编程语句也允许嵌套子程序。

9.5 参数传递方式

参数传递是指参数传入传出被调用的子程序的方法。首先介绍参数传递方法的不同语义模型。然后讨论由语言设计者给这些模型设计的不同实现模型。接下来调查几种语言的设计选择，讨论实现模型所用的实际方法。最后，思考语言设计者选择这些方法时考虑的问题。

9.5.1 参数传递的语义模型

形参有三个完全不同的语义模型：(1)它们接收来自对应实参的数据；(2)它们可以传递数据给实参；(3)两者皆可。这三种语义模式分别称为输入型、输出型以及输入输出型。例如，一个子程序取 int 类型的两个数组作为参数，分别为 list1 和 list2。这个子程序必须将 list1 与 list2 相加，将结果返回为 list2 的新版本。另外，该子程序还必须使用这两个已给的数组创建出一个新的数组，并返回它。对于这个子程序，list1 应该为输入型的，因为子程序不会改变它。1ist2 必须是输入输出型的，因为子程序需要这个数组的初始值，还必须给它返回新值。第三个数组就应该是输出型的，因为它没有初始值，还必须将计算的新值返回给调用程序。

关于数据在参数传递过程中怎样传输，有两种概念模型：复制实际的值(到调用程序，到被调用程序，或者到这两者)，或者通过访问途径来传输。最常见的访问途径是一个简单的指针或引用。图 9-1 描述了复制值时参数传递的三种语义模型。

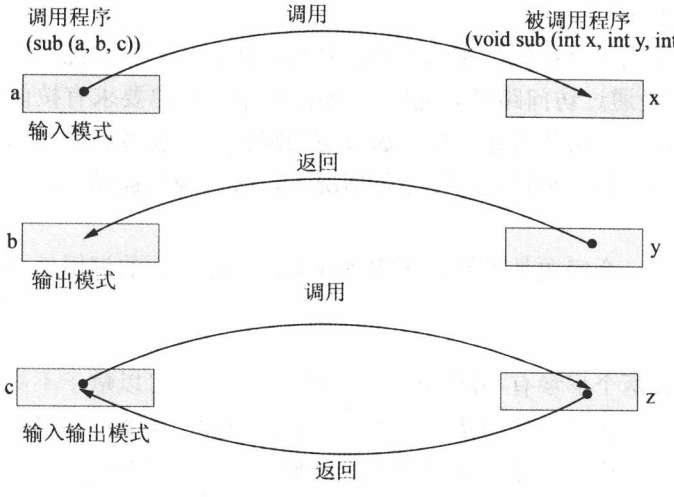


图 9-1 使用物理移动方式时，传递参数的三种语义模型

9.5.2 参数传递的实现模型

语言设计者开发了各种模型，来指导如何实现三种基本的参数传递模式。下面的几节将讨论其中的几种模式，评价它们的优缺点。

9.5.2.1 按值传递

按值传递参数时，实参的值用来初始化对应的形参，然后这个实参就用作子程序中的局部变量，来实现输入型语义。

按值传递一般通过复制来实现，因为通过这种途径，数据的访问常常更有效。其实现方式是：在调用程序中，通过访问路径传递实参的值，但这需要该值位于一个有写保护的存储单元中(只能读取的存储单元)。强制写保护并不总是很容易。例如，假定子程序接收传递过来的参数，再将其传递给另一个子程序。这是使用复制传递的另一个原因。如 9.5.4 节所述，C++提供了一种方便、有效的方法，给按值传递的参数指定写保护，这些按值传递的参数是通过访问路径传递的。

按值传递的优势在于，对于标量来说，无论是链接成本还是访问时间，它都是很快的。

如果使用复制操作，按值传递方法的主要缺点是，在被调用子程序的内部，或者调用子程序和被调用子程序外部的一些区域里，必须为形参分配额外的存储空间。此外，实参必须复制到对应形参的存储区域内。如果该参数太大，例如包含许多元素的数组，则存储空间和拷贝操作的开销会很大。

9.5.2.2 按结果传递

按结果传递是输出型参数的一种实现模型。参数按结果传递时，不会把值传递给子程序。对应的形参用作局部变量，但是仅在控制权传回调用程序之前，它的值才传递给调用程序的实参，很明显，实参必须是一个变量。(如果它是字面量或表达式，调用程序该如何引用计算出来的结果呢？)

按结果传递的方式具有按值传递的优势和缺点，还有另一些缺点。如果值像通常那样通过复制来返回(不是通过访问路径来返回)，则按结果传递也要求有按值传递所需的额外存储空间和复制操作。与按值传递一样，通过访问路径来实现按结果传递比较困难，所以通常会通过复制来实现按结果传递。在这种情况下，必须确保被调用的子程序不使用实参的初始值。

按结果传递的另一个问题是可能存在实参冲突，例如下面的调用就会产生一个冲突：

```
sub(p1, p1)
```

在 sub 中，假设两个形参有不同的名字，它们两个显然可以赋予不同的值。于是，两个形参中后一个复制给相应实参的就是调用程序中 p1 的值。因此，实参的复制顺序决定了它们的值。例如以下 C#方法，它在形参上用 out 修饰符指定了按结果传递方式[5]。

---

5. out 修饰符也必须在相应的实参上指定。

```
void Fixer(out int x, out int y) {
   x = 17;
   y = 35;
}
...
f.Fixer(out a, out a);
```

如果在 Fixer 执行到最后，形参 x 先赋值给了对应的实参，那么调用程序中实参 a 的值是 35。如果 y 首先赋值，那么调用程序中实参 a 的值是 17。

因为赋值次序在不同的语言实现中是不同的，所以不同的实现可能产生不同的结果。

使用其他参数传递方法时，用两个相同的实参调用函数也可能导致不同的问题，参见 9.5.2.4 节。

按结果传递可能导致的另一个问题是，实现者能够选择在两个不同的时间来获得实参的地址，即在调用时与返回时。例如以下 C#方法和下面的代码：

```
void DoIt(out int x, int index){
   x = 17;
   index = 42;
}
...
sub = 21;
f.DoIt(list[sub], sub);
```

list[sub]的地址在调用过程中会改变。实现者必须选择是在调用时还是在返回时，将这个参数绑定到地址上。如果在开始执行方法时计算地址，就把 17 返回给 list[21]；如果在返回前计算地址，就把 17 返回给 list[42]。于是，在子程序的开头计算输出参数的地址，和在退出子程序前计算它这两种实现方案之间，程序是不可移植的。避免这个问题的一个显而易见的方式是，语言设计者指定何时计算用于返回参数值的地址。

9.5.2.3 按值-结果传递

按值-结果传递是输入输出参数的一种实现模型，它会复制实参的值。它实际上是按值传递和按结果传递的一种组合。实参的值用来初始化相应的形参，再将形参用作本地变量。实际上，按值-结果传递的形参必须有与被调用的子程序相关的本地存储空间。当子程序调用结束，形参的值就传回给实参。

按值-结果传递有时称为按复制传递，因为在子程序调用之初，实参复制给形参，调用结束后，形参又复制回实参。

与按值传递和按结果传递一样，按值-结果传递，也需要存储参数的更多空间，以及复制操作的时间。它与按结果传递一样，也有实参的调用顺序问题。

按值-结果传递的优点同按引用传递的优点很类似，参见 9.5.2.4 节。

9.5.2.4 按引用传递

按引用传递是输入输出参数的第二种实现模型。按引用传递不像按值-结果传递那样来

回复制数据值，而是通过访问路径获取数据的地址，并传递给被调用的子程序。它提供了访问存储实参的单元的路径。因此，被调用的子程序可以在调用程序单元中访问实参。实际上，实参由被调用的子程序共用。

按引用传递的优点在于传递过程的时间、空间效率都很高，不需要额外的空间或者复制操作。

然而，按引用传递也有一些不足之处。首先，对形参的访问比按值传递的参数慢，因为引用要求额外的间接寻址操作[6]。其次，如果仅需要数值单向传递给被调用的子程序，实参就可能出现无意的、错误的改动。

按引用传递的另一个不足是可以创建别名。这是不可避免的，因为按引用传递允许被调用的子程序使用访问途径，从而可以访问非局部变量。在其他情况下，也会出现这种别名问题，降低了可读性和可靠性，程序的验证也更困难。

按引用传递参数有几种创建别名的方式。首先，实参之间的冲突可以创建别名。下面的 C++函数通过引用来传递两个参数(参见 9.5.3 节)：

```
void fun(int &first, int &second)
```

如果调用 fun 函数时，同一个变量传递了两次，如下：

```
fun(total, total)
```

则 fun 中的 first 和 second 就是别名。

第二，在数组元素之间的冲突也可以创建别名。例如用两个数组元素调用 fun 函数，这两个数组元素用变量下标指定，如下：

```
fun(list[i], list[j])
```

如果这两个参数是按引用传递的，且 i 恰好等于 j，那么 first 和 second 就是别名。

第三，如果子程序的两个形参分别是数组的一个元素和整个数组，且它们都按引用来传递，则下面的调用：

```
Fun1(list[i], list)
```

就可能在 fun1 中创建别名，因为 fun1 能够通过第二个参数访问 list 的所有元素，而通过第一个参数访问一个数组元素。

按引用传递参数还有一种创建别名的方式是由形参和可见的非局部变量之间的冲突造成的。比如下面的 C 代码：

```
int * global;
void main() {
   ...
   sub(global);
   ...
```

6. 参见 9.5.3 节。

```
}
void sub(int * param) {
    ...
}
```

在 sub 中，param 和 global 是别名。

如果用按值-结果传递代替按引用传递，就可以避免所有这些可能创建别名的情形。但是，即使没有别名问题，有时也会出现其他的问题，参见 9.5.2.3 节。

9.5.2.5 按名字传递

按名字传递是输入输出参数的一种传递方法，它不对应单个的实现模型。实际上，参数按名字传递时，实参会替换子程序里所有相应的形参。这种方法和前面讨论的其他方法大不相同。在以前讨论的方法中，形参在调用子程序时绑定到实参的值或地址上。按名字传递的形参在调用子程序时绑定到一个访问方法上，而到值或地址的实际绑定却推迟到引用形参或给形参赋值的时候。实现按名字传递参数需要将一个子程序传递到被调用的子程序来计算形参的地址或值。还必须传送被传递的子程序的引用环境。这个子程序/引用环境是一个闭包(参见 9.12 节)[7]。按名字传递的参数实现起来很复杂，也较低效。它们还会使程序变得非常复杂，降低了可读性和可靠性。

因为按名字传递不是任何广泛使用的语言中的一部分，所以这里不再讨论。然而，汇编语言的宏指令在编译时用到了它，C++、Java 5.0 和 C# 2005 中泛型子程序的泛型参数也用到了它，参见 9.9 节。

9.5.3 参数传递方法的实现

现在讨论各种参数传递的实现模型是如何实现的。

在大多数当代语言里，参数的交流通过运行时栈进行。运行时栈由运行时系统来初始化和维护，运行时系统管理着程序的执行。运行时栈广泛用于子程序的连接和参数的传递，参见第 10 章。在接下来的讨论中，假定运行时栈用于所有的参数传递过程。

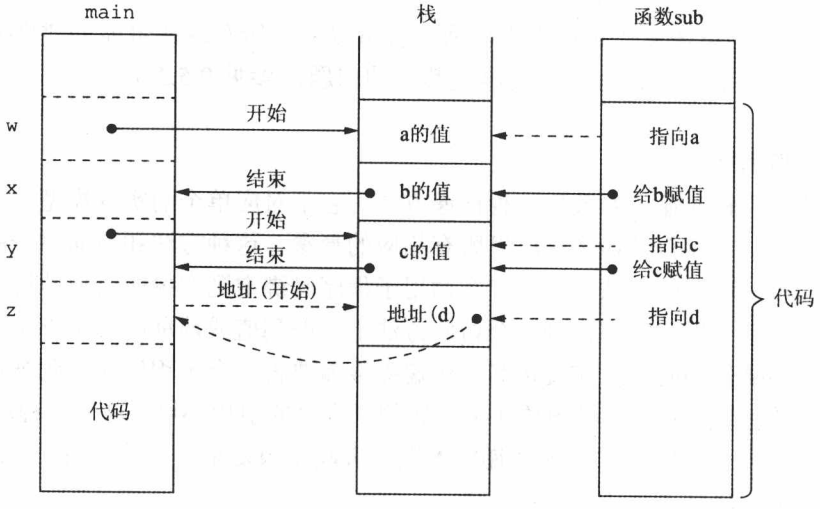
按值传递的参数把它们的值复制到栈中的存储位置。于是，这些栈中的存储位置就用作对应形参的存储空间。按结果传递的实现方式与按值传递正好相反。在按结果传递中，赋予实参的值放在栈中的存储位置，在被调用的子程序执行完毕时，调用程序单元就可以在那里检索到这些值。按值-结果传递的参数可以直接按照它的语义，实现为按值传递和按结果传递的结合体。这种参数在栈中的存储位置由调用程序初始化，然后在被调用的子程序中用作局部变量。

按引用传递的参数是最容易实现的。无论实参的类型是什么，都只需要将它的地址放在栈里。对于字面量，它的地址放进栈里。对于表达式，在控制权转移到被调用的子程序之前，编译器必须构建计算表达式的代码，然后把包含求值结果的存储单元地址放进栈里。编译器必须防止被调用的子程序改变常量或表达式参数的值。

---

7. 这些闭包最初(在 ALGOL 60 中)称为“形实转换程序”。闭包在 9.12 节讨论。

为了在被调用的子程序中访问形参，需要在地址的栈存储位置中进行间接寻址。按值传递、按结果传递、按值-结果传递和按引用传递使用运行时栈的实现方式见图 9-2。在 main 中调用子程序 sub(w, x, y, z)，其中 w 按值传递，x 按结果传递，y 按值-结果传递，z 按引用传递。



函数头：**void** sub (**int** a, **int** b, **int** c, **int** d)
main中的函数调用：sub (w, x, y, z)
w按值传递，x按结果传递，y按值-结果传递，z按引用传递

图 9-2 常见参数传递方法的一种可能的栈实现

9.5.4 常见语言的参数传递方法

C 使用按值传递。按引用传递(输入输出型)的语义是通过将指针用作参数来实现的。指针的值提供给被调用的函数，且不复制任何内容。然而，因为给数据的调用程序传递的是一个访问路径，所以被调用的函数可以改变调用程序的数据。C 使用按值传递的这种方式来源于 ALGOL 68。在 C 和 C++中，形参可以定义为指向常量的指针类型，而相应的实参不需要是常量，因为在这种情况中，它们会强制转换为常量。这就允许指针参数提供了按引用传递的高效性，和按值传递的单向语义。在被调用的函数中隐式指定了这些参数的写保护。

C++包含一种特殊的指针类型，称为引用类型，参见第 6 章，它常用于参数。引用参数在函数或方法中隐式地解除引用，而它们的语义是按引用传递。C++也允许把引用参数定义为常量，例如：

```
void fun(const int &p1, int p2, int &p3) { ... }
```

其中 p1 是按引用传递，但不能在函数 fun 中改变，p2 是按值传递，p3 是按引用传递。在 fun 中，p1 和 p3 都不需要显式地解除引用。

常量参数和输入型参数并不完全相同。显然，常量参数实现为输入参数。然而，在除 Ada 外的所有常用命令式语言中，输入型参数可以在子程序中赋值，即使这些改动从来不

在相应的实参值中反映出来。常量参数永远不能赋值。

历史注释

ALGOL 60 引入了按名字传递方式，它也允许使用按值传递方式。按名字传递参数的方式实现起来很困难，所以 ALGOL 60 和后来流行的语言都没有采用它(除了 SIMULA 67)。

历史注释

ALGOL W(Wirth 和 Hoare，1966)引入了按值-结果传递参数的方式，替代了低效的按名字传递方式，并避免了按引用传递方式存在的问题。

与 C 和 C++一样，所有的 Java 参数都是按值传递的。然而，因为对象只能通过引用变量来访问，所以对象参数实际上是按引用传递。尽管传递为参数的对象引用不能在被调用的子程序中修改，但是如果存在一个可以进行这个修改的方法，被引用的对象就可以修改。因为引用变量不能直接指向标量变量，Java 也没有指针，所以标量不能在 Java 中按引用传递(但包含了标量的对象引用可以按引用传递)。

Ada 和 Fortran 95+允许程序员对每个形参指定输入、输出或输入输出模式。

C#默认的参数传递方法是按值传递。要指定按引用传递，需要在形参和相应的实参前都加上 ref。例如下面 C#方法的框架和调用：

```
void sumer(ref int oldSum, int newOne) { ... }
...
sumer(ref sum, newValue);
```

sumer 的第一个参数是按引用传递的，第二个是按值传递的。

C#也支持输出参数，它们是按引用传递的参数，不需要初始值。这些参数在形参表中用 out 修饰符指定。

PHP 的参数传递方式类似于 C#，但形参或实参可以指定按引用传递。指定按引用传递时，需要在参数前面加上&符号。

Perl 使用了一种原始的参数传递方式。所有的实参都隐式地存放在一个预定义的名为“@_”的数组中(可以存放一切类型)。子程序在这个数组中检索实参的值(或地址)。这个数组最特殊的一点就是它魔术般的性质：其元素实际上是实参的别名。因此，数组@_中的元素在被调用的子程序中修改时，假如存在对应的实参(实参的个数不一定和形参的个数相同)，且是一个变量，则对这个元素的修改就会反映在对应的实参上。

Python 和 Ruby 的参数传递方法是按赋值传递。因为所有的数据值都是对象，每个变量都是对对象的引用。在按赋值传递中，实参的值赋给形参。因此，按赋值传递实际上就是按引用传递，因为所有的实参值都是引用。但是，只有在某些情形中，按赋值传递才会产生按引用传递参数的语义。例如，很多对象实质上是不可改变的。在纯面向对象的语言中，用赋值语句改变变量的值：

```
x=x+1
```

就没有改变 x 所引用的对象。它只是提取 x 所引用的对象，将其增加 1，然后(使用 x+1 的值)创建一个新对象，将 x 改为引用这个新对象。所以，把标量对象的引用传给子程序时，被引用的对象不能在子程序中改变。因为这个引用是按值传递的，即使在子程序中改变了形参，调用程序中的实参也不会受到任何影响。

现在假设把一个数组引用传递为参数。如果给对应的形参赋予一个新的数组对象，对调用程序就没有影响。可是，如果形参用于给数组里的元素赋值，例如

```
list[3]=47
```

实参就受到了影响。所以，改变对形参的引用对调用程序没有影响，但改变传递为参数的数组中的元素，就会影响调用程序。

9.5.5 参数的类型检查

人们都认为，软件的可靠性要求检查实参的类型是否与相应形参的类型一致。没有这样的类型检查，小的输入错误就可能导致很难诊断的程序错误，因为编译器和运行库系统检测不出它们。例如，函数调用

```
result=sub1(1)
```

实参是一个整型常量。如果 sub1 的形参是浮点型的，不检查参数的类型，就检查不出错误。尽管整数类型的 1 和浮点类型的 1 有相同的值，但二者的表示却完全不同。sub1 需要浮点型的值，却传入一个整型的实参值时，它不会产生正确的结果。

早期的编程语言，如 Fortran 77 和原始版本的 C 语言，不需要检查参数的类型，但后来的语言大多需要。当然，最近一些语言如 Perl、JavaScript 和 PHP 也不需要。

对于参数类型的检查，C 和 C++需要一些特殊的讨论。在原始的 C 语言中，不会检查参数的数量与类型。在 C89 中，函数的形参可以用两种方式定义。首先，它们可以像原始的 C 语言那样定义，这就是说，参数的名字在括号中列出，类型声明在参数名之前。如下面的函数：

```
double sin(x)
  double x;
  { ... }
```

用这种方法可以避免类型检查。因此下面的调用：

```
double value;
 int count;
 ...
value = sin(count);
```

是合法的，但它不可能正确。

另一种方法称为原型法，在这种方法中，形参的类型包含在列表中。如下：

```
double sin(double x)
```

```
{ ... }
```

如果 sin 的这个版本用同样的形式调用，即用下面的调用，则它也是合法的：

```
value=sin(count);
```

实参的类型(int)将与形参的类型(double)核对。尽管它们并不匹配，但 int 可以强制转换为 double(这是扩展强制转换)，所以这个转换操作会执行。如果不能转换类型(例如实参是一个数组)或者参数的个数是错误的，就会检测出一个语义错误。所以在 C89 中，用户可以选择是否对参数进行类型检查。

在 C99 和 C++中，所有的函数都必须确定形参的原型格式。但是，用省略号替换列表最后的部分参数，可以避免对这些参数进行类型检查。如

```
int printf(const char* format_string, ...);
```

调用 printf 函数至少要包含一个参数——指向常量字符串的指针。除此之外，任何参数(甚至没有参数)都是合法的。Printf 根据字符串参数中是否有格式代码来决定是否有额外的参数。比如，整数输出的格式代码是%d，它是字符串的一部分，例如：

```
printf("The sum is %d\n", sum);
```

%告诉 printf 函数，有一个额外的参数。

一个更有趣的问题是通过引用传递基本类型时，实参要强制转换为形参的类型，例如在 C#中。假设一个方法调用将一个 float 值传递给 double 类型的形参。如果这个参数是按值传送的，float 值就会强制转换成 double 类型，不会出问题。这个强制转换是很有用的，因为它允许通过一个库提供子程序的 double 版本，它可以用于 float 或 double 值。可是，假设这个参数是通过引用传递的。当 double 类型的形参值返回给调用程序中 float 类型的实参时，该值就会溢出。为了避免这个问题，C#要求用 ref 实参的类型匹配对应的形参类型(不允许强制转换类型)。

在 Python 和 Ruby 中，没有参数的类型检查，因为这些语言中的类型是另一个概念。在这些语言中，对象有自己的类型，而变量没有，因此形参也没有类型。这样就不允许进行参数类型检查。

9.5.6 多维数组作为参数

存储映射函数用于将多维数组元素的下标映射到它在内存中的地址，参见第 6 章。在一些语言如 C 和 C++中，多维数组作为参数传递给子程序时，编译器必须在只能知道子程序代码(不是调用子程序时)的条件下，为这个数组建立映射函数。这是可能的，因为子程序可以与调用它的程序分开编译。考虑 C 中向函数传递矩阵参数的程序。C 中的多维数组是数组的数组，它们是按行来存储的。下面的存储映射函数用于按行来存储的矩阵，所有索引的下界都是 0，元素的个数是 1：

```
地址(mat[i, j])=地址(mat[0, 0])+i*列数+j
```

注意这个映射函数需要列数，而不是行数。因此，在 C 和 C++中，当矩阵作为参数传送时，形参必须在第二对括号中包含列数。如下面的 C 框架程序所示：

```
void fun(int matrix[][10]) {
  ... }
void main() {
  int mat[5][10];
  ...
  fun(mat);
  ...
}
```

这种把矩阵传递为参数的方法，有一个问题：它不允许程序员编写一个函数，来接受不同列数的矩阵；而必须为每个列数不同的矩阵写一个新函数。事实上，这就不允许编写灵活的、可高效重用的函数，来处理多维数组。在 C 和 C++里，因为有指针运算，所以有一个办法来避免这个问题。矩阵可以传递为指针，矩阵的实际维数也可以传递为参数。然后，每次要引用矩阵中的一个元素时，这个函数就可以使用指针运算来计算用户写的存储映射函数，得到矩阵元素的地址。例如，考虑下面的函数原型：

```
void fun(float *mat_ptr,
         int num_rows,
         int num_cols);
```

下面的语句可以用来将变量 x 的值传递给 fun 函数中矩阵参数的[row][col]元素：

```
*(mat_ptr + (row * num_cols) + col) = x;
```

虽然这行得通，但显然很难阅读、非常复杂，易出错。使用宏指令来定义函数存储映射可以降低阅读的难度，比如：

```
#define mat_ptr(r,c) (*mat_ptr + ((r) *
                      (num_cols) + (c)))
```

如此，便可以把代码写成：

```
mat_ptr(row,col) = x;
```

其他的语言采用不同的方式来解决传递多维数组的问题。Ada 的编译器可以在编译子程序时，决定用作参数的所有数组的维数。在 Ada 中，无限制的数组类型可以成为形参。所谓无限制的数组类型是指，数组索引的范围在定义数组类型时并不给出。而无限制的数组类型的变量定义必须给出索引的范围。若给子程序传递了无限制的数组，子程序的代码就从与这个参数相关的实参中获得索引的范围。例如以下定义：

```
type Mat_Type is array (Integer range <>,
                        Integer range <>) of Float;
Mat_1 : Mat_Type(1..100, 1..20);
```

下面的函数返回 Mat_Type 数组元素之和：

```
function Sumer(Mat : in Mat_Type) return Float is
   Sum : Float := 0.0;
   begin
   for Row in Mat'range(1) loop
     for Col in Mat'range(2) loop
       Sum := Sum + Mat(Row, Col);
     end loop; -- for Col ...
 end loop; -- for Row ...
 return Sum;
end Sumer;
```

range 属性返回实参数组的给定下标的范围，所以无论实参的大小或者索引的范围如何，它都是有效的。

在 Fortran 中，这个问题通过下面的办法解决：数组形式的形参必须在函数定义之后声明。对于一维数组，这个声明中的下标是不相关的，但是对于多维数组，这种声明中的下标允许编译器构建存储映射函数。考虑下面示例的 Fortran 子例程框架：

```
Subroutine Sub(Matrix, Rows, Cols, Result)
   Integer, Intent(In) :: Rows, Cols
   Real, Dimension(Rows, Cols), Intent(In) :: Matrix
   Real, Intent(In) :: Result
   ...
End Subroutine Sub
```

只要 Rows 实参的值和所传递的矩阵的行数一样，这个子例程才能正确运行。因为 Fortran 按列存储数组，所以行数是必需的。如果所传递的数组目前没有将所定义的范围都装满有用的数据，就可以把所定义的数组下标范围和已填充的下标范围都传递给子程序。其中，所定义的数组下标范围用于数组的局部声明，已填充的下标范围用来控制所引用数组元素的计算。例如下面的 Fortran 子程序：

```
Subroutine Matsum(Matrix, Rows, Cols, Filled_Rows,
      Filled_Cols, Sum)
   Real, Dimension(Rows, Cols), Intent(In) :: Matrix
   Integer, Intent(In) :: Rows, Cols, Filled_Rows,
                          Filled_Cols
   Real, Intent(Out) :: Sum
   Integer :: Row_Index, Col_Index
   Sum = 0.0
   Do Row_Index = 1, Filled_Rows
     Do Col_Index = 1, Filled_Cols
       Sum = Sum + Matrix(Row_Index, Col_Index)
     End Do
 End Do
End Subroutine Matsum
```

Java 和 C#使用一种技术传递作为参数的多维数组，这与 Ada 的相似。在 Java 和 C# 中，数组是对象，都是一维的，但是数组的元素也可以是数组。每个数组都继承了一个命

名常量(Java 中的 length 和 C#中的 Length)，用来在创建数组对象时设置数组的长度。矩阵形参总是带两对空括号，下面的 Java 方法与 Ada 的示例函数 Sumer 功能相同：

```
float sumer(float mat[][]) {
  float sum = 0.0f;
  for (int row = 0; row < mat.length; row++) {
    for (int col = 0; col < mat[row].length; col++) {
      sum += mat[row][col];
    } //** for (int row ...
  } //** for (int col ...
  return sum;
}
```

因为每个数组有自己的长度值，所以在一个矩阵中，行可以有不同的长度。

9.5.7 设计考虑

选择参数传递方式有两个重要的考虑：是否有效，和需要单向还是双向数据传输。

目前的软件工程原则要求，尽可能减少通过子程序代码访问子程序外部的数据。要达到这个目标，只要没有数据通过参数传回调用程序，就应使用输入参数。而当没有数据传送给被调用的子程序，但子程序必须向调用程序返回数据时，就应使用输出参数。最后，只有数据必须在调用程序和被调用的子程序间双向移动时，才使用输入输出参数。

一个实际的问题与这个原则相冲突。有时为单路参数传输提供访问路径是合理的。例如，把一个巨大的数组传递给一个子程序，而该子程序又没有修改它时，单向方式会更好。然而，按值传递需要将整个数组移动到子程序的一个局部存储区域中。这将花费大量的时间和空间。因此，巨大的数组常常通过引用传递。这是 Ada 83 允许实现者在两种结构化参数传递方式中选择的原因。C++固定的引用参数提供了另一个解决方式。另一个可以选择的方式将允许使用者在方法间选择。

函数传递参数方式的选择关乎另一个设计问题：函数的副作用。参见 9.10 节。

9.5.8 参数传递的例子

考虑如下的 C 函数：

```
void swap1(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}
```

假如这个函数命名为：

```
Swap1(c, d);
```

C 采用的是按值传递。Swap1 的作用可以用下列伪代码来描述：

```
a = c          - Move first parameter value in
b = d          - Move second parameter value in
temp = a
a = b
b = temp
```

虽然最后 a 获得了 d 的值，b 获得了 c 的值，但是 c 和 d 的值都没有改变，因为没有给调用程序传递回任何内容。

可以修改 C 的交换函数，来处理指针参数，达到按引用传递的效果：

```
void swap2(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

swap2 可以这样调用：

```
swap2(&c, ad);
```

swap2 的作用可以这样来描述：

```
a = &c         - Move first parameter address in
b = &d         - Move second parameter address in
temp = *a
*a = *b
*b = temp
```

在这个例子中，交换操作是成功的，c 和 d 的值已经交换了。swap2 可以在 C++中用引用参数写成：

```
void swap2(int &a, int &b) {
   int temp = a;
   a = b;
   b = temp;
}
```

这个简单的交换操作不可能在 Java 中实现，因为 Java 既没有指针，也没有 C++中的那种引用。在 Java 中，引用变量只能指向对象，而不能指向标量值。

只要没有涉及别名，按值-结果传递的语义和按引用传递是一样的。Ada 为输入输出模式的标量参数使用按值传递方式。为了了解按值-结果传递，考虑下面的函数 swap3，假设采用了按值-结果传递的参数。它是用类似 Ada 的语法写的。

```
procedure swap3(a : in out Integer, b : in out Integer) is
   temp : Integer;
   begin
   temp := a;
   a := b;
   b := temp;
  end swap3;
```

假设调用 swap3：

```
swap3(c, d);
```

swap3 调用后的操作为：

```
addr_c = &c       - Move first parameter address in
addr_d = &d       - Move second parameter address in
a = *addr_c       - Move first parameter value in
b = *addr_d       - Move second parameter value in
temp = a
a = b
b = temp
*addr_c = a       - Move first parameter value out
*addr_d = b       - Move second parameter value out
```

这样，这个交换子程序运行正确。接着考虑下面的调用：

```
swap3(i, list[i]);
```

在这种情况下，操作为：

```
addr_i = &i             - Move first parameter address in
addr_listi= &list[i]    - Move second parameter address in
a = *addr_i             - Move first parameter value in
b = *addr_listi         - Move second parameter value in
temp = a
a = b
b = temp
*addr_i = a             - Move first parameter value out
*addr_listi = b         - Move second parameter value out
```

子程序仍运行正确，因为在这个例子中返回的参数值的地址是在调用子程序时计算的，而不是在返回时计算的。如果实参的地址是在返回时计算的，结果就是错误的。

最后，探讨一下在使用按值传递和按引用传递时创建了别名，会出现什么情况。考虑下面用类似 C 的语法写的程序框架：

```
int i = 3; /* i is a global variable */
void fun(int a, int b) {
  i = b;
}
void main() {
  int list[10];
  list[i] = 5;
  fun(i, list[i]);
}
```

在 fun 函数中，如果使用按引用传递，i 和 a 就都是别名。如果使用按值-结果传递，i 和 a 就不是别名。假定按值-结果传递，fun 函数的执行步骤是：

```
addr_i = &i              - Move first parameter address in
addr_listi = &list[i]    - Move second parameter address in
a = *addr_i              - Move first parameter value in
b = *addr_listi          - Move second parameter value in
i = b - Sets i to 5
*addr_i = a              - Move first parameter value out
*addr_listi = b          - Move second parameter value out
```

在这个例子中，fun 函数给全局变量 i 的赋值是从 3 到 5，但是复制回的第一个形参(例子中的倒数第二行)将它设置回 3。注意，如果使用了按引用传递，复制回的结果就不会是同一个含义，i 的值也还是 5。还要注意，因为第二个参数的地址是在 fun 函数开始执行时计算的，所以对全局变量 i 的任何修改都不影响最后用来返回 list[i]的地址。

9.6 子程序作为参数

在编程中，如果一个子程序的名字可以作为参数传递给其他子程序，常常是最容易处理的。一个常见的例子就是子程序必须对某个数学函数取样。例如，实现积分的子程序在函数曲线上取样大量不同的点，来估算函数曲线下的面积。编写出这个函数后，它就应能用于任何给定的函数，不必为每个需要积分的函数重写代码。因此很自然地，计算数学函数积分值的程序函数名就应作为参数传给积分子程序。

虽然这个想法很自然，看起来也很简单，但是实现的细节却让人感到困惑。如果只需要传递子程序的代码，就可以用一个指针来实现。但是，这有两个难题。

第一，检查作为参数传递的子程序的类型是有问题的。在 C 和 C++中，函数不能传递为参数，但是函数指针可以。函数指针的类型包括了函数的协议。因为这个协议包含所有的参数类型，这些参数可以进行完整的类型检查。Fortran 95+有一个机制，可以为当做参数传递的子程序提供类型，它们也必须进行类型检查。

第二个把子程序当做参数的难题只出现在允许嵌套子程序的语言中，问题是执行被传递的子程序应使用什么引用环境。有三个选择：

- 调用语句的环境，它执行被传递的子程序(浅层绑定)。
- 被传递的子程序的定义环境(深层绑定)。
- 调用语句的环境，它把子程序传递为实参(临时绑定)。

下面的程序例子是用 JavaScript 语法写的，它说明了这三种选择：

```
function sub1() {
  var x;
  function sub2() {
    alert(x); // Creates a dialog box with the value of x
    };
  function sub3() {
    var x;
    x = 3;
```

```
    sub4(sub2);
    };
  function sub4(subx) {
    var x;
    x = 4;
    subx();
    };
  x = 1;
  sub3();
};
```

考虑调用 sub4 时 sub2 的执行。对于浅层绑定，该执行的引用环境是 sub4 的引用环境，因此对 sub2 中 x 的引用绑定到 sub4 中的局部变量 x 上，程序的输出即为 4。对于深层绑定，sub2 的执行引用环境是 sub1 的引用环境，所以 sub2 中 x 的引用绑定到 sub1 的局部变量 x 上，输出是 1。对于临时绑定，绑定到 sub3 的局部变量 x 上，输出是 3。

在一些情况下，子程序声明一个子程序，将它作为参数传递。在这种情况中，深层绑定和临时绑定是一样的。从来不使用临时绑定，可能是因为过程显示为参数的环境与被传递的子程序之间没有自然连接。

浅层绑定不适合支持嵌套子程序、静态作用域的语言。例如，设想 Sender 过程将 Sent 过程作为参数传递给 Receiver 过程。问题是 Receiver 可能不在 Sent 的静态环境中，因此 Sent 访问 Receiver 的变量会很不自然。另一方面，在这种语言中，任意子程序，包括传递为参数的子程序，通过其定义的词典位置来决定引用环境是很正常的。所以在这类语言中使用深层绑定更加合理。一些动态作用域的语言使用浅层绑定。

历史注释

Pascal(Jensen 和 Wirth, 1974)的初始定义允许把子程序传递为参数，不必包含其参数类型信息。如果可以进行独立的编译(这在最初的 Pascal 中是不可能的)，编译器甚至不允许检查参数的数量是否正确。若不能进行独立的编译，就可以检查参数的一致性，但这是一个非常复杂的任务，通常是不检查的。

9.7 间接调用子程序

在一些情况下，子程序必须间接调用。当要调用的子程序在运行之前是未知的时，常常需要间接调用。调用子程序时需要使用子程序的指针或引用，而子程序的指针或引用在调用之前设置。间接调用子程序的两个常见应用是图形用户界面中的事件处理(现在，几乎所有的 Web 应用程序和许多非 Web 应用程序都有这种事件处理)和回调(在这种情况下，调用子程序，并在所调用的子程序完成其工作时通知调用者)。与往常一样，这里不讨论编程的这种特定类型，而考虑支持它们的编程语言。

间接调用子程序的概念并不是最近才开发出来的。C 和 C++都允许程序定义函数的指针，通过函数指针就可以调用函数。在 C++中，函数指针根据函数的返回类型和参数类型

来确定其类型，这样一个指针就只能指向有特定协议的函数。例如，下面的声明定义了一个指针(pfun)，它可以指向把 float 和 int 作为参数、返回 float 的任意函数：

```
float (*pfun)(float, int);
```

任何与这个指针遵循相同协议的函数，都可以用作这个指针的初始值，或者在程序中把该函数赋予这个指针。在 C 和 C++中，不带圆括号的函数名(例如没有方括号的数组名)是函数(或数组)的地址。所以下面的语句都是给函数指针指定初始值或赋值的合法方式：

```
int myfun2 (int, int);   // A function declaration
int (*pfun2)(int, int) = myfun2;   // Create a pointer and
                                   // initialize
                                   // it to point to myfun2
pfun2 = myfun2; // Assigning a function's address to a
                // pointer
```

现在，函数 myfun2 可以用如下语句调用：

```
(*pfun2)(first, second);
pfun2(first, second);
```

第一个语句明确地解除了指针 pfun2 的引用，这是合法的，但不是必不可少的。

C 和 C++的函数指针可以发送为参数，从函数中返回，但函数不能直接用于这两种情形。

在 C#中，方法指针变成了对象，从而提高了其功能和灵活性。它们称为委托，因为程序不是调用方法，而是将这个操作交托给委托来完成。

要使用委托，首先要使用特定的方法协议来定义委托类。委托的实例保存着方法的名称和能调用的委托协议。委托的声明语法与方法声明相同，但在返回类型之前要插入保留字 delegate。例如下面的语句：

```
public delegate int Change(int x);
```

这个委托可以用任何带 int 参数、返回 int 的方法来实例化。例如下面的方法声明：

```
static int fun1(int x);
```

把这个方法名发送给委托的构造函数，就可以实例化委托 Change，如下所示：

```
Change chgfun1 = new Change(fun1);
```

这个语句可以简写为：

```
Change chgfun1 = fun1;
```

下面通过委托 chgfun1 来调用 fun1：

```
chgfun1(12);
```

委托类的对象可以存储多个方法。第二个方法可以用操作符+=来添加，如下所示：

```
Change chgfun1 += fun2;
```

即使 fun2 以前的值为空，这个语句也把 fun2 放在委托 chgfun1 中。存储在一个委托实例中的所有方法都按它们在委托实例中的放置顺序来调用。这称为多播调用。无论方法返回什么值，该调用都只返回最后一个调用的方法所返回的值或对象。当然，这意味着在大多数情况下，通过多播委托调用的方法都返回 void。

在上例中，把一个静态方法放在委托 Change 中。实例方法也可以通过委托来调用，此时委托必须存储该方法的引用。委托也可以是泛型委托。

委托用于.NET 应用程序的事件处理，它们也用于实现闭包(参见 9.12 节)。

与 C 和 C++一样，Python 中不带圆括号的函数名就是函数的指针。Ada 95 有指向子程序的指针，但 Java 没有。在 Python、Ruby 和大多数函数式语言中，子程序的处理方式与数据相同，所以可以把它们赋予变量。因此，在这些语言中，并不需要指向子程序的指针。

9.8 重载子程序

重载的运算符可以有多种含义，重载运算符的实例由其操作数的类型决定。比如，如果在 Java 程序中，*运算符有两个浮点操作数，就表示浮点数乘法。但如果同样的运算符有两个整型操作数，就表示整数乘法。

重载的子程序就是在同样的引用环境中与另一个子程序同名的子程序。重载子程序的每个版本必须有一个独特的协定，也就是说，它必须与其他版本在参数的个数、顺序、类型或返回类型(假设它是一个函数)上有所不同。调用重载子程序的意义由实参列表和/或返回值类型(假设它是一个函数)决定。尽管不是必要的，重载子程序通常执行相同的过程。

C++、Java、Ada 和 C#包含了预定义的重载子程序。比如，C++、Java 和 C#的许多类都重载了构造函数。由于每个版本的重载子程序都有独特的参数列表，所以编译器可以通过不同类型的参数来区分对这些重载子程序的调用。遗憾的是，情况并不是这么简单。参数在允许时可以进行强制类型转换，使这个区分过程变得无比复杂。简单地说，问题是：如果没有一个方法的参数列表与方法调用中实参的数量和类型相匹配，然而有两个或多个方法的参数列表可以通过强制转换来匹配，应调用哪个方法呢？语言设计者要回答这个问题，必须决定怎样给不同的强制转换操作排序，使编译器能选择“最佳”的方法匹配调用的方法。这将是一项复杂的任务。为了了解这个过程的复杂程度，建议读者阅读 C++中区分方法调用的准则(Stroustrup，1997)。

由于 C++、Java 和 C#支持混合类型表达，返回类型就不能用于区分重载函数(或方法)。调用的上下文不允许确定返回类型。比如，如果 C++程序有两个名叫 fun 的函数，它们都有一个 int 参数，但其中一个返回 int，另一个返回 float，程序就不会编译，因为编译器不能决定用哪个版本的 fun 函数。

用户也可以在 Ada、Java、C++、C#和 F#中编写多种版本的同名子程序。在 C++、Java 和 C#中，最常用的用户定义的重载方法是构造函数。

重载含有默认参数的子程序会导致有歧义的子程序调用。例如下面的C++代码：

```
void fun(float b = 0.0);
void fun();
...
fun();
```

这个调用是有歧义的，会导致一个编译错误。

9.9 泛型子程序

软件重用技术是对软件业的一个重大贡献。提高软件重用率的一种方法是，削减创建不同子程序的需求，这些子程序为不同数据类型实现了相同的算法。例如，程序员不应编写四个不同的子程序，为仅仅只是元素类型不同的四个数组排序。

多态子程序带不同类型的参数，执行不同的操作。重载子程序提供了一种特殊的多态性——临时多态性。重载子程序的行为不需要彼此相似。

支持面向对象编程的语言通常也支持子类型多态性。子类型多态性表示，类型T的变量可以访问类型T或派生于T的任意对象。

Python和Ruby的方法提供了更一般的多态性。这些语言中的变量没有类型，所以形参也没有类型。因此，只要定义了方法用于形参的操作符，该方法就可以使用任意类型的实参。

参数多态性可以通过带泛型参数的子程序提供，泛型参数在类型表达式中使用，描述了子程序的参数类型。这种子程序的不同实例化可以用不同的泛型参数来提供，产生接受不同参数类型的子程序版本。子程序的参数定义都是一致的。参数多态子程序往往也叫做泛型子程序。Ada、C++、Java 5.0、C# 2005和F#都提供一种编译时的参数多态性。

9.9.1 C++中的泛型函数

C++泛型函数的描述性名称是模板函数，模板函数的一般定义形式如下：

```
template <模板参数>
可能包含模板参数的函数定义
```

模板参数(至少一个)有如下形式：

```
class 标识符
typename 标识符
```

这个类的形式用于类型名。typename形式用于向模板函数传递值。例如，有时向模板函数传递一个整型值作为数组的大小会很方便。

一个模板可以接受另一个模板作为参数，实际上，常常接受一个指定了用户自定义泛

型类型的模板类作为参数，但这里不考虑此种情况[8]。

下面是模板函数的一个例子：

```
template <class Type>
Type max(Type first, Type second) {
  return first > second ? first : second;
}
```

其中，参数 Type 指定了函数所操作的数据类型。这个模板函数可以用在任何定义了运算符>的类型上。例如可以用 int 作为参数实例化这个模板：

```
int max(int first, int second) {
  return first > second ? first : second;
}
```

这个过程可以定义为一个宏，但宏的缺点是，如果其参数是有副作用的表达式，宏就不能正确执行。例如假定宏定义为：

```
#define max(a, b) ((a) > (b)) ? (a) : (b)
```

这个定义是泛化的，因为它能用于任何数值类型。但如果用有副作用的参数调用它，它就不能总是正确执行，例如：

```
max(x++, y)
```

结果是

```
((x++)>(y))? (x++): (y))
```

只要 x 的值大于 y，x 就会自增两次。

在调用中指定 C++模板函数，或运算符&提取函数的地址时，模板函数就会隐式实例化。例如，下面的代码实例化了示例模板函数两次，一次以 int 为参数，一次以 char 为参数。

```
int a, b, c;
char d, e, f;
...
c = max(a, b);
f = max(d, e);
```

下面是 C++泛型排序子程序：

```
template <class Type>
void generic_sort(Type list[], int len) {
  int top, bottom;
  Type temp;
  for (top = 0; top < len - 2; top++)
    for (bottom = top + 1; bottom < len - 1; bottom++)
```

8. 模板类参见第 11 章。

```
        if (list[top] > list[bottom]) {
          temp = list[top];
          list[top] = list[bottom];
          list[bottom] = temp;
        } //** end of if (list[top] ...
    } //** end of generic_sort
```

下面是这个目标函数的实例化例子：

```
float flt_list[100];
...
generic_sort(flt_list, 100);
```

C++的函数模板是子程序的一个远亲，其形参类型在调用中动态绑定到实参类型上。在这种情况下，只需要代码的一个副本，而对于C++的实现方式，必须在编译期间为所需的每个不同类型创建一个副本，而子程序调用对子程序的绑定是静态的。

9.9.2 Java 5.0 中的泛型方法

Java 5.0 增加了对泛型类型和方法的支持。在 Java 5.0 中，泛型类的名字指定为类名后跟一个或多个用尖括号界定的类型变量。例如，

```
generic_class<T>
```

其中 T 是类型变量。泛型详见第 11 章。

Java 的泛型方法和 C++的泛型子程序有几个重要的区别。首先，泛型参数必须是类，不能是基本类型。因此不能用泛型方法来模仿前面的 C++示例，该例中数组单元是泛型类型，可以是基本类型。在 Java 中，数组单元(与容器相反)不能是泛型。其次，尽管 Java 的泛型方法可以实例化任意多次，但是只建立代码的一个副本。泛型方法的内部版本叫做原始方法，它操作的是 Object 类对象。当泛型值从泛型方法返回时，编译器会把它转换为合适的类型。其三，在 Java 中，作为泛型参数传送给泛型方法的类可以限制其范围。这种限制称作绑定。

下面是 Java 5.0 泛型方法的框架定义：

```
public static <T> T doIt(T[] list) {
  ...
}
```

这定义了一个方法 doIt，它将泛型类型的数组元素作为参数。这个泛型参数的名字叫 T，必须是数组。下面是调用 doIt 的例子：

```
doIt<String>(MyList);
```

现在考虑下面的 doIt，它有一个对泛型参数的绑定。

```
public static <T extends Comparable> T doIt(T[] list) {
  ...
}
```

这里定义的方法使用泛型数组参数，而数组元素的类型是实现了 Comparable 接口的类。这是对泛型参数的约束或绑定。保留字 extends 意味着泛型类会派生其后的类。但是，在上文中，extends 有不同的意思。表达式<T extends BoundingType>指出，T 应该是绑定类型的一个子类型。所以 extends 在这里的意思是，泛型类(或接口)扩展绑定类(如果绑定是一个类，就是绑定)，或者实现绑定接口(如果绑定是一个接口)。绑定保证了任何泛型的实例化元素都能与 Comparable 方法 CompareTo 进行比较。

若泛型方法的泛型类型有两个或两个以上的限制，就应将它们添加到以符号&分隔的 extends 子句中。泛型方法也可以有多个泛型参数。

Java 5.0 支持通配符类型。例如 Collection<?>是集合类的通配符类型，这个类型可用于任意类中的任何集合类型。例如下面的泛型方法：

```
void printCollection(Collection<?> c) {
  for (Object e: c) {
     System.out.println(e);
  }
}
```

无论 Collection 类的元素的类是什么，这个方法都会打印 Collection 类的元素。通配符类型的对象有一些需要注意的地方，例如，确定了这个类型的特定对象的元素类型时，其他类型的对象就不能添加到集合中。例如：

```
Collection<?> c = new ArrayList<String>();
```

除非放入集合的类型是 String，否则使用 add 方法往这个集合里添加元素是非法的。

通配符类型可以限制，就像非通配符类型那样。这样的类型称为受限的通配符类型。例如下面的方法头：

```
public void drawAll(ArrayList<? extends Shape> things)
```

这里的泛型类型是一个通配符类型，它是 Shape 类的一个子类。该方法可以画出任何 Shape 子类的对象。

9.9.3 C# 2005 中的泛型方法

C# 2005 中泛型方法的功能和 Java 5.0 中的类似，不过它不支持通配符类型。C# 2005 泛型方法的一个独特特性是，调用泛型方法时，如果编译器能够推断出某个实参的类型，这个实参类型就可以省略。例如下面的类定义框架：

```
class MyClass {
  public static T DoIt<T>(T p1) {
    ...
  }
}
```

如果编译器能够从调用的实参中推断出泛型类型，DoIt 方法就能够在不指定泛型参数

的情况下调用。例如，下面的两个调用都是合法的：

```
int myInt = MyClass.DoIt(17); // Calls DoIt<int>
string myStr = MyClass.DoIt('apples');
    //Calls DoIt<string>
```

9.9.4 F#中的泛型函数

F#的类型推断系统并不总是能确定参数的类型或函数的返回类型。此时，F#就给一些函数推断参数和返回值的泛化类型。这称为自动概括。例如下面的函数定义：

```
let getLast (a, b, c) = c;;
```

因为不包含类型信息，所以函数和返回值的类型都推断为泛化类型。因为这个函数没有包含任何计算，所以这是一个简单的泛型函数。

函数可以定义为包含泛型参数，如下所示：

```
let printPair (x: 'a) (y: 'a) =
    printfn "%A %A" x y;;
```

%A 格式规范可用于任何类型。类型 a 前面的撇号表示，它是一个泛型类型[9]。这个函数定义是有效的(带有泛型参数)，因为它不包含有类型限制的操作。

算术操作符就是有类型限制的操作符。例如下面的函数定义：

```
let adder x y = x + y;;
```

类型推断系统把 x、y 和返回值的类型设置为 int。因为 F#中没有类型强制转换，所以下面的调用是非法的：

```
adder 2.5 3.6;;
```

即使参数的类型设置为泛型，+操作符也把 x 和 y 的类型设置为 int。

泛型类型也可以在尖括号中明确指定，如下所示：

```
let printPair2<'T> x y =
    printfn "%A %A" x y;;
```

这个函数必须用一个类型[10]来调用，如下所示：

```
printPair2<float> 3.5 2.4;;
```

由于 F#有类型推断系统，但缺乏类型强制转换，所以与 C++、Java 5.0+和 C# 2005+相比，F#泛型函数的用处不大，尤其是数值计算。

---

9. a 没有什么特别的，它可以是任何合法的标识符。根据约定，它使用字母表开头的小写字母。

10. 显式类型转换指定，泛型类型必须用以 T 开头的大写字母来命名。

9.10 函数的设计问题

下面是函数的设计问题：

- 允许函数有副作用吗？
- 函数可以返回哪些类型的值？
- 一共能返回多少个值？

9.10.1 函数的副作用

由于函数在表达式中调用时有副作用，如第 5 章所述，所以函数的参数应总是输入型参数。实际上，一些语言需要这个条件，例如，Ada 函数只能有输入型形参。这种要求有效地防止了函数通过参数或参数和全局变量的别名而产生副作用。然而，在大多数的其他命令式语言中，函数可以有按值传递的参数和按引用传递的参数，这样函数就可以产生副作用和别名。

纯函数式语言，例如 Haskell，没有变量，所以其函数没有副作用。

9.10.2 返回值的类型

大多数命令式编程语言都限制了函数的返回值类型。C 允许其函数返回除数组和函数之外的任意类型。数组和函数可以用指针类型的返回值来处理，C++与 C 相似，但 C++还允许函数返回用户定义的类型或类。Ada、Python、Ruby 和 Lua 是当前命令式语言中仅有的允许函数(和/或方法)返回任意类型的语言。然而，在 Ada 中，由于函数不是它的一种类型，所以它不能返回函数。当然，函数能够返回函数指针。

在一些编程语言中，子程序是第一流的对象，这意味着它们能够传送为参数，从函数中返回，赋予变量。在一些命令式语言中，例如 Python、Ruby 和 Lua，方法是第一流的对象。在大多数函数式语言中，函数也是第一流的对象。

Java 和 C#都没有函数，但它们的方法与函数相似。在这两种语言中，方法可以返回任意类型或类。因为方法不是类型，所以不能返回。

9.10.3 返回值的个数

在大多数语言中，一个函数只能返回一个值。然而，事无绝对。Ruby 语言允许方法返回多个值。如果在 Ruby 方法中，return 语句后面没有跟着一个表达式，就返回 nil。如果后跟一个表达式，就返回表达式的值。如果后面有多个表达式，就返回所有表达式的值组成的数组。

Lua 也允许函数返回多个值。这些值用逗号隔开，跟在 return 语句后，如下例：

```
return 3, sum, index
```

调用函数的语句形式决定了调用者收到返回值的个数。如果函数是作为一个过程调用，

就像一个语句一样，那么所有的返回值都会被忽略。如果要让函数返回三个值且都要被调用者保留，那么函数应该像下例这样被调用：

```
a, b, c=fun()
```

在F#中，可以返回多个值，具体操作是将它们置于一个元组中，并使此元组成为函数的最后一个表达式。

9.11 用户定义重载运算符

在Ada、C++、Python和Ruby语言中，用户可以重载运算符。假定开发一个Python类，来支持复数及其算术操作。复数可以用两个浮点数表示。Complex类的两个成员是real和imag。在Python中，实现二元算术操作时，要给第一个操作数调用方法，并作为参数给该调用传送第二个操作数。对于加法，方法是__add__。例如表达式x+y实现为x.__add__(y)。为了给新类Complex的对象重载+运算，只需要为Complex提供一个__add__方法来执行该操作，如下：

```
def __add__ (self, second):
  return Complex(self.real + second.real, self.imag +
   second.imag)
```

在大多数支持面向对象编程的语言中，当前对象的引用会隐式传送给每个方法调用。在Python中，这个引用必须显式传送，所以self是__add__方法的第一个参数。

在C++中，可以给Complex类编写出该示例加法方法[11]：

```
Complex operator +(Complex &second) {
  return Complex(real + second.real, imag + second.imag);
}
```

9.12 闭包

定义闭包是很简单的，闭包就是一个子程序和定义它的引用环境。如果子程序可以从程序的任意位置调用，就需要引用环境。但闭包的解释并不简单。

如果作用域为静态的编程语言不允许嵌套子程序，闭包就没有什么用，所以这类语言不支持闭包。在这种语言中，子程序引用环境中的所有变量(其本地变量和全局变量)都是可访问的，无论子程序在程序的什么地方调用。

子程序可以嵌套时，子程序的引用环境除了包含本地和全局变量之外，还可以包含在所有外层子程序中定义的变量。如果子程序只能在活动的、可见的外层作用域中调用，这就不是个问题。但是，如果子程序可以在其他地方调用，这就是个问题了。如果子程序可

11. C++和Python都有复数的预定义类，所以例子中的方法是不必要的，这里仅用于演示。

以传递为参数或赋予变量，就可能在程序的几乎所有地方调用。它还有一个相关问题：子程序可能在嵌套它的子程序结束之后调用，这一般意味着，在外层子程序中定义的变量已释放了其内存——它们不再存在。对于可以在程序的任意地方调用的子程序，其引用环境必须在调用它的地方可用。因此，在嵌套子程序中定义的变量可能需要与整个程序相同的生存期，而不只是定义它们的子程序的生存期。生存期与整个程序相同的变量具有无限制的作用域。这通常意味着它们必须是堆动态变量，而不是栈动态变量。

几乎所有函数式编程语言、大多数脚本编程语言和至少一种命令式语言 C#都支持闭包。这些语言都有静态的作用域，允许嵌套子程序[12]，也允许把子程序传递为参数。下面是用 JavaScript 编写的闭包示例：

```
function makeAdder(x) {
    return function(y) {return x + y;}
}
...
      var add10 = makeAdder(10);
      var add5 = makeAdder(5);
      document.write("Add 10 to 20: " + add10(20) +
       "<br />");
      document.write("Add 5 to 20: " + add5(20) +
       "<br />");
```

这段代码假定嵌套在一个 HTML 文档中，并用浏览器显示其输出：

```
Add 10 to 20: 30
Add 5 to 20: 25
```

在这个例子中，闭包是在 makeAdder 函数中定义的匿名函数，makeAdder 函数也返回了这个闭包。在闭包函数中引用的变量 x 绑定到发送给 makeAdder 的参数上。

makeAdder 函数调用了两次，一次是用参数 10 来调用，另一次是用参数 5 来调用。每次调用都返回闭包的不同版本，因为它们绑定到 x 的不同值上。makeAdder 的第一次调用创建了一个函数，它把 10 加到其参数上，第二次调用也创建了一个函数，它把 5 加到其参数上。函数的两个版本绑定到 makeAdder 的不同操作上。显然，调用 makeAdder 时创建的 x 的生存期必须扩展为整个程序的生存期。

在 C#中，这个闭包函数可以用嵌套的匿名委托来编写。嵌套方法的类型指定为带 int 参数并返回匿名委托的函数。返回类型用该委托的特定记号 Func<int, int>来指定。尖括号中的第一个类型是参数类型。这个委托可以封装只有一个参数的方法。第二个类型是委托所封装的方法的返回类型。

```
static Func<int, int> makeAdder(int x) {
  return delegate(int y) { return x + y;};
}
```

12. 在 C#中，可以嵌套的方法只有匿名委托和 λ 表达式。

```
...
Func<int, int> Add10 = makeAdder(10);
Func<int, int> Add5 = makeAdder(5);
Console.WriteLine("Add 10 to 20: {0}", Add10(20));
Console.WriteLine("Add 5 to 20: {0}", Add5(20));
```

这段代码的结果与前面 JavaScript 闭包示例的结果完全相同。

匿名委托可以编写为 λ 表达式。下面用 λ 表达式替代委托，重写了 makeAdder 方法体。

```
return y => x + y
```

Ruby 中的一些块即使在变量已消失的地方调用，也可以引用在定义的位置可见的变量。这就把这些块变成了闭包。

9.13 协同程序

协同程序是一种特殊的子程序。在传统意义上的子程序模式里，调用程序与被调用的子程序之间存在主从关系，而调用程序和被调用的协同程序之间是一种更加平等的关系。事实上，协同程序的控制机制常称作“对称单元控制”模型。

协同程序可以有多个由它自己控制的入口点，能够在多次调用的过程中维持自身的状态。这就是说，协同程序都必须是历史敏感的，因此有静态的局部变量。协同程序的二次执行常常不是从它的开头部分开始的，因此，协同程序的调用称为恢复，而不是调用。

比如下面的协同程序框架：

```
sub co1(){
  ...
  resume co2();
  ...
  resume co3();
  ...
}
```

co1 第一次恢复执行时，它从第一条语句处开始执行，然后向下运行，在执行到恢复 co2 时，程序会将控制权传递给 co2。col 下次恢复执行时，会从调用 co2 后面的第一条语句开始执行。co1 第三次恢复执行时，会从恢复 co3 后面的第一条语句开始执行。

协同程序保有子程序的一个常见特征：在任意时刻，只有一个协同程序在运行。

如上例所示，协同程序常常在执行一部分语句后，将控制权移交给另一个协同程序，而不是一次执行完毕。重启动协同程序时，它会从移交控制权的那条语句的下一句开始执行。这种相互交织的执行序列与多任务操作系统的工作方式相关。在多任务操作系统中，虽然可能只有一个处理器，但所有运行的程序似乎在并行执行，其实它们共用同一个处理器。对协同程序而言，这有时叫做准并发。

一般情况下，应用程序中的协同程序是由名为“主单元”的程序单元创建的(主单元不

是协同程序)。协同程序创建时，会执行初始化代码，然后把控制权移交回主单元。构建了整个系列的协同程序后，主单元会继续执行一个协同程序，然后该系列的协同程序会以一定的顺序执行，直至任务完成(前提是任务是可以完成的)。如果一个协同程序执行到了末尾，控制权就会移交到创建它的主单元。这就是结束一组协同程序运行的机制——当然，这是希望它们结束的情况下。在一些程序里，只要电脑在运行，协同程序就在运行。

利用一组协同程序可以解决的一个问题是模拟扑克牌游戏。假设该游戏的四个玩家都使用同一种策略。要模拟这个游戏，可以让主单元创建一组协同程序，每个协同程序都有自己的一手牌。主单元在模拟打牌过程时，可以恢复执行一个协同程序，这个协同程序在出牌后，恢复执行下一个协同程序，如此循环，直至游戏结束。

假设 A 和 B 是两个协同程序，图 9-3 展示了 A 和 B 的两种执行序列。

在图 9-3(a)中，协同程序 A 的执行是由主单元启动的。在执行了一些语句后，A 启动 B。B 第一次把控制权移交给 A 时，A 会从上一次执行结束的地方继续执行，注意，A 的局部变量值与上一次执行结束时一致。图 9-3(b)展示了协同程序 A 和 B 的另一种执行序列，这里 B 的执行是由主单元启动的。

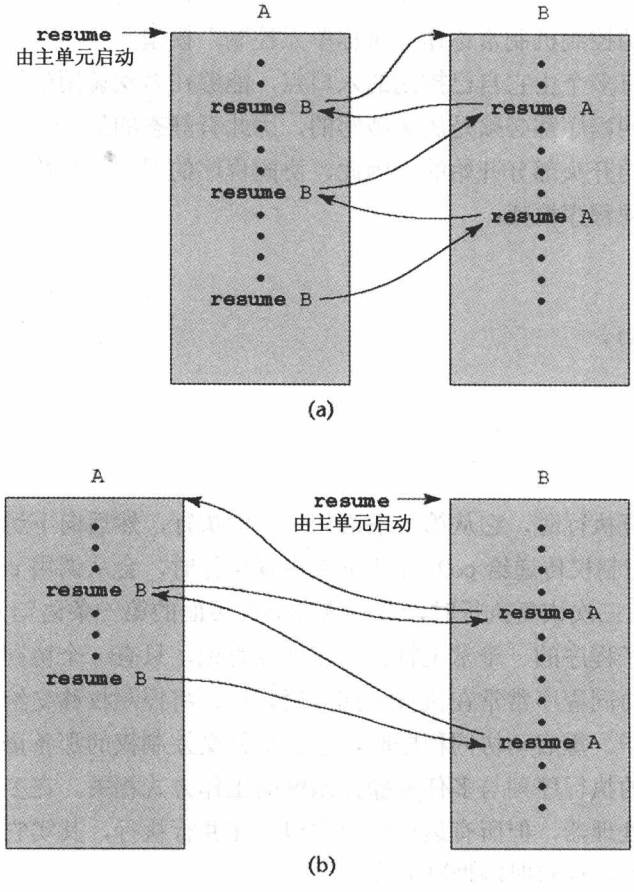


图 9-3 两个没有循环的协同程序的两种可能的执行控制顺序

协同程序通常有一个包含 resume 语句的循环，而不是像图 9-3 中那样。图 9-4 展示了这种情形的执行序列：A 由主单元启动，它在自己的主循环里启动 B，而 B 又会在自己的主循环里启动 A。

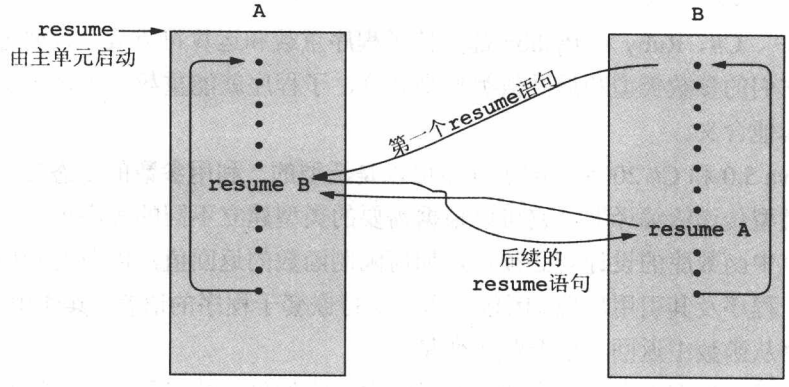


图 9-4 有循环的协同程序的执行顺序

在现代语言中，只有 Lua 完全支持协同程序[13]。

小 结

过程抽象在编程语言里由子程序表示。子程序的定义描述了这个子程序代表的行为，而对它的调用执行这些行为。子程序头标识了子程序定义并提供接口，也就是它的原型。

形参是用于表示子程序调用中实参的名字。Python 和 Ruby 使用数组和哈希表作为形参，来支持不定个数的参数。Lua 和 JavaScript 也支持不定个数的参数。实参可以通过地址或关键字与形参联系起来。参数可以有默认值。

子程序既可以是函数，也可以是过程。函数是对数学函数的模型化，用来定义新的操作，而过程用来定义新的语句。

子程序中的局部变量可以是栈动态变量，以提供对递归调用的支持；也可以是静态变量，以提供高效率和历史敏感的局部变量。

JavaScript、Python、Ruby 和 Lua 允许子程序的定义互相嵌套。

参数传递有三大基本语义模型：输入模式、输出模式和输入输出模式，它们有许多实现方式：按值传递，按结果传递，按值-结果传递，按引用传递，按名字传递。在大多数语言中，参数是在运行时栈里传递的。

在按引用传递参数时会出现别名，这种情况不仅会在两个或多个参数间发生，也会在某个参数和可访问的非局部变量间发生。

多维数组参数会给语言设计者带来难题，因为被调用的子程序需要知道如何计算存储映射函数。这不仅仅需要数组的名字。

---

13. 但是，Python 的生成器是一种协同程序。

把子程序名作为参数提供了必要的服务，但很难理解。难点在于，执行传递为参数的子程序时，如何确定它的引用环境。

C 和 C++支持函数指针。C#有委托，委托是可以存储方法引用的对象。委托可以存储多个方法引用，支持多播调用。

Ada、C++、C#、Ruby 和 Python 都支持子程序重载和运算符重载。只要各种重载的版本能够通过它们的参数类型和返回值类型来区分，子程序就能重载。函数定义可以用于构建操作符的其他含义。

C++、Java 5.0 和 C# 2005 中的子程序可以是泛型的。利用参数的多态性，可以把数据对象的期望类型传递给编译器，它可以根据需要的类型建立不同的程序单元。

编程语言中函数库的设计者必须决定如何限制函数的返回值，以及返回值的个数。

闭包是子程序及其引用环境。闭包可用于支持嵌套子程序的语言，其作用域是静态的，它允许子程序从函数中返回，以及赋予变量。

协同程序是一种有多个入口点的特殊子程序。它们可以让子程序交替运行。

复 习 题

1. 子程序的三个一般特征是什么？
2. 子程序处于活动状态，是什么意思？
3. 子程序头里有什么信息？
4. Python 子程序有什么与其他语言不同的特征？
5. 哪些语言支持数量可变的参数？
6. 什么是 Ruby 的数组形参？
7. 什么是参数描述？什么是子程序协议？
8. 什么是形参？什么是实参？
9. 关键字参数有什么优点和缺点？
10. 函数和过程有什么区别？
11. 子程序的设计问题有哪些？
12. 动态局部变量有什么优点和缺点？
13. 静态局部变量有什么优点和缺点？
14. 哪些语言允许子程序嵌套定义？
15. 参数传递的三大语义模型是什么？
16. 按值传递、按结果传递、按值-结果传递、按引用传递分别有什么优点和缺点？它们分别是以什么方式传递的？
17. 描述按引用传递参数时可能发生别名的所有情况。
18. 最初的 C 语言和 C89 处理实参和形参类型不统一的区别是什么？
19. 参数传递方法有哪两个基本设计问题？
20. 描述把多维数组作为参数传递带来的问题。

21. Ruby 使用的是什么参数传递方式？
22. 把子程序名作为参数传递会带来哪两个问题？
23. 对于作为参数传递的子程序，定义其引用环境的浅层绑定和深层绑定。
24. 什么是重载子程序？
25. 什么是参数多态性？
26. 什么会让 C++的模板函数实例化？
27. Java 5.0 泛型方法中的泛型参数与 C++方法的泛型参数有什么本质不同？
28. 如果 Java 5.0 方法返回一个泛型类型，实际上返回的是哪种类型的对象？
29. 如果用三个不同的泛型参数调用 Java 5.0 泛型方法，编译器会为这个方法创建多少种不同的版本？
30. 函数的设计需要考虑什么问题？
31. 哪两种语言允许函数返回多个值？
32. 委托是什么？
33. F#泛型函数的主要缺点是什么？
34. 什么是闭包？
35. 哪些语言特性使闭包有效？
36. 哪些语言允许用户重载运算符？
37. 协同程序与传统的子程序有什么不同？

习 题

1. 在 Python 和 C++中，用户程序可以为已有的运算符构建额外的定义，这样做有什么好处和坏处？你认为这种用户定义的运算符重载好还是不好？找论据支持你的论点。
2. 在大多数的 Fortran IV 实现中，参数是按引用传递的，而且只通过访问路径来传送。请举出这个设计的优缺点。
3. Ada 83 设计者决定，允许实现者选择以复制方式或引用方式实现输入输出参数。找论据支持这个决定。
4. 假设要编写一个方法，在新的输出页面上打印标题和页码。第一次调用该方法时，页码是 1，以后每调用一次增加 1。如果不用参数和非局部变量的引用来实现这个方法，这在 Java 中能做到吗？在 C#中呢？
5. 考虑下面的 C 语法程序：

```
void swap(int a, int b) {
  int temp;
  temp = a;
  a = b;
  b = temp;
}
void main() {
  int value = 2, list[5] = {1, 3, 5, 7, 9};
```

```
  swap(value, list[0]);
  swap(list[0], list[1]);
  swap(value, list[value]);
}
```

对以下的每种参数传递方法，在每个 swap 调用之后，变量 value 和 list 的值分别是什么？

a. 按值传递

b. 按引用传递

c. 按值-结果传递

6. 举一论据反对子程序中同时存在静态局部变量和动态局部变量。
7. 考虑下面的 C 语法程序。

```
void fun (int first, int second) {
  first += first;
  second += second;
}
void main() {
  int list[2] = {1, 3};
  fun(list[0], list[1]);
}
```

对以下的每一种参数传递方法，在程序执行完毕后，list 数组中的值是什么？

a. 按值传递

b. 按引用传递

c. 按值-结果传递

8. 举一论据反对 C 中只提供函数子程序的设计。
9. 找一本 Fortran 教材，学习语句函数的语法和语义。论证它们存在的合理性。
10. 学习 C++和 Ada 中用户定义的运算符重载方法，然后写一个报告，用评价语言的标准比较两种方法。
11. C#支持输出参数，但 Java 和 C++都不支持。解释这种不同之处。
12. 研究 Jensen 的策略，它是一种有名的按名字传递参数的方法。简短地描述其定义和用法。
13. 研究 Ruby 和 CLU 中的迭代器机制，并列出它们的相似之处和不同之处。
14. 考虑编程语言中的子程序嵌套问题。为什么很多现代的语言都不允许这样做？
15. 列出至少两个论据，反对按名字传递参数。
16. 详细比较 Java 5.0 和 C# 2005 中的泛型子程序。

编程练习

1. 用你熟知的语言写一个程序，来确定按引用传递和按值传递一个大数组所需的时间比。在你使用的机器和实现中，要让这个数组尽可能大。如果需要，多次传递这个数组，以获得传递操作的精确时间比。

2. 写一个 C#或者 Ada 程序，来确定何时计算输出参数的地址(是在子程序调用时还是执行结束时)。
3. 写一个 Perl 程序，把一个字面量按引用传递给一个子程序，并让子程序试图更改这个参数的值。考虑到 Perl 的设计思路，解释结果。
4. 把第 3 题用 C#重做一遍。
5. 用一种支持静态局部变量和动态栈变量的语言写一个程序。在子程序里创建 6 个大矩阵(至少 100×100)，其中三个是静态的，三个是栈动态的。把两个静态矩阵和两个栈动态矩阵用 1 到 100 的随机数填满。子程序中的代码必须对静态矩阵执行大量的矩阵乘法操作，并对这一过程计时。然后对栈动态矩阵重复同样的操作。比较结果并解释原因。
6. 写一个 C#程序，它包含两个调用很多次的方法。两个方法都把一个很大的数组作为参数，一个按值传递，一个按引用传递。比较调用这两个方法所用的时间，并解释其中的不同。一定要调用它们足够多次，来显示出所需时间的差异。
7. 用你喜欢的语言写一个程序，在分别以按引用传递和按值-结果传递方式来传递参数时，要能产生不同的结果。
8. 写一个 Ada 泛型函数，它的参数是一个由泛型元素构成的数组，和一个与数组元素类型相同的标量。数组元素和标量的类型是泛型参数。数组的下标是正整数。这个函数必须在给定的数组中查找给定的标量，并返回这个标量在数组中的下标。如果标量不在数组中，函数必须返回-1。用 Integer 和 Float 实例化这个函数，并进行测试。
9. 写一个 C++泛型函数，它的参数是一个由泛型元素构成的数组，和一个与数组元素类型相同的标量。数组元素和标量的类型是泛型的参数。数组的下标是正整数。这个函数必须在给定的数组中查找给定的标量，并返回这个标量在数组中的下标。如果标量不在数组中，函数必须返回-l。使用类型 int 和 float 来测试该函数。
10. 设计一个子程序和调用它的代码，使之按引用传递和按值-结果传递一个或多个参数时，会产生不同的结果。

第10章

实现子程序

10.1 调用和返回的一般语义
10.2 实现“简单”的子程序
10.3 通过栈动态局部变量实现子程序
10.4 嵌套子程序
10.5 块
10.6 动态作用域的实现

本章的主旨是探讨子程序的实现，讨论子程序链接是如何进行的，为什么 ALGOL 60 对于 20 世纪 60 年代早期那些缺少警惕性的编译器编写人员来说是一个挑战。我们将从最简单的情况开始，即有静态局部变量且不能嵌套的子程序；然后是有栈动态局部变量的更复杂的子程序；最后是有栈动态局部变量、静态作用域的嵌套子程序。在支持嵌套子程序的语言中，实现子程序的难度有所增加，因为需要包含访问非局部变量的机制。

我们将详细讨论在静态作用域的语言中访问非局部变量的静态链方法。接着描述实现代码块的技术。最后讨论在动态作用域的语言中访问非局部变量的几种方法。

10.1 调用和返回的一般语义

子程序的调用和返回操作统称为子程序链接。子程序的实现必须基于待实现语言的子程序链接的语义。

在一般语言中，会有很多操作与子程序调用关联。调用过程必须包含所用参数传递方式的具体实现。如果局部变量不是静态的，则调用过程必须为被调用子程序中声明的局部变量分配存储空间，并将这些变量绑定到分配的存储空间上。调用程序单元的执行状态也得保存下来。执行状态包括能让调用程序单元恢复执行的所有内容，包括寄存器的值、CPU 的状态位，以及环境指针(EP)。环境指针(详见 10.3 节)用于在子程序执行期间访问参数和局部变量。调用过程也必须把控制权移交给子程序的代码，并保证在子程序执行完毕时，控制权能够返回到正确的地方。最后，如果语言支持嵌套子程序，调用过程就必须产生某种机制，以访问对被调用子程序可见的非局部变量。

子程序返回所需要的操作没有调用操作那么复杂。如果子程序具有输出型或者输入输出型参数，并用复制方式来实现，返回过程的第一个操作就是把与形参关联的局部值移到实参上，然后，必须释放局部变量的存储空间，并恢复调用程序单元的执行状态。最后，必须把控制权返回给调用程序单元。

10.2 实现“简单”的子程序

我们从实现简单子程序的任务开始。这里所说的“简单”，是指子程序不能嵌套，而且所有局部变量都是静态的。早期的 Fortran 就是拥有这种子程序的语言。

调用“简单”子程序的语义要求进行以下操作：

(1) 保存当前程序单元的执行状态；

(2) 计算并传递参数；

(3) 将返回地址传递给被调用子程序；

(4) 把控制权交给被调用子程序。

从“简单”子程序返回的语义要求包括以下操作：

(1) 如果存在按值-结果传递或者输出型的参数，就将这些参数的当前值移到对应的实参上；

(2) 如果子程序是一个函数，就把函数值移到调用程序单元能访问的地方；

(3) 恢复调用程序单元的执行状态；

(4) 把控制权交还给调用程序单元。

调用和返回操作需要存储以下内容：

- 调用程序单元的状态信息；
- 参数；
- 返回地址；
- 函数的返回值；
- 子程序代码使用的临时数据。

这些信息，以及局部变量和子程序代码，就组成了子程序在执行和返回控制权给调用程序单元时所需要的全部信息。

现在的问题是，调用者和被调用者之间的调用操作和返回操作的分配。对于简单的子程序，答案对于整个过程的大部分内容是显而易见的。调用的最后三个操作显然必须由调用者完成。保存调用者的执行状态可以由调用者和被调用者中的任何一个来完成。在返回时，第一、第三、第四个操作必须由被调用者来做。恢复调用者的执行状态也可以由调用者和被调用者中的任何一个来完成。总的来说，被调用者的链接操作能在两个不同的时刻发生，即被调用子程序执行的开始或结束。这两个操作有时也称作子程序链接的“序幕”和“尾声”。对于简单的子程序，所有被调用者的链接操作发生在其执行结束时，所以不需要“序幕”。

简单子程序由两个独立的部分组成：子程序的实际代码(不会改变)，局部变量以及之前提到的数据(在子程序执行时会改变)。对于简单子程序，这两部分的大小是固定的。

格式或布局是子程序中的非代码部分，称作活动记录，因为它描述的数据只在活动期间或者子程序执行期间是相关的。活动记录的形式是静态的。活动记录实例是活动记录的一个具体例子，即活动记录形式的一组数据。

因为使用简单子程序的语言不支持递归，所以每次子程序只能有一个活动版本。因此，子程序的活动记录只能有一个实例。活动记录的一种可能布局如图 10-1 所示。这里和本章剩余部分省略了活动记录中应该存储的调用者执行状态，因为它比较简单，而且与本章的讨论无关。

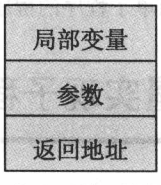


图 10-1　简单子程序的活动记录

由于“简单”子程序的活动记录实例有固定的大小，所以它可以静态地分配存储空间。事实上，它可以依附到子程序的代码部分。

图 10-2 展示了一个程序，它由主程序和 A、B、C 三个子程序组成。尽管在图上，所有的代码段与所有的活动记录实例是分开的，但在有些情况下，活动记录实例依附到它们

所关联的代码段上。

图 10-2 中完整程序的构造并不完全由编译器完成。事实上，由于语言允许单独编译，四个程序单元 MAIN、A、B、C 可能在不同的日子，甚至不同的年份被编译了。在编译每个程序单元时，它的机器代码与一系列对外部子程序的引用一起写入一个文件。图 10-2 显示的可执行程序由链接器(操作系统的一部分)放到一起。有时，链接器也称作装入程序、链接器/装入程序或链接编辑器。当主程序调用链接器时，它的第一个任务就是查找包含该程序引用的已翻译子程序的文件，并将它们载入内存。然后，链接器必须把主程序里调用子程序的目标地址设置成那些子程序的入口地址。当然，在被载入的子程序中对其他子程序的调用操作以及对库子程序的所有调用操作，链接器也必须完成相同的工作。在前面的例子中，为 MAIN 调用了链接器。链接器必须找到 A、B、C 的机器代码程序，以及它们的活动记录实例，并将之与 MAIN 的代码一起载入内存。然后，它必须为 A、B、C 的所有调用，以及对在 A、B、C 和 MAIN 中库子程序的所有调用设置目标地址。

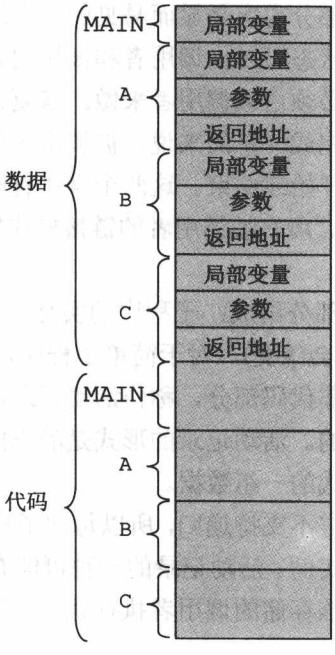


图 10-2 含有简单子程序的程序代码和活动记录

10.3 通过栈动态局部变量实现子程序

现在研究在局部变量是栈动态变量的语言里，如何实现子程序链接，重点是调用和返回操作。

栈动态局部变量的最大优势之一是对递归的支持。因此，使用栈动态局部变量的语言也支持递归。

子程序可以嵌套时，复杂度也将增加，这部分内容将在 10.4 节讨论。

10.3.1　更复杂的活动记录

在使用栈动态局部变量的语言里，子程序链接比简单子程序的链接更复杂，原因如下：

- 编译器必须生成代码来实现局部变量的隐式分配和释放。
- 递归增加了子程序有多个同时发生的活动的可能性，这意味着在某个给定的时刻，子程序可能存在多个(未执行完的)实例，其中包括至少一个来自子程序外部的调用和一个或多个递归调用。活动的数量只受限于机器内存的大小。每个活动需要自己的活动记录实例。

在大多数语言里，给定子程序的活动记录格式在编译时都是已知的。在很多情况下，活动记录的大小也是已知的，因为所有的局部数据都有固定的大小。但在其他一些语言里就不是这样。例如在 Ada 里，局部数组的大小可以由实参的值决定。此时，格式是静态的，但大小可以是动态的。在使用栈动态局部变量的语言里，活动记录实例必须动态创建。这种语言的典型活动记录如图 10-3 所示。

由于返回地址、动态链接和参数都由调用程序单元放在活动记录实例里，所以这些记录项必须首先出现。

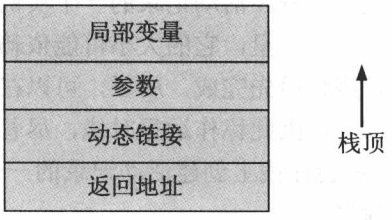


图 10-3　使用栈动态局部变量的语言里的典型活动记录

返回地址通常由一个指向操作的指针构成，该操作位于调用程序单元代码段中的调用语句之后。动态链接是指向调用者活动记录实例底部的指针。在静态作用域的语言里，当程序发生运行错误时，这个链接用来提供回溯信息。在动态作用域的语言里，这个动态链接用来访问非局部变量。活动记录里的实参是调用者提供的值或地址。

局部的标量变量绑定到活动记录实例内的存储空间上。结构类型的局部变量有时在别处分配空间，而且，只有它们的描述符和指向那部分存储空间的指针是活动记录的一部分。在被调用的子程序里，给局部变量分配空间并可能进行初始化，于是它们在最后出现。

考虑如下的 C 函数框架：

```
void sub(float total, int part) {
  int list[5];
  float sum;
  ...
}
```

sub 的活动记录如图 10-4 所示。

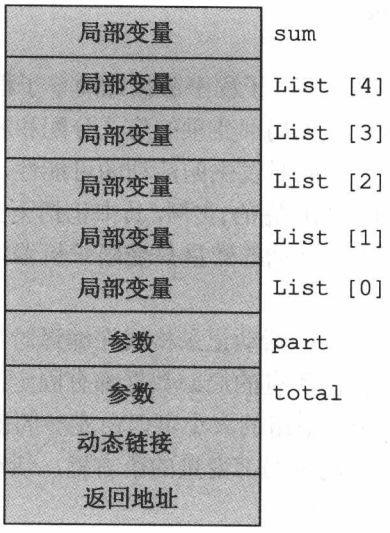


图 10-4 函数 sub 的活动记录

激活子程序需要动态地创建该子程序活动记录的一个实例。如前所述，在编译时，活动记录的格式是固定的，但在一些语言里，它的大小可能依赖于调用过程。因为调用和返回的语义规定，最后调用的子程序将最先完成，所以，可以在栈上创建这些活动记录的实例。这个栈是运行时系统的一部分，因此称作运行时栈，尽管我们通常只把它称作栈。每个子程序活动，无论递归与否，都会在栈上创建活动记录的一个新实例。这就提供了参数、局部变量和返回地址所需的单独复本。

还需要控制子程序的执行，即环境指针。一开始，环境指针指向底部，即主程序活动记录实例的第一个地址。因此，运行时系统必须保证它总是指向当前执行的程序单元的活动记录实例的底部。调用子程序时，当前的环境指针在新的活动记录实例中保存为动态链接。然后，环境指针设置成指向新活动记录实例的底部。当子程序返回后，栈顶设置成当前环境指针的值减去 1，环境指针设置成刚执行完的子程序的活动记录实例中的动态链接。重置栈顶可以有效地移除最上面的活动记录实例。

环境指针用作活动记录实例数据内容(参数及局部变量)偏移寻址的基址。

注意，当前使用的环境指针并不存储在运行时的栈里。唯一保存的版本以动态链接的形式存储在活动记录实例内。

前面讨论了链接过程中的几个新操作。10.2 节中的列表必须修订，将这些新操作考虑在内。使用本节给出的活动记录格式，新的操作如下：

调用者的操作是：

(1) 创建活动记录实例；

(2) 保存当前程序单元的执行状态；

(3) 计算并传递参数；

(4) 将返回地址传递给被调用的程序；

(5) 把控制权交给被调用的程序。

被调用者的预置操作如下：

(1) 在栈里以动态链接的形式保存旧的环境指针，并创建新的值；

(2) 给局部变量分配存储空间。

被调用者的后续操作如下：

(1) 如果存在按值-结果传递或者输出型的参数，就将这些参数的当前值移到对应的实参上；

(2) 如果子程序是一个函数，就把函数值移到调用程序单元能访问的地方；

(3) 恢复栈指针，把它设置成当前环境指针的值减去 1，并将环境指针设置成旧的动态链接；

(4) 恢复调用程序单元的执行状态；

(5) 把控制权交还给调用程序单元。

如第 9 章所述，子程序自调用起一直到执行完毕都是活动的。而在它变为不活动时，它的局部作用域将不复存在，其引用环境也不再有意义。所以在那一刻，它的活动记录实例就可以销毁了。

参数并不总是传递到栈中。在许多为 RISC 机器设计的编译器中，参数会传递到寄存器中。这是因为 RISC 机器的寄存器数量通常比 CISC 机器多。然而，在本章的剩余部分中，假设参数传递到栈中。很容易修改这个方法，将参数传递到寄存器中。

10.3.2　一个不含递归调用的例子

考虑如下的 C 程序框架：

```
void fun1(float r) {
  int s, t;
  ... <-------------- 1
  fun2(s);
  ...
}

void fun2(int x) {
  int y;
  ... <-------------- 2
  fun3(y);
  ...
}

void fun3(int q) {
  ... <-------------- 3
}
void main() {
  float p;
  ...
  fun1(p);
  ...
}
```

在这个程序中，调用函数的顺序是：

```
main 调用 fun1
fun1 调用 fun2
fun2 调用 fun3
```

图 10-5 是标记 1、2 和 3 处栈内的内容。

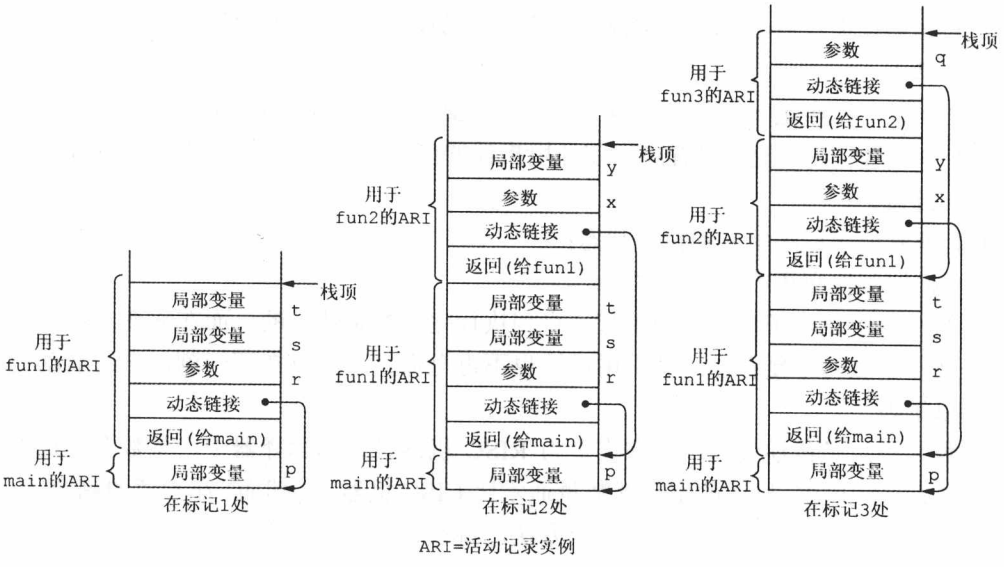


图 10-5 程序中三个标记点处的栈内容

在标记 1 处，只有 main 函数和 fun1 函数的活动记录实例储存在栈中。当 fun1 调用 fun2 时，就在栈上创建 fun2 函数的一个活动记录实例。当 fun2 调用 fun3 时，就在栈上创建 fun3 函数的一个活动记录实例。fun3 函数执行完后，它的活动记录实例就从栈上清除，并用 EP 重置栈顶指针。函数 fun2 和 fun1 终止时，也会执行类似的过程。从 main 调用 fun1 的地方返回后，栈中就只剩下 main 的活动记录实例了。注意，有些实现方案并不像以上的例子那样，在栈上为 main 函数使用活动记录实例。然而，它也可以这样做，并且这样能简化实现过程和我们的讨论。在这个例子以及本章的所有其他例子中，都假设栈从低位地址向高位地址增长，然而在某个具体的实现方案里，栈可能反方向增长。

在给定的时刻，栈中的动态链接集合称为动态链或者调用链。它表示程序是如何执行到当前位置的动态历史，而当前的执行位置总是在活动记录实例位于栈顶的子程序代码中。对局部变量的引用可以在代码中表示为从活动记录的局部作用域开始处算起的偏移量，而局部作用域开始处的地址储存在 EP 中。这个偏移量称为局部偏移量。

变量在活动记录中的局部偏移量可以在编译时，凭借与活动记录相关的子程序声明的变量的顺序、类型和大小来确定。为了使讨论更简单，假设所有变量均只在活动记录中占一个位置。第一个在子程序中声明的局部变量会在活动记录中分配到从下往上数，参数个数加二的位置(最下端的两个位置留给返回地址和动态链接)。第二个声明的局部变量离栈顶的位置比第一个声明的变量近一个位置，以此类推。例如之前的示例程序。在 fun1 函数中，s 变量的局部偏移量是 3，t 变量的是 4。同样，在 fun2 函数中，y 的局部偏移量是 3。要得到任意局部变量的地址，只需要给 EP 加上该变量的局部偏移量。

10.3.3　递归调用

下面的 C 程序例子使用递归来计算阶乘：

```
int factorial(int n) {
    <-------------- 1
  if (n <= 1)
    return 1;
  else return (n * factorial(n - 1));
    <-------------- 2
 }
void main() {
  int value;
  value = factorial(3);
    <-------------- 3
 }
```

图 10-6 展示了 factorial 函数的活动记录格式。注意它为函数返回值包括额外一项。

图 10-7 展示了程序三次执行到 factorial 函数内标记 1 处时的栈内容。每次执行到该位置，栈的内容都比之前多一个活动记录，但没有定义函数值。第一个活动记录实例包括了调用函数的返回地址，其他实例包含 factorial 函数的返回地址，这些地址都用于递归调用。

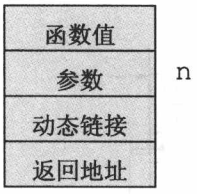


图 10-6　factorial 函数的活动记录

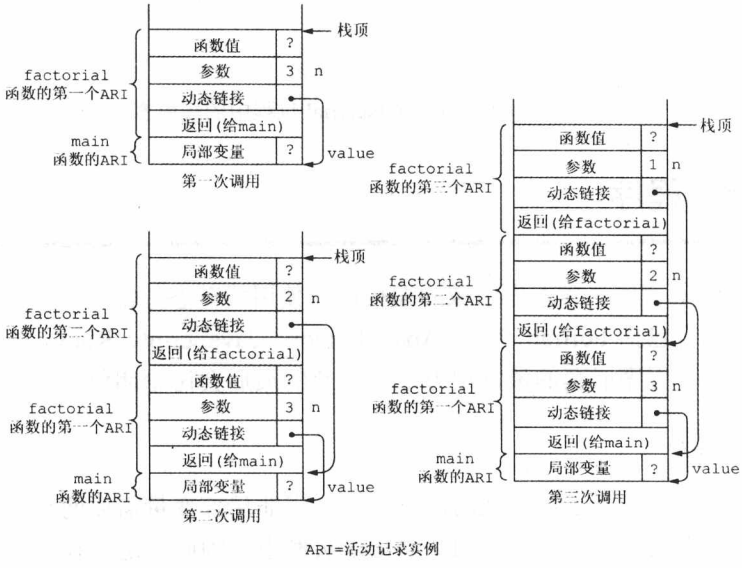


图 10-7　factorial 中标记 1 处的栈内容

图 10-8 展示了程序三次执行到 factorial 函数内标记 2 处时的栈内容。标记 2 处是指

return 指令执行之后，活动记录从栈上移除之前。factorial 函数的代码计算参数 n 的当前值和递归调用函数的返回值的乘积。Factorial 的第一个返回值是 1，该活动记录实例给参数 n 保存了值 1，计算出来的乘积 1 返回给 factorial 的第二个活动，该值乘以参数值 2，得到的 2 返回给 factorial 的第一个活动，该值再乘以参数值 3，得到最终的函数值 6，返回给 main 中对 factorial 的第一次调用。

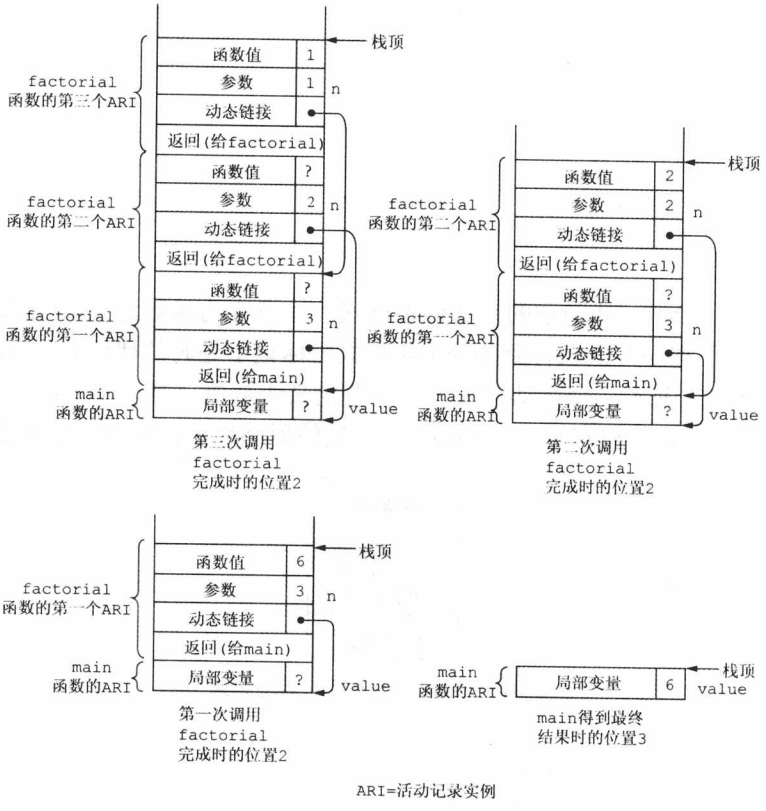


图 10-8 执行 main 和 factorial 过程中的栈内容

10.4 嵌套子程序

一些不基于 C 语言的、静态作用域的编程语言使用栈动态局部变量，并允许子程序相互嵌套。这样的语言包括 Fortran 95+、Ada、Python、JavaScript、Ruby、Lua 和函数式语言。本节研究实现子程序嵌套的最常用方法。本节的最后会讨论闭包。

10.4.1 基础知识

在支持嵌套子程序、静态作用域的语言中，引用非局部变量需要两步。所有能以非局部方式访问的非静态变量，都在已有的活动记录实例中，因此一定也在栈中某处。访问过程的第一步是在栈中找到定义变量的活动记录实例，第二步是在该活动记录实例中使用变量的局部偏移量来访问它。

找到正确的活动记录实例是这两步中较有趣、也较困难的一步。首先，注意在给定的

子程序中，只有在静态祖先作用域中声明的变量是可见的、可访问的。其次，当变量由嵌套子程序引用时，所有静态祖先的活动记录实例都一定在栈上。这由静态作用域语言的静态语义法则来保证：子程序仅当它的所有静态祖先子程序是活动的时，才能调用[1]。如果某个静态祖先不是活动的，它的局部变量就不会绑定到储存器中，所以允许访问它们是没有意义的。

非局部引用的语义指出，正确的声明是在外层作用域中找到的第一个声明，它最靠近第一个嵌套的子程序。所以，为了支持非局部引用，必须能在栈中找到对应静态祖先的所有活动记录实例。于是人们找到了下一小节解释的实现方法。

10.5 节解决程序块的问题。所以在本节的剩余部分里，假设所有的作用域都是由子程序定义的。因为在基于 C 的语言中，函数是不能嵌套的(在这些语言中，唯一的静态作用域是用块创建的作用域)，本节的讨论并不直接适用于这些语言。

10.4.2　静态链

在允许嵌套子程序的语言中，实现静态作用域的最常见方法是静态链。在这种方法中，给活动记录添加一个叫做静态链接的新指针。静态链接有时称为静态作用域指针，它指向静态父类的活动记录实例底部，用于访问非局部变量。通常，静态链接在活动记录中出现在参数的下方。在活动记录中加入静态链接，会使局部偏移量发生改变。之前在参数前有两个活动记录元素，而现在有返回地址、静态链接和动态链接三个元素出现在参数下方了。

静态链是一条由连接栈中特定活动记录实例的静态链接所组成的链。在子程序 P 的执行过程中，P 的活动记录实例中的静态链接指向 P 的静态父程序单元的活动记录实例。而该实例中的静态链接又指向 P 的静态祖父程序单元的活动记录实例(假设 P 有静态祖父程序单元)。所以，静态链按照静态父单元优先的顺序，连接了正在执行的子程序的所有静态祖先。这条链很显然可以用于在静态作用域语言中，实现对非局部变量的访问。

使用静态链接找到非局部变量的正确活动记录实例比较简单。要引用非局部变量时，可以搜索静态链，找到一个包括该变量的静态祖先活动记录实例，就可以找到包括该变量的活动记录实例。然而，这实际上还更简单。因为作用域的嵌套在编译时是已知的，所以编译器不仅能判断引用是否是局部的，还可以判定寻找包括该非局部变量的活动记录实例需要经过的静态链的长度。

静态层次是一个与静态作用域相关的整数，它表示从最外围的作用域算起，该静态作用域的嵌套深度。没有嵌套在其他任何单元中的程序单元，其静态层次为 0。如果子程序 A 在非嵌套程序单元中定义，它的静态层次就是 1。如果子程序 A 还包括嵌套子程序 B 的定义，则 B 的静态层次就是 2。

引用非局部变量 X 时，找到正确的活动记录实例所需要经过的静态链的长度是引用 X 的子程序和声明 X 的子程序的静态层次之差。这个差值称为引用的嵌套层次，或者链偏移量。实际的引用可以用一对有序的整数(链偏移量，局部偏移量)来表示，这里的链偏移量是到正确活动记录实例的链接数(局部偏移量在 10.3.2 节讨论)。例如下面的 Python 框架程序：

---

1. 当然，闭包违反了这个规则。

```
# Global scope
...
def f1():
  def f2():
    def f3():
      ...
    #end of f3
    ...
  #end of f2
  ...
# end of f1
```

全局作用域、f1、f2 和 f3 的静态层次分别是 0、1、2 和 3。如果过程 f3 引用了一个在 f1 中声明的变量，该引用的链偏移量就为 2(即 f3 的静态层次减去 f1 的静态层次)。如果过程 f3 引用了一个在 f2 中声明的变量，该引用的链偏移量就为 1。对局部变量的引用可以用相同的机制来处理，此时链偏移量为 0，但在声明变量的子程序中，不使用指向活动记录实例的静态指针来作为基址，而是使用环境指针 EP。

为了阐明非局部访问的完整过程，请看下面的 Ada 框架程序：

```
procedure Main_2 is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin -- of Sub1
      A := B + C; <------------1
      ...
    end; -- of Sub1
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, E : Integer;
        begin -- of Sub3
        ...
        Sub1;
        ...
        E := B + A; <------------2
      end; -- of Sub3
      begin -- of Sub2
      ...
      Sub3;
      ...
      A := D + E; <------------3
    end; -- of Sub2
    begin -- of Bigsub
    ...
    Sub2(7);
    ...
  end; -- of Bigsub
  begin -- of Main_2
  ...
  Bigsub;
  ...
end; -- of Main_2
```

程序调用的顺序是：

Main2 调用 Bigsub
Bigsub 调用 Sub2
Sub2 调用 Sub3
Sub3 调用 Subl

当程序第一次运行至标记 1 时，栈状态如图 10-9 所示。

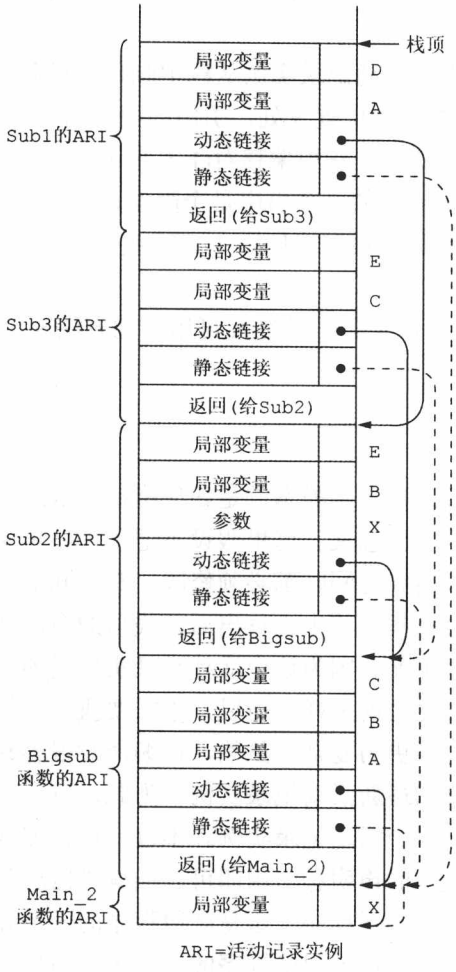


图 10-9　程序 Main_2 中标记点 1 处的栈内容

在过程 Sub1 的标记 1 处，引用了局部变量 A，而不是 Bigsub 中的非局部变量 A。这个引用的链偏移量/局部偏移量对为(0, 3)。对 B 的引用是 Bigsub 中的非局部变量 B，该引用可用偏移对(1,4)表示。其局部偏移量为 4，因为第一个局部变量的局部偏移量为 3(Bigsub 过程没有参数)。注意，如果用动态链接搜索一个包含 B 声明的活动记录实例，它会错误地找到在 Sub2 中声明的变量 B。而如果在动态链中使用偏移对(1,4)，就会使用 Sub3 中的变量 E。然而，静态链接指向 Bigsub 的活动记录，使用它能得到正确的变量 B。此时 Sub2

中的变量 B 将不在引用环境中，不能(正确地)访问。位置 1 上引用的 C 是 Bigsub 中定义的 C，它用偏移对(1,5)表示。

执行完 Sub1 后，Sub1 的活动记录实例将从栈中删除，并把控制权返回给 Sub3。而 Sub3 中位置 2 上引用的变量 E 为局部变量，可使用偏移对(0,4)访问。而该处引用的变量 B 为 Sub2 中声明的变量 B，因为 Sub2 是包含该声明的最近的静态祖先，它用偏移对(1, 4)访问。局部偏移量为 4，是因为 B 是在 Sub1 中声明的第一个变量，Sub2 有一个参数。位置 2 引用的变量 A 是 Bigsub 中声明的 A，因为 Sub3 及其静态父过程 Sub2 均没有变量 A 的声明。它用偏移对(2,3)来引用。

执行完 Sub3 后，Sub3 的活动记录实例从栈中删除，只剩下 Main_2、Bigsub 和 Sub2 的活动记录实例了。在 Sub2 中的位置 3 处，引用的变量 A 是 Bigsub 中定义的 A，因为现在只有活动的例程中有 A 的声明。它用偏移对(1,3)来访问。在这个位置，可见的作用域中都没有变量 D 的声明，所以对 D 的这个引用是个静态语义错误。编译器尝试计算链偏移量/局部偏移量对时，会检测到这个错误。位置 3 上的 E 是 Sub2 中的局部变量，可以用(0,5)访问。

综上所述，在位置 1、2 和 3 上引用的 A 可用下列方法表示：

- (0, 3)(局部)；
- (2, 3)(相隔 2 层)；
- (1, 3)(相隔 1 层)。

此时应了解在程序执行过程中，静态链是如何维持其状态的。如果它的维护太复杂，静态链的简洁性和高效性将毫无意义。这里假设不把子程序传递为参数。

静态链在每个子程序调用和返回时都必须修改，其返回值是不重要的：当子程序结束时，它的活动记录实例就会从栈中移出。移出后，栈顶的新活动记录实例就属于调用了刚刚结束的那个子程序的程序单元。因为这个活动记录实例里的静态链从不改变，它会正常工作，就好像调用其他子程序之前那样，所以，不需要执行任何操作。

调用子程序所需的操作会更为复杂。虽然，在编译时很容易确定正确的父作用域，但必须在调用时找到父作用域的最新活动记录实例。为此，可以在动态链上搜索活动记录实例，直到找到父作用域的第一个活动记录实例。不过，这种搜索是可以避免的，只要把子程序的声明和调用处理为变量的声明和引用即可。当编译器遇到子程序调用时，它会确定声明被调用子程序的子程序，该子程序一定是调用程序的一个静态祖先。然后，编译器计算嵌套深度，即在调用者与声明被调用子程序的子程序之间外层作用域的个数。这个信息可以保存起来，由子程序调用在执行期间进行访问。在调用时，为了确定被调用子程序的活动记录实例的静态链接，可以将调用者的静态链向下移动编译时计算出的嵌套深度。

再看看程序 Main_2 和图 10-9 中的栈状态。在 Sub3 中调用 Sub1 时，编译器确定，在声明了被调用过程 Sub1 的 Bigsub 中，Sub3(调用者)的嵌套深度是 2 级。在 Sub3 中执行调用 Sub1 的语句时，这些信息用于设定 Sub1 的活动记录实例的静态链接。这个静态链接设置为指向一个活动记录实例；在调用者的活动记录实例中，静态链中的第二个静态链接也指向这个实例。在这个例子中，调用者是 Sub3，它的静态链接指向其父亲(即 Sub2)的活动记录实例，而 Sub2 的活动记录实例的静态链接指向 Bigsub 的活动记录实例。所以 Sub1 的

新活动记录的静态链接设置为指向 Bigsub 的活动记录实例。

这种方法适用于一切子程序链接，除非把子程序传递为参数。

使用静态链来访问非局部变量的一个缺点是引用超出静态祖先作用域的变量需要耗费比引用局部变量更多的开销。必须沿着静态链前进，在每个外层作用域中，每次检查一个从引用到声明的链接。幸运的是，在实际中，长距离的变量引用是极少的，因此，这个缺点并不严重。静态链法的另一个缺点是程序员编写对时间要求极高的程序时，很难估计非局部变量引用的成本，因为每个引用的成本取决于引用和作用域声明之间的嵌套深度。更麻烦的是，代码的后续修改可能会改变嵌套深度，进而改变一些引用的时间成本，这些引用存在于已改变的代码和不修改的代码中。

人们已开发了一些替代静态链的方法，其中最有名的一种方法是使用名为 display 的辅助数据结构。但是，还没有一种方法优于静态链方法，因此静态链仍是使用最普遍的方法。所以，这里不讨论其他替代方法。

在不允许从函数中返回函数、不允许把函数赋予变量的语言中，本节描述的过程和数据结构正确实现了闭包。但是，对于不支持上述一个或两个操作的语言而言，这些仍是不够的。需要使用几个新机制来实现在这类语言中的非局部变量访问。首先，如果子程序在嵌套作用域而不是全局作用域中访问变量，该变量就不能仅存储在它自己的作用域的活动记录中。在子程序需要激活该活动记录之前，这个活动记录可能已经删除了。这种变量还可以存储在堆中，并进行无限制的扩展(其生存期是整个程序的生存期)。第二，子程序必须有访问存储在堆中的非局部变量的机制。第三，在堆上分配的非局部访问变量，每次在更新其栈上的版本时，也必须更新它。显然，这是对使用静态链实现静态作用域的重要扩展。

10.5　块

如第 5 章所述，许多语言，包括基于 C 的语言，都为变量提供了用户专用的局部作用域，称为块。下面的一段代码是块的一个例子：

```
{ int temp;
  temp = list[upper];
  list[upper] = list[lower];
  list[lower] = temp;
}
```

在基于 C 的语言中，块指定为一个复合语句，该语句由一个或多个数据定义开头。在上面的块中，变量 temp 的生存期是程序执行流从进入块到离开块的这段时间。使用这类局部变量的优点是它不会影响在程序其他地方声明的同名变量，更具体地说，是块的引用环境。

块可以使用 10.4 节为实现嵌套子程序而描述的静态链过程来实现。块可以看作无参子程序，它总是在程序的同一个地方被调用。因此，每个块都有活动记录。每次执行块时，都会创建其活动记录的一个实例。

块还可以用另一种更简单高效的方法实现。在程序执行的任何时候，块变量所需要的

最大储存空间都可以静态确定，因为块的进入和退出是严格按照文本顺序进行的。这些储存空间可以分配在活动记录中局部变量的后面。所有块变量的偏移量都可以静态计算，所以块变量可以像局部变量那样处理。

例如下面一段框架程序：

```
void main() {
  int x, y, z;
  while ( ... ) {
    int a, b, c;
    ...
    while (... ) {
      int d, e;
      ...
    }
  }
  while (... ) {
    int f, g;
    ...
  }
  ...
}
```

对于这个程序，静态内存的布局如图 10-10 所示。请注意，变量 f 和 g 占用的内存空间分别与 a 和 b 相同，因为当退出 a 和 b 的块时，a 和 b 就从栈中删除(此时 f 和 g 还未分配内存空间)。

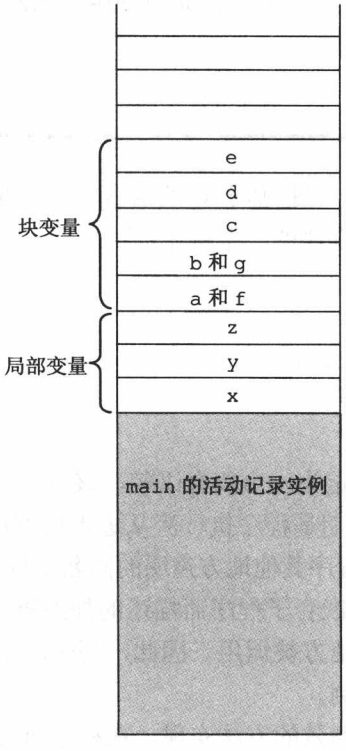


图 10-10 不把块看作无参过程时的块变量存储空间

10.6　动态作用域的实现

至少有两种不同的方法可以在动态作用域的语言中实现局部变量和对它们的非局部引用：深层访问和浅层访问。请注意深层访问和浅层访问的概念与深层绑定和浅层绑定的概念截然不同。它们的一个重要区别在于深层绑定和浅层绑定会导致不同的语义，而深层访问和浅层访问不会。

10.6.1　深层访问

如果局部变量是栈动态的，是动态作用域语言中活动记录的一部分，则非局部变量的引用可以通过搜索目前活跃的其他子程序的活动记录实例来实现，该搜索应该从最近刚激活的子程序开始。这种概念与在支持嵌套子程序的静态作用域语言中访问非局部变量相似，只是使用动态链，而不是静态链。动态链将所有子程序的活动记录实例按子程序触发顺序的倒序连接在一起。因此，在动态作用域语言中，动态链正是引用非局部变量所需要的链。这种方法称为深层访问，因为这种访问可能需要在栈中搜索得很深。

考虑下面的程序框架：

```
void sub3() {
  int x, z;
  x = u + v;
  ...
}

void sub2() {
  int w, x;
  ...
}

void sub1() {
  int v, w;
  ...
}

void main() {
  int v, u;
  ...
}
```

这个程序的语法结构使它看起来像是用基于 C 的语言写的程序，但它不一定是用某种特定语言写成的。假设有如下的函数调用：

```
main 调用 sub1
sub1 调用 sub1
sub1 调用 sub2
sub2 调用 sub3
```

执行上述调用序列后，在函数 sub3 执行过程中，栈的内容如图 10-11 所示。注意活动记录实例没有静态链接。静态链接在动态作用域的语言中没有任何作用。

考虑函数 sub3 中对变量 x、u 和 v 的引用。x 的引用在 sub3 的活动记录实例中。在栈中搜索所有活动记录实例，就会找到 u 的引用，因为只有 main 函数有 u 变量，而这个搜索过程涉及下面 4 个动态链接，检查了 10 个变量名。v 的引用在子程序 sub1 的最新活动记录实例中(最靠近动态链)。

动态作用域语言中用于非局部访问的深层访问方法和静态作用域语言中的静态链方法之间有两个重要的不同点。首先，在动态作用域的语言中，无法在编译时确定必须搜索的链长度。必须搜索链中的每一个活动记录实例，直到找到该变量的第一个实例为止。这是动态作用域语言的执行速度通常比静态作用域语言慢的一个原因。其次，活动记录必须为搜索过程存储变量名，然而在静态作用域语言的实现里，只需要存储变量的值(静态作用域不需要变量名，因为所有的变量都用链偏移量/局部偏移量对来表示)。

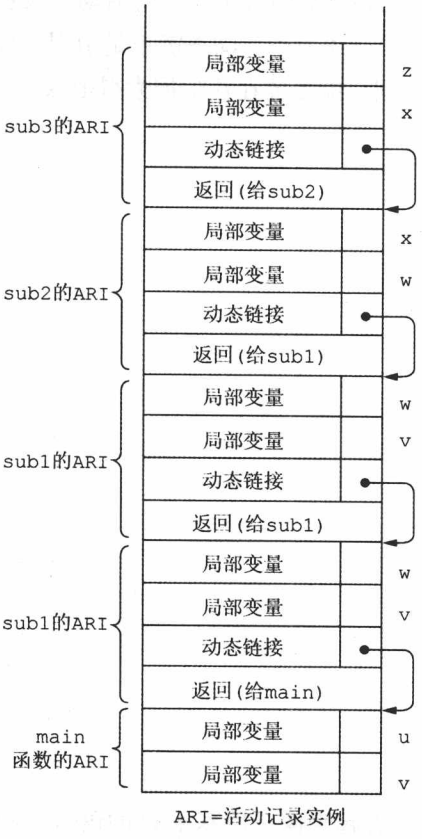


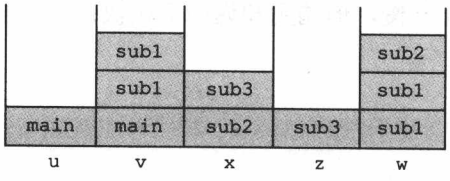
图 10-11 动态作用域程序的栈内容

10.6.2 浅层访问

浅层访问是一种实现方法，而不是一种语义。如前所述，深层访问和浅层访问的语义

是相同的。在浅层访问方法中，子程序中声明的变量不储存在该子程序的活动记录里。由于是在动态作用域中，给定名称的变量在给定的时刻最多只有一个可见，所以可以采用完全不同的方式。浅层访问的一个变体是在完整的程序里，每个变量名称都有一个单独的栈来储存。每次在被调用的子程序开头，通过声明创建某个名字的新变量时，就为这个变量提供栈顶的一个内存单元来存储其名称。对变量名的每个引用都是引用与栈顶那个名称相关的变量，因为只有栈顶的变量才是最近创建的。当子程序终止时，其局部变量的生命期就结束了，这些变量名就从栈中弹出。这种方法允许快速引用变量，但在子程序进入和退出时栈的维护代价却很高。

图 10-12 展示了栈内容和图 10-11 相同的情况下，前面示例程序的变量栈。



栈单元中的名字表示声明变量的程序单元

图 10-12　使用浅层访问实现动态作用域的一种方法

实现浅层访问的另一种方法是用一个中央表，使程序中的每个不同变量名都占有一个位置。在表中的每一项里，用 active 位来表示变量名是当前绑定的，还是和变量相联系的。对任何变量的任何访问都是中央表中的偏移量。这个偏移量是静态的，所以可以快速访问。SNOBOL 实现方案用到了中央表实现技术。

中央表的维护非常简单。调用子程序要求该子程序的所有局部变量在逻辑上都位于中央表上。如果新变量在中央表中的位置已经是激活的——即它包含的变量的生命期还没结束(用 active 位来表示)，则在新变量存在期间，该值必须保存在某个地方。每当变量开始其生命周期时，就在中央表的位置设置 active 位。

中央表的设计方法和被暂时替代的值的储存方法有几个变体。其中一种是用“隐藏”的栈来存储所有已保存的对象。因为子程序会进行调用和返回，于是局部变量的生命期是嵌套的，所以它可以很好地执行。

第二种变体可能是执行起来最干净和最低廉的。它使用一个由单个单元构成的中央表，用不同的名字存储每个变量的当前版本。被替换的变量存储在创建了替换变量的子程序的活动记录中。这是一个栈机制，但是它使用已有的栈，因此新的系统开销最小。

对非局部变量选择深层访问还是浅层访问，取决于子程序调用和非局部引用的相对频率。深层访问方式提供了快速的子程序链接，但是对非局部变量的引用，特别是引用远距离的非局部变量(根据调用链)，开销很大。浅层访问方式可以更加快速地引用非局部变量，特别是远距离的非局部变量。但是子程序链接的系统开销比较高。

小　结

子程序链接的语义在实现时需要很多操作。对于“简单”子程序，这些操作都相对简

单。调用时，必须保存执行状态，参数和返回地址必须传递给被调用的子程序，控制权也必须转移；而返回时，按结果传递的值和按值-结果传递的参数必须返回，如果子程序是一个函数，还必须传回返回值，并将控制权转交回调用方。在支持栈动态局部变量和嵌套子程序的语言中，子程序链接更加复杂。可能有一个以上的活动记录实例，那些实例必须存储在运行栈中，静态和动态链接必须保存在活动记录实例中。静态链接允许在静态作用域的语言中引用非局部变量。

在支持栈动态局部变量和嵌套子程序的语言中，子程序有两个组成部分：静态的实际代码和栈动态的活动记录。活动记录实例包括形参和局部变量。

在动态作用域的语言中，对非局部变量的访问可以使用动态链或者某个中央变量表的方法来实现。动态链虽然访问慢，但调用和返回都很快；中央表方法的访问速度快，但调用和返回都很慢。

复 习 题

1. 本章中“简单”子程序的定义是什么？
2. 调用程序和被调用程序中，哪个存储执行状态信息？
3. 为了链接子程序，必须存储什么？
4. 链接器的任务是什么？
5. 实现带有动态堆栈局部变量的子程序比实现简单子程序困难的两个原因是什么？
6. 活动记录和活动记录实例的区别是什么？
7. 为什么返回地址、动态链接和参数放在活动记录的底部？
8. 哪些类型的机器经常使用寄存器来传递参数？
9. 在支持栈动态局部变量和嵌套子程序的静态作用域的语言中，定位非局部变量的两个步骤是什么？
10. 定义静态链、静态层次、嵌套层次和链偏移量。
11. 环境指针是什么？它的作用是什么？
12. 静态链方法中如何表示变量引用？
13. 举出三个不支持嵌套子程序的常用编程语言。
14. 静态链方法的两个潜在问题是什么？
15. 解释实现块的两种方法。
16. 描述实现动态作用域的深层访问方法。
17. 描述实现动态作用域的浅层访问方法。
18. 在动态作用域语言中对非局部变量的深层访问方法与静态作用域语言中的静态链方法之间的两个不同是什么？
19. 从调用和非局部访问两个方面，比较深层访问方法和浅层访问方法的效率。

习　题

1. 当下面的框架程序执行到位置 1 时，说明栈的所有活动记录实例，包括静态链和动态链。假定 Bigsub 在第 1 级。

```
procedure Bigsub is
 procedure A is
 procedure B is
   begin -- of B
   ... <-------------1
   end; -- of B
 procedure C is
   begin -- of C
   ...
   B;
   ...
   end; -- of C
 begin -- of A
 ...
 C;
 ...
 end; -- of A
begin -- of Bigsub
...
A;
...
end; -- of Bigsub
```

2. 当下面的框架程序执行到位置 1 时，说明栈的所有活动记录实例，包括静态链和动态链。假定 Bigsub 在第 1 级。

```
procedure Bigsub is
 MySum : Float;
 procedure A is
   X : Integer;
 procedure B(Sum : Float) is
   Y, Z : Float;
   begin -- of B
   ...
   C(Z)
   ...
   end; -- of B
 begin -- of A
 ...
 B(X);
 ...
 end; -- of A
procedure C(Plums : Float) is
 begin -- of C
 ... <-------------1
```

```
 end; -- of C
L : Float;
begin -- of Bigsub
...
A;
...
end; -- of Bigsub
```

3. 当下面的框架程序执行到位置 1 时，说明栈的所有活动记录实例，包括静态链和动态链。假定 Bigsub 在第 1 级。

```
procedure Bigsub is
 procedure A(Flag : Boolean) is
  procedure B is
   ...
   A(false);
   end; -- of B
  begin -- of A
  if flag
   then B;
   else C;
  ...
  end; -- of A
 procedure C is
  procedure D is
   ... <-------------1
   end; -- of D
  ...
  D;
  end; -- of C
 begin -- of Bigsub
 ...
 A(true);
 ...
  end; -- of Bigsub
```

当这个程序执行到 D 时，调用的顺序是

```
Bigsub 调用 A
A 调用 B
B 调用 A
A 调用 C
C 调用 D
```

4. 当下面的框架程序执行到位置 1 时，说明栈的所有活动记录实例，包括动态链。这个程序用深层访问的方法来实现动态作用域。

```
void fun1() {
  float a;
  ...
```

```
}

void fun2() {
 int b, c;
 ...
}

void fun3() {
 float d;
 ... <---------- 1
}

void main() {
 char e, f, g;
 ...
}
```

当这个程序执行到 fun3 时，调用的顺序是

```
main 调用 fun2
fun2 调用 fun1
fun1 调用 fun1
fun1 调用 fun3
```

5. 假设题 4 中的程序是运用为每个变量名创建一个栈的浅层访问方法来实现的。说明程序执行到 fun3 时的堆栈。假设程序能够通过题 4 中的调用顺序到达 fun3。
6. 在 Java 方法中，局部变量是在每次活动开始时动态地分配内存的，在什么情况下，某个活动中的局部变量能够保留上一次活动的值？
7. 本章谈到，在动态作用域的语言中，非局部变量用动态链来访问时，变量名必须与变量值一起存储在活动记录中。如果这样做，每个非局部变量的访问就需要一系列费时的名字字符串比较。试设计另一种方法，来加快这些字符串的比较。
8. Pascal 允许给非局部目标变量使用 goto 语句。如果静态链用于非局部变量的访问，这些语句应如何处理？提示：考虑如何为新调用的过程的静态父过程寻找正确的活动记录实例(参见 10.4.2 节)。
9. 在每个活动记录实例中使用两个静态链接，可以稍稍扩展静态链方法，其中第二个链接指向静态祖先的活动记录实例。这个方法如何影响子程序链接和非局部引用所需要的时间？
10. 设计一个程序框架和一个调用序列，生成一个活动记录实例，使其静态链接和动态链接在运行栈中指向不同的活动记录实例。
11. 如果编译器使用静态链方法来实现块，在子程序的活动记录中，哪些条目需要出现在块的活动记录中？
12. 检查 3 个不同体系结构的子程序调用指令，至少包含一个 CISC 机和一个 RISC 机，简要比较它们的功能(这些指令的设计通常至少确定了部分编译器编写者设计子程序链接的方式)。

编程练习

写一个程序，它包含两个子程序，其中一个带一个参数，并对这个参数执行一些简单的操作，另一个带 20 个参数，并使用全部这些参数，但只执行一个简单的操作。主程序必须调用这两个子程序很多次。记录程序的执行时间，输出调用每个子程序的时间。分别在 RISC 和 CISC 机上运行这个程序，并比较两个子程序所需要时间之比。根据比较结果，请对这两种机器上的参数传递速度提出自己的看法。

第11章

抽象数据类型与封装结构

11.1 抽象的概念
11.2 数据抽象简介
11.3 抽象数据类型的设计问题
11.4 语言示例
11.5 参数化的抽象数据类型
11.6 封装结构
11.7 命名封装

本章探索支持数据抽象的编程语言结构。在最近 50 年里关于编程方法学和编程语言设计的诸多新思想中，数据抽象是最深远的思想之一。

首先讨论数据抽象在编程和编程语言中的一般概念。然后通过一个例子来定义和说明数据抽象。接下来描述 Ada、C++、Objective-C、Java、C#和 Ruby 语言对数据抽象的支持。为了说明这些支持数据抽象的语言设计的异同点，分别用 Ada、C++、Objective-C、Java 和 Ruby 实现前面的数据抽象例子。之后，讨论 Ada、C++、Java 5.0 和 C# 2005 创建参数化抽象数据类型的能力。

本章用于演示抽象数据类型的概念和结构的所有语言，都支持面向对象编程。原因是，几乎所有当代语言都支持面向对象编程，而且几乎所有不支持面向对象编程和抽象数据类型的语言都逐渐被淘汰了。

支持抽象数据类型的结构封装了该类型对象的数据和操作。包含多种类型的封装是构建大型程序所必不可少的。本章也讨论这些封装和相关的命名空间问题。

一些编程语句支持逻辑(而不是物理)封装，它也用于封装名字。这些参见 11.7 节。

11.1 抽象的概念

抽象是实体只包含最重要属性的表征。一般而言，进行数据抽象，可以给实体的实例分组，而不需要关注同组内实例的共同属性。例如，假设定义鸟具有如下属性：两只翅膀、两条腿、一个尾巴和一些羽毛。这样，如果说乌鸦是一只鸟，对乌鸦的描述就不需要包括这些属性。知更鸟、麻雀和黄腹吸汁啄木鸟也是这样。在描述不同种类的鸟时，这些共有的特性就可以抽象出来。而对于某个特定的物种，只需要考虑它区别于其他物种的特性。例如，乌鸦的属性有黑色、特定的大小和比较吵闹。乌鸦的描述需要提供这些属性，但不需要提供与其他鸟类相同的属性，这就大大简化了物种成员的描述。对于鸟类而言，需要关注更高层次的细节，而不是物种属性时，可以使用不太抽象的方式。

在编程语言中，抽象是对付编程复杂性的利器，它的目的是简化编程过程。它非常有效，因为它允许程序员忽略次要属性，只关注主要属性。

在目前的编程语言中，两种基本抽象方式是过程抽象和数据抽象。

过程抽象的概念是编程语言设计中最古老的东西之一。Plankalkül 在 20 世纪 40 年代就支持过程抽象。所有的子程序都是过程抽象，因为它们为程序提供了一种方式，用来指定过程，而不必提供过程完成其任务的方式(至少对于调用程序而言是这样)。例如，程序需要对某种类型的数值数组排序时，它通常会调用一个子程序来完成排序工作，在程序中需要排序的地方，会有这样一条语句：

```
sortInt(list, listLen)
```

这个调用是实际排序过程的抽象，排序的算法并没有给定。调用语句与被调用子程序中实现的算法是相互独立的。

在子程序 sortInt 中，基本属性是：待排序的数组名、数组中元素的类型、数组长度、运行结果——调用 sortInt 会得到排好序的数组。sortInt 是用什么算法实现的，用户并不关心。用户只需要知道排序子程序的名字及其调用方式。

数据抽象的广泛使用也必然遵循过程抽象的发展模式。因为每个数据抽象的核心组成部分是操作，而操作定义为过程抽象。

11.2　数据抽象简介

数据抽象的演变开始于 20 世纪 60 年代推出的 COBOL 第一版，它包含了记录数据结构[1]。基于 C 的语言有结构，其实结构也是记录。抽象数据类型是一种数据结构，也采用记录的形式，但它包含操作其数据的子程序。

在语法上，抽象的数据类型是一个包，它包含一种特定数据类型的数据描述，和在该类型上进行相关操作的子程序。通过访问控制，在此包的外部单元使用该数据类型时，可以隐藏不必要的类型细节。使用抽象数据类型的程序单元可以声明该类型的变量，但隐藏了该数据类型的实际表示。抽象数据类型的实例称为对象。

设计数据抽象的动机类似于过程抽象，它是对付复杂性的一个利器，有利于管理复杂的大程序。设计抽象数据类型的其他目的和优势在本节后面讨论。

第 12 章讲述的面向对象技术，是在软件开发中使用数据抽象的产物，数据抽象是它的基本组成部件之一。

11.2.1　抽象数据类型之浮点型

抽象数据类型(至少对于内置类型)，不是一个最近才有的概念。所有的内置数据类型都是抽象数据类型，就连 Fortran I 中的类型也不例外，尽管它们不叫做抽象数据类型。例如浮点数据类型，大部分编程语言都包含至少一种浮点类型。浮点类型提供了为浮点型数据创建变量的方法，以及操作浮点型对象的一组算术操作。

高级语言中的浮点类型使用一个关键概念表达数据抽象：信息隐藏。浮点数据值在存储单元中的实际格式对于用户来说是隐藏的，这种类型的操作仅由语言来提供。除了可以利用内置操作来创建的操作外，用户不允许在该类型的数据上创建新的操作，也不能直接操作浮点值的实际表述方式的各个部分，因为这些部分对用户是隐藏的。这个特征允许程序在特定语言的各种实现之间移植，尽管不同的语言实现可能使用该数据类型的不同表述方式。例如，IEEE 754 浮点标准表述方式制定于 20 世纪 80 年代中期，而在这之前，不同的计算机体系结构使用了数种不同的浮点表示法。虽然如此，这种差异没有阻止使用浮点类型的程序在各种体系结构之间移植。

---

1. 如第 2 章所述，记录数据结构存储了字段，字段有名称，其类型可以互不相同。

11.2.2 用户自定义的抽象数据类型

用户自定义的抽象数据类型应该提供和语言定义的类型(例如浮点类型)相同的特性：①类型定义，允许程序单元声明此类型的变量，而隐藏该类型对象的表示形式；②在该类型对象上的操作集合。

下面在用户定义类型的上下文中正式定义抽象数据类型。抽象数据类型满足下面两个条件：

- 此类型对象的表示方式对于使用它的程序单元来说是隐藏的，因此只有在类型定义中提供的操作才能直接操作这些对象。
- 把类型的声明和在该类型对象上的操作协议(它提供了类型的接口)包含在一个语法单元中。类型接口不依赖对象的表述方式或操作的实现方式，另外，其他程序单元可以创建所定义类型的变量。

信息隐藏有几个优点。一个优点是提高可靠性。使用特定抽象数据类型的程序单元称为该类型的客户程序。客户程序不能有意或无意地直接操作对象的底层表述方式，因此提高了该对象的完整性。对象只能通过所提供的操作来修改。

信息隐藏的另一个优点是，缩小了程序员在读写程序的某部分时必须知道的代码范围和变量数目。变量的值只能由限定范围内的代码来修改，所以代码更容易理解，更容易查找不正确修改的源头。

信息隐藏还减少了名称冲突的可能性，因为变量的作用域更小。

最后，考虑信息隐藏的下述优点：假定栈抽象的最初实现方式使用链表表述方式。后来，由于该表述方式有内存管理问题，栈抽象改为使用连续的表述方式(它在数组中实现栈)。因为使用了数据抽象，可以在定义栈类型的代码上进行这个修改，而不需要在栈抽象的客户程序中进行任何修改。尤其是，示例代码是不需要修改的。当然，若修改了操作的协议，就需要修改客户程序。

尽管抽象数据类型的定义指定，对客户程序而言，对象的数据成员必须是隐藏的，但在许多情况下，客户程序都需要访问执行数据成员。一个常见的解决方法是提供访问器方法，它们有时称为 getter 和 setter，允许客户程序间接访问所谓的隐藏数据——这种方法比简单地将数据声明为 public 更好，因为将数据声明为 public，就可以直接访问它们。访问器方法更好有 3 个原因：

(1) 只有 getter 方法，而没有对应的 setter 方法，就可以提供只读访问。

(2) 可以在 setter 中包含约束。例如，如果数据值应限制为某个范围，就可以用 setter 来强制实现。

(3) 如果 getter 和 setter 是唯一的访问方法，就可以修改数据成员的具体实现方式，而不会影响客户程序。

在抽象数据类型中把数据指定为 public，并为该数据提供访问器方法，是抽象数据类型原则的变体。一些人认为，这些都只是使不完美的设计变得有效的漏洞。如 11.4.6.2 节所述，Ruby 不允许把实例数据声明为 public，但是 Ruby 也很容易创建访问器函数。开发

人员在所有数据都是隐藏的情况下，设计抽象数据类型是比较困难的。

把类型的声明与其操作封装到一个语法单元中的主要优点在于，它可以将一个程序组织成能单独编译的逻辑单元。有时，实现与类型声明包含在一起，有时，它们分别放在不同的语法单元中。将类型与其操作的实现放在不同语法单元中的优点是，提高了程序的模块化，使设计和实现有清晰的分界。如果类型与操作的声明和定义放在同一个语法单元中，就必须用某种方式对客户程序单元隐藏指定定义的单元部分。

11.2.3　示例

栈是广泛应用的一种数据类型，它存储了一些数据元素，只允许在栈顶访问数据元素。假定要为栈构建一个抽象数据类型，它有如下抽象操作：

create(stack)	生成并可能初始化栈对象
destroy(stack)	回收栈的存储空间
empty(stack)	谓词(或 Boolean 类型)函数，当特定的栈为空时返回 true，否则返回 false
push(stack，element)	把特定的元素放入特定的栈
pop(stack)	删除特定栈的栈顶元素
top(stack)	从特定的栈中返回栈顶元素的副本

注意抽象数据类型的某些实现不需要构造操作和析构操作。例如，把一个变量简单地定义为抽象数据类型，会隐式地构造底层的数据结构，并初始化它。该变量的作用域结束后，它们的存储空间会隐式地回收。

栈类型的客户代码片段如下：

```
...
create(stk1);
push(stk1, color1);
push(stk1, color2);
temp = top(stk1);
...
```

11.3　抽象数据类型的设计问题

语言中用于定义抽象数据类型的机制必须提供一个语法单元，来封装类型的声明和子程序的原型，这些子程序在该类型的对象上执行操作。对于抽象类型的客户程序，这些类型声明和子程序原型必须是可见的。这样，客户程序才能声明抽象类型的变量，操作它们的值。尽管类型名必须具有外部可见性，但类型的表示方式必须是隐藏的。类型的表示方式与实现操作的子程序定义皆可放在这个语法单元的内部或外部。

除了在抽象数据类型定义时提供的操作外，很少需要给该类型的对象提供一般的内置

操作。简单地说，广泛适用于各种抽象数据类型的操作并不多，主要有赋值操作、相等和不等比较操作。如果这种语言不支持用户重载赋值操作，则赋值操作就必须包含在抽象类型中。有时应在抽象类型中预定义相等和不等比较操作，而其他情况则不然。例如，如果该抽象类型实现为一个指针，则相等比较意味着指针相同，而设计者希望指针所指向的数据结构是等价的。

很多抽象数据类型都需要某些操作，但是因为这些操作不是通用的，所以常常只能由类型的设计者来提供。这些操作有迭代器、访问器、构造器以及析构器。迭代器在第 8 章已讨论。访问器允许访问客户程序不能直接访问的数据。构造器用于初始化新建实例的部件。在不能隐式回收内存的语言中，析构器经常用来回收抽象数据类型对象中各部件使用的堆存储空间。

如前所述，抽象数据类型包封装了一个数据类型及其操作。包括 C++、Objective-C、Java 和 C#在内的很多现代语言都直接支持抽象数据类型。另一种可行的方法是，提供更一般的封装结构，它可以定义任意数目的实体，而这些实体可以有选择地指定为对外部单元可见。Ada 就用了这种方法。这些包不是抽象数据类型，而是它们的泛化。因此，它们可以用来定义抽象数据类型。尽管本节要讨论 Ada 的封装结构，但将其看作单个数据类型的最小封装。泛化的封装在 11.6 节中讨论。

所以，抽象数据类型的第一个设计问题是用何种容器作为数据类型的接口。第二个设计问题是，抽象数据类型能否参数化。例如，如果某语言支持抽象数据类型的参数化，就可以为某个结构设计一种抽象数据类型，用以存储任意类型的元素。参数化的抽象数据类型在 11.5 节中讨论。第三个设计问题是提供什么样的访问控制，以及如何指定这些访问控制。最后，语言设计者还必须确定类型的规范在物理上是否与其实现分开(或者这是否是开发人员的选择)。

11.4 语言示例

数据抽象的概念最早源于 SIMULA 67 语言，不过它没有为抽象数据类型提供完整的支持，因为它没有隐藏实现细节的方式。本节讨论 Ada、C++、Objective-C、Java、C#和 Ruby 语言对数据抽象的支持。

11.4.1 Ada 中的抽象数据类型

Ada 提供的封装结构可以用来定义一个抽象数据类型，并能隐藏它的表述方式。Ada 83 是最早对抽象数据类型提供完全支持的语言之一。

11.4.1.1 封装

Ada 语言中的封装结构叫做包(package)。包可以拥有两个部分，每部分也称为包。这些称为包规范和正文包。包规范提供这个封装结构的接口(和其他内容)，正文包提供相关

的包规范中所声明实体的大部分(如果说不是全部的话)实现。并非所有的包都有正文部分(只封装了类型与常量的包就没有或不需要正文部分)。

包规范与其相应的正文包公用同一个名字。包头中若出现关键字 body，就意味着它是一个正文包。包规范和正文包可以分别编译，在这种情况下先编译包规范。客户代码也可以在正文包之前编译，甚至在正文包编写之前编译。这意味着，一旦编写了包规范，就可以同时开始编写客户代码和正文包。

11.4.1.2　信息隐藏

Ada 包的设计者可以选择让定义的数据类型对客户程序完全可见，或只提供接口信息。当然，如果类型的表达形式不是隐藏的，所定义的类型就不是抽象数据类型。有两种方法可以在包规范中对客户程序隐藏类型的表达形式。一种方法是在包规范中包含两个部分，其一对客户程序可见，另一个隐藏其内容。对于抽象数据类型，声明放在包规范的可见部分，只提供类型名，而具体的表达形式是隐藏的。包规范中提供表达形式的部分称为私有部分，用保留字 private 声明。私有子句一般放在包规范的结尾处。私有子句对编译器可见，而对客户程序单元是隐藏的。

第二种隐藏表达形式的方法是把抽象数据类型定义成指针，在正文包中提供指针指向的结构定义，正文包中的所有内容对客户程序都是隐藏的。

声明为 private 的类型叫做私有类型。私有数据类型拥有内置的赋值、相等和不等比较操作。任何其他操作都必须在定义该类型的包规范中声明。

类型的表达形式放在包规范中的原因跟编译原理相关。客户程序只能看到包规范，看不到正文包；但是编译器在编译客户程序时，必须能够对导出类型的对象分配存储空间。此外，只有编译了抽象数据类型的包规范，并提供它时，客户程序才能编译。因此，编译器必须能够根据包规范确定对象的大小。于是，类型的表达形式必须对编译器可见，而对客户代码隐藏。这正是包规范中私有子句指定的情形。

私有类型的另一种选择是采用更受限的形式：受限私有类型。非指针的受限私有类型在包规范的私有部分描述，非指针的私有类型也是一样。唯一的语法区别是，受限私有类型在包规范的可见部分声明为 limited private。其语义区别是声明为受限私有类型的对象不支持内置操作。在预定义的赋值和比较操作没有意义或者没有用时，这种类型是很有用的。例如，栈很少用到赋值和比较操作。

11.4.1.3　示例

下面是栈抽象数据类型的包规范：

```
package Stack_Pack is
-- The visible entities, or public interface
  type Stack_Type is limited private;
  Max_Size : constant := 100;
  function Empty(Stk : in Stack_Type) return Boolean;
  procedure Push(Stk : in out Stack_Type;
```

```
                Element : in Integer);
  procedure Pop(Stk : in out Stack_Type);
  function Top(Stk : in Stack_Type) return Integer;
-- The part that is hidden from clients
  private
  type List_Type is array (1..Max_Size) of Integer;
  type Stack_Type is
    record
    List : List_Type;
    Topsub : Integer range 0..Max_Size := 0;
    end record;
  end Stack_Pack;
```

注意这没有包含创建或销毁操作，因为这些不是必要的。

Stack_Pack 的正文包如下：

```
with Ada.Text_IO; use Ada.Text_IO;
package body Stack_Pack is
function Empty(Stk: in Stack_Type) return Boolean is
  begin
  return Stk.Topsub = 0;
  end Empty;

procedure Push(Stk : in out Stack_Type;
    Element : in Integer) is
  begin
  if Stk.Topsub >= Max_Size then
    Put_Line("ERROR - Stack overflow");
  else
    Stk.Topsub := Stk.Topsub + 1;
    Stk.List(Topsub) := Element;
  end if;
end Push;

procedure Pop(Stk : in out Stack_Type) is
  begin
  if Empty(Stk)
    then Put_Line("ERROR - Stack underflow");
    else Stk.Topsub := Stk.Topsub - 1;
  end if;
  end Pop;

function Top(Stk : in Stack_Type) return Integer is
  begin
  if Empty(Stk)
    then Put_Line("ERROR - Stack is empty");
   else return Stk.List(Stk.Topsub);
  end if;
  end Top;
end Stack_Pack;
```

正文包的第一行代码包含两个子句：with 和 use。with 子句使外部包中定义的名字可见，本例中的是 Ada.Text_IO，它提供文本输入输出函数。use 子句可以省略对指定包中实体的明确引用。关于外部包和命名空间的访问问题在 11.7 节中讨论。

正文包中子程序定义的头部必须与对应包规范中的子程序头相匹配。包规范指定，这些子程序应在对应的正文包中定义。

下面的过程 Use_Stacks 是 Stack_Pack 包的一段客户程序。它演示了这个包的用法。

```
with Stack_Pack;
use Stack_Pack;
procedure Use_Stacks is
  Topone : Integer;
  Stack : Stack_Type; -- Creates a Stack_Type object
  begin
  Push(Stack, 42);
  Push(Stack, 17);
  Topone := Top(Stack);
  Pop(Stack);
  ...
  end Use_Stacks;
```

对于众多的现代语言来说，栈是一个傻瓜式的例子，因为对栈的支持都放在它们的标准类库中。不过，栈提供了一个简单的例子，可用于比较本节讨论的语言。

11.4.1.4　评价

Ada 和 Modula-2 都是最早支持抽象数据类型的商业语言[2]。尽管 Ada 的抽象数据类型设计有点复杂、重复，但它显然提供了必需的内容。

11.4.2　C++中的抽象数据类型

C++于 1985 年第一次发布，它是通过将一些特性加入 C 中而创建的。最重要的是增加了对面向对象编程的支持。因为面向对象编程的核心成分之一是抽象数据类型，所以显然 C++需要支持它。

Ada 提供的包能用来模拟抽象数据类型，C++则提供了两个彼此非常类似的结构，更直接地支持抽象数据类型：类和结构体。由于结构体最常用于只包含数据的情形，所以在此不讨论它。

C++中的类是类型。如前所述，Ada 包是更一般化的封装，它可以定义任意数量的类型。可访问某个 Ada 包的程序单元能够通过名字直接访问包中的公共实体。声明类实例的 C++程序单元也可以访问这个类中的任何公共实体，但只能通过该类的实例来访问。这是支持抽象数据类型的更清晰、更直接的方式。

---

2. CLU 语言是一种学术研究语言，而不是商业语言，它最先支持抽象数据类型。

11.4.2.1 封装

C++类中定义的数据称为数据成员，类中定义的函数(方法)称为成员函数。数据成员和成员函数分为两种：类和实例。类的成员与类相连，实例的成员与该类实例相连。本章只讨论类的实例成员。类所有的实例都共享一套成员函数，但每个实例都有自己的一套数据成员。类实例可以是静态变量、栈动态变量或是堆动态变量。如果是静态或栈动态的变量，就直接通过值变量来引用它们。如果是堆动态变量，就通过指针引用它们。栈动态的类实例总是通过对象声明来创建。此外，这种类实例的生命周期截止于其声明语句所在的程序段尾。堆动态的类实例通过 new 操作符创建，用 delete 操作符销毁。堆动态的实例和栈动态的实例都可以拥有指向堆动态数据的指针数据成员，因此就算是栈动态的类实例，也可以有引用堆动态数据的数据成员。

有两种不同的方式来定义类的成员函数：把函数的完整定义放在类中，或仅放在其头部。成员函数的声明和实现体都放在类定义中时，成员函数是隐式内联函数。这意味着，其代码放在调用程序的代码中，而不需要通常的调用和返回链接过程。如果在类定义中只有成员函数的声明，它的完整定义就放在类外，且单独编译。在实时应用程序中，运行效率是最重要的属性，将成员函数设计为内联函数，可以在实时应用程序中节省函数调用的开销。而内联函数设计模式的缺点是使类定义接口变得混乱，从而降低了可读性。

把成员函数的定义放在类定义外，可以把声明和实现相分离，这是现代编程思想的一个常见目标。

11.4.2.2 信息隐藏

C++类可以包含隐藏的实体和可见的实体(这意味着它们对类的客户程序是隐藏的或可见的)。要隐藏的实体放在 private 子句下，可见或公有的实体放在 public 子句下。因此 public 子句描述了类实例的接口[3]。

11.4.2.3 构造函数和析构函数

C++允许用户在类定义中包含构造函数，用于初始化新建对象的数据成员。构造函数也可以为新建对象的指针成员分配堆动态数据。创建类的对象时，会隐式调用构造函数。构造函数与所初始化的类同名。构造函数可以被重载，但是类的每个构造函数当然必须有不同的参数原型。

C++类也可以包含析构函数，它在类实例的生存期结束时隐式调用。如前所述，栈动态的类实例可以包含指向堆动态数据的指针成员。这种实例的析构函数可以在指针成员上使用 delete 操作符，来释放它们引用的堆空间。析构函数通常用作一种调试的手段，用于在释放对象的部分或所有数据成员之前，显示或打印它们的值。析构函数的名称是波浪号(~)后跟类名。

构造函数和析构函数都没有返回类型，也不使用 return 语句。它们都可以显式调用。

---

3. 还有第三种可见性 protected，在第 12 章的继承上下文中讨论。

11.4.2.4　示例

C++抽象数据类型的一个例子仍然是栈：

```
#include <iostream.h>
class Stack {
  private: //** These members are visible only to other
           //** members and friends (see Section 11.6.4)
    int *stackPtr;
    int maxLen;
    int topSub;
  public: //** These members are visible to clients
    Stack() { //** A constructor
      stackPtr = new int [100];
      maxLen = 99;
      topSub = -1;
    }
    ~Stack() {delete [] stackPtr;}; //** A destructor
    void push(int number) {
      if (topSub == maxLen)
        cerr << "Error in push--stack is full\n";
      else stackPtr[++topSub] = number;
    }
    void pop() {
      if (empty())
        cerr << "Error in pop--stack is empty\n";
      else topSub--;
    }
    int top() {
      if (empty())
        cerr << "Error in top--stack is empty\n";
      else
        return (stackPtr[topSub]);
    }
    int empty() {return (topSub == -1);}
}
```

我们只讨论这个类定义的几个方面，因为没有必要弄懂这些代码的所有细节。Stack 类的对象是栈动态变量，但是它包含一个引用堆动态数据的指针。Stack 类有三个数据成员：stackPtr、maxLen 和 topSub，这些成员都是私有的。stackPtr 用来引用堆动态数据，即实现栈的数组。这个类还有四个公有的成员函数 push、pop、top 和 empty，还有一个构造函数和一个析构函数。所有的这些成员函数定义都是在类的里面，但它们也可以在类外部定义。由于这些成员函数的函数体都包含在类中，所以它们都是隐式内联的。构造函数调用 new 操作符从堆里分配包含 100 个 int 元素的数组，还初始化了 maxLen 和 topSub。

下面的示例程序使用了 Stack 抽象数据类型：

```
void main() {
  int topOne;
  Stack stk; //** Create an instance of the Stack class
```

```
    stk.push(42);
    stk.push(17);
    topOne = stk.top();
    stk.pop();
    ...
}
```

下面是仅有成员函数原型的 Stack 类定义。这段代码存储在带有.h 扩展名的头文件中。成员函数的定义跟在类定义的后面，它们使用域解析运算符(::)来表示成员函数所属的类。这些成员函数的定义存储在后缀名为.cpp 的源代码文件中。

```
// Stack.h - the header file for the Stack class
#include <iostream.h>
class Stack {
  private: //** These members are visible only to other
           //** members and friends (see Section 11.6.4)
    int *stackPtr;
    int maxLen;
    int topSub;
  public: //** These members are visible to clients
    Stack(); //** A constructor
    ~Stack(); //** A destructor
    void push(int);
    void pop();
    int top();
    int empty();
}

// Stack.cpp - the implementation file for the Stack class
#include <iostream.h>
#include "Stack.h"
using std::cout;
Stack::Stack() { //** A constructor
   stackPtr = new int [100];
   maxLen = 99;
   topSub = -1;
}

Stack::~Stack() {delete [] stackPtr;}; //** A destructor

void Stack::push(int number) {
  if (topSub == maxLen)
    cerr << "Error in push--stack is full\n";
  else stackPtr[++topSub] = number;
}
void Stack::pop() {
  if (topSub == -1)
    cerr << "Error in pop--stack is empty\n";
  else topSub--;
}
```

```
int top() {
      if (topSub == -1)
        cerr << "Error in top--stack is empty\n";
       else
        return (stackPtr[topSub]);
    }
int Stack::empty() {return (topSub == -1);}
```

11.4.2.5　评价

C++通过类结构支持抽象数据类型，Ada 通过它的包来支持，两者的表达能力是相似的。它们都提供有效的机制来实现封装和抽象数据类型的信息隐藏。主要的区别是 C++的类是类型，而 Ada 包是更一般的封装。而且，Ada 的包结构不仅仅用于数据抽象，参见第 12 章。

11.4.3　Objective-C 中的抽象数据类型

如前所述，Objective-C 类似于 C++，因为它最初的设计是 C 语言，并带有支持面向对象编程的扩展。两者的一个基本区别是，Objective-C 的方法调用使用 Smalltalk 的语法。

11.4.3.1　封装

Objective-C 类的接口部分在 interface 容器中定义，其一般语法如下：

```
@interface 类名: 父类 {
实例变量声明
}
方法原型
@end
```

第一行和最后一行以符号@开头，是指令。

自然，类的实现代码封装在 implementation 容器中，其语法如下：

```
@implementation 类名
方法定义
@end
```

与 C++一样，在 Objective-C 中，类就是类型。

方法原型的语法如下：

```
(+ | -)(返回类型) 方法名 [: (形参)];
```

如果有加号，就表示该方法是一个类方法，而减号表示它是一个实例方法。形参外部的方括号表示它们是可选的。若没有参数，就不写圆括号和冒号。与大多数其他支持面向对象编程的语言一样，Objective-C 类的所有实例都拥有其实例方法的一个副本，但每个实例都有自己的实例数据副本。

形参列表的语法形式不同于更常见的语言 C、C++、Java 和 C#。如果有一个参数，其

类型就在参数名前面的圆括号中指定，如下面的方法原型所示：

```
-(void) meth1: (int) x;
```

这个方法的名称是 meth1: (注意冒号)。带两个参数的方法可以采用如下方法原型：

```
-(int) meth2: (int) x second: (float) y;
```

在这个例子中，这个方法的名称是 meth2: second，显然这是一个很糟糕的方法名。该名称中的最后一部分(second)可以省略，如下所示：

```
-(int) meth2: (int) x: (float) y;
```

在这个例子中，方法名是 meth2::。

方法定义类似于方法原型，但方法定义用以花括号分隔开的语句系列替代了方法原型中的冒号。

无参数方法的调用语法如下：

```
[对象名 方法名];
```

如果方法带一个参数，就把一个冒号放在方法名的后面，之后是参数。在方法名和参数之间没有其他标点符号。例如，在对象 myAdder 上调用带一个参数 7 的方法 add1，语句如下所示：

```
[myAdder add1: 7];
```

如果方法带两个参数，其名称只有第一部分，则第一个参数的后面要加上冒号，冒号后面是第二个参数。两个参数之间不使用任何标点符号。如果方法有多个参数，就重复这个模式。例如，如果 add1 带 3 个参数，且其名称没有其他部分，则调用它的语句是：

```
[myAdder add1: 7: 5: 3];
```

方法可以有多个参数，方法名也可以有多个部分，如下所示：

```
-(int) meth2: (int) x second: (float) y;
```

调用这个方法的示例如下：

```
[myObject meth2: 7 second: 3.2];
```

构造函数在 Objective-C 中称为初始化器，它们只提供初始值，其名称可以是任意的，因此必须显式调用。构造函数返回新对象的引用，所以它们的类型总是指向类名的指针。它们使用 return 语句来返回 self，即当前对象的引用。

在 Objective-C 中，调用 alloc 就会创建对象。一般情况下，调用 alloc 后，就显式调用类的构造函数。这两个调用可以是级联的，也常常是级联的，如下所示，该示例用 alloc 创建了 Adder 类的一个对象，再在新对象上调用其构造函数 init，把新对象的地址放在 myAdder 中：

```
Adder *myAdder = [[Adder alloc]init];
```

所有的类实例都是堆动态的，通过引用变量来引用。

C 程序几乎总是为输入/输出函数导入一个头文件 stdio.h。在 Objective-C 中，导入的头文件通常包含各种常用函数的原型，包括输入/输出函数和一些必要的数据，这通过下面的语句来完成：

```
#import <Foundation/Foundation.h>
```

导入 Foundation.h 头文件，会给程序创建一些数据。所以，主函数首先应初始化一个存储池，给这些数据分配内存。这通过下面的语句来完成：

```
NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
```

在主程序的 return 语句之前，调用 pool 对象的 drain 方法释放该存储池，如下所示：

```
[pool drain];
```

11.4.3.2　信息隐藏

在类定义中，Objective-C 使用指令@private 和@public 来指定实例变量的访问级别。这些指令与 C++中的保留字 private 和 public 作用相同，区别是在 Objective-C 中，default 表示受保护的，而在 C++中是私有的。与支持面向对象编程的大多数编程语言不同，在 Objective-C 中，无法限制对方法的访问。

在 Objective-C 中，根据约定，实例变量的获取器方法名就是变量的名称，设置器方法的名称是单词 set 再加上该变量名的首字母大写形式。因此，对于变量 sum，获取器方法名就是 sum，而设置器方法名是 setSum。假定 sum 是一个 int 变量，则这些方法的定义如下：

```
// The getter for sum
-(int) sum {
  return sum;
}

// The setter for sum
-(void) setSum: (int) s {
sum = s;
}
```

如果某变量的获取器方法和设置器方法没有任何操作上的限制，Objective-C 编译器就可以自动生成它们。为此，需要在接口部分的 property 指令上列出要生成获取器方法和设置器方法的实例变量，如下所示：

```
@property int sum;
```

在实现部分，变量列在 synthesize 指令上，如下所示：

```
@synthesize sum;
```

编译器为之生成获取器和设置器的变量常常称为属性，访问器方法称为要合成的方法。

实例变量的获取器和设置器可以用两种方式使用：在方法调用中或在点记号中使用，就好像它们可以公开访问一样。例如，如果为变量 sum 定义了获取器和设置器，就可以按如下方式使用它们：

```
[myObject setSum: 100];
newSum = [myObject sum];
```

或者，就好像它们可以公开访问，如下所示：

```
myObject.sum = 100;
newSum = myObject.sum;
```

11.4.3.3 示例

下面在 Objective-C 中定义了 stack 类的接口和实现代码：

```
// stack.m - interface and implementation of a simple stack

#import <Foundation/Foundation.h>

// Interface section

@interface Stack: NSObject {
  int stackArray [100];
  int stackPtr;
  int maxLen;
  int topSub;
}
  -(void) push: (int) number;
  -(void) pop;
  -(int) top;
  -(int) empty;
@end

// Implementation section

@implementation Stack
  -(Stack *) initWith {
    maxLen = 100;
   topSub = -1;
   stackPtr = stackArray;
   return self;
}

-(void) push: (int) number {
  if (topSub == maxLen)
    NSLog(@"Error in push--stack is full");
  else
    stackPtr[++topSub] = number;
```

```
}

-(void) pop {
  if (topSub == -1)
    NSLog(@"Error in pop--stack is empty");
  else
     topSub--;
}

-(int) top {
  if (topSub >= 0)
    return stackPtr[topSub]);
  else
    NSLog(@"Error in top--stack is empty");
}

-(int) empty {
   return topSub == -1);
}

int main (int argc, char *argv[]) {
  int temp;
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc]
     init];
  Stack *myStack = [[Stack alloc]initWith];
  [myStack push: 5];
  [myStack push: 3];
  temp = [myStack top];
  NSLog(@"Top element is:%i", temp);
  [myStack pop];
  temp = [myStack top];
  NSLog(@"Top element is:%i", temp);
  temp = [myStack top];
  [myStack pop];
  [myStack release];
  [pool drain];
   return 0;
  }
@end
```

这个程序的输出如下：

```
Top element is: 3
Top element is: 5
Error in top--stack is empty
Error in pop--stack is empty
```

Objective-C 程序的屏幕输出是通过调用一个名称古怪的方法 NSLog 而创建的，它把一个字面量字符串作为参数。字面量字符串用符号@后跟一个带引号的字符串来创建。如果输出字符串包含变量的值，就要在 NSLog 的调用中把变量名包含为参数。变量值在字面

量字符串中的位置用格式码来标记，例如%i 表示整数，%f 表示科学计数法的浮点数，类似于 C 的 printf 函数。

11.4.3.4 评价

Objective-C 对抽象数据类型的支持是很充分的。一些人觉得，它使用了两种完全不同的语言中的语法形式：Smalltalk 语法用于方法调用，C 语法用于几乎所有其他操作。这有点古怪。另外，它使用指令来表示类接口和实现部分，而不使用语言结构，这也不同于大多数其他编程语言。一个小缺陷是，它无法限制对方法的访问。所以，即使只能在类的内部使用的方法也可以由客户程序访问。另一个小缺陷是构造函数必须显式调用，因此需要程序员记得调用它们，这也导致客户程序比较混乱。

11.4.4 Java 中的抽象数据类型

Java 支持抽象数据类型的方式与 C++相似。但是，这两者还是有一些重要的不同之处。在 Java 中，所有的对象都是在堆中分配空间，并通过引用变量来访问。Java 中的方法必须完全在类中定义。方法体必须与和它对应的方法头放在一起[4]。因此，Java 的抽象数据类型在一个语法单元中声明和定义。Java 编译器可以内联任何未重写的方法。将定义声明为私有，则它就可以使定义对客户程序是隐藏的。

在 Java 中，访问修饰符可以关联到方法和变量的定义上，而不是在类的定义中包含公共和私有子句。如果实例变量或方法没有访问修饰符，则它就具有包访问权，参见 11.7.2 节。

11.4.4.1 示例

下面是 stack 例子的 Java 类定义：

```
class StackClass {
  private int [] stackRef;
  private int maxLen,
              topIndex;
  public StackClass() { // A constructor
    stackRef = new int [100];
    maxLen = 99;
    topIndex = -1;
  }
  public void push(int number) {
    if (topIndex == maxLen)
      System.out.println("Error in push—stack is full");
    else stackRef[++topIndex] = number;
  }
  public void pop() {
    if (empty())
      System.out.println("Error in pop—stack is empty");
```

4. Java 接口是这个规则的例外，接口包含方法头，但不能包含方法体。

```
    else --topIndex;
  }
  public int top() {
    if (empty()) {
      System.out.println("Error in top—stack is empty");
      return 9999;
    }
    else
      return (stackRef[topIndex]);
  }
  public boolean empty() {return (topIndex == -1);}
}
```

以下是使用 StackClass 的示例：

```
public class TstStack {
  public static void main(String[] args) {
    StackClass myStack = new StackClass();
    myStack.push(42);
    myStack.push(29);
    System.out.println("29 is: " + myStack.top());
    myStack.pop();
    System.out.println("42 is: " + myStack.top());
    myStack.pop();
    myStack.pop(); // Produces an error message
  }
}
```

一个明显的不同之处是 Java 版本中缺少了析构函数，因为 Java 采用隐式的垃圾回收机制[5]。

11.4.4.2　评价

C++和 Java 中对抽象数据类型支持是类似的，但在一些修饰方式上有区别。显然，Java 提供了设计抽象数据类型所需的内容。

11.4.5　C#中的抽象数据类型

C#是以 C++和 Java 为基础的，也引入了一些新的结构。与 Java 一样，所有的 C#类实例都是在堆上分配空间的。所有类都有一个预定义的构造函数，用来为实例数据提供初始值。这些构造函数为变量提供了典型的初始值，如将 int 类型初始化为 0，将 boolean 类型初始化为 false。用户可以为自己定义的类提供一个或多个构造函数，这样的构造函数可以对类的全部或部分实例数据提供初值。没有在用户定义的构造函数中初始化的任何实例变量都将由默认构造函数赋值。

尽管 C#允许定义析构函数，但因为 C#对绝大多数堆对象使用垃圾回收机制，所以很

5. 在 Java 中，finalize 方法用作析构函数。

少用到析构函数。

11.4.5.1 封装

如 11.4.2 节所述，C++包含类和结构，它们是几乎相同的构造。唯一的区别在于：类的默认访问修饰符是 private，而结构的默认访问修饰符是 public。C#也包含了结构，但和 C++中的结构大不相同。在 C#中，结构在某种意义上是轻量级的类。它们可以拥有构造函数、属性、方法和数据域，也可以实现接口，但不支持继承。C#中结构和类之间的另一个重要区别在于，结构是值类型，而不是引用类型。它们在运行时栈中分配存储空间，而不是堆。如果它们传递为参数，则与其他值类型一样，默认为按值传递。所有的 C#值类型，包括它所有的基本类型，实际上都是结构。结构可以通过声明来创建，与其他预定义的值类型相同，例如 int 或 float，它们还可以用关键字 new 来创建，new 调用构造函数来初始化它们。

在 C#中，结构主要用于实现相对小的简单类型，它们不需要用作基类，以进行继承。对象在栈上分配空间比在堆上分配空间更为方便时，也会使用结构类型。

11.4.5.2 信息隐藏

C#使用 private 和 protected 访问修饰符的方式与 Java 完全相同。

C#提供了属性，这是从 Delphi 中借鉴过来的，属性在不需要客户程序显式调用方法的情况下，实现 getter 和 setter 的方式。与 Objective-C 一样，属性可以隐式地访问特定的私有实例数据。例如下面简单的类和客户程序：

```
public class Weather {
  public int DegreeDays { //** DegreeDays is a property
    get {
      return degreeDays;
    }
    set {
      if(value < 0 || value > 30)
        Console.WriteLine(
             "Value is out of range: {0}", value);
      else
        degreeDays = value;
    }
  }
  private int degreeDays;
  ...
  }
...
Weather w = new Weather();
int degreeDaysToday, oldDegreeDays;
...
w.DegreeDays = degreeDaysToday;
...
```

```
oldDegreeDays = w.DegreeDays;
```

在类 Weather 中，定义了属性 DegreeDays。这个成员提供了访问私有数据成员 degreeDays 的 getter 和 setter 方法。在类定义之后的客户程序中，尽管 degreeDays 似乎看作一个公有成员变量，但是对它的访问只能通过属性来进行。注意在 setter 方法中使用了隐式变量 value。新的属性值正是通过这种机制来引用的。

这里没有列出栈示例的 C#版本。11.4.4.1 节中的 Java 版本与 C#版本的唯一区别是输出方法调用，以及 empty 方法的返回类型使用了 bool，而不是 boolean。

11.4.6　Ruby 中的抽象数据类型

Ruby 通过它的类对抽象数据类型提供了支持。在功能方面，Ruby 的类与 C++和 Java 中的类相似。

11.4.6.1　封装

在 Ruby 中，类是用关键字 class 开头、end 结尾的复合语句定义的。在语法形式上，实例变量的名字必须以@符号开始。实例方法与 Ruby 中的函数拥有相同的语法：它们以保留字 def 开始，以 end 结束。类函数在它们的名字前加上类名和一个句点分隔符，来和实例方法相区别。例如，在 Stack 类中，类方法名应该以 Stack 开头。在 Ruby 中构造函数名为 initialize。因为构造函数不能重载，所以每个类只有一个构造函数。

Ruby 中的类是动态的，因为可以随时给它添加成员。为此，只需要包含指定新成员的其他类定义即可。而且，甚至是语言中的预定义类(例如 String)也可以扩展。例如下面的类定义：

```
class myClass
  def meth1
    ...
  end
end
```

用第二个类定义添加第二个方法 meth2，就可以扩展这个类：

```
class myClass
  def meth2
    ...
  end
end
```

方法也可以从类中删除。为此，需要提供另一个类定义，在其中把要删除的方法作为参数发送给 remove_method 方法。Ruby 的动态类是语言设计者用可读性(也意味着可靠性)换取灵活性的另一个例子。允许动态修改类，显然提高了语言的灵活性，但会伤害可读性。为了在程序的特定位置确定类的行为，必须在程序中找到该类的所有定义，把它们都考虑在内。

11.4.6.2 信息隐藏

Ruby 中方法的访问控制是动态的，所以只能在执行过程中检测到违反访问控制的情况。默认的方法访问方式是公开访问，但它也可以是受保护的或私有的。指定访问控制有两种方式 private 和 public，它们都使用同名的函数作为访问级别。一种方式是调用无参数的函数，这会将类中以后定义的方法重置为默认访问。例如：

```
class MyClass
  def meth1
  ...
  end
  ...
private
  def meth7
  ...
  end
  ...
end # of class MyClass
```

另一种方式是把指定方法的名称作为参数，来调用访问控制函数。例如，下面的代码在语义上等同于前面的类定义：

```
class MyClass
  def meth1
  ...
  end
  ...
  def meth7
  ...
  end
  private :meth7, ...
    end # of class MyClass
```

在 Ruby 中，类的所有数据成员都是私有的，且不能改变。所以数据成员只能通过类的方法来访问，其中一些方法可能是访问器方法。在 Ruby 中，可通过访问器方法访问的实例数据称为属性。

假设有一个实例变量@sum，其 getter 和 setter 方法如下：

```
def sum
  @sum
end
def sum=(new_sum)
  @sum = new_sum
end
```

注意，getter 的名称是没有@的实例变量名，setter 方法的名字与对应的 getter 相同，只是还附加了一个等号(=)。

在类定义中，getter 和 setter 可以由 Ruby 系统分别调用 attr_reader 和 attr_writer 来隐式

生成。它们的参数是属性名的符号，如下所示：

```
attr_reader :sum, :total
attr_writer :sum
```

11.4.6.3　示例

下面是用 Ruby 实现的 stack 示例：

```
# Stack.rb - defines and tests a stack of maximum length
#            100, implemented in an array
class StackClass

# Constructor
  def initialize
    @stackRef = Array.new(100)
    @maxLen = 100
   @topIndex = -1
  end

# push method
  def push(number)
    if @topIndex == @maxLen
      puts "Error in push - stack is full"
    else
      @topIndex = @topIndex + 1
      @stackRef[@topIndex] = number
    end
  end

# pop method
  def pop
    if empty
      puts "Error in pop - stack is empty"
    else
      @topIndex = @topIndex - 1
    end
  end

# top method
  def top
    if empty
      puts "Error in top - stack is empty"
    else
      @stackRef[@topIndex]
    end
  end

 # empty method
   def empty
     @topIndex == -1
```

```
    end
end # of Stack class

# Test code for StackClass
myStack = StackClass.new
myStack.push(42)
myStack.push(29)
puts "Top element is (should be 29): #{myStack.top}"
myStack.pop
puts "Top element is (should be 42): #{myStack.top}"
myStack.pop

# The following pop should produce an
#  error message - stack is empty
myStack.pop
```

标记#{变量}将变量的值转换为字符串，并插入到字符串中。这个类定义了一个可以储存任何类型对象的 stack 结构。

11.4.6.4 评价

在 Ruby 中，一切都是对象，数组实际上是一系列对象的引用。这显然使这个 stack 比 Ada、C++和 Java 中的相似示例更加灵活。此外，简单地给构造函数传递所需的最大长度，类的对象就可以拥有给定的最大长度。当然，因为在 Ruby 中数组的长度是动态的，所以可以修改这个类，实现不限定长度的 stack 对象，唯一的限制是计算机的内存容量。因为类和实例变量名的形式不同，所以 Ruby 的可读性略优于本节讨论的其他语言。

11.5 参数化的抽象数据类型

参数化抽象数据类型常常是非常方便的。例如，可以设计出一个栈抽象数据类型，来存储任何标量类型的元素，而不需要为每种不同的标量类型设计不同的栈抽象。注意，这只是静态类型语言的一个问题。在动态类型语言中，如 Ruby，任何堆栈都可以存储任何类型的元素。实际上，栈的不同元素可以是不同的类型。以下 4 个小节将讨论在 Ada、C++、Java 5.0 和 C# 2005 中如何构建参数化的抽象数据类型。

11.5.1 Ada

第 9 章讨论、演示了 Ada 中的泛型过程。包也可以是泛化的，因此可以构造泛型的或参数化的抽象数据类型。

11.4.1 节中的 Ada 栈抽象数据类型示例有两个限制：①这种类型的栈只能存储整数类型的元素；②这种栈最多能有 100 个元素。这两个限制可以使用泛型包来消除，此泛型包可实例化为其他元素类型和任意大小(这是一个泛型实例化，完全不同于实例化类而创建对象)。以下包规范描述了有这些功能的泛型栈抽象数据类型的接口：

```
generic
  Max_Size : Positive; -- A generic parameter for stack
                       -- size
  type Element_Type is private; -- A generic parameter
                       -- for element type
package Generic_Stack is
-- The visible entities, or public interface
  type Stack_Type is limited private;
  function Empty(Stk : in Stack_Type) return Boolean;
  procedure Push(Stk : in out Stack_Type;
                 Element : in Element_Type);
  procedure Pop(Stk : in out Stack_Type);
  function Top(Stk : in Stack_Type) return Element_Type;
-- The hidden part
private
  type List_Type is array (1..Max_Size) of Element_Type;
  type Stack_Type is
    record
    List : List_Type;
    Topsub : Integer range 0..Max_Size := 0;
    end record;
  end Generic_Stack;
```

Generic_Stack 的正文包与 11.4.1.3 节中 Stack_Pack 正文包相同，但 Push 和 Top 中的 Element 形参是 Element_Type 类型，而不是 Integer 类型。

下面的语句把 Generic_Stack 实例化为一个有 100 个整数类型元素的栈：

```
package Integer_Stack is new Generic_Stack(100, Integer);
```

也可以为包含 500 个 Float 元素的栈建立一个抽象的数据类型，如下：

```
package Float_Stack is new Generic_Stack(500, Float);
```

在编译时，实例化的结果是为 Generic_Stack 包建立两个不同的代码版本。

11.5.2　C++

C++也支持参数化的抽象数据类型。为了把 11.4.2.4 节的 C++栈类示例泛化为适用于任意栈大小，只需要改变构造函数，如下所示：

```
Stack(int size) {
  stackPtr = new int [size];
  maxLen = size - 1;
  topSub = -1;
}
```

栈对象的声明可以如下：

```
Stack stk(150);
```

Stack 类的定义中可以包含这两个构造函数，这样用户可以使用默认大小的栈或指定其大小。

使类模板化，就可以泛化栈的元素类型。于是，元素类型可以是模板参数。栈类型的模板类的定义如下：

```
#include <iostream.h>
template <typename Type> // Type is the template parameter
class Stack {
  private:
    Type *stackPtr;
    int maxLen;
    int topSub;
  public:
// A constructor for 100 element stacks
    Stack() {
      stackPtr = new Type [100];
      maxLen = 99;
      topSub = -1;
    }
// A constructor for a given number of elements
    Stack(int size) {
      stackPtr = new Type [size];
      maxLen = size - 1;
      topSub = -1;
    }
    ~Stack() {delete stackPtr;}; // A destructor
    void push(Type number) {
      if (topSub == maxLen)
        cout << "Error in push-stack is full\n";
      else stackPtr[++ topSub] = number;
    }
    void pop() {
      if (empty())
        cout << "Error in pop-stack is empty\n";
      else topSub --;
    }
    Type top() {
      if (empty())
        cerr << "Error in top--stack is empty\n";
      else
        return (stackPtr[topSub]);
    }
    int empty() {return (topSub == -1);}
}
```

与 Ada 一样，C++的模板类在编译时实例化为类型化的类。例如，下面的声明可以创建 Stack 模板类的一个实例和类型化的类实例：

```
Stack<int> myIntStack;
```

但是，如果已经为 int 类型创建了 Stack 模板类的一个实例，就不需要创建类型化的类。

11.5.3　Java 5.0

Java 5.0 支持一种参数化的抽象数据类型，其中泛型参数必须是类，第 9 章曾经简要讨论过它。

最常见的泛型类型是集合类型，如 LinkedList 和 ArrayList，它们都是 Java 类库在不支持泛型之前添加的。最初的集合类型存储 Object 类的对象，于是它们能存储任意对象(但不是基本类型)。因此，集合类型总是能存储多种类型(只要它们是类)。这有三个问题：首先，每次从集合中删除一个对象时，对象必须转换成正确的类型。其次，在元素添加到集合中时，没有错误检查机制。这意味着，一旦集合创建好，任意类的对象都能添加到集合中，即使该集合只希望存储 Ingeter 对象。第三，集合类型不能存储基本类型。因此，要在 ArrayList 中存储 int 值，必须把该数值放到一个 Integer 类的实例中。例如下面的代码：

```
//* Create an ArrayList object
ArrayList myArray = new ArrayList();
//* Create an element
myArray.add(0, new Integer(47));
//* Get first object
Integer myInt = (Integer)myArray.get(0);
```

在 Java 5.0 中，集合类(最常用的是 ArrayList)成了泛型类。这些泛型类的实例化需要在类的构造函数上调用 new，并通过尖括号给它传递泛型参数。例如，下面的语句可以实例化 ArrayList 类，以存储 Integer 对象：

```
ArrayList <Integer> myArray = new ArrayList <Integer>();
```

这个新类克服了 Java 5.0 之前集合类型存在的两个问题。这里只有 Integer 对象才能放入集合 myArray 中。而且，从集合中删除对象时，也不必进行类型转换。

Java 5.0 还给列表、队列和集等集合提供了接口。

用户也可以在 Java 5.0 中定义泛型类。例如如下语句：

```
public class MyClass<T> {
  ...
}
```

这个类也可以用如下语句实例化：

```
MyClass<String> myString;
```

这些用户定义的泛型类有一些缺点。其一，它们不能存储基本类型。其二，其元素不能索引。必须使用 add 方法才能把元素添加到用户定义的泛型集合中。下面使用 ArrayList 实现泛型栈示例。注意，使用 size 方法可以找到 ArrayList 的最后一个元素，该方法返回结构中的元素个数。使用 remove 方法从结构中删除元素。下面是泛型类：

```
import java.util.*;
public class Stack2<T> {

  private ArrayList<T> stackRef;
  private int maxLen;
  public Stack2() { // A constructor
    stackRef = new ArrayList<T> ();
    maxLen = 99;
  }
  public void push(T newValue) {
    if (stackRef.size() == maxLen)
      System.out.println("Error in push-stack is full");
    else
      stackRef.add(newValue);
}
public void pop() {
  if (empty())
    System.out.println("Error in pop-stack is empty");
   else
    stackRef.remove(stackRef.size() - 1);
  }
  public T top() {
    if (empty()) {
      System.out.println("Error in top-stack is empty");
      return null;
    }
    else
      return (stackRef.get(stackRef.size() - 1));
 }
 public boolean empty() {return (stackRef.isEmpty());}
```

这个类可以用下面的语句实例化为 String 类型：

```
Stack2<String> myStack = new Stack2<String>();
```

第 9 章提到，Java 5.0 支持通配符类型。例如，Collection<?>是所有集合类的通配类。这允许方法把任何集合类型作为参数。由于集合本身可以是泛型类，因此 Collection<?>类在某种意义上是一般化的泛型类。

要注意通配类型的对象。例如，因为这种类型的对象是有类型的，所以其他类型的对象就不能加入此对象。比如

```
Collection<?> c = new ArrayList<String>();
```

用 add 方法把非 String 类型的变量加入这个集合是非法的。

很容易在 Java 5.0 中定义泛型类，该类仅用于有限的类型。例如，类可以声明泛型类型的变量，通过该变量调用方法，例如 compareTo。如果该类用于实例化不包含 compareTo 方法的类型，就不能使用这个类。为了防止泛型类用于实例化不支持 compareTo 方法的类型，可以用下面的泛型参数来定义：

```
<T extends Comparable>
```

Comparable 是声明 compareTo 的接口。如果这个泛型类型在类定义中使用，该类就不能用于实例化没有实现 Comparable 接口的类型。这里选择使用 extends 保留字似乎很古怪，但使用它与子类型的概念有关。显然，Java 的设计者不希望给该语言添加另一个更含蓄的保留字。

11.5.4　C# 2005

像 Java 一样，第一版 C#定义的集合类可以存储任何类的对象，它们是 ArrayList、Stack 和 Queue。这些类有过与 Java 5.0 以前版本中集合类一样的问题。

泛型类加进了 2005 版本的 C#。5 个预定义的泛型集合为 Array、List、Stack、Queue 和 Dictionary(Dictionary 类实现了哈希功能)。与 Java 5.0 一样，这些类消除了在集合里出现类型混杂的问题和删除元素时需要转换类型的问题。

像 Java 5.0 一样，用户在 C# 2005 中可以定义泛型类。用户定义的 C#泛型集合的一个功能是通过下标来访问它们的元素。尽管下标一般是整数类型，但也可以用字符串作为下标。

Java 5.0 支持的通配符类型，在 C# 2005 中不支持。

11.6　封装结构

本章的前 5 节讨论了抽象数据类型，它们都是最小化封装[6]。本节将讨论大型程序需要的多类型封装。

11.6.1　引言

程序的代码超出了几千行时，就会产生两个实际问题。从程序员的角度来看，一个问题是，子程序或者抽象数据类型定义的单个集合，不会使程序组织合理、便于管理。大程序的第二个实际问题是重编译。对于相当小的程序，每次修改后重新编译整个程序的成本并不高。但对于大型程序，重新编译的成本是巨大的。因此，显然需要设法避免重新编译程序中没有修改的部分。解决上述两个问题的方法是将程序组织为逻辑相关的代码和数据集合，每个逻辑单元可以单独编译。封装即是这样一个集合。

封装通常会放在库中，可在其他程序中重用，而不需要再次编写它们。至少在过去 50 年里，人们一直在写超过几千行代码的程序，因此提供封装的技术已演变了一段时间。

在支持嵌套子程序的语言中，程序在组织时，可以把子程序定义嵌套在使用它们的、逻辑上更大的子程序中。这可以在 Ada、Fortran 95、Python 和 Ruby 中实现。但如第 5 章所述，这种组织程序的方法使用了静态作用域，所以非常不理想。因此，甚至在支持嵌套子程序的语言中，它也不是组织封装结构的主要方法。

---

6. 在 Ada 中，包封装可以用于单个类型，也可以用于多个类型。

11.6.2 C 中的封装

尽管 C 并不提供抽象数据类型的完全支持，但抽象数据类型和多类型封装在 C 中都可模拟实现。

在 C 中，可以将一组相关的函数和数据定义放在一个文件中，该文件可以独立编译。这个文件用作库，通过它的实体来实现。该文件的接口包括数据、类型和函数声明，放在另一个文件中，称为头文件。在头文件中，将类型声明为指向结构体的指针，就可以将结构体类型的表达方式隐藏起来。这种结构类型的完整定义只需要出现在实现文件中。这种方法的缺点与 Ada 包中使用指针作为抽象数据类型相同：指针有其固有的问题，指针的赋值和比较操作也有潜在的问题。

源代码形式的头文件和实现文件的编译版本是向客户提供的。使用这样的库时，通过预处理指令#include 把头文件包含在客户代码中，以便在客户代码中检查函数引用和数据的类型。#include 语句也说明，客户程序依赖这个库的实现文件。这种方法有效地把封装的说明和实现相分离。

虽然这种封装是有效的，但是它引入了一些不安全因素。例如，用户可以把头文件中的定义剪切并粘贴到客户程序中，而不使用#include。这是可行的，因为#include 命令只是把头文件的内容复制到使用该语句的文件中。但这个方法有两个问题。第一，客户程序对库的依赖性说明没有了(当然头文件也不存在了)。第二，库的作者可能更改了头文件和实现文件，而客户程序可能尝试使用新的实现文件(不知道实现文件已修改)和旧的头文件，因为用户已经把头文件复制到客户程序中。例如，在旧的头文件中，变量 x 定义为 int 类型，客户程序仍使用它，而实现代码用新的头文件编译，新的头文件把 x 定义为 float，所以实现代码把 x 当做 int 类型编译，而客户程序把 x 当做 float 类型编译。在这种情况下连接器无法检测出错误。

因此，用户必须确保头文件和实现文件是最新版本，这通常用 make 工具来完成。

11.6.3 C++中的封装

C++提供两种不同的封装：C 语言定义的头文件和实现文件，或者是类头和类定义的封装。由于 C++模板和独立编译的相互影响的复杂性，所以 C++模板库的头文件经常包含资源的完整定义，而不只是数据声明和子程序协议；部分原因是 C++程序使用了 C 连接器。

把非模板类用于封装时，类的头文件只有成员函数的原型，而类外提供的函数定义在一个代码文件中，就像 11.4.2.4 节中的最后一个例子那样。这就非常清晰地分离了接口和实现。

使用类、但没有泛化封装结构而导致的一个语言设计问题是，有时，定义使用两个不同类对象的操作时，该操作不属于任何一个类。例如，假设一个抽象数据类型用于矩阵，一个抽象数据类型用于向量，而向量和矩阵要进行乘法操作。乘法代码必须能访问向量和矩阵类中的数据成员，而这些代码不属于任何一个类。而且，无论让这些代码属于哪个类，访问另一个类的成员都是有问题的。在 C++中，要解决这个问题，可以让非成员函数成为

类的友元函数。友元函数可以访问类的私有实体，对于向量和矩阵的乘法操作，C++的一个解决方法是在向量和矩阵类的外部定义该操作，但把它定义为向量和矩阵类的友元函数。下面的框架代码演示了这种情形：

```
class Matrix; //** A class declaration
class Vector {
  friend Vector multiply(const Matrix&, const Vector&);
  ...
};
class Matrix { //** The class definition
  friend Vector multiply(const Matrix&, const Vector&);
  ...
};
//** The function that uses both Matrix and Vector objects
Vector multiply(const Matrix& m1, const Vector& v1) {
  ...
}
```

除了函数之外，整个类都可以定义为另一个类的友元，这样该类的所有私有成员就可以由友元类的所有成员访问。

11.6.4　Ada 包

Ada 的包规范可以在共有和私有区域声明任何数量的数据成员和子程序。因此，它们可以包括任意数量的抽象数据类型的接口和其他程序资源。所以包是一个多类型的封装结构。

对于 11.6.3 节中的向量和矩阵类型，以及这两个类的私有成员的访问需求，在 C++中用友元函数来解决这个问题。在 Ada 中，矩阵和向量类型都可以在一个 Ada 包内定义，这样就不需要友元函数了。

11.6.5　C#程序集

C#拥有一个比类更大的封装结构：程序集。它由所有.NET 编程语言使用。程序集由.NET 编译器建立。.NET 应用程序包含一个或多个程序集，程序集是一个文件[7]，它在应用程序中显示为一个动态连接库(.dll)[8]或者可执行文件(.exe)。程序集包含几个不同的组件，其中一个主要的组件是其编程代码，它用中间语言编写，是从其源语言编译过来的。在.NET 中，中间语言称为通用中间语言(Common Intermediate Language，CIL)，由所有.NET 语言使用。由于程序集的代码是用 CIL 编写的，所以程序集可以用于任何体系结构、设备或操

---

7. 程序集可以包含任意多个文件。

8. 动态链接库(DLL)是一组类和方法，在程序执行过程中需要它们时，它们就单独链接到当前执行的程序上。因此，尽管程序可以访问某个 DLL 中的所有资源，也只有实际用到的部分会加载并链接到程序中。自从 Windows 首次发布起，DLL 就是 Windows 编程环境中的一部分，但是，.NET 的 DLL 与以前的 Windows 系统的 DLL 大不相同。

作系统。在执行时，CIL 会即时编译为体系结构上的本机代码。

除了 CIL 代码外，.NET 程序集包括元数据，它描述了程序集定义的每个类以及所使用的所有外部类。程序集还包括一个清单，其中列出了程序集引用的使用其他程序集和程序集的版本号。

在.NET 的世界中，程序集是软件部署的基本单元。程序集可以是私有的，此时它们只能用于一个应用程序；程序集也可以是公有的，这意味着所有的应用程序都可以使用它们。

如前所述，C#有一个访问控制符 internal。类的 internal 成员对于它所在的程序集中的所有类都是可见的。

Java 的一个文件结构类似于程序集，称为 Java Archive(JAR)，它也用于 Java 软件系统的部署。JAR 用 Java 工具 jar 建立，而不是编译器。

11.7 命名封装

封装就是为逻辑关联的软件资源(具体地讲，是抽象数据类型)提供的语法容器。这些封装的作用是提供一种方法，将程序组织到逻辑单元中，进行编译。这就允许程序的各个部分在修改后进行独立的重编译。组织大程序还需要另一种封装：命名封装。

大程序通常由许多开发者独立编写，甚至可能是在不同的地方编写。这就需要程序的逻辑单元是独立的，且仍可以协同工作。这也带来了一个命名问题：独立工作的开发人员如何为变量、方法和类命名，才能避免使用开发该软件系统的其他程序员已使用的名字？

库的使用也会产生这种命名问题。在过去的 20 年里，大型软件系统越来越依赖支持软件的库了。几乎所有用现代编程语言编写的软件都需要使用大型复杂的标准库，以及应用程序专用的库。这种多个库的广泛使用，就需要用新的机制来管理名称。例如，开发人员向现存的库中加入新的名称，或者创建一个新库时，所使用的新名称不能与客户应用程序或者其他库中的已存名称相冲突。没有语言处理器的帮助，这实际上是不可能实现的，因为库的作者无法预知客户程序会使用什么名称，或者客户程序所使用的其他库中使用了什么名称。

命名封装定义了名字作用域，有助于规避这些名称冲突问题。每个库可以创建自己的命名封装，以避免本库与其他库或者客户代码间出现名称冲突的问题。出于同样的目的，软件系统的每个逻辑单元都可以创建一个命名封装。

命名封装是一种逻辑封装，因为它不需要是连续的。几个不同的代码集可以放在一个命名空间中，即使它们存储在不同的地方。下面简要介绍命名封装在 C++、Java、Ada 和 Ruby 中的应用。

11.7.1 C++命名空间

C++包含 namespace 关键字，来帮助程序管理全局命名空间的问题。每个库都可以放在它自己的命名空间中，在该命名空间外使用其中的名称时，要使用命名空间的名字来修

饰程序中的名字。例如，假设有一个实现了栈的抽象数据类型头文件。由于其他库文件可能定义了栈抽象数据类型中已使用的名字，所以把定义栈的文件放在它自己的命名空间中。为此，把栈的所有声明都放在一个命名空间块中，如下：

```
namespace myStackSpace {
// Stack declarations
}
```

栈抽象数据类型的实现文件可以使用作用域解析符::来引用头文件中声明的名字，如下：

```
myStackSpace::topSub
```

实现文件也可以出现在与头文件使用的相同的命名空间块中，这样头文件中声明的所有名称对于实现文件都是可见的。这种方式更简便，但是略微降低了代码的可读性，因为不清楚实现文件的某个名称究竟是在何处声明的。

在库的头文件中，客户程序要访问其命名空间中的名称，有三种方式。第一种是用命名空间的名称修饰库中的名称。例如，变量 topSub 的引用如下：

```
myStackSpace::topSub
```

如果实现文件不在相同的命名空间中，实现代码就可以以这种方式引用它。

另外两个方法使用 using 指令。这个指令可以用于修饰命名空间中的单个名字，如

```
using myStackSpace::topSub;
```

这使 topSub 可见，但不改变 myStackSpace 名称空间中其他名字的可见性。

using 指令还能够用于修饰命名空间中的所有名字，如

```
using namespace myStackSpace;
```

包含这条指令的代码可以直接访问命名空间内定义的名称，例如

```
p = topSub;
```

注意命名空间是 C++中比较复杂的特性，这里只介绍了其最简单的一部分。

C#的命名空间与 C++类似。

11.7.2　Java 包

Java 使用的命名封装结构叫做包。包可以包含多个类型[9]定义。包中的类型之间是部分友元的关系，所谓“部分”是指，包中类型所定义的实体，无论是 public、protected(参见第 12 章)还是未指定访问修饰符，对于本包内的所有其他类型都是可见的。

没有访问修饰符的实体都具备包作用域，因为它们在整个包中都可见。因此 Java 不大需要明确的友元声明，也没有 C++的友元函数或友元类。

---

9. 这里的类型是指类或接口。

文件中定义的资源通过包声明指定到某个包中，如：

```
package stkpkg;
```

包声明语句必须位于文件的第一行。不含有包声明语句的文件资源都隐式地置于同一个未命名的包中。

包的客户程序可以使用完全限定的名字来引用包中定义的类型。例如，如果包 stkpkg 有一个类 myStack，这个类就可以在 stkpkg 的客户程序中引用为 stkpkg.myStack。同样，myStack 对象中的变量 topSub 可以引用为 stkpkg.myStack.topSub。包相互嵌套时，这种方式很快会使代码变得冗长，所以 Java 提供了 import 声明，它允许采用较短的代码引用包中定义的类型名称。例如，客户程序包含如下语句：

```
import stkpkg.myStack;
```

现在，类 myStack 可以仅通过其名称来引用。要访问包中的所有类型名称，可以在 import 语句中用星号代替类型名称。例如，如果希望导入 stkpkg 包中的所有类型，可以使用：

```
import stkpkg.*;
```

注意 Java 的 import 声明只是一种缩写机制，并没有使其他隐藏的外部资源可见。事实上，在 Java 中，对编译器或类装载器(使用包名和 CLASSPATH 环境变量)可见的所有资源都不是隐藏的。

Java 的 import 声明表明了程序对包的依赖关系。如果不使用 import 声明，这种依赖关系就不那么明显。

11.7.3 Ada 包

Ada 包常常用于封装库，它在层次结构中定义，对应于存储它们的目录层次结构。例如，如果 subPack 包定义为 pack 包的子包，则 subPack 的代码文件会存储在 pack 包存储目录的子目录中。Java 的标准类库也是在包的层次结构中定义，存储在对应的目录层次结构中。

如 11.4.1 节所述，包也定义命名空间。使用 with 子句在程序单元中访问包。例如，下面的子句使 Ada.Text_IO 包中的资源和命名空间变得可用。

```
with Ada.Text_IO;
```

访问在 Ada.Text_IO 包的命名空间中定义的名称，需要限定该名称。例如，要访问 Ada.Text_IO 中的 Put 过程，须用如下方式

```
Ada.Text_IO.Put
```

要不加限定地访问 Ada.Text_IO 中的名称，可以使用 use 子句：

```
use Ada.Text_IO;
```

使用这个子句后，访问 Ada.Text_IO 中的 Put 过程时，直接使用 Put 就可以了。Ada 中

的 use 与 Java 中的 import 类似。

11.7.4　Ruby 模块

Ruby 类用作命名空间的封装，其机制类似于其他支持面向对象编程的语言中的类。Ruby 有另一种命名封装，叫做模块。模块一般定义方法和常量集。所以，模块非常便于封装相关方法和常量组成的库，方法和常量的名称在一个独立的命名空间中，不会和使用模块的程序中的其他名称相冲突。模块和类是有区别的，因为模块不能实例化，不能派生子类，也不能定义变量。模块中定义的方法名称包含这个模块的名称。例如如下的模块定义：

```
module MyStuff
  PI = 3.14159265
  def MyStuff.mymethod1(p1)
  ...
  end
  def MyStuff.mymethod2(p2)
  ...
  end
end
```

假设 MyStuff 模块存储于它自己的文件中，则如果程序想使用 MyStuff 的常量和方法，就必须先访问这个模块。这需要使用 require 方法，它把字符串字面量形式的文件名作为参数。然后这个模块的常量和方法就能通过模块名来访问。下面的例子使用示例模块 MyStuff，该模块存储于 myStuffMod 文件中。

```
require 'myStuffMod'
...
MyStuff.mymethod1(x)
...
```

模块详见第 12 章。

小　结

在程序设计作为一门工程学科发展的过程中，抽象数据类型的概念及其在程序设计中的应用是一个里程碑。尽管该概念相对简单，但在支持它的语言设计出来之前，它的应用并不方便、安全。

抽象数据类型的两个基本特征是对数据对象与其相关操作的封装和信息隐藏。语言可能直接支持抽象数据类型，也可能用更一般的封装来模拟它们。

Ada 提供的封装称为包，它们可以用来模拟抽象数据类型。包通常包括两部分：一个是提供了客户接口的声明；另一个是提供了抽象数据类型的实现方法的正文。数据类型的表述形式可以出现在包规范中，不过把它们放在包的私有部分，就可以隐藏它们。抽象数据类型本身是包规范的公共区域定义为私有的。私有类型的内置操作可以执行赋值、相等和不等比较。

C++的数据抽象由类提供。类就是类型，它的实例可以是栈动态的或者堆动态堆的。成员函数(方法)可以把全部定义放在类中，也可以只把协议放在类中，将定义放在其他文件中，这样可以独立编译。C++类可以有两个子句，每个子句把一个访问修饰符作为前缀：private 和 public。构造函数和析构函数都可以在类定义中给出。在堆分配存储空间的对象必须用 delete 显式地释放。

与 C++一样，Objective-C 数据抽象是类。类是类型，而且都是堆动态的。方法声明必须放在类的接口部分，定义必须放在实现部分。构造函数称为初始化器，它们必须显式调用。实例变量可以是私有的或共有的。对方法的访问是无法限制的。方法调用使用的语法类似于 Smalltalk。Objective-C 支持属性，属性的访问方法可以用编译器提供。

Java 的数据抽象和 C++的相似，但不同的是，Java 中所有的对象都是在堆中分配内存，通过引用变量来访问。另外，所有的对象通过垃圾管理器回收。在 Java 中，访问修饰符放在各个声明或定义中，而不是放在子句中。

C#通过结构和类来支持抽象数据类型。结构是值类型，且不支持继承。C#的类和 java 的相似。

Ruby 用类来支持抽象数据类型。Ruby 的类和其他大部分语言的类不同，它是动态的——成员可以在执行过程中添加、删除、修改。

Ada、C++、Java 5.0 和 C# 2005 允许参数化抽象数据类型——Ada 通过它的泛型包，C++通过模板类，Java 5.0 和 C# 2005 通过集合类、接口和用户定义的泛型类来参数化抽象数据类型。

为了支持大型程序的创建，一些当代语言使用了多类型封装结构，它包含一个逻辑相关的类型集。封装也可以提供对其实体的访问控制。封装为程序员提供了一种组织程序的方式，这种方式也便于代码的重编译。

C++、C#、Java、Ada 和 Ruby 提供命名的封装。在 Ada 和 Java 中，它们称为包；在 C++和 C#中，它们称为命名空间；在 Ruby 中，它们称为模块。部分原因是 Java 可以使用包，所以没有友元函数或友元类。在 Ada 中，包也可以用作命名封装。

复 习 题

1. 在编程语言中，两种不同的抽象是什么？
2. 定义抽象数据类型。
3. 抽象数据类型定义分为两部分有什么好处？
4. 支持抽象数据类型的语言设计要求是什么？
5. 抽象数据类型的语言设计问题是什么？
6. 解释 Ada 包如何提供信息隐藏。
7. Ada 包规范的私有部分对于谁是可见的？
8. 在 Ada 中，私有类型和有限私有类型的区别是什么？
9. Ada 的包规范和正文包中的内容分别是什么？

10. Ada 中 with 子句的作用是什么？
11. Ada 中 use 子句的作用是什么？
12. C++类和 Ada 包的基本区别是什么？
13. C++对象是从哪里分配内存的？
14. C++成员函数定义可以放在哪些不同的地方？
15. C++构造函数的用途是什么？
16. 构造函数的合法返回类型是什么？
17. 所有的 Java 方法是在哪里定义的？
18. 如何创造 C++的类实例？
19. Objective-C 类的接口和实现部分是如何指定的？
20. Objective-C 类是类型吗？
21. Objective-C 方法的访问级别是什么？
22. Objective-C 中方法调用的语法来源是什么？
23. Objective-C 中的构造函数何时会隐式调用？
24. 为什么使用属性好于把实例变量指定为公用变量？
25. Java 的类对象是从哪里分配内存的？
26. 为什么 Java 没有析构函数？
27. 所有的 Java 方法是在哪里定义的？
28. Java 类是在哪里分配内存空间的？
29. 为什么 Java 中的析构函数没有 C++中的使用频繁？
30. 什么是友元函数？什么是友元类？
31. Java 没有友元函数和友元类的一个原因是什么？
32. 描述 C#中的结构与类的根本差异。
33. 在 C#中如何创建结构对象？
34. 列出使用私有类型的访问器好于将类型设置为 public 的三个原因。
35. C++中的结构和 C#中的结构有什么不同？
36. 为什么 Java 不需要像 Ada 中的 use 子句？
37. Ruby 中的所有构造函数名是什么？
38. Ruby 中的类和 C++、Java 中的有什么根本差异？
39. 如何创建 Ada 泛型类的实例？
40. 如何创建 C++模板类的实例？
41. 描述在构建大型程序时促使人们开发封装结构的两个问题。
42. 用 C 定义抽象数据类型时会发生什么问题？
43. 什么是 C++的命名空间？它的用途是什么？
44. 什么是 Java 的包？它的用途是什么？
45. 描述.NET 程序集。
46. Ruby 模块可以存放什么元素？

习 题

1. 一些软件工程师认为，所有导入的实体都应该用导出程序单元名来限定。你同意这个观点吗？说明理由。
2. 假设一个人设计了栈抽象数据类型，其中，函数 top 返回一个访问路径(或指针)，而不是返回第一个元素的副本。这样它就不是一个真正的数据抽象。为什么？举例说明。
3. 分析 C++中的命名空间和 Java 中包的异同。
4. 把抽象数据类型设计成指针有什么缺点？
5. 为什么 Ada 包规范必须给出非指针抽象数据类型的结构？
6. 和 C++、Java 中编写访问器方法相比，C#属性有什么优点？
7. 解释 C 中封装方式的危险性。
8. 为什么 C++没有消除第 7 题中的问题？
9. Objective-C 在语法上区分类方法和实例方法的优缺点是什么？
10. C++的方法调用在哪些方面比 Objective-C 更易读？
11. Objective-C 不限制对方法的访问，这有什么优缺点？
12. 为什么析构函数在 Java 中很少使用，而在 C++中必不可少？
13. 简述 C++内联函数的优点与缺点。
14. 在某些情况下，C#的结构优于 C#类，试举出一例。
15. 解释为什么命名封装对开发大型程序如此重要。
16. 描述客户程序从 C++的命名空间中引用名称的三种方式。
17. C#标准库的命名空间 System 对 C#程序并非隐式可用的。这合适吗？为什么？
18. Ruby 中的类可以动态改变，这有什么优点和缺点？
19. 比较 Java 的包和 Ruby 的模块。

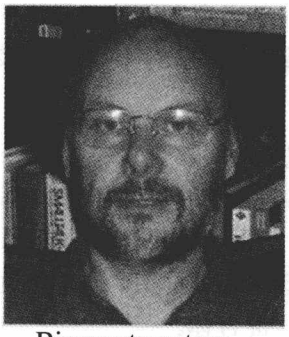
编程练习

1. 用你熟悉的一种语言，设计一个元素为整型的矩阵抽象数据类型，要支持加法、减法和矩阵乘法的操作。
2. 用你熟悉的一种语言，设计一个元素为浮点型的队列抽象数据类型，要支持入队、出队和清空的操作。出队操作删除元素，并返回该元素的值。
3. 修改 11.4.2 节中用于抽象栈类型的 C++类，改为使用链表来表示，并使用本章的代码来测试。
4. 修改 11.4.4 节中用于抽象栈类型的 Java 类，改为使用链表来表示，并使用本章的代码来测试。
5. 编写一个复数的抽象数据类型，支持的操作有：加法、减法、乘法、除法、提取实部与虚部，从两个浮点常量、变量或表达式中构建复数。使用 Ada、C++、Java、C#或 Ruby 语言皆可。

6. 编写一个队列抽象数据类型，队中的每个元素为 10 个字符的名字。这个队列的元素必须动态地从堆中分配存储空间。队列操作包括入队、出队和清空。使用 Ada、C++、Java、C#或 Ruby 语言。
7. 为队列写一个元素可以是任何基本类型的抽象数据类型。使用 Java 5.0、C# 2005、C++或 Ada 语言。
8. 为队列写一个抽象数据类型，它的元素是一个长度为 20 的字符串和一个表示优先级的整数。这个队列必须支持如下操作：入队，将一个字符串和一个整数作为参数；出队，从队列中返回优先级最高的字符串；和清空队列。这个队列不把其中的元素按照优先级进行排序，所以出队操作必须遍历整个队列。
9. 双端队列可以在队头和队尾添加和删除元素。调整编程练习 7 的解决方案，实现双端队列。
10. 为有理数(分子和分母)写一个抽象数据类型，包括构造函数和如下方法：获取分子、获取分母、加法、减法、乘法、除法、相等测试和显示。使用 Java、C#、C++、Ada 或 Ruby 语言。

访　谈

C++：它的诞生，它的无处不在和它受到的常见质疑


Bjarne stroustrup

Bjarne Stroustrup 是 C++的设计者和最初的实施者，*The C++ Programming Language* 和 *The Design and Evolution of C++*的作者，他的研究方向包括分布式系统、仿真、设计、编程和编程语言。Stroustrup 博士是德克萨斯 A&M 大学工程学院的计算机科学专业的教授。他积极参与 C++的 ANSI/ISO 标准化。在 AT&T 工作二十多年后，他仍保持着与 AT&T 实验室的联系，作为一个信息和软件系统研究室的成员来做研究。他是 ACM 的研究员，AT&T 贝尔实验室的研究员，还是 AT&T 的研究员。1993 年，凭借着自己的杰出贡献——“他所做的早期工作为 C++编程语言奠定了基础，在这个基础上由于 Stroustrup 先生持续的努力，C++语言成为在计算历史上最有影响力的编程语言之一”——他获得 ACM 格雷斯默里霍泊奖(ACM Grace Murray Hopper Award)。

关于你和计算技术的简史

你在 20 世纪 80 年代初期加入贝尔实验室之前，是做什么工作的，在哪里工作？

我在 1979 年加入了贝尔实验室，在那里我研究分布式系统的整体领域。在这之前，我在剑桥大学攻读这个领域的博士学位。

你是立即着手开发“含有类的 C”(后来变成了 C++)的吗？

在开始开发“含有类的 C”和 C++之前，我在做几个涉及分布式计算的项目。例如，

我试图找到一种方法，把 UNIX 的内核分配到几台机器中，并帮助许多项目建立模拟器。

你是因为对数学的兴趣而从事这个行业的吗？

我获得了“计算机科学数学”学士学位，硕士学位则是正式的数学学位。我曾错误地以为，计算是某种应用数学。我学习了几年的数学，自己的定位是一个可怜的数学家，但是，这仍然比不懂数学要好得多。我学士毕业时，从来没见过计算机。关于计算技术，我热爱的是编程，而非数学领域。

剖析成功的语言 C++

我喜欢往回看，列出一些使 C++变得无处不在的东西，然后得到用户的反应。它是开源的、非专有的、ANSI/ISO 标准化的。

ISO C++标准是很重要的。有很多的独立开发和演化的 C++实现。这些实现没有共同遵循的标准，也没有标准的过程来引导 C++的进化，就会产生一个紊乱的语言库。

既有开源的实现、又有收费的实现也是很重要的。另外，对于很多用户来说，标准可以提供一个不受实现提供者左右的保护方法也是非常关键的。

ISO 标准进程是开放的、民主的。C++标准委员会的出席人数几乎没有少于过 50 人，每次会议通常多于 8 个国家的代表出席。它不仅仅是供应商的论坛。

它非常适用于系统编程(在 C++刚诞生时，系统编程占有开发代码市场的最大份额)。

诚然，对于任何系统编程项目，C++都是强有力的竞争者。对于当今发展速度最快的嵌入式系统编程来说，它也是有效的。C++发展的另一个领域是高性能的数字、工程以及科学编程。

它的面向对象性质以及类与库的引入，使编程更加有效和透明。

C++是一种多范例编程语言。也就是说，它支持一些基本的编程风格(包括面向对象编程)和那些风格的混合体。如果应用得好，它提供的库将比只使用一个范例更干净、灵活、高效。C++标准库的容器和算法就是一个例子，它基本上是一个泛型编程框架。与面向对象的类体系一起使用时，结果是一个类型安全、有效、灵活的优化混合体。

它在 AT&T 发展环境中的孵化期。

AT&T 贝尔实验室提供了一种对 C++发展至关重要的环境。对于各种疑难问题，该实验室拥有格外丰富的资源，还对实证性研究提供了独一无二的支持。C++和 C 一样源出一个实验室，获益于同样的智能传统、经验和非凡人士。自始至终，AT&T 支持 C++的标准化。但是，不像许多现代语言那样，C++并不是大规模市场运动的受益者。这并不是简单地由实验室产生效用的方式。

我是否在你的排名清单中遗漏了什么东西？

当然。

现在，针对 C++的批评，我需要解释一下：C++是巨大的、难以驾驭的。“hello world”问题在 C++中比 C 中大 10 倍。

C++显然不是一个小型语言，现代语言大都不是小型语言。如果一种语言很小，那么

就需要巨大的库去完成工作，常常需要依靠惯例和扩展内容。我不愿意将语言中非常复杂的核心部分隐藏于系统某处，而更喜欢将这些部分暴露出来，这样才能思考，才能将其有效地标准化。在大部分情况下，我不认为 C++很笨拙。在我的机器中，C++的“hello world”程序并不比 C 版本大，在你的机器中也不会例外。

事实上，在我的机器上，“hello world”的 C++代码比 C 版本少。一种版本的代码比另一个版本的代码多，并没有任何语言方面的原因。这完全是实现者如何组织库的问题。如果一个版本的代码远远多于另一个版本，就该向大版本的实现者报告该问题。

有些批评家认为，与 C 相比，在 C++中编程的约束更加严厉。一旦承认这一点，就是搬起石头砸自己的脚。

是的，我确实说过“在 C 中更容易搬起石头砸自己的脚；而在 C++中比较困难，但是真的这样做时，C++会轰掉你的整条腿。”人们常常会忽略，我所说的关于 C++的话在不同程度上适用于所有功能强大的语言。你试图让人们远离显而易见的危险时，他们会把自己置于更难以察觉的新危险中。避开了小麻烦的人可能正落入一个并不明显的陷阱。在拥有强大的支持机制和保护措施的环境中，一个问题是，难题由于发现太晚，而无法修正，或者难以修正。另外，罕见的问题常常比常见问题更难发现，因为很难怀疑有这个问题。

C++适用于现在的嵌入式系统，但并不适合今天的 Internet 软件。

C++适用于嵌入式系统。也同样适用于且广泛应用于 Internet 软件。例如，如果浏览我的 C++ applications 网页，会发现一些主要的网络服务提供者，如 Amazon、Adobe、Google、Quicken 和 Microsoft，都非常依赖 C++。游戏编程是另一个广泛使用 C++的领域。

在你详述的见解中，我是不是还遗漏了什么？

当然!

第12章

面向对象程序设计的支持

12.1 概述
12.2 面向对象编程
12.3 面向对象语言的设计问题
12.4 Smalltalk 对面向对象编程的支持
12.5 C++对面向对象编程的支持
12.6 Objective-C 对面向对象编程的支持
12.7 Java 对面向对象编程的支持
12.8 C#对面向对象编程的支持
12.9 Ada 95 对面向对象编程的支持
12.10 Ruby 对面向对象编程的支持
12.11 面向对象构造的实现

本章首先简单介绍面向对象编程，接着讨论继承与动态绑定的主要设计问题。然后论述 Smalltalk、C++、Objective-C、Java、C#、Ada 95 和 Ruby 对面向对象编程的支持。本章最后简要综述了面向对象语言中从方法调用到对应方法的动态绑定的实现。

12.1 概述

现在，支持面向对象编程的语言已经稳固地进入了主流。从 COBOL 到 LISP(包括在这两种语言之间的几乎所有语言)都支持面向对象的编程。C++、Objective-C 和 Ada 95 除了支持面向对象的编程以外，还支持面向过程和面向数据的编程。CLOS 是 LISP 的面向对象版本(Paepeke，1993)，它也支持函数式编程。一些较新的支持面向对象编程的语言，并不支持其他的编程范型，但仍然采用基本的命令式结构，具有命令式语言的特征，如 Java 和 C#语言。Ruby 有点不好分类：它的所有数据都是对象，因此 Ruby 是一个纯面向对象的语言，但是它也是支持面向过程编程的混合语言。最后，还有一种非传统的完全面向对象的语言 Smalltalk。Smalltalk 是第一种全面支持面向对象编程的语言。在不同的语言之间，支持面向对象编程的细节各有不同，这是本章的一个主题。

本章与第 11 章联系紧密，在某种意义上说，本章是第 11 章的继续。这种关系充分反映了这样的事实：面向对象编程在本质上是对抽象数据类型中抽象原理的一种应用。具体而言，在面向对象的编程中，从一组相似的抽象数据类型中抽取出共性，以构成一种新类型。这组由相似的抽象数据类型组成的集合，它的各个成员都从新类型中继承了那些共性部分。这种特征就是继承，它是面向对象编程以及支持它的语言的核心。

面向对象编程的另一个特征是从方法调用到对应方法的动态绑定，本章也会详细讨论。

尽管一些函数式语言也支持面向对象编程，例如 CLOS、OCaml 和 F#，但本章不讨论这些语言。

12.2 面向对象编程

12.2.1 引言

面向对象编程概念的根源可以追溯到 SIMULA 67，但是直到 1980 年 Smalltalk 发展成 Smalltalk 80 之后，这个概念才得到全面发展。实际上，有些人认为 Smalltalk 是纯面向对象编程语言的基本模型。面向对象的语言必须支持三个关键的语言特性：抽象数据类型、继承以及方法调用到对应方法的动态绑定。第 11 章讨论了抽象数据类型，所以本章的重点是继承和动态绑定。

12.2.2 继承

提高软件生产率是软件开发者长期以来的压力，这种压力随着计算机硬件成本的持续

下降而不断增大。到 20 世纪 80 年代中后期，在软件开发领域里提高生产率的最好途径是软件复用，这对于很多的软件开发者来说是显而易见的。具有封装和访问控制的抽象数据类型显然是软件复用的候选者。不过，抽象数据类型的复用存在一个问题：几乎在所有的情形下，现有类型的特性和功能都不能完全适用于新的应用。旧的类型至少需要一些细微的修改，但这种修改工作可能有困难，因为这要求修改人员了解部分(如果不是全部的话)已有的代码。在许多情况下，修改人员并不是程序最初的作者。此外，在很多情况下，这种修改还需要更新所有的客户端程序。

用抽象数据类型编程的第二个问题是：所有数据类型的定义都是独立的，并都在同一个层次上。对于要用程序来解决的问题，这种设计常常无法构造出适合问题空间的程序。在许多情况下，原始问题具有多种彼此相关的对象，既有像兄弟关系的(彼此相类似)，又有像父子关系的(具有上下代关系)。

继承提供了一种解决方案，既解决了抽象数据类型复用引入的程序修改问题，也解决了程序组织问题。如果新的抽象数据类型能继承已有类型的数据和功能，而且还允许修改一些实体，新增一些实体，那么，复用就会极为便利，不再需要改变被复用的抽象数据类型。程序员能从已有的抽象数据类型开始，修改得到它的后代类型，满足新问题的要求。而且，继承提供了一种框架，用来定义相关类的层次关系，并反映问题空间中这些类的上下代关系。

仿效 SIMULA 67，在面向对象语言中，抽象数据类型通常称为类。与抽象数据类型的实例一样，类的实例称为对象。通过对另外一个类的继承来定义的类，称为派生类或子类。派生新类的类是父类或超类。定义类对象的操作的子程序称为方法。对方法的调用通常称为消息。对象方法的完整集合称为这个对象的消息协议或消息接口。面向对象的程序中的计算，是通过从一个对象向另外一个对象(在有些情况下是类)发送消息来说明的。

传递消息其实不同于调用子程序。子程序一般会处理数据，而数据要么由其调用者传递为参数，要么进行非局部访问或全局访问。发送给对象的消息是执行对象中某个方法的请求，至少方法所操作的部分数据是对象本身。对象方法所定义的过程可以处理对象本身。因为对象是抽象数据类型，所以这些方法应是处理对象的唯一方式。子程序定义的过程可以处理发送给它的(或者是可以非局部访问或全局访问的)任意数据。

下面是一个简单的继承例子：假定有一个类 Vehicles，它有表示年份、颜色或车型的变量。这个类的一个特例(或子类)是 Truck，它可以从 Vehicles 中继承变量，也可以添加变量，来表示运输能力和车辆数目。图 12-1 简单表示了 Vehicle 类和 Truck 类之间的关系，其中箭头指向父类。

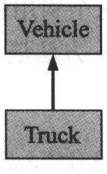


图 12-1　一个简单的继承例子

派生类在多个方面与它的父类存在着不同[1]，以下是父类和子类最常见的一些区别：

(1) 父类可以定义不能被子类访问的私有变量或方法。

(2) 子类可以在继承父类成员的基础上再添加成员。

(3) 子类可以对一个或多个继承来的方法进行修改。修改后的方法与原方法有相同的名字，通常也具有与原方法相同的协议。

新方法覆盖了被继承的方法，所以被继承的方法称为被覆盖的方法，新方法称为覆盖方法。覆盖方法要在子类上提供类似于父类的操作，但该操作是为子类的对象自定义过的。例如，父类 Bird 有一个方法 draw，该方法可以绘制一般的鸟。Bird 的子类 Waterfowl 可以覆盖从 Bird 继承来的 draw 方法，绘制一般的水鸟，例如鸭子。

类可以拥有两种方法和两种变量。最常用的方法和变量是实例方法和实例变量。类的每个对象都拥有一套自己的实例变量，用于存储对象的状态。同一类中两个对象的唯一区别就在于其实例变量的状态不同[2]。例如，汽车类有表示颜色、车型和年份的实例变量。实例方法只能用于操作类的对象。类变量属于类，而不属于类的对象，因此类变量对类只有一个副本。例如，如果希望计算类的实例数目，计数器就不能是实例变量，而必须是类变量。类方法可以对整个类进行操作，也可以对类的对象进行操作。

如果一个新类是个子类，它的父类只有一个，则称这个派生过程为单继承；如果一个类具有多个父类，这个派生过程就称为多继承。当若干个类都通过单继承彼此关联时，这些类的关系就可以用一棵派生树来表示；而多继承的类关系可以用一个派生图来表示。

作为增加复用可能性的一种手段，继承的不足之处是：在继承层次结构中的各个类之间形成了相互依赖。这种结果正好与抽象数据类型的优点相反，抽象数据类型的优点是类型之间是相互独立的。当然，并不是所有的抽象数据类型都必须是完全独立的。但一般来说，抽象数据类型的独立性是它们最突出的正面特征之一。尽管如此，要想增加抽象数据类型的复用性，又不在它们之间产生相互依赖，即使不是不可能，至少也是十分困难的。况且，在很多情况下，类的依赖关系也自然地反映了问题空间中的依赖关系。

12.2.3 动态绑定

面向对象编程语言的第三个特征(排在抽象数据类型和继承之后)，是消息到方法定义的动态绑定过程所提供的一种多态性[3]。这有时称为动态分派。考虑下面的情况：设有一个基类 A，它定义了一个方法 draw，用于绘制与基类相关的图。第二个类 B 定义为 A 的子类。这个新类的对象也需要一个 draw 方法，它类似于 A 提供的 draw，但有一点不同，因为子类对象与基类对象略有不同。于是，子类覆盖了继承来的 draw 方法。如果 A 和 B 的客户程序有一个引用 A 类对象的变量，这个引用也可以指向 B 的对象，该引用就会成为一个多态的引用。如果两个类中定义的 draw 方法通过多态的引用来调用，运行时系统就必须

1. 如果子类与父类没有区别，定义子类显然就没有什么意义。
2. 在 Ruby 中这是不正确的，Ruby 中同一个类的不同对象在其他方面有区别。
3. 多态性参见第 9 章。

在程序执行时决定应调用哪个方法，是 A 中的方法还是 B 中的方法(通过该引用当前所引用的对象类型来决定)[4]，如图 12-2 所示。

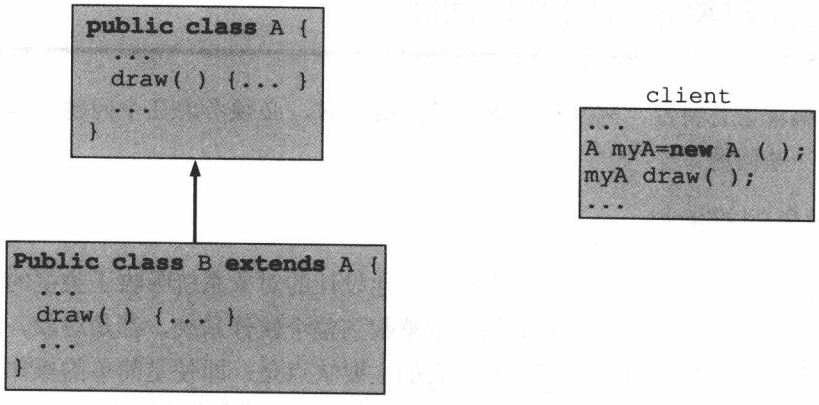


图 12-2　动态绑定

多态性是任何静态类型化的面向对象语言中的一个自然部分。在某种意义上，多态性使静态类型化的语言变得有点动态类型化，因为从方法调用到对应方法的一些绑定是动态的。多态性变量的类型其实是动态的。

上述方法并不是设计多态性引用的唯一方式。Objective-C 还使用了另一种方式，参见 12.6.3 节。

动态绑定的一个目的是允许软件系统在开发和维护过程中更容易扩展。假设某类二手车实现为 car 类，该类别中的每辆车都实现为一个子类。子类包含车的图片和特定信息。用户可以用一个程序借车，该程序在用户借车时会显示车的图片和信息。每辆车的图片和信息中包含一个按钮，如果用户对该车感兴趣，就可以单击该按钮。浏览了所有的车后，或者看过了用户想看的车后，系统会输出用户想查看的车的图片和信息。实现这个系统的一种方式是把感兴趣的每辆车的引用放在基类 car 的引用数组中。用户准备好后，他感兴趣的所有车的信息就可以打印出来，供用户研究和比较列表中的车。当然，车的列表可以频繁改变。这就需要 car 的子类进行相应的修改。但是，子类集合的变化并不需要修改系统的其他部分。

在某些情况下，继承层次结构的这种设计会产生一个或多个类，它们在层次结构中所处的位置太高，将这些类实例化不会有什么意义。例如，假设某个程序定义了一个 Building 类和特定 Building 类型的子类集合，如 French_Gothic。在 Building 类中实现 draw 方法可能就没有任何意义。但由于 Building 的所有后代类都应该有这种方法的实现，所以该方法的协议(不是方法的实现体)就包含在 Building 类中。这种方法常常称为抽象方法(在 C++中称为纯虚方法)。至少包含一个抽象方法的类称为抽象类(在 C++中称抽象基类)。这种类通常不能实例化，因为它有一些方法虽然有声明，但没有定义(没有实现体)。抽象类的任何

4. 方法调用到对应方法的动态绑定有时称为动态多态性。

子类要想实例化，都必须提供所有继承来的抽象方法的实现体(定义)。

12.3 面向对象语言的设计问题

当设计编程语言特性，来支持继承与动态绑定时，必须考虑几个问题。本节将讨论其中最重要的一些问题。

12.3.1 对象的排他性

完全遵循计算的对象模型的语言设计者，他设计的对象系统吸收了类型的其他所有概念。这个系统中的每个事物，从简单的标量整数到整个软件系统，都是对象。这种做法的优点是语言及其应用的优雅性和一致性。它的主要缺点是：即使是简单的操作，也必须通过消息传递过程来完成，这常常使操作比命令式模型中的类似操作要慢，在命令式模型中，单个机器指令可实现这些简单的操作。在这种最纯粹的面向对象计算模型中，所有的类型都是类。预定义的类和用户定义的类之间没有差别。事实上，所有的类都同样地对待，所有计算都是通过消息传递来完成的。

排他性对象的一种替代方法是，保持传统命令式编程语言中的完整类型集合，并简单地加上对象类型化模型，这常见于支持面向对象编程的命令式语言中。这会得到一种大型的语言，它的类型结构会使所有人感到困惑(个别专业用户除外)。

排他性对象的另一种替代方法是：对于原始的标量类型，使用命令式风格的类型结构，而所有结构化的类型实现为对象。这种做法提供了原始值上的操作速度，使之能与命令式模型中所期望的速度媲美。遗憾的是，这种替代方法造成了语言复杂化。不变的是，非对象的值必须与对象混合在一起。这样，对于非对象类型，就需要所谓的包装类，于是，一些经常需要的操作就可以实现为包装类的方法。非对象的值需要这种操作时，该值就转换为对应包装类的对象，并使用包装类的相应方法。这种设计是为了效率而在语言的一致性与纯粹性之间的一种折中。

12.3.2 子类是子类型吗

这个问题相对简单：“是一个(is-a)”关系是否在派生类与其父类之间成立？从纯语义的观点来看，如果一个派生类“是一个”父类，那么这个派生类的对象必须拥有父类对象拥有的所有成员。在不太抽象的层次上，“is-a”关系保证了客户程序中派生类变量可以出现在父类变量可以合法出现的任何地方，而不会引起类型错误。而且派生类的对象必须在行为上与父类对象一致。

Ada 语言的子类型就是数据继承的这种简单形式的例子。例如：

```
subtype Small_Int is Integer range -100..100;
```

Small_Int 类型的变量具有 Integer 变量的所有操作，但只能保存 Integer 所有可能值的一个

子集。此外，每个 Small_Int 变量都可以在任何可使用 Integer 变量的地方使用。也就是说，在某种意义上，Small_Int 变量是 Integer 变量。

子类与基类或父类有很多不同的方面。例如，子类可以有更多的方法，也可以有更少的方法，在一个或多个方法中，一些参数的类型可以不同，某个方法的返回类型可以不同，某个方法的参数数目可以不同，一个或多个方法的定义体可以不同。大多数编程语言对于区别子类与基类的方式有严格的限制。在多数情况下，语言规则要求，子类应是父类的子类型。

如前所述，如果派生类和其父类有 is-a 关系，这个派生类就称为子类型。确保子类是子类型的特征是：如果子类方法覆盖了基类方法，它就必须与对应的被覆盖方法在类型上兼容。这里“兼容”的意思是，对覆盖方法的调用能替换在客户程序任何地方出现的对被覆盖方法的调用，而不会引起类型错误。这就意味着，每个覆盖方法的参数数目必须和被覆盖方法相同，参数类型和返回类型也必须与父类的兼容。拥有相同的参数数目、相同的参数类型和返回类型，当然就能够保证方法的兼容。不过，把限制再放松一些也是可能的，这取决于语言的类型兼容规则。

子类型的定义清楚地说明：不允许父类的公有成员在子类中不是公有成员。因此，在子类型的派生过程中，要求父类的公有成员必须继承为子类的公有成员。

子类型关系和继承关系看起来几乎是完全相同的，然而，这种推测却远非正确。12.5.2 节将用一个 C++例子来解释这种错误的假设。

12.3.3　单继承与多继承

另一个简单的问题是：语言除支持单继承外，也支持多继承吗？也许这并不是一个很简单的问题。多继承的目的是允许一个新类继承两个或多个类。

既然多继承有时非常有用，那么为什么语言设计者不把多继承包括进来呢？这有两种原因：复杂性和效率。多继承新增的复杂性可用几个问题来说明。首先，设一个类有两个不相关的父类，两个父类所定义的名字都不在对方的定义中，这种情况下没有什么问题。然而，假设子类 C 继承了类 A 和类 B，类 A 和类 B 都定义了一个可继承方法 display。如果 C 需要引用这两个版本的 display，该怎么办？如果两个父类定义了相同名称的方法，且这些方法中的一个或两个必须在子类中覆盖时，这种有歧义的问题将会变得更加复杂。

如果类 A 和类 B 派生于同一个父类 Z，且类 A 和类 B 是类 C 的父类，就会出现另一个问题。这种情形称为菱形(或共享)继承。在这种情况下，A 和 B 都应该包含 Z 的可继承变量。假设 Z 有一个可继承变量 sum，那么 C 是应该继承 sum 的这两个版本呢，还是继承其中的一个呢？如果是其中的一个，那应该是哪一个呢？在编写程序时，有时需要继承两个版本中的一个，有时则需要两个都继承。12.11 节讨论了这几种情形的语言实现。菱形继承如图 12-3 所示。

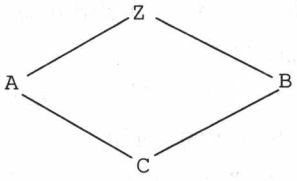


图 12-3 菱形继承示例

效率问题更多是感觉上的，而不是实际的。例如，在 C++中，至少在一些机器的体系结构上，要支持多继承，每个动态绑定的方法调用都只需要一次额外的数组访问和一次额外的加法操作(Stroustrup，1994，p.270)。即使程序没有使用多继承，这个操作也是必不可少的，不过额外的开销很小。

使用多继承容易导致程序组织复杂化。很多试图使用多继承的程序员发现，设计用作多父类的类是有难度的。使用多继承的系统的维护更是一个严峻的问题，因为多继承导致类间的依赖关系更复杂。多继承所带来的好处是否值得设计和维护使用多继承的系统，目前没有明确的结论。

接口是多继承的一种替代方式。接口提供了多继承的一些优点，但缺点较少。

12.3.4 对象的分配和释放

对象的分配和释放涉及两个设计问题。第一个问题是对象分配的位置。如果对象的行为与抽象数据类型类似，也许对象就能够在任何位置进行分配。这就是说，它们可以从运行的栈中分配，或用操作符或函数，如 new，显式地在堆中创建。如果它们都是堆动态变量，则将具有用统一的方法通过指针和引用变量来创建和访问对象的优点。这个设计将简化对象的赋值操作，即在所有情况下，只需要改变指针或引用的值，就可以完成赋值。它也允许对象引用被隐式解引用，从而简化访问的语法形式。

如果对象都是栈动态变量，就会有一个与子类型有关的问题。如果类 B 是类 A 的子类，并且 B 是 A 的子类型，则 B 类型的对象就可以赋给 A 类型的变量。例如，若 b1 是 B 类型的变量，a1 是 A 类型的变量，那么

```
a1 = b1;
```

就是一条合法的语句。如果 a1 和 b 1 都是堆动态对象的引用，就没有什么问题——赋值只是一个简单的指针赋值。但是，如果 a1 和 b1 都是栈动态变量，那么它们就都是值变量，如果赋给对象的值，它们就必须复制到目标对象的空间中。如果 B 在它从 A 类继承的基础上增加了一个数据域，a1 在栈中就没有足够的空间来存放 b1 的所有内容。多出的部分会简单地截掉，这将使编写或使用这段代码的程序员感到困惑。这种截掉称为对象分片。下面的例子和图 12-4 阐明了这一问题。

```
class A {
  int x;
  ...
```

```
};
class B : A {
  int y;
  ...
}
```

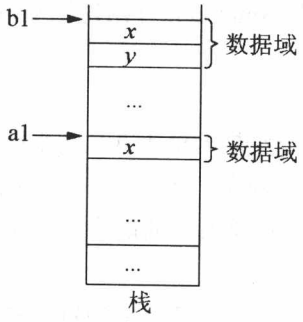


图 12-4　对象分片示例

第二个问题涉及对象从堆中分配的情形。这个问题是，释放空间应是隐式的还是显式的，或者两者兼而有之。如果释放空间是隐式的，就需要某种隐式方法回收存储空间；如果释放空间是显式的，就会带来是否会产生悬挂指针和悬挂引用的问题。

12.3.5　动态绑定与静态绑定

如前所述，消息与方法间的动态绑定，是面向对象编程的基本内容。这里的问题是，消息与方法间的所有绑定是否都是动态的。一种替代方案是，允许用户指定某个绑定是动态的还是静态的。这种方案的优点是：静态绑定的速度比动态绑定快，所以如果一个绑定不需要是动态的，为什么还要付出这个代价呢？

12.3.6　嵌套类

嵌套类定义的主要动机之一是信息隐藏。如果只有一个类需要新类，就没有理由让其他类看到它。在这种情况下，新类可以嵌套在使用它的类中。在某些情况下，新类嵌套在子程序中，而不是直接嵌套在另一个类中。

嵌套了新类的类称为嵌套类。与类嵌套相关的最明显的设计问题是可见性。具体而言，这个问题是：嵌套类中的哪些内容在被嵌套类中是可见的？另一个主要问题正好与此相反：被嵌套类中的哪些内容在嵌套类中是可见的？

12.3.7　对象的初始化

初始化的问题是：在创建对象时，对象是否需要用值初始化，以及如何用值初始化。这个问题比我们首先想到的还要复杂。第一个问题是：对象是必须手工初始化，还是通过某种隐式机制进行初始化。接着的问题是：当创建子类的对象时，被继承的父类成员是隐式地初始化，还是必须由程序员显式地处理。

12.4 Smalltalk 对面向对象编程的支持

很多人将 Smalltalk 视为权威的面向对象编程语言。它是第一个全面支持面向对象设计的语言。因此很自然地，我们将从 Smalltalk 开始讨论支持面向对象编程的语言。

12.4.1 一般特征

在 Smalltalk 中，对象的概念确实是无处不在。几乎所有的东西——从简单的整数常量 2，到复杂的文件处理系统——都是对象。作为对象，它们被同等对待。它们都有自己的内存空间，内在的处理能力，与其他对象通信的能力，以及从祖先那里继承方法和实例变量的可能性。类在 Smalltalk 中不能嵌套。

所有的计算都通过消息来完成，甚至是简单的算术操作。例如，计算表达式 x+7 时，需要把+消息传送给 x(以执行+方法)，并把 7 传送为参数。这个操作返回一个新的数值对象，其内容是相加的结果。

对消息的回复也具有对象形式，用于返回所请求或计算的信息，或者只是用于确认所请求的服务已完成。

所有 Smalltalk 对象都是从堆中分配，并通过引用变量来引用，该引用变量采用隐式解引用。没有显式释放空间的语句或操作。所有的释放操作都是隐式的，使用垃圾收集程序进行存储空间的回收。

在 Smalltalk 中，在对象创建时必须明确调用构造函数。一个类可以有多个构造函数，但是每个构造函数必须有唯一的名称。

与 C++和 Ada 95 这样的混合语言不同，Smalltalk 只是为一个软件开发设计技术——即面向对象——而设计的语言。而且，Smalltalk 没有采用任何命令式语言的语法形式。这种语言在设计上的简单优雅和一致性，反映了设计目的的纯粹性。

第 2 章包含 Smalltalk 程序的一个示例。

12.4.2 继承

Smalltalk 子类继承了其父类中的所有实例变量、实例方法以及类方法。子类也可以有自己的实例变量，但它们的名字必须与祖先类中的变量名不同。最后，子类可以定义新的方法，以重新定义某个祖先类里已有的方法。当子类的某个方法名称和协议与某个祖先类的方法相同时，这个子类的方法就隐藏了祖先类中的方法。要访问这个被隐藏的方法，需要给消息加上伪变量 super 作为前缀。这个前缀使方法搜索从超类开始，而不是从子类本身开始。

因为子类不能隐藏父类的实体，所以所有子类都是父类的子类型。

Smalltalk 语言支持单继承，不允许多继承。

12.4.3　动态绑定

在 Smalltalk 中，消息到方法的动态绑定是按如下操作进行的：发给一个对象的消息会在对象所属的类中搜索对应的方法。如果搜索失败，将在这个类的超类中继续搜索，以此类推，直到没有超类的系统类 Object 为止。Object 是类派生树的根，每个类都是这棵派生树中的一个结点。如果在上述搜索链中没有找到任何方法，就会产生一个错误。注意，这种方法搜索是动态的，即在发送消息时进行搜索。Smalltalk 在任何情况下都不会把消息与方法进行静态绑定。

Smalltalk 中唯一的类型检查方式是动态的，只有把消息发送给没有匹配方法的对象(无论方法是在对象所属类中局部定义，还是通过继承得到)，才会产生类型错误。这个类型检查的概念不同于其他大多数语言。Smalltalk 的类型检查有一个简单的目的，就是确保消息与某个方法相匹配。

Smalltalk 的变量是无类型的，任何名字都可以绑定到任意的对象上。这样的直接结果是 Smalltalk 支持动态多态性。只要变量类型前后一致，具体什么类型就无关紧要，在这个意义上，所有的 Smalltalk 代码都是与类型无关的泛型代码。变量上的操作(方法或操作符)，其含义由变量当前绑定对象的类来决定。

这个讨论的关键点是，只要表达式中引用的对象有方法与表达式的消息相匹配，则对象的类型就是无关紧要的。这就意味着代码与特定类型是无关的。

12.4.4　Smalltalk 的评价

尽管 Smalltalk 系统很庞大，但 Smalltalk 语言还是一种小型语言。语言的语法很简单，而且非常规则。如果语言是根据一个简单但内涵丰富的概念建立起来的，那么即使是一种小型语言，也可以提供强大的功能，Smalltalk 正是这样一种语言。对于 Smalltalk 来说，这个概念就是：只使用通过继承建立的类层次结构、对象以及消息传递，就能完成所有的程序设计。

与传统的命令式编译语言的程序相比较，相同功能的 Smalltalk 程序要慢得多。Smalltalk 的消息传递模型能够提供数组索引和循环，这一点虽然在理论上比较有趣，但效率仍然是评价编程语言的一个重要因素。因此，在 Smalltalk 实际可应用性的大部分讨论中，效率显然是个问题。

Smalltalk 的动态绑定允许直至程序运行时才检查类型错误。在编写程序时，可以包含发送给并不存在的方法的消息，而且直到发送消息时，这个问题才会检测出来，因此，与静态类型语言相比，Smalltalk 有更多的错误需要在后续开发中修正。但实际上，Smalltalk 程序的类型错误并不是个严重的问题。

总而言之，Smalltalk 语言的设计始终偏重语言的优雅性，并严格遵从支持面向对象编程的原则，经常不考虑实际问题，特别是执行的效率。表现得最明显的是：只使用对象以及无类型变量。

Smalltalk 的用户界面对计算有重要的影响：窗口的集成使用，鼠标指向设备以及弹出式和下拉式菜单，这些当代软件系统的主流技术都是首先出现在 Smalltalk 中的。

或许，Smalltalk 最深远的影响是面向对象编程的先进思想，如今这已经成为使用最广泛的设计与编码方法。

12.5 C++对面向对象编程的支持

第 2 章介绍了 C++是如何由 C 和 SIMULA 67 演化而来的，其设计目标是支持面向对象程序设计，并与 C 几乎完全向后兼容。C++的类支持抽象数据类型，参见第 11 章。本节将探讨 C++对面向对象程序设计的其他要素的支持。C++的类、继承和动态绑定的细节十分庞杂。本节只讨论其中最重要的一些问题，尤其是与 12.3 节描述的设计问题直接相关的问题。

C++是第一个广泛使用的面向对象程序设计语言，而且仍然是最流行的语言之一。因此自然而然地，它经常与其他语言进行比较。基于这两个原因，这里对 C++的介绍比本章讨论的其他语言更为详细。

12.5.1 一般特征

C++一个主要的设计考虑就是保持与 C 语言的向后兼容，因此，它保留了 C 语言的类型系统，并增加了类。于是，C++既有传统命令式语言的类型，也有面向对象语言的类结构。它既支持方法，又支持不属于具体类的函数。这使 C++成为一种混合语言，同时支持过程式程序设计和面向对象的程序设计。

C++的对象可以是静态的、栈动态的和堆动态的。因为 C++没有隐式的存储空间垃圾回收，所以堆动态对象需要用 delete 操作符显式地解除空间分配。

C++中许多类的定义都包含了一个析构方法，当类的一个对象不再存在时，就隐式地调用析构方法。析构方法用于释放数据成员引用的堆内存，它也经常用来记录对象在消亡时的部分或全部状态信息，以进行调试。

12.5.2 继承

C++的类可以派生于已有的类，已有的类就成为这个 C++类的父类或基类。与 Smalltalk 和大多数支持面向对象编程的语言不同的是，C++类也能够独立存在，而不需要超类。

在类定义中定义的数据称为这个类的数据成员，在类定义中定义的函数称为这个类的成员函数(在其他语言中，成员函数常常称为方法)。派生类会继承基类的部分或全部成员，也可以增加新的成员，修改继承的成员函数。

所有 C++对象都必须在使用前进行初始化。因此，所有的 C++类都至少包含一个构造函数，用来初始化新对象的数据成员。构造函数在对象创建时隐式地调用。如果有任何数据成员是指向堆分配数据的指针，则构造函数要为之分配存储空间。

如果一个类有基类，那么这个类继承而来的数据成员必须在创建子类对象时初始化。为此，就要隐式调用基类的构造函数。必须给基类构造函数提供初始化数据时，可以在调

用派生类构造函数时提供这些数据。一般是使用下面的结构来完成：

```
子类(子类参数)：父类(父类参数){
...
}
```

如果类定义中没有包含构造函数，则编译器将自动生成一个默认构造函数。如果这个类有基类，默认构造函数将自动调用基类的构造函数。

类成员可以是私有的、受保护的或者公有的。私有成员只能通过成员函数和类的友元访问。函数和类都可以声明为一个类的友元，从而有权访问这个类的私有成员。公有成员在任何地方都是可见的。受保护成员与私有成员相似，不过在派生类中有些例外，后面将讨论保护成员的访问权限。派生类可以修改它们所继承的成员的可访问性。派生类的语法格式如下：

```
class 派生类名称：派生方式 基类名称
{数据成员和成员函数的声明};
```

派生方式可以是 public 或者 private[5](不要把公有派生和私有派生与公有成员和私有成员搞混了)。在公有派生的类中，基类的公有成员和受保护成员分别是公有的和受保护的；在私有派生的类中，基类的公有成员和受保护成员都是私有的。因此，在类的层次结构中，私有派生的类断绝了所有后续类对于任何祖先类的成员的访问权限，而受保护成员或许可以或许不可以由后续的子类访问。基类的私有成员由派生类继承，但是它们对派生类的成员是不可见的，因此在那里没有什么用。私有派生提供了如下可能性：子类的成员可以有不同于父类对应成员的访问权。考虑以下的例子：

```
class base_class {
  private:
    int a;
    float x;
  protected:
    int b;
    float y;
  public:
    int c;
    float z;
};

class subclass_1 : public base_class {...};
class subclass_2 : private base_class {...};
```

在 subclass_1 中，b 和 y 是受保护的，c 和 z 是公有的。在 subclass_2 中，b、y、c 和 z 是私有的。subclass_2 的派生类的成员都不能访问 base_class 中的任何成员。base_class 的数据成员 a 和 x 在 subclass_1 和 subclass_2 中都是不能访问的。

---

5. 也可以是 protected，但这里不讨论这个选项。

注意，私有派生的子类不能是子类型。例如，如果基类有一个公有数据成员，那么在私有派生方式下，那个数据成员在子类中是私有的。因此，如果子类的一个对象替换了基类的一个对象，那么在子类对象上对该数据成员的访问是不合法的。is-a 关系将被破坏。

在私有派生方式下，父类成员对派生类的实例都不是隐式可见的。任何需要可见的成员必须在派生类中重新输出成公有成员。虽然派生方式还是私有的，但重新输出事实上使成员不再隐藏。例如，考虑下面的类定义：

```
class subclass_3 : private base_class {
  base_class :: c;
  ...
}
```

现在，subclass_3 的实例可以访问 c。对于 c 而言，派生方式好像已经成为公有的。在这个类定义中的双冒号(::)是域解析运算符，用来说明定义后面实体的类。

下面段落的例子说明了私有派生的目的和用途。

考虑下面 C++继承的例子，在此例中定义了一个普通的链表类，然后使用这个类来定义了两个有用的子类：

```
class single_linked_list {
  private:
    class node {
      public:
        node *link;
        int contents;
    };
    node *head;
  public:
    single_linked_list() {head = 0};
    void insert_at_head(int);
    void insert_at_tail(int);
    int remove_at_head();
    int empty();
};
```

被嵌套的类 node 定义了链表的一个结点，它包含一个整数变量和一个指向 node 对象的指针变量。node 类在 private 子句下，使 node 对其他类是不可见的。然而 node 的成员都是公有的，所以它们对于嵌套类 single_linked_1ist 是可见的。如果这些成员是私有的，则 node 类需要将嵌套类声明为友元，以使自己的成员在嵌套类中是可见的。注意被嵌套类对于嵌套类的成员并没有特殊的访问权。只有嵌套类中的静态数据成员对被嵌套类的方法是可见的[6]。

嵌套类 single_linked_1ist 只有一个数据成员，即作为链表头的一个指针。它还包含一

6. 类也可以在嵌套类的方法中定义，这种类的作用域规则与直接嵌套在其他类中的类相同，甚至也与在方法中声明的局部变量的作用域规则相同。

个构造函数，用来将 head 设置为空指针值。四个成员函数允许将节点插入到链表对象的任意一端，从链表的一端删除节点，以及测试链表是否为空。

下面的定义给出了栈和队列类，它们都基于 single_linked_list 类：

```
class stack : public single_linked_list {
  public:
    stack() {}
    void push(int value) {
      insert_at_head(value);
    }
    int pop() {
      return remove_at_head();
    }
};
class queue : public single_linked_list {
  public:
    queue() {}
    void enqueue(int value) {
      insert_at_tail(value);
    }
    int dequeue() {
      remove_at_head();
    }
};
```

注意，stack 和 queue 子类的对象都可以访问定义在基类 single_linked_list 中的 empty 函数(因为它采用公共派生方式)。两个子类都定义了不做任何事情的构造函数。在创建子类对象时，就隐式调用子类中相应的构造函数。然后调用基类中的可用构造函数。因此，在本例中，当创建 stack 类的对象时，将调用 stack 的构造函数，它不做任何事。然后调用 single_linked_list 中的构造函数，它会完成必要的初始化工作。

类 stack 和类 queue 都存在一个同样严重的问题：它们的客户程序都能访问父类 single_linked_list 的所有公共成员。因而，stack 对象的客户程序可以调用 insert_at_tail，因此破坏了栈的完整性。同样，queue 对象的客户程序可以调用 insert_at_head。允许进行这些不想要的访问，是因为 stack 和 queue 都是 single_linked_list 的子类型。当需要子类继承基类的整个接口时，使用公有派生方式。一种替代方案是允许子类只继承基类实现的派生方式。这两个例子使用 private 而不是 public 的派生方式来编写两个派生类[7]，使它们不是父类的子类型。这样，两者也都需要重新输出 empty，因为对这两个派生类的实例，empty 将会变成隐藏的。这种情况说明需要私有派生选项。下面是新栈 stack_2 和新队列 queue_2 的定义：

```
class stack_2 : private single_linked_list {
  public:
    stack_2() {}
    void push(int value) {
```

7. 它们不是子类型，因为父类的公有成员在客户程序中是可见的，但在子类的客户程序中不可见(因为在子类中这些父类成员是私有的)。

```
        single_linked_list :: insert_at_head(value);
    }
    int pop() {
      return single_linked_list :: remove_at_head();
    }
    single_linked_list:: empty();
};
class queue_2 : private single_linked_list {
  public:
    queue_2() {}
    void enqueue(int value) {
      single_linked_list :: insert_at_tail(value);
    }
    int dequeue() {
      single_linked_list :: remove_at_head();
    }
    single_linked_list:: empty();
};
```

注意这两个类使用重新输出功能，允许在客户程序中访问基类的方法。使用公有派生方式时，这不是必不可少的。

这两个版本的栈和队列说明了子类型和不是子类型的派生类型之间的区别。因为可以将栈和队列实现为链表，链表就是这两者的一般化形式。所以，可以通过继承一个链表来定义栈和队列的类。然而，栈和队列的类都不是链表类的子类型，因为它们都将父类的公有成员设置为私有，这使客户程序不能访问它们。

需要友元的原因之一是有时必须编写这样的子程序，它能够访问两个不同类的成员。比如，假设程序用一个类来表示向量，用另一个类来表示矩阵，并需要一个子程序将向量对象和矩阵对象相乘。在 C++中，这个乘法函数可以作为这两个类的友元。

C++允许多重继承，它允许将多个类指定为一个新类的父类。例如，假定需要一个类，用于绘制图形，且新类的方法需要在一个单独的线程中运行，就可以定义：

```
class Thread { ... };
class Drawing { ... };
class DrawThread : public Thread, public Drawing { . . . };
```

类 DrawThread 继承了类 Thread 和类 Drawing 的所有成员。如果 Thread 和 Drawing 碰巧包含了同名的成员，那么这些成员可以使用域解析操作符(::)由 DrawThread 类的对象无歧义地引用。这个多继承的例子如图 12-5 所示。

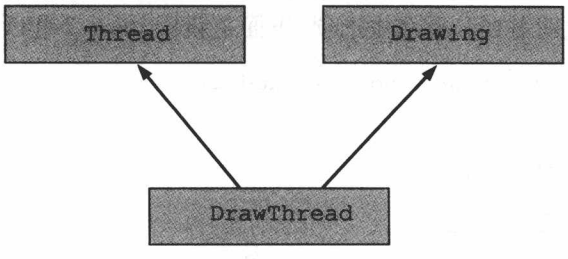


图 12-5 多继承

12.11 节将讨论 C++中实现多重继承的一些问题。

C++中的覆盖方法必须拥有和被覆盖的方法完全相同的参数形式。如果在参数形式中有任何区别，子类的方法就会被认为是一个新方法，与祖先类中同名的方法无关。覆盖方法的返回类型必须与被覆盖方法的返回类型相同，或者必须是被覆盖方法的返回类型的一个公有派生类型。

12.5.3　动态绑定

前面定义的所有成员函数都是静态绑定的，也就是说，每个成员函数的调用都静态绑定到函数定义上。C++对象可以通过值变量来操作，而不是通过指针或引用(这样的对象是静态的或栈动态的)。然而，在这种情况下，对象的类型是已知的、静态的，所以不需要动态绑定。另一方面，使用具有基类类型的指针变量可以指向任何由基类公有派生的类的堆动态对象，使它成为一个多态变量。如果基类的任何成员都不是私有的，公有派生的子类就是子类型。私有派生的子类不是子类型。指向基类的指针不能用来引用在不是子类型的子类中的方法。

C++不允许把值变量(不是指针和引用)设置为多态的。通过多态变量调用派生类中被覆盖的成员函数时，该调用必须动态绑定到正确的成员函数定义上。需要动态绑定的成员函数必须声明为虚拟函数，为此，需要将保留字 virtual 放在这些函数的前面，而这种声明只能出现在类体中。

假设有一个基类 shape 和为各种形状派生的类集，如圆形、矩形等等。如果要显示这些形状，则对于每个子类或每种形状来说，显示的成员函数 draw 都必须是唯一的。draw 成员函数的这些版本必须定义为虚函数。当使用指向派生类的基类的指针调用 draw 函数时，这个调用必须动态绑定到正确的派生类的成员函数上。下面给出了上述例子中的定义：

```
class Shape {
  public:
    virtual void draw() = 0;
  ...
};
class Circle : public Shape {
  public:
    void draw() { ... }
  ...
};
class Rectangle : public Shape {
  public:
    void draw() { ... }
  ...
};
class Square : public Rectangle {
  public:
    void draw() { ... }
  ...
};
```

给出这些定义后，下面的代码给出了静态绑定和动态绑定两类调用的例子：

```
Square* sq = new Square;
Rectangle* rect = new Rectangle;
Shape* ptr_shape;
ptr_shape = sq;          // Now ptr_shape points to a
                         // Square object
ptr_shape->draw();       // Dynamically bound to the draw
                         // in the Square class
rect->draw();            //  Statically bound to the draw
                         // in the Rectangle class
```

这个例子如图 12-6 所示。

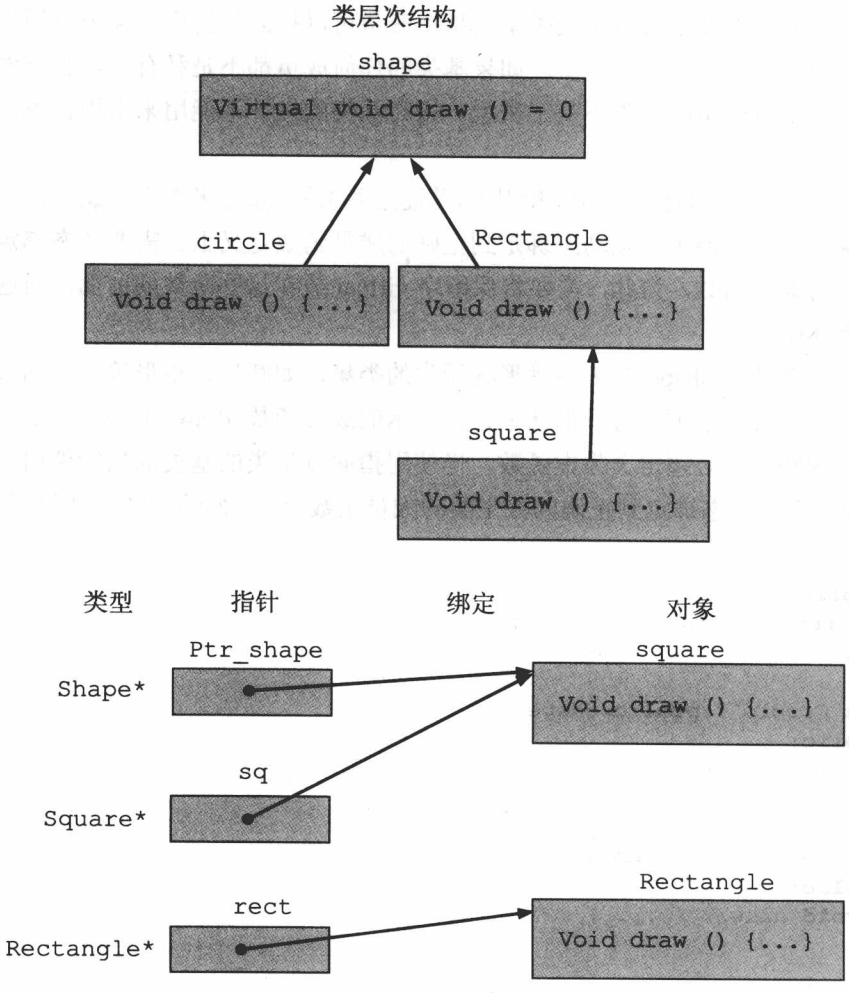


图 12-6 动态绑定

注意，基类 shape 中定义的 draw 函数设置为 0。使用这个特定的语法来指示这个成员函数是纯虚函数，这意味着这个函数没有函数体，且不能调用。如果派生类调用这个函数，

就必须在派生类中重新定义这个函数。纯虚函数的作用是提供函数的接口，而不用给出函数的任何实现。基类中的实际成员函数没有用时，通常定义纯虚函数。12.2.3 节讨论了基类 Building，每个子类都描述了某种特定的建筑。每个子类都有一个 draw 方法，但该方法在基类中没有用。所以 draw 是 Building 类中的一个纯虚函数。

任何包含纯虚函数的类都是抽象类。在 C++中，抽象类不用保留字来标识。抽象类可以包含完全定义好的方法。实例化抽象类是不合法的。在严格意义上，抽象类只用来表示类型的特征。C++提供抽象类来模拟这些真正的抽象类型。如果抽象类的子类没有重新定义其父类的纯虚函数，那么这个函数仍然是子类中的纯虚函数，且子类也是抽象类。

抽象类和继承一起为软件开发提供了一种功能强大的技术。这些技术允许按照层次定义类型，这样相关联的类型可以作为抽象类型的子类，这个抽象类型定义子类的共同抽象特征。

动态绑定允许编写 draw 的任何版本之前，就编写调用 draw 的代码。新的派生类可以在多年后再添加，而不需要修改使用这种动态绑定成员的代码。这正是面向对象语言的一个非常有用的特点。

栈动态对象的引用赋值与堆动态对象的指针赋值不同，例如下面的代码使用了与上例相同的类层次：

```
Square sq;          // Allocate a Square object on the stack
Rectangle rect;     // Allocate a Rectangle object on
                    // the stack
rect = sq;          // Copies the data member values from
                    // the Square object
rect.draw();        // Calls the draw from the Rectangle
                    // object
```

在赋值语句 rect=sq 中，sq 引用对象的成员数据赋给 rect 引用对象的数据成员，但是 rect 仍然引用 Rectangle 对象。因此，通过 rect 引用对象调用 draw 函数，将调用 Rectangle 中的 draw 函数。如果 rect 和 sq 是指向堆动态对象的指针，该赋值将是指针赋值，这种赋值使 rect 指向 Square 对象，并且通过 rect 调用 draw 将被动态绑定到 Square 对象的 draw 方法。

12.5.4　评价

比较 C++与 Smalltalk 的面向对象特性是很自然的。在访问控制方面，C++的继承要比 Smalltalk 更复杂。通过类定义中的访问控制、派生的访问控制以及友元函数和友元类，C++程序员能够高度细微地控制类成员的访问。此外，C++提供了多继承，而 Smalltalk 只允许单继承，但许多人认为这不是 C++的一个优点。多继承的缺点大大超过了它带来的价值。实际上，C++是本章讨论的唯一一个支持多继承的语言。另一方面，在多继承方面提供其他解决方案的语言，例如 Objective-C、Java 和 C#，显然在这个领域超过了 Smalltalk。

在 C++里，程序人员可以指定使用静态绑定还是动态绑定。因为静态绑定比较快速，

所以在不需要动态绑定的情况下，静态绑定具有一些优势。此外，C++的动态绑定与 Smalltalk 的动态绑定相比，速度较快。在 C++中，将虚拟成员函数调用绑定到函数定义上的成本是固定的，不论在继承层次结构中这个函数定义在什么地方。与静态绑定的调用相比，访问虚函数的调用只需要五次额外的内存引用(Stroustrup，1988)。然而，在 Smalltalk 中，消息总是动态绑定到方法上，正确的方法在继承层次结构中相距越远，消耗的时间就越长。由用户来选择静态绑定和动态绑定的缺点是，必须在最初的设计中包括这种选择决定，但这种选择决定也可能在后面修改。

C++中的静态类型检查优于 Smalltalk。在 Smalltalk 中，所有的类型检查都是动态的。Smalltalk 程序可以编写为把消息发送给不存在的方法，直到程序执行时才发现。而 C++编译器可以发现这种错误。编译器发现的错误比测试时发现的错误的修正代价要低。

Smalltalk 基本上是无类型的，这意味着所有的代码都是非常通用的。这提供了非常大的灵活性，但牺牲了静态类型检查。C++通过模板系统(参见第 11 章)提供了泛型类，这种类保持了静态类型检查的优点。

Smalltalk 的主要优点是语言的优雅和简单，这归功于其单一的设计思想。Smalltalk 只考虑了面向对象的范型，并没有为了一些固定用户的需求而改变。而 C++是一种庞大而复杂的语言，不以单一的设计思想为基础，还支持面向对象程序设计，包括 C 的用户基础。它的一个重要目标是，在提供面向对象程序设计的优点的同时，保留 C 语言的效率和风格。有些人认为，这种语言的一些特性并不总是相互兼容的，至少它的一些复杂性是不必要的。

根据 Chambers 和 Ungar(1991)的测试，Smalltalk 运行一组小型 C 程序的速度仅仅是优化过的 C 语言的 10%。而 C++程序仅仅需要比等同的 C 语言多一点的时间(Stroustrup，1988)。Smalltalk 和 C++在效率上有极大的差别，不难想象 C++的商业使用远远超过了 Smalltalk。这还有其他因素，不过效率无疑是选择 C++的一个重要理由。当然，支持面向对象编程的所有编译后的语言都比 Smalltalk 快约 10 倍。

12.6 Objective-C 对面向对象编程的支持

本节讨论 Objective-C 对面向对象编程的支持，前面讨论了 C++对面向对象编程的支持。这两种语言几乎是同时设计的，它们都添加了对 C 语言的面向对象编程的支持。表面上，它们最大的区别是方法调用的语法，C++的方法调用比较接近 C 的函数调用，而 Objective-C 的方法调用比较接近 Smalltalk 的方法调用。

12.6.1 一般特征

Objective-C 与 C#一样，都有基本类型和对象。类定义包含两部分：接口和实现。这两部分常常放在不同的文件中，接口文件使用.h 扩展名，实现文件使用.m 扩展名。把接口放在一个单独的文件中时，实现文件的开头如下：

```
#import "interface_file.h"
```

实例变量在一个以空格分隔开的、用花括号来界定的块中声明，跟在接口部分的头文件之后。Objective-C 不直接支持类变量，但在实现文件中定义的静态全局变量可用作类变量。

类的实现部分包含在对应接口部分声明的方法定义。

Objective-C 不允许嵌套类。

12.6.2　继承

Objective-C 仅支持单继承。每个类都必须有一个父类，但预定义的根类 NSObject 除外。只有一个根类的一个原因是，每个类都需要某些操作，其中包括类方法 alloc 和 init。在 interface 指令中，与新类关联的父类在冒号的后面声明，如下所示：

```
@interface myNewClass: NSObject {
```

因为基类的数据成员可以声明为私有，所以子类不一定是子类型。当然，父类中所有受保护和公共的数据成员都由子类继承。子类可以添加新方法和实例变量。所有的方法都是公共的，这不能改变。在子类中定义的方法若与父类的方法有相同的名称、相同的返回类型、相同数量和类型的参数，就会覆盖被继承的父类方法。被覆盖的方法可以在子类的另一个方法中通过 super(即父对象的引用)来调用。无法禁止覆盖被继承的父类方法。

与 Smalltalk 相同，在 Objective-C 中，可以在任何对象上调用任意方法名。如果运行系统发现，某对象没有某个方法(其协议是正确的)，就会出现错误。

Objective-C 不支持 C++中私有和受保护的派生方式。

与其他支持面向对象编程的语言一样，子类实例的构造函数应总是先调用父类的构造函数，再执行其他操作。如果父类构造函数的名称是 init，就使用如下语句：

```
[super init];
```

除了派生子类之外，Objective-C 还有两种扩展类的方式：类别和协议。使用所谓的“类别”结构可以给类添加一个方法集。类别是类的次级接口，它包含方法的声明。在次级接口中不能包含任何新的实例变量。这种接口的语法形式如下所示：

```
#import "Stack.h"
@interface Stack (StackExtend)
  -(int) secondFromTop;
  -(void) full;
@end
```

这个类别的名称是 StackExtend。初级接口是可以访问的，因为导入了它，所以不需要提及父类。新方法混合到初级接口的方法中。因此，类别有时称为 mixins。mixins 有时用于给不同的类添加某些功能。当然，类仍有一个正常的超类，从超类中继承其成员。所以 mixins 提供了多继承的一些优点，如果模块不需要在其函数上添加模块名称，也不会出现命名冲突。当然，类别还必须有一个实现部分，该实现部分在实现指令中把类别名包含在类名后面的圆括号中，如下所示：

```
@implementation Stack (StackExtend)
```

实现部分不需要实现类别中的所有方法。

在 Objective-C 中还有另一种方式可以提供多继承的一些优点：协议。尽管 Objective-C 没有像 C++那样提供抽象类，但协议与抽象类相关。协议是方法声明的列表。协议的语法如下所示：

```
@protocol MatrixOps
  -(Matrix *) add: (Matrix *) mat;
  -(Matrix *) subtract: (Matrix *) mat;
@optional
  -(Matrix *) multiply: (Matrix *) mat;
@end
```

在这个例子中，MatrixOps 是协议的名称。使用该协议的类必须实现 add 和 subtract 方法。这种用法称为实现或采纳协议。optional 部分表示，multiply 方法可能或不可能由采纳协议的类实现。

在接口指令中，采纳协议的类把协议的名称放在类名后面的尖括号中，如下所示：

```
@interface MyClass: NSObject <YourProtocol>
```

12.6.3 动态绑定

在 Objective-C 中，多态的实现方式不同于大多数其他常见的编程语言。把变量声明为 id 类型，就创建了一个多态变量。这种变量可以引用任何对象。运行系统会跟踪 id 类型的变量所指向的对象类。如果方法的调用是通过这种变量进行的，该调用就会动态绑定到正确的方法上（假设该方法存在）。

例如，假设程序定义了类 Circle 和 Square，它们都有 draw 方法。考虑下面的框架代码：

```
// Create the objects
Circle *myCircle = [[Circle alloc] init];
Square *mySquare = [[Square alloc] init];

// Initialize the objects
[myCircle setCircumference: 5];
[mySquare setSide: 5];

// Create the id variable
id shapeRef;

//Set the id to reference the circle and draw it
shapteRef = myCircle;
[shapeRef draw];

// Set the id to reference the square
shapeRef = mySquare;
[shapeRef draw];
```

这段代码先绘制圆，再绘制正方形，这两个 draw 方法都通过 shapeRef 对象引用来调用。

12.6.4　评价

Objective-C 对面向对象编程的支持是很充分的，但有几个小缺陷。无法禁止覆盖被继承的方法。用 id 数据类型支持多态性有点矫枉过正，因为它允许变量引用任意对象，而不是仅引用继承链上的对象。尽管 Objective-C 不直接支持多继承，但该语言包含一种形式的 mixin，即类别，类别提供了多继承的一些功能，但没有多继承的缺点。类别允许把一组行为添加到任意类中。协议提供了类似 Java 中接口的功能，也提供了多继承的一些功能。

12.7　Java 对面向对象编程的支持

Java 中类、继承和方法的设计都和 C++类似，所以本节重点讨论 Java 与 C++的不同之处。

12.7.1　一般特征

与 C++一样，Java 支持对象和非对象的数据。然而，在 Java 中，只有基本的标量类型(布尔、字符和数值类型)不是对象。Java 的枚举和数组是对象。效率是 Java 有非对象数据的原因。

在 Java 5.0+中，把基本类型的值放在对象的上下文中时，会对这个值进行隐式的强制转换，这将基本类型的值强制转换为基本值类型的封装类的对象。例如，把 int 值或变量放进对象的上下文中，会创建一个值为 int 的 Integer 对象。这个强制转换称为装箱。

虽然 C++类可以定义为没有父类，但这在 Java 中是不可能的。Java 中的所有类都必须是根类 Object 的子类，或者是 Object 的后代类的子类。

所有 Java 对象是显式堆动态的，大多用操作符 new 来分配空间，但没有显式的解除分配操作符。垃圾回收用于存储空间的回收。与许多其他的语言特征一样，尽管垃圾回收避免了一些严重的问题，例如悬挂指针等，但它也可以带来其他问题。其中的一个问题是垃圾回收器只回收对象占用的存储空间，而不进行其他操作。例如，如果一个对象可以访问其他资源，而不能访问堆存储空间，如文件或者共享资源的锁，则垃圾回收器就不会回收这些资源。对于这些情况，Java 允许包括一个特殊的方法 finalize，它与 C++析构函数类似。

垃圾回收器要回收对象占用的存储空间时，就隐式地调用 finalize 方法。finalize 方法的问题是它运行的时间是不可控制的，甚至是不可预测的。用 finalize 方法回收对象占用的存储空间的一种替代方法是定义一个回收方法。这样做的唯一问题是，对象的所有客户程序都必须知道这个方法，并必须记得调用它。

12.7.2　继承

在 Java 中，方法可以定义为 final，这意味着它不能在任何子类中被覆盖。在类的定义中使用了保留字 final 时，就意味着这个类不能派生任何子类。也意味着这个子类的方法到方法间的调用绑定是静态的。

Java 包含注释@Override，它告诉编译器进行检查，来确定其后的方法是否覆盖了祖先类中的方法，如果不是，编译器就发出错误消息。

与 C++一样，Java 也要求在调用子类构造函数之前，先调用父类的构造函数。如果参数要传递给父类的构造函数，就必须显式地调用该构造函数，如下：

```
super(100, true);
```

如果没有显式地调用父类的构造函数，则编译器将插入一个对父类无参数构造函数的调用。

Java 不支持 C++中的私有和受保护的派生方式。Java 程序员相信，子类应是子类型，但支持私有和受保护的派生方式时，子类不是子类型。因此，Java 不包含这些派生方式。Java 的早期版本包含一个集合 Vector，它包含了许多操作集合结构中的数据的方法。Java 的这些版本还包含 Vector 的一个子类 Stack，它添加了执行压入和弹出操作的方法。可惜，Java 没有私有继承方式，所以 Vector 的所有方法在 Stack 类中也是可见的，于是 Stack 对象可能执行许多使这些对象无效的操作。

Java 只直接支持单继承。然而，它包含了一种抽象类，称为接口，它提供了对多继承的部分支持[8]。接口的定义与类的定义相似，但接口只能包括命名常量和方法的声明(而不是定义)。它不能包含构造函数或者非抽象方法。因此，顾名思义，接口仅定义了类的说明(回忆一下 C++抽象类，它可以有实例变量，可以完全定义除了一个方法之外的所有方法)。类不继承接口，它可实现接口。事实上，类可以实现任意数量的接口。为了实现接口，类必须实现在接口中定义的所有方法。

接口可以用来模拟多继承。一个类可以从另一个类中派生，并实现接口，接口替代了第二个父类，有时这称为混合继承，因为接口的常量和方法与从超类继承来的方法和数据、子类中定义的新数据和/或方法混合在一起。

接口的另一个更有趣的能力是它们提供了另一种多态性。这是因为可以将接口视为类型。例如，方法可以指定接口形参，这种形参可接受任何实现了该接口的类的实参，从而使方法具有多态性。

非参数变量也可以声明为接口类型。这种变量可以引用任何实现了该接口的类的对象。

一个类派生于两个父类，且都用相同的名称和协议定义公有方法时，就会发生多继承引发的一个问题。此问题可通过接口来避免。尽管实现接口的类必须为接口指定的所有方法提供定义，但如果父类和接口都包含了同名和同协议的方法，类就不需要实现该方法。所以可能发生在多继承的名字冲突，就不可能发生在单继承和接口上。而且，变量同名冲突可以完全避免，因为接口不能定义变量。

接口并不是多继承的替代，因为在多继承中有代码重用，而接口不提供代码重用。这是一个重要的差别，因为代码重用是继承的一个主要优点。Java 提供了一种方式，来部分避免这个缺点。已实现的接口可以由抽象类来替代，抽象类可以包含能继承的代码，因此

8. Java 接口类似于 Objective-C 协议。

提供了一些代码重用。

用接口替代多继承的一个问题是，如果一个类尝试实现两个接口，而这两个接口定义了同名和同协议的方法，就无法在类中实现这两个接口。

作为接口的一个例子，考虑 Java 标准类 Array 的 sort 方法。任何使用这个方法的类都必须提供该方法的一个实现版本，来比较要排序的元素。Comparable 泛型接口为这个比较方法 compareTo 提供了协议。Comparable 接口的代码如下：

```
public interface Comparable <T> {
   public int compareTo(T b);
}
```

如果调用 compareTo 方法的对象在参数对象之前，则这个方法一定返回负整数；如果相等，则返回 0；如果调用 compareTo 方法的对象在参数对象之后，这个方法就返回正整数。实现 Comparable 接口的类可对所有泛型类型的对象数组进行排序，只要实现了泛型类型的 compareTo 方法，且提供了合适的值。

除了接口之外，Java 还支持抽象类，它类似于 C++抽象类。Java 抽象类的抽象方法用方法头表示，其中包含 abstract 保留字。抽象类也标记上 abstract。当然，抽象类不能实例化。

第 14 章描述了 Java 事件处理中的接口用法。

12.7.3　动态绑定

在 C++中必须将方法定义为虚拟的，才能进行动态绑定。而在 Java 中，所有方法调用都是动态绑定的，除非被调用的方法已定义为 final，在这种情况下，这个方法不能被覆盖，所有绑定都是静态的。如果方法是 static 或 private 的，也要使用静态绑定，static 或 private 的方法不能被覆盖。

12.7.4　被嵌套的类

Java 有多种被嵌套类，这些类的优点之一是，除了嵌套它们的类以外，对于包中的其他类，被嵌套类是不可见的。直接嵌套在另一个类中的非静态类称为内部类，内部类的每个实例都有一个指向嵌套类实例的隐式指针，这个指针允许被嵌套类的方法访问嵌套类中的所有成员，包括私有成员。静态被嵌套类则没有这个指针，因此不能访问嵌套类中的成员。所以，Java 中的静态被嵌套类和 C++中的被嵌套类非常相似。

在静态作用域的语言中，内部类的成员，甚至是私有成员，都可以在外部类中访问，这似乎有点古怪。这种引用必须包含引用内部类对象的变量。例如，假定外部类用如下语句创建了内部类的一个实例：

```
myInner = this.new Inner();
```

于是，如果内部类定义了一个变量 sum，它就可以在外部类中引用为 myI-nner.sum。

被嵌套类的实例只能存在于嵌套类的实例中。被嵌套类也可以是匿名的。虽然匿名嵌套类的语法比较复杂，却是定义只用在一处的类的一种简略方法。第 l4 章有一个匿名嵌套

类的例子。

局部被嵌套的类可定义在嵌套类的方法中，局部嵌套类不需要用访问修饰符(private 或 public)来定义，它们的作用域总是限定在它们的嵌套类内。局部嵌套类中的方法，可以访问定义在嵌套类中的变量和定义在其方法中的 final 变量。局部嵌套类的成员只对定义局部嵌套类的方法可见。

12.7.5 评价

Java 的支持面向对象编程的语言设计和 C++相似，但是 Java 更一致地遵循面向对象的原则。Java 不允许建立无父类的类，它使用动态绑定做为一种“通常”的方式将方法的调用绑定于方法的定义。当然，与许多方法绑定是静态的语言相比，这会略微增加执行时间。但在做出这个设计选择时，大多数 Java 程序都是解释性的，所以解释时间使增加的绑定时间并不明显。C++访问控制比较复杂(从派生控制到友元函数)，而 Java 对于类定义内容的访问控制相对简单。最后，Java 使用接口来提供对多继承的支持，它没有多继承的所有缺点。

12.8 C#对面向对象编程的支持

C#对面向对象编程的支持和 Java 类似。

12.8.1 一般特征

C#包含类和结构，其中 C#类和 Java 中的类非常相似，而结构是能力稍差的堆动态结构。一个重要的区别是，结构是值类型，即它们是堆动态的，这会导致对象切片问题，但把结构限制为不能派生子类，就可以避免该问题。C#结构与其类的区别参见第 11 章。

12.8.2 继承

C#使用 C++的语法定义类。例如，

```
public class NewClass : ParentClass { ... }
```

从父类继承的方法可以在派生的子类中被替换，这可通过在子类中用 new 标记其定义来实现。new 方法将父类中同名的方法隐藏起来，但在方法名上加以前缀 base，就可以调用父类中的方法。例如：

```
base.Draw();
```

C#对接口的支持与 Java 相同。

12.8.3 动态绑定

在 C#中，为了将方法调用动态地绑定于方法，必须特别地标记基类方法和它在派生类中的对应方法。与 C++一样，必须将基类方法标记上 virtual。为了弄清楚与祖先类中的虚

方法具有相同名称和协议的子类方法的内容，C#要求将该方法标记上 override(假设它们要覆盖父类的虚方法)[9]。例如，12.5.3 节的 C++中 Shape 类的 C#版本如下：

```
public class Shape {
  public virtual void Draw() { ... }
  ...
}
public class Circle : Shape {
  public override void Draw() { ... }
  ...
}
public class Rectangle : Shape {
  public override void Draw() { ... }
  ...
}
public class Square : Rectangle {
  public override void Draw() { ... }
  ...
}
```

C#包含的抽象方法和 C++相似，但使用不同的语法来说明。例如，下面是一个 C#抽象方法：

```
abstract public void Draw();
```

包含至少一个抽象方法的类是抽象类，每个抽象类都必须标记为 abstract。抽象类不能实例化。所以抽象类的任何子类要实例化，都必须实现其继承的所有抽象方法。

与 Java 一样，所有的 C#类最终都派生于一个基类 Object。Object 类定义了一系列方法，包括 ToString、Finalize 和 Equals，所有 C#的类型都继承这些方法。

12.8.4　被嵌套的类

C#类可以直接嵌套在另一个类中，被嵌套类的行为与 Java 中的静态嵌套类(就像 C++中的嵌套类一样)相似。与 C++一样，C#不支持类似 Java 中非静态嵌套类的被嵌套类。

12.8.5　评价

因为 C#语言是最新的基于 C 的面向对象语言，它的设计者向前辈学习，复制了前辈的成功，还解决了一些问题。其结果是，在对面向对象编程的支持上，C#与 Java 的差别相当小，但存在 Java 中的少许问题。C#允许使用结构，而 Java 不允许，这是一个进步。与 Java 一样，C#对面向对象编程的支持比 C++简单，这也是一个进步。

---

9. 在 Java 中，这可以用注释@Override 来指定。

12.9 Ada 95 对面向对象编程的支持

Ada 95 源于 Ada 83，并进行了一些重要的扩展。本节将简略探讨 Ada 95 支持面向对象编程的扩展。因为 Ada 83 已经包括了构造抽象数据类型的结构，所以 Ada 95 需要添加的特性就是支持继承与动态绑定。Ada 95 的设计目的是对 Ada 83 的类型和包结构进行尽量少的改变，保留尽量多的静态类型检查。注意 Ada 95 中的面向对象编程比较复杂，本节仅作简要介绍，并不完整。

12.9.1 一般特征

Ada 95 中的类是一种新类型，称为标志类型(tagged types)，它们可以是记录或私有类型。它们在包中定义，所以可以独立编译。之所以将这种类命名为标志类型，是因为标志类型的每个对象都隐式地包含一个指示其类型的、由系统维护的标志。定义标志类型的操作的子程序与类型声明位于同一个声明列表中。考虑下面的例子：

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
package Person_Pkg is
  type Person is tagged private;
  procedure Display(P : in Person);
  private
    type Person is tagged
      record
        Name : Unbounded_String;
        Address : Unbounded_String;
        Age : Integer;
      end record;
end Person_Pkg;
```

这个包定义了类型 Person，这个类型本身很有用，也可以用作其他派生类的父类。

与 C++不同的是，在 Ada 95 中不能隐式调用构造函数和析构函数。可以编写这样的子程序，但它们必须由程序员显式地调用。

12.9.2 继承

Ada 83 中派生类型和子类型的继承是相当有限的。在这两种继承中，可以在已有类型的基础上定义新的类型，唯一允许的修改是限制新类型的值范围。这不是面向对象编程所需要的完全继承，但 Ada 95 支持这种继承。

Ada 95 中的派生类型基于标志类型。把新的实体放在一个记录定义中，就可将新实体增加到被继承的实体中。考虑下面这个例子：

```
with Person_Pkg; use Person_Pkg;
package Student_Pkg is
  type Student is new Person with
    record
```

```
      Grade_Point_Average : Float;
      Grade_Level : Integer;
   end record;
  procedure Display(St : in Student);
end Student_Pkg;
```

在这个例子中，派生类型 Student 定义为具有其父类 Person 的实体以及新的实体 Grade_Point_Average 和 Grade_Level。这个例子也重新定义了过程 Display。这个新类在一个独立的包中定义，这样修改它时，不需要重新编译包含其父类型定义的包。

这种继承机制没有阻止父类的实体包含在派生类中，于是，派生类只能扩充父类，因此是子类型。然而，下面简要讨论的子库包可以用来定义不是子类型的子类。

假设有如下的定义：

```
P1 : Person;
S1 : Student;
Fred : Person := (To_Unbounded_String("Fred"),
                  To_Unbounded_String("321 Mulberry
                    Lane"), 35);
Freddie : Student :=
   (To_Unbounded_String("Freddie"),
    To_Unbounded_String("725 Main St."),
   20, 3.25, 3);
```

因为 Student 是 Person 的子类型，所以如下赋值语句应该是合法的：

```
P1  :=  Freddie;
```

在必需的强制转换中，会省略 Freddie 的两个实体 Grade_ Point_Average 和 Grade_Level。这是对象切片的又一个示例。

显然，现在的问题是，相反方向的赋值是否合法。也就是说，能将 Person 赋给 Student 吗？在 Ada 95 中，这个操作是合法的，因为它包含子类中的实体。在上例中，下面的语句是合法的：

```
S1 := (Fred, 3.05, 2);
```

Ada 95 不提供多继承。尽管泛型类和多继承是两个十分不同的概念，但是使用泛型也可以达到多继承的效果，但这个方法不如 C++方法优雅，这里不讨论这个问题。

12.9.3　动态绑定

Ada 95 提供了标志类型中过程调用和过程定义的动态绑定和静态绑定。使用类范围的类型来强制进行动态绑定，这种类型代表了根为某一类型的一个类层次中的所有类型。每个标志类型都隐含地具备一个类范围类型。对于标志类型 T，这种类范围类型用 T'class 来指定。如果 T 是一个标志类型，那么 T'class 类型的变量可以保存类型 T 的对象或者从 T 中派生而来的任何类型的对象。

再考虑 12.9.2 节中定义的 Person 和 Student 类。假设有一个 Person'class 类型的变量 Pcw，这个变量有时引用 Person 对象，有时引用 Student 对象。进一步假设，我们想要显示 Pcw 引用的对象，不论它是引用 Person 对象还是 Student 对象。这就要求将对 Display 的调用动态地绑定到正确的 Display 版本上。为此，可以使用带 Person 类型参数的新过程，并将它传送给 Display。下面就是这样的一个过程：

```
procedure Display_Any_Person(P: in Person) is
  begin
  Display(P);
  end Display_Any_Person;
```

这个过程可以用下面两种方式来调用：

```
with Person_Pkg; use Person_Pkg;
with Student_Pkg; use Student_Pkg;
P : Person;
S : Student;
Pcw : Person'class;
...
Pcw := P;
Display_Any_Person(Pcw); -- call the Display in Person
Pcw := S;
Display_Any_Person(Pcw); -- call the Display in Student
```

Ada 95+还支持多态指针，这些多态指针定义为具有类范围的类型，例如

```
type Any_Person_Ptr is access Person'class;
```

在 Ada 95+中，将保留字 abstract 包括在类型定义和子程序定义中，就可以定义纯抽象基类型。此外，子程序定义不能有定义体。考虑下面的例子：

```
package Base_Pkg is
  type T is abstract tagged null record;
  procedure Do_It (A : T) is abstract;
end Base_Pkg;
```

12.9.4 子包

包可以直接嵌套进其他包里，此时它们称为子包。这种设计的一个潜在问题是，如果一个包有大量的子包，而且这些子包都很大，那么这个嵌套包就会很大，以致不能成为有效的编译单元。这个问题的解决方法比较简单：允许子包成为物理上独立的单元(文件)，可以独立编译，此时子包称为子库包。

将保留字 private 放在保留字 package 之前，就可以将子包声明为私有的。私有子包的逻辑位置是嵌套包的包规范的声明部分的开头。私有子包的声明对嵌套包体是不可见的，除非嵌套包包括了带有该子包名字的 with 字句。

子包的一个重要特点是，甚至父包的私有部分对子包也是可见的。因为父实体的这种可见性，子包为类的派生提供了另一种方式。所以，父包的私有部分类似于父类中的受保

护成员，而子包用于扩展类。

子库包可以随时添加到程序中，它们不需要重新编译父包或父包的客户程序。

子库包可以用于替代 C++中的友元定义。例如，如果必须编写一个子程序，来访问两个不同类的成员，就可以在父包中定义一个类，在子包中定义另一个类。接着，子包中的子程序就可以访问两个类的成员。而且，在 C++中，如果定义类时并不知道需要一个友元，那么在发现需要这个友元时，就需要修改类，并重新编译。在 Ada 95+中，新类可以在新子包中定义，而不会影响父包，因为父包中定义的每个名字在子包中都是可见的。

12.9.5　评价

Ada 对面向对象编程提供了全面的支持，尽管其他面向对象语言的使用者可能会感觉 Ada 的支持很弱，还有些复杂。虽然包可以用来创建抽象数据类型，但是它们实际上是更一般化的封装结构。除非使用子库包，否则无法限制继承，在这种情况下，所有的子类都是子类型。与 C++、Java 和 C#相比，这种形式的访问限制是有限的。

C++显然提供了一种比 Ada 95 更好的多继承形式。然而，使用子库单元来控制对父类实体的访问，似乎是比 C++的友元函数和类更清楚的解决方式。

C++包含用来初始化对象的构造函数和析构函数，这是十分优良的设计，但 Ada 95 不包括这样的功能。

这两种语言的另一个不同之处，是 C++根类的设计者必须决定某个成员函数是静态绑定还是动态绑定。如果选择的是静态绑定，而系统后来又需要改变为动态绑定，就必须改变根类。在 Ada 95 中，这种设计决定不必与根类的设计联系在一起。每个调用本身可以确定是静态绑定还是动态绑定，不管根类是如何设计的。

12.10　Ruby 对面向对象编程的支持

如前所述，与 Smalltalk 类似，Ruby 是一种纯面向对象的编程语言。该语言中的几乎所有东西都是对象，通过消息传递来完成所有计算。虽然程序中的表达式使用中缀操作符，与 Java 语言中的表达式类似，但实际上这些表达式是通过消息传递来计算的。与 Smalltalk 类似，计算 a+b 时，它将消息+发送给 a 引用的对象，并将对象 b 的引用传递为参数。换言之，a+b 实现为 a.+b。

12.10.1　一般特征

Ruby 类定义与 C++和 Java 等语言不同，因为它们是可执行的。这允许它们在执行期间保持开放性。程序可以在类中添加成员任意多次，为此，只需要提供包含新成员的类的二次定义。在执行期间，类的现有定义就是已经执行的类的所有定义的联合。方法定义也是可执行的，这允许程序在执行期间，通过把两个方法定义放入选择结构的 then 和 else 子句来选择一个方法定义。

Ruby 里所有的变量都是对象的引用，而且都是无类型的。所有实例变量名都以符号@

开头。

Ruby 与其他常用编程语言的明显区别是，Ruby 的访问控制在访问数据和访问方法上不同。所有实例数据都默认为私有访问，且不能改变。如果需要对实例变量进行外部访问，就必须定义访问方法。例如下面的类定义框架：

```
class MyClass


# A constructor
 def initialize
   @one = 1
   @two = 2
  end

# A getter for @one

  def one
    @one
  end

# A setter for @one

  def one=(my_one)
    @one = my_one
  end

end # of class MyClass
```

与设置方法名相连的等号(=)表明它的变量是可赋值的。因此，所有设置方法名都有相连的等号。one 获取方法体说明了用 Ruby 设计的方法没有返回语句时，就返回已求值的最后一个表达式的值。本例是返回@one 的值。

由于需要经常使用获取和设置方法，Ruby 提供了它们的快捷方法。假如类有两个实例变量的获取方法@one 和@two，那么这些获取方法就可以在类中用一个语句指定：

```
attr_reader :one, :two
```

attr_reader 实际上是使用:one 和:two 作为实参的函数调用。在变量前加冒号(:)表示要使用变量名，而不是取消其引用对象的引用。

用类似方式创建设置方法的函数称为 attr_writer。这个函数与 attr_reader 具有相同的参数列表。

创建获取方法和设置方法的函数都指定了名称，因为它们提供了类对象的协议，这称为属性。类的属性定义了类对象的数据接口(通过访问方法来访问的公有数据)。

Ruby 对象用 new 创建，new 隐式调用了构造函数。Ruby 类中的常用构造函数命名为 initialize。子类的构造函数可以初始化定义了设置方法的父类数据成员。这一过程是通过调用 super，并以初始值作为实参来实现的。super 会调用父类的方法，这个方法与调用 super 时使用的方法同名。

类变量通过在名称前加两个符号(@@)来指定，它们对类和其实例是私有的。这种私有性不能改变。另外，与全局变量和实例变量不同，类变量必须在使用前初始化。

12.10.2　继承

在 Ruby 中定义子类采用小于号(<)，而不是 C++中的冒号(:)，例如：

```
class MySubClass < BaseClass
```

Ruby 的方法访问控制有一个显著的不同，那就是只要调用访问控制函数，就能在子类中改变它们。这意味着可以定义基类的两个子类，使一个子类的对象可以访问基类定义的方法，但是另一个子类的对象不能。这也允许改变基类中公有方法对子类私有方法的访问，这种子类明显不能是子类型。

Ruby 模块提供了一种命名封装，它常常用于定义函数库。然而，也许模块最有趣的一面是能从类中直接访问它们的函数。在类中访问模块用 include 语句来指定。例如：

```
include Math
```

包含模块的效果是类获得指向模块的指针，并有效地继承定义在模块中的函数。实际上，类包含模块时，模块就变成了类的一个代理超类。这样的模块称为混合类型(mixin)。

12.10.3　动态绑定

Ruby 对动态绑定的支持与 Smalltalk 相同。变量不能类型化，但它们是对任何类对象的引用。因此，所有变量都是多态的，并且所有对方法调用的绑定都是动态的。

12.10.4　评价

因为 Ruby 是最纯的面向对象编程语言，所以它对面向对象编程的支持是充足的。然而，它对类成员的访问控制比 C++弱，也不支持抽象类或接口，尽管 mixin 与接口紧密相关。最后，主要是因为 Ruby 是解释性的，其执行效率比编译性语言糟糕得多。

12.11　面向对象构造的实现

至少有两种支持面向对象程序设计的语言向语言实现者提出了有趣的问题：实例变量的存储结构和消息对方法的动态绑定。本节将简要讨论这两个问题。

12.11.1　存储实例数据

在 C++中，类定义为 C 记录结构的扩展。它们之间的相似性说明，可以将记录结构作为类实例变量的储存结构。这种形式的结构称为类实例记录(CIR)。CIR 的结构是静态的，因此它在编译时建立，用作创建类实例数据的模板。每个类都有自己的 CIR。派生一个类时，子类的 CIR 是父类的副本，新的实例变量项添加到末尾。

因为 CIR 的结构是静态的，所以对所有实例变量的访问可以像记录那样，使用从 CIR 实例开始的常量偏移值。于是，这些访问与访问记录的域一样高效。

12.11.2 方法调用到方法的动态绑定

静态绑定类中的方法并不需要涉及这个类的 CIR。然而动态绑定的方法必须在 CIR 结构中有入口。这种入口可以是指向方法代码的指针，它必须在创建对象时建立。然后，方法的调用可以通过 CIR 中的这个指针联系对应的代码。这种技术的缺点是，每个实例都需要存储所有动态绑定方法的指针，这些动态绑定方法可能从这个实例中调用。

注意，可以从类的一个实例中调用的动态绑定方法列表对于这个类的所有实例都是相同的。因此这个方法列表只需要储存一次。因而一个实例的 CIR 只需要一个指针来指向这个列表，以找到被调用的方法。这个列表的存储结构常称为虚方法表(vtable)。方法的调用可以使用 vtable 中的偏移来表示。祖先类的多态变量总是引用正确类型对象的 CIR，从而确保得到动态绑定方法的正确版本。考虑下面的 Java 示例，其中所有的方法都是动态绑定的：

```
public class A {
  public int a, b;
  public void draw() { ... }
  public int area() { ... }
}
public class B extends A {
  public int c, d;
  public void draw() { ... }
  public void sift() { ... }
}
```

图 12-7 显示了类 A 与类 B 的 CIR 及其 vtable。注意在 B 的 vtable 中，指向 area 方法的指针指向 A 的 area 方法代码。因为 B 没有覆盖 A 的 area 方法，所以，如果 B 的客户程序调用 area，就调用从 A 继承的 area 方法。另一方面，在 B 的 vtable 中，draw 和 sift 的指针指向 B 中的 draw 和 sift。draw 方法在 B 中被覆盖，并在 B 中增加 sift 的定义。

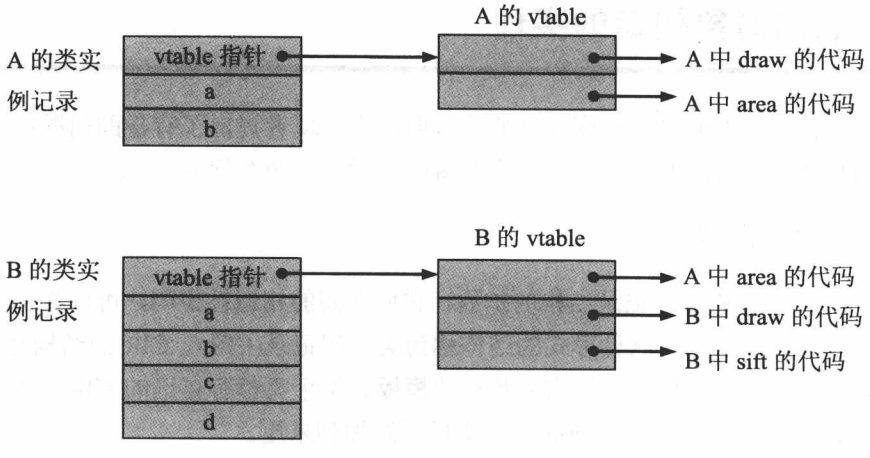


图 12-7 一个单继承 CIR 的例子

多重继承使动态绑定的实现复杂化，考虑下面的三个 C++类定义：

```
class A {
  public:
    int a;
    virtual void fun() { ... }
    virtual void init() { ... }
};
class B {
  public:
    int b;
    virtual void sum() { ... }
};
class C : public A, public B {
  public:
    int c;
    virtual void fun() { ... }
    virtual void dud() { ... }
};
```

类 C 继承了类 A 的变量 a 和方法 init。类 C 重新定义了方法 fun.，虽然类 C 中的 fun 和父类 A 中的 fun 通过(类型 A 的)多态变量都是可见的。类 C 从类 B 继承了变量 b 和方法 sum。类 C 定义了自己的变量 c，还定义了一个非继承的方法 dud。类 C 的 CIR 不仅要包括类 A、类 B、类 C 的数据，还要包含访问所有可见方法的某种方式。在单继承的情况下，CIR 包含一个指向 vtable 的指针，vtable 包括所有可见方法的代码的地址。然而，在多继承的情况下，事情就没有那么简单。CIR 至少必须包括两个不同的可用视图，其中一个用于两个基类，另外一个包括类 C 的视图。与单继承的实现相同，在父类的视图中包括子类的视图。

必须有两个 vtable：一个用于类 A 和类 C 的视图，另一个用于类 B 的视图。在这种情况下，类 C 的 CIR 的第一部分可以是类 C 和类 A 的视图，它从一个 vtable 指针开始，而这个指针指向类 C 的方法和从类 A 继承来的方法，也包括从类 A 继承来的数据。在 C 的 CIR 中，紧接着的是 B 的视图部分，其起始于一个指向 B 的虚拟方法的 vtable 指针，接着是从 B 继承的数据以及 C 中定义的数据。类 C 的 CIR 如图 12-8 所示。

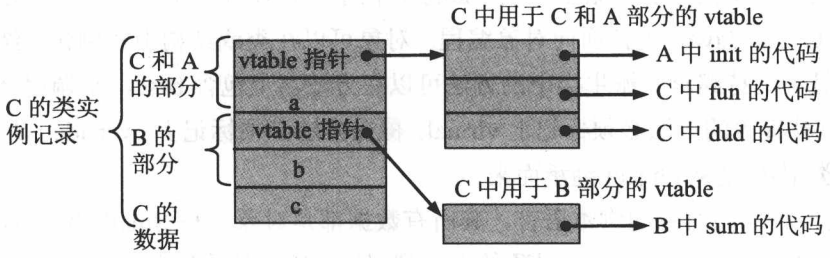


图 12-8　多继承的 CIR 子类示例

小　结

面向对象编程基于三个基本概念：抽象数据类型、继承和动态绑定。面向对象编程语

言通过类、方法、对象和消息传递，来支持面向对象范式。

本章关于面向对象编程语言的讨论主要围绕七个设计问题：纯对象、子类和子类型、类型检查和多态性、单继承和多继承、动态绑定、对象的显式或隐式释放以及嵌套类。

Smalltalk 是一种纯面向对象的语言，所有东西都是对象，所有的计算都通过消息传递来完成。在 Smalltalk 中，所有的子类都是子类型。所有的类型检查和消息对方法的绑定都是动态的，所有的继承都是单继承。Smalltalk 没有显式释放存储空间的操作。

C++提供了对数据抽象、继承和消息对方法的选择性动态绑定，以及 C 中所有传统特性的支持。这意味着它有两个不同的类型系统。C++支持多继承和显式的对象解除分配。C++包含了多种对类中实体的访问控制，其中的一些可以避免子类成为子类型。构造函数和析构函数都可以包含在类中，两者均是隐式调用的。

尽管 Smalltalk 的动态类型绑定提供了比混合式语言 C++更多的灵活性，但 Smalltalk 远不及 C++高效。

Objective-C 支持过程式和面向对象编程，它不如 C++复杂，也没有 C++应用广泛。它只支持单继承，但它有类别(支持 mixin，允许将额外的方法添加到类中)和协议(类似于 Java 的接口)。类可以采纳任意数量的协议。构造函数的名字可以任意，但必须显式调用。多态性通过预定义类型 id 来支持。id 类型的变量可以引用任何对象。通过 id 类型的变量引用的对象来调用方法时，绑定就是动态的。

与 C++不同，Java 不是混杂的语言，它只支持面向对象编程。Java 包含基本标量类型和类。所有对象都从堆中分配，并通过引用变量访问。Java 没有显式的对象释放空间的操作——使用垃圾回收机制。Java 的唯一子程序是方法，只能通过对象或类来调用。Java 直接支持的只有单继承，尽管使用接口可以实现某种形式的多继承。除了不能被覆盖的方法以外，所有消息对方法的绑定都是动态的。除了类以外，Java 还包含了包作为第二种封装结构。

Ada 95 通过标志类型提供了对面向对象编程的支持，也支持继承。动态绑定通过类范围的类型来支持。派生类型是父类型的延伸，除非这些派生类型在子库包中定义。在这种情况下，父类型的实例可以从派生类型中删除。在子库包以外，所有的子类都是子类型。

C#基于 C++和 Java，支持面向对象编程。对象可以在类或结构中实例化。结构对象是栈动态的，且不支持继承。派生类中的方法可以在方法名上包含 base，来调用父类中隐藏的方法。可以被覆盖的方法必须标记上 virtual，覆盖方法必须标记上 override。所有的类(和所有的基本类型)均是从 Object 继承而来。

Ruby 是一种面向对象的脚本语言，其所有数据都是对象。与 Smalltalk 一样，所有对象都是堆分配的，所有变量都是对对象的无类型引用。所有的构造函数都命名为 initialize。所有的实例数据都是私有的，但很容易包括获取和设置方法。已经提供访问方法的所有实例变量形成了类的共有接口。这种实例数据称为属性。Ruby 类是动态的，因为它们是可执行的，可以随时改变。Ruby 只支持单继承，其子类不一定是子类型。

类的实例变量存储在 CIR 中，其结构是静态的。子类有自己的 CIR 和父类的 CIR。动

态绑定通过虚方法表来支持，该表存储了指向特定方法的指针。多继承会极大地复杂化 CIR 和虚方法表的实现。

复 习 题

1. 简述面向对象语言的三个特点。
2. 类变量和实例变量的区别是什么？
3. 什么是多继承？
4. 什么是多态变量？
5. 什么是覆盖方法？
6. 简述一种情况，使用动态绑定比不用更好。
7. 什么是虚方法？
8. 什么是抽象方法？什么是抽象类？
9. 简述本章讨论的面向对象语言的八个设计问题。
10. 什么是嵌套类？
11. 什么是对象的消息协议？
12. Smalltalk 对象在哪里分配空间？
13. 解释 Smalltalk 消息是怎样绑定到方法上的，这种绑定何时发生？
14. 在 Smalltalk 中进行什么类型检查？何时进行？
15. Smalltalk 中支持什么类型的继承，单继承还是多继承？
16. Smalltalk 对计算有哪两种重要的影响？
17. 所有的 Smalltalk 变量实质上都是一个类型，这个类型是什么？
18. C++中的对象在哪里分配空间？
19. C++中堆分配的对象是怎样释放空间的？
20. 所有 C++的子类都是子类型吗？为什么？
21. 在什么情况下 C++的方法调用会静态绑定到方法上？
22. 允许设计人员来说明哪种方法可以静态绑定会有什么缺点？
23. 在 C++中，私有继承和公有继承有什么区别？
24. 什么是 C++中的友元函数？
25. 什么是 C++中的纯虚函数？
26. C++参数是如何传送到超类的构造函数中的？
27. Smalltalk 和 C++之间最重要的实际差别是什么？
28. 如果 Objective-C 方法没有返回值，在方法头应指定什么返回类型？
29. Objective-C 支持多继承吗？
30. Objective-C 类不能在其开头指定父类吗？
31. Objective-C 的根类是什么？

32. 在 Objective-C 中，一个方法如何指定它不能在子类中被覆盖？
33. Objective-C 类别的作用是什么？
34. Objective-C 协议的作用是什么？
35. Objective-C 中 id 类型的主要用处是什么？
36. 在类型系统上，Java 和 C++有什么差别？
37. Java 的对象是在哪里分配空间的？
38. 什么是装箱？
39. Java 的对象是如何释放空间的？
40. Java 的所有子类都是子类型吗？
41. 超类构造函数在 Java 中如何调用？
42. 在什么情况下 Java 方法调用可以静态绑定到方法上？
43. C#与 C++的覆盖方法在语法上有何不同？
44. 在 C#中，子类覆盖了从父类继承的一个方法时，如何从子类中调用这个方法的父类版本？
45. Ada 95 的所有子类都是子类型吗？
46. 如何指定 Ada 95 中对子程序的调用动态绑定到子程序定义上？这个决定是何时做出的？
47. Ruby 如何实现基本类型，如整型数据和浮点型数据？
48. 如何在 Ruby 类中定义获取方法？
49. Ruby 对实例变量支持怎样的访问控制？
50. Ruby 对方法支持怎样的访问控制？
51. Ruby 的所有子类都是子类型吗？
52. Ruby 支持多重继承吗？

习　题

1. 在 SIMULA 67 中，缺少了什么支持面向对象编程的重要内容？
2. 在何种情况下，“兼容性”能定义为被覆盖方法与覆盖方法之间的关系？
3. 比较 C++和 Java 中的动态绑定。
4. 比较 C++和 Java 中类实体的访问控制。
5. 比较 C++和 Ada 95 中类实体的访问控制。
6. 比较 C++中的多继承和 Java 中由接口提供的多继承。
7. 在哪种编程情形下，多继承明显优于接口？
8. 解释通过继承改善抽象数据类型的两个问题。
9. 描述子类可以对其父类做哪些改变。
10. 解释继承的一个缺点。
11. 解释一种语言中所有的值都是对象的优点和缺点。

12. 子类和其父类是 is-a 关系时，这是什么意思？
13. 描述覆盖方法的参数应匹配被覆盖方法的参数的问题。
14. 解释 Smalltalk 的类型检查机制。
15. 显然，Java 的设计者认为，像 C++那样由静态绑定来提高效率是不值得的。谈谈支持和反对 Java 的这种设计的理由。
16. 所有的 Java 对象都有一个共同祖先，其主要原因是什么？
17. Java 中 finalize 语句的作用是什么？
18. 如果 Java 支持栈动态对象和堆动态对象，能从中得到什么？支持这两种对象的缺点是什么？
19. 从程序设计的便利性方面，比较 Ada 95 和 C++提供多态性的方式。
20. C++抽象类和 Java 接口之间的区别是什么？
21. 比较 C++和 Objective-C 对多态性的支持。
22. 比较 Objective-C 协议和 Java 接口的功能和用法。
23. 评价 Objective-C 设计者决定使用 Smalltalk 的方法调用语法，而不是大多数支持面向对象编程的命令式语言使用的传统语法。
24. 解释为什么在 Java 和 C#中允许类实现多重接口，不会产生与 C++多重继承时的相同问题？
25. 研究和解释为什么 C#不包含 Java 的非静态内嵌类？
26. 能为抽象类定义引用变量吗？如何使用这个变量？
27. 比较 Java 和 Ruby 中实例变量的访问控制。
28. 比较 Java 和 Ruby 中实例变量的类型错误检测机制。

编程练习

1. 用 Java 重写 12.5.2 节中的 single_linked_list、stack_2 和 queue_2 类，并就可读性和编程难度与 C++版本进行比较。
2. 用 Ada 95 重写编程练习 1。
3. 用 Ruby 重写编程练习 1。
4. 用 Objective-C 重写编程练习 1。
5. 设计和实现一个 C++程序，这个程序定义一个基类 A，A 有一个子类 B，B 又有子类 C。A 必须实现一个被 B 和 C 覆盖的方法。再写一个测试类对 A、B 和 C 进行实例化，并包含这个方法的三个调用。其中一个调用必须静态绑定到 A 的方法，另一个调用必须动态绑定到 B 的方法，最后一个调用必须动态绑定到 C 的方法。所有方法的调用都必须通过指向类 A 的指针实现。
6. 用 C++写一个多次调用动态绑定方法和静态绑定方法的程序，对两种方法调用进行计时。比较计时结果，计算这两种方法调用所需的时间差，并解释这个结果。
7. 用 Java 重写编程练习 5，要求使用 final 进行静态绑定。

访 谈

关于程序设计范型和更好的程序设计

Bjarne stroustrup

Bjarne Stroustrup 是 C++的设计者和最早的实现者，也是 *The C++ Programming Language* 和 *The Design and Evolution of C++* 两本书的作者。他的研究领域包括分布式系统、仿真、设计、程序设计以及程序设计语言。Stroustrup 博士在美国 Texas 州 A&M 大学工学院任计算机科学教授。他积极参与了 C++的 ANSI/ISO 标准化工作。在 AT&T 工作了二十多年后，Stroustrup 博士仍然保持着与 AT&T 实验室的联系，作为信息和软件系统研究实验室的成员开展研究工作。他是 ACM 的高级会员，AT&T 贝尔实验室高级会员以及 AT&T 的高级成员。1993 年，“因为他的早期工作为 C++程序设计语言奠定了基础，在这些基础上，经过 Stroustrup 博士的持续努力，C++已经成为计算史上最有影响力的程序设计语言之一”，Stroustrup 荣获了 ACM 的 Grace Murray Hopper 奖。

程序设计范型

请谈谈你对于面向对象范型的看法：它的优点和缺点。

首先我想说一下我的“面向对象程序设计”是什么意思——许多人认为“面向对象”就是“好”的代名词。如果是这样，就不需要其他范型了。“面向对象”的关键，是类层次结构的使用，它通过一些大体相同的虚函数来提供多态的行为。要正确使用“面向对象”，很重要的一点是要避免直接访问这种层次结构中的数据，应该只使用设计良好的函数接口。

除了一些众所周知的优点，面向对象程序设计也有明显的不足。特别是，并非每一个概念都天然地适合于类层次结构，而且支持面向对象的机制比起其他方法来会显著地增加开销。对于很多简单的抽象，那些不依赖于类层次和运行时绑定的类反而能提供更简单和高效的选择。除此以外，在不需要运行时解析的地方，基于(编译时)参数化多态的通用程序设计将是一个表现更好且效率更高的设计方法。

那么，C++是“面向对象”的，还是其他什么？

C++支持多种范型，包括面向对象程序设计，通用程序设计以及过程化程序设计。这些范型的组合定义了多范型程序设计，即支持一种以上的程序设计风格(“范型”)以及这些风格的组合。

你能不能举一个多范型程序设计的小例子？

考虑“形状的集合”这个经典例子的一个变种(它源自早期第一种支持面向对象编程的语言 Simula 67)：

```
void draw_all(const vector<Shape*>& vs)
{
```

```
    for (int i = 0; i<vs. size(); ++i)
        vs[i]->draw();
}
```

这里同时用了泛型容器 vector 和多态类型 Shape。vector 提供了静态的类型安全和最佳的运行性能。Shape 提供了不必重新编译就能处理形状(即任何 Shape 派生类的对象)的能力。

很容易地将上述代码一般化为任何满足 C++标准库要求的容器：

```
template<class C>
        void draw_all(const C& c)
{
    typedef typename C::
        const_iterator CI;
    for (CI p = c. begin();
        p!=c. end(); ++p)
        (*p)->draw();
}
```

使用迭代器允许把这个 draw_all()函数应用在不支持下标的容器中，例如标准库的链表：

```
vector<Shape*> vs;
list<Shape*> ls;
// ...
draw_all(vs);
draw_all(ls);
```

甚至还可以将它进一步一般化，来处理任何由一对迭代器定义的元素序列：

```
template<class Iterator> void
draw_all(Iterator b, Iterator e)
{
    for_each(b,e,mem_fun(&Shape::draw));
}
```

为了简化实现，我使用了标准库算法 for_each。

可以对标准库链表和数组来调用这个最后版本的 draw_all()。

```
list<Shape*> ls;
Shape* as[100];
// ...
draw_all(ls. begin(),ls. end());
draw_all(as,as+100);
```

为工作选取“合适的”语言

具备多种程序设计范型的背景知识有什么用呢？或者说，是否应进一步熟悉面向对象语言，而不是学习其他范型？

要成为软件领域的专家，掌握多种语言和多种程序设计范型是必不可少的。当前，C++是最好的多范型程序设计语言，也是学习各种形式的程序设计方法的一种好语言。然而，

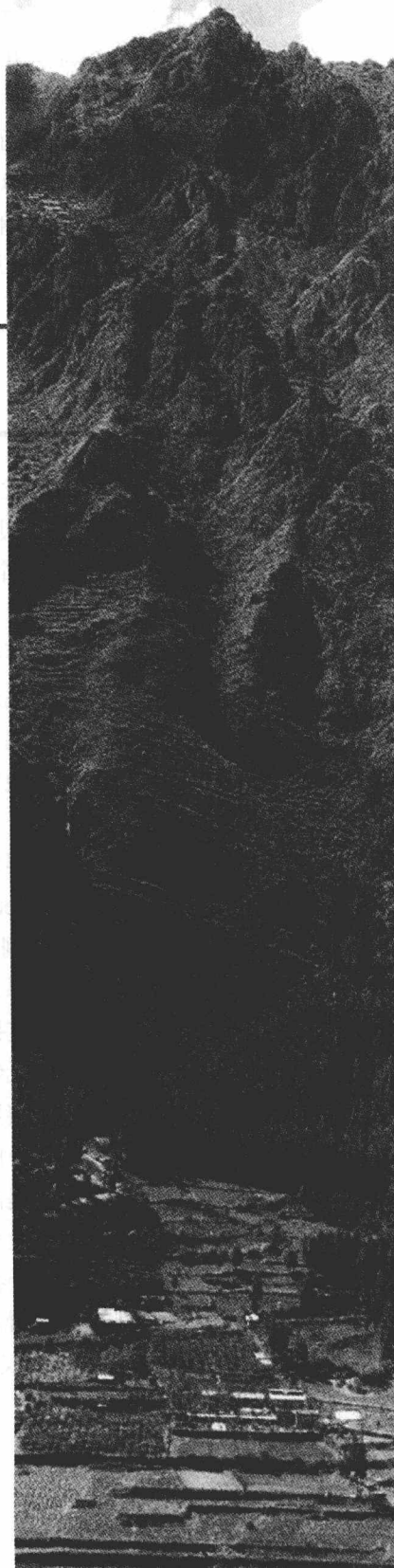
只了解 C++是不够的，更不用说只了解单一范型的语言了。这有点像色盲和只会一种语言的人：很难知道自己遗漏了什么。许多好的程序设计灵感来自于已学习和领悟的多种程序设计风格，而且知道如何将这些风格应用于不同的语言。

而且我认为，要设计重要的程序，都必须具备坚实和宽广的教育背景，而不能仅受过匆忙、简单的“速成培训”。

第13章

并　　发

13.1 概述
13.2 子程序级并发概述
13.3 信号量
13.4 管程
13.5 消息传递
13.6 Ada 对并发的支持
13.7 Java 线程
13.8 C#线程
13.9 函数式语言中的并发
13.10 语句级并发



本章首先介绍在子程序(单元)级和语句级的各种并发，包括对常见的各类多处理器计算机体系结构的简要介绍。然后详细介绍单元级并发，先介绍一些基本概念(在讨论对单元级并发提供语言支持所面临的问题与挑战之前，必须理解这些概念)，特别是竞争同步与合作同步，再介绍提供并发语言支持的设计问题，接下来详细介绍语言支持并发的三种主要解决方案：信号量、管程和消息传递。我们用一个伪代码例程来演示如何使用信号量，用 Ada 和 Java 语言来演示管程，用 Ada 来演示消息传递。我们对 Ada 语言支持并发的特性有一些详细的介绍，虽然主要关注任务，但也介绍受保护对象(实际上是管程)。接着介绍 Java 和 C#使用线程对单元级并发的支持，包括同步的方法。之后简要介绍几种函数式语言对并发的支持。本章最后一节是对语句级并发的简要介绍，包括高性能 Fortran 对语句级并发的部分语言支持。

13.1 概述

软件执行时的并发在四个不同层次上：指令级(同时执行两条或更多条机器指令)、语句级(同时执行两条或更多条高级语言语句)、单元级(同时执行两个或更多的子程序单元)和程序级(同时执行两个或更多的程序)。本章不介绍指令级和程序级并发，因为它们与语言设计问题无关。本章介绍子程序级和语句级并发，主要关注子程序级并发。

乍看起来，并发不过是个简单的概念，但是它对程序员、编程语言设计者和操作系统设计者提出了巨大的挑战(因为对并发的许多支持都由操作系统提供)。

并发控制机制增加了编程的灵活性。最初发明并发控制是为了解决操作系统所面对的特定问题，但各种其他的程序设计应用也需要并发控制。目前最常用的程序之一是网页浏览器，其设计主要基于并发。浏览器必须同时执行许多不同的函数，包括在网页服务器上发送和接收数据，在屏幕上显示文本和图像，对用户的鼠标和键盘动作给予响应。一些现代浏览器，例如 Internet Explorer 9，使用额外的核心处理器(是许多现代个人计算机的一部分)来完成一些处理，例如解释客户端脚本代码。另一个例子是设计用于模拟实际物理系统的软件系统，该物理系统由多个并发的子系统组成。对于这类应用，编程语言(或库，或者至少是操作系统)必须支持单元级并发。

语句级并发与单元级并发的差别很大。从语言设计者的角度来看，语句级并发的主要问题是说明应该如何将数据分布在多处内存上，哪些语句可以并发执行。

开发并发软件的目的是生成可伸缩、可移植的并发算法。如果有更多的处理器，并发算法的执行速度就能提高，这样的并发算法是可伸缩的，这一性质的重要性在于，每一代新计算机的处理器数量都在增长。算法必须是可移植的，因为硬件的生存期相对较短，因此软件系统不应该基于特定的体系结构，即它们应能在不同体系结构的计算机上高效运行。

本章的目的是介绍并发中与语言设计问题最相关的方面，而不是对所有并发问题(包括并发程序的开发)给出权威的研究结果，这显然不适于一本关于编程语言的书。

13.1.1 多处理器体系结构

大量不同的计算机体系结构都有一个以上处理器，能够支持某种形式的并发执行。在开始介绍程序与语句的并发执行前，先简要介绍一些这样的体系结构。

第一批具有多处理器的计算机有一个通用处理器，以及一个或多个其他处理器，通常称为外围处理器，只用于输入和输出操作。这种体系结构使计算机(产生于 20 世纪 50 年代晚期)在执行一个程序的同时，能执行其他程序的输入或输出操作。

到 20 世纪 60 年代早期，有一些计算机具有多个完整的处理器，这些处理器由操作系统的作业调度程序使用，将批处理作业队列中的各个作业分配到各个处理器上。这种结构的系统支持程序级并发。

20 世纪 60 年代中期出现的计算机具有几个相同的部分处理器，处理指令流中特定的指令。例如，一些计算机有两个或更多的浮点乘法器，另一些计算机有两个或更多的完整的浮点算术单元。这些计算机的编译器需要确定哪些指令可以并发执行，并据此调度这些指令。这种结构的系统支持指令级并发。

1966 年，Michael J. Flynn 提出，根据指令和数据流是一个还是多个，对计算机体系结构进行分类。这些名称从 20 世纪 70 年代到 21 世纪早期被广泛使用。使用多数据流的两类体系结构定义如下：多个处理器同时在不同数据上执行相同指令的计算机，称为单指令多数据(SIMD)体系结构的计算机。在 SIMD 计算机中，每个处理器都有自己的局部内存。有一个处理器控制其他处理器的操作。因为除了控制处理器外，所有的处理器在同一时间执行相同的指令，所以不需要软件的同步。使用最广泛的 SIMD 计算机可能是称为矢量处理器的一类计算机，它们有多组寄存器存储矢量运算的操作数，在这些寄存器中对整组操作数同时执行相同的指令。最初，从这种体系结构获益最大的程序主要是在科学计算领域中，这一计算领域通常是多处理器计算机的目标应用领域。但是，现在 SIMD 处理器用于各种应用领域，包括图形和视频处理。直到最近，大多数超级计算机都是矢量处理器。

如果计算机包含多个处理器，这些处理器独立地操作，且这些操作是同步进行的，那么这种计算机就称为多指令多数据(MIMD)计算机。MIMD 计算机中的每一个处理器都执行自己的指令流。MIMD 计算机有两种不同的配置：分布式的系统和共享存储的系统。分布式的 MIMD 计算机中，每一个处理器都有自己的内存，可以建在一个底盘上，或者分布在一个大的区域。共享存储的 MIMD 计算机显然必须提供某种同步的方法来防止存储访问冲突。即使分布式的 MIMD 计算机也需要同步来一起处理单个程序。MIMD 计算机比 SIMD 计算机更通用，支持单元级并发。本章主要讨论共享存储的 MIMD 计算机的语言设计，它常常称为多处理器。

随着功能强大且价格低廉的单片机的出现，在一个底盘上将大量这种微处理器连接成小型网络成为可能。这类计算机通常采用现成的微处理器，一些不同的生产厂商都有生产。

软件并没有发展到可以直接使用并发计算机，一个重要原因是处理器的性能在不断提高。使用并发计算机的一个最大动力是提高计算速度，然而有两个硬件因素结合在一起提供了更快速的计算，而不需要改变软件系统的体系结构。第一，处理器的时钟频率随着每

一代新处理器的出现(大约每 18 个月就出现新的一代)变得更快；第二，在处理器的结构中已经内建了一些不同类别的并发功能，其中包括指令和数据由内存到处理器的流水线操作(在当前指令执行时获取并解码其他指令)，对指令和数据采用独立的流水线，指令和数据的预取，以及算术运算的并行执行。所有这些统称为隐式并发。提高执行速度的结果就是不需要软件开发人员实现并发的软件系统，就能大大增加计算能力。

不过现在这种情况已经改变了，单个处理器速度的连续显著提升目前已经接近尾声，计算能力的显著提高源自于处理器数量的显著增加，例如由 Google 和 Amazon 允许的大型服务器系统和科学研究应用程序。许多其他的大型计算任务现在运行在包含大量小型处理器的计算机上。

最近，计算硬件的另一个进步是单个芯片上多个处理器的开发，例如 Intel Core Duo 和 Core Quad 芯片，给予软件开发人员更多的压力来更多地利用现有的多处理器计算机。如果软件开发人员不这么做，就浪费了并发的硬件，无法显著增加计算能力。

13.1.2 并发的种类

有两类不同的并发单元控制。最自然的一类并发是，假设有一个以上的处理器，同一程序的几个程序单元完全同时执行，这就是物理并发。将这一并发的定义稍微放宽一些，允许程序员和应用软件假设有多个处理器提供真正的并发，然而实际的程序执行是在单个处理器上以分时方式实现，这就是逻辑并发。从程序员和语言设计者的角度来看，逻辑并发与物理并发是相同的。语言实现者应使用底层操作系统的功能，将逻辑并发映射到主机硬件上。逻辑并发和物理并发都允许将并发的原理用作程序设计方法学。本章下面的部分，将介绍物理并发和逻辑并发。

使程序的执行流程可视化的一个有效方法是，想象在程序的源文本语句上放一条线。特定执行过程到达的每一条语句都由表示执行的线所覆盖。看着线通过源程序，就可以跟踪执行流程经过程序的可执行版本。当然，除非是最简单的程序，这条线都会经过一个非常复杂的路径，不可能可视化地跟踪。正规的说法是，程序的控制线就是控制流经程序时，所到达的程序点的序列。

具有协同程序(参见第 9 章)、没有并发子程序的程序只有一条控制线，虽然有时称为准并发。物理并发执行的程序有多条控制线，每一个处理器可以执行一条控制线。虽然逻辑并发的程序执行可能实际上只有一条控制线，但只要想象这类程序有多条控制线，就可以设计和分析它们。设计为有一条以上控制线的程序，称为多线程程序。当一个多线程程序在一台单处理器的计算机上执行时，它的多个线程被映射为一个线程，在这种情况下，它变成一个虚拟的多线程程序。

语句级并发是一个相对简单的概念。语句级并发的一个常见用法是：包含操作数组元素的语句的循环，可以拆开，使数据处理分布到多个处理器上。例如，一个执行 500 次且包含的语句操作 500 个数组元素之一的循环，可以拆开，使 10 个不同的处理器同时处理 50 个数组元素。

13.1.3 使用并发的目的

设计并发的软件系统至少有 4 个不同的原因，第一个原因是程序在多处理器的计算机上的执行速度。这些计算机提供了有效提升程序执行速度的方法，只要程序设计成可利用并发的硬件。现在有大量安装了多处理器的计算机，包括最近几年销售的许多个人计算机，不使用这种硬件能力是一种浪费。

第二个原因是即使计算机只有一个处理器，设计为使用并发执行功能的程序也比仅按顺序(非并发)执行的程序快。其要求是，程序不是计算绑定的(按顺序执行的版本没有充分利用处理器)。

第三个原因是并发提供了一种将问题的程序解决方案概念化的不同方法。如同递归是设计某些问题的解决方案的自然方法一样，许多问题领域自然要用并发方法。而且，许多程序是用于模拟物理实体和行为的，在许多情况下，所模拟的系统包含一个以上的实体，这些实体同时进行自己的活动。例如，在一块受控区域内飞行的飞行器，通信网络的中继站，以及工厂内的各种机器。为了精确模拟这样的系统，必须采用使用并发的软件。

使用并发的第四个原因是编写在多台计算机上本地分布或通过互联网分布的应用程序。许多机器，例如汽车，内置了一个以上的计算机，每个计算机都执行特定的任务。在许多情况下，这些计算机必须同步其程序的执行。互联网游戏是软件分布在多个处理器上的另一个例子。

并发现在用于许多日常的计算任务。Web 服务器并发处理文档请求。Web 浏览器现在使用辅助核心处理器来进行图形处理，解释嵌入在文档中的编程代码。在每个操作系统中，总是在执行许多并发进程，管理资源，从键盘上获取输入，从程序中显示输出，读写外部存储设备。简言之，并发计算是无处不在的。

13.2 子程序级并发概述

在考虑并发的语言支持前，必须理解并发的基本概念，以及使并发有用的要求。本节涵盖了这些主题。

13.2.1 基本概念

任务是程序的单元，与子程序类似，可以与同一程序的其他单元并发执行。程序中的每个任务可以支持一条控制线。任务有时也称为进程。在一些语言中，例如 Java 和 C#，某些方法用作任务。这种方法在线程对象中执行。

任务有三个特点与子程序相区别。第一，任务可以隐式启动，而子程序必须显式调用。第二，当程序单元调用一个任务时，不需要在继续自己的执行之前等待任务完成。第三，当任务执行结束时，控制可以返回启动任务执行的单元，也可以不返回。

任务有两类：重量级任务和轻量级任务。简单来说，重量级任务在自己的地址空间内

执行，轻量级任务运行在同一个地址空间中。实现轻量级任务比重量级任务更容易。此外，轻量级任务比重量级任务更高效，因为管理它们的执行只需要较少的处理。

任务可以通过共享的非局部变量、消息传递或者参数来与其他任务通讯。如果一个任务没有以任何方式与程序中其他任务通讯，或影响它们的执行，这个任务就是不相交的。由于任务通常共同作用来创建仿真效果或解决问题，因此，它们不是不相交的，它们必须通过某种形式的通讯来同步它们的执行，或共享彼此的数据。

同步是一种控制任务执行顺序的机制。当任务共享数据时，需要两类同步。当任务 A 在开始或继续执行之前，必须等待任务 B 完成特定的动作时，在任务 A 和任务 B 之间就需要合作同步。当两个任务需要使用某项不能同时使用的资源时，两者间就需要竞争同步。具体而言，如果任务 B 正在访问共享的数据位置 x 时，任务 A 也需要访问 x，那么任务 A 必须等待任务 B 完成对 x 的处理。因此，对于合作同步来说，任务可能需要等待完成正确操作所依赖的特定处理，而对于竞争同步来说，任务可能需要等待其他任务完成当前对特定共享数据的处理。

合作同步的简单形式可以用常见的生产者消费者问题来说明。这个问题起源于操作系统的开发，一个程序单元生成某个数据值或某项资源，另一个程序要使用它。所生成的数据通常由生产单元存放在存储缓冲区中，由消费单元从缓冲区中移除。在缓冲区中存放和移除的顺序必须同步。如果缓冲区为空，消费单元就不允许从缓冲区中取数据。同样，当缓冲区满时，生产单元不允许往缓冲区存放新的数据。这称为合作同步的问题，因为如果要正确使用缓冲区，共享数据结构的使用者就必须合作。

竞争同步防止两个任务在同一时刻访问共享的数据结构，这种情况会破坏共享数据的完整性。为了实现竞争同步，必须保证共享数据的互斥访问。

为了弄清楚竞争问题，考虑以下场景：假设任务 A 有语句 TOTAL=+1，其中 TOTAL 是共享的整型变量。再假设任务 B 有语句 TOTAL*=2。任务 A 和任务 B 试图同时改变 TOTAL。

在机器语言级别，每个任务都按下列三步过程来完成对 TOTAL 的运算：

(1) 获取 TOTAL 的值。

(2) 执行算术运算。

(3) 将新的值放回到 TOTAL。

如果没有竞争同步，根据操作的执行步骤，任务 A 和任务 B 在 TOTAL 上执行上述操作，会得到四种不同的结果。假定任务 A 和任务 B 在修改 TOTAL 之前，其值是 3。如果任务 A 在任务 B 开始之前完成运算，那么值将是 8，我们假设这是正确的。但如果任务 A 和 B 在两者把新值放回去之前都获取了 TOTAL 的值，结果就不正确。如果任务 A 先把值放回去，TOTAL 的值将是 6，这一情况如图 13-1 所示。如果任务 B 先把值放回去，TOTAL 的值将是 4。最后，如果任务 B 在任务 A 开始之前完成运算，那么值将是 7。导致这些问题的情况有时称为竞态条件，因为两个或更多任务竞争使用共享资源，程序的表现取决于哪一个任务先到达。竞争同步的重要性现在应该很清楚了。

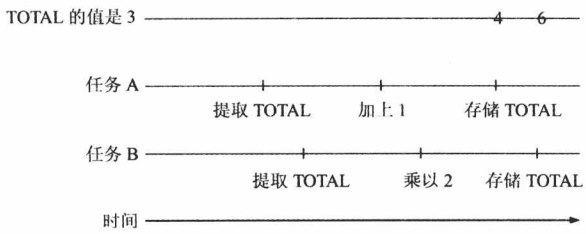


图 13-1　对竞争同步的需要

实现共享资源的互斥访问(支持竞争同步)的一种通常方法，是将资源看作某种可以由任务处理、但一次只允许处理一个任务的物品。为了获取共享资源的所有权，任务必须提出请求，只有其他任务没有所有权时，才将所有权授予该任务。当任务拥有一项资源时，其他任务就不能访问该资源。当任务不再拥有一项资源时，它必须放弃这项资源，使它能由其他任务访问。

实现共享资源互斥访问的三种方法是信号量(在 13.3 节介绍)、管程(在 13.4 节介绍)和消息传递(在 13.5 节介绍)。

同步机制必须能够延迟任务的执行。同步强制了任务(被强制延迟)的执行顺序。为了理解任务在生存期内发生了什么，必须考虑任务的执行是如何控制的。不管计算机有一个处理器或一个以上处理器，任务总是有可能比处理器更多。有一个运行时系统程序叫做调度程序，它在任务之间管理处理器的共享。如果没有中断，且所有任务具有相同的优先级，那么调度程序将简单地给每一任务一个时间片，例如 0.1 秒。当轮到一个任务时，调度程序就让它在一个处理器上执行这一段时间。当然，有一些事件会使它复杂化，例如，为了同步而延迟，以及等待输入或输出操作。因为输入输出操作比处理器的速度慢得多，所以任务在等待完成这类操作时，不会占用处理器。

任务有以下几种不同的状态。

- **新建**：任务已创建但还没有开始执行时，就处于新建状态。
- **就绪**：就绪的任务是已准备好运行，但目前还没有运行，或还没有由调度程序分配处理器时间，或者以前已经运行了但以本节前文中所介绍的某一种方式被阻塞了。准备运行的任务存储在一个所谓的任务就绪队列中。
- **运行**：运行的任务是目前正在执行，即拥有处理器且代码正在执行的任务。
- **阻塞**：被阻塞的任务已经在运行，但是因几个不同的事件之一而中断，最常见的是输入或输出操作。除了输入或输出以外，有些语言为用户程序提供了操作来指定一个任务被阻塞。
- **死亡**：死亡的任务在任何意义下都不再活跃。当任务执行完或被程序显式关闭后，它就死亡了。

任务状态的流程图如图 13-2 所示。

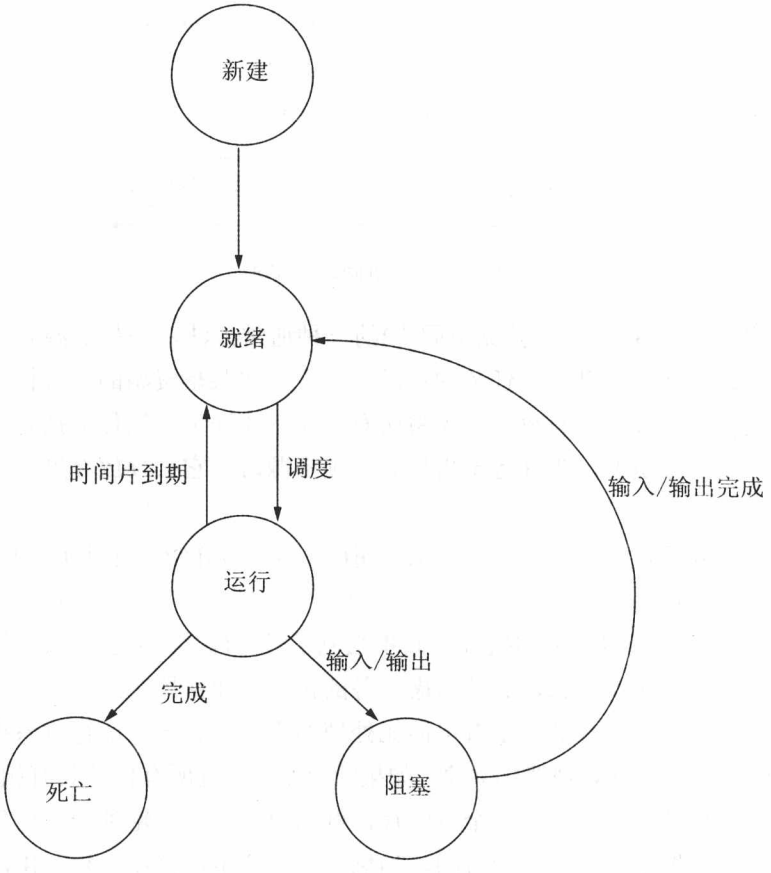


图 13-2 任务状态的流程图

任务执行的一个重要问题是：当目前运行的任务被阻塞，或其时间片已经结束时，如何选择一个就绪的任务，使之进入运行状态？人们采用一些不同的算法来进行选择，其中一些基于可列举的优先级等级。进行选择的算法在调度程序中实现。

活性(liveness)是与任务的并发执行和共享资源的使用相关的概念。在顺序执行程序的环境中，如果程序持续执行，并最终执行完成，程序就具有活性的特点。用更一般的话来说，活性意味着如果某个事件(例如程序完成)是应该发生的，它最终一定会发生，即执行是持续的。在具有共享资源的并发环境中，任务的活性可以不再存在，这意味着程序无法继续，永远不会完成。

例如，假设任务 A 和任务 B 都需要共享资源 X 和 Y 来完成它们的工作。再假设任务 A 获得了 X 的所有权，任务 B 获得了 Y 的所有权。执行一段时间后，任务 A 需要资源 Y 来继续，因此它请求 Y，但必须等到 B 释放它。同样，任务 B 请求 X，但必须等到 A 释放它。两个任务都不放弃所拥有的资源，结果两个任务都失去了活性，程序的执行永远不会正常完成。这种活性丢失称为死锁。死锁是对程序可靠性的严重威胁，因此在语言设计和程序设计中都需要认真考虑，以避免死锁。

现在讨论一些实现并发单元控制的语言机制。

13.2.2 为并发而设计的语言

在一些情况下，并发是通过库实现的。例如 OpenMP 是一个应用程序接口，它支持各种平台上 C、C++和 Fortran 中的共享内存多处理器编程。当然本书的重点是并发的语言支持。人们为了支持并发而设计了一些语言，从 20 世纪 60 年代中期的 PL/I 开始，包括现代的语言 Ada 95、Java、C#、F#、Python 和 Ruby[1]。

13.2.3 设计问题

语言并发支持的最重要的设计问题已经详细介绍过了，即竞争同步和合作同步。此外，还有几个次要的设计问题，其中最主要的是应用程序如何控制任务调度，任务在何时开始和结束执行，任务如何开始和结束执行，任务在何时创建，任务如何创建。

注意，本章不打算对并发做完整的介绍，只介绍与并发支持有关的最重要的语言设计问题。

下面几节介绍并发设计问题的三个可选答案：信号量、管程和消息传递。

13.3 信号量

信号量是用于实现任务同步的一种简单机制。虽然信号量是实现同步的早期方法，但它们仍在现代语言和基于库的并发支持系统中使用。下面介绍信号量以及如何将信号量用于任务同步。

13.3.1 概述

在通过对共享数据结构的互斥访问来实现竞争同步的工作中，Edsger Dijkstra 在 1965 年设计了信号量(Dijkstra，1968b)。信号量也可以用于实现合作同步。

为了提供数据结构的受限访问，可以在访问数据结构的代码周围放置守卫。守卫(guard)是一种语言设备，只有特定条件为真时，才允许执行被守卫的代码。守卫可用于实现每次只有一个任务访问共享的数据结构。信号量是守卫的一种实现。具体而言，信号量是一种数据结构，由一个整数和一个存储任务描述符的队列组成。任务描述符是一个数据结构，存储了任务执行状态的所有相关信息。

守卫机制中不可缺少的部分是一种方法，确保所有试图执行被守卫代码的要求最终都能实现。典型的方法是在不能授权访问时，将访问请求存储在任务描述符队列中，之后访问请求可以从队列中取出，并执行被守卫的代码。这就是信号量必须有计数器和任务描述符队列的原因。

---

1. 对于 Python 和 Ruby，程序是解释性的，所以它们只有逻辑并发。即使计算机有多个处理器，这些程序也不能使用多个处理器。

提供给信号量的仅有的两个操作，最初由 Dijkstra 命名为 P 和 V，来自于荷兰语单词 passeren(传送)和 vrygeren(释放)(Andrews 和 Schneider，1983)。本节后面的部分将这两种操作称为“等待”和“释放”。

13.3.2 合作同步

本章的许多地方都用生产者和消费者共享缓冲区的例子，来说明实现合作同步和竞争同步的不同方法。对于合作同步，这个缓冲区必须有某种方法记录缓冲区中空白位置的数目和已填充位置的数目(以防止缓冲区下溢或上溢)。信号量的计数器部分可用于这一目的。一个信号量变量，例如 emptyspots，可使用其计数器存储共享缓冲区中空白位置的数目；另一个变量，例如 fullspots，可使用其计数器存储缓冲区中已填充位置的数目。这两个信号量的队列可以存储被强制等待访问缓冲区的任务的描述符。emptyspots 的队列可以存储等待缓冲区中可用位置的任务；fullspots 的队列可以存储等待将数值放入缓冲区的任务。

示例缓冲区设计为一个抽象数据类型，其中所有的数据都通过子程序 DEPOSIT 进入缓冲区，通过子程序 FETCH 离开缓冲区。DEPOSIT 子程序只需要检查 emptyspots 信号量，确定是否有空白位置。如果至少有一个，就继续执行 DEPOSIT 子程序，该子程序必须包含对 emptyspots 的计数器减 1 的操作。如果缓冲区是满的，就必须让 DEPOSIT 的调用者等待在 emptyspots 队列中出现空白位置。当 DEPOSIT 完成时，DEPOSIT 子程序将 fullspots 信号量的计数器加 1，表明缓冲区中多了一个已填充位置。

FETCH 子程序具有与 DEPOSIT 相反的执行顺序。它检查 fullspots 信号量，确定缓冲区是否包含至少一项内容。如果是，就移除一项，并让 emptyspots 信号量的计数器加 1。如果缓冲区是空的，就把调用任务放入 fullspots 队列等待，直到出现一项内容。当 FETCH 完成时，它必须将 emptyspots 的计数器加 1。

作用在信号量类型上的操作通常不是直接的，而是通过 wait 和 release 两个子程序来实现。因此，上述 DEPOSIT 操作实际上要调用 wait 和 release 来完成。请注意，wait 和 release 必须能够访问任务就绪队列。

wait 子程序用于测试给定信号量变量的计数器。如果计数器的值大于 0，调用者就能执行操作，这时信号量变量的计数器减 1，表明目前少了一个计数项。如果计数器的值是 0，则必须将调用者放到信号量变量的等待队列中，并将处理器分配给其他就绪的任务。

任务使用 release 子程序来允许其他任务拥有特定信号量变量所计数的项。如果特定信号量变量的队列为空，这意味着没有任务在等待中，release 就将计数器加 1(表明现在多了一项可用的受控制内容)。如果有一个或更多的任务在等待，release 就将其中一个任务从信号量队列移往就绪队列。

以下是 wait 和 release 的简要伪代码描述：

```
wait(aSemaphore)
if aSemaphore 的计数 >0 then
aSemaphore 的计数器减 1
```

```
else
将调用者放在 aSemaphore 的队列
尝试将控制移交给某个就绪的任务
(如果任务就绪队列为空，发生死锁)
end if
release(aSemaphore)
if aSemaphore 的队列为空(没有任务正在等待中) then
aSemaphore 的计数器加 1
else
将调用任务放入任务就绪队列
将控制移交给 aSemaphore 队列中的一个任务
end
```

现在可以给出一个实现共享缓冲区合作同步的例程。在这个例子中，共享缓冲区存储整数值，是一个逻辑循环结构，用于多个生产者任务和消费者任务。

以下伪代码展示了生产者任务和消费者任务的定义，使用了两个信号量来避免缓冲区下溢或上溢，从而实现合作同步。假设缓冲区的长度为 BUFLEN，已经存在实际操作缓冲区的子程序 FETCH 和 DEPOSIT。用点号表示对信号量计数器的访问，例如，如果 fullspots 是一个信号量，它的计数器就用 fullspots.count 来表示。

```
semaphore fullspots, emptyspots;
fullspots.count = 0;
emptyspots.count = BUFLEN;
task producer;
  loop
  -- produce VALUE --
  wait(emptyspots);      { wait for a space }
  DEPOSIT(VALUE);
  release(fullspots);    { increase filled spaces }
  end loop;
end producer;

task consumer;
  loop
  wait(fullspots);        { make sure it is not empty }
  FETCH(VALUE);
  release(emptyspots);    { increase empty spaces }
  -- consume VALUE --
  end loop
end consumer;
```

如果缓冲区当前为空，信号量 fullspots 使 consumer 任务进入队列，等待一项缓冲区内

容。如果缓冲区当前已满，信号量 emptyspots 使 producer 任务进入队列，等待缓冲区的一个空白位置。

13.3.3 竞争同步

缓冲区示例没有实现竞争同步。对该数据结构的访问可以用额外的信号量来控制。这个信号量不必对什么内容计数，就可以简单地通过计数器表明，缓冲区当前是否正在使用。只有信号量的计数器值为 1 时(表明目前没有访问共享缓冲区)，wait 语句才允许访问缓冲区。如果信号量的计数器值为 0，则目前正在访问缓冲区，要将任务放入信号量的队列。请注意，信号量的计数器必须初始化为 1，信号量的队列在开始使用之前必须总是初始化为空。

若信号量只需要二个值的计数器，就像下面例子中实现竞争同步所用的信号量，称为二元信号量。

下面示例的伪代码，展示了如何使用信号量来实现共享缓冲区并发访问的竞争同步和合作同步。access 信号量用于确保缓冲区的互斥访问。注意，这里可能有一个以上的生产者和一个以上的消费者。

```
semaphore access, fullspots, emptyspots;
access.count = 1;
fullspots.count = 0;
emptyspots.count = BUFLEN;

task producer;
  loop
  -- produce VALUE --
  wait(emptyspots);       { wait for a space }
  wait(access);           { wait for access }
  DEPOSIT(VALUE);
  release(access);        { relinquish access }
  release(fullspots);     { increase filled spaces }
  end loop;
end producer;
task consumer;
  loop
  wait(fullspots);        { make sure it is not empty }
  wait(access);           { wait for access }
  FETCH(VALUE);
  release(access);        { relinquish access }
  release(emptyspots);    { increase empty spaces }
  -- consume VALUE --
  end loop
end consumer;
```

粗略地看看这个例子，可能会让人觉得有问题，具体而言，假设一个任务正在 consumer 中调用 wait(access)处等待时，另一个任务从共享缓冲区中取了最后一个值。好在这种情况

并不会发生，因为 wait(fullspots)会将 fullspots 计数器减 1，在缓冲区中为调用它的任务保留一个值。

到目前为止，还有一个信号量的关键点没有讨论。回顾先前介绍的竞争同步问题：共享数据的操作不能重叠。如果前一个操作仍在进行，就开始第二个操作，共享数据就会出错。信号量本身也是一个共享数据对象，因此信号量上的操作也会有同样的问题，所以信号量操作决不可中断。许多计算机有专门为信号量操作设计的、不可中断的指令。如果没有这样的指令，那么用信号量来实现竞争同步就会成为没有简单解决方案的难题。

13.3.4 评价

使用信号量来实现合作同步，会产生不安全的编程环境，无法对使用信号量的正确性进行静态检查，这种检查取决于信号量所在程序的语义。在缓冲区的例子中，在 producer 任务中漏掉 wait(emptyspots)语句，会导致缓冲区上溢；在 consumer 任务中漏掉 wait(fullspots)语句，会导致缓冲区下溢；漏掉任何一个 release 会导致死锁。这些都是合作同步失败的情况。

信号量在实现合作同步时导致的可靠性问题，在信号量用于竞争同步时也会产生。在上述两个任务中漏掉 wait(access)语句，会导致对缓冲区的不安全访问。在上述两个任务中漏掉 release(access)语句，会导致死锁。这些都是竞争同步失败的情况。意识到使用信号量的风险后，Per Brinch Hansen(1973)写道："对于从不犯错的理想程序员来说，信号量是一种优雅的同步工具。"遗憾的是，理想程序员非常少见。

13.4 管程

在并发环境中，某些信号量问题的一个解决方案，是将共享数据结构和它们的操作封装在一起，并隐藏它们的数据表示，即将共享数据结构变成抽象数据类型，并带有一些特定的限制。这种解决方案将同步的职责移交给运行时系统，不用信号量就可实现竞争同步。

13.4.1 概述

在数据抽象概念的形成过程中，从事数据抽象工作的人们将同样的概念应用到并发编程环境的共享数据上，产生了管程。根据 Per Brinch Hansen 的文献(Brinch Hansen，1977，p.xvi)，Edsger Dijkstra 在 1971 年建议，将作用于共享数据的所有同步操作，集成到一个程序单元中。Brinch Hansen (1973)将这一概念应用到操作系统环境中。第二年，Hoare(1974)将这种结构称为管程。

第一种加入了管程的编程语言是并发 Pascal(Brinch Hansen，1975)。Modula(Wirth，1977)、CSP/k(Holt 等，1978)和 Mesa(Mitchell 等，1979)也实现了管程。在现代语言中，Ada、Java 和 C#都支持管程，在本章也都有介绍。

13.4.2 竞争同步

管程最重要的特性之一是共享数据位于管程之内，而不是任何客户单元中。程序员不必通过使用信号量或其他机制来同步对共享数据的互斥访问。由于访问机制是管程的一部分，因此实现管程就可以使得每次只有一个访问，从而保证同步的访问。如果在调用管理过程时管程正忙，对它的调用就会隐式地阻塞，并存储到队列中。

13.4.3 合作同步

虽然对共享数据的互斥访问是管程所固有的，但进程间的合作仍然是程序员的职责。特别是，程序员必须保证共享缓冲区不会出现下溢或上溢。不同的语言提供了不同的编程实现合作同步的方法，所有方法都与信号量有关。

图 13-3 显示了一个程序，它包含四个任务和一个实现了对并发共享缓冲区的同步访问的管程。图中，管程的接口显示为两个标有“插入”和“删除”(用于插入和删除数据)的方框。管程看起来像是抽象数据类型——一个访问权有限的数据结构——这就是管程。

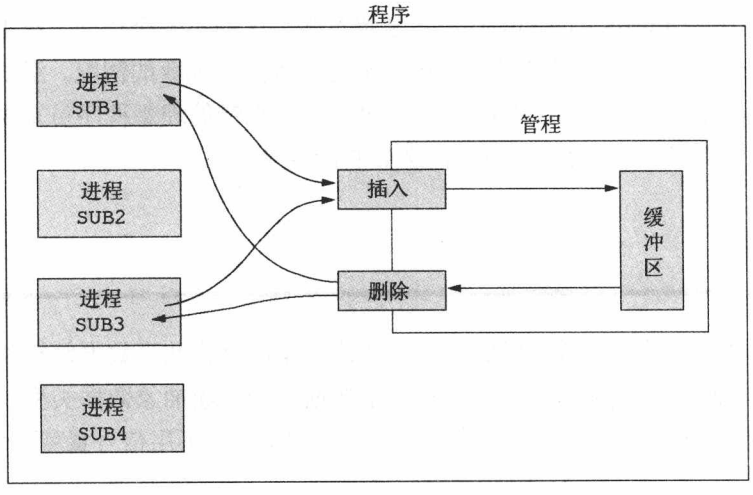


图 13-3 这个程序使用管程来控制对共享缓冲区的访问

13.4.4 评价

管程与信号量相比，是一种实现竞争同步的更好方法，主要是因为 13.3 节讨论的信号量的问题。对于管程来说，合作同步仍然是一个问题，在接下来的章节中介绍 Ada 和 Java 对管程的实现时会清楚这一点。

在表示并发控制上，信号量与管程的功能是一样的。信号量可用于实现管程，管程也可用于实现信号量。

Ada 提供了两种实现管程的方法。Ada 83 包含一个通用的任务模型，可用于支持管程。Ada 95 添加了一个更简洁更高效的管程构造方法，称为受保护对象。这两种方法都使用消息传递作为支持并发的基础模型。消息传递模型允许并发单元是分布的，而管程不允许。

消息传递在 13.5 节中介绍，Ada 对消息传递的支持在 13.6 节中介绍。

在 Java 中，管程可以在设计为抽象数据类型的类中实现，共享数据就是这种类型。通过在访问方法前添加 synchronized 修饰符，来控制对类对象的访问。13.7.4 节中给出了一个用 Java 写的共享缓冲区管程的例子。

C#有一个预定义类 Monitor，用于实现管程。

13.5 消息传递

本节介绍并发中消息传递的基本概念。注意这个消息传递的概念与面向对象编程中用于执行方法的消息传递无关。

13.5.1 概述

Brinch Hansen(1978)和 Hoare(1978)最早从事在并发任务之间提供消息传递能力的语言设计工作。这两位开发消息传递的先驱还提出了一项技术来处理以下问题：有多个任务同时请求与给定任务进行通信时，该怎么办。结论是需要某种形式的不确定性，从而在这些请求中公平地选择哪一个先处理。这种公平性可以用不同的方式来定义，但总的来说，这意味着所有请求都有与给定任务通讯的相同机会(假设每一个发出请求的任务都具有相同的优先级)。Dijkstra(1975)将语句级控制的非确定性语言结构称为保护命令(guarded command)，第 8 章介绍了保护命令。保护命令是用于控制消息传递的结构基础。

13.5.2 同步消息传递的原理

消息传递可以是同步的，也可以是异步的。这里只介绍同步消息传递。同步消息传递的基本概念是：任务经常处于忙的状态中，在忙时不能被其他单元打断。假设任务 A 和任务 B 都处在执行过程中，A 希望给 B 发送一条消息。显然，如果 B 正忙，就不希望被其他任务打断，因为那会打乱 B 当前的处理。而且，消息通常会导致接收者进行相关处理，如果其他处理还没有完成，这可能就不合适。一种替代的方案是提供一种语言机制，允许任务告诉其他任务，它何时准备好接收消息。这一方案有些像是主管，他会让秘书在另一项活动(可能是一个重要的会谈)结束前，让所有打入的电话都等着。然后，当目前的会谈完成后，主管才告诉秘书现在愿意接听等候者的电话。

可以将任务设计成这样：任务在某些时候因为空闲，或者因为在继续执行前需要从其他程序单元中获取信息，可以暂停执行。这好像是一个正在等重要电话的人，在有些情况下除了坐等之外无事可做。不过，如果任务 A 在任务 B 发送消息时，正在等待这条消息，就可以传递消息。这种实际的消息传递称为会合(rendezvous)。请注意，会合只会发生在发送者和接收者都希望发生时。在会合时，消息中的信息可以单向或双向传递。

如下一节所述，任务的合作同步和竞争同步很容易用消息传递模型来处理。

13.6 Ada 对并发的支持

本节介绍 Ada 对并发提供的支持。Ada 83 只支持同步消息传递。

13.6.1 基本原理

Ada 对任务的设计，部分基于 Brinch Hansen 和 Hoare 的工作，以消息传递为设计基础，并使用非确定机制从发送消息的多个任务中选择。

完整的 Ada 任务模型非常复杂，以下的介绍很有限，只关注 Ada 中的同步消息传递机制。

Ada 任务比管程更活跃。管程是提供对所存储的共享数据进行管理的被动实体，只在需要时才提供服务。Ada 任务在用于管理共享数据时，可以看成是与所管理资源在一起的管理者。任务具有好几种管理机制，有些是确定性的，有些是非确定性的，允许在访问资源的竞争请求中选择。

Ada 任务的语法形式类似于 Ada 包，也有两个部分——说明部分和体部分，它们有相同的名称。任务的接口就是它的入口点，即可以从其他任务中接收消息的地方。由于这些入口点是任务接口的一部分，所以自然位于任务的说明部分。由于会合会引起信息交换，所以消息可以带有参数，因此任务的入口点也必须允许有参数，也必须在说明部分指定。从外观上看，任务的说明与抽象数据类型的包规范很相似。

作为 Ada 任务说明的一个例子，考虑以下代码，它包含一个入口点 Entry_1，有一个输入参数：

```
task Task_Example is
  entry Entry_1(Item : in Integer);
end Task_Example;
```

任务体必须包含与任务说明部分中 entry 子句对应的入口点的语法形式。在 Ada 中，任务体的入口点由 accept 子句定义，由 accept 保留字所引出。accept 子句定义为以 accept 保留字开始，以配对的 end 保留字结束的语句部分。accept 子句本身相对简单，但嵌入其中的其他语言结构会使语义非常复杂。简单的 accept 子句的形式是

```
accept 入口名(形参)do
...
end 入口名;
```

accept 的入口名与相关联的任务说明部分 entry 子句中的名字相匹配。可选的参数提供了在调用者和被调用任务间进行数据通讯的方法。在 do 和 end 之间的语句定义了会合期间所进行的操作，这些语句合在一起称为 accept 子句体。在实际的会合期间，会挂起发送者。

只要 accept 子句收到一个不管什么原因不愿接受的消息时，就必须挂起发送者，直到接收者的 accept 子句准备好接受这条消息为止。当然，accept 子句还必须记得发送者已经发送了尚未被接受的消息，为此，任务中的每个 accept 子句都有一个相关联的队列，存储试图与它通讯但未成功的其他任务的列表。

以下是任务体的框架，其说明已经在前面给出：

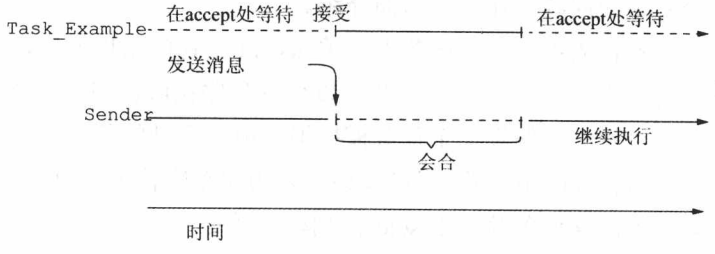
```
task body Task_Example is
  begin
  loop
    accept Entry_1(Item : in Integer) do
      ...
    end Entry_1;
end loop;
end Task_Example;
```

这一任务体的 accept 子句是任务说明中入口 Entry_1 的实现。如果 Task_Example 开始执行，并在任何其他任务向 Entry_1 发送消息之前到达 Entry_1 的 accept 子句，就挂起 Task_Example。如果 Task_Example 在 accept 处挂起时，另一个任务向 Entry_1 发送了一条消息，就会发生会合，并执行 accept 子句体。然后，由于循环，执行流返回到 accept。如果没有其他任务给 Entry_1 发送消息，执行就再一次挂起，等待下一条消息。

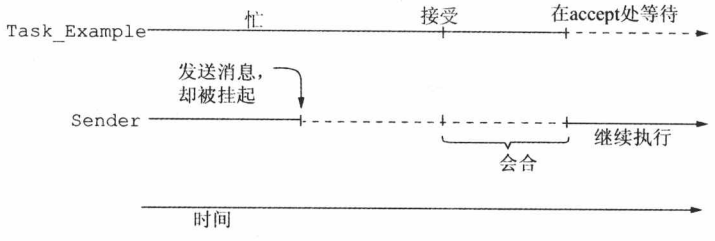
在这个简单例子中，有两种情况会发生会合。第一种情况是接收者任务 Task_Example 等待另一个任务向 Entry_1 入口发送消息，当消息发送后，会合就发生了。这种情况在前面介绍过了。第二种情况是当另一个任务试图发送消息给相同的入口时，接收者在会合中正忙，或者因其他与会合无关的处理正忙。在这种情况下，会挂起发送者，直到接收者有空，能够在会合中接受消息为止。如果在接收者正忙时有多条消息到达，发送者将进入队列，等待依次会合。

上面介绍的两种会合可以用图 13-4 中的时间线图来表示。

任务可以没有入口点，这种任务称为施动者任务(actor task)，因为它们不会等待会合来完成任务。施动者任务可以通过给其他任务发送消息而与它们会合。与施动者任务相反，任务可以有 accept 子句，但在 accept 子句外没有代码，因此只能响应其他任务，这种任务称为服务器任务(server task)。



(a) Task_Example等待Sender



(b) Sender等待Task_Example

图 13-4　两种与 Task_Example 发生会合的情况

Ada 任务向另一个任务发送消息时，必须知道那个任务的入口名。然而，反过来不是这样：任务入口不必知道它要接受的消息来自于哪个任务。这种不对称与 CSP (Communicating Sequential Processes)(Hoare，1978)语言的设计相反。CSP 也使用并发的消息传递模型，但任务只从显式命名的任务处接受消息。这种设计的缺点是无法建立通用的任务库。

图 13-5 显示了描述任务 A 向任务 B 发送消息并会合的常用图形化方法。

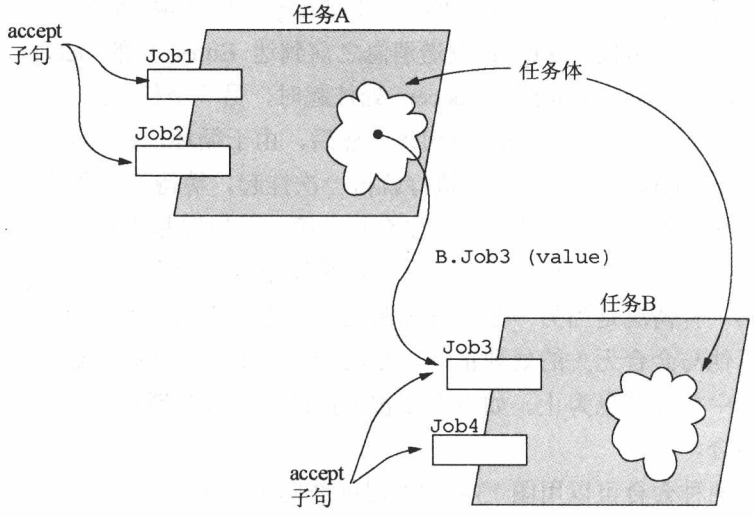


图 13-5 任务 A 向任务 B 发送消息而发生会合的图形化表示

任务在包、子程序或块的声明部分声明。静态创建的任务[2]与带有声明部分的代码语句同时开始执行。例如，在主程序中声明的任务与主程序代码体中的第一条语句同时执行。任务的结束是一个复杂的问题，在本节后面介绍。

任务可以有任意数目的入口。在任务中相关 accept 子句的出现顺序，规定了接受消息的顺序。如果任务有一个以上的入口点，要求以任意顺序接收消息，任务就要使用 select 语句来包含这些入口。例如，假设一个任务模拟银行柜台出纳的行为，这个出纳必须为银行内上步式柜台(walk-up station)的顾客服务，还必须为汽车穿梭窗口(drive-up window)的顾客服务。以下框架性的出纳任务演示了 select 结构：

```
task body Teller is
begin
  loop
    select
      accept Drive_Up(形参) do
        ...
      end Drive_Up;
      ...
    or
```

2. 任务也可以动态创建，但这儿不涉及这种任务。

```
      accept Walk_Up(形参) do
        ...
      end Walk_Up;
      ...
    end select;
  end loop;
end Teller;
```

在这个任务中有两个 accept 子句 Walk_Up 和 Drive_Up，每个子句都有相关联的队列。在任务执行时，select 的动作就是检查与两个 accept 子句相关联的队列。如果一个队列为空，但另一个队列包含至少一条等待着的消息(顾客)，那么与等待着的消息相关联的 accept 子句就与发送已接收的第一条消息的任务会合。如果两个 accept 子句都是空队列，select 就等待调用其中一个入口。如果两个 accept 子句都有非空队列，就非确定性地选择一个 accept 子句与其调用者会合。循环使 select 语句永远重复执行下去。

accept 子句的 end 标志了赋值或引用 accept 子句中形参的代码尾。在 accept 子句和下一个 or(如果该 accept 子句是 select 中的最后一个，则是 end select)之间的代码(如果有的话)称为扩展的 accept 子句。扩展的 accept 子句只在相关联的(它紧跟的)accept 子句执行后才执行。扩展 accept 子句的执行不是会合的一部分，可以与调用任务并发执行。调用者在会合期间挂起，但在 accept 子句结束后，放回到就绪队列中。如果 accept 子句没有形参，就不需要 do-end，accept 子句可以只包含扩展的 accept 子句。这样的 accept 子句专门用于同步。13.6.3 节的 Buf_Task 任务演示了扩展的 accept 子句。

13.6.2 合作同步

每个 accept 子句可以带有一个守卫，以 when 子句的形式延迟会合。例如：

```
when not Full(Buffer) =>
  accept Deposit(New_Value) do
    ...
  end
```

带有 when 子句的 accept 子句可以是开放的或是关闭的。如果 when 子句的布尔表达式当前为真，accept 子句就是开放的(open)，如果布尔表达式为假，accept 子句就是关闭的(closed)。没有守卫的 accept 子句总是开放的。开放的 accept 子句可以会合，关闭的 accept 子句无法会合。

假设在 select 子句中有多个带守卫的 accept 子句，这样的 select 子句通常放在一个无限循环内。这个循环使 select 子句重复执行，每次循环都对每个 when 子句求值。每次循环都会生成一系列开放的 accept 子句，如果有一个开放的子句具有非空队列，就从该队列中取出一条消息，并进行会合。如果有一个以上的 accept 子句具有非空队列，就非确定性地选择一个队列，从该队列中取出一条消息，并进行会合。如果所有开放子句的队列都为空，任务就等待有消息发送到这些 accept 子句中的某一个子句，那时就进行会合。如果执行 select 时所有 accept 子句都是关闭的，就抛出一个运行时异常或错误。保证有一个 when 子

句永远为真，或者在 select 中添加一个 else 子句，就可以避免这种可能。else 子句可以包含除 accept 子句以外的任何语句序列。

select 子句可以有一条特殊的语句 terminate，只有在该子句是开放的、且没有其他子句也开放的情况下，才会选中它。选中 terminate 子句意味着任务完成了它的作业，但还没有终止。本节后面会介绍任务的终止。

13.6.3 竞争同步

前面介绍的特性支持任务间合作同步和通信。接下来介绍 Ada 如何强制共享数据结构的互斥访问。

如果对数据结构的访问由任务来控制，就可以在任务内部声明数据结构，实现互斥访问。任务执行的语义通常保证了对结构的互斥访问，因为在指定时间内，任务中只能有一个 accept 子句是活跃的。唯一的例外发生在任务嵌套于过程或其他任务的情况下。例如，如果定义了共享数据结构的任务有一个嵌套的任务，嵌套的任务也可以访问共享结构，这就会破坏数据的完整性。因此，用于控制共享数据结构访问的任务不应该再定义任务。

以下是 Ada 任务的一个例子，为一个缓冲区实现了管程。该缓冲区的行为很像 13.3 节中的缓冲区，那个缓冲区是用信号量来控制同步的。

```
task Buf_Task is
  entry Deposit(Item : in Integer);
  entry Fetch(Item : out Integer);
end Buf_Task;

task body Buf_Task is
  Bufsize : constant Integer := 100;
  Buf : array (1..Bufsize) of Integer;
  Filled : Integer range 0..Bufsize := 0;
  Next_In,
  Next_Out : Integer range 1..Bufsize := 1;
begin
  loop
    select
      when Filled < Bufsize =>
        accept Deposit(Item : in Integer) do
          Buf(Next_In) := Item;
        end Deposit;
        Next_In := (Next_In mod Bufsize) + 1;
        Filled := Filled + 1;
    or
      when Filled > 0 =>
        accept Fetch(Item : out Integer) do
          Item := Buf(Next_Out);
        end Fetch;
        Next_Out := (Next_Out mod Bufsize) + 1;
        Filled := Filled - 1;
    end select;
```

```
  end loop;
end Buf_Task;
```

在这个例子中，两个 accept 子句都是扩展的。这两个扩展的子句可以与调用相应 accept 子句的任务并发执行。

使用 Buf_Task 的生产者任务和消费者任务具有以下形式：

```
task Producer;
task Consumer;
task body Producer is
  New_Value : Integer;
begin
 loop
   -- produce New_Value --
   Buf_Task.Deposit(New_Value);
 end loop;
end Producer;

task body Consumer is
  Stored_Value : Integer;
begin
  loop
    Buf_Task.Fetch(Stored_Value);
    -- consume Stored_Value --
  end loop;
end Consumer;
```

13.6.4 任务终止

执行流到达任务代码体的末尾时，任务的执行就完成了。因为没有处理程序而抛出异常时，这就可能发生。Ada 的异常处理在第 14 章介绍。如果任务没有创建任何其他任务(称为依赖任务)，那么它执行完后，就会终止。创建了依赖任务的任务，在它的代码执行完成、且所有依赖任务都终止后才终止。任务在开放的 terminate 子句处等待时，就可以结束执行，在这种情况下，只有当它的主代码(创建它的块、子程序或任务)和所有依赖主代码的任务都完成，或者在开放的 terminate 子句处等待时，任务才能终止。在这种情况下，所有这些任务都同时终止，块或子程序直到所有依赖任务都终止后才退出。

13.6.5 优先级

任务可以在其说明中指定优先级，这是用 pragma 指令[3]实现的，如下

```
pragma Priority(静态表达式);
```

静态表达式通常是一个整数字面值或预定义的常量，表达式的值给定了任务或任务所属类型定义的相对优先级。优先级的取值范围是与实现相关的，可能的最高优先级可以用

3. pragma 指令是编译器的一个指令。

priority 类型的属性 Last 来指定，Priority 类型在 System 中定义(System 是一个预定义的包)。例如，下面一行指定了任意实现中的最高优先级：

```
pragma Priority(System.Priority'Last);
```

任务指定了优先级后，在当前执行的任务被阻塞，或者所分配的时间到期，或者完成执行时，任务调度程序就用这些优先级来确定从任务就绪队列中选择哪个任务。此外，如果进入任务就绪队列的任务比当前执行的任务具有更高的优先级，那么正在执行的低优先级任务被抢占，高优先级任务开始执行(如果之前已经在执行，就恢复执行)。被抢占的任务失去处理器，并放入任务就绪队列。

13.6.6 受保护对象

如前所述，要控制对共享数据的访问，可以将数据包含在任务中，只允许通过任务入口来访问，这隐式地实现了竞争同步。这种方法的一个问题是难以高效地实现会合机制。Ada 95 的受保护对象提供了一种实现竞争同步的替代方法，不需要涉及会合机制。

受保护对象不是一个任务，它更像是 13.4 节介绍的管程。受保护对象可以通过受保护的子程序来访问，或者通过与任务中 accept 子句语法相似的入口来访问[4]。受保护的子程序可以是受保护的过程(提供了对受保护对象数据的互斥读写访问)，或者是受保护函数(提供了数据的并发只读访问)。入口不同于受保护的子程序，因为它们可以有守卫。

在受保护过程的过程体中，所包含的受保护单元的当前实例定义成一个变量；在受保护函数的函数体中，所包含的受保护单元的当前实例定义成一个常量，允许并发的只读访问。

对受保护对象的入口调用，为一个或多个使用同一个受保护对象的任务，提供了同步通信。这些入口调用所提供的访问，类似于对任务所含数据的访问。

在前面的章节中用任务解决的缓冲区问题，用受保护对象来解决更简单。请注意，这个例子不包含受保护的子程序。

```
protected Buffer is
  entry Deposit(Item : in Integer);
  entry Fetch(Item : out Integer);
private
  Bufsize : constant Integer := 100;
  Buf   : array (1..Bufsize) of Integer;
  Filled : Integer range 0..Bufsize := 0;
  Next_In,
  Next_Out : Integer range 1..Bufsize := 1;
  end Buffer;

protected body Buffer is
  entry Deposit(Item : in Integer)
     when Filled < Bufsize is
```

4. 受保护对象体的入口使用保留字 entry，而不是任务体使用的 accept。

```
    begin
    Buf(Next_In) := Item;
    Next_In := (Next_In mod Bufsize) + 1;
    Filled := Filled + 1;
    end Deposit;
  entry Fetch(Item : out Integer) when Filled > 0 is
    begin
    Item := Buf(Next_Out);
    Next_Out := (Next_Out mod Bufsize) + 1;
    Filled := Filled - 1;
    end Fetch;
end Buffer;
```

13.6.7 评价

使用并发的通用消息传递模式来构建管程，与使用 Ada 包支持抽象数据类型一样，两者都是比所需要功能更通用的工具。受保护对象是实现同步访问共享数据的更好方法。

在没有带独立内存的分布式处理器时，以消息传递作为在并发环境下实现同步访问共享数据的方法，选择管程还是任务是一种个人偏好。但是，对于 Ada 语言来说，受保护对象显然比任务更好地支持对共享数据的并发访问，不仅代码更简单，效率也高得多。

对于分布式系统，消息传递是一个较好的并发模式，因为它天然地支持在独立的处理器上并行地执行独立进程的概念。

13.7 Java 线程

Java 中的并发单元是名为 run 的方法，其代码可以与(其他对象的)其他方法以及 main 方法并发执行。执行 run 方法的进程称为线程(thread)。Java 的线程是轻量级任务，意思是它们运行在相同的地址空间内。这与 Ada 任务是不同的，Ada 任务是重量级线程(运行在自己的地址空间中)[5]。这种区别的一个重要结果就是线程比 Ada 的任务需要的代价小得多。

有两种方法来定义带有 run 方法的类。一种方法是定义一个预定义类 Thread 的子类，并重写它的 run 方法。但是如果新的子类必须有一个自然父类，那么显然不能把它定义为 Thread 的子类。在这种情况下，就定义一个从自然父类继承的子类，并实现 Runnable 接口。Runnable 提供了 run 方法协议，因此任何实现 Runnable 的类必须定义 run。实现了 Runnable 的类的对象被传递给 Thread 的构造函数。这种方法仍然需要一个 Thread 对象，在 13.7.5 节的例子中就会看到。

在 Ada 中，任务可以是施动者，也可以是服务器，任务通过 accept 子句相互通信。Java 的 run 方法总是施动者，它们没有相互通信的机制，但 join 方法(参见 13.7.1 节)和通过共

5. 实际上，尽管 Ada 任务像重量级任务那样运行，但在一些情况下，它们目前实现为线程。这有时用库来完成，例如 IBM Rational Apex Native POSIX Threading 库。

享数据通信的方法例外。

Java 线程是一个复杂的问题，本节只介绍它最简单但最有用的部分。

13.7.1 Thread 类

Thread 类不是其他任何类的自然父类，它为子类提供了一些服务，但与子类的计算目的毫无关系。Thread 是程序员创建并发 Java 程序唯一可用的类。如前所述，13.7.5 节将简要介绍 Runnable 接口的使用。

Thread 类包含五个构造函数和一些方法以及常量。run 方法描述了线程的行为，总是由 Thread 的子类覆盖。Thread 的 start 方法调用 run 方法[6]，将线程作为并发单元启动起来。对 start 的调用是很特殊的，因为过程控制会立即返回到调用者，调用者随后继续执行，与新启动的 run 方法同时执行。

下面是一个框架性的 Thread 子类，代码片断创建了该子类的一个对象，并在新线程中开始 run 方法的执行：

```
class MyThread extends Thread {
public void run() { ... }
}
...
Thread myTh = new MyThread();
myTh.start();
```

当 Java 应用程序开始执行时，会创建一个新线程(main 方法在该线程中运行)，并调用 main。因此，所有 Java 应用程序都运行在线程中。

当程序有多个线程时，调度程序必须决定在任意指定的时刻，运行哪个或哪些线程。在许多情况下只有一个处理器，因此实际上一次只有一个线程在运行。很难对 Java 调度程序如何工作给出精确的描述，因为不同的实现(Solaris、Windows 以及其他操作系统)调度线程的方式并不完全一样。不过一般情况下，假设所有线程都有相同的优先级，调度程序就以循环的方式给每个就绪的线程分配相等的时间片。13.7.2 节介绍如何给不同线程指定不同的优先级。

Thread 类为控制线程的执行提供了几个方法。yield 方法没有参数，是正在运行的线程自愿放弃处理器的请求[7]。线程会立即放入任务就绪队列，准备好运行。然后调度程序从任务就绪队列中选择优先级最高的线程。如果没有比刚才放弃处理器的线程优先级更高的其他就绪线程，刚才这个线程就可能仍然是下一个获得处理器的线程。

sleep 方法有一个参数，是 sleep 调用者希望线程被阻塞的毫秒数。指定数目的毫秒过去后，线程就放入任务就绪队列。由于无法知道线程在运行前会在任务就绪队列中待多久，因此 sleep 的参数只是线程暂停运行的最少时间。sleep 方法可以抛出 InterruptedException，

---

6. 直接调用 run 方法并不总会奏效，因为有时需要初始化，这包含在 start 方法中。

7. yield 方法实际上定义为向调度程序提出的建议，调度程序可以照做也可以不照做(虽然通常会照做)。

调用 sleep 的方法必须处理这个异常。第 14 章将详细介绍异常。

join 方法用于强制一个方法延迟执行，直到另一个线程的 run 方法完成执行。当一个方法的处理要等到其他线程的任务完成才能继续时，就使用 join。例如，我们可能有下面的 run 方法：

```
public void run() {
...
Thread myTh = new Thread();
myTh.start();
// do part of the computation of this thread
myTh.join(); // Wait for myTh to complete
// do the rest of the computation of this thread
}
```

join 方法将调用它的线程置为阻塞状态，只有调用 join 的线程完成后，才会结束阻塞状态。如果该线程正好被阻塞，就可能发生死锁。为了避免这种情况，可以用一个参数来调用 join，这个参数是以毫秒计的，是调用线程等待被调用线程完成的时间限制。例如：

```
myTh.join(2000);
```

将导致调用线程用 2 秒来等待 myTh 完成。如果 2 秒过去后，myTh 没有执行完，就把调用线程放回就绪队列，这意味着它一被调度，就会继续执行。

Java 的早期版本包含三个额外的 Thread 方法：stop、suspend 和 resume。所有这三个方法都因为安全问题而弃用了。stop 方法有时会被一个简单的方法覆盖，该方法通过将引用变量设置为 null 来破坏线程。

run 方法结束执行的正常方式是到达代码结尾处。然而在许多情况下，线程会一直运行直到被喊停。与此相关的问题是，线程如何确定该继续还是结束。interrupt 方法是一种告诉线程应该停止的方法。这种方法并不停止线程，而是向线程发送一个消息，实际上只是在线程对象中设置一个比特位，可以被线程检测到。检测该比特位的是谓词方法 isInterrupted。这不是一个完整的解决方案，因为当调用 interrupt 方法时，试图中断的线程可能正在休眠或者等待，即不会检测它是否已被中断。对于这种情况，interrupt 方法也会抛出异常 InterruptedException，从休眠或等待状态中唤醒线程。因此，可以周期性地检测线程是否被中断，如果是，则检测线程是否可以终止。线程不会错过中断，因为如果中断线程时它正在休眠或等待，则会被中断唤醒。实际上关于 interrupt 的行为和使用还有许多细节，但这里并不涉及(Arnold 等，2006)。

13.7.2 优先级

线程的优先级不必都相同。线程最初的默认优先级与创建它的线程相同。如果 main 创建了一个线程，它的默认优先级就是常量 NORM_ PRIORITY，通常是 5。Thread 定义了

另两个优先级常量 MAX_PRIORITY 和 MIN_PRIORITY，值通常分别是 10 和 1[8]。线程的优先级可以用 setPriority 方法来修改，新的优先级可以是任意一个预定义的常量，或者是在 MIN_PRIORITY 和 MAX_PRIORITY 之间的任意数值。getPriority 方法返回线程当前的优先级。优先级常量定义在 Thread 中。

当有不同优先级的线程时，调度程序的行为就受到优先级的控制。当正在执行的线程被阻塞或删除，或者它的时间片终止时，调度程序就从任务就绪队列中选择优先级最高的线程。当机会来临时，只有在任务就绪队列中没有更高优先级的线程时，较低优先级的线程才会运行。

13.7.3 信号量

java.util.concurrent.Semaphore 包定义了 Semaphore 类。这个类的对象实现了计数信号量。计数信号量有一个计数器，但没有存储线程描述符的队列。Semaphore 类定义了两个方法 acquire 和 release，它们响应 13.3 节介绍的 wait 和 release 操作。

Semaphore 的基本构造函数带一个整型参数，它初始化了信号量的计数器。例如，下面的代码可用于初始化 13.3.2 节中缓冲区示例的 fullspots 和 emptyspots 信号量：

```
fullspots = new Semaphore(0);
emptyspots = new Semaphore(BUFLEN);
```

生产者方法的 deposit 操作如下：

```
emptyspots.acquire();
deposit(value);
fullspots.release();
```

同样，消费者方法的 fetch 操作如下：

```
fullspots.acquire();
fetch(value);
emptyspots.release();
```

deposit 和 fetch 方法可以使用 13.7.4 节中的方式，实现访问缓冲区所需的竞争同步。

13.7.4 竞争同步

Java 方法(但构造函数除外)可以指定为 synchronized(同步的)。通过特定对象调用的同步方法，必须在运行于该对象之上的其他同步方法运行之前完成执行。为了实现对一个对象的竞争同步，需要指定访问共享数据的方法是同步的。同步机制实现如下：每个 Java 对象都有锁，同步方法在允许执行前，必须先获得对象的锁，这会在该方法执行时阻止其他方法在该对象上执行。同步方法完成执行时，即使是因为出现异常而完成执行，也会释放所运行对象上的锁。考虑以下框架性的类定义：

---

8. 优先级的数量是实现相关的，因此在某些实现中可能少于或多于 10 级。

```
class ManageBuf {
  private int [100] buf;
  ...
  public synchronized void deposit(int item) { ... }
  public synchronized int fetch() { ... }
  ...
}
```

ManageBuf 中的两个方法都定义为 synchronized，这就防止在相同对象上执行时彼此中断，即使它们是由独立的线程所调用。

所有方法都是同步方法的对象实际上是一个管程。请注意，对象可以有一个或多个同步方法，以及一个或多个非同步方法。非同步方法可以随时在对象上运行，甚至在执行同步方法时，也可以执行非同步方法。

在某些情况下，处理共享数据结构的语句数目，比所在方法中其他语句的数目少得多。在这种情况下，让修改共享数据结构的代码段同步，比同步整个方法要好。这可以通过所谓的同步语句来实现，其一般形式是

```
synchronized(表达式){
语句
}
```

其中，表达式必须对一个对象求值，语句可以是单条语句或是复合语句。对象在语句或复合语句执行时被锁定，因此语句或复合语句的执行完全像是一个同步方法的方法体。

定义有同步方法的对象必须有一个相关联的队列，当同步方法操作该对象时，将其他试图执行它的同步方法存储起来。实际上，每个对象都有一个称为内在条件队列的队列。这一队列是隐式提供的。当同步方法完成了在对象上的执行，那么在对象等待队列中等待的方法(如果有的话)就放入任务就绪队列。

13.7.5 合作同步

Java 中的合作同步使用 wait、notify 和 notifyAll 方法来实现，所有这些方法都在所有 Java 类的根类 Object 中定义。除了 Object 以外的所有类都继承了这些方法。每个对象都有一个等待列表，包含所有在该对象上调用 wait 的线程。调用 notify 方法可以告诉某个等待中的线程，它等待的事件已经发生。无法确定哪一个线程被 notify 唤醒，因为 Java 虚拟机(Java Virtual Machine，JVM)随机从线程对象的等待列表中选择一个线程。正因为如此，以及等待线程可能在等待不同的条件，所以通常使用 notifyAll 方法而不是 notify。notifyAll 方法唤醒对象等待列表中的所有线程，把它们放在任务就绪队列中。

wait、notify 和 notifyAll 方法只能在同步方法中调用，因为它们使用了由同步方法放置在对象上的锁。对 wait 的调用总是放在 while 循环中，由方法所等待的条件来控制。while 循环是必不可少的，因为唤醒线程的 notify 或 notifyAll 可能由于改变了条件(该条件不是线程等待的条件)而被调用。如果调用了 notifyAll，所等待的条件为真的可能性甚至更小。由于使用了 notifyAll，在上一次条件测试后，其他线程可能已经将条件修改为假。

wait 方法可以抛出 InterruptedException，这是 Exception 的子类。Java 的异常处理在第 14 章介绍。因此任何调用 wait 的代码必须捕获 InterruptedException。假设等待的条件称为 theCondition，使用 wait 的常规方法如下：

```
try {
  while (!theCondition)
    wait();
  -- Do whatever is needed after theCondition comes true
}
catch(InterruptedException myProblem) { ... }
```

下面的程序为存储 int 值实现了一个循环队列，它演示了合作同步和竞争同步。

```
// Queue
// This class implements a circular queue for storing int
// values. It includes a constructor for allocating and
// initializing the queue to a specified size. It has
// synchronized methods for inserting values into and
// removing values from the queue.

class Queue {
  private int [] que;
  private int nextIn,
              nextOut,
              filled,
              queSize;
  public Queue(int size) {
    que = new int [size];
    filled = 0;
    nextIn = 1;
    nextOut = 1;
    queSize = size;
} //** end of Queue constructor

public synchronized void deposit (int item)
       throws InterruptedException {
  try {
    while (filled == queSize)
      wait();
    que [nextIn] = item;
    nextIn = (nextIn % queSize) + 1;
    filled++;
    notifyAll();
  } //** end of try clause
  catch(InterruptedException e) {}
} //** end of deposit method

public synchronized int fetch()
    throws InterruptedException {
  int item = 0;
```

```
  try {
    while (filled == 0)
      wait();
    item = que [nextOut];
    nextOut = (nextOut % queSize) + 1;
    filled--;
    notifyAll();
  } //** end of try clause
  catch(InterruptedException e) {}
  return item;
 } //** end of fetch method
} //** end of Queue class
```

请注意，这里的异常处理程序(catch)什么都不做。

定义生产者对象和消费者对象的类，可以使用 Queue 类，其定义如下：

```
class Producer extends Thread {
  private Queue buffer;
  public Producer(Queue que) {
    buffer = que;
  }
  public void run() {
  int new_item;
  while (true) {
    //-- Create a new_item
    buffer.deposit(new_item);
  }
 }
}

class Consumer extends Thread {
  private Queue buffer;
  public Consumer(Queue que) {
    buffer = que;
  }
  public void run() {
    int stored_item;
    while (true) {
      stored_item = buffer.fetch();
      //-- Consume the stored_item
    }
  }
}
```

以下代码创建了一个 Queue 对象、一个 Producer 对象和一个 Consumer 对象，后两者都依附在 Queue 对象上并开始执行：

```
Queue buff1 = new Queue(100);
Producer producer1 = new Producer(buff1);
Consumer consumer1 = new Consumer(buff1);
```

```
producer1.start();
consumer1.start();
```

可以将 Producer 和 Consumer 中的一个或两个都定义成 Runnable 接口的实现，而不是 Thread 的子类。唯一的区别是在第一行，现在是

```
class Producer implements Runnable{...}
```

为了创建并运行这个类的一个对象，仍然需要创建一个连接到该对象的 Thread 对象，如下面的代码所示：

```
Producer producer1 = new Producer(buff1);
Thread producerThread = new Thread(producer1);
producerThread.start();
```

请注意，缓冲区对象传递给 Producer 的构造函数，Producer 对象传递给 Thread 的构造函数。

13.7.6 非阻塞同步

Java 中的一些类可以控制对某些不包含阻塞或等待状态的变量的访问。java.util.concurrent.atomic 包定义的类允许对 int、long 和 boolean 等基本类型变量，以及引用和数组进行某些非阻塞性的同步访问。例如，AtomicInteger 类定义了 getter 和 setter 方法，以及执行相加、递增和递减操作的方法。这些操作都是原子性的，即它们不能被打断，所以不需要锁定就可以保证多线程程序中受影响变量的值的完整性。这是细粒度的同步——仅一个变量。大多数机器现在都为 int 和 long 类型的这些操作提供了原子性指令，因此它们常常很容易实现(不需要隐式锁定)。

非阻塞性同步的优点是高效。在竞争过程中不会出现非阻塞性的访问，因此程序的执行不会变慢，通常比使用同步访问的程序更快。在竞争过程中出现的非阻塞性访问肯定比使用同步的访问更快，因为后者需要挂起，重新调度线程。

13.7.7 显式锁定

Java 5.0 将显式锁定引入为同步方法和阻塞(它们提供了隐式锁定)的替代方案。Lock 接口声明了 lock、unlock 和 tryLock 方法。预定义的 ReentrantLock 类实现了 Lock 接口。要锁定一个代码块。可以使用下面的语句：

```
Lock lock = new ReentrantLock();
...
Lock.lock();
try {
   // The code that accesses the shared data
} finally {
  Lock.unlock();
}
```

这些框架代码创建了一个 Lock 对象，在该对象上调用 lock 方法。接着使用 try 块封装重要的代码。对 unlock 的调用放在 finally 子句中，以保证无论在 try 块中发生了什么，都会释放该锁定。

至少有两种情形应使用显式锁定，而不是隐式锁定。首先，如果应用程序需要尝试获得一个锁，但不能永远等待它，Lock 接口为此包含了一个方法 tryLock，它带有一个时间限制参数。如果在该时间限制内没有获得锁定，就继续执行调用 tryLock 后面的语句。第二，不方便构建锁定-解锁对块时，也应使用显式锁定。隐式锁定总是在该锁定所在的复合语句尾解锁。显式锁定可以在代码的任何地方解锁，无论程序的结构如何。

使用显式锁定的一个危险(它不是使用隐式锁定的危险)是省略了解锁操作。隐式锁定在锁定块的末尾隐式解锁，而显式锁定在显式解锁之前一直处于锁定状态，显式解锁可能永远都不会发生。

如前所述，每个对象都有内在的条件队列，它存储了等待对象上某个条件的线程。wait、notify 和 notifyAll 方法是内在条件队列的 API。因为每个对象都只能有一个条件队列，所以队列可以有等待不同条件的线程。例如，缓冲区示例的队列 Queue 可以有等待两个条件(filled == queSize 或 filled == 0)的线程。所以缓冲区使用 notifyAll(如果它使用 notify，就只能唤醒一个线程，该线程等待的条件可能不是实际为真的条件)。但是，notifyAll 使用起来很昂贵，因为它会唤醒在对象上等待的所有线程，这些线程都必须检查它们的条件，来确定哪个线程在运行。而且，要检查条件，线程还必须先获得对象上的锁定。

使用内在条件队列的一种替代方案是 Condition 接口，它使用与 Lock 对象相关的条件队列。它还声明了 wait、notify 和 notifyAll 的替代方法 await、signal 和 signalAll。一个 Lock 对象可以有任意多个 Condition 对象。通过 Condition 可以使用 signal，而不是 signalAll，这样更容易理解、更高效，其部分原因是上下文的 switch 会更少。

13.7.8　评价

Java 对并发的支持相对简单但高效。所有的 Java run 方法都是施动者任务，没有通信机制，但通过共享数据来通信的方法除外，就像 Ada 任务的相互通信那样。Ada 的任务由于是重量级线程，易于分布到不同的处理器，特别是具有不同内存的处理器，这些处理器可以在不同地方的不同计算机上。这类系统不能使用 Java 的轻量级线程。

13.8　C#线程

虽然 C#中的线程松散地基于 Java 的线程，但有着巨大差别。以下是对 C#线程的简要综述。

13.8.1　基本线程操作

Java 只能使用 run 方法，而任何一个 C#方法都可以在自己的线程中运行。创建 C#线

程时，它们会关联上预定义委托 ThreadStart 的实例。在开始执行线程时，它的委托就有要运行的方法的地址。所以线程的执行通过其相关的委托来控制。

创建一个 Thread 对象就创建了 C#的线程。必须向 Thread 的构造函数发送一个预定义委托类 ThreadStart 的实例，向这个实例发送在线程中运行的方法的方法名。例如：

```
public void MyRun1() { ... }
...
Thread myThread = new Thread(new ThreadStart(MyRun1));
```

在这个例子中，创建了一个线程 myThread，其委托指向方法 MyRun1。所以在线程开始运行时，它会调用地址在其委托中的方法。在这个例子中，myThread 是委托，MyRun1 是方法。

与 Java 一样，C#也有两类线程：施动者和服务器。施动者线程不会特别地调用，而是直接启动。它们执行的方法也不带参数或返回值。与 Java 一样，创建线程并没有启动它的并发执行。对于施动者线程，必须通过 Thread 类的一个方法来请求线程的执行，在 C#中该方法名为 Start，如

```
myThread.Start();
```

与 Java 一样，使用类似的 Join 方法，可以让线程在继续执行前，等待另一个线程完成执行。例如，假定线程 A 有如下调用：

```
B.Join();
```

则线程 A 被阻塞，直到线程 B 退出为止。

Join 方法可以带一个 int 参数，它指定了调用者等待线程完成的毫秒数。

线程可以用 Sleep 来挂起指定的时间。Sleep 是 Thread 的一个公有静态方法，Sleep 的参数是毫秒数。与 Java 的相关函数不同，C#的 Sleep 不会抛出异常，因此不需要在 try 语句块中调用。

线程可以用 Abort 方法终止，虽然该方法实际上并不删除线程。相反，它抛出可被线程捕获的 ThreadAbortException。当线程捕获这个异常时，通常会释放所分配的资源，然后结束(到达代码的结尾处)。

服务器线程仅在通过其委托调用时才运行。这些线程称为服务器，是因为它们在被请求时提供了某个服务。服务器线程比施动者线程更有趣，因为它们通常与其他线程交互操作，而且常常必须与其他线程同步执行。

如第 9 章所述，任何 C#方法都可以通过委托间接调用。这种调用可以通过把委托对象当作方法名来实现。这实际上是调用委托方法 Invoke 的缩写。所以，如果委托对象名是 chgfun1，它引用的方法带一个 int 参数，就可以用如下语句调用这个方法：

```
chgfun1(7);
chgfun1.Invoke(7);
```

这些调用是同步的，即调用该方法时，调用者被阻塞，直到方法执行完为止。C#还支持异步调用在线程中执行的方法。线程异步调用时，被调用线程和调用线程会并发执行，因为调用者在被调用线程执行期间，没有被阻塞。

线程通过委托实例方法 BeginInvoke 来异步调用，给 BeginInvoke 方法发送委托方法的参数和另外两个参数，其中一个参数的类型是 AsyncCall- back，另一个参数的类型是 object。BeginInvoke 返回一个实现了 IAsync- Result 接口的对象。委托类也定义了 EndInvoke 实例方法，它带一个 IAsyncResult 类型的参数，其返回类型与封装在委托对象中的方法相同。要异步调用线程，就用 BeginInvoke 调用它。现在，给后两个参数使用 null。假定有如下方法声明和线程定义：

```
public float MyMethod1(int x);
...
Thread myThread = new Thread(new ThreadStart(MyMethod1));
```

下面的语句异步调用 MyMethod：

```
IAsyncResult result = myThread.BeginInvoke(10, null,
   null);
```

被调用线程的返回值用 EndInvoke 方法提取，EndInvoke 方法将 IAsyncResult 类型的对象(由 BeginInvoke 返回)作为参数。EndInvoke 返回被调用线程所返回的值。例如，要获取调用 MyMethod 的 float 结果，可以使用下面的语句：

```
float returnValue = EndInvoke(result);
```

如果在被调用线程执行期间，调用者必须继续执行某个工作，就必须采用某种方式确定被调用线程何时结束。为此，IAsyncResult 接口定义了 IsCompleted 属性。被调用线程在执行过程中，调用者可以包含一些代码，执行代码可以在一个条件为 IsCompleted 的 while 循环中执行。例如：

```
IAsyncResult result = myThread.BeginInvoke(10, null, null);
while(!result.IsCompleted) {
  // Do some computation
}
```

这是在调用线程中执行某个任务，同时等待被调用线程完成其工作的有效方式。但是，如果 while 循环中的计算量相当小，这种方式就很低效(因为测试 IsCompleted 需要一定的时间)。另一种方法是给被调用线程提供一个委托和回调方法的地址，并在被调用线程结束时调用该回调方法。委托发送为 BeginInvoke 的倒数第二个参数。例如下面对 BeginInvoke 的调用：

```
IAsyncResult result = myThread.BeginInvoke(10,
              new AsyncCallback(MyMethodComplete), null);
```

回调方法在调用者中定义。这种方法常常仅设置一个布尔变量，例如把 isDone 设置为

true。无论被调用线程执行多长时间，回调方法都只调用一次。

13.8.2 同步线程

C#线程有三种同步方式：Interlocked 类、lock 语句和 System.Threading 命名空间中的 Monitor 类，每种机制都是为特定需要而设计的。当需要同步的唯一操作是递增和递减一个整数时，就使用 Interlocked 类，这两个操作通过 Interlocked 的两个方法 Increment 和 Decrement 原子地实现，这两个方法以整数的引用作为参数。例如，为了增加一个线程中的共享整数 counter，可以使用

```
Interlocked.Increment(ref counter);
```

lock 语句用于标记线程中的一段关键代码，其语法如下：

```
lock(标志) {
   // The critical section
}
```

如果要同步的代码在私有实例方法中，标志就是当前对象，所以 this 用作 lock 的标志。如果要同步的代码在公有实例方法中，就创建 object 的一个新实例(在包含要同步代码的方法的类中)，将对它的引用作为 lock 的标志。

Monitor 类定义了五个方法：Enter、Wait、Pulse、PulseAll 和 Exit，可用于提供更多的线程同步控制。Enter 方法以一个对象的引用为参数，标记该对象上线程同步的开始。Wait 方法将线程的执行挂起，并通知.NET 的通用语言运行(Common Language Runtime，CLR)环境：该线程希望下次有机会时恢复执行。Pulse 方法也以一个对象的引用为参数，通知一个正在等待的线程：现在有机会再次运行。PulseAll 与 Java 的 notifyAll 相似。等待中的线程以它们调用 wait 方法的顺序依次运行。Exit 方法结束线程的关键段。

lock 语句编译到管程中，所以 lock 是管程的缩写。需要其他控制时(例如使用 Wait 和 PulseAll)，使用管程。

.NET 4.0 添加了一组泛型并发数据结构，包括队列、栈和盒的结构[9]。这些新类是线程安全的，这表示它们可以用于多线程程序，而不需要程序员考虑竞争同步。System.Collections.Concurrent 命名空间定义了这些类，其名称是 ConcurrentQueue<T>、ConcurrentStack<T>和 ConcurrentBag<T>。所以生产者-消费者队列程序可以用 C#编写，将 ConcurrentQueue<T>用于数据结构，且不需要考虑竞争同步。因为这些并发集合在.NET 中定义，所以它们也可用于所有其他.NET 语言。

13.8.3 评价

C#的线程比前辈语言 Java 的线程有少许改进。例如，任何方法都可以在自己的线程中运行。而在 Java 中，只有 run 方法可以在自己的线程中运行。Java 只支持施动者线程，

9. 盒是未排序的对象集。

而 C#支持施动者线程和服务器线程。C#的线程终止也更清晰(调用方法 Abort 比将线程指针设置为 null 更优雅)。C#中线程的同步执行更复杂，因为 C#有几种不同的机制，每一种都有特定应用。Java 的 Lock 变量类似于 C#的锁定，但在 Java 中，锁定必须用 unlock 调用显式解锁。这提供了生成错误代码的另一种方式。C#线程与 Java 线程一样是轻量级的，因此尽管它们更高效，但不如 Ada 的任务那样通用。与本章介绍的其他非函数式语言相比，C#提供的并发集合类是它的另一个优点。

13.9　函数式语言中的并发

本节简要介绍几种函数式编程语言对并发的支持。

13.9.1　Multilisp

Multilisp (Halstead, 1985)是对 Scheme 的一个扩展，它允许程序员指定可以并发执行的程序部分。这种形式的并发是隐含的，程序员仅告诉编译器(或解释器)，程序的某些部分可以并发执行。

程序员在告诉系统可能并发执行的信息时，一种方式是使用 pcall 结构。如果函数调用嵌入在 pcall 结构中，该函数的参数就可以并发计算。例如下面的 pcall 结构：

```
(pcall f a b c d)
```

其中函数是 f，参数是 a、b、c 和 d。pcall 的作用是，该函数的参数可以并发计算(任意多个或全部参数都可以是复杂的表达式)。可惜，这个过程是否可以安全地使用，即不影响函数计算的语义，是程序员的责任。如果语言不允许产生副作用，或者程序员设计的函数没有副作用，或者至少副作用有限，这就是一个很简单的事情。但是，Multilisp 允许存在某些副作用。如果函数在编写时没有避免产生副作用，程序员就很难决定 pcall 结构是否能安全地使用。

Multilisp 的 future 结构更有趣，并发的潜在效率更高。与 pcall 相同，函数调用也封装在 future 结构中。这种函数在一个单独的线程中执行，而其父线程继续执行，直到需要使用函数的返回值为止。如果在父线程需要函数的结果时，函数还没有执行完，父线程就等待函数执行完。

如果函数有两个或多个参数，它们也可以封装在 future 结构中，此时，它们的计算在单独的线程中并发执行。

这些仅是 Multilisp 中对 Scheme 的增强功能。

13.9.2　并发 ML

并发 ML(Concurrent ML，CML)是 ML 的一个扩展，它包括一种线程形式和一种传递同步信息以支持并发的形式。该语言在 Reppy(1999)中做了完整的描述。

在 CML 中，线程用 spawn 基元创建，它把函数作为其参数。在许多情况下，函数指定为匿名函数。一旦创建了线程，函数就在新线程中开始执行，并舍弃函数的返回值。函数产生的结果是生成的输出或通过与其他线程的交流而得到的输出。父线程(生成新线程的线程)或子线程(新线程)可以先停止，而不会影响另一个线程的执行。

通道提供了线程之间的交流方式。通道用 channel 构造函数创建。例如，下面的语句创建了一个任意类型的通道 mychannel：

```
let val mychannel = channel()
```

通道上的两个主要操作(函数)是发送(send)和接收(recv)消息。消息的类型是从发送操作中推断出来的。例如，下面的函数调用发送了整数值 7，因此通道的类型推断为整数：

```
send(mychannel, 7)
```

recv 函数把通道作为其参数，其返回值是它接收到的值。

因为 CML 通信是同步的，所以只有发送者和接收者都准备好，才能发送和接收消息。如果一个线程把消息发送到通道中，但没有其他线程准备接收该通道上的消息，发送者就被阻塞，等待另一个线程在该通道上执行 recv 操作。同样，如果一个线程在通道上执行 recv 操作，但没有其他线程准备在该通道上发送消息，则运行 recv 操作的线程就被阻塞，等待另一个线程在该通道上发送消息。

因为通道是类型，所以函数把它们作为参数。

与 Ada 的同步消息传送一样，CML 同步消息传送的一个问题是当多个通道都接收消息时，确定由哪个通道接收消息。这里也使用相同的解决方案：保护命令 do-od 结构，它在消息中随机选择，以区分通道。

CML 的同步机制是事件。这个复杂机制的解释超出了本章和本书的范围。

13.9.3 F#

F#对并发的支持基于.NET 类 System.Threading.Thread，C#也使用了这个类。例如，假定函数 myConMethod 要在自己的线程上运行。则下面的函数在调用时，会创建该线程，并在新线程中开始运行函数：

```
let createThread() =
    let newThread = new Thread(myConMethod)
    newThread.Start()
```

在 C#中，需要创建预定义委托 ThreadStart 的一个实例，给其构造函数发送子程序的名称，再把新委托实例发送为 Thread 构造函数的一个参数。而在 F#中，如果函数需要把委托作为参数，就可以发送 λ 表达式或函数，编译器就会像发送了委托那样处理。所以在上面的代码中，myConMethod 函数发送为 Thread 构造函数的参数，但实际发送的是 ThreadStart 的一个新实例(它发送给 myConMethod)。

Thread 类定义了 Sleep 方法，该方法使调用它的线程睡眠其参数指定的毫秒数。

共享的不可变数据不需要在访问它的线程之间同步。但是，如果共享数据是可变的，这在 F#中是可能的，就需要锁定线程，以禁止多个线程试图改变共享数据而造成数据失败。在函数操作可变的变量时，就可以用 lock 函数锁定该变量，以提供对对象的同步访问。这个 lock 函数带两个参数，第一个是要改变的变量，第二个是改变变量的 λ 表达式。

可变的堆-分配变量是 ref 类型。例如，下面的声明就创建了 ref 变量 sum，其初始值为 0：

```
let sum = ref 0
```

ref 类型的变量可以在 λ 表达式中改变，该 λ 表达式使用 ALGOL/Pascal/Ada 赋值操作符:=。ref 变量必须用感叹号(!)作为前缀，才能获取其值。在下面的语句中，λ 表达式给可变的变量 sum 加上 x 的值时，锁定了 sum：

```
lock(sum) (fun () -> sum := !sum + x)
```

线程可以异步调用，与 C#一样，也使用相同的子程序 BeginInvoke 和 EndInvoke，以及 IAsyncResult 接口，以便于确定异步调用的线程是否执行完毕。

如前所述，F#中.NET 的并发泛型集合可用于其程序。在建立多线程程序，以队列、堆栈或盒的形式建立共享数据结构时，这可以减少大量的编程工作量。

13.10 语句级并发

本节简要介绍语句级并发的语言设计。从语言设计的角度来看，这种设计的目的是提供一种机制，使程序员能够告诉编译器，如何将程序映像到多处理器体系结构上[10]。

本节只介绍一种语言中用于语句级并发的一组语法结构：高性能 Fortran。

高性能 Fortran

高性能 Fortran(High-Performance Fortran，HPF)(ACM，1993b)是 Fortran 90 的一组扩展，目的是允许程序员将信息告诉编译器，帮助编译器在多处理器计算机上优化程序的执行。HPF 包含新的说明语句和固有的(内建的)子程序。本节只介绍一些 HPF 语句。

HPF 基本的说明语句用于说明处理器的数目、这些处理器内存上的数据分布，以及在内存布局上数据与其他数据的对齐方式。HPF 说明语句以特殊注释的形式出现在 Fortran 程序中。每一条说明语句都以前缀!HPF$开始，其中!是 Fortran 90 中用于开始一行注释的符号。这个前缀使说明语句对 Fortran 90 编译器不可见，但很容易被 HPF 编译器识别。

PROCESSORS 说明语句的形式是

```
!HPF$ PROCESSORS procs(n)
```

这条语句用来向编译器说明本程序所生成的代码可用的处理器数目。这条信息与其他

10. 虽然 ALGOL 68 包含一个信号量类型，用于处理语句级并发，但这里不介绍信号量的这种应用。

说明结合在一起告诉编译器，如何将数据分布到与各处理器相关联的内存上。

DISTRIBUTE 和 ALIGN 说明用于给编译器提供不共享内存的机器信息，即每个处理器都有自己的内存。假定处理器对自己内存的访问要快于对其他处理器内存的访问。

DISTRIBUTE 语句说明了哪些数据要分布存储，以及使用哪种分布方式。它的形式是：

```
!HPF$ DISTRIBUTE(分布方式) ONTo procs: : 标识符列表
```

在这条语句中，分布方式可以是 BLOCK 或 CYCLIC，标识符列表是要分布存储的数组变量名。以 BLOCK 方式分布的变量被分成相等的 n 组，每一组都包含连续的若干数组元素，平均分布到所有处理器的内存中。例如，LIST 数组有 500 个元素，以 BLOCK 方式分布到 5 个处理器上，那么 LIST 的前 100 个元素存储在第一个处理器的内存中，第二组 100 个元素存储在第二个处理器的内存中，以此类推。CYCLIC 分布方式说明数组的各个元素循环地存储在处理器的内存中。例如，如果 LIST 以 CYCLIC 方式分布，仍然是分布在 5 个处理器上，那么 LIST 的第一个元素存储在第一个处理器的内存中，第二个元素存储在第二个处理器的内存中，以此类推。

ALIGN 语句的形式是：

```
ALIGN 数组 1 的元素 WITH 数组 2 的元素
```

ALIGN 用于将一个数组与另一数组的分布关联起来。例如，

```
ALIGN list1(index) WITH list2(index+1)
```

说明对于所有的 index 值，list1 的第 index 个元素与 list2 的第 index+1 个元素存储在相同处理器的内存中。ALIGN 中对两个数组的引用在程序某些语句中同时出现。将它们放在相同的内存中(意味着相同的处理器)，可确保对它们的引用尽可能接近。

考虑以下示例代码段：

```
      REAL list_1 (1000), list_2 (1000)
      INTEGER list_3 (500), list_4 (501)
!HPF$ PROCESSORS proc (10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: list_1, list_2
!HPF$ ALIGN list_3 (index) WITH list_4 (index+1)
    ...
    list_1 (index) = list_2 (index)
    list_3 (index) = list_4 (index+1)
```

在这些赋值语句的每一次执行中，所引用的两个数组元素将存储在相同处理器的内存中。

HPF 说明语句向编译器提供的信息可以用来优化所生成的代码，也可以不用。编译器实际的动作取决于它的复杂程度，以及目标机器特定的体系结构。

FORALL 语句说明一组可并发执行的赋值语句。例如，

```
FORALL (index = 1:1000)
  list_1(index) = list_2(index)
END FORALL
```

说明将 list_2 的元素赋值给 list_1 的相应元素。但是，该赋值必须遵循如下顺序：先计算所有 1000 个赋值的右边部分，再进行赋值。这就允许所有赋值语句并发执行。除了赋值语句之外，FORALL 语句可以出现在 FORALL 结构体中。FORALL 语句非常适用于向量机，在向量机中，同一个指令会应用于许多数据值，这些数据值通常在一个或多个数组中。HPF 的 FORALL 语句在 Fortran 95 和后续的 Fortran 版本中都有。

我们只是简要介绍了 HPF 的一小部分功能，不过已经足够使读者了解，在可能拥有大量处理器的计算机上进行程序设计时可用的语言扩展。

C# 4.0(和其他.NET 语言)包含的两个方法有点像 FORALL。它们是循环控制语句(其中迭代可以展开)和并发执行的循环体。这些语句是 Parallel.For 和 Parallel.ForEach。

小 结

并发执行可以是指令级、语句级或者子程序级的。当实际上使用多个处理器来执行并发单元时，称为物理并发；如果并发的单元在一个处理器上执行，称为逻辑并发。所有并发的基本概念模型都可以称为逻辑并发。

大多数多处理器计算机都属于两种类别：SIMD 或 MIMD。MIMD 计算机可以是分布式的。

支持子程序级并发的语言必须提供的两个基本功能，是对共享数据结构的互斥访问(竞争同步)和任务间的合作(合作同步)。

任务可以处在五种状态下：新建、就绪、运行、阻塞或死亡。

为了支持并发，有时不是设计语言结构，而是使用库，例如 OpenMP。

语言支持并发的设计问题是，如何实现竞争同步和合作同步，应用程序如何影响任务的调度，任务开始和结束执行的时间和方式，任务创建的时间和方式。

信号量是一个数据结构，由一个整数和一个任务描述符队列构成。信号量可用于实现并发任务间的竞争同步和合作同步。信号量很容易用错，导致编译器、链接器或运行时系统无法检测的错误。

管程是一种数据抽象，以一种自然的方式实现了任务间共享数据的互斥访问，有若干程序设计语言支持管程，包括 Ada、Java 和 C#。具有管程的语言必须与某种形式的信号量一起，才能实现合作同步。

并发消息传递模型的基本概念是任务发送每个消息，以同步它们的执行。

Ada 基于消息传递模型，提供了复杂而高效的语言结构用于并发。Ada 的任务是重量级任务。任务通过会合机制(同步的消息传递)与彼此通信。会合就是一个任务从另一个任务接受消息的动作。Ada 包含了控制任务会合并发的简单方法和复杂方法。

Ada 95+包含额外的并发支持功能，主要是受保护对象。Ada 95+以任务和受保护对象两种方式支持管程。

Java 以一种相对简单而高效的方式支持轻量级的单元并发。从 Thread 继承的类或实现了 Runnable 接口的类，都可以重写 run 方法，使该方法的代码能够与其他的这种方法或

main 程序并发执行。将访问共享数据的方法定义为隐式同步，就可以实现竞争同步。小段的代码也可以隐式同步。所有方法都是同步方法的类就是管程。合作同步用方法 wait、notify 和 notifyAll 来实现。Thread 类还提供了 sleep、yield、join 和 interrupt 方法。

Java 通过 Semaphore 类及其 acquire 和 release 方法，直接支持计数信号量。它还有一些类提供非阻塞的原子操作，例如整数的相加、递增和递减操作。Java 也通过 Lock 接口、ReentrantLock 类及其 lock 和 unlock 方法提供了显式锁定。除了使用 synchronized 实现隐式同步外，Java 还为 int、long 和 boolean 类型的变量以及引用和数组提供了隐式的非阻塞同步。在这些情况下，Java 提供了原子化的 getter、setter、相加、递增和递减操作。

C#的并发支持基于 Java 的并发支持，但稍微复杂一些。任何方法都可以运行在线程中，支持施动者和服务器线程。所有线程都通过相关的委托来控制。服务器线程可以用 Invoke 同步调用，或通过 BeginInvoke 异步调用。回调方法地址可以发送给被调用线程。使用 Interlocked 类、Monitor 类和 lock 语句支持三种线程同步。Interlock 类提供了原子递增和递减操作。

所有.NET 语言都为栈、队列和盒使用了泛型并发数据结构，因为竞争同步是隐式的。

Multilisp 略微扩展了 Scheme，允许程序员告诉实现代码，哪些程序部分可以并发执行。Concurrent ML 扩展了 ML，以支持一种形式的线程和这些线程之间的一种同步传递消息方式。这个消息传递是用通道设计的。F#程序可以访问所有.NET 支持类，以并发执行。线程之间共享的可变数据可以同步访问。

高性能 Fortran 包含了一些语句，说明如何将数据分布到同多个处理器相连接的内存单元上，还包含了一些语句说明可并发执行的语句集。

文献注释

Andrews 和 Schneider(1983)、Holt 等(1978)和 Ben-Ari(1982)对并发的一般性问题有详尽的介绍。

Brinch Hansen(1977)提出了管程的概念，介绍了管程在并发 Pascal 中的实现。

Hoare(1978)和 Brinch Hansen(1978)介绍了并发单元控制的消息传递模型早期的发展过程。在 Ichbiah 等(1979)中可以找到 Ada 任务模型发展过程的详细介绍。ARM(1995)中详细介绍了 Ada 95，ACM(1993b)中介绍了高性能 Fortran。

复 习 题

1. 程序并发有哪三个可能的层次？
2. 描述 SIMD 计算机的逻辑体系结构。
3. 描述 MIMD 计算机的逻辑体系结构。
4. SIMD 计算机对哪一级的程序并发支持最好？
5. MIMD 计算机对哪一级的程序并发支持最好？
6. 描述矢量处理器的逻辑体系结构。

7. 物理并发和逻辑并发的区别是什么？
8. 程序中的控制线是什么？
9. 为什么协同程序称为准并发？
10. 什么是多线程程序？
11. 学习语言的并发支持有哪四个原因？
12. 什么是重量级任务？什么是轻量级任务？
13. 给出任务、同步、竞争同步、合作同步、活性、竞态条件和死锁的定义。
14. 哪一类任务不需要同步？
15. 描述任务可能的五种不同状态。
16. 什么是任务描述符？
17. 在语言的并发支持意义下，守卫是什么意思？
18. 任务就绪队列的用途是什么？
19. 语言并发支持的主要设计问题是什么？
20. 描述信号量的等待和释放操作的行为。
21. 什么是二元信号量？什么是计数信号量？
22. 使用信号量来实现同步的主要问题是什么？
23. 与信号量相比，管程的优点是什么？
24. 哪三种常见的程序设计语言实现了管程？
25. 给出会合、accept 子句、entry 子句、施动者任务、服务器任务、扩展的 accept 子句、开放的 accept 子句、关闭的 accept 子句和已完成任务的定义。
26. 用管程实现并发和用消息传递实现并发，哪一个更具一般性？
27. Ada 任务是静态创建的还是动态创建的？
28. 扩展的 accept 子句的用途是什么？
29. Ada 任务是如何实现合作同步的？
30. Ada 中 terminate 子句的用途是什么？
31. Ada 95 在提供对共享数据对象的访问方面，受保护对象比任务好在哪里？
32. Java 的哪一种程序单元可以与应用程序的 main 方法并发运行？
33. Java 线程是轻量级的还是重量级的？
34. Java 的 sleep 方法有什么用？
35. Java 的 yield 方法有什么用？
36. Java 的 join 方法有什么用？
37. Java 的 interrupt 方法有什么用？
38. 哪两种 Java 语言结构可以声明为 synchronized？
39. Java 如何设置线程的优先级？
40. Java 线程是施动者任务、服务器线程，还是两者？
41. 描述 Java 用于支持合作同步的三种方法的行为。

42. 哪一种 Java 对象是管程？
43. 为什么 Java 包含 Runnable 接口？
44. 哪两个方法与 Java 的 Semaphore 对象一起使用？
45. Java 中非阻塞式同步的优点是什么？
46. Java 中 AtomicInteger 类有什么方法，这个类的作用是什么？
47. Java 如何支持显式锁定？
48. C#线程可以运行哪些方法？
49. C#线程是施动者任务、服务器线程，还是两者？
50. C#线程可以用哪两种方式同步调用？
51. C#线程如何异步调用？
52. C#如何提取异步调用线程中的返回值？
53. C#的 Sleep 方法与 Java 的 sleep 方法有何区别？
54. C#的 Abort 方法有什么用？
55. C#的 Interlocked 类的用途是什么？
56. C#的 lock 语句有什么用？
57. Multilisp 基于什么语言？
58. Multilisp 的 pcall 结构有什么语义？
59. 在 CML 中如何创建线程？
60. F#堆分配可变变量的类型是什么？
61. 为什么 F#的不可变变量需要在多线程程序中同步访问？
62. 高性能 Fortran 的说明语句的用途是什么？
63. 高性能 Fortran 的 FORALL 语句的用途是什么？

习　题

1. 清楚地说明，在支持协同程序而非并发的程序设计环境中，为什么竞争同步不是问题。
2. 当检测到死锁时，系统可采取的最佳动作是什么？
3. 忙碌等待是一种方法，指任务在等待指定事件时不停地检测事件是否发生。这种方法的主要问题是什么？
4. 在 13.3 节的生产者-消费者例子中，假设错误地将消费者进程中的 release(access)换成了 wait(access)，系统执行时会发生什么错误？
5. 根据一本在使用 Intel Pentium 处理器的计算机上进行汇编语言程序设计的书，确定哪些指令可支持信号量的构建。
6. 假设两个任务 A 和 B 必须使用共享变量 Buf_Size。任务 A 把 Buf_Size 加 2，任务 B 把 Buf_Size 减 1。假设这两种算术运算有三个步骤：获取当前值、执行算术运算和把新值放回去。在缺少竞争同步的情况下，可能的事件序列和运算结果的值是什么？假设 Buf_ Size 的初值是 6。

7. 比较 Java 与 Ada 的竞争同步机制。
8. 比较 Java 与 Ada 的合作同步机制。
9. 如果一个管程过程调用了同一管程的另一个过程，会发生什么事情？
10. 说明在任务中使用信号量和使用 Ada 的 when 子句，两者合作同步的相对安全性如何。

编程练习

1. 写一个 Ada 任务来实现通用的信号量。
2. 写一个 Ada 任务来管理共享缓冲区，就像示例中的那样，但要使用编程练习 1 中的信号量任务。
3. 用 Ada 语言来定义信号量，并在共享缓冲区的例子中用信号量来实现合作同步和竞争同步。
4. 用 Java 编写编程练习 3。
5. 用 C#编写本章的共享缓冲区例子。
6. 读取器-写入器问题可以描述如下：共享的内存地址可以由任意数目的任务并发读取，但当一个任务要写入共享内存地址时，它必须具有排他性访问权限。为读取器-写入器问题编写一个 Java 程序。
7. 用 Ada 编写编程练习 6。
8. 用 C#编写编程练习 6。

第14章

异常处理和事件处理

14.1 异常处理概述
14.2 Ada 中的异常处理
14.3 C++中的异常处理
14.4 Java 中的异常处理
14.5 事件处理概述
14.6 Java 的事件处理
14.7 C#中的事件处理

本章介绍对许多当代程序中两个相关部分的编程语言支持，这两个部分是异常处理和事件处理。异常和事件发生的时刻都无法事先确定，最好由特定的语言结构和过程来处理。一些语言结构和过程(例如传播)与异常处理和事件处理很相似。

首先介绍异常处理的基本概念，包括硬件可检测和软件可检测的异常、异常处理程序以及异常的产生；然后介绍并讨论异常处理的设计问题，包括将异常绑定到异常处理程序、控制继续、默认异常处理程序以及异常禁用。这部分之后是介绍和评价三种编程语言 Ada、C++和 Java 的异常处理机制。

本章后面的部分是关于事件处理的。首先介绍事件处理的基本概念，然后对 Java 和 C#的事件处理方法做简要介绍。

14.1 异常处理概述

大多数计算机硬件系统能够检测某些运行时的错误条件，例如浮点溢出。早期编程语言的设计和实现方式，使用户程序不能检测也不能处理错误。在这些语言中，出现这样的错误就会导致程序终止，程序控制转交给操作系统。操作系统对运行时错误的典型反应就是显示一条诊断信息，这条信息可能是有意义的，很有用，也可能非常模糊。在显示消息后，程序就终止了。

然而，在有输入和输出操作时，情况就不一样了。例如，Fortran 的 Read 语句可以发现输入错误和文件结束条件，这两种情况都可以由输入设备的硬件检测到。在这两种情况下，Read 语句都可以给出用户程序中处理这些情况的语句标号。对于文件结束的情况，显然文件结束并不总是错误。在大多数情况下，它只不过是某种处理已经结束、应开始另一种处理的信号。虽然文件结束与总是错误的事件(例如输入过程失败)有显著差别，但是 Fortran 用同一种机制来处理这两种情况。考虑 Fortran 的以下 Read 语句：

```
Read(Unit=5, Fmt=1000, Err=100, End=999) Weight
```

Err 子句说明，如果在读操作中发生错误，程序控制就转移到标号为 100 的语句。End 子句说明，如果读操作遇到文件结尾，程序控制就转移到标号为 999 的语句。因此，Fortran 用简单的分支来处理输入错误和文件结束。

有一类严重错误无法被硬件检测到，但可以被编译器生成的代码检测到。例如，数组下标范围错误几乎无法被硬件检测到[1]，但它们导致的严重错误经常直到程序执行才被发现。

对下标范围错误的检测有时是语言设计所要求的。例如，Java 编译器通常生成代码来检查每个下标表达式的正确性(如果在编译期就可以确定下标表达式不会出现超出范围的值，例如下标是一个字面值，编译器就不会生成这样的代码)。在 C 语言中并不检查下标范围，因为人们认为(现在也这样认为)，这种检查的代价相对于检测错误带来的好处而言并

1. 在 20 世纪 70 年代，有一些计算机可以在硬件中检测到下标范围错误。

不值得。对于某些语言的编译器，可以在程序中，或者在执行编译器的命令中，根据需要选择(如果默认没有打开设置)或关闭(如果默认打开设置)对下标范围的检查。

大多数现代语言的设计中都包含了一种机制，允许程序以标准的方式对某些运行时错误，以及其他程序可检测的非正常事件做出响应。当硬件或系统软件检测到某些事件时，也可通知程序，使程序可以对这些事件做出响应。这些机制统称为异常处理。

一些语言没有包含异常处理最貌似有理的原因，也许是给语言增加了复杂性。

14.1.1　基本概念

我们将硬件检测到的错误(例如磁盘读取错误)和非正常情况(如文件结束，也可由硬件检测到)，都认为是异常。我们进一步将异常的概念扩展至包括软件可检测的(由软件解释程序或用户代码本身检测)错误和非正常情况。因此，我们将异常定义为任意可被硬件或软件检测到的、需要特殊处理的错误或非错误的非正常事件。

检测到异常后，所需要的特殊处理称为异常处理。这种处理由称为异常处理程序的代码单元或代码段来执行。当相关联的事件发生时，就称抛出(raise)异常。在有些基于 C 的语言中，使用 throw 而不是 raise[2]。不同种类的异常需要不同的异常处理程序。对文件结束的检测几乎总是需要一些特殊的程序动作，但是显然这种动作对数组下标范围错误的异常不适合。在某些情况下，唯一的动作就是生成一条错误消息，并有条理地终止程序。

在某些情况下，可能需要暂时忽略某些硬件可检测的错误，例如除以零。这可以通过禁用异常来实现。被禁用的异常可以在以后再次启用。

语言没有单独的或特殊的异常处理机制，并不妨碍对用户自定义的软件可检测的异常进行处理。这种在程序单元中检测到的异常，通常由单元调用者或发起者来处理。一个可能的设计就是发送辅助的参数用作状态变量。状态变量在被调用的子程序中，根据计算的正确性和/或正常情况来赋值。一旦从被调用单元返回，调用者就立即测试状态变量。如果状态变量的值表明发生了异常，就调用异常处理程序(可能位于调用单元中)。许多 C 语言的标准库函数采用这种方法的一种变体：用返回值作为错误指示器。

另一种可能就是向子程序传递一个标号参数。当然，这种方法只能用于允许将标号用作参数的语言。传递标号可以使被调用单元在发生异常时，返回到调用者中别的地方。与第一种可选方法一样，异常处理程序通常是调用单元的一段代码。这是 Fortran 中标号参数的常见用法。

第三种可能是将异常处理程序定义为独立的子程序，其名字作为参数传递给被调用单元。在这种情况下，异常处理的子程序由调用者提供，但在异常抛出时，由被调用单元来调用异常处理程序。这种方法的一个问题是，不管是否需要，每个子程序都要有一个异常处理子程序作为参数，每次调用都要把一个异常处理子程序传给子程序。此外，为了处理

---

2. C++是第一种基于 C 语言并包含异常处理的编程语言，使用 throw 而不是 raise，是因为标准 C 库中包含一个名为 raise 的函数。

不同种类的异常，需要传递不同的异常处理子程序，使代码变得复杂。

如果希望在检测到异常的单元中处理异常，那么异常处理程序就以代码段的形式包含在该单元中。

在语言中内建异常处理肯定有一些优点。首先，如果没有异常处理，检测错误情况所需要的代码会大大扰乱程序。例如，假设子程序中的表达式包含 10 个对矩阵 mat 中元素的引用，每一个引用都可能有下标超出范围的错误。进一步假设语言不要求对下标范围进行检查。如果没有内建下标范围检查，每一个操作前面都需要有代码来检测可能的下标范围错误。例如，考虑以下对 mat 元素的引用(mat 有 10 行 20 列)：

```
if (row >= 0 && row < 10 && col >= 0 && col < 20)
  sum += mat[row][col];
else
  System.out.println("Index range error on mat, row = " +
                     row + " col = " + col);
```

语言中有异常处理，编译器就会在每个数组元素访问前，插入这种检查的机器代码，大大缩短和简化源程序。

语言支持异常处理的另一个优点来自于异常传播。异常传播使一个程序单元中抛出的异常，可以在它的动态或静态祖先的其他单元中处理。这允许一个异常处理程序用于任意数目的不同程序单元。这种复用可以大大减少开发成本、程序规模和程序复杂度。

支持异常处理的语言鼓励用户考虑在程序执行时发生的所有事件，以及如何处理。这种方法比不考虑这种可能性，只简单地希望不会出错要好得多。与这一优点相关的，是需要一种多选结构，来包含对控制表达式所有可能取值的响应，就像 Ada 语言需要的那样。

最后，有一些程序可以用异常处理来简化对不是错误但是非通常情况的处理，如果没有异常处理，程序结构会变得特别费解。

14.1.2 设计问题

现在研究当异常处理系统包含在程序设计语言中时的一些设计问题。这种系统可能允许预定义和用户自定义的异常以及异常处理程序。请注意，预定义的异常是隐式抛出的，而用户自定义的异常必须由用户代码显式抛出。考虑以下子程序框架，它包含一个隐式抛出异常的处理机制。

```
void example() {
  ...
  average = sum / total;
  ...
  return;
/* Exception handlers */
  when zero_divide {
    average = 0;
    printf("Error-divisor (total) is zero\n");
  }
```

```
    ...
}
```

除以零异常是隐式抛出的，它使程序控制转移到适当的处理程序上并执行。

异常处理的第一个设计问题是如何将异常绑定到异常处理程序上。这个问题发生在两个不同层次上。在单元层次上，问题是在一个单元的不同地点抛出的相同异常，如何绑定到单元内不同的处理程序上。例如，在示例子程序中，有一个处理除以零异常的处理程序，是为处理特定语句(所显示的语句)中发生除以零异常而编写的。但是假设该函数包含多个有除法运算的其他表达式。对于那些运算，这个处理很可能并不合适。因此，应该可以将由特定语句抛出的异常绑定到特定的处理程序上，即使许多不同的语句都抛出相同的异常。

在更高的层次上，在抛出异常的单元中没有局部的异常处理程序时，会出现绑定问题。在这种情况下，语言设计者必须决定是否将异常传播到其他单元上，如果是，应该传播到哪里。传播如何发生和传播多远，对异常处理程序的可写性有重要的影响。例如，如果处理程序必须是局部的，就必须写许多处理程序，这会使程序的编写和阅读都很复杂。另一方面，如果传播异常，一个处理程序就可能处理不同程序单元抛出的相同异常，这需要处理程序比希望的更通用。

与将异常绑定到异常处理程序有关的一个问题是，有关异常的信息可否用于处理程序。

历史注释

PL/I(ANSI，1976)开创了允许用户程序直接参与异常处理的概念。这种语言允许用户为一长列语言定义的异常编写异常处理程序。而且，PL/I 引入了用户自定义异常，允许程序创建软件可检测的异常。这些异常使用的机制与内建异常相同。

自从设计了 PL/I 之后，人们做了大量工作，来设计异常处理的替代方法，特别是 CLU (Liskov 等，1984)、Mesa (Mitchell 等，979)、Ada、COMMON LISP(Steele，1990)、ML(Milner 等，1990)、C++、Modula-3(Cardelli 等，1989)、Eiffel、Java 和 C#都包含了异常处理机制。

在异常处理程序执行后，控制转移到处理程序代码之外的程序的某处，或者程序的执行就此终止，这称为处理程序执行后的控制继续问题，或简称继续(continuation)。终止显然是最简单的选择，在许多错误异常的条件下，也是最佳的选择。但是在其他的情况下，特别是与非正常但不是错误的事件相关联的情况，选择继续执行是最佳的。这种设计称为重新开始(resumption)。在这种情况下，必须选择某种规范，来确定执行应从哪里继续，可以是抛出异常的语句、抛出异常的语句后面的语句，或者可能是某个其他单元。返回到抛出异常的语句看起来像是一个好的选择，但是在错误异常的情况下，只有处理程序能够修改导致抛出异常的值或运算，这才是有用的，否则异常会再次抛出。错误异常所需要的修改通常是非常难预测的，即使可以做到，也不是一种合理的做法，这只是让程序去除问题的表象，却并没有消除问题的根源。

图 14-1 显示了将异常绑定到处理程序和控制继续这两个问题。

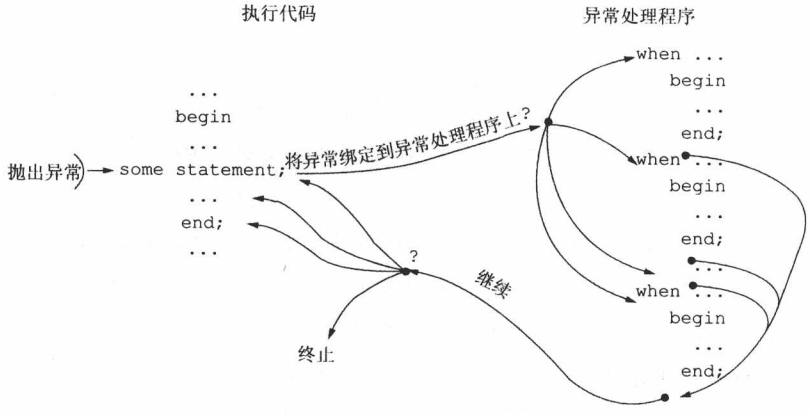


图 14-1 异常处理的控制流

有了异常处理后，子程序的执行就有两种终止方式：执行完成，或遇到异常。在某些情况下，不管子程序的执行是如何终止的，都需要完成某些计算，说明这种计算的功能称为析构(finalization)。选择是否支持析构显然是异常处理的一个设计问题。

另一个设计问题是：如果允许用户自定义异常，那么如何说明这些异常？通常的答案是需要在可抛出异常的程序单元的说明部分声明它们。所声明的异常的作用域，通常是包含声明的程序单元的作用域。

在语言提供预定义异常的情况下，还有其他一些设计问题。例如，语言的运行时系统应该为内建的异常提供默认处理程序，还是应该由用户为所有异常编写处理程序？另一个问题是，用户程序是否可以显式地抛出预定义的异常。如果在某些软件可检测的情况下，用户想使用预定义的异常处理程序，这样做就比较方便。

另一个问题是，硬件可检测的错误是否可以由用户程序来处理。如果不可以，显然所有的异常都是软件可检测的。一个相关的问题是，是否应该有预定义的异常。预定义的异常是隐式地由硬件或系统软件抛出的。

最后的问题是，预定义的或用户自定义的异常是否可以暂时或永久禁用。这个问题有一定的哲学意味，特别是预定义的错误条件。例如，假设语言有一个预定义异常，当发生下标范围错误时抛出。许多人认为应该总是检测下标范围错误，因此程序不应禁止对这类错误进行检测。另一些人认为对于商用软件来说，下标范围检查的代价太高，商用软件的代码应该是没有错误的，不应该出现下标范围错误。

可以将异常处理的设计问题总结如下：

- 如何以及在哪里说明异常处理程序，其作用域是什么？
- 如何将异常绑定到异常处理程序？
- 有关异常的信息可以传递给处理程序吗？
- 在异常处理程序完成执行后，程序在哪里继续执行(这是继续或重新开始的问题)？
- 提供某种形式的析构吗？
- 如何说明用户自定义异常？

- 如果有预定义异常，对于没有提供自己的异常处理程序的程序，应该有默认的异常处理程序吗？
- 可以显式地抛出预定义异常吗？
- 硬件可检测的错误也与可处理的异常一样对待吗？
- 有预定义异常吗？
- 应该允许禁用预定义异常吗？

现在可以研究三种当代编程语言的异常处理机制了。

14.2　Ada 中的异常处理

Ada 中的异常处理是构建更可靠的软件系统的强大工具，它基于两种具有异常处理的早期语言 PL/I 和 CLU 中较好部分的异常处理设计。

14.2.1　异常处理程序

Ada 的异常处理程序通常在可抛出异常的代码的局部(虽然可以传播到其他程序单元)。由于这样可提供相同的引用环境，所以异常处理程序不需要参数，也不允许有参数。因此，如果异常在抛出异常的单元之外的单元内处理，就无法把异常的信息传递给处理程序[3]。

异常处理程序具有以下用 EBNF 表示的一般形式：

when 异常选择{ | 异常选择}=>语句序列

花括号是元符号，表示所包含的部分可以没有或重复任意多次。异常选择的形式是

异常名 | **others**

异常名表明异常处理程序要处理的特定异常。语句序列是处理程序的程序体。保留字 others 表明处理程序要处理没有在其他局部的异常处理程序中指定的异常。

异常处理程序可以包含在程序块中，或者是子程序、包或任务的体内。不管出现在程序块还是单元中，处理程序都聚集在一条 exception 子句内，这条子句必须放在程序块或单元的结尾处。例如，exception 子句的通常形式如下所示：

```
begin
--程序块或单元体--
exception
when 异常名 1=>
--第一个处理程序--
when 异常名 2=>
--第二个处理程序--
--其他处理程序--
end;
```

3. 不是非常正确。异常处理程序可以获取异常的名字、简要描述以及异常被抛出的大概位置。

异常处理程序所在的程序块或单元内可出现的语句，也可以出现在处理程序内。

14.2.2 将异常绑定到处理程序

当抛出异常的程序块或单元包含该异常的处理程序时，异常就静态地绑定到处理程序上。如果抛出异常的程序块或单元不包含该异常的处理程序，异常就传播到其他某个程序块或单元中。异常传播的方式取决于产生异常的程序实体。

不管是在声明中还是在过程体的执行中，当异常在过程中抛出，且过程没有该异常的处理程序时，异常就隐式地传播到调用程序单元的调用点。这一策略是以下设计思想的体现：从子程序传播出去的异常应该沿控制路径(动态祖先)而不是静态祖先回溯。

如果异常传播到的调用单元也没有该异常的处理程序，它就再传播到该单元的调用者，如果必要，就一直继续下去，直到主过程，这是每个 Ada 程序的动态根。如果异常传播到主过程后，仍没有找到异常处理程序，程序就终止了。

在异常处理中，Ada 的程序块是没有参数的过程，当执行控制到达该块的第一条语句时，父块就“调用”这个块。当在块的声明或可执行语句处抛出异常，且块没有相应的异常处理程序时，异常就传播到下一个包含块的更大的静态作用域，也就是“调用”这个块的代码。异常传播到的地方就在异常发生所在块的结尾之后，也就是程序块的“返回”点。

当异常从包的包体中抛出，且包体中没有该异常的处理程序时，异常就传播到包含包声明的程序单元的说明部分。如果包恰好是一个库单元(独立编译的单元)，程序就终止了。

如果异常发生在任务体的最外层(不是在内嵌的程序块中)，且任务包含该异常的处理程序，就执行该处理程序，并将任务标记为已完成。如果任务没有该异常的处理程序，任务就简单地标记为已完成，异常并不传播。任务的控制机制太复杂了，对于未处理的异常应该传播到哪里的问题，无法给出合理且简单的答案。

在子程序、程序块、包和任务的声明部分也会抛出异常。在过程、包和程序块中抛出这类异常时，异常就像是在相关联的代码部分抛出一样向外传播。在任务中抛出这类异常时，任务标记为已完成，而没有进一步的动作，并且在任务的激活点抛出内建的异常 Tasking_Error。

14.2.3 继续

在 Ada 中，抛出异常的程序块或单元，以及异常传播到但没有处理异常的所有单元，都会终止。在异常处理后，程序控制不会隐式地返回到抛出异常的程序块或单元中，只是继续执行 exception 子句后的代码(exception 子句总是在程序块或单元的结尾处)。这就导致程序控制立即返回到更高一层。

程序单元中的异常处理程序执行完成后，需要决定执行在哪里继续时，Ada 的设计团队没有什么选择，因为 Ada 的需求说明(Department of Defense，1980a)清楚地表述了：抛出异常的程序单元不可继续或重新开始。不过对于程序块来说，在抛出异常并处理后，语句可以重试。例如，假设可以抛出异常的语句和该异常的处理程序都包含在程序块中，程

序块自身包含在一个循环中。以下示例的代码段从键盘获取四个在期望范围内的整数值，展示了这种结构：

```
...
type Age_Type is range 0..125;
type Age_List_Type is array (1..4) of Age_Type;
package Age_IO is new Integer_IO (Age_Type);
use Age_IO;
Age_List : Age_List_Type;
...
begin
for Age_Count in 1..4 loop
  loop -- loop for repetition when exceptions occur
 Except_Blk:
    begin -- compound to encapsulate exception handling
    Put_Line("Enter an integer in the range 0..125");
    Get(Age_List(Age_Count));
    exit;
    exception
      when Data_Error => -- Input string is not a number
    Put_Line("Illegal numeric value");
    Put_Line("Please try again");
      when Constraint_Error => -- Input is < 0 or > 125
    Put_Line("Input number is out of range");
    Put_Line("Please try again");
    end Except_Blk;
 end loop; -- end of the infinite loop to repeat input
           -- when there is an exception
end loop; -- end of for Age_Count in 1..4 loop
...
```

内部循环只包含程序块，程序控制保持在内部循环中，直到收到一个有效的输入数值。

14.2.4　其他设计选择

在默认的包 Standard 中，定义了四个异常：

```
Constraint_Error
Program_Error
Storage_Error
Tasking_Error
```

每一个异常实际上都是一类异常。例如，当数组下标超出范围时，当有范围限制的数值变量有范围错误时，当引用一个可辨识联合中不存在的记录域时，以及许多其他情况下，都会抛出 Constraint_Error 异常。

除了在 Standard 中定义的异常之外，其他预定义包定义了其他异常。例如，Ada.Text_IO 定义了 End_Error 异常。

用户自定义异常用以下声明形式来定义：

异常名列表：**exception**

除了必须显式抛出以外，这种异常与预定义异常完全一样。

预定义的异常有默认的异常处理程序，所有异常处理程序都会使程序终止。

异常用 raise 语句显式抛出，raise 语句的一般形式如下：

```
raise[异常名]
```

raise 语句可以不指定异常就出现的唯一地方是在异常处理程序内，这时它抛出引起处理程序执行的同一个异常，这就使异常根据前述的传播原则传播下去。若希望在抛出异常的地方打印错误消息，但在别的地方处理异常，就要在异常处理程序中使用 raise。

Ada 的 pragma 是给编译器的指令。某些运行时检查是内建异常的一部分，使用 Suppress pragma 可以在 Ada 程序中禁用这些运行时检查。Suppress pragma 的形式是

```
pragma Suppress(检查名)
```

其中检查名是特定异常检查的名字。本章后面将给出这种检查的例子。

Suppress pragma 只能出现在声明部分。当它出现时，在相关的程序块或声明部分所在的程序单元中，就会禁用特定的检查。显式的异常抛出不受 Suppress 的影响。虽然不需要，但大多数 Ada 编译器都实现了 Suppress pragma。

以下是可禁用检查的例子：Index_Check 和 Range_Check 表示在 Ada 程序中正常进行的两种检查，Index_Check 表示数组下标范围检查，Range_Check 表示检查赋给子类型变量的值的范围等。如果没有通过 Index_Check 或 Range_Check 检查，就会抛出 Constraint_Error。Division_Check 和 Overflow_Check 是与 Numeric_Error 相关联的可禁用检查。以下 pragma 禁用了数组下标范围检查：

```
pragma Suppress(Index_Check);
```

Suppress 有一个选项，允许进一步地将指定的检查限制到特定对象、类型、子类型和程序单元上。

14.2.5 例子

以下示例程序展示了 Ada 中异常处理程序的一些简单用法。程序用一个计数器数组计算并打印输入成绩的分布情况。输入是一系列成绩，以负数结束，这会抛出一个 Constraint_Error 异常，因为成绩是 Natural 类型(非负整数)。有 10 个类别的成绩(0～9、10～19、……、90～100)。成绩本身用于计算计数器数组的下标，每一个下标用于一类成绩。非法的输入成绩通过捕捉计数器数组中的下标错误来检测。成绩 100 在计算成绩分布时是特殊的，因为除了最高的一类成绩有 11 个值(90、91、……、100)以外，所有的成绩类别都只有 10 个值(成绩 A 比 B 或 C 更多，是老师慷慨大方的确实证据)。成绩 100 也在用于无效输入数据的异常处理程序中被处理。

```
-- Grade Distribution
```

```
-- Input: A list of integer values that represent
--        grades, followed by a negative number
-- Output: A distribution of grades, as a percentage for
--         each of the categories 0-9, 10-19, ...,
--         90-100.
with Ada.Text_IO, Ada.Integer.Text_IO;
use Ada.Text_IO, Ada.Integer.Text_IO;
procedure Grade_Distribution is
  Freq: array (1..10) of Integer := (others => 0);
  New_Grade : Natural;
  Index,
  Limit_1,
  Limit_2 : Integer;
begin
Grade_Loop:
  loop
  begin -- A block for the negative input exception
  Get(New_Grade);
  exception
    when Constraint_Error => -- for negative input
      exit Grade_Loop;
end; -- end of negative input block
 Index := New_Grade / 10 + 1;
   begin -- A block for the subscript range handler
   Freq(Index) := Freq(Index) + 1;
   exception
   -- For index range errors
     when Constraint_Error =>
       if New_Grade = 100 then
          Freq(10) := Freq(10) + 1;
        else
              Put("ERROR -- new grade: ");
              Put(New_Grade);
              Put(" is out of range");
              New_Line;
            end if;
     end; -- end of the subscript range block
   end loop;
-- Produce output
      Put("Limits Frequency");
      New_Line; New_Line;
      for Index in 0..9 loop
        Limit_1 := 10 * Index;
        Limit_2 := Limit_1 + 9;
        if Index = 9 then
          Limit_2 := 100;
        end if;
        Put(Limit_1);
        Put(Limit_2);
        Put(Freq(Index + 1));
```

```
            New_Line;
        end loop; -- for Index in 0..9 ...
    end Grade_Distribution;
```

请注意，处理无效输入成绩的代码在自己局部的程序块中，这使程序在处理了这类异常后继续运行，就像前面从键盘读入数值的例子一样。负输入的异常处理程序也在自己的程序块中，其原因是在负输入抛出异常时，限制 Constraint Error 处理程序的作用域。

14.2.6 评价

与其他一些语言结构一样，Ada 的异常处理设计体现了多数人的意见，至少也体现了设计它的那个时期(20 世纪 70 年代晚期到 80 年代早期)关于这方面的意见。在一段时期内，Ada 都是广泛使用的唯一包含异常处理的语言。

Ada 的异常处理有一些问题。一个问题是传播模型，它允许异常传播到外部作用域中，在那里异常是不可见的。而且，也不是总能确定所传播的异常的来源。

另一个问题是对于任务来说异常处理的不合理性。例如，抛出异常但没有处理它的任务会停止执行。

最后，当 Ada 95 加入了面向对象编程的支持后，它的异常处理没有被扩展来处理新的语言结构。例如，在程序块中创建并使用类的多个对象，且其中一个对象传播了异常时，无法确定是哪一个对象抛出了异常。

Ada 的异常处理问题在 Romanovsky 和 Sandén(2001)中有所介绍。

14.3 C++中的异常处理

C++的异常处理在 1990 年得到 ANSI C++标准委员会的承认，后来 C++的多种实现都采纳了它。其设计部分基于 CLU、Ada 和 ML 的异常处理。C++与 Ada 的异常处理之间的一个主要区别是，在 C++中没有预定义的异常(除了标准库中的异常以外)。因此在 C++中，异常由用户或库定义，并显式抛出。

14.3.1 异常处理程序

14.2 节提到，Ada 使用程序单元和块来定义异常处理程序的作用域。C++使用一种以保留字 try 开头的专门结构来定义。try 结构包含一个称为 try 子句的复合语句和一系列异常处理程序。复合语句定义了后面异常处理程序的作用域。这一结构的一般形式是

```
try {
//** Code that might raise an exception
}
catch(形参) {
//** A handler body
}
...
```

```
catch(形参) {
//** A handler body
}
```

每个 catch 函数就是一个异常处理程序。catch 函数只能有一个形参，与 C++函数定义中的形参相似，形参也可以是一个省略号(…)。带有省略号形参的异常处理程序，是捕获所有异常的处理程序。对于任何异常，如果没有找到合适的异常处理程序，就使用捕获所有异常的处理程序。形参也可以是不加修饰的类型指定符，例如 float，与函数原型一样。在这种情况下，形参的唯一用途是唯一标识该异常处理程序。异常的信息传递给异常处理程序时，形参包含了一个用于传递异常信息的变量名。由于参数的类可以是任意的用户自定义类，所以参数可以包含任意所需数目的数据成员。14.3.2 节介绍异常与异常处理程序的绑定。

在 C++中，异常处理可以包含任意的 C++代码。

14.3.2　异常与处理程序的绑定

C++的异常只能由语句 throw 显式抛出，throw 的一般形式用 EBNF 表示为

```
throw [表达式];
```

其中的方括号是元符号，表示表达式是可选的。没有操作数的 throw 只能出现在异常处理程序中。当它出现时，会再次抛出异常，使异常在其他地方处理。这种作用与 Ada 是完全一样的。

throw 表达式的类型选择了特定的异常处理程序，这个异常处理程序当然必须有“匹配”类型的形参。在这里，“匹配”的含义是：形参为类型 T、const T、T&(对类型 T 的对象引用)或 const T&的异常处理程序与表达式类型为 T 的 throw 相匹配。当 T 是类时，参数为类型 T 或任何 T 的祖先的异常处理程序可以匹配。throw 表达式与形参相匹配还有更复杂的情况，但这里不介绍。

在 try 子句中抛出的异常会使 try 子句中的代码立即结束执行，从紧跟 try 子句的异常处理程序开始，搜索相匹配的处理程序。搜索匹配的过程对多个异常处理程序依次进行，直到找到匹配，即如果在完全匹配的异常处理程序之前有任何其他匹配的处理程序，就不会用到完全匹配的异常处理程序。因此，用于特定异常的处理程序放在异常处理程序列表的最上面，后跟更通用的异常处理程序。最后一个异常处理程序通常以省略号(…)为形参，它匹配任意的异常，这可保证会捕获所有的异常。

如果在 try 子句中抛出异常，并且没有匹配的异常处理程序与这个 try 子句相关联，异常就会传播。如果 try 子句内嵌在另一个 try 子句中，异常就会传播到与外层 try 子句相关联的异常处理程序。如果外层的 try 子句都没有相匹配的异常处理程序，异常就传播到抛出异常的函数的调用者。如果对函数的调用不在 try 子句中，异常就传播到函数的调用者。如果在这个传播过程中没有在程序中找到相匹配的异常处理程序，就会调用默认的异常处理程序。14.3.4 节会深入介绍这个异常处理程序。

14.3.3 继续

当异常处理程序完成执行后，程序控制到达 try 结构后面的第一条语句处(紧跟在一系列异常处理程序中的最后一个处理程序后面的语句)。异常处理程序可以使用没有表达式的 throw，再次抛出一个异常，这就传播了该异常。

14.3.4 其他设计选择

根据 14.1.2 节总结的设计问题，C++的异常处理很简单，只有用户自定义的异常，而且没有专门的说明(虽然可以声明为新的类)。有一个默认异常处理程序 unexpected，它的唯一动作是终止程序。这个异常处理程序捕获所有未被程序捕获的异常，它可以用一个用户自定义的异常处理程序来替换。替换的异常处理程序必须是一个返回 void 的无参函数。这个替换的函数通过将名字赋给 set_terminate 来设置。异常不能禁用。

C++函数可以列出可能抛出的异常类型(throw 表达式的类型)，为此，需要在函数头添加保留字 throw，后跟用圆括号括起来的类型列表。例如，

```
int fun() throw (int, char *) { ... }
```

表明函数 fun 可能抛出的异常类型为 int 和 char*，不可能是其他类型。throw 子句的目的是向函数的使用者表明，函数可能会抛出哪些异常。throw 子句实际上是函数与其调用者之间的协议，它保证函数不会抛出其他异常。如果函数抛出了某个未列出的异常，程序就会终止。请注意，编译器忽略 throw 子句。

如果 throw 子句中的类型是类，函数就可以抛出任何从所列类派生的异常。如果函数头有 throw 子句，且抛出的异常没有列在 throw 子句中，也不是从所列的类中派生，就会调用默认的异常处理程序。请注意，这种错误无法在编译时检测到。类型列表可以为空，表示函数不会抛出任何异常。如果函数头没有 throw 说明，函数就可能抛出任何异常。这个列表不是函数类型的一部分。

如果一个函数重定义了一个有 throw 子句的函数，那么重定义函数中的 throw 子句不能拥有比被重定义的函数更多的异常。

虽然 C++没有预定义的异常，但标准库定义并抛出异常，例如 out_of_range 可以由库的容器类抛出，overflow_error 可以由数学库函数抛出。

14.3.5 例子

以下例子与 14.2.5 节中的 Ada 程序有相同的目的，都使用了异常处理。它为 10 个类别使用了计数器数组，生成输入成绩的分布，通过检查用于递增所选计数器的非法下标，来检测非法的成绩。

```
// Grade Distribution
// Input: A list of integer values that represent
//        grades, followed by a negative number
// Output: A distribution of grades, as a percentage for
```

```
//        each of the categories 0-9, 10-19, ...,
//        90-100.
#include <iostream>
int main() { //* Any exception can be raised
  int new_grade,
      index,
      limit_1,
      limit_2,
      freq[10] = {0,0,0,0,0,0,0,0,0,0};
// The exception definition to deal with the end of data
class NegativeInputException {
  public:
   NegativeInputException() { //* Constructor
    cout << "End of input data reached" << endl;
  } //** end of constructor
} //** end of NegativeInputException class
 try {
   while (true) {
     cout << "Please input a grade" << endl;
     if ((cin >> new_grade) < 0) //* End of data
       throw NegativeInputException();
     index = new_grade / 10;
     {try {
       if (index > 9)
         throw new_grade;
       freq[index]++;
       } //* end of inner try compound
     catch(int grade) { //* Handler for index errors
       if (grade == 100)
         freq[9]++;
       else
         cout << "Error -- new grade: " << grade
              << " is out of range" << endl;
       } //* end of catch(int grade)
     } //* end of the block for the inner try-catch
          pair
    } //* end of while (1)
   } //* end of outer try block
   catch(NegativeInputException& e) { //**Handler for
                                      //** negative input
     cout << "Limits Frequency" << endl;
     for (index = 0; index < 10; index++) {
       limit_1 = 10 * index;
       limit_2 = limit_1 + 9;
       if (index == 9)
         limit_2 = 100;
       cout << limit_1 << limit_2 << freq[index] << endl;
      } //* end of for (index = 0)
    } //* end of catch (NegativeInputException& e)
  } //* end of main
```

这个程序意在展示 C++的异常处理机制。请注意，在 C++中，下标范围异常通常是通过重载下标运算(可以抛出异常)来处理的，而不是像本例这样用选择结构来直接检测下标运算。

14.3.6 评价

在某些方面，C++的异常处理机制类似于 Ada，例如函数中未处理的异常传播到函数的调用者。但是在另一些方面，C++的设计非常不一样：没有可由用户处理的、硬件可检测的预定义异常，且异常没有名字。通过一个可省略形参的参数类型，将异常连接到异常处理程序上。异常处理程序的形参类型决定了调用它的条件，但与所抛出的异常内容没有任何关系，因此为异常使用预定义类型肯定不会提升可读性。在用于定义异常的有意义的类层次中，用有意义的名字来为异常定义类要好得多。异常参数为向异常处理程序传递异常的信息提供了一种方法。

14.4 Java 中的异常处理

在第 13 章中，Java 的示例程序包含了异常处理的使用，但没有多少解释。本节介绍 Java 异常处理功能的细节。

Java 的异常处理基于 C++的异常处理，但其设计与面向对象的语言规范更一致。此外，Java 包含了一系列预定义的异常，可由 Java 虚拟机(Java Virtual Machine，JVM)隐式抛出。

14.4.1 异常类

所有的 Java 异常都是从 Throwable 类派生的类的对象。Java 系统包含两个预定义的异常类 Error 和 Exception，它们是 Throwable 的子类。Error 类和它的派生类与 Java 运行库系统抛出的错误有关，例如堆内存不足。这些异常不是由用户程序抛出的，不应该由用户程序来处理。有两个系统定义的 Exception 的直接派生类：RuntimeException 和 IOException。顾名思义，IOException 是在输入或输出操作发生错误时抛出的，这些输入输出操作都在包 java.io 中定义为各种类的方法。

Java 中有派生自 RuntimeException 的预定义类。在大多数情况下，当用户程序产生错误时，会(由 JVM[4])抛出 RuntimeException。例如，定义在 java.util 中的 ArrayIndexOutOfBoundsException 是从 RuntimeException 派生的、经常抛出的异常。另一个从 RuntimeException 派生的、经常抛出的异常是 NullPointer Exception。

用户程序可以定义自己的异常类，Java 中的约定是，用户自定义异常都是 Exception 的子类。

4. Java 规范也要求 JIT 编译器检测这些异常，并在异常发生时抛出 RuntimeException。

14.4.2　异常处理程序

除了每个 catch 必须有一个参数，且参数的类必须是预定义类 Throwable 的派生类之外，Java 的异常处理程序形式与 C++相同。

除了 14.4.6 节介绍的 finally 子句外，Java 中 try 结构的语法与 C++完全相同。

14.4.3　异常与处理程序的绑定

抛出异常非常简单，只要将异常类的一个实例作为 throw 语句的操作数。例如，假设定义一个异常 MyException，如下：

```
class MyException extends Exception {
  public MyException() {}
  public MyException(String message) {
    super (message);
  }
}
```

这个异常可以这样抛出：

```
throw new MyException();
```

创建用于 throw 的异常实例，可以与 throw 语句分开，如：

```
MyException myExceptionObject = new MyException();
...
throw myExceptionObject;
```

在新类中，两个构造函数之一是没有参数的，另一个有一个 String 对象的参数，传递给超类(Exception)，超类会显示该 String 对象。因此新异常可以这样抛出：

```
throw new MyException
      ("a message to specify the location of the error");
```

Java 中异常与处理程序的绑定与 C++相似。如果在 try 结构的复合语句中抛出异常，它就绑定到 try 子句后面的第一个异常处理程序(catch 函数)，其参数的类型与所抛出对象的类相同，或是其祖先类。如果找到相匹配的异常处理程序，throw 就绑定到这个处理程序并执行它。

在异常处理程序的结尾处，包含一个没有操作数的 throw 语句，就可以在处理异常后再次抛出异常。新抛出的异常不会在最初抛出异常的同一个 try 中处理，所以不必担心出现循环。通常当某些局部动作很有用，但又需要包含它的 try 子句或调用者中的 try 子句进行进一步处理时，就会再次抛出异常。异常处理程序中 throw 语句抛出的异常也可以不同于将程序控制移交到该处理程序的异常。

为了确保 try 子句中抛出的异常总是在一个方法中处理，只需要用 Exception 类型参数来定义处理程序，就可以捕获所有从 Exception 派生的异常，如：

```
catch (Exception genericObject) {
  ...
}
```

由于类名总是与它自己或祖先类相匹配，所以任何从 Exception 派生的类都匹配 Exception。当然，这样的异常处理程序应该总是放在异常处理程序列表的最后，因为它会阻碍 try 结构中任何跟在它后面的异常处理程序(因为搜索匹配的异常处理程序是按顺序进行的，当找到一个匹配时，就停止搜索)。

14.4.4 其他设计选择

在程序执行时，Java 的运行时系统存储了程序中每一个对象的类名。方法 getClass 可用于获取一个存储了类名的对象，类名本身可以用 getName 方法得到。因此可以从 throw 语句(它会启动处理程序的执行)中获得实参的类名。对于前面所示的异常处理程序，可以如下面这样获取类名：

```
genericObject.getClass().getName()
```

此外，构造函数创建了参数对象，可以如下面这样得到与参数对象相关的消息：

```
genericObject.getMessage()
```

而且，对于用户自定义异常，被抛出的对象可以包含对异常处理程序有用的任意数目的数据域。

Java 中 throws 子句的外观与(在程序中的)位置，与 C++的 throw 子句相似。不过 throws 的语义与 C++的 throw 子句有一些不同。

Java 方法的 throws 子句中出现的异常类名，说明该方法会抛出该异常类或从它派生的异常类，但不会处理。例如，方法说明它会抛出 IOException，意味着它会抛出一个 IOException 对象，或其派生类的一个对象(例如 EOFException)，但不处理所抛出的异常。

类 Error 和 RuntimeException 及其派生类的异常，称为未检查的异常，所有其他异常称为要检查的异常。编译器不会处理未检查的异常，但是编译器要确保方法可抛出的所有要检查的异常，都在方法的 throws 子句中列出，或者在方法中处理。请注意，在编译时对此进行检查，这与 C++不同，在 C++中是在运行时检查的。不检查类 Error 和 RuntimeException 及其派生类的异常，是因为任何方法都可以抛出它们。程序可以捕获未检查的异常，但不一定要这样做。

与 C++一样，重定义方法在 throws 子句中声明的异常不能多于、但可以少于被重定义方法所声明的异常。因此，如果方法没有 throws 子句，那么重定义它的方法也没有 throws 子句。方法可以抛出任何在 throws 子句中列出的异常，以及任何其派生类的异常。

如果方法不直接抛出某个异常，但调用另一个可以抛出该异常的方法，就必须在它的 throws 子句中列出该异常。所以使用 readLine 方法的 buildDist 方法(在下一节的示例程序中)，必须在方法头的 throws 子句中说明 IOException。

不包含 throws 子句的方法不能传播任何要检查的异常。在 C++中，没有 throw 子句的函数可以抛出任意异常。

若第一个方法调用了第二个方法，第二个方法在 throws 子句中列出某个要检查的异常，则该异常有三种处理方式：第一种，捕获该异常并处理它；第二种，捕获该异常，并抛出自己的 throws 子句中列出的一个异常；第三种，在自己的 throws 子句中声明该异常，但不处理它，这可以有效地将异常传播到外层的 try 子句中(如果有的话)，或者传播到该方法的调用者(如果没有外层的 try 子句)。

没有默认的异常处理程序，也不能禁用异常。Java 中的异常继续与 C++完全一样。

14.4.5　例子

以下的 Java 程序与 14.3.5 节中的 C++程序功能一样。

```
// Grade Distribution
//  Input: A list of integer values that represent
//         grades, followed by a negative number
// Output: A distribution of grades, as a percentage for
//         each of the categories 0-9, 10-19, ...,
//         90-100.
import java.io.*;
// The exception definition to deal with the end of data
class NegativeInputException extends Exception {
  public NegativeInputException() {
    System.out.println("End of input data reached");
  } //** end of constructor
} //** end of NegativeInputException class

class GradeDist {
  int newGrade,
      index,
       limit_1,
       limit_2;
  int [] freq = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

void buildDist() throws IOException {
  DataInputStream in = new DataInputStream(System.in);
  try {
    while (true) {
      System.out.println("Please input a grade");
      newGrade = Integer.parseInt(in.readLine());
      if (newGrade < 0)
    throw new NegativeInputException();
  index = newGrade / 10;
  try {
   freq[index]++;
  } //** end of inner try clause
  catch(ArrayIndexOutOfBoundsException e) {
    if (newGrade == 100)
```

```
        freq [9]++;
      else
        System.out.println("Error - new grade: " +
                  newGrade + " is out of range");
    } //** end of catch (ArrayIndex...
  } //** end of while (true) ...
} //** end of outer try clause
catch(NegativeInputException e) {
  System.out.println ("\nLimits Frequency\n");
  for (index = 0; index < 10; index++) {
    limit_1 = 10 * index;
    limit_2 = limit_1 + 9;
    if (index == 9)
      limit_2 = 100;
    System.out.println("" + limit_1 + " - " +
                    limit_2 + " " + freq [index]);
  } //** end of for (index = 0; ...
} //** end of catch (NegativeInputException ...
} //** end of method buildDist
```

负输入异常 NegativeInputException 在程序中定义，它的构造函数在创建类的对象时显示一条消息，它的处理程序生成了方法的输出。ArrayIndexOutOfBoundsException 是一个预定义的未检查异常，由 Java 运行时库系统抛出。对于以上两种异常，处理程序都没有在参数中包含对象名字，在这两种情况下名字都没有用。虽然所有的异常处理程序都以对象作为参数，但经常没有用。

14.4.6 finally 子句

在某些情况下，不管 try 子句是否抛出异常，也不管所抛出的异常是否在方法中捕获，都必须执行一个过程。这种情况的一个例子就是关闭文件，另一个例子是，不管方法的执行是如何终止的，都必须释放方法中的某种外部资源。finally 子句就是为这种需要设计的。finally 子句放在完整 try 结构后一系列异常处理程序的结尾处。通常，try 结构及其 finally 子句的形式如下：

```
try {
...
}
catch (...) {
...
}
... //** More handlers
finally {
...
}
```

这个结构的语义如下：如果 try 子句不抛出异常，就执行 finally 子句，然后继续执行 try 结构后面的语句；如果 try 子句抛出异常，异常被后面的异常处理程序所捕获，finally

子句就在异常处理程序执行完后执行；如果 try 子句抛出异常，但是异常没有被 try 结构后面的异常处理程序所捕获，finally 子句就在异常传播之前执行。

没有异常处理程序的 try 结构可以跟一个 finally 子句。显然，只有复合语句有 throw、break、continue 或 return 语句时，这才是有意义的。在这些情况下，它的用途与有异常处理程序时一样。例如以下程序：

```
try {
  for (index = 0; index < 100; index++) {
    ...
    if (... ) {
      return;
    } //** end of if
    ...
 } //** end of for
} //** end of try clause
finally {
  ...
} //** end of try construct
```

不管循环是由 return 终止的还是正常退出的，这里的 finally 子句都会执行。

14.4.7　断言

在第 2 章对 Plankalkül 的介绍中，提到它包含断言。Java 的 1.4 版加入了断言。为了使用断言，需要在运行程序时以 enableassertions(或 ea)标志来启用断言，如

```
java -enableassertions MyProgram
```

assert 语句的两种可能形式是：

```
assert 条件;
assert 条件：表达式;
```

在第一种情况下，当执行到达 assert 时，就测试该条件。如果条件为真，就什么也不做，如果条件为假，就抛出 AssertionError 异常。在第二种情况下，行为是一样的，只不过表达式的值会作为一个串传递给 AssertionError 的构造函数，成为调试输出。

assert 语句用于防御性编程。程序可以有许多 assert 语句，这会确保跟踪程序的计算，得到正确的结果。许多程序员在写程序时，即使所使用的语言不支持断言，也会进行这样的检查来帮助调试。充分地测试程序后，就可以去掉这些检查。assert 语句有同样的作用，其优点是可以禁用而不必将它们从程序中去掉。这就节省了去掉它们的工作量，在以后的程序维护中也可以使用它们。

14.4.8　评价

Java 的异常处理机制是所基于的 C++异常处理机制的改善。

第一，C++程序可以抛出程序中定义的或由系统定义的任何异常。在 Java 中，只有

Throwable 或从它派生的类的实例对象，才可以抛出。这就将可抛出的对象与程序中所有其他对象(和非对象)区别开。抛出 int 值的异常能有什么意义呢？

第二，不包含 throw 子句的 C++程序单元，可以抛出任何异常，这没有告诉读者任何信息。不包含 throws 子句的 Java 方法不能抛出任何它不处理的、要检查的异常。因此，Java 方法的读者根据方法头就知道它会抛出但不处理哪些异常。C++编译器忽略 throw 子句，但 Java 编译器确保方法可抛出的所有异常都在 throws 子句中列出。

第三，增加的 finally 子句在某些情况下特别方便。它允许执行清理操作，而不管复合语句是如何终止的。

最后，Java 运行库系统隐式地抛出多种预定义异常，例如数组下标超出范围和对空引用的解引用，这些异常可以由任何用户程序来处理。C++程序只能处理显式抛出的异常(或所使用的库类抛出的异常)。

Java 和 Ada 的异常处理功能大致相当。Java 方法中有 throws 子句，有助于提高可读性，而 Ada 没有相应的特性。在允许程序处理系统可检测的异常方面，显然 Java 与 Ada 的关系比与 C++的关系更接近。

C#包含非常类似于 Java 的异常处理结构，但 C#没有 throws 子句。

14.5 事件处理概述

事件处理与异常处理类似，对于两者来说，异常或事件发生时，就会隐式地调用处理程序。异常可以由用户代码显式创建，或者由硬件或软件解释器隐式创建，而事件是由外部动作创建的，例如通过图形用户界面(GUI)的用户交互动作来创建。本节介绍事件处理的基础知识，这远没有异常处理那么复杂。

在传统的(非事件驱动的)程序设计中，代码本身说明了代码执行的顺序，虽然该顺序通常会受到程序输入数据的影响。在事件驱动的程序设计中，程序某些部分是在完全不可预测的时刻执行的，通常由用户与执行程序的交互动作来触发。

本章介绍的特定类型的事件处理与 GUI 相关。因此，大多数事件都是通过图形对象或组件(称为窗口小部件)由用户的交互动作引起的。最常见的窗口小部件是按钮。实现对用户与 GUI 组件的交互动作的响应，是最常见的事件处理形式。

事件就是发生某件事情的通知，例如鼠标在图形按钮上点击。严格来说，至少在这里讨论的事件处理中，事件是一个对象，由运行时系统隐式创建，来响应用户动作。

事件处理程序是一段代码，为响应事件的发生而执行。事件处理程序使程序可以响应用户动作。

虽然事件驱动的程序设计在 GUI 出现之前已经用了很久，但只有在 GUI 广泛应用时，它才成为一种广泛应用的程序设计方法学。例如呈现给网页浏览器用户的 GUI。现在许多呈现给浏览器用户的网页文档都是动态的，这样的文档可能向用户展示一个订单，让用户点击按钮来选择商品。所需要的与点击按钮相关联的内部计算，就由响应该点击事件的事

件处理程序来执行。

事件处理的另一个常见应用是在修改表单时，或表单提交到网页服务器准备进行处理时，检查表单元素中的简单错误和遗漏。在浏览器上使用事件处理来检查表单数据的有效性，可节省将数据发送到服务器的时间，在服务器上处理表单数据之前，必须由驻存在服务器的程序或脚本来检查它们的正确性。这类事件驱动的程序设计通常用客户端的脚本语言(如 JavaScript)来实现。

14.6　Java 的事件处理

除了 Web 应用程序之外，非 Web Java 应用程序也可以给用户显示 GUI。本节讨论 Java 应用程序中的 GUI。

Java 的最初版本提供了一种有点原始的方式支持 GUI 组件。在 1988 年年底发布的 Java 1.2 版中，增加了一组新的组件，统称为 Swing。

14.6.1　Java Swing 的 GUI 组件

类和接口的 Swing 集在 javax.swing 中定义，包含一组 GUI 组件。由于这里关注的是事件处理，而不是 GUI 组件，所以只介绍两种窗口小部件：文本框和单选按钮。

文本框是类 JTextField 的对象，最简单的 JTextField 构造函数有一个参数，是文本框的字符长度，例如：

```
JTextField name =new JTextField(32);
```

JTextField 的构造函数也可以把一个文字串作为可选的第一个参数。有这个字符串参数时，它就显示为文本框的初始内容。

单选按钮是一种放在按钮组容器中的特殊按钮。按钮组是类 ButtonGroup 的对象，该类的构造函数没有参数。在单选按钮组中，一次只能按下一个按钮。如果按下了组中的一个按钮，先前按下的按钮就隐式地放开。用于创建单选按钮的 JRadioButton 构造函数，有两个参数：标签和单选按钮的初始状态(true 或 false，分别表示按下和未按下)。如果组中的一个单选按钮最初设置为按下，组中的其他按钮就默认为未按下。在创建了单选按钮后，用按钮组对象的 add 方法将它们放入按钮组中。考虑以下例子：

```
ButtonGroup payment = new ButtonGroup();
JRadioButton box1 = new JRadioButton("Visa", true);
JRadioButton box2 = new JRadioButton("Master Charge");
JRadioButton box3 = new JRadioButton("Discover");
payment.add(box1);
payment.add(box2);
payment.add(box3);
```

JFrame 对象是一个框架，显示为一个独立的窗口。JFrame 类定义了框架所需的数据和

方法，所以使用框架的类可以是 JFrame 的子类。JFrame 有几层，称为面板。这里只对其中一层感兴趣，即内容面板。GUI 的组件放在一个 JPanel 对象中(面板)，它用于组织和定义组件的布局。首先创建框架，再在框架的内容面板中添加包含组件的面板。

预定义的图形对象，例如 GUI 组件，可以直接放在面板中。下面创建的面板对象将用于后面对组件的讨论：

```
JPanel myPanel = new JPanel();
```

用构造函数创建组件之后，必须用 add 方法将组件放入面板，如：

```
myPanel.add(button1);
```

14.6.2 Java 事件模型

用户与 GUI 组件交互时，例如单击按钮，组件就会创建一个事件对象，并通过名为事件监听器的对象传递事件对象，来调用事件处理程序。事件处理程序提供了相关的操作。GUI 组件是事件生成器，它们都会生成事件。在 Java 中，事件通过事件监听器与事件处理程序相连接。监听器注册是通过实现了监听器接口的类的一个方法实现的，本节后面会介绍。当事件发生时，只通知已为特定事件注册过的事件监听器。

接收到消息的监听器方法实现了事件处理程序。为了使事件处理方法符合标准的协议，使用了接口。接口描述了标准的方法协议，但不提供这些方法的实现。

需要实现事件处理程序的类，必须为该处理程序实现监听器的接口。有许多事件类和监听器接口。ItemEvent 就是一个事件类，它与选择复选框、单选按钮或列表项的事件相关联。ItemListener 接口包含一个方法 itemStateChanged 的协议，该方法是 ItemEvent 事件的处理程序。因此，为了实现由单选按钮动作触发的行为，必须实现接口 ItemListener，该接口需要定义方法 itemStateChanged。

如前所述，实现了监听器接口的类的一个方法将组件与事件监听器连接起来。例如，由于 ItemEvent 是由单选按钮的用户动作所创建的事件对象的类名，所以必须使用 addItemListener 方法来注册单选按钮的监听器。这个在面板中生成的按钮事件的监听器，可以在面板或 JPanel 的子类中实现。因此，对于 myPanel 面板中的 button1 单选按钮，myPanel 实现了 button1 的 ItemEvent 事件处理程序，可以用以下语句来注册该监听器：

```
button1.addItemListener(this);
```

每一个事件处理方法都接收一个事件参数，该参数给出事件的信息。事件类有一些方法来访问这些信息。例如，通过单选按钮来调用 isSelected 方法，该方法就根据按钮是打开还是关闭(按下还是未按下)，分别返回真或假。

所有的事件相关类都在 java.awt.event 包中，因此在使用事件的任何类中通常会导入这个包。

以下是一个示例应用程序 RadioB，显示了事件和事件处理的用法。这个应用程序构建了一些单选按钮，控制一个文本框内容的字体。它为四种字体各创建了一个 Font 对象，每

个对象都有一个单选按钮，让用户选择该字体。

这个例子的目标是显示如何处理 GUI 组件触发的事件，来修改程序动态显示的结果。由于这个只关注事件处理，因此这里不会解释这个程序的某些部分。

```
/* RadioB.java
   An example to illustrate event handling with interactive
   radio buttons that control the font style of a textfield
  */
package radiob;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RadioB extends JPanel implements
        ItemListener {
    private JTextField text;
    private Font plainFont, boldFont, italicFont,
                 boldItalicFont;
    private JRadioButton plain, bold, italic, boldItalic;
    private ButtonGroup radioButtons;

// The constructor method is where the display is initially
// built
   public RadioB() {

// Create the test text string and set its font
        text = new JTextField(
              "In what font style should I appear? ", 25);
                text.setFont(plainFont);

// Create radio buttons for the fonts and add them to
// a new button group
       plain = new JRadioButton("Plain", true);
       bold = new JRadioButton("Bold");
       italic = new JRadioButton("Italic");
       boldItalic = new JRadioButton("Bold Italic");
       radioButtons = new ButtonGroup();
       radioButtons.add(plain);
       radioButtons.add(bold);
       radioButtons.add(italic);
       radioButtons.add(boldItalic);

       // Create a panel and put the text and the radio
       // buttons in it; then add the panel to the frame
       JPanel radioPanel = new JPanel();
       radioPanel.add(text);
       radioPanel.add(plain);
       radioPanel.add(bold);
       radioPanel.add(italic);
       radioPanel.add(boldItalic);
       add(radioPanel, BorderLayout.LINE_START);
```

```
// Register the event handlers
        plain.addItemListener(this);
        bold.addItemListener(this);
        italic.addItemListener(this);
        boldItalic.addItemListener(this);

// Create the fonts
       plainFont = new Font("Serif", Font.PLAIN, 16);
       boldFont = new Font("Serif", Font.BOLD, 16);
       italicFont = new Font("Serif", Font.ITALIC, 16);
       boldItalicFont = new Font("Serif", Font.BOLD +
                                    Font.ITALIC, 16);
    } // End of the constructor for RadioB

// The event handler
    public void itemStateChanged (ItemEvent e) {

// Determine which button is on and set the font
// accordingly
       if (plain.isSelected())
           text.setFont(plainFont);
       else if (bold.isSelected())
          text.setFont(boldFont);
       else if (italic.isSelected())
          text.setFont(italicFont);
       else if (boldItalic.isSelected())
          text.setFont(boldItalicFont);
   } // End of itemStateChanged

// The main method
    public static void main(String[] args) {
// Create the window frame
        JFra me myFrame = new JFrame(" Radio button
                                       example");

// Create the content pane and set it to the frame
       JComponent myContentPane = new RadioB();
       myContentPane.setOpaque(true);
       myFrame.setContentPane(myContentPane);

// Display the window.
        myFrame.pack();
        myFrame.setVisible(true);
    }
} // End of RadioB
```

RadioB.java 应用程序生成的屏幕如图 14-2 所示。

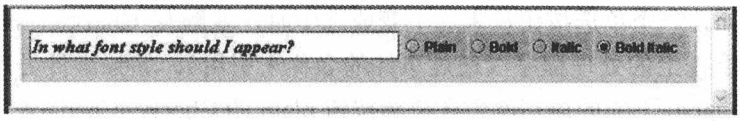


图 14-2　RadioB.java 的输出

14.7　C#中的事件处理

C#(和其他.NET 语言)中的事件处理类似于 Java。.NET 提供了在应用程序中创建 GUI 的两种方式：原来的 Windows Forms 和最近的 Windows Presentation Foundation，后者比较高级、复杂。因为这里只讨论事件处理，所以使用较简单的 Windows Forms 来讨论。

C#应用程序使用 Windows Forms 构建 GUI 时，会创建预定义类 Form 的子类，Form 在 System.Windows.Forms 名称空间中定义。这个类隐式提供了一个窗口来包含组件，而不必显式建立框架或面板。

文本放在 Label 对象中，单选按钮是 RadioButton 类的对象。Label 对象的大小不在构造函数中显式指定，而可以通过将 Label 对象的 AutoSize 数据成员设置为 true 来指定，这是根据它包含的内容来设置 Label 对象的大小。

把一个新的 Point 对象赋予组件的 Location 属性，就可以将该组件放在窗口中特定的位置上。Point 类在 System.Drawing 名称空间中定义。Point 构造函数带两个参数，它们是对象的像素坐标。例如，Point(100, 200)距离窗口左边界 100 像素，距离顶边界 200 像素。将一个字面量字符串赋予组件的 Text 属性，就可以设置组件的标签。创建组件后，就把它发送给表单中 Controls 子类的 Add 方法，将它添加到表单窗口上。因此，下面的代码创建了一个单选按钮，其标签是 Plain，位于输出窗口的位置(100, 300)上：

```
private RadioButton plain = new RadioButton();
plain.Location = new Point(100, 300);
plain.Text = "Plain";
Controls.Add(plain);
```

所有 C#事件处理程序都有相同的协议：返回类型是 void，两个参数的类型是 object 和 EventArgs。这两个参数都不需要用于简单的情形。事件处理程序方法可以有任意名称。使用单选按钮的布尔属性 Checked 来测试该按钮，可以确定单击了哪个按钮。下面是事件处理程序的框架示例：

```
private void rb_CheckedChanged (object o, EventArgs e){
  if (plain.Checked) ...
  ...
}
```

要注册一个事件，必须创建一个新的 EventHandler 对象。为此，要给这个类的构造函数发送处理方法的名称。在组件对象的事件上，使用+=赋值操作符把新对象添加到预定义

的委托上。例如，一个单选按钮从未选中变成选中时，就触发 CheckedChanged 事件，调用注册到相关委托上的处理程序（通过事件的名称来引用）。如果事件处理程序的名称是 rb_CheckedChanged，下面的语句就会在单选按钮 plain 的 CheckedChanged 事件上注册处理程序：

```
CheckedChanged +=
      new EventHandler(rb_CheckedChanged);
```

下面用 C#重写 14.6 节的 RadioB 示例，因为这里关注事件的处理，所以不解释该程序的细节：

```
// RadioB.cs
// An example to illustrate event handling with
// interactive radio buttons that control the font
// style of a string of text


namespace RadioB {

   using System;
   using System.Drawing;
   using System.Windows.Forms;

public class RadioB : Form {
  private Label text = new Label();
  private RadioButton plain = new RadioButton();
  private RadioButton bold = new RadioButton();
  private RadioButton italic = new RadioButton();
  private RadioButton boldItalic = new RadioButton();

 // Constructor for RadioB
public RadioB() {

  // Init ialize the attributes of the text and radio
  // buttons
  text.AutoSize = true;
  text.Text = "In what font style should I appear? ";
  plain.Location = new Point(220,0);
  plain.Text = "Plain";
  plain.Checked = true;
  bold.Location = new Point(350, 0);
  bold.Text = "Bold";
  italic.Location = new Point(480, 0);
  italic.Text = "Italics";
  boldItalic.Location = new Point(610, 0);
  boldItalic.Text = "Bold/Italics";

  // Add the text and the radio buttons to the form
  Controls.Add(text);
```

```
  Controls.Add(plain);
  Controls.Add(bold);
  Controls.Add(italic);
  Controls.Add(boldItalic);

  // Register the event handler for the radio buttons
  plain .CheckedChanged +=
       new EventHandler(rb_CheckedChanged);
  bold. CheckedChanged +=
       new EventHandler(rb_CheckedChanged);
  itali c.CheckedChanged +=
       new EventHandler(rb_CheckedChanged);
  boldI talic.CheckedChanged +=
       new EventHandler(rb_CheckedChanged);
 }
 // The main method is where execution begins
static void Main() {
  Application.EnableVisualStyles();
  Appl ication.SetCompatibleTextRenderingDefault
       (false);
  Application.Run(new RadioB());
 }

 // The event handler
private void rb_CheckedChanged ( object o,
                                 EventArgs e) {
 // Determine which button is on and set the font
 // accordingly
  if (plain.Checked)
      text.Font =
           new Font( text.Font.Name, text.Font.Size,
                     FontStyle.Regular);
  if (bold.Checked)
      text.Font =
           new Font(text.Font.Name, text.Font.Size,
                    FontStyle.Bold);
  if (italic.Checked)
       text.Font =
           new Font(text.Font.Name, text.Font.Size,
                     FontStyle.Italic);
  if (boldItalic.Checked)
      text.Font =
            new Font(text.Font.Name, text.Font.Size,
                     FontStyle.Italic ^ FontStyle.Bold);
     } // End of radioButton_CheckedChanged

  } // End of RadioB
 }
```

这个程序的输出如图 14-2 所示。

小 结

目前大多数广泛使用的编程语言都包含异常处理。

Ada 提供了大量的异常处理支持，以及一组小而全的内建异常。异常处理程序附加在程序实体上，如果没有局部异常处理程序，异常可以隐式或显式地传播到其他程序实体。

C++不包含预定义的异常(除了在标准库中定义的异常之外)。C++的异常是基本类型、预定义类或用户自定义类的对象。通过将 throw 语句中表达式的类型连接到异常处理程序的形参类型上，可以将异常绑定到异常处理程序上。所有的异常处理程序拥有相同的名字 catch。C++方法的 throw 子句列出了该方法可能抛出的异常类型。

Java 的异常是对象，其祖先必须可追溯到一个从 Throwable 类派生的类。有两类异常——要检查的异常和未检查的异常。用户程序和编译器考虑要检查的异常。未检查的异常可能在任何地方发生，用户程序通常会忽略它们。

Java 方法的 throws 子句列出可能抛出但不处理的要检查的异常，这些异常从它所调用的方法抛出，并传播回它的调用者。

Java 的 finally 子句提供了一种机制，保证不管 try 复合语句的执行如何终止，都要执行某些代码。

Java 现在包含了一条 assert 语句，用于防御性编程。

事件是对某事发生并需要处理的通告。事件通常通过图形用户界面，由用户和程序的交互动作产生。Java 的事件处理程序通过事件监听器来调用。如果在事件发生时要通知事件监听器，就必须为该事件注册该事件监听器。两个最常用的事件监听器接口是 actionPerformed 和 itemState- Changed。

Windows Forms 是.NET 语言中建立 GUI 组件、处理事件的最初方法。C#应用程序用这种方法建立 GUI 时，要从 Form 类中派生子类。所有的.NET 事件处理程序使用相同的协议。事件处理程序在注册时，要创建一个 EventHandler 对象，将它赋予与 GUI 对象(它可以触发事件)相关的预定义委托。

文献注释

关于异常处理最重要的论文之一出自 Goodenough(1975)，并不局限于特定的编程语言。PL/I 关于异常处理的设计问题在 MacLaren(1977)中有涉及。Liskov 和 Snyder(1979)清楚地介绍了 CLU 的异常处理设计。Ada 语言的异常处理支持在 ARM(1995)中有介绍，Romanovsky 和 Sandén(2001)对此有细致的评价。Stroustrup(1997)介绍了 C++中的异常处理，Campione 等(2001)介绍了 Java 的异常处理。

复 习 题

1. 给出异常、异常处理程序、抛出异常、禁用异常、继续、析构和内建异常的定义。
2. 设计继续的两种可选方法是什么？

3. 在语言中内建对异常处理的支持有什么好处？
4. 异常处理的设计问题有哪些？
5. 将异常绑定到异常处理程序是什么意思？
6. Ada 中异常的可能框架是什么？
7. 在 Ada 中，如果从子程序中抛出的异常未处理，会传播到哪里？从程序块中抛出呢？从包的体中抛出呢？从任务中抛出呢？
8. 在 Ada 中处理异常后，执行从哪里继续？
9. 在 Ada 中如何显式抛出异常？
10. 在 Ada 的 Standard 包中定义了哪四种异常？
11. 在 Ada 中如何定义用户自定义异常？
12. 在 Ada 中如何禁用异常？
13. 说出 Ada 异常处理的三个问题。
14. 所有 C++异常处理程序的名字是什么？
15. 在 C++中如何显式抛出异常？
16. 在 C++中如何将异常绑定到异常处理程序？
17. 在 C++中如何编写异常处理程序，可以让它处理任意异常？
18. 当 C++的异常处理程序完成执行后，执行控制如何继续？
19. C++包含内建异常吗？
20. 为什么在 C++中抛出异常不称为 raise？
21. 所有 Java 异常类的根类是什么？
22. 大多数 Java 用户自定义异常类的父类是什么？
23. 在 Java 中如何编写异常处理程序，可以让它处理任意异常？
24. C++的 throw 说明与 Java 的 throws 子句有何区别？
25. 在 Java 中要检查的异常与未检查的异常有何区别？
26. 在 Java 中如何编写异常处理程序，可以让它处理任意异常？
27. 可以禁用 Java 异常吗？
28. Java 的 finally 子句有什么用？
29. 语言定义的断言与简单的 if-write 结构相比，有什么优点？
30. 异常处理与事件处理在哪些方面相关？
31. 定义事件和事件监听器。
32. 什么是事件驱动的程序设计？
33. Java 的 JFrame 有什么作用？
34. Java 的 JPanel 有什么作用？
35. 哪个对象常常用作 Java GUI 应用程序中的事件监听器？
36. Java 事件处理程序的协议起源于什么？
37. 哪个方法用于注册 Java 中的事件处理程序？

38. 使用.NET 的 Windows Forms，需要哪个名称空间为 C#应用程序建立 GUI?
39. 如何使用 Windows Forms 将一个组件放在表单上?
40. .NET 事件处理程序的协议是什么?
41. 必须创建哪个类的对象，才能注册.NET 事件处理程序?
42. 委托在注册事件处理程序的过程中有什么作用?

习　题

1. C 语言的设计者没有要求下标范围检查，换取的是什么?
2. 说出在不直接提供异常处理支持的语言中，异常处理的三种方法。
3. 在关于 PL/I 和 Ada 的程序设计教科书中，查找内建的异常集合。从完整性和灵活性上，对两者做全面的评价。
4. 根据(ARM，1995)，确定在会合时发生的异常是如何处理的。
5. 根据一本 COBOL 的教科书，确定在 COBOL 程序中是如何进行异常处理的。
6. 在没有异常处理结构的语言中，大多数子程序普遍包含一个“错误”参数，可以设置某个值表示 OK，或设置别的值表示处理过程中出错。像 Ada 中那样的异常处理语法结构，比这种方法好在哪里?
7. 在没有异常处理结构的语言中，可以把异常处理过程作为参数，发送给每一个可以检测到必须处理的错误的过程。这种方法有什么缺点?
8. 比较题 6 和题 7 中的两种方法。你认为哪种方法更好，为什么?
9. 写出 C++的 throw 子句和 Java 的 throws 子句的对比分析。
10. 比较 C++和 Ada 的异常处理结构。在你看来，哪一种设计最灵活?哪一种语言可以写出更可靠的程序?
11. 考虑以下的 C++框架程序:

```
class Big {
  int i;
  float f;
  void fun1() throw int {
      ...
      try {
        ...
        throw i;
    ...
    throw f;
    ...
   }
  catch(float) { ... }
  ...
 }
}
class Small {
```

```
int j;
float g;
void fun2() throw float {
  ...
  try {
    ...
    try {
      Big.fun1();
      ...
      throw j;
      ...
      throw g;
      ...
    }
    catch(int) { ... }
    ...
  }
  catch(float) { ... }
}
}
```

在四个 throw 语句中，异常在哪里处理？请注意，fun1 在类 Small 中被 fun2 调用。

12. 写出 C++和 Java 的异常处理功能的详细比较。
13. 参考一本 ML 语言的书，写出 ML 和 Java 的异常处理功能的详细比较。
14. 总结支持控制继续的终止模型和重新开始模型的各种意见。

编程练习

1. 写一段 Ada 代码，尝试重新调用过程 Tape_Read，这个过程从磁带驱动器中读取输入，可抛出 Tape_Read_Error 异常。
2. 假设要编写一个 C++函数，有三种可选的方案来实现需求。写出这个函数的框架程序，当第一种方案抛出异常时，尝试第二种方案，当第二种方案抛出异常时，尝试第三种方案。假设三种方案是名为 alt1、alt2 和 alt3 的过程，写出代码。
3. 编写一个 Java 程序，从键盘上输入一系列范围为-100 到 100 的整数值，计算输入值的平方和。这个程序必须使用异常处理来确保输入值在范围内且是合法的整数，处理平方和大于标准 Integer 变量可存储值的错误，还要检测文件结束并用来产生结果输出。对于和溢出的情况，必须打印一条错误消息并终止程序。
4. 根据编程练习 3 的说明，编写一个 C++程序。
5. 根据编程练习 3 的说明，编写一个 Ada 程序。
6. 修改 14.4.5 节中的 Java 程序，使用 EOFException 来检测输入的结束。
7. 用 C++重写 14.4.6 节中使用 finally 子句的 Java 代码。

第15章

函数式程序设计语言

15.1 概述
15.2 数学函数
15.3 函数式程序设计语言基础
15.4 第一种函数式程序设计语言 LISP
15.5 Scheme 概述
15.6 Common LISP
15.7 ML
15.8 Haskell
15.9 F#
15.10 基本命令式语言对函数式编程的支持
15.11 函数式语言和命令式语言的比较

本章介绍函数式程序设计，以及为了使用这种软件开发方法而设计的一些程序设计语言。我们先复习数学函数的基本思想，因为函数式语言是以数学函数为基础的。然后介绍函数式程序设计语言的思想，描述第一种函数式语言 LISP，以及它的表数据结构和基于 λ 标记法的函数语法。接着用较长的一节来介绍 Scheme，包括它的基本函数、特殊形式、函数形式，以及一些用 Scheme 编写的简单函数示例。接下来简要介绍 Common LISP、ML、Haskell 和 F#。之后讨论对函数式编程的支持开始出现在一些命令式语言中。再后面的一节介绍函数式程序设计语言的一些应用。最后对函数式语言和命令式语言进行简短的比较。

15.1 概述

本书的前 14 章主要涉及命令式程序设计语言。如第 1 章所述，命令式语言之间的高度相似性部分来自于语言设计的共同基础：冯·诺依曼结构。命令式语言大都是为了提高基本模型 Fortran I 而在开发上的进步。所有命令式语言的设计都是为了高效地使用冯·诺依曼结构的计算机。虽然命令式程序设计被大多数程序员所接受，但是一些人认为，它对底层体系结构的严重依赖，是软件开发过程中不必要的限制。

其他一些语言的设计基础，有的更加面向特定的程序设计模式或方法学，而不是在特定计算机体系结构上的高效执行。然而至今为止，只有一小部分程序是用非命令式语言编写的。

函数式程序设计模式基于数学函数，是最重要的非命令式语言的设计基础。这种程序设计方式由函数式程序设计语言支持。

由于 John Backus 在 Fortran 开发中的贡献，获得了 1977 年的 ACM 图灵奖。在正式颁奖时，每一位获奖者都要做一个演讲，这个演讲随后会发表在 *Communications of the ACM* 杂志上。Backus 在他的图灵奖演讲上，给出了纯函数式程序设计语言比命令式语言好的理由：因为纯函数式程序设计语言得到的程序可读性更强，更可靠，更可能正确(Backus，1978)。他的看法的要点是纯函数式程序在开发时和开发后都更容易理解，这主要因为表达式的意义独立于表达式的上下文(纯函数式程序设计语言的一个独特属性就是表达式和函数都没有副作用)。

在这个演讲中，Backus 提出了一种纯函数式语言——FP(functional programming)，也就是他用来支持自己观点的语言。虽然这个语言没有成功，至少没有获得广泛使用，但他的思想激发了对纯函数式程序设计语言的讨论与研究。这里的重点是，一些著名的计算机科学家试图推广这样的概念：函数式程序设计语言比传统的命令式语言更优越，虽然这些努力显然没有达到目的。但是，在过去的十年里，随着类型化函数式语言的成熟，例如 ML、Haskell、OCaml 和 F#，对函数式编程语言的兴趣和使用一直在增加。

用命令式语言编写的程序的一个基本特点是它们有状态，状态在程序执行过程中会改变。这个状态用程序的变量来表示。程序的作者和所有读者都必须理解程序使用了变量，以及程序的状态在执行过程中如何改变。对于大型程序，这是一个令人望而生畏的任务。

用命令式语言编写的程序有这个问题，而用纯函数式语言编写的程序没有，这种程序没有变量，也没有状态。

LISP 最初是纯粹的函数式语言，但很快就获得了一些重要的命令式特性，提高了它的执行效率。它是唯一得到广泛使用的函数式语言，至少从这个意义上来说，它仍然是最重要的函数式语言。它主要应用于知识表达、机器学习、智能训练系统和语音建模等领域。Common LISP 是 20 世纪 80 年代早期一些 LISP 方言的混合语言。

Scheme 是一个小型的、静态作用域的 LISP 方言，广泛用于讲授函数式编程，也在一些大学中用于讲授初级编程课程。

类型化函数式编程语言——主要是 ML、Haskell、OCaml 和 F#——的发展，显著扩展了函数式语言目前所应用的计算领域。随着这些语言的成熟，它们的实际应用也在增加，它们现在用于数据库处理、财务建模、统计分析和生物信息学等领域。

本章的一个目的是用 Scheme 核心来介绍函数式程序设计，并有意不涉及它的命令式特性。本章包含了充足的 Scheme 材料，使读者可以编写一些简单而有趣的程序。没有实际的程序设计经验，很难获得函数式程序设计的实际感受，因此这些程序的编写是强烈推荐的。

15.2　数学函数

数学函数是元素从一个集合(称为定义域集合)到另一个集合(称为值域集合)的映射。函数的定义显式或隐式地给出了定义域和值域，以及该映射。该映射用表达式来描述，在某些情况下用表来描述。函数通常应用在定义域集合的特定元素(作为函数的参数)上。请注意，定义域集合可以是一些集合的叉积(表明可以有一个以上的参数)。函数返回值域集合的一个元素。

数学函数的基本特征之一是，映射表达式的求值顺序由递归表达式和条件表达式控制，而不是命令式程序设计语言中常见的顺序执行或迭代重复。

数学函数另一个重要的特征是，由于它们没有副作用，不能依赖任何外部值，所以给定相同的自变量集合，总是映射值域集合中的同一个元素。但命令式语言中的子程序可能依赖几个非局部或全局变量的当前值，因此很难静态确定子程序输出的值，在某次执行时会产生什么副作用。

在数学中，没有像模拟存储单元的变量这样的东西。在命令式程序设计语言中，函数中的局部变量维护了函数的状态。计算时，要求出赋值语句中的表达式，改变程序的状态。在数学中，没有函数状态这一概念。

数学函数将其参数映射到值，而不是对内存中的值指定一系列操作来产生一个值。

15.2.1　简单函数

函数定义通常写成函数名后跟圆括号内的一组参数，之后是映射表达式。例如：

cube(x)≡x* x*x，其中 x 是实数。

在这个定义中，定义域和值域都是实数，符号 ≡ 表示“定义为”。参数 x 可以表示定义域集合内的任意元素，但在对函数表达式求值时，固定地表示某个特定元素。这就是数学函数的参数不同于命令式语言中的变量的一个方面。

指定函数的应用时，要将函数名和定义域集合的特定元素配对。对函数映射表达式求值时，用定义域元素来替换参数，得到值域元素。另外要注意，在求值时，函数的映射不包含未绑定的参数，绑定的参数是特定值的名字。每一个参数都绑定为定义域的一个值，在求值过程中是常量。例如下面对 cube(x)的求值：

cube (2.0) = 2.0 * 2.0 * 2.0 = 8

在求值过程中，参数 x 绑定到 2.0 上，没有未绑定的参数。而且，x 是一个常量(其值不能改变)。

早期关于函数的理论研究，将定义函数与命名函数的任务分隔开。Alonzo Church(1941)提出的 λ 标记法，提供了一种定义匿名函数的方法。λ 表达式说明了函数的参数和映射。λ 表达式就是函数本身，是没有名字的。例如，考虑下述 λ 表达式：

```
λ(x)x * x* x
```

Church 定义了一个使用 λ 表达式的形式计算模型(函数定义、函数应用和递归的形式系统)，称为 λ 计算。λ 计算可以是类型化的或未类型化的。未类型化的 λ 计算是函数式编程语言的灵感所在。

如前所述，在求值之前，参数表示定义域集合的任意元素，但在求值时，它绑定到特定元素上。为给定的参数计算 λ 表达式，称为将表达式应用到该参数上。这种应用的方法与任意函数的求值相同。以下例子给出了示例 λ 表达式的应用：

```
(λ(x)x*x*x)(2)
```

它的结果是 8。

λ 表达式与其他函数定义一样，可以有一个以上的参数。

15.2.2 函数形式

高阶函数(或函数形式)以一个或多个函数作为参数，或者生成一个函数作为其结果，或者两者都有。一种常见的函数形式是函数组合，它有两个函数参数，并生成一个函数，该函数的值是将第一个实参函数应用到第二个实参函数的结果上。函数组合用° 作为运算符，写成如下所示这样的表达式：

$h \equiv f^{\circ}$ g

例如，如果

$f(x)\equiv x+2$

$g(x)\equiv 3*x$

那么 h 定义为

$h(x)\equiv f(g(x))$ 或 $h(x)\equiv(3*x)+2$

apply-to-all 是以单个函数为参数的函数形式[1]。如果应用到一组自变量上，apply-to-all 就将它的函数参数应用到这组自变量的每个值上，并将结果组合成列表或序列。apply-to-all 用 α 来表示。考虑以下例子：

令

$h(x)\equiv x*x$

那么

$\alpha(h, (2, 3, 4))$生成$(4, 9, 16)$

有许多其他的函数形式，但以上两个例子展示了所有函数形式的基本特点。

15.3　函数式程序设计语言基础

设计函数式程序设计语言的目的是尽可能地模拟数学函数。这一目的产生了一种解决问题的方案，与命令式语言所用的方案有根本性的不同。在命令式语言中，是计算表达式，结果存储在存储单元中(表示为程序中的变量)。这就是赋值语句的作用。这种不可避免的对存储单元(其值表示程序的状态)的关注，导致了相对低级的程序设计方法学。

用汇编语言写的程序通常也必须存储对表达式的部分求值的结果。例如，为了计算

$(x+y)/(a-b)$

要先计算(x+y)的值，这个值必须在计算(a-b)时存储起来。在高级语言中，编译器负责存储表达式求值的中间结果。对中间结果的存储仍然是必不可少的，只不过对程序员隐藏了其细节。

纯函数式程序设计语言不使用变量或赋值语句，因此程序员不必考虑与程序的存储单元或状态相关的问题。没有变量，就不可能有迭代结构，因为迭代结构是由变量控制的。重复执行必须用递归来表示，而不是迭代结构。程序就是函数定义和函数应用的说明，程序的执行是由对函数应用的求值组成的。没有变量，纯函数式程序的执行就没有操作语义和指称语义下的状态。当给定相同的参数时，执行函数总是生成相同的结果。这一特性称为引用透明，它使纯函数式语言的语义比命令式语言(以及包含命令式特性的函数式语言)

1. 在编程语言中，这些常常称为映射函数。

的语义简单得多。这也更便于测试，因为每个函数都可以单独测试，不必考虑其上下文。

函数式语言提供了一组基本函数，一组由这些基本函数构造复杂函数的函数形式，一个函数应用操作，以及表示数据的某种或某些结构。这些结构用于表示参数和函数计算得到的值。如果函数式语言定义得很好，就只需要相对少量的基本函数。

如前面的章节所述，第一种函数式编程语言 LISP 为数据和代码使用一种非常不同于命令式语言的语法形式。但是，许多函数式语言设计者后来为代码使用类似于命令式语言的语法形式。

尽管有一些纯函数式语言，例如 Haskell，但大多数称为函数式的语言都包含一些命令式语言特性，例如用作赋值语句的可变的变量和结构。

一些来自于函数式语言的概念和结构，例如懒惰求值和匿名子程序，现在也出现在一些命令式语言中。

虽然早期的函数式语言通常用解释器来实现，但用函数式语言写的许多程序现在是编译执行的。

15.4 第一种函数式程序设计语言 LISP

人们已经开发了许多函数式程序设计语言，最早、使用最广泛的是 LISP(或它的一个派生语言)，是由 MIT 的 John MoCarthy 在 1959 年开发的。通过 LISP 来学习函数式语言，有些类似于通过 Fortran 来学习命令式语言：LISP 是第一种函数式语言，虽然它持续发展了近半个世纪，但它已经不再代表函数式语言最新的设计思想。此外，除了第一个版本以外，所有的 LISP 方言都包含命令式语言的特性，例如命令式风格的变量、赋值语句和迭代(命令式风格的变量用于命名存储单元，变量的值在程序执行时可以改变多次)。尽管这样，而且有一些奇怪的形式，原始 LISP 的后代仍然很好地展现了函数式程序设计的基本思想，因此值得学习。

15.4.1 数据类型和结构

在最初的 LISP 中只有两类数据对象：原子和表。表元素是数据对，其中第一部分是元素的数据，它是指向原子或内嵌表的指针；第二部分可以是指向原子的指针、指向另一个元素的指针或空表。表中的元素通过第二部分链接在一起。原子和表都不是命令式语言意义中的类型。实际上，最初的 LISP 是没有类型的语言。原子是以标识符为形式的符号或者都数字字面值。

如第 2 章所述，LISP 最早使用表作为它的数据结构，因为表是表处理的关键部分。然而最终发展的结果，LISP 很少需要通常的插入和删除(除了在表的开头)这两种表操作。

在 LISP 中，表是用圆括号内相互分隔的元素来表示的。简单表的元素只能是原子，如

```
(A B C D)
```

嵌套的表结构也用圆括号来表示，例如，表

```
(A (B C) D (E(F G)))
```

是一个有四个元素的表。第一个元素是原子 A，第二个元素是子表(B C)，第三个元素是原子 D，第四个元素是子表(E(F G))，第四个元素的第二个元素是子表(F G)。

在内部，表通常是以链表结构来存储的，其中每一个结点有两个指针，一个引用结点的数据，另一个指针构成了链表。表用指向第一个元素的指针来引用。

两个示例表的内部表示如图 15-1 所示。请注意，表的元素是水平显示的，表的最后一个元素没有后继，因此它的链接为 NIL。子表也用相同的结构表示。

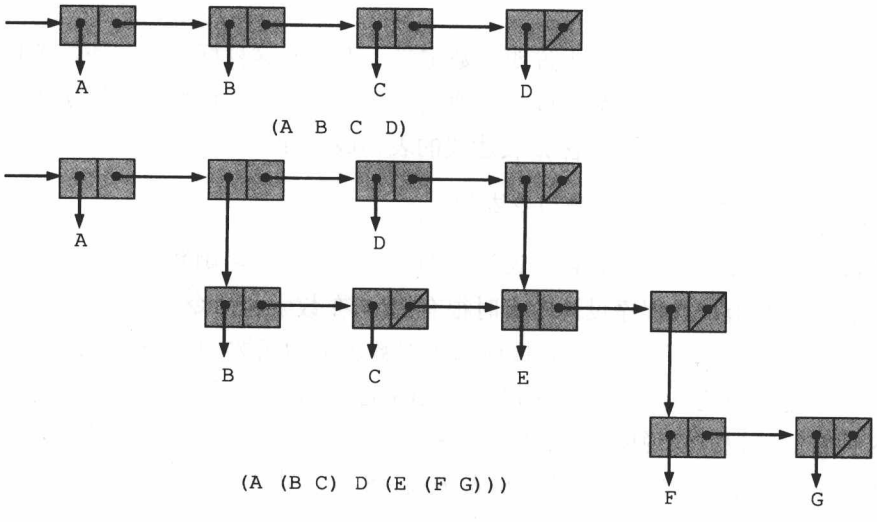


图 15-1　两个 LISP 表的内部表示

15.4.2　第一个 LISP 解释器

LISP 最初的设计意图是为了能够有一种尽可能接近 Fortran 的程序标记方法，必要时可加入一些内容。这一标记方法称为 M-标记法，即元标记法。编译器需要将用 M-标记法写的程序翻译为语义相当的 IBM 704 机器码程序。

在 LISP 发展的早期，McCarthy 写了一篇论文，建议将表处理作为通用的符号处理方法。McCarthy 相信，表处理可用于研究可计算性，这一研究在当时通常是用图灵机进行的，图灵机基于计算的命令式模型。McCarthy 认为，对符号表的函数式处理是比图灵机更自然的计算模型，后者只能操作写在磁带上的符号，这些符号表示状态。计算研究的一个普遍要求是，必须证明所有计算模型都具有某种可计算性特征。对于图灵机模型，可以构造一个通用图灵机，模拟任意其他图灵机的操作。根据这一原理，就产生了构造通用的 LISP 函数，计算任意其他 LISP 函数的想法。

通用 LISP 函数的第一个要求就是一种标记方法，允许函数以表达数据的方式来表达。15.4.1 节中带圆括号的表的标记方法，已经用于表示 LISP 数据，于是需要确定函数定义和

函数调用的规范，它也可以用表的标记方法来表达。函数调用用一种称为 Cambridge Polish[2] 的前缀表来表示，如下：

(函数名 参数$_1$ … 参数$_n$)

例如，如果+是一个带两个或多个数值参数的函数，那么下面这两个表达式的值分别是 12 和 20：

```
(+ 5 7)
(+ 3 4 7 6)
```

15.2.1 节介绍的 λ 标记法用于说明函数定义。不过，这种标记方法还需要修改，以允许将函数绑定到名字上，使函数可以被其他函数和自己引用。这种名字绑定用一个表来表示，这个表由函数名字和一个包含 λ 表达式的表构成，如

(函数名 (LAMBDA (参数$_1$ … 参数$_n$) 表达式))

如果以前没有接触过函数式程序设计，即使考虑一个匿名函数，也可能会觉得奇特。但是，匿名函数在函数式程序设计中有时很有用(与在数学和命令式编程中一样)[3]。例如，考虑一个函数，其行为是生成一个直接应用到参数表上的函数。所生成的函数不需要名字，因为它只在构造它的地方用到。15.5.14 节给出了这样的一个例子。

用这种新的标记方法说明的 LISP 函数称为 S-表达式，即符号表达式。最终，所有的 LISP 结构，包括数据和代码，都称为 S-表达式。S-表达式可以是一个表，或一个原子。S-表达式通常简称为表达式。

McCarthy 成功地设计出一个通用函数，可对任意其他函数求值。这一函数称为 EVAL，自身也是用表达式的形式来表示的。AI 项目中开发 LISP 的两个人 Stephen B. Russell 和 Daniel J. Edwards 发现 EVAL 的实现可以用作 LISP 解释器，于是立即构建了这样的一个实现(McCarthy 等，1965)。

这一快速、简单、出乎意料的实现，有一些重要的结果。第一，所有早期的 LISP 实现都复制 EVAL，因此都是解释执行的。第二，M-标记法的定义，即计划中 LISP 的程序设计标记方法，没有完成或实现，所以 S-表达式成为 LISP 的唯一标记方法。数据和代码使用相同的标记方法具有重要的结果，其中之一将在 15.5.14 节中介绍。第三，许多最初的语言设计实际上都冻结了，在语言中留下了一些奇怪的特性，例如条件表达式形式，以及用()表示空表和逻辑假。

早期 LISP 系统的另一个显然意外的特性，就是动态作用域的使用。函数是在调用者

---

2. 这个名字最初用于 LISP 的早期开发。选择这个名字是因为 LISP 表类似于波兰(Polish)逻辑学家 Jan Lukasiewicz 使用的前缀符号，而且 LISP 诞生于马萨诸塞州 Cambridge 市的 MIT。一些人现在把它称为 Cambridge 前缀。

3. 在命令式编程中也使用匿名子程序。

环境中求值的。那时没有人了解作用域，对选择作用域也没什么考虑。在 1975 年之前，大多数 LISP 方言都使用动态作用域。当代的 LISP 方言使用静态作用域，或者允许程序员选择静态作用域或动态作用域。

LISP 的解释器可以用 LISP 编写。这种解释器并不是很大的程序，但它用 LISP 描述了 LISP 的操作语义。这是该语言的语义非常简单的一个鲜活证据。

15.5　Scheme 概述

本节介绍 Scheme(Dybvig，2003)的核心部分。选择 Scheme 是因为它相对简单；流行于学院和大学中；各种计算机上的 Scheme 解释器都容易得到(而且是免费的)。本节介绍的 Scheme 版本是 Scheme 4。请注意，本节只涉及 Scheme 中的一小部分，不包括 Scheme 解释器的特性。

15.5.1　Scheme 的起源

Scheme 语言是 LISP 的方言，在 20 世纪 70 年代中期出现于 MIT(Sussman 和 Steele，1975)。它的特点是小巧，只使用静态作用域，将函数作为第一类实体。作为第一类实体，Scheme 函数可以是表达式的值，可以是表的元素、可以传递为参数，可以从函数中返回。LISP 的早期版本并不提供所有这些功能。

作为一种语法和语义简单的小规模语言，Scheme 很适合教学应用，例如函数式程序设计课程，以及对程序设计的一般性介绍。

以下小节中大多数的 Scheme 代码，只需要很少的修改就可以重写为 LISP 代码。

15.5.2　Scheme 解释器

在解释模式下，Scheme 解释器是一个读-求值-写(通常简写为 REPL)的无限循环，它反复地读入用户输入的表达式(以表的形式)，解释这个表达式，并显示结果值。Ruby 和 Python 也使用这种形式的解释器。表达式用 EVAL 函数来解释。对字面值的求值就是字面值本身，因此如果向解释器输入一个数，它就简单地显示这个数。调用基本函数的表达式用以下方法求值：首先，对每一个参数表达式求值，这没有特定顺序；然后将基本函数应用到参数值上，并显示结果值。

当然，存储在文件中的 Scheme 程序可以加载和解释。

在任何代码行上，Scheme 中的注释可以是跟在一个分号后面的任意文本。

15.5.3　基本数值函数

Scheme 包含基本算术运算的基本函数，有+、-、*和 / ，分别表示加法、减法、乘法和除法。*和+可以有零个或多个参数。如果*没有参数，就返回 1；如果+没有参数，就返回 0。+把所有参数加在一起，*把所有参数乘在一起。 / 和-可以有两个或多个参数。减法

是从第一个参数中减去所有其他的参数。除法与减法类似。下面是一些例子：

表达式	值
42	42
(* 3 7)	21
(+ 5 7 8)	20
(- 5 6)	-1
(- 15 7 2)	6
(- 24 (* 4 3))	12

Scheme 还有许多其他数值函数，包括 MODULO、ROUND、MAX、MIN、LOG、SIN 和 SQRT。如果参数值非负，SQRT 就返回其数值参数的平方根；如果参数值为负，SQRT 就生成一个复数。

在 Scheme 中，注意所有保留字和预定义函数都使用大写字母。该语言的正式定义指定，大写字母和小写字母没有区别。但是，一些实现，例如 DrRacket 的教学语言，要求保留字和预定义函数使用小写字母。

如果函数的参数个数固定，例如 SQRT，则调用中的参数个数就匹配这个数字；否则，解释器就会生成一个错误消息。

15.5.4 定义函数

Scheme 函数是一组函数定义。因此，知道如何定义这些函数，是编写最简单的程序的前提条件。在 Scheme 中，匿名函数实际上包含了 LAMBDA，称为 λ 表达式，例如

```
(LAMBDA (x) (* x x))
```

是一个匿名函数，返回所给数值参数的平方。这个函数能够以与命名函数同样的方式应用：将它放在包含实参的表的起始处。例如，以下表达式得到 49：

```
((LAMBDA (x) (* x x)) 7)
```

在这个表达式中，x 称为 λ 表达式中的绑定变量。在计算这个表达式时，x 绑定到 7 上。在 λ 表达式求值开始时，绑定变量被绑定到实参值，之后在表达式中不再改变。

λ 表达式可以有任意多个参数。例如，可以有如下定义：

```
(LAMBDA (a b c x) (+ (* a x x) (* b x) c))
```

Scheme 中特殊形式的函数 DEFINE 对 Scheme 程序设计有两个基本作用：将名字绑定到值上，以及将名字绑定到 λ 表达式上。前一种用法听起来好像是 DEFINE 可用于创建命令式语言风格的变量，但是这样的名字绑定只创建命名常量，而不是变量。

DEFINE 称为特殊形式，因为它的解释方式(由 EVAL 解释)不同于正常的基本函数(如算术函数)，如下所示。

DEFINE 的最简单形式用来将名字绑定到表达式的值上。这一形式是

```
(DEFINE 符号 表达式)
```

例如

```
(DEFINE pi 3.14159)
(DEFINE two_pi ( * 2 pi))
```

如果这两个表达式输入到 Scheme 解释器中，然后输入 pi，就会显示 3.14159；输入 two_pi，会显示 6.28318。在这两种情况下，显示的数可能比这里给出的小数位更多。

这种形式的 DEFINE 类似于命令式语言中命名常量的声明。例如在 Java 中，上述定义应如下：

```
final float PI = 3.14159;
final float TWO_PI = 2.0 * PI;
```

Scheme 中的名字可以包含字母、数字和除圆括号以外的特殊字符，名字不区分大小写，但不能以数字开头

DEFINE 函数的第二种用法是将 λ 表达式绑定到名字上，这时通过去掉 LAMBDA 来简化 λ 表达式。为了将名字绑定到 λ 表达式上，DEFINE 以两个表作为参数。第一个参数是函数调用的原型，在函数名字后面跟形参，都放在一个表中。第二个参数包含名字要绑定的表达式。这一 DEFINE 的一般形式是[4]：

```
(DEFINE(函数名 参数)
   (表达式)
)
```

当然，这种形式的 DEFINE 仅是命名函数的定义。

以下例子调用 DEFINE，将名字 square 绑定到带一个参数的函数表达式上：

```
(DEFINE (square number) (* number number))
```

解释器对函数求值后，它就可以使用了，如

```
(square 5)
```

显示 25。

为了显示基本函数与 DEFINE 特殊形式的区别，考虑

```
(DEFINE x 10)
```

如果 DEFINE 是一个基本函数，EVAL 对这个表达式的第一个动作是对 DEFINE 的两个参数求值。如果 x 还没有绑定到值上，就是一个错误。而且，如果 x 已经被定义，仍然是错误，因为这个 DEFINE 试图重定义 x，这是不允许的。注意，x 是一个值的名字，不是命令式语言中的变量。

---

4. 实际上，DEFINE 的一般形式以一个表为函数体，该表包含一个或多个表达式，但在大多数情况下，只包含一个表达式。为了简单起见，这里只包含了一个表达式。

以下是函数的另一个例子。给定直角三角形的两条边，计算斜边(最长边)的长度。

```
(DEFINE (hypotenuse side1 side2)
    (SQRT(+(square side1)(square side2)))
)
```

请注意，hypotenuse 使用了前面定义的 square。

15.5.5 输出函数

Scheme 包含一些简单的输出函数，然而与交互式解释器一起使用时，Scheme 程序大多数的输出都是解释器的正常输出，显示将 EVAL 应用到顶层函数的结果。

Scheme 包含一个格式化输出函数 PRINTF，它类似于 C 中的 printf 函数。

注意显式输入输出不是纯函数式编程模型的一部分，因为输入操作改变了程序的状态，输出操作有副作用。它们都不是纯函数式语言的一部分。

15.5.6 数值谓词函数

谓词函数是一个返回布尔值(真或假)的函数。Scheme 包含一组用于数值数据的谓词函数，其中有：

函数	含义
=	等于
<>	不等于
>	大于
<	小于
>=	大于等于
<=	小于等于
EVEN?	是偶数吗
ODD?	是奇数吗
ZERO?	是零吗

请注意，所有用单词作为名字的预定义谓词函数，其名字都以问号结尾。在 Scheme 中，两个布尔值是#T 和#F(或#t 和#f)，但一些实现使用空表表示假[5]。Scheme 预定义的谓词函数返回空表()表示假。

表解释为布尔值时，非空表就表示真，空表表示假。这类似于 C 中把整数解释为布尔值，0 表示假，非 0 值表示真。

为了提高可读性，本章所有示例谓词函数都返回#F，而不是()。

NOT 函数用于反转布尔表达式的逻辑。

---

5. 其他一些实现显示 true 和 false，而不是#T 和#F。

15.5.7　控制流

Scheme 使用三种不同的控制流结构：一种模仿了命令式语言的选择结构，另两种基于数学函数中使用的求值控制。

Scheme 的双选结构函数 IF 有三个参数：谓词表达式、then 表达式和 else 表达式。调用 IF 的形式如下：

```
(IF 谓词 then 表达式 else 表达式)
```

例如

```
(DEFINE (factorial n)
  (IF (<= n 1)
    1
    (* n (factorial (- n 1)))
))
```

第 8 章讨论了 Scheme 的多选结构 COND。下面的简单函数示例使用了 COND：

```
(DEFINE (leap? year)
  (COND
    ((ZERO? (MODULO year 400)) #T)
    ((ZERO? (MODULO year 100)) #F)
    (ELSE (ZERO? (MODULO year 4)))
))
```

后面的小节包含了其他使用 COND 的例子。

Scheme 的第三种控制机制是递归，与数学中一样，用于控制重复执行。15.5.10 节中大多数示例函数都使用了递归。

15.5.8　表函数

基于 LISP 的程序设计语言的一个常见用途就是表处理。这一小节介绍 Scheme 中处理表的函数。第 6 章简要介绍了 Scheme 的表处理。下面详细讨论 Scheme 的表处理。

Scheme 程序用工具函数 EVAL 来解释。当应用到基本函数时，EVAL 先对给定函数的参数求值。当函数调用的实参本身就是函数调用时(这种情况经常出现)，这样的做法是必要的。不过在有些调用中，参数是数据元素而不是函数引用。当参数不是函数引用时，显然不应该求值。前面没有涉及这一点，因为数值字面值的求值总是它自己，不会误认为是函数名。

假设一个函数有两个参数，一个原子和一个表，该函数的用途是确定给定的原子是否在给定的表中。原子和表都不应该求值，它们是要检查的字面数据。为了避免对参数求值，先把参数提供为基本函数 QUOTE 的参数，该函数只是返回参数而不做改变。下面的例子展示了 QUOTE：

```
(QUOTE A)返回A
(QUOTE (A B C))返回(A B C)
```

在本章的余下部分，使用 QUOTE 调用的常用缩写，即在要引用的表达式前放一个单引号(')。因此，使用'(A B)来代替(QUOTE(A B))。

需要 QUOTE 函数是因为 Scheme(和其他基于 LISP 的语言)的基本特性：数据和代码的形式相同。尽管这对于命令式语言程序员有些古怪，但会得到一些有趣、强大的过程，参见 15.5.14 节。

第 6 章介绍了 CAR、CDR 和 CONS 函数。下面是 CAR 和 CDR 操作的示例：

```
(CAR '(A B c))返回A
(CAR '((A B) C D))返回(A B)
(CAR 'A)是错误的，因为A不是一个表
(CAR '(A))返回A
(CAR '())是错误的
(CDR '(A B C))返回(B c)
(CDR '((A B) C D))返回(C D)
(CDR 'A)是错误的
(CDR '(A))返回()
(CDR '())是错误的
```

CAR 和 CDR 函数的名字是非常特殊的，这两个名字起源于 LISP 的第一个实现，那时是在 IBM 704 计算机上。704 计算机的内存字有两个域，称为减量(decrement)和地址(address)，用于各种寻址策略的操作数。这两个域都可以存储一个机器内存地址。704 机器还包含两条机器指令，也称为 CAR(地址寄存器的内容)和 CDR(减量寄存器的内容)，它们可提取相关域的值。用这两个域来存储一个表结点的两个指针，使一个内存字可以巧妙地存储一个结点，这很自然。使用这样的约定，704 计算机的 CAR 和 CDR 指令提供了高效的表选择函数。这两个名字沿用到所有 LISP 方言的基本函数中。

另一个简单函数的例子如下

```
(DEFINE (second a_list) (CAR (CDR a_list)))
```

一旦对该函数求值，它就可以使用了，如

```
(second '(A B C))
```

返回 B。

Scheme 中一些最常用的函数组合内建为单个函数。例如，(CAAR x)等价于(CAR(CAR x))，(CADR x)等价于(CAR (CDR x))，(CADDAR x)等价于(CAR (CDR (CDR (CAR x))))。在函数名的C和R之间，A和D的任意组合至多有4个，这是合法的。例如下面的CADDAR：

```
(CADDAR '((A B (C) D) E)) =
(CAR (CDR (CDR (CAR '((A B (C) D) E))))) =
(CAR (CDR (CDR '(A B (C) D)))) =
```

```
(CAR (CDR '(B (C) D))) =
(CAR '((C) D)) =
(C)
```

下面是 CONS 的示例调用：

```
(CONS 'A '())返回(A)
(CONS 'A '(B C))返回(A B C)
(CONS '() '(A B))返回(() A B)
(CONS '(A B) '(C D))返回((A B) C D)
```

这些 CONS 运算的结果如图 15-2 所示。请注意，CONS 在某种意义上是 CAR 和 CDR 的反函数。CAR 和 CDR 将表拆开，CONS 根据给定表的部分构建新表，CONS 的两个参数成为新表的 CAR 和 CDR。因此，如果 a_list 是一个表，那么

```
(CONS (CAR a_list) (CDR a_list))
```

返回与 a_list 的结构和元素相同的表。

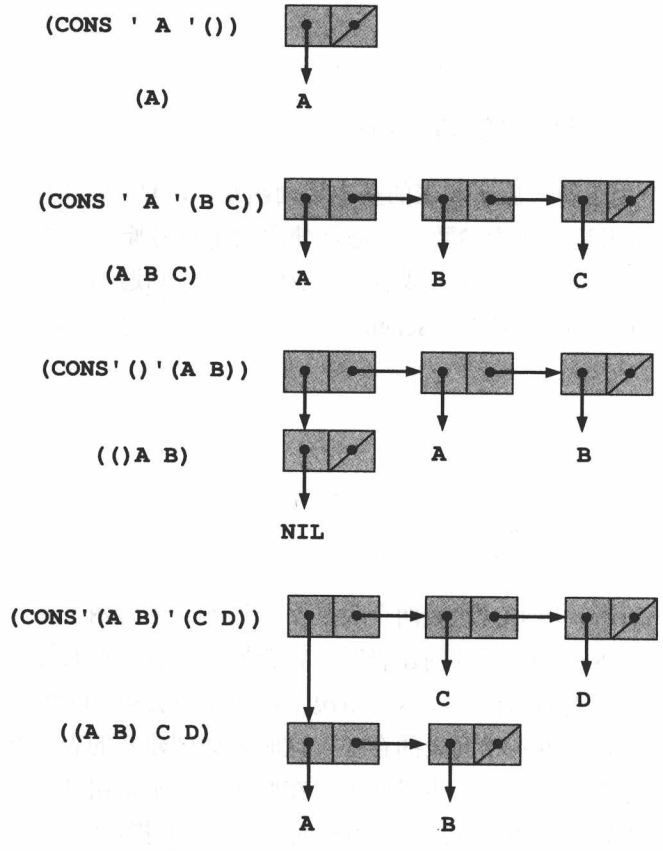


图 15-2　一些 CONS 运算的结果

只处理本章介绍的相对简单的问题与程序，不可能故意将 CONS 应用到两个原子上。

然而这是合法的。这种应用的结果是一个点对，这一名字来自于 Scheme 显示的方式。例如，考虑以下调用：

```
(CONS 'A 'B)
```

如果显示这一结果，似乎是：

```
(A . B)
```

这个点对表示这一结点有两个原子，而不是一个原子和一个指针，或两个指针。

LIST 是一个函数，根据数目不定的参数构造一个表，它是嵌套 CONS 函数的速记版本，如下所示：

```
(LIST 'apple 'orange 'grape)
```

返回

```
(apple orange grape)
```

使用 CONS，上述 LIST 调用应为：

```
(CONS 'apple (CONS 'orange (CONS 'grape '())))
```

15.5.9 用于符号原子和表的谓词函数

Scheme 有三个用于符号原子和表的基本谓词函数：EQ?、NULL?和 LIST?。

EQ?函数以两个表达式作为参数，但通常使用两个符号原子参数。如果两个参数有相同的指针值，即它们指向相同的原子或表，就返回#T，否则返回#F。如果两个参数都是符号原子，若它们是相同的符号(因为 Scheme 不复制符号)，EQ? 就返回#T，否则返回#F。考虑以下例子：

```
(EQ? 'A 'A) 返回 #T
(EQ? 'A 'B) 返回 #F
(EQ? 'A '(A B)) 返回 #F
(EQ? '(A B) '(A B)) 返回 #F 或 #T
(EQ? 3.4 (+ 3 0.4)) 返回 #F 或 #T
```

如第四个例子所示，用 EQ?比较两个表是不一致的，因为两个完全相同的表通常在内存中不是复制的。当 Scheme 系统创建表时，它会检查是否已经有这样一个表，如果有，新的表就只是指向已有表的指针。在这种情况下，两个表会被 EQ?判定为相等。但是在另一些情况下，可能很难检测到相同表的存在，这时就会创建新的表，此时 EQ?给出#F。

最后一个例子说明，加法可以生成新值，在此例中它不是用 EQ?得到 3.4，因为该例可能发现已经有值 3.4 了，此时 EQ?会使用指向旧值 3.4 的指针，返回#T。

EQ?用于符号原子，但不一定可用于数值原子。谓词=用于数值原子，而不是符号原子。如前所述，EQ?不能可靠地用于表参数。

有时不知道两个原子是符号原子还是数值原子，但能测试它们是否相等会很方便。为

此，Scheme 有一个不同的谓词 EQV?，可用于数值原子和符号原子。考虑下面的例子：

```
(EQV? 'A 'A) 返回 #T
(EQV? 'A 'B) 返回 #F
(EQV? 3 3) 返回 #T
(EQV? 'A 3) 返回 #F
(EQV? 3.4 (+ 3 0.4)) 返回 #T
(EQV? 3.0 3) 返回 #F
```

注意最后一个例子说明，浮点值不同于整数值。EQV? 不比较指针，而比较值。

在可以用的情况下使用 EQ? 或=而不是 EQV?的主要原因是，EQ?和=比 EQV?快。

当单个的参数是表时，LIST?谓词函数返回#T，否则返回#F，如下面的例子所示：

```
(LIST? '(X Y)) 返回 #T
(LIST? 'X) 返回 #F
(LIST? '()) 返回 #T
```

NULL?函数测试参数，确定它是否为空表，如果是，则返回#T。考虑以下例子：

```
(NULL? '(A B)) 返回 #F
(NULL? '()) 返回 #T
(NULL? 'A) 返回 #F
(NULL? '(())) 返回 #F
```

最后一个函数调用得到#F，因为参数不是空表，相反它是包含了一个元素的表，该元素是空表。

15.5.10　Scheme 函数示例

这一节包含了 Scheme 中函数定义的一些例子。这些程序解决简单的表处理问题。

考虑给定原子是否在一个不包含子表的表内这个问题。这样的表称为简单表。如果函数名为 member，则可以像下面这样使用它：

```
(member 'B '(A B C)) 返回 #T
(member 'B '(A C D E)) 返回 #F
```

从迭代的角度看，上述问题就是比较给定原子与给定表中的单个元素，按某种顺序一次一个，直到找到匹配或者表中没有别的元素要比较了。类似的过程可以用递归来实现。函数可以比较给定原子和表的 CAR，如果匹配就返回#T，如果不匹配就忽略表的 CAR，在表的 CDR 上继续搜索，为此，可以以表的 CDR 作为表参数来调用函数，返回递归调用的结果。在表中找到给定原子时，这一过程结束。如果原子不在表中，最终会以一个空表作为实参来调用函数，这时必须强制函数返回#F。在这一过程中，有两种退出递归的方式：在某次调用时表为空，并返回#F，或者找到匹配，并返回#T。

同时，在函数中还必须处理三种情况：空的输入表、原子与表的 CAR 匹配，以及原子与表的 CAR 不匹配并导致递归调用。这三种情况是 COND 的三个参数，最后一种

情况是由 ELSE 谓词触发的默认情况。完整函数如下[6]：

```
(DEFINE (member atm a_list)
  (COND
    ((NULL? a_list) #F)
    ((EQ? atm (CAR a_list)) #T)
    (ELSE (member atm (CDR a_list)))
))
```

这种形式在简单的 Scheme 表处理函数中很典型。在这个函数中，表中的数据一次处理一个元素，单个元素用 CAR 指明，使用在表的 CDR 上的递归来继续处理过程。

请注意，空表测试必须在相等测试之前，因为将 CAR 应用在空表上是错误的。

作为另一个例子，考虑判定两个给定表是否相等的问题。如果两个表都是简单表，解决方案相对容易，虽然可能会涉及读者不熟悉的某些程序设计技巧。谓词函数 equalsimp 比较两个简单表，如下：

```
(DEFINE (equalsimp list1 list2)
  (COND
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((EQ? (CAR list1) (CAR list2))
          (equalsimp (CDR list1) (CDR list2)))
    (ELSE #F)
))
```

第一种情况是：当第一个表参数是空表时，由 COND 的第一个参数处理。如果第一个表参数一开始就是空表，这一情况就会在外部调用时发生。因为递归调用使用两个参数表的 CDR 作为参数，如果第一个表的所有元素都被先前的递归调用移除，那么在这个调用中第一个表就为空。当第一个表参数为空时，必须检查第二个表是否也为空。如果也为空，它们就相等(一开始就相等，或者它们的 CAR 在先前的递归调用中都相等)，NULL?会正确地返回#T。如果第二个表不为空，那么它比第一个表大，应该返回#F，NULL?也会这样返回#F。

第二个表为空而第一个表非空，由第二种情况来处理。这种情况只在第一个表比第二个表长时发生。只需要测试第二个表，因为第一种情况捕获了第一个表为空的所有情况。

第三种情况是递归的步骤，测试两个表中的对应元素是否相等，这通过比较两个非空表的 CAR 来实现。如果相等，那么两个表到这一点都是相等的，因此用递归来处理两者的 CDR。当找到两个不相等的原子时，测试失败，这时处理过程不必继续，因此选择默认情况 ELSE，返回#F。

请注意，equalsimp 把表作为参数，如果一个或两个参数是原子，它就不能正确运算。

---

6. 大多数 Scheme 系统都定义了函数 member，且不允许用户重定义它。所以，如果用户希望试用这个函数，就必须用其他名称定义它。

比较一般表的问题比这个问题更复杂，因为必须在比较过程中完整地跟踪子表。在这种情况下，唯有递归才适合，因为子表的形式与给定表的形式相同。当两个给定表的对应元素也是表时，就把它们拆分成两部分 CAR 和 CDR，都用递归来处理。这是分治方法有效的完美例子。如果两个给定表的对应元素都是原子，则可以简单地用 EQ?来比较。

完整的函数定义如下：

```
(DEFINE (equal list1 list2)
  (COND
    ((NOT (LIST? list1)) (EQ? list1 list2))
    ((NOT (LIST? list2)) #F)
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((equal (CAR list1) (CAR list2))
            (equal (CDR list1) (CDR list2)))
    (ELSE #F)
))
```

COND 的前两种情况，处理某个参数为原子而不是表的情况。第三种和第四种情况用于一个表或两个表为空的情况。这些情况使后面的情况不会试图将 CAR 应用到空表中。COND 的第五种情况是最有趣的，谓词用两个表的 CAR 作为参数递归调用。如果递归调用返回#T，那么再对两个表的 CDR 使用递归。这个算法允许两个表包含任意深度的子表。

这个 equal 的定义适用于任何表达式对，而不仅仅是表。equal 与系统的谓词函数 EQUAL?相当。请注意，EQUAL?应该只在必要时使用(实参的形式未知)，因为它比 EQ?和 EQV?要慢得多。

另一个经常需要的表操作是构建一个新表，它包含两个给定表的所有元素，通常用 Scheme 函数 append 实现。反复使用 CONS 将第一个表参数的元素放入第二个表参数中，就构造出了最终的表。为了讲清楚 append 的行为，考虑以下例子：

```
(append '(A B) '(C D R)) 返回 (A B C D R)
(append '((A B) C) '(D (E F))) 返回 ((A B) C D (E F))
```

append 的定义是[7]：

```
(DEFINE (append list1 list2)
  (COND
    ((NULL? list1) list2)
    (ELSE (CONS (CAR list1) (append (CDR list1) list2)))
))
```

COND 的第一种情况用来在第一个参数表为空时终止递归，返回第二个表。在第二种情况下(ELSE)，第一个参数表的 CAR 被 CONS 到递归调用的结果上，该调用以第一个表的 CDR 作为第一个参数。

---

7. 与 member 一样，用户通常不能定义 append 函数。

考虑以下 Scheme 函数 guess，它使用本节介绍的 member 函数。在阅读后面的说明之前，请研究确定该函数的功能是什么。假设参数都是简单表。

```
(DEFINE (guess list1 list2)
  (COND
    ((NULL? list1) '())
    ((member (CAR list1) list2)
             (CONS (CAR list1) (guess (CDR list1) list2)))
    (ELSE (guess (CDR list1) list2))
))
```

guess 生成一个简单表，包含两个参数表的公共元素。因此，如果参数表表示集合，guess 计算的表就表示这两个集合的交集。

15.5.11 LET

LET 是一个函数(参见第 5 章)，它创建了一个局部作用域，在其中将名字暂时绑定到表达式的值上，通常用于从复杂的表达式中分解出公共的子表达式。这些名字随后可用于对另一个表达式的求值，但它们不能在 LET 中重新绑定到新值上。以下例子展示了 LET 的用法，它计算一个二次方程的根(假设根为实数[8])。二次方程 $ax^2 + bx + c$ 的实数根(不是复数根)的数学定义是：根 1= $(-b + \sqrt{b^2 - 4ac})/2a$，根 2=$(-b - \sqrt{b^2 - 4ac})/2a$。

```
(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
                (/ (SQRT (- (* b b) (* 4 a c))) (* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))
        )
(LIST (+ minus_b_over_2a root_part_over_2a)
          (- minus_b_over_2a root_part_over_2a))
))
```

这个例子使用 LIST 来创建一个包含两个元素的表作为结果。

因为 LET 结构中第一部分绑定的名字不能在后面的表达式中改变，所以它们不同于命令式语言中程序块的局部变量。

LET 实际上只是应用于参数的 λ 表达式的速记写法。以下两个表达式是等价的：

```
(LET ((alpha 7))(* 5 alpha))
((LAMBDA (alpha) (* 5 alpha)) 7)
```

在第一个表达式中，用 LET 把 7 绑定到 alpha 上；在第二个表达式中，通过 LAMBDA 表达式的参数把 7 绑定到 alpha 上。

---

8. 有些版本的 Scheme 包含复数这一数据类型，不管方程的根是实数还是复数，都可以计算。

15.5.12　Scheme 中的尾递归

如果函数递归调用是函数中最后的操作，该函数就称为尾递归。这表示递归调用的返回值就是函数非递归调用的返回值。例如 15.5.10 节中的 member 函数就是尾递归。

```
(DEFINE (member atm a_list)
  (COND
    ((NULL? a_list) #F)
    ((EQ? atm (CAR a_list)) #T)
    (ELSE (member atm (CDR a_list)))
))
```

这个函数可以由编译器自动地转换为迭代，从而比递归形式执行得更快。

然而，许多使用递归来进行重复计算的函数都不是尾递归。关注效率的程序员发现了一些方法，将某些函数重写成尾递归，其中一个例子是使用累积参数和辅助函数。作为这种方法的一个例子，考虑 15.5.7 节中的阶乘函数，即

```
(DEFINE (factorial n)
  (IF (<= n 1)
    1
    (* n (factorial (- n 1)))
))
```

该函数的最后一个操作是乘法，工作原理是先创建要乘在一起的数列，然后做乘法，得到结果。每个数值都要执行函数才能创建，且都存储在一个活动记录实例中。递归展开时，把数值乘在一起。第 9 章在递归调用几次阶乘函数后，显示了栈的内容。这个函数可以用一个辅助函数来重写，该辅助函数使用一个参数来累积部分的阶乘。这个尾递归的辅助函数，也具有 factorial 的参数。两个函数如下：

```
(DEFINE (facthelper n factpartial)
  (IF (<= n 1)
    factpartial
    (facthelper (- n 1) (* n factpartial))
))
(DEFINE (factorial n)
   (facthelper n 1)
)
```

使用这些函数，会在递归调用的过程中计算出结果，而不是在递归展开时计算。因为在活动记录实例中没有有用的信息，所以不需要这些实例。无论请求多少个递归调用，都只需要一个活动记录实例。这使尾递归版本远比非尾递归版本更高效。

Scheme 语言的定义要求 Scheme 语言处理系统转换所有尾递归的函数，用迭代来代替递归。因此至少出于效率的原因，将用递归表示重复执行的函数定义成尾递归是很重要的。某些函数式语言的优化编译器甚至可以将一些非尾递归的函数转化为等价的尾递归函数，再用迭代来代替递归实现重复执行，进行函数编码。

15.5.13 函数形式

本节介绍 Scheme 提供的两个常用的数学函数形式：函数组合和 apply-to-all。这两个函数形式在 15.2.2 节中都有数学的定义。

15.5.13.1 函数组合

函数组合是最初的 LISP 提供的唯一基本的函数形式。所有后来的 LISP 方言，包括 Scheme，也提供了这个函数形式。如 15.2.2 所述，函数组合带两个函数参数，返回一个函数，该函数先将第二个参数函数应用于其参数，再将第一个参数函数应用于第二个参数函数的返回值。换言之，如果 h(x)=f(g(x))，则函数 h 是函数 f 和 g 的组合。例如如下例子：

```
(DEFINE (g x) (* 3 x))
(DEFINE (f x) (+ 2 x))
```

现在 f 和 g 的函数组合可以写为：

```
(DEFINE (h x) (+ 2 (* 3 x)))
```

在 Scheme 中，函数组合 compose 可以写为：

```
(DEFINE (compose f g) (LAMBDA (x)(f (g x))))
```

例如：

```
((compose CAR CDR) '((a b) c d))
```

这个调用会输出 c。这是 CADR 的另一种效率较低的形式。现在考虑对 compose 的另一个调用：

```
((compose CDR CAR) '((a b) c d))
```

这个调用会输出(b)。这是 CDAR 的另一种形式。

使用 compose 的另一个例子如下：

```
(DEFINE (third a_list)
  ((compose CAR (compose CDR CDR)) a_list))
```

这是 CADDR 的另一种形式。

15.5.13.2 apply-to-all 函数形式

在函数式程序设计语言中，最常用的函数形式是各种数学上的 apply-to-all 函数形式。其中最简单的就是 map，它有两个参数：一个函数和一个表。map 将给定的函数应用到给定表的每个元素上，返回这些应用的结果列表。Scheme 定义的 map 如下[9]：

---

9. 与 member 一样，map 是不能由用户重定义的预定义函数。

```
(DEFINE (map fun a_list)
  (COND
    ((NULL? a_list) '())
    (ELSE (CONS (fun (CAR a_list))
                (map fun (CDR a_list))))
))
```

请注意，map 的简单形式表示了一个复杂的函数形式。

作为使用 map 的一个例子，假设要计算表中所有元素的立方。其实现如下：

```
(map (LAMBDA (num) (* num num num)) '(3 4 2 6))
```

该函数调用返回(27 64 8 216)。

请注意，在这个例子中，map 的第一个参数是一个 LAMBDA 表达式。当 EVAL 计算 LAMBDA 表达式时，会创建一个具有与预定义函数相同形式但没有名字的函数。在这个示例的表达式中，这个匿名函数立即应用到参数表的每个元素上，结果以表的形式返回。

15.5.14　构建代码的函数

程序与数据具有相同的结构这一事实，可以用于构建程序。Scheme 解释器使用了 EVAL 函数。Scheme 系统将 EVAL 应用于每一个输入的表达式，无论表达式是在交互式解释器中由 Scheme 提示输入的，还是要解释的程序中的某部分。EVAL 函数也可以由 Scheme 程序直接调用，这为 Scheme 程序创建表达式并调用 EVAL 来计算它们提供了可能。这不是 Scheme 独有的，但其表达式的简单形式更便于在执行过程中创建它们。

以上处理过程最简单的例子之一，用到了数值原子。Scheme 包含+函数，它以任意数目的数值原子为参数，返回它们的和。例如，(+ 3 7 10 2)返回 22。

问题如下：假设在一个程序中，有一个数值原子的表，要计算它们的和。不能直接将+应用到表上，因为+只能以原子作为参数，而不是数值原子的表。当然，可以写一个函数，用递归来遍历表，反复地将表的 CAR 加到表的 CDR 的和上。这样的函数如下：

```
(DEFINE (adder a_list)
  (COND
    ((NULL? a_list) 0)
    (ELSE (+ (CAR a_list) (adder (CDR a_list))))
))
```

下面是 adder 的示例调用，以及递归调用和返回值：

```
(adder '(3 4 5))
(+ 3 (adder (4 5)))
(+ 3 (+ 4 (adder (5))))
(+ 3 (+ 4 (+ 5 (adder ()))))
(+ 3 (+ 4 (+ 5 0)))
(+ 3 (+ 4 5))
(+ 3 9)
(12)
```

这一问题的另一个解决方案是写一个函数，该函数创建一个用正确参数形式对+的调用。为此，可以用 CONS 建立一个新表，这个新表与参数表相同，但将原子+插入到表的开头。然后将新的表提交给 EVAL，进行计算，如下：

```
(DEFINE (adder a_list)
  (COND
    ((NULL? a_list) 0)
    (ELSE (EVAL (CONS '+ a_list)))
))
```

请注意，+函数的名字加了引号，以防止 EVAL 在计算 CONS 时计算它。下面的示例调用了 adder 的这个新版本，还调用了 EVAL，显示了返回值：

```
(adder '(3 4 5))
(EVAL (+ 3 4 5)
(12)
```

在所有早期版本的 Scheme 中，EVAL 函数在程序最外层的作用域中计算表达式。Scheme 的最新版本 Scheme 4，需要 EVAL 的第二个参数来指定要计算的表达式的作用域。出于简化的原因，例子中去掉了作用域参数，这里也不介绍作用域名。

15.6 Common LISP

在将 20 世纪 80 年代早期一些 LISP 方言(包括 Scheme)的特性组合成一种语言的工作中，产生了 Common LISP(Steele，1990)。作为若干语言的联合体，它相当大而且复杂，类似于 C++和 C#，不过其基础是最初的 LISP，因此它的语法、基本函数和基本特性都来自最初的 LISP。

下面是用 Common LISP 编写的阶乘函数：

```
(DEFUN factorial (x)
  (IF (<= n 1)
    1
    (* n factorial (- n 1)))
))
```

这个函数只有第一行在语法上不同于其 Scheme 版本。

Common LISP 有一长串的特性：大量数据类型和结构，包括记录、数组、复数和字符串；强大的输入和输出操作；一种将成组的函数和数据模块化，并提供访问控制的包。Common LISP 包含几个命令式结构和一些可变类型。

认识到动态作用域提供的灵活性，以及静态作用域的简单性，Common LISP 允许这两种作用域。变量的默认作用域是静态的，但将变量声明为 special，就可以让变量变成是动态作用域的。

宏在 Common LISP 中常常用于扩展该语言。实际上，一些预定义的函数就是宏。例

如 DOLIST 就是一个宏，它带两个参数，分别是变量和表。考虑下面的例子：

```
(DOLIST (x '(1 2 3)) (print x))
```

它输出如下：

```
1
2
3

NIL
```

这里的 NIL 是 DOLIST 的返回值。

宏用两个步骤达到其效果。第一，扩展宏。第二，计算所扩展的宏，它其实是 LISP 代码。用户可以用 DEFMACRO 定义自己的宏。

Common LISP 的反引号操作符(`)类似于 Scheme 的 QUOTE，但一些参数可以不加引号，而是在它们的前面加上逗号。例如，考虑下面的两个例子：

```
`(a (* 3 4) c)
```

这个表达式计算得(a (* 3 4) c)。但下面的表达式

```
`(a ,(* 3 4) c)
```

计算得(a 12 c)。

LISP 实现有一个前端，叫做阅读器(reader)，它将 LISP 程序的文本转换为代码表示。接着代码表示中的宏调用扩展为代码表示。这一步的输出解释或编译为主机的机器语言，或者编译为中间代码。有一种特殊的宏，称为阅读器宏或阅读宏，它们在 LISP 语言处理器的阅读阶段扩展。阅读器宏把特定的字符扩展为一串 LISP 代码。例如，LISP 中的撇号是一个阅读宏，它扩展为对 QUOTE 的调用。用户可以定义自己的阅读器宏，来创建其他简洁结构。

Common LISP 和其他基于 LISP 的语言都有符号数据类型。保留字是求值为其自身的符号，T 和 NIL 也是这样。在技术上，符号要么绑定，要么不绑定。在计算函数时，参数符号是绑定的。另外，还要绑定本身是命令式变量名且已赋值的符号。其他符号是不绑定的。例如，考虑下面的表达式：

```
(LIST '(A B C))
```

符号 A、B 和 C 是不绑定的。Ruby 也有符号数据类型。

从某种意义上说，Scheme 和 Common LISP 是相反的。Scheme 小巧得多，语义也简单，部分原因是因为它只使用静态作用域，也因为设计它是为了用于讲授程序设计，而 Common LISP 则计划成为一种商业语言。Common Lisp 已成功地成为一种在人工智能应用领域广泛使用的语言。另一方面，Scheme 更多地在大学课程中用于函数式程序设计，更可能作为一种函数式语言来学习，因为它的规模相对较小。Common LISP 成为一种大规模语言的一个

重要设计方针，就是希望它兼容所派生的 LISP 方言。

Common LISP Object System (CLOS) (Paepeke, 1993)在 20 世纪 80 年代晚期开发为 Common LISP 的面向对象版本。这种语言支持泛型函数和多继承。

15.7 ML

ML(Milner 等，1990)与 Scheme 一样，是一种静态作用域的函数式程序设计语言。但是它在一些重要的方面，与 LISP 或包含 Scheme 在内的 LISP 方言不同。最重要的差别在于，ML 是一种强类型语言，而 Scheme 本质上是没有类型的。ML 要对函数的参数和返回值进行类型声明，虽然由于有类型推断，类型声明并不常用。每个变量和表达式的类型可以静态确定。ML 与其他函数式编程语言一样，并没有命令式语言中的变量。ML 有标识符，虽然标识符具有命令式语言中变量名的外观，但最好看作值的名字。一旦设置了标识符，就不能改变。它们类似于命令式语言中的命名常量，像 Java 中的 final 声明。ML 标识符没有固定的类型，任何标识符都可以是任意类型的数值名。

求值环境表存储了程序中所有隐式和显式声明的标识符及其类型。它类似于运行时的符号表。隐式或显式声明标识符时，标识符会放在求值环境中。

Scheme 和 ML 间另一个重要的差别是，ML 使用的语法比 LISP 更接近于命令式语言。例如，用 ML 编写的算术表达式使用中缀表示法。

ML 中的函数声明具有如下一般形式：

```
fun 函数名(形参)=表达式;
```

调用函数时，函数会返回表达式的值。实际上，函数可以包含一列表达式，中间用分号隔开，并放在圆括号中。此时的返回值是最后一个表达式的值。当然，除非表达式有副作用，否则最后一个表达式之前的所有表达式都不起作用。这里不考虑有副作用的 ML 部分，只考虑带一个表达式的函数定义。

现在讨论类型推断。考虑下面的 ML 函数声明：

```
fun circumf(r) = 3.14159 * r * r;
```

这指定了一个函数 circumf，它带一个浮点(在 ML 中是 real)参数，输出一个浮点结果。类型从表达式中字面量的类型中推断。同样，函数

```
fun times10(x) = 10 * x;
```

参数和函数值的类型都推断为 int。

考虑下面的 ML 函数：

```
fun square(x) = x * x;
```

ML 根据函数定义中的*运算符来推断参数和返回值的类型。由于*是算术运算符，所

以参数和函数的类型都假定为数值。在 ML 中，默认的数值类型是 int。所以推断 square 中参数和返回值的类型是 int。

如果用浮点值调用 square，如下：

```
square(2.75);
```

就会出错，因为 ML 不会把 real 值强制转换为 int 类型。如果 square 希望接受 real 参数，就要把它重写为：

```
fun square(x) : real = x * x;
```

因为 ML 不允许重载函数，所以这个版本不能与前面的 int 版本共存。最后定义的版本将是唯一的版本。

知道函数值的类型是 real，就足以推断出参数的类型也是 real。下面的函数定义都是合法的：

```
fun square(x : real) = x * x;
fun square(x) = (x : real) * x;
fun square(x) = x * (x : real);
```

类型推断也用于函数式语言 Miranda、Haskell 和 F#。

ML 的选择控制流结构与命令式语言类似，具有以下一般形式：

```
if 表达式 then then 表达式 else else 表达式
```

其中第一个表达式的结果必须是布尔值。

Scheme 的条件表达式可以出现在 ML 的函数定义层次上。在 Scheme 中，COND 函数用于确定给定参数的值，这个值又指定了 COND 的返回值。在 ML 中，函数执行的计算可以根据给定参数的不同形式来定义。这一特性是为了模仿数学中条件函数定义的形式与意义。在 ML 中，定义函数返回值的特定表达式是根据与给定参数的模式匹配来选择的。例如，若不用模式匹配，计算阶乘的函数可以写成：

```
fun fact(n : int): int = if n <= 1 then 1
                         else n * fact(n - 1);
```

参数的模式匹配可以用于编写一个函数的多个定义。取决于参数形式的不同函数定义用 OR 符号(|)分隔。例如，使用模式匹配，阶乘函数可以写成：

```
fun fact(0) = 1
|   fact(1) = 1
|  fact(n : int): int = n * fact(n - 1);
```

如果 fact 以实参 0 来调用，就使用第一个定义，如果实参是 1，就使用第二个定义。如果发送一个不是 0 和 1 的 int 值，就使用第三个定义。

如第 6 章所述，ML 支持表和表操作，ML 中的 hd、tl 和::对应 Scheme 的 CAR、CDR 和 CONS。

由于 ML 具有模式化的函数参数，hd 和 tl 函数在 ML 中的使用频率远远低于 Scheme 中的 CAR 和 CDR。例如，在形参中，表达式

```
(h :: t)
```

实际上是两个形参，是给定表参数的头和尾，而相应的单个实参是表。例如，给定表中元素的数量可以用以下函数计算：

```
fun length([]) = 0
|   length(h :: t) = 1 + length(t);
```

作为这一概念的另一个例子，考虑 append 函数，它与 Scheme 的 append 函数功能相同：

```
fun append([], lis2) = lis2
|   append(h :: t, lis2) = h :: append(t, lis2);
```

函数中的第一种情况，处理以空表为第一个参数来调用函数的情况，当最初调用的第一个参数非空时，这种情况会终止递归。函数的第二种情况将第一个参数表拆成头和尾(hd 和 tl)，头被 CONS 到递归调用的结果上，而递归调用以尾作为第一个参数。

在 ML 中，名字可以用值声明语句绑定到值上，形式是

```
val 新名字=表达式;
```

例如：

```
val distance=time * speed;
```

不要认为这个语句就像命令式语言的赋值语句一样，因为它不是这样。val 语句将一个名字绑定到值上，但这个名字以后不能再重新绑定到新的值上。其实从某种意义上说是可以的。如果用另一个 val 语句来重新绑定一个名字，就会在求值环境中产生一个与前一名字无关的新项。实际上，在新的绑定之后，旧的求值环境项(前一个绑定的)就不可见了。新绑定的类型不必与前一个绑定的类型相同。val 语句没有副作用，只是将一个名字添加到当前求值环境中，并为它绑定一个值。

val 通常用于 let 表达式中[10]，考虑下面的例子：

```
let
  val radius = 2.7
  val pi = 3.14159
in
  pi * radius * radius
end;
```

ML 包含几个高阶函数，它们一般用于函数式程序设计。其中一个是表的过滤函数 filter，它带一个谓词函数参数，该谓词函数常常给定为 λ 表达式，在 ML 中，λ 表达式的

10. let 表达式参见第 5 章。

定义类似于函数，但使用 fn 保留字，而不是 fun，当然，λ 表达式是匿名的。filter 返回一个函数，该函数带一个表参数，它用谓词测试表中的每个元素，并将谓词返回真的每个元素添加到新表中，这个新表就是函数的返回值。考虑下面使用 filter 的例子：

```
filter(fn(x) => x < 100, [25, 1, 50, 711, 100, 150, 27,
    161, 3]);
```

这个应用程序返回[25, 1, 50, 27, 3]。

map 函数带一个函数参数，返回一个函数，该返回函数带一个表参数。map 将函数参数应用于表的每个元素，返回这些应用结果的表。考虑下面的代码：

```
fun cube x = x * x * x;
val cubeList = map cube;
val newList = cubeList [1, 3, 5];
```

执行后，newList 的值是[1, 27, 125]。将 cube 函数定义为 λ 表达式会更简单，如下：

```
val newList = map (fn x => x * x * x, [1, 3, 5]);
```

ML 比较两个函数的二元操作符是 o(小写的"oh")。例如，要建立函数 h，它先应用函数 f，再对 f 的返回值应用函数 g，就可以使用如下代码：

```
val h = g o f;
```

严格地说，ML 函数带一个参数。定义带多个参数的函数时，ML 会把参数看作一个元组，但界定元组值的圆括号是可选的。分隔参数(元组元素)的逗号是必须有的。

currying 的过程是用单参函数替代多参函数，其中单参函数返回一个函数，它带多参函数的其他参数。

ML 多参函数可以用 curry 形式定义，即省略参数之间的逗号(和界定的圆括号)[11]。例如下面的代码：

```
fun add a b = a + b;
```

尽管这看起来是定义了带两个参数的函数，但实际上是定义了一个单参函数。add 函数带一个整数参数 a，返回一个函数，该函数带一个整数参数 b。调用这个函数时，也要去掉参数之间的逗号，如下所示：

```
add 3 5;
```

这个调用返回 8，这正是期望的结果。

curried 函数很有趣，也很有用，因为新函数是通过部分求值从 curried 函数中构建的。部分求值表示，用左边的一个或多个形参所对应的实参对函数求值。例如，可以定义一个新函数：

---

11. 这种函数形式以 Haskell Curry 命名，Haskell Curry 是研究它们的英国数学家。

```
fun add5 x = add 5 x;
```

add5 函数带有实参 5，它把 5 作为第一个形参的值来计算 add 函数，返回一个函数，该函数把 5 添加为它的唯一参数，如下所示：

```
val num = add5 10;
```

num 的值现在是 15。可以从 curried 函数中创建任意多个新函数，将任意数值加到给定的参数上。

curried 函数还可以用 Scheme、Haskell 和 F#编写。考虑下面的 Scheme 函数：

```
(DEFINE (add x y) (+ x y))
```

这个函数的 curried 版本如下：

```
(DEFINE (add y) (LAMBDA (x) (+ y x)))
```

其调用如下：

```
((add 3) 4)
```

ML 也有枚举类型、数组和元组。ML 有异常处理，以及实现抽象数据类型的模块化机制。

ML 对程序设计语言的发展具有重要影响。对于语言研究人员来说，ML 已经成为最精心设计的语言之一，而且它产生了一些后续语言，例如 Haskell、Caml、OCaml 和 F#。

15.8 Haskell

Haskell(Thompson，1999)与 ML 很相似，它使用相似的语法，使用静态作用域，是强类型的，使用相同的类型推断方法。Haskell 有三个特性使它不同于 ML。第一，Haskell 中的函数可以重载(ML 中的函数不可以)。第二，在 Haskell 中使用不严格的语义，而在 ML(以及大多数其他程序设计语言)中都使用严格的语义。第三，Haskell 是一种纯函数式编程语言，即它没有带副作用的表达式或语句，而 ML 允许有某些副作用(例如，ML 有可变数组)。本节后面会深入介绍不严格的语义和函数重载。

本节的代码是用 1.4 版的 Haskell 写的。

考虑以下阶乘函数的定义，它使用了参数的模式匹配：

```
fact 0 = 1
fact 1 = 1
fact n = n * fact (n - 1)
```

请注意这一定义与 15.7 节中 ML 版本的定义在语法上的差异。第一，函数定义不是以保留字开始的(在 ML 中是 fun)；第二，不同形参的函数的不同定义其外观是一样的。

使用模式匹配，可以定义一个函数，来计算第 n 个斐波纳契数，如下：

```
fib 0 = 1
```

```
fib 1 = 1
fib (n + 2) = fib (n + 1) + fib n
```

可以在函数定义的行中加入守卫，表示定义可适用的环境，例如：

```
fact n
  | n == 0 = 1
  | n == 1 = 1
  | n > 1 = n * fact(n - 1)
```

这个阶乘的定义比前面的更精确，因为它限制实参的取值范围为可处理的范围。这种函数定义的形式称为条件表达式，来自于它所基于的数学表达式。

在条件表达式中可以有一个 otherwise 作为最后的条件，其语义是显然的。例如：

```
sub n
  | n < 10 = 0
  | n > 100 = 2
  | otherwise = 1
```

请注意这里的守卫与第 8 章介绍的守卫命令的相似之处。

考虑以下函数定义，其用途与 15.7 节中相应的 ML 函数相同：

```
square x = x * x
```

不过，这里由于 Haskell 支持多态，这个函数可以接受任意数值类型的参数。

与 ML 一样，表在 Haskell 中也写在方括号内，如：

```
colors = ["blue", "green", "red", "yellow"]
```

Haskell 包含一组表操作符。例如，表可以用++连接，:用作 CONS 的中缀版本，..用来表示表中的算术序列。例如：

```
5: [2, 7, 9]得到[5, 2, 7, 9]
[1, 3..11]得到[1, 3, 5, 7, 9, 11]
[1, 3, 5]++[2, 4, 6]得到[1, 3, 5, 2, 4, 6]
```

请注意，:运算符就像是 ML 的::运算符[12]。使用:和模式匹配，可以定义一个简单的函数来计算给定数列的乘积：

```
product [] = 1
product (a:x) = a * product x
```

使用 product，可以用更简单的形式写出阶乘函数：

```
fact n = product [1..n]
```

---

12. 有趣的是，ML 用:来为名字附加类型名，用::表示 CONS，而 Haskell 以完全相反的方式使用这两个运算符。

Haskell 包含了一个 let 结构，与 ML 的 let 和 val 相似。例如，可以写：

```
quadratic_root a b c =
    let
     minus_b_over_2a = - b / (2.0 * a)
     root_part_over_2a =
                sqrt(b ^ 2 - 4.0 * a * c) / (2.0 * a)
    in
     minus_b_over_2a - root_part_over_2a,
     minus_b_over_2a + root_part_over_2a
```

第 6 章介绍了 Haskell 的列表包含。例如下面的代码：

```
[n * n * n | n <- [1..50]]
```

这定义了 1～50 的数的立方表，读作“一个 n*n*n 的表，其中 n 取自 1～50 范围内的数”。在这个例子中，限定词是一个生成器的形式，生成 1～50 的数。在其他情况下，限定词可以是布尔表达式的形式，这时称为测试。这一表示方法可以用来描述做许多事情的算法，例如寻找表的全排列和表排序。考虑以下函数，当给定数值 n 时返回所有因子的列表：

```
factors n = [ i | i <- [1..n 'div' 2], n 'mod' i == 0]
```

factors 中的列表包含创建了一列数，每个数都暂时绑定到名字 i，范围是从 1 到 n/2，且 n `mod` i 是零。这实际上是定义给定数的因子的一个非常严格和简短的定义。在 div 和 mod 两边的反引号表示以中缀形式使用这些函数。当它们以函数表示法来调用时，如 div n 2，不使用反引号。

接下来考虑在以下快速排序算法的实现中，Haskell 所展现出的简洁性：

```
sort [] = []
sort (h:t) = sort [b | b <- t, b <- h]
             ++ [h] ++
             sort [b | b <- t, b > h]
```

在这个程序中，小于或等于表头的元素集合在排序后，与表头元素连接，然后大于表头的元素集合在排序后，连接到前面的结果上。这一快速排序的定义比任何命令式语言编码的相同算法都要短小和简单得多。

如果程序设计语言要求对所有实参求值，该语言就是严格的，这确保函数的值不会取决于参数的求值顺序。如果语言不满足严格的要求，该语言就是非严格的。非严格的语言与严格的语言相比，有一些独特的优点。第一，非严格的语言通常更高效，因为可以避免一些求值计算[13]。第二，一些有趣的功能在非严格的语言中是可能的，而在严格的语言中不可能，其中包括无限表。非严格的语言可以使用一种称为懒惰求值(lazy evaluation)的求值形式，意思是只有当需要表达式的值时，才对它们求值。

13. 请注意，这与一些命令式语言中布尔表达式的短路求值有关。

在 Scheme 中，函数的参数在函数调用前要全部求值，因此它是严格语义的。懒惰求值意味着实参只有当需要它的值来计算函数时才求值，因此如果函数有两个参数，但是在一次特定的函数执行过程中没有用到第一个参数，那么这次执行所传递的实参就不会求值。而且，如果函数的执行只需要对实参的一部分求值，那么其余部分也不求值。最后，即使在函数调用中，同一个实际参数出现多次，该实参也只求值一次。

如前所述，懒惰求值允许定义无限的数据结构。例如以下程序：

```
positives = [0..]
evens = [2, 4..]
squares = [n * n | n <- [0..]]
```

显然没有计算机可以实际表示这些表中所有的数，但如果使用懒惰求值，并不妨碍使用这些表。例如，如果要知道某个数是否是完全平方数，就可以用一个成员资格函数来检查平方表。假设有一个谓词函数 member，可判定给定表中是否包含给定原子，就可以使用如下代码：

```
member 16 squares
```

它返回 True。squares 的定义会计算，直到找到 16。member 函数需要小心地编写。具体而言，假设它的定义如下：

```
member b [] = False
member b (a:x)= (a == b) || member b x
```

这个定义的第二行将第一个参数拆成头和尾，如果头匹配查找的元素 b，或者表尾的递归调用返回 True，它的返回值就为真。

只有给定的数是完全平方数，这一 member 的定义才会配合 squares 正确工作。如果给定数不是完全平方数，squares 就会永远不停地生成平方数，在表中查找指定的数，直到达到内存上限。以下函数执行有序表的成员资格测试，如果找到了一个大于待查找数的数，就放弃查找，并返回 False[14]。

```
member2 n (m:x)
  | m < n = member2 n x
  | m == n = True
  | otherwise = False
```

懒惰求值有时是一个模块化工具。假设在程序中有一个对函数 f 的调用，f 的参数是函数 g 的返回值[15]。因此，我们有 f(g(x))。进一步假设 g 生成大量数据，每次少量，然后 f 必须处理这些数据，每次少量。在传统的命令式语言中，g 运行在全部输入上，生成了一个输出文件。接着 f 把该文件用作其输入来运行。这个方法需要读写文件的时间和文件的

14. 这里假定表按升序排列。

15. 这个例子在 Hughes(1989)中。

存储空间。若使用懒惰求值，f 和 g 的执行就会隐式地同步。函数 g 只执行足够长的时间，生成足够 f 启动处理过程的数据。当 f 准备好处理更多数据时，g 可以重新开始生成更多数据，而 f 等待。如果 f 没有得到 g 的所有输出就结束了，g 就中止，从而避免无用的计算。而且，g 不必是一个会结束的函数(可能因为它生成无限数目的输出)，当 f 结束时 g 会强制结束。因此在懒惰求值的情况下，g 的运行尽可能少。这一求值过程支持将程序模块化为生成器单元和选择器单元，其中生成器产生大量可能的结果，选择器选择适当的子集。

懒惰求值不是没有代价的。如果这种表达能力和灵活性是免费的，肯定会令人惊讶。在这种情况下，代价是语义复杂了许多，结果是执行速度慢得多。

15.9 F#

F#是一种.NET 函数编程语言，其核心基于 OCaml，OCaml 是 ML 和 Haskell 的衍生物。尽管它基本上是一种函数式语言，但还包括命令特征，支持面向对象编程。F#最重要的一个特点是它具备功能完整的 IDE，这是一个支持命令式、面向对象式和函数式编程的全面的实用工具库。F#，还具备与许多非函数式语言(所有.NET 语言)的交互操作功能。

F#是一种一流的.NET 语言，这意味着 F#程序可以用所有的方式与其他.NET 语言交互操作。例如，F#类可以由其他语言编写的程序使用和派生子类，反之亦然。而且，F#程序可以访问所有的.NET Framework API。F#实现代码可以在 Microsoft (http://research.microsoft.com/fsharp/fsharp.aspx)上免费获得，它还得到 Visual Studio 的支持。

F#包括许多数据类型，其中有元组(类似于 Python、ML 与 Haskell 等函数式语言中的元组)、列表、识别联合(ML 联合的一个扩展)和记录(类似于 ML 中的记录，它也类似于元组，但组件是命名了的)。F#还有可变和不可变数组。

如第 6 章所述，F#的表类似于 ML 中的表，但元素用分号分隔开，且 hd 和 tl 必须调用为 List 的方法。

F#支持序列值，它们是.NET 名称空间 System.Collections.Generic.IEnumerable 中的类型。在 F#中，序列缩写为 seq<type>，其中<type>表示泛型的类型。例如，类型 seq<int>是一系列整数值。序列值可以用生成器创建，它们可以迭代。最简单的序列用范围表达式创建，如下所示：

```
let x = seq {1..4};;
```

在 F#的例子中，假定使用了交互式解释器，它需要在每个语句的末尾加上两个分号。这个表达式生成了 seq [1; 2; 3; 4](表和序列元素用分号分隔开)。序列的生成是懒惰的，例如，下面的语句将 y 定义为一个非常长的序列，但只生成了必要的元素。为了便于显示，只生成了前 4 个元素：

```
let y = seq {0..100000000};;
y;;
val it: seq<int> = seq[0; 1; 2; 3;...]
```

上述第一行定义了 y，第二行指定显示 y 的值，第三行是 F#交互式解释器的输出。

整数序列定义的默认步长是 1，但在范围规范的中间指定步长，就可以设置步长，如下所示：

```
seq {1..2..7};;
```

这会生成 seq [1; 3; 5; 7]。

序列的值可以用 for-in 结构来迭代，如下所示：

```
let seq1= seq {0..3..11};;
for value in seq1 do printfn "value = %d" value;;
```

输出如下：

```
value = 0
value = 3
value = 6
value = 9
```

迭代器还可以用于创建序列，如下所示：

```
let cubes = seq {for i in 1..5 -> (i, i * i * i)};;
```

生成如下元组表：

```
seq [(1, 1); (2, 8); (3, 27); (4, 64); (5, 125)]
```

这种使用迭代器生成集合的方式是列表包含的一种形式。

序列可以用于生成列表和数组，但在这种情况下，生成过程不是懒惰的。实际上，在 F#中，列表和序列的主要区别是，序列是懒惰的，因此可以是无限的；而列表不是懒惰的，列表会完全存储在内存中，而序列不是这样。

F#的函数类似于 ML 和 Haskell。如果指定了函数的名称，函数就用 let 语句定义。如果函数未指定名称，在技术上这意味着它们是 λ 表达式，所以用 fun 保留字定义。下面的 λ 表达式说明了其语法：

```
(fun a b -> a / b)
```

注意，用 let 定义的名字和没有用 let 定义的无参函数没有任何区别。

使用缩进是为了显示函数定义的范围。例如下面的函数定义：

```
let f =
    let pi = 3.14159
    let twoPi = 2.0 * pi
twoPi;;
```

注意 F#与 ML 一样，也没有强制数值转换，所以如果这个函数在倒数第二行使用了 2，而不是 2.0，就会报告出错。

如果函数是递归的，在其定义中就必须把保留字 rec 放在该函数名的前面。下面是阶

乘的 F#版本：

```
let rec factorial x =
    if x <= 1 then 1
    else n * factorial(n - 1)
```

在函数中定义的名称可以超出作用域，这意味着它们可以重新定义，结束以前的作用域。例如，可以编写如下语句：

```
let x4 x =
    let x = x * x
    let x = x * x
    x;;
```

在这个函数中，x4 函数体的第一个 let 创建了 x 的一个新版本，把它定义为参数 x 的平方。这会停止参数的作用域。所以函数体的第二个 let 在它的右边使用新的 x，创建了 x 的另一个版本，因此停止了前一个 let 中创建的 x 的作用域。

F#中有两个重要的函数操作符：管道(|>)和函数组合(>>)。管道操作符是一个二元操作符，它将左操作数(这是一个表达式)的值传送给函数调用(右操作数)的最后一个参数。该操作符用于链接函数调用，并将要处理的数据传送给每个调用。在下面的示例代码中，使用了高阶函数 filter 和 map：

```
let myNums = [1; 2; 3; 4; 5]
let evensTimesFive = myNums
    |> List.filter (fun n -> n % 2 = 0)
    |> List.map (fun n -> 5 * n)
```

evensTimesFive 函数从列表 myNums 开始，用 filter 过滤掉不是偶数的数，再用 map 映射一个 λ 表达式，该表达式为给定列表中的数乘以 5。evensTimesFive 的返回值是[10; 20]。

函数组合操作符会建立一个函数，把其左操作数(这是一个函数)应用于一个给定的参数，再把从该函数返回的结果传送给其右操作数(这也是一个函数)。所以 F#表达式(f >> g) x 等价于数学表达式 g(f(x))。

与 ML 一样，F#支持 curried 函数和局部计算。15.7 节中的 ML 示例可以用 F#重写为：

```
let add a b = a + b;;
let add5 = add 5;;
```

注意，与 ML 不同，F#中形参列表的语法对所有的函数都是相同的，所以所有多参的函数都可以 curried。

F#很有趣有几个原因：第一，它基于以前的函数式语言。第二，它支持目前广泛应用的几乎所有编程方法。第三，它是第一个为与其他广泛使用的语言交互操作而设计的函数式语言。第四，它一开始就具备了精心开发出来的 IDE 和用.NET 及其框架开发的实用软件库。

15.10　基本命令式语言对函数式编程的支持

命令式编程语言总是只提供了对函数式编程的有限支持。这种有限支持导致这些语言很少应用于函数式编程。过去，命令式语言中与函数式编程相关的最重要的限制是缺乏对高阶函数的支持。

人们对使用函数式编程的兴趣有所提高的一个标志是在编程语言的过去 10 年中，基本命令式语言开始提供对函数式编程的部分支持。例如，匿名函数(类似于 λ 表达式)现在是 JavaScript、Python、Ruby 和 C#的一部分。

在 JavaScript 中，命名的函数用如下语法定义：

```
function 函数名 (形参) {
 函数体
}
```

在 JavaScript 中，匿名函数的定义也使用相同的语法，只是忽略了函数的名称。

C#支持 λ 表达式，但其语法不同于 C#函数。例如下面的语句：

```
i => (i % 2) == 0
```

这个 λ 表达式根据给定参数 i 是偶数(true)还是奇数(false)，返回一个布尔值。C#的 λ 表达式可以有多个参数和多个语句。

Python 的 λ 表达式定义了简单的单语句匿名函数，它可以有多个参数。Python 中 λ 表达式的语法如下所示：

```
lambda a, b : 2 * a - b
```

注意形参用冒号与函数体分隔开。

Python 包含高阶函数 filter 和 map。这两个函数都把 λ 表达式用作其第一个参数。第二个参数是一个序列类型，它们都把相同的序列类型返回为第二个参数。在 Python 中，字符串、列表和元组都看作序列。下面的例子使用了 Python 中的 map 函数：

```
map(lambda x: x ** 3, [2, 4, 6, 8])
```

这个调用返回[8, 64, 216, 512]。

Python 还支持部分函数应用。考虑下面的例子：

```
from operator import add
add5 = partial (add, 5)
```

这里的 from 声明从 operator 模块中导入了加号操作符的函数版本，其名称为 add。

定义了 add5 后，就可以用一个参数调用它，如下所示：

```
add5(15)
```

这个调用返回 20。

如第 6 章所述，Python 包括列表和列表包含。

Ruby 的块是发送给方法的有效子程序，它将方法变成高阶子程序。Ruby 块可以用 lambda 转换为一个子程序对象。例如下面的代码：

```
times = lambda {|a, b| a * b}
```

下面是使用 times 的例子：

```
x = times.(3, 4)
```

这个语句把 x 设置为 12。times 对象可以用下面的语句 curried：

```
times5 = times.curry.(5)
```

这个函数可以这样使用：

```
x5 = times5.(3)
```

它把 x5 设置为 15。

C#包含 list 类的 FindAll 方法。FindAll 的作用类似于 ML 的 filter 函数。C#还支持泛型列表数据类型。

15.11 函数式语言和命令式语言的比较

本节讨论函数式语言和命令式语言的一些区别。

函数式语言可以具有非常简单的语法结构。LISP 的表结构既可用于代码，也可用于数据，显然就是一个例子。命令式语言的语法复杂得多，因此学习和使用都比较困难。

函数式语言的语义也比命令式语言更简单。例如，在 3.5.2 节中给出的命令式循环结构的指称语义描述中，循环从一种迭代结构转换为递归结构。这种转换在纯函数式语言中是不必要的，在这样的语言中没有迭代。而且第 3 章假设，在所有命令式结构的指称语义描述中没有表达式副作用，这一限制是不现实的，因为所有基于 C 的语言都有表达式副作用。对于纯函数式语言的指称语义描述来说，这样的限制并不需要。

在函数式程序设计社区的一些人声称，使用函数式程序设计可使开发效率增长一个数量级，主要因为函数式程序只有相应命令式程序的 10%大小。这一数字已经在某些问题领域实际体现出来，而在另一些问题领域，函数式程序大约是解决同样问题的命令式程序的 25%大小(Wadler，1998)。这些因素使函数式程序设计的支持者声称，函数式程序设计的开发效率是命令式程序设计的 4～10 倍。然而，单单程序大小并不是开发效率的良好度量标准。显然并非所有源代码行都有相同的复杂度，也不是花费相同的时间开发出来的。实际上，由于必须处理变量，命令式程序有许多极其简单的行来初始化变量，以及使变量有较小的变化。

执行效率是另一个比较标准。函数式程序是解释执行的，显然比编译后的命令式程序慢得多。不过现在大多数函数式语言有了编译器，函数式语言和编译后的命令式语言执行速度的差别不再那么大了。有人可能想说，由于函数式程序比等价的命令式程序小得多，它们应该比命令式程序执行起来快得多。不过通常不是这样，因为函数式语言的一些语言特性对执行效率有很大的负面影响，例如懒惰求值。考虑函数式程序和命令式程序的相对效率，有理由估计，一般函数式程序的执行时间是等价的命令式程序的两倍(Wadler, 1998)。这听起来好像是个巨大的差别，通常会使人放弃对指定应用使用函数式语言。然而 2 倍的差异只在执行速度是最重要因素的情况下，才那么重要。在许多情况下，2 倍的执行速度并不是很重要，例如，考虑许多用命令式语言写的程序，如用 JavaScript 和 PHP 写的网页软件，是解释执行的，因此比等价的编译版本慢得多。对这些应用来说，执行速度不是最重要的。

函数式和命令式程序的执行效率区别的另一个来源是，命令式语言设计为在冯·诺依曼体系结构的计算机上高效运行，而函数式语言的设计基于数学函数，这使命令式语言有一个很大的优势。

函数式语言另一个可能的优点是可读性。在许多命令式程序中，处理变量的细节模糊了程序的逻辑。考虑一个计算前 n 个正整数立方和的函数，在 C 语言中，这个函数如下所示：

```
int sum_cubes(int n){
  int sum = 0;
  for(int index = 1; index <= n; index++)
    sum += index * index * index;
  return sum;
}
```

在 Haskell 中，这一函数是：

```
sumCubes n = sum (map (^3) [1..n])
```

这一函数版本只说明了三个步骤：

(1) 建立数值表([1..n])。

(2) 将一个计算立方的函数映射到表中的每一个数上，创建一个新表。

(3) 对新表求和。

由于没有变量和迭代控制的细节，这一函数版本比 C 版本更易读[16]。

在命令式语言中，并发执行难以设计和使用，如第 13 章所述。在命令式语言中，程序员必须静态地将程序划分为可并发的部分，再写成任务，它们的执行通常必须是同步的。这会是一个复杂的过程。函数式语言的程序会自然地分成函数。在纯函数式编程中，这些函数是独立的，因为它们不会产生副作用，其操作也不依赖非局部或全局变量。因此确定哪些函数可以并发执行就容易得多。调用中的实参表达式常常可以并发执行。只要指定函

16. 当然，C 版本还可以编写得更具有函数风格，但大多数 C 程序员可能不会这样编写函数。

数可以并发执行，函数就可以在单独的线程中隐式求值，就像 Multilisp 那样。当然，对共享不可变数据的访问不需要同步。

强烈影响命令式或过程式编程的复杂性的一个简单因素是，程序员需要考虑在开发的每一步考虑程序的开头。在大型程序中，程序的开头是大量的值(用于大量的程序变量)。而在纯函数式编程中，没有状态，所以不需要考虑程序的开头。

准确地弄清楚为什么函数式语言没有得到广泛应用，不是一件容易的事。早期实现的低效显然是一个因素，很可能至少一些当代的命令式程序员认为，用函数式语言写的程序很慢。另外，绝大多数程序员是从命令式语言开始的，使函数式程序在他们看来显得奇怪且难以理解。对于许多喜欢命令式程序设计的人来说，切换到函数式程序设计毫无吸引力而且可能非常困难。另一方面，从函数式语言开始的人却从未发现函数式程序有什么奇怪的。

小　结

数学函数是命名或匿名的映射，只使用条件表达式和递归来控制求值。复杂函数可以用高阶函数或函数形式来构建，把函数用作参数或返回值，或者两者都用。

函数式程序设计语言是对数学函数的模拟。在纯函数式语言中，不使用变量和赋值语句来产生结果，而是使用函数应用、条件表达式、用于执行控制的递归，以及构建复杂函数的函数形式。LISP 起初是一种纯函数式语言，但很快就添加了一些命令式语言的特性来提高效率和易用性。

LISP 的最初版本来自于对人工智能应用的表处理语言的需求。LISP 仍然是这一领域使用最广泛的语言。

LISP 的第一个实现是偶然出现的：开发最初版本的 EVAL 只是为了证明可以写一个通用的 LISP 函数。

由于 LISP 的数据和程序有相同的形式，所以可以让一个程序创建另一个程序。有了 EVAL，就可以动态创建可立即执行的程序。

Scheme 是一种相对简单的只使用静态作用域的 LISP 方言。与 LISP 一样，Scheme 主要的基本函数包括构建和拆分表的函数、条件表达式函数，以及用于数值、符号和表的简单谓词。

Common LISP 是一种基于 LISP 的语言，为包含 20 世纪 80 年代早期的 LISP 方言的大部分特性而设计。它既允许静态作用域的变量，也允许动态作用域的变量，并且包含许多命令式特性。Common LISP 使用宏来定义一些函数。用户可以定义子句的宏。该语言包含阅读器宏，用户也可以定义阅读器宏。阅读器宏定义单符号宏。

ML 是一种静态作用域的强类型函数式程序设计语言，使用一种比 LISP 更接近于命令式语言的语法。它包含一个类型推断系统、异常处理、各种数据结构，以及抽象数据类型。

ML 没有类型强制转换，不允许重载函数。使用实参形式的模式匹配，可以指定函数的多个定义。currying 是由单参函数替代多参函数，返回带多参函数中其他参数的函数的

过程。ML 和其他一些函数式语言支持 currying。

Haskell 与 ML 类似，不过 Haskell 中所有的表达式都是用懒惰求值方法来计算的，这使程序能够处理无限表。Haskell 还支持列表包含，为描述集合提供了一种方便且熟悉的语法。与 ML 和 Scheme 不同，Haskell 是一种纯函数式语言。

F#是支持函数式和命令式编程的.NET 编程语言，包括面向对象编程。其函数式编程核心基于 OCaml，OCaml 是 ML 和 Haskell 的衍生物。F#由一个广泛使用的 IDE 支持，它还与其他.NET 语言交互操作，可以访问.NET 类库。

文献注释

第一个发行的 LISP 版本可以在(McCarthy，1960)中找到。从 20 世纪 60 年代中期到 70 年代晚期广泛使用的版本，在(McCarthy 等，1965)和(Weissman，1967)中有介绍。(Steele，1990)中介绍了 Common LISP。在(Dybvig，2003)中介绍了 Scheme 语言在(Milner 等，1990)中定义了 ML。(Ullman，1998)是一本优秀的介绍 ML 的教科书。(Thompson，1999)介绍用 Haskell 进行程序设计。(Syme 等，2010)介绍了 F#。

本章使用的 Scheme 程序是用 DrRacket 的旧语言 R5RS 编写的。

(Henderson，1980)中对函数式程序设计进行了概括性的严格讨论。(Peyton Jones，1987)详细地介绍了通过图归约来实现函数式语言的过程。

复 习 题

1. 给出函数形式、简单表、绑定的变量和引用透明的定义。
2. λ 表达式定义了什么内容？
3. 最初的 LISP 语言具有哪种数据类型？
4. LISP 的表通常用哪种常用的数据结构来存储？
5. 说明当参数是数据表时，为什么需要 QUOTE。
6. 什么是简单表？
7. 缩写 REPL 表示什么？
8. IF 的三个参数是什么？
9. =、EQ?、EQV?和 EQUAL?之间的差别是什么？
10. Scheme 的特殊形式 DEFINE 和 Scheme 的基本函数所用求值方法有何不同？
11. DEFINE 的两种形式是什么？
12. 说出 COND 的语法和语义。
13. 为什么 CAR 和 CDR 这样命名？
14. 如果用两个原子，例如'A 和'B，来调用 CONS，返回值是什么？
15. 说出 Scheme 中 LET 的语法和语义。
16. CONS、LIST 和 APPEND 的差别是什么？
17. 说出 Scheme 中 mapcar 的语法和语义。

18. 什么是尾递归?为什么将使用递归来表示重复执行的函数定义成尾递归很重要?
19. 为什么在大多数的 LISP 方言中加入了命令式特性?
20. 在哪些方面 Common LISP 和 Scheme 是相反的?
21. Scheme 使用哪一种作用域规则? Common LISP 呢? ML 呢? Haskell 呢? F#呢?
22. Common LISP 处理器的阅读器阶段会发生什么?
23. 在哪两个方面 ML 与 Scheme 有很大不同?
24. ML 求值环境中存储了什么?
25. ML 的 val 语句和 C 中的赋值语句有什么区别?
26. 在 ML 中使用的类型推断是什么?
27. ML 中 fn 保留字的作用是什么?
28. 处理标量的 ML 函数可以是通用的吗?
29. 什么是 curried 函数?
30. 部分求值是什么意思?
31. 描述 ML 中 filter 函数的操作。
32. ML 中的哪个运算符对应于 Scheme 的 CAR?
33. ML 中的哪个运算符用于函数组合?
34. Haskell 的哪三个特性使它显著区别于 ML?
35. 懒惰求值意味着什么?
36. 什么是严格的程序设计语言?
37. F#支持什么编程范型?
38. F#与其他什么编程语言交互操作?
39. F#中 let 的作用是什么?
40. F#中 let 结构的作用域如何终止?
41. F#中序列和表的根本区别是什么?
42. 在范围方面，ML 和 F#的 let 有什么区别?
43. F#中 λ 表达式的语法是什么?
44. F#表达式中的数值要强制转换吗? 给出支持这个设计选择的论据。
45. Python 为函数式编程提供了什么支持?
46. Python 中的什么函数用于创建 curried 函数?
47. 函数式语言的使用率是上升还是下降?
48. 函数式程序设计语言的哪一个特点使它的语义比命令式语言更简单?
49. 用代码行来比较函数式语言和命令式语言的开发效率，有什么缺点?
50. 为什么函数式语言比命令式语言更容易并发?

习　题

1. 阅读 John Backus 关于 FP 的论文(Backus，1978)，比较本章介绍的 Scheme 特性与相应的 FP 特性。

2. 查找 Scheme 函数 EVAL 和 APPLY 的定义，解释它们的行为。
3. Teitelmen 和 Masinter 的文章《INTERLISP 程序设计环境》(*IEEE Computer*，第 14 卷，第 4 期，1981 年 4 月)中说道：用于 LISP 的 INTERLISP 系统是最现代最完整的程序设计环境之一。仔细阅读这篇文章，比较在你的系统上编写 LISP 程序和用 INTERLISP 编写 LISP 程序的难易程度(假设你平时不使用 INTERLISP)。
4. 查阅一本关于 LISP 程序设计的书，弄清楚支持在 LISP 中包含 PROG 特性的原因是什么。
5. 查找至少一种类型化函数式编程语言在如下领域用于建立商业系统的例子：数据库处理、财务建模、统计分析和生物信息学。
6. 函数式语言可以使用除表之外的某些数据结构。例如，可以使用单字符符号序列。这种语言可以有哪些基本函数来代替 Scheme 中的 CAR、CDR 和 CONS 函数？
7. 列出 ML 中没有的 F#特性清单。
8. 如果 Scheme 是一种纯函数式语言，它可以包含 DISPLAY 吗？为什么？
9. 以下 Scheme 函数的功能是什么？

```
(define (y s lis)
  (cond
    ((null? lis) '() )
    ((equal? s (car lis)) lis)
    (else (y s (cdr lis)))
))
```

10. 以下 Scheme 函数的功能是什么？

```
(define (x lis)
  (cond
    ((null? lis) 0)
    ((not (list? (car lis)))
      (cond
        ((eq? (car lis) #f) (x (cdr lis)))
        (else (+ 1 (x (cdr lis))))))
(else (+ (x (car lis)) (x (cdr lis))))))
```

编程练习

1. 写一个 Scheme 函数，给定球的半径，计算它的体积。
2. 写一个 Scheme 函数，计算给定二次方程的实数根。如果根是复数，函数必须显示一条消息说明。这个函数必须使用 IF 函数，函数的三个参数是二次方程的三个系数。
3. 用 COND 函数而不是 IF 函数，再做编程练习 2。
4. 写一个 Scheme 函数，有两个数值参数 A 和 B，返回 B 的 A 次方。
5. 写一个 Scheme 函数，返回给定的数值简单表中零的个数。
6. 写一个 Scheme 函数，以一个数值的简单表为参数，返回输入表中最大数和最小数组成的表。

7. 写一个 Scheme 函数，以一个表和一个原子为参数，从给定的表中删除给定原子的所有顶层实例。
8. 写一个 Scheme 函数，删除给定表中的最后一个元素。
9. 再做编程练习 7，但是给定原子可以是一个原子或一个表。
10. 写一个 Scheme 函数，以两个原子和一个表作为参数，将表中所有出现的第一个给定的原子替换为第二个给定的原子，且不管第一个原子嵌套得多深。
11. 写一个 Scheme 函数，返回逆序的简单表参数。
12. 写一个 Scheme 谓词函数，测试两个给定表的结构是否相同。如果两个表有相同的表结构，就称这两个表的结构相同，即使它们的原子有可能不同。
13. 写一个 Scheme 函数，返回表示集合的两个简单表参数的并集。
14. 写一个 Scheme 函数，有两个参数，一个是原子一个是表，返回删除了所有给定原子的表(不管原子有多深)。返回的表不能包含任何代替所删除原子的内容。
15. 写一个 Scheme 函数，以一个表作为参数，返回删除了顶层第二个元素的表。如果给定表没有两个元素，则返回()。
16. 写一个 Scheme 函数，以一个数值的简单表作为参数，返回以数值的升序排列的表。
17. 写一个 Scheme 函数，以一个数值的简单表作为参数，返回表中最大和最小的数。
18. 写一个 Scheme 函数，以一个数值的简单表作为参数，返回一个包含给定表所有排列的表。
19. 用 Scheme 编写快速排序算法。
20. 将以下 Scheme 函数重写为尾递归的函数：

```
(DEFINE (doit n)
  (IF (= n 0)
    0
    (+ n (doit (- n 1)))
))
```

21. 用 F#重写前 19 个编程练习中的任一题。
22. 用 ML 重写前 19 个编程练习中的任一题。

第16章

逻辑程序设计语言

16.1 概述
16.2 谓词演算简介
16.3 谓词演算与定理证明
16.4 逻辑程序设计概述
16.5 Prolog 的起源
16.6 Prolog 的基本元素
16.7 Prolog 存在的缺陷
16.8 逻辑程序设计的应用

本章的目的是介绍逻辑程序设计的原理和逻辑程序设计语言，包括对一个 Prolog 子集的简要介绍。将首先介绍谓词演算，这是逻辑程序设计语言的基础，接着介绍如何将谓词演算用于自动定理证明系统。然后给出逻辑程序设计的一般概述。接下来很长的一节介绍 Prolog 程序设计语言的基础知识，包括算术运算、表处理，以及用于帮助调试程序并显示 Prolog 系统如何工作的跟踪工具。最后两个小节介绍 Prolog 作为一种逻辑语言的一些问题，以及 Prolog 的一些应用领域。

16.1 概述

第 15 章介绍了函数式程序设计的例子，与用命令式语言的软件开发方法有巨大差别。本章介绍另一种不同的程序设计方法。在这种方法中，解决方案是用符号逻辑的形式来表示程序，并使用逻辑推导过程来生成结果。逻辑程序是声明性的而不是过程性的，这意味着只说明所需要结果的详细要求，而不是产生结果的详细过程。用逻辑编程语言编写的程序是事实和规则集。这种程序在使用时，是向它提问题，它则尝试通过咨询事实和规则来回答。这里的“咨询”可能有误导作用，因为该过程比这个字的含义要复杂得多。这种解决问题的方法听起来像是只能解决非常少的问题，但它比我们想象中的更灵活。

用一种符号逻辑作为程序设计语言来进行程序设计，通常称为逻辑程序设计，基于符号逻辑的语言称为逻辑程序设计语言，或声明性语言。我们选择介绍逻辑程序设计语言 Prolog，因为它是唯一广泛使用的逻辑语言。

逻辑程序设计语言的语法与命令式和函数式语言差别巨大。逻辑程序的语义与命令式语言程序的语义也少有相似之处。这些现象会使读者对逻辑程序设计和声明性语言的本质感到好奇。

16.2 谓词演算简介

在介绍逻辑程序设计之前，必须简单地研究一下它的基础，即形式逻辑。这不是我们在本书中第一次接触形式逻辑，它在第 3 章介绍的公理语义中用得很多。

命题是一个逻辑陈述，可能是真也可能是假。命题由对象以及对象间的关系构成。形式逻辑的提出为描述命题提供了一种方法，其目的是能够检查形式化陈述的命题是否正确。

符号逻辑可用于满足形式逻辑的三个基本要求：表示命题、表示命题间的关系，以及描述如何根据假设为真的命题推断出新的命题。

形式逻辑和数学有紧密的联系。实际上，数学中的许多内容可以用逻辑来看待。数论与集合论的基本公理是命题的初始集合，假设为真。定理是可从初始集合推断得到的新命题。

用于逻辑程序设计的符号逻辑的特定形式，称为一阶谓词演算。通常称为谓词演算，虽然这种说法有些不准确。下面的小节简单地介绍谓词演算，目的是为讨论逻辑程序设计

和逻辑程序设计语言 Prolog 奠定了基础。

16.2.1　命题

逻辑程序设计命题中的对象用简单项来表示，简单项是常量或变量。常量是一个表示对象的符号，变量是一个在不同时间表示不同对象的符号，从某种意义上来说比命令式程序设计语言中的变量更加接近数学。

最简单的命题称为原子命题，由复合项构成。复合项是数学关系的一个元素，表示为数学函数的形式。如第 15 章所述，数学函数是一个映射，可以表示成一个表达式、一个表或者一个元组的表。复合项是函数的表格定义中的元素。

复合项由两部分组成：函数算符(函数算符是命名关系的函数符号)和有序的参数表，两者在一起表示关系的一个元素。有一个参数的复合项是一元组，有两个参数的复合项是二元组，以此类推。例如，有两个命题：

man(jake)
like(bob，steak)

这两个命题声明{jake}是名为 man(男人)的关系中的一个一元组，{bob，steak}是名为 like(喜欢或相似)的关系中的一个二元组。如果在前两个命题上添加命题

man(fred)

那么关系 man 就有两个不同的元素{jake}和{fred}。这些命题中的所有简单项 man、jake、like、bob 和 steak 都是常量。注意这些命题没有内在的语义。我们希望它们具有什么样的意义，它们就具有什么样的意义。例如，第二个例子的意义可以是 bob 喜欢(1ike)牛排(steak)，或者牛排(steak)喜欢(1ike)bob，或者 bob 在某一方面与牛排(steak)很像。

命题有两种陈述方法，一种是将命题定义为真的方法，另一种是命题的真假不确定的方法。换句话说，命题可以陈述为事实或查询。示例的命题既可以是事实，也可以是查询。

复合命题有两个或更多个原子命题，以与命令式语言中复合逻辑表达式相同的构造方式，由逻辑连接符(或称逻辑运算符)连接而成。谓词演算逻辑连接符的名称、符号和意义如下：

名称	符号	示例	含义
非	¬	¬a	不是 *a*
合取	∩	*a* ∩ *b*	*a* 和 *b*
析取	∪	*a* ∪ *b*	*a* 或 b
等价	≡	*a* ≡ *b*	*a* 与 *b* 相等
蕴涵	⊃	*a* ⊃ *b*	*a* 意味着 *b*
	⊂	*a* ⊂ *b*	*b* 意味着 a

以下是复合命题的例子：

a ∩ b ⊃ c
a ∩ ¬b ⊃ d

¬运算符具有最高优先级。运算符∩、∪和≡都具有比⊃和⊂更高的优先级。因此第二个例子等价于

(a ∩ ¬(b)) ⊃ d

只有在以称为量词的特殊符号开始时，变量才能出现在命题中。谓词演算包含两个量词，如下所示，其中 X 是一个变量，P 是一个命题：

名称	示例	含义
全称	$\forall X.P$	对于所有的 X，P 都为真
存在	$\exists X.P$	X 存在一个使 P 为真的值

X 和 P 中间的点号只是将变量与命题分开。例如，考虑以下复合命题：

$\forall X.(\text{woman}(X) \supset \text{human}(X))$
$\exists X.(\text{mother}(\text{mary}, X) \cap \text{male}(X))$

第一个命题的意思是，对于 X 的任意取值，如果 X 是一个女人(woman)，那么 X 是一个人(human)。第二个命题的意思是，存在 X 的一个取值使 mary 是 X 的母亲(mother)，并且 X 是男的(male)，换句话说 mary 有一个儿子。全称量词和存在量词的作用域是它们所附加的原子命题。这个作用域可以用圆括号来扩展，就像刚才介绍的两个复合命题那样。因此全称量词和存在量词的优先级比其他任何运算符都高。

16.2.2 子句形式

介绍谓词演算，是因为它是逻辑程序设计语言的基础。与其他语言类似，逻辑语言在最简单的形式下是最好的，这意味着应该将冗余最小化。

前面介绍的谓词演算的一个问题是，有太多不同的方式来声明具有相同含义的命题，也就是说有大量冗余。这对于逻辑学家来说不是个问题，但如果谓词演算要用于自动的(计算机化的)系统，这就是个严重的问题。为了简化问题，需要一种标准的命题形式。作为一种相对简单的命题形式，子句形式就是这样一种标准形式。所有的命题都可以用子句形式来表示。以子句形式表示的命题具有以下通用的语法：

$$B_1 \cup B_2 \cup \cdots \cup B_n \subset A_1 \cap A_2 \cap \cdots \cap A_m$$

其中 A 和 B 都是项。这一子句形式的命题含义如下：如果所有的 A 为真，那么至少有一个 B 为真。子句形式的命题的主要特点如下：不需要存在量词，全称量词隐含在对原子命题的变量使用中，不需要除合取和析取以外的运算符，而且合取和析取只需要以通用子

句形式中的顺序出现：析取在左边，合取在右边。所有谓词演算命题可以通过算法转化为子句形式。(Nilsson，1971)证明这是可以做到的，同时给出了一个简单的转化算法。

子句形式命题的右边称为前提，左边称为结论，因为它是前提为真的结果。下面是子句形式命题的例子：

$$\text{likes(bob, trout)} \subset \text{likes(bob, fish)} \cap \text{fish(trout)}$$

$$\text{father(louis, al)} \cup \text{father(louis, violet)} \subset \text{father(al, bob)} \cap \text{mother(violet, bob)} \cap \text{grandfather(louis, bob)}$$

第一个命题的英语版本是，如果 bob 喜欢(likes)鱼(fish)，并且鲑鱼(trout)是鱼(fish)，那么 bob 喜欢 likes)鲑鱼(trout)。第二个命题是，如果 al 是 bob 的父亲(father)，并且 violet 是 bob 的母亲(mother)，并且 louis 是 bob 的祖父(grandfather)，那么 louis 是 al 的父亲(father)或者是 violet 的父亲(father)。

16.3　谓词演算与定理证明

谓词演算提供了一种表示成组命题的方法。成组命题的用途之一是确定是否可以从中推断出感兴趣的或有用的事实。这与数学家的工作非常类似，数学家就是努力从已知的公理和定理推理发现新的定理。

人们在计算机科学的早期时代(20 世纪 50 年代和 20 世纪 60 年代早期)就对自动的定理证明过程有很大兴趣。在自动定理证明方面最重要的突破，是锡拉库扎大学的 Alan Robinson 发现了归结原理(Alan Robinson，1965)。

归结是一条推理规则，可以从给定的命题计算出推理的命题，从而提供了一种可能应用于自动定理证明的方法。归结应用于子句形式的命题。归结的原理如下：假设有两个命题具有如下形式：

$$P_1 \subset P_2$$

$$Q_1 \subset Q_2$$

它们的含义是 P_2 蕴涵 P_1，并且 Q_2 蕴涵 Q_1。进一步假设 P_1 与 Q_2 相同，就可以将 P_1 和 Q_2 重命名为 T，那么可以将两个命题重写为：

$$T \subset P_2$$

$$Q_1 \subset T$$

现在，由于 P_2 蕴涵 T，且 T 蕴涵 Q_1，逻辑上显然 P_2 蕴涵 Q_1，可写为：

$$Q_1 \subset P_2$$

从开始的两个命题推理出这一命题的过程就是归结。

另一个例子考虑以下两个命题：

older(joanne, jake) ⊂ mother(joanne, jake)
wiser(joanne, jake) ⊂ older(joanne, jake)

根据这两个命题，可以使用归结推出以下命题：

wiser(joanne, jake) ⊂ mother(joanne, jake)

这个归结推定的过程很简单：将两个子句命题左边的项 OR 在一起，得到新命题的左边，然后将两个子句命题右边的项 AND 在一起，得到新命题的右边，接着将任何在新命题两边都出现的项从两边移除。当命题一边或两边有多个项时，这一过程也完全相同。新推理得到的命题的左边最开始包含两个给定命题左边所有的项，新的右边也类似地构成，然后去掉出现在新命题两边的项。例如，如果有

father(bob, jake) ∪ mother(bob, jake) ⊂ parent(bob, jake)
grandfather(bob, fred) ⊂ father(bob, jake) ∩ father(jake, fred)

归结得到

mother(bob, jake) ∪ grandfather(bob, fred) ⊂
parent(bob, jake) ∩ father(jake, fred)

具有除了两个最初命题都有的一个原子命题之外的所有原子命题。去掉第一个命题左边和第二个命题右边的 father(bob, jake)这一原子命题，完成了归结操作。用英语来表达就是：

- 如果：bob 是 jake 的父母(parent)，蕴涵 bob 是 jake 的父亲(father)或母亲(mother)
- 并且：bob 是 jake 和父亲(father)，并且 jake 是 fred 的父亲(father)，蕴涵 bob 是 fred 的祖父(grandfather)
- 那么：如果 bob 是 jake 的父母(parent)，并且 jake 是 fred 的父亲(father)，那么 bob 是 jake 的母亲(mother)，或者 bob 是 fred 的祖父(grandfather)

归结实际上比这些简单例子更加复杂。特别是，命题中出现的变量需要归结过程发现这些变量，使匹配过程能够成功。这一确定变量有用值的过程称为合一(unification)。为了合一而临时给变量赋值，称为实例化。

归结过程常常用一个值来实例化一个变量，但是无法完成所需的匹配，不得不回溯，用另一个值来实例化变量。在介绍 Prolog 的章节中会更详细地介绍合一和回溯。

归结的一个非常重要的性质是它能够检测给定命题集的不一致性，它基于归结原理的形式属性——反驳完整性。其含义是，给定一组不一致的命题，归结可以证明它们是不一致的。这一性质使归结可用来证明定理，方法如下：可以把谓词演算的定理证明想象成给定的一个相关命题的集合，同时定理本身的非命题声明了一个新的命题。将定理取非，以便用归结查找不一致性，来证明定理。这就是反证法，是数学上证明定理时的一种常用方

法。最初的命题通常称为假设，定理的非命题称为目标。

从理论上说，这一过程是正确且有效的。不过归结所需的时间可能是一个问题。当命题集合有限时，归结是一个有限过程，但是在一个很大的命题数据库中找到不一致性所需的时间可能极长。

定理证明是逻辑程序设计的基础，所计算的内容大部分都可以表示为：一个给定事实和关系的表作为假设，以及一个根据假设应用归结推理得到的目标。

在一般命题的假设和目标上应用归结，即使它们采用子句形式，也常常是不切实际的。尽管可以使用子句形式的命题证明定理，也不能在有限的时间内完成。简化归结过程的一种方式是限制命题的形式。一种有效的限制是要求命题采用 Horn 子句，Horn 子句只能有两种形式：左边是单个原子命题或左边为空[1]。子句形式命题的左边有时称为首(head)，左边非空的 Horn 子句称为有首的 Horn 子句。有首的 Horn 子句用于声明关系，如

likes(bob, trout) ⊂ likes(bob, fish) ∩ fish(trout)

左边为空的 Horn 子句通常用于声明事实，称为无首的 Horn 子句，例如：

father(bob, jake)

大部分(但不是所有的)命题可以用 Horn 子句来声明。Horn 子句的限制使归结成为证明定理的有效过程。

16.4　逻辑程序设计概述

用于逻辑程序设计的语言称为声明性语言，因为用逻辑程序设计语言写的程序由声明而不是赋值和控制流语句构成。这些声明实际上是用符号逻辑表示的语句(命题)。

逻辑程序设计语言的基本特性之一是它们的语义，称为声明语义。这种语义的基本原理是，有一种简单方法来确定每一条语句的意义，并且不取决于如何用语句来解决问题。声明语义比命令式语言的语义简单得多。例如，在逻辑程序设计语言中，给定命题的意义可以根据语句本身来精确判定。在命令式语言中，简单赋值语句的语义需要检查局部声明、了解语言的作用域规则的内容，甚至可能要检查其他文件中的程序，来确定赋值语句中变量的类型。然后，如果赋值语句的表达式包含变量，必须跟踪在赋值语句前执行的程序来确定这些变量的值。接着，语句的结果行为取决于它运行时的上下文。将这种语义与逻辑语言的命题相比，显然声明语义比命令式语言的语义简单得多，不需要考虑文本上下文和执行顺序。因此，声明语义通常是声明性语言相对于命令式语言的优点之一(Hogger，1984，240～241 页)。

命令式语言和函数式语言的程序设计主要是过程式的，这意味着程序员知道程序要完

1. Horn 子句以 Alfred Horn(1951)命名，Horn 研究这种形式的子句。

成的是什么，并且完全指挥计算机如何做计算。换句话说，计算机是一个服从命令的简单设备，所计算的任何内容都必须给出计算的所有细节。有人认为这是使用命令式语言和函数式语言进行程序设计时有困难的根本原因。

用逻辑程序设计语言进行程序设计是非过程式的。这种语言的程序并不完全描述如何计算得到结果，而是描述结果的形式。区别在于，我们假设计算机系统在某种程度上能确定如何计算得到结果。逻辑程序设计语言为了提供这个功能，只需要以一种精确的方式给计算机提供相关信息，以及计算所需要结果的推理方法。谓词演算提供了与计算机沟通的基本形式，归结提供了推理技术。

经常用来显示过程式系统和非过程式系统的区别的例子就是排序。在类似 Java 的语言中，为了实现排序，需要在一个 Java 程序中，向拥有 Java 编译器的计算机说明某排序算法的所有细节。计算机将 Java 程序翻译为机器码或某种可解释执行的中间代码后，按照指令生成有序表。

在非过程式语言中，只需要描述有序表的特点：它是给定表的某种排列，对于每一对相邻的元素，两个元素间满足给定的关系。为了形式化地说明，假设要排序的表放在 list 数组中，其下标的范围是 1 到 n。对给定表 old_list 的元素排序，并存放在另一个数组 new_list 中，这一概念可以表示如下：

$$\text{sort(old_list, new_list)} \subset \text{permute(old_list, new_list)} \cap \text{sorted(new_list)}$$

$$\text{sorted(list)} \cap \forall j \text{ 使 } 1 \leqslant j < n, \text{list}(j) \leqslant \text{list}(j+1)$$

其中 permute 是一个谓词，如果第二个参数数组是第一个参数数组的一个排列，就返回真。

根据这一描述，非过程式语言系统可以生成有序表。这让非过程式程序设计看起来好像只需要生成软件需求的清晰说明即可，这是一个美好的想法。但遗憾的是，事实并不是这么简单。只使用归结的逻辑程序面临严重的执行效率问题。在排序例子中，如果表很长，排列的个数就会很大，必须逐个生成、测试它们，直到找到需要的排列为止，这是一个非常长的过程。而且，逻辑语言的最佳形式可能还没有确定，人们还没有找到用逻辑程序设计语言为大型问题创建程序的好方法。

16.5 Prolog 的起源

如第 2 章所述，艾克斯·马赛大学的 Alain Colmerauer 和 Phillippe Roussel 在爱丁堡大学 Robert Kowalski 的帮助下，提出了 Prolog 的基本设计。Colmerauer 和 Roussel 对自然语言的处理感兴趣，Kowalski 对自动的定理证明感兴趣。艾克斯·马赛大学和爱丁堡大学的合作一直持续到 20 世纪 70 年代中期。从那时起，对这种语言的开发和使用的研究在这两个地方独立开展，其结果就是产生了两个语法不同的 Prolog 方言。

Prolog 的发展和逻辑程序设计的其他研究工作，在爱丁堡和马赛以外，只得到很有限

的关注，直到 1981 年日本政府宣布启动一个大型研究项目，该项目称为第五代计算机系统(the Fifth Generation Computing Systems，FGCS)(Fuchi，1981；Moto-oka，1981)。这个项目的主要目标之一是开发智能的计算机，该项目选择 Prolog 作为工作基础。FGCS 的宣布，引起了美国和一些欧洲国家的研究人员和政府对人工智能和逻辑程序设计突然而强烈的兴趣。

经过 10 年的努力，FGCS 项目无声地中止了。除了对逻辑程序设计和 Prolog 潜力的巨大期望，人们并没有发现多少重要的东西。于是对 Prolog 的兴趣和使用下降了，虽然 Prolog 仍然有它的应用领域和支持者。

16.6　Prolog 的基本元素

现在有一些不同的 Prolog 方言，可以分成几类：从马赛小组发展而来的一类、从爱丁堡小组发展而来的一类，以及一些为微机开发的方言，如(Clark 和 McCabe，1984)介绍的 Micro-Prolog。这些方言的语法形式有些差别。这里不介绍几种 Prolog 方言的语法或是某种混合的语法，而是选择一种特定的广泛可用的方言，即在爱丁堡开发的方言。这种语言的形式有时称为爱丁堡语法(Edinburgh syntax)，它最早在 DEC System-10 计算机上实现(Warren 等，1979)。在几乎所有流行的计算机平台上都可以获取 Prolog 实现，例如可以从自由软件组织(Free Software Organization)的网站(http://www.gnu.org)上获取。

16.6.1　项

与其他语言的程序一样，Prolog 程序由成组的语句构成。在 Prolog 中只有少数几类语句，但是语句可以很复杂。所有的 Prolog 语句和 Prolog 数据都是由项构成的。

Prolog 的项是常量、变量或结构。常量是一个原子或一个整数。原子是 Prolog 的符号值，与 LISP 中的原子类似。特别地，原子是以小写字母开头的以字母、数字和下划线组成的串，或者是由单引号为界的任意可打印 ASCII 字符组成的串。

变量是以大写字母或下划线开头的以字母、数字和下划线组成的串。变量不是由声明绑定到类型上的，为变量绑定一个值就绑定了类型，称为实例化。实例化只出现在归结过程中。没有赋值的变量称为未实例化的变量。实例化只持续到它满足完整的目标为止，这个目标涉及命题的证明或反证。Prolog 变量在语义和使用方法上，只是命令式语言中变量的远亲。

最后一类项称为结构。结构表示谓词演算的原子命题，其一般形式是相同的：

```
函数算符(参数表)
```

函数算符是任意的原子，用于标识结构。参数表可以是任意原子、变量或其他结构的表。如下面小节所述，结构是 Prolog 中说明事实的方式。结构也可以看作对象，可以用一些相关的原子来声明事实。从这个意义上说，结构就是关系，结构声明了项之间的关系。当结构的上下文表明它是一个查询(问题)时，结构也是一个谓词。

16.6.2 事实语句

对 Prolog 语句的介绍从用于构成假设(假定信息库)的语句开始，根据这些语句可以推理出新的信息。

Prolog 有两种基本的语句形式，与谓词演算的无首 Horn 子句和有首 Horn 子句对应。在 Prolog 中，无首的 Horn 子句最简单的形式是一个结构，解释为无条件的断言，即事实。从逻辑上说，事实只不过是假定为真的命题。

以下例子显示了在 Prolog 程序中能包含的几类事实。请注意，每条 Prolog 语句以点号结尾。

```
female(shelley).
male(bill).
female(mary).
male(jake).
father(bill, jake).
father(bill, shelley).
mother(mary, jake).
mother(mary, shelley).
```

这些简单结构声明了关于 jake、shelley、bill 和 mary 的某些事实。例如，第一个结构声明 shelley 是女性(female)。最后四个结构将两个参数用函数算符原子命名的关系联结起来，例如第五个命题可以解释为 bill 是 jake 的父亲(father)。请注意，这些 Prolog 命题与谓词演算的命题一样，没有固有的语义，它们表示程序员想让它们表示的任何含义。例如，命题

```
father(bill, jake).
```

可以表示 bill 和 jake 有相同的父亲(father)，或者 jake 是 bill 的父亲(father)。不过，最常见、最直接的含义就是 bill 是 jake 的父亲(father)。

16.6.3 规则语句

构成数据库的 Prolog 语句的另一种基本形式，与有首的 Horn 子句对应。这种形式与数学中已知的定理有关，如果满足给定的条件集合，就可以得出结论。语句的右边是前提，即 if 部分，左边是结论，即 then 部分。如果 Prolog 语句的前提为真，那么语句的结论必须也为真。由于是 Horn 子句，所以 Prolog 语句的结论是单个项，而前提可以是单个项或者是合取。

合取包含多个项，由逻辑 AND 运算分隔。在 Prolog 中，AND 运算是隐含的。合取中说明原子命题的结构用逗号分隔，因此可以认为逗号就是 AND 运算符。下面是合取的一个例子：

```
female(shelley), child(shelley).
```

Prolog 有首的 Horn 子句语句的一般形式是：

```
结论:-前提表达式.
```

它读作：如果前提表达式为真，或者通过变量的某种实例化使它为真，就可以得出结论。例如：

```
ancestor(mary, shelley) :- mother(mary, shelley).
```

声明了：如果 mary 是 shelley 的母亲(mother)，那么 mary 是 shelley 的先辈(ancestor)。有首的 Horn 子句称为规则，因为它们声明了命题间蕴涵的规则。

就像谓词演算中的子句形式命题一样，Prolog 语句可以使用变量来使它们的含义一般化。子句形式中的变量提供了一种隐含的全称量词。以下语句展示了 Prolog 语句中变量的使用：

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
grandparent(X, Z) :- parent(X, Y) , parent(Y, Z).
```

这些语句给出一些变量(全称对象)间的蕴涵规则。在这个例子中，全称对象是 X、Y 和 Z。第一条规则声明，如果存在 X 和 Y 的实例化使 mother(X, Y)为真，那么对于 X 和 Y 相同的实例化，parent(X, Y)为真。

=运算符是一个中缀运算符，如果它的两个项操作数相同，它就返回真，例如 X=Y。not 运算符是一个一元运算符，对其操作数取反，即如果它的操作数为假，它就返回真。例如，如果 X 不等于 Y，not(X=Y)就返回真。

16.6.4　目标语句

前面介绍了用于逻辑命题的 Prolog 语句，它描述已知事实和说明事实间逻辑关系的规则。这些语句是定理证明模型的基础。定理用一种命题的形式表示，我们希望系统证明或反证这些命题。在 Prolog 中，这些命题称为目标或查询。Prolog 目标语句的语法形式与无首的 Horn 子句相同。例如，

```
man(fred).
```

对此系统会给出 yes 或 no 的响应。答案 yes 表示系统已经证明：在给定的事实数据库和关系下，该目标为真。答案 no 表示该目标确定为假或系统无法证明它。

合取命题和有变量的命题也是合法的目标。有变量时，系统不仅要断言目标的正确性，而且要确定使目标为真的变量的实例化。例如，可以问

```
father(X, mike).
```

系统会通过合一来尝试找到使目标为真的 X 的实例化。

由于目标语句和一些非目标语句有相同的形式(无首的 Horn 子句)，Prolog 实现必须有某种方法来区分两者。交互式 Prolog 的实现采用两种模式，由不同的交互式提示来表示：一种用于输入事实语句和规则语句，另一种用于输入目标。用户可以在任何时候改变模式。

16.6.5 Prolog 的推理过程

本节研究 Prolog 的归结。高效地使用 Prolog 需要程序员确切地知道 Prolog 系统对程序做了什么。

查询称为目标。当目标是复合命题时，每一个事实(结构)称为子目标。为了证明目标为真，推理过程必须找到一条推理规则的链和/或数据库中的事实，将目标与数据库中的一个或多个事实连接起来。例如，如果 Q 是目标，那么或者在数据库中找到 Q 这一事实，或者推理过程找到事实 P_1 和一系列命题 P_2，P_3，…，P_n，使

$P_2 : - P_1$

$P_3 : - P_2$

…

$Q : - P_n$

当然，由于右边为复合命题的规则和带变量的规则，这一过程可以很复杂，且经常很复杂。找到这些 P(如果存在的话)的过程，基本就是项和项之间比较或匹配的过程。

由于证明子目标的过程是通过命题匹配过程实现的，所以有时称为匹配。在某些情况下，证明一个子目标称为满足此子目标。

考虑以下查询：

```
man(bob).
```

这一目标语句是最简单的一类，归结相对容易确定它为真还是假：将这一目标的模式与数据库中的事实与规则比较，如果数据库包含事实：

```
man(bob).
```

则证明就十分容易；但是如果数据库包含以下事实和推理规则：

```
father(bob).
man(X) :- father(X).
```

就需要 Prolog 找到这两条语句，并使用它们来推理出目标为真。这就必须要用合一来临时将 X 实例化为 bob。

现在，考虑以下目标：

```
man(X).
```

这时 Prolog 必须将目标与数据库中的命题相匹配。Prolog 找到的第一个具有目标形式的命题，带有任意一个对象作为它的参数，会使 X 以这个对象的值实例化，然后将 X 作为结果显示出来。如果没有命题具有目标形式，系统会说 no，表示目标不能满足。

有两种相反的方式来尝试将给定目标与数据库中的事实相匹配。系统可以从数据库的事实和规则开始，尝试找到通向目标的一系列匹配，这一方式称为自底向上归结，或称为

前向链接。另一种方式是从目标开始，尝试找到一系列匹配的命题，通向数据库中原始事实的某个集合，这种方式称为自顶向下归结，或称为后向链接。一般来说，当候选答案的集合较小时，后向链接很有效；当有可能正确的答案数目巨大时，前向链接的方式更好，这时后向链接需要非常大量的匹配来得到答案。Prolog 的实现使用后向链接来归结，大概是因为它的设计者认为，后向链接比前向链接适合更多类别的问题。

以下例子显示了前向链接与后向链接之间的区别。考虑查询

```
man(bob).
```

假设数据库包含

```
father(bob).
man(X) :- fathe r(X).
```

前向链接会查找到第一个命题，然后通过 X 实例化为 bob，将第一个命题与第二个命题的右边(father(X))匹配，再将第二个命题的左边与目标匹配，从而推理出目标命题。后向链接会首先通过 X 实例化为 bob，将目标与第二个命题的左边(man(X))匹配，在最后一步将第二个命题的右边(现在是 father(bob))与第一个命题匹配。

当目标有一个以上结构时，会出现又一个设计问题，如后面的例子所示。这个问题是以深度优先的方式还是以宽度优先的方式来进行证明过程的搜索。深度优先的搜索在处理其他子目标前，先为第一个子目标找到完整的命题序列——证明。宽度优先的搜索则并行地处理给定目标的所有子目标。Prolog 的设计者选择了深度优先的方法，主要是因为使用较少的计算机资源就可以实现。宽度优先方法是一个并行的搜索，需要占用大量内存。

Prolog 归结机制的最后一个必须介绍的特性是回溯。当处理带有多个子目标的目标命题，且系统无法证明一个子目标为真时，系统会放弃无法证明的子目标。如果有之前的子目标，系统会重新考虑并尝试找到另一个证明方法。这种在目标中回退并重新考虑先前已经证明的子目标的机制，称为回溯。系统在之前对子目标的搜索停止之处重新搜索，来找到新的证明方法。子目标的多个证明方法来自于变量的不同实例化。回溯需要大量时间和空间，因为可能需要找到每一个子目标的所有可能证明。这些子目标的证明没法组织起来，从而用最少的时间找到能够导致最终完整证明的那一个，使问题更加严重。

为了巩固对回溯的理解，考虑以下例子。假设数据库中有事实和规则集，Prolog 给出了以下复合目标：

```
male(X),   parent(X, shelley).
```

这一目标询问是否有 X 的实例化，使 X 是男性(male)且是 shelley 的父母(parent)。第一步，Prolog 在数据库中找到以 male 作为函数算符的事实，然后将 X 实例化为所找到事实的参数，例如 mike。然后 Prolog 尝试证明 parent(mike，shelley)为真。如果不能证明，就回溯到第一个子目标 male(X)，尝试用 X 的其他实例化来满足它。归结过程在找到 shelley 的父母(parent)前，可能必须找到数据库中每一位男性(male)，一定要找到所有男性(male)才能证明目标无法满足。请注意，如果两个子目标的顺序相反，示例目标可能处理得更高

效一些。于是，只有当找到 shelley 的父母(parent)时，归结过程才会试着将这个人与 male 子目标匹配。如果 shelley 的父母(parent)人数比数据库中男性(male)人数少，效率就会更高，这看起来也是一个合理的假设。16.7.1 节介绍了 Prolog 系统的一种限制回溯的方法。

在 Prolog 中，数据库的搜索总是以从第一个命题到最后一个命题的顺序进行。

以下两小节介绍的 Prolog 例子，进一步说明了归结过程。

16.6.6 简单算术

Prolog 支持整数变量和整数的算术运算。最初，算术运算符是函数算符，因此 7 和变量 X 的和可以写成

```
+(7, X)
```

现在 Prolog 允许算术运算语法用 is 运算符更简短地表示。这个运算符以一个算术表达式作为右操作数，以一个变量作为左操作数。表达式中所有变量必须是已经实例化的，但是左边的变量在之前不能实例化。例如，在

```
A is B / 17 + C.
```

中，如果 B 和 C 是已实例化的而 A 还没有，那么这个子句就会让 A 以表达式的值来实例化。实例化时，就满足了子句。如果 B 或 C 不是已实例化的，或者 A 已实例化，就不满足子句，A 的实例化就不会发生。is 命题的语义与命令式语言中的赋值语句有很大区别，这种区别会导致一个有趣的场景。由于 is 运算符使子句看起来像一个赋值语句，Prolog 的初学者可能写出下面的语句：

```
Sum is Sum + Number.
```

这在 Prolog 中是没有用的，甚至是不合法的。如果 Sum 没有实例化，那么右边对它的引用就是未定义的，子句就会失败。如果 Sum 已经实例化，子句仍会失败，因为当 is 求值时，左操作数不能实例化。在这两种情况下，Sum 实例化为新值的情况都不会发生(如果需要 Sum+Number 的值，可以将它绑定到某个新的名字上)。

Prolog 没有命令式语言相同意义的赋值语句，在 Prolog 所设计应用的大部分程序中，都不需要赋值语句。在命令式语言中赋值语句很有用，因为程序员要能控制嵌入了赋值语句的代码的执行流。由于这类控制在 Prolog 中并非总是可能的，所以赋值语句就远没有那么有用。

作为在 Prolog 中使用数值计算的一个简单例子，考虑以下问题：假设已知在某条跑道上几辆汽车的平均速度，以及它们在跑道上的时间。这些基本信息可以编码为事实，速度(speed)、时间(time)和距离(distance)间的关系可以写成规则，如下：

```
speed(ford, 100).
speed(chevy, 105).
speed(dodge, 95).
speed(volvo, 80).
```

```
time(ford, 20).
time(chevy, 21).
time(dodge, 24).
time(volvo, 24).
distance(X, Y) :- speed(X, Speed),
                  time(X, Time),
                  Y is Speed * Time.
```

现在，可以对特定汽车行驶的距离(distance)进行查询。例如，查询

```
distance(chevy, Chevy_Distance).
```

将 Chevy_Distance 实例化为值 2205。距离计算语句右边的前两条子句，用给定汽车函数算符的相应值来实例化变量 Speed 和 Time。在满足目标后，Prolog 还显示名字 Chevy_Distance 和它的值。

下面从操作的角度看看 Prolog 系统如何产生结果会很有益。Prolog 有一个内置的语言结构 trace，能显示在尝试满足给定目标的每一步中实例化变量的值。trace 用于理解和调试 Prolog 程序。为了理解 trace，最好介绍 Prolog 程序执行的另一个模式，称为跟踪模式。

跟踪模式用四个事件来描述 Prolog 的执行：(1)调用(call)，在开始尝试满足一个目标时发生；(2)退出(exit)，当已满足目标时发生；(3)重做(redo)，当回溯导致尝试重新满足一个目标时发生；(4)失败(fail)，当目标不能满足时发生。如果将像 distance 这样的过程看成子程序，就可以直接将调用和退出关联到命令式语言中子程序的执行模式。另两个事件是逻辑程序设计独有的。在下面的跟踪例子中，要对 Chevy_Distance 值的计算进行跟踪，这一目标不需要重做事件和失败事件。

```
trace.
distance(chevy, Chevy_Distance).

(1) 1 Call: distance(chevy, _0)?
(2) 2 Call: speed(chevy, _5)?
(2) 2 Exit: speed(chevy, 105)
(3) 2 Call: time(chevy, _6)?
(3) 2 Exit: time(chevy, 21)
(4) 2 Call: _0 is 105*21?
(4) 2 Exit: 2205 is 105*21
(1) 1 Exit: distance(chevy, 2205)

Chevy_Distance = 2205
```

在跟踪中，以下划线字符(_)开头的符号是用于存储实例化值的内部变量。跟踪的第一列指示当前尝试匹配的子目标。例如在跟踪的例子中，带有指示(3)的第一行尝试用 chevy 的 time 值来实例化临时变量_6，其中 time 是描述计算 distance 的语句右边的第二项。第二列指示匹配过程的调用深度。第三列指示当前行为。

为了对回溯进行解释，考虑以下示例的数据库和跟踪的复合目标：

```
likes(jake, chocolate).
```

```
likes(jake, apricots).
likes(darcie, licorice).
likes(darcie, apricots).

trace.
likes(jake, X), likes(darcie, X).

(1) 1 Call: likes(jake, _0)?
(1) 1 Exit: likes(jake, chocolate)
(2) 1 Call: likes(darcie, chocolate)?
(2) 1 Fail: likes(darcie, chocolate)
(1) 1 Redo: likes(jake, _0)?
(1) 1 Exit: likes(jake, apricots)
(3) 1 Call: likes(darcie, apricots)?
(3) 1 Exit: likes(darcie, apricots)

X = apricots
```

可以用图形化的方式来想象 Prolog 的计算：将每一个目标看作有四个口(调用、失败、退出和重做)的盒子。控制通过调用口向前进入目标，控制也可以通过重做口反向进入目标，控制可以有两种方式离开目标：如果目标满足，控制通过退出口离开；如果目标不满足，控制通过失败口离开。这一例子的模型如图 16-1 所示。在这个例子中，控制流经每个子目标两次。第二个子目标第一次时没有满足，使控制通过重做口返回第一个子目标。

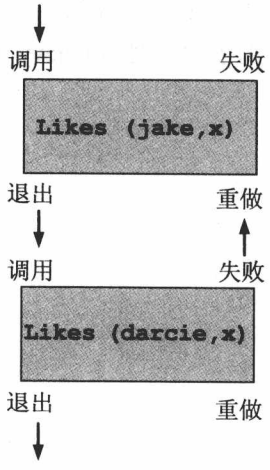


图 16-1 目标 likes(jake, X)和 likes(Darcie, X)的控制流模型

16.6.7 表结构

前面介绍的 Prolog 数据结构只有原子命题，它看起来更像是函数调用，而不是数据结构。原子命题也称为结构，实际上是某种形式的记录。Prolog 支持的另一种基本数据结构是表，表是任意数目的元素序列，其中元素可以是原子、原子命题或其他项，包括其他的表。

Prolog 使用 ML 和 Haskell 的语法来表示表。表元素由逗号分隔，完整的表用方括号括起来，如

```
[apple, prune, grape, kumquat]
```

记号[]用于表示空表。Prolog 没有显式的函数来构造和拆分表，而是简单地使用一个特殊的记号。[X | Y]表示头为 X 尾为 Y 的表，其中头和尾对应于 LISP 中的 CAR 和 CDR，这与 ML 和 Haskell 中使用的记号类似。

表可以用简单的结构来创建，如

```
new_list([apple, prune, grape, kumquat]).
```

声明常量表[apple, prime, grape, kumquat]是关系 new_list(刚创建的名字)的一个新元素。这条语句并不是把表绑定到变量 new_list，而是与命题

```
male(jake)
```

做了同一类的事情，即声明[apple, prune, grape, kumquat]是 new_list 的一个新元素。因此，可以再有一个以表为参数的命题，例如

```
new_list([apricot, peach, pear])
```

在查询模式下，new_list 的一个元素可以拆分成

```
new_list([New_List_Head | New_List_Tail]).
```

如果 new_list 已经设置有如前所示的两个元素，那么这一语句就会用第一个表元素的头(在这个例子中是 apple)来实例化 New_List_Head，用表的尾([prune, grape, kumquat])来实例化 New_List_Tail。如果这是复合目标的一部分，并且回溯强制对它进行新的计算，New_List_Head 和 New_List_Tail 会分别重新实例化为 apricot 和[peach, pear]，因为[apricot, peach, pear]是 new_list 的下一个元素。

用于拆分表的运算符“|”还可以用于从给定的实例化的头和尾部分来创建表，如：

```
[Element_1 | List_2]
```

如果 Element_1 已经实例化为 pickle，List_2 已经实例化为[peanut，prune，popcorn]，示例的写法就会创建表[pickle，peanut，prune，popcorn]。

如前所述，包含“|”符号的表的标记方法是通用的，可以表示表的构建，也可以表示表的拆分。还要注意，以下写法是等价的：

```
[apricot, peach, pear | []]
[apricot, peach | [pear]]
[apricot | [peach, pear]]
```

对于表来说，通常需要某些基本操作，例如 LISP、ML 和 Haskell 中的那些操作。在 Prolog 中，作为这类操作的一个例子，研究一下 append 的定义，它与 LISP 中的 append 函数类似。在这个例子中，可以看到函数式语言和声明性语言的异同点。我们不必指明 Prolog 是如何从给定表中构建新表的，而只需要用给定的表来指明新表的特性。

在外观上，Prolog 的 append 定义与第 15 章中的 ML 版本非常相似，在归结中以相似

的方式使用递归来生成新表。在 Prolog 中，递归是归结过程发起并控制的。与 ML 和 Haskell 一样，用模式匹配过程根据实参从 append 过程的两个不同定义中选择。

在以下代码中，append 操作的前两个参数是要参与 append 操作的两个表，第三个参数是结果表：

```
append([], List, List).
append([Head | List_1], List_2, [Head | List_3]) :-
            append(List_1, List_2, List_3).
```

第一个命题指明，当空表后追加任意表时，结果就是那个表。这一语句与 ML 的 append 函数的递归终止步骤相对应。请注意，终止命题放在递归命题的前面。这么做是因为 Prolog 会从第一个命题开始，按顺序匹配这两个命题(因为使用了深度优先顺序)。

第二个命题指明新表的一些特征，与 ML 函数的递归步骤相对应。谓词的左边声明，新表的第一个元素与第一个给定表的第一个元素相同，因为它们都名为 Head。只要 Head 实例化为一个值，目标中的所有 Head 实际上都同时实例化为这个值。第二条语句的右边指明，在第一个给定表(List_1)的尾后添加第二个给定表(List_2)，形成结果表的尾(List_3)。

append 的第二条语句可以用一种方式来读：只有当表 List_3 是由在 List_1 后追加 List_2 形成时，在表[Head | List_1]后追加任意表 List_2，可生成结果表[Head | List_3]。在 LISP 中，这会是

```
(CONS (CAR FIRST) (APPEND (CDR FIRST) SECOND))
```

在 Prolog 和 LISP 两个版本中，结果表都是直到递归产生终止条件时才构建的，这时第一个表必然为空，然后结果表利用 append 函数本身来构建，将从第一个表中取出的元素按逆序添加在第二个表中。逆序是通过递归的拆分实现的。

Prolog 的 append 与 Lisp 和 ML 的 append 之间的根本差别是，Prolog 的 append 是谓词——它不返回表，而是返回 yes 或 no。新表是第三个参数的值。

为了说明 append 过程如何工作，考虑以下跟踪示例：

```
trace.
append([bob, jo], [jake, darcie], Family).

(1) 1 Call: append([bob, jo], [jake, darcie], _10)?
(2) 2 Call: append([jo], [jake, darcie], _18)?
(3) 3 Call: append([], [jake, darcie], _25)?
(3) 3 Exit: append([], [jake, darcie], [jake, darcie])
(2) 2 Exit: append([jo], [jake, darcie], [jo, jake,
                    darcie])
(1) 1 Exit: append([bob, jo], [jake, darcie],
                   [bob, jo, jake, darcie])
Family = [bob, jo, jake, darcie]
yes
```

前两个调用表示子目标，所用的 List_1 非空，因此根据第二条语句的右边创建递归调

用。第二条语句的左边有效地指明了递归调用(目标)的参数，因此每一步都从第一个表中拆分出一个元素。在一个调用(子目标)中，若第一个表变成空表，第二条语句右边的当前实例因为与第一条语句匹配而得到满足，结果作为第三个参数返回，其值就是在空表后追加第二个初始参数表。在接下来退出时(表匹配成功)，从第一个表中移除的元素添加在结果表 Family 中。当第一个目标的退出完成时，这一过程就完成了，并显示结果表。

Prolog 与 LISP 和 ML 的 append 之间的另一个区别是，Prolog 的 append 比其他语言灵活。例如，在 Prolog 中，可以使用 append 确定，给哪两个表执行如下指令，以获得[a, b, c]：

```
append(X, Y, [a, b, c]).
```

结果如下：

```
X = []
Y = [a, b, c]
```

如果在这个输出中输入一个分号，就会得到另一个结果：

```
X = [a]
Y = [b, c]
```

继续添加分号，得到：

```
X = [a, b]
Y = [c];
X = [a, b, c]
Y = []
```

append 命题也可以用来创建其他表操作，例如以下的操作，请读者来研究其功能。请注意，list_op_2 的用法是：提供一个表作为第一个参数，一个变量作为第二个参数，list_op_2 的结果是第二个参数实例化的值。

```
list_op_2([], []).
list_op_2([Head | Tail], List) :-
list_op_2(Tail, Result), append(Result, [Head], List).
```

可以看出，list_op_2 使 Prolog 系统用一个表来实例化第二个参数，这个表拥有第一个参数表的元素，但顺序相反。例如，([apple, orange, grape]，Q)用表[grape, orange, apple]来实例化 Q。

再一次，虽然 LISP 和 Prolog 语言存在根本性的差别，但类似的操作可使用类似的方法。在逆序操作的例子中，Prolog 的 list_op_2 和 LISP 的 reverse 函数，包含了递归终止条件，其基本的过程是，将表的 CDR 或尾的逆序添加到表的 CAR 或头，生成结果表。

下面是这一过程的跟踪，现在命名为 reverse：

```
trace.
reverse([a, b, c], Q).

(1) 1 Call: reverse([a, b, c], _6)?
```

```
(2) 2 Call: reverse([b, c], _65636)?
(3) 3 Call: reverse([c], _65646)?
(4) 4 Call: reverse([], _65656)?
(4) 4 Exit: reverse([], [])
(5) 4 Call: append([], [c], _65646)?
(5) 4 Exit: append([], [c], [c])
(3) 3 Exit: reverse([c], [c])
(6) 3 Call: append([c], [b], _65636)?
(7) 4 Call: append([], [b], _25)?
(7) 4 Exit: append([], [b], [b])
(6) 3 Exit: append([c], [b], [c, b])
(2) 2 Exit: reverse([b, c], [c, b])
(8) 2 Call: append([c, b], [a], _6)?
(9) 3 Call: append([b], [a], _32)?
(10) 4 Call: append([], [a], _39)?
(10) 4 Exit: append([], [a], [a])
(9) 3 Exit: append([b], [a], [b, a])
(8) 2 Exit: append([c, b], [a], [c, b, a])
(1) 1 Exit: reverse([a, b, c], [c, b, a])

Q = [c, b, a]
```

假设需要确定给定符号是否在给定表中。对此，Prolog 的直接描述是：

```
member(Element, [Element | _]).
member(Element, [_ | List]) :- member(Element, List).
```

下划线表示匿名变量，用于表示我们不关心变量在合一时的实例化。在上面的例子中，如果在一开始或者通过第二条语句进行若干次递归后，Element 是表头，那么满足第一条语句；如果 Element 在表的尾，那么满足第二条语句。考虑以下跟踪示例：

```
trace.
member(a, [b, c, d]).
(1) 1 Call: member(a, [b, c, d])?
(2) 2 Call: member(a, [c, d])?
(3) 3 Call: member(a, [d])?
(4) 4 Call: member(a, [])?
(4) 4 Fail: member(a, [])
(3) 3 Fail: member(a, [d])
(2) 2 Fail: member(a, [c, d])
(1) 1 Fail: member(a, [b, c, d])
no

member(a, [b, a, c]).
(1) 1 Call: member(a, [b, a, c])?
(2) 2 Call: member(a, [a, c])?
(2) 2 Exit: member(a, [a, c])
(1) 1 Exit: member(a, [b, a, c])
yes
```

16.7　Prolog 存在的缺陷

虽然 Prolog 是一个有用的工具，但它不是纯粹的、完美的逻辑程序设计语言。本节介绍 Prolog 的一些问题。

16.7.1　归结的顺序控制

Prolog 为了效率的原因，允许用户在归结过程中控制模式匹配的顺序。在纯粹的逻辑程序设计环境中，在归结过程中发生的尝试匹配的顺序是不确定的，而且可以同时尝试所有匹配。但是，由于 Prolog 总是以同样的顺序来匹配，即从数据库的起始处与目标的左边开始匹配，所以用户可以通过对数据库语句的排序来优化特定的应用，大大地影响效率。例如，如果用户知道在特定的“执行”中，某些规则比其他规则更可能被满足，就可以将这些规则放在数据库的前面，使程序更高效。

除了允许用户控制数据库和子目标的顺序外，Prolog 另一个迁就效率的让步就是允许回溯的某种显式控制。这是通过截断(cut)运算符来实现的，这个运算符用感叹号(!)来表示。截断运算符实际上是一个目标，而不是一个运算符。作为目标，它总是立刻满足，但不会通过回溯而再次满足，因此截断的副作用就是复合目标中左边的子目标也不会通过回溯再次满足。例如，在目标

```
a,b,! ,c,d.
```

中，如果 a 和 b 都满足，但 c 不满足，那么整个目标就不能满足。如果知道只要 c 不满足，那么再次满足 b 或 a 都是浪费时间，就可以使用这个目标。

所以，截断的目的是在推测子目标不会导致完整证明时，使用户能够告诉系统，不应该再尝试重新满足子目标，从而使程序更高效。

作为使用截断运算符的例子，考虑 16.6.7 节中的 member 规则，这一规则是：

```
member(Element, [Element | _]).
member(Element, [_ | List]) :- member(Element, List).
```

如果 member 的表参数表示一个集合，那么目标只能满足一次(集合不包含重复的元素)。因此，如果 member 在一个多子目标的目标语句中用作一个子目标，这就会是一个问题：如果 member 满足但是下一个子目标不满足，回溯就会继续先前的匹配，尝试重新满足 member。但是由于开始时 member 的表参数只有一份元素拷贝，所以不管再如何尝试重新满足 member，它都不可能再满足，最终导致整个目标不满足。例如，考虑目标：

```
dem_candidate(X) :- member(X, democrats), tests(X).
```

这个目标确定，给定的人是否是一个民主党人，并且是特定职位的合适候选者。tests 子目标对给定民主党人的一些特点进行检查，确定这个人对于这一职位是否合适。如果民主党人的集合没有重复，那么如果 tests 子目标不满足，我们就不想回溯到 member 子目标，

因为 member 会搜索所有其他民主党人，并且得不到满足，因为没有重复的人。第二次尝试 member 子目标只会浪费计算时间。解决这种低效的方法就是在 member 定义的第一条语句中，用截断运算符作为单个元素加到右边，如

```
member(Element, [Element | _]) :- !.
```

回溯不会尝试重新满足 member，而是会使整个子目标失败。

在名为生成与检测的 Prolog 程序设计策略中，截断特别有用。在使用了生成与检测策略的程序中，目标由生成可能证明的子目标构成，子目标随后由后面的“检测”子目标检查。被否定的证明需要回溯到“生成器”子目标，生成新的可能证明。作为生成与检测程序的一个例子，考虑以下出现在(Clocksin 和 Mellish，2003)中的程序：

```
divide(N1, N2, Result) :- is_integer(Result),
                          Product1 is Result * N2,
                          Product2 is (Result + 1) * N2,
                          Pr oduct1 =< N1, Product2 >
                             N1, !.
```

这个程序使用加法和乘法来执行整数除法。由于大多数 Prolog 系统将除法作为一个运算符，所以这个程序除了展示一个简单的生成与检测程序外，实际上并没有用。

只要参数实例化为某个非负整数，就满足谓词 is_integer。如果参数没有实例化，is_integer 就将它实例化为 0；如果参数实例化为一个整数，is_integer 就将它实例化为下一个更大的整数值。

所以在 divide 中，is_integer 是生成器子目标，它生成序列 0, 1, 2, …的元素，每一次都满足这个子目标。其他所有的子目标都是检测子目标，它们检查确定 is_integer 生成的值是否是前两个参数 N1 和 N2 的商。截断作为最后一个子目标的目的很简单：一旦找到答案，就防止 divide 尝试寻找其他答案。虽然 is_integer 能够生成大量候选，但答案只有一个数，因此这里的截断防止为生成第二个答案进行无用的尝试。

(van Emden，1980)将截断运算符的使用与命令式语言中 goto 的使用进行了比较。虽然有时需要截断，但可能会被滥用。实际上，截断有时用于使逻辑程序具有命令式程序设计风格的控制流。

在 Prolog 程序中能够改变控制流是一个缺点，因为它直接危害了逻辑程序设计的重要优点之一：程序不说明如何找到答案，只是简单地说明答案的样子。这一设计使程序写起来更简单，读起来更容易。程序不需要有如何确定答案的细节，特别是不需要生成答案的精确计算顺序。因此，虽然逻辑程序设计不需要控制流说明，但主要出于效率的原因，Prolog 程序大量地使用了它们。

16.7.2 封闭世界假设

Prolog 归结的本质有时会产生错误的结果。Prolog 关心的真实性，只是用数据库可以证明的真实性，对于数据库以外的世界一无所知。在数据库中没有足够信息来完全证明的

任何查询，都假定为假。Prolog 能够证明给定的目标为真，但无法证明给定目标为假，只是简单地认为：因为不能证明目标为真，所以目标为假。实际上，Prolog 是一个“真与失败”的系统，并非是一个“真与假”的系统。

实际上，封闭世界假设并不陌生——司法系统就以同样的方式运作。嫌疑人是无罪的，直到被证明有罪。他们不必证明是无罪的。如果审判无法证明一个人有罪，他就被认为是无罪的。

封闭世界假设的问题与否定问题有关，否定问题在下一节介绍。

16.7.3　否定问题

Prolog 的另一个问题就是处理否定时的困难。考虑以下两条事实和一条关系的数据库：

```
parent(bill, jake).
parent(bill, shelley).
sibling(X, Y) :- (parent(M, X), parent(M, Y).
```

现在，假设输入查询

```
sibling(X, Y).
```

Prolog 将响应为

```
X = jake
Y = jake
```

所以，Prolog 认为 jake 是自己的兄弟姐妹(sibling)。这是因为系统首先把 M 实例化为 bill，X 实例化为 jake，来使第一个子目标 parent(M, X)为真，然后从数据的起始处开始匹配第二个子目标 parent(M, Y)，得到 M 的实例化 bill 和 Y 的实例化 jake。由于两个子目标都是从数据库的起始处开始匹配，独立地满足，所以出现所显示的响应。为了避免这一结果，只有 X 和 Y 有相同的父母亲(parent)，并且不是同一个人时，X 是 Y 的 sibling 才满足。遗憾的是，在 Prolog 中不能直接声明它们不同，如后面所述。最确切的方法是为每一对原子添加一个事实，声明它们不同。这显然会使数据库变得非常大，因为通常否定信息比肯定信息多得多。例如，大多数人的非生日天数比生日天数多 364 天。

一个简单的替换方案是在目标中声明 X 不能与 Y 相同，如

```
sibling(X, Y) :- parent(M, X), parent(M, Y), not(X = Y).
```

在其他情况下，解决方案没有这么简单。

在这个例子中，如果归结过程不能满足子目标 X=Y，就会满足 Prolog 的 not 运算符。因此，如果满足了 not，并不意味着 X 不同于 Y，而是意味着归结过程不能根据数据库来证明 X 与 Y 相同，所以 Prolog 的 not 运算符并不等价于逻辑的 NOT 运算符，逻辑的 NOT 运算符意味着可以证明操作数为真。如果有以下形式的子目标，这种不等价就会产生问题：

```
not(not(some_goal)).
```

如果 Prolog 的 not 运算符是真的逻辑 NOT 运算符，那么上述子目标就与下面的目标等价：

```
some_goal.
```

然而在某些情况下，它们并不相同。例如，再次考虑 member 规则：

```
member(Element, [Element | _]) :- !.
member(Element, [_ | List]) :- member(Element, List).
```

为了找到给定表的一个元素，需要使用目标

```
member(X, [mary, fred, barb]).
```

这会使 X 实例化为 mary，然后打印出来。但是如果用

```
not(not(member(X, [mary, fred, barb]))).
```

就会发生一系列事件：首先，满足了内部目标，X 实例化为 mary；然后 Prolog 尝试满足下一个目标：

```
not(member(X, [mary, fred, barb])).
```

这个语句不能满足，因为 member 满足了。这一目标失败时，X 会取消实例化，因为 Prolog 总是对所有失败目标中的变量取消实例化。接着，Prolog 会尝试满足外部的 not 目标，它会满足，因为其参数不能满足。最后，结果 X 会打印出来，但是 X 当前没有实例化，系统也会指出这一点。一般来说，未实例化的变量是以下划线开头的数字串形式打印出来的。因此 Prolog 的 not 与逻辑 NOT 不等价这一事实，至少会引起误解。

逻辑 NOT 不能集成在 Prolog 中的根本原因，是 Horn 子句的形式：

$$A \text{ :- } B_1 \cap B_2 \cap \ldots \cap B_n$$

如果所有的 B 命题为真，就可以推出 A 为真。但不管任何一个或所有的 B 真假如何，都无法推出 A 为假。从正逻辑只能推出正逻辑，因此使用 Horn 子句形式不能得出否定的结论。

16.7.4 固有的限制

如 16.4 节所述，逻辑程序设计的基本目标，是提供非过程式的程序设计，即一个只需要程序员说明程序应该做什么、但不需要说明怎么做的系统。该节给出的排序例子，在这里可以重新写为：

$$\text{sort(old_list, new_list)} \subset \text{permute(old_list, new_list)} \cap \text{sorted(new_list)}$$

$$\text{sorted(list)} \subset \forall j \text{ such that } 1 \leqslant j < n, \text{list}(j) \leqslant \text{list}(j+1)$$

用 Prolog 来写这个算法是很直接的。例如，sorted 子目标可以表示为：

```
sorted ([]).
sorted ([x]).
```

```
sorted ([x, y | list]) :- x <= y, sorted ([y | list]).
```

这一排序过程的问题是它不知道如何排序，只知道简单地枚举给定表的所有排列，直到碰巧创建一个有序排列的表——实际上是一个非常缓慢的过程。

至今为止，没有人找到一个过程可以将有序表的描述转换为某种高效的排序算法。归结可用于许多有趣的东西，但显然不能用于排序。因此，对表排序的 Prolog 程序必须说明排序如何实现的细节，就像命令式语言或函数式语言那样。

所有这些问题都意味着逻辑程序设计应该抛弃吗？绝对不是！实际上，逻辑程序设计能够完成许多有用的应用，而且它基于一种吸引人的概念，从而本身也很有趣。最后，有可能提出新的推理技术，使逻辑程序设计语言系统能高效地处理日益增多的各类问题。

16.8　逻辑程序设计的应用

本节简要地介绍逻辑程序设计，特别是 Prolog 的一些现有的和潜在的应用。

16.8.1　关系数据库管理系统

关系数据库管理系统(RDBMS)以表的形式存储数据。这种数据库上的查询通常用结构化查询语言(Structured Query Language，SQL)来表示。SQL 是非过程式的，与逻辑程序设计是非过程式的意义一样，用户并不说明如何获得答案，只需要说明答案的特点。逻辑程序设计与 RDBMS 的联系是显然的。简单的信息表可以用 Prolog 结构来表示，表之间的关系可以方便且容易地用 Prolog 规则来表示；检索过程是归结操作所固有的；Prolog 的目标语句提供了对 RDBMS 的查询。所以，逻辑程序设计自然地与 RDBMS 的实现需求相匹配。

采用逻辑程序设计来实现 RDBMS 的优点之一是只需要一种语言。在典型的 RDBMS 中，数据库语言包含数据定义语句、数据操作语句和查询语句，它们都嵌入在一种通用程序设计语言(如 COBOL)中。通用语言用于处理数据和输入输出函数。所有这些函数都可以在逻辑程序设计语言中实现。

使用逻辑程序设计来实现 RDBMS 的另一个优点是内置了推理功能。传统的 RDBMS 无法从数据库中推理出非显式存储的东西，它们只包含事实，而不是事实和推理规则。与传统 RDBMS 相比，使用逻辑程序设计实现 RDBMS 的主要缺点，是逻辑程序设计的实现速度比较慢。逻辑推理比通常采用命令式程序设计技术的查表方法，要耗费更长的时间。

16.8.2　专家系统

专家系统是设计模拟某一特定领域的人类专家的计算机系统，由事实数据库、推理过程、关于该领域的一些启发式方法，以及友好的人机接口(使系统很像一个专业的人类顾问)构成。在人类专家所提供的初始知识以外，专家系统在使用的过程中学习，因此其数据库必须能够动态增长。而且，专家系统应该有能力在确定需要额外信息时，向用户询问获取。

专家系统的设计者的中心问题之一，是处理数据库中不可避免的不一致性和不完备性。

逻辑程序设计看起来很适于处理这些问题。例如，默认的推理规则有助于处理不完备性问题。

Prolog 可以并且已经用于构建专家系统。它使用归结作为查询过程的基础，使用添加事实和规则的能力来提供学习功能，使用跟踪工具来告诉用户给定结果背后的推理过程，很容易满足专家系统的基本要求。Prolog 所没有的，是在需要时系统自动向用户询问额外信息的功能。

将逻辑程序设计用于专家系统最广为人知的工作之一，就是称为 APES 的构建专家系统的系统，在(Sergot，1983)和(Hammond，1983)中都有介绍。APES 系统包含一个非常灵活的工具，用于在构建专家系统时从用户那里收集信息。它还包含第二个解释器，用于产生查询答案的解释。

APES 已经成功地用来生成一些专家系统，其中一个专家系统用于政府社会福利计划的规则，一个专家系统用于英国国籍法(英国国籍规则的权威资料)。

16.8.3 自然语言处理

某些类别的自然语言处理可以用逻辑程序设计来实现。特别地，计算机软件系统(例如智能数据库和其他基于知识的智能系统)的自然语言接口，可以方便地用逻辑程序设计来实现。人们发现，对于描述语言语法来说，逻辑程序设计的形式等价于上下文无关文法，逻辑程序设计系统中的证明过程，等价于某些语法分析策略。实际上，后向链接的归结可直接用于分析由上下文无关文法描述结构的句子。人们还发现，自然语言的某些语义用逻辑程序设计来为语言建模会很清楚，特别是基于逻辑的语义网络研究表明，自然语言的一些句子集合可以用子句形式来表示(Deliyanni 和 Kowalski，1979)。(Kowalski，1979)也介绍了基于逻辑的语义网络。

小 结

符号逻辑为逻辑程序设计和逻辑程序设计语言提供了基础。逻辑程序设计的方法，将事实和声明事实间关系的规则用作数据库，用自动推理过程来检查新命题的正确性(假设数据库中的事实和规则为真)。这种方法是为了自动定理证明而提出的。

Prolog 是应用最广泛的逻辑程序设计语言。逻辑程序设计的起源是 Robinson 提出的用于逻辑推理的归结规则。Prolog 主要是由马赛的 Colmeraur 和 Roussel，在爱丁堡的 Kowalski 的帮助下开发的。

逻辑程序应该是非过程式的，意思是给出答案的特点，但不给出得到答案的完整过程。

Prolog 语句包括事实语句、规则语句和目标语句，大部分都由结构(原子命题)和逻辑运算符组成(虽然也允许用算术表达式)。

归结是 Prolog 解释器的主要行为。这一过程大量使用回溯，主要包括命题间的模式匹配。当涉及变量时，变量可以实例化为值来获得匹配。这种实例化过程称为合一。

当前的逻辑程序设计有一些问题。为了高效地进行推理，甚至是为了避免无限循环，程序员必须经常在程序中声明控制流信息。另外还有封闭世界假设和否定的问题。

逻辑程序设计已经用于一些不同的领域，主要是关系数据库系统、专家系统和自然语言处理。

文献注释

在好几本书中都介绍了 Prolog 语言。(Clocksin 和 Mellish，2003)中涉及了 Prolog 的爱丁堡形式。(Clark 和 McCabe，1984)中介绍了 Prolog 在微机上的实现。

(Hogger，1991)是一本逻辑程序设计通用领域的优秀图书，它是本章关于逻辑程序设计应用一节的材料来源。

复 习 题

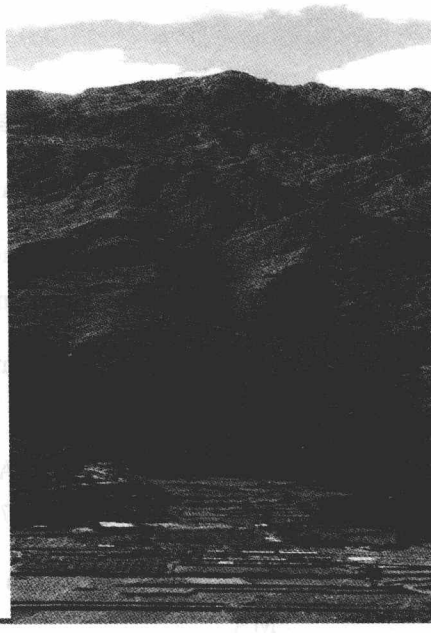
1. 在形式逻辑中，符号逻辑的三个基本用途是什么？
2. 复合项的两个部分是什么？
3. 声明命题的两种模式是什么？
4. 子句形式的命题的一般形式是什么？
5. 什么是前提?什么是结论？
6. 给出归结和合一的一般性(非严格)定义。
7. Horn 子句的形式是什么？
8. 声明语义的基本概念是什么？
9. 语言是非过程式的，是什么意思？
10. Prolog 项的三种形式是什么？
11. 什么是未实例化的变量？
12. 在 Prolog 中事实语句和规则语句的语法形式与用法是什么？
13. 什么是合取？
14. 说明将目标与数据库中事实相匹配的两种方法。
15. 当讨论如何满足多个目标时，说明深度优先搜索和宽度优先搜索的区别。
16. 说明回溯在 Prolog 中是如何工作的。
17. 说明 Prolog 语句 K is K+1 有什么问题。
18. 在归结过程中 Prolog 程序员控制模式匹配顺序的两种方法是什么？
19. 说明 Prolog 中的生成与检测策略是什么。
20. 说明 Prolog 使用的封闭世界假设是什么。为什么这是一种局限性？
21. 说明 Prolog 的否定问题是什么。为什么这是一种局限性？
22. 说明自动定理证明和 Prolog 推理过程的联系。
23. 说明过程式语言和非过程式语言的区别。
24. 说明为什么 Prolog 系统必须回溯。
25. 在 Prolog 中归结和合一的关系是什么？

习 题

1. 比较 Ada 中数据类型和 Prolog 中数据类型的概念。
2. 描述如何使用多处理器的计算机来实现归结。当前定义的 Prolog 可以用这一方法吗？
3. 用 Prolog 描述你的家谱(仅基于事实)，上溯到你的祖父母并包含所有的后代。一定要包含所有的亲戚。
4. 写出一组家庭关系的规则，包括从祖父母开始往下两代的所有关系。现在将这些关系加到习题 3 的事实上，删除尽可能多的事实。
5. 用 Prolog 有首的 Horn 子句写出以下英文的条件语句：
   a. 如果 Fred 是 Mike 的父亲(father)，那么 Fred 是 Mike 的先辈(ancestor)。
   b. 如果 Mike 是 Joe 的父亲(father)，且 Mike 是 Mary 的父亲(father)，那么 Mary 是 Joe 的姐妹(sister)。
   c. 如果 Mike 是 Fred 的兄弟(brother)，且 Fred 是 Mary 的父亲(father)，那么 Mike 是 Mary 的叔叔(uncle)。
6. 说明 Scheme 与 Prolog 的表处理功能在哪两方面相似。
7. Scheme 与 Prolog 的表处理功能有什么不同？
8. 比较 Prolog 与 ML，写出两个相似点和两个不同点。
9. 根据一本 Prolog 的书，学习并描述出现-检查(occur-check)问题。为什么 Prolog 允许在它的实现中出现这个问题？
10. 找一份有关 Skolem 范式信息的材料，简短、清晰地说明 Skolem 范式。

编程练习

1. 使用结构 parent(X, Y)、male(X)和 female(X)，写出一个定义 mother(X, Y)的结构。
2. 使用结构 parent(X, Y)、male(X)和 female(X)，写出一个定义 sister(X, Y)的结构。
3. 编写一个 Prolog 程序，找到一列数中的最大数。
4. 编写一个 Prolog 程序，如果两个给定表参数的交集为空，就返回真。
5. 编写一个 Prolog 程序，返回一个包含两个给定表元素的并集的表。
6. 编写一个 Prolog 程序，返回给定表的最后一个元素。
7. 编写一个 Prolog 程序，实现快速排序。



参考文献

ACM. (1979) "Part A: Preliminary Ada Reference Manual" and "Part B: Rationale for the Design of the Ada Programming Language." SIGPLAN Notices, Vol. 14, No. 6.

ACM. (1993a) History of Programming Language Conference Proceedings. ACM SIGPLAN Notices, Vol. 28, No. 3, March.

ACM. (1993b) "High Performance FORTRAN Language Specification Part 1." FORTRAN Forum,Vol. 12, No. 4.

Aho, A. V., M. S. Lam, R. Sethi, and J. D. Ullman. (2006) Compilers: Principles, Techniques, and Tools. 2e, Addison-Wesley, Reading, MA.

Aho, A. V., B. W. Kernighan, and P. J. Weinberger. (1988) The AWK Programming Language. Addison-Wesley, Reading, MA.

Andrews, G. R., and F. B. Schneider. (1983) "Concepts and Notations for Concurrent Programming." ACM Computing Surveys, Vol. 15, No. 1,pp. 3 - 43.

ANSI. (1966) American National Standard Programming Language FORTRAN. American National Standards Institute, New York.

ANSI. (1976) American National Standard Programming Language PL/I. ANSI X3.53 - 1976. American National Standards Institute, New York.

ANSI. (1978a) American National Standard Programming Language FORTRAN. ANSI X3.9 - 1978. American National Standards Institute, New York.

ANSI. (1978b) American National Standard Programming Language Minimal BASIC. ANSI X3. 60 - 1978. American National Standards Institute,New York.

ANSI. (1985) American National Standard Programming Language COBOL. ANSI

X3.23 - 1985. American National Standards Institute, New York.

ANSI. (1989) American National Standard Programming Language C. ANSI X3.159 - 1989. American National Standards Institute, New York.

ANSI. (1992) American National Standard Programming Language FORTRAN 90. ANSI X3. 198 - 1992. American National Standards Institute, New York.

Arden, B. W., B. A. Galler, and R. M. Graham. (1961) "MAD at Michigan." Datamation, Vol. 7, No. 12, pp. 27 - 28.

ARM. (1995) Ada Reference Manual. ISO/IEC/ANSI 8652:19. Intermetrics, Cambridge, MA.

Arnold, K., J. Gosling, and D. Holmes (2006) The Java (TM) Programming Language, 4e. Addison-Wesley, Reading, MA.

Backus, J. (1954) "The IBM 701 Speedcoding System." J. ACM, Vol. 1, pp. 4 - 6.

Backus, J. (1959) "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference." Proceedings International Conference on Information Processing. UNESCO, Paris, pp. 125 - 132.

Backus, J. (1978) "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." Commun. ACM, Vol. 21, No. 8, pp. 613 - 641.

Backus, J., F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. (1963) "Revised Report on the Algorithmic Language ALGOL 60." Commun. ACM, Vol. 6, No. 1, pp. 1 - 17.

Balena, F. (2003) Programming Microsoft Visual Basic .NET Version 2003, Microsoft Press, Redmond, WA.

Ben-Ari, M. (1982) Principles of Concurrent Programming. Prentice-Hall, Englewood Cliffs, NJ.

Birtwistle, G. M., O.-J. Dahl, B. Myhrhaug, and K. Nygaard. (1973)Simula BEGIN. Van Nostrand Reinhold, New York.

Bodwin, J. M., L. Bradley, K. Kanda, D. Litle, and U. F. Pleban. (1982) "Experience with an Experimental Compiler Generator Based on Denotational Semantics." ACM SIGPLAN Notices, Vol. 17, No. 6, pp. 216 - 229.

Bohm, C., and G. Jacopini. (1966) "Flow Diagrams, Turing Machines, and Languages

with Only Two Formation Rules." Commun. ACM, Vol. 9, No. 5, pp. 366 - 371.

Bolsky, M., and D. Korn. (1995) The New KornShell Command and Programming Language. Prentice-Hall, Englewood Cliffs, NJ.

Booch, G. (1987) Software Engineering with Ada, 2e. Benjamin/Cummings, Redwood City, CA.

Bradley, J. C. (1989) QuickBASIC and QBASIC Using Modular Structures. W. C. Brown, Dubuque, IA.

Brinch Hansen, P. (1973) Operating System Principles. Prentice-Hall,Englewood Cliffs, NJ.

Brinch Hansen, P. (1975) "The Programming Language Concurrent-Pascal." IEEE Transactions on Software Engineering, Vol. 1, No. 2, pp. 199 - 207.

Brinch Hansen, P. (1977) The Architecture of Concurrent Programs. Prentice-Hall, Englewood Cliffs, NJ.

Brinch Hansen, P. (1978) "Distributed Processes: A Concurrent Programming Concept." Commun. ACM, Vol. 21, No. 11, pp. 934 - 941.

Brown, J. A., S. Pakin, and R. P. Polivka. (1988) APL2 at a Glance.Prentice-Hall, Englewood Cliffs, NJ.

Campione, M., K. Walrath, and A. Huml. (2001) The Java Tutorial, 3e. Addison-Wesley, Reading, MA.

Cardelli, L., J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. (1989) Modula-3 Report (revised). Digital System Research Center, Palo Alto, CA.

Chambers, C., and D. Ungar. (1991) "Making Pure Object-Oriented Languages Practical." SIGPLAN Notices, Vol. 26, No. 1, pp. 1 - 15.

Chomsky, N. (1956) "Three Models for the Description of Language." IRE Transactions on Information Theory, Vol. 2, No. 3, pp. 113 - 124.

Chomsky, N. (1959) "On Certain Formal Properties of Grammars." Information and Control, Vol. 2, No. 2, pp. 137 - 167.

Church, A. (1941) Annals of Mathematics Studies. Volume 6: Calculi of Lambda Conversion. Princeton Univ. Press, Princeton, NJ. Reprinted by Klaus Reprint Corporation, New York, 1965.

Clark, K. L., and F. G. McCabe. (1984) Micro-PROLOG: Programming in Logic. Prentice-Hall, Englewood Cliffs, NJ.

Clarke, L. A., J. C. Wileden, and A. L. Wolf. (1980) "Nesting in Ada Is for the Birds." ACM SIGPLAN Notices, Vol. 15, No. 11, pp. 139 - 145.

Cleaveland, J. C. (1986) An Introduction to Data Types. Addison-Wesley,Reading, MA.

Cleaveland, J. C., and R. C. Uzgalis. (1976) Grammars for Programming Languages: What Every Programmer Should Know About Grammar.American Elsevier, New York.

Clocksin, W. F., and C. S. Mellish. (2003) Programming in Prolog, 5e. Springer-Verlag, New York.

Cohen, J. (1981) "Garbage Collection of Linked Data Structures." ACM Computing Surveys, Vol. 13, No. 3, pp. 341 - 368.

Converse, T., and J. Park. (2000) PHP 4 Bible. IDG Books, New York.

Conway, M. E. (1963). "Design of a Separable Transition-Diagram Compiler." Commun. ACM, Vol. 6, No. 7, pp. 396 - 408.

Conway, R., and R. Constable. (1976) "PL/CS—A Disciplined Subset of PL/I." Technical Report TR76/293. Department of Computer Science, Cornell University, Ithaca, NY.

Cornell University. (1977) PL/C User' s Guide, Release 7.6. Department of Computer Science, Cornell University, Ithaca, NY.

Correa, N. (1992) "Empty Categories, Chain Binding, and Parsing." pp. 83 - 121, Principle-Based Parsing. Eds. R. C. Berwick, S. P. Abney, and C. Tenny. Kluwer Academic Publishers,Boston.

Cousineau, G, M.Mauny, and K. Callaway. (1998) The Functional Approach to Programming. Cambridge University Press,

Dahl, O.-J., E. W. Dijkstra, and C. A. R. Hoare. (1972) StructuredProgramming. Academic Press, New York.

Dahl, O.-J., and K. Nygaard. (1967) "SIMULA 67 Common Base Proposal." Norwegian Computing Center Document, Oslo.

Deitel, H. M., D. J. Deitel, and T. R. Nieto. (2002) Visual BASIC .Net: How to Program, 2e. Prentice-Hall, Inc. Upper Saddle River, NJ.

Deliyanni, A., and R. A. Kowalski. (1979) "Logic and Semantic Networks." Commun. ACM, Vol. 22, No. 3, pp. 184 - 192.

Department of Defense. (1960) "COBOL, Initial Specifications for a Common Business Oriented Language." U.S. Department of Defense, Washington, D.C.

Department of Defense. (1961) "COBOL—1961, Revised Specifications for a Common Business Oriented Language." U.S. Department of Defense, Washington, D.C.

Department of Defense. (1962) "COBOL—1961 EXTENDED, Extended Specifications for a Common Business Oriented Language." U.S. Department of Defense, Washington, D.C.

Department of Defense. (1975a) "Requirements for High Order Programming Languages, STRAWMAN." July. U.S. Department of Defense,Washington, D.C.

Department of Defense. (1975b) "Requirements for High Order Programming Languages, WOODENMAN." August. U.S. Department of Defense,Washington, D.C.

Department of Defense. (1976)"Requirements for High Order Programming Languages, TINMAN." June. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1977)"Requirements for High Order Programming Languages, IRONMAN." January. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1978)"Requirements for High Order Programming Languages, STEELMAN." June. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1980a) "Requirements for High Order Programming Languages, STONEMAN." February. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1980b) "Requirements for the Programming Environment for the Common High Order Language, STONEMAN."U.S. Department of Defense, Washington, D.C.

DeRemer, F. (1971) "Simple LR(k) Grammars." Commun. ACM, Vol. 14, No. 7, pp. 453 - 460.

DeRemer, F., and T. Pennello. (1982) "Efficient Computation of LALR(1) Look-Ahead Sets." ACM TOPLAS, Vol. 4, No. 4, pp. 615 - 649.

Deutsch, L. P., and D. G. Bobrow. (1976) "An Efficient Incremental Automatic Garbage Collector." Commun. ACM, Vol. 11, No. 3, pp. 522 - 526.

Dijkstra, E. W. (1968a) "Goto Statement Considered Harmful." Commun.ACM, Vol.

11, No. 3, pp. 147 - 149.

Dijkstra, E. W. (1968b) "Cooperating Sequential Processes." In Programming Languages, F. Genuys (ed.). Academic Press, New York, pp. 43 - 112.

Dijkstra, E. W. (1972) "The Humble Programmer." Commun. ACM, Vol.15, No. 10, pp. 859 - 866.

Dijkstra, E. W. (1975) "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs." Commun. ACM, Vol. 18, No. 8, pp. 453 - 457.

Dijkstra, E. W. (1976). A Discipline of Programming. Prentice-Hall, Englewood Cliffs, NJ.

Dybvig, R. K. (2003) The Scheme Programming Language, 3e. MIT Press, Boston.

Ellis, M. A., and B. Stroustrup (1990) The Annotated C++ Reference Manual. Addison-Wesley, Reading, MA.

Farber, D. J., R. E. Griswold, and I. P. Polonsky. (1964) "SNOBOL, aString Manipulation Language." J. ACM, Vol. 11, No. 1, pp. 21 - 30.

Farrow, R. (1982) "LINGUIST 86: Yet Another Translator Writing SystemBased on Attribute Grammars." ACM SIGPLAN Notices, Vol. 17, No. 6, pp. 160 - 171.

Fischer, C. N., G. F. Johnson, J. Mauney, A. Pal, and D. L. Stock. (1984) "The Poe Language-Based Editor Project." ACM SIGPLAN Notices,Vol. 19, No. 5, pp. 21 - 29.

Fischer, C. N., and R. J. LeBlanc. (1977) "UW-Pascal Reference Manual." Madison Academic Computing Center, Madison, WI.

Fischer, C.N., and R. J. LeBlanc. (1980) "Implementation of Runtime Diagnostics in Pascal." IEEE Transactions on Software Engineering, SE-6, No. 4, pp. 313 - 319.

Fischer, C. N., and R. J. LeBlanc. (1991) Crafting a Compiler in C. Benjamin/Cummings, Menlo Park, CA.

Flanagan, D. (2002) JavaScript: The Definitive Guide, 4e. O' Reilly Media, Sebastopol, CA

Floyd, R. W. (1967) "Assigning Meanings to Programs." Proceedings Symposium Applied Mathematics. Mathematical Aspects of Computer Science Ed. J. T. Schwartz. American Mathematical Society, Providence, RI.

Frege, G. (1892) "Über Sinn und Bedeutung." Zeitschrift für Philosophie und Philosophisches Kritik, Vol. 100, pp. 25 - 50.

Friedl, J. E. F. (2006) Mastering Regular Expressions, 3e. O' Reilly Media, Sebastopol, CA.

Friedman, D. P., and D. S. Wise. (1979) "Reference Counting' s Abilityto Collect Cycles Is Not Insurmountable." Information Processing Letters, Vol. 8, No. 1, pp. 41 - 45.

Fuchi, K. (1981) "Aiming for Knowledge Information Processing Systems." Proceedings of the International Conference on Fifth Generation ComputingSystems. Japan Information Processing Development Center, Tokyo. Republished (1982) by North-Holland Publishing, Amsterdam.

Gehani, N. (1983) Ada: An Advanced Introduction. Prentice-Hall,Englewood Cliffs, NJ.

Gilman, L., and A. J. Rose. (1976) APL: An Interactive Approach,2e. J. Wiley, New York.

Goldberg, A., and D. Robson. (1983) Smalltalk-80: The Language and Its Implementation. Addison-Wesley, Reading, MA.

Goldberg, A., and D. Robson. (1989) Smalltalk-80: The Language. Addison-Wesley, Reading, MA.

Goodenough, J. B. (1975) "Exception Handling: Issues and Proposed Notation." Commun. ACM, Vol. 18, No. 12, pp. 683 - 696.

Goos, G., and J. Hartmanis (eds.) (1983) The Programming Language Ada Reference Manual. American National Standards Institute. ANSI/MIL-STD-1815A - 1983. Lecture Notes inComputer Science 155. Springer-Verlag, New York.

Gordon, M. (1979) The Denotational Description of Programming Languages, An Introduction. Springer-Verlag, Berlin - New York.

Graham, P. (1996) ANSI Common LISP. Prentice-Hall, Englewood Cliffs, NJ.

Gries, D. (1981) The Science of Programming. Springer-Verlag, New York.

Griswold, R. E., and M. T. Griswold. (1983) The ICON Programming Language. Prentice-Hall, Englewood Cliffs, NJ.

Griswold, R. E., F. Poage, and I. P. Polonsky. (1971) The SNOBOL 4 Programming Language, 2e. Prentice-Hall, Englewood Cliffs, NJ.

Halstead, R. H., Jr. (1985) "Multilisp: A Language for Concurrent Symbolic Computation." ACM Transactions on Programming Language and Systems, Vol. 7, No. 4, October 1985, pp. 501-538.

Hammond, P. (1983) APES: A User Manual. Department of ComputingReport 82/9. Imperial College of Science and Technology, London.

Harbison, S. P. III, and G. L. Steele, Jr. (2002) A. C. Reference Manual, 5e, Prentice-Hall, Upper Saddle River, NJ.

Henderson, P. (1980) Functional Programming: Application and Implementation. Prentice-Hall, Englewood Cliffs, NJ.

Hoare, C. A. R. (1969) "An Axiomatic Basis of Computer Programming." Commun. ACM, Vol. 12, No. 10, pp. 576 - 580.

Hoare, C. A. R. (1972) "Proof of Correctness of Data Representations." Acta Informatica, Vol. 1, pp. 271 - 281.

Hoare, C. A. R. (1973) "Hints on Programming Language Design." Proceedings ACM SIGACT/SIGPLAN Conference on Principles of ProgrammingLanguages. Also published as Technical Report STAN-CS-73-403, Stanford University Computer Science Department.

Hoare, C. A. R. (1974) "Monitors: An Operating System Structuring Concept." Commun. ACM, Vol. 17, No. 10, pp. 549 - 557.

Hoare, C. A. R. (1978) "Communicating Sequential Processes." Commun.ACM, Vol. 21, No. 8, pp. 666 - 677.

Hoare, C. A. R. (1981) "The Emperor' s Old Clothes." Commun. ACM,Vol. 24, No. 2, pp. 75 - 83.

Hoare, C. A. R., and N. Wirth. (1973) "An Axiomatic Definition of the Programming Language Pascal." Acta Informatica, Vol. 2, pp. 335 - 355.

Hogger, C. J. (1984) Introduction to Logic Programming. Academic Press, London.

Hogger, C. J. (1991) Essentials of Logic Programming. Oxford Science Publications, Oxford, England.

Holt, R. C., G. S. Graham, E. D. Lazowska,and M. A. Scott. (1978)Structured Concurrent Programming with Operating Systems Applications. Addison-Wesley,Reading, MA.

Horn, A. (1951) "On Sentences Which Are True of Direct Unions of Algebras." J. Symbolic Logic, Vol. 16, pp. 14 - 21.

Hudak, P., and J. Fasel. (1992) "A Gentle Introduction to Haskell, ACMSIGPLAN Notices, 27(5), May 1992, pp. T1 - T53.

Hughes, (1989) "Why Functional Programming Matters", The Computer Journal, Vol. 32, No. 2, pp. 98 - 107.

Huskey, H. K., R. Love, and N. Wirth. (1963) "A Syntactic Description of BC NELIAC." Commun. ACM, Vol. 6, No. 7, pp. 367 - 375.

IBM. (1954)"Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN." IBM Corporation, New York.

IBM. (1956) "Programmer' s Reference Manual, The FORTRAN Automatic Coding System for the IBM 704 EDPM." IBM Corporation, New York.

IBM. (1964) "The New Programming Language." IBM UK Laboratories.

Ichbiah, J. D., J. C. Heliard, O. Roubine, J. G. P.Barnes, B. Krieg-Brueckner, and B. A. Wichmann. (1979) "Rationale for theDesign of the Ada Programming Language." ACM SIGPLAN Notices, Vol. 14,No. 6, Part B.

IEEE. (1985) "Binary Floating-Point Arithmetic." IEEE Standard 754,IEEE, New York.

Ierusalimschy, R. (2006) Programming in Lua, 2e, Lua.org, Rio deJaneiro, Brazil.

INCITS/ISO/IEC (1997) 1539-1-1997 Information Technology—Programming Languages—FORTRAN Part 1: Base Language. American National Standards Institute, New York.

Ingerman, P. Z. (1967). "Panini-Backus Form Suggested." Commun. ACM, Vol. 10, No. 3, p. 137.

Intermetrics. (1993) Programming Language Ada, Draft, Version 4.0.Cambridge, MA.

ISO. (1982) Specification for Programming Language Pascal. ISO7185 - 1982. International Organization for Standardization, Geneva, Switzerland.

ISO/IEC (1996) 14977:1996, Information Technology—SyntacticMetalanguage—Extended BNF. International Organization for Standardization, Geneva, Switzerland.

ISO. (1998) ISO14882-1, ISO/IEC Standard - Information Technology—Programming Language—C++. International Organization for Standardization,Geneva, Switzerland.

ISO. (1999) ISO/IEC 9899:1999, Programming Language C. AmericanNational Standards Institute, New York.

ISO/IEC (2002) 1989:2002 Information Technology—Programming Languages—COBOL. American National Standards Institute, New York.

ISO/IEC (2010) 1539-1 Information Technology—Programming Languages—Fortran. American National Standards Institute, New York.

Iverson, K. E. (1962) A Programming Language. John Wiley, New York.

Jensen, K., and N. Wirth. (1974) Pascal Users Manual and Report.Springer-Verlag, Berlin.

Johnson, S. C. (1975) "Yacc—Yet Another Compiler Compiler." Computing Science Report 32. AT&T Bell Laboratories, Murray Hill, NJ.

Jones, N. D. (ed.) (1980) Semantic-Directed Compiler Generation. LectureNotes in Computer Science, Vol. 94. Springer-Verlag, Heidelberg, FRG.

Kay, A. (1969) The Reactive Engine. PhD Thesis. University of Utah,September.

Kernighan, B. W., and D. M. Ritchie. (1978) The C ProgrammingLanguage. Prentice-Hall, Englewood Cliffs, NJ.

Knuth, D. E. (1965) "On the Translation of Languages from Left to Right." Information & Control, Vol. 8, No. 6, pp. 607 - 639.

Knuth, D. E. (1967) "The Remaining Trouble Spots in ALGOL 60." Commun. ACM, Vol. 10, No. 10, pp. 611 - 618.

Knuth, D. E. (1968a) "Semantics of Context-Free Languages." Mathem-atical Systems Theory, Vol. 2, No. 2, pp. 127 - 146.

Knuth, D. E. (1968b) The Art of Computer Programming, Vol. I, 2e. Addison-Wesley, Reading, MA.

Knuth, D. E. (1974) "Structured Programming with GOTO Statements." ACM Computing Surveys, Vol. 6, No. 4, pp. 261 - 301.

Knuth, D. E. (1981) The Art of Computer Programming, Vol. II, 2e. Addison-Wesley, Reading, MA.

Knuth, D. E., and L. T. Pardo. (1977) "Early Development of Programming Languages." In Encyclopedia of Computer Science and Technology,Vol. 7. Dekker, New York, pp. 419 - 493.

Kochan, S. G. (2009) Programming in Objective-C 2.0. Addison-Wesley,Upper Saddle River, NJ.

Kowalski, R. A. (1979) Logic for Problem Solving. Artificial Intelligence Series, Vol. 7. Elsevier-North Holland, New York.

Laning, J. H., Jr., and N. Zierler. (1954) "A Program for Translation of Mathematical Equations for Whirlwind I." Engineering memorandum E-364. Instrumentation Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Ledgard, H. (1984) The American Pascal Standard. Springer-Verlag, New York.

Ledgard, H. F., and M. Marcotty. (1975) "A Genealogy of Control Structures." Commun. ACM, Vol. 18, No. 11, pp. 629 - 639.

Lischner, R. (2000) Delphi in a Nutshell. O' Reilly Media, Sebastopol, CA.

Liskov, B., R. L. Atkinson, T. Bloom, J.E.B. Moss, C. Scheffert, R. Scheifler, and A. Snyder (1981) "CLU Reference Manual." Springer, New York.

Liskov, B., and A. Snyder. (1979) "Exception Handling in CLU." IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, pp. 546 - 558.

Lomet, D. (1975) "Scheme for Invalidating References to Freed Storage." IBM J. of Research and Development, Vol. 19, pp. 26 - 35.

Lutz, M., and D. Ascher. (2004) Learning Python, 2e. O' Reilly Media, Sebastopol, CA.

MacLaren, M. D. (1977) "Exception Handling in PL/I." ACM SIGPLAN Notices, Vol. 12, No. 3, pp. 101 - 104.

Marcotty, M., H. F. Ledgard, and G. V. Bochmann. (1976) "A Sampler of Formal Definitions." ACM Computing Surveys, Vol. 8, No. 2, pp. 191 - 276.

Mather, D. G., and S. V. Waite (eds.) (1971) BASIC. 6e. University Press of New England, Hanover, NH.

McCarthy, J. (1960) "Recursive Functions of Symbolic Expressions and Their Computatio n by Machine, Part I." Commun. ACM, Vol. 3, No. 4,pp. 184 - 195.

McCarthy, J., P. W. Abrahams, D. J. Edwards, T. P. Hart, and M.Levin. (1965) LISP 1.5 Programmer' s Manual, 2e. MIT Press, Cambridge,MA.

McCracken, D. (1970) "Whither APL." Datamation, Sept. 15, pp. 53 - 57.

Metcalf, M., J. Reid, and M. Cohen. (2004) Fortran 95/2003 Explained,3e. Oxford University Press, Oxford, England.

Meyer, B. (1990) Introduction to the Theory of Programming Languages.Prentice-Hall, Englewood Cliffs, NJ.

Meyer, B. (1992) Eiffel: The Language. Prentice-Hall, Englewood Cliffs, NJ.

Microsoft. (1991) Microsoft Visual Basic Language Reference. DocumentDB20664-0491, Redmond, WA.

Milner, R., M. Tofte, and R. Harper. (1990) The Definition of Standard ML. MIT Press, Cambridge, MA.

Milos, D., U. Pleban, and G. Loegel. (1984) "Direct Implementation ofCompiler Specifications." ACM Principles of Programming Languages 1984,pp. 196 - 202.

Mitchell, J. G., W. Maybury, and R. Sweet. (1979) Mesa Language Manual, Version 5.0, CSL-79-3. Xerox Research Center, Palo Alto, CA.

Moss, C. (1994) Prolog++: The Power of Object-Oriented and Logic Programming. Addison-Wesley, Reading, MA.

Moto-oka, T. (1981) "Challenge for Knowledge Information Processing Systems." Proceedings of the International Conference on Fifth Generation Computing Systems. Japan Information Processing Development Center, Tokyo. Republished (1982) by North-Holland Publishing, Amsterdam.

Naur, P. (ed.) (1960) "Report on the Algorithmic Language ALGOL 60." Commun. ACM, Vol. 3, No. 5, pp. 299 - 314.

Newell, A., and H. A. Simon. (1956) "The Logic Theory Machine—A Complex Information Processing System." IRE Transactions on Information Theory, Vol. IT-2, No. 3, pp. 61 - 79.

Newell, A., and F. M. Tonge. (1960)"An Introduction to Information Processing Language V." Commun. ACM, Vol. 3, No. 4, pp. 205 - 211.

Nilsson, N. J. (1971) Problem Solving Methods in Artificial Intelligence.McGraw-Hill, New York.

Ousterhout, J. K. (1994) Tcl and the Tk Toolkit. Addison-Wesley, Reading, MA.

Paepcke, E. (editor). Object-Oriented Programming: The CLOS Perspective. MIT Press, 1993.

Pagan, F. G. (1981) Formal Specifications of Programming Languages.Prentice-Hall, Englewood Cliffs, NJ.

Papert, S. (1980) MindStorms: Children, Computers and Powerful Ideas.Basic Books, New York.

Perlis, A., and K. Samelson. (1958) "Preliminary Report—International Algebraic Language." Commun. ACM, Vol. 1, No. 12, pp. 8 - 22.

Peyton Jones, S. L. (1987) The Implementation of Functional Programming Languages. Prentice-Hall, Englewood Cliffs, NJ.

Pratt, T. W. (1984) Programming Languages: Design and Implementation, 2e. Prentice-Hall, Englewood Cliffs, NJ.

Pratt, T. W., and M. V. Zelkowitz (2001) Programming Languages:Design and Imple mentation, 4e. Prentice-Hall, Englewood Cliffs, NJ.

Raymond, E. (2004) Art of UNIX Programming. Addison Wesley, Boston.

Remington-Rand. (1952) "UNIVAC Short Code." Unpublished collection of dittoed notes. Preface by A. B. Tonik, dated October 25, 1955 (1 p.);Preface by J. R. Logan, undated but apparently from 1952 (1 p.); Preliminary exposition, 1952? (22 pp., where in which pp. 20 - 22 appear to be a laterreplacement); Short code supplementary information, topic one (7 pp.); Addenda #1, 2, 3, 4 (9 pp.).

Reppy, J. H. (1999) Concurrent Programming in ML. Cambridge University Press, New York.

Richards, M. (1969) "BCPL: A Tool for Compiler Writing and SystemsProgramming." Proc. AFIPS SJCC, Vol. 34, pp. 557 - 566.

Robinson, J. A. (1965) "A Machine-Oriented Logic Based on the Resolution Principle." Journal of the ACM, Vol. 12, pp. 23 - 41.

Romanovsky, A. and B. Sandén (2001) "Except for Exception Handling," Ada Letters, Vol. 21, No. 3, September 2001, pp. 19 - 25.

Roussel, P. (1975) "PROLOG: Manual de Reference et D' utilisation." Research Report. Artificial Intelligence Group, Univ. of Aix-Marseille, Luming, France.

Rubin, F. (1987) "'GOTO Statement Considered Harmful' considered harmful" (letter to editor). Commun. ACM, Vol. 30, No. 3, pp. 195 - 196.

Rutishauser, H. (1967) Description of ALGOL 60. Springer-Verlag, New York.

Sammet, J. E. (1969) Programming Languages: History and Fundamentals.Prentice-Hall, Englewood Cliffs, NJ.

Sammet, J. E. (1976) "Roster of Programming Languages for 1974 - 75." Commun.

ACM, Vol. 19, No. 12, pp. 655 - 669.

Schneider, D. I. (1999) An Introduction to Programming Using VisualBASIC 6.0. Prentice-Hall, Englewood Cliffs, NJ.

Schorr, H., and W. Waite. (1967) "An Efficient Machine Independent Procedure for Garbage Collection in Various List Structures." Commun. ACM,Vol. 10, No. 8, pp. 501 - 506.

Scott, D. S., and C. Strachey. (1971) "Towards a Mathematical Semantics for Computer Language." In Proceedings, Symposium on Computers and Automation, J. Fox (ed.). Polytechnic Institute of Brooklyn Press, New York,pp. 19 - 46.

Scott, M. (2000) Programming Language Pragmatics, Morgan Kaufman,San Francisco, CA.

Sebesta, R. W. (1991) VAX Structured Assembly Language Programming,2e. Benjamin/ Cummings, Redwood City, CA.

Sergot, M. J. (1983) "A Query-the-User Facility for Logic Programming." In Integrated Interactive Computer Systems, P. Degano and E. Sandewall (eds.). North-Holland Publishing, Amsterdam.

Shaw, C. J. (1963) "A Specification of JOVIAL." Commun. ACM, Vol. 6,No. 12, pp. 721 - 736.

Smith, J. B. (2006) Practical OCaml. Apress, Springer-Verlag, New York.

Sommerville, I. (2005) Software Engineering, 7e. Addison-Wesley, Reading, MA.

Steele, G. L., Jr. (1990) Common LISP The Language, 2nd ed. DigitalPress, Burlington, MA.

Stoy, J. E. (1977) Denotational Semantics: The Scott - Strachey Approach to Programming Language Semantics. MIT Press, Cambridge, MA.

Stroustrup, B. (1983) "Adding Classes to C: An Exercise in Language Evolution." Software—Practice and Experience, Vol. 13, pp. 139 - 161.

Stroustrup, B. (1984) "Data Abstraction in C." AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8.

Stroustrup, B. (1986) The C++ Programming Language. Addison-Wesley,Reading, MA.

Stroustrup, B. (1988) "What Is Object-Oriented Programming?" IEEE Software, May 1988, pp. 10 - 20.

Stroustrup, B. (1991) The C++ Programming Language, 2e. Addison-Wesley, Reading, MA.

Stroustrup, B. (1994) The Design and Evolution of C++. Addison-Wesley,Reading, MA.

Stroustrup, B. (1997) The C++ Programming Language, 3e. Addison-Wesley, Reading, MA.

Sussman, G. J., and G. L. Steele, Jr. (1975) "Scheme: An Interpreter for Extended Lambda Calculus." MIT AI Memo No. 349 (December 1975).

Suzuki, N. (1982) "Analysis of Pointer 'Rotation."' Commun. ACM, Vol. 25, No. 5, pp. 330 - 335.

Syme, D., A. Granicz, and A. Cisternino. (2010) Expert F# 2.0. Apress,Springer-Verlag, New York.

Tanenbaum, A. S. (2005) Structured Computer Organization, 5e. Prentice-Hall, Englewood Cliffs, NJ.

Tenenbaum, A. M., Y. Langsam, and M. J. Augenstein. (1990) DataStructures Using C. Prentice-Hall, Englewood Cliffs, NJ.

Teitelbaum, T., and T. Reps. (1981) "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment." Commun. ACM, Vol. 24, No. 9, pp. 563 - 573.

Teitelman, W. (1975) INTERLISP Reference Manual. Xerox Palo AltoResearch Center, Palo Alto, CA.

Thomas, D., C. Fowler, and A. Hunt. (2005) Ruby: The Pragmatic Programmers Guide, 2e, The Pragmatic Bookshelf, Raleigh, NC.

Thompson, S. (1999) Haskell: The Craft of Functional Programming, 2e. Addison-Wesley, Reading, MA.

Turner, D. (1986) "An Overview of Miranda." ACM SIGPLAN Notices,Vol. 21, No. 12, pp. 158 - 166.

Ullman, J. D. (1998) Elements of ML Programming. ML97 Edition. Prentice-Hall, Englewood Cliffs, NJ.

van Emden, M.H. (1980) "McDermott on Prolog: A Rejoinder." SIGART Newsletter, No. 72, August, pp. 19 - 20.

van Wijngaarden, A., B. J. Mailloux, J.E.L. Peck, and C.H.A. Koster. (1969) "Report on the Algorithmic Language ALGOL 68."Numerische Mathematik, Vol. 14, No. 2, pp. 79 - 218.

Wadler, P. (1998) "Why No One Uses Functional Languages." ACM SIGPLAN Notices, Vol. 33, No. 2, February 1998, pp. 25 - 30.

Wall, L., J. Christiansen, and J. Orwant. (2000) Programming Perl, 3e. O' Reilly & Associates, Sebastopol, CA.

Warren, D.H.D., L. M. Pereira, and F.C.N. Pereira. (1979) "User' s Guide to DEC System-10 Prolog."Occasional Paper 15. Department of Artificial Intelligence, Univ. of Edinburgh, Scotland.

Watt, D. A. (1979) "An Extended Attribute Grammar for Pascal." ACM SIGPLAN Notices, Vol. 14, No. 2, pp. 60 - 74.

Wegner, P. (1972) "The Vienna Definition Language." ACM Computing Surveys, Vol. 4, No. 1, pp. 5 - 63.

Weissman, C. (1967) LISP 1.5 Primer. Dickenson Press, Belmont, CA.

Wexelblat, R. L. (ed.) (1981) History of Programming Languages. Academic Press, New York.

Wheeler, D. J. (1950) "Programme Organization and Initial Orders for the EDSAC." Proc. R. Soc. London, Ser. A, Vol. 202, pp. 573 - 589.

Wilkes, M. V. (1952)"Pure and Applied Programming."In Proceedings of the ACM National Conference, Vol. 2. Toronto, pp. 121 - 124.

Wilkes, M. V., D. J. Wheeler, and S. Gill. (1951) The Preparation of Programs for an Electronic Digi tal Computer, with Special Reference tothe EDSAC and the Use of a Library of Subroutines. Addison-Wesley, Reading, MA.

Wilkes, M. V., D. J. Wheeler, and S. Gill. (1957) The Preparation of Programs for an Electronic Digital Computer, 2e. Addison-Wesley, Reading, MA.

Wilson, P. R. (2005) "Uniprocessor Garbage Collection Techniques." Available at http://www.cs.utexas.edu/users/oops/papers.htm#bigsurv.

Wirth, N. (1971) "The Programming Language Pascal." Acta Informatica,Vol. 1, No. 1, pp. 35 - 63.

Wirth, N. (1973) Systematic Programming: An Introduction. Prentice-Hall, Englewood Cliffs, NJ.

Wirth, N. (1975) "On the Design of Programming Languages." Information Processing 74 (Proceedings of IFIP Congress 74). North Holland, Amsterdam,pp. 386 - 393.

Wirth, N. (1977) "Modula: A Language for Modular Multi-Programming." Software—Practice and Experience, Vol. 7, pp. 3 - 35.

Wirth, N. (1985) Programming in Modula-2, 3e. Springer-Verlag, NewYork.

Wirth, N. (1988) "The Programming Language Oberon." Software—Practice and Experience, Vol. 18, No. 7, pp. 671 - 690.

Wirth, N., and C.A.R. Hoare. (1966) "A Contribution to the Development of ALGOL." Commun. ACM, Vol. 9, No. 6, pp. 413 - 431.

Wulf, W. A., D. B. Russell, and A. N. Habermann. (1971)"BLISS: A Language for Systems Programming." Commun. ACM, Vol. 14, No. 12, pp. 780 - 790.

Zuse, K. (1972) "Der Plankalkül." Manuscript prepared in 1945, published in Berichte der Gesellschaft für Mathematik und Datenverarbeitung,No. 63 (Bonn, 1972); Part 3, 285 pp. English translation of all but pp. 176 - 196 in No. 106 (Bonn, 1976), pp. 42 - 244.

编程语言原理(第10版)

Concepts of Programming Languages, Tenth Edition

本书从为什么学习程序设计语言入手，深入细致地讲解了命令式语言的主要结构及其设计与实现，内容涉及变量、数据类型、表达式和赋值语句、控制语句、子程序、数据抽象机制、对面向对象程序设计的支持(继承和动态方法绑定)、并发、异常处理和事件处理等方面。最后两章介绍了函数式程序设计语言和逻辑程序设计语言。

本书内容丰富，讲解透彻，既可用做高等院校计算机及相关专业本科生程序设计语言课程的教材和参考书，也可供程序设计人员参考。

本书特色

- 介绍了编程语言的相关主题
- 介绍了高级面向对象主题和语言
- 提供了与一些著名计算机科学家和语言之父的访谈
- 提供了大量重要的历史史料
- 深入讨论了几种常见语言的设计问题
- 提供了函数式和逻辑两种编程泛型

配套学习资源网站

www.pearsonhighered.com

提供了幻灯片和几种语言的简明手册

作者简介

Robert W. Sebesta是科罗拉多大学斯普林斯分校的计算机科学系的荣誉退休副教授。他获得了科罗拉多大学博尔德分校应用数学的学士学位，宾夕法尼亚州立大学计算机科学的硕士学位和博士学位。他从事计算机科学的教学已超过38年。他的专业兴趣是编程语言的设计与评价。

ISBN 978-7-302-31112-6

9 787302 311126 >

定价：98.00元

清华大学出版社数字出版网站

WQBook 书文局泉

www.wqbook.com

PEARSON

www.pearson.com

[General Information]
书名=编程语言原理（第10版）
页数=633
SS号=13226225