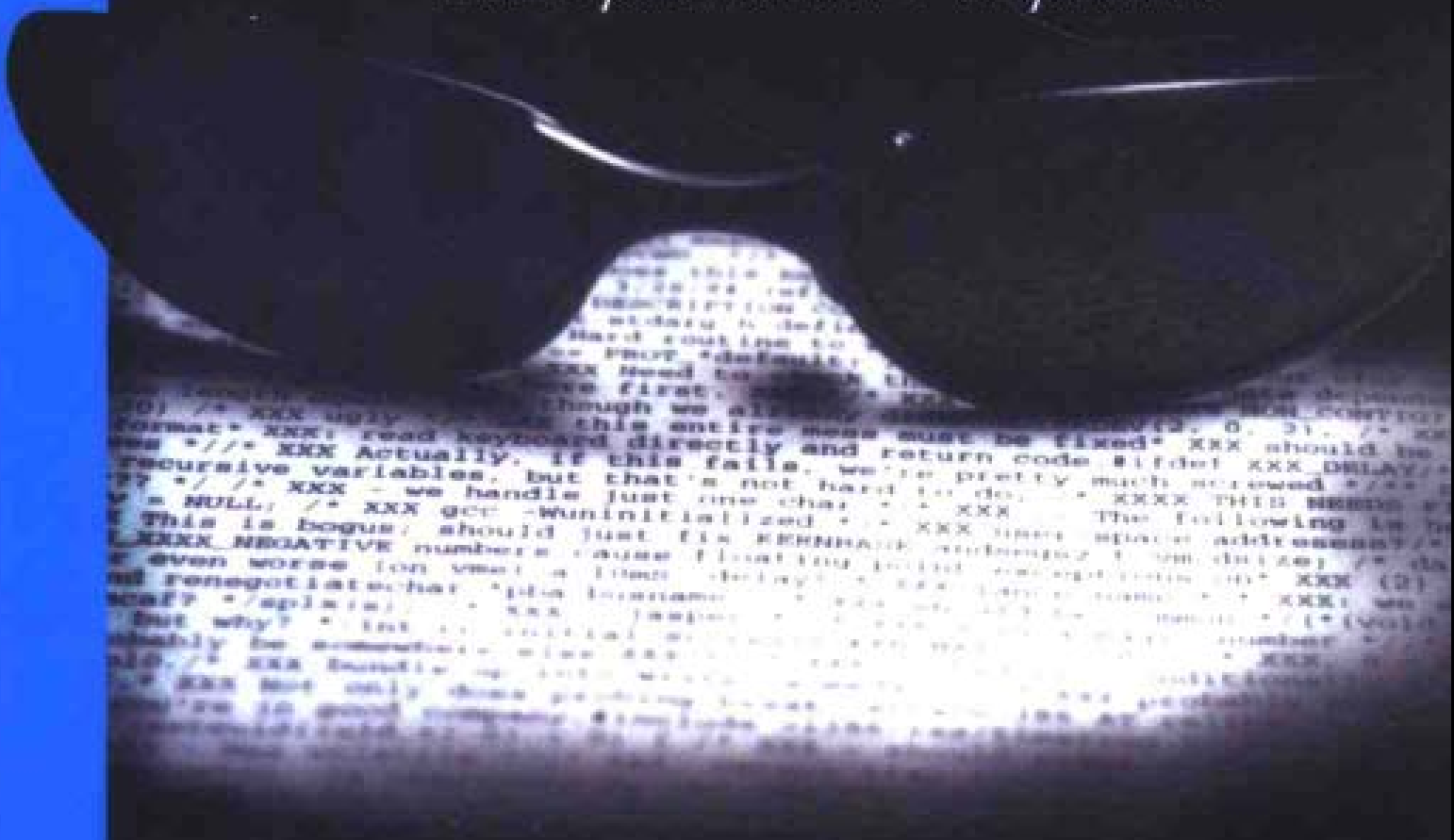
高级编程丛书

PEARSON
Addison Wesley

代码阅读方法与实践

CODE *Reading*

The Open Source Perspective



14th Annual
SOFTWARE
Development
PRODUCTIVITY
AWARD

(希腊)Diomidis Spinellis 著
赵学良 译

清华大学出版社

高级编程丛书

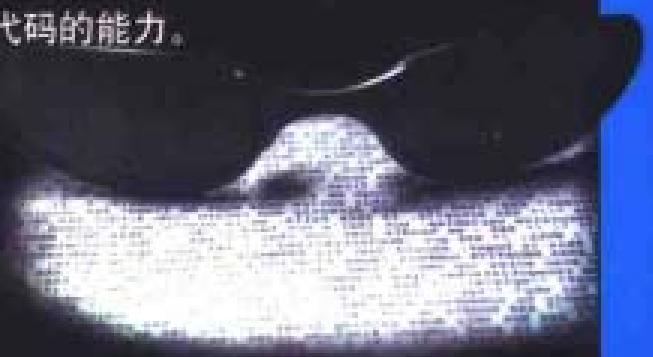
如果您是程序员，那么本书不容错过！

代码阅读方法与实践

阅读代码有许多原因：
或是为了修复、检查、改进代码；
或是为了发现它的运作机理；
或是提取可重用的材料加以利用。

代码阅读有自身的一套技能，重要的是能够确定什么时候使用哪项技术。本书中，作者使用600多个现实的例子，向读者展示如何区分好的（和坏的）代码；如何阅读，应该注意什么，以及如何使用这些知识改进自己的代码。

养成阅读高品质代码的习惯，可以提高编写代码的能力。



ISBN 7-302-08072-0
9 787302 080725 >
定价：45.00元（附光盘1张）

PEARSON
Addison Wesley

文稿编辑：汤浦涛
封面设计：陈刘源
读者邮箱：Book@21bj.com
信息网站：http://www.ePress.cn
http://www.34.cn
http://www.pearsoned.com

高级编程丛书

代码阅读方法与实践

（希腊） Diomidis Spinellis 著

赵学良 译

清华大学出版社

北京

内 容 简 介

阅读代码是程序员的基本技能，同时也是软件开发、维护、演进、审查和重用过程中不可或缺的组成部分。本书首次将阅读代码作为一项独立课题，系统性地加以论述。本书引用的代码均取材于开放源码项目——所有程序员都应该珍视的宝库。本书围绕代码阅读，详细论述了相关的知识与技能。“他山之石、可以攻玉”，通过仔细阅读并学习本书，可以快速地提高读者代码阅读的技能与技巧，进而从现有的优秀代码、算法、构架、设计中汲取营养，提高自身的开发与设计能力。

本书适用于对程序设计的基本知识有一定了解，并想进一步提高自身开发能力的读者。

图书在版编目（CIP）数据

代码阅读方法与实践/ （希腊）斯平内利斯（Spinellis,D.）著；赵学良译. —北京：清华大学出版社，2004
（高级编程丛书）
书名原文：Code Reading : The Open Source Perspective
ISBN 7-302-08072-0
Ⅰ.代… Ⅱ.①斯… ②赵… Ⅲ.程序语言—代码 Ⅳ.TP311.11
中国版本图书馆 CIP 数据核字（2004）第 008220 号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦
http://www.tup.com.cn　　**邮　　编**：100084
社 总 机：010-62770175　　**客户服务**：010-62776969
文稿编辑：汤涌涛
封面设计：陈刘源
印 刷 者：北京市季蜂印刷有限公司
装 订 者：三河市化甲屯小学装订二厂
发 行 者：新华书店总店北京发行所
开　　本：185×260　**印张**：21.5　**字数**：520 千字
版　　次：2004 年 3 月第 1 版　2004 年 3 月第 1 次印刷
书　　号：ISBN 7-302-08072-0/TP·5841
印　　数：1～4000
定　　价：45.00 元(附光盘 1 张)

读者评论

标题:对代码阅读这一极端困难课题的出色处理

日期:2003 年 11 月 4 日

评论人:美国德克萨斯州读者 Cltss

为什么不同的人会用不同的方式处理这一问题呢?

没有人能够回答。无论从哪方面讲,代码阅读都不是一个容易的课题。在此之前,我不知道是否其他书籍曾尝试过阐述这一主题。因此,当我看到这本书时,我毫不犹豫地选择了它。

阅读这本书时,我的第一反应是"这本书在介绍什么呢?"。我在阅读和理解他人的代码时,曾有过无数次这种感觉。

就如同,当您遇到一段新代码时,需要花一段时间来理解与消化一样,理解与消化这本书背后的思想也要花费一些时间。实际上,我多次打开这本书,重新阅读部分内容,并将它应用到实际的环境中,从而通过具体的形式不断加深对这个课题相关理论的理解。

如果您的情况和我类似,那么这本书可能很适合您。这就是我对本书内容的说明。

如果您喜欢编程,为了谋生必须阅读代码,希望了解一些技术,那么这本书绝对不容错过。

标题:睿智幽默地传授丰富的经验

日期:2003 年 8 月 29 日

评论人:美国北卡罗来纳州读者 Snoeyink

我曾在 UNC Chapel Hill 主持过生物信息研讨会,这个研讨会的目的是将生物-化学-物理系的学生和计算机科学系的学生召集到一起,力图提高前者的编程技巧,提高后者对生物化学/生物物理的理解。在这期间,我发现这本书对我主持该研讨会大有裨益。它阐述了为什么以及如何阅读代码的例子,指出了习惯用法和易犯的错误,能够帮助读者编写、维护或改进代码。它充满闪光的思想,道出了许多经验,表述也比较幽默。

惟一的问题是,可能从本书中受益最大的初学者,往往不会选择这样一本介绍如何阅读 C 程序的书籍,除非有人告诉他们这本书的确必不可少。专家级的技术人员可能已经从过去的经历中了解到了本书中讲述的大部分内容,尽管这样,他们依旧会十分喜爱这本书,因为它可以再次证实他们已经掌握的东西。但我认为专家们也会乐意将本书推荐给初级编程人员,传递提炼自经验的丰富知识。

标题:阅读他人的代码可以学到很多东西

日期:2003 年 7 月 21 日

评论人:美国加利福尼亚州读者

希望了解开放源码中存在的各种编程风络以及编程方法的人,以及寻求开拓(或加深)自己对软件工程理解的人,选择这本书将是明智之举。这不仅是因为本书能够帮助您了解软件的实现细节,还因为它提供了这些软件的创建者在开发过程中做出各项选择的初衷。这本书不同于其他优秀的教科书,它不讲授计算机科学的内容,但它会使您理解并意识到这些程序员在创建他们的复杂软件时头脑中进行权衡的内容。

标题:独特且创新性的读物

日期:2003 年 6 月 22 日

评论人:美国加利福尼亚州读者 Jack D Herrington

这本书是自 Dave Thomas 和 Andrew Hunt 的 *Pragmatic Programmer* 之后,最引人注目的一本读物。这并不奇怪,因为 Dave 好像也参与了本书的编撰。

本书传达的并非未经加工的信息,它给出了实用的建议,对于初级和高级工程师日复一日的编码与项目开发工作,都有莫大的帮助。

我开始厌烦那些"XX 傻瓜书"以及"三周内学会 XX"风格的图书。如果您也是这样,并对本书介绍的经过时间检验的技术和实用的方案感兴趣,那么这本书很适合您。

看过本书前面几章后,我开始质疑我购买这本书是否正确。但随后的章节迅速打消了任何疑问,我的购买完全正确。本书给出了许多有用的提示,在工作中可以节省我数分钟甚至数小时的时间,使得我的工作更轻松,也更敏捷。

我在 Palo Alto Borders 看到这本书纯属偶然。这些伟大的图书的确需要有效地营销。也就是说,我赞同 Addison-Wesley 对这类书籍的投资,虽然这本书的内容可能不太主流,但对于软件工程团体来说极具价值。

译 者 序

有一扇窗，从未打开，却要永远关闭；
有一些人，确实存在，我们却无缘相见；
有一种生活，还没有到来，我们却已永远离开。

开放源码——程序员的天堂

好的程序就如同好的音乐一样，它们完成得那么巧妙，那么完美，体现出完全没有词藻的美丽。就如同好的音乐能够改变你对人生的看法，让你重新审视你的生活，和多年前的人进行跨时空的交流一样，作为软件开发人员，好的程序所能够带来的感受丝毫不逊于音乐给你带来的冲击。更为难得的是，这些宝贵的程序往往并非由“权威人士”、“享誉海内外的专家”所编写，它们是由一个个普通的程序员写就。这些程序员和读者没有什么不同。虽然这些程序员并非叱咤风云的人物，但专业造就了专家，长时间集中在某个领域中就能够创建出所有程序员都应该珍视的财富。就如同古代的米开朗基罗，相比于当时那些声名显赫的主教、贵族来说，他是多么的微不足道，但时至今日，那些人和他们的名字都随风飘逝，昔日的荣光都随时间而暗淡，而米开朗基罗的名字却铭刻在大多数甚至是普通人的心中，许多人的脑海中都能浮现出那些完美的雕塑——“大卫”、“创世纪”。这份荣耀在文艺史上可能无人能及。

但一个不可忽视的现实是，对于大部分正在使用的软件，我们很难看到它们的源代码，从而也就失去了欣赏和从中学习的机会。和音乐不同的是，你不可能花钱买到一份拷贝回家欣赏。幸运的是，现在情况已经发生了很大的改变，我们拥有了一个宝贵的资源——开放源码项目。

开放源码和商业产品的源代码有着本质上的不同。这种不同体现在以下几个方面：

1. 开放源码项目中的大部分代码都经历过许多人的阅读和检查，在商业产品中由于人力和版权方面的限制，根本不可能做到这一点。同时，由于作者在开发时就意识到自己编写的代码要被别人阅读，故而在规范的遵守、算法的优化方面都格外用心。

2. 开放源码由于没有严格的时间限制，勿需时间与质量的权衡，从而产生的代码质量更高，对于读者也更有借鉴作用。而面向普通用户的软件，比如游戏、娱乐软件等，大都会选择在圣诞节和暑假之前发布，面为了满足市场销售的需要，软件新版本的开发工作也大都定在圣诞节和暑假之前两到三个月必须完成。这就要求软件的开发人员必须严格遵守时间上的要求。

3. 开放源码能够做到优胜劣汰，拥有更快的更新速度。开放源码项目一般接受各种形式的反馈。来自全世界的代码阅读者都可以及时地反馈自己发现的问题或好的看法。在这里，评判优劣的惟一标准是代码的质量。好的代码经受住时间的考验，留存下来。不好的代码和不太合格的代码随着时间的推移迅速地淘汰。而在商业产品的源代码中，一些遗留代

码中存在的问题可能永远得不到改正，因为开发它的人员已经离开公司或者从事其他工作，只要这段代码能够工作，没有人再去关注它的优劣。

4. 开放源码没有权威崇拜。开放源码的世界里没有阿谀奉承、阳奉阴违、沽名钓誉、挂着羊头卖狗肉，所有的程序都经过无数人的检验。正是由于开放源码具有这些特性，使得它成为程序员学习和提高的首选材料。

代码阅读——入门与提高的必由之路

信息时代，我们整个社会的知识更新速度越来越快，而位于信息时代风口浪尖上的计算机技术，更是日新月异。作为程序员，总是要不断地学习新的知识。但我们要保持清醒，如何体现出我们的价值呢？难道是我们掌握了多少知识吗？不对！我们的价值归根到底要体现在我们创造出的系统上。否则空有满腹经纶，到头来也不过是纸上谈兵，于国于人毫无益处。

计算机科学是一门实践性很强的科学，许多内容往往在书本上根本学不到。就拿项目的组织来说，没有什么书籍专门论述应该如何组织与管理项目的目录结构，因为这本身就是一种见仁见智的活动，要受到各种因素的影响。代码中往往凝聚着许多实践性的知识，通过阅读代码才能真正掌握软件开发的真谛。我们可以将“书籍是人类进步的阶梯”，扩展成“一切承载人类知识的载体，都是人类进步的阶梯”。

代码阅读是一件如此基本的事情，以至于人们根本没有意识到它的存在。这本书第一次将代码阅读这一主题单独列出来加以探讨，可以说极具开创性。

代码阅读不是一件容易的工作，但却是一件不得不做的工作，无论是工作的移交、新手的入门或是加入新的项目，都要阅读大量由他人编写的代码。我们可能不止一次地听到过他人抱怨：与其读他人的代码，还不如自己去写更为轻松。可见，代码阅读任务可能远比实现一个链表或树形数据结构要复杂困难。但代码阅读也并非完全无章可循。掌握了一些常见的技巧以及常用的工具之后，能够有效地降低代码阅读的难度，提高工作效率。

我曾向一位资深的开发人员询问过，应该如何提高自身的开发与设计能力。他的回答是，广泛地阅读一些现有的构架，比如 Apple 的开发框架、Borland 的 OWL 和 Delphi、Smalltalk、MFC 等，了解它们是如何组织的。计算机科学就是这样一门学科，入门十分容易，不需要深奥的数学知识，也没有复杂的物理模型；但要想提高却很难，真正能够叱咤风云、引领潮流的人却少之又少。这就有如人生，在蹒跚学步之时，主要的精力都要放在看脚下的路面上，注意路上的坎坷和荆棘；随着年龄的增长，当走路越来越稳健时，却迷失了方向，不知道应该走向何方。这个时候脚下的路已经不再重要，你需要从先哲们留下的只字片言，别人走过的“路”中汲取灵感，确定自己的方向。

由于 Microsoft Windows 的广泛采用，使得许多程序员只将注意力放在 Windows 平台上，这种做法失之偏颇。作为文秘人员，当然可以这样做，因为 Microsoft 在桌面系统上的确十分成功，基本覆盖了文秘人员对计算机的所有需求。但作为开发人员，必须跳出 Windows 这口“井”。总是局限在 Windows 这口“井”中，可能会限制您的思路与视野。当设计与实现复杂的系统时，需要借鉴许多前人的经验，面这时候，您就需要开阔的视野，才能从更为广泛的空间中吸收所需要的知识。跳出 Windows 之后，您能看到各种各样的构架，各种各样的

抽象方法，各种各样不同的系统设计，也许其中一种就很适合您使用，不再需要冥思苦想，绞尽脑汁，不需进行大量的尝试，不再受大量的挫折之苦。Microsoft 是一家出色的公司，为用户提供了很有用的产品，但 Microsoft 并非软件的全部。有许许多多出色的操作系统、编辑工具、各种软件需要我们去认识和借鉴，甚至包括那些已经逐渐淡出人们记忆的软件。因为我们不是软件的用户，我们是软件的开发者，我们需要借鉴前代的经验，才能更好地前进。那些消逝的软件并非由于它们不优秀，只不过它们不适合于生存下来而已。存活下来的软件也并不一定是优秀的软件，只不过它们适合于生存下来而已。就好像许多古代文明，我们现在只能通过一些石块来追溯它们昔日的辉煌，有记录的内容都已经随着时间的流逝而消失，尽管它们比石块更能清楚地告诉我们想要知道的内容，尽管石块根本就是这些文明中最无关紧要的部分。

在夕阳西下的时间，泡上一杯茶，合上笔记本，揉一下看了一会源代码有些累的眼睛，伸个懒腰。这是不是有一些“采菊东篱下，悠然见南山”的意境呢？只不过陶渊明种的是菊花，我们播种与收获的是代码而已。

阅读建议——以本书为主线

这是一本外延极广的书籍，只需要看看本书每章列出的进阶读物，以及附在书后的 200 多项参考书目就能够认识到这一点。在阅读时，要将本书作为掌握代码阅读的相关知识，提高自身设计与开发能力的大纲。在一本书中详尽地介绍代码阅读过程中涉及的所有知识是不可能的，这就决定本书必然会对相关内容做出取舍，有些部分详加叙述（和一些书籍相比，即使是详加叙述的内容也要简短得多），有些部分只能粗略带过。如果在阅读过程中对本书提及的内容不甚了了，则要停下来，查找相应的参考书，掌握相关的内容。

所以，在阅读本书的过程中，最好以本书为主线，不时地查阅相关的参考资料，分析研读本书引用的源代码，然后回到本书继续后面的内容。这样，才能在掌握本书精髓的基础上真正有所提高。

本书的另一大特色是所引用的每段代码，哪怕只有一行，也都摘自真正的开放源码项目。首先，这使得本书所讲述的内容都言之确凿；其次，也使得读者在阅读本书的过程中逐渐熟悉了许多开放源码项目，对它们有了大致的了解。由于所有的代码都有真实的上下文，读者可以切实感受到本书讲授的内容，同时，还可以适度扩大阅读量。

在阅读本书的过程中，不要急于读完，而是要不断停下来，将其中的每部分内容都加以参详，才不至于“入宝山而空回”。

致　　谢

作为一名程序员，我首先要感谢清华大学出版社能够及时引进这部图书，使我们能以更便宜的价格（相对于英文原版）、更通畅的渠道和更高的质量得到这样一本影响深远的优秀书籍。

感谢在我成长与工作过程中一直帮助我的家人、同学和朋友。

序

我们是程序员。我们的工作(许多情况下,还有我们的激情)是通过编写代码来创造新的事物。大量的图示、详尽的项目进度表、一堆堆厚厚的设计文档并不能满足用户的需求。这些都是愿望,表达出我们希望实现的东西。我们只有通过编写代码才能交付真正满足用户需求的东西:代码才是现实的。

这也正是我们被灌输的思想,好像很合理。我们的工作是编写代码,所以我们需要学习如何编写代码。大学课程教我们编写程序。培训课程告诉我们如何调用新的库和 API。这是整个行业中最大的悲剧之一。

因为,学习编写伟大代码的方式是阅读代码,阅读大量的代码:高品质的代码、低品质的代码;汇编语言代码、Haskell 代码;千里之外的陌生人所写的代码;以及我们自己上周刚刚编写的代码。因为,如果不这样,我们就会不断地重做别人已经完成的工作,重复过去已经发生过的成功和错误。

恐怕没有哪个伟大的小说家从未读过其他人的著作,没有哪个伟大的画家从未研究过他人的绘画作品,没有哪个技术熟练的外科医生从未观摩过同事如何动手术,没有哪个波音 767 的机长不是首先在副驾驶员的位置上观看如何实际操作的。

可是,我们却期望程序员能够做到这些(即,不用读他人的代码就能够编写出优秀的代码。——译者注)。"本周的任务是编写……"。我们告诉开发人员语法和构造规则,之后,我们希望他们能够编写出相当于伟大小说的软件。

具有讽刺意味的是,对阅读代码来说,从来没有过比现在更好的时光。感谢开放源码社团所做出的巨大贡献,现在网络上有数以千兆的源代码,等待我们去阅读。选定任何一种语言,您都能找到源代码。选择一个问题域,也存在大量的源代码。选择一个级别,从微码(microcode)到高级的商业功能,您都能够找到大量的源代码。

代码阅读充满了乐趣。我喜欢阅读他人的代码。我阅读代码是为了学习技巧,并分析陷阱。有时,我会碰到很小,但很宝贵的精华。我依旧记得,当我在 PDP-11 汇编程序中无意中找到一个二进制到八进制转换例程时所获得的惊喜,这段例程设法不用循环计数,在一个紧凑的循环中输出一个 6 位八进制数。

有时,我阅读代码是为了寻求一些故事,就如同在长途飞行之前,您在机场会选择一本书。我期望,从灵巧的结构和意外的对称美中能够获得愉悦。Jame Clark 的 gpic 程序(属于他的 GNU groff 包)就是此类代码的一个绝佳例子。他用简明且优雅的结构实现了极为复杂的功能(一种说明性的、设备无关的绘图语言)。看完代码后,我受到了激励,想着如何同样整洁地构造自己的代码。

有时,我还批判地阅读代码。在这种情况下,阅读速度会放慢。当我阅读时,我会不断地向自己提问,比如:"为什么要这样写?"或"作者背景中的什么东西使他/她做出了这种选择?"我这样做经常是因为,我要复查代码来解决存在的问题。我寻找能够给予我指示的模式与线索。如果我看到,作者在代码中某处没有对共享的数据结构进行锁定,我就会怀疑同

样的问题是否在别处同样存在，接下来，就会想是否就是这个错误引起了我正在处理的问题。我还使用我发现的不一致性，进一步验证我的理解；我经常发现，那些本来认为可能存在问题的地方，经过进一步仔细分析后，常常是完美的好代码。从中，我也学到了一些东西。

实际上，代码阅读是消除程序中存在问题的最有效方式之一。Robert Glass，本书的审稿人之一，说过："通过正确地使用（代码）审查，软件产品中90%以上的错误能够在测试之前消除。"同一篇文章中，他引用的调查研究表明，"将注意力放到代码上的审查员比将注意力放到过程上的审查员要多发现90%的错误。"有趣的是，当阅读本书引用的代码片段时，我就碰到几个bug和几个可疑之处。存在这些问题的代码正在全球数千个站点上运行。虽然实际上没有严重的问题，这个练习表明我们编写的代码总是存在改进的空间。代码阅读能力显然拥有巨大的现实好处，如果您曾和明显不知道如何阅读代码的人共同做过代码复查，肯定已经体会到这些好处。

接下来就是维护——软件开发令人讨厌的远亲。虽然没有精确的统计数字，但大多数研究员都同意，我们花费在软件上的时间当中，超过一半是用在检查现有代码上：增加新的功能、修复bug、将其集成到新环境等。代码阅读技能极其重要。如果在一个10万行规模的程序中存在一个bug，给您一个小时的时间来找到它。您如何开始工作呢？您怎么知道您在找什么？并且，您如何评估做出的更改所造成的影响呢？

因为所有这些原因，以及更多的其他原因，我喜欢这本书。从本质上讲，这是一本实用主义的书。它没有采取抽象的、学院式的方式，而是将注意力放在代码本身上。本书分析了几百个代码片段，指出了其中的技巧、陷阱和（同等重要的）习惯用法。本书在代码原来的环境中谈论代码，同时论述了环境如何影响代码。本书还突出了代码阅读者的一些重要工具，从常见的工具如grep和find，到更为特殊的工具。同时，本书强调了构建工具的重要性：编写代码帮助阅读代码。此外，本书附带了书中论及的所有代码，并在配套盘上做了交叉引用，方便使用。

本书应该包括在每个编程课程中，每个开发人员都应该有一本。作为一个社团，如果我们在代码阅读投入更多的精力，既可以节省我们的时间，又能够减少我们的痛苦。我们将会节省本行业的资金。同时，我们也会在工作中体会到更多的乐趣。

Dave Thomas
The Pragmatic Programmers, LLC
http://www.pragmaticprogrammer.com

前　言

今天，我们从阅读中得到的东西，甚至相当于我们在过去 14 年中的亲身体会。

——Graham Greene

阅读代码可能是计算领域的专业人士最常见的活动之一，但人们很少将它作为一门课程来讲授，或正式地将它作为学习程序设计的方法。

造成这种现状的原因，最初可能是由于得不到源代码，或缺乏高品质的代码可供阅读。公司常常将源代码作为商业机密加以保护，很少允许其他人阅读、添加注释、试验并从中学习。仅在少数情况下，重要的专利代码允许扩散到公司之外，它们引起了公众极大的兴趣，并得到了创造性的发展。例如，整整一代程序员都受益于 John Lions 的 *Commentary on the Unix Operating System*（《Unix 操作系统注释》），该书列出了 Unix 核心第 6 版的完整源代码，并做了注释。虽然 Lions 的书在最初编写时受到 AT&T 相关授权的约束，只能用于操作系统课程，不对大众开放，但是，该书的副本已经通过私下影印流通了多年。

但是，在过去的几年里，开放源码软件的普及为我们提供了大量的代码，我们可以自由地阅读所有这些代码。今天最广泛使用的一些软件系统，比如 Apache Web 服务器、Perl 语言、GNU/Linux 操作系统、BIND 域名服务器和 sendmail 邮件传输代理程序，实际上都有开放源码的形式。因此，我很幸运，能够使用开放源码软件（比如上面列出的这些软件系统）来编写本书，作为软件代码的初级读本和教科书。我的目标是为阅读他人的代码提供背景知识和技术。通过使用实际的例子（这些例子都取自正在使用的开放源码项目），我想介绍有可能出现在软件开发人员面前，与代码相关的大部分概念，包括编程构造、数据类型、数据结构、控制流程、项目组织、代码规范、文档和构架。另有一本书介绍接口和面向应用的代码，包括国际化和可移植问题、常用库和操作系统中的元件、低级代码、领域专有的声明性语言、脚本语言和混合语言系统。

本书是（就我所知）第一本专门将代码阅读作为一项独立活动加以讲述的书籍，是一本物有所值的书。将代码阅读作为一种独立的活动来介绍，肯定会存在不可避免的缺点：有可能其他的方式对内容的处理会更好，也有可能在讲述过程中缺失重要的材料。但我坚定地认为，代码阅读应该得到正确地训练，并用作提高编程能力的一种方法。因此，我希望本书将激起人们的兴趣，将代码阅读课程、活动和实践包括到计算教育的课程中，以便几年后我们的学生能从现有的开放源码系统中学习有用的知识，就如同他们的同辈们从伟大的文学作品中学习语言一样。

辅助材料

本书中提供的许多源代码示例都来自于 NetBSD 的源代码。NetBSD 是一个免费的、高度可移植的类 Unix 操作系统，适用于多种平台，从 64 位 AlphaServer 到手持式设备。它清

晰的设计和先进的特性，使得它无论对于生产环境，还是对于研究环境，都是一个极好的选择。之所以选择 NetBSD，而不选择其他同样优秀，且广泛应用的免费类 Unix 系统（比如 GNU/Linux、FreeBSD 和 OpenBSD），是因为 NetBSD 主要的目标就是强调正确的设计和编写良好的代码，这使得它成为提供源代码示例的最佳选择。根据其开发者的思想，有些系统好像拥有这样的哲学："如果它能够工作，那么它就是正确的，"而 NetBSD 可以描述为"除非它正确，否则它就不能工作。"另外，NetBSD 系统的一些其他目标与本书的目标相当符合。尤其是，NetBSD 避开了许可的拖累，提供一个可以在多种硬件平台上运行的可移植系统，与其他系统的互操作性良好，并且符合开放系统的实用规范。本书中使用的代码是一个（现已成为历史）export-19980407 快照。有几个例子用到了我在代码中找出的一些错误；由于 NetBSD 的代码不断发展，如果使用来自更近版本的例子，这些珍贵的代码极有可能已经得到更正。

本书例子中，对系统其余部分的选用也都基于类似的前提：代码品质、结构、设计、效用、普及和不会使我的出版商不安的许可。我努力平衡语言方面的选择，积极地寻找合适的 Java 和 C++代码。但是，当相似的思想能够使用不同的语言来演示时，我选择使用 C 语言，因为它拥有最基本的共同特性。

有时，我会使用真实的代码来说明不安全、不可移植、难以阅读或其他不好的编码实践。我认识到，我可能会由于贬低代码而受到指责，这些代码可能是其作者出于推进开放源码运动的良好信仰而贡献出来的，它们需要不断地改进而非仅仅是批评。如果我的注释冒犯了源代码的作者，我真诚地预先道歉。作为辩护，我得说，大部分情况下，注释并非针对特定的代码，而是用它来说明一种应该避免的情况。我用作反面教材的代码经常是"跛足的鸭子"，在编写它们的时候，技术和其他的限制证实了这种做法，或者对它们的批评脱离了特定的上下文。任何情况下，我都希望大家能够愉快地接受这些注释，我公开地承认，我自己的代码中包含相似的，甚至可能更坏的错误。

目　录

第1章 导 论

我很遗憾地告诉大家，就在最近，我再次查看了我的程序（质因子和井字游戏），它们没有任何形式的注释或文档。

——Donald E. Knuth

无论是沟通程序的操作，还是将知识存储为可执行的形式，软件的源代码都是最终的介质。我们可以将源代码编译成可执行程序，也可以阅读代码来了解程序的功能及其工作方式，还可以修改源代码来改变程序的功能。大多数编程课程和书籍都将重点放到如何从零开始编写程序上。然而，在软件系统的工作投入中，40%～70%是用在系统首次编写完成之后。这些工作一定涉及到阅读、理解、以及修改最初的代码。另外，遗留代码持续不断、不可避免的累积；对软件重用的强调；软件行业中人员的高流动性；同时，随着开放源码开发工作和协同开发过程（包括外包、代码走查和极限编程）日益重要，使得代码阅读成为当今软件工程师的一项基本技能。此外，阅读实际的、编写良好的代码，可以更加深入地了解如何构造与编写重要的系统，仅仅编写小型的程序学不到这种能力。在编写程序时就应该考虑到使之易于阅读，并且，不管程序是否容易阅读，人们都需要去阅读它们。虽然代码阅读是“一项少有人称颂、缺乏训练的活动”（Robert Glass，[Gla00]），但它不应该这样（即应该给予足够的重视，并提供相应的培训。——译者注）。

本书中，您将会通过一些具体的例子，学习如何阅读别人编写的代码，这些例子都取自重要的、现实的、开放源码系统的代码。我们对代码进行了大量的释义，并应用到项目中所有机器可读的部分：源代码（及其注释）、文档、可执行程序、源代码储存库、设计图和配置脚本。掌握了本书的内容之后，您将：

- 能够阅读和理解重要软件的代码；
- 领会许多重要的软件开发概念；
- 知道如何探索大型的代码体；
- 拥有大多数重要的高级、低级编程语言的阅读能力；
- 认识到现实软件项目的错综复杂性。

虽然本书在开始部分对基本的编程结构做了简短的介绍，但是，我们假定读者熟悉 C，C++或 Java，并能使用简单的工具在计算机上分析我们提供的代码。另外，如果曾经接触过我们论及的系统和应用程序——虽然对阅读本书来说这种经历并非必需，对本书介绍的内容会有更为深刻的理解。

本章余下的部分将讨论促使读者阅读代码的各种原因，以及适当的阅读策略；还提供一份简短的“指令手册”，介绍如何跟从我们提供的材料。祝您阅读愉快！

1.1 为什么以及如何阅读代码

有时，阅读代码是一件不得不去做的事，比如：为了修复、检查或改进现存的代码，都必须去阅读相关的代码。有些时候，阅读代码也许是为了了解程序是如何工作的，对于任何能够“打开盖子”的事物，作为工程技术人员，我们总是倾向于分析一下它的内部结构。您阅读代码可能是想提取可供重用的材料，或者仅仅出于个人兴趣（很少，但我们希望在阅读了本书后能够多起来），将代码作为一种文献。每种原因的代码阅读都有自己的一套技术，强调不同方面的技能①。

1.1.1 将代码作为文献

Dick Gabriel 指出，有几个创造性行业不允许作者相互阅读各自的著作，我们从事的行业即为其中之一[GG00]。

所有权规则带来的影响已经使得没有软件能够作为文献。就如同所有的作家都拥有自己的公司，只有属于 Melville 公司的人才能够阅读 Moby-Dick，只有 Hemingway 公司的雇员才能够阅读 The Sun Also Rises。您能够想像，在这样的环境下能够产生伟大的文学作品吗？在这样的条件下，既不会有文学课程，也没有办法讲授写作。我们希望人们在这种情况下学习编程吗？

（Melville，梅尔维尔，他根据自己的航海经历创作小说，*Moby-Dick*，中文译名《白鲸》，是他 1851 年的作品。Hemingway，海明威，美国作家，1954 年获诺贝尔文学奖，*The Sun Also Rises*，中文译名《太阳照样升起》，1926 年。——译者注）

开放源码的软件（Open-Source Software，OSS）改变了这种情况：我们现在能够访问数百万行的代码（代码的质量也有好有坏），我们可以阅读、评论和改进这些代码，从中学到许多东西。实际上，作为科学交流的媒介，社会活动对数学定理的成功做出了很大的贡献，它们也同样适用于开放源码软件。许多开放源码的软件程序已经被：

- 以源代码的形式归档、发表和检查；
- 论证、消化吸收、通用化和释义；
- 用来解决实际的问题——经常与其他程序协同工作。

要养成一个习惯，经常花时间阅读别人编写的高品质代码。就像阅读高品质的散文能够丰富词汇、激发想像力、扩展思维一样，分析设计良好的软件系统的内部结构可以学到新的构架模式、数据结构、编码方法、算法、风格和文档规范、应用程序编程接口（API），甚至新的计算机语言。阅读高品质的代码还可以提高您编写代码的水准。

阅读代码的过程中，不可避免地会遇到一些在实践中应该尽量避免的代码。能够快速地从坏代码中区分出好代码是一项有价值的技能；接触一些编码的反面例子可能有助于您提高这种能力。通过下面这些征兆，可以容易地识别出低品质的代码：

---

① 我很感激 Dave Thomas 对本节的建议。

- 编码风格不一致；
- 结构不必要地复杂或难以理解；
- 明显的逻辑错误或疏忽；
- 过度使用不可移植的构造；
- 缺乏维护。

然而，不应该期望能够从编写品质低下的代码中学习到如何正确地编程（看来爱迪生试验灯丝失败时采取的态度在阅读代码中是不可取的。——译者注）；如果将这样的代码作为文献来读，则是在浪费时间，尤其是考虑到现在能够访问到的高品质代码的数量。

请回答这个问题："我正在阅读的代码真的是最好的吗？"开放源码运动的优点之一就是，成功的软件项目和想法激励竞争者不断地改进它们的结构和功能。我们常常有幸看到对软件设计的第二次或第三次重复；大多数情况下（但不总是如此）后面的设计要比先前的版本有显著的提高。根据您正在寻找的功能，使用相关的关键词，在 Web 上搜索，可以容易地找到各种互相竞争的实现。

要有选择地阅读代码，同时，还要有自己的目标。您是想学习新的模式（pattern）、编码风格（coding style）、还是满足某些需求的方法？或者，您也许只是在浏览代码，获得其中的某些亮点。在这种情况下，要随时准备仔细地研究那些有趣但尚不了解的部分：语言特性（即使您对一种语言了解得很透彻，也不可能尽知它的所有特性，因为现代语言的发展离不开新特性，新的特性总是不断涌现）、API、算法、数据结构、构架和设计模式（design pattern）。

要注意并重视代码中特殊的非功能性需求，这些需求也许会导致特定的实现风格。对可移植性、时间或空间效率、易读性、甚至迷惑性的需求都可能导致代码具有非常特殊的特征。

- 我们曾看到过，在有些代码中，为了与老一代的链接程序保持可移植性，使用由 6 个字母组成的外部标识符。
- 有些高效算法的实现（就源代码的行数来说）比它们对应的自然实现要复杂两个数量级。
- 嵌入式或有限空间的应用程序（如 GNU/Linux 或 FreeBSD 在软盘上发布的各种分发版），它们的代码可能会为了节省几个字节的空间而增长很多。
- 专为演示算法如何运行的代码，可能会使用相对来说很长的标识符。
- 某些应用程序领域，如复制保护（copy-protection）方案，可能要求代码难以阅读，从而妨碍逆向工程（reverse engineering）的进行（经常是徒劳的）。

当您阅读属于上述类型的代码时，要注意这些特定的非功能性需求，并注意您的同行如何满足这些需求。

有时，您阅读的代码可能来自于对您来说完全陌生的环境（计算机语言、操作系统或 API）。熟悉了编程和基本的计算机科学概念后，大多数情况下，您能够通过源代码来了解新环境的基本情况。但要注意从小型的程序开始阅读；不要立即陷入对大型系统的研究中。编译研究的程序并运行它们。这样您可以得到即时的回馈，了解代码预想的工作方式，同时还可以获得成就感。下一步就是主动地修改代码来检验您对代码的理解是否正确。再次强

调，要从小的改动做起，逐渐地增大它们的范围。通过积极地介入现实的代码，您能够快速地从中了解到新环境的一些基本情况。当您认为已经掌握了它们之后，要考虑投入一些努力（可能还需要投入一些资金），采取更有组织的方式来学习该环境。阅读相关的书籍、文档或手册，或者参加培训课程；这两种学习方式互相补充。

另一种积极地阅读现有代码（作为文献）的方式是改进它。与其他文字作品相比，软件代码是活的人工制品，它们总是被不断地改进。如果代码对您或您的团体有价值，请考虑如何来改进它。这可能涉及到使用更好的设计或算法、为某些代码编制文档，或增加功能。开放源码项目中的代码常常没有很好地编制文档；请将您对代码的理解应用到改进文档上。在现有的代码上工作时，请与作者和维护人员进行必要的协调，以避免重复劳动或因此产生厌恶情绪。如果您的更改更为健壮，则考虑申请成为一个并发式版本控制系统（Concurrent Versions System，CVS）的提交者——拥有直接向项目的源代码库中提交代码的授权。请将从开放源码软件中得到的益处看作是一项贷款；尽可能地寻找各种方式来回报开放源码社团。

1.1.2 以代码为范例

有时，您也许想要了解一个特定的功能是如何实现的。对于某些类型的应用，您也许能够在普通的书籍或专业的出版物和研究论文中得出问题的答案。然而，多数情况下，如果您想要了解“别人会如何完成这个功能呢？”，除了阅读代码以外，没有更好的方法。如果想创建与给定实现相兼容的软件，代码阅读也很可能是最可靠的方式。

当将代码用作范例时，关键的思想就是要灵活。要准备使用大量不同的策略与方法来了解代码的工作方式。开始时，要阅读能够找到的尽可能多的文档（参见第 8 章）。最理想的情况是能够找到正式的软件设计文档，但即使是用户文档也很有帮助。实际使用该系统，了解它的外部接口。要清楚您正在寻找的是什么：是一个系统调用、一种算法、一段代码序列或是一种构架？设计一种发现目标代码的策略。不同的搜索策略适用于不同的目的。您可能需要跟踪指令执行序列，运行程序并在关键位置设置断点，或者在代码中进行文字搜索，找出特定的代码或数据元素。我们可以使用工具（参见第 10 章）来完成这些工作，但不要局限于任何一种工具。如果一种策略不能快速地产生期望的结果，停止使用它并选用其他不同的策略。切记，代码就在那里，我们需要做的只是找到它。

一旦定位到目标代码，就针对它进行研究分析，忽略不相关的其他部分。这是一种必须掌握的技能。本书中的许多练习都会要求读者完成这项任务（在众多的源代码中快速地找到目标代码，书中的许多练习都要求读者在本书的配套光盘中查找特定的代码序列，比如使用某种数据结构的代码、使用某种风格的代码等。——译者注）。如果您觉得在原来的上下文中，理解代码有困难，就将它复制到一个临时文件中，删除所有不相关的部分。这个过程的正式名称是切片（slicing）（参见 9.1.6 节），本书中，我们已经非正式地将这种方法应用到添加了注释的代码示例中，分析我们的用法，同样可以了解这个思想。

1.1.3 维护

其他情况下，代码可能不是我们学习的范例，而是需要得到修复。如果您认为自己发现了大型系统中的一个 bug，您需要采取一些战略战术，不断细化对代码的阅读，直到发现问

题为止。这种情况下,关键的思想是使用工具。我们要充分利用调试器、编译器给出的警告或输出的符号代码、系统调用跟踪器、数据库的结构化查询语言(Structured Query Language,SQL)的日志机制、包转储工具和 Windows 的消息侦查程序,定位 bug。(第 10 章有更多关于工具如何帮助代码阅读的内容。)请从问题的表现形式到问题的根源来分析代码。不要沿着不相关的路径(误入岐途)。编译程序时请加入调试支持,并使用调试器的栈跟踪机制、单步执行、以及数据和代码断点来缩小搜索的范围。

如果调试器不能共同运转(有时调试某些程序极端困难,诸如:在后台运行的程序——如守护程序(daemon)和 Windows 的服务、C++中基于模板的代码、小服务程序(servlet)和多线程代码),在这种情况下,可以考虑在程序执行路径中的关键位置加入输出语句。在分析 Java 代码时,可以使用 AspectJ 向程序中插入只在特定情况下才运行的代码元。如果问题与操作系统的接口有关,系统调用跟踪机制常常能够定位出问题之所在。

1.1.4 演进

大多数情况下(某些测量显示,超过 80%的时间),您阅读代码不是为了修复一个缺陷,而是为了增加新的功能、修改现存的特性、改编代码使之适应新的环境和需求,或者重构它们来增强代码的非功能性品质。在这些情况下,关键的思想是对所分析代码的范围有所选择;大多数情况下,实际需要理解的代码只是系统全部实现的很小一部分。在实际工作中,通过选择性地了解与更改一、二个文件,就能够修改一个具有上百万行代码的系统(如典型内核或窗口系统);我强烈建议,读者应尽可能地亲身体会一下这种操作的成功所带来的令人愉悦的感受。在有选择地处理大型系统的各个组成部分时,应该采用的策略概括如下:

- 定位到感兴趣的代码;
- 单独了解各个特定的部分;
- 推断节选出的代码与其余代码的关系。

当向系统中增加新功能时,首先的任务就是找到实现类似特性的代码,将它作为待实现功能的模板。相似地,当修改一个现存的特性时,首先需要定位底层的代码。从特性的功能描述定位到代码的实现,可以按照字符串消息,或使用关键词来搜索代码。例如,要找到 ftp 命令中进行用户身份验证的代码,可以在代码中搜索"Password"字符串①:

```
if (pass == NULL)
    pass = getpass("Password:");
n = command("PASS %s", pass);
```

定位到该特性后,就可以开始研究它的实现(跟踪所有相关的代码)、设计新的特性或增加的功能,以及定位可能影响到的区域——代码中与新代码发生交互的其他部分。在大多数情况下,您只需要彻底了解这些代码就足够了。

改写代码使之适应新的环境和上面的任务有所不同,需要采取另一套策略。有时,两个环境提供相似的功能:或许是将代码从 Sun Solaris 移植到 GNU/Linux,或从 Unix 系统移植到 Microsoft Winddws。在这些情形中,编译器可以成为您最有价值的朋友。在刚刚开始

① netbsdsrc/usr.bin/ftp/util.c:265-267

时，假定您已经完成任务，正尝试编译整个系统。依照编译与链接错误的指示系统性地修改代码，直到最终能够完全编译、链接通过为止，然后验证系统的各项功能。您会发现，采用这种方式可以极大地减少需要阅读的代码量。在修改了函数、类、模板或数据结构的接口后，也可以采用相似的策略。很多情况下，不用手动地定位更改所造成的影响，只需要依照编译器的错误与警告消息来定位有问题的点。对这些地方的修复经常会产生新的错误；通过这个过程，编译器会为您找出受您的代码所影响的代码。

当代码的新环境和原来的环境完全不同时（例如，将一个命令行工具移植到图形窗口环境中），则要采用不同的方法。在此，最小化代码阅读工作的惟一希望就是，集中注意老代码和新环境之间接口可能不同的地方。在我们概括的例子中，这意味着要专注于和用户进行交互的代码，完全忽略系统中所有与算法有关的方面。

代码演进变更中，一种完全不同的类型就是重构（refactoring）。由于某些开发工作采用极限编程（eXtreme Programming，XP）和敏捷编程方法（agile programming methodologies），这些更改也变得日益重要。重构中对系统的更改，保持系统的静态外部行为不变，而增强其非功能性品质，比如：简单性、灵活性、易理解性或性能。重构和整容手术有一个共同的属性。进行重构时，您从一个能够正常工作的系统开始做起，希望确保结束时系统能够正常工作。一套恰当的测试用例（test case）可以帮助您满足此项约束，所以重构应该从编写测试用例着手。一种类型的重构专注于修复一种已知的问题点。在此，您必须理解老的代码（这也是本书关注的问题）、设计新的实现、研究新的实现对相关的其他代码造成的影响（多数情况，新代码能够"无声无息"地完成替换）并实现更改。

另一种不同类型的重构是在软件系统上花费一些"品质时间"，主动寻找可以改进的代码。有几种情况您需要了解系统整体的设计与构架，此处即为这种情况。大规模的重构好像要比小规模的重构带来更多的好处。第 6 章讨论理清大型系统的各种方式，第 9 章概括如何从代码转换到系统构架。阅读代码寻找重构机会时，先从系统的构架开始，然后逐步细化，能够获得最大的效益。

1.1.5 重用

阅读代码也可能是为了寻找可供重用的元件。在此，关键是不要期望太高。代码的可重用性是一个很诱人，但难以掌握的思想；降低期望就不会感到失望。编写可重用的代码很困难。多年来，只有比较少的软件经受住时间的考验，在多种不同的解决方案中被重复使用。软件部件一般要经过逐渐地扩展，并重复改写以适用于两个或三个不同系统之后，才会成为可重用的部件；专为特别的目的而开发的软件很少满足这些条件。实际上，根据 COCOMO II 软件成本模型（software cost model）[BCH+95]，编写可重用的软件可能会增加 50%的开发工作量。

在寻找代码，重用到您正在处理的具体问题时，首先要将可以解决问题的代码分离出来。多数情况下，基于关键字搜索系统的代码就能够找到相关的实现。如果您希望重用的代码十分棘手、难以理解与分离，可以试着寻找粒度更大一些的包，甚至其他代码。例如，如果一个代码段与其相关部分之间的关系十分复杂，难以处理，则不必费力去理解它们，而是使用整个库、组件、进程，甚至代码所在的系统。

另一种重用活动是先行检查代码，收集有价值的可重用代码。在此，最好的办法是寻找

已经被重用的代码，也许就在您所分析的系统之中被重用。显示代码可以重用的积极信号包括使用合适的包装方法（参见 9.3 节）或配置机制。

1.1.6 审查

最后，在一些工作安排中，代码阅读工作可能是您工作的一部分。大量的软件开发方法学将技术检查作为开发过程中不可缺少的一部分，如：走查（walkthrough，即粗略地进行检查。——译者注）、审查（inspection）、循环复查（round-robin review）和其他类型的技术评估。在应用极限编程方法学过程中，进行组对编程（pair programming）时，您经常需要在您的伙伴编写代码的同时阅读它们。在这些情形中，代码阅读要求不同级别的理解、领会和警惕性。您需要一丝不苟，分析代码，发现功能和逻辑上的错误。另外，您应该随时准备讨论没弄明白的部分；验证代码是否满足所有的需求。

对代码的非功能性问题也应该给予同样的重视。代码是否符合组织的开发规范和风格呢？是否存在重构的可能呢？部分代码是否可以编写得更易读、更高效？某些部分是否可以重用现存的库或组件？在复查软件系统时，要注意，系统是由很多部分组成的，不仅仅只是执行语句。还要注意分析以下内容：文件和目录结构、生成和配置过程、用户界面和系统的文档。

软件审查和相关的活动涉及到许多工作人员之间的交互。可以将软件复查作为一个学习、讲授、援之以手和接受帮助的机会。

1.2 如何阅读本书

在本书中，我们展示了重要的代码阅读技术，并概括了常见的编程思想在实际应用中的具体形式，致力于提高您代码阅读的能力。虽然，您将会在后面的章节中看到许多有关重要的计算机科学和计算实践思想的讨论，如：数据和控制结构、编码规范和软件构架等，但是，由于本书的目的是引导您分析这些思想在代码生产中的应用，而不是介绍这些思想本身，所以对它们的处理必然比较粗略。我们对这些材料进行了组织，使得读者从基本的内容开始，逐渐过渡到更为复杂的部分。然而，这本书是一本教科书，不是侦探小说，可以按照您自己的意愿与兴趣，自由地选择阅读的先后次序。

1.2.1 印刷约定

所有的代码清单和引用程序元素的文字（例如：函数名、关键字、运算符）都设为代码字体。部分例子中用到了在 Unix 或 Windows 外壳（shell）中执行的命令序列。我们用命令提示符 $ 表示 Unix 外壳命令，用 DOS 的命令提示符 C:\>来表示 Windows 控制台提示符。Unix 外壳命令可以扩展到多行；我们使用>作为续行符号。

```
$ grep -l malloc *.c |
> wc -l
       8
C:\> grep -l malloc *.c | wc -l
       8
```

显示提示符和续行符号只是为了将您的输入与系统的输出区分开来：您只需要输入提示符后面的命令。

在本书后面的一些地方，我们将讨论不安全的编码实践或常见的陷阱。当进行代码走查，或只是阅读代码寻找 bug 时，应该警惕这类代码。当我们阅读文章时，我们都是识别整个单词而非单个的字母；同样，识别代码中的习惯用法可以更快、更高效地阅读代码，从更高的层次来理解程序。

本书中使用的代码示例都来自于实际程序。我们在脚注①中注明使用的程序（如图 1.1 所示），给出该程序在本书源代码目录树中的精确位置，以及具体的代码段所涉及的行号。当一个图包括不同源代码文件中的代码时（图 5.17 即为如此），脚注将指出这些文件所在的目录②。

有时，我们会省略代码的某些部分，用省略号将它们标示出来[...]。在这些情况下，行号代表列出的代码所覆盖的整个范围。和原来的代码相比，其他方面的更改包括：大部分 C 声明都从老式的“Kernighan and Ritchie”风格改为 ANSI C 风格，省略了部分注释、空格和程序许可信息。我们希望，这些更改能够在不过分影响原来示例的真实性的基础上，增加所提供示例的易读性。对于重要的代码示例，我们使用一个定制的软件应用程序对其进行了图形化注解。使用注解软件保证了示例的正确性，并能进行计算机验证。有时，我们会用叙述性的文字对一个注解进行详述。在这些情形中（图 1.1:1），注解从一个印在方框中的数字开始；相同的数字——在冒号后面，用来在正文中引用该注解。

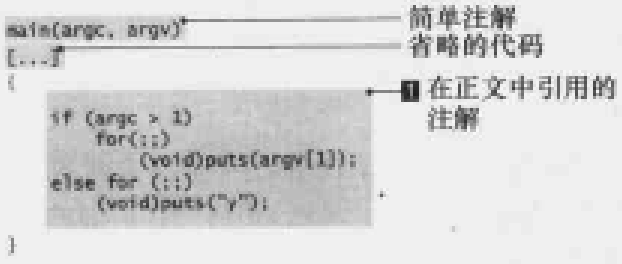


图 1.1 含有注解的程序清单

1.2.2 图解

我们选择 UML 来表达我们的设计图（design diagram），这是因为它已经是事实上的行业标准。在准备本书的过程中，我们发现，开发一个开放源码的说明性语言，用来生成 UML 图很有用③，同时，我们对 GraphViz④ 工具的底层代码库做了一些小的改进。我们希望生成的 UML 图（UML diagram）能够帮助您更好地理解书中分析的代码。

图 1.2 展示出在我们的图示中使用的一些记法（notation）。请记住：

- 我们使用 UML 活动类（active class）的记法——边框加粗的类框（class box）（参见图 6.14），来绘制过程（如过滤器风格（filter-style）的程序）。

① netbsdsrc/usr.bin/yes/yes.c:53-64

② netbsdsrc/distrib/utils/more

③ *http://www.spinellis.gr/sw/umlgraph*

④ *http://www.graphviz.org*

- 我们用关联导航(association navigation)关系——带有开放箭头的实线,来描绘数据元素之间的指针。我们还将每个数据结构划分成水平或垂直的分隔间,以便更好地描绘内部的组织结构(参见图 4.10)。
- 我们用关联(association)线上的实心箭头表示关联的方向(例如,为了说明数据的流向),而不是按照 UML 所描述的那样在关联线的顶部(参见图 9.3)。

所有其他的关系都使用标准的 UML 符号。

- 类的继承关系绘制为泛化关系(generalization relationship,也称为类属关系):带有中空箭头的实线(参见图 9.6)。
- 接口实现(interface implementation)绘制为实现关系(realization relationship):带有中空箭头的虚线(参见图 9.7)。
- 两个元件之间的依赖(dependency)(例如,编译过程中文件之间的关系)用带有开箭头的虚线来表示(参见图 6.8)。
- 组合(composition)(比如,一个库由各种模块组成)通过聚集关联(aggregation association)来描绘:一端为菱形的线(参见图 9.24)。

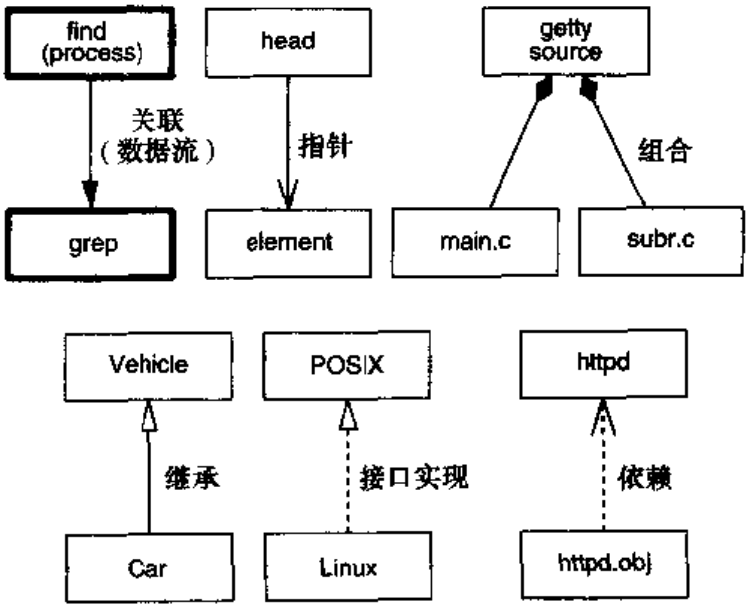


图 1.2 基于 UML 的图解记法

1.2.3 练习

大多数节的结尾都有一些练习,目的是鼓励您应用我们讲述的技术,并进一步研究感兴趣的特定问题,也可以将它们作为进一步讨论的起点。大多数实例中,您可以交替使用本书配套盘中的内容和"您工作中的实际代码"。重要的是要阅读与分析来自现实世界中重要系统的代码。如果您正在这样的系统上工作(无论是专利的开发工作,还是开放源码的项目),那么,将代码阅读的重点放到该系统上要比放在本书的配套盘上更卓有成效。

许多练习开始时都要求您定位特定的代码序列。这项任务可以自动完成。首先,将您正在寻找的代码表达为一个正则表达式(regular expression)。(关于正则表达式的更多内容请参见第 10 章。)之后,在代码库中使用命令进行查找,比如 Unix 环境中的下列命令:

```
find / cdrom - name '*.c' - print | xargs grep 'malloc.*NULL'
```

或在 Windows 环境中使用 Perl 脚本 codefind. pl[①]。(源代码库中的某些文件和老的 MS-DOS 设备同名,当试图访问它们时,某些版本的 Windows 实现会挂起;Perl 脚本显式地解决了这个问题。)

1.2.4 辅助材料

本书中的所有示例都基于现有的开放源码的软件代码。源代码库由超过 53 000 个文件组成,大小超过 540MB。所有对代码示例的引用都在脚注中明确地标出,所以您能够在代码原来的环境中对它们进行分析。另外,您可以用 3 种不同的方式,将源代码库与本书的正文对应起来。

(1)在索引中查找每个引用源文件的文件名(全文件路径的最后部分)。

(2)浏览附录 A,它提供了源代码库的概况。

(3)搜索附录 C,附录 C 中包含书中引用的源代码文件列表,按照代码的目录结构排序。

1.2.5 工具

我们提供的示例中,有一些依赖于类 Unix 操作系统中的某些程序,比如 grep 和 find。大量的此类系统(例如,FreeBSD,GNU/Linux,NetBSD,OpenBSD 和 Solaris)现在都可以免费下载,并能够安装到十分广泛的各种硬件平台之上。如果您使用的不是这类系统,通过利用已经完成的对这些工具的移植工作,也可以享受到这些工具提供的便利。(10.9 节包含工具可用性的更详尽信息。)

1.2.6 大纲

在第 2 章中,我们给出两个完整的程序,并一步一步地分析它们是如何工作的。为此,我们概述了一些基本的代码阅读策略,并标识出常见的 C 控制结构、构建块、习惯用法和陷阱。我们将 C 语言中一些更为高级(同时也易于误用)的元素放到第 3 章和第 5 章讨论。第 4 章分析如何阅读代码,收录了常用的数据结构。第 6 章介绍如何处理真正大型项目中的代码:分布在不同地点的小组共同工作,包括数千个文件和数以百万行的代码。大型项目一般采用共同的编码规范和约定(在第 7 章中讨论),并且可能包括正式的文档(在第 8 章中介绍)。第 9 章提供背景信息,并且建议要从大处着眼:系统的构架,而非具体的代码。阅读代码时,有大量的工具可供使用,这是第 10 章的主题。最后,第 11 章中包含一个完整的例子,它使用本书其他部分讲述的代码阅读和代码理解技术,在 NetBSD 的源代码库中找出并提取月相算法,将其作为 SQL 函数加入到基于 Java 的 HSQL 数据库引擎中。

在本书的最后,以附录的形式概括了我们在示例中使用的代码和附书光盘中的代码(附录 A);同时,列出了一些阅读代码的格言(附录 B)。

1.2.7 "重要语言"的争论

本书中的大部分示例都基于 C 程序,这些程序在 POSIX 字符终端环境上运行。这项选

① tools/codefind. pl

择背后的原因与开放源码软件中丰富的可移植C代码，以及C示例的简明性有关——与用C++或Java编写的类似示例相比。（这一现象背后的原因，极有可能与代码的年龄或流行的编码风格有关，而非语言自身的特征。）很遗憾，我们的例子中很少出现基于图形用户界面的程序，阅读与推理此类程序真的需要单独的一本书才能完成。无论何时我们提及的Microsoft Windows API函数都是指Win32 SDK API，而非.NET平台。

常常有人问及用来编写开放源码软件的语言。表1.1汇总了SourceForge.net储存库(repository)[①]中前10个最常使用的语言开发的项目数量。C语言占据了列表顶部的重要位置，并且可能还没有充分表示出其所占的份额，因为许多超大型的C开放源码项目，如FreeBSD和GNU/Linux都存放在独立的主机之上，许多项目声称使用C++，但实际上是用C语言编写而成，只使用了很少的C++特性。另一方面，还要注意，汇编非开放源码项目的类似列表可能会全然不同，COBOL，Ada，Fortran和各种4GL语言可能会占据列表中顶部的位置。另外，如果您正在维护代码——阅读代码的一个很流行的理由——您将阅读的语言极有可能是5（如果幸运的话）或10年前所采用的语言，反映出那个年代编程语言的特点。

表1.1 开放源码项目中最常使用的10种语言

语言	项目数量	占项目的百分比
C	8 393	21.2
C++	7 632	19.2
Java	5 970	15.1
PHP	4 433	11.2
Perl	3 618	9.1
Python	1 765	4.5
Visual Basic	916	2.3
Unix Shell	835	2.1
Assembly	745	1.9
JavaScript	738	1.9

我们的示例所表达的代码结构大多数情况下也适用于Java和C++程序；大多数还适用于Perl，Python和PHP。您可以跳过本书中下面的部分：

- 第3章，如果您从未遇到过C（或C++）代码；
- 9.3.4节，如果您不喜欢C++和Ada；
- 9.1.3节，如果您计划避免面向对象的解决方案、C++、Java、Python、Smalltalk和Perl中面向对象的特性；
- 5.2节，如果您不涉及Java或C++；
- 2.8节，如果您是结构化编程的狂热者，或Java的热心支持者。

① *http://sourceforge.net/softwaremap/trove_list.php?form_cat=160*

进 阶 读 物

Armour[Arm00]对软件做为存储可执行知识的介质这一主题进行了较为详细的描述。Petzold 的 Code[Pet99]一书对计算机技术中使用的各种编码系统做了介绍，十分易读，适合于所有人阅读。Kernighan 和 Plauger 提倡将仔细研究和效仿好的程序，作为提高编写代码能力的一种方法[KP76]。阅读与理解代码也是软件维护工作中一个不可或缺的部分。Pressman[Pre00]在第 30 章对软件维护的开销进行了分析，更详细的分析可以参考 Boehm 的著作[Boe81, Boe83]。Glass 介绍了代码阅读在开放源码世界中的社会规模[Gla00]。Knuth 的"文学化编程(*Literate Programming*)"方法学中提倡在编写代码过程中要考虑到阅读的需要。更易读的介绍可以参见 Bentley 的"Programming Pearls"栏目[BK86]中 Knuth 的原创[Knu92]，Knuth 编著的两本文学化编程的书籍[Knu86a, Knu86b]，以及一系列基于最初思想的文章[BKM86, Han87, Jac87, Ham88, WL89]。有关阅读老的计算机代码的行为，最近的文献是 *software archeology*[HT02]。

经过了二十几年，现在 Lions 的书籍有了不受限制的版本[Lio96]。Raymond[Ray01]介绍了开放源码软件的开发过程。El-Emam[El－01]讨论了将开放源码软件作为研究(和教学)材料时，应该遵循的道德尺度。社交过程曾经作为科学沟通的媒介，为数据定理的成功作出了贡献，在 DeMillo 等著的[DLP77]中进行介绍。面向方面的编程(Aspect-oriented programming)对软件维护人员来说是一个有用的补充工具；请阅读 Kiczales 等著的[KHH+01]中 AspectJ 的介绍以及期刊中的其他文章。Lientz 和 Swanson[LS80]对软件维护的管理(以及在 1.1.4 节中使用的工作图)进行了分析；Fowler[Fow00]中介绍了更多有关重构的内容。Beck 编著了极限编程的权威介绍[Bec00]；还可以在网络上找到相应的敏捷软件开发宣言(*Manifesto for Agile Software Development*)①。[Kru92, MMM95]概述了最重要的软件问题——重用。[Spi99a, SR00]描述了基于开放源码的代码，进行进程级别软件重用的实例。Boehm 等著的[BCH+95]首次提出了 COCOMO II 软件成本模型。

① *http://www.AgileManifesto.org/*

第2章　基本编程元素

我们所观测到的不是自然本身，而是大自然在我们所用的观察方法下展现出来的特性。

——Werner Heisenberg

多数情况下，代码阅读是一项自底向上(bottom-up)的活动。本章中，我们将检查组成程序的基本代码元素，并简要说明如何阅读和推理它们。在2.1节中，我们将剖析一个简单的程序，借以说明代码阅读过程中必需的一些推理类型。我们还会首次认识一些常见的陷阱与隐患，在阅读和编写代码时应该特别注意它们，同时还会接触到一些习惯用法，它们有助于对代码含义的把握。2.2节以及其后的内容分析组成程序的函数、控制结构和表达式，加深我们的理解。此外，我们还将推理一个具体的程序，同时分析C，C++，Java和Perl中常见的控制结构。前两个完整的示例都是C程序。然而，我们在此介绍的大部分概念和结构都适用于任何用C的派生语言编写的程序，如C++，C#，Java，Perl和PHP。在本章的最后一节中，我们详细描述，如何在抽象层推理程序的控制流程，从程序的代码元素中解析出语义上的含义。

2.1　一个完整的程序

echo是Unix系统上一个十分简单，但比较实用的程序。它在标准输出上(一般为屏幕)打印出它的参数。该程序经常用来向用户显示信息，如下面的语句所示——取自NetBSD中的upgrade脚本[①]：

```
echo "Cool! Let's get to it..."
```

图2.1是echo的全部源代码[②]。

可以看到，超过一半的程序代码是法律与管理信息，如版权信息、许可信息和程序版本标识符。大型的、组织有序的系统通常都会提供此类信息，同时还会归纳具体程序或模块的功能。重用(reuse)来自开放源码组织的源代码时，要注意版权公告规定的许可要求(图2.1:1)。

C和C++程序中，使用库函数需要包括相应的头文件(图2.1:2)。函数库的文档中一般会列出每个函数所需的头文件。如果在没有包括正确头文件的情况下使用库函数，C编译器常常只给出警告，但程序在运行时会失败。因此，代码阅读过程中可以使用的武器之一就是：用编译器对代码进行编译，检查产生的警告消息(参见10.6节)。

标准C，C++和Java程序从函数(Java中为"方法")main(图2.1:3)开始执行。第一次

① netbsdsrc/distrib/miniroot/upgrade.sh:98

② netbsdsrc/bin/echo/echo.c:3-80

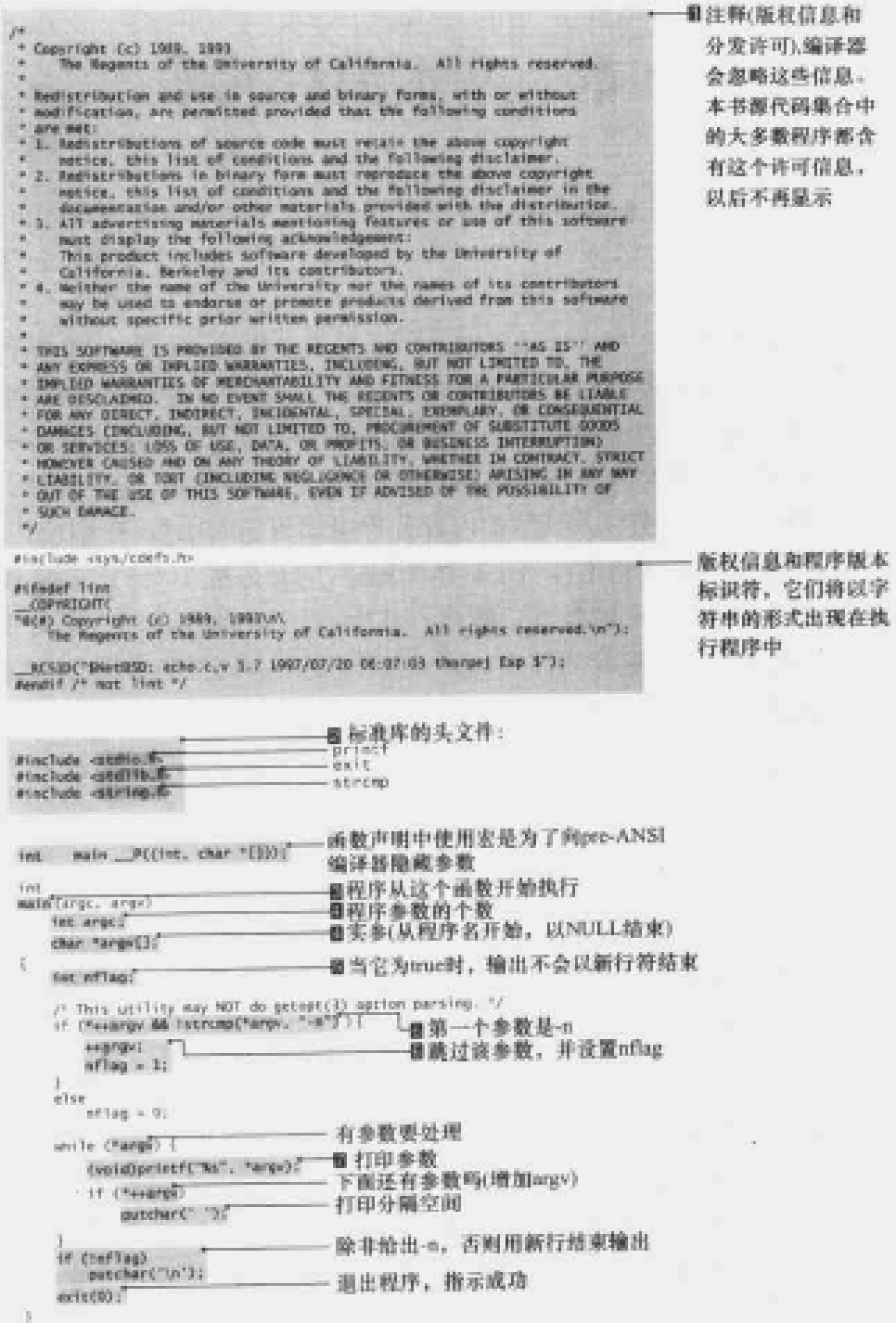


图 2.1 Unix 程序 echo

分析一个程序时,main 是一个好的起始点。要注意,一些操作环境,如 Microsoft Windows、Java applet(小应用程序)和 servlet(小服务程序)的宿主、掌上电脑和嵌入式系统可能使用其他函数作为程序的入口点,例如 WinMain 或 init。

在 C/C++程序中,main 函数的两个参数(通常命名为 argc 和 argv)用来将指定的命令行参数从操作系统传递到程序。argc 变量存储程序参数的个数,argv 是一个字符串数组,包含所有的实参(包括程序自身的名称——位置 0,即 argv 的第一个元素)。argv 数组由一个 NULL 元素结束,因此可以用两种不同的方式对参数进行处理:或者基于 argc 进行计数;或

者遍历 argv，将每个值与 NULL 进行比较。在 Java 程序中，String 数组 argv 及其 length 方法用作相同的目的，Perl 代码中等价的构造是@ARGV 数组和 $ # ARGV 标量。

在上面的例子中，argc 和 argv 的声明不太常见。main 函数的典型 C/C++定义是：①

```
int
main(int argc, char ** argv)
```

相应 Java 类方法的定义是：②

```
public static void main(String args[]) {
```

图 2.1:4 中的定义使用旧式(pre-ANSI C)的 C 语法，也称为是 K&R C。在以前的程序中，您可能遇到过这样的函数定义；要注意，(不同风格的函数定义)参数传递的方式稍有不同，编译器所做的检查也依赖于函数定义的风格。

在分析命令行程序时，您会发现，参数的处理要么通过手动编写代码，要么使用 getopt 函数——在 POSIX 环境中。Java 程序可以使用 GNU gnu.getopt 包③来完成参数的处理。

echo 命令的标准定义与 getopt 的行为并不兼容；所以，要手工编写代码处理-n 参数(指定输出不以新行结束)(图 2.1:6)。在比较开始时，首先将 argv 向前移动到 echo 的第一个参数(别忘记位置 0 是程序名)，并检验该参数是否存在。只在该参数存在的情况下，才调用 strcmp 将该参数与"-n"进行比较。首先检查参数是否合法，接下来使用该参数，中间用布尔与运算符(&&)将二者组合起来，这种序列是一种常见的习惯用法。之所以能够采用这种用法，是因为如果左操作数求值结果为 false，&& 运算符就不再对右操作数求值。许多操作环境中，在调用 strcmp 或任何其他字符串函数时，如果传递给这些函数的不是指向实际字符数据的指针，而是 NULL 值，会引起程序崩溃。

请注意，在比较两个字符串的相等性时，函数 strcmp 的返回值很不直观。当字符串相等时，它返回 0，即 C 语言中的 false。为此，很多 C 程序中会定义一个 STREQ 宏，使之在两个字符串相等时返回 true，在这个宏中，常常通过立即比较前两个字符来优化比较过程。④

```
# define STREQ(a, b) (* (a) == * (b) && strcmp((a), (b)) == 0)
```

幸运的是，Java 中 equals 方法的行为阅读起来更为直观：⑤

```
if (isConfig) {
    configFile = args[i];
    isConfig = false;
} else if (args[i].equals("-config")) {
    isConfig = true;
} else if (args[i].equals("-debug")) {
    debug = true;
} else if (args[i].equals("-nonaming")) {
```

① netbsdsrc/usr.bin/elf2aout/elf2aout.c:72-73

② jt4/catalina/src/share/org/apache/catalina/startup/Catalina.java:161

③ *http://www.gnu.org/software/java/packages.html*

④ netbsdsrc/usr.bin/file/ascmagic.c:45

⑤ jt4/catalina/src/share/org/apache/catalina/startup/CatalinaService.java:136-143

上面的序列还介绍了用层叠 if 语句表达选择结构时，另一种处理缩进(indentation)的方式。层叠 if-else if-...-else 序列可以看作是由互斥选择项组成的选择结构。

检查-n 标志的 if 语句具有一个重要的特征，那就是 nflag 总会被赋予一个值：0 或 1。在定义 nflag 时，并没有给出一个初始值(图 2.1:5)。因此，在它被赋值之前，它的值是未定义的：该值是存储 nflag 的内存当前的随机数值。使用未经初始化的变量是引发问题的一个常见原因。在检查代码时，一定要核实：是否所有的程序控制路径在使用变量前都恰当地对它们进行了初始化。一些编译器可以检测出部分此类错误，但是不应该依赖于这项功能。

接下来的代码循环处理所有剩余参数，将它们用空格符隔开打印出来，这部分代码比较直观。通过使用 printf 和字符串格式化说明将每个参数打印出来，避免了一个潜在的隐患(图 2.1:7)。prinf 函数总会将它的第一个参数——格式说明，打印出来。因此，有时候程序中会通过格式说明参数将字符串变量直接打印出来：①

```
printf(version);
```

如果将任意字符串作为格式说明传递给 printf，想将其打印出来，那么，当这些字符串中包含转换说明(conversion specification)时(例如，上面的 SCCS 修订控制标识符就包含%字符)，printf 会产生不正确的结果。

即使如此，printf 和 putchar 的使用也并非完全正确。请注意 printf 的返回值被转换成 void 类型。printf 返回实际打印出的字符数；向 void 类型的转换是想告诉我们这个结果被有意忽略。类似地，如果未能写出字符，putchar 将返回 EOF。所有输出函数——特别在程序的标准输出重定向到文件时——都有可能会由于各种各样的原因而失败。

- 存储输出的设备可能没有剩余空间；
- 设备上分配给用户的空间可能耗尽；
- 进程对文件的写入可能超出进程或系统文件大小的最大限制；
- 输出设备上可能发生硬件错误；
- 文件描述符或与标准输出关联的流或许不能写入。

不检查输出操作的结果可能会引起程序悄然失败，在没有任何警告的情况下丢失输出。检查所有输出操作的结果可能会很麻烦。一个实用的折衷办法是在程序结束前检查标准输出流上的错误。在 Java 中，可以使用 CheckError 方法来完成这项工作(在实践中，我们甚至看到过在标准输出流上采用这种方法：即使是 JDK 中的一些程序，当输出设备的空间用完时，也会在不给出任何错误的情况下失败)；在C++ 程序中，可以使用流的 fail，good 或 bad 方法；在 C 代码中使用 ferror 函数：②

```
if (ferror(stdout))
    err(1, "stdout");
```

在用一个新行结束了它的输出后，echo 调用 exit 结束程序，用(0)表示成功。我们还会经常看到在 *main* 函数中返回 0，这两种做法的结果相同。

---

① netbsdsrc/sys/arch/mvme68k/mvme68k/machdep.c:347

② netbsdsrc/bin/cat/cat.c:213-214

练习 2.1　试着找出 C,C++和 Java 编译器如何处理未初始化的变量。概括结果,并就如何定位未初始化的变量提出检查流程。

练习 2.2　(Dave Thomas 建议)为什么 echo 程序不能使用 getopt 函数呢?

练习 2.3　讨论定义类似 STREQ 宏的优点与缺点。考虑 C 编译器能够如何优化 strcmp 调用。

练习 2.4　检查您的环境中或本书配套盘中的程序,看看它们中间是否有不检验库调用结果的情况。给出实用的修复建议。

练习 2.5　有时,要想了解程序在某一方面的功能,运行它可能比阅读源代码更为恰当。设计一个测试过程或框架,用以分析程序如何处理在它们标准输出上发生的输出错误。针对大量基于字符界面的 Java 和 C 程序,试用前面设计的测试过程或框架(比如命令行版本的编译器),并报告您的结果。

练习 2.6　找出使用库函数 sscanf, qsort, strchr, setjmp, adjacent_find, open, FormatMessage,和 XtOwnSelection 所需的头文件。后 3 个函数与操作环境相关,可能不存在于您所在的环境。

2.2　函数和全局变量

expand 程序将参数指定的文件(如果没有指定文件参数,则为标准输入)中硬制表符(\t,ASCII 字符 9)展开成若干空格。默认的行为是每 8 个字符设置一个制表位(tab stop);可以使用-t 选项指定一个由逗号或空格隔开的数字列表来改变默认的行为。该程序实现中一个有趣的特征,同时也是我们分析它的原因,就是它使用了 C 语言家族中所有控制流程的语句。图 2.2 列出了 expand 中变量和函数的声明①,图 2.3 是代码的主体②,图 2.5(在 2.5 节中)包含用到的两个补充函数③。

在分析重要的程序时,最好首先识别出重要的组成部分。就我们的情况来说,重要的组成部分是全局变量(图 2.2:1)和函数 main(图 2.3)、getstops(参见图 2.5:1)和 usage(参见图 2.5:8)。

整型变量 nstops 和整型数组 tabstops 是全局变量,在函数体的作用域之外。因此,对于该文件中的所有函数,它们都是可见的。

接下来是 3 个函数声明,声明了该文件后面将会出现的函数。由于它们中间的一些函数可能会在它们定义之前使用,在 C/C++程序中,这些声明允许编译器检验传递给函数的

① netbsdsrc/usr. bin/expand/expand. c:36-62

② netbsdsrc/usr. bin/expand/expand. c:64-151

③ netbsdsrc/usr. bin/expand/expand. c:153-185

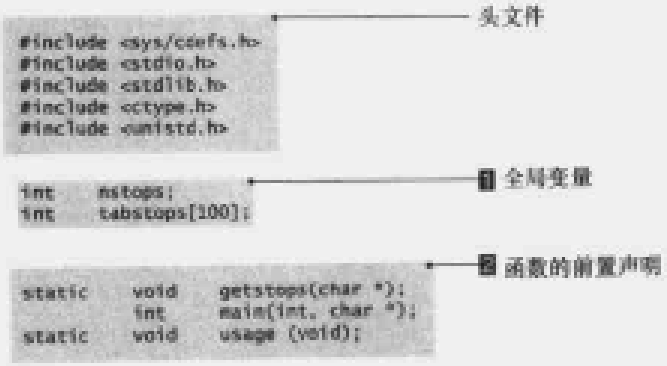


图 2.2 展开制表位(声明)

参数,以及它们的返回值,相应地生成正确的代码。如果没有给出前置声明(forward declaration),C 编译器将依据函数第一次使用时的情况对函数的返回值类型和参数做出假定;C++编译器将这种情况标记为错误。如果之后的函数定义与这些假定不相符,编译器将发出一条警告或错误消息。但是,如果定义于另一个文件中的一个函数满足这个错误的声明,则程序也许能够编译通过,但在运行时会失败。

请注意,两个函数被声明为 static,而变量并非 static。这意味着,这两个函数只在该文件中可见,而变量则对组成程序的所有文件都是可见的。由于 expand 只由一个文件组成,这个差别在此并不重要。大多数链接器(linker)——用来合并编译后的 C 文件,都十分原始:对所有程序文件都可见的那些变量(即没有声明为 static 的变量),可能会与定义于其他文件中同名的变量产生意外的相互作用。因此,在检查代码时,一个好的做法是确保所有只用于单一文件的变量都声明为 static。

现在,让我们来查看一下组成 expand 的函数。要了解函数(或方法)的功用,可以使用下面的策略。

- 猜,基于函数名;
- 阅读位于函数开始部分的注释;
- 分析如何使用该函数;
- 阅读函数体的代码;
- 查阅外部的程序文档。

就我们的具体情况,我们能够肯定地猜出函数 usage 将显示程序的用法信息,然后退出;许多命令行程序都拥有名称与功能相同的这样一个函数。在分析大量的代码时,您会逐渐得出变量和函数的名称和命名约定。这些知识有助于正确地猜出它们的用途。但是,您应该随时准备,根据代码阅读过程中必然会发现的一些新的证据,对最初的猜测做出修正。此外,当基于猜测修改代码时,您应该设计能够验证最初假设的过程。这个过程可能包括用编译器进行检查、引入断言、或者执行适当的测试用例。

getstops 的角色要难理解一些。此处没有任何注释,函数体中的代码比较复杂,难以容易地得出具体的含义,它的名称可以用不同的方式来解释。它用在程序中一个单独的部分(图 2.3:3),注意到这一点之后,我们就可以继续进行分析。程序中使用 getstops 的部分负责处理程序的选项(图 2.3:2)。因此,我们能够肯定地认为,getstops 将处理制表位说明(tab stop specification)选项。在阅读代码时,常常会采用渐进式理解的方式:理解了这一部

```
int
main(int argc, char *argv)
{
    int c, column;
    int n;

    while ((c = getopt (argc, argv, "t:")) != -1) {
        switch (c) {
        case 't':
            getstops(optarg);
            break;
        case '?': default:
            usage();
        }
    }
    argc -= optind;
    argv += optind;

    do {
        if (argc > 0) {
            if (freopen(argv[0], "r", stdin) == NULL) {
                perror(argv[0]);
                exit(1);
            }
            argc--, argv++;
        }
        column = 0;
        while ((c = getchar()) != EOF) {
            switch (c) {
            case '\t':
                if (nstops == 0) {
                    do {
                        putchar(' ');
                        column++;
                    } while (column & 07);
                    continue;
                }
                if (nstops == 1) {
                    do {
                        putchar(' ');
                        column++;
                    } while (((column - 1) % tabstops[0]) != (tabstops[0] - 1));
                    continue;
                }
                for (n = 0; n < nstops; n++)
                    if (tabstops[n] > column)
                        break;
                if (n == nstops) {
                    putchar(' ');
                    column++;
                    continue;
                }
                while (column < tabstops[n]) {
                    putchar(' ');
                    column++;
                }
                continue;
            case '\b':
                if (column)
                    column--;
                putchar('\b');
                continue;
            default:
                putchar(c);
                column++;
                continue;
            case '\n':
                putchar(c);
                column = 0;
                continue;
            }
        }
    } while (argc > 0);
    exit(0);
}
```

main中的局部变量

使用getopt进行参数处理

处理-t选项

聚合起来的switch标记

switch块的结尾

至少循环一次

处理其余的参数

读入字符，直到EOF为止

制表符

处理下一个字符

退格符

所有其他字符

新行符

switch块的结尾

while块的结尾

do块的结尾

图 2.3　展开制表位(主体)

分代码可能会使得其他部分变得十分明晰。基于这种渐进式理解的方式，您可以使用类似组合拼图游戏时采用的策略，来理解困难的代码：从容易的部分开始。

练习 2.7　分析周围程序中函数与变量的可见性。能够对其做出改进吗(使之变得更为

稳妥)?

练习 2.8 从本书配套盘,或周围的程序中,抽取一些函数或方法,使用我们概括的策略来确定它们的角色。尽量缩小花在每个函数或方法上的时间。依照这些策略成功的比率对它们进行排序。

2.3 while 循环、条件和块

现在,我们可以分析,程序中如何处理选项。虽然 expand 只接受一个简单的选项,但是它使用 Unix 库函数 getopt 来处理选项。图 2.4 是 getopt 函数 Unix 在线文档的一个汇总版本。大多数开发环境都提供库函数、类和方法的在线文档。在 Unix 系统上,可以使用 man 命令,在 Windows 上可以使用微软开发者网络资源库(Microsoft Developer Network Library,MSDN)①,而 Java API 的文档都是 HTML 格式,作为 Sun JDK 的一部分提供。要养成遇到库元素就去阅读相关文档的习惯;这将会增强您阅读和编写代码的能力。

基于我们对 getopt 的理解,现在,我们可以分析相关的代码(图 2.3:2)。传递给 getopt 的选项字符串可以是简单选项-t,后面将会跟上一个参数。getopt 在 while 语句中用作条件表达式。只要圆括号中指定的条件为 true(在 C/C++ 中,它求值的结果不为 0),while 语句就会重复地执行其循环体。在我们的例子中,while 循环的条件是,调用 getopt,将其返回的结果赋给 c,并将 c 与 1 进行比较,1 这个值是用来表示已经处理完所有选项。为了在单一表达式中执行所有这些操作,该代码利用了下述事实:在 C 语言家族中,赋值由运算符(=)来执行,也就是说,赋值表达式有一个值。赋值表达式的值就是在赋值发生后,左操作数中存储的值(本例中为变量 c)。许多程序都会在单一表达式中调用函数,将其返回值赋给一个变量,并将结果与某种特殊情况下的值进行比较。下面这个典型的例子将 readLine 的结果赋给了 line,并将它与 null 进行比较(null 表示已经到达流的结尾)②。

```
if ((line = input.readLine()) == null) [...]
    return errors;
```

一定要将赋值语句包括在圆括号之内,如我们分析过的两个例子中那样。由于比较运算符(一般与赋值结合使用)比赋值结合更为紧密,所以下面的表达式:

```
c = getopt(argc, argv, "t:") != -1
```

在求值时,将会作为:

```
c = (getopt(argc, argv, "t:") != -1)
```

从而,赋给 c 的是 getopt 的返回值与-1 比较的结果,而非 getopt 的返回值。此外,用来接受函数调用结果的变量,应该能够保存正常的函数返回值和任何标识错误的异常值。因此,典型地,那些既返回字符(如 getopt 和 getc),又可能返回错误值(如-1 或 EOF)的函数将

① *http://msdn.microsoft.com*

② cocoon/src/java/org/apache/cocoon/components/language/programming/java/Javac.java:106-112

GETOPT(3)　　UNIX Programmer's Manual　　GETOPT(3)

NAME

getopt - get option character from command line argument list

SYNOPSIS

```
#include <unistd.h>

extern char *optarg;
extern int optind;
extern int optopt;
extern int opterr;
extern int optreset;

int
getopt(int argc, char *const *argv, const char *optstring)
```

DESCRIPTION

The **getopt**() function incrementally parses a command line argument list *argv* and returns the next *known* option character. An option character is *known* if it has been specified in the string of accepted option characters, *optstring*.

The option string *optstring* may contain the following elements: individual characters, and characters followed by a colon to indicate an option argument is to follow. For example, an option string "x" recognizes an option "-x", and an option string "x:" recognizes an option and argument "-x *argument*". It does not matter to **getopt**() if a following argument has leading white space.

On return from **getopt**(), *optarg* points to an option argument, if it is anticipated, and the variable *optind* contains the index to the next *argv* argument for a subsequent call to **getopt**(). The variable *optopt* saves the last *known* option character returned by **getopt**().

The variable *opterr* and *optind* are both initialized to 1. The *optind* variable may be set to another value before a set of calls to **getopt**() in order to skip over more or less argv entries.

The **getopt**() function returns -1 when the argument list is exhausted, or a non-recognized option is encountered. The interpretation of options in the argument list may be cancelled by the option '--' (double dash) which causes **getopt**() to signal the end of argument processing and returns -1. When all options have been processed (i.e., up to the first non-option argument), **getopt**() returns -1.

DIAGNOSTICS

If the **getopt**() function encounters a character not found in the string *optstring* or detects a missing option argument it writes an error message to *stderr* and returns '?'. Setting *opterr* to a zero will disable these error messages. If *optstring* has a leading ':' then a missing option argument causes a ':' to be returned in addition to suppressing any error messages.

Option arguments are allowed to begin with "-"; this is reasonable but reduces the amount of error checking possible.

HISTORY

The **getopt**() function appeared 4.3BSD.

BUGS

The **getopt**() function was once specified to return EOF instead of -1. This was changed by POSIX 1003.2-92 to decouple **getopt**() from<stdio.h>.

4.3 Berkeley Distribution　　April 19, 1994　　1

图 2.4　getopt 手册页

它们的结果存储在整型变量中，而非字符型变量，来保存所有字符和异常值的超集(图 2.3:7)。下面是同一构造的另一种典型应用：从文件流 pf 中复制字符到文件流 active 中，直到 pf 文件结束为止[①]。

```
while ((c = getc(pf)) != EOF)
    putc(c, active);
```

① netbsdsrc/usr.bin/m4/eval.c:601-602

while 语句的循环体既可以是单个语句，也可以是一个代码块——括在花括号中的一个或多个语句。所有控制程序流程的语句，也就是 if，do，for 和 switch，都是如此。程序中一般会将组成控制语句的相关语句（即组成 if，do，for 或 switch 代码体的语句）缩进。但是，缩进只是对人类程序阅读者的一种直观提示；如果没有给出花括号，控制结构将只影响控制语句之后的单个语句，不管缩进如何。例如，下面代码的实际执行情况就与代码缩进给出的暗示不符①。

```
for (ntp = nettab; ntp != NULL; ntp = ntp -> next) {
    if (ntp -> status == MASTER)
        rmnetmachs(ntp);
        ntp -> status = NOMASTER;
}
```

for 循环的每次迭代，代码行 ntp-> status=NOMASTER; 都将被执行，而非仅仅当 if 条件为 true 时才执行。

练习 2.9 看看您正在使用的编辑器能否标识出匹配的花括号和圆括号。如果它做不到，请考虑换另外的编辑器。

练习 2.10 expand 的源代码中有一些多余的花括号。请找出它们。检查所有不使用花括号的控制结构，并标出将会执行的语句。

练习 2.11 验证 expand 中的缩进是否与控制流程匹配。对您手头的一些程序也进行同样的尝试。

练习 2.12 Perl 语言要求所有的控制结构都使用花括号。评论这会对 Perl 程序的易读性造成什么样的影响。

2.4 switch 语句

getopt 正常的返回值由一个 switch 语句来处理。switch 语句用于处理若干离散的整型或字符类型的值。处理每个值的代码之前是一个 case 标记。当 switch 语句中表达式的值与 case 标记中的某个值匹配时，程序将从该点向前执行语句。如果没有标记值与表达式的值匹配，并且存在 default 标记，则控制将转移到该处；否则，switch 块中没有代码会执行。请注意，执行控制权转移到一个标记后，遇到其他标记不会结束 switch 块中语句的执行；要停止处理 switch 块中的代码，跳转到 switch 语句之外继续进行，必须执行一个 break 语句。这项特性经常用来将多个 case 标记归到一起，合并公共的代码元素。在我们的例子中，当 getopt 返回 't' 时，处理-t 的语句就会得到执行，其后的 break 语句使得执行控制权立即转移到 switch 块的右花括号之后（图 2.3:4）。另外，我们可以看到，由于 switch 语句的 default 标记和错误返回值？对应的标记被归到一起，所以它们的代码是公共的。

① netbsdsrc/usr.bin/timed/timed/timed.c:564-568

给定 case 或 default 标记的代码，在结束时如果没有语句（比如 break，return，或 continue）将控制权转移到 switch 块外部，程序将继续执行下一个标记后的代码。分析代码时，要留心这种错误。极少数情况下，程序员或许真的希望发生这种行为。为了警告维护人员注意这种情况，一般用一个注释标记这些地方，比如 FALLTHROUGH，如下面的例子所示①。

```
case 'a':
    fts_options |= FTS_SEEDOT;
    /* FALLTHROUGH */
case 'A':
    f_listdot = 1;
    break;
```

上面的代码来自于 Unix 中 ls 命令的选项处理部分，ls 用来列举目录中的文件。选项-A 将以点号开头的文件也列出来（按约定来说是隐藏文件），而选项-a 对这个行为做出修改，将两个目录项加入到列表中。一些自动验证源代码中常见错误的程序，如 Unix 中的 lint 命令，能够使用 FALLTHROUGH 注释抑制假性的警告。

缺少 default 标记的 switch 语句会默然忽略意外的值。即使知道 switch 语句只处理一系列确定的值，也要尽量包括 default 标记，这是一个好的保护性编程习惯。这类 default 标记能够捕获产生意外值的编程错误，并能警告程序的维护人员，如下面例子所示②。

在我们的例子中，该 switch 语句可以处理 getopt 返回的两种值。

(1)返回't'是为了处理-t 选项。Optind 将指向-t 的参数。通过调用函数 getstops，将制表符说明作为其参数，来进行处理。

(2)遇到未知选项或 getopt 发现其他错误时，返回'?'。这种情况下，usage 函数会打印程序的使用信息，并退出程序。

switch 语句还用作程序中字符处理循环的一部分（图 2.3:7）。程序对每个字符进行分析，其中某些字符（制表符、新行符和退格符）受到特殊处理。

练习 2.13　本书的源代码集合中，switch 语句的代码体与其他语句的格式不同。说明这些代码中使用的编排规则，并解释这样做的理由。

练习 2.14　在您阅读的代码中，分析 switch 语句中对意外值的处理。建议如何更改以便检测出错误的值。讨论在生产环境中，这些更改对程序健壮性的影响。

练习 2.15　在您的操作环境中，是否有一种工具或一个编译器选项可以用来检测 switch 代码中 break 语句的缺失？在一些示例子程序上使用它，并分析产生的结果。

2.5　for　循　环

为了完善我们对 expand 如何处理命令行选项的理解，现在，我们需要分析 getstops 函

① netbsdsrc/bin/ls/ls.c:173-178

② netbsdsrc/usr.bin/at/at.c:535-561

数。虽然仅仅从该函数仅有的参数 cp 的名字不能得出它的作用，但是，当我们分析了 getstops 的用法后，它的作用就会很明显。传递给 getstops 的是-t 选项的参数，也就是制表位(tab stop)的列表，例如：4,8,16,24。前面概括的用来确定函数角色的策略(2.2 节)也可以用在它们的参数上。从而，一个阅读代码的模式已经慢慢形成。代码阅读有许多可选择的策略：自底向上(bottom-up)和自顶向下(top-down)的分析、应用试探法和检查注释和外部文档，应该依据问题的需要尝试所有这些方法。

在将 nstops 设为 0 后，getstops 进入一个 for 循环。for 循环一般由 3 个表达式指定：在循环开始之前求值的表达式、每次迭代前都要求值以确定是否进入循环体的表达式、以及循环体执行结束后求值的表达式。for 循环经常用于对一段代码体执行指定的次数。①

```
for (i = 0; i < len; i++ ) {
```

这种类型的循环在程序中应用十分普遍；应该学会将它们读作“执行代码体 len 次。”另外，和这个风格的任何差异，比如：初始值不是 0 或比较运算符不是<都警告您要小心地推理该循环的行为。请考虑下面的例子中代码体被执行的次数。

循环 extrknt +1 次：②

```
for (i = 0; i <= extrknt; i++ )
```

循环 month -1 次：③

```
    for (i = 1; i < month; i++ )
```

循环 nargs 次：④

```
for (i = 1; i <= nargs; i++ )
```

请注意，最后一个表达式不必一定是递增运算符。下面的行将会循环执行 256 次，在这个过程中递减 code：⑤

```
for (code = 255; code >= 0; code -- ){
```

另外，for 语句有时还用于循环处理由库函数返回的结果集。下面的循环将针对 dir 目录中的所有文件执行⑥。

```
if ((dd = opendir(dir)) == NULL)
    return (CC_ERROR);
for (dp = readdir(dd); dp != NULL; dp = readdir(dd)) {
```

对 opendir 的调用返回一个值，这个值可以传递给 readdir，从面顺序地访问 dir 目录中的每个项。当该目录中没有更多的项时，readdir 将返回 NULL，循环也将结束。

---

① cocoon/src/java/org/apache/cocoon/util/StringUtils.java:85

② netbsdsrc/usr.bin/fsplit/fsplit.c:173

③ netbsdsrc/usr.bin/cal/cal.c:332

④ netbsdsrc/usr.bin/apply/apply.c:130

⑤ netbsdsrc/usr.bin/compress/zopen.c:510

⑥ netbsdsrc/usr.bin/ftp/complete.c:193-198

描述 for 语句的 3 个部分是表达式，不是语句。因此，如果在循环开始或每次迭代结束时需要执行多个操作，不能用花括号将表达式聚合到一起。然而，我们经常会看到这种用法，使用逗号(,)运算符将表达式序列化，以此将表达式聚合到一起。①

```
for (cnt = 1, t = p; cnt <= cnt_orig; ++ t, ++ cnt) {
```

通过逗号运算符结合起来的两个表达式的值，只是第二个表达式的值。在我们的例子中，对表达式求值只是为了它们的副作用：在循环开始前，将 cnt 设为 1，t 设为 p，每次循环迭代后，t 和 cnt 都递增 1。

for 语句的任何表达式都可以省略。当第二个表达式缺失时，则取其为 true。许多程序使用 for (;;) 形式的语句来执行“无限”循环。只有很少的情况，此类循环真的是无限。下面的例子——取自 init(不间断地循环，控制所有的 Unix 进程)——是一个例外。②

```
for (;;) {
    s = (state_t) (*s)();
    quiet = 0;
}
```

大多数情况下，无限循环用来表达在循环开始或结束时退出条件无法指定的循环。这些循环一般不是通过 return 语句退出函数，就是通过 break 语句退出循环体，或者调用 exit 或类似的函数退出整个程序。C++，C# 和 Java 程序还可以通过异常跳出这类循环。

快速地浏览一下图 2.5 中循环的代码，就可以看到可能的退出方式。

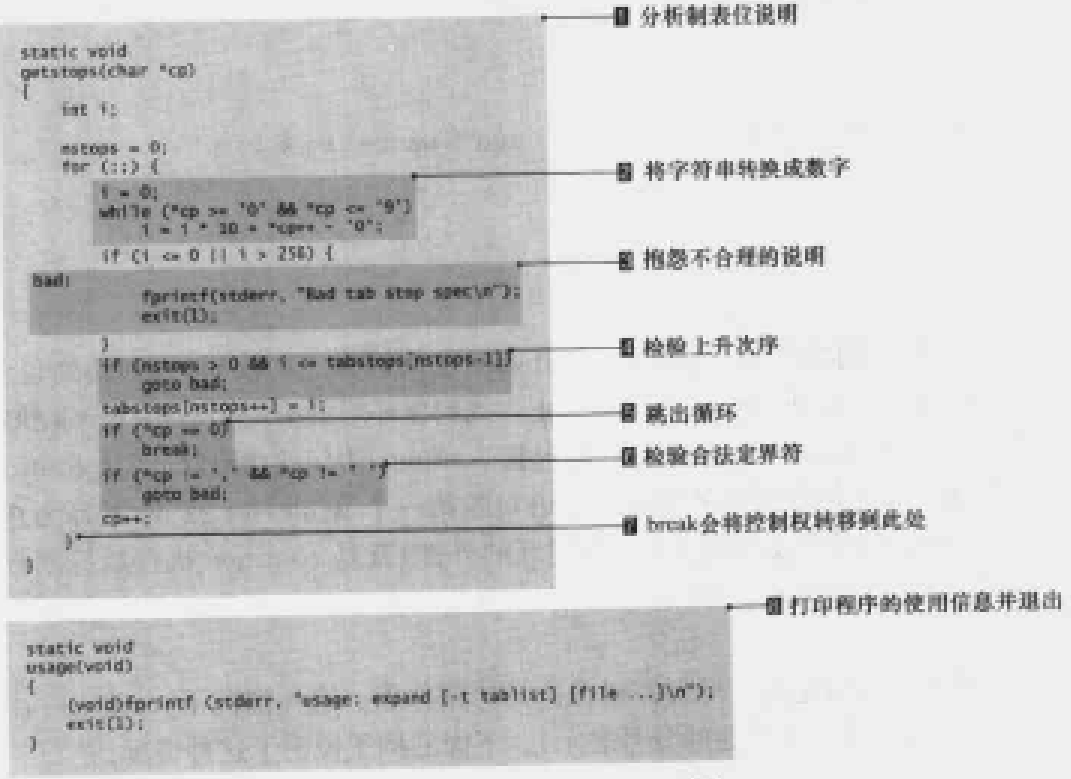


图 2.5　展开制表位(补充函数)

① netbsdsrc/usr.bin/vi/vi/vs_smap.c:389

② netbsdsrc/sbin/init/init.c:540-545

- 错误的制表位说明会引起程序报告一个消息，然后终止(图 2.5:3)。
- 到达制表符说明字符串的结尾，也会退出循环。

练习 2.16 在 C 语言家族中，for 语句十分灵活。从本书提供的源代码中找出十种不同的应用。

练习 2.17 使用 while 替代 for 表达本节中的例子。您觉得哪种形式更易读呢?

练习 2.18 设计一个风格指南，规定在选择循环语句时，什么时候 while 循环优先于 for 循环。对照本书配套盘中的典型例子验证该指南。

2.6 break 和 continue 语句

break 语句将程序转移到最内层的循环或 switch 语句之后执行(图 2.5:7)。大多数情况下，break 用于提前退出循环。continue 语句则跳过该语句到循环末尾之间的语句，继续最内层循环的迭代。continue 语句会再次计算 while 条件表达式的值，并执行循环。在 for 循环中，该语句将首先计算第三表达式的值，之后是条件表达式。continue 用在循环体分开处理不同情况的地方；每种情况一般都以 continue 结束，以便进行下一次循环迭代。在我们分析的程序中，处理完每个不同类型的输入字符后都用到了 continue 语句(图 2.3:8)。

在阅读 Perl 代码时，请注意 break 和 continue 相应地被命名为 last 和 next。①

```
while ( < UD > ) {
    chomp;
    if (s/0x[\d\w]+ \s+ \((.*?)\) // and $ wanted eq $ 1) {
        [...]
        last;
    }
}
```

要确定 break 语句的作用，请从 break 开始向上阅读程序的代码，直到遇到包含 break 语句的第一个 while，for，do，或 switch 块为止。找到循环后的第一个语句；break 执行后，控制权就会转移到这个地点。类似地，在分析包括 continue 语句的代码时，请从 continue 语句开始向上阅读程序，直到遇到包含 continue 语句的第一个 while，for 或 do 循环为止。找到循环的最后一个语句；紧随其后(但不是在循环的外部)就是 continue 执行后控制权将转移到的地点。请注意，continue 忽略 switch 语句，break 和 continue 都不会影响到 if 语句的操作。

有时，循环的执行只是为了得到控制表达式的副作用。这种情况下，continue 往往用作占位符，替代空语句(用一个单独的分号表示)。下面的例子说明了这种情况。②

```
for (; *string && isdigit(*string); string++ )
```

① perl/lib/unicode/mktables. PL:415-425

② netbsdsrc/usr. bin/error/pi. c:174-175

```
continue;
```

Java 程序中，break 和 continue 可以跟上一个标记标识符(label identifier)。后跟冒号的相同标识符，还用来标记一个循环语句。之后，就可以用带标记的 continue 语句跳过嵌套循环的迭代；标记标识出相应的 continue 语句将会跳转到的循环语句。因此，在下面的例子中，continue skip; 语句将跳过最外层 for 语句的迭代①。

```
skip:
    for ( [...] ) {
        if ( ch == limit.charAt(0) ) {
            for (int i = 1 ; i < limlen ; i++ ) {
                if ( [...] )
                    continue skip;
            }
            return ret;
        }
    }
```

类似地，带标记的 break 语句用于从嵌套循环中退出；标记标识出相应的 break 语句的结束位置。某些情况下，即使没有嵌套循环，为了澄清相应的循环语句，也会使用带标记的 break 或 continue 语句②。

```
comp : while(prev < length) {
    [...]
    if (pos >= length || pos == -1) {
        [...]
        break comp;
    }
}
```

练习 2.19　在本书提供的源代码中找出 10 个 break 和 continue 语句。标明对应的语句执行后程序将会转移到何处执行，并解释为什么要使用该语句。不要试图理解代码的所有逻辑；只要基于语句的使用模式提供一个解释。

2.7　字符和布尔型表达式

getstops 函数中，for 循环代码体的开始部分，乍看起来可能显得比较神秘(图 2.5:2)。要理解这段代码，我们需要仔细剖析组成它的表达式。第一个表达式，while 循环中的条件，是将 * cp(cp 指向的字符)与两个字符，'0'和'9'，进行比较。所有的比较结果必须为 true，并且它们都涉及 * cp 与不同的不等运算符和其他表达式的组合。重写比较，将被比较的值放到表达式的中央，安排另外两个值以升序排列，这样的测试经常更易于理解。重写后的表达式为：

① jt4/jasper/src/share/org/apache/jasper/compiler/JspReader.java:472-482

② cocoon/src/scratchpad/src/org/apache/cocoon/treeprocessor/MapStackResolver.java:201-244

```
while ('0' <= *cp && *cp <= '9')
```

之后，这个比较就可以简单地看作是，测试字符 c 是否属于某个区间。

$$0 \leqslant c \leqslant 9$$

请注意，这个测试，假定在底层的字符集中数字字符是以升序连续排列。这对于我们所知的所有字符集来说都是正确的，涉及字母表字符的比较在大量的字符集和地区中可能会产生令人意外的结果。考虑下面这个典型的例子①。

```
if ('a' <= *s && *s <= 'z')
    *s -= ('a' - 'A');
```

这段代码试图通过将发现的小写字母（通过 if 测试判断）都减去字符集中从'a'到'A'的距离，将小写字符转换成大写。当字符集中存在位于区间 a...z 之外的小写字母时、当字符集区间 a...z 包含非小写字符时、当每个小写字母的代码与对应的大写字母之间的距离不固定时，这段代码都不能工作。许多非 ASCII 字符集至少存在这些问题中的一个。

代码块中的下一行（图 2.5:2）也可能显得不易理解。它基于 i 和 *cp 的值，以及两个常量 10 和'0'，修改变量 i，同时递增 cp。变量的名称并不是特别有意义，程序的作者也没有使用宏或常量定义来记录这些常量；我们必须充分利用现有的信息。

我们经常可以将表达式应用在样本数据上，借以了解它的含义。在例子中，我们可以从初始值 i(0) 开始，假定 cp 指向一个包含数字的字符串（例如，24），选择的数字要能够被 expand 接受，这要基于我们对 expand 格式说明的了解。然后，我们就可以创建一个表，表中列出了计算表达式的各个组成部分的过程中，所有变量和部分表达式的值。我们使用 i' 和 $*cp'$ 表示表达式求值后变量的值。

Iteration	i	i * 10	*cp	*cp-'0'	i'	$*cp'$
0	0	0	'2'	2	2	'4'
1	2	20	'4'	4	24	0

表达式 *cp - '0'应用了一个常见的习惯用法：通过从 *cp 中减去'0'的序数值，得到 *cp 指向的数字字符所表示的整数值。基于这个表，我们可以得出：在循环结束后，i 将是 cp 在循环开始时指向的数字字符串的十进制值。

了解 i 的用途之后，现在就可以分析那些检验制表位说明的行。验证 i 是否为合理值的表达式比较直观。它是一个布尔型的表达式，由两个另外的表达式通过逻辑或（||）组合而成。虽然这种表达式可以自然地解读为英语文本（如果 i 小于等于零，或大于 255，则打印一个错误信息），但是，有时候还是会需要将布尔型表达式转换成更易读的形式。例如，如果我们想将表达式转换为我们在前面用到过的区间成员表达式（range membership expression，即验证某个值是否属于某个区间的表达式），就会需要将逻辑或替换为逻辑与（&&）。使用 De Morgan 法则可以容易地完成这项工作②。

① netbsdsrc/games/hack/hack.objnam.c:352-353

② 我们使用<=> 运算符表示两个表达式等价。这并非 C/C++/C#/Java 运算符。

```
! (a || b) <=> ! a && ! b
! (a && b) <=> ! a || ! b
```

从而，我们可以将测试代码转换成下面的形式：

```
i <= 0 || i > 256 <=>
! (! (i <= 0) && ! (i > 256)) <=>
! (i > 0 && i <= 256) <=>
! (0 < i && i <= 256) <=>
¬ (0 < i ≤ 256)
```

从例子中，我们发现，初始表达式和最终表达式的可读性相同；在其他情况下，我们可能会发现 De Morgan 法则为解开复杂的逻辑表达式提供了一种快速、容易的方法。

阅读布尔表达式时，要注意，多数现代语言中，布尔表达式只对需要的部分进行求值。在用 && 运算符（逻辑与）连接起来的表达式序列中，第一个表达式的求值结果如果为 false，则会结束整个表达式的求值，并生成 false 结果。类似地，在用 || 运算符（逻辑或）连接起来的表达式序列中，如果第一个表达式求值为 true，则会终止对整个表达式的求值，产生一个 true 结果。很多表达式都基于这种短路求值（short-circuit evaluation）特性，在阅读时也应该采用同样的方式。在阅读逻辑乘表达式时，总是可以认为正在分析的表达式以左的表达式均为 true；在阅读逻辑和表达式时，类似地，可以认为正在分析的表达式以左的表达式均为 false。例如，下面 if 语句中的表达式仅当所有组成部分都为 true 时，结果才为 true，仅当 t 不是 NULL 时才会计算 t-> type[①]。

```
if (t != NULL && t -> type != TEOF && interactive && really_exit)
        really_exit = 0;
```

相反地，在下面的例子中，只有 argv 不为 NULL 时，才会检查 argv[1]是否为 NULL[②]。

```
if (argv == NULL || argv[1] == NULL || argv[2] == NULL)
    return -1;
```

在这两个例子中，第一项检查用来防止随后对 NULL 指针进行解除引用操作。在检查指定的定界符(i)是否大于前一个(tabstops[nstops -1])时（图 2.5:4），getstops 函数也使用短路求值法。只有在至少已经处理过一项其他定界符描述之后（nstops >0），才会执行该测试。大多数从 C 派生的语言，比如C++，Perl，和 Java，都有短路求值特性；另一方面，Fortran，Pascal，和大多数 Basic 的分支语言总要计算布尔表达式的各个组成部分。

练习 2.20　在本书配套盘中，找出对字符代码值的假定存在问题的表达式。阅读 Java Character 类的测试和转换方法，如：isUpper 和 toLowerCase，或 C 函数中对应的 ctype 系列（isupper，tolower 等）。建议应该如何修改代码，使之较少地依赖于目标构架的字符集。

练习 2.21　在本书的源代码库中查找、化简并推理 5 个重要的布尔表达式。不要花时

① netbsdsrc/bin/ksh/main. c:606-607

② netbsdsrc/lib/libedit/term. c:1212-1213

间了解表达式各部分的含义;要将注意力放在使表达式成为 true 或 false 的条件上。尽可能地标出和使用短路求值特性。

练习 2.22 在源代码库中找出并推理 5 个重要的整数或字符表达式。要力图最小化推理每个表达式所需理解的代码数量。

2.8 goto 语句

抱怨制表位说明不合理的那段代码(图 2.5:3)从一个后跟冒号的单词开始。这是一个标记(label),goto 指令的目标位置。在阅读代码时,遇到标记和 goto 语句应该立即引起足够的重视。它们容易被误用,创造出"意大利面条"式的代码:代码的控制流程难以跟踪和断定。因此,Java 的设计者决定不支持 goto 语句。幸运的是,大多数现代程序仅仅在不对程序的结构造成负面影响的特殊环境中,少量地使用 goto 语句。

执行某些行动后(比如打印一条错误消息,或释放分配的资源),常常用 goto 语句退出程序或函数。在我们的例子中,代码块结尾调用的 exit(1)会终止程序,并向系统外壳返回错误代码(1)。因此,所有通向 bad 标记的 goto 语句只不过是打印错误消息后终止程序的快捷方式。类似地,图 2.6[①] 中的程序清单说明如何将一个公共的错误处理器(图 2.6:4)作为公共退出点,用到所有发生错误的地方(图 2.6:1,图 2.6:2)。函数的正常退出路径位于错误处理器之前(图 2.6:3),从而保证当没有错误发生时,处理器不会被调用。

```
static int
gen_init(void)
{
    [...]
    if ((sigaction(SIGXCPU, &n_hand, &o_hand) < 0) &&
        (o_hand.sa_handler == SIG_IGN) &&
        (sigaction(SIGXCPU, &o_hand, &o_hand) < 0))
        goto out;                                          1 失败,退出并标示发生错误

    n_hand.sa_handler = SIG_IGN;
    if ((sigaction(SIGPIPE, &n_hand, &o_hand) < 0) ||
        (sigaction(SIGXFSZ, &n_hand, &o_hand) < 0))
        goto out;                                          2 失败:退出并标示发生错误

    return(0);                                             3 正常函数退出(成功)

                                                           4 公共错误处理代码
    out:
    syswarn(1, errno, "Unable to set up signal handler");
    return(-1);
}
```

图 2.6 goto 语句在公共错误处理器中的应用

goto 语句还经常用来重新执行某一部分代码,可能是在某些变量的值发生改变,或者执行完某些处理之后。虽然这样的构造经常可以用结构化循环语句(例如,for (;;))结合 break 和 continue 来完成,但实践中,使用 goto 有时能够更好地传达编码者的意图。一个单独的标记——几乎总是命名为 again 或 retry,用作 goto 的目的地。图 2.7 的例子[②]就是一种典型的用法,它在系统数据库中定位一项具体服务,忽略注释以及过长的行。(有趣的是,

① netbsdsrc/bin/pax/pax.c:309-401

② netbsdsrc/lib/libc/net/getservent.c:65-104

这段代码好像有一个 bug。如果读入了不完整的行，它继续读余下的部分，就如同新行一样，因此，如果一个长行的尾部恰好和服务的定义相像，它将会被使用。这类疏忽经常成为计算机安全中利用的目标）。

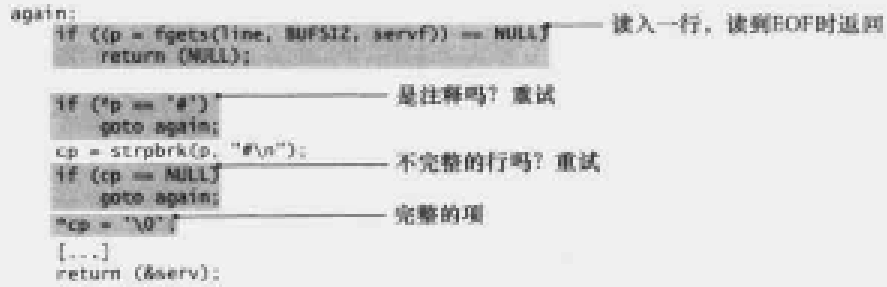


图 2.7　goto 用于重新执行代码

最后，在嵌套循环和 switch 语句中，goto 语句可以用来替代 break 和 continue——这两个语句只影响最内层循环的控制流程，改变程序的控制流程。有时，即使嵌套的层次允许使用 break 和 continue 语句，也会使用 goto 语句。在大型、复杂的循环中使用这种做法可以澄清控制流程的走向，同时避免了向嵌套循环中添加特定 break 或 continue 语句引发错误的可能性。在图 2.8 的例子[①]中，goto have_msg 语句用来替代 break，退出 for 循环。

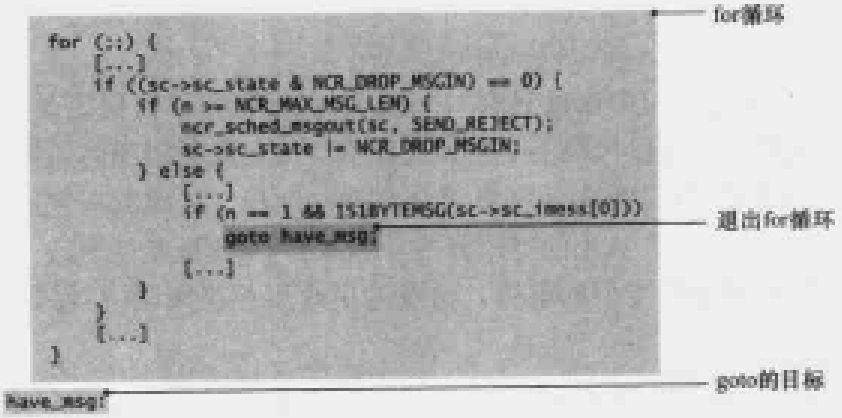


图 2.8　使用 goto 语句退出循环

练习 2.23　在代码库中找到 5 处使用 goto 语句的实例。对它的应用进行分类（针对我们介绍的每种可能的应用，至少找出一个实例），讨论每个特定的 goto 是否能够并且应该替换为循环或其他语句。

练习 2.24　对于大量不同的错误，函数 getstops 都生成相同的错误消息。说明在不使用 goto 语句的情况下，如何使函数的错误报告更用户友好。讨论什么时候对源代码的这种修改是恰当的，什么时候应该避免。

① netbsdsrc/sys/dev/ic/ncr5380sbc.c:1575-1654

2.9 小范围重构

getstops 代码的其余部分相对直观易懂。在核实每个制表位都比前一个大之后(图 2.5:4),制表位的偏移被存储在 tabstops 数组中。将单独的制表位数字转换成整数后(图 2.5:2),cp 将指向字符串中第一个非数字字符(即,循环将处理所有的数字,在第一个非数字字符处停止)。此时,一系列由 if 语句指定的检查控制了程序的操作。如果 cp 指向制表位描述字符串的结尾(值为 0 的字符,标志 C 字符串的结束),那么,循环将会结束(图 2.5:5)。最后一个 if(图 2.5:6)检查不合法的定界符,如果发现,则结束程序的运行(使用 goto bad 语句)。

这些 if 语句中的代码体中都通过 goto 或 break 语句将控制权转移到其他地方。因此,我们也可以将该序列理解为:

```
if (*cp == 0)
    break;
else if (*cp != ',' && *cp != ' ')
    goto bad;
else
    cp++;
```

经过改动,凸显出 3 个语句中永远只有一个被执行的事实,使代码更易于阅读和推理。如果您控制着一段代码(即,这段代码不是由外部提供商或开放源码团体负责支持和维护),重新组织相关的代码段,使它们更易于阅读,会使您受益匪浅。代码编写完成后对设计进行的改进称为重构(refactoring)。从小的更改开始,比如我们刚刚概括的更改——在相关的文献中,可以找到超过 70 种不同类型的重构。适度的更改累积起来,往往暴露出更大规模的改进。

作为进一步的例子,考虑下面这个仅仅只有一行的代码精华①。

```
op = &(!x ? (!y ? upleft : (y == bottom ? lowleft : left)) :
(x == last ? (!y ? upright : (y == bottom ? lowright : right)) :
(!y ? upper : (y == bottom ? lower : normal))))[w -> orientation];
```

该代码过度使用了条件运算符?:。使用条件运算符的表达式就如同 if 代码。例如,要将下面的表达式②:

```
sign ? -n : n
```

读为:

"如果 sign 为 true,那么该表达式的值为-n;否则,表达式的值为 n。"

由于我们将表达式读作 if 语句,我们也可以像 if 语句那样对它进行格式化:将它看作是用 x ? 替代 if (x),圆括号替代花括号,-: 替代 else。重新编排表达式时,我们要用到编辑器

① netbsdsrc/games/worms/worms.c:419

② netbsdsrc/bin/csh/set.c:852

具备的缩进特性，以及显示匹配圆括号的能力，结果参见图 2.9(左侧)。

阅读展开形式的条件表达式肯定要容易，但依旧存在可以提高的地方。在此，我们可以得出：表达式实际上是将控制表达式求值的变量 x 和 y，与 3 种不同的值做测试：

(1) 0(表达为! x 或! y)

(2) bottom 或 last

(3) 所有其他值

```
op = &(
!x ? (
    !y ?
        upleft
    : (
        y == bottom ?
            lowleft
        :
            left
    )
) : (
    x == last ? (
        !y ?
            upright
        : (
            y == bottom ?
                lowright
            :
                right
        )
    ) : (
        !y ?
            upper
        : (
            y == bottom ?
                lower
            :
                normal
        )
    )
))[w->orientation];
```

```
op = &(
!x ? (
    !y ?
        upleft
    : ( y == bottom ?
        lowleft
    :
        left
    )
) : ( x == last ? (
    !y ?
        upright
    : ( y == bottom ?
        lowright
    :
        right
    )
) : (
    !y ?
        upper
    : ( y == bottom ?
        lower
    :
        normal
    )
)
))[w->orientation];
```

图 2.9　条件表达式的类 if 语句编排(左)和类层叠 if-else 语句编排(右)

因此，我们能够重写该表达式，将它作为一系列层叠 if-else 语句进行编排(使用 ?: 运算符来表达)，使得表达式能够展现出我们业已得出的结论。参见图 2.9(右侧)的结果。

这个表达式的意图现在变得十分明确：基于 x 和 y 的组合值从 9 个不同的位置值中选取一个。然而，所有的替代公式，视觉上强调了标识符号，牺牲了语义内容，同时需要用到大量垂直空间。不过，基于对该表达式的已有理解，我们能够创建一个二维数组，其中包含这些位置值，并使用从 x 和 y 导出的偏移对它进行索引。图 2.10 中是新的结果。请注意，在

locations 数组的初始化过程中，我们如何使用二维文本结构来说明所执行计算的二维特性。初始值在程序正文中以二维排列，数组以违反常规的次序[y][x]进行索引，并且是将 x 和 y 变换到整数"0,2,1"，而非更为明显的"0,1,2"，从而使得二维表达与单词 upleft,upper 等的语义含义相一致。

图 2.10 替代条件表达式的位置检测代码

这段代码，约 20 行，比原来的一行要长，但仍然比用层叠 else 形式的表达方式要少 7 行。就我们看来，它看起来更易读，自我注解，且易于检验。有人可能会争辩说原来的版本执行起来比新的快。这种想法来源于代码的可读性与效率在某种程度上不能兼顾的谬论。在此，并不需要为了效率，牺牲代码的易读性。高效的算法和特殊的优化确实有可能使得代码更为复杂，从而更难理解，但这并不意味着使代码更为紧凑和不易读会提高它的效率。在我们的系统和编译器上，代码的最初版本和最终版本执行时间完全相同：0.6μs。即使存在速度上的差别，考虑软件维护开销、程序员薪水和 CPU 性能，进行经济学分析，大多数时候会倾向于代码的可读性，而非效率。

然而，即使图 2.10 中的代码也不尽如人意：它的优点建立在两个明显的缺点之上。首先，它将代码分成两块，虽然在图 2.10 中显示在一起，但在现实的代码中必须分开。其次，它引入了额外的编码(0,1,2)，从而使得理解代码正在做什么需要两个思维步骤，而非一步完成(将"0,last,other"，变换为"0,2,1"，然后将一对"0,2,1"值变换到 9 项中的某一项)。我们能否以某种方式直接将计算中的二维结构引入到条件代码中呢？下面的代码片断[①]又回归到条件表达式，但是对表达式进行了精心的安排，以表达出计算的意图。

```
op =
  &(          !y ? (!x ?  upleft : x != last ?  upper :  upright ) :
    y != bottom ? (!x ?    left : x != last ? normal :    right ) :
                  (!x ? lowleft : x != last ?  lower : lowright )
   )[w -> orientation];
```

① 由 Guy Steele 提出。

有时，创造性的代码布局可以用来提高代码的易读性，上面的表述就是一个基本的例子。请注意，9 个值在它们所在的 3 列中向右对齐，使得它们在视觉上比较突出，并且可以借用它们名称中“left”和“right”的重复。还要注意，！＝回避了在运算符周围放置空格的惯常准则，这是为了将测试表达式缩小为单一可视标记，使得 9 个数据值更为突出。最后，由于整个表达式可以放在 5 个代码行中，使我们更容易将第一个和最后一个圆括号垂直对齐，也更容易看出整个语句的基本结构是：

```
op = &( < conditional- mess> )[w -> orientation];
```

在两种新的替代表达方式中，选择哪一种主要依赖个人口味；然而，如果没有将代码表达成最初的、更为冗长和清晰的形式，我们可能拿不出第二种表述。

我们对其进行重写的表达式极端庞大，明显难以阅读。稍微简单一些的表达式也能够受益于重写。我们常常可以通过添加空格、利用临时变量将表达式分解成较小的部分、或使用圆括号提高特定运算符的优先次序等方法，使得表达式更为易读。

使程序更易读，并不总是需要更改它的结构。一些不会影响程序运行的项（比如注释、空格的使用和变量、函数、类名的选择），常常会影响程序的易读性。考虑我们为了理解函数 getstops 的代码所做的工作。在函数定义前放置一段简洁的注释就会增强程序将来的易读性。

```
/*
  * Parse and verify the tab stop specification pointed to by cp
  * setting the global variables nstops and tabstops[].
  * Exit the program with an error message on bad specification.
  */
```

在阅读您所控制的代码时，要养成添加注释的习惯。

在 2.2 节和 2.3 节中，我们解释了如何利用名称和缩进来理解代码的功能。遗憾的是，有时程序员会选择没有什么帮助的名称，并且胡乱地缩进他们的程序。我们可以用好的缩进以及对变量名称的明智选择，提高编写欠佳程序的易读性。这些措施都很极端：仅当您全权负责和完全控制源代码，并确信您的更改比原来的代码要好得多，而且在发生问题时能够回复到原始的代码时，才能应用这种方法。使用版本管理系统，如修订控制系统（Revision Control System，RCS）、源代码控制系统（Source Code Control System，SCCS）、并发版本系统（Concurrent Versions System，CVS）或 Microsoft Visual SourceSafe 能够帮助您控制代码的修改。为变量名和缩进采用一种具体的风格可能是一项乏味的任务。在修改隶属于大型代码体的代码使之更为易读时，要力图理解并遵照其余代码使用的约定（参见第 7 章）。许多组织都拥有特殊的编码风格，学习并尽量遵循它。如果没有的话，则应该采用一种标准的风格（比如 GNU① 和 BSD② 小组所采用的风格），并始终如一地使用它。当代码的缩进十分混乱，并且不能手动挽救时，许多工具（比如 indent）可以帮助我们自动对代码进行重新缩排，使代码更易读（参见 10.7 节）。在使用这类工具时要小心：明智地使用空格可以为阅读

① *http://www.gnu.org/prep/standards_toc.html*

② netbsdsrc/share/misc/style:1-315

者提供可视化的线索，这是自动格式化工具力所不及的。将 indent 应用到图 2.10 中的代码上，肯定会降低它的易读性。

要注意，虽然重新缩排代码或许有助于提高代码的易读性，它也有可能弄乱程序在修订控制系统中的更改历史。因此，最好不要将重新编排与对程序逻辑的任何实际更改合并起来。应该先对代码进行重新编排，将它检入(check-in)，然后再做其他更改。用这种方式，将来的代码阅读者能够选择性地读取和复查对程序逻辑的更改，不会受到整体重新编排的影响。另一方面，用 diff 程序分析程序的修订历史时，如果这段历史跨越了整体重新缩排，通常可以通过指定-w 选项，让 diff 忽略空白差异，避免由于更改了缩进层次而引入的噪音。

练习 2.25 从您所处的环境，或本书配套盘中找出 5 个可以通过改进代码的结构使之更易读的例子。

练习 2.26 在 International Obfuscated C Code Contest(国际混乱 C 代码竞赛)网站[①]上，可以找到数 10 个有意编写的难以阅读的 C 程序。它们当中的大多数都使用好几个迷惑层来隐藏它们的算法。参考代码的逐次变化，能够帮助您解开他们的代码。如果您不熟悉 C 预处理器，请避开那些拥有大量 # define 行的程序。

练习 2.27 修改我们分析过的位置定位代码，使它能够在镜像上运行(将左侧和右侧进行交换)。计算更改最初的代码，以及更改图 2.10 列出的最终版本所花费的时间(即以不同版本的代码为蓝本，进行修改，比较花费的时间。——译者注)。不要查看易读的表达方式；如果您发现它们比较有用，请从零开始创建它们。基于当前的程序员工资水平(不要忘记加上其他开销)，计算成本的差异。如果易读的代码运行速度只有最初代码的一半(并非如此)，通过合理地假定给定价格的计算机在生命期内执行该代码的次数，计算这种减速的成本。

练习 2.28 如果您尚不熟悉任何具体的编码规范，请找到一个，并采用它。对照编码规范检验本地代码。

2.10 do 循环和整型表达式

最后，我们将注意力转移到 expand 程序中完成处理功能的代码体(图 2.3)，结束对该程序理解。开始部分是一个 do 循环。do 循环的循环体至少执行一次。在本例中，对于余下的每个参数，do 循环的主体都要执行。这些参数可以指定进行制表符展开的文件名。处理文件名参数的代码(图 2.3:6)重新打开 stdin 文件流，来访问每个后续的文件名参数。如果没有指定文件名参数，if 语句的主体(图 2.3:6)不会被执行，expand 将处理标准输入。实际的处理过程是不断读入字符，并跟踪当前的列位置。switch 语句是字符处理的主力，它处理所有以特定方式影响列位置的各种字符。我们不详细分析制表符定位之后的逻辑。容易看

① *http://www.ioccc.org*

出，前 3 个和后 2 个块又可以写成层叠 if-else 序列。我们会将注意力放在代码中的一些表达式上。

一些相等性测试，如用来测试 nstops 的那个测试(例如，nstops == 0)，常常会错误地使用赋值运算符 = ，而非相等性运算符 == 。在 C，C++和 Perl 中，类似下面的语句：①

```
if ((p = q))
    q[-1] = '\n';
```

在 if 语句中使用了一个合法的测试表达式，将 q 赋给 p 并测试结果是否为 0。假如程序员真实的意图是测试 p 是否等于 q，大多数编译器并不能检测出这种错误。在我们分析的语句中，(p = q) 两边的圆括号，可能用来表示程序员确实希望一个赋值操作，并且之后与 0 进行测试。另一种清楚表达这种意图的方式是显式地与 NULL 做测试。②

```
if ((p = strchr(name, '=')) != NULL) {
    p++;
```

这种情况下，这个测试也可以写为 if (p = strchr(name, '=')),但如果写成这样，我们会不知道这是一种有意的赋值，还是一个错误。

最后，可能采用的另一种解决方法是：采用一种风格，所有有常量参与的比较，都将常量写在比较表达式的左边。③

```
if (0 == serconsole)
    serconsinit = 0;
```

采用了这种风格后，对常量的错误赋值会被编译器标记为错误。

阅读 Java 或 C#程序时，很少会遇到这类错误，这是因为这些语言在相应的流程语句中只接受布尔型的值作为控制表达式。实际上，本书配套盘上的 Java 代码中，找不出单一的可疑语句。

表达式 column & 7，用于控制循环处理代码中第一个 do 循环，也很有意思。& 运算符执行两个操作数之间的逐位与(bitwise-add)操作。在这里，我们不是为了处理二进制位，而是屏蔽掉 column 变量的一些最高有效位，它返回 column 除以 8 后的余数。执行算术运算时，当 $b = 2^n-1$ 时，可以将 a & b 理解为 a % (b+1)。这样书写表达式是为了将除法替换为逐位与指令(有时计算起来更高效)。实际上，现代的优化编译器能够识别出这类情况，独立地完成替换，而且，除法和逐位与指令在现代处理器上运行时，速度上的差别已经不像过去那么大。因此，我们应该学会阅读使用这些技巧的代码，但要避免使用它。

还有两种常见的情况，会用位指令来替代算术指令。它们是移位(shift)运算符<< 和 >> ——将整数的位向左或向右移动。由于整数每个二进制位的值均为 2 的幂，移位一个整数就等同于用 2 的幂去乘或除该整数，幂的次数就是移位的数目。因此，您可以从算术意义来思考移位运算符，如下所示。

---

① netbsdsrc/bin/ksh/history.c:313-314

② netbsdsrc/bin/sh/var.c:507-508

③ netbsdsrc/sys/arch/amiga/dev/ser.c:227-228

- 将 a<< n 理解为 a * k,$k = 2^n$。下面的示例用移位运算符来完成乘以 4 的运算[①]。

```
n = ((dp - cp)) << 2) - 1; /*  4 times + NULL */
```

- 将 a>> n 理解为 a / k,$k = 2^n$。下面的示例来自于一个二分查找(binary search)例程,它使用向右移位运算符来完成除以 2 的运算[②]。

```
bp = bp1 + ((bp2 - bp1) >> 1);
```

切记,Java 的逻辑右移位运算符>> 不能用来对有符号数执行除法运算,因为当用于负数时,它会产生错误结果。

练习 2.29 大多数编译器都提供一种功能,让用户能够查看编译后的汇编语言代码。找出在您的操作环境中,编译 C 程序时,如何生成汇编代码,并检查编译器为算术表达式和相应使用位指令的表达式生成的代码。试着使用不同的编译器优化级别。评论这两种方式的易读性和代码效率。

练习 2.30 什么样的参数会引起 expand 失败?在什么情况下会给出这种参数?提出一种简单的修复方法。

2.11 再论控制结构

分析了控制流程的语句的语法细节之后,现在,我们可以将注意力集中在,如何在抽象层对它们进行推导。

第一件应该记住的事情就是:每次只分析一个控制结构,将它的内容看作是一个黑盒。结构化编程的优点就是:它采用的控制结构,允许抽象和选择性地推理程序的各个组成部分,而不必考虑程序总体的复杂性。

考虑下面的代码序列[③]。

```
while (enum.hasMoreElements()) {
  [...]
  if (object instanceof Resource) {
      [...]
      if (!copy(is, os))
          [...]
  } else if (object instanceof InputStream) {
      [...]
      if (!copy((InputStream) object, os))
          [...]
  } else if (object instanceof DirContext) {
```

① netbsdsrc/bin/csh/str.c:460

② netbsdsrc/bin/csh/func.c:106

③ jt4/catalina/src/share/org/apache/catalina/loader/StandardLoader.java:886-905

```
            [...]
        }
    }
```

虽然我们移除了这 20 行代码中一大部分代码，这个循环依旧显得十分复杂。然而，应该采用下面的方式来阅读上面的循环：

```
while (enum.hasMoreElements()) {
    // Do something
}
```

从而，在抽象层面，您就能够将注意力集中在循环体上（而非具体的代码），并对循环体的功能进行分析，无须关心循环体之内的控制结构。这种思想道出了分析程序的控制流程时，应该遵循的第二条规则：将每个控制结构的控制表达式看作是它所包含代码的断言。虽然上面的代码可能有些愚钝或无足轻重，但是它对代码理解却是意义重大。再考虑一下我们分析的 while 语句。对这个控制结构的典型解读是：当 enum. hasMoreElements()为 true 时，循环中的代码会执行。但是，在分析循环体内的代码时（像上面我们建议的那样，以隔离开来的方式进行），总可以认为 enum. hasMoreElements()为 true，因此，包含在循环体中的语句

```
NameClassPair ncPair = (NameClassPair) enum.nextElement();
```

在执行时不会发生任何问题。同样的推理过程也适用于 if 语句。在下面的代码中，我们能够确信，当 links. add 执行时，links 集合中不会包含 next 元素①。

```
if (! links.contains(next)) {
    links.add(next);
}
```

遗憾的是，一些控制语句污染了我们在上面勾画出的美好画面。return，goto，break 和 coutinue 语句，还有异常，都会影响结构化的执行流程。由于这些语句一般都会终止或重新开始正在进行的循环，因此要单独推理它们的行为。在此，假定 goto 语句的目标是循环体的开始或结尾，即将 goto 语句用作多级 break 或 continue。如果不是这种情况，所有的假设都不成立。

在浏览循环代码时，我们总是想确保代码能够在所有的环境中都按照它的规格说明执行。多数情况下，只需要简单地论证一下就够了，但有时需要使用更为严格的方法（即后面将提到的，用不变式和变式来证明循环的方法）。

考虑二分查找算法。要正确地实现这个算法十分困难。它的应用首次出现在 1946 年，Knuth[Knu98]对此做了详细的描述，但直到 1962 年，没人发表能够在大小不是 2^n-1 的数组上正确工作的算法。Bentley[Ben86]补充说，当他要求大批专业程序员作为练习实现它时，只有 10%是正确的。

图 2. 11 列出了标准 C 语言库中对二分查找算法的实现②。从中，我们可以看出，它通

① cocoon/src/java/org/apache/cocoon/Main. java:574-576

② netbsdsrc/lib/libc/stdlib/bsearch. c

过逐渐减小存储在 lim 变量中的查找区间，并调整查找区间的起始位置(存储在 base 中)来完成工作，但是从中无法得知该算法的计算方法是否能够在任何环境下都正确工作。如果您觉得难以推理这段代码，阅读在它之前的这段说明或许有所帮助。

下面的代码有些隐蔽。在比较失败后，我们通过向左或向右移动查找区间，将工作划分为原来的一半。如果 lim 是奇数，向左移动查找区间只需简单地对分 lim；例如，当 lim 是 5，正在检查的项是第 2 项时，我们将 lim 改为 2，以便对第 0 项和第 1 项进行检查。如果 lim 是偶数，也同样适用。如果 lim 是奇数，向右移动查找区间同样会对分 lim，这次将 base 移动到 p 后面的一项；例如，当 lim 是 5 时，我们将 base 改为指向第 3 项，将 lim 设为 2，从而我们将检查第 3 项和第 4 项。但是，如果 lim 是偶数，我们必须在对分前将其减小 1；例如，当 lim 是 4 时，我们首先检查的项依旧是第 2 项，我们必须将 lim 改为 3，然后对分，获得结果 1，从而，我们将只检查第 3 项。

如果您，和我一样，觉得上面的注释还不够明白或依旧不能打消您的疑虑，可以考虑采用更为复杂的方法来验证它的正确性。

```
void *
bsearch(key, base0, nmemb, size, compar)
    register const void *key;                    查找项
    const void *base0;                           元素数组的开始
    size_t nmemb;                                元素的数目
    register size_t size;                        每个元素的大小
    register int (*compar) __P((const void *, const void *));   用来比较两个元素的函数
{
    register const char *base = base0;
    register int lim, cmp;
    register const void *p;

    for (lim = nmemb; lim != 0; lim >>= 1) {
        p = base + (lim >> 1) * size;            在中间定位一个指针
        cmp = (*compar)(key, p);                 将元素与键进行比较
        if (cmp == 0)                            找到；返回它的位置
            return ((void *)p);
        if (cmp > 0) {    /* key > p: move right */
            base = (char *)p + size;             向上调整base
            lim--;                               不确定为什么需要这样做
        } /* else move left */
    }
    return (NULL);                               没有找到
}
```

图 2.11 二分查找的实现

对于推导循环的属性，一种有用的抽象是基于变式(variant)和不变式(invariant)的概念。循环不变式是程序状态的一个断言，在循环的开始和结尾都成立。如果能够证明特定的循环维护了不变式，并且在循环结束时，选定的不定式可以用来表明期望的结果业已获得，那么，我们就能够确信，算法的循环始终工作在正确算法结果的包络之内。但是，仅仅确定这种情况是不够的。我们还需要确保循环能够终止。为了做到这一点，我们要使用变式(variant)，一种表示与最终目标距离的量度，每次循环迭代都要递减它。如果我们能够演示出一个循环的操作在维护不变式的同时递减变式，我们就能确定，循环能够正确终止。

我们从一个简单的例子开始。下面的代码查找 depths 数组中的最大值。①

```
max = depths[n];
```

① XFree86-3.3/xc/lib/Xt/GCManager.c:252-256

```
while (n -- ) {
  if (depths[n] > max)
    max = depths[n];
}
```

如果我们将 n_0 定义为 depths 数组中元素的个数(最初保存在变量 n 中),我们可以将循环结束时我们希望的结果形式上表示为:

$$\text{max} = \text{maximum}(\text{depths}[0:n_0))$$

我们使用符号$[a:b)$来表示一段区间,该区间包括 a 但在 b 的前一个元素结束(即不包括 b,数学上称为闭半开区间,或半开区间),即,$[a:b\text{-}1]$。那么合适的不变式可以是

$$\text{max} = \text{maximum}(\text{depths}[n:n_0))$$

不变式在第一次向 max 赋值时建立,所以,在循环开始时,它是成立的。n 递减后,它不可避免地会有可能不成立,由于区间 $[n:n_0)$ 包含索引为 n 的元素,这个元素有可能会比 max 保持的最大值要大。在 if 语句执行后,不变式被重新建立,这是因为,如果区间扩展后新增的成员确实大于我们此前的最大值,if 语句会调整 max。这样,我们已经展示出不变式在每次循环迭代结束时都成立,故而,在循环终止时,它也成立。由于循环在 n(我们可以将其认为是循环的变式)到达 0 时会终止,那时,我们的不变式就可以重写为我们想要满足的最初规格,从而,说明循环确实达到了我们希望的结果。

可以将这种推理方式应用到二分查找示例上。图 2.12 例举出了同样的算法,但稍微做了一些重新编排,以便简化不变式的推理。

- 我们将右移位运算符替换为除法;
- 由于 size 变量只是用来在勿需知道指针类型的情况下,模拟指针算法,所以我们去除了该变量;
- 我们将 for 语句的最后一个表达式移到循环的结尾,以澄清循环内部操作的次序。

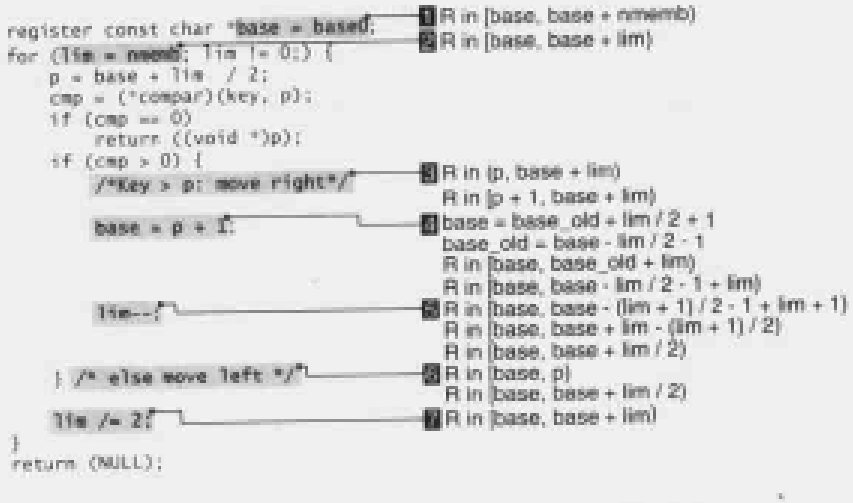


图 2.12　维护二分查找的不变式

如果正在寻找的值位于一个特定区间,那么适合使用不变式。我们使用标记 $R\in[a:b)$ 表示查找的结果在数组元素 a(包括 a)和 b(b 除外)之间。由于循环中使用 base 和 lim 限定

查找区间，所以我们的不变式就是 R∈[base；base+lim)。我们将会证实这个不变式在每次循环迭代之后都成立，从而说明 bsearch 函数确会找出数组中的这个值——如果存在的话。由于调用比较函数 compar 时总会传入一个来自于不变式区间(base +lim/2) 之内的参数，同时，还由于 lim(我们的变式)每次循环迭代都会对分，所以，我们能够确定，如果存在这样的值，compar 最终会找出它。

在 bsearch 函数的开始处，我们只能断言函数的规格：R∈[base0；base0+nmemb)。但是，在图 2.12；1 之后，它可以表达为 R∈[base；base+nmemb)，在 for 语句赋值(图 2.12；2)后，可以表达为 R∈[base；base+lim)——这就是我们的不变式。从而，在循环的开始处我们的不变式成立。

如果我们正在查找的值比 p 处的值大，则 compar 函数的结果为正。因此，在图 2.12；3 处，我们可以说

$$R\in(\mathrm{p};\mathrm{base}+\mathrm{lim})\equiv$$
$$R\in[\mathrm{p}+1;\mathrm{base}+\mathrm{lim})$$

如果我们将最初的 base 值表示为 base_{old}，那么，经过 2.12；4 处的赋值后，我们最初的不变式现在成为：

$$R\in(\mathrm{base};\mathrm{base}_{old}+\mathrm{lim})$$

由于 p 被赋予的值是 base+lim/2，因此得到：

$$\mathrm{base}_{old}+\frac{\mathrm{lim}}{2}+1\Leftrightarrow$$

$$\mathrm{base}_{old}=\mathrm{base}-\frac{\mathrm{lim}}{2}-1$$

通过将上面的结果代入我们已有的不变式中，我们得到：

$$R\in\left[\mathrm{base};\mathrm{base}-\frac{\mathrm{lim}}{2}-1+\mathrm{lim}\right).$$

当 lim 在图 2.12；5 处减 1 时，我们将 lim+1 代入我们的不变式中，得到：

$$R\in\left[\mathrm{base};\mathrm{base}-\frac{\mathrm{lim}+1}{2}-1+\mathrm{lim}+1\right)\equiv$$
$$R\in\left[\mathrm{base};\mathrm{base}+\mathrm{lim}-\frac{\mathrm{lim}+1}{2}\right)\equiv$$
$$R\in\left[\mathrm{base};\mathrm{base}+\frac{\mathrm{lim}}{2}\right)$$

通过类似的过程，在 compar 函数的结果是负数的情况下，表示我们正在查找的值比 p 处的值小，我们可以得到：

$$R\in[\mathrm{base};\mathrm{p})\equiv$$
$$R\in\left[\mathrm{base};\mathrm{base}+\frac{\mathrm{lim}}{2}\right)$$

请注意，现在针对不同的比较结果，得出的不变式均相同。此外，lim 在图 2.12；7 处对分之后，我们可以将它的新值代入到不变式中，得到 R∈[base；base+lim)，即我们在循环顶部时的不变式。因此，我们展示出该循环维护了这个不变式，从而能够正确地从数组中找出正在寻找的值。最后，当 lim 变为 0 时，寻找的值可能位于的区间为空，因此，返回 NULL 是

正确的，表示值没有找到。

练习 2.31　在本书配套盘中找出 5 个超过 50 行的控制结构，用一行注释说明其功能。

练习 2.32　从上面找出的控制结构中选取一个，推理其代码体，标出您将控制表达式作为断言的地方。

练习 2.33　插入排序(insertion sort)函数[①]是本书配套盘中基数排序(radix sort)实现的一部分，请证明它能够正确地运行。提示：最内层的 for 循环只比较两个元素；swap 函数只在它们没有正确排序的情况下才执行。

进　阶　读　物

Kernighan 和 Plauger[KP78]，以及最近的 Kernighan 和 Pike[KP99，第 1 章]提供了许多改进代码风格的建议；在阅读一些编写混乱的代码时，可以用这些建议理清它们。除了 2.9 节中提及的具体风格表以外，另外还有一个编写清晰代码的风格指南，那就是 Indian Hill C 风格和编码规范(*Indian Hill C Style and Coding Standard*)；将它的标题输入到 Web 搜索引擎中，可以轻松地在网络上找到它。有关编程风格的全面书目，参见 Thomas 和 Oman[TO90]。Dijkstra[Dij68]介绍 goto 语句相关问题的文章现已成为经典之作。程序缩进对理解的影响在 Miara 等人的著作[MMNS83]中做了研究。Oman 和 Cook[OC90]研究了编排(也称格式化、安排)和注释的影响。有关注释和过程如何影响程序的易读性，参见 Tenny[Ten88]。重构作为提高代码设计(和易读性)的活动，在 Fowler[Fow00，第 56～77 页]进行了介绍。如果希望看到语言的设计者如何介绍一门语言，请阅读 Kernighan 和 Ritchie[KR88](介绍 C 语言)，Stroustrup[Str97](C++)，Microsoft 公司[Mic01](C#)，以及 Wall 等人所著的[WCSP00](Perl)。另外，Ritchie[Rit79]提供了 C 及其函数库的深度介绍，Linden[Lin94]清楚地解释了许多 C 语言中的出色之处。

不变式是由 C. A. R. Hoare[Hoa71]引入的。在[Ben86，第 36～37 页；Mey88，第 140～143 页；Knu97，第 17 页；HT00，第 116 页]也可以找到相关的介绍。二分查找算法的完整分析，在 Knuth[Knu98]中给出。

① netbsdsrc/lib/libc/stdlib/radixsort.c:310-330

第3章　高级C数据类型

人们在设计完全傻瓜式的系统时，常犯的错误就是低估了十足傻瓜的能力。

—— Douglas Adams

指针、结构、共用体(union)、动态内存和类型名称声明是构成更复杂C数据类型和算法的基本元素。虽然，到目前为止，我们已经和它们有过广泛的接触，但由于它们的应用十分多样化、专门化，并且拥有自己独特的习惯用法，故而值得深入探讨。本章中，我们将分析指针、结构和共用体在程序中的典型应用，并为每个应用模式提供相应的例子。了解特定语言构造所服务的功能之后，就能够更好地理解使用该语言的代码。

在程序执行期间，动态地分配和处理内存区域，建立新数据存储单元的能力，是一种十分有用的重要概念，它经常与指针和结构联合使用，创建和操纵新的数据类型。掌握了几种管理动态内存的常见用法后，对它们的了解就会更为明晰。

3.1　指　　针

大多数学习C语言的人，在开始时都对指针又敬又怕。确实，基于指针的代码有时难以理解，并且常常会导致难以捉摸的错误和没有任何征兆的崩溃。主要的原因是，指针是一个低级、强大，但却较难掌握的工具。因此，学会分析它的应用和常见编码模式就显得比较紧迫；仅仅查看那些一成不变的少数代码实例很难有所提高。

在C程序中，指针一般用来：

- 构造链式数据结构
- 引用动态分配的数据结构
- 实现引用调用(call by reference)
- 访问和迭代数据元素
- 传递数组参数
- 引用函数
- 作为其他值的别名
- 代表字符串
- 直接访问系统内存

接下来，我们将依次分析每种应用中具有代表性的一些例子。在适当的地方，我们将指出详细论述特定应用的章节。下面的段落对基于指针的代码进行概括和分类，并提供了阅读这类代码的路线图。

3.1.1　链式数据结构

在最底层，指针就是一个内存地址。因此，它特别适合于表示数据结构中各个元素之间

的链接。在概念层，数据结构，诸如链表、树和图等，都是由顶点(vertex)或边(edge)结合起来的元素(结点)构成的。通过在一个结点中存储另外结点的地址，指针可以表示两个结点间的有向顶点(directed vertex)。此外，指针还常常用在链式数据结构的处理中，比如访问它们的元素、执行内务操作等。我们在 4.7 节和其后的章节中分析链式数据结构。

3.1.2　数据结构的动态分配

向量类型(vector-type)的数据结构经常动态地分配，以使其大小符合运行时对元素数量的需求。我们在 3.4 节中对此进行详细分析。指针用来存储这类数据结构的起始地址。此外，程序经常动态地构造 C 结构，将它们在函数间传递或使用它们来构建链式数据结构。这种情况下，指针用来存储分配给 struct 的内存单元，如下面的例子所示：①

```
static struct diskentry *
finddisk(const char * name)
{
    struct diskentry * d;
    [...]
    d = emalloc(sizeof(* d));
    d -> d_name = estrdup(name);
    d -> d_name[len] = '\0';
    [...]
    return d;
}
```

这段代码分配 struct diskentry 大小的内存并将结果转换成恰当的指针，这项任务经常交由一个宏来完成，宏的名称可以是 new，位于C++相等运算符之后。②③

```
# define new(type) (type * ) calloc(sizeof(type), 1)
      [...]
      node = new(struct codeword_entry);
```

3.1.3　引用调用

指针还用在以引用传递(passed by reference)的方式接收参数的函数中。以引用传递的参数可以用来返回函数的结果，或者避免参数复制带来的开销。在用指针类型的参数返回函数的结果时，我们会看到，函数体内会对这些变量赋值的语句，例如，下面的函数将 gid 设为给定组名的组标识符。④

```
int
gid_name(char * name, gid_t * gid)
{
```

① netbsdsrc/sbin/fsck/preen.c:250-280

② XFree86-3.3/xc/doc/specs/PEX5/PEX5.1/SI/xref.h:113

③ XFree86-3.3/xc/doc/specs/PEX5/PEX5.1/SI/xref.c:268

④ netbsdsrc/bin/pax/cache.c:430-490

```
    [...]
    *gid = ptr -> gid = gr -> gr_gid;
    return(0);
}
```

该函数的返回值只用来表明出错的情况。Windows API 中的许多函数使用这样的约定。在调用者一方，一般通过对变量应用取址运算符(address-of operator，&)，将相应的参数传递给函数。

指针类型的参数，还可以用来避免每次函数调用时复制大型元素所带来的开销。这类参数一般是结构，少数情况下，也可能是双精度浮点数。数组总是以引用传递；大部分构架中，对于其他的基本 C 类型，在函数调用时直接复制要比以引用传递更为高效。下面的函数使用了两个以引用传递的结构参数(now 和 then)，函数只是用它们来计算结果，不需要修改结构的内容。该函数编写为这样的方式大概就是为了消除 timeval 结构在每次调用时进行复制的开销。①

```
static double
diffsec(struct timeval *now,
    struct timeval *then)
{
    return ((now -> tv_sec - then -> tv_sec)*1.0
        + (now -> tv_usec - then -> tv_usec)/1000000.0);
}
```

现代 C 和C++程序中，常常能够确定出为了提高效率而以引用传递的参数，因为它们一般都用 const 声明符来修饰②。(既然用 const 修饰，说明该参数不是用于返回结果，所以可以确定，以引用传递仅仅是为了避免参数复制的开销。——译者注)

```
static char *ccval (const struct cchar * , int);
```

程序中，有时会用注释 IN 或 OUT 将每个参数的作用标示出来，下面的函数定义(pre-ANSI C)就用到了这种方法。③

```
int
atmresolve(rt, m, dst, desten)
    register struct rtentry *rt;
    struct mbuf *m;
    register struct sockaddr *dst;
    register struct atm_pseudohdr *desten; /*  OUT */
{
```

3.1.4 数据元素的访问

我们将在 4.2 节介绍如何使用指针作为数据库游标来访问表中的元素。在阅读使用指

① netbsdsrc/sbin/ping/ping.c:1028-1034

② netbsdsrc/bin/stty/print.c:56

③ netbsdsrc/sys/netinet/if_atm.c:223-231

针对数组进行访问的代码时，需要掌握的关键概念是用指向数组元素地址的指针，可以访问位于特定索引位置的元素；对于指向数组元素的指针和相应的数组索引，二者的算法具有相同的语义。表 3.1 展示出访问数组 a(元素类型为 T)时，最常用到的操作。图 3.1[①] 提供一个使用指针对栈进行访问的例子。

表 3.1　分别使用索引和指针访问数组 a(元素类型为 T)的代码

数组索引代码	指针代码
int i;	T *p;
i = 0;	p = a 或 p = &a[0]
a[i];	*p
a[i].f	p -> f
i++	p++
i += K	p += K
i == N	p == &a[N] 或 p == a + N

```
stackp = de_stack;          ——— 初始化栈
[...]
*stackp++ = finchar;        ——— 将finchar压入栈中
[...]
do {
    if (count-- == 0)
        return (num);
    *bp++ = *--stackp;      ——— 从栈弹出到*bp中
} while (stackp > de_stack);——— 检查栈是否为空
```

图 3.1　使用指针访问基于数组的栈

3.1.5　数组型的参数和结果

在 C 和C++程序中，将数组传递到函数和作为结果返回，都要用到指针。在 C 代码中，将数组名作为函数的参数时，实际上传递给函数的是数组第一个元素的地址。因而，函数执行时对数组中的数据做出的任何修改，都将影响到数组中的元素。数组这种隐式的引用调用行为，与 C 语言中所有其他数据类型的传递方式都不相同(所有其他类型，如 char，int，float，甚至结构和共用体，都是值传递。——译者注)，因此，可能容易造成混淆。

类似地，C 函数只能返回指向数组元素的指针，不能返回整个数组。因此，当函数在数组中生成结果，并返回相应的指针时，一定要确保该数组不是在函数的栈上分配的局部变量。如果数组是局部变量，在函数退出后，数组的存储空间可能被重写，从而函数的结果也就变为无效。避免出现这类问题的一种方法是将这类数组声明为 static，下面的函数即使用这种方法，该函数将 Internet 地址转换成圆点分隔的十进制数表示。[②]

```
char * inet_ntoa(struct in_addr ad)
```

① netbsdsrc/usr.bin/compress/zopen.c:523-555

② netbsdsrc/libexec/identd/identd.c:120-137

```
{
    unsigned long int s_ad;
    int a, b, c, d;
    static char addr[20];
    s_ad = ad.s_addr;
    d = s_ad % 256;
    s_ad /= 256;
    c = s_ad % 256;
    s_ad /= 256;
    b = s_ad % 256;
    a = s_ad / 256;
    sprintf(addr, "%d.%d.%d.%d", a, b, c, d);
    return addr;
}
```

该函数在缓冲区 addr 中生成最终的结果。如果没有将该缓冲区声明为 static，那么它的内容(Internet 地址的可读表示法)在程序返回后将变为无效。即使上面这种做法也并非完全安全。使用全局或 static 局部变量的函数大多数情况都不可重入(reentrant)。这意味着，如果这个函数的一个实例正在运行时，那么程序中其他执行线程就不能再调用它。更坏的情况是，在我们的示例中，函数的结果必须在该函数再次被调用之前，保存到其他地方(例如，使用 strdup 调用)；否则，它将被新的结果重写。例如，下面的代码中，使用 naddr_ntoa 的地方就不能使用我们列出的函数 inet_ntoa 的实现。①

```
(void)fprintf(ftrace, "%s Router Ad"
        " from %s to %s via %s life = %d\n",
        act, naddr_ntoa(from), naddr_ntoa(to),
        ifp ? ifp -> int_name : "?",
        ntohs(p -> ad.icmp_ad_life));
```

在这种情况下，为了避免我们描述的问题，函数 naddr_ntoa 用作 inet_ntoa 的包装函数，它将 inet_ntoa 的结果存储到由 4 个不同临时缓冲区组成的循环列表中。②

```
char *
naddr_ntoa(naddr a)
{
# define NUM_BUFS 4
    static int bufno;
    static struct {
        char str[16]; /*  xxx.xxx.xxx.xxx\0 */
    } bufs[NUM_BUFS];
    char *s;
    struct in_addr addr;
```

① netbsdsrc/sbin/routed/rdisc.c:121-125

② netbsdsrc/sbin/routed/trace.c:123-139

```
    addr.s_addr = a;
    s = strcpy(bufs[bufno].str, inet_ntoa(addr));
    bufno = (bufno + 1) % NUM_BUFS;
    return s;
}
```

3.1.6 函数指针

将函数参数化，即把函数作为参数传递给另外的函数，常常很有用。但是，C语言不允许将函数作为参数传递给其他函数。然而，它允许传递指向函数的指针。图3.2[①]中，函数getfile的功能是在备份/恢复过程中对文件进行处理，它接收两个参数，fill和skip，这两个参数均为指向函数的指针；它们用来标明如何读取或跳过数据。代码中使用不同的参数(图3.2:3)调用该函数(图3.2:1)，对数据进行扫描，恢复或跳过数据文件。

许多C库文件，比如qsort和bsearch，都接受函数参数，来指定它们应该如何运作。[②]

```
getnfile()
{
    [...]
    qsort(nl, nname, sizeof(nltype), valcmp);
    [...]
}

valcmp(nltype *p1, *p2)
{
    if ( p1 -> value < p2 -> value ) {
        return LESSTHAN;
    }
    if ( p1 -> value > p2 -> value ) {
        return GREATERTHAN;
    }
    return EQUALTO;
}
```

在上面的例子中，qsort用valcmp函数比较排序数组中的元素。

最后，函数指针可以用来参数化代码体内的控制。在下面的例子中，closefunc用来保存关闭fin流时需要调用的函数，实际调用的函数依赖于fin流打开方式的不同。[③]

```
void
retrieve(char * cmd, char * name)
{
    int (* closefunc) (FILE * ) = NULL;
    [...]
```

① netbsdsrc/sbin/restore/tape.c:177-837

② netbsdsrc/usr.bin/gprof/gprof.c:216-536

③ netbsdsrc/libexec/ftpd/ftpd.c:792-860

```
/*
 * Verify that the tape drive can be accessed and
 * that it actually is a dump tape.
 */
void
setup()
{
    [...]
    getfile(xtrmap, xtrmapskip);
    [...]
}

/* Prompt user to load a new dump volume. */
void
getvol(int nextvol)
{
    [...]
    getfile(xtrlnkfile, xtrlnkskip);
    [...]
    getfile(xtrfile, xtrskip);
    [...]
}

/*
 * skip over a file on the tape
 */
void
skipfile()
{
    curfile.action = SKIP;
    getfile(xtrnull, xtrnull);
}

/* Extract a file from the tape. */
void
getfile(void (*fill)(char *, long), (*skip)(char *, long))
{
    [...]
    (*fill)((char *)buf, (long)(size > TP_BSIZE ?
        fssize : (curblk - 1) * TP_BSIZE + size));
    [...]
    (*skip)(clearedbuf, (long)(size > TP_BSIZE ?
        TP_BSIZE : size));
    [...]
    (*fill)((char *)buf, (long)((curblk * TP_BSIZE) + size));
    [...]
}

/* Write out the next block of a file. */
static void
xtrfile(char *buf, long size)
{ [...] }
/* Skip over a hole in a file. */
static void
xtrskip(char *buf, long size)
{ [...] }
/* Collect the next block of a symbolic link. */
static void
xtrlnkfile(char *buf, long size)
{ [...] }
/* Skip over a hole in a symbolic link */
static void
xtrlnkskip(char *buf, long size)
{ [...] }
/* Collect the next block of a bit map. */
static void
xtrmap(char *buf, long size)
{ [...] }
/* Skip over a hole in a bit map */
static void
xtrmapskip(char *buf, long size)
{ [...] }
/* Noop, when an extraction function is not needed. */
void
xtrnull(char *buf, long size)
{ return; }
```

1 用函数作为参数进行调用

2 调用函数参数

3

4

5 作为参数传递的函数

图 3.2 使用函数参数的参数化

```
if (cmd == 0) {
    fin = fopen(name, "r"), closefunc = fclose;
[...]
if (cmd) {
    [...]
```

```
        fin = ftpd_popen(line, "r", 1), closefunc = ftpd_pclose;
      [...]
    (*closefunc)(fin);
}
```

3.1.7 用作别名的指针

指针经常用于创建特定值的别名。①

```
struct output output = {NULL, 0, NULL, OUTBUFSIZ, 1, 0};
[...]
struct output *out1 = &output;
```

上面的代码中,任何原来使用变量 output 的地方,都可以使用 *out1。我们会因为许多不同的原因使用别名。

1. 效率上的考虑

指针的赋值要比对较大对象的赋值更有效率。在下面的例子中,也可以不将 curt 定义成指向结构的指针,而直接定义成结构变量;但是,相应的赋值语句将会效率低下,因为它将不得不复制整个结构的内容。②

```
static struct termios cbreakt, rawt, *curt;
[...]
curt = useraw ? &rawt : &cbreakt;
```

2. 引用静态初始化的数据

用一个变量指向不同的静态数据。最常见的情况是指向不同字符串的字符指针。③

```
char *s;
[...]
s = *(opt -> bval) ? "True" : "False";
```

3. 在全局语境中实现变量引用语义

这个唬人的标题所说的用法,与引用调用(call-by-reference)的指针用法等效,只不过使用的是全局变量,而非函数参数。从而,在其他地方,可以使用全局的指针变量引用要访问和修改的数据。下面的随机数生成器中,程序首先将 fptr 初始为指向随机数种子表中的一个入口;在 srrandom()中,也用类似的方式对它进行了设定。最后,在 rrandom()中,使用变量 fptr 修改它指向的值。④

```
static long rntb[32] = {
    3, 0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
    [...]
};
```

① netbsdsrc/bin/sh/output.c:81-84

② netbsdsrc/lib/libcurses/tty.c:66-171

③ netbsdsrc/games/rogue/room.c:629-635

④ netbsdsrc/games/rogue/random.c:62-109

```
static long * fptr = &rnth[4];
[...]

void
srrandom(int x)
{
    [...]
    fptr = &state[rand_sep];
    [...]
}

long
rrandom()
{
    [...]
    * fptr += * rptr;
    i = (* fptr >> 1) & 0x7fffffff;
```

也可以将 fptr 作为参数传递到 srrandom()和 rrandom()中，达到同样的目的。

类似的方式还常常用来修改全局数据。①

```
WINDOW scr_buf;
WINDOW * curscr = &scr_buf;
[...]
move(short row, short col)
{
    curscr -> _cury = row;
    curscr -> _curx = col;
    screen_dirty = 1;
}
```

3.1.8 指针和字符串

在 C 语言中，字符串常量用以字符 '\0' 结尾的字符数组来表示。因而，字符串由指向 null 结尾序列中第一个字符的指针来表示。从下面的代码中，可以看到，C 库函数 strlen 的代码在参数字符串中持续向前移动一个指针，直到指针到达串的结尾为止。之后，它从指向结尾 null 字符的指针中，减去指向字符串起始字符的指针，得出并返回字符串的长度。②

```
size_t
strlen(const char * str)
{
    register const char * s;
```

① netbsdsrc/games/rogue/curses. c:121-157

② netbsdsrc/lib/libc/string/strlen. c:51-59

```
    for (s = str; *s; ++s)
        ;
    return(s - str);
}
```

在阅读基于字符串的代码时，要注意区分字符指针和字符数组。虽然这两者都经常用来表示字符串(因为将字符数组传递给函数时，它被自动转换为指向数组第一个字符的指针)，但底层类型和可以对它们执行的操作是不同的。例如，下面的代码片段将 pw_file 定义为字符指针，它指向包含 "/etc/passwd" 的字符串常量。①

```
static char * pw_file = "/etc/passwd";
```

在我的机器上，pw_file 变量的大小为 4，可以将它改为指向其他的地方，但更改它指向的内存会产生不可预知的行为。

另一方面，下面的代码将 line 定义为字符数组，并将它初始化为 null 结尾的序列"/dev/XtyXX"。②

```
static char line[] = "/dev/XtyXX";
```

对 line 应用 sizeof 运算符返回 11；line 总是指向同一存储区域，可以自由地修改它所包含的元素。③

```
line[5] = 'p';
```

在阅读代码时，要始终牢记字符数组与字符指针之间的区别，因为它们都可以用于相同的目的，经常可以互换。重要的是，当这两者在不同的文件间意外混淆时，我们所知道的所有 C 编译器都不会给予警告。考虑下面的两种(不相关)定义和相关的声明。④⑤⑥

```
char version[] = "332";
char * version = "2.1.2";
extern char * version;
```

两个定义都是定义一个版本字符串，接下来可能会将它传递给 printf 函数打印出来。但是，extern char * version 声明只能用来访问定义为 char * version 的变量。虽然将源文件与包含 char version[]定义的文件链接起来一般不会产生错误，但生成的程序将会在运行期间出错。此时的 version 变量，并非指向包含 version 字符串的内存区域，而是可能指向字符串"322"(某些处理器构架下是 0x33333200)的位模式(bit pattern)所表示的地址。

3.1.9　直接内存访问

由于指针在计算机内部以内存地址来表示，所以，低级代码常常使用指针访问与硬件专

① netbsdsrc/distrib/utils/libhack/getpwent.c:45
② netbsdsrc/lib/libutil/pty.c:71
③ netbsdsrc/lib/libutil/pty.c:81
④ netbsdsrc/usr.bin/less/less/version.c:575
⑤ netbsdsrc/libexec/identd/version.c:3
⑥ netbsdsrc/libexec/identd/identd.c:77

有的内存区域。许多外设，比如图形和网络适配器，使用一段公共可访问的内存区域与系统的主处理器交换数据。将一个指针变量初始化为指向该区域后，就可以方便地使用这个变量与给定的设备进行通信。下面的例子将变量 video 设为指向屏幕适配器(screen adapter，或称显示适配器)占用的内存区域(0xe08b8000)。通过向该区域中由指定的行和列(vid_ypos 和 vid_xpos)确定的偏移地址执行写入操作，可以使对应的字符(c)显示在屏幕上。①

```
static void
vid_wrchar(char c)
{
    volatile unsigned short * video;

    video = (unsigned short * )(0xe08b8000) +
          vid_ypos * 80 + vid_xpos;
    * video = (* video & 0xff00) | 0x0f00 | (unsigned short)c;
}
```

要注意，除非经过特定的安排，否则现代操作系统会阻止用户程序直接访问硬件。只有在分析嵌入式系统的代码，或者内核和设备驱动程序时，才有可能遇到这类代码。

练习 3.1 使用数组索引，重新实现 compress 库②中基于指针的栈。测量两种实现在速度上的差异，并评论它们的易读性。

练习 3.2 在本书配套盘中，针对我们概括的使用指针的每个理由，找出 3 个代码实例。

练习 3.3 如果您熟悉C++或 Java，请解释如何尽量减少(C++)或避免(Java)指针的使用。

练习 3.4 C指针与内存地址有何不同？这会如何影响您对代码的理解？哪个工具利用了这种差别？

练习 3.5 程序的版本字符串应该表示为字符数组呢？还是指向字符串的指针？验证您的答案！

3.2 结　　构

C语言中的结构将多个数据元素集合在一起，使其可以作为一个整体来使用。结构在C程序中的应用如下：

- 将一般作为一个整体来使用的数据元素集合到一起；

① netbsdsrc/sys/arch/pica/pica/machdep.c:951-958

② netbsdsrc/usr.bin/compress/zopen.c

- 从函数中返回多个数据元素；
- 构造链式数据结构(参见 4.7 节)；
- 映射数据在硬件设备、网络链接和存储介质上的组织方式；
- 实现抽象数据类型(9.3.5 节)；
- 以面向对象的方式编程。

下面的段落将介绍在其他小节中没有涉及到的结构应用。

3.2.1　聚合数据元素

结构经常用来将一般作为整体使用的互相关联的元素聚合在一起。最典型的例子是屏幕上某个位置坐标的表达。①

```
struct point {
    int col, line;
};
```

其他可能的情况包括复数和数据表中组成行的各个字段。

3.2.2　从函数中返回多个数据元素

在单个基本数据类型不足以表达函数结果的情况下，返回多个数据的方式或是通过那些以引用的方式传递给函数的参数(参见 3.1.3 节)，或是将它们聚合成一个结构，然后返回这个结构。下面的例子中，difftv 以 timeval 结构的形式，返回两个时间之间的差，以秒(tv_sec)和微秒(tv_usec)表示。②

```
static struct timeval
difftv(struct timeval a , struct timeval b)
{
    static struct timeval diff;

    diff.tv_sec = b.tv_sec - a.tv_sec;
    if ((diff.tv_usec = b.tv_usec - a.tv_usec) < 0) {
        diff.tv_sec -- ;
        diff.tv_usec += 1000000;
    }
    return(diff);
}
```

3.2.3　映射数据的组织方式

当数据在网络上移动或传入/传出辅助存储介质时或当程序直接与硬件进行交互时，结构经常用来表示数据在其他介质上的组织方式。下面的结构表示 Intel EtherExpress 网卡

① netbsdsrc/games/snake/snake/snake.c:75-77

② netbsdsrc/usr.sbin/named/dig/dig.c:1221-1233

的一个命令块。①

```
struct fxp_cb_nop {
    void *fill[2];
    volatile u_int16_t cb_status;
    volatile u_int16_t cb_command;
    volatile u_int32_t link_addr;
}
```

volatile 限定符(qualifier)用来标明,底层的内存字段要被程序之外的实体使用(在这里,是网卡)。从而,禁止编译器对这些字段执行优化,比如移除冗余引用。

某些情况下,程序中可能会声明位字段(bit field),用来指定一段精确的位范围,保存给定设备上的特定值。②

```
struct fxp_cb_config {
    [...]
    volatile u_int8_t byte_count:6,
             :2;
    volatile u_int8_t rx_fifo_limit:4,
             tx_fifo_limit:3,
             :1;
```

上面的例子中,指定传输字节数(byte_count)占据 6 个二进制位,而接收者和传送者的 FIFO 队列限制分别在硬件设备上占据 4 个二进制位和 3 个二进制位。

网络数据包在编码时,也经常使用 C 结构来描绘它们的组成元素,下面列出的 TCP 包头的经典定义就说明了这种用法③。

```
struct tcphdr {
    u_int16_t th_sport;     /* source port */
    u_int16_t th_dport;     /* destination port */
    tcp_seq th_seq;     /* sequence number */
    tcp_seq th_ack;     /* acknowledgement number */
    [...]
```

最后,结构还用于映射数据在外设介质上(比如磁盘和磁带)的存储方式。例如,MS-DOS 磁盘分区的磁盘特性由所谓的 BIOS 参数块(BIOS parameter block)来决定。它的各个字段使用下面的结构来指定。④

```
struct bpb33 {
  u_int16_t bpbBytesPerSec; /* bytes per sector */
  u_int8_t bpbSecPerClust;  /* sectors per cluster */
  u_int16_t bpbResSectors;  /* number of reserved sectors */
```

① netbsdsrc/sys/dev/pci/if_fxpreg.h:102-107
② netbsdsrc/sys/dev/pci/if_fxpreg.h:116-125
③ netbsdsrc/sys/netinet/tcp.h:43-47
④ netbsdsrc/sys/msdosfs/bpb.h:22-34

```
    u_int8_t bpbFATs;          /*  number of FATs */
    u_int16_t bpbRootDirEnts; /*  number of root directory entries */
    u_int16_t bpbSectors;      /*  total number of sectors */
    u_int8_t bpbMedia;         /*  media descriptor */
    u_int16_t bpbFATsecs;      /*  number of sectors per FAT */
    u_int16_t bpbSecPerTrack; /*  sectors per track */
    u_int16_t bpbHeads;        /*  number of heads */
    u_int16_t bpbHiddenSecs;  /*  number of hidden sectors */
};
```

结构内字段的次序依赖于所处的构架和所用的编译器。此外，结构中各种元素的表达也依赖于构架和操作系统(操作系统可能会强制特定的处理器采用某种字节次序)。即使是简单数据类型，比如整型，字节的存储方式也会有所不同。因此，用结构来映射外部数据天生不可移植。

3.2.4　以面向对象的方式编程

C 程序中，经常将数据元素和函数指针聚合成结构，模拟类的字段和方法，创建与类相仿的实体。下面的例子中，domain 结构用来表示网络协议的不同领域(例如，Internet，SNA，IPX)，它聚合了特定领域的数据(比如它所属的家族 dom_family)和操作这些数据的方法(比如路由表初始化方法 dom_rtattach)。①

```
struct domain {
    int     dom_family;             /*  AF_xxx */
    char    *dom_name;
    void    (*dom_init)(void);      /*  initialize domain data structures */
    [...]
    int     (*dom_rtattach)(void **, int);
                                    /*  initialize routing table */
    int     dom_rtoffset;           /*  an arg to rtattach, in bits */
    int     dom_maxrtkey;           /*  for routing layer */
};
```

按照这样的声明，属于这种结构类型的变量，或指向这种结构类型的指针(更为常见)，能够像C++或 Java 对象一样使用(在恰当初始化之后)。②

```
for (dom = domains; dom; dom = dom -> dom_next)
    if (dom -> domfamily == i && dom -> dom_rtattach) {
        dom -> dom_rtattach((void **)&nep -> ne_rtable[i],
            dom -> dom_rtoffset);
        break;
```

由于“对象”可以用不同的“方法”(函数指针)来初始化，并通过相同的界面来使用(通过同一结构成员名调用)，我们概括的这个技术实现了面向对象语言中的“虚方法(virtual

① netbsdsrc/sys/sys/domain.h:50-65

② netbsdsrc/sys/kern/vfs_subr.c:1436-1440

method)"和"多态编程(polymorphic programming)"。实际上,由于同一类别的不同对象共享它们的方法(不是它们的字段),所以,程序中常常在"对象"结构中只存储指向另外结构的指针,由这个结构给出指向实际方法的指针,从而达到不同对象间共享方法指针的目的。①

```
struct file {
  [...]
  short f_type;          /*  descriptor type */
  short f_count;         /*  reference count */
  short f_msgcount;      /*  references from message queue */
  struct ucred * f_cred; /*  credentials associated with descriptor */
  struct fileops {
    int   (* fo_read) (struct file * fp, struct uio * uio,
                       struct ucred * cred);
    int   (* fo_write) (struct file * fp, struct uio * uio,
                       struct ucred * cred);
    int   (* fo_ioctl) (struct file * fp, u_long com,
                       caddr_t data, struct proc * p);
    int   (* fo_poll) (struct file * fp, int events,
                       struct proc * p);
    int   (* fo_close) (struct file * fp, struct proc * p);
  } * f_ops;
  off_t f_offset;
  caddr_t f_data;        /*  vnode or socket */
};
```

上面的例子中,每个对象表示一个打开的文件或插口(socket),它们通过指向共享 fileops 结构的结构成员 f_ops,共享 read,write,ioctl,poll,和 close 方法。

练习 3.6 要想通过值传递,将数组类型的数据传递给函数,并作为实际的值来返回,一种可选的方式是将数组嵌入到结构之中。在本书配套盘中找出这类例子。请说明为什么不会经常使用这种方法。

练习 3.7 在本书配套盘中找出 20 个单独的结构实例,并归类使用它们的原因。了解一种通用的数据结构库,比如C++ STL 或 java.util,标出哪些实例可以使用库提供的功能进行编码。尽量减少花在每个实例上的时间。

练习 3.8 许多技术、库和工具支持数据的可移植编码,能够使数据在应用程序间进行传输。概括适用于您的操作环境的技术,将它们与基于 C 结构的特殊方案进行比较。

3.3 共 用 体

C 共用体(union)将共享同一存储区域的项聚合起来。某一时刻,共享该区域的这些项

① netbsdsrc/sys/sys/file.h:51-73

中，只有一项可以访问。在C程序中共用体主要用于下述用途：

- 有效地利用存储空间
- 实现多态(polymorphism)
- 使用不同的内部表达方式对数据进行访问

3.3.1 有效地利用存储空间

使用共用体通常是将同一存储区域用作两种不同的目的；这样做是节省几个字节的珍贵内存。有时使用共用体仅仅是为了节省内存。但是，在这样一个即使嵌入式设备都装备几兆内存的行业中，这种用途变得越来越稀少。除了遗留代码之外，有时大量对象要靠共用体来调整编写适当代码所需的附加工作。下面这个例子(取自标准C函数库中的内存分配函数malloc)，就很典型。①

```
union overhead {
    union overhead * ov_next;    /*  when free * /
    struct {
        u_char ovu_magic;        /*  magic number * /
        u_char ovu_index;        /*  bucket #  * /
# ifdef RCHECK
        u_short ovu_rmagic;      /*  range magic number * /
        u_long ovu_size;         /*  actual block size * /
# endif
    } ovu;
# define ov_magic  ovu.ovu_magic
# define ov_index  ovu.ovu_index
# define ov_rmagic ovu.ovu_rmagic
# define ov_size   ovu.ovu_size
};
```

内存块要么空闲，要么被占用。每种情况下存储的值是不同的，因为一个内存块不可能在同一时间又空闲、又被占用，所以它们可以共享同一内存空间。为了维护这种方案所付出的额外编程开销，会分摊到所有调用C函数库生成的、数以万亿计的项上。

请注意，宏定义跟在ovu结构的定义之后；这类定义，经常用作直接引用共用体内结构成员的快捷方式，无须预先考虑共用体的成员名。因此，在代码中会看到：②

```
op -> ov_index = bucket;
```

而非更为冗长的op ->ovu.ovu_index。

3.3.2 实现多态

到目前为止，共用体最常见的应用是实现多态。此处的意思就是同一对象(一般用C结构来表示)用于代表不同的类型。这些不同类型的数据都存储在独立的共用体成员中。存

① netbsdsrc/lib/libc/stdlib/malloc.c:78-92

② netbsdsrc/lib/libc/stdlib/malloc.c:213

储在共用体中的多态数据(polymorphic data)比其他内存管理方式节省空间;但是,这种情况中,使用共用体是为了让一个对象存储不同类型的数据,而非显示我们的节约。这种用途的共用体一般会嵌入到包含 type 字段的 C 结构中,这个 type 字段表示共用体中存储的数据类型。这个字段经常用一个枚举类型来表示,它的名称经常是 type。下面的例子是 Sun RPC(Remote Procedure Call,远程过程调用)库的一部分,其中有用来表示 RPC 消息的结构。它支持两种不同的消息类型:调用或应答。msg_type 枚举变量能够区分这两种类型,共用体中包含了每种类型的数据元素的结构。①

```
enum msg_type {
    CALL= 0,
    REPLY= 1
};
[...]
struct rpc_msg {
    u_int32_t       rm_xid;
    enum msg_type       rm_direction;
    union {
        struct call_body RM_cmb;
        struct reply_body RM_rmb;
    } ru;
};
```

3.3.3 访问不同的内部表达

我们最后分析的共用体应用是将数据存入共用体的一个字段,然后访问另外的字段,以完成数据在不同内部表达方式之间的转换。尽管这种应用天生不能移植,但有些转换是能够安全执行的,其他一些情况虽然比较有用,但要依赖具体机器。tar 文件归档程序就使用下面定义的结构来表示档案中每个文件的信息。②

```
union record {
    char        charptr[RECORDSIZE];
    struct      header {
        char    name[NAMSIZ];
        char    mode[8];
        char    uid[8];
        char    gid[8];
        char    size[12];
        char    mtime[12];
        char    chksum[8];
        char    linkflag;
        char    linkname[NAMSIZ];
        char    magic[8];
```

① netbsdsrc/include/rpc/rpc_msg.h:54-158

② netbsdsrc/usr.bin/file/tar.h:36-54

```
            char    uname[TUNMLEN];
            char    gname[TGNMLEN];
            char    devmajor[8];
            char    devminor[8];
        } header;
};
```

为了检测数据是否受到破坏，校验和字段 chksum 保存组成文件的所有记录字节的和（包括 header）。程序使用共用体成员 charptr 逐字节地对头部数据进行迭代，计算校验和；并使用 header 的成员访问具体的头部字段。由于 C 语言中整型（包括字符）所能表达的任何位模式都是合法有效的，所以将其他 C 类型的内部表达（指针、浮点数、其他整数类型）作为整型来访问一定是合法操作。但是，相反的操作（从整型表达生成其他类型）不是总能得出正确的值。在大多数构架中，由非整数值的原先的副本生成非整型，是一种安全的操作。

其次，结构有一种天生不可移植的应用，那就是访问以某种其他格式存储的构架专有的数据元素。这对基于数据类型的表达方式对它进行解析，或从它的表达方式创建一个数据类型来说都很有用。在图 3.3[①] 的示例中，共用体 u 被用来访问浮点数 v 的内部表示（尾数、指数和符号）。该转换用来将浮点数拆分成标准化的尾数和 2 的整数幂。

```
double
frexp(double value, int *eptr)
{
    union {
        double v;
        struct {
            u_int u_mant2 : 32;
            u_int u_mant1 : 20;
            u_int   u_exp : 11;
            u_int  u_sign :  1;
        } s;
    } u;
    if (value) {
        u.v = value;
        *eptr = u.s.u_exp - 1022;
        u.s.u_exp = 1022;
        return(u.v);
    } else {
        *eptr = 0;
        return((double)0);
    }
}
```

返回的指数
将值保存在这个字段中
从这个字段访问(双精度型的数值)内部表达
尾数
指数
保存值
取出并设置有符号指数
将指数清零
返回标准化的尾数

图 3.3　使用共用体访问内部类型（double）的表达

练习 3.9　在本书配套盘中找出 20 个单独的应用共用体的实例，并归类使用共用体的理由。尽量减少花在每个实例上的时间。创建一个图表，描述因为特定理由而使用共用体的频率。

练习 3.10　提出可移植的替代方案，来实现结构和共用体的不可移植构造。从实现成本、可维护性和效率几个方面分析您的提议。

① netbsdsrc/lib/libc/arch/i386/gen/frexp.c:48-72

3.4 动态内存分配

程序编写时大小未知的数据结构，或程序运行时大小会增长的结构，都存储在程序运行期间动态分配的内存中。程序使用指针来引用动态分配的内存。这一节中，我们将分析如何动态地对向量结构分配内存；然而，我们列出的代码序列，与那些用于存储其他数据结构的代码十分相似，甚至完全相同。

图 3.4[①] 是一个典型的例子，它展示如何动态地分配和使用数据空间。本例中，代码分配内存空间，以数组的形式存储一系列的整数。故而，程序中首先将 RRlen 定义为指向这类整数的指针（图 3.4:1）。指针类型的变量必须正确初始化，使它们指向合法的内存区域。本例中，程序使用 C 库函数 malloc 从系统中获取足够存储 c 个整数的内存区域的地址。malloc 的参数是需要分配的字节数。它的计算很典型：分配的元素数量（c）乘以每个元素的大小（sizeof(int)）。如果系统内存业已耗尽，malloc 函数返回 NULL；如果想要编写健壮的程序，则应该总是检查此类情况（图 3.4:2）。从此处开始，程序就能够存取 RRlen 的值（使用[]或 * 运算符），就如同它是由 c 个元素组成的数组一样。但是，需要注意，这两者并不等同：对数组变量显式地应用 sizeof 将返回数组的大小（例如，10 个双字整数组成的数组，返回 40），而相同的运算符应用到一个指针时，将仅仅返回内存中存储指针所需的存储空间（现代的许多构架中返回 4）。最后，当分配的存储空间不再需要时，必须调用 free 函数释放它。释放之后，再试图存取指针指向的内容将会导致不可预料的结果。没有释放该内存也是一种错误，可能导致程序出现内存泄露，逐渐占用和浪费系统的内存资源。

```
int
update_msg(uchar *msg, int *msglen, int Vlist[], int c)
{
    [...]
    int *RRlen;                                  1 指向整数的指针
    [...]
    RRlen = (int *)malloc((unsigned)c * sizeof(int));   元素的数量 / 每个元素的大小
    if (!RRlen)                                  2 处理内存枯竭
        panic(errno, "malloc(RRlen)");
    [...]
    for (i = 0; i < c; i++) {                    迭代所有的元素
        [...]

        RRlen[i] = dn_skipname(cp, msg + *msglen);   将RRlen用作数组
        [...]
    }
    [...]
    free((char *)RRlen);                         Free allocated memory
    return (n);
}
```

图 3.4 动态存储空间的分配

当内存耗尽时，程序能够做的事情并不多；大多数情况下，打印一个错误消息然后退出是惟一可行的策略。因为这个原因，同时也是为了避免每次调用都检查 malloc 返回值的负担，一般将 malloc 封装到执行这种检查的函数中（一般命名为 xmalloc），在需要调用 malloc

① netbsdsrc/usr.sbin/named/named/ns_validate.c:871-1231

的地方用这个函数来替代，如下例所示。①

```
void *
xmalloc(u_int size)
{
    void *p;

    if ((p = malloc(size)) == NULL)
      err(FATAL, "%s", strerror(errno));
    return (p);
}
    [...]
    oe = xmalloc(s);
    (void)regerror(errcode, preg, oe, s);
```

有些情况下，数组的大小在元素实际存储到其中时才能确定。处理输入数据时经常是这种情况：数据存储在数组中，但需要存储的元素的数目可能在所有的元素被读入之前是未知的。图 3.5② 说明了这种情况。在存储元素之前，将当前的数组索引与分配的内存块中元素的数目做核查。如果需要更多的空间，则调用 C 库函数 realloc，将第一个参数所指向的空间，调整为由函数的第二个参数指定的新大小。该函数返回一个指向调整后内存块的指针，它的地址可能和最初块的地址并不相同。原内存块中的内容被复制到新的位置。请记住，任何指向原来内存块中某个位置的指针变量，现在指向的都是未定义的数据。图 3.5 中的例子线性地增加内存块的大小，每次分配的内存用尽就再分配 16 字节。分配的内存空间呈指数增长也十分常见。③

```
void
remember_rup_data(char *host, struct statstime *st)
{
        if (rup_data_idx >= rup_data_max) {
                rup_data_max += 16;
                rup_data = realloc (rup_data,
                rup_data_max * sizeof(struct rup_data));
                if (rup_data == NULL) {
                        err (1, "realloc");
                }
        }
    rup_data[rup_data_idx].host = strdup(host);
    rup_data[rup_data_idx].statstime = *st;
    rup_data_idx++;
}
```

索引大于分配的大小吗？
新大小
调整分配的空间
保存数据
新索引

图 3.5　内存分配的再调整

```
if (cur_pwtab_num + 1 > max_pwtab_num) {
  /*  need more space in table */
  max_pwtab_num *= 2;
  pwtab = (uid2home_t * ) xrealloc(pwtab,
                          sizeof(uid2home_t) *  max_pwtab_num);
```

① netbsdsrc/usr.bin/sed/misc.c:63-107
② netbsdsrc/usr.bin/rup/rup.c:146-164
③ netbsdsrc/usr.sbin/amd/hlfsd/homedir.c:521-525

3.4.1 管理空闲内存

早些时候，我们说过，如果不将内存块释放掉，会导致程序出现内存泄露。但是，有些程序常常会只分配内存，但从不释放。[①] 当程序的运行时间比较短时，可以忽略内存释放的问题；当程序终止后，它所分配的所有内存都由操作系统自动回收。skeyinit 程序的实现即为这种情况。[②]

```
int
main(int argc, char *argv[])
{
    [...]
    skey.val = (char *)malloc(16 + 1);
    [... no call to free(skey.val) ]
    exit(1);
}
```

skeyinit 程序，在 Bellcore 的 S/Key 身份验证系统中，用来更改密码或添加用户。该程序执行工作后立即退出，从而也就释放了它所分配的内存。然而，同样的代码用到生命期长一些的程序中时（例如，作为路由软件系统的一部分），这种欠考虑的编码实践可能会产生问题。这种情况下，程序就会泄露内存，我们常常可以使用进程查看命令查明是否发生内存泄露，比如 Unix 下的 ps 或 top，Windows 下的任务管理器。请注意，在上面的例子中，skeyinit 可以简单地将 skey. val 指向一个作为局部变量在栈上分配的大小固定的数组。

```
main(int argc, char * argv[])
{
    char valspace[16 + 1];
    [...]
    skey.val = valspace;
}
```

初学 C/C++的程序员对指针的一个常见误解是：所有的指针必须初始化为由 malloc 分配的内存块。虽然以这种风格编写的代码并非错误，但生成的程序经常较难阅读与维护。

少数情况下，程序中会包含垃圾回收器（garbage collector），自动回收不再使用的存储空间。如果分配的内存块，由于某种原因——比如不同的变量共享同一内存块，难以跟踪它们的整个生命期时，常会用到这种技术。这种情况下，每个内存块都会拥有相关联的引用计数（reference count）。每次创建对该内存块的新引用时，都递增引用计数。[③]

```
req.ctx = ctx;
req.event.time = time
ctx -> ref_count ++ ;
```

① netbsdsrc/bin/mv/mv. c:260

② netbsdsrc/usr. bin/skeyinit/skeyinit. c:34-233

③ XFree86-3. 3/xc/lib/Xt/Selection. c:1540-1542

每当一个引用销毁时，递减引用计数。①

```
XtFree((char* (req);
ctx -> req = NULL;
ctx -> ref_count -- ;
```

引用计数到达 0 时，表示该内存块不再被使用，可以释放。②

```
if (-- ctx -> ref_count == 0 && ctx -> free_when_done)
    XtFree((char* )ctx);
```

另一种较少见的方法是使用保守垃圾回收器(conservative garbage collector)，它扫描进程的所有内存，寻找与现有分配内存块相匹配的地址。将扫描中没有遇到的所有内存块都释放掉。

最后，某些版本的 C 函数库实现了一个非标准函数，名为 alloca。这个函数使用与 malloc 相同的接口分配一个内存块，但却不是在程序的堆(heap)上(程序的通用内存区域)分配块，而是在程序的栈上(用来存储函数的返回地址和局部变量的内存区域)分配。③

```
int
ofisa_intr_get(int phandle, struct ofisa_intr_desc *descp,
               int ndescs)
{
    char *buf, *bp;
    [...]
    buf = alloca(i);
```

由 alloca 返回的内存块，在分配它的函数返回时会被自动回收；不需要调用 free 来处理分配的块。自然地，alloca 所分配的内存，永远不能传回给函数(内存在其中分配)的调用者，因为函数返回后，它就不再有效。在一些编程环境中，如 FreeBSD，禁止使用 alloca，因为这种用法被认为是不可移植且依赖于机器；在其他文化中，如 GNU，则鼓励这种用法，因为它减少了偶然的内存泄露。

3.4.2　含有动态分配数组的结构

有时，程序中会使用动态分配的单个结构，存储一些相关的字段和一个数组，数组中存储的是结构专有的变长数据。如果在结构中定义一个元素，使之指向变长数据，那么在使用这个结构时要用到指针间接寻址(pointer indirection)和更多的内存，而这种结构可以避免这些开销。故而，类似下面的定义。④

```
typedef struct {
    XID     id_base;
    [...]
```

① XFree86-3.3/xc/lib/Xt/Selection.c:744-746

② XFree86-3.3/xc/lib/Xt/Selection.c:563-564

③ netbsdsrc/sys/dev/ofisa/ofisa.c:563-564

④ XFree86-3.3/xc/include/extensions/record.h:99-108

```
    unsigned char     *data;
    unsigned long     data_len;     /*  in 4- byte units */
} XRecordInterceptData;
```

在程序中一般不易见到，取而代之的是类似下面的定义：①

```
typedef struct {
    char *user;
    char *group;
    char *flags;
    char data[1];
} NAMES;
```

作为结构元素的 data 数组，用作实际数据的占位符。分配了用来存储结构的内存之后，它的大小会根据 data 数组中元素的个数向上调整。此后，就可以使用数组中的元素，如同数组包含这些元素的空间一样。②

```
if ((np = malloc(sizeof(NAMES) +
    ulen + glen + flen + 3)) == NULL)
      err(1, "%s", "");
np -> user = &np -> data[0];
(void)strcpy(np -> user, user);
```

请注意，上面的示例分配的内存要比实际所需多一个字节。内存块的大小是结构的大小和相关数据元素的大小——3 个字符串的大小(ulen,glen,flen)和它们对应的 null 结束符(3)——一之和。然而，在计算内存块的大小时，没有考虑到结构的大小中已经包括一个占位字节。内存存储结构的微管理是一个危险活动，时刻与错误相伴。

练习 3.11 在本书配套盘中找出一个中等大小、依赖于动态内存分配的 C 程序。估量一下程序的代码中，用于管理动态分配的内存结构的代码，要占到全部代码的百分比。假定内存处理由垃圾回收器自动进行管理，如同 Java 程序中那样，再次估计百分比。

练习 3.12 大多数现代编程环境都提供特殊的库、编译选项或其他工具来检测内存泄露。找出在您的工作环境中可以使用的设施，并在 3 个不同的程序上使用它。说明得到的结果。

3.5 typedef 声明

前一节的好几个例子都使用 typedef 声明来创建新的数据类型名。typedef 声明为一个已有的类型增加一个新的名称，或称替代名。因此，在下面的声明③之后

① netbsdsrc/bin/ls/ls.h:69-74

② netbsdsrc/bin/ls/ls.c:470-475

③ netbsdsrc/libexec/telnetd/defs.h:124

```
typedef unsigned char cc_t;
```

每当看到标识符 cc_t，应该将其理解为 unsigned char。C 程序使用 typedef 声明促进抽象，并增强代码的易读性，从而防范可移植问题，并模拟C++和 Java 的类声明行为。

在 C 声明中组合使用前缀和后缀类型，有时会使得一些 typedef 声明难以阅读①。

```
typedef char ut_line_t[UT_LINESIZE];
```

然而，解读此类声明并不十分困难。只需要将 typedef 看作是类似 extern 或 static 的存储类型描述符，将该声明理解为一个变量定义。

```
static char ut_line_t[UT_LINESIZE];
```

变量的名称(在上面的示例中是 ut_line_t)就是类型的名称；变量的类型就是与该名称对应的类型。

将 typedef 声明用作一种抽象机制时，实际上就是为原来的具体实现定义一个抽象名称作为替代名。因而，由于 typedef 声明的名称是按照恰当命名的抽象概念制订的，不是偶然的实现细节，所以使用这种名称的代码会更容易理解。在下面的例子中，DBT 定义了一个数据库实体——包含一个键和一个数据元素的结构②。

```
typedef struct {
    void *data;             /* data */
    size_t size;            /* data length */
} DBT;
```

依照上面的声明，所有的数据库访问例程都指定在 DBT(以及其他由 typedef 定义的类型)实体上进行操作③。

```
int   __rec_get (const DB * , const DBT * , DBT * , u_int);
int   __rec_iput(BTREE * , recno_t, const DBT * , u_int);
int   __rec_put (const DB *dbp, DBT * , const DBT * , u_int);
int   __rec_ret (BTREE * , EPG * , recno_t, DBT * , DBT * );
```

由于在 C 和C++中，语言中各种数据类型的硬件细节依赖于底层构架、编译器和操作系统，所以 typedef 声明经常用来通过下面两种方式来增强程序的可移植性：(1) 用一系列依赖于具体实现的声明，为已知的硬件量值创建可移植的名称④。

```
typedef __signed char        int8_t;
typedef unsigned char      u_int8_t;
typedef short               int16_t;
typedef unsigned short    u_int16_t;
typedef int                 int32_t;
typedef unsigned int      u_int32_t;
```

① netbsdsrc/libexec/rpc.rusersd/rusers_proc.c:86

② netbsdsrc/include/db.h:72-75

③ netbsdsrc/lib/libc/db/recno/extern.h:47-50

④ netbsdsrc/sys/arch/alpha/include/types.h:61-68

```
typedef long              int64_t;
typedef unsigned long     u_int64_t;
```

(2) 用前面声明的名称之一，为硬件表达方式已知的数量创建抽象名称：[①]

```
typedef u_int32_t in_addr_t;
typedef u_int16_t in_port_t;
```

最后，typedef 还经常用来模拟C++和 Java 中类型声明引入新类型的行为。在 C 程序中，typedef 经常为 struct(C 语言中与类最相近的概念)引入用同一名称标识的一个类型名。因此，下面的例子将 path 声明为 struct path 的替代名(synonym，也称同义词)。[②]

```
typedef struct path path;
struct path {
    [...]
```

练习 3.13 针对我们介绍的 typedef 的功能，在本书配套盘中找出 5 个不同的实例。

练习 3.14 应用 typedef 声明会对代码的易读性造成什么样的负面影响。

进 阶 读 物

如果不熟悉本章中我们讨论的这些概念，那么请考虑更新 C 语言的知识[KR88]。Hoare[Hoa73]中对不显式使用指针实现递归数据结构进行了理论论证。C 语言中指针的应用在 Sethi 和 Stone[SS96]中做了简明扼要的介绍，Koenig[Koe88，pp. 27-46]对使用指针的许多陷阱作了讨论。[CWZ90]包含指针和结构的分析。许多有趣的文章[FH82，Suz82，LH86]对操作基于指针的数据结构，利用指针的属性作了探讨。Ellis 和 Stroustrup[ES90，pp. 217-237]对C++中虚函数的典型实现进行了介绍。动态存储分配所依据的算法在 Knuth[Knu97，pp. 435-452]中做了讨论，实用的建议可以在两本其他的著作中[Bru82，DDZ94]找到。采用引用计数的垃圾回收的概念在 Christopher[Chr84]中进行了论述，Boehm[Boe88]概括了保守垃圾回收器的实现。

① netbsdsrc/sys/arch/arm32/include/endian.h:61-62

② netbsdsrc/sbin/mount_portal/conf.c:62-63

第4章　C数据结构

忽视数据，实际上，是人们协调内心矛盾时采取的最简单，也最普遍的方式。

——William James

程序的作用是将算法应用到数据之上。数据的内部组织对算法的执行至关重要。

许多不同的机制都可以将相同类型的元素组织成集合，每种机制都提供各自不同形式的存储与访问数据的方式。用数组实现的向量(vector)，提供对所有元素的随机访问，但它不适合运行期间大小发生改变的情况。向量还可以用来将多组元素组织成表(table)的形式，或堆叠成二维，创建矩阵(matrix)。向量的操作有时被限定只能发生在向量的某一端，栈(stack)就是如此；向量的操作也有可能被限定为以先入先出的次序(first-in first-out order)进行，这时候，向量就成为了队列(queue)。在元素的次序并不重要的情况下，常常用映射(map)创建查找表(lookup table)和集合(set)，后二者都可以用来形成元素的集合。

指针可以将数据元素链接到一起，许多数据结构都使用了指针的这种能力。链表(linked list)能够容易地动态增长，然而，对链表的访问只能以顺序的方式进行(栈和队列的操作也是如此)，而组织得当的树(tree)既允许基于键来访问其中的数据元素，也可以容易地动态增长，同时还允许以安排好的方式进行遍历。最后，我们分析与图(graph)——目前为止最灵活的链式数据结构的表达方式——相关的一些应用程序和代码模式。

诸如Java，C++和C#等语言，都在语言库中提供实现这些数据结构所需的抽象。在C中，这些数据结构一般在使用它们的代码体中直接进行编码，但是，它们的属性和可以执行的操作是通用的。本章的目标是，了解如何根据底层的抽象数据类型，阅读直接的数据结构操作。

4.1　向　　量

到目前为止，在C程序中，我们遇到的最常用的数据结构就是向量(用于临时存储时，常被称为缓冲区)。向量在一个内存区块中存储相同类型的元素，并且可以线性或随机两种方式对元素进行处理。C语言中，一般使用内建的数组类型实现向量，不再对底层实现进行抽象。例如，向量的第一个元素总是位于数组索引为0的位置。[1]

```
ProgramName = argv[0];
```

为了避免混淆，从此处开始，我们使用术语——数组(array)，来指代基于数组的向量实现，这在C程序中很普遍。向量的应用领域很广，激发我们释放出所有的聪明才智和创造力。可以辨别的编码模式很少(除了数组的名称中有buf字样之外——在BSD源码库中共计出现

① XFree86-3.3/contrib/programs/viewres/viewres.c:884

超过 50 000 次),但却有无数的方式可以导致错误,只有很少的方法能够帮助我们解开困难的代码序列。

处理数组中所有元素的典型方式是用 for 循环。①

```
char pbuf0[ED_PAGE_SIZE];

for (i = 0; i < ED_PAGE_SIZE; i++)
  pbuf0[i] = 0;
```

N 个元素的数组可以被序列 for (i=0; i<N; i++)完全处理;所有其他变体都应该引起警惕。continue 语句可以提前结束本次循环的处理,break 语句可以中断循环的执行,这两个语句都可以安全地使用。但是,如果循环体内的代码增加了循环的索引,则需要十分仔细地推理代码的工作方式。

C 函数库中的 memset 和 memcpy 函数经常用来初始化数组和复制数组的内容。前面我们列出的初始化代码,在其他程序中可能表达得更为简洁,如下所示:②

```
static char buf[128];

memset(buf, 0, sizeof(buf));
```

memset 将一段内存区域设为由第二个参数指定的值。使用 memset 函数可以方便地将单字节字符序列初始化为预定的值(例如空格),或者将整型序列初始化为 0。所有的其他应用,比如对宽字符(例如 Unicode)或浮点数的初始化都有风险,并且不可移植。完全不能保证一个给定字节值的重复会产生确定的浮点数或任意 Unicode 字符。

表达式 sizeof(x)总会得到用 memset 或 memcpy 处理数组 x 所需的正确字节数。它的优点(与显式地传递数组的大小相比)是数组中元素数目或个体大小的改变,可以通过 sizeof 表达式的值自动地反映出来。但要注意,在指针上应用 sizeof 得到的只是实际指针的大小,而不是指针指向的动态分配的内存块的大小。

相应地,memcpy 经常用于在两个数组间复制数据。③

```
int   forcemap[MAPSZ];   /* map for blocking < 1,x> combos */
int   tmpmap[MAPSZ];     /* map for blocking < 1,x> combos */
[...]
    memcpy(forcemap, tmpmap, sizeof(tmpmap));
```

(注意,上面例子中,将 sizeof 运算符应用到输出缓冲区 — forcemap 数组,会更安全,因为如果 memcpy 复制的字节超过目标的大小,会产生不可预料的结果。)memcpy 和 memset 削弱了语言的类型(检查)系统;需要手动地检验源参数和目的参数是否指向相同类型的元素。此外,如果源区域和目的区域互相重叠,那么 memcpy 的行为不明确;在这种情况下,应该使用 memmove 函数。

---

① netbsdsrc/sys/arch/bebox/isa/if_ed.c:1184-1187

② netbsdsrc/bin/chio/chio.c:659-662

③ netbsdsrc/games/gomoku/pickmove.c:68-324

数组的元素经常用 C 库函数 fwrite 保存到文件。①

```
if (fwrite(buf, sizeof(char), n, fp) != n) {
  message("write() failed, don't know why", 0);
```

之后，使用对应的 fread 函数可以从文件中将这些元素读入到主内存。②

```
if (fread(buf, sizeof(char), n, fp) != n) {
  clean_up("read() failed, don't know why");
```

这些函数将数据的内部表示存储到文件中。这种数据天生不能在不同的系统构架之间进行移植(例如，Intel 的机器和苹果的 Power PC)。两个函数都返回完成读或写的元素的数目。程序中应该测试函数的返回值，以核实函数是否执行了预期的行为。

在检查涉及数组处理的程序时，最常碰到的问题可能就是代码访问了数组之外的元素。一些语言，如 Java，C# 和 Ada，认为这种行为是一种错误，会引发一个异常。其他的语言，如 C 和C++，则返回未定义的值，或将值存储到随机的内存地址(可能是存储其他变量的地址)，而另外的语言，如 Perl 和 Tcl/Tk，则会自动扩展数组的区间。不管哪种情况，我们都希望确保不会发生这类错误。在简单循环中，常常只需进行非正式的论证，来演示程序只访问数组中合法有效的值。更为复杂的情况下，我们可以使用基于数组索引的循环不变式(loop invariant)的概念，构造一个更为严格的论证过程，证实您的程序。我们在 2.11 节介绍了一个应用这种方案的例子。

当我们调用以数组为参数的函数时，也会发生这个问题(数组越界——译者注)。数组的大小经常使用一种带外(out-of-band)机制传递给函数，一般是附加的参数，例如 C 库函数 qsort 和 memcpy 就采用了这种方式。一些不那么知名的函数，诸如 gets 和 sprintf，作为被调用者，无法得知数组的大小。在这种情况下，被调用的函数会轻易地写到数组边界之外，程序员几乎没有办法阻止它的发生。考虑下面的代码：③

```
main()
{
    char    p[80];
    [...]
    if (gets(p) == NULL || p[0] == '\0')
        break;
```

如果该程序接收到超过 80 个字符的输入，那么这些输入会重写其他的变量，甚至存储在栈上的控制信息。这类问题被称为是缓冲区溢出(buffer overflow)，它一直被蠕虫和病毒的编写者所利用，巧妙地获得对程序的控制，进而获得对系统的访问特权。因此，对那些可能重写它们的缓冲区参数的 C 库函数，比如 strcat，strcpy，sprintf，vsprintf，gets 和 scanf，在使用时要很小心，或者根本不使用这些函数，如果使用也应该仅仅用在对写出数据的数量能够进行控制与检验的环境中。标准 C 函数库或操作系统的特定扩展，为这些函数提供了安全的

① netbsdsrc/games/rogue/save.c:382-383

② netbsdsrc/games/rogue/save.c:370-371

③ XFree86-3.3/xc/config/util/mkshadow/wildmat.c:138-157

替代品，比如：strncat，strncpy，snprintf，vsnprintf，fgets。由于 gets 函数的数据就是程序的输入，所以控制数据的复制十分困难。下面的代码摘录恰如其分地表达出，在使用写缓冲区的函数，但不检查它的大小时，程序员的负罪感①。

```
/* Yes, we use gets not fgets. Sue me. */
extern char * gets();
```

请注意，虽然访问数组边界以外的元素不合法，但是 ANSI C 和C++允许计算紧接在数组结尾之后的元素的地址。计算数组边界以外的其他任何元素的地址都是不合法的，而且在某些构架下，即使没有任何实际的访问发生，也可能导致未知的行为。结尾后面元素的地址，在迭代数组时可以作为区间结尾的指示器。②

```
# define MAX_ADS 5
struct dr {                 /* accumulated advertisements */
    [...]
    n_long dr_pref;         /* preference adjusted by metric */
} *cur_drp, drs[MAX_ADS];
[...]
    struct dr *drp;
    [...]
    for (drp = drs; drp < &drs[MAX_ADS]; drp++ ) {
        drp -> dr_recv_pref = 0;
        drp -> dr_life = 0;
    }
```

在上面的示例中，虽然访问 drs[MAX_ADS]不合法，但它的地址可以合法地用于表达数组区间之外的第一个元素。区间(range)一般用区间内的第一个元素和区间后的第一个元素来表示。和这种约定等同的数学公式是包括(inclusive，数学上也称为关闭、封闭)低位边界元素，不包括(exclusive，数学上也称为打开)高位边界元素的区间(即数学上的左闭右开区间，一般表达为[a, b)，运用统一的区间表达方式，可以使代码更易于阅读且不易出错。——译者注)。这种表达区间的习惯用法可以帮助程序员避免“漏掉一个”的错误。在阅读使用这种不对称边界(asymmetric bound)的代码时，可以容易地推导出它们表达的区间所具有的一些性质：

- 不对称区间中元素的数目等于高位边界与低位边界的差。
- 当不对称区间的高位边界等于低位边界时，区间为空。
- 不对称区间中的低位边界代表区间的第一个元素；高位边界代表区间外的第一个元素。

练习 4.1 选择一个大型的软件系统，找到使用数组的地方，将它们的应用归类。说明 3 处更适合于使用可选的、规范的数据结构(比如C++标准模板库(STL)的 vector 类)的实例。讨论使用 vector 抽象数据类型和直接使用内建的数组类型，二者的优点与缺点。试找

① XFree86-3.3/xc/config/util/mkshadow/wildmat.c:134-135

② netbsdsrc/sbin/routed/rdisc.c:84-120

出不能使用 vector 接口的代码序列。

练习 4.2　定长(fixed-length)数组对程序的操作施加了强制的限制。在现有的代码中找出 10 处使用了这种限制的实例，并检查它们是否记录在相应系统的文档之中。

练习 4.3　设计一个表达式，当传递给 memcpy 的参数所代表的内存区域不发生重叠时，该表达式的值为 true。(假定内存地址是线性编码，但要注意，在许多处理器构架中，这个假定是错误的。)请说明，在将这类表达式编写为断言的情况下，它们如何使得系统更为健壮。

练习 4.4　有些函数容易受到缓冲区溢出的影响，找出调用这种函数的实例；要么证实这类问题不会发生，要么提供导致缓冲区溢出的环境。

4.2　矩阵和表

在实践中，经常会遇到二维的数据结构，在数据处理领域中称为表(table)，在数学领域中称为矩阵(matrix)。这两种结构在其他方面也存在不同：矩阵的元素均为相同的类型，而表的元素大多数情况下类型不同。这个区别决定了在 C 语言中每种结构的存储方式。

表一般作为 C 的结构(struct)数组进行存储与处理。表与关系型数据库紧密相关。表的每一行(数组元素)用来表示一条记录(record)，表的一列(结构成员)用来表示一个字段。基于内存的表可以静态地分配，比如我们在 4.5 节中分析的查找表，或者下面的示例——将 Internet 控制消息协议(Internet control message protocol，称 ICMP)的数字编码转换成字符串表示。①

```
static struct tok icmp2str[] = {
    { ICMP_ECHOREPLY,          "echo reply" },
    { ICMP_SOURCEQUENCH,       "source quench" },
    { ICMP_ECHO,               "echo request" },
    [...]
    { 0,                  NULL }
};
```

表还可以使用 malloc 动态分配，malloc 的参数是表的大小乘以表所包含的 C 结构的大小所得的积(图 4.1:1)。②

在处理表的过程中，我们经常需要引用表中特定的行，并访问其元素。另一种方案是使用整型索引变量(即下面示例中的 tvc)来表示表中的某个行。③

```
struct vcol *vc;
```

① netbsdsrc/usr.sbin/tcpdump/print-icmp.c:109-120

② netbsdsrc/usr.sbin/quot/quot.c:423-445

③ netbsdsrc/usr.bin/pr/pr.c:314-530

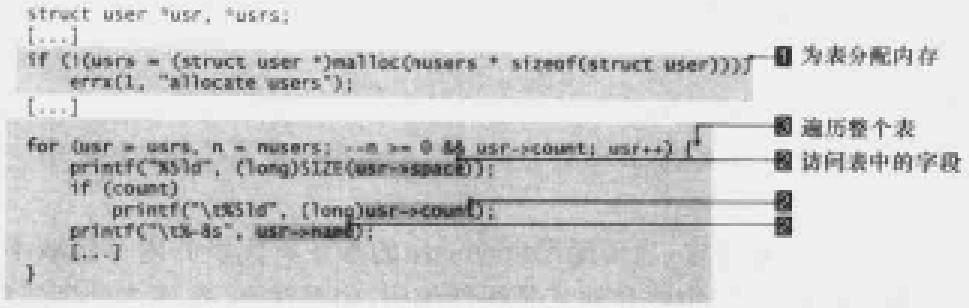


图 4.1 用结构的指针作为表的游标

```
int tvc;
[...]
cnt = vc[tvc].cnt;
```

然而，使用整型索引变量对表中的行进行访问，丢失了与访问的表有关的信息，只剩下一个可以表示许多不同(不兼容)含义的索引值。因而，最好使用指向特定结构元素的指针来表达表以及表内的单元。这个指针可以递增或递减，在表中移动，并且可以使用-> 运算符访问结构的字段(图 4.1:2)。数据库系统将这种抽象称为游标(cursor)。

矩阵的分配与处理使用一系列不同的技术。由于矩阵中所有元素的类型都相同，因此，可以使用多种方案对它们进行存储和处理。最明显的方式是使用二维数组分配存储空间并访问其中的元素。①

```
typedef double Transform3D[4][4];
[...]
IdentMat(Transform3D m)
{
    register int i;
    register int j;

    for (i = 3; i >= 0; -- i)
    {
        for (j = 3; j >= 0; -- j)
            m[i][j] = 0.0;
        m[i][i] = 1.0;
    }
}
```

但是，这种方法不能用来处理动态分配的矩阵。当矩阵的大小固定，或者列的数目已知时，代码中常常会声明指向列数组的变量，并根据行的数目和每列的大小分配内存。下面的例子②将 rknots 声明为一个拥有 MAXORD 个列的矩阵，并且动态地分配存储 numKnots 个

① XFree86-3.3/contrib/programs/ico/ico.c:110-1151

② XFree86-3.3/xc/programs/Xserver/PEX5/ddpex/mi/level2/miNCurve.c:364-511

行所需的内存。①

```
ddFLOAT (*rknots)[MAXORD]= 0; /*  reciprocal of knot diff */
[...]
if ( ! ( rknots = (ddFLOAT) (*)[MAXORD])
    xalloc( MAXORD *  numKnots *  sizeof(float))) )
```

之后，这些变量能够传递给函数，也可以用作二维数组(在下面的例子中为 kr)，因为在 C 语言中，数组的第一维可以省略。②

```
    mi_nu_compute_nurb_basis_function( order,i, knots, rknots, C )
    [...]
void
mi_nu_compute_nurb_basis_function( order,. span, knots, kr, C )
    [...]
    ddFLOAT kr[][MAXORD]; /*  reciprocal of knots diff */
    [...]
{
    [...]
    for ( k = 1; k < order; k++ )
        t0 = t1 *  kr[span- k+ 1][k];
```

如果矩阵中每一行的大小都不相同(对角线矩阵或对称矩阵即为这种情况)，存储矩阵时可以单独为每一行分配内存。然后，用指向元素指针的指针构成的数组保存每一行的地址。因此，给定指向这种结构(不是 C 语言中的 struct，而是指上面的指针数组——译者注)的指针 p，第 5 行的位置是 *(p+5)，该行中第 3 个元素可以表示为 *(*(p+5)+3)。令人欣慰的是，由于在 C 语言中可以将相应的偏移作为数组索引，快捷地存取指针所指向的内容，上面的表达方式可以写为 p[5][3]。重要的是，要注意 p 不是一个二维数组；它占用了额外的存储空间来存储行的指针，对元素的访问要进行指针的查找，而非仅仅通过元素位置的乘法计算就能够完成。如图 4.2 所示③，这个结构的元素分两个阶段进行分配：首先是指向每个行的指针构成的数组，然后是每行的个体元素。之后，就可以使用类似于访问二维数组的操作。

最后，还可以使用显式的编码或宏，模拟 C 语言访问二维数组的方式。图 4.3 展示出了这样一个例子④。该例中，矩阵作为元素的平面(即一维)数组进行分配，根据矩阵行的大小(n)的乘法可以确定出特定行的偏移；之后，通过将列的编号加到该偏移上，就可以定位到行中的元素。

练习 4.5　针对您所选用的关系型数据库管理系统，找到有关游标的文档。对比游标所

---

① Dave Thomas 在检查原稿时评论说：rknots 的代码迟早会发生问题，因为，矩阵中元素的类型与代码中用来分配空间的 xalloc 没有关系——声明使用了 ddFLOAT，xalloc 使用 float。我个人比较偏好下面这种写法：rknots =xalloc(MAXORD * numKnots * sizeof(rknots[0][0]));。

② XFree86-3.3/xc/programs/Xserver/PEX5/ddpex/mi/level2/minurbs.c:154-179

③ XFree86-3.3/xc/programs/Xserver/Xprint/pcl/PclText.c:638-655

④ XFree86-3.3/xc/programs/xieperf/convolve.c:91-434

```
PclFontMapRec ** index;                                    矩阵的声明
[...]
index = (PclFontMapRec **)xalloc(sizeof(PclFontMapRec *)*nindex_row);   分配行指针的存储空间
[...]
for (i=0; i<nindex_row; i++) {                             分配行内容的存储空间
    index[i] = (PclFontMapRec *)xalloc(sizeof(PclFontMapRec)*nindex_col);
    [...]
    for (j=0; j<=nindex_col; j++)
        index[i][j].fid = 0x0;                             Array-like access
}
```

图 4.2 基于指针的矩阵

```
#define    DATA( n, i, j ) ( float * ) ( *data + ( i * n ) ) + j      用来访问数组的宏
[...]
int    Boxcar3( float **data )
{
    int    i, j;

    if ( ( *data = ( float * ) malloc( sizeof( float ) * 9 ) ) ==      分配平面内存
        ( float * ) NULL )
        return( -1 );
    for ( i = 0; i < 3; i++ )
        for ( j = 0; j < 3; j++ )
            *(DATA( 3, i, j )) = 0.11111111;                       访问数组的元素
    return( 3 );
}
```

图 4.3 对矩阵元素的显式访问

提供的功能和结构的指针所提供的功能。

练习 4.6 表还常常存储为向量的形式，每个字段对应不同的数组。在本书配套盘中找到这样的一个实例，并说明如何用结构的数组来重写该代码。

练习 4.7 网络上存在大量开放源码的线性代数和矩阵库。下载两个这类软件包，找出矩阵的存储方式。

4.3 栈

相比于向量提供的访问方式，对栈的访问方式极为有限。栈只允许以后入先出(last-in-first-out，简称 LIFO)的方式在栈的结尾添加或移除元素。另外，我们还可以查询栈是否为空。栈提供的 LIFO 访问次序与递归定义的数据结构和算法十分匹配。以算术表达式中圆括号的处理为例：最后打开的圆括号将被首先关闭。实际上，常常可以将表达式定义为递归地应用到纯操作数或括号表达式(parenthesized expression，指括在圆括号内的表达式，整体上可以看作是一个操作数)上的运算符。存储中间结果的栈是处理此类表达式的理想结构。除了算术表达式外，递归定义和栈还用于块结构(block－structure)编程和标记语言(markup language)，数据类型，层次数据结构，程序执行、遍历和查找策略，图形算法，中断处理和用户应用程序中 undo 命令的实现。

C 语言中，栈经常用数组来实现。在结构化的程序中，基本的栈操作可能会抽象成抽象

数据类型(abstract data type,ADT)形式的单独函数,如图 4.4 所示。[①] 对程序的最初作者与维护人员来说,ADT 通过提供不依赖于底层实现的接口,将数据类型的实现与对它的使用隔离开来。作为一名读者,ADT 为底层实现的使用(或误用)方式提供一种信心的量度。一些语言和实现方法能够完全将接口与实现隔离开来,从而,不使用 ADT 提供的接口,很难对数据进行操作。基于数组的栈一般由存储栈元素的数组和栈指针组成,栈指针是一个整型数组索引,表明下一个存入或取出元素的位置(图 4.4:2)。我们所举的例子遵循所有这些约定,栈指针 sp 指出可以存储新的栈元素的位置。请注意,当添加元素时,栈指针执行后增量运算(图 4.4:3),但当移除元素时,栈指针执行的是前减量运算(图 4.4:4)。进栈和出栈操作之间总是存在这种对称(或对立)。在阅读含有显式编码(即明确地编写代码来完成对栈中元素的操作与栈指针的处理,而非依靠库或其他形式的封装所提供的功能——译者注)的栈访问代码序列时,要注意寻找这种对称,如下面的例子所示;请注意,两个操作在不同的函数中,中间有 84 行其他代码[②]。

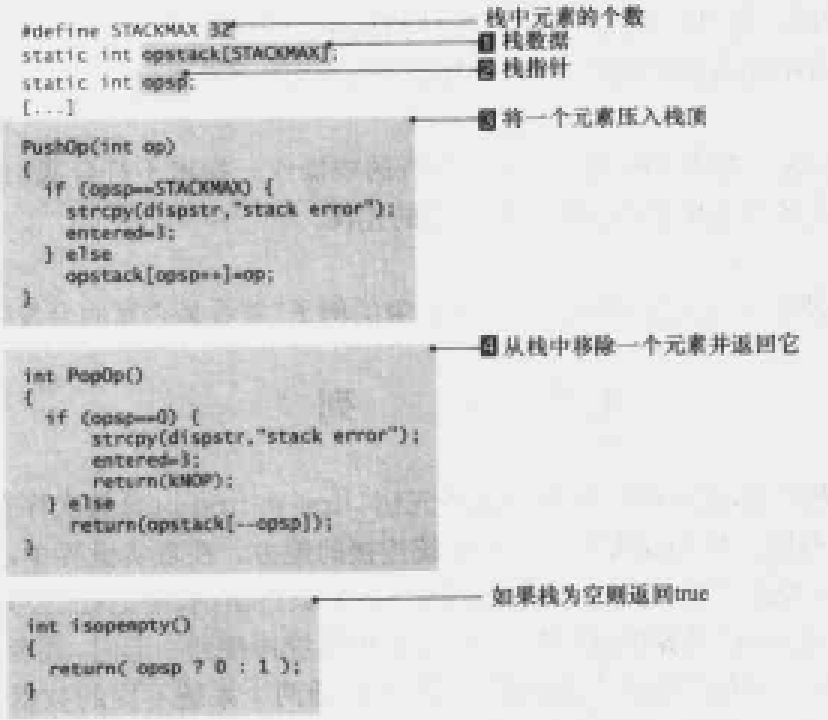


图 4.4　栈的一种抽象数据类型

```
if (regretp >= 0) {
    strcpy(lexstring, string_stack[regretp]);
    lexnumber = numberstack[regretp];
    return(regretstack[regretp --]);
}
[...]
```

① XFree86-3.3/contrib/programs/xcalc/math.c:827-860

② netbsdsrc/usr.bin/mail/list.c:518-617

```
if (++regretp >= REGDEP)
    errx(1, "Too many regrets");
regretstack[regretp] = token;
```

与栈操作相伴的一个特定问题是对上溢(overflow)和下溢(underflow)的检查，它随机地散布在程序的代码中。请注意，我们分析的栈的 ADT 对这两种情况都进行了检查。当栈的操作在行内执行时，这些检查经常由于懒惰或对简洁和效率的追求而省略掉。在这种情况下，由于我们不能确定，原作者是正确地推断出在这种情况下栈不会上溢或下溢，还是忘记了检查，所以也就难以确定程序的正确性。①

```
de_stack[--stack_top] = '0' + c % 8;
de_stack[--stack_top] = '0' + (c / 8) % 8;
de_stack[--stack_top] = '0' + c /64;
```

练习 4.8 在本书配套盘中找出栈数据结构的实例。找出这些实例中有多少使用 ADT 实现，有多少直接编码。统计这些实例中，栈指针指向第一个空数组元素的实现有多少，栈指针指向最后一个占用元素的实现有多少。讨论您所获得的结果。

练习 4.9 分析基于直接编码的实现中所执行的栈操作。找出不符合我们描述的模型的那些操作。就函数名和实现它们的代码提出您的建议。

练习 4.10 推理本节提供的进栈时不检查上溢的例子，看看是否真的会发生溢出。

4.4 队　　列

队列是一个元素集合，它允许数据项以先入先出(first-in-first-out)的次序在尾部(tail)加入，从头部(head)移除。队列经常用在两个系统连接的地方。在现实世界中，队列往往形成于一个实体(例如，普通大众、订单)与其他实体(例如，银行出纳、登记处、公共汽车、生产线)的交界处。类似地，软件世界中，队列用在两个软件系统连接到一起时，或者用在软件系统与硬件交界的地方。所有的情况下，队列都是用来管理两个系统不同的数据生成与处理特征。前一种情况的例子包括：窗口系统与用户应用程序的连接，操作系统对输入网络包的处理，以及邮件消息在邮件传输代理程序之间转发的方式。队列用在硬件交互中的例子包括：对网卡、磁盘驱动器、串行通信设备和打印机所生成的数据和请求的处理。所有情况下都是由一个系统生成数据，放入一个队列，在适当的间隔后，由另外的系统对其进行处理。

如果队列用数组来实现，则常常将队列称为循环缓冲区(circular buffer)或环形缓冲区(ring buffer)。每次从头部移除元素，都向前移动整个队列的所有元素，这样做效率很低，因此，基于数组的队列经常通过使用两个数组索引来实现：一个指向往数组中添加元素的位置，一个指向从数组中移除元素的位置。这些索引到达数组的末尾时，必须回绕到数组的开始。完成这项工作一般使用条件语句：②

① XFree86-3.3/xc/util/compress/compress.c:1325-1327

② netbsdsrc/sys/arch/mac68k/dev/adb_direct.c:1610-1611

```
if (++ adbInHead >= ADB_QUEUE)
    adbInHead= 0;
```

条件表达式①：

```
static int history_head, history_tail;
#define hist_bump(h) ((++(h) -= NUM_HISTORY) ? ((h) = 0) : 0)
[...]
    for (i = history_tail; i != history_head; hist_bump(i))
        DrawPoints (old_pixmaps[i], bit_0_gc, xp, n);
```

或取模算法②：

```
adb_evq[(adb_evq_len + adb_evq_tail) %  ADB_MAX_EVENTS] =
    * event;
```

许多队列使用一个头索引和一个尾索引来实现。③

```
volatile int        rnd_head;
volatile int        rnd_tail;
[...]
volatile rnd_event_t    rnd_events[RND_EVENTQSIZE];
```

这类实现中，经常将头索引指向第一个要移除的元素，将尾索引指向第一个空元素。因此，如果头索引和尾索引包含同样的索引，则表示队列为空。④

```
/*
  *  check for empty queue
  */
if (rnd_head == rnd_tail)
    return;
```

尾索引指向头索引的前一个元素表示队列已满⑤。

```
/*
  *  check for full ring. If the queue is full and we have not
  *  already scheduled a timeout, do so here.
  */
nexthead = (rnd_head + 1) & (RND_EVENTQSIZE - 1);
if (nexthead == rnd_tail) {
```

请注意，在我们分析的这个实现中，元素均为在队列的头部添加，从其尾部移除。还要注意，由于要保持一个元素为空，以区分满队列与空队列，所以这种方案浪费了队列中的一个元素。考虑队列中只有一个元素时，如何表示空队列和满队列，就能够了解上面所说的情

① XFree86-3.3/xc/programs/beforelight/b4light.c:76-166
② netbsdsrc/sys/arch/mac68k/dev/adb.c:133-134
③ netbsdsrc/sys/dev/rnd.c:99-102
④ netbsdsrc/sys/dev/rnd.c:826-830
⑤ netbsdsrc/sys/dev/rnd.c:763-768

况。我们可以显式地存储队列中元素的数目，避免这个问题以及由这个问题引发的混淆，如图 4.5 所示①。这种存储方法还可以优化成只存储队列的尾索引和元素的数目②。

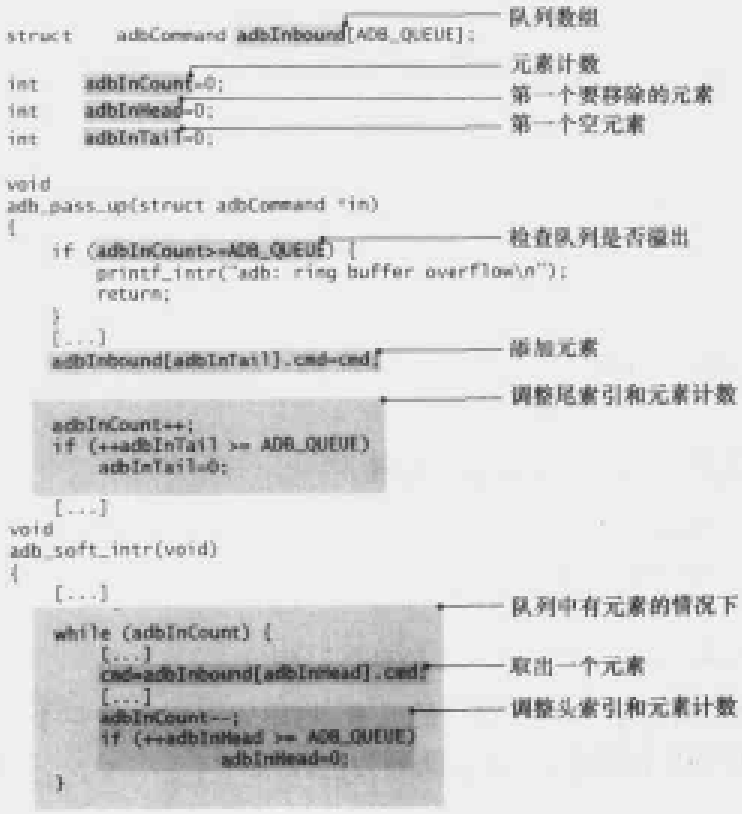


图 4.5 拥有显式元素计数的队列

```
static adb_event_t adb_evq[ADB_MAX_EVENTS]; /*  ADB event queue */
static int adb_evq_tail = 0;    /*  event queue tail */
static int adb_evq_len = 0;    /*  event queue length */
```

练习 4.11 在本书配套盘中找出 10 个队列的实例，并确定它们所连接的系统。

练习 4.12 在本书配套盘中，统计元素添加到队列头部和元素添加到队列尾部的实例。提出避免这种混乱的策略。

练习 4.13 在源代码文档中查找单词“queue”，并试着确定对应的数据结构。至少找出两例没有使用队列数据结构的情况，并论证出现这种分歧的原因。

4.5 映　　射

用数组的索引变量访问数组的元素效率非常高，一般可以用一个或两个机器指令实现。

① netbsdsrc/sys/arch/mac68k/dev/adb_direct.c:239-1611

② netbsdsrc/sys/arch/mac68k/dev/adb.c:72-74

对于组织以从零开始的连续整数为键(key)的简单映射(map)和查找表(lookup table)来说，这项特性使得数组成为它们的理想之选。在预先知道数组元素的情况下，可以直接用这些元素初始化数组。①

```
static const int        year_lengths[2] = {
        DAYSPERNYEAR, DAYSPERLYEAR
};
```

本示例中定义的数组只有两个元素，分别表示正常年份和闰年的天数。重要的不是数组的大小，而是通过它的索引使得统一的随机访问成为可能。②

```
janfirst += year_lengths[isleap(year)] *  SECSPERDAY;
[...]
while (days < 0) || days >= (long) year_lengths[yleap = isleap(y)]) {
[...]
yourtm.tm_mday += year_lengths[isleap(i)];
```

多维的表中也常使用同样的原则。③

```
static const int        mon_lengths[2][MONSPERYEAR] = {
  { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
  { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
};
```

上面的代码将一个二维数组初始化为存储正常年份和闰年中每个月的天数。您可能会觉得存储在 mon_lengths 数组中的信息没有什么用处，仅仅是浪费空间。并非如此，数组经常用来对控制结构进行高效编码，简化程序的逻辑。考虑下面的代码如果使用 if 语句，应该怎样编写。④

```
value += mon_lengths[leapyear][i] *  SECSPERDAY;
[...]
if (d + DAYSPERWEEK >= mon_lengths[leapyear][rulep -> r_mon - 1])
[...]
i = mon_lengths[isleap(yourtm.tm_year)][yourtm.tm_mon];
```

使用数组编码程序的控制部分，所得出的逻辑结果完全是表驱动操作。最常见的情况是，数组在每个位置存储一个数据元素和一个函数指针——指向处理数据元素的函数，从而将代码与数据关联起来。图 4.6⑤ 的例子使用了一个用命令名和函数指针(指向用来执行相应命令的函数)初始化的数组。ksearch 函数对数组的内容(按命令名升序排列)进行二分查找，查找命令名。如果找到该命令，ksearch 将调用相应的函数。归纳上面的例子，我们注意到，数组可以通过存储供程序内的抽象机(abstract machine)或虚拟机(virtual machine)使

① netbsdsrc/lib/libc/time/localtime.c:453-455
② netbsdsrc/lib/libc/time/localtime.c:832-1264
③ netbsdsrc/lib/libc/time/localtime.c:448-451
④ netbsdsrc/lib/libc/time/localtime.c:682-1373
⑤ netbsdsrc/bin/stty/key.c:74-148

用的数据或代码，控制程序的运作。这种方案的典型例子包括使用下推和栈自动机实现的语法分析器和词法分析器，还有基于虚拟机指令的执行面实现的解释器。


```
static struct key {
    char *name;
    void (*f)(struct info *);
    [...]
} keys[] = {
    { "all",    f_all,      0 },
    { "cbreak",    f_cbreak,    F_OFFOK },
    [...]
    { "tty",    f_tty,      0 },
};
[...]
int
ksearch(char ***argvp, struct info *ip)
{
    char *name;
    struct key *kp, tmp;

    name = **argvp;
    [...]
    tmp.name = name;
    if (!(kp = (struct key *)bsearch(&tmp, keys,
        sizeof(keys)/sizeof(struct key), sizeof(struct key), c_key)))
        return (0);
    [...]
    kp->f(ip);
    return (1);
}
```

```
void
f_all(struct info *ip)
{
    print(&ip->t, &ip->win, ip->ldisc, BSD);
}
```

图 4.6 表驱动的程序运作

图 4.6 中，一个次要但经常遇到的惯用法就是 key 数组中元素数目的计算。在定义 key 数组时，没有给出它的大小；编译器根据用来初始化它的元素个数来确定数组的实际大小。程序中，需要用到数组元素数目的地方，可以使用表达式 sizeof(keys)/sizeof(struct key)的结果。可以将表达式 sizeof(x)/sizeof(x[0])理解为数组 x 中元素的个数。元素的数目是编译期间的常量，通过将整个数组所需的大小除以数组中每个元素的大小，可以计算出这个常量的值。由于这项操作的结果是一个编译期间常量，所以它可以用来指定或初始化其他的数组。①

```
struct protosw impsw[] = {
{ SOCK_RAW,     &impdomain, 0, PR_ATOMIC|PR_ADDR,
  [...]
},
};

struct domain impdomain =
  { AF_IMPLINK, "imp", 0, 0, 0,
    impsw, &impsw[sizeof(impsw)/sizeof(impsw[0])] };
```

① netbsdsrc/sys/netinet/in_proto.c:167-177

4.5.1　散列表

查找表(lookup table)并非总是用静态数据初始化,使用从 0 开始的键索引来访问。大多数情况下,我们使用数组提供的高效随机访问方式来组织在程序运行期间会发生改变的映射,使用不同于简单数组索引的键来查找表中的元素。我们将这种方案称为散列表(hash table)。这种方案基于的中心思想是从我们希望用来查找元素的键构造出一个数组索引。当查找键是其他(可能很大)整数时,一种简单的解决方案是将键除以数组中元素数目的余数作为数组的索引。①

```
/*
  * hash inode number and look for this file
  */
indx = ((unsigned)arcn -> sb.st_ino)% L_TAB_SZ;
if ((pt = ltab[indx]) != NULL) {
```

上面的示例将文件的惟一标识数 st_ino 用作 ltab 数组的索引,ltab 数组共有 L_TAB_SZ 个元素。取模运算的结果一定在数组索引允许的区间之内。从而可以直接用来访问数组的元素。经常可以使数组中元素的数目等于 2 的幂,用逐位与(bitwise-AND)运算(参见 2.10 节)替代取模运算,避免取模运算带来的开销。

当用在映射上的索引变量不是一个整数时,需要使用散列函数将其转换成一个整数。散列函数合并键的数据元素,将它们转换成一个整数。例如,图 4.7:1② 的散列函数可以将字符串转换为整数,它采用的方法是:散列值每次向左移 3 个二进制位,依次与字符串中每个字符进行异或运算;并在转换结束时将负值转换为正值。在数组中进行查找之前,使用逐位与运算(图 4.7:2)将计算得出的值限制在数组的有效索引之内。要注意,多个不同的映射索引值可能会计算到数组中相同的位置。因此,在定位到数组中某个位置之后,我们需要核实该位置的元素确实是我们正在查找的元素(图 4.7:3)。多个元素可能需要存储在相同的位置,这种情况最复杂,也最难处理。这种情况下,常用的策略是使用另外的函数计算新的候选位置(图 4.7:4)。之后,在放置元素时,如果元素的主位置被占用,就使用这些备用位置;在查找元素时,如果主位置不是我们正在寻找的元素,则切换到备用位置继续查找(图 4.7:5)。

练习 4.14　在本书配套盘中找出一个使用数组编码控制信息的实例。直接使用控制语句重写这段代码。估量二者的代码在行数上的差别,并评论两种方案的可维护性、可扩展性和效率。

练习 4.15　在代码中,找出可以为数组查找机制所替代的控制结构。说明代码应该如何实现。

---

① netbsdsrc/bin/pax/tables.c:163-167

② XFree86-3.3/xc/lib/font/util/atom.c:55-173

图 4.7 散列函数和散列表的访问

练习 4.16 在本书配套盘中，找出至少 3 处字符串散列函数的实现。选择一系列具有代表性的输入字符串，根据数组中同一位置发生冲突的元素数目以及函数的执行时间，对比这几种实现。

练习 4.17 您是否考虑过，如果将基于散列算法的映射实现为一种抽象数据类型可能会遇到什么样的困难？给出克服它们的建议。

4.6 集　　合

有些情况下，我们希望高效地表达和处理元素的集合(Set)。当这些元素可以表达为比较小的整数时，经常使用的典型实现是将该集合表达为二进制位的数组，集合中每个元素都基于特定位的值。C 语言中，没有数据类型可以用来将二进制位表达成数组，并像数组一样进行操作。为此，程序中一般使用某种整型数据类型(char，int)作为底层存储元素，用移位和逐位与/或运算符来定位特定的位。例如，下面的代码会使 j 成为集合 pbitvec 的一员。①

① XFree86-3.3/xc/programs/Xserver/record/set.c:269

```
pbitvec[j/BITS_PER_LONG] |= ((unsigned long)1 << (j % BITS_PER_LONG));
```

数组 pbitvec 由长整型(long)整数组成，每个长整型整数包含 BITS_PER_LONG 个二进制位。因此，每个数组元素可以存储 BITS_PER_LONG 个集合成员的信息。故而，我们用集合成员编号 j 除以 BITS_PER_LONG，找到数组中包含特定元素信息的位置，之后，使用该除法的余数对 1 左移位，将它定位到存储特定元素信息的位。通过将现有的数组值与我们创造的位值执行逐位或(bitwise-OR)运算，我们将数组中的该位设为 1，表示 j 属于集合的成员。类似地，我们可以将构造的位值与相应的数组元素进行逐位与(bitwise-AND)运算，测试一个元素是否为位集的一个成员。①

```
# define FD_ISSET(n, p) |
((p) -> fds_bits[(n)/NFDBITS] & (1 << ((n) % NFDBITS)))
```

最后，我们可以将数组元素与我们构造的位值的补码，进行逐位与运算，将一个元素从集合中移除。②

```
# define FD_CLR(n, p) \
  ((p) -> fds_bits[(n)/NFDBITS] &= ~ (1 << ((n) % NFDBITS)))
```

由于我们构造的数值仅有一个位为 1，因此它的补码除了我们希望移除的位以外，所有的位都为 1，从而能够屏蔽掉数组的其他值，清除特定的位。在 2.10 节中，我们曾简短地提到过，移位和逐位与运算经常用来替代除法和余数运算符；我们叙述的集合操作一般都使用基于位的运算符来实现。③

```
resourceQuarks[q >> 3] |= 1 << (q & 0x7);
```

在程序中常常会看到，应用我们概括的表达方式，通过直接用逐位或(bitwise-OR)和逐位与(bitwise-AND)运算符合并数组的元素，实现集合的合并与相交④。

```
#define   XFD_ANDSET(dst,b1,b2)\
  (dst)-> fds_bits[0] = ((b1)-> fds_bits[0] & (b2)-> fds_bits[0];\
```

练习 4.18　在本书配套盘中，分别找出使用位运算符和算术运算实现集合的 ADT。将相同的 ADT 实现为字符、布尔和整型数组。编写一个小程序，比较 5 种不同实现在效率上的差异。针对中型(1000 个元素)和超大型(您机器内存大小的 5 倍)的集合，调整所得到的结果。

练习 4.19　在本书配套盘中，找出使用集合的实例。针对每个特定的问题，建议另外的实现策略。评论一下，使用集合数据类型，是否部分原因是出于为它创建高效的实现较为容易。

① XFree86-3.3/xc/include/Xpoll.h:99
② XFree86-3.3/xc/include/Xpoll.h:96
③ XFree86-3.3/xc/lib/X11/Xrm.c:990
④ XFree86-3.3/xc/include/Xpoll.h:133-134

4.7 链　　表

C程序中，最简单，也是最常用的链式数据结构是链表(linked list)。链表通过指针将表达链表元素的结构联合到一起，构造而成。元素一般添加到链表的前面，即链表的头部(head)。然而，由于链表中所有的元素都通过指针链接在一起，因此元素可以高效地从链表中的任何位置添加或移除，而在大型的数组中执行这类操作可能需要巨大的开销。要在链表中定位一个特定的项，必须从链表的开始遍历链表；不可能随机地访问其中的元素。链表经常用来存储顺序访问的数据，这些数据在程序运行期间可以动态地扩展。

链表经常使用十分特殊的编码风格；因此，代码序列的用途比较容易确定。如果结构中含有指向结构自身、名为 next 的元素，一般说来，该结构定义的是单向链表的结点。链表的结点由数据(除 next 以外的其他结构字段，见图 4.8:2①)和指向下一个链表元素的 next 指针组成(图 4.8:1)。指向链表结点的持久性(如全局、静态或在堆上分配)指针常常表示链表的头部(图 4.8:3)。图 4.9 演示了这种方案。当链表头部的值为 NULL 时，链表为空。对链表的遍历，要使用指向链表结点的指针，从列表头部开始，使用每个结点的 next 指针向前推进。这种访问序列经常用 for 循环来完成，如图 4.8:4 所示。请注意，图 4.8 中定义的链表只能前向(forward)遍历列表中的元素。在查找特定的元素时，元素找到后会提前退出 for 循环；如果循环正常结束，则说明没有找到该元素。将新元素添加到链表中时要用到一个精心设计，但也很通用的序列：在为新结点分配内存并存入数据后，将新结点指向原来链表头部指向的结点，将链表头部指向这个新链表结点(图 4.8:5)。移除链表的头元素比较容易，只需将头部指向链表的下一个元素。因此，链表还经常用来表达栈②。

```
/*  push alias expansion */
[...]
s -> next = source;
source = s;
[...]
source = s -> next;    /*  pop source stack */
```

移除链表中的任意元素是一件很复杂的事，因为在移除的过程中，我们需要用到一个指向特定元素指针的变量。指向链表指针的指针就是用于这个目的，如下面的例子所示③。

```
STATIC int
unalias(char * name)
{
    struct alias * ap, * * app;

    app = hashalias(name);
    for (ap = * app; ap; app = &(ap -> next), ap = ap -> next) {
```

① netbsdsrc/usr. bin/rup/rup. c:60-97

② netbsdsrc/bin/ksh/lex. c:639-644,845

③ netbsdsrc/bin/sh/alias. c:120-152

```
struct host_list {
    struct host_list *next;          [1] 下一个链表元素
    struct in_addr addr;             [2] 链表结点的数据
} *hosts;                            [3] 链表的头部
```

查找链表中的某一项

```
int
search_host(struct in_addr addr)
{
    struct host_list *hp;

    [...]
    for (hp = hosts; hp != NULL; hp = hp->next) {     [4] 在链表的元素中迭代
        if (hp->addr.s_addr == addr.s_addr)           找到了查找项
            return(1);
    }
    return(0);                                        未找到查找项
}
```

添加一个链表项

```
void
remember_host(struct in_addr addr)
{
    struct host_list *hp;

    if (!(hp = (struct host_list *)malloc(sizeof(struct host_list)))) {
        err(1, "malloc");
        /* NOTREACHED */
    }
    hp->addr.s_addr = addr.s_addr;        存储数据
    hp->next = hosts;                     [4] 将元素链接到列表中
    hosts = hp;
}
```

图 4.8　链表的定义与基本操作

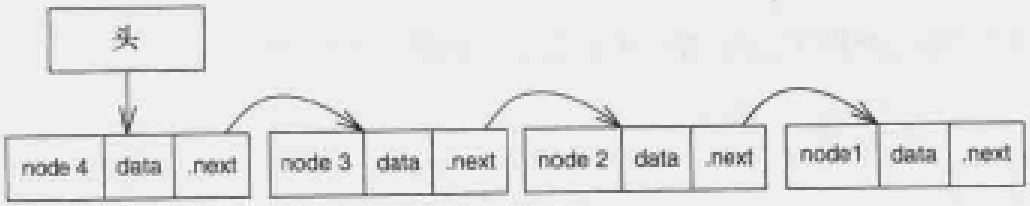


图 4.9　单向链表

```
        if (equal(name, ap -> name)) {
            [...]
            * app = ap -> next;
            ckfree(ap -> name);
            ckfree(ap -> val);
            ckfree(ap);
            [...]
            return (0);
        }
    }
    return (1);
}
```

在此，app 最初指向链表的头部。变量 ap 用于遍历这个链表，与此同时，app 总是指向存储当前元素指针的位置。因此，当找到要移除的元素时，只需将 * app 设为指向该元素的下一个元素，就可以高效地将找到的元素从链表的链中移除。

请注意，在前面的例子中，元素被删除后，就不能再使用 ap 向前访问下一个元素，因为

ap 的内容(包括 ap-> next)已经被释放。访问已经删除的链表元素的 next 成员是一个常见的错误[①]。

```
for (; ihead != NULL; ihead = ihead -> nextp) {
    free(ihead);
    if ((opts & IGNLNKS) || ihead -> count == 0)
        continue;
    if (lfp)
        log(lfp, "%s: Warning: missing links\n",
            ihead -> pathname);
}
```

上面的示例中,释放 ihead 之后,又访问了 ihead-> nextp 和 ihead-> count。虽然这段代码在一些构架和 C 函数库的实现中能够工作,但已经释放的内存块中的内容不能再进行访问,它能够工作只是出于偶然。这个错误在删除整个链表时尤其普遍。删除这类元素的正确方式是使用临时变量存储下一个元素的地址[②]:

```
for (j = job_list; j; j = tmp) {
    tmp = j -> next;
    if (j -> flags & JF_REMOVE)
        remove_job(j, "notify");
}
```

或在向前推进链表的指针之前,将要释放的元素的地址存储在临时变量中。[③]

```
struct alias * ap, * tmp;
[...]
while (ap) {
    ckfree(ap -> name);
    ckfree(ap -> val);
    tmp = ap;
    ap = ap -> next;
    ckfree(tmp);
}
```

链表有许多不同的种类,我们前面介绍的单向链表最常使用。链表的一种变体是将每个链表元素与它的前驱与后继都链接起来。这类链表称为双向链表(doubly linked list)(图 4.10)。双向链表的标志信号就是以 prev 命名的指针[④]。

```
struct queue {
    struct queue * q_next, * q_prev;
}
```

① netbsdsrc/usr.bin/rdist/docmd.c:206-213
② netbsdsrc/bin/ksh/jobs.c:1050-1054
③ netbsdsrc/bin/sh/alias.c:164-177
④ netbsdsrc/sys/arch/arm32/arm32/stubs.c:82-84

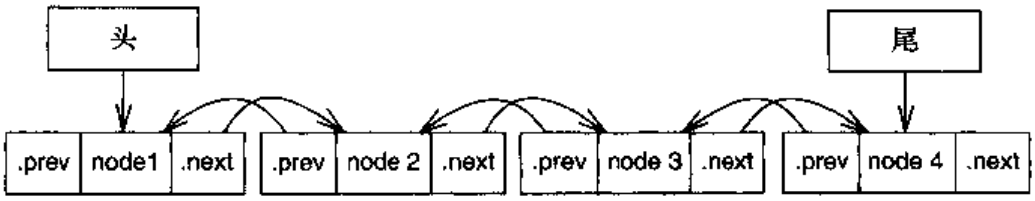


图 4.10　双向链表

双向链表有两个优点：(1)能够在链表的任意位置插入和移除元素，除了需要一个指针，指向插入或移除发生在哪个元素上之外，不需要任何额外的环境。(2)可以对链表进行逆向遍历。双向链表经常用来实现双端队列(double-ended queue，也称为 deque)——两端都可以添加或移除元素的队列。

逆向遍历双向链表一般使用 for 循环来执行。①

```
for (p2 = p2 -> prev; p2 -> word[0] != ')'; p2 = p2 -> prev)
```

往双向链表中添加元素和从中移除元素的操作比较复杂，通过用方框代表元素，用箭头代表指针的改变，可以使它们更易于理解。考虑下面的代码片断，它将 elem 链表结点加到 head 结点之后。②

```
register struct queue * next;
next = head -> q_next;
elem -> q_next = next;
head -> q_next = elem;
elem -> q_prev = head;
next -> q_prev = elem;
```

图 4.11 所示的对应步骤，清楚地说明了数据结构的指针如何逐渐调整，将 elem 链接到链表中。上面的示例使用 next 临时变量，简化了指针表达式。其他情况下，邻近的链表元素直接使用锚元素(achor element)的 prev 和 next 指针来表示。③

```
ep -> prev -> next = ep -> next;
if (ep -> next)
    ep -> next -> prev = ep -> prev;
```

虽然，这样的代码乍看起来有些令人害怕，但是，使用一只铅笔和一张纸，稍加推理就能够快速地解开它的功能(上面的代码从双向链表中移除元素 ep)。

有时，链表中最后一个元素并不指向 NULL，而是指回链表的第一个元素(图 4.12)。这样的链表称为循环链表(circular linked list)，它经常用来实现环形缓冲区或其他具有回绕行为的结构。循环链表不能使用我们前面看到过的 for 序列来进行遍历，因为没有结束元素来终止循环。一般地，遍历循环列表的代码将访问的第一个元素存储到一个变量中，然后

① netbsdsrc/bin/csh/parse. c:170

② netbsdsrc/sys/arch/arm32/arm32/stubs. c:96-102

③ netbsdsrc/usr. bin/telnet/commands. c:1735-1747

进行循环，直至再次到达该元素为止。①

```
void
prlex(FILE * fp, struct wordent * sp0)
{
    struct wordent * sp = sp0 -> next;

    for (;;) {
        (void) fprintf(fp, "%s", vis_str(sp -> word));
        sp = sp -> next;
        if (sp == sp0)
            break;
        if (sp -> word[0] != '\n')
            (void) fputc(' ', fp);
    }
}
```

上面的例子从位置 sp 开始，使用 sp0 作为循环的退出标记，打印循环链表的内容。

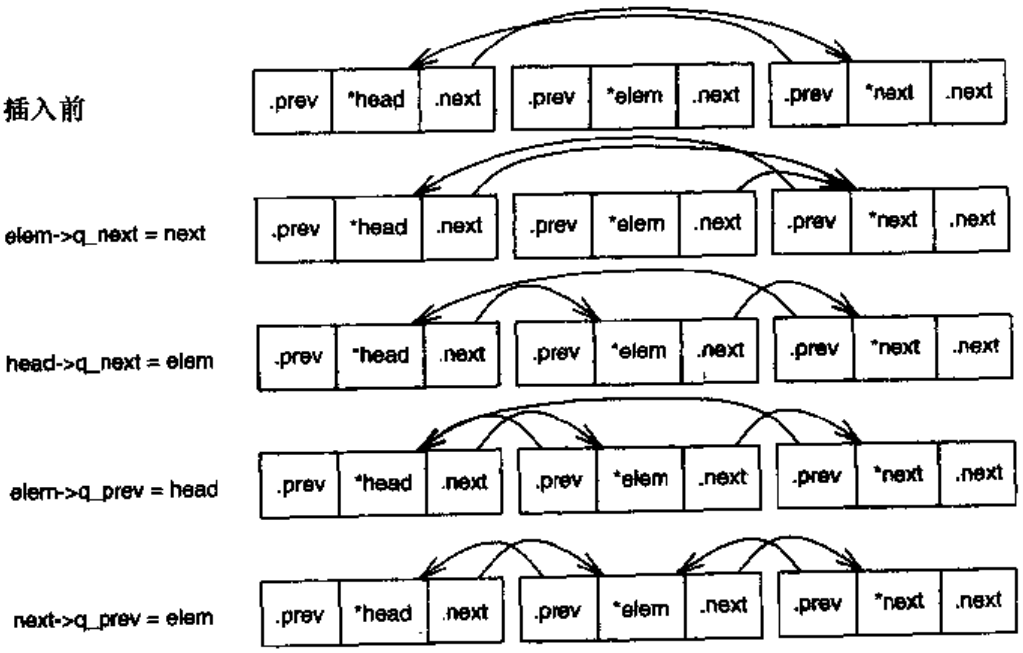


图 4.11 向双向链表添加一个元素

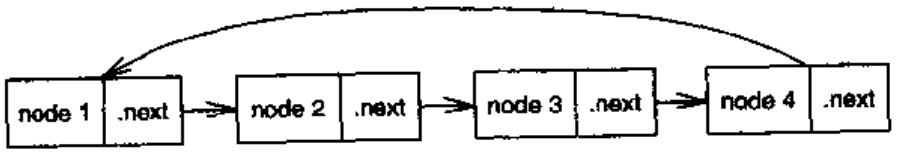


图 4.12 循环链表

链表还用来在单一散列表入口中存储多个元素，如图 4.13 所示。当在映射中查找某个

① netbsdsrc/bin/csh/lex.c:187-202

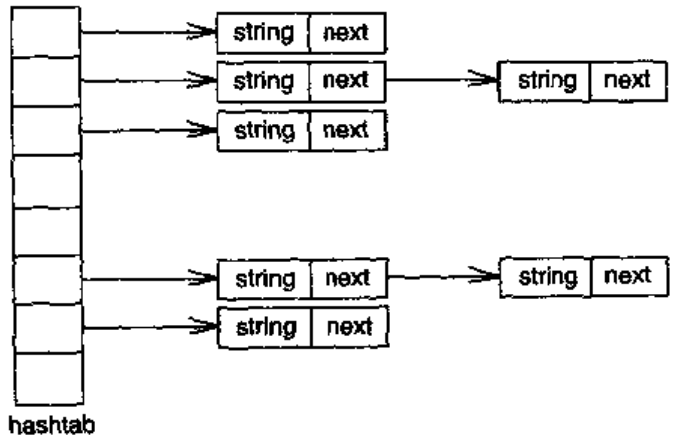


图 4.13　由链表构成的散列表

元素时，散列函数能够高效地定位出应该在哪个链表中查找该元素，然后进行简单的列表遍历，找出特定的元素。①

```
struct wlist *
lookup(char *s)
{
    struct wlist *wp;

    for (wp = hashtab[hash(s)]; wp != NULL; wp = wp -> next)
        if (*s == *wp -> string && strcmp(s, wp -> string) == 0)
            return wp;
    return NULL;
}
```

这种结构，用适度的实现成本提供对数据元素（标识符）的高效访问。它经常用于构造符号表（symbol table）。（符号表经常用在语言的各种处理器中，比如解释器和编译器，用来存取与特定标识符相关联的细节，例如，给定变量名所对应的类型信息。）

练习 4.20　在本书配套盘中，找出 5 个单向和双向链表，说明它们执行的功能。

练习 4.21　调用下面的函数时，要传入一个指针，指向双向链表中的元素。②

```
void
_remque(void *v)
{
    register struct queue *elem = v;
    register struct queue *next, *prev;

    next = elem -> q_next;
```

① netbsdsrc/games/battlestar/parse.c:70-80

② netbsdsrc/sys/arch/arm32/arm32/stubs.c:109-121

```
    prev = elem -> q_prev;
    next -> q_prev = prev;
    prev -> q_next = next;
    elem -> q_prev = 0;
}
```

使用框图，说明它执行的操作。

4.8 树

大量的算法和组织信息的方法依靠树作为底层数据结构。树的正式定义表明：树的结点通过边连接到一起，从每个结点到树的根有且只有一条路径。树的结点可以在每一层进行扩展，这种方式经常用来有效地组织和处理数据。20 层深的二叉树（从最低层到根有 20 个结点）能够保持大约 1 百万（2^{20}）个元素。许多查找①、排序②、语言处理③、图形④和压缩⑤算法都依赖于树形数据结构。另外，树还用来组织数据库文件⑥、目录⑦、设备⑧、内存体系⑨、属性（如 Microsoft Windows 的注册表，X Window 系统的默认规格）、网络路由⑩、文档结构⑪和显示元素⑫。

在支持指针的语言（Pascal，C，C++），或支持对象引用的语言（Java，C#）中，一般通过将父结点与它的子结点链接起来实现树形数据结构。这要用到递归的类型定义（即在类型的定义中要用到该类型，比如下面的代码中，结构 tree_s 的定义中就用到了 tree_s 结构——译者注），定义中声明树由指向其他树的指针或引用构成。下面的代码定义了一个二叉树（binary tree）：每个结点至多只能含有两个其他结点。⑬

```
typedef struct tree_s {
    tree_t      data;
    struct tree_s   *left, *right;
    short       bal;
    }
    tree;
```

① netbsdsrc/games/gomoku/pickmove.c
② netbsdsrc/usr.bin/ctags/ctags.c
③ netbsdsrc/bin/sh/eval.c
④ XFree86-3.3/xc/programs/Xserver/mi/mivaltree.c
⑤ netbsdsrc/lib/libz/infblock.c
⑥ netbsdsrc/lib/libc/db/btree
⑦ netbsdsrc/usr.bin/find/find.c
⑧ netbsdsrc/sys/dev/isapnp/isapnpres.c
⑨ XFree86-3.3/xc/programs/Xserver/hw/xfree86/accel/cache/xf86bcache.c
⑩ netbsdsrc/sbin/routed/radix.c
⑪ XFree86-3.3/xc/doc/specs/PEX5/PEX5.1/SI/xref.c
⑫ XFree86-3.3/xc/programs/Xserver/dix/window.c
⑬ netbsdsrc/usr.sbin/named/named/tree.h:34-39

二叉树的结点一般命名为 left 或 right，以表示它们在树中的位置。二叉树经常用来高效地排序和查找可以定义次序的数据。Internet 名称服务器 named，就使用这种方式来组织 Internet 地址。二叉树在构造过程中遵循：如果结点的值大于给定父结点中存储的值，那么要将它放在父结点的右方；如果小于父结点中存储的值，则放在父结点的左方。我们在图 4.14 中例举出这种树的一个例子。

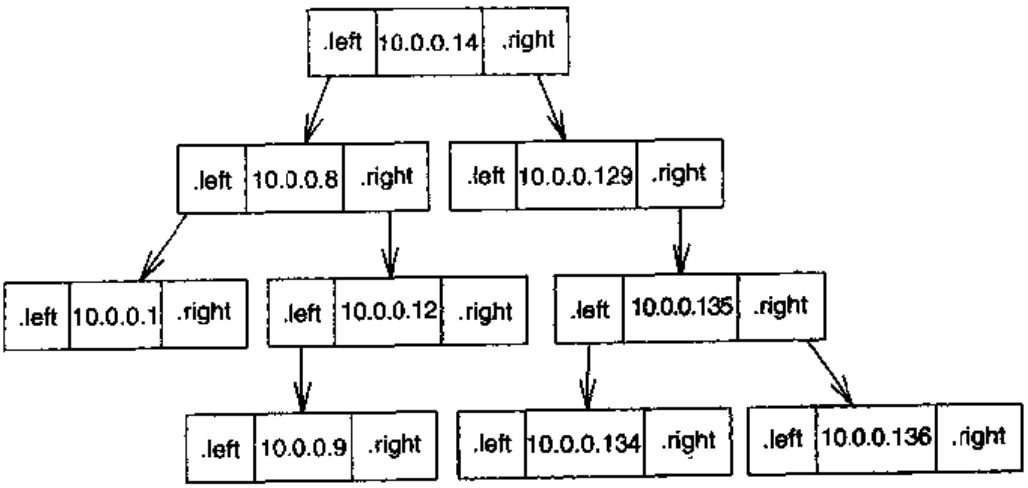


图 4.14　名称服务器中存储 Internet 地址的二叉树

树形数据结构的递归定义使其适用于递归算法。由于这些算法比我们之前看到的用于处理链表的算法要复杂得多(4.7 节)，所以，在编写函数时，一般将它们定义为取另外的函数为参数，由作为参数的函数指定在数据结构上执行的操作(即将函数的功能作一个划分，一部分函数专门处理对树的一些操作，比如遍历；而另一部分函数作为前述函数的参数，完成对结点的操作，比如比较。这样有两个好处，其一：简化函数的复杂性，分开编写的两类函数都集中于自己的任务，有利于函数的编写、阅读与维护；其二：以对结点进行操作的函数为参数，使得操作树的函数具有通用性，相同的操作函数，用不同的结点处理函数就可以处理结点类型不同的树。　　译者注)。在支持类型参数化(type parameterization)的语言中，比如 Ada，C++和 Eiffel，使用一个模板参数可以更高效地达到相同的效果。因此，在下面的例子中，对包含 IP 地址和包含主机名的二叉树都使用了相同的树结构和处理函数。前一种情况中，比较函数 pfi_compare 比较 IP 地址的整数表达；后一种情况中，它比较字符串。下面列出的这个函数在 ppr_tree 指向的树中查找 p_user 元素。①

```
tree_t
tree_srch(tree **ppr_tree; int (*pfi_compare)(); tree_t p_user;
{
    register int i_comp;

    ENTER("tree_srch")
    if (*ppr_tree) {
        i_comp = (*pfi_compare)(p_user, (**ppr_tree).data);
```

① netbsdsrc/usr.sbin/named/named/tree.c:98-129

```
        if (i_comp > 0)
            return (tree_srch(&(**ppr_tree).right, pfi_compare, p_user))
        if (i_comp < 0)
            return (tree_srch(&(**ppr_tree).left, pfi_compare, p_user))
        /*  not higher, not lower... this must be the one.
          */
        return ((**ppr_tree).data)
    }
    /*  grounded. NOT found.
      */
    return (NULL)
}
```

函数所做的就是将元素与当前的树元素进行比较，递归地查找树的左结点或右结点，找到元素时返回该结点。第一个 if 检查树是否为空，确保函数能够结束。

针对树的另一种常见操作是系统性地访问树中的所有元素。这种遍历一般以递归的方式进行编码，并且经常使用一个函数来指定在每个结点上执行的操作，从而将这个过程参数化。下面的例子中[①]，tree_trav 对每个树结点调用函数 pfi_uar。它首先递归地访问左结点，调用 pfi_uar，然后递归地访问右结点。和往常一样，当树为空时，早期的检查会立即退出函数。

```
int
tree_trav(tree **ppr_tree; int (*pfi_uar)();)
{
    [...]
    if (!*ppr_tree)
        return(TRUE)
    if (!tree_trav(&(**ppr_tree).left, pfi_uar))
        return (FALSE)
    if (!(*pfi_uar)((**ppr_tree).data))
        return (FALSE)
    if (!tree_trav(&(**ppr_tree).right, pfi_uar))
        return (FALSE)
    return (TRUE)
}
```

我们不提供往树中添加和移除元素的任何例子，因为在当前的实践中有太多的变体。各种树的实现中，添加和移除元素的代码从根本不存在(许多树常常只增长，从不将元素从树中移除)，到普通，到巴洛克风格(涉及保持树平衡的算法，用以保持算法的效率)。要记住，大多数情况下树的构建要遵循特定的规则，这些规则建立在树中元素的次序之上，直接修改元素的数据可能会致使整个树变为无效。作用在这类树上的算法可能返回不正确的结果，或者进一步破坏树的次序。

树的另一种常见应用是表达各种语言的句法结构。许多信息处理任务形式上都要用语

① netbsdsrc/usr.sbin/named/named/tree.c:164-181

言(language)来说明。这种语言不需要是一种全功能的通用编程语言。命令行界面、算术表达式、宏语言、配置文件、数据交换标准和许多文件格式经常使用相同的标准技术来说明和处理。在处理过程中,需要对一系列的元素(语言的标记)进行分析,得出它们相互之间的关系。之后,将这种关系(一般用"语法(grammar)"来描述)表达为分析树(parse tree)的形式。

为了避免基于模糊语言的例子所带来的复杂性,我们将演示 C 程序验证器 lint 如何将 C 表达式存储为树的形式。为 C 表达式定义的树形结构,是我们熟悉的递归定义①。

```
typedef struct tnode {
    op_t tn_op; /*  operator */
    [...]
    union {
        struct {
            struct tnode *_tn_left; /*  (left) operand */
            struct tnode *_tn_right; /*  right operand */
        } tn_s;
        sym_t  *_tn_sym;   /*  symbol if op == NAME */
        val_t  *_tn_val;   /*  value if op == CON */
        strg_t *_tn_strg; /*  string if op == STRING */
    } tn_u;
} tnode_t;
```

表达式被表达成一系列的结点,运算符存储在 tn_op 中,左操作数和右操作数分别存储在_tn_left 和_tn_right。② 其他元素(例如,符号、常量和字符串),由不同的树元素指向它们。与二叉树相比,树结点的放置并非按照运算符的固定次序,而是按照 lint 对源代码中表达式的分析方式。因此,表达式③

```
xp -> flags & F_NEEDARG && ! (ip -> arg = *++*argvp)
```

将会表达成图 4.15 中所示的分析树。和我们的预期相同,分析树也经常使用递归算法来进行处理。分析树的生成过程中,经常使用递归下降分析器(recursive descent parser)④递归地识别语言的各个部分,或者使用专门的分析器生成程序⑤,如 yacc。

练习 4.22　将您 e-mail 地址薄中的主机名绘制成一个二叉树。添加主机名时要按照随机的次序,不要按照字母顺序。系统性地周游这棵树,导出主机名的有序清单。构造一棵新的树,将主机名以它们出现在清单中的次序加入到树中。评论在这两棵树中查找数据的效率。

① netbsdsrc/usr. bin/xlint/lint1/lint1. h:265-280
② 注意,ANSI C 禁止用户程序中的标识符以下划线开头。
③ netbsdsrc/bin/stty/key. c:135
④ netbsdsrc/bin/expr/expr. c
⑤ netbsdsrc/usr. bin/xlint/xlint1/cgram. y

练习 4.23 从本书配套盘中找出 AVL 树的实现，阅读算法书籍中有关 AVL 树的介绍并跟随它的相应操作。

练习 4.24 从本书配套盘中找出其他生成分析树的例子。针对具体的输入数据，绘制一个具有代表性的树。

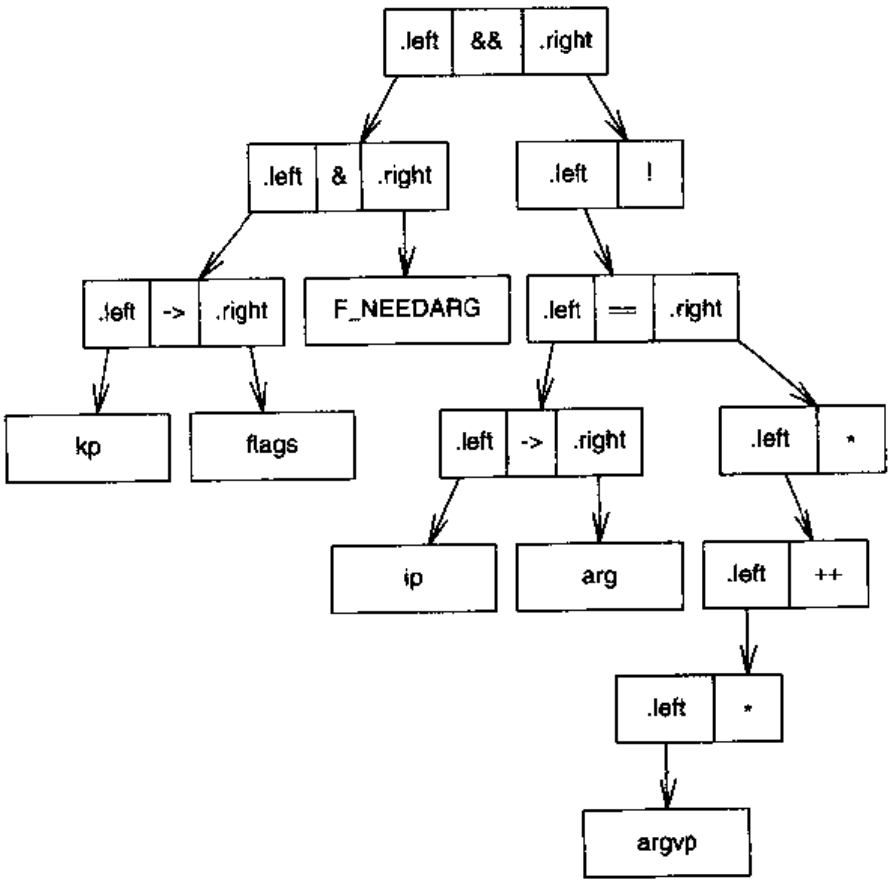


图 4.15 lint 生成的语法分析树

4.9 图

图(graph)是由边连接起来的一系列的顶点(或称结点)所组成。这种定义极为宽泛，包括树(没有回路的连通图)、集合(没有边的图)和链表(有且仅有一个边通向每个顶点的连通图)等数据组织结构。本节中，我们将分析那些不能归入上述分类的情况。遗憾的是，图的普遍性与应用程序对它广泛多样的需求，为图的存储与操作提供了大量令人迷惑的选项。虽然从中抽取出几个具有代表性的应用模式并非完全不可能，但是，通过在几个设计轴中找出图所处的位置，可以对各种类型的图进行分析。为此，我们要回答下列问题：

- 结点如何存储?
- 边如何表达?
- 边如何存储?
- 图的属性有哪些?

- “图”真正表示的是什么样的结构？

我们将依次分析每个问题。

4.9.1　结点存储

对图进行处理的算法，需要一种可靠的方式访问其中的所有结点。不同于链表和树，图中的结点不一定通过边联接起来；即使联接起来，图结构中的回路(cycle)可能致使沿着边进行系统性遍历难以实现。为此，经常要使用一种外部数据结构，将图的结点作为集合来存储与遍历。二种最常使用的方案是，将所有的结点存入一个数组或将它们链接成链表，就如同 Unix tsort(拓扑排序)程序中结点的存储方式。①

```
struct node_str {
    NODE **n_prevp;      /*  pointer to previous node's n_next */
    NODE *n_next;        /*  next node in graph */
    NODE **n_arcs;       /*  array of arcs to other nodes */
    [...]
    char n_name[1];      /*  name of this node */
};
```

在上面的例子中，结点使用 n_prevp 和 n_next 字段链接成一个双向链表。在链表和数组这两种表达方式中，可以使用各种方法将边叠加到存储结构之上，我们在接下来的章节中将概括这些方法。少数情况下，确实会使用边来表达和访问图的结点集。这种情况下，单个结点用作遍历所有图结点的起始点。最后，这两种表达方法有时会混合使用，有时还会使用其他的数据结构。作为例子，接下来我们将分析用在网络包捕获库 libpcap 中的图结构。这个库被诸如 tcpdump 等程序使用，用于分析网络上传输的数据包。需要捕获的包用一系列的控制块来指定，这些控制块就是图的结点。为了优化包捕获的规格说明，程序将控制块构成的图映射到一个树形结构中；每层树结点由结点的链表来表示，链表放置在数组中与层相关的位置(即数组中每个元素均为指向链表的指针，数组中不同的元素对应树中不同的层，每个链表存储对应层中的所有树结点。——译者注)。每个控制块还包含一个由到达它的边组成的链表；每条边会指明它们链接的控制块。②

```
struct edge {
    int id;
    int code;
    uset edom;
    struct block *succ;
    struct block *pred;
    struct edge *next; /*  link list of incoming edges for a node */
};

struct block {
```

① netbsdsrc/usr.bin/tsort/tsort.c:88-97

② netbsdsrc/lib/libpcap/gencode.h:96-127

```
        int id;
        struct slist * stmts; /*  side effect stmts */
        [...]
        struct block * link; /*  link field used by optimizer */
        uset dom;
        uset closure;
        struct edge * in_edges;
        [...]
    };
```

图 4.16 例举了这种数据结构具有代表性的一个快照。代码示例中，控制块使用 link 字段链接成一个链表，每个控制块的边放置在 in_edges 字段中；这些边通过 edge next 字段链接成一个链表，同时，边所联结的控制块由 succ 和 pred 字段指定。每个控制块链表都从 levels 数组中不同的元素开始。使用这种表达方式的情况下，对所有图结点的遍历可以编写为数组循环中对链表的遍历。①

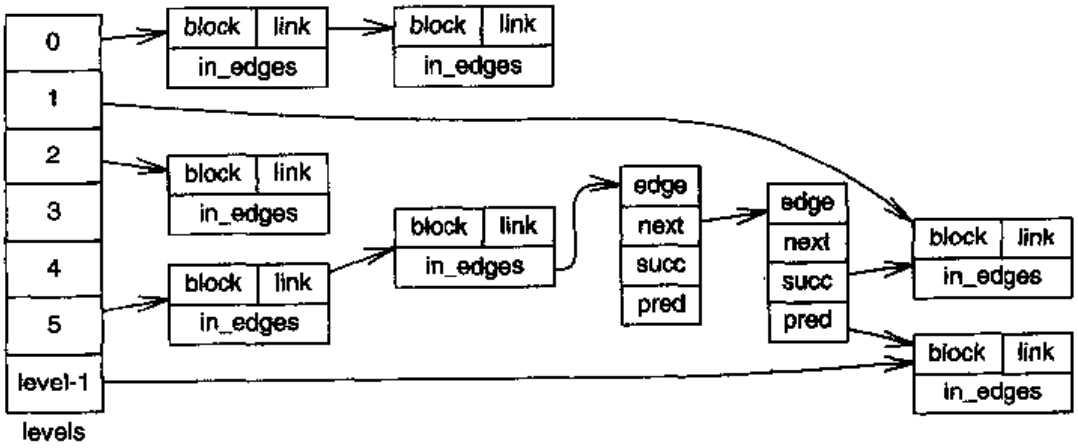


图 4.16 通过链表数组访问图的结点

```
struct block * * levels;
    [...]
        int i;
        struct block * b;
        [...]
        for (i = root -> level; i >= 0; -- i) {
            for (b = levels[i]; b; b = b -> link) {
                SET_INSERT(b -> dom, b -> id);
```

4.9.2 边的表示

图中的边一般不是隐式地通过指针，就是显式地作为独立的结构来表示。在隐式模型中，只是简单地将边表示为从一个结点指向另一个结点的指针。4.9.1 节中我们列出的 tsort 程序就是使用这个模型来表示其结点。在图的每个结点中，数组 n_arcs 存储的是指向

① netbsdsrc/lib/libpcap/optimize.c:150,255-273

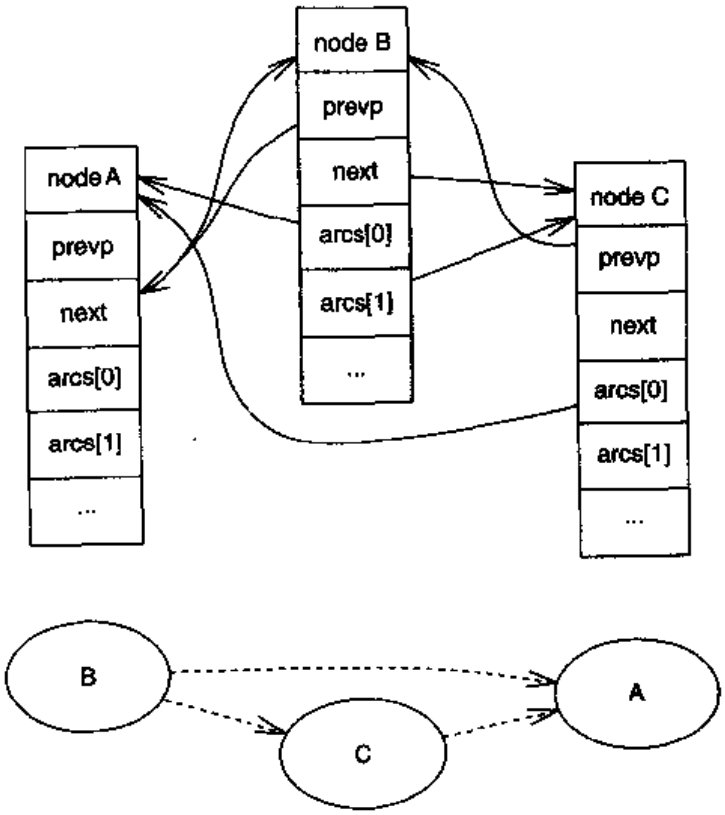


图 4.17　3 个结点构成的图以及它在 tsort 中的表达方式

其他结点的指针，这些结点均为与该结点连接的结点。在图 4.17 中您可以看到一个由 3 个结点构成的图，以及将它的边表示为指针的方式。

许多情况下，需要将图的边显式地表示出来，以便在其中存储附加信息。请考虑程序 gprof——用来分析程序在运行期间的行为。（在 10.8 节中可以找到 gprof 运作的细节。）这类信息的一个重要组成部分是程序的调用图（call graph）：程序中函数互相调用的方式。在 gprof 中用图来表示这些调用信息；图的边（在 gprof 的术语中称为弧）用来存储其他相关边的信息[①]。

```
struct arcstruct {
  struct nl          * arc_parentp;     /*  pointer to parent's nl entry * /
  struct nl          * arc_childp;      /*  pointer to child's nl entry * /
  long               arc_count;         /*  num calls from parent to child * /
  double             arc_time;          /*  time inherited along arc * /
  double             arc_childtime;     /*  childtime inherited along arc * /
  struct arcstruct   *arc_parentlist;   /*  parents- of- this- child list * /
  struct arcstruct   *arc_childlist;    /*  children- of- this- parent list * /
  struct arcstruct   *arc_next;         /*  list of arcs on cycle * /
  unsigned short     arc_cyclecnt;      /*  num cycles involved in * /
  unsigned short     arc_flags;         /*  see below * /
};
```

① netbsdsrc/usr.bin/gprof/gprof.h:110-121

每个弧将两个结点(用 struct nl 来表示)联结起来,用以保存从父亲到儿子的调用关系,如 main 调用 printf。这个关系用指向相应的父结点(arc_parentp)和子结点(arc_childp)的弧来表示。给定一个弧,程序需要找出与该结点的父亲(调用者)和儿子(被调用者)相关的其他存储类似关系的弧。(注意,在此用父亲和儿子,而不是双亲和子女,因为无论是调用者还是被调用者,都是单数,用双亲和子女会造成理解上的偏差。——译者注。)这类弧之间的链接以链表的形式存储在 arc_parentlist 和 arc_childlist 中。为了说明这种结构,我们可以分析如何表达一段小型的程序。考虑下面的代码:①

```
int
main(int argc; char * * argv)
{
[...]
        usage();
[...]
        fprintf(stderr, "%s\n", asctime(tm));
[...]
exit(EXIT_SUCCESS);
}

void
usage()
{
    (void) fprintf(stderr, "%s%s%s%s",
        "Usage: at [-q x] [-f file] [-m] time\n",[...]
    exit(EXIT_FAILURE);
}
```

图 4.18 图解出上述代码的调用图;其中的箭头表示函数调用。图 4.19 是 gprof 中相应

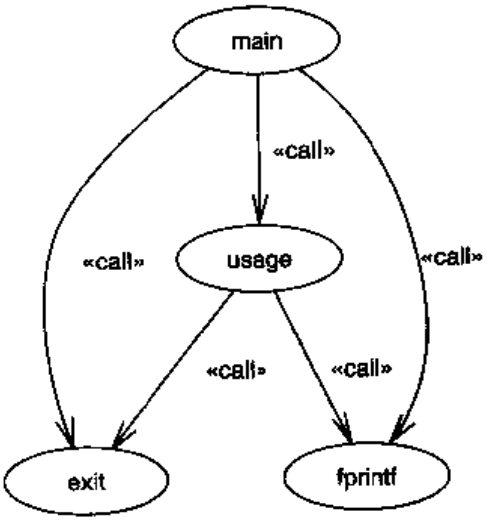


图 4.18 一个简单的调用图

① netbsdsrc/usr. bin/at(编者注:取自 at. c 和 panic. c)

的数据结构。在此，图 4.18 中的箭头用 arcstruct 记录的 parent 和 child 字段来表示；这些记录还链接成链表，表示同一儿子和父亲的多个父亲和儿子。为了简化我们的演示，我们没有讨论结点中指向边的指针。如果您觉得这种复杂程度有些难以接受……无可非议，可以试着只跟踪某几个调用者-被调用者关系，以及我们在图中列举出的某个弧链表。另外，我们分析的数据结构，就我们的观点，已经接近人类大脑所能够总体理解的复杂结构的极限。更为复杂的结构应该，一般也的确如此，拆分成独立的实体，每个实体都独立地起作用。

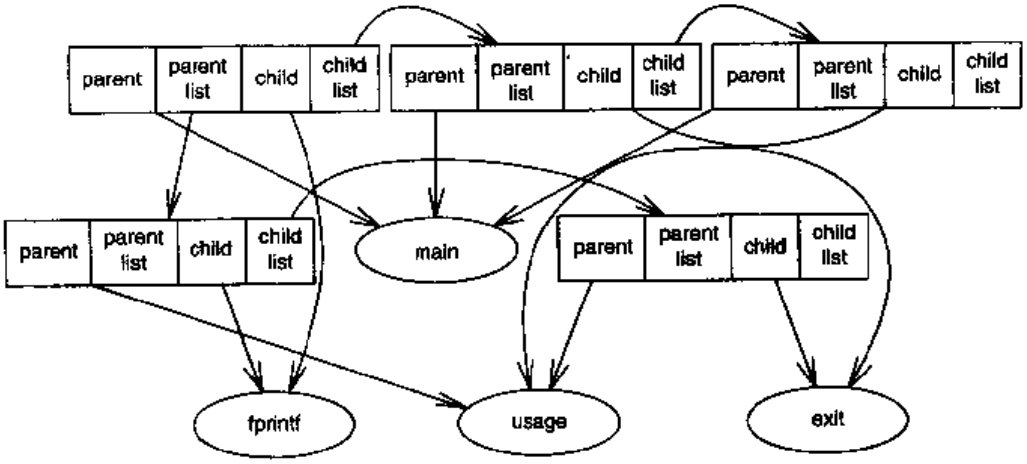


图 4.19　以 gprof 数据结构表示的调用图

4.9.3　边的存储

图的边一般使用两种不同的方法来存储：作为每个结点结构中的数组（图 4.17 例举的 tsort 即为如此），或作为锚定在每个图结点上的链表（一种在定义结点的结构中声明一个数组，一种在定义结点的结构中声明一个指向链表元素的指针，也就是链表的头部。“锚定”的意思是指通过存储在结点中的一个变量，可以访问到整个链表，就好像找到船的锚，就能够沿着铁链找到船一样。——译者注）。大多数情况下，图在生命期内都会更改自身的结构，因此这个数组一般需要动态分配，并通过存储在结点中的指针来使用（更改自身的结构是指图内的边或结点发生变化，比如增加一条边、删除一条边等，因此，存储边的数组不是固定大小，故而要动态分配。——译者注）。对于将图的边存储为链表，锚定在结点上的情况，我们可以参考 make 程序存储元素之间依赖关系的方式。

下面，我们再次分析 make 程序使用的结点的完整定义。每个结点（一般是程序编译配置的一部分）依赖于其他结点（它的子女），同时被用在其他结点编译中（它的双亲）。在结点结构中，这二者都存储为链表。①

```
typedef struct GNode {
    [...]
    Lst     parents;        /* Nodes that depend on this one */
    Lst     children;       /* Nodes on which this one depends */
    [...]
```

① netbsdsrc/usr.bin/make/make.h:112-165

```
} GNode;
```

现在，请考虑下面这个简单的 Makefile(make 程序的编译说明)所表达的依赖关系，这个文件用于在 Windows NT 下编译 patch 程序。①

```
OBJS = [...] util.obj version.obj
HDRS = EXTERN.h INTERN.h [...]

patch.exe: $ (OBJS)
    $ (CC) $ (OBJS) $ (LIBS) - o $ @  $ (LDFLAGS)

util.obj: $ (HDRS)
version.obj: $ (HDRS)
```

图 4.20 描绘了 Makefile 所表达的依赖关系(patch.exe 依赖于 util.obj，util.obj 依赖于 EXTERN.h，依次类推)。在 make 程序生成的相应的数据结构中，每个结点都包含一个结点子女的列表(version.obj 是 patch.exe 的子女)和一个结点双亲列表(EXTERN.h

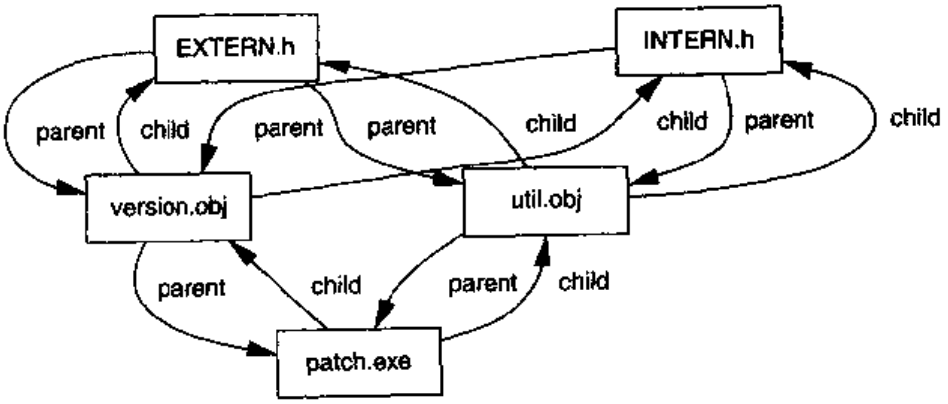


图 4.20 一个 Makefile 表达的程序依赖关系

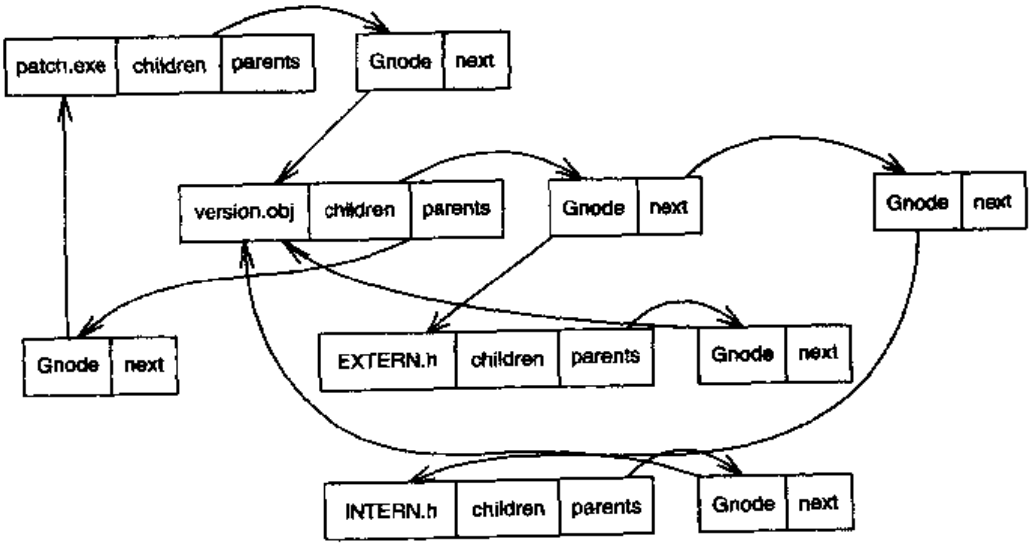


图 4.21 make 程序的数据结构中表达的程序依赖关系

① XFree86-3.3/xc/util/patch/Makefile.nt:15-40

和 INTERN. h 是 version. obj 的双亲)。所有的关系都表达在数据结构中。如图 4.21 所示,链表中的每个元素都包含一个 Gnode 指针——用来表示与给定结点有特定关系(双亲或子女)的结点;一个 next 指针——用来链接到拥有相同关系的其他边。

4.9.4　图的属性

图的一些属性很重要:它们能够引导我们如何阅读与图相关的代码。在有向图中,边(在这种情况下,经常称为弧)与从一个结点到另一个结点的方向关联起来。这项属性影响到图中元素的表达、添加和处理。在无向图中(与边交叉的结点没有方向或属性),表达数据时应该将所有的结点看作是等同的,类似地,进行处理任务的代码也不应该基于它们的方向来区分边。

在连通图(connected graph)中,任意两个结点之间总存在一条路径,在包含回路的图中,连接两个结点的路径可能不止一条。所有的属性都会影响到图的遍历方式。在非连通图中,执行遍历代码应该能够接通孤立的子图。处理包含回路的图时,遍历代码应该避免在处理图的回路时进入循环。避免回路一般通过将结点标记为已访问,以避免后续的访问。①

```
/*
  * avoid infinite loops and ignore portions of the graph known
  * to be acyclic
  */
if (from -> n_flags & (NF_NODEST|NF_MARK|NF_ACYCLIC))
    return (0);
from -> n_flags |= NF_MARK;
```

4.9.5　隐含结构

许多称为"图"的数据结构,有时远比它们的名称给出的暗示要复杂。由于没有具体的抽象数据类型或确定的准则来指明如何表达一个图,因此在程序中经常用单一的数据结构来表达许多不同的实体,彼此之间互有重叠。这种情况的标志信号就是维护多个边结构。请再次考虑 make 程序中结点的完整定义。②

```
typedef struct GNode {
    [...]
    Lst   cohorts;      /*  Other nodes for the :: operator */
    Lst   parents;      /*  Nodes that depend on this one */
    Lst   children;     /*  Nodes on which this one depends */
    Lst   successors;   /*  Nodes that must be made after this one */
    Lst   preds;        /*  Nodes that must be made before this one */
    [...]
} GNode;
```

让结点包含 parent 和 children Gnode 链表,来表达两个方向的依赖关系是合理的。但

① netbsdsrc/usr. bin/tsort/tsort. c:397-403

② netbsdsrc/usr. bin/make/make. h:112-165

是，我们看到，这个图中的每个结点还包含 successors，preds 和 cohorts 链表！显然，在这个结点的依赖关系图中，还有另一个图要引起我们的注意。

4.9.6 其他表示方法

在我们分析过的图的表达方式之外，当然还存在其他的表达方式。例如，计算机科学文献中，还有使用二维数组表示图的例子。数组元素 A(m,n)表示一个从顶点 m 到顶点 n 的边。但在实际的软件产品中，很少使用这种表示方法；或许动态调整二维数组的大小十分困难，使得这种表达方式不那么理想。

练习 4.25 在本书配套盘中，或者您的操作环境中，找出 3 例使用图结构的代码。分析结点和边的存储与表达方式，概括图的重要属性，并绘制一个框图来说明图的结构是如何组织的。

进 阶 读 物

对于我们所论述的这些内容，权威的参考是 Knuth 的 *The Art of Computer Programming* 系列，特别是第 1 卷[Knu97]第 2 章——介绍信息的构成，以及第 3 卷[Knu98]第 6 章——论述查找技术，如散列算法。Aho 等人的著作[AHU74]从理论的角度，介绍了我们分析过的许多结构的性能，具体的算法(用 C 语言)可以在 Sedgewick[Sed97,Sed01]中找到。要注意，有关数据结构效率的理论预期并不总是和实践一致。例如，已经证实，散列数据结构或简单的二叉树，在实际应用中经常比更为复杂的自平衡树结构更有效率[WZH01]。

在对系统的各种非法使用中，缓冲区溢出攻击是一个持续的威胁。早期的这类攻击是 1988 年的 Internet 蠕虫[ER89,Spa89]。许多编译期间、运行期间和基于库的防范缓冲区溢出问题的方法已经开发出来；Baratloo 等人的著作[BTS00]中描述了最近的一种方案。不对称区间以及它们如何有助于处理漏掉一个的错误，在 Koenig[Koe88,pp. 36-46]中进行了极具说服力的描述。

有时，您会遇到没有任何冲突或空间浪费的静态散列映射。用来生成这种映射的散列函数称为理想散列函数(perfect hash function)。这类函数往往出自专注的、极具创造力的或无聊的程序员之手；也可以使用 gperf 程序自动创建这类函数[Sch90]。

第5章　高级控制流程

由于技术上的原因，导弹的弹头必须倒转过来存放，即顶部在下，底部在上。为了使哪一端是底部，哪一端是顶部不再有争议，每个弹头的底部会立即标记上 TOP(顶部)。

——英国海军部条例

在第 2 章中，我们分析了大量能够影响程序指令序列的控制流程(control flow)的语句。虽然我们讲述的这些控制语句已经能够满足大多数常见的编程任务，并且也是最常遇到的语句，然而，一些并不常见的部分对许多应用程序也很重要。递归代码(recursive code)经常用相似的定义来反映数据结构或算法。异常(exception)在 C++和 Java 中用来组织对错误的处理。通过使用软件或硬件的并行性(parallelism)，程序可以增强响应性、有条理地分配工作，或者有效地使用多处理器的计算机。当并行机制不可用时，程序可能必须采用异步信号(asynchronous signal)(能够在任意时间发出的信号)和非局部跳转(nonlocal jump)来响应外部事件。最后，为了提高效率，程序员往往在平常调用 C 函数的地方，使用 C 语言预处理器(C preprocessor)的宏替换(macro substitution)功能。

5.1　递　　归

许多数据结构——如树和堆，操作——如类型推断(type inference)和类型合一(unification)，数学实体——如斐波纳契数和分形图，以及算法——如快速排序、树遍历和递归下降分析，都采用递归定义。实体和操作的递归定义用它自身来定义它的对象。虽然这些定义乍看起来好像是无限循环，但实际上并非如此，这是因为基准范例的定义(base case definition)一般会定义一个特例，它不依赖于递归定义。例如，虽然整数 n 的阶乘 $n!$，可以定义为 $n(n\text{-}1)!$，我们还定义一个基准范例 $0!=1$。

采用递归定义的算法和数据结构经常用递归的函数定义来实现。虽然函数的递归调用只不过是我们在 2.2 节中分析的函数调用的一个特例，但它们的编写方式，以及它们的阅读方式都有很大的不同，值得特殊对待。

以对外壳命令(shell command)做句法分析为例。Unix Bourne 外壳支持的命令结构相当复杂。图 5.1 给出了外壳命令语法中一个有代表性的部分；冒号前面是命令类型，随后在缩进行中给出的规则定义了该命令类型。请注意其中多少命令是以多种方式递归定义的：流水线(pipeline)可以是一个命令(command)，也可以是一个流水线后跟管道符(pipe symbol)|，再跟一个命令。命令可以是由圆括号括起来的命令列表(command-list)，命令列表可以是一个与或式，与或式可以是流水线，而流水线又可以是命令。对于这样的语法，我们一般希望首先对命令进行识别、将它们存入适当的数据结构，并处理该数据结构。对所有这些任务来说，递归都是一个关键的工具。

单个的外壳命令在计算机内部存储为一个树。由于各种命令类型可能由不同的元素组

simple-command:
item
simple-command item
command:
simple-command
(*command-list*)
{ *command-list* }
for *name* do *command-list* done
for *name* in *word* ... do *command-list* done
while *command-list* do *command-list* done
until *command-list* do *command-list* done
case *word* in *case-part* ... esac
if *command-list* then *command-list* else *else-part* fi
pipeline:
command
pipeline | *command*
andor:
pipeline
andor && *pipeline*
andor || *pipeline*
command-list:
andor
command-list ;
command-list &
command-list ; *andor*
command-list & *andor*

图 5.1 Bourne 外壳命令语法

成,因此,所有树结点都用一个所有可能元素的共用体来表示①②。

```
union node {
    int type;
    struct nbinary nbinary;
    struct ncmd ncmd;
    struct npipe npipe;
    struct nif nif;
    struct nfor nfor;
    struct ncase ncase;
    struct nclist nclist;
    [...]
};
```

① netbsdsrc/bin/sh/nodetype:用作自动生成结点定义的输入文件。
② 要注意,同一名称既用作结构的标记,又用作共用体的成员。

每个元素类型递归地定义为一个结构，这个结构中包含指针，指向适合于给定类型的命令树结点。

```
struct nif {
    int type;
    union node * test;
    union node * ifpart;
    union node * elsepart;
};
```

请注意，在上面的代码中，union 和对应的 struct 共享公共的 int type 元素。这样，各个结构都能够用对应的同名结构字段，访问共用体的 type 字段。这种做法不太好，因为 struct 和 union 字段间布局的对应是由程序员手动地维护，未经编译器或运行系统的检验。如果任何一个的布局发生变化，程序依旧可以编译通过并运行，但却可能会以不可预期的方式失败。另外，依照我们对 C 标准的解释，两种布局之间的相关，只有在通过指针对结构的第一个元素取值时，才能够得到保证。

```
STATIC void
cmdtxt(union node *n)
{
    union node *np;
    struct nodelist *lp;

    if (n == NULL)                          1 基准范例：如果是空命令，则返回
        return;
    switch (n->type) {                      2 命令 1；命令 2
    case NSEMI:
        cmdtxt(n->nbinary.ch1);             2 递归地打印第一条命令
        cmdputs("; ");
        cmdtxt(n->nbinary.ch2);             2 递归地打印第二条命令
        break;

                                            3 命令 1 && 命令 2
    case NAND:
        cmdtxt(n->nbinary.ch1);
        cmdputs(" && ");
        cmdtxt(n->nbinary.ch2);
        break;

    /* ... */                               4 命令流水线
    case NPIPE:                             4 递归地打印每条命令
        for (lp = n->npipe.cmdlist ; lp ; lp = lp->next) {
            cmdtxt(lp->n);
            if (lp->next)
                cmdputs(" | ");
        }
        break;

    /* ... */
    }
}
```

图 5.2　递归地打印分析命令树

图 5.2[①] 是用来打印 Unix 外壳命令的代码，它响应外壳的内建命令 jobs，在后台运行。cmdtxt 函数只有一个参数——一个指针，指向要显示的命令树。cmdtxt 函数中第一个测试（图 5.2：1）极为重要：它确保 NULL 指针表示的命令（例如，if 命令的空 else 部分）也能够得到正确的处理。请注意，空命令的处理并不涉及递归；cmdtxt 函数简单地返回。由于将命

① netbsdsrc/bin/sh/jobs.c:959-1082

令表示为树时，所有的树叶(结点，terminal node)都为 NULL，每个树分支(tree branch)递归向下最终都会到达一个树叶并终止。这项基准范例测试和它的非递归定义保证递归函数能够最后终止。上面这种非正式的论证对理解涉及递归的代码来说，是一种重要手段。理解了定义中用到自身的函数如何能够工作之后，函数的其他代码也就会变得清清楚楚。每种不同的非空命令类型(结点)对应一个 case 语句(图 5.2:4)，根据命令的组成部分将命令打印出来。例如，由分号分开的命令列表在打印时，首先打印出列表的第一个命令(图 5.2:2)，然后是一个分号，接下来是第二个命令(图 5.2:3)。

我们在图 5.1 中看到的递归语法定义，还适用于一系列遵循这种语法结构的函数——递归下降分析器(recursive descent parser)。这种方法经常用于分析相对简单的结构，比如命令专有或领域专有的语言、表达式、或基于文本的数据文件。图 5.3[①] 是用于分析外壳命令的代码中具有代表性的一些函数。分析代码与语法之间存在对称性；这种对称性甚至外延到标识符的命名，虽然该分析器(parser)是 Kenneth Almquist 在 Stephen Bourne 实现最初的外壳并编制了外壳的语法之后，十多年后编写的。递归下降分析器的结构十分简单。每个函数负责分析一种特定的语法元素(命令列表、与或式、流水线、或单个命令)，并返回与该元素一致的结点。所有函数都通过调用其他函数来分析组成给定命令的语法元素，用 readtoken()函数来读取纯标记，如 if 和 while。

打印命令结点并对命令进行分析的这些函数严格地说并非递归，这是由于函数体中没有语句直接调用包含这些语句的函数。然而，这些函数调用了其他的函数，其他的函数又调用了另外的函数，它们有可能会调用最初的函数。这种类型的递归称为互递归(mutual recursion)。虽然基于互递归的代码可能看起来比简单的递归函数更难以理解，但是人们还是能够根据递归定义的底层概念容易地推断出这种类型的递归。

如果对函数的所有递归调用恰恰发生在函数返回点之前，我们称该函数是尾递归(tail recursive)。尾递归是一个重要的概念，因为编译器可以将这些调用优化成简单的跳转，因而节省函数调用在时间和内存上的开销。尾递归调用等同于一个回到函数开始处的循环。程序中有时会用尾递归来替代常规循环、goto 或 continue 语句。图 5.4[②] 中的代码就是一个具有代表性的例子。这段代码的目的是定位一个多用户游戏 hunt 的服务器(在源代码中称为 driver)。如果没有找到任何服务器，代码则试图启动一个新的服务器，等待服务器注册，然后使用尾递归调用再次执行服务器定位过程。

练习 5.1 用来推理 cmdtxt 函数终止的非正式论证只是一种非正式论证。请提供一个能够导致 cmdtxt 进入无限递归的数据结构。为了阻止这种情况发生，需要采取什么额外的保护措施？

练习 5.2 基于 10.5 节中提供的例子，构造一个简单的工具，定位递归函数的定义。在提供的源代码上运行它，推理 3 种不同类型的递归函数的工作方式。

① netbsdsrc/bin/sh/jobs.c:157 513

② netbsdsrc/games/hunt/hunt/hunt.c:553-619

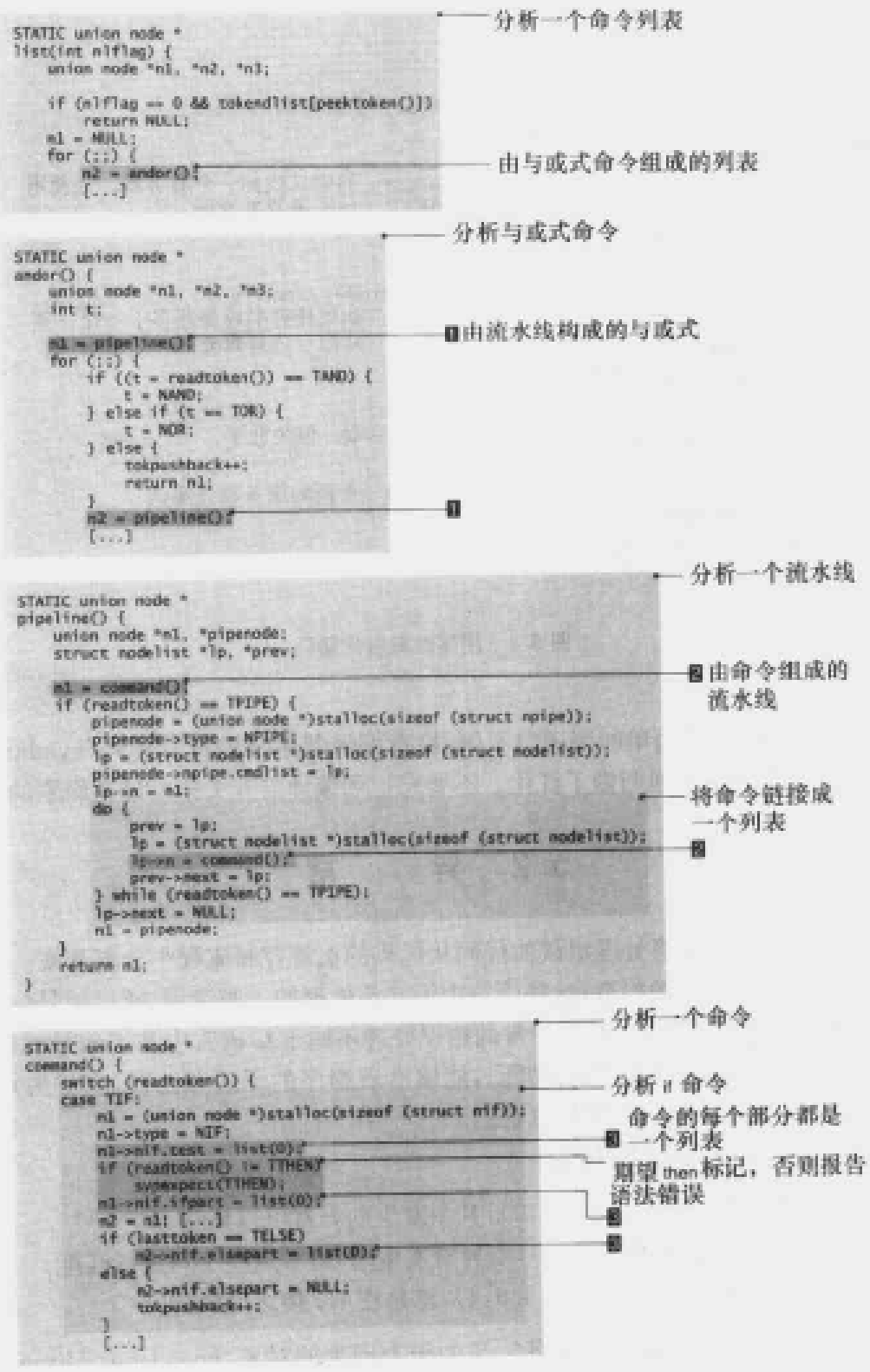


图 5.3　用于外壳命令的递归下降分析器

练习 5.3　根据 expr 命令用来计算表达式的递归下降分析器，构造可以为 expr 命令所接受的语法[①]。

① netbsdsrc/bin/expr/expr.c:240-525

图 5.4 用尾递归替代循环

练习 5.4 编译一个简单的尾递归示例，检查编译器生成的符号代码(symbolic code)，看看您的编译器是否对尾递归做了优化。不要忘记用编译器开关指定最高级别的优化。

5.2 异 常

异常机制允许程序员将处理错误的代码从代码的正常控制流程中分离开来。在 C++和 Java 程序中都会遇到类似的构造；这些语言中用异常处理的一些错误，通过信号(signal)(在 5.4 节中分析)报告给 C 程序。基于异常的错误处理不同于 C 语言中基于信号的代码，异常作为语言的一部分(而非由库提供的功能)，能够沿着程序的词法和函数(或方法)的调用栈传播，允许程序员以结构化的方式处理它们。

Java 的异常处理代码可能包括：

- try 块中的一系列语句(图 5.5:2)，其中发生的异常可以捕获。①
- 0 或多个 catch 子句(图 5.5:3)，当异常发生时，将和这些子句进行匹配。
- 一个可选的 finally 子句块(图 5.5:4)，总是在 try 块之后执行。

Java 异常都是 java.lang.Throwable 类的子类产生的对象，java.lang.Throwable 类有两个标准的子类：java.lang.Error(多用来表明不可恢复的错误)和 java.lang.Exception(用来表明可以捕获和恢复的情况)。很多时候，新的异常类都定义为上述两个类的子类。②

```
public final class LifecycleException extends Exception{
```

① jt4/catalina/src/share/org/apache/catalina/session/JDBCStore.java:572-595

② jt4/catalina/src/share/org/apache/catalina/LifecycleException.java:77

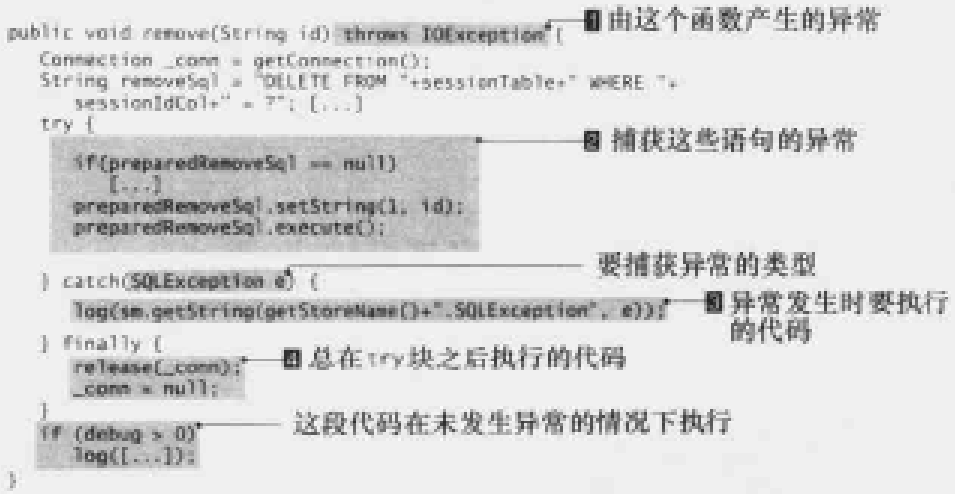


图 5.5　Java 中的异常处理

异常要么是为了响应一个错误条件而显式地生成[①]

```
public void addChild(Container child){[...]
    if (children.get(child.getName())!= null)
      throw new IllegalArgumentException (
"addChild:Child name'" + child.getName()+ "'is not unique");
```

要么是调用了可以触发异常的方法，从而隐式地生成。这两种情况中，如果生成的异常不是 Error 或 RuntimeException 异常的子类，那么该方法必须用 throws 子句声明该异常（图 5.5:1）。图 5.5 中的 IOException 就是一个隐式生成的异常。将 throws 子句从方法的定义中移除，然后运行 Java 编译器对类的源代码进行编译，就可以容易地找到那些可能隐式地生成异常的方法。

如果一个异常没有在抛出它的方法中被捕获，那么它将会沿着方法的调用栈向上传播给方法的调用者，调用者的调用者，依次类推，直到被捕获为止。在图 5.6 的代码中，start 方法[②]调用 validatePackages 方法时，并未捕获 validatePackages 方法[③]中抛出的 LifecycleException 异常，但在 ContainerBase setLoader 方法[④]调用 start 方法时被捕获。未捕获的异常将终止程序的运行，同时报告一段诊断消息以及栈的踪迹。这种行为和含糊不清的错误报告将会使毫无防范的用户感到一头雾水，所以我们应该确保程序检查并恰当地处理了所有可能产生的异常。

程序中有时会有空的 catch 子句。[⑤]

```
try{
    [...]
```

① jt4/catalina/src/share/org/apache/catalina/core/ContainerBase.java:775-781
② jt4/catalina/src/share/org/apache/catalina/loader/StandardLoader.java:583-666
③ jt4/catalina/src/share/org/apache/catalina/loader/StandardLoader.java:1237-1268
④ jt4/catalina/src/share/org/apache/catalina/core/ContainerBase.java:345-378
⑤ jt4/catalina/src/share/org/apache/catalina/loader/StandardLoader.java:1028-1044

图 5.6 异常跨方法调用的传播

```
}catch (MalformedURLException e){
}catch (IOException e){
}
```

空 catch 子句用来忽略异常，这样做可能是因为这些异常对正在执行的代码并不重要，如下面的注释所述。①

```
}catch (NamingException e){
    // Silent catch: it's valid that no /WEB-INF/lib directory
    // exists
}
```

您可能还会遇到只使用 finally 子句，而没有相应 catch 子句的情况。大多数情况下，finally 子句是用来指定 try 子句退出后要执行一次的一些处理，不管 try 子句的退出是代码执行完毕，还是通过 return 或(带标记的)break、continue 语句。例如，下面的代码中，reset(mark)调用总会得到执行——即使控制流程到了 return values 语句。②

```
try {
    if ( nextChar() == '> ' )
  return values;
} finally {
    reset(mark);
}
```

从而，这段代码可以理解为：

① jt4/catalina/src/share/org/apache/catalina/startup/ContextConfig.java:1065-1068

② jt4/jasper/src/share/org/apache/jasper/compiler/JspReader.java:606-611

```
if ( nextChar() == '>' ) {
  reset(mark);
  return values;
}
reset(mark);
```

finally 子句还经常用于清理已获得的资源，比如文件描述符、连接、锁和临时对象。在下面的代码中，不管 try 块中的 28 行语句中发生什么问题，Connection conn 都会被关闭。[①]

```
conn = getConnection();
try {
    [...]
} catch ( SQLException e ) {
    transformer.getTheLogger().error("Caught a SQLException", e);
    throw e;
} finally {
    conn.close();
    conn = null;
}
```

C++对异常的处理和 Java 十分类似。它们的不同之处如下：

- 异常对象可以是任何类型。
- 没有提供 finally 子句。
- 在栈展开时，会调用局部对象的析构函数。
- 函数使用声明符 throw 来声明抛出异常，而非 Java 的 throws 子句。

练习 5.5　找到通过异常来发出错误信号的 C++或 Java 函数调用序列，通过修改函数的返回值，重写这段代码，使之不再依赖于异常。对比这两种方法的可读性。

练习 5.6　选择一个 Java 方法，将它的 throws 子句移除后，对它进行编译，找出抛出异常的方法。对抛出异常的方法执行同样的过程，绘制出一个异常传播树。

练习 5.7　BSD 备份恢复程序的磁带处理代码[②]中，有大量显式地核对系统调用的返回值是否为-1(标志失败的返回值)的用法。请简略地说说，如何将部分代码改写为使用C++的异常。

5.3　并 行 处 理

有些程序并行地执行部分代码，以增强对环境的响应，安排工作的分配，或有效地使用多个计算机或多处理器计算机。这种程序的设计与实现属于一个不断发展的研究领域；因

① cocoon/src/java/org/apache/cocoon/transformation/SQLTransformer.java:978-1015

② netbsdsrc/sbin/restore/tape.c

此，您可能要遇到许多不同的抽象和编程模型。本节中，我们将分析常见的分布式工作模型(work distribution model)中一些具有代表性的例子，避开较为特殊的实现，如涉及细粒度并行(fine-grained parallel)或向量计算(vector computation)的操作。并行运作的能力要受到硬件层和软件层的影响。

5.3.1 硬件和软件并行性

在硬件层，可能遇到的并行处理类型包括：

(1) 同一处理器运行多个执行单元

除非正在编写编译器或汇编器，否则只有在阅读和编写针对某些处理器构架(将指令同步的责任交给程序员来处理)的符号代码时，才会处理这些操作。

(2) 智能的集成或外部设备

现代外部设备，如磁盘和磁带驱动器、图形和网络适配器、调制解调器和打印机都有自己的处理器，并能和计算机的主处理器并行地执行高级命令(例如，写磁道、绘制多边形)。针对这些设备的代码都由操作系统和设备驱动程序进行分离与抽象。

(3) 多任务的硬件支持

大多数现代处理器都提供许多特性，如中断和内存管理硬件，允许操作系统对多个任务进行调度，看起来就如同它们在并行执行。实现这项功能的代码是操作系统的一部分，我们将分析两种常用的软件抽象，进程(process)和线程(thread)，这两种抽象都用来封装并行执行的任务。

(4) 多处理器计算机

这类计算机越来越平常，从而，也促使软件的实现要能够充分利用已有的计算能力。在这种计算机上，还是使用进程和线程，帮助底层的操作系统在处理器间分配工作。通过对单处理器多任务和多处理器环境使用相同的抽象，使得我们编写的代码可以在两种不同类型的硬件构架上运行。

(5) 粗粒度分布模型(coarse-grained distributed model)

联网的计算机，越来越多地用来并行地解决那些需要大量计算资源的问题，或是提供更好的性能或可靠性。典型的应用程序包括基于多服务器的科学计算(比如：大数的因式分解，或寻找外星智慧生命)和集中式电子商务设施的运作。为这类系统编写的代码要么依赖现有的集群操作系统、中间件或应用程序，要么使用基本的联网功能针对具体应用进行编写。

在软件层，用来表示代码并行执行，或表面上并行执行的模型如下：

(1) 进程

进程是一个操作系统抽象，表示正在运行的程序的单个实例。操作系统使用底层硬件在各个进程之间切换，从而提供对硬件的更有效利用，为复杂系统提供一种结构化机制，同时还使运行环境更为灵活和响应迅速。创建、终止和切换进程都是相对费时的事件；因此，一般倾向于为进程分配大量的工作。进程间的通信与同步由操作系统提供的各种机制来处理，比如共享内存对象(shared memory objects)、信号量(semaphore)和管道(pipe)。

(2) 线程

一个进程可以由多个线程组成；并行运行的控制流程。线程可以在用户层实现，也可以由操作系统提供内在的支持。进程中的所有线程共享许多资源，最重要的是共享全局内存；

从而，线程的创建、终止和切换是轻量级的事件，一般倾向于为线程分配一些小型的工作。线程间的通信经常通过使用进程的全局内存来实现，同时使用一些同步原语(synchronization primitive)——如互斥函数，来维护一致性。

(3) 特别模型

各种现实或假想的原因都会影响能否使用我们前面描述的两种抽象，因此，出于可维护性、效率或无知等原因，一些程序自己实现了对真实或虚拟并行执行的支持。这类程序直接依赖于底层原语(primitive)，比如：中断、异步信号、轮询(polling)、非局部跳转(nonlocal jump)，以及与构架相关的栈操作。在 5.4 节和 5.5 节，我们将分析某些原语。这类代码的典型例子就是操作系统的内核，一般由它提供对进程和线程的底层支持。

许多操作系统会在多个处理器上分发进程和线程。出于此因，用于多处理器计算机的代码通常围绕进程和线程进行组织。

5.3.2　控制模型

应用并行处理的代码，一般构造为下面 3 种不同的模型：

1. 工作群模型(work crew model)，一系列相似的任务并行地运转。

2. 管理者/工人模型(boss/worker model)，一个管理者为不同的工人分配工作。

3. 流水线模型(pipeline model)，对数据进行操作并将处理后的数据传递给下一个任务的一系列任务。

在下面的段落中，我们将对每个模型进行详细地分析。

工作群模型用于将相似的操作分配给多个任务，并行地执行它们。每个任务都执行相同的代码，常常只是分配给各个任务的工作有所不同。工作群并行模型用于在多个处理器间分配工作，或者创建一个任务池，然后将大量需要处理的标准化的工作进行分配。例如，许多 Web 服务器的实现都会分配一个由进程或线程组成的工作群，处理到来的 HTTP 请求。图 5.7 展示出工作群代码的一个典型实例。① 该程序在多个窗口中绘制跳动的图形。每个窗口由不同的线程负责处理(执行 do_ico_window 函数)。由于所有的线程共享相同的全局内存池，故而创建线程的代码分配 closure 内存区域来存储线程的私有信息。在实际的工作中，虽然很可能遇到大量各种各样管理线程的接口，但大部分接口启动线程的方式都与示例中的相似：线程的启动函数指定要执行的函数和一个参数，该参数将传递给指定线程的函数实例。这个参数一般指向一段内存区域，用来存储线程相关的数据或传递针对指定线

```
main(int argc, char **argv)
{
    [...]
    /* start all but one here */
    for (i=1; i<nthreads; i++) {
        closure = (struct closure *) xalloc(sizeof(struct closure));
        xthread_fork(do_ico_window, closure);          ← 为每个窗口启动一个线程
    }
[...]                                                  ← 每个线程执行的函数

void * do_ico_window(closure)
    struct closure *closure;
{
```

图 5.7　工作群的多线程代码

① XFree86-3.3/contrib/programs/ico/ico.c:227-369

程的信息。

管理者/工人模型——也形象地称为主/从模型，由一个任务，即管理者，分配不同数量的工作给工人任务（由管理者根据需要创建）。这项技术一般用在管理者线程从用户那里接收命令，之后启动一个工人线程处理每个新任务的情况，可以很好地维护程序的响应性。应用程序经常将可能阻塞或较长时间才能完成的操作——比如保存或打印，委托给独立的线程，然后继续处理用户的命令。图 5.8[①] 中的代码创建至少 4 个不同的线程来管理 Amoeba 分布式操作系统环境中 X Window 系统的服务器与客户之间的连接。

```
param = (XAmChanDesc **) xalloc(sizeof(XAmChanDesc *));
*param = chandesc;
result = thread_newthread(XAmReaderThread, THREAD_STACK_SIZE,
                          (char *)param, sizeof(XAmChanDesc *));
[...]
if (thread_newthread(TcpIpReaderThread, MAXBUFSIZE + CONNECTOR_STACK,
                     (char *)param, sizeof(XAmChanDesc *)) == 0) {
    Error("TCPconn refused: Cannot start reader thread\n");
[...]
if (thread_newthread(AmConnectorThread,
                     CONNECTOR_STACK, 0, 0) <= 0) {
    Fatal("Cannot start Amoeba connector thread\n");
}
[...]
if (thread_newthread(AmTCPConnectorThread,
                     CONNECTOR_STACK, 0, 0) <= 0)
    Fatal("Cannot start TCP connector thread\n");
```

分配和设置阅读器线程的私有内存

启动 TCP/IP 阅读器线程

启动 TCP/IP 连接线程

启动本地的 Amoeba 服务线程

启动 TCP/IP 服务线程

图 5.8 管理者/工人的多线程代码

当使用多线程或进程时，一个重要的问题就是共享资源的处理。该问题在多线程程序中最为普遍，因为多线程程序共享它们所有的全局内存。图 5.9[②] 的函数 XrmUniqueQuark 将全局计数器 nextUniq 递减，返回一个惟一的整数。然而，如果一个线程将当前的 nextUniq 值赋给 q（图 5.9:1），在 nextUniq 递减之前，另一个线程获得了执行控制权并请求一个惟一整数，那么两个请求都将返回相同的非惟一值。通过定义一个指定对资源独占访问的互斥抽象（mutual exclusion，或 mutex），可以避免这个问题。同一时刻只能有一个线程或进程能够拥有对互斥量的访问。互斥量锁定操作尝试获得对互斥量的访问权；如果该互斥量业已被另外的任务锁定，那么，当前任务被阻塞，直到互斥量解锁为止。在一个任务完成对共享资源的访问后，它解开对互斥量的锁定，让其他任务可以使用它。因此，访问共享全局资源的代码要放在互斥量的锁定/解锁对中，如图 5.9 所示。

在单处理器的机器上，任务并发性常常通过处理器响应外部事件（称为中断）切换执行上下文来实现。中断用来通知机器，一个慢速外设需要服务（例如，磁盘块的读取已经完成或新的网络数据包到达），或一段确定的时间间隔已经过去，从而，可以将当前执行的任务切换到另一个。在不能使用互斥操作时，访问共享资源的代码可以在更低级的层面上控制并发性。下面的这段代码[③]通过关闭中断，直到函数返回为止，从而保证没有任何其他操作会

① XFree86-3.3/xc/lib/xtrans/Xtransam.c:495-1397

② XFree86-3.3/xc/lib/X11/quarks.c:401-433

③ netbsdsrc/sys/arch/arm32/ofw/ofw.c:325-327

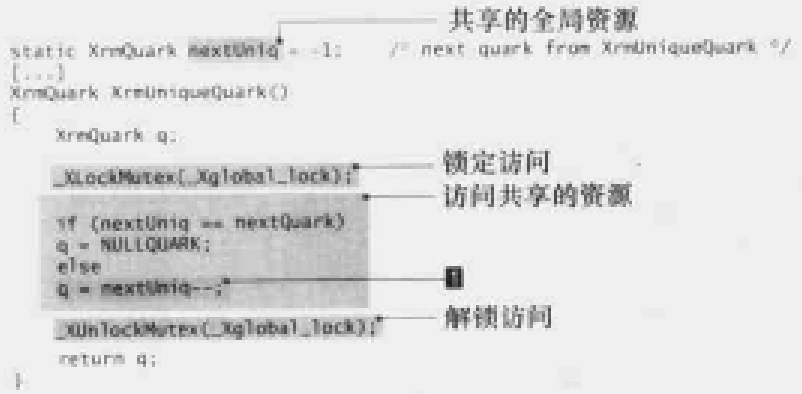


图 5.9　被互斥锁保护的代码块

与 ofw_client_services_handle 并发执行。

```
saved_irq_state = disable_interrupts(I32_bit);
ofw_result = ofw_client_services_handle(args);
(void)restore_interrupts(saved_irq_state);
```

请注意上面的代码与基于互斥量的代码之间的相似性。二者都将访问共享资源的代码，封入由限制对这些资源访问的语句所定义的块中。通过查找这种对称的结构，可以检验出大部分访问共享资源的代码。虽然不遵照这种结构并非错误，但是对于这类代码块中缺少适当锁定语句的入口点或退出点(goto 标记或 return 语句)，应该给予足够的警惕。

并行处理的管理者/工人模型还可以用来隔离系统中不同部分之间的关联。系统中定义明确的任务由不同的进程来处理，从而获得重用带来的好处，断开任务间不需要的交互。在两个进程间的通信接口较简单，并且委派的工作比较重要时，常常会应用这个模型。这种例子包括邮件的发送(在 Unix 系统中，通过调用 sendmail 进程来处理)和文档的打印。另外，需要特殊安全特权的操作经常会委托给专门负责该操作的进程，以避免让大量的代码体都隐含地拥有这些特权。另一种应用管理者/工人模型的情况是，将后台运行的操作委托给低优先级的工人进程，从而允许管理者高效地处理高优先级的请求。

图 5.10[①] 中取自 at 程序的代码就是一个典型的例子。当程序需要向一个用户发送邮件消息时，它使用 Unix 的系统调用 fork，产生自身的一个独立运行的副本。这个子副本将自身的标准输入重定向到需要发送的文件，用 sendmail 和合适的参数来替代它的实例。sendmail 会发送指定的邮件。之后，父进程等待 sendmail 进程的结束。我们概括的这个过程使用拥有标准接口、普遍存在的程序(sendmail)来处理发送邮件的任务。sendmail 的使用是如此广泛，以至于 Unix 上运行的其他可选的邮件传送代理程序，比如 smail 或 MMDF，都提供一个与 sendmail 兼容的程序来处理邮件的发送。另一种用在 Microsoft Windows 环境中的方式是使用特定的 API 来发送邮件。其他服务也同样存在这种现象：在 Unix 世界中，打印和关闭系统一般通过调用独立的程序(lpr 和 shutdown)来处理，但在 Windows 中是由单独的 API 来完成。

① netbsdsrc/libexec/atrun/atrun.c:228-239

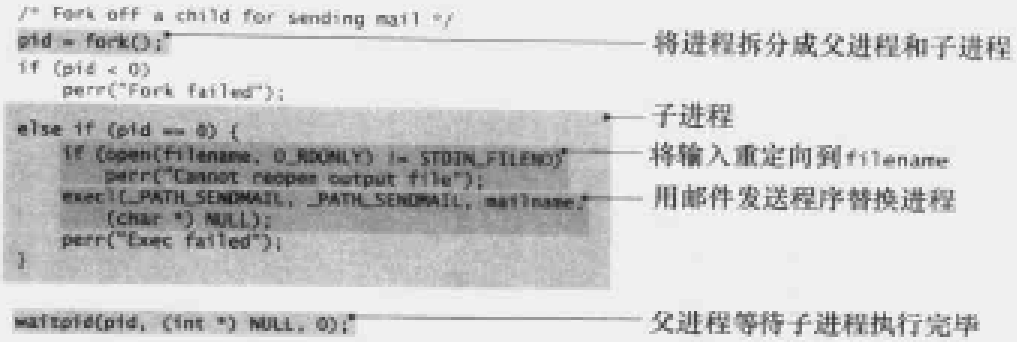


图 5.10 生成一个独立的进程来发送邮件

在分析产生独立进程的代码时，一定要验证，传递给每个进程的参数不能被恶意的用户操纵，用以欺骗进程，执行设计之外的操作。请考虑图 5.11①② 中的代码片断。和-c 选项一起传递给外壳执行的参数中，有可能包含外壳的元字符(metacharacter)，这些元字符在外壳执行由 snprintf 语句构造的命令之前被解释。例如，如果用户将反引号(backquotes)包围起来的命令 rm /etc/passwd 作为第一个参数传递给程序，外壳将会执行该命令，将倒引号中的内容替换为执行的输出。如果命令以超级用户的特权执行，系统的密码文件会被删除。

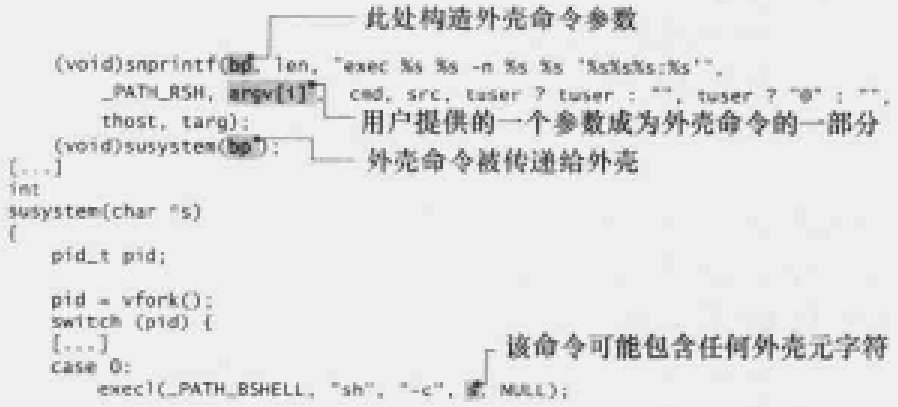


图 5.11 向外壳传递未经检验的参数引发的潜在安全漏洞

最后，流水线模型经常用在任务以串行的方式对数据进行处理的情况。每个任务都接收到一些输入，对它们进行一些处理，并将生成的输出传递给下一个任务，进行不同的处理。Unix 基于过滤器的程序(filter-based program)就是这项技术的典型应用。图 5.12③ 中的代码是一个具有代表性的实例。该代码的目标是创建一个文件，其中包含每个手册页文件的汇总信息。对于含有手册页目录中所有文件名称的文件，egrep 只从清单中匹配压缩格式的手册页。流水线的下一阶段是一个循环，它读取文件名，使用另外单独的流水线对每个文件进行处理。该流水线解压缩每个文件(gzip)，之后使用 sed 脚本从结果中提取汇总信息。

① netbsdsrc/bin/rcp/rcp.c:292-297

② netbsdsrc/bin/rcp/util.c:113-127

③ netbsdsrc/libexec/makewhatis/makewhatis.sh:31-34

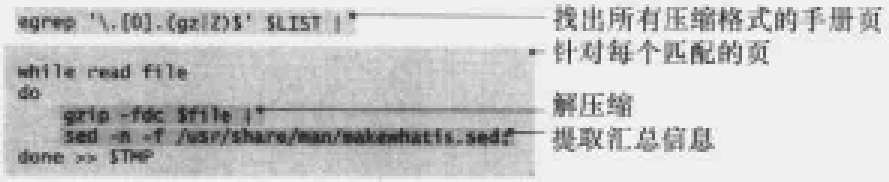


图 5.12　将压缩格式的手册页文件汇总成一个清单的流水线

5.3.3　线程的实现

您可能已经注意到，到目前为止，在我们分析过的线程示例中，许多不同的函数都可以达到相似的结果。这种现象背后的原因是：C 和 C++语言没有提供标准的线程和同步库。结果是，人们为了满足需求创建了许多轻度不兼容的库。一些软件的实现（比如 X Window 系统[①]、Perl[②③④⑤] 和 apache Web 服务器[⑥⑦⑧]）都自定义代码来实现高层的接口，以可移植的方式提供线程和互斥执行的功能。

表 5.1 展示出许多常用线程和同步原语在 3 种不同的 API 中的命名方法。另一种常见的线程 API——Unix System V，与 POSIX API 极为相似，只是将线程函数中的 pthread 替换为 thr 前缀，移除 mutex 和 cond 函数的 pthread_前缀，并使用 mutex_clear 替代 pthread_mutex_destroy。

pthread_create 和 CreateThread 函数启动一个新线程，并从作为参数传递给它的函数开始执行。Java 中线程的实现有所不同：Thread 类既是从它派生的线程类的基类，还是一个构造函数，它接受实现了 runnable 接口的类作为参数。所有的情况下，类都必须支持 run 方法，它是线程的入口点。Java 线程通过调用类的 start 方法开始执行。[⑨]

```
class LogDaemon extends Thread { [...]
  public void run() {
  while (true)
          flusher.run();
    } [...]
}
public class JasperLogger extends Logger { [...]
  private void init() { [...]
    LogDaemon logDaemon = new LogDaemon([...]);
    logDaemon.start();
```

① XFree86-3.3/xc/include/Xthreads.h
② perl/thread.h
③ perl/os2/os2thread.h
④ perl/win32/win32thread.h
⑤ perl/win32/win32thread.c
⑥ apache/src/os/netware/multithread.c
⑦ apache/src/os/win32/multithread.c
⑧ apache/src/include/multithread.h
⑨ jt4/jasper/src/share/org/apache/jasper/logging/JasperLogger.java:77-260

```
    } [...]
}
```

表 5.1 常见 POSIX、Win32 和 Java 的线程和同步原语

POSIX	Win32	Java
pthread_create	CreateThread	Thread
pthread_exit	ExitThread	stop
pthread_yield	Sleep(0)	yield
sleep	Sleep	sleep
pthread_self	GetCurrentThreadId	currentThread
pthread_mutex_init	CreateMutex	-
pthread_mutex_lock	WaitForSingleObject	synchronized[a]
pthread_mutex_unlock	RelcaseMutex	synchronized[b]
pthread_mutex_destroy	CloseHandle	-
pthread_cond_init	CreateEvent	-
pthread_cond_wait	WaitForSingleObject	wait
pthread_cond_signal	PulseEvent	notify
pthread_cond_destroy	CloseHandle	-

a:进入方法或代码块
b:退出方法或代码块

线程通过调用 pthread_exit 和 ExitThread 函数,或者对象的 stop 方法终止它们的执行。由于线程并行地执行,有时线程或许需要将控制释放给其他的线程;这项任务由 pthread_yield 和 Sleep(0)函数、yield 方法完成。对 sleep 的各种调用允许线程暂停一段给定的时间间隔。

这 3 种 API 对互斥量的处理大不相同。POSIX 和 Win32 API 提供函数来创建、锁定(等待)、解锁(释放)和销毁(关闭)互斥量(Win32 还提供一套单独的临界区域和信号量函数),而 Java 语言则通过 synchronized 关键字提供相同的功能。同一底层对象中声明为 synchronized 的对象方法永远不会并行执行。类似地,声明为 synchronized 的类方法,在执行期间将锁定整个类,而 synchronized 语句将在执行其后的语句之前,取得其参数的独占锁。①

```
synchronized (this) {
    mThread = mCurrentThread++ ;
}
```

在 Java 中,同步方法(synchronized method)还用来安排一个线程等待另一个线程。在 POSIX 中使用 pthread_cond_wait 和 pthread_cond_signal 来完成相同的任务,Win32 中则

① hsqldb/src/org/hsqldb/ServerConnection. java:70-72

使用事件(event)。Java 中,在分别对一个条件进行测试与更改时,可以使用同一线程对象的同步方法,结合对 wait 和 notify 方法的调用,达到同样的效果。图 5.13[①] 展示了这种编码的惯用法,其中,队列的 pull 方法在循环中 wait,直到接到队列的 put 方法发出的通知有元素加入为止。

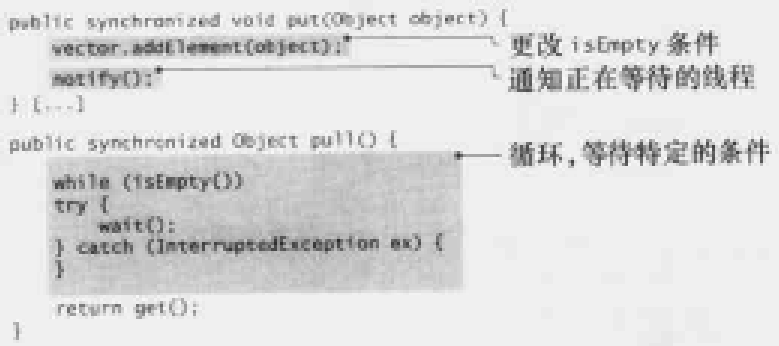


图 5.13　Java 中的线程同步

练习 5.8　概括您周围最流行的软件和硬件并行处理的类型。

练习 5.9　进程和线程间对资源的共享经常依赖于操作系统和各种软件库。针对您所在的操作环境,研究并列出在进程和线程间共享的软件抽象(文件描述符、信号、内存、互斥量)。

练习 5.10　在内核的源代码中找出调用 tsleep 的函数。[②] 说明如何用它们来支持并发性。

练习 5.11　查找调用子进程(subprocess)的代码,讨论相关的安全问题。

练习 5.12　定义一条规则,规定什么时候 Java 方法应该声明为 synchronized。找出使用该关键字的代码,并检验您的规则。

5.4　信　　号

应用程序级别的 C 程序以信号的形式接收外部异步事件。信号的例子包括:用户中断了一个进程(SIGINT),执行了非法的指令或引用了无效的内存(SIGILL,SIGSEGV),浮点异常(SIGFPE),或者一个已经停止的进程恢复运行(SIGCONT)。响应信号的代码构成信号处理器(signal handler):信号处理器是应用程序接收到给定的信号后,指定运行的一个函数。在图 5.14[③] 中,程序将进行一个无限循环,不断地向用户询问算术问题。但是,当用户输入 C,中断应用程序时,信号处理器 intr 会被调用。该处理器打印出测验的统计,然后退

① jt4/catalina/src/share/org/apache/catalina/util/Queue.java:77-93

② netbsdsrc/sys

③ netbsdsrc/games/arithmetic/arithmetic.c:115-168

出程序。

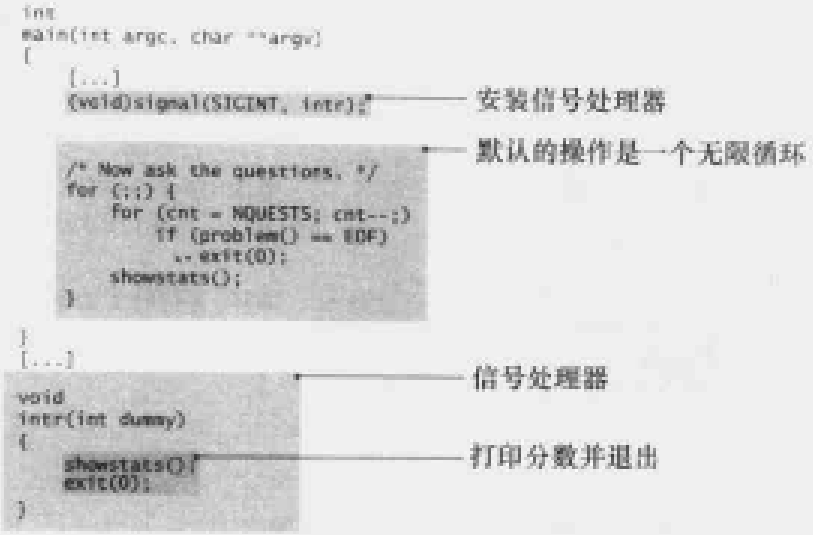


图 5.14 一个信号处理器和它的安装

信号处理器并非仅仅是用来优雅地结束应用程序;它们经常可以提供更为复杂的功能。但是,由于信号处理器可能在程序运行期间的任何时刻被执行,因此,信号处理器中的代码在做出假定时要特别小心。C 函数库的许多部分都是不可重入的,这意味着,如果代码并行地调用这类函数,则会产生不可预期的结果。图 5.15[①] 中的代码是一个信号处理器,它响应用户要求程序显示输出信息的请求(在 BSD 系列的 Unix 操作系统中使用^T 键)。为了避免使用基于文件的 stdio 文件例程(不可重入),处理器在一段内存缓冲区中输出消息,并使用 write 系统调用将其复制到运行程序的终端。

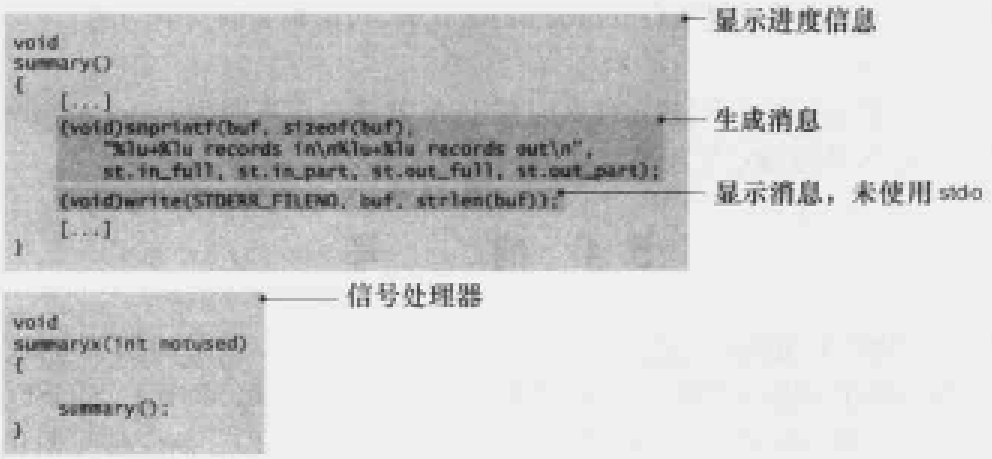


图 5.15 避免在信号处理器中使用不可重入的代码

如果程序中两个并发执行的部分改变公共的数据结构,使得最终的结果依赖于它们执行的次序时,就产生了竞争条件(race condition)。竞争条件很难捉摸,相关的代码常常会将

① netbsdsrc/bin/dd/misc.c:61-98

竞争条件扩散到多个函数或模块；因而，很难隔离由于竞争条件导致的问题。考虑图 5.16[①] 中的 C++代码。ForkProcess 构造函数通过 fork 系统调用创建一个新的进程。即将终止的 Unix 进程必须使用 wait 系统调用与它们的父进程再次会合（被回收）；否则，它们就会成为一个僵进程：结束运行并一直等待父进程收集它们的退出状态和资源使用统计。我们的构造函数通过为 SIGCHLD 信号安装一个信号处理器来完成这项任务，当进程的子进程结束（死亡）时，这个信号会送达进程。安装该信号处理器时要使用 sigaction 系统调用；它的功能类似 signal 系统调用，但提供更多的选项，能够精确地指定对信号的处理方式。所有创建的进程都被插入到一个链表中，信号处理器使用这个链表找出有关结束进程的详细信息。

这就是麻烦产生的地方。信号处理器遍历该列表，使用进程标识符 wpid 查找结束的进程。找到该进程之后，将其从链式列表中移除，并使用 delete 运算符释放为该进程动态分配的内存。然而，由于信号处理器是被异步调用的，它可能在图 5.16：1 中的语句执行后立即运行。如果结束的进程是列表中的第一项，则图 5.16：2 中的语句会试图从列表中移除该进程，但是，由于新插入的元素业已指向这个已经结束的线程，最终，列表中就会包含一个已经处理过的进程的细节，存储在已经释放的内存空间中。delete 操作可能触发第二竞争条件（second race condition）（图 5.16：3）。如果信号处理器，以及随之发生的 delete 操作，在程序的其他部分执行 new 或另一个 delete 操作时运行，这两个操作可能会混杂在一起，破坏程序

```
Fork::ForkProcess::ForkProcess (bool kill, bool give_reason)
  : kill_child (kill), reason (give_reason), next (0)
{
  if (list == 0) {                          ← No handler installed?
    struct sigaction sa;                    ← 安装 SIGCHLD 信号处理器
    sa.sa_handler = sighnd (&Fork::ForkProcess::reaper_nohang);
    sigemptyset (&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sigaction (SIGCHLD, &sa, 0);
  }
  pid = fork ();
  if (pid > 0) {
    next = list;                            ← 1 可能会指向一个业已移除的元素
    list = this;
  } [...]
}

void Fork::ForkProcess::reaper_nohang (int signo)   ← 信号处理器
{ [...]
    ForkProcess* prev = 0;
    ForkProcess* cur  = list;
    while (cur) {
      if (cur->pid == wpid) {               ← 这是已结束的进程吗
        cur->pid = -1;
        if (prev)
          prev->next = cur->next;
        else
          list = list->next;                ← 2 可能移除新插入的项指向的元素
        [...]
        delete cur;                         ← 3 可能会与其他内存池操作发生冲突
        break;
      }
      prev = cur;
      cur  = cur->next;
    } [...]
}
```

图 5.16　由信号处理器引发的竞争条件

① socket/Fork.C:43-159

的动态内存池。C++语言库的多线程版本或许包含防范第二竞争条件的措施，但在我们所了解的应用程序中，由于程序没有显式地使用线程，故而也就没有采取相应的措施。结果是，该应用程序能够可靠地创建数千个进程，然而，一旦系统的负载超过确定的阈值，它会出乎意料地崩溃。由于崩溃由被破坏的内存池触发，所以崩溃的位置和引起崩溃的竞争条件完全没有任何关联。这个例子的寓意是，您应该对出现在信号处理器中的数据结构操作代码和库调用保持极高的警惕。特别要注意，ANSI C 标准规定隐式调用的信号处理器只可以调用 abort，exit，longjmp 和 signal 函数，并且仅能给声明为 volatile sig_atomic_t 的静态对象赋值；任何其他操作的结果都是未定义的。

一种经常用来干净地处理信号的方案是，设置一个标志来响应信号，然后在方便的上下文中检查该标志。图 5.17 中的代码[①]说明了这类设置，以及一些额外的信号处理思想。函数 init_signals，用来启用信号处理，它将会安装响应 SIGWINCH（该信号通知应用程序其显示窗口的大小发生了改变）的信号处理器 winch。信号处理器中的代码十分简单：它再次启用对该信号的处理（信号处理在接收到信号后将恢复默认的行为），并将 sigs 标志设为信号已接收。程序主循环的每次迭代都检查 sigs 标志，在需要的情况下，执行必需的处理工作（图 5.17：3）。另外需要注意的是禁用信号处理器的代码：中断信号 SIGINT 的处理过程被恢复到实现中定义的默认行为（图 5.17：1），窗口大小改变的信号将被忽略（图 5.17：2）。

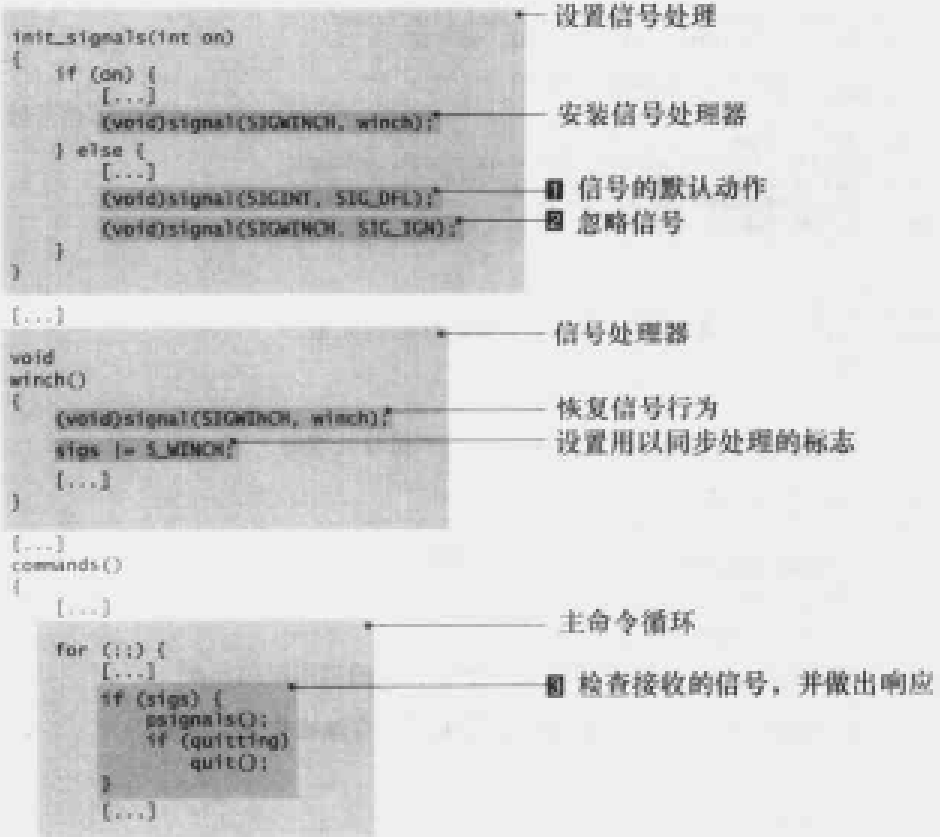


图 5.17 信号的同步处理

① netbsdsrc/distrib/utils/more

练习 5.13　找出 3 个重要的信号处理器，推理它们工作的方式。

练习 5.14　分析您所处的操作环境中，线程如何与信号处理器进行互操作。

练习 5.15　ANSI C 不保证库函数能够重入。分析代码库中信号处理器对不可重入函数的调用。

5.5　非局部跳转

C 函数库中的 setjmp 和 longjmp 函数允许程序从一个函数中直接跳回，勿需从调用序列中的每个函数 return。这两个函数一般用来从深度嵌套的函数中直接退出，这样做常常是为了响应一个特定的信号。这两个函数要一同使用。在嵌套函数调用将要返回的点，用 setjmp 将环境的上下文存储到一个缓冲区变量中。在调用完 setjmp 函数之后，包含 setjmp 调用的函数返回之前这段时间内，调用 longjmp，以指向保存环境的指针作为参数，程序将会跳回到最初调用 setjmp 的点执行。为了将最初的 setjmp 调用与随后通过 longjmp 返回这两种情况区分开来，setjmp 在第一种情况下返回 0，在后一种情况下返回非 0 值。

图 5.18[①] 中的例子是一种典型的情况。用户在使用面向行的文本编辑器 ed 时，能够通过发出一个中断(^C)，中断耗时很长的操作，返回到命令行模式。这是通过安装一个键盘中

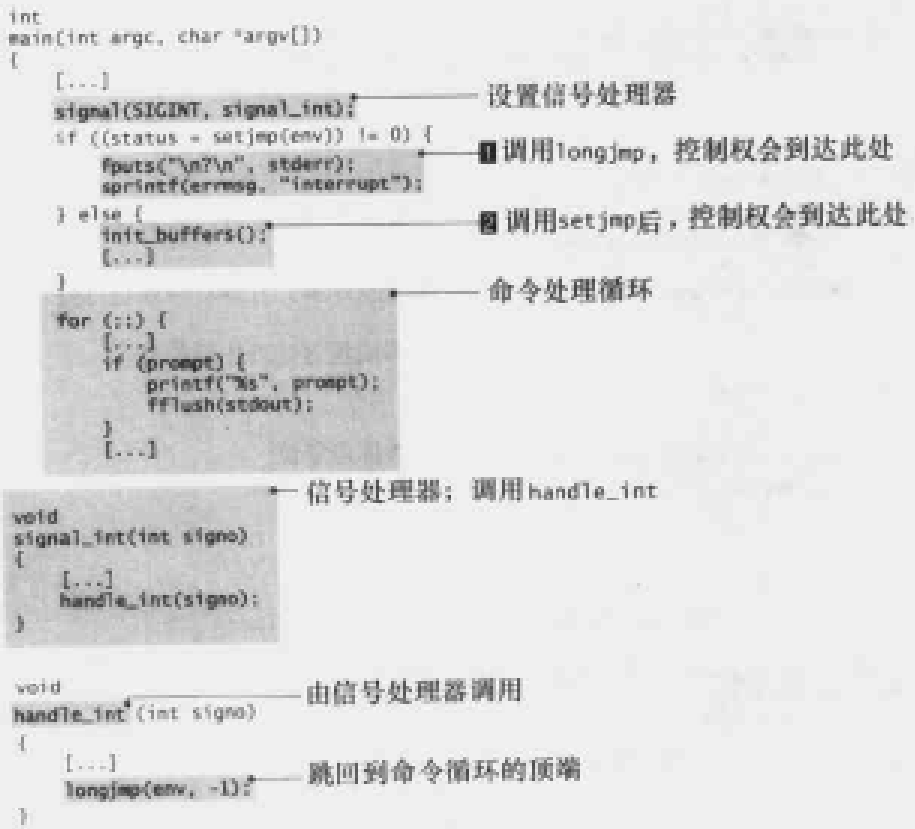


图 5.18　使用 longjmp 的非局部跳转

① netbsdsrc/bin/ed/main.c:114-1417

断信号处理器，并在命令处理循环之前调用 setjmp 来实现的。对 setjmp 的最初调用返回 0，程序将执行正常的启动处理(图 5.18:2)。当用户按下 C 时(例如，为了中断耗时很长的打印工作)，信号处理器 signal_int 将会调用 handle_int，handle_int 会调用 longjmp，返回到命令循环的顶部(图 5.18:1)。

虽然，乍看起来这段代码没有错误，但会产生许多复杂的问题。一个在不恰当时刻到来的信号结合一个返回到命令循环顶部的 longjmp 调用，可能会导致资源泄露(例如，文件没有关闭，或没有释放临时分配的内存块)，甚至更坏，导致数据结构处于一种不一致的状态。我们分析的代码中使用互斥区(mutual exclusion region)来避免这个问题。图 5.19[①] 中的代码片断详细展示了如何完成这项工作。代码中定义了两个宏，SPL1 和 SPL0，分别用来进入和退出临界区域。(宏的名称来源于 Unix 内核的早期版本中用来设置中断优先级别的函数。)SPL1 设置 mutex 变量。信号处理器检查这个变量；如果设置了该变量，那么代码只是将信号的号码存入 sigflags 变量，而不去调用 handle_int。SPL0——在临界区域的结尾调用——递减 mutex，检查 sigflags，看看在临界区域执行期间有没有接收到任何未决的信号，如果需要，则调用 handle_int。使用这种方案，在程序从临界区域退出之前，只是延迟了对信号的响应。

```
int mutex = 0;              /* if set, signals set "sigflags" */
int sigflags = 0;           /* if set, signals received while mutex set */
#define SPL1() mutex++      ——— 进入临界区域，延迟信号处理

#define SPL0() \                                                    ——— 退出临界区域
if (--mutex == 0) { \
    if (sigflags & (1 << (SIGHUP - 1))) handle_hup(SIGHUP); \       ——— 处理未决的信号
    if (sigflags & (1 << (SIGINT - 1))) handle_int(SIGINT); \
}

                                                                    ——— 信号处理器
void
signal_int(int signo)
{
    if (mutex)                                                      ——— 如果处理互斥区域，只是设置标志，延迟处理
        sigflags |= (1 << (signo - 1));
    else                                                            ——— 未设互斥量，立即对中断进行处理
        handle_int(signo);
}

                                                                    ——— 立即或延迟处理中断信号
void
handle_int(int signo)
{
    [...]
    sigflags &= ~(1 << (signo - 1));                                ——— 中断标志复位
    longjmp(env, -1);                                               ——— 跳回到命令循环
}

void
clear_active_list()
{
    SPL1();                 ——— 进入临界区域
                                                                    ——— 临界区域
    active_size = active_last = active_ptr = active_ndx = 0;
    free(active_list);
    active_list = NULL;
    SPL0();                 ——— 退出临界区域
}
```

图 5.19 设置互斥区保护数据结构

① netbsdsrc/bin/ed

练习 5.16　说明为什么 SPL0 和 SPL1 的定义(图 5.19)允许临界区域的嵌套执行。

练习 5.17　找出 5 处调用 longjmp 的案例,并列举使用它们的原因。为了使结果更好,请试着寻找那些和信号不相关的 longjmp 调用。

练习 5.18　您所找到的,含有 longjmp 的代码如何防止资源泄露和数据结构的不一致呢?

练习 5.19　建议一种验证包含 setjmp 的函数退出后,没有调用 longjmp 的方法。这是一种编译期间的方法,还是运行期间的方法?

5.6　宏　替　换

程序中经常会利用 C 预处理器提供的灵活性,将简单的函数定义为宏。①

```
# define IS_IDENT(c) (isalnum(c) || (c) == '_' || (c) == '.' \
      || (c) == '$')
# define IS_OCTAL(c) ((c) >= '0' && (c) <= '7')
# define NUMERIC_VALUE(c) ((c) - '0')
```

阅读含有宏的代码,要注意,宏既非函数,又非语句。定义宏的时候,应该采用谨慎的编码准则,以避免一些常见的陷阱。比如,在上面 NUMERIC_VALUE 宏的定义中,c 都被括号括起来。如果不加括号使用 c,那么像 NUMERIC_VALUE(charval =b)这样的表达式就会被作为 charval =(b -'0')进行求值,而非期望的(charval =b) -'0'。通过将宏的所有参数都括号化,代码避免了优先次序的问题。更为困难的情况是宏参数被求值的次数。下面形式的代码:

```
while (IS_IDENT(a = getch())
    putchar(a);
```

将被扩展为:

```
while ((isalnum(a = getchar()) || (a = getchar()) == '- ' ||
      (a = getchar()) == '.' || (a = getchar()) == '$'))
        putchar(a);
```

每次循环迭代都会读入 4 个不同的字符,而这当然并非本意。一般地,当看到宏参数有副作用时,比如赋值、前缀/后缀递增/递减、输入/输出和含有这些效果的函数调用,应该十分小心。

将宏用作语句时,会导致另外的问题。考虑下面的定义:②

```
# define ASSERT(p) if (! (p)) botch(__ STRING(p))
```

① netbsdsrc/usr. bin/yacc/defs. h:131-133

② netbsdsrc/lib/libc/stdlib/malloc. c:130

用在下面的上下文中：

```
if (k > n)
    ASSERT(process(n) == 0);
else
    ASSERT(process(k) == 0);
```

宏扩展后，这段代码将被作为：

```
if (k > n)
      if (! (process(n) == 0))
        botch(__STRING(process(n) == 0));
      else
        if (! (process(k) == 0))
          botch(__STRING(process(k) == 0));
```

这并非原来的意图。为了避免这类问题，我们可以将包含 if 语句的宏定义为一个整体。①

```
# define NOTE(str) \
    { if (m -> eflags&REG_TRACE) printf("=%s\n", (str)); }
```

然而，在下面的上下文中使用 NOTE 宏

```
if (k > 1)
    NOTE("k > 1");
else
    process(k);
```

会扩展为：

```
if (k > 1) {
    if (m -> eflags&REG_TRACE) printf("=%s\n", ("k > 1"));
}; else
    process(k);
```

这段代码不能编译通过，因为在 else 前面有一个多余的分号。因为这些原因，在宏定义中，经常会将语句序列放入 do { ... } while (0)块中。②

```
# define getvndxfer(vnx) do {                                          \
  int s = splbio();                                                    \
  (vnx) = (struct vndxfer *)get_pooled_resource(&vndxfer_head);        \
  splx(s);                                                             \
} while (0)
```

这些控制块，除了创建一个作用域，可以定义局部变量以外，还包围在 do... while 语句中，以保护使用这个宏的 if 语句不会受到意外的交互作用。例如，下面的代码③

① netbsdsrc/lib/libc/regex/engine.c:128
② netbsdsrc/sys/vm/vm_swap.c:293-297
③ netbsdsrc/sys/vm/vm_swap.c:101-104

```
# define DPRINTF(f, m) do {          \
    if (vmswapdebug & (f))           \
      printfm;                 \
} while(0)
```

可以放入一个 if 分支，不会产生任何上述的问题。宏扩展的结果不是一个语句，而是在所有情形中都需要一个后续分号的代码片断。因此，调用以这种形式扩展的宏必须后跟一个分号；没有选择。从而，这类宏调用看起来总是与一个语句完全相同，宏的用户不会受到类似的误导和混淆。

另一种替代方案要用到 C 的？：运算符或布尔运算符的短路求值属性，将条件运算编写为表达式。①

```
# define SEETWO(a, b) (MORE() && MORE2() && PEEK() == (a) && \
                       PEEK2() == (b))
# define EAT(c) ((SEE(c)) ? (NEXT(), 1) : 0)
```

虽然上面的宏代码看起来很像函数定义，但要牢记这些序列并不是像函数一样被调用，而是在每个使用它的地方进行词法替换。这意味着，宏定义体中的标识符在使用宏的函数体的上下文中进行解析，并且，宏定义体中对任何变量值的修改都会传播回“调用”宏的点。考虑宏 PEEK，GETNEXT 和 MORE（图 5.20:1）定义中参数 p 的使用。② 用在函数 p_str 中时，p 指代该函数的参数（图 5.20:2），而在后面，p 指代函数 p_count 的参数（图 5.20:3）。

```
#define    PEEK()    (*p->next)
#define    GETNEXT() (*p->next++)
#define    MORE()    (p->next < p->end)

static void
p_str(register struct parse *p)
{
    REQUIRE(MORE(), REG_EMPTY);
    while (MORE())
        ordinary(p, GETNEXT());
}
[...]
static int              /* the value */
p_count(register struct parse *p)
{
    register int count = 0;
    register int ndigits = 0;

    while (MORE() && isdigit(PEEK()) && count <= DUPMAX) {
        count = count*10 + (GETNEXT() - '0');
        ndigits++;
    }
```

图 5.20　使用局部定义变量的宏

另外，传递给宏的变量可以在宏定义体内修改，如下面的代码所示：③

```
# define getnum(t) (t) = 0; while (isdigit(*cp)) (t) = (t) * 10 + (*cp++ - '0');
    cp = buf;
    if (*cp == 'T') {
```

① netbsdsrc/lib/libc/regex/regcomp.c:153-154

② netbsdsrc/lib/libc/regex/regcomp.c:150-667

③ netbsdsrc/bin/rcp/rcp.c:595-606

```
setimes++ ;
cp++ ;
getnum(mtime.tv_sec);
if (*cp++ != ' ')
  SCREWUP("mtime.sec not delimited");
getnum(mtime.tv_usec);
if (*cp++ != ' ')
  SCREWUP("mtime.usec not delimited");
getnum(atime.tv_sec);
```

上面的例子中，传递给 getnum 的每个参数都依照 cp 指向的字符串被赋予一个值。如果将 getnum 定义为一个适当的 C 函数，则需要传递指向相应变量的指针。

宏替换和函数定义一个最根本的不同是：预处理器能够使用 ## 运算符连接宏的参数。下面的代码①

```
# define DOVREG(reg) tf -> tf_vm86_##reg = (u_short) vm86s.regs.vmsc.sc_##reg
# define DOREG(reg) tf -> tf_##reg = (u_short) vm86s.regs.vmsc.sc_##reg
    DOVREG(ds);
    DOVREG(es);
    DOVREG(fs);
    DOVREG(gs);
    DOREG(edi);
    DOREG(esi);
    DOREG(ebp);
```

将扩展为：

```
tf -> tf_vm86_ds = (u_short )vm86s.regs.vmsc.sc_ds;
tf -> tf_vm86_es = (u_short )vm86s.regs.vmsc.sc_es;
tf -> tf_vm86_fs = (u_short )vm86s.regs.vmsc.sc_fs;
tf -> tf_vm86_gs = (u_short )vm86s.regs.vmsc.sc_gs;
tf -> tf_vm86_ds = (u_short )vm86s.regs.vmsc.sc_ds;
tf -> tf_edi = (u_short )vm86s.regs.vmsc.sc_edi;
tf -> tf_esi = (u_short )vm86s.regs.vmsc.sc_esi;
tf -> tf_ebp = (u_short )vm86s.regs.vmsc.sc_ebp;
```

从而使用字段的名称，如 tf_vm86_ds 和 sc_ds，动态地创建访问结构字段的代码。

早期的 pre-ANSI C 编译器不支持 ## 运算符。但可以通过用一个空注释(/**/)隔离两个标记，来欺骗它们完成连接；在阅读 pre-ANSI C 代码时，可能会看到此类应用。在下面的代码中，您能够看到在早期的程序中，如何达到 ## 运算符的效果。②

```
# ifdef __STDC__
# define CAT(a,b) a## b
# else
```

① netbsdsrc/sys/arch/i386/vm86.c:419-428

② netbsdsrc/sys/arch/bebox/include/bus.h:97-103

```
# define CAT(a,b) a/* * /b
# endif
```

练习 5.20　在源代码库中，找出所有 min 和 max 宏的定义。统计这些宏定义中定义恰当的有多少，可能导致错误应用的有多少。

练习 5.21　C++和某些 C 编译器支持 inline 关键字，这个关键字用来定义在被调用处进行直接编码的函数，避免了函数调用的开销，并且为编译器提供了在调用者上下文中对它进行额外优化的机会。根据使用每种方法时代码的易读性、可维护性和灵活性，讨论这种机制与宏定义的关系。

练习 5.22　比较 C++基于模板的代码，与使用宏实现类似的功能，二者在易读性上的差别(9.3.4 节)。使用这两种方法，创建一个简单的例子，并比较在代码中加入语法或语义错误时，编译器报告出来的错误消息。

进　阶　读　物

5.1 节中提到的 Unix 外壳的完整语法发表在 Bourne[Bou79]中。Aho 等著的[ASU85,pp44-55]中对递归下降分析做了更为详细的论述。计算机构架和操作系统方面的著作，为如何提供以及如何在各种软硬件层面上利用并行处理提供了大量的资料[Tan97, HP96,PPI97,BST89]。更特殊地，两本其他参考书目[NBF96,LB97]对线程做了详细的介绍，Robbins 和 Robbins[RR96]中对信号和线程的操作做了介绍。Kernighan 和 Pike[KP84]中介绍了大量基于流水线的编程示例，流水线的理论背景在 Hoare[Hoa85]中进行了详细的阐述。操作系统的设计为处理并发性问题提供了大量的机会；这些内容在下面几本参考书中有过论述[Bac86, Tan97, LMKQ88, MBK96]。宏定义的常见问题在 Koenig[Koe88]中进行了论述，Meyers[Mey98]对内联函数的优点与缺点做了巨细无遗的论述。

第6章　应对大型项目

通过扩展小型系统的规模而生成的大型系统，其行为不同于小型系统。

——John Gall

大型的多个文件项目与小型项目之间的不同，并非仅仅在于分析它们的代码时，您会遇到更多挑战，还在于它们提供了许多理解它们的机会。在本章中，我们将回顾一些用在实现大型项目中的常用技术，之后，分析这类项目开发过程中具体的构成部分。我们将描述大型项目的组织方式，它们的编译和配置过程，不同文件版本如何控制，项目专用工具的特殊角色，以及典型的测试策略。我们简略地给出这些内容常见的典型设置，并提供一些提示，帮助您了解如何使用它们来增强您的导航和理解能力。

6.1　设计与实现技术

大型的编码工作，由于它们的大小与范围，经常能够证明应用一些技术的必要性，而在其他情况下这些技术可能根本不值得使用。本书中多处论及了许多这类技术；但是，为了便于您的阅读，此处还是概括了常见的一些设计与实现方法，并指出了对它们进行详细论述的章节。

(1)可视化软件过程和实用准则

在大型项目中，软件生命周期的各个组成部分常常被划分成更小的任务进行处理，它们经常是软件代码库的一个组成部分。另外，重大项目所固有的复杂性，以及大量开发人员的参与，都要求必需采用某种正式的准则来组织开发工作。这些准则一般会规定如何使用特定的语言、项目各组成部分的组织方式、应该记录一些什么内容、以及项目生命周期中大部分活动的过程。我们在第7章介绍编码规范，6.3节介绍编译过程，6.7节介绍测试过程，6.4节介绍配置方法，6.5节介绍修订控制，第8章介绍文档编制的准则。

(2)重要的构架

尽管小型的项目有时通过简单地堆砌代码块，直至满足要求的规格说明为止，也能够侥幸成功，但是，大规模的软件开发工作必须使用一种合适的构架来构造所要创建的系统，控制其复杂性。这个构架一般指定系统的结构，控制的处理方式，以及如何对系统各个组成部分进行模块分解。大型的系统还可能会从框架(framework)、设计模式(design pattern)和特定领域的构架中汲取灵感，重用它们构架上的思想。我们在第9章进行全面讨论。

(3)积极的分解

大型项目中，在实现层，经常使用诸如函数、对象、抽象数据类型和组件等机制，对系统的组成元素进行积极的分解。我们在第9章进行全面的讨论。

(4)多平台支持

大型的应用程序经常能够吸引相当广泛的用户团体。因此，这类应用程序经常需要处理大量的可移植问题，而在那些不怎么受人关注的项目中，常常可以不考虑这些问题。

(5)面向对象技术

大型系统的复杂性经常可以使用面向对象的设计和实现技术加以控制。一般将对象组织成为不同的层次。之后，就可以使用继承(在 9.1.3 中论述)来提取出公共行为，动态调度技术(9.3.3 节)使得单一代码体能够处理众多不同的对象。

(6)运算符重载

C++和 Haskell 等语言编写的实用代码集合中，常常会使用运算符重载，将项目专有的数据类型提升为“一等公民”，然后就可以使用语言的内建运算符集对它们进行操作。可以在 9.3.3 节中找到一些具有代表性的例子。

(7)库、组件和进程

在更大的粒度上，大型系统的代码经常分解为对象模块库，可重用组件，甚至独立的进程。我们在 9.3 节中论述这些技术。

(8)领域专用和定制的语言和工具

大规模的编码工作经常涉及专门工具的创建，或者从相似用途的现有工具中受益。关于编译过程中如何使用专门的工具，在 6.6 节做了更多的介绍。

(9)对预处理的积极使用

用汇编、C 和 C++语言实现的项目经常使用预处理器，用领域专有的结构对语言进行扩展。现代的项目很少用符号语言编写而成；我们在 6.6 节中论述 C 预处理器的使用。

练习 6.1　请建议一些方法，可以快速地确定给定的项目是否遵循我们描述过的某个设计或实现技术。在本书配套盘上一个主要的项目上测验您的提议。

练习 6.2　在软件工程学的书籍中，找出它所推荐的设计与实现准则。说明这些准则是如何体现在项目的源代码中的？

6.2　项目的组织

我们可以通过浏览项目的源代码树——包含项目源代码的层次目录结构，来分析一个项目的组织方式。源码树常常能够反映出项目在构架和软件过程上的结构。考虑 apache Web 服务器的源代码树(图 6.1)。在最顶层，它的目录结构与典型的 Web 服务器安装目录完全相同：这些目录分别用来存储服务器脚本(cgi-bin)、配置文件(conf)、静态内容(htdocs)、图标和服务器日志。应用程序的源代码树经常会是该应用程序的部署结构的镜像。实际的程序源代码存储在名为 src 的目录中，这个目录名经常用于这种目的。其他目录中是一些示例文件和模板，用来安装和部署这个应用程序。另外，就 apache 来说，它的文档以静态 Web 页面的形式存储在 htdocs/manual 目录中。程序的源代码也被组织在一些典型的目录中：lib 中是库代码，main 中是服务器的基础代码，include 中是公共的头文

件，modules 中是可选安装部件的代码，os 中是操作系统相关的代码。其中的一些目录又做了进一步的划分。lib 目录划分成几个独立的目录，它们是存储 XML 语法分析程序代码的目录(expat-lite)和提供散列数据库支持的库(sdbm)。os 目录也划分成几个不同的子目录：每种操作系统或平台类型对应一个目录。开发多平台应用程序的一个通常策略是将平台相关的代码隔离开来，针对每种平台进行定制。然后，将这些代码放入分离的目录中；依据目标平台，配置其中的某一个参与到编译过程中。我们在表 6.1 中汇总了不同项目中常见的目录名。依据系统大小的不同，我们描述的目录可能适用于整个系统，或者系统中每个部分都有自己独立的目录结构。在这两种组织策略中选择哪一种依赖于开发与维护过程的组织方式。集中式的过程倾向于使用公共的目录，从而能够利用不同应用程序之间的共性，而联盟式的开发过程则在每个不同的项目中都重复这种结构，从而获得维护上的独立性。

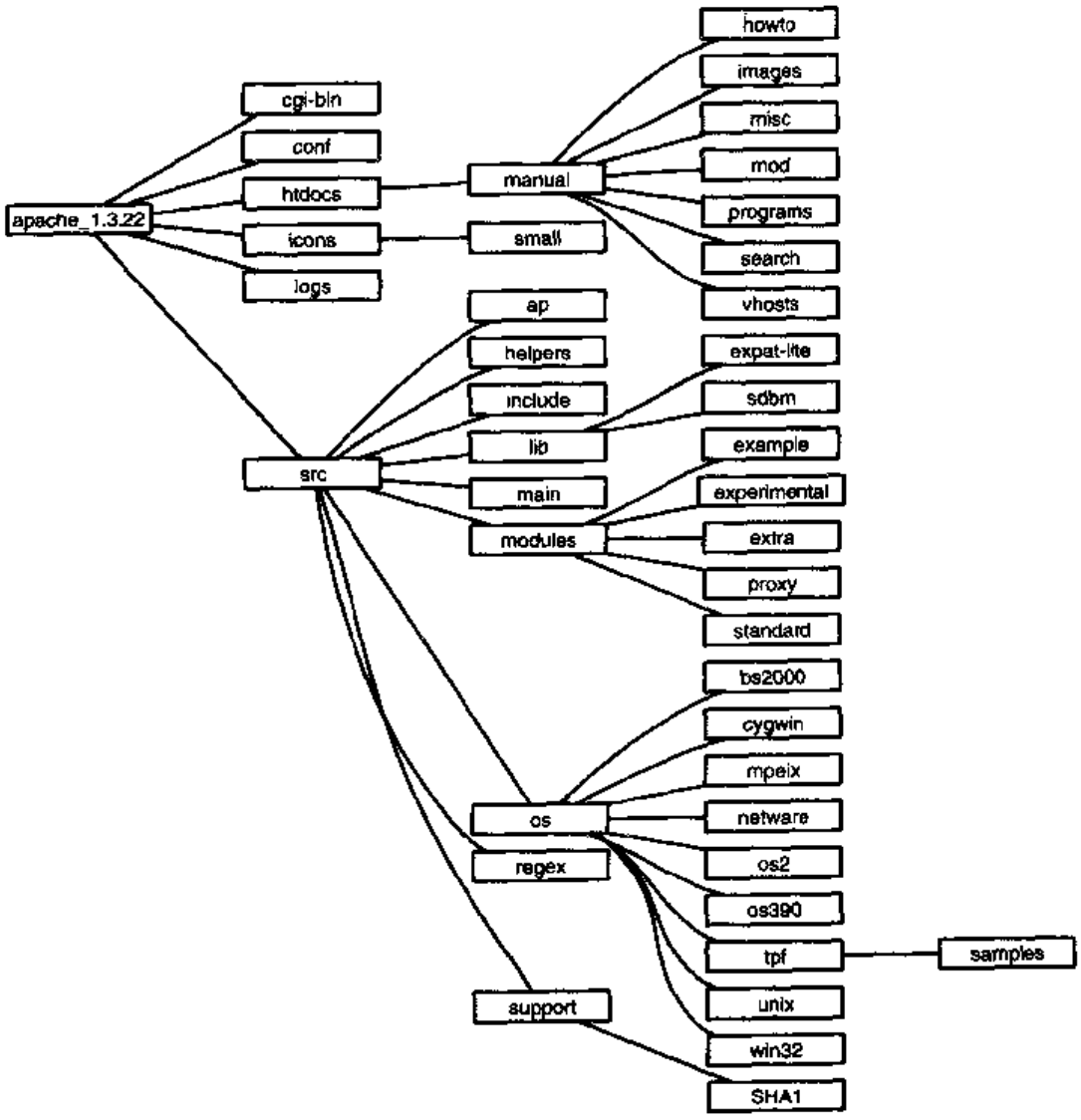


图 6.1 apache Web 服务器源代码树的结构

表 6.1　常见的项目子目录名

目录	内容
src	源代码
TLD(例如 org 或 edu)	Java 源代码的根目录
main	主程序的源代码;平台无关的外部库
program-name	*program-name* 的源代码
lib	库的源代码
libname	*name* 库的源代码
common	应用程序间共享的代码
include	公共的 C、C++头文件
doc	文档
man	Unix 手册页
rc/res	Microsoft Windows 中源码/编译格式的资源文件(位图、资源文件、对话框、图标)
arch	专门针对给定处理器构架的代码
os	操作系统的专属代码
CVS/RCS/SCCS	修订控制系统的文件
build/compile/classes/obj/Release/Debug	编译目录
tools	用在编译过程中的工具
test	测试脚本和输入输出文件
conf	编译期间的配置信息
etc	运行期间的配置信息
eg	示例
bin	存储可执行文件和外壳脚本的储存库
contrib	用户提供的工具和代码,一般在主项目之外进行维护

随着项目大小的增长,它的目录结构也会不断得到剪裁,以适应项目的特殊需要。历史常常会影响项目的目录结构,因为开发人员和维护人员不愿意重新构造他们已经熟悉的代码库。请比较 NetBSD 内核的目录结构(图 6.2)与 Linux 内核的目录结构(图 6.3)。Linux 内核的目录结构被比较统一地划分为许多子目录;而在 NetBSD 中,许多应该归结到一个公共目录下的目录,却由于历史的原因,出现在最顶层。典型的例子是网络代码的组织。不同的网络协议(TCP/IP①,ISO②,ATM③,AppleTalk④)在 NetBSD 内核中作为最顶层的目录出现,而在 Linux 中,则组织到一个公共的 net 目录下。虽然 BSD 网络代码的开发人员,在设计之时,就预先考虑了要使用一种能够进行扩展以支持额外协议的联网目录命名方案,但是,他们没有采取额外的步骤,创建一个单独的目录,来保存支持单个联网协议的相关文件。

如果创建新代码的过程定义得十分清晰明确,那么,产生的目录结构往往会十分整洁有序。NetBSD 内核中与构架相关的部分即为如此,见图 6.4。多年来,NetBSD 已经移植到数

① netbsdsrc/sys/netinet

② netbsdsrc/sys/netiso

③ netbsdsrc/sys/netnatm

④ netbsdsrc/sys/netatalk

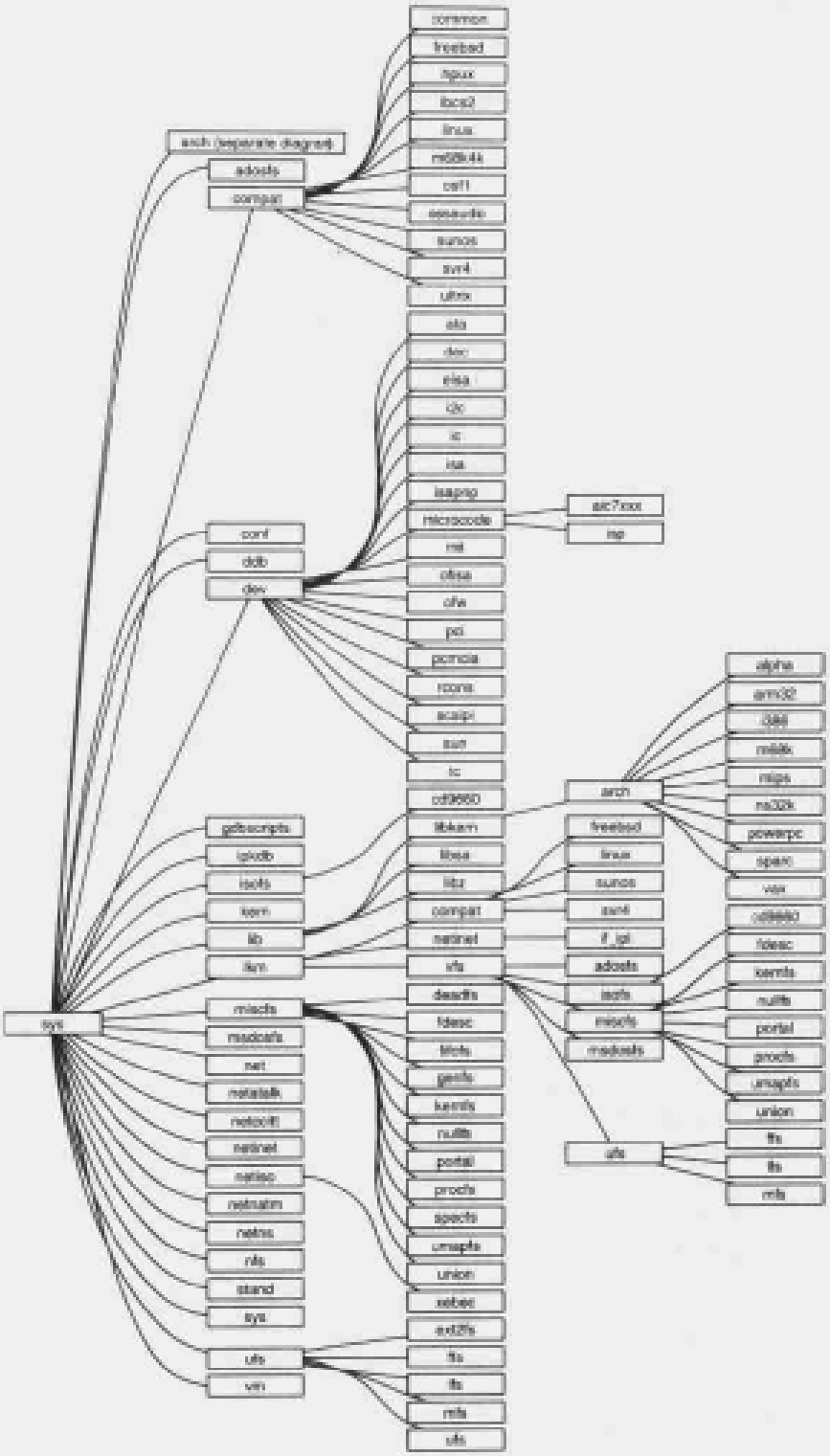


图 6.2 NetBSD 内核的主源码树

十种不同的构架。与构架相关的部分被归组到一个单独的目录下①，在配置与编译过程中，它们会相应地链接到主控源代码树(master source code tree)。

① netbsdsrc/sys/arch

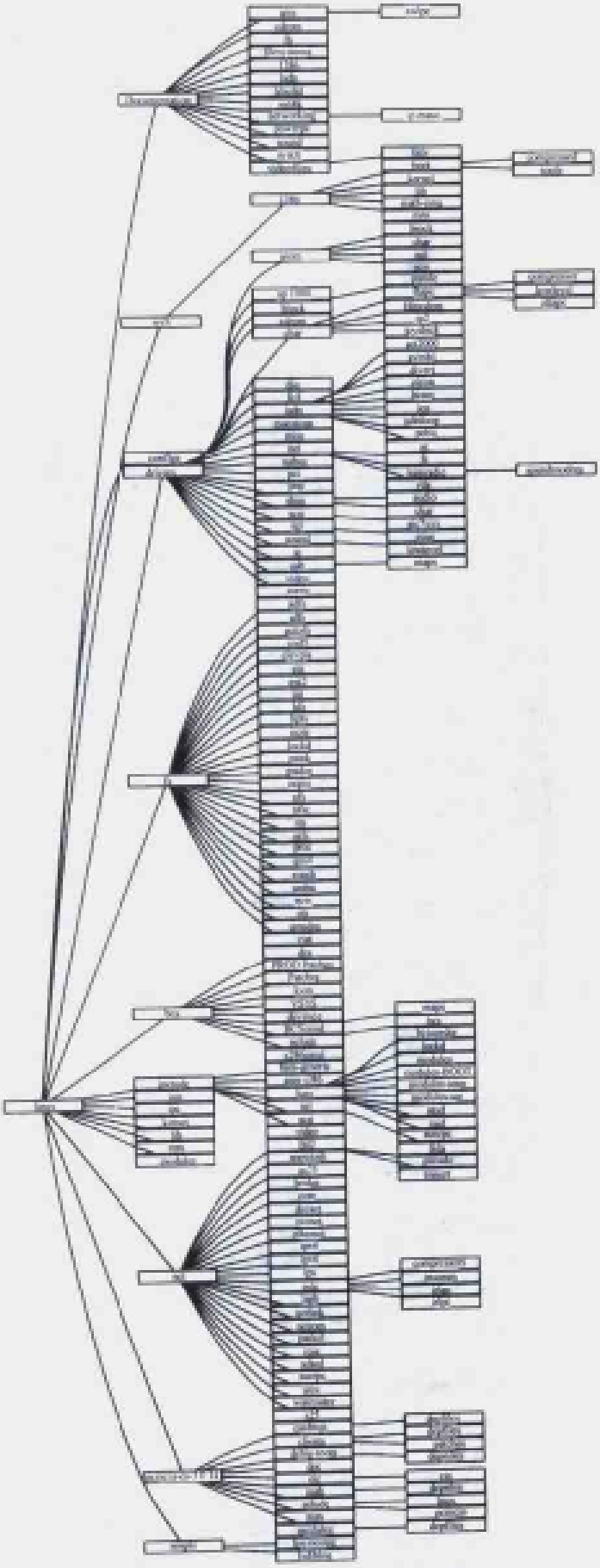


图 6.3　Linux 内核的源码树

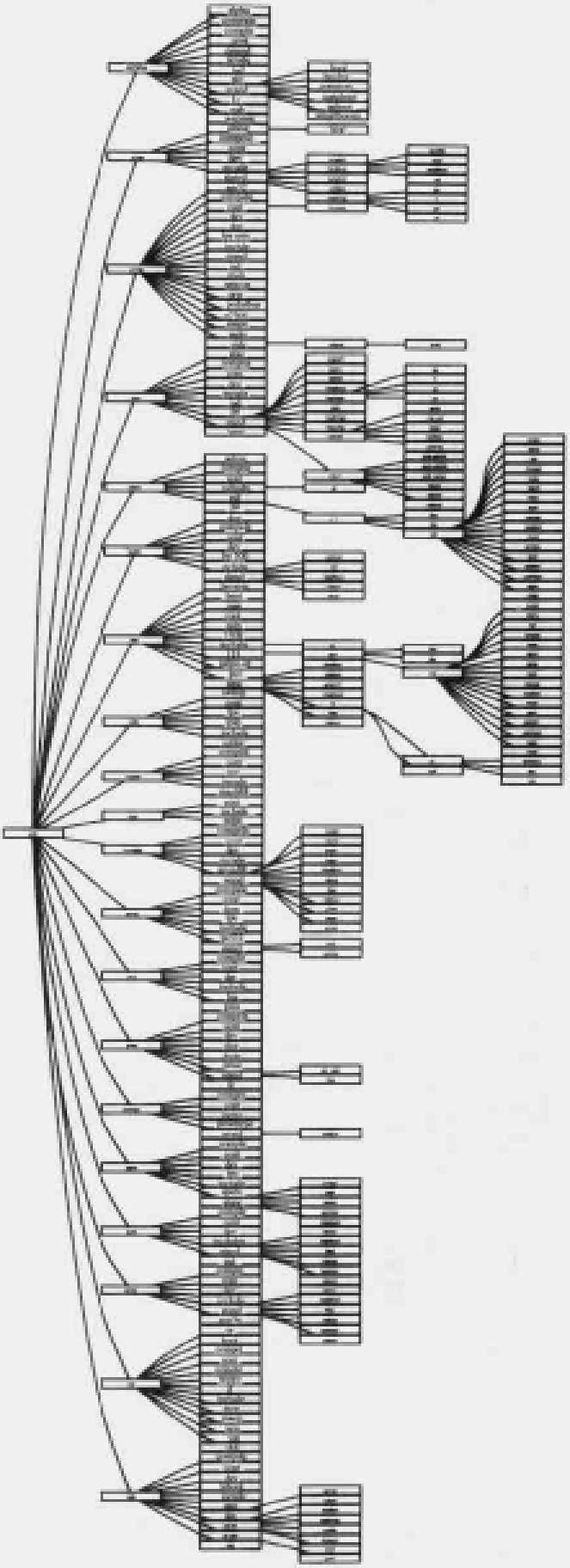


图 6.4 NetBSD 中与构架相关部分的源码树

不要被庞大的源代码集合吓倒：它们一般比小型的专门项目组织得更出色。图 6.5 描绘出整个 FreeBSD 系统的源代码概况，包括内核、库、文档、用户和系统工具。尽管它十分庞大（它由近 3 000 个目录组成，这还未将配置文件储存库统计在内），但找到特定工具的源

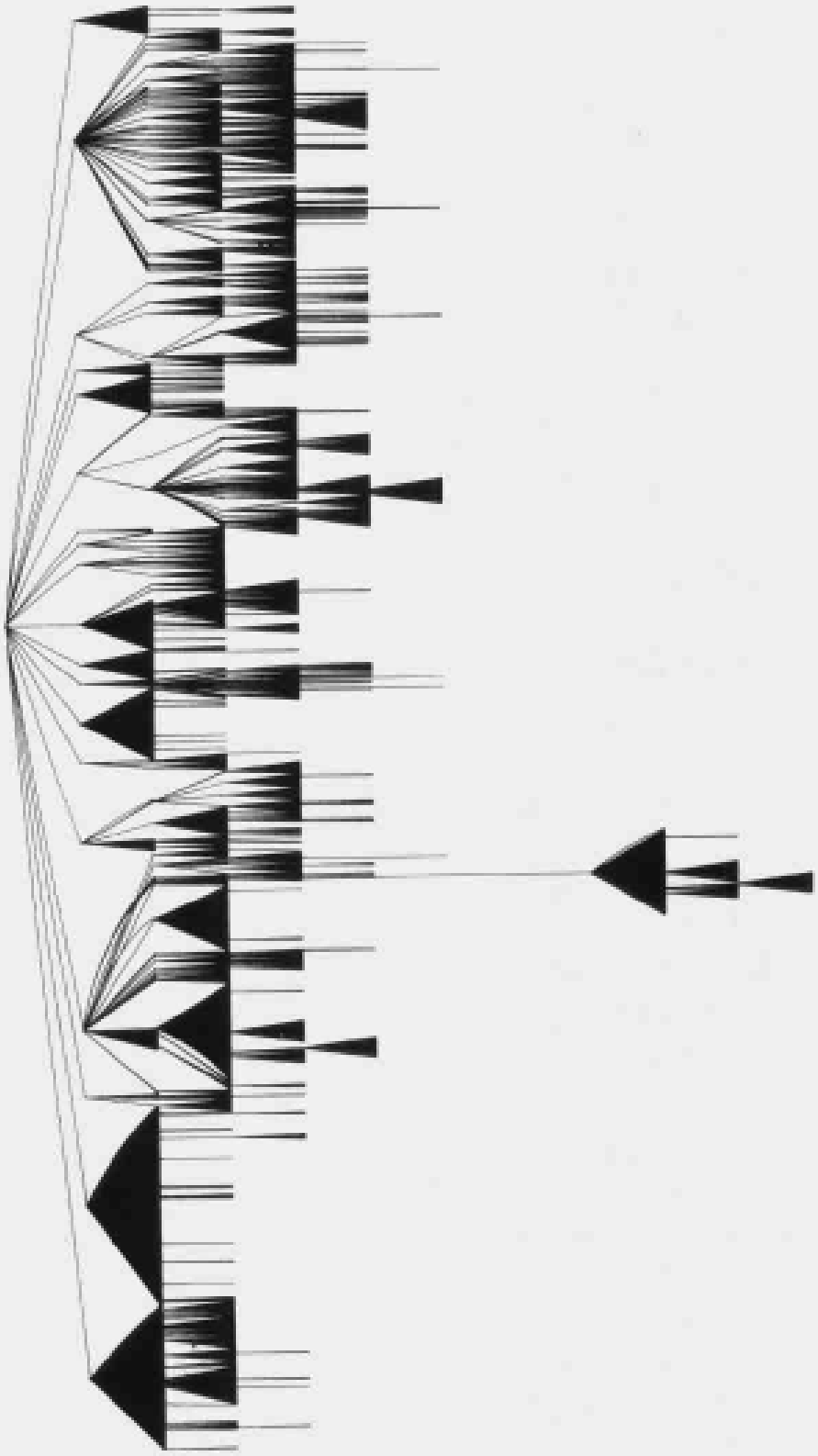


图 6.5　FreeBSD 系统的源码树

代码依旧轻而易举：工具的源码所在的目录是按照工具所在路径的匹配名称＋工具的名称，来命名的，例如，make 工具的源代码（安装后的程序为/usr/bin/make）在目录/usr/src/usr. bin/make 中。当您首次接触一个大型项目时，要花一些时间来熟悉项目的目录树结构。许多人喜欢使用图形化的工具来完成这项任务，比如 Windows 资源管理器，或 GNU midnight commander[①]。如果这两种工具都不能使用，您可以使用 Web 浏览器打开本地源码目录，对目录结构进行图形化浏览。在大多数 Unix 系统上，您还可以使用 locate 命令找出所寻找的文件，在 Windows 平台上，资源管理器的文件搜索机制可以完成类似的任务。

在浏览大型的项目时，要注意，项目的源代码远不只是编译后可以获得可执行程序的计算机语言指令；一个项目的源码树一般还包括规格说明、最终用户和开发人员文档、测试脚本、多媒体资源、编译工具、例子、本地化文件、修订历史、安装过程和许可信息。

练习 6.3 描述本书配套盘中或者您的项目中，所使用的目录结构。

练习 6.4 如何用标准化的目录结构自动体现软件开发的过程。

练习 6.5 分析并描述某个版本的 Microsoft Windows 安装后的目录结构。

练习 6.6 简述哪些元素，除了源代码以外，被打包到 Perl 的分发包（本书配套盘中提供）中。

6.3 编译过程和制作文件

大多数大型的项目使用一个复杂的编译过程。这类过程一般能够处理配置选项、多种类型的输入输出文件、错综复杂的相互依赖和多个编译目标。由于编译过程最终会影响生成的输出，所以能够“阅读”项目的编译过程和阅读项目的代码同样重要。遗憾的是，对于如何描述与执行编译过程，没有标准化的方式。每个大型项目和开发平台都使用自己专有的方式来组织编译工作。然而，大多数编译过程中，一些元素是公共的；我们在下面的段落中对它们进行概括。

图 6.6 例举出一个典型编译过程所涉及的各个步骤。第一步是配置软件的选项，并精确地确定所要编译的内容。（我们在 6.4 节中讨论配置。）根据项目的配置，我们可以创建一个依赖关系图（dependency graph）。大型项目一般涉及数万个不同的组件；其中的大部分组件都与其他组件有关。依赖关系图说明各个项目组件的正确编译次序。编译过程中的某些部分常常不是使用标准的开发工具，比如编译器或链接器，来完成，而是需要使用专为特定项目开发的工具。一个典型的例子是 imake 工具，可以用它来标准化跨 X Window 系统的编译过程。项目专用的工具编译后，就能够与其他标准工具一同使用，预处理、编译和链接项目的可执行文件。同时，项目的文档经常同步地从“源码”格式转换为最终的发行格式。

① *http://www.ibiblio.org/mc/*

这可能涉及编译 Microsoft Windows 应用程序使用的帮助文件，或者 Unix 手册页的排版。最后是将生成的执行文件和文档安装到目标系统，或准备用于大规模的部署或发行。典型的发行方法包括 Red Hat Linux RPM 文件，Windows installer 格式，或者将适当的文件上载到 Web 站点。

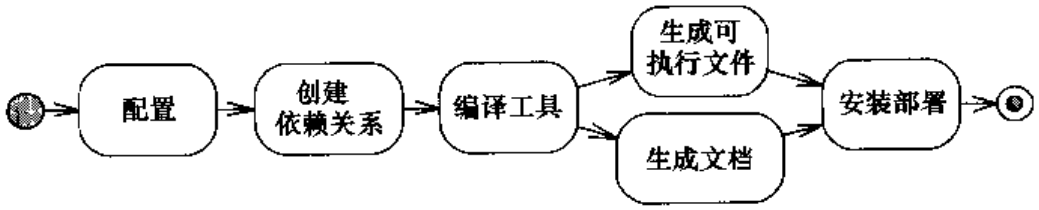


图 6.6　典型编译过程的各个步骤

编译过程中最为错综复杂的部分是项目依赖关系的定义与管理。这些关系说明项目中不同部分之间的相互依赖关系，以及项目中各个部分进行编译的次序。我们在图 6.7 中例举了项目中一组典型的依赖关系。这个项目的发行文件依赖于执行文件和文档文件的存在；有时，项目专有的动态载入库和软件组件也常常是最终发生文件的一部分。可执行文件依赖于目标文件、库和组件，因为可执行文件就是这些文件链接到一起产生的。库和组件也依赖于目标文件。目标文件依赖于相应源文件的编译，在使用 C 和 C++等语言的情况下，还依赖于源文件中包括的头文件。最后，一些源文件是从领域专有的文件中自动产生的（例如，yacc 语法文件经常用于创建一个用 C 语言编写的分析器）；因此，源文件依赖于领域专有的代码和相应的工具。在我们的描述中，可以看到依赖关系与编译过程是如此紧密地链接在一起。在描述完具体的依赖关系图后，我们将讨论如何利用这种关系引导编译过程。

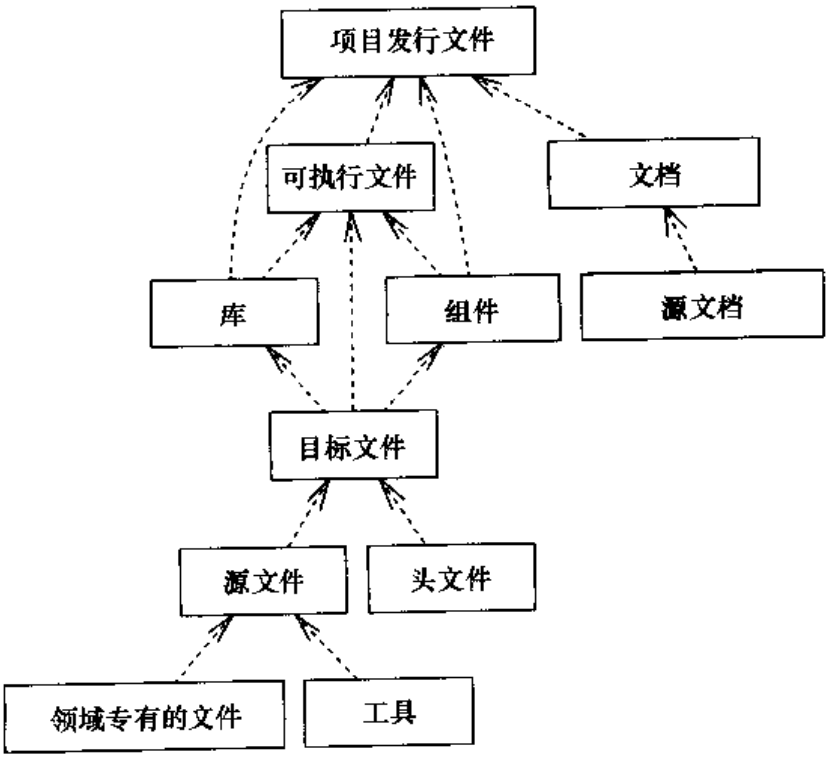


图 6.7　一组典型的项目依赖关系

图 6.8 是一个项目依赖关系的具体示例，它描绘了编译 apache Web 服务器的过程中涉及的一个虽然小，但很有代表性的部分。安装过程依赖于 apache 守护进程 httpd。守护进程的执行文件 httpd 依赖于多个目标文件（其中有 buildmark.o 和 modules.o）和库（比如 XML 库 libexpat.a 和 libap.a）。XML 分析库依赖多个不同的目标文件；我们例举出其中的一个（xmltok.o），它依赖于相应的 C 源文件 xmltok.c。

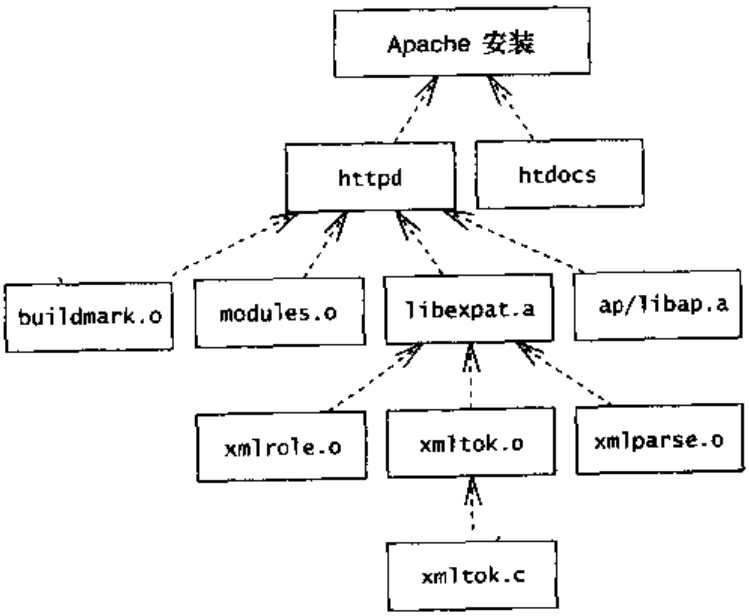


图 6.8 apache Web 服务器中具有代表性的依赖关系

在大型项目中，依赖关系涉及数以千计的源代码文件，并会根据具体的项目配置做出更改。例如，apache 服务器的 Windows 编译不依赖于 Unix 相关的源代码文件。用 C 和 C++ 编写的项目使依赖关系变得更为复杂，这是因为每个目标文件不但依赖于对应的 C 或 C++ 源文件，还依赖于源文件中包括的所有头文件。将依赖关系发展为具体的编译过程并不容易。它涉及对依赖关系图执行拓扑排序；对目标的编译就要依照拓扑排序的结果（例如，在编译库之前，首先要编译库使用的目标文件）。令人高兴的是，这个过程一般能够使用专门的工具自动完成，比如 make，nmake 和 ant。

在使用诸如 make 之类的工具时，由文件的各个构成部分构造文件需要遵循的依赖关系与规则在一个单独的文件中指定，一般命名为 makefile 或 Makefile。之后，以这个制作文件为参数（常常隐式）运行 make，就可以完成编译过程中大部分的工作。make 读取并处理该制作文件，之后发布适当的命令编译指定的目标。制作文件中绝大部分内容是一个个的项，每个项描述目标、目标的依赖关系和从这些依赖关系生成目标的规则。例如，下面这二行内容指定目标 patch.exe 依赖于一些目标文件（$(OBJS)）和一条调用 C 编译器编译该目标文件的规则。①

```
patch.exe: $(OBJS)
  $(CC) $(OBJS) $(LIBS) -o $@  $(LDFLAGS)
```

① XFree86-3.3/xc/util/patch/Makefile.nt:23-24

制作文件中使用了许多机制来增强表达性；您可以在图 6.9 中看到用在 apache 制作文件中的一些机制。要注意，大型项目中，制作文件常常由配置步骤动态地生成；在分析制作文件之前，需要先执行项目特定的配置。

为了提取出广泛使用的文件列表和工具选项，常常要定义一些变量（也称为宏），使用的语法是：变量＝文本（VARIABLE=text）。图 6.9:1 中的定义，将变量 OBJS 定义为 apache 可执行文件直接依赖的目标文件。变量一般全部大写，另外，一些标准的变量名经常用来表达典型的概念。在制作文件中使用时，变量使用语法 $L（单字母变量名）或 $(NAME)——多个字母组成的变量。我们在表 6.2 中汇总了一些常见的用户自定义变量。

为了避免对普通规则的重复，make 允许用户定义，根据参与转换过程的文件的后缀生成目标文件的规则。在图 6.9:3 中，这条规则规定如何调用 emximp 命令从它的定义（.def）生成一个 OS/2 输入库（.a）。大多数 make 的实现都维护一套巨大的内部转换规则和现有的文件后缀。它们一般规定如何编译 C、C++、汇编或 Fortran 文件，如何将 yacc 和 lex 文件转换成 C 语言，以及如何提取出存储在修订控制系统中的源文件。当一个后缀不属于 make 已知的后缀时，需要使用一个特殊的.SUFFIXES 命令对它进行定义（图 6.9:2）。为了允许用户在规则中指定抽象文件名，make 自动地维护许多变量，根据正在处理的目标文件和规则的依赖关系更改它们的内容。我们在表 6.3 中汇总了大多数重要的、由 make 维护的变量。

表 6.2　Makefiles 中经常定义的用户变量

变量	内容
SRCS	源文件
INCLUDES	包含文件
OBJS	目标文件
LIBS	库
CC	C 编译器
CPP	C 预处理器
CFLAGS	C 编译器的标志
LFLAGS	链接器的标志
INSTALL	安装程序
SHELL	命令外壳

表 6.3　由 make 维护的变量

变量	内容
$$	$ 符号
$@	正在制作的文件的名称
$?	比目标更早的文件的名称
$>	目标依赖的所有源文件的清单
$<	目标文件转换自的文件的名称
$*	目标文件的前缀，不含目录部分以及后缀

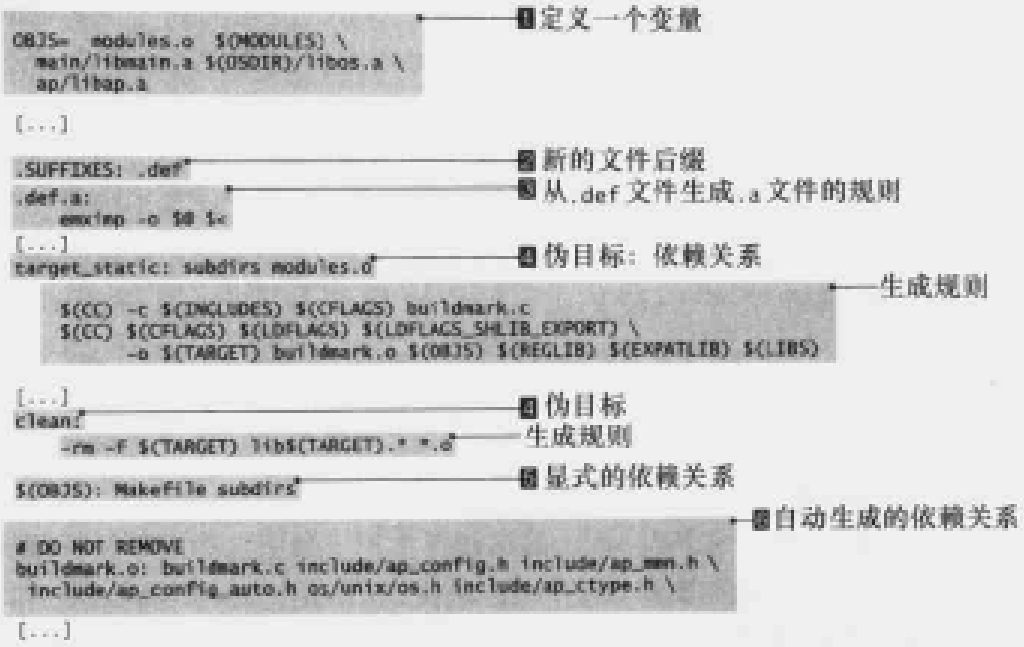


图 6.9 apache Web 服务器的制作文件(节选)

制作文件(makefile)中的目标不一定是编译过程中生成的文件。在制作文件中,常常会定义任意名称的伪目标(Pseudo-target),用以指定编译的顺序,当将具体的目标(伪目标)作为参数传递给 make,或作为另外目标的依赖关系时,这种编译顺序就会得到执行。一些常见的伪目标名称包括 all 或 build——生成项目中的所有目标,doc——生成项目的文档,depend——生成依赖信息,test——运行编译后的回归测试,install——执行安装步骤,以及 clean——移除编译过程中生成的文件。

不是所有的依赖关系都需要跟上一条编译规则(图 6.9:5)。没有规则的依赖关系用来规定如果某个源文件比目标更新,则目标文件需要重新编译(使用另外的规则来生成该目标文件)。图 6.9:5 中的依赖关系用来表达,一旦制作文件发生更改,目标文件需要重新编译,估计是因为制作文件中的更改或许涉及编译期间配置标志的更改。没有规则的依赖关系还常用来指定目标文件依赖哪些头文件,当 C 代码中包含的头文件被修改后,对应的 C 文件需要重新编译。

在大型项目中,每个 C 或 C++ 文件一般包括数十个头文件,某些头文件可能会递归地包括其他文件。手动地生成恰当的依赖关系来反映这种情况十分困难。幸运的是,可以使用多种方式自动地生成这些信息。GNU 的 C 编译器提供了一个-M 选项,用来生成此类依赖关系,在 BSD 系统中,mkdep 工具提供类似的功能。自动生成的依赖关系一般添加到制作文件的末尾,在一个恰当的标记之后(图 6.9:6)。而后,由制作文件中的一条规则(一般称为 depend)负责在源代码或项目的配置发生更改之后,维护和刷新这些依赖关系。

最初的 make 程序取得了极大的成功,进而繁衍出大量的子孙。新一代的 make 程序一般都支持条件化处理,包括文件和系统广度的规格说明,并维护庞大的内部隐式规则集。某些版本的 make 能够跨多台机器并行地运行编译过程,以提高处理速度。大型的项目一般使用这些特性将跨多个子项目和目录的编译过程的配置集中处理。例如,X Window 系统使用 imake 对制作文件执行预处理,从而,系统广度的定义和工具能够跨不同的子项目一致地

使用。类似地，FreeBSD 系统中的 make 工具在处理一个本地文件之前，会读取/usr/share/mk 中维护的常用定义和默认规则；从而，使得每个项目的制作文件都简洁有效到了极致。

依赖关系并非一定要作为制作文件的一部分；有时，它们会作为编译过程显式说明的一部分，以过程化的方式描述出来。强制规格说明的例子可以在 apache Web 服务器的编译过程中找到：它在编译和链接主可执行程序之前，先在所有的子目录中显式地执行编译步骤。①

```
target_static: subdirs modules.o
    [...]
subdirs:
    @ for i in $ (SUBDIRS); do \
        [...]
        cd $ $ i && $ (MAKE) [...]
    done
```

上面的例子中，编译 target_static 时总是会执行 subdirs 伪目标。subdirs 目标使得 make 在 SUBDIRS 变量中出现的每个目录中执行。编译过程中的过程性规格说明经常用来拆开一个长的依赖链，提供一种更为模块化的描述。make 进程与它的子孙之间的信息通过环境变量自动传递：所有变量定义都作为环境变量输出给子孙，所有环境变量在 make 进程中都作为变量定义出现。过程性的规格说明还常用在不支持制作文件的平台，或需要声明性说明的特殊特性。以在 Windows 平台上编译 apache 的制作文件为例。两个目标将编译和安装 apache 的庞大命令序列作为它们的规则。②

随着制作文件越来越大，它们变得难以阅读和推理。幸运的是，有一种方便的快捷方式，可以用来解开一个复杂的编译过程。您可以使用-n 开关预演 make，检查大型编译过程中的各个步骤。在进行预演时，make 将显示它要执行的命令，但不会真的执行它们。从而，您可以将 make 的输出重定向到一个文件中，在您喜欢的编辑器中打开该文件仔细地检查它。这时，在编译清单中查找字符串（代表性的工具、命令行参数、输入输出文件）的能力特别有用。

制作文件很少能在不同的操作系统间进行移植，因为它们经常包含平台相关的编译命令。新的编译管理工具，如 jam③ 和 ant④，在设计时就明确地考虑到操作系统的可移植性。jam 通过安装期间的配置，内部处理操作系统相关的不同，如目录路径分隔符，实现 jamfile（制作文件的 jam 等价物）的可移植性，并处理复杂的任务，如扫描 C 头文件的依赖关系。ant 使用一种不同的方案，它使用 XML 来表达编译任务，并用 Java 类来扩展其功能（参见图 6.10⑤）。但是，不同方式的背后，阅读对应文件的基本思想依旧是相同的。

练习 6.7　下载并编译一个相对大型的开放源码项目（比如 apache、Perl、X Window 系统或类 Unix 内核）。记录编译过程的每个步骤。

---

① apache/src/Makefile.tmpl:33-71

② apache/src/makefile.win:61-242

③ *http://www.perforce.com/jam/jam.html*

④ *http://jakarta.apache.org/ant/index.html*

⑤ jt4/webapps/manager/build.xml

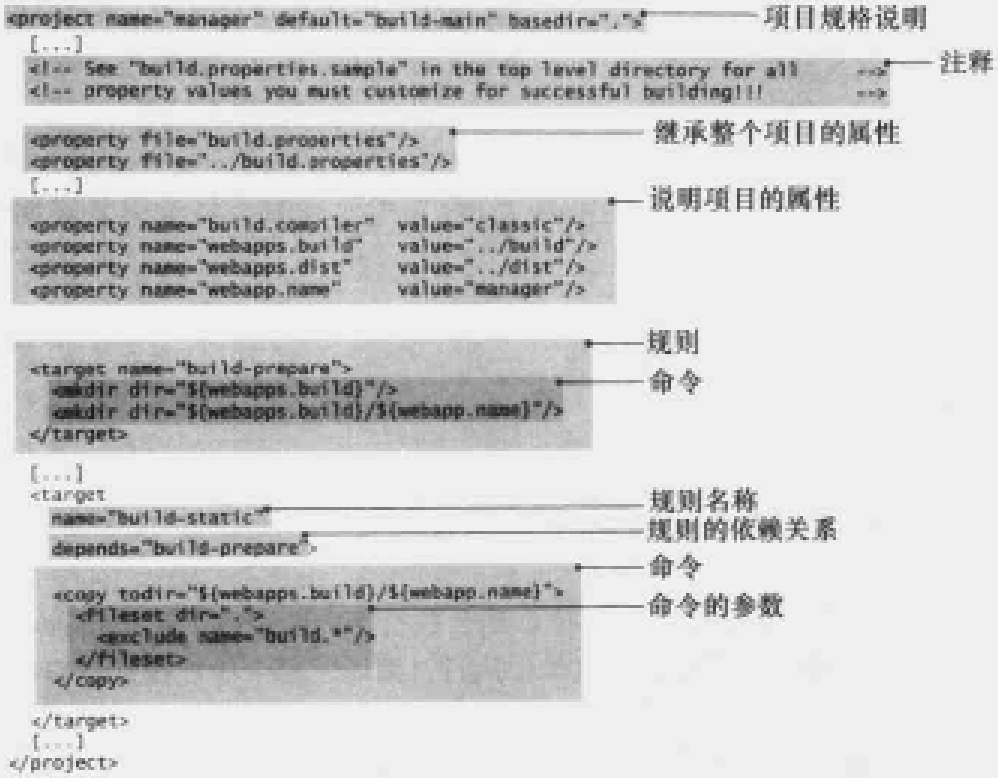


图 6.10 XML 形式的 ant 编译文件

练习 6.8 比较编译过程中声明性规格说明与过程性规格说明的使用。

练习 6.9 如果您正在使用集成开发环境，描述如何指定编译过程。评论规格说明的易读性、可维护性和可移植性。

练习 6.10 选择 make 程序的 4 个变体，比较它们提供的附加特性。

练习 6.11 阅读一种 make 程序的文档，找出如何创建隐式定义的规则的清单。选择其中 5 个规则，解释它们的运作。

6.4 配　　置

配置(configuration)可以控制的软件系统，允许开发者编译、维护和发展源代码的单一正式版本。只维护源代码的单一副本简化了更改和演化管理。通过使用适当的配置，相同的源代码体可以：

- 创建拥有不同特性的产品；
- 为不同的构架或操作系统构造产品；
- 在不同的开发环境下进行维护；
- 与不同的库链接；
- 使用运行期间指定的配置选项来运行。

系统的配置可以发生在编译期间，在这种情况下，它既会影响编译过程又会影响最终结果；也可以在运行期间动态地配置，这种情况下它只影响程序的运行。和其他大型系统的特性一样，您应该能够识别和推理配置过程中的各个构成部分。在编译期间实施的典型配置选项包括：产品支持的特性，编译的目标构架和操作系统，编译器及其选项，对头文件和库的需求，安装和管理方针，以及默认值。后两部分还经常在运行期间进行实施。

根据依赖关系隔离的原则，配置信息一般分为几个文件。影响系统如何编译的配置信息（例如，安装目录、工具名称和路径、是否在可执行文件中包括调试符号、应用的优化级别、使用的库和目标文件），经常以变量定义的形式表示在项目的制作文件中。①

```
ARCHDIR         = ..\lib\$(ARCHNAME)
COREDIR         = ..\lib\CORE
AUTODIR         = ..\lib\auto
[...]
LIBC            = msvcrt.lib
[...]
OPTIMIZE        = -01-MD -DNDEBUG
```

当系统使用集成开发环境（IDE），或其他编译管理工具进行编译时，相应的信息存储在对应的项目配置文件中，如下面的代码所示——取自一个 ant 配置文件②。

```
# - - - - - Compile Control Flags - - - - -
compile.debug= on
compile.deprecation= off
compile.optimize= on

# - - - - - Default Base Path for Dependent Packages - - - - -
base.path= /usr/loca

# - - - - - Jakarta Regular Expressions Library, version 1.2 - - - - -
regexp.home= ${base.path}/jakarta-regexp-1.2
regexp.lib= ${regexp.home}
regexp.jar= ${regexp.home}/jakarta-regexp-1.2.jar
```

类似地，影响实际程序如何编译的配置数据（运行期间文件的目录、影响条件编译特性的宏、可移植性相关的选项），经常以编译器宏定义的形式，存储在名为 config.h③ 的文件中。

```
/* Define if your OS maps references to /dev/fd/n to file descriptor n */
# define HAVE_DEV_FD 1

/* Default PATH (see comments in configure.in for more details) */
# define DEFAULT_PATH "/bin:/usr/bin:/usr/ucb"
```

① perl/win32/Makefile:333-388

② jt4/build.properties.sample:13-26

③ netbsdsrc/bin/ksh/config.h:192-199

```
/*  Include ksh features? (see comments in configure.in for more details) */
# define KSH 1
```

由于许多配置选项(尤其是那些影响操作系统和处理器构架可移植性的选项),在不同的系统中是通用的,因此,人们已经开发出大量的方法和工具,自动化这些配置选项的生成工作。作为一个具体的例子,众多GNU autoconf工具生成的配置系统在编译期间都遵循图6.11例举的过程。配置过程的第一部分有一个外壳脚本,称为configure,它用来确定系统的许多特性。检测函数的精确排序保证了比较复杂的函数能够建立在早期确定的结果之上。特别地,元件的检查遵照下面的次序:可用的程序、库、头文件、typedef、结构、编译器特性、库函数和系统服务。这项系统探测的结果存储在人能够阅读的文件中,名为config.log,它们是以可以被重新读取的格式进行存储的,可以避免config.cache中极为耗时的探测序列,同时还可以用于config.status文件中的最终配置。在配置过程的第二步,config.status文件读取两个参数化的模板文件,config.h.in和Makefile.in,生成实际的Makefile和config.h文件。

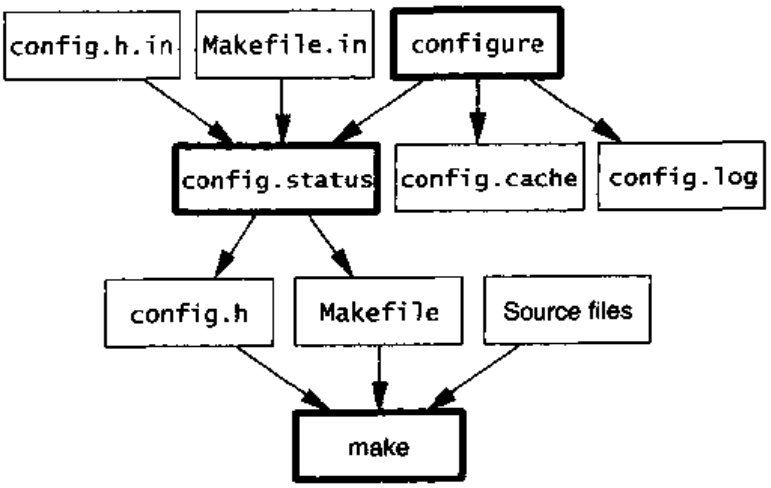


图6.11 GNU autoconf的配置过程

在运行期间,可移植的程序要么从专门的配置文件①②中读取配置数据

```
/*
*  set_profile reads $HOME/.indent.pro and ./.indent.pro
*  and handles arguments given in these files.
*/
void
set_profile()
{
    FILE * f;
    char fname[BUFSIZ];
    static char prof[] = ".indent.pro";
```

① netbsdsrc/usr.bin/indent/args.c:284-303

② 您能看出这段代码有发生缓冲区溢出的可能性吗?

```
    sprintf(fname, "%s/%s", getenv("HOME"), prof);
    if ((f = fopen(option_source = fname, "r")) != NULL) {
      scan_profile(f);
      (void) fclose(f);
    }
    if ((f = fopen(option_source = prof, "r")) != NULL) {
      scan_profile(f);
      (void) fclose(f);
    }
```

要么通过处理用通用解释型语言，如 Tcl/Tk 或 Perl，编写的配置命令。

另外，在类 POSIX 平台上运行的程序，经常使用 getenv 函数从环境变量中得到配置数据。①

```
arcname = getenv("TAPE");
```

Java 程序使用 System. getProperty 方法②。

```
String useNamingProperty = System.getProperty("catalina.useNaming");
```

Windows 程序使用 SDK 中的 RegOpenKey，RegQueryValueEx 和 RegCloseKey 函数查询 Windows 的注册表。③

```
bSuccess = RegOpenKey(HKEY_LOCAL_MACHINE,newkey,&hk);
[...]
bSuccess = RegQueryValueEx(hk,/*  subkey handle * /
    "Bind",                   /*  value name * /
    NULL,                     /*  must be zero * /
    NULL,                     /*  value type not required * /
    (LPBYTE) &bindservicenames,/*  address of value data * /
    &sizeofbindnames);        /*  length of value data * /
[...]
RegCloseKey(hk);
```

X Window 系统中运行的程序使用诸如 XtGetApplicationResources。④

```
static struct resources {
    Boolean rpn;     /*  reverse polish notation (HP mode) * /
    Boolean stipple;/*  background stipple * /
    Cursor  cursor;
} appResources;

# define offset(field) XtOffsetOf(struct resources, field)
static XtResource Resources[] = {
```

① netbsdsrc/bin/pax/options. c:800

② jt4/catalina/src/share/org/apache/catalina/core/StandardContext. java:800

③ netbsdsrc/usr. sbin/xntp/xntpd/ntp_io. c:2355-2377

④ XFree86-3. 3/contrib/programs/xcalc/xcalc. c:95-131

```
{"rpn", "Rpn", XtRBoolean, sizeof(Boolean),
    offset(rpn), XtRImmediate, (XtPointer) False},
{"stipple", "Stipple", XtRBoolean, sizeof(Boolean),
    offset(stipple), XtRImmediate, (XtPointer) False},
{"cursor", "Cursor", XtRCursor, sizeof(Cursor),
    offset(cursor), XtRCursor, (XtPointer)NULL}
};
[...]
XtGetApplicationResources(toplevel, (XtPointer)&appResources, Resources,
        XtNumber(Resources), (ArgList) NULL, ZERO);
```

在运行期间读入的配置数据可以自由地被终端用户操作。因此，程序应该像对待任何其他用户输入一样，仔细地对它进行验证，同时，在较高安全特权下运行的程序，应该保证不让它们的任何函数允许无特权的用户使用。作为一个反面的例子，早期实现的共享库允许敌意的用户自行配置搜索路径，让拥有特权的程序载入他们（恶意用户）自己制作的系统库文件。之后，这些库就可以用来操作拥有特权的程序，绕过安全限制。

练习 6.12 选择本书配套盘中的一个程序，在程序支持的环境中对程序进行配置、编译并运行。分析配置过程的输出文件，并手动地检验 4 个不同的配置选项。

练习 6.13 阅读一个配置脚本的源代码，说明如何确定两个不同的配置选项。

练习 6.14 根据它们提供的功能性和可用性，比较我们分析过的运行期间配置机制。

练习 6.15 Microsoft Windows 的早期版本用 Windows 目录中以.ini 为扩展名的初始化文件（initialization file），支持运行期间配置。它们是 ASCII 文件，划分成多个节，包含键/值项。将这种方案与当前基于注册表的实现进行比较。

6.5 修订控制

我们可以将系统的源代码想像成在空间和时间两个方向上延伸。代码，组织成文件和目录的形式，占据空间，同时，同一代码还随着时间的推移不断演化。修订控制系统可以跟踪代码的演化，标记重大的事件，并记录更改背后的原由，允许我们查看和控制时间要素。本节中，我们详述如何使用修订控制系统来理解代码的时间要素。当前正在使用的修订控制系统有很多种，比如：SCCS，RCS，CVS 和 Visual Source Safe。它们大都具有类似的功能；我们将以 CVS 为例，因为它在开放源代码社团中被广泛采用。

在深入研究如何应对修订控制系统控制下的项目之前，最好先了解一下，在典型的开发过程中如何跟踪修订的概况。修订控制系统将与文件相关的所有历史信息保存到一个中心储存库中。每个文件的历史都保存在一系列可辨别的版本中，一般使用自动生成的版本号，如 3.9.2.11.还经常使用开发人员为标记系统开发过程中某个里程碑面选定的关联标记（例如 R4.3-STABLE-p1）。在项目发布之前，版本的里程碑用来标记开发的目标（例如，提供指

定的功能、干净的编译、成功的回归测试)；在发布之后，版本一般跟踪改进性的更改和 bug 的修复。当开发人员在文件的某个版本上完成工作后，他或她会将文件交付(commit)或检入(check in)储存库。修订控制系统保存了文件所有版本的全部细节(大多数系统通过存储版本之间的差异来节省空间)，同样还有每次交付操作的元数据(一般是日期、时间、开发人员的姓名和对更改的注释)。某些开发模型允许开发人员在使用文件时将它锁定(lock)，从而禁止同时更改。其他的系统则提供一种协调方式，当冲突发生时，可以协调互相冲突的更改。有时，针对给定文件的开发工作可能被分为两个不同的分支(branch)(例如，一个稳定分支和一个维护分支)；后期，两个分支可能会再次结合。有些系统由大量的文件组成，为了协调整个系统的发布工作，许多系统提供一种标记方式，用一个标识系统版本的符号名称，标记属于该系统的一系列文件。从而，我们能够确认组成完整系统的每个文件的版本。

图 6.12 是一个文件版本树以及相关的符号名称。虚线箭头表示 UML 的可溯依赖关系(trace dependency)，它表明箭头指向的结点是箭头尾部结点的历史祖先。例如，文件 cat.

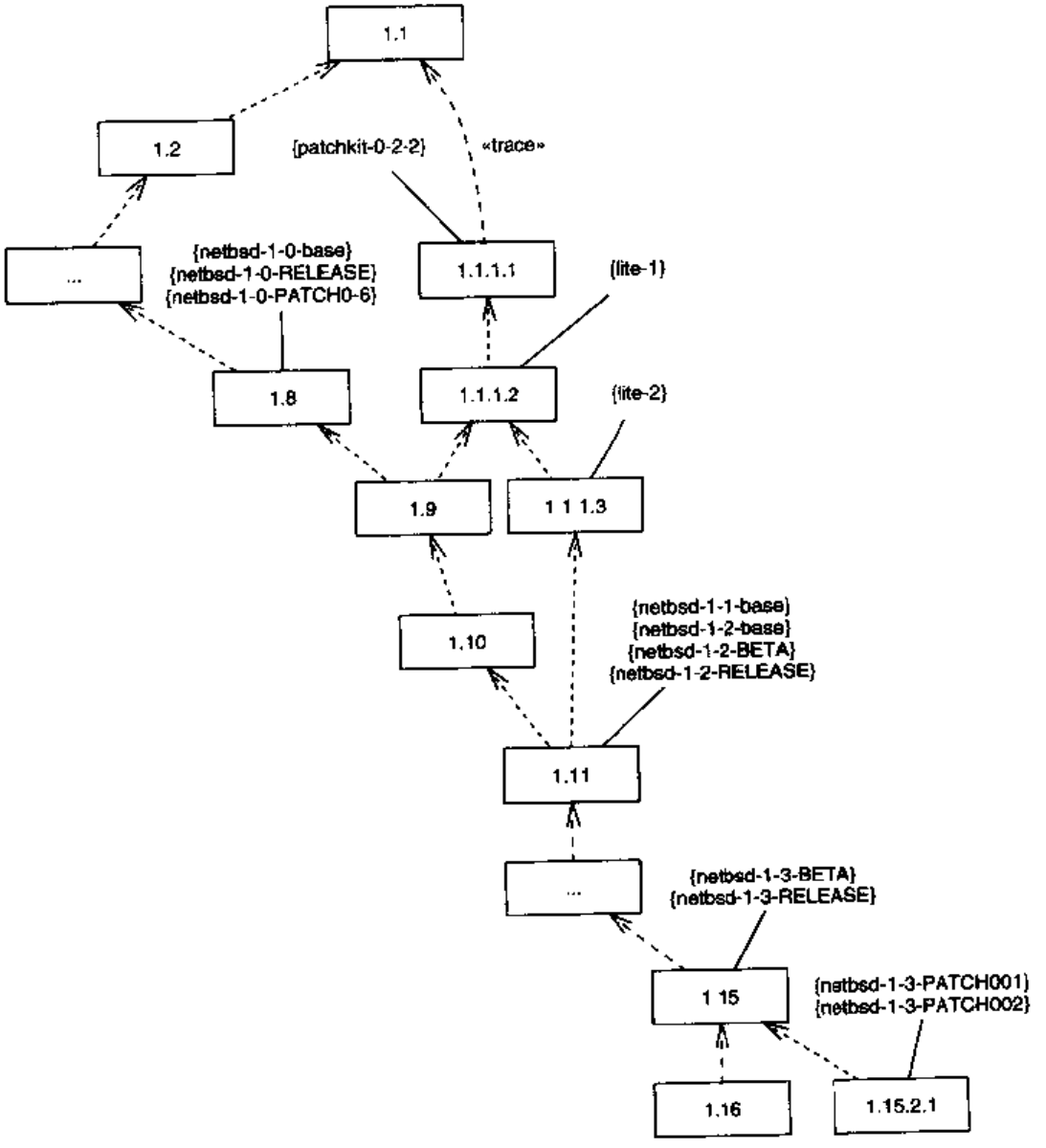


图 6.12　cat.c 的修订树和符号名称

c 的 1.1.1.2 版本是 1.9 版本和 1.1.1.3 版本的一个祖先;相应地,1.9 版本基于 1.8 版本和 1.1.1.2 版本。花括号中是标示完整 BSD 系统特定版本的符号版本名称。因此,我们可以得知,BSD lite-2 发行程序中包括 cat.c 的 1.1.1.3 版本。

在分析代码时,您肯定希望确保您阅读的确实是最新的版本。修订控制系统提供从储存库中获取源代码的最新版本的方式。例如,在 CVS 中,使用下面的命令

```
cvs update -d -P
```

可以从当前的源码树中获得每个文件的最新版本。-d 标记指定创建那些在储存库中存在,但工作目录中缺失的目录,-P 标记指定在更新后移除那些空的目录。

有时,您不想分析源代码的最新版本(或许正处在开发过程中,不断发生变动),而想分析一下用在特定软件版本中的稳定版本。您可以使用它的符号标记(symbolic tag)来获取您希望使用的版本。例如,您可以使用下面的 CVS 命令,获取组成 sed 流编辑器的所有文件对应 NetBSD netbsd-1-2 软件版本的版本。

```
cvs co -rnetbsd-1-2 basesrc/usr.bin/sed
```

您还可以使用如下的命令请求给定文件的特定版本。

```
cvs co -r1.17 basesrc/usr.bin/sed/compile.c
```

版本控制系统的一项重要功能是帮助用户标识他们正在使用的文件。这项功能通过在源文件中嵌入指定格式的关键字来完成。当系统从储存库中取出文件时,它将相应的数据,比如文件的版本、名称或完整路径,添加到每个关键字。下面是一些出现在源文件的注释和字符串中的扩展关键字。①

```
/*  $NetBSD: cat.c,v 1.15 1997/07/20 04:34:33 thorpej Exp $  */
[...]
#ifndef lint
#if 0
static char sccsid[] = "@ (#)cat.c 8.2 (Berkeley) 4/27/95";
#else
__RCSID("$NetBSD: cat.c,v 1.15 1997/07/20 04:34:33 thorpej Exp $");
#endif
#endif /* not lint */
```

表 6.4 列出了,在 3 种不同的常用修订控制系统中,最常使用的关键字标记。关键字可能出现在注释中(一般在文件的头部),用来帮助源代码的阅读者,或者用在字符串中,从而可以将它们编译到可执行文件中。修订控制系统提供一些命令,可以分析源文件、目标文件或可执行文件中的数据,并显示找到的关键字。识别 CVS,RCS 和 Visual Source Safe 关键字的命令是 ident。

① netbsdsrc/bin/cat/cat.c:1-51

表 6.4　用于修订控制系统的文件标记

关键字	内容
RCS 和 CVS	
$ Author $	检入修订文件的用户的登录名
$ Date $	修订文件检入的日期和时间
$ Header $	路径名、修订文件、日期、时间、作者、状态、锁定者
$ Id $	同 $ Header $；只给出文件名，不带路径
$ Locker $	锁定修订文件的用户的登录名
$ Log $	检入时提供的信息，前面有一个标题
$ Name $	用来检出修订文件的符号名称
$ RCSfile $	不带路径的 RCS 文件名
$ Revision $	赋予修订文件的修订号
$ Source $	RCS 文件的全路径名
$ State $	修订文件所处的状态
SCCS	
%M%	模块名(Module name)
%R%	发布(Release)
%L%	级别(Level)
%B%	分支(Branch)
%S%	顺序(Sequence)
%I%	发布、级别、分支和顺序
%D%/%H%	当前 get 的日期
%T%	当前 get 的时间
%E%/%G%	新的变化创建的日期
%U%	新的变化创建的时间
%F%/%P%	SCCS 文件/全路径名
%Y%	模块类型
%C%	文件中当前行的行号
%Z%	由 what 使用的四字符串@(#)
%W%	%Z%%M%%tab%I%的简写
%A%	%Z%%Y% %M% %I%%Z%的简写
Visual Source Safe(VSS)	
$ Archive: $	VSS 存档文件的位置
$ Author: $	最后对文件做出更改的用户
$ Date: $	最后一次检入发生的日期和时间
$ Header: $	Logfile, Revision, Data, Author 的简写
$ History: $	文件的历史，VSS 格式
$ JustDate: $	仅仅是日期，没有时间
$ Log: $	文件的历史，RCS 格式
$ Logfile: $	和 $ Archive: $ 相同
$ Modtime: $	最后一次修改发生的日期和时间
$ NoKeywords: $	接下来的关键字不进行关键字展开
$ Revision: $	VSS 版本号
$ Workfile: $	文件名

```
$ ident cat.c
cat.c:
     $ NetBSD: cat.c,v 1.28 2001/09/16 12:12:13 wiz Exp $
     $ NetBSD: cat.c,v 1.28 2001/09/16 12:12:13 wiz Exp $
```

SCCS 系统中使用的相应命令是 what。

```
$ what cat.c
cat.c:
        Copyright (c) 1989, 1993
      cat.c 8.2 (Berkeley) 4/27/95
```

可以使用相关的命令，显示可执行文件中的修订标识关键字，从而将可执行文件与它的源代码匹配起来。在大型的程序中，通过使用这种方法，可以获得用来生成可执行文件的所有源代码的清单，包括库在内。

```
$ NetBSD: crt0.c,v 1.9 1999/07/02 15:53:55 simonb Exp $
$ NetBSD: version.c,v 1.5 1997/10/19 05:04:07 lukem Exp $
$ NetBSD: aux.c,v 1.10 1998/12/19 16:30:52 christos Exp $
$ NetBSD: cmd1.c,v 1.13 2000/02/10 12:34:42 tron Exp $
$ NetBSD: cmd2.c,v 1.10 2000/02/10 12:34:43 tron Exp $
$ NetBSD: cmd30.c,v 1.11 1999/02/09 04:51:30 dean Exp $
$ NetBSD: cmdtab.c,v 1.8 1997/10/19 05:03:08 lukem Exp $
$ NetBSD: collect.c,v 1.20 2000/02/10 12:34:43 tron Exp $
$ NetBSD: dotlock.c,v 1.4 1998/07/06 06:51:55 mrg Exp $
$ NetBSD: edit.c,v 1.7 1997/11/25 17:58:17 bad Exp $
$ NetBSD: fio.c,v 1.12 1998/12/19 16:32:34 christos Exp $
$ NetBSD: getname.c,v 1.6 1998/07/26 22:07:27 mycroft Exp $
$ NetBSD: head.c,v 1.8 1998/12/19 16:32:52 christos Exp $
$ NetBSD: v7.local.c,v 1.10 1998/07/26 22:07:27 mycroft Exp $
$ NetBSD: lex.c,v 1.14 2000/01/21 17:08:35 mycroft Exp $
$ NetBSD: list.c,v 1.9 1998/12/19 16:33:24 christos Exp $
$ NetBSD: main.c,v 1.10 1999/02/09 04:51:30 dean Exp $
$ NetBSD: names.c,v 1.8 1998/12/19 16:33:40 christos Exp $
$ NetBSD: popen.c,v 1.9 1998/12/19 16:34:04 christos Exp $
$ NetBSD: quit.c,v 1.11 2000/02/10 12:34:43 tron Exp $
$ NetBSD: send.c,v 1.11 2000/02/10 12:34:44 tron Exp $
$ NetBSD: strings.c,v 1.6 1997/10/19 05:03:54 lukem Exp $
$ NetBSD: temp.c,v 1.7 1998/07/26 22:07:27 mycroft Exp $
$ NetBSD: tty.c,v 1.11 1999/01/06 15:53:39 kleink Exp $
$ NetBSD: vars.c,v 1.6 1998/10/08 17:36:56 wsanchez Exp $
```

修订控制系统保存每次修订的元数据(metadata)。在分析系统的源代码时，这项数据可以提供极为有用的信息。使用如下的命令可以获得所有修订元数据的完整清单。

```
cvs log cat.c
```

图 6.13 就是获取的相关数据，其中添加了注释。在 8.2 节中，我们概括一些启发式的

方法，您可以在记录变更的日志数据上应用这些方法，以获取有用的细节。部分变更注释可能会引用一个特定的数值码（图 6.13:1）。这个数值码一般标识 bug 跟踪数据库（比如 GNATS，Bugzilla 或 SourceForge Bug Tracker）中描述的某个问题。使用 bug 的跟踪编号，可以在数据库中找到特定的问题，得到该问题的完整描述，这些描述常常会指示如何重现该问题，以及解决该问题的方法。

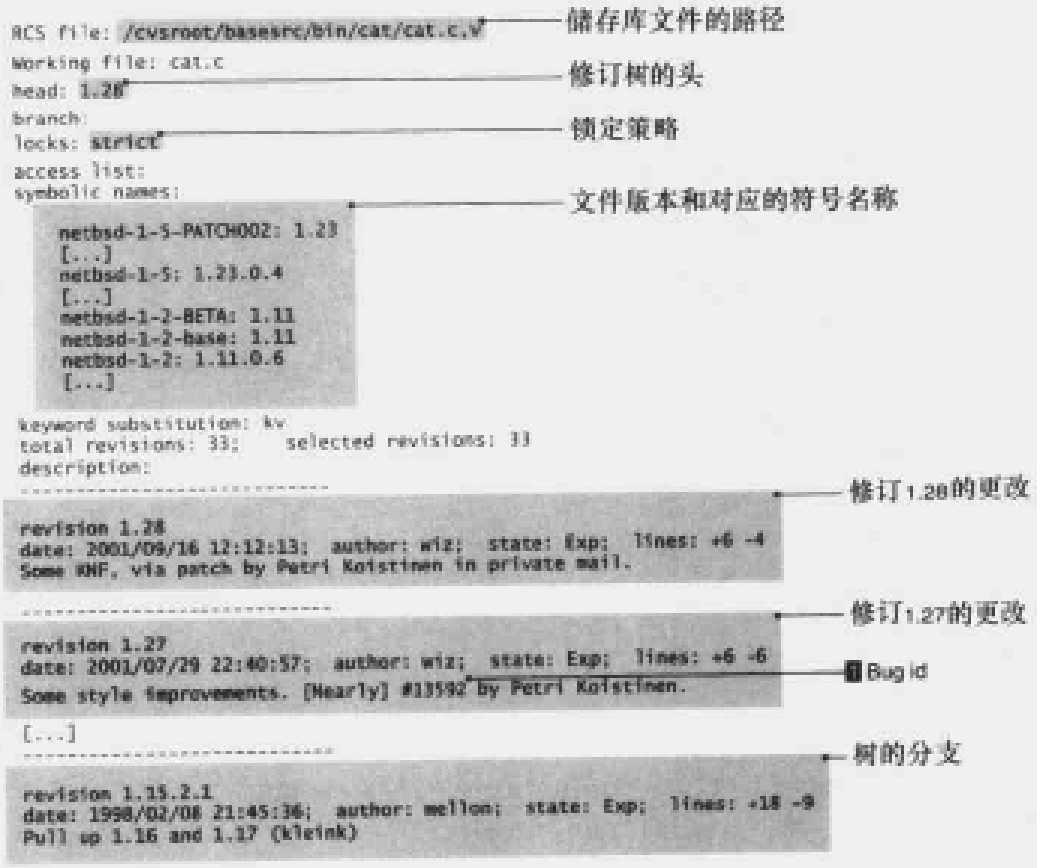


图 6.13　RCS/CVS 的日志输出

我们还可以用一些工具来处理大量的日志数据，获得统一的结果。例如，假定我们希望知道项目中哪些文件经历的修订最多（常常表明该处可能是个问题点），那么，我们可以用 Unix 的文本处理工具 awk，创建一个文件名和版本号的清单，然后依据第二版本号对清单进行按数字逆序排序。清单中前面的文件就是修订号最高的文件。

```
$ cvs log sed |
> awk '/^Working/ {w = $3}/^head:/{print &2,w}'|
> sort -n -t.+1 -r
> head
1.30 sed/process.c
1.20 sed/compile.c
1.14 sed/sed.1
1.10 sed/main.c
1.8 sed/Makefile
1.7 sed/defs.h
1.6 sed/misc.c
```

```
1.6 sed/extern.h
1.4 sed/POSIX
```

我们还可以使用修订控制系统的版本储存库，找出特定的变更是如何实现的。考虑下面的 cat.c 1.13 版本的日志项。

```
revision 1.13
date: 1997/04/27 18:34:33; author: kleink; state: Exp; lines: +8 -3
Indicate file handling failures by exit codes> 0; fixes PR/3538 from
David Eckhardt davide@ piper.nectar.cs.cmu.edu.
```

为了精确地得出文件处理失败现在是如何表示的，我们可以发布一条命令，检查(参见 10.4 节)1.12 版本和 1.13 版本之间的差异。

```
$ cvs diff -c -r1.12 -r1.13 basesrc/bin/cat/cat.c
Index: basesrc/bin/cat/cat.c
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
RCS file: /cvsroot/basesrc/bin/cat/cat.c,v
retrieving revision 1.12
retrieving revision 1.13
diff -c -r1.12 -r1.13
[...]
* * * * * * * * * * * * * * *
* * * 136,141 * * * *
- - - 136,142 - - - -
                        fp = stdin;
                else if ((fp = fopen(*argv,"r")) == NULL) {
                        warn("%s",*argv);
                        rval = 1;
                        ++argv;
                        continue;
                }
```

如上面的这段摘录所示，代码中添加了一行，当文件打开命令失败时，设置 rval 标志(大概是表示该命令的返回值)。

练习 6.16 您如何跟踪修订历史呢？如果您没有采用修订控制系统，请考虑采用一个。

练习 6.17 根据您手头的一些源代码，解释为什么可执行文件可能包含不同的标识标记(identification tag)。您如何解决这类问题？

练习 6.18 许多开放源码项目都开放基于 CVS 的修订控制储存库，向公众提供只读访问。找到这样一个项目，并在您的机器上创建该项目的一个本地副本。使用命令 cvs log，分析一个给定文件的日志(最好包含分支)，将符号标记(symbolic tag)与文件版本进行匹配。说明如何使用分支来管理连续的软件发行版本。

6.6　项目的专有工具

大型项目经常拥有独特的问题，并且拥有足够的资源构造专门的工具。定制编译工具用在软件开发过程的许多方面，包括配置、编译过程管理、代码的生成、测试和文档编制。在下面的段落中，我们将针对每项任务，提供一些具有代表性的工具。

操作系统内核的配置是一项特别复杂的任务，因为这项任务需要在数百个设备驱动程序和软件选项中进行选择(许多驱动程序和软件选项相互关联)，创建一个与具体硬件配置相匹配的定制内核。NetBSD 内核的配置工作由 *config*[①] 来处理，*config* 是一个工具程序，它读取描述系统选项的配置，创建系统中 I/O 设备的描述，以及用以生成特定内核的制作文件。config 的输入格式如下。[②]

```
#  CPU options
options   CPU_SA110   #  Support the SA110 core

#  Architecture options
options   OFW         #  Enale common Open Firmware bits
options   SHARK       #  We are a Shark

#  File systems
file-system FFS       #  UFS
file-system MFS       #  memory file system
file-system NFS       #  Network file system
[...]
#  Open Firmware devices
ofbus*      at root
ofbus*      at ofbus?
ofrtc*      at ofisa?
```

基于这样的输入，config 将生成一个制作文件，其中只包含特定配置中需要进行编译的文件，和文件 ioconf. c，它为选定的设备定义系统结构，如下所示。

```
/*  file systems * /
extern struct vfsops ffs_vfsops;
extern struct vfsops mfs_vfsops;
extern struct vfsops nfs_vfsops;
[...]

struct vfsops * vfs_list_initial[] = {
        &ffs_vfsops,
        &mfs_vfsops,
        &nfs_vfsops,
```

① netbsdsrc/usr. sbin/config

② netbsdsrc/sys/arch/arm32/conf/GENERIC

```
    [...]
    NULL,
};
```

专门工具还常常用来管理编译过程，尤其在项目移植到许多不同的平台，难以使用与特定平台相关的工具的情况下。例如，X Window 系统使用 imake[①]（一个定制编译制作文件的预处理器）来管理编译过程。imake 用来从一个模板、一套 C 预处理器宏函数和每个目录一个的输入文件，生成制作文件。它将与机器相关的内容（比如编译器选项、替换的命令名称和特殊的制作规则）与对各种编译项的说明分离开来。从而，可以根据一个系统广度的配置文件，对超过 500 个独立的制作文件进行剪裁。另外，X Window 系统的设计者，不能依赖于存在特定的依赖关系生成器，能够为系统生成 C 包含文件的依赖关系。因此，他们设计出一个项目专用的生成器——makedepend[②]。

到目前为止，项目专用工具的最通常的用途是生成特定的代码。因此，用工具动态地创建软件代码是编译过程的一部分。大多数情况下，这些代码使用的语言与系统的其他部分相同（例如，C），并且大都比较简单，一般由查找表或简单的 switch 语句组成。使用这些工具一般是为了在编译期间构造代码，以避免在运行期间执行等同的操作所带来的开销。这种方式下生成的代码常常更高效，并且消耗较少的内存。代码生成工具的输入可以是另外的文件（以文本形式表达所要生成代码的规格说明），简单的领域专用语言，甚至系统源代码的组成部分。以 IBM 3270 终端仿真器为例。它由 77 个文件组成，仿真器项目的大小适中。然而，它却用到 4 个不同的工具来创建 IBM 字符和键盘代码，与 Unix 工作站上对应内容之间的映射。图 6.14 描绘出这些工具在编译过程中的使用方式。两个不同的工具，mkastods[③] 和 mkdstoas[④]，分别创建 ASCII 到 EBCDIC（IBM 专用）显示字符和 EBCDIC 显示字符到 ASCII 的映射。之后，这些工具的输出文件，asc_disp. out 和 disp_asc. out，被包括到 C 文件 disp_asc. c[⑤] 中，这个文件将被编译成 tn3270 终端仿真器可执行文件的一部分。类似地，mkastosc[⑥] 用来创建从 ASCII 代码到 IBM 键盘扫描码的映射，mkhits[⑦] 用来创建从扫描码和 shift 键的状态到 IBM3270 功能和字符的映射。所有工具都是通过处理 Unix 键盘定义文件 unix. kbd[⑧]，IBM 3270 命令文件 hostctlr. h[⑨]，以及编辑功能定义文件 function. h[⑩] 和 function. c[⑪]（预处理为 TMPfunc. out）来生成代码。之后，这些工具生成的代码（astosc.

① XFree86-3.3/xc/config/imake
② XFree86-3.3/xc/config/makedepend
③ netbsdsrc/usr.bin/tn3270/tools/mkastods/mkastods.c
④ netbsdsrc/usr.bin/tn3270/tools/mkdstoas/mkdstoas.c
⑤ netbsdsrc/usr.bin/tn3270/api/disp_asc.c
⑥ netbsdsrc/usr.bin/tn3270/tools/mkastosc/mkastosc.c
⑦ netbsdsrc/usr.bin/tn3270/tools/mkhits/mkhits.c
⑧ netbsdsrc/usr.bin/tn3270/ctlr/unix.kbd
⑨ netbsdsrc/usr.bin/tn3270/ctlr/hostctlr.h
⑩ netbsdsrc/usr.bin/tn3270/ctlr/function.h
⑪ netbsdsrc/usr.bin/tn3270/ctlr/function.c

out 和 kbd. out)以结构数组的形式(用于查找表),包括在 C 文件中(astosc. c[①] 和 inbound. c[②])。例如,kbd. out 将会包含下面类型的键映射。

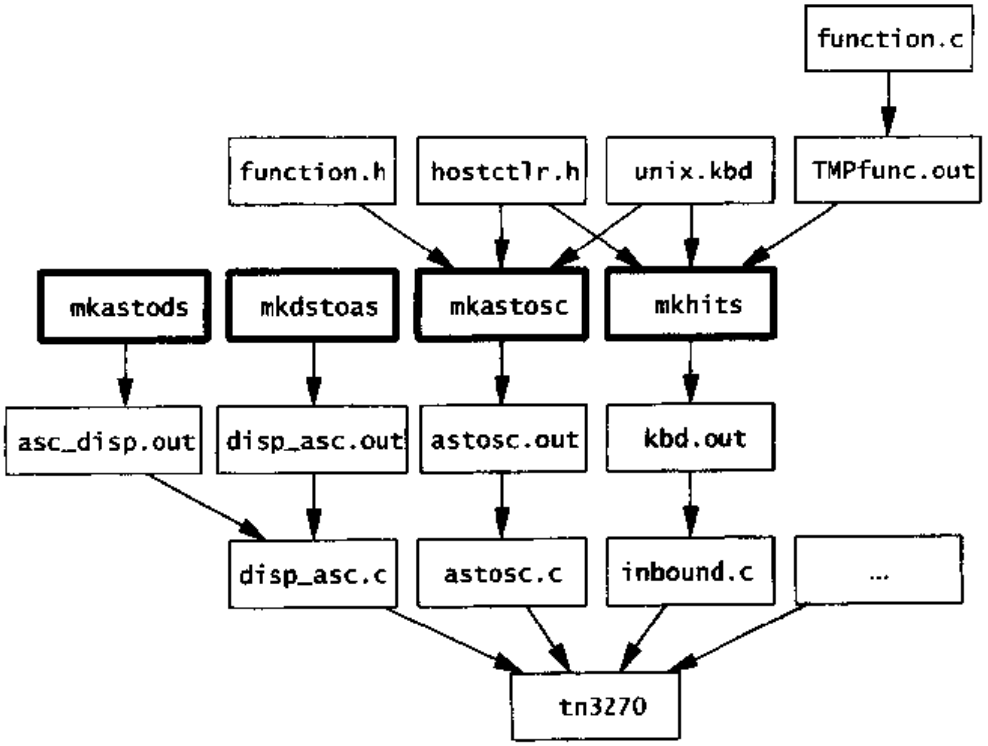


图 6.14　用在生成 IBM 3270 终端仿真器中的工具

```
struct hits hits [] = {
        { 0,{{undefined},{undefined},{undefined},{undefined}
        }},
        [...]
        {70,{/*  0x05 * /
              {FCN_ATTN},
              {undefined},
              {FCN_AID,AID_TREQ},
              {undefined},
        }},
        {65,{/*  0x06 * /
              {FCN_AID,AID_CLEAR},
              {undefined},
              {FCN_TEST},
              {undefined},
        }},
```

项目专用工具还经常用于对产品进行回归测试。一个值得注意的例子是,Perl 语言的

① netbsdsrc/usr. bin/tn3270/api/astosc. c

② netbsdsrc/usr. bin/tn3270/ctlr/inbound. c

实现中使用的测试套件(Test suite)。该套件由超过 270 个测试用例(test case)组成。由驱动程序 TEST[①] 运行每项测试,并报告测试的结果。有趣的是,测试的驱动程序也是用 Perl 编写的,只不过采取了特殊的保护措施,避免了测试过程中可能失败的构造。

最后,定制的工具常常用来自动完成项目文档的编制工作。javadoc 程序,现在作为一种通用工具随 Java SDK 一同分发,它最初极有可能是一个为 Java API 创建文档的工具。javadoc 分析一系列 Java 源文件中的声明和文档性质的注释,生成一系列的 HTML 页,描述类、内部类、接口、构造函数、方法和字段。图 6.15 中含有 javadoc 专用注释的 Java 代码,就是一个具有代表性的例子。[②]

```
/**                                                                    ← 方法说明
 * Retrieves the value of a JDBC <code>CHAR</code>, <code>VARCHAR</code>,
 * or <code>LONGVARCHAR</code> parameter as a <code>String</code> in
 * the Java programming language.
 * <p>
 * For the fixed-length type JDBC CHAR, the <code>String</code> object
 * returned has exactly the same value the JDBC CHAR value had in the
 * database, including any padding added by the database.
 * @param parameterIndex the first parameter is 1, the second is 2,      ← 参数说明
 * and so on
 * @return the parameter value. If the value is SQL NULL, the result    ← 返回值
 * is <code>null</code>.
 * @exception SQLException if a database access error occurs            ← 引发异常
 */
public String getString(int parameterIndex) throws SQLException;       ← 声明
```

图 6.15 嵌入在 Java 文件中的 javadoc 注释

Perl 程序,由一群热心的工具构建者为我们开发而成,也包含一系列基于 pod 标记语言的工具,可以用它们来创建各种不同格式的文档。pod 工具[③]能够以 LaTex、Unix 手册页、纯文本和 HTML 格式创建输出。和 javadoc 相同,它们也是从最初的有限应用发展壮大成通用的文档工具。常常用定制工具来管理的另一类文档是错误消息。它们一般散布在程序代码的大部分区域。有些项目开发了专用工具,用来在代码中找出错误消息(例如,通过查找对创建它们的函数的调用),并生成有序的最终用户文档——常常通过在消息后面加上特殊格式的注释。

练习 6.19 分析本书配套盘中的内核配置程序 config[④],记录它能够配置系统的哪些方面。

练习 6.20 在您喜欢的集成开发环境中,如何描述安装过程呢?说说所采用的方案有什么优点和缺点。

练习 6.21 建议一种方法,找出本书配套盘中的代码生成工具。找出 5 个这类工具,说

① perl/t/TEST
② hsqldb/src/org/hsqldb/jdbcPreparedStatement.java:837-851
③ perl/pod
④ netbsdsrc/usr.sbin/config

明它们所服务的目的。

练习 6.22　分析 Perl 的自动测试处理程序 TEST[①] 以及一个测试用例，例如，array.t[②]。说明应该怎样编写测试用例，以及在回归测试期间如何处理测试用例。

练习 6.23　将您的简历转换成 Perl pod 格式。使用 Perl 的 pod 处理工具，将它转换成 HTML 和纯文本。评论这种方案的适用性。

6.7　测　　试

设计良好的项目，都会预先为测试系统的全部或部分功能提供相应的措施。这些措施可能隶属于一份经过深思熟虑，用来验证系统运作的计划，也可能是系统的开发者在实现系统的过程中实施的非正式测试活动的残余。作为源代码阅读活动的一部分，我们首先应该能够识别并推理测试代码和测试用例，然后使用这些测试产物帮助理解其余的代码。在下面的段落中，我们将分析许多常见的不同类型的测试代码。

最简单的测试代码是一条生成日志(logging)或调试(debugging)输出的语句。开发人员一般用这种语句测试程序的运作是否与他们的预期相同。程序的调试输出可以帮助我们理解程序控制流程和数据元素的关键部分。此外，跟踪语句所在的地点一般也是算法运行的重要部分。大多数程序中，调试信息一般输入到程序的标准输出流或文件中。[③]

```
# ifdef DEBUG
    if (trace == NULL)
      trace = fopen("bgtrace", "w");
    fprintf(trace, "\nRoll: %d %d%s\n", D0, D1, race ? " (race)" : "");
    fflush(trace);
# endif
```

请注意，在上面的代码中，如何用 DEBUG 宏来控制特定的代码是否编译到最终可执行文件中。系统的产品版本一般在编译时不会定义 DEBUG 宏，从而略去了跟踪代码和它生成的输出。

在后台运行的程序(例如，Unix 守护程序或 Microsoft Windows 服务)需要用一种更为复杂的方式来生成跟踪信息，尤其是在同一程序的多个实例或线程同时运行的情况下。在 Unix 系统中，syslog 库函数用来将信息添加到适当的系统日志。[④]

```
syslog(LOG_DEBUG, "Successful lookup: %d , %d : %s\n",
    lport, fport, pwp -> pw_name);
```

在 Microsoft Windows 中用来生成日志信息的等价函数是 ReportEvent，而 Java 程序常常

① perl/t/TEST
② perl/t/op/array.t
③ netbsdsrc/games/backgammon/backgammon/move.c:380-385
④ netbsdsrc/libexec/identd/parse.c:372-373

使用免费的 log4j 库来组织和管理日志信息的高效生成。

太多的跟踪信息可能会掩盖至关重要的信息，或者会将程序的运行速度降到根本没办法使用的程度。因此，程序中常常定义一个称为调试级别(debug level)的整数，它用来过滤生成的消息。①

```
if (debug > 4)
    printf("systime: offset %s\n", lfptoa(now, 6));
```

在上面的例子中，该消息只在调试级别大于 4 的情况下才出现；更高的调试级别表示程序的输出更为冗长详尽。可以用程序的选项、信号或配置标志来改变程序的调试级别。在支持异常的语言中，比如 Ada，C++，C# 和 Java，调试信息经常作为异常处理代码的一部分。②

```
try {
    if (servlet != null) {
      wrapper.deallocate(servlet);
    }
} catch (Throwable e) {
    log(sm.getString("standardWrapper.deallocateException",
            wrapper.getName()), e);
```

我们分析的跟踪语句能够产生丰富的输出，但大多数情况下，它们并不能告诉我们输出是否正确。只在很少的情况下，调试语句会显示正确的期望值。③

```
printf("free %u bytes @%lx, should be <= %u\n",
size, (u_long)ptr, a -> size);
```

更为主动的方案是，在程序代码中测试各种在程序执行时应该为 true 的条件。这些条件，称为断言(assertion)，基本上是一些语句，这些语句指定当程序以开发者预想的方式工作时期望得到的结果。许多语言和库都能够在对程序结构影响最小的情况下，测试断言，并在断言失败时，自动生成调试输出(常常还会终止程序)。可以用断言来检验算法运作的步骤、函数接收的参数、程序的控制流程、底层硬件的属性和测试用例的结果。C 和 C++ 库都使用 assert 进行断言。

下面的这段代码演示了断言在检验正则表达式模式匹配算法中的应用。④

```
if (dp != NULL)
    break;
/*  uh-oh... we couldn't find a subexpression-level match * /
assert(g -> backrefs); /*  must be back references doing it * /
assert(g -> nplus == 0 || m -> lastpos != NULL);
```

上面的例子中，在执行了模式匹配(pattern-matching)函数近 100 行代码后，算法没有找到子表达式级别(subexpression-level)的匹配，可能是由于反向引用(back reference)的存在。

---

① netbsdsrc/lib/libntp/systime.c:216-217

② jt4/catalina/src/share/org/apache/catalina/core/StandardWrapperValve.java:295-301

③ netbsdsrc/sys/arch/powerpc/stand/ofwboot/alloc.c:222-223

④ netbsdsrc/lib/libc/regex/engine.c:244-249

程序员以断言的形式指出了这种情况下他期待变量应该拥有的值。接下来是另外的 50 行代码;此时,断言用来检查算法的运行。可以使用对算法进行检验的断言来证实您对算法运作的理解,或将它作为推理的起点。

断言还经常用来检查函数接收的参数的值。①

```
void
pmap_zero_page(register vm_offset_t phys)
{
    [...]
    assert(phys != vm_page_fictitious_addr);
```

类似的断言有时还用来检验函数的返回值。②

```
static char *
aalloc(int size, int align)
{
    [...]
    assert(p > &last);
    assert(p <= (char * ) RAMSIZE);
    return top = p;
}
```

对函数参数和结果的断言经常记录了函数的前置条件(precondition)和后置条件(postcondition)。

使用 false 常量作为参数的断言,用来标记正常情况下程序的控制流不应该到达的点。③

```
switch (mappad) {
case BitmapFormatImageRectMin:
    [...]
case BitmapFormatImageRectMaxWidth:
    [...]
case BitmapFormatImageRectMax:
    [...]
    break;
default:
    assert(0);
}
```

在上面的例子中,assert 语句表示 mappad 应该与其中的一个 case 常量匹配。

断言语句还用来检验底层硬件的属性或编译过程。下面的语句检验整数至少占用 4 个字节的存储空间。④

---

① netbsdsrc/sys/arch/alpha/alpha/pmap.c:2402-2410

② netbsdsrc/sys/arch/i386/netboot/ne2100.c:50-66

③ XFree86-3.3/xc/lib/font/Speedo/spglyph.c:84-120

④ netbsdsrc/usr.sbin/amd/amd/amd.c:319

```
assert(sizeof(int) >= 4);
```

最后，断言有时还用来实现完整的测试用例。下面C程序中的main函数完全由对函数调用的测试，以及相应的assert语句构成。①

```
int
main()
{
    assert(fpgetround() == FP_RN);
    assert(FLT_ROUNDS == 1);
    assert(fpsetround(FP_RP) == FP_RN);
    [...]
    assert(fpsetround(FP_RN) == FP_RZ);
    assert(fpgetround() == FP_RN);
    assert(FLT_ROUNDS == 1);
    exit(0);
}
```

我们可以将测试整个函数的断言作为每个给定函数的规格说明。

在分析代码时，要注意，assert宏中的表达式，在程序作为“产品”版本进行编译时，即NDEBUG宏设为1，一般不会进行编译。这种情况下，有副作用的断言（比如上面例子中对fpsetround的调用）可能会与您所预期的行为有所不同。

Java程序常常使用KentBeck和Erich Gamma的JUnit框架，将断言组织成完整的测试用例。JUnit支持测试数据的初始化、测试用例的定义、测试套件的组织和测试结果的收集。它还提供一个执行测试用例的GUI工具，TestRunner。图6.16②③中列出的是JUnit测试设置中最常见的部分代码。测试用例都被组织成派生自TestCase类的子类。TestCase类中的字段用来保存不同的测试用例都要用到的数据（图6.16:1）。setUp方法负责初始化字段的值，tearDown方法负责销毁专为特定测试用例设立的支持。测试用例都是以test开头的单独方法。每个测试用例内部，都有许多不同类型的assert方法，用来检验获取的结果。一个类中实现的多个测试用例可以归结成一个测试套件，名为suite；在此（图6.16:2），套件的构造函数从类中收集了所有以test开头的公开方法。

在没有测试框架可供使用的情况下，测试经常通过一个定制的测试马具（test harness，取驾驭“测试”这辆马车的工具之意。——译者注）程序来完成，它负责执行所有的测试用例。在Java程序中，经常会在Java类中加入一个main方法，用它对该类进行单元测试。④

```
public static void main(String[] args) {
  String format = "yyyy-MM-dd HH:mm:ss.SSS";
  if (args.length > 0)
    format = args[0];
```

① netbsdsrc/regress/lib/libc/ieeefp/round/round.c:13-44

② jt4/catalina/src/test/org/apache/naming/resources/BaseDirContextTestCase.java

③ jt4/catalina/src/test/org/apache/naming/resources/WARDirContextTestCase.java

④ jt4/jasper/src/share/org/apache/jasper/util/FastDateFormat.java:128-139

图 6.16　使用 JUnit 测试框架

```
SimpleDateFormat sdf = new SimpleDateFormat(format);
FastDateFormat fdf = new FastDateFormat(sdf);
Date d = new Date();

d.setTime(1);
System.out.println(fdf.format(d) + "\t" + sdf.format(d));
d.setTime(20);
System.out.println(fdf.format(d) + "\t" + sdf.format(d));
d.setTime(500);
System.out.println(fdf.format(d) + "\t" + sdf.format(d));
[...]
}
```

在 C/C++程序中，有时会将模块的测试代码，放在由 DEBUG 预处理器宏控制的条件编译块中，来达到相同的效果。①

```
# ifdef DEBUG
main()
    {
```

① XFree86-3.3/xc/programs/Xserver/hw/hp/input/drivers/hp7lc2k.c:355-406

```
    int fd;

    makedata();             /* make the test data */
    fd = open ("data", O_RDWR);
    process_test_data(fd);
    close(fd);
    }
[...]
# endif /* DEBUG */
```

测试程序经常设计为交互式运行，如 vttest VT-100 终端仿真器测试程序。[①] 然而，更有效率的测试形式是将程序的结果与已知正确的输出进行测试。回归测试——通常基于早期经过人工检验的运行结果，经常在程序的实现被修复后执行，以确保程序中的各个部分没有被偶然地破坏。回归测试一般由一个执行测试用例的测试马具、每个测试用例的输入、期望的结果和顺序执行所有用例的框架组成。execve 完成 Unix 系统调用的回归测试就是一个典型的例子。下面这个微小的驱动程序可以用来执行不同类型的程序，并打印出结果。[②]

```
int
main(int argc, char *argv[])
{ [...]
    if (execve(argv[1], &argv[1], NULL) == -1) {
      printf("%s\n", strerror(errno));
      exit(1);
    }
}
```

针对执行的每种程序（测试用例），比如：[③]

```
#!/foo/bar/baz

echo foo
```

都有一个单独的文件包含期望的正确结果。[④]

```
No such file or directory
```

最后，用一条 Makefile 规则顺序运行所有的回归测试[⑤]。

```
regress: test-empty test-nonexist \
    test-nonexistshell test-devnullscript test-badinterplen \
    test-goodscript test-scriptarg test-scriptarg-nospace \
    test-goodaout test-truncaout
```

① netbsdsrc/sys/arch/bebox/isa/pcvt/Util/vttest

② netbsdsrc/regress/sys/kern/execve/doexec.c:38-52

③ netbsdsrc/regress/sys/kern/execve/tests/nonexistshell:1-3

④ netbsdsrc/regress/sys/kern/execve/good/nonexistshell:1-3

⑤ netbsdsrc/regress/sys/kern/execve/Makefile:15-21

```
test-empty: ${PROG} ${TD}/empty
    ${RP} ${TD}/empty | diff - ${OD}/empty
```

在阅读源代码时，要记住，测试用例可以部分地代替函数规格说明。另外，可以使用测试用例的输入数据对源代码序列进行预演。

练习 6.24　运行本书配套盘中的一个程序，匹配运行过程中生成的调试输出和实际生成它的源代码。

练习 6.25　在本书配套盘中找出两个不同的测试用例，说明它们的执行是如何自动化的？

练习 6.26　参考 execve 回归测试的测试用例，得出相应手册页的“Errors”节。比较您获得的结果与实际的手册页。①

练习 6.27　Perl 的 bug 报告文档②建议：

好的测试用例拥有下列属性中的绝大部分：尽可能少的行数；对外部命令、模块或库的依赖很少；可以流畅地在大多数平台上运行；并且自我解释。

在本书配套盘中找出一些测试用例，依据上面的标准对它们进行判断。

进　阶　读　物

Feldman[Fel79]介绍了 make 程序最初的设计与基本原理；[OT91，DE96]深入地论述了 make 及其变体。实现 Haskell 语言时对专用工具的使用在 Spinellis[Spi93]中做了介绍，有关管理错误消息的内容参见 Douglas[Dou90]。专用工具还常常用来实现领域专用的语言，参见[Spi99b，Ram97，FNP97，Spi01，KS97，SG97a，SG97b，Cre97]。最初两篇关于修订控制系统的文章是 Rockkind[Roc75]——介绍 SCCS，Tichy[Tic82]——介绍 RCS；比较容易得到的较新的参考书是 Bolinger 等著的[BBL95]，Hunt 和 Thomas[HT00，p271]列出了一系列的工具，还有相关的评注。两本书[C+01，BF01]对 CVS 做了介绍。Vaughan 等著的[VETL00]对 GNU autoconf 工具做了论述。Humphrey[Hum89，pp113-135，225-246]对配置过程和修订管理做了进一步的介绍。Kaner 等著的[KFN99]详细介绍了测试方法和测试管理，用 JUnit 对 Java 程序进行测试在 Beck 和 Gamma[BG98]中做了描述。

① doc/execve.pdf

② perl/utils/perlbug.PL:1015-1017

第 7 章　编码规范和约定

自然界的法则只不过是上帝的数学思维而已。

——Euclid

意义重大的编码工作，或大型、有组织体制之下的项目，比如 GNU 和 BSD，都会采纳一套编码规范、指导原则或约定。计算机语言和编程系统为程序员如何表达一个给定的算法提供了大量的余地。代码规范提供风格上的指导，目标是增强代码的可靠性、易读性和可维护性。在阅读代码时，规范和约定可以为您提供特定代码的额外指引，从而增加阅读的效率。本章中，我们分析在记述编码规范的文档中出现的一些典型的项。为了使我们的讨论更为具体，我们在 3 种常用的编码规范的语境下分析我们的示例：GNU 编码规范，被数百个 GNU 项目和 Linux 操作系统所采用；BSD(内核)风格指导，用于 BSD 操作系统家族，比如 FreeBSD、NetBSD 和 OpenBSD，同样还应用在如 apache 和 Perl 之类的项目中；Java 编码约定，应用在大部分 Java 代码。此外，我们将介绍匈牙利命名标记，这种方法常用在 Microsoft Windows 世界中，对标识符进行命名。

练习 7.1　找出您所在的组织采用的编码规范。以表格的形式，与另外两种常用的规范对比它们涵盖的范围。

7.1　文件的命名及组织

大多数规范都会说明文件应该如何命名，应该使用什么样的扩展名。例如，您也许没有意识到，C 头文件惯常的.h 后缀就是一种风格，没有任何语言或编译器规定必须要使用它。遵守文件名和后缀的约定能够优化对源代码的搜索。许多文件名和扩展名跨不同项目和开发平台通用；我们在表 7.1 和表 7.2 中将它们列了出来。

表 7.1　通用文件名

文件名	内容
README	项目概况
MANIFEST	项目文件的清单，带有简要的解释
INSTALL	安装指示
LICENSE 或 Copying	许可信息
TODO	希望将来进行扩充的清单
NEWS	介绍用户可视的更改的文档
ChangeLog 或 Changes	代码更改汇总
configure	平台配置脚本
Makefile	描述应该怎样进行编译链接，也称制作文件
Makefile.SH	用来生成上面文件的外壳脚本
config.h	平台相关的配置定义
config_h.SH	用来生成上面文件的外壳脚本
version.h 或 patchlevel.h	项目的版本号

表 7.2　常见的文件扩展名

文件的扩展名	内容
. digit . man	Unix 手册页的源文件
. encoding	特定编码的文本(例如. uft8)
. language-code	特定语言的文本(例如,德语为. de)
. a, . lib	目标文件库的集合
. asm, . s	汇编语言源码(Microsoft DOS/Windows, Unix)
. asp, . cgi, . jsp, . psp	Web 服务器可以执行的源文件
. awk	awk(语言)代码
. bas, . frm	Microsoft Visual Basic 源文件
. bat, . cmd, . com	MS-DOS/NT, OS/2 Rexx, VMS 命令
. bmp, . xbm, . png	Microsoft Windows, X Window 系统中的可移植位图文件
. c	C 源文件
. C, . cc, . cpp, . cxx	C++源文件
. cs	C#源文件
. class	Java 编译后的文件
. csh, . sh	用于 Unix csh, sh(Bourne)外壳的源文件
. def	Microsoft Windows 或 OS/2 可执行或共享库的定义
. dll, . so	共享的对象库(Microsoft Windows, Unix)
. dsp, . dsw, . vbp	Microsoft Developer Studio 项目和工作区文件
. dvi	troff 或 Tex 设备无关输出格式的文档
. el	Emacs Lisp 源文件
. eqn, . pic, . tbl	要由 eqn, pic, tbl 进行排版的公式、图片、表
. gz, . Z, . bz2	由 gzip, compress, bzip2 压缩的文件
. h	C 或 C++头文件
. hpp	C++头文件
. ico, . icon	Microsoft Windows, X Window 系统的图标
. idl	接口定义文件
. info	由 GNU Texinfo 生成的文档
. jar, . shar, . tar	Java, Unix Shell, 或磁带存档格式的文件集合
. java	Java 源代码
. jcl	IBM JCL 指令
. l	lex(词汇分析程序的生成器)的源文件
. m4	m4(宏处理程序)代码
. mak, . mk	制作文件(还经常没有任何扩展名)
. mif	FrameMaker 输出的文档
. mm, . me	使用 troff mm, me 宏的文档
. nr, . roff	纯 troff 格式的文档
. o, . obj	目标文件
. ok	拼写检查字典的本地附加
. pl, . pm, . pod	Perl、模块源文件, 文档
. ps	Postscript 源文件, 或 Postscript 格式的文档
. py	Python 源文件
. rb	Ruby 源文件
. rc, . res	Microsoft Windows 资源, 编译后的资源描述
. sed	sed(流编辑器)源文件
. t, . test	测试文件
. tcl	Tcl/Tk 源文件
. tex, . texi	TeX 或 LaTeX、GNU Texinfo 格式的文档
. y	yacc(解析程序生成器)源文件

此外，许多风格指南都会建议如何安排源文件中不同的程序项。例如，BSD 风格指南明确地指定了 include 语句的次序：内核包含文件，接下来是/usr 包含文件，接下来是用户的包含文件。类似地，Java 的代码约定规定，Java 类中的元素应该按照如下的次序：

(1)类变量

(2)实例变量

(3)构造函数

(4)方法

另外，Java 约定还指定，变量和方法应该以 private，protected 和 public 的次序来定义。了解了给定代码库所遵循的文件组织方式后，就能更有效率地浏览它的源代码。

练习 7.2 讨论按照严格规定的次序，定义和声明代码元素的优点和缺点。现代集成开发环境如何影响您的观点？

7.2 缩　　进

现代块结构语言编写的程序，都使用缩进来强调每个块的嵌套层次。风格指南经常规定用来缩排代码块的空格的数量和类型。例如，Java 代码约定规定使用 4 个空格，同时，允许使用 8 个字符宽度的制表符(tab)，而 BSD 风格指南规定使用 8 个字符宽度的制表符。是否允许使用制表符，以及它们的精确宽度十分重要。缩排代码块有两种方式：使用若干空格符或一个制表符。遗憾的是，许多编辑器将 tab 键重载为缩进给定数量的空格，另外一些编辑器也大都提供一个选项，让用户指定 tab 字符(ASCII 码 9)生成的空格数量。如果用来编排最初程序的制表符设置与用来阅读它的设置不匹配，则会影响代码的外观，最好的情况是混乱，最坏的情况是无法阅读、难以理解。问题往往出现在，用制表符来缩排某些开头代码(例如，注释和变量声明)之后的元素，或者混合使用制表符和空格缩排代码时(一些编辑器自动将 tab 字符替换成一定数量的空格)。从图 7.1 中的代码[①]可以得出，使用 tab 和空格格式化的 Java 代码，当将制表符设置(错误地)改为 4，而非 8 时，会是什么样子。

```
 for (int i = 0; i < tTable.size(); i++) {          ← 1 错误缩进
Table table = (Table) tTable.elementAt(i);
[...]
for (int j = 0; j < columns; j++) {
    Object o[] = t.getNewRow();
    [...]
     if (table.getColumnIsNullable(j)) {            ← 1 错误缩进
    nullable = DatabaseMetaData.columnNullable;
    } else {
    nullable = DatabaseMetaData.columnNoNulls;
    }
    [...]
    t.insert(o, null);
}
}
[...]
o[2] = new Integer(0);        // precision           ← 2 注释没有排齐
[...]
o[9] = new Boolean(false);      // unsigned          ← 2
```

图 7.1 对 Java 代码应用了错误的制表符设定

① hsqldb/src/org/hsqldb/DatabaseInformation.java:143-444

阅读代码时，首先要确保您的编辑器和打印机的 tab 设置，与代码遵循的风格规范一致。如果代码不能正确排列，则表明设置存在错误。如果找不到正在分析的项目的风格指南，就试用一些常见的设置。另外，源文件有时会在注释中包括指示编辑器应该如何安排它们的编排设置。一些编辑器，如 Emacs 和 vim，能够提取这些设置中描述的格式，自动配置它们的编辑会话。即使您的编辑器不支持这项功能，您也可以读出这些值（一般在文件的顶部或底部），手动地设置编辑器的参数。下面的注释块就是一个典型的例子：①

```
/*
 * Local Variables:
 * tab-width: 4
 * c-indent-level: 4
 * c-argdecl-indent: 4
 * c-continued-statement-offset: 4
 * c-continued-brace-offset: -4
 * c-label-offset: 4
 * c-brace-offset: 0
 * End:
 */
```

大多数风格指南精确地规定程序的各个构成部分应该如何缩进，以反映出程序的块结构。可以使用代码块的缩进，快速地掌握代码的总体结构。更重要的是，大多数风格指南还规定了跨越多行的语句应该采取什么样的缩进规则。了解这些规则，就可以快速地从控制块的语句中区分出延续行。例如，知道了 BSD 风格指南规定延续行的缩进为 4 个空格，控制块语句为 8 个之后，在下面的代码中，无需匹配对应的圆括号，就能够正确地得出序列 fp =fdp ... 属于 if 子句②。

```
if ((u_int)fd >= fdp -> fd_nfiles ||
    (fp = fdp -> fd_ofiles[fd]) == NULL)
        return (EBADF);
```

练习 7.3　请说说如何找出源代码文件的正确 tab 设定。

7.3　编　　排

所有的代码规范都会规定声明以及每个具体语句的编排方式。规格说明中包括空格和换行符（line break）的分布。对于花括号的放置，存在两个不同的学派。GNU，Linux，以及许多 Windows 程序倾向于将花括号放在单独的行中，经常还要缩排。③

```
if (ia -> ia_maddr != MADDRUNK)
{
    sc -> pPacketPagePhys = ia -> ia_maddr;
```

① netbsdsrc/usr.sbin/bootpd/hash.c:415-425
② netbsdsrc/sys/kern/kern_descrip.c:786-788
③ netbsdsrc/sys/arch/arm32/isa/if_cs_isa.c:521-524

```
}
```

BSD和Java程序经常在语句的同一行打开花括号，使用和语句相同的缩进级别关闭花括号。①

```
if ((cp = strchr(*argv, ':')) != NULL) {
    *cp++ = '\0';
    a_gid(cp);
}
```

只要一致地使用，两种变体都十分易读。BSD/Java变体的优点之一是可以少占用一些垂直空间，从而使代码块更容易放入单个页面中。存在问题的明显信号是代码的编排不一致。考虑下面这段代码。②

```
xpupil.y = Xy(w -> eyes.pupil[0].x, w -> eyes.pupil[0].y, &w -> eyes.t);
xnewpupil.x = Xx(newpupil[0].x, newpupil[0].y, &w -> eyes.t);
xnewpupil.y = Xy(newpupil[0].x, newpupil[0].y, &w -> eyes.t);
if (! XPointEqual (xpupil, xnewpupil)) {
```

请注意，= 运算符之后使用的空格数量不同，函数名后使用的空格也不一致。对于编排不一致的代码，应该立即给予足够的警惕。如果编写该段代码的程序员没有经过正确编排的训练，那么极有可能存在大量其他更严重的错误。对不一致性的其他解释也都指向问题点。该代码可能是集成不同代码库中的元件的结果。同样，如果集成者不协调代码的编排（在多数情况下，使用代码编排工具，如indent，就能够轻易地修复），那么他们就有可能也没有处理更为重要的集成问题。不一致性也可能是多个程序员在同一段代码上工作的结果。在这种情况下，编排上的冲突反映出团队的协调很差，或者许多程序员不尊重前辈工作。作为这类软件的使用者要当心。

风格指南还规定了注释的编排和内容。不管大学编程课程中是如何讲授的，注释内容并非专门说给人类的。

- 由/*-序列开始的注释告诉格式编排程序indent不要重新编排它们的内容。这类注释可能经过仔细地编排（例如，其中包含数据结构的ASCII图），因此，它们的内容如果被重新编排，极有可能会受到破坏。
- /** 序列开始的Java注释由javadoc来处理，自动生成源代码文档。在这类注释内，javadoc关键字要以@开头（见图6.15）。
- 包含单词FALLTHROUGH的注释向代码检验程序，如lint，表明程序员有意在switch case代码的结尾不添加break语句。③

```
case 10:        /* yy */
    lt -> tm_year = ATOI2(p);
    if (lt -> tm_year < 69)         /* hack for 2000 ;- ) */
```

① netbsdsrc/usr.sbin/chown/chown.c:158-161

② XFree86-3.3/contrib/programs/xeyes/Eyes.c:378-381

③ netbsdsrc/bin/date/date.c:162-168

```
        lt -> tm_year += 100;
    /* FALLTHROUGH */
case 8:                                 /* mm */
    lt -> tm_mon = ATOI2(p);
```

- 类似地，包含单词 NOTREACHED 的注释表明，程序的控制流程永远不会到达特定的点，因此，不要总是给出警告。例如，下面的摘录中，NOTREACHED 注释避免了函数在结尾没有返回一个值的警告。①

```
size_t
strcspn(const char *s1, const char *s2)
{
    register const char *p, *spanp;
    register char c, sc;

    /*
     * Stop as soon as we find any character from s2. Note
     * that there must be a NUL in s2; it suffices to stop
     * when we find that, too.
     */
    for (p = s1;;) {
        c = *p++;
        spanp = s2;
        do {
          if ((sc = *spanp++) == c)
            return (p - 1 - s1);
        } while (sc != 0);
    }
    /* NOTREACHED */
}
```

- 形如 $Identifier$ 或%Letter%的注释成分有可能是修订控制系统的标记。当修订控制系统处理这些文件时，会自动地用文件的属性替换这些成分。我们在 6.5 节中详细讨论了这些内容。

最后，编码规范还规定了一些特殊的注释单词，使得所有的实现人员都能够容易地搜索和确认。标识符 XXX 传统上用于标志虽然不正确、但(大多数时候)能够工作的代码。②

```
/* XXX is this correct? */
v_evaluate(vq, s, FALSE);
```

FIXME 标记错误的、需要修复的代码。③

① netbsdsrc/lib/libc/string/strcspn.c:53-74

② netbsdsrc/bin/ksh/var.c:372-373

③ netbsdsrc/lib/libcompat/regexp/regexp.c:483-485

```
switch (*regparse++) {
/* FIXME: these chars only have meaning at beg/end of pat? */
case '^':
```

TODO用来标记将来需要增强的地方。①

```
/*
 * TODO - sort output
 */
int,
aliascmd(argc, argv)
```

一些编辑器，如vim，能够智能地用特殊的颜色将这些注释标记出来，以引起您对它们的注意。分析代码时，对标记为XXX，FIXME和TODO的代码序列要格外注意：错误可能就潜伏在其中。

练习7.4 设计一份指南，规定什么情况下应该改变引入代码的格式编排，以顺应本地编码规范。注意代码的品质、易读性、维护、修订控制和将来的集成问题。

7.4 命名约定

大多数编码规范中，一个重要部分就是关于标识符的命名。标识符名称的指导原则既有简洁明了型（“选择不会造成误解的变量名”[KP78，p. 15]（仅仅一句话，够简洁。——译者注），也有详尽复杂型（后面您将看到匈牙利记法）。变量、类和函数的标识符大致有3种不同的构造方式。

(1)首字母大写

标识符中每个新单词的首字母都大写。两个具有代表性的例子是C Windows API函数SetErrorMode②和Java类RandomAccessFile③。多数Java和Windows编码约定都推荐首字母大写的方法。首字母是否大写要依附加的规则而定。

(2) 用下划线分隔单词

标识符中每个附加单词都通过一个下划线与前面的单词分隔开来（例如，exponent_is_negative④。这种方法在受Unix影响的代码中很流行，并且也是GNU编码规范的推荐方法。

(3) 只取首字母

这个方案中，取每个单词的首字母合并组成新的标识符。例如，splbio⑤的意思是“set processor (interrupt) level (for) buffered input output”（设置缓冲输入输出的处理器（中断）级别）。除了很少数广泛使用的标识符外，GNU编码规范推荐不使用这种命名方法；在

① netbsdsrc/bin/sh/alias.c:200-204
② apache/src/os/win32/util_win32.c:529
③ hsqldb/src/org/hsqldb/Cache.java:50
④ apache/src/ap/ap_snprintf.c:569
⑤ netbsdsrc/sys/kern/vfs_subr.c:410

BSD 的世界中，该方法在接口文档编制中的应用已经得到认可。这个方法还和词尾或元音移除方法结合使用，例如 rt_refcnt[①]；还有标准的 C 函数 strcmp。

所有这 3 种编码规范在如何命名常量上都是一致的：常量使用大写字母命名，单词用下划线分隔。请记住，这种大写命名约定可能和宏互相冲突，因为宏也遵循相同的命令方案。现代 C 和 C++程序中，整型的常量应该用 enum 声明来定义。当 enum 定义的常量与同名的宏发生冲突时，生成的编译器错误在大多数情况下都会含糊不清，因为这种情况下，enum 值已经被宏的替换序列所替换。

另外，在遵循 Java 编码规范的程序中，包名(package name)总是从一个顶级的域名开始(例如，org.，com. sun.)，类名和接口名由大写字母开始，方法和变量名由小写字母开始。这个约定使得您能够快速区分出类的引用(静态)和实例变量和方法。例如，Library. register (hAlias)[②]是调用 Library 类的类方法 register；而 name. equals(". ")[③]则是调用 name 对象的实例方法 equals。

Windows 世界的程序中，经常可以看到难以发音的标识符名，看起来好像是应用某种复杂的方案构成的。实际上，我们可以对标识符名进行解码，从中获得有用的信息。例如，变量名 cwSz 可能用来表示以 0 结尾的字符串中单词的个数。这种编码方案称为匈牙利命名记法(Hungarian naming notation)，它是以其开发者，Charles Simonyi.，的祖国命名的。开发这种记法的前提是合理的：标识符名应该易于被程序员记住，并对阅读代码的人有提示价值，类似标识符的名称应该一致，并能够快速判定。遗憾的是，这种记法时常被误用，将与具体实现相关的信息嵌入到标识符名中，而这些信息本来应该对标识符是不透明的。

所有标识符的基本元素就是它们的标记(tag)——表明元素类型的简短标识。程序员应该为程序中使用的每种数据结构或数据类型发明新的标记名称。标记可以加上一个或多个类型前缀，构造表示合成类型的标识符(例如，指向 X 的指针，或 X 的数组)；或加上基本名称(base name)后缀，用以区分同一类型的不同元素；或使用限定符(qualifier)，表示通用规范的分化。在表 7.3 中，我们列出了一些常用基本类型的标记、类型构造和名称限定符。例如，如果 Point 是一个标记，用来表示屏幕上的点，rgPoint 使用一个前缀来表示这类点的数组，pPointFirst 用一个前缀和一个限定符来表示指向数组中第一个元素的指针，PointTop 和 PointBottom 用基本名称来区分可能表示图形的两个点。

匈牙利记法并非仅仅用于 C 和 C++标识符。用 Visual Basic 编写的程序，不管是独立的版本，还是 Visual Basic for Application，都经常用匈牙利记法来命名标识符。Visual Basic 提供丰富的内建类型和可编程 GUI 控件。大多数常见类型和控件的标记(tag)都已经标准化为表 7.4 中的名称。在阅读 GUI 应用程序中处理用户界面的代码时，窗体中所有用户界面控件的操作经常放在同一个代码模块中。在中等复杂的应用程序中，会有数十个全局控件在该代码中都可见。用户界面控件名称之前的匈牙利记法的前缀类型标记可以帮助我们确定它的作用。

---

① netbsdsrc/sys/netinet/if_arp. c:608

② hsqldb/src/org/hsqldb/Database. java:82

③ hsqldb/src/org/hsqldb/Database. java:92

表 7.3 匈牙利记法的基本类型、类型构造和名称限定符

构成	意义
基本类型	
f 或 b	标志(布尔)基本类型
dw	双字(32 位),任意内容
w	字,任意内容(一般无符号)
b	字节,任意内容
ch	字符
sz	指针,指向 null 结尾字符串的第一个字符
h	对象(在堆上分配)的句柄
fn	函数
i	整型
l	长整型(32 位)
n	短整型(16 位)
类型构造	
pX	指向 X 的指针
dX	两个 X 的实例之间的差
cX	类型 X 实例的计数
mpXY	由 X 索引的 Y 类型的数组(map)
rgX	X 类型的数组(range)
iX	rgX 数组的索引
cbX	X 实例的大小,以字节为单位
cwX	X 实例的大小,以字为单位
名称限定符	
XFirst	一组有序 X 类型的值中的第一个元素
XLast	一组有序 X 类型的值(包括 XLast)中的最后一个元素。
XNext	一组有序 X 类型的值中的下一个元素
XPrev	一组有序 X 类型的值中的前一个元素
XLim	一组有序 X 类型的值(不包括 XLim)的上限
XMac	当前的上限,对应数组常量或分配限制
XNil	类型为 X 的特定 Nil 值,比如 NULL,0 或-1
XT	类型为 X 的临时值

表 7.4　Visual Basic 和 Microsoft Access 通用类型和控件的匈牙利记法

标记	类型
基本类型	
cur	货币
time	时间
date	日期
dt	日期和时间的组合
qry	数据查询
tbl	数据表
通用控件	
frm	窗体
mnu	窗体菜单
cmd	命令按钮
chk	选择框
opt	单选框
lbl	文本标签
txt	文本编辑框
pb	图片框
pic	图片
lst	列表框
cbo	组合框
tmr	定时器

练习 7.5　有人认为，匈牙利命名约定重复了由编译器执行的类型检查工作。请针对这一观点进行讨论。

7.5　编 程 实 践

许多编码指导原则都对如何编写可移植的软件做了明确的规定。但是，要知道，不同的编程规范对可移植构造的构成有不同的主张。对那些主要关注的并非是最大可能的可移植性的工作来说，尤为如此。例如，GNU 和 Microsoft 编码指导原则都将可移植性的焦点放在处理器构架的不同上，并且认为特定的软件只会在 GNU，Linux 或 Microsoft 操作系统上运行。这么狭窄的可移植性定义，无疑是来源于实用的考虑，或许还有附加的暗示：GNU 编码规范明确地表明，GNU 项目不支持 16 位构架。在审查代码的可移植性，或以某种给定的编码规范作为指南时，要注意了解规范对可移植性需求的界定与限制。

编码指导原则还经常规定应用程序应该如何编码，以保证正确或一致的用户界面。特

定的相关例子就是对命令行选项的解析。BSD 和 GNU 指导原则规定，提供命令行接口的程序应该相应地使用 getopt[①] 和 getopt_long 来解析程序的选项。请考虑下面这段解析命令行参数的代码。[②]

```
while (argc > 1 && argv[1][0] == '-') {
    switch(argv[1][1]) {
    [...]
    case 's':
      sflag = 1;
      break;
    default:
      usage();
    }
    argc-- ; argv++ ;
}
nfiles = argc - 1;
```

请将上面这段代码与使用库函数 getopt 的标准表达进行对比。[③]

```
while ((ch = getopt(argc, argv, "mo:ps:tx")) != - 1)
    switch(ch) {
    [...]
    case 'x':
      if (! domd5)
        requiremd5("- x");
      MDTestSuite();
      nomd5stdin = 1;
      break;
    case '?':
    default:
      usage();
    }
argc -= optind;
argv += optind;
```

除了更易于理解以外，getopt 版本与所有使用它的 C 程序看起来形式完全相同，在您的头脑中形成一个精确的应用模式，并且应该警惕与这种应用模式的任何背离。我们曾看到过数十种不同的具有创造性的、特殊的参数解析代码序列；每种代码都需要我们重新投入努力来理解与检验。标准化的用户界面例程带来的优点在 GUI 应用程序中更为显著。各种 Microsoft 指导原则都推荐程序使用 Windows 所支持的界面控件，比如文件、颜色和字体对话框，而不是自己来编写。所有情况下，如果 GUI 功能都使用相应的编程结构来实现，则通过代码审查可以轻易地验证给定用户界面的规格说明是否被正确地采用。

---

① 关于 getopt 的手册页，参见图 2.4。

② netbsdsrc/usr.bin/checknr/checknr.c:209-259

③ netbsdsrc/usr.bin/cksum/cksum.c:102-155

练习 7.6　创建一个详细清单，列出有助于提高代码易读性的编程准则。尽可能引用现有的例子。

7.6　过程规范

软件系统并非只是代码。因此，许多编码指导原则都延伸到开发过程的其他领域，包括文档、生成和发布过程的组织。

至少，许多指导原则都会规定标准的文档，以及编写它们的格式。由于最终用户文档经常与应用程序或发布过程紧密地绑定在一起，故而常常组织的最好。Microsoft Windows 应用程序一般都会包括一个帮助文件，GNU 项目提供 Texinfo 手册，传统的 Unix 应用程序提供标准化的手册页。我们在 8.2 节中讨论如何利用文档协助阅读代码。

如 6.3 节所述，没有普遍适用的方式可以规定和执行编译过程，这就使得理解每个不同编译机制的特殊性变得比较困难。但是，许多指导原则都为如何组织编译过程建立了精确的规则。这些规则经常是基于特定的工具或标准化的宏；熟悉了它们之后，您就能够很快地理解，遵循给定指导原则的任何项目的编译过程。为了认识标准化的编译过程，请对比 18 行的制作文件①——用来编译 NetBSD 网络时间协议(NTP)库的 64 个文件，和制作文件模板②——40 行手工编写，37 行自动生成，用来编译 apache 支持库的 12 个文件。

有时也会对发布过程做出精确的规定，一般是为了满足应用程序安装过程的需求。标准化的软件发行格式，比如 Microsoft Windows installer 使用的格式，或 Red Hat Linux RPM 格式，对组成发行包的文件类型、版本控制、安装目录和由用户控制的选项都有严格的规则。当检查系统的发布过程时，常常可以将相应发行格式的需求作为基线。

练习 7.7　确定适用于您的组织的过程规范。解释如果要验证系统是否与这些规范一致，需要检查哪些软件元素(文件、目录)。

进　阶　读　物

重要的编码约定包括 Indian Hill 风格指南[CEK+]、GNU 编码规范[S+01]、FreeBSD 内核源文件风格指南[Fre95]和 Java 代码约定[Sun99a]。您还可以在 Kernighan 和 Plauger [KP78]的经典图书、Kernighan 和 Pike[KP99]最近的图书、以及一系列 Spencer 的文章[Spe88, SC92, Spe93]中找到编码的建议。匈牙利变量编码记法在 Simonyi 的博士论文[Sim76]中首次出现，更普遍且易于理解的版本是他最近的著作[Sim99]。

① netbsdsrc/lib/libntp/Makefile

② apache/src/ap/Makefile.tmpl

第8章 文 档

文档就如同性爱：当它很完美时，那真是太美妙了；当它不合谐时，也比没有要好一些。

——Dick Brandon

任何重大的软件开发项目都会伴随各种正式或兴之所至的文档。在接下来的内容中，我们介绍在分析项目过程中会常常遇到的一些典型文档，并提供具体的例子，说明这些文档怎样帮助您理解软件代码，同时分析一些文档错误的种类，并概括常见的开放源码文档格式。阅读代码时，应该尽可能地利用任何能够得到的文档。我们可以将那句提倡使用库的格言变换成：阅读一小时代码所得到的信息只不过相当于阅读一分钟文档。

8.1 文档的类型

应用传统工程方法的项目，在开发过程中，会生成大量不同的文档。当这些文档得到正确维护时，它们确实能够帮助您理解系统的基本原理、设计和实现。尽管不同的项目制造林林总总不同类型的文档，下面的段落只概括了您可能遇到的一些具有代表性的文档。

系统规格说明文档（system specification document）详细描述系统的目标、系统的功能需求、管理和技术上的限制、以及成本和日程等要素。通过系统的规格说明文档能够了解所阅读代码的实际运行环境。同样一段绘图代码，用在屏幕保护程序中，或作为核反应控制系统的一部分，您对它的解读态度会完全不同。

软件需求规格说明（software requirements specification）提供对用户需求和系统总体构架的高层描述，并且详细记述系统的功能和非功能性需求，比如数据处理、外部接口、数据库的逻辑模式以及设计上的各种约束。由于软硬件环境和用户需求的不断改变，文档中还可能描述预期的系统演化。软件需求规格说明是阅读和评估代码的基准。

设计规格说明（design specification）一般描述系统的构架、数据和代码结构，还有不同模块之间的接口。面向对象的设计会勾画出系统的基本类型以及公开方法。细化的设计规格一般还包括每个模块（或类）的具体信息，比如它执行的处理任务、提供的接口，以及与其他模块或类之间的关系。另外，设计规格说明还会描述系统采用的数据结构，适用的数据库模式等。我们可以将系统的设计规格说明作为认知代码结构的路线图、阅读具体代码的指引。

系统的测试规格说明（test specification）包括测试计划、具体的测试过程、以及实际的测试结果。每个测试过程都会详细说明它所针对的模块以及测试用例使用的数据（从中可以得知特定的输入由哪个模块处理）。利用测试规格说明文档提供的数据，可以预演正在阅读的代码。

最后是由许多不同文档组成的用户文档（user documentation），这些文档包括功能描述、安装说明、介绍性的导引、参考手册和管理员手册。用户文档的最大优点是，它常常是我

们惟一有可能获得的文档。在接触一个未知系统时，功能性的描述和用户指南可以提供重要的背景信息，从而更好地理解阅读的代码所处的上下文。从用户参考手册中，我们可以快速地获取，应用程序在外观与逻辑上的背景知识，从管理员手册中可以得知代码的接口、文件格式和错误消息的详细信息。

练习 8.1　从本书配套盘中选取 3 个大型项目，归类可以得到的文档。

练习 8.2　评论我们描述的那些文档类型在开放源码开发中的适用性。

8.2　阅读文档

如果您怀疑有用的文档实际上根本不存在，此处就是几个具体的例子，说明怎样使文档为您服务。

利用文档可以快捷地获取系统的概况，了解提供特定特性的代码。以 Berkeley Unix 快速文件系统(fast file system)的实现为例。在系统的管理者手册中，这个文件系统的帮助信息由 14 个编排好的页面组成，详细描述系统的组织方式、参数化、布局策略、性能以及功能上的增强。阅读这些文字[①②]要比分析构成系统的 4 586 行源代码容易得多。[③] 即使是在分析较短的代码片断时，通过阅读文档了解了它们的用途之后，它们的功能也会变得更为清晰。请分别在阅读 cat 命令-s 标志(用来设置 sflag 标识符)的文档[④⑤]之前和之后，试着理解下面的代码[⑥]：

```
line = gobble = 0;
for (prev = '\n'; (ch = getc(fp)) != EOF; prev = ch) {
    if (prev == '\n') {
      if (ch == '\n') {
         if (sflag) {
           if (! gobble && putchar(ch) == EOF)
               break;
          gobble = 1;
          continue;
        }
        [...]
      }
    }
    gobble = 0;
```

① netbsdsrc/share/doc/smm/05.fastfs
② doc/ffs.pdf
③ netbsdsrc/sys/ufs/ffs
④ netbsdsrc/bin/cat/cat.1
⑤ doc/cat.pdf
⑥ netbsdsrc/bin/cat/cat.c:159-207

```
    [...]
}
```

-s Squeeze multiple adjacent empty lines, causing the output to be single spaced.
(压缩相邻的多个空行,将输出用单个空格隔开。)

文档给出软件系统的规格说明,我们可以依照它对代码进行审查。功能性规格说明可以作为阅读代码的起点。代码大多会支持某种特定的产品或接口规范,所以,即使找不到指定系统的功能性规格说明,也可以使用对应的规范作为向导。以审查 apache Web 服务器对 HTTP 协议的顺从性为例。在 apache 源代码中,您会发现下面的代码序列。①

```
switch (*method) {
    case 'H':
      if (strcmp(method, "HEAD") == 0)
        return M_GET; /*  see header_only in request_rec */
      break;
    case 'G':
      if (strcmp(method, "GET") == 0)
        return M_GET;
      break;
    case 'P':
      if (strcmp(method, "POST") == 0)
        return M_POST;
      if (strcmp(method, "PUT") == 0)
        return M_PUT;
      if (strcmp(method, "PATCH") == 0)
        return M_PATCH;
```

对照 HTTP 协议的规格说明文档,RFC-2068②,对代码进行检查,可以容易地验证所实现命令的完全性,以及是否存在扩展。

The Method token indicates the method to be performed on the
resource identified by the Request-URI. The method is
case-sensitive.

```
Method      = "OPTIONS"          ; Section 9.2
            | "GET"              ; Section 9.3
            | "HEAD"             ; Section 9.4
            | "POST"             ; Section 9.5
            | "PUT"              ; Section 9.6
            | "DELETE"           ; Section 9.7
            | "TRACE"            ; Section 9.8
            | extension-method
```

文档经常能够反映和揭示出系统的底层结构。邮件传送代理程序 sendmail 的系统管理

① apache/src/main/http_protocol.c:749-764

② doc/rfc2068.txt:1913-1923

者手册[①②]就是一个具有代表性的例子。先看一下该系统的源代码文件[③]。

```
arpadate.c, clock.c, collect.c, conf.c, convtime.c, daemon.c, deliver.c, domain.c,
envelope.c, err.c, headers.c, macro.c, main.c, map.c, mci.c, mime.c, parseaddr.c,
queue.c, readcf.c, recipient.c, safefile.c, savemail.c, srvrsmtp.c, stab.c, stats.
c, sysexits.c, trace.c, udb.c, usersmtp.c, util.c, version.c
```

从表8.1中,我们可以看出,仅仅检查文档中特定的标题,就能够容易地将它们匹配起来。

表8.1 与sendmail文档标题相对应的源文件

文档标题		源文件
2.5.	Configuration file(配置文件)	readcf.c
3.3.1.	Aliasing(别名)	alias.c
3.4.	Message collection(消息收集)	collect.c
3.5.	Message delivery(消息投递)	deliver.c
3.5.	Queued messages(消息排队)	queue.c
3.7.	Configuration(配置)	conf.c
3.7.1.	Macros(宏)	macro.c
3.7.2.	Header declarations(报头声明)	headers.c, envelope.c
3.7.4.	Address rewriting rules(地址重写规则)	parseaddr.c

文档有助于理解复杂的算法和数据结构。复杂深奥的算法一般难以理解和掌握;当改写成高效的代码后,它们可能变得更不可理解。代码的文档常常会提供所使用算法的细节,或者,更为常见,代码中的某行注释为读者指出适当的出版物[④]。

```
*  This algorithm is from Knuth vol. 2 (2nd ed),
*  section 4.3.3, p. 278.
```

算法的文字描述能够使不透明(晦涩,难以理解)的代码变得可以理解。以理解下面这段代码[⑤]为例。

```
for (arcp = memp -> parents ; arcp ; arcp = arcp -> arc_parentlist) {
    [...]
    if ( headp -> npropcall ) {
       headp -> propfraction += parentp -> propfraction
          * ( ( (double) arcp -> arc_count )
            / ( (double) headp -> npropcall ) );
```

现在,请试着将这段代码与下面的算法描述进行对照[⑥⑦]。

① netbsdsrc/usr.sbin/sendmail/doc/intro
② doc/sendmail.pdf
③ netbsdsrc/usr.sbin/sendmail/src
④ netbsdsrc/lib/libc/quad/muldi3.c:86
⑤ netbsdsrc/usr.bin/gprof/arcs.c:930-950
⑥ netbsdsrc/usr.bin/gprof/PSD.doc/postp.me:71-90
⑦ doc/gprof.pdf: p.4

设 C_e 是对例程 e 的调用次数，C_r^e 是调用者 r 对被调用者 e 的调用次数。由于我们假定每次调用例程所花费的时间，是所有对该例程的调用的平均时间，所以，调用者要负责的时间是被调用者执行时间的 C_r^e/C_e。设 S^r 是例程 e 自身的执行时间(selftime)。例程自身的执行时间可以由在程序剖析执行过程中收集的计时信息来决定。那么，我们感兴趣的、例程 r 的总共执行时间 T^r，可以由下面的递归方程给出：

$$T_r = S_r + \sum_{\text{r call e}} T_e \times \frac{C_r^e}{C_e}$$

r calls e 是一个关系，表示例程 r 对例程 e 的所有调用。

文档常常能够阐明源代码中标识符的含义。考虑下面的宏定义，以及与之关联的简明注释。①

```
# define TCPS_ESTABLISHED 4 /*  established */
```

该宏定义是 TCP/IP 网络代码的一部分。在对应的文档(RFC-2068)中搜索单词 ESTABLISHED，可以得到下面的描述②。

ESTABLISHED—represents an open connection, data received can be delivered to the user. the normal state for the data transfer phase of the connection.

(表示一个打开的连接，接收到的数据可以交付给用户。这是连接处于数据传输阶段时的正常状态。)

更有帮助的是，同一份 RFC 文档中还有一份用 ASCII 字符精心绘制的图，详细说明 TCP 协议如何建模为有限状态机(finite state machine)，以及不同状态间的精确变迁(图 8.1)。在没有相应的状态图在手边的情况下，对 TCP 的源代码进行分析，绝非明智之举。

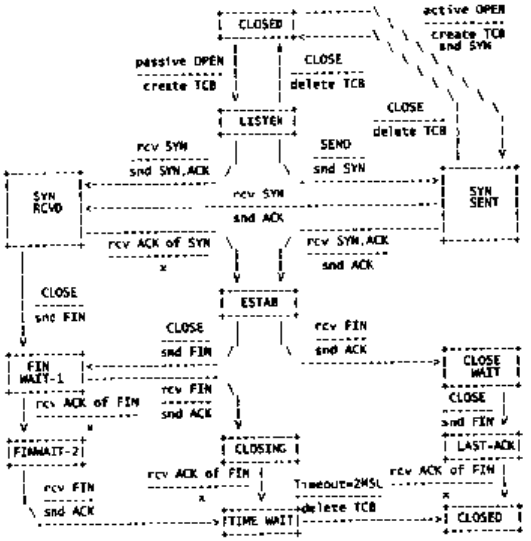


图 8.1 TCP 连接图示

① netbsdsrc/sys/netinet/tcp_fsm.h:50

② doc/rfc793.txt: p.21

文档能够提供非功能性需求背后的理论基础。下面的这段代码取自域名系统(domain name system,DNS)服务器。①

```
if (newdp -> d_cred > dp -> d_cred) {
  /* better credibility.
   * remove the old datum.
   */
  goto delete;
}
```

针对数据的"可信度"做出的类似选择,在整个源代码文件中多次出现。DNS的功能性规格说明并没有定义数据可靠性的概念,也未规定对来自不同来源的数据采取特定的行为。然而,在应用程序的doc目录中②,一系列的文档概括了大量与DNS有关的安全问题,其中有一个专门讨论如何为每个缓存条目标记一个"可信度"级别,基于它的值更新缓存的数据。③④

5.1 缓存标记

BIND现在为每个缓存的RR维护一个"可信度"级别,表示数据是来自一个区域、一个授权的应答者、一个权威段或附加数据段。当更为可信的RRset到来时,老的条目就被完全擦掉。老一些的BIND盲目地从所有的来源聚合数据,根本无视某些来源比其他更好这一原理。

这样,文档对非功能性需求背后的理论基础——安全,做了详尽的描述,对理解它在源代码中的实现很有帮助。

此外,在系统的文档中,您经常能够找到设计者当时的观点;系统需求的目标、目的和意图,构架,以及实现;还有当时否决的其他方案,以及否决它们的理由。一个具有代表性的例子就是,Pike和Thompson如何推断在Plan 9操作系统[PT93]中采用UTF Unicode表达方式,摒弃16位的替代方案。

Unicode定义了一个充分的字符集,但它的表达方式不尽合理。Unicode标准规定,所有的字符都用16个二进制位来表示,并且以16位为单位进行通信……。采用Unicode,我们将不得不把所有输入和输出到Plan 9的文本进行ASCII码和Unicode之间的转换,这是不可能做到的。对于单纯的程序,在它所有的输入和输出命令中,尚有可能将字符定义为16位的量,但是,在联网的系统中,面对各种各样机器上运行的由不同的制造商开发的数百个应用程序,这是不可能的。

……

UTF编码有几项好的属性。到目前为止,最重要的是,位于ASCII码区间0-127内的字节在UTF中就用它本身来表示。从而,UTF对ASCII码后向兼容。

文档还会说明内部编程接口。大型的系统一般会划分成多个小一些的子系统,这些子

① netbsdsrc/usr.sbin/named/named/db_update.c:447-452

② netbsdsrc/usr.sbin/named/doc

③ netbsdsrc/usr.sbin/named/doc/misc/vixie-security.ps

④ doc/vixie-security.pdf:p.5

系统通过精确定义的接口进行互操作。另外，重要的数据集合经常组织为抽象数据类型，或类似的接口定义完善的类。下面是一些常见的典型 API 文档。

- Hypersonic SQL 数据库引擎 hsqldb 的开发者文档。[①] 这份文档是用 Together ControlCenter 系统从带注释的 Java 源代码中生成的，可以用 HTML 浏览器来查看它。它在框架视图(frame view)中提供一个包括所有类的 UML 表示，一个包层次的树形结构，以及每个类的字段、构造函数和方法的汇总和详细视图。
- perlguts 手册页中，对 Perl 内部函数和数据表达格式的描述。[②③] 图 8.2 是该文档的首页。要注意"Description"节中的免责声明：由于文档很少像实际的程序代码那样进行测试，并受人关注，所以它常常可能存在错误、不完整或过时。
- FreeBSD 中用来管理联网代码内存的函数和宏的文档。[④⑤] 实际上，FreeBSD 和 NetBSD 手册的整个第 9 节中，包含超过 100 个与系统内核的内部操作相关的条目，它们描述了由系统和设备驱动程序开发人员使用的函数接口和变量。

文档也提供测试用例，以及实际应用的例子。源代码或系统的功能性文档，常常不会为我们提供如何实际使用系统的信息。tcpdump 程序支持一种复杂的语法，可以精确地指定希望分析的网络包。然而，如果没有实际应用的例子，就难以理解源代码正试图完成什么；测试用例为我们提供可以对代码进行预演(dry-run)的材料。令人高兴的是，程序的手册页——和大多数 Unix 手册页相同——提供了 10 种不同的典型应用，如下：[⑥⑦]

To print all ftp traffic through internet gateway snup: (note that the expression is quoted to prevent the shell from (mis-)interpreting the parentheses):

打印出通过 internet 网关 snup 的所有 ftp 流量(注意，该表达式被引起来，以防止外壳错误地对括号进行解释)：

```
tcpdump 'gateway snup and (port ftp or ftp-data)'
```

To print the start and end packets (the SYN and FIN packets) of each TCP conversation that involves a non-local host.

打印涉及非本地主机的每个 TCP 会话中的开始与结束包(SYN 和 FIN 包)：

```
tcpdump 'tcp[13] & 3 != 0 and not src and dst net localnet'
```

文档常常还会包括已知的实现问题或 bug。有时，您可能为了解源代码的特定部分如何正确地处理一个给定的输入，在该代码上花费数个小时的时间。通过阅读文档，您或许会发现系统根本不处理您正在分析的特殊情况，这是一个文档编制上的 bug。Unix 手册页，常

① hsqldb/dev-docs/hsqldb/index.html
② perl/pod/perlguts.pod
③ doc/perlguts.pdf
④ netbsdsrc/share/man/man9/mbuf.9
⑤ doc/mbuf.pdf
⑥ netbsdsrc/usr.sbin/tcpdump/tcpdump.8
⑦ doc/tcpdump.pdf：p.6

PERLGUTS(1)　　User Contributed Perl Documentation　　PERLGUTS(1)

NAME

perlguts - Perl's Internal Functions

DESCRIPTION

This document attempts to describe some of the internal functions of the Perl executable. It is far from complete and probably contains many errors. Please refer any questions or comments to the author below.

Variables

Datatypes

Perl has three typedefs that handle Perl's three main data types.

```
SV  Scalar Value
AV  Array Value
HV  Hash Value
```

Each typedef has specific routines that manipulate the various data types.

What is an "IV"?

Perl uses a special typedef IV which is a simple integer type that is guaranteed to be large enough to hold a pointer (as well as an integer).

Perl also uses two special typedefs, I32 and I16, which will always be at least 32-bits and 16-bits long, respectively.

Working with SVs

An SV can be created and loaded with one command. There are four types of values that can be loaded: an integer value (IV), a double (NV), a string, (PV), and another scalar (SV).

The six routines are:

```
SV*  newSViv(IV);
SV*  newSVnv(double);
SV*  newSVpv(char*, int);
SV*  newSVpvn(char*, int);
SV*  newSVpvf(const char*, ...);
SV*  newSVsv(SV*);
```

To change the value of an *already-existing* SV, there are seven routines:

```
void  sv_setiv(SV*, IV);
void  sv_setuv(SV*, UV);
void  sv_setnv(SV*, double);
void  sv_setpv(SV*, const char*);
void  sv_setpvn(SV*, const char*, int)
void  sv_setpvf(SV*, const char*, ...);
void  sv_setpvfn(SV*, const char*, STRLEN, va_list *, SV **, I32, bool);
void  sv_setsv(SV*, SV*);
```

Notice that you can choose to specify the length of the string to be assigned by using `sv_setpvn`, `newSVpvn`, or `newSVpv`, or you may allow Perl to calculate the length by using `sv_setpv` or by specifying 0 as the second argument to `newSVpv`. Be warned, though, that Perl will determine the string's length by using `strlen`, which depends on the string terminating with a NUL character.

The arguments of `sv_setpvf` are processed like `sprintf`, and the formatted output becomes the value.

`sv_setpvfn` is an analogue of `vsprintf`, but it allows you to specify either a pointer to a variable argument list or the address and length of an array of SVs. The last argument points to a boolean; on return, if that boolean is true, then locale-specific information has been used to format the string, and the string's contents are therefore untrustworthy (see the *perlsec* manpage). This pointer may be NULL if that

2/May/1999　　perl 5.005, patch 03　　1

图 8.2　perlguts 手册的首页

常包含标题为"Bugs"的节，专门用来记录这类 bug，这种直率十分引人注目，并值得表扬。实际上，在此节中还会说明系统的某个具体实现在设计或接口上的局限：①②

① netbsdsrc/usr. bin/at/at. 1

② doc/at. pdf

At and *batch* as presently implemented are not suitable when users are competing for resources. If this is the case for your site, you might want to consider another batch system, such as nqs.

(当前实现的 at 和 batch 程序并不适合于用户竞争资源的情况。如果您就属于这种情况,请考虑使用另外的批处理系统,比如 nqs。)

对系统应用的告诫:①②

Because of the shell language mechanism used to perform output redirection, the command "cat file1 file2 >file1" will cause the original data in file1 to be destroyed! This is performed by the shell before *cat* is run.

(根据外壳语言中执行输出重定向所采用的机制,命令"cat file1 file2 >file1"将会破坏 file1 中的原始数据! 这是由外壳在 cat 命令运行之前完成的。)

幽默:③④

There is no conversion specification for the phase of the moon.

(月亮的阴晴圆缺不存在转换规格说明。)

同样还有实际存在的 bug:⑤⑥

Recognition of functions, subroutines and procedures for FORTRAN and Pascal is done in a very simpleminded way. No attempt is made to deal with block structure; if you have two Pascal procedures in different blocks witb the same name you lose.

(FORTRAN 和 Pascal 语言在识别函数、子例程和过程时采取的方式十分简单。这两种语言都不会试图去处理控制块结构;如果在不同的控制块中有两个同名的 Pascal 过程,您就会失败。)(也就是说,要注意在 FORTRAN 和 Pascal 中不要定义同名的函数、子例程和过程,即使是在不同的文件或控制块中。它们没有 C/C++或 Java 中那么复杂的命名空间、作用域的概念。——译者注)

开发或运行环境中存在的缺点和明确的 bug,可能是问题产生的根源。它们有时记录在提供商的客户支持系统中,或是在伴随产品修补程序的修复清单中,这些地方都不是程序出现异常行为时最常搜索的地方。幸运的是,环境中已知的缺点一般都会记录在源代码中;当程序员谴责和攻击相关的提供商,发泄由于某个问题受到的挫折时,代码是最显然的地方。⑦

```
// the following function is not inline, to avoid build (template
```

① netbsdsrc/bin/cat/cat. 1
② doc/cat. pdf
③ netbsdsrc/lib/libc/time/strftime. 3
④ doc/strftime/pdf
⑤ netbsdsrc/usr. bin/ctags/ctags. 1
⑥ doc/ctags. pdf
⑦ ace/TAO/tao/Sequence_T. cpp:130-131

```
// instantiation) problems with Sun C++4.2 patch 104631-07/SunOS 5.6
```

(下面的函数之所以没有定义为 inline，是为了避免 Sun C++4.2 patch 104631-07/SunOS 5.6 的编译(模板实例化)问题。)

文档的变更能够标出那些故障点。许多系统的源代码中都包括某种形式的维护文档。它们至少会记录每次发布中所做的更改，不是以我们在 6.5 节中描述的那样以修订控制系统日志条目的形式，就是记录在纯文本文件中，一般命名为 ChangeLog(对更改的注释按发生的时间进行排列)。分析变更日志常常会揭示出那些重复，有时互相冲突，做出更改的区域，或类似的修复应用到源代码的不同部分。有时，第一种类型的更改表示存在根本性的设计缺陷，从而使得维护人员需要用一系列的修补程序来修复。Linux smbfs(Windows 网络文件系统)代码的 ChangeLog 中的下列条目就是一个典型的例子。

```
2001-09-17 Urban [...]
  * proc.c: Go back to the interruptible sleep as reconnects
    seem to handle it now.
[...]
2001-07-09 Jochen [...]
  * proc.c, ioctl.c: Allow smbmount to signal failure to reconnect
    with a NULL argument to SMB_IOC_NEWCONN (speeds up error
    detection).
[...]
2001-04-21 Urban [...]
  * dir.c, proc.c: replace tests cn conn_pid with tests on state
    to fix smbmount reconnect on smb_retry timeout and up the
    timeout to 30s.
[...]
2000-08-14 Urban [...]
  * proc.c: don't do interruptable_sleep in smb_retry to avoid
    signal problem/race
[...]
1999-11-16 Andrew [...]
  * proc.c: don't sleep every time with win95 on a FINDNEXT
```

很明显，proc.c 文件中的代码对细微的计时问题和信号竞争条件十分敏感。(公平地讲，该底层协议已经很长时间没有编制文档，并且它极为复杂，同时大量不同实现的运作都互不相同，甚至互不兼容。)

相比而言，相似的修复应用到源代码的不同部分，常常表示一种易犯的错误或疏忽，它们同样可能会在其他地方存在。vi 编辑器源码中的下列条目就是一个这种类型的更改。

```
1.65 -> 1.66 (05/18/96)
  + Send the appropriate TI/TE sequence in the curses screen
  whenever entering ex/vi mode. This means that :shell now
  shows the correct screen when using xterm alternate screens.
[...]
1.63 -> 1.64 (05/08/96)
```

```
  + Fix bug where TI/TE still weren't working - I didn't put
  in the translation strings for BSD style curses.
[...]
1.62 -> 1.63 (04/29/96)
  + Fix bug where nvi under BSD style curses wasn't sending
  TI/TE termcap strings when suspending the process.
```

这些条目表明，每当 vi 开始或结束执行面向命令行(commandline-oriented)的动作时，都必须将相应的光标定位命令(TI/TE)发送到终端。检查代码时，遇到这类条目时，您应该想到系统的其他地方也可能存在类似的忽视。在上面的例子中，您应该考虑工作在命令行模式下的其他编辑器命令(比如编辑器环境中的编译)，并检查相关的函数，查找在屏幕模式下正确处理更改的代码。

练习 8.3 分析 apache Web 服务器提供的文档，给出 apache Web 服务器源码的大概组织方式。

练习 8.4 如何使用 Unix 手册页的的结构形式，自动构造简单的测试用例?

练习 8.5 将 BSD Unix 说明[LMKQ88]的目录映射到本书配套盘中源码树的最顶层两级目录。

练习 8.6 在本书配套盘中找出一个已发表算法的实例。对比该算法的已发表版本与它的实现。

练习 8.7 在本书配套盘提供的 TCP 实现中，标识出两个状态变迁的实例，同时标识出 TCP 状态变迁图中对应的更改。①

练习 8.8 在本书配套盘中，找出 3 例使用 Perl 内部接口的例子，要求使用的 Perl 内部接口必须已经编入文档。

练习 8.9 归类 Unix 手册页中“Bugs”节中描述的问题所属的类型，并将它们制成表格，按照它们出现的频率排序。讨论您获得的结果。

练习 8.10 在大型应用程序的修订控制系统变更日志中，查找单词“buffer overflow”。根据单词出现的频度绘制图表。

练习 8.11 设计一个简单的过程或工具，使用修订控制系统的数据库，将特定文件或文件区域发生变更的频率可视化。

---

① doc/rfc793.txt: p. 21

8.3 文档存在的问题

在阅读文档时，要时刻注意，文档常常会提供不恰当的信息，误导我们对源代码的理解。两种不同类型的歪曲是未记录的特性(undocumented feature)和理想化表述(idealized presentation).

软件代码中已经实现的特性可能由于许多不同的原因(有些是正当的、可以理解的)故意没有编入文档。特性之所以没有记录可能是因为该特性：

- 并非正式支持；
- 只是作为一种支持机制，提供给合适的、经过培训的工程师使用；
- 试验性或打算用在将来的版本中；
- 由产品提供商使用，从而在竞争中取得优势；
- 实现的不好；
- 安全威胁；
- 仅仅供少数用户或产品版本使用；
- 特洛伊木马、时间炸弹或后门。

特性没有归档有时纯粹是出于工作上的疏忽，当这些特性在产品开发周期的后期加入时，尤其如此。例如，apropos 命令的-C 标志允许指定一个配置文件，这个标志就没有记录在 apropos 命令的手册页中；①②这就可能是一个疏忽。在阅读代码时，要警惕那些未归档的特性。将每个实例归类为合理、疏忽或有害，相应地决定是否应该修复代码或文档。

在说明源代码的过程中，文档常常犯的第二种错误是对系统的理想化表述。有时，文档在描述系统时，并非按照已完成的实现，而是按照系统应该的样子或将来的实现。造成这种差异的本意常常是可敬的：文档的作者可能基于功能性规格说明来编写最终用户文档，并真诚地希望系统能够按照规定来实现。其他情况下，当结构性的更改在软件发布之前仓促实现时，构架性的设计文档没有随之更新。所有的情况中，传递给您——代码阅读者——的消息是，请降低您给予文档的信任级别。

文档不同于代码，代码可以容易地由编译器或回归测试进行检验，文档却无法做到这一点，因此，文档有时还可能自相矛盾。例如，NetBSD 中 last 命令的手册页在命令说明中记录了一个-T 选项，但在提纲中并没有列出。③④ 如果文档中存在大量的这类问题(不是出现在我们提供的例子中)，那么说明文档的编写与复查都比较草率，或者我们应该以一种豁达的思维去阅读它们。

有时，文档还可能使用一种约定俗成的语言，通常是为了说得更为简短。本书配套盘中，大约可以找出 40 处使用单词 grok 的不同实例。如果您不熟悉这个词，并不是因为您对现代英语的了解不够。Grok 是 Robert A. Heinlein 的科幻小说 *Stranger in a Strange*

① netbsdsrc/usr. bin/apropos/apropos. 1

② doc/apropos. pdf

③ netbsdsrc/usr. bin/last/last. 1

④ doc/last. pdf

Land[Hei61]中火星人的词汇，字面意思是“喝”，喻义是“成为他们/它们的一员”。在源代码文档中，单词 grok 一般意味着“理解”。①

```
// for linkers that cant grok long names.
# define ACE_Cleanup_Strategy ACLE
```

如果未知的或特殊用法的单词阻碍了您对代码的理解，可以试着在文档的术语表（如果存在的话）、*New Hacker's Dictionary*[Ray96]或在 Web 搜索引擎中查找它们。

练习 8.12 本书配套盘中包括超过 40 处对未归档行为的引用。找出它们，并讨论造成文档差异最常见的原因。

8.4 其他文档来源

在寻找源代码文档时，要考虑到非传统的来源，比如注释、规范、出版物、测试用例、邮件列表、新闻组、修订日志、问题跟踪数据库、营销材料和源代码本身。在研究大量的代码时，人们一般会寻找更为正式的来源，比如需求和设计文档，常常会忽略嵌在注释中的文档。然而，源代码注释经常比相应的正式文档维护得更好，并且常常隐藏着珍贵的信息，有时甚至包括用 ASCII 码画成的图示。图 8.3 中的图示和图 8.4② 中正式的数学证明就是典型的例子，它们都来自于实际的代码注释。ASCII 码图示描绘了声频接口硬件（左上）的块结构③，

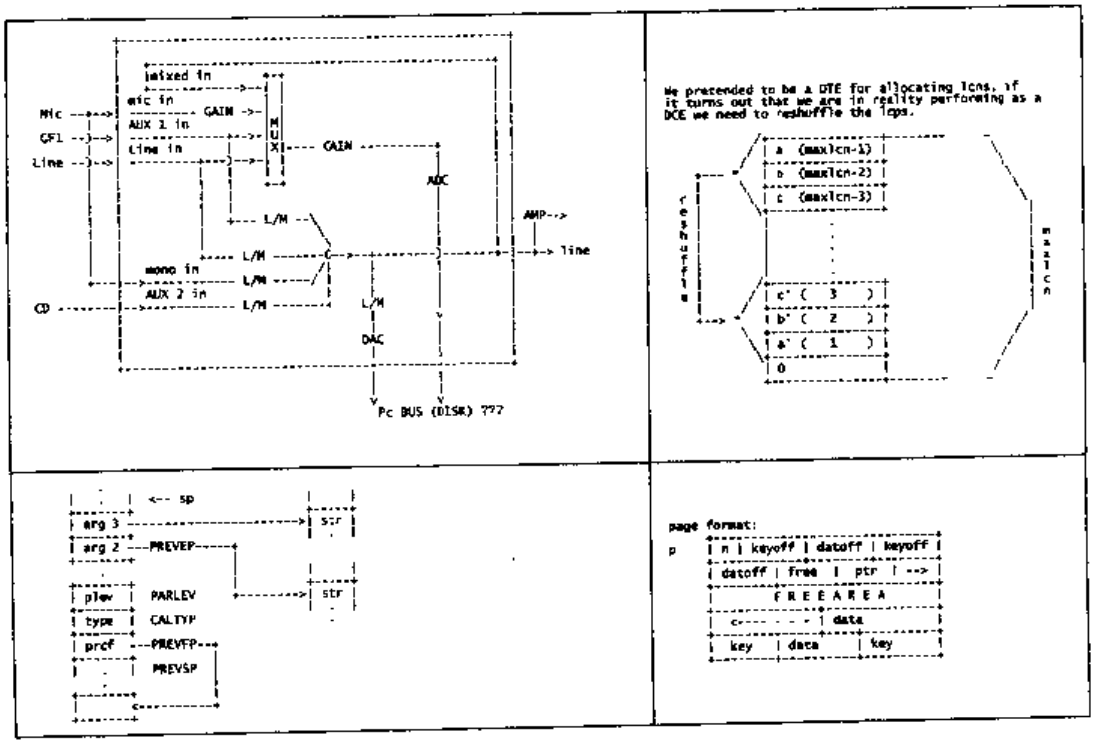


图 8.3 源代码注释中的 ASCII 码图示

① ace/ace/Cleanup_Strategies_T. h:24-25

② netbsdsrc/sys/kern/kern_synch. c:102-135

③ netbsdsrc/sys/dev/ic/cs4231reg. h:44-75

X.25 联网代码中逻辑通道重组过程[①](右上),m4 宏处理器基于栈的数据结构[②](左下),以及散列数据库 db 的实现中采用的页面格式[③](右下)。然而,要注意,精心制作的图示,在代码更改时很少能够同步更新。

总是要以批判的态度来看待文档。由于文档永远不会执行,对文档的测试和正式复查(如果有的话)也很少达到对代码的同样水平,所以文档常常会误导读者,或者完全错误。例如,图 8.4 中的注释就存在一些问题。[④]

```
* We wish to prove that the system's computation of decay
* will always fulfill the equation:
*  decay ** (5 * loadavg) ~= .1
*
* If we compute b as:
*  b = 2 * loadavg
* then
*  decay = b / (b + 1)
*
* We now need to prove two things:
* 1) Given factor ** (5 * loadavg) ~= .1, prove factor == b/(b+1)
* 2) Given b/(b+1) ** power ~= .1, prove power == (5 * loadavg)
*
* Facts:
*         For x close to zero, exp(x) =~ 1 + x, since
*              exp(x) = 0! + x**1/1! + x**2/2! + ... .
*              therefore exp(-1/b) =~ 1 - (1/b) = (b-1)/b.
*         For x close to zero, ln(1+x) =~ x, since
*              ln(1+x) = x - x**2/2 + x**3/3 - ...     -1<x<1
*              therefore ln(b/(b+1)) = ln(1 - 1/(b+1)) =~ -1/(b+1).
*         ln(.1) =~ -2.30
*
* Proof of (1):
*    Solve (factor)**(power) =~ .1 given power (5*loadav):
* solving for factor,
*      ln(factor) =~ (-2.30/5*loadav), or
*      factor =~ exp(-1/((5/2.30)*loadav)) =~ exp(-1/(2*loadav)) =
*          exp(-1/b) =~ (b-1)/b =~ b/(b+1).                    QED
*
* Proof of (2):
*    Solve (factor)**(power) =~ .1 given factor == (b/(b+1)):
* solving for power,
*      power*ln(b/(b+1)) =~ -2.30, or
*      power =~ 2.3 * (b + 1) = 4.6*loadav + 2.3 =~ 5*loadav.  QED
```

图 8.4　源代码注释中的数学证明

- 它莫名其妙地前 3 次使用符号～＝表示“约等于”,之后使用＝～。
- 它使用 loadavg 和 loadav 来指代同一数量。其余的代码,在图中没有列出,使用 loa-

① netbsdsrc/sys/netccitt/pk_subr.c:567-591

② netbsdsrc/usr.bin/m4/mdef.h:155-174

③ netbsdsrc/lib/libc/db/hash/page.h:48-59

④ Guy Steele 提供。

dav 和一个名为 ldavg 的结构字段!

- 取近似值十分随便,也未证明它们是否足够接近。例如,除法 5/2.3 的结果约等于 2。这就相当于将原来表达式中的常量.1 替换为 0.08,20%的误差。这段注释并未证明这种近似的正确性。
- 对 exp(x)和 ln(1+x)采取的近似,依赖于变量 x 要"接近 0",但此处没有解释如何接近才算是足够接近,也没有提供证据或解释来证明 x 在实际的应用程序中确实足够接近 0 的假设。(稍加分析,可以得出,如果 x"接近 0",loadavg 需要"大",但这里也未讨论这一点。)
- 在证明之前,注释系统地罗列出两个有关近似的有用"事实",证明就是依赖于这些事实。这种做法不错。但是,证明还依赖于第三个关于近似的事实,这个事实没有提前列出:如果 b"足够大",则(b-1)/b =~ b/(b+1)。第一个项证明的 QED(证明完毕,或称结论)行简单地采用了执行较快的一个。
- 最后,这段注释实际上过于冗长,因为两个证明是冗余的! 它们根本上证明的是同一个数学事实,不过是从两个方面面已。

另一种类型的文档与源代码密切相关,那就是修订控制系统。这个储存库包含源代码演进的详细历史,并且常常评注每次更改的理由。在 6.5 节中,我们分析了,在阅读代码的过程中如何利用这类系统。与修订控制系统相关联的经常是一个问题跟踪数据库(issue-tracking database)。其中存储着详细的 bug 报告、变更请求和其他维护文档。当这些内容来自于开发组织的内部时,它们可能会提供代码在设计与实现方面的背景知识。

尽管这种重言式好像是一种矛盾修饰法,但是有时源代码确实能够作为它自己的文档。除了明显自我注释的代码以外,有时可以将某些源代码当作是一种规格说明,即使实际的代码没有实现代码应该实现的潜在意图。考虑下面(没有实际意义)的外壳脚本。①

```
for file in $*  ; do echo $file":" ; done
```

这段代码将 $* 列表中用空格隔开的每个参数在单独的行中显示出来。然而,根据这个循环结构,很明显这段代码的意图是在单独的行中显示 $* 列表中的每个文件名。尽管 file 变量应该更为贴切地命名为 filename,并且,这段代码在文件名中包含空格时不能够正常工作,我们依旧要将这段代码应该做到的事情看作是它的规格说明,而非依照这段代码实际做到的事情。

最后,经常可以找到来自开发组织外围或者外部的补充文档。我们可以将规范文档(standards document)作为实现特定标准的软件(例如,MP3 播放器)的功能性规格说明。类似地,在期刊或会议的出版物中,常常能够找出给定的设计、系统、算法和实现的介绍。如果软件系统的验证和确认工作由独立的测试组来处理,可以将他们的测试用例作为功能性规格说明的替代或补充。实在找不到其他文档时,甚至营销材料(marketing material)也能够提供一系列系统特性的(夸大)清单。在 Web 上搜索相关的讨论、非官方信息、FAQ 页和用户经验是一种通用的手段;已归档的新闻组或邮件列表有时可能会揭示您所阅读的代码背后的设计原理。当代码属于开放源码时,一种特别有效的搜索策略是,从您所阅读的代码

① netbsdsrc/usr.bin/lorder/lorder.sh:82

中，找出 3 或 4 个非单词标识符（例如，bbp，indouble，addch），将它们作为搜索项，在主要的搜索引擎上进行搜索。开放源码的代码还提供一种可能，就是可以联系最初的作者。不要滥用这项特权，如果您从开放源码中得到了帮助，一定要对社团有所回报。切记：大多数开放源码项目是由（一般都超负荷工作）志愿者开发和维护的。

练习 8.13　创建一份 apache Web 服务器或 Perl 文档来源的清单。基于它们所提供信息的类型进行分类。力争覆盖更广的范围（例如，规格说明、设计、用户文档），而非完全性。

8.5　常见的开放源码文档格式

文档一般有两种类型：二进制文件，它们的生成和阅读都要使用专利产品，比如 Microsoft Word 或 Adobe FrameMaker；文本文件，其中包含标记语言（markup language）形式的结构和格式化命令。在处理开放源码软件时，最常遇到的文档格式如下：

(1) troff

troff 文档编排系统一般与传统的 man 或新的 BSD mdoc 宏一同使用，共同创建 Unix 手册页。Unix 手册中其他说明性的部分也用 troff 排版。文档编排命令不是在行的开头以. 开始，就是前面有反斜杠。图 8.5[①] 是使用 mdoc 宏编排的 Unix 手册页，表 8.2 中列出了最常用的 troff 命令、man 和 mdoc 宏。

```
.\"     $NetBSD: cut.1,v 1.7.2.1 1997/11/04 22:25:05 mellon Exp $
.Dd June 6, 1993
.Dt CUT 1
.Os
.Sh NAME
.Nm cut
.Nd select portions of each line of a file
(...)
.Sh DESCRIPTION
The
.Nm
utility selects portions of each line (as specified by
.Ar list
)
from each
.Ar file
(or the standard input by default), and writes them to the
standard output.
```

注释
日期命令
文档的标题和节
操作系统陈述
节标题
命令名
描述
含有嵌入宏的文字

图 8.5　mdoc 格式的文档

(2) Texinfo

Texinfo 的宏由 TeX 文档编排系统来处理。许多 GNU 计划下实现的项目都使用它来创建在线和打印文档。Texinfo 命令以一个@字符开头，可以在花括号内或同一行中有参数。图 8.6[②] 是 Texinfo 文件的一个例子，表 8.3 列出了最常用的 Texinfo 命令。

① netbsdsrc/usr.bin/cut/cut.1:1-66

② netbsdsrc/usr.sbin/amd/doc/am-utils.texi:383-395

(3) DocBook

DocBook 是一个 XML/SGML 应用程序，除了其他应用以外，它还用在 FreeBSD 文档项目中。和我们的预想相同，它们的标记用<tag>序列表示；它们的名称一般自我解释。

表 8.2 常见的 troff 命令、Man 和 mdoc 宏

宏	功能
troff	
.\"	注释
.br	换行符
man	
.B	以粗体排版
.BI	切换粗体和斜体字型
.BR	切换粗体和罗马字型
.I	设为斜体字型
.IP	开始缩排段落
.IR	切换斜体和罗马字型
.RB	切换罗马和粗体字型
.SH	节的标题
.TH	标题节
mdoc	
.Ar	命令行参数
.Bl	列表开始
.Dd	以月、日、年的格式指定文档的日期
.Dt	指定文档的标题、节、卷
.Dv	定义变量
.Ic	交互命令
.It	列表项
.Nm	命令名称
.Op	选项
.Os	指定操作系统、版本、发布程序
.Pa	路径或文件名
.Sh	节的标题
.Xr	手册页的交叉索引

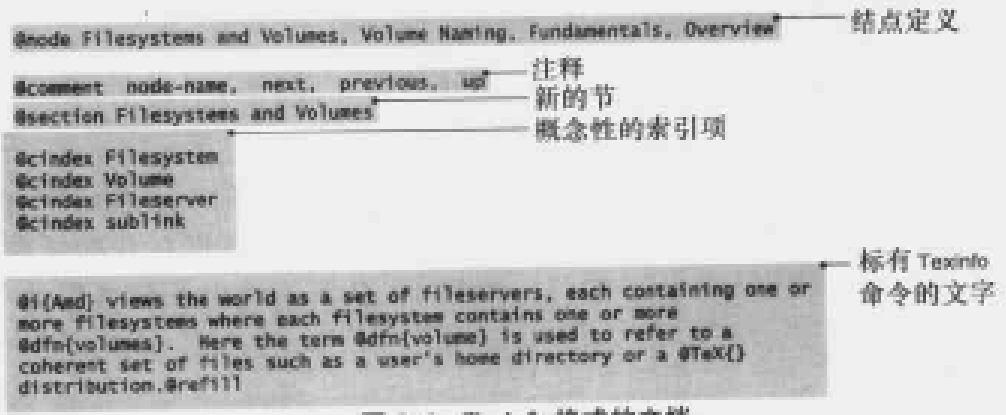


图 8.6 Texinfo 格式的文档

表 8.3　常见的 Texinfo 命令

命令	功能
@c	该行余下的部分为注释
@cindex	向概念索引中增加一项
@code {*sample*}	sample 是一段程序
@emph {*text*}	将 text 加重显示
@end *environment*	设定 environment 的结尾
@example	开始一段代码示例
@file {*name*}	标记一个文件名或目录名
@item	开始一个逐条列举或枚举段落
@node *name*, *next*, *prev*, *up*	定义一个超文本结点
@pxref {*node name*}, @xref	创建一项交叉索引
@samp {*text*}	text 是一段代码
@section *title*	开始一个新的节
@strong {*text*}	将 text 加重(粗体)显示
@var {*name*}	name 是一个伪变量(metasyntactic variable)

(4) javadoc

javadoc 应用程序能够处理用特殊标记的注释进行评注的 Java 源代码，生成多种格式的文档。javadoc 的注释都以/**开头，可以包含标有@符号的标记。嵌入在文本中的标记要括在花括号中。表 8.4 是 javadoc 命令的汇总，图 6.15 显示了带有嵌入式 javadoc 注释的 Java 代码。

表 8.4　javadoc 标记

命令	功能
@author	用来标示作者
{@docRoot}	从 root 生成文档的相对路径
@deprecated	添加注释，表明一项不赞成的特性
@exception or @throws	表明方法抛出的异常
{@link}	添加到另外 Java 代码的链接
@param	指定方法的参数
@return	指定返回值
@see	在"see also"标题下添加一个链接
@serial	表明默认的可序列化字段
@serialData	指定序列数据的类型和次序
@serialField	将字段记录为可序列化
@since	表明某项特性的首次发布
@version	支持某个类或成员的软件版本

(5) Doxygen

Doxygen[①] 文档系统针对的是以 C++、Java、IDL 和 C 编写的源代码。它可以从一系列

① *http://www.stack.nl/dimitri/doxygen/*

适当标注的源代码文件，生成在线文档(HTML 格式)和离线参考手册(LaTeX 格式)。它还可以生成 RTF、Postscript、超链接 PDF 和 Unix man 手册页。

基于标记语言的文档一般与源代码集成得更为紧密，这是因为这类文档常常使用修订控制系统进行维护，能使用文本处理工具自动操纵和生成，并且完全集成到产品的编译过程中。更为紧密的集成是将文档作为源码的一部分，常用在 Java 源代码中的 CWEB 系统和 javadoc 注释即为如此。

在阅读大型系统的文档时，首先要熟悉文档的总体结构和约定。例如，您可能已经注意到 Java 平台 API 的文档是围绕包和类的平面清单进行组织的，Unix 手册页一般围绕表 8.5 中编号的节进行构造。

表 8.5 Unix 参考手册中的节

节	内容
1	一般命令(工具和实用程序)
2	操作系统调用
3	C 和其他语言的库
4	特殊的文件和硬件支持
5	文件格式
6	在线游戏
7	其他信息页
8	系统维护和操作命令
9	内核内幕

在对付体积庞大的文档时，可以使用工具，或将文本输出到高品质输出设备上，比如激光打印机，来提高阅读的效率。我们在第 10 章分析的许多工具都可以用在基于标记语言的文档的源代码上。在源代码上应用这些工具，可以像搜索文本一样搜索标记命令。例如，我们使用下面的外壳命令创建一个列表，其中包含对参数分析程序 getopt 的调用以及与每个 Unix 命令对应的标志文档。

```
(
    find . -name '*.c' -print | xargs grep getopt
    find . -name '*.1' -print | xargs egrep '^\.(Op|Fl)'
) | sort -u
```

我们使用上面的命令找出文档与 Unix 工具的具体实现之间的差异。第一个 find 流水线在所有的 C 源代码文件中搜索对函数 getopt 的调用；这些调用中一般包含程序所接受的参数列表。第二个 find 流水线在所有的手册页中搜索介绍选项或标记的行；它们都用宏序列.Op 或.Fl 标记。由于上面 grep 搜索的输出，都以与特定模式相匹配的文件名开头，对它们进行排序可以生成一个归档选项和对应的 getopt 分析参数的正确合并清单，其中的条目如下：

```
cut.1:.Fl b Ar list
cut.1:.Fl c Ar list
cut.1:.Fl f Ar list
cut.1:.Op Ar
```

```
cut.1:.Op Fl d Ar delim
cut.1:.Op Fl n
cut.1:.Op Fl s
cut.c:while ((ch = getopt(argc, argv, "b:c:d:f:sn")) != - 1)
```

很多人可能不知道，我们习惯性地在计算机屏幕上以 75dpi 进行阅读的一些文档格式，可以容易地格式化，在 1200dpi 激光打印机上输出。下面是如何将一些常见的文档格式排版为高品质的输出。

(6) Unix 手册页

某些版本的 man 命令支持使用-t 标志，生成设备无关或 Postscript 输出；在其他系统上，可以使用如下的序列直接将手册页排版成 Postscript。

```
gnuzip -c /usr/share/man/man1/ls.1.gz | grcff -Tps -man
troff -Tpsc /usr/share/man/man1/ls.1 | psdit
```

(7) javadoc 注释

除了 javadoc 的标准 HTML 输出格式以外，许多 doclet(定制的 javadoc 后端)可以与 javadoc 一同使用，创建 MIF(Adobe FrameMaker 交换格式)、RTF(Microsoft Rich Text Format)、LaTeX 和 DocBook 格式的文档。

(8) Texinfo 文档

可以在. texi 源文件(不是处理后的. info 文件)上运行 TeX 程序，排版 Texinfo 文件。

(9) DocBook 文件

将 Jade 或 OpenJade 包与 docBook 样式表一同使用，输出带格式的文本，如 RTF 或 Tex。

练习 8.14　比较我们描述的文档格式在可用性、易读性、提供的特性，以及用特殊工具进行自动化处理的适用性。

练习 8.15　在本书配套盘中，找出每种不同的文件格式，并试着在一种高品质输出设备上进行排版输出。讨论您所遇到的困难。

进　阶　读　物

文学化编程(literate programming)是指在程序中嵌入介绍程序运作的文档[Knu92]。在 *Communications of the ACE*[BKM86, Han87, Jac87, Ham88, WL89]的旧专栏中可以找到许多文学化编程的例子。Douglas[Dou90]描述了如何使用一系列的项目专用工具来管理错误消息。最初的 troff，以及设备无关的后继者 ditroff 在两种参考著作中进行了介绍[Oss79, Ker82]，更普遍的参考是 Gehani[Geh87]和各种宏指令包(macro package)的在线文档。Knuth 和 Levy[KL93]中对 CWEB 文学化编程系统做了介绍，TeX 在 Knuth[Knu89]，LaTeX 在 Lamport[Lam94]，DocBook 在 Walsh 和 Muellner[WM99]分别做了介绍，Komarinski 等著的[KGM01]介绍了 DocBook 在 Linux 文档项目中的应用。

第9章 系统构架

物理学家能够将他们的错误掩盖起来，但设计师只能建议他们的客户种植一些葡萄藤（来弥补设计上的缺陷）。

Frank Lloyd Wright

从系统的源代码库洞悉系统的构架，不是一件容易做到的事。然而，在识别出重要的构架元素之后，那么，浏览整个系统，了解系统的结构和属性，以及规划增加、修改和重构活动都会变得更为容易。这是因为，一旦提取出系统的构架性特征，我们就立即与系统的创建者共享一套语义丰富的词汇。另外，对系统构架的理解，能够帮助我们了解交互的类型、通信模式和代码结构。

我们首先运用自顶向下的方法，分析如何将多个子系统组织成大型的系统（9.1节），以及如何控制系统的各个单元（9.2节）。在对系统进行分析时，它的总体结构可能并不十分明显，成在我们处理的细节层面上根本不存在。因此，在9.3节中，作为补充，我们自底向上介绍了大量的元素包装方案。

有人说，构架是老人的艺术：在实际工作中，经验丰富的设计者更可能预知特定的设计如何处理具体的问题，以及哪些构架性的解决方案可以组合起来形成一个功能体。在9.4节中，我们介绍通常情况下如何再次应用成功的设计，以及阅读代码时经常遇到的一些构架重用的形式。

9.1 系统的结构

我们分析的许多系统都遵循一种简单的“主程序和子例程”结构。其他的系统采用更为复杂的构架性结构，来组织它们的各个构成子系统。常见的、重要的结构可以归类为少数迥然相异的构架类型：集中式储存库（centralized repository）、数据流（data-flow）、面向对象（object-oriented）或分层（layered）构架。这些构架类型常常结合成一个层次结构（hierarchical structure），用来控制大型系统的复杂性。接下来，我们将独立地分析每种构架类型；但是，要注意，一个系统可以（在重大的系统中也确实如此）同时展示出多种不同的构架类型。以不同的方式检查同一系统、分析系统的不同部分、或使用不同级别的分解，都有可能发现不同的构架类型。

9.1.1 集中式储存库和分布式方案

集中式储存库的构架模型依赖于一个中心过程或数据结构，它在系统中担任控制或信息的集线器（hub）。在超大规模的系统中，协同的任务可能会分配给多个集线器进程。当几个半自治进程需要协同对共享信息或资源的访问时，一般采用集中式储存库构架。这种类

型的例子包括窗口管理器①②、文件③和打印④服务器、以及网络资源管理程序⑤。集中式储存库构架还是众多协作应用程序的基础，比如 World Wide Web、在线消息传递、多人游戏和修订控制程序，如 CVS 和 RCS。⑥

作为一个小而典型的例子，请考虑下面这段描述，该方法在一对一在线交谈会话中连接两个独立 Unix talk 进程。

交谈服务器起到邀请服务器的作用，负责处理那些希望聚集起来进行会话的客户发出的请求。在正常情况下，客户，也就是呼叫发起者，通过向服务器发送一个 LOOP_UP 类型的 CTL_MSG 消息，发起一项约会。接收到这条消息后，服务器会在自身的邀请表中进行查找，检查当前是否有针对该调用者的邀请。如果没有找到，那么呼叫发起者会发送 ANNOUNCE 消息，使得服务器向相应的端口（被呼叫者为请求联系而注册的端口）广播一项宣告。当被呼叫者响应时，本地服务器会根据记录下的邀请信息，用相应的约会地址回应被呼叫者，从而，呼叫发起者和呼叫者程序就可以建立一个流连接，通过这个流连接就可以进行会话。

当集中式储存库和访问它的应用程序分别作为独立的进程运行时，这种构架模型也常常称为客户机-服务器模型。遵守客户机-服务器模型的系统提供一项重要的非功能属性——可集成性(integrability)，该属性允许大型环境中的不同客户程序和服务程序进行无缝地互连。您或许已经注意到，前一段中的 talk 守护协议(daemon protocol)将它的 C 头文件放入了系统的头文件目录 include/protocols，而非应用程序的源代码所在的目录。这实际上构成了一个隐式的承诺，允许其他客户程序和服务程序与之进行互操作，只要它们遵循协议的约定。位于同一目录下的其他头文件指定 dump 备份格式⑦，routed 路由信息交换协议⑧，timed 时间同步机制⑨，以及 rwhod 本地网络用户信息交换⑩。这些规格说明由客户程序和服务程序共享，但要依赖于具体的操作系统头文件。如果期望应用程序在多种操作系统共存的环境中运行，那么，一般要将它们使用的通信协议标准化并正式发布。对于 Internet 协议，这些描述都以 RFC(Request for Comments)文档⑪的形式存储在 Network Information Center，可以公开取得。编制为 RFC 的重要协议包括 IP(RFC-791)，TCP(RFC-793)，SMTP(RFC-821)，FTP(RFC-959)，POP-3(RFC-1939)和 HTTP(RFC-2068)。

大量集中式储存库方案都使用关系型数据库作为数据仓库。客户程序要么直接使用数据库的数据操作语言(一般为 SQL)来访问数据库，要么通过单独的服务进程提供高层、面向

① XFree86-3.3/xc/programs/Xserver
② netbsdsrc/usr.bin/window
③ netbsdsrc/usr.sbin/nfsd
④ netbsdsrc/usr.sbin/lpr/lpd
⑤ netbsdsrc/usr.sbin/dhcp
⑥ netbsdsrc/include/protocols/talkd.h:44-56
⑦ netbsdsrc/include/protocols/dumprestore.h
⑧ netbsdsrc/include/protocols/routed.h
⑨ netbsdsrc/include/protocols/timed.h
⑩ netbsdsrc/include/protocols/rwhod.h
⑪ *ftp://ftp.internic.net/rfc*

任务的服务。这两种方案分别称为二层构架(two-tier architecture)和三层构架(three-tier architecture)。某些情况下,系统会使用单独的中间服务器作为事务监控器(transaction monitor),它提供集中式的事务收集点,提供弹性(resiliency)、冗余(redundancy)、负载分配(load distribution)和消息排序(message sequencing)等功能。

即使不需要数据库提供的数据持久性,系统也常常使用集中式储存库。在这种情况下,储存库一般作为大量不同代码元素的通信集线器。不相关的代码块,能够通过访问储存库互相传递信息,勿需付出努力对程序进行构造,使之适合于数据流。我们将这样的储存库称为黑板系统(blackboard system)。它一般将松散的结构化数据存储为键/值对,在它的程序中含有通信代码块,使用公共的键命名约定进行信息交换。黑板允许其他代码读取键所对应的数据、将数据写入到给定的键、移除一项、以及在给定的项出现或更改时通知其他实体。使用黑板协调不同的进程时,常常会使用一段共享内存区域或一个文件来访问公共的信息。当进程处于不同的系统中时,经常会用到中心服务器和某种请求一回答机制,提供对黑板的远程访问。

apache Web 服务器使用黑板完成隔离请求和应答报头等处理任务,并且,更重要的是,黑板机制允许数十个不同的插入模块与服务器无缝地协同工作。从图 9.1[①] 中,可以看到,服务器默认页面处理器的两个阶段之间如何传递页面内容的长度信息。主页面处理器函数(default_handler[②])调用函数 ap_set_byterange,来检查客户请求的 HTTP Range 报头。通过这种机制,Web 客户能够请求将一个页面分成多个部分进行发送,从而影响到每一部分的长度。当服务器将 HTTP 报头发送回客户后,它需要确定在页面传输完成后,是否需要维护该连接,而后,据此设置相应的报头字段。只有在页面内容的长度确定之后,才能完成上述工作;否则应该终止该连接,表示没有更多的数据。因此,ap_set_keepalive 检查 Content-Length 黑板项,看看是否其他的过程(如 ap_set_byterange)已经为它赋予了一个值。

分布式构架的一个要素就是底层的通信协议。到此为止,我们所分析的客户端-服务器和黑板协议都是针对特殊的应用领域专门设计的。更为通用的方案是使用远程过程调用(remote procedure call)或远程方法调用(remote method invocation)的概念。它允许客户代码直接调用远程服务器端的过程,并接收执行的结果。正如我们所预期的那样,这种用法需要客户端和服务器之间进行相当可观的协调工作,同时还需要足够数量的支持基础——中间件(middleware)。简单地说,最常用的中间件构架包括:

- COBRA(Common Object Request Broker Architecture,公共对象请求代理构架),它是一个与构架和语言无关的规格说明,用来在应用程序和应用程序对象之间进行透明的通信。它由 OMG(Object Management Group)定义并提供支持,OMG 是一个由 700 多个开发人员、提供商和最终用户组成的非盈利性行业联盟。基于 COBRA 的系统中,都有一个对象请求代理(object request broker,ORB),它负责将客户的请求转发到适当的对象。
- DCOM(Distributed Component Object Model,分布式组件对象模型)是一个面向对象的构架,其设计目的是促进分布式、异构环境中软件对象之间的互操作性。由于

① apache/src/main/http_protocol.c:276-510

② apache/src/main/http_core.c:3696

```
API_EXPORT(int) ap_set_byterange(request_rec *r)
{
    [...]
    else if (ranges == 1) {
        [...]
        ap_table_setn(r->headers_out, "Content-Length",
            ap_psprintf(r->pool, "%ld", one_end - one_start + 1));
        [...]
    } else {
        [...]
        ap_table_setn(r->headers_out, "Content-Length",
            ap_psprintf(r->pool, "%ld", length));
    [...]
}
[...]
API_EXPORT(int) ap_set_keepalive(request_rec *r)
{
    [...]
    if ((r->connection->keepalive != -1) &&
        ((r->status == HTTP_NOT_MODIFIED) ||
         (r->status == HTTP_NO_CONTENT) ||
         r->header_only ||
         ap_table_get(r->headers_out, "Content-Length")) ||
         [...]
        ) {
         [...]
        ap_table_setn(r->headers_out, "Keep-Alive",
            ap_psprintf(r->pool, "timeout=%d",
                        r->server->keep_alive_timeout));
    [...]
}
```

向黑板写入"Content-Length"

向黑板中读"Content-Length"

向黑板写入"Keep-Alive"

图 9.1　apache Web 服务器中的黑板

它是 Microsoft 定义的，所以它主要以 Windows 系统为目标。它的演进是 Microsoft .NET 构架。

- RMI(Remote Method Invocation，远程方法调用)是 Java 专有的一项技术，使用这项技术，可以无缝地远程调用不同 Java 虚拟机中的对象。
- Sun 的 RPC 是一个与构架和语言无关的消息传递协议，Sun 微系统公司和其他厂商使用它跨越不同的网络基础设施调用远程过程。许多高层次的领域专用协议，如 NIS(Network Information System)分布式主机配置数据库和 NFS(Network File System)，都构建在 Sun 的 RPC 规格说明之上。

使用某个面向对象中间件系统(CORBA，DCOM，或 RMI)的程序，会在类中定义服务器端方法，扩展相应中间件中特有的基类。在客户端，由另外的类负责找出网络上的服务器，并提供相应的封送处理方法，将调用重定向到服务器。在 Sun RPC 中，服务器可以使用 API 函数注册 RPC 过程，然后，由客户端调用服务器端的过程(图 9.2①②③)。我们概括的所有系统，除了 RMI 之外，都必须处理不同处理器构架下数据表达方式的不同。通过将数据类型封装成专用于在任何主机之间进行通信的、与构架无关的格式，可以解决这个问题。封送处理既可以透明地完成(用接口定义语言(interface definition language，IDL)描述接口，之后编译为相应的封送代码)，也可以调用数据转换函数显式地完成。

① netbsdsrc/usr.sbin/ypserv/ypserv/ypserv.c:125-399

② netbsdsrc/lib/libc/yp/yp_first.c:166-168

③ netbsdsrc/lib/libc/yp/xdryp.c:130-136

图 9.2 yp/NIS 实现中的远程过程调用

9.1.2 数据流构架

当处理过程可以建模、设计和实现成一系列的数据变换时，常常会使用数据流（或管道-过滤器）构架。以创建一个数据库，其中为 Unix 手册页中的每个条目保存一段简短描述为例。例如，对应 getopt 函数的条目（其手册页在 2.3 节中出现过）是：

```
getopt (3) - get option character from command line argument list
```

这项任务可以建模为多个分立的步骤：

(1) 找出所有的手册页。

(2) 移除重复条目。（当单一手册页中记述多个命令时，可能会链接到多个文件名。）

(3) 提取出合适的条目。

(4) 将它们加入到数据库中。

(5) 将数据库安装到最终的位置。

这些步骤中，每一步都从前面一步接收输入，并生成供后继者进行处理的输出。

虽然数据流构架好像相当受限制，但是它们提供许多优点。首先，数据流构架常常模拟现实的过程和人们实际的工作方式。另外，在开发应用数据流构架的系统时，既可以顺序地进行，也可以并行地进行。最后，已实现的数据变换（data transformation）可以容易地重用：Unix 在快速应用程序原型的制作和开发环境上取得的成功，就是根源于对它许多过滤器（filter）程序的积极重用。在批量进行自动数据处理的环境中，经常会采用数据流构架，在对

数据变换工具提供大量支持的平台上尤其如此。另一方面，数据流构架不太适用于设计反应性的系统，比如 GUI 前端和实时应用程序。

数据流构架的一个明显征兆是：程序中使用临时文件或流水线(pipeline)在不同进程间进行通信。makewhatis 命令的实现(用来创建图 9.3[①] 显示的手册页数据库)就是一个典型的例子(图 9.4)。流水线首先创建一个文件，其中是系统中所有手册页的惟一清单。之后，生成的文件由 3 个不同的流水线进行处理，每种流水线分别处理不同类型的手册页：无格式手册页(troff 源代码)、有格式手册页(嵌入控制字符的文本)和压缩格式的手册页。3 个流水线生成的结果都追加到一个单独的文件中，该文件接下来由 sort 进行处理，将残余的重复条目移除(例如，当一个手册页存在有格式和无格式两种形式时，就会发生这种情况)。

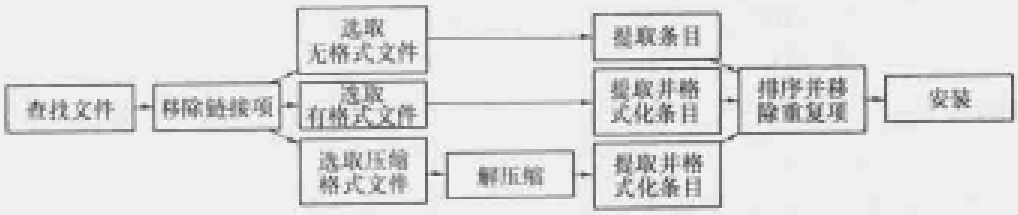


图 9.3　创建手册页说明的数据流图示

```
find $MANDIR \( -type f -o -type l \) -name '*.[0-9]*' -ls |          创建手册页清单
        sort -n |                                                    移除重复项(链接)
        awk '{if (u[$1]) next; u[$1]++ ; print $11}' > $LIST

egrep '\.[1-9]$' $LIST |                                             查找无格式的手册页
xargs /usr/libexec/getNAME |                                         提取条目名
sed -e 's/ [a-zA-Z0-9]* \\-/ -/' > $TMP                              格式化为数据库条目

egrep '\.0$' $LIST |                                                 查找有格式的手册页
while read file                                                      处理每个文件
do
    sed -n -f /usr/share/man/makewhatis.sed $file                    提取并格式化数据库条目
done >> $TMP

egrep '\.[0].(gz|Z)$' $LIST |                                        查找压缩格式的手册页
while read file                                                      处理每个文件
do
    gzip -fdc $file |                                                取得解压缩后的内容
    sed -n -f /usr/share/man/makewhatis.sed                          提取并格式化数据库条目
done >> $TMP

sort -u -o $TMP $TMP                                                 排序数据库，移除重复项

install -o bin -g bin -m 444 $TMP "$MANDIR/whatis.db"                安装到最终的位置

rm -f $LIST $TMP                                                     清理临时文件
```

图 9.4　makewhatis 命令基于过滤器的实现

要理解基于数据流构架的系统如何运作，您需要理解两件事：它的处理过程和处理过程间流动的数据。处理过程一般是数据变换。试着单独了解每个变换的输入和输出数据。如果您所分析的系统中，处理过程使用流水线进行通信，可能需要将流水线的数据内容临时重定向到一个文件中，并分析它的内容。Unix 工具 tee 允许我们设置此类分接头(tap)。当需要用到临时文件时，只需要略去在处理回路结束时移除这些文件的命令。在商业应用中，数

① netbsdsrc/libexec/makewhatis/makewhatis.sh:20-39

据经常由一行行的记录组成，各个字段或者占据固定宽度，或者用特殊的字符分开。然而，数据也可以是固定长度的记录或应用程序专有的对象，或遵循特定二进制流约定，netpbm图像处理程序和sox声音控制工具即为如此。画一个数据流图可能会对了解整个系统的结构有所帮助，如图9.3所示。如果运气好的话，在系统的设计文档中会包括此类图示。

9.1.3 面向对象的结构

采用面向对象结构的系统，将它们的设计建立在维护自身状态并互相作用的对象上。系统的构架由不同的类或对象之间的关系，以及对象交互的方式来定义。现实世界中的系统可能会由数百个不同的类组合而成。例如，Cocoon XML出版框架的源代码[①]包含500多个Java类。确定这类系统的秩序确实是一件困难的任务。幸运的是，您可以依赖标准化的记法和词汇——统一建模语言(Unified Modeling Language，UML)，来表达系统的模型，或阅读现有的文档。另外，许多基于UML的工具允许我们对构架的重要部分进行逆向工程，用UML记法来表示它们。

UML可以用来建模系统生命周期内的方方面面，从整个世界的概念性模型扩展到抽象软件规格说明模型，直至具体的实现模型。UML(正受到轻度记法超载的困扰)，提供不少于9种不同的图来表达这些视图。在分析现实系统的代码时，您一般会将它的结构建模为一个类图或对象图，将它的内部交互建模为序列图(sequence diagram)或协作图(collaboration diagram)。大多数UML图中的基本构成元件是类；图9.5例举出RequestFilterValve[②]类在UML模型中的样子。方框中的3部分分别是类名(class name)、属性(attribute，或property)和操作(operation，或method，方法)。斜体显示的类名表示指定的类是抽象的(定义中使用了abstract关键字，不能直接定义该类的对象。)属性和操作用符号－表示该成员被定义为public(所有的方法都能够访问它)，－表示定义为private的成员(仅仅在类的方法中可以访问)，#表示定义为protected的成员(仅能从类自身及其派生类的成员中访问)。另外，定义为static(属于类，而非类的个体对象)的成员显示时会加下划线；abstract成员显示为斜体。不要让个体成员的定义分散您的注意力；在使用UML对现有系统进行逆向工程时，语法一般遵守该系统的实现语言。模型，比如我们在本节中分析的几个，则适用于支持对象的所有语言，包括C++，C#，Eiffel，Java，Perl，Python，Ruby，Smalltalk和Visual Basic .NET。在绘制系统的构架时，常常会隐藏掉类的部分描述(属性和操作)，从而增加描绘大量类之间的关系所需的空间。

UML类图(class diagram)展示一系列的类、接口、协作和关系，有助于我们理解系统构架在结构上的特征。例如，图9.6的图示说明了Tomcat servlet容器中的valves[③]包的层次：ValveBase抽象类是一个基类，由它派生出6个其他的类。连接类的箭头是UML中表达泛化关系(generalization relationship)的记法(在此为继承)：表明特化类(specialized class)(例如，RemoteAddrValve)共享泛化类(generalized class)(RequestFilterValve)的结

① cocoon/src

② jt4/catalina/src/share/org/apache/catalina/valves/RequestFilterValve.java

③ jt4/catalina/src/share/org/apache/catalina/valves

RequestFilterValve
-static final info: String = "org.apache.catalina.valves.RequestFilterValve/1.0"
#allow: String = null
#deny: String = null
+getAllow(): String
+setAllow(allow: String): void
+getDeny(): String
+setDeny(deny: String): void
+getInfo(): String
+invoke(request:): void
#precalculate(list: String): RE[]
#process(property: String, request:): void

图 9.5　类在 UML 中的表示

构和行为。因此，这个特定的关系表示下面的 Java 代码。①

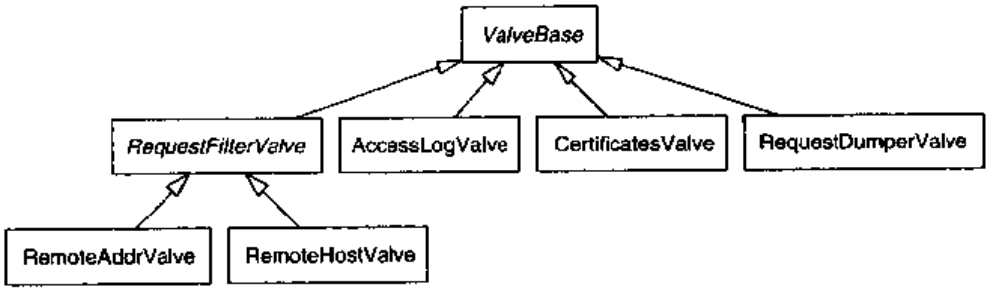


图 9.6　简单的泛化关系

```
public final class RemoteAddrValve
   extends RequestFilterValve {
```

类之间的关系可能会很快变得更为复杂。在支持多重继承的面向对象的语言中，比如 C++，Eiffel 和 Perl，特化类可以继承多个父类。另外，有时类并不从其他类继承功能，而是建立一种实现一个或多个特定接口的契约关系(contractual relationship)。9.1.4 节中论述的例子中出现的部分功能即为这种情况。Java 使用 interface 关键字和有 implements 声明的类定义明确地支持这类关系。您可以使用 UML 的实现关系(realization relationship)建模这种情形，位于箭头首部的分类符规定一个契约(contract)，位于箭头尾部的分类符保证执行这个契约。图 9.7 的类结构中既有实现关系，也有泛化关系。这个模型，描绘了 Tomcat connector 包②中的一些类，包含两个接口——Response 及其特化 HttpResponse，还有 4 个其他的类。ResponseWrapper 和 ResponseBase 类实现了 Response 接口，而 HttpResponseWrapper 和 HttpResponseBase 类实现了 HttpResponse 接口，并分别继承了

① jt4/catalina/src/share/org/apache/catalina/valves/RemoteAddrValve.java:83-84

② jt4/catalina/src/share/org/apache/catalina/connector

ResponseWrapper 和 ResponseBase 的功能。这些关系在模型中比较直观，但要直接从分散在 6 个不同文件、两个目录中的 Java 源代码定义看出这个结构，可能并不容易做到。因此，使用图示来建模面向对象构架中的关系是值得的。

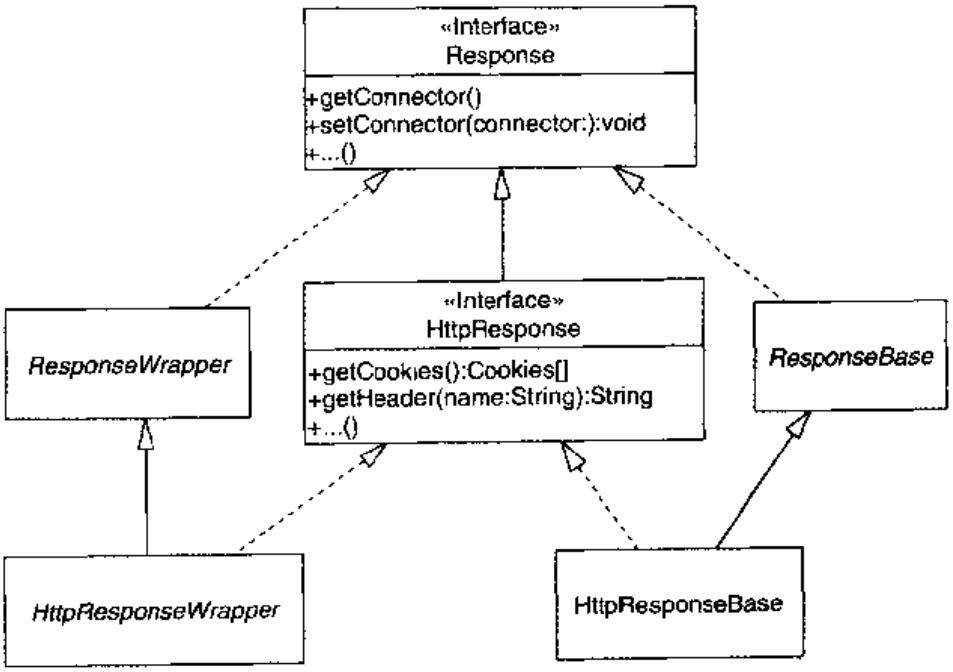


图 9.7 实现关系和泛化关系

尽管手动地在白板或纸上绘制图示可以帮助您理解系统的结构，您还可以受惠于众多能够对现有代码进行逆向工程，构建系统模型的众多工具。我们分析过的所有 UML 图示最初都是由开放源码的 ArgoUML[①] 建模工具生成的。将源代码输入到建模工具中进行逆向工程，常常能够得出系统的构架。我们使用的 ArgoUML 版本可以输入 Java 代码，并确定类之间的泛化和实现关系。商业的建模工具，如 Rational Rose 和 Together ControlCenter，可以输入更多的编程语言，并能在您的模型中检查和显示其他的关系，如类之间的关联。这些工具还能够执行往返逆向工程建模（round-trip engineering modeling），将代码的更改反映到模中，反之亦然。

9.1.4 分层构架

拥有大量同级（alternative peer）子系统的系统，常常按照分层构架进行组织。在这类构架中，每个独立的层对它上面的层提供明确的接口，并使用另一种不同但规范的接口与它下面的层进行通信。在这种框架中，每个层可以使用不同的技术和方法来实现，不会影响整体的构架完整性。计算机网络和操作系统一般都遵循这样的方法：网络物理层不会影响其上传递的数据格式，也不受数据格式的影响；用户应用程序不需要考虑它们用来通信的特定硬件设备。分层构架一般通过堆叠拥有标准化接口的软件组件来实现。每一层向上面的层提供一个虚拟机接口（virtual machine interface）。多数情况下，通过这些接口层的规格说明和

① *http://www.argouml.org*

文档，您能够识别出这类构架。

分层构架的一个重要规则是，较低的层不能使用较高的层——从上到下地应用（注意不是调用，而是应用）。假如任意层可以使用任意层，那么就不能够容易地替换或拿走这些层。例如，网络的物理层实现（例如，Ethernet 或 SONET）不能对其上运行的上层协议（如 TCP）做任何假定。这就保证了其他上层协议（如 UDP/IP 或 NETBIOS）能够在 Ethernet 上实现，而无须修改 Ethernet 的实现。另一个更进一步的例子是，不同 Java 编译器生成的代码能够在任何遵循相同标准的 Java 虚拟机上运行。在分析分层构架时，这个规则给出的暗示是：系统中的层可以将下面的层看作是抽象的实体，并且（只要该层能够满足对它的需求）不并心上面的层如何使用它。

为了具体地说明相对复杂一些的分层构架，我们下面将分析：在 NetBSD 操作系统上运行的 C 语言程序中，一个简单的 fwrite 请求，如何最终成为，一个 Linux 文件系统（位于 Adaptec 控制器连接的 SCSI 磁盘之上）专有的位模式。我们在图 9.8 中列举出服务于这个请求的不同的函数调用、每个函数所表达的层、以及该层的作用。当程序作为一个用户进程执行时，fwrite① 函数提供 stdio 库缓冲，而 write② 系统调用处理标准的 POSIX 系统调用接口。当系统调用到达内核时，序列 sys_write③→vn_write④→VOP_WRITE⑤ 对系统调用进行译码，将它定向到恰当的文件系统专有函数，ext2fs_write⑥（本例中为 Linux ext2 文件系统）。从该处开始，序列 bwrite⑦→VOP_STRATEGY⑧→ufs_strategy⑨ 将数据写入到设备上恰当的逻辑块中，而序列 VOP_STRATEGY⑩→spec_strategy⑪→sdstrategy⑫ 将请求定向到特定设备（在本例中是 SCSI 磁盘）的排队处理。本例中，硬件请求的完成，使用中断从设备向上传播到通知函数（notification function）；这些层并不直接与上面的层发生交互，而是使用 sleep/wakeup 接口进行同步。因而，设备中断——标志命令的完成，首先由 INTR 进行处理⑬，然后传播到设备专有的函数（aha_intr）⑭，再到处理 SCSI 专有控制块的函数（aha_finish_ccbs）⑮，再到处理命令结果的函数（aha_done）⑯，并最终到达通知等待进程，排

① netbsdsrc/lib/libc/stdio/fwrite. c:56-59
② netbsdsrc/sys/kern/syscalls. master:52-53
③ netbsdsrc/sys/kern/sys_generic. c:228-286
④ netbsdsrc/sys/kern/vfs_vnops. c:370-394
⑤ netbsdsrc/sys/kern/vnode_if. src:98-103
⑥ netbsdsrc/sys/ufs/ext2fs/ext2fs_readwrite. c:169-298
⑦ netbsdsrc/sys/kern/vfs_bio. c:297-350
⑧ netbsdsrc/sys/kern/vnode_if. src:228-230
⑨ netbsdsrc/sys/ufs/ufs/ufs_vnops. c:1616-1651
⑩ netbsdsrc/sys/kern/vnode_if. src:228-230
⑪ netbsdsrc/sys/miscfs/specfs/spec_vnops. c:479-489
⑫ netbsdsrc/sys/dev/scsipi/sd. c:429-492
⑬ netbsdsrc/sys/arch/i386/isa/vector. s:148-200
⑭ netbsdsrc/sys/dev/ic/aha. c:464-508
⑮ netbsdsrc/sys/dev/ic/aha. c:374-459
⑯ netbsdsrc/sys/dev/ic/aha. c:836-921

队新请求的函数(scsipi_done)①。

	Function name	Layer	Role
↓	fwrite	stdio library	User-space buffering
↓	write	System call	POSIX interface
	User / kernel-space boundary		
↓	sys_write	Kernel entry point	Argument processing
↓	vn_write	Vnode	Adapt to vnode interface
↓	VOP_WRITE	Vnode switch	Multiple interfaces
↓	ext2fs_write	File system	Data structure
↓	bwrite	Kernel buffer cache	Kernel-space buffering
↓	VOP_STRATEGY	Vnode switch	Multiple interfaces
↓	ufs_strategy	Block	Logical blocks
↓	VOP_STRATEGY	Vnode switch	Multiple interfaces
↓	spec_strategy	Device switch	Multiple devices
	sdstrategy	Asynchronous operations	I/O queues
	Synchronous / asynchronous code boundary		
	scsipi_done	SCSI interface	Queuing and process notification
↑	aha_done	Device commands	Result processing
↑	aha_finish_ccbs	Command control blocks	Result processing
↑	aha_intr	Device interrupt	Message box handling
↑	INTR	System interrupt	Interrupt processing

图 9.8 文件写入操作:从用户程序到设备

层的接口既可以是支持特定概念的互补函数族,也可以是一系列支持同一抽象接口不同底层实现的可互换函数。例如,在我们分析的代码中,write 系统调用,和其他函数如 read,open,close,fork 和 exec 一起,构成了操作系统调用接口层的一部分。类似地,bwrite 和函数 bremfree,bufinit,bio_doread,bread,brelse,biowait 和 biodone 一起,提供了内核缓冲接口②。另一方面,ext2fs_write 与 msdosfs_write③,nfsspec_write④,union_write⑤ 和 ufsspec_write⑥ 一同,形成文件系统写入的一个具体实现(抽象接口规格说明)。类似地,

① netbsdsrc/sys/dev/scsipi/scsipi_base.c:266-346
② netbsdsrc/sys/kern/vfs_bio.c
③ netbsdsrc/sys/msdosfs/msdosfs_vnops.c:501
④ netbsdsrc/sys/nfs/nfs_vnops.c:3194
⑤ netbsdsrc/sys/miscfs/union/union_vnops.c:917
⑥ netbsdsrc/sys/ufs/ufs/ufs_vnops.c:1707

aha_intr 和 asc_intr[①]，fxp_intr[②]，ncr_intr[③]，pcic_intr[④]，px_intr[⑤]，sbdsp_intr[⑥]，以及其他函数一起处理设备的中断。在我们介绍的例子中，有些层的存在只是为了依据具体的请求，对不同的层接口进行多路复用(multiplex)和解多数复用(demultiplex)。这项功能使用由函数指针构成的数组来实现，如下所示[⑦]。

```
struct vnodeopv_entry_desc ext2fs_vnodeop_entries[] = {
    { &vop_default_desc, vn_default_error },
    { &vop_lookup_desc, ext2fs_lookup }, /* lookup */
    { &vop_create_desc, ext2fs_create }, /* create */
    [...]
    { &vop_update_desc, ext2fs_update }, /* update */
    { &vop_bwrite_desc, vn_bwrite },     /* bwrite */
    { (struct vnodeop_desc* )NULL, (int(*) __P((void*)))NULL }
};
```

用面向对象的语言实现的系统，使用虚方法调用直接表达对层接口的多路复用操作。

9.1.5　层次

构建和理解复杂构架的核心要素是层次分解(hierarchical decomposition)的概念。层次常常用来以结构化和易于浏览的方式，通过将系统的各个部分分开考虑，控制系统的复杂性。层次分解常常与系统的构架特征无关；构架既可以以层次的方式定义，也可以定义为一个或多个相互正交、相互影响的不同层次。重要的是，要了解系统可以使用不同的、独特的层次分解模型跨各种坐标轴进行组织。作为一个具体的例子，请考虑图 9.8 中描绘的层(显然涉及对各种考虑的层次分解)，将它们与包含底层函数的目录结构进行对比。这二者并不匹配，因为它们分属两个不同的层次模型，一个关心的是系统中的控制流程和数据流，另一个关注的是源代码的组织。

在分析系统的构架时，从下面的实例中，我们可以探知系统的层次结构：

- 源代码在目录中的安排
- 静态或动态过程调用图
- namespace(C++)，package(Ada，Java，Perl)，或标识符名称
- 类和接口的继承关系
- 结构化黑板项(blackboard entry)的命名
- 内部类和嵌套过程
- 异构数据结构和对象关联中的导航

① netbsdsrc/sys/dev/tc/asc. c:824
② netbsdsrc/sys/dev/pci/if_fxp. c:908
③ netbsdsrc/sys/dev/pci/ncr. c:4240
④ netbsdsrc/sys/dev/ic/i82365. c:467
⑤ netbsdsrc/sys/dev/tc/px. c:153
⑥ netbsdsrc/sys/dev/isa/sbdsp. c:1481
⑦ netbsdsrc/sys/ufs/ext2fs/ext2fs_vnops. c:1389-1433

要将构架的不同(甚至可能互相冲突)层次分解方法,看作是同一场景的不同角度。树形图示可以帮助您建模任何遇到的层次。

9.1.6 切片

在推导程序结构的细节时,一个有价值的概念性工具就是切片(slicing)。非正式地,您可以将程序片(program slice)看作能够影响变量的值的一段程序。因而,切片准则(slicing criterion)(组成切片的各个部分的规格说明)是一组变量和一个程序单元。图 9.9 说明了这项技术[①]。最初的代码(图 9.9:2)依照变量 charct 和图 9.9:1 所示的程序点进行切片,如图 9.9:3 所示,同时还依照变量 linect 和同一程序点,如图 9.9:4 所示。作为对程序进行分析和理解的一项技术,切片技术的价值在于,它将程序的数据与控制依存关系(数据流和语句执行的次序)放到一起。因此,切片允许您将不分次序的语句集合进行归组,重新对大型系统进行严格准确地层次分解。通过将大段的代码按照一个关键的结果变量进行切片,可以去除不重要的信息,并且有希望通过逆向工程得出底层的设计或算法。切片还提供一种模块内聚性(cohesion)的量度,内聚性是指模块中各个处理单元之间的相关性。模块内高度的内聚性证明,将模块各个单元组合在一起的构架性设计是正确的;模块内低水平的内聚性则表明,这些出于巧合归结到一起的模块单元还可以进行重新归组。为了确定模块的内聚性,可以基于模块不同的输出变量创建一系列的模块切片。这些切片之间的交互作用(公共的代码行)表明处理单元之间的关系;它们的大小是模块内聚性的一个量度。对不同的模块使用相同的过程,还可以测量出模块的耦合度(coupling):模块间相互关联(interrelatedness)的程度。高度的耦合度表明,代码难以理解与更改;这种情况下,切片可以揭示不同模块之间相互关联的程度。

练习 9.1 说说看,您如何确定您正在使用的关系型数据库中采用什么样的数据库模式。设计一个简单的工具,将该模式转换成一份编排精致的报告。

练习 9.2 全局变量会妨碍代码的可读性。黑板方案(blackboard approach)有什么不同呢?

练习 9.3 使用中间件的开放源码项目并不多见,为什么?

练习 9.4 在本书配套盘中找出基于数据流构架的应用程序。您采用的查找策略是什么呢?

练习 9.5 下载 GNU tar 程序[②],并以数据流图的形式说明,程序如何处理远程备份、压缩和固定块(fixed block)I/O。

练习 9.6 使用一种 UML 建模工具,逆向推导出本书配套盘上某个面向对象应用程序

① netbsdsrc/usr.bin/wc/wc.c:202-225

② *http://www.gnu.org/software/tar*

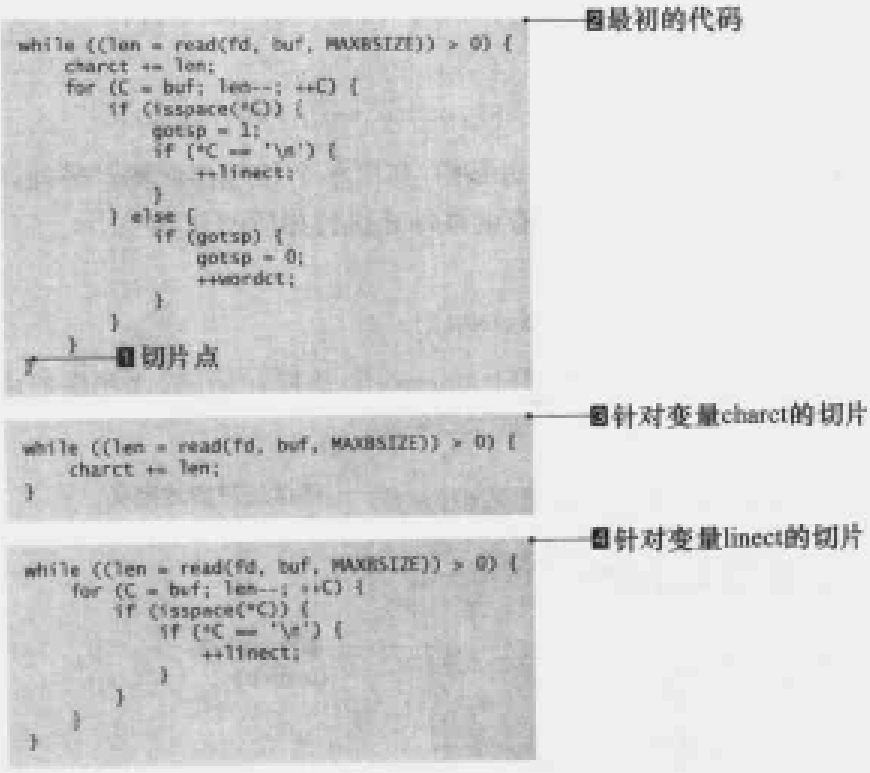


图 9.9 程序切片的例子

的结构。

练习 9.7 找出 OSI 网络层与 NetBSD 联网代码的实现之间的对应关系。

练习 9.8 切片技术能够帮助您理解面向对象的程序吗？讨论得墨忒耳定律(Law of Demeter，见进阶读物一节)如何影响程序切片的适用性？

9.2 控制模型

系统的控制模型描述构成系统的各个子系统之间如何进行交互。和我们在前一节中分析过的系统结构一样，许多系统采用一种简单的、集中化的、单线程的调用返回控制模型(call and return control model)。我们分析的一些系统结构隐式地基于一种特殊的控制模型：在数据流结构中，子系统受控于数据的流动，而在面向对象的结构中，控制一般通过方法调用来协调。其他常见的、重要的控制模型有事件驱动型、基于系统管理器或围绕状态变迁。

9.2.1 事件驱动的系统

在许多系统中，控制决策通过响应外部生成的事件来完成。这类基于事件驱动构架(event-driven architecture)的系统涉及所有可能的软件抽象：从底层由中断驱动的汇编代码处理器例程和进程调度，到高层 GUI 结构的实现和关系型数据库的触发器。事件驱动系统

的实现可以采用多种不同的形式。事件可以广播给一系列的事件侦听者(listener)。①

```
for (int i= 0; i< list.length; i++ )
    ((ContainerListener) list[i]).containerEvent(event);
```

或者,如同经常硬件中断中经常出现的那样,调用单个处理器例程。另外,有些应用程序围绕一个事件循环构建,该循环不断地查询事件并执行相应的处理②。

```
while (XtAppPending(appCtx))
   XtAppProcessEvent(appCtx, XtIMAll);
```

其他的一些应用程序可能与某种框架(framework)集成在一起,这些框架中,事件处理例程可以注册处理指定的事件,如图 9.10 所示①④⑤。

图 9.10 Xt 中隐式的事件注册和处理

事件可能在没有任何预先安排的情况下发送到一个应用程序,因此,事件特别适合于现

① jt4/catalina/src/share/org/apache/catalina/core/ContainerBase.java:1330-1331

② XFree86-3.3/contrib/programs/xfontsel/xfontsel.c:1331

③ XFree86-3.3/contrib/programs/xman/Xman.ad:67-78

④ XFree86-3.3/contrib/programs/xman/main.c:119-221

⑤ XFree86-3.3/contrib/programs/xman/handler.c:643-652

代基于GUI的环境，这些环境中，用事件将不同输入设备、窗口管理器和操作系统生成的数据导向众多正在执行的应用程序。此类事件一般包括键盘和鼠标移动、接收到应用程序焦点、菜单选取、滚动条移动、按钮压下、电源管理系统和进程间通信生成的事件。显式地处理这些事件可能致使应用程序的结构变得十分复杂，如图9.11所示①。因此，大量GUI应用程序框架，比如Microsoft Foundation Classes(MFC)、Java Abstract Windowing Toolkit(AWT)和Swing库、以及基于X Window系统的Xt和Motif工具箱都提供相关的机制，可以注册特定事件的处理器。每种工具箱内部都包含一个从底层GUI系统检索事件的事件循环，和一个将事件分发到注册处理器的事件泵(event pump)。例如，下面的方法，当系统的事件管理器在接收到相应的事件后，会被自动调用。②

```
public class TabResults extends TabSpawnable
implements Runnable, MouseListener, ActionListener, ListSelectionListener {
[...]
////////////////////////////////////////////////////////////////
// MouseListener implementation

public void mousePressed(MouseEvent me) { }
public void mouseReleased(MouseEvent me) { }
public void mouseClicked(MouseEvent me) {
   if (me.getClickCount() >= 2) myDoubleClick(me.getSource());
}
```

面向对象的系统，常常通过围绕类来组织事件，向事件处理结构引入另一种有趣的曲折。基于类的事件处理给应用程序带来两个明显的优点。

(1) 事件可以以对象属性的形式承载附加的信息，消除了无类型、通用的事件参数，例如Microsoft Windows SDK中的事件参数就是无类型且通用的参数(图9.11:1)。

(2) 通过派生子类和继承，可以实现一个事件层次，事件层次又能够驱动事件分派策略。

下面的代码例举出这些概念的简单应用。③

```
public abstract class ArgoEvent extends EventObject
implements ArgoEventTypes {

    protected int _eventType = 0;

    public ArgoEvent(int eventType, Object src) {
        super(src);
        _eventType = eventType;
    }
```

上面的ArgoEvent类扩展了Java EventObject，ArgoEvent接下来又被ArgoModuleEvent④

① demogl/CPP/DemoGL/dgl_dllstartupdialog.cpp:86-554

② argouml/org/argouml/ui/TabResults.java:46-160

③ argouml/org/argouml/application/events/ArgoEvent.java:34-42

④ argouml/org/argouml/application/events/ArgoModuleEvent.java

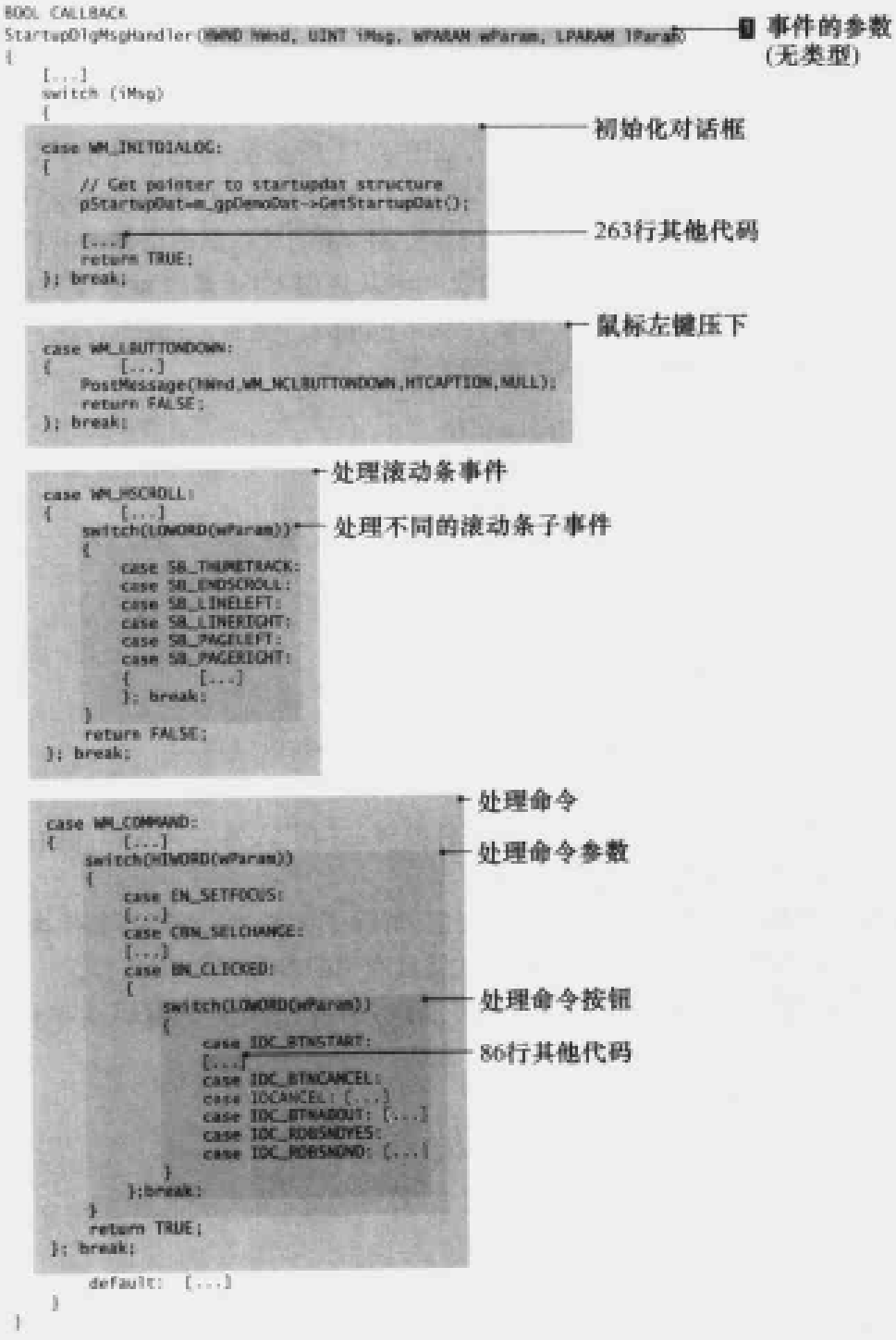


图 9.11 对 Microsoft Windows 消息的显式处理

和 ArgoNotationEvent[①] 类所扩展。

9.2.2 系统管理器

需要多个进程并发执行的系统，经常采用系统管理器控制模型(system manager control

① argouml/org/argouml/application/events/ArgoNotationEvent.java

model)。一个单独的系统组件起到集中式管理器的作用，负责启动、停止和协调其他系统进程和任务的执行。这是 BSD Unix 内核用来调度进程切换的模型。尽管正常情况下，进程都是根据它们动态调整的优先顺序，以循环的方式周期性地调度，但在某些极端情况下，这种调度方式并不高效。当一个过程完全不活动时、当系统内存行将耗尽时、或者内存池变得充满碎片时，系统会将进程的完整映像交换到磁盘上。这是进程调度器的任务，它是 Unix 系统的中心管理器(参见图 9.12)①。系统的内核，在执行所有的启动活动，比如调度事件驱动的周期性进程控制任务，之后，调用调度器，进入它的无限循环②。

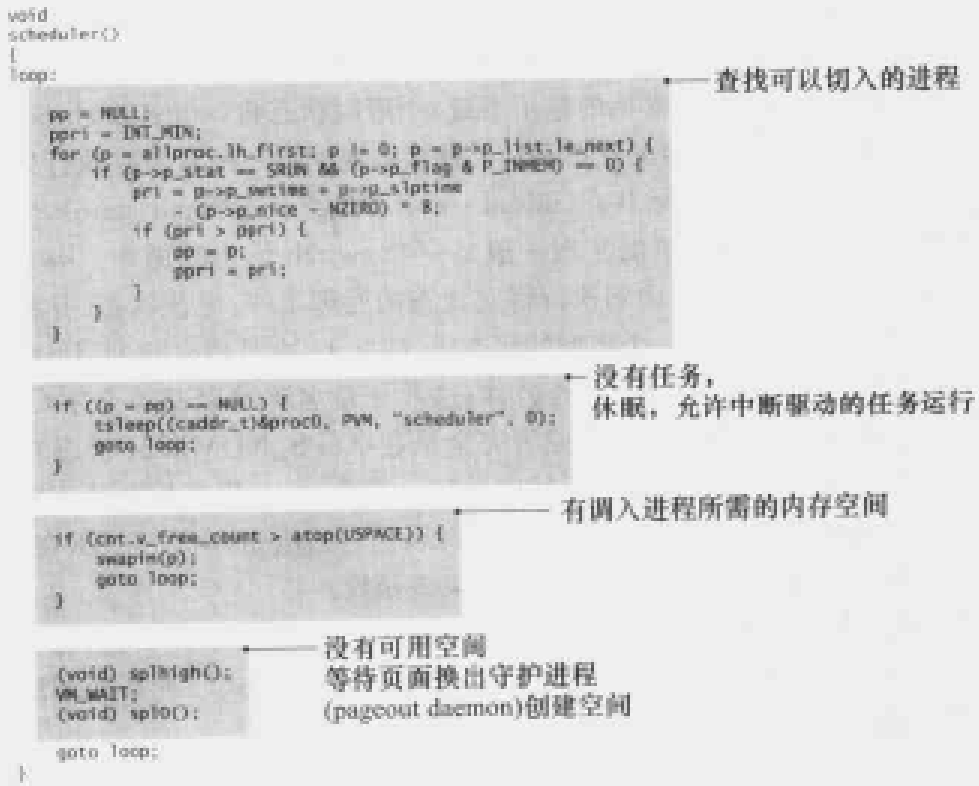


图 9.12 BSD Unix 的进程切换调度器

```
void
main (void * framep)
{
        [...]
        /* Kick off timeout driven events by calling first time. */
        roundrobin(NULL);
        schedcpu(NULL);
        [...]
        /* The scheduler is an infinite loop. */
        scheduler();
        /* NOTREACHED */
}
```

① netbsdsrc/sys/vm/vm_glue.c:350-419

② netbsdsrc/sys/kern/init_main.c:156-424

我们描述的系统实际上是两种控制模型的混合。一个事件驱动型系统负责协调正常的进程活动，同时我们上面分析的更为健壮的进程管理模型用作极端情形下的故障切换(fail-over)机制。许多现实的系统都会博采众家之长。当处理此类系统时，不要徒劳地寻找无所不包的构架图；应该将不同构架风格(Architectural style)作为独立但相关的实体来进行定位、识别并了解。

9.2.3 状态变迁

状态变迁模型(state transition model)通过操作数据——系统的状态，来管理系统的控制流程。状态数据决定执行控制权应该走向何处，对状态数据的更改可以重定向执行的目标。采用状态变迁控制模型的系统(更常见的是子系统)一般以状态机(state machine)的形式进行建模和构造。状态机由一组有穷的状态和从一种状态变到另一种状态时执行处理和变迁的规则来定义。两种特殊情况，初始状态(initial state)和最终状态(final state)，分别规定状态机的起始点和结束条件。状态机的实现一般是一个 switch 语句的循环。switch 语句的分支对应状态机的状态；每个 case 语句执行特定状态的处理工作、更改状态、并将控制权返回到状态机的顶端。图 9.13[①] 是一个典型的状态机，图 9.14 中是相应的 UML 状态变迁图。这个特殊的状态机用来识别外壳变量的名称；这些变量名称可以是单个字母(PS_VAR1 状态)，字母和数字(PS_IDENT 状态)，或者完全是数字(PS_NUMBER 状态)。PS_SAW_HASH 状态对以单个 # 开头的变量进行特殊处理。变量 state 用来保存状态机的状态。它从 PS_INITIAL(在 UML 图中用填充圆表示)开始，当它的值变为 PS_END(在 UML 图中表示为一个填充圆，外面被未填充的圆围绕)后结束该函数。

状态机一般用来实现简单的反应系统(simple reactive system)(用户界面、恒温器和电动机控制、以及处理自动化应用程序)、虚拟机、解释器、正则表达式模式匹配器和词法分析器。状态变迁图常常有助于理清状态机的运作。

练习 9.9 说说如何能够将图 9.11 说明的 Microsoft Windows SDK API 消息处理过程，组织成使用面向对象的结构。

练习 9.10 分析本书配套盘对自动电源管理(automatic power managment，APM)事件的处理。

练习 9.11 找出基于线程的应用程序，并说明它们使用的控制模型。

练习 9.12 本书配套盘中的代码一般如何实现状态变迁控制模型？当系统变得更为复杂时，怎样组织这类模型的代码，才能依旧使其易于阅读和理解呢？

① netbsdsrc/bin/ksh/lex.c:1124-1194

```
static char *
get_brace_var(XString *wsp, char *wp)
{
    enum parse_state {
            PS_INITIAL, PS_SAW_HASH, PS_IDENT,
            PS_NUMBER, PS_VAR1, PS_END
          }
          state;
    char c;

    state = PS_INITIAL;
    while (1) {
        c = getsc();
        /* State machine to figure out where the variable part ends. */
        switch (state) {
          case PS_INITIAL:
            [...]
          case PS_SAW_HASH:
            if (letter(c))
                state = PS_IDENT;
            else if (digit(c))
                state = PS_NUMBER;
            else if (ctype(c, C_VAR1))
                state = PS_VAR1;
            else
                state = PS_END;
            break; [...]
          case PS_VAR1:
            state = PS_END;
            break;
        }
        if (state == PS_END) {
            [...]
            break;
        }
    [...] }
    return wp;
}
```

状态

状态变量

开始状态

状态变迁规则

最终状态

到达最终状态时结束

图 9.13 一个状态机的代码

9.3 元素封装

自底向上的结构，和自顶向下构架结构相似，都涉及到系统中单个元素的包装方式。在处理大量的代码时，了解将代码分解成单独单元的机制极为重要。

9.3.1 模块

最常见的分解单元可能就是模块(module)：一个独立命名并寻址的构件，为其他模块提供服务。虽然我们的定义涵盖了许多种不同类型的分解单元，比如对象和过滤器，但要注意，我们将单独说明这些更为特殊的实体，我们仅在不适用其他任何情况时才会使用术语模块(module)。模块一般通过过程调用和数据共享与其他模块进行交互。大多数情况下，模块的物理边界是单个文件、组织到一个目录中的多个文件或拥有统一前缀的文件的集合。另外，在允许嵌套定义过程的语言中(例如 Pascal)，模块一般实现为全局可见的过程。尽管使用 C++中的 namespace 功能或 Modula module 抽象可以用类似的方法将多个模块放到一个单一文件中，但实践中很少这样做。然而，这些功能，如 9.3.2 节所述，经常用来将多个文件中实现的功能归并到单个模块中。

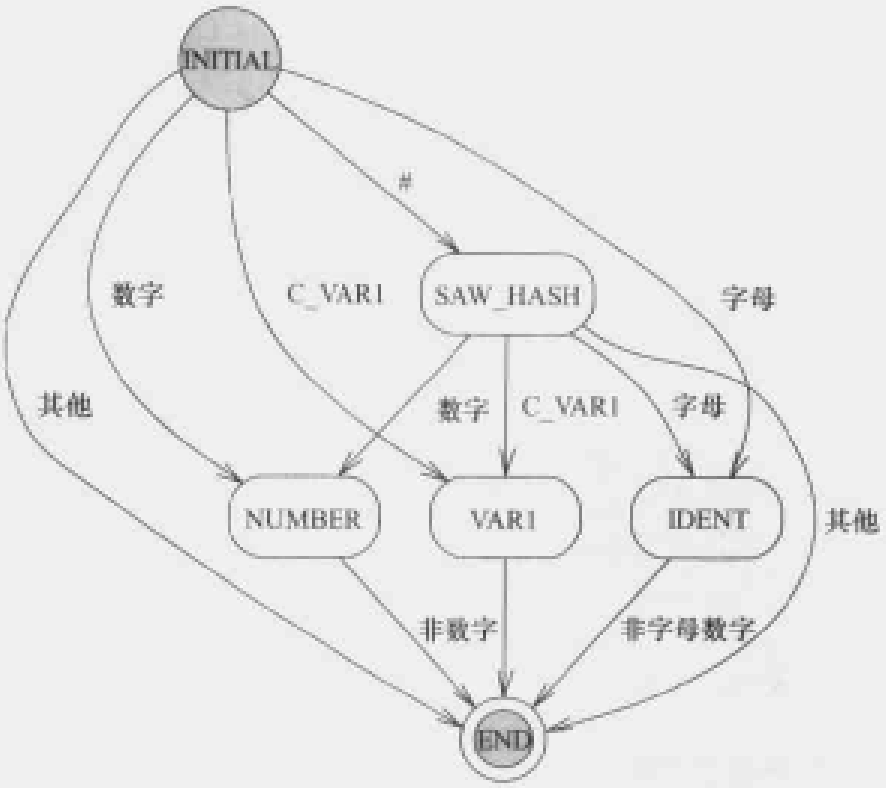


图 9.14 状态机的 UML 状态变迁图

一个依照文件/模块等同的概念进行组织的典型系统，是 BSD nvi 编辑器[①]。组成该程序的 113 个文件中，每个文件都定义了一个或多个公开函数或服务于某些私有函数(static)的数据元素(图 9.15)[②]。

BSD 的 window 包[③](一个基于文本的窗口界面)是一个使用前缀文件命名约定系统，它用文件组构造模块。一些较明显的模块和对应的文件名如下：

- 键盘命令处理：cmd.c，cmd1.c，cmd2.c，cmd3.c，cmd4.c，cmd5.c，cmd6.c，cmd7.c。
- 命令行处理：lcmd.c，lcmd.h，lcmd1.c，lcmd2.c。
- 语法分析：parser.h，parser1.c，parser2.c，parser3.c，parser4.c，parser5.c。
- 终端的驱动程序：tt.h，ttf100.c，ttgeneric.c，tth19.c，tth29.c，ttinit.c，ttoutput.c，tttermcap.c，tttvi925.c，ttwyse60.c，ttwyse75.c，ttzapple.c，ttzentec.c。
- 窗口系统：ww.h，wwadd.c，wwalloc.c，wwbox.c，wwchild.c，wwclose.c，wwclreol.c，wwclreos.c，wwcursor.c，wwdata.c，wwdelchar.c，wwdelete.c，wwdelline.c，wwdump.c，wwend.c，wwenviron.c，wwerror.c，wwflush.c，wwframe.c，wwgets.c，wwinit.c，wwinschar.c，wwinsline.c，wwiomux.c，wwlabel.c，wwmisc.c，wwmove.c，wwopen.c，wwprintf.c，wwpty.c，wwputc.c，wwputs.c，wwredraw.c，wwredrawwin.c，wwrint.c，wwscroll.c，wwsize.c，wwspawn.c，wwsuspend.c，

① netbsdsrc/usr.bin/vi

② netbsdsrc/usr.bin/vi/vi/v_increment.c:31-267

③ netbsdsrc/usr.bin/window

图 9.15　nvi 编辑器的一个模块

wwterminfo. c，wwtty. c，wwunframe. c，wwupdate. c，wwwrite. c。

从而，系统的 75 个文件归组到 5 个不同的模块中。余下的 25 个文件很可能表示不同的模块，许多情况下由提供公开接口的头文件和提供对应实现的 C 文件组成一组(char. c/char. h，string. c/string. h，var. c/var. h，xx. c/xx. h)。需要在此说明的是，虽然窗口系统和终端驱动程序文件的名称是合理的，但用在键盘命令处理、命令行处理和语法分析文件名称中的命名约定尚有很多需要改进的地方。

表 9.1　NetBSD 内核目录和对应的模块

目录	模块
ddb	内核调试器
isofs/cd9660	ISO CD 文件系统
msdosfs	MS-DOS 文件系统
netatalk	AppleTalk 联网模块
netccitt	CCITT 联网模块
netinet	Internet 联网模块
nfs	网络文件系统
vm	虚拟内存支持
ufs/mfs	基于内存的文件系统
ufs/lfs	日志结构的文件系统
ufs/ffs	“快速”文件系统

最后，BSD 内核[①]是一个将模块归组到单独目录中的典型例子。表 9.1 列出了一些具有代表性的目录。但要注意，某些功能使用通用的文件前缀进行了进一步的细分。例如，netinet 目录中的文件使用 if_，in_，ip_，tcp_和 udp_前缀来表示网络协议栈中他们处理的相应部分。另外，虽然模块一般存储在单独的目录中，但反之并不成立：目录经常用来将不必构成更大整体的许多相关模块归组到一起。

9.3.2 命名空间

模块的一个重要思想就是信息隐藏(information hiding)的原则，它规定与模块相关的所有信息都应该为私有，除非它被特别声明为公开。在单个文件实现的 C 模块中，经常会看到全局标识符用 static 关键字声明，将它们的可见性限制在单个编译单元中(文件)。[②]

```
static int zlast=- 1;
static void islogin(void);
static void reexecute(struct command * );
```

然而，这项技术不能阻止用在头文件中的标识符泄露给包括它们的文件。例如，在 C 和 C++语言中，头文件中的 typedef 或预处理宏定义，如果和其他文件中定义的全局函数或变量同名，那么可能会导致冲突。尽管有些情况可以通过重新命名所开发的程序中相应的标识符来解决，但其他情况，比如两个不同的现有模块互相冲突，可能就难以解决，因为它们可能超出了开发者的控制范围。考虑编译下面的代码(精心设计的)。

```
# include "libz/zutil.h"
# include "libc/regex/utils.h"
```

这两个头文件中都定义了 uch 标识符[③④]，从而导致下面的错误。

```
libc/regex/utils.h:46: redefinition of 'uch'('uch'的重复定义)
libz/zutil.h:36: 'uch' previously declared here('uch'先前定义在此处)
```

命名空间污染(namespace polution)的问题，在 C 语言中有许多特殊的方式可以解决；其他语言，如 Ada，C++，Eiffel，Java，Perl 和 Modula 系统提供特殊的构造来克服这个问题。在不利用语言提供的额外功能的情况下，抑制命名空间污染的一个通用解决方案，是在标识符前加上特定的惟一前缀。请注意，下面的示例中，rnd.h 头文件中所有的类型、函数和宏标识符都加上了一个 rnd 前缀。[⑤]

```
void rnd_attach_source(rndsource_element_t* , char* , u_int32_t);
void rnd_detach_source(rndsource_element_t * );
[...]
# define RND_MAXSTATCOUNT 10 /*  10 sources at once max * /
```

① netbsdsrc/sys
② netbsdsrc/bin/csh/func.c:64-66
③ netbsdsrc/lib/libz/zutil.h
④ netbsdsrc/lib/libc/regex/utils.h
⑤ netbsdsrc/sys/sys/rnd.h:134-178

```
[...]
typedef struct {
    u_int32_t start;
    u_int32_t count;
    rndsource_t source[RND_MAXSTATCOUNT];
} rndstat_t;
```

实际上，由于 ANSI C 中将所有以下划线开头的标识符保留给语言的实现使用，故而，用前缀进行标识符隔离的方法已经得到 ANSI C 标准的正式认可。在阅读库的头文件时，会注意到所有标识符都以下划线开头，从而与用户可能定义的标识符区别开来①。

```
struct __sbuf {
    unsigned char * _base;
    int _size;
};
```

虽然上面示例中的_base 和_size 结构标记属于——依据 ANSI C——一个独立的命名空间（__sbuf 结构标记的命名空间），但它们依旧需要加上下划线前缀，这是为了避免与宏定义冲突。

现代 C++程序经常使用 namespace 功能，来避免我们前述的这些问题。将逻辑上同属一组的构成模块的程序实体，定义或声明在 namespace 控制块中（图 9.16:2）②③。之后，其他作用域就可以通过下面两种方式使用这些标识符了：在标识符名之前明确地加上正确的命名空间（图 9.16:1）；使用 using 指令输入命名空间中声明的所有标识符（图 9.16:3）。第二种方法输入的命名空间标识符可以直接使用，不需要任何前缀（图 9.16:4）。

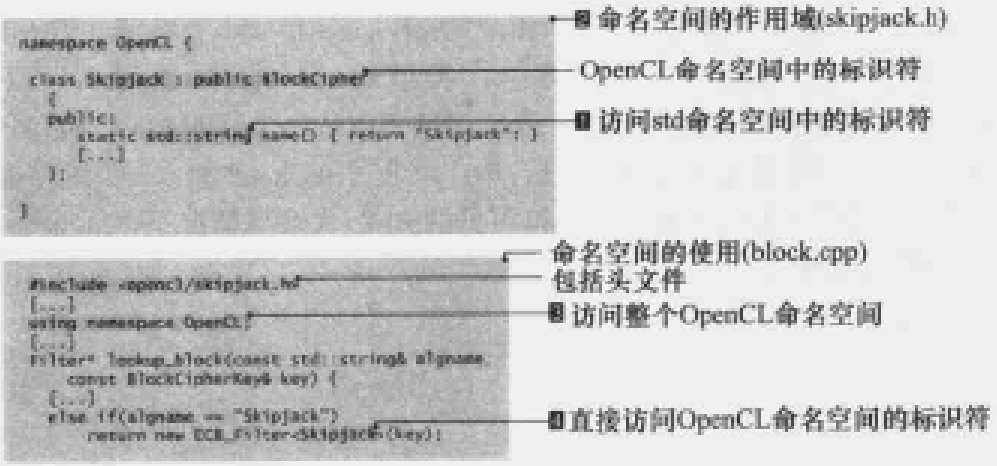


图 9.16　C++中命名空间的定义和使用

Java 通过 package 关键字提供一种类似的解决方案，来解决命名空间污染的问题。如

① netbsdsrc/include/stdio.h:82-85
② OpenCL/include/skipjack.h:11-34
③ OpenCL/checks/block.cpp:36-139

图 9.17 所示①②，package 关键字用来说明，其后的代码定义的类和接口属于命名包(named package)，并能够访问包内的所有类型。相应地，import 语句用来输入给定包中的类，使得程序中可以使用这些类的简写名称(abbreviated name)。由于 Java 中完全限定的类名(fully qualified class name)总是可以访问的，因此，Java 社团已经采纳了下面约定，即：所有的包名都要用 Internet 域名作为前缀加以限定，或使用包生产者的公司名称③④⑤⑥。

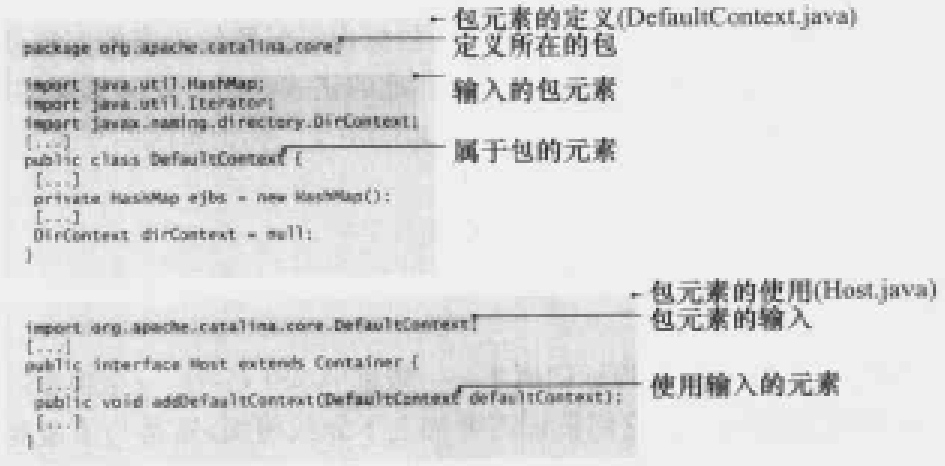


图 9.17 Java 中包的定义与使用

```
package org.argouml.cognitive.critics;
package org.hsqldb;
package org.apache.cocoon.components.sax;
import ru.novosoft.uml.foundation.core.* ;
```

Ada 中类似的功能十分相似，它基于相应的 package 和 use 关键字。惟一的差异是，Ada 包必须使用 with 关键字显式地输入。类似地，Modula-3 使用 INTERFACE 和 MODULE 声明来定义接口和实现所属的命名空间，FROM ... IMPORT 和 IMPORT ... AS 语句将名称引入到本地上下文中。最后，Perl 也通过 package 和 use 关键字提供命名空间功能。但是，Perl 程序员可以配置标识符可见的方式；大多数情况下，Exporter 模块用来指定输出标识符的清单⑦。

```
package CPAN;
[...]
use Config ();
use Cwd ();
use DirHandle;
```

① jt4/catalina/src/share/org/apache/catalina/core/DefaultContext.java:65-1241
② jt4/catalina/src/share/org/apache/catalina/Host.java
③ argouml/org/argouml/cognitive/critics/Critic.java:29
④ hsqldb/src/org/hsqldb/WebServer.java:36
⑤ cocoon/src/java/org/apache/cocoon/components/sax/XMLByteStreamCompiler.java:8
⑥ argouml/org/argouml/ui/ActionGoToEdit.java:33
⑦ perl/lib/CPAN.pm:2-66

```
use Exporter ();
[...]
@ EXPORT = qw(
            autobundle bundle expand force get cvs_import
            install make readme recompile shell test clean
            );
```

9.3.3　对象

对象(object)是一个运行期间的实体,它为自身所封装的数据提供一种访问机制。在完全基于对象的系统中,计算都是通过对象发送和接收消息(message)或调用其他对象的方法来完成的。相似的对象,它们的行为被抽象成类的形式,相关的类经常组织成继承层次。

在基于对象的语言中,我们可以通过语言中相应关键字的使用,识别出对象类型的定义:C++,C#,Eiffel 和 Java 中是 class,Modula-3 中是 object,Ada-95 中是 package,Perl 中是 bless。另外,与集成开发环境相关联的面向对象和基于对象的语言,如 Smalltalk 和 Visual Basic,都通过内建对象或类浏览器(class brower),支持对象的定义和检查。最后,甚至在并不支持显式创建对象的语言中,通过将数据元素,以及用来处理这些数据元素的函数集合到一个结构中,也可以定义对象。

1. 字段和方法

图 9.18 中是一个具有代表性的 C++类定义①,图 9.19 是一个 Java 类的定义②。每个类都定义了一个名称,之后,这个名称将用来声明、定义和创建这个类的对象。参与到继承层次中的类还会声明它们的父类。Java 只允许类继承自单一的类,但类可以使用 implements 关键字来表示它会实现多个类的功能。类一般会封装许多成员变量(member variable)或字段(field),或属性(attribute)——它们存储与对象相关的信息,还有一些成员函数或方法——它们将作用于对象的字段。每个成员函数被调用时都会结合一个具体的对象③。

```
hash_input.final(input_hash);
```

因此,每个成员函数都能够直接访问具体对象的字段,以及该对象的其他相关方法。下面的方法访问对象的 crc 字段(定义在图 9.18 中),并调用对象的 clear()方法。

```
void CRC32::final(byte output[HASHLENGTH])
   {
   crc ^= 0xFFFFFFFF;
   for (u32bit j = 0; j != HASHLENGTH; j++ )
      output[j] = get_byte(j, crc);
   clear();
   }
```

C++和 Java 都提供 this 关键字,它可以显式地寻址方法的底层对象。有时为了明确起

① OpenCL/include/crc32.h:16-31

② cocoon/src/java/org/apache/cocoon/util/PostInputStream.java:20-255

③ OpenCL/checks/block.cpp:121

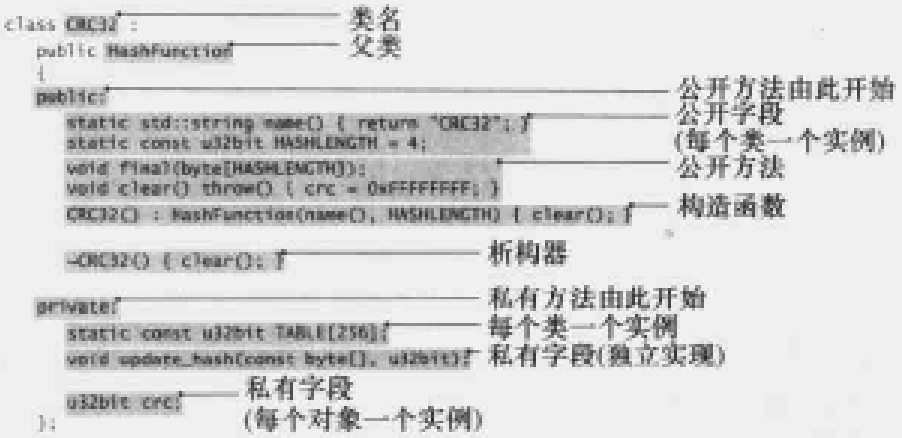


图 9.18 一个 C++类的声明(用于计算 CRC)

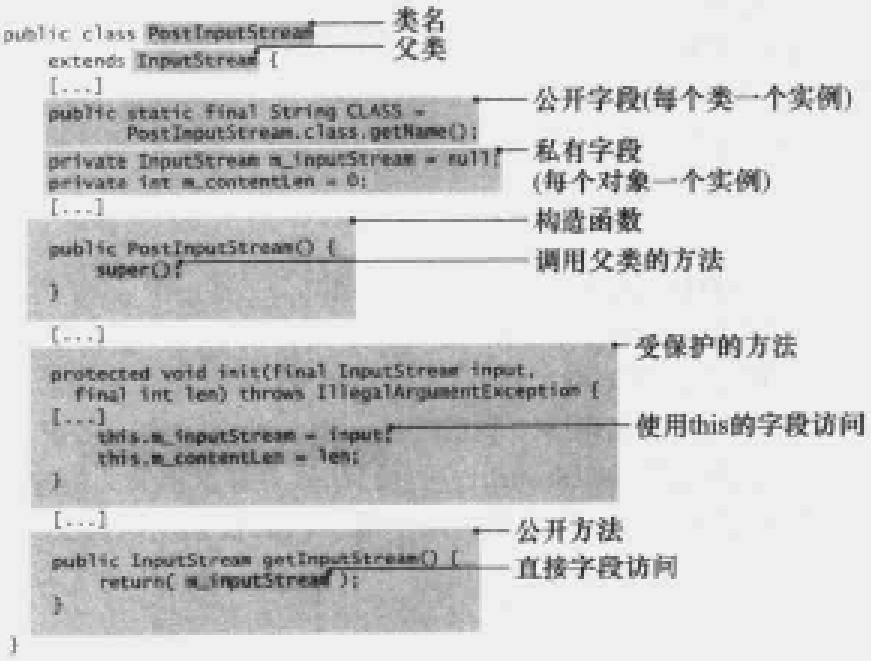


图 9.19 Java 类:HTTP post 操作的输入流

见,或者消除命名冲突的歧义,需要用到这项功能。

2. 构造函数和析构函数

由于对象的创建与消失方式多种多样(作为全局或局部变量、作为动态创建的数据结构的元素、或者作为临时实体),我们可以定义一个特殊函数(称为构造函数),只要创建对象,该函数就会被调用,和对应的析构函数(destructor)或终结器(finalizer),在对象销毁之前调用。在 C++和 Java 中,构造函数定义为与类同名的方法;C++析构函数还是使用类名称,前面加上否定字符(~)前缀;相应的 Java 方法则简单地命名为 finalize。对象构造函数经常用来分配与对象相关的资源,并初始化对象的状态。析构函数一般用来释放对象在生命期中占用的资源。在下面的示例中,在对象构造时,将一个事件侦听器关联到该对象,在对象中

止并退出时移除。①

```
public abstract class FigEdgeModelElement[...][...]

  public FigEdgeModelElement() {
    _name = new FigText(10, 30, 90, 20);
    _name.setFont(LABEL_FONT);
    [...]
    ArgoEventPump.addListener(ArgoEvent.ANY_NOTATION_EVENT, this);
  } [...]

  public void finalize() {
    ArgoEventPump.removeListener(ArgoEvent.ANY_NOTATION_EVENT, this);
  } [...]
}
```

在 C++语言中，使用 new 运算符为对象分配的内存，必须显式地使用 delete 运算符加以释放。(Java 自动执行这种垃圾回收任务。)因此，C++析构函数还用来释放在对象的生命期中分配的内存②。

```
isockunix::isockunix (const sockunixbuf& sb)
  : ios (new sockunixbuf (sb))
{}

isockunix::~ isockunix ()
{
  delete ios::rdbuf ();
}
```

在 C++程序中，在对象的构造函数中用 new 分配的内存，如果与对象的析构函数中用 delete 处理的内存不对称，可能是内存泄漏的一种征兆，必须加以改正。另外，在 C++中，由于系统提供的对象复制机制会导致两个对象共享同一动态分配的内存块，这就有可能导致重复删除，为了恰当地处理这种情况，必须提供一个复制构造函数(copy constructor)(以对自身对象的引用为惟一参数的构造函数)。一般地，这会涉及到共享对象或跟踪引用计数(浅表副本，shallow copy)，或者执行对象内容的完全(深度)复制。

3. 对象和类成员

有时，数据或处理过程与整个类有关，而非个别的对象。在 C++和 Java 中，这种类方法或类字段通过在它们前面加上 static 关键字来表示。请注意，类方法不能访问对象方法和对象字段，因为它们是结合一个特殊的对象加以调用的。然而，对象方法总是可以访问类方法和类字段。对象方法经常使用类字段来存储控制所有方法运作的数据(比如查找表或字典)或维护类运作的状态信息(例如，赋给每个对象一个惟一标识符的计数器)。下面的例子

① argouml/org/argouml/uml/diagram/ui/FigEdgeModelElement.java:63-458

② socket/sockunix.C:75-82

说明了这种情况。①②

```
class Test_Any {
public: [...]
  int reset_parameters (void); [...]
private:[...]
  static size_t counter;[...]
};
int
Test_Any::reset_parameters (void)
{[...]
 CORBA::ULong index = (counter++ % Test_Any::ANY_LAST_TEST_ITEM);
```

4. 可见性

类的定义，除了为对象的数据设立容器，并指定处理这些数据的共享方法之外，还是一种信息隐藏机制。成员(字段和方法)可以声明为 private——表示它们只能从它们所在的类自身的方法中访问，public——表示程序中任何地方都可以访问它们(通过对象或作为类方法的全局函数)，protected——将它们的可见性限定在它们所在的类以及从其派生的子类。在设计良好的类中，所有的字段都应声明为 private，并用公开的访问方法(access method)提供对它们的访问。这些方法，经常命名为属性名附加相应的 get 或 set 前缀或后缀，应用很普遍，以至于 C# 和 Visual Basic 等语言包含对它们的内在支持。通过用访问方法将字段隐藏起来，允许功能的接口(以访问方法的形式提供)与相应的实现隔离开来。这种隔离使得对功能做出修正或访问特定的字段都比较容易。下面的例子中包含对私有字段 bAutoCommit 的访问方法③。

```
public class jdbcConnection implements Connection {[...]
    private boolean bAutoCommit;[...]
    public void setAutoCommit(boolean autoCommit) throws SQLException {
        bAutoCommit = autoCommit;
        execute("SET AUTOCOMMIT " + (bAutoCommit ? "TRUE" : "FALSE"));
    }[...]
    public boolean getAutoCommit() {
        if (Trace.TRACE) {
            Trace.trace();
        }
        return bAutoCommit;
    }[...]
}
```

访问方法通常还用来限制对字段的访问类型。下面的代码提供一个 get 方法，但没有

① ace/TAO/tests/Param_Test/any.h:27-97

② ace/TAO/tests/Param_Test/any.cpp:115-119

③ hsqldb/src/org/hsqldb/jdbcConnection.java:76-997

相应的 set 方法，实际上创建了一个只读字段。①

```
public class jdbcDatabaseMetaData implements DatabaseMetaData {
    private jdbcConnection cConnection; [...]
    public Connection getConnection() {
    if (Trace.TRACE) {
        Trace.trace();
        return cConnection;
    }
}
```

最后，在 C++中，私有方法有时会与运算符重载结合起来使用，用来完全禁止将某些运算符应用到特定的对象身上。下面的示例阻止在底层对象上使用赋值运算符。②

```
private:
  // = Disallow these operations.
  ACE_UNIMPLEMENTED_FUNC (void operator = (const ACE_Future_Set< T > &))
```

有时，函数、单个方法或整个类的所有方法可能需要访问另一个类的私有成员。在C++中，当看到函数、方法或类用 friend 关键字声明在其他的类中时，这将为前者提供对后者私有成员的完全访问。在下面的示例中，main 可以访问 chatters，以及 Chatter 类的所有私有成员。③

```
class Chatter : public QListBoxItem {[...]
    private:
        static ChatterRef * chatters[MAX_NUM_CHATTERS];
        [...]
        friend int main(int, char * argv[]); // to destroy chatters[]
};
```

在遇到 friend 声明时，要停下来分析一下，看看绕过类封装在设计上的理由。

5. 多态

隶属于继承层次的对象，所具有的一项重要特性是，派生类的对象可以用作父类的对象。有时，父类仅仅提供期望派生类实现的公共功能和接口；而不会创建父类的对象。这种父类，在 Java 中使用 abstract 关键字，在 C++中用=0 来定义，将期望子类实现的方法声明为抽象类型。程序中每次调用基类(在 C++中，也可以是对象指针、引用)的方法时，系统都会自动地将调用分派给恰当的派生类方法。从而，在运行期间，同一个方法调用会依底层对

① hsqldb/src/org/hsqldb/jdbcDatabaseMetaData.java:61-2970

② ace/ace/Future_Set.h:81-83

③ qtchat/src/0.9.7/core/chatter.h:36-124

象的类型，产生对不同方法的调用。考虑图 9.20 中给出的例子①②③④⑤。所有的算法（Algorithm 的子类）都要提供一个 clear 方法，实现散列功能（一种将大的信息压缩成具有代表性的签名的方法）的所有类都要提供一个 final 方法。这些方法在父类中都声明为 virtual，表示它们将由派生类提供。在类 MD2 和类 CRC24 的声明中（二者均为 HashFunction 的子类），可以看到各自的声明，以及 clear 函数的一个实现。函数 derive 使用 OpenPGP S2K 算法创建一个密钥，它根据 OpenPGP_S2K 类的 hash 成员的不同变换它的功能。可以将 OpenPGP_S2K 的 hash 成员设为指向任何属于 HashFunction 子类的对象。当代码中调用 clear 和 final 方法时，调用将会动态地分派到恰当的类方法。

```
class Algorithm {
   public:
      virtual void clear() throw() = 0;
   [...] };

class HashFunction : public Algorithm {
   public:
      virtual void final(byte[]) = 0;
   [...] };

class MD2 : public HashFunction {
   public:
      void final(byte[HASHLENGTH]);
      void clear() throw();
   [...] };

class CRC24 : public HashFunction {
   public:
      void final(byte[HASHLENGTH]);
      void clear() throw() { crc = 0xB704CE; }
   [...] private: [...]
      u32bit crc;
   };

class OpenPGP_S2K : public S2K {
   [...] private:
      HashFunction* hash;
   };

SymmetricKey OpenPGP_S2K::derive(const std::string& pw, u32bit keylen) const
   { [...]
   hash->clear();
   while(keylen > generated)
      {
      [...]
      hash->final(hash_buf);
      }
   return SymmetricKey(key, key.size());
   }
```

需要由所有的Algorithm派生类实现的抽象方法

满足Algorithm接口

需要由所有的HashFunction派生类实现的抽象方法

满足HashFunction接口

类对方法的具体实现

满足HashFunction接口

类对方法的具体实现

指向HashFunction对象的指针，可以指向MD2、CRC24及其他对象

调用派生类中相应的函数

调用派生类中相应的函数

图 9.20 C++中运行期间的多态性

6. 运算符重载

C++语言允许程序员通过运算符重载方法（operator overloading method），重新定义语言中标准运算符的含义。缺乏经验的程序员在编写代码时，往往会以违背人们预期的方式使用运算符，这是对这项特性的误用。实际上，我们可以有节制地用运算符增强特定类的可

① OpenCL/include/opencl.h:19-105
② OpenCL/include/md2.h:13-28
③ OpenCL/include/crc24.h:16-29
④ OpenCL/include/pgp_s2k.h:13-22
⑤ OpenCL/src/pgp_s2k.cpp:13-59

用性，但用运算符重载，将类实现为拥有内建算术类型相关的全部功能的类数字实体，是不恰当的。重载的运算符用方法或 friend 函数来定义，其形式为 operator 关键字后跟实际的运算符符号或数据类型。用 friend 函数实现的运算符函数，将所有的操作数都作为参数接收，而用方法实现的运算符函数将方法的底层对象作为第一操作数。

有选择地重载特定运算符的最普遍原因如下：

- 进一步重载移位运算符(<< 和>>)，从而允许对新的数据类型执行格式化输入/输出。①

```
ostream& operator << (ostream& 0o, smtp& s)
{
  char buf[1024];
  int cont = 1;
  while (cont) {
    cont = s.get_response (buf, 1024);
    o << buf << endl;
  }
  return o;
}
```

- 提供比较运算符，从而需要可比较对象的库函数能够工作。②

```
ACE_INLINE int
ACE_Addr::operator == (const ACE_Addr &sap) const
{
  return (sap.addr_type_ == this-> addr_type_ &&
  sap.addr_size_ == this-> addr_size_ );
}
```

- 通过重载赋值和取对象(dereference)运算符，使智能的类指针对象的行为更为直观。③

```
* @ brief A smart pointer stored in the in- memory object database
* ACE_ODB. The pointee (if any) is deleted when reassigned.
*/
class ACE_Export ACE_Dumpable_Ptr
{
public:
  ACE_Dumpable_Ptr (const ACE_Dumpable * dumper = 0);
  const ACE_Dumpable * operator-> () const;
  void operator = (const ACE_Dumpable * dumper) const;
private:
  /// "Real" pointer to the underlying abstract base class
```

① socket/smtp. C:156-165

② ace/ace/Addr. i:19-24

③ ace/ace/Dump. h:90-104

```
    /// pointer that does the real work.
    const ACE_Dumpable * dumper_;
};
```

- 为字符串提供在符号处理语言(symbol-processing language),如 Basic,awk 和 Perl,中支持的赋值和拼接运算符。①

```
// concatenation.
const CStdString& operator+= (const CStdString& str);
```

当用运算符重载创建新的算术类型时(比如复数、分数、大整数、矩阵或多倍精度浮点数),设计者的目标是支持所有的算术运算符。图 9.21② 例举出一类对象的声明,它们支持无限宽度的整数运算,从不会溢出。请注意,其中大量的声明用来支持合理的运算和转换。

7. Perl 中的类

如图 9.22 所示③,Perl 中大多数面向对象的操作,都是基于对命令式语法的细微扩展。Perl 中,可以借助于 bless 关键字,声明一个给定的变量(标量、数组、散列、于过程)为属于某个类的对象,与 C++和 Java 相比,Perl 靠提供这种方式来支持对象和类。引用经常赋予一个名为 self 的变量,该变量的使用类似于 C++和 Java 的 this 伪变量(pseudo-variable)。封装是通过使用现有的 package 机制来完成,而方法就是子例程,它的参数为调用对象的引用。构造函数是一个包子例程,一般命名为 new,它返回一个"保佑(blessed)"属于给定类的数据元素。此后,这个"受到保佑(blessed)"的数据元素就可以用来调用"保佑(bless)"它的类的方法④。

```
my $dh = new DirHandle ($target);
if (defined $dh) {
    foreach my $entry ($dh->read()) {
```

析构函数是一个名为 DESTORY 的子例程。Perl 中不能显式地创建对象字段。字段一般实现为"受到保佑"的散列。因而,下面的例子⑤中,通过在 load 构造函数中"保佑"包含变量 Handle,File 和 Queue 的散列,将它们用作成员变量,并在 find 方法中通过对象引用访问它们。

```
sub load
{
  [...]
  bless {Handle => $handle, File => $file, Queue => 'SESSION' },
    "OS2::DLL::$file";
}
sub find
```

① dernogl/Include/Misc/StdString.h:363-364
② purenum/integer.h:79-199
③ perl/lib/DirHandle.pm:1-64
④ ace/bin/pippen.pl:157-160
⑤ perl/os2/OS2/REXX/DLL/DLL.pm:47-48

图 9.21 C++中的运算符重载

```
{
    my $self   = shift;
    my $file   = $self->{File};
    my $handle = $self->{Handle};
}
```

Perl 中对继承的支持机制也十分简单。类通过在 ISA 数组中设定相应的类名(它从这些类名所标示的类继承方法),表示自身参与到继承层次中①。

```
package IO::Socket;
[...]
@ISA = qw(IO::Handle);
```

① perl/ext/IO/lib/IO/Socket.pm:7-24

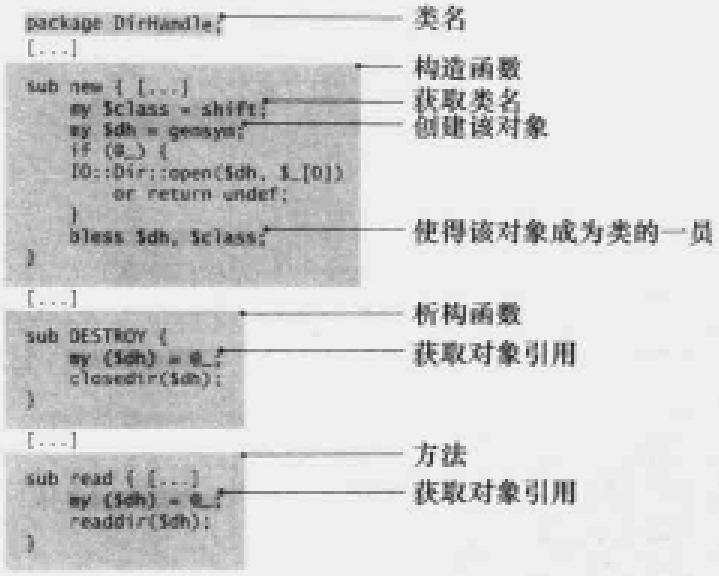


图 9.22 Perl 目录访问类的声明

上面的代码仅仅是表示，如果调用的方法在 IO::Socket 中没有定义，则调用 IO::Handle 中对应的方法。请注意，和 C++，Java 相比，此处没有字段和构造函数的隐式继承。

8. C 中的类

在前面我们曾说过，即使在不直接支持对象的语言中，也可以定义和使用对象。在 C 代码中，最常见的实现方式是用结构来代表每个对象，其中对象的字段就是结构的成员，对象的方法作为函数的指针，如下面的代码所示。①②

```
typedef struct {
    char *name;                     /*  name of format */
    int bsz;                        /*  default block size. */
    int hsz;                        /*  Header size in bytes. */
    int hlk;                        /*  does archive store hard links info? */
    int (*id)(char *, int)          /*  checks if a buffer is a valid header */
    int (*st_rd)(void)              /*  initialize routine for read. so format */
    int (*rd)(ARCHD *, char *)      /*  read header routine. passed a pointer to */
    [...]
    int (*options)(void)            /*  process format specific options (-o) */
} FSUB;

FSUB fsub[] = {
/*  0 : OLD BINARY CPIO */
     { "bcpio", 5120, sizeof(HD_BCPIO), 1, 0, 0, 1, bcpio_id, cpio_strd,
     bcpio_rd, bcpio_endrd, cpio_stwr, bcpio_wr, cpio_endwr, NULL,
     cpio_subtrail, rd_wrfile, wr_rdfile, bad_opt },
[...]
```

① netbsdsrc/bin/pax/pax.h,140-218

② netbsdsrc/bin/pax/options.c,99-129

```
/*  5 : POSIX USTAR */
    { "ustar", 10240, BLKMULT, 0, 1, BLKMULT, 0, ustar_id, ustar_strd,
    ustar_rd, tar_endrd, ustar_stwr, ustar_wr, tar_endwr, tar_trail,
    NULL, rd_wrfile, wr_rdfile, bad_opt }
};
```

要注意，在上面的示例中，对象承载了所有方法指针的完整副本，这些方法与底层对象的数据没有直接的关系。更为复杂的方案是，将这些函数指针替换为一个单独的、每个类对应一个的、方法指针的结构，并且采用特殊的约定向每个方法传递对象的指针。图 9.23 中所示的代码即为如此。①②③④⑤ 实际上，这段 C 代码显式地实现了许多面向对象的语言才有的特性。与每个对象相关联的 v_op 指针(图 9.23:1)指向一个共享表，其中是指向相应方法的指针(图 9.23:3)。大多数基于对象的软件系统中，编译器都内部实现了该指针，术语称为虚函数表(virtual function table 或 vtbl)。同样，指向对象的指针在 VOP_RECLAIM 中传递给每个方法，并在方法中读取出来，在基于对象的系统中，它也是隐式地传递给对象的方法，并可以在方法中通过 this 伪变量进行访问。尽管这些实现努力可能有些极端，但在其他的系统，比如 X Window 系统的 Xt 工具箱中，确实用到了类似的方案⑥。但是，希望您能够认同，面向对象的语言为相同的功能提供了更易于理解的方案。

9.3.4 泛型实现

数据结构或算法的泛型实现(generic implementation)的设计目的是能够在任意数据类型上工作。泛型实现不是在编译期间通过宏替换或语言所支持的功能(比如 C++模板和 Ada 的泛型包)来实现，就是在运行期间通过使用数据元素的指针和函数的指针、或对象的多态性来实现。我们将参照一个简单的函数，对这些可能性进行展开描述，该函数返回两个元素中的最小值。在 C 和C++程序中，经常会用预处理器宏来实现简单的泛型函数⑦。

```
# define ace_min(a,b)     (((b) > (a)) ? (a) : (b))
```

不管给它指定的实际参数是什么，ace_min 宏的定义体会进行字面扩展。这种字面扩展使得 min 函数适用于能够使用>运算符进行比较的任何参数。由于它消除了函数调用的开销，所以它还很高效。另外，它的两个参数可能会被两次求值；如果任何参数的求值有相关的副作用(例如，传递给 min 的参数应用了后增量运算符(postincrement))，那么这种副作用(对程序员来说也许很意外)将会发生两次。另外，含有嵌套宏扩展的代码的大小和复杂性可能会快速地扩大到不合理的程度。下面的例子展示一个简单的表达式⑧：

---

① netbsdsrc/sys/sys/vnode.h:71-341
② netbsdsrc/sys/kern/vnode_if.src:208-210
③ netbsdsrc/sys/kern/vnode_if.c:467-481
④ netbsdsrc/sys/sys/vnode_if.h:663-676
⑤ netbsdsrc/sys/ufs/ffs/ffs_vnops.c:69-257
⑥ XFree86-3.3/xc/lib/Xt
⑦ ace/ace/Min_Max.h:62
⑧ ace/ace/OS.cpp:3089-3090

图 9.23 C 语言中实现的含共享方法表的对象类

```
ACE_MIN (ACE_THR_PRI_FIFO_MAX,
         ACE_MAX (ACE_THR_PRI_FIFO_MIN, priority))
```

和它扩展之后的产物：

```
((((((((ACE_THR_PRI_FIFO_MIN)) > ((priority))) ?
((ACE_THR_PRI_FIFO_MIN)) : ((priority)))))) >
((ACE_THR_PRI_FIFO_MAX))) ? ((ACE_THR_PRI_FIFO_MAX)) :
((((((ACE_THR_PRI_FIFO_MIN)) > ((priority))) ?
((ACE_THR_PRI_FIFO_MIN)) : ((priority))))))
```

由于编译器一般在使用宏的地方报告生成错误代码的宏定义，而不是在定义宏的地方，所以宏错误难以跟踪与改正。通过C预处理器运行该程序，经常是调试宏错误时采取的一种好的策略。

在C++程序中，泛型实现的代码通过template机制来表达。①

```
template < class T>
inline const T &
ace_min (const T &t1, const T &t2)
{
  return t2 > t1 ? t1 : t2;
}
```

上面的示例中，ace_min函数以两个未定类型T的const引用为参数，返回其中一个元素作为结果。这个函数的具体实例在程序被编译时被实例化，以满足对该函数的具体应用。因此，由于代码中priority变量的类型是long，所以编译器会自动地生成下面的内部函数。

```
inline const long &
ace_min (const long &t1, const long &t2)
{
  return t2 > t1 ? t1 : t2;
}
```

和宏替换相同，当前许多C++模板机制的实现也会使得执行代码的大小过度增长，并产生令人费解的错误消息，但至少底层的语言设施提供了必要的基础，实现了对程序员更为友好的转换工具。

在面向对象的语言中，多态函数(polymorphic function)可以通过在运行期间动态地调用适当的对象方法来实现。下面的示例就试图实现一个泛型的、面向对象的min函数。②

```
static Object min(Object a, Object b, int type) throws SQLException {
    if (a == null) { return b; }
    if (b == null) { return a; }
    if (compare(a, b, type) < 0) {
        return a;
    }
    return b;
}
```

① ace/ace/Min_Max.h:25-30

② hsqldb/src/org/hsqldb/Column.java:769-783

但是，这段代码只是将问题推给了 compare 函数，由它来显式地检查传递给它的参数的类型。①

```
static int compare(Object a, Object b, int type) throws SQLException {
int i = 0;
[...]
switch (type) {
[...]
case INTEGER:
    int ai = ((Integer) a).intValue();
    int bi = ((Integer) b).intValue();
    return (ai > bi) ? 1 : (bi > ai ? -1 : 0);
```

这种实现风格不灵活，易于出错(编译器不能检测失败的类型转换)，并且没有效率。针对这个问题，我们建议的面向对象的方案是：从支持 lessThan 方法的类或接口派生所有应用 min 函数的对象，并将 min 中的代码改为下面的样子。

```
if (a.lessThan(b))
    return a;
else
    return b;
```

作为一个具体的例子，下面的代码通过调用对象的 toString 方法来完成工作，这个函数根据底层对象的类型产生相应的结果②。

```
boolean compare(Object o) { [...]
    String s = o.toString();
    if (bIgnoreCase) { s = s.toUpperCase(); }
    return compareAt(s, 0, 0, s.length());
}
```

最后，和我们的预期相同，在 C 语言中，基于运行期间函数调用的泛型实现，常常用 void 指针(可以指向任意的值)表示数据元素，并使用函数指针来完成处理功能。下面这个数组顺序查找函数就是一个典型的例子③。

```
typedef int (*cmp_fn_t) (const void * , const void * );
static void *
linear_base (const void *key , const void *base,
size_t *  nelp, size_t, width, cmp_fn_t, compar, int add_flag);
{
register const char *element, *end;
end = (const char * )base + *nelp *  width;
for (element = base; element < end; element += width)
```

① hsqldb/src/org/hsqldb/Column.java:825-852

② hsqldb/src/org/hsqldb/Like.java:97-109

③ netbsdsrc/lib/libcompat/4.3/lsearch.c:50-101

```
            if (! compar(element, key)) /*  key found */
                return((void * )element);
        if (! add_flag) /*  key not found */
            return(NULL);
    [...]
    }
```

上面的函数在一个数组中查找 key 指向的元素，该数组由 * nelp 个宽度为 width 的元素组成，从 base 开始。传递给比较函数 compar 的参数是指向键的指针和每个后续元素，由该函数来确定特定的键是否找到。请注意，数组元素的两个要素，它们的大小与确定相等与否的方式，是如何显式地作为参数传递给函数的。在我们分析的另外一个例子中，二者都自动由编译器根据底层的类型自动导出。

编写特定概念的泛型实现需要切实的智力劳动。因此，泛型实现最常用在，通过对实现的广泛应用能够获得相当多好处的情况下。典型的应用包括(1)容器(经常作为抽象数据类型来实现，我们在 9.3.5 节中进行分析)；(2)相关的算法，比如查找、排序和其他作用在集合和有序区间上的操作；(3)泛型数值算法。

9.3.5 抽象数据类型

抽象数据类型(abstract data type，ADT)是一种封装数据结构的机制。它依赖于许多我们已描述过的内容：封装、接口与实现的分离、以及泛型实现。另外，ADT 的接口有时使用形式代数(formal algebraic)的规范，根据操作之间的关系进行描述；例如，C++标准模板库(standard template library，STL)文档规定，连续容器中 a. resize(n)⇒a. size()== n。ADT 的一项基本特征是，它表示的数据接口与底层数据格式保持独立。BSD 中数据库访问方法 dbopen[①] 就是一个典型的例子。对 dbopen 的调用，返回一个指向 DB 元素的指针。接下来，由它来提供对函数 get，put，del 和 seq(分别用来读取、添加、删除和顺序处理数据库记录)的访问。同一接口可以用来访问许多种不同的数据组织方法——B+树、扩展线性散列、内存池和定长或变长记录文件。

ADT 经常用来封装常用的数据组织方案(比如树、列表或栈)，或者对用户隐藏数据类型的实现细节。前者的例子包括我们业已分析过的 BSD dbopen 接口；BSD 中抽象单向链表、单向链式尾队列(singly linked tail queue)、表、尾队列(tailed queue)和循环队列(circular queue)的 queue 宏[②]；Java 的 java. util Hashtable，Vector 和 Stack 类；还有 C++的 STL 容器。ADT 用来隐藏底层实现细节的例子，可以参考 POSIX 目录访问函数：opendir，readdir，telldir，seekdir 和 closedir。在 Unix 的早期，程序要想获得目录中文件的列表，只需要打开包含目录内容的文件(例如，/usr/include)，并按照目录结构在磁盘上已知的存储方式对文件的内容进行解释。这种方式意味着，对目录数据的格式做出任何更改(例如，增加了文件名的允许长度)，所有依赖它的程序都要进行调整，以适应新的结构。目录访问函数将目录的内容呈现为 ADT，从而屏蔽了这类变更对实现的影响。

---

① netbsdsrc/lib/libc/db

② netbsdsrc/sys/sys/queue.h

9.3.6 库

库是模块的有组织集合，其中的所有模块都服务于公共的目标。许多情况下，库被打包成单个文件，其中包含一系列对象代码格式(object code format)的预编译模块。相应地，库经常伴有详细的文档，介绍库的接口以及(如果需要)允许它与其他代码进行集成的相关元素，比如头文件、定义文件或输入库占位程序(stub)。使用库的目的多种多样：重用源代码或目标代码，组织模块集合，组织和优化编译过程，或是用来实现应用程序各种特性的按需载入。

1. 代码重用

通用实用程序中稳定和高效的代码，在打包装成库的形式并编制适当的文档后，就可以高效地与其他程序共享。另外，打包库的元素、为它们的接口编制文档都需要大量的努力，故而，可以认为打包成库的代码具有很好的实用性、稳定性、易用性，并且能够得到很好的支持。因此，包装成库的代码比其他形式，例如以源代码形式分发的对象，更易于被采用。例如，NetBSD 系统在发行程序中包括 10 个以上不同的库，涵盖密码学①、屏幕光标定位②、命令行编辑③、内核数据访问④和网络包的捕获⑤。

2. 封装

库可以在不显式地揭示目标文件内部构成的情况下，方便地将多个目标文件集成到单个包中。例如，NetBSD 中的 C 库⑥由超过 600 个编译后的 C 文件组成，提供公共接口，如 bsearch⑦，和内部使用的函数，如__errno()⑧。因此，库将大量相关的文件打包为一个有机整体。

3. 结构

库经常用于将大型的系统组织成多个易于管理的单元。考虑图 9.24 中列出的 rayshade 光线跟踪渲染程序⑨的目录结构。除了 4 个文件位于 rayshade/rayshade 目录中以外，程序由 8 个不同的库组合构建而成。libshade 库包含程序的操作和场景描述模块，而 libray 库由提供光线跟踪功能的 6 个其他的库组成：libimage(图像载入函数)，libcommo(公共数学函数)，liblight(光源)，libobj(对象定义)，libsurf(表面属性)和 libtext(纹理)。

4. 编译过程的优化

库还经常用来降低编译过程的复杂性，提高它的效率。一般地，编译过程并非每次都要链接数百个目标文件，而是常常组织为链接较大的目标文件集合——库。库常常比等同的目标文件集合组织得更为合理有效，从而能够减少链接的时间。您可能会反驳，这只不过是

---

① netbsdsrc/lib/libcrypt
② netbsdsrc/lib/libcurses
③ netbsdsrc/lib/libedit
④ netbsdsrc/lib/libkvm
⑤ netbsdsrc/lib/libpcap
⑥ netbsdsrc/lib/libc
⑦ netbsdsrc/lib/libc/stdlib/bsearch.c
⑧ netbsdsrc/lib/libc/gen/_errno.c
⑨ *http://graphics.stanford.edu/~cek/rayshade/*

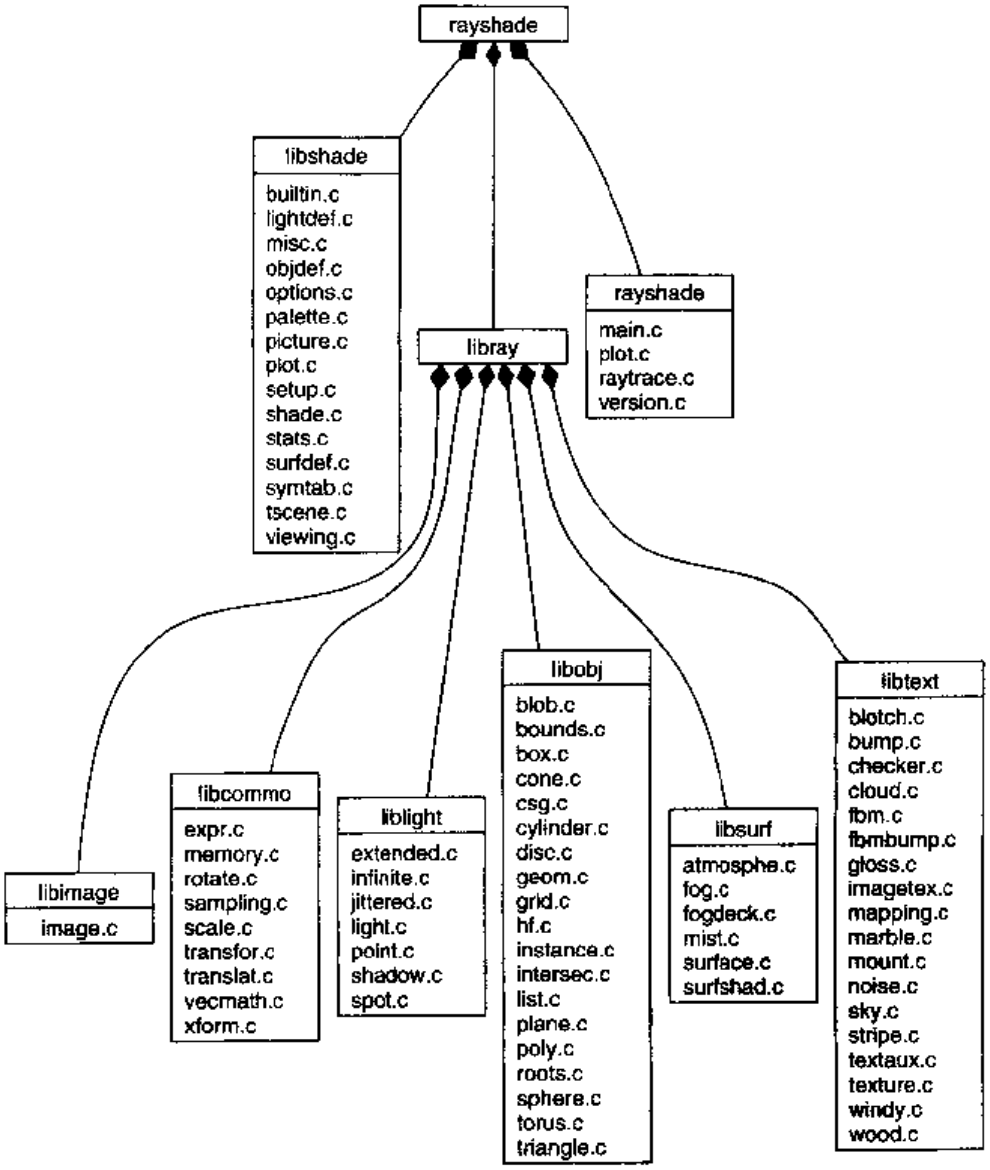


图 9.24　rayshade 程序中库的结构

转换成对实际库的编译，但由于大多数软件开发工作中都存在引用的局部性，所以这种说法并不正确。在系统的生命周期内，大部分的代码可能相对比较稳定；将这些代码包装到库中常常会更有效率。例如，apache Web 服务器的重要部分都是独立地编译，然后以库的形式进行链接。这些库包括 expat XML 语法分析工具箱[①]、sdbm 散列数据库的支持库[②]和 regex 包[③]。

5. 压缩代码的大小

在许多系统中，恰当地使用库，能够节省程序在存储和运行时所占用的空间。共享库（shared library）（在很多 Unix 变种中也称为动态共享对象，在 Microsoft Windows 中称为

① apache/src/lib/expat-lite

② apache/src/lib/sdbm

③ apache/src/regex

动态链接库或DLL)，允许机器代码或数据的单一实例被多个程序共享。提供相应代码的库，存储在一个全局已知的位置；需要用到这些库的程序，在载入后与库进行链接，从而减小了运行程序的大小。大多数操作系统还安排只载入库的单一副本，由多个程序共享，从而节省了内存的使用。例如，在Linux中，X Window系统共享库的大小约为1MB；每个存储在磁盘上的程序、每个载入内存的客户程序都节省了这1MB。然而，由于共享库与使用它们的程序彼此独立，经常会发生当前库的版本与程序当初链接的版本不匹配的情况。在此类情况下，最好的情况，程序或动态链接程序可以检测出这个问题并报告一个错误消息，然后终止运行；最坏的情况，程序可能表现出难以捉摸的bug，并且在不同的安装上不能重现。在Windows环境中，这种情形被准确地命名为"DLL地狱"。

6. **动态链接**

最后，库经常用来简化大型系统的配置工作——将配置工作从编译期间延期到系统的实际运行时执行。用这种方式构建的系统一般只将最通用的核心功能合并到系统的主体中。所需的其他所有特性都在系统执行时以库的形式动态进行链接。这种方式在保持基本系统很瘦的同时，并不影响它提供的功能。NetBSD和Linux的核心模块[①]，还有apache[②]和Perl[③]的动态模块载入机制，都是应用这种方案的重要例子。在Microsoft Windows中，DLL还经常用于动态地设置应用程序的本地化版本。另外，动态链接将开发过程中各个元素分离，允许分离的小组完全独立地工作，独立地提交编译库的可交付最新版本。

9.3.7 进程和过滤器

如果包所提供的功能定义十分良好，那么相关的接口会极为稳定，从而通信的需求会很简单，或者仅仅存在不可避免的开销，此时，常常用进程(process)封装对应的功能。大型的、分布式的系统经常实现为许多互相协作的进程，它们之间常常使用我们在9.1.1节中描述的机制进行通信。

作为一种封装机制，进程有许多优点。首先，进程起到故障隔离屏障的作用。例如，进程中不正确的指针，不能直接影响到其他进程内存中的内容。另外，进程允许我们单独分析和管理它使用的资源。通过操作系统管理工具，比如Microsoft Windows任务管理器和Unix ps命令，我们能够直接查看进程的内存和对CPU的利用情况、网络连接、打开的文件、线程和其他资源。一些分布式系统(比如apache Web服务器)甚至通过运行单个进程的多个实例来实现吞吐量的增长。在多处理器系统中，多个进程——和多个线程一样——自动地在CPU之间分配处理负载。

最后，进程，作为一个独立的编译实体，可以减轻团队协调的需求，减少编译错误对整个项目的影响，从而简化大型系统的编译过程。

作为一个具体的例子，请考虑图9.25中所示的GNU编译器集合[④]以及关联的后端工具(as,ld)的进程结构。一般通过运行编译器驱动程序(compiler driver)，比如传统的Unix

① nethsdsrc/sys/kern/kern_lkm.c

② apache/src/modules/standard/mod_so.c

③ perl/ext/DynaLoader/dlutils.c

④ *http://www.gnu.org/software/gcc*

C编译器接口cc或GNU C++前端g++，来调用编译序列。驱动程序分析命令行参数，运行若干进程：预处理器cpp，特有编译器(compiler-proper)，汇编器和链接器，编译每个源文件。进程之间通过中间文件或流水线进行通信。GNU编译器集合支持许多不同的语言；每种语言实现为不同的特有编译器进程，该进程的输入是预处理过的源文件，编译结果是汇编语言。我们描述的每个进程都单独进行维护，许多进程拥有独立的发布时间表。大多数更改一般只影响单个进程：新的Fortran语言特性只需要最新的f771可执行文件，可执行文件的新类型或许只影响到链接器。只在接口发生更改时(例如，新调试标记的加入)，才需要对多个可执行文件的发布进行协调。

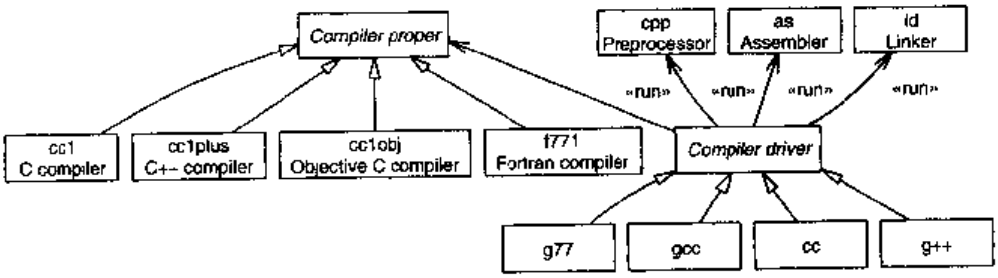


图9.25 GNU编译器系统的处理结构

过滤器(filter)是一种特殊类型的进程，它通过从输入到输出之间的数据流与其他过滤器进行交互。在9.1.2节中，我们介绍了如何通过组合多个过滤器来构造大型的系统。许多程序类型，只要它们遵循一些简单的规则(可以从程序的文档、操作和源代码中得到验证)，都可以看作和用作过滤器。首先，您要确保程序能够同步运行，并且不是在后台；Unix守护程序类型(daemon-style)的程序，以及Microsoft Windows服务和非控制台应用程序都不合格。另外，进程应该能够从标准输入接收数据(键盘或其他重定向的源)，将结果发送到标准输出(屏幕或重定向目标)，并且能够在没有用户干涉的情况下运行。如果该程序有交互式用户界面，那么界面操作应该是可选的，并且应该有选项可以禁止它。最后，程序的输入与输出应该遵循标准约定。许多有用的过滤器都操作简单的文本数据，将每一行看作是一项记录；某些情况下，由空格隔开的项也可以解释为单独的字段。如果输出的编排比较复杂，并且数据项跨越多个行，那么其他过滤器程序就难以使用这些输出。类似地，修饰性的标题、版权信息和依赖于屏幕的输出都应该避免，或者将它们设为可以选择。过滤器包装(filter-packaging)这种抽象的好处是，执行单一处理功能的简单程序常常还是高效的可重用过滤器。

9.3.8 组件

软件组件(component)是程序构成中典型的自包含单元(self-contained unit)，它有论述详尽的、定义良好的接口。大多数情况下，组件的部署不需要原开发者的参与，并且经常由第三方进行分发和使用。和对象相同，组件封装了状态，可以通过独立描述的接口对它进行访问，组件还支持以隔离为重点的模块化设计。然而，组件在许多方面又不同于对象：组件可以用不同的语言来实现，它们常常包装在二进制容器内，它们可以封装多个对象，并且它们的包装一般比对象要健壮。许多开发环境中，可以将新的组件无缝地集成到环境的框架中。常见的组件包装技术包括COBRA，ActiveX和JavaBeans。

图 9.26[①] 和图 9.27[②] 分别是一个简单的 JavaBean 组件和它在 JSP 中的应用。该 bean 实现了一个简单的猜数字游戏。可以看出，它使用标准的命名约定，提供对数据字段的访问（在组件上下文中，经常称为属性(property)）。在 Java 开发环境中，可以使用 Java 的反射(reflection)功能分析 bean 的内部结构并将它恰当地暴露给使用组件的开发者。如果 bean 的设计者觉得默认的行为没有效率，她或他总能够提供定制的方法，区别于默认行为。注意，各种组件技术和编程环境之间，组件开发者必须编写的支持代码的数量有根本性的不同。例如，用 Visual Basic 编写 ActiveX 只需要很少的支持代码，而相同的组件，用 C++来实现时，即使用 Microsoft 的 Active Template Library(ATL)，也需要大量复杂的代码结构。

```
public class NumberGuessBean {
  int answer;
  boolean success;
  String hint;
  int numGuesses;

  public NumberGuessBean() {
    reset();
  }

  public void reset() {
    answer = Math.abs(new Random().nextInt() % 100) + 1;
    success = false;
    numGuesses = 0;
  }

  public void setGuess(String guess) {
    numGuesses++;
    int g;
    [...] g = Integer.parseInt(guess); [...]
    if (g == answer) {
      success = true;
    } else if (g == -1) {
      hint = "a number next time";
    } else if (g < answer) {
      hint = "higher";
    } else if (g > answer) {
      hint = "lower";
    }
  }

  public boolean getSuccess() {
    return success;
  }
  public String getHint() {
    return "" + hint;
  }
  public int getNumGuesses() {
    return numGuesses;
  }
}
```

内部组件状态

组件的初始化

方法

1 set属性访问函数

2 get属性访问函数

图 9.26　一个简单的 JavaBean servlet 组件

① jt4/webapps/examples/WEB-INF/classes/num/NumberGuessBean.java:66-119

② jt4/webapps/examples/jsp/num/numguess.jsp:10-27

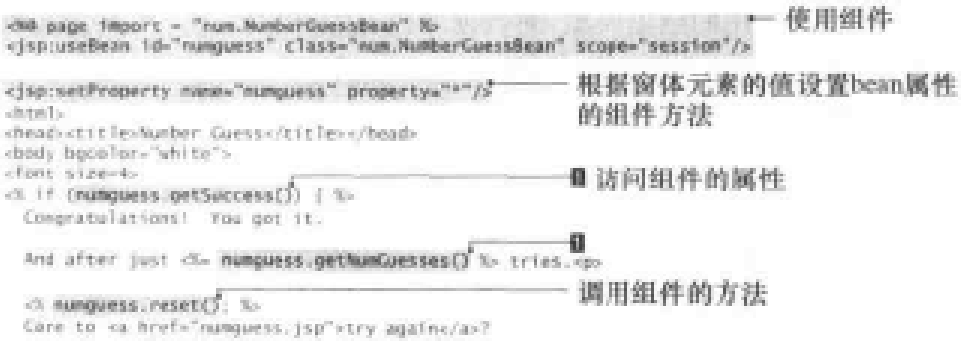


图 9.27　在 JSP 中使用 JavaBean

9.3.9　数据储存库

有时，分解会围绕纯数据储存库进行。数据储存库可能是简单的文件、目录层次、结构化文档、甚至完整的数据库。例如，Microsoft Windows 的最近版本将用户信息存储为 Document and Settings 目录下的一棵树，其中包括 Cookies，Desktop，Favorites，My Documents 和 Start Menu。Unix 系统将多种类型的配置数据存储在平面文本文件中。例如，文件/etc/inetd.conf 指定给定主机支持的 Internet 服务。对于那些可以组织成记录和字段形式的数据，可以使用关系型数据库作为集中化储存库；Microsoft Internet Information Server 的日志文件就是使用这种方法进行维护的。复杂的数据还经常存储在结构化文件中，对该文件的访问需要遵循不同应用程序共享的表达假定(RCS 修订控制系统的修订储存库就是使用这种方法)。虽然许多这类文件都按照专有的格式进行组织，但 XML 越来越广泛地被用作通用的底层表达架构。

很多情况下，对于基于文本的数据储存库，可以通过浏览存储在其中的数据，破译出它的结构。以 Unix Internet 守护程序(inetd)的配置文件①为例。

```
ftp     stream tcp nowait root /usr/libexec/ftpd ftpd -ll
telnet stream tcp nowait root /usr/libexec/telnetd telnetd
[...]
ntalk  dgram udp wait nobody.tty /usr/libexec/ntalkd ntalkd
```

通过查看这些项，可以看出，每行都包含服务名，使用的网络协议，守护程序是否应该等待程序的结束，服务以什么身份运行，以及调用服务的方式。另外，大多数 Unix 文件格式都在联机手册的第 5 节中提供。Microsoft 也在 Microsoft Developer Network Library(MSDN)中记述了它的应用程序使用的一些文件格式②。有时，可以通过从储存库中添加、修改或移除项，观察程序行为的变化，以此来增进对储存库的理解。

对于存储在关系型数据库中的数据，我们得采用一种不同的方案来得出它的结构。数据库数据是以二进制格式存储，并使用查询语言进行操作；从而，在程序的源代码和底层数据库文件之间，常常不存在易于辨别的对应关系。然而，所有的数据库都提供分析数据库模

① netbsdsrc/etc/inetd.conf,7-18

② *http://msdn.microsoft.com*

式的功能，模式是依照所采用的数据模型，对数据库内容的高层描述。关系型数据库系统一般在单独的数据库中存储每个表和列的信息，常称为数据字典(data dictionary)。可以通过查询数据字典中的表，或使用数据库专有的 SQL 命令，比如 show table，来分析关系型数据库的模式。

练习 9.13 讨论命名空间的管理问题，以及对代码重用的影响，并概括标准 C 语言库如何占用全局函数标识符命名空间。

练习 9.14 在本书配套盘中，找出 4 个能够重用于其他情形的模块。针对每个模块，描述最合适的包装机制。

练习 9.15 找到用您不熟悉的一种面向对象语言编写的代码，在其中找出处理类、对象、方法、字段、可见性、继承和多态的基本构造。如果您通晓多种语言，请画个表，描述上述特性在您所知道的语言中的语法。

练习 9.16 在本书配套盘中，找出受益于泛型实现的代码实例。

练习 9.17 给出您在本书配套盘中可以找出的链表实现的近似数目。

练习 9.18 找出使用运算符重载的实例，并归类它们的应用。如何确定给定的运算符被重载了呢?

练习 9.19 在 Web 上搜索包含代码库的流行网站。在本书配套盘中查找可以使用这些库来实现的代码。

练习 9.20 组件软件提供商经常将他们的商品列成按组件的流行程度排序的图表。找找这种图表，并讨论它的构成。

练习 9.21 找到一台运行 Microsoft Windows 的 PC，用 regedit 命令浏览它的注册表(存储系统配置的数据库)。不要使用其他信息，独立地描述它的结构。

9.4 构架重用

许多系统在设计时就遵循已经确立的构架。框架和代码向导(code wizard)是代码级构架重用中的两种机制。在更高层的抽象中，设计模式(design pattern)和各种领域专有的构架规定了一些构架性的元素，可以对这些元素进行剪裁以满足数量众多的各种需求。阅读重用现有构架的代码可能会极具挑战性，因为它的实现者常常会认为底层构架已经为人所知，不值得再去编写文档或进行注释。此外，可重用的构架，由于要满足一系列各种各样的应用程序，所以会比量身定做的、应用程序专有的结构更为复杂。另一方面，如果您正在分

析的应用程序确实遵循一种精心设计的构架风格，那么，可以确定这种构架风格在其他上下文中一定有过成功的应用，并且拥有相关的文档。识别出重用的构架元素后，可以查找其最初的描述，了解正确地使用这种构架的方式，以及可能出现的误用。

9.4.1　框架

大型系统的设计，常常会实现或依靠服务于共同目标（比如对 GUI 前端的支持或通信设施）的类的有组织集合。一般将这些类的集合和相关的接口称为框架。和单个类（细粒度的设计抽象）相比，框架拥有容纳足够结构性材料（类）的空间，可以表达复杂的构架。以 Adaptive Communication Environment（ACE）为例，在本书配套盘中可以找到①，它支持被动（reactive）和主动（proactive）式 I/O、对象的动态配置、层次通信服务的模块集成、各种高层的并发性、控制和同步模式、共享和局部内存的动态分配、以及 CORBA 集成。这些框架中都由多个可以和谐地一同工作的类构成，为用户提供可以方便地重用到其他系统中的子系统构架。

框架结构的一种特殊类型是 GUI 应用程序前端的实现②。提供一致的用户界面来正确地处理键盘和鼠标输入、窗口系统事件和协调多个输入输出配件（控制），可能是一项极度困难的任务。相关的系统，比如 Microsoft Foundation Classes（MFC）、Java Abstract Windowing Toolkit（AWT）和 Swing 库、以及基于 X Window 系统的 Xt 和 Motif 工具箱，一般实现一种 Model-View-Controller（MVC）框架。用户和环境输入的事件由控制器（controller）类接收和处理。处理过程负责更新应用程序的模型（model），生成相应的视图修改消息。应用程序视图（view）总可以通过查询模型（model）的状态重新生成。

尽管框架对开发工作提供了许多帮助，但使用框架构建的应用程序都不会简单。例如，在 Xt 中，在屏幕上显示一个消息框的简单应用程序③合计超过 800 行代码。可以用下面的事实来解释这种程序的复杂性：框架一般设计用来构建大型和复杂的应用程序，从而会将包罗万象和不可避免的复杂视图强加到使用它们的系统上。值得注意的例外是以领域专用语言包装的框架，比如 Tcl/Tk 和 Visual Basic（这二者都常常用来创建 GUI 应用程序）或 awk 和 Perl（经常应用于报告生成任务）。

要详细分析建立在某种框架之上的应用程序，行动的最佳路线就是从研究框架自身开始。许多框架中都提供不加渲染的应用程序，这些应用程序遵循框架的约定，不加入任何主要的功能。Xt 的消息框应用程序就是一例，NetBSD 空文件系统的实现是另一个例子④；但是，最常见的示例可能还是一个微不足道的“hello world”程序。仔细研究这个例子，编译它，试着修改它。这些行动能够显著地减弱框架最初陡峭的学习曲线，并能帮助您将功能性的代码，与只是满足框架需求的代码区分开来。

9.4.2　代码向导

许多框架结构陡峭的学习曲线，催生了一类我们称之为代码向导（code wizard）的工具。

---

① ace/ace

② vcf/src

③ XFree86-3.3/contrib/programs/xmessage

④ netbsdsrc/sys/miscfs/nullfs

它们向应用程序的开发者提出一系列的问题和选择项，然后生成固定的代码来满足用户陈述的需求。在阅读向导生成的代码时，不要期望太高，否则您会感到失望。创建这些代码的程序员，相信向导的判断更胜于他们自己，随后对代码的修改也肯定是基于对底层代码结构的有限理解。

由向导生成的代码，所具有的典型特征就是由向导编写的、指引应用程序定制者的注释①。

```
BEGIN_MESSAGE_MAP(CServerApp, CWinApp)
    //{{AFX_MSG_MAP(CServerApp)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_MSG_MAP
    // Standard file based document commands
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
    // Standard print setup command
    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
[...]
CServerApp::CServerApp()
{
  // TODO: add construction code here,
  // Place all significant initialization in InitInstance
}
```

向导生成的代码存在的另一个问题是，大多数向导，在它们生成的代码经过定制后，就不能对代码做出修改。从而，更改最初选项的惟一方式常常是从零开始再次生成代码。有时可以通过查看向导生成的代码（使用您假定使用的选项）与您手头的代码之间的不同，来节省一些工作。有些情况下，您甚至可以将这些更改（以上下文 diff 的形式生成——见 10.4 节），重新应用到使用不同选项生成的新向导代码中；但需要说明的一点是，所需的工作量可能会截然不同。

9.4.3 设计模式

对构架形式最通用的重用方式可能就是设计模式（design pattern）。设计模式的概念最初来源于建筑师 Christopher Alexander 的开创性工作，他将重复出现的问题以及它们各自的解决模式概括如下[AIS+ 77]：

每个模式描述一个在我们周围重复出现的问题，之后，介绍解决该问题的核心方案，这样，您就可以多次使用这个解决方案，甚至不需要重新考虑它。

20 年后，这些思想启发了可重用面向对象软件的设计领域。设计模式提供一种方便的

① ace/TAO/examples/mfc/server.cpp:26-110

方式，可以用一致和易于理解的格式，捕捉、记录、组织和传播给定领域的现有知识。模式所描述的概念不能编写或用作一个子例程或一个目标类，在这一点上，它不同于算法和数据结构。模式还不同于框架，它们并不描述一个完整系统的结构；相关的模式一般会一同使用，共同解决给定上下文中通用的设计问题。对模式的描述一般都遵照一种相当标准的轮廓，由下面这些项组成：

- 模式名，比如 Singleton 或 Reactor，用来标识该模式；
- 模式的结构图解，使用 UML 图；
- 模式所属的分类，例如，创建型（creational）、动作型（behavioral）或结构型（structural）；
- 设计问题的说明，提供使用该模式的动机；
- 模式适用情形的概括；
- 模式参与者的概括；
- 模式如何支持它的目标；
- 模式实现的例子和习惯性的准则。

通过分析 Singleton 模式的实现，我们能够得出一般代码中如何使用模式。Singleton 模式用在设计者需要确保给定对象只会创建单一实例的情况。一种解决方案是将该实例存储在一个全局变量中。然而，这种方案不能阻止其他的实例被实例化并使用，并且污染了程序的命名空间，还阻碍了通过派生子类对类进行扩展。Singleton 模式让类中的一个变量（在 Java 和 C++中声明为 static）存储单一对象的惟一实例，同时提供一个方法返回这个惟一实例（并且，在需要的时候，创建这个实例），并保护类的构造函数不被外部访问从而解决了这个问题。图 9.28[①] 和图 9.29[②] 说明了同一模式在两种不同情况下的应用。它们在很多方面存在不同：它们用不同的语言编写，它们用来保护不同的底层类，C++实现是泛型的，它们用在不相似的系统中。然而，模式的定义特征、类的惟一实例变量和访问方法、以及受保护的构造函数都十分明显。学习几个基本的设计模式之后，您会发现，您查看代码构架的方式会发生改变：您的视野和词汇将会扩展到能够识别和描述许多通用的形式。在阅读遵循特定模式的代码时，可以试着以模式的说明作为理解代码设计的捷径。

构架性设计的重用经常先于模式的形成。实际上，大多数模式说明都会引用之前对给定模式的已知应用。因此，经常会遇到频繁使用的模式，但并不显式地指出它们的名称。例如，之前在图 9.8 中说明的同步/异步代码界限就是一个标准的设备驱动程序设计，从 Unix 的早期版本直到 MS-DOS 和 Microsoft Windows，都应用这种设计；直到 1995 年，它才被正式描述为一种设计模式并称为 Half-Sync/Half-async 模式。因此，请试着按照底层模式来理解构架，即使代码中并没有明确地提及模式。

9.4.4 领域专有的构架

某些应用程序类别的结构经过多年的稳定，几乎已经成为普遍采用的领域专有构架。

① cocoon/src/java/org/apache/cocoon/components/renderer/ExtendableRendererFactory.java:26-81

② ace/TAO/tao/TAO_Singleton.h:83-112

```
public class ExtendableRendererFactory implements RendererFactory {

    protected final static RendererFactory singleton =
        new ExtendableRendererFactory();

    private ExtendableRendererFactory() {
        // Add the default renderers which come with Apache FOP.
        addRenderer("application/pdf", PDFRenderer.class);
        addRenderer("application/postscript", PSRenderer.class);
        addRenderer("application/vnd.hp-PCL", PCLRenderer.class);
    }

    public final static
        RendererFactory getRendererFactoryImplementation() {
            return singleton;
    }

    [...]
}
```

惟一的实例字段

受保护的构造函数

惟一的实例访问方法

图 9.28 Singleton 模式在 Java 文档提供程序中的应用

```
template <class TYPE, class ACE_LOCK>
class TAO_Singleton : public ACE_Cleanup
{
public: [...]
  static TYPE *instance (void);
  [...]
protected: [...]
  TAO_Singleton (void);
  [...]
  static TAO_Singleton<TYPE, ACE_LOCK> *singleton_;
  [...]
}
```

惟一的实例访问方法

受保护的构造函数

惟一的实例字段

图 9.29 一个 C++Singleton 模板

例如，大多数处理某种计算机语言的应用程序，不管是配置文件或编程语言，一般都将任务划分成，将字符组织成标记(token)的过程——(lexical analysis，词法分析)，以及将标记汇编成更大结构的过程——(parsing，语法分析)。从图 9.30①，图 9.31②，图 9.32③ 和图 9.33④ 中，可以看到同样的构架(词法分析器为语法分析器提供标记)如何用在 4 种不同的应用程序中。这个领域的子领域拥有更为特殊的构架：编译器一般遵循，用不同的源代码语义分析进行词法分析和语法分析、优化和代码生成等步骤；大多数解释器将语法分析后的代码转换成内部分析树或字节码表示，之后用状态机或虚拟机对程序进行处理。

从图 9.34⑤ 中，可以看出，sed(stream editor，流编辑器)命令解释器围绕一个循环构建，该循环不断读取指令码，并按指令的内容执行相应的动作。分析后的程序在内部存储为一个链表，其中每个元素包含一个指令码和相关的参数。变量 cp 是解释器状态(interpreter state)的一部分，它总是指向要执行的指令。许多变量，比如 appendx，保存程序状态(program state)。大多数解释器都遵循类似的处理构架，围绕一个状态机进行构建，状态机的操作依赖于解释器的当前状态、程序指令和程序状态。

① netbsdsrc/usr.sbin/dhcp/server/confpars.c:167-192

② netbsdsrc/usr.sbin/named/named/db_load.c:164-277

③ netbsdsrc/lib/libc/time/zic.c:773-829

④ hsqldb/src/org/hsqldb/Parser.java:503-533

⑤ netbsdsrc/usr.bin/sed/process.c:105-257

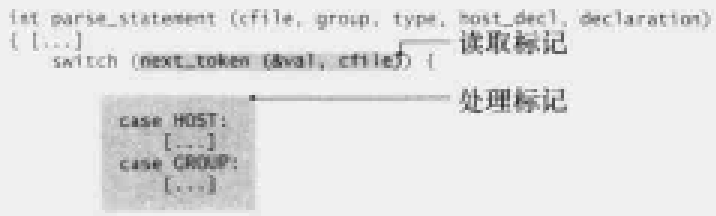


图 9.30　读取 DHCP 守护程序的配置文件

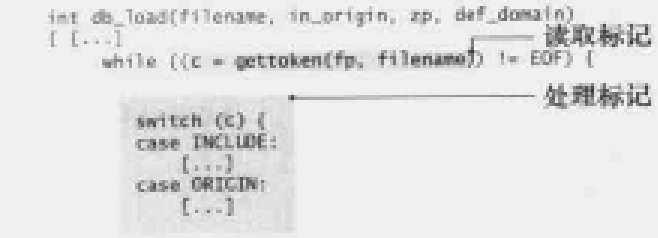


图 9.31　读取 DNS 规范

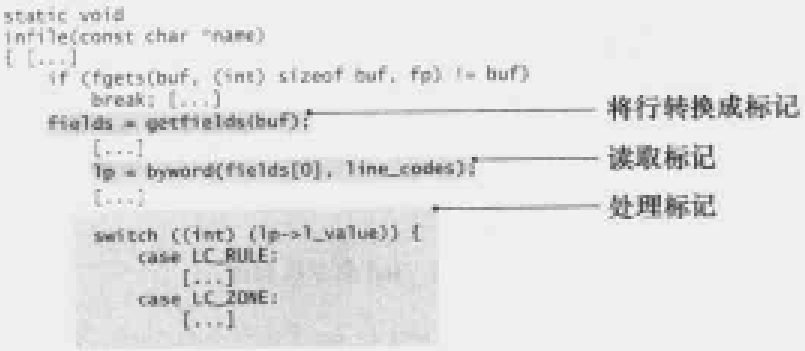


图 9.32　编译时区文件

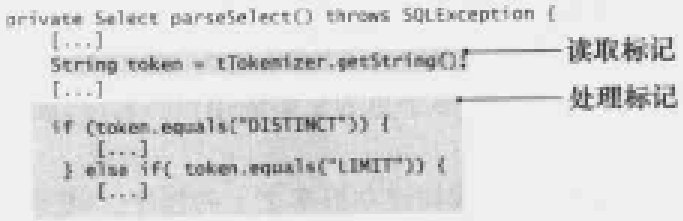


图 9.33　分析 SQL SELECT 语句

另一种常见的领域专有构架称为参考构架(reference archchitecture)。多数情况下，参考构架只是为应用程序域指定一种概念性的结构，具体的实现并非必须遵照这种结构。Open System Interconnection(OSI)的概念性构架就是这种类型的构架。OSI 构架规定网络系统应该划分为由 7 个网络层组成的栈，它们是：应用层、表示层、会话层、传输层、网络层、数据链路层和物理层。我们经常会听到或读到有关特定层功能的争论。①

① netbsdsrc/sbin/mount_nfc/mount_nfs. c:577-578

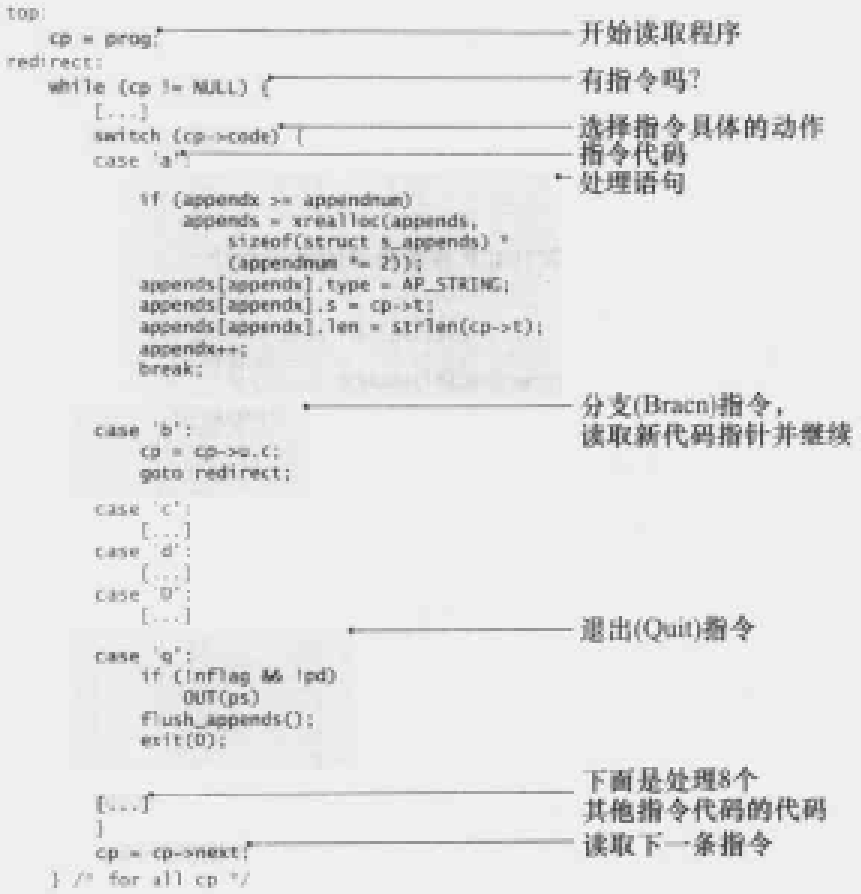


图 9.34 sed 命令解释器

```
 * DUMP!! Until the mount protocol works on iso transport, we must
 * supply both an iso and an inet address for the host.
```

然而,软件很少采用 ISO 模型中指定的 7 层结构。考虑到效率、历史实践和互相冲突的互操作性需求,在实现中,常常会由同一软件模块处理不同的 ISO 层,或不同的子系统互相协作服务于单个层。例如,会话层一般不适用于 TCP/IP 互连,而 BSD Unix 系统一般将路由的处理划分成一个简单的内核模块①,和多个更为复杂的用户态程序,如 route② 和 routed③。

练习 9.22 在本书配套盘中,查找构建在框架之上的程序。(您将会如何寻找呢?)区分执行处理任务的代码和满足框架需求的代码,比较这两种代码的大小。

练习 9.23 使用某种代码向导创建一个应用程序。分析产生的代码,并标识出如果不使用自动代码生成机制,能够略去的代码。

① netbsdsrc/sys/net/route.c

② netbsdsrc/sbin/route

③ netbsdsrc/sbin/routed

练习 9.24　ACE 通信框架[①]使用许多不同的模式：Acceptor，Active Object，Adapter，Asynchronous Completion Token，Component Configurator，Connector，Decorator，Double Checked Locking，Extension Interface，External Polymorphism，Half-Sync/Half-Async，Interceptor，Iterator，Leader/Followers，Monitor Object，Object Lifetime Manager，Observer，Proactor，Reactor，Scoped Locking，Service Configurator，Singleton，Strategized Locking，Strategy Bridge，Thread-per Request，Thread-per Session，Thread-Safe Interface，Thread Pool，Visitor 和 Wrapper Facade。针对每种模式，找出相应的代码，并参照模式的文档，在代码中确定出模式的不同功能。

练习 9.25　下载 GNU C 编译器的源代码[②]，找出它的代码与典型的编译器构架之间的对应关系。

练习 9.26　讨论 NetBSD 网络代码[③④⑤⑥]与 ISO 网络栈的对应关系。

进　阶　读　物

几种参考书中论述了软件构架[Sha95，SG96，BCK98，BHH99]和设计模式[CS95，GHJV95，BMR+96，SSRB00]。Sinha[Sin92]对客户端-服务器计算做了概括，Comer 和 Stevens[CS96]中对许多广泛应用的 Internet 客户端-服务器应用程序进行了详细的分析。Hunt 和 Thomas[HT00 pp. 155-170]中对黑板系统做了简明扼要的描述。我们概括的中间件规格说明在有关 CORBA[Obj02，Sie99]，DCOM[Mic95]，Java RMI[Sun99b]和 Sun RPC[Sun88a，Sun88b]的参考书目中进行了描述。Unix 系统提供大量的机制，支持数据流构架：参见 Kernighan 和 Pike[KP84]，Kernighan[Ker89]——它介绍了数据流构架在文档准备领域中的应用。Meunier[Meu95]中将管道和过滤器数据流构架(piples-and-filters data-flow architecture)作为一种设计模式进行了描述。我们用来建模面向对象构架的统一建模语言(Unified Modeling Language，UML)定义在 Rumbauth 等著的[RJB99]中，Booch 等著的[BRJ99]中进行了介绍；Daniels 的文章[Dan02]简明地描述了如何在实践中使用各种 UML 图。Robbins 和 Redmiles[RR00]中介绍了 ArgoUML 工具。Dijkstra 的操作系统实现中首先将分层的概念用作一项设计原则，并已经应用到操作系统开发[Org72]、计算机网络[DZ83]和 Java 可移植层[LY97]。然而，您还应该研究描述端到端系统设计的开创性论文，以及其他影响传统分层网络协议的构架性考虑[CT90]。切片技术最初由 Weiser[Wei82]引入；可以在后续的著作[BE93，JR94，WHGT99]中阅读切片如何协助代码的理解和逆向工程。Constantine 和 Yourdon[CY79]中对耦合性和内聚性的概念进行了定义。Ott 和

① ace/ace

② *http://www.gnu.org/software/gcc*

③ netbsdsrc/sys/net

④ netbsdsrc/sys/netinet

⑤ netbsdsrc/sys/netiso

⑥ netbsdsrc/sys/netnatm

Thuss[OT89]对切片和内聚性之间的关系做了分析。Lieberherr 和 Holland[LH89]以及 Hunt 和 Thomas[HT00, pp. 140-142]对得墨忒耳定律(Law of Demeter)做了介绍。Leffler 等著的[LMKQ88, pp. 69-165]介绍了更多有关 BSD Unix 内核处理管理的内容。Ran [Ran95]以模式的形式介绍了以事件为中心的通用构造;Schmidt[Sch95]介绍了事件的解多路复用。

我们在 9.3 节中介绍的元素包装方案是基于 Shaw 和 Garlan[SG95]中的结构化属性模型。Meyer[Mey00]更为详尽地介绍了模块性,Parnas[Par72]持续不断地对这一主题进行研究。Salus[Sal98]是介绍面向对象语言的方便参考;下面是特定语言的权威性参考,Stroustrup[Str97]介绍 C++,Microsoft[Mic01]介绍 C#,Arnold 和 Gosling[AG00]介绍 Java,Conway[Con00]介绍面向对象的 Perl,Goldberg[Gol80]介绍 Smalltalk。在 Ellis 和 Stroustrup[ES90]中可以找到有关对象内部实现的更为详细的论述。Plauger[Pla93, pp. 177-209]论述了抽象方法和方案,[Aus98, Ale01]论述了 C++框架下的泛型编程。[BW98, Szy98, Wil99, SR00]这几本参考材料介绍了基于组件的软件开发。有关使用领域专有框架的实践报告、应用程序和相关问题可以在 Fayad 等著的[FJ99, FJS99a, FJS99b]中找到。Johnson[Joh92]介绍了一种使用模式来记录框架的方法。

许多书都涵盖模式和模式语言[GHJV95, BMR+ 96, Ris00, Tic98, Vli98]。*International Conference on Pattern Languages of Programming and the European Conference on Pattern Languages of Programming* 汇编中的某些文章[CS95, CKV96, MRB97, HFR99]也很有意义。Schmidt 等著的[SSRB00]详细介绍了 Adaptive Communication Environment(ACE)(可以在本书配套盘中找到①)。如果您对这个主题感兴趣,那么您会很喜爱 Alexander 关于构架性模式的著作[Ale64, AIS+77]。

有关语言处理器构架的权威性参考是 Aho 等著的[ASU85]。其他涉及领域专用构架的领域包括操作系统[Tan97]、计算机网络[Com00, CS98]和客户端-服务器系统[DCS98]。

① ace/ace

第 10 章　代码阅读工具

给我们工具，我们就会完成工作。

——Winston Churchill

阅读代码时，绝大多数情况下所阅读的代码都可以在线访问。这意味着，我们能够使用工具来处理源代码，以提高阅读的效率，加强对代码的理解。下面是在代码体上可能执行的一些典型任务：

- 识别出一个特定实体的声明，确定函数、变量、方法、模板或接口的类型。
- 找到特定实体定义在什么地方，例如，找到函数或类的定义体。
- 找出所有用到某个实体的位置。
- 列举违背编码规范的代码。
- 发现对理解给定代码段有所帮助的代码结构。
- 查找解释特定特性的注释。
- 检查常见的错误。
- 查看代码的结构。
- 了解代码如何与它所在的环境交互。

本章中，我们将介绍一些工具，它们可以用来自动执行上述任务，并且是以最高效的方式执行。另外建模工具经常能够帮助逆向工程系统的构架，同时，大量的文档工具能够从特殊格式的源代码中自动地创建项目的文档。（在 9.1.3 节和 8.5 节中，有更多关于如何使用此类工具的介绍。）本章中许多工具和示例基于 Unix 环境和 Unix 的工具家族。如果您使用的工具与 Unix 的设施不相配，请检查本章最后一节，得到如何获取兼容工具的建议。我们基于源代码和执行文件之间日益增长的亲合性，对工具进行分析。我们从在词汇层面对源代码进行操作的工具开始（即：它们仅仅处理字符，不分析程序的结构），接下来介绍基于代码分析和编译的工具，最后概括依赖于代码执行的工具。

10.1　正则表达式

在源代码上执行词汇层面操作的工具很强大，但是人们常常对它们没有给予足够的重视。它们可以用于任何编程语言或平台，不需要预处理或编译文件就可以运作，处理很多种不同的任务，反应灵敏，并且能够处理任意数量的程序文本。使用这类工具，可以高效地在一个大代码文件中或者跨多个文件查找某种模式。这些工具天生就不精确，然而，使用它们能够节省时间，避免一些手动的浏览工作。

许多词汇工具的能力与灵活性都来自于正则表达式的应用。可以将正则表达式看作是匹配字符串的“处方”。正则表达式由一系列字符构成。大多数字符只与它们自己匹配，一些字符，称为元字符（meta-character），它们拥有特殊的含义。我们通过将常规字符与元字

符组合起来，创建正则表达式，指定一个能够精确匹配我们想要查找的代码项的处方。我们在表 10.1 中列出了正则表达式中常用的构建块。

表 10.1 正则表达式中常用的构建块

字符	匹配
.	任意字符
[*abc*]	字符 *a*，*b* 或 *c*（字符类）中的任意字符
[^*abc*]	除 *a*，*b* 和 *c* 之外的所有字符
a *	0 个或多个 a
\m	（元字符）m
^	行的开始
$	行的结尾
\<	单词的开始
\>	单词的结尾

大多数程序编辑器，都提供一种使用正则表达式查找字符串的命令。表 10.2 列出一些流行的编辑器中使用的具体命令。表 10.1 中概括的正则表达式的语法适用于大多数编辑器；vi，Emacs 和 Epsilon 都支持这种形式。大多数编辑器都加入了额外的功能，扩展了正则表达式的语法；有些编辑器使用稍微不同的语法。例如，BRIEF 编辑器使用？匹配任意字符，%匹配行的开始，～表示非匹配字符类，＋表示与前面正则表达式的一个或多个匹配。可以查询编辑器的文档，了解它所支持的完整正则表达式语法。

表 10.2 流行编辑器中的正则表达式查找命令

编辑器	前向查找命令	后向查找命令
BRIEF[a]	Search Forward	Search Backward
Emacs	C-M-s[b] isearch-forward-regexp	C-M-r isearch-backward-regexp
Epsilon	C-A-s[c] regex-search	C-A-R reverse-regex-search
vi	/	?
Visual Studio	FindRegExpr	FindRegExprPrev

a：首先要做正则表达式切换

b：Control-Meta-s

c：Control-Alt-s

有了支持正则表达式的编辑器，并对它们的语法有了基本的了解之后，就能够用它们来高效地浏览较大的代码文件。为了避免意外地更改文件的内容，请确保您对文件没有写权限，或以只读模式使用编辑器（例如，vi 编辑器中使用 vi -R 命令）。

图 10.1[①] 中，我们列出了一些使用正则表达式可以执行的查找类型。利用程序员使用

① netbsdsrc/sys/arch/sparc/fpu/fpu_sqrt.c:189-300

的编程风格，可以找出特定的程序元素。常见的情况是查找函数的定义(图 10.1:1)。大多数编程风格指导都规定在函数定义中要将函数名从单独的一行写起。从而，我们可以通过搜索正则表达式"function name，找出函数的定义。这个正则表达式将会与所有以 function name 开始的行匹配。这种表达式还会与名称中开头的字符序列与您指定的函数名相同的函数匹配。可以修正对应的正则表达式，使它将函数名作为一个完整的单词进行匹配："function name\>。

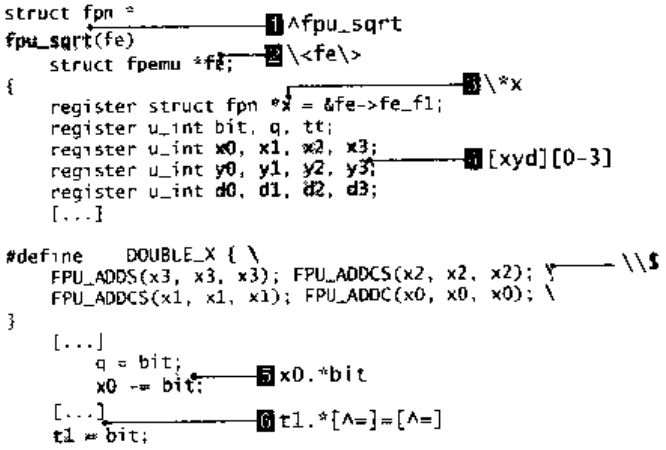


图 10.1　正则表达式的匹配

实践中，您会发现，简单的正则表达式规则完全能够胜任查找文本的工作，只有在最初的表达式生成太多不相关的匹配项时，才需要给出更为详细的表达式。有时，您还会发现仅仅指定查找字符串的一部分就能够正确地找到相关的内容。另一些情况下，当您寻找一个名称比较短的变量，该名称可能用在许多不同的上下文中，您会发现需要在它周围精确地指定单词的定界符(\<,\>)(图 10.1:2)。不幸的是，许多正则表达式元字符还用在编程语言中。从而，在查找包含它们的程序元素时，要对它们进行转义(使用反斜杠)(图 10.1:3)。正则表达式的字符类对于查找名称遵循特定模式的所有变量很有用(图 10.1:4)，而否定字符类可以用来避免非积极匹配(false-positive match)。例如，图 10.1:6 中的表达式能够找出对 tl 赋值的地方(找出变量在什么地方被修改对于理解代码来说往往至关重要)，但不会匹配出 tl 出现在相等运算符(==)左边的那些行。用来表示 0 或多个(zero-or-more)匹配的元字符(*)，常常和表示任意字符(match-any)的元字符(.)一同使用，来指定任意数量的未知文本(.*)。当给出这类表达式时，正则表达式的匹配代码，不会贪婪地匹配所有的字符直到行的结尾，而是试图只匹配足够的字符，创建整个正则表达式的一个匹配(包括模式在.*后面的部分)。图 10.1:5 中，我们使用这项特性找出所有变量 x0 后跟变量 bit 的行。

练习 10.1　了解您所使用的编辑器提供的正则表达式语法。体验一下它提供的所有额外特性。试着只用我们描述的元字符来表达这些额外特性。评论丰富的正则表达式语法相关的优点与学习它的成本。

练习 10.2　编写正则表达式，找出字符串中的整数、浮点数和给定单词。您会在什么地方用到这类表达式?

练习 10.3 编写正则表达式，找出您正在阅读的源代码中，与代码风格指南相违背的地方。

10.2 用编辑器浏览代码

一些编辑器（例如 Emacs 和 vi），将正则表达式的查找与索引文件结合起来使用，可以高效地找出源代码中的各种定义。首先要用 ctags 工具处理所有感兴趣的源文件，创建一个索引文件。该索引文件，命名为 tags，包含来自源文件的一系列有序定义。对于 C 语言，这些定义包括函数，#include，以及可选的 typedef，struct，union 和 enum。tags 文件中的每一行都包含识别出来的实体名称，所在文件，以及在文件中定位该实体所用的正则表达式。下面这些行来自 sed 的源代码所产生的 tags 输出：[①②]

```
NEEDSP process.c        /^# define      NEEDSP(reqlen) \\ $ /
compile_addr    compile.c       /^compile_addr(p, a) $ /
e_args defs.h   /^enum e_args { $ /
s_addr defs.h   /^struct s_addr { $ /
```

使用正则表达式，而非行号，使得即使在源文件中插入或移除某些行之后，tags 文件依旧适用。在浏览源代码时，一个特殊的命令（在 vi 中是^]，Emacs 中是 M-.）可以扫描光标下的实体名称，并跳转到该实体的定义点。因此，使用 tags 设施，您能够快速地查看，阅读代码过程中遇到的新实体的定义。由于 tags 文件的格式十分简单明了，从而可以容易地编写简单的工具，为其他源语言或具体的应用程序创建定制的 tag 文件。

图 10.2 中的 Perl 程序，可以从 Microsoft Visual Basic 源代码创建 tags 文件，之后就可以用任何支持 tags 的编辑器浏览 Visual Basic 的源代码。Visual Basic 源代码文件的精确格式化简化了程序的运作。这个程序的主循环检查当前目录下所有可能的源文件，匹配好像是控件、子例程、或函数定义的行。数组 lines 包含生成的所有查找模式的清单，从而，当读完所有的输入后，可以将它以有序的形式写入 tags 文件中。虽然这个程序尚有许多有待改进的地方，但是，和其他基于词汇结构的操作一样，它对于大多数用途来说已经足够好了。

```
@ARGV = (<*.frm>,<*.bas>,<*.ctl>);          ← Visual Basic源文件

while (<>) {                                ← 遍历所有文件的全部行
    chop;
    if (/Begin\s+[^ ]+\s+(\w+)/) {          ← 这是一个控件吗?
        push(@lines, "$1\t$ARGV\t?$_?\n");  ← 将标记行加入到tags数组中
    } elsif (/(Sub|Function)\s+(\w+)/) {    ← 这是子例程? 还是函数?
        push(@lines, "$2\t$ARGV\t?$_?\n");  ← 标记行包括实体名、文件名、模式
    }
}

open(T, '>tags') || die "tags: $!\n";       ← 从tags数组生成有序标记文件
print T sort @lines;
```

图 10.2 从 Visual Basic 源文件生成 tags

大量的工具都是构建在最初的 ctags 思想之上。

① netbsdsrc/usr.bin/sed

② 除非另外注明，本章中所有的例子都使用同一组源文件。

- idutils 索引工具[①]创建一个数据库，其中存储程序的标识符以及它们在一系列文件中出现的位置。然后，就可以使用单独的命令行工具在数据库中查询包含某个标识符的文件、编辑满足某个查询的一系列文件、列出文件中的权标(token)、或用 grep 查找包含某个标识符的文件子集，输入出现标识符的行。idutils 可以使用用户提供的扫描器，进一步增强自身的功能；许多编辑器，比如 Emacs 和 vi 的新变体，如 vim，都支持它们。与 ctags 相比，idutils 处理更为广泛的“标识符”，包括包含文件(include file)中的文字和名称在内。
- exuberant ctags 索引工具[②]提供对 23 种不同语言的支持，增强 ctags 的功能。

在浏览大型的代码文件时，可以使用编辑器的大纲视图鸟瞰源代码的结果。GNU Emacs 编辑器提供选择性显示命令(selective display)(C-x $ —— set-selective-display)，这个命令可以隐藏缩进超过特定列号的行。这个命令取一个数值型参数——需要隐藏的缩进列号；通过使用适当的数字，可以隐藏掉嵌套较深的源代码。还可以使用 Emacs 的大纲模式(outline mode)(M-x outline-mode)，并恰当地定义标题行，在文件的树状视图与相应的结点内容之间快速切换。vim 编辑器也通过 folding 命令支持大纲的概念(例如，set foldenable，Zm，Zr，Zo，Zc)。这些功能都允许我们折叠起部分源代码，从而获得文件结构的概况。

另一种快速整体查看许多源代码的方式是，在 Windows 中将源代码载入 Microsoft Word 中，然后将显示比例设为 10%。每个代码页将会显示为大约一张邮票大小，从行的形状中可以得出大量代码结构的信息。例如，在图 10.3[③]，可以一目了解地得出比较重要的语句、方法块和大量缩进的代码。

我们还可以使用编辑器来检测匹配的圆括号、方括号和花括号。每种编辑器相应的命令各不相同。Emacs 提供一系列列表命令：C-M-u(backward-up-list)在列表结构中向上反向移动，C-M-d(down-list)向下正向移动，C-M-m(forward-list)正向移动，以及 C-M-p(backward-list)反向移动。vi 编辑器提供%命令，该命令将光标移到匹配的括弧元素。最后，在 Microsoft Visual Studio IDE 中，我们可以使用 LevelUp 和 LevelDown 命令跳转到不同的级别，用 GoToMatchBrace 和 GoToMatchBraceExtend 命令来检测或选择与一个给定花括号匹配的文本。(在 Visual C++中，使用【Tools】|【Customize...】可以打开 Customize 对话框，然后选择【Keyboard】标签，在【Category】中选择【All commands】，就可以在下面的【Commands】列表框中找到相应的命令。——译者注)

练习 10.4　了解并试验您正在使用的编辑器提供的 tags 功能。能够用外部工具对它进行扩展呢?

练习 10.5　本地化 vi 编辑器的输出消息时，需要复制基于英语的文件，将其中的消息替换为使用新的本地语言的字符串。遗憾的是，为了确认每个消息参数的目的，需要在整个

① *http://www.gnu.org/directory/idutils.html*

② *http://ctags.sourceforge.net/*

③ cocoon/src/java/org/apache/cocoon/transformation/LDAPTransformer.java

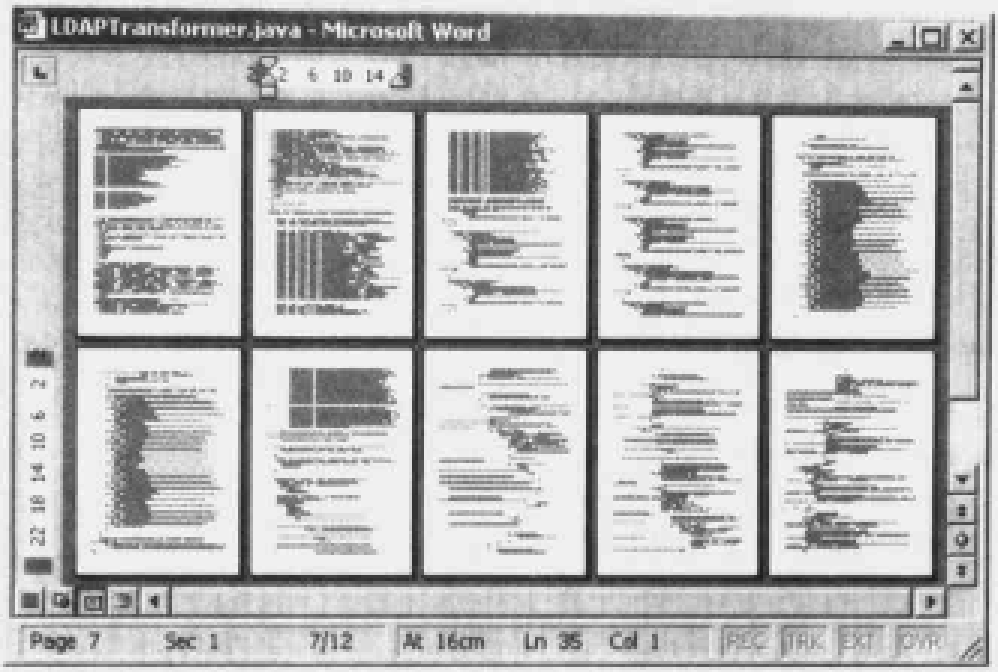


图 10.3 Microsoft Word 中源代码的鸟瞰视图

源代码中查找每条消息。[①] 编写一个 tag 工具，自动执行这项处理。

练习 10.6 建议 tags 功能的其他应用。考虑庞大的代码，比如 Unix 内核[②]。

练习 10.7 您正在使用的编辑器或编程环境支持查看源代码文件的大纲吗？使用本书配套盘中提供的例子试验这项特性。

10.3 用 grep 搜索代码

大型项目一般拆分成多个文件，这些文件有时还会组织到一个目录结构中。在编辑器内查找每个文件一般不太现实；幸运的是，有些工具能够自动完成这项任务。所有用来在大量代码中进行查找的工具都起源于 grep，grep 这个名字来自于 ed/ex 编辑器中的一条命令（g/RE/p——在文件中全局查找与给定正则表达式匹配的行，并打印出来），该命令打印所有与给定模式匹配的行。grep 的参数是所要查找的正则表达式和查找的文件列表。文件一般用通配符模式来指定，例如，*.c *.h。正则表达式中使用的许多字符对命令行外壳还有一种特殊含义，所以最好将正则表达式用引号引起来。下面的命令序列，显示 sed 的源代码中包含函数 strregerror 定义的文件和相关的行。

```
$ grep '^strregerror' * .c
misc.c:strregerror(errcode, preg)
```

① netbsdsrc/usr. bin/vi/catalog/README:113-125

② netbsdsrc/sys

如果要找出函数的定义以及所有的应用，我们应该仅仅使用函数名作为正则表达式。

```
$ grep strregerror * .c
compile.c:  err(COMPILE, "RE error: %s", strregerror(eval, * repp));
misc.c:strregerror(errcode, preg);
process.c: err(FATAL, "RE error: %s", strregerror(eval, defpreg));
```

grep 不需要像程序代码那样严格。如果您不确信所要查找的准确内容，或者不知道从何处着手，可以使用 grep 在代码中查找一个关键词，希望这个单词出现在注释中或是某个标识符的一部分。为了确保大写和单词的词尾和衍生词不会过分限制您的查找空间，在查找时可以使用词干，省掉它的首字母。下面的例子中，我们查找与编译过程相关联的代码。

```
$ grep ompil * .c
[...]
compile.c:                p = compile_re(p, &a->u.r);
main.c: *  Linked list pointer to compilation units and pointer to
main.c: compile();
main.c: *  Add a compilation unit to the linked list
```

grep 还可以用来查找其他工具的输出。例如，在含有大量文件的项目中，可以使用 grep 在文件清单中查找感兴趣的文件。①

```
$ ls | grep undo
v_undo.c
```

有时，您不希望阅读包含某个正则表达式的代码，而只是对包含它的文件感兴趣。grep 的-l 选项会显示包含指定正则表达式的每个文件的文件名(不重复显示)。

```
$ grep -l xmalloc * .c * .h
compile.c
main.c
misc.c
extern.h
```

然后，可以使用这些输出，在所有这些文件上自动执行一项特殊的任务。典型的任务也许是进一步审查每个文件，或者锁定修订控制系统下的每个文件。在类 Unix 外壳中，可以通过将 grep 的输出包在`(backtick，反撇号)中来完成这项任务。

```
emacs `grep -l xmalloc * .c`
```

在其他环境中(例如 Windows)，则需要修改 grep 的输出，在每一行的前面加上对编辑器的调用命令，并将结果保存到一个批处理文件中，然后执行该批处理文件。我们可以使用流编辑(stream editing)执行这项任务以及许多类似的任务。流编辑就是指定一个命令序列，自动对文本流进行修改。常用于流编辑的两个工具是 sed 和 Perl。虽然 sed 专门用于流编辑，但是由于 Perl 的语法更为正规，并且具有就地编辑能力，所以在示例中，我们将使用 Perl。如果您在一个生产环境中使用流编辑，学习 sed 或许是值得的，因为 sed 对某些任务

① netbsdsrc/usr.bin/vi

来说更有效率。

最有用的编辑命令是替换命令，它的表述如下：

S/regular expression/replacement/flags

(S/正则表达式/替换内容/标志)

这个命令找出与正则表达式匹配的文本，然后将其替换为替换文本。在替换文本中，可以使用特殊表达式 $n 来表示正则表达式的第 n 次插入部分(例如，$1 将对应正则表达式的第一次插入部分)。flags 是单字符修饰符，用来改变命令的行为。最有用的标志是 g，它使得替换更改所有正则表达式出现(但不重叠)的地方，而不是仅仅更改第一个——默认的行为。如果使用 Perl 作为流编辑器，在执行 Perl，需要用-p 命令行标志(表示它应该对输入执行循环打印)，并将替换命令指定为-e 命令行标志的参数。多个替换命令可以用分号分开。还可以使用-ibackupextension 标志指定 Perl 应该就地编辑文件，而非仅仅作为输入的过滤器。更详细的信息可以参考 Perl 的文档。

回到我们前面的问题：创建批处理文件，编辑与某个模式匹配的文件时，我们需要做的就是将 grep 的输出输送到 Perl，将每一行的开始替换为 edit 命令。

```
C:\> grep -l xmalloc *.c | perl -p -e 's/^/edit /' > cont.bat
C:\> cont
```

在 Unix 类型的外壳中，可以将 Perl 的输出输送回外壳立即进行处理。

```
grep -l OBSOLETE *.c | perl -p -e 's/^/gzip /' | sh
```

这段代码压缩所有包含单词 OBSOLETE 的文件。甚至可以将该输出输送到一个外壳循环中，针对每个匹配的文件执行若干命令。例如，图 10.4 中的代码使用 grep 找出所有包含单词 xmalloc 的文件，将输出输送到 do 循环中。循环中的代码将做下面的工作：

- 将每个文件名读取为 $f 变量。
- 从 RCS 储存库中检出(check out)每个文件，并锁定它，以进行修改。
- 使用 Perl 的就地编辑特性，将所有使用 xmalloc 的地方更改为 safe_malloc。
- 将文件连同一个适当的日志信息一同检入(check in)到 RCS 储存库中。

```
grep -l xmalloc *.c *.h |                          列出包含xmalloc的文件
while read f                                       循环将每个文件名读入$f中
do
    co -l $f                                       检出文件进行编辑
    perl -p -i.bak -e 's/xmalloc/safe_malloc/g' $f 执行就地替换
    ci -m"Changed xmalloc to safe_malloc" -u $f    检入文件
done
```

图 10.4 更改 RCS 控制下的标识符名

当依照 grep 的输出，自动执行任务时，要注意，如果仅仅指定单个文件作为 grep 的参数，即使在文件中找到匹配项，grep 也不会显示文件的名称。如果您的脚本依赖于文件名的存在，那么，当在单个文件上运行时，它会失败。GNU 版本的 grep 支持一个总是显示文件名的开关-with-filename；在其他平台上，可以保护性地指定系统的空文件(在 Unix 中为/dev/null，在 Windows 中为 NUL)作为 grep 的附加参数。这个文件不会和任何模式匹配，

但会强制 grep 总是显示文件名。

和其他所有在词汇层面运作的工具一样，基于 grep 的查找不是总能精确地找出您正在查找的实体。特别地，C 预处理器所做的替换，以及跨越多行的项、注释和字符串都有可能使 grep 陷入混乱。幸运的是，大多数情况下，grep 的查找规范会产生噪音（无关的输出）而非寂静（应该出现却错过的行）。可以通过指定一个更为精确的正则表达式来消除输出噪音。例如，如果查找 malloc，自然的正则表达式描述会产生下面的输出：

```
$ grep malloc *.c *.h
[...]
misc.c: if ((p = malloc(size)) == NULL)
misc.c:         return (xmalloc(size));
misc.c: oe = xmalloc(s);
extern.h:void   *xmalloc __P((u_int));
```

为了消除不合逻辑的 xmalloc 行，我们可以在正则表达式中规定第一个字符不能是 x，即[^x]malloc。但是，请注意，这个正则表达式与 malloc 出现在行的开头这种情况不匹配，因为没有首字母与[^x]模式匹配。一种十分有效、同时不需要过多人工干预的方案是过滤 grep 的输出，消除不合逻辑的匹配。例如，查找结构字段 code 的应用会产生下面的输出：

```
$ grep code *.c *.h
[...]
process.c: *  This code is derived from software contributed to
process.c: *  1. Redistributions of source code must retain the
process.c:                          switch (cp-> code) {
process.c:                  switch(cp-> code) {
defs.h: *  This code is derived from software contributed to
defs.h: *  1. Redistributions of source code must retain the
defs.h: char code;                                  /*  Command code */
```

-v 开关指示 grep 只显示与给定正则表达式不匹配的行，即过滤掉所有与正则表达式匹配的行。要注意到，所有不合逻辑的行都是以 * 开始的块注释行，我们可以用下面的方法消除它们：

```
$ grep code *.c *.h | grep -v "^ \* "
process.c:                          switch (cp-> code) {
process.c:                  switch(cp-> code) {
defs.h: char code;                                  /* Command code */
```

使用外壳的命令行编辑特性，我们可以递增地应用这种方案，逐渐地用附加的过滤器提炼 grep 的输出，直到得出我们想要的精确输出。

有时，一项复杂和单调乏味的代码阅读任务可以通过组合诸多工具自动进行。考虑下面这项任务：找出项目中所有含下划线标识符的某个子集（例如，项目中的本地标识符）。当切换到采用混合字符大小写（例如，IdentifierFileName）来格式化标识符的编码风格时，可能就会执行此类任务。这项任务可以分为 3 步，半自动执行。

（1）创建一个候选标识符的列表。

（2）手动编辑这个列表，移除不合适的标识符（例如，库函数名和类型定义）。

(3) 将这个列表作为搜寻所有项目文件的输入。

首先,我们要找出所有含有此类标识符的行:

```
$ grep "[a-z0-9]_[a-z0-9]" *.c *.h
[...]
process.c:          size_t len;
process.c:          enum e_spflag spflag;
process.c:          size_t tlen;
process.c:          struct s_command *cp, *end;
```

然后,我们要移除无关的代码,将标识符分离出来。完成这项任务可以通过指定一个Perl匹配标识符的正则表达式,使用Perl的相关功能(表达式)将它括起来,并且,针对每一行,在将匹配的表达式替换为空后,打印出这种匹配的结果($1)。这种习惯用法可以循环打印出每个行中所有匹配的标识符。

```
$ grep "[a-z0-9]_[a-z0-9]" *.c *.h
> perl -n -e 'while (s/\b(\w+_\w+)//) {print "$1\n}'
[...]
rm_so
rm_eo
rm_eo
rm_so
rm_so
size_t
e_spflag
size_t
s_command
```

(Perl正则表达式构造\b表示单词边界,\w表示单词中的字符,+表示一个或多个匹配;所有这些只是方便的简化操作,可以使正则表达式更为简洁。)然后,我们希望使每个标识符在输出中只出现一次。一般通过对输出进行排序(使用sort程序),然后用uniq程序移除重复的行,来完成这项工作。

```
$ grep "[a-z0-9]_[a-z0-9]" *.c *.h
> perl -n -e 'while (s/\b(\w+_\w+)//) {print "$1\n}'
> sort |
> uniq > idlist
$ cat idlist
add_compunit
add_file
cmd_fmts
compile_addr
compile_ccl
compile_delimited
```

在手动编辑该列表,移除项目作用域以外的标识符(例如,C函数库的成员)之后,我们就有了一份以新行符分隔的标识符列表,后面我们将在项目文件中查找列表中的这些标识

符。我们可以用 fgrep——grep 的亲戚——来完成这项任务。fgrep 只在文件中查找固定的字符串(而非正则表达式),但可以接受新行符分隔的固定字符串列表作为参数,进行匹配。fgrep 使用的算法专为执行这种查找进行了优化;字符串列表可以轻易地包含数百个元素。因此,fgrep 对在源代码中查找预先计算出的固定字符串列表十分有用。在本例中,我们已经将手动制作的标识符列表保存到 idlist 文件中,现在可以在所有的文件中查找这些标识符了。

```
$ fgrep -f idlist *.c
compile.c          struct   labhash *lh_next;
compile.c:         u_int    lh_hash;
compile.c:         struct   s_command *lh_cmd;
compile.c:         int      lh_ref;
[...]
```

当项目的源文件存储在多个目录中时,可以采用许多不同的方案。在简单的两级目录结构中进行查找的一种方案是,用合适的文件通配符模式运行 grep①。

```
$ grep isblank */*.c
ex/ex_write.c:              for (++p; *p && isblank(*p); ++p);
ex/ex_write.c:              for (p += 2; *p && isblank(*p); ++p);
vi/getc.c:        if (csp->cs_flags != 0 || !isblank(csp->cs_ch))
```

另外,某些外壳(比如 zsh)允许使用下面的模式指定执行递归查找:

```
**/*.c
```

对于比较深的、结构复杂的目录层次,最好的方案是使用 find 命令创建一个希望查找的文件列表,并将输出输送到 xargs 命令,同时将 grep 和相关的正则表达式指定为 xargs 的参数②。

```
$ find . -name '*.c' -print | xargs grep 'rmdir('
./nfs/nfs_serv.c:nfsrv_rmdir(nfsd, slp, procp, mrq)
./ufs/ext2fs/ext2fs_vnops.c:ext2fs_rmdir(v)
./ufs/lfs/lfs_vnops.c:lfs_rmdir(v)
./ufs/lfs/lfs_vnops.c:        ret = ufs_rmdir(ap);
./ufs/ufs/ufs_vnops.c:ufs_rmdir(v)
[...]
```

find 命令遍历指定的目录层次,打印出满足给定条件的文件(在此,这个条件是,文件的名称必须与模式 *.c 匹配)。xargs 命令的参数是一个命令名和一系列初始参数,并从标准输入读入附加参数,然后使用指定的初始参数后跟从标准输入读入的参数,单次或多次执行指定的命令。在我们的例子中,xargs 运行 grep,将 find 的输出作为 grep 的文件参数,即在当前目录下找到的所有.c 文件。使用这种方案,grep 可以处理任意数量的文件,不会受到系统中命令行参数大小的限制。如果您所在的基于 Windows 的环境不支持上面的结构,那么只

① netbsdsrc/usr.bin/vi

② netbsdsrc/sys

好动态地构造一个临时批处理文件，其中包含所需的命令。

最后，我们给出与3个有用的grep命令行开关相关的建议，结束对grep风格的工具的讨论。

- 遍历所有的注释，以及用标识符大小写不敏感的语言（例如，Basic）编写的代码时，可以使用大小写不敏感的模式匹配（grep i）。
- grep会将短划线开头的正则表达式解释为开关选项。要避免这种问题，可以使用-e开关指定这个特殊的正则表达式。
- 使用grep-n命令行开关，可以创建与给定正则表达式匹配的文件和行号的检查表。

在希望使用grep查找体积十分庞大的数据时，可能必须研究替代的方案。我们在10.2节中讨论了为流行编辑器创建标记（tag）数据库的一些工作。另外，如果您首先关心的不是源代码编辑与浏览，那么可以考虑使用一种通用的索引工具，如Glimpse[①]。

练习10.8 程序员经常使用特殊的标记，比如XXX或FIXME，标识需要格外注意的代码。在本书的源代码树中查找、统计并分析这类实例。

练习10.9 如果要找出下划线分隔标识符，可能会涉及手动编辑输出列表，移除其中属于系统库的标识符（例如，size_t）。建议一种自动执行这项任务的方式，请注意，系统库中使用的标识符都声明在您能够处理的包含文件中。

练习10.10 本书Unix源代码中所有的系统调用声明都来自一个单一文件。[②] 在源代码树中，找出前20个系统调用的定义。尽可能地自动化这项任务。

练习10.11 编写一个外壳脚本或批处理文件，使用grep、正则表达式和其他工具，列出一系列源文件中违背编码规范之处。记录产生这种违背的原因，以及能够预期的虚假噪音的类型。

10.4 找出文件的差异

重用代码的各种方式中，一种较为简单的方式是创建相应代码的副本，并按照需要对它进行修改。这种方式的代码重用存在许多问题；其中之一就是要创建代码库的两个分支版本。另外的情况是，在您分析代码体的演化时，一个源代码文件存在两种不同的版本。所有情况下，最终都会得出同一源代码文件的两个稍有差异的版本。比较它们的一种方式是将它们都打印在折叠纸上，并排放置这两份代码清单，查找其中的差异。更为有效的一种方式是使用工具。diff程序能够比较两个不同的文件或目录，列出要让文件一致必须加以修改的那些行。

① *http://webglimpse.org/*

② netbsdsrc/sys/kern/syscalls.master

diff 工具能够以各种方式输出文件的差异。有些格式很简洁，专门为其他程序，比如 ed，RCS，或 CVS 进行了优化。其中一种格式，上下文差异，十分用户友好；它在显示差异的同时还显示差异周围的代码。diff 的-g 选项用来指定这种输出格式；其他工具，如 CVS 和 RCS(参见 6.5 节)，在比较源代码的不同版本时，也支持这个选项。两个文件中不同的行用！来标记，新增的行标记为＋，能够推断出的删除行用-。图 10.5 是 diff 输出的一个例子。被比较的两个文件是特定网卡驱动程序代码的不同实例；一个文件是操作系统网络接口的一部分，①另一个用于通过网络独立启动。②

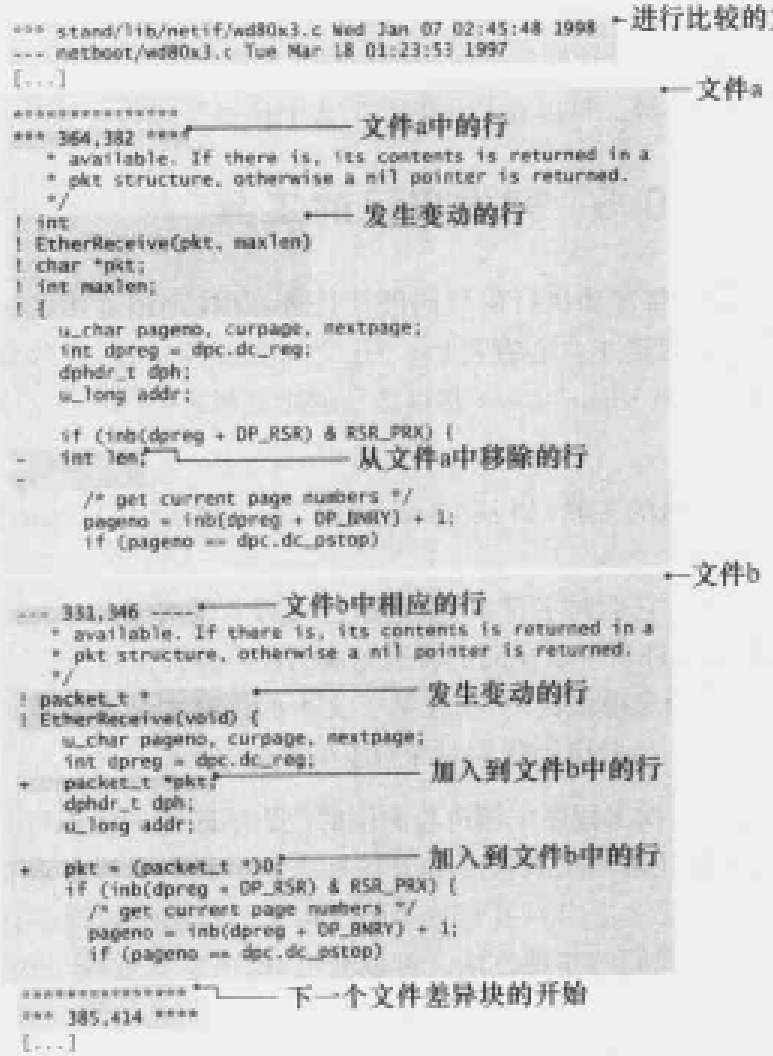


图 10.5　文件比较：diff -c 命令的输出

diff 程序在词汇级别工作，一行一行地比较文件。因此，它能够比较任何编程语言编写的文件，只要该语言可以将源代码保存为纯文本的方式。有时，一些聪明的程序编辑器和其他类似工具会修改程序的格式，从而引起 diff 产生无关的输出。可以使用 diff 的-b 标志，使文件比较算法忽略结尾的空格，-w 标志忽略所有空白区域的差异，-i 标志使文件比较大小写不敏感。

① netbsdsrc/sys/arch/i386/stand/lib/netif/wd80x3.c:364-382

② netbsdsrc/sys/arch/i386/netboot/wd80x3.c:331-346

显示文件间的差异是少数在图形环境下执行更有效率的编程任务之一。Microsoft Visual Studio 开发环境包含一个名为 windiff 的程序，它以图形的方式显示文件或目录的差异。类似的程序(xxdiff，xdiff，gtkdiff)运行在 X Window 系统中。

练习 10.12 建议一种简单的方式，计算两个不同文件的“相似度(measure of similarity)”。将这种量度应用到源代码集合中同名的文件上，并创建可能得益于结构化代码重用的文件列表。

练习 10.13 许多工具支持对文件修订的有序控制。如果您正在使用这类工具，分析如何显示不同文件版本之间的差异。可以在相关的上下文中显示不同的行吗？

10.5 开发自己的工具

有时，很平常并且明显能够自动执行的代码阅读任务，却找不出适用的工具来进行处理。不要对创建自己的代码阅读工具心存畏惧。Unix 外壳及其工具，现代解释型编程语言，比如 Perl，Python，Ruby，和 Visual Basic 都很适合创建定制的代码阅读工具。每种环境都有其自身的特长。

- Unix 外壳，它提供大量的工具，如 sed，awk，sort，uniq 和 diff，可以通过累加组合获取所需的功能。
- Perl，Python 和 Ruby，它们对字符串、正则表达式和关联数组(associative array)的支持都很强大，简化了许多可以在词汇级别完成的代码分析工作。
- Visual Basic，它提供对象模型代码，这在基于文本的工具中是不常见的。相关的例子包括 Microsoft Access 验证函数和对深对象层次的遍历。

考虑一个特定的例子——找出程序中缩进与周围的代码不匹配的代码行。这类代码会误导读者(最好的情况)，并且经常是重大程序 bug 的标志。当我们希望在现有的源代码库中查找这类代码的例子时(参见 2.3 节)，我们找不到任何工具能够完成这类任务，所以，我们只好自己动手。图 10.6 是我们的实现方法。可以看出，这个工具依赖于词汇试探法，并且对代码做了大量的假定。它首先找出不以开花括号结尾的 if，while 和 for 语句，定位后面相邻的两个缩进行，据此检测可疑代码。这个工具不能处理注释或字符串中的保留字和标记，跨越多行的语句，以及使用空格而非制表符的情况。然而，它的 23 行代码是在不到一个小时的时间里写成的。在这期间我们逐渐改进最初的想法，直至输出的信噪比(signal to noise rate)能够接受。(之前的一个版本，甚至不能处理首行出现新花括号代码块的情况——这种情况很常见)。我们实现缩进校验工具的方式也适用于类似工具的开发。

Cunningham[Cun01]介绍了他如何编写两个 40 行的 CGI Perl 脚本来归纳与解说大型的 Java 源代码集合。两个脚本都创建 HTML 输出，所以可以使用 Web 浏览器浏览兆字节级别的源代码集合。图 10.7 中是归纳工具的一个输出①。归纳脚本将每个 Java 方法浓缩

① jt4/catalina/src/share/org/apache/catalina

```
#!/usr/bin/perl
use File::Find;                                    ← 处理指定树中的所有文件
find(\&process, $ARGV[0]);

sub process
{
    return unless -f;                              ← 只处理C文件
    return unless (/\.c$/i);
    open(IN, $fname = $_) || die "Unable to open $_:$!\n";   ← 针对每行源代码
    while (<IN>) {
        chop;
        if (/^(\t+)(if|for|while)/ && !/\{/) {     ← 这是没有花括号的if/for/while吗?
            $tab = $1;
            $n = <IN>;                             ← 取出下两行代码
            $n1 = <IN>;
                                                   ← 它们是以分号(;)终结,以附加
            if ($n =~ m/^$tab\t.*;/ &&                 tab开始的纯语句吗?
                $n1 =~ m/^$tab\t.*;/ &&
                $n !~ m/\t(if|for|while|switch)/) {

                print "$File::Find::name\n$_\n$n$n1\n";   ← 我们发现了问题,
            }                                              打印文件的位置与代码行
        }
    }
}
```

图 10.6　定位错误缩进的代码块

为单独的行,每个行由方法定义体中出现的 {};" 字符组成。方法的签名(signature)可以揭示大量的信息:从行的长度和花括号的安排,很容易得出方法的大小和结构。超链接允许我们定位到每个特定的方法,对它进行详细的分析,但实际上,这个工具的长处在于,它能够简明地将巨量代码集形象化。

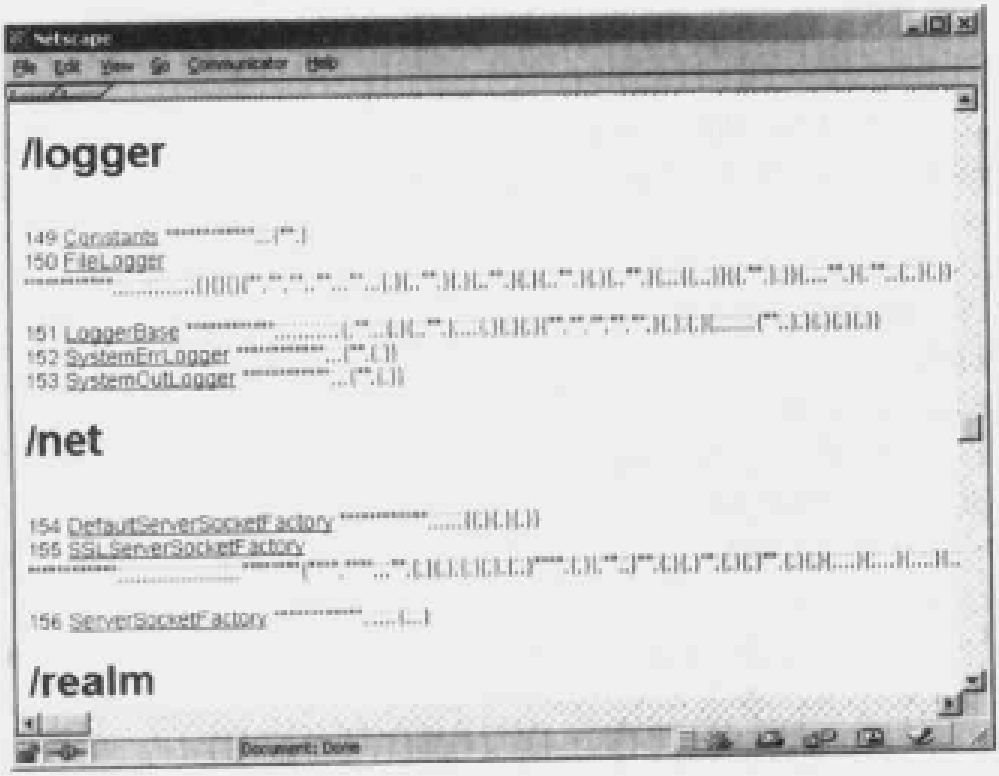


图 10.7　Java 代码的签名调查

在构建工具时应该遵循的准则如下:

- 要充分利用现代快速原型语言(rapid-prototyping language)所提供的能力。

- 从简单开始，根据需要逐渐改进。
- 使用利用代码词汇结构的各种试探法。
- 要允许一些输出噪音或寂静(无关输出或缺失输出)，但在使用该工具时要考虑到这种噪音和寂静。
- 使用其他工具对输入进行预处理，或者对输出进行后期处理。

练习 10.14 确定哪些代码阅读任务可以得益于定制工具的应用。简要地描述如何构建这类工具。实现您描述的某个工具。

练习 10.15 优美打印程序(pretty-printer)以易读的方式对源代码进行排版(参见10.7节)。一般地，用不同的字体或字形来表示注释、变量和保留字。选定一种语言，基于词汇试探法实现一个简单的优美打印程序。可以使用 Postscript 或调用其他程序对输出进行排版：在 Unix 中可以输出 LaTeX 或 troff 命令，在 Windows 中可以输出 RTF(rich text format)，或使用 OLE 自动化调用 Microsoft Word。

练习 10.16 编写一个工具，对源代码文件进行索引，简化对源代码的浏览。索引项可以是函数定义或声明。该索引可以作为按字母顺序排序的列表，或是含有超链接的 HTML 页。

10.6 用编译器来协助代码阅读

对您的代码进行精确分析的工具就是代码的编译器。编译器不仅仅能够完成源代码到目标代码的转换；还可以对程序进行不同级别的分析。下面是利用编译器获取程序信息的6种方式。

(1)生成警告消息。
(2)抓出生成错误消息的代码。
(3) 生成程序清单。
(4) 获取预处理输出。
(5) 分析生成的符号(汇编)代码。
(6) 完成最终的目标代码。

这些方式并不适用于所有的编译器和所有的语言，但其中的一些可能有助于您更好地理解正在处理的代码。

许多合法的程序构造能够生成可疑或明显错误的程序行为。编译器能够检测出其中一些问题，并标记警告消息。相关的例子包括：

- 行为依底层构架的不同而不同的表达式，可能导致移植问题；
- 类型不同，但相互兼容的隐式类型转换和变换；
- 在 if 或 while 语句中本应是条件表达式的地方使用了常量；
- 函数应该返回一个值却并未返回；
- 变量声明与其他变量同名；

- 缺失枚举成员或 switch 语句中的无关值；
- 未知的 # pragma 值；
- 没有引用的变量、结构成员或标记；
- 没有初始化用 const 声明的对象。

许多编译器提供一个选项来增加和降低警告的严重性，从而决定是否只是显示该警告，还是将警告当作是错误。在阅读代码时，要使编译器成为您的朋友：指定恰当级别的编译器警告，并小心地评估生成的结果。如果您对代码体拥有控制权，您的目标应该是使该程序在编译中不会出现任何警告。选择与局部编码准则，或您想要达到的代码品质相匹配的编译器警告级别，纠正或修改可疑的构造，直到所有的警告消息都被消除为止。有时，这些警告消息的生成交由另外的专门工作来完成。我们将许多这类工具都通称为 lint，lint 最初是 Steve Johnson 编写的 Unix 程序，目的是从代码中移除残余的“纤维屑(lint)”。

我们还可以使用编译器定位出所有使用变量、方法、类型或类的地方。只需要将要查找的相应声明，或者标识符定义重新命名，并且执行完全编译。指出给定标识符现在没有定义的错误，就标明了使用该标识符的位置。然而，要注意，有些编译器对每个未定义标识符只产生一条错误消息，在允许重载函数定义的语言中，需要重命名给定标识符所有的定义和声明。我们还可以使用类似的策略检测与给定标识符链接的文件。这种情况下，保持标识符声明，但移除其定义。所有依赖于给定标识符的对象文件，在程序的链接阶段，都会报告未解决的引用(unresolved reference)。

一些编译器——主要是那些在不受 Unix 环境所影响的文化中开发而成的编译器，提供选项，可以生成被编译代码的详细清单。这些清单常常伴随变量和函数定义的索引、编译信息和个别代码块的汇总。在不能使用 grep 风格的工具(10.3 节)，也没有具有代码浏览功能(10.7 节)的集成开发环境时，可以使用这些索引来浏览大型的代码体。例如，图 10.8 中就是由 Microsoft Macro Assembler 生成的符号清单。

C 编程语言极度依赖于 C 预处理器对众多特性的支持，比如分开编译和条件编译、常量和内联代码的定义、以及对注释的处理。C 预处理器还经常用来(或滥用)创建附加的控制结构、提供面向对象的结构、以及支持基于模板(泛型)的编程。由于能否正确地应用这些非标准的语言特性现，完全依赖于程序作者的个人能力，最终的程序可能会难以读懂。在 International Obfuscated C Code(IOCCC)中，许多胜出的程序都大量应用 C 预处理器。您的 C 编译器能够帮助您回击这类程序。大多数编译器有一个选项(常常为-E)，可以将程序的源代码传递给预处理器，然后输出生成的代码。之后，您就能够阅读代码，理解宏如何扩展，或者找出为什么库包含文件中用 # define 定义的一个常量，用在看似合法的代码中会产生一个晦涩的错误消息。要注意，浏览预处理后的代码可能会比较困难，包含的头文件，扩展后的宏和常量，还有注释的移除都会使得阅读工作十分费力。最好的策略是瞄准某个惟一的 C 标识符(变量、标记或函数名)——您希望分析的位置就在这个标识符后而，然后使用编辑器在预处理后的代码中查找这个标识符。

要彻底地了解编译器如何处理特定的代码块，需要查看生成的符号(汇编)代码。虽然可以将生成的代码进行反编译来做到这一点，但通过相关的选项，用编译器生成的代码更易于理解。大多数编译器生成符号代码的原因有两条，要么作为目标代码生成之前的中间步骤，要么作为一种调试辅助手段，帮助程序员和开发编译器的人员。Unix 编译器的-S 选项

```
Microsoft (R) Macro Assembler Version 5.10B        11/2/2
winclip.c                                          Symbols-1
Symbols:

                N a m e                 Type     Value   Attr

[...]
_argc$ . . . . . . . . . . . . .        NUMBER   0008
_argv$ . . . . . . . . . . . . .        NUMBER   000C
[...]
_error . . . . . . . . . . . . .        N PROC   091D    _TEXT   Length = 0065
_exit  . . . . . . . . . . . . .        L NEAR   0000     External
_fileno  . . . . . . . . . . . .        L NEAR   0000     External
_fname$  . . . . . . . . . . . .        NUMBER   -0004
_fname$22039 . . . . . . . . . .        NUMBER   -1030
_fopen . . . . . . . . . . . . .        L NEAR   0000     External
_fprintf . . . . . . . . . . . .        L NEAR   0000     External
_fread . . . . . . . . . . . . .        L NEAR   0000     External
_fwprintf  . . . . . . . . . . .        L NEAR   0000     External
```

图 10.8 由 Microsoft Macro Assembler 生成的符号清单

可以用于这种目的；Visual C/C++中对应的开关是/Fa。阅读生成的符号代码来了解一段代码是一项极端的措施。一般用于下面这些情况：

- 了解具体的编译器如何完成定义的操作(例如，类型转换)；
- 查看编译器优化如何影响生成的代码；
- 使自己信服，编译器确实正确地转换了某段代码，或者查找编译器的 bug；
- 检验用 C 语言编写的与硬件相关的代码，是否执行了预期的操作；
- 学习如何用符号代码来编写某种特定的构造。

将编译器用作代码阅读工作的终极方式是将程序编译成目标代码。在这一层面，您能够获取的有用信息主要是定义在每个文件中的公开(extern)和私有(static)符号。许多编程环境提供工具来分析目标文件中定义、输出和使用的符号(Unix 中为 nm，Windows 中为 dumpbin)。通过在编译后的文件上运行这类工具，您可以找出特定的符号在什么地方定义，以及哪些文件中实际使用了它。图 10.9 和图 10.10 列出了代表性的符号定义，它们都来自于编译 printf 命令生成的目标文件①。和 grep 类型的搜索产生的杂乱含糊的结果相比，通过分析目标文件获取的结果极端精确。编译过程保证考虑了所有头文件、条件编译、宏定义和编译器优化的影响，并反映到文件的输入和输出列表中。这个过程还保证您不会受到字符串、注释、标记或枚举中出现的标识符名称的干扰。但是，请注意，内联函数在某种情况下会完全从目标文件中省略。

大多数 C++编译器为了支持函数重载和跨文件类型检查，会应用标识符名称改编(name mangling)。这些编译器，向外部可见的标识符(一般是方法名或函数名)添加编码了相应类型信息的字符序列，并整体重写该标识符。例如，Microsoft C/C++编译器将会把

① netbsdsrc/usr. bin/printf/printf. c

图 10.9　目标代码中的符号(Unix nm)

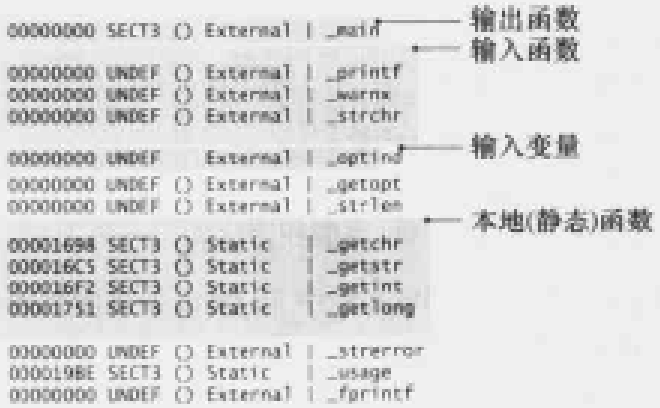


图 10.10　目标代码中的符号(Windows dumpbin)

MenuItem 类的公开方法 setItem[①] 编码为? setItemMenuItemQAEHPBDHZ,而 GNU C 编译器中会将其编译为_setItem__8MenuItemPCci。大多数情况下,您依旧可以读到函数的名称,不过您需要抛弃修饰性的类型信息,或对它们进行解码。

练习 10.17　提供可以产生本节讨论的某些编译器警告的代码样本。您的编译器能够检测出它们吗? 使用这样的代码是否有合理的理由?

练习 10.18　C 编译器生成的许多警告信息,在其他强类型(strongly typed)语言中被看作是错误。列出您使用的 C 编译器生成的警告,标出那些代码完全合法,但编译器却给出警告的地方。看看其他语言能够检测出哪些警告。

练习 10.19　建议并编制处理警告的方针。这项方针如何影响代码的易读性? 考虑什么情况下需要引入额外的代码,抑制特定的编译器警告。

练习 10.20　用 C 预处理器处理某些 IOCCC 优胜程序[②],试着理解程序如何运作,以及最初的模糊化是如何完成的。

① qtchat/src/0.9.7/Tree/Menu.cc:23-33

② *http://www.ioccc.org*

练习 10.21 编译一个程序(禁止编译器优化),阅读生成的符号代码。开启所有优化重复这个过程。描述每种情况下参数如何在函数间传递。

练习 10.22 编写一个工具,给定一系列目标文件,显示每个文件中可能定义为 static 的符号。提示:您不需要分析目标文件;使用现有的工具来完成这项工作。

10.7 代码浏览器和美化器

许多工具就是为了浏览代码而设计的。浏览器所提供的典型功能包括显示下面这些内容:

- 定义:找出实体在什么地方定义;
- 引用:找出所有使用某个实体的地点;
- 调用图:列出给定函数调用的所有函数;
- 调用者图:列出调用给定函数的所有函数;
- 文件大纲:列出文件中的函数、变量、类型和宏定义。

面向对象语言的浏览器,比如 Smalltalk,C++和 Java,提供附加的功能,可以对类进行分析。给定一个类名,您能够获得:

- 类在何处定义;
- 引用类的地方;
- 哪些类派生自这个类;
- 这个类的基类;
- 它的公开、受保护和私有的方法和字段。

图 10.11 呈现了 groff[①] node 类的派生类,并详细列出了 ligature_node 类的成员、属性、定义和引用。

我们在下面的段落中介绍一些广为人知的开放源码浏览器。

- cscope 工具[②]。它是一个基于 Unix 的字符界面源代码浏览器。它的设计目的是对 C 进行分析,但由于它的分析器很灵活,故而也可以应用到 C++和 Java 代码中。cscope 可以和大量的编辑器集成,如 Emacs,nvi 和 vim。
- cbrowser 工具[③]。它是一个图形化的 C/C++源代码查找与浏览工具,并能用来查看函数的调用层次。它建立在 cscope 之上。
- Source-Navigator[④]。它提供对大量语言的浏览能力,包括 C,C++,Java,Tcl,FORTRAN 和 COBOL。

① *http://www.gnu.org/software/groff/*
② *http://cscope.sourceforge.net/*
③ *http://cbrowser.sourceforge.net*
④ *http://sources.redhat.com/sourcenav/*

- LXR 浏览器[①](图 10.12)。它提供大型代码集合的基于 Web 的交叉引用。

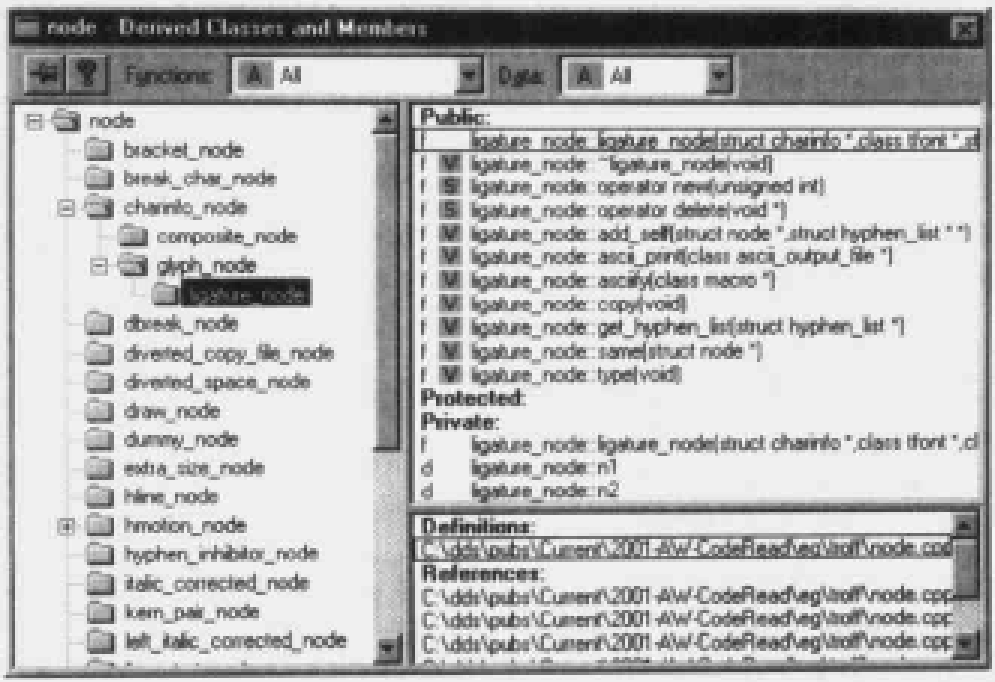


图 10.11　Microsoft Visual Studio 源代码浏览器中的 troff 类

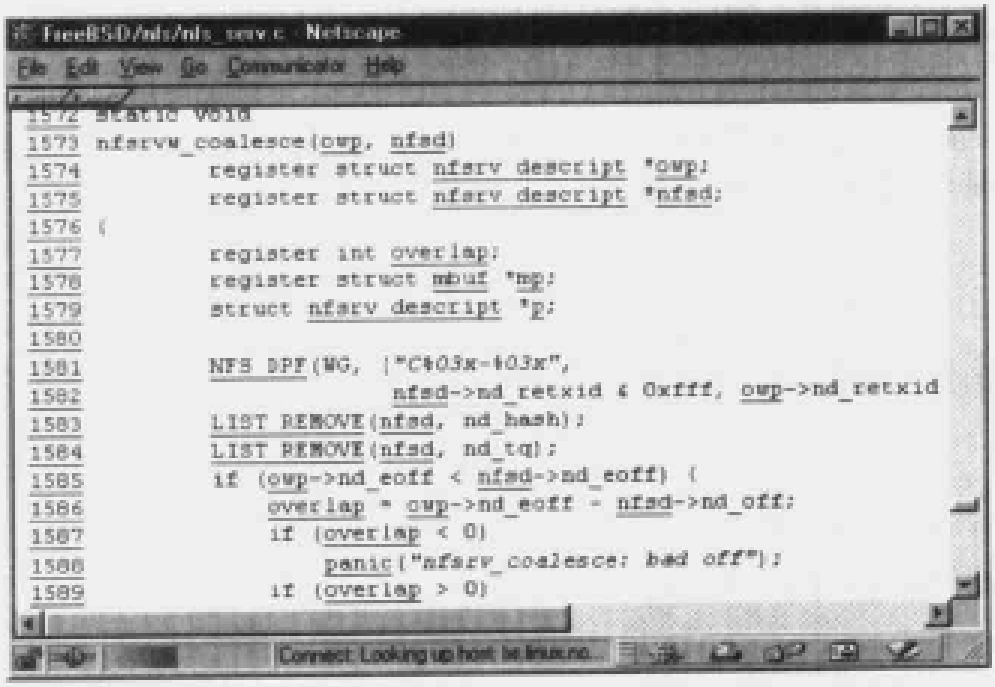


图 10.12　基于 Web 的源代码浏览器 LXR 的样本输出

另外，大多数 IDE(集成开发环境)也提供某些浏览功能。在处理大型多文件项目时，可以将它作为项目载入到集成开发环境中(即使您不想使用 IDE 来进行开发)，利用集成开发环境提供的源代码浏览功能。

① *http://lxr.linux.no/*

如果说源代码浏览器可以帮助您航行于大量源代码之中的话，那么美化器和优美打印程序可以帮助您理清代码的细节。美化器，如 Unix 程序 cb(美化 C 语言源代码)和 indent，按照具体的代码编排约定对源代码文件进行重新编排。这些程序能够处理缩进，花括号的布局，运算符和关键字周围的空格，声明和换行符。您可以通过设定命令行选项以及创建配置文件，使程序的行为遵循本地的编排约定。要抵制住按照您的编码规范对外部代码进行美化的诱惑；不必要的编排更改会创建不同的代码，并妨碍维护工作的组织。现今，大部分代码都是由出色的专业技术人员编写，并遵循一系列的编排约定。美化器只会机械地执行，常常会破坏代码中精心安排的格式，而这些格式会为理解代码提供附加的暗示。然而，在某些情况下，美化器还是能够起到它的作用。使用美化器可以：

- 修复那些在编写过程中，确实没有一致地遵循任何格式规范的代码；
- 采用那些无人维护的代码；
- 创建代码的临时版本，协助对代码的解读；
- 集成同一大型项目中的代码。

优美打印程序可以对程序的源代码进行整洁的排版，使它更易读。一种常见的风格(由 Unix vgrind 程序使用)是将注释以斜体显示，将关键字设为粗体，并将当前函数的函数名称列在每页右边的空白中。图 10.13 是用 vgrind 对 rmdir 程序[①]进行排版后的部分结果。

在线阅读代码时，可以使用许多现代编辑器和集成开发环境提供的语法着色特性，将不同的语言元素突出显示。要注意，编辑器和优美打印程序，常常允许您指定某种语言语法中的关键元素，从而可以将它们应用到任何语言中。

有时，您可能需要采用更为特殊的工具。C 声明就是一种常见的情况：前缀运算符和后缀运算符的组合使用，可能会使得某些声明令人畏惧和费解。开放源码 cdecl[②] 程序可以将这类声明翻译成纯英语。考虑下面的定义[③]：

```
int (*elf_probe_funcs[])() = {
    [...]
};
```

通过简单的复制-粘贴操作，将上面的定义粘贴到 cdecl 中，我们立即可以得到一份清楚的解释。

```
cdecl> explain int (*elf_probe_funcs[])()
declare elf_probe_funcs as array of pointer to function returning int
```

您还可以将同样的程序(C++decl)应用到 C++类型上。[④]

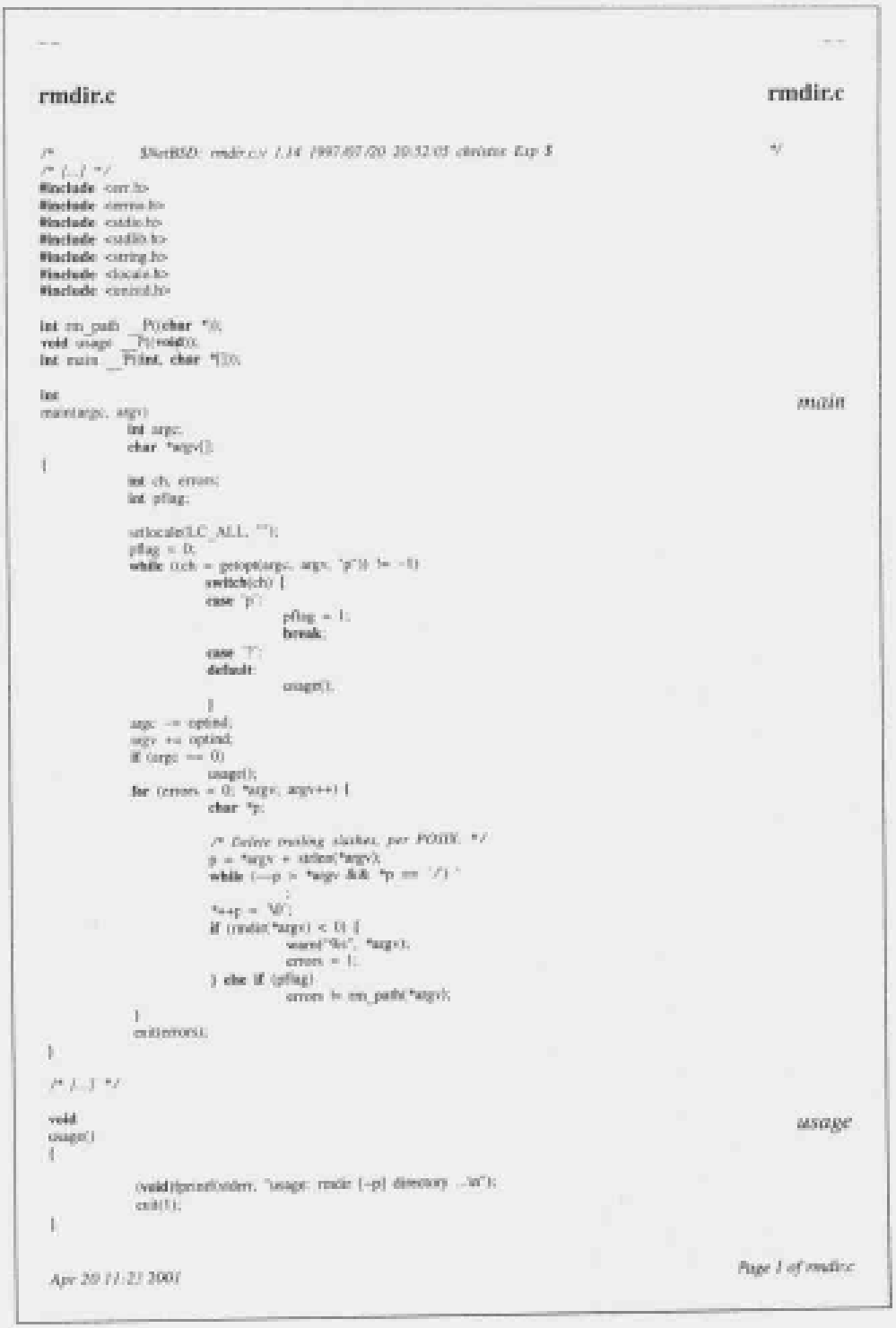
```
C++decl> explain const char * (Drwho_Node::* get_name)(void)
declare get_name as pointer to member of class Drwho_Node
function (void) returning pointer to const char
```

① netbsdsrc/bin/rmdir/rmdir.c:1-134

② *ftp://metalab.unc.edu/pub/linux/devel/lang/c/cdecl-2.5.tar.gz*

③ netbsdsrc/sys/arch/mips/mips/elf.c:62-69

④ ace/apps/drwho/PMC_Ruser.cpp:119



rmdir.c　　rmdir.c

```
/*      $NetBSD: rmdir.c,v 1.14 1997/07/20 20:52:05 christos Exp $      */
/* [...] */
#include <err.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>
#include <unistd.h>

int rm_path __P((char *));
void usage __P((void));
int main __P((int, char *[]));

int                                                              rmdir
main(argc, argv)
        int argc;
        char *argv[];
{
        int ch, errors;
        int pflag;

        setlocale(LC_ALL, "");
        pflag = 0;
        while ((ch = getopt(argc, argv, "p")) != -1)
                switch(ch) {
                case 'p':
                        pflag = 1;
                        break;
                case '?':
                default:
                        usage();
                }
        argc -= optind;
        argv += optind;
        if (argc < 1)
                usage();
        for (errors = 0; *argv; argv++) {
                char *p;

                /* Delete trailing slashes, per POSIX. */
                p = *argv + strlen(*argv);
                while (--p > *argv && *p == '/')
                        ;
                *++p = '\0';
                if (rmdir(*argv) < 0) {
                        warn("%s", *argv);
                        errors = 1;
                } else if (pflag)
                        errors |= rm_path(*argv);
        }
        exit(errors);
}

/* [...] */

void                                                             usage
usage()
{

        (void)fprintf(stderr, "usage: rmdir [-p] directory ...\n");
        exit(1);
}
```

Apr 20 11:23 2001　　Page 1 of rmdir.c

图 10.13　用 vgrind 排版 rmdir 的源代码

练习 10.23　在本书源代码库中的 Unix 内核上试用您的 IDE 源码浏览器①。列出并评论您遇到的困难。

练习 10.24　您的 IDE 源码浏览器能够与其他工具协作吗？例如，您能够查找或处理

① netbsdsrc/sys

您的浏览器列出的结果吗？您能够让浏览器自动生成报告，列出您定义的每个函数吗？建议应该怎样设计源码浏览器，使之支持这种灵活性。

练习 10.25 编写一个工具，读取 C 源文件，演绎出它的编排约定，为 indent 生成相应的开关。

练习 10.26 使用优美打印程序编排某个示例文件。

练习 10.27 检查您的编辑器是否能够让用户定义语法的颜色。选择一种不为该编辑器支持的语言，定义语法着色规则。

练习 10.28 将 cdecl 集成到您喜爱的 IDE 中。

10.8 运行期间的工具

实际运行程序，往往可以更深刻地理解程序的运作。尤其是那些缺乏足够文档，或文档已经过时的程序。在这种情况下，可以不去一行一行地理解程序的代码，而是用测试数据运行该程序，观察它的外部行为。

为了获取更为详尽的情况，我们可以分析程序如何与操作系统进行互操作。由于操作系统控制着程序的所有资源，观察这种交互能够更好地理解程序的功能。许多操作系统平台提供相关的工具，可以监控和显示程序对操作系统的所有调用。这类工具包括 MS-DOS 平台的 trace(图 10.14)，Microsoft Windows 平台的 API spy(图 10.15)，以及 Unix 平台的 strace(图 10.16)。系统调用日志将会包含程序打开的所有文件，对应的输入输出数据，系统调用的返回值(包括错误)，对其他进程的执行，信号处理，文件系统操作和对系统数据库的访问。由于勿需事先准备就能够监控程序的系统调用，因此，即使访问不到源代码，也可以对程序执行这个过程。

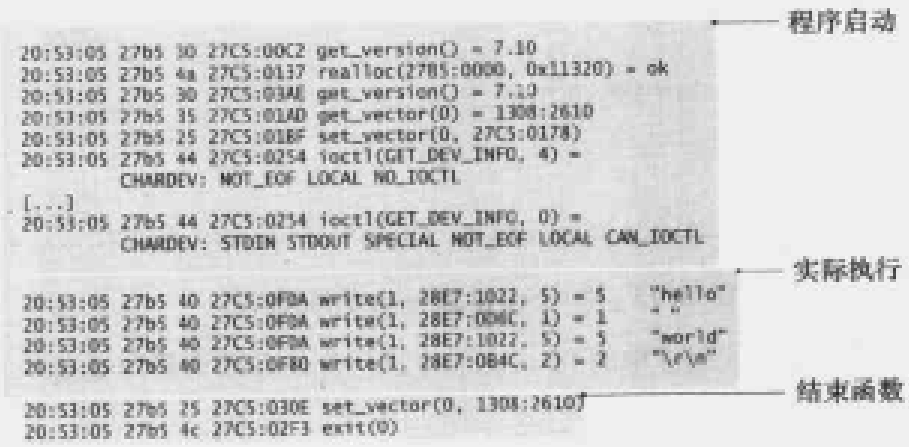


图 10.14 MS-DOS 下 trace 程序的输出

操作系统边界并非惟一可以接入并监控程序运作的地方。其他两个地方是网络接口和用户界面。许多程序，如 tcpdump 和 WinDump，可以监控和显示网络的数据包。限定它们

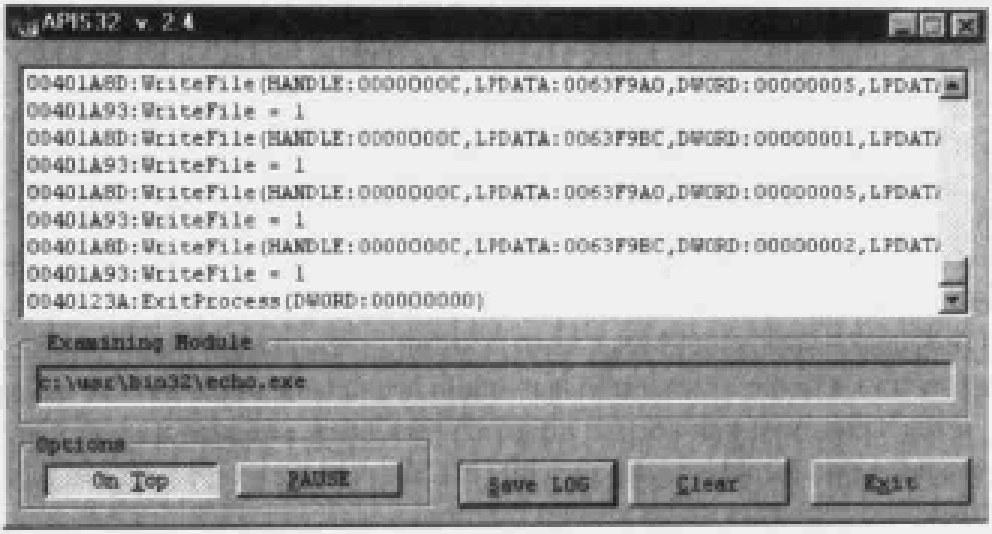


图 10.15　Microsoft Windows API Spy 程序

```
execve("/bin/echo", ["echo", "hello", "world"], [/* 22 vars */]) = 0        ← 程序启动
brk(0)                                  = 0x804a668
[...]
getpid()                                = 652
brk(0)                                  = 0x804a668
brk(0x804a808)                          = 0x804a808
[...]
open("/usr/share/locale/en_US/LC_MESSAGES/SYS_LC_MESSAGES", O_RDONLY) = 4
fstat(4, {st_mode=S_IFREG|0644, st_size=44, ...}) = 0
mmap(0, 44, PROT_READ, MAP_PRIVATE, 4, 0) = 0x40013000
close(4)                                = 0
[...]
write(1, "hello world\n", 12)           = 12       ← 实际执行
munmap(0x40017000, 4096)                = 0        ← 程序结束
_exit(0)                                = ?
```

图 10.16　Linux 下 strace 程序的输出

只监控给定的主机和 TCP 端口对，就能够获取程序网络操作的精确描述。通过这种方法，我们可以在程序运行时观察通信协议，而非努力从源代码中推演它的功能。另一类程序允许您根据用户界面对程序进行分析。X Window 系统的 Xev，和 Microsoft Windows 的 Spy++可以监控窗口接收到的事件，并将它们以可读的形式显示出来。基于窗口交互的程序常常围绕一个事件循环进行编写；我们可以通过观察给定动作产生的事件，匹配特定的用户动作与底层的代码。

使用执行剖析器(execution profiler)可以得到不同级别的细节。这项功能，或者集成在诸如 Microsoft Visual Studio 等 IDE 中，或在 Unix 中以 gprof 工具的形式提供，它可以监控程序的执行，并提供相应的剖析信息。剖析信息一般列出程序花在每个函数(以及可选的子函数)上的时间。大多数执行剖析器还能够得出程序的动态调用图：函数在运行期间对其他函数的调用。得出花在每个例程上的时间之后，我们就能够快速地查出能够得益于优化的代码。通过使用调用图，我们可以获取程序的结构图，并了解各个例程之间如何交互。如果测试数据足够详尽，我们甚至可以找出死代码(dead code)：从未执行的程序代码。要激活剖析数据的收集，一般需要用适当的编译器选项重新编译代码，并与特殊的库链接。程序结束后，剖析数据会以原始格式保存到文件中。可以用报告生成器对这些数据进行处理，生成最

终的剖析报告。

要得到更细粒度的执行剖析信息，可以利用行计数(line counting)和基本块覆盖分析(basic block coverage analysis)等技术。在此，编译期间的开关(gcc 中，-a 和-pg，-g 组合使用)或目标代码的后处理过程(在 Microsoft Visual Studio 中为 prep)可以创建计算每个基本块执行次数的代码。基本块(basic block)是只拥有一个入口和一个出口的指令序列；因此，通过计算每个基本块执行的次数，可以将结果扩展到单独的代码行。通过检查从未执行的代码行，可以找出测试覆盖的弱点，并据此修正测试数据。通过检查每个代码行执行的次数，我们可以揭示算法是如何运行的。

考虑图 10.17 中的输出。这段输出是针对 wc[①](字符、单词和行计数程序)的源代码运行该程序，由 gprof 生成的清单。gprof 选项-l，-A 和-x 用来生成指定的输出。源代码由 269 个行，968 个单词，以及 6149 个字符构成。这些数字分别反映在特定源代码行的执行次数(图 10.17:3，图 10.17:4，图 10.17:2)。需要注意的是循环的评注方式：在行的开头列出的 3 个数字分别表示进入循环的次数、循环分支返回开始处的次数、以及循环退出的次数。请注意，while 循环通过分支返回到开始处的次数(3)，如何反映在整个循环体执行的次数(2)。从未执行过的代码行也特殊标记出来(图 10.17:5)，可以据此得出特定测试输入下，没有得到执行的代码。在编写本书的时候，由 gcc 编译的代码生成的基本块剖析输出，尚且与 gprof 所期望的数据不兼容：一个小型的 Perl 脚本(bbconv.pl，gprof 发行程序中提供)可以将数据转换成所需的格式。

```
    1 -> gotsp = 1;
1,3,1 -> while ((len = read(fd, buf, MAXBSIZE)) > 0) {
    2 ->         charct += len;
2,6151,2 ->      for (C = buf; len--; ++C) {
 6149 ->                 if (isspace(*C)) {
 1487 ->                         gotsp = 1;
 1487 ->                         if (*C == '\n') {
  269 ->                                 ++linect;
                                 }
 1487 ->                 } else {
 4662 ->                         if (gotsp) {
  968 ->                                 gotsp = 0;
  968 ->                                 ++wordct;
                                 }
                         }
                 }
    2 -> }
    1 -> if (len == -1) {
##### ->          warn("%s", file);
##### ->          rval = 1;
```

1 while循环的执行计数
2 for循环的执行计数
3 针对每个字符执行
4 针对每一行执行
4 针对每个单词执行
5 从未执行的代码行

图 10.17 gprof 输出的基本块计数

要探究程序动态运行时的每个细节，需要在调试器中运行它。虽然调试器主要用于查找程序的错误，它们还是分析程序运作的万能工具。下面的列表概括了对阅读代码最有帮助的调试器特性。

- 程序的单步执行允许我们针对给定的输入，跟踪程序执行的精确顺序。大多数编译器允许我们跳过子例程调用(当对特定的例程不感兴趣时)或者进入调用(当希望分析例程的操作时)。

① netbsdsrc/usr.bin/wc/wc.c:201-228

- **代码断点**能够在程序执行到特定的点时，让程序停下来。我们可以使用断点快速地跳转到感兴趣的点，或检验某块代码是否得到执行。
- **数据断点**可以当特定的数据项（如变量）被读取或修改时停止程序的执行。使用现代处理器提供的硬件支持，这项原本不受人关注的特性，能够使您有效地监控程序中对数据的访问。
- **变量显示**为我们提供相应的视图，显示变量的值；使用它们可以监控这些变量在程序运作过程中如何变更。
- **栈转储**为我们提供通向当前执行点的调用历史。栈转储包含每个例程及其参数（在 C 和 C++中从 main 开始）。
- **结构浏览**允许我们折叠或展开结构的成员，并跟踪指针，对数据结构进行分析、理解和检验。

调试器是另一类极大地受益于图形界面的工具。如果您使用基于行的调试器，则需要考虑换另外一种拥有图形前端的调试器，如 xxgdb。

练习 10.29　在系统调用监控程序的监控下运行某个程序，推理程序做出的操作系统调用。非交互式程序往往比交互式程序产生更少的噪音（指无关的输出——译者注）。

练习 10.30　获取运行 printf 程序产生的剖析数据[①]。使用这些数据（不要查看源代码），画出程序的结构图（例程之间的互相调用）。

练习 10.31　启用行计数剖析，编译快速排序的实现[②]。编写一个小的程序（我们称之为测试马具）对代码行进行排序，同时分析当给予程序随机数据、升序或降序次序的数据时，相应代码行的执行次数。您可以使用 BSD Unix jot 程序，创建随机数据元素序列，并用 sort 对它们进行排序。代码行执行的次数与您的预期相符吗？

练习 10.32　创建快速索引卡，概括您的编程环境中如何应用我们介绍的调试器特性。

10.9　非软件工具

最后，我们讨论另外一些可以协助代码阅读的强大工具，但它们与传统意义上的工具有所不同。

将您觉得难以理解的代码打印到纸上。计算机屏幕一般由 100 万左右像素组成；而激光打印机输出的材料，一页能够包括超过 3000 万个点。这种不同使得每页可以容纳更多信息，并能减轻眼睛的疲劳。更为重要的是，打印出来的材料可以带到任何地方，可以方便地用荧光笔标示代码、画线、以及加边注。可以使用荧光笔、颜色笔、报事贴和任何其他有助于代码理解的工具。

① netbsdsrc/usr.bin/printf/printf.c

② netbsdsrc/lib/libc/stdlib/qsort.c:1-182

可以绘制图示来描绘代码的运作。通过使用打印材料，我们可以离开计算机屏幕，现在可以取出一张纸，针对希望理解的代码绘制相应的图示。不要将精力花在制作完美甚至正确的图示上；它们的目的是帮助您理解您所阅读的代码。可以绘制成图示的内容包括：

- 函数，将函数名放在小方框中，用箭头表示函数调用或数据流向。
- 类层次或具体的对象实例图，填上正确的属性。
- 数据结构（树、表等），用箭头表示指针。
- 位映射寄存器中位字段和对应的掩码。
- 状态变迁图，用圆圈表示状态，箭头表示状态变迁。
- 字符串或数组，以及对应的索引和指针。
- 实体关系图，理解大型系统的各个部分如何组合在一起。

在绘制时使用铅笔，并随着您对程序运作的逐渐理解做出修正。

可以试着向别人介绍您在阅读的代码，这样做一般会增进您对代码的理解。尝试向第三者详细表述程序的运作，使他们能够理解，这会强制您以不同的步调思考，并会发现可能为您所忽略的细节。另外，要正确地解释程序，您可能需要利用补充材料，比如书籍、图示和打印材料，在您尝试理解代码时，可能出于懒惰而没有使用它们。

要理解复杂的算法或巧妙的数据结构，要选择一个安静的环境，然后聚精会神地考虑，不要借助于任何计算机化或自动化的帮助。Dijkstra 在一次讲演中详细谈到，当要求人们推理特定同步算法的推导过程时，如果他们抓着铅笔或钢笔，那么总是会受到干扰[Dij01]。打断也同样会使人分心。心理学家使用术语“流（flow）”来表示处理特定任务时陷入深思甚至冥思的条件，该条件常常与思维的愉悦状态相关，并会忘却时间的流逝。这就是我们在分析复杂代码时希望处于的状态，遗憾的是，需要 15 分钟才能进入这种状态，而任何打断（电话铃、新电子邮件的到达、或同事的求助）都会将您拉回到工作间中。因此，您需要创建一个环境和工作习惯，使得在需要时您能够不受打扰地工作。

练习 10.33 将拓扑排序程序[①]打印出来，向一个朋友解释 tsort 的运作。为了更为可信，可以向非程序员、宠物或盆景树进行解释。

练习 10.34 在绘制了合适的图示后，重复上面的练习。

练习 10.35 测量您的工作环境中，可以多长时间不受到打扰。计算处于“流（flow）”状态的时间，并将它与您总共花在工作上的时间进行比较。

可用工具及进阶读物

在本章开始处，我们就曾提到，我们提供的大部分例子都基于 Unix 工具家族。在各种 Unix 变体以及最近的 MacOS 中都可以使用这些工具。使用 Microsoft Windows 系统，也

① netbsdsrc/usr.bin/tsort

可以受益于这些工具：人们已经为在 Windows 环境中提供 Unix 工具的功能做了大量的工作。Unix 工具的 UWIN 移植[Kor97]和库支持 X/Open Release 4 的所有头文件、接口和命令。Cygwin[①][Noe98]是为 Unix 应用程序提供的完全的 Win32 移植层。它支持 GNU 开发工具，并通过支持所有 POSIX. 1/90 调用以及其他 Unix 版本专有的功能，使得我们勿需付出任何努力，就可以将许多 Unix 程序移植到 Windows 平台。OpenNt[Wal97]，当前作为 Interix[②] 进行营销，它是一个完整的移植和运行环境，可以将传统 Unix 上开发的应用程序源码，直接移植到 Windows NT。已经完成移植的软件包括许多 X11R5 客户，以及超过 200 个其他工具。另外，微软提供一个软件包，Windows Services for Unix[③]，其中包括大量常用 Unix 工具，以及 Mortice Kern Systems[④]（用以推销完全版的 MKS Toolkit）。如果您想查找平台无关的 IDE，应该了解一下 Eclipse IDE[⑤]。

作为上述工作的补充，outwit[⑥] 工具集[Spi00]提供遵循 Unix 工具设计原则的工具，允许使用复杂的数据操纵流水线，对 Windows 应用程序的数据进行处理。outwit 工具可以访问 Windows 的剪贴板、注册表、事件日志、关系型数据库、文档属性和外壳链接。

最后，如果在您的工作环境中找不到立即可以使用的工具，可以从本书的附书光盘中找出相应的源代码，移植并编译工具。我依旧深深地记得，1970 年代，当我们在运行 OS/32 的 Perkin-Elmer 机器上实现一个简单版本的 grep 后，对生产率的巨大推进。

有关正则表达式，最完整的参考书是 Friedl 和 Oram[FO02]。要得到词汇工具的实现细节，可以阅读 Aho 和 Corasick[AC75]和 Hume[Hum88]等文章。Aho 等著的[AHU74，ASU85]介绍了用正则表达式对字符串进行查找的理论基础。除了 grep 和您的编辑器之外，其他强有力的文本处理工具是 sed[McM79，DR97b]，awk[AKW88，DR97a]，Unix 外壳[Bou79，KP84，Bli95，NR98]，Perl[WCSP00，SCW01]，以及 Python[Lut02]。Perl 是编写小型工具最通用的语言之一，Schwartz 等著的[SCW01]是关于 Perl 的标准介绍性文章，一些著作[WCSP00，Sri97，CT98]专门针对高级用户。在开发自己的代码阅读工具时，您可以利用领域专用语言[Spi01]提供的快速开发技术。签名调查（signature survey）方法和相关的工具在 Cunningham[Cun01]中做了介绍。如果您对可视化技术感兴趣，那么 Tufte[Tuf83]对您很有参考价值，并且还是不错的装饰品。Manber 和 Wu[MW94]介绍了 glimpse 工具。Oman 和 Cook 的著作[OC90]出色地概括了排版风格对代码易读性的影响。最初的程序检查器，lint，开始时只是记述在技术报告[Joh77]中，然后复制到 Unix 程序员手册[Joh79]中。[Koe88，Hor90，RS90]详细记述了编译器经常警告的 C 问题和可移植性问题。如果您对不可读的程序十分着迷，可以参考 Libes 的著作[Lib93]。Graham 等著的[GKM82，GKM83]详细介绍了 gprof 执行剖析器的运作。Adams 和 Muchnick[AM86]介绍了图形化调试器的前端设计。当前，绘制各种与程序相关的图示时，采用的标准记法是 UML[FS00，BRJ99，RJB99]。极限编程（extreme programming）[Bec00]是一种开发过程，它通过推广组对编程，积极地鼓励程序员进行交流。[DL99，Bra86]介绍了生产工作环境中“流（flow）”的概念。

① *http://sources.redhat.com/cygwin/*

② *http://www.interix.com/*

③ *http://www.microsoft.com/windows/sfu*

④ *http://www.mks.com/*

⑤ *http://www.eclipse.org/*

⑥ *http://www.dmst.aueb.gr/dds/sw/outwit/*

第 11 章　一个完整的例子

一项成功的技术，必须先要经受事实的考验，因为没有人能够欺骗大自然。

——Richard Phillips Feynman

到目前为止，本书列举的例子都是针对孤立代码。本章中，我们将通过一个大一些的例子，分析如何在实践中应用代码阅读与理解技能。我们的目标是增强 hsqldb 数据库，使它本地支持一个新 SQL 日期/时间函数。我们选择添加的函数，PHASEOFMOON，返回一个 0～100 的数字，表示给定日期的月相，0 表示新月，100 表示满月。在 34 000 多行代码中，hsqldb 并非一段无足轻重的代码。在比较庞大的代码上执行一项小的修改，比如 bug 修复或扩展，是我们经常要做的一项任务，这项任务总是要求高级的代码阅读技能。另外，我们将会采用代码阅读技术，找出并移植现有的算法的实现，并调试我们引入的修改。

接下来我们以个人日志的形式，按照时间次序，介绍整个过程（因而使用第一人称来叙述）。

11.1　概　　况

我首先从分析系统的顶层目录入手，了解 hsqldb 发行程序的大概结构。这里没有 README 文件，但有一个名为 index.html[①] 的文件，就其名称来说，好像是应该从这个文件入手。确实，这个文件概括了所有顶层目录，并提供了系统的总体布局。

下一步，我要根据源代码编译该系统，并试着运行它。这一步有许多好处。首先，我可以从中了解系统的编译过程；由于在更改过程中，我会多次编译该系统，能够控制编译过程是最基本的。另外，我还要确保我手头的代码确实能够编译通过，从而，我不会将时间无谓地浪费在一个根本不能编译的系统上。编译过程失败的原因各种各样：源代码被破坏；不合适的编译环境、编译器或工具；缺失库或组件；配置不合法。然而，所有这些因素必须在早期解决，尽量减少误入歧途的风险。另外，通过运行程序，我可以对程序的功能有更多的了解（注意，和对待代码一样，我希望尽量少地阅读手册），同时还可以试验测试用例，之后，我们将要用这些测试用例测试我们的更改。最后，一个能够编译通过并运行的系统为我提供了一个可以依赖的牢固起点；从这一点开始往后，任何编译或运行失败都将是我的错误。相反地，最初编译过程中遇到的任何问题，都与我后面的修改无关。

index.html 文件提供一个链接，其中记述了系统的编译过程。运行 build.bat[②] 脚本，将会进行干净的编译。为了验证编译过程，我需要运行系统。index.html 中另一项链接将

① hsqldb/index.html

② hsqldb/src/build.bat

我引向 hsqldb Web 站点的一个副本[①]。通过有选择地浏览它的内容，我了解到包是如何工作的，以及它的基本操作模式。我现在运行 bin 目录中的数据库管理器（run DatabaseManager），开始我的首次交互会话。由于我将要加入一个新的日期函数，所以我阅读了系统 Web 页中提供的 SQL 语法，并试着使用几个相关的 SQL 命令。

```
create table test (d date);
insert into test values('2002-09-22')
select year(d) from test;
```

最后一条命令在结果窗口中显示单独的一列，其中包含预期的内容，2002。

11.2　攻坚计划

我现在制订如何开展工作的计划。首先，我在 SQL 文档中查找属性和类型都与我希望加入的函数相似的那些函数。然后，依照这些现有函数的结构构造我自己的函数。为了能够有效地使用纯文本匹配，查找整个代码，我选择一个名称比较罕见的函数（DAYOFWEEK），放弃那些名为 YEAR 和 HOUR 的函数。

现在，我可以查找该函数实例，从中大致地了解我要修改哪些文件。

```
$ cd src/org/srq ldb
$ grep -i dayofweek * .java
Library.java:    [...],"DAYOFWEEK",
Library.java:   "org.hsq ldb.Library.dayofweek","DAYOFYEAR",
Library.java:    public static int dayofweek(java.sql .Date d) {
```

我打开 Library. java 文件，查找 DAYOFWEEK 字符串。第一个实例出现在下面的上下文中[②]。

```
final static String sTimeDate[] = {
    "CURDATE", "org.hsqldb.Library.curdate", "CURTIME",
    [...]
    "DAYOFMONTH", "org.hsqldb.Library.dayofmonth", "DAYOFWEEK",
    "org.hsqldb.Library.dayofweek", "DAYOFYEAR",
```

从中可以很容易地得出，数组 sTimeData 将 SQL 函数映射到对应的 Java 实现。

我以大小写不敏感的方式重复查找，继续定位 dayofweek 函数的实例。该字符串的第二个（也是最后一个）实例是一个静态方法定义[③]。

```
/**
*
* Method declaration
* @ param d
```

① hsqldb/doc/internet/hSql. html
② hsqldb/src/org/hsqldb/Library. java:85-89
③ hsqldb/src/org/hsqldb/Library. java:814-824

```
* @ return
*/
public static int dayofweek(java.sql.Date d) {
    return getDateTimePart(d, Calendar.DAY_OF_WEEK);
}
```

从中可以看出，这个方法没有给出任何有意义的注释，但我觉得，我应该了解它的功能，因为我的新代码就要模拟类似的行。假定 getDateTimePart 是 Java 库的一部分，我打开 JDK 文档，详细阅读 Calendar 类。然而，文档中没有给出任何有关 getDateTimePart 函数的参考信息，因而，我只好继续在 hsqldb 源代码中进行查找。我发现它用在不同的实例中，我还分离出它的定义①。

```
private static int getDateTimePart(java.util.Date d, int part){
    Calendar c = new GregorianCalendar();
    c.setTime(d);
    return c.get(part);
    }
```

同样，该定义之前是一个没有填写的 javadoc 注释模板；没有提供什么帮助。该函数显然作为 Calendar 类的实例创建 c，将它的值设为传递进来的日期，并返回 Calendar 中一个用 part 参数指定的元素。

我的任务是返回与月相相关的部分。首先，我在 Java Calendar 和 GregorianCalendar 类的文档中查找单词“moon”。我没有找出任何有用的信息，接下来我继续阅读 GregorianCalendar 的 JDK 注解，了解该类的构架。GregorianCalendar 的实现细节很吸引人；然而，我需要编写新的代码来计算月相。

11.3 代码重用

我不想去重新发明车轮，因此，我决定查找月相计算的现有算法。我可以使用搜索引擎搜索整个 Web 或开放源码软件的储存库，但我还是先从手边最近的材料开始，在本书配套盘中查找单词“moon”。我立即意识到我需要去伪存真，从查找出的结果中分离出真正有用的内容。

```
$ find. -type -f -print | xargs grep -i moon
./netbsdsrc/games/battlestar/nightfile.c:
  " Feather palms outlined by mellow moonlight and a silvery black ocean line\n\
./netbsdsrc/games/battlestar/nightfile.c:
  by a huge tropical moon stretches at least 30 meters inland.\n\
[...]
./netbsdsrc/games/hack/hack.main.c:
  pline("You are lucky! Full moon tonight.");
./netbsdsrc/games/pom/pom.c:
```

① hsqldb/src/org/hsqldb/Library. java:777-783

```
    * Phase of the Moon. Calculates the current phase of the moon.
[...]
./netbsdsrc/lib/libc/stdlib/malloc.c:
    * a memory fault if the old area was tiny, and the moon
```

在大量虚假的引用和讽刺性注释之中，pom.c[①] 中的代码好像就是实际计算月相的代码。快速地查看了 NetBSD 的手册页之后，我的直觉得到了验证，并获得了稍许解脱感。

The *pom* utility displays the current phase of the moon. Useful for selecting software completion target dates and predicting managerial behavior.

（pom 实用程序显示当前的月相。用于选择软件完成的目标日期以及预测管理上的行为。）

我继续阅读程序的源代码，试图提取出月相算法，将它移植到 hsqldb 的 Java 源代码中。一段注释列出了详细介绍这个算法的书籍。[②]

```
* Based on routines from `Practical Astronomy with Your
* Calculator`, by Duffett-Smith. Comments give the section
* from the book [...]
```

这条信息很有用；但，我还是先试着直接从代码中提取含义。

该文件中最大的函数是 potm 函数，最终的计算可能就是由它来完成的。[③]

```
double
potm(days)
    double days;
```

同样，该函数的注释（potm—return phase of the moon）除了可以证明函数的名称之外，没有什么用处。由于我对 days 参数代表的内容毫无头绪，因此，我找到调用函数的地方，试图从它的使用中找出一些线索。[④]

```
today = potm(days) + .5;
```

之后，我继续逆向阅读，查看 days 变量的值由何而来。[⑤]

```
struct timeval tp;
struct timezone tzp;
struct tm *GMT;
time_t tmpt;
double days, today, tomorrow;
int cnt;

if (gettimeofday(&tp, &tzp))
    err(1, "gettimeofday");
```

① netbsdsrc/games/pom/pom.c

② netbsdsrc/games/pom/pom.c:54-55

③ netbsdsrc/games/pom/pom.c:133-135

④ netbsdsrc/games/pom/pom.c:104

⑤ netbsdsrc/games/pom/pom.c:89-103

```
tmpt = tp.tv_sec;
GMT = gmtime(&tmpt);
days = (GMT -> tm_yday + 1)+ ((GMT -> tm_hour +
    (GMT -> tm_min / 60.0) + (GMT -> tm_sec/3600.0)) /24.0);
for (cnt = EPOCH; cnt< GMT -> tm_year; ++cnt)
    days += isleap(cnt) ? 366 : 365;
```

这段代码，除了用到 EPOCH 常量以外，大部分使用标准的 ANSI-C 和 POSIX 函数。因此，我推断在 Java 中，以这种方式使用的 potm 的参数，不需要再去了解 days 参数实际上表示什么。我注意到，这段代码好像在计算从 EPOCH 到 tp 之间的天数。我不知道 isleap 函数的用途，以及为什么要使用这个函数，但我决定以后再处理这个问题。

我继续逆向查找源代码，检查 EPOCH 常量。通过查找，我发现，可能有许多常量需要移植到 Java 代码中。①

```
# define PI 3.141592654
# define EPOCH 85
# define EPSILONg 279.611371 /*  solar ecliptic long at EPOCH */
# define RHOg 282.680403 /*  solar ecliptic long of perigee at EPOCH */
# define ECCEN 0.01671542 /*  solar orbit eccentricity */
# define lzero 18.251907 /*  lunar mean long at EPOCH */
# define Pzero 192.917585 /*  lunar mean long of perigee at EPOCH */
# define Nzero 55.204723 /*  lunar mean long of node at EPOCH */
```

我在编辑器中(vi)②中执行下面基于正则表达式的替换操作：

```
:'a,'bs/#define^I\([^^I]*\)[ ^I]* \(.*\)^I\(.*\)/^I\3^M^I
private static final double \\1 = \\2;
```

自动将上面大部分预处理器定义转换成 Java 常量定义。

```
private static final double PI = 3.141592654;
private static final double EPOCH = 85;
/*  solar ecliptic long at EPOCH */
private static final double EPSILONg = 279.611371;
/*  solar ecliptic long of perigee at EPOCH */
private static final double RHog = 282.680403;
/*  solar orbit eccentricity */
```

---

① netbsdsrc/games/pom/pom.c:70-77

② 虽然这个语句看起来好像是老式调制解调器突发的 tty 噪音，但分析它并非十分困难。这个语句所做的全部事情是(长吸一口气)：从标记为 a 的代码点到标记为 b 的代码点('a,'b)，替换(s/)#define 后跟制表符(^I)后跟存储串(\(...\))\1；除空格或制表符之外([^I])的任何字符重复 0 或多次(*)，后跟一个空格或制表符([^I])，后跟存储串\2；任何字符重复 0 或多次，后跟一个制表符(^I)，后跟存储串\3；同样包含任何字符重复 0 或多次，为(/)一个制表符，后跟存储串\3(注释)，后跟新行符(^M)，后跟一个制表符和 private static final double，后跟存储串\1(常量名)，后跟=，后跟存储串\2(常量的值)(其实就是从每一行中提取出变量名、常量和注释，然后将注释放在单独的一行，换行，将 #define 替换成 private static double，然后是提取出的变量名，符号=，常量。——译者注)。

```
private static final double ECCEN = 0.01671542;
/*  lunar mean long at EPOCH */
private static final double lzero = 18.251907;
/*  lunar mean long of perigee at EPOCH */
private static final double Pzero = 192.917585;
/*  lunar mean long of node at EPOCH */
private static final double Nzero = 55.204723;
```

为了查看 isleap 是否为一个标准的 C 函数，我迅速地编写了一个简单的测试程序。

```
# include < time.h >
main() {int i = isleap(444); }
```

这个函数好像没有定义，所以，显然该函数不是标准函数；因此，我在该文件包含的文件中查找它的定义①。

```
# define isleap(y) ((((y) %  4) == 0 && ((y) %  100) != 0) || \
                    ((y) %  400) == 0)
```

从它的定义中，我们得出，这个函数以年份为参数，当该年是闰年时返回 true。我在 Java GregorianCalendar 类中查找单词“leap”；我发现它用 isLeapYear 方法提供同样的功能。

我继续浏览 pom.c 文件，查找其他需要移植到 Java 语言的函数。针对找出的每个函数（dtor 和 adj360），我搜索源代码，看对应的函数在什么地方使用。两个函数都用在 potm 中，因而都需要移植。

现在，我将注意力转移到理解 potm 函数的返回值表示什么。potm 中的代码根本不可理解。②

```
return(50 * (1- cos(dtor(D))));                    /* sec 63 # 3*/
```

然而，根据调用函数的代码序列，我们可以看出，返回的月相是 0～100 之间的浮点数，表示从新月到满月的月相。③

```
if ((int)today == 100)
    (void)printf("Full\n");
else if (!(int)today)
    (void)printf("New\n");
```

对这个值的随后应用证实了这个理论，由于 50 显然表示月亮的二分之一值（半月）。④

```
if ((int)today == 50)
    (void)printf("%s\n", tomorrow > today ?
        "at the First Quarter" : "at the Last Quarter");
```

① netbsdsrc/include/tzfile.h:151
② netbsdsrc/games/pom/pom.c:163
③ netbsdsrc/games/pom/pom.c:106-109
④ netbsdsrc/games/pom/pom.c:112-114

原来的 pom.c[①] 代码显示当前的月相。由于新的 moonPhase 方法将会计算作为参数传递给它的日期所对应的月相。我觉得针对当前日期月相的 C 代码不容易移植到 Java 中。我需要花一些时间来了解 days 变量的意义。计算中的第一步[②]

```
days = (GMT -> tm_yday + 1) + ((GMT -> tm_hour +
    (GMT -> tm_min / 60.0) + (GMT -> tm_sec / 3600.0)) / 24.0);
for (cnt = EPOCH; cnt < GMT -> tm_year; ++cnt)
    days += isleap(cnt) ? 366 : 365;
today =potm(days) + .5;
```

包含除以 24 和加常量 365 等操作;这些事实都强烈地表明 days 可能代表从 EPOCH 开始的天数(包括小数)。如果我的理解正确,那么,可以根据作为参数传递给 moonPhase 方法的 Date 对象,与表示 EPOCH 的 Date 对象,二者 Java getTime 方法的差,对这段代码进行重写。

```
GregorianCalendar e =
    new GregorianCalendar(EPOCH, Calendar.JANUARY, 1);
return potm((d.getTime() - e.getTime().getTime()) /
    1000.0 / 60.0 / 60.0 / 24.0);
```

然而,许多问题依旧没有解决,还不知道代码如何处理时区问题,以及如何在 C 和 Java 方法间一致地处理闰年的问题。为了解答这些问题,我决定将我的 Java 实现计算的结果与从 C 版本获得的结果进行对照,而不是花更多的时间试图理解代码。

我几乎是逐字逐句地,不加理解地将余下的代码转换到 Java 中。adj360 函数需要多花一些时间,因为它的参数是指针。阅读函数体的代码,我明白这个指针用来以引用传递参数,并在调用上下文中修改它的值[③]。

```
void
adj360(deg)
    (double *deg)
{
    for (;;)
        if (*deg < 0)
            *deg += 360;
        else if (*deg > 360)
            *deg -= 360;
        else
            break;
}
```

通过在函数的末尾加上一个 return 语句(这是函数的惟一出口),用源到源的转换,可以容易地完成对该函数以及相关调用的修改。从而,Java 中等同的代码如下:

① netbsdsrc/games/pom/pom.c

② netbsdsrc/games/pom/pom.c:100-104

③ netbsdsrc/games/pom/pom.c:181-192

```
private static double adj360(double deg)
{
    for (;;)
        if (dge < 0)
            deg += 360;
        else if(dge > 360)
            deg -= 360;
        else
            break;
    return(deg);
}
```

对 sin 和 cos 的调用也需要添加上 Java 的 Math 前缀；同样，我利用这个机会使用 Java 的 Math.PI 定义替换 C 程序的定义。月相算法转换成 Java 后的草稿参见图 11.1。

```
import java.util.*;

class MoonPhase {
    private static final int EPOCH = 85;
    /* solar ecliptic long at EPOCH */
    private static final double EPSILONg = 279.611371;
    [...]

    public static double moonPhase(Date d) {
        GregorianCalendar e = new GregorianCalendar(EPOCH, Calendar.JANUARY, 1);
        return potm((d.getTime() - e.getTime().getTime()) /
            1000.0 / 60.0 / 60.0 / 24.0);
    }

    public static void main(String args[]) {
        GregorianCalendar t = new GregorianCalendar(2002, Calendar.SEPTEMBER, 30);
        System.out.println(moonPhase(t.getTime()));
    }

    private static double potm(double days) {
        [...]
        Ec = 360 / Math.PI * ECCEN * Math.sin(dtor(Msol));
        [...]
    }
    private static double dtor(double deg) {
        return(deg * Math.PI / 180);
    }
    private static double adj360(double deg)
    {
        [...]
    }
}
```

其他常量

驱动测试的函数

测试马具

使用Java Math

几乎是原封不动地复制原来的C代码

见正文

图 11.1　转换到 Java 中的月相算法

11.4　测试与调试

最后，我编写了一个简单的测试程序，测试 MoonPhase 类是否正确运作，并以此结束实现 MoonPhase 类的工作。

```
public static void main(string args[]){
    GregorianCalendar t = new GregorianCalendar (2002,
                                Calendar.SEPTEMBER,30);
    System.out.println(moonPhase(t.getTime()));
```

```
}
```

用几个满月日(从日历中收集而来)测试这个类之后,我发现最初的实现不能完成功能。最可能出错的部分是我在 Java 中完全重新实现的代码。①

```
for (cnt = EPOCH; cnt < GMT -> tm_year; ++cnt)
    days += isleap(cnt) ? 366 : 365;
```

上面的代码将 EPOCH 和 tm_year 进行比较,从而,我可以合理地认为这二者都引用相同的底层数量。简略地查看 ctime 函数的手册页,得出下面 tm_year 的定义。

```
int tm_year; /*  year - 1900 */
```

从而,可以得出 tm_year 表示到 1900(隐式常量)的偏移。Java Calendar 类和 GregorianCalendar 类的对应的文档并未标示这种偏移的使用;因此,我需要相应地调整 EPOCH。

```
private static final int EPOCH = 1985;
```

经过这种更改,Java 和 C 代码好像生成兼容的结果,因而,我可以继续修改 hsqldb 的代码。根据我的理解,需要做出的惟一更改就是在 sTimeDate 数组中加入一个新的条目。

```
"org.hsqldb.MonnPhase.moonPhase", "PHASEOFMOON",
```

在重新编译完源代码后,我开始测试这个新的函数。我准备了一个小型的 SQL 测试脚本,从此以后,我要用这个脚本来测试程序的运作。

```
create table test (d date);
insert into test values('2002-09-22')
select phaseofmoon(d) from test;
```

运行上面的脚本,我得到了来自 hsqldb 的第一个错误:"unexpected token: PHASEOFMOON"。显然,我需要更好地理解如何扩展 hsqldb。我开始检查编译后的类文件,以确保我的代码得到编译。我在 classes/org/hsqldb 中找到了它们,但是这个目录中找不到重新编译的 Library 类或编译过的 MoonPhase 类。重新阅读我使用的编译脚本,我发现编译后的文件被直接存储一个 jar(Java archive,Java 存档文件)文件中。因此,我对 jar 文件进行分析,查找 MoonPhase。

```
$ jar tvf hsqldb.jar | grep -i moon
   2061 Mon Sep 30 11:30:32 GMT+03:00 2002 org/hsqldb/MoonPhase.class
```

编译文件就在这里,所以标记没有找到的原因可能在其他地方。

我再次查找 daysofweek 字符串,看是否错过了什么地方,但一无所获。然而,重新分析我加入的代码:

```
"org.hsqldb.MonnPhase.moonPhase", "PHASEOFMOON",
```

我注意到一个拼写错误,并更正了它。

```
"org.hsqldb.MoonPhase.moonPhase", "PHASEOFMOON",
```

① netbsdsrc/games/pom/pom.c:102-103

不过，在重新编译 hsqldb 之后，我依旧得到了同样的错误“unexpected token: PHASEOFMOON”。

现在，我查找产生错误消息的代码。

```
$ grep "unexpected token" *.java
(no results)
$ grep unexpected *.java
(no results)
```

我觉得可能忽视了某些源代码：那些包括“unexpected token”错误消息的代码，当然还有我应该修改的代码。递归地在目录中查找其他 Java 源文件，没有发现什么有用的东西。我试了大小写不敏感的搜索；该消息可能以一种古怪的方式编排。这一次，我获得了大量匹配项。

```
S grep - i unexpected *. java
Access.java:throw Trace.error(Trace.UNEXPECTED_TOKEN,right);
Database.java:throw Trace.error(Trace.UNEXPECTED_TOKEN,sToken);
[...]
Function.java:Trace.check(i != - 1,Trace.UNEXPECTED_TOKEN,function);
Parser.java:throw Trace.error(Trace.UNEXPECTED_TOKEN,token);
[...]
Tokenizer.java:throw Trace.error(Trace.UNEXPECTED_TOKEN,sToken);
[...]
Trace.java:UNEXPECTED_TOKEN = 10,
[...]
Trace.java:"37000 Unexpected token",
    "37000 Unexpected end of command",
```

最终证实，我在查找错误消息的过程中犯了拼写错误；在应用程序中，它确实拼作大写的“U”。受拼写的影响，我找出了 Parser. java 和 Tokenizer. java 文件中对 Trace. UNEXPECTED_TOKEN 常量的引用。通过文件名判断，Parser 类处理 SQL 语法，而 Tokenizer 类处理词汇分析；我推测错误可能来源于后者。

我快速浏览 Tokenizer. java 类的源代码，试图找出问题之所在。我的眼睛首先落在，用关键字对一个表进行初始化的代码序列上。我使用的编译器以一种不同的颜色显示字符串，所以整个代码块很突出[①]。

```
String keyword[] = {
        "AND", "ALL", "AVG", "BY", "BETWEEN", "COUNT", "CASEWHEN",
        "DISTINCT", "DISTINCT", "EXISTS", "EXCEPT", "FALSE", "FROM",
        "GROUP", "IF", "1NTO", "IFNULL", "IS", "IN", "INTERSECT", "INNER",
        "LEFT", "LIKE", "MAX", "MIN", "NULL", "NOT", "ON", "ORDER", "OR",
        "OUTER", "PRIMARY", "SELECT", "SET", "SUM", "TO", "TRUE",
        "UNIQUE", "UNION", "VALUES", "WHERE", "CONVERT", "CAST",
        "CONCAT", "MINUS", "CALL"
```

① hsqldb/src/org/hsqldb/Tokenizer.java:66-75

```
    };
```

紧跟在数组初始化之后，我看到，这些字符串被插入到另外的结构中[①]。

```
for (int i = 0; i < keyword.length; i++ )
    hKeyword.put(keyword[i], hKeyword);
}
```

之后，由于大多数 SQL 函数都不在这里，故而，我在代码中查找 hKeyword，看看这个结构中还包括什么。我查出 hKeyword 是 Hasable 的实例，隶属于该类(static)，wasName 方法用它鉴别关键字。其他方法并未向 hKeyword 变量添加元素，由于它隶属于类，所以没有必要对它进一步进行分析。然而，对 hKeyword 的应用为我提示了下一步的线索。[②]

```
return ! hKeyword.containsKey(sToken);
```

我对 sToken 的用法进行了分析。然而，它是词汇分析的核心，所以我放弃了这种做法。

在进入了死胡同之后，我不得不将注意力移回到 Library. java 源代码中，并分析我插入新函数的数组在其中的使用。我注意到，这个数组隶属于该类。[③]

```
final static String sTimeDate[] = {
```

因而，我只需要对这个特定的文件进行分析。之后，我的眼睛聚焦在最后两行代码的不对称上。

```
"YEAR", "org.hsqldb.Library.year",
"org.hsqldb.MoonPhase.moonPhase", "PHASEOFMOON"
```

显然，我插入定义的次序不正确。前面的行都以不同的格式编排[④]。

```
"org.hsqldb.Library.dayofweek", "DAYOFYEAR",
"org.hsqldb.Library.dayofyear", "HOUR", "org.hsqldb.Library.hour",
```

我没太在意常量的精确措词，因此产生了错误。如果我控制着源代码，我会重新将这段初始化代码组织成明显的两列，从

```
"CURDATE", "org.hsqldb.Library.curdate", "CURTIME",
"org.hsqldb.Library.curtime", "DAYNAME", "org.hsqldb.Library.dayname",
"DAYOFMONTH", "org.hsqldb.Library.dayofmonth", "DAYOFWEEK",
"org.hsqldb.Library.dayofweek", "DAYOFYEAR",
"org.hsqldb.Library.dayofyear", "HOUR", "org.hsqldb.Library.hour",
[...]
```

到

```
"CURDATE", "org.hsqldb.Library.curdate",
"CURTIME", "org.hsqldb.Library.curtime",
```

① hsqldb/src/org/hsqldb/Tokenizer. java:76-78
② hsqldb/src/org/hsqldb/Tokenizer. java:192
③ hsqldb/src/org/hsqldb/Library. java:85
④ hsqldb/src/org/hsqldb/Library. java:89-90

```
"DAYNAME", "org.hsqldb.Library.dayname",
"DAYOFMONTH", "org.hsqldb.Library.dayofmonth",
"DAYOFWEEK", "org.hsqldb.Library.dayofweek",
"DAYOFYEAR", "org.hsqldb.Library.dayofyear",
"HOUR", "org.hsqldb.Library.hour",
[...]
```

为了在将来避免这类问题。我修正了这个错误并重新编译。这一次，我没有得到标记错误，但却获得了一个异常：

```
"java.lang.NoClassDefFoundError:  org/hsqldb/MoonPhase   (wrong name: MoonPhase)."
```

我推断该问题来源于不正确的包规格说明。因此，我分析 Library.java 如何构造成包[①]。

```
package org.hsqldb
```

并且向 MoonPhase 类加入相同的规格说明。

用我的测试脚本重新测试 hsqldb 的代码，得到一个新的错误。

```
Wrong data type: java.util.Date in statement
   [select phaseofmoon(d) fron test;]
```

同样，我决定自底向上地在代码中查找错误的来源。通过查找错误消息(这一次我注意了大小写的问题)，我很快找到它的位置[②]。

```
$ grep "Wrong data type" * .java
   Trace.java:[...], "37000 Wrong data type",
```

对应的常量在上面几行代码中声明；查找单词 WRONG，我找出一行相关的代码[③]。

```
WRONG_DATA_TYPE= 15,
```

在分析 Trace.java 代码中错误常量和消息之间的关系时，看到表示错误的常量和对应的错误消息存储在两个不同的数组中，并且没有自动的机制(甚至一段注释)保证它们同步[④]，我感到有些心烦意乱。

```
final static int DATABASE_ALREADY_IN_USE = 0,
                 CONNECTION_IS_CLOSED = 1,
                 CONNECTION_IS_BROKEN = 2,
                 DATABASE_IS_SHUTDOWN = 3,
                 COLUMN_COUNT_DOES_NOT_MATCH = 4,
[...]
private static String sDescription[] = {
"08001 The database is already in use by another process",
"08003 Connection is closed", "08003 Connection is broken",
```

① hsqldb/src/org/hsqldb/Library.java:36

② hsqldb/src/org/hsqldb/Trace.java:108

③ hsqldb/src/org/hsqldb/Trace.java:76

④ hsqldb/src/org/hsqldb/Trace.java:63-103

```
"08003 The database is shutdown",
"21S01 Column count does not match", "22012 Division by zero",
```

我开始有所警觉。也许我做出的更动与类似的隐含相互依赖性发生了冲突。我在代码中查找 WRONG_DATA_TYPE 常量。

```
$ grep WRONG_DATA *.java
Column.java:Trace.check(i!= null,Trace.WRONG_DATA_TYPE,type);
Column.java:throw Trace.error(Trace.WRONG_DATA_TYPE,type);
Expression.java:throw Trace.error(Trace.WRONG_DATA_TYPE);
jdbcResultSet.java:throw Trace.error(Trace.WRONG_DATA_TYPE,s);
Log.java:Trace.check(check,Trace.WRONG_DATABASE_FILE_VERSION);
Table.java:Trace.check(type == Column.INTEGER,
            Trace.WRONG_DATA_TYPE,name);
Trace.java:WRONG_DATA_TYPE = 15,
Trace.java:WRONG_DATABASE_FILE_VERSION = 29,
```

我快速浏览了所有查找出来的项，它们看起来都比较复杂，我不知道每段代码想要完成什么。因此，我决定采用另一种替代方案（自顶向下）：不注意错误本身，而是检查代码如何调用我的函数。在 Library. java 中查找 sTimeDate，我发现它通过一个公开方法注册到一个 Hashable 类[①]。

```
static void register(Hashtable h) {
register(h, sNumeric);
register(h, sString);
register(h, sTimeDate);
register(h, sSystem);
}
```

这个方法在 Database. java 中被调用[②]。

```
Library.register(hAlias);
```

Hashable 字段的名称（hAlias）强烈地暗示出，SQL 名称只不过是对应 Java 函数的别名，并没有其他信息与它们关联。因此，我重新分析了我正在模仿的函数声明，注意到它的参数是 java. sql. Date 类型[③]。

```
public static int dayofweek(java.sql.Date d) {
```

而新加入函数的参数是 Date 类型。我相应地修改 moonPhase 的实现。

```
public static double moonPhase(java.sql.Date d) {
```

由于这项改动，该类的测试马具有些部分不再合法，我也对它做出了修正。

```
public static void main(string args[]){
```

① hsqldb/src/org/hsqldb/Library. java:108-113
② hsqldb/src/org/hsqldb/Database. java:82
③ hsqldb/src/org/hsqldb/Library. java:823

```
        java.sql.Date d = new jave.sql.Date(102,
                                Calendar.SEPTEMBER,30);
        System.out.println(d);
        System.out.println(moonPhase(d));
    }
```

这个函数现在看起来工作正常，于是，我从桌面的日历中选取更多的日期对它进行测试。

```
create table test (d date);
insert into test values('1999-1-31');
insert into test values('2000-1-21');
insert into test values('2000-3-20');
insert into test values('2002-09-8');
insert into test values('2002-09-15');
insert into test values('2002-09-22');
insert into test values('2002-10-21');
select d,phaseofmoon(d) from test;
```

11.5　文　　档

“我们完成工作了吗?”，不耐烦的读者一定会问。还没有完全结束——我们还需要更新系统的文档。我用相同的策略，在 doc 目录中查找 dayofweek 字符串，定位出文档中需要更改的地方。

```
find . -type f-print | xargs grep -li dayofweek
```

查找的结果显示，我可能需要修改两个文件。

```
./internet/hSqlSyntax.html
 ./src/org/hsqldb/Library.html
```

第一个文件包含日期/时间函数的有序列表。①

```
<b> Date / Time< /b>
<br>CURDATE() (returns the current date)
<br>CURTIME() (returns the current time)
[...]
<br>WEEK(date) (returns the week of this year (1 - 53)
<br>YEAR(date) (returns the year)
```

因此，我在相应的地方加入一项新条目。

```
<br> PHASEOFMOON(date) (returns the phase of the moon (0 - 100))
```

第二个文件的位置强烈地暗示出，它在某种程度上反映源代码的结果，并由 javadoc 自

① hsqldb/doc/internet/hSqlSyntax.html:477-492

动生成。因此，我检查文件的开头（自动生成的文件常常会有一段注释，说明这种情况），验证了我的猜想。①

```
< ! -- Generated by javadoc on Mon Apr 02 13:25:37 EDT 2001 --
```

最后，如果我在维护 hsqldb 的源代码，我还要向更改日志文件②中添加一条注释，并更新它的版本。另一方面，如果这项更改只是为了组织内部的某个项目，我会在源代码的根目录中加入一个 README 文件，说明我做出的修改。

11.6 观察报告

典型的软件演进活动，可能会由于许多不同的目的，促使我们理解特定的代码。本章中，为了找出我们想要重用的代码，并移植到 Java 中，同时理解和修改我们加入的新代码引起的错误，我们需要阅读代码，定位需要更改的 Java 源代码和文档。但要注意，在代码演进的过程中，代码阅读是一项机会主义的、目标导向的活动。大多数情况下，我们没有时间和精力去阅读和理解软件系统的全部代码。如果试图精确地分析代码，一般会陷入数量众多的类、文件和模块中，这些内容会很快将我们淹没；因此，我们极力减少必须理解的代码。我们不去努力实现全局地和完全地理解系统的代码，我们力图找出各种试探性（启发性）的捷径（本书充满了这类例子），并利用编译过程和实际运行系统，直接定位出需要给予注意的那些代码。grep 工具和编辑器的查找功能是首选的工具，并且常常是我们最后的手段。使用它们可以快速查找庞大的代码体，减少我们需要分析和了解的代码。许多情况下，由于遇到极难理解的代码，我们可能会进入死胡同。为了走出这种困境，我们采用一种广度优先查找策略，从多方攻克代码阅读问题，直到找出克服它们的方法为止。

① hsqldb/doc/src/org/hsqldb/Library. html:5

② hsqldb/CHANGELOG. txt

附录A 代码概况

如果我们能够查看源代码的话，许多事情都会变得极为容易。

——Dave Olson

表A.1列出了本书配套盘中的代码。下面的段落都来自于这些软件包的开发者。

表A.1 本书配套盘中的内容

目录	原始文件	语言
ace	ACE-5.2+TAO-1.2.zip	C++
apache	apache_1.3.22	C
argouml	ArgoUML-0.9.5-src.tar.gz	Java
cocoon	cocoon-2.0.1-src.tar.gz	Java
demoGL	demogl_src_v131.zip demogl_docs_v131.zip	C++
doc	(带格式文档)	PDF
hsqldb	hsqldb_v.1.61，hsqldb_devdocs.zip	Java
jt4	jakarta-tomcat-4.0-src	Java
netbsdsrc	netBSD 1.5_ALPHA	C
OpenCL	OpenCL-0.7.6.tar.gz	C++
perl	perl-5.6.1	C
purenum	purenum.tar.gz	C++
qtchat	qtchat-0.9.7.tar.gz	C++
socket	socket++-1.11.tar.gz	C++
vcf	vcf.0.3.2.tar.gz	C++
XFree86-3.3	netBSD 1.5_ALPHA	C

ACE[①②]是一个开放源码框架，它为开发高性能、分布式实时和嵌入式系统提供了许多组件和模式。ACE为socket、多路分解循环、线程和同步原语提供了强大且高效的抽象。

Apache项目[③④]是一项合作软件开发项目，目的是创建健壮的、商用级的、全功能和免费的源代码，实现HTTP(Web)服务器。

① *http://www.cs.wustl.edu/~schmidt/ACE.html*

② ace

③ *http://httpd.apache.org/*

④ apache

ArgoUML[①②]是一个强大且易用的交互式图形软件设计环境，它支持设计、开发和记述面向对象的软件。

Apache Cocoon[③④]是一个 XML 发布框架，它将 XML 和 XSLT 技术在服务器程序上的应用提升到了一个新高度。Cocoon 专为提高流水线 SAX 处理的性能和可伸缩性而设计，它将内容、逻辑和风格分隔开来，提供一种灵活的环境。集中式的配置系统和复杂的高速缓存机制屏蔽了所有这些复杂性，帮助我们创建、部署和维护磐石般稳定的 XML 服务器程序。

DemoGL[⑤⑥]是一个基于 OpenGL，C++和 Win32 的执行平台，用来创建各种视听效果。它允许在 Microsoft Windows 平台上开发独立的执行程序或屏幕保护程序。

HSQL 数据库引擎[⑦⑧](HSQLDB)是一个用 Java 编写的关系型数据库引擎，带有 Java 数据库连接(JDBC)驱动程序，支持 ANSI-92 SQL 的一个子集。它提供一个小型(约 100K)，且快速的数据库引擎，它的表既可以位于内存之中，也可以存储在磁盘上。

NetBSD[⑨⑩]是一个免费、安全和高度可移植的类 Unix 操作系统，它支持许多平台，从 64 位的 AlphaServer 到桌面系统，直至手持式和嵌入式设备。它清晰的设计与先进的特性使得它成为生产和研究环境的极好选择，它向用户提供全部源代码。

OpenCL[⑪⑫]是(或者说正在力图成为)一个可移植的、易用且高效的 C++密码类库。

Perl[⑬⑭⑮]是一种用以扫描任意文本文件，从这些文件中提取信息，并基于这些信息输出报告的语言，并专门针对这些任务进行了优化。对于许多系统管理任务，它也是一种很好的语言。该语言的目标是实用(易用、高效、完全)，而非优美(体积小、优雅、最小)。

Purenum[⑯⑰]是一个 C++大数库。它提供的无限宽度的 Integer 类型可以和所有的C++数学运算符一同使用，但不同于 int 类型，它永远不会发生溢出错误。除零或内存耗尽都是可以捕获的异常。它使用优化的内联代码，单宽度(single-width)值的软件 Integer 操作几乎和硬件 int 操作一样快。Array 类型提供可以调整大小的大数数组。

---

① *http://argouml.tigris.org/*
② argouml
③ *http://xml.apache.org/cocoon/*
④ cocoon
⑤ *http://sourceforge.net/projects/demogl*
⑥ demogl
⑦ *http://hsqldb.sourceforge.net/*
⑧ hsqldb
⑨ *http://www.netbsd.org*
⑩ netbsdsrc
⑪ *http://opencl.sourceforge.net/*
⑫ OpenCL
⑬ *http://www.perl.org*
⑭ *http://www.cpan.org*
⑮ perl
⑯ *http://freshmeat.net/projects/purenum/*
⑰ purenum

QtChat[①②]是一个基于Qt库的Yahoo! Chat客户。它包括诸如自动忽略选项、高亮显示、悄悄话和其他许多功能。

socket++[③④]定义了一系列的C++类，通过它们来使用socket比直接调用底层的低级函数要有效率。Socket++的另一个优点是它和iostream有相同的接口，使得用户可以执行类型安全的输入/输出。

Tomcat 4[⑤⑥]是Servlet 2.3和JavaServer Pages 1.2技术的官方参考实现。

Visual Component Framework[⑦⑧]是一个C++框架，它的设计目的是实现完全跨平台的GUI框架。它的灵感来源于NEXTStep Interface Builder、Java IDE如JBuilder、Visual J++和Borland Delphi和C++Builder等的易用性。

X Window系统[⑨⑩]是一个与提供商无关、与系统构架无关的网络透明窗口系统和用户界面标准。X可以在广泛的计算和图形机器上运行。

我们还在一个单独的目录[⑪]中提供本书正文中引用的包文档。

① *http://www.mindspring.com/abjenkins/qtchat/*
② qtchat
③ *http://www.netsw.org/softeng/lang/C+-/libs/socket++/*
④ socket
⑤ *http://jakarta.apache.org/tomcat/*
⑥ jt4
⑦ *http://vcf.sourceforge.net/*
⑧ 29:vcf
⑨ *http://www.x.org/*
⑩ XFree86—3.3
⑪ doc

附录 B　阅读代码的格言

这里不是没有规则，我们正在努力制订！

——Thomas Edison

第 1 章：导　论

1. 要养成一个习惯，经常花时间阅读别人编写的高品质代码。
2. 要有选择地阅读代码，同时，还要有自己的目标。您是想学习新的模式、编码风格、还是满足某些需求的方法？
3. 要注意并重视代码中特殊的非功能性需求，这些需求也许会导致特定的实现风格。
4. 在现有的代码上工作时，请与作者和维护人员进行必要的协调，以避免重复劳动或因此产生厌恶情绪。
5. 请将从开放源码软件中得到的益处看作是一项贷款；尽可能地寻找各种方式来回报开放源码社团。
6. 多数情况下，如果您想要了解“别人会如何完成这个功能呢？”，除了阅读代码以外，没有更好的方法。
7. 在寻找 bug 时，请从问题的表现形式到问题的根源来分析代码。不要沿着不相关的路径（误入岐途）。
8. 我们要充分利用调试器、编译器给出的警告或输出的符号代码、系统调用跟踪器、数据库结构化查询语言的日志机制、包转储工具和 Windows 的消息侦查程序，定出 bug 的位置。
9. 对于那些大型且组织良好的系统，您只需要最低限度地了解它的全部功能，就能够对它做出修改。
10. 当向系统中增加新功能时，首先的任务就是找到实现类似特性的代码，将它作为待实现功能的模板。
11. 从特性的功能描述到代码的实现，可以按照字符串消息，或使用关键词来搜索代码。
12. 在移植代码或修改接口时，您可以通过编译器直接定位出问题涉及的范围，从而减少代码阅读的工作量。
13. 进行重构时，您从一个能够正常工作的系统开始做起，希望确保结束时系统能够正常工作。一套恰当的测试用例（test case）可以帮助您满足此项约束。
14. 阅读代码寻找重构机会时，先从系统的构架开始，然后逐步细化，能够获得最大的效益。
15. 代码的可重用性是一个很诱人，但难以掌握的思想；降低期望就不会感到失望。
16. 如果您希望重用的代码十分棘手、难以理解与分离，可以试着寻找粒度更大一些的

包，甚至其他代码。

17. 在复查软件系统时，要注意，系统是由很多部分组成的，不仅仅只是执行语句。还要注意分析以下内容：文件和目录结构、生成和配置过程、用户界面和系统的文档。
18. 可以将软件复查作为一个学习、讲授、援之以手和接受帮助的机会。

第 2 章：基本编程元素

19. 第一次分析一个程序时，main 是一个好的起始点。
20. 层叠 if-else if-...-else 序列可以看作是由互斥选择项组成的选择结构。
21. 有时，要想了解程序在某一方面的功能，运行它可能比阅读源代码更为恰当。
22. 在分析重要的程序时，最好首先识别出重要的组成部分。
23. 了解局部的命名约定，利用它们来猜测变量和函数的功能用途。
24. 当基于猜测修改代码时，您应该设计能够验证最初假设的过程。这个过程可能包括用编译器进行检查、引入断言、或者执行适当的测试用例。
25. 理解了代码的某一部分，可能帮助您理解余下的代码。
26. 解决困难的代码要从容易的部分入手。
27. 要养成遇到库元素就去阅读相关文档的习惯；这将会增强您阅读和编写代码的能力。
28. 代码阅读有许多可选择的策略：自底向上和自顶向下的分析、应用试探法和检查注释和外部文档，应该依据问题的需要尝试所有这些方法。
29. for (i=0; i<n; i++)形式的循环执行 n 次；其他任何形式都要小心。
30. 涉及两项不等测试（其中一项包括相等条件）的比较表达式可以看作是区间成员测试。
31. 我们经常可以将表达式应用在样本数据上，借以了解它的含义。
32. 使用 De Morgan 法则简化复杂的逻辑表达式。
33. 在阅读逻辑乘表达式时，总是可以认为正在分析的表达式以左的表达式均为 true；在阅读逻辑和表达式时，类似地，可以认为正在分析的表达式以左的表达式均为 false。
34. 重新组织您控制的代码，使之更为易读。
35. 将使用条件运算符？：的表达式理解为 if 代码。
36. 不需要为了效率，牺牲代码的易读性。
37. 高效的算法和特殊的优化确实有可能使得代码更为复杂，从面更难理解，但这并不意味着使代码更为紧凑和不易读会提高它的效率。
38. 创造性的代码布局可以用来提高代码的易读性。
39. 我们可以使用空格、临时变量和括号提高表达式的易读性。
40. 在阅读您所控制的代码时，要养成添加注释的习惯。
41. 我们可以用好的缩进以及对变量名称的明智选择，提高编写欠佳的程序的易读性。
42. 用 diff 程序分析程序的修订历史时，如果这段历史跨越了整体重新缩排，常常可以通过指定- w 选项，让 diff 忽略空白差异，避免由于更改了缩进层次而引入的噪音。

43. do 循环的循环体至少执行一次。
44. 执行算术运算时，当 $b=2^n-1$ 时，可以将 a & b 理解为 a % (b+1)。
45. 将 a≪n 理解为 a * k，k=2^n。
46. 将 a≫n 理解为 a / k，k=2^n。
47. 每次只分析一个控制结构，将它的内容看作是一个黑盒。
48. 将每个控制结构的控制表达式看作是它所包含代码的断言。
49. return，goto，break 和 coutinue 语句，还有异常，都会影响结构化的执行流程。由于这些语句一般都会终止或重新开始正在进行的循环，因此要单独推理它们的行为。
50. 用复杂循环的变式和不变式，对循环进行推理。
51. 使用保持含义不变的变换重新安排代码，简化代码的推理工作。

第 3 章：高级 C 数据类型

52. 了解特定语言构造所服务的功能之后，就能够更好地理解使用它们的代码。
53. 识别并归类使用指针的理由。
54. 在 C 程序中，指针一般用来构造链式数据结构、动态分配的数据结构、实现引用调用、访问和迭代数据元素、传递数组参数、引用函数、作为其他值的别名、代表字符串、以及直接访问系统内存。
55. 以引用传递的参数可以用来返回函数的结果，或者避免参数复制带来的开销。
56. 指向数组元素地址的指针，可以访问位于特定索引位置的元素。
57. 指向数组元素的指针和相应的数组索引，作用在二者上的运算具有相同的语义。
58. 使用全局或 static 局部变量的函数大多数情况都不可重入(reentrant)。
59. 字符指针不同于字符数组。
60. 识别和归类应用结构或共用体的每种理由。
61. C 语言中的结构将多个数据元素集合在一起，使得它们可以作为一个整体来使用，用来从函数中返回多个数据元素、构造链式数据结构、映射数据在硬件设备、网络链接和存储介质上的组织方式、实现抽象数据类型、以及以面向对象的方式编程。
62. 共用体在 C 程序中主要用于优化存储空间的利用、实现多态、以及访问数据不同的内部表达方式。
63. 一个指针，在初始化为指向 N 个元素的存储空间之后，就可以作为 N 个元素的数组来使用。
64. 动态分配的内存块可以显式地释放，或在程序结束时释放，或由垃圾回收器来完成回收；在栈上分配的内存块当分配它的函数退出后释放。
65. C 程序使用 typedef 声明促进抽象，并增强代码的易读性，从而防范可移植性问题，并模拟 C++和 Java 的类声明行为。
66. 可以将 typedef 声明理解成变量定义：变量的名称就是类型的名称；变量的类型就是与该名称对应的类型。

第 4 章:C 数据结构

67. 根据底层的抽象数据类型理解显式的数据结构操作。
68. C 语言中,一般使用内建的数组类型实现向量,不再对底层实现进行抽象。
69. N 个元素的数组可以被序列 for (i=0; i<N; i++)完全处理;所有其他变体都应该引起警惕。
70. 表达式 sizeof(x)总会得到用 memset 或 memcpy 处理数组 x(不是指针)所需的正确字节数。
71. 区间一般用区间内的第一个元素和区间后的第一个元素来表示。
72. 不对称区间中元素的数目等于高位边界与低位边界的差。
73. 当不对称区间的高位边界等于低位边界时,区间为空。
74. 不对称区间中的低位边界代表区间的第一个元素;高位边界代表区间外的第一个元素。
75. 结构的数组常常表示由记录和字段组成的表。
76. 指向结构的指针常常表示访问底层记录和字段的游标。
77. 动态分配的矩阵一般存储为指向数组列的指针或指向元素指针的指针;这两种类型都可以按照二维数组进行访问。
78. 以平面数组形式存储的动态分配矩阵,用自定义访问函数定位它们的元素。
79. 抽象数据类型为底层实现元素的使用(或误用)方式提供一种信心的量度。
80. 数组用从 0 开始的顺序整数为键,组织查找表。
81. 数组经常用来对控制结构进行高效编码,简化程序的逻辑。
82. 通过在数组中每个位置存储一个数据元素和一个函数指针(指向处理数据元素的函数),可以将代码与数据关联起来。
83. 数组可以通过存储供程序内的抽象机(abstract machine)或虚拟机(virtual machine)使用的数据或代码,控制程序的运作。
84. 可以将表达式 sizeof(x)/sizeof(x[0])理解为数组 x 中元素的个数
85. 如果结构中含有指向结构自身、名为 next 的元素,一般说来,该结构定义的是单向链表的结点。
86. 指向链表结点的持久性(如全局、静态或在堆上分配)指针常常表示链表的头部。
87. 包含指向自身的 next 和 prev 指针的结构可能是双向链表的结点。
88. 理解复杂数据结构的指针操作可以将数据元素画为方框、指针画为箭头。
89. 递归数据结构经常用递归算法来处理。
90. 重要的数据结构操作算法一般用函数参数或模板参数来参数化。
91. 图的结点常常顺序地存储在数组中,链接到链表中,或通过图的边链接起来。
92. 图中的边一般不是隐式地通过指针,就是显式地作为独立的结构来表示。
93. 图的边经常存储为动态分配的数组或链表,在这两种情况下,边都锚定在图的结点上。
94. 在无向图中,表达数据时应该将所有的结点看作是等同的,类似地,进行处理任务

的代码也不应该基于它们的方向来区分边。

95. 在非连通图中，执行遍历代码应该能够接通孤立的子图。
96. 处理包含回路的图时，遍历代码应该避免在处理图的回路时进入循环。
97. 复杂的图结构中，可能隐藏着其他类型的独立结构。

第 5 章：高级控制流程

98. 采用递归定义的算法和数据结构经常用递归的函数定义来实现。
99. 推理递归函数时，要从基准范例测试开始，并论证每次递归调用如何逐渐接近非递归基准范例代码。
100. 简单的语言常常使用一系列遵循该语言语法结构的函数进行语法分析。
101. 推理互递归函数时，要基于底层概念的递归定义。
102. 尾递归调用等同于一个回到函数开始处的循环。
103. 将 throws 子句从方法的定义中移除，然后运行 Java 编译器对类的源代码进行编译，就可以容易地找到那些可能隐式地生成异常的方法。
104. 在多处理器计算机上运行的代码常常围绕进程或线程进行组织。
105. 工作群并行模型用于在多个处理器间分配工作，或者创建一个任务池，然后将大量需要处理的标准化的工作进行分配。
106. 基于线程的管理者/工人并行模型一般将耗时的或阻塞的操作分配给工人子任务，从而维护中心任务的响应性。
107. 基于进程的管理者/工人并行模型一般用来重用现有的程序，或用定义良好的接口组织和分离粗粒度的系统模块。
108. 基于流水线的并行处理中，每个任务都接收到一些输入，对它们进行一些处理，并将生成的输出传递给下一个任务，进行不同的处理。
109. 竞争条件很难捉摸，相关的代码常常会将竞争条件扩散到多个函数或模块；因而，很难隔离由于竞争条件导致的问题。
110. 对于出现在信号处理器中的数据结构操作代码和库调用要保持高度警惕。
111. 在阅读包含宏的代码时，要注意，宏既非函数，也非语句。
112. do ... while (0)块中的宏等同于控制块中的语句。
113. 宏可以访问在它的使用点可见的所有局部变量。
114. 宏调用可以改变参数的值。
115. 基于宏的标记拼接能够创建新的标记符。

第 6 章：应对大型项目

116. 我们可以通过浏览项目的源代码树——包含项目源代码的层次目录结构，来分析一个项目的组织方式。源码树常常能够反映出项目在构架和软件过程上的结构。
117. 应用程序的源代码树经常是该应用程序的部署结构的镜像。
118. 不要被庞大的源代码集合吓倒；它们一般比小型的专门项目组织得更出色。

119. 当您首次接触一个大型项目时，要花一些时间来熟悉项目的目录树结构。
120. 项目的源代码远不只是编译后可以获得可执行程序的计算机语言指令；一个项目的源码树一般还包括规格说明、最终用户和开发人员文档、测试脚本、多媒体资源、编译工具、例子、本地化文件、修订历史、安装过程和许可信息。
121. 大型项目的编译过程一般声明性地借助依赖关系来说明。依赖关系由工具程序，如 make 及其派生程序，转换成具体的编译行动。
122. 大型项目中，制作文件常常由配置步骤动态地生成；在分析制作文件之前，需要先执行项目特定的配置。
123. 检查大型编译过程的各个步骤时，可以使用 make 程序的- n 开关进行预演。
124. 修订控制系统提供从储存库中获取源代码最新版本的方式。
125. 可以使用相关的命令，显示可执行文件中的修订标识关键字，从而将可执行文件与它的源代码匹配起来。
126. 使用修订日志中出现的 bug 跟踪系统内的编号，可以在 bug 跟踪系统的数据库中找到有关问题的说明。
127. 可以使用修订控制系统的版本储存库，找出特定的变更是如何实现的。
128. 定制编译工具用在软件开发过程的许多方面，包括配置、编译过程管理、代码的生成、测试和文档编制。
129. 程序的调试输出可以帮助我们理解程序控制流程和数据元素的关键部分。
130. 跟踪语句所在的地点一般也是算法运行的重要部分。
131. 可以用断言来检验算法运作的步骤、函数接收的参数、程序的控制流程、底层硬件的属性和测试用例的结果。
132. 可以使用对算法进行检验的断言来证实您对算法运作的理解，或将它作为推理的起点。
133. 对函数参数和结果的断言经常记录了函数的前置条件和后置条件。
134. 我们可以将测试整个函数的断言作为每个给定函数的规格说明。
135. 测试用例可以部分地代替函数规格说明。
136. 可以使用测试用例的输入数据对源代码序列进行预演。

第 7 章：编码规范和约定

137. 了解了给定代码库所遵循的文件组织方式后，就能更有效率地浏览它的源代码。
138. 阅读代码时，首先要确保您的编辑器或优美打印程序的 tab 设置，与代码遵循的风格规范一致。
139. 可以使用代码块的缩进，快速地掌握代码的总体结构。
140. 对于编排不一致的代码，应该立即给予足够的警惕。
141. 分析代码时，对标记为 XXX，FIXME 和 TODO 的代码序列要格外注意：错误可能就潜伏在其中。
142. 常量使用大写字母命名，单词用下划线分隔。

143. 在遵循 Java 编码规范的程序中，包名(package name)总是从一个顶级的域名开始(例如，org.，com.sun.)，类名和接口名由大写字母开始，方法和变量名由小写字母开始。
144. 用户界面控件名称之前的匈牙利记法的前缀类型标记可以帮助我们确定它的作用。
145. 不同的编程规范对可移植构造的构成有不同的主张。
146. 在审查代码的可移植性，或以某种给定的编码规范作为指南时，要注意了解规范对可移植性需求的界定与限制。
147. 如果 GUI 功能都使用相应的编程结构来实现，则通过代码审查可以轻易地验证给定用户界面的规格说明是否被正确地采用。
148. 了解项目编译过程的组织方式与自动化方式之后，我们就能够快速地阅读与理解对应的编译规则。
149. 当检查系统的发布过程时，常常可以将相应发行格式的需求作为基准。

第 8 章：文　档

150. 阅读代码时，应该尽可能地利用任何能够得到的文档。
151. 阅读一小时代码所得到的信息只不过相当于阅读一分钟文档。
152. 使用系统的规格说明文档，了解所阅读代码的运行环境。
153. 软件需求规格说明是阅读和评估代码的基准。
154. 可以将系统的设计规格说明作为认知代码结构的路线图，阅读具体代码的指引。
155. 测试规格说明文档为我们提供可以用来对代码进行预演的数据。
156. 在接触一个未知系统时，功能性的描述和用户指南可以提供重要的背景信息，从而更好地理解阅读的代码所处的上下文。
157. 从用户参考手册中，我们可以快速地获取，应用程序在外观与逻辑上的背景知识，从管理员手册中可以得知代码的接口、文件格式和错误消息的详细信息。
158. 利用文档可以快捷地获取系统的概况，了解提供特定特性的代码。
159. 文档经常能够反映和揭示出系统的底层结构。
160. 文档有助于理解复杂的算法和数据结构。
161. 算法的文字描述能够使不透明(晦涩，难以理解)的代码变得可以理解。
162. 文档常常能够阐明源代码中标识符的含义。
163. 文档能够提供非功能性需求背后的理论基础。
164. 文档还会说明内部编程接口。
165. 由于文档很少像实际的程序代码那样进行测试，并受人关注，所以它常常可能存在错误、不完整或过时。
166. 文档也提供测试用例，以及实际应用的例子。
167. 文档常常还会包括已知的实现问题或 bug。
168. 环境中已知的缺点一般都会记录在源代码中。
169. 文档的变更能够标出那些故障点。