Tcl/Tk入门经典

（第2版）

（美）John K. Ousterhout　Ken Jones　著
张元章　译

清华大学出版社
北　京

内 容 简 介

本书介绍了 Tcl 语言、Tk 工具集以及 Tcl 和 C 语言结合编程。本书第 I 部分首先介绍了 Tcl 语言的基本概念和基础知识。第 II 部分集中介绍如何使用 Tk 工具集开发图形用户界面。第III部分讲解了如何结合 Tcl 和 C 语言进行程序开发。

本书原第一作者是 Tcl 的创造者，所以本书内容覆盖了 Tcl 语言的主要方面，且示例程序丰富，大部分示例代码可在 Tcl 安装目录的 demos 目录中找到。

本书适用于 Tcl 语言的初学者，也适用于希望了解 Tcl 8.5 版和 Tk 8.5 版新特性的读者。

北京市版权局著作权合同登记号　图字：01-2010-1107

图书在版编目(CIP)数据

Tcl/Tk 入门经典(第 2 版)/(美)奥斯德奥特(Ousterhout, J. K.)，(美)琼斯(Jones, K.)著；张元章译.—北京：清华大学出版社，2010.10
书名原文：Tcl and the Tk Toolkit, Second Edition
ISBN 978-7-302-23517-0
Ⅰ. ①T…　Ⅱ. ①奥…　② 琼…　③张…　Ⅲ. ①程序设计　Ⅳ. ①TP311.1

中国版本图书馆 CIP 数据核字(2010)第 158536 号

责任编辑：文开琪　汤涌涛
封面设计：杨玉兰
责任校对：周剑云
责任印制：王秀菊
出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编**：100084
社　　总　　机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：三河市李旗庄少明装订厂
经　　销：全国新华书店
开　　本：185×260　**印　张**：35.25　**字　数**：846 千字
版　　次：2010 年 10 月第 1 版　　**印　　次**：2010 年 10 月第1 次印刷
印　　数：1～4000
定　　价：69.00 元

产品编号：035377-01

序

在 John 完成本书的第 1 版之后，Tcl 和 Tk 已经有了巨大的改变。在几年前 John 编写的“History of TCL”(TCL 的历史)中(http://www.tcl.tk/about/history.html)，列出了一些已经出现的显著改进之处。

我于 1994 年 5 月加入了 Sun 公司，开始组建一个 Tcl 开发团队……Sun 公司提供了更多的资源使我们能够对 Tcl 和 Tk 进行重大改进。Scott Stanton 和 Ray Johnson 完成了 Tcl 和 Tk 到 Windows 以及 Macintosh 的移植，从而使 Tcl 成为了一个优秀的跨平台开发环境……Jacob Levy 和 Scott Stanton 彻底检查了 I/O 系统并增加了套接字支持，从而使 Tcl 可以方便地用于各种网络应用。Brain Lewis 为 Tcl 脚本构建了字节码编译器，将其速度提高了十多倍。Jacob Levy 实现了 Safe-Tcl，这是一个强大的安全模式，可以安全地执行非可信脚本。Jacob Levy 和 Laurent Demailly 构建了 Tcl 插件，使得 Tcl 脚本可以在网页浏览器中执行。我们还创建了 Jacl 和 TclBlend，使得 Tcl 和 Java 更紧密地协同工作。我们还增加了其他很多小的改进，如动态加载、命名空间、时间和日期支持、二进制 I/O、更多的文件操作命令以及改进的字体机制。

1997 年，John 组建了 Scriptics 公司，以创建开发工具，并从商业角度为 Tcl 提供培训和支持。我在 Scriptics 公司建立不久后加入其中，有幸与 John 以及其他很多为 Tcl 和 Tk 的成功做出贡献的才俊共同工作。Tcl 成为了第一种拥有原生 Unicode 支持(有益于国际化)和线程安全技术(有益于多线程应用程序)的动态语言，由 Henry Spencer 发布的全新语法包就包含了很多新功能以及 Unicode 支持。1998 年，John 因 Tcl/Tk 获得 ACM“软件系统奖”而广为人知，该一年一度的奖项用于奖励“有持续影响力的软件系统”。

在担任 Tcl/Tk 开发的“温和的独裁者”多年以后，John 已经准备好专注于新的探险。2000 年，他把 Tcl/Tk 的掌控权交给了 Tcl 核心小组(Tcl Core Team，TCT)，该小组由 Tcl 专家组成，集体管理 Tcl/Tk 的开发。TCT 通过 Tcl 改进建议(Tcl Improvement Proposal，TIP)来进行管理活动。每个 TIP 都是一个描述特定项目、活动或过程的简短文件。每个人都可以撰写 TIP，然后 TIP 会由 Tcl 核心小组进行讨论和发布。在以下网址您可以了解到这一过程的更多信息：http://www.tcl.tk/community/coreteam。TCT 欢迎社区成员加入规划 Tcl/Tk 未来的工作。

近年来最激动人心的进展之一是 Starkits 和 Starpacks 采用的技术，它支持 Tcl 运行环境和基于 Tcl/Tk 的应用程序的单文件发布，第 14 章将展示如何利用这项技术发布应用程序。而 Starkit 又基于另一个强有力的创新——虚拟文件系统，这个系统允许您的应用程序把 ZIP 包、FTP 站点、HTTP 站点以及 WebDAV 共享都视为可挂载的文件系统，第 11 章

将描述这项技术的使用。Tk 8.5 提供了一系列新的主题组件(widget)，比经典的 Tk 组件外观更具现代感、更统一，第 19 章将介绍新的主题组件。

除我的合著者之外，还有很多人为第 2 版的成功做出了贡献。Clif Flynt、Jeff Hobbs、Brian Kernighan、Steve Landers 以及 Mark Roseman 都用了大量的时间和精力审阅改进本书的技术内容。Joe English、Jeff Hobbs 和 Don Porter 提供了关于 Tcl 和 Tk 的一些不易察觉的细节和陷阱的认识。Mark Roseman 和他的 TkDocs 站点(http://www.tkdocs.com)提供了关于主题组件、风格以及主题的真知灼见。从本书初步成形到第 2 版的结构改进，Cameron Laird 都是一位积极的宣传者。在本书的出版过程中，Addison-Wesley 出版社的 Mark Taub 和 Debra Williams Cauley 一直与我并肩作战，Michael Thurston 帮助我改善本书的连贯性。最后，Dean Akamine 用了很长的时间从事吃力不讨好的工作：把文件一再从一种格式转换成另一种格式，帮助我学会使用 FrameMaker。我向他们以及这里没有列出姓名的贡献者们表示感谢。

Ken Jones
加州旧金山
2009 年 4 月

第 1 版序

Tcl 诞生于挫败之中。在 20 世纪 80 年代初期，我的学生和我就在加州大学伯克利分校开发了很多交互工具，大部分都是用于集成电路设计的，随后我们发现我们总是耗费大量的时间来创建差劲的命令语言。各个工具都需要某种类型的命令语言，但我们的主要兴趣在于工具本身而不是它的命令语言。我们为命令语言所花的时间很少，因而总是得到既无力又诡异的语言。更麻烦的是，用于一个工具的命令语言总是不能适用于另一个，于是我们最后得为每一个工具都开发一个新的糟糕的命令语言。挫败因此持续不断。

1987 年秋天，我意识到解决这个问题的方法是创建一种可复用的命令语言。如果能够创建一种有普遍用途的脚本语言，就像 C 语言的库函数包一样，那么它或许可以用于很多不同应用领域的不同用途。当然，这种语言需要是可扩展的，这样不同的使用者就可以向语言库提供的内核中加入他们所需要的特别的功能。1988 年春天，我决定开发这样一种语言，最终得到的就是 Tcl。

Tk 同样诞生于挫败之中。Tk 的基本概念源于 1987 年秋苹果公司关于 HyperCard 的发布。HyperCard 的功能以及它对脚本化很多不同交互元素并让它们协同工作的支持都激动人心。然而，我却感到气馁。很明显 HyperCard 系统投入了很大的开发力量，而从事大学研究项目的小组不可能具有那样强的力量。这意味着我们可能无法参与开发未来的新型交互软件。

我得出结论：我们唯一的希望就是组件化。我们不去创建一个独立包含数十万行代码的新应用程序，我们要找到把一个应用程序分解为很多较小的可复用的组件的方法。理想状态下，每一个组件都小到可以由一个小组来实现，而所需要的应用程序可以通过集成组件来实现。在这样的环境下，开发一个新的组件，把它和一些现有的组件结合起来，就能够创建出激动人心的新应用程序。

这种基于组件的方法需要一种既强劲又柔韧的“黏合剂”把组件结合到一起，我意识到共享脚本语言有可能作为这样的“黏合剂”。出于这种考虑开发了 Tk，这是基于 Tcl 的 X11 工具集。Tk 接受的组件可以是独立的用户界面控件，也可以是完整的应用程序，在两种情况下组件都可以独立地开发，然后用 Tcl 来集成各个组件，完成它们之间的通信。

刚开始时我用业余时间编写 Tcl 和 Tk，把这作为我的一个爱好。随着更多的人开始使用这个系统，我用在这方面的时间变得越来越多。到了今天，这方面的工作已经占据了我所有醒着的时间，甚至睡着也魂牵梦萦。

Tcl 和 Tk 的成功是我始料未及的。Tcl/Tk 开发者社区已经有了上万名成员，有数千个 Tcl 应用程序正在使用或正在开发。Tcl 和 Tk 的应用领域几乎覆盖了图形和工程应用的全

部范围，包括计算机辅助设计、软件开发、测试、仪器控制、科学可视化以及多媒体方面。有很多应用程序只使用了 Tcl，也有很多应用程序结合使用了 Tcl 和 Tk。使用 Tcl 和 Tk 的有数百个大大小小的公司以及一些大学和研究实验室。

仅仅依靠 Tcl 脚本，就可以创建图形化用户界面(GUI)，这对我来说也是一个惊喜。我过去一直认为每个新的 Tcl 应用程序都需要一些新的 C 语言代码，用来实现新的 Tcl 命令，再使用一些 Tcl 脚本把新的命令和 Tcl 提供的内建功能结合起来。然而，一个名为 wish 的很简单的 Tcl/Tk 应用程序出现了，很多人开始用 Tcl 脚本创建用户界面，完全不必编写任何 C 语言代码。Tcl 和 Tk 就这样提供了一个进行 GUI 编程的高级接口，隐藏了 C 程序员需要面对的很多细节。其结果是学习使用 wish 要比学习使用基于 C 语言的工具集容易得多，用户界面的编程工作量也要小得多。很多 Tcl/Tk 用户从不编写 C 程序代码，很多 Tcl/Tk 应用程序的代码只包括 Tcl 脚本。

本书向将要编写或修改 Tcl/Tk 应用程序的程序员介绍 Tcl 和 Tk。本书假定读者已经有 C 语言编程经验，并且至少熟悉一种外壳，如 sh、csh 或 ksh，并且假定读者使用过 X Windows 系统，熟悉鼠标使用、窗口缩放等概念。阅读本书不需要具有任何关于 Tcl 或 Tk 的经验，也不需要具有使用其他工具集(如 Motif)编写 X 应用程序的经验。

本书的组织方式使您可以根据自己的需要只学习 Tcl 而不学习 Tk，有关如何编写 Tcl 脚本的讨论，也与有关如何使用 Tcl 和 Tk 提供的 C 库接口的讨论分开。前两个部分在脚本编写的层次上讲述 Tcl 和 Tk，最后一个部分讲述 Tcl 的 C 语言接口。如果您和大多数 Tcl/Tk 用户一样，只编写脚本代码，那么可以只阅读本书的前两个部分。

尽管我尽了最大努力，但错误在所难免。我很愿意知道您所遇到的任何问题，不管是笔误、格式错误，还是某个章节或概念难以理解，或是示例程序的错误。在将来重印的时候我会尽力修正这些问题。

撰写本书得到了很多人的帮助。首先而且最重要的，我想要感谢 Brian Kernighan，他一丝不苟地审阅了本书的几版手稿，发现了无数大大小小的问题。我还要感谢 Addison-Wesley 的技术评审们提出的详细建议，他们是 Richard Blevins、Gerard Holzmann、Curt Horkey、Ron Hutchins、Stephen Johnson、Oliver Jones、David Korn、Bill Leggett、Don Libes、Kent Margraf、Stuart McRobert、David Richardson、Alexei Rodrigues、Gerald Rosenberg、John Slater 和 Win Treese。还要感谢 Bob Sproull，他从头到尾地阅读了定稿前的那版手稿，修正了不计其数的问题，提出了不计其数的建议。

我曾把初期的手稿通过互联网向 Tcl/Tk 社区公开，收到了来自全世界的不计其数的意见和建议。恐怕我无法一一列出做出了贡献的人员，不过，贡献者名单至少包括：Marvin Aguero、Miriam Amos Nihart、Jim Anderson、Frederik Anheuser、Jeff Blaine、John Boller、David Boyce、Terry Brannon、Richard Campbell、J. Cazander、Wen Chen、Richard Cheung、Peter Chubb、De Clarke、Peter Collinson、Peter Costantinidis、Alistair Crooks、Peter Davis、Tal Dayan、Akim Demaille、Mark Diekhans、Matthew Dillon、Tuan Doan、Tony Duarte、Paul DuBois、Anton Eliens、Marc R. Ewing、Luis Fernandes、Martin Forssen、Ben Fried、Matteo Frigo、Anderej Gabara、Steve Gaede、Sanjay Ghemawat、Bob Gibson、Michael Halle、Jun Hamano、Stephen Hansen、Brian Harrison、Marti Hearst、

Fergus Henderson、Kevin Hendrix、David Herron、Patrick Hertel、Carsten Heyl、Leszek Holenderski、Jamie Honan、Rob W. W . Hooft、Nick Hounsoe、Christopher Hylands、Jonathan Jowett、Paul-Henning Kamp、Karen L. Karavanic、Sunil Khatri、Vivek Khera、Jon Knight、Roger Knopf、Ramkumar Krishnan、Dave Kristol、Peter LaBelle、Tor-Erik Larsen、Tom Legrady、Will E. Leland、Kim Lester、Joshua Levy、Don Libes、Oscar Linares、David C. P. Linden、Toumas J. Lukka、Steve Lord、Steve Lumetta、Earlin Lutz、David J. Mackenzie、B. G. Mahesh、John Maline、Graham Mark、Stuart McRobert、George Moon、Michael Morris、Russell Nelson、Dale K. Newby、Richard Newton、Peter Nguyen、David Nichols、Marty Olevitch、Rita Ousterhout、John Pierce、Stephen Pietrowicz、Anna Pluzhnikov、Nico Poppelier、M. V. S. Ramanath、Cary D.Renzema、Mark Roseman、Samir Tiongson Saxena、Jay Schmidgall、Dan M. Serachitopol、Hume Smith、Frank Stajano、Larry Streepy、John E. Stump、Michael Sullivan、Holger Teutsch、Bennett E. Todd、Glenn Trewitt、D. A. Vaughanpope、Richard Vieregge、Larry W. Virden、David Waitzman、Matt Wartell、Glenn Waters、Wally Wedel、Juergen Weigert、Mark Weiser、Brent Welch、Alex Woo、Su-Lin Wu、Kawata Yasuro、Chut Ngeow Yee、Richard Yen、Stephen Ching-SingYen 和 Mike Young。

很多人对 Tcl 和 Tk 的开发做出了重要的贡献。没有他们的努力，本书就没有什么东西可写。虽然我不能一一感谢为今天的 Tcl/Tk 做出贡献的所有人，但我想对以下贡献者表示特别的感谢：Don Libes，编写了第一个被广泛使用的 Tcl 应用程序；Mark Diekhans 和 Karl Lehenbauer，为他们对 TclX 的贡献；Alastair Fyfe，支持了 Tcl 的早期开发；Mary Ann May-Puphrey，开发了最初的 Tcl 测试套件；George Howlett、Michael McLennan 和 Sani Nassif，进行了 BLT 扩展；Kevin Kenny，展示了 Tcl 可以用于几乎所有能想象到的通信；Joel Bartlett，为他挑战性的谈话，他的 ezd 程序正是 Tk 的画布组件的灵感来源；Larry Rowe，开发了 Tcl-DP，提供了普遍的建议和支持；Sven Delmas，基于 Tk 开发了 XF 应用程序生成器；Andrew Payne，他制作了组件导航，大力传播 Tcl。

Tcl 和 Tk 的开发得到了一些公司的资金支持，包括 DEC、Hewlett-Packard(惠普)公司、Sun Microsystems 以及 Computerized Processes Unlimited。特别感谢 Digital 的西方研究实验室及其主任 Richard Swan，允许我每周消失一天，让我有时间积累各种想法，从事 Tcl 和 Tk 的工作。

Terry Lessard-Smith 与 Bob Miller 为 Tcl/Tk 和我的所有项目提供了卓越的行政支持。我无法想象在没有他们的情况下如何才能完成项目。

最后，特别感谢我的同事兼朋友 Dave Patterson，他幽默明智的建议一直激励着我并影响着我的大部分职业生涯。感谢我的妻子 Rita，女儿 Kay 和 Amy，她们以难得的感情与欣赏包容着我的工作狂倾向。

John Ousterhout
加州伯克利
1994 年 2 月

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

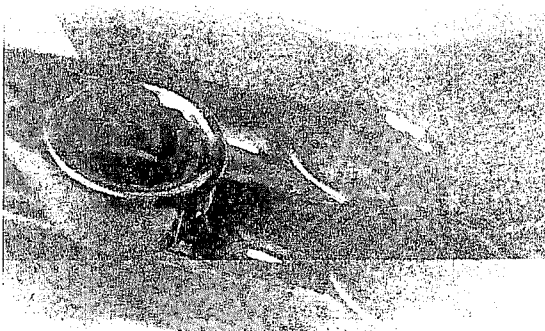
Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引



前　言

本书介绍两个分别称为 Tcl 和 Tk[①]的软件包。Tcl 是一种用于控制和扩展应用程序的动态语言(也称为脚本语言)；它的名字代表“工具命令语言”(tool command language)。Tcl 提供的通用编程能力可以满足大多数应用程序的需要。而且，Tcl 既是可嵌入的(embeded)，也是可扩展的(extensible)。它的解释器是一个 C 函数库，可以很容易地整合到应用程序中；而任何一个应用程序都可以通过增加命令来扩展 Tcl 内核的功能，这些命令可以是为应用程序特别开发的，也可以是由附加库提供的，这些附加库在 Tcl 社区中也称为扩展。

Tcl 最有用的一个扩展就是 Tk，这是一个用于开发图形用户界面(graphical user interface，GUI)应用程序的工具集。Tk 扩展了 Tcl 内核的功能，增加了构建用户界面的命令，使您可以使用 Tcl 脚本来构建图形用户界面，而不必编写 C 语言代码。和 Tcl 一样，Tk 也是由一个 C 函数库实现的，也可以用于各种不同的应用程序。

提示： 本书针对的是 Tcl/Tk 8.5 版。各版 Tcl/Tk 的发布说明描述了各版本的变化和新功能。Tcl 用户的 Wiki 网站(http://wiki.tcl.tk)提供了各个版本的变化列表，您可以搜索含有 Changes 的标题来找到这些页面。

Tcl/Tk 的好处

Tcl 和 Tk 一起为应用程序开发者和使用者提供了很多好处。首先是快速开发。很多有意思的应用程序可以完全用 Tcl 脚本编写。这使您可以在比 C/C++或 Java 更高的层次上进行开发，Tk 隐藏了 C 或 Java 程序员必须关注的很多细节。与低级的工具集相比，要使用 Tcl 和 Tk 所需要学习的知识更少，需要编写的代码也更少。通过几个小时的学习，Tcl/Tk 新手用户就可以创建有意思的用户界面，很多开发人员从其他工具集转而使用 Tcl 和 Tk 工具集后，应用程序开发所需的代码数量和开发时间都减少了 90%。

Tcl 和 Tk 适于快速开发的另一个原因在于 Tcl 是解释语言。使用 Tcl 应用程序时，可以在运行中生成和使用新的脚本，而无需重新编译和重启应用程序。这使您可以迅速尝试新的想法，迅速修正程序中的错误。因为 Tcl 是解释语言，它的运行速度比 C 代码编译的程序慢。但是通过内部优化，如字节码编译，再加上不断增强的处理器性能，与编译语言相比的大部分性能差距都可以消除。例如，您可以运行有数百条 Tcl 命令的脚本，鼠标的

① Tcl 的官方发音是“tickle”，不过“tee-see-ell”也很常用。Tk 的发音是“tee-kay”。

每一次移动都不会有能感知的延迟。在一些特别的场合，当性能成为重要问题时，可以把Tcl 脚本中影响性能的关键部分替换为 C 代码。

Tcl 的第二个好处在于它是跨平台的语言，它的大多数扩展包括 Tk 也是如此。这意味着在一个平台(如 Linux)上开发的应用程序，在大多数情况下都可以不加改动地在另一个平台上运行，如在 Macintosh 或 Windows 上运行。

Tcl 还是第一种拥有原生 Unicode 支持的动态语言。因此，Tcl 可以处理这个世界上几乎所有的书面语言。Tcl 无需扩展就可以处理 Unicode 支持的所有文本，像 msgcat 这样的标准扩展则提供了简单的本地化支持。

使用 Tcl 的另一个显著优点在于，它和它的大多数扩展都是免费的开源软件。Tcl 和 Tk 遵循 BSD 授权，允许所有人免费下载、查看、修改以及再发布。

Tcl 是一种绝妙的“胶合语言”。一个 Tcl 应用程序可以包含很多不同的扩展，每个扩展都分别提供一系列 Tcl 命令。Tk 就是函数库套件的一个例子；Tcl/Tk 社区还开发了很多其他的套件，您也可以自行编写套件。应用程序中的 Tcl 脚本可以使用各个套件提供的命令。

另外，Tcl 还可以让应用程序很容易地拥有强大的脚本语言。例如，要为一个已经存在的应用程序添加脚本能力，您只需实现几条新的 Tcl 命令，用来为应用程序提供相应的基本功能。然后把您的新命令和 Tcl 库链接起来生成全功能的脚本语言，该语言就包含了Tcl(称为 Tcl 内核)提供的命令和您编写的那些命令。

Tcl 还为用户提供了方便。一旦学习了 Tcl 和 Tk，就能为任何 Tcl 和 Tk 应用程序编写脚本，只需要学习该应用程序特有的少数几条命令即可。这使得更多的用户有能力对应用程序进行个性化改造和强化。

本书的组织结构

第 1 章通过一些简单的脚本展示了 Tcl 和 Tk 重要功能的概况，旨在让您感受一下这个系统，并让您认识到它们确实是有用的，所以不对具体细节进行深入讲解。本书余下的部分将更全面深入地讲解这个系统。全书共分为三部分(不包括附录)。

- 第 I 部分：Tcl 脚本语言介绍。通读这一部分，您就可以为 Tcl 应用程序编写脚本了。如要编写 Tk 应用程序，也需要知道本章提供的一些信息。
- 第 II 部分：讲解 Tk 提供的更多 Tcl 命令，这些命令用于创建用户界面组件，如菜单和滚动条，并把这些组件置于 GUI 应用程序中。通读这一部分，您就可以创建新的 GUI 应用程序，或是编写脚本来强化现有的 Tk 应用程序。
- 第III部分：讨论 Tcl 库中的 C 函数，以及如何使用它们创建新的 Tcl 命令。通读这一部分，您就可以用 C 语言编写新的 Tcl 套件和应用程序。不过，即使不了解这一部分的信息，您也可以用 Tcl/Tk 完成很多工作(可能是您所需要完成的所有工作)。

每部分都有许多较短的章。每一章都独立地讲解这个系统的某一部分，您不必严格按目录顺序阅读各章。

本书没有覆盖 Tcl 和 Tk 的全部功能，给出的阐释文字也只是提供一个连贯的介绍，而不是提供完整的参考文件。Tcl 和 Tk 的发行版，就包括了独立的参考手册，名为 *reference documentation*。手册内容十分简洁，但绝对覆盖了 Tcl 和 Tk 系统的全部功能。本书最后为附录部分。附录 A 讲解了如何在互联网上取得 Tcl 和 Tk 发行版，包括其参考手册。附录 B 列出了一些流行的 Tcl 扩展。附录 C 列出了 Tcl 和 Tk 其他的一些线上和纸质资源。附录 D 为 Tcl 源码发布许可(Tcl Source Distribution License)全文。

本书假定您已经知道如何使用操作系统，知道如何通过命令行方式与应用程序进行交互。本书第III部分假定您熟悉 ANSI C 标准的 C 程序语言；了解 C 语言对学习第 I 和第 II 部分也有所帮助，不过并不是必需的。阅读本书不要求您对 Tcl 和 Tk 有任何基础，我们将从零开始进行介绍。

表达说明

本书所有能输入计算机的内容都采用等宽字体，如 Tcl 脚本、C 代码、变量名、过程、命令等。采用等宽字体的 Tcl 脚本示例如下：

```
    set a 44
⇒ 44
```

Tcl 命令，如前面例子中的 set a 44，就是用等宽字体表示的；而结果，如前面例子的 44，用斜体等宽字体表示。结果前面的符号⇒表示这是一个正常返回值。如果一条 Tcl 命令出现了错误，错误信息也用斜体等宽字体表示，前面的符号是∅，表示这是一个错误而不是正常返回。

```
    set a 44 55
∅  wrong # args: should be "set varName ?newValue?"
```

在描述 Tcl 命令的语法时，用斜体等宽字体来表示形式参数名。一个或一组放置在问号间的参数表示可选参数。例如，set 命令的语法描述如下：

```
set varName ?newValue?
```

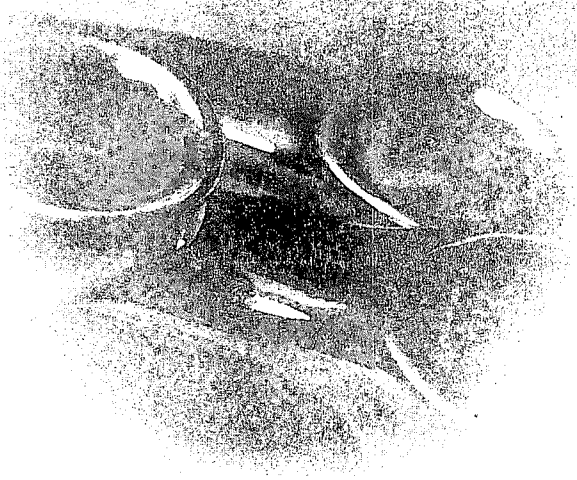
这表示您可以直接输入 set 来调用命令，*varName* 和 *newValue* 是 set 的参数名；在调用这个命令时，您应该输入一个变量名来代替 *varName*，输入一个值来代替 *newValue*。这里参数 *newValue* 是可选的。

目　录

第I部分　Tcl语言

第Ⅲ部分 C 语言中 Tcl 应用程序的编写

第Ⅳ部分 附录



第Ⅰ部分 Tcl语言

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

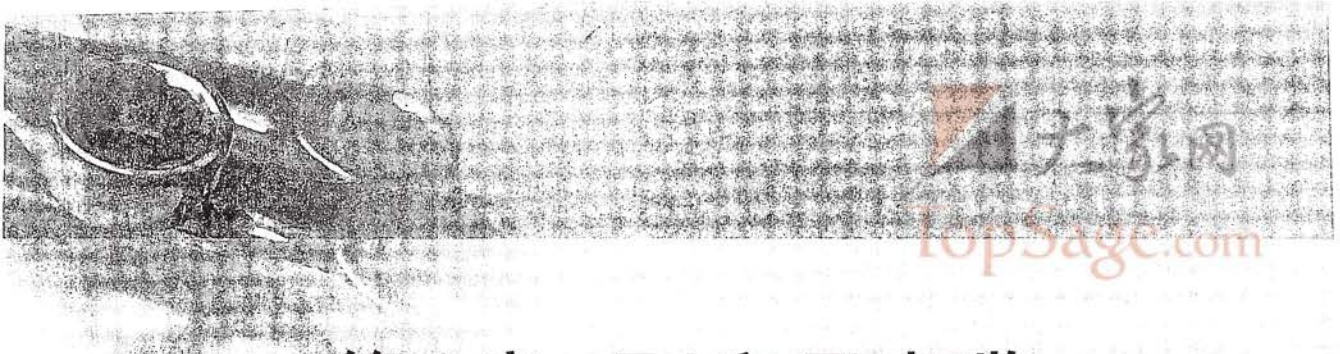
Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引



第 1 章　Tcl 和 Tk 概览

本章用一系列脚本来介绍 Tcl 和 Tk，演示它们的主要功能。尽管阅读本章后您就能够编写一些简单的脚本，但请注意本章的介绍并不全面。本章的目的在于展示 Tcl 和 Tk 的整体结构，以及它们能够做到什么，这样在后面的章节具体讨论各个功能时您就能知道为什么那些功能是有用的。本章的所有信息在后面的章节中都有更详细的阐述，而有一些重要方面，如 Tcl C 语言接口，在本章中没有加以讨论。

1.1　从这里起步

要调用 Tcl 脚本，必须运行一个 Tcl 应用程序。如果您的系统已经安装了 Tcl，应该有一个名为 tclsh 的 Tcl 外壳应用程序，您可以用它来试验本章给出的一些示例。如果您的系统还没有安装 Tcl，请参阅附录 A 中有关获取和安装它的信息。

提示：安装 tclsh 时通常把它的版本号作为其名称的一部分(如 Unix 上的 tclsh8.5 及 Windows 上的 tclsh85)。这样做的优势在于可以在同一个系统中同时存在多个版本的 Tcl，不利之处则在于编写开头相同却针对不同版本 Tcl 的脚本时会更麻烦一些。大多数人安装时会把 tclsh 连接到系统上最新的那个版本。后面将介绍的 wish 解释器也有相同的情况。因此，除了希望使用您的系统上某个特定版本 Tcl 应用程序的情况外，都可以简单地使用 tclsh 或 wish。

您可以通过 Macintosh 或 Unix 系统的终端窗口启动 tclsh 应用程序，也可以从 Windows 系统的命令提示符窗口启动。只需要输入以下命令：

```
tclsh
```

这样会在交互模式下启动 tclsh，从键盘读取 Tcl 命令，把它们传递给 Tcl 解释器进行处理。作为初学者，常要先在 tclsh 提示符后输入以下命令：

```
expr 2 + 2
```

tclsh 会输出其结果(4)，然后提示您输入其他的命令。

这个例子展示了 Tcl 的一些特点。每一条命令都由一个或多个“单词”组成，单词之间用空格或制表符(即空白字符)隔开。这个示例中共有 4 个单词：expr、2、+和 2。每条命令的第一个单词是要执行的命令名。其他单词是执行时传递给命令的参数。expr 是 Tcl 库提供的核心命令之一，任何一个 Tcl 应用程序都有这个命令。该命令将其参数连接成一个

字符串，把这个字符串按算术表达式进行处理。

每一条 Tcl 命令都会返回一个结果。如果该命令没有有意义的结果，它会返回一个空字符串。对于 expr 命令，其结果是该算术表达式的值。

Tcl 中所有的值都有字符串表达形式，其中一部分有效率更高的内部表达形式。这个示例中 expr 的结果是一个数值，可能有二进制整型或浮点型的内部表达形式。内部表达形式可以让信息处理过程更快、更有效率。如果这个值被赋给一个变量，或者另一条命令把这个值作为数值使用，就不必再进行字符串转换，从而大大提高了效率。在需要的时候，Tcl 会自动生成所需要的值的字符串表达形式，例如，当需要向控制台显示该值的时候。

提示： 在脚本开发的层次，您可以把所有的值都作为字符串对待，Tcl 会根据需要自动转换字符串和内部表达形式。随着您更加熟悉 Tcl，理解什么情况会导致转换，从而可以帮助您回避那些转换，得到更有效率、更快速的程序。一般来说，总是一致地对待一个数值(例如，总是使用 list 命令来完成列表，总是使用 dictionary 命令来进行字典操作等)，同时避免不必要的输出以及其他字符串操作、列表操作、字典操作，就能大大加速代码执行。有关 C 语言层次内部表达形式的更多信息，参见第 32 章。

从现在开始，我们使用如下表示法来描述示例：

```
    expr 2 + 2
⇒ 4
```

第一行是输入的命令，第二行是该命令返回的结果。符号⇒表示这一行的内容是返回值，实际运行 tclsh 时并不会输出⇒这个符号。我们会省略那些不重要的返回值，例如，有一系列命令而我们只关心最后一条命令的结果时，前面一些命令的返回值就会略去。

命令通常由换行符结束(通常就是键盘上的 Enter 或 Return 键)，因此您输入 tclsh 的每一行都会形成一条独立的命令。分号也用作命令分隔符，可以用它在一行之中输入多个命令。一条命令也可以扩展到几行，稍后您会看到具体的写法。

命令 expr 支持的表达式语法与 ANSI C 相似，包括同样的优先级规则和大部分 C 操作符。下面是您可以输入 tclsh 的一些示例：

```
    expr 2 * 10 - 1
⇒ 19
    expr 14.1*6
⇒ 84.6
    expr sin(.2)
⇒ 0.19866933079506122
    expr rand()
⇒ 0.62130973004797
    expr rand()
⇒ 0.35263291623100307
    expr (3>4)||(6<=7)
⇒ 1
```

第一个示例展示了乘法操作符和减法操作符，以及乘法操作符拥有更高的优先级。第二个示例展示了表达式可以含有实数和整数。接下来的示例演示了 expr 支持的一些内建函数，包括生成 0～1 之间的随机数的 rand()函数。最后一个示例展示了关系算符>和<=，以及逻辑或操作符||的使用。和 C 语言一样，布尔型结果用数值表达，1 表示真，0 表示假。

要退出tclsh，可以调用exit命令：

```
exit
```

这个命令终止应用程序并返回到外壳。

1.2 用Tk编写“Hello, world!”程序

Tcl提供了一整套的编程功能，如变量、循环和过程。这些功能可以通过执行Tcl内核提供的命令，或执行附加的扩展命令来实现。

Tcl一个很重要的扩展就是由Tk工具集提供的视窗命令。Tk提供的命令可用于创建图形用户界面。本书中很多示例都使用了一个名为wish(windowing shell)的应用程序，它和tclsh相比唯一的不同就是它还包含由Tk定义的命令。如果您已经安装Tcl和Tk，就可以像前面调用tclsh一样从终端或命令提示符窗口调用wish，它会在屏幕上显示一个小的空白窗口，然后从控制台读取命令。或者如果是Tcl/Tk 8.4或更新版本，可以先调用tclsh应用程序，然后使用命令package require Tk来动态加载Tk扩展。

提示： 在Windows系统中，调用交互会话工具wish时，会显示一个空白窗口和一个独立的控制台窗口。控制台窗口是实际控制台的替代品，允许经标准I/O通道的输入输出。执行脚本文件时控制台窗口通常是隐藏的，不过您也可以执行console show命令显示控制台窗口。更多相关信息，可参见console参考文档。

下面就是可以用wish运行的一个简单的Tk脚本。

```
button .b -text "Hello, world!" -command exit
grid .b
```

把这两行Tcl命令输入wish，窗口会变为图1.1所示。如果把鼠标移到“Hello, world!”上，然后单击鼠标主键(大多数情况下就是鼠标左键)，这个窗口就会消失，并且退出wish。

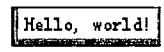


图1.1 “Hello,world!”应用程序

这个示例有几点需要解释。首先我们来看一下语法问题。这个示例包含两个命令，button和grid，两者都由Tk实现。虽然这两个命令看上去不太像前文的expr命令，但它们都有Tcl命令的共同基本结构：由空格符分开的一个或多个单词。button命令有6个词，grid命令有2个词。

button命令的第4个词括在双引号中。双引号中的词可以包含空格；如果没有双引号，这里的“Hello,”和“world!”就会被看作两个单词。这个双引号是标识符，而不是单词的一部分，在命令执行前Tcl解释器会把双引号移除。

对于expr命令而言，单词结构的意义不大，因为expr本身已经包含了它所需的命令参数。然而，对于button和grid命令以及大多数Tcl命令而言，单词结构是有重要意义

的。对于 button 命令，第一个参数是将要创建的新窗口的名字。这个命令的其他参数必须是成对出现的，各对参数中，前一个是“配置选项”(configuration option)，后一个是该选项的参数值。如果没有双引号，选项-text 的值就会被认为是“Hello,”，而“world!”会被看作一个配置选项名。因为 button 并没有定义名为“world!”的选项，这个命令就会返回一个错误。

现在让我们看一下这些命令的行为。Tk 中构建图形用户界面的基本单元是“组件”(widget，也称部件，是一种微小的应用程序视图，可嵌入其他应用程序并接收定期更新)。一个组件就是一个有特定外观和行为的窗口(在 Tk 中“组件”和“窗口”两个词有时作为同义语在使用)。组件可以分为很多类型，如按钮、菜单、滚动条等。同一类型的所有组件都有相同类型的外观和行为。例如，所有的按钮组件都要显示一个文本字符串、位图或图片，在用户点击按钮时执行相应的 Tcl 脚本。

在 Tk 中，组件是分层组织的，它们的名字就反映了它们在分层结构中的位置。主组件(main widget)，即当您启动 wish 后会出现在屏幕上的组件，都有一个名字.，而.b 代表主组件的 b 子组件。Tk 中的组件名与文件路径相仿，只不过它们使用.作为分隔符而不是/或\。因此，.a.b.c 代表.a.b 组件的一个子组件，而.a.b 是.a 组件的一个子组件，而.a 是主组件的一个子组件。

Tk 为每一类型的组件提供一个命令，称为类型命令，您可以调用它来创建该类型的组件。例如，button 命令创建按钮组件。这与标准的面向对象编程原则相似，不过 Tk 并不支持直接从组件类派生子类。所有的类型命令都有相同的格式，第一个参数是要创建的新组件的名字，其他参数指定配置选项。不同的组件类型支持不同的选项集合。通常来说组件有很多选项，在您没有特别指定时使用各自的默认值。在调用像 button 这样的类型命令时，它会根据指定的名字和配置选项创建一个新的组件。

前面示例中的 button 命令指定了两个选项：-text，指定了在按钮上显示的字符串；-command，指定了用户激活这个按钮时所执行的 Tcl 脚本。这个示例中，-command 选项的值是 exit。下面是其他一些按钮选项，您可以尝试一下。

- -background——按钮的背景色，如 blue。
- -foreground——按钮的文字颜色，如 black。
- -font——按钮使用的字体，如“times 12”代表 12 磅 Times Roman 字体。

创建的组件并不会自动显示。命令 grid 让这个按钮组件出现在屏幕上。一个名为几何管理器的独立程序负责计算组件外观的大小和位置，并把它们在屏幕上显示出来。组件的创建与几何管理的分离，使您可以很自由地设计应用程序界面的组件排列。示例中的 grid 命令要求几何管理器调用栅格管理器(gridder)来管理.b。栅格管理器在一个纵横栅格中排列显示组件。这里的命令把.b 放在了栅格的第一行第一列，并让该栅格的大小正好可以显示该组件；而且如果父栅格的大小超过需要显示的组件，那么该栅格就会变小，就像这个示例一样。当您使用 grid 命令时，主窗口(.)从它的原始大小缩小为图 1.1 中所显示的大小。

1.3 脚本文件

目前为止，您已经见过了与 tclsh 或 wish 交互的 Tcl 命令的示例。您也可以把命令写在脚本文件中，然后调用这些脚本文件，就像调用外壳脚本一样。如前文的“Hello, world!”示例，可以将如下文本内容放入名为 hello.tcl 的文件中。

```
#!/usr/local/bin/wish
button .b -text "Hello, world!" -command exit
pack .b
```

您可以调用 wish 解释器来执行这个脚本，把脚本文件名作为命令行参数给出。

```
wish hello.tcl
```

这就会让 wish 显示图 1.1 中那个窗口，等待您与其交互。这种情况下您不能输入 wish 命令，只能点击该按钮。

1.3.1 在 Unix 和 Mac OS X 中运行脚本

这个脚本和您前面在命令行输入的内容是一样的，只是增加了第一行。对于 wish 而言，这一行只是一个注释，但如果您让这个文件成为 Unix 系统中的可执行文件(例如，在您的外壳中执行 chmod +x hello.tcl)，那就可以在您的外壳中输入 hello.tcl 以直接调用这个文件。(这要求包含 hello.tcl 脚本的文件夹在 PATH 环境变量中列出。)这样做的时候，系统调用了 wish，把这个文件作为脚本传给它进行解释。

对于这个例子，当且仅当 wish 安装在/usr/local/bin 中时，这个脚本才能作为可执行脚本工作，不过您也可以把脚本文件名作为命令行参数，调用 wish 来运行脚本。如果 wish 被安装在其他地方，您就需要把第一行修改为它的安装位置。如果脚本文件的第一行长度超过 32 个字符，有的系统会出现奇怪的错误，因此，最好不要让 wish 安装位置的全路径长于 27 个字符。

满足了这些限制条件，在 Unix 上，脚本应该以如下三行代码开头：

```
#!/bin/sh
# Tcl ignores the next line but 'sh' doesn't \
exec wish "$0" "$@"
```

或者使用以下更复杂但也更健壮的三行代码：

```
#!/bin/sh
#Tcl ignores the next line but 'sh' doesn't \
exec wish "$0" ${1+"$@"}
```

不过，在大多数现代 Unix 实现中，只要在您的 PATH 环境变量中列出的某个位置安装了 wish，用下面这一行代码作为开头，脚本就可以正常运行了。

```
#!/usr/bin/env wish
```

1.3.2 在 Windows 中执行脚本

在 Windows 中，您可以使用标准系统工具把 wish 解释器和指定文件扩展名(习惯上用.tcl)关联起来，然后双击 Tcl/Tk 脚本文件的图标，就可以自动调用 wish 解释器，将该文件的文件名作为一个待解释脚本传给解释器。Tcl/Tk 的大多数 Windows 安装器都会自动为您生成这个关联。通常把 wish 作为默认关联的解释器，是因为大多数基于 Windows 的 Tcl/Tk 程序是有图形化用户界面的。不过，如果您的多数 Tcl 脚本不使用 Tk 命令，也可以改变这个默认设置，将其关联到 tclsh。

如果计划发布可以在多种平台上运行的脚本，就应该使用前文介绍 Unix 可执行脚本时所用的#!开头，这样脚本就可以在 Unix 中直接运行。而 Windows 中并无#!约定，wish 解释器只把它看成是一个注释，因此在 Windows 系统中运行时这一行会被忽略。

1.3.3 在交互式解释器中运行脚本

实际应用中，Tk 应用程序的使用者极少输入 Tcl 命令，他们使用鼠标和键盘以图形化程序通常的操作方式与应用程序互动。Tcl 在后台工作，而用户无需直接看到它。hello.tcl 脚本的行为，就与用 C 语言和 GUI 工具集编写，再编译成的二进制可执行文件一样。

但是在调试过程中，应用程序开发者经常需要交互式地输入 Tcl 命令。例如，可以交互式地启动 wish，来测试 hello.tcl 脚本(在外壳中输入 wish 而不是 hello.tcl)。然后输入如下 Tcl 命令：

```
source hello.tcl
```

source 是一条 Tcl 命令，需要一个文件名作为其参数。它读取该文件并将其作为 Tcl 脚本处理。这会生成与您直接从外壳中调用 hello.tcl 相同的用户界面，但同时您还可以交互式地输入 Tcl 命令。例如，可以编写脚本文件，将-command 选项改为：

```
-command "puts Good-bye!; exit"
```

然后，在 wish 中交互式地输入如下命令而无需重启程序。

```
destroy .b
source hello.tcl
```

第一条命令删除已经存在的按钮，第二条命令采用新的-command 选项重建按钮。现在当您单击按钮时，在退出 wish 之前 puts 命令会在标准输出端输出一条信息。

1.4 变量与替换

Tcl 允许您将值存放在变量中，然后在命令中通过变量使用这些值。例如下面这个脚本，您可以在 tclsh 或 wish 中运行它。

```
    set a 44
⇒   44
    expr $a*4
⇒   176
```

第一条命令将值 44 赋给变量 a，并返回该变量的值。第二条命令中，$让 Tcl 执行变量替换(variable substitution)：Tcl 解释器将$符和它后面的变量名替换为该变量的值，因此 expr 接收到的参数实际上是 44*4。在 Tcl 中变量无需声明；在调用 set 时它们会被自动创建。变量值总是可以用字符串表示，但可能是以原生二进制形式保存的。字符串可以包含二进制数据，也可以是任意长度。当然，在这个 expr 语句中，如果 a 的值无法解释为整数或实数，就会产生错误。

Tcl 还支持命令替换(command substitution)，允许把一条命令的结果作为另一条命令的输入参数。

```
    set a 44
    set b [expr $a*4]
⇒   176
```

使用方括号启用命令替换：方括号中的所有内容都作为一个独立的 Tcl 脚本处理，其结果将替换到这个方括号的位置。这个示例中，第二条命令 set 的第二个参数是 176。

Tcl 中还有一种反斜线替换(backslash substitution)，其作用是给普通字符添加特殊含义，或者取消特殊字符的特殊含义，示例如下：

```
set x \$a
set newline \n
```

第一条命令将变量 x 设置为字符串$a(字符\$被替换为$，而不再进行$所代表的变量替换)。第二条命令将变量 newline 赋值为包含一个换行符的字符串(字符\n 会由一个换行符替换)。

1.5 控制结构

下面这个示例使用了变量和替换以及一些简单的控制结构，创建了一个名为 factorial 的 Tcl 过程(procedure)，这个过程可以计算非负整数的阶乘。

```
proc factorial {val} {
    set result 1
    while {$val>0} {
        set result [expr $result*$val]
        incr val -1
    }
    return $result
}
```

如果在 wish 或 tclsh 中输入以上代码，或是把它们写入一个文件，然后把这个文件加入 source，就可以使用这个新的命令 factorial。这个命令接受一个非负整数作为参数，其结果是这个数的阶乘。

```
    factorial 3
⇒   6
    factorial 20
⇒   2432902008176640000
    factorial 0.5
∅   expected interger but got "0.5"
```

这个示例还用到了一点 Tcl 语法：大括号。大括号和双引号有一点相似，它们所包含的内

容中都可以嵌入空格。但大括号和双引号有两点不同。第一，大括号可以嵌套。proc 命令的最后一个词就是左大括号，这组大括号包括的内容从第一行左大括号开始，到最后一行的右大括号为止。Tcl 解释器会移除最外层的大括号，将其中的内容作为参数传递给 proc，包括其中嵌套的大括号。第二，这个过程中不发生替换，而双引号中则照常进行替换。大括号中的所有字符都将原封不动地传递给 proc，这一过程中不会进行任何特殊处理。

proc 命令获取三个参数：过程的名称、用空格分隔的参数名列表以及过程块(过程块实际上就是一个 Tcl 脚本)。proc 会把过程名称作为一个新的命令加入 Tcl 解释器。只要这个命令被调用，就处理这个过程块。在运行这个过程块时，它可以把它的参数作为变量访问：这里的第一个也是唯一一个参数就是 val。

factorial 过程块包括三个 Tcl 命令：set、while 和 return。while 命令完成了这个过程的大部分工作。它获取了两个参数，一个是表达式$val>0，还有一个代码块，这是另一个 Tcl 脚本。while 命令首先计算它的表达式参数，如果其值非零，则处理那个 Tcl 脚本块。它会重复这个过程，直到最终其表达式的值为零。这个示例中，while 块的命令就是把其结果与 val 相乘，然后调用 incr 命令将指定的整型改变量(这里是-1)加到 val 当中。当 val 到达零时，result 所包含的值就是所求的阶乘。

return 命令让过程退出，将变量 result 中的值作为这个过程的结果。如果没有 return 命令，一个过程的返回值就是该过程块中所执行的最后一条命令的返回值。在这个 factorial 中这个值就是 while 的返回值，而该值永远是一个空的字符串。

这个示例中对大括号的使用是关键。在编写 Tcl 脚本时一个很困难的问题就是“替换”，在您需要它的时候保证它发生，在您不需要它的时候保证它不发生。过程块必须用大括号括起来，是因为当过程块作为一个参数传递给 proc 时，我们不希望发生变量替换或命令替换，我们希望这些替换在过程块作为 Tcl 脚本处理的时候发生。while 命令块用大括号括起来的原因是一样的：我们不是希望替换在解析 while 命令的时候发生一次，而是希望每次 while 循环处理循环块的时候都进行替换。while 的参数{$val>0}也需要这个大括号。否则变量 val 的值只在解析 while 命令时替换一次，那样这个 while 就会陷入死循环。您可以试验一下把这个示例中的大括号用双引号代替，看看会出现什么情况。

本书中的示例遵守这样的格式：当 Tcl 脚本作为参数时，左大括号出现在一行的末尾，Tcl 脚本在随后的各行中缩进书写，标志结束的右大括号单独占一行，位于该行的开头。这样可以提高脚本的可读性，不过 Tcl 语法并不要求这样的格式。作为参数的脚本，与其他参数服从同样的语法规则：事实上，在 Tcl 解释器解析参数的时候它根本就不知道这是不是脚本。这种机制的一个后果就是左大括号必须在需要获取该输入参数的命令的同一行。如果左大括号换到了新的一行，那它前面的换行符就标志命令结束。

通常来说，过程中的变量都是局部变量，在过程外是不可见的。在这个 factorial 示例中，局部变量包括参数 val 和变量 result。每次调用一个过程，都会为它创建新的局部变量集(参数会通过复制它们的值的方式完成传入)，而每当一个过程返回时，都会删除它的局部变量。在过程之外命名的变量被称为全局变量，它们会一直存在，直到明确地将其删除。后面会介绍如何在过程中访问全局变量，以及访问其他活动的过程中的局部变量。另

外，还可以指定命名空间，然后创建永久性变量，从而避免变量名冲突。第 10 章将讨论命名空间的使用。

1.6 关于 Tcl 语言

作为一门编程语言，Tcl 和大部分语言差别很大。大多数语言都有称为语法的东西，定义了整个语言。以下面这个 C 语句为例：

```
while (val>0) {
    result *= val;
    val -=1;
}
```

C 语言的语法确定了这个语句的结构，包括一个保留字 while、一个表达式和一个子语句，这个子语句将被一次又一次地执行直到该表达式的值为零。C 的语法确定了 while 语句的结构，以及其表达式与子语句的内部结构。

而 Tcl 没有确定的语法来解释整个语言。Tcl 是由一个解释器定义的，这个解释器解析单个的 Tcl 命令，以及执行许多单个命令的过程集合。解释器和它的替换规则是固定的，但是新的命令可以随时定义，并用来取代已经存在的命令。控制流、过程以及表达式等功能都是通过命令实现的，它们并不能由 Tcl 解释器直接理解。例如，考虑下面这个 Tcl 命令，它和前面那个 while 循环是等价的。

```
while {$val>0}{
    set result [expr $result*$val]
    incr val -1
}
```

在处理这个命令时，Tcl 解释器只知道这个命令有三个词，其中第一个是命令名。Tcl 解释器并不知道 while 的第一个输入参数是表达式，第二个是 Tcl 脚本。完成对这个命令的解析之后，Tcl 解释器会把这个命令中的单词都传给 while，这个命令会把第一个参数作为表达式，把第二个参数作为 Tcl 脚本处理。如果表达式的值非零，那么 while 就会把第二个参数传回 Tcl 解释器进行处理。到了这一步，解释器就会把这第二个参数作为脚本对待(即执行命令，进行变量替换，调用 expr、set 以及 incr 命令)。

现在考虑如下命令：

```
set {$val>0}{
    set result [expr $result*$val]
    incr val -1
}
```

对于 Tcl 解释器来说，set 命令与 while 命令唯一的不同就是它们的命令名不同。解释器处理这个命令的方式与 while 命令一样，唯一的不同之处是执行命令时调用的过程不同。set 命令将其第一个参数作为变量名，第二个参数作为变量的值，因此，这里它会设置一个变量，变量名为$val>0。

Tcl 新手最常见的一个错误就是用语法的观点来理解 Tcl 脚本，这会导致人们认为解释器有很复杂的行为，而事实上并非如此。例如，一位熟悉 C 语言的人第一次使用 Tcl 时可

能会以为 while 后面的两组大括号有不同的用途。事实上，它们并无不同。这两个大括号的用途，都是让 Tcl 解释器把括号内的字符原封不动地传给命令，不要进行任何替换操作。

因此 Tcl“语言”包括的只是十多条用于解析参数和执行替换的简单原则。Tcl 脚本的具体行为是由所执行的命令确定的。命令决定了应该把一个参数视为它所代表的值，还是变量名，或是要执行的代码块等。这样做的一个有意义的结果是，脚本可以为命令定义全新的控制结构，这一功能是大多数语言不具备的。

1.7 事件绑定

下面这个示例为 factorial 过程提供一个图形化的前台。除了介绍两个新的组件类型，还将演示 Tcl 的绑定机制。绑定，使得特定窗口中特定事件发生时，就执行特定的 Tcl 脚本。按钮的-command 选项就是由特定组件类型实现的简单绑定。Tk 还拥有更通用的机制，几乎可以随意扩展组件的行为。

将如下脚本写入 factorial.tcl 文件，然后从命令外壳中调用该文件，运行这个示例。

```
#! /usr/bin/env wish
    proc factorial {val} {
        set result 1
        while {$val>0} {
            set result [expr $result*$val]
            incr val -1
        }
        return $result
    }
    entry .value -width 6 -relief sunken -textvariable value
    label .description -text "factorial is"
    label .result -textvariable result
    button .calculate -text "Calculate" \
        -command {set result [factorial $value]}
    bind .value <Return> {
        .calculate flash
        .calculate invoke
    }
    grid .value .description .result -padx 1m -pady 1m
    grid .calculate - - -padx 1m -pady 1m
```

这个脚本会显示图 1.2 所示的图形界面。这里有一个输入组件，您可以用鼠标单击它然后输入一个数字。如果单击名为“Calculate”的按钮，求阶乘的结果就会出现在窗口右侧；如果在输入框中按下 Return 键，也会出现同样的结果。

这个应用程序有 4 个组件：一个输入组件、一个按钮和两个标签。输入组件可以显示一行输入的文本。这里它的-width 选项设置为 6，意思是它的大小可以显示 6 位数字；-relief 选项设置为 sunken，意思是这个输入组件向内嵌入窗口。选项-textvariable 为输入组件指定一个全局变量，用来保存输入的内容——您在输入组件的输入框中所做的任何改变都会在这个全局变量中反映出来，反之亦然。

标签组件.description 控制说明文本，.result 标签则控制计算结果。.result 的选

项-textvariable 会显示设定的全局变量中的字符串，并在全局变量被改变时即时更新。与此不同，.description 显示的是一个常量字符串。

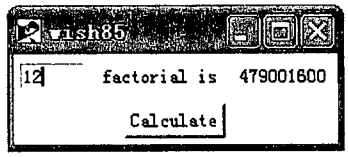


图 1.2 计算阶乘的图形化用户界面

第一条 grid 命令在第一行从左到右排列输入组件和两个标签组件。选项-padx 和-pady 可以优化显示效果，显示时在各组件的左右各加上 1 mm 空白，在各组件的上下各加上 1 mm 空白。这里的设定值的后缀 m 表示毫米，还可以用 c 表示厘米，用 i 表示英寸，用 p 表示磅，或者不加后缀表示像素。

第二条 grid 命令在第二行排列了一个按钮。因为这个组件的名字是第一个参数，栅格管理器会把第二行第一列这个位置分配给该按钮组件。组件名后面的两个-参数，向栅格管理器说明分配给这个按钮组件的显示位置需要再加两列。栅格管理器会把按钮组件排列在分配给它的区域的正中。

这个脚本中创建.calculate 按钮的命令占了两行；第一行最右端的反斜线就是续行符，它使解释器将这两行视为一行。按钮的-command 脚本将用户接口链接到 factorial 过程。这个脚本调用 factorial，将输入组件中得到的值传给它，将结果存储在 result 变量中，然后由.result 组件显示出来。

命令 bind 有三个参数：组件名、事件说明以及指定组件中指定事件发生时调用的 Tcl 脚本。<Return>指定的事件为用户通过键盘输入回车(在“Mac”键盘中这个键的名称仍然是 Return，不过在大多数英语键盘上这个键的名称是“Enter”)。表 1.1 展示了其他一些可能有用的事件说明符。

表 1.1 事件说明符

事件说明符	含 义
<Button-1>	按下 1 号鼠标键
<1>	<Button-1>的简写
<ButtonRelease-1>	释放 1 号鼠标键
<Double-Button-1>	双击 1 号鼠标键
<Key-a>	按下 a 键
<a>或 a	<Key-a>的简写
<Motion>	鼠标移动了，无论是否有键被按下
<B1-Motion>	1 号键按下时鼠标移动

进行绑定的脚本会访问有关事件的各方面信息，例如当事件发生时鼠标所在的位置。例如：交互式地启动 wish，然后输入如下命令：

```
bind . <Motion> {puts "pointer at %x,%y"}
```

现在在窗口中移动鼠标，每次鼠标移动之后，在标准输出端都会输出它的新坐标。当鼠标移动这个事件发生时，Tk 会扫描脚本中的%标记，将其替换为相关事件的信息之后，再将脚本传给 Tcl 进行处理。%x 会替换为鼠标的 *x* 坐标，%y 会替换为鼠标的 *y* 坐标。

这里绑定的目的在于用指定的应用程序行为，扩展输入组件的内建行为(不仅能编辑文本字符串)。在这个脚本中，为了方便，我们希望可以让用户直接在输入框中按下 Return 键，进行阶乘计算，而不必单击 Calculate 按钮。我们只需把按钮的命令脚本再复制一遍，但是如果我们以后修改了这个按钮的命令脚本，我们就必须记得把绑定脚本的内容再修改一遍。因此，我们将绑定脚本处理为“由程序来”单击按钮。

绑定脚本执行的两个命令称为组件命令。只要创建了一个新的组件，一个与该组件同名的 Tcl 命令也就创建了，您可以调用这个命令来与该组件通信。组件命令的第一个参数是某一个选项，后面的其他参数是该选项所需的变量。这个脚本中第一条组件命令是让按钮闪烁。(您不一定能看到按钮闪烁，这和计算机系统的显示方案有关。)第二条组件命令让按钮组件调用它自己的-command 选项，就如同您用鼠标单击按钮一样。

不同类型的组件的组件命令支持不同的选项集，但不同类型之间也有很多选项是相似的。例如，所有的组件都支持一个组件命令 configure，用来修改组件的设置选项。如果交互式地运行 factorial.tcl 脚本，可以用如下命令将输入组件的背景改为黄色：

```
.value configure -background yellow
```

或者可以输入：

```
.calculate configure -state disabled
```

让用户界面上的这个按钮无法响应。

1.8 Tcl 和 Tk 的更多功能

本章的示例已经用到了 Tcl 语言的各个方面，展示了 Tcl 和 Tk 的很多功能。然而，Tcl 和 Tk 还有很多有用的功能本章未能演示，本书接下来将描述所有这些功能。下面列出本章未能演示但很有用的一些功能。

- 数组、字典和列表——Tcl 提供了关联数组和字典，用于高效率地存放键/值对；提供了列表，用于管理数据集合。
- 更多的控制结构——Tcl 提供了更多的命令，用于管理执行流程，如 eval、for、foreach 以及 switch。
- 字符串操作——Tcl 包含了很多操作字符串的命令，如长度测量，正则表达式的模式匹配与替换，以及格式转换。
- 文件访问——您可以通过 Tcl 脚本读取和写入文件，获取字典信息，获取文件大小、创建时间等文件属性。
- 更多的组件——除了这里演示的这些组件，Tk 还有很多的组件类型，例如菜单、滚动条、名为 canvas 的绘画组件，以及便于制作超文本效果的文本组件。

- 访问其他的窗口功能——Tk 提供了用于访问主要的窗口工具的命令，例如，与窗口管理器通信的命令(例如设置窗口标题)，用于追溯选择的命令，用于管理输入焦点的命令。
- 应用程序间的通信——通过进程间的管道和 TCP/IP 套接字，Tcl 具有进行应用程序间通信的能力。
- C 语言接口——Tcl 提供 C 语言库，可以用 C 语言定义新的 Tcl 命令。(Tk 提供了一个库，可以用 C 语言来创建新的组件类型以及图形管理器，不过这个功能使用得非常少，本书对此不做更多介绍。)

第2章 Tcl语言的语法

要编写 Tcl 脚本，必须学会两件事。第一，必须学会 Tcl 的语法，其内容是决定如何解析命令的十多条规则。第二，必须掌握在脚本中使用的独立命令。Tcl 提供大约 100 条内建命令，Tk 另外提供了几十条，而且所有基于 Tcl 和 Tk 的应用程序都很可能会有一些自己独有的命令。您现在就需要掌握这些语法规则，但是命令可以根据需要逐步学习。

本章讲述 Tcl 语言的语法。第 I 部分其余各章将讲述 Tcl 的内建命令，第 II 部分将讲述 Tk 提供的命令。

2.1 脚本、命令和单词

Tcl 脚本包含一条或更多的命令。命令通过换行符或分号隔开，例如：

```
set a 24
set b 15
```

上述代码就是由两条命令组成的脚本，命令间由换行符分隔。这个脚本也可以写成一行，用分号把命令隔开。

```
set a 24; set b 15
```

每一条命令都包含一个或多个单词，第一个单词是命令名，其他的单词是命令的参数。单词通过空格或制表符隔开，这两种符号常被通称为空白符或空白。上面这个例子中每条命令都有三个单词。一条命令中可以有任意多个单词，而每个单词都可以是任意的字符串值。分隔单词的空白不是单词的一部分，分隔命令的换行符和分号也不是单词的一部分。

2.2 处理命令

如图 2.1 所示，Tcl 处理一个命令分两步：解析和执行。在解析这一步，Tcl 解释器应用本章讲述的规则，将命令分解为单词，并执行替换。对每个命令进行解析的方法是完全一样的。在解析阶段，Tcl 解释器不认为各单词的值有任何具体意义。Tcl 只是进行一系列简单的字符串操作，例如，将字符串$input 用 input 变量中存放的字符串代替；Tcl 不知道也不关心替换后的字符串是一个数、一个组件名称还是其他内容。

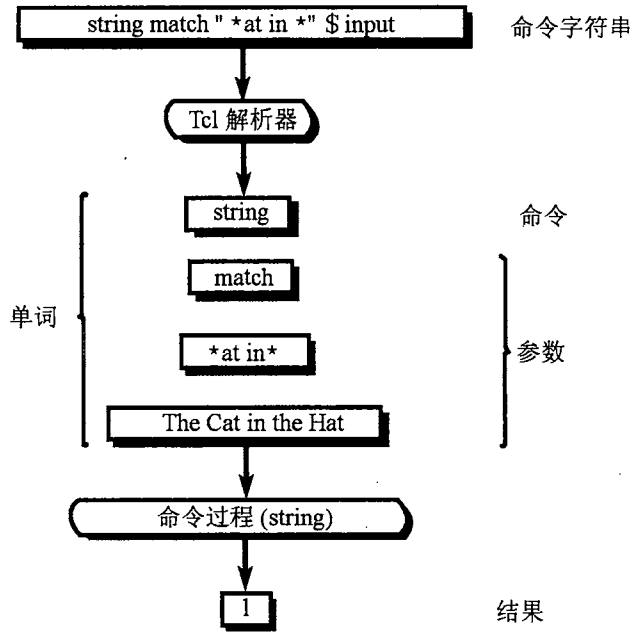


图 2.1 Tcl 命令的解析和执行

在执行阶段，命令中的各个单词都有了具体的意义。Tcl 把第一个单词作为命令名称，检查这个命令是否已经定义，并且查找完成该命令功能的命令过程。如果命令已经定义，Tcl 解释器就调用该命令过程，把命令中的全部单词传递给该过程。命令过程会根据自己的需要来分辨这些单词的意义；因为各个命令要完成的功能不同，各个命令所用的参数的意义自然也各不相同。

提示： Tcl 社区使用“单词”和“参数”两个词指代传递给命令过程的值，这两个词是可以互换的，唯一的不同在于第一个参数就是第二个单词。

下面这些命令展示了一些常见的参数的意义。

- `set a 122`
  在很多情况下，例如在 set 命令中，参数可以是任意形式的。set 命令就是将第一个参数作为变量名，第二个参数作为变量的值。命令 set 122 a 也是有效的：它创建一个名为 122 的变量，其值是 a。
- `expr 24 / 3.2`
  expr 命令把它的参数连接起来，其结果必须是一个数学表达式，并满足第 4 章所讲述的规则。其他一些命令也接受表达式作为输入参数。
- `lindex {red green blue purple} 2`
  lindex 的第一个参数是包含 4 个值的一个列表，这些值由空格隔开。该命令返回列表中索引为 2 的元素值(实际上是第 3 个元素 blue，索引是从零开始编号的)。第 6 章将讲述 Tcl 有关列表操纵的命令。

- `string length abracadabra`
  有一些命令，如 string 以及 Tk 组件命令，实际上是把多条命令整合为一条。命令的第一个参数决定要进行的操作以及其他参数的含义。例如 string length 需要另外一个参数，然后计算它的长度，而 string compare 还需要另外两个参数。这样的整合式命令也称为集合命令。
- `button .b -text Hello -fg red`
  .b 后面的参数即选项-值对，用来设定您所关心的选项，未被设定的选项将使用默认值。

在编写 Tcl 脚本的时候，要记住很重要的一点，Tcl 解析器在解析命令的时候，并不会为其中的单词赋予任何含义。这些单词的含义决定于具体的命令过程，而不是 Tcl 解析器。这种）方法与大多数壳语言相近，而与大多数编程语言不同。例如，考虑下面这段 C 程序代码：

```
x = 4;
y = x+10;
```

第一个 C 语句将整型值 4 存储在变量 x 中。第二个 C 程序计算表达式 x+10 的值，取得变量 x 的值然后再加上 10，然后把这个值存储在变量 y 中。在运行完成后，y 中有一个整型值 14。如果想在 C 程序中使用文字字符串而不是对其取值，需要用双引号把字符串括起来。现在来看一下用 Tcl 编写的类似程序：

```
set x 4
set y x+10
```

第一条命令将字符串 4 赋给变量 x。而变量的值没有被指定为任何特定形式。第二个命令直接把字符串 x+10 作为新的变量 y 的值进行存储。结果，y 的值为字符串 x+10，而不是整型值 14。在 Tcl 中，如果要对一个表达式求值，必须进行明确要求。

```
set x 4
set y [expr $x+10]
```

第二条命令中一共要求进行两次求值。第一次，该命令的第二个参数被括在方括号中，这对方括号告诉 Tcl 解析器将其中的单词作为 Tcl 脚本处理，并把其结果作为这个单词的值。第二次，在 x 之前放置了一个$符号。当 Tcl 解析 expr 命令时，它会把$x 替换为变量 x 的值。如果没有这个$符号，expr 的参数会是一个包含 x 的字符串，最后导致语法错误。结果，y 的值为字符串 14。

2.3 变量替换

Tcl 提供了三种替换形式：变量替换、命令替换以及反斜线替换。每种替换都会把单词中的一些原始字符替换为另外一些值。Tcl 解释器在执行命令过程之前进行这些替换。替换可以发生在命令中的任何一个单词上，包括命令名本身，在一个单词中也可以进行任意多个替换。

提示：替换并不影响命令中每个单词的分隔，即使替换后的字符中包括空格、制表符、换行符等空白字符，也不会影响。

替换的第一种形式是变量替换。它由$符号引发，将 Tcl 变量的值插入单词中。例如，考虑如下命令：

```
set kgrams 20
expr $kgrams*2.2046
⇒ 44.092
```

第一条命令将变量 kgrams 的值设置为 20。第二条命令将 kgrams 的值乘以 2.2046，计算出相应的磅数。这里就用到了变量替换：字符串$kgrams 替换为变量 kgrams 的值，所以命令 expr 接收到的参数实际上是 20*2.2046。

变量替换可以在一个单词中的任何位置进行，可以进行任意多次，例如：

```
expr $result*$base
```

变量名由$符号后面所有的数字、字母以及下划线组成。因此第一个变量名 result 到星号为止，第二个变量名就是 base。变量替换可以用于各种目的，例如产生新的名称。

```
foreach num{1 2 3 4 5} {
    button .b$num
}
```

这个示例创建了 5 个按钮组件，分别名为.b1、.b2、.b3、.b4 以及 .b5。

这些示例是变量替换最简单的形式。关联数组使用了另外两种变量替换形式，这两种形式可以提供对变量名更加直接的控制(例如，在变量名后可以立即跟一个字母或数字)。这些形式在第 3 章中有所讨论。

2.4 命令替换

Tcl 提供的第二种替换形式是命令替换。命令替换可以把一个单词的部分或全部替换为一个命令的结果。命令替换通过方括号表示，会调用括号中的命令。

```
set kgrams 20
set lbs [expr $kgrams*2.2046]
⇒ 44.092
```

方括号内的字符必须构成有效的 Tcl 脚本。脚本可以包含任意多条命令，命令之间用换行符隔开，也可以用分号隔开。括号和它们之间的所有字符会被替换为脚本的结果。因此，在前面这个示例中，命令 expr 会在解析 set 的单词时执行；它的结果，即字符串 44.092，会成为 set 的第二个参数。和变量替换一样，命令替换也可以在单词的任何位置进行，在一个单词中可以有不止一处命令替换。

2.5 反斜线替换

Tcl 提供的最后一种替换方式是反斜线替换。用于向单词中插入像换行符这样的特殊字符，以及像[、$这样的会被 Tcl 解析器认为是有特殊含义的字符。例如下面的命令：

```
set msg Eggs:\ \$2.18/dozen\nGasoline:\ \$2.49/gallon
⇒ Eggs: $2.18/dozen
  Gasoline: $2.49/gallon
```

这里有两处反斜线后面跟着空格，在单词中这会被替换为一个空格，而这个空格符不会被视为单词分隔符。还有两处反斜线后面跟着$符号，在单词中这会被替换为一个$符号，而这个$符号会被作为普通字符处理(它们不会触发变量替换)。在反斜线后跟着 n，会被替换为换行符。

表 2.1 列出了 Tcl 支持的所有反斜线序列。这包括了 ANSI C 定义的全部序列，如\t 插入一个制表符，\x7d 插入一个十六进制值为 0x007d 的 Unicode 字符。(第 5 章将深入讨论 Unicode 编码。)如果反斜线后面跟的不是这个表中列出的字符，如\$和\[，反斜线就会从单词中移除，后面的字符作为普通字符处理。这使得您可以在单词中包含任何 Tcl 的特殊字符，而不会触发 Tcl 解析器的特殊处理。序列\\就向单词中插入了一个反斜线。而反斜线后面的空格也会被作为一个普通的空格字符处理，而不是单词分隔符。

表 2.1 Tcl 支持的反斜线序列

反斜线序列	替换结果
\a	警告音(0x7)
\b	删除(0x8)
\f	换页符(0xc)
\n	换行符(0xa)
\r	回车(0xd)
\t	制表符(0x9)
\v	垂直制表符(0xb)
\ooo	八进制值为 ooo(一个、二个或三个 o)的 8 位 Unicode 字符
\xhh	十六进制值为 hh 的 8 位 Unicode 字符(可以有任意个 h，不过除了最后两个以外都会被忽略)
\uhhhh	十六进制值为 hhhh 的 16 位 Unicode 字符(1～4 个 h)
\newline whitespace	一个空格字符

反斜线接换行符可以用于将一条很长的命令写成几行，示例如下：

```
pack .base .label1 .power .label2 .result \
    -side left -padx 1m -pady 2m
```

反斜线加换行符再加下一行开头的空白，会被替换为一个空格符。这样两行就一起组

成了一条命令。

提示： 反斜线-换行符序列和一般的替换不同，这种替换在 Tcl 解释器解析命令前就要单独进行。这意味着，用于替换反斜线-换行符的空格符会被作为单词分隔符看待，除非它们被双引号或大括号括起来。

2.6　双引号引用

Tcl 提供了一些方法，可以阻止解析器对$和分号等字符进行特殊处理。这些方法称为引用。您已经见过了反斜线替换中的引用方式。例如，/$就向单词中插入一个普通的$字符，而不会引发变量替换。除了反斜线替换外，Tcl 还提供了另外两种引用形式：双引号引用和大括号引用。双引号取消其中的单词和命令分隔符的特殊解释，大括号取消其中所有特殊字符的特殊解释。

如果一个单词的第一个字符是双引号，那么这个单词就由另一个双引号标记结束。注意双引号本身并不是单词的一部分，它们就是定界符而已。如果一个单词包含在双引号中，那么其中的空格、制表符、换行以及分号都作为普通字符处理。前面的示例可以用双引号更清楚地重写如下：

```
    set msg "Eggs: \$2.18/dozen\nGasoline: \$1.49/gallon"
⇒   Eggs: $2.18/dozen
    Gasoline: $1.49/gallon
```

示例中的\n 也可以用一个真正的换行符取代，写成如下形式：

```
set msg "Eggs: \$2.18/dozen
Gasoline: \$1.49/gallon"
```

不过多数人认为使用\n 会增加脚本的可读性。

如果单词不是以双引号开头，那么单词中的任何双引号都会作为普通字符处理，而不是定界符。例如，下面这个示例也是合法的，把双引号作为文本字符使用，这里双引号没有任何特殊含义。

```
    puts This "is" poor"usage
⇒   This"is"poor"usage
```

变量替换、命令替换以及反斜线替换在双引号中正常进行。例如，下面这段脚本就把 msg 设置为包含一个变量名、该变量的值、该变量值的平方等信息的字符串。

```
    set a 2.1
    set msg "a is $a; the square of a is [expr $a*$a]"
⇒   a is 2.1; the square of a is 4.41
```

如果想要在由双引号括起来的单词中包含双引号字符，则应该使用反斜线替换。

```
    set name a.out
    set msg "Couldn't open file \"$name\""
⇒   Couldn't open file "a.out"
```

2.7 大括号引用

大括号提供了更彻底的引用形式，它会取消其中所有特殊字符的特殊意义。如果一个单词以左大括号开头，那么直到与它配对的右大括号为止，所有字符都将被原封不动地识别为这个单词的值。这个单词中不会发生任何替换，所有的空格、制表符、换行符以及分号都作为普通字符处理。2.5 节的示例可以用大括号重写如下：

```
set msg {Eggs: $2.18/dozen
Gasoline: $1.49/gallon}
```

这个单词中的$符号不会触发变量替换，换行符也不会被视为命令分隔符。这里就不能使用\n 来向单词中插入换行符了，因为\n 会被作为单词中的字符对待，而不会触发反斜线替换。

```
    set msg {Eggs: $2.18/dozen\nGasoline: $1.49/gallon}
⇒   Eggs: $2.18/dozen\nGasoline: $1.49/gallon
```

提示： 在大括号中会执行替换的唯一替换形式是反斜线-换行符序列。正如 2.5 节所述，这一替换实际上是在解析命令之前进行的。

和双引号不同，大括号可以嵌套使用。在过程定义和命令流控制中，经常会用到嵌套的大括号来把一个或多个参数表达为需要处理的脚本。因为某段脚本需要作为一个参数出现，它就必须被大括号括起来。而在这个作为参数的脚本中，又可能包含其他的控制流命令，这些命令的一些参数也可能是脚本。

提示： 如果一个大括号放置在反斜线后，它就不加入大括号的配对。直到这个单词被解析后，这个反斜线才会被移除。

大括号最重要的应用之一就是“延期处理”。延期处理意味着特殊字符不会被 Tcl 解析器立即处理。它们会被作为参数传递给命令过程。命令过程自己处理这些特殊字符，实际上多数时候就是把这些参数传给 Tcl 解释器进行处理。例如下面这个过程，它统计在列表中某个特定值出现的次数。

```
proc occur {value list}{
    set count 0
    foreach el $list{
        if $el==$value{
            incr count
        }
    }
    return $count
}
```

整个过程块都括在大括号中，它会被原封不动地传递给 proc。在解析 proc 命令时变量 list 的值不会被替换进来。如要这个过程正常工作，这一点就是必需的：在每次调用这个过程时$list 都需要读入不同的值。注意大括号嵌套，proc 的最后一个参数包括的内容直到最后一个大括号为止。图 2.2 演示了下面这句 Tcl 脚本运行时会发生什么。

```
occur 18 {1 34 18 16 18 72 1994 -3}
```

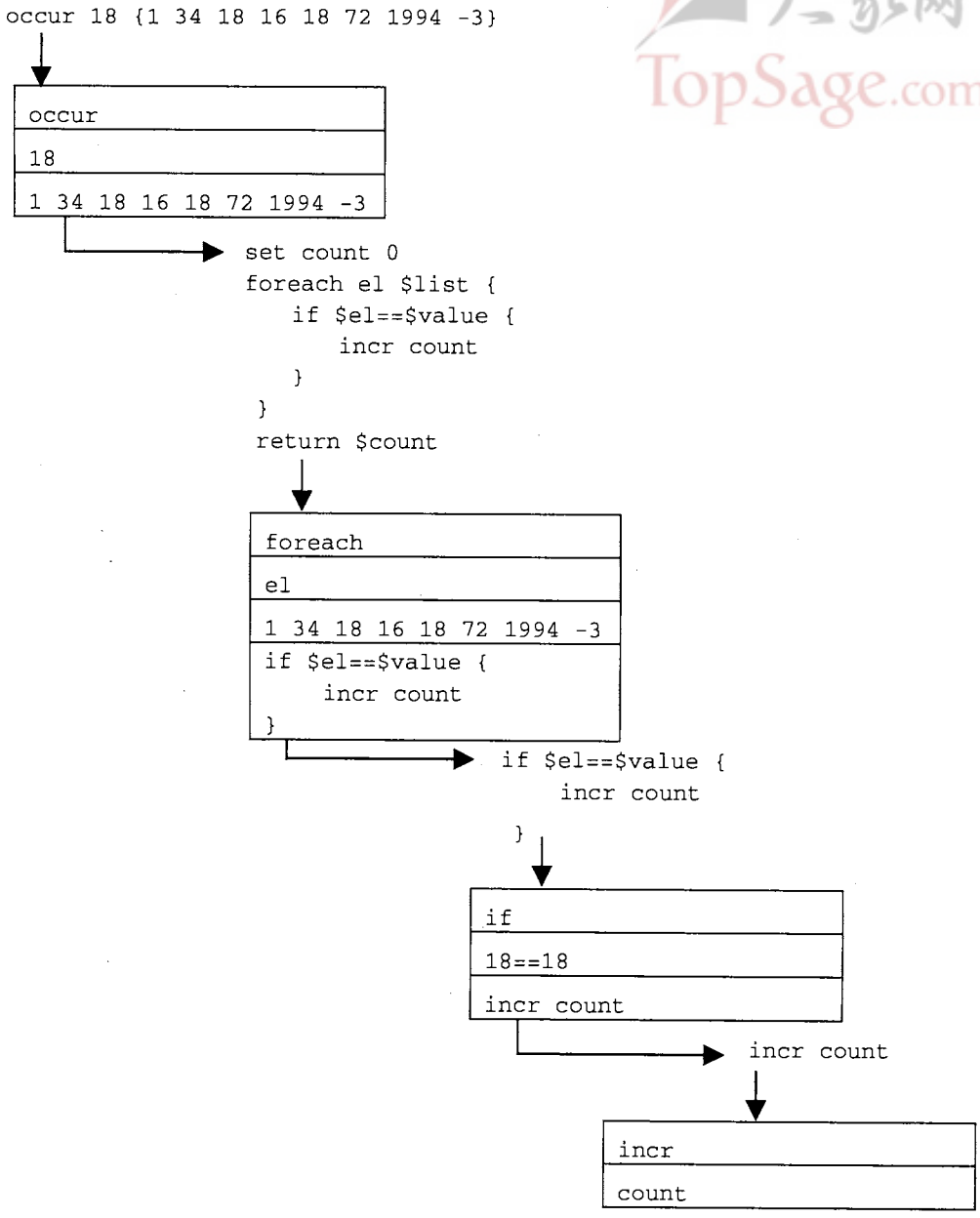


图 2.2 处理嵌套脚本的快照

occur 的第二个参数由大括号引用，这会把整个数字列表作为一个单词传给 occur。Tcl 过程管理机制又会把整个过程块传给 Tcl 解释器进行处理。在 Tcl 解析该过程块中的 foreach 命令时，变量 list 的值就会替换进来，但这时 foreach 的最后一个参数(循环体)还在一个大括号当中，此时不会进行替换。foreach 命令过程会把变量 el 依次赋值为列表中的每一个元素，调用 Tcl 解释器对每一个元素运行这个循环体。在运行循环体时，Tcl 解析 if 命令，替换变量 el 和 value 的值。在图 2.2 中，if 判断通过，处理了 incr 命令。

2.8 参数展开

2.7 节的示例展示了将一个列表作为参数传递给过程。在命令行中列表内容是直接给出的。通常来说，列表的值可能是变量或命令替换的结果。但是因为 Tcl 解释器是从左向右进行替换的，而列表的值被视为一个单词，即列表值中嵌入的空格不会被视为单词分隔符。

在某些情况下，单层替换规则有害无益。例如下面这个示例，这个脚本就不能完成删除所有以.o 结尾的文件的任务。

```
file delete [glob *.o]
```

glob 命令返回的是符合*.o 形式的文件名列表，例如 a.o b.o c.o。然而，整个文件名列表是作为一个参数传递给 file delete 的，file delete 会因为找不到名为 a.o b.o c.o 的文件而失败。要想 file delete 正常工作，glob 的结果必须被正确分隔为多个单词。可以通过参数展开来完成这一任务。

如果一个单词以字符串{*}开头，之后紧接着非空白字符，Tcl 会移除开头的{*}，把该单词的剩余部分作为含有单词分隔符的语句进行解析与替换。在替换之后，Tcl 会再次解析这些单词，但不进行替换，校验确定它们的确是一个或多个语法完整的单词。如果校验通过，这些单词会被独立地加入命令行进行处理；否则，Tcl 会报语法错误。因此，

```
file delete {*}[glob *.o]
```

在由 Tcl 解释器进行解析和参数展开后，与下面这条语句相同：

```
file delete a.o b.o c.o
```

语法{*}首次出现在 Tcl 8.5 中。更早的版本则要求您在需要时明确添加解析层次。记得 Tcl 的命令是分两个阶段进行处理的：解析和执行。替换规则仅适用于解析阶段。一旦 Tcl 将命令所需的单词传递给命令过程，进入执行阶段，命令过程就可以进行它们所需要的任何处理。一些命令会再解析它们的单词——例如，把这些单词重新传给 Tcl 解释器。eval 就是这样一个命令，可以用它来解决刚才 file delete 遇到的问题。

```
eval file delete [glob *.o]
```

eval 把它的所有单词连接起来，单词之间用空格隔开，然后把这个结果作为 Tcl 脚本处理，也就是再进行了一轮解析和运行。这个示例中，eval 接收到的单词为 file、delete 以及 a.o b.o c.o。它把它们连接为字符串 file delete a.o b.o c.o。当这个字符串再由 Tcl 解释器处理时，它会生成 5 个单词。每一个文件名都会作为独立的参数传递给 file delete，从而成功删除这些文件。eval 命令的正确使用详见 8.6 节。

2.9　注释

如果一条命令的第一个非空白字符是#，那么这一行将被视为注释而忽略。注意注释符必须出现在 Tcl 预期将获得命令的第一个字符的位置上。如果注释符出现在其他地方，会被看作一个普通字符，看成一个命令单词的一部分。

```
    # This is a comment
    set a 100                # Not a comment
Ø   wrong #args: should be "set varName ?newValue?"
    set b 101                ;# This is a comment
⇒   101
```

第二行的符号#就没有被视为注释符，因为它出现在一条命令的中间。其结果是，第一个命令 set 接收到 6 个参数，产生了错误。最后一个#就被视为注释符，因为它紧接在标记一条命令中止的分号后面。

Tcl 的语法虽然简单而统一，但也可能产生一些意料之外的结果。它要求作为注释符的#必须出现在预期将获得命令的第一个字符的位置上，这就导致下面这个示例中的#行不是注释。

```
set example {
    # This is not a comment
}
```

大括号之间的所有字符都视为一个参数处理，作为字符串值赋给 set 指定的变量。相反，考虑下面这个示例：

```
if {$x < 0}{
    # This is a comment
    puts "The result is negative."
}
```

这里，Tcl 解析器把两个大括号里的内容作为两个参数传给 if 命令。if 命令会把第一个参数视为布尔型表达式处理，如果结果为真，它就调用 Tcl 解释器将第二个参数作为 Tcl 脚本处理。在 Tcl 解释器再次进行解析时，以#开头的这一行才被识别为注释。

Tcl 的一致语法引起的另一个后果是，在您运行代码时，出现在注释中的大括号常常会导致错误。考虑下面这个示例：

```
proc countdown {x}{
    puts "Running countdown"
    # Incorrectly comment out this code block {
        while {$x >= 0}{
            puts "x = $x"
            incr x -1
        }
    }
}
```

当 Tcl 解释器解析这个命令时，第一行末尾的左大括号标记着一个单词的开始。标记单词结束的右大括号出现在最后一行。这个单词中包含着什么脚本与这一步解析无关，Tcl 解释器现在只是把命令行解析为单词集，对单词中嵌套的其他大括号只作为普通字符对

待。然后 Tcl 将单词传递给 proc 命令，该命令会把这些单词作为 countdown 过程的实现脚本存储。

当您试图运行 countdown 过程时，Tcl 会解析并运行这个脚本。首先执行 puts 命令，然后 Tcl 忽略了整个第二行(包括那个左大括号)，将其视为注释。然后处理 while 命令，它的脚本参数由 incr 命令后面那个右大括号标记结束。然后 Tcl 期待找到下一条命令，却在该行开头获取到右大括号，它会将这个右大括号视为要处理的命令名，结果导致如下错误：

```
    countdown 3
⇒   Running countdown
    x = 3
    x = 2
    x = 1
    x = 0
Ø   invalid command name "}"
```

提示：一般来说，尽量避免在注释中出现大括号。如果您确实需要在注释中使用大括号，确保它们是配对的；对于每一个左大括号，都保证有一个正确配对的右大括号。

下面这个示例展示了正确注释的一段代码。注释中的大括号是正确配对了的。

```
proc countdown {x} {
    puts "Running countdown"
    #  while { $x >= 0} {
    #    puts "x = $x"
    #    incr x -1
    # }
}
```

要把一大段格式正确、可以解析的代码段注释掉，可以如下使用 if 命令：

```
proc countdown {x} {
    puts "Running countdown"
    if 0 {
        while {$x >= 0} {
            puts "x = $x"
            incr x -1
        }
    }
}
```

2.10 正常返回和异常返回

Tcl 命令可能以多种方式完成。正常返回是最常见的情况，这意味着命令正常执行完成，返回了一个字符串值。Tcl 还支持命令的异常返回。错误是异常返回的最常见形式。当错误返回产生时，就意味着命令未能完成所期望的功能。该命令被放弃，它后面的脚本中的命令被略过。一个错误返回包含说明错误的字符串值，这个字符串通常由应用程序显示。例如，下面这条 set 命令因参数过多产生了一个错误。

```
    set state West Virginia
Ø   wrong # args: should be "set varName ?newValue?"
```

不同的命令产生错误的条件不同。例如，expr 可以接受任意数量的参数，但要求这些参数满足特定的语法。例如，它会在圆括号不能配对的情况下产生错误。

```
    expr 3 * (20+4
Ø   unbalanced open paren
    in expression "3 * (20+4"
```

Tcl 异常返回机制的完整介绍可参见第 13 章。除了错误以外，Tcl 还支持很多其他的异常返回，除了前面提到的错误消息之外，还可提供有关错误的更多信息，允许错误被“捕获”从而允许 Tcl 代码处理错误。不过现在您应该知道的是：命令通常会返回字符串结果，但它们有时候会返回错误，导致 Tcl 命令解释器放弃处理。

提示：您会发现全局变量 errorInfo 十分有用。出现错误后，Tcl 会把 errorInfo 设置为一个栈，保存产生错误的确切位置。您可以用 puts $errorInfo 输出这个变量的值。

2.11 有关替换的更多信息

Tcl 初学者有一个常见的困难，就是理解替换何时发生，何时不发生。脚本出现令人奇怪的行为，常常是因为希望进行替换时替换没有发生，或是不希望进行替换时替换却发生了。然而，您应该发现 Tcl 的替换机制是很简单的，其行为是很容易预测的，只需您记住以下两条相关规则。

(1) Tcl 解析一条命令时，只从左向右解析一次，进行一轮替换。每一个字符只会被扫描一次。

(2) 每一个字符只会发生一层替换，而不会对替换后的结果再进行一次扫描替换。

如果有 Unix 外壳编程经验，会发现 Tcl 的替换规则更加简单，更具一致性。初学者在应用 Tcl 替换进行编程时，通常问题是出在他们以为解释器进行了很复杂的操作，而实际上解释器的工作很简单。

例如，考虑如下命令：

```
    set x [format {Earnings for July: $%.2f} $earnings]
⇒   Earnings for July: $1400.26
```

在命令替换中，方括号之间的字符被扫描一次，只是一次，earnings 变量的值完成了变量替换。Tcl 并不是首先扫描整个 set 命令，进行变量替换，然后再进行一次传递以完成命令替换；所有的事情只发生在一次扫描中。format 命令的结果原封不动地传递给 set，作为它的第二个参数，而没有进行更多的处理(例如，format 语句中的$符号就没有触发变量替换)。

这些替换规则的一个后果是，除非您使用 2.8 节介绍的{*}语法，否则，在命令语句中替换进来的空白并不会分隔单词。例如下面这段脚本：

```
set city "Los Angeles"
set bigCity $city
```

第二条 set 命令一定是三个单词，无论 city 变量的值是什么。这里 city 的内容有一个空格符号，但这个空格不会被视为单词分隔符。

最后注意一点：替换可以被用得十分复杂，但应该避免这种情况。替换最好的用法就是最简单的用法，例如 set a $b。如果在同一条命令中使用太多的替换，特别是如果使用太多反斜线，代码会变得不可读，也不可靠。在那种情况下，为了清楚和安全起见，最好把那条繁琐的命令分解为多条命令，以更简单的层次关系管理参数。Tcl 提供了一些命令，如 format、subst 以及 list，用于协助处理复杂的替换情况。另一个值得考虑的方法是建立“过程”，将复杂的操作隔离开来，这样编写的脚本通常比“内联”所有操作的脚本更易读，更容易维护。

第3章 变 量

Tcl 支持两种类型的变量：简单变量(通常也称为数量变量)和关联数组。本章讲述用于变量管理和数组管理的基本 Tcl 命令，还提供了关于变量替换的更完整的说明。

3.1 本章出现的命令

本章讨论的用于变量管理的基本命令如下所述。

- `append varName value ?value...?`
  将每一个 *value* 参数的值依次添加到变量 *varName* 中。如果 *varName* 变量不存在，则在添加前新建一个名为 *varName* 的空变量。返回值是 *varName* 变量的新值。
- `incr varName ?increment?`
  将变量 *varName* 的值加上 *increment*。*increment* 的值和 *varName* 的原有值都必须是整数字符串(十进制、十六进制或八进制)。如果没有设置 *increment* 参数，其默认值为 1。新的值以十进制字符串形式存储在 *varName* 中，并作为该命令的结果返回。就 Tcl 8.5 而言，对一个不存在的变量进行增加操作，会创建该变量，并将其设置为增量值。
- `set varName ?value?`
  如果指定了 *value*，则将变量 *varName* 的值设置为 *value*。任何情况下这个命令都将返回变量 *varName* 的新值。
- `unset ?-nocomplain? ?--? varName ?varName varName...?`
  删除 *varName* 参数指定的变量。返回一个空字符串。如果要求删除不存在的变量，而且没有设置-nocomplain 选项，就会产生错误。
- `array exists arrayName`
  返回一个布尔型值，反映名为 *arrayName* 的数组是否存在。
- `array get arrayName ?pattern?`
  返回包含名为 *arrayName* 的数组的内容的字典。如果设定了 *pattern*，则只将在 string match 规则下符合 *pattern* 指定的形式要求的内容加入字典。
- `array names arrayName ?mode? ?pattern?`
  返回名为 *arrayName* 的数组的元素名列表。如果设定了 *pattern*，则只返回在 string match 规则下符合 *pattern* 指定的形式要求的元素名。如果给定了*-mode*，其值应该是-exact、-glob 或-regexp，分别代表在用 *pattern* 选定元素名时遵守严格匹

配(如 string equal)、通配符式匹配(如 string match，即默认规则)或正则表达式匹配(如 regexp)。

- `array set arrayName dictionary`
  将 *dictionary* 的内容并入名为 *arrayName* 的数组中。返回空字符串。
- `array size arrayName`
  返回名为 *arrayName* 的数组中的元素个数。
- `array statistics arrayName`
  返回有关名为 *arrayName* 的数组的内部设置的描述。
- `array unset arrayName ?pattern?`
  移除名为 *arrayName* 的数组中所有符合 *pattern*(遵照 string match 规则)的元素，返回空字符串。如果没有设置 *pattern*，则将整个数组设置为空。

3.2 简单变量和 set 命令

一个 Tcl 简单变量包含一个名字和一个值。名字和值都可以是任意字符串。例如，变量名可以设为 xyz !# 22 或 March earnings: $100,472。在实际应用中，变量名通常以一个字母开头，由一些字母、数字和下划线组成，这样在使用变量替换时可以更方便一些。变量名和 Tcl 中的所有名称一样，是区分大小写的：numwords 和 NumWords 是两个不同的变量。

set 命令可以用来创建、读取和修改变量，该命令需要一个或两个参数。第一个参数是变量名，第二个参数，如果给出的话，是变量的新值。

```
    set a {Four score and seven years ago}
⇒   Four score and seven years ago
    set a
⇒   Four score and seven years ago
    set a 12.6
⇒   12.6
```

Tcl 变量在赋值时自动创建。这个示例中，第一条命令创建了一个新的变量 a(如果它以前不存在)，并把它的值设为字符序列 Four score and seven years ago。这个命令的结果是该变量的新值。第二条 set 命令只有一个参数：a。这种情况下该命令返回变量的当前值。第三条 set 命令把 a 的值改变为 12.6，并返回这个新值。

3.3 Tcl 的内部数据存储

Tcl 变量可以代表很多内容，如整数、实数、名称、列表以及 Tcl 脚本，这些类型的值各有高效率的内部表达形式。但是这些值也都可以表达为字符串。根据需要，Tcl 解释器会自动地把这些值在它们的内部表达形式和字符串表达形式之间转换。例如，如果用下面这条命令创建变量，Tcl 最初只存储字符串表达形式。

```
set a 12.6
```

在后面的命令中如有相关的 expr 命令，Tcl 会把字符串值解析为该值的内部浮点数表达形式。

```
set a [expr $a + 1.2]
```

当 set 命令把 expr 的返回值赋给 a 时，只有内部浮点数表达形式存在。然而，如果接下来用 puts 输出 a 的值，或者使用其他把 a 的值当作字符串对待的命令，Tcl 就会在保持该值的内部表达形式的同时，产生该值的字符串表达形式。

```
puts $a
```

最后，如果您修改 a 的字符串表示形式，例如，使用 append 命令在字符串的末尾添加一个字符，Tcl 就会更新该值的字符串表示形式，释放其内部浮点数表达形式(即使新的字符串可以被解析为一个浮点数)。

```
append a 32
```

内部表达形式可以高效率地处理变量的值，字符串表达形式允许使用同样的方法来处理不同的值，而且更容易进行通信。另外，因为在脚本层次上所有的值都表示为字符串，Tcl 解释器会自动为变量管理内存，因此无需进行变量声明。仅在需要时才进行字符串表达形式与内部形式之间的转换，Tcl 的性能因此有了显著的提升。

提示： 大多数 Tcl 程序受益于变量值的字符串形式与内部形式之间的自动转换。然而，如果程序经常改变一个变量的值，同时又在执行需要在变量值的字符串形式与内部形式之间进行转换的命令，那么就会出现反复转换。这个过程被称为“闪烁”，在操纵大的列表，特别是进行严格循环时最容易出现。要减少闪烁，就要使用不修改内部表达形式的命令来操纵变量(如使用列表命令来操纵列表，使用算术命令来操纵数字等)并且尽量避免强制使用字符串表达形式。

3.4 数组

除了简单变量，Tcl 还提供了数组。数组是元素的集合，每一个元素是有自己的名称和值的变量。数组元素的名称由两部分组成：数组名和数组中的元素名。数组名和元素名都可以是任意的字符串。因此 Tcl 的数组有时也被称为关联数组，以便同其他语言中元素名只能是数字的那种数组区分。

数组元素使用如 earnings(January)这样的表达式，即数组名(这里是 earning)加上圆括号中的元素名(这里是 January)。可以使用简单变量的地方都可以使用数组，例如 set 命令。

```
    set earnings(January) 87966
⇒ 87966
    set earnings(February) 95400
⇒ 95400
    set earnings(January)
⇒ 87966
```

如果 earningss 数组此前不存在，第一条命令创建该数组。如果元素 January 不存在，就在数组中创建元素 January 并赋值为 87966。第二条命令将一个值赋给数组中的 February 元素，第三条命令返回 January 元素的值。

提示： 不要把元素名用双引号括起来。如果这样做，双引号也会被识别为元素名的一部分，而不是元素名的定界符。

你也可以使用诸如 0、1、2 之类的元素名，但是这些名称仍然会被解释为字符串而不是整型值。因此，名称为 1 的元素和名称为 1.0 的元素是不同的。

在 Tcl 中，数组是无序的数据结构。(Tcl 内部使用一个哈希表存储数组元素。)当然，可以指定特别的元素名，使用本章后面将介绍的命令把它们排序。但是如果希望设计一个有序的数据结构，元素的值可以按顺序管理，那么 Tcl 列表是一个更好的选择。Tcl 列表的更多信息可参见第 6 章。

3.5 变量替换

第 2 章介绍了在 Tcl 命令中使用$符号将变量的值替换进来。本节更详细地讲述变量替换机制。

变量替换由 Tcl 单词中的$符号触发。在$后面的字符被视为一个变量名，$和变量名都会被变量的值替换。Tcl 提供了三种变量替换形式。目前为止您见过的只是最简单的形式，例如：

```
expr $a+2
```

这种形式就是$符号后面紧接着一个由字母、数字以及下划线组成的变量名。第一个不是字母、数字或下划线的字符表示变量名结束(这里就是+)。如果$符号后面紧接着的字符不是字母、数字或下划线，则$符号作为文本字符处理。

第二种变量替换形式是替换数组元素。和第一种形式相似，在数组变量名后面紧接着位于圆括号中的元素名。变量替换、命令替换和反斜线替换在元素名上作用的方式，与双引号中的命令单词相同。例如下面这段脚本：

```
set yearTotal 0
foreach month {Jan Feb Mar Apr May Jun Jul Aug Sep \
        Oct Nov Dec} {
    set yearTotal [expr $yearTotal+$earnings($month)]
}
```

在 expr 命令中，$earnings($month)由数组 earnings 中一个元素的值替换。元素名由 month 变量的值给出，这个值随着迭代的进行而改变。

元素名可能含有空白字符，而圆括号并非 Tcl 的引用字符，因此这些空白字符会视为单词分隔符处理。

```
    set capital(New Jersey) Trenton
Ø   wrong # args: should be "set varName ?newValue?"
    set capital("New Jersey") Trenton
Ø   wrong # args: should be "set varName ?newValue?"
```

第二个示例发生错误的原因是双引号没有作为单词的第一个字符出现，因此 Tcl 将其视为文本字符而非单词定界符。结果造成 capital("New 是一个单词，Jersey")是另一个单词，set 命令接受了多于两个参数。

要在元素名中使用空格，可以使用反斜线替换的空格符，或者将整个变量都用双引号括起来。

```
    set capital (New\ Jersey) Trenton
⇒   Trenton
    set "capital(South Dakota)" Pierre
⇒   Pierre
```

还有一个选择是在元素名称上使用替换。因为替换不影响命令中的单词定界，即使替换进来的内容包括空白字符也不影响，所以这样做是可行的。例如：

```
    set state "New Mexico"
⇒   New Mexico
    set capital($state) "Santa Fe"
⇒   Santa Fe
```

最后一种替换形式适用于简单变量，即在变量后面跟一个数字或字母或下划线的情形。例如，假设希望把值 1.5m 作为一个参数传给一个命令，但这个数值是变量 size(在 Tk 中这样做实际上是将单位设置为毫米)。如果您写成$sizem 这样的形式进行替换，Tcl 会把 m 视为变量名的一部分。要解决这个问题，可以将变量名括在大括号中，示例如下：

```
.canvas configure -width ${size}m
```

还可以使用大括号来指定包括除字母、数字、下划线以外的字符的变量名。

提示：大括号只可以用于简单变量的定界；不能用来指定数组中的某个元素。

设置 Tcl 的变量替换机制的目的只是为了处理最常见的情况，有些情况下这些替换机制都不能达到要求。更复杂的情况可以通过命令序列来处理。例如，format 命令可以用来产生几乎您能想象到的任何形式的变量名，set 命令可以用来读写具有复杂变量名的变量，命令替换可以用于将变量的值替换进其他命令中。

提示：一般来说，在变量名中只使用字母、数字以及下划线，这样可以让代码可读性更强，也更易编写。

3.6 多维数组

Tcl 只实现了一维数组，但是通过将多个索引串接到一个元素名中，也可以模拟多维数组。下面这个程序就模拟了有整数索引的二维数组。

```
set matrix(1,1) 140
set matrix(1,2) 218
set matrix(1,3) 84
set i 1
set j 2
set cell $matrix($i,$j)
```

```
⇒ 218
```

matrix 是有三个元素的数组，元素名分别为 1,1 和 1,2 和 1,3。然而，这个数组的行为就如同它是二维数组一样，具体来说，当 set 命令扫描元素名时会发生变量替换，i 和 j 的值被整合进一个适当的元素名中。

提示： 这个示例中空格是很有影响的：matrix(1,1)和 matrix(1, 1)指向的是两个不同的变量。在实际应用中最好不要使用空格，它们在命令解析中可能引发莫名其妙的错误。例如，命令：

```
set matrix(1, 1) 140
```

就是有 4 个单词的命令，第三个单词是 1)。结果是向 set 传入了三个参数，造成错误。要避免这个问题，需要把整个变量名括在双引号中。

3.7 查询数组的元素

array 命令能够提供被定义为数组元素的变量的相关信息以及整个数组的选项设定。该命令以不同的方式提供这些信息，具体取决于传入的第一个参数。命令 array size 返回一个数字，反映指定的数组中定义了多少个元素；命令 array names 则返回指定数组的元素名列表。

```
set currency(France) euro
set "currency(Great Britain)" pound
set currency(Japan) yen
array size currency
⇒ 3
array names currency
⇒ {Great Britain} Japan France
```

前面这些命令，第二个参数都必须是一个数组变量。作为可选项，还可以指定额外的 pattern 参数，指定该参数后，返回值就只包含按照 string match 规则符合指定形式的那些元素的名称(5.10 节中对 string match 命令有详细介绍)。由 array names 返回的列表中的元素没有特定的顺序。

array names 命令可以与 foreach 联合使用，遍历数组中的各个元素。例如，下面这段代码删除元素中值为 0 或空的元素项。

```
foreach i [array names a] {
    if {($a($i) == "") || ($a($i) == 0)} {
        unset a($i)
    }
}
```

提示： array 命令还提供了另一种搜索数组元素的方法，就是使用 startsearch、anymore、nextelement 以及 donesearch 选项。这种方法比上面给出的 foreach 方法功能更强，在某些情况下也更有效率，但是也更加繁琐，而且并不常用。该方法细节请阅读参考文献。

array exists 命令也可以用于检测数组中的某个特定变量是否存在，array get 和 array set

命令可用于数组和字典之间的相互转化。

```
    set a(head) hat
    set a(hand) glove
    set a(foot) shoe
    set apparel [array get a]
⇒   foot shoe head hat hand glove
    array exists a
⇒   1
    array exists apparel
⇒   0
    array set b $apparel
    lsort [array names b]
⇒   foot hand head
```

3.8 incr 命令和 append 命令

incr 命令和 append 命令提供了改变变量值的简单方法。incr 读入两个参数，分别是变量名和一个整数。incr 命令将这个整数加到变量值上，将结果存到变量中，返回变量的新值作为结果。

```
    set x 43
    incr x 12
⇒   55
```

这个整数可以是正数，也可以是负数。也可以不加指定，使用默认值 1。

```
    set x 43
    incr x
⇒   44
```

变量的初始值和这个增量值都必须是可解析为整数的字符串，无论是十进制、八进制(以 0 开头)还是十六进制(以 0x 开头)。

在 Tcl 8.5 以前的版本中，对不存在的变量进行增量操作将发生错误。在 Tcl 8.5 及以后的版本中，incr 在这种情况下的行为被修改为创建该变量，并将其值设为增量值。

```
    incr y
⇒   1
    incr z 42
⇒   42
```

append 命令将文本添加到一个变量的结尾。它需要获得两个或更多参数，第一个参数是变量的名称，其余的参数是要添加到变量中的文本字符串。它把这些字符串添加到变量值的结尾，返回变量的新值。下面这个示例使用 append 得到一个平方表。

```
    set msg ""
    foreach i {1 2 3 4 5} {
        append msg "$i squared is [expr $i*$i]\n"
    }
    set msg
⇒   1 squared is 1
    2 squared is 4
    3 squared is 9
    4 squared is 16
    5 squared is 25
```

提示：append 命令在向变量的值中添加字符时不会自动插入空格。如果想要空格，需要在作为参数的文本字符串中明显地包含空格。

incr 和 append 都没有为 Tcl 增加新的功能，这两个命令的功能可以通过使用其他命令达成。然而，它们提供了进行这些操作的简单途径。另外，append 可以用于避免进行字符复制。如果需要聚集片段构建一个很长的字符串，使用以下命令：

```
append x $piece
```

要比如下命令有效率得多。

```
set x "$x$piece"
```

3.9 移除变量：unset 和 array unset

unset 命令用于销毁变量。它可以获取任意多个参数，每一个参数都是变量名，它的功能是删除这些变量。以后再对这些变量进行读取操作就会导致错误，就如同这些变量没有被设置过。unset 的参数必须是简单变量、数组元素或数组，示例如下：

```
unset a earnings(January) b
```

这里变量 a 和 b 会被完全移除，数组 earnings 的 January 元素会被移除。在这个 unset 命令执行后，数组 earnings 还是继续存在。如果 a 和 b 是数组，那么数组及数组中所有的元素都会被移除。

您也可以使用 array unset 命令删除数组中的元素，这个命令接受一个数组名和一个样式作为参数。数组中的元素名称按 string match 命令规则符合指定样式的(5.10 节有详细介绍)，就会被删除。

3.10 预定义变量

Tcl 库自动创建和管理了一些全局变量。tclvars 的说明文档讲述了所有这些变量，本节只介绍最为常用的那部分。

当调用 tclsh 或 wish 脚本文件时，脚本文件的文件名就存放在变量 argv0 中，命令行参数以列表形式存放在变量 argv 中，命令行参数的个数存放在变量 argc 中。考虑下面这段 tclsh 脚本：

```
#!/usr/bin/env tclsh
puts "The command name is \"$argv0\""
puts "There were $argc arguments: $argv"
```

如果您把这个脚本放在名为 printargs 的文件中，把这个文件设为可执行文件，然后从命令外壳中调用它，则会输出一些有关它的参数的信息。

```
    printargs red green blue
⇒ The command name is "printargs"
⇒ There were 3 arguments: red green blue
```

变量 env 是由 Tcl 预定义的。它是一个数组变量，其元素是所有过程的环境变量。例如，下面这条命令输出用户的主文件夹，由 HOME 环境变量设定。

```
puts "Your home direcotry is $env(HOME)"
```

变量 tcl_platform 是一个数组变量，其元素是对应用程序正在运行的平台的描述，例如操作系统的名称和当前版本号，以及机器的指令集。

```
   puts $tcl_platform(platform)
⇒ unix
   puts $tcl_platform(os)
⇒ Darwin
   puts $tcl_platform(machine)
⇒ i386
```

当您在编写必须不加改动就同时可以在 Windows 和 Unix 中运行的脚本时，该数组特别有用。根据这个数组的值，可以执行相应平台所需要的平台特有代码。

提示： Mac Os X 以前的版本，tcl_platform(platform)的值是 macintosh。从 Mac Os X 开始，这个值是 unix，而且 tcl_platform(os)的值是 Darwin，如上例所示。使用 Tk 编写的脚本可能会依赖于系统(例如 X11 与 Mac OS Aqua)，这种情况下可以使用第 29 章介绍的 tk windowingsystem 命令查询正在运行脚本的计算机系统。

3.11 其他变量功能预览

Tcl 还提供了很多其他用于变量操纵的命令。随着您对 Tcl 语言学习的深入，本书会全面介绍这些命令。本节是对部分功能的预览。

trace 集合命令可用于监视变量，在变量设置、被读取或被删除时调用指定的 Tcl 脚本。变量跟踪在调试时会很有用，该命令还允许您创建只读变量。您还可以把变量跟踪用于“传播”，例如，只要变量值发生变化，数据库或屏幕上显示的值就即时更新。变量跟踪在 15.6 节中有详细讨论。

global 和 upvar 命令可由过程使用，访问不由它所有的局部变量。这个命令在第 9 章中讨论。

namespace 集合命令创建并管理命名空间，命名空间是命令和变量的命名集合。命名空间可以将命令和变量分隔开来，确保它们不会干扰其他命名空间中的命令和变量。命名空间在第 10 章中讨论。

第4章 表 达 式

表达式将值(或操作数)与操作符联合起来，产生新的值。例如，表达式 4+2 就包含两个操作数(4 和 2)和一个操作符(+)，处理它得到的值是 6。很多 Tcl 命令的一个或多个参数是表达式。这样的命令中最简单的就是 expr，这个命令就是把它的参数作为表达式处理，获得表达式的值并把这个值作为字符串返回。

```
    expr {8+4} * 6.2
⇒ 74.4
```

另一个例子是 if，这个命令会将第一个参数作为表达式处理，用其结果决定是否把第二个参数作为 Tcl 脚本进行处理。

```
if {$x < 2} {set x 2}
```

本章的示例都使用 expr 命令，但同样的语法、替换以及处理规则适用于所有的表达式应用。

4.1 本章出现的命令

本章讨论 expr 命令。

- `expr` *`arg ?arg arg ...?`*
  把所有的 *arg* 值连接起来(之间用空格隔开)，再把连接起来的值作为表达式处理，返回表达式的值。

4.2 数值操作数

表达式的操作数通常是整数或实数。整数可能是十进制的(普通格式)、二进制的(开头两个字符是 0b)、八进制的(开头两个字符是 0o)或十六进制的(开头两个字符是 0x)。例如，如下操作数表示的是同一个整型值：335(十进制)，0o517(八进制)，0x14f(十六进制)，0b101001111(二进制)。从 Tcl 8.5 开始，使用了新的二进制和八进制前缀。

提示： 为了与更早发布的 Tcl 兼容，八进制整型值也可以只在操作数的开头用 0 表示，无论第二个字符是不是 o。因此，0517 等于十进制的 335。而 092 就不是一个有效的整型数，因为开头的 0 表示这是一个八进制数，9 不是一个八进制数字。处理零开

头的十进制整型数最安全的方法是使用 scan 命令，详见 5.9 节的讨论。Tcl 用户的维基网页 http://wiki.tcl.tk/498 展示了处理这个问题的一些方法，例如下面这个过程：

```
proc forceInteger { x } {
  set count [scan $x {%d %c} n c]
  if { $count !=1} {
      return -code error "not an integer: \"$x\""
  }
  return $n
}
```

您可以使用 ANSI C 定义的大多数格式来表达实数，示例如下：

```
2.1
7.91e+16
6E4
3.
```

提示： Tcl 中任何需要整数或实数值的地方都可以接受这些格式，而不仅仅是在表达式中。

表达式操作数也可以是非数字的字符串。4.6 节将讨论字符串操作数。

4.3 操作符及其优先级

表 4.1 列出了 Tcl 表达式支持的所有操作符，它们与 ANSI C 表达式中的操作符相似。表中横线将操作符按优先级分组，有更高优先级的操作符在表中的位置更靠上。例如，4*2<7 的处理结果为 0，因为*的优先级高于<的优先级。除了最简单明确的情况之外，最好使用圆括号来明确指定操作符的分组方式与处理顺序，这样可以帮助您或其他需要修改该程序的人避免一些错误。

相同优先级的操作符从左向右进行计算。例如，10-4-3 与(10-4)-3 是相同的，其结果都是 3。

表 4.1 Tcl 表达式中的操作符总表

语 法	结 果	操作数类型
-a	*a* 的负值	int，real
+a	对 *a* 作一元加操作	int，real
!a	逻辑非：*a* 为 0 则结果为 1，否则结果为 0	int，real
~	*a* 按位取反	int
*a**b*	指数：*a* 的 *b* 次方	int，real
*a*b*	*a* 乘以 *b*	int，real
a/b	*a* 除以 *b*	int，real

续表

语　法	结　果	操作数类型
a%b	*a* 除以 *b* 取余	int
a+b	*a* 加 *b*	int，real
a-b	*a* 减 *b*	int，real
a<<b	*a* 左移 *b* 位	int
a>>b	*a* 算术右移 *b* 位	int
a<b	如果 *a* 小于 *b*，结果为 1，否则为 0	int，real，string
a>b	如果 *a* 大于 *b*，结果为 1，否则为 0	int，real，string
a<=b	如果 *a* 小于或等于 *b*，结果为 1，否则为 0	int，real，string
a>=b	如果 *a* 大于或等于 *b*，结果为 1，否则为 0	int，real，string
a==b	如果 *a* 等于 *b*，结果为 1，否则为 0	int，real，string
a!=b	如果 *a* 不等于 *b*，结果为 1，否则为 0	int，real，string
a eq *b*	如果 *a* 等于 *b*，结果为 1，否则为 0	string
a ne *b*	如果 *a* 不等于 *b*，结果为 1，否则为 0	string
a in *b*	列表包含：如果 *a* 是列表 *b* 中的元素，则为 1，否则为 0	*a*:string;*b*:list
a ni *b*	非列表包含：如果 *a* 不是列表 *b* 中的元素，则为 1，否则为 0	*a*:string;*b*:list
a&b	*a* 和 *b* 按位与	int
a^b	*a* 和 *b* 按位异或	int
a|b	*a* 和 *b* 按位或	int
a&&b	逻辑与：如果 *a* 和 *b* 都非零则为 1，否则为 0	int，real
a||b	逻辑或：如果 *a*、*b* 中至少有一个非零则为 1，否则为 0	int，real
a?b:c	选择：如果 *a* 非零则 *b*，否则 *c*	*a*:int，real

4.3.1　算术操作符

Tcl 表达式支持算术操作符+、-、*、/、%以及**。操作符-可以是二元操作符减号，如 4-2，也可以是一元操作符负号，如-(6*$i)。如果两个操作数都是整型数，操作符/就会把结果取整。%是模运算符，其结果是它的第一个操作数除以第二个操作数所得的余数。%操作符的两个操作数都必须是整型数。**是取幂操作符，Tcl 8.5 开始引入，计算第一个操作数的第二个操作数次方。

提示： 操作符/和%在 Tcl 中的行为比 ANSI C 中更有一致性。在 Tcl 中，余数总是大于或等于零，且它的绝对值小于除数的绝对值。ANSI C 只保证了后一点。在 ANSI C 和 Tcl 中，对于任意的 x 和 y，商总是有这样的性质：(x/y)*y+x%y 的值是 x。

4.3.2　关系操作符

操作符<(小于)、<=(小于或等于)、>=(大于或等于)、>(大于)、==(等于)和!=(不等于)用于比较两个值。每个操作符都在操作数满足指定关系时产生 1(真)，否则产生 0(假)。这

些操作符可以进行数字比较，也可进行字符串比较(关于字符串比较的更多信息参见4.6节)。

4.3.3 逻辑操作符

逻辑操作符&&、||以及!，通常用于结合关系操作符的结果，例如：

```
($x > 4) && ($x < 10)
```

其中每一个操作符的结果都是0或1。如果逻辑与(&&)的两个操作数都是非零的数，它的结果就是1，如果逻辑或(||)的两个操作数中至少有一个不是零，它的结果就是1，如果逻辑非(!)的操作数为零，它的结果就是1。

在Tcl中，与在ANSI C中一样，零值代表假，其他所有非零值代表真。Tcl还可以用字符串来表示布尔变量：false、no以及off表示假，true、yes以及on表示真。注意这些字符串不区分大小写，并且可以采用不会产生歧义的缩写形式。不过在Tcl产生一个真/假值时，它总是用1代表真，用0代表假。

与C语言一样，&&和||也是按顺序处理的，即如果&&的第一个操作数为0，或||的第一个操作数为1，就不再处理第二个操作数。

4.3.4 按位操作符

对整型数的各位进行操作，Tcl提供了6个操作符：&、|、^、<<、>>以及~。这些操作符要求其操作数必须是整型数。&、|、^操作符分别执行按位与、或、异或：每一位的结果是把操作数的相应位的值从左向右按指定操作处理一遍得到的。注意&和|得到的结果并不总是与&&和||的结果相同。

```
    expr 8&&2
⇒   1
    expr 8&2
⇒   0
```

操作符<<和>>把左边的操作数按位移动，移动次数等于右边的操作数。左移时低位补0，右移则是算术右移，即正数补0，负数补1。这里的右移行为与ANSI C中不同，在ANSI C中右移行为是依赖于机器的。

操作符~(“取反符号”)只接收一个操作数，将操作数中的各位取反作为结果：用0取代1，用1取代0。

4.3.5 选择操作符

三元操作符?:可以在两个结果中选择一个。

```
expr {($a < $b) ? $a : $b}
```

这个语句返回$a和$b中的较小值。选择操作符首先检查第一个操作数的值是真还是假。如果为真(非零)，则处理?后面紧接着的参数；如果第一个操作数为假(零)，则处理第三个操作数。第二个和第三个参数中，只会处理其中的一个。

4.4 数学函数

Tcl 表达式支持数学函数，如 sin 和 exp。数学函数可以由标准函数标记进行调用。

```
expr 2*sin($x)
expr hypot($x , $y) + $z
```

数学函数的参数可以是任意表达式，多个参数之间用逗号分开。表 4.2 列出了所有的内建函数。

表 4.2 Tcl 表达式支持的预定义数学函数

函 数	结 果
abs(*x*)	*x* 的绝对值
acos(*x*)	*x* 的反余弦函数值，值域从 0 到 π
asin(*x*)	*x* 的反正弦函数值，值域从-π/2 到 π/2
atan(*x*)	*x* 的反正切函数值，值域从-π/2 到 π/2
atan2(*x*,*y*)	*x*/*y* 的反正切函数值，值域从-π/2 到 π/2
bool(*x*)	将可转化的表达式转化为布尔型值，1 表示真，0 表示假
ceil(*x*)	不小于 *x* 的最小整数
cos(*x*)	*x* 的余弦值(*x* 用弧度表示)
cosh(*x*)	*x* 的双曲余弦值
double(*i*)	数值等于整数 *i* 的实数
exp(*x*)	e 的 *x* 次方
floor(*x*)	不大于 *x* 的最大整数
fmod(*x*,*y*)	*x* 除以 *y* 的实余数
hypot(*x*,*y*)	(x^2+y^2)的平方根
int(*x*)	把 *x* 截断到个位所得的整型值
log(*x*)	*x* 的自然对数
log10(*x*)	*x* 的以 10 为底的对数
max(*arg*,...)	所有给出的数值参数的最大值
min(*arg*,...)	所有给出的数值参数的最小值
pow(*x*,*y*)	*x* 的 *y* 次方
rand()	产生在[0,1]区间的伪随机浮点数
round(*x*)	对 *x* 进行四舍五入得到的整型值
sin(*x*)	*x* 的正弦值(*x* 用弧度表示)
sinh(*x*)	*x* 的双曲正弦值
sqrt(*x*)	*x* 的平方根
srand(*x*)	使用整数种子 *x* 重设随机数生成器

续表

函 数	结 果
tan(*x*)	*x* 的正切值(*x* 用弧度表示)
tanh(*x*)	*x* 的双曲正切值
wide(*x*)	表示 *x* 的宽度的整型值(至少是 64 位)

在 Tcl 8.5 中，当表达式解析器遇到像 sin($x)这样的数学函数时，它会把函数置换为对 tcl::mathfunc 命名空间中的一个普通 Tcl 命令的调用。(第 10 章将讨论 Tcl 的命名空间。)如果数学函数的参数中包含逗号，则由 expr 处理参数，将各个分开的参数传给函数的实现过程。因此，表达式

```
expr {sin($x+$y)}
```

和下面这个表达式的处理过程是完全一样的。

```
expr {[tcl::mathfunc::sin [expr {$x+$y}]]}
```

而且，表达式

```
expr {atan2($y-0.3, $x/2)}
```

也等同于下列表达式。

```
expr {[tcl::mathfunc::atan2 [expr {$y-0.3}] [expr {$x/2}]]}
```

提示： 将数学函数映射为命令的机制，使用了关联命名空间引用。如果一个命名空间定义了命令[namespace current]::tcl::mathfunc::sin，在该命名空间的表达式处理中调用 sin 就会优先指向这个命令而不是 Tcl 预定义的::tcl::mathfunc::sin。

该功能允许您定义自己的数学函数，只需要在 tcl::mathfunc 命名空间中创建新的命令即可。例如，定义一个函数来计算一到多个数字的平均值。

```
proc tcl::mathfunc::avg {args} {
    if {[llength $args] == 0} {
        return -code error \
            "too few arguments to math function \"avg\""
    }
    set total 0
    foreach val $args{
        set total [expr {$total + $val}]
    }
    return [expr {double($total) / [llength $args]}]
}
expr avg(2, 5, 1, 7)
⇒ 3.75
```

4.5 替换

表达式操作数的替换有两种方式。第一种是普通的 Tcl 解析器机制，例如下面的命令：

```
expr 2*sin($x)
```

这里 Tcl 解析器会在执行命令前替换变量 x 的值，传递给 expr 的第一个参数的值类似于 2*sin(0.8)。第二种方式则是通过表达式处理，在处理表达式时会再进行一轮变量替换和命令替换。例如：

```
expr {2*sin($x)}
```

这里大括号阻止了 Tcl 解析器替换 x 的值，因此传给 expr 的参数是 2*sin($x)。当表达式处理器遇到$符号时，它自己会进行一次变量替换，把变量 x 的值作为参数传给 sin。

提示： 当表达式处理器执行变量或命令替换时，替换的值必须是整数或实数(或者后文将介绍的字符串)。这里替换的值不能是任意的表达式。

有这样两层替换对 expr 命令的操作来说通常不会造成什么不同，但是对于 while 这样会反复处理一个表达式，而各次处理所得的值不同的命令就有很大的影响了。例如下面这个脚本，该脚本寻找比指定的数大的、最小的 2 的幂。

```
set pow 1
while {$pow<$num} {
    set pow [expr $pow*2]
}
```

表达式$pow<$num 在每次迭代开始之前都要处理一次，判断是否终止循环。该表达式在每次处理的时候都必须替换进 pow 的新值。如果没有这个大括号，那这个替换就会在解析 while 命令时进行——例如，while $pow<$num ...——那样 while 的第一个参数应该是常量表达式，如 1<44；这个循环要么不进行，要么就会进入死循环。

提示： 最好总是把表达式用大括号括起来，即使是在使用 expr 命令的时候。Tcl 处理括起来的表达式的效率，大大高于处理没有括起来的表达式的效率。把表达式用大括号括起来还能避免在代码中出现一些难以捉摸的安全漏洞。考虑以下情况，程序提示用户给出一个值，用户输入了一个表达式，例如：

```
set x [expr $input +2]
```

如果一个恶意用户在 Windows 系统中输入[format C:\]，那么 Tcl 解释器会把这个字符串作为 input 变量，而表达式处理器会执行其中的命令——格式化 C 盘。如果把这个表达式用括号括起来，表达式处理器会把它的值作为 input 的值替换进来，这个值并不是一个有效的数学表达式，于是会产生一个错误。

4.6 字符串操作

和 ANSI C 不同，Tcl 的部分操作符接受字符串操作数，例如下面这条命令：

```
if {$x eq "New York"} {
    ...
}
```

这个示例中，表达式处理器比较变量 x 的值和字符串 New York，使用字符串相等操作符进行这个比较，如果它们相同，就执行 if 块。字符串以字典的形式进行比较。具体的整理顺序因系统的不同而不同。

要指定一个字符串操作数，必须把它放在大括号中或双引号中，或使用变量替换或命令替换。上面这个示例把整个表达式括起来是很重要的，这样表达式处理器就只替换进来x的值。考虑如下命令：

```
set result [expr $x eq "New York"]
```

没有那个大括号，传给expr的变量就会串接起来，结果该变量的表达式变为：

```
Los Angeles eq New York
```

表达式解析器无法解析Los(它不是数字，不是函数名，也不能解析为字符串，因为它不在引号当中)，于是就会发生语法错误。

如果字符串被双引号括起来，就由表达式处理器进行引号之内的命令替换、变量替换以及反斜线替换。如果字符串被大括号括起来，就不进行这些替换。在字符串中嵌套使用大括号的效果，与在命令单词中嵌套使用的效果相同。

除了检查字符串相等还是不相等的eq和ne操作符，<、>、<=、>=、==、!=操作符都可以进行字符串比较。不过，这些操作符仅在至少有一个操作数不能解析为数值时才进行字符串比较。例如下面这段脚本：

```
set x 8
set y 010
expr {$x == $y}
⇒ 1
```

expr进行的将是数学比较，因而其处理结果是1。

提示： 如果想要把两个看似数字的值(例如，存储在变量中的值，您并不知道那些值具体是怎么样的)作为字符串进行比较，必须使用字符串比较操作符，或是string compare这样的命令，该命令将在第5章中介绍。

4.7 列表操作

Tcl表达式还支持两个列表操作符。(第6章将详细讨论列表操作。)如果指定字符串是列表的元素，in操作符返回1，否则返回0；如果字符串不是列表的元素，ni操作符返回1，否则返回0。例如下面这段脚本，检测Los Angeles是否是cities列表的一个元素。

```
if {"Los Angeles" in $cities} {
    ...
}
```

in和ni操作符在字符串和列表元素间进行严格比较，不包含子字符串和样式匹配。因此，前面这个表达式与lsearch -exact测试是完全相同的。

```
if {[lsearch -exact $cities "Los Angeles"] != -1} {
    ...
}
```

4.8 类型与转换

只要可能，Tcl 就把表达式作为数字处理。只有在使用字符串比较操作符时，或在使用关系操作符且至少一个操作数无法理解为数字时才进行字符串操作。大多数操作符接受整数和实数，不过有一小部分，如<<和&，只接受整型数。

Tcl 支持任意大的整数，但是整数实际上通常存储为 C 的 long 类型，在多数机器上这个类型占 32 位。对于 long 类型不能存放的整数，Tcl 自动使用任意精度的内部整型数据类型。实数以 C 的 double 类型表达，通常是 64 位(大约 15 位十进制数)，遵循二进制浮点数的 IEEE 标准。

如果一个操作符的各个操作数类型不同，Tcl 会自动进行类型转换。如果一个操作数是整数，另一个操作数是实数，那么整数会被转化为实数。如果一个操作数是非数字的字符串，另一个操作数是整数或实数，那么这个整数或实数会被转化为字符串。

比较操作符返回的结果总是布尔型的 0 或 1。算术操作符如果有一个或多个操作数是 double，返回值类型就是 double。整型算术操作的结果会尽可能存储为原生长整型数据，如果需要的话也可以存储为任意精度的整型数据。您可以使用函数 double 明确指示把整型数转换为实数，使用 int、wide、entier 以及 round 把实数转化为整数，转化方式有截断和四舍五入。int 函数总是返回短的整型数(存放不了的高位部分会被舍去)。wide 函数总是返回长整型数(如果原参数是 32 位，进行符号的扩展；如果原参数超过 64 位，舍去高位部分)。entier 函数把参数转化为适当长度的整型数。entier 与 int 和 wide 不同，int 返回的整型值长度为机器字长，wide 返回的整型值长度为 64 位，而 entier 的返回值可以占用任意长度的存储空间，保证能存放完整的值。

4.9 精度

处理完表达式后，数据是以内部形式进行保存的，只在必要的时候才会转化为字符串。整数可以转化为带符号的十进制字符串而不损失任何精度。实数在保证转化之后仍能与机器可接受的与它相邻的浮点数相区分的条件下，转换为尽可能短的字符串。

第5章　字符串操作

本章讲述 Tcl 的字符串操作功能。Tcl 内部将字符串作为 Unicode 字符存储，但是 Tcl 也支持很多其他的字符集，并可以自动地或依照命令在字符集之间进行翻译。Tcl 还支持二进制字符串。Tcl 的字符串操作命令包含两种不同的模式匹配方法：一种与外壳中的文件名展开规则相似，另一种将正则表达式作为样式。Tcl 也有用于格式化输入输出的命令，与 C 程序中的 scanf 和 printf 相似。最后，还有一些工具性的命令，用于计算字符串的长度，从字符串中取出字符，完成大小写转换以及其他一些任务。

5.1　本章出现的命令

本章讨论的字符串操作命令如下。

- `binary format` *`format ?value value ...?`*
  返回二进制字符串，输出方式由 *format* 指定，其余的参数是输出的内容。
- `binary scan` *`string format varName ?varName varName ...?`*
  解析二进制的 *string* 的字段，返回值是执行的转换的次数。*string* 是待解析的输入，*format* 指明了如何解析。每一个 *varName* 都给出了一个变量名，扫描 *string* 的字段时，其结果会被赋给对应的变量。
- `encoding convertfrom` *`?encoding? string`*
  将 *string* 从指定的 *encoding* 转化为 UTF-8 Unicode。如果不指定 *encoding*，默认为系统编码。
- `encoding convertto` *`?encoding? string`*
  将 *string* 从 UTF-8 Unicode 转化为 *encoding*。如果不指定 *encoding*，默认为系统编码。
- `encoding names`
  返回识别到的编码名称。
- `encoding system` *`?encoding?`*
  将系统编码设置为 *encoding*。如果不指定 *encoding*，这个命令会返回当前的系统编码。
- `format` *`format ?value value ...?`*
  返回等同于 *format* 的结果，不过 *format* 中的%序列都由 value 参数按顺序替换。
- `regexp` *`?options? exp string ?matchVar? ?subVar subVar ...?`*

检查正则表达式 *exp* 是否与 *string* 全部或部分匹配，如果匹配，返回 1，否则返回 0。如果发现匹配，匹配范围的信息存入 *matchVar* 和 *subVar* 变量中(如果指定了这些变量)。*options* 的完整列表请查阅手册。

- `regsub ?options? exp string subSpec ?varName?`
  像 regexp 一样，在 *string* 中检查 *exp* 的匹配，如果有一个匹配，就返回 1，没有匹配就返回 0。把 *string* 复制到名为 *varName* 的变量中，同时把 *string* 中匹配的部分替换为 *subSpec*。如果不给定 *varName*，返回值就是替换的结果。*options* 的完整列表请查阅手册。
- `scan string format varName ?varName varName ...?`
  解析 *string* 中由 *format* 指定的字段，把与%序列匹配的值存入名为 *varName* 的变量中，返回值是成功解析的字段个数。另外，如果没有提供 *varName* 变量，返回值就是与 *format* 匹配的字段的列表。
- `source ?-encoding encoding? Filename`
  读取并运行 *fileName* 文件中保存的脚本。Tcl 使用系统编码读取文件，如果使用 -encoding 指定了 *encoding*，则用指定编码读取文件。
- `string bytelength string`
  返回值为 *string* 内部存储时所占的字节数。大多数情况下，可以用 string length 代替这个命令。
- `string compare ?-nocase? ?-length num? string1 string2`
  返回值为-1、0、1，分别对应 *string1* 在字典上小于、等于、大于 *string2*。使用-nocase 选项可以忽略大小写。如果使用-length 选项，就只使用从头开始的前 *num* 个字符进行比较。
- `string equal ?-nocase? ?-length num? string1 string2`
  如果 *string1* 和 *string2* 相同，返回 1，否则返回 0。使用-nocase 选项，可以忽略大小写。如果使用-length 选项，就只使用从头开始的前 *num* 个字符进行比较。
- `string first string1 string2 ?startIndex?`
  返回 *string2* 中第一次出现与 *string1* 完全相同的子字符串的开头字符的索引，如果没有匹配，则返回-1。如果指定了 startIndex，则从 *string2* 的索引为 *startIndex* 的字符开始搜索。
- `string index string charIndex`
  返回 *string* 中索引为 *charIndex* 的字符，如果没有那个字符，则返回空字符串。注意 *string* 中的第一个字符的索引是 0。
- `string is class ?-strict? ?-failindex varname? string`
  如果 *string* 的字符是指定字符 *class* 的有效成员，返回 1；否则返回 0。如果不指定-strict，空的 string 也会返回 1，指定时则返回 0。如果指定-failindex，*varname* 会被赋值为 *string* 中第一个不符合指定 *class* 的字符的索引值。如果命令返回 1，这个 *varname* 不会被设置。相关的字符类型列表请参阅手册。
- `string last string1 string2 ?lastIndex?`

返回 *string2* 中与 *string1* 精确匹配的最右边的子字符串的开头字符的索引，如果没有匹配，则返回-1。如果指定 *lastIndex*，则 *string2* 中只对索引小于或等于 *lastIndex* 的部分进行搜索。

- `string length string`
  返回 *string* 中的字符个数。
- `string map ?-nocase? mapping string`
  基于 *mapping* 字典对 *string* 进行子字符串替换，返回新的字符串。凡在 *string* 中出现的 *mapping* 关键字都替换为字典中对应的值。选项-nocase 指定进行不区分大小写的匹配。
- `string match ?-nocase? pattern string`
  如果根据通配符样式匹配规则(*、？、[]以及\)*string* 与 *pattern* 匹配则返回 1，否则返回 0。选项-nocase 指定进行不区分大小写的匹配。
- `string range string first last`
  返回 *string* 当中从 *first* 到 *last* 之间所有(包含 *first* 和 *last*)的子字符串。
- `string repeat string count`
  返回值为把 *string* 重复 *count* 遍所得到的字符串。
- `string replace string first last ?newstring?`
  把 *string* 中从 *first* 到 *last* 之间所有(包含 *first* 和 *last*)的字符移除，如果设定了 *newstring*，置换为 *newstring*，返回新得到的字符串。
- `string tolower string ?first? ?last?`
  返回值即 *string* 字符串，只是把所有大写字母替换成小写字母。如果设定 *first* 和/或 *last*，则替换 *first* 和 *last* 之间的部分，否则对整个字符串进行替换。
- `string totitle string ?first? ?last?`
  将 *string* 字符串的第一个字符转换为标题格式，将其余字符转换为小写，返回所得的字符串。如果设定了索引 *first* 和/或 *last*，则转换 *first* 和 *last* 之间的部分，否则对整个字符串进行转换。
- `string toupper string ?first? ?last?`
  返回值即 *string* 字符串，只是把所有小写字母替换成大写字母。如果设定 *first* 和/或 *last*，则替换 *first* 和 *last* 之间的部分，否则对整个字符串进行替换。
- `string trim string ?chars?`
  把 *string* 字符串中开头和结尾处的所有 *chars* 字符删除，返回新的字符串。*chars* 默认设置为空白符。
- `string trimleft string ?chars?`
  与 string trim 相同，只是仅删除开头字符。
- `string trimright string ?chars?`
  与 string trim 相同，只是仅删除结尾字符。
- `string wordend string charIndex`
  返回 *string* 当中最后一个包含字符 *charIndex* 的词的结尾字符的索引加一。一个词是指一段连续的字母、数字以及下划线，其他的单个符号分别视为一个词。

- `string wordstart` *`string charIndex`*
  与 string wordend 相同，只不过返回的是 *string* 当中第一个包含字符 *charIndex* 的词的开头字符的索引。

5.2 取得字符：string index 和 string range

很多字符串操作命令是作为 string 命令的选项实现的。例如，string index 从字符串中取得一个字符。

```
    string index "Sample string" 3
⇒ p
```

index 后面的参数是一个字符串，最后一个参数给出了要取得的字符的索引。对于所有的字符串命令，索引 0 对应着字符串的开头字符，1 对应着第二个字符，以此类推。索引 end 对应着字符串的结尾字符，end-1 对应着倒数第二个字符，以此类推。从 Tcl 8.5 开始，可以把两个整型值加减的表达式设为索引值。在使用 end±整数或整数±整数的形式时，不能在索引参数中任意使用空白，即使把参数括起来也不行。如果索引值指向的地方超出了字符串，string index 会返回空字符串。

```
    set i 2
    string index "Sample string" end-$i
⇒ i
    string index "Sample string" 5+$i
⇒ s
    incr i
    string index "Sample string" 5+$i
⇒ t
```

命令 string range 与 string index 相似，只不过它需要两个索引，返回从第一个索引指向的位置到第二个索引之间的所有字符，包括这两个索引本身指向的字符。

```
    string range "Sample string" 3 7
⇒ ple s
    string range "Sample string" 3 end
⇒ ple string
```

5.3 长度、大小写转换、裁剪以及重复

string length 命令计算字符串中字符的个数，然后返回这个数字。

```
    string length "sample string"
⇒ 13
```

string toupper 命令将字符串中所有小写字母转换为大写字母。string tolower 命令将字符串中所有大写字母转换为小写字母。

```
    string toupper "Watch out!"
⇒ WATCH OUT!
    string tolower "15 Charing Cross Road"
⇒ 15 charing cross road
```

string 命令提供了三种裁剪方式：trim、trimleft 和 trimright。每种方式都还需要两个参

数，一个指定要裁剪的字符串，一个指定裁剪方式。string trim 命令把在开头和结尾出现的要裁剪的字符都删去，返回删除后的字符串作为结果。

```
    string trim aaxxxbab abc
⇒ xxx
```

trimleft 和 trimright 的工作方式也是一样的，只不过它们只从字符串的开头或结尾进行裁剪。裁剪命令在移除过多的空白符时最为常用；如果没有指定要裁剪的字符，默认的就是空白符(空格、制表符、换行符以及换页符)。

另一个字符串工具命令是 string repeat，将现有字符串重复指定次数形成新字符串，然后将其返回。

```
    string repeat "*" 20
⇒ ********************
    string repeat "abc" 5
⇒ abcabcabcabcabc
```

5.4 简单搜索

命令 string first 读取两个字符串参数，示例如下：

```
    string first th "There is the tub where I bathed today"
⇒ 9
    string first th "There is the tub where I bathed today" 12
⇒ 27
```

它在第二个字符串中搜索与第一个字符串相同的子字符串。如果找到，返回最左边的相同子字符串的开头字符的索引值；如果找不到，返回-1。搜索从第二个字符串的开头字符开始，也可以用一个参数来指定开始搜索的字符的索引。

命令 string last 与此相似，只不过它返回的是最右边的相同子字符串的开头字符的索引值。

```
    string last the "There is the tub where I bathed today"
⇒ 27
```

提示：使用正则表达式可以进行更复杂的搜索，详见 5.11 节。

5.5 字符串比较

命令 string compare 读入两个字符串参数，并对它们进行比较。如果字符串相同，返回 0；如果第一个字符串在字典中先于第二个字符串，返回-1；如果第一个字符串在字典中后于第二个字符串，返回 1。

```
    string compare Michigan Minnesota
⇒ -1
    string compare Michigan Michigan
⇒ 0
```

string equal 命令对两个字符串进行简单的字符串比较，如果它们严格相同，就返回

1，否则返回 0。string compare 和 string equal 都是区分大小写的，除非特别指定-nocase 选项。您还可以设置-length 选项，指定只对前 length 个字符进行比较。

```
    string equal cat cat
⇒   1
    string equal dog Dog
⇒   0
    string equal -nocase dog Dog
⇒   1
    string equal -length 3 catalyst cataract
⇒   1
```

5.6 字符串置换

您可以使用 string replace 命令进行简单的字符串置换。它接受一个字符串作为参数，以及要删除的字符序列的开头和结尾索引，还可选地接受一个字符串参数作为置换用字符串。

```
    string replace "San Diego, California" 4 8 "Francisco"
⇒   San Francisco, California
    string replace "parsley, sage, rosemary, and thyme" 0 8
⇒   sage, rosemary, and thyme
```

string map 命令根据字典把字符串中的相应文本置换为值。这可以用于模板功能。其基本语法如下：

```
string map dictionary string
```

string map 命令将 *string* 中出现的所有 *dictionary* 关键字置换为相应的值，返回置换后的字符串。置换是按顺序进行的，列表中先出现的关键字先处理。只对字符串迭代一次，所以前面进行的置换不会影响接下来的匹配查找，例如：

```
    set entities {
        & &amp;
        ' &apos;
        > &gt;
        < &lt;
        \" &quot;
    }
    string map $entities {if (index > 0 && nbAtts == 0)}
⇒   if (index &gt; 0 &amp;&amp; nbAtts == 0)
```

使用-nocase 选项，就不区分关键字的大小写，例如：

```
    string map -nocase \
    { RESOURCE "Ms. Ripley" CORPORATION "Weyland-Yutani" }\
    "Dear ResouRcE, welcome to your first day at corporation"
⇒   Dear Ms. Ripley, welcome to your first day at Weyland-Yutani
```

提示：可以使用正则表达式进行更复杂的置换操作，详见 5.12 节。

5.7 确定字符串类型

在操纵字符串时，常常需要确定其值的类型是否正确，例如它是不是一个数。使用 string is 命令就可以完成这一任务，它分析一个字符串，如果它是数字或指定类型的字符，就返回 1，否则返回 0。例如：

```
    string is digit 1234
⇒   1
    string is digit "A man, a plan ,a canal. Panama."
⇒   0
```

默认情况下，如果字符串为空，对任何类型 string is 都返回 1。使用-strict 选项可以强制要求在字符串为空时 string is 返回 0。

```
    string is control ""
⇒   1
    string is control -strict ""
⇒   0
```

选项-failindex 允许您指定一个变量，如果测试失败则设置这个变量。这里，命令会把这个变量设置为字符串中第一个不能通过测试的字符的索引，例如：

```
    string is digit -failindex idex "123c5"
⇒   0
    puts $idex
⇒   3
```

表 5.1 列出了 string is 命令支持的字符类型。

表 5.1 string is 命令支持的字符类型

类 型	测试对象
Alnum	全为 Unicode 字母或数字
alpha	全为 Unicode 字母
ascii	全为 7 位 ASCII 字符
boolean	可识别为布尔型值(0、false、no、off、1、true、yes 以及 on)
control	全为 Unicode 控制字符
digit	全为 Unicode 数字
double	双精度浮点值(忽略前后的空白符)。如果检测到上下界溢出，则将-failindex 变量值设为-1
false	可识别为布尔型值的“非”(0、false、no、off)
graph	全为非空白的 Unicode 打印字符
integer	32 位整型值(忽略前后的空白符)。如果检测到上下界溢出，则将-failindex 变量值设为-1
list	一个有效的列表结构。如果列表结构不正确，-failindex 变量会设置为列表中第一个导致结构无效的元素
lower	全为 Unicode 小写字母
print	全为 Unicode 打印字符(包括空白)

续表

类　型	测试对象
punct	全为 Unicode 标点符号
space	全为 Unicode 空白符号
true	可识别为布尔型值的“是”(1、true、yes、on)
upper	全为 Unicode 大写字母
wideinteger	长整型变量(忽略前后的空白)。如果检测到上下界溢出，则将-failindex 变量设置为-1
wordchar	全为字母和连接符(主要就是指下划线)
xdigit	全为十六进制数字，包括 0～9，a～f，以及 A～F

提示： string is 命令用 Unicode 规定在测试字符。这一点需要注意，例如，Unicode 的数字字符就不止 ASCII 字符的 0～9。

5.8　用 format 创建字符串

Tcl 的 format 命令提供与 ANSI C 库中的 sprintf 程序相似的功能。例如：

```
    format "The square root of 10 is %.3f" [expr sqrt(10)]
⇒ The sqare root of 10 is 3.162
```

format 的第一个参数是格式字符串，其中可以包含任意多个像%.3f 这样的转换符。针对每一个转换符，format 会改动后面的参数的格式，然后用它置换相应的转换符。format 命令的结果是这个格式字符串中的转换符被置换成相应的字符串。上面这个示例中，%.3f 就指明了下一个变量(expr 命令的结果)的格式应该是有三位小数的实数。format 基本上支持 ANSI C 的 sprintf 定义的所有转换符，如%d 表示十进制整数，%x 表示十六进制整数，%e 表示指数形式的实数。转换符的完整列表参见相关手册。

在 Tcl 中 format 命令的重要性不像 C 语言中的 printf 那样显著。很多时候使用 printf 和 sprintf 就是为了把二进制数值转化为字符串，从而进行字符串替换操作。Tcl 不需要这一转化，因为这些值已经存储为字符串形式了，字符串替换操作可以直接由 Tcl 解释器完成。例如以下命令：

```
set msg [format "%s is %d years old" $name $age]
```

可以写成如下更简洁的形式：

```
set msg "$name is $age years old"
```

format 命令中的%d 转换符也可以用%s 代替，format 会把 age 的值转化为二进制整数，然后又会把它转化为一个字符串。

在 Tcl 里使用 format，通常是为了改变一个值的格式，以改善显示效果，或者把它从一种表现形式改为另一种表现形式(例如上面的例子，从十进制改为十六进制)。作为改变格式的一个示例，下面这段脚本输出 e 的前十次幂的列表。

```
puts "Number Exponential"
for {set i 1} {$i <= 10} {incr i} {
```

```
    puts [format "%4d %12.3f" $i [expr exp($i)]]
}
```

这个脚本会在标准输出端产生如下输出：

```
Number      Exponential
   1           2.718
   2           7.389
   3          20.085
   4          54.598
   5         148.413
   6         403.429
   7        1096.630
   8        2980.960
   9        8103.080
  10       22026.500
```

转换符%4d 让第一列的制表位宽度正好可以显示 4 个数字，看上去它们就呈末位对齐排列。转换符%12.3f 让显示实数的列宽正好可以显示 12 个数字，并且也使这些实数右对齐；它还指定了显示精度为 3 位小数。

format 的第二项主要用途是改变一个值的表现形式，可由下面的脚本展示。这一脚本输出了与特定的整型值对应的 ASCII 字符。

```
puts "Integer ASCII"
for {set i 95} {$i <= 101} {incr i} {
    puts [format "%4d        %c" $i $i]
}
```

这个脚本在标准输出端输出如下内容：

```
Integer ASCII
  95       _
  96       `
  97       a
  98       b
  99       c
 100       d
 101       e
```

在 format 命令中，i 的值被使用了两次，一次是%4d，一次是%c。%c 指明读入整型参数，将其表现形式改为 ASCII 中与该整数相对应的字符。

这个示例过程重复使用了 i 的值，也可以在格式字符串中使用位置符来完成相同的过程。如果%是转换符，并且紧跟着一个整数和$符号，如%2$d，那么要转换的就不是下一个参数了，而是由这个数字代表的参数，这里 1 代表的是第一个参数。有了位置符，就可以把同一个参数使用任意次。然而，只要格式字符串中有一个转换符使用了位置符，那么所有的转换符都必须使用位置符。下面这个过程输出的结果与前面的脚本相同。

```
puts "Integer ASCII"
for {set i 95} {$i <= 101} {incr i} {
    puts [format "%1$4d        %1$c" $i]
}
```

5.9 用 scan 解析字符串

scan 命令提供的功能与 ANSI C 库中的 sscanf 程序几乎完全一样。scan 可以说就是 format 的逆操作。它从一个有格式的字符串开始，在格式字符串控制下解析这个字符串，取得与格式字符串中%转换符相对应的字段，把这些字段的值置于 Tcl 变量中。例如，在执行下面这条命令以后，变量 a 的值为 16，变量 b 的值为 24.2。

```
scan "16 units, 24.2% margin" "%d units, %f" a b
⇒ 2
```

scan 获取的第一个参数是待解析的字符串，第二个参数是控制解析方式的格式字符串，其他的参数是用来存储转换出的值的变量。返回值 2 说明成功完成了两次转换。

scan 操作同时扫描字符串和格式。除了被忽略的空格和制表符及%字符，格式必须与字符串中对应的字符匹配。format 中的%标记着转换符的开始：scan 会把下一个输入字符按转换符的说明进行转换，将其结果存入 scan 给定的下一个变量中。除了少数几种情况，如%c，字符串中的空白都会被忽略。关于转换符语法的完整信息请参阅手册。

scan 的一个常见用途是简单的字符串解析，例如，在上述示例中。另一个常见用途是将 ASCII 字符转换为它们对应的整型值，即使用%c 转换符。下面这一过程就使用该功能返回按字典顺序紧接着给定字符的下一个字符。

```
proc next c {
    scan $c %c i
    format %c [expr {$i+1}]
}
next a
⇒ b
next 9
⇒ :
```

scan 命令把 c 参数的值从 ASCII 字符转换为用于表示该字符的相应整数，再把这个整数加一，然后使用 format 命令把它转换回 ASCII 字符。

scan 还有一种典型的用法，把一个可能由 0 开头的数字组成的字符串强制转换为十进制整数，通常情况下开头的 0 会让 Tcl 把该数字字符串解析为八进制的数值以供计算使用。下面这个过程强制将一个数字字符串解析为十进制数。

```
proc forceDecimal {x} {
    set count [scan $x {%11d %c} n c]
    if {$count !=1 } {
        error "not an integer: \"$x\""
    }
return $n
}
set val 0987
expr {$val + 1}
Ø can't use invalid octal number as operand of "+"
expr { [forceDecimal $val] + 1}
⇒ 988
forceDecimal xyz
Ø not an integer: "xyz"
```

在这个 forceDecimal 实现中，使用了 scan 命令处理参数格式，可以把跟在数字字符后面的非数字字符识别出来。如果只是用%d 作为格式参数，那么 scan 命令会把字符串 123xyz 中的 123 识别出来。转换符%11d 中的 11 指定了用于存储整型数的空间的大小，它支持无限精度。如果没有这个 11，转换后的值的存储空间会被限制为正在运行的系统中的机器字的大小。

```
    set val 01234567890123456789012345678901234567890
    expr { $val + 1}
Ø   can't use invalid octal number as operand of "+"
    expr { [forceDecimal $val] + 1}
⇒   12345678901234567890123456789012345678901
```

5.10 通配符样式的模式匹配

Tcl 有两种模式匹配方式，较简单的一种称为“通配符”样式。这个名称源于 Unix 外壳中的文件名展开机制，这个机制使用的符号也称为“通配符”。通配符样式匹配相对易学，比后面两节介绍的正则表达式更易用，而且在简单情况下工作得很好。对于更复杂的模式匹配，可能需要使用正则表达式。

命令 string match 就实现了通配符样式的匹配。

```
string match ?-nocase? pattern string
```

命令 string match 在模式与字符串匹配时返回 1，否则返回 0。对用于匹配的模式，除了少数几个会被特别处理的字符之外，模式中的字符必须与字符串中相应的字符相同。例如，模式中的*可以和任意长度的子字符串匹配，所以模式中的 Tcl*可以和以 Tcl 三个字母开头的任何字符串匹配。表 5.2 列出了通配符样式的匹配中所支持的特殊字符。除非指定-nocase 选项，否则匹配是区分大小写的。

表 5.2 string match 命令的通配符样式匹配中的特殊字符

字 符	说 明
*	可以与零个或多个任意字符组成的字符串匹配
?	可以与一个任意字符匹配
[*chars*]	与 *chars* 中的任意一个字符匹配。如果 *chars* 的内容包括范围表达式 *a-b*，那么 *a* 到 *b* 之间的任意一个字符都可以匹配，包括 *a* 和 *b*
\x	与单个字符 *x* 匹配。可以用于指定会被特殊处理的字符，如*?[]\

下面的示例展示了 string match 命令的使用。

```
    string match a* alphabet
⇒   1
    string match a* bat
⇒   0
    string match {[ab]*} brown
⇒   1
    string match a* Arizona
⇒   0
    string match -nocase a* Arizona
⇒   1
```

```
    string match {*\?} "Wow!"
⇒ 0
    string match {*\?} "What?"
⇒ 1
```

通配符样式的模式可以完成很多简单的任务。例如，*.[ch]就可以用来匹配所有以.c或.h 结尾的字符串。然而，更复杂的匹配模式无法由通配符样式完成。例如，无法由通配符样式的模式检测一个字符串是否包含了所有的数字：模式[1-9]只能与单个数字匹配，却不能指明多个数字需要用于匹配。

5.11 使用正则表达式进行模式匹配

Tcl 进行模式匹配的第二种方法就是使用正则表达式，这比通配符样式的模式更复杂，也更强大。Tcl 的正则表达式基于 Henry Spencer 公开的实现方法，下面的部分介绍也直接来自 Spencer 的文档。

Tcl 支持三类正则表达式，分别称为基本正则表达式(BRE)、扩展正则表达式(ERE)和高级正则表达式(ARE)。BRE 和 ERE 主要是在过去的版本中使用。ERE 由 POSIX 定义，而 ARE 受到了 Perl 的一点启发。从 Tcl 8.1 开始，所有的 Tcl 命令都默认支持 ARE 语法。因此，本书接下来介绍的都是 ARE 语法的使用。有关 BRE 和 ERE 的更多信息请查阅参考文献。

提示： 正则表达式语法可能相当复杂。本书介绍了 Tcl 中最基本的正则表达式语法和应用。如要了解完整的正则表达式语法，应该阅读相关参考文献。可以到网站 http://www.regular-expressions.info 和 http://regexlib.com 了解更多信息。另外 Jeffrey E. F. Friedl 所著的 *Mastering Regular Expressions*, Third Edition(ISBN 0-596-52812-4) 对正则表达式语法和应用进行了深入讨论。

5.11.1 正则表达式的原子

正则表达式模式可能有多层结构。而它的基本单位称为原子，正则表达式最简单的形式就是由一个或几个原子组成。如果正则表达式与输入的字符串匹配，那么输入的字符串中必须含有可与各个原子(或者与别的组件匹配，后面会介绍这些组件)对应匹配的子字符串。大多数情况下原子就是一个字符，分别用于匹配。因此正则表达式 abc 与含有 abc 的字符相匹配，如 abcdef 和 xabcy。

在正则表达式中很多字符有特殊含义，表 5.3 对此进行了总结。字符^和$称为约束符，分别与输入字符串的首尾字符进行匹配。因此^abc 与任意以 abc 开头的字符串匹配，abc$与任意以 abc 结尾的字符串匹配，而^abc$则只与字符串 abc 匹配。类似地，转义序列\m 和\M 也分别与一个单词的开头和结尾匹配。因此\mcat\M 只与单词 cat 匹配，而不能与 catalog 或 concatenate 中的 cat 匹配。

原子.与任意单个字符匹配，而原子\x 中 x 可以是任意一个字符，与 x 匹配。例如，正则表达式.\$与任意包含一个$符号，且该$符号不是第一个字符的字符串匹配，而\.$与任意

以句点结尾的字符串匹配。

Tcl 的高级正则表达式包含更多的字符-条目转义集合，均以反斜线作为前缀。可以使用这些转义序列指定一些难以输入的字符。表 5.4 列出了这些特殊转义序列。

表 5.3 正则表达式模式中接受的特殊字符

字 符	说 明
.	匹配任意的单个字符
^	指定与输入字符串的开头进行匹配
$	指定与输入字符串的结尾进行匹配
\m	指定与单词的开头进行匹配
\M	指定与单词的结尾进行匹配
\k	匹配非字母也非数字的字符 k(如\.匹配字面上的句点符号)
\c	当 c 是字母或数字时(c 后面可能还跟着其他内容)，用 c 替换\c
[chars]	与 chars 中的任意单个字符匹配。如果 chars 中的第一个字符是^，那么这个模式与 chars 中剩余字符以外的任何字符匹配。chars 中的范围形式 a-b 作为 ASCII 字符 a 到 b 的所有字符的简写，包括 a 和 b。如果 chars 中的第一个字符是](后面可能还跟了一个^)，它会被作为文本处理(作为 chars 的一部分而不是终止符)。如果~出现在 chars 的第一个或最后一个，它会被作为文本处理
(regexp)	匹配正则表达式 regexp。用于分组或识别匹配的子字符串
*	与由零个或多个前述的原子组成的序列匹配
+	与由一个或多个前述的原子组成的序列匹配
?	与空字符串匹配，或与前述的一个原子匹配
{m}	匹配与前述原子正好匹配 m 次的序列
{m,}	匹配与前述原子至少匹配 m 次的序列
{m,n}	匹配与前述原子至少匹配 m 次、最多匹配 n 次(包括 m 和 n)的序列
re1|re2|...	与任意一个指定正则表达式匹配

表 5.4 正则表达式中的字符-条目转义序列

转义序列	表示的内容
\a	铃声、警报声、字符
\b	退格
\B	反斜线，与\相同；用于整理正则表达式语法
\cX	取得给定字符 X 的低 5 位，然后以这 5 位为低 5 位，其他位补 0 作为转义后的字符
\e	转义字符
\f	换页
\n	换行
\r	回车
\t	制表符

续表

转义序列	表示的内容
\u*wxyz*	由给定的四位十六进制数指定的 Unicode 字符
\v	垂直制表符
\x*hhh*	由给定的十六进制数作为 ASCII 码值指定的字符
\0	空、零、字符
xyz	由给定的两位或三位八进制数作为 ASCII 码值指定的字符，除非该值被指定为逆向引用，有关逆向引用的信息请查阅参考文献

除了前面已经讲述的原子，正则表达式中还有两种类型的原子。第一种类型由圆括号中的正则表达式组成，如(a.b)。圆括号用于分组，一对圆括号中的正则表达式常被称为子表达式或子模式。*这样的操作符可以用于整个子表达式，就像用于原子一样。它们还被用于识别特殊过程中的子字符串匹配。后面会详细讨论这些用途。

原子还有一种类型：范围，即方括号中的字符之间的内容的集合。范围与方括号中的任意一个字符匹配。而且，如果范围中有 a-b 这种形式的序列，那么所有 ASCII 码在 a 和 b 之间的字符都在这个集合当中。因此，正则表达式[0-9a-fA-F]与任意含有十六进制数的字符串匹配。如果在[后面的字符是^，那么就反转范围：即只与不含^到]之间的字符的字符串匹配。

在用方括号表示的范围中，还可以使用“字符类型”来代表一种类型的全部字符。例如[:alpha:]的范围就是全部的字母。注意全部的字母不只是 ASCII 字母[A-Za-z]，还包括了所有的 Unicode 字母字符。表 5.5 列出了可用的字符类型。注意在范围定义中，可以联合使用明确规定的字符与一个或多个字符类型，例如：

```
[[:alpha:][:blank:],;.!?]
```

表 5.5　正则表达式中的字符类型

类　型	说　明
[:alpha:]	字母
[:alnum:]	字母和数字
[:blank:]	空格和制表符
[:cntrl:]	控制字符
[:digit:]	十进制数字
[:graph:]	图形字符，有对应的可视表现形式的字符，如数字、字母字符或标点字符
[:lower:]	小写字母
[:print:]	可打印字符，包括[:graph:]类型中的全部字符再加上空白符
[:punct:]	标点字符
[:space:]	空白字符
[:upper:]	大写字母
[:xdigit:]	十六进制数字

为了编写方便，还可以使用表 5.6 中列出的转义字符作为字符类型的简写。注意\w(字母数字)和它的反转\W 转义序列在定义时都把下划线_视为字母字符之一。

表 5.6 正则表达式中字符类型的转义序列

转义序列	表示的类型
\d	[[:digit:]]
\D	[^[:digit:]]
\s	[[:space:]]
\S	[^[:space:]]
\w	[[:alnum:]_]
\W	[^[:alnum:]_]

5.11.2 正则表达式的分支和量词

正则表达式可以用操作符|连接。可与|连接的某个正则表达式匹配的字符串，则可与所得的正则表达式匹配。用|连接起来的那些正则表达式称为分支。例如，下面这个模式与包含 this、that 或 other 的字符串匹配。

```
this|that|other
```

注意，各个分支可以是任意的正则表达式，包括子表达式，因此可以构建相当复杂的结构。如果两个或多个分支都可以匹配，那么 Tcl 会选择最长的进行匹配。可以用圆括号把分支括起来，避免分支变得太长。例如，下面这个模式与字符串 this 或 that car 匹配。

```
this|that car
```

而以下模式与 this car 或 that car 匹配。

```
(this|that) car
```

操作符*、+、?被称为量词，可以跟在原子后面，用来指定它的重复次数。一个后面跟着*的原子与由零个或多个该原子组成的序列匹配。一个后面跟着+的原子与由一个或多个该原子组成的序列匹配。一个后面跟着?的原子与空字符串或与这个原子匹配的字符串匹配。例如：^(0x)?[0-9a-fA-F]+$与有效的十六进制数匹配，即由可选的 0x 开头，后面跟着一个或多个十六进制数字。

另一组量词，被称为边界，用于确切指定在匹配时原子应该重复出现几次。在一对大括号内，可以指定原子至少重复多少次，可选地指定原子最多重复多少次，指定的重复次数是 0～255 之间的整型数。{*num*}与重复了 *num* 次的该原子组成的字符串严格匹配，{*min*,}与至少重复了 *min* 次原子的字符串匹配，{*min*,*max*}与至少重复了 *min* 次，至多重复了 *max* 次原子的字符串匹配。

当一个正则表达式可以与给出的字符串中的多个子字符串匹配时，正则表达式与字符串中最先开始的子表达式匹配。然后，根据正则表达式引用，它再与最长的或最短的子字符串匹配。正则表达式的量词设置默认为“贪婪”，即与尽可能多的字符进行匹配。任意量词都可以在后面跟一个?，将其设置为非贪婪，从而与尽可能短的子字符串进行匹配。因此，+和{3,7}是贪婪量词，而+?和{3,7}?是非贪婪量词。

提示：在 Tcl 中，不能在一个正则表达式中混合使用贪婪和非贪婪模式。整个正则表达式的引用模式由表达式中的第一个量词确定。如果允许混合使用，将导致一个过程出现歧义，因此 Tcl 正则表达式实现时设计为不允许混合使用贪婪和非贪婪模式。

5.11.3 逆向引用

在一个正则表达式中，逆向引用匹配与前面一个子表达式相同的字符集。逆向引用的语法是在\后面跟一个前面的子表达式的编号。子表达式依据它们的左括号进行编号。例如，表达式([ab])\1 与字符串 aa 或 bb 匹配，但不与 ab 或 ba 匹配。下面演示这一点，考虑要编写一个正则表达式与括在单引号或双引号中的字符串匹配。一种方法是使用两个分支。

```
'.*? '|".*?"
```

另一种方法，您可以用一个子表达式“捕获”第一个引号，然后用逆向引用来与对应的引号匹配。

```
(['"]).*?\1
```

另一个示例是查找意外重复的单词，例如 the the。

```
\m(\w+)\M\s+\m\1\M
```

5.11.4 非捕获子表达式

通常您会需要一些只用于“结构”目的的子表达式，例如限定分支，或给一个序列指定量词，但并不需要对该子表达式进行逆向引用，也不需要解析，本节就讨论这样的表达式。这些情况下，可以使用非捕获子表达式，其格式为(?:expression)。例如，下面这两个正则表达式都匹配同样的字符串。

```
((ab){1,3}|(cd)+) xyz
(?:(?:ab){1,3}|(?:cd)+) xyz
```

非捕获子表达式的优点是处理速度更快。缺点是不能逆向引用非捕获子表达式，与非捕获子表达式匹配的子字符串也不能作为独立的匹配变量取出，下一节就介绍这种取出操作。

5.11.5 regexp 命令

regexp 命令调用正则表达式匹配。它最简单的形式要获取两个参数：正则表达式模式和输入字符串。它返回 0 或 1，代表输入字符串能否匹配模式。

```
    regexp {^[0-9]+$} 510
⇒ 1
    regexp {^[0-9]+$} -510
⇒ 0
```

提示：这里的模式被括在大括号中，使得字符$、[以及]被传给命令 regexp，而不会触发变量替换和命令替换。建议总是把正则表达式括在大括号中。

如果 regexp 在输入字符串后还调用更多的参数，每一个参数都被作为一个变量名对待。第一个变量会存入与整个正则表达式匹配的子字符串。第二个变量会存入与捕获到的第一个子表达式相匹配的子字符串；第三个变量会存入与捕获到的下一个子表达式相匹配的子字符串；以此类推。如果变量名的数量比捕获到的子表达式数量更多，多余的变量会被设置为空字符串。例如，执行如下命令后变量 a 的值为 10km，b 的值为 10，c 的值为 km。

```
regexp {([0-9]+) *([a-z]+)} "Walk 10 km" a b c
```

这种取得部分配对的子字符串的能力使 regexp 可以用于解析工作。

还可以在正则表达式前为 regexp 指定更多的选项。您可以使用-start 选项，在后面跟一个字符串中的字符索引，指定 regexp 从该位置开始查找匹配。-all 选项告诉 regexp 在字符串中查找尽量多次的匹配，并返回总的匹配次数。-nocase 选项指定了模式中的字母在字符串中查找匹配时不区分大小写。

-indices 选项指明额外的变量不应该用于存放匹配的子字符串的值，而是存放给出子字符串范围的首字符和尾字符的索引列表。执行下面这个命令后，变量 a 的值为 5 9，b 的值为 5 6，c 的值为 8 9。

```
regexp -indices {([0-9]+) *([a-z]+)} "Walk 10 km" \
        a b c
```

-inline 选项让 regexp 把匹配变量返回为一个数据列表。例如，下面这条命令返回了含有三个元素的列表，第一个元素是与整个表达式匹配的字符，后面两个元素是与捕获到的子表达式匹配的字符。

```
    regexp{([0-9]+) *([a-z]+)} "Walk 10 km"
⇒  {10 km} 10 km
```

-line 选项激活区分换行的匹配。指定这个选项后，[^括号表达式和.绝不会与新行匹配，^原子除了它的普通功能外，与新行后的空字符串匹配，$原子除了它的普通功能外，与新行前的空字符串匹配。作为示例，下面这条命令返回了所有以 ERROR 字符串开头(这个字符串前面可能有空白)的行数。

```
regexp -all -line -- {^[[:blank:]]*ERROR:} $text
```

提示： regexp 命令也支持用--选项明确地标志选项结束。在实际工作中建议总是使用--选项；否则如果您的模式以-字符开头，它可能被错误地解析为另一个选项。

5.12 使用正则表达式进行替换

使用 regsub 命令，正则表达式也可以用于替换。考虑如下命令：

```
    regsub there "They live there lives" their x
⇒  1
```

regsub 的第一个参数是正则表达式模式，第二个参数是输入字符串，与 regexp 相同。而且，与 regexp 相似，regsub 返回 1 代表字符串与模式匹配，返回 0 代表不匹配。然而，

regsub 所完成的工作不仅仅是查找匹配：它把匹配的子字符串替换为指定的替换字符串，从而创建一个新的字符串。替换字符串就是 regsub 的第三个参数，存放这个新的字符串的变量名即 regsub 的最后一个参数。因此，在上面这个命令完成后，x 的值为 They live their lives。如果字符串不能与模式匹配，则会返回 0，x 的值为 They live there lives(无论是否发生替换，都会设置这个变量)。

在正则表达式参数之前，还可以提供一个或多个选项，regexp 支持的很多选项在 regsub 中也可用。通常 regsub 只进行简单的替换，替换掉输入字符串中最先出现的匹配项。然而，如果指定了-all 选项，regsub 就会继续查找，直到把所有的匹配项都加以替换，然后它会返回替换的次数。例如，执行以下命令：

```
regsub -all a ababa zz x
```

x 的值为 zzbzzbzz。如果没有-all 选项，x 的值为 zzbaba。

-nocase 选项要求在模式中的字母原子进行匹配时不区分大小写。-start 命令指定字符串中一个字符的索引，从指定字符处开始查找匹配。-line 选项启动区分换行的匹配，和 regexp 相同。您可以使用--来明确地标记命令选项结束。

前面的示例中，替换字符串只是简单的文本。然而，如果替换字符串包含&或\0，其中的&或\0 会被替换为与正则表达式相匹配的子字符串。如是\n 形式的序列出现在置换字符串中(n 是一个十进制数)，与第 n 个捕获的子表达式相匹配的子字符串会替换掉这个\n。例如，以下命令会把输入字符串中所有的 a 和 b 都写两遍。

```
regsub -all -- a|b axaab && x
```

这里它把 x 赋值为 aaxaaaabb。另外，以下命令会把后面有一个 b 而前面没有一个 b 的 a 的序列替换为一个 z。

```
regsub -all -- (a+)(ba*) aabaabxab {z\2} x
```

这里 x 被赋值为 zbaabxzb。反斜线可以用来代入有特殊含义的字符，避免解释器对它们进行特殊处理。

提示： *像前面的例子那样，把复杂的替换字符串用括号括起来是一个好办法；否则 Tcl 就会处理反斜线序列，由 regsub 接收到的替换字符串可能就会缺少必要的反斜线了。*

5.13 字符集专题

字符编码就是编写语言使用的字符与计算机使用的二进制代码之间的映射。以标准 ASCII 码为例，拉丁字母 A 表示为十六进制数 0x41。其他广泛使用的编码包括欧洲语言使用的 ISO 8859-1，日文使用的 Shift-JIS 和 EUC-JP，中文使用的 Big5。

Unicode 标准是包含这个世界上几乎所有主要书面语言的统一编码方案。它的文本元素包括希腊文和罗马文，日文的平假名和片假名，以及像中文这样的象形文字。有关 Unicode 的更多信息，请访问 Unicode 官方网站 http://www.unicode.org。

UTF-8 是 Unicode 字符的标准变形格式。可以用它来把所有的 Unicode 字符变形为变

长的码值；一个 Unicode 字符可以由一个或几个字节表示。UTF-8 的优势是它和 Unicode 标准的设计保证了与标准 7 位 ASCII 集(到 ASCII 码十六进制值 0x7F 止)对应的字符在 UTF-8 和 Unicode 编码中都有相同的码值。换句话说，在 UTF-8 和 Unicode 编码中，大写字母 A 的码值都是 0x41。

Tcl 8.1 版把所有的字符串内部存储为 UTF-8 格式的 Unicode 字符。Tcl 还内建支持数十条有关字符编码标准的命令，可以把字符串从一种编码转换成另一种。encoding names 命令返回可识别的编码的名称列表。参考文献描述了如何增加对更多编码的支持。

5.13.1 字符编码和操作系统

系统编码是操作系统为自己的文件名和环境变量使用的字符编码。文本编辑器和其他应用程序使用的文本文件通常也使用系统编码，除非应用程序在生成文件时明确指定使用其他格式存储(例如，在一个 ISO 8859-1 的系统上使用 Shift-JIS 文本编辑器)。

在与操作系统通信时，Tcl 的内建命令可以自动把 UTF-8 格式的字符串转换为系统编码，反之亦然。例如，如果执行下面的命令，Tcl 会根据需要自动管理编码转换。

```
glob *
set fd [open "Español.txt" w]
exec myprog << "¡Bienvenido a Tcl!\n"
```

默认情况下，Tcl source 命令使用系统编码读取文件。如果脚本文件使用的不是系统编码，可以用 source -encoding 选项提供所用的编码名称。

```
source -encoding shiftjis script.tcl
```

Tcl 在初始化时会根据平台和具体设置确定系统编码。Tcl 通常会基于这些设置确定一种适用的系统编码，不过如果因为什么原因它不能选定，就会使用 ISO 8859-1 编码作为默认选择。

提示：可以使用 encoding system 命令让 Tcl 改变默认的系统编码，但您应该尽可能避免使用这条命令。如果把 Tcl 的系统默认编码修改得和您的当前系统实际使用的编码不同，Tcl 可能会无法与您的操作系统正常通信。

5.13.2 编码和通道输入/输出

由通道读写数据时，需要确保 Tcl 为通道选用了适当的字符编码。新开通道默认编码与系统编码相同。多数情况下，您不需要为读写数据进行特殊处理，因为多数文本文件都是以系统编码创建的。只在需要访问的文件的编码与系统编码不同时才需要特殊处理(例如：当您的系统编码是 ISO 8859-1 时读取一个编码格式为 Shift-JIS 的文件)。

fconfigure -encoding 选项允许您为一个通道指定编码。因此，要读取编码格式为 Shift-JIS 的文件，可以执行如下命令：

```
set fd [open $file r]
fconfigure $fd -encoding shiftjis
```

Tcl 会把从文件中读取到的所有文本自动转化为标准 UTF-8 格式。类似地，如果向通道中写入信息，也可以用 fconfigure -encoding 指定目标字符编码，在输出时 Tcl 会自动把 UTF-8 的字符串转化为指定编码格式。

5.13.3 转化字符串的编码格式

您可以使用 encoding convertfrom 和 encoding convertto 命令转化字符串的编码格式。命令 encoding convertfrom 将一个指定编码的字符串转化为 UTF-8 Unicode 字符串；命令 encoding convertto 把一个 UTF-8 Unicode 编码的字符串转化为指定编码格式。在这两个命令中，如果不指定编码参数，默认值都是当前系统编码。

作为示例，下面这条命令将表达日文片假名 HA 的字符从 EUC-JP 编码转换到 UTF-8 编码。

```
set ha [encoding convertfrom euc-jp "\xA4\xCF"]
```

5.14 消息目录

除了操作不同的字符集编码，国际化应用程序开发面临的另一个挑战则是如何为用户和其他应用程序提供一个本地化的界面。为了帮助创建本地化的应用程序，Tcl 集成了 msgcat 工具包，它提供了一系列管理多语言用户界面的功能。它允许您在消息目录中定义字符串，消息目录与您的应用程序和发布包相独立，您可以对消息目录进行修改和本地化，而不必改动应用程序源码。

msgcat 工具包的基本工作原理是：您创建一系列消息文件，每一个文件对应一种语言，包含您的应用程序或发布包所要显示的全部字符串的本地化版本。然后在您的应用程序或发布包中，不要直接地使用字符串，而是调用::msgcat::mc 命令返回您想要的字符串的本地化版本。

5.14.1 使用消息目录

要在您的应用程序或发布包中使用消息目录，必须先用如下命令加载 msgcat 工具包：

```
package require msgcat
```

默认情况下，Tcl 会根据用户的环境设置来确定区域设置值，msgcat 的参考文档中有详细讨论。如果 Tcl 无法通过用户环境确定区域设置值，那区域设置值会默认设置为 C。您也可以选择::msgcat::mclocale 命令明确地指定区域设置值。

```
::msgcat::mclocale ?newLocale?
```

如果不指定 *newLocale* 参数，那么 mclocale 返回当前的区域设置值。一个区域设置字符串由一个语言代码、一个可选的国家代码和一个可选的修饰语组成，均用下划线隔开。国家和语言代码由 ISO-639 和 ISO-3166 标准规定。例如，en 表示英语，en_US 表示美国英语，en_GB 表示英国英语。

提示：mclocale 命令在实现语言菜单这类功能时很有用，用户可以通过该菜单选择您的应用程序所显示的语言。

下一步就是调用::msgcat::mcload 加载适当的消息文件。mcload 命令需要一个路径参数，用于指定消息文件。下一节将介绍消息文件的格式。

加载消息文件以后，在您的脚本中需要指定用于显示的字符串的地方，都改用::msgcat::mc 命令。mc 的命令需要一个参数，即源字符串，然后返回该字符串翻译为本地语言的结果。

下面这段代码演示了如何在脚本中使用 msgcat 工具包。

```
package require msgcat

# Use the default locale as determined by the system.
# You could explicitly set the locale with a command such as
# ::msgcat::mclocale "en_GB"

# Load the message files. In this example, they are stored
# in a subdirectory named "msgs" which is in the same
# directory as this script.

::msgcat::mcload \
    [file join [file dirname [info script]] msgs]

# Display a welcome message

puts [::msgcat::mc "Welcome to Tcl!"]
```

这个示例中，应用程序不是直接显示消息“Welcome to Tcl!”，而是调用 mc 获取该字符串的本地版本。例如，在 es 区域设置下 mc 会返回西班牙语的欢迎词“¡Bienvenido a Tcl!”。

mc 命令在返回翻译的字符串时执行“最佳匹配”搜索。如果它不能为用户的环境设置或 mclocale 命令指定的区域设置找到严格匹配，它会逐渐放宽匹配模式进行搜索。例如，如果区域设置值是 en_GB_Funky，那么 en_GB_Funky、en_GB、en 以及""(空字符串，对应的区域设置为 ROOT)会依次用于搜索，直到找到一个匹配的翻译字符串为止。如果在当前命名空间中找不到这些区域设置值的匹配，mc 会在其父命名空间中再依次搜索这些匹配，直到对全局命名空间完成搜索为止。这允许子命名空间“继承”父命名空间的消息。(Tcl 命名空间将在第 10 章中讨论。)

如果在所有应该搜索的命名空间的所有区域设置中都没有匹配的翻译字符串，mc 会执行过程::msgcat::mcunknown。mcunknown 的默认行为就是返回源字符串(这里就是 Welcome to Tcl!)，不过您可以重定义它来执行您希望的任何过程。

5.14.2 创建本地消息文件

如要使用 msgcat 工具包，需要为应用程序或发布包准备一系列消息文件，把它们都放在同一个文件夹中。消息文件的扩展名是.msg(例如，es.msg 表示西班牙语消息文件，en_gb.msg 表示英国英语消息文件)。这些文件名必须小写。唯一的例外是为根设置""提供的消息文件，它的文件名必须是 ROOT.msg。

提示： 定义 ROOT.msg 消息文件，会在无法找到指定的区域消息文件时提供翻译。如果在调用 mc 时使用类型 file_notfound 这样的符号式源字符串，ROOT.msg 的功能就相当有用。

每个消息文件都包括一系列对::msgcat::mcset 和/或::msgcat::mcmset 的调用，为该语言设置翻译字符串。mcset 命令的格式如下：

```
::msgcat::mcset locale src-string ?translation-string?
```

mcset 命令为给定的 *src-string* 设定一个指定区域的翻译。如果没有出现参数 *translation-string*，*src-string* 的值就直接作为指定的翻译字符串。

因此，如果美国英语是应用程序的“源语言”，一个 en_gb.msg 文件可能会包含以下命令：

```
::msgcat::mcset en_GB "Welcome to Tcl!"
::msgcat::mcset en_GB "Select a color:" "Select a colour:"
```

注意第一行没有提供翻译字符串，因此为 en_GB“翻译”的结果和源字符串是一样的，即“Welcome to Tcl!”。如果在消息文件中删去这一条目，那么调用 mc 为 en_GB 区域翻译字符串 Welcome to Tcl!时，就会按 5.14.1 节所讲述的方式搜索“最佳匹配”，如果搜索不到，就可能引起对 mcunknown 的调用。尽管 mcunknown 的默认行为会提供期待的结果(返回了 Welcome to Tcl!)，但在自定义了 mcunknown 的行为后就可能出现问题。因此，为安全起见，应该总是为应用程序中所用到的所有源字符串建立映射，即使特定的区域并不需要对该字符串进行“翻译”。

相应的西班牙语消息文件 es.msg，可能包含以下命令：

```
::msgcat::mcset es "Welcome to Tcl!" "¡Bienvenido a Tcl!"
::msgcat::mcset es "Select a color:" "Elige un color:"
```

除了使用多条 mcset 命令，还可以使用 mcmset 命令。

```
::msgcat::mcset locale translation-dict
```

translation-dict 是一个 Tcl 字典结构，它是由空格隔开的关键字和关联值的序列。(有关 Tcl 字典的更多信息参见第 7 章。)字典中的关键字就是源字符串，关联值就是相应区域的翻译字符串。包括空格的关键字和关联值都应该用双引号括起来。mcmset 的优势在于其运行速度远快于调用多条 mcset。因此，上面那个西班牙语消息文件也可以如下编写：

```
::msgcat::mcmset es {
    "Welcome to Tcl!" "¡Bienvenido a Tcl!"
    "Select a color:" "Elige un color:"
}
```

5.14.3 在源字符串和翻译字符串中使用转换符

有时候您可能想在本地化的字符串中插入一个或多个参数。msgcat 允许您在源字符串和翻译字符串中包含转换符，其解析方式与 5.8 节的 format 命令的解析方式完全相同。然后您可以把转换符的值作为额外的变量传给 mc 命令。

作为示例，考虑一个消息文件 fr.msg 中的以下条目：

```
::msgcat::mcset fr\
    "Directory contains %d files" "Il y a %d fichiers"
```

在应用程序中可以如下使用翻译：

```
puts [::msgcat::mc "Directory contains %d files" $num]
```

提示： *在消息文件中位置说明符特别有用，因为有些值的顺序在不同的语言中可能不同。*

5.14.4 在命名空间中使用消息目录

正如 5.14.1 节所述，当运行 mc 时，它进行“最佳匹配”搜索，返回翻译字符串，例如，在 en_GB 中搜索匹配的字符串，找不到时再在 en 中搜索匹配的字符串。应该注意这个搜索过程也与命名空间有关。如果在当前命名空间中没有找到匹配的翻译，mc 会在它的父命名空间中进行同样的一轮搜索，直到对全局命名空间完成搜索为止。这允许子命名空间“继承”父命名空间的消息。它还允许您为库或模块提供特有的翻译集合，如果它被包含在特定命名空间中。(第 10 章将详细讨论 Tcl 的命名空间。)

您用 mcset 和 mcmset 注册的消息文件中的翻译，会与全局命名空间关联，除非您明确地以 namespace eval 脚本来运行这些命令。例如，如果在开发一个使用 Mylib 命名空间的库，可以在 es.msg 文件中用以下代码把西班牙语翻译与该命名空间关联。

```
namespace eval Mylib {
    ::msgcat::mcmset es {
        "Welcome to Tcl!" "¡Bienvenido a Tcl!"
        "Select a color:" "Elige un color:"
    }
}
```

相应的英国英语消息文件 en_gb.msg，可以使用如下代码：

```
namespace eval Mylib {
    ::msgcat::mcmset en_GB {
        "Welcome to Tcl!" "Welcome to Tcl!"
        "Select a color: " "Select a colour:"
    }
}
```

5.15 二进制字符串

Tcl 最初设计为主要处理文本数据。当时认为对二进制数据结构的操作可以由用户编写自己的 C 代码实现，这些功能就没有包含在 Tcl 基本功能中。然而，有关二进制数据的操作是如此常用，Tcl 还是增加了 binary 命令来管理二进制数据。

binary format 命令在一个 Tcl 变量中创建二进制字符串。大多数情况下，您会把二进制字符串写入通道，即写入文件或网络套接字。使用 binary format 命令，您应该在 format 后面指定格式，然后给出要进行格式处理的字符串。

```
binary format formatString ?arg arg ...?
```

这里的 *formatString* 由一系列字段指示符组成，可以包含分隔符(任意长度的空白)。每一个字段指示符包含一个字符，描述将要格式化的数据的类型，还可能给出需要格式化的条目数。如果不给定数目，默认为 1。对于大多数类型，使用*表示关联参数中的各项都应该格式化。

表 5.7 列出了 binary format 命令支持的格式类型。大多数类型适合于存放数值信息，例如：

```
binary format c3 {1 2 120}
```

这条命令返回一个二进制字符串，各字节为\x01\x02\x80。格式字符串 c3 的意思是格式化三个 8 位有符号整数。如果提供更多的格式说明，那么后面的参数应该与指定格式一致，例如：

```
binary format I2a {1 -32 412534} E
```

第一个格式指示符 I2，只针对后面参数{1 -32 412534}的开头两个数，因为该指示符的计数是 2，会把这两个数作为 32 位有符号整数，按大端序(最高有效字节在前)存放。然后，第二个格式指示符 a，从后面第二个参数中获取一个字符，存储为 8 位 Latin-1 字符。

表 5.7 binary format 和 binary scan 命令支持的格式类型

格 式	用 法
a	指定计数个 ISO 8859-1(Latin-1)二进制字符串。仅使用 Unicode 字符的低位。如果 binary format 需要扩展的话补零
A	与 a 相同，不过用 binary format 扩展时补空格。使用 binary scan 则会裁剪掉尾部的空格和空白
b	二进制数字，字节中各位从低到高
B	和 b 相同，不过字节中各位从高到低
c	8 位有符号整型值
d	双精度浮点值，字节序采用运行脚本的计算机的原生序(与 q 或 Q 相同)
f	单精度浮点值，字节序采用运行脚本的计算机的原生序(与 r 或 R 相同)
h	十六进制数字的二进制字符串，字节中各位从低到高。此格式几乎从未被使用
H	十六进制数字的二进制字符串，字节中各位从高到低
i	32 位有符号整型数，小端序
I	与 i 相同，不过是大端序
m	64 位有符号整型数，字节序采用运行脚本的计算机的原生序(与 w 或 W 相同)
n	32 位有符号整型数，字节序采用运行脚本的计算机的原生序(与 i 或 I 相同)
q	双精度浮点值，小端序
Q	与 q 相同，不过是大端序
r	单精度浮点值，小端序

续表

格 式	用 法
R	与 r 相同，不过是大端序
s	16 位有符号整型数，小端序
S	与 s 相同，不过是大端序
t	16 位有符号整型数，字节序采用运行脚本的计算机的原生序(与 s 或 S 相同)
w	64 位有符号整型数，小端序
W	与 w 相同，不过是大端序
x	使用 binary format 在二进制字符串中存储 *count* 个 null(0x00)字节；此类型并不需要使用参数值。使用 binary scan 将二进制字符串的指针正向移动 *count* 个字节；*count* 为*表示移动到位置 0
X	将二进制字符串的指针反向移动 *count* 个字节，*count* 为*表示移动到位置 0
@	将二进制字符串的指针移动到指定位置的字节，从位置 0 开始计算

binary scan 命令从二进制字符串中取得数据。

```
binary scan binaryString formatString ?varName varName ...?
```

这里的 *formatString* 包含一系列的字段指示符，可以用任意数量的空白隔开。binary scan 的字段指示符的格式与 binary format 相同，只不过在类型字符后多了一个可选的修饰符 u。对于整型字符，u 表示这是一个无符号数，其他情况下这个 u 会被忽略。对于每一个要从二进制字符串中取得值的字段指示符，应该为它提供保存结果的变量名；否则命令执行时会产生错误。如果提供了比需要的参数更多的变量名，多余的变量不会受到 binary scan 的影响。如果字段指示符包含计数值，就会从二进制字符串中读入计数个该类型的值，在相应变量中存储为一个列表(面向字符的类型除外)。如果二进制字符串中没有足够的数据与所有的字段指示符对应，多余的变量也不会被处理。binary scan 的返回值是该命令设置的变量的个数。

下面是一些从二进制字符串中取得数据的示例。

```
    set data "\x40\x41\x42\x43\x44\x45\x46\x47\x48"
    binary scan $data a4c4 str chars
⇒   2
    puts $str
⇒   @ABC
    puts $chars
⇒   68 69 70 71
    binary scan $data a*c4 str2 chars2
⇒   1
    puts $str2
⇒   @ABCDEFGH
    puts $chars2
∅   can't read "chars2": no such variable
    binary scan $data IS3 int shorts
⇒   1
    puts $int
⇒   1078018627
    puts $shorts
∅   can't read "shorts": no such variable
    binary scan $data IS2 int shorts
```

```
⇒ 2
  puts $int
⇒ 1078018627
  puts $shorts
⇒ 17477 17991
  binary scan $data B8B5 bits1 bits2
⇒ 2
  puts $bits1
⇒ 01000000
  puts $bits2
⇒ 01000
  binary scan $data H * hex1
  puts $hex1
⇒ 404142434445464748
  binary scan $data h* hex2
  puts $hex2
⇒ 041424344454647484
```

第6章 列 表

Tcl 使用列表来处理各种集合，例如一个组内的所有用户、一个文件夹中的所有文件以及一个组件的所有选项。列表允许您把任意数量的值集合在一起，把集合作为一个实体传递，从集合中取得各成员的值。列表是元素的有序集合，各个元素可以有任何的字符串值，例如一个字、一个人名、一个窗口名或是一个 Tcl 命令的单词。列表表现为特定结构的字符串，这意味着可以把列表存放在变量中，将它们传给命令，以及将它们嵌套进别的列表。

6.1 本章出现的命令

本章讨论列表的结构，展示了列表操作的十多个命令。这些命令包括列表的创建、取得元素、搜索特定元素。后面各章还会描述其他以列表作为参数或返回列表的一些命令。

- `concat ?list list ...?`
  将多个列表合并为一个列表(各个 *list* 作为结果列表的元素)并返回新的列表。
- `join list ?joinString?`
  把 *joinString* 作为分隔符，把列表元素串接起来。*joinString* 默认为空格。
- `lappend varname value ?value...?`
  把 *value* 作为列表元素添加到 *varName* 变量中，然后返回变量的新值。如果这个变量不存在，就创建它。
- `lassign list varName ?varName ...?`
  把 list 中的元素顺次赋给名为 *varName* 的变量。如果变量名比元素数量多，多余的变量设为空字符串。如果元素数量比变量名多，就返回由未赋给变量的元素组成的列表。
- `lindex list ?index ...?`
  使用 *list* 的第 *lindex* 个元素(从 0 开始数)。使用多个索引值时，可以是分别独立的变量，也可以是一个列表，每个索引依次对前一个索引的结果进行操作，从而访问嵌套的列表元素。
- `linsert list index value ?value ...?`
  将所有的 *value* 参数作为列表元素插入 *list* 中第 *index* 个元素(从 0 开始数)之前，返回新的列表。

- `list ?value value ...?`
  返回一个列表，其元素就是 *value* 参数。
- `llength list`
  返回 *list* 中的元素个数。
- `lrange list first last`
  返回一个列表，由 *list* 列表中 *first* 到 *last* 的元素组成。
- `lrepeat number value ?value ...?`
  把 *value* 参数作为元素，重复 *number* 次得到列表，返回这个列表。
- `lreplace list first last ?value value ...?`
  把 *list* 中从 *first* 到 *last* 的元素替换为零个或多个元素，这些元素由 *value* 参数给出，每一个 *value* 参数对应一个元素，返回处理得到的列表。
- `lsearch ?option ...? list pattern`
  在 *list* 中搜索与 *pattern* 匹配的一个或多个元素。*option* 选项控制模式匹配方式(-exact、-glob、-regexp)，是返回元素值(-inline)还是索引，是搜索所有的匹配(-all)还是只搜索最先出现的匹配等。默认进行通配符匹配，返回第一处匹配的索引，没有匹配元素时返回-1。
- `lset varName ?index ...? newValue`
  将 *varName* 列表中由 *index* 索引指向的元素值设置为 *newValue*。返回存储在 *varName* 中的新列表。
- `lsort ?option ...? list`
  对 list 中的元素排序，返回排序后的列表。选项控制了比较函数和排序方式(默认：-ascii -increasing)。
- `split string ?splitChars?`
  把 *string* 在出现 *splitChars* 处分开，将 *splitChars* 之间的部分作为元素，组成一个新的列表，返回这个列表。

6.2 基本列表结构与 lindex 和 llength 命令

最简单的列表就是包含由任意个空格、制表符、换行符分隔开的任意多个元素的字符串。例如，以下字符串：

```
John Anne Mary Jim
```

就是一个有 4 个元素的列表。一个列表中可以有任意多个元素，每一个元素都可以是任意的字符串。在这种简单形式中，元素不能包含空白，也有另外的列表语法允许在元素中加入空白，后面有专门的讨论。

lindex 命令从一个列表中取得元素。

```
lindex {John Anne Mary Jim} 1
⇒ Anne
```

lindex 至少要获取两个参数，即一个列表和一个索引值，并返回从列表中取得的元素。对所有的列表命令，索引 0 都对应第一个元素，索引 1 都对应第二个元素，以此类推；索引 end 对应列表中的最后一个元素，end-1 对应倒数第二个元素，以此类推。在 Tcl 8.5 中，还可以把两个整数相加减的表达式作为索引。在使用 end±整数或整数±整数形式的索引时，索引参数中不能有空白符，即使这个索引参数被括起来也不能。如果索引指向的位置超出了列表，lindex 会返回空字符串。

llength 命令返回列表中的元素的个数。

```
    llength {a b c d}
⇒   4
    llength a
⇒   1
    llength {}
⇒   0
```

从上面这些示例可以看出，像 a 这样的简单字符串就是只有一个元素的列表，空白字符串就是有零个元素的列表。

在 Tcl 命令中输入文字列表时，列表通常用大括号括起来，就像前面几个示例那样。这对大括号不是列表的一部分；命令行需要它们是要使整个列表作为一个单词传递。存放在变量中的列表，或是打印显示列表时，都没有命令行中那个大括号。

```
    set x {John Anne Mary Jim}
⇒   John Anne Mary Jim
```

列表命令对列表元素中的大括号和反斜线的处理，与 Tcl 命令解析器对命令中的单词的处理是一样的。这就是说可以把一个含有空白符的元素括在括号里，还可以使用反斜线替换来给出特殊字符，如括号等。

```
    set test1 {a b\ c d}
    llength $test1
⇒   3
    lindex $test1 1
⇒   b c
    set test2 {a b\nc d}
    llength $test2
⇒   3
    lindex $test2 1
⇒   b
    c
    set test3 {a \} b \{ c}
    llength $test3
⇒   5
    lindex $test3 1
⇒   }
```

提示：在创建其元素包含特殊字符的列表时，使用后文讨论的 list 命令，是确保特殊字符被正确处理的最安全方法。

大括号常用于在列表中嵌套列表，示例如下：

```
    lindex {a b {c d e} f} 2
⇒   c d e
```

这里列表中索引值为 2 的元素本身又是一个列表，它由三个元素组成。列表可嵌套的层数没有限制。

在操作嵌套的列表时，lindex 命令允许您指定一个或多个索引值，可以是分别独立的参数，也可以是一个列表，通过多个索引值可以从子列表中取得元素，例如：

```
    set elements {{a b} {c {d e f}} g}
    lindex $elements 1 1 2
⇒   f
    lindex $elements {0 0}
⇒   a
```

第一个示例与下面这条命令相同：

```
    lindex [lindex [lindex $elements 1] 1] 2
⇒   f
```

而它很明显更短，也更不容易犯错。

6.3 创建列表：list、concat 和 lrepeat

Tcl 提供了三个把字符串联合为列表的命令：list、concat 和 lrepeat。每一个命令都接受任意多个参数，返回结果为一个列表。它们的不同之处在于把参数组成列表的方式。

list 命令把它的参数加入列表，即每个参数作为列表的一个独立元素。

```
    list {a b c } {d e} f {g h i}
⇒   {a b c} {d e} f {g h i}
```

这里，结果列表一共只有 4 个元素。list 命令总是会产生一个适当的列表结构，无论它的参数的结构是什么样的(它会自动添加必要的括号或反斜线)，lindex 命令总是可以从由 list 命令生成的列表中取得原始元素。传给 list 的参数本身不必是形式完整的列表。

提示： 如果您不知道元素的值是什么，使用 list 命令创建列表是安全的方法(例如，在您提示用户进行输入，或是从文件中读取一个值时)。

命令 concat 接受任意多个列表作为参数，把参数列表中的所有元素串接为一个大的列表。如果某个输入列表的某个元素是嵌套的列表，这个元素会保持为嵌套的列表。

```
    concat {a b c} {d e} f {g h i}
⇒   a b c d e f g h i
    concat {a b} {c {d e f }}
⇒   a b c {d e f}
```

concat 需要它的参数有适当的列表结构；如果某个参数不是形式完整的列表，那么这个命令给出的结果也可能不具备完整的列表形式。事实上，concat 做的工作就是把它的参数字符串开头和结尾的空白裁掉，然后把各个元素以空格隔开，串接起来形成一个大的列表。concat 完成的任务可以用双引号来完成。

```
    set x {a b c}
    set y {d e}
    set z [concat $x $y]
⇒   a b c d e
```

```
    set z "$x $y"
⇒ a b c d e
```

lrepeat 命令重复一个元素集合来创建列表，集合中各个元素作为单独参数给出，而第一个参数是指定的重复次数，例如：

```
    lrepeat 3 a
⇒ a a a
    lrepeat 4 a b c
⇒ a b c a b c a b c a b c
    lrepeat 3 {a b} c
⇒ {a b} c {a b} c {a b} c
```

6.4 修改列表：lrange、linsert、lreplace、lset 和 lappend

lrange 命令返回列表中某范围内的元素。它获取的参数包括一个列表和两个索引值，返回的是列表中这两个索引值对应的元素之间的所有元素组成的新列表(包括这两个索引值所对应的元素)。

```
    set x {a b {c d} e}
⇒ a b {c d} e
    lrange $x 1 3
⇒ b {c d} e
    lrange $x 0 1
⇒ a b
```

linsert 命令把一个或多个元素插入已经存在的列表，从而形成新的列表。

```
    set x {a b {c d} e}
⇒ a b {c d} e
    linsert $x 2 X Y Z
⇒ a b X Y Z {c d} e
    linsert $x 0 {X Y} Z
⇒ {X Y} Z a b{c d} e
```

linsert 需要获取三个或更多个参数。第一个参数是一个列表，第二个参数是列表中的第一元素所对应的索引值，第三个以及其他参数是将要插入这个列表的新元素。linsert 的返回值是一个列表，由原列表插入新的元素得到，这些元素插入到指定索引值对应的原列表中的元素之前。如果索引值是 0，那么新的元素就插入到列表的开头；如果是 1，新的元素就插入到原列表的第一个元素之后，以此类推。如果索引值大于或等于原列表的元素个数，那么新元素会插入列表的末尾。

lreplace 命令从列表中删除元素，且可选地在它们的位置添加新的元素。它需要获取三个或更多个参数。第一个参数是一个列表，第二个和第三个参数给出了要删除的那部分元素的开头元素和结尾元素分别对应的索引值。如果只给定了三个参数，那么结果就是从原列表中删除了指定部分元素得到的新列表。

```
    lreplace {a b {c d} e} 3 3
⇒ a b {c d}
```

如果 lreplace 指定了更多的参数，如下面这个例子，这些参数会插入到被删除的元素的位置。

```
    lreplace {a b {c d} e} 1 2 {W X} Y Z
⇒ a {W X} Y Z e
```

提示： Tcl 并没有单独的明确命令用于从列表中删除元素，如果 lreplace 命令不指定用于替换的元素，就已经提供了这个功能。

有一个常见的操作是在列表的某个元素的值变化后更新存放在变量中的列表。过去 lreplace 常用于这个目的，例如：

```
    set person {{Jane Doe} 30 female}
    set person [lreplace $person 1 1 31]
⇒ {Jane Doe} 31 female
```

因为 lreplace 并不是直接修改变量的值，您需要进行一次命令替换来执行它，然后把返回的结果赋给存有列表的变量作为它的新值。这不仅写起来麻烦，而且缺乏效率，因为 lreplace 在创建新列表的时候必须复制原列表中的元素。如果经常需要更新一个大列表中的元素，效率会受明显影响。

当列表存放在一个变量中时，lset 命令是一种快速简练地修改元素值的方法。它获取变量名，列表中已经存在的一个元素对应的索引——或者一系列索引从而指向某个子列表的元素——以及赋给这个元素的新值。lset 返回变量的新值。

```
    lset person 1 32
⇒ {Jane Doe} 32 female
    lset person {0 1} Johnson
⇒ {Jane Johnson} 32 female
    lset person 0 0 Janice
⇒ {Janice Johnson} 32 female
```

提示： 您不能用 lset 命令创建新的列表。它只能修改已经存在的列表。如果给定的索引指向不存在的元素，lset 会返回一个错误。

lappend 命令提供了一种高效率的方法，把新的元素添加到存放在一个变量中的列表里。它要获取一个存放列表的变量名作为参数，以及任意多个其他参数，这些参数作为元素添加到第一个参数指明的那个列表中，lappend 会返回该列表变量的新值。

```
    set x {a b {c d} e}
⇒ a b {c d} e
    lappend x XX {YY ZZ}
⇒ a b {c d} e XX {YY ZZ}
    puts $x
⇒ a b {c d} e XX {YY ZZ}
```

lappend 与 append 相似，只不过它会强制生成适当的列表结构。而 append 并不这样做。例如，命令：

```
lappend x $a $b $c
```

可以改写为：

```
set x [concat $x [list $a $b $c]]
```

然而，与 append 相比，lappend 的运行效率更高。对于大的列表，可能会有显著的效率差异。

提示：lappend 和 lset 与其他像 lreplace 这样的列表命令不同，它们所接受的是存放列表的变量，而不是直接接受列表，因此您需要用存放列表的变量名为它们指明要处理的列表。

6.5 从列表中取得元素：lassign

使用 lassign 命令，可以方便地把列表中的值分发到一个或多个变量中。它的第一个参数是一个列表，后面的参数是变量名。lassign 把列表中的元素依次分发到这些变量中。如果变量名比列表中的元素个数多，多余的变量会被设置为空字符串。如果列表中的元素比变量个数多，则会返回一个由未分发的元素组成的列表。

```
    lassign {a b c} x y z  ; # Produces an empty return value
    puts "<$x>,<$y>,<$z>"
⇒   <a>,<b>,<c>
    lassign{d e} x y z     ; # Produces an empty return value
    puts "<$x>,<$y>,<$z>"
⇒   <d>,<e>,<>
    lassign {f g h i} x y
⇒   h i
    puts "<$x>,<$y>"
⇒   <f>,<g>
```

当然，类型的结果也可以通过一系列 lindex 命令获得，不过就不那么简洁了。

```
set x [lindex $vals 0]
set y [lindex $vals 1]
set z [lindex $vals 2]
```

lassign 的行为使得对一些语言中的“上档”(shift)命令的模仿变得容易。

```
set argv [lassign $argv nextArg]
```

Tcl 8.5 版以前，大家常利用 foreach 命令的副作用将列表中的元素分发给独立变量。例如，要将变量 coords 的前三个元素分发给三个独立变量。

```
foreach {x y z} $coords {break}
```

这个示例中的 break 命令作为一个“失效安全”保障，以处理 coords 变量中包含了多于三个元素的情况。

6.6 搜索列表：lsearch

lsearch 命令在列表中查找指定的元素。它获取两个参数，第一个是一个列表，第二个是一个模式。

```
    set x {John Anne Mary Jim}
    lsearch $x Mary
⇒   2
    lsearch $x Phil
⇒   -1
```

lsearch 返回列表中第一个与指定模式匹配的元素的索引，如果没有匹配的元素则返回-1。

可以通过在列表参数前指定下列标志之一来设定模式匹配的方式：-exact、-glob和-regexp。

```
    lsearch -glob $x A*
⇒ 1
```

-glob 指定匹配按照 string match 命令的规则进行，该规则详见 5.10 节。-regexp 指定匹配按照正则表达式规则进行，相应规则详见 5.11 节，-exact 则要求进行严格匹配。如果没有指定匹配方式，那么默认设置为-glob。您还可以用-not 选项对匹配结果取反。

默认情况下，lsearch 只查找最先出现的匹配的元素。不过，您可以使用-all 命令，要求将所有匹配的元素组成一个列表返回。

```
    set states {California Hawaii Iowa Maine Vermont}
    lsearch -all $states *a
⇒ 0 2
```

-inline 选项指定返回元素，而非元素的索引。在使用模式匹配时这一选项很有实用性，它可以让您不必再多做一步，用 lindex 来取得元素的值。

```
    lsearch -all -inline $states *ai*
⇒ Hawaii Maine
```

提示： 记得如果您要探测一个确切的字符串是否是列表中的一个元素，可以在表达式中分别使用 in 和 ni 操作符，更多信息参见 4.7 节。

6.7 排序列表：lsort

lsort 命令获取一个列表作为参数，返回有相同元素的列表，但新列表中的元素已经按字典顺序排序了。

```
    lsort {John Anne Mary Jim}
⇒ Anne Jim John Mary
```

您可以在列表前设置很多选项来控制排序。例如，-decreasing 指定结果应该把“最大”的元素放在最前面；-integer 和-real 指定列表中的元素应该被视为整数或是实数，然后按照其值的大小进行排序；-dictionary 指定不区分大小写的排序，并且元素中嵌入的数字都作为非负整数处理；-unique 返回的结果中，原列表里重复出现的元素只会出现一次。

```
    lsort -decreasing {John Anne Mary Jim}
⇒ Mary John Jim Anne
    lsort {10 1 2}
⇒ 1 10 2
    lsort -integer {10 1 2}
⇒ 1 2 10
    lsort {Peach banana Apple pear}
⇒ Apple Peach banana pear
    lsort -dictionary {Peach banana Apple pear}
⇒ Apple banana Peach pear
    lsort -dictionary {n11.gif n1.gif n10.gif n9.gif}
```

```
⇒ n1.gif n9.gif n10.gif n11.gif
  lsort -unique {c a b q a z q}
⇒ a b c q z
```

如果有一个嵌套的列表结构，-index 选项允许您指定子列表中元素的索引，根据指定的元素对子列表进行排序。

```
  lsort -integer -index 1 {{First 24} {Second 18} {Third 30}}
⇒ {Second 18} {First 24} {Third 30}
```

另外，如果列表包含了不能进行字词排序或数学排序的数据，可以使用-command 选项定义自己的排序函数(详见参考文档)。

6.8 在字符串和列表之间转化：split 与 join

split 命令将字符串分成几个部分，然后可以对各个部分独立地进行处理。它会创建一个列表，列表中的元素就是字符串的各个部分，您可以使用列表命令对它进行处理。例如，有一个变量，其内容是由逗号分开的值，您希望把它转化成一个列表，变量中的每一个值成为列表的一个元素。

```
  set x "Anita Sanchez,35,VP Marketing"
  set y 39,72,,-17,
  split $x ,
⇒ {Anita Sanchez} 35 {VP Marketing}
  split $y ,
⇒ 39 72 {} -17 {}
```

split 的第一个参数是待分割的字符串，第二个参数是一个或多个分割字符。split 会找到字符串中所有的分割字符。然后它会创建一个列表，其元素就是分割字符之间的子字符串。字符串的边界也被作为分割字符对待。如果子符串中连续出现分割字符，或者分割字符出现在字符串的开头或结尾，就像上面的第二个示例那样，结果中就会产生空元素。分割字符本身会被抛弃。

提示： 在实际应用中，分隔字符值(CSV)需要谨慎对待，因为用作分隔符的字符有可能出现为值的一部分。Tcllib，即标准 Tcl 库，包含了一个 csv 工具包，用于处理 CSV 数据。有关 Tcllib 的更多信息参见附录 B。

也可以设定由多个字符组成分隔符，例如：

```
  split xbaybz ab
⇒ x {} y z
```

如果指定为分隔字符的是空字符串，那么，会把字符串的每一个字符都分开，作为新列表中的独立元素。

```
  split {a b c} {}
⇒ a {} b {} c
```

join 命令大体上是 split 命令的逆操作。它把列表元素串接成一个字符串，元素之间用指定的分隔符号隔开。

```
    join {{} usr include sys types.h} /
⇒   /usr/include/sys/types.h
    set x {24 112 5}
    expr [join $x +]
⇒   141
```

join 获取两个参数：一个列表和一个分隔字符串。它从列表中取出所有的元素，把它们串接成一个字符串，各元素转换成的子字符之间由分隔字符串隔开。分隔字符串可以包括任意多个字符，包括 0。这里的第一个示例中，以/作为分隔符，连接产生了一个 Unix 类型的路径名称(第 11 章介绍的 file join 命令是创建这种路径的更好的方法，而且它是与平台无关的)。第二个示例，以+作为分隔符，连接产生了一个 Tcl 表达式。

6.9 用列表创建命令

Tcl 的列表和命令之间有很重要的关系。一个形式完整的 Tcl 命令的结构与列表是相同的。处理列表中的元素的方式，也和处理命令中的单词的方式是相同的。换句话说，Tcl 解析器不管遇到列表还是命令，都不进行替换操作。它只是取得列表的元素，而各个元素也就成为命令的各个单词。这一属性十分重要，因为它可以生成一定会按特定形式解析的命令，即使命令中的某些单词包含了空格、$这样的有特殊含义的字符。

例如，要在 Tk 中创建一个按钮组件，当用户单击这个组件时就把一个变量设为特定值。可以用下面这条命令创建这样一个组件。

```
button .b -text Reset -command {set x 0}
```

Tcl 脚本 set x 0 会在用户单击按钮时运行。现在假设该变量要设为的值不是一个常数，而是在 button 命令执行之前计算出来的，必须从一个名为 initValue 的变量中读取呢？并且，假设 initValue 可能包含着任意字符串。您可能会把该命令重写为：

```
button .b -text Reset -command {set x $initValue}
```

脚本 set x $initValue 会在用户单击按钮时运行。然而，脚本使用的是用户单击按钮时的全局变量 initValue 的值，这个值与按钮创建时未必是同一个值。例如，同样的变量可能被用于创建多个按钮，每一个按钮都有不同的复位值。或者，如果创建按钮的代码被包含在一个过程中，initValue 可能是一个局部变量，在用户单击按钮时甚至可能根本就不存在，这些情况都可能导致错误。

要解决这个问题，必须生成一个 Tcl 命令，控制 initValue 变量的值，而不是它的名字，将它作为 button 命令的-command 选项。遗憾的是，下面这样的简单做法大多数情况下并不能奏效。

```
button .b -text Reset -command "set x $initValue"
```

如果 initValue 的值是 47 之类的简单内容，那这句脚本可以工作得很好。它会得到命令 set x 47，正好是我们希望的结果。然而，如果 initValue 的值是 New York 呢？那样的话，它会得到命令 set x New York，共有 4 个单词；set 就会因为参数过多而发生错误。还有更糟糕的情况，如果 initValue 的值是$或者[呢？在处理的时候这些字符可能导致不该发生的替换操作。

保证 initValue 的值无论是什么都可以正常工作的解决方案只有一个，使用列表命令来生成这条命令，示例如下：

```
button .b -text Reset -command [list set x $initValue]
```

list 命令的结果是一条 Tcl 命令，这条命令的第一个单词是 set，第二个单词是 x，第三个单词是创建按钮时(而不是单击按钮时)initValue 的值。例如，假设 initValue 的值是 New York，那么 list 生成的命令如下：

```
set x {New York}
```

这条命令能被正确解析执行。调用 button 命令时 initValue 中的值，无论它是什么，都会在单击按钮时赋给 x，无论 initValue 的值是什么，这条命令都能正确处理。其中出现的 Tcl 特殊字符也能够被 list 正确处理。

```
    set initValue {Earnings: $1410.13}
    list set x $initValue
⇒   set x {Earnings: $1410.13}
    set initValue "{ \\"
    list set x $initValue
⇒   set x \{\ \\
```

第7章 字 典

您有了像列表这样的集合，如果想给其中的每一个元素指定一个独有的名字，然后通过元素的名字访问它们，最好的方法就是组成字典。和列表类似，字典也可以包含任意多个值，如数字、人名、窗口名、通道以及命令名。字典把这些值整理成条目，您可以把它作为一个值来传递，或者嵌套到别的字典或列表中，然后还可以通过它们的关键名查找它们并取得各个成分的值。

字典提供一个有序的集合；字典会维护关键字的顺序，并且在需要时按顺序遍历它们。字典表现为有特定结构的字符串，看上去就像有偶数个元素的列表，这意味着可以把它们存放到变量中(包括数组元素中)，把它们传给命令，把它们嵌套在它们自己或是列表中。当然也可以把列表放到字典中。

字典和数组有一些根本性的不同。数组是变量的无序集合，而不是值的集合，而且不能嵌套。这就是说只有数组的元素可以有追踪设置，而只有字典能够可靠地按一定顺序遍历，或作为一个值传给其他命令(特别地，非全局数组需要使用 upvar 或明确进行打包和解包)。

提示： Tcl 8.5 引入了字典数据类型，作为在 array 命令的 get 和 set 子命令中，在 Tk 组件选项等当中，对字符串的管理的成功经验的正式定形。第三方的扩展包 dict 可以为 Tcl 8.4 提供大部分字典功能，更早期的 Tcl 版本也可以通过适当脚本支持字典，但从 8.5 版开始，Tcl 才支持高效率的有序字典。

7.1 本章出现的命令

本章讲述字典的结构，展示用于字典操作的 dict 命令。dict 的子命令可以用于查找指定元素，插入、替换和移除元素，列出字典的名称，等等。后面几章会讨论一些使用字典作为参数，或返回字典作为结果的命令。

- `dict append` *varName key value ?value ...?*
  把给定的字符串 *value* 添加到字典 *varName* 的 *key* 对应的值当中。
- `dict create` *key value ?key value ...?*
  用给定的 *key* 和 *value* 创建字典。如果在参数列表中某个关键字出现了多次，那么它所关联到的值是这个关键字最后一次出现时对应的值。

- dict exists *dictionary key ?key ...?*
  检测 *dictionary* 中是否存在 *key*。可以给定多个关键字以检测嵌套的字典中是否存在那些关键字。
- dict filter *dictionary filterType...*
  对给出的 *dictionary* 进行筛选，筛选结果是一个新的字典。筛选可以通过对关键字或值的匹配(使用 string match 规定)进行，也可以由脚本进行。
- dict for {*keyVar valueVar*} *dictionary body*
  遍历 *dictionary* 中的关键字和值，将给定变量依次设为相应的关键字和关联值，然后分别运行 *body* 参数。
- dict get *dictionary key ?key ...?*
  返回该字典中与给定的 *key* 对应的值。可以给定多个关键字，以取得嵌套的字典中相应的值。
- dict incr *varName key ?increment?*
  对 *varName* 字典中与给定的 *key* 对应的值进行增加操作。如果增加量没有给定，默认为 1。如果字典中开始时没有该关键字，那么创建该项，设关联值为 0，再进加增加操作。
- dict keys *dictionary ?pattern?*
  返回 *dictionary* 中的全部关键字的列表。如果设定了 *pattern* 参数，则只返回按照 string match 规则与 *pattern* 匹配的那些关键字。
- dict lappend *varName key value ?value ...?*
  把给定的列表添加到字典 *varName* 的 *key* 对应的值当中。
- dict merge *?dictionary dictionary ...?*
  联合所有给出的字典形成新的字典并返回。有相同的关键字时，后出现的关键字与关联值会覆盖先出现的关联值。
- dict remove *dictionary ?key ...?*
  把提供的 *dictionary* 里在命令中列出的 *key* 项删去，返回一个新的字典。如果列出的 *key* 在原 *dictionary* 中不存在，也不会引起错误。
- dict replace *dictionary ?key value ...?*
  在提供的 *dictionary* 里加入命令中给出的 *key* 关键字关联值对(如果原字典中有同名关键字，则覆盖原字典中该关键字项)，返回新的字典。
- dict set *varName key ?key ...? value*
  把变量 *varName* 中的字典里，由命令中给出的 *key* 指定的项关联到命令中给定的 *value*，然后把改动后的字典写入 *varName*。可以使用多个关键字来更新嵌套的字典中某项的值。
- dict size *dictionary*
  返回给定字典的大小(关键字的数量)。
- dict unset *varName key ?key...?*
  将变量 *varName* 中的字典里与给定的 key 相应的映射删除。可以使用多个关键字

来移除嵌套的字典中某项的值。如果没有发现指定的映射，也不会产生错误。

- `dict update varName key localVar ?key localVar ...? body`
  对 *varName* 字典里所有在命令中列出的 *key*(如果给出了多个关键字，则是指子字典中的项)，将其值绑定到 *localVar*，然后运行 *body*。在 *body* 运行结束后，把变量内容写回字典。
- `dict values dictionary ?pattern?`
  返回 *dictionary* 中的值的列表。如果给定 *pattern*，仅有与它匹配(按照 string match 规则)的值会加入返回列表。
- `dict with varName ?key ...? body`
  对 *varName* 字典里所有在命令中列出的 *key*，将其值绑定到 *varName*，然后运行 *body*。在 *body* 运行结束后，把变量内容写回字典。

7.2 基本字典结构与 dict get 命令

字典是类似于有偶数个元素的列表的一种结构，其中第一、三、五(等等)个元素(关键字)都互不相同。这些字符串元素用作集合的值的索引，这一点与列表中用位置作为索引不同。字典可以用于表示结构(名称集合是固定的)和映射(名称集合是任意的)。例如下面这个字符串：

```
firstname Joe surname Schmoe title Mr
```

就是有三个值(Joe、Schmoe 和 Mr)的字典，三个值的名称分别为关键字 firstname、surname 和 title，它把每一个关键字映射到它后面跟着的那个值。字典中可以有任意多个元素，但是每个值的关键字都必须是独一无二的。关键字和关联值都可以是任意的值。

dict get 命令从字典中取得一个元素。

```
set example {firstname Joe surname Schmoe title Mr}
dict get $example surname
⇒ Schmoe
```

dict get 获取两个参数，一个字典和一个要在字典中查找的关键字，然后返回这个关键字的关联值；如果该关键字没有与它关联的值，则会发生错误。在字典中查找一个值是很快速的，而且字典还可用作有序哈希表，在那里面进行查找的时间消耗几乎是一个常数。

和列表相似，在 Tcl 脚本中使用字典时，它常常被大括号括起来。对于比较复杂的字典，可以使用换行把各个关键词-关联值对分隔开。

```
set prefers{
    Joe          {the easy life}
    Jeremy       {fast cars}
    {Uncle Sam} {motherhood and apple pie}
}
dict get $prefers Joe
⇒ the easy life
```

这个字典有三个值(如果是列表的话，就是 6 个值)。您还可以把字典和列表随意互相嵌套；Tcl 语言对此也没有限制。下面这个示例将雇员号映射到与该雇员的详细信息结

构；这里的映射和详细信息结构都是字典。

```
set employees {
    0001 {
        firstname   Joe
        surname     Schmoe
        title       Mr
    }
    1234 {
        firstname   Ann
        initial     E
        surname     Huan
        title       Miss
    }
}
puts [dict get [dict get $employees 1234] firstname]
⇒ Ann
```

使用数组和结构关键字也可以实现上面这个示例的功能。但是使用数组的话，要么与每个雇员相关的记录得做成一个列表(在获取信息和更新时都要格外小心)，要么就没有代表雇员的整体数据的记录；当需要那样的记录时(例如，要把它传给一个过程)，就需要先把它从数组中提取出来，这会产生大量的重复操作。与此相对，嵌套的字典可以提供适应性更强也更有效率的解析方案。

的确，因为 dict get 命令可以接受多个关键字作为参数，要从嵌套的字典中获取一个值就变得更方便，可以像给出文件系统中的一条路径一样给出嵌套字典中的一条路径，例如：

```
puts [dict get $employees 1234 firstname]
⇒ Ann
```

7.6 节对嵌套字典的控制有深入的讨论(包括如何进行高效率地更新)。

作为对比，下面是使用数组和结构关键字的脚本，这种方法在 Tcl 8.5 以前的版本中是常用的方法。

```
set employees(0001,firstname)  Joe
set employees(0001,surname)    Schmoe
set employees(0001,title)      Mr
set employees(1234,firstname)  Ann
set employees(1234,initial)    E
set employees(1234,surname)    Huan
set employees(1234,title)      Miss
puts $employees(1234,firstname)
⇒ Ann
```

这种方法乍一看还比较简洁，但是当您需要有关 Ann 的全部记录时，脚本就不像字典那样简单、清楚、快速了。

```
# The dictionary version
set AnnsRecords [dict get $employees 1234]
⇒ firstname Ann initial E surname Huan title Miss
# The array version
set AnnsRecords [array get employees 1234,*]
⇒ 1234,title Miss 1234,initial E 1234,surname Huan 1234,firstname Ann
```

随着对记录处理过程的复杂度的增加，根据元素名称筛选取得信息变得越来越麻烦。

而且因为数组是无序的，生成输出需要对数组中的每一个变量都进行检查。

7.3 创建和更新字典

Tcl 提供了协助创建字典的命令：dict create。这个命令与 list 相似(创建列表的命令)，它获取任意多个关键字-关联值对，然后生成一个包含这些关键字-关联值对的字典。当一个关键字多次出现时，采用最后与它关联的值。在执行字典创建命令时，如果关键字或关联值还不确定，上述规则十分有用。

```
    dict create a b c {d e} {f g} h a "b repeated"
⇒ a {b repeated} c {d e} {f g} hc {d e} a {b repeated} {f g} h
```

正如您所看到的，字典中关键字的顺序是它们第一次出现的先后顺序，它们的关联值是该关键字最后一次被给出的关联值。使用 dict replace 基于旧字典创建新字典时也适用这条规则。

```
    set example [dict create firstname Ann initial E\
        surname Huan]
⇒ firstname Ann initial E surname Huan
    dict replace $example initial Y
⇒ firstname Ann initial Y surname Huan
    dict replace $example title Mrs surname Boddie
⇒ firstname Ann initial E surname Boddie title Mrs
```

正如您所看到的，dict replace 也可以用来添加关键字。如果希望删除一个字典中的某个关键字，使用 dict remove 命令。注意，试图删除一个不存在的关键字并不会导致错误。

```
    dict remove $example initial
⇒ firstname Ann surname Huan
    dict remove $example firstname title
⇒ initial E surname Huan
```

dict merge 命令通过融合两个或更多字典，创建一个新字典，源字典分别作为独立的参数提供。当两个或多个字典有同样的关键字时，生成的字典会把这个关键字映射到命令中最后出现该关键字的字典中的关联值。

```
    set colors1 {foreground white background black}
    set colors2 {highlight red foreground green}
    dict merge $colors1 $colors2
⇒ foreground green background black highlight red
```

当有一个作为变量的字典时，可以通过添加关键字，改变关键字的映射或删除关键字来更新它。前两个操作由 dict set 命令完成，第三个操作由 dict unset 命令完成。dict set 命令获取要更新的变量名，要创建或更新的关键字以及关键字对应的关联值，并把更新后的字典写回原字典变量中。dict unset 命令也要获取同样的参数，只是不需要关联值。

```
    set example [dict create firstname Ann initial E\
        surname Huan title Miss]
⇒ firstname Ann initial E surname Huan title Miss
    dict set example title Mrs
⇒ firstname Ann initial E surname Huan title Mrs
    dict get $example title
```

```
⇒ Mrs
   dict set example surname Boddie
⇒ firstname Ann initial E surname Boddie title Mrs
   dict get $example surname
⇒ Boddie
   dict unset example initial
⇒ firstname Ann surname Boddie title Mrs
   dict get $example initial
Ø  key "initial" not known in dictionary
```

7.4 检测字典：子命令 size、exists、keys 和 for

一旦有了一个字典，就可以执行很多操作来检测它。最简单的一个是确定字典元素的个数。这可以通过 dict size 命令完成。

```
   dict size {firstname Ann surname Huan title Miss}
⇒ 3
   dict size {}
⇒ 0
   set example {}
   dict set example a alpha
   dict set example b bravo
   dict set example c charlie
   dict set example d delta
   dict set example e epsilon
   dict size $example
⇒ 5
```

您可以用 dict exists 子命令检查字典中是否存在指定的关键词。如果它返回 1，则 dict get 命令可成功地从这个字典中取得该关键词对应的值，如果返回 0，则 dict get 命令会失败，因为字典中没有那个关键字。

```
   set example {title Miss firstname Ann surname Huan}
   dict exists $example firstname
⇒ 1
   dict exists $example initial
⇒ 0
```

要获得字典中所有关键字的列表(按顺序)，可以使用 dict keys 命令。这个命令将字典的关键字组成列表，还可以根据关键字与指定模式的 string match 匹配情况进行筛选。继续上面那个示例。

```
   dict keys $example
⇒ title firstname surname
   dict keys $example {*name}
⇒ firstname surname
```

类似地，字典中的关联值也可以按顺序组成列表，即 dict values 命令，该命令也可以进行模式筛选。

```
   dict values $example
⇒ Miss Ann Huan
   dict value $example *n*
⇒ Ann Huan
```

要遍历字典的关键字和关联值，并分别对它们执行一些代码，可以使用 dict for 命令。它获取一对变量的列表作为一个参数(一个是关键字，一个是与关键字相关联的值)，并获取一个字典以及构成循环块的 Tcl 脚本。该命令的返回值是空字符串。与 foreach 相似，也可以使用 break 和 continue 来停止循环或跳到字典中的下一个关键字-关联值对。

例如，整齐地打印输出一个字典的内容。

```
# Pretty print using the format command
dict for {key value} $dict {
    puts [format "%s: %s" $key $value]
}
# The result is empty, and the following lines are
# printed to the console
title: Miss
firstname: Ann
surname: Huan
```

利用字典会保持其关键字的顺序这一事实，可以方便地进行排序操作。从第一个字典创建第二个字典，关键字已经处于排序状态，然后使用 dict merge 子命令与第一个字典的值融合。

```
    proc sortDict {dictionary} {
        set sorted{}
        foreach key [lsort [dict keys $dictionary]] {
            dict set sorted $key {}
        }
        return [dict merge $sorted $dictionary]
    }
    sortDict $example
⇒   firstname Ann surname Huan title Miss
```

7.5 更新字典中的值

有时候需要对字典中的值进行更新，而不是用新的值来替换。dict 命令提供了很多便利的子命令(基于其他 Tcl 命令)让这些操作更容易，现在无需把字典的值取出并放入变量，更新变量的值，再把值写回去。

要在字典的值中添加一个字符串或一些字符串，最简单的方法是使用 dict append 命令。它获取想要更新的字典变量名，想要更新其关联值的关键字，想要添加到值中的一个或多个字符串，返回更新后的字典，并把它写回原字典变量。

```
    set example {firstname Ann surname Huan title Miss}
    dict append example firstname ie
⇒   firstname Annie surname Huan title Miss
```

类似地，当您想要在字典值中创建列表时，可以使用 dict lappend 子命令。

```
    set shopping {fruit apple veg carrot}
    dict lappend shopping fruit orange
⇒   fruit {apple orange} veg carrot
    dict lappend shopping fruit banana
⇒   fruit {apple orange banana} veg carrot
    dict lappend shopping veg cabbage beans
⇒   fruit {apple orange banana} veg {carrot cabbage beans}
```

dict 命令支持的另一个更新操作是 dict incr。它获取一个包含字典的变量，一个将要对其关联值进行增加操作的关键字，以及一个可选的值，用来设置增加量。命令的结果是更新后的字典，也会写回原字典变量中。与普通的 incr 命令相似(Tcl8.5 及更新版本)，指定的关键字不必事先在字典中存在，如果不存在，就视为事先存在值为 0 的该关键字项。在统计一段文字中各单词的出现次数时，这一点特别有用。例如下面这个过程：

```
proc computeHistogram  {text} {
    set frequencies {}
    foreach word [split $text] {
        #Ignore empty words caused by double spaces
        if {$word eq " "} continue
        dict incr frequencies [string tolower $word]
    }
    return $frequencies
}
computeHistogram "this day is a happy happy day"
⇒ this 1 day 2 is 1 a 1 haapy 2
```

显然，dict 命令无法用一个命令来支持所有类型的更新操作。而是提供了一个通用命令，把字典中指定的关键字与一些值临时性地关联起来。利用这个命令，可以创建任何类型的更新方案。这就是 dict update 子命令，它获取包含一个字典的变量，字典中的关键字的一个列表，及与关键字关联的变量，以及由 Tcl 脚本描述的、要对这些变量进行的更新操作。这个脚本的结果也是整个 dict update 命令的结果。执行完脚本后，变量会被写回原来的字典变量中，从而允许对字典进行任意复杂的更新。

提示： 如果一个指定的关键字在开始 dict update 命令时的字典中不存在，那么在开始执行脚本块的时候该关键字的值为未设置。而在脚本的最后，没有关联值的关键字会从字典中移除；即如果关联值不存在，关键字也不存在。

使用这种机制可以进行各种各样的更新。例如，下面这个示例是对 dict unset 命令的复用。

```
dict update aDictionaryVariable $theKey localVar {
    unset localVar
}
```

还可以进行更复杂的更新。下面是在两个关键字之间交换关联值的示例。

```
set example {firstname Ann surname Huan title Miss}
dict update example firstname v1 surname v2 {
    lassign [list $v1 $v2] v2 v1
}
puts $example
⇒ firstname Huan surname Ann title Miss
```

下面这个示例演示了如何对一个值求平方。

```
proc squareValue {dictVar key} {
    upvar 1 $dictVar d
    dict update d $key v {
        set v [expr {$v ** 2}]
    }
}
set polyFactors {C 1 x 2 y 3}
squareValue polyFactors y
```

```
⇒ C 1 x 2 y 9
```

这里有一个重要特点，更新只发生在 dict update 命令块结束的时候，未被子命令映射的关键字-关联值对是不会受到影响的。明白这个特点可以更容易地理解进行复杂的更新操作时系统的行为。

```
    set example {firstname Ann surname Huan title Miss}
⇒ firstname Ann surname Huan title Miss
    set i "a dummy value"
    dict update example surname s notes n initial i {
        dict set example title Mrs
        unset s
        set n "have initital = [info exists i]"
        # Print the current contents of the dictionary
        puts $example
    }
⇒ first name Ann surname Huan title Mrs
    # Get the contents of the variable after the dict
    # update command has completed; note it is different
    # in several respects, but the value for 'title' is
    # unchanged because that key was not listed at the
    # start of the command.
    puts $example
⇒ firstname Ann title Mrs notes {have initial = 0}
```

注意尽管在 dict update 的脚本块中 s 变量被移除了，这个移除操作要在命令结束，更新后的字典写回变量时才会发生，并移除 surname 关键字。类似地，关键字 initial 开始时在字典中并不存在，所以变量 i 未设置。

7.6 使用嵌套字典

很多 dict 命令都可以支持嵌套字典。允许在命令中指定多个关键字，为命令指明在嵌套字典中进行操作的位置，这与文件系统中的目录路径很相似。支持这种工作方式的子命令有 dict get、dict exists、dict set、dict unset 以及 dict with。

dict get 命令在嵌套的字典中的行为最简单。当要求嵌套字典中的一个值时，它就依次使用指定的关键字，使用前面的关键字选择一个字典，在这个字典中查找后面一个关键字。因此，命令：

```
dict get $dictionary keyOne keyTwo
```

与下面这条命令完全相同。

```
dict get [dict get $dictionary keyOne] keyTwo
```

针对嵌套字典的 dict exists 命令与针对一般字典的一致。它会检查指定的各个关键字是否存在，而且沿着关键字路径中的各级关键字本身都应该是字典；如果某个关键字存在，却不是字典，那就会引起错误。因此，下面这两条命令是等同的。

```
dict exists $dictionary keyOne keyTwo
expr {
    [dict exists $dictionary keyOne] &&
    [dict exists [dict get $dictionary keyOne] keyTwo]
}
```

正如您所看到的，在使用嵌套字典时应用 dict get 和 dict exists 的多关键字版本可以让编程更加容易。

dict set 和 dict unset 命令的嵌套字典版本的行为，要用单层字典的对应 dict 命令来完成就更加复杂，复杂到我们这里不给出使用单层字典的对应命令编写的有同样功能的代码。dict set 命令会沿着必要的路径创建字典，而 dict unset 会删除路径最里层的字典的指定关键字；即使字典变成空字典，它也不会删除构成路径结构的那些字典。

```
    set nestedDict {firstname Ann surname Huan}
⇒ firstname Ann surname Huan
    dict set nestedDict address street {Ordinary Way}
⇒ firstname Ann surname Huan address {street {Ordinary Way}}
    dict set nestedDict address city Springfield
⇒ firstname Ann surname Huan address {street {Ordinary Way} city
  Springfield}
    dict get $nestedDict address street
⇒ Ordinary Way
    dict unset nestedDict address street
⇒ firstname Ann surname Huan address {city Springfield}
```

另一个设计为使用嵌套字典的命令是 dict with 子命令。与 dict update 命令相似，它允许把一个字典“打开”到变量，但是也有一个不同之处。dict with 并没有让您控制哪些关键字要处理，也没有让您控制它们绑定到哪些值，dict with 把指定的字典或子字典(由关键字路径指定)完全地打开。

```
set example {
    A {
        alphabet    {a alpha b bravo c charlie}
        animals     {cow calf sheep lamb pig ? goose ?}
    }
    C {
        comedians   {laurel&hardy morecambe&wise}
    }
}
dict with example C {
    puts "comedians: $comedians"
    lappend comedians "steven martin"
}
⇒ comedians: laurel&hardy morecambe&wise
⇒ laurel&hardy morecambe&wise {steve martin}
dict with example A alphabet {
    puts "NATO ABC: $a $b $c"
}
⇒ NATO ABC: alpha bravo charlie
dict with example A animals {
    set pig piglet
    set goose gosling
}
dict with example A {
    dict for {k v} $animals {
        puts "$k has baby $v"
    }
}
⇒ cow has baby calf
⇒ sheep has baby lamb
⇒ pig has baby piglet
⇒ goose has baby gosling
```

dict with 命令可以用于向全局变量装载需要长期存在的过程状态，而不至于用大量的变量污染全局命名空间；也不必去把对全局状态的使用都封装到数组结构当中，虽然这的确是一种解决办法。对这种用法的演示见如下示例的 stepCounter 过程。

```
set counters {
    next 0
}
proc makeCounter{{step 1} {offset 0}} {
    global counters
    set id counter[dict get $counters next]
    dict incr counters next
    dict set counters $id [dict create \
        state 0 step $step offset $offset]
    return $id
}
proc stepCounter {id} {
    global counters
    dict with counters $id {
        return [expr {[incr state $step] + $offset}]
    }
}
```

使用时，这会创建一个非常简单的状态计数系统，而且可以指定任意的步长和初值，而全部的消耗只是一个全局变量。下面是这个易扩展机制的一个示例，这里创建了两个计数器，然后交替使用它们。

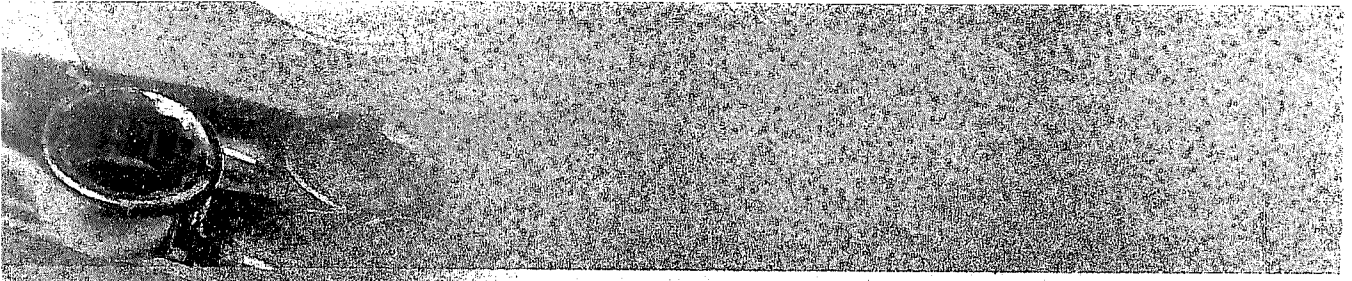
```
    set a [makeCounter]
⇒ counter0
    set b [makeCounter 5 -4]
⇒ counter1
    stepCounter $a
⇒ 1
    stepCounter $a
⇒ 2
    stepCounter $b
⇒ 1
    stepCounter $a
⇒ 3
    stepCounter $b
⇒ 6
    stepCounter $a
⇒ 4
    stepCounter $b
⇒ 11
    puts $counters
⇒ next 2 counter0 {step 1 state 4 offset 0} counter1 {step 5 state 15
  offset -4}
```

字典的 dict with 子命令的另一个重要应用是表示数据库查询的结果。结果各行的每一列都可以给定一个独有的名称(如表格的各列的名称)，而该列的内容就作为字典的值与名称关联[①]，字典的迭代顺序就是这些列的迭代顺序。对于格式齐整的查询结果(实际上大部

① 空列是表示无关键字的最好方法，因为 NULL 值实际上表示的是该行特定列没有值。当然，数据库界面可能也会提供处理特定列的 NULL 的选项(例如特定的字符串)。不过这是字典之外的问题了。它们只是提供机制，而不决定接口的方针。

分结果都是如此)，就可以很方便地使用 dict with 把这个字典映射到变量。

```
set result [dbConn query{
    SELECT firstname, surname, title FROM staff
    LIMIT 2
}]
⇒ {firstname Joe surname Schmoe title Mr} {firstname Annie surname Huan
title Miss}
foreach row $result{
    dict with row {
        puts "found $title $firstname $surname"
    }
}
⇒ found Mr Joe Schmoe
⇒ found Miss Annie Huan
```



第8章 流程控制

本章讲述用于控制脚本执行流程的 Tcl 命令。Tcl 的流程控制命令与 C 程序语言和 Unix 外壳 csh 的流程控制语句相似，包括 if、while、for、foreach、switch 以及 eval。

8.1 本章出现的命令

您可以使用下面的命令控制 Tcl 脚本的执行流程。

- `break`
  终止最内层的循环命令。
- `continue`
  终止最内层循环的当前迭代步，进行该命令的下一个迭代步。
- `eval` *`arg ?arg arg...?`*
  用空格分隔符把所有的 *arg* 串接起来，然后把它作为 Tcl 脚本处理，返回其结果。
- `for` *`init test reinit body`*
  将 *init* 作为一个 Tcl 脚本运行，然后将 *test* 作为一个表达式处理。如果 *test* 的值为真，将 *body* 作为 Tcl 脚本运行，然后将 *reinit* 作为 Tcl 脚本运行，然后再将 *test* 作为一个表达式进行处理。重复这个过程直到 *test* 的值为假。此命令返回一个空字符串。
- `foreach` *`varName list body`*
  `foreach` *`varlist1 list1 ?varlist2 list2...? body`*
- 把变量 *varName* 按顺序设置为 *list* 中的每一个元素，然后将 *body* 作为 Tcl 脚本运行。返回一个空字符串。*list* 必须是有效的 Tcl 列表。通常情况下，foreach 可以对多个列表进行迭代，在每次迭代中对各个列表的多个元素进行处理。
- `if` *`test1 body1 ?elseif test2 body2`* `elseif ...?` *`?else bodyn?`*
  将 *test1* 作为表达式处理。如果其值为真，将 *body1* 作为 Tcl 脚本运行，返回它的值。否则将 *test2* 作为表达式处理；如果其值为真，将 *body2* 作为 Tcl 脚本运行，返回它的值。如果所有的测试都为假，则将 *bodyn* 作为 Tcl 脚本运行，返回它的值。
- `source ?-encoding` *`encodingName? Filename`*
  读取名为 *fileName* 的文件，将它的内容作为 Tcl 脚本处理。返回这个脚本的结

果。Tcl 使用操作系统默认字符编码读取文件，除非用-encoding 选项指定编码。

- `switch ?options? string {pattern body ?pattern body ...?}`
  `switch ?options? string pattern body ?pattern body...?`
  把 *string* 和各个 *pattern* 进行匹配，直到找到一个匹配的 *pattern*，然后执行它后面对应的 *body*。返回执行该 *body* 的结果，如果没有匹配的 *pattern*，则返回空字符串。*options* 可以是-exact、-glob、-regexp 或--(用来代表选项结束)。使用-regexp 匹配时，可以用-matchvar 和-indexvar 访问匹配的正则表达式子模式。
- `while test body`
  将 *test* 作为表达式处理。如果其值为真，则将 *body* 作为 Tcl 脚本运行，然后再处理 *test* 重复这个过程，直到 test 的值为假。返回一个空字符串。

8.2 if 命令

if 命令处理一个表达式，检测它的结果，然后根据这个结果选择执行脚本。考虑下面的命令，它在变量 x 以前的值为负时将它设置为 0。

```
if {$x < 0} {
    set x 0
}
```

这种情况下 if 获取了两个参数。第一个参数是一个表达式，第二个参数是一个 Tcl 脚本。这个表达式可以是第 4 章讲述的任意形式的表达式。if 命令处理这个表达式并检测它的值：如果为真，则处理 Tcl 脚本；如果为假，这个 if 就不再进行更多的操作而直接返回。

if 命令可以拥有多个 elseif 子句，每个子句有相应的表达式和脚本，最后还可以拥有一个 else 子句，这个子句中的脚本在前面各子句的测试都不通过时运行。

```
if {$x < 0} {
    ...
} elseif {$x == 0} {
    ...
} elseif {$x == 1} {
    ...
} else {
    ...
}
```

这条命令会执行用...表示的 4 块脚本中的某一块，执行哪一块取决于 x 的值。这个命令的结果是被执行的那块脚本的结果。如果一条 if 命令没有 else 子句，而所有的检测都不通过时，它就不会执行任何脚本，直接返回一个空字符串。

记住，对 if 和其他控制结构获取的表达式和脚本所用的解析方法与对 Tcl 所有命令的所有参数所用解析方法相同。建议总是把表达式和脚本放在大括号中，这样直到命令执行前都不会有替换发生。而且，每一个左大括号都必须在它的前一个单词的同一行，因为换行符就是命令分隔符。下面这段脚本就会被解析为两个命令，报告 if 命令因参数不足而导致错误。

```
if {$x < 0}
{
    set x 0
}
```

8.3 switch 命令

switch 命令用一个值与很多模式比较，执行能匹配的那个模式所对应的 Tcl 脚本。switch 命令的效果也可以由 if 命令加上很多 elseif 子句达到，但是 switch 支持的表达式结构更加广泛。Tcl 的 switch 命令有两种形式，以下为第一种。

```
switch $x {a {incr t1} b {incr t2} c {incr t3}}
```

switch 的第一个参数是要检测的值(这个示例中就是变量 x 的内容)。第二个参数是包含一个或多个元素对的列表。每一对的第一个元素是要与检测值进行比较的模式，第二个元素是如果模式匹配将执行的脚本。switch 命令顺序处理这些元素对，用它们的模式和检测值进行比较。一旦找到一个匹配，它就执行相应的脚本，将脚本的值作为它的值返回。如果没有匹配的模式，就不执行脚本，switch 返回一个空字符串。在上述示例命令中，如果 x 的值为 a，将其加 t1；如果 x 为 b，加 t2；如果 x 为 c，加 t3；其他情况下什么也不做。

第二种形式将模式和脚本作为独立的变量传入，而不是把它们先组成一个列表。

```
switch $x a {incr t1} b {incr t2} c{incr t3}
```

这种形式的优势在于，想对模式参数调用替换更加容易。但是更多人喜欢第一种形式，因为可以很方便地把模式和对应的脚本写成多行格式，例如：

```
switch $x {
    a {incr t1}
    b {incr t2}
    c {incr t3}
}
```

最外层的大括号避免了换行符被作为命令分隔符处理。而第二种形式就必须用反斜线来达到相同的效果，例如：

```
switch $x \
    a {incr t1} \
    b {incr t2} \
    c {incr t3}
```

switch 命令支持三种模式匹配方式。可以在给出检测值之前指定选项，选择需要的方式：-exact 表示严格的字符串比较，-glob 表示 string match 命令下的匹配(详见 5.10 节)，-regexp 选择 5.11 节讨论的正则表达式匹配。默认行为是-exact。

对于正则表达式匹配，还可以指定-matchvar 参数，在它后面跟一个变量名。switch 命令会在这个变量中存入一个列表。第一个元素是与整个正则表达式匹配的字符串；第二个元素由与首先捕获到的子模式匹配的字符组成；以此类推。如果没有与正则表达式匹配的字符，匹配变量(-matchvar 后面那个参数)的值就是一个空列表。-indexvar 选项与-matchvar

相似，但存入的不是匹配的字符，这个索引变量会接收到有关字符索引的一个列表，其中的索引指向匹配的子字符串。索引变量列表的第一个元素是有两个元素的一个子列表，索引指向匹配正则表达式的子字符串的开头和结尾字符；第二个元素中的两个索引指向匹配首先捕获到的子模式的子字符串的开头和结尾字符，以此类推。

提示： 如果检测值从-符号开始，switch 命令可能会把它误认为选项导致错误。所以，通常情况下，应该总是使用--选项来标记选项的结束，确保 switch 命令在任何情况下都能正确地识别检测字符串。

如果 switch 命令的最后一个模式是 default，那么它可以与任意值匹配，因此 switch 在其他的模式都无法匹配时就会执行 default 对应的脚本。例如下面这个示例脚本，它检测一个列表，进行三个计数。第一个，t1，计列表中包含 a 的元素的个数；第二个，t2，计列表中为无符号十进制整数的元素的个数；第三个，t3，计所有其他元素的个数。

```
set t1 0
set t2 0
set t3 0
foreach i $x {
    switch -regexp -- $i {
        a           {incr t1}
        ^[0-9]+$    {incr t2}
        default     {incr t3}
    }
}
```

如果 switch 命令的某个脚本是-，那么 switch 会使用下一个模式对应的脚本。在多个模式对应相同的脚本的时候，这可以使脚本更加简洁，示例如下：

```
switch -- $x {
    a   -
    b   -
    c   {incr t1}
    d   {incr t2}
}
```

这个脚本在 x 是 a、b 或 c 时将 t1 加一，在 x 是 d 时将 t2 加一。

提示： Tcl 新手的一个常见错误是在 switch 语句中错误地进行了注释。您只能在 Tcl 解释器期望找到 Tcl 命令的地方添加注释。也就是说在 switch 语句中，只能把注释加在脚本部分。

```
switch -- $x {
  # This comment will cause an error
  abc {...}
  …
}
swithc -- $x {
  abc {
        # This comment is okay
        ...
```

```
    }
    ...
}
```

8.4 循环命令：while、for 和 foreach

Tcl 提供三个用于循环的命令：while、for 和 foreach。这些命令都用来把一段脚本执行一遍又一遍，它们的不同之处在于进入迭代前的设置，以及它们决定退出循环的方式。

while 命令获取两个参数：一个表达式和一个 Tcl 脚本。它先处理表达式，如果结果非零，就执行 Tcl 脚本。这个过程不断重复直到表达式为假，此时 while 命令终止，返回一个空字符串。例如下面这段脚本，把列表从变量 a 中复制到变量 b 中，并在复制时倒转列表中元素的顺序。

```
set b {}
set i [expr {[llength $a] - 1}]
while {$i >= 0} {
    lappend b {lindex $a $i}
    incr i -1
}
```

for 命令与 while 命令相似，不过它提供更直接的循环控制。用 for 命令倒转列表元素顺序的程序如下：

```
set b {}
for {set i [expr {[llength $a] - 1 }]} {$i >= 0} {incr i -1} {
    lappend b [lindex $a $i]
}
```

for 的第一个参数是初始化脚本；第二个参数是决定终止循环的表达式；第三个参数是再初始化脚本，它在每次执行完一次循环块之后，再次检测终止表达式之前执行；第四个参数是构成循环块的脚本。for 将它的第一个参数(初始化脚本)作为 Tcl 脚本运行，然后处理表达式。如果表达式的值为真，for 执行循环块，然后执行再初始化脚本，然后再处理这个表达式。这个过程不断重复直到表达式结果为假。如果第一个检测时表达式就为假，那么循环块和再初始化脚本都不会运行。和 while 一样，for 也返回一个空字符串作为结果。

for 和 while 是等价的，用其中一个命令能实现的功能，用另一个命令也能完成。然而，for 的优势在于将所有的循环控制信息集中放在一起，易于查看。当然，在某些情况下初始化脚本和再初始化脚本可能太过复杂或者根本不存在，那样的情况下使用 while 循环更合适。

foreach 命令遍历一个列表中的所有元素。例如下面这段脚本，提供了倒转列表的又一个实现方式。

```
set b {}
foreach i $a {
    set b [linsert $b 0 $i]
}
```

最简单的 foreach 应用要获取三个参数。第一个是变量名，第二个是列表，第三个是构成循环体的 Tcl 脚本。foreach 对列表中的每一个元素顺序执行 Tcl 脚本块。在每次执行循环脚本块前，foreach 将变量(第一个参数指定的)设为列表的下一个元素。因此，如果前面的示例中变量 a 的值为 first second third，那个循环块就会被执行三次。第 1 次，i 的值为 first；第 2 次，i 的值为 second；第 3 次，i 的值为 third。在循环结束时，b 的值为 third second first，而 i 的值为 third。和其他循环命令一样，foreach 总是返回空字符串。

除了使用一个简单变量名，foreach 命令也接受变量名列表。这种情况下，每次循环都会依次把元素值赋给对应的变量，因此如果提供三个变量名，foreach 每次就处理传到列表中的三个元素。循环重复到所有的元素都被使用过为止；如果最后一次循环时循环列表中剩下的元素少于变量名列表中变量名的个数，没有对应元素的变量就设为空字符串。

```
foreach {x y} {a b c d e} {
    puts "<$x><$y>"
}
⇒ <a><b>
  <c><d>
  <e><>
```

foreach 命令还可以并行地处理多个列表，每个列表需要提供独立的变量集。

```
foreach i {a b} {j k} {v w x y z} {
    puts "i:<$i>, j:<$j>, k:<$k>"
}
⇒ i:<a>,j:<v>,k:<w>
  i:<b>,j:<x>,k:<y>
  i:<>,j:<z>,k:<>
```

8.5 循环控制：break 与 continue

Tcl 提供两个循环控制命令，用于退出部分或全部循环。这些命令的行为与 C 语言中对应的语句相同。它们都不需要任何参数。break 命令让引起最内层循环的命令立即终止循环。例如，假设在前文的倒转列表示例中，我们希望在源列表中发现 ZZZ 时就停止循环。换句话说，希望结果列表包含反转顺序的源列表中从开头到 ZZZ(不包括 ZZZ)的元素。这可以用 break 实现如下：

```
set b {}
foreach i $a {
    if {$i == "ZZZ"} break
    set b [linsert $b 0 $i]
}
```

continue 命令只终止最内层循环的当前迭代步；循环继续执行它的下一迭代步。对于 while 循环，这意味着略过循环块并重新处理终止条件表达式；对于 for 循环，再初始化脚本是由终止条件表达式再处理之前进行的。下面这个示例也是反转列表顺序程序的一个变体，这里 ZZZ 元素就被忽略了，不会复制到新的列表中。

```
set b {}
foreach i $a {
    if {$i == "ZZZ"} continue
```

```
    set b [linsert $b 0 $i]
}
```

8.6 eval 命令

eval 是用于创建和运行 Tcl 脚本的通用构造块。它接受任意多个参数，把它们用分隔符串接起来，然后把串接的结果作为一个 Tcl 脚本处理。所有的 Tcl 解析规则都正常应用于这个脚本，因此这个脚本可以包含多个命令，展开为多行，包含注释等。

eval 的一个用途是生成命令，把它们存放在变量中，然后再把这个变量作为 Tcl 脚本运行。例如下面这段脚本：

```
set reset {
    set a 0
    set b 0
    set c 0
}
...
eval $reset
```

将变量 a、b、c 的值清零，然后调用 eval 命令。这种情况下，把这些脚本赋给一个变量，再用 eval 执行它没有什么优势；直接执行三条 set 命令更为合理。但是如果编写一个应用程序，其中的脚本是作为一个动态过程的结果产生的，那么 eval 就是运行该脚本的适当方法。

历史上 eval 最重要的用途曾经是强制进行另一级的解析。在解析一条命令时，Tcl 解析器只进行一级解析和替换；一次替换的结果不会再被解析用于进行一次替换。然而，有时候另一级的解析是必需的，eval 就提供了完成这个任务的机制。

通常，这种情况出现在您有存放在一个变量中或作为一个命令的返回值的一个值的列表，而您需要把这个列表作为分开的值传给一个命令的时候。例如，假设变量 vars 包含了一个变量列表，您希望删除列表中的变量。一个解决方案是使用如下脚本：

```
set vars {a b c d}
foreach i $vars {
    unset $i
}
```

这个脚本能完成任务，但是 unset 命令可以接受任意多个参数，因此应该可以用一条命令就删除所有的变量。遗憾的是，下面这个脚本不能工作。

```
set vars {a b c d}
unset $vars
```

这个脚本的问题在于传给 unset 命令的变量列表是一个参数，而不是分开的各个变量名。因此 unset 会试图删除一个名为 a b c d 的变量。

对于 Tcl 8.5，更好的解决方案是使用{*}语法进行参数展开，如 2.8 节所述。

```
set vars {a b c d}
unset {*}$vars
```

对于 Tcl 8.5 以前的版本，唯一的解决方案就是使用 eval，示例命令如下：

```
set vars {a b c d}
eval unset $vars
```

eval 将它的参数串接起来，形成新的命令 unset a b c d，将它传给 Tcl 进行处理。命令字符串会得到解析，各个变量名作为独立的参数传给 unset。

提示： 只要这个示例中的变量名是在一个规则的 Tcl 列表中，只有空格或制表符作为元素分隔符，这种方法甚至在有些变量名包含空格或$这样的特殊字符时也能正常工作。命令：

```
eval unset $vars
```

等同于命令：

```
eval [concat unset $vars]
```

在上面两种情况下，由 eval 处理的脚本都是规则的列表，其第一个元素是 unset，其他元素是 vars 的元素。

8.7 从文件运行：source

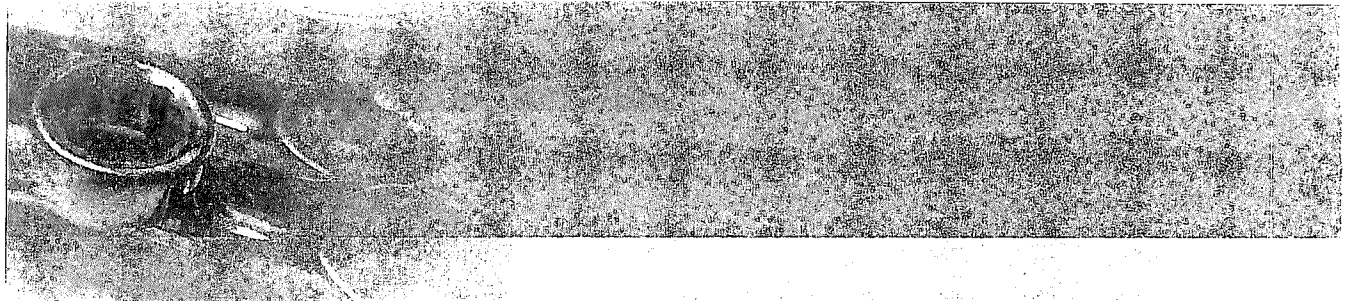
source 命令读取一个文件，将其内容作为 Tcl 脚本运行。source 获取一个参数，该参数指定要读取的文件名。例如，命令：

```
source init.tcl
```

运行 init.tcl 文件的内容。您可以用绝对路径指定文件，或是与当前运行的脚本的工作目录相对的路径指定文件。

source 的返回值就是运行文件内容的返回值，即文件中最后一条命令的返回值。另外，source 允许在文件内的脚本中使用 return 命令终止过程。有关 return 的更多信息参见 9.2 节。

使用 source 命令，可以将一个大的脚本分解为小的模块，由一个主脚本用 source 调用其他的脚本模块。您可以通过把过程定义放到一个文件中，把可重用的过程建成库，然后可以从多个应用程序中用 source 调用它。更多有关脚本库的详细信息参见第 14 章。



第9章 过 程

一个 Tcl 过程就是用 Tcl 脚本定义的一个命令。本章讲述 proc 命令，可以用它随时定义新的过程。过程概念让您可以轻松地把问题的解析方案打包，使它们易于重用。

Tcl 还提供了处理变量作用域的特殊命令。这些命令还允许您使用引用而非值来传递参数，并能把新的 Tcl 控制结构实现为过程。

9.1 本章出现的命令

与过程和变量作用域相关的命令如下。

- `proc name argList body`
  定义一个名为 *name* 的过程，如果已经有名为 *name* 的命令，则将其取代。*argList* 是一个列表，其元素是过程的参数，*body* 包含的 Tcl 脚本即为过程块。返回一个空字符串。
- `apply {argList body ?namespace?} ?arg1 arg2 ... ?`
  对一个匿名过程应用参数并返回结果。过程定义包括两个或三个元素列表。*argList* 和 *body* 元素的用法与 proc 中相同。可选元素 *namespace* 指定处理这个过程的命名空间。
- `return ?option? ?value?`
  从最里层的嵌套过程或 source 命令中返回，将 *value* 作为过程的结果。*value* 的默认值为空字符串。更多的选项可以用于触发异常返回(参见 13.5 节)。
- `global name1 ?name2 ... ?`
  将变量名 *name1*、*name2* 等绑定为全局变量。在当前过程中引用这些名字将指向全局变量而非局部变量。该命令返回一个空字符串。
- `upvar ?level? otherVar1 myVar1 ?otherVar2 myVar2 ...?`
  将名为 *myVar1* 的局部变量绑定到堆栈层级为 level 名为 *otherVar1* 的变量。在当前过程中引用 *myVar1* 会直接指向 *otherVar1*。还可以指定 *otherVar2*、*myVar2* 等绑定更多的变量。*level* 的语法和含义与 uplevel 中相同，默认值为 1。该命令返回一个空字符串。
- `uplevel ?level? arg ?arg arg ... ?`
  把所有的 *arg* 参数用空格作分隔符串接起来，然后把串接的结果作为 Tcl 脚本运

行，运行时变量上下文关系所在的堆栈层级为 *level*。*level* 为一个数字或以#接一个数字，默认值为 1。返回该脚本的结果。

9.2 过程基础：proc 与 return

过程由 proc 命令创建，示例如下：

```
proc plus {a b} { expr {$a+$b} }
```

传给 proc 的第一个参数是要创建的过程的名称，这里是 plus。第二个参数是该过程使用的参数的名称(这里是 a 和 b)的一个列表。proc 的第三个参数是构成新的过程块的 Tcl 脚本。在 proc 命令完成后，新的命令 plus 就存在了，可以像任何其他 Tcl 命令一样调用。调用 plus 时，Tcl 会把 a 和 b 的值设为给出的参数值，处理过程块。在调用 plus 时必须给定两个参数；如果调用过程时给出的参数数目不对，Tcl 解释器会产生一个错误。plus 命令的返回值就是它的过程块中最后一个命令的返回值。下面是一些对 plus 正确的或错误的调用。

```
    plus 3 4
⇒   7
    plus 3 -1
⇒   2
    plus 1
∅   wrong # args: should be "plus a b"
```

认识到 proc 就是一个普通的 Tcl 命令这一点很重要。它不是有特殊语法的声明，例如您在 C 语言等其他语言中看到的情形。传给 proc 的参数的处理方法和传给任何其他 Tcl 命令的参数的处理方法是相同的。例如，参数{a b}中那个大括号并不是针对这个命令的特殊语法结构；它的作用与通常情况相同，就是把 plus 的两个参数名作为参数名的一个列表传给 proc。技术上说，如果一个过程仅有一个参数，那这个参数名从语法上说不必用大括号括起来，不过从一致性的角度考虑，很多 Tcl 程序开发者还是会使用大括号。类似地，proc 的最后一个变量外的大括号用于将整个过程块作为一个参数传给 proc 而不对它进行替换操作。

如果一个过程在执行完全部脚本前就提前返回，可以调用 return 命令：它让最内层的过程立即返回，return 的参数就是该过程的返回值。下面是使用 return 命令的阶乘功能实现。

```
    proc fac {x} {
        if {$x <= 1} {
            return 1
        }
        return [expr {$x * [fac [expr {$x-1}]]}]
    }
    fac 4
⇒   24
    fac 0
⇒   1
```

如果 fac 的参数小于或等于 1，fac 调用 return 立即返回。其他情况下 fac 会执行 expr 命令，该命令递归地调用 fac 并返回结果。

提示： 在这个示例中调用 expr 命令时可以不必把它“框在” return 命令里边。这个 expr 命令是过程块中的最后一个命令，因此它的结果会作为过程的结果返回。不过，使用 return 命令可以让意图更加明显，有利于代码维护，这样其他人就不太会在过程的尾部添加命令，从而避免无意中改变过程的行为。

9.3 局部和全局变量

当 Tcl 处理过程块时，它会使用与调用者不同的变量集。这些变量称为局部变量，它们仅由这个过程访问，在过程返回后就被删除。在过程之外定义的变量称为全局变量，全局变量是长期存在的，仅在明确删除后才会消失。Tcl 还支持命名空间变量，这些变量在特定的命名空间上下文环境中长期存在。第 10 章将讨论命名空间和命名空间变量的使用。一个局部变量可以与全局变量、命名空间变量或另一个过程中的局部变量同名，而它们是不同的变量：即改变其中某个变量不会影响其他的变量。如果过程是递归调用的，每一层递归都有不同的局部变量集。

传给过程的参数就是局部变量，它们的值根据调用过程的命令中的单词进行设置。在过程开始运行的时候，只有那些与传给过程的参数对应的局部变量有值。其他的局部变量在设置它们时自动创建。

过程可以使用 global 命令引用全局变量。例如，下面这条命令让全局变量 x 和 y 在过程中可见。

```
global x y
```

global 命令把它的每一个参数作为全局变量的名称对待，将过程中对这些变量名的引用定向到全局变量而非局部变量。global 命令可以在过程中的任何时候调用；一旦调用它，它就会一直有效，直到过程返回。

提示： Tcl 没有提供与 C 中的静态变量等同的变量形式，静态变量的作用域限定在指定过程中，但它的值在对该过程的各调用之间持续存在。Tcl 中必须使用全局变量或命名空间变量来达成相同的目标。一般来说，您应该使用命名空间变量以避免变量名冲突。

9.4 参数变量的数目和默认设置

在目前为止的示例中，proc 的第二个参数(描述过程所需获取的参数)都是由参数名组成的简单形式。参数指定方面还有三个其他的功能。首先，参数列表可以指定为一个空字符串。这种情况下过程不获取参数，如果带参数地调用它会产生错误。例如下面这条命令，定义了输出两个全局变量的过程。

```
proc printVars {} {
    global a b
    puts "a is $a, b is $b"
}
```

第二个功能是为部分或全部参数设置默认值。参数列表实际上是列表的列表，每一个子列表对应一个参数。如果子列表只有一个元素(前面的示例都是这样的情况)，那么这个元素就是参数的名称。如果子列表含有两个元素，第一个元素就是参数名，而第二个元素则是它的默认值。例如下面这个过程，把一个给定值加上一个给定增量，给定增量的默认值设为 1。

```
proc inc {value {increment 1}} {
    expr $value+$increment
}
```

参数列表中的第一个元素 value，指定了名称而没有指定默认值。第二个元素指定了名为 increment 的参数，并指定其默认值为 1。这意味着调用 inc 时可以给出一个或两个参数。

```
    inc 42 3
⇒ 45
    inc 42
⇒ 43
```

如果 proc 命令中没有给一个参数设置默认值，在调用过程时就必须给出这个参数。而默认参数，无论是多少个，都必须放在过程的参数列表的尾部。在 proc 命令中和参数调用中都是如此。如果为一个特定参数设置了默认值，那么列表中它后面的所有参数都必须设置默认值。类似地，如果在调用过程时没有指定某个参数，那么它后面的参数也不能再指定。

参数列表中的第三个特殊功能支持可变数量个参数。如果参数列表中的最后一个元素是特殊名称 args，那么调用过程时可以给出可变数量个参数。args 之前的参数的处理方式与前面相同，在它们之后可以再指定任意个参数。过程的局部变量 args 会被设置成一个列表，其元素就是这些更多的参数。如果没有更多的参数，args 会设置为空字符串。例如下面这个过程，获取任意个数字参数，返回它们的和。

```
    proc sum {args} {
        set total 0
        foreach val $args {
            set total [expr {$total + $val}]
        }
        return $total
    }
    sum 1 2 3 4 5
⇒ 15
    sum
⇒ 0
```

如果过程的参数列表在 args 前面还有参数，它们的默认设置方式与前面所讲的相同。args 不能设置默认值——它的默认值就是空字符串。

9.5 传引用调用：upvar

Tcl 只支持参数的传值调用。当调用 Tcl 的命令时，是复制了参数的值然后将其传给命令的。即使参数值来自一个变量也是如此，因为 Tcl 解释器执行命令前会把参数替换为它

的值。因此，在下面这个示例中，sum 命令获取到的是变量 a 和 b 中存储的值的副本。

```
sum $a $b
```

Tcl 也不支持指针和引用类型，因此初看上去无法编写一个过程来改变已经存在的变量值。不过，变量的名称也是一个字符串值，它也可以存到一个变量当中。因此，通过要求进行多一轮的替换，我们可以模拟出引用的行为，例如：

```
    set x "The value of x"
    set y x ; # This stores the single character "x" in y
    set $y
⇒   The value of x
```

这个示例中，Tcl 解释器把$y 替换为它的字符串值 x。然后执行 set 命令，把它的参数解析成一个变量的名称，返回存放在该变量中的值。有了这个概念，使用 Tcl 的 upvar 命令，就可以实现与传引用调用相同的行为。

upvar 命令提供访问当前过程的上下文范围之外的变量的通用机制。可以用于访问全局变量、命名空间变量以及其他活动中的过程内的局部变量。它更常见的使用则是模拟传引用调用的行为，这对数组特别有用。如果 a 是一个数组，就不能像 myproc $a 这样把它传给 myproc 过程，因为并没有对应整个数组的值；只有对应各个数组元素的值。但是，可以把数组的名字传给过程，如 myproc a，然而使用 upvar 命令在过程中访问数组的元素。

下面是在过程中使用 upvar 的简单示例，输出一个数组的内容。

```
    proc printArray {name} {
        upvar $name a
        foreach el [lsort [array names a]] {
            puts "$el = $a($el)"
        }
    }
    set info(age) 37
    set info(position) "Vice President"
    printArray info
⇒   age = 37
    position = Vice President
```

当调用 printArray 时，给出数组的名称作为参数。upvar 命令使得过程可以通过变量 a 访问这个数组。upvar 的第一个参数是过程的调用环境可见的变量。这可以是一个全局变量，如这个示例，也可以是一个命名空间变量，也可以是调用过程的内部变量。第二个参数是一个局部变量的名称。upvar 命令把对局部变量 a 的返回重定向到调用环境中名为 name 的变量。这个示例中，当 printArray 读取 a 时，它读取的都是 info 全局变量的元素。如果 printArray 向 a 写入，它就会修改 info。printArray 使用 array names 命令取得数组中所有元素的列表。然后用 lsort 把它们排序，再顺序输出这些元素。

提示： 这个示例中看上去好像输出是作为过程的结果返回的；事实上，这输出是由过程直接发送到标准输出的，这个过程的结果是一个空字符串。

---

upvar 命令的第一个变量名默认指向当前过程的调用者的上下文环境。不过，它也可以访问调用堆栈中任意层级的变量，包括全局层级。例如：

```
upvar #0 other x
```

使用全局变量 other 可以通过局部变量 x 访问(参数#0 指明 other 应该解释为全局变量，无论现在已经激活了多少层嵌套过程)，而

```
upvar 2 other x
```

使得当前过程的调用者的调用者的上下文环境中的 other 可以通过局部变量 x 访问(2 指明 other 应该在比当前调用堆栈高 2 个层级的环境中)。层级 0(与#0 不同)表示的是当前上下文环境。有关在 upvar 中指定层级的更多信息参见参考文档。

提示： *尽管不指定层级参数时，upvar 默认为当前过程的调用者的上下文环境，但在实际应用中最好明确地指定层级为 1。如果变量名以#或数字开头，明确指定层级可以避免 upvar 产生错误。*

9.6 创建新的控制结构：uplevel

uplevel 命令像是 eval 和 upvar 的结合。它把它的参数作为脚本处理，正如 eval，但处理脚本的变量上下文环境却不在调用堆栈层级中，正如 upvar。使用 uplevel 可以用 Tcl 过程定义新的控制结构。例如，以下为一个新的流程控制命令 do。

```
proc do {varName first last body} {
    upvar $varName v
    for {set v $first} {$v <= $last} {incr v} {
        uplevel $body
    }
}
```

do 的第一个参数是一个变量名。do 把这个变量设为整数，值在第二个参数到第三个参数的范围之间，对该变量的每一个值都把第四个参数作为一个 Tcl 命令处理。有了 do 的定义，下面这个脚本就能生成由 1～5 的 5 个数的平方组成的列表。

```
    set squares {}
    do i 1 5 {
        lappend squares [expr $i*$i]
    }
    set squares
⇒ 1 4 9 16 25
    set i
⇒ 6
```

do 过程使用 upvar 通过局部变量 v 访问循环变量(这里是 i)。然后 do 过程使用 for 命令在指定的范围内对循环变量进行增加操作。对于每一个值，它都调用 uplevel 在调用者上下文环境中的执行循环块；这使得在循环块中对变量 squares 和 i 的调用都指向 do 过程的调用者中的变量。如果用 eval 替代 uplevel，squares 和 i 会被作为 do 的局部变量对待，那就不会产生期望的效果了。

提示： 这里 do 的实现并不能适当地处理异常情况。例如，若循环块中含有 return 命令，它就只会让 do 过程返回，看上去更像是 break 的行为。而 for 或 while 这样的内建流程控制命令的循环块中，如果出现 return 命令，它就会让调用这个过程的命令返回。在第 13 章您会看到如何让 do 也实现这样的功能。

类似于 upvar，uplevel 也接受一个可选的初始化参数，明确地指定堆栈层级。虽然默认情况下脚本会在调用者的上下文环境中处理，但实际应用中最好还是明确地指定堆栈层级为 1。否则，如果脚本参数以#或一个数字开头，它可能被误认为是层级参数。更多细节详见参考文档。

9.7 应用匿名过程

Tcl 中的过程不仅提供了模块化编程的机制，还可以提升性能。过程块在 Tcl 内部会由高效率的二进制代码表示。过程还提供了一个局部范围，可以在其中不透明地创建变量，避免与已经存在的变量发生命名冲突。您可能会遇到希望提升某个过程的性能或把它更好地封装，但这个过程只会被使用一次或有限几次的情况。代码回调，如对组件命令和文件事件处理程序的回调，以及只用一次的转换函数就是典型的例子。

apply 命令(由 Tcl 8.5 引入)提供了把参数集合应用于匿名过程的功能。

```
apply {argList body ?namespace?} ?arg1 arg2 ...?
```

apply 的第一个参数是过程定义，由包含两个或三个元素的列表构成。第一个元素是过程的形式参数，定义方式与 proc 相同。第二个元素是实现过程块的 Tcl 脚本。第三个元素是可选的，如果提供，就指明运行过程的命名空间(有关命名空间的讨论参见第 10 章)。传给 apply 的其他参数是赋给过程参数的值。

作为示例，考虑下面这个匿名过程，它对一些数求和。

```
    apply { {args} {
        set total 0
            foreach val $args {
            set total [expr {$total + $val}]
        }
        return $total
    } } 1 2 3 4 5 6 7
⇒   28
    set total
Ø   can't read "total": no such variable
```

这里，数字 1～7 作为一个列表传给匿名过程的形式参数 args，就和把它们传给一个 proc 定义的有名称的过程一样。然后匿名过程计算并返回这些值的和。注意这个示例中变量 total 是过程的局部变量，在过程终止时会被自动删除。

下面是一个更有代表性的示例，考虑对一个列表依据各元素的字符串长度进行排序的任务。lsort 命令(第 6 章讨论了此命令)没有按照元素的长度排序的选项，但可以用 -command 选项指定自己的排序过程。这个过程必须接受两个元素，如果认为第一个元素小于、等于或大于第二个元素，相应地返回小于、等于或大于 0 的整数。虽然可以定义一

个命名过程来实现比较，但也可以使用匿名过程实现。

```
set states {California Delaware Hawaii Indiana Iowa}
lsort -command { apply { {e1 e2} {
    expr {[string length $e1] - [string length $e2]}
} } } $states
⇒ Iowa Hawaii Indiana Delaware California
```

匿名过程的另一个用途是实现回调，例如变量跟踪(变量跟踪的更多信息参见第 15 章)。调用变量跟踪过程时要给出三个值，指定变量和访问类型。下面这个示例中，当跟踪变量变化时，一个匿名过程在控制台上报告其新值。

```
set vb1 "initial"
trace add variable vb1 write {apply {{v1 v2 op} {
    upvar 1 $v1 v
    puts "updated variable to \"$v\""
}}}
set vb1 123
⇒ updated variable to "123"
set vb1 abc
⇒ updated variable to "abc"
```

另外，apply 可以用于创建实现不同功能的程序结构块。您可以在 Tcl 开发人员的维基网(http://wiki.tcl.tk)上找到一些示例。下面这个示例展示了 map 命令的实现，该命令获取一个列表，按照给定的规则对每个元素进行变化，得到新的列表并返回。

```
proc map {lambda list} {
    set result {}
    foreach item $list {
        lappend result [apply $lambda $item]
    }
    return $result
}
map {x { expr {$x**2} }} {1 2 3 4 5}
⇒ 1 4 9 16 25
map {x { return [list [string length $x] $x]}} {A BB CCC DDDD}
⇒ {1 A} {2 BB} {3 CCC} {4 DDDD}
```

第 10 章 命名空间

Tcl 解释器将所有的命令和全局变量分组管理，这些分组被称为命名空间。一个命名空间中的命令和变量不会影响到另一个命名空间。这些命名空间可以呈树形组织，一个命名空间中包含多个可以被称为子命名空间的对象。命名空间树的根就是全局命名空间，它包含了没有明确在其他命名空间中创建的所有命令和变量。

任何已经存在的命名空间中的命令和变量，都可以从命名空间内部或外部创建。在命令或变量的名称前面加上命名空间前缀就可以完成这一点，前缀与名称之间用命名空间分隔符分开，分隔符为双冒号。全局命名空间的名称是空字符串，但通常只写一个双冒号就可以表达同样的意思。

命名空间的主要用途之一是作为创建相关命令包的机制。命名空间可以帮助创建集合命令，把一个命名空间中的公共 API 编组，呈现为公共命令加子命令的模式。

10.1 本章出现的命令

- `namespace children ?namespace? ?pattern?`
  返回指定的命名空间的所有子命名空间的列表，如果没有给出 *namespace* 参数，就返回当前命名空间名。如果指明了 *pattern*，则只返回与 *pattern* 在通配符规则下匹配的子命名空间名(非限定名称)。
- `namespace code script`
  返回特定格式的 *script*，这个特定格式的含义是：当处理这个 *script* 脚本时，应该在当前命名空间进行处理。这很适合用于生成回调脚本，因为如果脚本回调时有需要添加的参数，在控制命名空间后这些参数可以作为额外参数正确地传递给 *script* 中的命令。
- `namespace current`
  返回当前命名空间的完全限定名称。
- `namespace delete ?namespace ...?`
  删除用名称指定的命名空间。
- `namespace ensemble create ?option value ...?`
  创建一个集合，绑定到当前命名空间，返回集合的完全限定名称。
- `namespace ensemble configure ensemble ?option? ?value ...?`
  配置名为 ensemble 的集合。如果没有指明 *option*，则返回包含所有选项及其值的

字典。如果给出 *option* 而没有给出对应的 *value*，则返回该选项的当前值。或者给出各选项和要把它们设置到的值的列表，设置完成后返回一个空字符串。

- `namespace ensemble exists` *ensemble*
  如果 *ensemble* 是一个集合命令，则返回 1，否则返回 0。
- `namespace eval` *namespace body*
  在名为 *namespace* 的命名空间中处理脚本，如果该命名空间还不存在，就创建它。该命令的结果是 *body* 中最后一条命令的结果。
- `namespace exists` *namespace*
  如果名为 *namespace* 的命名空间存在，返回 1，否则返回 0。
- `namespace export ?-clear? ?`*pattern* `...?`
  如果指定了一个或多个 string match 类型的模式，把它们添加到当前命名空间的命令导出模式列表中。如果指定了-clear 选项，就先清空导出模式列表。如果没有提供模式，则返回当前的导出模式列表。
- `namespace forget ?`*pattern* `...?`
  前面导入到当前命名空间的命令中，按 string match 规则与给定的模式相匹配的那些命令将被“遗忘”：即从当前命名空间中删除。非导入的命令和导出的原始命令不受影响。
- `namespace import ?-force? ?`*pattern* `...?`
  按 string match 规则与给出的一个或多个模式相匹配的命令会被导入当前命名空间中。命令还必须与它们所在的原始命名空间的导出模式相匹配。每一个模式都是一个限定的通配符模式，由命名空间名前缀和通配符模式后缀构成。当前命名空间中导入的命令的局部名称，与它在原始命名空间中的局部名称相同。导入与已经存在的命令同名的命令将导致错误，除非指定了-force 选项，该选项表示将已经存在的命令替换掉。
- `namespace origin` *command*
  返回由 *command* 实现的命令的完全限定名称。如果 *command* 是由另一个命名空间导入的，返回实际实现的限定名称。
- `namespace parent ?`*namespace*`?`
  返回 *namespace* 命名空间的父命名空间，如果没有给定 *namespace* 参数，则返回当前命名空间的父命名空间。
- `namespace path ?`*list*`?`
  设置或返回当前命名空间的名称解析路径。如果给出 *list*，将其设置为列表指出的名称解析路径。如果没有给出 *list*，则返回当前的名称解析路径。
- `namespace qualifiers` *string*
  返回直到 *string* 中的最后一个命名空间分隔符的全部。如果 *string* 中没有包含任何命名空间分隔符，返回一个空字符串。注意这个命令不会检查出现的命名空间是否都存在。

- namespace tail *string*
  返回 *string* 中的最后一个命名空间分隔符后面的全部。如果 *string* 中没有包含任何命名空间分隔符，则不加改动地返回 *string*。注意这个命令不会检查出现的命名空间是否都存在。
- namespace unknown ?*script*?
  返回当前命名空间的未知命令处理程序脚本，或将其设置为给出的 *script* 脚本(如果给出的话)。对所有命名空间，默认的未知命令处理程序都是::unknown。
- namespace upvar *namespace otherVar localVar* \? *otherVar localVar ...*?
  将命名空间 *namespace* 中名为 *otherVar* 的变量映射到局部变量 *localVar*。这些变量会被分别链接起来；对两个链接变量中任一个的修改都会改变另一个。
- namespace which ?-command|-variable? *name*
  返回名为 *name* 的实体的完全限定名称。如果指定-command 选项，那 name 会被认为表示的是命令。如果指定-variable 选项，那 *name* 会被认为表示的是变量。
- variable *varName* ?*value*? ?*varName value ...*?
  创建或指向当前命名空间中的变量，并进行可选的初始化，即如果给出值，将其设置为指定的值。

10.2 在命名空间中处理 Tcl 脚本

要在一个命名空间中处理 Tcl 脚本，应该使用 namesacpe eval 命令。这个命令获取命名空间的名称，在指定的命名空间中处理 Tcl 脚本，返回脚本的结果。如果指定的命名空间不存在，则创建该命名空间。proc 命令除给出完全限定的情况外，总是创建与当前命名空间相关的命令。注意脚本中的命令如果在指定命名空间中找不到，就会在全局命名空间中查找。例如：

```
proc whoAmI {} {
    return "global command"
}
namespace eval ns {
    proc whoAmI {} {
        return "namespace command"
    }
}
whoAmI
⇒ global command
ns::whoAmI
⇒ namespace command
namespace eval ns {
    whoAmI
}
⇒ namespace command
```

您还可以声明将在指定的命名空间中进行过程，这需要给 proc 名字指定完全限定名称，如 ns::newCmd。

要删除命名空间，而不是删除空间中的某个命令，需要使用 namespace delete 命令。该命令获取命名空间的一个列表，进行删除。

命名空间中的变量可由 variable 命令设置或访问。这个命令获取变量的名称，在当前命名空间中创建变量，如果给出变量的值，就设置变量的值。如果是在该命名空间的一个过程中进行处理，它还会使得指定名称的变量不作限定就在过程中可见。

```
namespace eval counter {
    variable num 0
    proc next {} {
        variable num
        return [incr num]
    }
proc reset {} {
    variable num
    set num 0
}
}
counter::next
⇒ 1
counter::next
⇒ 2
counter::reset
⇒ 0
counter::next
⇒ 1
```

提示： 总是使用 variable 命令声明变量。在一个命名空间中，如果访问一个没有在该命名空间明确声明的变量，Tcl 会首先在全局变量中查找这个名称的变量。如果找到这样的全局变量，Tcl 会使用它而不是创建一个命名空间变量。Tcl 语言的这种行为初看起来好像不方便，其实这样设计是为了在各个命名空间中，都能更容易地访问预定义的全局变量 argv、env 等。

variable 命令不能初始化数组的值，但它可以在命名空间中设置变量，允许过程访问它们。也就是说，数组需要单独进行一步初始化操作。

```
namespace eval catalog{
    variable entries
    array set entries {}
    proc add {item} {
        variable entries
        incr entries($item)
    }
    proc getEntries {} {
        variable entries
        return [lsort [array names entries]]
    }
    proc countInstances {item} {
        variable entries
        return $entries($item)
    }
}
catalog::add apple
⇒ 1
catalog::add orange
⇒ 1
catalog::add apple
```

```
⇒ 2
    catalog::add banana
⇒ 1
    catalog::getEntries
⇒ apple banana orange
    catalog::countInstances apple
⇒ 2
```

如果需要生成一个处理回调的脚本，例如 lsort 命令的-command 选项，组件的-command 选项，或者本书后面会讨论的 after 或 trace 命令，最好是用 namespace code 命令来生成这些脚本。这会把与脚本相关的代码打包在一起，确保它是在当前命名空间中处理的，而不是在它的调用者的命名空间中处理。例如，我们可以扩展上面那个示例的功能，让它可以把目录存放到一个文件中，这样我们退出解释器时目录就不会丢失了，而且我们可以设置每 10 秒钟自动保存一次目录。

```
namespace eval catalog{
    proc save {file} {
        variable entries
        set f [open $file w]
        puts $f [list array set ::catalog::entries \
            [array get entries]]
        close $f
    }
    proc autocommit {interval file} {
        after $interval [namespace code [list \
            autocommit $interval $file]]
        save $file
    }
    # Save to catalogDB.tcl every 10 seconds
    autocommit 10000 catalogDB.tcl
}
```

10.3 操作限定名称

命名空间中的一切都是经由它的限定名称进行访问的。限定名称由命名空间名称，加上::，再加上命令、变量或子命名空间的局部变量名称组成。限定名称可以是绝对的，以::开头，这个开头的::表示全局命名空间；也可以是相对的，即相对于当前命名空间(不是以双冒号开头)。这与文件系统中的绝对路径和相对路径类似。

创建限定路径时，可以只用::前缀表示全局命名空间。即::set 指向全局命名空间中的 set 命令，::env 指向全局 env 数组。前文创建的 catalog 命名空间是全局命名空间的直接子命名空间。因此，在全局命名空间中可以使用相对限定名称 counter::add 调用 add 过程。从其他命名空间调用时则需要使用绝对限定名称::counter::add。

Tcl 提供了很多工具来操作这些限定名称。要取得限定名称的命名空间部分，应该使用 namespace qualifiers 命令。该命令返回名称中的命名空间部分。限定名称的局部名称部分可以使用 namespace tail 命令取得。

```
    namespace qualifiers alpha::beta
⇒ alpha
    namespace qualifiers ::alpha::beta::gamma
⇒ ::alpha ::beta
```

```
    namespace qualifiers delta
⇒
    namespace tail alpha ::beta
⇒ beta
    namespace tail ::alpha::beta::gamma
⇒ gamma
    namespace tail delta
⇒ delta
```

要把各部分名称组成限定名称，只需用文本的::把它们串接起来。注意，如果把命名空间保存在一个变量中，需要用下面的后一种形式进行变量替换，否则$变量替换时::会引起错误的解析。

```
    set theNS ::alpha::beta
    set theCommand $theNS::gamma
∅ can't read "theNS::gamma": no such variable
    set theCommand ${theNS}::gamma
⇒ ::alpha::beta::gamma
```

10.4 在命名空间中导出和导入命令

如果一个命名空间提供某种形式的公共 API，那么它应该可以由 namespace export 命令从命名空间中导出。这个命令会修订和解析与命名空间相关的一个通配符模式列表，将命名空间中与其匹配的命令作为它的公共 API 导出。

有了从命名空间中导出的公共 API，就可以在另一个命名空间中导入 API 命令，这样就可以使用导入的命令而无需给出该命令的完整限定名称。这一操作由 namespace import 命令管理，该命令获取一个通配符模式列表，以此确定要导入的命令(从其他命名空间的导入命令集合中搜索匹配，选择要导入的命令)。

```
namespace eval src {
    proc a   {} {return "alpha"}
    proc b   {} {return "beta"}
    proc etc {} {return "..."}
    namespace export a b
}
namespace eval dst {
    namespace import ::arc::*
    proc c {} { return "charlie"}
    expr {" [a] [b] [c]"}
}
⇒ alpha beta charlie
```

提示： 全局命名空间不会默认导出任何命令，而且这一命名空间习惯上是留给应用程序脚本管理的。库不应该从全局命名空间导出命令，也不应该向全局命名空间导入命令。

namespace import 命令有快照的含义：它只导入在调用它时处于可导出状态的命令。而且，默认情况下它不会覆盖已经存在的命令，即使这些命令是以前导入的。可以用-force 选项强制指定进行覆盖。如果希望从命名空间中移除以前导入的命令，而又不冒意外删除自己创建的命令的风险，应该使用 namespace forget 命令。

10.5 检查命名空间

Tcl 提供了很多检查命名空间的工具。可以用 namespace current 命令获得当前命名空间的名称，用 namespace parent 获得当前命名空间的父命名空间的名称，用 namespace children 获得当前命名空间的子命名空间的名称(这个命令获取可选的命名空间和通配符模式参数，以确定要检查的命名空间，把要检查的命名空间形成列表)。

```
namespace eval example {
    namespace current
}
⇒ ::example
namespace eval example {
    namespace parent
}
⇒ ::
namespace eval example {
    namespace eval subNS {
        namespace current
    }
}
⇒ ::example::subNS
namespace eval example {
    namespace eval abc1 {}
    namespace eval abc2 {}
    namespace eval abc3 {}
}
lsort [namespace children example abc*]
⇒ ::example::abc1 ::example::abc2 ::example::abc3
```

要查看命名空间中的命令和变量，应使用通用的 info 子命令，然后指定想查看的命名空间名称以及模式参数。

```
namespace eval example {
    proc eg1 {} {}
    proc eg2 {} {}
}
lsort [info command example::*]
⇒ ::example::eg1 ::example::eg2
```

另外，如果知道一个命令或变量的非限定名称，可以使用 namespace which 获得它的完全限定名称。

```
namespace eval example {
    proc eg   {} {}
    proc env  {} {}
    list [namespace which eg] [namespace which set] \
            [namespace which -variable env]
}
⇒ ::example::eg ::set ::env
```

当前导出模式的列表，可以不带参数地使用 namespace export 命令得到。

要查看一个可被导入的命令是由哪个命名空间原始导出的，可使用 namespace origin 命令。该命令获取一个命令名作参数，返回这个作为参数的命令的原始来源的完全限定名

称。如果这个命令就是在本地命名空间中定义的，返回的就是这个命令的完全限定名称。

```
namespace eval example {
    proc eg1 {} {}
    proc eg2 {} {}
    namespace export *
}
namespace import example::*
namespace origin eg1
⇒ ::example::eg1
```

注意 namespace origin 可以查看到命令的原始来源，即使它被重命名过。

```
rename eg2 example2
namespace origin example2
⇒ ::example::eg2
```

10.6 有关集合命令

把有关的命令组合成群组的常见系统，就是集合命令。这是 Tcl 中的一种常见命令模式——例如 string、namespace 以及 clock 命令等——在 Tk 和很多扩展包中这种模式也很常用。Tcl 提供的集合命令机制允许程序开发者很轻松地创建自己的相关命令群组。只需使用 namespace ensemble 命令就能完成这一任务。

10.6.1 基本的集合命令

创建集合命令的最简单方法是使用 namespace export(如前文所述)发布一个命名空间中组成 API 的命令。在那个命名空间中不带参数地使用 namespace ensemble create，就会基于公共 API 创建集合命令，与命名空间有相同的完整限定名。您还可以用 namespace ensemble exists 命令测试一个命令是否为集合命令，如果给定的参数是集合命令的名称，就会返回真。

```
namespace eval example {
    proc plus {x y} {expr {$x + $y}}
    proc multiply {x y} {expr {$x * $y}}
    proc minus {x y} {expr {$x - $y}}
    namespace export add multiply minus
    namespace ensemble create
}
⇒ ::example
example add 2 3
⇒ 5
example multiply 6.0 7
⇒ 42.0
namespace ensemble exists example
⇒ 1
namespace ensemble exists example::add
⇒ 0
```

命名空间中的子命令在应用时可以缩写(只写开头的一些字符)，但要保证缩写后的子命令是独一无二的。集合命令会调用正确的子命令，但如果缩写后的子命令对应的完整子命令不唯一，集合命令就会抛出匹配错误。

```
    example mul 3.1 1.3
⇒   4.03
    example m -4 -5
Ø   unknown or ambiguous subcommand "m": should be add, minus, or multiply
```

注意集合命令内部的过程在被传入错误数量的参数调用时，会产生“有集合知觉”的错误消息。

```
    example mul 42
Ø   wrong # args: should be "example mul x y"
```

10.6.2 在集合命令中设置集合命令

您可以嵌套地使用集合命令。在用集合命令处理复杂的 API 时这一点常常用到，而在 Tk 组件中复杂的 API 十分常见。在下面的示例中，compute 命令有两个子命令，分别计算形状的面积和实体的体积。对于它们支持的不同形状和不同实体，这两个子命令又各有其子命令。

```
namespace eval compute {
    variable pi 3.1415927
    namespace export *
    namespace ensemble create
    namespace eval area {
        namespace export *
        namespace ensemble create
    }
    namespace eval volume {
        namespace export *
        namespace ensemble create
    }
}
proc compute::area::circle {radius} {
    variable ::compute::pi
    return [expr {$pi * $radious**2}]
}
proc compute::area::square {side} {
    return [expr {$side ** 2}]
}
proc compute::volume::sphere {radius} {
    variable ::compute::pi
    return [expr {4 *$pi / 3 * $radious**3}]
}
proc compute::volume::cube {edge} {
    return [expr {$edge ** 3}]
}
compute ?
Ø   unknown or ambiguous subcommand "?": should be area, or volume
    compute area ?
Ø   unknown or ambiguous subcommand "?": should be circle, or square
    compute area circle 4
⇒   50.2654832
    compute vol sphere 2
⇒   33.5103221333
```

10.6.3　控制集合命令的设置

namespace ensemble 命令的 create 子命令可获取很多选项及其设置值，大多数选项也可以由 namespace ensemble configure 子命令控制。可以使用-subcommands 选项明确定义子命令列表。如果要定义的集合命令的子命令并不是该命名空间所有公开导出的命令的集合时，这一功能十分有用，在使用全局命名空间时这种情况特别常见。您可以设置-prefix 选项，指定无歧义的命令词头是否可以使用(默认值为真，表示可以使用)；还可以使用-command 选项设置由 namespace ensemble create 子命令创建的命令是什么。如果有-namespace 选项，集合命令绑定到的命名空间就是可读的，不过这个选项不能修改设置，而是在创建时自动确定的。

还有另外两个配置选项：-map 和-unknown。-map 选项包含一个字典，把子命令名映射到一个单词列表。每一个列表给出命令的前两个单词的替换值(即集合命令的名称和子命令的名称)，而整个命令是通过执行相对应的子命令来实现。这提供了把更多的参数传给子命令实现的重写方法，这一点非常有用。

```
proc raisePower {power value} {
    return "$value ^ $power *\
        [expr {$value ** $power}]"
}
namespace ensemble create -command power -map {
    square  {raisePower 2}
    cube    {raisePower 3}
    sqroot  {raisePower 0.5}
    invert  {raisePower -1}
}
power cube 4.0
⇒ 4.0 ^ 3 = 64.0
power sqroot 49
⇒ 49 ^ 0.5 = 7.0
power inv 0.0625
⇒ 0.0625 ^ -1 = 16.0
```

如您所看到的，-map 选项可以只写很少的代码，实现相当复杂的行为。

10.6.4　管理集合 unknown 子命令

选项-unknown 是单词的一个列表，当指定的子命令不能解析为集合命令中存在的子命令时，这些单词就组成被调用的命令前缀。调用 unknown 子命令时，传给集合命令的所有参数(包括集合命令自身的名称)都添加到命令前缀之后(会被适当地括起来)，然后执行所得到的命令。如果这时返回的是空字符串，集合命令会再解析子命令名称，将其发给现在实现子命令的命令(即集合命令的-map 和-subcommand 选项至少有一个更新了 unknown 处理程序)或者产生一个错误。如果返回值非空，它包含了运行子命令所需的完整参数列表(包括命令名本身)，可以进行一次映射。

当对 Tcl 内核命令添加功能时-unknown 选项十分有用，它允许您这样做而不必列出 Tcl 内核命令支持的所有子命令的列表。例如，string 命令没有反转字符串的子命令，但是如下添加一个子命令也很容易。

```
rename string strCore
proc strReverse {string} {
    set result {}
    for {set i [strCore length $string]} {$i>0} {} {
        incr i -1
    append result [strCore index $i]
    }
    return $result
}
proc unknownStrCmd {string subcommand args} {
    puts "passing $subcommand through to strCore"
   return [list strCore $subcommand {expand}$args]
}
namespace ensemble create -command string -map {
    reverse {strReverse}
    repeat  {strCore repeat}
    replace {strCore replace}
}   -unknown unknownStrCmd
```

unknown 处理程序(unknownStrCmd)所做的就是写一个消息，把子命令传给(重命名过的)内核命令 string。注意 repeat 和 replace 子命令已经明确地加入集合命令的子命令映射，因此无歧义的子命令词头简写可以正常工作。

```
    string reverse abc
⇒   passing length through to strCore
    passing index through to strCore
    passing index through to strCore
    passing index through to strCore
    cba
```

从实际应用的角度考虑，让 unknown 处理程序更新集合子命令映射更好，示例如下：

```
    proc unknownStrCmd {string subcommand args} {
        catch {strCore $subcommand} msg
        if {![string match {*: must be *} $msg]} {
            puts "adding $subcommand to ensemble"
            set map [namespace ensemble \
                    configure $string -map]
            dict set map $subcommand [list \
                    strCore $subcommand]
            namespace ensemble configure $string -map $map
        }
        puts "passing $subcommand through to strCore"
        return [list strCore $subcommand {expand}$args]
    }
    string reverse abcdef
⇒   adding length to ensemble
    passing length through to strCore
    adding index to ensemble
    passing index through to strCore
    fedcba
```

注意有关 length 和 index 子命令的消息仅在它们第一次被集合命令使用时输出。在那以后，集合命令就保持着一个动态创建的缓冲区存放着映射关系。

10.7 访问其他命名空间的变量

有时候需要访问其他命名空间的变量，而不仅是当前命名空间，有可能是因为变量表示着一些共享状态，也可能是因为它有特殊的属性。这种访问有多种方法实现。

第一，可以使用该变量的完整限定名称。这种方法主要用于从代码中某处访问变量时，或变量是特殊变量，如::env 数组时。

第二，可以把该变量的完全限定名称提供给 global、upvar 或 variable，将变量导入当前命名空间。这种方法常用于要访问的变量数量有限且名称固定时。

第三，可以使用 namespace upvar 将一个命名空间中的一组变量导入当前命名空间。当命名空间代表某些设置的上下文环境，要访问的变量的名称中又含有变量时特别有用。

```
proc blob {name a b c} {
    namespace eval $name {}
    namespace upvar $name a va b vb c vc
    set va $a
    set vb $b
    set vc $c
    namespace ensemble create -map [list \
        set [list ::blobSet $name] \
        sum [list ::blobSum $name] \
        end [list ::blobEnd $name] \
        } -command $name
    }
    proc blobSet {ns var value} {
        # more elegant than: upvar #0 ${ns}::${var} v
        namespace upvar $ns $var v
        set v $value
        return
    }
    proc blobSum {ns} {
        namespace upvar $ns a va b vb c vc
        return [expr {$va + $vb + $vc}]
    }
    proc blobEnd {ns} {
        rename $ns {}
        namespace delete $ns
    }
    # Now for some example usage…
blob example 1 2 3
⇒ ::example
example set a 4
example sum
⇒ 9
example set b 5
example sum
⇒ 12
example end
```

10.8 名称解析路径的控制

在一个命名空间中调用命令时，是根据命令名，由一个称为解析的过程查找命令的。以命名空间分隔符::开始的名称总是在全局命名空间中进行查找，而含有命名空间分隔符但不是以分隔符开头的名称，会首先作为相对于当前命名空间的名称进行查找，如果找不到，再作为相对于全局命名空间的名称进行查找。

对于不包含命名空间分隔符的名称，使用命名空间的解析路径进行解析。这个路径是由 namespace path 命令控制的命名空间列表，而这个列表的元素总是以当前命名空间开头，以全局命名空间结尾。查找命令时依次检查路径列表中的各个命名空间，查找命令的定义。namespace path 命令将路径设定为给定的列表，如果没有给定列表，就返回当前的路径。

下面这个示例，就使用了一个服务命名空间，允许使用 puts 命令来生成结果。这使得从交互式测试到部署都十分简单。

```
namespace eval ::collector {
proc collect {command args} {
    variable accumulator {}
    set ns [namespace parent \
            [namespace origin $command]]
    namespace eval $ns {
        namespace path ::collector
    }
    $command {expand}$args
    namespace eval $ns {
        namespace path {}
    }
    append accumulator "\[generated \
        [string length $accumulator] \
            characters\]"
    return [string trimright $accumulator \n]
}
proc puts {args} {
    if {[llength $args] ==1} {
        variable accumulator
        append accumulator [lindex $args 0] \n
            return
        }
        ::puts {expand}$args
    }
}
namespace eval ::example {
    proc makeMessage {name age} {
    puts "Hello $name"
        puts "You are $age years old"
    }
}
collector::collect ::example::makeMessage \
 "Joe Bloggs" "37 and three quarters"
⇒ Hello Joe Bloggs
 You are 37 and three quarters years old
 [generated 57 characters]
```

namespace path 命令也可用来把命名空间系统做得很类似于对象和类。

```
set counter 0
proc new {class} {
    global counter
    set cmd ::obj[incr counter]
    namespace eval $cmd [list namespace path ::$class]
    proc $cmd {args} "namespace eval $cmd \$args"
    return $cmd
}
```

这会为每一个命令创建一个新的命名空间，设置命名空间解析顺序为先使用类命名空间，再使用全局命名空间。然后创建一个在命名空间中处理参数的命令，其效果是在对象命名空间中没有命令的情况下解析了类命名空间中的命令。类方法的过程可以使用 upvar 1 命令来访问对象命名空间；使用 upvar 2 命令可以到达原始调用者的栈帧，而类的命名空间可以使用 variable 命令访问。

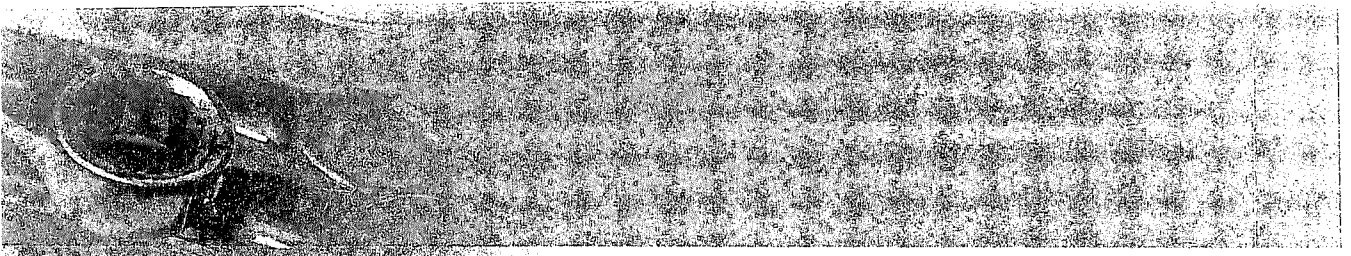
```
    namespace eval example {
        proc foo {args} {puts foo:$args}
        proc bar {args} {puts bar:$args}
        proc set {var args} {
            upvar 1 $var v
            set v {expand}$args
        }
        proc setClass {var args} {
            variable $var
            set $var {expand}$args
        }
    }
    set o1 [new example]
⇒   ::obj1
    $o1 foo a b c
⇒   foo:a b c
```

这里处理对象中的变量是很容易的，只需以对象名为前缀，使用普通 Tcl 命令即可。

```
    $o1 set x 0
⇒   0
```

还应该注意到前面的变量是在对象的命名空间而非类的命名空间中创建的。类中的变量除非明确地进行同步，否则是没有关联的。

```
    incr ::o1::x
    $o1 set x
⇒   1
    set ::example::x "In the class!"
    $o1 set x
⇒   1
    $o1 setClass x
⇒   In the class!
```



第11章 访问文件

本章讲述 Tcl 用于处理文件和目录的命令。这些命令用于顺序地或随机地读写文件。它们还用于取得系统保管的关于文件的信息，例如最后一次访问的时间。您可以重命名、复制以及删除文件。最后，这些命令可以用于操作文件名，例如，可以移除一个文件的扩展名，或查找所有与指定模式匹配的文件名。

11.1 本章出现的命令

本章讲述的文件和目录操作命令如下。这些命令中的大多数也可以应用于进程间的管道和网络套接字，第 12 章将对此进行讨论。在 Tcl 术语中，文件、管道以及网络套接字都称为通道。所有与输入/输出相关的命令都可应用于任何通道。

- `cd ?dirName?`
  将当前工作目录改为 *dirName*，如果没有给出 *dirName* 则转到 home 目录(由环境变量 HOME 给出)。此命令返回一个空字符串。
- `glob ?option ...? ?--? pattern ?pattern ...?`
  返回所有文件中名称与 *pattern* 参数(特殊字符?、*、[]、{}和\)匹配的文件名列表。可选的--参数表示选项结束。可用选项的更多信息详见参考文档。
- `pwd`
  返回当前工作目录的完整路径。
- `chan close channelID`
  `close channelID`
  关闭由 *channelID* 指定的通道。返回一个空字符串。Tcl 8.5 引入的 chan chose 命令，只是把 close 功能整合进 chan 集合命令。
- `chan eof channelID`
  `eof channelID`
  如果最近一次对 *channelID* 通道的读操作遇到了文件结束符，返回 1，否则返回 0。从 Tcl 8.5 开始，建议使用 chan eof。
- `file option name ?arg arg ...?`
  根据 *option* 的设定，对名为 *name* 的文件，或 *name* 所指代的文件，进行一个或多个操作。更多信息详见本章内容及参考文档。

- `chan flush channelID`
  `flush channelID`
  将为 *channelID* 通道生成的输出缓冲区中的内容输出。返回一个空字符串。从 Tcl 8.5 开始，建议使用 chan flush。
- `chan gets channelID ?varName?`
  `gets channelID ?varName?`
  从 *channelID* 读取下一行，抛弃它结尾的行结束符。如果给定了 *varName*，将这一行设为变量的值，返回这一行的字符个数(如果已经是文件尾，则返回-1)。如果没有给定 *varName*，返回读取到的这一行字符(如果已经是文件尾，则返回一个空字符串)。Tcl 8.5 引入的 chan gets 命令，只是把 gets 的功能整合进 chan 集合命令。
- `open name ?access?`
  以 *access* 指定的模式打开名为 *name* 的文件。*access* 可以是 r、r+、w、w+、a 或 a+，或 RDONLY 这样的标志列表，其默认值为 r。返回的文件描述符可用于其他的命令，如 gets 和 close。如果 *name* 的开头字符是|，命令打开的将不是文件而是管道(更多信息参见 12.4 节)。
- `chan puts ?-nonewline? ?channelID? string`
  `puts ?-nonewline? ?channelID? string`
  将 *string* 写入 *channelID*，如果不指定-*nonewline*，就添加一个换行符。*channelID* 默认为 stdout。返回一个空字符串。Tcl 8.5 引入的 chan puts 命令，只是把 puts 的功能整合进 chan 集合命令。
- `chan read ?-nonewline? channelID`
  `chan read channelID numChars`
  `read ?-nonewline? channelID`
  `read channelID numChars`
  从 *channelID* 中读取接下来的 *numChars* 个字符并返回(如果文件中余下的字符不足 *numChars* 个，就一直读到文件尾)。如果没有给出 *numChars*，就读取并返回文件中余下的所有字符；如果指定-nonewline，如果最后是一个换行符，会把换行符抛弃。Tcl 8.5 引入的 chan read 命令，只是把 read 的功能整合进 chan 集合命令。
- `chan seek channelID offset ?origin?`
  `seek channelID offset ?origin?`
  在 *channelID* 中定位，下一次访问从 *origin* 开始偏移 *offset* 个字节处开始。*origin* 可设置为 start、current 或 end，默认为 start。返回一个空字符串。从 Tcl 8.5 开始，建议使用 chan seek 命令。
- `chan tell channelID`
  `tell channelID`
  返回 *channelID* 中当前访问位置距文件头的偏移字节数。从 Tcl 8.5 开始，建议使用 chan tell 命令。

- `chan configure channelID ?optionName? ?value?`
  `?optionName value ...?`
  `fconfigure channelID ?optionName? ?value?`
  `?optionName value ...?`
  查看或设定 channelID 通道的配置。如果没有提供 *optionName* 及 *value* 参数，该命令返回一个字典，内容是所有的设置选项和它们对应的值。如果提供 *optionName* 而没有提供 *value*，这个命令返回指定的选项的当前设置值。如果指定一对或多对 *optionName* 和 *value*，该命令将指定的选项设为指定的值；这种情况下返回值是一个空的字符串。从 Tcl 8.5 开始，建议使用 chan configure 命令。
- `chan copy inputChan outputChan ?-size size?`
  `?-command callback?`
  `fcopy inputChan outputChan ?-size size?`
  `?-command callback?`
  从 *inputChan* 读取数据，该通道必须已经为读取打开，数据读取到 *outputChan*，该通道必须已经为写操作打开。chan copy 命令从 *inputChan* 转移数据，直到遇见文件尾或是转移的字节数达到 *size*(如果指定 *size*)。默认地，这个命令在转移数据时会阻塞前台，它的返回值是写入 *outputChan* 中的字节数。如果指定-command 选项，chan copy 会立即返回，当复制完成后，再调用 *callback*，传递一或两个其他参数表示写入 *outputChan* 的数据的字节数。如果在后台复制时发生了错误，第二个参数就是与错误有关的错误字符串。从 Tcl 8.5 开始，建议使用 chan copy 命令。

11.2 操纵文件和目录名

每一个文件都有一个文件名。Tcl 中指定文件名时遵循普通的 Unix 语法。例如，文件名 x/y/z 的意思就是在一个名为 y 的目录中有一个名为 z 的文件，而这个 y 目录又处在名为 x 的目录中，这个 x 目录又必须在当前工作目录中。文件名/top 指的是根目录下的 top 文件。您还可以用~表示用户的 home 文件夹。例如，名称~ouster/mbox 指的是用户 ouster 的 home 文件夹中的名为 mbox 的文件，而~/mbox 指的是正在运行 Tcl 脚本的用户的 home 文件夹中的 mbox 文件。这些约定(包括~符号)适用于所有 Tcl 命令所用的文件名参数。

Windows 系统与 Unix 的这些约定有所不同。Windows 系统用反斜线分隔目标，而不是正斜线。反斜线\在 Tcl 和其他基于 C 程序约定的语言中可能会造成问题。因为 Tcl 的替换操作，您需要在反斜线后面再接一个反斜线才能让路径被正确解析(如\\apps\\tcl)，或者避免对路径进行替换操作(如{\apps\tcl})。有网络共享的时候情况就变得更加复杂，这时在转义序列被替换之后还需要有两个反斜线，就会出现这样的写法：\\\\host\\share。另外，Windows 的磁盘驱动器并没有挂载在一个根目录下(在 Unix、Linux 以及 Mac Os X 系统中都是这样挂载的)。Windows 使用驱动盘符(如 C：)来表示相应的磁盘驱动器。

要处理文件命名的差异情况，可以使用 Tcl 的 file join 和 file split 命令把文件名和目录

名以跨平台的方式适当地组合起来。file 是一个有很多选项的通用命令，可用于操纵文件名以及取得文件的信息。本节介绍的命令就是对文件的名称进行操作。它们不会进行系统调用，也不会检查那些文件名是否与实际的文件对应。

考虑下面这个示例，它生成一个文件相对于用户的 home 目录的全路径名。

```
    set env(HOME)
⇒ C:\Documents and Settings\Owner
    file join $env(HOME) lib mylib.tcl
⇒ C:/Documents and Settings/Owner/lib/mylib.tcl
```

注意，无论各个路径部分的原始样式如何，file join 命令返回的都是 Unix 样式的路径，以/符号作为目录分隔符。可以在访问文件系统的 Tcl 命令中使用这样的“Tcl 原生”路径样式，也可以使用特定平台的目录分隔符。另外，如果路径中有一个部分是从根层次开始的，例如以一个/符号开头，那么 file join 命令会抛弃它前面的所有部分。

提示：使用 file join 命令可以让脚本不因此而受限于某种平台上，如 Unix 或 Windows。

注意，某些目录名可能含有空格。Tcl 命令把空格作为分隔符，因此需要把这些参数正确地组织起来，再传给 file join 命令，例如：

```
    file join c:/ {Program Files} Tcl tclsh.exe
⇒ c:/Program Files/Tcl/tclsh.exe
```

使用 file split 命令将一个文件名路径分割为它的各组成部分，例如：

```
    file split x/y/z
⇒ x y z
```

file nativename 命令返回原生格式的文件名。在调用 exec 命令时这一点特别有用，第 12 章将讨论这个问题。在将路径写入磁盘或将路径呈现给用户时，这一命令也很有用。在使用 exec 命令时，应该按照需要的格式或原生格式传送文件名称。例如，file nativename 命令将~符号展开为指定用户的 home 目录。

```
    file nativename ~ericfj/scripts/coolscript.tcl
⇒ /Users/ericfj/scripts/coolscript.tcl
```

file dirname 移除文件名中的最后一个部分，表面上看这个命令的功能就是得到包含该文件的目录的名称。

```
    file dirname /a/b/c
⇒ /a/b
    file dirname main.c
⇒ .
```

然而，file 命令并不会检查是否真的存在这样一个文件，存在这样一个目录。

file extension 返回一个文件的扩展名(从名称中最后一个.符号开始的部分)，如果文件没有扩展名，则返回空字符串。

```
    file extension src/main.c
⇒ .c
```

file rootname 返回文件名中除扩展名之外的部分。

```
    file rootname src/main.c
```

```
⇒ src/main
  file rootname foo
⇒ foo
```

file tail 返回文件名的最后一个部分(即在其目录下的文件本身的名称)。

```
  file tail /a/b/c
⇒ c
  file tail foo.txt
⇒ foo.txt
```

file normalize 命令返回独一无二的标准化路径，代表该文件或目录，这个返回的路径字符串可以作为文件或目录独一无二的指示符。标准化的路径是指不含./和../的绝对路径。另外，在 Unix 和 Mac OS 中，构成标准化路径的各部分不能是符号链接/别名，在 Windows 系统中会根据范例依存返回长格式，例如：

```
  file normalize ~/Documents/../tcltk/intro/examples
⇒ /Users/ken/tcltk/intro/examples
```

file pathtype 命令告诉您关于一个文件名的信息，如它是一个绝对路径还是相对路径。这个命令的返回值为 absolute、relative 或 volumerelative，取决于传递给命令的文件名类型。相对文件名是相对于当前工作目录的路径。类似地，volumerelative 表示该文件名与一个指定的挂载卷或磁盘相对的路径。在 Windows 系统中会发现很多的 volumerelative 文件名。下面的示例展示了 file pathtype 的应用。

```
  file pathtype foo
⇒ relative
  file pathtype [pwd]
⇒ absolute
```

最后，file volumes 命令列出所有已挂载的卷。在 Unix 和 Mac OS X 系统中，它应该返回/代表根目录。在 Windows 系统中，它会返回所有已挂载的驱动器的 Tcl 版描述，如 c:/等。

11.3 当前工作目录

在 Tcl 中，提供的文件名称可以用绝对路径，也可以是相对于当前工作目录的相对路径。因此，知道当前工作目录是什么通常是十分重要的。

Tcl 提供两个命令帮助管理当前工作目录：pwd 和 cd。pwd 不获取参数，返回当前工作目录的完整路径。cd 获取一个参数，将当前工作目录转为该参数值所指的目录。如果不带参数地调用 cd，它会把当前工作目录转为运行 Tcl 脚本的用户的 home 目录(cd 使用环境变量 HOME 的值作为 home 目录的路径名)。

11.4 列出目录的内容

glob 命令获取一个或更多模式参数，返回与这些模式匹配的文件名列表。

```
glob *.c *.h
```

```
⇒ main.c hash.c hash.h
```

glob 使用 string match 命令的匹配规则(参见第 5 章)。前面这个示例中，glob 返回当前目录下以.c 或.h 结尾的文件名列表。glob 还允许模式参数包含逗号分隔符，在大括号中用逗号分隔符分隔的多个模式为或的关系，如下例所示：

```
    glob {{src,backup}/*.[ch]}
⇒ src/main.c src/hash.c src/hash.h backup/hash.c
```

glob 对待这个模式的方式就如同这是每个模式中包含一个字符串的多个模式一样，如下例所示：

```
glob {src/*.[ch]} {backup/*.[ch]}
```

提示： 这些示例中额外的大括号用来避免模式字符串中的大括号触发命令替换。它们会在 glob 命令的过程开始以前由 Tcl 解析器移除。

如果一个 glob 模式以斜线结尾，那么它只与目录名匹配。例如，以下命令返回的是当前目录下的所有子目录名的列表。

```
glob */
```

可以用-type 选项指定 glob 命令只返回特定类型的文件。-type 的参数可以是一个或多个下列参数组成的列表：b 代表块设备，c 代表字符设备，d 代表目录，f 代表无格式文件(即不是目录)，l(不是数字 1)代表符号链接，p 代表命名管道，s 代表套接字。您还可以为文件访问授权提供标志——r 表示读取授权，w 表示写入授权，x 表示执行授权——以及 hidden 表示隐藏文件，readonly 表示只读授权。可以联合使用一个或多个这些条目，而 glob 返回的文件需要与指定的至少一种文件类型，与指定的所有文件授权项相匹配。例如，要查找您拥有读授权和写授权的所有文件(不包括目录)，可以使用如下命令：

```
    glob -types { f r w } *
⇒ b.txt c.txt
```

如果只需要指定一个类型条件，可以使用简单一点的语法。

```
    glob -types f *
⇒ b.txt c.txt
```

-directory 选项告诉 glob 从指定目录开始搜索，例如：

```
    glob -directory /usr/local -types d *
⇒ /usr/local/bin /usr/local/lib
```

这种情况下，传递给 glob 的*在指定的目录(即/usr/local 目录)中被解释。当目录名包含在 glob 模式中会被作为通配符或转义序列的字符时，可以使用这个选项。

-path 选项告诉 glob 命令搜索名称以-path 参数值开头的那些文件。当要搜索的文件名可能干扰 glob 命令时这一选项特别有用。不能同时使用-path 选项和-directory 选项。这两个选项主要的不同在于-directory 告诉 glob 从指定的目录开始搜索，而-path 告诉 glob 要搜索的名称有指定的词头。传给-path 的值可以包含目录路径名以及文件名的词头。-directory 选项只接受目录名称。

-tails 选项让 glob 命令只返回文件名中-directory 或-path 选项值后面的部分。例如，对比下面的示例：

```
    glob -directory /usr/local -types d *
⇒ /usr/local/bin /usr/local/lib
    glob -directory /usr/local -types d -tails *
⇒ bin lib
```

特殊选项-join 告诉 glob 把所有余下的选项作为一个大的模式进行处理，使用目录分隔符将各元素串接起来。其基本工作与 file join 命令相似，但将其结果用作 glob 的路径。

默认情况下，glob 在没有与您提供的模式相匹配的文件时产生错误。您可以为 glob 指定-nocomplain 选项制止这个错误产生，让它在没有匹配文件时只是返回一个空的字符串。

11.5 处理磁盘上的文件

Tcl 提供了很多命令对磁盘上的文件进行操作，如移动、重命名、复制以及删除等。这些命令直接调用操作系统函数来实现它们的功能，因此它们是高效率的，也是与平台无关的。您的脚本应该总是使用这些 Tcl 命令来执行这些操作，而不是使用 exec 来执行平台特有的命令。

11.5.1 创建目录

使用 file mkdir 命令来创建新的目录，这是“make directory”的缩写(Unix 和 DOS 命令也是这样命名的)。例如：

```
    file mkdir foo
    cd foo
    pwd
⇒ /Users/ericfj/foo
```

11.5.2 删除文件

使用 file delete 命令删除文件。

```
file delete a.txt
```

必须将每个文件作为独立的参数列出，例如：

```
file delete a.txt b.txt
```

这个命令删除了两个文件。file delete 命令不执行通配符展开，因此，下面这条命令只删除一个名为*.tmp 的文件。

```
file delete *.tmp
```

处理上面这种情况的正确方法是先使用 glob 命令返回与指定模式匹配的文件列表，然后使用 Tcl 参数展开语法把列表元素作为独立参数提供给 file delete。

```
file delete {*}[glob *.tmp]
```

您可以使用 file delete 删除目录，只要把目录名作为参数提供即可。然而，如果目录内容非空，file delete 就会产生一个错误。

```
    file delete foo
Ø   error deleting "foo": directory not empty
```

使用-force 选项可以删除一个非空的目录。指定这个选项后，会递归地删除指定目录下的所有目录和文件，因此，最好谨慎使用这个选项。

11.5.3 复制文件

当编写 Tcl 过程读取 a 文件的内容并把这些内容写到另一个文件中时，使用 Tcl file copy 命令来复制文件要简单得多。file copy 命令把源文件复制到目标文件，例如：

```
    glob *.txt
⇒   a.txt
    file copy a.txt b.txt
    glob *.txt
⇒   a.txt b.txt
```

file copy 命令把文件 a.txt 复制到 b.txt。glob 命令校验新的文件 b.txt 已经存在。

如果目标文件已经存在，file copy 命令会产生一个错误，如下例所示：

```
    file copy a.txt b.txt
Ø   error copying "a.txt" to "b.txt": file already exists
```

可以用-force 选项让 file copy 命令覆盖已经存在的目标文件。

```
file copy -force a.txt b.txt
```

还可以将多个文件复制到一个目标目录中。例如：

```
file copy a.txt b.txt Documents
```

此命令将两个文件 a.txt 和 b.txt 复制到目标目录 Documents 中。这种情况下，目标必须是一个目录，而不是一个文件。

11.5.4 重命名和移动文件

使用 file rename 命令给一个文件或目录指定新的名称。需要提供原文件或目录的名称以及新的名称，例如：

```
file rename b.txt c.txt
```

此命令将 b.txt 文件重命名为 c.txt。如果名为目标名称的文件已经存在，需要指定-force 选项让 file rename 将已经存在的文件覆盖；否则它会产生一个错误。

如果目标名称指向了另一个目录，file rename 命令就会把文件移动到新的位置。还可以把一个目录指定为源，从而重命名或移动该目录。

11.5.5 文件信息命令

除了已经讨论过的选项，file 命令提供了很多用于取得文件相关信息的选项。除了 stat 和 lstat 外，这些选项都有相同的格式。

```
file option name
```

这里 *option* 指明了需要的信息，如 exists 或 readable 或 size，而 *name* 是指定的文件名。

选项 exists、isfile、isdirectory 以及 type 返回有关文件性质的信息。如果指定的文件存在，exists 返回 1，如果文件不存在，或当前用户无权搜索存放它的目录，则返回 0。如果指定的名称指向的是普通磁盘文件，则 isfile 返回 1，如果指向的是目录或设备文件等其他类型的文件，则返回 0。如果指定的名称指向的是目录，则 isdirectory 返回 1，否则返回 0。file type 返回一个字符串，如 file、directory 或 socket，给出文件的类型信息。

readable、writable 以及 executable 选项在指定文件存在并且当前用户有执行指定操作的权限时返回 1，否则返回 0。如果当前用户是文件的所有者，owned 选项返回 1，否则返回 0。

size 选项返回一个十进制字符串，以字节为单位给出指定文件的大小。filemtime 返回文件最后一次被修改的时间。返回的时间值是标准 POSIX 格式的时间，即从 UTC 时间 1970 年 1 月 1 日零时零分零秒开始[该时间也被称为新纪元时间(epoch time)]，到当前时间的秒数。atime 选项与 mtime 相似，不过它返回的是文件最后一次被访问的时间。有关 Tcl 中对新纪元时间的操作的更多信息详见第 15 章。

stat 选项提供了一种简单的方法，可以一次得到文件的各种信息。使用 stat 获取信息比多次使用 file 命令获得各项信息要快得多。file stat 还提供了一些用其他命令不能取得的信息。它要获取两个额外的参数，分别是文件名和一个变量名，示例如下：

```
file stat main.c info
```

这种情况下，文件名是 main.c，变量名是 info。这个变量会被作为数组对待，其元素见表 11.1，每一元素都是十进制字符串。

表 11.1 file stat 命令返回数组的元素

关 键 字	值
atime	最后一次被访问的时间
ctime	最后一次改变状态的时间
dev	包含文件的设备的识别符
gid	文件的群组识别符
ino	文件在设备中的序列号
mode	文件模式位

续表

关 键 字	值
mtime	最后一次被修改的时间
nlink	链接到文件的链接数量
size	文件大小，单位为字节
uid	拥有文件的用户标识

atime、mtime 以及 size 元素的值与前面讨论过的相应 file 选项提供的值相同。其他元素的信息来自系统调用 stat；这些与文件系统相关的信息都是直接从 stat 返回的结构体的对应域中取得的。

lstat 和 readlink 选项用于对符号链接的处理，而且它们仅能在支持符号链接的系统上运行。file lstat 对于普通文件来说，与 file stat 相同；但应用于符号链接时，它返回的是符号链接本身的信息，而 file stat 返回的是符号链接指向的文件的信息。file readlink 返回符号链接的内容，即它指向的文件的名称；它只能应用于符号链接。对于所有其他的 file 命令，如果给定的文件名指定的是一个符号链接，它们都会对链接指向的文件进行操作，而非链接本身。

11.5.6 处理名称怪异的文件

Tcl 用短横线开头的单词表示这是为命令提供的选项。例如，file copy 命令可以接受-force 选项。然而，如果要处理的文件的名称也以短横线开头，就会遇到问题。因为 Tcl 命令会认为以短横线开头的单词是一个选项，而不是文件名。

要处理这样的情况，就在命令之后，文件名之前，写入双短横线--。例如，要复制一个名为-force 的文件，可以使用如下命令：

```
file copy -force -- -force b.txt
```

此命令指定了强制复制选项-force，然后用--把选项部分与要复制的文件名分开。

11.6 读写文件

Tcl 作为脚本语言，在其历史上十分强调对文本文件的处理。Tcl 最初是在 Unix 系统上创建的，该系统中大多数信息，包括应用程序数据以及系统配置设定，都存放在普通文本文件中。因此，大多数 Tcl 文件命令假定所处理的文件是文本文件。不过，Tcl 也提供了对二进制文件进行读、写以及数据修改的能力。

11.6.1 基本文件 I/O

用于文件 I/O 的 Tcl 命令与 C 的标准 I/O 库相似，它们的名称及行为都是相似的。下面这段调用 tgrep 的脚本演示了文件 I/O 的基本功能。

```
#!/usr/bin/env tclsh
if {$argc != 2} {
```

```
    error "Usage: tgrep pattern fileName"
}
set f [open [lindex $argv 1] r]
set pat [lindex $argv 0]
while {[gets $f line] >= 0} {
    if {[regexp -- $pat $line]} {
        puts $line
    }
}
close $f
```

这段脚本的行为与 Unix grep 程序十分相似，在文件中搜索与给定模式匹配的文本。您可以从命令外壳中调用它，为它传入两个参数(一个正则表达式模式和一个文件名)，然后它会输出指定文件中与指定模式匹配的那些行。

在确认它接收到足够多的参数之后，脚本调用 open 命令打开要搜索的文件，文件名就是传给脚本的第二个参数。open 要获取两个参数：文件名和访问模式。访问模式提供的信息是您要读还是写这个文件，想在它尾部添加一些内容，还是从文件头开始访问。访问模式可以是下列各值之一。

- r——只读。指定文件必须存在。如果不指定访问模式，这是默认设置。
- r+——可读写。指定文件必须存在。
- w——只写。如果文件存在，将其清空(删除文件中的全部内容)；若文件不存在，则创建一个新的空文件。
- w+——可读写。如果文件存在，将其清空(删除文件中的全部内容)；若文件不存在，则创建一个新的空文件。
- a——只写。将初始访问位置设为文件尾。因此，写入的内容直接添加到文件原有内容后，除非重新设置了访问位置。如果文件不存在，则创建一个新的空文件。
- a+——可读写。将初始访问位置设为文件尾。如果文件不存在，则创建一个新的空白文件。

访问模式也可以用 POSIX 标志的列表来设置，如 RDONLY、CREAT 以及 TRUNC。有关这些标志的更多信息参见参考文档。

open 命令返回如 file3 这样的字符串，作为打开的文件的描述符。这个文件描述符由其他的命令调用，用于对打开的文件进行操作，如 gets、puts、close 等。一般来说在打开文件时就把文件描述符存放到一个变量中，然后使用这个变量指向打开的文件。您不应当期待 open 返回的文件描述符有什么特定的格式。

有三个文件描述符有定义好的名称，并且总是可用，即使没有明确地打开任何文件。它们是 stdin、stdout 以及 stderr，分别指向标准输入、标准输出以及运行 Tcl 脚本的进程的错误信息通道。

提示： *在 Windows 中，这些默认通道仅为控制台应用程序工作，而非窗口应用程序。*

---

file channels 命令列出所有的通道。例如，除非明确地打开了另外的通道，否则这条命令返回的就只有标准 I/O 通道。

```
    file channels
⇒ stdin stdout stderr
```

在打开待搜索的文件之后，tgrep 脚本用 gets 命令一次一行地读取文件内容。gets 通常要获取两个参数：一个文件描述符和一个变量名。它读取打开的文件的下一行，抛弃行尾的一个或多个连续的行结束符，将这一行的内容存放到指定变量中，返回存放到变量中的字符个数，字符个数不包括行结束符。如果在读取到任何字符之前就遇到文件尾，gets 就存放一个空字符串，返回-1。

提示： Tcl 还提供了第二种使用 gets 的形式，读入的行将作为命令的返回值，而 read 命令可以用于非面向行的输入。

tgrep 脚本把文件中的每一行与指定模式进行比较，用 puts 输出匹配结果。puts 命令获取两个参数，分别是文件描述符和要输出的字符串。puts 为字符串添加一个行结束符，然后将这一行输出到指定文件。这个脚本没有指定用于输出的文件描述符，因此使用了默认的 stdout，这一行会被送到标准输出端。

当 tgrep 到达文件尾时，gets 返回-1，终止 while 循环。然后脚本用 close 命令关闭文件，释放与打开文件相关的资源。大多数系统中，一个应用程序可以同时打开的文件数有一定的限制，因此在完成读写操作后应该尽快关闭文件。这个示例中使用 close 并不必要，因为随着应用程序退出文件会自动关闭，但这是一种很好的应用习惯。

11.6.2 输出缓冲区

puts 命令使用 C 标准 I/O 库的缓冲方案。这意味着传递给 puts 的信息可能不会立即在目标文件中出现。很多情况下(特别是文件不是终端设备的情况下)，输出信息缓冲存放在应用程序的内存中，直到为该文件输出的数据累积到一定数量，才通过一次写操作把这些数据真正写入文件；很多操作系统的缓冲区大小设置为 4KB。(缓冲选项的更多信息参见 12 章。)如果需要数据立即出现在文件中，应该调用 flush 命令。

```
flush $f
```

flush 命令获得一个文件描述符作为参数，把缓冲区中待写入该文件的数据写入文件。直到数据写入完成，flush 才返回。在文件关闭时，缓冲区中的数据也会这样写入文件。

11.6.3 处理各平台的行结束约定

在 Unix 和 Linux 系统中，文本文件的每一行都以换行符结尾(ASCII 码十进制的 10)。旧的 Macintosh 系统(Mac OS X 之前)使用单个回车符(ASCII 码十进制的 13)；现代的 Macintosh 系统与 Unix 的约定相同。而 Windows 系统使用两个字符：一个回车符紧接一个换行符。仅仅是快速地访问不同操作系统中的文本文件也不那么简单。

大多数情况下，Tcl 会自动地处理这些不同。当向一个文件中写入内容时，puts 命令会根据平台自动地将换行符转化为适当的字符或字符序列。因此，可以在 Tcl 脚本中总是使用\n 表示一行的结束。

提示： 记得 puts 命令会自动地在它输出的字符串尾部加上行结束符号。可以使用 -nonewline 选项取消添加行结束符。

默认情况下，gets 命令把任一种支持的行结束约定都解释为行的结束。各行的行结束符可以各不相同。read 命令的默认设置也是如此，输入中的任一种行结束约定在返回的字符串中都翻译成行结束符。另外，-nonewline 选项让 read 命令在返回的字符串的最后一个字符(或字符序列)是一个行结束约定时不在尾部再添加行结束符。

可以使用 chan configure 命令(或 Tcl 早期版本的 fconfigure)对任意给定的通道或文件查看或修改关于行结束符的行为，例如：

```
    chan configure stdin -translation
⇒ auto
```

不给出更多的参数时，该命令返回指定通道的行结束符翻译模式。您也可以给出更多的参数，为通道指定翻译模式。如果通道是双向的，单个元素的列表可以设置输入与输出模式。您也可以提供两个元素的列表，第一个元素指定输入的翻译模式，第二个元素指定输出的翻译模式。

默认情况下，通道的默认设置为 auto。对于输入通道，auto 的意思是所有的行结束序列均作为换行符对待；对于输出通道，自动使用平台指定的行结束符号。值 cr、lf 或 crlf 分别表示只使用换行符或回车，或使用回车-换行符序列作为通道的行结束符约定。例如，把一个输出通道的翻译选项设置为 cr，就表示只把换行符作为行结束符对待，读取时将其翻译为 Tcl 使用的换行符。您可以把 translate 设置为 binary，表示不进行行结束符的翻译；把通道的-encoding 选项设置为 binary 也会自动将翻译选项设置为 binary(参见 11.6.4 节)。

利用 Tcl 对行结束符的自动翻译，可以使用 read 命令来实现前面的 tgrep 脚本的功能。

```
#!/usr/bin/env tclsh
if {$argc != 2} {?
    error "Usage: tgrep pattern fileName"
}
set f [open [lindex $argv 1] r]
set pat [lindex $argv 0]
set content [read $f]
close $f
foreach line [split $content "\n"]
    if [regexp -- $pat $line] {
        puts $line
    }
}
```

注意这里使用了 split 命令将 read 返回的字符串在各个换行符处分割。其结果是各行的列表，而 foreach 命令可以对其进行遍历。通过对 read 的使用，这个脚本比前面的 tgrep 脚本效率更高，该命令使用 gets 命令为每行分别访问文件系统。而这个脚本中的 read 一次性地把整个文件的内容加载到内存中，然后检查匹配的模式。对于不是太大的文件，这样做效率要高得多。但这样处理很大的文件可能会占用太多的内存，如果动用了虚拟内存，那速度就会显著下降。

11.6.4 管理字符编码集

如第 5 章所述，Tcl 把所有字符串内部存储为 UTF-8 格式的 Unicode 字符。不过，在对通道进行文本读写时，Tcl 会自动地把编码在内部的 UTF-8 格式与系统格式之间转换。因此，如果正在处理的文本文件使用的是与系统编码相同的格式，不需要进行任何特殊的操作就能让 Tcl 正确处理文件。

另一方面，如果文件使用的编码与系统编码不同(或者希望创建一个编码与系统编码不同的文件)，可以在 chan configure 命令中使用-encoding 选项(或 Tcl 早期版本的 fconfigure)为 Tcl 指定该文件使用的编码，例如：

```
chan configure $fid -encoding shiftjis
```

提示： *在处理二进制文件时要避免 Tcl 进行字符编码翻译，需要用 chan configure 把 -translation 选项设为 binary。这不仅阻止了对平台特定的行结束符的翻译，也自动将通道的-encoding 选项设为 binary，阻止了对字符编码翻译。*

11.6.5 处理二进制文件

Tcl 默认所有的文件都是文本文件。如前文所述，如果操作的是包含二进制信息的文件，应该使用 chan configure 将-translation 设置为 binary。

```
chan configure $fid -translation binary
```

要读取文件，应该再调用 read 命令，指定要读取的数据的字节数。可以使用第 5 章讲述的 binary scan 命令，将二进制信息“解包”到 Tcl 变量中。

```
set data [read $fid 12]
binary scan $data iiss int1 int2 short1 short2
```

相对地，向一个文件中写入二进制信息，需要用 binary format 命令将二进制字符串“打包”到变量当中，然后使用 puts 命令把字符串写入文件。在多数情况下，应该为 puts 命令设置-nonewline 选项，阻止它在输出二进制信息后自动添加换行符。

```
set data [binary format s3 {3 -7 1}]
puts -nonewline $fid $data
```

11.6.6 随机访问文件

文件 I/O 默认为顺序的——每条 gets 或 read 命令返回上一条 gets 或 read 命令后面的字节，每一条 puts 命令将数据写到前一条 puts 命令输出的数据后面。不过，可以使用 chan seek、chan tell 以及 chan eof 命令非顺序地访问文件。(与 Tcl 早期版本中的 seek、tell 和 eof 命令相同。)

每一个打开的文件都有一个访问位置，它指示了文件中下一条读命令或写命令将操作的位置。但一个文件打开时，根据您在 open 中指定的访问模式，访问位置被设置到文件头或文件尾。每一次读或写操作之后，访问位置会根据处理的字节数进行移动。可以使用 chan seek 命令改变当前的访问位置。最简单的 chan seek 形式需要获取两个参数：一个文

件描述符和一个整型的偏移值。例如，命令：

```
chan seek $f 2000
```

就改变了文件的访问位置，下一条读或写命令将从文件的2000个字节处开始。

提示： 偏移值的单位是字节，而不是字符。因此，对于占用多个字节的字符，访问位置可能会被设置到它中间的某个字节上。

chan seek 也可以获取第三个参数，指定偏移的起始点。第三个参数必须是 start、current 或 end 之一。start 的效果和没有指定这个参数一样，偏移值从文件头开始计算。current 的意思是偏移值从文件的当前访问位置开始计算，end 表示偏移值是相对于文件尾的。例如，下面这条命令将访问位置设置为文件尾之前的100个字节。

```
chan seek $f -100 end
```

如果起始点是 current 或 end，偏移值可以是正值或负值；如果是 start，偏移值必须是正值。

提示： 有可能把访问位置指定到当前文件末尾的后面，这种情况下文件可能会含有空洞。这种情况的具体含义需查阅操作系统的相关文档。

chan tell 命令返回指定文件描述符的当前的访问位置(以字节为单位，而不是字符)。

```
chan tell $f
⇒ 186
```

这允许您记录下某个位置，在以后返回这里。

chan eof 命令获取一个文件描述符为参数，如果最近一次对文件的 gets 或 read 操作遇到文件尾，返回1，否则返回0。

```
chan eof $f
⇒ 0
```

在使用非阻塞式通道的大多数情况下，可以使用 chan eof 命令检测文件尾。更多信息参见第12章。

11.6.7 复制文件内容

11.5.3 节讨论了 file copy 命令，它复制磁盘上的一个文件。file copy 命令使用方便，但也有它的限制：它创建的目标文件是源文件逐个字节复制的副本，而且它只能应用于磁盘上的文件。一个应用更广泛的选择是 chan copy 命令(Tcl 较早版本中的 fcopy)，它允许在复制时进行一些改变，例如改变使用的字符编码，并且可应用于任何 Tcl 通道类型，如管道、套接字以及磁盘上的文件。

命令 chan copy 最简单的形式，需要获取一个为读取打开的输入通道，一个为写入打开的输出通道作为参数。从输入通道进行读取，直到遇见文件尾为止，然后把读取到的内容写入输出通道。或者可以使用-size 选项指定从输入通道中进行读取的最大字节数。下面这个示例就复制一个文件的内容，同时把 EUC-JP 编码转化为 Shift_JIS 编码。

```
set input [open $inFile r]
chan config $input -encoding euc-jp
set output [open $outFile w]
chan config $output -encoding shiftjis
chan copy $input $output
close $input
close $output
```

默认情况下，chan copy 命令会进行阻塞，直到复制完成；它的返回值是写入输出通道的数据的字节数。您也可以用-command 选项指定回调，那样 chan copy 会立即返回。复制在后台进行，复制完成后 Tcl 调用指定的回调，为它传入一个或两个参数。第一个参数是写入输出通道的数据的字节数。只在复制中出现错误的情况下才有第二个参数，它包含了错误消息。

提示： 仅在 Tcl 的事件循环运行时后台复制才会进行。如果事件循环并不在运行，需要用 vwait 命令启动它。另外，在后台复制进行时，不应该让任何通道事件处理程序活动，例如由 chan event 或 file event 命令注册的那些。有关事件循环、vwait 命令和通道事件处理程序的更多信息参见第 12 章。

下面的代码展示了一个后台复制的示例。

```
proc copyComplete {args} {
    global enter_loop
    if {[llength $args] > 1} {
        puts "Error during file copy; [lindex $args 1]"
    } else {
        puts -nonewline "Copy completed successfully. "
    }
    puts "[lindex $args 0] bytes copied."
    set enter_loop 1
}
set input [open $inFile r]
chan config $input -encoding euc-jp
set output [open $outFile w]
chan config $output -encoding shiftjis
chan copy $input $output -command copyComplete
vwait enter_loop

# Copy operation has completed at this point

catch {close $input}
catch {close $output}
```

这个 copyComplete 过程在文件复制完成后实现回调。如果只给它传递一个参数，那么复制操作就成功完成；如果提供了不止一个参数，那么在复制过程中出现了错误，这个过程把错误信息发送到控制台。在输入和输出通道打开并完成设置后，chan copy 命令启动了后台复制。vwait 命令启动了事件循环；vwait 命令会等待直到全局变量 enter_loop 被设置(该参数在执行回调脚本 copyComplete 时被设置)，这时事件循环终止，vwait 命令返回。然后关闭两个通道。

11.7 虚拟文件系统

从 Tcl 8.4 开始，Tcl 的文件命令支持虚拟文件系统。这意味着 Tcl 扩展程序可以把增加的服务作为文件系统呈现给 Tcl。例如，对 FTP 网络文件的访问可以呈现给 Tcl 命令 glob 或 file，就如同远端的文件就在本地硬盘中一样，或者访问 ZIP 压缩包中的文件，就如同它们是文件系统中的独立文件一样。使用这样的虚拟文件系统大大简化了 Tcl 脚本的编写。只需要在 Tcl 中使用普通的文件命令，Tcl 扩展包就会完成不同的工作。而且，如果映射的文件系统支持，也可以在挂载的虚拟文件系统中创建新的文件或修改文件，即使它事实上并不是一个文件。一些 Tcl 中最激动人心的开发成果，如 Starkit 和 Starpack，就使用了虚拟文件系统。更多的相关信息参见第 14 章。

tclvfs 扩展包提供了一系列到虚拟文件系统的绑定，可以用 Tcl 命令进行访问。这些虚拟文件系统包括 ZIP 存档、FTP 网络访问、Metakit 文件(由 Starkit 和 Starpack 使用)，以及 WebDAV 和 HTTP 网络协议。不过最常见的应用是把压缩文件像文件系统一样挂载。例如，浏览 ZIP 存档中的内容，或从存档中读取单独的文件。

下面这段代码把一个 ZIP 文件挂载为当前工作目录下的名为 zip 的虚拟目录。

```
    puts {pwd}
⇒ /Users/ericfj/Documents/tcl
    package require vfs::zip
    vfs::zip::Mount tcl852-src.zip zip
```

这个使用 ZIP 文件的 Tcl 8.5 版源程序示例下载自 http://tcl.sourceforge.net；您可以这样使用任何 ZIP 文件。在挂载它以后，就可以像访问普通文件系统一样访问虚拟文件系统。例如，要列出上面这段 Tcl 源程序刚刚挂载的顶层文件，可以使用如下命令：

```
foreach f [glob zip/tcl8.5.2/*] {
    puts $f
}
```

这个示例中，glob 命令返回了目录 zip/tcl8.5.2 下的所有文件。目录名称中的 zip 部分就是我们挂载虚拟文件系统的地方。tcl8.5.2 部分是 ZIP 文件中的顶层目录。

提示： 如果 package require 命令失败，说明需要安装 tclvfs 扩展包。最简单的方法是下载 ActiveTcl 发行版，其中包含了 tclvfs 的预编译版。也可以从 http://sourceforge.net/projects/tclvfs 处下载 tclvfs 的源代码。

一旦挂载了虚拟文件系统，就可以使用 vfs::filesystem info 命令来取得所有挂载的虚拟文件系统的列表。例如：

```
    puts [vfs::filesystem info]
⇒ {/Users/ericfj/Documents/tcl/zip}
```

当使用完一个虚拟文件系统后，使用 vfs::filesystem unmount 命令可以将其卸载。

```
vfs::filesystem unmount zip
```

另一个示例是挂载 mk4，或者说 Metakit 文件，将它挂载为一个虚拟文件系统。Tcl

Starkit 和 Starpack 都是 Metakit 文件。可以从类似 http://tcl.tk/starkits 的站点下载像 tclhttpd.kit(一个用 Tcl 编写的 HTTP 服务器)这样的 Starkit 文件。该文件是 Metakit 文件很好的示例，还有一段优秀的 Tcl 源码展示了如何构建一个 HTTP(网页)服务器。

可以用访问 ZIP 文件的方法访问一个 Starkit 文件的内容。例如：

```
package require vfs::mk4
vfs::mk4::Mount tclhttpd.kit kit
foreach f [glob kit/*] {
    puts $f
}
vfs::filesystem unmount kit
```

当运行这个脚本时，会看到如下输出：

```
⇒ kit/bin
⇒ kit/custom
⇒ kit/htdocs
⇒ kit/lib
⇒ kit/main.tcl
```

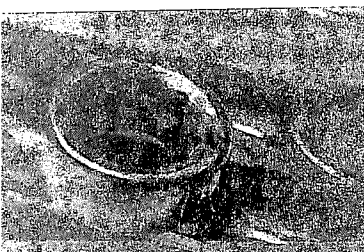
如何使用 vfs 和 vfs-filesystems 处理虚拟文件系统的更多信息请查阅参考文档。

11.8 系统调用中的错误

本章中讲述的大多数命令都调用操作系统中的系统调用，很多情况下系统调用会返回错误。例如，如果对一个不存在的文件调用 open 或 file stat，如果在读取一个文件时发生了 I/O 错误，系统调用就会返回错误。Tcl 命令检测这些系统调用错误，在大多数时候会返回错误本身。错误消息可以用来识别发生了的错误。

```
    open bogus
Ø   couldn't open "bogus": no such file or directory
```

当系统调用中出现错误时，Tcl 将设置全局变量 errorCode，返回的选项字典中的 -errorCode 关键字对应的值提供了更多的有关错误的信息。这个值通常是三个元素的列表，第一个元素是字符串 POSIX，最后一个元素是适合阅读的错误消息。有关错误处理的更多信息详见第 13 章。



第12章 进程间通信

Tcl 提供了一些用于处理进程的命令。您可以用 exec 命令创建新的进程，也可以用 open 命令创建新的进程，然后使用文件 I/O 命令进行通信。Tcl 还支持通过 TCP/IP 套接字进行进程间通信。您可以通过 pid 命令访问进程描述符，通过 env 变量对环境变量进行读写操作，还可以使用 exit 命令终止当前进程。

12.1 本章出现的命令

本章讨论的与进程控制和进程间通信相关的命令如下。

- `exec ?-keepnewline? ?-ignorestderr? ?--? arg ?arg ...?`
  使用一个或多个子进程运行由 *arg* 指定的命令管线，返回管线的标准输出，如果输出被重定向，返回一个空字符串(如果末尾有换行符，都会被丢弃，除非指定了-keepnewline)。I/O 重定向和管道都可以加以指定。如果最后一个 *arg* 是&，那么管线会在后台运行，返回值是它的进程 ID 的列表。如果任一进程对标准错误通道写入内容，而且该输出未被显式地重定向，exec 就会产生错误，如果这时指定了-ignorestderr 选项，就不会产生错误。
- `exit ?code?`
  终止一个进程，将 *code* 作为退出状态信息返回给父进程。*code* 必须是一个整型值，默认值为 0。
- `open |command ?access?`
  将 *command* 作为一个列表处理，保持其结构作为一个参数传递给 exec，创建子进程以执行其中给出的命令(可以是一个命令，也可以是多个命令)。根据 *access* 的指定，它为管线创建写入和/或读出的管道。返回用于子进程通信的一个文件描述符。
- `pid ?fileID?`
  如果没有指明 *fileID*，则返回当前进程的进程描述符。否则返回所有与 *fileID* 相关的管线的进程 ID 列表(*fileID* 必须以|作开头)。
- `socket ?options? host port`
  打开一个连接到指定的 *host* 和 *port* 的客户端 TCP/IP 套接字。*host* 可以是一个域类型的服务器名，也可以是一个数字 IP 地址。*port* 可以是一个整型数，也可以是由服务器操作系统支持和解析的服务名称。命令支持的选项参见后面相关章节和

参考文档。

- `socket -server command ?options? port`
  为 *port* 打开一个 TCP/IP 服务器套接字，返回创建的服务器套接字的通道描述符。*port* 可以是一个整型数，也可以是由服务器操作系统支持和解析的服务名称；如果 *port* 值为 0，操作系统会为服务器套接字分配一个未使用的端口。当一名用户连接到服务器端口时，Tcl 自动创建一个新的通道，用于与该用户通信，然后调用指定的 *command*，再提供三个参数：新用户通信通道、用户 IP 地址以及用户端口号。关闭服务器通道会使服务器关闭，因此就不接受新的连接了。命令支持的选项参见后面相关章节和参考文档。
- `chan configure channelID ?optionName? ?value? ?optionName value ...?`
  `fconfigue channelID ? optionName? ?value? ?optionName value ...?`
  查询或设置 *channelID* 通道的配置选项。如果没有提供 *optionName* 及 *value* 值，该命令返回一个字典，其内容是所有的配置项和对应的值。如果提供了 *optionName* 但没有提供 *value* 值，命令返回指定选项的当前值。如果提供了一对或多对 *optionName* 和 *value*，那么该命令将指定选项设为对应的值，这种情况下返回值是一个空字符串。命令支持的选项参见后面相关章节和参考文档。从 Tcl 8.5 开始，推荐使用 chan configure 而非 fconfigure。
- `chan event channelID event ?script?`
  `fileevent channelID event ?script?`
  将 Tcl 脚本 *script* 安装为文件事件处理器，在 *channelID* 的状态为 *event* 描述的状态时调用脚本(必须是可读的或可写的)。同一时刻一个通道一个事件只能安装一个这样的处理器。如果 *script* 是空字符串，则删除当前处理器。如果没有给出 *script*，则返回当前安装的脚本(如果当前没有安装脚本，则返回空字符串)。从 Tcl 8.5 开始，推荐使用 chan event 而非 fileevent。
- `vwait variableName`
  进入 Tcl 事件循环，继续执行所有接收到的事件，直到某个事件处理器设置 *variableName* 指定的变量的值。在指定的变量的值设置完成时，vwait 命令才最终返回。

12.2 用 exit 终止 Tcl 进程

调用 exit 命令终止执行这个命令的进程。exit 接受一个可选的整型参数。如果提供了这个参数，它将存放退出状态信息，返回给父进程。0 表示正常退出，非零值对应非正常的退出；这个值很少使用 0 和 1 以外的数。如果没有给 exit 提供参数，它以状态 0 退出。因为 exit 终止了进程，它本身没有任何返回值。

12.3 用 exec 调用子进程

exec 命令创建一个或多个子进程，并且等待它们完成再返回。例如：

```
exec rm main.o
```

将 rm 作为子进程运行，为它传入参数 main.o，在 rm 完成后返回。exec 的参数与您在 sh 或 csh 外壳程序的命令行中输入的类似。exec 的第一个参数是要运行的程序名，其他参数均为子进程的参数。

提示： 上面这个示例只是演示。如果您的目标是要删除文件、复制文件，或是列出一个目录中的内容，等等，像 file delete 这样的 Tcl 内建命令是更好的选择，原因有二：首先，可以避免与平台相关的命令，像 Unix 中的 rm 和 Windows 中的 del，您可以在 Tcl 支持的任意操作系统中运行您的脚本；其次，内建 Tcl 命令通过系统调用来删除文件，而无需耗费资源开启一个新的进程。

要运行一个子进程，exec 首先查找名称为 exec 的第一个参数的可执行文件。如果该名称包含/或者以-开头，exec 查找由该名称指定的单独文件。其他情况下，exec 在环境变量 PATH 提供的所有目录中查找指定的可执行文件。exec 使用它所找到的第一个可执行文件。

exec 收集子进程写到标准输出的所有信息，将这些信息作为它的结果返回，示例如下：

```
    exec wc /usr/include/stdio.h
⇒       71      230     1732 /usr/include/stdio.h
```

如果输出的最后一个字符是换行符，那 exec 会丢弃它。这个行为看上去有点奇怪，但它使得 exec 与其他 Tcl 命令相融，它们通常不会让结果的最后再存在一个空行。您可以把-keepnewline 选项作为第一个参数提供给 exec，保留这个换行符。

exec 支持与 Unix 外壳类似的 I/O 重定向。例如，如果 exec 的一个参数是>foo，进程的输出就会存放到文件 foo 当中，而不是作为 exec 的结果返回给 Tcl。这种情况下 exec 的结果是空字符串。还有很多其他形式的输入输出重定向，如表 12.1 所示。

提示： Windows 不支持*@fileID* 语法。另外，并不是所有的 Windows 应用程序都能在 exec 命令下正常工作。

表中展示的各种形式中，*file* 或 *value* 或 *fileID* 都可以跟在重定向符后面结合在一起作为一个参数，或者作为独立参数，重定向的一个应用示例如下：

```
exec cat << "test data" > foo
```

这个命令用 test data 字符串覆盖了文件 foo。这个字符串作为 cat 的标准输入传入，cat 将这个字符串复制到它的标准输出，这个标准输出现在已经重定向到 foo 文件。如果没有指定输入重定向，这个子进程将从 Tcl 应用程序中继承标准输入通道；如果没有指定输出重定向，这个子进程的标准输出就是 exec 的返回结果。

表 12.1 子进程的 I/O 重定向语法

重定向	说明
>file	重定向标准输出到*file*，覆盖它以前的内容
>>file	将标准输出添加到*file*，而不是替换它
>@fileID	重定向标准输出到描述符(由 open 返回)为*fileID* 的已打开的文件
2>file	重定向标准错误到*file*
2>@fileID	重定向标准错误到描述符(由 open 返回)为*fileID* 的已打开的文件
2>@1	将所有程序中的标准错误重定向到命令的结果
>&file	将标准输出和标准错误重定向到*file*，覆盖它以前的内容
>>&file	将标准输出和标准错误重定向到*file*，添加到它以前的内容之后
>&fileID	重定向标准输出和标准错误到描述符(由 open 返回)为*fileID* 的已打开的文件
<file	使标准输入从*file* 中获取信息
<<value	将 *value*(直接的值)作为子进程的标准输入传给子进程
<@fileID	使标准输入从描述符为*fileID* 的已打开的文件中获取信息
|	管道，前一程序的标准输出作为后一程序的标准输入
|&	管道，前一程序的标准输出和标准错误作为后一程序的标准输入

除了调用单个进程外，还可以调用进程管线，示例如下：

```
    exec grep #include tclInt.h | wc
⇒       8       25      212
```

grep 程序取得 tclInt.h 文件中所有包含#include 字符串的行。这些行通过管道传递给 wc 程序，该程序计算 grep 输出的行数、单词数和字符数，然后将这些信息输出到标准输出。wc 的输出作为 exec 的结果返回。

如果 exec 的最后一个参数是&，相应的子进程会采取后台运行。exec 会立即返回，而不等待子进程完成。它返回管线中所有进程的进程号的列表；除非特别指定，子进程的标准输出和标准错误分别定向到 Tcl 应用程序的标准输出和标准错误。

如果某个子进程挂起或异常退出，exec 命令就会产生一个错误(例如，它被强行终止，或者返回了非零的退出状态值)。如果某个子进程向标准错误通道进行输出，而标准错误未被重定向，exec 命令也会产生一个错误；可以为 exec 命令指定-ignorestderr 选项让它在这种情况下不产生错误。错误消息的内容首先是最后一个子进程产生的输出(除非它被>重定向了)，然后是各个非正常退出的进程的错误消息，然后是进程为标准错误所产生的信息。另外，如果存在的话，exec 会设置 errorCode 变量以及返回的选项字典中-errorcode 关键字所对应的值，保存最后一个非正常终止的进程的信息(详见 13.2 节和参考文档)。

提示： 很多 Unix 程序并没有认真对待它们返回的退出状态。如果用 exec 命令调用这样的程序，然后它在正常完成后返回了非零的状态值，那 exec 命令就可能产生虚假错误。要避免这样的错误导致您的脚本异常中断，可以在 catch 命令中调用 exec，详见第 13 章。

尽管 exec 的功能与那些 Unix 外壳相似，但也有一个显著的不同：exec 不进行文件名称展开。例如，您调用下面这条命令，希望删除当前目录中的所有.o 文件。

```
    exec rm *.o
Ø   rm: *.o nonexistent
```

rm 会将*.o 作为它的参数接受，然后报错退出，因为它不能找到这个文件。如果希望进行文件名称展开，可以使用 glob 命令，但也不是简单地使用。例如下面这条命令也是不能工作的。

```
    exec rm [glob *.o]
Ø   rm: a.o b.o nonexistent
```

失败的原因是 glob 返回的文件名列表作为一个参数传给了 rm。例如，有两个.o 文件，a.o 和 b.o，那么 rm 的参数就是 a.o b.o；因为没有叫做 a.o b.o 的文件，rm 会返回一个错误。这种情况下推荐的处理方法是使用 Tcl 的参数展开语法，将列表中的元素分别作为独立的变量。

```
exec rm {*}[glob *.o]
```

如 2.8 节所述，在 Tcl 8.5 以前的版本中，可以用 eval 命令达到类似的效果。

```
eval exec rm [glob *.o]
```

12.4 命令管线的输入输出

您还可以使用 open 命令创建子进程。一旦您这样做了，就可以使用 gets 和 puts 之类的命令与管线进行通信。下面是两个简单的示例：

```
set f1 [open {|sort -k 2 > newfile.txt} w]
set f2 [open | prog r+]
```

如果传递给 open 的文件名的第一个字符是管道符号|，这个参数并不是真正的文件名称。它所指明的，是一个命令管线。参数中|后面的部分作为一个列表对待，其元素的意义和传给 exec 命令的各参数的意义相同。open 创建了子进程的一个管线，返回一个描述符，可以使用这个描述符向管线中传入或从管线中取出数据。第一个示例中，管线是为写操作打开的，管道用于 Unix sort 程序的标准输入，然后您就可以调用 puts 通过管道写入数据；sort 的输出重定向到名为 newfile.txt 的文件。第二个示例打开的是读写管线，为 prog 的标准输入和标准输出分别创建了管道。puts 这样的命令可以用于向 prog 输入数据，像 gets 和 read 这样的命令可以用于从 prog 中取得输出。总之，第 11 章讨论过的 I/O 相关命令都可以用于管线通道的输入输出操作。

提示： 向一个管线中写入数据时，不要忘了输出缓冲机制。在您调用 chan flush 命令强制写出缓冲区数据之前，数据很可能没有输出到子进程中。您可以使用 chan configure 命令改变通道的缓冲区设置，详见 12.5.2 节。

当您关闭对应命令管线的文件描述符时，close 命令会强制输出该管线的所有输出缓冲区的内容，关闭通向该管线和由该管线伸出的管道，等待管线中所有的进程退出。如果任

一个进程非正常退出，close 都会像 exec 那样返回一个错误。

12.5 配置通道选项

在第 11 章已经讨论过，chan configure 命令可以设置通道选项(为与更早期的 Tcl 版本兼容，fconfigure 命令仍然可用)。第 11 章讨论了用这个命令设置字符编码，以及设置文件 I/O 的行结束符翻译方式。使用 chan configure 也可以为命令管线和套接字通道配置同样的设置。本节还将介绍另外一些对命令管线和套接字通道特别有用的 chan configure 设置。

12.5.1 通道阻塞模式

默认情况下，当您对一个通道调用 gets 或 read 时，Tcl 会一直阻塞到可以读取数据为止。gets 阻塞到它读取一行完整的数据为止，而 read 阻塞到它读取了指定个字节的数据，或到达文件尾为止。类似地，puts 命令把数据输出到缓冲区，如果缓冲区满，puts 会阻塞到数据送出为止。这称作阻塞模式。阻塞可应用于很多目的，也可简化程序。例如，您有一个程序要读取文件的内容，处理数据，然后输出结果。大多数情况下，这些应用中阻塞的工作效果很好。

然而，如果应用程序打开了一个进程管线或网络套接字，数据可能是零零星星地间隔着到来的，I/O 命令的阻塞可能让进程暂停相当长一段时间。如果应用程序有一个图形化用户界面，在 I/O 命令阻塞时 Tcl 也不能提供按键、按钮、窗口刷新以及其他交互服务。在这些应用程序中，使用非阻塞式的通道通常来说更好。类似地，网络服务器通常需要非阻塞式的通道以同时处理多个用户。服务器不能等待一个用户发送完数据，然后才为其他的用户服务，特别是服务器等待的用户可能现在并没有在发送数据。在阻塞模式下，其他用户都会长时间无法获得服务。

chan configure 的-blocking 选项指定了通道是否设置为阻塞模式。您可以提供布尔型值来改变设置。例如，将一个通道设为非阻塞模式。

```
chan configure $chan -blocking 0
```

如果不能为 chan configure 提供这个值，该命令会返回当前的选项值。例如，查看一个通道是否被设置为阻塞模式，可以使用以下命令：

```
chan configure $socketid -blocking
```

当一个通道设置为非阻塞模式时，对该通道的 read 返回通道中当前可见的字符，如果指定了单次读取最大字节数，返回最多长达指定字节数的字符。对非阻塞通道使用 gets 时，仅在有完整的一行可读时才返回字符。如果没有完整的一行，gets 不会取回任何字符。如果没有提供一个变量名作为参数，gets 返回一个空字符串；如果提供了变量名，gets 将该变量设为空字符串，返回-1。这和 gets 遇到文件尾的结果是完全一样的，因此 Tcl 还提供了两个命令用于区分这两种情况。

chan eof 命令在通道的上一次操作遇到文件尾时返回 1，否则返回 0。chan blocked 命令在通道的最近一次输入操作中因缺少可用的信息而未取得数据时返回 1，否则返回 0。

(为与更早期的 Tcl 版本兼容，eof 和 fblocked 命令仍然可用。)

12.5.2 通道的缓冲模式

Tcl 自动为所有 I/O 通道缓冲，以提高效率。通道的输入和输出缓冲区有相同的大小，默认为 4 KB。您可以使用 chan configure 命令的-buffersize 选项查看或修改通道的缓冲区大小。现在可以把缓冲区的大小设为从 1 字节到 1 MB 的任意值；这个范围之外的设置值会被调整为所能支持的最小字节数或最大字节数处理。

输入缓冲区只有缓冲区大小一个可设置的参数。如果输入缓冲区中没有足够的数据满足通道的一次 read 要求，Tcl 会从相应的源(如文件、管线、套接字等)中读取当前可用的直到最大缓冲区大小的尽可能多的数据。

通道的输出缓冲区的行为也是由 chan configure 命令的-buffering 选项控制，该选项决定 Tcl 是否自动输出通道缓冲区的内容。-buffering 选项支持的值有 none(每次输出都自动输出缓冲区的内容)、line(每一行自动输出一次)以及 full(填满缓冲区时输出，或脚本调用 chan flush 时输出)。除了连接到类似终端的设备的通道，选项-buffering 的默认值为 full；类似终端的设备的通道的初始设置为 line。另外，stdin 和 stdout 初始设置为 line，stderr 初始设置为 none。

12.6 事件驱动的通道交互

12.5.1 节讲述了通道的默认阻塞行为，一般来说，对多数应用程序那都不是问题。如果应用程序由控制台驱动而不是由图形化用户界面驱动，或者应用程序根本不需与用户交互，对通道的 I/O 操作阻塞几秒钟甚至几分钟都不是什么大问题。另一方面，如果应用程序必须提供一个有响应的用户界面，必须同时处理多个通道，在某个通道进行 I/O 操作时把整个程序冻结一阵就不可接受了。将通道改为非阻塞模式有助于解决这个问题。但即使那样，从通道中进行读取也可能返回不了数据，因为在源头可能没有可用的数据。反复尝试从一个通道进行读取，直到数据可用，则会浪费处理时间。

其他很多语言使用线程处理与阻塞的通道交互的问题。如果一个线程被 I/O 操作阻塞，其他线程仍可继续运行。不过多线程程序有它们自己的问题，例如资源争夺的管理以及同步问题等。尽管 Tcl 也有 Thread 扩展包，在脚本层次支持多线程编程(详见附录 B)，但传统的 Tcl 采用了另一种解决方法：事件驱动的编程。

在事件驱动的编程模式中，应用程序将脚本注册为事件处理器，在运行时由相应事件的发生触发。第 1 章已经介绍了事件处理器的概念，用来实现响应用户交互行为的组件动作。在 Tk 中使用事件绑定的方法把用户交互行为和其他窗口事件与 Tk 动作相关联，事件与绑定的更多信息参见第 17 章和第 22 章。Tcl 还通过 after 命令支持时间相关事件，在第 15 章中有专门的讨论。

Tcl 还支持另一类绑定，将反应绑定到通道上发生的事件，这些事件称为文件事件或通道事件。例如，可以把一个脚本注册为：在管线或套接字通道有数据到达时调用。如果

应用程序需要与多个通道交互作用，每一个通道都可以有它自己的文件事件处理器，根据发生的事件进行适当的操作。同时，Tcl 还可以监视窗口事件和时间事件，调用相关的任何事件处理器。

12.6.1 用 vwait 进入 Tcl 事件循环

要检测事件的发生并调用相应的事件处理器，Tcl 必须处于它的事件循环中。如果一个应用程序由 wish 解释器调用，或加载有 Tk 工具包，它会在执行完脚本文件中的所有命令后自动进入事件循环。对于没有使用 Tk 的应用程序，必须使用一个命令显式地进入 Tcl 事件循环。

vwait 命令通常用于调用事件循环。vwait 接受一个参数，这是一个全局变量或长期存在的命名空间变量的名称，例如：

```
vwait enter_loop
```

vwait 命令进入 Tcl 事件循环，根据接收到的事件运行，直到某个事件处理器设置了指定变量的值(这个示例中就是 enter_loop)。然后，在该变量的值被事件处理器设置完成后，vwait 命令最终返回。

12.6.2 注册文件事件处理器

您可以使用 chan event 命令注册文件事件处理器。(为与更早期的 Tcl 版本兼容，fileevent 命令仍然可用。)除了给定通道描述符和事件发生时应该调用的脚本，还要说明是为通道的可读事件还是可写事件注册处理器。例如，下面这条命令在通道中出现可读数据时调用命令 ReadLine，指定通道的通道描述符保存在变量 pipe 中。

```
chan event $pipe readable ReadLine
```

对于一个给定通道，只能创建最多一个可读事件处理器和一个可写事件处理器。调用 chan event 为一个通道安装指定类型的事件处理器，会替换掉该通道以前的同类型事件处理器。如果不给出脚本参数，chan event 会返回当前安装的事件处理器(如果当前已安装)。

事件循环必须激活，Tcl 才能探测和处理文件事件。另外，一般来说应该使用 chan configure 命令将通道设为非阻塞模式。否则，当处理器试图向通道进行读写操作时可能会被阻塞。

当一个通道可以向它对应的文件或设备写入至少一个字节的数据，而不引起阻塞或达到对应文件或设备发生错误的条件，就认为这个通道是可写的。可写文件事件处理器最常见的应用就是检查以异步方式打开的客户端套接字是连接上了，还是连接失败。12.9 节对此进行了详细讨论。

当一个通道对应的文件或设备中有未被读取的数据时，就认为这个通道是可读的。如果读取缓冲区中有未被读取的数据时，也认为这个通道是可读的，只有一种例外：最近尝试从通道中读取信息调用的是 gets，而在输入缓冲区中找不到完整的一行。这一特点允许在非阻塞的方式下使用事件处理一次一行地读取一个文件。如果对应的文件或设备呈现了文件结束符或错误条件，通道也被认为是可读的。

提示： 事件处理器必须能够检测这些条件，正确地处理它们。例如，如果对文件尾不作检查，那处理器可能会读不到数据，返回，然后立即又被调用，结果陷入死循环。

下面的脚本是使用文件事件的简单示例，它处理一个进程管线的输出。这个示例展示了安全有效地处理异步通信通道的一些功能。

```
set fid [open "| du /usr " r]
chan configure $fid -blocking 0
chan event $fid readable [list ReadLine $fid]

proc ReadLine {fid} {
    global done
    if {[catch {gets $fid line} len] || [chan eof $fid]} {

        # The last gets encountered EOF or we had an
        # abnormal termination

        catch {close $fid}
        puts stderr "Channel closed"
        set done 1
        return
    }elseif {$len >= 0} {

        # We read a complete line, otherwise gets would have
        # read no characters and returned -1.
            puts "Read line: $line"
    }
    # We reach this point if there wasn't a complete line
    # to read. Simply return to the event loop to wait
    # for more data.
}

vwait done
```

脚本的第一行为读取访问打开了一个进程管线。这里，它启动了 Unix du 命令，该命令递归地报告目录的磁盘使用情况。然后脚本将这个通道设置为非阻塞式通道。如果这是一个您要向其中写入信息，把数据传给过程管线的双向通道，您可能还想为通道配置缓冲区，例如将其设置为 line 模式，这样每一行都能被直接写入管道。

然后脚本注册了一个文件事件处理器，当管道中有可读的数据时就调用这个处理器。这里的脚本是一个格式正确的列表，包括 ReadLine 命令以及它的通道描述符参数。对于这样简单的处理器，我们可以只用双引号把处理器脚本括起来，允许变量替换，因为通道描述符并不含有空格或其他特殊字符。然而，如果要把其他可能是任意字符串的值作为参数传递，只使用双引号就可能生成错误的命令。在注册回调时，最安全的方法是使用 list 命令这样的技术，保证得到的脚本参数是格式正确的 Tcl 命令。在定义事件处理器过程之后，脚本最终调用 vwait 命令进入事件循环。在事件循环中 Tcl 继续根据事件运行，直到某个处理器设置了变量 done 的值，然后 vwait 命令返回，脚本终止。

ReadLine 处理器本身首先试图从通道中读取完整的一行数据。如果通道非正常关闭，gets 命令可能失败，因此最安全的方法是在一个 catch 语句中运行。如果 catch 报错，或者通道探测到文件尾，那我们就已经从通道中读取了所有可能的数据。处理器关闭通道(使用

catch 静默丢弃了错误)，将全局变量 done 设为 1，然后返回事件循环。因为 vwait 命令在等待 done 被设置，这也就终止了事件循环。

另一方面，如果没有探测到错误或文件尾，ReadLine 就检查 gets 命令的返回值。如果它是-1，说明 gets 没能找到完整的一行，这种情况下它不会从通道中读取任何字符。(另一种返回-1 的情况，是遇到了文件尾，这一情况前面已经检查过了。)这种情况下，ReadLine 就是简单地返回事件循环，等待更多的数据到达。而如果 gets 能够读取完整的一行，读取到的字符就保存在变量 line 中，可以对它们进行处理了。

12.7 进程 ID

Tcl 提供三种方法访问进程描述符。第一，在后台用 exec 调用管线，返回管线中所有子进程的进程描述符的列表。您可以使用这些描述符来结束进程。第二，可以不带参数地调用 pid 命令，它将返回当前进程的进程描述符。第三，可以用一个文件描述符作为参数，调用 pid 命令，例如：

```
set f [open {| tbl | ditroff -ms} w]
pid $f
⇒ 7189 7190
```

如果有与打开的文件相关的管线，例如上面这个示例，pid 命令就会返回该管线中所有进程的描述符的列表。

12.8 环境变量

在第 3 章中已经提到过，全局数组变量 env 包括了所有的环境变量，这些环境变量存储为数组的元素，env 中的元素名就对应环境变量的名称。如果修改 env 数组，变化就会反映在进程的环境变量上，新的环境变量值也会传递给由 exec 或 open 创建的新的子进程。

12.9 TCP/IP 套接字通信

套接字链接应用程序使用如 TCP/IP(传输控制协议/互联网协议)这样的网络协议。因为基于套接字的网络操作是标准化的，Tcl 对套接字的支持也为您提供了另一种方法来整合 Tcl 应用程序与其他非 Tcl 应用程序。

套接字由主机名或 IP 地址加上端口号描述组成。端口号，某种程度上就像电视的频道号。大多数网络服务的端口号是标准化的(例如，80 端口用于 HTTP)。一般来说，小于 1024 的端口号都是由特权应用程序使用的。另外，在 1024～49151 之间的端口号也有不少是标准化的，例如，6000 和 6001 由 X Windows 系统使用，通常在互联网地址指派机构(IANA)注册。更高的端口号未被标准化，可以由您的系统中安装的其他应用程序使用。在

Unix 系统中，查看文件/etc/services 可以找到预定义端口号的列表。

和大多数系统不同，Tcl 的通信通道是由事件驱动的。其他语言编写的服务器在处理新的连接时一般来说要么从当前进程进入分支，要么展开新的线程。而 Tcl 服务器通常使用事件处理器过程来处理网络请求。这一处理方式的细节情况 12.6 节已有介绍。

12.9.1 创建客户通信套接字

使用 socket 命令创建套接字。可以使用这个命令为客户端应用程序创建套接字以连接到服务器，也可以为服务器创建等待用户请求的套接字。创建客户端套接字需要提供服务器名或 IP 地址以及端口号作为参数。

```
set sockid [socket localhost 12345]
```

这条命令在本地系统中打开一个客户端套接字，端口号为 12345。返回的通道描述符保存在 sockid 中。

提示： localhost 指的是您运行 Tcl 应用程序的计算机。

对于客户端套接字，可以使用-myaddr 选项指定本地系统的网络接口的互联网地址，对于有多于一个网络接口(或网卡)的系统，这一选项十分好用。如果需要的话，使用-myport 选项指定客户端端口；如果没有给出该选项，客户端端口号由系统软件随机选择，客户端套接字不指定端口号的情况是很常见的。

一旦建立了套接字连接，就可以配置通道，可以像对进程管线那样使用 I/O 命令。例如，可以将它设置为非阻塞模式，为可读数据到达套接字的事件注册一个事件处理器，就像 12.6.2 节中对进程管线的处理一样。默认情况下，套接字管道设置为阻塞式全缓冲。-translation 默认设置为{auto crlf}，-encoding 默认设置为系统编码。这些设置的更多信息以及如何使用 chan configure 命令进行设置的内容参见 11.6 节。

对于套接字通道，chan configure 命令还支持另外两个只读的选项，用来报告有关套接字连接的信息。-sockname 的值是包含三个元素的列表，三个元素分别是 IP 地址、主机名以及套接字的端口号。-peername 选项的值也是有三个元素的列表，对应有关同位套接字的相同的信息。

默认情况下，在连接建立后 sock 命令才返回。您也可以指明-async 异步地创建一个连接。这意味着 socket 命令会在连接建立前返回。如果套接字设置为阻塞模式，接下来的 gets、read 或 chan flush 命令都会等待套接字建立连接。如果套接字不在阻塞模式下，接下来的 gets、read 或 chan flush 命令会立即返回，而对 chan blocked 的调用会返回 1。对通道调用这些命令时，套接字连接中发生的任何错误都会作为错误返回。可以使用 chan event 命令注册一个 writable 事件处理器，用来探测什么时候连接成功建立了；以异步方式打开的客户端套接字在建立连接或连接断开时成为可写的套接字。然后可以用-peername 选项检查通道，确定连接是否成功，示例如下：

```
proc connected {fid} {

    # Remove the writable event handler to prevent it
    # from being invoked again
```

```
        chan event $fid writable {}

        # Test for the existence of -peername in the channel
        # configuration to see if the connection was successful.

        If {[dict exists [chan configure $fid] -peername]} {
           # We're connected. Do whatever you like.
        } else {
           # We're not connected. Clean up and report the
           # situation however you like.
        }
}

set fid [socket -async $host $port]
chan event $fid writable [list connected $fid]
```

12.9.2 创建服务器套接字

要创建服务器套接字，应该用-server 选项调用 socket。

```
socket -server command ?option? port
```

这条命令在当前主机上以指定的端口号创建了一个服务器套接字，也称为监听套接字。您还可以传入服务器名代替端口号，它的值决定于您的操作系统。对于有多个网络接口的系统，-myaddr 选项可通过给定主机名或 IP 地址指定要使用的特定网络接口；不加指定的话，服务器套接字可以接受来自任一接口的连接。

当服务器套接字探测到客户连接时，Tcl 为客户创建读写通道，调用 *command*，您需要为 *command* 提供三个参数：新的套接字描述符、客户的地址和客户的端口号。(Tcl 的事件循环必须在运行中，服务器套接字才能探测到客户连接请求。)一旦与客户建立了连接，就可以使用 12.9.1 节中讲述的普通通道命令：配置通道，与客户端进行通信。因为每一个客户端的连接都指定了独一无二的通道描述符，服务器可以同时管理多个客户连接。可以使用数组、字典或其他数据结构存储与各个客户相关的信息，这种情况下套接字通道描述符提供了完美的唯一关键字。

下面这段脚本展示了一个简单的多用户响应服务器的完整实现。

```
set listen [socket -server ClientConnect 9001]

proc CLientConnect {sock host port} {
    chan configure $sock -buffering line -blocking 0
    chan event $sock readable [list ReadLine $sock]
    SendMessage $sock "Connected to Echo server"
}

proc ReadLine {sock} {
    if {[catch {gets $sock line} len] || [eof $sock]} {
        catch {close $sock}
    } elseif {$len >= 0} {
        EchoLine $sock $line
    }
}

proc EchoLine {sock line} {
    global forever
```

```
    switch -nocase -- $line {
        exit {
            SendMessage $sock "Killing Echo server"
            catch {close $sock}
            set forever 1
        }
        quit {
            SendMessage $sock \
                "Closing connection to Echo server"
            catch {close $sock}
        }
        default {
            SendMessage $sock $line
        }
    }
}

proc SendMessage {sock msg} {
    if {[catch {puts $sock $msg} error]} {
        puts stderr "Error writing to socket: $error"
        catch {close $sock}
    }
}

vwait forever
catch {close $listen}
```

这段脚本在本地系统中创建了端口为 9001 的监听套接字。在探测到客户端连接请求时，Tcl 调用 ClientConnect 过程。该过程配置与客户连接的通信通道，为通道注册一个可读文件事件处理器，然后将连接确认信息发送给客户。注意，这里套接字通道描述符包含在 ReadLine 过程的参数中，ReadLine 过程被注册为事件处理器。其结果是 ReadLine 过程在被调用时就知道那个通道有可用的数据，而我们就可以把这个过程应用于所有的通道。

ReadLine 过程与 12.6.2 节中给出的用于管线通道的事件处理器十分相似。如果用户断开连接或者从通道中读取数据时发生错误，它就关闭通信套接字。否则，它会检查是否能从通道中读取到完整的一行，如果能，它就把读取到的一行传给 EchoLine 过程进行处理；如果不能，ReadLine 就返回事件循环，等待更多的数据到达通道。

EchoLine 过程检查刚才读取到的一行是不是特殊的控制信息。包括 exit 字符串的行通过设置 vwait 变量并返回来终止响应服务器。包括 quit 字符串的行让响应服务器向客户发送提示信息，然后关闭客户的套接字通道，与客户断开连接。如接收到其他的行，就直接把接收到的行发回给客户。

SendMessage 过程用于将消息安全地发送给客户。如果在发送消息时检测到错误，套接字不响应。这个过程会向标准错误通道提供信息，然后关闭套接字通道。

提示： 这个示例中对 catch 命令的扩展使用，是编写多客户端服务器程序的很好的方法。您不会希望因为与一个客户端的连接发生错误就关掉整个服务器。使用 catch 命令管理在打开通道时以及对通道的 I/O 操作中出现的所有错误，进行相应处理。

12.10 向 Tcl 程序发送命令

TCP/IP 套接字使您可以在应用程序之间发送数据，Tk 工具包允许您直接从一个基于 Tk 的应用程序中向另一个 Unix 系统中的应用程序发送命令。使用 send 命令，任意 Tk 应用程序都可以调用另外的 Tk 应用程序中的任意 Tcl 脚本；这些命令可用于获取信息，也可用于进行改变目标应用程序状态的操作。

提示： send 命令是 Tk 的一部分，不是基本 Tcl 语言的一部分。

send 命令是基于 X Windows 系统的数据传输功能创建的，在很多 Unix 系统中用于图形处理。在不使用 X Windows 系统的操作系统中，不能使用 send 命令。

提示： 在大多数情况下，应该使用套接字在应用程序之间传递数据。不仅是因为套接字更加安全，还因为 Tcl 套接字可以在多种操作系统中工作。

12.10.1 send 基础

要使用 send，只需给出应用程序的名称以及要执行的该应用程序中的 Tcl 脚本。例如下面这条命令：

```
send myapp {selectLine 200}
```

send 的第一个参数是目标应用程序的名称，第二个参数是要在该程序中运行的 Tcl 脚本。Tk 找到名为第一个参数的应用程序，把脚本传递给它，然后用该应用程序的解释器运行脚本。脚本产生的结果和错误都传回发起 send 命令的源应用程序，由 send 命令返回。

send 是同步式的命令：直到指定的远端应用程序中脚本运行结束，结果返回后命令才完成。send 与 X 事件的不同在于它会等待远端应用程序响应，这段时间内应用程序的用户界面无响应。在 send 命令结束后，应用程序再进行普通事件处理，处理刚才那段时间里等待处理的事件。一个正在使用 send 命令的应用程序会在它自己的 send 命令完成之后，再处理它所接收到的 send 请求。这意味着，send 的目标可以给发送 send 命令的程序 send 回一个命令，而不会有死锁的危险。另外，可以为 send 提供-async 选项，告诉 send 命令以异步方式工作，不需等待发送的命令执行完成。

send 最常见的用途之一是在已经运行的 Tk 应用程序中进行简单的试验，例如换一种颜色，修改与某个按钮绑定的命令。大多数 Tk 应用程序不提供向应用程序中直接输入 Tcl 命令的界面。然而，您总是可以启动交互式的 wish 程序，然后调用 send 命令在那些应用程序中运行 Tcl 命令。例如，下面这条命令改变名为 scan 的应用程序中指定窗口的背景色。

```
send scan {.menubar.file configure -bg blue}
```

12.10.2 应用程序名称

要向一个应用程序发送命令，必须先知道它的名称。展示设备中的每一个应用程序都应有独一无二的名称，这些名称是任意取定的，只要保证其独有性即可。很多情况下，应

用程序名就是创建它的程序的名称。例如，wish 的默认应用程序名就是 wish；如果它在一个脚本文件的控制下运行，那么它使用的应用程序名称就是脚本文件的名称。而编辑器这样与一个文件或对象关联的程序，其应用程序名称通常有两个部分：程序的名称，和它所操作的文件或对象的名称。例如，如果一个名为 mx 的编辑器正在处理名为 tk.h 的文件，应用程序名称就可能是 mx tk.h。

如果一个应用程序请求已经被使用的名称，Tk 会在名称后加上一个数字，以免与已经存在的名称冲突。例如，如果在同一个展示中同时启动了两次 wish，第一个实例的名称为 wish，第二个实例的名称可能是 wish #2。类似地，如果在同一个文件上打开了第二个编辑窗口，它的名称就可能是 mx tk.h #2。

Tk 提供了三个命令来返回有关应用程序名称的信息。第一条命令：

```
    winfo name
⇒ wish #2
```

返回调用的应用程序的名称。第二条命令：

```
    winfo interps
⇒ wish {wish #2} {mx tk.h}
```

返回一个列表，它的元素是在显示设备中定义的所有应用程序的名称。第三条命令：

```
selection get APPLICATION
```

返回的是现在拥有指定选择的 Tk 应用程序的名称；如果当前没有 Tk 应用程序拥有该选择，这个命令会产生一个错误。

提示： 使用 tk appname 命令可以改变应用程序名称。

使用-displayof 选项指定不同的 X Window 显示设备。将其路径传递给一个窗口，然后 Tk 可以用它来识别显示设备。

12.10.3 有关 send 的安全问题

send 命令是一个潜在的重大安全漏洞。任何使用您的显示设备的应用程序都可以向该显示设备中任意 Tk 应用程序发送脚本，而且该脚本可以使用 Tcl 的全部文件读写功能和您的账号所拥有的子过程调用权限。在 X 显示设备服务器上这个安全问题必须解决，因为在同一显示设备上甚至不使用 Tk 的应用程序也可能受到利用。不使用 Tk 的话创建一个侵犯性的程序相对困难，而有了 Tk 和 send，这就是小事一桩了。

您可以为显示设备使用基于关键字的保护机制，如 xauth 而不使用 xhost 这样的基于主机的机制，将您自己很好地保护起来。xauth 产生一个不透明的授权字符串，并且告诉服务器只在应用程序能产生这个字符串时才允许它使用显示设备。通常这个字符串存储在一个文件当中，该文件只有指定用户才能阅读，从而限制可以使用特定显示设备的用户。

为了至少能提供最低限度的安全性，Tk 会检查由服务器使用的访问控制，拒绝掉那些 xhost 类型访问控制未被允许的 send(即只有特定的主机可以建立连接)，并且得到允许的主机的列表为空。这意味着除非应用程序用特定形式的授权(例如由 xauth 提供的授权)，否则它们就不能连接到您的服务器。

第 13 章　错误与异常

一个 Tcl 命令可能因为各种原因返回错误，例如，它没有接收到正确数量的参数，或者参数的格式错误，或者在执行命令时遇到其他一些问题，例如系统调用文件 I/O 时错误。大多数情况下，错误表示导致应用程序无法完全运行它正在处理的脚本的严重问题。Tcl 的错误功能旨在让应用程序更易于展现过程中的工作，把表示错误所在的错误消息展示给用户。假定用户会修正问题，再次运行。

错误只是一种更常见的现象的一个特例，这种现象称为异常。异常是导致脚本放弃运行的事件，这些事件包括 break、continue 和 return 命令，以及错误。Tcl 允许异常被脚本“捕获”，从而在出现异常时只绕开程序中的某部分工作。在捕获到异常以后，脚本就可以从异常中恢复。如果脚本不能恢复，它就会重新引发这个异常。

13.1　本章出现的命令

与异常有关的 Tcl 命令如下。

- `catch` *`command ?returnVar? ?optionsVar?`*
  将 *command* 作为 Tcl 脚本处理，返回标志该命令的完成状态的整型值。如果指定了 *returnVar*，它给出的是一个变量的名称，其值将被设置为 *command* 的返回值或产生的错误消息。如果给出了 *optionsVar*，它给出的变量的值将被设为返回的选项字典。
- `error` *`message ?info? ?code?`*
  产生一个错误，以 *message* 作为错误消息。如果给定了 *info*，而且它不是空字符串，那么它将用于初始化 errorInfo 变量。如果给定了 *code*，它将存放到 errorCode 变量中。
- `return` *`?option value ...? ?result?`*
  使当前过程返回一个异常状态。如果使用-code，它会指定返回状态，它的值只能是 ok、error、return、break、continue 或一个整型值。而-errorinfo 选项可用于为 errorInfo 变量指定起始值，而-errorCode 可用于为 errorCode 变量指定一个值。result 给出了与 return 相关的返回值或错误消息；默认为空字符串。为了支持更高级的自定义控制结构，还可以提供任意多对 *option value* 对，*option* 的名称可以任意指定，它会成为返回的选项字典的条目。您也可以包含一条显式的-options 参数，它的值必须是一个有效的字典，字典中的条目会和 *option value* 对一起，加入

返回的选项字典。

- `interp bgerror path ?cmdPrefix?`
  将 *cmdPrefix* 注册为由 *path* 指定的解释器的后台错误处理器。*cmdPrefix* 是一个命令的名称，后面可以接任意个参数。当后台错误发生时，就会调用该命令，传入指定的参数以及另外两个参数：一个错误消息和一个返回的选项字典。如果没有提供 *cmdPrefix*，interp bgerror 返回前面一条为解释器注册的命令。

13.2 在出现错误后会发生什么

当一个 Tcl 错误出现时，Tcl 解释器放弃对当前命令的运行，产生一个错误状态。如果命令属于某个脚本，这个错误也会导致放弃脚本的运行。如果错误发生在一个 Tcl 过程的运行中，会导致放弃这个过程及其调用者，以及调用者的调用者，直到所有活动的过程都被放弃。在所有这些 Tcl 活动都被展开之后，控制会最终返回到运行 Tcl 代码的应用程序，返回表示错误的代码，以及描述错误的消息。由应用程序决定如何处理错误情况，大多数交互式应用程序，例如，以交替模式运行的 tclsh，会把错误消息显示给用户，继续处理用户的输入。而面向批处理的应用程序，例如，tclsh 调用一个脚本文件的名称加以运行时，应用程序通常会向控制台输出错误消息，然后退出。

例如下面这段脚本，其目的是对一个列表的元素求和。

```
set list {44 16 123 98 57}
set sum 0
foreach el $list {
    set sum [expr {$sum+$element}]
}
```

```
Ø  can't read "element": no such variable
```

这个脚本错误，因为 element 没有值：expr 命令中的变量名 element 应该与 foreach 命令的循环变量 el 相对应。在脚本运行时，Tcl 解析 expr 命令时会发生错误：Tcl 试图替换变量 element 的值，却不能找到叫这个名字的变量，因而报错。这个错误信号会返回给 foreach 命令，就是这个命令调用 Tcl 解释器处理循环块。当 foreach 看到错误已经发生时，它就放弃循环将这个错误信号作为它自己的结果返回。这样放弃了整个脚本的运行。错误消息：

```
can't read "element": no such variable
```

和错误信号一起返回，这个错误消息将显示给用户。

在很多情况下，错误消息为您修正问题提供了足够的信息。然而，如果错误发生在很深层的嵌套过程中，可能就不能确定到底是哪里产生这个错误。为了帮助确定错误发生的位置，Tcl 对过程中的命令进行展开时，会创建一个追踪栈，并且把追踪栈存放在全局变量 errorInfo 中。追踪栈描述了对 Tcl 解释器的每一次嵌套的调用。例如，在遇到错误之后，errorInfo 的值为：

```
can't read "element": no such variable
    while executing
"expr {$sum+$element}"
```

```
    ("foreach" body line 2)
    invoked from within
"foreach el $list {
    set sum [expr {$sum+$element}]
}"
```

Tcl 还在另一个全局变量 errorCode 中提供了有关错误情况的信息。errorCode 的格式易于被 Tcl 脚本处理；在 Tcl 脚本中使用 catch 命令从错误中恢复时常常用到它，后面会对此进行讨论。errorCode 变量包含由一个或多个元素构成的列表。第一个元素描述了错误的普通类型，其他的元素提供了更多的基于类型的相关信息。例如，如果 errorCode 的第一个元素是 POSIX，就说明错误是在系统调用 POSIX 中发生的。errorCode 还包含两个元素，给出了产生这个错误的 POSIX 名称，如 ENOENT，以及一个描述错误的可读消息。还有其他的分别表示数学、文件访问以及子过程异常等的错误代码。并不是所有的 Tcl 命令都显式地设置 errorCode。如果一条命令产生了错误，而没有设定 errorCode，则 Tcl 会将其值设为 NONE。errorCode 可以采用的所有格式请查阅参考文档。

13.3 由 Tcl 脚本生成错误

Tcl 错误可能由实现 Tcl 解释器的 C 代码或内建命令生成，也可能由 Tcl 命令 error 生成，例如：

```
if {($x < 0) || ($x > 100)} {
    error "x is out of range ($x)"
}
```

error 命令生成一个错误，并且将其参数作为错误消息。

出于编程风格的考虑，应该只在需要放弃正在运行的脚本时才使用 error 命令。如果认为不必放弃整个脚本，就可以从错误中恢复，更好的方法可能是使用普通的返回机制来表示成功或失败(即如果命令成功，返回一个值，如果失败，返回另一个值，或者设置一个变量，用来表示成功或失败)。尽管可能从错误中恢复(详见 13.4 节)，但恢复机制比普通的返回机制复杂得多。因此，应该仅在不太可能不放弃脚本运行而恢复的情况下生成错误。

13.4 用 catch 捕获错误

错误通常会让所有正在活动的 Tcl 命令放弃运行，但在某些情况下，需要在某个错误发生后，继续运行某个脚本。例如，假设需要用 open 命令打开一个文件以读取它的内容。(Tcl 的文件操作命令参见第 11 章。)尽管 Tcl 提供了 file readable 命令检测文件是否存在以及进程是否有读取它的权限，但在打开它之前，系统中的另一个进程有可能删除了文件。如果文件不存在，open 会产生一个错误。

```
    open msg.txt
Ø   couldn't open "msg.txt": no such file or directory
```

在这种情况下，可以使用 catch 命令让脚本忽略掉这个错误。

```
    catch {open msg.txt}
⇒ 1
```

catch 的参数是一个 Tcl 脚本，由 catch 进行处理。如果脚本正常完成，catch 返回 0。如果在脚本中出现错误，则 catch 捕获该错误(catch 命令本身不因为这个错误而放弃运行)并返回 1。这个示例忽略了 open 中的任何错误。然而，我们仍然需要检查 catch 的返回值，以确定 open 是否成功完成。

catch 命令还可以接受第二个参数。如果提供了这个参数，它是一个变量的名称，catch 让这个变量保存脚本的返回值(如果正常返回)或错误消息(如果脚本产生了一个错误)。

```
    catch {open msg.txt} fid
⇒ 1
    set fid
⇒ couldn't open "msg.txt": no such file or directory
```

这种情况下，open 命令产生了一个错误，于是 fid 内容设置为错误消息。如果文件存在，open 就会正常返回，于是 catch 的返回值是 0，而 fid 的内容应该是 open 的返回值，即打开的那个文件的通道描述符。如果脚本成功完成时需要访问返回值时这种带两个参数的形式是十分有用的。如果在发生错误后需要使用错误消息，例如将错误消息写到文件中，这种形式也十分有用。

您还可以再提供一个变量名作为可选的第三个参数，获取返回的选项字典，后文将对此进行讨论。

13.5 异常概述

错误并不是 Tcl 中引起工作中止的唯一事件。错误只是名为异常的一类事件的特例。除了错误，Tcl 中还有其他类型的异常，由 break、continue 以及 return 命令产生。所有的异常都让活动的脚本以同样的方式放弃，除了两点不同。第一，仅在错误这种异常情况下，才会设置 errorInfo 和 errorCode 变量。第二，错误之外的异常几乎总是会被捕获，而错误常常会展开过程中的所有工作。例如，break 和 continue 命令通常在一个循环命令中调用，如 foreach；foreach 会捕获 break 和 continue 异常，终止循环或跳到下一步迭代。类似地，return 通常在过程或 source 的文件中调用。过程实现以及 source 命令都能捕获 return 异常。

提示： 如果在循环外调用了 break 或 continue，活动的脚本会被展开，直到达到过程的最外层脚本，或者过程中的所有脚本都被展开。这时 Tcl 将 break 或 continue 异常转变为错误进行处理，给出相应的错误信息。

所有的异常都伴随着一个字符串值。对于错误来说，这个字符串就是错误消息。对 return 来说，这个字符串就是过程或脚本的返回值。对于 break 和 continue，这个字符串总是空的。

catch 命令实际捕获了所有的异常，而不仅仅是错误。catch 的返回值表示了发生的异常的类型，catch 命令中的第二个参数被设置为与异常相应的字符串值。

表 13.1 给出了标准异常类型。第一列是各种情况下 catch 的返回值。第二列描述了异常在什么时候出现，以及与该异常相应的字符串的含义。最后一列给出了捕获该类异常的命令(“procedures”表示当 Tcl 过程的整个过程块都被放弃时，该过程将捕获这个异常)。第一行给出的是没有异常发生时的正常返回情况。

表 13.1 Tcl 异常总结

捕获的返回值	说 明	捕获异常的命令
0	正常返回。字符串给出返回值	不用捕获
1	错误。字符串给出对错误的描述	catch
2	return 命令被调用。字符串给出过程或 source 命令的返回值	catch，source，procedures
3	break 命令被调用。字符串为空	catch，for，foreach，while，procedures
4	continue 命令被调用。字符串为空	catch，for，foreach，while，procedures
其他	由用户或实用程序定义	catch

下面这个示例中，catch 命令的返回值为 2，表示它在处理脚本参数时遇到了返回异常。catch 将与该异常相应的字符串存储在变量 string 中。

```
    catch {return "all done"} string
⇒   2
    set string
⇒   all done
```

catch 提供了捕获所有类型的异常的通用机制，return 提供了生成所有类型的异常的通用机制。如果它的第一个参数是关键字-code，例如：

```
return -code return 42
```

它的第二个参数是异常的名称(这里是 return)，第三个参数是与异常相应的字符串。包含它的过程会立即返回，但这不是一个普通的返回，它返回的是由 return 命令的参数描述的异常。上面这个示例中，过程产生了一个返回异常，引起调用过程的返回。

另外，return 命令接受任意多个选项-值对，您可以为选项指定任意的名称。某些选项名称(参见表 13.2)会被特殊对待。所有的选项-值对都会成为返回的选项字典的条目，catch 命令可以通过指定第三个可选的参数来捕获返回的选项字典。这允许您在产生异常的时候传递任意的信息。通常这一功能只用于高级控制结构的实现。

表 13.2 return 命令中有含义的选项名称

return 命令选项的名称	说 明
-code *code*	异常代号，参见表 13.1
-errorcode *list*	如果异常代号是 error，Tcl 将 errorCode 变量的值设为 *list*，如果没有给出 *list*，则将其值设为 NONE

续表

return 命令选项的名称	说　明
-errorinfo *info*	如果异常代号是 error，Tcl 将 errorInfo 变量的初始值设为 *info*，如果没有给出 *info*，则生成追踪堆栈
-level *level*	*level* 的值必须是非负整数。它指定的是当前执行的命令位于堆栈中的 *level* 个层级，该命令的返回值设置为-code 指定的值。level 的默认值为 1
-option *options*	*options* 的值必须是有效的字典。字典的条目会被作为额外的选项-值对添加到返回的选项字典中

9.6 节介绍了如何用 upvar 和 uplevel 把一个新的循环命令 do 实现为 Tcl 过程。然而，9.6 节中给出的示例不能正确处理循环块中出现的异常。下面是改进的 do 命令的实现，使用了 catch 和 return 来进行异常处理。

```
proc do {varName first last body} {
    upvar 1 $varName v
    for {set v $first} {$v <= $last} {incr v} {
        set code [catch {uplevel 1 $body} string options]
        switch -- $code {
            0 -
            4       {}
            3       {return}
            default {
                    dict incr options -level
                    return -options $options $string
            }
        }
    }
}
```

这个新的实现处理异常的方式与 foreach 和 while 这样的内建循环命令相同。它处理 catch 命令中的循环块，然后查看这个循环是如何终止的。如果没有异常(code 为 0)，或者异常是继续(code 为 4)，do 进行下一次迭代。如果出现了中止异常(code 为 3)，do 正常返回它的调用者，结束循环。如果错误、返回或者其他完成代码出现，do 就只是将异常交给它的调用者。

当 do 把异常交给它的调用者时，它要做的主要工作就是把返回的选项字典传给它的调用者。唯一要做的变动是把返回的选项字典中-level 的值加 1。在典型情况下，这就让异常出现在调用者的上下文环境中，而非 do 过程的环境中。如果在运行的代码块中把-level 的值显式地设为异常所处层级，也可以正确处理异常情况。

13.6　后台错误与 bgerror

后台错误是在事件处理器中发生的错误。例如，在执行由 after 命令或某组件的 -command 选项指定的命令时发生的错误，就是后台错误。对于非后台错误，错误可以通过嵌套处理的 Tcl 命令逐层返回，直到到达应用程序的顶层代码，或被一个 catch 命令捕获

处理，前一节讲述了这种情况。另一方面，当后台错误发生时，展开会在 Tcl 库中结束，对 Tcl 来说，没有明确报告错误的方法。

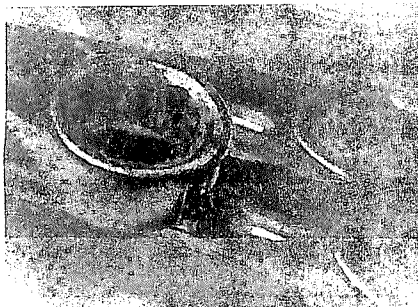
当 Tcl 解释器探测到后台错误时，它存储关于错误的信息，调用一个处理器命令。您可以用 interp bgerror 命令注册后台错误处理器；您的应用程序中的每一个解释器都有它自己的后台错误处理器。如果没有显式地注册后台错误处理器，那么 Tcl 将使用默认的处理器。

为与 Tcl 8.5 以前的版本兼容，默认的处理器会查看解释器中是否有名为 bgerror 的命令。如果有，它就调用 bgerror 命令，传入一个包含错误生成消息的参数；全局变量 errorCode 和 errorInfo 也被设置为它们在错误发生时的值。如果没有注册 bgerror 命令，或在执行 bgerror 命令时发生错误，Tcl 会报告这个错误本身，将其消息写入 stderr 通道。

从 Tcl 8.5 开始，可以使用 interp bgerror 命令注册更加高级的后台错误处理器。interp bgerror 的第一个参数是指向解释器的路径，其定义参见第 15 章；路径{}指向当前解释器。然后，您可以提供更多的参数，指定一个命令和把该命令作为后台错误处理器调用时需要传递给该命令的参数。当调用后台错误处理器时，还会提供两个额外的参数，包含了错误消息和返回的选项字典，13.5 节对此进行了讨论。其结果在解释器的全局命名空间中处理。运行在解释器路径后不带参数的 interp bgerror 会返回当前注册为后台错误处理器的命令前缀。

下面的示例展示了如何注册后台错误处理器，该处理器只是将错误消息和返回的选项字典发送给标准错误通道。

```
proc myHandler {errMsg returnOpts} {
    puts stderr "Background error: $errMsg"
    puts stderr "Return options dictionary:"
    dict for {key value} $returnOpts {
        puts stderr " $key: $value"
    }
}
interp bgerror {} myHandler
after 0 { expr 1/0}
update
⇒ Background error: divide by zero
Return options dictionary:
    -code: 1
    -level: 0
    -errorcode: ARITH DIVZERO {divide by zero}
    -errorinfo: divide by zero
    invoked from within
"expr 1/0"
    ("after" script)
    -errorline: 1
```



第 14 章　创建与使用 Tcl 脚本库

和其他现代编程语言一样，Tcl 允许您将常用的过程定义编组，形成可由很多应用程序使用的库。最简单的 Tcl 库可以是给出了一些工具过程的单个脚本文件。您也可以把多个库脚本文件联合为一个发布的工具包。工具包可以有指定的版本号，这样应用程序就可以分辨不同历史版本的工具包，进行正确的选择。

另外，Tcl 允许您把整个 Tcl 应用程序以及多个库打成一个文件包，称为 Starkit。与 Tclkit 的单文件解释器联合，您可以很容易地发布 Tcl 应用程序。您还可以把应用程序创建成单个可执行文件，这种文件称为 Starpack，它将您的软件所对应的 Starkit 与特定平台的 Tclkit 联合起来。

现代的扩展应该以模块或包的形式实现，并且利用自动加载，自动加载是指 Tcl 解释器会根据需要自动地加载模块或包。为了支持自动加载，Tcl 解释器使用了一些简单的约定，这些约定随着早期版本 Tcl 中对库的使用产生。

后文将介绍如何基于 Tcl 对过程库、对自动加载的支持创建现代的模块和包。最后一节将介绍如何轻松地使用 Tclkit 与 Starkit 或 Starpack 为不同的目标平台发布您的应用程序。

14.1　本章出现的命令

- `auto_mkindex` *`dir ?pattern ...?`*
  生成适于 Tcl 自动加载机制使用的索引。该命令在 *dir* 中搜索其名称与参数 *pattern* 相匹配的文件(由 glob 命令匹配)，生成所有匹配的文件中定义的全部 Tcl 命令过程的索引，然后将索引信息存放在 *dir* 中名为 tclIndex 的文件中。如果没有给出 *pattern*，默认的模式为*.tcl。
- `info library`
  返回存放了标准 Tcl 脚本的库目录名。返回的实际上是全局变量 tcl_library 的值，这个值可通过设置 tcl_library 加以改变。
- `info loaded` *`?interp?`*
  返回一个列表，列表给出使用 load 命令加载到 *interp* 中的所有的包。列表中的每个元素都是有两个元素的子列表，内容包括包是从哪个文件中加载的，以及包的名称。对于静态加载的包，文件名是空字符串。如果没有给定 *interp*，返回进程中所有解释器加载的包。要取得当前解释器所加载的包，应将 *interp* 参数指定为

空字符串。

- `info sharedlibextension`
  返回该平台为共享库使用的文件扩展名。
- `package ifneeded` *`package version script`*
  在 pkgIndex.tcl 文件中使用，指明如果需要的话，*package* 的指定版本 *version* 可用。可选的 *script* 参数在 *package* 被包的 require 命令请求时自动运行。
- `package names`
  返回解释器中所有包的名称列表，解释器可以指定版本(通过 package provide 命令)，或是使用了 package ifneeded 脚本。
- `package prefer ?latest|stable?`
  设置首选的包的类型：latest，表明使用最新版本的包，即使那是测试版本；stable，表明只在没有满足要求的稳定版本时才使用测试版本的包。一旦某个解释器的首选包类型被设置为 latest，那么所有将它设回 stable 的尝试都会被忽略。如果不给出参数，package prefer 返回当前的首选包类型设置。
- `package provide` *`package ?version?`*
  指定在解释器中使用 *version* 版本的 *package* 包。它通常作为 package ifneeded 脚本的一部分被调用，并在包本身被加载后再次调用。如果没有给定版本参数，则命令返回当前提供的版本号，如果没有为解释器中的 *package* 调用过 package provide 命令，则返回空字符串。
- `package require` *`package ?requirement?`*
  `package require -exact` *`package requirement`*
  加载指定的 *package*，确保其版本满足 *requirement*。加载的第一个 *package* 版本是系统中可用的最新版本，只要它满足 *requirement* 指定的最低版本号，并且其主版本号与 *requirement* 相同。如果指定-exact 选项，则表示仅接受 *requirement* 版本号的包。命令返回加载的 *package* 的版本号，如果没有可用的满足 *requirement* 版本要求的包，则返回一个错误。
- `package vcompare` *`version1 version2`*
  比较 *version1* 和 *version2* 给出的两个版本号。如果 *version1* 比 *version2* 早，返回 -1；如果它们相同，返回 0；如果 *version1* 比 *version2* 新，返回 1。
- `package versions` *`package`*
  返回 *package* 的所有版本号的列表，其信息由 package ifneeded 命令提供。
- `pkg_mkIndex ?options? dir ?pattern ...?`
  生成适于 Tcl 包的机制使用的 pkgIndex.tcl 索引。该命令搜索 *dir* 中所有名称与 *pattern* 参数匹配的文件(由 glob 命令匹配)，生成所有匹配的文件中定义的 Tcl 命令过程的索引，将索引信息存储在 *dir* 中名为 pkgIndex.tcl 的文件中。如果没有给定 *pattern*，默认的模式为*.tcl 和*.[info sharedlibextension]。关于所支持的 *options* 的更多信息详见 14.5.2 节及参考文档。

- `::tcl::tm::path list`
  返回一个列表，其内容为所有注册的 Tcl 模块的路径，顺序为模块搜索的顺序。

14.2 load 命令

load 命令用于将预编译的库加载到 Tcl 解释器中，这些库通常由 C 或 C++编写。要这样做，二进制文件必须作为 Unix 的共享库，或 Windows 的动态链接库(DLL)创建。因为不同的平台使用不同的扩展名来描述共享库文件，Tcl 提供了 info sharedlibextension 命令，返回当前系统使用的共享库文件扩展名。因此，共享库通常如下加载：

```
load myextension[info sharedlibextension]
```

C 或 C++代码必须遵循特定的 Tcl 约定。一旦某个共享库实现的包被加载，Tcl 解释器首先调用包中的初始化函数。初始化函数的名称是基于包的名称给定的。例如，名为 star1 的包，其初始化方法名应该是 Star1_Init。如果二进制文件被安全模式 Tcl 解释器加载，则初始化函数名应该是 Star1_SafeInit。有关用 C 语言编写包的更多信息参见第 36 章。

加载了二进制文件后，就可以使用 info loaded 命令列出 load 命令所加载的所有包的列表。

14.3 库的使用

Tcl 库就是包含了一个或多个实现一系列相关过程的 Tcl 脚本文件的目录。Tcl 提供了一个标准过程库，实现了它的某种默认行为。理解 Tcl 如何工作，有利于学习如何添加自己的库和包。

命令 info library 返回 Tcl 库目录的完整路径名称，例如：

```
info library
⇒ C:/Program Files/Tcl8.5/lib/tcl8.5
```

另外，全局变量 tcl_library 也保存着相同的值。

```
puts $tcl_library
⇒ C:/Program Files/Tcl8.5/lib/tcl8.5
```

这个目录用于存放 Tcl 使用的标准脚本，例如下面将讨论的对 unknown 过程的默认定义。

14.4 自动加载

15.10 节讲述 Tcl 如何调用名为 unknown 的过程处理未知命令。对于未知命令，Tcl 有一个默认的实现，不过也可以定义自己的实现过程。默认的未知命令实现的最重要的功能之一就是自动加载。自动加载允许您编写 Tcl 过程集，把它们放置在库目录的脚本文件中。然后您就可以在 Tcl 应用程序中使用这些过程，而不必显式地 source 定义它们的文

件，即您只需要调用它们就行。在第一次调用库过程时它还不存在，因此会调用unknown。unknown会查找定义过程的文件，对该文件应用source命令，然后在原命令中再次调用该过程。下一次调用过程时它就已经存在了，不再触发自动加载机制。

自动加载有两点优势。第一，它使得创建更大规模的过程库和在Tcl脚本中使用它们都更容易了。您不需要知道具体是哪一个文件定义了过程，然后source该文件，自动加载会为您处理好这件事。第二，自动加载提高了效率。如果没有自动加载，那么应用程序在开始的时候就必须source所有它要用到的脚本文件。自动加载则允许一个应用程序在开始运行时不加载任何脚本文件；当文件中的过程被用到时才加载相应文件，而某些文件可能根本不会被加载。因此自动加载缩短了启动时间，节省了内存。

自动加载的使用方法也简单直接。首先，进行库的创建，即创建指定目录中的脚本文件集。通常来说这些文件的名称以.tcl结尾，如db.tcl或stretch.tcl等。每一个文件的内容可以是任意多个过程的定义。实际应用中最好保持文件相对较小，将有关系的几个过程放在一个文件中。为了让自动加载能正确地管理文件，每一个过程的proc命令必须出现在一行的开头，前面不能有空白符，proc后面必须在同一行中立即接着一个空白，然后是过程的名称。否则，尽管它们是有效的Tcl脚本，自动加载机制也无法正确地处理它们。

使用自动加载的第二步是建立索引。为此，可以在tclsh这样的Tcl应用程序中调用auto_mkindex命令，示例如下：

```
auto_mkindex . *.tcl
```

auto_mkindex不是一条内建命令，而是Tcl脚本库中的过程。它的第一个参数是目录名，第二个参数是通配符模式，根据与它的匹配情况选中目录中的脚本文件。auto_mkindex扫描所有名称与指定模式匹配的文件，建立一个索引，记录哪个过程由哪个文件定义。它将这个索引存放在该目录中名为tclIndex的文件中。如果修改脚本文件，添加或删除了过程，就应该重新生成这个索引。

最后一步是在要使用库的应用程序中设置变量auto_path的值。auto_path包含了目录名的一个列表。调用自动加载器时，它会依次搜索中列表中的各个目录，在它们的tclIndex文件中查找需要的过程。如果同名的过程在多个文件中定义，则会加载按auto_path的目录顺序最先找到的那个过程。通常来说，auto_path会在程序启动时加以设置。例如，如果一个应用程序使用路径/usr/local/lib/shapes中的库，在它的启动脚本中很可能包含如下内容：

```
set auto_path\
    [linsert $auto_path 0 /usr/local/lib/shapes]
```

这条命令把/usr/local/lib/shapes加到目录列表的开头，保持了那些已经存在的目录记录，那些Tcl和Tk脚本库仍可使用，但会优先使用/usr/local/lib/shapes定义的过程。一旦一个目录正确地生成了索引，并添加到auto_path中，通过自动加载，它当中的所有过程就成为可用的了。

提示： 更复杂的库(特别是调用Tcl脚本和C语言混合内容的情况，或是需要保持一个库的多个版本的情况)最好通过包的形式实现。当把库发布给他人时推荐使用包，因为它们更易于管理和安装。

14.5 包

库的创建使您能够把有用的工具过程集合起来，创建可重用的库代码。依靠自动加载，Tcl 应用程序可以在需要它们时高效率地加载。但即使如此，Tcl 库还是没有提供很多实用的功能，例如版本管理以及更好的代码分隔方式。例如，如果有两个同名的过程，那么只有 auto_path 中的库路径中首先被找到的那个会被调用。

提示：可以使用命名空间在一定程度上避免这个问题，详见第 10 章所述。

通过包的创建，可以进行版本管理，并且更好地组织库代码。

14.5.1 包的使用

Tcl 本身就包含了大量的包，而且您还可以下载更多。(附录 B 介绍了一些扩展 Tcl 功能的广受欢迎的 Tcl 包。)要使用一个包，需要用 package require 命令指定包的名称和版本，例如：

```
package require platform 1.0.3
```

这条命令试图加载指定的包。一旦成功加载，就可以在 Tcl 应用程序中调用该包中的功能。

一般来说，主号相同的版本被认为是相互兼容的，即如果代码要求 platform 的版本为 1.0，然后要求 1.0.3，那么 1.1、1.2 或其他主号相同的更高版本都满足要求。这种情况下，1.0.4 版本的包满足要求，而 1.0.2 的包不满足。另外，2.0 版本的包也不满足要求。Tcl 包的机制会选择满足要求的包中版本最新的。

如果不关心版本号，可以使用如下命令：

```
package require platform
```

这条命令会把指定包的最新版本加载到您的系统。或者，如果需要确切版本的包，可以使用-exact 选项。

```
package require -exact platform 1.0.3
```

提示：版本号也可以包含 a(表示 α 测试)或 b(表示 β 测试，如 1.3b1)。这些字母表示非稳定的包。没有这些字母的版本号表示稳定版本。Tcl 通常优先使用稳定的版本。您也可以指定加载最新版本的包，即使它是测试版本，命令如下：

```
package prefer latest
```

14.5.2 包的创建

Tcl 包是包含一个或多个 Tcl 脚本和/或二进制共享库的目录。包当中的 Tcl 脚本应该用 package provide 命令为它提供的包进行声明，例如：

```
package provide platform 1.0.3
```

这条命令说明本文件中的源码提供了(如果目录中有多于一个 Tcl 脚本，则是共同提供)名为 platform 的包。版本号为 1.0.3。说明主版本号为 1，次版本号为 0.3。未给出的号视为零，故 1.0.3 与 1.0.3.0 以及 1.0.3.0.0.0 等相同。版本号的长度可以任意选择，例如 1.0.3.0.3.0.3.0.3.0.3，不过大多数情况下都不需要这么长的版本号。您也可以在版本号中增加创建号或时间戳部分，如 1.1.20080829。

提示： 第一个数字，即主版本号，是最为重要的。如果修改之后，新版本不再与早期版本兼容，就应该改变主版本号。

在创建一个包时，先把所有的相关代码都放到一个目录下。然后需要创建 pkgIndex.tcl 文件，Tcl 解释器用它来加载包。您可以手动创建这个文件，而且如果情况十分复杂，可能还必须手动处理。但是大多数包可以使用 pkg_mkIndex 过程。pkg_mkIndex 创建 Tcl 包系统管理包的加载所需要的索引文件。和 auto_mkIndex 一样，pkg_mkIndex 不是内建命令，而是 Tcl 脚本库中的一个过程。

提示： 即使自己编写 pkgIndex.tcl 脚本，也可以使用 pkg_mkIndex 作为向导，它可以告诉您 Tcl 解释器期待在 pkgIndex.tcl 脚本中找到什么。

创建索引的命令的基本格式如下：

```
pkg_mkIndex . *.tcl
```

这条命令查找当前目录中所有名称以.tcl 结尾的文件，在目录中创建文件 pkgIndex.tcl，该文件中包含了包的索引信息。pkgIndex.tcl 中有一个或多个 package ifneeded 命令，以及与其相应的用于包的加载的脚本，例如：

```
package ifneeded app-star1 1.0 \
    [list source [file join $dir star1.tcl]]
```

这条命令指明了名为 app-star1 的包，其版本号为 1.0。加载包的命令即对 Tcl 脚本文件 star1.tcl 应用 source。$dir 的值在 Tcl 解释器加载这个包时传入。第一次处理 package require 命令时，Tcl 解释器执行了 package unknown 脚本。这个脚本对全局变量 auto_path 中列出的所有目录及其直接子目录中的所有 pkgIndex.tcl 文件调用 source。

提示： 所有与平台相关的二进制文件都必须由 load 命令加载，详见 14.2 节。另外，该二进制文件必须包括对 C 函数 Tcl_PkgProvide 的调用。

pkg_mkIndex 支持很多选项，包括-direct，指明在遇到 package require 命令时加载包文件，这一设置即默认设置。-lazy 选项不鼓励使用，它指定在包中的某一个过程被首先调用时，才加载包。-verbose 选项告诉 pkg_mkIndex 在工作时输出状态信息。还有一个特殊的选项，-load，用模式指明对包的预加载，可用于索引包中含有依赖于其他包的二进制文件的情况。

14.5.3 使用::pkg::create

::pkg::create 过程创建了 pkgIndex.tcl 文件中使用的 package ifneeded 命令。您还可以在手动创建 pkgIndex.tcl 文件时使用::pkg::create。一般来说，可以简单地使用 pkg_mkIndex 创建包的索引脚本。基本语法如下：

```
::pkg::create -name package_name\
    -version version_number -source tcl_scripts
```

-source 参数是一个列表，第一个元素是一个文件名，第二个元素是该文件提供的命令列表，例如：

```
    ::pkg::create -name foo -version 99.5 -source star1.tcl
⇒ package ifneeded foo 99.5 [list source [file join $dir star.tcl]]
```

如果有一个二进制文件，可使用-load 选项。

```
::pkg::create -name package_name \
    -version version_number\
    -load {binary_file commands_provided}
```

-load 参数是一个列表，第一个元素是一个文件名，第二个元素是该文件提供的命令列表。

14.5.4 包的安装

要安装新包，就将包的目录作为::tcl_pkgPath 变量中列出的任一目录的子目录。一旦包完成安装，package require 命令就可以找到它。

提示： 习惯上，如果::tcl_pkgPath 变量包含多于一个目录，应该把平台相关的包放在第一个目录中，把平台无关的包放在列表内的第二个目录中。

如果需要把包安装到其他的地方(即::tcl_pkgPath 变量所列出的目录之外)，或者系统没有设置::tcl_pkgPath 变量，就必须把包放在::auto_path 变量列出的某个目录中。通常来说，Tcl 查找初始化代码的目录是::auto_path 列出的目录之一；info library 命令可以返回该目录的名称。当有疑问时，这个目录通常可作为包安装的目标目录。

您还可以把安装了包的目录名添加到::auto_path 变量中。仅在包安装到非标准位置时，才需要这样做。

14.5.5 包的实用命令

package names 命令列出所有安装的包。

```
    package names
⇒ activestate::teapot::link http platform tcl::tommath tcltest msgcat
  ActiveTcl Tcl
```

提示： 这个示例展示了活动的 Tcl 所安装的包。

---

package versions 命令返回系统中可用的包的所有版本号的列表。

```
    package versions mypackage
⇒ 1.0 1.1
```

当 package provide 命令被不带参数地执行时，它返回当前提供的包的版本号，如果解释器没有为包调用过 package provide 命令，则返回空字符串。

```
    package provide Trf
⇒ 2.1.2
```

14.6　Tcl 模块

Tcl 内核使用的定位机制和包加载机制具有很强的适应性，但也有自身的弱点。历史机制的主要问题是它在文件系统中查找包，而且还不得不读取一个文件(pkgIndex.tcl)来获取指定的包的相关信息。所有这些操作都要消耗时间。而“索引脚本”能够扩展搜索路径列表的事实，使得搜索成本更加高昂，因为它会强制要求对文件系统的重新扫描。如果 auto_path 中的安装文件夹很大，或是有挂载远端主机的内容时，情况更加严重(特别是网络延迟明显的情况下)。所有这些原因导致 Tcl 和基于 Tcl 的程序启动缓慢，特别是对于那些有位于远端主机的扩展包的系统而言。

Tcl 8.5 增加了对 Tcl 模块的支持，它没有传统的包那样强的适应性，但大大减少了 Tcl 解释器查找可用的包和模块时对文件系统的访问。对 Tcl 8.5 及更新版本，推荐采用模块方法创建和发表扩展包。

最主要的简化是：每一个模块必须存放在一个文件中，这个文件由 Tcl source 命令读取。在文件中，这个模块可以定义一个或多个 Tcl 包，如 14.5 节讨论的包。一般情况下，一个模块只定义一个包。您可以在同一个文件中一个 Control -z 之后存放二进制数据，Control -z 会结束对 Tcl 命令的读取。另一个简化是：模块没有索引文件(即使用模块时不必创建 tclIndex 或 pkgIndex.tcl 文件)。模块文件的名称和位置足够让解释器明白模块提供的包是什么，版本号是多少。

提示： 有关创建含有二进制数据、嵌套 ZIP 档案以及其他更复杂结构的模块的更多信息，详见 TIP 190 的文档，地址为 http://tip.tcl.tk/189.html，并可在 Tcl 开发者维基 http://wiki.tcl.tk 搜索相关文章。

14.6.1　使用 Tcl 模块

要使用 Tcl 模块定义的包，就在 Tcl 程序中使用 package require 命令，和使用非模块中的包是一样的，例如：

```
package require platform::shell 1.1.3
```

Tcl 解释器首先在 Tcl 模块文件中查找包。Tcl 根据“模块路径”中的目录列表进行搜索，该路径与 14.5.4 节中描述的 auto_path 列表中的目录没有关联。仅在不能找到适合的 Tcl 模块之后，Tcl 才使用传统方法扫描 auto_path 列出的目录，查找符合要求的包。

14.6.2 安装 Tcl 模块

在创建模块时，存放它们的文件名必须与以下正则表达式匹配。

```
([[:alpha:]][:[:alnum:]]*)-([[:digit:]].*)\.tm
```

文件名的第一部分指明了包的名称，如 myutils。文件名中-后面的部分是版本号。模块文件名必须以.tm 扩展名结尾。下面列出的就是一些有效的模块文件名：

```
tcltest-2.3.0.tm
http-2.7.tm
shell-1.1.3.tm
myutils-1.1.20080829.tm
```

存放 Tcl 模块的目录树与 auto_path 相互独立。Tcl 查找模块时会查找::tcl::tm::path list 命令列出的目录。这个目录称为模块路径。这里不会使用 auto_path 和 tcl_pkgPath 变量。

当一个应用程序请求一个包时，在开始搜索之前，所请求的包的名称会翻译为部分路径，包的名称中出现的所有::都会由该平台所用的目录分隔符替换。因此在 Unix 中，请求的包的名称中的::翻译为/。例如：

```
package require encoding::base64
```

结果是部分路径 encoding/base64。在翻译之后，Tcl 把所得的部分路径串接到模块路径列表(这个列表的内容是 Tcl 查找模块的目录)中的每一个目录后面。然后依次对各个路径进行模式匹配检查，查找所需的模块文件。

默认的模块路径列表由 tclsh 基于 Tcl 解释器的主版本号和次版本号给出。因此，用 *X* 表示主版本号，用 *Y* 表示小于或等于当前次版本号的次版本号，默认的路径列表包含如下内容。

- `file normalize [info library]/../tcl`*X*`/`*X*`.`*Y*
  `file normalize [info nameofexecutable]/../lib/tcl`*X*`/`*X*`.`*Y*
  因此，Tcl 8.5 版的解释器会包含一系列目录：tcl8/8.5、tcl8/8.4、tcl8/8.3、tcl8/8.2、tcl8/8.1 以及 tcl8/8.0，其中 tcl8 目录与 info library 返回的 Tcl 库目录处在同一层。
- `file normalize [info library]/../tcl`*X*`/site-tcl`
  注意这总是一个条目，因为 *X* 是当前 Tcl 的主版本号。
- `$::env(TCL`*X*`_`*Y*`_TM_PATH)`
  一个路径列表，在 Unix 中用:隔开，在 Windows 中用;隔开。注意 *X* 和 *Y* 的含义与上面所说的相同，所以对于 Tcl 8.5，这意味着 6 个不同的环境变量，从 TCL8_5_TM_PATH 到 TCL8_0_TM_PATH。

命令::tcl::tm::path list 返回当前的模块路径列表。可以使用::tcl::tm::path add 和::tcl::tm::path remove 命令为这个列表添加或删除目录。更多信息参见 tm 的参考文档。

作为示例，假设您已经创建了一个包的第 1 个版本，其他人会用如下命令把它加载到自己的应用程序中。

```
package require struct::btree
```

再假设您是用 Tcl 8.4 功能设计的那个包，它不会用到 Tcl 8.5 的功能。如果这个模块只是供您自己使用，您可能会把它放在 home 目录的子目录中。假设您的 home 目录是/Users/ken，那么模块的完整路径名应该如下：

```
/Users/ken/modules/struct/btree-1.0.tm
```

您可以创建名为 TCL8_4_TM_PATH 的环境变量，将/Users/ken/modules 包含为它的一个值。

另一方面，如果模块是要被系统中的所有用户共同使用的，而 Tcl 的安装位置使 info library 命令返回 C:\Tcl 8.5\lib，那么，模块的完整路径名应该如下：

```
C:\Tcl8.5\lib\tcl8\site-tcl\struct\btree-1.0.tm
```

如果计划发布这个模块，可以创建一个简单的安装脚本，由它查询 info library 命令，然后安装模块。

```
file normalize [info library]/../tcl8/8.4/struct/btree-1.0.tm
```

14.7　把脚本打包为 Starkit

Starkit 提供单个文件发布的方式，发布您的 Tcl 过程、应用程序数据以及(如果需要)平台相应的编译过的代码。这允许您把整个 Tcl 应用程序打包成一个文件。

提示： Star 的名称来自“独立运行”(stand-alone runtime)的缩写。

Starkit 本身使用了 Tcl 的虚拟文件系统支持，将应用程序呈现为一个普通的目录，用来定位代码、应用程序数据以及其他文件。Starkit 中的文件对应用程序来说是作为普通文件出现的，使应用程序免于打包。

要运行 Starkit 中的应用程序，需要使用一种特殊的 Tcl 解释器，称为 Tclkit。一个 Tclkit 就是由一个平台相关的 Tcl 解释器和特定的 Tcl 扩展包(需要用来从 Starkit 的虚拟文件系统中取得和运行应用程序)创建的。普通安装的 Tcl 包括数百个文件，而使用 Tclkit 您可以由单个文件运行整个发行版 Tcl。

提示： 作为单文件发布机制，Starkit 和 Java 的 jar 文件类似；Tclkit 与 Java 的 JRE(或者叫 Java 运行引擎)类似。主要的不同在于，运行 Starkit 时，是直接使用与特定平台相应的 Tclkit 解释器。

联合应用 Tclkit/Starkit 提供了最为便利的打包与发布方法，对 Tcl 应用程序的用户十分便利。如果没有 Tclkit 和 Starkit 功能，需要完成如下准备才能发布 Tcl 应用程序。

- Tcl 解释器
- 平台相关的编译过的库
- 构成 Tcl 标准库的 Tcl 脚本
- 应用程序需要的所有 Tcl 扩展包

- 不是标准发布的 Tcl 库的一部分，但应用程序要使用的所有 Tcl 库的代码
- 应用程序脚本
- 应用程序需要的所有平台相关的编译过的代码
- 应用程序需要的所有数据文件

Starkit 系统将这些项分解为两个包。Tclkit 提供了 Tcl 解释器以及 Tcl 扩展包，包括运行 Starkit 应用程序所需要的一切。您创建的 Starkit，包含了应用程序脚本、所需的平台相关的编译过的代码以及应用程序的数据。

后文将演示如何创建和使用 Starkit。要创建 Starkit，首先要安装一个 Tcl 解释器。

14.7.1 安装 Tclkit

从 http://www.equi4.com/tclkit/download.html 为您的平台下载 Tclkit 解释器。每一个 Tclkit 解释器都包括 Tcl、Tk、IncrTcl、Metakit(小型嵌入式数据库)以及支持虚拟文件系统的 TclVFS。

提示：有关 Tclkit 的更多信息参见 http://www.equi4.com/tclkit/。

通常，只需要解压下载的文件，就拥有了完整的 Tclkit 解释器。您可以像调用 tclsh 那样调用 Tclkit 解释器，用它运行普通的 Tcl 脚本应用程序以及 Starkit 应用程序。用 Tclkit 运行 Starkit 应用程序的方式与标准的 Tcl 脚本很相近。

```
tclkit starkitfile ?arguments_to_starkit ...?
```

14.7.2 创建 Starkit

创建 Starkit 的下一步是下载 sdx Starkit。sdx 是设计用于帮助您创建自定义 Starkit 的 Starkit，可用来将应用程序打包成 Starkit，对 Starkit 解包，或显示 Starkit 的内容信息。从 http://www.equi4.com/pub/sk/sdx.kit 下载 sdx，更多信息参见 http://wiki.tcl.tk/sdx。

提示：在进行与 Starkit 有关的工作时保持 sdx.kit 可用。

让我们快速开始，您可以用 sdx gwrap 命令为单个的 Tcl 脚本创建 Starkit。例如，创建下面这句 Tcl 脚本，存放在名为 star1.tcl 的文件中。

```
puts "Hello from a Starkit."
```

然后运行如下命令为这个很短的 Tcl 脚本创建 Starkit。

```
tclkit sdx.kit qwrap star1.tcl
```

这条命令创建了一个名为 star1.kit 的小文件。用如下命令运行这个新的 Starkit。

```
    tclkit star1.kit
⇒   Hello from a Starkit.
```

运行 sdx lsk 命令查看 Starkit 文件中的目录结构。

```
    tclkit sdx.kit lsk star1.kit
⇒ star1.kit:
⇒                          dir  lib/
```

```
⇒      75 2008/06/19 19:25:21   main.tcl
⇒
⇒star1.kit/lib:
⇒                        dir  app-star1/
⇒
⇒star1.kit/lib/app-star1:
⇒      76 2008/06/19 19:25:21   pkgIndex.tcl
⇒      62 2008/06/19 19:25:21   star1.tcl
```

您可以用 sdx unwrap 命令取得这些文件。

```
    tclkit sdx.kit unwrap star1.kit
⇒  5 updates applied
```

这条命令创建了 Starkit 文件中的虚拟文件系统的一份本地文件副本，存放在 star1.vfs 目录中。在该目录中您会看到如下文件结构：

```
main.tcl
lib/
lib/app-star1/
lib/app-star1/pkgIndex.tcl
lib/app-star1/star1.tcl
```

名称 star1 来自源文件的名称 star1.tcl。按照 Starkit 的约定，为应用程序创建的包会使用前缀 app-。

要运行展开的 Starkit，使用如下命令：

```
    tclkit star1.vfs/main.tcl
⇒  Hello from a Starkit.
```

main.tcl 脚本包含如下代码：

```
package require starkit
starkit::startup
package require app-star1
```

starkit 包提供了必要的代码来初始化 Starkit 虚拟文件系统。starkit::startup 过程将内部的 Starkit lib 目录添加到 Tcl auto_path 变量中。这使得 Starkit 中的所有包都对应用程序可用。您可以将包添加到 lib 目录，然后从脚本中正常访问这些包。最后，package require 命令对程序的脚本应用 source。

pkgIndex.tcl 脚本定义了包的索引。

```
package ifneeded app-star1 1.0 \
    [list source [file join $dir star1.tcl]]
```

而 sdx qwrap 命令修改原始 Tcl 脚本，让它包含一条 package provide 命令。

```
package provide app-star1 1.0
puts "Helloe from a Starkit."
```

为一个包含多个 Tcl 脚本和其他扩展，编码、图形文件、数据文件以及其他辅助文件的应用程序创建 Starkit 则困难得多。首先，需要创建一个目录结构，类似于前面 sdx unwrap 命令在解包 Starkit 时产生的那个目录结构。顶层目录名就是应用程序名加上.vfs。里面应该有一个名为 main.tcl 的文件，在运行 Starkit 时会自动执行它。该文件的内容应该和前面展示的相同，除了最后一句 package require 指定的应该是您的应用程序“包”的名

称。您的应用程序主脚本应该包含适当的 package provide 命令。您还必须有一个 pkgIndex.tcl 文件，描述如何 source 主脚本。然后，您的脚本可以用 source、package require 以及其他 Tcl 命令从.vfs 目录结构中加载脚本和其他文件。一旦完成了应用程序开发，就可以使用 sdx wrap 命令把.vfs 目录中的所有文件打包为一个 Starkit。

```
tclkit sdx.kit wrap star1.kit
```

提示： 您还可以把平台相关的 C 语言扩展也打包进 Starkit。有关包括更多文件，包括基于 C 语言的扩展的信息，请参阅 http://www.equi4.com/tclkit/处的文档。

14.7.3 创建平台相关的可执行文件

在使用 Starkit 时，需要不同的平台相关的 Tclkit 解释器来运行应用程序。Starpack 则是平台相关的 Tclkit 解释器和 Starkit 的结合，允许您为用户提供单文件的应用程序。其优点在于，用户不必知道您是用 Tcl 编写的程序，而且用户也不必关心与平台相关的运行技术。

要创建一个 Starpack，应该使用 sdx wrap 命令，并指定-runtime 选项。

```
    tclkit sdx.kit wrap star1 -runtime tclkit-darwin-univ-aqua
⇒ 4 updates applied
```

这条命令中，star1 是输出的可执行文件的名称。在 Windows 中其名称为 star1.exe。-runtime 选项指示 Tclkit 解释器要创建新的可执行文件。您不能用正在运行 sdx 命令的 Tclkit 解释器创建 Starpack。当然，您总是可以使用 Tclkit 的一份副本。

创建完成后，就可以使用新命令了。例如：

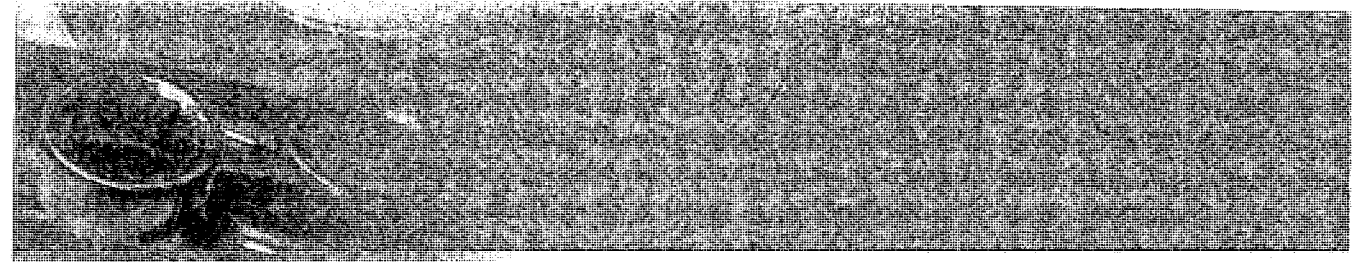
```
    ./star1
⇒ Hello from a Starkit.
```

在 Unix 系统中，可以使用 file 命令来校验刚才创建的文件。例如：

```
    file star1
⇒ star1: Mach-O universal binary with 2 architectures
⇒ star1 (for architecture ppc):  Mach-O executable ppc
⇒ star1 (for architecture i386): Mach-O executable i386
```

对于 Mac OS X，这种情况下会创建可在 PowerPC 和 Mac OS X 下的 Intel 平台中运行的可执行文件。

有关 Starkit 的更多信息，参见 http://www.equi4.com/papers/skpaper1.html。您还可以下载其他的 Starkit，然后用 sdx unwrap 命令看看里面都有些什么。



第15章 Tcl 内部管理

本章讲述的内部管理命令，用于查询和修改 Tcl 解释器的内部状态，并且对特定的任务创建专用的替换解释器。例如，可以使用这些命令查看一个变量是否存在，监视对一个变量或命令的所有使用，重命名或删除一个命令，处理对未定义的命令的引用，创建一个适用于运行处理来自非信任源数据的脚本的解释器。本章还讨论 Tcl 关于时间和日期操作的功能、测量代码执行的用时、暂时性中断脚本运行以及预定在一定延迟后发生的动作。

15.1 本章出现的命令

本章讨论的用于解释器内部管理的 Tcl 命令如下。

- `after` *ms*
  休眠 *ms* 毫秒然后返回。
- `after` *ms* `?`*script script ...*`?`
  将 *script* 参数以 concat 命令的串接方式串接起来，*ms* 毫秒后在全局空间处理得到的脚本。返回描述注册的脚本的独一无二的标志。
- `after cancel` *id*
  取消对前面用 after 注册的延迟运行的脚本的执行。*id* 是注册脚本时 after 返回的独一无二的描述符。如果脚本已经在运行，则 after cancel 命令没有效果。
- `after idle` *script* `?`*script ...* `?`
  将 *script* 参数以 concat 命令的串接方式串接起来，休眠回调时在全局空间处理得到的脚本。这个脚本只会运行一次，下一次进行事件循环时不再处理。返回描述注册的脚本的独一无二的标志。
- `clock seconds`
  `clock microseconds`
  `clock milliseconds`
  返回整型值，为从新纪元时间(参见 15.3 节)开始到当前的秒数、毫秒数或微秒数。
- `clock format` *timeVal* `?`*-option value ...*`?`
  接受一个 *timeVal* 参数，为从新纪元时间开始的秒数，返回时间和日期字符串，供用户或外部程序使用。更多细节参见本章正文和参考文档。

- `clock scan inputString ?-option value ...?`
  接受由用户或外部程度提供的时间和日期字符串，返回从新纪元时间开始的秒数。如果字符串不能被转化时产生一个错误。更多细节参见本章正文和参考文档。
- `clock add timeVal ?count unit ...? ?-option value ...?`
  接受一个 *timeVal* 参数，为从新纪元时间开始的秒数，然后在其值上加上 *count* 单位的时间(*count* 值可以为负)，返回的时间结果为整型数。更多细节参见本章正文和参考文档。
- `info args procName`
  返回一个列表，其元素为 *procName* 过程的参数，依次排列。
- `info body procName`
  返回名为 *procName* 的过程的过程块。
- `info cmdcount`
  返回解释器已经执行的 Tcl 命令的总条数。
- `info commands ?pattern?`
  返回当前命名空间或指定命名空间内所有可用的命令的列表，包括内建命令、应用程序定义命令、过程、集合命令以及别名。如果指明了 *pattern*，仅返回与 *pattern* 匹配的命令；匹配遵循 string match 的规则。如果 *pattern* 包含绝对或相对命名空间引用，则仅列出指定命名空间中与 pattern 匹配的命令。
- `info complete script`
  如果 *script* 是语法上完整的一条或多条 Tcl 命令，则返回 1，否则返回 0。该命令仅检查脚本中的命令是否有配对的括号、引号等；它不能确保提供给命令的参数格式正确。
- `info default procName argName varName`
  检查 *procName* 过程的 *argName* 参数是否有默认值。如果有，将默认值存放到变量 *varName* 中，返回 1。否则，返回 0，设置 *varName* 为空字符串。
- `info exists varName`
  如果当前上下文环境中存在名为 *varName* 的变量，返回 1，如果当前可访问的变量中没有它，则返回 0。
- `info globals ?pattern?`
  返回当前定义的所有全局变量的列表。如果给定 *pattern*，则仅返回其名称与 *pattern* 匹配的全局变量，匹配遵循 string match 规则。
- `info hostname`
  返回运行 Tcl 解释器的机器的主机名。
- `info level ?number?`
  如果没有给出 *number*，返回当前的堆栈层级号(0 表示顶层，1 表示第一层过程调用，以此类推)。如果给定了 *number*，则返回一个列表，其元素为 *number* 层级调用过程的名称和参数。

- `info library`
  返回存放标准 Tcl 脚本的库目录的完整路径名称。
- `info locals ?pattern?`
  返回一个列表，其内容为当前过程定义的所有局部变量，如果当前没有活动的过程，则返回空列表。如果给定了 *pattern*，仅返回其名称与 *pattern* 匹配的局部变量，匹配遵循 string match 规则。
- `info nameofexecutable`
  返回包含 Tcl 解释器的可执行文件的完整名称。
- `info patchlevel`
  返回 Tcl 解释器的当前版本。
- `info proce ?pattern?`
  返回当前命名空间中定义的所有过程名称的列表。如果给定了 *pattern*，仅返回其名称与 *pattern* 匹配的过程，匹配遵循 string match 规则。如果 *pattern* 包含绝对或相对命名空间引用，则仅列出指定命名空间中与 *pattern* 匹配的过程。
- `info script ?filename?`
  如果没有给出 *filename*，返回正在处理的脚本文件名，如果没有脚本文件正被处理，则返回空字符串。如果指定了 *filename*，则以后不带参数的 info script 返回的值被设定为 *filename*，直到过程结束。
- `info sharedlibextension`
  返回当前平台下共享库的默认扩展文件名。
- `info tclversion`
  以 *major.minor* 格式返回 Tcl 解释器的版本号。
- `info vars ?pattern?`
  返回当前可以访问的所有变量名的列表。如果给定了 *pattern*，则仅返回其名称与 pattern 匹配的变量，匹配遵循 string match 规则。
- `interp alias srcPath srcCmd ?targetPath? ?targetCmd? ?arg arg ...?`
  创建、删除和操作解释器间及解释器内的命令别名。如果没有给出 *targetPath* 和后面的参数，返回解释器 *srcPath*(如果为空，表示当前解释器)中名为 *srcCmd* 的命令别名的定义。如果 *targetPath* 作为空字符串给出，又没有给出更多的参数，则删除 *srcPath* 解释器中的 *srcCmd* 别名。如果给出了 *targetPath* 和更多的参数，则 *srcPath* 解释器中的 *srcCmd* 别名设定为指向 *targetPath* 解释器中的 *targetCmd* 命令，这里提供的更多的参数在调用者提供的参数之前传入。
- `interp creats ?-safe? ?--? ?path?`
  创建由 *path* 指定的解释器，并返回解释器的名称。参数--表示选项结束，如果 *path* 有可能以-开头时一定要给出这个参数。选项-safe 参数返回的是安全的解释器，该解释器中所有非安全的内核命令都被隐藏。如果没有给出 *path*，interp 命令用自动生成的名称创建当前解释器的直接子解释器。

- `interp delete path`
  删除 *path* 指向的解释器以及它的所有从属解释器。
- `interp eval path arg ?arg ...?`
  将给出的 *arg* 参数串接起来，由 *path* 指向的解释器将串接的结果作为脚本处理。
- `interp expose path hiddenName ?cmdName?`
  取消对 *path* 解释器中 *hiddenName* 命令的隐藏。如果给出 *cmdName*，则命令以 *cmdName* 展现，否则命令以其自身的名称展现。
- `interp hide path cmdName ?hiddenName?`
  隐藏 *path* 解释器中 *cmdName* 命令。如果给出 *hiddenName*，则命令是以 *hiddenName* 隐藏，否则命令以其自身的名称隐藏。
- `interp invokehidden path ?-global? hiddenName ?arg ...?`
  调用 *path* 解释器中名为 *hiddenName* 的隐藏命令，将后面的 *arg* 作为隐藏命令本身的参数转入。
- `interp limit path limitType ?option? ?value? ...`
  配置 *path* 解释器 *limitType* 类型的运行限制。
- `interp recursionlimit path ?newLimit?`
  查询和设置 *path* 解释器的递归限制(命令嵌套调用的最大深度)。不给出 *newLimit* 则查询当前限制，给出 *newLimit* 则将其设置为新的限制值。注意基于可用的堆栈空间大小，在递归限制问题上 Tcl 还受到硬件限制。
- `interp share srcPath channelID targetPath`
  把 *srcPath* 解释器中的 *channelID* 通道关联到 *targetPath* 解释器中的同名变量，即也可通过这个同名变量使用 *srcPath* 解释器中的 *channelID* 通道。
- `interp transfer srcPath channelID targetPath`
  把 *srcPath* 解释器中的 *channelID* 通道关联到 *targetPath* 解释器中的同名变量，可通过这个同名变量使用 *srcPath* 解释器中的 *channelID* 通道。但此后 *srcPath* 解释器不可使用该通道。
- `rename old new`
  把命令 *old* 重命名为 *new*，如果 *new* 是空字符串，则删除 *old*。返回空字符串。
- `time script ?count?`
  运行 *script* 脚本 *count* 次，返回的字符串是平均运行时间，单位为微秒，*count* 默认值为 1。
- `trace add command name ops scriptPrefix`
  建立对 *name* 命令的跟踪，当 *name* 的 *ops* 选项指定的操作进行时调用 *scriptPrefix*。*ops* 必须由列表构成，列表至少包含以下单词之一：delete 和 rename。返回空字符串。
- `trace add execution name ops scriptPrefix`
  建立对 *name* 命令的跟踪，当 *name* 的 *ops* 选项指定的操作进行时调用 *scriptPrefix*。*ops* 必须由列表构成，列表至少包含以下单词之一：enter、leave、

enterstep 和 leavestep。返回空字符串。

- `trace add variable` *`name ops scriptPrefix`*
  建立对 *name* 变量的跟踪，对 *name* 进行 *ops* 选项指定的操作时调用 *scriptPrefix*。*ops* 必须由列表构成，列表至少包含以下单词之一：array、read、write 和 unset。返回空字符串。
- `trace info command` *`name`*
  `trace info execution` *`name`*
  `trace info variable` *`name`*
  返回一个列表，描述对 *name* 命令或变量设置的所有跟踪。列表中的每一个元素都是有两个元素的列表：第一个元素是跟踪的操作名，第二个元素是该跟踪使用的脚本。
- `trace remove command` *`name ops scriptPrefix`*
  `trace remove execution` *`name ops scriptPrefix`*
  `trace remove variable` *`name ops scriptPrefix`*
  如果对 *name* 命令或变量建有跟踪，操作和脚本分别由 *ops* 和 *scriptPrefix* 指定，则移除跟踪，*scriptPrefix* 以后不再被调用。返回空字符串。如果 *name* 不存在，则抛出一个错误。
- `unknown` *`cmd ?arg arg ...?`*
  该命令由 Tcl 解释器在遇到未知命令时调用。*cmd* 是未知命令的名称，*arg* 是传给该命令的所有参数。unknown 返回的结果作为未知命令 *cmd* 的结果。

15.2 时间延迟

after 命令允许您在应用程序中结合时间控制。它有两种格式。如果给出一个参数调用 after，则该参数以毫秒为单位指定了延迟，命令则延迟指定的时间，然后返回。例如：

```
after 500
```

就延迟 500 毫秒，然后返回。这种格式的 after 命令可用于阻塞，这样在命令执行时应用程序不会对事件做出响应。而如果给出了更多的参数，如下面这条命令：

```
after 5000 {puts "Time's up!"}
```

则 after 不作延迟，立即返回。然而，它像 concat 命令一样把后面的参数串接起来，在指定的延迟时间后执行所得到的脚本。该脚本作为事件处理器在全局空间中处理，类似用于组件创建和文件事件的脚本。上面这个示例，在 5 秒钟后向标准输出显示一条消息。after 命令可以一次给定任意条脚本命令。

after idle 命令注册一个休眠回调。

```
after idle script ? script ... ?
```

在这种格式的 after 命令下，*script* 参数以 concat 的处理方式串接起来，得到的脚本只运行一次，下一次进行事件循环时则没有要处理的事件。这种格式的 after 命令也会立即返回。

后面两个 after 命令的返回值通常是表示注册的脚本的独一无二的描述符。如果脚本还没有运行，可以将其描述符作为参数传给 after cancel 命令，取消相应脚本。

提示： after 命令的最后一种格式要求事件循环在活动，才能调用注册的处理器。有关事件循环和事件驱动的编程详见 12.6 节。

作为在事件驱动的应用程序中使用 after 命令的示例，下面这段脚本使用 after 创建了通用闪烁工具。

```
proc blink {w option value1 value2 interval} {
    $w config $option $value1
    after $interval [list blink $w $option \
        $value2 $value1 $interval]
}
blink .b -bg red black 500
```

blink 过程获取 5 个参数，分别是组件名称、组件的选项名称、该选项的两个值和以毫秒为单位的闪烁间隔。这个过程让选项的值以指定的闪烁间隔在给出的两个值之间交替变换。它首先将选项设为第一个值，然后在指定的闪烁间隔后将选项设为给定的第二个值。这个过程在每次执行时都再次预定运行，因此它会持续运行下去。blink 在“后台”运行：它总是立即返回，然后在指定的延时后由 Tk 的计时器代码再调用。这使得应用程序可以在出现闪烁的时候做其他的事。

15.3 时间和日期操作

Tcl 的 clock 命令提供了很多的时间和日期操作工具，在 Tcl 中，大多数与时间相关的命令以新纪元时间为参照物来表示时间，即从 UTC 时间 1970 年 1 月 1 日零时到现在的秒数。

clock seconds 命令返回当前时刻，以整型数表达从新纪元时刻到当前时刻的秒数，这是在 Tcl 中最为常用的基于新纪元时间的格式。clock milliseconds 和 clock microseconds 返回当前相对于新纪元的时间，单位分别为毫秒和微秒。

15.3.1 产生可读的时间和日期字符串

clock format 命令接受一个表示从新纪元时刻开始的秒数的整数，返回可读的时间和日期字符串。

```
clock format time ?-option value ... ?
```

clock format 的返回值的默认格式如下：

```
    clock format [clock seconds]
⇒ Fri Aug 29 13:05:00 PDT 2008
```

如果应用了-format 选项，那么它之后的字符串指定了日期和时间的格式。该字符串由任意个除%号外的字符，和任意个格式组构成，格式组是以%号开头的长为两个字符的序列。Tcl 支持很多的格式组，Tcl 8.5 中为了支持本地化又加入了很多。表 15.1 列出了较常

用的格式组。所有的格式组列表参见 clock 参考文档。

表 15.1　常用的 clock 格式组

格 式 组	说　明
%a	简写的星期几的名称(Mon…)
%A	完整的星期几的名称(Monday…)
%b %h	简写的月份名称
%B	完整的月份名称
%d	日，两位十进制数
%e	日，一位或两位十进制数，一位十进制数时以空白开头
%H	以 24 小时制给出的该日的小时数(00～23)，两位十进制数
%I	以 12 小时制给出的该日的小时数(12～11，即 12，1，2，…，11)，两位十进制数
%j	给出该日是该年的第几天(001～366)，三位十进制数
%k	以 24 小时制给出的该日的小时数(0～23)，一位或两位十进制数
%l	以 12 小时制给出的该日的小时数(12～11，即 12，1，2，…，11)，一位或两位十进制数
%m	月份(01～12)，两位十进制数
%M	该小时的分钟数(00～59)，两位十进制数
%N	月份(1～12)，一位或两位十进制数，一位十进制数时以空白开头
%p	小写的该日的分段，如 am 或 pm
%P	大写的该日的分段，如 AM 或 PM
%S	该分钟的秒数(00～59)，两位十进制数
%t	制表符
%T	与%H:%M:%S 同义
%u	星期几(1 表示星期一，7 表示星期日)
%w	星期几(0 表示星期日，6 表示星期六)
%y	该世纪的年数，两位十进制数；从 1938 年到 2037 年。
%Y	四位日历年数
%z	当前时区，以小时分钟形式表示格林尼治东(+hhmm)区或西(-hhmm)区
%Z	当前时区名称
%%	文本字符%
%+	与%a %b %c %H:%M:%S %Z %Y 同义

下面是一些示例：

```
    set time [clock seconds]
    clock format $time -format "%d %b %Y"
⇒   29 Aug 2008
    clock format $time -format "%I:%M:%S %p"
⇒   03:05:00 PM
```

从 Tcl 8.5 开始，可以使用-timezone 选项指定进行日期时间格式处理的时区。时区可以用多种标准格式表示，详见 clock 参考文档。

```
    clock format $time
⇒   Fri Aug 29 15:05:00 CDT 2008
    clock format $time -timezone ":Europe/Berlin"
⇒   Fri Aug 29 22:05:00 CEST 2008
    clock format $time -timezone ":UTC"
⇒   Fri Aug 29 20:05:00 UTC 2008
```

Tcl 8.5 还引入了时间和日期的本地化。默认情况下，clock format 返回的时间和日期字符串根据当前系统进行本地化。通过指明-locale 选项，可以直接指定不同的区域设置，clock format 会根据区域设置生成相应的当地时间和日期。默认情况下已经支持很多区域设置选项；如何添加自己的本地化方案详见 clock 的参考文档。

```
    clock format $time -format "%A, %B %d %Y" -locale de
⇒   Freitag, August 29 2008
    clock format $time -format "%A, %B %d %Y" -locale pt
⇒   Sexta-feira, Agosto 29 2008
    clock format $time -format "%A, %B %d %Y" -locale fr
⇒   vendredi, août 29 2008
```

15.3.2 扫描可读的时间和日期字符串

clock scan 命令接受可读的时间和日期字符串，返回新纪元时刻到该时间的秒数。

```
clock format inputString ?-option value ...?
```

为与 Tcl 8.5 以前的版本兼容，clock scan 命令提供了在没有规则可循的情况下的试探式字符串解析功能，例如：

```
    clock format [clock scan "3:37 pm"]
⇒   Fri Oct 31 15:37:00 CDT 2008
    clock format [clock scan "August 29, 1965"]
⇒   Sun Aug 29 00:00:00 CDT 1965
    clock format [clock scan "4:15 pm February 8, 2037"]
⇒   Sun Feb 08 16:15:00 CST 2037
```

然而，在 Tcl 8.5 及以后的版本中，为了减少出现歧义的可能，应该为 clock scan 命令提供一个-format 选项，以及描述输入格式的字符串。该字符串由任意个除%号外的字符，和任意个格式组构成。格式组与前文中 clock format 命令支持的格式组相同。如果传入的字符串格式与指定格式不符，clock scan 会产生一个错误。

```
    clock format [clock scan "Sept 9, 1961" -format "%B %e, %Y"]
⇒   Sat Sep 09 00:00:00 CDT 1961
    clock format [clock scan "20080829T142305" \
        -format "%Y%m%dT%H%M%S"]
⇒   Fri Aug 29 14:23:05 CDT 2008
    clock scan "1:23am" -format "%B"
Ø   input string does not match supplied format
```

提示： 一般情况下，clock scan 的格式组的解析不像 clock format 那样严格。例如，%d 在 clock format 中表示日，为两位十进制数，如果必要在第一位补零；而在 clock scan 中，它也能与以空白开头的一位十进制数表示的日匹配。类似地，在 clock format 中%B 提供完整的月份名，而 clock scan 允许简写的或完整的月份名，以及这两种月份名格式中的可区分的词头。

Tcl 8.5 及以后的版本中，clock scan 命令还支持本地化的时间和日期表达。默认情况下，时间和日期字符串根据系统区域设置进行。不过，可以联合使用-locale 选项和-format 选项为字符串解析器指定不同的区域设置。

```
    clock scan "Sexta-feira, Agosto 29 2008" \
        -format "%A, %B %d %Y"
∅   input string does not match supplied format
    clock scan "Sexta-feira, Agosto 29 2008" \
        -format "%A, %B %d %Y" -locale pt
⇒ 1219986000
    clock scan "vendredi, août 29 2008" \
        -format "%A, %B %d %Y" -locale fr
⇒ 1219986000
```

15.3.3 进行时间计算

为了与 Tcl 8.5 以前的版本兼容，clock scan 命令支持一些简单的时间计算。除了使用一系列关键字如 now、today、yesterday 以及 tomorrow 之外，可以对任意单位的时间进行加减操作，例如：

```
    clock format [clock scan "now"]
⇒   Tue Jan 27 02:49:10 CST 2009
    clock format [clock scan "12:5pm yesterday"]
⇒   Mon Jan 26 12:15:00 CST 2009
    clock format [clock scan "tomorrow - 2 months + 3 weeks"]
⇒   Fri Dec 19 00:00:00 CST 2008
```

Tcl 8.5 引入了 clock add 命令，以消除在 clock scan 命令中进行时间计算可能引起的含义不明。

```
clock add timeVal ?count unit ...? ?-option value ...?
```

clock add 的第一个参数是从新纪元时刻开始的秒数。后面的参数依次分别是整数和关键字，关键字可以从 seconds、minutes、hours、days、weeks、months 或 years，或这些单词的独一无二的可区分词头中选择。整数指定了要加上(或减去，如果整数为负)的时间。

```
    set time [clock scan "2008-01-30 05:00:00" \
        -format "%Y-%m-%d %H:%M:%S"]
    clock format $time
⇒   Wed Jan 30 05:00:00 CST 2008
    clock format [clock add $time 5 weeks]
⇒   Wed Mar 05 05:00:00 CST 2008
    clock format [clock add $time 1 month]
⇒   Fri Feb 29 05:00:00 CST 2008
    set time [clock scan {2004-10-30 05:00:00} \
        -format {%Y-%m-%d %H:%M:%S}]
    clock format $time
⇒   Sat Oct 30 05:00:00 CDT 2004
    clock format [clock add $time 1 day]
⇒   Sun Oct 31 05:00:00 CST 2004
    clock format [clock add $time 24 hours]
⇒   Sun Oct 31 04:00:00 CST 2004
```

提示： clock add 命令可以很好地处理夏令时的开始与结束，儒略历与格利高里历的转换。完整信息见 clock 的参考文档。

15.4 运行计时命令

time 命令可以用于测量 Tcl 脚本的运行时间。它需要两个参数：一个表示脚本，另一个表示可选的重复次数。

```
    time {format "%d%s%f" 123 xyz 4.56} 100000
⇒ 3.85866 microseconds per iteration
```

time 将给定的脚本按照指定的重复次数(默认为 1)重复运行，将总运行时间除以重复次数，返回像上面示例中这样的消息，给出单次运行的平均耗时数，以毫秒为单位。即使时钟不够精确，重复运行统计仍可显著地提高准确度。

提示：一般来说，要准确地测量时间，time 命令重复运行脚本的总时间应为一秒左右。

15.5 info 命令

info 命令提供与 Tcl 解释器内部管理相关的状态信息。该命令有丰富的选项，在后文将加以讨论。

15.5.1 有关变量的信息

info 的一些选项用于提供有关变量的信息。info exists 返回 1 或 0，代表指定名称的变量(包括数组元素)是否存在。

```
    set x 24
    info exists x
⇒ 1
    unset x
    info exists x
⇒ 0
    info exists env(PATH)
⇒ 1
```

选项 vars、globals 以及 locals 返回满足指定要求的变量名的列表。info vars 返回当前调用过程层级中可访问的所有变量名；info globals 返回所有的全局变量名，无论它们是否可被访问；而 info locals 返回局部变量名，包括当前过程的参数(如果有的话)，但不返回全局变量。在这些命令中，还可以再指定一个模式参数。如果指定了模式，则仅返回其名称与模式匹配的变量(遵循 string match 匹配规则)。

例如，假设已经定义了全局变量 global1 和 global2，运行下面的过程：

```
proc test {arg1 arg2} {
    global global1
    set local1 1
    set local2 2
    …
}
```

在过程中执行下面的命令所得到的结果如下：

```
    info vars
⇒   global1 arg1 arg2 local2 local1
    info globals
⇒   global2 global1
    info locals
⇒   arg1 arg2 local2 local1
    info vars *a1*
⇒   global1 local2 local1
```

另外，在 info vars 的模式参数前可加上命名空间，这种情况下，列出该命名空间中定义的变量。

```
    namespace eval example {
        variable var1
        variable var2
        variable anotherVar
    }
    info vars example::var*
⇒   ::example::var2 ::example::var1
```

15.5.2　有关过程的信息

另一组 info 选项提供了有关过程的信息。命令 info procs 返回当前命名空间中所有 Tcl 过程的列表。与 info vars 相似，它也接受可选的模式参数，仅返回与给定模式匹配或指定命名空间中的过程。info body、info args 以及 info default 返回有关过程定义的信息。

```
    proc maybePrint {a b {c 24}} {
        if {$a < $b} {
            puts "c is $c"
        }
    }
    info body maybePrint
⇒
    if {$a < $b} {
        puts "c is $c"
    }

    info args maybePrint
⇒   a b c
    info default maybePrint a x
⇒   0
    info default maybePrint c x
⇒   1
    set x
⇒   24
```

info body 返回在 proc 命令定义过程时所给定的过程块。info args 返回一个列表，其内容是过程的参数名，顺序与 proc 定义时给出的顺序相同。info default 返回参数的默认值的信息。它需要获取三个参数：过程名称、传给过程的参数名称以及一个变量名。如果指定的参数没有默认值(例如上面示例中的 a)，info default 返回 0。如果参数有默认值(例如上面示例中的 c)，info default 返回 1，并将给出的变量设置为该参数的默认值。

作为如何使用上述命令的示例，下面是 Tcl 脚本文件中的一个 Tcl 过程。这个脚本包含了 proc 命令的 Tcl 代码，用来在解释器的全局命名空间重建所有的过程。这个文件可以在其他的某个解释器中应用于 source，以从源解释器中复制过程状态。这个过程需要获取

一个参数，即用于写入信息的文件名。

```
proc printProcs {file} {
    set f [open $file w]
    foreach proc [info procs] {
        set argList {}
        foreach arg [info args $proc] {
            if {[info default $proc $arg default]} {
                lappend argList [list $arg $default]
            } else {
                lappend argList [list $arg]
            }
        }
        puts $f [list proc $proc $argList \
                [info body $proc]]
    }
    close $f
}
```

info 还提供了另一个与过程相关的选项：info level。如果 info level 不加参数地调用，则返回当前过程的层级：0 表示当前没有活动的过程，1 表示当前过程由顶层调用，等等。如果 info level 获得了一个参数，则该参数表示的是过程的层级，info level 返回的是一个列表，其元素为该层级的过程的名称及其参数。例如，下面这个过程输出当前的调用堆栈，展示了每个活动的过程的名称和参数。

```
proc printStack {} {
    set level [info level]
    for {set i 1} {$i < $level} {incr i} {
        puts "Level $i: [info level $i]"
    }
}
```

15.5.3 有关命令的信息

info command 子命令与 info procs 相似，不同之处在于它返回的是有关所有当前可见的命令的信息，而非过程的信息。如果不带参数地调用，它返回的是当前命名空间上下文环境中可用的所有命令名称的列表。如果提供了参数，则参数应该是一个遵循 string match 匹配规定的模式限定，加上可选的命名空间前缀(与 info procs 和 info vars 相同)，仅有(指定命名空间中)满足匹配模式的命令名会被返回。

```
    info commands for *
⇒ for format foreach
    namespace eval example {
        proc foo {} {return foobar}
        proc bar {} {return barfoo}
    }
    info commands example::*
⇒ ::example:foo ::example::bar
```

命令 info cmdcount 返回一个数字，表示在这个 Tcl 解释器中已经执行了多少条命令。它可以用于查看为了实现一定功能，已经执行了多少 Tcl 命令，也可通过解释器的使用资源限制(详见 15.11.4 节)控制该值。

命令 info script 查看一个脚本文件是否正被处理。如果是，该命令返回正在活动的嵌套最内层的脚本文件。如果没有活动的脚本文件，info script 返回空字符串。该命令常常用于定位与脚本文件处于同一目录的文件，例如：

```
set otherFile [file dirname [info script]]/other.tcl
```

提示： 您也可以为 info script 传入一个新的脚本文件名作为参数，从而重设当前脚本文件，在为 source 命令编写用户自定义的替代时就可以这样做。这种方法可以在处理脚本之前对其进行预处理，但极少使用。

在从通道(如 stdin)中进行输入时，使用 info complete 命令检查确定一个字符串是否表示了语法上完整的 Tcl 命令。这可用于编写允许将交互式取得的脚本作为程序过程的一部分运行的处理器；当 tclsh 不带脚本参数地运行时，它的主要读取-处理循环就使用 info complete 确定是否将一个字符串传给 Tcl 解释器进行处理。

15.5.4 Tcl 解释器版本及其他运行环境信息

info tclversion 命令返回 Tcl 解释器的版本号，其格式为主版本号.次版本号，如 8.5。主版本号和次版本号都是十进制字符串。如果新发布的 Tcl 的版本只包含向下兼容的改动，如错误修正和添加新功能，则只有次版本号增大，主版本号维持不变。如果新的版本包含了不向下兼容的改动，现有的 Tcl 脚本或调用 Tcl 库过程的 C 代码需要修改，则主版本号会增大，次版本号设为 0。命令 info patchlevel 返回了更详细的版本信息，给出了 Tcl 解释器确切的打包层级，如 8.5.3；这一信息通常仅用于提供错误报告，以及决定是否进行升级。

命令 info library 返回 Tcl 库目录的完整路径名称。该目录保存着 Tcl 使用的标准脚本，如 15.10 节中讨论的 unknown 过程的默认定义。

命令 info hostname 返回运行 Tcl 解释器的机器的主机名。注意这个命令只返回机器的主名。一些机器(特别是拥有多个网络连接的服务器系统)可能有多个名称，如果需要获得各个名称，则需要更加复杂的方法。

命令 info nameofexecutable 和 info sharedlibextension 分别返回调用 Tcl 解释器的可执行文件名和运行 Tcl 解释器的平台使用的标准共享库文件的扩展名。info nameofexecutable 用于通过 exec 或 open 将另一个 Tcl 解释器作为独立进程调用；而 info sharedlibexetension 用于创建跨平台的 Tcl 脚本，从而在脚本中使用 load 来访问以 C 语言等编程语言编写的扩展模块。

15.6 对简单变量的跟踪操作

trace 命令用于监视 Tcl 解释器很多方面的使用，包括变量。这种监视称为跟踪。如果对一个变量建立了跟踪，则在该变量被读、写、删除时都会调用 Tcl 命令。跟踪可用于多种目的，例如：

- 监视一个变量的使用(例如，每当对变量进行读写操作时都输出一条消息)。

- 将变量中的变化传播到系统的其他部分(例如，确保某个组件总是显示某变量指定的人的图像)。
- 通过拒绝对变量的某种操作(例如，如果试图将变量的值改为十进制数字字符串之外的值，就产生一个错误)或重设对变量的某种操作(例如，如果变量被删除，就重新创建它)，限制对变量的使用。

下面是一个简单的示例，在两个变量中任一个被修改时就发送一条消息。

```
trace add variable color write pvar
proc pvar {name element op} {
    upvar 1 $name x
    puts "Variable $name set to $x"
}
```

这个示例中，trace 命令使得对 color 变量进行写操作时就会调用 pvar 过程。参数 variable 指明要创建的是变量跟踪，color 给出了要跟踪的变量的名称，write 指定了要跟踪的操作(可以是 read、write 以及 unset 任意组合的列表)，最后一个参数是要调用的命令。

当 color 被修改时，Tcl 调用 pvar，并给出额外的三个参数：变量名称；变量的元素名称(如果变量是一个数组元素)，或是空字符串(pvar 过程忽略这个参数)；表示实际调用哪个操作的参数。如果变量被读取，则该参数为 read；如果变量被写入，则该参数为 write；如果变量被删除，则该参数为 unset。例如，如果命令 set color purple 被执行，跟踪设置会触发 Tcl 处理命令 pvar color {} write。

pvar 过程完成两件事。首先，该过程使用 upvar 让变量值在过程中可以通过局部变量 x 访问。然后它在标准输出端输出变量的名称和值。上面一段的命令会引起如下输出：

```
Variable color set to purple
```

写跟踪在变量名已经被改动之后且写命令将新值作为结果返回之前调用。trace 命令可以向变量中写入一个新值，重设最初的写操作指定的值，而这个新值作为被跟踪的写操作的结果返回。读跟踪在变量的结果被读取之前调用。trace 命令可以修改变量的值，从而影响读操作返回的结果。在一个变量执行读或写的 trace 命令时，暂时不能对其进行跟踪。也就是说 trace 命令对变量的访问不会递归地触发跟踪。

如果读或写跟踪返回了任何形式的错误，都会放弃跟踪操作。这可用于只读变量的实现。下面这段脚本强制要求变量的值为正整数，拒绝任何将它设置为非整数值的尝试。

```
trace add variable size write forceInt
proc forceInt {name element op} {
    upvar 1 $name x $(name)_old x_old
    if {![regexp {^\d+s} $x]} {
        set x $x_old
        error "value must be a positive integer"
    }
    set x_old $x
}
```

在调用 trace 命令时，变量已经被修改过了。因此如果 forceInt 要拒绝写操作，它必须重新取得变量原来的值。要完成这一任务，在后缀_old 的影子变量中保存了变量原来的值。如果变量的值被设置为非法的值，forceInt 会将变量的值设回原来的值并返回一个错误。

```
    set size 47
⇒   47
    set size red
Ø   can't set "size": value must be a positive integer
    set size
⇒   47
```

您可以使用跟踪来创建只读变量。最简单的实现方法就是将变量的值保存为跟踪回调本身的一部分；能这么做是因为跟踪回调是一段脚本，而不仅仅是一个命令名。在对跟踪进行注册时提供的额外参数会在回调过程中作为参数传入，例如：

```
set pi [expr {4*atan(1)}]
trace add variable pi write [list constant $pi]
proc constant {value name element op} {
    upvar 1 $name var
    set var $value
    error "variable is a constant"
}
set pi 3
Ø   can't set "pi" :vairable is a constant
```

提示： 在为回调和其他延迟处理产生脚本片段的时间，强烈推荐像示例中这样使用 list 命令。这样会应用适当的括号，保证回调过程获取的参数就是提供的那些参数，而不受 Tcl 解释器干扰。

对不存在的变量设置跟踪也是合法的；在对不存在的变量设置跟踪之后变量仍然是不存在的。例如，可以对一个数组设置读取跟踪，在第一次读取时用它自动创建数组元素。对一个变量取消设置会删除变量和与它有关的跟踪，然后调用变量的取消跟踪。取消跟踪后立即重建对同一变量的跟踪，以便监视这一变量以后是否被再次创建，这种用法是合法的，虽然并不常用。

要删除一个跟踪，可调用 trace remove variable，提供传递给 trace add variable 同样的参数。例如前面对 color 的跟踪，可以用如下命令删除：

```
trace remove variable color write pvar
```

如果 trace remove variable 的参数与存在的跟踪不吻合，则该命令不会有任何效果。

命令 trace info variable 返回指定变量当前的跟踪信息。调用它时需要提供一个变量名参数，如下所示：

```
    trace info variable color
⇒   {write pvar}
```

trace info variable 的返回值是一个列表，其中每一个元素描述了对变量的一个跟踪。各元素本身是有两个元素的列表，列出了跟踪的操作和各跟踪对应的命令。结果列表中出现的跟踪的顺序按照它们被调用的顺序排列。如果 trace info variable 中指定的变量是一个数组元素，则只返回针对该元素的跟踪，而不返回针对整个数组进行的跟踪。

提示： 使用跟踪可能创造出极不清晰的代码，例如在读或写一个变量时做出诸多改动。这是一种非常糟糕的风格。如果一个操作的副作用很多，那么它(通常)应该通过命令调用来实现。

15.7 跟踪数组变量

使用 trace 命令不仅能跟踪普通变量，还能跟踪数组元素和整个数组。跟踪一个数组元素和跟踪简单变量是完全一样的，只不过在为跟踪回调命令传递变量名时，名称分为两个部分(数组名和元素名)，例如：

```
set a(length) 42
trace add variable a(length) write pAry
proc pAry {aName eName op} {
    upvar 1 $aName ary
    puts [format "Notice: %s(%s) has changed to %s" \
            $aName $eName $ary($eName)]
}
incr a(length)
⇒ Notice: a(length) has changed to 43
43
```

跟踪整个数组有一些细微的不同。首先，对数组中任意元素的读、写以及删除都会被跟踪(根据访问类型)，只有使用 upvar 命令为被跟踪的数组中的某个元素创建了链接的例外。其次，有一种新的跟踪类型，用于跟踪对整个数组进行的操作，例如 array get 或 array names 命令。这种跟踪类型称为 array 类型。每当数组被作为整体访问时，就会触发这种类型的跟踪。这种类型的跟踪不会传递单个元素的名称。

```
set a(b) c
trace add variable a {read write array} pWholeAry
proc pWholeAry {aName eName op} {
    puts "Traced: $op on ${aName}($eName)"
}
append a(b) c
⇒ Traced: write on a(b)
⇒ cc
array set a {c d e f}
⇒ Traced: array on a()
⇒ Traced: write on a(c)
⇒ Traced: write on a(e)
array get a {[a-c]}
⇒ Traced: array on a()
⇒ Traced: read on a(b)
⇒ Traced: read on a(c)
⇒ b cc c d
```

提示： 数组跟踪可用于实现全局 env 数组。那些跟踪使得每次数组被读写时都进行重建。这意味着强烈建议您避免用 upvar 把一个局部变量链接到 env 数组的任一元素，应该总是使用 global 来引入整个数组。这种重建行为也是对 env 访问速度慢的原因，如果您确实需要访问环境变量，使用其他机制会更有效率。

15.8 重命名和删除命令

rename 命令用于改变一个应用程序的命令结构。它需要获取两个参数：

```
rename old new
```

rename 的功能正如其名：它是一个重命名命令，将旧的名称 *old* 改变成 *new*。如果 *new* 作为命令名已经存在，那么调用 rename 时就会发生错误。

您还可以在使用 rename 时将 *new* 提供为空字符串，从而删除命令。例如，下面的脚本通过删除解释器中的相关命令移除了事件循环过程。

```
foreach cmd {after fileevent update vwait} {
    rename $cmd {}
}
```

任意 Tcl 命令都可以被重命名或删除，包括内建命令和应用程序定义的过程与命令。在重命名或删除内建命令时请小心，因为它会毁掉基于该命令的脚本(有可能包括其他一些内建命令)，但在某些情况下这样做也很有用。例如，Tcl 定义的 exit 命令会立即退出过程。如果一个应用程序希望在退出前清理它的内部状态，它可以为 exit 创建一个如下的“包装”。

```
rename exit exit.old
proc exit {{status 0}} {
    # application-specific cleanup
    # ...
    exit.old $status
}
```

这个示例中，exit 命令被重命名为 exit.old，定义了新的 exit 过程，这个过程首先执行应用程序需要的清理工作，然后调用被重命名的 exit 退出过程。这使得已经存在的调用 exit 的脚本仍可以不加修改地使用，而这个应用程序在退出前也可以进行它所需要的清理。

15.9 跟踪命令

trace 命令可用于对有关命令的事件进行跟踪，就如同跟踪有关变量的事件一样。通过使用 trace add command，可以跟踪命令名称的改变(使用 rename 选项)，探测命令被删除的时间(使用 delete 选项)。在命令与某些资源相关，删除命令的同时应该释放资源的情况下，这种跟踪功能特别有用。例如，下面这个过程实现了简单的日志系统，当日志命令被删除时也关闭日志文件。

```
proc log {name filename} {
    set f [open $filename a]
    interp alias {} $name {} logCommand $f
    trace add command $name delete [list logDone $f]
    return $name
}
proc logCommand {f args} {
    set now [clock seconds]
    set time [clock format $now -format "%D %T: "]
    puts $f $time[join $args " "]
}
proc logDone {f old new op} {
    puts "Renaming \"$old\" to \"$new\" ($op)"
```

```
    close $f
}
```

这个示例中，通过创建实现日志命令的 logCommand 命令的解释器别名(更多信息参见 15.11.1 节)，log 命令创建了一个新的命令，用给出的文件记录日志。为了确保在该关闭的时间文件确实被关闭了，log 命令添加了一个删除跟踪，确保由命令用于写入信息的日志通道正确销毁。这个命令跟踪的回调，由 logDone 实现，会在命令被删除或重命名时调用，跟踪的事件由 trace add command 确定。调用回调时提供三个额外的参数：重命名的命令是哪个，它将被重命名为什么(如果是删除，则是空字符串)，要执行什么操作。下面演示了这个命令的使用。

```
    log example eg.log
⇒   example
    example This is a message
    example This is another message
    rename example eg
    eg This is a third example
    rename example {}
⇒   Renaming "eg " to "" (delete)
    read [open eg.log]
⇒   05/07/05 14:56:14: This is a message
    05/07/05 14:56:19: This is another message
    05/07/05 14:57:02: This is a third example
```

这个示例还演示了另一个重要的跟踪技术：在回调中添加用户定义的参数。这些添加的参数总是出现在跟踪回调函数添加的参数之前。这就是 logDone 发现要关闭的通道的方法。

使用 trace add execution 还可以跟踪命令的执行行为，在对复杂的脚本进行调试时这一功能相当有用。可以跟踪的执行事件有 4 种。要在命令被调用时接到通知，应该对它进行 enter 跟踪；查明命令的结果是什么，应该对它进行 leave 跟踪。还可以得到有关命令运行的更细节的信息。使用 enterstep 和 leavestep 执行跟踪，可以得到命令执行过程中的所有命令条目和命令行为(在对过程进程跟踪时这显而易见，但它同时也可用于任意 Tcl 命令)。

使用 enter 和 enterstep 跟踪，回调命令增加了两个参数。第一个是命令的替换完成的参数列表，第二个是触发跟踪回调的那个操作。使用 leave 和 leavestep 跟踪，添加了 4 个参数。与 enter 跟踪一样，第一个参数是代入参数的替换完成的列表，最后一个参数是触发回调的操作名。不过，第二个参数是此时 catch 检查是否捕获的返回码(0 表示成功，1 表示错误等)，第三个参数是命令的返回值。

```
proc tracer {args} {puts TRACE:$args}
proc subcmd {s} {string length $s}
proc demo {args} {
    expr {[llength $args]+[subcmd $args]}
}
trace add execution demo {enter leave} tracer
demo a b c
⇒ TRACE:{demo a b c} enter
TRACE:{demo a b c} 0 8 leave
8
demo "another demo"
⇒ TRACE:{demo {another demo}} enter
TRACE:{demo {another demo}} 0 15 leave
```

```
    15
    trace remove execution demo {enter leave} tracer
    trace add execution demo { enterstep leavestep } tracer
    demo a b c
⇒   TRACE:{expr {[llength $args]+[subcmd $args]}} enterstep
    TRACE:{llength {a b c}} enterstep
    TRACE:{llength {a b c}} 0 3 leavestep
    TRACE:{subcmd {a b c}} enterstep
    TRACE:{string length {a b c}} enterstep
    TRACE:{string length {a b c}} 0 5 leavestep
    TRACE:{subcmd {a b c}} 0 5 leavestep
    TRACE:{expr {[llength $args]+[subcmd $args]}} 0 8 leavestep
    8
```

和变量跟踪相似，trace info 用于得到关于命令和执行跟踪的信息，trace remove 可用于删除它们。

```
    trace info execution demo
⇒   {{enterstep leavestep} tracer}
```

15.10 未知命令

Tcl 解释器为处理未知的命令提供了一种特殊机制。如果解释器发现某 Tcl 命令中有不存在的命令名，它会检查是否存在名为 unknown 的命令。如果存在这样的命令，解释器就调用 unknown 命令而非原命令，将不存在的命令的名称和参数都传给 unknown。例如，输入如下命令：

```
set x 24
createDatabase library $x
```

如果没有名为 createDatabase 的命令，就会调用下面这条命令：

```
unknown createDatabase library 24
```

注意在调用 unknown 之前，原命令参数中的替换操作已经进行。传给 unknown 的每一个参数都在原始命令的单词的基础上完成了替换。

unknown 过程可以进行任何操作，而它返回的内容就作为原命令返回的内容。例如，下面这个过程检查未知命令是否是现有命令的可区分的词头，如果是，就调用相应的命令。

```
proc unknown {name args} {
    set cmds [info commands $name*]
    if {[llength $cmds] != 1} {
        error "unknown command \"$name\""
    }
    uplevel 1 $cmds $args
}
```

注意当名称完整的命令被再次调用时，必须用 uplevel 调用，才能使命令在与原命令相同的上下文环境中运行。

Tcl 脚本库包括 unknown 的一个默认版本，该版本依次进行如下工作。

(1) 如果命令是一个库文件中定义的过程，source 该文件以定义过程，然后重新调用命令。这被称为自动加载，详见第 14 章。

(2) 如果与该命令同名的应用程序存在，使用 exec 命令调用程序。这个功能称为自动运行。例如，在 Unix 系统中输入 ls 作为命令，而 unknown 就会调用 exec ls 来列出当前目录的内容。如果命令没有明确的重定向，则自动运行会将标准输入、标准输出和标准错误分别定向到 Tcl 应用程序的相应通道。这和 exec 的一般行为不一致，但它允许 Tcl 应用程序直接调用交互式程序，如 more 和 vi。

(3) 如果命令名含有!!这样的特殊格式，使用历史替换获得新命令后再调用它。例如，如果命令是!!，则再次调用前一条命令。有关历史替换的更多信息参见第 16 章。

(4) 如果命令名为现存命令独一无二的词头，则获得完整的命令名，然后再次调用命令。

提示： 后三个操作的目标都在于方便交互式使用，它们仅在交互式地调用命令时才会发生。在编写脚本时不能依赖于这些操作。例如，在脚本中不能使用自动运行功能，而应该总是显式地使用 exec 命令。

如果不喜欢 unknown 过程的默认行为，可以编写自己的版本，或是修改相应的库，提供其他的功能，Tcl 语言的表现特性可以因对此的修改而显著不同。如果不希望在遇到未知命令时进行某个操作，还可以直接删除 unknown 过程，那样调用未知命令时就会产生错误。

提示： 自动运行系统常用的功能之一是 auto_execok 命令，它获得您可能想要用 exec 运行的命令的外部名，返回 exec 使用的单词列表，描述具体如何运行。其中包括为程序搜索 PATH，如果它是一个外壳的内部命令，而不是应用程序的内部命令，则为调用添加适当的代码，等等。例如，在 Windows 中，start 命令就是一个外壳的内部命令，但它可用于打开很多种类型的文件，就像在 Windows 资源管理器中那种情况。因此可以如下使用它，来打开一个保存好的网页。

```
exec {*}[auto_execok start] example.html
```

15.11 从解释器

Tcl 中从解释器是另外的解释器(称为主解释器)的子解释器。它们用于需要在与 Tcl 主程序极为不同的上下文环境中处理 Tcl 脚本的情况，例如：

- 运行非信任的代码。
- 为一个大程序处理插件，该插件不向应用程序主体展示其细节。
- 为配置应用程序创建更友好的 API。
- 把部分代码隔离运行。
- 限制可用于特定操作的资源。

从解释器的变量与主解释器和其他的从解释器相互独立，只有一个例外，一个需要注意的例外：全局数组 env。这样，保持不同的变量空间互不干扰就更加容易了。

例如，可以为应用程序修改配置文件，而不用担心配置文件中临时的变量会影响到主程序的运行。就像一个象棋系统，棋盘状态和规则由主解释器管理，而每个棋手的代码由独立的从解释器管理，显然棋手之间不会相互干扰。

从解释器使用 interp 命令操作。从解释器可以嵌套，interp 命令的子命令通过给出的解释器路径来确定要操作的解释器，这路径是解释器名的列表，从当前解释器开始。除了通常都共享的 env 全局数组之外，从解释器相互之间是完全独立的。

从解释器是使用 interp create 命令创建的(如果允许线程操作，则创建在当前线程中)，可以为它指定解释器名称，也可以不指定。使用 interp eval 可以在这个解释器中使用脚本，然后可以用 interp delete 删除解释器。

提示： 使用 interp eval 不会开始一个新的独立线程。一个线程中可以有多个解释器，就像一个 OS 进程中可以有多个线程一样。解释器提供脚本级别的隔离、独立的运行和独立的全局上下文环境。通过对多线程的使用可以完成并行的运行，这需要使用 Tcl 的 Thread 扩展包来创建线程。完全的隔离需要使用多进程和进程间通信以及上面提供的技术。

从下面的示例可以看出，从解释器技术使得创建应用程序独有的语言变得十分容易。

```
    set slave [interp create]
⇒   interp 0
    interp eval $slave rename expr   calculate
    interp eval $slave rename value  set
    interp eval $slave rename incr   advance
    interp eval $slave {
        value x 3
        calculate {1 + 2 + $x + [advance x] + 5}
    }
⇒   15
    interp delete $slave
```

当创建从解释器时，也创建了与它同名的命令。这使您可以直接调用很多操作，而且可以通过删除该命令删除从解释器。这意味着前面那个例子可以修改如下：

```
    set slave [interp create]
⇒   interp0
    $slave eval rename expr  calculate
    $slave eval rename value  set
    $slave eval rename incr   advance
    $slave eval {
        value x 3
        calculate {1 + 2 + $x + {advance x} + 5}
    }
⇒   15
    rename $slave {}
```

15.11.1 命令别名

有时候在从解释器中运行的脚本需要能触发它的主解释器中的命令。例如，网络服务器配置脚本可能需要告诉服务器该如何配置，尽管它自己没有权限去直接运行配置代码。这个强大的功能是通过用 interp alias 在从解释器中设置命令别名，由它触发主解释器中的命令来实现的。可以使用 interp aliases 命令获得解释器中定义的别名的列表，通过 interp alias 和一些必要的参数来配置各个别名。

```
interp create configurator
interp alias configurator service  {} initNamedService
interp alias configurator usePort  {} setServicePort
interp alias configurator dump     {} parray config
proc initNamedService {name} {
    global servName
    set servName $name
}
proc setServicePort {port} {
    global servName config
    if {! [string is integer -strict $port]} {
        return -code error \
            "\"$port\" is not an integer"
    }
    set config($servName) $port
}
interp eval configurator {
    service http
    usePort 80
    service "http-alt"
    usePort 8080
    dump
}
⇒ config(http)      = 80
  config(http-alt)  = 8080
  interp eval configurator {
    service {Whale watching!}
    usePort {Key West}
  }
∅ "Key West" is not an integer
```

习惯上，在创建从解释器时自动创建的那个命令，拥有 alias 子命令，不过这个子命令只能由从解释器向主解释器定义别名。(完整的 interp alias 命令有更多的功能，例如，允许由一个从解释器向另一个从解释器定义别名。)也就是说，上面示例中对 service、usePort 和 dump 别名的定义可以如下完成。

```
configurator alias service initNamedService
configurator alias usePort setServicePort
configurator alias dump    parray config
```

在解释器中定义别名时甚至可以使用空字符串作为解释器名，这样就可以按照一个命令定义另一个命令。

```
interp alias {} print      {} puts stdout
interp alias {} printError {} puts stderr
print      "This message goes to standard output"
printError "and this one goes to standard error!"
```

15.11.2 安全从解释器和隐藏命令

从解释器的最重要的用途之一是解析不被主解释器信任的 Tcl 脚本。这方面的例子包括解析嵌入网页、电子邮件或其他环境中的代码，这些代码的创建者通常都是未知的。Tcl 提供了处理这种情况的机制，称为安全解释器。安全解释器是在创建时设置了特殊标志的从解释器，并且隐藏了其中所有非安全的命令(如 exec、open、socket 和 source)，这些命令只在得到了主解释器的明确授权时才能调用。这个机制，与操作系统处理安全进程的方法类似，是一种很强的机制，因为在 Tcl 语言中不执行命令是无法发起改变状态的操作的。尽管安全解释器可以改变它内部的状态，例如创建过程和设置变量，但默认情况下，安全解释器中没有任何可用于与主解释器和运行应用程序的系统交替的命令。而且与一般从解释器不同，安全从解释器不能访问 env 变量，没有预定义的 unknown 命令(unknown 命令参见 15.10 节)。

通过在 interp create 中给出-safe 标志来创建安全解释器。可以像使用其他从解释器一样使用它们，不过命令的默认设置在某些方面受到了更多的限制。

```
    set safe [interp create -safe]
⇒   interp0
    interp eval $safe {
        file delete -force /home
    }
∅   invalid command name "file"
```

安全解释器使用别名时，主解释器的有危险的行为以一种受控制的形式给出。这允许访问一条命令的某部分的行为，而不是授权访问整个命令。

```
    interp alias $safe file {} safeFile
    proc safeFile {subcommand args} {
        switch -- $subcommand {
            dirname - extension - rootname - tail {
                return [file $subcommand {*}$args]
            }
            default {
                return "unknown subcommand \"$subcommand\""
            }
        }
    }
    interp eval $i  {
        file extension example.tcl
    }
⇒   .tcl
    interp eval $i {
        file delete -force /home
    }
∅   unknown subcommand "delete"
```

另一个选择性地允许安全解释器访问被禁止的命令的常见示例是对 source 命令的受控访问。您很可能需要在安全解释器中 source 文件或加载包，但是默认情况下安全解释器中没有 source 命令可用。

在创建安全解释器时，所有非安全的命令都会被移除，但是这些非安全的命令事实上并未删除。解释器只是隐藏了它们。主解释器可以通过 interp invokehidden 命令调用隐藏

命令，正如下面的示例。这里用 source 命令的安全版本检查 source 作用的文件是否的确是 Tcl 库中的 Tcl 脚本文件，如果不是，则拒绝操作，声明指定文件不存在。

```
interp alias $i source {} safeSource $i
proc safeSource {slave filename} {
    # Get rid of elements like .. from the name
    # Very important in real code!
    set fullFilename [file normalize $filename]

    # Get the parts of the filename to examine
    set dir [file dirname $fullFilename]
    set ext [file extension $fullFilename]

    # Perform the saftety check
    if {[string first [info library] $dir] != 0 ||
            $ext ne ".tcl" } {
        return -code error \
            "unknown file \"$filename\""
    }

    # Really source the file
    return [interp invokehidden $slave \
            source $fullFilename]
    }
```

主解释器可以使用 interp hide 和 interp expose 命令控制在从解释器中各命令的隐藏和显示设置，修改能在安全解释器中正常运行的代码的范围。下面这个示例，设定 pwd 命令可见，在安全解释器中的代码可以获得当前目录路径，而 vwait 和 update 命令被隐藏，因此不能进行事件循环。

```
# Expose [pwd] to make a low-priority information leak
interp expose $i pwd
# Hide the ways to run the event loop
interp hide $i update
interp hide $i vwait
```

提示： 虽然安全解释器能够以默认方式创建它们自己的从解释器，但它们创建的所有从解释器都必须是安全解释器，而 interp 命令的很多子命令都不可用。Tcl 的很多扩展在安全解释器中也对自身有所限制。例如，在安全解释器中，Tk 只允许使用它的小部分组件，阻止了所有可能用于影响已经封装的应用程序的访问，还阻止脚本访问系统级设置选项以及剪贴板等。

15.11.3 解释器之间的传输通道

默认情况下，从解释器不能访问它们的父解释器的 I/O 通道。如果解释器希望允许它的某个从解释器能够访问某个通道，则应该使用 interp share 或 interp transfer。interp share 在主解释器创建通道的同时允许从解释器使用该通道(要关闭文件、套接字或管线的通道，两个解释器都必须将其关闭)，interp transfer 将对一个通道的访问由一个解释器传给另一个解释器，同时不在源解释器保持它。两种情况下，在目标解释器和源解释器中，两个通道

的名称都是相同的。

当希望允许子解释器中的脚本打开一个文件，但不希望它知道这个文件在哪里，甚至不需要子解释器知道这个通道对应的是不是文件时，就可以应用这一功能。

```
    set slave [interp create]
⇒   interp0
    interp alias $slave openCurrentLogfile {} openLog $slave
    proc openLog {slave} {
        global loggingHost loggingPort
        set chan [socket $loggingHost $loggingPort]
        interp transfer {} $chan $slave
        return $chan
    }
    interp eval $slave {
        set f [openCurrentLogfile]
        puts $f "script running"
        close $f
    }
```

15.11.4　为解释器设定限制

每个解释器都有它可以进行的递归深度的限制[①]。因此那些意外地无限制地调用自身的过程不会导致解释器崩溃，只会抛出一个可捕获的错误，提供堆栈跟踪信息。

```
    proc recursive {} {
        puts "again and"
        recursive
    }
    recursive
⇒   again and
    again and
    again and
    again and
    again and
    again and
    again and
    again and
    again and
    …
∅   too many nested evaluations (infinite loop?)
```

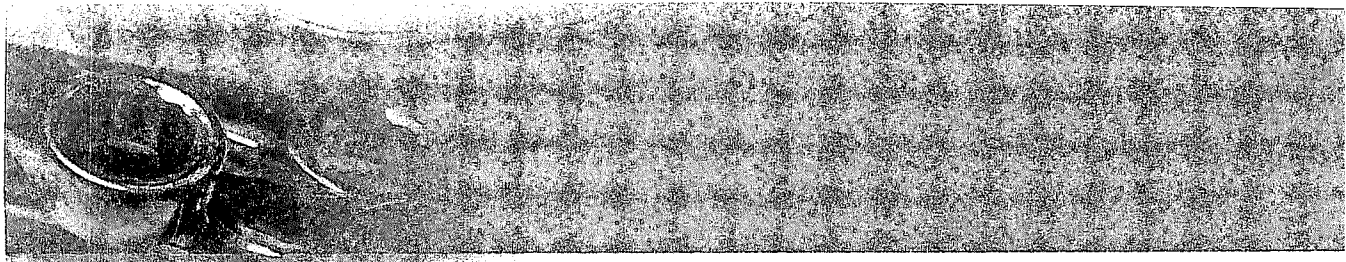
对于大多数应用程序来说，默认的限制已经足够大，但是一些递归层次很多的应用程序可能还需要允许大得多的递归深度，而使用安全解释器的小心翼翼的程序开发者可能希望设置一个小得多的允许深度。这个限制可以用 interp recursionlimit 命令查询和设置。

```
# Read the current recursion limit
set depth [interp recursionlimit $i]
# Compute the new recursion limit (half the old one)
set newDepth [expr {$depth / 2}]
# Set the new limit on the interpreter
interp recursionlimit $i $newDepth
```

通过 interp limit 命令还可以设置很多其他的运行限制，允许主解释器保证某个从解释

① 注意，依赖于进程的堆栈大小，对于递归深度，还有一个硬件上的限制。

器只为某些操作消耗限制范围内的资源。interp limit 命令还提供了一个框架，用于在超出限制时设置闸门和触发器，然后主解释器可以决定是否继续运行。interp 命令的更多细节参见参考文档。还有其他一些内容可以由主解释器相对容易地控制，例如打开的套接字的数量，以及用于写入操作的本地文件的大小。



第16章 历 史

本章讨论 Tcl 的历史机制。在应用程序中交互式地输入命令时，历史机制保持着对前面的命令的跟踪，使您可以方便地调用它们而不必重新输入。您还可以创建与旧命令略有不同的新命令，而不必把整个命令输入一遍，例如修正命令的语法错误时。Tcl 的历史机制提供了很多与 Unix csh 类似的功能，但语法不尽相同。

提示： TkCon 是 Tk 提供的标准控制台的一个替代品。它的众多功能包括复杂的命令历史和编辑功能。您可以使用光标键在历史中的命令间循环，修改取得的命令，然后加以执行。TkCon 还支持跨段地访问历史命令。有关 TkCon 的更多信息以及如何获得 TkCon 参见附录 B。

16.1 本章出现的命令

历史由 history 命令实现。本章只讨论了最常用的一些历史功能，更完整的信息见参考文档。

- `history`
  history info 的快捷方式。
- `history clear`
  擦除所有的历史列表，重置事件数。当前的限制保持不变。
- `history info ?count?`
  返回一个有格式的字符串，给出事件数和历史列表中每个事件的命令。如果给定了 *count*，则只返回最近的 *count* 个事件。
- `history keep count`
  改变历史列表的大小，最多保持 *count* 个事件。列表的初始大小是 20 个事件。
- `history nextid`
  返回历史列表中下一条将要记录的事件的编号。
- `history redo ?event?`
  重新执行 *event* 对应的命令，返回其结果。

16.2 历史列表

交互式输入的每一条命令都会进入历史列表。历史列表中的每一个条目都称为一个事件，它包括命令的文本以及一个用来描述命令的编号。命令的文本就是您输入的字符，在Tcl 解释器进行$、[]等替换之前的字符。编号从 1 开始，1 表示输入的第一条命令，然后依次增加。

假设向交互式的 Tcl 应用程序输入了如下命令：

```
set x 24
set y [expr {$x*2.6}]
incr x
```

这里，历史列表中就有三个事件。可以不带参数地调用 history 来检查历史列表的内容。

```
    history
⇒   1 set x 24
    2 set y [expr {$x*2.6}]
    3 incr x
    4 history
```

history 的返回值是可以阅读的字符串，描述了历史列表的内容，包括 history 命令本身。history 命令的结果设计为输出，而非由 Tcl 脚本处理。如果希望编写处理历史列表的脚本，会发现使用参考文档中讲述的一些 history 选项更加方便，如 history event。

历史列表有固定的大小，其初始值为 20。如果输入更多的命令，则只保留最新的 20 条命令。可以用 history keep 命令改变历史列表的大小。

```
history keep 100
```

这条命令将历史列表的大小改为 100，以后历史列表就会保留最近的 100 条命令。

16.3 描述事件

history 命令的一些选项要求您从历史列表中选中一个事件。事件由字符串描述，格式可能为如下格式之一。

- 正的数字——选中编号为该数字的事件。
- 负的数字——选中相对于当前事件距离为该数字的事件。-1 表示上一条命令，-2 表示倒数第二条命令，以此类推。
- 其他——选中与该字符串匹配的最近的事件。如果字符串与事件对应命令的开头匹配，或字符串与命令按 string match 规则匹配，都认为字符串与事件匹配。

假如已经输入了 16.2 节中的那三个命令。命令 incr x 可以用事件-1 或 3 或 inc 指明，set y [expr {$x*2.6}]可以用事件-2 或 2 或*2*指明。如果没有给出事件描述符，则默认为-1。

16.4 从历史列表中再次执行命令

history redo 命令可以取回命令，再次执行它，就如同重新输入整条命令一样。例如，在输入 16.2 节中的前三条命令之后，命令：

```
history redo
```

会执行最近的一条命令，即 incr x；它将变量 x 的值加 1 并返回其新值(26)。如果为 history redo 提供了参数，则它如 16.3 节所述选择事件。例如：

```
    history redo 1
⇒ 24
```

重新执行了第一条命令，set x 24。

16.5 利用 unknown 实现的快捷方式

基本的 history 命令是很长的，在前面的示例中，使用 history 命令比重新输入要执行的命令按的键还要多。幸好有一些快捷方式，可以只按很少的键就能实现这些功能。

- !!
  重新执行上一条命令；与 history redo 相同。
- !*event*
  重新执行 *event* 命令；与 history redo *event* 相同。
- ^*old*^*new*
  取得上一命令，将其中出现的所有 *old* 替换成 *new*，然后执行得到的命令。

最后一种快捷方法用于修正语法错误时十分方便。

```
    set x "200 illimeters"
⇒ 200 illimeters
    ^ill^mill
⇒ 200 millimeters
```

注意 Tcl 的快捷替换操作会替换掉旧字符串中所有出现的目标。

```
    set x "out and about"
⇒ out and about
    ^out^in
⇒ in and abin
```

所有这些快捷方式都是通过 15.10 节介绍的 unknown 过程实现的。unknown 检测到这些事先有特别说明的命令，然后调用相应的 history 命令来处理。

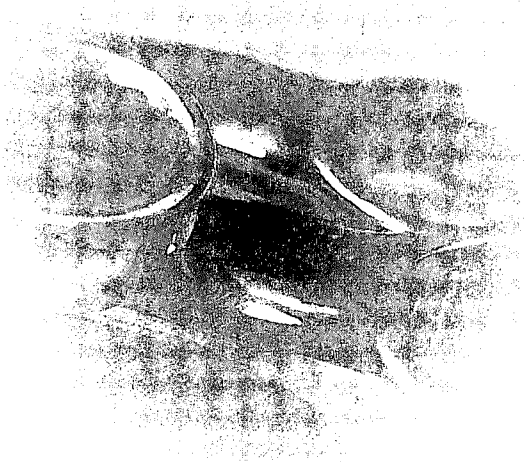
提示：如果系统使用的不是 Tcl 提供的默认版本的 unknown，那么这些快捷方式不可用。

16.6 当前事件号：history nextid

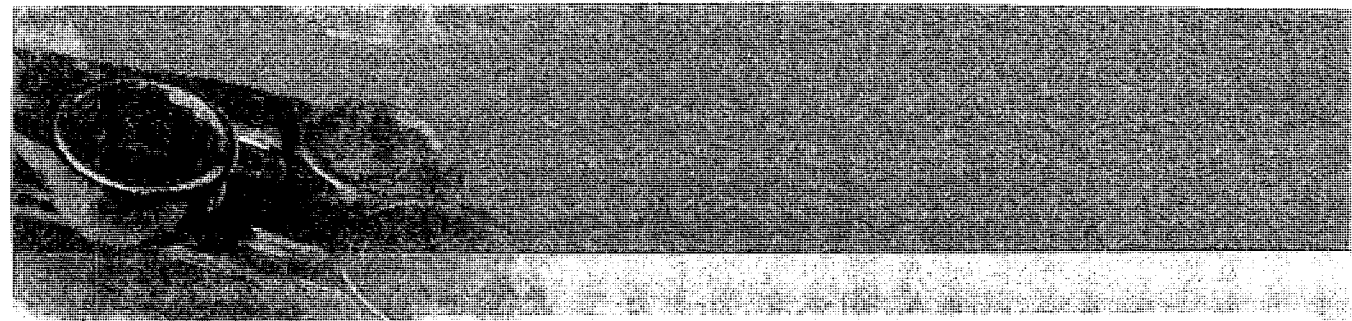
命令 history nextid 返回将加入历史列表的下一事件的编号。

```
    history nextid
⇒ 3
```

交互式的应用程序常用它来产生包含事件号的提示消息。



第II部分　编写Tk脚本



第17章 Tk入门

Tk 是一个通过编写 Tcl 代码来创建图形用户界面的工具集。Tk 扩展了 Tcl 一部分所命名的问题 Tcl 命令，供一些新的命令来创建组件的用户界面元素，并用一个几何管理器将这些元素整合成美观的版面，继而将这个界面与应用程序关联起来。虽然 Tk 的设计初衷是为了扩展 Tcl，但是其他的动态语言，如 Perl、Python 和 Ruby，都采用了 Tk 来使得自己具有创建图形界面的能力。在 Tcl 中关于 Tk 的命令和描述也适用于上述几种语言，但具体的语法有所不同。这一部分介绍的是在 Tcl 当中的 Tk 命令。

像 Tcl 一样，Tk 是作为一个能被 C 语言应用程序使用的 C 库程序包来实现的，并且它能提供一系列能被应用程序调用的函数，以创建用 C 语言编写的新的组件和几何管理器。这些库函数在参考文档中有详细的介绍。

本章介绍了 Tk 用来创建用户界面的基本结构，包括构成用户界面的组件的层级和 Tk 提供的 Tcl 命令的主要分组。第II部分后续的章节将更详细地介绍各个组件的功能。这部分中的所用命令都可以用 wish、第 1 章中介绍的窗口外壳或者包含 package require Tk 命令的所有基于 Tcl 的应用程序执行。

17.1 窗口系统简介

窗口系统提供了通过键盘或鼠标来操纵显示窗口的功能。鼠标用来在屏幕上移动一个指示器，它通常包括一个或两个按键(可能还有一个滚轮)，用来调用动作。每一个屏幕上都显示了一组分层的矩形窗口，最底层的根窗口(通常叫桌面)覆盖了整个屏幕，如图 17.1 所示。根窗口可以有任意多的子窗口，分别称为一个顶层窗口。一个应用程序通常会用到好几个顶层窗口，每个顶层窗口都是这个程序的主面板和对话框。顶层窗口可能还有自己的子窗口，以此类推。一般而言，顶层窗口的下层窗口称为组件或者内部窗口，它们能够被嵌套在任意一个层级。组件用于单独的控件，如按钮、滚动条或者文本项，还可以用来组合其他的组件。

当用户敲击键盘、移动光标或单击鼠标时，说明用户动作的事件被发送到应用程序。每一个事件包含了刚才的行为(例如，鼠标移动到窗口范围内)，还提供了事件发生的窗口、事件发生的时间、光标的位置和鼠标按键的状态等有关事件的信息。事件也可能因为结构的变化(例如窗口大小调整或者关闭)而产生，这样的事件用来通知应用程序窗口必须刷新，例如当一个窗口移开了，不再遮盖住另一个窗口时。

窗口系统不要求窗口有特定的外形或行为。一个程序能在任意窗口中绘制任何内容，还

可以以任意方式来响应事件。因此，应用程序可能以各种各样的外观出现，执行各种各样的交互作用。窗口系统最底层的界面不支持任何特定的外观，也不提供任何内建控件，如按钮及菜单。这些功能由应用程序层次的工具集来完成。Tk 工具集提供了可移植的功能组件库，这些组件能融入任意的外观环境中。在这方面，Tk 的独特之处在于它使用了适当的原生工具集。这使得用 Tcl/Tk 编写的应用程序能在各种平台上运行，适应各种本地环境。

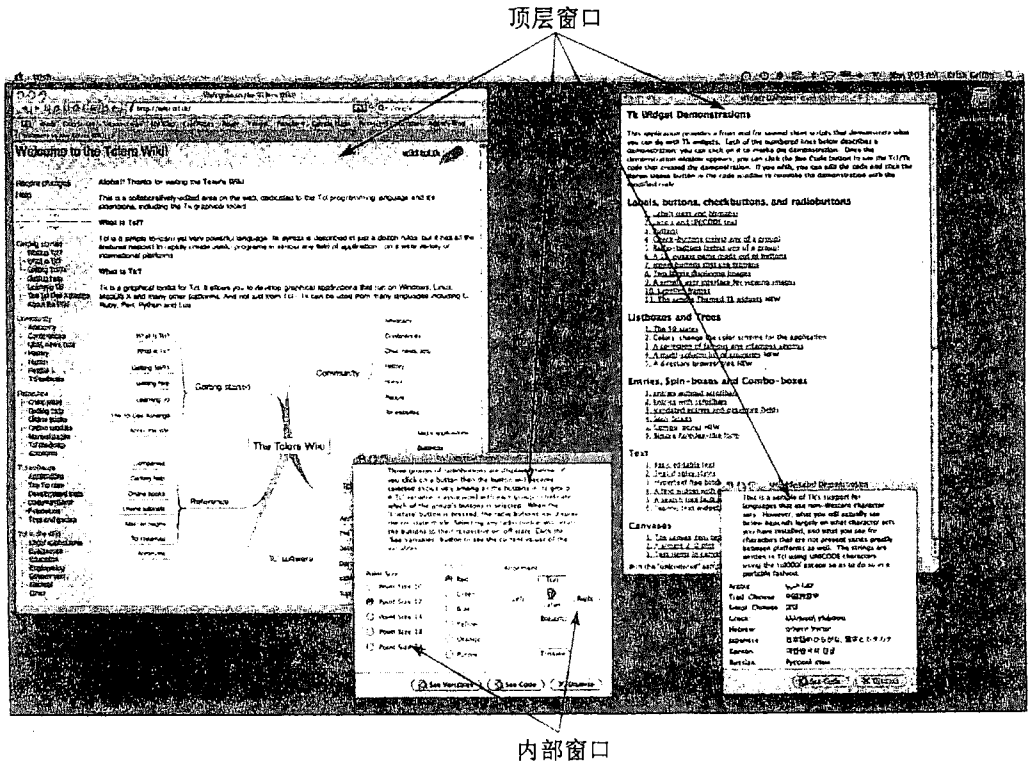


图 17.1　每一屏幕都包含多层级的可重叠窗口的集合

所有的窗口系统的共有元素是一个窗口管理器，也称作桌面环境。这个窗口管理器允许用户在所有应用程序中都用同一种方法来控制顶层窗口。它还为每一个顶层窗口提供了一个装饰框架，框架中的控制动作让用户能交互式地对窗口进行移动、改变大小、图标化、取消图标化等操作。框架还为顶层窗口显示一个标题。这样的窗口管理器有很多：在一些操作系统中，例如 Mas OS X 和微软的 Windows 操作系统，默认的窗口管理器只有一个；而在 Unix 和 Linux 操作系统中，有很多窗口管理器可供选择。本书中的示例使用了各种各样的窗口管理器，以演示 Tk 在所有这些环境中的多功能性和可移植性。

应用程序必须与窗口系统和窗口管理器通信。和前者通信是为了创建窗口、绘制窗口以及接收事件；和后者通信是要指明标题、首选尺寸以及其他关于顶层窗口的信息。

Tk 支持的三种主要窗口系统分别是：X(用于 Unix 和 Linux 操作系统)；微软 Windows 操作系统和苹果的 Mac OS X(也称为 Aqua)。想了解这些操作系统的更多信息，可以搜索以下内容：X.org、Microsoft User Interface Design and Development 或者 Apple Human

Interface Guidelines。

17.2 组件

Tk 使用低层绘图 API 来实现现成的控件集合，这些控件常称为组件。图 17.2 展示了按钮、输入框和滚动条的示例。每一个组件都是一个类的成员，根据所在的类决定其外观和行为。例如，button 类的组件显示一个文本字符串或一个图像。不同的按钮可以用不同的方式显示自己的字符串或图像(例如用不同的字体和颜色来显示)，但每一个按钮都显示一个字符串或一个图像，或两者都显示。每个按钮都和一个 Tcl 脚本关联在一起，当鼠标 1 键(在三键鼠标上是最左边的键)在这个组件上按下然后释放时调用关联的脚本。不同的按钮组件可以与不同的脚本关联，来实现不同的操作。当创建一个组件时，要选择它的类和基于该类的其他选项，例如要显示的字符串或图像，以及要调用的脚本。

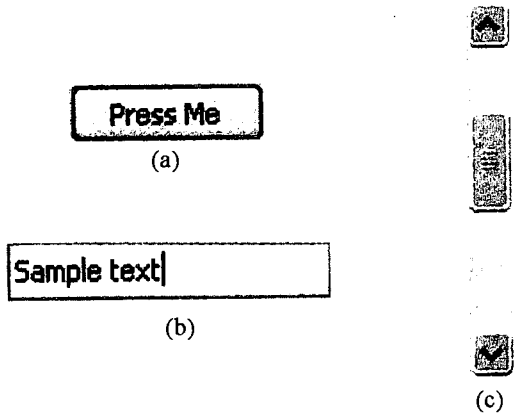


图 17.2 组件的示例：(a)按钮组件；(b)输入框组件；(c)滚动条组件

每一个 Tk 组件由一个窗口实现。[①]和窗口一样，组件也被嵌入层级结构中。一个组件可以包含任意数量的子组件，组件树可以有无限深的分支。组件，例如对用户而言对应于实用行为的按钮，是组件树的叶；而非叶组件通常只是用来组织叶组件的容器。

每一个组件或窗口都有一个文本名字，如“.a.b.c”。组件的名称和文件的路径名称相似，只不过使用的分隔符不是“/”或“\”而是“.”。名为“.”的组件处在层级的最上层，被称为主窗口。名为“.a.b.c”的组件是指名为“.a.b”的组件的子组件 c，而“.a.b”又是“.a”的一个子组件，后者是主窗口的一个子窗口。

图 17.3 显示了一个窗口的组成。(a)所展示的是被窗口系统添加装饰后的窗口；(b)显示了该组件的层级结构；(c)列出了该组件的各个成分。最顶层的组件(名为“.”)包含了三个子组件：一个处于顶部的菜单、右侧的滚动条和一个占据了剩余部分的列表框。菜单上端的菜单栏又有两个子组件：一个 File 菜单(左侧)和一个 Help 菜单(右侧)。每一个组件都有一个名字，反映着它们在层级中的位置，如 Help 菜单的名字.menu.help。

① 在本书和参考文档中，“组件”和“窗口”作为同义词使用。

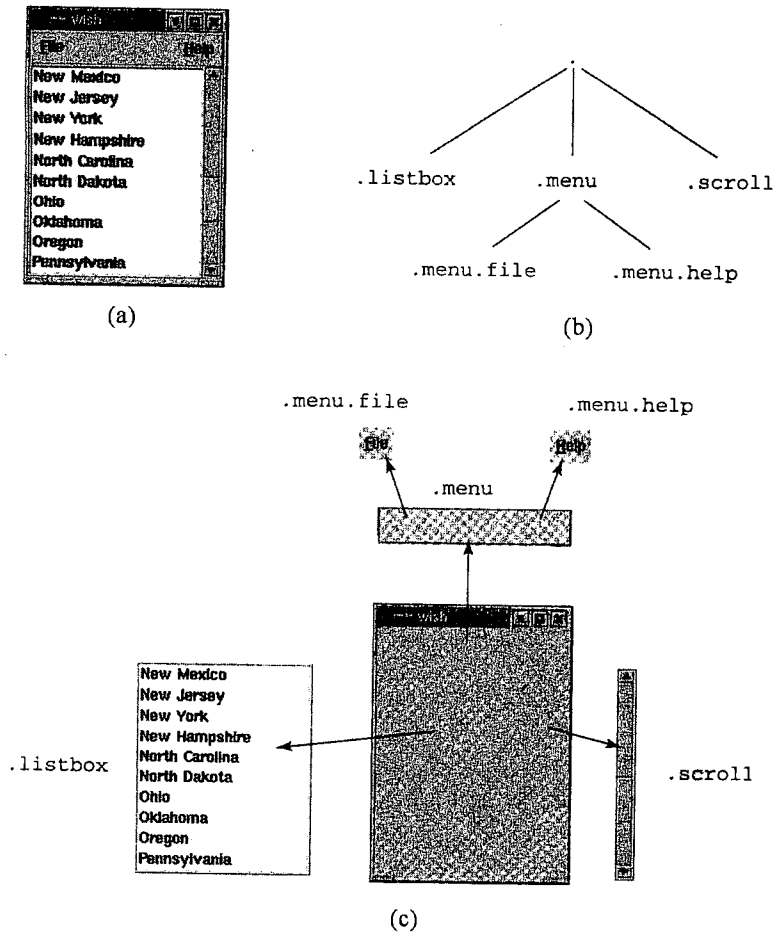


图 17.3　窗口的组成

17.3　应用、顶层组件和屏幕

在 Tk 中，应用程序是指一个组件层级结构(一个主组件和它的子组件)、与这个组件层级结构相关的 Tcl 解释器以及由这个解释器提供的全部命令。每一个应用程序都有自己的组件层级，所以名称.指的是应用程序里的主窗口，或称根窗口。通常一个应用程序在一个进程中运行，但 Tk 也允许一个进程使用多个独立的组件层级结构管理多个应用程序：每个应用程序使用一个 Tcl 解释器。Tk 不请求，也不提供任何多线程的特别支持，但是可以在一个多线程的环境中使用，这时给定窗口层级的所有 Tk 命令和操作都在一个进程和解释器中进行。

每个应用程序的主组件占据一个顶层的窗口，所以它由窗口管理器修饰和管理。应用程序中除主组件外的其他组件大多使用内部窗口。toplevel 组件的类用于创建另外的顶层窗口，从而让窗口管理器可以独立地操纵这些窗口。顶层组件和内部组件的区别在于，前者的窗口是屏幕根窗口的一个子窗口，而后者是 Tk 层级中其父组件的窗口的子窗口。顶

层组件通常被用作面板和对话框的容器，如图 17.4 所示。在这个示例中，对话框.dig 是一个顶层组件，也被称为主组件。图 17.4(a)展示了组件怎样显示在屏幕上，而(b)显示了在应用程序中的 Tk 组件层级。

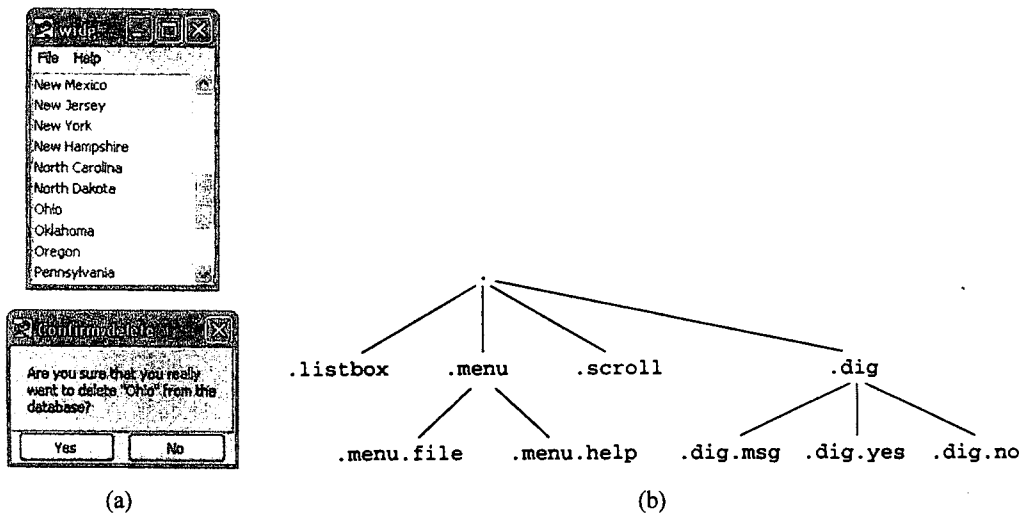


图 17.4 顶层组件

17.4 脚本和事件

一个 Tk 应用程序被两种类型的 Tcl 脚本控制：初始化脚本和事件处理器。当应用程序开始运行时，初始化脚本被执行。它创建应用程序的用户界面，加载应用程序的数据结构并进行应用程序所需的所有其他初始化操作。在初始化完成后，程序进入一个事件循环，等待与用户交互。不论一个事件在什么时候发生，例如用户调用了一个菜单输入框或移动了鼠标，一个 Tcl 脚本都会被调用来处理这个事件。这些脚本被称为事件处理器，它们能调用应用程序特有的 Tcl 命令来向一个数据库中输入信息或者是修改用户界面(例如通过弹出一个对话框的方式)，还可以完成很多其他事情。一些事件处理器是由初始化脚本创建的，但事件处理器也可以由其他事件处理器创建和修改，例如一个关于菜单的事件处理器可以创建一个新的对话框和关于这个对话框的新的事件处理器。

大多数 Tk 应用程序的 Tcl 代码都在事件处理器中。复杂的应用程序可能会包含上百个事件处理器，这些处理器又可能创建其他的面板和对话框，它们各自有着更多的事件管理器。因此 Tk 应用程序是事件驱动的。因为没有单独的任务供应用程序执行，在程序的脚本里也就没有被事先完全定义的控制流。应用程序为用户界面提供了很多功能，由用户决定下一步要做什么。应用程序要做的，是响应用户的指令。这种响应由事件处理器来实现。事件处理器最好比较简洁，并且让大多数处理器之间相互独立。

17.5 创建和删除组件

Tk 提供了 4 个主要的 Tcl 命令组：(1)创建和删除组件；(2)控制组件在屏幕上的布局；(3)和已有的组件通信；(4)应用程序内和应用程序之间的组件互连。本节以及随后的三节将介绍这些命令组，从而使您对 Tk 的功能有一些大致的了解。在后续的章节中，我们将进一步讨论所有这些命令的细节。

要创建一个组件，需要调用一个命令，这个命令根据组件的类命名：例如，按钮组件称为 button，滚动条组件称为 scrollbar，等等。这些命令被称为类命令。例如，下面这个命令就创建了一个显示 Press me 的按钮。

```
button .b -text "Press me" -command {puts Ouch!}
```

所有类命令的形式都与此相似。命令的名称就是组件类的名称。第一个参数是这个新组件的路径名，上面这个示例中是.b。这条命令会创建指定的组件和相应的窗口。组件的名称的后边可以有任意多对参数，每对参数中的第一个指定了这个组件的配置选项，例如 -text 或-command，而每一对参数中的第二个则给定了这个参数的值，例如 Press me 或 {puts Ouch!}。每一个组件类支持一组不同的配置选项，但是很多选项，例如“-foreground”在不同的类中都有相同的用处。并不是被组件支持的每一个选项都必须要指定一个值；Tk 为未赋值的选项提供了各自的默认值。例如，在前面那个示例里，按钮支持大约 35 个不同选项，但只有其中的 2 个被用户赋予了数值。第 18 章将讲述很多最通用的配置选项。

要删除一个组件，可以调用 destroy 命令。

```
destroy .menubar
```

这条命令删除了一个名为.menubar 的组件和它的所有派生窗口。

17.6 几何管理器

组件并不决定自己的尺寸和在屏幕上的位置，这些功能由几何管理器来实现。每一个几何管理器实现一个特定的版面。给出要管理的组件和控制信息，几何管理器就会赋予组件相应的尺寸和位置。例如，可以让几何管理器在一个垂直列上排列组件，管理器就会安排组件的位置，使后者相邻而不重叠。如果一个组件突然需要占有更大的空间(例如当组件换了一个更大的字体)，那么组件会通知几何管理器，使后者能调整其他组件的位置来保持这一垂直列的结构。

17.5 节中提到的 4 组 Tk 主命令组中的第二个就包括了与这些几何管理器通信的命令。Tk 现在包含三个几何管理器，以及三个能管理嵌入式组件的组件。Tk 的主几何管理器是网格管理器，将组件按行和列排列管理。Tk 的主要几何管理器还有打包器和定位器，前者的作用是按顺序将一系列组件放置在空白的边缘，后者采用简单的固定位置或相对位置处理组件。另外，窗口管理器用于和桌面环境一起管理顶层窗口。画布组件、文本组件

和窗格组件都有为嵌入式组件提供的内部几何管理器。第 21 章将进一步介绍几何管理器的细节。

当调用了一个类命令，如 button 时，新的组件并不是立刻就显示在屏幕上。只有当要求几何管理器管理它时它才会显示出来。如果想在全面了解几何管理器前就实验一下组件，可以调用 grid 命令来显示组件(将组件的名称作为它的参数)。出现的父组件大小正好可以容下组件，组件正好占满父组件的全部空间。如果还创建了其他的组件，并以同样的形式调用 grid 命令，那么网格管理器将把两个组件放在同一列，让父组件的大小正好能容纳这两个组件，如图 17.5 所示。下面这段脚本创建了两个按钮组件，并将它们排在了同一列(第一个组件在第二个组件的上方)。

```
button .top -text "Top button"
button .bottom -text "Bottom button"
grid .top
grid .bottom
```

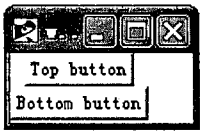


图 17.5 在窗口中安排组件的布局

17.7 组件命令

创建一个新的组件后，Tk 同时也创建了一条新的 Tcl 命令，名称与新组件名相同。这条命令被称为组件命令，所有的组件命令(每一条命令对应应用程序中的一个组件)构成了 Tk 的三个主命令组。在 17.5 节介绍过的 button 命令执行后，一个名为.b 的组件命令就出现在应用程序的解释器中。当组件存在时，这条命令就存在；如果组件被删除了，相应的命令也同样会被删除。

组件命令被用来和已有的组件进行通信。这里是一些在 17.5 节中的 button 命令后能被调用的命令。

```
.b configure -command runCommand
.b invoke
```

第一条命令的作用是，当按钮被按下时就改变了已设定的回调，第二条命令调用了按钮的命令，就好像用户在组件上单击了鼠标 1 键一样。在组件命令里，命令名就是组件的名称，第一个参数指定了对组件调用的动作。一些动作，如 configure，还有其他的参数，这些参数的作用决定于具体的动作。

提示： “动作”、“组件命令”和“组件子命令”在本书中的意义一致，可以互换。例如，“configure 组件命令”和“组件命令的 configure 动作”同义。

给定的组件支持哪些组件命令取决于组件的类。在同一个类中的所有组件都支持同样的那些组件命令，而不同的类支持不同的组件命令集。一些通用的动作在很多类里都得到

了支持。例如，每一个组件类都支持 configure 组件命令，这条命令用于查询和修改组件的配置选项。

17.8 互连命令

Tk 主命令组中的第四组用于互连。这些命令的作用是使组件协同工作，使它们和应用程序中定义的对象协同工作，还能使不同的应用程序以特定方式分享同一个显示器。

一些通信命令是由事件处理器实现的。例如，每一个按钮都有一个-command 选项，用来指定单击鼠标 1 键调用的 Tcl 脚本。滚动条是通过事件处理器进行通信的另一个示例。每一个滚动条都用于控制另外一个组件的外观。当在滚动条中单击，或拖动滚动块时，关联组件的外观就会被改变。这种组件之间的关联采用 Tcl 命令来实现，使得在滚动块被拖动的同时滚动条的相应命令就会被调用。这条命令还调用了一条组件命令，用于改变关联组件的外观。第 22 章将介绍另外几种事件处理器和事件生成器。

一些 Tk 组件通过对 Tcl 变量的链接来支持一种“模式-视图-控制器”范式。这个变量的值用来定义组件的文本内容或者状态。复选按钮组件和单选按钮组件就需要这个变量才能顺利地工作。在第 18 章，我们将对这种特点作进一步解释，有关一些组件提供的-textvariable 和-variable 选项的更多信息见参考文档。

另外一种互连的形式是选择管理。这种选择表现为屏幕上一些被突出显示的信息，例如一些文本或一幅图像。Tk 提供了和窗口管理器一起控制选择的方法，使得应用程序可以声明对选择的所有权，并且可以从任意拥有它的程序中取得其内容。第 25 章将进一步讨论选择管理，并介绍 Tk 的 selection 命令。

第 26 章将讲述 Tk 的窗口管理器命令 wm，这条命令用来和窗口管理器进行通信。窗口管理器作为顶层窗口的几何管理器而存在，而且 wm 命令能为窗口管理器指明特定的几何要求，例如“不允许用户把这个窗口高度变得小于 80 像素”。另外，wm 还能用来指定一个标题，这个标题会显示在窗口的装饰边框上，或者当窗口被图标化后显示，还可指定其他很多属性。

在任何时候，对应用程序进行的键盘输入直接定向到一个特定的组件，而不考虑鼠标指针的位置。这个组件被称为焦点组件或者拥有焦点的组件。第 27 章介绍 focus 命令，它用来查询当前的焦点组件或者是改变焦点。第 27 章还将介绍“攫取”，它用来限制键盘和鼠标事件，使得它们只在组件层级的特定子树中处理。攫取子树之外的窗口不能接收任何事件，直到攫取被释放。攫取也被用来阻止一个程序的部分功能，强迫用户立即处理更高优先级的窗口，例如一个对话框。

最后，第 12 章讲述了程序通信的一些方法。特别是对 Tk 而言，send 命令提供了应用程序间通信的通用方法。可以利用 send 命令来将任意 Tcl 命令发送给显示器上的任意 Tk 应用程序。

第 18 章　Tk 组件概览

本章介绍 Tk 实现的典型的组件类，这些类几乎在刚有 Tcl 的时候就有了。它们提供了一种跨平台的方法，用于为脚本创建用户界面。另外，这些典型的组件还提供了一种便捷的方法，来为已有的应用程序添加用户界面，无论这个程序是不是用 Tcl 来编写的。因为 Tcl 解释器可以嵌入应用程序，可以用一个图形用户界面来扩展应用程序，或者可以使用 Tcl 命令来驱动一个命令行应用程序，然后使用 Tk 组件来引入一个用户界面。

除了经典的 Tk 组件外，还有一种近几年发展起来的主题组件。主题的目的是将组件的功能与组件的外观分离。另外，主题还允许更美观的外观出现。经典的 Tk 外观是从早期的 Windows 95 和 Unix 上的 Motif 发展而来，而主题组件则提供了一个更现代的外观，这个外观也是统一的，同时又是能修改的。本章将介绍更早期的经典 Tk 组件。第 19 章将介绍主题组件。

使用主题组件的主要优势是，通过选择适合用户当前的桌面环境的样式，可以让 Tk 应用程序具备原生的外观。当 Tk 在 Microsoft 操作系统和 Mac OS X 上使用原生组件来显示滚动条、复选按钮和单选按钮时，基于 X 的经典 Tk 组件有基本的 Motif 外观。这种主题组件提供了多种样式来适应 Unix 操作系统上的多种桌面环境。还能够加入新的样式来创建新的主题应用程序，以适应新的桌面环境。更新的主题组件仍然在发展当中，但还不能提供早期组件的全部功能。还有一些经典的 Tk 组件没有对应的主题。但是，在应用程序界面的实现中，可以联合应用经典和主题组件。

提示： 在 Tk 8.5 中，用来创建经典的 Tk 组件的组件类命令是在::tk 命名空间中定义的，例如，::tk::button 创建一个经典的 Tk 按钮组件。而用来创建主题组件的组件类命令是在::ttk 命名空间中定义的。为了和已有的脚本兼容，经典的 Tk 组件类命令被自动导入全局命名空间。未来开发的 Tk 未必如此，那时您可以在已有的脚本中插入一条 namespace import 命令，来明确地导入所需的组件类命令。关于命名空间的详细介绍参见第 10 章。

提示： 本章中的示例使用 grid 命令来安排组件，不过了解这些示例不一定需要理解 grid 命令的操作。第 21 章更详细地讨论了 grid 命令。

18.1　组件基础

每一个经典的 Tk 组件类都由三部分内容确定：配置选项、组件命令和默认的绑定。

配置选项表示了大多数组件的状态信息，例如在按钮组件上要显示的颜色、字体和文本，以及当用户单击这个按钮时要调用的 Tcl 脚本。在同一个类中的所有组件都支持相同的配置选项。不同的类可能有不同的选项，但是一些选项，例如-foreground 和-font，被大多数类支持。组件的配置选项可以在创建该组件时指定，也可以在创建后修改，这时使用 configure 组件命令。默认值可以用选项数据库(参见第 28 章)指定。

定义组件类的第二部分是它的组件命令。正如 17.7 节所述，在一个应用程序中，一个组件命令对应着一个组件，且能被用来调用组件中的动作。组件命令提供的动作在不同的类之间差别很大，但有一些动作能被很多类所支持。例如，每一个类都支持一个 configure 动作，用来改变组件的配置选项。对更复杂的组件，如菜单和列表框，组件的内在状态太复杂以至于很难通过配置选项来表述所有的状态，这时就提供另外的动作来操纵状态。例如，菜单的组件命令提供了一些动作来创建和删除条目，或是修改已有的条目。

定义组件类的第三部分是它的默认绑定。默认绑定响应用户动作来处理 Tcl 脚本，从而设定了组件的行为。Tk 通过 Tk 库目录中的初始化脚本创建默认的绑定，这些脚本调用了将在第 22 章中讲述的 bind 命令。这意味着这些行为并不是被硬编码进组件的，可以使用 bind 命令修改各个组件或者整个类的行为。本章中介绍的是默认行为。

Tk 现在定义了表 18.1 中列出的组件类。除了这些经典的组件外，主题组件扩展还提供了 18 个主题组件(参见第 19 章)。这些主题组件提供了额外的功能和更新的外观感觉，因此假如可以选择，对新开发的应用程序而言，这些组件应该比经典的 Tk 组件更受欢迎。

本章将对大多数的经典组件类和最常用的配置选项进行全面的介绍，画布和文本组件将分别在第 23 章和第 24 章中介绍。本章不会解释每一个组件的每一个功能，因此应该参阅参考文档，来了解每一个组件类的细节。如果想看一些除本章中列出的以外的关于组件应用的示例，可以运行 Tk 发行版中的演示脚本。脚本 widget 包含了很多组件的示例。可以在 wish 解释器中运行如下命令来进行 widget 演示。

```
cd $tk_library
cd demos
source widget
```

这里的 widget 脚本创建了一个包含许多到多种 Tk 组件示例的链接的窗口。对于每一个示例，可以看到一个动态的组件演示以及示例的源代码。

表 18.1 经典 Tk 组件类

button	canvas	checkbutton	entry	frame
label	labelframe	listbox	menu	menubutton
panedwindow	radiobutton	scale	scrollbar	spinbox
text	toplevel			

18.2　框架

框架是最简单的组件，它们是一些彩色的、带三维边框的长方形区域。就像您即将看到的那样，框架通常作为一个整合其他组件的容器来使用。在一个应用程序中，大多数的非叶组件都是框架。框架对于利用几何管理器来构建嵌套结构特别重要，在这种情况下，框架对用户而言常常是不可见的，这一点将在第 21 章中讨论。框架也可以用来生成装饰，例如一条颜色块或者在一组组件周围的一个凸起或凹下的边框。框架并没有任何默认行为，它们一般并不响应鼠标或键盘的动作。

18.2.1　浮雕选项

框架只提供不多的几个配置选项，它们中的大多数都是能被所有的基本组件类支持的。例如，-relief 和-borderwidth 选项能被用作指定一个三维边框。-relief 选项决定了这个边框的外观，而且必须具有以下这些值中的一个：raised、sunken、flat、groove 或者 ridge。Tk 采用浅色和深色的阴影来描绘组件的边框，以便能造成不同的效果。例如，如果一个组件的边框浮雕是 raised，那么相比组件的背景，Tk 将用更浅的颜色刻画上端和左端的边框，用更深的颜色来刻画下端和右端的边框。这使得组件看起来像是从屏幕中凸出一样。图 18.1 给出了一个-relief 设置的示例。

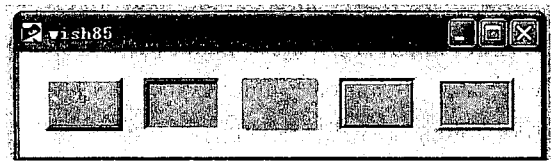


图 18.1　不同浮雕设置的 5 种框架

下面是这个示例的源代码。每一个框架都被涂成更深的颜色以使框架的边缘更突出。

```
set c -1
foreach relief {raised sunken flat groove ridge} {
    frame .$relief -width 15m -height 10m -relief $relief \
    -borderwidth 4 -background {dark grey}
    grid .$relief -column [incr c] -row 0 -padx 2m -pady 2m
}
. configure -background {light grey} -padx 12 -pady 12
```

一个主题组件的三维外观来自该主题，因此通常不需要为 ttk::frame 或其他主题组件设置浮雕。

18.2.2　屏幕距离选项

一些常用的配置选项控制屏幕距离的值。例如，-borderwidth 选项控制了边缘的宽度，-width 和-height 选项可以用来指定边框的维数，而文本组件提供了 insertwidth 选项来

指定插入光标的尺寸。屏幕距离可以用像素或者独立于显示器分辨率的绝对单位指定。距离包括一个整数或浮点数，后面是一个可选的字母，用来给出单位。如果没有指定单位，那么默认是像素。如果想指定单位，可以采用下面的几个字母之一。

- c：厘米
- i：英寸
- m：毫米
- p：磅(1/72 英寸)

举个例子，如果一个距离被指定为“2.2c”，那么显示的距离将由最接近于 2.2 厘米的像素组成。像素的数目在不同的显示器之间会有不同。生成图 18.1 的代码包含了一些屏幕距离选项的示例，它们的单位是毫米和像素。

除了用作组件配置选项外，屏幕距离也可以用于 Tk 的其他目的。一个示例是 grid 命令中的-padx 和-pady 选项，图 18.1 就使用了这一命令和选项的组合，在每一个边框组件的周边留出了额外的空白。

18.3 颜色选项

-background 选项决定了组件的背景颜色，也决定了在边框中使用的阴影颜色。-background 选项的值可以通过符号或者数值的形式指定。颜色的符号值是如 white 或 red 或 SeaGreen2 这样的名称。基于标准的 Unix 颜色值，Tk 定义了大量的能在所有平台上使用的符号颜色。颜色名称不区分大小写：black、Black 和 bLaCk 是一样的。您可以到参考文档中的 colors 页面查到有效的颜色名称列表。Windows 和 Mac OS X 平台也支持一系列的颜色别名，例如 ActiveBorder 和 systemWindowBody，它们是指在这些环境中用户的首选设置。这些符号颜色值在 colors 参考文档中也能找到。

颜色还可以用数字来指定，具体来说，是指定它们的红、绿和蓝三种成分的组成。可以使用以下 4 种形式，三种颜色的成分分别由 4 位、8 位、12 位或 16 位值指定。

```
#RGB
#RRGGBB
#RRRGGGBBB
#RRRRGGGGBBBB
```

上面这些示例中的 R、G 以及 B 分别是表示红色，绿色和蓝色强度的一个十六进制的值。第一个字符必须是“#”，每一种颜色的位数必须相同。如果给单种颜色的值少于 4 位，它们将作为颜色值的高位。例如，“#3a7”和“#3000a0007000”意义相同。全为 1 的值“打开”所有的颜色，而全为 0 表示“关”。因此，#000 是黑色，#f00 是红色，#fff 是白色。

提示： 如果为一个黑白显示器指定了黑色和白色以外的颜色，那么 Tk 将使用黑色或白色来替换该颜色，具体使用哪一种取决于您所要求的颜色的整体亮度。如果正在使用一台彩色显示器并且在使用色图上的所有条目(例如，因为您在显示器上显示一个复杂的图像)，Tk 会把它作为单色图像对待。

几乎每一个基本的组件类都支持-background 选项，同时大多数组件类还支持更多的颜色选项。例如，大多数组件类都能提供-foreground 选项，用于确定组件中的文本颜色及图像。对于主题组件而言，颜色选项是由主题定义的。

同义词

Tk 为最常用的一部分选项提供了简短的表达形式。例如，-bd 是-borderwidth 的同义词，-bg 是 background 的同义词，而-fg 是 foreground 的同义词。

18.4 顶层

顶层组件和框架几乎相同，只有一点不同：顶层组件占据顶层窗口，而框架占据的是内部窗口。顶层组件通常被作为装载程序面板和对话框的最外层容器来使用。一个应用程序的主要组件也是一个顶层组件。在创建一个新的顶层组件时，可以使用-menu 选项来指定一个菜单组件作为窗口的菜单栏使用。菜单栏作为窗口管理器装饰的一部分沿着窗口的顶部来布置。在 Mas OS X 上，菜单栏顺着屏幕的顶部设置。第 26 章将介绍关于顶层组件管理的更多信息。

18.5 标签

标签组件将信息显示给用户。标签组件可以显示文本字符串、位图或图像，或者是文本字符串和位图或图像的组合。标签提供一些额外的配置选项，用来指定要在组件上显示的内容。图 18.2 是一个用以下代码生成的组合标签示例。

```
label .label -text "No new mail" -bitmap \
    @$tk_library/demos/images/flagdown.xbm \
    -compound top
grid .label
```

图 18.2 显示一个位图和一个文本字符串的标签组件

18.5.1 文本选项

大多数显示简单文本字符串的组件，如标签组件，提供两个用来指定文本字符串的选项。如果指定了-text 选项(如创建图 18.2 的脚本中的.label)，它的值就是组件要显示的字符串。如果指定的是-textvariable 项，可以将它的值设为对全局空间中一个变量的引用名称。-textvariable 选项使得组件显示出变量的内容，如果变量的值发生改变，组件也能相应地改变自身的大小并/或刷新显示新的变量值。

用标签组件显示的文本字符串可以包括嵌入式换行符，这种情况下的标签组件将显示多行文本。-justify 选项决定了这些行是如何对齐的，可以将它的值设为 left、center 或 right。

提示： 一个常见的实际应用是用-textvariable 引用命名空间变量来避免太多的变量污染全局命名空间。这就需要使用完全限定的命名空间名(如::Application::country)。

以下是一个使用-textvariable 的简单示例。

```
proc watch {name} {
    toplevel .watch
    label .watch.label -text "Value of \"$name\": "
    label .watch.value -textvariable $name
    grid .watch.label .watch.value -pady 12
}
```

这个过程将创建一个新的顶层组件并用网格使组件里的两个标签分别显示全局变量的名称和值。例如，调用下面这段脚本能创建如图 18.3(a)所示的面板。

```
set country Japan
watch country
```

图 18.3(b)展示了当调用下面这个命令改变了变量的值时，显示是怎样改变的。

```
set country "Great Britain"
```

(a) (b)

图 18.3 -textvariable 选项将组件的值与变量绑到一起

18.5.2 字体选项

-font 选项指定了在组件中显示的文本字体。Tk 使用一个标准的字体描述形式，即“字形 大小 样式……”，其中的“大小”和“样式”都是可选的。下面是一些字体描述的示例。

```
{Times 12 bold italic}
{Helvetica 14 bold}
{Courier 10 normal}
```

除上述示例中的格式外，其他的格式也是支持的，包括标准的 X11 字体描述格式。Tk 的 font 命令可以用来创建和管理应用程序的字体，更多的细节将在第 20 章中讨论。

提示： 对于几乎所有的组件类，一个组件里的所有字母都使用相同的字体和样式。另一方面，文本组件可以显示多行文本，并为子文本字符串设置不同的字体和其他的显示属性。关于文本组件的更多信息参见第 24 章。

18.5.3 图像选项

除了文本，标签组件和很多其他组件可以以位图或图像的形式来显示图像。位图是一

种由两种颜色构成的图，分为前景和背景。位图用-bitmap 选项指定，而它的值有两种形式。如果值的第一个字母是@，那么剩下的值就是包含 X 位图文件格式位图的文件的名称。因此-bitmap @face.xbm 指定了一个包含在文件 face.xbm 中的位图。图 18.2 的脚本以@$tk_library/demos/images/flagdown.xbm 指定了一个位图。它指向了在 Tk 发行包中的演示库中的一个位图(tk_library 的库文件)。

如果值的第一个字符不是@，那么这个值必须是内部定义的位图的名字。Tk 自己定义了一些能在消息框中使用的内部位图，单独的应用程序还可以另外定义新的内部位图。

-bitmap 选项只用 1 和 0 的形式决定了位图的构成。用来显示位图的前景和背景颜色通过组件中的-foreground 和-background 指定。这意味着一个应用程序中的同一个位图可以用不同的颜色在不同的地方显示，且给定的位图的颜色可以通过修改-foreground 和-background 选项来改变。

图像通过 Tk 的 image create 命令创建。图 18.4 显示了一些图像的示例。图像是直接由图像数据创建的，或者从不同的文件格式转变来的对象。Tk 直接支持 XBM、XPM、GIF 和 PPM/PGM 格式。Tk 的扩展，如 Img 和 TkMagick，能支持几乎所有常用的其他图像文件格式。在下面这个示例中，图像通过-image 选项指定。

```
image create photo hiker -file hiker.gif
toplevel .images
label .images.l -text hiker -image hiker -compound top
grid .images.l
```

图像能被所有引用图像的组件共享。任何对图像的改变都会在已显示的图像中表现出来。在第 20 章中，我们将进一步介绍关于图像的详细信息。

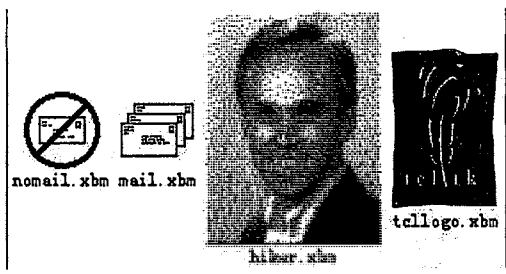


图 18.4　使用 Tk 的 image 命令创建的范例图像

18.5.4　复合选项

标签可以显示图像和文本，如图 18.2 所示。图像相对于文本的位置通过-compound 选项指定。值 top、bottom、left、right 和 center 将会分别把图像放在文本的上端、下端、左端、右端以及中间。如果值被设为 none，那么如果指明-image 或者-bitmap 选项，则只显示图片；否则只显示文本。

18.6　标签框架

前面介绍过，框架被用于将组件组织成组，标签框架组件提供了框架组件的所有功

能，此外，正如它的名字，它还提供了一个标签和一个边框，如图 18.5 所示。标签可以通过-labelanchor 放在框架的 8 个罗盘方位中的任一个。标签框架是对话框和面板中常见的元素。下面的脚本生成了图中所示的标签框架。

```
proc watch {name} {
    toplevel .watch -padx 14 -pady 15
    labelframe .watch.lf -text "Value of \"$name\""
    label .watch.lf.value -textvariable $name
    grid .watch.lf.value -padx 3 -pady 3
    grid .watch.lf -sticky nsew
}
set country "United States of America"
watch country
```

18.7 按钮

按钮、复选按钮、单选按钮和菜单按钮构成了一个有相似属性的组件类。这些类具备标签的所有功能，它们还能够响应鼠标。当鼠标光标指向一个按钮时，按钮会变亮，以显示此时按下鼠标按钮是有用的，一个处在这种状态的按钮被称为活动的按钮。这是 Tk 组件的普遍属性：如果它们准备好响应鼠标按钮了，当有鼠标指向它们时会变亮。鼠标离开这个按钮后，按钮会处于非活动的状态。

如果当一个按钮组件活动时按下了鼠标 1 键，那么组件将显示出下沉的样子，就像是一个真正的按钮被按下去一样。当鼠标按键松开时，组件又恢复成原先的样子。具体来说，当鼠标按钮被松开时，组件将-command 选项作为 Tcl 脚本处理。图 18.6 是用下面这段脚本创建的。

```
button .ok -text ok -command
button .apply -text Apply -command
button .cancel -text Cancel -command cancel
button .help -text Help -bitmap question -command help
grid .ok .apply .cancel .help
```

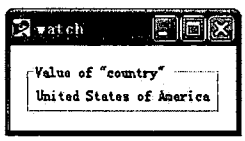


图 18.5 标签框架组件的示例

图 18.6 4 个按钮组件

当用户用鼠标单击相应的按钮时，过程 ok、apply 和 cancel 将被调用，这些过程的代码没在这里列出来。在这个示例中， -command 选项都是一些单词，但事实上它们可以是任意脚本。

18.7.1 复选按钮

复选按钮被用来作出二选一的选择，例如是否启用一条下划线或网格对齐，图 18.7 给出了其中的一个示例。复选按钮和按钮很相似，除了以下两点：第一，在一个复选按钮上单击鼠标 1 键时，全局变量的值会在 0 和 1 之间变化，变量名可以采用-variable 选项赋给

组件；第二，复选按钮在文本或位图的左侧显示了一个方形的选择器，当变量的值为 1 时，它显示为一个选中标记，同时复选按钮被认为是选中了，否则选择器为空。每一个复选按钮都监视着相关的变量的值，假如变量的值变了(例如因为使用了`set 命令)，那么复选按钮会更新选择器的显示。复选按钮还提供了一些配置选项，来指定选择器的颜色，除 0 或 1 外还能指定表示 off 或 on 的值。

图 18.7 中的三个按钮是用下面的脚本创建的。每一个按钮都使得一个特定的变量值在 0 和 1 之间变化，并且它的选择器方框展示了变量当前的值(变量 bold 和 underline 现在是 1，而 italic 是 0)。-anchor 选项使得按钮中的文本和选择器是左对齐的，如 18.15.3 节所述。

```
checkbutton .bold -text Bold -variable bold -anchor w
checkbutton .italic -text Italic -variable italic -anchor w
checkbutton .underline -text Underline \
 -variable underline -anchor w
grid .bold -sticky ew
grid .italic -sticky ew
grid . underline -sticky ew
```

在一些情况下，复选按钮可以用来表示多个项目的状态。这些情况下，让组件显示出一种同时表示开和关的状态会很有用处。在前面那个示例中，文本的 bold、italic 和 underline 状态都可以被显示出来。如果文本里有粗体字和普通字体，按钮将会表示两种状态。这一点通过使用组件的三态值就能实现，如图 18.8 所示。三态值使得组件能显示一个表示这种混合情况的标记。可以用组件的-tristatevalue 选项指定三态值。

```
.bold configure -tristatevalue 3
set bold 3
```

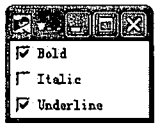


图 18.7 三个复选按钮

图 18.8 处在三态或混合状态的复选按钮的示例

18.7.2 单选按钮

单选按钮组件提供了从相互排斥的选项中选择的方法，就像用汽车收音机上的频道旋钮从很多电台中选出一个来一样。几个单选按钮合在一起能用来控制单个的全局变量。每一个单选按钮都提供了一个-variable 选项来命名这个变量，还提供了一个-value 选项来赋予该变量一个值，当您单击组件时，它能将变量值设成指定的值。例如，图 18.9 显示了一组用来从几个选择中选出一种字体的单选按钮。每一个单选按钮都在文本或位图的左边显示了一个圆形选择器，当某个选择器被选中时(例如当变量值和它的-value 选项匹配时)，选择器会被点亮。每一个单选按钮监视一个变量，并在它的值改变时，改变相应的开关状态。用户可以通过单击按钮来选择全局变量 font 的 4 个值中的一个。在图中，.courier 组件的圆形选择器被标记上了，显示这个组件被选中(此时这个变量的值为 courier)。

```
radiobutton .times -text Times -variable font \
    -value times -anchor w
```

```
radiobutton .helvetica -text Helvetica -variable font \
    -value helvetica -anchor w
radiobutton .courier -text Courier -variable font \
    -value courier -anchor w
radiobutton .symbol -text Symbol -variable font\
    -value symbol -anchor w
grid .times -sticky ew
grid .helvetica -sticky ew
grid .courier -sticky ew
grid .symbol -sticky ew
```

单选按钮也支持三态值及其显示，类似于复选按钮，如图 18.10 所示。

图 18.9　4 个单选按钮

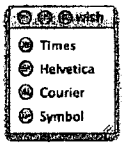
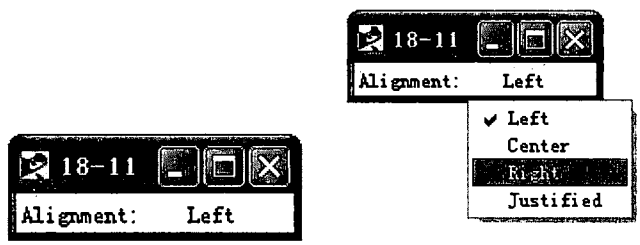


图 18.10　单选按钮组件的三态外观

18.7.3　菜单按钮

菜单按钮组件创建了一个被按下时能拉出一个菜单的按钮。在 18.12 节中将详细地介绍菜单。像其他的按钮组件一样，菜单按钮能被设置成任意的文本和图像的组合，还可以有一个指示器。图 18.11 显示了使用单选菜单项来选择对齐选项的一个菜单按钮。这个菜单按钮由下面的脚本创建。

```
menubutton .mb -width 9 -textvariable justify
set m [menu .mb.menu -tearoff 0]
.mb configure -menu $m
$m add radiobutton -value Left -variable justify \
    -label Left
$m add radiobutton -value Center -variable justify \
    -label Center
$m add radiobutton -value Right -variable justify \
    -label Right
$m add radiobutton -value Justified -variable justify \
    -label Justified
label .l -text Alignment:
grid .l -row 0 -column 0 -sticky e
grid .mb -row 0 -column 1 -sticky w
set justify Left
```



(a)　(b)

图 18.11　(a)普通状态的菜单按钮；(b)下拉状态的菜单按钮

18.8　列表框

列表框是显示一系列字符串的组件，它允许用户选中一个或多个字符串。如图 18.12 中的列表框，显示了可用字体的名称。如果一个列表框有太多项需要同时显示(如图 18.12 所示)，那么列表框会根据窗口大小显示数据。在 18.9 节中介绍的滚动条可以和列表框结合使用，同时，当条目对窗口而言太宽时，列表框还可以水平滚动。

列表框中的条目可用两种方法来控制。一是配置选项-listvariable。这个选项获取包含值的列表的全局或命名空间变量。修改这个变量后，列表框会更新显示列表的内容。listbox 组件命令也提供了一些操作条目的动作，例如，insert 用来添加新的条目，delete 用来删除条目，get 用来获取条目。利用各种 Tcl 列表命令来修改变量值通常比直接使用列表框组件命令更容易。下面的代码实现了图 18.12 中的列表框，用相关联的列表变量来填充列表框。

```
listbox .fonts -listvariable ::fontlist
grid .fonts
set ::fontlist \
    [lsort -dictionary [font families]]
bind .fonts <Double-Button-1> {
    %W configure -font [list [%W get [%W curselection]] 12]
}
```

列表框经常被设置为用户可以用鼠标 1 键单击来选择条目。有些情况下，用户也可以通过单击和拖曳鼠标 1 键来选择一系列的条目。单选模式和多选模式都是支持的(要了解更多的细节可参见参考文档)。被选中的条目显示为不同的颜色，或者是其他的样式或效果。当想要的条目被选中后，用户通常可以使用它们调用另一个组件，如一个按钮或一个菜单。例如，用户可以从图 18.12 中的列表框中选择一个字体名称，然后单击按钮组件，将该字体用在一些对象上，和按钮关联在一起的 Tcl 脚本可以读取选定的条目。

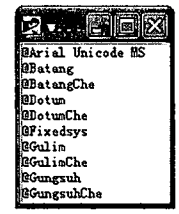


图 18.12　显示了系统中所有可用字体的列表框

在用户双击一个列表框条目时调用一个针对该条目的操作也是很常见的。例如，创建图 18.12 的脚本创建了一个用于双击的绑定，它可以用 get 组件命令从已有的字体表中选择一个来设置列表框的字体。这段脚本用以下方式填充列表框：扫描字体名称列表，按首字母排序，然后用存储在列表框相关的列表变量中的列表填充列表框。用户可以通过单击一个条目，如 Courier，来选中它。双击会使列表框的字体变成指定的字体。

18.9　滚动条

滚动条控制其他组件的外观。每一个滚动条组件都和其他一些组件相联系，例如，一个列表框或一个输入框。如图 18.13 显示了在列表框旁边的一个滚动条，这个列表框显示

出一个目录下所有文件的名字。滚动条通常在两端显示箭头，在中间的槽里显示一个长方形滚动块。滚动块的大小和位置决定了列表框的哪一部分现在在窗口中可见。图 18.13 中，滚动块高度约为整个滚动条高度的 20%，这表示当前显示了列表框中约20%的条目。用户可以在箭头上单击鼠标 1 键来调整可见的区域，这会向箭头指向的方向缓慢移动，或者通过单击槽里不是滚动块的空白部分，来使滚动块朝着鼠标的方向一屏一屏地移动。另外，可见区域还可以通过单击和拖动滚动块来改变。列表框显示了一个文件夹中的文件名，而滚动条用来改变列表框的外观。

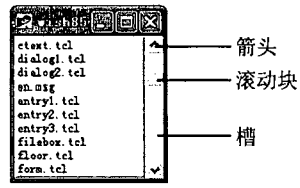


图 18.13　一个带有滚动条的列表框

18.9.1　移动单个的组件

用户调用滚动操作时，滚动条利用自己的-command 选项来通知列表框，同时列表框在调整自己外观时利用自己的-yscrollcommand 选项通知滚动条。例如，图 18.13 显示了一个和滚动条联合的列表框，它的脚本如下：

```
listbox .files -relief raised -borderwidth 2 \
    -yscrollcommand ".scroll set" \
    -listvariable files
scrollbar .scroll -command ".files yview"
grid .files -row 0 -column 0 -sticky nsew
grid .scroll -row 0 -column 1 -sticky ns
grid rowconfigure . 0 -weight 1
grid columnconfigure . 0 -weight 1
set files [lsort -dictionary \
    [glob -directory $tk_library/demos -tails *]]
```

一个滚动条使用 Tcl 脚本和与自己关联的组件通信。例如，考虑图 18.13 的脚本。它采用值为.scroll set 的-yscrollcommand 选项配置列表框。当列表框的外观发生改变时，列表框会向-yscrollcommand 添加两个数字，生成如下的命令：

```
.scroll set 0.80 1.0
```

上例中的数字给定了滚动块所占的百分比。第一个数表示在窗口中可见的文档的第一条信息，第二个数表示在最后一个可见的部分后面的信息。前一条命令和图 18.13 中的视图相对应。接着列表框将处理这条命令。此时，这条命令恰好是滚动条的组件命令，它调用 set 动作，这个动作需要列表框提供的这两个参数。set 组件命令使得滚动条能够按照新参数重绘滚动块。

当用户单击滚动条来改变列表框的外观时，滚动条使用-command 选项来告知列表框。例如，假定用户单击了图 18.13 中上方的箭头。滚动条可以通过将 scroll -1 unit 赋给-command 选项，生成一条 Tcl 命令。

```
.files yview scroll -1 unit
```

然后滚动条会处理这条命令。列表框的组件命令就有一个这样形式的，这个动作使列表框调整自己的外观，使得当前指定的顶端条目就显示在第一行。在调整外观以后，列表框调

用-yscrollcommand 选项来告知滚动条这个新的外观，然后滚动条便可以刷新自己的滚动块了。每一个组件自己确定移动的单位是什么。在这种情况下，列表框以条目作为单位。当滚动块移动时，会生成不同的命令。命令告知组件，使其移动到一个新的百分比位置。例如：

```
.files yview moveto 0.73
```

这里，组件被滑动，使得显示器顶部的信息是从文档的开头开始往下数的73%处的信息。

同样的方法也能被所有支持滚动的组件使用，且它还可以支持不同的滚动条实现。想让一个组件能够滚动，只需为它的组件命令提供一个 yview 动作和一个-ysrcollcommand 选项。滚动条并不和任何特定的组件或组件类有硬性连接。相反，信息只是通过设定滚动条的-command 选项给出的。同样，可以创建新的滚动条组件，并使它们和任何一个已有的组件协同工作，只要为它们的组件命令提供一个-command 选项和一个 set 动作。

Tk 支持水平滚动和垂直滚动。默认的是垂直滚动，但可以通过指定选项-orient horizontal 的方式来设置为水平滚动。列表框必须为它们的组件命令设置一个单独的-xscrollcommand 选项和一个 xview 动作，来支持水平滚动。

18.9.2 多个组件的同步滚动

并不需要将滚动条和组件直接联系起来。过程可以实现用单一滚动条滚动多个组件。得到准确的同步滚动的技巧是给定一个主组件，使其能够控制滚动条和其他的被动组件。当每一个组件的浮点数运算产生了一些小小的舍入误差时，准确性变得十分必要。当这些运算被强制要求由一个组件处理时，其他的组件将和这个组件保持同步。另外，很多 Tk 扩展都为多列滚动组件提供支持以便解决相同的问题。

下面这段脚本创建了用户看起来是两列滚动列表的组件，如图 18.14 所示。

```
# called by the listbox
proc Yset {widgets master sb args} {
    if {$master eq "master"} {
        #Only the master sets the scrollbar
        $sb set {expand}$args
        set wl [lrange $widgets 1 end]
    } else {
        set wl [lrange $widgets 0 0]
    }
        Yview $wl moveto [lindex $args 0]
    }
    # called by the scrollbar
    proc yview {widgets args} {
        foreach w $widgets {
            $w yview {expand}$args
        }
    }
    set widgets [list .lb1 .lb2]
    .lb1 configure \
        -yscrollcommand [list yset $widgets master .sb]
    .lb2 configure \
        -yscrollcommand [list yset $widgets slave .sb]
    .sb configure -command [list yview $widgets]
```

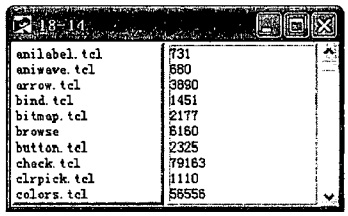


图 18.14　同时滚动多个组件

18.10　标尺

标尺组件有和滚动条相似的外观，但它并不是用于滚动另一个组件，它是用来通过沿着槽移动滑块来选择一个数值的。可以用-from 和-to 选项来指定标尺的最小值和最大值。移动这个滑块会改变用-resolution 给定的增量的值，这个值默认为 1。-orient 选项决定了标尺是 vertical 还是 horizontal，您还可以控制标尺的大小：用-length 选项控制槽的长度，用-width 选项控制槽的宽度，这两者都可以设置为任意的屏幕距离。默认情况下，标尺显示它现在的值，但可以通过将-showvalue 设为 false 来禁用显示。可以将一个字符串的值赋给-label 选项，作为可选的文本标签。您也可以通过-tickinterval 选项设定数字间隔的方式设置数字标记。

您还可以利用-variable 选项来把标尺和一个全局或完全限定的命名空间变量关联起来，使得标尺的值和变量的值能自动同步。您还可以使用 get 子命令来取得标尺的值，或者用 set 子命令来设置它。标尺也支持-command 选项，允许您注册一条命令，在标尺值变化时调用它。在调用命令之前，标尺的更新值会添加到命令的参数当中。

下面这段脚本创建了三个水平标尺组件，每一个组件都允许从 0 到 5 的整数。每一个标尺都有一个描述性的文本标签和数字记号，但标尺值并没有被显示出来。每当标尺的值改变时，它都会调用 updateScore 过程来更新标尺的平均值，这个均值会通过一个标签组件显示。图 18.15 展示了运行下面这段脚本的结果。

```
set score "Average score : 0.0"
set food 0
set ambiance 0
set service 0

# Calculate an average of the ratings to display
proc updateScore {val} {
    global food ambiance service score
    set average [expr {($food + $ambiance + $service) / 3.0}]
    set score [format {Average score: %3.1f} $average]
}

scale .food -label "Food" -variable food \
    -length 5c -width .25c -from 0 -to 5 \
    -resolution 1 -tickinterval 1 -showvalue 0 \
    -orient horizontal -command {updateScore}
scale .ambiance -label "Ambiance" -variable ambiance \
    -length 5c -width .25c -from 0 -to 5 \
    -resolution 1 -tickinterval 1 -showvalue 0 \
```

```
    -orient horizontal -command {updateScore}
scale .service -label "Service" -variable service \
    -length 5c -width .25c -from 0 -to 5 \
    -resolution 1 -tickinterval 1 -showvalue 0 \
    -orient horizontal -command {updateScore}
label .score -textvariable score

grid .food -padx 2 -pady 2 -sticky w
grid .ambiance -padx 2 -pady 2 -sticky w
grid .service -padx 2 -pady 2 -sticky w
grid .score -padx 2 -pady 2 -sticky w
```

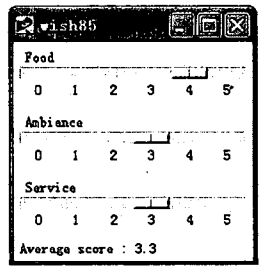


图 18.15　标尺组件的示例

18.11　输入框

输入框是允许用户输入及编辑单行文本字符串的组件。Tk 提供了两种不同类型的输入框组件：输入框和调节框。调节框就是将输入框和上/下按钮组合起来。

18.11.1　输入框组件

图 18.16 显示了一个可能被用于输入一个文件名的输入框组件。要想向输入框组件中输入文本，用户可以在输入框中单击鼠标 1 键。这样就会出现一个闪烁的垂直条，称为插入光标。然后用户就可以输入字符，它们会插入到插入光标所在的位置。这个插入光标可以通过单击输入框中任意位置来移动。按住并拖动鼠标 1 键可以选择输入框中的文本，文本可以通过各种键盘动作来编辑，例如使用 Delete 和 Backspace 来删除插入光标前的字符，更多细节可参见参考文档。输入框的组件命令提供了各种动作，例如 insert 用来插入文本，delete 用来删除文本，icursor 用来定位插入光标，而 index 用来查找在窗口中特定位置显示的字符。

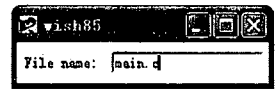


图 18.16　带有识别标签的输入框组件

下面这段脚本创建了图 18.16 所示的组件。

```
label .label -text "File name:"
entry .entry -width 20 -relief sunken -bd 2 \
    -textvariable name
grib .label .entry -padx 1m -pady 2m
```

这里的代码为输入框指定了一个-width 选项，以字符为单位指明窗口应该有多宽。如果文本太长，空间不够，那么只能显示一部分文本，可以为输入框关联滚动条组件、方向

键、Home 键以及 End 键来进行调整。生成图 18.16 的脚本也指定了-textvariable 选项，这个选项把输入框中的文本和全局变量 name 关联起来。一旦输入框中的文本被修改，name 就会更新，反之亦然。

18.11.2 调节框

调节框除具有输入框组件的所有特征外，还拥有一对上/下按钮。单击其中的一个按钮可以按照升序或者降序移动或者调节一系列的值，如时间、日期或者整数。这些值可以像在输入框组件中那样直接被编辑，除非状态设定为 readonly。要在一个月份的列表中找出指定的月份，例如，当使用-value 选项时，如图 18.17 所示。在这个示例中，输入框是不可编辑的。

```
set months {January February
        March April May June July
        August September October
        November December}
    label .label -text "Month:"
    spinbox .spin -width 2 \
        -relief sunken -bd 2 \
        -textvariable month \
        -state readonly \
        -values $months
    grid .label .spin -padx 1m \
        -pady 2m
```

调节框也可以使用-from、-to 和-increment 选项来从一系列数值中找出想要的值。图 18.18 显示了数值范围为整数 6 到整数 72 的一个调节框，相邻两个整数的差值为 2。因为-state 被设为 normal，字体大小便可以通过直接输入一个数字来设定。

```
label .label -text "Font size:"
spinbox .spin -width 2 \
    -relief sunken -bd 2 \
    -textvariable size \
    -from 6 -to 72 \
    -increment 2 \
    -state normal
grid .label .spin -padx 1m\
    -pady 2m
```

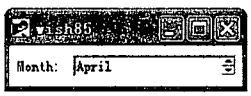


图 18.17 允许从一列值中选出指定值的调节框

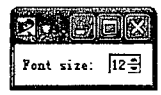


图 18.18 从一列整数中选出指定整数的调节框

18.11.3 show 选项

输入框组件上的-show 选项会隐藏键入的字符，替之以-show 字符。这可被用来接受秘密的信息，如密码，这样，这些信息就不会出现在屏幕上。图 18.19 演示了由下列代码创建的含有-show 选项的输入框。

图 18.19 用-show 选项来隐藏内容的输入框组件示例

```
label .label -text Password:
entry .passwd -show #
grid .label .passwd -sticky ew
```

18.11.4　验证

输入框组件可以使用脚本来进行输入验证。使用-validate 选项，可以进行每一次按键和焦点改变的验证，或者两者都进行验证。验证是通过一个含有-validationcommand 选项的脚本来完成的。该脚本必须返回一个布尔运算值，来表示这种改变是有效的、可接受的，还是无效的、不可接受的。您还可以指定一个-invalidcommand 选项，它使得当验证脚本返回 0 时，脚本被执行；一种典型的应用是调用“bell”，它在该显示框架中响铃。

提示：验证脚本的处理中的一个未被捕获的错误会自动设置-validate 为 none，从而使进一步的验证失效。

图 18.20 中的示例验证了输入框，允许输入合法的十进制整数或者浮点数。注意这个示例代码中的-validatecommand 选项，它有一个以%开头的参数。这样的参数在由输入框组件进行验证之前会被合适的值替代。在输入框组件中，“%W”会被组件名代替，“%P”会被输入值代替(如果输入值可以编辑)，“%S”被正插入或者删除的文本字符串代替。这是一个将关于组件状态的信息传给命令的简便方法。Tk 中有很多地方都可以执行百分号替换。您可以在输入框参考文档中找到验证所支持的一个百分号替换的完整列表。

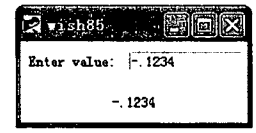


图 18.20　输入验证

下面这段示例脚本通过使用 Tcl 中的 string is 命令来验证输入框，它会检查某个值是否能表示为一个双精度浮点数。但因为每一次按键都需要调用这个脚本，它必须也同时接受一些过渡字符串，将它们视为有效，例如用户输入了一个可选的符号字符但还未输入任何数值。为实现这一点，脚本在 string is 命令返回 0 时会调用 regexp。注意，当它的值是一个空字符串时，string is 将返回 1，因此这也允许用户从输入框中删除所有字符。这段脚本也“重复”了输入框下方的标签中输入的值。

```
label .label -text "Enter value:"
entry .value -width 15 -validate all \
    -validatecommand { CheckValue %P }
label .echo -textvariable echo
grid .label .value  -padx 1m -pady 2m
grid .echo  -       -padx 1m -pady 2m

proc CheckValue { newValue } {
    global echo
    if { [string is double $newValue]
        || [regexp -- {^[+-]?\.?$} $newValue]} {
        set echo $newValue
        return 1
    } else {
        return 0
    }
}
```

提示： 因为输入框中的值在大部分时候是无效的，验证一个输入框会非常复杂。无效的过渡数字必须被允许，否则绝不可能输入一个有效值。因此，按键的验证很少有用。这个示例说明了输入框执行验证的能力，还介绍了这种验证的复杂性。

18.12 菜单

Tk 的菜单组件是一个构造模块，能用来实现多样的菜单，下拉菜单、级联菜单和弹出式菜单。菜单是一个顶层组件，包含了排成一列的一些条目，如图 18.21 所示。菜单条目并不是单独的组件，但它们可以像按钮、复选按钮和单选按钮那样使用。下面就是可以在菜单中使用的几种条目。

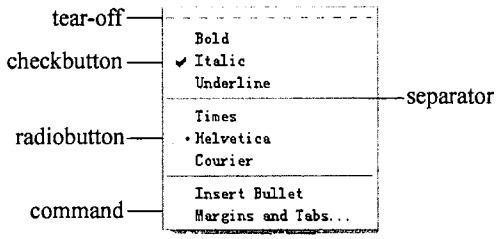


图 18.21 菜单

- command
  和按钮组件类似。用来显示一个字符串或图像，当置于其上的鼠标 1 键释放时会调用一个 Tcl 脚本。
- checkbutton
  和复选按钮组件类似。用来显示一个字符串或图像，当置于其上的鼠标 1 键释放时会调整一个变量为 0 或 1。也可以像复选按钮组件那样显示一个复选记号选择器。
- radiobutton
  和单选按钮组件类似。用来显示一个字符串或图像，当置于其上的鼠标 1 键释放时会将一个特定的关联数值赋给一个变量。也可以采用与单选按钮相同的方法来显示一个单选选择器。
- cascade
  和菜单按钮组件类似。当鼠标指向它时，会激活一个级联的子菜单。更多的细节可参见后文的介绍。
- separator
  显示一条用于装饰的水平线。不响应鼠标动作。

可用菜单的 add 组件命令来创建菜单条目，它们具有和按钮、复选按钮和单选按钮相似的配置选项。

菜单在屏幕上出现的时间很短。大多数时候它们处在不可见的状态，被称为“未激活”。而要调用一个菜单，则首先需要激活它。这种激活可以通过组件命令中的 post 动作来实现。例如，图 18.21 中的菜单可用.m post 100 100 命令来激活。当菜单被激活时，用户可以将光标移到任一菜单条目上并释放鼠标 1 键，以便调用该条目。在菜单被调用以后，它通常会重新处于未激活状态，直到下一次需要激活为止。

下面这段脚本可以创建一个菜单。菜单组件命令的 add 动作被调用来创建菜单中的独立条目。

```
option add *Menu.tearoff 0
   menu .m
```

```
.m add checkbutton -label Bold -variable bold
.m add checkbutton -label Italic -variable italic
.m add checkbutton -label Underline -variable underline
.m add separator
.m add radiobutton -label Times -variable font \
    -value Times
.m add radiobutton -label Helvetica -variable font \
    -value Helvetica
.m add radiobutton -label Courier -variable font \
    -value Courier
.m add separator
.m add command -label "Insert Bullet" \
    -command "insertBullet"
.m add command -label "Margins and Tabs..." \
    -command "mkMarginPanel"
```

菜单可以通过很多方法来激活或者不激活，以达到不同的效果。后文将介绍下拉菜单和级联菜单，它们的激活和不激活都是自动处理的。其他的方法，例如对弹出式菜单和选项菜单的处理，也是可能的。

提示： Tk 菜单支持“分离”(tear-off)的功能。在启用该功能时，一个特殊的分隔符会被作为第一个选项；如果被选中，菜单将显示为一个独立的顶层窗口。可分离的菜单是 Motif 界面的一个典型功能，因此在 Tk 菜单中它们默认处于启用状态。虽然可分离的菜单在现代的界面中很少见，但您通常都希望将可分离菜单默认设为不启用。这一点可由在创建菜单组件前，在脚本中包含命令“option add *Menu.tearOff 0”来实现。而这一点又是由设定一个能应用在选项数据库中的所有菜单的值来实现的，详见第 28 章。

18.12.1　下拉菜单

菜单最常见的形式是下拉菜单。这时，应用程序会在主窗口的顶端附近显示一个菜单栏，如图 18.22 所示。菜单栏会出现在窗口的顶端，并包含一些类似按钮的条目，每一个条目都和一个菜单相关联。(在 Mac OS X 中，菜单栏出现在屏幕的顶端。)当用户在菜单条目上单击鼠标 1 键时，相关的菜单在条目的下方被激活(例如，图 18.22 中的 Text 菜单)。接着用户可以朝菜单下方滑动光标，这时鼠标键是一直被按住的，直到光标移到想要激活的条目为止。当鼠标键被松开后，菜单条目被调用，接着菜单又处于未激活的状态。用户可以在菜单以外松开鼠标键，这样就不会激活菜单，也不会调用任何条目。

下面这个示例能用来创建一个菜单栏。这个菜单栏框架(.mbar)有 6 个子菜单，每一个都是和主菜单有关联的级联菜单。这些菜单的定义并不完全：只显示出了一个菜单定义的一部分。

```
option add *Menu.tearOff 0
menu .mbar
. configure -menu .mbar

.mbar add cascade -label File -menu .mbar.file -underline 0
.mbar add cascade -label Edit -menu .mbar.edit -underline 0
.mbar add cascade -label View -menu .mbar.view -underline 0
.mbar add cascade -label Graphics -menu .mbar.graphics \
    -underline 0
```

```
.mbar add cascade -label Text -menu .mbar.text -underline 0
.mbar add cascade -label Help -menu .mbar.help -underline 0

menu .mbar.text

.mbar.text add checkbutton -label Bold -variable bold \
    -underline 0
.mbar.text add checkbutton -label Italic -variable italic \
    -underline 0
...
```

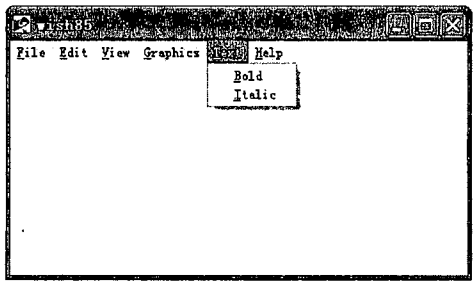


图 18.22　包含被激活的菜单的下拉菜单

如果用户在菜单条目上单击了鼠标 1 键，并在另一个菜单条目上移动了光标，那么旧的条目就会处于未激活状态，而新的条目会激活它的菜单。这使得用户可以通过在菜单栏上移动光标来浏览所有的菜单。

如果用户在一个条目上释放了鼠标键，菜单将会被激活，同时用户不能对应用程序进行任何操作，直到菜单处在未被激活的状态为止，对于这种状态，用户可以通过调用任意一个条目或者在菜单以外单击鼠标键而不调用任意一个条目来实现。像这样的状态被称为模态用户界面元素，即用户必须响应应用程序中一个特定的部分，并且在这种响应之前不能对程序的其他部分进行操作。菜单和对话框是模态用户界面元素的示例，可以用将在第 27 章中介绍的攫取机制来实现。

在 Tk 中，菜单栏是顶层窗口的一个集成部分。顶层窗口在不同的平台环境中用不同的方式管理菜单栏的位置。菜单栏能通过创建一个菜单来构建，接着运用顶层窗口的-menu 选项使得菜单栏置于顶层。在创建图 18.22 的脚本里，-menu 和-underline 选项在每一个条目中都被指定，同时被指定的，还有每一个条目的标签字符串。-menu 选项指定了和条目有关联的菜单。-underline 选项用于在显示条目标签时为索引字符加上下划线。这种字符用于键盘快捷操作，在 18.12.3 节中将有介绍。

18.12.2　级联菜单

级联菜单是另外一个菜单的子菜单。它和自己的上层菜单中的级联菜单条目有关联。注意，菜单栏只是一个级联菜单的列表。当光标指向菜单中的级联条目时，相应的菜单将在级联选项的右边被激活，如图 18.23 所示。用户可以向右移动光标到级联菜单，然后调用里面的条目。当级联菜单中的一个条目被调用后，它和它的上层菜单都会处于未激活状态。创建级联菜单所需要的只是定义这个级联菜单，并在它的上层菜单中创建一个级联条目，使用级联条目中的-menu 选项可以指定这个级联菜单的名称。菜单可以有任意多条

目，但出于方便的考虑，通常的经验是条目最多有三个。

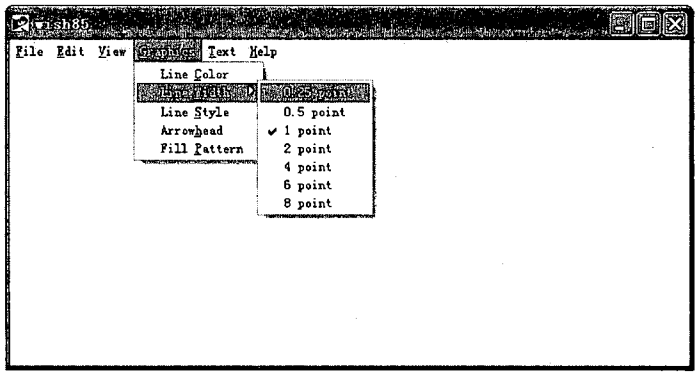


图 18.23　级联菜单

下面这个示例显示了为创建级联菜单对创建图 18.22 的脚本进行的一些改动。

```
menu .mbar.graphics
...
.mbar.graphics add cascade -label "Line Color" \
    -menu .mbar.graphics.color -underline 5
.mbar.graphics add cascade -label "Line Width" \
    -menu .mbar.graphics.width -underline 5
...
.mbar.graphics.width add radiobutton -label "0.25 point" \
    -variable lineWidth -value 0.25
...
```

18.12.3　键盘遍历和快捷键

使用一项名为“键盘遍历”的技术可以在没有鼠标时激活或不激活下拉菜单。每一个菜单条目都有一个字母被选为遍历字符，这一点是使用-underline 选项来实现的，它在这个条目的窗口里被标上下划线。如果在 Alt 键被按住时输入了这个字母，那么相应的条目菜单就会被激活。或者用户可以按下 F10 键来激活菜单栏最左端的菜单。当一个菜单被激活时，用户可以用箭头键在菜单和它们的条目之间移动。右箭头和左箭头可以在顶层条目上右移或左移，使得前一个条目处于未激活状态，而当下的条目则被激活。上下两个箭头键可以在一个菜单的不同条目之间移动，激活上一个或者下一个条目。您可以按下 Return 键来调用已被激活的菜单条目，或者按 Escape 键来放弃菜单遍历而无需调用任何条目。遍历字符也可以用-underline 选项对每一个单独的菜单条目进行定义，如创建图 18.22 的代码。当菜单条目被显示出来时，遍历字符下会有下划线，且如果在菜单被激活时用户输入它或者它相应的小写字母，那这个条目将被调用。

在很多时候，可能通过输入键盘快捷键来调用菜单条目的功能，而无需激活菜单。如果有一个菜单输入框的快捷键，这个快捷键的具体形式将在菜单条目的右端显示出来(例如，想显示 Ctrl+Z 来作为 Undo 菜单的快捷键)。这种组合键可以在应用程序中被输入，来调用和菜单条目一样的功能(例如，在按住 Ctrl 键时输入 Z 可以调用 Undo 操作，无需再在菜单中去找)。菜单条目的-accelerator 选项指定了要在一个条目右端显示的字符串。常见的修饰符包括 Control、Ctrl、Shift、Command、Cmd、Option 和 Opt，它们可以组合使用。

提示：在 Mac OS X 中，这些修饰符自动映射到菜单上出现的修饰符。

按键绑定必须被定义，才能让快捷键工作。这一点可以使用 bind 命令来实现，例如：

```
bind . <Control-Key-z> {.mbar.edit invoke "Undo"}
```

当键盘事件被绑定到“.”时，顶层窗口中的事件会被捕获，无论哪一个内部窗口有键盘焦点。关于 bind 命令和事件扩展的更多信息可参见第 22 章。

18.12.4 针对平台的菜单

每一个平台都有一个或多个菜单，Tk 采用特定的方法来控制它们。为了充分利用这个特征，必须为创建的菜单组件给定一个特定的名字，以支持这些特定的菜单。在各种情况下，这些菜单组件必须使用菜单栏菜单的名字来创建，这些菜单要和特定的名字联系起来(也就是说，应用程序上的“.menuBar.special”或者第二个顶层窗口中的“.toplevel.menuBar. special”)。

在 X11 窗口系统里，Help 菜单应该总在窗口的右端出现。当使用“.help”作为执行 Help 菜单的菜单组件名称的最后一部分时(例如，应用程序的主窗口的“.mbar .help”，或者第二个顶层窗口的“.docl.mbar.help”)，Tk 能够自动地实现这一点。

在 Windows 系统中，可以访问窗口的系统菜单，这个菜单通过在窗口的标题栏上单击程序图标来激活。加入的任何条目都会在标准的 Windows 条目的后面。要访问系统的菜单，可以使用“.system”作为菜单名称的最后一部分(例如“.mabr.system”)。

在 Mac OS X 中，可以通过创建后缀名为“.help”(例如“.mbar. help”)的菜单组件的方式来向标准的 Help 菜单加入条目。Mac OS X 还有所谓的应用程序菜单，此菜单是菜单栏上的第二个菜单，菜单标题与应用程序同名。要访问应用程序菜单，需要创建后缀名为.apple(如“.mbar.apple”)的菜单。您往“.apple”菜单加入的任何条目都会在系统标准条目之前出现。

提示：真正访问程序菜单(从左数第二个)的是“.apple”菜单，而不是含有苹果图标的最左端的菜单。这个名字来源于 OS X 系统之前的时期，那时针对应用程序的条目确实在有苹果图标的菜单下面运行。

Mac OS X 的应用程序菜单也提供了一个标准的 Preferences 条目。这个条目通常是不可用的，除非定义了名为“::tk::mac::ShowPreferences”的过程。通常，您可以定义这样一个过程来显示任何已经完成的偏好设置对话框。

18.12.5 弹出式菜单

tk_popup 应用是激活弹出式菜单的一种常规而简便的方法。弹出式菜单通常和 Windows 和 Unix 系统下的鼠标 3 键，也就是鼠标右键绑定在一起。而在 Mac OS X 上，它们应该响应鼠标左键(唯一的键)的单击，此时 Ctrl 键被按下了，或者多键鼠标的右键，也就是 2 键，也被按下了。因此，在一个独立于平台的应用程序里，应该使用 tk

windowingsystem 命令来决定程序是否正在原始的 Mac OS X 窗口系统中被使用，如果是，那么应该有一个 aqua 的返回值。

弹出式菜单也对鼠标按钮被按下时的鼠标光标位置特别敏感。这意味着显示的菜单有和鼠标光标下的条目关联的动作。tk_popup 应用使绑定鼠标键到一个菜单上变得很容易。

```
if {[tk windowingsystem] == "aqua"} {
    bind .lbox <ButtonPress-2> {tk_popup .menu %X %Y}
    bind .lbox <Control-ButtonPress-1> {tk_popup .menu %X %Y}
} else {
    bind .lbox <ButtonPress-3> {tk_popup .popup_menu %X %Y}
}
```

当合适的鼠标键单击“.lbox”组件时，“%X”和“%Y”参数被鼠标光标的 x 和 y 坐标取代。在此示例里，tk_popup 在相对鼠标光标位置合适的位置调用了.popup_menu，通常是在光标的右下方。第 22 章将详细地介绍 bind 命令。

提示：独立于平台的绑定也能通过对虚拟事件的定义来控制，这一点在第 22 章中也有介绍。

18.13　分栏窗口

一些常用的应用程序，如电子邮件客户端和文件浏览器，有和图 18.24 相似的窗口结构。主窗口被分为两个或更多的窗格，一条可移动的分隔线把窗格分开。在 Tk 中，分栏组件来完成这一过程。分栏可以是水平分栏，也可以是垂直分栏，这取决于您是把分栏窗口的-orient 选项设置为 horizontal 还是 vertical。您可以将-showhandle 选项设定为一个布尔运算值，以此来判定是否想要在每一个窗格边框上显示一个调整大小的控制柄。

每一个窗格都是一个包含另一个组件的小格子。通常，您可以将每一个窗格的内容整合进一个框架组件，然后使用 add 子命令将每一个框架嵌入分栏窗口中。(第 21 章将解释如何在一个框架里布置组件。)图 18.24 的左窗格含有一个显示文件列表的列表框，而右窗格包含一个能显示所选文件内容的文本组件。

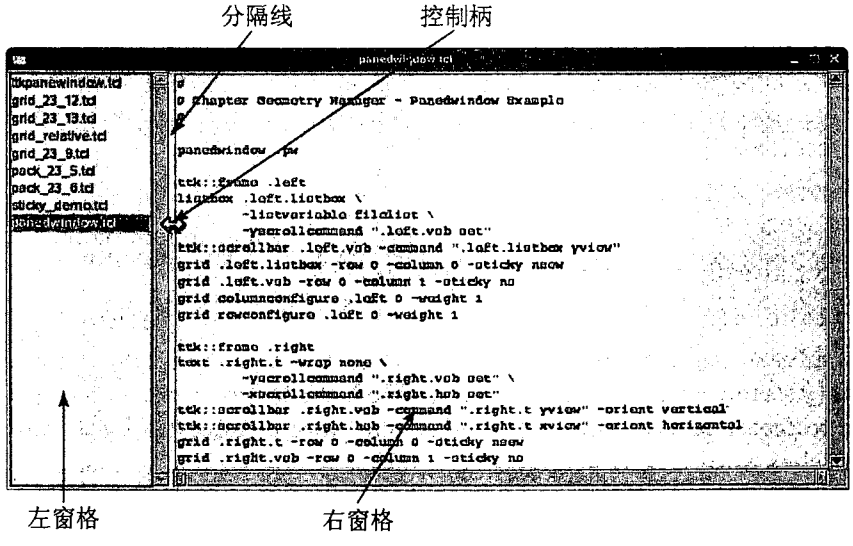


图 18.24　分栏窗口将一系列的组件横排或者竖排

下面这段脚本显示了如何往分栏窗口中添加已有的组件。

```
# The .left frame contains a listbox and scrollbar
# The .right frame, a text widget and scrollbars
# These are placed in a panedwindow:

panedwindow .pw -orient horizontal
.pw add .left .right
```

您添加的每一个窗格都会被一系列选项控制。当用 add 来添加窗格时，可以设定选项的值，或者稍后用 paneconfigure 子命令来添加。您可以使用 panecget 子命令来求得一个窗格选项的值。例如：

```
    .pw panecget .left -sticky
⇒ nesw
    .pw paneconfigure .left -minsize 2i
```

支持的窗格选项如下所述。

- `-after widget`
  在 *widget* 后面插入组件，这个组件应该以被分栏窗口控制的组件来命名。
- `-before widget`
  在 *widget* 前面插入组件，这个组件应该以被分栏窗口控制的组件来命名。
- `-height size`
  为组件设定一个高度，这个高度用一个屏幕距离来表示。如果 *size* 是一个空字符串，或者没有设定-height，那么组件内在要求的高度就会作为初始高度，这个高度可以稍后通过移动窗格边框而调整。
- `-hide boolean`
  控制一个分栏的可见性。当 *boolean* 为真时，窗格不可见，但仍然会出现在窗格列表中。
- `-minsize size`
  为窗格的尺寸设定一个下限，即 *size*，用屏幕距离来表示。
- `-padx size`
  设定组件周边的水平补白，用屏幕距离来表示。
- `-pady size`
  设定组件周边的垂直补白，用屏幕距离来表示。
- `-sticky style`
  控制组件的位置和扩展，当一个组件的窗格超过了自己要求的尺度时可以扩展这个组件。*style* 是一个不包含或包含 n、s、e 或 w 中某些字符的字符串。每一个字母代表组件粘贴到的边(北、南、东、西)。如果 n 和 s(或者 e 和 w)被指定了，窗口就能往该方向扩展并占据窗格的全部高度(或宽度)。
- `-stretch when`
  控制多余的空白是怎样分配到每一个窗格的。您可以将窗格设置为 always 或 never。另外，如果窗格是 first、last 或者 middle 窗格，可以指定窗格扩展的范围。

- -width size
  为窗口指定一个宽度，用屏幕距离来表示。如果 *size* 是一个空白字符串，或者没有指定-width 时，那么组件内在要求的宽度就会作为初始宽度，这个宽度可以稍后通过移动窗格边框而调整。

以前加入分栏窗口的组件能用 forget 子命令来移除。您还可以使用 panes 子命令来获得被分栏窗口控制的组件列表，这些组件按照它们被显示的顺序出现。

```
    .pw panes
⇒ .left .right
```

提示： 分栏窗口既是一个组件也是一个几何管理器。在第 21 章中将有关于几何管理器的重点介绍。

18.14 标准对话框

Tk 提供了一套实用程序和对话框。下面将介绍三个对话框，未介绍的内容请参考文档。您也可以创建自定义的对话框，这个话题将在第 27 章中讨论。

在一个应用程序里，经常需要通告一个状态，或询问一个需要即时注意的问题。这在 Tk 里可以很容易地完成，只要使用 tk_messageBox 对话框。调用这个过程会给出一个含有一则信息和一组按钮的对话框，具体取决于设定的-type。-icon 选项定义的图像位于消息文本旁。此图像被用来帮助用户决定信息的重要性，因此可能的值有 error、info、question 和 warning。此图像根据平台环境而变，并且它在某个环境的所有应用程序中都能进行特定的标准化。图 18.25 显示了一个简单的 yesno 消息框。-parent 选项指定了一个应用程序组件，这个组件被 tk_messageBox 用于调整对话框的位置。这个对话框置于顶层组件的上方的中间。这个消息框是模态的，它阻止了 Tcl 应用程序的进一步执行，直到用户单击按钮中的一个进行响应。返回值是一个显示按钮是否按下的字符串，如 yes 或者 no。

下面这段脚本显示了一个消息框和响应。这个函数基于用户按住的按钮返回一个值。

```
tk_messageBox -type yesno -icon question \
-message "Do you like coffee?" -parent .
⇒ yes
```

为打开和保存操作选择操作系统文件是常见任务，因此每一个平台环境都有一个标准的方法来显示用于选择文件的对话框。这些对话框在所有的应用程序上是共用的，为用户减轻了学习的负担。Tk 提供了利用 tk_getOpenFile、tk_getSaveFile 和 tk_chooseDirectory 过程访问这些共用对话框的功能。当这些过程被调用时，Tk 应用程序会一直等待，直到用户完成了对文件的选择。这个过程随后会返回用户所选择的文件(如果用户没选择文件，则返回空白的字符串)，然后 Tcl 脚本会继续执行。在创建图 18.26 的代码中，-initialfile 选项被用来选出对话框，这样就能显示出给定文件的名字了。-filter 用来筛选出对当前的应用程序有意义的文件。

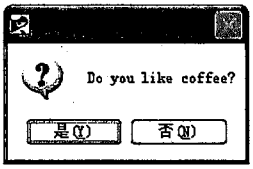


图 18.25　Tk 内建的 messageBox 对话框的示例

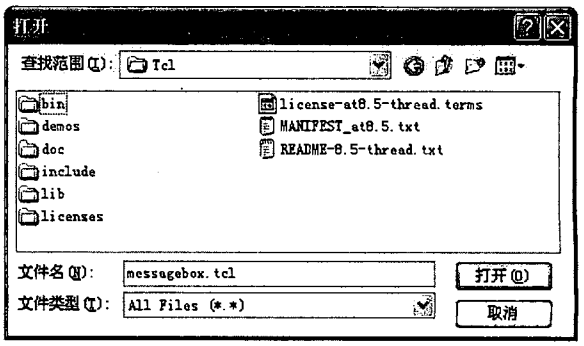


图 18.26　调用基本的文件导航对话框

下面的命令创建了一个用于打开已有文件的文件对话框。调用会返回被选中的文件，如果对话框已经被消除了，则返回空字符串。很多初始的状态(这里显示的-initialfile)和筛选器可以通过对调用设置相应选项来配置。

```
tk_getOpenFile -initialfile messagebox.tcl
    -filetypes {{Tcl .tcl} {All *}}
```

18.15　其他的常见选项

本章中介绍的组件包括了大多数常见的选项，如颜色和字体。本节将介绍其他一些被许多组件支持的选项。

18.15.1　组件状态

交互式组件和菜单条目有一个相关的-state 选项。-state 的默认值是 normal，此时默认的组件类绑定使组件能响应用户的动作。例如，按钮通过调用自己的-command 脚本来响应鼠标的单击，列表框让用户能滚动并选择自己的条目，输出框组件使得用户能编辑文本，等等。将-state 选项的值设为 disable 会导致默认的组件类绑定让组件不响应用户输入，此时大多数组件也会改变它们的外观，例如“灰显”。一些交互式组件也支持一个 active 值，这个值主要被默认的类绑定用来突出显示组件(当鼠标光标处在组件上方时)。

-state 选项最常见的用法是在特定的情况下使应用程序的某种功能失效，并在其他情况下使此功能有效。例如，如果用户已经在应用程序中选择了一些文本，则启用 Copy 按钮(将-state 设置为 normal)，否则，禁用该按钮(将-state 设置为 disabled)。

其他的组件有一些其他的状态，例如输入框或调节框的 readonly 和画布的 hidden，在本书的相关章节和参考文档中能找到关于这些状态的更多信息。

18.15.2　组件尺寸选项

许多组件支持-width 选项，一些组件还支持-height 选项，以此来设定一个预期的最小

尺寸。如果组件含有文本，那么这些值会经常以数值的形式(例如字符宽度和行的高度)被解释，因此实际的尺寸依赖于字体和文本的大小。如果组件含有图像或者位图，这个值通常以像素为单位。如果这个组件不够大，以至于不能显示所有的内容，那么它通常会用后文介绍的-anchor 来修改内容。

虽然可以试着计算或者预测组件的尺寸，但在通常情况下，让诸如标签和按钮的组件使用选项默认值(0)是最简单的。这时，组件会自动调整尺寸来适应它们的内容。而对于类似输入框或者文本组件的组件，会经常使用几何管理器来调整基于窗口尺寸的组件，这样将会覆盖为-height 和-width 设定的任意值。

18.15.3　锚定选项

锚定位置决定了如何将一个对象和另一个联系起来。例如，如果按钮组件的窗口大于组件文本所需要的尺寸，锚定选项可以被用来指定文本在窗口中应该被放置的位置。锚定位置也能被用于其他目的，例如告知一个画布组件相对于和条目相关的点，该在何处放置位图，或者告知包装几何管理器在框架内何处可以放置一个窗口。

锚定位置可以用下面的方向点中的一个来指定。

- n：对象上边缘的中心。
- ne：对象的右上角。
- e：对象的右边缘的中心。
- se：对象的右下角。
- s：对象的下边缘的中心。
- sw：对象的左下角。
- w：对象的左边缘的中心。
- nw：对象的左上角。
- center：对象的中心。

锚定位置给定了和对象相连的点的位置，就像一颗图钉钉在那个点一样，然后根据它将对象钉在某处。例如，为按钮指定值为 w 的-anchor 选项意味着按钮的文本或位图被钉在了左侧的中心，并且那个点是在窗口的相应点上的。因此，w 意味着文本或位图垂直居中，并且和窗口的左边缘对齐。对画布组件中的位图条目而言，-anchor 选项说明了位图相对于和该条目有关联的点的位置。此时，w 意味着位图左侧的中心应该被钉在那个点，这样位图事实上就处于该点的东方。图 18.8 和图 18.9 显示了-anchor 选项的用途，它用来使按钮与窗口的左侧(朝西)对齐。

18.15.4　内部补白

许多组件支持-padx 和-pady 选项，它们能控制内部补白。-padx 选项接收一个屏幕距离，来指定组件的边缘和内容之间水平间距的最小值；-pady 选项指定组件的边缘和内容之间垂直间距的最小值。

18.15.5 光标选项

Tk 中的每一个组件类都支持-cursor 选项，该选项决定了当光标在组件上方时应该显示的图像。如果-cursor 选项没被定义或者它的值是一个空字符串，组件可以使用它的上级光标。另外，光标选项的值必须是一个合适的 Tcl 列表，列表要采用下列几种格式之一。

- *name fgColor bgColor*
- *name fgColor*
- *name*
- *@sourceFile maskFile fgColor bgColor*
- *@sourceFile fgColor*
- *@sourceFile*

在前三种形式中，*name* 指某个预定义的光标。您可以在光标参考文档中找到所有预定义名称的完整列表。

如果 *name* 的后面跟的是另外两个列表元素，例如在下面这个命令中，第二个和第三个元素指定了光标的前景色和背景色。

```
.f config -cursor {arrow red white}
```

正如所有的颜色值一样，它们含有 18.3 节中介绍过的形式。如果只提供一种颜色值，那么它将为光标给出前景的颜色，而背景将成为透明色。如果没有给出任何的颜色值，前景将是黑色，而背景将是白色。

如果-cursor 值中的第一个字符是@，光标的图像来自位图格式文件而不是 X 光标字体文件。如果两个文件的名字和两种颜色被指定了值，例如：

```
.f config -cursor \
    {@cursors/bits cursors/mask red white}
```

第一个文件是一个包含光标形式的位图(1 代表前景，0 代表背景)，第二个文件是一个掩码位图。掩码为 0 时，光标是透明的；当掩码为 1 时，光标会显示前景或者背景。如果只给定了一个文件名和一种颜色，光标会有一个透明的背景。这种特征只在 X 窗口系统中才被支持。

最后一种形式只在 Windows 系统中使用，它会从由 *sourceFile* 指定的文件中下载一个 Windows 系统光标(使用扩展名.ani 或.cur)。

在 Windows 或 Macintosh 系统中，一些光标被映射为原始的光标。光标的外观取决于用户的喜好。另外一些光标被定义为只在 Windows 或 Macintosh 系统中才能被使用。您可以在光标参考文档中找到一个完整的列表。

第19章 主题组件

主题组件点经典 Tk 组件集中采用不同的方法来处理问题。它们在可能的范围内，区别了我们组件应用的代码和显示外观的代码。经典组件自选项数据库或者通过直接修改每个组件选项的方式来配置，主题组件的尺寸、形状、颜色、字体等由它们的样式决定。主题组件允许应用程序的外观由环境(例如，Windows 用户首选项或者应用程序的中心)样式来控制。由此产生的外观在应用程序中更一致，也与用户窗口系统的原生外观更为一致。

19.1 比较经典组件和主题组件

主题组件的主要优点是选择适合用户当前桌面环境的主题，来给出应用程序的原生外观。Microsoft Windows 和 Mac OS X 的经典组件尽量模仿按钮、滚动条、复选按钮和单选按钮等原生组件，但这些外观可以在更新的窗口系统版本中更新。这一点在基于 X 的经典组件上特别明显，该组件仍然大部分基于 Motif 组件工具集的外观。图 19.1 显示了经典 Tk 复选按钮组件及其在不同平台中针对不同主题的变体。在一些情况下，不同平台之间的差别是很小的，但一些小的变化可能会导致巨大的差异，例如 Mac OS X 复选按钮上的背景混合。

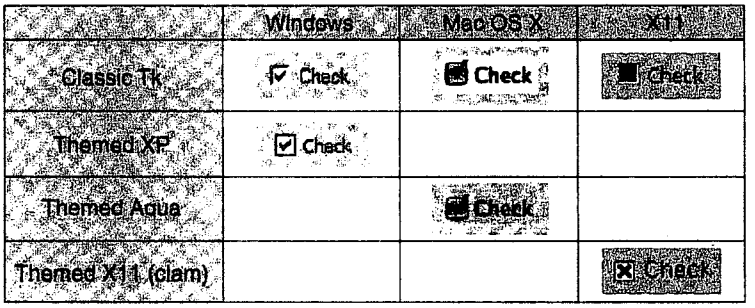


图 19.1 经典和主题化的复选按钮

主题组件的另一个优势是它们比起经典组件更容易使用。主题组件只需很少的配置，就能给应用程序一致和有吸引力的外观，特别当您希望组件的子集能有特殊的外观或行为时。要自定义应用程序中的经典组件，可以单独配置每一个组件的选项，也可以由 Tk 选项数据库(参见第 28 章)来配置。对主题组件而言，可以定义一种新的样式(通常继承自已有样式的定义)来自定义新样式的特定配置选项，然后将新的样式赋给所需的单个组件。这

个过程在 19.9 节里将有详细介绍。

表 19.1 列出了经典组件以及它们和主题组件之间的比较。一些组件和基本的 Tk 组件集是相同的，另外一些则提供在经典 Tk 组件中没有的行为。一些经典组件，如 canvas 和 text 类，没有等价的主题组件，而主题组件包括一些新的组件类型，如 combobox、notebook 和 treeview 类。

表 19.1 经典组件类和它们相应的主题组件类

经典组件	主题组件
button	ttK::button
canvas	
checkbutton	ttk::checkbutton
	ttk::combobox
entry	ttk::entry
frame	ttk::frame
label	ttk::label
labelframe	ttk::labelframe
listbox	
menu	
menubutton	ttk::menubutton
	ttk::notebook
panedwindow	ttk::panedwindow
	ttk::progressbar
radiobutton	ttk::radiobutton
scale	ttk:scale
scrollbar	ttk::scrollbar
	ttk::separator
	ttk::sizegrip
spinbox	
text	
toplevel	
	ttk::treeview

同时含有经典 Tk 类和主题组件类的组件功能相同，但不可互换。主题组件没有用来控制颜色、边界等的选项；相反，组件的这些属性由主题控制。同时，一些主题组件有不同于其他经典主题组件的组件命令。在本章后文会着重介绍主题组件添加的新组件类。

提示： *如果决定在一个已有 Tk 应用程序中用主题组件来替换经典组件，以便更新程序的外观，应该仔细阅读参考文档，以确认该主题组件与被替换的经典组件之间的差别。*

19.2　组合框

ttk::combobox 组件包括一个下拉列表框和一个输入框组件。这个组件能用于从一系列固定值中选择值，类似于调节框，但它也能直接选择一个值，而无需逐一查看每个值。或者它能用于累积直接输入输入框的历史数值。图 19.2 显示了一个能进行检索的组合框。在这个示例里，每当输入一个新的检索字符串，字符串会被添加到组合框的值中。单击输入框右侧的向下箭头会显示出一个包含检索历史的下拉列表框。选择这些值中的一个可以将输入框更新为新值。下列脚本创建了图 19.2 中的组合框。

```
ttk::combobox .cb
grid .cb
proc UpdateCombo {w} {
    set new [$w get]
    set values [$w cget -values]
    if {$new ni $values} {
        set values [linsert $values 0 $new]
        $w configure -values $values
    }
}

bind .cb <Return> {UpdateCombo %w}
```

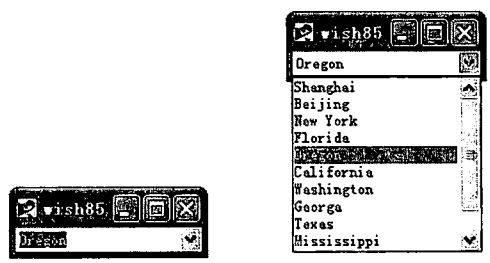


图 19.2　主题组合框组件的示例

19.3　记事本

ttk::notebook 组件能将窗口划分为几页来管理，每一页会与用户可选的标签关联，且一次只显示一页。这个组件通常被用在复杂的对话框中，它还有其他一些有用的应用程序，例如图 19.3 所示的简单文本编辑器。

记事本在作为其他组件的控制器方面类似于分栏窗口。当标签被选中时(程序调用或者用户单击)，记事本会显示一个窗口，隐藏此前显示的子窗口。一般而言，可以将每一个窗格的内容分配到一个 ttk::frame 或 frame 组件，然后用 add 子命令向记事本添加每个框架。(第 21 章将介绍如何在框架中布置组件。)

标签由标识符给出。下面这些方法可以用于指定标签标识符。

- 一个整型索引，第一个标签为 0。
- 由记事本控制的子组件名。

- 以@*x*, *y* 的格式表示的屏幕位置，其中 *x* 和 *y* 的值为相对于记事本组件左上角的距离，单位为像素。
- 用字母表达的字符串 *current*，它指向当前选中的标签。

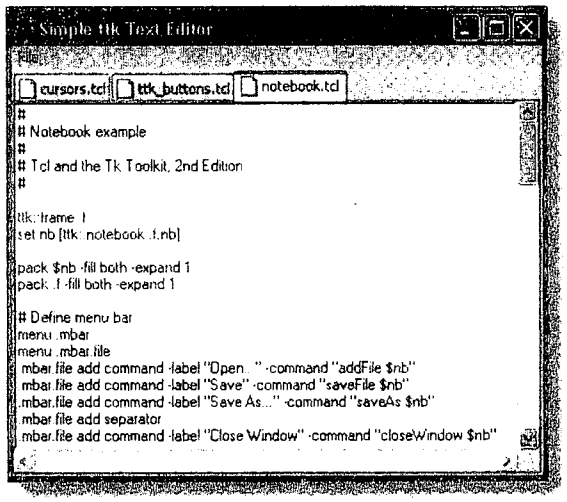


图 19.3　主题记事本组件的示例

您添加的每个子窗口都被一些选项控制。在使用 add 增加窗口时，可以设定选项值，或者在稍后使用 tab 子命令来设定，该命令将标签标识符作为它的第一个参数。

```
notebook tab tabid ?option? ?value option value ...?
```

可用的标签选项如下所述。

- `-compound style`
  指定当-text 和-image 同时给出时，如何以文本为基准显示图像。19.10 节将介绍合法的值。
- `-image image`
  指定在标签上显示的图像对象名。
- `-padding space`
  指定添加到标签窗口外围的多余空间，单位为屏幕距离。补白用最多 4 个值指定长度：left、top、right 和 bottom。如果指定值少于 4 个，那么 bottom 会默认为 top，right 会默认为 left，top 会默认为 left。
- `-state state`
  指定状态，其值为 normal、disabled 或 hidden 中的一个。如果状态为 disabled，该标签就不可选。如果状态为 hidden，该标签就不会显示出来。
- `-sticky style`
  如果组件标签窗口大于它要求的尺寸，则控制组件的位置并扩展该组件。*style* 是一个不包含字符或者包含 n、s、e 或 w 中某些字符的字符串。每个字母都给出了组件会“粘贴”到的一条边(北、南、东或西)。如果指定了 n 和 s(或 e 和 w)，窗口会扩展来占满窗口的整个高度(或者宽度)。

- -text string
  指定要在标签上显示的字符串。
- -underline index
  指定字符的整形索引(从 0 开始)，并为文本字符串添加下划线。如果 ttk::notebook::enableTraversal 被调用，带下划线的字符可以用作助记符号。

insert 组件命令可以在指定位置插入标签。

```
notebook insert position widget ?option? \
   ?value option value ...?
```

position 可以是一个从零开始索引的整数值，或者是关键字 end，来向末尾添加标签。如果 *widget* 已由记事本控制，那它将移动到指定位置。

您可以使用 hide 组件命令来隐藏标签，这一点和将标签的-state 设为 hidden 的作用是相同的。另一方面，forget 组件命令从记事本上删除标签，且不再映射和管理关联的子组件。

如果给出支持的标签标识符，index 组件命令会返回标签的整型索引；如果将 end 作为参数，它会返回被管理的标签的数目。tabs 组件命令会返回一个列表，内容为由记事本控制的组件。

在选定新标签后，记事本组件会生成一个名为<<NotebookTabChanged>>的虚拟事件。第 22 章将介绍虚拟事件。您也可以使用下述命令使包含记事本组件的顶层窗口的键盘遍历可用。

```
ttk::notebook::enableTraversal notebook
```

这条命令会自动扩展包含记事本的顶层窗口的绑定，以支持下述行为。

- Ctrl+Tab 选择位于当前选中标签之后的标签。
- Shift+Ctrl+Tab 选择位于当前选中标签之前的标签。
- Alt+*k*(这里的 *k* 是用标签的-underline 选项指定的下划线字符)会选中相应标签。

19.4 进度条

ttk::progressbar 组件为用户提供了长时间运行的操作的状态的虚拟反馈。进度条的虚拟外观由-orient 选项决定，选项值可以是 horizontal 或 vertical，它的-length 用来指定长边的长度，单位为屏幕距离。它的行为由-mode 选项决定，如果希望它显示到某个已知点的累积进度，该选项的值可以设为 determinate，如果无法预测操作何时完成，该选项的值可以设为 indeterminate。

对于 determinate 进度条，-maximum 选项是一个表示操作完成点的浮点数。-maximum 的默认值为 100.0，能用于显示百分数；您也可以给出一个显示时间的数值、要处理的记录数或别的标识。使用-value 选项，可以说明应用程序完成了多少进度。您也可以调用 ttk:: progressbar 的 step 子命令，指定增量值(默认为 1.0)。

```
progressbar step ?amount?
```

另外，可以使用-variable 选项来指定希望进度条的-value 选项同步显示的变量名(由全

局命名空间给出)。

使用 indeterminate 进度条，就不会有程序的预期完成点。这时，进度条的用途只是为用户提供一些虚拟的显示，说明这个程序正在进行中且没有被冻结。要使用 indeterminate 进度条，只需要在操作开始时调用 start 子命令，在结束时调用 stop 子命令。

19.5 分隔符

ttk::separator 组件显示了一个简单的水平或垂直分隔符，具体的形式由-orient 选项指定。您可以用它来提供复杂界面上的逻辑单元之间的可视化分界。分隔符不存在交互式控制和另外的提示选项。

19.6 尺寸控制柄

ttk::sizegrip 组件可以用于拖住窗口的右下角，对其进行调整。它允许用户按下并拖曳控制柄，调整其中的顶层窗口的大小。只有顶层窗口右下角的尺寸控制柄才支持相应的调整行为。

您必须明确地将尺寸控制柄放在窗口的准确位置上。例如，如果使用 pack 来给出顶层窗口底部状态栏中尺寸控制柄的位置，可以使用如下代码：

```
set statusbar [ttk::frame $top.statusbar]
pack $statusbar -side bottom -fill x
set grip [ttk::sizegrip $statusbar.grip]
pack $grip -side right -anchor se
```

另一方面，如果使用 grid，则代码会变为：

```
set grip [ttk::sizegrip $top.grip]
grid $grid -row $lastRow -column $lastColumn -sticky se
```

提示： *在 Mac OS X 里，顶层窗口自动包含一个内建的尺寸控制柄。由于内建的控制柄恰好遮住了组件，添加 ttk::sizegrip 并没有坏处。*

19.7 目录树

ttk::treeview 组件是用来显示一列或多列信息的有力工具，可以在带有动态折叠和展开功能的层级结构中使用。

19.7.1 管理目录树条目

目录树的基本构造单元是条目，它代表目录树上的一行信息。每个条目都有一个文本标签、一幅可选图像和一些可选数值，它们会在标签后面以目录树连续列的形式显示出来。目录树中的每个条目也必须有唯一的标识符；可以在创建条目时给出自己的标识符，或者让目录树自动创建一个。

条目可以嵌套，这样一个条目就可以视为一个叶节点，或者视为含有一个或更多子条目的父条目。创建目录树实例时，每个目录树会自动创建一个根条目；它的标识符是空字符串。这个根条目自身在目录树中没有显示。根条目的直接子条目会出现在层级结构的最上层。

您可以用 insert 子命令向目录树组件增加条目。

```
treeview insert parent index ?-id id? ?option value ...?
```

parent 参数是新条目的父条目的标识符。*index* 是新条目在父条目中的位置；它可以是一个从 0 开始的整数，或者是关键字 end，用来在所有已有子条目后面安置条目。您可以用-id 选项为新条目明确提供一个唯一的字符串标识符，也可以忽略这一步，而让目录树自动赋给它一个标识符；这个标识符是命令的返回值。每个条目也支持一些决定外观的选项。当插入一个新条目时，可以设定这些选项，或者在稍后用目录树的 item 子命令来设定。您还可以使用 item 子命令来查询条目的一个或所有选项。

```
treeview item id ?option? ?value option value ...?
```

可用的条目选项如下所述。

- -text——目录树中的条目所显示的文本字符串。
- -image——标签左侧所显示的可选图像对象名。
- -values——一列与条目相关的数值。
- -open——一个布尔值，用来表明条目的子条目应该显示出来(布尔值为真)，还是应该隐藏(布尔值为假)。
- -tags——一些与条目关联的标记，下面将会讨论这个选项。

例如，下面的代码会创建一个目录树，并为其添加三个顶层条目。它还会为每个顶层条目创建一些子条目。

```
ttk::treeview .tree -columns {capital} \
    -xscrollcommand {.hbar set} -yscrollcommand { .vbar set}
ttk::scrollbar .hbar -orient horizontal \
    -command {.tree xview}
ttk::scrollbar .vbar -orient vertical \
        -command {.tree yview}
    grid .tree -row 0 -column 0 -sticky nsew
    grid .vbar -row 0 -column 1 -sticky ns
    grid .hbar -row 1 -column 0 -sticky ew
    grid columnconfigure .  0 -weight 1
    grid rowconfigure    .  0 -weight 1

    .tree insert {} end -id newengland -text "New England"
    .tree insert {} end -id midatlantic -text "Mid Atlantic"
    .tree insert {} end -id pacific -text "Pacific"

    set states {
        newengland  me  Maine              Augusta
        newengland  nh  "New Hampshire"    Concord
        newengland  vt  Vermont            Montpelier
        newengland  ma  Massachusetts      Boston
        newengland  ri  "Rhode Island"     Providence
        newengland  ct  Connecticut        Hartford
        midatlantic ny  "New York"         Albany
```

```
    midatlantic nj   "New Jersey"        Trenton
    midatlantic pa   Pennsylvania        Harrisburg
    pacific     wa   Washington          Olympia
    pacific     or   Oregon              Salem
    pacific     ca   California          Sacramento
}

foreach {region id state capital} $states {
    .tree insert $region end -id $id -text $state \
        -values [list $capital]
}

.tree column #0 -minwidth 150
.tree column capital -minwidth 150
.tree heading #0 -text "State"
.tree heading capital -text "Capital"
```

上述脚本运行的结果如图 19.4 所示。

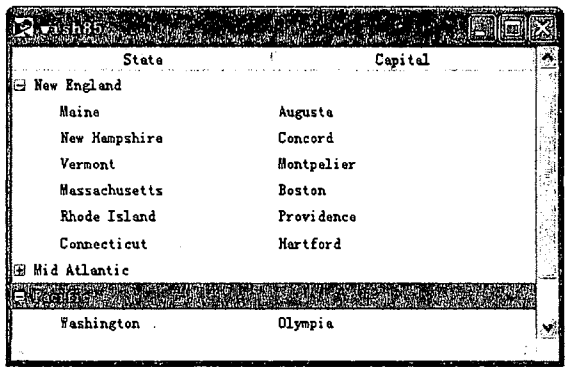


图 19.4　目录树组件的示例

您可以将-open 属性设为真或假，以使得程序能展开或折叠一个条目。默认的目录树组件类绑定会自动执行用户交互作用的折叠/展开的行为；在一个条目上单击鼠标左键能切换该条目的-open 选项。组件也能生成<<TreeviewOpen>>和<<TreeviewClose>>虚拟事件，来响应这些动作，您还可以执行自定义的条目开/关行为。关于虚拟事件和执行响应事件脚本的详细信息可参见第 22 章。您也可以使用目录树的 see 子命令来保证一个特定条目是可见的；它将全部的上层条目设为-open true，并在需要的时候滚动组件来使条目在屏幕上显示出来。

treeview see *item*

您可以用 move 子命令将一个已有条目移到条目分层结构中的新位置。

treeview move *item parent index*

item 参数是要移动的已有条目的标识符，*parent* 和 *index* 参数会与 insert 一起被解释。不可以将条目移动到其下层条目下。

delete 子命令以条目标识符为参数，并删除对应条目和它们所有的下层条目。相反，detach 子命令以条目标识符为参数，使相应条目和目录树“断开链接”。这样条目和它们的下层条目再也不会显示在目录树上，但它们仍然是存在的。您可以用 move 子命令将它

们和目录树中的任意一个位置重新关联起来。

您可以使用 index 子命令，在父条目中确定条目的整型索引。next 和 prev 子命令能返回条目的后一个或前一个并列条目的标识符；如果该条目是父条目中的最后一个或是第一个，那么将分别返回一个空字符串。parent 子命令会返回父条目的标识符，或者如果该条目在分层结构的顶端，则将返回一个空字符串。children 子命令会按照子条目出现的顺序返回它们的标识符；您也可以给出一些条目标识符，此时已有子条目会和目录树失去关联，而给出的条目会成为新的子条目。

19.7.2 控制目录树的列和标题

目录树可以显示任意多列。第一列会包含目录树条目自身；该列会显示与该条目关联的图像，图像的前面是文本标签，分别由条目的-image 属性和-text 属性指定。这两个属性都是可选的。其他列使用条目的可选-values 属性来显示元素。列也有可配置的标题。

目录树组件的-columns 属性决定了目录树包含的数据的列数。它的值是一些逻辑列名。-columns 的默认值是一个空列，它表示目录树会显示一个包含目录树条目的列，且条目的-values 属性被忽略了。如果给出了-columns 的逻辑列名，每个条目中列元素的-values 属性会按顺序和相应的逻辑列名关联在一起。如果-values 列比逻辑列中的元素更少，那么多余的列会被赋予一个空字符串；如果该列比逻辑列中的元素更多，那么超出的元素会被忽略。19.7.1 节中的示例定义了一个名为 Capital 的逻辑列，且每个条目有一个关于-values 属性的单元素列表，该列表提供了逻辑列中的数值。下列代码使用名为 country、capital 和 currency 的三个逻辑列定义了一个目录树组件，每列都带有-values 属性的三个对应值。

```
ttk::treeview .tree -columns {country capital currency}
.tree insert {} end -values {Canada Ottawa CAD}
.tree insert {} end -values {France Paris EUR}
.tree insert {} end -values {Germany Berlin EUR}
.tree insert {} end -values {Japan Tokyo JPY}
```

逻辑列显示的顺序不一定要和它们出现在-columns 里的顺序一样。您也可以配置目录树，使之只显示逻辑列的一个子集。这个行为由目录树的-displaycolumns 属性控制。您可以按照它们应该显示的顺序将它设为一些逻辑列名(或者从 0 开始的索引)。-displaycolumns 的默认值是#all，它表示所有逻辑列都应该按照它们被定义的顺序来显示。

目录树条目总是出现在第一个显示列上。您不能在别的列上显示它们，但您可以选择不显示它们。目录树的-show 属性包含一列用来指定目录树中要显示的元素的值。它的默认值为{tree headings}，用来显示目录树条目和列标题。

column 命令能查询或修改显示列的配置。

```
treeview column column ?option? ?value option value...?
```

如果逻辑列名被显示出来，可以用它来指定一个 column；也可以用从 0 开始的逻辑列索引来标识-columns 属性中的列。另外，可以用一个格式为#*n* 的字符串来指明显示列，这里的 *n* 是一个以 0 开始从左往右数的列序号。

提示： 列#0 总会指向目录树列，即使并没有显示它。

可用的列选项如下所述。

- -id——包含逻辑列名的只读选项。
- -anchor——确定这一列中的文本相对单元格如何对齐；它的值为 n、ne、e、se、s、sw、w(默认)、nw 或者 center 中的一个。
- -minwidth——最小列宽，单位为像素；默认值为 20。
- -stretch——布尔值，在调整组件大小时，可用于指定是否需要调整列宽；默认值为 1(真)。
- -width——列宽，单位为像素；默认值为 200。

您可以用 heading 命令来查询或修改列标题。

```
treeview heading column ?option? ?value option value ... ?
```

可用的标题选项如下所述。

- -text——列标题中显示的文本。
- -image——列标题的右侧显示的图像。
- -anchor——指定标题文本应该如何对齐；它的值为 n、ne、e、se、s、sw、w、nw 或者 center(默认)中的一个。
- -command——在标题标签被按下时，所处理的一个脚本。

19.7.3 目录树条目选择管理

当用户单击条目时，默认的目录树组件类绑定可以自动地选择一个条目。但是，绑定的方式取决于目录树的-selectmode 选项值。

- extended (default)
  单击一个条目会选中它，并放弃对当前选中的任何其他条目的选定。用户可以选定多个条目，Control 加单击增加对条目的选择，Shift 加单击可选中连续区域内的条目。
- browse
  单击一个条目会选中它，并放弃对当前选中的任何其他条目的选定。用户最多只能选中一个条目。
- none
  单击条目将不会影响条目的选择。因此，它会使用户对条目的选择失效。

-selectmode 选项只影响用户控制目录树条目选择的能力。无论-selectmode 值为何，应用程序都可以用目录树的 selection 子命令来控制对条目的选择。

```
treeview selection ?selectOp itemList?
```

如果不给出其他参数，selection 子命令会返回一列选中的条目。selection set 子命令会使 *itemList* 成为新的选择；提供一个空的 *itemList* 将使所有的条目从选择中删除。selection add 子命令会向选择中添加 *itemList* 里的条目；selection remove 子命令会从选择中删除 *itemList* 里的条目；selection toggle 子命令会切换 *itemList* 中每个条目的选择状态。

目录树组件也会生成一个<<TreeviewSelect>>虚拟事件，不管一个条目有没有被选

中。关于虚拟事件和执行响应事件脚本的详细信息可参见第 22 章。

19.7.4 目录树条目标记

目录树组件允许您定义符号标记，这些标记可以用在任意多条目上。标记用于对目录树中的每个条目应用格式。您可以用 tag configure 子命令来定义、查询或修改标记。

```
treeview tag configure tag ?option? ?value option value ...?
```

可用的标记选项如下所述。

- -background——单元格显示条目时使用的背景颜色。
- -foreground——条目的文本前景颜色。
- -font——条目的文本字体。
- -image——条目要显示的图像，但只当该条目没有自己的-image 属性集时才可使用。

所有的标记选项都有一个默认值，即空字符串，这说明标记不会影响条目的属性。下面这个定义了名为 highlight 和 important 的两个标记的代码就是一个示例。

```
.tree tag configure highlight \
    -background blue -foreground white
.tree tag configure important -foreground red
```

要将标记用于条目，可以将标记名列表作为条目的-tags 属性值，例如：

```
.tree item vt -tags {highlight important}
```

标记是按照优先级从低到高来排序的。如果多个标记有相同的标记选项设置，最高优先级的标记的值会被使用。例如，因为在刚才的代码中，important 出现在 highlight 的后面，它的前景颜色 red 会用于 vt 条目。但因为 important 没有为自己的-background 参数设值，highlight 标记设定的 blue 值就会被使用。

目录树组件也允许您用 tag bind 子命令将绑定和标记关联起来。您可以使用绑定来“激活”条目，这样它们就可以响应鼠标、键盘或者虚拟事件。(关于虚拟事件和执行响应事件脚本的详细信息可参见第 22 章。)和其他组件相比，它使您可以实现超文本效果。例如，当用户单击含 important 标记的条目时，下述绑定将向控制台打印一条信息。

```
.tree tag bind important <ButtonPress-1> {
    puts "This is an important item"
}
```

只有当条目有焦点时，键盘事件才会触发一个标记绑定。在一个目录树中，最多同时允许一个条目有焦点。默认的目录树组件类绑定会自动为用户选中的最后一个条目赋予焦点。您可以使用 focus 子命令来查询或设定焦点条目。

```
treeview focus ?item?
```

如果该命令在调用时没有条目，它会返回当前焦点条目的标识符；如果条目都没有焦点，则会返回一个空字符串。如果该命令在调用时有条目，则它会为指定的 *item* 赋予焦点。

多个绑定可能会和一个特殊的事件匹配，例如，如果为多个标记创建了绑定，然后在

一个条目上应用这些标记的组合。此时，所有匹配的绑定会被调用，且按照标记的优先级从低到高排列。如果单个标记有多个匹配的绑定，则只有指定的绑定才会被调用。绑定脚本的 continue 命令会终止该脚本，break 命令在终止该脚本的同时，也会跳过事件的任何其他脚本，正如针对 bind 命令那样。

提示： 如果使用 bind 命令创建了目录树组件的绑定，则可以使用 tag bind 子命令加上标记绑定来调用这些绑定。标记绑定会在任何组件整体绑定之前被调用。

19.8 主题组件状态

组件的状态能控制它的外观和行为。经典组件的状态由-state 选项值决定，正如 18.15.1 节所述。大多数经典组件只支持 normal 值和 disabled 值，但一些组件，如输入框和调节框，还支持其他状态，如 readonly。经典组件在同一时间只能处于一种状态。

主题组件也使用状态，但使用方法和经典组件有两点重要的区别。第一，状态包含了一些独立的标志，它们都同时开或关；因此，组件的整体状态和开或关的标志的特定组合对应。第二，所有主题组件都支持状态，所有组件都采用完全一样的状态标志集，虽然我们将会发现，一些标志可能被特定的组件忽略。所有主题组件现在都支持下述状态。

- active
  鼠标光标位于组件上方，此时按下鼠标按钮会触发一些动作。
- disabled
  组件在程序控制下失效。
- focus
  组件含有键盘焦点。
- pressed
  组件被按住。
- selected
  组件为“开”、“真”或者“当前”，用于诸如复选按钮和单选按钮之类的组件。
- background
  Windows 和 Mas OS X 有被激活的窗口(或称前景窗口)这一概念。用于设定背景窗口中组件的 background 状态，并删除前景窗口组件的状态。
- readonly
  组件不允许用户修改。
- alternate
  这是一个针对组件的可替代显示格式。例如，它可用于处于“三态”或“混合态”的复选按钮和单选按钮，也可用于-default 选项为 active 的按钮。
- invalid
  组件值无效(例如，输入框组件值未能通过验证)。

对应用程序中每个主题组件而言，一些状态可以为“开”或者“设置”，而另外一些可以为“关”或者“取消设置”。每个主题组件类的默认绑定能自动管理状态。例如，TButton 类(主题按钮)有自己默认的绑定，可以检查主题按钮的 disabled 状态是否为“关”；如果是，则鼠标进入该按钮会使 active 状态处于“开”，且当鼠标离开按钮时，会退回“关”的状态。您可以用 state 组件命令，使程序修改和查询该主题组件的状态。

```
widget state ?stateSpec?
```

stateSpec 是要设置或取消设置的状态的列表。如果一个元素由状态名组成，则该状态会被设置；如果一个元素由以！开头的状态名组成，则该状态被取消。返回值显示了哪个标志被改变。如果未用 *stateSpec* 来执行 state 组件命令，则它会返回当前设置的状态值。例如：

```
    .e state {focus readonly}
⇒ !focus !readonly
    .e state
⇒ focus readonly
    .e state {!focus !readonly disabled}
⇒ !disabled focus readonly
    .e state
⇒ disabled
```

您可以使用 instate 组件命令来测试主题组件的状态。

```
widget instate stateSpec ?script?
```

如果没有 *script* 参数，则命令会在组件状态和 *stateSpec* 匹配时返回 1，否则返回 0。例如：

```
    .e instate disabled
⇒ 1
    .e instate {disabled !readonly}
⇒ 1
    .e instate alternate
⇒ 0
```

如果提供了一个脚本参数，且组件状态与设置的状态匹配，则脚本会被执行。例如，当且仅当主题按钮.b 在当前被按下且有效时，下列代码会调用该按钮。

```
.b instate {pressed !disabled} { .b invoke }
```

提示： instate 组件命令允许您创建组件甚至是类绑定，这样就可以很方便地执行基于状态的行为。在 19.9.3 节中您会看到，ttk::style map 命令也允许您改变组件在特定状态下的外观。当两者共同作用时，可以在不改变实现组件的底层代码的情况下，对既有组件设置进行重要的调整。

19.9 主题组件样式

每一个主题组件都被赋予一种样式，它指定了不同环境中组件的外观。记住每一个组件都有一个关联的默认类(例如，对所有标签组件而言是 TLabel)。与主题组件关联的默认

样式和组件类有相同的名字(例如，一种叫 TLabel 的样式)。因此，所有同一类的主题组件都有相同的外观，除非应用程序明确为一个组件设好了样式。

您可以使用 winfo class 命令来决定特殊组件的类，例如：

```
ttk::label .l
winfo class .l
⇒ TLabel
```

您可以将样式名设为组件的-style 属性的值，以此来覆盖主题组件的默认样式。(默认值是一个空字符串，它表示组件使用由类名定义的默认样式。)例如，如果已经定义了一个叫 AlertLabel.TLabel 的新样式，可以将它赋给一个组件。

```
.l configure -style AlertLabel .TLabel
```

您会在后续几节中发现如何定义新的样式。

19.9.1 使用主题

主题是一个命名的样式集合，用来为应用程序中的所有主题组件提供一致的外观。ttk::style theme names 命令列出了当前的系统中被定义的主题名。

```
ttk::style theme names
⇒ aqua clam alt default classic
```

当您执行应用程序时，Tk 会自动为窗口系统选择一个合适的主题。ttk::currentTheme 变量包含了当前的主题名。

```
puts $ttk::currentTheme
⇒ aqua
```

您可以通过执行 ttk::style theme use 来使用别的主题。

```
ttk::style theme use clam
```

这个命令使用新主题中的定义来更新所有的样式，并刷新所有的主题组件来反映新的主题。

19.9.2 样式的元素

主题组件在内部作为元素的集合执行。样式的一个功能就是控制元素的位置。

提示： 一般情况下，应用程序开发者很少会改变组件的布局。但是，当您想自定义它的配置时，可以指定组成特殊主题组件的元素，如后文所述。

您可以使用 ttk::style layout 命令来查看或修改一个指定样式的元素的布局。以下是 Aqua 主题下的示例。

```
ttk::style layout TCheckbutton
⇒ Checkbutton.button -sticky nswe -children {Checkbutton.padding
  -sticky nswe -children {Checkbutton.label -side left -sticky {}}}
```

布局由一系列元素构成，每个元素之后都会有一个或多个选项，来指定如何布置这些元素。在上面的返回值中，Checkbutton.button、Checkbutton.padding 和 Checkbutton.label 是在 Aqua 主题下组成一个主题复选按钮组件的元素。重新设定输出的格式可以使结构更清晰。

```
checkbutton.button -sticky nswe -children {
    checkbutton.padding -sticky nswe -children {
        checkbutton.label -side left -sticky {}
    }
}
```

元素在组件所分配的空间中按列出的顺序来布置，这种空间被称为空腔。可选的-side 属性决定了元素中所分配的空间的包的位置，它可设为 left、right、top 或 bottom 中的一个。如果忽略-side 属性，则整个空腔会分配给元素。-sticky 选项显示了元素会“粘贴”到的包的边，以此指定元素在包中的位置。例如，上一个示例中的-sticky nswe 选项说明这个元素将填满整个包。-children 属性指定了在父元素中布置的一个或多个元素，它们是按照上面讨论过的规则来布置的。

注意，特殊样式的布局可能因主题而不同。这是因为一些窗口系统使用组件的元素(如焦点指示器)，而该元素没有被其他窗口系统使用。例如，在 Aqua 主题下，TButton 布局由下面的代码定义。

```
Button.button -sticky nswe -children {
    Button.padding -sticky nswe -children {
        Button.label -sticky nswe
    }
}
```

与之相比，在 XP 的原生主题下的 TButton 布局是如下定义的。

```
Button.button -sticky nswe -children {
    Button.focus -sticky nswe -children {
        Button.padding -sticky nswe -children {
            Button.label -sticky nswe
        }
    }
}
```

19.9.3 创建和配置样式

作为应用程序开发者，您或许想要改变程序中不同组件的外观，例如，设置前景颜色，来显示一些按钮中的文本。这时，您不需要为每个单独的组件设置配置选项，而只需要为这个样式设定配置选项即可，这样的配置会自动影响每个使用该样式的组件。

在配置与样式关联的选项前，需要知道该样式的哪些选项是可用的。那些作为主题组件一部分的元素都有一些它们能用的选项。您可以用 ttk::style element options 来获取一些与某个指定元素关联的选项，例如：

```
    ttk::style element options TLabel.label
⇒   -compound -space -text -font -foreground -underline -width -anchor
    -justify -wraplength -embossed -image -stipple -background
```

包含该元素的样式可以提供该元素选项的实际值，或者在一些情况下(如-text)由主题组件的实际实例提供。在配置一种样式时，可以为任一与任意样式元素关联的选项提供值。单个选项，如-background，可能与样式中的多个元素关联。此时，样式提供的值可能被所有关联元素使用。这种特性保证了组件的外观一致。

ttk::style configure 命令可以查询或修改某种样式的选项值。

```
ttk::style configure style ?option? ?value option value ... ?
```

例如，您可以配置 TLabel 样式，这样程序中的每个主题按钮都可以将文本显示为绿色。

```
ttk::style configure TLabel -foreground green
```

但您可能更希望自定义配置，只将它应用于应用程序中一个特殊类的某些组件。例如，您可能希望大多数标签使用主题定义的标准颜色，但另外一些标签是特别警示，应该使用红色的文本。此时，可以定义一个从 TLabel 样式演化来的新样式，如下所示：

```
ttk::style configure AlertLabel.TLabel -foreground red
```

样式名 AlertLabel.TLabel 显示了它是从 TLabel 中衍生出来的。如果元素查询了 -foreground 选项，那它会得到值 red；如果它查询了其他选项，那它将得到在 TLabel 基本样式中配置的任意值。您可以创建任意深的衍生样式层级结构，使用“.”来隔开每一层。当元素查询某个选项时，它会得到第一种样式的值，而该元素正是在该样式中被定义的。同时，也存在一个名为“.”的基本样式，用来为所有样式提供默认选项值。“.”样式主要用作一个基本的构造模块，其作用是定义主题的外观。

使用 ttk::style configure 命令来设置的选项值是应用的默认值。但是，也可以使用 ttk::style map 命令来定义一个新值，这取决于组件的状态。

```
ttk::style map style ?option? ?{stateSpec value ...} ...?
```

style 和 *option* 参数和 ttk::style configure 一起被解释。下一个参数是指定状态的列表(用 19.8 节中介绍过的格式来表示)和对应的选项值。特殊组件的实际状态和指定状态按照它们列出来的顺序被比较。第一个和组件状态一致的指定状态决定了选项使用的值。状态映射也被继承下来，因此如果在当前样式中没有选项的合适映射，则衍生它的样式会被检查，以此类推。如果任一指定状态都不匹配，选项将得到由前面介绍过的 ttk::style configure 定义的默认值。

ttk::style lookup 命令用于检索一种样式的特殊选项值。

```
ttk::style lookup style option ?stateSpec ?default??
```

如果没有 *statetSpec* 和 *default* 参数，则命令会返回默认值，正如由 ttk::style configure 定义的那样。例如，要检索默认的按钮字体：

```
    ttk::style lookup TButton -font
⇒ TkDefaultFont
```

如果给定了 *stateSpec*，命令会使用元素选项的标准查找规则返回一个值(这就是说，首先会检查与 ttk::style map 指定状态匹配的状态，如果没有，则使用由 ttk::style configure 定义的默认值)。如果给出了 *default* 参数，它会作为一个应急值，以免该选项没有匹配的指

定状态。

19.10 其他标准主题组件选项

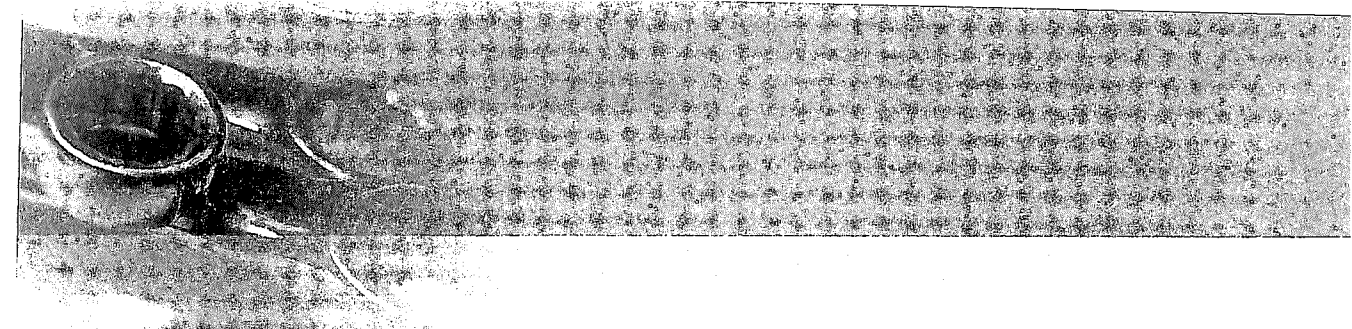
像经典组件一样，主题组件支持控制它们的外观和行为的选项。当组件用 configure 组件命令来创建和修改时，这些主题组件的配置选项可以被指定；您可以使用 cget 组件命令来查询组件的配置选项值。

主题组件支持的标准选项集远远小于经典组件集。和经典组件不同的是，大多数主题组件不支持诸如-background、-relief 或者-font 等用来控制颜色、边缘或者字体的选项。相反，这些外观属性由组件样式控制，该样式通常是由应用程序中整个被选中的主题来决定的。19.9 节有关于样式的更多讨论。

主题组件支持-cursor 选项，当鼠标置于组件之上时，该选项能控制鼠标光标的外观，正如在经典组件下的作用那样，18.15.5 节对此有过讨论。可滚动的主题组件也支持-xscrollcommand 和-yscrollcommand 选项，且遵从 18.9 节介绍过的滚动规则。

标签、按钮和其他类似按钮的主题组件支持-text 和-textvariable 选项，它们的行为与在经典组件中相似。这些组件同时也支持-image 选项，用来指定显示的图像对象名。但是，主题组件并不局限于单个的图像名，它允许您使用-image 选项指定一列数值。第一个元素是显示的默认图像名。下层元素是一个两元素嵌套列表，其中第一个元素是一个指定状态，正如 19.8 节中定义的那样，第二个元素是该状态下显示的图像名。第一个与组件当前状态匹配的指定状态决定了显示的图像。所有图像都应该有相同的尺寸。

如果组件的图像和文本都被指定了，那么-compound 选项能决定如何相对于文本给定图像的位置。如果选项值为默认值 None，在存在图像时会显示该图像，否则显示文本。选项值为 text 或 image 只显示其中一种类型的内容时。选项值为 top、bottom、left 或 right，将图像相对于文本放在指定的位置。选项值为 center，则将文本显示在图像上方的中间。



第20章 字体、位图和图像

任何一个图形界面的首要功能是显示文本和图片，而要显示它们就需要字体和图像。不同平台下的可用字体很不相同，即使在不同 Windows PC 机之间也不同。如果能在应用程序中控制字体的应用，则会简化对各种平台和环境的支持。Tk 提供了 font 命令来创建命名的字体并在应用程序中控制这些字体的使用。类似地，在现代的应用程序中，图标变得和任何字母的字形一样重要。这些图像(一些是公用的，另外一些有特定用途)在任何一个带有图形用户界面的应用程序中都是需要的。Tk 的 image 命令用于载入、控制和分享应用程序里的图像数据。本章我们将介绍这两条命令。

20.1 本章出现的命令

本章将讨论下述用来控制字体和图像的命令。

- `font actual` *`font`* `?-displayof` *`window`*`? ?`*`option`*`? ?--? ?`*`char`*`?`
  返回字体的实际属性值。实际值可以因为平台的限制而与指定的值不同。这些选项与 font configure 支持的相同。
- `font configure` *`fontname`* `?`*`option`*`? ?`*`value option value ...`*`?`
  查询或配置一个命名字体。如果单个 *option* 在没有 *value* 时被指定，它会返回该属性的当前值。如果一个或更多的 *option value* 对被指定了，则命令会修改指定的命名字体，使其具有指定的值。关于被支持的属性和它们的值的介绍可参见本书和参考文档。
- `font create ?`*`fontname`*`? ?`*`option value ...`*`?`
  创建一个带有由 *option value* 对所指定属性的一个新的命名 *fontname*，并返回它的名称。如果 *fontname* 被忽略了，Tk 会生成一个新的名称。选项和 font configure 支持的选项是相同的。
- `font delete` *`fontname`* `?`*`fontname ...`*`?`
  删除给定的命名字体。
- `font families ?-displayof` *`window`*`?`
  返回 *window* 显示上的所有字体名称，不区分大小写。
- `font measure` *`font`* `?-displayof` *`window`*`?` *`text`*
  测量显示在 *window* 上采用给定字体的字符串 *text* 所用的空间。返回值是 *text* 的总宽度，单位为像素，并不包括特别夸大的字符所用的多余像素，例如花体的 f。

诸如换行符和制表符之类的格式控制字符会被忽略。

- `font metrics` *font* `?-displayof` *window*`? ?`*option*`?`
  在 *window* 显示上使用时，会返回 *font* 的规格信息。更多信息可参见参考文档。
- `font` *names*
  返回当前定义的所有命名字体的列表。
- `image create` *type* `?`*name*`? ?`*option value ...*`?`
  创建新的 *type* bitmap 或者 photo 图像。如果 *name* 被忽略了，会自动创建一个新的名字。返回所创建的图像的名字。同时也为进一步的图像操作创建相同名称的命令。所支持选项的更多信息可参见本书和参考文档。
- `image delete ?`*name ...* `?`
  删除每一幅命名图像。每一个显示的图像实例的大小将保持，但其区域将变为空白。直到所有的实例被释放，图像才会被真正删除。
- `image height` *name*
  返回图像的高度，单位为像素。
- `image inuse` *name*
  返回一个布尔运算值，以说明图像是否正被任何组件使用。
- `image names`
  返回一个包含所有已有图像名的列表。
- `image type` *name*
  返回图像 *name* 的类型，如 bitmap 或者 photo。
- `image types`
  返回一列数，它的元素是 image create 命令中类型参数的全部有效值。
- `image width` *name*
  返回图像的宽度，单位为像素。

20.2　font 命令

Mac OS X 和 Windows 使用相同的命名法来指定字体，而以 X 为基础的窗口系统使用另外一套命名方式。在 X 里面，字体名指定了许多参数，例如：

```
-adobe-times-bold-r-normal--18-180-75-75-p-99-iso8859-1
```

这个字体名指定了 Adobe 里的 Times Bold 字体，大小为 18 磅，是为 75 DPI 屏幕设计的，且支持 Western European ISO-8859-1 字符集。Mac OS X 和 Windows 都采用短得多的名字来指定字体，如下所示：

```
Times Bold 18
```

Tk 使用命名字体来缩小这种差异。Tk 组件接受以 5 种不同格式指定的字体，但是，使用 Tk 的最佳方法是使用命名字体。这样可以方便地针对不同平台设置。

提示：如果为组件设置的主字体不包括想要显示的特定 Unicode 字符的字形，Tk 会试图提供一个包含该字形的字体。如果可能，Tk 会试图采用一种与尽可能多的组件主字体匹配的字体(例如，粗细、倾斜等)。当 Tk 发现一个合适的字体时，它会以这种字体来显示字符。换言之，组件使用主字体来显示所有它能显示的字符，只有当需要的时候，才会换成别的字体。在一些情况下，Tk 不能识别合适的字体，这时，组件便不会显示出该字符。作为替代，组件会显示出一个依赖于系统的应急字符，例如？。

Tk 在所有与相应系统默认值匹配的平台上都有一个预定义的命名字体集。您不应该改变这些字体，Tk 自身可以调整字体，来响应系统的改变。字体名和它们默认的用法如下所述。

- TKDefaultFont——所有 GUI 条目的默认值，除非特别指定。
- TKTextFont——在输入框组件、列表框等中的用户文本使用的字体。
- TkFixedFont——标准的固定宽度的字体。
- TkMenuFont——菜单条目所使用的字体。
- TkHeadingFont——在列表和表格的列标题中使用的字体。
- TkCaptionFont——在窗口和对话框标题栏中使用的字体。
- TkSmallCaptionFont——内置窗口或工具对话框的标题的字体。
- TkIconFont——图标标题的字体。
- TkTooltipFont——工具栏窗口(临时信息窗口)的字体。

20.2.1 控制和使用命名字体

要使用命名字体，首先要创建新字体，给出想使用的名字。在创建字体时，可以指定字体、尺寸等，这时需要使用 font create 命令。

```
font create ?fontname? ?option value ...?
```

您可以为命名字体提供一个特定的 *fontname*，或者可以使用 font create 自动赋予一个含有 font 的名字，后跟一个唯一的整数。返回值是创建的字体名。其后的 *option value* 对参数指定了字体的属性。您可以使用下列选项中的任何一个来配置字体。

- `-family` *name*
  不区分大小写的字体 *name*。Tk 保证支持 Courier、Times 和 Helvetica 字体。当它们中的一个被使用时，最匹配的原生字体会自动被取代。名称也可以是一个原生的、针对平台的字体名。如果该字体未被指定，或者未被识别，一个针对平台的默认字体会被选中。
- `-size` *size*
  字体的指定尺寸。正的尺寸参数会被解释为一个尺寸，单位为磅。负的尺寸参数会被解释为一个尺寸，单位为像素。如果字体不能用指定的尺寸显示出来，那么将会使用邻近的尺寸。如果 *size* 未被指定或为 0，一个依赖于平台的默认值会被使用。

- `-weight` *`weight`*
  字体中字符的很小的厚度。有效值为 normal，用来显示正常粗细的字体，而 bold 会显示粗体字。
- `-slant` *`slant`*
  字体中字符偏离垂直方向的大小。有效的 *slant* 为 roman 和 italic。
- `-underline` *`boolean`*
  值为一个布尔运算值，以指定使用该字体的字符是否应该被加上下划线。
- `-overstrike` *`boolean`*
  值为一个布尔运算值，以指定是否应该在使用该字体的字符中间画上一根水平线。

例如，下列代码会基于 Helvetica 字体创建一个名为 LobsterBold 的字体。

```
font create LobsterBold -family Helvetica \
    -size 12 -slant roman -weight bold
```

在创建命名字体后，可以将其用在任何 Tk 接受字体的地方，例如特殊组件的-font 属性，或文本组件中标记的-font 属性。

```
.mylabel configure -font LobsterBold
```

在定义命名字体后，可以稍后使用 font configure 命令来修改它的定义。

```
font configure fontname ?option? ?value option value ... ?
```

您可以给指定选项提供值，来修改命名字体的任意属性。被支持的选项和 font create 的相同。如果提供一个未带有新值的选项名，font configure 命令将返回该属性的当前值。如果只将字体名称作为一个参数，则命令会返回一个包含所有字体属性及其值的字典。

当使用 font configure 来改变命名字体的定义时，应用程序中采用命名字体的所有组件会即时更新来反映这种变化。因此，下面的命令会使所有使用 LobsterBold 字体的组件将显示的文本尺寸减小到 10 磅。

```
font configure LobsterBold -size 10
```

font names 命令返回所有的命名字体。font actual 命令用于确定命名字体的实际属性。例如，需要的字体或其他字体属性在本地系统中不可用，这将会使字体与要求的值不同。下面的示例演示了在 Windows 操作系统中 Arial 字体是如何被由 LobsterBold 定义的 Helvetica 替换的。

```
⇒ font actual LobsterBold
  -family Arial -size 10 -weight bold -slant roman -underline 0
  -overstrike 0
```

font delete 命令删除一个或多个命名字体。如果一些正在使用命名字体的组件已被删除，命名组件不会真正被删除，直到所有实例被释放。那些组件会继续使用最后一个已知的命名字体值。如果新的命名字体使用相同的名称来创建，而此时一些组件仍在使用以前的命名字体，则组件会刷新，并使用字体的新属性。

下面的脚本显示了如何在 Tk 中创建命名字体，并改变字体尺寸。

```
#
# Demonstrate named fonts
#

set top [toplevel .dialog]
set icon [label $top.icon -bitmap info]
set l [label $top.msg -textvariable message]
set message "Named Font Demonstration"
set defaultFont [$l cget -font]
set f [frame $top.f]

set bDefault [button $f.def -text A -command \
    "configDefault" -font {helvetica}]
set bSmaller [button $f.sm -text A -command \
    "configSize -2" -font {helvetica 10}]
set bBigger [button $f.bg -text A -command \
    "configSize +2" -font {helvetica 18}]

pack $bDefault $bSmaller $bBigger -side left -pady 5 -padx 5
grid $icon  -row 0 -column 0 -padx 5 -pady 10
grid $l     -row 0 -column 1 -sticky ew -padx 5 -pady 10
grid $f   - -row 1

set defaultSize     [font actual $defaultFont -size]
set defaultFamily   [font actual $defaultFont -family]
set defaultWeight   [font actual $defaultFont -weight]

set size 18
set family Helvetica
set weight bold
font create LobsterBold -size $size -family $family \
     -weight bold
$l configure -font LobsterBold

proc configDefault {} {
    font configure LobsterBold \
        -family $::defaultFamily \
        -size $::defaultSize \
        -weight $::defaultWeight
    set ::message "Default Font"
}

proc configSize {delta} {
    font configure LobsterBold \
        -family Helvetica \
        -size [expr {[font actual LobsterBold -size ] + $delta}]
    if {$delta < 0} {
        set ::message "Smaller Font"
    } else {
        set ::message "Bigger Font"
    }
    puts "font size is [font actual LobsterBold -size]"
}
```

图 20.1 显示了运行这个脚本的结果，三幅不同的屏幕截图反映出按下按钮后改变字体配置的结果。

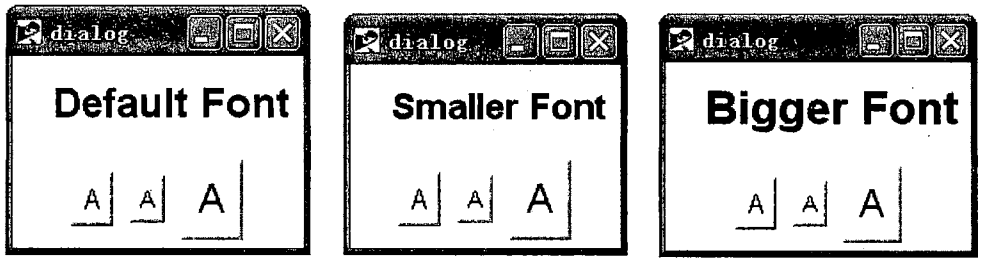


图 20.1　改变命名字体配置的示例

20.2.2　其他的字体应用

除了控制命名字体，font 命令还提供了可查询字体信息的其他子命令。

font families 命令能返回所有有效的字体。对于多个显示器的系统来说，不同的字体集可以在不同的显示器上使用，所以命令会支持-displayof 选项，允许您提供一个组件名来检索显示组件的显示器上的有效字体。如果没有提供-displayof 选项，命令会默认使用主显示器。font families 命令用于创建字体选择器菜单或者对话框，来允许用户为应用程序选择字体。

font measure 命令会返回采用指定字体的字符串的宽度，单位为像素。

```
font measure times "This is a text"
⇒ 67
```

20.2.3　字体描述

如前所述，Tk 中的字体有 5 种可能的格式。任意一种都可以在指定字体的地方使用，例如窗口属性、文本或画布组件标记属性或 font 命令中。下面将介绍这 5 种格式。

- *fontname*
  命名字体的名称，由 font create 命令创建。命名字体在使用时总是没有误差的。如果命名字体不能完全按照指定属性来显示，那么另外一些接近的字体会自动替代它们。
- *systemfont*
  字体针对平台的名字，由窗口系统解释。这也包括了 X(参见后文)下的 XLFD。Tk 提供了一系列预定义的命名字体，它们被映射为系统字体。更多的细节可参见参考文档。
- *family ?size? ?style? ?style ...?*
  一个格式正确的列表，其中第一个元素是字体，可选的第二个元素是尺寸。*size* 属性遵从与 font configure 的-size 参数相同的规则。任何位于尺寸之后的多余参数是字体样式。*style* 的可能值为 normal、bold、roman、italic、underline 和 overstrike。
- X-字体的名字(XLFD)
  以 Unix 为中心的字体名，格式如下：

```
-foundry-family-weight-slant-setwidth-addstyle-pixel-point
-resx-resy-spacing-width-charset-encoding
```

“*”字符用于为一个域赋予默认值。每个默认域都必须恰好有一个“*”，如果“*”位于 XLFD 末尾，表示为所有余下的域赋予默认值；最短的有效 XLFD 仅有一个“*”，它为所有域赋予默认值。

- *option value?option value ...?*
  一个格式正确的 *option value* 对列表，用来指定字体的属性，和 font configure 中的格式相同。

当一个字体被指定时，Tk 会按顺序使用上述 5 种规则来解析字体，并识别出该字体。对前两种情况而言，名字必须精确匹配。对剩下的三种情况，将使用最接近的有效字体，如果不行，将会选择一个根据系统而定的默认值。如果字体的描述不与这些模式的任何一种匹配，将会生成误差。

20.3 image 命令

Tk 的 image 命令可以创建和控制图像。它创建一个图像对象，该对象在稍后会用 -image 或者-bitmap 选项传给组件，或者由对象的子命令来管理。当一幅图像被创建时，一个相同名称的新命令也会被创建。该命令可以管理图像中的像素数据。

提示： *在默认的情况下，image 命令在全局命名空间中创建。新创建的 image 命令也覆盖了任何具有相同名字的已有命令。注意在选择图像名时，不要覆盖已有的命令，或者让 image create 为您创建图像名。一种控制图像名的方法是给图像使用完全符合规范的命名空间名称，如 20.3.3 节所述。*

图像有两种嵌入类型：bitmap，一种双色图像；photo，一种多色图像。新类型可以作为一种扩展由 Tk_CreateImageType C API 定义。例如，Img 扩展定义 X11 像素图的 pixmap 类型。photo 图像类型也可以由 Tk_CreatePhotoImageFormat C API 扩展，它能提供对不同数据格式的支持。Img 扩展也能起到这样的作用，除了那些嵌入 Tk 的格式外，还能支持很多格式。

位图可由一个简单的位平面定义，设定开/关，或者带有掩码的前景/背景颜色。掩码定义了图像显示的范围，而该范围外的部分是透明的，下面的组件能透过这部分显示。Tk 中有一些地方只能使用位图，例如，wm iconbitmap 命令。这条命令定义了当窗口最小化时能在窗口系统中使用的图标。(一些窗口系统此时只允许出现简单的位图。)

图像的第二种类型是相片。相片是内部全色彩的(每个像素有 32 位)图像，在需要时，它会采用递色处理来显示。Tk 对由 GIF 和 PPM/PGM 文件格式创建的图像提供了内建支持，且它的扩展还能添加对其他格式的支持。后文会更详细地讨论两种内建类型的图像。

20.3.1　位图图像

您可以用参数为 bitmap 类型的 image create 命令来创建位图。

```
image create bitmap ?name? ?option value ...?
```

您可以为位图提供一个特定的 *name*，或者让 image create 自动赋予一个含有 image 的名称，image 之后会有一个唯一的整数。返回值是所创建位图的名称。后面的 *option value* 对参数指定了位图的属性。您可以使用以下选项中的任意一个来配置位图。

- `-background color`
  为图像指定一种背景颜色。如果该选项被设定为空字符串，则背景像素将是透明的。将源位图设为掩码位图，且忽略任何-maskdata 或-maskfile 选项，能达到这种效果。
- `-data string`
  将源位图的内容指定为 X11 位图格式的字符串。如果-data 和-file 选项都被指定了，-data 选项会优先于-file 选项。
- `-file fileName`
  给出一个文件名，该文件的内容定义了采用 X11 位图格式的源位图。
- `-foreground color`
  指定图像的前景颜色。
- `-maskdata string`
  将掩码的内容指定为 X11 位图格式的字符串。如果-maskdata 和-maskfile 选项都被指定，则-maskdata 选项会优先于-maskfile 选项。
- `-maskfile fileName`
  给出一个文件名，该文件的内容定义了采用 X11 位图格式的掩码。

对位图类型而言，image create 返回的图像命令支持两个子命令：cget 和 configure。cget 子命令会返回指定选项的当前值。configure 子命令可以查询或修改指定选项，或者在未给定参数时返回所有选项和它们的值。位图可以在任何支持-bitmap 选项的组件中使用。

20.3.2　相片图像

相片是全色的图像。image create photo 命令会创建一个图像对象和一个控制该图像的 Tcl 命令。image create 命令接受相片类型的下述选项。

- `-data string`
  将图像内容指定为一个字符串。该字符串将包含基本 64 位加密数据或者二进制数据。字符串的格式必须是这样的：存在一个能接受字符串数据的图像文件格式处理器。
- `-format formatName`
  为使用-data 或-file 选项的数据指定文件格式名。

- `-file` *fileName*
  给出了一个包含相片图像数据的文件名。文件格式必须是这样的：存在一个能读取数据的图像文件格式处理器。
- `-gamma` *value*
  指定用在图像的色图上的 gamma 校正。指定值必须大于 0。默认值为 1(没有校正)。通常，大于 1 的值会使图像变亮，小于 1 的值会使图像变暗。
- `-height` *number*
  指定图像高度，单位为像素。默认值为 0，它允许图像在垂直方向扩展或收缩，以恰好容纳存储在其中的数据。
- `-palette` *paletteSpec*
  指定图像所使用的调色板。对单色图像，*paletteSpec* 字符串可以是单个十进制数，表示灰度。而对彩色图像，*paletteSpec* 可以是三个十进制数，由斜杠(/)隔开，分别表示红色、绿色和蓝色的色度。
- `-width` *number*
  指定图像宽度，单位为像素。默认值为 0，它允许图像在水平方向扩展或收缩，以恰好容纳存储在其中的数据。

返回的图像命令支持很多子命令。对于那些向图像中写入数据的命令，图像尺寸可以按需要调整，除非-width 和-height 选项已被配置为一个非 0 的值。一些子命令还接受其他选项。一些选项会在下面的示例中讨论；要全面了解相关内容，可以阅读参考文档。

- *imageName* `blank`
  删除图像，使它完全透明。
- *imageName* `cget` *option*
  返回用 *option* 指定的配置选项当前值。
- *imageName* `configure ?`*option*`? ?`*value option value ...* `?`
  查询或修改图像的配置选项。
- *imageName* `copy` *sourceImage* `?`*option value ...*`?`
  从 *sourceImage* 复制一个区域到 *imageName*。您可以选择指定源和/或目标图像的长方形子区域，还可以选择对它们进行修剪、重采样或缩放，并指定源图像是替换目标图像还是遮住目标图像。
- *imageName* `data ?`*option value ...*`?`
  以字符串格式返回图像数据。您可以选择指定要返回图像的一个长方形子区域，数据格式，透明像素是否被一种颜色替代，以及是否要将数据转为灰度。
- *imageName* `get` *x y*
  返回图像上坐标为 *x* 和 *y* 的像素的颜色，其值为 0～255 之间的三个数字，分别代表红色、绿色和蓝色成分。
- *imageName* `put` *data* `?`*option value ...*`?`
  将 *imageName* 中的像素设为 *data* 里指定的数据。您可以选择为目标数据和数据字符串的图像格式指定一个长方形子区域。数据也可被指定为一个 Tcl 扫描线

列，每一根扫描线由一些像素颜色组成。

- *imageName* read *fileName* ?*option value ...* ?
  从名为 *fileName* 的文件中读取图像数据，传给图像。您可以选择指定源和/或目标图像的长方形子区域，并指定 *imageName* 是否要被修剪以适合读取的数据。
- *imageName* redither
  重新计算显示图像的每个窗口中的递色图像。
- *imageName* transparency get *x y*
  返回一个显示在 *x*，*y* 点的像素是否为透明的布尔运算值。
- *imageName* transparency set *x y boolean*
  如果 *boolean* 为真则将 *x*，*y* 点的像素设为透明，否则设为不透明的。
- *imageName* write *fileName* ?*option value ...* ?
  从 *imageName* 中读取图像数据，写入名为 *fileName* 的文件。您可以选择指定要返回的图像的一个长方形子区域，数据格式，透明像素是否被一种颜色替代，以及是否要将数据转为灰度。

下面的示例演示了如何加载 Tcl Powered logo(Tk 数据库中一个简便的图像实例)，以及如何使用 image 命令来控制该图像。

第一个脚本会读取 Tcl Powered logo 图像并显示在一个标签上。结果如图 20.2 所示。

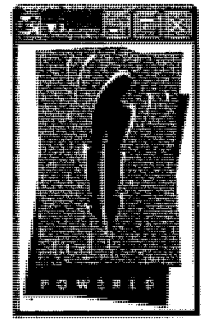


图 20.2　Tcl Powered logo 图像

```
# Fetch and display a source image
set img [image create photo \
    -file $tk_library/images/pwrdLogo200.gif]
label .img -image $img -anchor nw
grid .img -sticky nw
```

第二个脚本创建了一个新图像，其中的 Tcl Powered logo 被旋转了 90°。它在开始时创建了一个空图像对象。原始图像中的数据由 get 子命令逐个读取每个像素。返回值是一个三元素列表，其值分别表示红色、绿色和蓝色，单位为像素，值的范围为 0～255。该列表会转换为一个标准的#RRGGBB 颜色字符串，并用 put 子命令来向新图像写入合适像素。如要正确处理透明度，需测试原始像素的透明度设置，且在需要时将其复制到目标图像上。结果如图 20.3 所示。

```
#Rotate image 90 degrees clockwise
set img1 [image create photo]
toplevel .out1
label .out1.img -image $img1 -anchor nw \
    -height [image width $img]\
    -width [image height $img]
grid .out1.img -sticky nw

set wid [image width $img]
set hgt [image height $img]
for {set x 0} {$x < $wid} {incr x} {
```

```
    for {set y 0} {$y < $hgt} {incr y} {
        set color [$img get $x $y]
        set rgb [format {#%02x%02x%02x} {*}$color]
        $img1 put $rgb -to [expr {$hgt - $y}] $x
        if {[$img transparency get $x $y]} {
            $img1 transparency set [expr {$hgt - $y}] $x 1
        }
    }
}
```

第三个脚本创建了一个尺寸为原始尺寸一半的新图像。它通过创建一个新的空白图像对象，然后使用 copy 子命令从原始图像复制图像数据到新图像来实现。-subsample 选项接受另外两个参数，来指定复制应该在 *xy* 坐标系中每 *x* 个和每 *y* 个像素中抽取一个。通过设定它们的值为 2 和 2，则在 *x* 和 *y* 方向上每隔一个像素就抽取一个像素用于复制。结果如图 20.4 所示。虽然这样做很快，但它会产生浓淡不均的图像。要创建更美观的图像，通常可以平均像素值，就像下面这个代码。

```
# Create half size image by subsampling
set img2 [image create photo]
toplevel .out2
label .out2.img -image $img2 -anchor nw
grid .out2.img -sticky nw
$img2 copy $img -subsample 2 2
```

最后一个脚本也创建了一个新的一半尺寸的图像。在这种情况下，它通过 get 子命令来读取四像素块，平均这些颜色值，并将结果写入目标图像。如果像素中的三个或四个是透明的，则结果就会是透明的。结果如图 20.5 所示。这种平均技术的结果通常比重取样更美观，但它会消耗长得多的时间。

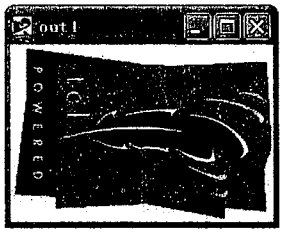


图 20.3 顺时针旋转了 90° 的图像

图 20.4 使用重取样复制将尺寸缩小一半后的图像

图 20.5 使用像素平均技术将尺寸缩小一半后的图像

```
# Create half-size image
# by averaging 4-pixel blocks
set img3 [image create photo]
toplevel .out3
label .out3.img -image $img3 -anchor nw
grid .out3.img -sticky nw

set wid [image width $img]
set hgt [image height $img]
for {set y 0} {$y < ($hgt - 1)} {incr y 2} {
        set dy [expr {$y / 2}]

    for {set x 0} {$x <($wid - 1)} {incr x 2} {
```

```
            set r 0
            set g 0
            set b 0
            set n 0
            for {set i 0} {$i < 2} {incr i} {
                set xi [expr {$x+$i}]
                for {set j 0} {$j < 2} {incr j} {
                    set yj [expr {$y+$j}]
                    if {[$img transparency get $xi $yj]} {
                        continue
                    }
                    incr n
                    set cx [$img get $xi $yj]
                    incr r [lindex $cx 0]
                    incr g [lindex $cx 1]
                    incr b [lindex $cx 2]
                }
            }
            set dx [expr {$x / 2 }]
            if {$n < 2} {
               $img3 put "#000000" -to $dx $dy
               $img3 transparency set $dx $dy 1
            } else {
              set r [expr {$r / $n}]
              set g [expr {$g / $n}]
              set b [expr {$b / $n}]
              set color [format "#%02x%02x%02x" $r $g $b]
              $img3 put $color -to $dx $dy
            }
        }
    }
```

对 image 命令，Tk 有一个 C 语言的接口，它用来提供对其他图像文件格式的支持。Img 扩展包恰好可以这样做，它提供对 BMP、ICO、JPEG、PCX、PNG、PPM、PS、SGI、SUN、TGA、TIFF、XBM 和 XPM 文件类型的支持。这种扩展也有一个选项来将任意 Tk 窗口捕获为一张图像。

另一种扩展是 TkMagick/TclMagick 包，这个包支持大约 100 种类型。它也提供扩展的图像操控能力，这种能力在没有 Tk 时也可以在 Tcl 中使用。这促进了在服务端应用程序中对图像的控制。

本节中的示例简单地介绍了 image 命令是如何工作的。像图 20.5 中那样的转换如果使用扩展，如 TclMagick，会更有效率。基本的 Tk image 命令只是提供了一个对扩展(如 Img 和 TkMagick)的标准接口，这样在 Tcl 中可以方便地进行复杂的图像控制，并将结果用 Tk 显示出来。

20.3.3 图像和命名空间

因为在创建图像对象时也会创建有相同名字的图像命令，所以必须注意别覆盖了已有命令名。一种解决方法是让 image create 自动为图像命名，这时它们会有诸如 image0，image1 等名字。在默认的情况下，所有的图像命令都在全局命名空间里创建。

另外一种策略是为图像名创建一个单独的命名空间，如::img，然后用完全符合规范的命名空间名称来创建图像，如::img::logo 和::img::large。使用这个手段，任何与特殊模块或库关联的图像都能在库的命名空间中创建。关于命名空间的更多信息可参见第 10 章。

提示： 要使得这项技术有效，应该总使用完全符合规范的命名空间来给出图像名，如::img::logo，而不是部分符合规范的，如 img::logo。图像命令相对于当前的命名空间创建，所以当创建图像并使用部分符合规范的名称给出图像名时，没有采用全局命名空间，那么图像命令的名称将不会与图像自身的名称匹配。

第 21 章　几何管理器

几何管理器决定了组件的尺寸和位置。Tk 在这一点上和其他的工具集相似，即它不允许组件决定它们自己的几何外观。组件甚至不会在屏幕上显示，除非它被几何管理器管理。几何管理器与内部组件行为的分离允许多个几何管理器同时存在，并允许任意几何管理器使用任意组件。如果组件控制它们自己的几何外观，就会失去这种灵活性：这样每个已有组件可能需要修改，以引入一个新的布局样式。

本章将介绍几何管理器的整体结构，并给出三种几何管理器。网格管理器是 Tk 里最常用的几何管理器，用于将组件按行列布置。打包器将组件布置在一个区域周围。定位器提供了组件的简单固定和“橡胶板”定位。一些组件类也能起到几何管理器的作用：分别将在第 23 章和第 24 章介绍的 canvas 和 text；panedwindow(和相应的主题组件类 ttk::panedwindow)，它在第 18 章中介绍过；还有在第 19 章介绍过的 ttk::notebook。

21.1　本章出现的命令

所有的几何管理器都有一个公共的子命令集，也有一个独特功能集。下面就是它们共有的子命令，其中 *gm* 是 place、pack 或 grid 中的一个。

- `gm slave ?slave ... ? option value ? option value ...?`
  和下面介绍的 configure 子命令相同，除非您正在查询几何管理器选项。
- `gm configure slave ? slave...? ?option?`
  `?value option value ...?`
  安排几何管理器控制由 *slave* 参数命名的组件的几何外观。*option* 和 *value* 参数提供了决定从组件尺寸和位置的信息。
- `gm forget slave ?slave ...?`
  使几何管理器停止管理用 *slave* 参数命名的组件，并解除它们到屏幕的映射。如果 *slave* 现在没有被这个几何管理器控制，则这条命令不会有任何效果。
- `gm info slave`
  返回一列值，给出 *slave* 的当前配置。这列值由选项-值对组成，它们的格式与指定给几何管理器的 configure 命令的格式完全相同。如果 *slave* 现在没由几何管理器控制，则会返回一个空字符串。
- `gm slaves master ?option value?`
  返回该几何管理器的 *master* 列表上的从组件。

本章还讨论了以下用来管理组件堆栈顺序的命令。

- `lower` *`widget ?belowThis?`*
  改变 *widget* 在堆栈顺序中的位置，这样它就恰好处在由 *belowThis* 给出的组件下方。如果 *belowThis* 被忽略，它会将 *widget* 移到堆栈顺序的底部。
- `raise` *`widget ?aboveThis?`*
  改变 *widget* 在堆栈顺序中的位置，这样它就恰好处在由 *aboveThis* 给出的组件上方。如果 *aboveThis* 被忽略，它会将 *widget* 移到堆栈顺序的顶部。

21.2 几何管理器概览

几何管理器的工作是相对于主组件布置一个或多个从组件。例如，它可能将三个从组件从左到右布置在主组件区域上，或者它可能布置两个从组件，这样它们便能隔开主组件空间，此时，一个从组件会占据上半部分，而另一个会占据下半部分。不同几何管理器会采用不同类型的布局。主组件通常是从组件的父组件，但有时使用其他窗口作为主组件更方便一些(稍后会介绍这样的示例)。

主组件通常是框架或者顶层组件，但并不是任意组件都行。主组件只为从组件定义几何边界，从组件的内容不会受到影响。从组件通常显示在主组件的上方，但也可以在需要时布置到下方。堆栈顺序是组件创建的顺序(详情参见 21.8 节)。

几何管理器接收三条信息，用来计算布局(参见图 21.1)。首先，每个从组件都需要一个特定的宽度和高度。它们通常是组件显示自身信息所需要的最小尺寸。例如，按钮组件通常需要一个能恰好显示自己的文本或图像的尺寸。虽然几何管理器并没有被规定必须要满足从组件的要求，但一般都会满足这些要求的。

几何管理器的第二类输入来自应用程序设计者，用来控制布局算法。这条信息的属性因几何管理器而异。例如，网格管理器按行列布置从组件，所以它需要知道行数和列数，来识别组件的位置。

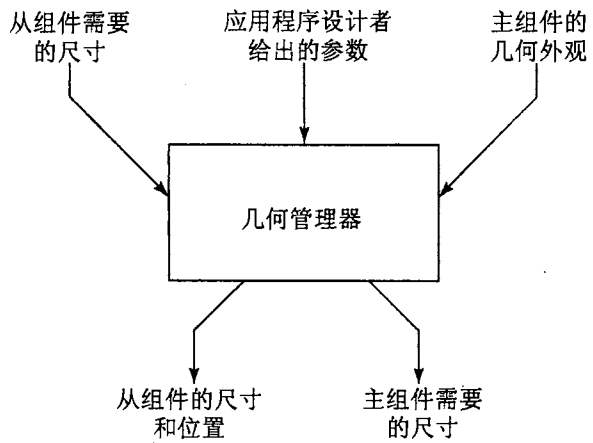


图 21.1 几何管理器决定了从组件的尺寸和位置

第三个被几何管理器使用的信息是主窗口的几何外观。例如，几何管理器可能将一个从组件放在主组件的左下角，或者将主组件区域分给一个或多个从组件，或者如果从组件不适合主组件的区域，几何管理器将不显示从组件。

在接收到上述所有信息后，几何管理器执行一个布局算法来决定每个从组件的宽度、高度和位置。如果几何管理器给定从组件一个与要求不符的尺寸，则组件必须尽力让几何管理器满足自己的要求。几何管理器通常试图给予组件它们所要求的空间，但有时，它们可能给组件额外的空间，来生成更多有吸引力的布局。如果主组件内部没有足够空间用于所有从组件，一些从组件可能得到少于它们所要求的空间。在一些极端的情况下，几何管理器可能选择完全不显示一些从组件。

在应用程序运行时，几何管理器正在控制的信息可能改变。例如，按钮可能用另外的字体或位图来重新配置，此时它会改变它所需要的尺寸。另外，几何管理器可能被告知采用另一种方法(例如，按照从上到下的顺序布置窗口，而不是从左到右)，一些从组件可能会被删除，或者用户可能交互式地重新设定主窗口尺寸。若上述任一项改变，几何管理器会重新计算布局。

一些几何管理器为主窗口设定它所要求的尺寸。例如，网格管理器计算出主窗口按照应用程序设计者要求的模式布置所有从窗口所需要的空间。然后几何管理器会将其设为主窗口需要的尺寸，覆盖此前主窗口自身要求的任何值。这种方法考虑到了层级几何管理系统，在这种系统里，每一个主窗口都是另一个更高层主窗口的从窗口。尺寸的要求由层级结构从每一个从组件向上传到它的主组件，最终产生顶层窗口所需要的尺寸，该尺寸又传给窗口管理器。接着实际的几何信息会由层级结构向下传递，每一层的几何管理器使用主组件的几何外观来计算一个或多个从组件的几何外观。结果，整个层级结构会调整自身的尺寸来满足最底层从窗口的要求；而最终效果就是主组件能“恰好包住”它们的从组件。

在同一时间，每个组件最多可由一个几何管理器管理，虽然在从组件生存周期内，可能转为由别的几何管理器来管理。一个组件可作为任意多从组件的主组件，单个几何管理器可以同时管理与不同主组件关联的从组件组。虽然不同几何管理器可能控制同一主组件下不同的从组件，但这样的控制并不被提倡。

只有内部组件才可能是几何管理器的从组件。这项技术并不用于顶层组件，因为它们由显示器的窗口管理器管理。关于如何控制顶层组件几何外观的信息可参见第26章。

21.3 网格管理器

网格管理器是一个简单明了的几何管理器，用来将从组件按行列布置，并允许它们占据多行或多列。网格管理器的选项可以满足大多数应用程序的布局要求，而无需求助于另外的框架和层级结构布局技术(参见21.7节)。

主组件的每个从组件在网格中被赋予一个由行和列决定的位置，或者称为单元格。网格管理器按照一列中最宽从组件的宽度决定每列宽度，并按照一行中最高从组件的高度决定每行高度，如图21.2所示。操作步骤如下。

(1) 根据每个从组件所需要的尺寸，以完全容纳所有组件为度，以此决定每行和每列最小的尺寸。这样，最小的尺寸就会变为主组件所需要的尺寸。

(2) 主组件的实际尺寸会和这个要求的尺寸对比，然后就可以按需要向布局中添加或删除空间。

(3) 每个从组件按照它的配置标识布置在单元格内。

网格几何管理器将从组件按行列布置，如图 21.2(a)所示。从组件也可以占据多行或多列，这样就可以占据多个单元格，如图 21.2(b)所示。行和列“恰好包住”组件，这样主组件只需要大到能装满所有从组件即可，如图 21.2(c)所示。

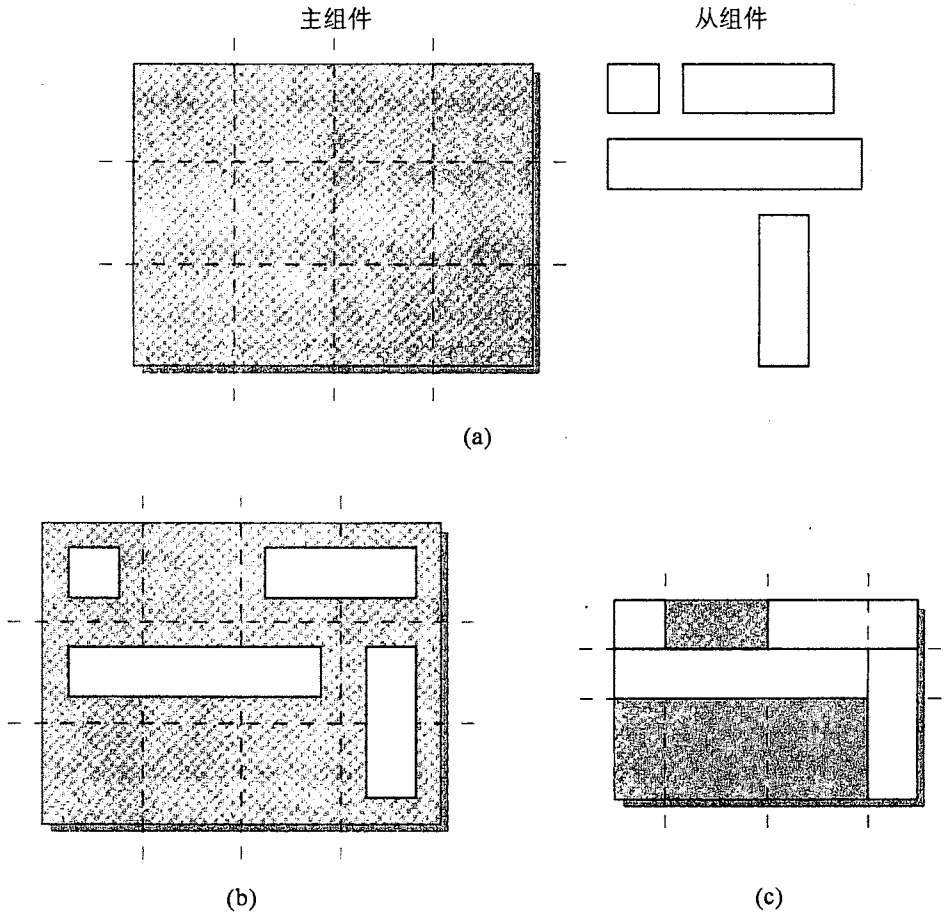


图 21.2 网格几何管理器将从组件按行列布置

grid configure 命令(configure 是可选的，除非您正在查询网格选项)为网格控件添加一个或多个组件，或者改变已由网格控制的组件的网格选项。

```
grid slave ?slave ...? option value ?option value...?
grid configure slave ?slave ...? ?option? \
    ?value option value ...?
```

网格管理器有很多不同的配置选项，用来控制从组件位置、拉伸行为、单元格尺寸和网格尺寸。

- `-column n`
  在第 *n* 列插入从组件。
- `-columnspan n`
  插入一个能占据 *n* 列的从组件(参见 21.3.3 节)。
- `-in master`
  将 *master* 作为从组件的主组件。*master* 必须是从组件的父组件，或是从组件的父组件的下层组件。如果该选项未被设定，则主组件默认为从组件的父组件。
- `-ipadx distance`
  在组件边框和内容之间添加内部水平补白，单位为屏幕距离。
- `-ipady distance`
  在组件边框和内容之间添加内部垂直补白，单位为屏幕距离。
- `-padx distance`
  在组件周围添加外部水平补白，单位为屏幕距离；可以使用二元素列表分别指定左侧和右侧补白(参见 21.5 节)。
- `-pady distance`
  在组件周围添加外部垂直补白，单位为屏幕距离；可以使用二元素列表指定上侧和下侧补白(参见 21.5 节)。
- `-row n`
  在第 *n* 行插入从组件。
- `-rowspan n`
  插入一个能占据 *n* 行的从组件(参见 21.3.3 节)。
- `-sticky style`
  如果从组件单元格大于需要的尺寸，则可控制从组件的位置和扩展(参见 21.3.1 节)。

网格管理器也支持 columnconfigure 和 rowconfigure 子命令，用来设置或查询主组件中列和行的配置。

```
grid columnconfigure master index ?option? ?option? \
    ?value option value ...?
grid rowconfigure master index ?option? ?option? \
    ?value option value ...?
```

如果提供了一个或多个选项，*index* 可能作为列索引列表给出，配置选项会用在该索引上。索引可能是整数、网格中的组件名或关键字 all。当前支持的选项包括：

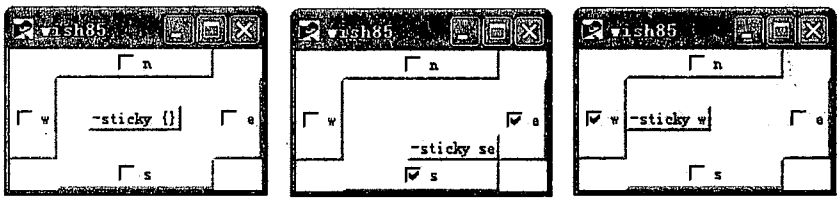
- `-minsize amount`
  一行或一列的最小尺寸，单位为屏幕距离。
- `-pad amount`
  向行/列中的每个组件添加外部补白，单位为屏幕距离。
- `-weight int`
  在改变尺寸时，列周围多余的空间中各部分的相对权重(参见 21.3.3 节)。
- `-uniform groupName`
  一组列/行的符号名，这些列/行按比例改变尺寸(参见 21.3.3 节)。

21.3.1 grid 命令和-sticky 选项

grid 命令用来和网格管理器通信。该命令最简单的形式是以一些从组件和选项为参数，将每个从组件放在以 0 开始索引的连续列中。同一主组件的每个后续的 grid 命令都从新行开始。-row 和-column 选项可以提供一个特定的单元格或者起始点。

如果从组件不是整个填满单元格，则它默认会放在单元格的中间。-sticky 选项将从组件钉在任意一边或者任一个方向，或向该边或该方向扩展。该选项的值为 n、s、e 或 w 中的一个或多个。每个字符会使从组件靠在单元格的相应边上；如果给出了相对边，则从组件会扩展到两边。

下列代码使用一个在 3×3 网格中被四个复选按钮包围的标签组件，来显示不同-sticky 选项的效果。同时要注意一些单元格，例如这个示例中的四个角，可能是空白的。这时，主组件会显示出来。图 21.3 显示了该应用程序的一些示例。



(a) (b) (c)

图 21.3 中间的标签演示了不同的-sticky 选项

```
label .demo -textvariable stickyLabel -bd 2 -relief raised
checkbutton .n -text "n" -bd 2 -relief raised  \
    -variable stickyN -onvalue n -offvalue {}  \
    -command redo_sticky
checkbutton .s -text "s" -bd 2 -relief raised  \
    -variable stickyS -onvalue s -offvalue {}  \
    -command redo_sticky
checkbutton .e -text "e" -bd 2 -relief raised  \
    -variable stickyE -onvalue e -offvalue {}  \
    -command redo_sticky
checkbutton .w -text "w" -bd 2 -relief raised  \
    -variable stickyW -onvalue w -offvalue {}  \
    -command redo_sticky

grid .demo  -row 1  -column 1
grid .n     -row 0  -column 1   -sticky nsew
grid .s     -row 2  -column 1   -sticky nsew
grid .e     -row 1  -column 2   -sticky nsew
grid .w     -row 1  -column 0   -sticky nsew

grid rowconfigure . 1 -weight 1
grid columnconfigure . 1 -weight 1

wm geometry  . 180x100

proc redo_sticky {} {
    global stickyN stickyS stickyE stickyW stickyLabel

    append s2 $stickyN $stickyS $stickyE $stickyW
```

```
    grid .demo -row 1 -column 1 -sticky $s2
    set stickyLabel [list -sticky $s2]
}
```

21.3.2 跨行和跨列

网格化组件可以跨越多行或多列。在下面的代码中，-columnspan 选项会使.label 组件使用第一行中的两列。该区域在稍后会作为单个单元格，如图 21.4 所示。类似地，-rowspan 选项也可以跨越多行。

图 21.4 跨越多列的标签

```
grid .label -columnspan 2
grid .listbox -sticky nsew
grid .scrollbar -row 1 -column 1 -sticky ns
grid .xscrollbar -sticky ew
```

在网格管理器计算最小主组件尺寸时，它最先检查行和列跨度为 1 的所有组件。然后是行和列跨度大于 1 的组件，如果还需要多余空间，则该空间会在扩展行或列上分布。

21.3.3 拉伸行为与-weight 和-uniform 选项

如果主窗口空间超过了从窗口所需，例如，当用户交互式扩展窗口，则网格管理器必须合理安排多余的空间。一些选项可以控制多余空间如何分布。例如，在图 21.4 中，用户想让所有多余空间首先赋给列表框组件，然后按需要来扩展滚动条。给第 0 列和第 1 行.listbox 组件占据的单元格赋予更大的权重可以方便地实现这一点，如图 21.5 所示。权重通过 grid rowconfigure 和 grid columnconfigure 命令的-weight 选项赋给行和列。默认权重为 0，此时图 21.5(a)中的窗口会调整为如图 21.5(b)所示，则组件不会填满整个空间。如果第一行的权重为 1，则这些行会扩展来填满整个空间，如图 21.5(c)所示；如果第 0 列的权重为 1，则列会扩展来填满整个空间，如图 21.5(d)所示。

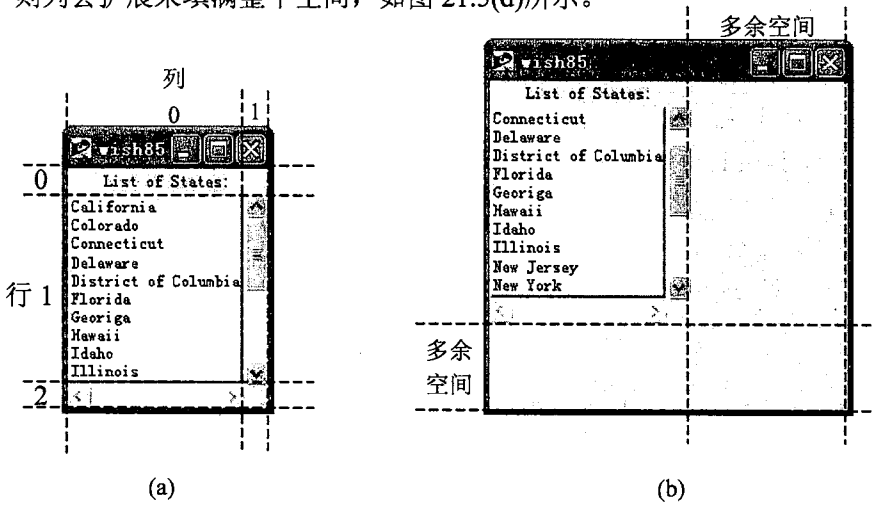


图 21.5 在网格管理器中使用不同权重的行和列的效果

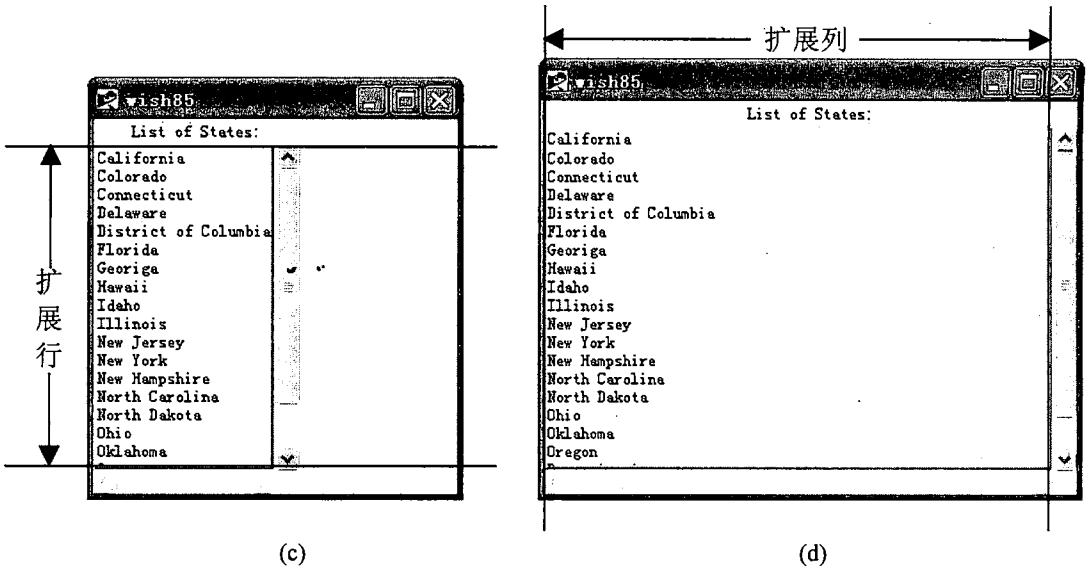


图 21.5 （续）

网格管理器默认赋给所有行和列的权重为 0。这些权重为 0 的行和列不会分配到多余的空间。大于 0 的权重会使多余空间按比例分配，用-weight 2 得到的空间为用-weight 1 的两倍。

-uniform 选项将使用类似权重的行和列划为一组。用这种方法布置的行或列总是有相同的尺寸。例如，用-weight 1 -uniform a 配置的多行都有相同高度。更多信息可参见参考文档。

21.3.4 相对位置字符

grid 命令提供了用来放置从组件的简便记号，无需明确地指定行数和列数。grid 命令中每个指定从组件默认放在连续列中，每个新的 grid 命令都从新行开始。这便在源代码中创建了一个组织形式，能表示屏幕上的布局。

grid 命令中用于从组件的相对位置字符有三个：“-”、“x”和“^”。“-”字符可以将前一个从组件扩展到下一列，每个后续的“-”字符会为从组件增加列宽。“x”字符会使单元格为空。“^”字符可以将前一行的从组件向下扩展到当前单元格，grid 命令中每个后续的“^”字符会为从组件增加行高。图 21.6 演示了每个相对位置字符的用法，代码如下：

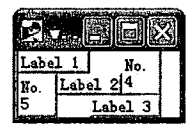


图 21.6 相对位置字符能简化网格布局命令

```
ttk::label .w1 -text "Label 1" -relief raised
ttk::label .w2 -text "Label 2" -relief raised
ttk::label .w3 -text "Label 3" -relief raised
ttk::label .w4 -text "No. \n4" -relief raised
ttk::label .w5 -text "No. \n5" -relief raised
grid   .w1  -   x   .w4 -sticky nsew
grid   .w5 .w2  -    ^  -sticky nsew
grid    ^   x   .w3  -  -sticky nsew
```

21.4 打包器

打包器每次将一个从组件放在主组件窗口中，从边缘向中间布置。在简单的情况下，例如只需一行或一列，打包器用起来很方便，但它也能用于复杂的布局。虽然可以使用打包器或网格管理器来布置组件，但用网格管理器来创建复杂的布局通常更简单一些。

打包器维护着给定主窗口的子窗口列表，该列表称为打包列表。打包器按列表顺序，一次一个地布置从组件。如果正在处理某个从组件，而主窗口的部分区域已被分配给列表前头的从组件，则会将一个未分配的长方形区域留给所有剩余的从组件，该区域称为空腔，如图 21.7(a)所示。当前的从组件用三个步骤定位：分配块区，扩展这个从组件，并将该组件放入该区域。

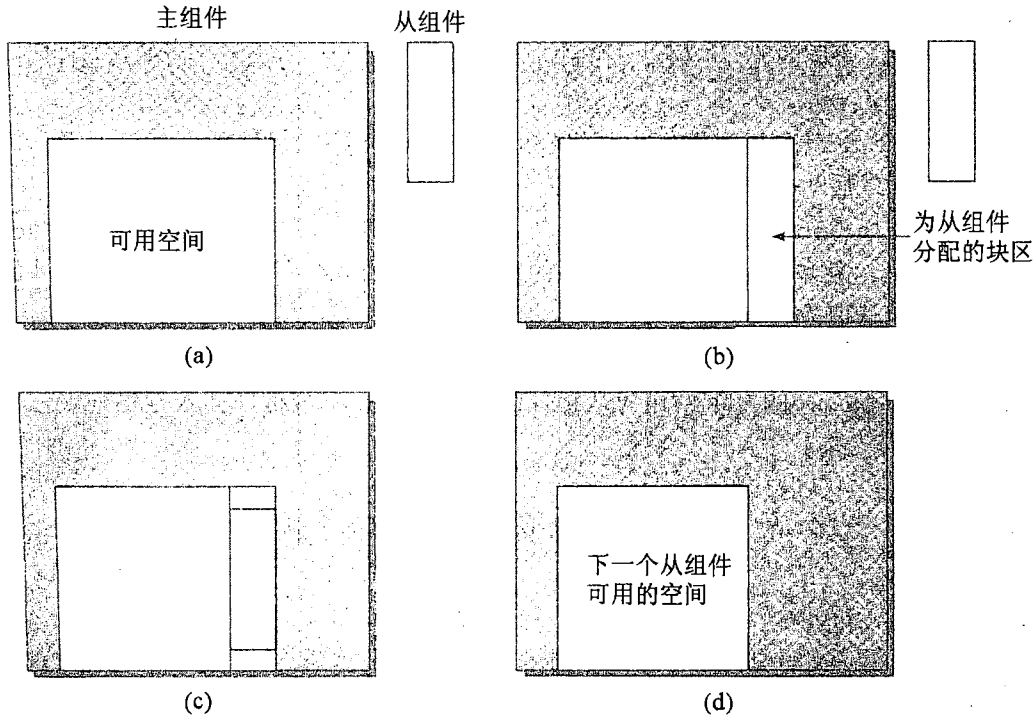


图 21.7 打包器在主窗口中布置从窗口

在第一步里，称为块区(parcel)或打包区的长方形区域来自可用空间。沿可用空间一边“切下”一小片就可以得到一个块区。例如，在图 21.7(b)中，块区从可用空间的右侧切下。该块区的一个尺度由可用空间的尺寸决定，另外一个尺度是受控的。受控尺度通常来自该尺度上从组件要求的尺寸，但打包器允许您要求更多空间。图 21.7(b)中的宽度就是一个受控尺度；如果从上方或者下方切下了一小块区域，则该块区的高度是受控尺度，宽度则不是。

在第二步里，打包器选择从组件的尺寸。从组件默认得到它所要求的尺寸，但是可以另外指定，这样它应该在一个或两个尺度上扩展，并填满块区的空间。如果从组件需要的尺寸大于块区，则它会缩小，以适合块区的尺寸。

第三步是在块区中定位从组件。如果从组件小于块区，从组件会给出一个锚定位置，如 n、s 或者 center。图 21.7(c)中的从组件默认放在块区的中间。

在定位好从组件后，一个更小的长方形区域被留给下一个从组件，如图 21.7(d)所示。如果一个从组件未用完块区的所有空间，如图 21.7 所示，则剩余的空间不会分给后面的从组件。因此，在打包过程中的每一步都从可用的空间中的一个长方形区域开始，至一个更小的长方形区域结束。

在打包完所有从组件后，打包器会调整主组件的尺寸，以满足最底层从组件的需要，整体的效果就是主组件会“恰好包住”它的从组件。

pack configure 命令(configure 是可选的，除非您正在查询打包器选项)为打包器控件添加一个或多个组件，或者改变已被打包器控制的组件的打包选项。

```
pack slave ?slave ...? option value ?option value ...?
pack configure slave ?slave ... ? ?option? \
        ?value option value ...?
```

打包器有很多不同的配置选项，来控制诸如位置、扩展行为、单元格尺寸等从组件属性。

- `-after widget`
  使用 *widget* 的主组件作为从组件的主组件，并将从组件插入打包列表，位于 *widget* 之后。
- `-anchor position`
  如果从组件单元格大于它所需要的尺寸(参见 21.4.4 节)，则控制该从组件的位置。默认值为 center。
- `-before window`
  使用 *window* 的主组件作为从组件的主组件，并将从组件插入打包列表，位于 *window* 之前。
- `-expand boolean`
  如果布尔运算值为真，从组件的块区会扩大，以能占满主组件剩下的任意多余空间(参见 21.4.3 节)。默认值为假。
- `-fill style`
  指定如果从组件的块区大于其要求的尺寸，是否(或怎样)扩大该组件(参见 21.4.2 节)。*style* 必须是 none、x、y 或 both 中的一个。默认值为 none。
- `-in master`
  使用 *master* 作为从组件的主组件。*master* 必须是从组件的父组件或者从组件父组件的下层组件。如果没有指定选项，则主组件默认为从组件的父组件。
- `-ipadx distance`
  在组件的边缘和内容之间添加内部水平补白，单位为屏幕距离。
- `-ipady distance`
  在组件的边缘和内容之间添加内部垂直补白，单位为屏幕距离。
- `-padx distance`
  在组件周围添加外部水平补白，单位为屏幕距离；可以使用二元素列表分别指定

左侧和右侧的补白(参见 21.5 节)。

- `-pady` *distance*
  在组件周围添加外部垂直补白，单位为屏幕距离；可以使用二元素列表分别指定上侧和下侧的补白(参见 21.5 节)。
- `-side` *side*
  side 指定了主组件中从组件应该靠着的边(参见 21.4.1 节)。必须是 top、bottom、left 或 right。默认值为 top。

21.4.1　pack 命令和-side 选项

pack 命令用于与打包器通信。它最简单的格式是将一个或多个组件名作为参数，后面是一个或多个能说明如何控制组件的其他参数对。例如，考虑下面的脚本，它创建三个按钮，并将它们排在一行。

```
ttk::button .ok -text OK
ttk::button .cancel -text Cancel
ttk::button .help -text Help
pack .ok .cancel .help -side left
```

pack 命令让打包器将.ok、.cancel 和.help 作为从组件来管理，并按照此顺序布置它们。从组件的默认主组件是它们的父组件，也就是主组件“.”。选项-side left 作用于全部的三个组件，它意味着每个从组件的块区应该布置在可用空间的左侧。因为没有指定其他选项，每个从组件的块区会恰好容纳从组件。这使得从组件在主组件上方从左到右排成一行。而且，打包器会计算出主组件要安置所有从组件所需的最小尺寸。窗口管理器将主组件尺寸设定为需要的尺寸，因此主组件最终会恰好包住从组件，如图 21.8(a)所示。

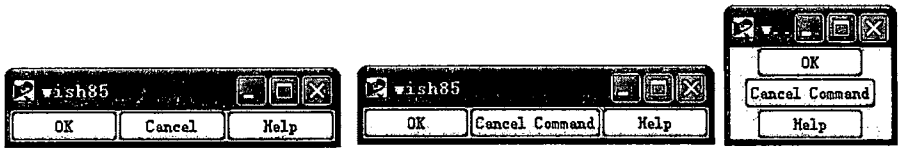
在任意相关信息改变时，打包器会重新计算布局。如果输入命令

```
.cancel configure -text "Cancel Command"
```

来改变中间按钮的文本，按钮会改变它所需的尺寸，这时打包器会调整布局，以留出更多空间，如图 21.8(b)所示。如果继续输入命令

```
pack .ok .cancel .help -side top
```

布局将发生改变，这样每个从组件会分配到剩余空间的顶层，制造出图 21.8(c)中的列状分布。此时，主组件的宽度将可以安排下最大的从组件(.cancel)。其他的每个从组件会接受一个比需要的更宽的块区，这样从组件就会在块区的中间。



(a)　(b)　(c)

图 21.8　简单的打包器示例

21.4.2 充满

如果块区最终大于它的从组件所需的尺寸，可以使用-fill 选项来扩展该从组件，使得它在一个或两个方向上充满该块区。例如，考虑图 21.8(c)中的组件列：给出了每个组件所需的尺寸，由于各个按钮的尺寸不同，故会导致参差不齐的外观。-fill 选项可以调节按钮的尺寸，使所有按钮尺寸相同。

```
pack .ok .cancel .help -side top -fill x
```

图 21.9 显示了这条命令的效果：每个从组件水平扩展，以填满它的块区，因为块区和主窗口一样宽，故所有从组件最终宽度相同。

图 21.10 显示了充满的另一个简单示例。三个窗口按照不同的方式配置，因此对每个窗口分别使用 pack 命令。pack 命令的顺序决定了堆栈列表中窗口的顺序。

```
pack .label -side top -fill x
pack .scrollbar -side right -fill y
pack .listbox
```

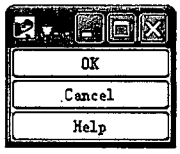


图 21.9 使用-fill 的效果

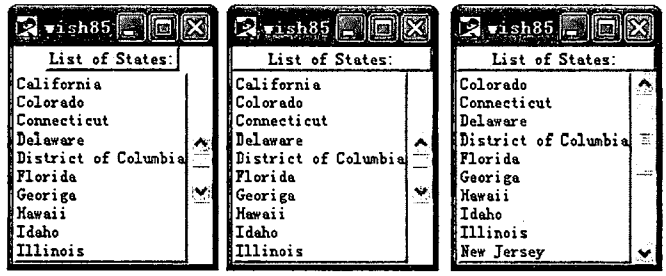


图 21.10 使用-fill 和-side 选项的另一个打包器示例

.label 组件最先被打包，它会占据主窗口的上面部分。-fill x 选项指定窗口应该水平扩展，这样就能填满整个块区。接下来打包的是滚动条组件，打包方式和.label 组件相似，但是它被布置在窗口的右侧，且垂直扩展。最后一个打包的组件是.listbox 组件。它不需要指定任何选项：它会显示在相同的位置，而无需考虑靠着哪一边。

如果没有-fill 选项，标签和滚动条可能只是占据块区的中间，如图 21.10(b)所示。使用-fill 会使它们扩展并占据整个块区，如图 21.10(c)所示。

21.4.3 扩充

主窗口的空间有时会比它的从组件所需的更大。这种情况可能出现，例如，用户交互式地扩展窗口。这时，打包器的默认行为是不使用多余的空间，如图 21.11(a)所示。但是，可以使用-expand 选项，让打包器将多余空间分给某个特定的从窗口(例如，扩充文本组件，来充满所有多余空间)，或者将一系列从组件的多余空间均匀划分(例如，在可用空间中均匀分布一些按钮)。-expand 选项可以制造出有吸引力的布局，而无需考虑用户如何调整窗口尺寸，这样用户就可以选择偏爱的尺寸了。在图 21.11 的每个示例中，主组件的尺寸由 wm geometry . 232x62 命令来固定，以模拟用户交互式改变窗口尺寸的情况。

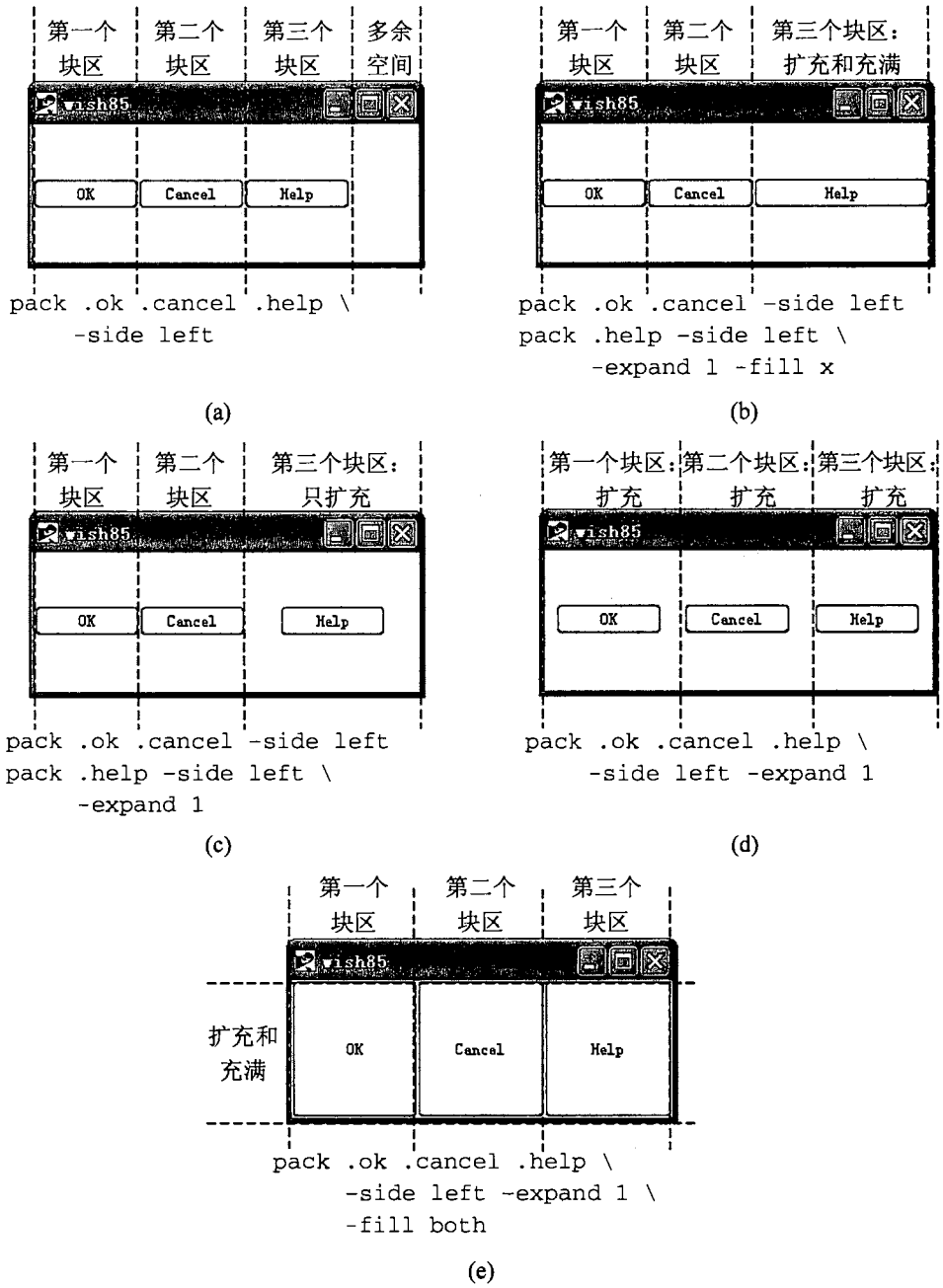


图 21.11　使用-expand 和-fill 的示例

图 21.11 显示了-expand 是如何工作的。如果为从组件指定了-expand，就像图 21.11(b)中的.help 那样，那么从组件的块区会扩充来包含主组件中任何多余的空间。在图 21.11(b)中，指定了.help 的-fill x，因此从组件窗口会延伸，来覆盖块区的全部宽度。图 21.11(c)和图 21.11(b)相似，只是前者没有指定充满，因此.help 在块区中只是居中放置。

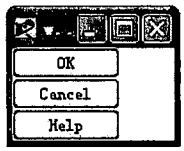
如果为多个从组件指定了-expand，它们的块区会均匀占据多余的空间。在图 21.11(d)中，所有窗口都由-expand 打包，但是没有充满，而在图 21.11(e)中，从组件在两个方向上都充满。

提示：-expand 和-fill 选项通常会被弄混，因此它们的名称很接近。-expand 选项决定了块区是否要占据主组件的多余空间，而-fill 决定了从组件的窗口是否要占据它的块区中的多余空间。这两个选项通常在一起使用，这样从组件的窗口便会占据它的主窗口中全部的多余空间。

21.4.4 锚定

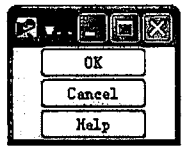
如果块区的空间比它的从组件所要求的多(例如，因为指定了-expand)，并且如果选择了不用-fill 选项扩展从组件，打包器通常会将从组件置于它的块区中间。-anchor 选项用于设置另一个位置：它的值指定了从组件在块区中应该被放置的位置，该值的格式为 18.15.3 节中的格式之一。例如，-anchor nw 指定了从组件应该放置在块区左上角。图 21.12(a)显示了-anchor 如何将一些按钮左对齐。

-fill 和-anchor 选项保留了-padx 和-pady 所需的外部补白。例如，图 21.12(b)和图 21.12(a)相同，只是前者要求了外部补白。此时，按钮不会放置在它们的块区的左边缘，而是按照-padx 距离来嵌入。




```
pack .ok .cancel .help -side left
```

(a)




```
pack .ok .cancel -side left
pack .help -side left -expand 1 -fill x
```

(b)

图 21.12 -anchor 选项的示例

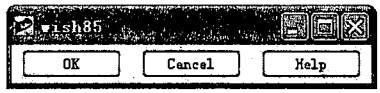
21.4.5 打包顺序

-before 和-after 选项用于在主窗口的打包列表中控制从窗口的位置。如果没有指定这种选项中的任一个，则每个新的从窗口就会出现在打包列表的末尾。如果指定了-before 或-after，它的值就必须是已被打包的窗口的名称，且新的从窗口会分别放置在给定从组件之前或之后。

21.5　补白

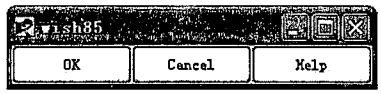
打包器和网格管理器都提供了 4 个选项，用于为从组件要求额外空间。额外的空间称为补白，它有两种格式：外部补白和内部补白。外部补白可以使用-padx 和-pady 选项给出；这些选项使打包器能分配一个比从组件要求的更大的块区，并将额外的空间放在该从组件周围。例如，图 21.13(a)中的组件可以使用下面这个选项来打包。

```
-padx 2m -pady 1m
```



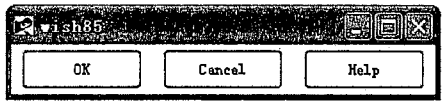

```
grid .ok .cancel .help -padx 2m -pady 1m
```

(a)




```
grid .ok .cancel .help -ipadx 2m -ipady 1m
```

(b)




```
grid .ok .cancel .help -padx 2m -pady 1m \
     -ipadx 2m -ipady 1m
```

(c)

图 21.13　补白的示例

这些选项指定了每个组件的每边应该有 2 mm 的多余空间，每个组件的上方和下方边缘还可以有 1 mm 的多余空间。-ipadx 和-ipady 选项用于请求内部补白；它们也会使一个块区大于从窗口的要求，但此时，从窗口会被扩大，以包含多余的空间，如图 21.13(b)所示。外部和内部的补白可以一起使用，如图 21.13(c)所示，不同从组件可以有不同数目的补白。

21.6　定位器

定位器是最简单的几何管理器，通常用在一些特定的应用程序中。从组件可由管理器精确地放在由 place 命令选项指定的位置，不需要对布局应用特别的算法。这是一种过分的简化，因为它只做了很小一部分工作，让应用程序来控制布局。定位器应该只在网格管理器或打包器不能实现想要的布局时使用。定位器会给出一个相对位置选项，用来为从组件产生一个“橡胶板”。被定位器控制的从窗口不影响其他从窗口的几何外观，在其他几何管理器中也是这样的，传递给主组件的尺寸信息也不会受影响。

place 命令用于和别的定位器通信。它最简单的形式，是将一个或多个组件名作为参数，后面接上一个或多个指定如何放置组件的其他参数对。位置由一个精确的坐标(-x，-y)和一个精确的尺寸(-width，-height)控制，且该位置在主窗口中，如果将位置(-relx，-rely)和尺寸(-relwidth，-relheight)指定为相对主窗口的浮点数，则可以使用相对位置。定位器的其他有效选项可参见参考文档。

21.7 层级结构几何管理

定位器和网格管理器都可用于层级结构布置系统，该系统中的从窗口也是其他从窗口的主窗口。图 21.14 是一个层级结构打包的示例。该布局包含左侧的一列单选按钮和右侧的一列复选按钮，每个按钮组都垂直居中。要达到这种效果，两个额外的框架组件.left 和 right 被打包放在主窗口的两边，而按钮则打包放在它们之间。打包器会设置.left 和.right 所需的尺寸，为按钮提供足够的空间，然后用该信息为主组件设定想要的尺寸。主组件的几何外观设定为窗口管理器所需要的值，然后打包器会在其中布置.left 和.right，最后它会将按钮布置在.left 和.right 里面。以下是生成该布局的代码：

```
frame .left
frame .right
set buttonList [list]
foreach size {8 10 12 18 24} {
    ttk::radiobutton .pts$size -text "$size points" \
        -variable pts -value $size
        lappend buttonList .pts$size
}
ttk::checkbutton .bold -text Bold -variable
bold
ttk::checkbutton .italic -text Italic -variable
italic
ttk::checkbutton .underline -text Underline \
    -variable underline
pack {*}$buttonList -in .left -side top -anchor w
pack .bold .italic .underline -in .right -side top \
    -anchor w
pack .left -side left -padx 3m -pady 3m
pack .right -side right -padx 3m -pady 3m
```

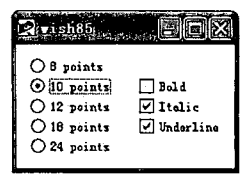


图 21.14 层级结构打包的示例

这个代码也显示了这样一种情况，此时从组件的主组件和它的父组件不同(用-in 选项指定主组件)。创建按钮窗口，将之作为.left 和.right 的子窗口是可能的(例如，用.left.pts8 而不用.pts8)，但是创建它们，将其作为“.”的子窗口，并打包放到.left 和.right 中会更清晰一些。窗口.left 和.right 在应用程序中的唯一用途，是在几何管理系统中起帮助的作用；它们的背景颜色和主组件相同，因此它们在屏幕上不可见。如果按钮是它们的几何主窗口的子窗口，则改变几何管理系统(例如，向打包层级结构添加更多的层)或许需要重命名按钮窗口，也可能会删除使用旧名称的任何代码。给窗口一个能反映它们在应用程序中的逻辑用途的名称，在几何管理器需要的地方构建单独的框架层级结构，并将功能性窗口打包进框架会是一个更好的做法。

提示： 在图 21.14 的示例中，框架.left 和.right 必须在按钮之前创建，这样，按钮在堆栈顺序中会处在框架的顶层。最近创建的组件会处在堆栈顺序的顶端。如果.left 晚于.pts8 创建，它会出现在.pts8 之上，因此.pts8 不可见。另外，raise 和 lower 命令可以调整堆栈的顺序(参见 21.8 节)。

从组件的主组件必须要么是它的父组件，要么是它的父组件的下层组件。作出这种限制的原因是一些窗口系统中的修剪规则。每个窗口都被修剪，以适合它的父窗口的边缘，因此在父窗口之外的子窗口就不会显示出来。Tk 主窗口的这种限制保证，如果从组件的主组件可见且未被修剪，则该从组件也可见且不会被修剪。假定这条限制未被执行，那么.x.y 窗口会有一个.a 作为它的主窗口。同样也假定.a 和.x 一点也不重叠。这时，如果让打包器将.x.y 置于.a 中，则打包器会按要求设定.x.y 的位置，但这可能会使.x.y 处在.x 区域之外，所以窗口系统可能不会显示它，即使此时.a 是完全可见的。这种行为会给程序设计者造成困扰，因此 Tk 限制了该行为。这种限制应用在所有的 Tk 几何管理器上。

21.8 组件堆栈顺序

堆栈顺序决定了屏幕上组件的分层情况。如果两个同级组件重叠了，堆栈顺序可以决定哪一个出现在上方。组件通常按照它们创建的顺序来进行堆栈排列：每个新组件会出现在堆栈顺序的顶端，将之前创建的同级组件遮住。raise 和 lower 命令用于改变组件的堆栈顺序。

```
raise .w
raise .w .x
lower .w
lower .w .x
```

第一个命令提升了.w，这样它就处在堆栈顺序的顶端(遮住所有同级组件)。第二个命令将.w 的堆栈顺序放在.x 之前。第三个命令降低.w 到堆栈顺序的最底部(被所有同级组件遮住)，最后一个命令会将.w 的堆栈顺序放在.x 之后。

大多数时候，堆栈顺序并不重要，因为大多数组件在放置时并不重叠。在使用 place 窗口管理器时，组件可以重叠，当同时使用-in 选项和任一窗口管理器时，从组件必须置于主组件之上，否则从组件不可见。

提示： 您可以用 raise 和 lower 处理顶层组件和内部组件，但并非对所有窗口管理器都有效。一些窗口管理器会阻止应用程序提升和降低它们的顶层窗口；在这些窗口管理器中，必须用窗口管理器功能来手动提升或降低顶层组件。

21.9 其他几何管理器选项

到现在为止我们已经讨论了几何管理器命令最常见的格式，第一个参数是从窗口名，最后一个参数指定了配置选项。21.1 节显示了几何管理器命令的其他一些常见格式，第一个参数选择了若干操作中的一个。这些命令应用于全部三个几何管理器：grid、pack 和 place。例如 grid configure 命令，它的效果和已用过的短格式的效果相同；剩下的参数指定了窗口和配置选项。如果 grid configure(或不带命令选项的短格式)应用于已被网格管理器控制的窗口，则会改变该从组件的配置；grid 命令中未设定的配置选项会保留它们以前的值。

命令选项 slaves 会返回给定主窗口的所有从窗口。对打包器而言，从组件的顺序反映了它们在打包列表中的顺序。

```
    pack slaves .left
⇒ .pts8 .pts10 .pts12 .pts18 .pts24
```

命令选项 info 会返回指定从组件的所有配置选项。

```
    pack info .pts8
⇒ -in .left -anchor w -expand 0 -fill none -ipadx 0?-ipady 0
    -padx 0 -pady 0 -side top
```

返回值是包含配置选项名和值的一个列表，格式应与 pack configure 的完全相等。这条命令可以保存一个从组件的状态，这样它可以在稍后恢复。

命令选项 forget 使几何管理器停止管理一个或多个从组件，并清除对它们所有配置状态的记忆。它也会取消对窗口的映射，这样它们将不再显示在屏幕上。该命令可以将对窗口的控制从一个几何管理器转到另一个，或者只是从屏幕上删除该窗口。如果一个被 forget 的窗口是其他从窗口的主窗口，那么这些从窗口的信息会被保存下来，但这些从窗口不会显示在屏幕上，直到主窗口重新被控制。

提示： 只有网格管理器才有 remove 命令选项。该命令从网格中删除从窗口，并取消了它们对窗口的映射，就像 forget 命令选项那样。但是，该窗口的配置选项会被记录下来，这样当从窗口再次由网格管理器管理时，可以恢复以前的值。

命令选项 propagate 用于控制是否让打包器或网格管理器为主窗口设定要求的尺寸。通常这些几何管理器会为每个主窗口设定要求的尺寸，以满足它们的从组件的要求，并且在需要改变从组件时，会更新所要求的尺寸。该功能被称为几何传播，它会覆盖主窗口可能要求的任何尺寸。您可以使用下列命令之一来使该传播失效，这取决于哪一个几何管理器正在控制主窗口的布局。

```
pack propagate master 0
grid propagate master 0
```

其中的 *master* 是主窗口名。该命令让 Tk 不为 *master* 设置它要求的值，这样就可以使用主组件要求的值。您可以为这条命令赋值 1，而不是 0，以恢复几何传播。

每个几何管理器都有自身算法的独特选项。选项的完整列表可参见参考文档。

21.10 Tk 里的其他几何管理器

如前所述，存在可作为几何管理器的 Tk 组件。这就是说，它们可以向显示器映射组件并控制它们的尺寸和位置。文本组件能够向文本文档插入窗口，每个窗口会被视为一个字体字形。窗口要求的尺寸会成为字形尺寸，而文本组件会相应地映射该窗口。画布组件也可以插入窗口并向显示器映射它们。它用自己的方式在画布上放置对象。您可以分别在第 23 章和第 24 章找到关于画布和文本组件的更多信息。

分栏窗口组件(和它相应的主题组件 ttk::panedwindow)里面有一个或多个组件，它们会显示在水平行或垂直列上。组件之间由分隔线分开，该分隔线可由用户用鼠标调节。当分隔线移动时，可以调节在它两侧的每个组件的尺寸。图 21.15 是一个有两个窗格的分栏窗口。左窗格是一个带列表框和滚动条的框架。右窗格是一个带文本组件和滚动条的框架。分隔线的宽度和控制柄的尺寸都可调节。

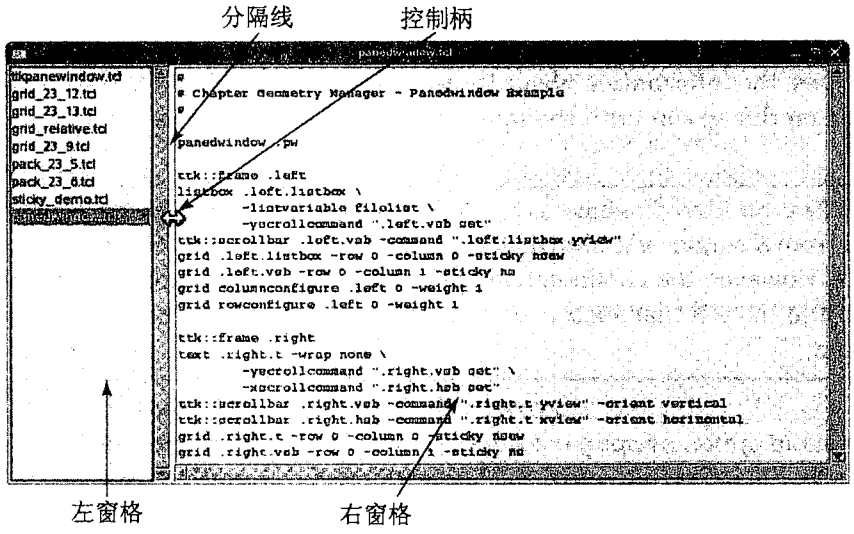


图 21.15 分栏窗口组件有几何管理器的功能

同样，主题记事本组件 ttk::notebook 可以用作几何管理器的一部分。当您为其添加更多需要控制的组件时，它会根据它们的从组件来显示标签。除标签被选中的从组件外，其他组件都被断开映射。选择另一个标签会取消对当前从组件的映射，并建立其所选标签的从组件的映射。图 21.16 是带三个标签的记事本的示例。被选中的标签映射了一个包含一个网格化文本组件、垂直滚动条和水平滚动条的从属框架。

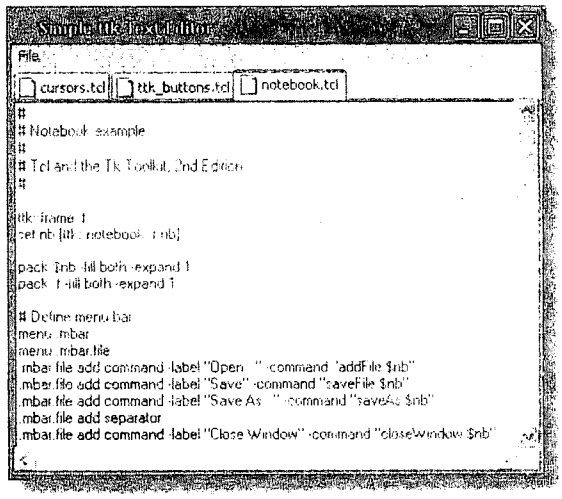
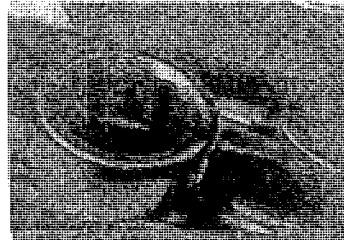


图 21.16　记事本组件有几何管理器的功能

最后，wm 命令用来控制顶层窗口，特别是用 toplevel 命令创建的窗口，但它也能用于 dock(停止管理一个顶层窗口)或者 undock(管理一个顶层窗口)任何组件。在第 26 章中会更详细地介绍 wm 命令。



第22章 事件和绑定

如前所述，Tcl 脚本可以与一些事件关联，如按钮或者菜单，这样在一些事件发生时（如在组件上点击鼠标），就会调用脚本。这样的机制也称为指定对件类的特定功能来提供。Tk 也提供一个更通用的机制，称为绑定，它用于为任意窗口中的任意事件创建处理器。“绑定”将一个 Tcl 脚本和一个或多个窗口中的一个事件或一系列事件“绑”在一起，这样在给定的事件序列出现在任一个窗口中时，脚本会被处理。您可以使用绑定来扩展一个组件的基本功能，例如，通过为常用动作定义快捷键。您还可以覆盖或修改组件的默认行为，因为它们都是用绑定来执行的。

22.1 本章出现的命令

本章将介绍用于管理事件和绑定的下述命令。

- `bind tag sequence script`
  设定由 *tag* 给出的窗口中出现由 *sequence* 给出的事件序列时，处理 *script*。如果 *tag* 和 *sequence* 已经有一个绑定，则该绑定会被替换。如果 *script* 是一个空字符串且存在 *tag* 和 *sequence* 的绑定，则该绑定会被删除。如果 *script* 的第一个字符是“+”，该脚本会附到任一个绑定的已有脚本之后。
- `bind tag sequence`
  如果 *tag* 和 *sequence* 的绑定存在，就返回它的脚本。否则它会返回一个空字符串。
- `bind tag`
  返回 *tag* 绑定的序列。
- `bindtags widget ?tagList?`
  将与 *widget* 关联的绑定标记设为 *tagList*。如果 *tagList* 被忽略，则返回当前与 widget 关联的绑定标记。
- `event add <<virtual>> sequence ?sequence ...?`
  将虚拟事件 *virtual* 和物理事件 *sequence* 关联。
- `event delete <<virtual>> ?sequence sequence ...?`
  删除与虚拟事件 *virtual* 关联的每个 *sequence*。如果没有给出 *sequence*，则会删除所有序列。

- `event generate widget event ?option value option value ...?`
  创建一个窗口事件，并将其作为窗口系统事件处理。*widget* 给出了生成事件的窗口路径名。*event* 可能采用 bind 命令的序列参数允许的任意格式，但它必须由单个事件模式组成，而不是一个序列。给出的 *option value* 对用于事件绑定的“%”置换中。
- `event info ?<<virtual>>?`
  返回与一个特定虚拟事件关联的序列，如果没有指定事件，会返回当前定义的虚拟事件列表。

22.2 事件

事件是窗口系统生成的记录，用来告知应用程序一个可能出现的有意义的情况。每个事件都有一种类型，用来表示出现的事件的一般类别。绑定中最常用的事件类型是那些用于用户动作的类型，例如按键或者鼠标光标位置的变化。此外还存在其他很多事件类型，例如在改变窗口尺寸、删除窗口、需要重新显示窗口等时候生成的事件类型。下面是最常用的事件类型，更多关于 bind 命令的细节可参见参考文档。

- Key 或者 KeyPress——按下按键。
- KeyRelease——释放按键。
- Button 或者 ButtonPress——按下鼠标键。
- ButtonRelease——释放鼠标键。
- Enter——移动鼠标光标到组件内(处在组件可见部分之上)。
- Leave——从组件上移开鼠标光标。
- Motion——在某个组件内，将鼠标光标移到另一个点。
- MouseWheel——用户移动鼠标滚轮。
- FocusIn——组件接收键盘焦点。
- FocusOut——组件失去键盘焦点。
- Configure——在开始时显示组件，或者改变它的尺寸、位置或边缘宽度。
- Map——组件可见。
- Unmap——组件不再可见。
- Destroy——删除组件。

提示： 给出事件类型及其修饰符的命令是区分大小写的。要确保使用准确的大小写表达事件类型和它的修饰符，否则命令将出错。

除了它的类型以外，每个事件还包含了一些其他的域。对鼠标事件而言，其中一个域指定了一个组件，另外两个给出了组件内光标的 *x* 和 *y* 坐标。对于<Enter>和<Leave>事件，光标分别进入组件和离开组件。对于<ButtonPress>、<ButtonRelease>和<Motion>事件，组件当前正在光标下方。对于<KeyPress>和<KeyRelease>事件，组件有应用程序的输入焦点(输入焦点的更多细节可参见第 27 章)。

按钮和按键事件也含有一个被称为细节的域，它表明被按下的特殊按钮或按键。对于<ButtonPress>和<ButtonRelease>事件，细节是一个按钮编号，1 通常表示鼠标最左边的按钮。对于<KeyPress>和<KeyRelease>事件，细节是一个键码(按键符号)，它是描述键盘上特殊按键的文本名称。按字母顺序排列的 ASCII 码的键码，如 a 或 A 或 2，正是它自身。另外一些键码的示例，例如，Return 用来表示回车键(在现代键盘上，通常是 Enter 键)，BackSpace 用来表示 Backspace 键，Delete 用来表示 Delete 键，Help 用来表示 Help 键。功能键的键码，如 F1，通常和显示在按键上的名字是一样的。22.6 节包含了一个用来找出键盘上各个键的键码的脚本。

某些键被定义为特别的修饰符键；包括 Shift 键，Control 键，标准 Mac 键盘上的 Command 和 Option 键，再加上别的一些键，如 Meta 和 Alt。刚才列出的事件包含了一个状态域，用来指明当事件发生时，会按下哪个修饰符键和哪个鼠标键。例如，当按下鼠标 1 键或按下 Ctrl+A 时，状态域会触发一个脚本。

除了上述域，事件也包含另外一些域，每个事件的域集都不完全相同。关于每种事件类型的全部有效域可参见参考文档。

22.3 bind 命令概览

bind 命令用于创建、修改、查询和删除绑定。本节将介绍 bind 的基本功能，稍后几节会对这些功能进行更详细的介绍。

使用下述命令会创建绑定：

```
bind .entry <Control-KeyPress-d> {.entry delete insert}
```

命令的第一个参数指定了要应用绑定的窗口的路径名。它也可以是一个组件类名，如 Entry，这时，绑定会应用于该类的所有组件(这样的绑定称为类绑定)，或者它可以是 all，此时，绑定会用于所有组件。您还可以创建自己的符号绑定标记，详情参见 22.8 节。第二个参数指定了一个或多个事件。该示例指定了单个事件，它是在 Control 修饰符键被按下时，按下 d 字符的按键事件。第三个参数可能是任意 Tcl 脚本。示例中的这段脚本调用了.entry 的组件命令，来删除插入光标后的字符。如果当应用程序的输入焦点在.entry 中时，按下 Control+d，这段脚本将被触发。绑定可以触发任意次。它会一直起作用，直到.entry 被删除或者调用一个带空脚本的 bind 来明确删除绑定。

```
bind .entry <Control-KeyPress-d> {}
```

如果在脚本的执行过程中出现了一个错误，则会出现一个背景错误。当一个背景错误在 Tk 脚本中出现，默认的行为是弹出一个对话框，告知用户这个错误。您可以改变这种行为，如 13.6 节所述。

bind 命令也可以检索关于绑定的信息。如果 bind 在没有脚本时由事件序列触发，那它会返回给定事件序列的脚本。

```
    Bind .entry <Control-KeyPress-d>
⇒   .entry delete insert
```

如果使用单个参数调用了 bind，它会返回该窗口或类的全部绑定事件序列。

```
    bind .entry
⇒ <Control-Key-d>
    bind Button
⇒ <ButtonRelease-1> <Button-1> <Leave> <Enter> <Key-space>
```

第一个示例返回了.entry 的绑定序列，第二个示例返回了按钮组件所有类绑定的信息。类绑定由 Tk 的初始脚本创建(Tk 库目录中的文件 button.tcl)，用来生成按钮组件的默认行为。

22.4 事件模式

事件序列由称为事件模式的基本单元组成，Tk 会将应用程序接收的事件流与事件模式相对照。一个事件序列可以包含任意多模式，但大多数序列只有单个模式。

模式最常见的格式是在尖括号中包含一个或多个域，采用下述语法：

```
<modifier-modifier ... -modifier-type-detail>
```

可以使用空格代替短横线来隔开不同的域，大多数域是可选的。*type* 域标识了特定的事件类型，如 KeyPress 或 Enter。常见事件类型的列表可参见 22.2 节。例如，下面的脚本使得，当鼠标指向输入框组件.entry，背景会变成浅绿色，当鼠标离开该组件，背景会重新变为白色。

```
bind .entry <Enter> {.entry config -background lightgreen}
bind .entry <Leave> {.entry config -background white}
```

对于按键和按钮事件而言，事件类型后面可能是细节域，后者指定了一个特定的按钮或键码。如果没有给出细节域，如<KeyPress>，模式会与指定类型的任何事件匹配。如果给出了细节域，如<KeyPress-Escape>或者 ButtonPress-2，模式会只与特定按键或按钮的事件匹配。如果指定了细节，可以忽略 Key 或者 Button 事件类型：<Escape>和<KeyPress-Escape>的意义相同。

表 22.1 事件模式的修饰符键名；用逗号隔开的修饰符键意义相同

Control	Mod1，M1，Command	Button1，B1
Shift	Mod2，M2，Option	Button2，B2
Lock	Mod3，M3	Button3，B3
Alt	Mod4，M4	Button4，B4
Meta，M	Mod5，M5	Button5，B5
Double	Triple	Quadruple

提示：模式<1>和<Button-1>意义相同，和<KeyPress-1>则不同。

事件类型可能置于任意多的修饰符之后，这些修饰符必须来自表 22.1 给出的数值。如果指定了修饰符，模式会只与在给出特定修饰符时出现的事件匹配。例如，模式<Shift-

Control-d>要求当输入 d 时，要同时按住 Shift 键和 Control 键，而<B1-Motion>要求当按下鼠标 1 键时，移动光标。

并非全部修饰符在所有系统中都有效。例如，Command 和 Option 修饰符分别和 Mod1 和 Mod2 意义相同，并对应于针对 Macintosh 的修饰符键。一些非 Macintosh 系统的键盘可能有 Meta 键，该键会对应于这些修饰符，但很多系统可能没有生成 Mod1～Mod5 修饰符的按键。

“修饰符”Double、Triple 和 Quadruple 分别用于指定两次、三次和四次鼠标单击及其他重复性事件。它们和两事件、三事件或者四事件的序列匹配，每个序列都与剩余的模式匹配。例如，<Double-1>与两次单击鼠标 1 键匹配，而<Triple-2>与三次单击鼠标 2 键匹配。

可对模式使用特定的快捷键格式，它指定了用于打印 ASCII 码(如 a 或@)的按键。您可以使用单个字符来指定这些事件的一个模式。例如：

```
bind .entry a { .entry insert insert a }
```

如果在.entry 有键盘焦点时，输入字符 a，那么 a 会在插入光标处插入.entry。这条命令和下面这条命令等价。

```
bind .entry <KeyPress-a> {.entry insert insert a}
```

22.5 事件序列

事件序列由一个或多个可选的事件模式组成，模式之间用空格隔开。例如，序列<Escape>a 包含了两种模式。在 Escape 键被按下且 a 键也被按下时，它会触发。

序列不一定要与连续事件匹配。例如，序列<Escape>a 与这样一个事件序列匹配：在 Escape 上的 KeyPress，一个 Escape 的 KeyRelease，按下 a；在决定这种匹配时，会忽略 Escape 的释放。Tk 通常会忽略输入事件流中相互冲突的事件，除非它们与期望事件的类型相同。因此，如果在 Escape 和 a 之间按下另一个键，序列不会匹配。又如，序列 aB 与按下 a 键，释放 a 键，按下 Shift 键，按下 b 键匹配。这时，会忽略 Shift 键的按下，因为它是一个修饰符键。同样，如果一些 Motion 事件在一行中出现，只有最后一个才会与绑定序列匹配。

22.6 脚本中的置换

处理事件的脚本需要使用事件中的一些域，例如指针的坐标或者按下的特殊键码。Tk 从脚本中含有“%”字符的事件里置换域，以使用这些域，采用的模式与 Tcl 的 format 命令很类似。在处理绑定的脚本前，Tk 会将每个“%”和其后的字符置换为关于事件的信息，以从原始脚本中生成一个新脚本。“%”之后的字符选择一种特定的置换。一共存在大约 30 种被定义的置换，但表 22.2 只列出了最常用的几种。并不是所有的置换对每一种事件类型都有效。关于不同事件类型和它们的含义的有效置换的完整细节可参见参考文档。

作为使用置换的一个示例，下面的脚本执行了一次简单的鼠标追踪。

```
bind all <Enter> {puts "Entering %W at {%x,%y}"}
bind all <Leave> {puts "Leaving %W"}
bind all <Motion> {puts "Pointer at (%x,%y)"}
```

表 22.2 常见的事件细节替换

序 列	替 换
%%	替换为一个%
%b	按下或释放的按钮号。仅对 ButtonPress 和 ButtonRelease 事件有效
%d	以特定格式给出的内建事件细节。对于用 event generate 命令产生的虚拟事件，这可以是提供给 event generate 命令的任意字符串值。更多细节见参考文档
%h	事件的窗口高度值
%t	事件的时间值
%w	事件的窗口宽度值
%x	鼠标指针的 *x* 坐标
%y	鼠标指针的 *y* 坐标
%A	对应于 KeyPress 或 KeyRelease 事件的 Unicode 字符值，如果事件是关于 Shift 这类没有对应 Unicode 的键的事件，则对应一个空字符串
%D	MouseWheel 事件的差值，表示鼠标滚轮滚动过了多少个单位。值的符号表示鼠标滚轮滚动的方向
%K	KeyPress 或 KeyRelease 事件的键码
%W	接收这个事件的组件的完整组件路径名称
%X	鼠标指针相对于根窗口的 *x* 坐标
%Y	鼠标指针相对于根窗口的 *y* 坐标

如果在 wish 中输入这些命令，那么当您在应用程序的窗口上移动指针时，信息会打印出来。或者可以使用下述脚本来决定键盘上按键的键码。

```
bind . <KeyPress> {puts "The keysym is %K"}
focus .
```

如果在 wish 中输入了这两个命令，您所输入的任何键的键码名都会打印出来。试着输入一些常规的键，例如 a、A 和 1，加上特别的键，例如 F1、Return 或者 Shift。

提示： 在 Tk 做“%”置换时，它将脚本作为一个没有任何特定属性的普通字符串来处理。不需要考虑 Tcl 命令的一般引用规则，因此，即使“%”插入到括号中或置于反斜线符号之后，它也会被置换。唯一能阻止“%”置换的方式是使用两个连续的“%”字符。

22.7 解决冲突

可能会有几个绑定与单个事件匹配。例如，假定存在<Button-1>和<Double-Button-1>的绑定，并且 1 键被单击了三次。第一个按钮按下的事件只与<Button-1>绑定匹配，但第

二个和第三个按钮按下的事件却与两个绑定都匹配。处理冲突的方案取决于绑定是如何声明的。如果所有匹配的绑定都使用相同的标记，例如：

```
bind .b <Button-1> ...
bind .b <Double-Button-1> ...
```

这时只有一个绑定触发，且它是所有匹配的绑定中最具体的那个。<Double-Button-1>比<Button-1>更具体，因此它的脚本在第二次和第三次按键时执行。同样，<Escape>a 比<Key-a>更具体，而<Control-Key-d>比<Key-d>更具体，<Key-d>比<KeyPress>更具体。

如果不止一个绑定与一个特殊事件匹配，且绑定有相同的标记，那么会选中最具体的绑定，并处理它的脚本。下列测试会按顺序使用，以决定一些匹配序列中的哪一个更具体。

(1) 指定特定按钮或按键的事件模式比没有这样指定的模式更具体。

(2) 长一些的序列(按匹配的事件数衡量)比短一些的序列更具体。

(3) 如果一种模式指定的修饰符是另外一种模式中修饰符的子集，那么有更多修饰符的模式更具体。

(4) 如果上述测试不能决定最具体的模式，那么选用最近注册的序列。

作为一个示例，考虑只由事件<Button-1>的单个绑定提供的标签组件。

```
label .l -text "Sample 1"
bind .l <Button-1> {
    puts "Left mouse button pressed"
}
```

这个事件将一个<Button-1>事件与一个<Shift-Button-1>事件匹配。在下面这种情况中，对标签给出两个绑定，一个带有 Shift 修饰符，另一个则没有。

```
label .l -text "Sample 2"
bind .l <Button-1> {
    puts "Left mouse button pressed"
}
bind .l <Shift-Button-1> {
    puts "Shift-Left mouse button pressed"
}
```

因为该组件有两个绑定，事件匹配会区分出这两种不同的情况，且更具体的绑定阻止了匹配却没有修饰符的绑定的触发。

22.8　事件绑定层级结构

到目前为止，我们主要关注的是特定窗口的绑定，但事实上，存在一个能处理事件的绑定层级结构。每个窗口都有一些绑定标记，它们也被简称为标记。每个绑定标记可以包含一系列事件绑定。假定存在拥有不同标记的匹配绑定，例如：

```
bind all <Return> ...
bind .b <Any-KeyPress> ...
```

如果按下 Return 键，则这两个绑定都会匹配。不同标记是单独处理的，因此匹配的绑定会为每种标记触发。处理这些标记的顺序由 bindtags 命令定义。

```
bindtags widget ?tagList?
```

tagList 是标记列表，按照它们在事件匹配中被检索的顺序来排列。组件默认按下列顺序给出 4 个标记：组件自身、组件类名、包含组件的顶层窗口和 all。如果 bindtags 命令使用了单个参数，则它会返回指定组件的当前标记列表。所以对上述示例中的组件.b 而言，默认的标记列表如下：

```
    bindtags .b
⇒   .b Button . all
```

这意味着.b 的绑定最先触发，接下来是在按下 Return 键时 all 的绑定。

标记列表可以修改，以改变标记的顺序，或者添加或删除窗口、类甚至用户定义的标记。例如，因为按钮的行为用 Button 类上的绑定定义，则按钮的新行为可以赋给一个新标记，如 SpecialButton。简单地改变一个给定按钮的绑定标记会立刻改变该按钮的行为，例如：

```
bind SpecialButton <Return> { # Some script ... }
bindtags .b {.b SpecialButton .all}
```

有时，需要阻止位于层级结构前段的事件绑定被触发。这时可以用 break 命令或 return -code break 命令。在事件的绑定代码返回一个“中断”状态时，对绑定标记列表的进一步处理将被搁置。例如，将前面的示例改成如下所示：

```
bind all <Return> {puts "all return!"}
bind .b <Return> {puts " .b return!"; break}
```

这会改变行为，使得 all 绑定不被触发，至少不为.b 窗口触发。break 命令可以用在绑定脚本正文中。如果绑定脚本调用了一个 Tcl 过程，该过程必须使用 return -code break 命令来从过程自身返回一个中断条件，以达到同样的效果。

22.9 事件何时被处理

Tk 只在少数被精确定义的时刻才会处理事件。在 Tk 应用程序完成其初始化后，它进入一个事件循环，等待窗口系统事件和其他事件，例如计时器和文件事件。如果一个事件发生了，事件循环会执行 C 或 Tcl 代码来响应该事件。一旦响应完成，控制会回到事件循环，等待下一个事件。几乎所有事件都由顶层事件循环处理。在事件循环响应当前事件时，新事件不被考虑，所以通常不存在一个绑定在另一个绑定的脚本中触发的危险。这种方法应用于所有的事件处理器，包括那些绑定的事件处理器、与组件关联的脚本选项的事件处理器和其他一些要讨论的事件处理器，例如窗口管理器协议处理器。

提示： 一些特别的命令，如 tkwait、vwait 和 update，会明确触发事件循环。如果从一个正在执行的事件处理器中调用它们，则它们会调用事件循环中嵌套的实例，这常常造成不可预知的效果。最好的办法是不从您的事件处理脚本中调用这些命令。在参考文档中应该特别指出了调用事件循环的命令和过程，可以在这些文献中查找相关信息。所有其他的命令无需重新进入事件循环就能立刻完成。

事件处理器总是在全局命名空间中的全局范围中触发，即使事件循环由一个过程中的tkwait、vwait 或 update 命令触发。这意味着应该只使用全局变量，或绑定脚本中完全符合规范的命名空间变量。处理处理器的上下文并不与定义处理器的上下文完全一样。例如，从一个过程内部调用了 bind。当绑定中的脚本最终触发，它并不会读取过程的局部变量；在大多数情况下，该过程会在绑定被调用前返回，故局部变量再也不存在了。

最好的方法是用 Tcl 过程执行一个绑定脚本，特别是在脚本需要使用临时变量来处理事件时。这防止了全局命名空间被临时变量干扰。当临时变量被无意中共享时，这也能防止潜在的漏洞。另一种共享变量的好方法是使用命名空间。(第 10 章详细介绍过命名空间。)命名空间变量可由命名空间定义的过程共享，因此就避开了潜在的无意中交互的问题。绑定脚本调用一个命名空间定义的过程没有任何问题；在调用该过程时，只需使用一个完全限定的名称即可，例如：

```
namespace eval SpecialButton {
    variable last_x
    variable last_y
    proc b1_event {W x y} {
        variable last_x
        variable last_y
        set last_x $x
        set last_y $y
        puts "Button-1 pressed"
        puts "at location $x,$y"
        puts "in window $W"
}

bind .b <Button-1> {SpecialButton::b1_event %W %x %y }
```

使用 Tcl 过程的另外一个优势是性能。Tcl 解释器为了提高性能会为过程块进行字节码编译。因为每次触发绑定时都会进行替换，不可以用字节码方式编译一个绑定脚本。在绑定调用了一个 Tcl 过程时，该过程块可以用字节码方式编译，以提升处理性能。

22.10 命名虚拟事件

Tk 的绑定提供了一种有效的方法，来将窗口系统或用户的输入与应用程序里的任务关联。使用命名虚拟事件可以实现这一点。我们来看看这一切是如何进行的。以下是一个典型应用程序绑定的示例：

```
bind .text <Control-c> {tk_textCopy %W}
```

现在这个绑定可能也为应用程序中的其他组件复制。Windows 的应用程序通常用<Control-c>作为复制的快捷键；然而，在苹果 Macintosh 系统中，该操作通常用<Command-c>来完成。如果只为了修改特定操作的快捷键而不得不修改每一个事件绑定，无疑会很繁琐，且容易产生错误。通过虚拟事件，我们可以简化键盘快捷键的赋值。首先，创建对一个命名虚拟事件的绑定。

```
bind .text <<Copy>> {tk_textCopy %W}
```

您应该对每个支持复制动作的组件或组件类进行这种操作。下一步，使用 Tk 的 event 命令将物理按键绑定和命名虚拟事件联系在一起。

```
event add <<Copy>> <Control-c>
```

这个操作只需要进行一次。定义一个新的快捷键与添加一个新的事件定义一样简单。

```
event add <<Copy>> <Command-c>
```

所以现在的复制操作有两个快捷键：<Control-c>和<Command-c>。前面那个快捷键可以使用 event delete 命令来删除。

```
event delete <<Copy>> <Control-c>
```

当然，在一些情况下，将两个平台的事件序列和虚拟事件相关联会导致不兼容。在这种情况下，可以使用 tk windowingsystem 命令(参见第 29 章)来决定应用程序正在哪一个窗口系统上运行，稍后就只设置适合该平台的虚拟事件定义。例如，将脚本与“单击鼠标右键”绑定的情况。大多数 Windows 和 Unix 操作系统使用三键鼠标，右键被认为是<ButtonPress-3>事件。但是，Macintosh 系统的鼠标往往只有一个按钮，在按下 Control 键时单击该按钮通常被视为单击鼠标右键的等价操作。甚至在那些使用多键鼠标的 Macintosh 系统中，右键被视为一个<ButtonPerss-2>事件。所以可以定义一个名为<RightClick>的虚拟事件，例如：

```
if {[tk windowingsystem] eq "aqua"} {
    # Set up Mac OS X virtual event definitions
    event add <<RightClick>> <Button-2> <Control-Button-1>
} else {
    # Set up Windows and Unix virtual event definitions
        event add <<RightClick>> <Button-3>
}
```

如果虚拟绑定和物理绑定的序列相同，物理绑定优先。所以如果

```
bind .text <Control-C> {puts "copy %W"}
```

和

```
bind .text <<Copy>> {tk_textcopy %W}
event add <<Copy>> <Control-c>
```

同时存在，则会执行 puts 命令，因为物理绑定优先于<<Copy>>虚拟绑定。

Tk 将一些常见的操作预定义为组件绑定的虚拟事件。它们的默认值是适合平台的，因此您的应用程序不大可能需要改变它们。表 22.3 列出了 Tk 中用户动作的预定义虚拟事件。

表 22.3 用户动作的预定义 Tk 虚拟事件

虚拟事件	说 明
<<Clear>>	删除当前选中的组件内容
<<Copy>>	将当前选中的组件内容复制到剪贴板

续表

虚拟事件	说 明
<<Cut>>	将当前选中的组件内容移动到剪贴板
<<Paste>>	把当前选中的组件内容用剪贴板中的内容替换
<<PasteSelection>>	将选中的内容插入到当前鼠标位置(本事件可用于%x 和 %y 替换)
<<PrevWindow>>	切换到前一个窗口
<<Redo>>	重做一个刚被撤销的动作
<<Undo>>	撤销上一个动作

22.11 生成事件

除了可以如前所述管理命名虚拟事件，event 命令还有 generate 选项，能用来生成任何窗口系统事件，并将其传给应用程序中的任意窗口，就像它来自窗口系统。生成事件的能力提供了一种测试应用程序用户界面的方法。event generate 的语法如下：

```
event generate widget event ?option value option value ...?
```

按下按钮、移动鼠标和单击按键都可以用 event generate 命令来模拟。*option value* 对定义了事件的属性，例如鼠标的位置。(支持选项的完整列表可参见参考文档。)例如，下列命令会生成一个鼠标按钮事件：

```
event generate .copy <ButtonPress> -button 1 -x 10 -y 10
```

在这种情况下，鼠标 1 键会按下，它通常是鼠标最左边的按键。此外，该命令还提供了相对于组件左上角的鼠标位置，Tk 将这个位置以及%x 和%y 用于进行事件处理的脚本。在这个示例里，事件被传到组件.copy，但鼠标指针仍停在屏幕的该位置上。event generate 命令也支持一个-warp 选项，如果提供了一个布尔运算真值，该选项也会移动鼠标指针。

您也可以生成虚拟事件。生成的虚拟事件甚至不需要有一个与之关联的物理事件序列。这可视为应用程序中非同步消息传递系统的基础。例如，考虑下列代码：

```
label .display -textvariable msg -width 30 -anchor w
grid .display -sticky ew
bind .display <<Message>> {
    set msg "[clock format %t -format %%T]: %d"
}
event generate .text <<Message>> \
    -data "Hello" -time [clock seconds]
```

在这个示例里，名为.display 的组件接收了一个<<Message>>事件。-data 选项允许您用该事件传递一个任意字符串值，事件绑定可以通过%d 事件替换获得该值；-time 选项允许您设定一个时间戳，它可通过%t 替换获得。在这个示例中，事件处理器只是对接收到的信息进行格式处理，并设置与标签关联的文本变量值。

event generate 命令的另一种用途是将一条命令动作与一个组件的虚拟绑定关联，例如：

```
    button .copy -text Copy -command {
        event generate [focus] <<Copy>>
    }
```

或者

```
    .menu.edit add command -text Copy \
        -command {event generate [focus] <<Copy>>}
```

用这种方法，在按下按钮时，单个按钮或菜单命令可以与拥有键盘焦点的组件绑定在一起。

22.12 逻辑动作

到目前为止讨论的大多数绑定用于在应用程序和用户之间，或是应用程序和窗口系统之间，处理交互作用。命名虚拟事件和绑定也可以在多个组件和应用程序之间传递消息和符号。这一点通过一个示例会得到最好的阐述。我们将使用一个简单的文本编辑器来演示如何在文本组件和一个应用程序工具栏之间使用虚拟事件。

文本编辑器示例包含一个带有复制、剪切和粘贴操作按钮的工具栏。当文本组件里的一部分或全部被选定时，应用程序工具栏里的剪切和复制按钮，和编辑菜单里的剪切和复制菜单条目都应该变为有效的状态。当选择改变时，文本组件会自动生成一个<<Selection>>事件。它允许每个与选择相关的组件对<<Selection>>创建一个绑定，并相应地更新它们的状态。

```
    bind all <<Selection>> {updateToolbar %W}
```

在这个示例里，代码和所有的组件绑定在一起，因此不管选择在何处发生，工具栏都会适当更新。在后面的代码中，工具栏按钮根据文本是否被选中，来处于有效或无效的状态。

updateToolbar 过程检查了组件%W 的选择状态，来判断选择是否为空。它用两个另外的虚拟事件<<MayCopy>>和<<MayCut>>，来将选择状态传给所有需要知道的组件。

```
    proc updateToolbar {w} {
        set class [winfo class $w]
        if {$class eq "Text"} {
            set select_range [$w tag ranges sel]
            if {[llength $select_range] > 0} {
                set state normal
            } else {
                set state disabled
            }
            event generate $w <<MayCopy>> -data $state
            event generate $w <<MayCut>> -data $state
        }
    }
```

这时，您只需将<<MayCopy>>和<<MayCut>>事件的绑定脚本硬编码，以更新相应的菜单条目和工具栏按钮。但是，您可能想构造一个模板化的应用程序，它能用可选插件来扩展。这些插件可能需要被告知复制和剪切何时会发生。此时，充分利用 bind 命令的能力

来扩展已知绑定脚本会非常有用。记住，如果 bind 脚本参数的第一个字符是“+”，则该脚本会附在任何已知绑定脚本之后，而不是覆盖它。因此，我们能让组件在粘贴/剪切通知中注册感兴趣的信息，例如：

```
bind all <<MayCopy>> \
    "+.toolbar.copy configure -state %d"
bind all <<MayCut>> \
    "+.toolbar.cut configure -state %d"
bind all <<MayCopy>> \
    "+.mbar.edit entryconfigure 0 -state %d"
bind all <<MayCut>> \
    "+.mbar.edit entryconfigure 1 -state %d"
```

现在，在<<MayCopy>>或<<MayCut>>事件创建时，代码会分别更新工具栏按钮，来复制和剪切文本。该代码也更新了菜单条目，来复制和剪切文本。

同样，将文本组件和<<Cut>>事件绑定在一起，并使工具栏按钮和菜单条目命令生成一个<<Cut>>事件会将动作与组件关联，而无需窗口之间有直接的了解。注意，文本组件已包含了一个<<Copy>>和<<Cut>>的虚拟事件绑定，因此只需在按下工具栏按钮时生成这个事件。

```
button .toolbar.copy \
    -image copy-image \
    -command {event generate [focus] <<Copy>>}
button .toolbar.cut \
    -image cut-image \
    -command {event generate [focus] <<Cut>>}

.mbar.edit add command -label Copy
    -command {event generate [focus] <<Copy>>}
.mbar.edit add command -label Cut \
    -command {event generate [focus] <<Cut>>}
```

使用这项技术，代码只会产生一个表明剪切或者复制操作的事件，让 Tk 文本组件处理剩下的部分。focus 命令告诉了代码哪个组件会有文本焦点。该组件便是用来复制或剪切文本的组件。像这个示例一样，Tk 的虚拟事件用处很大，特别在您使用预置的用于处理这些事件的 Tcl 代码实体时。

图 22.1 显示了处在初始状态的编辑器示例，其中没有文本被选中。注意此时工具栏按钮是如何失效的。一旦一些文本被选中后，复制和剪切工具栏按钮会变为有效，如图 22.2 所示。

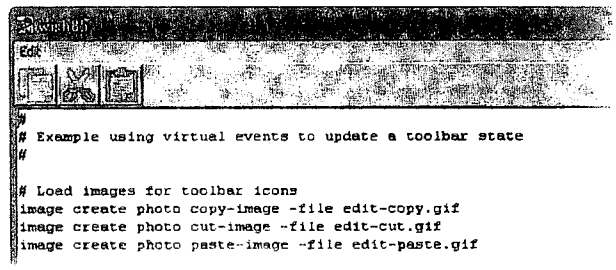


图 22.1 文本被选中前的工具栏按钮。图中的图标来自 Tango Desktop Project (http://tango.freedesktop.org/Tango_Desktop_Project)

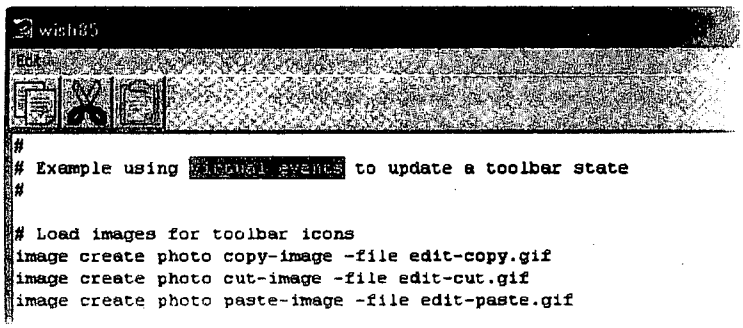


图 22.2　文本被选中后的工具栏按钮。图中的图标来自 Tango Desktop Project (http://tango.freedesktop.org/Tango_Desktop_Project)

在一些文本被选中后，剪切按钮被按下。这时，复制和剪切按钮都会失效，这是因为此时没有文本被选中，而粘贴按钮则变为有效，这是因为此时一些文本存在于剪贴板中，如图 22.3 所示。

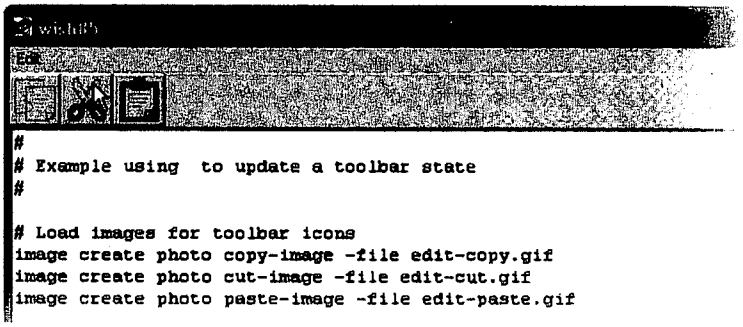


图 22.3　所选中的文本被剪切后的工具栏按钮。图中的图标来自 Tango Desktop Project (http://tango.freedesktop.org/Tango_Desktop_Project)

随着应用程序变得更加复杂，组件交互作用的数目会使用户界面的发展成为一项繁重的任务。当组件命令和状态被组织进逻辑动作时，该交互作用会变为可控。

22.13　绑定的其他用途

本章中描述的绑定机制应用于组件。然而，相似的机制在一些组件中也是内在有效的。例如，画布组件允许绑定与图形条目关联，如长方形或多边形，而文本组件允许绑定与一系列字符关联。这样的绑定可用与事件序列和“%”置换相同的语法创建，但是它们是用组件的组件命令创建的，且会涉及组件的内部对象而不涉及窗口。画布组件的更多信息可参见第 23 章，文本组件的更多信息可参见第 24 章。

第23章 画布组件

本章将介绍 Tk 的画布组件。画布用于显示和控制各种图形对象，例如长方形、直线、图像和文本字符串。画布组件支持强有力的功能，允许为对象做标记，并控制带有该标记的所有对象。例如，可以移动带有某标记的所有条目，或者为它们重新着色。您也可以为标记创建事件绑定，这样在光标指向画布中带标记的长方形，或者在组件的字符串上单击鼠标时，Tcl 脚本会触发。这种标记和绑定的机制使“激活”文本和图形变得简单，这样它们就会响应鼠标动作。本章将对画布做简单的介绍，并举出一些示例。但是，不会介绍每一个细节，您应该查阅参考文档来了解细节。

23.1 画布基础：条目和类型

画布是一个显示二维绘图平面的组件，您可以在该平面上放置各种类型的条目。Tk 现在支持下述类型。

- rectangle——一个轮廓，或者是被填充的区域；您可以指定轮廓线的颜色和宽度，填充的颜色和点画模式。
- oval——圆形和椭圆形，属性和长方形相同。
- line——由一条或多条相互连接的线段组成，有宽度、颜色、点画模式和其他一些属性，例如，在两条线段的结合处是倒角还是圆弧，在线段的末端是否画箭头符号。
- 贝兹曲线(Bézier curve)——带有-smooth 属性的 line 条目，该参数确定了如何画 Bézier 三次样条，而不是直线段。
- polygon——由三个或更多的点组成，含有填充颜色、点画模式等属性。和 line 条目一样，可以使用-smooth 参数来指定多边形的轮廓线应该是一个贝兹曲线，而不是一系列直线段。
- arc——一个弧形楔形物或一段圆弧。您可以指定一些属性，例如圆弧的张角、轮廓线宽度、轮廓线颜色、填充颜色和填充点画线。
- text——由采用单一字体的一行或多行文字组成。您可以选择一种字体、颜色、点画模式、对齐模式和线长。
- bitmap——由位图名称、背景颜色和前景颜色组成。
- image——使用 Tk 的 image create 命令或任意 Tk 图像扩展创建的相片或位图。
- window——任何种类的嵌入式组件，这时画布将充当组件的几何管理器。画布参

考文档和命令语法用术语“窗口”来指代这些嵌入式组件。

要创建画布组件，使用 canvas 命令。

```
canvas widgetName ?option value ...?
```

画布组件支持大多数标准组件选项，例如-background 用来设置背景颜色，-borderwidth 用来设置组件的边框的宽度。您也可以使用任何支持的 Tk 屏幕距离来设置画布的-height 和-width，如 18.2.2 节所述，虽然显示画布的几何管理器或许会覆盖掉这个要求的尺寸(参见第 21 章)。

create 子命令在画布上创建新条目。例如，以下脚本创建名为.c 的画布，并在该画布上创建一个新的长方形条目。

```
canvas .c
grid .c -sticky nsew
.c create rectangle 1c 2.5c 4c 4.75c -width 2m \
    -outline blue -fill yellow
```

在 create 动作后的第一个参数给出了该条目的类型。在类型后面的是一对 x 和 y 坐标。坐标是浮点数，单位为任意 Tk 支持的屏幕距离，如 18.2.2 节所述。在创建画布组件时，它的左上角为“0,0”原点，坐标会向下和向右增加。x 和 y 坐标可能由单独的参数给出，也可能由类型之后的单个列表参数给出。

对长方形而言，必须恰好有四个坐标，给出长方形四个对角的位置。椭圆和弧线需要四个坐标，来描述外接长方形四个对角的位置。直线接受两个或多个点的坐标，来描述相互连接的线段。多边形接受三个或多个点的坐标，来描述顶点；Tk 自动将最后一点与指定的第一点相连。所有其他的类型要求条目的某个定位点的坐标。那些类型的-anchor 选项决定了放在定位点的条目的位置。例如，-anchor nw 将条目的左上角置于定位点。

在坐标之后可以是指定配置选项的参数对。配置选项采用与组件相同的自由格式类型指定。所有未指定的选项都采用默认值。在前面的示例中，轮廓线的宽度和颜色及填充颜色都指定了，但没有指定填充的点画模式(默认为完全填充)。所有类型支持的所有选项的详情可参见参考文档。

画布上的条目有一个堆栈顺序(有时候称为显示列表)。在堆栈顺序低端(先出现在列表中)的条目会被堆栈顺序高端(后出现在列表中)的条目遮住。新创建的条目会出现在列表的末尾，在画布中所有其他条目之上。后文将要提到，您也可以在创建条目后提升或降低它们的堆栈顺序。

提示： 窗口条目(嵌入式组件)是堆栈顺序规则的一个例外。下方的窗口系统要求它们总是画在所有其他条目之上，它们绝不会被其他的画布条目遮住。如果嵌入画布的窗口条目相互重叠，那么可用 Tk 的 raise 和 lower 命令来改变它们的相对堆栈顺序，详情参见第 29 章。

以下脚本使用直线和文本条目来创建一把简单直尺。

```
# ruler: draw a ruler on a canvas
canvas .c -width 12c -height 1.5c
pack .c
.c create line 1c 0.5c 1c 1c 11c 1c 11c 0.5c
```

```
for {set i 0} {$i < 10} {incr i} {
    set x [expr $i+1]
    .c create line ${x}c 1c ${x}c 0.6c
    .c create line $x.25c 1c $x.25c 0.8c
    .c create line $x.5c 1c $x.5c 0.7c
    .c create line $x.75c 1c $x.75c 0.8c
    .c create text $x.15c .75c -text $i -anchor sw
}
```

图 23.1 展示了这段脚本生成的画布。长度的每一厘米用 4 条高度不同的刻度线隔开，同时还有一个显示厘米数的文本条目。没有指定直线条目的任何选项，因为对该示例而言，默认值已经很精细了。文本条目的选项-anchor sw 使它们的左下角置于指定的坐标上。

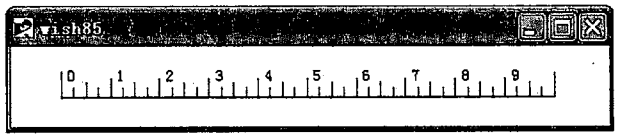


图 23.1　用 ruler 脚本生成的画布

23.2　控制带标识符和标记的条目

特殊画布组件上的每个条目都有一个独特的整数标识符。条目的标识符由 create 子命令返回，且它可以用于其他的组件命令，以引用该条目。例如，可能调用以下命令：

```
set circle [.c create oval 1c 1c 2c 2c -fill black \
    -outline {}]
⇒ 12
```

来创建一个黑色实心圆圈，并将它的标识符(在这个示例中是 12)保存在变量 circle 中。稍后可以使用以下命令来删除该条目。

```
.c delete $circle
```

画布的组件命令提供了一些控制条目的子命令。

- 可以 move 和 scale 条目(但在 Tk 8.5 中不能旋转条目)。
- 可以 raise 和 lower 它们的相对位置(也就是，改变它们的 *z* 轴次序)。
- 可以查询或改变条目的坐标(coords)和配置选项(itemcget 和 itemconfigure)。
- 可以 find 离给定点最近的条目，或者给定长方形内的所有条目，或者带有给定标记的所有条目。
- 对于文本条目，可以 insert 和删除(dchars)文本，显示并控制插入光标(icursor)，设置 focus，并 select 一些字符。
- 可以将它们与事件 bind 在一起，例如<ButtonPress>或<KeyPress>事件。

所有子命令和它们用途的详细介绍可参见画布参考文档。

画布也提供了另一种引用条目的方法，称为标记。标记只是一个与条目关联的文本字符串。标记可能有除整数外的各种格式。单个条目可能有任意多标记，单个标记可能用于任意多条目。一旦一个条目被贴上标记，可以用它的任一标记来引用它。

```
.c delete circle
```

在这个示例里，所有带 circle 标记的条目都被删除了。

标记在画布中有三个作用。第一，它们用于为条目赋予人类能读懂的名称，这样您不需要将条目标识符保存在变量中。第二，它们提供了一种组合机制，这样您可以同时控制关联条目。例如，考虑下述命令：

```
.c itemconfigure circle -fill red
.c move circle 0 1c
```

第一条命令将所有带 circle 标记的条目的填充颜色设为红色，第二条命令将所有圆形条目下移 1 厘米。通常，可以在能用条目标识符的地方使用标记名。您还可以使用简单的逻辑表达式标记，如 circle&&element，完整的细节可参见参考文档。

标签的第三种用途是定义行为。正如后文将介绍的那样，绑定可以与标记关联。这就允许条目组响应用户事件。您也可以将多个标记应用于一个条目，来组合若干绑定行为，或者添加或删除标记，来动态地改变条目行为。

可以将一个或多个标记赋予该条目，方法是在创建条目时提供标记列表作为-tags 选项的值，如下面这条命令：

```
.c create oval 1c 1c 2c 2c -fill black -outline {} \
    -tags {circle element}
```

这条命令将 circle 和 element 作为与新条目关联的标记。您也可以用 itemconfigure 子命令将另一些标记赋予一个已有条目，这时需要改变与-tags 选项关联的值。

画布组件命令也提供可动态添加或删除条目标记的命令。dtag 子命令可以删除标记，使之无效，例如：

```
# Remove the tag "circle" from item 12
.c dtag 12 circle

# Remove the tag "highlight" from all items with
# the tag "element"
.c dtag element highlight

# Remove the tag "element" from all items that use it
.c dtag element
```

您还可以用 addtag 子命令为一个或多个条目添加标记。

```
canvas addtag tagName searchSpec
```

您需要指定要添加的标记名和搜索条件，来标识应该做标记的条目。表 23.1 给出了搜索条件的格式。您也可以使用与画布的 find 子命令相同格式的搜索条件，find 命令会返回匹配的所有条目的标识符。

画布组件会自动控制另外两个预定义标记，您或者不能明确地添加或者删除这些标记。标记 all 通常引用画布上所有的条目。标签 current 会自动引用顶层条目，这些条目的绘图区域覆盖了鼠标光标的位置。如果鼠标不在画布组件内，或不在条目上方，则条目都不会有 current 标记。

表 23.1 画布搜索条件

条 件	功 能
above *tagOrID*	选择由标记名称或 ID 指定的条目的上一层条目
All	选择画布上的所有条目
below *tagOrID*	选择由标记名称或 ID 指定的条目的下一层条目
closest *x y* ?*offset*? ?*start*?	选择离给定位置最近的条目。距给定位置距离小于 *offset* 的条目都认为与该位置重叠。如果多个条目与该点距离相同，或与该点重叠，则选中最上层的一个。*start* 标记名或 ID 用于查找在 *start* 条目之后的最近的条目
enclosed *x1 y1 x2 y2*	选择给定的矩形中包含的条目
overlapping *x1 y1 x2 y2*	选择与给定矩形重叠的条目，以及该矩形中包含的条目
withtag *tagOrID*	选择给定标记或 ID 指定的所有条目

23.3 绑定

在创建画布组件时，可以使用 bind 命令，用常规方法来创建绑定，这样的绑定会作为一个整体应用于组件。(Tk 事件和使用 bind 命令的完整讨论可参见第 22 章。)另外，可以用 bind 子命令为画布内的单独条目创建绑定。该命令格式如下：

```
.c bind itemOrTag sequence script
```

itemOrTag 参数会给出单个条目的标识符，这时绑定作用于该条目，或者它会给出一个标记名，这时它会作用于所有带这个标记的条目。*sequence* 和 *script* 指定了一个事件序列和脚本，就像 bind 命令一样。事件序列可能只使用 Enter 和 Leave 事件(在鼠标移入条目或离开条目时触发)，鼠标运动事件，按钮按下和释放，按键按下和释放，或者虚拟事件。

多个绑定可能与一个特别的事件匹配，例如，如果您已经创建了多个标记的绑定，并在稍后将这些标记的一个组合用在一个特殊条目上。此时，所有匹配的绑定都会被调用，开始是 all 标记上的任意匹配绑定，然后是每个条目的标记的任意匹配绑定(按顺序)，再后面是与该条目的 ID 关联的任意匹配绑定。如果单个标记有多个匹配的绑定，那么只会调用最具体的绑定。绑定脚本中的 continue 命令会终止该脚本，break 命令在终止该脚本的同时，还会跳过这个事件所有剩余的脚本，正如 bind 命令那样。

提示： 如果用 bind 命令创建画布窗口的绑定，则在创建的画布条目绑定外，该绑定会用 bind 子命令来调用。这种条目的绑定会在组件的整体绑定之前被调用。

作为绑定的一个示例，下述脚本用于在画布上交互式创建球-棍图形。

```
# graph: simple interactive graph editor

canvas .c
pack .c

# Create a node (circle) at the given x,y position
proc mkNode {x y} {
```

```
    global nodeX nodeY edgeFirst edgeSecond
    set new [.c create oval [expr {$x-10}] [expr {$y-10}] \
            [expr {$x+10}] [expr {$y+10}] -outline black \
            -fill white -tags node]

    # Stores position of the node in two global arrays
    set nodeX($new) $x
    set nodeY($new) $y

    # Creates empty lists for edges in two global arrays
    set edgeFirst($new) {}
    set edgeSecond($new) {}
}

# Draw an edge (line) between two nodes (circles)
proc mkEdge {first second} {
    global nodeX nodeY edgeFirst edgeSecond
    set edge [.c create line $nodeX($first) $nodeY($first) \
        $nodeX($second) $nodeY($second)]
    .c lower $edge

    # Stores the edge ID in lists held in global arrays
    lappend edgeFirst($first) $edge
    lappend edgeSecond($second) $edge
}

# Draw a node at mouse position every time user clicks
# left mouse button
bind .c <ButtonPress-1> {mkNode %x %y}
# Highlight current item by changing its fill color
.c bind node <Enter> {
    .c itemconfigure current -fill black
}

# Remove highlight color
.c bind node <Leave> {
    .c itemconfigure current -fill white
}

# Typing "1" key stores item as first node for a line
bind .c  <KeyPress-1> {
    set firstNode [.c find withtag current]
}

# Typing "2" key identifies end of the new line
bind .c <KeyPress-2> {
    set curNode [.c find withtag current]
    if {($firstNode != "") && ($curNode != "")} {
        mkEdge $firstNode $curNode
    }
}

# Assign keyboard focus to the canvas
focus .c
```

如果您 source 这段脚本到 wish 中，可以在画布上单击鼠标 1 键，来创建节点。您可以将指针指向某个节点，输入 1，然后指向另一个节点，输入 2，来创建边。图 23.2 就是该脚本创建的图形的一个示例。

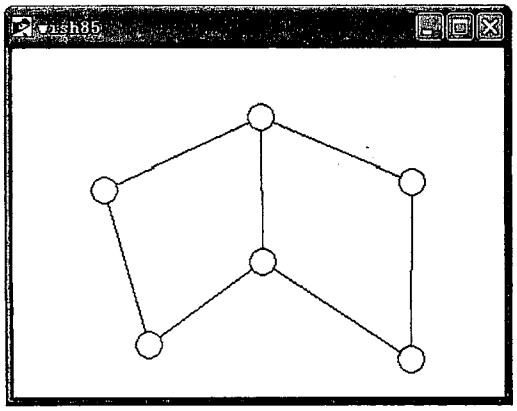


图 23.2　使用 graph 脚本交互式创建的图形

这段脚本包含了两个过程(创建节点和边)以及 5 个绑定。同时也需要 focus 命令明确地将键盘焦点赋给画布组件。该脚本使用 6 个全局变量来存储图形信息。

- nodeX 和 nodeY 是关联数组，其中索引是一个节点的条目标识符，值为节点中心的 *x* 或者 *y* 坐标。该信息可能在需要时从画布中提取出来，但此时将其保存在 Tcl 数组中更简单。
- edgeFirst 和 edgeSecond 是关联数组，其中索引是一个节点的标识符，值为连接各节点的边的条目标识符列表。edgeFirst 储存了以节点为起点的所有边的标识符，而 edgeSecond 指定以节点为终点的所有边。这条信息会用于交互式地拖曳节点。
- firstNode 存储了当输入 1 时鼠标指针下方的节点的标识符，如果输入 1 时指针不在某个节点上，则会存储一个空字符串。
- curNode 存储了指针下的节点的标识符，如果指针未在节点上方，则存储一个空字符串。

mkNode 和 mkEdge 过程只在画布中创建新条目，并在全局变量里记录了它们的信息。mkEdge 里的 lower 子命令会使边置于画布显示列表底部，这样节点就会显示在它们之上。

使用 bind 命令设置整个画布的第一个绑定；在按下鼠标 1 键时，会在指针的位置创建一个新节点。Enter 和 Leave 绑定应用于画布中的所有节点；当鼠标滑过节点上方时，它们会改变节点的颜色。最后两个绑定用来创建边；当在画布中输入 1 或 2 时，它们会触发。两个绑定都使用了特别的标记 current，该标记由 Tk 管理。1 的绑定会调用下述命令：

```
.c find withtag current
```

它返回了带 currentt 标记的条目的标识符(此时，结果要么是一个单元素列表，要么是空列表)，并且将结果保存在 firstNode 中。2 的绑定获取当前条目，也创建了一条新边。

下面这段脚本扩展了 graph，以允许在按住 Control 键时，用鼠标 1 键拖曳节点。

```
proc moveNode {node xDist yDist} {
    global nodeX nodeY edgeFirst edgeSecond
    .c move $node $xDist $yDist
    incr nodeX($node) $xDist
```

```
    incr nodeY($node) $yDist
    foreach edge $edgeFirst($node) {
            .c coords $edge $nodeX ($node) $nodeY($node) \
                [lindex [.c coords $edge] 2] \
                [lindex [.c coords $edge] 3]
    }
    foreach edge $edgeSecond($node) {
        .c coords $edge [lindex [.c coords $edge] 0] \
            [lindex [.c coords $edge] 1] \
            $nodeX($node) $nodeY($node)
    }
}

.c bind node <Control-ButtonPress-1> {
    set curX %x
    set curY %y
}
.c bind node <Control-B1-Motion> {
    moveNode [.c find withtag current] [expr {%x-$curX}] \
        [expr {%y-$curY}]
    set curX %x
    set curY %y
}
# Do nothing on the canvas when receiving these events. This
# binding exists to prevent the "best match" algorithm from
# matching the <ButtonPress-1> canvas binding when all we
# want to trigger is the item binding.

bind .c <Control-ButtonPress-1> { }
```

moveNode 过程负责移动一个节点：它使用 move 动作来移动节点条目，然后更新所有以该节点为终点的边。moveNode 用 coords 动作来读取每条边的当前坐标，并替换头两个或者末两个坐标，以反映节点的新位置。

两个新绑定应用于所有带有标记 node 的条目。如果在按住 Control 键时按下鼠标 1 键，指针坐标会存储到变量 curX 和 curY 里。如果在按住 Control 键时按下鼠标 1 键并拖曳，则当前条目会被鼠标带动，且新的指针位置会被记下来。

注意，<Control-ButtonPress>事件有另外一个组件绑定，它提供了一个不进行任何操作的脚本。如果忽略这个绑定，在处理完画布条目绑定后，画布组件会给出一个<Control-ButtonPress-1>事件。如果没有一个精确的绑定匹配，Tk 可能会寻找最好的匹配，找出并执行画布上的<ButtonPress-1>绑定。结果，每次移动一个节点时，也同时创建一个新节点，而这种行为并不是您所期望的。

23.4 画布滚动

在 Tk 组件的标准模式中，画布支持垂直和水平滚动(参见 18.9 节)。但是，如果滚动了一个画布，画布窗口中的坐标将不会和画布逻辑平面上的坐标相等。为处理这样的情况，画布提供了另外的组件命令动作，来由屏幕转换为画布坐标。

因为画布可以视为一个至少有 32 000×32 000 像素大小的虚拟画板，滚动只有在定义了滚动区域后才有效。这限制了滚动条用作画布可见区域的范围。滚动区域通过设定画布

的-scrollregion 选项来定义。选项值是用来分别定义区域的左、上、右、下边缘的 4 个数字。画布组件用这些边缘来设置滚动条的区域(如图 23.3 所示)。

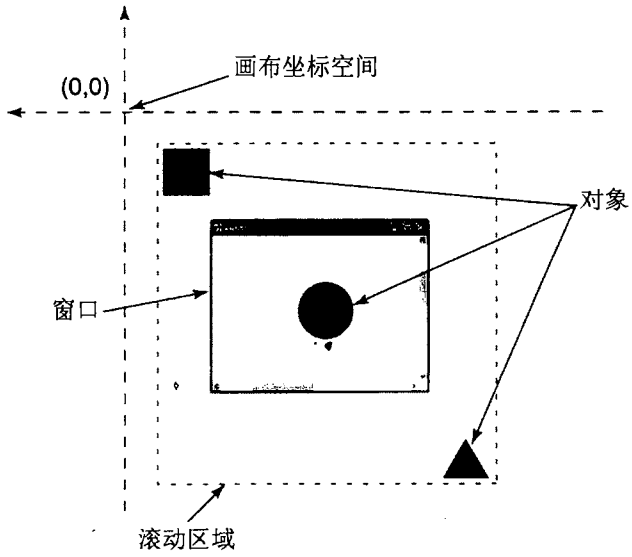


图 23.3　画布滚动区域

设置滚动区域最常见的办法是使用画布的 bbox 子命令，来得到当前画布上所有条目的边缘区域。

```
.c configure -scrollregion [.c bbox all]
```

提示： 在对画布上的条目进行创建、删除、移动、调整大小或者修改操作时，最终都会调整画布上所有条目的边缘区域。

现在画布可以滚动，来看到虚拟画布的不同区域，将鼠标的 x 和 y 坐标转换为画布坐标非常必要，这样可以识别当前视图的画布中的位置。鼠标的 x 和 y 坐标是相对于窗口的左上角给出的。如果滚动了画布，则窗口的左上角将不再处在画布的“0，0”坐标。画布组件的 canvasx 和 canvasy 子命令将相对于窗口的坐标转换为基于当前滚动位置的画布坐标。

例如，要使 23.3 节中给出的图形应用程序的滚动有效，可能需要调整所有的鼠标事件绑定脚本，来将相对窗口的鼠标坐标转换为画布坐标。

```
bind .c <ButtonPress-1> {
      mkNode [.c canvasx %x] [.c canvasy %y]
}
```

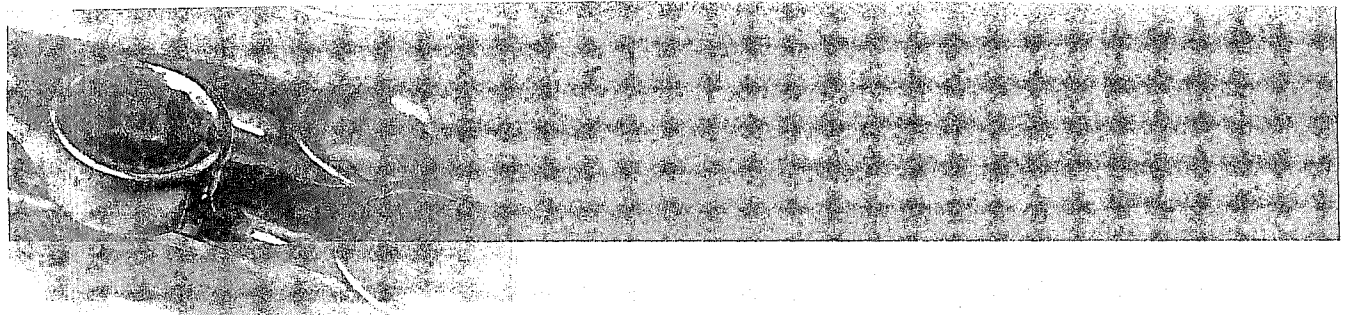
23.5　生成 Postscript

画布可以生成内容的 Encapsulated PostScript 描述，来打印并将它们插入其他文档。使用 postscript 命令可以输出画布组件的内容到 PostScript，例如：

```
.c postscript -file canvas.eps
```

这条命令将画布组件.c 的内容写入 canvas.eps 文件。如果没有提供-file 选项，则 PostScript 数据会成为命令的返回值。

postscript 子命令含有一些配置 PostScript 结果的选项。更多信息可参见画布参考文档。http://pdf4tcl.berlios.de/中的 pdf4tcl 扩展用于将画布组件的内容输出到 PDF 文件。许多用户发现使用 PDF 格式比 Encapsulated PostScript 格式更简单。



第24章 文本组件

本章介绍可以显示一行或者多行文字的 Tk 文本组件。该组件也可以显示嵌入其中的图像和组件。Tk 文本组件用于显示文本，提供编辑文本的功能，甚至为交互式使用 HTML 以及其他标记文本提供接口。

24.1 文本组件的基本原理

使用 text 命令创建文本组件。

```
text widgetName ?option value ... ?
```

该文本组件支持很多常见的组件选项，如-background、-foreground 和-font，这些选项决定显示文本的默认设置；其他文本组件特有的选项，如-tabs 和-wrap，会提供制表位和换行设置。在 24.4 节中将会看到，在创建和应用标记后，所有的这些设置会由于特殊字符而被覆盖。该文本组件也为文本行的高度和字符的宽度提供-height 和-width 选项，当然，用于显示画布的几何管理器也许会覆盖要求的大小(参见第 21 章)。如 24.8 节所述，-undo 选项用于对启动或关闭面向撤销操作的功能提供支持。文本组件同时也支持垂直和水平滚动采用 Tk 组件标准方式(参见 18.9 节)。支持选项的完整描述可参考文本组件参考文档。

操控文本内容的基本文本组件子命令如下。

- *widget* delete *index1 ?index2 ...?*
  删除文本中指定范围内的字符(从 *index1* 一直到 *index2* 但不包括 *index2* 的字符)。如果只有 *index1* 被指定，那么将删去单个字符串。可以指定多个范围。
- *widget* get ?-displaychars ? ?--? *index1 ?index2 ... ?*
  返回文本中指定范围内的字符(从 *index1* 一直到 *index2* 但不包括 *index2* 的字符)。如果只有 *index*1 被指定，那么将返回单个字符串。可以指定多个范围。由选项-elide 隐藏的字符在没有指定-displaychars 时也会返回。
- *widget* insert *index chars ?tagList chars tagList ...?*
  在 *index* 的字符前插入 *chars* 字符串，可以选择将列于 *tagList* 中的标记应用于字符。可以插入多个字符串和标记。
- *widget* replace *index1 index2 chars ?tagList chars tagList ... ?*
  删除文本中指定范围内的字符(从 *index1* 一直到 *index2* 但不包括 *index2* 的字符)，用 *chars* 字符串取代它们，可以选择将列于 *tagList* 中的标记应用于字符。可以插入多个字符串和标记。

包含在文本组件中的每一行文字都必须以换行符结束。在读取文本文件并在文本组件中显示文本文件内容时必须注意这一点。根据应用于文本的-wrap 设置，单个逻辑行可能在文本组件中以可见方式断行，或可能按显示的需要进行文本截断操作，并需要水平滚动以便看到不能显示的部分。

举个例子，下面的脚本创建了一个带滚动条的文本组件，并将一个文件读入文本组件。

```
# text: read a file into a text widget

text .text -relief raised -bd 2 \
    -yscrollcommand {.scroll set}
scrollbar .scroll -command {.text yview}
grid .text -row 0 -column 0 -sticky nsew
grid .scroll -row 0 -column 1 -sticky ns

proc loadFile {file} {
    .text delete 1.0 end
    set f [open $file]
    .text insert end [read $f]
    close $f
}
loadFile $tk_library/demos/README
```

脚本的第一部分创建了一个文本组件和一个滚动条，在主窗口中并排隔开，然后给文本和滚动条建立关联。脚本中接下来的部分定义并调用了过程 loadFile，后者用来打开一个文件，读入文件内容，并将其插入文本组件的末尾。在该例中，loadFile 中的.text delete 命令(删除所有组件中的所有文字)不是必需的，因为新组件本来就为空。然而，之后它允许再次调用 loadFile，以用一个新文件替换组件中的内容。在脚本被执行后，屏幕应如图 24.1 所示(假设您的系统有标准 Tk 组件演示——Tk 标准发行版中的一部分)。

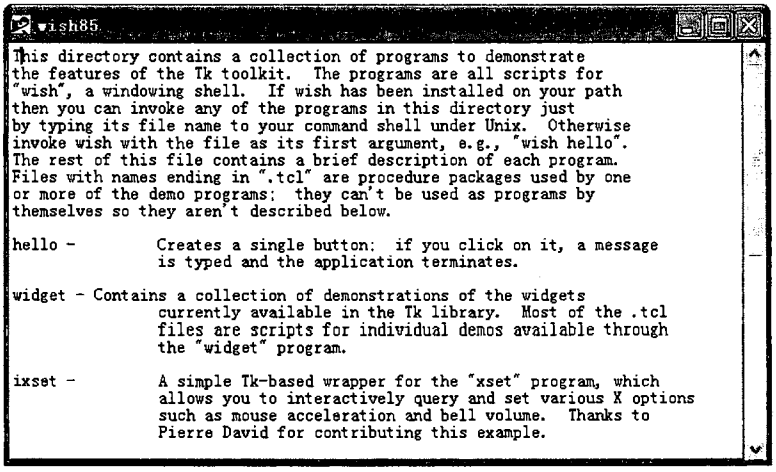


图 24.1 文本组件和关联的滚动条

对文本组件的默认绑定允许您以多种方式处理文本。

- 可以用滚动条滚动文本，或者通过鼠标 2 键来浏览。
- 可以编辑文本。例如，单击鼠标 1 键来设置插入光标，然后输入新的字符。

- 可以通过拖动鼠标 1 键来选择信息，然后将选中的信息复制到其他应用程序。

默认绑定的完整描述可参见文本组件参考文档。

24.2 文本索引与记号

许多文本组件命令需要您在文本中指定特定位置。例如，来自 24.1 节的脚本中的 insert 操作将 end 作为从文件中读取的文字插入的位置。文本中的位置说明符称为索引，它可以有多种形式。最简单的索引形式由一个点分开两个数字来表示，如 2.3。第一个数字为行号，第二个数字为该行内的字符索引。一行中的所有字符是从 0 开始编号的，就像 Tcl 中通常的情况那样。然而，行号从 1 开始计数，以便与其他大多数操作文本文件的程序兼容。如果字符索引是字符串 end，如 5.end，那么它代表终止该行的换行符。索引 end 指文件的末尾，而@*x*,*y* 形式的索引中的数字 *x* 和 *y* 指在窗口中最靠近 *x*, *y* 坐标处像素的字符。

您也可以对文本中表征位置的符号设定名称；这些符号名称称为记号。例如，命令

```
.text mark set first 2.3
```

设置了一个名为 first 的记号，指向第二行第三个字符与其前面字符的间隔。今后您就能用 first 而不是 2.3 来引用记号后的字符。即使在文本中执行了字符增加或者删除操作，记号依然继续指向同样的逻辑位置。例如，删除了第二行的第一个字符，first 就成了索引 2.2 的同义词而不是原来的 2.3。

每个记号都有一个“重心”，不是 left 就是 right。重心指定了当文字按记号插入时，记号会发生什么改动。在重心为左时，该记号看似附在了字符的左侧，而且它将一直存在于在记号位置处添加的字符的左侧。如果重心为右(默认情况)，在记号位置新插入的文字就会出现在记号的左侧(因此记号依旧在最右边)。您可以用 mark gravity 子命令来查询或设置记号的重心，例如，在下例中，将记号 first 的重心设为 left。

```
.text make gravity first left
```

在文本组件中，两个记号有特别的意义。insert 记号识别插入光标的位置；如果修改 insert，Tk 会在新的位置显示插入光标。第二个特殊记号是 current，Tk 会不停更新，来识别鼠标指针下的字符。

索引也能接受更多复杂的格式，这些格式由基本词加修饰符组成。例如，考虑下述命令：

```
.text delete insert  "insert + 2 chars"
```

delete 命令的两个索引都使用 insert 作为基本词，但第二个索引还有一个修饰符+ 2 chars，表示提前两个字符。因此，该命令将删除插入光标右侧的两个字符。还有一些其他使用修饰符的示例，如 first lineend，它在含有标记 first 的行的行末指向新的一行，又如 insert wordstart——指向包含插入光标的单词的第一个字符。

24.3 搜索与替换

文本组件支持 search 子命令和 replace 子命令。search 子命令通过检索整篇文档寻找完全匹配的字符串或者与正则表达式模式匹配的字符串。该命令可以返回给定的起始点之后的下一个匹配的对象，或者返回所有匹配位置。replace 子命令则将删除和插入的操作结合成单个命令。

下面的过程是用记号和其他索引来为文本组件提供一般性的搜索功能。

```
proc forAllMatches {w pattern script} {
    set countList [list]
    set startList [$w search -all -regexp \
        -count countList $pattern 1.0 end]
    foreach first $startList count $countList {
        $w mark set first $first
        $w mark set last [$w index  "$first + $count chars" ]
        uplevel 1 $script
    }
}
```

forAllMatches 过程用了三个参数：文本组件名、正则表达式模式和脚本。它使用文本组件的 search 子命令为所有匹配模式的文本找出开始索引。-count 选项为变量 countList 中的每一个匹配对象返回字符的个数。对每个匹配，forAllMatches 设置记号 first 和 last 分别表示范围的起始和末端，然后调用脚本。例如，下述脚本打印出文本中所有“Tk”实例的位置。

```
    forAllMatches .text Tk {
        puts "[.text index first] --> [.text index last]"
    }
⇒   2.20 --> 2.22
    16.29 --> 16.31
    20.18 --> 20.20
    34.47 --> 34.49
```

在每个匹配范围中，为 first 和 last 记号调用 index 子命令；该命令返回与记号对应的数值索引，然后由标准输出打印。作为另外一个示例，您可以用下述脚本清除文本中多余的单词 the。

```
forAllMatches .text  "the the"   {
    .text delete first  first + 4 chars
}
```

在此脚本中，对每个范围采取的操作是删除其前 4 个字符，这样就清除多余的 the(该案例只有在两个 the 都在同一行时才起作用)。

由于文本组件索引看起来就像是真正的数字，于是就有将它们用作真正数字的趋势，但这可能会导致意想不到的结果。例如，在用一个简单的表达式来比较两个索引时，得不出正确结果。

```
.t mark x 3.37
set x [.t index x]
set mark_insert [.t index insert]
```

```
if {$mark_x > $mark_insert} {
    puts  "Insert is before mark x"
} else {
    puts  "Insert is after mark x"
}
```

在此示例中，如果插入记号定位在索引 3.7，代码便会错误报告插入记号是在 x 记号之后。

文本组件提供 compare 子命令来进行索引的相关比较。

```
widget compare index1 op index2
```

其中，将由 *index1* 和 *index2* 给定的索引通过由 op 给定的相关运算符进行比较，若关系满足则返回 1，不满足则返回 0。op 必须是操作符<、<=、==、>=、>或!=之一。以下是先前的示例中正确的比较索引数值的方式。

```
if {[.t compare x > insert]} {
    puts  "Insert is before mark x"
} else {
    puts  "Insert is after mark x"
}
```

24.4 文本标记

文本标记提供了一种类似于画布的标记机制，但区别在于，文本标记作用于字符而不是图像。文本组件中标记提供三种功能。第一，它们可以用于调整字符的排版格式，如前景和背景的颜色，字体，间隔和左右页边距。第二，它们提供了一种管理选择的方式。第三，它们提供事件绑定功能，能将纯文本转换为活动的控件。

标记的名称可以为任意字符串值，也能应用于文本中的任何范围。单个字符也可能有多个标记，而单个标记可能会与多个字符关联。例如，命令

```
.text tag add theWholeEnchilada 1.0 1.end
```

将标记 theWholeEnchilada 应用到文本的第一行；命令

```
.text tag remove wrd  "insert wordstart" "insert wordend"
```

将标记 wrd 从插入光标附近的语句中的所有字符中移除；而命令

```
    .text tag ranges hot
⇒ 1.0 1.3 1.8 1.13
```

为每个使用了 hot 标记的字符域的开始和末尾，返回一个索引列表。在先前的示例里，标记 hot 在索引 1.0 到 1.2 及 1.8 到 1.12 的范围中出现过。

我们用 24.3 节中的 forAllMatches 的过程来演示如何使用标记来控制文本格式。下述脚本改变了所有“Tk”的实例，采用了更大的字体，更暗的背景色和浮雕模式。

```
forAllMatches .text Tk {
    .text tag add big first last
}
.text tag configure big -background  "cornflower blue"   \
    -font {Helvetica 24} \
```

```
-borderwidth 2
-relief raised
```

结果如图 24.2 所示。

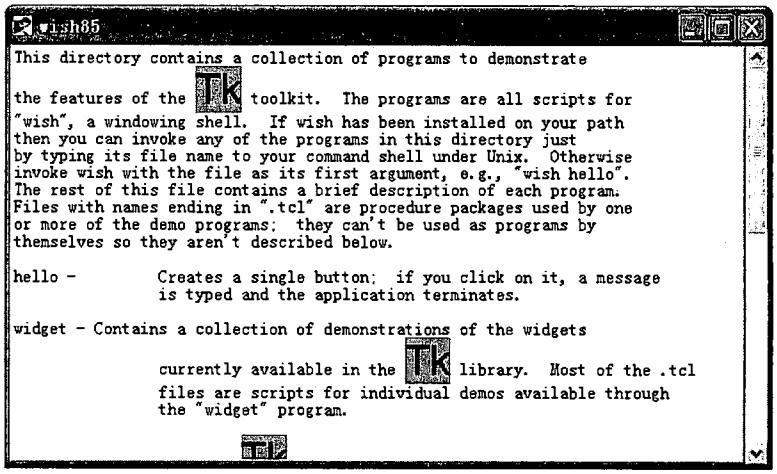


图 24.2 在文本组件中使用标记格式化字符

有一个特殊的标记，sel，用于管理选择，且具有以下特性。

- 一旦字符被标为 sel，那文本组件就拥有了选择。
- 文本组件支持对选择进行检索，返回所有用 sel 标记的字符。
- 如果选中的范围被另外的应用程序或当前应用程序中的其他窗口使用了，那么 sel 标记会从文本的所有字符中移除。
- 一旦 sel 标记的范围有所改变，一个虚拟事件<<Selection>>就会产生。更多介绍可参见 24.5 节。
- tag delete 命令不能删除 sel 标记。
- sel 标记不能与同等文本组件共享。相反，每个文本组件拥有它自身的 sel 标记。在 24.9 节中将介绍同等文本组件。

24.4.1 标记选项

大概共有 24 个标记选项，可分为三类：可视性类，用于隐藏文本；格式化类，影响字符的外观，如字体、大小和颜色等；还有布局类，影响显示中字符的版式。

唯一的可视性类标记选项是-elide，为布尔运算值。如果该值为真，那么被添加标记的字符在文本组件中不会被显示出来。默认状态下，其他文本子命令仍旧对隐藏的字符有效，尽管其中有许多命令都有忽略隐藏字符的选项。

格式化标记选项类似于文字处理程序中的命名字符类型，它们影响单个字符的外观。因此，文本中单独的一行能拥有若干个标记，这些标记用于对字符组进行格式化，例如设置不同的颜色、字体、大小等。下述格式化标记选项是有效的。

- -background——字符的背景色。
- -bgstipple——背景的点画模式。

- -borderwidth——文本边界的宽度，单位为屏幕距离。
- -fgstipple——应用于字符的点画模式。
- -font——字符字体。
- -foreground——字符的颜色。
- -offset——文本基线与整行基线垂直偏移的像素个数；值为正则会提升文本至上标，而负值将降低文本至下标。
- -overstrike——指定是否在字符中央画删除线的布尔值。
- -relief——用于文本的三维边界样式。
- -underline——指定是否给字符标下划线的布尔值。

布局标记选项类似于在文字处理程序中的命名段落样式，其中它们决定整体显示行的显示设置。布局标记选项不同于格式化标记选项，只有当标记应用于显示行的第一个未被隐藏(可见的)的字符时它才起作用。由于预言哪些字符会在显示行的开始部分显示是困难的，所以当使用这些标记选项时，通常的方法是将相关的标记应用到整个字符逻辑行。可用的布局标记选项如下所述。

- -justify——如何对齐显示行；值为 left、right 或 center。
- -lmargin1——组件中首行的左侧行缩进，单位为屏幕距离。
- -lmargin2——组件中非首行的左侧行缩进，单位为屏幕距离。
- -rmargin——组件中行的右侧行缩进，单位为屏幕距离。
- -spacing1——首行之上的多余空间，单位为屏幕距离。
- -spacing2——非首行之上的多余空间(行间距)，单位为屏幕距离。
- -spacing3——末行之下的多余空间，单位为屏幕距离。
- -tabs——制表位列表，如下所述。
- -tabstyle——制表符样式，如下所述。
- -wrap——是否在单词分界(word)或字符分界(char)处折行，或者根本不折行(none)。

标记的制表位和组件整体由-tabs 选项一起设置，它可以接受一个制表位的列表，而该列表以从组件左侧边缘开始的屏幕间隔值给出。在列表的每个位置元素之后，可以选择提供以下的关键字作为另外一个元素来设置制表位的对齐：left(默认)、right、center 或者 numeric。left 将制表字符后的文本的左边缘定位到制表位置，right 意味着制表字符后的文本的右边缘被定位在制表位置上，center 意味文本相对于制表位置居中。numeric 意味着将文本中的小数点置于制表位。如果没有小数点，则最低位数字与制表位的左侧对齐；如果文本中没有数字，那么文本在制表位右对齐。例如：

```
-tabs {2c left 4c 6c center}
```

在 2 厘米的间隔上创建了 3 个制表位：前两个左对齐，第三个中间对齐。

如果制表位列表没有足够的元素来覆盖文本行中的所有制表符，Tk 可以使用空格和对齐方式从列表中最后的制表位来外插新的制表位。如果没有指定-tabs 选项，或者如果被指定成一个空的列表，则 Tk 使用默认的以每 8 个(平均大小)字符为间隔的制表符。

-tabstyle 选项指定如何解释行制表位和那一行文本中制表符之间的关系。其值必须为

tabular(默认)或者 wordprocessor。如果制表符样式是 tabular，那么该行文本中第 *n* 个制表符字符就被关联到为该行定义的第 *n* 个制表位。如果制表符字符的 *x* 坐标落到第 *n* 个制表位的右边，文本组件就会显示一个空格来应急。如果制表符样式是 wordprocessor，任意安置的制表符字符都使用在该行已存在的前一字符的右侧的第一个制表位。

24.4.2 标记优先级

如前所述，某个范围的字符能对应多个标记。如果两个或更多的标记指定的选项相互冲突，则使用具有最高优先级的标记的选项。如果没有为某个标记指定一个特殊的显示选项，或者被指定成一个空字符串，那么就不会使用那个选项，而是由下一个拥有最高优先权标记的选项取代之。如果没有标记指定某个特殊的显示选项，则启用组件的默认样式。

当定义了新的标记时，它的优先级要高于文本组件中其他已有标记。您能够使用 tag lower 和 tag raise 的子命令来改变标记的优先级。例如，以下命令使用 important 标记并将其提升到最高标记优先级。

```
.text tag raise important
```

下一个示例降低 bold 的优先级至低于 URL 标记。

```
.text tag lower bold URL
```

24.4.3 标记绑定

文本组件允许您使用 tag bind 子命令关联绑定和标记。您能用绑定使部分文本处于激活状态，从而可以响应鼠标、键盘、<Enter>、<Leave>或虚拟事件。标记绑定还能让您应用超文本效果。例如，以下的绑定使得所有的 Tk 字都变成绿色(只要当鼠标经过它们中的任何一个)。

```
.text tag bind big <Enter> {
    .text tag configure big -background SeaGreen2
}
.text tag bind big <Leave> {
    .text tag configure big -background Bisque3
}
```

下述绑定可以实现在用户按住 Control 键并用鼠标 1 键单击任一 Tk 单词时，文本组件会重新加载，显示 Tk 演示中的 README 文件。

```
.text tag bind big <Control-Button-1> {
    .text delete 1.0 end
    loadFile $tk_library/demos/README
}
```

提示： 只有在鼠标光标处于包含绑定的标记区域时，键盘事件才触发标记绑定。而插入光标不一定必须在标记区域内。

多个绑定可能与同一个特定的事件相匹配，例如，已经创建了多个标记的绑定，然后将那些标记结合起来应用于文本中的某个部分。当这样的事件发生时，所有匹配的绑定都

会按标记优先级从低到高的顺序调用。如果多个绑定与单个标记匹配，则只有特定的绑定才能调用。绑定脚本中的 continue 命令用来终止该脚本，而 break 命令则用来终止该脚本，并跳过该事件的脚本的剩余部分，就像 bind 命令那样。

提示： 如果已经为使用 bind 命令的文本组件创建了绑定，那么它们会与用 tag bind 子命令为标记创建的绑定一起调用。标记绑定的调用优先于整个组件的绑定。

24.5　虚拟事件

文本组件产生两个虚拟事件。您可以用您喜欢的任何处理方式来为这些事件创建绑定。通常它们用来动态启用或禁用其他接口特性，从而以一种环境敏感的模式来运转。

<<Modified>>事件与 24.8 节介绍的撤销机制相关联。该特性需要将-undo 选项设置为真(1)来启用。在组件的修改标志被清除后，第一次修改组件中的文本会产生<<Modified>>事件。在用 edit undo 组件命令或 edit modified 命令清除标记时，该事件也产生。<<Modified>>事件通常用于启用一个 save 特性，而且只有在上一次保存操作过后组件内容被修改时才会启用该特性。24.8 节有类似的示例。

每当组件的选择发生变化时，<<Selection>>事件都会产生。当且仅当 sel 标记应用于任意字符时，它会启用接口特性，例如剪切和复制，若没有应用，则它会禁用该特性。第 22 章有这样的一个示例。

24.6　嵌入式窗口

除文本外，文本组件可以包含任意多个组件。这些被称为嵌入式窗口，且能作为任意组件类的实例。图 24.3 展示了 Tk 组件演示的嵌入式窗口示例。

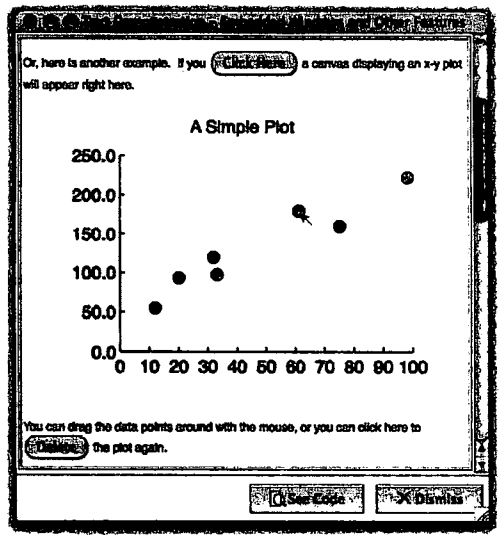


图 24.3　文本组件中嵌入的按钮和画布组件

组件随文本以内联的方式插入文本行中，并使用与文本相同的补白、间隔、对齐和折行规则，就好像当作字符处理一样，区别只是在于它们是非常大的字符。

文本组件充当了嵌入式窗口的几何管理器，因此组件应该创建为文本组件的子组件。window create 子命令将嵌入式窗口插入文本组件中。下述代码展示了如何完成该过程，结果参见图 24.4。

```
$t insert end  "You can click on the first button to"
button $t.on -text  "Turn on"   -command  "textWindOn $w"
$t window create end -window $t.on
$t insert end   "horizontal scrolling, which also turns"
$t insert end   "off word wrapping."
```

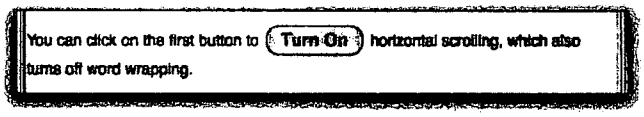


图 24.4　嵌入一个已有按钮到文本组件

插入组件的另一种方法是使用-create 选项而不是-window 选项。-window 选项要求组件被单独创建。-create 选项提供一个脚本，当需要嵌入窗口时可以用文本组件对其进行处理。由于嵌入式窗口无法在同级文本组件中共享，就需要为每个同级组件创建独特的组件。而创建脚本通常用于实现该目的。下述代码显示如何实现这一点，结果参见图 24.5。

```
$t insert end  "Or, here is another example. If you "
$t window create end -create {
    button %W.click -text  "Click Here"   \
        -command  "textWindPlot %W"
    }
$t insert end "a canvas displaying an x-y plot will"
$t insert end "appear right here."
```

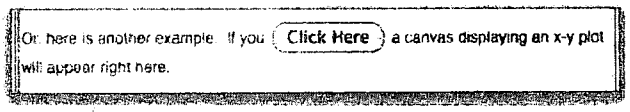


图 24.5　用创建脚本创建嵌入式按钮

24.7　嵌入图像

另一种能嵌入文本窗口的类型是图像。可以用 image create 组件命令将图像插入文本中，并可以出现在文本的一行中。当插入特殊用途的标记、图释甚至照片时，该命令很有用。下述脚本会显示在文本中插入一个作为标记的特殊停止符的图像，如图 24.6 所示。值得注意的是，并没有删除<STOP>文本并代之以图像，而是使用了一个将-elide 选项设置为真的标记。这使用户可能用

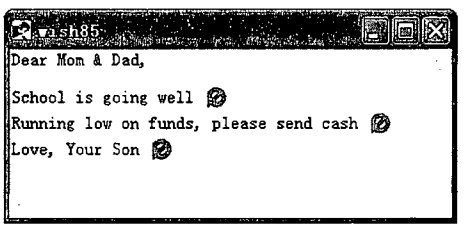


图 24.6　使用停止符作为嵌入图像的示例

get 子命令，或者执行一个复制-粘贴操作，来获得最初的文本。

```
text .edit -yscrollcommand   ".vsb set"  \
    -xscrollcommand  ".hsb set"   -wrap none
scrollbar .vsb -command  ".edit yview"   -orient vertical
scrollbar .hsb -command  ".edit xview"   -orient horizontal

set letter {Dear Mom & Dad,

School is going well <STOP>
Running low on funds, please send cash <STOP>
Love, Your Son <STOP>
}

.edit insert end $letter

set stopImage [image create photo \
    -file stop_sign.gif]

.edit tag configure symbol -elide 1

proc insertStops {w} {
    foreach ix [$w search -all -forward  "<STOP>"   1.0] {
        $w tag add $ix  "$ix + 6 char"
        $w image create $ix -image $::stopImage
    }
}

grid .edit -row 0 -column 0 -sticky nsew
grid .vsb -row 0 -column 1 -sticky ns
grid .hsb -row 1 -column 0 -sticky ew
grid rowconfigure . 0 -weight 1
grid columnconfigure . 0 -weight 1

insertStops .edit
```

图像采用和普通文本一样的折行和对齐规则。在下述示例中，只有图像被置于文本组件中，而且文本组件就像一个几何管理器一样以合理而美观的模式来安置照片。

```
text .t -yscrollcommand  ".vsb set"
scrollbar .vsb -command  ".t yview"

grid .t     -row 0 -column 0 -sticky nsew
grid .vsb   -row 0 -column 1 -sticky ns
grid rowconfigure . 0 -weight 1
grid columnconfigure . 0 -weight 1

foreach f [lsort -dictionary [glob *.gif]] {
    set img [image create photo -file $f]
    .t image create end -image $img -padx 2 -pady 2
    .t insert end " "
}
```

24.8 撤销

文本组件有无限的撤销/重做机制，将组件的-undo 选项设为 1 就可以启用。该机制记

录了在一个堆栈上的每个插入和删除操作。这些操作可以通过发出 edit undo 子命令来取消。在编辑操作之间，会插入边界，以定义操作组。这些组可用于定义复合编辑操作，它只需一步就可以完成撤销命令。在启用-autoseparators 选项时，可以自动插入边界，将插入或删除操作分组。您可以用 edit separator 子命令来明确地增加堆栈边界。如果想在应用程序中精确控制撤销操作，这将非常有用。

文本组件也可使用修改标志和<<Modified>>虚拟事件来跟踪修改记录。当修改标志设置为真，则自上次清除标志后，如对数据有任何修改(插入或删除)，都会触发虚拟事件。该标志可以决定组件的内容是否需要保存，启用或禁用按钮(例如“保存”按钮或者“撤销”按钮)。用 edit modified 组件命令可以读取和设置标志。

以下代码向本章前面的示例中添加了撤销和重做按钮。

```
# text: read a file into a text widget, with undo

text .text -relief raised -bd 2 \
    -yscrollcommand ".scroll set"   \
    -undo 1
scrollbar .scroll -command ".text yview"
proc loadFIle file {
    .text delete 1.0 end
    set f [open $file]
    .text insert end [read $f]
    close $f

    # Mark as not being changed
    .text edit modified 0
}

proc undoText {tw} {
    catch {$tw edit undo}
}

proc dataModified {tw} {
    if {[$tw edit modified]} {
        .undo configure -state normal
    } else {
        .undo configure -state disabled
    }
}

frame .toolbar
button .undo -text "Undo"  -command {undoText .text}

bind .text <<Modified>> {
    dataModified .text
}

pack .undo -in .toolbar -side left
pack .toolbar -side top -expand 1 -fill x
pack .scroll -side right -fill y
pack .text -side left

loadFile README
```

注意，我们可以增加下面的命令来修改 loadFile 功能，以便在加载文件后清除修改标志。

```
.text edit modified 0
```

另外，我们可以将-undo 1 选项添加到文本组件命令，并以合适的顺序打包组件。图 24.7 显示了第一次执行应用程序的结果。

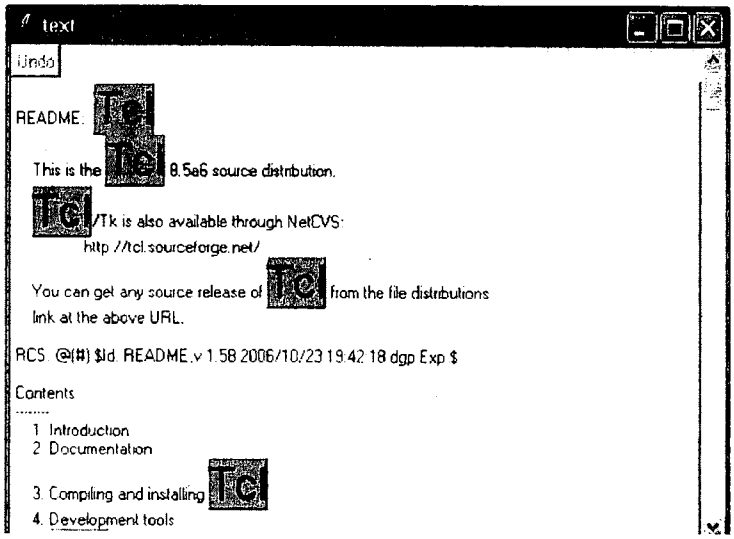


图 24.7 用 Undo 按钮更新的文本窗口

有了这些添加，只要对文本组件进行修改，如录入或者删除一些信息，便可启用 Undo 按钮，如图 24.8 所示。之后只需单击 Undo 按钮，就可取消编辑。一旦撤销堆栈用尽，Undo 按钮便自动禁用，如图 24.9 所示。

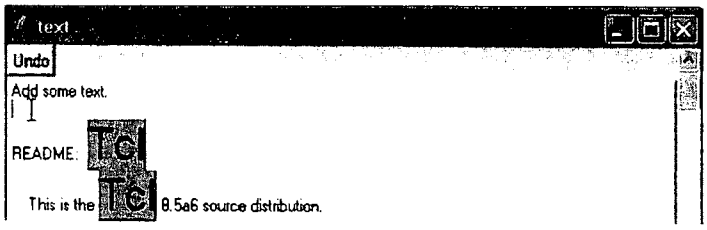


图 24.8 编辑操作会自动启用 Undo 按钮

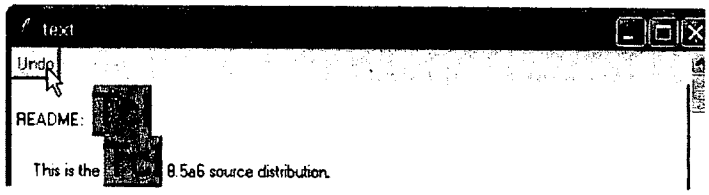


图 24.9 用尽 Undo 堆栈会使 Undo 按钮禁用

24.9 同级文本组件

peer 文本组件命令用于创建文本组件副本，它们能共享相同的文本数据。一个文本组件中的任何改变都能立即出现在它的同级组件中。该命令对创建分隔视图很有用，在大多数现代文本编辑器中很常见。同级组件共享所有的文本、图像和标记。它们不共享嵌入式窗口、sel 标记，或记号 insert 和 current。下述脚本是创建两个同级文本组件的示例，您对其中一个做出的修改都会立即反映到另一个中，如图 24.10 所示。

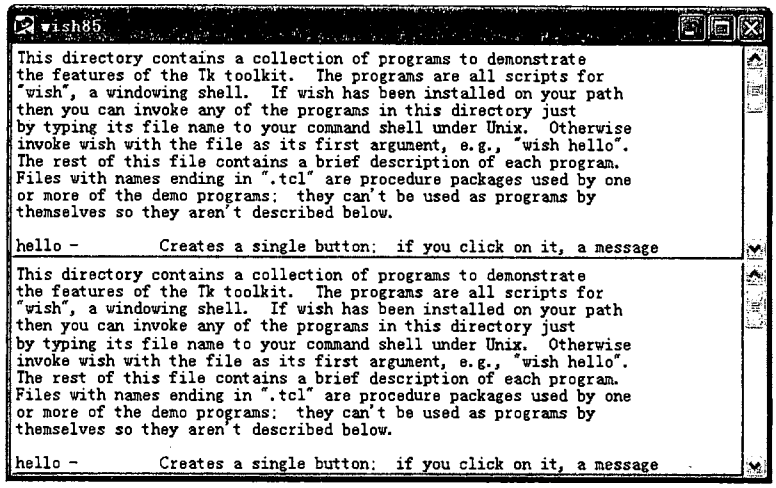


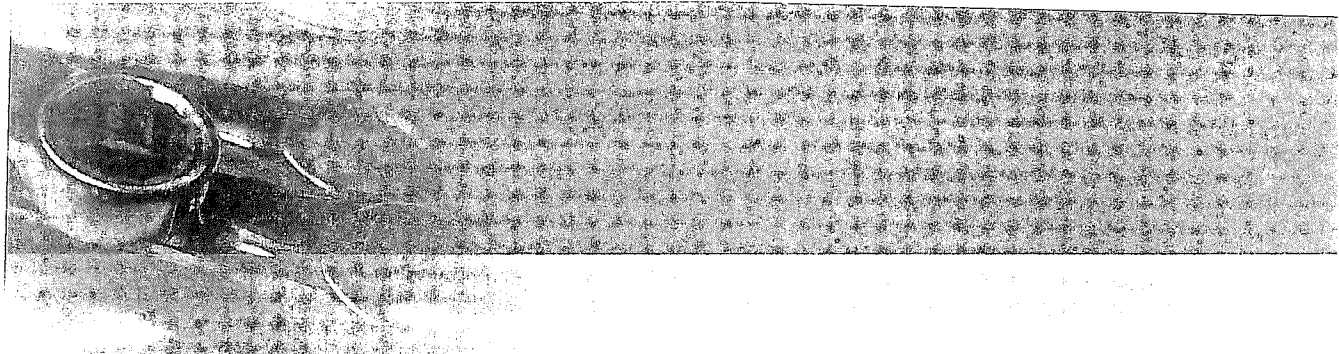
图 24.10 使用同级文本组件的分隔视图

```
panedwindow .pane -orient vertical
frame .tf ; # Top Frame
frame .bf ; # Bottom Frame
text .text -relief raised -bd 2 \
    -yscrollcommand ".scroll set"   \
    -undo true -autoseparators true \
    -height 12

.text peer create .text2 -relief raised -bd 2 \
    -yscrollcommand ".scroll2 set"   \
    -undo true -autoseparators true \
    -height 12
scrollbar .scroll -command ".text yview"
scrollbar .scroll2 -command ".text2 yview"
pack .scroll -in .tf -side right -fill y
pack .text -in .tf -side left -fill both -expand yes
pack .scroll2 -in .bf -side right -fill y
pack .text2 -in .bf -side left -fill both -expand yes
.pane add .tf -sticky nsew
.pane add .bf -sticky nsew
```

```
pack .pane -side top -fill both -expand 1
loadFile README
```

同级文本组件是实际同等的。这意味着删除原始文本组件并不影响已有的同级组件，仅当所有同级文本组件都被删除时，文本组件的内容才会被删除。同级组件也能创建其他的同级组件。peer names 子命令用于找到某个文本组件的所有同级组件的组件路径名。



第 25 章　选择与剪贴板

选择是一种在组件和应用程序间传递信息的机制。用户首先选择组件中的一个或者多个对象，例如，在文本中拖动鼠标选择内容或者单击一个图形对象。在选中以后，用户可以在其他组件中调用命令来获取该选择的信息，例如在所选范围内的字符或者包含选择的文件名。在当今的窗口环境中，存在两种不同的选择模式：选择所有者模式和剪贴板模式。

选择所有者模式在 X 窗口系统中使用。在任何给定时间都有单个的选择所有者。包含选择的组件和请求选择的组件可以在同一个或不同的应用程序中。接收应用程序或动作可以查询选择所有者并获得选择的信息。

剪贴板模式用于 Microsoft Windows 系统和 Mac OS X 以及 X 窗口系统中，在这种模式下，使用称为剪贴板的一种全局的、虚拟的存储设备。选中的条目会复制到剪贴板，之后它们可被另一个组件或应用程序获取。这种模式也使用选择，但在这里，所有者关系局限于当前的活动窗口。

选择，如同剪贴板一样，最常用于将信息从一个地方复制到另一个地方。Tk 组件有默认绑定来完成这些常见的选择和剪贴板操作，因此几乎很少需要明确地管理应用程序中的选择或者剪贴板。然而，如果应用程序需要提供自定义选择或者剪贴板处理，可以使用 selection 和 clipboard 命令来实现。selection 命令用于管理选择所有者关系，并提供对选择数据的访问。clipboard 命令用于管理剪贴板，虽然 selection 命令也能用于这个目的。

25.1　本章出现的命令

本章将介绍以下管理选择的命令。

- `selection clear ?-displayof` *`window`*`? ?-selection` *`selection`*`?`
  清除或“取消选择”当前的选择。*window* 默认为“.”，*selection* 默认为 PRIMARY。
- `selection get ?-displayof` *`window`*`? ?-selection` *`selection`*`?`
  `?-type` *`type`*`?`
  根据 *selection* 和 *type* 检索选择。*window* 默认为“.”，*selection* 默认为 PRIMARY，*type* 默认为 STRING。
- `selection handle ?-selection` *`selection`*`? ?-type` *`type`*`?`
  `?-format` *`format`*`? window script`

创建一个请求选择的处理器。在 *selection* 由 *window* 拥有，且您试图按 *type* 给出的格式检索它时，会处理 *script*。*selection* 默认为 PRIMARY，*format* 和 *type* 默认为 STRING。更多信息可参见本书和参考文档。

- `selection own ?-displayof window? ?-selection selection?`
  返回 *selection* 的当前所有者，或者如果没有窗口拥有 *selection*，则返回空字符串。*window* 默认为“.”，*selection* 默认为 PRIMARY。
- `selection own ?-command script? ?-selection selection? window`
  声明对 *window* 的 *selection* 的所有权关系。如果指定了 *script*，则它会在 *window* 失去选择时执行。*selection* 默认为 PRIMARY。
- `clipboard clear ?-displayof window?`
  声明对 *window* 的显示中剪贴板的所有权关系，并移除之前的任何内容。*window* 默认为“.”。
- `clipboard append ?-dispalyof window? ?-format format? ?-type type? ?--? data`
  按 *type* 给出的格式，使用 *format* 给出的说明，添加 *data* 到 *window* 的显示中的剪贴板，并声明对 *window* 的显示中剪贴板的所有权关系。*window* 默认为“.”，*format* 和 *type* 默认为 STRING。
- `clipboard get ?-displayof window? ?-type type?`
  从 *window* 的显示中的剪贴板检索数据。*type* 指定了返回数据的格式。*window* 默认为“.”，*type* 默认为 STRING。

25.2　选择、检索和类型

在选择所有者模式中，当用户选中窗口中的信息时，窗口需要选择所有权关系。传统上使用 PRIMARY 选择，但 X 窗口系统也支持其他的选择。一个组件中的多个不连续对象可能同时被选中(如一个列表框中的三个不同条目或者绘图窗口中的若干个不同多边形)，但选择通常由单个对象或一些相邻对象组成。在窗口需要获得选择的所有权关系时，任何先前的所有者都会被告知，它已失去了对选择的所有权。任何时候，选择所有者也许会接收到采用特殊格式表示选择内容的请求，因此选择所有者必须处理好这些请求服务，直至失去对选择的所有权。

在剪贴板模式中，用户选择了窗口中的一些信息，然后明确要求将数据复制到剪贴板。这时，客户端会用 CLIPBOARD 选择来请求从剪贴板中复制数据。

当检索选择时，可以要求若干不同种类的信息，这些种类称为检索类型，也被称为目标。最常见的类型是 STRING。此时，选择的内容就返回为字符串。例如，如果选中文本，采用类型 STRING 的检索会返回所选文本的内容；如果选中图形，那么采用类型 STRING 的检索会返回所选图形的一些字符串说明。如果采用类型 FILE_NAME 检索，返回值会是与选择关联的文件名。如果使用类型 LINE，返回值会是文件中被选中行的编

号。存在很多被精确定义的类型，更多信息可参见 X 联合标准客户端间通信约定手册(X Consortium Standard Inter-Client Communication Conventions Manual，ICCCM)。

selection get 命令会检索选择。该类型也许会被明确指定，也许不被指定，这时它默认为 STRING。例如，当选择由包含莎士比亚的《罗密欧与朱丽叶》文本的文件一行中的一些单词组成时，下述命令可能被调用。

```
    selection get
⇒   star-crossed lovers
    selection get -type FILE_NAME
⇒   romeoJuliet
    selection get -type LINE
⇒   6
```

该命令能在任何含有该选择的显示上的 Tk 应用程序中发出；它们不必在包含选择的应用程序中发出。

不是每个组件都支持每个可能的选择类型。例如，如果组件中的信息与文件无关联，FILE_NAME 类型也许不能被支持。如果尝试用一个不被支持的类型检索选择，则会产生错误。由选择所有者来定义支持选择的类型，并执行转换。幸运的是，每个组件应该都支持采用类型 TARGETS 的检索，这样的检索会返回由当前选择所有者支持的所有目标格式。您可以用 TARGETS 检索的结果来获取最方便的有效类型。例如，下述过程会尽可能取得 PostScript 格式的选择，否则获取一个未格式化的字符串。

```
proc getSelection {} {
    set types [selection get -type TARGETS]
    if {"POSTSCRIPT" in $types } {
        return [selection get -type POSTSCRIPT]
    } else {
        return [selection get -type STRING]
    }
}
```

Tk 组件总是支持类型 STRING。Tk 组件也支持下述 ICCCM 类型：MULTIPLE、TARGETS、TIMESTAMP、TK_APPLICATION 和 TK_WINDOW，以及只在 X 窗口系统中使用的 UTF8_STRING。MULTIPLE 用于同时选择多个条目或者文本块。TARGETS 给出支持的类型。TIMESTAMP 显示选择何时发生。这两种 Tk 类型只针对 Tk 应用程序，详情参见后文。

25.3 定位和清除选择

Tk 提供了两种机制来找出选择的拥有者。selection own 命令(没有别的参数)检查选择是否由正调用的应用程序的组件拥有。如果是，它会返回该组件的路径名；如果不存在选择或者该选择由其他应用程序拥有，则 selection own 会返回一个空字符串。

第二种定位选择的方法是使用 TK_APPLICATION 和 TK_WINDOW 检索类型。这些类型都由 Tk 执行，且对 Tk 应用程序中的选择有效。命令

```
selection get -type TK_APPLICATION
```

返回拥有选择的 Tk 应用程序名(例如，采用适用于 send 命令的格式)，而命令

```
selection get -type TK_WINDOW
```

会返回拥有选择的组件路径名。如果拥有选择的应用程序并不是基于 Tk，它不会支持 TK_APPLICATION 和 TK_WINDOW 类型，且 selection get 命令会返回一个错误。如果不存在选择，这些命令也会返回错误。

selection clear 命令会清除应用程序主窗口中显示的任何选择，不管选择是在正调用的应用程序中，还是在同一显示上别的其他应用程序中。当且仅当选择在正调用的应用程序中时，下述脚本会清除选择。

```
if {[selection own] ne ""} {
    selection clear [selection own]
}
```

25.4 用 Tcl 脚本提供选择

标准组件(如输入框和文本)已经包含了能提供选择的 C 代码，因此在写 Tcl 脚本时，不用担心这一点。然而，编写能提供选择的 Tcl 脚本是可能的。例如，您也许想让组件支持另一种类型，如 FILE_NAME，这样可与您环境中的其他应用程序更好地整合。

提供选择的规范包括三部分。

(1) 组件必须声明对选择的所有权关系。这会撤销之前的任何选择，然后一般会采用高亮的模式重新显示新选中的材料。

(2) 选择所有者必须响应其他组件和应用程序的检索要求。

(3) 所有者也许会要求当选择撤销时被告知。组件一般会取消高亮显示，以响应对选择的撤销。

后文将介绍两种情况。第一种只向支持选择的组件添加新的输入，因此它只与规范的第二部分有关。第二种情况是为以前不支持选择的组件组提供完整的选择支持，它会涉及规范的所有三个部分。

假设您想向一个特殊组件添加支持的新类型。例如，文本组件包含 STRING 类型的内置支持，但它们不会自动支持 FILE_NAME 类型。您可以用以下脚本为 FILE_NAME 检索添加支持。

```
selection handle -type FILE_NAME .t MyApp::getFile
proc MyApp::getFile {offset maxBytes} {
    variable fileName
    set last [expr {$offset + $maxBytes -1}]
    return [string range $fileName $offset $last]
}
```

该代码假定文本组件被命名为.t，且与之关联的文件名存储在命名空间变量 MyApp::fileName 中。在.t 拥有选择且您试图用 FILE_NAME 类型检索它时，selection handle 命令会让 Tk 调用 MyApp::fileName。在该检索发生时，Tk 会采用指定命令(此例中的 MyApp::fileName)，附加两个额外的数值参数，并将结果字符串作为 Tcl 命令来调用。在这种情况下，命令：

```
MyApp::getFile 0 4000
```

会产生结果。额外的参数会用第一个字节和最大长度来标识选择的子区间，且该命令必须返回选择的份额。如果要求的区间超过了选择的末端，则命令应该返回从给定的起点到选择的终点的所有信息。Tk 会返回信息到请求它的应用程序中。在大多数情况下，整个选择是在命令的一次调用中获取的；对于大型选择，Tk 会进行若干单独调用，这样它可以将选择以可管理片段的方式传回给请求者。

在先前的示例中，一种新类型被简单添加到一个提供内置选择支持的组件中。如果选择支持正在被添加到一个完全没有内置支持的组件中，就需要额外的 Tcl 代码声明对选择的所有权，并响应撤销选择的通告。例如，有三个单选按钮，分别叫.a、.b 和.c，它们用-variable 和-value 选项来配置，以将被选中按钮的信息存储到 MyApp::state 变量中。假定您想将单选按钮和选择绑定在一起，以实现下列目的：(1)一旦按钮被选中，它会声明 X 选择；(2)选择检索返回 MyApp::state 的内容；(3)当其他组件声明了选择而按钮失去选择时，MyApp::state 会被清除，且所有按钮会被撤销选择。以下代码实现这些功能。

```
    selection handle -type STRING .a MyApp::getValue
proc MyApp::getValue {offset maxBytes} {
    variable state
    set last [expr {$offset + $maxBytes -1}]
    return [string range $state $offset $last]
}
foreach w {.a .b .c} {
    $w config -command {
        selection own -command MyApp::selGone .a
    }
}
proc MyApp::selGone {} {
    variable state
    set state {}
}
```

selection handle 命令和 MyApp::getValue 过程与之前的示例类似：它们会返回 MyApp::state 变量的内容，来响应对.a 的 STRING 选择请求。foreach 循环为每个组件指定了一个-command 选项。这样，如果用户单击任何一个单选按钮，就会调用 selection own 命令，且 selection own 命令会声明对组件.a 选择的所有权(不管用户选择的是哪个单选按钮，.a 都拥有选择，且会返回 MyApp::state 值，来响应选择请求)。selection own 命令也指定了当其他组件声明选择所有权时，过程 MyApp::selGone 应该被调用。MyApp::selGone 会设置 MyApp::state 为一个空字符串。所有单选按钮会监控 MyApp::state 的变化，因此当它被清除时，单选按钮会撤销对自身的选择。

25.5 clipboard 命令

clipboard 命令与窗口系统的剪贴板互相结合。剪贴板中的数据可以由 selection 命令的-selection CLIPBOARD 选项或 clipboard get 命令来获得。和采用 PRIMARY 选择时不同，剪贴板中的数据在稍后的时间也可检索，不管谁拥有选择，也不管是否选择了任何东西。

剪贴板的典型用法是为窗口执行剪切、复制和粘贴操作。文本组件为虚拟<<Copy>>事件提供了一个绑定，如下所示：

```
proc tk_textCopy {w} {
    if {![catch {set data [$w get sel.first sel.last]}]} {
        clipboard clear -displayof $w
        clipboard append -displayof $w $data
    }
}
```

这项功能会从文本组件 w 中检索选择，清除当前的剪贴板内容，然后用 clipboard append 命令复制数据到剪贴板。粘贴功能执行相反操作。

```
proc tk_textPaste {w} {
    if {![catch {clipboard get -display $w} sel]} {
        $w insert insert $sel
    }
}
```

提示： *这只是为了演示而对实际功能进行的简化。实际功能也会删除目标窗口中的当前选择，并为组件的撤销功能管理编辑历史。*

25.6　拖曳和释放

拖曳和释放(DND)是用鼠标选择对象再将其“拖曳”到目标窗口的能力。该动作简化了应用程序中常见的复制/粘贴或者剪切/粘贴动作。Tk 不直接支持这种类型的交互作用，但使用 selection 或者 clipboard 命令可以方便地实现内部应用程序 DND。DND 要求的步骤如下。

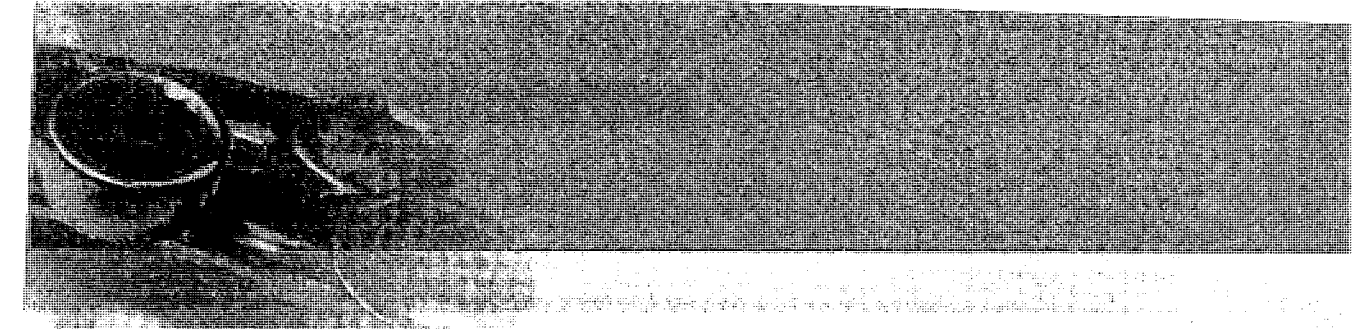
(1) 对于<ButtonPress-1>，标记当前窗口和位置，假设鼠标光标在一个已选对象上。

(2) 对于鼠标运动，要等到鼠标有效地从起点移动后再开始“拖曳”操作。一旦鼠标已经移动了足够远，可以视为“拖曳”了，才开始拖曳并提供反馈，例如，改变光标。

(3) 对于<ButtonPress-1>，使用相对显示根窗口的 x 和 y 坐标来标识鼠标下方的窗口(winfo containing 命令可以做到这一点)。

(4) 获得选择并插入数据到鼠标下方位置，该位置由目标窗口中的%X 和%Y 的位置指定。

这种简单的方法在 Tk 应用程序中效果很好，但要是您想在两个应用程序之间，或者在应用和窗口系统中支持 DND 呢？有一些扩展可以在多个平台之间执行标准的 DND 规范，例如 TkDND(参见 http://www.sourceforge.net/projects/tkdnd)。这种扩展为您处理操作的序列，因此应用程序需要提供的唯一代码就是执行实际选择的提取和在释放点插入数据的代码。



第26章 窗口管理器

窗口管理器是窗口系统的一部分，它的主要功能是控制每个屏幕所有顶层窗口的布置与装饰。在这些方面，它和几何管理器类似，只是它管理的是所有应用程序的顶层窗口而不是应用程序的内部窗口。窗口管理器允许每个应用程序为其顶层窗口申请一个特殊位置和尺寸，用户可以交互地覆盖这些数据。除充当几何管理器以外，窗口管理器还有以下用途：它们在顶层窗口周围添加装饰框架；它们允许窗口图标化和去图标化；它们可以向应用程序通告一些事件，如用户关闭窗口的请求。

一些操作系统只有一个桌面环境或窗口管理器，例如 Microsoft Windows 系统和苹果公司的 Mac OS X Aqua，但 Linux 和其他的基于 Unix 的操作系统允许多个实现不同样式布局的不同窗口管理器存在，并提供不同种类的装饰和图标管理系统等。例如 GNOME 和 KDE 桌面环境，还有早期的 CDE 或称为 Common Desktop Environment(公共桌面环境)。在任一给定的时间一个显示器上只运行单个窗口管理器，用户需要选择使用哪个窗口管理器。

使用 Tk 中的 wm 命令，可以与桌面管理器通信。Tk 执行 wm 命令，这样任何基于 Tk 的应用程序都应该与任何窗口管理器协同工作，而不用考虑当前使用的窗口管理器或桌面环境的类型。为了在桌面上创建一个功能良好的应用程序，程序需要调用 wm 命令，正确地告知窗口管理器关于窗口标题、图标等的信息。

提示： 通常，应用程序仅可以向窗口管理器发送请求或者提供注意事项。而窗口管理器可以选择忽略或修改那些请求和提示信息。例如，应用程序可以为自己的顶层窗口要求一定的尺寸，但窗口管理器可能会由于受限于有效显示尺度而给出一个小一些的尺寸，或者应用程序可能会要求一种特定窗口管理器不支持的功能。

26.1 本章出现的命令

本章讨论与窗口管理器进行交互的 wm 命令。在下述所有的命令中，*window* 必须是一个顶层窗口名。许多命令，例如 wm aspect 或 wm group，用来设置和查询与窗口管理关联的多个参数。对于这些命令，如果参数被指定为空字符串，那么该参数将被彻底移除；如果参数被省略，命令将会返回参数的当前设置。

- `wm aspect` *`window ?xThin yThin xFat yFat?`*
  将 *window* 的宽高比设为指定值。

- `wm attribute window ?attribute? ?value attribute value ...?`
  设置或查询针对窗口系统的属性。详情参见 26.8 节和参考文档。
- `wm client window ?name?`
  设置或查询 *window* 的 WM_CLIENT_MACHINE 属性，该属性给出了正在运行 *window* 的应用程序的机器的名称。
- `wm command window ?value?`
  设置或查询 *window* 的 WM_COMMAND 属性，该属性包括了用来初始化 *window* 应用程序的命令行。
- `wm deiconify window`
  将 *window* 设为常规显示(非图标化)。
- `wm focusmodel window ?model?`
  设置或查询 *window* 的焦点模式。*model* 必须为 active 或 passive。
- `wm forget window`
  设置 *window*，使其不再作为一个顶层窗口被窗口管理器管理。
- `wm geometry window ?value?`
  设置或查询为 *window* 要求的几何外观。*value* 必须具有=*width*x*height*±*x*±*y* 的格式(任何=、*width*x*height* 或±*x*±*y* 可被忽略)。
- `wm grid window ?baseWidth baseHeight widthInc heightInc?`
  表明将 *window* 视为网格化窗口来管理，并指定网格单元与像素单元的关系。更多信息可参见参考文档。
- `wm group window ?leader?`
  设置或查询 *window* 所属的窗口组。*leader* 必须是一个顶层窗口名，或者是空字符串，这样可将 *window* 从当前组中移除。
- `wm iconbitmap window ?-default? ?bitmap?`
  设置或查询 *window* 图标的位图。如果指定了-default 选项，该图标会应用于未使用其他特定图标的所有顶层窗口(现有和将来的)。
- `wm iconify window`
  为 *window* 设置图标。
- `wm iconmask window ?bitmap?`
  设置或查询 *window* 图标的掩码位图。
- `wm iconname window ?string?`
  设置或查询显示在 *window* 图标上的字符串。
- `wm iconphoto window ?-default? image1 ?image2 ... ?`
  基于命名相片设置 *window* 的标题栏图标。允许多图像提供不同图像尺寸。如果指定了-default 选项，则图标也会应用于所有将来创建的顶层窗口。
- `wm iconposition window ?x y?`
  设置或查询在屏幕的哪个位置显示 *window* 图标的提示信息。

- `wm iconposition window ?icon?`
  设置或查询用作 *window* 图标的窗口。*icon* 必须为顶层窗口的路径名。
- `wm manage window`
  设置 *window*，使其由窗口管理器作为顶层窗口管理。
- `wm maxsize window ?width height?`
  设置或查询在交互式调整尺寸的操作中允许的最大 *window* 尺寸。
- `wm minsize window ?width height?`
  设置或查询在交互式调整尺寸的操作中允许的最小 *window* 尺寸。
- `wm overrideredirect window ?boolean?`
  设置或查询 *window* 的覆盖-重定向标志。
- `wm positionfrom window ?whom?`
  设置或查询为 *window* 指定位置的来源。*whom* 必须为 program 或 user。
- `wm protocol window ?protocol? ?script?`
  设定在窗口管理器向带有指定的 *protocol* 的 *window* 发送信息时，执行 *script*。*protocol* 必须是窗口管理器协议的元素的名称，例如 WM_DELETE_WINDOW。如果 *script* 是一个空字符串，则当前的 *protocol* 处理器会被删除。如果 *script* 被忽略，当前的 *protocol* 脚本会被返回(如果没有 *protocol* 处理器，则返回一个空字符串)。如果 *protocol* 和 *script* 都被忽略，命令会返回一个为 *window* 定义的所有带处理器的协议。
- `wm resizable window ?width height?`
  根据给定的布尔运算值，控制用户是否能交互式改变 *window* 的 *width* 与 *height*。
- `wm sizefrom window ?whom?`
  设置或查询指定 *window* 大小的来源。*whom* 必须为 program 或 user。
- `wm state window ?newstate?`
  设置或查询 *window* 的当前状态。状态可以是 normal、iconic、withdrawn 或 zoomed(仅适用于 Windows 和 Mac OS X)中的一个。
- `wm title window ?string?`
  设置或查询 *window* 的装饰边缘中显示的标题字符串。
- `wm transient window ?master?`
  设置或查询 *window* 的瞬时状态。*master* 必须是一个顶层窗口名，*window* 正代表该窗口作为一个实例来运行。
- `wm withdraw window`
  使 *window* 退出屏幕。

26.2 窗口尺寸

即便 Tk 的应用程序不触发 wm 命令，Tk 仍会代表应用程序与窗口管理器通信，这样，顶层窗口才会显示在屏幕上。每个顶层窗口默认按自然尺寸显示，该尺寸是应用程序

使用几何管理的正常 Tk 机制来要求的。Tk 将窗口自然尺寸传给窗口管理器，且大部分窗口管理器会批准该请求。如果顶层窗口的自然尺寸应该改变，Tk 会将新值传给窗口管理器，窗口管理器会根据最后一次请求来调整窗口尺寸。

如果用户交互式地调整了顶层窗口的尺寸，那么从那时起窗口的自然尺寸就被忽略了。不管窗口的内部需求如何变化，它的尺寸依然保持用户的设置。如果您调用了 wm geometry 命令，则会产生相似的效果，如下例：

```
wm geometry .w 300x200
```

这条命令强制.w 为 300 像素宽，200 像素高，就像用户交互式地调整了窗口尺寸一样。.w 的自然尺寸会被忽略，且 wm geometry 命令中指定的尺寸会覆盖任何用户可能交互式指定的尺寸(尽管用户可以重新调整窗口的尺寸来覆盖 wm geometry 命令中的尺寸)。

如果想恢复一个窗口到其自然尺寸，可以用一个空的几何字符串来调用 wm geometry 命令。

```
wm geometry .w {}
```

这会让 Tk 忽略用户或 wm geometry 定义过的任何尺寸，所以窗口会返回到自然尺寸。

如果想限制对尺寸的交互式调整，可以调用 wm minsize 和/或 wm maxsize 来指定允许的尺寸范围。例如，脚本

```
wm minsize .w 100 50
wm maxsize .w 400 150
```

允许.w 重新调整尺寸，但限制其宽度为 100～400 像素，高度为 50～150 像素。如果命令

```
wm minsize .w 1 1
```

被调用，则.w 将失去尺寸下限。如果设置了最小值而没设置最大值，则最大值就是显示器的尺寸；如果设置了最大值而没设置最小值，则最小值不受限制。您可以用 wm resizable 命令使交互式调整窗口尺寸有效或失效，它允许您设置两个布尔运算值来指示是允许水平调整还是垂直调整。下述命令使水平和垂直调整失效，将窗口尺寸固定为执行命令时的尺寸。

```
wm resizable .w 0 0
```

除了限制一个窗口的尺寸，也可以用 wm aspect 限制它的宽高比。例如：

```
wm aspect .w 1 3 4 1
```

告诉窗口管理器不允许用户将窗口的宽高比设置为小于 1/3 或者大于 4。图 26.1 是各种宽高比的实例。在给定命令后，窗口管理器允许用户将.w 调整为两条虚线之间的任意形状，而不允许调整为两条虚线之外的形状。

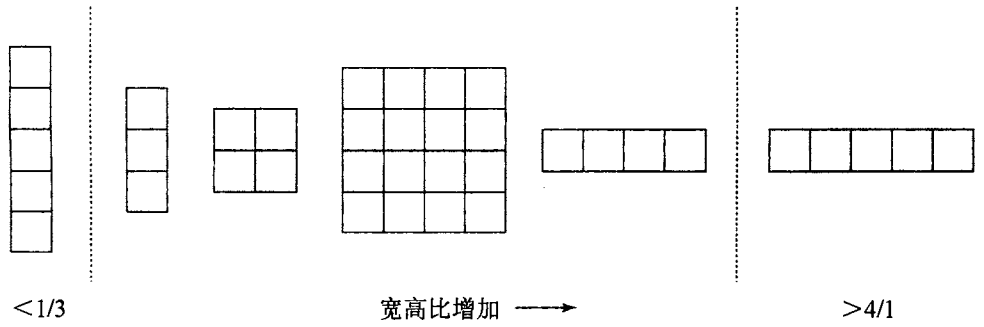


图 26.1　窗口的宽高比为它的宽度除以高度

26.3　窗口位置

控制顶层窗口的位置比控制它的尺寸更简单。用户可以交互式移动窗口，应用程序也可以用 wm geometry 命令来移动自己的窗口。例如，命令

```
wm geometry .w +100+200
```

定位.w，以使它的左上角为显示器上坐标为(100，200)的像素点。如果两个“+”中的任一个被替换为“-”，坐标就从显示器的右下角开始。例如；

```
wm geometry .w -0-0
```

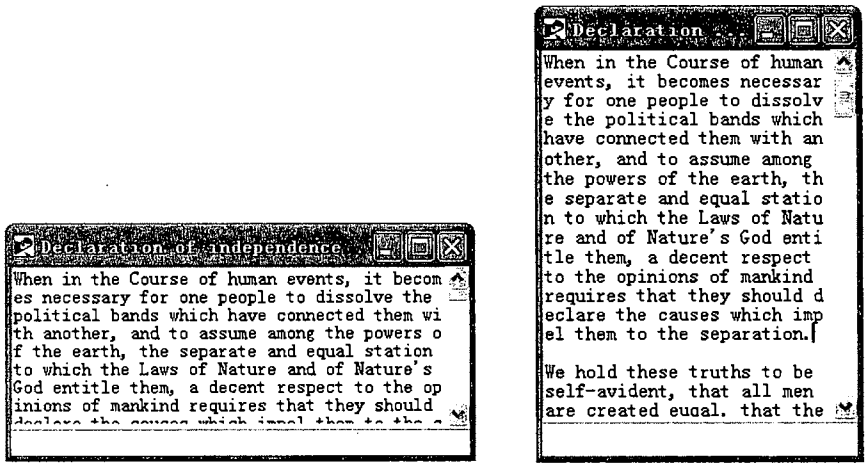
将.w 定位在显示器的右下角。

提示： 如果正在使用一个虚拟的根窗口管理器，您在 wm geometry 命令中指定的位置可能是相对屏幕的，也可能是相对虚拟根窗口的。具体的情况可参见窗口管理器文档。在有些窗口管理器中，可以使用窗口管理器选项和 wm positionfrom 命令来控制所使用的解释。

26.4　网格化窗口

在一些情况下，将窗口的尺寸调整为任意大小是没有意义的。例如，考虑图 26.2 中的应用程序。在用户调整顶层窗口的尺寸时，文本组件也会有相应的变化。理想情况下，文本组件的每个尺寸都应该是一个整数，导致字符不全的尺寸应该取整。

网格化几何管理可以达到这个效果。如果顶层窗口的网格化有效，则窗口尺度就受限于一个虚构的网格。网格的几何外观由顶层窗口中的一个组件决定(例如，图 26.2 中的文本组件)，这样组件尺寸的内部对象数目就总是整数了。通常控制网格的组件是面向文本的组件，例如输入框、列表框或者文本。如果用户交互式地将窗口尺寸由图 26.2(a)所示变为图 26.2(b)所示，窗口管理器会将尺寸调整为整数，以使文本组件能在每个方向都有整数个字符。



(a) (b)

图 26.2 网格化的几何管理系统实例

为了使组件能控制顶层窗口的网格，可以在控制组件中将-setgrid 选项设置为 1。只有一个组件可以控制一个给定顶层窗口的网格。下述代码用在了生成图 26.2 的示例中，文本组件为.t。

```
.t configure -setgrid 1
```

该命令有几个效果。第一，它告知顶层窗口，该组件决定了整个窗口的网格尺寸。第二，它改变了在 wm geometry 命令中使用的尺寸的测量单位。对文本组件，测量单位是一个字符。例如，命令

```
wm geometry .50x10
```

设置了主窗口的尺寸，这样.t 的宽为 50 个字符，高为 30 行。wm minsize 和 wm maxsize 里的单位，还有组件的-width 和-height 选项都用网格单位而不是用磅来测量。

网格也能用 wm grid 命令来定义。该命令在网格单位尺寸不由组件决定时会用到。

提示： 只有对有固定尺寸的对象的窗口(如采用等宽字形的文本窗口)，网格化才会顺利工作。如果不同的字符有不同的尺寸，那么窗口的尺寸不一定要是整数个字符。而且，为使网格化正确工作，必须配置应用程序的内部几何管理系统，这样控制窗口就可以扩展和收缩，来响应顶层窗口尺寸的变化，例如，用选项-sticky nsew 进行网格化。

26.5 窗口状态

在任何给定的时间，每个顶层窗口都处在一个特定的状态。处于正常或去图标化状态的窗口会显示在屏幕上，处于 zoomed 状态(被 Windows 系统和 Mac OS X 支持)的窗口会在屏幕上最大化。处于图标化状态的窗口不会显示在屏幕上，但是在任务栏或桌面上显示一个图标。处于关闭状态的窗口不会在屏幕的任何地方显示，而且该窗口会完全被窗口管理器忽略。

新的顶层窗口以正常状态开始。您可以用窗口管理器设备来交互式地对一个窗口进行图标化，或者可以在窗口的应用程序中调用 wm iconify 命令。例如：

```
wm iconify .w
```

如果您在窗口第一次显示在屏幕之前，就立刻调用了 wm iconify 命令，它便会从图标状态开始。命令 wm deiconify 会使窗口返回正常状态。

命令 wm withdrawn 将一个窗口置于关闭状态。如果在窗口显示在屏幕上之前调用了该命令，窗口便会从关闭状态开始。这条命令最常见的用途是阻止应用程序的主窗口显示在屏幕上(在一些应用程序中，主窗口没有任何作用，用户界面是由顶层窗口给出的)。一旦窗口退出，它可以用 wm deiconify 或 wm iconify 返回屏幕。

wm state 命令设置或返回一个窗口的当前状态。

```
    wm iconify .w
    wm state .w
⇒ iconic
```

26.6 装饰

当窗口以正常状态显示在屏幕上，窗口管理器通常会在其周围增加一个装饰框架。该框架通常为窗口显示一个标题，且会包含用来调整窗口尺寸、移动窗口等的交互式控件。例如，Mac OS X Aqua 装饰了图 26.2 中的窗口。

wm title 命令用于设置显示在窗口装饰框架上的标题。例如，命令

```
wm title . "Declaration of Independence"
```

被用来设置图 26.2 中的窗口标题。

其他的 wm 命令对交互式框架控制有间接效果。例如，如果可调整尺寸的标志为假(wm resizable . 0 0)，最大化按钮或者调节控制柄(右下角)将不会显示。控制的准确变化取决于正在使用哪一个窗口管理器。

wm 命令提供了一些选项，来控制当窗口被图标化时会显示的内容。第一，可以使用 wm iconname 来指定在图标上显示的标题。第二，一些窗口管理器允许指定图标上显示的图像。wm iconphoto 命令用于设置该图像。一些早期的窗口管理器只支持图标使用简单的两色位图。wm iconbitmap 和 wm iconmask 分别用来设置一个双色图标和透明度蒙版。第三，一些窗口管理器允许将一个窗口作为另一个窗口的图标；如果窗口管理器支持该功能，可以由 wm iconwindow 命令来实现。第四，可以用 wm iconposition 来指定图标在屏幕上的位置。

提示： 几乎所有的窗口管理器都支持 wm iconname。wm iconphoto 命令在不同的窗口管理器上处理方式不同，很少有窗口管理器会支持 wm iconbitmap，几乎没有窗口管理器能很好地支持 wm iconwindow。您需要逐个尝试这些命令，看看在应用程序需要支持的平台上，哪个命令可以满足要求。

26.7　特殊处理：瞬态、组和覆盖-重定向

您可以要求窗口管理器为窗口进行特别的处理，以实现诸如工具栏、非模态对话框和闪现屏幕等特征。

可以用如下命令将一个顶层窗口标识为瞬态。

```
wm transient .w .
```

这告知了窗口管理器，.w 是一个短期窗口(例如对话框)，代表了应用程序的主窗口。wm transient 的最后一个参数(此示例中为“.”)被当作瞬态窗口的主窗口。大多数窗口管理器能确保一个瞬态窗口总是位于其主要窗口之上。另外，窗口管理器可以采用不同方式来处理瞬态窗口，例如提供更少装饰，或对它们进行图标化或去图标化，而不管它们的主窗口是被图标化或是被去图标化的。

在一些长期窗口共同工作的情况下，可以用 wm group 命令告知窗口管理器关于组的信息。下述脚本告知了窗口管理器，窗口.top1、.top2、.top3 和.top4 被当作一个组共同工作，且.top1 是该组的组长。

```
foreach i { .top2 .top3 .top4} {
    wm group $i .top1
}
```

接下来，窗口管理器会将组视为一个单元，且它能对组长进行一些特殊的操作。例如，在组长被图标化时，组中的所有其他窗口可能从显示器上删除，而不显示图标：此时，组长代表了整个组。在组长的图标被再次被去图标化时，组中所有窗口可能在显示器上重新显示。对组的精确处理取决于窗口管理器，而不同的窗口管理器会以不同的方式来处理它们。一个组的组长不需要真正显示在屏幕上(例如，它能被关闭)。

在一些罕见的情况下，需要让窗口管理器彻底忽略一个顶层窗口：没有装饰，没有窗口管理器对窗口的交互式操作，没有图标化，等等。此类窗口的经典示例是悬浮式帮助窗口或闪现屏幕。此时，使用如下命令可以将窗口标记为覆盖-重定向。

```
wm overrideredirect .splash true
```

该命令必须在窗口显示在屏幕上前调用。

提示：菜单会自动地将自己标记为覆盖-重定向，所以无需自己去标记。

26.8　针对系统的窗口属性

每个窗口系统都有独一无二的特征，这些特征很难在其他窗口系统中找到等价物。wm attribute 命令用于查询和设置这些独有特征。

```
wm attribute window ?attribute? ?value attribute value ...?
```

现在，所有窗口系统都支持下述属性。

- -fullscreen *boolean*
  值为真时，窗口会填满整个屏幕，而没有任何窗口管理器装饰。该特征经常与-topmost 协同工作。
- -topmost *boolean*
  值为真时，窗口总是显示在最前面。

在 Windows 系统中，还支持以下属性。

- -alpha *number*
  控制顶层窗口的透明度。有效值范围为 0.0(透明)到 1.0(不透明)。
- -disabled *boolean*
  值为真时，窗口处于无效状态，且不可以拥有焦点。
- -toolwindow *boolean*
  值为真时，窗口会采用工具窗口的样式。
- -transparentcolor *color*
  指定顶层窗口的透明色。如果指定为空字符串(默认)，则不使用任何透明色。

在 Mac OS X Aqua 中，还支持以下属性。

- -alpha *number*
  控制顶层窗口的透明度。有效值范围为 0.0(透明)到 1.0(不透明)。
- -modified *boolean*
  值为真时，窗口的关闭图标显示为修改状态。它经常用于显示应用程序修改了数据，且未保存这些数据。
- -notify *boolean*
  值为真时，它使停靠栏中的图标弹出，并且如果启用的话，触发声音提示，以表明应用程序需要注意。
- -titlepath *pathname*
  当被设置时，指定窗口标题栏中的图标引用的文件路径，可以拖曳和释放该图标，以代替文件的 Finder 图标。

在 X11 中，还支持以下属性。

- -zoomed *boolean*
  值为真时，将窗口最大化，就像使用 Windows 和 Mac OS X 系统的 zoomed 的 wm state 那样。

26.9 可停靠的窗口

wm manage 命令用来将任何窗口变为由窗口管理器管理的顶层窗口，而 wm forget 命令执行了相反的操作——通知窗口管理器停止管理该窗口。尽管该命令适用于任何组件，但它最适合用于顶层窗口和框架组件。这些命令会产生一个“停靠”操作(两个顶层窗口合起来成为单个顶层窗口)和一个“取消停靠”操作(从顶层窗口中删除一个子窗口，将其变为一个单独的顶层窗口)。单靠这些命令并不能完成整个操作，但若加上另一个几何管理

器，命令就可以完成这个任务了。典型的停靠操作可能如下：

```
wm forget .top.frame1
grid .top.frame1 -row 0 -column 1
```

取消停靠的操作可能如下：

```
grid forget .top.frame1
wm manage .top.frame1
```

在使用 wm manage 命令时，必须注意以下限制。

- 在“停靠”时，该组件必须是几何管理器的主组件的子组件。
- 在不由窗口管理器管理时，顶层窗口像框架一样起作用，所以仅有组件的框架属性适用。这就意味着-menu 设置被忽略，且菜单栏在窗口中不再可用。当顶层窗口重新由窗口管理器控制时，它会重新出现。
- 只有顶层组件可以拥有菜单栏，因为这是唯一一个带-menu 选项的 Tk 组件。
- 由停靠或非停靠窗口的子窗口明确设置的绑定标记必须重新设置或更新。这是因为绑定标记包含了顶层窗口的组件路径名。由于操作改变了顶层窗口的路径，旧的绑定标记将不再合适。(绑定标记的更多信息可参见 22.8 节。)

26.10 关闭窗口

Tcl 脚本可以使用 destroy 命令来关闭一个顶层窗口，或者在用户按下装饰框架上的关闭按钮时，由窗口管理器来关闭。在窗口管理器关闭窗口时，有时需要通告应用程序，这样就可能协同进行其他动作，或者阻止操作。这个操作可以用 wm protocol 命令截获。在 X11 中，执行了很多窗口管理器协议，但唯一一个在所有平台和窗口管理器中重要且能识别出的是 WM_DELETE_WINDOW 协议。窗口管理器在它希望应用程序删除窗口时会调用该协议，如在用户请求窗口管理器关闭窗口时。

无论何时触发了一个特殊的协议，wm protocol 命令都会执行一段脚本。例如，每当窗口管理器要求应用程序关闭主窗口时，命令

```
wm protocol . WM_DELETE_WINDOW {
    set response [tk_messageBox -type yesno \
                    -message "Really quit?"]
    if {$response eq "yes"} {
        destroy .
    }
}
```

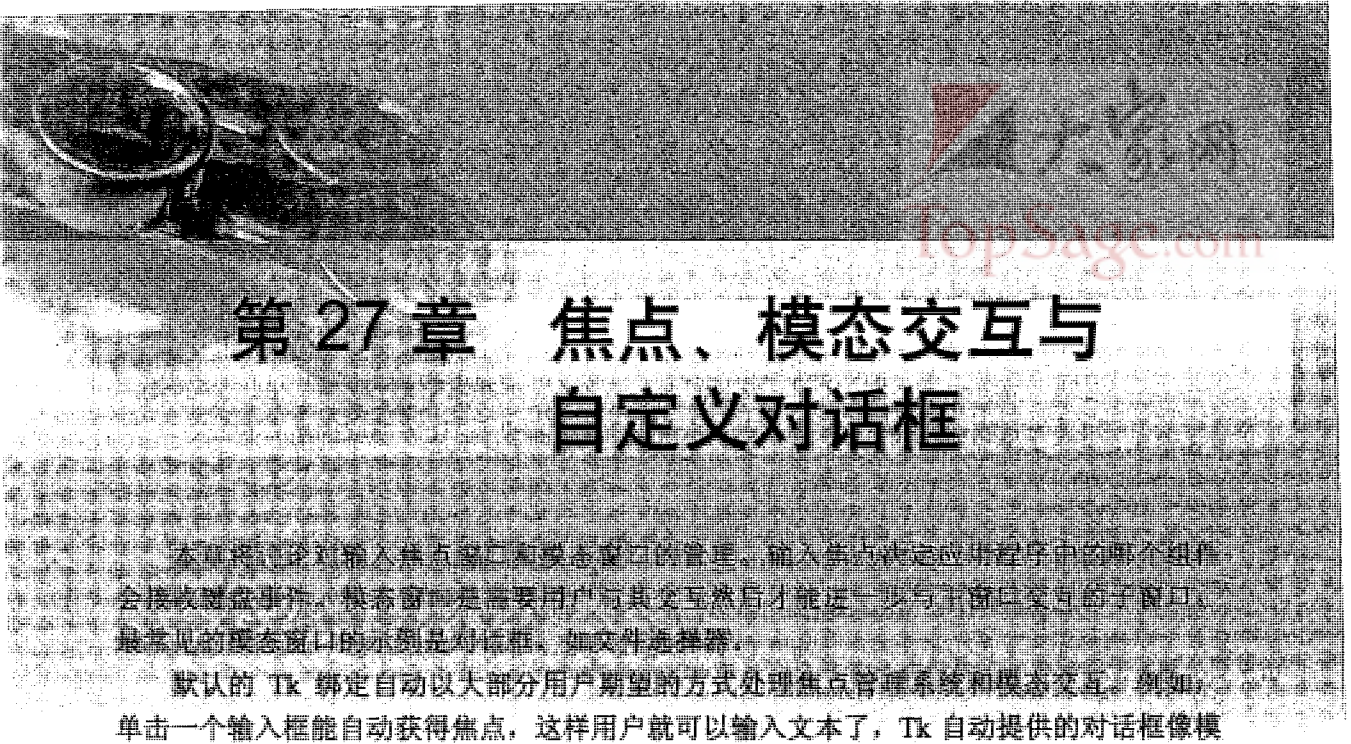
会提示用户确认退出。如果用户选择“否”，窗口就不会真的被删除。如果未使用 wm protocol 命令为顶层窗口上的 WM_DELETE_WINDOW 注册一个处理器，那么 Tk 会默认删除窗口。对其他协议而言，除非您指定了一个明确的处理器，否则什么也不会发生。

26.11 会话管理

一些窗口管理器管理用户会话，记录会话关闭时(例如，当用户从系统中注销)哪个应用程序正在运行并在建立新会话时启动那些相同的应用程序。两个特殊的 wm 命令可以提供必要的信息，这样，窗口管理器可以成功地保存和恢复 Tk 应用程序。wm client 命令提供了正在运行应用程序的主机名。wm command 命令提供一个带参数的程序路径名(例如，一个完整的命令行)，会话管理器可用该路径名来重启应用程序。例如：

```
wm client . sprite.berkeley.edu
wm command . {browse /usr/local/bin}
```

表明应用程序正运行在 sprite.berkeley.edu 机器上，且它由 shell 命令 browse/usr/local/bin 调用。



第27章 焦点、模态交互与自定义对话框

本章将讨论对输入焦点窗口和模态窗口的管理。输入焦点决定应用程序中的哪个组件会接收键盘事件。模态窗口是需要用户与其交互然后才能进一步与主窗口或其他子窗口交互的子窗口。最常见的模态窗口的示例是对话框，如文件选择器。

默认的 Tk 绑定自动以大部分用户期望的方式处理焦点管理系统和模态交互。例如，单击一个输入框能自动获得焦点，这样用户就可以输入文本了，Tk 自动提供的对话框像模态窗口一样起作用。一般而言，唯一您可能明确地管理焦点和模态交互的情况是：需要为应用程序创建自定义对话框时。

27.1 本章出现的命令

本章描述了下列用来操控焦点、模态交互和自定义对话框的命令。

- `focus`
  返回包含应用程序主窗口的显示器上的焦点窗口路径名；如果该程序上的窗口在显示器上都没有焦点，则返回空字符串。
- `focus -displayof` *window*
  返回包含 *window* 的显示器上的焦点窗口名。如果 *window* 的显示器的焦点窗口不在该应用程序中，则返回空字符串。
- `focus -lastfor` *window*
  返回在 *window* 顶层窗口的所有窗口中最近拥有输入焦点的窗口名。如果该顶层窗口中没有窗口有过输入焦点，或者最近使用的焦点窗口被删除了，则返回顶层窗口名。
- `focus ?-force ?` *window*
  设置应用程序的焦点窗口为 *window*。-force 选项也会强制应用程序拥有焦点；该特征应该慎用。
- `tk_focusFollowsMouse`
  将 Tk 转为一个隐含焦点模式，此时若鼠标进入窗口，则该窗口会拥有焦点。
- `tk_focusNext` *window*
  按焦点顺序返回 *window* 的下一个窗口。

- `tk_focusPrevious` *window*
  按焦点顺序返回 *window* 的前一个窗口。
- `grab ?-global?` *window*
  与 grab set 命令相同。
- `grab current ?`*window*`?`
  返回 *window* 显示器当前的攫取窗口名，当该显示器没有攫取，则返回空字符串。如果 *window* 被忽略，它会为所有显示器返回由该应用程序攫取的所有窗口。
- `grab release` *window*
  如果存在对 *window* 的攫取，则释放它。
- `grab set ?-global?` *window*
  设置 *window* 的攫取，释放 *window* 的显示器之前的任何攫取。如果指定了 -global，则攫取就是全局的，否则仅限于局部。
- `grab status` *window*
  如果 *window* 上没有设置攫取，则返回 none；如果设置了本地攫取，则返回 local；设置了全局攫取，则返回 global。
- `tkwait variable` *varName*
  在变量 *varName* 的值改变时起作用，然后返回。
- `tkwait visibility` *window*
  在 *window* 的可视状态改变时起作用，然后返回。
- `tkwait window` *window*
  在 *window* 被删除时起作用，然后返回。

27.2 输入焦点

在任何给定的时间，应用程序的一个窗口被指定为输入焦点窗口，或简称为焦点窗口。应用程序接收到的所有按键输入会导入焦点窗口，并根据事件绑定来处理；如果焦点窗口没有对按键的绑定，则该按键输入会被忽略。

提示： 焦点窗口只用于按下按键和释放按键的事件。与鼠标关联的事件，如光标进入，离开，按下按钮，释放按钮，总会被传给鼠标下的窗口，不管该窗口是否为焦点窗口。而且，焦点窗口只决定了一次按键事件到达一个特殊应用程序的结果，它不决定由显示器上的哪个应用程序接收按键。焦点应用程序的选择由窗口管理器来完成，而不是 Tk。

27.2.1 焦点模式：显式与隐式

处理输入焦点有两种方式，分别称作隐式焦点模式和显式焦点模式。在隐式模式中，焦点会跟随鼠标：按键输入定向到鼠标指针下的窗口，当鼠标从一个窗口移到另一个窗口，焦点窗口会暗中切换。在显式模式中，焦点窗口被明确地设置且不会改变，直到它被

明确地重新设置，鼠标运动不会自动改变焦点。

由于以下原因，Tk 使用显式焦点模式。第一，当您正在窗口中打字时，显示模式允许您将鼠标光标移到远处；而在隐式模式中，当您正在窗口中打字时，必须使鼠标一直处在窗口内。第二，也是更重要的一条，显式模式允许应用程序改变焦点窗口，无需用户移动鼠标。例如，在应用程序弹出需要输入的对话框时(如询问文件名)，它可以将输入焦点设置在对话框里合适的窗口中，无需您移动鼠标，且在使用完对话框后，它可以将焦点转回开始那个窗口。这样您可以将手一直放在键盘上。同样，在一个表格中输入时，每当按下 Tab 键，应用程序都可以将焦点转向表格里的下一个输入框，这样可以把手一直放在键盘上，高效率地工作。第三，如果想使用隐式焦点模式，可以通过事件绑定实现，指定在每次鼠标光标进入一个新窗口时，切换焦点。Tk 为隐式焦点模式提供了 tk_focusFollowsMouse 命令：它重新配置 Tk 绑定，这样每当鼠标进入一个窗口时,该窗口就会被设为焦点窗口。

Tk 应用程序无需经常担心输入焦点，因为对面向文本的组件的默认绑定已经考虑了最常见的情形。例如，在输入框或文本组件上单击鼠标按钮 1 时，该组件会自动成为焦点窗口。作为应用程序设计者，只有在先前介绍的情况下(需要在应用程序的众多窗口中移动焦点，来反应工作流)，您才需要设置焦点。然而您应该在这样做时谨慎一些，因为大部分窗口系统采用显式焦点模式。改变模式会使您的用户感到困惑。

27.2.2 设置输入焦点

要设置输入焦点，可以调用以组件名作为参数的 focus 命令。

```
focus .dialog.entry
```

它将应用程序接收的连续按键输入.dialog.entry；之前的焦点窗口不再接收按键。新的焦点窗口以高亮方式显示，如闪烁的插入光标，以表明该窗口拥有焦点，且之前的焦点窗口停止高亮显示。

提示： focus -force 选项也试图强制应用程序拥有焦点，并不是所有的窗口管理器都会实现这种请求。通常，应该谨慎采用此强制方式使应用程序拥有焦点，这就像在宴会上走向一个人并大喊：“关注我！”

27.2.3 查询输入焦点

有时候查询当前焦点窗口是有用的，尤其在应用程序需要暂时将焦点转给另一个窗口时。focus 命令返回应用程序的当前焦点窗口名，如果应用程序中的所有窗口都不是焦点窗口，则返回一个空字符串。相反，focus -lastfor 会返回包含特定窗口的顶层窗口中最近使用的焦点窗口名，该窗口就是在窗口管理器将焦点赋予顶层窗口之前的那个接受输入焦点的窗口。

27.3 模态交互

通常 Tk 应用程序的用户对要执行的操作和执行的顺序有完全的控制。应用程序提供了多种面板和控件，用户可以在其中做选择。然而，有时限制用户的选择范围或强制用户按一定顺序执行操作是有益的，这被称为模态交互。模态交互的最好示例是对话框：应用程序正在执行一些用户请求执行的程序，如向一个文件写信息时，此时它发现需要来自用户的其他输入，如要写入的文件的名字。它会显示一个对话框并强制用户响应对话框，例如在继续执行操作前选择一个文件名。在用户作出响应后，应用程序完成了操作并返回操作的普通模式，在这种模式中，用户可以进行任何操作。

在模态交互中，Tk 提供了两种机制。第一，grab 命令允许您暂时限制用户，这样他可以与很少一部分应用程序的窗口交互作用(例如，只有对话框)。第二，tkwait 命令允许您搁置对脚本的处理，直到一个特定的事件发生(如用户响应对话框)，在事件发生后，会接着执行脚本。

提示： *在这里需要给出一些警告。尽管模态交互有时是有用的，但大多数专家认为应该尽可能少地使用。如果选择范围总是被限制，用户会变得很恼火，且切换模式会使人困惑。*

27.3.1 攫取

鼠标事件，如按下按钮和鼠标运动，通常会传给指针下的窗口。然而，窗口可能声明对鼠标的所有权，这样鼠标事件只会传给那个窗口和其在 Tk 层级结构中的下层窗口。这便是攫取。当鼠标在攫取子目录树的一个窗口上，鼠标事件会被传送和处理，就像攫取没有效果一样。当鼠标在攫取子目录树之外，按钮的按下和释放，鼠标运动等事件就会传给攫取窗口，而不是鼠标下的窗口，同时窗口的输入输出事件会被丢弃。因此，攫取会阻止用户与攫取子目录树之外的窗口交互作用。

grab 命令设置和释放攫取。例如，如果您已经创建了一个名为.dlg 的对话框，且希望阻止用户和除.dlg 与其子窗口外的任何窗口交互作用，那么可以调用以下命令：

```
grab set .dlg
```

在用户响应了对话框后，可以用下述命令释放攫取：

```
grab release .dlg
```

如果对话框在用户响应完后被删除，就没有必要调用 grab release 了：当攫取窗口被删除后，Tk 会自动释放攫取。

为看出攫取如何工作，可以尝试下述脚本：

```
button .b1 -text "Unclickable 1"
button .b2 -text "Unclickable 2"
button .b3 -text "Grabby Button" -command { destroy .b3}
label .l -text "Text entry"
entry .e
```

```
pack .b1 .b2 .b3 .l .e
grab .b3
```

在该脚本中，您可以只与一个组件交互作用，即按钮.b3，它对输入事件存在攫取。您不可以将键盘焦点赋予输入框或输入文本。如果单击.b3(贴有“Grabby Button”的标签)，则回调脚本会删除该按钮。这最终会允许您与其他按钮和输入框组件交互作用。

攫取最常见的方式是在顶层窗口中设置攫取，这样在攫取时就只有一个面板或对话框是活动的。然而，攫取子目录树也可能包含其他顶层窗口，此时，所有对应于那些顶层窗口的面板或对话框都会是活动的。

提示： 如果试图对一个不可见的窗口设置攫取，一些窗口管理器会使 grab 失效或者引发一个错误。27.3.4 节会介绍如何使用 tkwait visibility 命令来使一个窗口可见。

27.3.2　局部和全局攫取

Tk 提供了两种形式的攫取：局部和全局。局部攫取只影响正在攫取的应用程序：如果用户将指针移入显示器上另外的应用程序，他就可以像通常一样与另一个应用程序交互作用。建议尽量使用局部攫取，它们是 grab set 命令的默认设置。全局攫取接管了整个显示器，所以您只能与设置攫取的应用程序交互作用。如要要求全局攫取，可以为 grab set 指定一个-global 开关，例如：

```
grab set -global .dlg
```

很少需要全局攫取，而且它们也很难使用(如果忘了释放攫取，显示器会被锁定，无法使用)。Tk 内部使用全局变量的一个地方是下拉菜单。

提示： 如果需要在应用程序中执行全局攫取，那么就可以安装一种后门来重新恢复控制或终止程序，以免不小心锁定了显示器。例如，可以在 all 绑定标记上安装一个关键绑定，或设置一个 after 事件处理器来在适合的时候退出程序。

27.3.3　攫取中的键盘处理

局部攫取对键盘的处理方式没有任何作用：应用程序中任何地方获得的按键输入像平常一样送给焦点窗口。最可能的是当您设定攫取时，应当将焦点赋予攫取子目录树中的一个窗口。攫取子目录树外的窗口不能获得任何鼠标事件，所以它们很可能不会声明焦点离开了攫取子目录树。因此，攫取有可能达到这样的效果：将键盘焦点只赋予攫取子目录树，然而，您可以任意移动焦点。如果将指针移向另一个应用程序，焦点会移向该应用程序，就好像没有攫取一样。

在全局攫取时，Tk 会设置键盘上的一个攫取，这样键盘事件会输入攫取应用程序，即使此时指针在其他应用程序上。这意味着您不可以使用键盘来与任何其他应用程序交互作用。一旦键盘事件到达攫取应用程序，它们就会以通常的方式被传给焦点窗口。

27.3.4 等待：tkwait 命令

模态交互的第二个方面是等待。通常，您想在模态交互过程中搁置一个脚本，并在交互完成时恢复执行。例如，如果在写文件操作过程中显示了一个文件选择对话框，您很可能希望等待用户对对话框作出回应，之后再用对话框交互提供的名称完成文件写操作。或者当启动一个应用程序时，您可能希望显示一个引导面板来描述该应用程序，并在程序初始化过程中使该面板始终可见；在开始主初始化过程前，您希望确保面板一直在屏幕上可见。tkwait 命令可以用于类似情况中的等待。

tkwait 有三种格式，每一种都等待另一个事件发生。第一种格式等待一个窗口被删除，命令如下：

```
tkwait window .dlg
```

直到.dlg 被删除该命令才会返回。您或许在建立一个对话框且设置攫取之后调用该命令；该命令直到用户与对话框交互，并将其删除后才会返回。当 tkwait 正在等待时，应用程序会响应事件，这样用户就可以与应用程序窗口交互了。在对话框示例中，一旦与用户的响应完成(例如用户单击 OK 按钮)，绑定就必须存在，以删除对话框。对话框的绑定也可能将其他信息保存在变量中(例如文件名或按下按钮的标识符)。当 tkwait 返回后，这些信息就会可用了。

下面的脚本创建了一个面板，上面有两个标签分别为 OK 和 Cancel 的按钮，等待用户单击其中的一个，来删除该面板。

```
toplevel .panel
    button .panel.ok -text OK -command {
        set label OK
        destroy .panel
    }
    button .panel.cancel -text Cancel -command {
        set label Cancel
        destroy .panel
    }
    pack .panel.ok -side left
    pack .panel.cancel -side right
    grab set .panel
    tkwait window .panel
    puts "You clicked $label"
```

当 tkwait 命令返回时，变量 label 会包含您单击的按钮的标签。

tkwait 的第二种格式等待窗口可视状态的改变。例如，命令

```
tkwait visibility .intro
```

直到.intro 的可视状态改变时才会返回。通常，该命令在创建一个新窗口后会被调用，此时，它只在窗口开始在屏幕上可见时才会返回。tkwait visibility 可以用于在对窗口设置攫取前等待该窗口变为可见，或者确保在调用一个长初始化脚本前，引导面板始终在屏幕上。就像 tkwait 的所有格式那样，tkwait visibility 会在等待中响应事件。

tkwait 的第三种格式与 vwait 命令相同。在这种格式下，命令只当一个给定变量被修

改时才会返回。例如，命令

```
tkwait variable x
```

在变量 x 被修改前不会返回。这种形式的 tkwait 通常与修改变量的事件绑定合用。例如，下述过程使用 tkwait variable 来执行与 tkwait window 相似的操作，不同之处在于可以指定多个窗口，且一旦任何命名窗口被删除，该过程就会返回(它返回被删除窗口的名称)。

```
proc waitWindows args {
    global dead
    foreach w $args {
        bind $w  <Destroy> "set dead $w"
    }
    tkwait variable dead
    return $dead
}
```

27.4 自定义对话框

Tk 包含了一些内置对话框，例如 tk_getOpenFile，如果它们符合您的需求，您就应该使用。Tk 提供的内置对话框的介绍可参见 18.14 节。如果它们都不适用，您可以创建自定义对话框。自定义对话框应该是一个顶层窗口。您还应该使用下述命令，将顶层窗口设为应用程序主窗口的瞬时窗口，如第 26 章所述。

```
wm transient .dialog.
```

该命令将顶层窗口.dialog 设置为代表主窗口“.”的一个瞬时窗口，这样窗口管理器可以为对话窗口选择针对对话的装饰。

提示： 如果父窗口被关闭或被图标化，一些窗口管理器会将创建的窗口设为关闭状态。结合对话窗口上的攫取，可以暂停整个应用程序。因此，最安全的方式是当且仅当父窗口可见时，才使对话框变为瞬时。可以使用 winfo viewable 命令完成上述操作，例如：

```
if {[winfo viewable .]} {
  wm transient .dialog
}
```

如果对话框中包含多个组件，可以在所有组件准备好前隐藏窗口。对于窗口管理器而言，就是撤销窗口并在稍后将其去图标化(所有这些术语都来自 X 窗口系统)。例如：

```
toplevel .dialog
wm withdraw .dialog

# Create and manage all the widgets ...

wm deiconify .dialog
```

您也应该设置标题和一个脚本，来处理关闭按钮(详情参见第 26 章)。例如：

```
wm title .dialog "Create User Account"
wm protocol .dialog WM_DELETE_WINDOW "cancelDialog"
```

该例在用户单击窗口的关闭按钮时，会调用 cancelDialog 过程。如果对话框有 Cancel 按钮，建议为该按钮使用相同的代码。实际上，您可以简单地调用这个例子中 Cancel 按钮的脚本，例如：

```
wm protocol .dialog WM_DELETE_WINDOW {
    .dialog.cancel invoke
}
```

对数据输入组件，您通常会使用一个基于网格的布局来管理组件的位置。网格的使用允许提示，也允许数据输入组件排列整齐。如果希望该对话框成为模态，可以使用本章介绍的技术。例如：

```
wm deiconify .dialog
tkwait visibility .dialog
grab set .dialog
tkwait window .dialog
```

这些命令会一直等待，直到对话框被删除。

提示： 在一些窗口管理器上，tkwait 或 grab set 命令有可能引发错误。因此，使用 catch 命令来运行它们是最安全的，该命令会静默地忽略这些错误。

下述脚本将所有这些小片段组合起来，创建一个简单的主窗口和一个模态对话框。此时，对话框会将输入集中起来，为某假定的 Internet 站点设置一个用户账户。为使示例简单，主窗口显示了两个按钮：一个显示 Create Account 对话框，另一个用来退出应用程序。脚本如下：

```
# Tk custom dialog example

proc cancelDialog {} {
    destroy .dialog
}

proc createAccount {} {
    global username password firstname lastname
    puts "Create account for $firstname $lastname."
    puts "With user name: $username password $password."

    destroy .dialog
}

proc showCreateAccount {}  {
    global username password firstname lastname
    set firstname ""
    set lastname ""
    set username ""
    set password ""

    toplevel .dialog
    wm withdraw .dialog

    ttk::frame .dialog.f -relief flat

    # Login
    ttk::labelframe .dialog.f.login -text "Login:"
    ttk::label .dialog.f.login.user1 -text "User Name:"
```

```
    ttk::entry .dialog.f.login.usert -textvariable username
    ttk::label .dialog.f.login.passl -text "Password:"
    ttk::entry .dialog.f.login.passt -show "*" \
        -textvariable password

    grid config .dialog.f.login.userl \
        -column 0 -row 0 -sticky e
    grid config .dialog.f.login.passl \
        -column 0 -row 1 -sticky e
    grid config .dialog.f.login.usert \
        -column 1 -row 0 -sticky snew
    grid config .dialog.f.login.passt \
        -column 1 -row 1 -sticky snew

    pack .dialog.f.login -padx 5 -pady 10

    # User information
    ttk::labelframe .dialog.f.userinfo \
        -text "User Information:"
    ttk::label .dialog.f.userinfo.first1 -text "First Name:"
    ttk::entry .dialog.f.userinfo.firstt \
        -textvariable firstname
    ttk::label .dialog.f.userinfo.last1 -text "Last Name:"
    ttk::entry .dialog.f.userinfo.lastt \
        -textvariable lastname

    grid config .dialog.f.userinfo.first1 \
        -column 0 -row 0 -sticky e
    grid config .dialog.f.userinfo.last1 \
        -column 0 -row 1 -sticky e
    grid config .dialog.f.userinfo.firstt \
        -column 1 -row 0 -sticky snew
    grid config .dialog.f.userinfo.lastt \
        -column 1 -row 1 -sticky snew

pack .dialog.f.userinfo -padx 5 -pady 10

# Action buttons
ttk::frame .dialog.f.buttons -relief flat

ttk::button .dialog.f.buttons.ok -text "Create Account" \
    -command {createAccount}
ttk::button .dialog.f.buttons.cancel -text "Cancel" \
    -command {cancelDialog}

pack .dialog.f.buttons.ok -side left
pack .dialog.f.buttons.cancel -side right
pack .dialog.f.buttons -padx 5 -pady 10
pack .dialog.f

        # Window manager settings for dialog
        wm title .dialog "Create User Account"
        wm protocol .dialog WM_DELETE_WINDOW {
            .dialog.f.button.cancel invoke
        }
        wm transient .dialog .

        # Ready to display the dialog
        wm deiconify .dialog

        # Make this a modal dialog.
```

```
        catch {tk visibility .dialog}
        focus .dialog.f.login.usert
        catch {grab set .dialog}
        catch {tkwait window .dialog}
}

ttk::button .createAccount -text "Create Account ... " \
-command {showCreateAccount}

ttk::button .exitButton -text "Exit" -command { exit }

pack .createAccount .exitButton -padx 10 -pady 20
```

在该脚本中，cancelDialog 过程删除了对话框窗口。createAccount 过程对于创建新用户账户的实际代码来说起到了占位符的作用。图 27.1 显示了主窗口。showCreateAccount 过程在用户单击 Create Account 按钮时会创建并填充创建账户对话框。图 27.2 显示了这个对话框。在 Create User Account 对话框可见时，主窗口不接受键盘或鼠标输入。用户接着可以输入账户信息，包括以星号显示的密码。

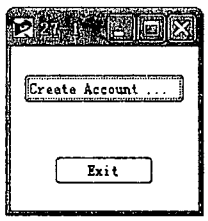


图 27.1　主窗口

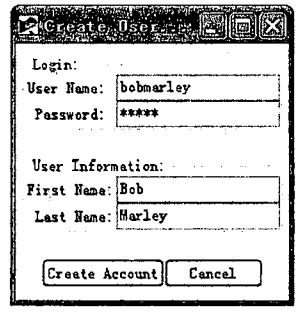
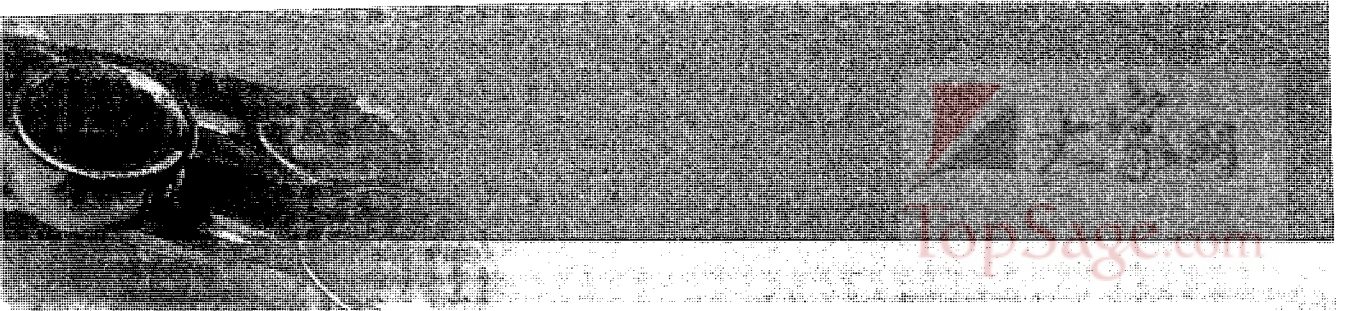


图 27.2　自定义的创建账户对话框

提示： 另外一种方法是在建立对话框后，用 wm deiconify 和 wm withdraw 来按需要显示和隐藏对话框。如果是特别复杂的对话框，这能节省时间。它也同时允许组件保持(甚至可以更新)它们的状态，即使是在隐藏的时候，这种优点在有些方面很有用，如偏好设置对话框。



第28章 更多配置选项

前面已经介绍过组件的配置选项。在创建新组件时，可以指定配置选项，并且组件命令的 configure 动作来修改它。本章将介绍与选项相关的另两个内容。本章第一部分会介绍选项数据库，它可以指定选项的默认值。第三部分会介绍 configure 组件命令的元数据功能，它可用于检索选项值默认及修改选项。

28.1 本章出现的命令

本章将讨论如下管理配置选项的命令。对特定组件类有效的所有选项，可参见相应类命令(如 button)的参考文档。

- *class widget ?optionName value optionName value ...?*
  用类 *class* 和路径名 *widget* 创建一个新的组件，并将新组件选项设置为由 *optionName value* 对给出的值。未指定的选项由选项数据库或组件默认值填充。返回 *widget* 作为结果。
- *widget* cget *optionName*
  返回当前赋予 *widget* 的选项 *optionName* 的值。
- *widget* configure
  返回一个列表，其中的元素是描述 *widget* 的所有选项的子列表。每个子列表都描述了采用后文所述格式的某个选项。
- *widget* configure *optionName*
  返回一个列表，它们用于描述 *widget* 的选项 *optionName*。该列表通常包含 5 个值：*optionName*、选项数据库中该选项的名字、它的类、它的默认值和它的当前值。如果该选项是另一个选项的同义词，那么该列表会包含两个值：选项名和该选项名的数据库名同义词。
- *widget* configure *optionName value ?optionName value ...?*
  将相应的 *value* 赋予 *widget* 的每一个 *optionName* 值。
- option add *pattern value ?priority?*
  向由 *pattern* 和 *value* 指定的选项数据库添加新选项。*priority* 必须要么是一个 0～100 的数字，要么是一个符号名称(符号名称的细节可参阅参考文档)；如果被忽略了，它的默认值为 interactive(80)。

- option clear
  从选项数据库中删除所有条目。
- option get *widget dbName dbClass*
  如果选项数据库包含一个和 *widget*、*dbName* 和 *dbClass* 匹配的模式，则返回最匹配的模式的值。否则返回一个空字符串。
- option readfile *fileName ?priority?*
  读取 *fileName*，它必须采用.Xdefaults 文件的标准格式，并向处于优先级 *priority* 的选项数据库添加所有由 *fileName* 文件指定的选项。如果没给出 *priority*，则优先级会默认为 interactive(80)。

28.2 选项数据库

选项数据库提供了配置选项的值，这些选项没有由应用程序设计者明确的指定。在创建组件时，会询问选项数据库：对于命令行上的每个未指定的选项，组件会查询选项数据库，并使用数据库里的值，如果存在的话。如果选项数据库里没有值，那么组件类会提供一个默认值。选项数据库中的值通常由用户提供给个性化的应用程序，例如，要指定统一的大号字体。在 Unix 系统中，Tk 支持 RESOURCE_MANAGER 属性和.Xdefaults 文件，支持的方式与其他 X 工具箱一样。

在 Tk 应用程序中，应该不需要频繁使用选项数据库，因为组件选项都有适当的默认值。如果一定需要改变选项值，那么从 Tcl 脚本中调用 configure 组件命令来完成这种改变，通常会比在.Xdefaults 文件里创建条目更简便。选项数据库主要用于提供与其他 X 工具箱的兼容，应该尽量少使用它。

28.3 选项数据库条目

选项数据库包含任意数量的条目，每个条目都由两个字符串组成：一个模式和一个值。模式指定了一个或多个组件和选项，而值是一个字符串，用在与模式匹配的选项中。

模式最简单的格式是由一个应用程序名、一个可选组件名和一个选项名组成，它们之间用点隔开。例如，以下是两个采用这种格式的模式。

```
wish.a.b.foreground
wish.background
```

第一个模式应用于应用程序 wish 的组件.a.b 的 foreground 选项。在第二个模式中，组件名被忽略了，所以模式应用于 wish 的主组件。每个模式都只应用于单个组件的单个选项。

模式也能包含类或者通配符，这样就能与很多不同的选项或组件匹配。组件名的任何部分都能被一个类代替，这时模式会与任意作为该类实例的组件相匹配。例如，下述模式应用于所有为复选按钮的.a 的子组件。

```
wish.a.Checkbutton.foreground
```

应用程序和选项名也能被类替换。应用程序名是可执行程序的名称(如果启动的是交互式解释器，该名称就是解释器名；如果创建了自执行的脚本，就是脚本名)；应用程序类是首字母大写的应用程序名。单独的选项也有类。例如，foreground 选项的类是 Foreground。一些其他的选项，例如 insertBackground(插入光标的颜色)也有这个类 Foreground，因此下述模式会应用于任一个为 wish 中.a 的子组件的任意输入框组件的任意选项。

```
wish.a.Entry.Foreground
```

最后，模式能包含“*”通配符。一个“*”与任意数量的窗口名或类匹配，如下面这个示例：

```
*Foreground
*Button.foreground
```

第一个模式应用于任何应用程序的任何组件的任何选项，只要该选项的类是 Foreground。第二个模式应用于任何应用程序的任何按钮组件的 foreground 选项。“*”通配符只能用于窗口或者应用程序名，它不可以用于选项名(对一个组件的所有选项而言，指定相同的值是没有多大意义的)。

提示：模式的语法和由 X 窗口系统中标准 X 资源数据库机制支持的语法是一样的。

一个选项的数据库名通常和您可能在组件创建命令或 configure 组件命令中使用的名称一样，不同之处在于没有前缀“-”，且大写字母用于标记内部单词的边缘。例如，-borderwidth 选项的数据库名称是 borderWidth。一个选项的类通常和它的数据库名相同，除了首字母要大写。例如，-borderwidth 选项的类是 BorderWidth。需要记住的是，在 Tk 中，类总是由一个大写字母开头；任何一个由大写字母开头的名称都视为一个类。

28.4　RESOURCE_MANAGER 属性和.Xdefaults 文件

当 Tk 应用程序开始运行，Tk 会自动对选项数据库进行初始化。对 X 窗口系统而言，如果在屏幕根窗口上有一个 RESOURCE_MANAGER 属性，那么数据库就从这里进行初始化。否则，Tk 会检查用户的主目录，看有没有.Xdefaults 文件，若有则使用该文件。无论初始化信息是从 RESOURCE_MANAGER 属性而来还是从.Xdefaults 文件而来，信息的格式都一样。这里描述的语法和其他 X 工具箱支持的语法是一样的。

初始化数据的每一行都在资源数据库中设定了一个条目，格式如下：

```
*Foreground: blue
```

这一行包含了一个模式(在这个示例里是*Foreground)，后面是一个冒号、一个空格和一个与模式关联的值(在这个示例里是 blue)。如果这个值太长，以至于无法填入一行，它可以显示为多行，除最后一行外的每一行的末尾都要有一个反斜线换行符。

```
*Gizmo.text : This is a very long initial \
```

```
value to use for the text option in all \
"Gizmo" widgets.
```

反斜线和换行符并不属于这个值。

空白行会被忽略掉，同样会被忽略掉的是以字符#或者！开头的行。

28.5 选项数据库的优先级

选项数据库中的几个模式可能与一个特定选项匹配。此时，Tk 使用一个由两部分组成的优先级方案来决定应用哪个。Tk 用来处理冲突的机制与 X 工具箱(Xt)支持的标准机制不一样。

通常，数据库中一个选项的优先级根据它们进入数据库的顺序来决定：新选项的优先级高于旧选项。在指定选项时(例如，将它们输入.Xdefaults 文件中)，应该首先指定更通用的选项，然后用更具体的选项去覆盖。例如，如果希望按钮组件有一个 Bisquel 背景颜色，同时让其他组件有白色背景，则可以将下述几行代码放入.Xdefaults 文件。

```
*background: white
*Button.background: Bisquel
```

*background 模式与任何与*Button.background 匹配的选项匹配，但*Button.background 的优先级更高，因为它是最后被指定的。如果模式的顺序颠倒过来，所有组件(包括按钮)就可能拥有白色背景，而*Button.background 模式可能会失效。

在一些情况下，不能在特定模式前指定通用模式(例如，您可能在选项数据库已被来自 RESOURCE_MANAGER 属性的一些特定模式初始化后，向其中添加更通用的模式)。为满足这些情况，每个条目也有一个介于 0～100(包括 0 和 100 本身)的整数优先级。优先级高的条目优先于优先级低的条目，无需考虑它们插入选项数据库的顺序。优先级在 Tk 中不会经常使用，它们如何工作的完整细节可参阅参考文档。

Tk 的优先级方案和其他 X 工具箱(如 Xt)所使用的方案不一样。Xt 为最具体的模式提供更高的优先级。例如，.a.b.foreground 比*foreground 更具体，所以它的优先级更高，而无需考虑模式出现的顺序。在大多数时候，这一点不是问题：您可以使用 Xt 规则为 Xt 应用程序指定选项，并用 Tk 规则为 Tk 应用程序指定选项。如果希望指定一些能同时应用于 Tk 应用程序和 Xt 应用程序的选项，那就使用 Xt 规则，但要确保按 Xt 规则具有更高优先级的模式在.Xdefaults 文件中的位置更靠后。通常，不应该为 Tk 应用程序指定太多选项(默认值应该总是合理的)，因此对模式优先级的给定不应该过于频繁。

提示： 选项数据库只用于查询组件创建命令中未明确指定的选项。这就意味着用户不可以覆盖任何组件创建命令指定的选项。如果想指定一个选项的值，但允许用户通过 RESOURCE_MANAGER 属性来覆盖该值，应该使用 option 命令来指定该选项的值，详情参见后文。

28.6　option 命令

option 命令用于在应用程序正运行时控制选项数据库。命令 option add 会在数据库中创建一个新条目，并带有两个或三个参数。头两个参数是新条目的模式和值，第三个参数(如果指定)是新条目的优先级。如果被忽略了，则优先级默认为 interactive(80)。例如：

```
option add *Button.background Bisquel
```

增加了一个条目，它将所有按钮组件的背景颜色设为 Bisquel。选项数据库的改变只影响调用 option add 的应用程序，它们只应用于执行 option add 后新创建的组件，数据库的改变不会影响已有组件。

option clear 命令会从选项数据库中删除所有的条目。在下一次访问数据库时，数据库会由 RESOURCE_MANAGER 属性或.Xdefaults 文件重新初始化。

option readfile 命令会读取一个采用早前介绍过的 RESOURCE_MANAGER 属性格式的文件，并为选项数据库中的每一行创建条目。例如，下述命令使用文件 newOptions 里的信息扩充了选项数据库。

```
option readfile newOptions
```

option readfile 命令也可以被赋予一个优先级，它被视为文件名之后的多余参数，用来指定新添加的选项的优先级。如果被忽略了，优先级默认为 interactive(80)。

要查询在选项数据库中是否有一个应用于特定选项的条目，可以使用 option get 命令。

```
option get .a.b background Background
```

该命令会带三个参数：一个组件的路径名(.a.b)、一个选项的数据库名(background)和该选项的类(Background)。该命令会搜索选项数据库，找出是否有任何条目与给定的窗口、选项和类匹配。如果有，最高优先级的匹配选项的值会返回。如果没有匹配的条目，则会返回一个空字符串。

28.7　configure 组件命令

每个组件类支持一个 configure 组件命令。该命令有三种格式，既可以改变选项值，也可以检索组件选项的信息。

如果 configure 组件命令带有两个其他参数，它会改变一个选项的值，示例如下：

```
.button configure -text Quit
```

如果 configure 组件命令只带有一个多余参数，则它会返回该命名选项的信息。

```
    .button configure -text
⇒  -text text Text { } Quit
```

该返回值通常是一个五元素列表。列表里的第一个元素是选项名，在创建或者配置一个组件时，可在 Tcl 命令行上指定它。第二和第三个元素是一个名称和一个类，用来在选项数据库中查找选项。第四个元素是组件类提供的默认值(在前面那个示例中的单个空格字

符)，第五个元素是选项的当前值。

一些组件选项只是其他选项的同义词(例如，按钮的-bg 选项和-background 选项的意义是一样的)。同义词的配置信息会作为一个二元素列表返回，列表中的两个元素分别是选项的命令名和它的选项数据库名同义词。

```
    .button configure -bg
⇒ -bg background
```

如果 configure 组件命令在没有其他参数时调用，它会返回一个列表，列表包含所有组件选项的信息，而每个元素就是每个选项的嵌套的信息子列表。

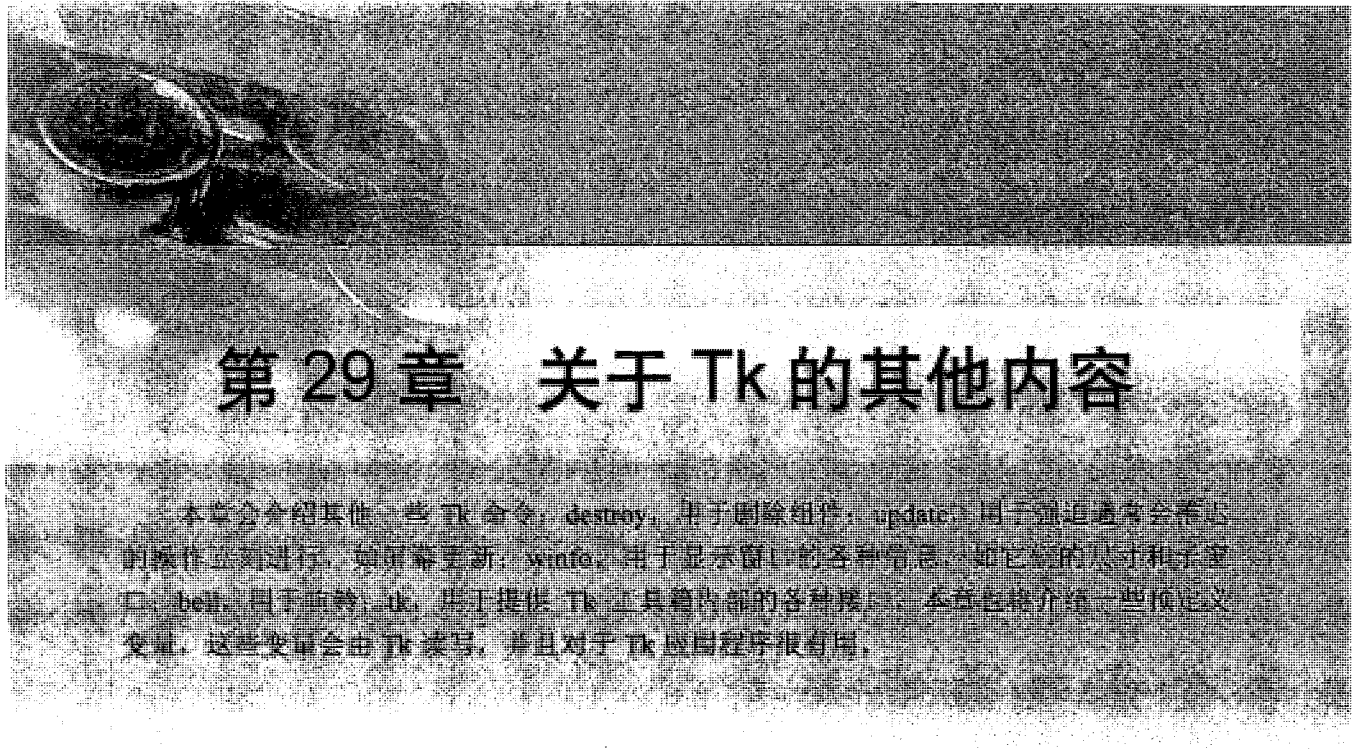
```
    .button configure
⇒ {-activebackground activeBackground Foreground
   systemButtonFacePressed systemButtonFacePressed}
   {-activeforeground activeForeground Background
   systemPushButtonPressedText systemPushButtonPressedText} {-anchor
   anchor Anchor center center} {-background background Background
   White White} {-bd -borderwidth} {-bg -background} {-bitmap bitmap
   Bitmap {} {} } {-borderwidth borderWidth BorderWidth 2 2 } {-command
   command Command {} {} } {-compound compound Compound none none}
   {-cursor cursor Cursor {} {} } {-default default Default disabled
   disabled } {-disabledforeground disabledForeground
   DisabledForeground #a3a3a3 #a3a3a3} {-fg -foreground} {-font font
   Font TkDefaultFont TkDefaultFont} {-foreground foreground
   Foreground systemButtonText systemButtonText} {-height height
   Height 0 0} {-highlightbackground highlightBackground
   HighlightBackground White White} {-highlightcolor highlightColor
   HighlightColor systemButtonFrame systemButtonFrame}
   {-highlightthickness highlightThickness HighlightThickness 4 4}
   {-image image Image {} {} } {-justify justify Justify center
   center} {-overrelief overRelief OverRelief {} {} } {-padx padX Pad
   12 12} {-pady padY Pad 3 3} {-relief relief Relief flat flat}
   {-repeatdelay repeatDelay RepeatDelay 0 0} {-repeatinterval
   repeatInterval RepeatInterval 0 0} {-state state State normal
   normal} {-takefocus takeFocus TakeFocus {} {} } {-text text Text {}
  Quit} {-textvariable textVariable Variable {} {} } {-underline
  underline Underline -1 -1} {-width width Width 0 0} {-wraplength
   wrapLength WrapLength 0 0}
```

28.8 cget 组件命令

每一个组件类都支持一个 cget 组件命令。这条命令将一个选项的名称作为一个参数来接收，且会返回该选项的当前值。例如：

```
    .button cget -text
⇒ Quit
    .button cget -background
⇒ White
    .button cget -padx
⇒ 12
    .button configure -padx 150
    .button cget -padx
⇒ 150
```

configure 组件命令在检索组件值时，会返回一个列表，cget 与其不同，只返回给定选项的值。



第29章 关于Tk的其他内容

本章介绍其他一些Tk命令：destroy，用于删除组件；update，用于强迫通常会等待的操作立刻进行；刷新屏幕更新；winfo，用于显示窗口的各种信息，如它们的尺寸和子窗口；bell，用于响铃；tk，用于提供Tk工具箱内部的各种接口。本章也将介绍一些预定义变量，这些变量会由Tk读写，并且对于Tk应用程序很有用。

29.1 本章出现的命令

本章将要讨论下述命令。

- `destroy` *`window ?window window ...?`*
  删除每个*window*和所有的下层窗口。相应的组件命令(和所有组件状态)也会被删除。
- `tk appname` *`?newName?`*
  返回当前应用程序的名称，或者将应用程序名设为*newName*。
- `tk inactive ?-displayof` *`window?`* `?reset?`
  返回自上次用户在包含*window*的屏幕上交互作用以来的时间，单位为毫秒，*window*默认为“.”。包含reset参数可以重新设定空闲计时器。
- `tk scaling ?-displayof` *`window?`* *`?number?`*
  返回包含*window*的屏幕的比例系数，*window*默认为“.”。比例系数是一个浮点数，表示每磅字的像素数目。
- `tk windowingsystem`
  返回当前的Tk窗口系统，值为x11(基于X11)、win32(MS Windows)或者aqua(Mac OS X Aqua)。
- `update ?idletasks?`
  及时更新显示器，并处理所有搁置的事件。如果指定了idletasks，除了那些在闲置任务队列(被推迟的更新)中的任务外不会执行别的事件。
- `winfo` *`option ?arg arg ...?`*
  返回窗口的各种信息，具体内容取决于option参数。细节可参见参考文档。

29.2 删除组件

destroy 命令可删除一个或多个组件。它可以将任意数目的组件名视为参数。例如：

```
destroy .dlg1 .dlg2
```

该命令会删除.dlg1 和.dlg2，包括它们的组件状态和在窗口之后命名的组件命令。它也能递归地删除它们的子组件。destroy .命令会删除应用程序中的所有组件，这时，大多数 Tk 应用程序仍存在。

29.3 update 命令

Tk 通常推迟一些操作，如屏幕更新，直到该应用程序被闲置。例如，如果调用了一个组件命令来改变按钮中的文本，按钮不会立刻重新显示。相反，它会让重新显示稍后进行，并立刻返回。在一些时候，程序会变成闲置，这意味着所有的已有事件都被处理完了，并且应用程序是在事件循环中，并正在等待另一个事件发生。这时，所有被推迟的操作都会完成。Tk 推迟重新显示的原因是在相同的窗口一次又一次被修改时，它会保存自己的工作：通过被推迟的重新显示，窗口可以只在最后才显示出来。Tk 也推迟了很多其他操作，如几何外观重新计算和窗口创建。

通常，推迟不可见。交互式应用程序几乎不会在同一时间进行很多工作，因此 Tk 就会很快变为闲置，并在用户察觉到任何推迟前更新屏幕。但是，有时候这种推迟是不方便的。例如，如果脚本运行需要很长时间，可能希望在运行过程中屏幕能在给定的时间更新。update 命令可以实现上述目的。如果调用以下命令：

```
update idletasks
```

所有被推迟的操作，例如重新显示都会立刻执行，该命令直到推迟的操作全部完成才会返回。

提示： 像 after 命令一样，update 命令通常是 Tk 的一部分。因此，它被移入 Tcl 核心语言，所以可以在应用程序中使用 update 命令，如网络服务器这一类不需要显示图形用户界面的应用程序。

下列过程使用 update 来同步闪现一个组件。

```
proc flash {w option value1 value2 inerval count} {
    for {set i 0} {$i < $count } {incr i} {
        $w config $option $value1
        update idletasks
        after $interval
        $w config $option $value2
        update idletasks
        after $interval
    }
}
```

这个过程会以一个给定的次数来闪现这个组件，但直到闪现结束才会返回该组件。Tk

在这个过程执行时不会变为闲置(当它在等待时，after 命令不返回到事件循环)，因此 update 命令需要用来强迫组件重新显示。如果没有 update 命令，在脚本完成前屏幕上不会显示任何改动，而完成时组件的选项会改为 value2。

如果在没有 idletasks 参数时调用了 update，所有搁置的事件也会处理。您可能需要在一个长期计算的过程中进行这样的操作，来允许应用程序响应用户的交互作用(例如，用户可以调用一个 Cancel 按钮来终止计算)。

提示： update 命令可以是危险的，因为它开始了 Tcl 事件循环的一个实例。如果脚本在一个事件处理器脚本中执行了一个 update 命令，那么它会真正开始一个事件循环的嵌套实例，该循环必须在外部事件循环重新受控前终止。这可以使您的信息流产生严重的问题，特别是在外部循环由一个 vwait 或 tkwait 命令启动时，如果它们正在寻找的事件在 nested 更新事件循环中出现，它们就不会“看见”该事件，因此就不会如预想的一样终止。您应该尽可能避免嵌套事件循环。一种办法是将一个长期的活动分解为小块，并由 after 命令让每一小块安排下一小块的执行。另一种办法是用 Thread 扩展来创建一个多线程 Tcl 应用程序，因此某个线程中的一个长期活动不会耽误另一线程的事件循环。

29.4　关于组件的信息

winfo 命令提供了组件的信息。它有大约 50 种不同的子命令，用来检索一个组件不同种类的信息。例如，winfo exists 会返回一个 0 或者 1 值，来显示一个组件是否存在。winfo children 会返回列表，列表的元素是该组件的子组件。winfo width 和 winfo height 会返回组件的当前尺寸。winfo class 会返回组件的类，如 Button 或者 Text。winfo 提供的所有选项的细节可参见参考文档。

29.5　tk 命令

tk 命令用于访问 Tk 内部状态的不同方面。从该命令获得的大部分信息与 Tk、应用程序或者整个显示器有关。

了解正在执行应用程序的窗口系统非常重要。历史上，人们会检测::tcl_platform(platform)数组元素的值，Tcl 可能将这个值设置为 windows、unix 或者 macintosh。然而，由于引入了 Mac OS X，该数组元素现在(正确地)将一个 Mac OS X 系统报告为 unix。tk windowingsystem 命令是当前确定窗口系统的正确方法。它会返回 aqua、win32 或 x11 中的一个值。

tk inactive 返回自用户上一次与系统交互作用后(如移动鼠标，按下一个按键等)的时间，单位为毫秒。

```
    tk inactive
⇒ 3
```

如果在一个安全的解释器上执行，或者在一个不支持查询非活动时间的系统上执行，tk inactive 会返回-1。您可以包含-displayof 选项，并指定显示器上的窗口名，来查询一个特定窗口的闲置时间。另外，您可以包含 reset 参数，来重设闲置计时器。

当像素和度量的物理单位之间正在进行转换时，tk scaling 会返回使用的比例因子。它会是一个浮点数，给出单位磅(1/72 英寸)上的像素数目，您可以用-displayof 选项指定一个特定的、您想检索其比例因数的显示器上的窗口名。

```
    tk scaling
⇒ 1.000492368291482
```

Tk 尽力来决定应用程序开始时合适的值，但系统经常不会给 Tk 一个真正精确的值。如果一个精确的比例因数对应用程序而言很重要，应该给用户提供精确度量显示器上的尺寸的方法，然后将新的比例因数作为一个参数传给 tk scaling，来设置应用程序显示器的比例因数。下例用 winfo screenmmwidth 命令检索窗口屏幕的计算宽度，单位为毫米。

```
    winfo screenmmwidth .
⇒ 677
    tk scaling 1.25
    winfo screenmmwidth .
⇒ 542
```

如果应用程序使用 Tk 的 send 命令来向其他 Tk 应用程序传递信息，您需要知道，并且有时候需要设定应用程序名。tk appname 命令会返回应用程序的当前名称，或者将它设为一个新值。关于 send 命令的更多信息可参见第 12 章。

29.6 Tk 控制的变量

一些全局变量对 Tk 而言很重要，要么是因为 Tk 会设置这些值，要么是因为它会读取这些变量，并相应地调整它的行为。您可能发现下述变量是有用的。

- tk_library
  由 Tk 设置，存储包含标准 Tk 脚本和演示的数据库的目录的路径名。如果 TK_LIBRARY 环境变量存在，则该变量由环境变量设置；如果不存在，则由其他标准位置设置。详情可参见 tkvars 参考文档。
- tk_version
  由 Tk 设置，为它当前的版本号。它具有诸如 8.5 的格式，其中 8 是主版本号，而 5 是次版本号。主版本号的改变意味着 Tk 中出现不兼容的变化。
- tk_patchLevel
  由 Tk 设置，为它当前的补丁层级。它具有诸如 8.5.4 的格式，其中 8 是主版本号，5 是次版本号，而 4 是特定的补丁层级。

除了这些对应用程序可能有用的变量外，Tk 还使用了关联数组 tk::Priv 来存储它的私有信息。应用程序不应该只是常规地使用或改变 tk::Priv 中的任何值。

29.7 响铃

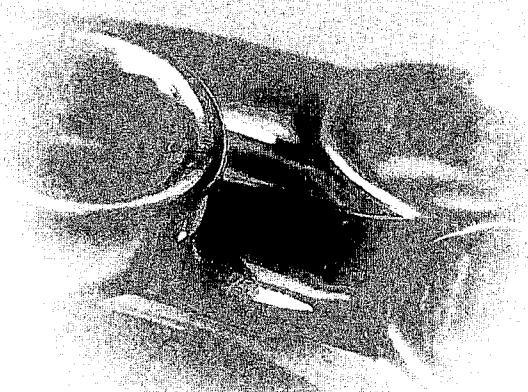
使用 bell 命令来响铃或在给定的显示框架上创造一种可视化效果。基本格式如下：

```
bell
```

您也可以用-displayof 选项定义哪一个显示框架中应该响铃。

```
bell -displayof window
```

使用-nice 选项，bell 命令会试图重置给定显示框架的屏保，这样通常会使屏幕重新可见。



第III部分 C 语言中 Tcl 应用程序的编写

第 30 章 Tcl 与 C 语言的集成原理

虽然单用 Tcl 就可以编写许多种程序，但是在一些情况下需要把 Tcl 和 C 语言结合起来，而且这种结合常见于大型应用程序。

有时候编写 Tcl 比 C 代码要好一些，Tcl 程序能比 C 更容易编写和快速修改。需要的专业知识相对少些，而且更容易被其他的程序员所理解。脚本不需要在每次改动后重新编译，而且对脚本的调试也不是很困难。然而，有时候 C 是最好的选择。

有幸的是，Tcl 能够非常容易且有效地与 C 结合，这是设计初期就确立的观念。

混合 Tcl 与 C 的最常见的理由如下。

- 向 Tcl 中添加 Tcl 中没有的功能，例如，创建一个面向 C 的图形操作库的 Tcl 接口，或者编写一段可以连接到 Linux 系统上的某个设备的代码，该设备可以由一些特定的参数启用 ioctl 系统调用来访问。
- 优化要求快速运行关注性能的任务。在高性能操作方面，C 比 Tcl 快很多倍，因此速度优先时 C 是较好的选择。需要扩展的数值计算或者二进制信息处理的任务经常得益于 C 代码的植入。

本书的这一部分介绍如何使这两种语言各司其位，通过合作的方式达到最好的效果。您将发现这是一个容易而且强大的编程方式，它能带来一些新的应用。

程序员通常通过编写 C 格式的 Tcl 命令将 C 代码植入 Tcl 应用程序中。接下来，这些命令被置入扩展程序，然后扩展程序根据需求被载入应用程序中。这些新的命令就像内建的 Tcl 命令或已建立的过程一样，便于 Tcl 控制。如果想通过编写 C 程序来提高应用程序的性能，一个聪明的优化方式是开始时程序全部使用 Tcl，分析程序运行缓慢的地方，然后用 C 重新编写这些内容。这是一种非常具有吸引力的开发应用程序的方式，因为它允许程序员使用高级语言，同时在必要的地方集中使用较低级的 C 语言的优点，从而减少了程序员花费的时间，并能提高程序运行效率。

第二种方式是将 Tcl 嵌入一个现有的应用程序当中，这是一种用脚本语言加强现行应用程序的强大技术。当处理合适的时候，可以使程序变成一个动态的、可配置的系统，运行时其行为可以被 Tcl 脚本修改，而不用修改一大堆的代码再重新编译，也不需重新启动来读入配置文档。例如，用相对不多的 C 代码，Rivet Apacha 网页服务器模块可以很容易地创建在速度上可与 PHP 竞争的动态网页。图 30.1 展示了整合 C 代码与 Tcl 的两种方法。

大多数 Tcl/C 集成项目都包括几个部分。代码中的大部分用来实现特定用途的 Tcl 命令。然后是启动代码，它的作用是在 Tcl 解释器里注册那些命令并管理其他的启动项，如

创建全局变量。如果在编写嵌入 Tcl 的应用程序，有时需要编写处理 Tcl 脚本的代码段。另一方面，如果编写的是一个扩展程序，您的 C 代码很可能不需要分析处理脚本，通常让扩展包把它的命令注册到 Tcl 解释器中就已经足够了。

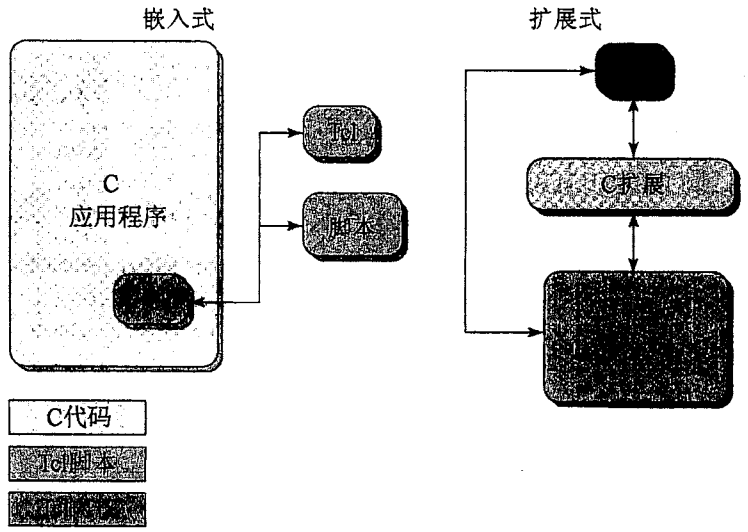


图 30.1　嵌入式 Tcl 与扩展式 Tcl 的比较

30.1　Tcl 与 C：如何选用

为了增强编写的应用程序的适应性，最好将 C 代码组合为一组简单操作的集合——基本构造模块，它足够灵活，可以用多种方式使用，而不是让所有可能的选项以单独的 C 代码方式出现。这样您就可以在 Tcl 脚本中混合与匹配这些简单的操作。最需要决定的是何时采用一个大的功能全面的程序，何时采用多个小的细致的命令。如果在一条命令中包括太多的功能，就很难使用原作者从未考虑过的新且有趣的方法来重组这些小的功能。另一方面，创建的命令过于简单化，对程序开发者而言也不会提供真正的好处和更高的灵活度。正确地从 C 中分离 Tcl 的时机，是让您拥有最大的自由度，同时 Tcl 不至于成为 C API 的复制品。

Tcl 的 socket 命令是复杂底层系统接口的好例子。它并没有复制一系列的用于打开一个套接字的程序调用。取而代之的是，只需提供一个地址(名字或 IP 号码)和一个端口，它就会为您返回一个可以用来读写的通道。更高级的配置可以通过 fconfigure 命令来完成，但基本的操作非常简单。

再举一个例子，考虑一下监视工厂中一条流水线的传感器阵列，每个传感器测量过程中不同点的环境。界面的用户可能希望执行下列任务。

- 打印一份完整的生产线当前状态报告到磁盘上的一个文件中。
- 保存能耗最小的监测站到数据库中。
- 显示监测站 X 的温度。

- 返回空闲时间超过25%的监测站。
- 向网页服务器传送平均温度、最高温度和最低温度。

您需要编写一些C代码才能访问这些信息，这些信息通常只能作为操作系统提供的底层API使用。

一种解决途径是编写从所有传感器读取数据的代码，然后将这些数据送到标准输出。这不失为一种快速的方案，但从长远来看，它的适应性不强，速度也可能会慢，这依赖于从传感器获取数据的时间。

另外一种可能的策略是提供Tcl命令并选择一个有数据需要传输的传感器，再打开读取数据，这样就获得了一个测量结果。这种方法当然是灵活的，但使用API的结果会导致编写代码的速度减慢。

在两者之间保持平衡的方法可能是提供一条sensor命令，它获取如下参数。

- list——返回所有传感器的列表。
- ready——返回所有当前在线的传感器的列表，并传输数据。
- read $sensor——读取并返回指定传感器的信息。

采用标准Tcl命令如foreach来对传感器列表进行循环操作，从每一个传感器中读取数据的方法可能会比较简单。尽管sensor命令不会直接为前面列出的任何一个任务返回结果，但有可能把它和一些脚本联合应用，快速开发所有这5个问题的解决方案，而且这并不需要是一个太低层的解决方案。

打印完整报告到一个文件中将会是一个对所有监测站循环操作的过程，分别通过open和puts命令读取数据，打印到文件中。最低能耗可能由对在线传感器循环，保存出现的能耗值中相对较小者来确定，这时结果会保存在一个相关的数据库中。显示一个选定的传感器数据和读取该数据以及保存需要的数据、废弃其他的数据一样简单。收集闲置的监测站的统计信息同样也包括读取所有的监测站，接着从搜集到的数据中寻找那些特定的值。

30.2 资源名称——把C结构连接到Tcl

C扩展程序常见的应用之一，是允许Tcl代码处理来自您自己的应用程序或其他的C库中的复杂的数据结构或对象。在C程序中，一般通过一个指针来指向该数据结构或对象。

由于Tcl里的所有变量都可以被看作字符串(所以我们不能使用指针)，我们需要想办法连接这些抽象字符串与它们代表的C中的数据——很有可能是C中的结构体。在前文的示例中，可能为传感器提供一些关于它们位置和功能的名称，例如1-heat、2-pressure和3-oxygen。这些名称简洁而且描述性强，人们只需要看到名称，就能够对对象是什么有个大体的印象。在实用的这种系统中，Tcl使用诸如file1和sock7之类的名称分别用来进行文件通道处理和套接字通道处理。这些简单的名称告诉用户某个感兴趣的东西是一个套接字或文件。 在这种情况下我们也使用POSIX文件描述符编号来创建一个独一无二的名称，尽管那个信息并不是为脚本层级的使用准备的，也不应该在脚本层级使用。不过这些字符串可以映射到特定的结构，最有可能是通过一个哈希表，对于每一个输入的字符串都能返回一个相应类型的C结构。Tcl实现了一个强大的哈希表，使得上述方法能够轻松实

现。第 40 章将会更加详细地介绍 Tcl 的哈希表功能。

30.3 “面向动作”与“面向对象”

应用程序中有两种定义命令的方法：“面向动作”与“面向对象”。在“面向动作”的方法中，命令定义为对对象的动作，而该对象作为参数传递给命令。例如，Tcl 用于文件操作的命令都会把一个由 open 返回的文件标识符当成一个参数，对它执行一些动作。

```
set fid [open "/tmp/foobar" w]
fconfigure $fid -translation lf
puts $fid "Some text"
close $fid
```

相应地，“面向对象”的方法只提供一个直接代表对象的命令。这种情况下，有很多 Tcl 命令可以调用：为每个对象调用的，用来创建新对象的。回顾一下，把这种方法用于文件可能更合逻辑：用 open 创建一个文件，然后把它作为一个命令使用。

```
set fid [file "/tmp/foobar" w]
$fid fconfigure -translation lf
$fid puts "Some text"
$fid close
```

不管何种方法，要记得以这种方式产生的命令只是普通 Tcl 命令。然而在许多方面它们表现得与 Smalltalk、Ruby 或者 Java 语言中的对象一样(就是说，次级命令好像“方法”一样；大部分都有一些和自己相关联的数据，等等)。实际上，它们是一些过程化的命令；它们并不能继承，也不支持其他真正的面向对象程序的功能。还有些其他的方法(实际有很多种)来在 Tcl 中建立基于类、方法、继承等的对象。

适合使用“面向动作”的方法的情况如下。

- 资源的数量很大(例如，字符串或者大量整数)，如果为每一个对象创建一条命令将会很浪费。
- 资源是短期的，所以对象的设置及销毁将会很花费时间，很不值得。
- 一条命令可以接受多种类型的对象作为参数，并且对它们运行相同的代码。例如，destroy 在 Tk 中可以在多种类型的组件上运行，所以最好将此类功能存放在一个地方，而不是为每一个组件建立一个 destroy 子命令。

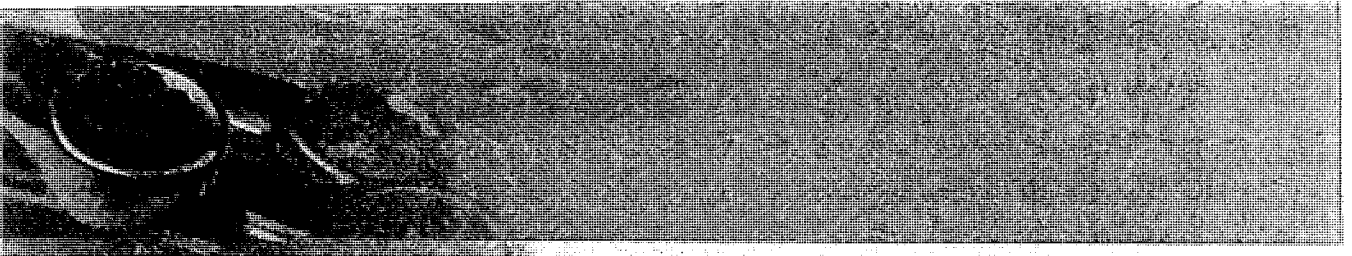
我们的传感器监控系统的例子就使用这种方式，因为有许多传感器需要监测，而且不需要在每一个传感器上运行太多操作。当并不是所有的传感器都不断地读数据时，使用“面向动作”的方式更有效，这样系统没必要一直为每一个传感器都保留信息，只在被请求时才保留。

适合使用“面向对象”的方法的情况如下。

- 没有太多的对象(几十个或几百个)。
- 对象已被很好地定义。
- 它们很有可能持续存在而且会被使用一段时间。
- 它们有一定的复杂度，可能对它们进行不少的操作。

30.4 描述性信息

传入传出 Tcl 命令的信息应该被格式化成 Tcl 脚本容易处理的格式，而不必让它们更易于被人工阅读。传感器监控命令不应该返回类似于 hot、real hot 或者 The temperature is 70 C 这样的关于传感器状态的描述，或者是格式美观的待打印的表格，只应该返回一些容易传输和被其他 Tcl 命令使用的结果。如果需要以某种方式组织数据，您可以用 C 建立 Tcl 列表和数组(或 Tcl 8.5 中的字典)。为了帮助其他的程序员使用您的数据，您应该使自己的数据具有良好的自我描述性，不要让人们疑惑列表中的哪个位置对应哪个数据。举例而言，列表 105 89 96 自身没有任何意义。如果我们使用如 max 105 min 89 average 96 的字典，那么这样的数据一眼就可以知道它的含义，同时这样的数据结构也更易于 Tcl 操作。



第31章 解 释 器

本章讲解什么是解释器，如何创建、删除它们以及怎样使用它们来处理Tcl脚本。

31.1 本章出现的函数

- `Tcl_Interp *Tcl_CreateInterp()`
  创建并返回一个新的Tcl解释器。
- `Tcl_DeleteInterp(Tcl_Interp *interp)`
  删除Tcl 解释器。
- `Tcl_InterpDeleted(Tcl_Interp *interp)`
  如果解释器未被删除，则返回非零值。
- `Tcl_Interp *Tcl_CreateSlave(Tcl_Interp *interp, CONST charslaveName, int isSafe)`
  创建一个名为 *slaveName* 的从解释器。*isSafe* 参数决定是否创建一个安全解释器。
- `int Tcl_IsSafe(Tcl_Interp *interp)`
  如果是安全解释器，则返回1，否则返回0。
- `int Tcl_MakeSafe (Tcl_Interp *interp)`
  将解释器转换成为一个安全解释器，移除所有“危险”的命令和变量。但并不移除扩展包中的命令，所以只调用这个函数并不能保证安全。
- `Tcl_Interp *Tcl_GetSlave(Tcl_Interp *interp, CONST char *slaveName)`
  返回 *interp* 的名为 *slaveName* 的从解释器。
- `Tcl_Interp *Tcl_GetMaster(Tcl_Interp *interp)`
  返回给定的从解释器的主解释器。
- `int Tcl_GetInterpPath(Tcl_Interp *askingInterp, Tcl_Interp *slaveInterp)`
  将 *askingInterp* 的结果设为 *askingInterp* 和 *slaveInterp* 之间的路径。返回TCL_OK，如果路径无法计算则返回TCL_ERROR。
- `int Tcl_HideCommand(Tcl_Interp *interp, CONST char *cmdName, CONST char *hiddenCmdName)`
  将 *cmdName* 设为隐藏命令，且命名为 *hiddenCmdName*。如果 *cmdName* 并不作为

可见命令存在，则返回 TCL_ERROR。

- int Tcl_ExposeCommand(Tcl_Interp *interp,
  CONST char *hiddenCmdName, CONST char *cmdName)

  将由 *hiddenCmdName* 所隐藏的命令设为可见命令 *cmdName*。如果命令不存在，则返回 TCL_ERROR。

- int Tcl_CreateAlias(Tcl_Interp *slaveInterp,
  CONST char *slaveCmd, Tcl_Interp *targetInterp,
  CONST char *targetCmd, int argc,
  CONST char **argv)
  int Tcl_CreateAliasObj(Tcl_Interp *slaveInterp,
  CONST char *slaveCmd, Tcl_Interp *targetInterp,
  CONST char *targetCmd, int objc,
  Tcl_Obj **objv)

  这两个命令本质上是相同的，差别在于为了效率，Tcl_CreateAliasObj 接受的参数是 Tcl 对象的数组，而不是字符串。两条命令都在 *slaveInterp* 中创建了一条 *slaveCmd* 的命令，设为在 *targetInterp* 中的 *targetCmd* 的别名。*argv* 和 *objv* 中的参数会作为传入这条命令的全部参数的前缀。

- int Tcl_GetAlias(Tcl_Interp *interp,
  CONST char *slaveCmd, Tcl_Interp *targetInterpPtr,
  CONST char *targetCmdPtr, int *argcPtr,
  CONST char ***argvPtr)
  int Tcl_GetAliasObj(Tcl_Interp *interp,
  CONST char *slaveCmd, Tcl_Interp *targetInterpPtr,
  CONST char *targetCmdPtr, int *objcPtr,
  Tcl_Obj ***objvPtr)

  以上两个函数都获得关于别名命令的信息，在 *interp* 中查找别名 *slaveCmd*，设置 *targetInterpPtr*、*targetCmdPtr* 和指定参数数目的指针，该指针在两个函数中分别为 *objcPtr* 和 *argvPtr*。

31.2 解释器概述

Tcl 库使用的中心数据结构是 Tcl_Interp 类型的 C 结构体。在本书这一部分，我们会把这些数据结构称作解释器。解释器包括了 Tcl 脚本的运行状态，包括 Tcl 过程、由 C 语言实现的命令、变量以及反映命令和脚本处理的内部状况的运行堆栈。某种意义上，这就是 Tcl 脚本运行的世界，而且 Tcl 脚本不能看到它们所处解释器之外的世界。在处理脚本时，解释器跟踪脚本处理到哪里，可被调用的命令以及已被赋值的变量的信息。大部分的

Tcl 库中的过程都把指向 Tcl_Interp 结构的指针用作一个参数。

Tcl 应用程序通常只用一个解释器，然而，一个单独的进程可以管理多个独立的解释器。例如，一个需要响应多个客户端的请求，在获得网络连接时运行一些 Tcl 代码的服务器。每个解释器需要负责执行一个特定用户的代码，所以需要使用不同的解释器，它们之间的数据不是共享的，尽管占有相同的地址空间。需要牢记的是多解释器并不代表多线程，所以不要并发地解决问题。想了解并发性问题可以参考第 43 章的事件和第 46 章的线程。

如果将 Tcl 嵌入您自己的代码中，应该为它创建一个或多个解释器。如果您在编写一个 Tcl 扩展程序，当其被加载时，Tcl 会把扩展程序的命令和变量加入加载该程序的解释器中。

31.3 简单的 Tcl 应用程序

迄今为止，您已经用 tclsh 或 wish 二进制程序运行过 Tcl 程序，但是当然这不是唯一的途径。

下面的程序演示了如何创建和使用一个解释器。它是一个简单的但是完整的 Tcl 应用程序，功能是处理一个文件中的 Tcl 脚本并打印结果，如果出现错误，则返回错误信息。

```
#include <tcl.h>
int main (int argc, char *argv[]) {
    Tcl_Interp *interp;
    char *result
    int code;

    if (argc != 2) {
        fprintf(stderr,
            "Wrong number of arguments: ");
        fprintf(stderr,
            "should be \"%s filename\"\n", argv[0]);
        exit(1);
    }

    interp = Tcl_CreateInterp();
    code = Tcl_EvalFile(interp, argv[1]);
    result = Tcl_GetStringResult(interp);
    if (code != TCL_OK) {
        printf("Error was: %s\n", result);
        exit(1);
    }
    printf("Result was: %s\n", result);
    Tcl_DeleteInterp(interp);
    exit(0);
}
```

如果您安装的是 Tcl 完整版，那么编译这段代码会显得相对容易些，只需要在 Debian Linux 系统下使用以下命令：

```
cc -o simple simple.c -ltcl8.5
```

依赖于您安装的 Tcl，例如，在 Debian Linux 操作系统上，或许需要添加-I/usr/include/

tcl18.5 这样的标志。创建 Tcl 应用程序和扩展程序的更多信息参见第 47 章。

现在就可以用程序运行 Tcl 脚本了，例如：

```
./simple hello.tcl
```

回到 simple.c。对于这个简单的例子，我们不需要 tcl.h 以外的更多任何头文件，该头文件本身已经逐次包括了很多其他的头文件。理论上，不管何时使用 Tcl，都需要包含头文件 tcl.h。在 main 函数中，我们创建了一个指向解释器的指针、一个整数返回值和一个指向结果字符串的指针。

然后我们需要确认程序传入了一个参数——一个包含需要处理的 Tcl 脚本的文件。接着，我们创建了解释器。这个新的解释器包含了 Tcl 库(这里是 libtcl8.5.so)中所有的内建命令，但是只有标准 tclsh 中的小部分过程和变量，因为我们没有加载标准初始化脚本。有关什么变量出现在哪里的信息，参见 tclvars 和 tclsh 参考文档。文件处理完成后，就有两条可用的信息：第一，Tcl_EvalFile 的返回码，如果一切正常，对应于 TCL_OK，这就是一个整型数，如果出现问题则是 TCL_ERROR，或者对应其他各种情况分别为 TCL_RETURN、TCL_BREAK 或 TCL_CONTINUE。要获取字符串格式的结果，或是一个错误字符串，我们可以调用 Tcl_GetStringResult，然后就会根据实际情况的结果或错误或其他情况，打印相应的信息。

从您自己的应用程序和扩展程序中处理 Tcl 代码的更多方法参见第 33 章。

31.4 删除解释器

如果不再需要一个 Tcl 解释器，可以调用 Tcl_DeleteInterp 函数来删除它。它将释放解释器以及与它相关的全部变量、命令和未与其他解释器共享的文件描述符。

31.5 多重解释器

Tcl 非常有趣的功能之一就是允许存在多个活动的解释器。如何用 Tcl 脚本管理多个解释器的讨论参见第 15 章。就像前面讲的那样，多重解释器并不是并发机制，然而对于隔离必须独立运行的代码，它们是非常有用的。建立子解释器(也称为从解释器)的基本机制是由 Tcl_CreateSlave 函数提供的，示例如下：

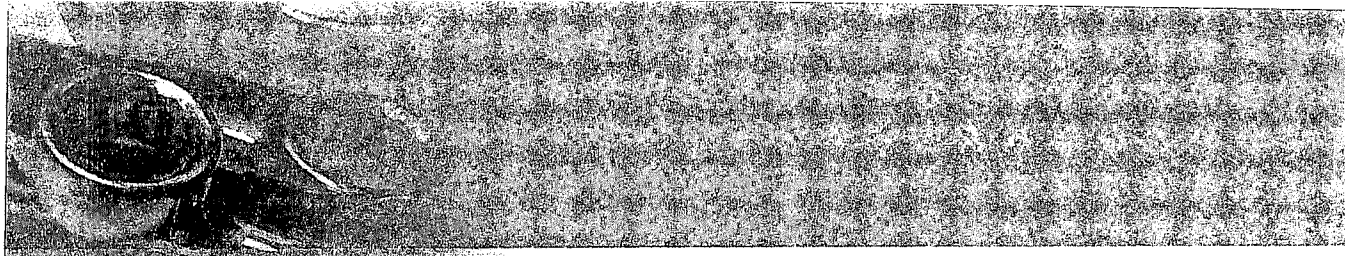
```
char *interp_names[] = {"a" , "b" , "c" , "d" , "e" , "f"};
/* Port numbers to use for interpreters. */
int interp_ports[] = {10000, 10001, 10002,
                      10003, 10004, 10005};

...

for (i = 0; i < NUM_INTERPS; i++) {
    /* Create a slave interpreter. */
    slave_interps[i] = Tcl_CreateSlave(
        interp, interp_names[i], 0);
   /* Set the 'port' variable in the slave
```

```
            interpreter. */
        Tcl_SetVar2Ex(slave_interps[i], "Port", NULL,
            Tcl_NewIntObj(interp_ports[i], 0);
        /* Run the script in the new slave interpreter. */
        Tcl_EvalFile(slave_interps[i], argv[1]);
    }
```

上面这个程序中的循环操作会创建一系列的从解释器，每一个都指定了一个监听端口，所以 Tcl 代码可以对每一个端口进行监听。如果完全相信某个连接者，甚至可以处理由套接字传来的代码并返回结果，并且一个解释器中出现的问题不会影响其他的解释器。Tcl_CreateSlave 的入口参数包括一个 Tcl 解释器、一个名称和一个表征是否要将从解释器设置为安全解释器的整型值。



第 32 章　Tcl 对象

在 Tcl 中，所有的值都当作字符串来处理。早期的 Tcl 版本将所有值，甚至包括数值类型的数据都内部存储为字符串类型。这种方法的速度很慢，尤其在需要经常将数据转化为二进制格式(例如浮点数)的情况下。

相反，现代版本的 Tcl(从 1997 年发布的 8.0 起)采用一种名为 Tcl 对象的高效数据存储方式。Tcl 对象有一个字符串表示，还有一个其他类型的“内部”表示，例如整数、双精度、列表，以及 Tcl 8.5 开始才具备的字典类型。变量值、命令参数、脚本结果和脚本在 Tcl 内部都采用这种 Tcl 对象形式存储。

Tcl 对象的这种与生俱来的双重性质使得它们可以满足速度方面的要求，可以高效地使用内存，而且容易操作。例如 while 循环中的计数变量，会被转化为整数形式并在循环过程中仍然保持为整数形式。因为不需要多次重复地进行数据格式转换，从而节省时间、提高性能。

由于对象是 Tcl 新添加的，所以对于很多操作有两种 API 函数：一种使用对象格式，一种使用基于字符串的接口。早期的 API 简单些，但是这是以牺牲速度为代价的，尤其是任务数量很多的时候。

在 C 的层面上，Tcl 对象由 Tcl_Obj 结构表达。

```
Tcl_Obj *obj = Tcl_NewObj();
```

这条代码产生一个新的空白对象。大多数情况下，它是一个不透明的类型——您不需要去处理它的结构成员，这些由 Tcl 的 API 完成。

32.1　本章出现的函数

- `Tcl_Obj *Tcl_NewObj()`
  创建一个新对象。
- `Tcl_Obj *Tcl_DuplicateObj(Tcl_Obj *objPtr)`
  返回对象的一个新副本，引用计数为 0。
- `Tcl_IncrRefCount(Tcl_Obj *objPtr)`
  增加对象的引用计数。
- `Tcl_DecrRefCount(Tcl_Obj *objPtr)`
  减少对象的引用计数，如果引用计数降为 0 或 0 以下，则将其释放。

- int Tcl_IsShared(Tcl_Obj *objPtr)
  对象被共享则返回 1，否则返回 0。
- int_InvalidateStringRep(Tcl_Obj *objPtr)
  标记对象的字符串表达形式为无效并释放其相关的空间。
- Tcl_Obj *Tcl_NewBooleanObj(int boolValue)
  用初始值 *boolValue* 创建一个新的对象。
- Tcl_Obj *Tcl_NewIntObj(int intValue)
  用初始值 *intValue* 创建一个新的对象。
- Tcl_Obj *Tcl_NewLongObj(long longValue)
  用初始值 *longValue* 创建一个新的对象。
- Tcl_Obj *Tcl_NewWideIntObj(Tcl_WideInt wideValue)
  用初始值 *wideValue* 创建一个新的对象。
- Tcl_Obj *Tcl_NewBignumObj (mp_int *bigValue)
  用初始任意精度的整数 *bigValue* 创建一个新的对象。
- Tcl_Obj *Tcl_NewDoubleObj(double doubleValue)
  用初始值 *doubleValue* 创建一个新的对象。
- Tcl_Obj *Tcl_NewStringObj(const char *bytes, int length)
  以字符型指针 *bytes* 指向的 UTF-8 型字符串作为初值创建一个新的对象。只复制前 *length* 个字节，当 *length* 为负值时，复制第一个空字符之前的所有字节。
- Tcl_Obj *Tcl_NewByteArrayObj(CONST unsigned char *bytes, int length)
  从 *bytes* 创建一个新的字节数组对象。
- Tcl_SetBooleanObj(Tcl_Obj *objPtr, int boolValue)
  将已存在的对象 *objPtr* 赋值为布尔值 *boolValue*。
- Tcl_SetIntObj(Tcl_Obj *objPtr, int intValue)
  将已存在的对象 *objPtr* 赋值为 *intValue*。
- Tcl_SetLongObj(Tcl_Obj *objPtr, long longValue)
  将已存在的对象 *objPtr* 赋值为 *longValue*。
- Tcl_SetWideIntObj(Tcl_Obj *objPtr, Tcl_WideInt wideValue)
  将已存在的对象 *objPtr* 赋值为 *wideValue*。
- Tcl_SetBignumObj(Tcl_Obj *objPtr, mp_int *bigValue)
  将已存在的对象 *objPtr* 赋值为任意精度整数 *bigValue*。
- Tcl_SetDoubleObj(Tcl_Obj *objPtr, double doubleValue)
  将已存在的对象 *objPtr* 赋值为 *doubleValue*。
- Tcl_SetStringObj(Tcl_Obj *objPtr, const char *bytes, int length)
  将已存在的对象 *objPtr* 赋值为 *bytes* 指向的 UTF-8 型字符串。只复制前 *length* 个字节，当 *length* 为负值时，复制第一个空字符之前的所有字节。

- `void Tcl_SetByteArrayObj(Tcl_Obj *objPtr, CONST unsigned char *bytes, int length)`

  将已存在的对象 *objPtr* 赋值为包含 *bytes* 内容的字节数组。

- `unsigned char *Tcl_SetByteArrayLength(Tcl_Obj *objPtr, int length)`

  设置字节数组的长度，截取或延伸任意长度，然后返回对象的新字节数组。

- `int Tcl_GetBooleanFromObj(Tcl_Interp *interp, Tcl_Obj *objPtr, int *boolPtr)`

  获取 *objPtr* 的布尔值且将其赋予 *boolPtr*。失败时返回 TCL_ERROR。

- `int Tcl_GetIntFromObj(Tcl_Interp *interp, Tcl_Obj *objPtr, int * intPtr)`

  获取 *objPtr* 的整数变量值且将其赋予 *intPtr*。失败时返回 TCL_ERROR。

- `int Tcl_GetLongFromObj(Tcl_Interp *interp, Tcl_Obj *objPtr, long *longPtr)`

  获取 *objPtr* 的长整型变量值且将其赋予 *longPtr*。失败时返回 TCL_ERROR。

- `int Tcl_GetWideIntFromObj(Tcl_Interp *interp, Tcl_Obj *objPtr, Tcl_ WideInt *widePtr)`

  取得 *objPtr* 的宽整型值且将其赋予 *widePtr*。失败时返回 TCL_ERROR。

- `int Tcl_GetBignumFromObj(Tcl_interp *interp, Tcl_Obj *objPtr, mp_int *bigValue)`

  取得 *objPtr* 的任意精度的整型值且将其赋予 *bigValue*。失败时返回 TCL_ERROR。

- `int Tcl_GetDoubleFromObj(Tcl_Interp *interp, Tcl_Obj *objPtr, double *doublePtr)`

  取得 *objPtr* 的双精度整型值且将其赋予 *doublePtr*。失败时返回 TCL_ERROR。

- `char *Tcl_GetStringFromObj(Tcl_Obj *objPtr, int *lengthPtr)`

  返回指向对象的字符串表达形式的指针，如果字符串非空，则将其长度赋予 *lengthPtr*。

- `unsigned char *Tcl_GetByteArrayFromObj(Tcl_Obj *objPtr, int *lengthPtr)`

  返回字节数组对象中包含的字节，如果它非空，则将其长度赋予 *lengthPtr*。

- `char *Tcl_Alloc(int size)`
  `char *Tcl_AttemptAlloc(int size)`
  `char *ckalloc(int size)`
  `char *attemptckalloc(int size)`

  返回指向至少 *size* 个字节的区块的指针，可以用于各种用途。优先使用这些子程序而不是原生的 malloc。Tcl_AttemptAlloc 在内存分配失败时不引起 Tcl 解释器的崩溃，而 Tcl_Alloc 会因失败导致崩溃。ckalloc 和 attemptckalloc 与它们各自对应

的函数功能相似，只不过它们还支持高级的内存调试。更多信息详见参考文档。

- `char *Tcl_Realloc(char *ptr, int size)`
  `char *Tcl_AttemptRealloc(char *ptr, int size)`
  `char *ckrealloc(char *ptr, int size)`
  `char *attemptckrealloc(char *ptr, int size)`
  将 *ptr* 指向的区块的大小改为 *size* 字节，并且返回指向新块的指针。块中的内容从开头一直到与新旧块中较小的那个相等大小的空间位置为止，不发生变动。返回的地址可能有别于 *ptr*。Tcl_AttemptRealloc 在内存分配出错的情况下不会引起 Tcl 解释器崩溃，而 Tcl_Realloc 会。ckrealloc 和 attemptckrealloc 与它们各自对应的函数功能相似，只不过它们还支持高级的内存调试。
- `Tcl_Free(char *ptr)`
  `ckfree(char *ptr)`
  让 *ptr* 指向的地址空间重新用于分配。ckfree 是一个功能相似且支持高级内存调试的宏包。

32.2 字符串对象

既然所有的 Tcl 对象都可以表示为字符串，那么我们首先介绍字符串形式的 Tcl 对象。可使用以下命令新建字符串对象：

```
Tcl_Obj *strobj;
strobj = Tcl_NewStringObj("Tcl!", -1);
```

一个新的字符串对象由一系列的字节(UTF-8 编码，参见第 39 章)和它们的长度创建。在这种情况下，值-1 代表由 Tcl 决定字符串的长度，而不是我们自己输入一个确定的值，这通常是一个便利的捷径。

字符串和操纵它们的命令在第 39 章中有更详细的介绍。

32.3 数值对象

创建包含一个整数的对象可采取相似的方式。

```
Tcl_Obj *intobj;
intobj = Tcl_NewIntObj(42);
```

在数学操作中这样的 intobj 比用字符串表示的 42 效率更高，因为它的内部表达形式是一个真正的整数。

以下是可以创建的其他数值类型。

- Boolean——真或假，对应 1 或 0。由 Tcl_NewBooleanObj(*int*)创建。
- long——保存一个长整型的数。由 Tcl_NewLongObj(*long*)创建。
- wide——保存一个宽的整数：64 位。由 Tcl_NewWideIntObj(*wideValue*)创建，其

中 *wideValue* 是一个至少 64 位的值。

- bignum——保存一个任意精度的整数。由 Tcl_NewBignumObj(*bigValue*)创建，*bigValue* 是一个指向 mp_int 结构的指针，该结构在 LibTomMath 的任意精度的整数库里有声明。
- double——包含一个 C 的双精度类型。Tcl 用这种类型来处理浮点数。由 Tcl_NewDoubleObj(*double*)来创建。

32.4 从对象中获取 C 语言数据

要在 C 代码中使用一个 Tcl 的对象，需要获得它们背后的数据格式，如字符串、整数、双精度数或者其他格式。您的程序用 Tcl 库把给定的对象翻译为您需要的类型。这也许可以直接由字符串或内部表达式获得，也许需要做一些转换(例如，从一个有整数内部表达的对象中提取到一个双精度的数据)。虽然 Tcl 可以保证任何的对象都可以被获取为一个字符串，但对其他的数据类型并非如此。还要注意当要求一个特定类型的值时，Tcl 会把该值的内部表达形式修改为要求的类型——当然，是在可以执行这样的操作的情况下。例如，如果一个对象的内部表达为整数，而且 Tcl_GetDoubelFromObj 被顺利执行，那么新的内部表达就成了双精度型。

无论什么时候，调用 Tcl_GetStringFromObj 可以将一个对象转变为字符串，但是对象转换成整数、双精度型等其他类型则并不总是可行的。因此，Tcl_GetIntFromObj 和从 Tcl 对象获取其他类型的调用都要求将一个解释器作为参数，这样，如果它们在把对象转换为指定的类型时遇到问题，就可以抛出异常。

```
char *intstring;
intstring = Tcl_GetStringFromObj(intobj);
/* No problem. */

int strint;
Tcl_Obj *strobj;
strobj = Tcl_NewStringObj("Tcl!", -1);
Tcl_GetIntFromObj(interp, strobj, &strint);
/* Fails! */
```

在第一种情况下，对 Tcl_GetStringFromObj 的调用总是会成功的，Tcl 的体系就决定了这一点：Tcl 对象总是可以表现为字符串——这里 intstring 是 42。然而，对 Tcl_GetIntFromObj 的调用可能返回错误，因为我们试图针对对象 strobj 调用它。这个对象的内容是字符串 Tcl!，它不能转换成一个整型数，因此返回了 TCL_ERROR 作为结果，错误消息留给解释器。和创建对象一样，从 Tcl 对象中取得各种类型的值有相应的函数，例如 Tcl_GetLongFromObj、Tcl_GetDoubleFromObj 以及获得其他数据类型的很多函数。

32.5 Tcl 对象的动态本质

要演示 Tcl 对象的双重本质，让我们通过以下 Tcl 脚本检视它的状态。

```
set foo 35100
puts "The value of \$foo is: $foo"
incr foo 100
puts "Now \$foo is :$foo"
```

在第 1 行，变量 foo 被设置为一个对象，其内容为字符 35100。这时对象是一个字符串。

当第 2 行的脚本输出变量 foo 时，需要它的字符串的值，这里使用了 35100。

第 3 行的 foo 对象的内容就是转换以后的。虽然前面它是一个字符串，但现在我们希望把它的值加上 100，因此我们需要把它作为一个数字来处理。而它现在还不是一个数字，因此将其转化成整数类型，现在它的内部表示是数字 35100。incr 命令对这个整数进行操作，将它的值增加为 35200。而该对象的字符串表达形式则标记为不可用，同时也不生成新的字符串表达形式。如果这个数字接下来还要进行数学操作，现在生成它的字符串表达式就只会浪费工夫。

最后一行，我们再次输出了 foo 的值。现在它的字符串表达形式就是可用的了，更新为 35200，然后 puts 命令将其输出。在脚本的结尾，该对象包含了一个有效的字符串表达形式和一个整数 35200。

32.6 字节数组

字节数组用于保存由任意的 8 位值组成的序列，这些值并不一定对应于字符，而字符串对象的内容则必须对应于字符。字节数组可用于保存二进制数据，例如 JPEG 文件或 MP3 声音文件，这些文件可能包含着内嵌的 NULL 字节。

```
unsigned char jpegheader[11] =
    {0xff, 0xff, 0xd8, 0xff, 0xe0, 0x00,
     0x10, 0x4a, 0x46, 0x49, 0x46);
Tcl_Obj *byteobj;
byteobj = Tcl_NewByteArrayObj(jpegheader, 11);
```

这个示例创建了一个 Tcl 对象，内容是 JPEG 文件头部的一些起始字节。

32.7 复合对象

在 Tcl 8.4 以前，Tcl 只有一种对象类型可以包含其他的对象：列表。从 C 语言的观点看，列表是一种 Tcl 对象，其内容是其他 Tcl 对象构成的数组。用 Tcl 脚本操纵数组的讨论参见第 6 章。Tcl 的 C API 可用于完成用 Tcl 完成的所有操作，更多细节参见第 41 章。

从 Tcl 8.5 版(现在 8.4 版也可以追加)起，Tcl 还提供了字典对象(也称为 dict 对象)，通过一个哈希表来引用其他对象。第 7 章讨论了用于操作字典对象的 Tcl 脚本命令。C 语言接口允许您完成同样的很多操作，详见第 41 章。

Tcl 列表和字典可以在 Tcl 与 C 之间很好地传送复杂的结构，我们在第 30 章介绍信息表现形式时见过这种应用。当元素可以由一个数字索引访问时(例如，取得第 5 个元素)，列表特别适用；而根据描述对象的字符串查找对象时，字典的效率很高，例如，将“Rome”映射到“Italy”，将“Paris”映射到“France”，将“Berlin”映射到“Germany”，等等。

32.8 引用计数

为了避免为一个对象创建不必要的副本，应用程序的各个部分可以在同时共享同一个 Tcl 对象。例如，一个 Tcl 对象可能由一个过程使用，存放在一个变量中，在 Tcl 库中使用，等等。为了管理对象和它们消耗的内存，Tcl 使用了一个名为引用计数的机制。引用计数也解决了 Scheme、Ruby 及 Java 这样的语言中存在的垃圾引用问题。这种机制健壮、高速、明了而且移植性好，它使在 C 层级使用 Tcl 的工作变得简单。

新创建的 Tcl 对象的初始引用计数是 0。每一段需要标记这个对象为使用中的代码，都应该使用 Tcl_IncrRefCount 函数来增加引用计数。当代码不再需要这个对象时，它应该使用 Tcl_DecrRefCount 函数减小引用计数。一旦某个对象的引用计数减到 0 或者更小，Tcl 就可以释放这个对象了。

很多 Tcl 函数会暂时地修改作为参数提供的对象的引用计数。如果代码创建了一个新的对象(引用计数为 0)，然后将它传给一个 Tcl 函数，传入之前不增加它的引用计数，那么 Tcl 函数会内部地增加然后减小它的引用计数。函数内部地将该对象的引用计数减小为 0 时，就会触发对该对象的删除操作。接下来后面的代码如果试图使用这个对象，就会产生错误。

一般说来，新创建一个对象后应该首先增加它的引用计数，然后在指针变量超出范围时减小它的引用计数。不过，也有一些 Tcl 函数为您提供的已经存在的对象保存新的引用，增加对象的引用计数。这些函数将对象放置到变量中，存入列表或字典对象中，或者把它放入解释器的结果中。(这些动作在后面的几章中会一一讨论。)如果创建的对象只需要传入这些函数之一，那么不对它使用 Tcl_IncrRefCount/Tcl_DecrRefCount 也是安全的。

32.9 共享对象

Tcl 为对象实现了“写出时复制”的系统。这意味着一个对象可以由多个，例如说变量来共享，只要没有向它们中的任何一个进行写操作。一旦出现写操作，就会生成该对象的一个副本。重点在于，当您为 Tcl 编写直接修改一个对象的 C 代码时，需要检查确认它没有由 Tcl_IsShared 函数设置共享，如果它已经被共享，则使用 Tcl_DuplicatedObj 来复制它。

```
if (Tcl_IsShared(objPtr)) {
    objPtr = TclDuplicateObj(objPtr);

    /* objPtr now points to a duplicate of the original
     * object, with a reference count of 0.
     */
}
```

需要这个检查的命令通常就是那些访问和修改作为变量值的对象的函数，例如 incr 和 lappend。下面这段 Tcl 代码展示了一个对象，它首先被共享，然后在修改时被复制。

```
set foo 1
set bar $foo;
# foo and bar share the same object - '1'.
incr bar;
# bar now has its own object -'2'.
```

最初，创建了一个包含 1 的对象，赋给 foo 变量。然后，同一个对象被赋给 bar 变量。这时，两个变量都指向同一个对象，该对象的值是 1。下一条命令是 incr bar，注意因为这个值是共享的，首先要生成一个副本，然后再增加副本中的那个值。现在 foo 变量包含的对象的值是 1，而 bar 包含了另一个值为 2 的整型对象。如果我们还要再一次处理 incr bar，Tcl_IsShared 就会返回“假”，因为 bar 中的对象和 foo 中的对象现在已经不是同一个了。

32.10 新的对象类型

因为它们很少被用到，这里我们不进行深入的讨论，但您应该知道如果需要的话，通过 Tcl 的 C API 可以创建新的对象类型。

创建新的对象类型的意义在于对象可以容易地表达为字符串，可以在字符串和所需要的特定类型之间反复转换而不损失信息。对于更复杂的 C 数据结构，更好的方法是通过哈希表将结构映射到独一无二的标记，30.4 节提到过这种技术。

我们必须确认在用户把对象的值作为字符串修改后，将它转换回需要的类型时不会发生不好的情况。考虑一下，假设我们创建了一个 Tcl“引用”对象类型，它包含了一个 C 的指针，如 0x10001800。如果用户把它作为一个字符串修改为)0x10001810，然后又试图把它作为指针类型使用，最可能的结果就是数据冲突或段错误。

创建新的 Tcl 对象类型的最典型的目的是，使得信息可以新的数据类型很简单地表达(而不是使用 C 中的复杂结构)，同时让信息能够毫无问题地由字符串表示。

如何创建新的 Tcl 对象类型的一个很好的示例就是 Salvatore Sanfilippo 的 tclsbignum 包，在 Tcl 8.5 支持原生 BignumObj 之前，这个包就提供了任意精度的整型数转换能力。

新类型的核心就是 Tcl_ObjType 结构体。

```
typedef struct Tcl_ObjType {
    char *name;
    Tcl_FreeInternalRepProc *freeIntRepProc;
    Tcl_DupInternalRepProc *dupIntRepProc;
    Tcl_UpdateStringProc *updateStringProc;
    Tcl_SetFromAnyProc *setFromAnyProc;
} Tcl_ObjType;
```

创建一个新类型时，应该提供如下函数。

- 由 Tcl_FreeInternalPepProc 释放类型的存储空间。在 bignum 库中，这就意味着释放相关联的内存。
- 由 Tcl_DupInternalRepProc 生成类型的副本。如果类型是一个数据结构，那么程序员要决定是否进行“深层复制”。深层复制的意思是复制整个数据结构，而不仅是它的顶部或头部。例如，在复制列表时，可以复制这个列表结构本身，让其

中的元素继续指向相同的对象，或者也可以把这些元素都复制一遍(如果元素本身也是列表，继续复制其下一层的元素)。这就是一个深层复制。事实上，Tcl 的列表复制不是深层复制。在 bignum 库中，bignum 的值会被复制。

- 由 Tcl_UpdateStringProc 更新对象的字符串表现形式，让它与对象的当前值相匹配。bignum 库中的字符串更新函数将字符串设置为数字的十进制表达形式。
- 上面操作的逆操作——根据字符串重置内部表达形式——由 Tcl_ObjType 结构体中的 SetFromAnyProc 函数执行。这种情况下，bignum 库必须把字符串解析为一个数字，如果无法解析，就会返回错误。

32.11 解析字符串

在 Tcl 对象出现之前，所有内容都内部表达为字符串，就需要有能力把那些字符串解析为所需要的其他类型。Tcl 仍然提供了那些调用，在处理字符串时使用它们可能会比较方便。

```
int Tcl_GetInt(Tcl_Interp *interp, CONST char *string,
    int *intPtr);
int Tcl_GetDouble(Tcl_Interp *interp,
    CONST char *string, int *doublePtr);
int Tcl_GetBool(Tcl_Interp *interp, CONST char *string,
    int *intPtr):
```

这些调用都需要传入一个解释器和一个含有可解析为参数的信息的字符串，指向目标类型的 C 变量的指针，以及一个为 1 或者 0 的布尔型值。如果成功，它们就返回 TCL_OK，失败时返回 TCL_ERROR。例如：

```
int num = 0;
Tcl_GetInt(interp, "42", &num);
```

这些代码把 num 设为 42，并且返回 TCL_OK。

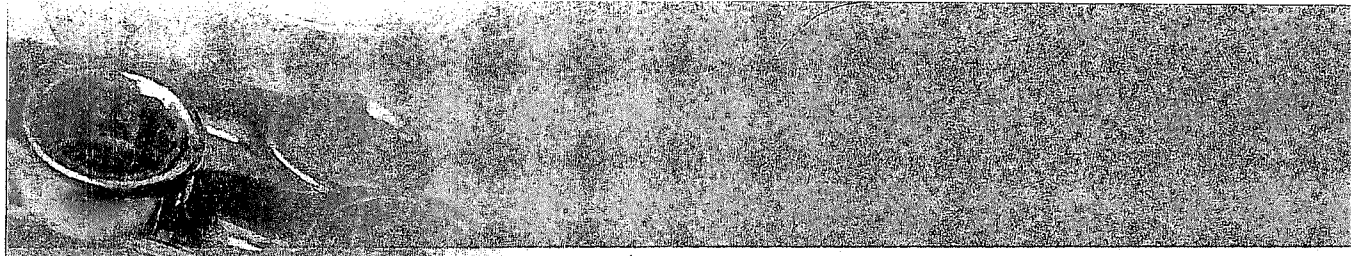
32.12 内存分配

在后面的章节中，您会看到 Tcl 提供的很多函数，以跨平台一致的形式执行常用动作。Tcl 提供的最重要的动作之一就是内存分配。我们使用 Tcl_Alloc 和 Tcl_Free 在 Tcl 中分配和释放内存。这些函数和 C 中的 malloc 和 free 相同，但是注意不要混合匹配它们，例如不要调用 Tcl_Free 来释放由 malloc 或 strdup 分配的内存，在有某些特定设置的平台上，这样的操作会导致问题。另外，Tcl 提供了 Tcl_Realloc 函数，可以代替 realloc，改变已分配的内存区块的大小。返回的位置可能和原始的位置不同，但原区块中新旧两个区块中较小的区块内的那部分的内容是不会被改变的。

Tcl_Alloc 和 Tcl_Realloc 在内存分配失败时会让 Tcl 退出。要避免这种情况，可以调用 Tcl_AttemptAlloc 和 Tcl_AttemptRealloc，它们在失败时只是返回 NULL。

与这些函数对应的是一系列的宏：ckalloc、ckfree、ckrealloc、attemptckalloc 以及

attemptckrealloc。一般来说，它们是前面描述的对应过程的同义词。然而，如果 Tcl 和所有调用 Tcl 的模块在 TCL_MEM_DEBUG 限定时进行编译，这些宏会被重新定义为这些过程的特殊的调试版本。这样会启动 memory 集合，对内存分配进行跟踪和报告，更多信息参见 memory 的参考文档。如果希望对代码进行内存调试，应该使用这些宏，而不是直接调用 Tcl_Alloc 和其他函数。



第 33 章　处理 Tcl 代码

本章展示如何在 C 代码中处理 Tcl 脚本以及 Tcl 数学表达式。

33.1　本章出现的函数

- `int Tcl_EvalObjEx(Tcl_Interp *interp, Tcl_Obj *objPtr, int flags)`
  处理对象中包含的 Tcl 脚本。
- `int Tcl_EvalFile(Tcl_Interp *interp, CONST char *fileName)`
  处理由 *fileName* 指定的文件中的 Tcl 脚本。
- `int Tcl_EvalObjv(Tcl_Interp *interp, int objc, Tcl_Obj **objv, int flags)`
  处理一条预解析的命令，而非一个脚本。*objc* 和 *objv* 参数包含了 Tcl 命令的单词，*objv* 中的每一个对象就有一个单词。
- `int Tcl_Eval(Tcl_Interp *interp, CONST char *script)`
  处理 *script*。
- `int Tcl_EvalEx(Tcl_Interp *interp, CONST char *script, int numBytes, int flags)`
  处理 *script*。比 Tcl_Eval 效率更高。
- `int Tcl_GlobalEval(Tcl_Interp *interp, CONST char *script)`
  在全局命名空间中处理 *script*。不推荐。
- `int Tcl_GlobalEvalObj(Tcl_Interp *interp, Tcl_Obj *objPtr)`
  在全局命名空间中处理 *objPtr* 中的脚本。不推荐。
- `int Tcl_VarEval(Tcl_Interp *interp, char *string, char *string, ... (char *) NULL)`
  获取任意个任意长度的字符串参数，将它串接成一个字符串，然后处理这个字符

串。不推荐。

- `int Tcl_VarEvalVA(Tcl_Interp *interp, va_list argList)`
  与 Tcl_VarEval 相似，不过获取的是 *va_list* 而不是多个字符串。不推荐。
- `Tcl_Obj *Tcl_GetObjResult(Tcl_Interp *interp)`
  为 *interp* 返回一个对象结果。对象引用计数不增加。
- `const char *Tcl_GetStringResult(Tcl_Interp *interp)`
  为 *interp* 返回一个字符串结果。

33.2 处理 Tcl 代码

在第 31 章中我们看到了使用 Tcl_EvalFile 来处理文件中的 Tcl 脚本。Tcl 提供了其他一些用于处理脚本的函数。这些函数都把一个解释器作为它们的第一个参数，返回一个完成码，然后设置解释器的结果。最直接的就是 Tcl_Eval。

```
Tcl_Obj *result;
char script[] = "set a [expr {20 * 30}]";
...
code = Tcl_Eval(interp, script);
result = Tcl_GetObjResult(interp);
```

上述代码只是处理传给它的字符串。如果成功，返回 TCL_OK；如果失败，则返回 TCL_ERROR，并且在解释器中设置一个结果。

您可以使用 Tcl_GetObjResult 函数取得指向结果中的 Tcl 对象的指针。对象的引用计数不增加；如果需要保持指向这个对象的长期指针，就使用 Tcl_IncrRefCount 增加该对象的引用计数。对于以前的代码，使用 Tcl_GetStringResult 返回字符串的结果；开发新代码时应该使用 Tcl_GetObjResult。

提示： 由于 Tcl 8.4 版以前的内部工作方式，处理字符串时可能需要对它进行一些临时性的修改，因此要把可修改的字符串传入 Tcl_Eval 而非直接传递字符串常量，这一点是很重要的。像 Tcl_Eval(interp, "your script")这样的代码可能会失败。

如果想为脚本保证最大的效率，可以考虑使用 Tcl_EvalObjEx。除了使用对象接口，Tcl 还把传给它的代码逐段编译缓存，如果未来再用到，速度就会显著提高。

```
Tcl_Obj *script = Tcl_NewStringObj(
    "while { $i < 10 } { .... }", -1);
Tcl_IncrRefCount(script);
Tcl_EvalObjEx(interp, script, 0);
...
```

这个示例创建了一个新的字符串对象(包括了我们想要处理的脚本)，增加它的引用计数，然后处理它。Tcl_EvalObjEx 调用中的 0 就是它的“标志”参数。有两个有效的标志：一个是 TCL_EVAL_DIRECT，指示 Tcl 不要逐段编译脚本；一个是 TCL_EVAL_GLOBAL，确保脚本在全局层级处理，在全局命名空间运行，只使用全局变量。

对 Tcl_Eval 这样的过程的每一次调用都必须包含一个完整的脚本。下列代码是不能工作的。

```
char part1[] = "set ";
char part2[] = "a ";
char part3[] = "32";
Tcl_Eval(interp, part1); /* Fails. */
Tcl_Eval(interp, part2);
Tcl_Eval(interp, part3);
```

上面的代码在第一次调用 Tcl_Eval 时就会失败，因为它试图处理的脚本是不完整的。

33.3　动态创建脚本

有很多方法可以把碎片组织成一个脚本。最简单的就是把命令建立为一个列表，以便正确处理各个独立元素之间的空白。

```
Tcl_Obj *cmd;
cmd = Tcl_NewObj();
Tcl_ListObjAppendElement(interp, cmd,
    Tcl_NewStringObj("puts', -1));
Tcl_ListObjAppendElement(
    interp, cmd, Tcl_NewStringObj("hello world", -1));
Tcl_IncrRefCount(cmd);
Tcl_EvalObjEx(interp, cmd, 0);
Tcl_DecrRefCount(cmd);
```

注意因为我们把命令创建为一个列表，通过 Tcl_ListObjAppendElement，我们不必自己去处理 hello world 的引用括号，尽管它中间有空白。当 Tcl 处理这个对象时，它处理的对象类似于 puts {hello world}。使用列表比使用字符串效率更高。例如，如果某个元素是一个数字，它就可以作为一个数字加入列表，而无需使用该数字的字符串表示形式。

另一个创建动态命令的方法是使用 Tcl_EvalObjv 函数。除了一个解释器，它还获取一个 Tcl 对象数组和对象数目作为参数。

```
int i = 0;
Tcl_Obj *scriptArr[3];
scriptArr[0] = Tcl_NewStringObj("set", -1);
scriptArr[1] = Tcl_NewStringObj("a",-1);
scriptArr[2] = Tcl_NewIntObj(32, -1);
for (i = 0; i < 3; i ++) {
    /* Increase the reference counts. */
    Tcl_IncrRefCount(scriptArr[i]);
}
Tcl_EvalObjv(interp, scriptArr, 3, 0);
for (i = 0; i < 3; i++) {
    /* Decrease the reference counts. */
    Tcl_DecrRefCount(scriptArr[i]);
}
```

在这段代码中，我们创建了对象的数组，然后填充它，增加引用计数，进行处理，然后再次减小引用计数。

33.4 Tcl 表达式

除了处理 Tcl 代码之外，您还可以计算得到 Tcl 表达式的结果。表达式可用作 if 及 while 这样的命令的条件，或者用来执行对 Tcl 变量的一些数学运算。表达式的语法和应用的更多信息参见第 4 章有关 expr 的内容。

对表达式求值的基本方法是使用 Tcl_ExprTYPE 和 Tcl_ExprTYPEObj 这样的函数，这里 TYPE 是 Long、Double 或 Boolean 之一，其结果保存在对应的 C 类型变量中。例如：

```
long result = 0;
Tcl_Obj *expr = Tcl_NewStringObj("($foo / 2) + 1", -1);
Tcl_IncrRefCount(expr);
Tcl_ExprLongObj(interp, expr, &result);
Tcl_DecrRefCount(expr);
```

这段代码从字符串($foo / 2) +1 中创建了一个字符串对象，对它求值并将结果存放在结果变量中。

如果不需要将结果作为特定的 C 类型返回，有两个对表达式求值的通用函数：Tcl_ExprObj 和 Tcl_ExprString。字符串形式是最简单的，但是因为把结果保存为字符串，它牺牲了一些速度。

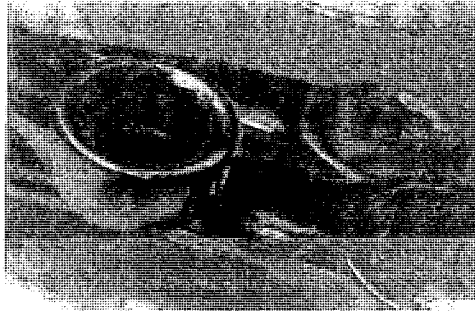
```
char *result = NULL;
char *exprstr = "2 + 2";
Tcl_ExprString(interp, exprstr);
result = Tcl_GetStringResult(interp);
```

上述代码计算 2 + 2，求值为一个字符串的结果是 4。然后您可以调用 Tcl_GetStringResult 获取这个结果。

下面我们使用基于对象的调用 Tcl_ExprObj。

```
Tcl_Obj *result;
Tcl_Obj *expr = Tcl_NewStringObj("($foo / 2) + 1", -1);
Tcl_IncrRefCount(expr);
Tcl_ExprObj(interp, expr, &result);
Tcl_DecrRefCount(expr);
...
...
Tcl_DecrRefCount(result);
/* You are responsible for lowering the reference count of the
result */
```

对象 API 使用起来稍微复杂一点，但因为它把结果作为某种数值类型的 Tcl 对象保存，而不是把它重新转换回字符串，其运行速度要快一些。



第 34 章 访问 Tcl 变量

本章的内容是如何在 C 中设置 Tcl 变量，如何读取它们的值，以及如何删除它们。我们还会讲述如何链接 C 变量和 Tcl 变量，使得当两个变量中的一个变化时，另一个也随之变化。本章还包括对变量跟踪的介绍。

34.1 本章出现的函数

本章包含的在 C 代码中操纵 Tcl 变量的函数如下。

- `Tcl_Obj *Tcl_SetVar2Ex(Tcl_Interp *interp, CONST char *name1, CONST char *name2, Tcl_Obj *newValuePtr, int flags)`

  将 *name1*(如果是一个数组元素，还有 *name2*)给出的变量设置为 *newValuePtr*。

- `CONST char *Tcl_SetVar(Tcl_Interp *interp, CONST char *varName, CONST char *newValue, int flags)`

  将变量 *varName* 设置为 *newValue*。

- `CONST char *Tcl_SetVar2(Tcl_Interp *interp, CONST char *name1, CONST char *name2, CONST char *newValue, int flags)`

  将 *name1*(如果是一个数组元素，还有 *name2*)给出的变量设置为 *newValue*。

- `Tcl_Obj *Tcl_objSetVar2(Tcl_Interp *interp, Tcl_Obj *part1Ptr, Tcl_Obj *part2Ptr, Tcl_Obj *newValuePtr, int flags)`

  将 Tcl 对象 *part1Ptr*(如果是一个数组元素，还有 *part2Ptr*)给出的变量设置为 *newValuePtr*。

- `Tcl_Obj *Tcl_GetVar2Ex(Tcl_Interp *interp, CONST char *name1, CONST char *name2, int flags)`

  取得由 *name1* 和 *name2* 指定的变量中的内容，类型为 Tcl 对象。

- `CONST char *Tcl_GetVar(Tcl_Interp *interp, CONST char *varName, int flags)`

  取得变量 *varName* 的内容，类型为字符串。

- CONST char *Tcl_GetVar2(Tcl_Interp *interp,
  CONST char *name1, CONST char *name2, int flags)
  取得由 *name1* 和 *name2* 指定的变量的内容，类型为字符串。
- Tcl_Obj *Tcl_ObjGetVar2(Tcl_Interp *interp,
  Tcl_Obj *part1Ptr, Tcl_Obj *part2Ptr, int flags)
  取得由 Tcl 对象 *part1Ptr* 和 *part2Ptr* 指定的变量的内容，类型为 Tcl 对象。
- int Tcl_UnsetVar(Tcl_Interp *interp, CONST char *varName,
  int flags)
  删除 *varName* 变量。
- int Tcl_UnsetVar2(Tcl_Interp *interp, CONST char *name1,
  CONST char *name2, int flags)
  删除由 *name1* 和 *name2* 指定的变量。
- int Tcl_TraceVar(Tcl_Interp *interp,
  CONST char *varName, int flags,
  Tcl_VarTraceProc proc, ClientData clientData)
  使得对 *varName* 的跟踪与 *flags* 指定的条件匹配时，调用函数 *proc*。
- Tcl_UntraceVar(Tcl_Interp *interp,
  CONST char *varName, int flags,
  Tcl_VarTraceProc proc, ClientData clientData)
  删除跟踪。参数必须与设置跟踪时的参数匹配。
- ClientData clientData Tcl_VarTraceInfo(Tcl_Interp *interp,
  CONST char *varName, int flags,
  Tcl_VarTraceProc proc, ClientData prevClientData)
  返回前面设置的跟踪的 *clientData*。
- int Tcl_TraceVar2(Tcl_Interp *interp,
  CONST char *name1, CONST char *name2, int flags,
  Tcl_VarTraceProc proc, ClientData clientData)
  Tcl_UntraceVar2(Tcl_Interp *interp,
  CONST char *name1, CONST char *name2, int flags,
  Tcl_VarTraceProc proc, ClientData clientData)
  ClientData Tcl_VarTraceInfo2(Tcl_Interp *interp,
  CONST char *name1, CONST char *name2, int flags,
  Tcl_VarTraceProc proc, ClientData prevClientData)
  这三个函数和前面三个相同，只不过前面三个获取的是一个变量名作为参数，这里获取的是 *name1* 和 *name2*，*name2* 用于指定数组元素。

34.2　设置变量值

Tcl 变量与 Tcl 对象不同，对象只是一些可以由 Tcl 使用的值，而变量则是在脚本层级可以通过一个变量名访问的客体。

创建 Tcl 变量的最简单的方法是使用 Tcl_SetVar 命令。

```
Tcl_SetVar(interp, "foo", "42", 0);
/* Sets the value of the foo variable to "42". */
```

变量名称和它的值都以字符串形式给出，这说明用这种方法创建数值变量的效率不高。如果不关心速度，那这是最简单的设置变量值的方法。变量可以是一个普通变量，或是一个数组元素——例如 foo 或 foo(bar)。除此之外，这里根本就不加以解析，也不会发生任何替换。如果愿意，可以使用字符串[$a]作为变量名称，不过您不应该使用这样的名称，因为它们会使您的代码变得令人迷惑，难以阅读。

这个命令的基于对象的版本如下：

```
Tcl_Obj *intObj;
intObj = Tcl_NewIntObj(42);
Tcl_IncrRefCount(intObj);
Tcl_SetVar2Ex(interp, "foo", NULL, intObj, 0);
Tcl_DecrRefCount(intObj);
...
```

运行此代码片段时，它会像前面那句代码一样创建 foo 变量。不同之处在于它的值是一个包含了数值 42 的对象，如果接下来对 foo 的命令针对数值进行操作，这种变量效率更高。

Tcl_SetVar2Ex 函数还可以使用第二个和第三个参数来创建数组元素，例如：

```
Tcl_SetVar2Ex(interp, "foo", "bar", intObj, 0);
```

这会创建数组元素 foo(bar)。

大多数变量函数的最后一个参数——上例中的 0，是由位或生成的标志。

- TCL_GLOBAL_ONLY
  只在全局命名空间中查找变量，即使函数是在很深的嵌套过程中调用的。
- TCL_NAMESPACE_ONLY
  只在当前命名空间中查找变量。
- TCL_LEAVE_ERR_MSG
  如果在创建变量时发生错误则将解释器的结果设置为错误。如果不指定，大多数设置命令只会返回 NULL。如果命令不能设置需要的变量就应该失效，则这个标志指定的功能是很有用的。
- TCL_APPEND_VALUE
  不去替代变量已有的值，而是将值添加到它后面。如果变量还不存在，则正常创建该变量。

- TCL_LIST_ELEMENT
  变量的新值转化为列表元素(如果必要会加引号)。根据是否设置TCL_APPEND_VALUE，变量或者被设置为包含元素的列表，或者设置为元素并添加到变量所包含的值当中。

除了刚才介绍的 Tcl_SetVar 和 Tcl_SetVar2Ex 函数以外，Tcl API 提供了 Tcl_ObjSetVar2，获取作为变量名称和/或数组元素名称的 Tcl 对象。例如，创建一个数字列表。

```
int i = 0;
Tcl_Obj *intObj;
...
for(i = 1; i <= 10; i++){
    intObj = Tcl_NewIntObj(i);
    Tcl_IncrRefCount(intObj);
    Tcl_SetVar2Ex(interp, "numlist", NULL, intObj,
                TCL_LIST_ELEMENT | TCL_APPEND_VALUE);
    Tcl_DecrRefCount(intObj);
}
```

在 Tcl 中，输出 numlist 变量的结果如下：

```
    puts $numlist
⇒ 1 2 3 4 5 6 7 8 9 10
```

34.3 读取变量

“写入”对应的是 Tcl_SetVar，“读取”对应的是 Tcl_GetVar。

```
char *value;
value = Tcl_GetVar(interp, "foo", 0);
```

这会返回一个指向变量 foo 的值的字符串指针。返回的字符串属于 Tcl，因此如果希望保持它或修改它，应使用 strdup 生成一份副本。对这个变量调用 Tcl_SetVar 或 Tcl_SetVar2 就会使这个指针失效，因此如果想要保存它，就必须立即复制它。

用于各种 GetVar 操作的标志是 TCL_GLOBAL_ONLY 及 TCL_LEAVE_ERR_MSG，其含义与 SetVar 函数中的标志含义相同。

基于对象的 GetVar 函数是 Tcl_GetVar2Ex，它返回一个 Tcl 对象。

```
Tcl_Obj *value;
value = Tcl_GetVar2Ex(interp, "foo", NULL, 0);
```

有了 Tcl_Obj，就可以使用 Tcl_GetIntFromObj 来获取其中的整型值，然后进行操作，这比解析字符串 42 来获得它的整型值要有效率得多。

```
int intval = 0;
if (Tcl_GetIntFromObj(interp, value, $intval) != TCL_OK) {
    return TCL_ERROR;
}
intval = intval * 42;
```

34.4　删除变量

使用 Tcl_UnsetVar 或 Tcl_UnsetVar2 命令可以移除变量。例如：

```
Tcl_UnsetVar(interp, "foo(bar)", 0);
```

或

```
Tcl_UnsetVar2(interp, "foo", "bar", 0);
```

就从 foo 数组中移除了 bar 元素。这些函数的效果与 Tcl unset 命令相同。与上面的代码等价的 Tcl 语句如下：

```
unset foo(bar)
```

这些函数成功时返回 TCL_OK，如果变量不存在，或者因为什么原因不能删除，则返回 TCL_ERROR。TCL_GLOBAL_ONLY 和 TCL_LEAVE_ERR_MSG 标志可以在这些调用中使用。如果指定的是一个数组名称，而没有给定具体的元素，则会移除整个数组。

34.5　链接 Tcl 和 C 变量

Tcl 提供了一种名为变量链接的机制，用于把一个 Tcl 变量和一个 C 变量关联起来。每当读取 Tcl 变量时，值由与其链接的 C 变量提供，每当对这个 Tcl 变量进行写操作时，新的值也被存入与其链接的 C 变量。函数 Tcl_LinkVar 创建变量链接。考虑下面的 C 代码：

```
int value = 32;
...
Tcl_LinkVar(interp, "x", (void *) &value, TCL_LINK_INT);
```

这里把 Tcl 变量 x 链接到 C 变量 value。每当读取 Tcl 变量 x 时，Tcl 将 value 转化为一个十进制数的字符串，然后返回这个字符串作为 x 的值(本例中是 32)。如果修改了 value，那么下一次读取 Tcl 变量 x 时就会返回新的值。每当对 Tcl 变量 x 进行写操作时，x 的新值也会由 Tcl 对象转化为整型数，存放在 value 中。如果一个 Tcl 脚本试图向 x 中写入一个不能理解为整数的值，写操作就会被拒绝并产生错误。

```
    set x "oops!"
⇒   can't set "x": variable must have integer value
```

Tcl_LinkVar 的最后一个参数表明了 C 变量的类型。前面这个示例中，该类型是 TCL_LINK_INT，表示 C 变量是一个整型数，仅有整数值可以存入这个 Tcl 变量。表 34.1 列出了类型参数可选用的值，以及它们对应的 C 类型。对于由 Tcl_LinkVar 支持的所有数值类型，不能被转化为指定类型的对 Tcl 变量的赋值(例如，非数字的或超范围的值)都会被拒绝，产生一个 Tcl 错误。

表 34.1 Tcl_LinkVar 允许的类型参数

类型标志	C 类 型
TCL_LINK_INT	int
TCL_LINK_UINT	unsigned int
TCL_LINK_CHAR	char
TCL_LINK_UCHAR	unsigned char
TCL_LINK_SHORT	short
TCL_LINK_USHORT	unsigned short
TCL_LINK_LONG	long
TCL_LINK_ULONG	unsigned long
TCL_LINK_DOUBLE	double
TCL_LINK_FLOAT	float
TCL_LINK_WIDE_INT	Tcl_WideInt
TCL_LINK_WIDE_UINT	Tcl_WideUInt
TCL_LINK_BOOLEAN	int
TCL_LINK_STRING	char *

对于 TCL_LINK_STRING 类型，如果 C 变量的值不是 NULL，那它必须指向由 Tcl_Alloc 或 ckalloc 分配空间的一个字符串。当 Tcl 变量被修改时，会释放旧的字符串，为新字符串分配空间，保存新的字符串值。Tcl 变量可以赋给任意字符串值。读取 Tcl 变量时如果与其链接的 C 变量是 NULL 则返回字符串 NULL。

提示：仅全局地或动态地分配空间的 C 变量应该用来建立链接，不要与堆栈中的那些 C 变量链接。

例如，要链接到一个字符串，应该如下操作：

```
char *foo = "Hello, World";
/* Note the pointer to the pointer. */
Tcl_LinkVar(interp, "hi", (void *) &foo, TCL_LINK_STRING);
```

标志 TCL_LINK_READ_ONLY 也可以与类型标志进行或操作，例如：

```
Tck_LinkVar(interp, "x", (void *), &value,
    TCL_LINK_INT | TCL_LINK_READ_ONLY);
```

上述代码使得 Tcl 变量只读；任何从 Tcl 中修改该变量的尝试都会被拒绝，并产生错误。

如果一个 Tcl 变量被链接，然后又被删除了，那么 Tcl 会自动重建这个变量。直到移除链接之后，Tcl 变量才能被永久删除。函数 Tcl_UnlinkVar 移除前面由 Tcl_LinkVar 建立的变量链接。例如，下面这条语句就移除了前面创建的链接。

```
Tcl_UnlinkVar(interp, "x");
```

34.6　设置与删除变量跟踪

变量跟踪允许您设定一个 C 函数，在变量被读取、写入或删除时调用。跟踪可以用于很多目的。例如，在 Tk 中，可以配置一个按钮组件，显示一个变量的值，并且在变量被修改时自动更新。这个功能就是由变量跟踪实现的。跟踪还可以用于调试、创建只读变量以及其他很多目的。

提示： *尽管跟踪功能十分强大，但应谨慎使用，因为它们很容易形成“魔法变量”，就是在用户看来其行为十分怪异的那种变量。*

调用 Tcl_TraceVar 和 Tcl_TraceVar2 创建变量跟踪，示例如下：

```
Tcl_TraceVar(interp, "x", TCL_TRACE_WRITES,
    WriteProc, (ClientData)NULL);
```

上述代码对 interp 中的 x 创建了一个写跟踪，意味着只要 x 被修改，就会调用 WriteProc。Tcl_TraceVal 的第三个参数是标志位或的结果，用于选择要跟踪的操作，包括：TCL_TRACE_READS，表示读操作；TCL_TRACE_WRITES，表示写操作；TCL_TRACE_UNSETS，表示删除操作。另外，标志 TCL_GLOBAL_ONLY 可以用来强制要求变量名称在全局空间中解析。Tcl_TraceVar 和 Tcl_TraceVar2 正常时返回 TCL_OK。如果出现错误，它们在解释器中设置错误结果，并返回 TCL_ERROR。

库函数 Tcl_UntraceVar 和 Tcl_UntraceVar2 删除变量跟踪。例如，下面的调用就移除了前面设置的跟踪。

```
Tcl_Untrace(interp, "x", TCL_TRACE_WRITE,
        WriteProc, (ClientData)NULL);
```

Tcl_UntraceVar 找到指定的变量；查找与标志、跟踪函数以及它的参数指定的 clientData 匹配的跟踪；如果这个跟踪存在，就将其移除。如果没有匹配的跟踪，Tcl_UntraceVar 就什么也不做。Tcl_UntraceVar 和 Tcl_UntraceVar2 接受与 Tcl_TraceVar 相同的标志。

34.7　跟踪回调

前面几节中像 WriteProc 这样的跟踪回调函数必须与如下原型匹配：

```
typedef char *Tcl_VarTraceProc(
    ClientData clientData, Tcl_Interp *interp,
    char *name1, char *name2, int flags);
```

参数 *clientData* 和 *interp* 与传给 Tcl_TraceVar 或 Tcl_TraceVar2 中的对应参数相同。*clientData* 通常指一个包含了跟踪回调需要的信息的结构体。*name1* 和 *name2* 给出了变量的名称，格式与 Tcl_SetVar2 中的参数相同。*flags* 是指定标志位或的结果。TCL_TRACE_READS、TCL_TRACE_WRITES 以及 TCL_TRACE_UNSETS 的设置表明了

触发跟踪的操作。TCL_GLOBAL_ONLY 的设置指明变量是一个全局变量，从当前运行环境中不可见；跟踪回调必须将这个标志传回 Tcl_GetVar2 这样的函数，才能访问变量。TCL_TRACE_DESTROYED 和 TCL_INTERP_DESTROYED 标志位只在很特殊的情况下设置，详见 34.10 节。

对于读取跟踪，回调在 Tcl_GetVar 或其他变量读取操作返回变量值之前调用；如果回调修改了变量的值，那么返回的是修改过的变量。对于写入跟踪，回调在变量的值被改变之后调用。回调可以修改变量的值以覆盖这次改动，被回调再修改的值会作为 Tcl_SetVar 或其他写入函数的结果返回。对于删除跟踪，回调在变量被删除之后调用，因此回调不能访问跟踪的那个变量。删除回调可以在一个变量被删除时发生，在一个 Tcl 过程返回时(返回时会删除它的所有局部变量)发生，或者在一个解释器被销毁时(销毁时会删除解释器中的所有变量)发生。

跟踪回调可以调用 Tcl_GetVar 以及 Tcl_SetVar 或类似的函数来对跟踪的变量进行读写操作。回调运行时，对该变量的所有跟踪都暂时禁用，因此在回调函数中对 Tcl_GetVar、Tcl_SetVar 及类似函数的调用不会再触发更多的回调。前面已经讲到，在变量被删除之后才调用删除跟踪，所以在删除回调中试图读取跟踪的变量是会失败的。不过，可以在删除回调中对这个变量进行写入操作，这里实际上就会创建一个新的变量。

下列代码为变量 x 设置了写入跟踪，每当它被修改时就输出它的值。

```
Tcl_TraceVar(interp, "x", TCL_TRACE_WRITES, Print,
(ClientData)NULL);
...
char *Print(ClientData clientData, Tcl_Interp *interp,
        char *name1, char *name2, int flags) {
    char *value;
    value = Tcl_GetVar2(interp, name1, name2,
            flags & TCL_GLOBAL_ONLY);
    if (value != NULL) {
        if (name2 == NULL) {
            printf("new value of %s is %s\n",
                    name1, value);
        } else {
                printf("new value of %s(%s) is %s\n",
                name1, name2, value);
        }
    }
    return NULL;
}
```

Print 必须把它的 flags 参数中的 TCL_GLOBAL_ONLY 位传给 Tcl_GetVar2，确保变量被正确地访问。Tcl_GetVar2 应该永远都不会返回错误，但是 Print 还是进行检查，如果这里发生错误，就不去尝试输出变量的值。这个示例中我们没有使用 Tcl_GetVar 的 Tcl_Obj 变体，因为 Tcl_GetVar2 更简单、更直接。

跟踪回调通常的返回值是 NULL；一个非 NULL 的值表明有错误发生。这种情况下返回值必须是指向包含了错误消息的静态字符串的指针。取消跟踪的访问，将错误消息返回给访问的发起者。例如，如果访问是由 Tcl_GetVar 调用的，它返回 NULL，在标志

TCL_LEAVE_ERR_MSG 被设置的情况下在解释器中设置一个错误结果。

下面这段代码使用了跟踪，设置变量 zip 为只读，其值为 94114。

```
Tcl_Obj *numobj = Tcl_NewIntObj(94114);
Tcl_IncrRefCount(numobj);
Tcl_TraceVar(interp, "zip", TCL_TRACE_WRITES, Reject,
                (ClientData)numobj);
...
char *Reject(ClientData clientData, Tcl_Interp *interp,
            char *name1, char *name2, int flags) {
    Tcl_Obj *correct = (Tcl_Obj *) clientData;
    Tcl_SetVar2Ex(interp, name1, name2,
            correct, flags & TCL_GLOBAL_ONLY);
    return "variable is read-only";
}
```

函数 Reject 就是向 zip 进行写入操作时的跟踪回调。它返回一条错误消息，取消写入访问。因为在调用 Reject 之前 zip 已经被修改了，Reject 必须重置变量的正确值，消除写入操作的影响。正确值使用了它的 clientData 参数传递。这一实现函数可用于多个不同的只读变量的写入回调；各变量不同的正确值可以分别传给 Reject。

34.8 全数组跟踪

您可以向 Tcl_TraceVar 或 Tcl_TraceVar2 指定一个数组名称，而不指定元素名，从而建立对整个数组进行的跟踪。这就是全数组跟踪：对数组的任何一个元素进行特定的操作时，都会调用回调函数。如果整个数组被删除，回调函数只被调用一次，调用时 *name1* 为数组名称，*name2* 参数为 NULL。

34.9 多重跟踪

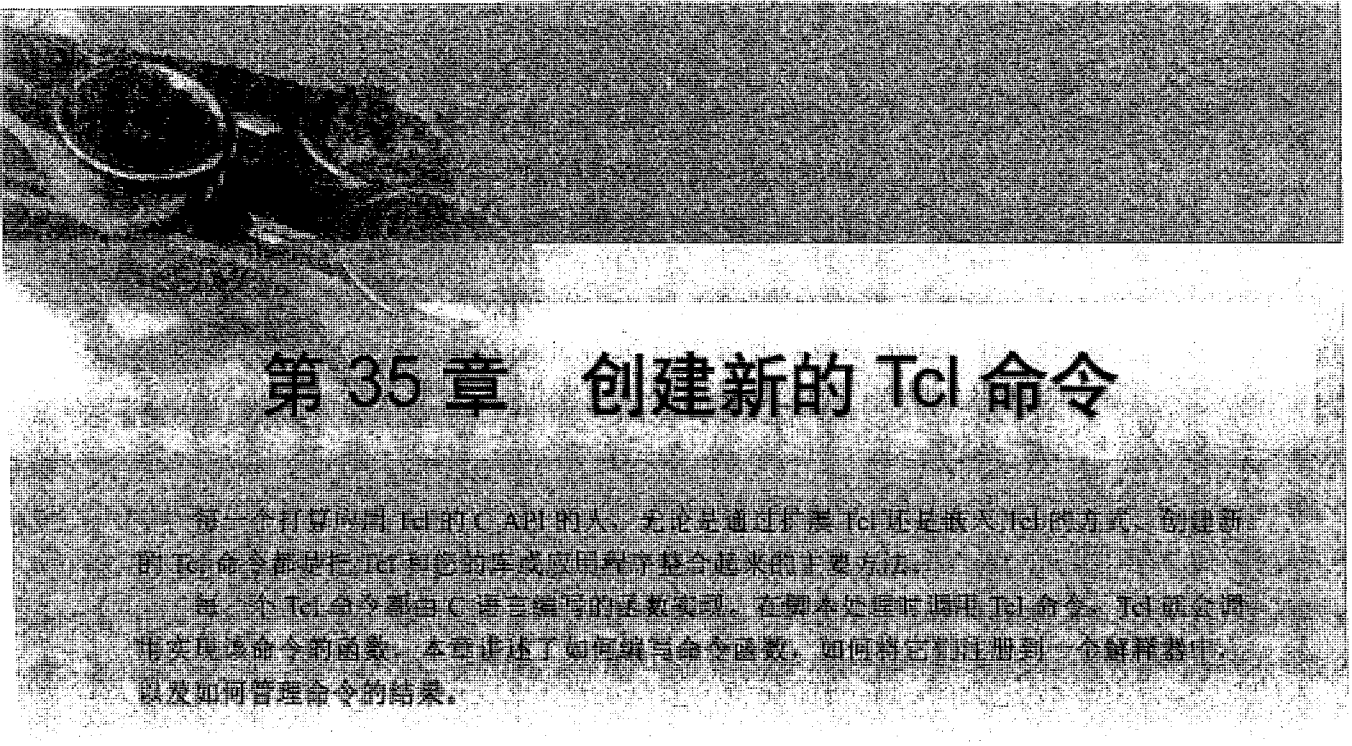
可以对同一个变量进行多重跟踪。当跟踪触发时，会对访问变量分别调用相应的回调。回调调用的顺序是从最后创建的开始，直到最早创建的。如果同时存在全数组跟踪和独立元素跟踪，先调用全数组回调，再调用元素回调。如果某个回调返回了错误，那后面的回调就不再被调用。

34.10 删除回调

删除回调与读、写回调在某些方面有所不同。首先，删除回调不能返回错误状态，它们必须总是成功的。其次，删除回调有两个独有的标志：TCL_TRACE_DESTROYED 和 TCL_INTERP_DESTROYED。删除一个变量时，也会删除它的所有跟踪；但变量上的删除跟踪仍然会被调用，但它们还会传递 TCL_TRACE_DESTROYED 标志，表示跟踪已经被删除而且不会再被调用。如果一个数组元素被删除，而同时对它所在的数组有全数组跟

踪，那么删除跟踪不会被移除，回调时标志 TCL_TRACE_DESTROYED 也不会设置。

如果 TCL_INTERP_DESTROYED 标志在删除回调中设置，就标志着包含该变量的解释器已经被销毁。这种情况下，回调必须完全避免使用那个解释器，因为此时在进程中该解释器的状态是已被删除。这个回调应该只是清理它自己内部的数据结构。



第35章 创建新的Tcl命令

对一个打算利用Tcl的C API的人，无论是通过扩展Tcl还是嵌入Tcl的方式，创建新的Tcl命令都是把Tcl和您的库或应用程序整合起来的主要方法。

每一个Tcl命令都由C语言编写的函数实现。在脚本处理中调用Tcl命令。Tcl就会调用实现该命令的函数。本章讲述了如何编写命令函数，如何将它们注册到一个解释器中，以及如何管理命令的结果。

35.1 本章出现的函数

- `Tcl_Command Tcl_CreateObjCommand(Tcl_Interp *interp,`
  `CONST char *cmdName, Tcl_ObjCmdProc proc,`
  `ClientData clientData,`
  `Tcl_CmdDeleteProc deleteProc)`
  在Tcl中创建一个新的命令，命令名为*cmdName*，当调用这个命令时，就会运行函数*proc*。
- `Tcl_WrongNumArgs(Tcl_Interp *interp, int objc,`
  `CONST Tcl_Obj *objv[], CONST char *message)`
  生成一条标准错误消息，将它存放在*interp*的结果对象中。这条消息包含了*objv*的*objc*初始元素以及*message*。
- `int Tcl_DeleteCommand(Tcl_Interp *interp,`
  `CONST char *cmdName)`
  删除命令*cmdName*。
- `int Tcl_DeleteCommandFromToken(Tcl_Interp *interp,`
  `Tcl_Command token)`
  删除由*token*指向的命令。
- `Tcl_SetObjResult(Tcl_Interp *interp, Tcl_Obj *objPtr)`
  将*objPtr*设置为*interp*的结果，并增加其引用计数。这个函数将替换掉任何已经存在的结果，并减少旧的结果对象的引用计数。
- `Tcl_SetResult(Tcl_Interp *interp, char *result,`
  `Tcl_FreeProc *freeProc)`
  将*result*安排为*interp*中的当前Tcl命令的结果，替换掉任何现有的结果。

freeProc 参数指定了结果参数的存储问题如何管理。

- `Tcl_AppendResult(Tcl_Interp *interp, char *result,`
  `char *result , ... , (char *) NULL)`
  `Tcl_AppendResultVA(Tcl_Interp *interp, va_list argList)`
  获取每一个 *result* 参数，将它们依次添加到 *interp* 相应的当前结果中。Tcl_AppendResultVA 与 Tcl_AppendResult 基本相同，只不过它获取的不是很多个结果参数，而是一个参数列表。
- `Tcl_AppendElement(Tcl_Interp *interp, char *element)`
  仅获取一个 *element* 参数，将它作为 Tcl 列表的元素添加到当前结果中。
- `Tcl_ResetResult(Tcl_Interp *interp)`
  清除 *interp* 的结果，让它的结果处于正常初始化时的空状态。如果结果是一个对象，减小它的引用计数。
- `int Tcl_GetCommandInfo(Tcl_Interp *interp,`
  `CONST char *cmdName, Tcl_CmdInfo infoPtr)`
  获取有关命令 *cmdName* 的信息，放入 *infoPtr* 中。
- `int Tcl_SetCommandInfo(Tcl_Interp *interp,`
  `CONST char *cmdName, Tcl_CmdInfo infoPtr)`
  修改与命令 *cmdName* 相关的函数和客户数据。
- `int Tcl_GetCommandInfoFromToken(Tcl_Command token,`
  `Tcl_CmdInfo infoPtr)`
  获取由 *token* 指向的命令的信息，放入 *infoPtr* 中。
- `int Tcl_SetCommandInfoFromToken(Tcl_Command token,`
  `Tcl_CmdInfo infoPtr)`
  修改由 *token* 指向的命令的相关函数和客户数据。
- `CONST char *Tcl_GetCommandName(Tcl_Interp *interp,`
  `Tcl_Command token)`
  获取由 *token* 指向的命令的名称。
- `void Tcl_GetCommandFullName(Tcl_Interp *interp,`
  `Tcl_Command token, Tcl_Obj *objPtr)`
  为 *token* 的命令产生完全限定名称。该名称(包括所有的命名空间前缀)添加到由 *objPtr* 指定的对象中。
- `Tcl_Command Tcl_GetCommandFromObj(Tcl_Interp *interp,`
  `Tcl_Obj *objPtr)`
  返回由 *Tcl_Obj* 中的名称指定的命令的标记。命令名称以相对当前命名空间的模式解析。如果未找到命令，则返回 NULL。
- `Tcl_Trace Tcl_CreateObjTrace(Tcl_Interp *interp,`
  `int level, int flags, Tcl_CmdObjTraceProc objProc,`
  `ClientData clientData`

Tcl_CmdObjTraceDeleteProc *deleteProc*)

创建一个跟踪。在运行解释器 *interp* 中的命令之前调用 *objProc* 函数。只对处于或低于嵌套层级 *level* 的命令进行跟踪。返回的跟踪可以传给 Tcl_DeleteTrace。

- Tcl_Trace Tcl_CreateTrace(Tcl_Interp **interp*,
  int *level*, Tcl_CmdTraceProc *func*,
  ClientData *clientData*)

  与 Tcl_CreateObjTrace 相似，只不过用于使用较早期的接口实现的命令函数。
- Tcl_DeleteTrace(Tcl_Interp **interp*, Tcl_Trace *trace*)

  删除一个跟踪。

35.2 命令函数

基于对象的命令函数的接口，由 Tcl_CmdProc 函数原型定义。

```
typedef int Tcl_ObjCmdProc(
    ClientData clientData, Tcl_Interp *interp,
    int objc, Tcl_Obj *CONST objv[]);
```

换句话说，实现新的 Tcl 命令的函数都有这样的特征。这替代了旧式的基于字符串的命令，那种命令函数的特征如下——这里给出它只是为了帮助理解需要更新的旧式代码。

```
typedef int Tcl_CmdProc(
    ClientData clientData, Tcl_Interp *interp,
    int argc, char *argv[]);
```

命令函数在执行对应的 Tcl 命令时被调用，会为它传递 4 个参数。第一个参数是 *clientData*，相关内容参见 35.7 节。第二个参数 *interp* 是执行命令的那个解释器。第三个和第四个参数的含义与 C main 程序中的 *argc* 和 *argv* 相同：*objc* 指明了传给 Tcl 命令的参数的总数量(多少个单词)，而 *objv* 是一个指针数组，指向这些单词的 Tcl_Obj 值。在调用命令函数之前 Tcl 会处理所有的特殊字符，例如$和[]等，因此 *objv* 中的值是替换过程完成后的值。

命令的名称也计入 *objc* 中(就是说命令 cmd foo 的 *objc* 是 2)，且被包含为 *objv* 的第一个元素。*objv*[*objc*]为 NULL。命令函数返回两个值，就像 Tcl_Eval 和 Tcl_EvalFile 那样。C 函数返回一个整型的完成码，例如 TCL_OK 或 TCL_ERROR，并且用 Tcl_SetObjResult 在 Tcl 解释器中设置一个结果。Tcl_WrongNumArgs 调用也常用于表示传给 Tcl 命令的参数数量不正确的错误情况。

下面是一个名为 eq 的新命令的命令函数，该命令比较它的两个参数是否是相同的字符串。

```
static int
EqCmd(ClientData clientData, Tcl_Interp *interp,
    int objc, Tcl_Obj *CONST objv[] {
    char *arg1;
    char *arg2;
    Tcl_Obj *result;
    if (objc != 3) {
```

```
        Tcl_WrongNumArgs(interp, 1, objv, "string1 string2");
        return TCL_ERROR;
    }
    arg1 = Tcl_GetString(objv[1]);
    arg2 = Tcl_GetString(objv[2]);
    if (strcmp(arg1, arg2) == 0) {
        result = Tcl_NewBooleanObj(1);
    } else {
        result = Tcl_NewBooleanObj(0);
    }
    Tcl_SetObjResult(interp, result);
    return TCL_OK;
}
```

EqCmd 检查它是否正好得到了两个参数(objc 的值是 3，这个计数包括了命令名单词在内)，如果不是，它就使用 Tcl_WrongNumArgs 设置错误结果，然后返回 TCL_ERROR。否则，它从它的两个参数中取得字符串，比较它们，创建一个布尔型的对象，表示它们相同或不相同。然后它返回 TCL_OK 表示命令正常完成。EqCmd 没有使用它的 clientData 参数，但是这个参数在其他很多 Tcl 命令中扮演着重要的角色，详情参见后文。

提示：这个命令不是检查两个对象的相等性，而只是检查它们的字符串。如果是两个数字，如 1 和 1.0，EqCmd 会认为它们是不同的。

命令函数应该以只读的方式对待 objv 数组中的内容。一般来说，让命令函数修改这些对象是不安全的。

35.3 注册命令

为了能在 Tcl 中调用一个命令函数，必须先调用 Tcl_CreateObjCommand 注册它，格式如下：

```
Tcl_Command token Tcl_CreateObjCommand(
    Tcl_Interp *interp, char *commandName,
    Tcl_ObjCmdProc *proc, ClientData clientData,
    Tcl_CmdDeleteProc deleteProc);
```

这就是把 Tcl 中的字符串与实现它的 C 函数关联起来的“咒语”。下面就是 31.3 节中的简单程序，只是增加了对 Tcl_CreateObjCommand 的调用。

```
#include <stdio.h>
#include <tcl.h>

int main(int argc, char *argv[]) {
    Tcl_Interp *interp;
    char *result;
    int code;
    if (argc != 2) {
        fprintf(stderr, "Wrong # arguments: ");
        fprintf(stderr, "should be \"%s fileName\" \n",
                argv[0]);
        exit(1);
    }
```

```
    interp = Tcl_CreateInterp();
    Tcl_CreateObjCommand(interp, "eq", EqCmd,
                         (ClientData) NULL,
                         (Tcl_CmdDeleteProc *) NULL);
    code = Tcl_EvalFile(interp, argv[1]);
    result = Tcl_GetStringResult(interp);
    if (code != TCL_OK) {
        printf("Error was: %s\n", result);
        exit(1);
    }

    printf("Result was: %s\n", result);
    exit(0);
}
```

传给 Tcl_CreateObjCommand 的第一个参数指明了将会使用这个命令的解释器。第二个参数指定了命令的名称，第三个参数指定了它的命令函数。第四个和第五个参数在 35.7 节中讨论，在本例这样的简单函数中它们可以指定为 NULL。Tcl_CreateObjCommand 为 interp 创建了名为 eq 的新命令。如果已经存在同名的命令，则原来的命令会被无警告地删除。在 interp 中调用 eq 时，Tcl 调用 EqCmd 来完成它的功能。Tcl_CreateObjCommand 返回一个 Tcl_Command 标记(“令牌”)，这个标记可以用于操纵及获取关于这个命令的信息，更多信息参见 35.8 节和 35.9 节。

如果 EqCmd 的代码位于与 main 函数相同的文件中，它可以像 31.3 节中的简单程序一样编译调用。或者，也可以把 EqCmd 编译为扩展包，然后按第 36 章介绍的方式加载。这两种情况下，脚本都可以使用新的 eq 命令了。

```
    eq abc def
⇒   0
    eq 1 1
⇒   1
    set w .dlg
    set w2 .dlg.ok
    eq $w.ok $w2
⇒   1
```

处理脚本时，Tcl 在调用命令函数之前，先进行所有的命令行替换，因此上面最后一条 eq 命令最终调用 EqCmd 时，objv[1]和 objv[2]中的对象都是字符串.dlg.ok。

Tcl_CreateObjCommand 通常由应用程序在初始化期间调用，以注册应用程序特有的命令，不过在应用程序的运行过程中，随时都可以创建新的命令——甚至可以通过其他命令来创建。例如，proc 命令为每一个定义的 Tcl 过程创建一个新的命令，而 Tk 为每一个新组件创建一个新的组件命令。在 35.7 节和 35.9 节中，您会看到用一个命令的命令函数创建新的命令的示例。

由 Tcl_CreateObjCommand 创建的命令与 Tcl 的内建命令是不能区分的。每一个内建命令都有与 EqCmd 形式相同的命令函数，而且您可以调用 Tcl_CreateObjCommand，为命令名称指定一个新的命令函数，从而重新定义内建命令。

35.4 结果协议

EqCmd 函数通过调用 Tcl_SetObjResult 函数返回一个结果。这是设置结果的最高效的方法。例如在前文的示例中，我们创建了一个布尔类型结果，因此如果 eq 命令的结果由另一个需要获取一个布尔型参数的命令读取，就无需类型转换。如果我们提供的是字符串结果，那就必须创建一个字符串变量，然后把它解析回布尔型值(整数)。

在解释器中将一个对象设置为结果时，Tcl 自动地增加它的引用计数，因此不必显式地进行该操作。

35.5 Tcl_AppendResult

管理结果还有另一个有用的且被普遍接受的方法(Tcl 内部就使用这种方法)，那就是 Tcl_AppendResult，它可以很容易地把一块一块的结果组建起来。它可以获取任意多个字符串作为参数，将它们依次添加到解释器的结果当中。随着结果的长度的增加，Tcl_AppendResult 会为它分配新的内存。可以反复调用 Tcl_AppendResult 以逐渐增加的方式创建一个很长的结果，而且这样做的效率很高，即使结果所占空间非常大(它会分配多余的内存，因此不会在每次被调用时都把现有的结果复制到更大块的内存空间去)。下面是来自 Tcl 源码的一个示例，它在不能打开文件时设置错误消息。

```
Tcl_AppendResult(interp, "couldn't read file \"".
                 filename, "\": ",
                 Tcl_PosixError(interp),
                 (char *) NULL);
return TCL_ERROR;
```

Tcl_AppendResult 中的 NULL 参数标记了添加的字符串的结尾。

Tcl_AppendElement 与 Tcl_AppendResult 类似，只不过它一次只向结果中添加一个字符串，而且把它作为一个列表元素而非原始字符串添加。在现代 Tcl 应用程序中，最好创建一个列表对象，把它作为一个参数提供给 Tcl_SetObjResult。

如果为一个解释器设置了结果，然后决定抛弃它(例如，遇到了一个错误，您想把当前的结果替换为错误消息)，需要调用函数 Tcl_ResetResult。它会释放当前结果，将解释器的结果重置为初始状态。然后您可以用任意的普通方法把新的值存放到结果中。如果准备用 Tcl_SetObjResult 或 Tcl_SetResult 来保存新的结果，就不必调用 Tcl_ResetResult 了，因为这些函数本身就会释放所有现存的结果。

35.6 Tcl_SetResult 和 interp->result

在 Tcl 对象出现之前，使用的是 Tcl_SetResult 函数，在更老的版本中还可以直接操纵解释器结构体的相关域来设置结果。

提示： 这样做又慢又容易出错，在未来的 Tcl 发行版中可能不再支持，而且这已经被批评了很多年。我们在这里讲述它是为了帮助需要处理以前的代码(而且，我们希望，要更新那些代码)的人。

在 Tcl 库之外可见的 Tcl_Interp 结构体的完整定义如下：

```
typedef struct Tcl_Interp {
    char *result;
    Tcl_FreeProc *freeProc;
    int errorLine;
} Tcl_Interp;
```

第一个域，result，指向解释器的当前结果。可以直接将它设置为一个字符串，例如：

```
interp->result = "Maximum temperature exceeded!";
```

Tcl 还默认提供了不大的内存空间，可以用于直接写入。空间的具体大小是由 TCL_RESULT_SIZE 指定的，这个值保证至少有 200。下面的代码将其用于数值结果。

```
sprintf(interp->result, "%d", argc);
```

也可以分配内存(通过 malloc 之类)，然后将它赋给 interp->result。这时就需要用到 interp 结构体中的第二项 freeProc。如果结果是由动态分配的内存设置的，freeProc 函数就用于释放不再使用的内存。freeProc 必须有如下形式的定义：

```
typedef void Tcl_FreeProc(char *blockPtr);
```

调用这个函数需要提供一个参数，其内容是 interp->result 中存放的地址。在更早期的代码中，malloc 用于动态分配内存，而在新代码中则使用 ckalloc 或 Tcl_Alloc。因此，interp->freeProc 常常设置为 free 函数。

Tcl_SetResult 是直接管理 interp->result 的发展产物，虽然它不再使用内存时仍然需要一个函数来负责释放。传给 Tcl_SetResult 的第一个参数是一个解释器，第二个参数是用作结果的字符串，第三个参数给出了有关字符串的更多信息。TCL_STATIC 表示那个字符串是静态的，于是 Tcl_SetResult 就仅仅把它的地址存放到 interp->result。第三个参数的值为 TCL_VOLATILE 则表示这个字符串将会改变(例如，它存放在函数的栈帧当中)，因此必须复制这个结果。Tcl_SetResult 将这个字符串复制到预先分配的空间中；如果放不下，就为这个结果分配新的内存空间，将结果复制到那里(正确设置 interp->freeProc)。如果第三个参数是 TCL_DYNAMIC，就表示字符串是由 malloc 分配的，而且属于 Tcl：Tcl_SetResult 像前面讲的那样设置 interp->freeProc 为 free。最后，第三个参数可能是一个函数的地址，适用于 interp->freeProc；这种情况下字符串是动态分配的，而 Tcl 最终会调用指定的函数来释放它。

提示： 再次说明，在开发新代码时不要使用本节的这些方法。如果在现有的代码中遇到它们，请花几分钟时间来升级那些代码。

35.7 clientData 和删除回调

Tcl_CreateObjCommand 的第四个和第五个参数，clientData 和 deleteProc，在 35.3 节中没有进行讨论，但是当命令与“对象”(不是 Tcl_Obj 对象，而是 30.3 节讲述的那种对象)相关时它们十分有用。参数 clientData 用来将一个单词的值传入一个命令函数(传递的通常是一个对象的 C 数据结构体的地址)。Tcl 保存了 clientData 的值，将它用作命令函数的第一个参数。ClientData 类型的大小足够保存一个整数或一个指针的值。

clientData 的值用于与回调函数连接。回调，是一个函数，它的地址传给一个 Tcl 库函数并保存在 Tcl 数据结构中。然后，在某些特定的时刻，就使用这个地址调用函数(“把它调回来”)。命令函数就是回调的示例：Tcl 把函数的地址和 Tcl 命令的名称关联起来，当调用这个命令时就会调用相应的函数。在 Tcl 或 Tk 中指明回调的时候，参数 clientData 和函数的地址一起提供，clientData 的值会传递给回调，作为它的第一个参数。

Tcl_CreateObjCommand 的 deleteProc 参数指定了一个删除回调。如果它的值不是 NULL，它就是当命令被删除时 Tcl 要调用的函数的地址。这个函数必须与如下原型匹配：

```
typedef void Tcl_CmdDeleteProc(ClientData clientData);
```

删除回调获取一个参数，即创建命令时指定的 *clientData* 的值。删除回调用于释放与命令相关的对象之类的目的。

```
static int
ObjectCmd(ClientData clientData,
          Tcl_Interp *interp, int objc,
          Tcl_Obj *CONST objv[]) {
    Counter *counterPtr = (Counter *)clientdata;
    char *subcmd;
    if (objc != 2) {
        Tcl_WrongNumArgs(interp, 1, objv,
                         "get | next");
            return TCL_ERROR;
        }
        subcmd = Tcl_GetString(objv[1]);
        if (strcmp(subcmd, "get") == 0) {
            Tcl_SetObjResult(interp,
                Tcl_NewIntObj(counterPtr->value));
        } else if (strcmp(subcmd, "next") == 0) {
            counterPtr->value ++;
        } else {
            Tcl_Obj *result = Tcl_NewStringObj ("", 0);
            Tcl_AppendStringsToObj(result,
                "bad counter command\"",
                Tcl_GetString(objv[1]),
                "\": should be get or next",
                (char *)NULL);
            Tcl_SetObjResult(interp, result);
            return TCL_ERROR;
        }
        return TCL_OK;
    }

    static int
```

```
CounterCmd(ClientData clientData, Tcl_Interp *interp,
           int objc, Tcl_Obj *CONST objv[]) {
    Counter *counterPtr;
    Tcl_Obj *countername;
    static int id = 0;
    if (objc != 1) {
        Tcl_WrongNumArgs(interp, 1, objv, NULL);
        return TCL_ERROR;
    }

    counterPtr = (Counter *)Tcl_Alloc(sizeof(Counter));
    counterPtr->value = 0;
    countername = Tcl_NewStringObj("ctr", -1);
    Tcl_AppendObjToObj(countername,
                       Tcl_NewIntObj(id));
    id ++;
    Tcl_CreateObjCommand(interp,
                       Tcl_GetString(countername),
                       ObjectCmd,
                       (ClientData) counterPtr,
                       DeleteCounter);
    Tcl_SetObjResult(interp, countername);
    return TCL_OK;
}
```

这些函数演示了如何应用 clientData 和 deleteProc 来实现计数对象。CounterCmd 创建新的计数器，这些计数器由 ObjectCmd 代码实现。包含这些代码的应用程序必须首先将 CounterCmd 注册为 Tcl 命令，使用如下调用注册。

```
Tcl_CreateObjCommand(interp, "counter", CounterCmd,,
                     (ClientData) NULL,
                     (Tcl_CmdDeleteProc *) NULL);
```

然后就可以调用 Tcl 命令 counter 来创建新的计数器。每次调用都创建一个新的对象并且返回该对象的名称。

```
    counter
⇒   ctr0
    counter
⇒   ctr1
```

CounterCmd 是 counter 的命令函数。它为新的计数器分配一个结构，并初始化其值为 0。然后它使用静态变量 id 为计数器创建一个名称，把这个名称设为命令的返回结果，增加 id 使得下一个新的计数器能得到一个不同的名称。

这个示例使用了 30.3 节所讲述的“面向对象”的模式，这里每一个计数器对象都有一个命令。作为创建新计数器的一部分，CounterCmd 创建一个新的 Tcl 命令，其名称与计数器名称相同。它使用 Counter 结构体的地址作为命令的 clientData，将 DeleteCounter 指定为新命令的删除回调。

计数器可以通过调用与它们同名的命令来操作。每一个计数器都支持两个命令选项：get，返回计数器的当前值；next，增加计数器的值。一旦创建了 ctr0 和 ctr1，就像前面展示的那样，就可以调用下面的 Tcl 命令：

```
ctr0 next; ctr0 next; ctr0 get
⇒ 2
```

```
    ctr1 get
⇒   0
    ctr0 clear
⇒   bad counter command "clear": should be get or next
```

函数 ObjectCmd 为所有存在的计数器实现了 Tcl 命令。每个计数器会传入一个不同的 clientData 参数，它会重新转换回 Counter *类型的值。ObjectCmd 检查 objv[1]以查看调用命令的选项。如果是 get，就把计数器的值作为整型对象返回；如果是 next，就增加计数器的值，不设置解释器的结果。如果调用的是未知的命令，由 ObjectCmd 调用 Tcl_AppendStringsToObj 与 Tcl_SetObjResult，创建一条错误消息。

要从 Tcl 中销毁计数器，可以直接删除它们的 Tcl 命令，例如：

```
rename ctr0 {}
```

作为删除命令操作的一部分，Tcl 调用 DeleteProc，释放与计数器相关的内存。

这种面向对象的计数器对象实现方式，与 Tk 的组件的实现方式相似：有一个 Tcl 命令用于创建一个计数器或组件的新实例，每一个存在的计数器或组件都对应着一个 Tcl 命令。一个命令函数实现了所有的计数器，或特定类型的组件命令；都获取一个指明特定的计数器或组件的 clientData 参数。删除 Tk 组件与删除计数器的不同机制在前面已经展示了，但在两种情况下，对象被删除的同时与之对应的命令就也删除了。

35.8 删除命令

Tcl 命令可以通过调用 Tcl_DeleteCommand 从解释器中移除(无论是基于对象的命令还是早期的基于字符串的命令，这个函数都是相同的)。例如，下面这条语句删除 ctr0 命令，与上面的 rename 命令的方法是相同的。

```
Tcl_DeleteCommand(interp, "ctr0");
```

如果命令设置有删除回调，那么回调会在命令被移除之前调用。任何命令都可以被删除，包括内建命令、应用程序特有的命令以及 Tcl 过程。

在计数器的实现中，可以在 ObjectCmd 中提供一个“删除”方法，用来删除命令，例如：

```
Tcl_DeleteCommand(interp, Tcl_GetString(objv[0]));
```

也可以使用由 Tcl_CreateObjCommand 返回的命令标记来删除命令。

```
Tcl_DeleteCommandFromToken(Tcl_Interp *interp,
                           Tcl_Command token);
```

即使命令已经被重命名，这条命令仍然能将其删除。

35.9 获取与设置命令参数

Tcl 库提供了一些用于获取与存储关于命令的信息的函数。Tcl_GetCommandInfo 和

Tcl_SetCommandInfo 可以对 Tcl_CmdInfo 结构体调用，该结构体拥有如下的域：

```
typedef struct Tcl_CmdInfo {
        int isNativeObjectProc;
        Tcl_ObjCmdProc *objProc;
        ClientData objClientData;
        Tcl_CmdProc *proc;
        ClientData clientData;
        Tcl_CmdDeleteProc *deleteProc;
        ClientData deleteData;
        Tcl_Namespace *namespacePtr;
    } Tcl_CmdInfo;
```

这些域的含义如下所述。

- isNativeObjectProc
  布尔型值，如果命令是由“对象 API” Tcl_CreateObjCommand 创建的，则为 1；如果命令不是基于对象的，而由“字符串 API” Tcl_CreateCommand 创建的，则为 0。如果这个域为真，那么调用 objProc 而非 proc 可能会有更高的速度。
- objProc
  一个指针，指向实现命令的函数。
- objClientData
  客户数据，用于命令的对象版 API。
- proc
  如果命令是由 Tcl_CreateCommand 创建的，指向的是实现命令的函数。如果命令是由 Tcl_CreateObjCommand 创建的，这就只是用来维持与早期的字符串 API 实现的兼容性的函数。
- clientData
  客户数据，用于命令的字符串版 API。
- deleteProc
  对于对象及字符串生成的命令，这个函数都是相同的，它指向删除命令时要调用的函数。
- deleteData
  传递给 deleteProc 的 clientData。
- namespacePtr
  指向命令所在的命名空间的指针。

下面的命令函数使用 Tcl_GetCommandInfo 获取一个命令的信息，然后创建了新的命令，从而克隆了这个命令。

```
static int
CloneCmd(ClientData clientData, Tcl_Interp *interp,
        int objc, Tcl_Obj *CONST objv[]) {
    char *oldCmdName;
    char *newCmdName;
    Tcl_CmdInfo cmdinfo;
    if (objc != 3) {
        Tcl_WrongNumArgs(interp, 1, objv,
                         "oldCmdName newCmdName");
```

```
        return TCL_ERROR;
    }
    oldCmdName = Tcl_GetString(objv[1]);
    newCmdName = Tcl_GetString(objv[2]);
    if (Tcl_GetCommandInfo(
        interp, oldCmdName, &cmdinfo) == 0) {
        Tcl_AppendResult(interp,
                         "Command not found: ",
                         oldCmdName, NULL);
        return TCL_ERROR;
    }

    Tcl_CreateObjCommand(interp, newCmdName,
                         cmdinfo.objProc,
                         cmdinfo.objClientData,
                         cmdinfo.deleteProc);
    return TCL_OK;
    }
```

这段代码中，oldCmdName 是要克隆的命令，如 expr。如果没有给出一个有效的名称，函数会产生错误，Tcl_GetCommandInfo 会返回 0。newCmdName 是要创建的新命令的名称，该命令的功能与待克隆的命令完全一样。

将 Tcl_CreateObjCommand 用如下代码替换，也可以得到相同的结果。

```
Tcl_CreateObjCommand(interp, newCmdName,
                     (Tcl_ObjCmdProc *)NULL,
                     (ClientData)NULL,
                     (Tcl_CmdDeleteProc *)NULL);
Tcl_SetCommandInfo(interp, newCmdName, &cmdinfo);
```

上述代码创建了一个“空”的命令，然后让它使用 cmdinfo 结构体中的数据。

除了根据名称查找命令，还可以根据标记(“令牌”)来查找命令。获得一个命令的标记有两种方法。如果希望确保总是拥有给定命令的标记，那么应该保存 Tcl_CreateObjCommand 返回的标记。这个标记总是指向该命令，即使它被重命名也一样。您还可以使用 Tcl_GetCommandFromObj 函数根据命令名称获得相应的标记。作为示例，下列代码为 puts 命令获得了该信息。

```
Tcl_Command token;
token = Tcl_GetCommandFromObj(
        interp, Tcl_NewStringObj("puts", -1));
```

这里的命令使用 Tcl_Command 标记获取并设置命令参数，与 35.1 节中讲述的命令是相同的。

35.10 Tcl 过程如何工作

所有的 Tcl 命令，包括那些 Tcl 内核中的命令，如 set、if 以及 proc 等，都使用了本章讲述的机制。

提示： 事实上，最常用的命令是被字节码编译过的，但本书不包括那些内部优化方面的内容。

例如，考虑创建并运行一个 Tcl 过程：

```
proc inc {x} { expr { $x + 1} }
inc 23
⇒ 24
```

在处理 proc 命令时，Tcl 调用了它的命令函数。proc 命令函数是 Tcl 库的一部分，它的参数和结果与 35.2 节中所讲述的相同。proc 命令函数分配一个新的数据结构来描述 inc，包括与 inc 的参数和本体相关的信息。然后 proc 命令函数调用 Tcl_CreateObjCommand，为 inc 创建了 Tcl 命令。它指定了一个名为 TclObjInterpProc 的函数(这是 Tcl 库内部的内容)作为 inc 的命令函数，然后它把新的数据结构的地址用作该命令的 clientData。然后 proc 命令函数返回。

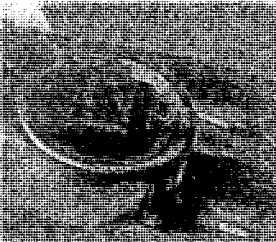
在处理 inc 过程时，Tcl 调用了 TclObjInterpProc，作为该命令的命令函数。TclObjInterpProc 使用它的 clientData 参数，为 inc 访问数据结构，然后编译处理所需要的过程块。然后，TclCompEvalObj(另一个 Tcl 库内部的函数)处理 inc 的过程块。在调用 TclCompEvalObj 之前，TclObjInterpProc 为过程创建了一个新的变量域，使用它的 objv 参数取得 inc 命令的第一个参数，将这个值赋给新的变量域中的 x 变量。当 TclCompEvalObj 调用完成时，TclObjInterpProc 会销毁该过程的变量域，返回来自 TclCompEvalObj 的完成码。所有的 Tcl 过程都把 TclObjInterpProc 作为它们的命令函数。然而，每一个 Tcl 过程有着不同的 clientData，它指向描述了该过程的独一无二的结构体。

35.11 命令跟踪

除了能够跟踪对变量的访问之外(如 34.6 节所述)，在 Tcl 中还可以跟踪命令的执行。跟踪命令的基本方法就是使用 Tcl_CreateObjTrace 函数，它能创建一个有如下特征的函数跟踪：

```
typedef int Tcl_CmdObjTraceProc(
    ClientData clientData,
    Tcl_Interp* interp,
    int level,
    CONST char* command,
    Tcl_Command commandToken,
    int objc,
    Tcl_Obj *CONST objv[] );
```

这可用于创建低层的 Tcl 调试器，因为在程序中的每一步您都可以检查将要执行的命令，程序的状态，等等。这是一个非常强大的功能，不过要注意使用它就意味着关闭 Tcl 的字节码编译器，这会显著降低 Tcl 代码的运行速度。



第36章 扩 展 包

第 35 章介绍了将 Tcl 嵌入已经存在的应用程序，与用户定义的命令扩展 Tcl 的方法。一般的开发者发现，使用 Tcl 编写的应用程序与一些 C 语言的扩展，比编写一个 C 语言的程序然后从程序中调用 Tcl 脚本要更容易、更快速。与 Tcl 编程相比，编写扩展包只需编写很少的 C 程序，可以更好地利用程序员的时间。

本章阐释如何用 C 语言编写 Tcl 的扩展包。

36.1 本章出现的函数

- ```
  CONST char *Tcl_PkgPresent(Tcl_Interp *interp,
          CONST char *name, CONST char *version, int exact)
  CONST char *Tcl_PkgRequire(Tcl_Interp *interp,
          CONST char *name, CONST char *version, int exact)
  ```
  分别等价于 **package present** 和 **package require** 命令。返回一个指针，指向表示解释器提供的扩展包的版本的版本字符串(这可能与 *version* 不同)；如果发生任何错误，则返回 NULL，在解释器的结果中遗留一个错误消息。
- ```
  int Tcl_PkgProvide(Tcl_Interp *interp,
          CONST char *name, CONST char *version)
  ```
  等价于 **package provide** 命令。如果成功完成则返回 TCL_OK；如果出现错误，则返回 NULL，在解释器的结果中遗留一个错误消息。
- ```
  CONST char *Tcl_PkgRequireEx(Tcl_Interp *interp,
          CONST char *name, CONST char *version, int exact,
          ClientData *clientDataPtr)
  CONST char *Tcl_PkgPresentEx(Tcl_Interp *interp,
          CONST char *name, CONST char *version, int exact,
          ClientData *clientDataPtr)
  int Tcl_PkgProvideEx(Tcl_Interp *interp,
          CONST char *name, CONST char *version,
          ClientData clientData)
  ```
  这些调用与它们前面相应的版本相同，只不过允许设置及获取与扩展包相关的客户数据。
  ```

- CONST char *Tcl_InitStubs(Tcl_Interp *interp
  CONST char *version, int exact)
  在与 Tcl 占位符一起编译的扩展包中，这个函数必须是扩展包的第一个 Tcl 库调用。它会尝试初始化占位符表格的指针。

36.2 Init 函数

实际应用中，扩展包就是在 Tcl 运行时用 load 命令加载进 Tcl 的共享库，它通常会被包装进一些 Tcl 代码中，使得我们可以用加载纯 Tcl 扩展包的方法加载 C 扩展包——使用类似 package require *XYZ* 这样的命令。作为共享对象，扩展包在需要它们的系统中根据其文件扩展名识别。在 Windows 中扩展文件以.dll 结尾，在大多数 Unix 系统中以.so 结尾。

Tcl 扩展包通常是以一个或多个新的 Tcl 命令，提供 Tcl 内核未能提供的功能。扩展文件以包的形式编写，就如同它们是 Tcl 脚本包一样。

扩展包的接入点都是它的 Init 函数，形如：

```
int Myextension_Init(Tcl_Interp *interp) {
    ...
}
```

函数名，这里是 Myextension_Init，其格式必须是 *Packagename*_Init。*Packagename* 对应于扩展包的文件名称：一个名为 libPackagename.so 的扩展包，其初始化函数必须是 Packagename_Init；而名为 Foopackage_Init 的初始化函数所在的扩展包的名称必定是 libFoopackage.so。因此，Tcl 只要知道了共享库文件的名称，就能确定 Init 函数，然后调用它。事实上，也可以绕开这个要求，但是最好保持这个命名习惯，这会让事情变得容易一点。

这个 Init 函数在第一次加载扩展包时被调用，如果多次加载扩展包，也只调用一次。这个函数为扩展包进行所有的设置——注册命令、创建变量和初始化数据结构。例如，要创建一个哈希表来跟踪特定种类的对象，就在 Init 函数中创建它。

如果这个函数返回 TCL_OK，表示成功；如果发生错误，就返回 TCL_ERROR。

36.3 包

与第 14 章介绍的纯 Tcl 包类似，基于 Tcl_PkgProvide 函数，编译后的扩展文件可以包的形式提供。

```
int Tcl_PkgProvide(Tcl_interp *interp, char *name,
                   char *version);
```

参数 *interp* 是一个解释器，*name* 是包的名称，*version* 是版本号字符串，例如 1.2 或 5.2.7 等。和使用纯 Tcl 包一样，当脚本调用 package require 命令时就要用到名称和版本号。尽量为包选择独一无二的名字，保证它不会与其他的包冲突。

当然，在复杂的系统中，扩展包可能需要使用别的扩展包。可以用 Tcl_PkgRequire 函

数请求别的扩展包。

```
Tcl_PkgRequire(Tcl_Interp *interp, char *name,
          char *version, int exact);
```

name 是请求的扩展包的名称。*version* 号是满足需要的最低版本号。如果 *exact* 标志不是 0，那么版本号必须精确匹配；否则，任何大于或等于 *version* 的版本都认为可以满足依赖关系。

如果需要关联扩展包中的数据，应该使用一个 ClientData 指针，Tcl 为此提供了 Tcl_PkgProvideEx 和 Tcl_PkgRequireEx 函数。除了要获取一个 ClientData 参数之外，它们与前面讲述的对应函数是相同的。

在创建用来生成并安装您的 Tcl 扩展包的代码时(参见第 47 章)，还需要提供如下 pkgIndex.tcl 文件。

```
package ifneeded counter 0.1 \
    [list load [file join $dir \
        libcounter[info sharedlibextension]]]
```

这就是脚本用 package require counter 请求扩展包时所实际进行的加载操作。自动生成 pkgIndex.tcl 文件的更多细节参见第 14 章。

36.4 命名空间

Tcl 命名空间帮助程序员进行代码的模块化和封装。第 10 章从脚本层级讨论了 Tcl 命名空间。当前版本的 Tcl 没有提供有命名空间功能的 C API，但有一些工作与在C中创建新的命名空间有关。

在一个命名空间中创建命令时，如果该命名空间不存在，则会自动创建命名空间。

```
Tcl_CreateObjCommand(interp, "foo::bar", BarCmd,
        (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);
```

您还可以调用 Tcl_Eval 来快速创建它。

```
char *nsscript = "namespace eval ::foo {}";
Tcl_Eval(interp, nsscript);
```

如果仅仅是要提供一两个命令，那么可能不需要创建一个命名空间，但如果有很多命令，特别是您创建了很多变量的话，就应该考虑使用命名空间，避免别的程序员在全局命名空间中照看太多的命令的麻烦。

36.5 Tcl 占位符

Tcl 提供了一种机制，确保不同版本的 Tcl 的同一扩展函数不必为各个 Tcl 一一编译链接。这极大地增强了不同的扩展包发行版和不同的 Tcl 版本号间的兼容性，它们中一个的版本变化时，另一个不必非得随之变化。这是通过形成自查找的符号表，而不去使用普通的共享库机制来实现的。这种迂回的方式让我们把扩展包和特定的 Tcl 库的版本脱钩。在

扩展包中，由函数 Tcl_InitStubs 调用占位符机制。

```
Tcl_InitStubs(Tcl_Interp *interp,
        char *version, int exact);
```

第一个参数指明了将要加载扩展包的解释器。*version* 字符串参数指定了扩展包需要的最低 Tcl 版本号(主版本号必须与 Tcl 解释器匹配)。例如，8.3 表示扩展包需要 Tcl 8.3 版或以后的版本，而 8 表示任何第 8 版的 Tcl 解释器都是适用的。如果希望使用指定版本的 Tcl 库，就把 exact 标志设置为 1，要求占位符与一个并且只与一个 Tcl 版本匹配——不过这种情况下更合情理的做法是直接链接而非使用占位符机制。

使用占位符的 Tcl 扩展包必须首先调用 Tcl_InitStubs；要访问 Tcl 库中的其他函数，必须首先加载占位符。

为了编写健壮的代码，通常最好使用#ifdef 来隔离 Tcl_InitStubs 的调用，以便在某些情况下可以不使用占位符机制来编译扩展包。

```
#ifdef USE_TCL_STUBS
    if (Tcl_InitStubs(interp, "8", 0) == NULL) {
        return TCL_ERROR;
    }
#endif
```

这样，是否使用占位符就由一个编译标志来确定了，第 47 章将就此进行更深入的讨论。

36.6 ifconfig 扩展包

下面的源代码实现了一个名为 ifconfig 的扩展包，用于在 Linux 系统中查询网络接口的状态。

```
/*
    Copyright 2004, David N. Welton <davidw@dedasys.com>

    This code may be used, modified and redistributed under the
    same terms as Tcl itself.
*/

#include <stdlib.h>
#include <strings.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <tcl.h>

#define IFC_BUFLEN 8192

/*
 *----------------------------------------------------------
```

```
 *
 *  ListInterfaces --
 *
 *  List available interfaces.
 *
 *  Results:
 *  A list of interfaces.
 *
 *  Side Effects:
 *  None.
 *
 *----------------------------------------------------------
 */

static int ListInterfaces(Tcl_Interp *interp) {
    struct ifconf if_request;
    char *ifc_buffer;
    int i, j;
    int fd;
    Tcl_Obj *iflist;
    Tcl_Obj *ifname = NULL;

    /* Allocate space for the interface request. */
    ifc_buffer = (char *)malloc(IFC_BUFLEN);

    bzero(ifc_buffer, IFC_BUFLEN);
    bzero(&if_request, sizeof(if_request));
    /* We need a socket file descriptor to work with, even if it
     * isn't connected to anything. */
    fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);

    if_request.ifc_buf = ifc_buffer;
    if_request.ifc_len = IFC_BUFLEN;

    /* Fetch information from the kernel. */
    ioctl(fd, SIOCGIFCONF, &if_request);

    iflist = Tcl_NewObj();
    /* Loop through interfaces. */
    for (i = 0, j = 0, i < if_request.ifc_len;
        i += sizeof(struct ifreq), j++) {
        /* Fetch interface name, create a new string object. */
        ifname = Tcl_NewStringObj(if_request.ifc_req[j].ifr_name,
                                  -1);

        /* Add it to the list. */
        Tcl_ListObjAppendElement(interp, iflist, ifname);
    }
    /* Use the list as the result. */
    Tcl_SetObjResult(interp, iflist);

    /* Clean up other resources. */
    close(fd);
    free(ifc_buffer);

    return TCL_OK;
}

/*
 *----------------------------------------------------------------------
```

```
 *
 * GetInterface --
 *
 * Fetch information about a particular interface in the form
 * of a key/value list suitable for use as an array or dic.
 *
 * Results:
 * A list of keys and their values.
 *
 * Side Effects:
 * None
 *
 *----------------------------------------------------------------------
 */

static int GetInterface(Tcl_Interp *interp,
            int objc,
            Tcl_Obj *CONST objv[]) {
    int fd;

    struct ifreq ifreq;
    unsigned char *hw;
    struct sockaddr_in *sin;
    char *itf;
    char hwaddr[40];
    Tcl_Obj *addr;
    Tcl_Obj *braddr;
    Tcl_Obj *netmask;
    Tcl_Obj *result;

    if (objc != 3) {
        Tcl_WrongNumArgs(interp, 2, objv, "interface");
        return TCL_ERROR;
    }
    itf = Tcl_GetString(objv[2]);

    fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);

    strcpy(ifreq.ifr_name, itf);

    /* hardware address */
    ioctl(fd, SIOCGIFHWADDR, &ifreq);
    hw = ifreq.ifr_hwaddr.sa_data;
    sprintf(hwaddr, "%02x:%02x:%02x:%02x:%02x:%02x",
            *hw, *(hw + 1), *(hw + 2), *(hw +3),
            *(hw + 4), *(hw + 5));

    result = Tcl_NewObj();
    Tcl_ListObjAppendElement(interp, result,
                Tcl_NewStringObj("hwaddr", -1));
    Tcl_ListObjAppendElement(interp, result,
                Tcl_NewStringObj(hwaddr, -1));

    /* address */
    if (ioctl(fd, SIOCGIFADDR, *ifreq) == 0) {
        sin = (struct sockaddr_in *)&ifreq.ifr_broadaddr;
        addr = Tcl_NewStringObj(inet_ntoa(sin->sin_addr), -1);

        /* broadcast */
        ioctl(fd, SIOCGIFBRDADDR, &ifreq);
        sin = (struct sockaddr_in *)&ifreq.ifr_broadaddr;
```

```
            braddr = Tcl_NewStringObj(inet_ntoa(sin->sin_addr), -1);

            /* netmask */
            ioctl(fd, SIOCGIFNETMASK, &ifreq);
            sin = (struct sockaddr_in *)&ifreq.ifr_broadaddr;
            netmask = Tcl_NewStringObj(inet_ntoa(sin->sin_addr), -1);
        } else {
            addr = Tcl_NewObj();
            braddr = Tcl_NewObj();
            netmask = Tcl_NewObj();
        }

        Tcl_ListObjAppendElement(interp, result,
                    Tcl_NewStringObj("addr", -1));
        Tcl_ListObjAppendElement(interp, result, addr);
        Tcl_ListObjAppendElement(interp, result,
                    Tcl_NewStringObj("braddr", -1));
        Tcl_ListObjAppendElement(interp, result, braddr);
        Tcl_ListObjAppendElement(interp, result,
                    Tcl_NewStringObj("netmask", -1));
        Tcl_ListObjAppendElement(interp, result, netmask);

        /* flag */
        ioctl(fd, SIOCGIFFLAGS, &ifreq);

        Tcl_ListObjAppendElement(interp, result,
                    Tcl_NewStringObj("up", -1));
        Tcl_ListObjAppendElement(interp, result,
                    Tcl_NewBooleanObj(
                    (ifreq.ifr_flags & IFF_UP)));

        Tcl_ListObjAppendElement(interp, result,
                    Tcl_NewStringObj("running", -1));
        Tcl_ListObjAppendElement(interp, result,
                    Tcl_NewBooleanObj(
                    (ifreq.ifr_flags & IFF_RUNNING)));

        Tcl_ListObjAppendElement(interp, result,
                    Tcl_NewStringObj("noarp", -1));
        Tcl_ListObjAppendElement(interp, result,
                    Tcl_NewBooleanObj(
                    (ifreq.ifr_flags & IFF_NOARP)));

        close(fd);

        Tcl_SetObjResult(interp, result);
        return TCL_OK;
}

/*
 *----------------------------------------------------------------------
 *
 * Ifconfig --
 *
 * The main function, which calls secondary functions depending
 * on the "subcommand".
 *
 * Results:
 * Depends on the function requested.
 *
 * Side Effects:
```

```
 * None.
 *
 *----------------------------------------------------------------------
 */

static int Ifconfig(ClientData clientdata,
            Tcl_Interp *interp,
            int objc,
            Tcl_Obj *CONST objv[]) {
    char *subcmd;
    if (objc < 2) {
        Tcl_WrongNumArgs(interp, 1, objv, "list | get | set");
        return TCL_ERROR;
    }

    subcmd = Tcl_GetString(objv[1]);

    if (strcmp(subcmd, "list") == 0) {
        return ListInterfaces(interp);
    } else if(strcmp(subcmd, "get") ==0) {
        return GetInterface(interp, objc,objv);
    } else {
        return TCL_ERROR;
    }
    return TCL_OK;
}

/*
 *----------------------------------------------------------------
 *
 * Tclifconfig_Init - -
 *
 * Initializes the package, creating the ifconfig command.
 *
 * Results:
 * None.
 *
 * Side Effects:
 * Creates ifconfig command and package.
 *
 *----------------------------------------------------------------
 */

int Tclifconfig_Init(Tcl_Interp *interp) {

#ifdef USE_TCL_STUBS
    if (Tcl_InitStubs(interp, "8", 0) == NULL) {
        return TCL_ERROR;
    }
#endif

    /* Create the command. */
    Tcl_CreateObjCommand(interp, "ifconfig", Ifconfig,
            (ClientData) NULL, (Tcl_CmdDeleteProc *)NULL);

    if ( Tcl_PkgProvide(interp,
            "ifconfig",
            "0.1") != TCL_OK) {
        return TCL_ERROR;
    }
    return TCL_OK;
}
```

考虑这个 Init 函数。在 Linux 中编译的库的名称是 libtclifconfig.so，所以 Tclifconfig_Init 函数名中的 Tclifconfig 对应 tclifconfig。当 Tcl 加载这个库时，它会根据共享库的名称查找这个函数。这时，依照我们在编译时对标志 USE_TCL_STUBS 的设置，确定是否加载占位符表。

下一件任务就是创建 ifconfig 命令。我们使用 Tcl_CreateObjCommand 把它和 Ifconfig 函数关联起来，这也在 Tcl 脚本中注册了 ifconfig 命令。

接下来调用 Tcl_PkgProvide 注册 ifconfig 扩展包——以免因为这个命令感到迷惑——设定其版本号为 1.0。

Init 函数正常完成时必须返回 TCL_OK，否则说明扩展包加载失败。

在编译了扩展包之后，我们可以使用新的 ifconfig 命令在 Linux 系统中搜集关于网络接口的信息。

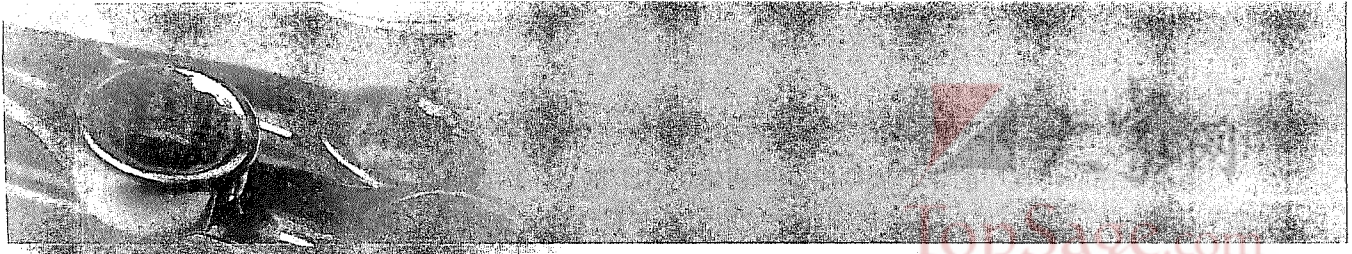
```
load ./libtclifconfig.so
ifconfig list
⇒ lo eth0 eth1
ifconfig get eth1
⇒ hwaddr 00:02:2d:1f:41:46 addr 10.0.0.10 braddr 10.255.255.255
netmask 255.255.255.0 up 1 running 64
noarp 0
array set eth1 [ifconfig get eth1]
set eth1(hwaddr)
⇒ 00:02:2d:1f:41:46
```

算上所有的内容，包括源码、注释，这里一共只有不到 300 行C代码，就获得了我们在 Tcl 中可以很容易使用的数据。

作为对比，考虑另一种完成同样任务的方法：解析由/sbin/ifconfig 命令输出的结果。那样的话，您必须完全确定其输出格式不会随着版本更新而变化，也不会因平台的不同而变化。如果会发生变化，脚本就可能是在处理错误的信息。我们的 Tcl/C 扩展包靠自己就具备一定的跨平台的能力(ifdef)，而且它运行速度快，不涉及解析操作——获得的数据立即就可以用于其他的 Tcl 命令。

要把这些代码打包发布，可以如下创建 pkgIndex.tcl 文件。

```
if {[catch {package present ifconfig 1.0}]} { return }
    package ifneeded ifconfig 1.0 [list load \
        [file join /usr/lib libifconfig1.0.so.0] Tk]
```



第 37 章　嵌入 Tcl

在用 C 语言编写的应用程序中使用 Tcl 的行为称为嵌入。本章介绍如何把 Tcl 加入应用程序，以及如何创建一个新的 tclsh 风格的应用程序。

37.1　本章出现的函数

- `void Tcl_FindExecutable(argv[0])`
  计算可执行文件的路径，Tcl 的一些内部机制需要该路径。
- `CONST char *Tcl_GetNameOfExecutable()`
  返回应用程序的完整路径名称。
- `int Tcl_Init(Tcl_Interp *interp)`
  完成 Tcl 的很多初始化工作，例如应用 Tcl 自己的 init.tcl。
- `int Tcl_AppInit(Tcl_Interp *interp)`
  在新建 tclsh 风格的程序时用户提供的挂接过程。

37.2　将 Tcl 添加到应用程序

如果要从零开始创建一个新的应用程序，也许应该考虑直接用 Tcl 编写，不过出于各种原因可能无法直接使用 Tcl。在已经存在的应用程序中嵌入 tcl 是很容易的——只需要很少一点代码，给应用程序提供 Tcl 解释器后就能让应用程序的功能大大增强。我们前面看到的扩展模式，大多数代码是由 Tcl 编写的，在必需 C 语言来获得更多的功能、更快的速度时使用 C 代码。在“嵌入”模式中，Tcl 只是链接到主应用程序的一个库，提供相应的服务，这里就是对 Tcl 脚本的处理。

例如，可以把 Tcl 加到网页服务器中，允许用户用 Tcl 创建动态网页。这就是 AQL 服务器和众多 Apache Tcl 项目采用的方法。开源的 PostgreSQL 数据库允许使用 Tcl 编写函数和触发过程。让用户可以用 Tcl 编写新的有意思的脚本的大型复杂程序不胜枚举。如果一个程序拥有强大的脚本界面，那么它的价值将大大提升。用户可以创建和分享代码——甚至还会发现极有新意的方法，把过程通过 Tcl 挂接到别的系统中。

第 31 章包含了处理文件中的 Tcl 脚本的非常简单的应用。嵌入 Tcl 需要的工作并不比那多多少。

您需要在应用程序的生命周期中选择初始化 Tcl 的点。例如，您可能希望将 Tcl 用作配置语言——可以启动 Tcl，然后用它来读取保存了系统设置选项的文件，例如：

```
set port 80
set hostname "www.tcl.tk"
set errorlog /var/log/myserver/error.log
```

或者，您可以创建一些命令，不必把这些配置选项作为变量对待。

```
port 80
hostname "www.tcl.tk"
errorlog /var/log/server/error.log
```

在调用这个命令时它会设置应用程序内部的变量。如果已经有了一个配置系统，您可能会希望延迟 Tcl 的启动，直到您可以建立 Tcl 环境来反映它运行所在系统的信息；然后您可以设置一些 Tcl 变量，根据系统的设置情况进行不同的操作。

37.3 初始化 Tcl

要处理 Tcl 代码，需要调用如下设置函数：

```
...
Tcl_Interp *interp;
Tcl_FindExecutable(argv[0]);
interp = Tcl_CreateInterp();

if (interp == NULL) {
    fprintf(stderr, "Tcl Interp creation failed, exiting\n");
    exit(1);
}

if (Tcl_Init(interp) == TCL_ERROR) {
    fprintf(stderr, "Tcl Init failed, exiting\n");
    exit(1);
}
...
```

在调用其他 Tcl 库函数之前，应该调用 Tcl_FindExecutable，传入 argv[0]作为参数，帮助 Tcl 完成对自身的初始化。

然后可以用 Tcl_CreateInterp 创建解释器，如第 31 章所述。如果在创建解释器这一步，或是下一步调用 Tcl_Init 时，有一个失败，系统可能就存在问题，可能应该放弃这次应用程序的运行，因为将无法在其中使用 Tcl。Tcl_Init 函数查找并处理 init.tcl 脚本，这个脚本作为 Tcl 的一部分发布。这些函数中的任意一个失败，就意味着 Tcl 没有安装正确。

现在 Tcl 库就准备好处理 Tcl 脚本了。在 C 代码的同样位置，可能还应该注意您所需要的其他初始化工作，例如创建命令、变量以及通道等。如果使用多个解释器，需要首先初始化它们，然后在各个解释器中分别完成设置任务，因为现在不能复制解释器状态(命令、变量等)。

在确定了应用程序中进行 Tcl 初始化的位置之后，需要做的第二个决定是什么时候处理 Tcl 脚本。对某些应用程序而言，什么时候需要处理是显而易见的。对使用 Tcl 动态生

成页面的网页服务器，Tcl 脚本在用户请求某些资源(通常是网页)时处理。如果用户发出请求，例如 index.tcl，网页服务器的响应就是处理相应的 Tcl 代码，把 HTML 结果发送给用户。其他的应用程序可能需要更加复杂的钩子，例如 Tcl 的事件循环(参见第 43 章)。不过，某些时候应用程序可能应该调用 Tcl_Eval 系列的命令处理 Tcl 代码。

37.4 创建新的 Tcl 外壳

在 Tcl 可以加载共享库扩展包之前，增强语言功能的流行做法是创建一个新版本的 Tcl 外壳(tclsh)，在其中增加所需要的功能。这是一种特别的“嵌入”，这里 C“应用程序”唯一的功能就是初始化设置然后处理 Tcl 代码。这种方法的缺点是当您需要不止一个扩展时，需要把它们整合到一起编译，而加载扩展包则无需如此。动态加载扩展包是适应性更好的方法。

有时候可能用到旧的代码，或者您希望得到没有加载库的静态链接的可执行文件。那种情况下，创建自己的 Tcl 脚本可能是最好的方法，尽管 Starkit(参见第 14 章)这样的解决方案可能更受欢迎。

您还应该注意到 Tcl 库和 tclsh 程序的不同。tclsh 提供了一些 Tcl 库没有设置的变量，包括 argc、argv、argv0 以及 tcl_interactive，第 3 章讨论了这些参数。另外，tclsh 读取并执行一个“点文件”，.tclshrc。如果希望创建自己的外壳，建议复制这个行为，使得外壳的行为与 tclsh 相类似。

为了完成这个任务，Tcl 提供了 Tcl_AppInit 函数。Tcl_AppInit 函数完成应用程序特定的初始化，例如创建新的命令。tclsh 和 wish 的 main 函数在它们自身的初始化完成后调用 Tcl_AppInit，然后才开始处理 Tcl 脚本。这样做就让 main 函数具备所有 tclsh 类型(或 wish 类型)应用程序所共有的功能，而 Tcl_AppInit 则提供了应用程序独有的功能。

tclsh 使用 Tcl_AppInit 的示例如下：

```
int Tcl_AppInit(Tcl_Interp *interp) {
    if (Tcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    tcl_RcFileName = "~/.tclshrc";
    return TCL_OK;
}
```

调用 Tcl_AppInit 时要提供一个参数，即应用程序使用的 Tcl 解释器。如果 Tcl_AppInit 正常完成，它返回 TCL_OK；如果遇到错误，它返回 TCL_ERROR，并在解释器中设置一个结果(这样应用程序会输出错误消息并退出)。这个示例中的 Tcl_AppInit 做了两件事。首先，它调用函数 Tcl_Init，完成附加的 Tcl 初始化(它从 Tcl 库中读取并执行脚本以定义 unknown 过程，设置自动加载机制，如第 14 章所述)。Tcl_Init 的参数和结果与 Tcl_AppInit 相同。Tcl_AppInit 做的第二件事是设置全局变量 tcl_RcFileName，它给出了用户指定的起始脚本名称。在 tclsh 和 wish 的 main 函数中，这个变量默认设置为 NULL；如果 Tcl_AppInit 修改了 tcl_RcFileName，或者如果应用程序是交互式运行的(而不是从一个脚本文件运行的)，main 会读取并执行由 tcl_RcFileName 指定的文件。

要创建新的 tclsh 类型的壳，需要编写自己的 Tcl_AppInit 函数，把它和 Tcl 库一起编译。例如，可以从 Tcl 源目录中复制 unix/tclAppInit.c，重新生成一个 tclsh 应用程序，这个文件就包含了前面的 Tcl_AppInit 代码。然后编译这个文件，与 Tcl 库链接到一起，如第 46 章所述。

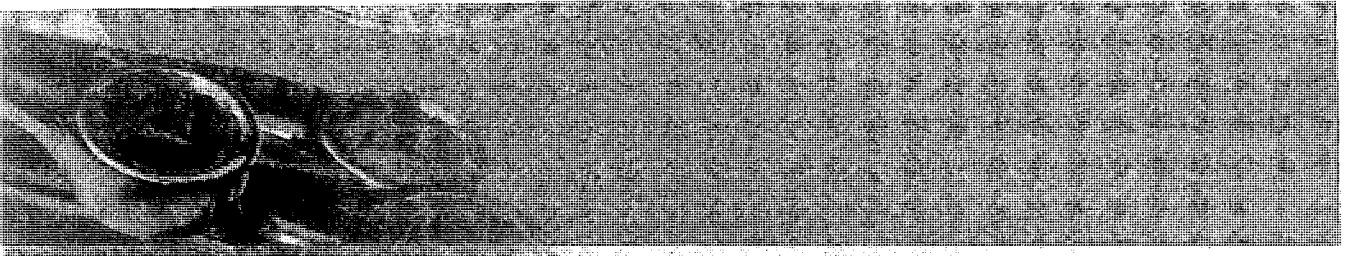
```
cc tclAppInit.c -ltcl -lm -o mytclsh
```

得到的应用程序和系统版的 tclsh 是一样的：它支持交互式输入、脚本文件以及 tclsh 所支持的其他所有功能。

如果希望在应用程序中包含第 35 章所展示的 counter 包，应该编写如下 Tcl_AppInit：

```
int Tcl_AppInit(Tcl_Interp *interp) {
    if (Tcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    if (Counter_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    tcl_RcFileName = "~/.tclshrc";
    return TCL_OK;
}
```

然后，应该把 counter.c 作为应用程序的一部分，和包含前述 Tcl_AppInit 函数的文件一起进行编译。



第38章 异 常

很多Tcl命令，例如if和while，都有由Tcl脚本构成的参数。这些命令将C函数通过结构处理它们以处理脚本。如果Tcl_EvalObjEx或者同与它相关的处理函数返回TCL_OK之外的完成码，就说明发生了异常。异常包括TCL_ERROR(参见第31章)以及其他一些还没有提到过的情况。本章介绍完整的异常集合，讲述如何展开嵌套的处理过程，把有用的信息保存在errorInfo以及errorCode变量中。

38.1 本章出现的函数

- `Tcl_AddObjErrorInfo(Tcl_Interp *interp, char *message, int length)`
  向errorInfo变量中添加错误文本。
- `Tcl_AddErrorInfo(Tcl_Interp *interp, char *message)`
  与Tcl_AddObjErrorInfo相似，不过如果错误结果是新的，将errorInfo初始化为解释器的结果字符串的值。
- `Tcl_SetObjErrorCode(Tcl_Interp *interp, Tcl_Obj *errorObjPtr)`
  根据*errorObjPtr*设置errorCode变量。
- `Tcl_SetErrorCode(Tcl_Interp *interp, char *element, char *element, ... (char *) NULL)`
  根据一系列字符串设置errorCode变量。
- `Tcl_SetErrorCodeVA(Tcl_Interp *interp, va_list argList)`
  根据va_list设置errorCode变量。
- `Tcl_Obj *Tcl_GetReturnOptions(Tcl_Interp *interp, int code)`
  取得在处理脚本之后解释器返回的选项字典。
- `int Tcl_SetReturnOptions(Tcl_Interp *interp, Tcl_Obj *options)`
  将*interp*的返回选项设置为*options*。

- `CONST char *Tcl_PosixError(Tcl_Interp *interp)`
  在 POSIX 内核调用错误之后设置 errorCode。它读取 C 变量 errno 的值，然后调用 Tcl_SetErrorCode 以 POSIX 格式设置 errorCode 的值。返回可阅读的错误消息。
- `void Tcl_LogCommandInfo(Tcl_Interp *interp, CONST char *script, CONST char *command, int commandLength)`
  将产生错误的命令的信息添加到 errorInfo 变量，并设置在解释器中内部存放的行号。
- `void Tcl_Panic(CONST char *format, arg, arg, ...)`
  向该进程的标准错误文件输出带格式的错误消息，然后调用 abort 终止进程。Tcl_panic 不会返回。
- `void Tcl_PanicVA(CONST char *format, va_list argList)`
  与 Tcl_Panic 相似，不过获取 va_list 参数。
- `void Tcl_SetPanicProc(Tcl_PanicProc panicProc)`
  设置调用 Tcl_Panic 时使用的函数。

38.2 完成代码

表 38.1 列出了 Tcl 定义的完成代码集(完成代码均为整数值)。如果命令过程返回了 TCL_OK 以外的任何值，Tcl 都会放弃处理包含该命令的脚本，将完成代码作为 Tcl_EvalObjEx(或 Tcl_EvalFile)的结果返回。TCL_OK 和 TCL_ERROR 已经讨论过了，它们分别用于正常返回和产生错误的情况。如果命令 break 或 continue 被脚本调用，则会遇到 TCL_BREAK 或 TCL_CONTINUE 调用，这两种情况下解释器的结果都是空字符串。如果调用 return，会遇到 TCL_RETURN 完成代码，这种情况下解释器的结果就是所包含的过程中指定的结果。应用程序特有的命令还可以定义更多的完成代码。

表 38.1 由脚本处理函数返回的完成代码

完成代码	含 义
TCL_OK	命令正常完成
TCL_ERROR	遇到不可恢复的错误
TCL_BREAK	调用了 break 命令
TCL_CONTINUE	调用了 continue 命令
TCL_RETURN	调用了 return 命令
其他	由应用程序定义

下面展示如何产生 TCL_BREAK 完成代码，这是为 break 命令准备的命令过程。

```
int Tcl_BreakObjCmd(dummy, interp, objc, objv)
    ClientData dummy;           /* Not used. */
    Tcl_Interp *interp;         /* Current interpreter. */
    int objc;                   /* Number of arguments. */
```

```
    Tcl_Obj *CONST objv[];      /* Argument objects. */
{
    if (objc != 1) {
        Tcl_WrongNumArgs(interp, 1, objv, NULL);
        return TCL_ERROR;
    }
    return TCL_BREAK;
}
```

事实上，在编译好的代码中，情况与此略有不同，仅在没有编译时才会调用这个命令。不过，这是个很好的演示。

TCL_BREAK、TCL_CONTINUE 以及 TCL_RETURN 用于展开循环命令中或过程调用中的嵌套处理的脚本。在大多数情况下，从 Tcl_EvalObjEx 中接收到 TCL_OK 之外的任何完成代码的过程，应该将相同的完成代码立即返回给调用者，并且不改动解释器的结果。不过，有不多的一些命令过程，在遇到某些特定的完成代码时不返回它们。例如，以下是 while 命令的一个实现。

```
int Tcl_WhileObjCmd(dummy, interp, objc, objv)
        ClientData dummy;           /* Not used. */
        Tcl_Interp *interp;         /* Current interpreter. */
        int objc;                   /* Number of arguments. */
        Tcl_Obj *CONST objv[];      /* Argument objects. */
{
    int result, value;
    if (objc != 3) {
        Tcl_WrongNumArgs(interp, 1, objv,
                "test command");
        return TCL_ERROR;
    }
    while (1) {
        /* Get the results of the expression which
            indicates whether to continue or not. */
        result = Tcl_ExprBooleanObj(interp, objv[1],
                                    &value);
        if (result != TCL_OK)
            return result;
        }
        if (value == 0) {
            return TCL_OK;
        }
        /* Evaluate the while body. */
        result = Tcl_EvalObjEx(interp, objv[2], 0);
        if (result == TCL_CONTINUE) {
            continue;
        } else if (code == TCL_BREAK) {
            return TCL_OK;
        } else if (code != TCL_OK) {
            return code;
        }
    }
}
```

在检查它的参数个数之后，Tcl_WhileObjCmd 进入了循环，每次迭代时都把命令的第一个参数作为表达式，第二个参数作为脚本处理。如果在处理表达式时发生了错误，Tcl_WhileObjCmd 就返回该错误。如果表达式处理成功，其值为 0，则命令正常返回并结束，否则就处理脚本参数。如果完成代码是 TCL_CONTINUE，Tcl_WhileObjCmd 就进入

下一次循环迭代。如果代码是 TCL_BREAK，Tcl_WhileObjCmd 结束命令的执行，将 TCL_OK 返回给它的调用者。如果 Tcl_EvalObjEx 返回了 TCL_OK 以外的其他完成代码，Tcl_WhileObjCmd 就简单地将这个代码反映给上一层。这使得遇到 TCL_ERROR 或 TCL_RETURN 时可以正确地完成展开，而且在以后添加了新的完成代码时也能正确展开。

如果异常返回一直展开到最顶层的脚本，Tcl 会检查这个完成代码，确认它是 TCL_OK 或 TCL_ERROR。如果返回的是其他代码，Tcl 会把返回代码转变为错误代码，提供适当的错误消息。例如，如果 TCL_BREAK 或 TCL_CONTINUE 异常引起了对整个过程的展开，Tcl 会将其转换为错误。例如：

```
    break
⇒ invoked "break" outside of a loop
    proc badbreak {} {break}
    badbreak
⇒ invoked "break" outside of a loop
```

因此，应用程序在处理最顶层的脚本时，不需考虑 TCL_OK 和 TCL_ERROR 之外的完成代码。使用 Tcl_AllowException，函数可以改变这一行为，详见参考文档。

38.3 设置 errorCode

在错误发生后设置的最后一条信息是 errorCode 变量，它提供 Tcl 脚本易于处理的错误消息——errorInfo(详见后文)是设计为让人阅读的，而不是易于程序使用的。errorCode 设计为在脚本可能会捕获错误时使用，用来确定是什么东西错了，如果可能，会尝试进行恢复。如果一个命令过程向 Tcl 返回了错误而没有设置 errorCode，Tcl 默认将其设置为 NONE。如果一个命令过程希望在 errorCode 中提供信息，它应该在返回 TCL_ERROR 之前调用 Tcl_SetErrorCode 或 Tcl_SetObjErrorCode。

Tcl_SetErrorCode 接受的参数包括获得一个解释器、任意个字符串参数以及一个空指针。它将字符串组成列表，将该列表存储为 errorCode 的值。假设您已经编写了一些命令实现一个 GIZMO 对象，在操作这个对象时命令可能会遇到各种错误，例如试图使用不存在的对象等。如果某个命令过程探测到对象不存在的错误，它可以如下设置 errorCode：

```
Tcl_SetErrorCode(interp, "GIZMO", "EXIST",
                 "no object by that name",
                 (char *) NULL);
```

这会把 GIZMO EXIST {no object by that name}存放在 errorCode 中。GIZMO 说明了错误的大类(与 GIZMO 对象相关)，EXIST 是遇到的特定错误的符号名，最后一个参数是可阅读的错误消息的列表。只要第一个列表元素不和其他已经使用的值冲突，就可以在 errorCode 中保存要存放的任何信息，但通常都是提供 Tcl 脚本易于处理的符号信息。例如，访问 GIZMO 的脚本可以捕获错误，如果错误是不存在该 GIZMO，它可能自动创建一个新的。

对象版的设置函数是 Tcl_SetObjErrorCode。

```
Tcl_SetObjErrorCode(Tcl_Interp *interp, Tcl_Obj
                        *errorListObj);
```

这里 errorListObj 是一个 Tcl 列表，与 Tcl_SetErrorCode 所获取的那个列表格式相近。

```
Tcl_Obj *err;
err = Tcl_NewObj();
Tcl_ListObjAppendElement(
    interp, err, Tcl_NewStringObj("GIZMO", -1));
Tcl_ListObjAppendElement(
    interp, err, Tcl_NewStringObj("EXISTS", -1));
Tcl_ListObjAppendElement(
    interp, err, Tcl_NewStringObj(
                    "no object by that name", -1));
Tcl_SetObjErrorCode(interp, err);
```

这种情况下使用非对象版的设置可能更简单一些，除非您需要在列表中直接插入数字。这种错误出现的机会很少，所以，因使用非对象版代码引起的性能下降可以忽略。

38.4 管理返回的选项字典

第 13 章中曾经提到过，可以用 return 命令提供返回的选项字典，在异常发生时传递任意的信息。然后 catch 命令可以捕获返回的选项字典，对它进行任意的处理。

在 C 代码中完成同样任务的方法是使用 Tcl_SetReturnOptions 设置返回的选项字典，使用 Tcl_GetReturnOptions 来获取它。

Tcl_GetReturnOptions 的一个典型应用是在脚本处理返回错误时获得堆栈跟踪，例如：

```
int code = Tcl_Eval(interp, script);
if (code == Tcl_ERROR) {
    Tcl_Obj *options = Tcl_GetReturnOptions(interp, code);
    Tcl_Obj *key = Tcl_NewStringObj("-errorinfo", -1);
    Tcl_Obj *stackTrace;
    Tcl_IncrRefCount(key);
    Tcl_DictObjGet(NULL, options, key, &stackTrace);
    Tcl_DecrRefCount(key);
    /* Do something with stackTrace. */
}
```

上述代码以字典对象 options 为参数，返回关键字-errorinfo 的值，该值是详细描述 script 处理出现错误时的堆栈跟踪信息。

38.5 在 errorInfo 中添加堆栈跟踪

在遇到错误时，Tcl 会把对错误的描述添加到产生错误的命令的堆栈跟踪内，用户可以用 errorInfo 全局变量取得这个堆栈跟踪。Tcl 通过调用 Tcl_AddObjErrorInfo 或 Tcl_AddErrorInfo 过程来完成这一任务，它们的原型如下：

```
void Tcl_AddObjErrorInfo(Tcl_Interp *interp,
                char *message, int length);
```

```
void Tcl_AddErrorInfo(Tcl_Interp *interp,
                char *message);
```

两者的不同之处是 Tcl_AddObjErrorInfo 中的 *message* 可以包含嵌入的 NULL。在大多数情况下，使用 Tcl_AddErrorInfo 更为简单。

在遇到错误之后对 Tcl_AddErrorInfo 的第一次调用将 errorInfo 设置为解释器中的错误消息，然后添加上 *message*。后面因同样的错误的每一次调用都把 *message* 添加到 errorInfo 当前值的后面。而一个命令过程返回 TCL_ERROR 时，Tcl_Eval 调用 Tcl_AddErrorInfo 对正在执行的命令的信息进行日志记录。如果有对 Tcl_Eval 的嵌套调用，在展开时每一调用都会添加它自己的命令的信息，于是在 errorInfo 中形成了堆栈跟踪。

命令过程自身也可以调用 Tcl_AddErrorInfo，提供有关错误所在环境的更多信息。对于递归地调用 Tcl_Eval 命令的过程，这一功能特别有用。例如，考虑下面这个 Tcl 过程，该过程试图获取列表中最长的元素的长度，不过这个过程是有错误的。

```
proc longest {list} {
    set i [llength $list]
    while {$i >= 0} {
        set length [string length [lindex $list $i]]
        if {$length > $max} {
            set max $length
        }
        incr i -1
    }
    return $max
}
```

这个过程失败的原因是它从没初始化变量 max，因此在 if 命令试图读取它时会出现错误。如果用命令 longest {a 12345 xyz}调用该过程，在出现错误之后 errorInfo 中保存有如下堆栈跟踪信息：

```
can't read "max": no such variable
    while executing
"if {$length > $max} {
    set max $length
        }"
    (procedure "longest" line 5)
    invoked from within
"longest {a 12345 xyz}"
```

除了括号中的注释，这里所有的信息都由 Tcl_Eval 提供。括号中的消息由处理过程块中的代码生成。

敏锐的人会发现这里没有关于 while 命令的错误消息。这是因为它已经完成了编译，而并不作为一个普通过程存在。如果我们可以强制将它作为过程调用，我们会得到的堆栈跟踪信息如下：

```
can't read "max": no such variable
    while executing
"if {$length > $max} {
        set max $length
        }"
        ("while" body line 3)
        invoked from within
"$whilecmd {$i >=0 } {
```

```
        set length [string length [lindex $list $i]]
        if {$length > $max} {
            set max $length
        }
        incr i -1
       }"
    (procedure "longest_with_while" line 4)
    invoked from within
"longest_with_while {a 12345 xyz}"
```

如果不对 while 命令完成编译，我们会得到第二对括号中的消息，它为我们提供了在 while 块中错误发生的位置。

如果使用 38.2 节中的 while 的实现，而不是 Tcl 内建的实现，就不会得到第一对括号中的消息。下面的 C 代码可替代 Tcl_WhileObjCmd 中 while 循环以及它后面的代码，它使用 Tcl_AppendResult 添加括号标志。

```
...
while (1) {
        /* Get the results of the expression which indicates
        whether to continue or not. */
    result = Tcl_ExprBooleanObj(interp, objv[1], &value);
    if (result != TCL_OK) {
        return result;
    }
    if (!value) {
        break;
    }

    /* Evaluate the while body. */
    result = Tcl_EvalObjEx(interp, objv[2], 0);
    if ((result != TCL_OK) &&
        (result != TCL_CONTINUE)) {
    if (result == TCL_ERROR) {

        Tcl_Obj *options = Tcl_GetReturnOptions(interp, code);
        Tcl_Obj *key = Tcl_NewStringObj("-errorline", -1);
        Tcl_Obj *value;
        int line;

        Tcl_DictObjGet(NULL,options, key, &value);
        Tcl_GetIntFromObj(NULL, value, &line);
        Tcl_DecrRefCount(key);
        Tcl_DecrRefCount(options);

        char msg[32 + TCL_INTEGER_SPACE];
        sprintf(msg, "\n    (\"while\" body line %d)", line);
        Tcl_AddErrorInfo(interp, msg);
    }
    break;
  }
}

if (result == Tcl_BREAK) {
    result = TCL_OK;
}
if(result == TCL_OK) {
    Tcl_ResetResult(interp);
}
return result;
```

返回的选项字典中的关键字-errorline 在命令过程返回 TCL_ERROR 时由 Tcl_EvalObjEx 设置，它给出了产生错误的命令的行号。行号 1 表示处理的脚本的第一行，这个示例中就是包含左大括号的那一行；产生错误的 if 命令处在第 3 行。

提示： Tcl 8.5 以前的版本，错误所在的行号仅能由 interp->errorLine 直接访问。在 Tcl 8.5 中，强烈建议不要使用这种机制，Tcl 8.6 可能会将其取消。以前的代码中使用 interp->errorLine 进行访问的语句应该用前面展示的在返回的选项字典中以-errorline 关键字进行访问的语句。未来的 Tcl 版本可能会提供更直接地处理行号信息的工具，更多信息参见 Tcl 参考文档。

对于简单的 Tcl 命令，不必调用 Tcl_AddErrorInfo：由 Tcl_Eval 提供的信息就足够了。然而，如果编写了调用 Tcl_Eval 的代码，那么在它返回错误时调用 Tcl_AddErrorInfo 则是一个很好的做法，可以提供 Tcl_Eval 的调用信息，获取发生错误的行号。

想要设置 errorInfo 变量的值时必须调用 Tcl_AddErrorInfo，而不是直接设置，因为 Tcl_AddErrorInfo 包含了特别的代码，用于检测错误出现后的第一个调用，并清除 errorInfo 中原来的内容。

38.6 TcL_Panic

当 Tcl 系统遇到它无法处理的问题(例如内存不足)时，它就会调用 Tcl_Panic 函数。通常，这会向标准错误通道输出错误消息，调用 abort 结束当前进程。您可以在自己的扩展中使用它，形如：

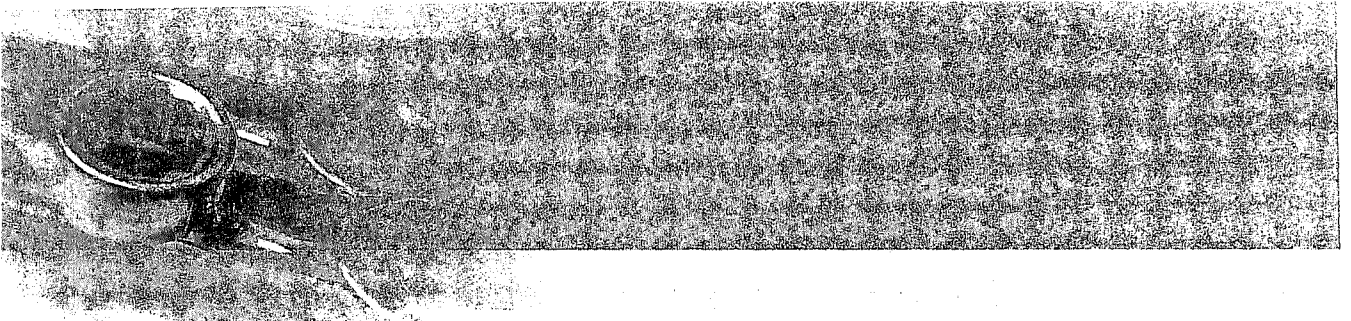
```
Tcl_Panic(CONST char* format, arg, arg, arg, ... );
```

format 是 printf 中那种格式字符串，*arg* 是要输出的参数。这个函数不会返回，所以如果要在扩展程序中调用它，确保它是您进行的最后一个调用。如果普通的错误机制就足够用了，就不要使用这个函数，只在应用程序应该立即挂起时使用它。

当 Tcl 嵌入应用程序中时，可能需要关闭 Tcl，同时让进程继续运行，或者是让应用程序按照它自己的规则关闭。当然，一旦 Tcl_Panic 被调用，就必须确定应用程序不再试图使用 Tcl。使用 Tcl_SetPanicProc 可以注册一个函数取代默认的 PanicProc 被调用。

```
static void
My_Panic TCL_VARARGS_DEF(CONST char *, arg1) {
    ...
    Application-specific shutdown
    ...
    fprintf(stderr,
        "Tcl has panicked, terminating application\n");
    abort();
}
...
Tcl_SetPanicProc(My_Panic);
```

这样就注册了一个函数，用来进行应用程序特定的清理，输出错误消息，然后终止进程。



第39章 字符串工具

Tcl 提供了很多处理函数，使得对字符串和哈希表的使用更为简便。C 程序也可以独立地使用 Tcl 库中的大部分内容(编码框架和正则表达式引擎除外)。本章介绍 Tcl 中用于字符串操作的库函数，包括：用于创建任意长字符的动态字符串机制；进行简单字符串匹配的函数；处理 Unicode 字符和 UTF-8 字符串的一系列函数；检测命令的语法完整性的函数。

39.1 本章出现的函数

本章出现的字符串工具函数如下。

- `void Tcl_DStringInit(Tcl_DString *dsPtr)`
  初始化 *dsPtr* 的值为空字符串(*dsPtr* 以前的内容不经清理直接抛弃)。
- `char *Tcl_DStringAppend(Tcl_DString *dsPtr,`
  `const char *string, int length)`
  把 *string* 中的 *length* 个字节添加到 *dsPtr* 的值当中，返回 *dsPtr* 的新值。如果 *length* 小于 0，则添加整个 *string*。
- `char *Tcl_DStringAppendElement(Tcl_DString *dsPtr,`
  `const char *string)`
  将 *string* 的值转化为适当的列表元素，添加到 *dsPtr* 的值当中(如果需要，加入分隔空白)。返回 *dsPtr* 的新值。
- `Tcl_DStringStartSublist(Tcl_DString *dsPtr)`
  向 *dsPtr* 中添加适当字节的内容(如{)以开始一个子列表。
- `Tcl_DStringEndSublist(Tcl_DString *dsPtr)`
  向 *dsPtr* 中添加适当字节的内容(如})以结束一个子列表。
- `char *Tcl_DStringValue(Tcl_DString *dsPtr)`
  返回指向当前的 *dsPtr* 值的指针。
- `int Tcl_DStringLength(Tcl_DString *dsPtr)`
  返回 *dsPtr* 的值的字节数，不包括作为结束符的空字节。

- `Tcl_DStringTrunc(Tcl_DString *dsPtr, int newLength)`
  如果 *dsPtr* 的长度超过 *newLength* 个字节，则截取为只包含前 *newLength* 个字节的字符串。
- `Tcl_DStringFree(Tcl_DString *dsPtr)`
  释放为 *dsPtr* 分配的所有内存，将 *dsPtr* 的值重新初始化为空字符串。
- `Tcl_DStringResult(Tcl_Interp *interp, Tcl_DString *dsPtr)`
  将 *dsPtr* 的值移动到解释器的结果中，然后将 *dsPtr* 的值重新初始化为空字符串。
- `Tcl_DStringGetResult(Tcl_Interp *interp,`
  `Tcl_DString *dsPtr)`
  将解释器的结果内容移动到 *dsPtr* 中，然后将解释器的结果重新初始化为空字符串。
- `int Tcl_StringMatch(const char *string,`
  `const char *pattern)`
  如果按照通配符模式匹配规则，*string* 与 *pattern* 匹配，则返回 1，否则返回 0。
- `int Tcl_StringCaseMatch(const char *string,`
  `const char *pattern, int nocase)`
  如果按照通配符模式匹配规则，*string* 与 *pattern* 匹配，则返回 1，否则返回 0。如果 *nocase* 为 0，匹配区分大小写；如果它是 1，匹配不区分大小写。
- `int Tcl_RegExpMatchObj(Tcl_Interp *interp,`
  `Tcl_Obj *textObj, Tcl_Obj *patObj)`
  检测 *patObj* 中的正则表达式是否与 *textObj* 中的字符串匹配，如果匹配，返回 1，否则返回 0。如果编译 *patObj* 时出错，错误消息会留给解释器(如果不是 NULL)，返回-1。
- `int Tcl_RegExpMatch(Tcl_Interp *interp, char *text,`
  `const char *pattern)`
  检测 *pattern* 中的正则表达式是否与 *text* 中的字符串匹配，如果匹配，返回 1，否则返回 0。如果在编译 *pattern* 时出错，错误消息会留给解释器(如果不是 NULL)，返回-1。
- `Tcl_RegExp Tcl_RegExpCompile(Tcl_Interp *interp,`
  `const char *pattern)`
  将 *pattern* 编译为正则表达式，返回它的句柄。如果在编译 *pattern* 时出错，错误消息会留给解释器(如果不是 NULL)，返回 NULL。
- `int Tcl_RegExpExec(Tcl_Interp *interp, Tcl_RegExp regexp,`
  `char *text, char *start)`
  检测编译过的正则表达式 *regexp* 是否与字符串 *text* 匹配，*text* 应该是字符串 *start*(重复匹配时的必要信息)的一个子字符串。如果有匹配，返回 1；如果没有匹配，返回 0，如果遇到错误则返回-1(如果错误消息不是 NULL，将它交给解释器)。

- Tcl_RegExpRange(Tcl_RegExp *regexp*, int *index*,
  const char **startPtr*, const char **endPtr*)
  在与表达式 *regexp* 成功匹配后，提供有关子表达式范围 *index* 的更多信息。*startPtr* 和 *endPtr* 参数分别指明了匹配范围的起点和终点。
- Tcl_RegExp Tcl_GetRegExpFromObj(Tcl_Interp **interp*,
  Tcl_Obj **patObj*, int *cflags*)
  将 *patObj* 中的模式编译为一个正则表达式，给出 *cflags* 中的编译标志。如果在编译中遇到错误，返回 NULL，将错误消息存储在解释器中。支持的编译标志详见参考文档。
- int Tcl_RegExpExecObj(Tcl_Interp **interp*,
  Tcl_RegExp *regexp*, Tcl_Obj **textObj*, int *offset*,
  int *nmatches*, int *eflags*)
  将正则表达式 *regexp* 与 *textObj* 中的字符串匹配，从字符串的第 *offset* 个字符开始。*nmatches* 指定了需要的最大子表达式数量，*eflags* 给出了正则表达式执行标志的数量。如果表达式匹配，返回 1，否则返回 0。如果在匹配中遇到错误，则返回-1，将错误消息留给解释器。支持的执行标志详见参考文档。
- Tcl_RegExpGetInfo(Tcl_RegExp *regexp*,
  Tcl_RegExpInfo **infoPtr*)
  获取关于正则表达式 *regexp* 的上一次匹配的扩展信息。这些信息给出了可能的子表达式的数量，实际匹配的数量，每个子表达式在字符串中匹配的位置，以及(根据编译标志的规定)是否能够与正则表达式完全匹配，可能还有其他文本信息。
- char *Tcl_ExternalToUtfDString(Tcl_Encoding *encoding*,
  const char **src*, int *srcLen*, Tcl_DString **dsPtr*)
  将 *src*(长度为 *srcLen*，如果 *srcLen* 为负，到第一个为零的字节为止)中的字符数据从 *encoding* 编码转换为 Tcl 内部使用的 UTF-8 编码，返回结果字符串。*dsPtr* 中的缓冲器被作为工作空间使用；不再需要字符串转换时应该用 Tcl_DStringFree 将其释放。如果 *encoding* 为 NULL，则使用当前的系统编码。
- char *Tcl_UtfToExternalDString(Tcl_Encoding *encoding*,
  const char **src*, int *srcLen*, Tcl_DString **dsPtr*)
  将 *src*(长度为 *srcLen*，如果 *srcLen* 为负，到第一个为零的字节为止)中的字符数据从 Tcl 内部使用的 UTF-8 编码转换为 *encoding* 编码，返回结果字符串。*dsPtr* 中的缓冲器被作为工作空间使用；不再需要字符串转换时应该用 Tcl_DStringFree 将其释放。如果 *encoding* 为 NULL，则使用当前的系统编码。
- Tcl_Encoding Tcl_GetEncoding(Tcl_Interp **interp*,
  const char **name*)
  返回名为 *name* 的编码，如果没有该名称的编码，则返回 NULL，在解释器结果中放置错误消息。不再需要返回的编码时，应该用 Tcl_FreeEncoding 将其释放。
- Tcl_FreeEncoding(Tcl_Encoding *encoding*)

释放前面由 Tcl_GetEncoding 返回的编码。

- `int Tcl_UniCharToUtf(int ch, char *buf)`
  将 *ch* 转换为 UTF-8，存放在 *buf* 缓冲区中。返回进行了写操作的字节数。
- `int Tcl_UtfToUniChar(const char *src, Tcl_UniChar *chPtr)`
  将 *src* 中的第一个 UTF-8 字符转换为 Unicode，写入 *chPtr* 指向的变量。返回读取到的字节数。
- `char *Tcl_UniCharToUtfDString(const Tcl_UniChar *uniStr, int uniLength, Tcl_DString *dsPtr)`
  将 *uniStr* 字符串中 *uniLength* 个 Unicode 字符转换为 UTF-8，存放在给出的 Tcl_Dstring 中。返回生成的 UTF-8 字符串。
- `Tcl_UniChar *Tcl_UtfToUniCharDString(const char *src, int length, Tcl_DString *dsPtr)`
  将 *src* 中字符串的 *length* 个 UTF-8 字符转换为 Unicode 字符串，将其存放在给出的 Tcl_Dstring 中。返回生成的 Unicode 字符串。如果 *length* 为-1，则对输入字符串中所有的 UTF-8 字符进行转换。
- `int Tcl_UniCharLen(const Tcl_UniChar *uniStr)`
  返回 *uniStr* 中由 NULL 终止的 Unicode 字符串的长度(以 Unicode 字符为单位)。
- `int Tcl_UniCharNcmp(const Tcl_UniChar *ucs, const Tcl_UniChar *uct, int numChars)`
  比较两个 Unicode 字符串的前 *numChars* 个字符是否相同，如果相同，返回 0。如果在它们第一次出现不同的地方，*ucs* 的字符小，则返回一个负数；如果 *uct* 的字符小，则返回一个正数。
- `int Tcl_UniCharNcasecmp(const Tcl_UniChar *ucs, const Tcl_UniChar *uct, int numChars)`
  不区分大小写地比较两个 Unicode 字符串的前 *numChars* 个字符是否相同，如果相同，返回 0。如果在它们第一次出现不同的地方，*ucs* 的字符小，则返回一个负数；如果 *uct* 的字符小，则返回一个正数。
- `int Tcl_UniCharCaseMatch(const Tcl_UniChar *uniStr, const Tcl_UniChar *uniPattern, int nocase)`
  如果按照通配符匹配规则 *uniStr* 和 *uniPattern* 匹配，则返回 1，否则返回 0。如果 *nocase* 为非零值，匹配中不区分大小写，否则区分大小写。
- `int Tcl_UtfNcmp(const char *cs, const char *ct, int numChars)`
  比较两个 UTF-8 字符串的前 *numChars* 个字符是否相同，如果相同，返回 0。如果在它们第一次出现不同的地方，*cs* 的字符小，则返回一个负数；如果 *ct* 的字符小，则返回一个正数。
- `int Tcl_UtfNcasecmp(const char *cs, const char *ct, int numChars)`

不区分大小写地比较两个 UTF-8 字符串的前 *numChars* 个字符是否相同，如果相同，返回 0。如果在它们第一次出现不同的地方，*cs* 的字符小，则返回一个负数；如果 *ct* 的字符小，则返回一个正数。

- `int Tcl_UtfCharComplete(const char *src, int length)`
  检查 *src* 中的 *length*(如果 *length* 为-1，则到下一个为零的字节为止)个字节是否可表示为至少一个 UTF-8 字符。
- `int Tcl_NumUtfChars(const char *src, int length)`
  返回 *src* 中的字符串的 UTF-8 字符数目。只使用前 *length* 个字节，如果 *length* 为-1，则使用遇到第一个 NULL 之前的全部字节。
- `const char *Tcl_UtfFindFirst(const char *src, int ch)`
  返回在以 NULL 结尾的 UTF-8 编码的字符串 *src* 中的第一个出现的 *ch* 字符的实例，如果该字符没有出现，则返回 NULL。
- `const char *Tcl_UtfFindLast(const char *src, int ch)`
  返回在以 NULL 结尾的 UTF-8 编码的字符串 *src* 中的最后一个出现的 *ch* 字符的实例，如果该字符没有出现，则返回 NULL。
- `const char *Tcl_UtfNext(const char *src)`
  返回字符串 *src* 中下一个 UTF-8 字符的位置。
- `const char *Tcl_UtfPrev(const char *src,`
  `const char *start)`
  返回 *src* 的前一个 UTF-8 字符的位置，字符的起始位置由 *start* 指定。
- `Tcl_UniChar Tcl_UniCharAtIndex(const char *src, int index)`
  返回从 UTF-8 字符串 *src* 中取得的第 *index* 个 Unicode 字符。字符串必须包含至少 *index* 个 Unicode 字符。*index* 不能为负。
- `const char *Tcl_UtfAtIndex(const char *src, int index)`
  返回指向 UTF-8 字符串 *src* 中第 *index* 个 UTF-8 字符的指针。字符串必须包含至少 *index* 个 UTF-8 字符。
- `int Tcl_UtfBackslash(const char *src, int *readPtr,`
  `char *dst)`
  解析 *src* 中的 Tcl 反斜线序列，将确认的 UTF-8 字符存放在 *dst* 字符串中。*readPtr* 指向的变量更新为 *src* 被解析的字节数目。返回写入 *dst* 的字节数目。
- `int Tcl_CommandComplete(const char *cmd)`
  如果 *cmd* 为一个或多个语法上完整的命令，则返回 1；如果 *cmd* 的最后一条命令因为括号不配对或其他原因在语法上不完整，返回 0。

39.2　动态字符串

动态字符串是可以无限添加内容的字符串。向动态字符串中添加内容时，Tcl 自动地根据需要扩大分配给它的内存空间。如果字符串很短，Tcl 会为它使用一个静态缓冲区，

而不去分配内存。它还形成了一个使用方便的动态管理缓冲区的系统，因为它为您处理了内存管理的低层细节。Tcl 为操作动态字符串提供了 11 个函数和宏。

- Tcl_DStringInit 将一个动态字符串初始化为空字符串。注意仅仅初始化过但没有加入内容的动态字符串无需传给 Tcl_DStringFree。
- Tcl_DStringAppend 将内容加入动态字符串。动态字符串仍以 NULL 结尾。
- Tcl_DStringAppendElement 将新的列表元素加入动态字符串。
- Tcl_DStringStartSublist 和 Tcl_DStringEndSublist 用于在动态字符串中创建子列表。
- Tcl_DStringValue 返回动态字符串的当前值。
- Tcl_DStringLength 返回动态字符串的当前长度。
- Tcl_DStringTrunc 截短动态字符串。
- Tcl_DstringFree 释放为动态字符串分配的存储空间，重新初始化动态字符串。
- Tcl_DStringResult 将动态字符串的值移动到解释器的结果中，然后重新初始化动态字符串。Tcl_DStringGetResult 进行它的逆操作，将结果移动到动态字符串中，然后重置解释器。

下面的代码使用了这些功能的一部分，实现了 map 命令，该命令获取一个列表，然后对源列表中的每个元素做一些操作，产生新的列表。map 命令需要两个参数：一个列表和一个 Tcl 命令。对列表中的每一个元素，把该元素作为额外的参数调用给出的 Tcl 命令。对所有元素分别执行命令的结果组成了一个新的列表，然后将这个列表作为结果返回。下面是使用 map 命令的一些示例。

```
    proc inc {x} {expr {$x + 1}}
    map {4 18 16 19 -7} inc
⇒   5 19 17 20 -6
    proc addz {x} {concat $x z}
    map {a b {a b c}} addz
⇒   {a z} {b z} {a b c z}
```

下面是实现 map 命令的命令函数。

```
int MapCmd(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[]) {
    Tcl_DString command, newList;
    int listArgc, i, result;
    char **listArgv;

    if (argc != 3) {
        Tcl_SetResult(interp, "wrong # args",
            TCL_STATIC);
        return TCL_ERROR;
    }
    if (Tcl_SplitList(interp, argv[1], &listArgc,
        &listArgv) != TCL_OK) {
        return TCL_ERROR;
    }
    Tcl_DStringInit(&newList);
    Tcl_DStringInit(&command);
    for (i=0; i<listArgc ; i ++) {
        Tcl_DStringAppend(&command, argv[2], -1);
        Tcl_DStringAppendElement(&command, listArgv[i]);
```

```
            result = Tcl_Eval(interp,
                Tcl_DStringValue(&command));
            Tcl_DStringFree(&command);
            if(result != TCL_OK) {
                Tcl_DStringFree(&newList);
                ckfree((char *) listArgv);
                return result;
            }
            Tcl_DStringAppendElement(&newList,
                Tcl_GetResult(interp));
        }
        Tcl_DStringResult(interp, &newList);
        ckfree((char *) listArgv);
        return TCL_OK;
    }
```

MapCmd 使用了两个动态字符串。一个保存结果列表，另一个保存各步要执行的那个命令。需要第一个字符串的原因是命令的长度是不可预知的，需要第二个字符串的原因是在生成列表时就要存放结果列表(该信息不能由命令的结果保存，因为命令处理下一个列表元素时产生的结果会覆盖它)。动态字符串的类型结构为 Tcl_DString。这个结构保存着有关字符串的信息，例如指向当前值的指针、用于小字符串的小数组以及长度。Tcl 并不为 Tcl_DString 结构进行配置，而是由您对它进行配置(例如作为一个局部变量)，然后把它的地址传给动态字符串库函数。您应该从不直接访问 Tcl_DString 结构的域，而是使用 Tcl 提供的宏和函数。

在检查它的参数数目之后，取得源列表中所有的元素，初始化动态字符串，MapCmd 进入一个循环，处理列表的元素。首先创建为每个元素执行的命令。它调用 Tcl_DStringAppend 为命令添加了由 argv[2] 提供的部分，然后它调用 Tcl_DStringAppendElement 将列表元素添加为额外的参数。这两个函数的相似之处在于它们都向动态字符串中添加了内容。然而，Tcl_DStringAppend 添加的信息是原始的文本，而 Tcl_DStringAppendElement 把它的字符串参数转化为适当的列表元素，再将列表元素加到动态字符串中(如果需要，添加相应的分隔空白)。

这种情况下，用 Tcl_DStringAppendElement 处理列表元素的重要意义在于，在 Tcl 命令的构建过程中它会成为一个单词。如果使用的是 Tcl_DStringAppend，而元素为 a b c(如本节开始的那个示例所示)，那么传给 Tcl_Eval 的命令就是 addz a b c，这会导致错误(因为给 addz 过程传入了过多的参数)。当使用 Tcl_DStringAppendElement 时，命令 addz {a b c} 则是正确的。[①]

一旦 MapCmd 为每个元素创建了执行的命令，它就调用 Tcl_Eval 来处理命令。Tcl_DStringFree 调用释放了为命令字符串分配的内存，将动态字符串重设为空，以便下一条命令使用。如果命令返回了错误，则 MapCmd 返回同样的错误；否则，它使用 Tcl_DStringAppendElement 将命令的结果作为新的元素添加到结果列表中。

MapCmd 在所有的元素都完成处理后调用 Tcl_DstringResult，将字符串的值高效率地

① 还有一些其他方法可以正确地在调用命令时提供额外的参数。例如，Tcl_NewListObj 以及其他列表创建函数都可以构建正确的用引号括起来的字符串，可以把它提供给 Tcl_EvalObjEx，而 Tcl_EvalObjv 允许执行一条命令，而完全不对它进行解析。

转化为解释器的结果(例如，它可能将动态分配的缓冲区的所有权转给解释器，而不是复制该缓冲区)。

在返回之前，MapCmd 必须释放所有为动态字符串分配的内存空间。实际上，Tcl_DStringFree 已经释放了命令字符串的内存，而 Tcl_DStringResult 已经释放了 newList 的内存。

使用 Tcl_Obj API 可能更有效率地实现这样的 MapCmd 命令。然而，虽然那样可能使过程的很多部分更有效率，这里介绍的机制(使用 Tcl_DString)仍然是最简单的方法。的确，这个命令也可以如下编写。

```
int MapCmd(ClientData clientData, Tcl_Interp *interp,
        int objc, Tcl_Obj *const objv[]) {
    Tcl_DString command;
    int listObjc, i, result;
    Tcl_Obj *newList, **listObjv;

    if (objc != 3) {
        Tcl_WrongNumArgs(interp, 1, objv,
                "list command");
        return TCL_ERROR;
    }
    if (Tcl_ListObjGetElements(interp, objv[1],
        &listArgc, &listObjv) != TCL_OK) {
        return TCL_ERROR;
    }
    newList = Tcl_NewObj();
    Tcl_DStringInit(&command);
    for (i=0 ; i<listObjc ; i++) {
        Tcl_DStringAppend(&command,
            Tcl_GetString(objv[2]), -1);
        Tcl_DStringAppendElement(&command,
            Tcl_GetString(listObjv[i]));
        result = Tcl_Eval(interp,
            Tcl_DStringValue(&command));
        Tcl_DStringFree(&command);
        if (result != TCL_OK) {
            Tcl_DecrRefCount(newList);
            return result;
        }
        Tcl_ListObjAppendElement(NULL, newList,
                Tcl_GetObjResult(interp));
    }
    Tcl_SetObjResult(interp, newList);
    return TCL_OK;
}
```

如您所见，这段代码和前一段是十分相似的，只不过函数名称不同。

39.3 字符串匹配

函数 Tcl_StringMatch 和 Tcl_StringCaseMatch(扩展之处在于允许控制匹配是否区分大小写)提供与 string match 命令相同的功能。给定一个字符串和一个模式，如果按照通配符匹配规则字符与模式匹配，则返回 1，否则返回 0。例如，下面这个命令函数使用 Tcl_StringMatch 来实现简化版的 lsearch，只提供通配符规则的匹配。它返回列表中第一个

与给定模式匹配的元素的索引，如果没有元素匹配，则返回-1。

```
int LsearchCmd(ClientData clientData,
        Tcl_Interp *interp, int objc,
        Tcl_Obj *const objv[]) {
    char *pattern;
    int i, elemc;
    Tcl_Obj **elemv;
    if (objc != 3) {
        Tcl_WrongNumArgs(interp, 1, objv,
            "list pattern");
        return TCL_ERROR;
    }
    if (Tcl_ListObjGetElements(interp, objv[1],
            &elemc, &elemv) != TCL_OK) {
        return TCL_ERROR;
    }
    pattern = Tcl_GetString(objv[2]);
    for (i=0 ; i<elemc ; i++) {
        if (Tcl_StringMatch(Tcl_GetString(elemv[i]),
                pattern)) {
            Tcl_SetObjResult(interp, Tcl_NewIntObj(i));
            return TCL_OK;
        }
    }
    Tcl_SetObjResult(interp, Tcl_NewIntObj(-1));
    return TCL_OK;
}
```

39.4 正则表达式匹配

当 Tcl_StringMatch 的简单匹配功能不能满足需要时，通常就需要使用正则表达式。第5章在 Tcl 脚本层级讲述了正则表达式的语法和功能；本节讲述 Tcl 中处理正则表达式的 C 语言 API。

用于正则表达式匹配的最简单的函数是 Tcl_RegExpMatch 和 Tcl_RegExpMatchObj。这两个函数(它们的不同仅在于参数类型的不同)检测正则表达式模式是否与给出的字符串匹配。我们可以利用它们来改写前面的类似 lsearch 的简单示例，如下使用正则表达式。

```
int RLsearchCmd(ClientData clientData,
        Tcl_Interp *interp, int objc,
        Tcl_Obj *const objv[]) {
    Tcl_Obj *pattern;
    int i, elemc, result;
    Tcl_Obj **elemv;
    if (objc != 3) {
        Tcl_WrongNumArgs(interp, 1, objv, "list pattern");
        return TCL_ERROR;
    }
    if (Tcl_ListObjGetElements(interp, objv[1],
        &elemc, &elemv) != TCL_OK) {
        return TCL_ERROR;
    }
    pattern = objv[2];
    for (i=0 ; i<elemc ; i++) {
        result = Tcl_RegExpMatchObj(interp, elemv[i], pattern);
        if (result == -1) {
            return TCL_ERROR;
```

```
        }
        if (result == 1) {
            Tcl_SetObjResult(interp, Tcl_NewIntObj(i));
            return TCL_OK;
        }
    }
    Tcl_SetObjResult(interp, Tcl_NewIntObj(-1));
    return TCL_OK;
}
```

这里复杂度的增加主要来自于对正则表达式的编译在表达式无效时就会失败。除此之外，代码结构与前面那个简单的模式匹配代码是基本相同的。

当情况比较复杂时，例如需要把正则表达式高效率地应用到很多字符串时，或在匹配中会重复用到某些字符串，或者需要访问匹配的子表达式时，就需要以更加复杂的方式使用正则表达式 API。第一层的复杂性要求通过 Tcl_RegExpCompile 和 Tcl_RegExpExec 把正则表达式的编译和处理阶段分隔开。这些函数也可以控制从字符串的什么位置开始进行匹配。它们还允许使用 Tcl_RegExpRange 来确定与正则表达式的子表达式匹配的子字符串。

还有更深一层的复杂性。Tcl_GetRegExpFromObj 允许使用编译标志，生成正则表达式匹配器，控制正则表达式的文法，以及在字符串中的换行符具体如何处理等事项。Tcl_RegExpExecObj 允许的控制项包括了 Tcl_RegExpExec 的功能，以及指定需要处理的子表达式数量，指定对字符串的结束符如何处理。Tcl_RegExpGetInfo 提供的信息与 Tcl_RegExpRange 相似，但是更加详细，也更适于由 Tcl_Obj API 使用。

提示： 在与正则表达式匹配时，基于 Tcl_Obj 的 API 效率明显更高。因为 Tcl_Obj 系统对缓存的使用更有效，在使用大量的表达式时尤其如此。然而，当使用的表达式数量较小时，效率差距没有那么显著，因为正则表达式引擎为编译后的正则表达式，在每个线程中都保有独立的缓存。还应该注意，在几乎所有情况下，只要能够由文本字符串匹配和 Tcl_StringMatch 处理，在进行匹配时它们都拥有更高的效率；仅当确实需要正则表达式的强大功能时才应该使用它。

39.5 处理字符编码

字符编码是从书面语言的字符和符号到计算机使用的二进制格式的映射。Tcl 内部将字符串用 Unicode 编码以 UTF-8 格式表达。然而，当需要把一个字符串传递给其他非 Tcl API 时，很可能需要将字符串转换到不同的编码。例如，如果将一个字符串传递给操作系统函数，就需要以系统编码发送字符串(北美通常是 ISO 8859-1，在其他地区则各有不同)；如果不这样做，很多字符(例如版权标记、引号、各种货币符号等)会被由一个字符解析成多个字符，通常还会出现一些莫名其妙的重音符号。第 5 章在 Tcl 脚本层级讲述了命令；本节讲述处理字符编码的 Tcl 的 C 语言 API。

提示： Tcl 内部将零字节反向表达为相应的多字节序列，这样允许普通的 C 语言字符串工具函数正确处理 UTF-8 字符串。注意只有内部的字符串是这样表达的；外部读写的字符串不会使用这种编码方案。

Tcl 提供了很多函数，处理在 Tcl 的 UTF-8 的表达形式与外部编码之间转换的任务。Tcl_UtfToExternalDString 获取一个 Tcl 内部字符串，将它转换为其他的编码格式，存放在 Tcl_Dstring 当中，而 Tcl_ExternalToUtfDString 则进行它的逆操作。为了使用这些函数，需要用 Tcl_GetEncoding 获取编码的句柄。如果给出 NULL 而非编码句柄，那么就使用系统编码(通常都可以这样做，特别是在使用操作系统 API 时)。在使用了由 Tcl_GetEncoding 获取的句柄之后，应该总是使用 Tcl_FreeEncoding 来释放句柄。

例如，下面的函数 LogCmd 就实现了向 POSIX 系统日志中写入日志消息的 Tcl 命令。检查确认提供的参数数目正确后，它使用 Tcl_UtfToExternalDString 将 Tcl 内部编码的字符串转换为系统编码，然后将它作为一条消息写入日志中。

```
int LogCmd(ClientData clientData, Tcl_Interp *interp,
        int objc, Tcl_Obj *const objv[]) {
    Tcl_DString buffer;
    char *messageStr;

    if (objc != 2) {
        Tcl_WrongNumArgs(interp, 1, objv, "message");
        return TCL_ERROR;
    }
    Tcl_DStringInit(&buffer);
    messageStr = Tcl_UtfToExternalDString(NULL,
        Tcl_GetString(objv[1]), -1, &buffer);
    openlog("TclExample", 0, LOG_USER);
    syslog(LOG_INFO, "%s", messageStr);
    closelog();
    Tcl_DStringFree(&buffer);
    return TCL_OK;
}
```

与编码有关的函数还有 Tcl_UtfToExternal 和 Tcl_ExternalToUtf。这些函数致力于提供高效率的流转换支持；在转换整个字符串时，使用基于 Tcl_DString 的 API 更方便一些。

必须处理编码问题的最常见的地方就是网络套接字。因为在国际互联网中，不能指望其他人都使用和您一样的编码。的确，为各个通道提供各自的编码支持是很好用的，也就是说，只要正确地为通道配置了编码设置，就不必再去显式地管理编码工作。然而，在某些协议下这会变得特别地复杂，特别是那些混合着不同的编码方案的情况(例如，二进制和文本编码)。

39.6 处理 Unicode 和 UTF-8 字符串

虽然从概念上说它们使用的是相同的字符集，Unicode 字符串(严格地说，指定了大小端的主机系统的 UCS-2 字符串)和 UTF-8 字符串的属性还是有着本质的不同。具体来说，Unicode 字符串可以建立索引，到达任意偏移量的位置都只需要等同的时间，可以快速地对它们进行各种复杂的操作；而 UTF-8 字符串更易于在传统的 API 之间传递。Tcl 提供了很多用于处理 Unicode 和 UTF-8 字符串的函数。

函数 Tcl_UniCharToUtf 和 Tcl_UtfToUniChar 提供了 Unicode 和 UTF-8 字符之间的逐一基本映射。对整个的 Unicode 和 UTF-8 字符串进行处理的扩展函数有

Tcl_UniCharToUtfDString 和 Tcl_UtfToUniCharDString，它们的目标都是 Tcl_DString 中的缓冲区。

Tcl 还提供了很多类似的 C 字符串函数，表 39.1 展示了 C 字符串函数与类似的 Tcl 函数的对应关系。对于处理 UTF-8 字符串的 Tcl_StringCaseMatch 命令，也有一个类似的处理 Unicode 字符串的命令：Tcl_UniCharCaseMatch。

表 39.1 与 C 字符串函数相类似的 Tcl Unicode 和 UTF-8 字符串函数

C 字符串函数	类似的 Unicode 字符串函数	类似的 UTF-8 字符串函数
strlen	Tcl_UniCharLen	Tcl_NumUtfChars
strncmp	Tcl_UniCharNcmp	Tcl_UtfNcmp
strncasecmp	Tcl_UniCharNcasecmp	Tcl_UtfNcasecmp
strchr		Tcl_UtfFindFirst
strrchr		Tcl_UtfFindLast

另外还有 Tcl_UtfNext 和 Tcl_UtfPrev 函数，用于在 UTF-8 编码的字符串中逐个向前移动与逐个向后移动。而 Tcl_UniCharAtIndex 和 Tcl_UtfAtIndex 分别用于在 Unicode 和 UTF-8 字符串中查看给定索引指向的字符：对于 Unicode 字符串，返回 Unicode 字符；对于 UTF-8 字符串，返回指向该位置的指针。

在处理可能包含部分 UTF-8 字符的缓冲区时，函数 Tcl_UtfCharComplete 返回它的参数指向的字符串中是否含有至少一个完整的 UTF-8 字符。需要这个函数是因为很多协议要求只对消息中的完整字符进行转化。

最后，Tcl_UtfBackslash 提供了 Tcl 支持的反斜线序列解析器。它把反斜线序列从源缓冲区复制到目标缓冲区，在这个过程中依照 Tcl 语法规则将这些序列转换为对应的字符，返回在这个过程中写入目标缓冲区的数据的字节数。

下面演示了这些函数的应用。它获取一个字符和一个指向 UTF-8 字符串的指针，返回一个新的字符串，这个字符串由这个字符和在源字符串中它左右两边的字符构成。这可以用下面的 Tcl 脚本实现：

```
set i [string first $ch $string]
expr {$i<0 ? "" : [string range $string $i-2 $i+2]}
```

下面是 strchr 对 ASCII 字符串的简单应用，这比 UTF-8 字符串要稍微复杂一点，因为字符的宽度不是常数值。

```
char *SubstringAbout(int ch, const char *string) {
    const char *end, *loc;
    char *buf;
    int i;

    /* Find the character. */
    loc = end = Tcl_UtfFindFirst(string, ch);
    if (loc == NULL) {
        return NULL;
    }
    /* Find one char past the end of the area. */
```

```
    for (i=0; *end!='\0' && i<3; i++) {
        end = Tcl_UtfNext(end);
    }
    /* Find the start of the area. */
    for (i=0; i<2; i++) {
        loc = Tcl_UtfPrev(loc, string);
    }
    /* Copy and return the substring. */
    buf = ckalloc((end-loc)+1);
    memcpy(buf, loc, end-loc);
    buf[end-loc] = '\0';
    return buf;
}
```

提示： 尽管 SubstringAbout 函数的结果字符串仅有 5 个字符长，但那 5 个字符可能会占用 15 个字节，这取决于写入系统。将来支持基础多语言平台之外的 Unicode 字符之后，占用的字节还可能更多。但是因为这个函数没有预计这个结果字符串的实际长度是多少，它仍然能够正常地完成功能。

39.7 命令完整性

当应用程序读取交互式输入的命令时，它必须等待一条完整的命令被输入之后才处理。例如，应用程序正从标准输入端读取命令，用户输入了下面三行：

```
foreach i {1 2 3 4 5} {
    puts "$i*$i is [expr $i*$i]"
}
```

如果应用程序分别读取每一行，将它传输给 Tcl_Eval，那第一行就会产生“缺少右括号”的错误。应用程序应该收集它所读取到的命令，直到命令完整(例如，括号或引号配对都完整了)，然后把完整的命令作为一个脚本来处理。函数 Tcl_CommandComplete 使得这一操作成为可能。它将一个字符串作为参数读取，如果字符串包含了语法上完整的命令，则返回 1，如果最后一条命令不完整，则返回 0。

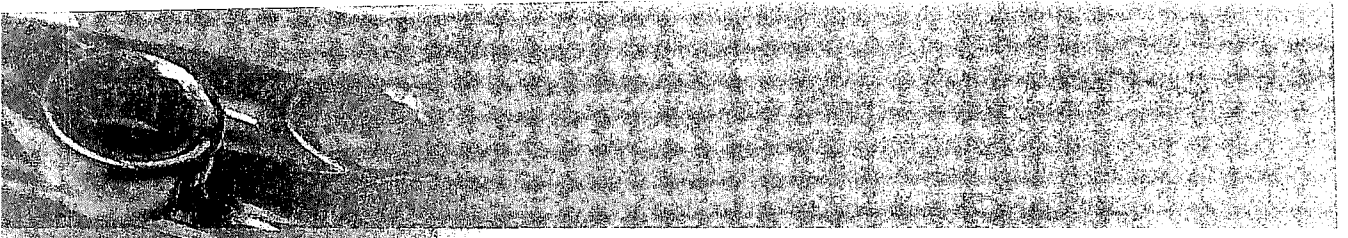
下面的 C 函数使用动态字符串和 Tcl_CommandComplete 读取和处理由标准输入端输入的命令。(为清楚起见，这里避免了使用 Tcl 通道 API。否则代码会更长，而真正需要演示的函数操作也会变得不够清晰。)它收集输入信息，直到读取到的所有命令都是完整的，然后处理所读取到的命令，返回该处理的完成代码。这里使用 Tcl_RecordAndEval 来处理命令，所以命令会被记录到历史列表中。

```
int DoOneCmd(Tcl_Interp *interp) {
    char line[200];
    Tcl_DString cmd;
    int code;
    Tcl_DStringInit(&cmd);
    While (1) {
        if (fgets(line, 200, stdin) == NULL) {
            break;
        }
        Tcl_DStringAppend(&cmd, line, -1);
        if (Tcl_CommandComplete(Tcl_DStringValue(&cmd))) {
            break;
```

```
        }
    }
    code = Tcl_RecordAndEval(interp,
            Tcl_DStringValue(&cmd), 0);
    Tcl_DStringFree(&cmd);
    return code;
}
```

在这个 foreach 示例中，DoOneCmd 在处理它们之前收集全部三行命令。如果遇到了文件尾，fgets 就会返回 NULL，即使命令还不是完整的，DoOneCmd 也会处理命令。

Tcl_CommandComplete 只为了解析正确才会检查命令的完整性。它并不保证脚本就能有正确的行为。例如，如果用户意外把 set x y 这样的命令分成了两行，即在 x 之后输入了换行符，那么每一行都会被视为完整的。第一行就是不加改动地查询了变量，第二行会调用命令 y，而这一调用很可能会产生错误。



第40章 哈 希 表

哈希表是一系列条目的集合，每一个条目包含一个关键字和一个值。不能有含有同样的关键字的两个条目存在。给出关键字，哈希表可以十分迅速地定位到它的条目及其对应的值。Tcl 包含一个用于通用目的的哈希表集，它在不少内部场合有所应用。例如，解释器中的所有命令都存放在哈希表中，每个条目的关键字就是命令名，对应的值是指向有关命令的信息的指针。一个命名空间中的所有变量也存放在另一个哈希表中，每个条目的关键字就是变量的名称，对应的值是指向有关变量的信息的指针。

Tcl 通过一系列库函数提供了哈希表功能，使得应用程序也可以使用它们。哈希表最常见的应用就是把名称和对象关联起来。为了让应用程序实现一种新的对象，它必须给出 Tcl 命令使用的该对象的文本名称。当命令函数将对象名称作为参数获得时，它必须为这个对象分配 C 数据结构。通常来说每类对象都有一个哈希表，条目的关键字是对象名，值是指向表示该对象的 C 数据结构的指针。当命令函数需要找到一个对象时，它会在哈希表中查找它的名称。如果没有关键字为该名称的条目，命令函数会返回错误。

本章中的示例使用了一个假设的应用程序，实现了名为“GIZMO”的对象。这些 GIZMO 的结构体声明如下：

```
typedef struct Gizmo {
    ... fields of gizmo object ...
} Gizmo;
```

应用程序在 Tcl 命令中使用 gizmo42 这样的名称指向 GIZMO，每个 GIZMO 的名称的尾部各有一个不同的数字。应用程序遵循 30.3 节中描述的面向动作的方法，提供了一系列 Tcl 命令来操作对象，例如用 gizmo::create 创建新的 GIZMO，用 gizmo::delete 删除已经存在的 GIZMO，用 gizmo::search 查找具有指定特征的 GIZMO，等等。

40.1 本章出现的函数

本章讨论的用于创建和操作哈希表的函数如下。

- `void Tcl_InitHashTable(Tcl_HashTable *tablePtr, int keyType)`

  创建新的哈希表，将有关该表的信息存放在 *tablePtr* 中。*keyType* 应为 TCL_STRING_KEYS 或 TCL_ONE_WORD_KEYS，或是一个大于 1 的整数。

- `Tcl_InitObjHashTable(Tcl_HashTable tablePtr)`

创建新的哈希表，将有关该表的信息存放在*tablePtr 中。哈希表的关键字是Tcl_Obj*引用(强制转换为 char*)，根据关键字的字符串表示(Tcl_GetString 返回的结果)是否相同来判断关键字是否相同。注意这是 Tcl_InitCustomHashTable 的前端，关键字的类型定义为 Tcl_Obj 的引用。

- `Tcl_InitCustomHashTable(Tcl_HashTable *tablePtr,`
  `int keyType, Tcl_HashKeyType *typePtr)`
  创建新的哈希表，将有关该表的信息存放在 *tablePtr* 中。*keyType* 的值应该是 TCL_CUSTOM_TYPE_KEYS 或 TCL_CUSTOM_PTR_KEYS，关键字的具体类型由 *typePtr* 参数更完整地指定，它必须是一个指针，指向包含关键字类型描述符的静态结构。关键字类型描述符以及如何使用该函数的内容详见参考文档。
- `void Tcl_DeleteHashTable(Tcl_HashTable *tablePtr)`
  删除哈希表中的所有条目，释放相关的内存空间。
- `Tcl_HashEntry *Tcl_CreateHashEntry(`
  `Tcl_HashTable *tablePtr, char *key, int *newPtr)`
  返回指向 *tablePtr* 中关键字为 *key* 的条目的指针，如果需要，则创建新的条目。如果创建了新的条目，*newPtr* 设置为 1，如果条目已经存在，*newPtr* 设置为 0。
- `Tcl_HashEntry *Tcl_FindHashEntry(Tcl_HashTable *tablePtr,`
  `char *key)`
  返回指向 *tablePtr* 中关键字为 *key* 的条目的指针，如果没有该条目，则返回 NULL。
- `void Tcl_DeleteHashEntry(Tcl_HashEntry *entryPtr)`
  从哈希表中删除一个条目。
- `ClientData Tcl_GetHashValue(Tcl_HashEntry *entryPtr)`
  返回该哈希表条目所对应的值。
- `void Tcl_SetHashValue(Tcl_HashEntry *entryPtr,`
  `ClientData value)`
  设置与该哈希表条目所对应的值。
- `char *Tcl_GetHashKey(Tcl_HashTable *tabelPtr,`
  `Tcl_HashEntry *entryPtr)`
  返回与该哈希表条目所对应的关键字。
- `Tcl_HashEntry *Tcl_FirstHashEntry(Tcl_HashTable *tablePtr,`
  `Tcl_HashSearch *searchPtr)`
  对哈希表中的所有元素进行搜索。将有关搜索的信息存放在 *searchPtr* 中，返回哈希表的第一个条目，如果没有条目，返回 NULL。
- `Tcl_HashEntry *Tcl_NextHashEntry(`
  `Tcl_HashSearch *searchPtr)`
  返回 *searchPtr* 指定的搜索中的下一个条目，如果该表中的所有条目都已经被返回，则返回 NULL。

- `char *Tcl_HashStats(Tcl_HashTable *tablePtr)`
  返回一个字符串，给出 *tablePtr* 的使用状态。字符串是动态配置的，必须由调用者释放。

40.2 关键字和值

Tcl 的哈希表支持 4 种不同类型的关键字。一个哈希表中的所有条目都必须使用同一类型的关键字，不同的表使用的关键字类型可以不同。关键字最常用的类型是字符串。这种情况下，每个关键字都是以 NULL 结束的任意长度的字符串，例如 gizmo18 或 Waste not want not。同一哈希表中的不同条目的关键字的长度可以不同。这里实现的 GIZMO 就使用了字符串类型的关键字。

关键字的第二种类型是单字。这种情况下，关键字是可以与单字对应的任何值，例如一个整型数。单字关键字以 char *类型作为 Tcl 使用的值进行传送，因此关键字的长度由字符指针大小限制。

第三种关键字类型是数组。这种情况下每一个关键字都是一个整型(C 的 int 类型)数组。表中的所有关键字的长度必须一样。

关键字的最后一种类型是 Tcl_Obj 实例。这种情况下，每一个关键字都是指向 Tcl_Obj 的指针，哈希表保持着对每一个独一无二的关键字值的引用。传给哈希表的 Tcl_Obj*关键字会经强制类型转换为 char*类型，与它们的字符串表达形式(由 Tcl_GetStringFromObj)进行对比。

哈希表各条目的值是 ClientData 类型的项，这种类型足够保存一个整数或一个指针。在大多数应用程序中，例如这里的 GIZMO 示例，哈希表的值就是指向记录对象的指针。当把这些指针存放到哈希表中时会把它们的类型强制转换为 ClientData，当从哈希表中取得信息时再从 ClientData 类型强制转换为对象指针。

40.3 创建和删除哈希表

每一个哈希表都表现为一个 Tcl_HashTable 类型的 C 结构体。由客户程序，而非 Tcl，为该结构体分配空间，这个结构体通常是全局变量或其他结构体的元素。在调用哈希表函数时，把指向 Tcl_HashTable 结构体的指针作为哈希表的标记使用。您不应该直接使用或修改 Tcl_HashTable 的任何成员。在需要进行这些操作时使用 Tcl 库函数。

以下是 GIZMO 应用程序如何创建哈希表的例子：

```
Tcl_HashTable gizmoTable;
...
Tcl_InitHashTable(&gizmoTable, TCL_STRING_KEYS);
```

Tcl_InitHashTable 的第一个参数是一个 Tcl_HashTable 指针，第二个参数是一个整数，指明了该表使用的关键字类型。TCL_STRING_KEYS 表示该表将使用字符串作为关键字。如果指定了 TCL_ONE_WORD_KEYS，就说明关键字的值应该是单字，例如整型数或指

针。如果第二个参数既不是 TCL_STRING_KEYS 也不是 TCL_ONE_WORD_KEYS，那它必须是一个大于 1 的整型值，这表示关键字是数组类型的，每个数组的元素数量即该参数给定的值。Tcl_InitHashTable 初始化指向空哈希表的结构体，该表拥有规定的关键字。(仅有 Tcl_InitCustomHashTable 才能使用额外的标志值 TCL_CUSTOM_TYPE_KEYS 和 TCL_CUSTOM_PTR_KEYS。)

关键字为 Tcl_Obj 引用的哈希表由 Tcl_InitObjHashTable 创建，它只获取一个参数，即一个 Tcl_HashTable 指针，与 Tcl_InitHashTable 的第一个参数相同。

Tcl_DeleteHashTable 从哈希表中移除所有的条目，释放为条目和哈希表分配的内存(为 Tcl_HashTable 结构体本身分配的空间除外，它由客户端调用的 Tcl_DeleteHashTable 删除)。例如，下面这句代码可用于删除我们前面初始化的那个哈希表。

```
Tcl_DeleteHashTable(&gizmoTable);
```

40.4 创建条目

函数 Tcl_CreateHashEntry 用给定的关键字创建一个条目，Tcl_SetHashValue 将一个值设为与该条目关联的值。例如，下面的代码可用于实现 gizmo::create 命令，创建一个新的 GIZMO 对象。

```
int GizmoCreateCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    static unsigned int id = 1;
    int new;
    Tcl_HashEntry *entryPtr;
    Gizmo *gizmoPtr;
    char nameBuf[5 + TCL_INTEGER_SPACE];

    ...check argc, etc. ...

    do {
        sprintf(nameBuf, "gizmo%u", id);
        id++;
        entryPtr = Tcl_CreateHashEntry(&gizmoTable,
            nameBuf, &new);
    } while (!new);

    gizmoPtr = (Gizmo *) ckalloc(sizeof(Gizmo));
    Tcl_SetHashValue(entryPtr, gizmoPtr);

    ... initialize *gizmoPtr, etc. ...

    Tcl_SetResult(interp, nameBuf, TCL_VOLATILE);
    return TCL_OK;
}
```

这段代码将 gizmo 和静态变量 id 的值串接起来，生成对象的名称。它将新对象的名称作为 Tcl 的结果返回，不过在生成它的时候使用了局部堆栈缓冲区 nameBuf 来保存这个名称。然后 GizmoCreateCmd 对 id 进行增加操作，使得每一个新的对象都拥有独一无二的名称；在线程安全的代码中，这需要一个互斥体加以保护，或是使用解释器的或线程的局部

变量。调用了 Tcl_CreateHashEntry 创建关键字为对象名称的新条目，它返回这个条目的标记。正常情况下，指定关键字对应的条目应该尚不存在，这种情况下 Tcl_CreateHashEntry 将 new 设置为 1，表示它创建了一个新条目。然而，有可能调用 Tcl_CreateHashEntry 时指定的关键字所对应的条目已经在哈希表中存在。在 GizmoCreateCmd 中出现这种情况只可能是因为哈希表中的条目太多，id 的增加操作已经再次把值加到了零。如果发生这种情况，Tcl_CreateHashEntry 会将 new 设置为 0；GizmoCreateCmd 会用下一个 id 值进行尝试，直到它最终找到一个未被使用的关键字。

在创建哈希表条目之后，GizmoCreateCmd 为对象的记录分配内存，调用 Tcl_SetHashValue 将记录的地址存储为哈希表条目的值。Tcl_SetHashValue 的第一个参数是哈希表条目的标记，它的第二个参数，就是该条目的新值，可以是 ClientData 的空间可以存储下的任意值。在设置了哈希表条目的值之后，GizmoCreateCmd 初始化新对象的记录，把对象的名称(在 nameBuf 中)存储为解释器的结果。

提示： Tcl 的哈希表在您添加条目时重建自身。哈希表的条目很少时，哈希桶所占用的内存也很少，但随着条目的增多，桶数组的大小也会增加。Tcl 的哈希表即使在它们的条目数量很多时也能进行高效率的操作。

在处理 Tcl_Obj 的关键字时，哈希表系统负责管理关键字的生命周期。唯一的约束是传给 Tcl_CreateHashEntry 的对象参数必须将类型转换为 char*，以保证与其类型要求符合。一旦从对象哈希表中取得了 Tcl_HashEntry 指针，对它的使用就与字符串哈希表中取得的指针相同；哈希表对它们的值的类型没有任何约束。

40.5 查找已存在的条目

函数 Tcl_FindHashEntry 在一个哈希表中定位已经存在的条目。它与 Tcl_CreateHashEntry 有些相似，只是如果哈希表中没有该关键字的条目存在时它不会创建新的条目。Tcl_FindHashEntry 通常用于根据名称查找对象。例如，GIZMO 的实现中可能包括一个名为 Getgizmo 的工具函数，与 Tcl_GetInt 相似，只不过它将字符串变量转换为 Gizmo 指针而非整型数。

```
Gizmo *GetGizmo(Tcl_Interp *interp, char *string) {
    Tcl_HashEntry *entryPtr;
    entryPtr = Tcl_FindHashEntry(&gizmoTable, stirng);
    if (entryPtr == NULL) {
        Tcl_AppendResult(interp, "no gizmo named \"",
            string, "\"", NULL);
        return NULL;
    }
    return (Gizmo *) Tcl_GetHashValue(entryPtr);
}
```

GetGizmo 在 GIZMO 哈希表中查找一个 GIZMO 名称。如果该名称存在，则 GetGizmo 使用 Tcl_GetHashValue 取得该条目中的值，将它转换为 Gizmo 指针，然后将该指针返回。如果指定名称不存在，GetGizmo 将错误消息存放到解释器的结果中，返回 NULL。在处理

Tcl_Obj*哈希表时也是如此，只不过在将关键字传给 Tcl_FindHashEntry 之前要将其类型转换为 char*。

GetGizmo 可以由任意命令函数调用，用于查找 GIZMO 对象。例如，假设有一个命令 gizmo::twist 对 GIZMO 进行一个“扭曲”操作，它需要一个 GIZMO 名称作为它的第一个参数。这个命令可以如下实现：

```
int GizmoTwistCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    Gizmo *gizmoPtr;
    ... check argc, etc. ...
    gizmoPtr = GetGizmo(interp, argv[1]);
    if (gizmoPtr == NULL) {
        return TCL_ERROR;
    }
    ... perform twist operation ...
    return TCL_OK;
}
```

40.6 搜索

Tcl 提供两个用于搜索哈希表中的条目的函数。Tcl_FirstHashEntry 发起一个搜索，返回找到的一个条目，而 Tcl_NextHashEntry 返回后续条目，直到对整个哈希表完成搜索。例如，假设您希望提供一个 gizmo::search 命令，用来对所有存在的 GIZMO 进行搜索，返回满足指定条件的 GIZMO 的名称的列表。这个命令可以如下实现：

```
int GizmoSearchCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    Tcl_HashEntry *entryPtr;
    Tcl_HashSearch search;
    Gizmo *gizmoPtr;
    ... process arguments to choose search criteria ...
    for (entryPtr = Tcl_FirstHashEntry(&gizmoTable, &search);
            entryPtr != NULL;
            entryPtr = Tcl_NextHashEntry(&search)) {
        gizmoPtr =(Gizmo *) Tcl_GetHashValue(entryPtr);
        if (...object satisfies search criteria... ) {
            Tcl_AppendElement(interp,
                Tcl_GetHashKey(&gizmoTable, entryPtr));
        }
    }
    return TCL_OK;
}
```

此例使用了一个 Tcl_HashSearch 类型的结构体来跟踪搜索；为各个搜索提供单独的 Tcl_HashSearch 结构体，就可以同时进行多个搜索。Tcl_FirstHashEntry 初始化这个结构体，返回表中的第一个条目的标记(如果表为空，返回 NULL)。Tcl_NextHashEntry 使用该结构体的信息移动到表中的后续条目；每一次对 Tcl_NextHashEntry 的调用都返回指向下一个条目的指针(没有特定的顺序)，返回 NULL 说明已经到达了哈希表的尾端。GizmoSearchCmd 从各个条目中取得相应的值，将其转换为 Gizmo 指针，然后检查该对象是否满足命令参数中指定的条件。如果满足，则 GizmoSearchCmd 使用 Tcl_GetHashKey 函

数获取对象的名称(即条目的关键字)，并调用 Tcl_AppendElement 将该名称作为一个元素添加到解释器的结果中。尽管 Tcl_GetHashKey 的结果的类型是 char*，它实际上的类型决定于创建哈希表时设置的关键字类型。

提示： 在搜索时修改哈希表的结构体是一个不安全的操作，删除由 Tcl_FirstHashEntry 及 Tcl_NextHashEntry 返回的当前条目例外。如果创建或删除了当前条目以外的条目，应该终止对该哈希表正在进行的所有搜索。

40.7　删除条目

函数 Tcl_DeleteHashEntry 从哈希表中删除条目。例如，下面这个函数使用 Tcl_DeleteHashEntry 实现了 gizmo::delete 命令，该命令获取任意个参数，删除指定的 GIZMO 对象。

```
int GizmoDeleteCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    Tcl_HashEntry *entryPtr;
    Gizmo *gizmoPtr;
    int i;
    for (i = 1; i < argc; i++) {
        entryPtr = Tcl_FindHashEntry(&gizmoTable, argv[i]);
        if (entryPtr == NULL) {
            continue;
        }
        gizmoPtr = (Gizmo *) Tcl_GetHashValue(entryPtr);
        Tcl_DeleteHashEntry(entryPtr);
        ...clean up *gizmoPtr ...
        ckfree((char *) gizmoPtr);
    }
    return TCL_OK;
}
```

GizmoDeleteCmd 检查它的所有参数，确认它们是 GIZMO 对象的名称。如果不是，则将该参数忽略；如果是，则 GizmoDeleteCmd 从哈希表的条目中取得对应的 Gizmo 指针，调用 Tcl_DeleteHashEntry 从哈希表中将该条目移除。然后对 GIZMO 对象进行必要的内部清理，释放对象的记录。

提示： Tcl 不负责管理哈希值的生命周期。如果它们是值，那它们需要显式的释放，在使用 Tcl_DeleteHashEntry 或 Tcl_DeleteHashTable 时就应该完成这个释放操作，否则会发生内存泄漏。

40.8　统计

函数 Tcl_HashStats 返回一个字符串，其内容是有关哈希表的一些统计信息。例如，可以用它来为 GIZMO 实现一个 gizmo::stat 命令。

```
int GizmoStatCmd(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[]) {
```

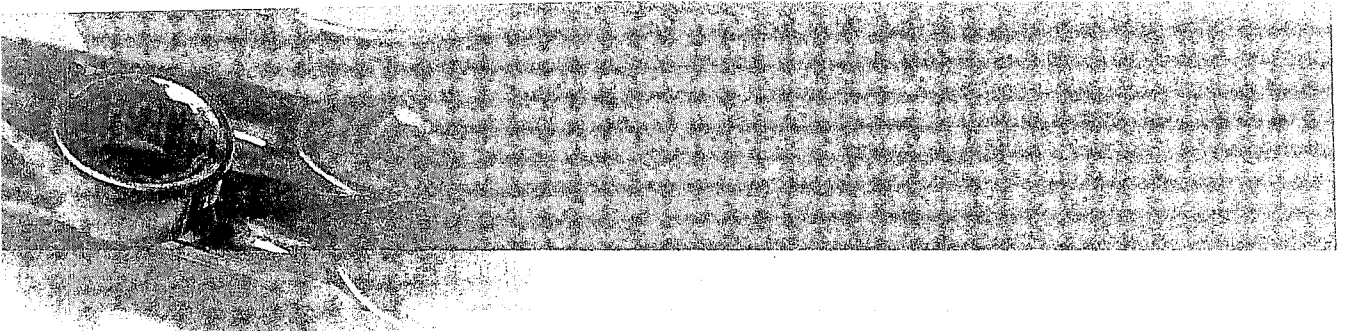
```
    if (argc != 1) {
        Tcl_SetResult(interp, "wrong # args", TCL_STATIC);
        return TCL_ERROR;
    }
    Tcl_SetResult(interp, Tcl_HashStats(&gizmoTable), TCL_DYNAMIC);
    return TCL_OK;
}
```

由 Tcl_HashStats 返回的字符串是动态配置的，必须由 ckfree 或 Tcl_Free 释放；GizmoStatCmd 使用 Tcl_SetResult 命令和生命周期模式 TCL_DYNAMIC，将该字符串设置为解释器的结果。

由 Tcl_HashStats 返回的字符串并非正式定义的，它包括的是易于阅读的信息，例如：

```
1416 entries in table, 1024 buckets
number of buckets with 0 entries: 60
number of buckets with 1 entries: 591
number of buckets with 2 entries: 302
number of buckets with 3 entries: 67
number of buckets with 4 entries: 5
number of buckets with 5 entries: 0
number of buckets with 6 entries: 0
number of buckets with 7 entries: 0
number of buckets with 8 entries: 0
number of buckets with 9 entries: 0
number of buckets with more than 10 entries: 0
average search distance for entry: 1.4
```

您可以通过这些信息看到条目是如何高效地存放在哈希表中的。例如，最后一行给出了假定对所有的条目的访问几率相等时，Tcl 在哈希表中查找条目需要消耗的平均时间。



第41章　列表和字典对象

Tcl 8.5 版提供了两种在 C 中和 Tcl 中都易于访问的数据结构：列表和字典。Tcl 脚本层次的列表操作参见第 6 章，字典参见第 7 章。当前使用的 Tcl 列表 API 已经应用了一些年，可以相信在大多数不算落伍的 Tcl 安装包中都包含了这些 API。另一方面，字典(其内部实现基于第 40 章讨论的哈希表)则要求 Tcl 8.5 或更高版本。

41.1　本章出现的函数

本章讨论的用于列表和字典操作的函数如下。

- `Tcl_Obj *Tcl_NewListObj(int objc, Tcl_Obj *CONST objv[])`
  根据指向 Tcl 对象的 *objv* 数组和 *objc* 指明的对象数目，创建一个新的列表对象。
- `int Tcl_ListObjAppendElement(Tcl_Interp *interp,`
  `Tcl_Obj *listPtr, Tcl_Obj *objPtr)`
  给出指向包含一个列表的对象的指针 *listPtr*，将 *objPtr* 添加到列表中。
- `int Tcl_ListObjAppendList(Tcl_Interp *interp,`
  `Tcl_Obj *listPtr, Tcl_Obj *elemListPtr)`
  给出指向包含一个列表(或可转换为一个列表)的对象的指针 *listPtr*，将第二个列表 *elemListPtr* 添加到 *listPtr* 当中。
- `Tcl_SetListObj(Tcl_Obj *objPtr, int objc,`
  `Tcl_Obj *CONST objv[])`
  设置列表对象 *objPtr*，令其包含 *objv* 中包含的 *objc* 元素。
- `int Tcl_ListObjGetElements(Tcl_Interp *interp,`
  `Tcl_Obj *listPtr, int *objcPtr,`
  `Tcl_Obj ***objvPtr)`
  返回列表对象中的元素数组的计数及指针。
- `int Tcl_ListObjIndex(Tcl_Interp *interp,`
  `Tcl_Obj *listPtr, int index, Tcl_Obj **objPtrPtr)`
  给出指向包含一个列表的对象的指针 *listPtr*，将对象置于 *objPtrPtr* 的 *index* 索引对应的位置。如果 *index* 越界(-1，或大于最后一个元素的索引值)，将在 *objPtrPtr* 中存放 NULL，并返回 TCL_OK。

- `int Tcl_ListObjLength(Tcl_Interp *interp,`
  `Tcl_Obj *listPtr, int *intPtr)`
  给出指向包含一个列表的对象的指针 *listPtr*，将列表的长度存储在 *intPtr* 中。
- `int Tcl_ListObjReplace(Tcl_Interp *interp,`
  `Tcl_Obj *listPtr, int first, int count, int objc,`
  `Tcl_Obj *CONST objv[])`
  删除 *listPtr* 中的 *count* 个元素，从 *first* 索引对应的元素开始，然后在它们的位置换入 Tcl_Obj 的 *objv* 数组的 *objc* 个元素，从第一个元素开始。如果 *objv* 为 NULL，则不添加新的元素。如果参数 *first* 是 0 或负值，它指向第一个元素。如果 *first* 大于或等于列表中的元素个数，则不会删除任何元素；新的元素添加到列表尾部。*count* 给定了要替换的元素的个数。如果 *count* 是零或负值，则不会删除任何元素；新的元素直接插入由 *first* 参数指定的元素之前。
- `Tcl_Obj *Tcl_NewDictObj()`
  创建一个新的空字典对象。
- `int Tcl_DictObjPut(Tcl_Interp *interp,`
  `Tcl_Obj *dictPtr, Tcl_Obj *keyPtr,`
  `Tcl_Obj *valuePtr)`
  向字典中添加一个关键字-值对，如果字典中已经存在该关键字，则更新该关键字所对应的值。
- `int Tcl_DictObjPutKeyList(Tcl_Interp *interp,`
  `Tcl_Obj *dictPtr, int keyc, const Tcl_Obj *keyv,`
  `Tcl_Obj *valuePtr)`
  向嵌套的字典中添加一个关键字-值对，如果字典中已经存在该关键字，则更新该关键字所对应的值。*keyv* 参数指定了关键字的列表(外层的关键字放在前面)，作为将要处理的关键字-值对的路径使用。当嵌套的字典不存在时，为非终止符关键字进行创建。
- `int Tcl_DictObjGet(Tcl_Interp *interp,`
  `Tcl_Obj *dictPtr, Tcl_Obj *keyPtr,`
  `Tcl_Obj **valuePtrPtr)`
  给出一个关键字，从字典中取得它的值(如果在字典中没有找到这个关键字，返回 NULL)。
- `int Tcl_DictObjRemove(Tcl_Interp *interp,`
  `Tcl_Obj *dictPtr, Tcl_Obj *keyPtr)`
  根据给定的关键字从字典中删除相应的关键字-值对；关键字不要求一定是字典中出现了的。
- `int Tcl_DictObjRemoveKeyList(Tcl_Interp *interp,`
  `Tcl_Obj *dictPtr, int keyc, const Tcl_Obj *keyv)`
  从嵌套的字典中移除一个关键字-值对。*keyv* 参数指定了关键字的列表(外层的关

键字放在前面)，作为将要处理的关键字-值对的路径使用。所有的非终止符关键字都必须存在，并且它们的值都是字典。

- `int Tcl_DictObjSize(Tcl_Interp *interp, Tcl_Obj *dictPtr, int *sizePtr)`

  对于给定的字典，将关键字-值对的数量存放到 *sizePtr* 变量中。

- `int Tcl_DictObjFirst(Tcl_Interp *interp, Tcl_Obj *dictPtr, Tcl_DictSearch *searchPtr, Tcl_Obj **keyPtrPtr, Tcl_Obj **valuePtrPtr, int *donePtr)`

  对给定的字典中的所有关键字-值对进行迭代，将关键字和值存放在由 *keyPtrPtr* 和 *valuePtrPtr* 参数指向的变量中。对了安全，遍历字典时可将字典锁定。将搜索标记存储在 *searchPtr* 指向的变量中。如果过程中还有更多的关键字-值对，则 *donePtr* 指向的变量为 0，如果已经完成遍历，则该变量为非零值。

- `void Tcl_DictObjNext(Tcl_DictSearch *searchPtr, Tcl_Obj *keyPtrPtr, Tcl_Obj **valuePtrPtr, int *donePtr)`

  给定搜索标记，取得字典中的下一个关键字-值对，将关键字和值存储在 *keyPtrPtr* 和 *valuePtrPtr* 参数所指向的变量中。如果过程中还有更多的关键字-值对，则 *donePtr* 指向的变量为 0，如果已经完成遍历，则该变量为非零值。

- `void Tcl_DictObjDone(Tcl_DictSearch *searchPtr)`

  给定搜索标记，在到达字典尾之前就终止迭代，解锁字典。如果前一条 Tcl_DictObjFirst 或 Tcl_DictObjNext 调用表明迭代已经终止，那就不必调用 Tcl_DictObjDone。使用同样的搜索标记多次调用 Tcl_DictObjDone 是安全的。

41.2　列表

列表在内部是由以位置为索引的 C 数组实现的(而不是链表或 Tcl 数组)，因此对它们的操作通常都十分快速。列表 C 程序 API 支持 Tcl 列表命令能完成的那些基本操作：创建列表，向列表中添加元素，联合列表，设置和替换列表元素，以及获取列表的长度。

和其他对象类型一样，列表也有一个创建新列表的函数，Tcl_NewListObj，用于创建一个空的 Tcl 对象，示例如下：

```
Tcl_Obj *listobj, strobj, intobj;
listobj = Tcl_NewListObj(0, NULL);
strobj = Tcl_NewStringObj("Answer", -1);
intobj = Tcl_NewIntObj(42);
Tcl_ListObjAppendElement(interp, listobj, strobj);
Tcl_ListObjAppendElement(interp, listobj, intobj);
```

这个示例中，我们首先创建了一个空的列表。然后我们使用 Tcl_ListObjAppendElement 函数为它添加了两个元素。不必去增加 intobj 和 strobj 对象的引

用计数，Tcl_ListObjAppendElement 会处理这件事，因为对象对它们的引用至少有一个是与列表关联在一起的。您还可以使用 Tcl_ListObjAppendList 函数获取一个列表对象，用另一个列表对象中的元素来扩展它。

30.4 节讨论了带标记的列表的设计原理与使用，例如 max 105 min 89 average 96，作为在 Tcl 和 C 之间传递信息的方法。这样的列表是很有用的，它自己保持了对数据的描述。而且，它还可以被传给 array set 命令，从而创建一个数组。

```
array set temps {max 105 min 89 average 96}
```

在 C 程序中使用 Tcl_ListObjAppendElement 构建这个数组是很容易的，然后就像第一个示例那样，可以用它一片一片地建起整个列表。另一种方法是使用 Tcl_Obj 指针的数组，每一个 Tcl_Obj 都是结果列表中的一个元素，然后将数组和对象数目一起传给 Tcl_NewListObj。

```
Tcl_Obj *templist;
Tcl_Obj *objs[6];
min = 89;
max = 105;
average = 96;

objs[0] = Tcl_NewStringObj("max", -1);
objs[1] = Tcl_NewIntObj(max);
objs[2] = Tcl_NewStringObj("min", -1);
objs[3] = Tcl_NewIntObj(min);
objs[4] = Tcl_NewStringObj("average", -1);
objs[5] = Tcl_NewIntObj(average);
templist = Tcl_NewListObj(6, objs);
Tcl_IncrRefCount(templist);
```

还可以对一个起初并非作为列表对象创建的对象进行列表操作。只要对象的字符串表达形式是一个形式完好的 Tcl 列表，访问函数就可以将对象进行转换，以便列表函数使用。如果对象没有有效的列表结构，访问函数会返回 TCL_ERROR。例如，下面这段代码首先构建了一个基于字符串的 Tcl_Obj。接下来的列表操作就会将该对象转换为列表的表现形式。

```
int objnum = 0;
int listlength = 0;
Tcl_Obj *element;
Tcl_Obj *mylist = Tcl_NewStringObj("a b c d", -1);
Tcl_Obj **objs;

Tcl_ListObjIndex(interp, mylist, 1, &element);
Tcl_ListObjGetElements(interp, mylist, &objnum, &objs);
Tcl_ListObjLength(interp, mylist, &listlength);
```

这里的 Tcl_ListObjIndex 返回了一个对象，表示的是列表中的第二个元素，这里就是 b，然后将结果放置在 element 中。Tcl_ListObjGetElements 取得所有的列表元素，形成数组 Tcl_Obj，然后将指向数组的指针存放在 objs 中，将元素的个数存放在 objnum 中。指向的 Tcl_Obj 数组由 Tcl 进行管理，不应该由其他调用者释放或进行写操作。最后，将列表元素的个数存放到变量 listlength 中。

Tcl_ListObjReplace 函数可以完成元素的插入和删除，它是很多列表操作的基础。它的参数依次是 Tcl 解释器、指向要处理的列表对象的指针、要删除的第一个元素的索引、要删除的元素的个数、要在它们的位置插入的元素的个数和作为插入元素的 Tcl_Objs 数组。从列表中删除任何元素导致对应的 Tcl_Obj 计数的减小都会被自动处理，对列表添加元素导致对应的 Tcl_Obj 计数的增大也会被自动处理。

例如，下面的替换操作将索引为 3 的元素替换为一个新的元素。

```
Tcl_Obj *newObj = Tcl_NewStringObj("Carol", -1);
Tcl_ListObjReplace(interp, mylist, 2, 1, 1, &newObj);
```

而下面这个示例向列表的开头插入一个新的元素。

```
Tcl_Obj *newObj = Tcl_NewStringObj("Dean", -1);
Tcl_ListObjReplace(interp, mylist, 0, 0, 1, &newObj);
```

自然地，列表也可以包含其他的列表对象，这使得我们可以在列表中嵌套地使用列表，或在列表中使用字典，或在字典中使用列表，等等。然而，不要构建过于复杂的结构，太复杂的情况不利于其他人理解它是如何工作的，甚至在一段时间以后您自己也难以搞清楚它是如何工作的——如果您有了一个字典的列表的列表，也许就应该重新考虑代码了！

41.3　字典

我们在 Tcl 中已经见过字典，就是从值到独一无二的关键字的映射。从脚本的观点看，值可以是任意的字符串，这意味着值可以被解析为列表，也可以被解析为嵌套的字典。字典的内部实现基于 Tcl 的哈希表，前面第 40 章已经对此进行了讨论，而 Tcl 的 C API 提供了一系列函数，可以很容易地将字典作为对象处理。

您可以使用 Tcl_NewDictObj 函数创建一个新的空字典。然后可以使用 Tcl_DictObjPut 函数对字典添加新的关键字-值对。例如，前文讲述的有形如“max 105 min 89 average 96”格式的“标记列表”。显然，这是一种字典格式，我们可以这样直接用它创建字典。

```
Tcl_Obj *dictObj;
Tcl_Obj *objs[6];
int i;
min = 89;
max = 105;
average = 96;

objs[0] = Tcl_NewStringObj("max", -1);
objs[1] = Tcl_NewIntObj(max);
objs[2] = Tcl_NewStringObj("min", -1);
objs[3] = Tcl_NewIntObj(min);
objs[4] = Tcl_NewStringObj("average", -1);
objs[5] = Tcl_NewIntObj(average);
dictObj = Tcl_NewDictObj();
for (i=0 ; i<6 ; i+=2) {
    Tcl_DictObjPut(interp, dictObj, objs[i], objs[i+1]);
}
```

如果证明有必要向字典中存入关键字-值对(即它们不是已经存在于字典中)，则函数Tcl_DictObjPut自动增加与关键字和值对应的计数。

提示： *在将字典对象传给 Tcl_DictObjPut 或任何其他修改字典的函数时，字典不能被共享；如果这时它被共享，则会触发 Tcl_Panic。*

Tcl_DictObjGet 函数取得与指定关键字对应的值。例如下面这段代码，访问刚才创建的字典，取得关键字 min 对应的值对象的指针，将它存放在变量 value 中。

```
Tcl_Obj *key, *value;
result int;
key = Tcl_NewStringObj("min", -1);
result = Tcl_DictObjGet(interp, dictPtr, key, &value);
```

如果该关键字不存在，则存放在 value 中的值为 NULL。如果 dictPtr 指向的对象不能被转换为一个字典，则 Tcl_DictObjGet 函数会返回 TCL_ERROR。

如果将一个关键字映射到字符串值，而这个值具有嵌套字典的格式时，那个值不会被自动地作为字典对待。例如，虽然下面这段代码中演示的值具有字典的字符串格式，它仍然只被作为一个字符串处理。

```
Tcl_Obj *key = Tcl_NewStringObj("NAME", -1);
Tcl_Obj *value = Tcl_NewStringObj(
                    "{FIRST Dean LAST Akamine}", -1);
Tcl_Obj *dictObj = Tcl_NewDictObj();
Tcl_DictObjPut(interp, dictObj, key,value);
```

然而，如果对象被字典函数访问，则 Tcl 会自动把对象转换为字典表现形式。另一方面，如果知道在操作一个嵌套的字典结构，就可以使用 Tcl_DictObjPutKeyList 函数，传入关键字的数组。例如下面对 Tcl 的 dict set 命令函数的实现，允许用户指定任意多个嵌套的关键字作为参数，然后给出映射到终止关键字的值。

```
static int
DictSetCmd(
    ClientData dummy,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const *objv)
{
    Tcl_Obj *dictPtr, *resultPtr;
    int result, allocatedDict = 0;

    if (objc < 4) {
        Tcl_WrongNumArgs(interp, 1, objv,
                         "varName key ?key ... ? value");
        return TCL_ERROR;
    }

    dictPtr = Tcl_ObjGetVar2(interp, objv[1], NULL, 0);
    if (dictPtr == NULL) {
        allocatedDict = 1;
        dictPtr = Tcl_NewDictObj();
    } else if (Tcl_IsShared(dictPtr)) {
        allocatedDict = 1;
        dictPtr = Tcl_DuplicatedObj(dictPtr);
    }
```

```
    result = Tcl_DictObjPutKeyList(interp, dictPtr, objc-3,
                                   objv+2, objv[objc-1]);
    if (result != TCL_OK) {
        if (allocatedDict) {
            TclDecrRefCount(dictPtr);
        }
        return TCL_ERROR;
    }

    resultPtr = TCL_ObjSetVar2(interp, objv[1], NULL,
                               dictPtr, TCL_LEAVE_ERR_MSG);
    if (resultPtr == NULL) {
        return TCL_ERROR;
    }
    Tcl_SetObjResult(interp, resultPtr);
    return TCL_OK;
}
```

提示： 留心函数是如何检查字典对象是否被共享的，如果被共享了，则在修改它之前先复制。否则，如果共享了字典对象，对 Tcl_DictObjPutKeyList 的调用就会引起 Tcl_Panic。

Tcl_DictObjSize 函数用于提供字典的大小(即关键字-值映射对的数目)。您可以使用 Tcl_DictObjRemove 函数从字典中移除关键字-值对。这个函数会自动减小与字典中的关键字和值对应的计数。如果指定的关键字并不存在，也不会导致错误。为了从嵌套的字典结构中移除关键字-值对，可以调用 Tcl_DictObjRemoveKeyList 函数，它接受关键字的数组，与 Tcl_DictObjPutKeyList 相似。

最后，您可以遍历一个字典的关键字-值对。Tcl_DictObjFirst 函数启动对指定字典的遍历，返回第一个关键字-值对。然后对 Tcl_DictObjNext 的后续调用则迭代这些关键字-值对。这些函数都有一个终止指示器，让您知道是否已经完成迭代。如果希望在到达字典尾之前就结束搜索，必须调用 Tcl_DictObjDone 来清理搜索状态；如果 Tcl_DictObjNext 本身已经遇到了字典尾，就不必调用 Tcl_DictObjDone，不过如果调用了也不会出错。每一个对字典的遍历都有对应的标记，该标记必须传给这些函数。

提示： 如果字典的值在迭代期间被改动，无论是由 Tcl_DictObj 函数调用引起的，还是由将对象转换为非字典的表达形式的函数引起的，迭代都会被终止。下一次对 Tcl_DictObjNext 的调用会显示它已经到达迭代的结束点。

作为使用字典迭代的一个示例，下面的代码创建了一个列表，该列表由来自一个字典的所有值组成。

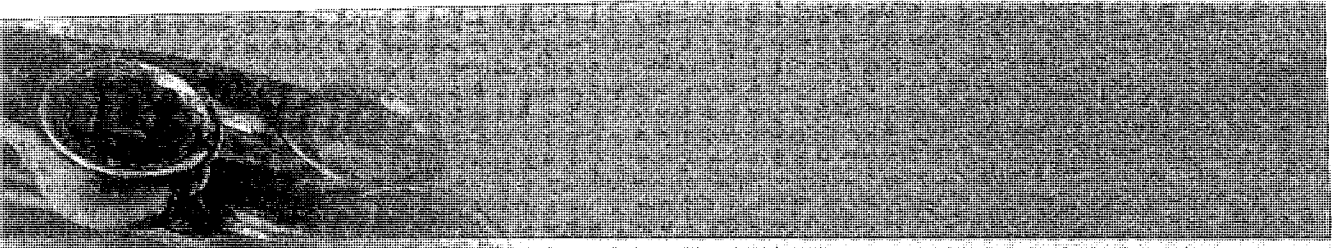
```
Tcl_DictSearch search;
Tcl_Obj *key, *value, *valueList;
int done;

/*
 *Assume interp and objPtr are parameters.
 */
if (Tcl_DictObjFirst(interp, objPtr, &search,
        &key, &value, &done) != TCL_OK) {
```

```
    return TCL_ERROR;
}

valueList = Tcl_NewListObj(0, NULL);

for (; !done ; Tcl_DictObjNext(&search, &key, &value, &done)) {
        Tcl_ListObjAppendElement(interp, valueList, value);
}
Tcl_DictObjDone(&search);
```



第42章 通 道

基于通道的概念，Tcl 提供了具有适应性的输入输出管理系统。在 Tcl 脚本层次，我们能够经对文件和 TCP 套接字通道一视同仁了。它们是 Tcl 本身提供的两种通道类型。通道系统是为了提供灵活的、可扩展的、跨平台的、与设备无关的管理输入输出信息的方法。通道设计需要两个层次的方法：Tcl，或 C 的高级方法。所有的通道的使用方法或多或少都是相同的。像 puts、read、chan flush 等标准操作对所有通道都是一样的，无论实现它们的是什么样的低层设备(文件、TCP 套接字、真实的硬件设备等)。实现指定的通道类型的低层驱动，创建了通用层与“设备”之间的桥梁。这种设计意味着您可以创建一种新的通道类型，其行为与内建通道完全一样。

本章解释如何在 C 中使用通道，以及如何为一个设备编写新的通道类型。

42.1 本章出现的函数

Tcl 提供了很多用于通道交互的函数。这些函数中的大部分都获取一个 Tcl_Channel 作为参数，或返回一个这种类型的变量作为结果。

42.1.1 基本通道操作

下面的函数实现基本通道操作，如打开、关闭、读取、写入等。

- `Tcl_Channel Tcl_FSOpenFileChannel(Tcl_Interp *interp*, Tcl_Obj *pathPtr, const char *modeString, int permissions)`
  打开由 *pathPtr* 指定的文件，返回通道句柄。所有参数的语法和含义与用 Tcl open 命令打开文件时的参数类似。(取代了以前的基于字符串的 Tcl_OpenFileChannel 函数。)
- `Tcl_Channel Tcl_GetStdChannel(int type)`
  返回一个标准 I/O 通道的通道句柄，这里 *type* 是 TCL_STDIN、TCL_STDOUT 或 TCL_STDERR。
- `Tcl_SetStdChannel(Tcl_Channel channel, int type)`
  将一个已经存在的通道设置为标准 I/O 通道来使用，这里 *type* 是 TCL_STDIN、TCL_STDOUT 或 TCL_STDERR。

- `int Tcl_Close(Tcl_Interp *interp, Tcl_Channel channel)`
  销毁通道。在销毁之前，缓冲区的中输出内容写入输出设备，缓冲的输入内容都被放弃。调用 Tcl_Close 时通道不应该在任何一个解释器中注册；否则应该调用 Tcl_UnregisterChannel。
- `int Tcl_ReadChars(Tcl_Channel channel,`
  `Tcl_Obj *readObjPtr, int charsToRead,`
  `int appendFlag)`
  从 *channel* 中按字节取走信息，根据通道的编码将它们转化为 UTF-8 格式，以字符串的表现形式把生成的数据存放在 *readObjPtr* 中。返回值是字符的个数，最多有 *charsToRead* 个，存放在 *readObjPtr* 中。如果在读取中遇到错误，返回值为-1。(取代了以前的 Tcl_Read 函数，那个函数不支持编码翻译。)
- `int Tcl_ReadRaw(Tcl_Channel channel, char *readBuf,`
  `int bytesToRead)`
  与 Tcl_ReadChars 类似，只不过是由堆叠通道应用程序的转换通道驱动使用。从通道中读取少于 *bytesToRead* 个字节的信息，将它们复制到 *readBuf*，而不作编码翻译及其他任何改动。
- `int Tcl_GetsObj(Tcl_Channel channel, Tcl_Obj *lineObjPtr)`
  `int Tcl_Gets(Tcl_Channel channel, Tcl_DString *lineRead)`
  Tcl_GetsObj 从 *channel* 中按字节取走信息，根据通道的编码将它们转化为 UTF-8 格式，直到读取了输入的完整的一行。这一行中除了行终止符之外的所有字符都添加到字符串表现形式的 *lineObjPtr* 当中；行终止符也会被读取，然后抛弃。返回存放到 *lineObjPtr* 中的字符个数，如果遇到错误或已经到达文件尾，返回-1；在非阻塞式的通道中没有完整的一行数据可用时，也会返回-1，不取走任何数据。Tcl_Gets 与 Tcl_GetsObj 大致相同，只不过其结果字符添加到由 *lineRead* 给定的动态字符串中，而非 Tcl 对象中。
- `int Tcl_Ungets(Tcl_Channel channel, const char *input,`
  `int inputLen, int addAtEnd)`
  从 *input* 中把数据添加到通道的输入队列中；*inputLen* 给定了要添加的字节数。非零的 *addAtEnd* 值表明数据应该添加到队列的尾部；否则就添加到队列的头部。Tcl_Ungets 返回 *inputLen*，如果遇到错误，则返回-1。
- `int Tcl_WriteObj(Tcl_Channel channel,`
  `Tcl_Obj *writeObjPtr)`
  `int Tcl_WriteChars(Tcl_Channel channel,`
  `const char *charBuf, int bytesToWrite)`
  Tcl_WriteObj 将字符串表现形式的 *writeObjPtr* 写入通道中，将字符转换为通道的字符编码。Tcl_WriteChars 进行同样的工作，只不过 *charBuf* 包含了那些 UTF-8 字符；*bytesToWrite* 指定了要输出的字节数，如果是-1，则表示 *charBuf* 是一个非终止的字符串，应该输出其中的所有字符。(Tcl_WriteChars 替换了以前的 Tcl_Write

函数，那个函数不支持编码翻译。)返回写入通道的字节数，如果遇到错误，则返回-1。

- int Tcl_WriteRaw(Tcl_Channel *channel*, const char **byteBuf*,
  int *bytesToWrite*)
  与 Tcl_WriteChars 相似，只不过是由堆叠通道应用程序的转换通道驱动使用。向通道写入 *bytesToWrite* 个字节，不作编码翻译及其他任何改动。
- int Tcl_Eof(Tcl_Channel *channel*)
  如果 *channel* 在上一次输入操作中遇到了文件尾，则返回非零值。
- int Tcl_Flush(Tcl_Channel *channel*)
  将 *channel* 的输出缓冲区中的所有数据尽快写入它连接到的文件或设备。
- int Tcl_InputBlocked(Tcl_Channel *channel*)
  如果 *channel* 是非阻塞模式的通道，而且上一次输入操作因为可用数据不足返回的数据比需要的少，则返回非零值。如果通道是阻塞模式的，调用这个函数总是返回 0。
- int Tcl_InputBuffered(Tcl_Channel *channel*)
  返回通道的内部缓冲区中当前存放的输入信息的字节数。如果通道不是为读取打开的，则该函数总是返回 0。
- int Tcl_OutputBuffered(Tcl_Channel *channel*)
  返回通道的内部缓冲区中当前存放的输出信息的字节数。如果通道不是为写出打开的，则该函数总是返回 0。
- Tcl_WideInt Tcl_Seek(Tcl_Channel *channel*,
  Tcl_WideInt *offset*, int *seekMode*)
  移动 *channel* 中的访问点，以后的读写操作将在新的访问点进行。要求的访问点由 *offset* 偏移变量设置，偏移量是相对 *seekMode* 而言的，*seekMode* 应该是 SEEK_SET(起始)、SEEK_CUR(当前)或 SEEK_END(结束)。在进行访问点移动之前，缓冲中的输出内容立即写出，缓冲中的输入内容被抛弃。返回新的访问点，如果遇到错误，则返回-1。
- Tcl_WideInt Tcl_Tell(Tcl_Channel *channel*)
  返回通道的当前访问点，如果该通道不支持访问点选取操作，则返回-1。
- int Tcl_TruncateChannel(Tcl_Channel *channel*,
  Tcl_WideInt *length*)
  将 *channel* 连接到的文件截短为 *length* 个字节。
- int Tcl_GetChannelOption(Tcl_Interp **interp*,
  Tcl_Channel *channel*, const char **optionName*,
  Tcl_DString **optionValue*)
  取得名为 *optionName* 的通道选项值，将结果存放在 *optionValue* 中。如果 *optionName* 是 NULL，则函数将所有的通道选项名称及其值组成列表，存放在 *optionValue* 中。这里 *interp* 可以是 NULL。

- `int Tcl_SetChannelOption(Tcl_Interp *interp,`
  `Tcl_Channel channel, const char *optionName,`
  `const char *newValue)`

  将 *optionName* 指定的通道选项的值设置为 *newValue*。这个过程在正常情况下返回 TCL_OK。如果遇到错误，则返回 TCL_ERROR；另外，如果 *interp* 非空，则 Tcl_SetChannelOption 将一个错误消息留在解释器的结果中。

42.1.2 通道注册函数

在解释器和线程中注册通道的管理函数如下。

- `void Tcl_RegisterChannel(Tcl_Interp *interp,`
  `Tcl_Channel channel)`

  将通道加入 *interp* 的访问通道集中。在调用之后，在该解释器中运行的 Tcl 程序可以使用 Tcl_CreateChannel 调用时给出的名称来指定进行输入输出操作的通道。*interp* 参数可以是 NULL，表示为通道添加一个独立于所有解释器的引用。
- `int Tcl_UnregisterChannel(Tcl_Interp *interp,`
  `Tcl_Channel channel)`

  `int Tcl_DetachChannel(Tcl_Interp *interp,`
  `Tcl_Channel channel)`

  从 *interp* 的访问通道集中移除通道。在调用之后，在该解释器中运行的 Tcl 程序不再可以使用通道的名称来指定通道。*interp* 参数可以是 NULL，表示为通道移除一个独立于所有解释器的引用。使用 Tcl_UnregisterChannel 时，如果该操作之后通道不在任何一个解释器中有注册，则通道会被关闭，然后销毁。
- `int Tcl_IsChannelShared(Tcl_Channel channel)`

  如果通道被多个解释器共享，返回 1，否则返回 0。
- `int Tcl_IsChannelRegistered(Tcl_Interp *interp,`
  `Tcl_Channel channel)`

  `void Tcl_CutChannel(Tcl_Channel channel)`

  从属于当前线程的所有通道的列表中移除指定的通道(如果有使用线程的话)。这个操作可能不会对注册在解释器中的通道生效。
- `void Tcl_SpliceChannel(Tcl_Channel channel)`

  将指定通道添加到属于当前线程的所有通道的列表中(如果有使用线程的话)。这个操作可能不会对注册在解释器中的通道生效。此前该通道必须由 Tcl_CutThread 命令从线程中切分出来。

42.1.3 通道属性函数

下面这些函数用于查询及设置指定通道的属性。

- `int Tcl_IsChannelExisting(CONST char *channelName)`

如果指定名称的通道存在，则返回 1，否则返回 0。

- `int Tcl_IsStandardChannel(Tcl_Channel channel)`
  如果指定的通道是三个标准通道之一——stdin、stdout 或 stderr——则返回 1，否则返回 0。
- `ClientData Tcl_GetChannelInstanceData(Tcl_Channel channel)`
  取得给定通道的实例数据。
- `Tcl_ChannelType *Tcl_GetChannelType(Tcl_Channel channel)`
  取得指向该通道的类型的指针。
- `CONST char *Tcl_GetChannelName(Tcl_Channel channel)`
  返回给定的通道的名称。
- `int Tcl_GetChannelHandle(Tcl_Channel channel,`
  `int directioin, ClientData handlePtr)`
  将系统特定的设备句柄(如 FILE *)与一个通道和 *handlePtr* 中的方向(TCL_READABLE 或 TCL_WRITABLE)关联起来。如果没有指定的句柄，则返回 TCL_ERROR。
- `Tcl_ThreadId Tcl_GetChannelThread(Tcl_Channel channel)`
  返回当前正在管理给定通道的线程的 ID。
- `int Tcl_GetChannelMode(Tcl_Channel channel)`
  返回对 TCL_READABLE 和 TCL_WRITABLE 进行或运算的结果。
- `int Tcl_GetChannelBufferSize(Tcl_Channel channel)`
  返回通道的缓冲区大小，单位为字节。
- `Tcl_SetChannelBufferSize(Tcl_Channel channel, int size)`
  设置通道的缓冲区大小，单位为字节。
- `int Tcl_IsChannelRegistered(Tcl_Interp interp,`
  `Tcl_Channel channel)`
  如果 *channel* 已经在 *interp* 解释器中注册，则返回 1，否则返回 0。

42.1.4　通道查询函数

下面的函数用于取得解释器中通道的有关信息。

- `Tcl_Channel Tcl_GetChannel(Tcl_Interp interp,`
  `const char *channelName, int *modePtr)`
  给出 Tcl 中的通道句柄名，返回相应的 Tcl_Channel，返回 *modePtr*，这是对 TCL_READABLE 和 TCL_WRITABLE 进行或运算得到的整型值。
- `int Tcl_GetChannelNames(Tcl_Interp interp)`
  `int Tcl_GetChannelNamesEx(Tcl_Interp interp,`
  `const char *pattern)`
  将注册的通道的名称作为列表对象写入解释器的结果。Tcl_GetChannelNamesEx 筛选与 *pattern* 匹配的名称，遵循 string match 的匹配规定。

42.1.5 通道类型定义函数

与创建和使用用户自定义的通道类型有关的函数如下。

- `Tcl_Channel Tcl_CreateChannel(`
  `Tcl_ChannelType *typePtr, CONST char *channelName,`
  `ClientData instanceData, int mask)`
  创建一个新的通道，类型为 *typePtr*，名称为 *channelName*。
- `Tcl_Channel Tcl_StackChannel( Tcl_Interp interp,`
  `Tcl_ChannelType *typePtr, ClientData instanceData,`
  `int mask, Tcl_Channel channel)`
  在已经存在的 *channel* 上以 Tcl_RegisterChannel 为 *channel* 注册的名称堆叠一个新的通道。*mask* 参量指定了新通道所允许进行的操作。这是原通道所允许进行的操作的一个子集。其他选项与 Tcl_CreateChannel 的选项相同。
- `int Tcl_UnstackChannel (Tcl_Interp interp,`
  `Tcl_Channel channel)`
  反转堆叠通道的过程。将通道名称关联到旧的通道，由 Tcl_StackChannel 添加的过程模块会被销毁。如果没有旧的通道存在，Tcl_UnstackChannel 的效果与 Tcl_Close 相同。
- `Tcl_Channel Tcl_GetStackedChannel(Tcl_Channel channel)`
  返回堆叠通道中位于 *channel* 下面的那个通道。
- `Tcl_Channel Tcl_GetTopChannel(Tcl_Channel channel)`
  返回 *channel* 参数是其中之一的堆叠通道的最顶层通道。
- `int Tcl_BadChannelOption(Tcl_Interp *interp,`
  `CONST char *optionName, CONST char *optionList)`
  由通道选项获取/设置函数调用，反映 *optionName* 不是该通道的有效选项。
- `Tcl_NotifyChannel(Tcl_Channel channel, int mask)`
  调用通道驱动，向通用层反映由 *mask*(TCL_READABLE、TCL_WRITABLE 和 TCL_EXCEPTION 的或操作的结果)指定的事件已经出现。
- `void Tcl_ClearChannelHandlers(Tcl_Channel channel)`
  移除与指定的通道相关的所有通道处理器与事件脚本。
- `int Tcl_ChannelBuffered(Tcl_Channel channel)`
  返回通道的输入缓冲区中当前数据的长度，以字节为单位。

42.2 通道操作

Tcl 提供了丰富的通道操作，如 42.1.1 节所述，用于与通道进行交互。这些函数中的大多数都获取一个 Tcl_Channel 结构体作为参数或返回一个该类型的结果。下面这段代码

展示了 filetovar 命令的命令函数实现。

```
static int
FileToVarCmd(ClientData clientData, Tcl_Interp *interp,
             int objc, Tcl_Obj *CONST objv[]) {
    Tcl_Channel chan;
    Tcl_Obj *buffer;
    if (objc != 3 ) {
        Tcl_WrongNumArgs(interp, 1, objv, "file varname");
        return TCL_ERROR;
    }

    chan = Tcl_FSOpenFileChannel(interp, objv[1], "r", 0);
    if (chan == NULL) {
        return TCL_ERROR;
    }

    buffer = Tcl_NewObj();
    if(Tcl_ReadChars(chan, buffer, -1, 0) <0) {
        Tcl_Close(interp,chan);
        return TCL_ERROR;
    }
    Tcl_Close(interp, chan);

    Tcl_SetVar2Ex(interp, Tcl_GetString(objv[2]),
                  NULL, buffer, 0);

    return TCL_OK;
}
```

运行下列脚本

```
filetovar /tmp/somefile content
```

的效果与运行以下脚本相同。

```
set fid [open /tmp/somefile]
set content [read $fid]
close $fid
```

命令过程需要获取两个参数，一个是要读取的文件的名称，另一个是要存放文件内容的变量。Tcl_FSOpenFileChannel 函数获取一个 Tcl 对象(包括了要打开的文件的名称)和一个模式字符串(它的解析方式与 Tcl 的 Open 命令相同)。它返回一个 Tcl_Channel 对象，这个对象可以传给其他的通道 I/O 命令。

读取操作本身是由 Tcl_ReadChars 命令处理的，该命令需要获取的参数包括一个通道，一个用于保存数据的 Tcl 对象，要读取的数据的长度以及一个表示数据是否应该添加的标志。这里长度参数是-1，它告诉 Tcl_ReadChars 读取整个文件。函数与 Tcl read 命令相似。然后 Tcl_Close 关闭了文件，文件的内容由 Tcl_SetVar2Ex 设为变量的值。

如您所见，这个示例中使用的大多数通道函数都与 Tcl 脚本级的命令相似。表 42.1 给出了大致相当的 Tcl I/O 命令和 CF 函数。命令 Tcl_GetChannelOption 和 Tcl_SetChannelOption 特别常用，分别用于获取和设置通道选项。下面的代码为一个通道设置了缓冲区大小。

```
Tcl_SetChannelOption(interp, chan, "-buffersize", "10000");
```

表 42.1 Tcl 通道命令和类似的 C 函数

Tcl 命令	类似的 C 函数
open	对文件使用的 Tcl_FSOpenFileChannel
exec,"open |"	Tcl_OpenCommandChannel(第 45 章将进一步讨论)
close	Tcl_Close
puts	Tcl_WriteObj, Tcl_WriteChars, Tcl_WriteRaw
read	Tcl_ReadChars, Tcl_ReadRaw
gets	Tcl_Gets, Tcl_GetsObj
chan eof	Tcl_Eof
chan blocked	Tcl_InputBlocked
chan flush	Tcl_Flush
chan seek	Tcl_Seek
chan tell	Tcl_Tell
chan truncate	Tcl_TruncateChannel
chan configure	Tcl_GetChannelOption, Tcl_SetChannelOption
chan pending	Tcl_InputBuffered, Tcl_OutputBuffered

下面这个示例取得套接字通道的地址、主机名以及端口号，将其结果存放在 DString 中。

```
Tcl_DString optionValue;
Tcl_DStringInit(optionValue);
Tcl_GetChannelOption(interp, chan, "-sockname", &optionValue);
```

还有其他一些用于查询通道属性的函数，详见 42.1.3 节。

42.3 注册通道

如果在 Tcl 中创建了通道(例如，使用了 open)，返回的通道描述符用于在脚本层级上对通道进行操作。另一方面，如果使用 Tcl 的 C API 创建通道，那么创建的通道并不会自动地呈现到脚本层级。您必须在 C 语言层级对 Tcl_Channel 进行通道操作。

如果想把通道呈现到脚本层级，必须调用 Tcl_RegisterChannel 函数来在解释器中注册它，然后就可以在该解释器中访问该通道。您可能把通道注册到任意多个解释器中，只要这些解释器处在同一个线程中。

下面的代码实现了一个 randomfile 包。它打开 Unix /dev/random 文件作为随机数据的源，将通道编码设置为二进制编码，然后把通道注册到 Tcl 解释器。然后它取得创建的通道的名称，将它赋给变量 random。

```
#include <tcl.h>
```

```
int Randomfile_Init(Tcl_Interp *interp) {
    Tcl_Channel chan;
    Tcl_Obj *filename;
#ifdef USE_TCL_STUBS
    if (Tcl_InitStubs(interp, "8", 0) == NULL) {
        return TCL_ERROR;
    }
#endif

    filename = Tcl_NewStringObj("/dev/random" , -1);
    Tcl_IncrRefCount(filename);
    chan = Tcl_FSOpenFileChannel(interp, filename, "r", 0);
    Tcl_DecrRefCount(filename);

    if (chan == NULL) {
        return TCL_ERROR;
    }
    Tcl_SetChannelOption(interp, chan, "-encoding",
                         "binary");
    Tcl_RegisterChannel(interp, chan);
    Tcl_SetVar(interp, "random",
               Tcl_GetChannelName(chan), 0);

    if (Tcl_PkgProvide(interp,
                   "randomfile",
                   "0.1") != TCL_OK) {
        return TCL_ERROR;
    }
    return TCL_OK;
}
```

一旦编译了这段代码，将它加载到一个 Tcl 脚本，就可以使用这样的命令来读取 10 个随机的字节。

```
read $random 10
```

一旦为 Tcl 注册了一个通道，就不应该在 C 语言层级用 Tcl_Close 来关闭它。从 Tcl 解释器中移除通道的正确方法是使用 Tcl_UnregisterChannel 函数，它使得通道对指定的解释器不可见，而且，如果此操作后就不再有对该通道的引用，则将其关闭。与此不同，Tcl_DetachChannel 从解释器中移除通道，但不会试图关闭它。

虽然可以在多个解释器中共享通道，但是所有的这些解释器必须处在同一个线程中。您不能在多个线程之间共享通道。不过，可以把通道从一个线程中移到另一个线程中。被移动的通道必须未被注册到任何解释器；如果需要，则调用 Tcl_DetachChannel。然后使用 Tcl_CutChannel 在当前线程中剪切通道。在通道不再与某个线程相关联后，可以在某个线程中调用 Tcl_SpliceChannel 让该线程与通道关联起来。有关 Tcl 线程编程的更多信息参见第 46 章。

即使是在脚本层级建立的通道，仍然可以获得它的 Tcl_Channel 句柄。给出通道的名称，例如 file1，函数 Tcl_GetChannel 就返回它的 Tcl_Channel。

```
Tcl_Channel chan;
int mode = 0;
chan = Tcl_GetChannel(interp, "file1", &mode);
```

42.4 标准通道

您可能对 Tcl 标准通道的使用已经很熟悉了，它们是 stdin、stdout 和 stderr(参见第 11 章)。在 C 代码中也可以对它们进行操作，当想把一个标准通道替换为自己创建的通道时，这就很有用了。

Tcl_GetStdChannel 函数返回一个标准 I/O 通道的通道句柄。它唯一的参数应该是 TCL_STDIN、TCL_STDOUT 或 TCL_STDERR。使用 Tcl_SetStdChannel 函数，可以把一个已经存在的通道作为一个标准 I/O 通道来使用。

```
Tcl_SetStdChannel(myChan, TCL_STDOUT);
Tcl_RegisterChannel(NULL, myChan);
```

提示： 当前版本的 Tcl 要求在 Tcl_SetStdChannel 之后调用 Tcl_RegisterChannel，更多信息详见参考文档。

作为使用这一功能的示例，Apache Rivet 创建了一个特殊的 Apache 通道，使用 Apache 网页服务器的 API 通过服务器将数据传送给浏览者。Rivet 将这一通道设置为标准输出通道，因此普通的 puts 命令，例如 puts "Hello, World!"，就被重定向到 Apache 通道而非普通的标准输出通道。这意味着普通的 Tcl 脚本可以不加修改地在 Apache 中运行。

提示： 如果某个标准通道被设置为 NULL，这可能是在调用 Tcl_SetStdChannel 时传入了 NULL 通道参数，也可能是因为对该通道调用了 Tcl_Close。那么下一次调用 Tcl_CreateChannel 命令时创建的通道会被自动设置为标准通道。如果多于一个标准通道为 NULL，新创建的通道依次被自动设置为标准输入、标准输出和标准错误。

42.5 创建新的通道类型

前面已经讲到，Tcl 为新的通道类型的创建提供了完整的 API。UDP 套接字就是实现新的通道类型的扩展示例；memchan 内存通道的实现用于向内存进行读写，就如同它是一个文件；Rivet 的 Apache 通道，使用了 Apache 服务器的 C API 将数据发送给浏览器。

创建新的驱动类型的中心概念是函数的实现，由函数来完成驱动所需要完成的不同任务，例如读、写、转存等。表示一个通道类型的基本结构体是 Tcl_ChannelType。

```
typedef struct Tcl_ChannelType {
    char *typeName;
    Tcl_ChannelTypeVersion version;
    Tcl_DriverCloseProc *closeProc;
    Tcl_DriverInputProc *inputProc;
    Tcl_DriverOutputProc *outputProc;
    Tcl_DriverSeekProc *seekProc;
    Tcl_DriverSetOptionProc *setOptionProc;
    Tcl_DriverGetOptionProc *getOptionProc;
    Tcl_DriverWatchProc *watchProc;
    Tcl_DriverGetHandleProc *getHandleProc;
```

```
    Tcl_DriverClose2Proc *close2Proc;
    Tcl_DriverBlockModeProc *blockModeProc;
    Tcl_DriverFlushProc *flushProc;
    Tcl_DriverHandlerProc *handlerProc;
    Tcl_DriverWideSeekProc *wideSeekProc;
} Tcl_ChannelType;
```

每一个条目都是指向一个函数的指针，该函数完成一个特定的操作。您不需要为每个通道操作都提供一个函数；如果不需要，可以把一些项设置为 NULL，对于那些对这个通道没有意义的操作，也可以将对应函数实现为在调用时返回 EINVAL。函数定义、函数的用途以及需要哪些函数定义等详细信息见参考文档。在 42.5.3 节中我们会看到一个自定义通道类型的示例。

42.5.1　创建自定义通道实例

在实现了通道类型，定义了它的通道操作函数之后，就可以用 Tcl_CreateChannel 函数来创建该类型的一个通道实例。

```
Tcl_Channel Tcl_CreateChannel(
        Tcl_ChannelType *typePtr, CONST char *channelName,
        ClientData instanceData, int mask)
```

typePtr 是指向前面定义的 Tcl_ChannelType 结构体的指针。*channelName* 使用的名称必须是独一无二的(如 file1、file2、socket7、socket8 等)。变量 *instanceData* 用于在实现通道时传递一些实例特有的数据。最后，*mask* 指导 Tcl 对该通道允许哪些操作，使用的标志如下。

- TCL_WRITABLE——允许进行写操作。
- TCL_READABLE——允许进行读操作。

一旦通过创建指定类型的通道的方法将该通道类型注册到 Tcl 中，那个通道类型就不能被移除。

42.5.2　堆叠通道

在 Tcl 中，不仅可以为特殊的“设备”创建新的驱动，还可以创建“堆叠”的通道，这种通道不是直接地访问设备，而是作为 Tcl 和一些其他通道之间的媒介筛选层存在。TclTLS(Tcl 和 OpenSSL 之间的接口)就是这样工作的。它并没有真正创建访问网页服务器的套接字，而是在套接字和 Tcl 之间增加了一个 SSL 筛选器。

要将一个通道加入堆栈中，应该使用 Tcl_StackChannel。

```
Tcl_Channel Tcl_StackChannel( Tcl_Interp interp,
        Tcl_ChannelType *typePtr, ClientData instanceData,
        int mask, Tcl_Channel channel)
```

除了经常出现的解释器参数，这个函数的参数包括指向 Tcl_ChannelType 结构体的指针(与前面那个示例展示的类似)，实例数据，描述模式(TCL_READABLE、TCL_WRITABLE 或 TCL_EXCEPTION)的掩码，以及将加于其上的那个通道。返回新创建

的处于顶层的这个通道结构体。

一旦通道进行了堆叠，试图对原通道进行 I/O 操作的函数，例如 Tcl_ReadChars 及 Tcl_WriteChars，系统会自动地将它们的调用重定向到堆栈中最顶层的通道。换句话说，所有的 Tcl_Channel 标记都仍然是有效的，无论它们处在堆栈中的什么位置，但是除了标准的 I/O 函数之外，没有能对它们进行访问的“后门”。在脚本层次，Tcl 自动地为通道描述符重新赋值，让它们指向堆栈顶端的通道。

可以使用 Tcl_UnstackChannel 把一个通道从堆栈中取出，适用于移除筛选器的情况——例如不再需要把输出编码为一个文件了。它的原型很简单。

```
int Tcl_UnstackcChannel( Tcl_Interp interp,
        Tcl_Channel channel)
```

旧的通道将与通道符号名称关联在一起，由 Tcl_StackChannel 增加的过程模块会被销毁。如果 *channel* 指向一个非堆叠的通道，Tcl_UnstackChannel 与 Tcl_Close 功能相同。

42.5.3 ROT13 通道

作为创建自定义通道类型的一个示例，本节讲述实现“ROT13”密码的堆叠通道类型。这是一种很古老的用于消息加密的方法。它获取一个字符，然后在字母表中将它向下“旋转”13 个位置，然后使用该位置的字符。例如，字母 a 就成了 n，b 就成了 o，以此类推。使用 13 作为偏移数的优点在于，用这种加密方式再对消息进行一次加密，就完成了“解密”。当然，在如今这个时代 ROT13 完全没有什么保密性可言，但它可以作为一个简单的示例，让您了解通道可以做些什么。

因为我们是在操作一个筛选器，而不是与 Tcl 之外的设备或库进行交流的“终端”驱动，我们需要一种方法将通道连接到已经打开的 Tcl 通道上。为了完成这一点，我们创建了一个命令 addrot，它获取一个通道作为参数，并接受一个可选的数字，用来代替 13。在 Tcl 脚本中，它如下所示：

```
set fl [open /tmp/outfile w]
addrot $fl
```

下面是实现这个 addrot 命令的命令函数。

```
/* This structure serves as the instance data that gets
 * passed to the channel driver functions. */

typedef struct {
    Tcl_Channel self;
    Tcl_Channel parent;
    int rotate;
} RotInstance;

static int
AddRotCmd(ClientData clientData, Tcl_Interp *interp,
        int objc, Tcl_Obj *CONST objv[]) {
    Tcl_Channel parent;
    int modePtr = 0;
    int rotate = 0;
    RotInstance *rotinstance;
```

```
    if(objc < 2) {
        Tcl_WrongNumArgs(interp, 1, objv, "chan ?rotation?");
        return TCL_ERROR;
    }
    /* Get the channel from its name , fail if it's
    * not a valid channel. */
    parent = Tcl_GetChannel(interp, Tcl_GetString(objv[1]),
                    &modePtr);
    if (parent == NULL) {
            Tcl_AppendResult(interp, Tcl_GetString(objv[1]),
                    "is not a valid channel", NULL);
    return TCL_ERROR;
    }

    /* We take an additional argument that is the amount
     * to  'rotate' for our cipher. */
    if (objc == 3) {
        if (Tcl_GetIntFromObj(
                interp, objv[2], &rotate) != TCL_OK) {
                return TCL_ERROR;
        }
    } else {
        rotate = 13;
    }
    rotinstance =
        (RotInstance *)Tcl_Alloc(sizeof(RotInstance));

    rotinstance->rotate = rotate;
    rotinstance->parent = parent;

    /* Stack the channel */
    rotinstance->self = Tcl_StackChannel(
        interp, &RotChan, (ClientData)rotinstance,
        modePtr, parent);

    return TCL_OK;
}
```

对 Tcl_GetChannel 的调用返回了一个 Tcl_Channel 结构体，这是已经存在的通道的描述符。这个通道是对文件发送或取出数据的“真正”的通道，我们堆叠的通道从这个通道中读出数据或向这个通道中写入数据。然后代码向结构体 RotInstance 的域中存入这个通道实例要“旋转”的字符个数以及原通道的句柄。最后，我们使用 Tcl_StackChannel 来堆叠通道，把 RotInstance 结构体作为实例数据传递。这样就创建了新的通道，堆叠在已经存在的通道上面。如果是在为指定的设备创建驱动，在这个命令中可以用 Tcl_CreateChannel 代替 Tcl_StackChannel。

理解新通道的功能的关键，在于对 RotChan 结构的理解，其声明如下：

```
static
Tcl_ChannelType RotChan = {
    "rot13_channel",
    TCL_CHANNEL_VERSION_3,
    RotcloseProc,
    RotinputProc,
    RotoutProc,
    NULL,
    RotsetOptionProc,
    RotgetOptionProc,
```

```
    RotwatchProc,
    RotgetHandleProc,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};
```

我们的通道筛选器并不是特别复杂。RotcloseProc 所做的就是调用 Tcl_Free 释放分配给实例数据的存储空间。RotwatchProc 什么也不做，RotgethandleProc 返回一个错误，因为这里没有可返回的低层级句柄。

下面我们详细地检查输入和输出函数。首先我们来明确一下这里的输入和输出的含义：输出是指 Tcl 向外送出数据，例如 puts 命令；输入是指数据被读取，例如 read 或 gets。

```
static int
RotoutPutProc(ClientData instanceData, CONST84 char *buf,
        int toWrite, int *errorCodePtr) {
    RotInstance *rotinstance = (RotInstance *)instanceData;
    char *outbuf;

    outbuf = Tcl_Alloc(toWrite);
    rot13(buf, outbuf, toWrite, rotinstance->rotate);

    /* We have pushed the data down to the next layer
     * of the stack */

    Tcl_WriteRaw(rotinstance->parent, outbuf, toWrite);
    Tcl_Free(outbuf);
    return toWrite;
}
```

输出函数获取 4 个参数：实例数据，包含输出数据的缓冲区，缓冲区的内容是多少字节和指向一个整型数的指针(该整型数是遇到问题时的 POSIX 错误代码)。

RotoutputProc 配置了第二个缓冲区，用来保存进行 rot13 操作后变化得到的文本，该操作把编码后的数据放到 outbuf 缓冲区中。巧妙之处在于对 Tcl_WriteRaw 的调用。您必须对堆叠通道使用这个写函数，而不能使用 Tcl_WriteChars 这样的函数。这样做是因为我们是在对一个筛选器进行操作，必须将数据写入堆叠在它下面的那个通道中，我们用它来存放实例数据。另一方面，Tcl 自动地将对 Tcl_WriteChars 这类函数的调用重定向到堆叠通道的顶部。

如果这不是一个堆叠通道(即不是一个筛选器)，那么就不会写入 Tcl 另一层，而是写入指定的设备。例如，Apache Rivet 的输出通道将数据从网页服务器发送到浏览器，通过对 Apache API——ap_rwrite(buf, toWrite, globals->r)的调用——将缓冲区转交给 Apache。在前面的代码中，释放了输出缓冲区使用的内存之后，还返回了事实上写出了多少数据。就我们这个示例而言，这总是等于我们要求写出的数据量，因此我们可以就返回那个值。特别是在有关非阻塞式通道的情况下，这些数字就更不易处理了，更多细节见参考文档。

当从通道中读取数据时就要使用输入函数。例如：

```
set fl [open /tmp/outfile]
```

```
addrot $fl
set data [read $fl]
```

在读取操作中，Tcl 库调用这个函数从我们的通道获得数据。

```
static int
RitinputProc(ClientData instanceData, char *buf,
        int bufSize, int *errorCodePtr) {
    RotInstance *rotinstance = (RotInstance *) instanceData;
    char *inbuf;
    int read = 0;

    inbuf = Tcl_Alloc(bufSize);

    /* We need to read the data from the next
     * layer of the stack. */

    read = Tcl_ReadRaw(rotinstance->parent, inbuf, bufSize);
    rot13(inbuf, buf, read, rotinstance->rotate);
    Tcl_Free(inbuf);
    return read;
}
```

输入函数需要传入 4 个变量：实例数据；写入数据的缓冲区(别忘了 Tcl 还要从这个函数中读取数据)；缓冲区能够保存的数据量；一个指向 int 整型变量的指针，用于存放错误代码。对于我们这个通道，输入操作的实现就是输出操作的逆过程。因为这是一个简单的筛选器，我们自己并没有数据，需要从它堆叠其上的下一个通道中读取，并使用了 Tcl_ReadRaw，此处要注意实际上读取到的字符数量。与写函数类似，对堆叠通道必须使用这个读函数，而不能使用 Tcl_ReadChars 这样的函数，Tcl 自动地将其重定向到堆叠通道的顶部。我们使用 rot13 函数进行“解密”，将得到的数据传给输入缓冲区 buf，释放前面那个输入缓冲区，通过返回我们从下面那条通道中读取到的总数，让 Tcl 知道在 buf 中等待被收集的数据是多少字节。因为我们只是在进行筛选操作，所以不必对文件本身做任何事。

如果那些通道是一个 I/O 系统，那么输入和输出函数就是必须编写的核心内容。Tcl 通过 chan configure 提供了设置通道选项的方法。我们可以让用户使用 Tcl_DriverSetOptionProc 函数来修改“旋转”的数值。Tcl 代码：

```
puts [fconfigure $fl -rotate 10]
```

最终会调用下面的函数。它获取的参数包括一个实例数据指针、一个解释器、要设置的选项名称(-rotate)以及为该选项设置的字符串值(10)。

```
static int
RotsetOptionProc(ClientData instanceData, Tcl_Interp *interp,
        CONST char *optionName, CONST char *newValue) {
    RotInstance *rotinstance = (RotInstance *)instanceData;

    if (strcmp(optionName, "-rotate") == 0) {
        if (Tcl_GetInt(interp, newValue,
                &(rotinstance->rotate)) != TCL_OK) {
            return TCL_ERROR;
        }
    return TCL_OK;
    }
    return Tcl_BadChannelOption(interp, optionName,
                                "rotate");
}
```

这里只有一个可用的选项名称，我们就查找它并尝试为它设置对应的值。如果我们给出了错误的选项名称，Tcl_BadChannelOption 就会被调用，返回一个错误，将错误的名称保存在 optionName 变量中，并且返回可用的选项名称的列表(名称没有开头的那个-符号)。

还有一个函数，用于取得选项所对应的当前值。它可以返回指定的一个选项的值，例如-rotate，也可以返回所有选项的值。在 Tcl 中，这两种情况表现如下：

```
set rotate [chan configure $fl -rotate]
set alloptions [chan configure $fl]
```

获取选项的函数所需要的参数与设置选项的函数相同，只不过其结果会被添加到 Tcl_DString 当中。如果给定的 optionName 为 NULL，就表示要获取所有可用的选项及相应的值。

```
static int
RotgetOptionProc(ClientData instanceData, Tcl_Interp *interp,
            CONST char *optionName, Tcl_DString *optionValue) {
        RotInstance *rotinstance = (RotInstance *)instanceData;
        Tcl_Obj *rotval;

        if (optionName != NULL &&
            strcmp(optionName, "-rotate") != 0) {
        return Tcl_BadChannelOption(interp, optionName,
                            "rotate");
        }
        rotval = Tcl_NewIntObj(rotinstance->rotate);
        if (optionName == NULL) {
            Tcl_DStringAppendElement(optionValue, "-rotate");
        }
        Tcl_DStringAppendElement(optionValue,
                            Tcl_GetString(rotval));
        return TCL_OK;
}
```

在处理堆叠的通道时，我们需要考虑的不仅是我们的通道的“旋转”选项，还有它下面的所有通道的所有选项，这里就是文件通道。在前面的代码中，我们首先进行了检查，确保对它的这个请求是有效的，否则就通过 Tcl_BadChannelOption 返回错误。然后我们将结果添加到 DString 中——如果-rotate 指定到 chan configure，则只有旋转值，其他情况下返回字符串-rotate 和相应的值。

最后，我们可以很简单地实现 ROT13 算法本身。

```
static void
rot13( char *in, char *out, int len, int rot) {
    int i;
    int inc;
    int outc;

    for (i = 0; i < len; i ++) {
        inc = in[i];
        if (inc >= 65 && inc <= 90) { /* A-Z */
            if (inc + rot >90) {
                outc = 64 +((inc + rot) -90);
            } else {
            outc = inc +rot;
            }
```

```
        } else if (inc >= 97 && inc <= 122) {/* a-z */
            if(inc + rot > 122) {
                outc = 96 + ((inc + rot) - 122);
            } else {
                outc = inc + rot;
            }
        } else { /* Other characters passed through. */
            outc = inc;
        }
        out[i] = outc;
    }
}
```

第43章 事件处理

本章讲述 Tcl 中关于事件处理的库函数。您为事件处理所编写的代码可以分为三个部分。第一个部分是创建事件处理器的代码；它通知 Tcl 在遇到指定事件时调用指定的回调函数。第二个部分是回调函数本身。第三个部分是调用 Tcl 的事件调度器来处理事件的顶层代码。

Tcl 支持两大类事件：文件事件和时间事件。Tcl 还允许您创建休眠回调，当 Tcl 无事可做的时候就可以调用这个函数；Tk 中使用了休眠回调来推迟刷新和其他消耗时间的计算，直到所有的未决事件都得到了处理。

43.1 本章出现的函数

Tcl 中用于事件处理的函数如下。

- `void Tcl_CreateChannelHandler(Tcl_Channel` *`channel`*`,`
  `int` *`mask`*`, Tcl_FileProc *`*`callback`*`,`
  `ClientData` *`clientData`*`)`
  设定当句柄为 *channel* 的通道出现 *mask* 描述的情形之一时就应该调用 *callback*。
- `void Tcl_DeleteChannelHandler(Tcl_Channel` *`channel`*`,`
  `Tcl_FilePro *`*`callback`*`, ClientData` *`clientData`*`)`
  删除前面以给定的 *callback* 和 *clientData* 参数调用 Tcl_CreateChannelHandler 函数为 *channel* 配置的处理器。
- `Tcl_TimerToken Tcl_CreateTimerHandler(int` *`milliseconds`*`,`
  `Tcl_TimerProc *`*`callback`*`, ClientData` *`clientData`*`)`
  设定在经过了 *milliseconds* 毫秒之后，调用 *callback*。返回可用于取消这次回调的一个标记。
- `void Tcl_DeleteTimerHandler(Tcl_TimerToken` *`token`*`)`
  如果 *token* 指向的时间回调还没有被触发，就取消它。
- `void Tcl_DoWhenIdle(Tcl_IdleProc *`*`callback`*`,`
  `ClientData` *`clientData`*`)`
  设定在 Tcl 无事可做的时候调用的 *callback*。
- `void Tcl_CancelIdleCall(Tcl_IdleProc *`*`callback`*`,`
  `ClientData` *`clientData`*`)`

删除由 *callback* 和 *clientData* 参数指定的现有的休眠回调。

- `void Tcl_DoOneEvent(int flags)`
  处理某一类型的一个事件，然后返回。*flags* 通常是 0，但也可以用来限制要处理的事件，或者在没有未决事件时立即返回。
- `void Tcl_SetMainLoop(Tcl_MainLoopProc *mainLoopProc)`
  向 Tcl 安装主循环函数，在应用程序的启动代码(包括命令行给出的脚本)完成后由 Tcl_Main 调用。
- `void Tk_MainLoop(void)`
  一个很方便的函数，进行事件处理直到该线程中所有的窗口都被关闭。在有 Tk 时可用。

43.2 通道事件

像网络服务器和 Tk 应用程序这类事件驱动的程序，在执行任一操作时都不应该出现长时间的阻塞，因为那样会阻止对其他事件的服务。例如，如果一个 Tk 应用程序试图从标准输入中读取数据，而当时没有可用数据，那么应用程序就会阻塞直到输出数据出现。这一期间进程就会被操作系统挂起，不能为 GUI 事件提供服务。这就是说，应用程序当时就对鼠标事件或是重绘命令没有响应。这种行为很可能引起用户的困扰，因为他们期望能一直与应用程序保持交互。

通道处理器提供了一种事件驱动的机制，用来对可能有长时间 I/O 延迟的通道进行读写操作。函数 Tcl_CreateChannelHandler 创建一个新的文件处理器。

```
void Tcl_CreateChannelHandler(Tcl_Channel channel,
        int mask, Tcl_FileProc *callback,
        ClientData clientData)
```

参数 *channel* 给定了一个已经存在的 Tcl 通道的句柄，可以指向各种不同类型的通道(如网络套接字、串行线、管道等)，而 *mask* 指明了在什么时候调用 *callback*。它是如下内容的位或结果。

- TCL_READABLE——Tcl 应该在 *channel* 中有等待读取的数据时调用 *callback*。
- TCL_WRITABLE——Tcl 应该在 *channel* 可以接受更多的输出数据时调用 *callback*。
- TCL_EXCEPTION——Tcl 应该在 *channel* 出现异常的时候调用 *callback*。

channel 参数必须与如下原型匹配：

```
typedef void Tcl_FilePro(ClientData clientData, int mask);
```

这里 *clientData* 参数与 Tcl_CreateChannelHandler 的 *clientData* 参数相同，*mask* 包含了 TCL_READABLE、TCL_WRITABLE 和 TCL_EXCEPTION 的位或结果，用于指定进行回调时的通道状态。您可以一次就为一个通道指定任意多的回调，只要它们各自有不同的 *callback* 和 *clientData* 值。

要删除一个文件处理器，调用 Tcl_DeleteChannelHandler，传入与创建时相同的

channel、*callback* 以及 *clientData* 参数。

```
void Tcl_DeleteChannelHandler(Tcl_Channel channel,
        Tcl_FileProc *callback, ClientData clientData)
```

提示： 您可以调用 Tcl_CreateChannelHandler，给定 mask 参数为 0，临时性地关闭指定通道。当需要使用处理器时可以再调用 Tcl_CreateChannelHandler 重置 mask。这种方法比调用 Tcl_DeleteChannelHandler 删除处理器更有效率。

使用通道处理器可以进行事件驱动的文件 I/O。不是打开一个通道，从头到尾进行读取，然后关闭通道，而是打开一个通道，然后为它创建一个通道处理器，然后返回。当通道可读时，就会调用回调函数。它对通道确切地进行一次读取请求，处理读取命令返回的数据，然后返回。当通道再次可读时(有可能立即就再次可读)，就再次调用这个回调。最终，通道中的所有数据都被读出，当通道再次可读时，读取调用会返回文件结束符。这时就可以关闭通道，删除通道处理器。使用这种方法，即使应用程序在读取通道时遇到很长时间的延迟，它也一直能对其他的事件保持响应。

例如，当 wish 的标准输入连接到一个真正的控制台时，它使用了一个文件处理器从它的标准输入读取命令。wish 的主程序为标准输入(文件描述符为 0)创建一个通道处理器，语句如下：

```
...
Tcl_Channel inChan = Tcl_GetStdChannel(TCL_STDIN);
Tcl_CreateChannelHandler(inChan, TCL_READABLE,
        StdinProc, (ClientData) inChan);
Tcl_DStringInit(&command);
Tcl_DStringInit(&line);
...
```

除了将 StdinProc 注册为标准输入的回调，这段代码还初始化了一个动态字符串，用作输入的各行的缓冲区，直到一条完整的 Tcl 命令准备好进行处理，另一个动态字符串用于从通道中读取各行文本。主程序进入事件循环的情况见 43.5 节。当标准输入端的数据可用时，StdinProc 就会被调用。它的代码[①]如下：

```
void StdinProc(ClientData clientData, int mask) {
        Tcl_Channel chan = (Tcl_Channel) clientData;
        Tcl_DString line, command;
        char *cmdStr;
        int code, count;
        count = Tcl_Gets(chan, &line);
        if (count < 0) {
            ... handle errors and end of file ...
        }
        Tcl_DStringAppend(&command,
            Tcl_DStringValue(&line), -1);
        cmdStr = Tcl_DStringAppend(&command, "\n", -1);
        Tcl_DStringFree(&line);
        if (Tcl_CommandComplete(cmdStr)) {
            Tcl_CreateChannelHandler(chan, 0,
                StdinProc, chan);
```

① StdinProc 的真正的实现比这要复杂得多，因为 Tk 使用了线程独有的数据来允许多线程操作。

```
            code = Tcl_Eval(interp, cmdStr);
            Tcl_CreateChannelHandler(chan, TCL_READABLE,
                StdinProc, chan);
            Tcl_DStringFree(&command);
            ...
        }
        ...
}
```

从标准输入进行读取，检查错误与文件尾之后，StdinProc 将新的数据添加到命令缓冲区当前的内容后面。然后检查命令缓冲区中是否已经包含了一条完整的 Tcl 命令(例如，如果有这样一行：foreach i $x {存在，而 foreach 循环块没有完整给出，那就不是完整的命令)。如果命令已经完整，StdinProc 就关闭通道处理器，处理命令，重建通道处理器，清除动态字符串缓冲区，从而准备好执行下一条命令。

提示： 通常最好对文件处理器使用非阻塞式 I/O，以便完全确定 I/O 操作不会导致阻塞。在使用非阻塞式 I/O 时，应该将正确的标志位传给 Tcl_FSOpenFileChannel，将 async 标志设置为 Tcl_OpenTcpClient，或者使用 Tcl_SetChannelOption 关闭阻塞功能。如果使用通道处理器向通道中写入数据，而又有很长的输出延迟，例如管道和网络套接字的情形，使用非阻塞式的 I/O 就至关重要；否则，如果在 Tcl_Write 调用时给出了太多数据，填满了输出缓冲区之后系统就会让进程进入休眠状态。

提示： 对于普通的磁盘文件，不应该使用本节讲述的事件驱动的方法，因为对这些文件的读写的延迟都非常小，操作系统可以认为它们根本没有延迟。通道处理器主要用于终端、管道、网络连接等通道，如果它们因延迟阻塞，可能阻塞相当长时间。不幸的是，基于网络的文件的延迟常常不可忽略，但操作系统仍然如此认为，仅仅允许阻塞式地等待它们。有时候提取到的数据就难以避免出现不完整的情况。

43.3 时间处理器

时间处理器在指定的时间间隔之后触发回调。例如，输入框组件就使用了时间事件让输入提示符闪烁，http 包使用时间事件进行对非常慢的连接的超时处理。对于输入框组件，当输入框获得输入焦点时，它显示插入光标，创建一个时间回调，在指定的时间长度后调用。这个时间回调擦除插入光标，再重设自身。下一次回调时，就会再显示出插入光标。这个过程无限重复，光标就会不断闪烁。当组件失去输入焦点时，就结束这个时间回调，擦除插入光标。

函数 Tcl_CreateTimerHandler 创建了一个时间回调。

```
Tcl_TimerToken Tcl_CreateTimerHandler(int milliseconds,
        Tcl_TimerProc *callback, ClientData clientData);
```

参数 *milliseconds* 指定了在多少毫秒之后进行回调。Tcl_CreateTimerHandler 立即返回，其返回值是一个标记，可以用来终止这个回调。过去了指定的时间之后，Tcl 调用 *callback*，它必须与如下原型匹配：

```
void Tcl_TimerProc(ClientData clientData);
```

它的参数是传给 Tcl_CreateTimerHandler 的 *clientData*。回调只会被调用一次，然后 Tcl 会自动删除回调函数。如果希望回调被一次又一次地调用，回调应该在每次被调用之后都用 Tcl_CreateTimerHandler 来重设自身。

提示： 并不能保证回调就确切地在指定的时间之后被调用。如果在指定的时候，应用程序因为处理其他的事件而处于忙碌状态，那么，直到 Tcl 下一次调用事件调度器时才会调用回调函数，详见 43.5 节。

函数 Tcl_DeleteTimerHandler 用于取消时间回调。

```
void Tcl_DeleteTimerHandler(Tcl_TimerToken token);
```

它只获取一个参数，即前面调用 Tcl_CreateTimerHandler 时返回的标记，然后删除不再会被调用的回调。即使在回调函数已经被调用时，调用 Tcl_DeleteTimerHandler 也是安全的，这种情况下这个删除函数没有任何效果。

43.4 休眠回调

函数 Tcl_DoWhenIdle 创建一个休眠回调。

```
void Tcl_DoWhenIdle(Tcl_IdleProc *callback,
        ClientData clientData);
```

设定在应用程序下次进入休眠时就调用这里的 *callback*。应用程序在调用了 Tcl 的主事件处理函数 Tcl_DoOneEvent 之后，没有要进行处理的通道事件和时间事件时，就会进入休眠状态。这时通常来说 Tcl_DoOneEvent 会挂起进程，等待事件出现。不过，如果存在休眠回调，那就调用它们。休眠回调在调用 Tcl 命令 update 时也会被调用。作为休眠回调的 *callback* 必须与如下原型匹配：

```
typedef void Tcl_IdleProc(ClientData clientData);
```

它不返回结果，只获取一个参数，即传给 Tcl_DoWhenIdle 的 *clientData* 参数。

Tcl_CancelIdleCall 删除休眠回调，以后就不会再调用它了。

```
void Tcl_CancelIdleCall(Tcl_IdleProc *callback,
        ClientData clientData);
```

Tcl_CancelIdleCall 删除所有与 *callback* 和 *clientData* 匹配的回调(可能多于一个)。如果没有匹配的回调，这个函数没有效果。

休眠回调大量用于 29.3 节讲述的延迟操作。Tk 中休眠回调最常见的用处就是组件刷新和几何重算。因为在组件状态被修改之后立即重画通常不好，当相关的修改同时发生时，就会产生毫无意义的刷新。例如，调用下面的 Tcl 命令，修改了一个标签组件.l。

```
.l configure -background purple -textvariable msg
set msg "Hello, world!"
.l configure -bd 3m
```

每一条命令都修改了组件，以不同的方式需要它刷新，让每条命令都重画组件却不明智。那样会进行三次刷新，不仅不必要，还会导致组件闪烁，因为它快速进行了多次重画。最好是等到这些命令都处理完成，然后刷新一次。休眠回调提供了一种方法，可以知道什么时候所有可用的事件都被处理完了。

43.5　调用事件调度器

前文讲述了事件管理的前两个部分：创建事件处理器和编写回调函数。事件管理的最后一部分内容是调用 Tcl 事件调度器，它等待事件发生并调用适当的回调。如果不调用事件调度器，就不会处理事件，也不会调用回调函数。

Tcl 提供了一个用于事件调度的函数，Tcl_DoOneEvent，Tk 提供了另一个，Tk_MainLoop。Tk 应用程序通常使用 Tk_MainLoop，而非 Tk 应用程序总是使用 Tcl_DoOneEvent，通常是在某类循环中使用。

提示： Tcl 应用程序可以使用 Tcl_SetMainLoop 将一个主循环函数安装到 Tcl 中。Tcl_Main 在完成初始启动脚本(即在命令行中指定的脚本文件)后就调用安装的主循环函数。Tk 默认将 Tk_MainLoop 作为主循环函数；如果希望在使用 Tk 的同时使用自定义的主循环函数，就需要确保在加载 Tk 包之后安装了自定义主循环函数。

Tk_MainLoop 不需要参数，也不返回结果；通常它只在主程序的初始化完成后调用一次。Tk_MainLoop 反复地调用 Tcl 事件调度器，处理事件。在所有可用的事件处理完成之后，它挂起线程，直到更多的事件出现，然后反复重复这个过程。它仅在该线程创建的 Tk 窗口关闭后返回(例如，在执行 destroy .命令之后)。Tk 应用程序典型的主程序会创建一个 Tcl 解释器，创建主 Tk 窗口和应用程序，执行一些应用程序特有的初始化(例如处理一段 Tcl 脚本以创建应用程序的界面)，然后调用 Tk_MainLoop。当 Tk_MainLoop 返回时，应用程序就退出了。由此 Tk 提供了应用程序运行的顶层控制，应用程序执行的工作都是由 Tk_MainLoop 调用的事件处理器完成的。

另一个用作事件调度器的函数是 Tcl_DoOneEvent，它提供了一个较低层的事件调度器接口。

```
int Tcl_DoOneEvent(int flags)
```

flags 参数通常是 0(或者 TCL_ALL_EVENTS，与 0 的意义相同)。这种情况下 Tcl_DoOneEvent 处理一个事件，然后返回 1。如果没有未决事件，Tcl_DoOneEvent 挂起线程，直到事件出现，然后处理事件，然后返回 1。

例如，Tk_MainLoop 就用 Tcl_DoOneEvent 实现。

```
void Tk_MainLoop(void) {
    while (/* Number of main windows > 0 */) {
        Tcl_DoOneEvent(0);
    }
}
```

如您所见，Tk_MainLoop 就是一次又一次地调用 Tk_DoOneEvent，直到所有的主窗口都被关闭。

Tcl_DoOneEvent 也可以用于 vwait 和 tkwait 这样的命令，等待某些事情发生，然后处理事件。例如，vwait 命令进行事件处理，直到给定变量更新，然后返回。下面是实现这个命令的 C 代码：

```
int done;
...
Tcl_TraceVar(interp, nameString,
    TCL_GLOBAL_ONLY | TCL_TRACE_WRITES | TCL_TRACE_UNSETS,
    VwaitVarProc, &done);
done = 0;
while (!done) {
    Tcl_DoOneEvent(0);
}
Tcl_UntraceVar(interp, nameString,
    TCL_GLOBAL_ONLY | TCL_TRACE_WRITES | TCL_TRACE_UNSETS,
    VwaitVarProc, &done);
}
...
```

变量 nameString 指定了要等待其更新的变量。这段代码创建了一个跟踪回调，在该变量被更新时调用，然后就一次又一次地调用 Tcl_DoOneEvent，直到 done 标志被设置，说明指定变量已经被设置或删除。然后变量跟踪进行的回调如下：

```
char *VwaitVarProc(ClientData clientData,
    Tcl_Interp *interp, const char *name1,
    const char *name2, int flags)
{
    int *donePtr = clientData;

    *donePtr = 1;
    return NULL;
}
```

clientData 参数是一个指向标志变量的指针。VwaitVarProc 只在该变量被设置或删除时调用一次，因此它必须要做的就是在被调用时把标志变量设置为 1。

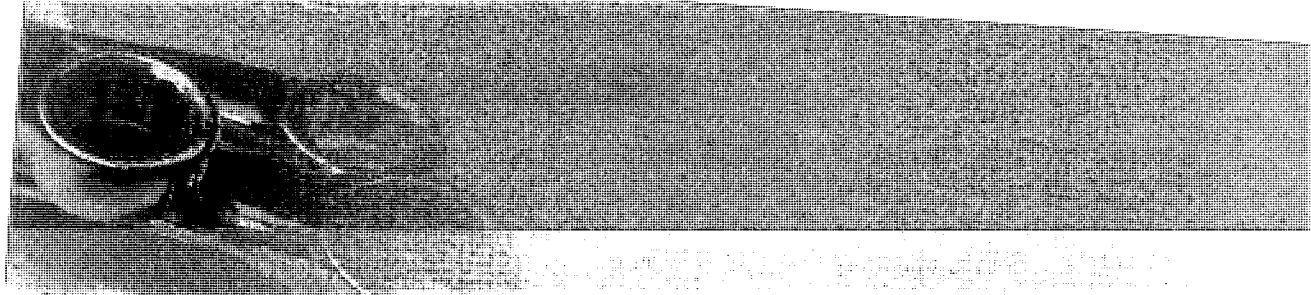
传给 Tcl_DoOneEvent 的 flags 参数可以用于限制需要考虑的事件的类型。如果它包含了 TCL_FILE_EVENTS[①]、TCL_TIMER_EVENTS 和 TCL_IDLE_EVENTS 中的任何位标志，则仅考虑标志位指定的相应类型的事件。还有，如果 flags 参数包含了 TCL_DONT_WAIT 位标志，或者文件事件和时间事件都没有被选定，则在没有待处理的事件时 Tcl_DoOneEvent 也不会挂起线程。它会立即返回结果 0，表示它无事可做。例如，update idletasks 命令用如下代码实现：

① Tk 的 X 事件在事件调度的最底层是作为 TCL_FILE_EVENTS 实现的，调度器直接监察 X 库用来与 X 服务器通信的通道。这是相当有技巧的代码，它允许 Tk 的用户，即使是在使用 C 库时，也可以不必关注细节。

```
while (Tcl_DoOneEvent(TCL_IDLE_EVENTS) != 0) {
    /* empty loop body */
}
```

作为对比，普通的 update 命令的核心代码如下：

```
while (Tcl_DoOneEvent(TCL_ALL_EVENTS | TCL_DONT_WAIT)) {
    /* empty loop body */
}
```



第44章 文件系统的交互

对有关文件和目录的工作，Tcl 提供了独立于平台的 C 程序 API。使用这些函数而非平台的原生系统调用，可以让代码更具可移植性。Tcl API 还会自动处理字符字符串的 Tcl 原生 UTF 表达形式与系统编码之间所需要的转换。这些函数的所有输入与输出字符串变量均基于 Tcl 原生的 UTF 编码字符串。而且，Tcl API 自动支持 Tcl 的虚拟文件系统。如果把路径作为变量提供给这些函数，而该路径表达的是应用程序所挂载的虚拟文件系统中的位置(例如一个 ZIP 文件或 FTP 站点)，这些函数可以访问指定的文件，就如同那是普通的磁盘文件系统一样。

提示： 这些函数中的大部分都有与之对应的脚本层次的 Tcl 命令。一般来说，与文件系统进行交互时尽可能使用 Tcl 脚本，因为那样开发更快，将来修改也更容易。这些 C 层次的命令适合于在 C 程序开发中需要执行不常见的文件系统操作时使用。

44.1 Tcl 文件系统函数

表 44.1 列出了为文件系统访问提供的函数和功能相同的脚本层次的命令。在 Tcl 的参考文档 FileSystem.3 当中可以找到更多细节。这些函数大部分在成功时返回 0，在失败时返回-1，并且将 errno 设置为失败原因的描述。另外，Tcl 的文件系统 API 几乎完全基于 Tcl 对象。一些函数可能自行缓冲存储内容的内部表达形式和其他路径相关的字符串，因此传给这些函数的对象的引用计数必须大于 0，例如：

```
Tcl_Obj *fileName = Tcl_NewStringObj("/tmp/foobar", -1);
Tcl_IncrRefCount(fileName);
ret = Tcl_FSDeleteFile(fileName);
Tcl_DecrRefCount(fileName);
```

如果函数返回一个对象，应该假设它的引用计数是 0。与任何其他创建的 Tcl 对象一样，如果在其他地方使用对象，可能需要为对象调用 Tcl_IncrRefCount，确保它不在错误的地方被意外释放。管理 Tcl 对象的引用计数的更多信息参见第 32 章。

表 44.1 Tcl 用于文件系统交互的 C 函数和与之功能相同的 Tcl 命令

C 函 数	功能相同的 Tcl 命令
Tcl_FSAccess	file readable file writable

续表

C 函 数	功能相同的 Tcl 命令
Tcl_FSChdir	cd
Tcl_FSCopyDirectory	file copy
Tcl_FSCopyFile	
Tcl_FSCreateDirectory	file mkdir
Tcl_FSEvalFile	source
Tcl_FSFileAttrsGet Tcl_FSFileAttrsSet Tcl_FSFileAttrStrings	file attributes
Tcl_FSFileSystemInfo	file system
Tcl_FSGetCwd	pwd
Tcl_FSGetNormalizedPath	file normalize
Tcl_FSGetPathType	file pathtype
Tcl_FSJoinPath	file join
Tcl_FSJoinToPath	
Tcl_FSLink	file link file readlink
Tcl_FSListVolumes	file volumes
Tcl_FSLoadFile	load
Tcl_FSLstat	file lstat
Tcl_FSMatchInDirectory	glob
Tcl_FSOpenFileChannel	open
Tcl_FSPathSeparator	file separator
Tcl_FSRemoveDirectory	file delete
Tcl_FSDeleteFile	
Tcl_FSRenameFile	file rename
Tcl_FSSplitPath	file split
Tcl_FSStat	file stat
Tcl_FSUtime	file mtime
Tcl_TranslateFileName	file nativename

44.2 虚拟文件系统

Tcl 文件系统交互实现的另一个重要方面是，它构建在可通过 Tcl 的 C API 访问的“虚拟文件系统”的基础上。换句话说，可以编写一个新的“文件系统”，让 Tcl 像访问普通文件一样访问它。这种“文件系统”的一些示例包括档案文件，例如 ZIP 和 tar，以及远程文件——WebDAV、FTP 和 HTTP。当用适当的扩展包“挂载”一个 ZIP 文件后，就可以用 cd 命令进入，用 open 命令打开文件，使用 glob，甚至对其中的 Tcl 文件应用 source 命令，就像它是 Tcl 所在的磁盘的一部分一样。第 11 章讨论了在应用程序中如何使用 tclvfs 扩展来挂载和访问这样的虚拟文件系统。

实现新的虚拟文件系统的详细过程超出了本书的范围，可以在 Tcl 那些卓越的参考文档中找到——特别是 FileSystem.3 参考页面。其基本过程与第 42 章介绍的自定义通道类型的实现相近：创建一个结构体，将它的域设置为函数，那些函数实现了本章讲述的这些操作。

第45章 操作系统工具

Tcl 提供了很多用于访问操作系统服务的函数。这一层级是与平台无关的，除了有些应该让用户可以访问的低层选项之外，因为这些选项并不是所有平台都会提供的。

45.1 本章出现的函数

本章讨论的用于访问操作系统服务的函数如下。

- `Tcl_Channel Tcl_OpenCommandChannel(Tcl_Interp *interp, int argc, CONST char *argv, int flags)`
  打开一个命令通道，以 *argv* 作为命令及其参数。
- `Tcl_DetachPids(int numPids, int *pidPtr)`
  让 Tcl 管理经由 *pidPtr* 数组传入的 *numPids* 子进程。
- `Tcl_ReapDetachedProcs()`
  对每一个后台进程调用 waitpid 系统调用，如果它已经退出，则清理其状态。如果进程还未退出，Tcl_ReapDetachedProcs 不等待它退出。
- `Tcl_Pid Tcl_WaitPid(int pid, int *statPtr, int options)`
  waitpid 系统调用的包装。
- `Tcl_AsyncHandler Tcl_AsyncCreate(Tcl_AsyncProc *func, ClientData clientData)`
  创建一个可以中断 Tcl 执行的异步处理器，为它返回一个标记。
- `Tcl_AsyncDelete( Tcl_AsyncHandler async)`
  给定标记，删除该异步处理器。
- `Tcl_AsyncMark(Tcl_AsyncHandler async)`
  将 *async* 指定的异步处理器标记为做好了运行准备。
- `int Tcl_AsyncInvoke(Tcl_Interp *interp, int code)`
  调用所有已经准备好了的处理器。
- `int Tcl_AsyncReady()`
  检查是否所有的处理器都做好了运行准备。
- `CONST char *TCL_SignalId(int sig)`
  返回机器可读的字符串，描述信号，例如 SIGPIPE。

- `CONST char *Tcl_SignalMsg(int sig)`
  返回人工可读的字符串，描述信号，例如 bus error。
- `Tcl_Exit(int status)`
  退出 Tcl 和指定进程。
- `Tcl_Finalize()`
  与 Tcl_Exit 相似，但是只结束对 Tcl 的使用，它并不终止整体进程。
- `Tcl_CreateExitHandler(Tcl_ExitProc proc,`
  `ClientData clientData)`
  创建一个处理器，在调用 Tcl_Exit 或 Tcl_Finalize 时调用。
- `Tcl_DeleteExitHandler(Tcl_ExitProc proc,`
  `ClientData clientData)`
  删除前面由 Tcl_CreateExitHandler 创建的退出处理器。
- `CONST char* Tcl_GetHostName()`
  返回指向一个字符串的指针，该字符串的内容为当前主机名。
- `Tcl_GetTime(Tcl_Time *timePtr)`
  为前面配置的 *timePtr* 结构体填充内容，包括从新纪元时刻到当前时刻的总秒数，以及从上一秒开始到当前时刻的毫秒数。
- `int Tcl_PutEnv(const char *assignment)`
  根据用 *NAME=value* 格式给出的 *assignment* 字符串，设置环境变量。

45.2 进程

Tcl_OpenCommandChannel 函数的结果是 Tcl 外部的进程——等同于处理 exec 或 open |命令。这里通常情况下也应该尽可能使用 Tcl 代码而非 C 代码，因为功能基本相同，使用 Tcl 代码编写要快得多。语法如下：

```
Tcl_Channel Tcl_OpenCommandChannel(
        Tcl_Interp *interp, int argc,
        CONST char **argv, int flags);
```

argc 是通过字符串数组 *argv* 传递给命令的参数的个数。*argv*[*argc*]必须是 NULL，因此要确认让 *argc* 比 *argc* 大一个元素。记得这里可以启动管线序列命令，例如 ls、|、grep 以及 foo，而且标准的 exec 参数(参见第 12 章)，如>、<、&等也都是 *argv* 的有效元素。返回的通道受到如下标志的由或操作联合的结果的影响。

- TCL_STDIN
  新创建的通道作为后面 Tcl_OpenCommandChannel 启动的第一个进程的标准输入。换句话说，写入该通道的数据会被送入那个进程。如果这个标志未设置，第一个进程使用当前(调用)进程的标准输入。
- TCL_STDOUT
  新创建的通道作为前面 Tcl_OpenCommandChannel 启动的上一个进程的标准输

出。对这个通道进行的读取操作返回的数据是由上一个进程创建的。如果这个标志未设置，上一个进程的标准输出输出到(当前)调用进程的标准输出。

- TCL_STDERR
  由 Tcl_OpenCommandChannel 发起的任何进程向标准错误通道进行的写操作都定向到这个新的通道。当这个通道关闭时，如果标准错误通道中还有数据，则会产生错误。如果这个标志未设置，标准错误数据输出到(当前)调用进程的标准错误。
- TCL_ENFORCE_MODE
  设置这个标志时，说明标准的 exec 参数，例如<和>，不能覆盖 TCL_STDIN、TCL_STDOUT 以及 TCL_STDERR 标志；如果试图进行重定向操作，则会发生错误。

提示： 新创建的通道并没有在提供的解释器中注册。有关通道注册的更多信息参见第 42 章。

在下面这个示例中，我们创建了 subtcl 命令，将一段脚本管道传输给 tclsh，作为子进程处理然后返回其输出。

```
static int
SubTclCmd(ClientData clientData, Tcl_Interp *interp,
        int objc, Tcl_Obj *CONST objv[]) {
    Tcl_Obj *out;
    CONST char **argv;
    Tcl_Channel chan;
    int argc = 1;
    static char *argv0 = NULL;

    if (objc != 2) {
        Tcl_WrongNumArgs(interp, 1, objv, "script");
        return TCL_ERROR;
    }
    if (argv0 == NULL) {
        Tcl_Eval(interp, "info nameofexecutable");
        argv0 = strdup(Tcl_GetStringResult(interp));
    }

    out = Tcl_NewObj();
    argv = (CONST char **) ckalloc(sizeof(char *) * 2);
    argv[0] = argv0;
    argv[1] = NULL;
    chan = Tcl_OpenCommandChannel(
        interp, argc, argv,
        TCL_STDOUT | TCL_STDIN | TCL_STDERR);
    ckfree((char *) argv);

    if (chan == NULL) {
        return TCL_ERROR;
    }
    if (Tcl_WriteObj(chan, objv[1]) < 0 ) {
        return TCL_ERROR;
    }
    if (Tcl_Flush(chan) != TCL_OK) {
        return TCL_ERROR;
```

```
    }
    if (Tcl_ReadChars(chan, out, -1, 0) < 0) {
        return TCL_ERROR;
    }
    if (Tcl_Close(interp, chan) != TCL_OK) {
        return TCL_ERROR;
    }
    Tcl_SetObjResult(interp, out);
    return TCL_OK;
}
```

我们构建了 argv 中的命令，将它传递给 Tcl_OpenCommandChannel。我们设置了 TCL_STDIN 标志，因此子进程能够进行读取操作，然后用 Tcl_WriteObj 将 objv[1]送到了通道，然后进行了转存操作确保数据到达。利用 TCL_STDOUT 标志，子进程将结果返回给了 chan，这里我们顺序读取结果，将其作为命令的结果返回。在 Tcl 中可以如下调用 subtcl 命令：

```
set result [subtcl {
    puts hello
    exit
}]
```

subtcl 命令返回它发起的 tclsh 进程的所有输出。这个示例可以进行的一个改进是去调用事件循环，这样在等待子进程结束时调用进程就不会被阻塞。

45.3 收割子进程

Tcl_OpenCommandChannel 足够用于管理外部进程的生命周期。然而，如果想自己发起外部进程，例如使用 fork 和 exec 系统调用等，仍然可以从 Tcl 中管理它们。

如果应用程序创建了一个子进程然后放弃了它(即父进程从不调用一个系统调用来等待子进程退出)，子进程就在后台运行，然后在它退出之后就成了一个僵尸进程。它一直作为一个僵尸进程存在，直到它的父进程正式地等待它，或者它的父进程退出。僵尸进程会占用系统进程表的空间，因此如果创建的僵尸进程够多，可能导致进程表溢出，其他人就无法创建更多的进程。要避免这种情况发生，必须调用如 wait 或 waitpid 这样的系统调用，它会返回僵尸进程的退出状态。一旦状态被返回，僵尸进程就会放弃它在进程表中占据的位置。这一过程通常被称为收割子进程。

Tcl_DetachPids 函数通知 Tcl 负责收割一个或多个子进程。

```
Tcl_DetachPids(int numPids, int *pidPtr);
```

pidPtr 参数是进程描述符的一个数组，numPids 是数组中的描述符的数量。现在这些进程都成为了 Tcl 的财产，调用者以后不应该再指向它们。

要收割这些子进程，可以调用 Tcl_ReapDetachedProcs，它负责在进程退出后的低层的清理工作。如果某些分离的进程仍然在运行，Tcl_ReapDetachedProcs 不会等待它们退出；它只清理那些已经退出的进程，然后返回。

如果需要等待特定的子进程，那么函数 Tcl_WaitPid 包装了 waitpid 系统调用，是完成

这个任务的正确工具。

```
Tcl_Pid Tcl_WaitPid(int pid, int *statPtr, int options);
```

pid 参数是指定进程描述符，statPtr 是指向一个整型数的指针，函数将其设置为 0 或 ECHILD，表明子进程的状态。options 依赖于底层的操作系统，但是 WNOHANG 的意思都是表示函数应该立即返回，即使有未退出的子进程也不进行阻塞。

45.4 异步事件

Tcl 没有提供处理 ANSI 或 POSIX 信号的命令(不过如果需要这些功能，可以使用 TclX 或 Expect 扩展包)，但是它的确提供了处理信号这类异步事件的通用框架。其基础结构的中心概念就是当信号(或者其他事件)到达时，如果 Tcl 忙，它就不应该打断 Tcl。例如，您试图进行 eval 操作，Tcl 也已经开始了这个命令的执行，如果这时一个信号到达，就可能导致稳定性被破坏。因此，Tcl 通过一个标志确定感兴趣的事件发生，然后在 Tcl 能够去处理的时候，去处理事件。

下面的示例演示了一个很简单的信号处理器。

```
#include <tcl.h>
#include <signal.h>

static Tcl_AsyncHandler SignalHandler = NULL;
static int sigs = 0;  /* Number of signals received. */

int Signals_Init(Tcl_Interp *interp) {

#ifdef USE_TCL_STUBS
    if (Tcl_InitStubs(interp, "8", 0) == NULL) {
        return TCL_ERROR;
    }
#endif

    Tcl_SetVar2Ex(interp, "::counter", NULL,
                Tcl_NewIntObj(0), 0);

    SignalHandler = Tcl_AsyncCreate(HandleSignals,
                (ClientData) interp);
    signal(SIGHUP, lowlevelhandler);
    signal(SIGINT, lowlevelhandler);

    if (Tcl_PkgProvide(interp, "signals", "0.1") != TCL_OK) {
        return TCL_ERROR;
    }
    return TCL_OK;
}
```

这段代码使用了 Tcl_AsyncCreate 函数，注册了名为 HandleSignals 的 Tcl_AsyncHandler 函数。返回值是描述该处理器的独一无二的标记。后文会提到，解释器对异步事件处理器来说并不是一直可用的，因此我们将当前解释器作为客户数据传入。这样会带来一些风险，解释器可能在调用处理器之前就被删除掉，但为了示例简单，我们假设这种情况不会出现。下面两行代码为 SIGHUP(挂起)和 SIGINT(中断)初始化了两个信号

处理器，使用了以下函数：

```
void lowlevelhandler(int signum) {
    sigs++;
    Tcl_AsyncMark(SignalHandler);
}
```

lowlevelhandler 函数就是在一个全局变量中增加接收到的信号的记录，然后调用 Tcl_AsyncMark 告诉 Tcl SignalHandler 的回调已经准备好了。当 Tcl 能够处理异步事件的时候，它就会调用 HandlerSignals 函数。

```
static int
HandleSignals(ClientData clientData,
            Tcl_Interp *interp, int code) {
    int i = 0;
    int counterVal = 0;
    Tcl_Obj *counterObj;
    if (interp == NULL) {
        interp = (Tcl_Interp *)clientData;
        if (interp == NULL) {
            /* It's deleted, can't do anything useful. */
            return TCL_ERROR;
    }
    }

    counterObj = Tcl_GetVar2Ex(interp, "::counter", NULL, 0);
    Tcl_GetIntFromObj(interp, counterObj, &counterVal);
    while(sigs) {
        Tcl_SetVar2Ex(interp, "::counter", NULL,
                    Tcl_NewIntObj(counterVal+1), 0);
        i++;
        sigs--;
    }
    return code;
}
```

传给信号处理器的第一个参数对应于 Tcl_AsyncCreate 函数的 clientData 参数，和其他的 Tcl 子系统一样，作为将数据传给由异步处理器调用的函数的方法。如果在解释器刚刚完成对一条命令的处理后就调用了这个处理器，那么 interp 参数指明了处理命令的解释器，而 code 是该命令返回的完成代码。(命令的结果表达为解释器的结果。)在处理器返回时，它留在解释器中的结果都作为命令的结果返回，处理器的整型的返回值是该命令的新的完成代码。另外，如果 Tcl 处于事件循环中，或处于其他的没有活动中的解释器的状态，interp 就是 NULL，而 code 是 0，处理器的返回值会被忽略。

提示： 让异步事件处理器修改解释器的结果或返回与它的 code 参数不同的完成代码，是不好的做法。这种行为可能会干扰对脚本的细微处理，并导致异常难以找到的错误。如果一个异步事件处理器需要处理 Tcl 脚本，它应该首先调用 Tcl_SaveInterpState 保存解释器状态，传进 code 参数。当异步处理器的任务完成后，它应该调用 Tcl_RestoreInterpState 重置解释器状态，返回 code 参数。

在这个简单的示例中，我们首先尝试获得一个有效的解释器，如果不行，就返回错误情况。然后每次信号到达，我们就把计数变量加 1。

提示：即使处理器获得了一个正在活动的解释器作为参数，如果有多个解释器正在活动，也不能保证具体是哪个解释器，也不能保证解释器处于什么状态。因此，建议您不要在这个处理器中做太多的事情。一个可能的解决方案是把异步事件作为 Tcl 的事件循环中的一个事件源，详见第 42 章。

45.5 信号名称

有关信号本身，Tcl 还提供了两个函数，用于把信号号码转化为更可读的消息。Tcl_SignalId 和 Tcl_SignalMsg 都获取一个 POSIX 信号号码作为参数，返回描述该信号的字符串。Tcl_SignalId 返回的是该信号在 signal.h 中定义的官方 POSIX 名称，而 Tcl_SignalMsg 返回的是人工可读的消息，描述了这个信号。例如：

```
Tcl_SignalId(SIGALRM)
```

返回了字符串 SIGALRM，而

```
Tcl_SignalMsg(SIGALRM)
```

返回的是“alarm clock”。

45.6 退出与清理

Tcl_Exit 函数调用可以干净地退出进程。它与 Tcl 的 exit 命令等同。永远不要直接调用 C 语言的 exit 函数；Tcl 经由 Tcl_CreateExitHandler 为退出处理器创建了回调机制。Tcl_Exit 确保进行了适当清理，而只调用 exit 则可能会略过这一步。和其他的回调一样，这个回调也把一个函数作为参数。

```
static void
OnExit(ClientData clientData) {
    fprintf(stderr, "We are down for the count\n");
}

...

Tcl_CreateExitHandler(OnExit, NULL);
```

可以把一个 clientData 的值传入退出处理器。这可以用于干净地关闭与其他系统、其他网络资源(如数据库服务器、连接的客户端等)的通信。您可以使用 Tcl_DeleteExitHandler 函数删除已经存在的退出处理器。

如果希望只是清理应用程序中的 Tcl，而让应用程序继续运行，不要关掉整个进程，这项工作可以由 Tcl_Finalize 函数完成。如果 Tcl 作为一个共享库(.so 或.dll)加载到应用程序中，在卸载该库前也应该调用这个函数。很多应用程序只加载模块而不进行卸载，因此这样的用法不常见到。

提示：如果在编写一个多线程的 Tcl 应用程序，对于退出线程、在指定的线程中结束对 Tcl 的使用、注册线程退出处理器，也有类似的函数，更多信息参见第 46 章。

45.7 其他

Tcl_GetHostName 函数返回一个字符串，包含了正在运行该 Tcl 的机器的主机名。为此分配的存储空间属于 Tcl，不应该尝试将其释放。

Tcl_PutEnv 函数用作 putenv 系统调用的替代。它接受一个格式为“变量名称=值”的字符串。经由 Tcl 的全局数组 env 处理环境变量也是可以的，需要使用 Tcl_SetVar 这类函数。

为了获得关于当前时间的信息，Tcl 提供了 Tcl_GetTime 函数，它取得由两个长整型数组成的结构体：从 UTC 时间 1970 年 1 月 1 日零时(新纪元时刻)到当前时刻的秒数，以及当前这一秒已经经过的微秒数。例如：

```
Tcl_Time timePtr;
...
Tcl_GetTime(&timePtr);
printf("seconds %ld\n", timePtr.sec);
printf("microseconds %ld\n", timePtr.usec);
```

第46章 线 程

本章内容包括对 Tcl 的 C 语言线程 API 的使用，以及在多线程应用程序中使用 Tcl。多线程的复杂程序的编写相当有技巧性，而且如果不小心对待，可能导致难以查找的错误。我们将讨论 Tcl 的 API，但不会介绍线程编程的各个细微之处。有关内容的参阅相关书籍，如 David Butenhof 所著的 *Programming with POSIX Threads*（《POSIX 线程编程》，ISBN 0-201-63392-2）。

46.1 本章出现的函数

本章讨论的与 Tcl 中的线程管理有关的函数如下。

- `int Tcl_CreateThread(Tcl_ThreadId *idPtr,`
  `Tcl_ThreadCreateProc threadProc,`
  `ClientData clientData, int stackSize, int flags)`
  创建一个新的线程，其 ID 为 *idPtr*，在 *threadProc* 函数中开始运行。有关 *stackSize* 和 *flags* 参数的信息见正文及参考文档。
- `Tcl_ExitThread(int status)`
  终止当前线程，调用针对该线程的退出处理器。
- `Tcl_FinalizeThread()`
  清理该线程的 Tcl 状态，调用线程退出处理器，但不终止线程。
- `Tcl_CreateThreadExitHandler(Tcl_ExitProc proc,`
  `ClientData clientData)`
  为线程注册一个退出处理器 *proc*，它将以 *clientData* 为参数调用。
- `Tcl_DeleteThreadExitHandler(Tcl_ExitProc proc,`
  `ClientData clientData)`
  删除该线程的退出处理器。
- `int Tcl_JoinThread(Tcl_ThreadId id, int result)`
  等待经由设置 TCL_THREAD_JOINABLE 标志的 Tcl_CreateThread 创建的线程退出。试图等待一个设定为不可加入的线程会返回错误。
- `void *Tcl_GetThreadData( Tcl_ThreadDataKey *keyPtr,`
  `int *size)`
  返回指向线程私有数据块的指针。它的参数是由所有线程共享的关键字以及存储

块的大小 *size*。在线程第一次请求存储块时，会自动分配，并将其值初始化为全零。Tcl_FinalizeThread 会自动取消对该存储块的分配。

- `TCL_DECLARE_MUTEX(mutexName)`
  以可移植的方式声明互斥体的宏。如果没有使用的线程，则没有效果。
- `void Tcl_MutexLock(Tcl_Mutex *mutexPtr)`
  锁定互斥体，如果有其他线程保持锁定，等待直到它解锁。
- `void Tcl_MutexUnlock(Tcl_Mutex *mutexPtr)`
  解锁互斥体。
- `void Tcl_MutexFinalize(Tcl_Mutex *mutexPtr)`
  在一个互斥体不再被使用后，释放与它相关的所有资源。
- `void Tcl_ConditionNotify(Tcl_Condition *condPtr)`
  解除正在等待条件变量的线程的阻塞。
- `void Tcl_ConditionWait(Tcl_Condition *condPtr,`
  `Tcl_Mutex *mutexPtr, Tcl_Time *timePtr)`
  等待 *condPtr* 中的条件满足。返回时锁定 *mutexPtr*。如果 *timePtr* 不是 NULL，则只为指定的条件等待指定的时间长度。
- `void Tcl_ConditionFinalize(Tcl_Condition *condPtr)`
  在不再需要一个条件配置之后，释放与它相关的所有资源。

46.2 线程安全

Tcl 是线程安全的，就是说多线程应用程序可以毫无问题地使用它。然而，Tcl 解释器必须被限制到一个线程之中，仅由一个线程访问，即创建了该解释器的那个线程。这意味着不能创建一个解释器，然后把它由多个线程共享。

46.3 构建支持线程的 Tcl

目前大部分 Tcl 发行版都是支持线程使用的。您可以检查解释器中是否存在::tcl_platform(threaded)来确认这一点。如果自己构建 Tcl，构建时必须以--enable-threads 运行./configure 脚本，以构建支持线程的 Tcl。关于构建 Tcl 的更多信息参见第 47 章。如果有了一个支持线程的 Tcl，记住还应该使用--enable-threads 来构建那些扩展包。

46.4 创建线程

新的线程由跨平台的 Tcl_CreateThread 函数创建，定义如下：

```
int Tcl_CreateThread(Tcl_ThreadId *idPtr,
                     Tcl_ThreadCreateProc threadProc,
                     ClientData clientData,
```

```
int stackSize, int flags);
```

idPtr 是指向一个整型数的指针，那里存放的就是新创建的线程的 ID。操作系统创建新的线程时，它在 *threadProc* 函数中开始，以 *clientData* 作为它唯一的参数。可以为新的线程设定堆栈大小，但是推荐使用 TCL_THREAD_STACK_DEFAUTL 宏，让操作系统使用默认的堆栈大小。现在，*flags* 应该是 TCL_THREAD_NOFLAGS 或 TCL_THREAD_JOINABLE，分别表示默认的行为和新创建的线程是可加入的。

如果另一个线程可以等待这个线程退出，这个线程就是可加入的。Tcl 提供了 Tcl_JoinThread 函数来完成这个任务。它需要两个参数，要加入的线程的 Tcl_ThreadID，以及指向一个整型数的指针，被加入的线程将退出代码存放在那里。Tcl_JoinThread 函数会阻塞到指定的线程退出。试图等待一个不可加入的线程，或者已经有线程在等待的线程，都会产生错误。可以等待一个已经退出的可加入的线程，系统会为 Tcl_JoinThread 调用保存必要的信息。这意味着不对一个可加入的线程调用 Tcl_JoinThread 就会导致内存泄漏。

提示： Windows 现在并不支持可加入的线程。因此，Tcl_JoinThread 函数在该平台上不被支持，而 Tcl_CreateThread 的 TCL_THREAD_JOINABLE 标志会被忽略。

46.5 终止线程

Tcl 提供了 Tcl_ExitThread 函数来终止当前线程。作为终止进程的一部分，Tcl 删除了所有的解释器和与线程相关的其他 Tcl 资源。您还可以使用 Tcl_CreateExitHandler 函数为线程注册一个退出处理器。退出处理器可以完成您希望在线程退出时所做的工作，例如刷新缓冲器，释放全局内存，等等。

Tcl 还提供了一个 Tcl_FinalizeThread 函数，它删除线程中的 Tcl 解释器以及其他资源，调用线程退出处理器，但是不终止线程。注意 Tcl_ExitThread 调用了 Tcl_FinalizeThread 作为它的操作的一部分，因此调用 Tcl_ExitThread 时就不必再显式地这样做了。

46.6 互斥体

互斥体(互斥禁止体的简称)用于限定同一时间只能有一个线程访问一块代码。如果有一段代码必须被“原子化”地执行——您不希望它被中断，或在同一时间被多于一个线程执行——那么最好使用互斥体。Tcl 有一个平台无关的宏，用于创建互斥体。

```
TCL_DECLARE_MUTEX(myMutex);
```

这个宏还保证了代码在无线程支持的编译时能够正确处理互斥体。

一旦创建了一个互斥体，就可以使用 Tcl_MutexLock 和 Tcl_MutexUnlock 来保护代码中的关键部分。

```
TCL_MutexLock(&myMutex);
    /* ... Critical section ... */
```

```
Tcl_MutexUnlock(&myMutex);
```

如果 Tcl 没有编译为支持线程，则这些函数没有任何影响。对于多线程的应用程序，如果一个线程锁定了这个互斥体，任何其他想要锁定这个互斥体的线程都必须等待这个线程解锁，然后才能进行锁定。这意味着如果由互斥体保护的这段代码经常由多个线程执行，程序效率会受到影响(当然这比数据冲突还是好得多了！)，因此，在创建代码架构时尽量不要让太多的资源成为共享资源。不再需要互斥体时，Tcl_MutexFinalize 函数释放与互斥体有关的所有资源。

46.7 条件变量

条件变量(也简称为条件)是多线程编程中的另一个常用工具。当一个线程需要另一个线程知道有关共享资源的状态信息的时候，就需要用到它们。

条件变量和互斥体联合使用以保护共享资源。在线程等待条件变量之前，它首先必须锁定保护该共享资源的互斥体。然后它调用 Tcl_ConditionWait 等待其他线程发来的通知；这也自动地解锁互斥体。然后发出通知的那个线程锁定互斥体，调用 Tcl_ConditionNotify 通知等待中的线程。一旦它解锁互斥体，等待线程中的 Tcl_ConditionWait 最终返回，让线程“醒来”。返回时，Tcl_ConditionWait 为接收到通知的线程自动锁定相关的互斥体。

Tcl_ConditionWait 获取的参数包括一个指向 Tcl_Condition 结构体的指针、一个指向互斥体的指针以及一个时间值，该时间值告诉 Tcl 在放弃等待之前等待多长时间，如果其值为 NULL，则表示永久等待。Tcl_ConditionNotify 函数获取一个指向 Tcl_Condition 的指针作为它唯一的参数。不再需要 Tcl_Condition 后，它使用的资源可以通过调用 Tcl_ConditionFinalize 释放。

在等待线程中的一种复杂的情形是它可能接收到一个虚假的“醒来”通知。这通常是因为有多个工作线程在同时等待同一个共享资源，例如一个消息队列。当消息到达的通知信号发出时，第一个线程继续了对消息进行处理的过程。而其他的工作线程有机会对通知信号作出反应时，消息已经被消耗掉了，于是它们仍然无事可做。这类情况下，对 Tcl_ConditionWait 的调用通常在一个循环中进行，这个循环检查 Tcl_ConditionWait 返回时共享资源的状态，如果需要继续等待，则再次调用它。

下面是来自 Tcl 自身的示例，源于 Tcl_InitNotifier 函数，这就是多线程版的 Tcl 如何为它的事件循环建立通知线程。

```
/* Tcl_InitNotifier: */
...

Tcl_MutexLock(&notifierMutex);
if (notifierCount == 0) {
    if (TclpThreadCreate(&notifierThread,
                         NotifierThreadProc, NULL,
                         TCL_THREAD_STACK_DEFAULT,
                         TCL_THREAD_NOFLAGS) != TCL_OK) {
    Tcl_Panic("Tcl_InitNotifier: unable to start notifier thread");
    }
}
```

```
notifierCount++;

/*
 * Wait for the notifier pipe to be created.
 */

while (triggerPipe < 0) {
    Tcl_ConditionWait(&notifierCV, &notifierMutex, NULL);
}
Tcl_MutexUnlock(&notifierMutex);
...

/* NotifierThreadProc: */
...
Tcl_MutexLock(&notifierMutex);
triggerPipe = fds[1];

/*
 * Signal any threads that are waiting.
 */

Tcl_ConditionNotify(&notifierCV);
Tcl_MutexUnlock(&notifierMutex);
...
```

这段代码中，Tcl 首先锁定了 notifierMutex。如果没有通知器存在，它通过内部私有函数 TclpThreadCreate 函数发起一个通知器线程。这个新的线程会进行处理直到它遇见 Tcl_MutexLock(&notifierMutex)，然后进行等待。第一个线程继续进行，处理 Tcl_ConditionWait，其副作用就是解锁了 notifierMutex，使得第二个线程可以继续进行，这时第二个线程继续进行，直到它处理 Tcl_ConditionNotify 和 Tcl_MutexUnlock，将控制权交还给第一个线程，最后第一个线程解锁互斥体并继续工作。如您所见，使用线程的程序的确比较复杂。

46.8 其他

要取得当前线程的线程 ID，应使用以下函数：

```
Tcl_ThreadId Tcl_GetCurrentThread();
```

要取得可用于线程独有数据的存储空间，使用 Tcl_GetThreadData：

```
void *Tcl_GetThreadData(Tcl_ThreadDataKey *keyPtr,
                int *size)
```

这个函数返回一个指向线程私有数据块的指针。它的参数是一个由所有线程共享的关键字以及该存储块的大小 *size*。在各个线程第一次请求存储块时，它会被自动分配空间，初始化为全零。它在一定程度上与 Tcl_Alloc 相似，不过如非必要，尽量不要使用它。这里分配的存储空间仅在线程退出时才由 Tcl_FinalizeThread 释放。

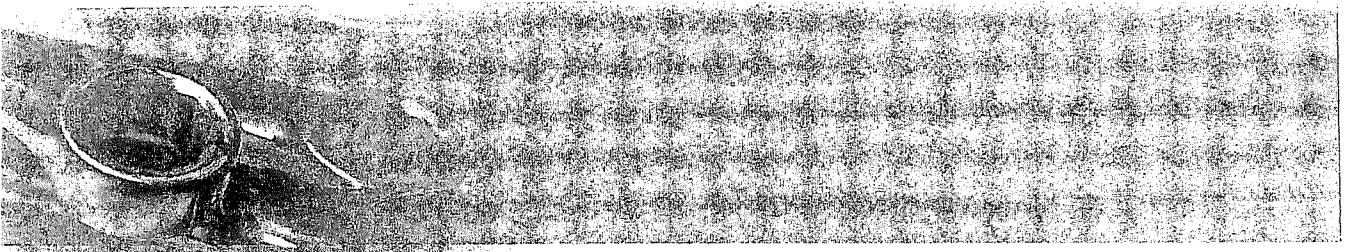
Tcl 自身在 ThreadSafeLocalTime 函数(由 Kevin Kenny 编写)中对 Tcl_GetThreadData 的使用是很有启发性的。关键字 tmKey 在文件中声明为静态。

```
static Tcl_ThreadDataKey tmKey;
```

```
/* And then, in the ThreadSafeLocalTime function: */

static struct tm *
ThreadSafeLocalTime(timePtr)
    CONST time_t *timePtr; /* Pointer to the number of
                              seconds since the local
                              system's epoch */
{
    /*
     * Get a thread-local buffer to hold the returned
     * time.
     */
    struct tm *tmPtr =
        (struct tm *) Tcl_GetThreadData(
            &tmKey, (int) sizeof (struct tm));
#ifdef HAVE_LOCALTIME_R
    localtime_r(timePtr, tmPtr);
#else
    Tcl_MutexLock(&clockMutex);
    memcpy((VOID *) tmPtr, (VOID *) localtime
        (timePtr), sizeof(struct tm));
    Tcl_MutexUnlock(&clockMutex);
#endif
    return tmPtr;
}
```

存储空间由 Tcl_GetThreadData 分配，然后由 localtime_r 函数，或者由互斥体保护下的 localtime 函数填写内容。



第47章 构建Tcl及其扩展

本章讨论在Unix(包括Linux和各种BSD系统)、Mac OS以及Windows三种平台上，如何编译Tcl、Tcl扩展包以及嵌入Tcl的代码。对很多人来说，编译Tcl并不是必需的，因为有很多途径得到二进制的发行版，详情参见附录A。然而，如果要把Tcl整合进自己的程序，就需要熟悉这个过程。构建扩展包并加载到Tcl通常是最好的方法，本章介绍如何以可移植的方式来完成这些工作。

47.1 构建Tcl和Tk

Tcl和Tk的源代码可以在SourceForge找到，它的项目主页为http://tcl.sourceforge.net。Tcl和Tk在该站点中作为相互独立的项目保存。您可以通过文件发布链接下载发行版的源代码，或者根据站点的指导匿名访问CVS档案库。发行版的源代码还可以从以下地址获取：http://www.tcl.tk/software/tcltk/download.html。如果从CVS中签出资源，顶层资源目录的名称为tcl和tk。如果下载了某个发行版本的源码，顶层目录的名称中包含版本号，例如tcl8.5.5。

每一个源码发布包都在顶层目录中有一个README文件，提供常用信息以及有关Tcl的更多信息的参考文献。还有一系列的子文件夹，包括：

- doc/——Tcl/Tk的参考文档。
- generic/——应用于所有支持的平台的源代码。
- library/——由Tcl/Tk使用的Tcl脚本库。
- macosx/——Macintosh特有的源代码、make文件以及Xcode项目文件。
- tests/——Tcl/Tk测试套件。
- tools/——生成Tcl发布包所用的工具。
- unix/——Unix特有的源代码、配置文件和make文件。
- win/——Windows特有的源代码，以及与Microsoft Visual C++共同使用的make文件。

每一个重要的子目录中都包含它自己的README文件，介绍目录中的内容。macosx、unix以及windows目录中的README文件还提供了在相应的平台上构建Tcl或Tk的最新信息。建议阅读这些README文件，检查是否出现了新的依赖条件、系统要求或构建过程。

47.1.1 在 Unix 中构建 Tcl 和 Tk

在 Unix 中构建 Tcl 的过程相当直接。Tcl 使用 GNU Autoconf 处理系统构建时的依赖条件。因此，可以使用典型的过程配置、构建、安装 Tcl 及 Tk。

```
# cd to tcl/unix or tk/unix
./configure
make
make install
```

提示： *在进行 make install 之前，可以运行 make test 来运行 Tcl 或 Tk 的测试套件，检验它们在您系统上的安装。注意，运行测试套件的耗时可能相当长。*

您可以为 configure 提供标准选项，例如--prefix 用于指定安装目录，--exec-prefix 用于为架构特有的文件指定安装目录。--prefix 选项适用于在同一个系统中安装多个版本的 Tcl，或出于测试目的而要在一个系统中互不干扰地安装多个同一版本的 Tcl/Tk 的情况。Unix 系统中默认的前缀是/usr/local，而 Tcl 通常也安装到/local 中(例如，在 Linux 中)。有关标准 configure 选项的更多信息，参见 Autoconf 文档(在线资源之一：http://www.gnu.org/software/autoconf)。

另外，Tcl 和 Tk 为 configure 提供了一些与包相关的选项，影响对它们的编译方式。大部分选项都遵循“启用/禁用某功能”的语法，形如--enable-*feature* 和--disable-*feature*。影响较大的一些选项包括：

- --enable-shared
  如果启用就编译为共享库(默认为启用)，否则编译为静态库。
- --enable-threads
  如果启用就按支持多线程的方式编译；默认为禁用(现在很多预编译的 Tcl 发行版本设置为启用)。
- --enable-64bit
  启用 64 位支持(如果可用)；默认为禁用。
- --enable-symbols
  如果启用就编译为含调试符号的版本；默认为禁用。
- --with-tcl=*location*
  (仅用于 Tk 的选项)指明构建了 Tcl 的目录。默认情况下，Tk 构建系统假设这个目录是../../tcl<版本号>，这里<版本号>与要构建的 Tk 的版本号相同。如果不是这种情况，必须用--with-tcl 指明适合的目录，才能正确地构建 Tk。

在构建 Tcl 时还会产生一个名为 tclConfig.sh 的文件，该文件由 Tcl 扩展架构(参见 47.2 节)使用，用于保存扩展包编译时需要的 Tcl 安装信息。它是一个外壳脚本，定义了很多变量：如 Tcl 版本，用于编译 Tcl 的 C 编译器，编译器和链接器标志，包含文件的位置，库以及路径等。构建 Tk 时也会产生一个类似的文件，tkConfig.sh。

47.1.2 在 Mac OS 上构建 Tcl 和 Tk

在 Macintosh 上构建 Tcl 需要的最低版本是 Mac OS X 10.1，不再支持更早版本的系统。(Tcl/Tk 8.4 是最后一个支持“经典”Mac OS 系列系统的版本。)您的系统中还必须安装苹果的开发工具。

在 Macintosh 上构建 Tcl/Tk 的一个方法是使用 Unix 的构建系统，在 tcl/unix 及 tk/unix 中。这种情况下，安装过程与 47.1.1 节中完全相同。以下 configure 选项仅应用于在 Mac OS 中的构建。

- --enable-corefoundation
  使用苹果的核心基础框架；默认为启用。
- --enable-framework
  在 Mac OS X 框架中打包共享库；默认为禁用。
- --enable-aqua
  (仅应用于 Tk)使用 Aqua 窗口系统，而非 X11；默认为禁用。要求 Tcl 和 Tk 都是在核心基础框架下构建的。

另一个方法是使用 tcl/macos 和 tk/macos 目录中的 GNUmakefile，可以用它直接构建和安装 Tcl 和 Tk。下面是一个标准的安装过程。这里假设 Tcl 和 Tk 源目录放在同一个目录中，为了更具一般性，假设存在一个外壳变量 ver，其内容是版本字符串(例如 8.5.5)，如果从 CVS 构建，则是空字符串。要构建这些包，首先需要打开一个终端，进入存放它们的那个文件夹，然后执行以下命令：

```
make -C tcl${ver}/macosx
make -C tk${ver}/macosx
```

然后，将这些包安装到系统的根卷中(这需要管理员密码)。

```
sudo make -C tcl${ver}/macosx install
sudo make -C tk${ver}/macosx install
```

要安装到别的位置，例如 HOME 文件夹(例如，您没有管理员权限，或者您希望有多个 Tcl 版本用于检测)。

```
make -C tcl${ver}/macosx install INSTALL_ROOT="${HOME}/"
make -C tk${ver}/macosx install INSTALL_ROOT="${HOME}/"
```

还有其他可用的 make 目标和选项。更多信息请参考 README 文件。

最后，macosx 子目录还包含了一些项目文件，您可以使用不同版本的苹果 Xcode 集成开发环境。再次提醒，使用哪些项目文件和如何配置等信息请参考 README 文件。

47.1.3 在 Windows 中构建 Tcl 和 Tk

在 Windows 中构建 Tcl 和 Tk 有不少选择，取决于您安装了什么工具。

如果安装了 Microsoft Visual C++，那么可以使用 nmake，利用 tcl/win 和 tk/win 目录中的 makefile.vc 文件。每一个文件中都有对可用的目标和选项的注释。您也可以使用 tcl/win

目录中包含的 Microsoft Developer Studio 的工作空间和项目文件。

另外，在 Windows 中还可以使用开源的 MinGW 及 MSYS 工具来构建 Tcl 和 Tk。(这些工具的更多信息以及下载信息参见 http://www.mingw.org。)这些工具提供了“Windows 下的最小 GNU”，包含标准 GNU 编译器和构建工具。对于这种选项，可以使用 tcl/win 和 tk/win 中的 configure 脚本，按照 Unix 中同样的 configure、make 和 make install 步骤。如果选择了这些方法，可以使用的 configure 选项与 47.1.1 节中相同。

47.2 Tcl 扩展架构(TEA)

如果用 C 语言编写自己的扩展包，可以使用喜欢的任何构建系统来编译应用，编写自己的构建文件，等等。不过，如果希望把扩展包发布为可在多种平台上运行的版本，想设计一个通用系统来处理所有的目标平台却是非常困难的。

Tcl 有一个用于构建扩展包的通用系统，称为 TEA，意为“Tcl 扩展架构”(Tcl Extension Architecture)。TEA 基于 GNU Autoconf 工具，提供了一个用于构建扩展包的系统。TEA 还为扩展包的开发和发布给定了最佳实用方针。

为扩展包使用 TEA 的起点，是一个示例扩展包，它设计用于演示 TEA 系统，同时作为扩展包开发的模板。这个示例扩展包在 SourceForge 中作为 Tcl 项目的一部分(http://tcl.sourceforge.net)。您可以按照站点的指示，从 CVS 档案库中匿名地获取它。其模块名称为 sampleextension。

提示： 这个示例扩展包还包含一个名为 tea 的子文件夹，包含有关 TEA 的详细信息以及开发可移植扩展包的一些指导。因为 TEA 一直在进步，这里的文档包含的可能是有关 TEA 的最新信息。

TEA 的核心是 Autoconf 和它产生的 configure 脚本。Autoconf 获取两个模板文件，configure.in 和 Makefile.in，以及一个 M4 宏定义集合(由名为 aclocal.m4 的文件提供)；生成一个适当的 configure 脚本，人们可以用它来构建扩展包。与示例扩展包在一起的还有注释完整的通用 configure.in 和 Makefile.in 文件，您可以修改它们，用于自己的扩展包，以及可以包含到您的 aclocal.m4 文件的 M4 宏定义集，用于实现构建 TEA 的基础。实际上，如果从示例扩展包开始构建，只需编辑 configure.in 和 Makefile.in，然后在包含了您的扩展文件的目录中运行 autoconf，为您的扩展包生成 configure 脚本。

提示： 除了示例扩展包以外，TEA 假定系统已经安装了开发工具，包括 Autoconf 在内。要在 Windows 的开发系统中使用 Autoconf 创建 configure 脚本，必须安装开源的 MinGW 和 MSYS 工具以及 Autoconf。(有关这些工具及其下载的更多信息参见 http://www.mingw.org。)

只是开发系统需要 Autoconf 来为扩展包创建 configure 脚本。而扩展包的用户在系统上构建时，只需要安装适当的开发工具就行了。然后他们可以使用您提供的 configure 脚本在他们的系统上构建扩展包。

在运行 configure 脚本之后，就可以构建扩展包，您可以用 make 命令来完成这一步。安装是由 make install 处理的。其他的目标包括：make test，如果您已经定义了测试并且提供了运行它们的方法；还有特别有用的 make dist，将您的源码和 configure 文件一起打包为一个发布版(默认会打包成一个 gzip 压缩的 tar 文件)，为发布做好准备。

47.2.1　TEA 标准配置选项

TEA 为它生成的 configure 脚本定义了一个很大的选项集合，扩展包的用户在他们的系统中进行构建时可以用这些选项来配置扩展包。大多数选项遵循“启用/禁用功能”这样的标准语法：--enable-*feature* 和--disable-*feature*。下面是较为重要的一些选项。

- --enable-shared
  如果启用(默认)，则按照共享库编译，否则编译为静态库。
- --enable-threads
  如果启用(默认)，则按照支持多线程编译。
- --enable-64bit
  启用 64 位支持(如果可能)；默认为禁用。
- --enable-symbols
  如果启用，则编译为含有调试符号的版本；默认为禁用。
- --with-tcl=*location*
  指明构建了 Tcl 的目录。这允许您为多种平台多个版本的 Tcl 构建扩展包。如果系统没有安装多个版本的 Tcl，Autoconf 通常可以自动获得这个信息。
- --with-tk=*location*
  指明构建了 Tk 的目录。只在扩展包需要使用 Tk C API 时才需要这个选项。如果系统没有安装多个版本的 Tk，Autoconf 通常可以自动获得这个信息。
- --with-tclinclude=*location*
  指明 Tcl 包含文件所在的目录(最显著的如 tcl.h)。
- --with-tcllib=*location*
  指明 Tcl 库所在的目录(最显著的如 libtclstubs.a)。
- --prefix
  确定安装的根目录。默认为构建 Tcl 时给出的值。
- --exec-prefix
  确定安装系统特有文件的根目录。默认为构建 Tcl 时给出的值。

47.2.2　TEA 扩展包的目录结构

TEA 推荐一个标准的目录结构，用于在一个父目录中组织扩展包。

- demos/——扩展包的演示脚本。
- doc/——扩展包的文档。
- generic/——适用于所有支持的平台的源码。

- library/——支持脚本，例如组件的默认绑定。
- macosx/——Macintosh 特有的源代码。
- tests/——扩展包的测试套件。
- unix/——Unix 特有的源代码。
- win/——Windows 特有的源代码。

另外，TEA 的示例扩展包提供了如下文件和目录。

- tclconfig/——包含 tcl.m4 的目录，该文件定义了 TEA 使用的 Autoconf 的宏。该目录还包含 install -sh：一个用于向安装位置复制文件的程序。
- tea/——包含当前版本的 TEA 的文档的目录。
- aclocal.m4——Autoconf 生成最终的 configure 脚本时使用的输入文件。
- configure.in——Autoconf 生成最终的 configure 脚本时使用的模板文件。
- Makefile.in——由 configure 生成最终的 Makefile 时使用的模板文件。
- pkgIndex.tcl.in——由 configure 生成最终的 pkgIndex.tcl 文件时使用的模板文件。

make install 命令将 TEA 包安装到--prefix 和--exec-prefix 目录。例如，对于 pkgIndex.tcl 文件和编译的库文件，安装程序会在$*prefix*/lib 下创建一个子目录，目录名为包的名称加上版本号，如 thread2.6.5。默认情况下，文件会被安装到下列位置。

- $*exec-prefix*/lib/$*package*$*version*/——pkgIndex.tcl 文件和编译的库文件。
- $*exec-prefix*/bin/——二进制可执行文件以及 Windows 中所依赖的.dll 文件。
- $*prefix*/include/——C 头文件。
- $*prefix*/man/——来自 doc 源目录的 Unix 的手册文件。

47.2.3 定制 aclocal.m4 文件

Autoconf 的输入之一就是 aclocal.m4 文件，位于源目录的根目录中。这个文件定义了处理 configure.in 文件以生成 configure 脚本时所用的 M4 宏。示例扩展包提供的 tclconfig/tcl.m4 定义了 TEA 构建系统所需要的所有的 M4 宏，因此大多数情况下您的 aclocal.m4 文件可以只有以下一行代码：

```
builtin(include, tclconfig/tcl.m4)
```

如果希望为扩展包进一步增加 configure 脚本选项，或者定制构建系统，可以在这个文件中添加更多的宏定义。

47.2.4 定制 configure.in 文件

configure.in 文件是一个包含了一系列 M4 宏的模板，决定了生成的 configure 脚本的行为。示例扩展包提供了该文件的一个很好的演示，大多数情况下您可以就在那个文件上根据需要进行修改。一般来说，宏的顺序是很重要的，搞乱了顺序可能会导致跟踪困难。

configure.in 文件中的第一个宏是 AC_INIT，它初始化环境，指明扩展包的名称和版本号。

```
AC_INIT([tclifconfig], [0.1])
```

下一步初始化 TEA 变量，指明 TEA 的版本，指导 Autoconf 使用 tclconfig 子目录中更多的构建脚本和定义。

```
TEA_INIT([3.7])
AC_CONFIG_AUX_DIR(tclconfig)
```

接下来的两个宏负责找到和定位 tclConfig.sh 文件，该文件在您构建 Tcl 时创建。它包括的信息有：Tcl 的版本；用于编译 Tcl 的 C 编译器；编译器和链接器标志；包含文件和库的位置，等等。TEA 使用试探法在包含 tclConfig.sh 文件的目标系统中找到安装好的 Tcl，不过构建扩展包的时候，用户也可以用--with-tcl 选项为 configure 指定一个特定的目录。当在多平台多 Tcl 版本的情况下构建扩展包时这一选项特别有用。以下宏没有参数：

```
TEA_PATH_TCLCONFIG
TEA_LOAD_TCLCONFIG
```

如果扩展包还使用了 Tk C API，就需要两个宏来找到和定位在构建 Tk 时生成的那个类似的 tkConfig.sh 文件。

```
#TEA_PATH_TKCONFIG
#TEA_LOAD_TKCONFIG
```

下面两个宏是处理--prefix 参数所需要的，用于确定适当的 C 编译器和它的选项。

```
TEA_PREFIX
TEA_SETUP_COMPILER
```

下面的宏则是扩展包特定的。最重要的是用空格隔开的需要编译的 C 源程序文件列表；如果所在目录是标准 TEA 源文件目录，就不必指明文件所在的子目录。

```
TEA_ADD_SOURCES([sample.c tclsample.c])
```

如果您的扩展包需要用到另一个扩展包或应用程序，可能需要提供一个公共头文件的列表，指明您要在目标系统中用 make install 命令安装的文件。

```
TEA_ADD_HEADERS([])
```

如果需要，可以列出编译扩展包所需要的额外的库，包含所需的包含文件的额外的目录以及额外的编译器标志。

```
TEA_ADD_LIBS([])
TEA_ADD_INCLUDES([])
TEA_ADD_CFLAGS([])
```

您还可以提供一个列表，指明作为您的扩展包的一部分的额外的 Tcl 源文件。

```
TEA_ADD_TCL_SOURCES([])
```

在下一部分，您可以提供平台特有的配置指令。在条件部分，您可以使用对 TEA_ADD_*宏的调用，列出平台特定的源文件、依赖库等。

```
if test "${TEA_PLATFORM}" = "windows" ; then
    ....
else
    ....
fi
```

然后您需要指明扩展包所用的 Tcl 和 Tk 头文件。至少您会需要 Tcl 公共头文件(tcl.h)。如果需要用 Tk C API，就需要 Tk 公共头文件(tk.h)以及定位 X11 头文件的宏。如果可能，应该尽量避免使用私有头文件，因为 API 及数据结构可能会变动而不发布通知，Tcl 核心小组一直在努力使公共 C API 的接口尽量稳定。

```
TEA_PUBLIC_TCL_HEADERS
#TEA_PRIVATE_TCL_HEADERS
#TEA_PUBLIC_TK_HEADERS
#TEA_PRIVATE_TK_HEADERS
#TEA_PATH_X
```

下一部分宏进行配置中的标准处理过程——检测是否要以支持多线程的方式进行编译，构建共享库还是静态库，包含调试符号还是不包含，以及一些通用编译器选项。

```
TEA_ENABLE_THREADS
TEA_ENABLE_SHARED
TEA_CONFIG_CFLAGS
TEA_ENABLE_SYMBOLS
```

第 36 章讨论了 Tcl 的占位符机制，允许用某个版本的 Tcl 编译的扩展包用于更新版本的 Tcl。扩展包应该几乎总是使用占位符机制，以求获得与 Tcl 版本之间的独立性。另一方面，如果您在把代码编译为链接到 Tcl 的库，而不是创建扩展包，就不应该使用占位符——使用它们毫无意义。使用下面的声明启用 Tcl C API 的占位符机制，以及 Tk C API 占位符机制(如果需要)。

```
AC_DEFINE(USE_TCL_STUBS)
#AC_DEFINE(USE_TCL_STUBS)
```

然后是在库的构建时，用于生成命令行的标准 TEA 宏。

```
TEA_MAKE_LIB
```

下面的宏指定了 tclsh 的名称和/或可执行的 wish，仅用于构建系统响应 make test 命令运行测试的情况。

```
TEA_PROG_TCLSH
#TEA_PROG_WISH
```

这个文件的最后一行列出了 configure 脚本需要创建的所有文件。

```
AC_OUTPUT([Makefile pkgIndex.tcl])
```

这里列出的每一个文件都必须在同一个目录中有对应的模板文件，扩展名为.in。至少您需要一个 Makefile.in 文件。您还可以让 Autoconf 创建 pkgIndex.tcl 文件，如示例扩展包所演示的那样。对于跨平台的扩展包，您可能会发现把不同的 makefile 放在不同的源子目录下进行维护要更容易一些，下面这段代码就在 generic，unix，还有 win 子目录中分别根据各自的模板创建了更多的 makefile 文件。

```
AC_OUTPUT([
Makefile
generic/Makefile
unix/Makefile
win/Makefile
])
```

还有一些可用的 TEA 宏本节未能介绍，而且您还可以使用标准 Autoconf 宏来管理构建过程。有关这些功能的更多信息详见示例扩展包中的 TEA 参考文档。

47.2.5 定制 Makefile.in 文件

Makefile.in 文件是生成 Makefile 文件的模板。要为 TEA 的使用从零开始创建 Makefile.in 文件是一件困难而繁琐的工作。推荐的做法是使用示例扩展包提供的 Makefile.in 文件进行修改，这个文件大多数情况下略作修改即可使用。为了超越基本的构建过程，可能需要定制它，例如将它作为一个运行工具，在不同的平台以不同的格式构建您的文档(例如 Unix 的手册文件与 Windows 的帮助文件)或是执行定制的清理操作。更多信息详见示例扩展包中的 TEA 参考文档。

47.2.6 在 Windows 中构建扩展包

TEA 是基于 GNU 构建系统的自动工具的。开源的 MinGW 和 MSYS 工具为 Windows 系统提供了构建 TEA 扩展包的必要环境(参见 http://www.mingw.org)。如果您觉得目标用户会希望在 Windows 系统中用 MinGW 和 MSYS 构建扩展包，可以使用与 Unix 相同的 configure 脚本；Windows 用户在他们的系统中进行同样的 configure—make—make install 过程。

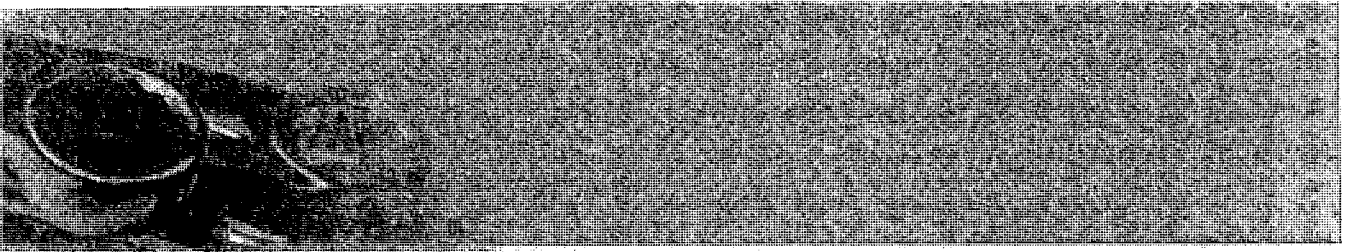
如果愿意，也可以为扩展包创建和发布一个 Visual C++的 make 文件。示例扩展包在 win/makefile.vc 中提供了一个例子。它被设计为尽可能通用，但是仍然需要一些额外的维护，以便与 TEA configure.in 和 Makefile.in 文件同步。使用 Visual C++ make 文件的指导写在 makefile.vc 文件的开头部分。

47.3 构建嵌入的 Tcl

当 Tcl 嵌入其他系统中时，可能需要把 Tcl 构建系统集成到您的程序中。如果嵌入了 Tcl 的程序使用自动构建工具，完成构建应该不太困难。基本的方法是包含 tcl.m4 文件，使用 TEA 宏，从而能够获得如何编译 Tcl 的信息。

第Ⅳ部分　附　　录

- ⊙ 附录A　安装Tcl和Tk
- ⊙ 附录B　扩展包和应用程序
- ⊙ 附录C　Tcl资源
- ⊙ 附录D　Tcl源码发布许可



附录A 安装Tcl和Tk

Tcl 和 Tk 的源代码和参考文档都是免费的。Tcl 和 Tk 遵循 BSD 风格许可，该许可条款具有极高的适应性，允许不付费地将 Tcl 和 Tk 用于商业程序，也不要求使用它的程序必须开源。本附录介绍为您的本地系统获取 Tcl 和 Tk 发行版的一些方法。

A.1 版本

Tcl 和 Tk 分别发行，每一个发行版都有一个版本号。一个版本号由两个或多个由句点分开的整数组成，例如 8.5.3 或 8.4。第一个数字是主版本号，第二个数字是次版本号。如果有第三个数字，它代表的是发行包号，只在进行了错误修正之后才会有这个号；一个新的发行包号并不表示功能有显著增强。每一版新发行的 Tcl 和 Tk 都有自己的版本号。如果新版本与上一个版本兼容，则称为次发布：次版本号增加，主版本号保持不变。例如 Tcl 8.5 就是 Tcl 8.4 之后的次发布。次发布意味着为旧版本编写的 C 代码和 Tcl 脚本可以不加修改地在新版本中使用。如果新发行的版本包含了与上一个版本不兼容的修改，则称为主发布：主版本号增加，次版本号设为 0。例如，Tcl 8.0 就是 Tcl 7.6 之后的主发布。当您升级使用新的主发布版本时，需要对以前的 C 代码和 Tcl 脚本进行相应的修改。

尽管 Tcl 可以独立使用，每一版的 Tk 却是为特定的某发行版本的 Tcl 设计的。在 Tk 启动的时候，它会检测 Tcl 的版本号，确认它们可以正常工作。匹配的 Tcl 和 Tk 的版本号不一定相同，例如，Tk 3.6 就要求 Tcl 7.3。不过，现在发行的从 8.0 版开始的 Tcl 和 Tk，相互匹配的版本使用相同的版本号，以减少混淆。

A.2 Tcl 发布包

针对很多 Unix 和类 Unix 系统有已经预编译过的 Tcl/Tk 发布包(也有很多 Tcl 和 Tk 源程序)。一些操作系统在安装时，或指定为“开发者”模式安装时就会自动包含 Tcl 和 Tk。如果操作系统的安装包中没有 Tcl 和 Tk，或者希望升级到新版本的 Tcl/Tk，应该可以在档案库中找到适用于您的操作系统的官方 Tcl/Tk 发布包。您可以使用操作系统支持的任何包管理工具(例如 rpm、yum、dpkg、apt-get 等)为系统安装、升级或卸载 Tcl/Tk。

使用官方 Tcl/Tk 发布包的不利之处在于这会有所延迟——从新版本的 Tcl/Tk 的发布到获得针对您的操作系统的官方发布包之间会有一段时间。在同一个系统中安装多个版本的

Tcl/Tk，在共享网络文件系统中为多种平台安装与维护 Tcl/Tk 也会更加困难。

A.3 ActiveTcl

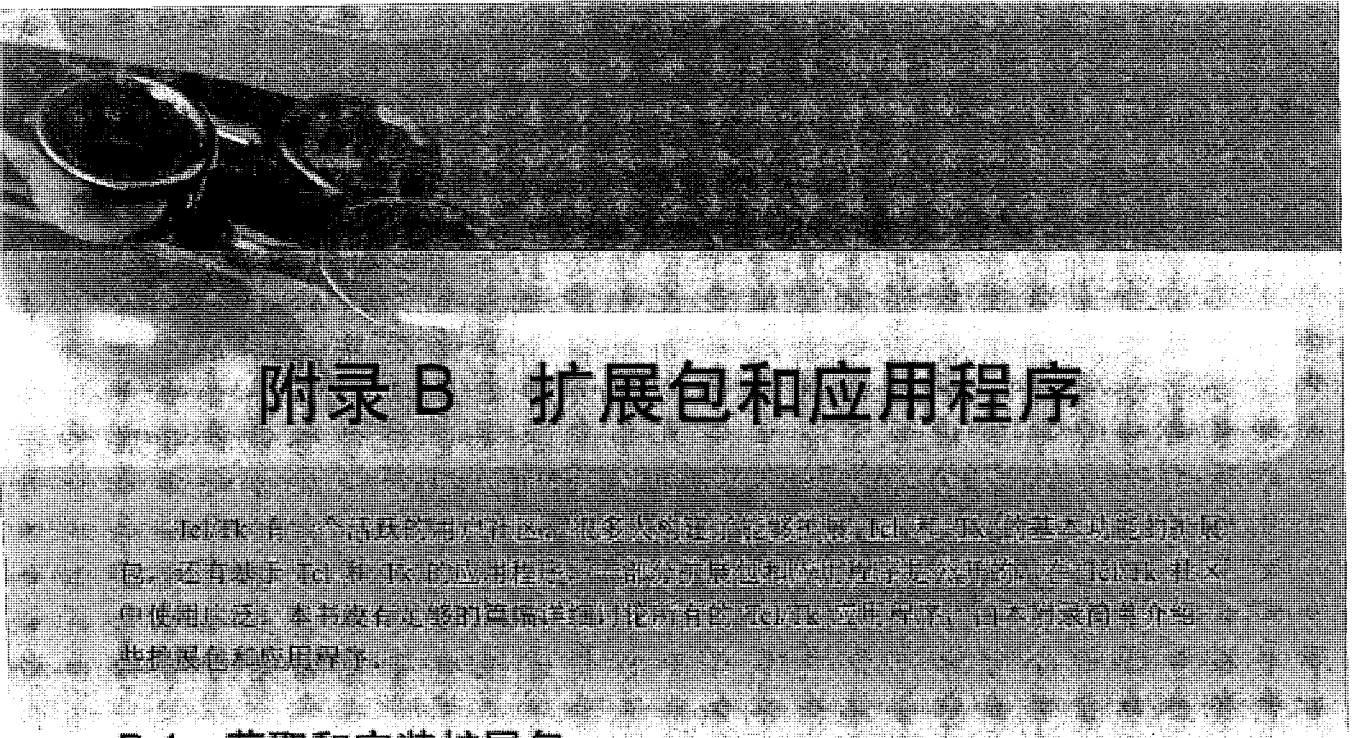
ActiveState 软件(http://www.activestate.com)是一个创建开发工具，为包括 Tcl 在内的一些动态语言提供服务和支持的公司。除了他们的商业产品，还提供了一款免费的 Tcl 预编译器，称为 ActiveTcl，其中还打包进一些流行的 Tcl 扩展包。ActiveTcl 可用于多种平台，目前包括 Windows、Linux、Mac OS X、Solaris、AIX 以及 HP-UX。如果仅关注 Tcl/Tk 的脚本开发，使用 ActiveTcl 是对操作系统安装 Tcl/Tk 的最简便的方法。

A.4 Tclkit

第 14 章讨论了使用 Tclkit 作为发布您基于 Tcl/Tk 的应用程序的工具。一个 Tclkit 就是一个单独的可执行文件，以很小的体积包含了对应的整个 Tcl 发行版(通常小于 2 M 字节)。您可以把 Tclkit 当作可执行的 Tcl 外壳使用，就像 tclsh 那样。预编译的 Tclkit 可用于各种平台。在 http://www.equi4.com/tclkit 和 http://wiki.tcl.tk/tclkit 处可找到更多关于 Tclkit 的信息，可在 http://www.equi4.com/tclkit/download.html 处下载 Tclkit。

A.5 用发布的源码编译 Tcl/Tk

Tcl 和 Tk 的源码保存在 SourceForge 处，项目主页为 http://tcl.sourceforge.net。在这个站点 Tcl 和 Tk 作为独立的项目管理。您可以用文件发布链接下载发行版的源码，或者根据站点的提示以匿名方式访问 CVS 档案库。发行的源码也可以从 http://www.tcl.tk/software/tcltk/download.html 处下载。编译和安装 Tcl 和 Tk 的详细指导参见第 47 章。



附录 B 扩展包和应用程序

Tcl/Tk 有一个活跃的用户社区，很多人创建了能够扩展 Tcl 和 Tk 的基本功能的扩展包，还有基于 Tcl 和 Tk 的应用程序。一部分扩展包和应用程序是免费的，在 Tcl/Tk 社区中使用广泛。本书没有足够的篇幅详细介绍所有的 Tcl/Tk 应用程序，因此附录简单介绍一些扩展包和应用程序。

B.1 获取和安装扩展包

有关可用的扩展包和应用程序的最好的信息来源就是 Tcl 开发者的维基(http://wiki.tcl.tk)。您可以在那里搜索感兴趣的主题，查找相关的扩展包和应用程序的链接。该维基网站还有一个网页列出了 Tcl/Tk 扩展包：http://wiki.tcl.tk/940。另一个很好的资源是由 Joe English 维护的“Great Unified Tcl/Tk Extension Repository”：http://www.flightlab.com/~joe/gutter/browse.html。上述页面为各个扩展包都提供了下载链接。作者们还常常在 comp.lang.tcl“世界新闻组”(见附录 C.1)处发布有关新版扩展包和应用程序的通知。

B.1.1 手动安装扩展包

一旦您下载了扩展包，就可以按照它附带的指导在系统上安装它。有一些扩展包提供了安装程序，而有一些扩展包您应该把它们复制到系统中的 Tcl 能够找到的适当位置。扩展包中应该有一个 README 文件，说明如何进行安装以及该扩展包的特点。

如果扩展包没有提供安装说明，通常就只需要把扩展包复制到系统中的 Tcl 默认查找扩展包的位置。如果扩展包是作为 Tcl 模块发布的，可以把文件复制到 14.6.2 节所介绍的目录中。如果扩展包发布为独立的一些文件，并包含 pkgIndex.tcl 文件，可以把包含扩展包文件的目录复制到 14.5.4 节所介绍的目录中。

B.1.2 为 ActiveState TEApot 档案库安装扩展包

ActiveState(http://www.activestate.com)开发了一个 Tcl 包管理系统，提供获取、安装以及更新 Tcl 扩展包和应用程序的简单方法。它允许您使用名为 teacup 的客户端应用程序浏览、搜索、安装由一个名为 teapot 的档案库服务器程序提供的扩展包。ActiveState 免费发布的 ActiveTcl 包含了 teacup 的应用程序和文档，您也可以从 http://teapot.activestate.com 单独下载。ActiveState 在 http://teapot.activestate.com 提供了一个公用 TEApot，包含了很多扩

展包和应用程序。您可以执行 teacup help 获取有关 teacup 的帮助及其选项。另外，您可以在 Tcl 开发者的维基网页 http://wiki.tcl.tk/teacup 处获得有关 teacup 的更多信息。ActiveTcl 包含的文档也讲述了如何用 teapot 建立自己的 TEApot 档案库(即为企业内部使用建立包含指定版本的开源及商用扩展包的档案库)。

teacup 应用程序包含了一系列子命令，用于管理一个或多个本地的安装档案库，作为您的 Tcl 安装所使用的扩展包来源。换句话说，如果您在安装档案库中安装了特定的扩展包，设置为使用这个档案库的 Tcl 外壳就可以成功地用 package require 命令加载该扩展包。大多数情况下您只使用一个安装档案库，称为默认安装档案库。您可以用 teacup default 命令查询及修改默认档案库。

```
    teacup default
⇒ /Library/Tcl/teapot
```

您还可以创建或删除更多的档案库，设置您的 Tcl 安装使用不同的档案库集合，详见 teacup 文档。

teacup list 命令列出该 TEApot 中可用的扩展包。您还可以指定扩展包名称以及可选地指明版本号。如果没有与该名称严格匹配的扩展包，该命令会以不区分大小写的子字符串匹配方式搜索候选的扩展包。要列出您在默认档案库中安装的扩展包，可以使用--at-default 选项。例如：

```
    teacup list --at-default file
⇒ entity  name     version    platform
  ------- -------  ---------  ----------
  package fileutil 1.13.4     tcl
  ------- -------  ---------  ------
  1 entity found
```

可以用 teacup describe 命令获得有关扩展包的描述。与 teacup list 类似，您还获得 TEApot 中可用的扩展包，也可以使用--at-default 选项要求列出已经安装的扩展包。例如：

```
    teacup describe Img
⇒ Entity      Img

    Description @ http://teapot.activestate.com
    The Img package provides support for several image formats
    beyond the standard formats in Tk (PBM, PPM, and GIF),
    including BMP, XBM, XPM, GIF(no LZW), PNG, JPEG, TIFF,
    and PostScript.
```

teacup install 命令安装指名的扩展包。您还可以指定最低或确切版本号；默认情况下会安装可用的最新版本。如果扩展包依赖于其他尚未安装的扩展包，teacup 会自动进行安装。您可以使用--dry-run 选项来模拟安装，这种情况下 teacup install 会报告它在安装时进行的步骤。

要更新安装档案库，使用 teacup update 命令。默认情况下，这个更新命令会安装 TEApot 可用的所有扩展包和应用程序的最新版本，并且从 TEApot 安装所有以前未安装的扩展包和应用程序的最新版本。您可以使用--only newer 选项指明只更新当前已经安装的扩展包，或者用--only uninstalled 选项指定只安装您尚未安装的扩展包和应用程序。选项--dry-run 也应用于模拟更新过程。

teacup remove 命令卸载指名的扩展包或应用程序。如果不指定扩展包或应用程序的名称而调用它，那么它会移除安装档案库中的所有项目。选项--dry-run 也应用于模拟删除过程。

您可以使用 teacup version 命令确定安装的 teacup 的版本号。要升级到最新的 teacup，执行 teacup upgrade-self 命令。

提示：如果希望在 ActiveTcl 之外的其他 Tcl 发布中使用 teacup，可以从 http://teapot.activestate.com 为您的平台下载一份副本。然后需要建立一个安装档案库(如果您还没有)，然后设置 Tcl 外壳启用档案库，然后将外壳链接到您创建的档案库。您可以用如下命令完成这一任务：

```
teacup create /path/to/your/repository
teacup setup /path/to/your/shell
teacup link make /path/to/your/repository /path/to/your/shell
```

B.2 TkCon 扩展控制台

TkCon 是一个与 Tcl 交互的增强控制台。尽管 wish 解释器已经有了一个骨架式的控制台，但 TkCon 在它的基础功能上扩展了很多用以辅助 Tcl 编程的功能，包括：

- 交互式命令编辑。
- 跨区保持的命令历史。
- 增强的历史搜索。
- 扩展的命令、变量以及路径。
- 电动字符匹配(与 emacs 相同)。
- 高亮命令和变量。
- 将输入、输出、错误以及命令捕获到日志文件。
- “热错误”，单击一个错误的结果会显示堆栈跟踪。
- 交互式调试功能。
- 扩展的配置选项。

TkCon，由 Jeff Hobbs 开发，完全由 Tcl/Tk 编写，可嵌入一个 Tcl/Tk 应用程序以提供控制台支持。更多信息参见 TkCon SourceForge 页面(http://tkcon.sourceforge.net)以及 Tcl 开发者维基的 TkCon 页面(http://wiki.tcl.tk/tkcon)。

B.3 标准 Tcl 库：Tcllib

Tcllib，也称作标准 Tcl 库，包括了一些独立的包，提供了很多常用的功能，包括：

- 网络协议的实现，包括电子邮件(POP3 和 SMTP)、域名解析(DNS)、文件传输(FTP)、目录(LDAP)、世界新闻组(NNTP)、聊天(IRC)以及网络时间(NTP)。
- 解析和生成 HTML 网页，生成 Java 脚本，编写 CGI 脚本，使用 URL 和 MIME

扩展的支持。

- CRC 校验和、DES、MD4、MD5、SHA1 以及基于 64 位的编码的加密和校验的支持。
- 日志、性能跟踪以及文档生成等编程工具。
- 数据结构的实现，包括堆栈、队列、矩阵、图和树。
- Snit 面向对象框架。
- 文本处理工具，包括 CSV 处理。
- 先进的数学函数。

更多信息参见 Tcllib SourceForge 页面(http://tcllib.sourceforge.net)，以及 Tcl 开发者维基的 Tcllib 页面(http://wiki.tcl.tk)。

B.4 Img 提供的额外的图形格式

Img 是一个 Tk 扩展包，支持了很多 Tk 本身不支持的图形格式。Img 支持的格式包括 BMP、GIF、ICO、JPEG、Pixmap、PNG、PPM、PostScript、SGI、Sun、TGA、TIFF、XBM 以及 XPM。Img 扩展包由 Jan Nijtmans 开发。更多信息参见 Tcl 开发者维基的 Img 页面(http://wiki.tcl.tk/img)以及 Img SourceForge 页面(http://sourceforge.net/projects/tkimg)。

B.5 Snack 提供的声音支持

Snack 声音扩展包增加了用于记录和播放音频的命令。Snack 支持内存声音对象、基于文件的音频、音频流以及后台音频进程。它能处理的文件格式包括 AIFF、AU、MP3、NIST/Sphere 以及 WAV。Snack 由 Kåre Sjölander 开发。更多信息参见 Snack 的主页 http://www.speech.kth.se/snack，以及 Tcl 开发者维基的 Snack 页面(http://wiki.tcl.tk/snack)。

B.6 面向对象的 Tcl

历史上对 Tcl 的抱怨之一是它没有原生的面向对象的结构。不过，这也导致人们开发了一些面向对象的扩展包，这些扩展包各有独特的功能和优势。

[incr Tcl]，也称为 Itcl(这个名称来自 tcl 加上“加号”，就像 C++的名称一样)，实现了与 C++非常相似的框架。您可以定义包含数据成员和方法的类，可以是公共的、受保护的、私有的，各有不同的可用性。[incr Tcl]支持继承和多重继承。而且，您可以在[incr Tcl]和[incr Tk]中使用面向对象的 Tk 组件。[incr Tcl]由 Michael Mclennan 创建。更多信息参见 Tcl 开发者维基的[incr Tcl]页面(http://wiki.tcl.tk/62)以及[incr Tcl] SourceForge 页面(http://incrtcl.sourceforge.net/itcl)。

XOTcl，或者称为扩展对象 Tcl，是早期的一个名为 OTcl 的面向对象扩展包的改进。XOTcl 提供了很多面向对象的概念，以降低大规模程序开发的复杂度。这些概念包括混元

类、嵌入类、集合、用于支持委托的转发器、数据存储槽以及提供类似面向方面编程的功能的筛选器。XOTcl 来自 Uwe Zdun 和 Gustaf Neumann。更多信息参见 XOTcl 的主页(http://www.xotcl.org)以及 Tcl 开发者维基的 XOTcl 页面(http://wiki.tcl.tk/xotcl)。

Snit 是“Snit' s Not Incr Tcl”的简写，提供了另一个纯由 Tcl 编写的面向对象扩展包。与大多数面向对象语言或语言扩展包不同，Snit 是基于委托而非继承的。Snit 对象将工作委托给子组件，而不是从基类中加以继承。它曾经是一个独立的扩展包，现在已经是附录 B.3 所述的 Tcllib 的一部分了。Snit 由 William Duquette 开发。更多信息参见 Tcl 开发者的维基 Snit 页面(http://wiki.tcl.tk/3963)以及 Snit 参考页面(http://tcllib.sourceforge.net/doc/snit.html)。

提示： Tcl 8.6 将会引入原生面向对象框架。尽管它足可以独立用作对面向对象脚本的支持，但设计它的目的是使其作为其他面向对象扩展包的内核，使得它们更容易实现，更有效率。

B.7 多线程 Tcl 脚本

第 47 章讨论了在多线程应用程序中嵌入 Tcl 的要求，讲述了 Tcl 支持平台无关的多线程程序开发的 C API。Tcl 没有任何用于脚本层次的多线程编程的内建命令。Thread 扩展(由 Brent Welch 开发，现由 Zoran Vasiljevic 维护)提供了一系列脚本层次的命令，用于在应用程序中创建和管理线程。

Thread 扩展支持通常也称为线程分离模式。在脚本层次创建的每一个线程都有它自己的 Tcl 解释器，由它保持自己的变量、过程、命名空间以及其他状态信息。Thread 扩展还实现了线程间通信的消息传递机制、共享变量、互斥体、条件变量以及线程池。

您只能在启用线程支持编译的 tclsh、wish 或其他应用程序中使用线程扩展，如第 46 章所述。而且，您的应用程序使用的其他的二进制扩展包也必须是以启用线程支持的方式编译的。

Thread 扩展的源码在 Tcl Sourceforge 的一个项目中维护：http://tcl.sourceforge.net。您可以在 Tcl 开发者维基 http://wiki.tcl.tk/thread 了解 Thread 扩展的更多信息。

B.8 XML 编程

近年来 XML 编程变得越来越重要，因为 XML 已经成为最常用的数据组织与交换格式。Tcl 有一些用于处理 XML 的扩展包。

TclXML 集成了很多与解析 XML 文档相关的扩展包。基础扩展包 TclXML，处理对 XML 文档的解析。TclDOM 添加了一个文档对象模型，可以在内存中把 XML 作为树结构操纵。TclXSLT 添加了 XSL 转换，TclTidy 可以清理 XML 和 HTML，特别适用于从网页解析 HTML，而很多数据可能不完全遵循 HTML 标准的情况。TclXML 由 Steve Ball 创建。更多信息参见 TclXML SourceForge 页面(http://tclxml.sourceforge.net)，以及 Tcl 开发者

维基的 TclXML 页面(http://wiki.tcl.tk/tclxml)。

tDOM 提供了解析 XML 文档的另一个选择，它包括了 XSLT 支持。tDOM 以其处理速度和低内存消耗著称。tDOM 项目由 Jochen Loewer 发行，现由 Rolf Ade 维护。更多信息参见 tDOM 主页(http://www.tdom.org)以及 Tcl 开发者维基的 tDOM 页面(http://wiki.tcl.tk/tDOM)。

TclSOAP 可以帮助您为 SOA 或者面向服务的架构创建 XML 消息，遵循简单对象访问协议，或者说 SOAP。使用 SOAP，应用程序可以发送或接收 XML 消息；它的主要优势在于您可以用任何语言编写程序，例如 Tcl，而不必担心另一端的程序实现。TclSOAP 由 Pat Thoyts 创建。更多信息参见 TclSOAP SourceForge 页面(http://tclsoap.sourceforge.net)以及 Tcl 开发者维基的 TclSOAP 页面(http://wiki.tcl.tk/tclsoap)。

B.9 数据库编程

尽管 Tcl 并没有对数据库访问的原生支持，但可以添加很多扩展包来从相关的数据库中取得信息。

Oratcl 允许应用程序访问 Oracle 数据库(http://oratcl.sourceforge.net)。Mysqltcl 提供了到 MySQL 数据库的相应接口(http://www.xdobry.de/mysqltcl)。PostgreSQL 数据库可由 pgtcl 扩展包支持访问(http://pgfoundry.org/projects/pgtcl)。Sybtcl 提供了对 Sybase 的支持(http://sybtcl.sourceforge.net)。sqlite2 扩展包提供了对 SQLite 的访问(http://www.sqlite.org)。TclODBC 允许 Tcl 应用程序调用 ODBC 数据库驱动，在 Windows 中最为常见(http://sourceforge.net/projects/tclodbc)。

提示： Tcl 8.6 将引入 TDBC，这个扩展包将会作为 Tcl 内核的一部分，提供访问 SQL 数据库的标准接口。更多信息参见 Tcl 开发者维基的 TDBC 页面(http://wiki.tcl.tk/tdbc)。

另一个流行的选择是 Metakit，它是为嵌入式数据库设计的一个精巧高效的库。Mk4tcl 扩展提供了访问 Metakit 的 Tcl API。更多信息参见 http://www.equi4.com/metakit。

B.10 整合 Tcl 和 Java

Tcl/Java 项目提供了两条路线来整合 Tcl 和 Java 代码。一个选项是 TclBlend，支持向已经存在的 Tcl 进程中加载 Java 解释器的扩展包。其结果是，您可以使用 Tcl 脚本命令来加载 Java 类，创建对象，调用方法，等等。另一个选项是 Jacl，纯由 Java 编写的完整的 Tcl 解释器实现。Jacl 的用途是把脚本功能整合到已经存在的 Java 应用程序中。更多信息参见 Tcl/Java 主页(http://tcljava.sourceforge.net)以及 Tcl 开发者维基的 Tcl/Java 页面(http://wiki.tcl.tk/tcljava)。

B.11 SWIG

如果已经有了一个 C/C++代码的库，您可能希望能在 Tcl 脚本中调用它的函数。虽然您可以编写自己的包装代码，把这些函数包装为 Tcl 命令，但有一个名为 SWIG 的工具可以自动完成这些包装。SWIG 不仅限于在 Tcl 中使用，它还能为其他很多动态语言进行包装。更多信息参见 SWIG 主页(http://www.swig.org)。

B.12 Expect

Expect 是最古老的 Tcl 应用程序之一，也是最流行的之一。它是一个与交互式程序“谈话”的程序。Expect 脚本描述了期望一个程序给出的输出是什么样的，正确的响应应该是什么样的。它可以用于自动地控制如 ftp、telnet、ssh、fsck 这样的程序，以及其他因为需要交互式输入而不能由外壳脚本自动控制的程序。Expect 还允许用户接过对需要的程序的控制，直接与其进行交互。例如，下面的 Expect 脚本用 ssh 程序登录远程机器，将工作目录设为原计算机，然后将控制权交给用户。

```
package require Expect
spawn ssh [lindex $argv 0]
expect -nocase "password:"
send [lindex $argv 1]
expect -re "(%|#) "
send "cd [pwd]\r"
expect -re "(%|#) "
interact
```

spawn、expect、send 以及 interact 命令都由 Expect 实现，而 lindex 和 pwd 是内建 Tcl 命令。spawn 命令启动 ssh，使用命令行参数作为远端计算机的名称(lindex 经由 argv 变量取得命令行的第一个参数)。expect 命令等待 ssh 输出密码提示符，以区分大小写的方式进行匹配。脚本认为命令行的第二个参数就是密码，将它传给 ssh 程序。(这种方法并不安全，这里只是为了演示。在 Expect 中还有其他方法来处理密码提示符，不过那已经超出了本书的范围。)脚本等待外壳提示符(%或#，后面接一个空格)，然后 send 输出一个命令用来改变工作目录，就像是用户交互式地输入了命令一样。下面的 expect 命令等待 cd 命令已经被执行的提示。最后，interact 使得 Expect 退出，现在调用 Expect 的用户可以直接与 ssh 交互了。

Expect 可以用于很多目的，例如作为一个可由脚本编程的前端，用于调试、发信以及执行其他该脚本语言没有的命令行驱动的命令。由 Expect 驱动的程序本身不需要任何修改。Expect 还用于对交互式程序的回归测试。Expect 可以与 Tk 和其他 Tcl 扩展包联合应用。例如，联合应用 Tk 和 Expect，可以为一个现有交互式应用程序编写一个图形化的前端，而无需修改应用程序。

Expect 由美国国家标准与技术研究院的 Don Libes 创建。更多信息参见 Expect 的主页(http://expect.nist.gov)以及 Tcl 开发者维基的 Expect 页面(http://wiki.tcl.tk/expect)。

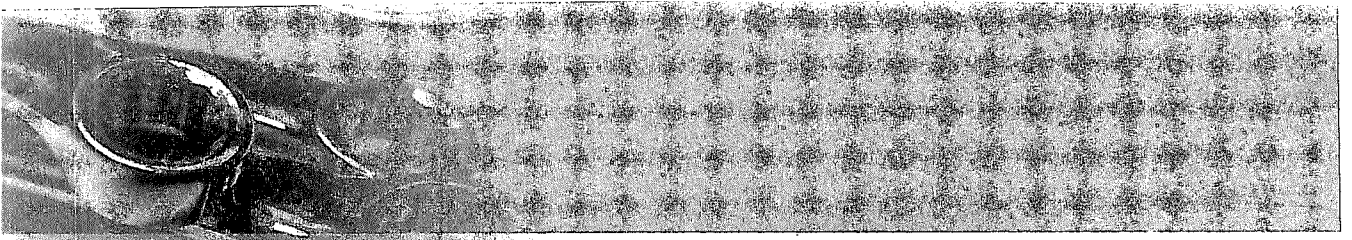
B.13 扩展 Tcl

扩展 Tcl(TclX)是一个库包，包括内建 Tcl 命令加上其他很多命令以及针对系统程序任务的过程。下面是 TclX 最重要的一些特点：

- 更多 POSIX 系统调用和函数，如 fork 和 kill。
- 文件扫描工具，与 awk 程序很相似。
- 带关键字的列表，提供了与 C 结构体相似的功能。
- 调试、配置以及程序开发工具。

TclX 的最优秀的特点中有很多已经不再是它独有的一部分：它们是如此地应用广泛，已经被整合进 Tcl 内核。TclX 曾比 Tcl 领先的功能包括文件输入输出、TCP/IP 套接字、数组变量、实运算和抽象函数、自动加载、incr 和 upvar 命令。

扩展 Tcl 由 Karl Lehenbauer 和 Mark Diekhans 创建。更多信息参见 TclX SourceForge 页面(http://tclx.sourceforge.net)和 Tcl 开发者维基的 TclX 页面(http://wiki.tcl.tk/tclx)。



附录 C　Tcl 资源

有关 Tcl/Tk 的更多信息，推荐参考如下资源。

C.1　在线资源

关于 Tcl/Tk 有很多可用的在线资源。

- Tcl 开发者 XChange，http://ww.tcl.tk
  跟上 Tcl 世界发展步伐的很好的切入点。这里会通知新的 Tcl 发布、将举行的会议以及其他重大的 Tcl 事件。这里还有在线版的 Tcl/Tk 参考文档，针对当前版本的和历史版本的都有，以及 Tcl 下载链接和 Tcl 演说信息。
- Tcl 开发者维基，http://wiki.tcl.tk
  关于 Tcl 的技巧、提示、示例代码的集合体。这个站点有很多宝贵的 Tcl 信息，当您要搜索 Tcl 的有关主题时这是第一个应该查看的地方。
- ActiveState 程序开发者网络(ASPN)Tcl 资源，http://aspn.activestate.com/ASPN/Tcl
  ActiveState 的 ASPN 提供了 ActiveState 的技术支持信息，包括 Tcl。这个站点有很多与 Tcl 相关的邮件清单、一份 Tcl“食谱”和其他一些资源。从这个站点您还可以免费下载 ActiveTcl、ActiveState 构建的二进制的 Tcl 可执行文件以及一些流行的扩展包。
- TkDocs，http://www.tkdocs.com
  Mark Roseman 的站点提供了 Tk 入门指导和文档。Mark 有一些关于跨平台 Tk 开发的很棒的技巧，他的入门介绍不仅包含了在 Tcl 中使用 Tk，还包含了其他的动态语言，例如 Perl 和 Python。
- Tcl SourceForge 项目，http://tcl.sourceforge.net
  Tcl/Tk 和内核扩展包的源码档案库。
- comp.lang.tcl Usenet 全球新闻组
  全球新闻组 Usenet 可用于有关 Tcl/Tk 以及相关的扩展包和应用程序的信息交换。这个新闻组回答来自用户的问题，交换关于错误及其修正的信息，讨论 Tcl 和 Tk 未来可能的功能。新发布的 Tcl/Tk 以及其他的扩展包和应用程序的发布也会在新闻组中通知。您可以用专用的新闻客户端应用程序，或是 Google Groups 这样的网页(http://groups.google.com)访问 Usenet 新闻组。

- Tcl 聊天室
  一个讨论 Tcl/Tk 的实时聊天服务，目前基于 XMPP 协议。很多经验丰富的 Tcl 程序开发者常常会在这里讨论，回答问题，解决问题，以及争议 Tcl/Tk 的发展。这个聊天室可以通过标准 XMPP 客户端访问，或通过一个专用的 TkChat 客户端(http://tkchat.tcl.tk)访问，也可以通过网页和 IRC 访问。如何访问 Tcl 聊天室的最新信息参见 Tcl 开发者维基页面(http://wiki.tcl.tk/1178)。

C.2 书籍

可以查阅以下相关书籍，获得更多有关 Tcl/Tk 的信息。

- Butenhof D R. *Programming with POSIX Threads*. Reading, MA: Addison-Wesley, 1997.
- FLynt C. *Tcl/Tk: A Developer's Guide*, Second Edition. San Francisco: Morgan Kaufmann, 2003.
- Friedl J. *Mastering Regular Expressions*, Third Edition. Sebastopol, CA: O'Reilly Media, Inc., 2006.
- Libes D. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*. Sebastopol, CA: O'Reilly Media, Inc., 1994.
- Welch B, Jones K. *Practical Programming in Tcl and Tk*, Fourth Edition. Upper Saddle River, NJ: Prentice Hall, 2003.

附录 D　Tcl 源码发布许可

本软件版权属于加州大学、Sun Microsystems 公司、Scriptics 公司、ActiveState 公司等团体。除非对单个文件明确声明，以下条文适用于与本软件相关的所有文件。

允许出于任何目的对该软件及其文档进行使用、复制、修改、发布以及授权，在所有的副本中都应该提供这个版权信息，本通知应该逐字包含在任何发布中。使用授权不需要任何书面协议、证书或版权费。对此软件进行修改后的版权由修改者决定，不要求必须继续遵循这里的开源条款，新的条款应清楚地出现在它们所适用的文件的第一页。

任何情况下，作者和发布者不对因为使用本软件或其文档或任何派生程序，而直接地或间接地，意外地或必然地造成的损害负责，即使作者已经被提醒该损害有出现的可能。

作者和发布者特别声明放弃一切权利，包括但不限于为商品应用的权利，用于特定目的适用性权利，且不存在版本侵犯问题。本软件以“它现在的状态”提供，作者和发布者没有义务提供维护、支持、更新、强化及修改。

政府应用：如果您代表美国政府获取本软件，政府只能获取本软件及相关文档的“有限权利”，遵照美国联邦采购法规(FAR)第 52.227.19(c)(2)条款。如果您代表国防部获取本软件，本软件应归类为“商用计算机软件”，政府只能获取“有限权利”，遵照 DFAR 第 252.227-7013(c)(1)条款。尽管有以上规定，作者允许美国政府及其代表者依照本许可使用和发布本软件。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引