

Tcl 编程初步

编辑版本 : 1.5
日 期 : 2004-3-2
著 者 : 陈 涛

上海贝尔阿尔卡特有限公司

总目录

总目录.....	I
例索引.....	IV
表索引.....	VI
图索引.....	VII
第 1 章. TCL 基本知识.....	1
1.1 什么是 Tcl	1
1.2 Tcl 自学工具	1
1.3 Tcl 软件包	2
1.4 Tcl 命令格式	2
1.5 Tcl 脚本文件和 SOURCE 命令	3
1.6 可执行脚本文件(EXECUTABLE FILE)	3
1.7 获得帮助	4
1.7.1 Windows 系统.....	4
1.7.2 Unix 系统.....	4
第 2 章. 输出、赋值与替换	5
2.1 PUTS	5
2.2 SET & UNSET	5
2.3 替换	6
2.3.1 \$.....	6
2.3.2 []	6
2.3.3 " 和 {}	6
2.3.4 \.....	7
第 3 章. 数学表达式与 EXPR 命令	8
3.1 数学和逻辑运算符	8
3.2 数学函数	8
3.3 数学运算举例	9
3.4 INCR 命令	9
第 4 章. 字符串	10
4.1 基本命令集	10
4.2 APPEND 命令	10
4.3 FORMAT 命令	10
4.3.1 format 命令说明	10
4.3.2 format 举例.....	11
4.4 SCAN 命令	12
4.5 BINARY 命令	13
4.6 SUBST 命令	14
4.7 STRING 命令	15
4.7.1 string 命令列表.....	15
4.7.2 字符串比较.....	16
4.7.3 string match 字符串匹配.....	16
4.7.4 字符串替换.....	18
4.7.5 字符类别(class) 测试.....	18
4.7.6 字符串映射.....	19

第 5 章. TCL 列表操作	20
5.1 列表命令集	20
5.2 LIST 命令	20
5.3 CONCAT 命令	21
5.4 LAPPEND 命令	21
5.5 LLENGTH 命令	22
5.6 LINDEX 命令	22
5.7 LRANGE 命令	22
5.8 LINSERT 和 LREPLACE 命令	22
5.9 LSEARCH 命令	23
5.10 LSORT 命令	24
5.11 JOIN 与 SPLIT 命令	24
5.12 FOREACH 控制结构	27
第 6 章. 数组	28
6.1 数组的定义与格式	28
6.2 数组变量	29
6.3 多维数组	29
6.4 数组操作命令	30
6.4.1 array get 命令	30
6.4.2 array names 命令	30
6.4.3 遍历数组	31
6.4.4 用数组定义结构	31
第 7 章. 控制结构命令	32
7.1 IF/ELSE 命令	32
7.2 FOR 命令	33
7.3 WHILE 命令	33
7.4 BREAK 与 CONTINUE 命令	34
7.5 SWITCH 命令	34
7.6 CATCH 命令	35
7.7 ERROR 命令	36
7.8 RETURN 命令	37
7.9 EXIT 命令	37
第 8 章. 过程与作用域	38
8.1 PROC—过程定义命令	38
8.2 作用域	40
8.2.1 过程的作用域	40
8.2.2 变量的作用域	40
8.3 UPVAR 命令	42
8.4 RENAME 命令	44
8.5 特殊变量	45
8.5.1 命令行参数	45
8.5.2 env—环境变量数组	45
8.6 EVAL 命令	46
8.7 Uplevel 命令	48
第 9 章. 正则表达式 (REGULAR EXPRESSIONS)	49
9.1 REGEXP 命令	49
9.2 REGSUB 命令	51
9.3 正则表达式的语法	52

9.3.1	分支(branch) 和原子(atom).....	52
9.3.2	基本语法.....	52
9.3.2.1	匹配字符.....	52
9.3.2.2	限定匹配.....	53
9.3.2.3	方括号表达式与字符集.....	53
9.3.2.4	匹配分支.....	54
9.3.2.5	量词 (Qulifier)	54
9.3.2.6	子模式与匹配报告捕获.....	55
9.3.2.7	反斜杠引用.....	56
9.3.2.8	匹配优先级.....	56
9.3.3	高级正则表达式 (AREs)	57
9.3.3.1	反斜杠换码(escape)序列.....	57
9.3.3.2	归整元素(collating element).....	57
9.3.3.3	等价类(equivalence class).....	58
9.3.3.4	字符类(character class).....	58
9.3.3.5	非贪婪量词.....	59
9.3.3.6	约束量词.....	59
9.3.3.7	回退引用.....	59
9.3.3.8	前瞻(lookahead).....	60
9.3.3.9	换行符敏感的匹配.....	60
9.3.3.10	嵌入式选项.....	60
9.3.3.11	扩展语法.....	60
9.3.4	语法小结.....	60
9.3.5	其它支持正则表达式的命令.....	60
第 10 章.	名字空间.....	62
10.1	创建名字空间.....	62
10.2	用::限定符来使用变量和过程.....	62
10.3	名字空间的变量.....	63
10.4	过程的进口与出口.....	64
10.5	自省(INTROSPECTION).....	65
10.6	名字空间命令集.....	65
第 11 章.	跟踪与调试.....	67
11.1	CLOCK 命令.....	67
11.1.1	clock clicks 命令.....	67
11.1.2	clock seconds 命令.....	67
11.1.3	clock format 命令.....	68
11.1.4	clock scan 命令.....	70
11.2	INFO 命令.....	71
11.2.1	info level.....	72
11.2.2	info exists.....	72
11.3	TRACE 命令.....	73
11.3.1	trace variable.....	73
11.3.2	trace vdelete.....	75
11.3.3	trace vinfo.....	75
第 12 章.	脚本库与软件包.....	76
12.1	声明和使用软件包.....	76
12.1.1	软件包定位.....	76
12.1.2	声明软件包命令.....	76
12.1.3	加载软件包命令.....	76
12.1.4	自动加载与软件包索引.....	77
12.1.5	用链接库提供软件包.....	79

12.2	PACKAGE 命令集	80
12.3	小结	80
第 13 章. 文件操作与程序调用.....		81
13.1	文件操作	81
13.1.1	文件 I/O.....	81
13.1.2	文件系统信息命令.....	82
13.1.2.1	glob 命令	83
13.1.2.2	file 命令集.....	83
13.2	程序调用	85
13.2.1	用 open 命令打开一个进程管道	85
13.2.2	用 exec 命令调用程序.....	87
13.2.3	pid 命令.....	88
第 14 章. 套接字与事件驱动编程简介.....		89
14.1	套接字编程	89
14.1.1	socket 命令.....	89
14.1.1.1	Client 端 socket 命令.....	89
14.1.1.2	Server 端 socket 命令选项.....	90
14.1.2	用 fconfigure 配置套接字.....	90
14.1.3	C/S 编程举例.....	90
14.2	事件驱动编程	93
14.2.1	after 命令.....	93
14.2.2	fileevent 命令.....	96
14.2.3	vwait 命令.....	96
14.2.4	fconfigure 命令	97
14.2.4.1	fconfigure 语法	97
14.2.4.2	非阻塞 I/O	98
14.2.4.3	缓冲	98
参考文献		100

例索引

例 1-1	输出一段字符串的例子	2
例 1-2	source 命令的简单例子	3
例 2-1	输出一个词的例子	5
例 2-2	参数定义、赋值与参数值引用	5
例 2-3	嵌套 \$ 用做替换操作	6
例 2-4	命令替换 [] 的例子	6
例 2-5	{ } 替换的例子	6
例 2-6	\ 的例子	7
例 2-7	不规范的续行	7
例 3-1	数学运算举例	9
例 4-1	append 命令的简单例子	10
例 4-2	位置说明符的例子	11
例 4-3	format 命令的简单例子	11
例 4-4	scan 命令的简单例子	12
例 4-5	体验 binary format 和 binary scan 的作用	13
例 4-6	binary 命令的简单应用	14
例 4-7	subst 命令的简单例子	14
例 4-8	string compare 和 string equal 进行字符串比较的例子	16

例 4-9 字符串替换简单例子	18
例 5-1 使用 list 命令创建列表	20
例 5-2 concat 命令的例子	21
例 5-3 lappend 命令简单例子	21
例 5-4 llength 命令的一个简单例子	22
例 5-5 lindex 命令的一个简单例子	22
例 5-6 linsert 和 lreplace 命令的例子	22
例 5-7 lsearch 命令的简单例子	23
例 5-8 lsearch 与 lreplace 结合删除列表元素	23
例 5-9 lsort 命令简单的例子	24
例 5-10 split 命令例子	24
例 5-11 空元素与独立字符元素的 split 例子	24
例 5-12 用 ldel 删除指定元素	25
例 5-13 根据步长调整 MAC 地址值	25
例 5-14 foreach 的简单例子: 依次打印列表元素	27
例 5-15 具有多个值列表的 foreach 命令处理过程	27
例 6-1 认识数组	28
例 6-2 混淆普通变量和数组时的错误例子	29
例 6-3 通过替换间接使用数组变量	29
例 6-4 数组和列表互换	30
例 6-5 array names 的简单例子	31
例 6-6 遍历数组的一个方法举例	31
例 6-7 用数组来定义结构	31
例 7-1 if/else 控制命令的简单例子	32
例 7-2 一个 for 循环	33
例 7-3 while 循环的例子	34
例 7-4 switch 命令的简单例子	34
例 7-5 switch 语句中不当注释引起的错误	35
例 7-6 用 catch 捕获命令错误信息	35
例 7-7 error 命令的例子	36
例 7-8 用 return 命令从过程中返回	37
例 8-1 带有默认参数的过程定义	38
例 8-2 不定输入参数过程的例子	39
例 8-3 参数名+参数值成对输入的过程定义	39
例 8-4 过程的定义	40
例 8-5 变量的作用域	40
例 8-6 全局变量与局部变量的关系	41
例 8-7 用 "::" 来声明全局变量	41
例 8-8 upvar 命令的例子	42
例 8-9 通过 upvar 命令来传递数组	44
例 8-10 用 rename 命令来取消一个命令	44
例 8-11 打印命令行参数信息	45
例 8-12 用 eval 创建新命令的简单例子	46
例 8-13 动态定义过程	47
例 8-14 uplevel 的简单例子	48
例 9-1 regexp 的简单例子	50
例 9-2 用 regsub 进行字符串替换的简单例子	51
例 9-3 匹配挂靠的简单例子	53
例 9-4 字符集匹配的例子	53
例 9-5 使用量词*和?不当引起的错误	55

例 9-6 子模式捕获.....	55
例 9-7 屏蔽子模式报告	55
例 9-8 子模式综合运用的例子: 搜索脚本文件中定义的过程	55
例 9-9 归整元素匹配字符串	58
例 9-10 字符类的简单例子	58
例 10-1 定义名字空间 Counter:.....	62
例 10-2 动态定义名字空间	62
例 10-3 限定名比较的简单例子	63
例 10-4 名字空间自动处理引用变量的归属	64
例 10-5 过程的进口与出口的简单例子	64
例 10-6 namespace origin 命令例子	65
例 11-1 计算系统时钟滴答数.....	67
例 11-2 clock format 在 windows 上的简单例子	69
例 11-3 Tcl 识别的系统编码方式	69
例 11-4 消除 clock format 输出中的乱码	70
例 11-5 用 info 测试变量是否存在	71
例 11-6 用 info level 命令控制过程的循环嵌套	72
例 11-7 用 trace variable 跟踪变量的简单例子	73
例 11-8 trace vinfo 的例子	75
例 12-1 auto_path 的内容与操作	76
例 12-2 pkg_mkIndex 命令	77
例 12-3 简单的软件包加载的例子	78
例 13-1 用 open 命令打开文件, 并输入数据	82
例 13-2 lstat 和 stat 命令举例	84
例 13-3 用 open 命令打开只读进程管道	86
例 13-4 用 open 命令打开进程管道又一例	86
例 13-5 用 exec 处理管道与 I/O 重定向	87
例 13-6 用 pid 命令检查进程 ID	88
例 14-1 基于 socket 实现的 C/S(Client/Server)简单模型	90
例 14-2 after 命令的简单应用	94
例 14-3 vwait 命令的简单例子	97

表索引

表 3-1 数学与逻辑运算符	8
表 3-2 数学函数	8
表 4-1 比较有用的字符串操作命令	10
表 4-2 格式转换符	11
表 4-3 格式标志符	11
表 4-4 二进制转换类型	13
表 4-5 string 命令	15
表 4-6 使用 string match 来匹配字符的结构	16
表 4-7 字符类名目	18
表 5-1 列表相关命令	20
表 6-1 数组操作命令表	30
表 9-1 regexp 选项	50
表 9-2 AREs 的换码表	57
表 10-1 namespace 命令	65
表 11-1 clock 命令	67
表 11-2 clock format 的域描述符	68

表 11-3 clock format 特定于 UNIX 系统的域描述符	68
表 11-4 info 命令集	71
表 12-1 pkg_mkIndex 命令开关选项	77
表 12-2 package 命令	80
表 13-1 文件操作命令	81
表 13-2 open 命令的 access 变量说明	81
表 13-3 file 命令集	83
表 13-4 file stat 命令数组元素	85
表 13-5 重定向指示标识和说明	88
表 14-1 after 命令	93
表 14-2 fconfigure 控制的 I/O 通道属性	98

图索引

图 1 TclTour 界面	1
图 2 启动 Tcl 交互界面	2
图 3 交互界面	2
图 4 启动 Windows 上的 Tcl Help	4
图 5 使用在另外一个 Tcl 文件中的过程	39
图 6 regexp 匹配过程示意图	50
图 7 文件访问控制权限说明	82
图 8 进程管道	85
图 9 C/S 界面	93

第 1 章. Tcl 基本知识

1.1 什么是 Tcl

Tcl 全称是 Tool command Language。它是一个基于字符串的命令语言，基础结构和语法非常简单，易于学习和掌握。

Tcl 语言是一个解释性语言，所谓解释性是指不象其他高级语言需要通过编译和联结，它象其他 shell 语言一样，直接对每条语句顺次解释执行。

Tcl 数据类型简单。对 Tcl 来说，它要处理的数据只有一种——字符串。Tcl 将变量值以字符串的形式进行存储，不关心它的实际使用类型。

内嵌的 Tk (toolkit) 图形工具可以提供简单而又丰富的图形功能，让用户可以轻松的创建简单的图形界面。

Tcl 的执行是交互式的，Tcl 提供了交互式命令界面，界面有两种：tclsh 和 wish。tclsh 只支持 Tcl 命令，wish 支持 Tcl 和 Tk 命令。通过交互界面，我们就可以象执行 UNIX shell 命令一样，逐条命令执行，并即时得到执行结果。

Tcl/Tk 可以提供跨平台的支持。Tcl 语言可以运行于绝大多数当今流行的 UNIX、WINDOWS 和 Macintosh 等系统上，而且命令通用，只是启动的细节有些不同。

Tcl/Tk 与 C/C++ 的良好兼容性。Tcl/Tk 脚本可以很好的集成到 C/C++ 程序中。

1.2 Tcl 自学工具

Clif Flynt 开发了一套简单的自学工具 TclTour，共有 43 个短小课程，每个课程一般用 10 分钟就可以学习完成，对初学者来说是个较好的入门工具。不过在使用 TclTour 之前，需要先安装 Tcl/Tk 软件。

图 1 是 TclTour 的界面。界面主要由四部分组成：菜单、介绍窗、脚本窗、输出窗。由 "file" 菜单可以选择课程、设置字体、颜色。由 "Terseness" 可以选择介绍内容的级别，如初、中、高级等等，随着级别的升高，介绍就越简略。介绍窗口中对命令的功能和语法进行介绍。脚本窗给出了本课相关的实例脚本，选择 "Run Example" 就可以执行脚本，并在输出窗中给出执行结果。

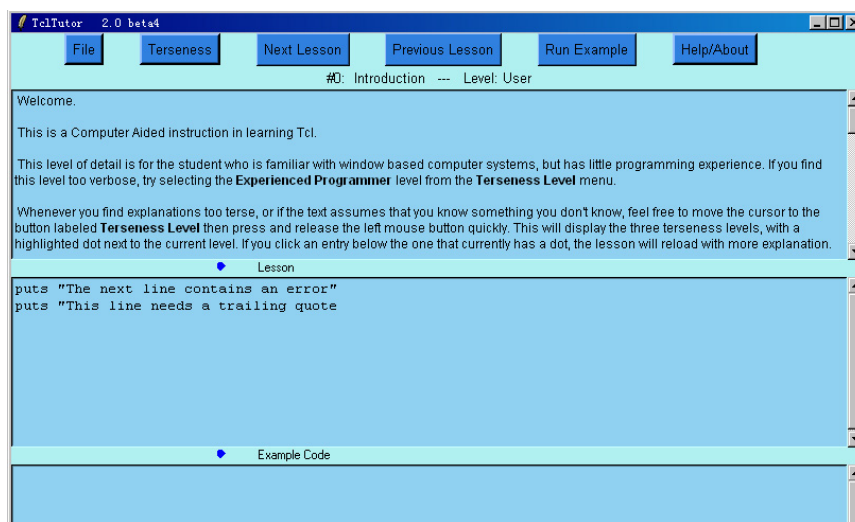


图 1 TclTour 界面

TclTour 可以从 \\dndnet\WBA-backup\SIT\training file\self study - TCL & SNMP\目录下获取。解压后运行 TclTour。

1.3 Tcl 软件包

最新的 Tcl 软件可以从 www.scriptics.com 下载，一些比较新的 Tcl 功能命令需要较高版本的 Tcl 软件。Tcl 软件的版本比较多，大家可根据需要选用。Tcl 软件安装好后，可以通过“开始”快捷菜单启动 Tcl 交互界面。本文使用的是 Active Tcl8.3.4 和 Tclpro1.41。Tclpro1.41 可以从\\dndnet\WBA-backup\SIT\training file\self study - TCL & SNMP\目录下获取。

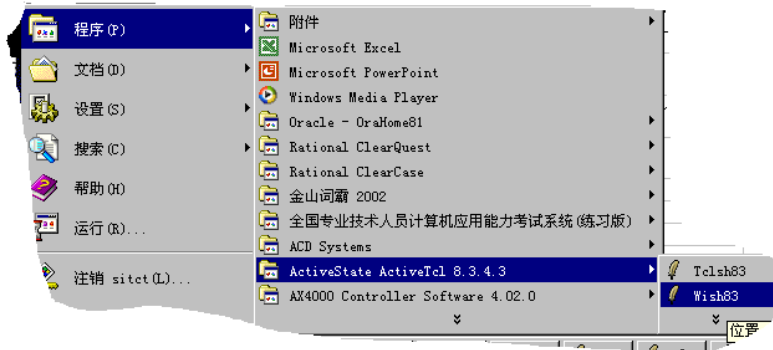


图 2 启动 Tcl 交互界面

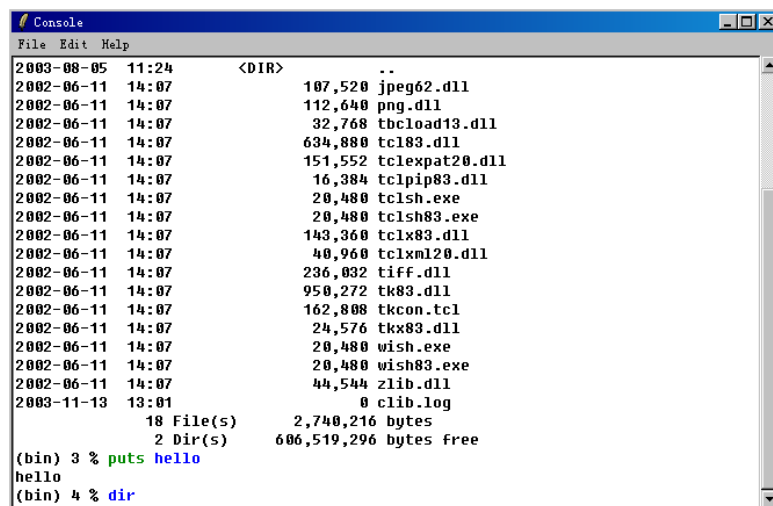


图 3 交互界面

从www.scriptics.com可获得相关的学习资料和实用工具（如 Tcl 脚本编辑工具和集成开发工具等）。

1.4 Tcl 命令格式

一条 Tcl 基本语法为：

command arg1 arg2 (命令 参数 1 参数 2...)

其中 command 为 Tcl 内建命令或者 Tcl 过程。各条命令可以单独一行，也可以在同一行，在同一行时，需要用“;”来分隔。即：

command1 arg1 ... ; command2 arg2; # (a comment 注释)

注释是以“#”标记开始。

如以下的输出命令会在输出终端输出一段字符串：

例 1-1 输出一段字符串的例子

```
%puts "The next line contains an error"
```

=> The next line contains an error

注意：字符串中如果有空格的话，一定要用""或者{}括起来。因为 Tcl 语句中各命令参数是以空格来分隔的，如果不用""或{}括起来，字符串中的单词会被误认为多余的参数而出错。

1.5 Tcl 脚本文件和 source 命令

除了通过交互式执行 Tcl 命令外，还可以将 Tcl 命令保存到一个后缀为 .tcl 的文件内，然后用 Tcl 的命令 source 来执行：

source fileName

source 命令从文件中读取 Tcl 命令并计算。

例 1-2 source 命令的简单例子

```

;#用文本编辑器创建一个文件，名为 e1.tcl 并保存到 C:\盘根目录
;#输入如下一条命令后保存、关闭文件：
;#puts "hello"

;#执行 tclsh:
% dir e1.tcl
=>2003-11-19 14:31      12 e1.tcl
      1 File(s)      12 bytes
%source e1.tcl
=>hello

```

1.6 可执行脚本文件(Executable File)

编写完一个 Tcl 脚本文件后，可用 source 命令来运行该脚本。但这需要先启动 tclsh。如果能让 Tcl 脚本在当前 shell，如 UNIX 的 ksh 或 csh 下自动启动 tclsh 来运行脚本就比较方便。可执行脚本文件就能满足这个要求。

要编写可执行脚本文件，必须有 Tcl 启动脚本(Tcl start-up scripts)。启动脚本完成诸如调用必须软件包、初始化必要的变量（包括环境变量），然后调用 Tcl 软件包内的程序。每个平台都有自己的创建启动脚本。

1. 在 UNIX 中，最普遍的编写可执行应用程序的方法就是用 “#!” 机制。用它生成的启动脚本如下：

```
#!/usr/bin/wish -f
```

本启动脚本告诉 UNIX shell，用 wish 来运行剩余的脚本。选项 -f 是用来支持 3.6 以下的 Tk 版本脚本的。这种启动脚本需要提供 wish 或 tclsh 的绝对路径名，不利于移植到不同的机器上。因为不同的机器，wish 或 tclsh 的路径有可能不同。在用 PATH 和 LD_LIBRARY_PATH 指定 wish/tclsh 及其库的路径的情况下，可以编写下面格式的启动脚本：

```

#!/bin/sh
# the next line restarts using wish \
exec wish8.0 "$0" "$@"

```

上面的脚本最后会将 wish 应用程序安装到用户目录下。第二行中的反斜杠对 UNIX sh 来说被认为是注释的一部分，但对 Tcl 来讲，反斜杠表明下一行则是注释的继续。

实验一下：在 UNIX 上建立一个简单的 .tcl 脚本，并将启动脚本放在开始。保存文件、设置好 tclsh 的路径。然后在 UNIX shell 内运行：

```
$. ./xxx.tcl
```

2. 对于 Windows 下的 Tcl 脚本，如过已经正确安装了 tclsh 或 wish，只要你将脚本文件保存为 .tcl 文件，不需添加启动脚本，只用鼠标双击文件就可以运行。

1.7 获得帮助

Tcl 软件包提供了详细的帮助信息，可以让使用者轻松获得在线帮助。

1.7.1 Windows 系统

在 Windows 系统上获得帮助信息比较简单，在安装 TCL/TK 软件包的时候，帮助手册也一同安装到系统中。运行 StartMenu->Programm 上的 Tcl Help 就可以打开帮助手册，它提供了标准的 Windows 帮助界面。



图 4 启动 Windows 上的 Tcl Help

建议学习每个命令的时候，能抽出一段时间阅读相关帮助信息，对学习 Tcl 语言将会有很大帮助。

1.7.2 Unix 系统

在 Unix 系统上，Tcl 通过标准 man 手册提供帮助。在使用手册前，需要设置环境变量 MANPATH 指向 Tcl man 手册所在路径。

比如，在一个 UNIX 系统上 TclPro 的 man 目录为 /opt/aries/tclpro/doc/man，在此目录下包含了两个子目录：man1、man3 和 mann。先检查以下 MANPATH 环境变量中是否包含了此路径：

```
$echo $MANPATH
=>:::/cm4/tools/WRS/visionXD/man:/usr/atria/doc/man:/ap/Rational/releases/quantify.sol.2002a.06.00/man:/ap/Rational/releases/purify.sol.2002a.06.00/man
```

MANPATH 中还没有包含 Tcl manual 的路径，这时使用 Tcl 的帮助手册，则会提示相应的参考不存在：

```
$man string
=>No manual entry for string.
```

将 Tcl 的 manual 路径加到 MANPATH 中，然后再调用 man 命令：

```
$export MANPATH=${MANPATH}:/opt/aries/tclpro/doc/man
```

```
$man string
=>Reformatting page. Please Wait... done
Tcl Built-In Commands          string(n)
```

第 2 章. 输出、赋值与替换

2.1 puts

[语法]: `puts ?-nonewline? ?channelId? string`

Tcl 的输出命令是“puts”，将字符串输出到标准输出 **channelId**。语法中两个问号之间的参数为可选参数。

例 2-1 输出一个词的例子

```
17%puts hello
=> hello
18%puts -nonewline "hello hello"
=>hello hello19%
```

但如果输出一段有空格的文本，则要用双引号或者花括号括起来（见例 1-1）。

-nonewline 选项告诉 puts 不输出回车换行。

注意：双引号和花括号的作用是将多个词组织成一个变元，但他们是有差别的！这种差别就是在处理“替换操作”时，前者允许替换发生，而后者则可能会阻止替换。关于两者用法与差别以后会陆续讲到。在这里两者作用相同。

2.2 set & unset

[语法] `set varName [value]`
`unset varName`

set: 变量定义和赋值命令。不必指定变量值的类型，因为变量值的类型仅一种——字符串。为变量赋值时，会为变量开辟一段内存空间来存储变量值。

unset 命令与 **set** 命令作用相反，它取消变量定义，并释放变量所占的内存空间。

通过前置“\$”符，可以引用变量的值（替换）。

set 命令也可以只跟变量而无变量值。若变量已经定义，则返回变量值，效果和 **puts** 类似。如果该变量不存在，则返回错误信息。

例 2-2 参数定义、赋值与参数值引用

```
%set a Hello      ;#定义变量 a 并赋值
=>Hello
%puts $a          ;#输出变量值
=>Hello
%set a "Test Tcl" ;#重新赋值
=>Test Tcl
%set a            ;#输出变量值，此时不加"$"
=> Test Tcl
%puts $a          ;#输出变量值，此时要加"$"
=>Test Tcl
%puts a           ;#输出字符"a"
=> a
%set b $a         ;#将 a 的值赋给 b
=>Test Tcl
%puts $b
=>Test Tcl
% unset a         ;#释放变量
%puts $a          ;#试图输出未定义变量的值
```

```
=> can't read "a": no such variable
```

puts \$a 由两步完成：首先用 a 的值替换 \$a，替换后命令变为 “puts {Test Tcl}”，然后输出字符串。

[注意：定义数组的命令有特定格式，和普通变量定义有所不同。详见数组一章。]

2.3 替换

在 2.2 的例子中，已经使用了替换操作。现进一步讨论不同情况下的替换操作。

2.3.1 \$

“\$”符实现引用替换，用以引用参数值。如例 2-2。

Tcl 对替换只进行一遍解释，对嵌套的“\$”不予理睬。如下例：

例 2-3 嵌套\$用做替换操作

```
%set foo oo
=>oo
%set dollar foo
=>foo
%set x $$dollar
=>$foo      ;#只解释一次，将“$dollar”用 dollar 的值（foo）代替，
              ;#命令等效为 set x {$foo}，大括阻止替换。

% set x {$foo}
=>$foo
%set y $x
=>$foo      ;#一轮替换
```

2.3.2 []

方括号“[]”完成命令替换。用“[]”将一条命令括起来，命令执行完成后，返回结果。

例 2-4 命令替换[]的例子

```
% set b [set a 5]      ;#set a 5 命令输出的结果赋给 b
=>5
%puts $b
=>5
%set c [expr 5 * 10]    ;#将乘式结果赋给 c
=>50
```

2.3.3 "" 和 {}

双引号和花括号将多个单词组织成一个参数，也是一种替换操作。“”和{}内的替换如何进行呢？一般的原则是在“”内的替换正常进行，而在{}内的替换有可能会被阻止。

对花括号内的替换操作可以概括为：如果花括号是用做替换操作，则它会阻止内部的嵌套替换，如例 2-3；如果花括号用做界限符，如过程定义时用做界限过程体时，不阻止替换操作，其他还有 if 条件语句、循环语句、switch 语句和过程声明、数学表达式等。{}的作用比较特殊，需要根据不同的情况区别处理。

例 2-5 {}替换的例子

```
%set s hello
=>hello
%puts stdout "The length of \"$s\" is [string length $s]" ;# "string length" 命令是计算
```

;#字符串长度,用反斜杠"\"来引用特殊字符。

```
=>The length of hello is 5
%puts stdout {The length of $s is [string length $s]}
=>The length of $s is [string length $s]    ;#替换操作被阻止
%set x 10
=>10
(bin) 35 % set y 20
=>20
(bin) 36 % set z [expr {$x + $y}]    ;#expr 表示进行数学运算, 要用[]括起来
=>30    ;#用{}来组织算术表达式, 不阻止$替换操作
% if { $x == 10 } { puts "x=$x" }
=> x=10    ;#在条件语句中, {}用来界定条件体和执行体, 不阻止内部替换
```

2.3.4 \

反斜杠“\”的作用是引用特殊字符、续行。

引用特殊字符的操作是告诉 Tcl 解释器需要使用一些特殊的字符, 如\$符, 或完成特定功能, 如打印换行、震铃等。

续行的作用是如果一条语句太长不容易阅读, 可以用“\”来续行, 这样一条语句可以多行完成。

例 2-6 \的例子

```
% set x 100
=>100
%set y "$x"
=>100
%set y "\"$x"
=>$x    ;#引用$符
%puts "he\nllo"    ;#回车换行
=>he
    llo
%puts "\a"    ;#震铃(需要在 tclsh 下运行)
% set b [puts \
    $y ]    ;#代码换行
=>$x
```

当用 set 命令为参数赋值并“\”用做续行时, 小心不要在\后同一行上再输入任何符号, 包括空格符和制表符。否则反斜杠就起不到续行的功能, 易引起错误, 如果是交互式行命令输入, 这种错误立刻就可以检查出来, 而如果是包含在脚本文件中则较难查出。下例就列举了一些不规范的续行操作:

例 2-7 不规范的续行

```
;#下面的命令, 反斜杠后多输入了两个空格, 换行后直接输入"]"结束。结果 a 被赋值为空格
% set a [list set var \
]
=>set var { }
%set ver \ ;#want set ver to "hello",but ver will be set to blank.

%
```

第 3 章. 数学表达式与 expr 命令

尽管 Tcl 是基于字符串操作的，但它仍旧提供了有效数学运算和逻辑运算的功能。通过命令 `expr` 可以实现对数学表达式的分析和计算。

3.1 数学和逻辑运算符

表 3-1 数学与逻辑运算符

运算符	说明
<code>- + ~ !</code>	一元减（取负）、一元加（取正）、比特反、逻辑非
<code>* / %</code>	乘、除、取余（二元操作符）
<code>+ -</code>	加、减（二元操作符）
<code><< >></code>	左移、右移（二元操作符）
<code>< <= > >=</code>	布尔小于、小于或等于、大于、大于或等于
<code>== !=</code>	布尔等、不等
<code>&</code>	比特与
<code>^</code>	比特异或
<code> </code>	比特或
<code>&&</code>	逻辑与
<code> </code>	逻辑或
<code>x ? y:z</code>	三重操作符：根据 x 的值（true or false）在 y 和 z 两个结果中进行选择。x 为 true（=1）则选择 y，否则选择 z。

3.2 数学函数

表 3-2 数学函数

函数名	说明	举例
<code>abs(arg)</code>	取绝对值	<code>set a -10 ; #a=-10</code> <code>set a [expr abs(\$a)]; # a=10</code>
<code>acos(arg)</code>	反余弦	<code>set p 1.0; set a [expr acos(\$p)] => a=0.0</code>
<code>asin(arg)</code>	反正弦	
<code>atan(arg)</code>	反正切	
<code>atan2</code>	比值取反正切	
<code>ceil(arg)</code>	返回不小于 arg 值的整数值	
<code>cos(arg)</code>	余弦	
<code>cosh(arg)</code>	双曲余弦	
<code>double(arg)</code>	转换双精度	
<code>exp(arg)</code>	exp 运算（e 的幂）	
<code>floor(arg)</code>		
<code>fmod</code>	取余（结果为浮点型）	
<code>hypot(x,y)</code>	根据直角三角形两直边长度计算出斜边长度	
<code>int(arg)</code>	取整	
<code>log(arg)</code>	自然对数	
<code>log10(arg)</code>	以十为底的对数	
<code>pow</code>	幂运算	
<code>rand()</code>	取 0 到 1 之间的随机实数（无输入参数）	<code>set a [expr 10 * [expr rand()]]; 0 到 10 之间随即实数</code>

round(arg)	四舍五入取整数	%set a [expr round(10.5)]; =>11
sin(arg)		
sinh(arg)		
sqrt(arg)	求二次根	
srand(arg)	以整数 arg 为随机数生成器的种子产生随机数	
tan(arg)		
tanh(arg)		

3.3 数学运算举例

例 3-1 数学运算举例

```
% set var1 5
% set var2 3
% set res [expr $var1 /$var2]
=> 1                      ;#因为 var1 和 var2 的值都是整数，结果也只取整数部分值
% set var1 5.0
% set res [expr $var1 /$var2]
=>1.66666666667          ;#结果为浮点数

%set var1 3
%incr var1 2
=>5
%puts $var1
=> 5
%incr var1 -4
=>
1                          ;#var1 的值现在为 1
```

3.4 incr 命令

incr 命令根据指定的步长来增加或减少参数的值。当步长为负时，减少参数值；当步长为正时，增加参数值。默认步长为+1。

[语法]: incr varName ?step?

如:

```
%set a 10 ; incr a
=>a=11
%set a 10 ; incr a -1
=> a=9
```

第 4 章. 字符串

4.1 基本命令集

Tcl 将所有的变量值视作字符串，并将他们作为字符串来保存。

下表列出了字符串操作的几个常用命令。

表 4-1 比较有用的字符串操作命令

命令	描述
append	将值追加到字符串尾
binary	二进制字符串操作
format	字符串格式化
regexp	正则表达式（用于字符串模式匹配）
regsub	用正则表达式进行字符串模式匹配和替换
scan	字符串分解
string options	字符串操作和命令集
subst	字符替代（替代特殊字符）

本章主要讲述 append、format、scan、binary、subst、string 等命令。regexp、regsub 在正则表达式一章讲述。

4.2 append 命令

append 命令比较简单，它将一段字符串连接到另一字符串尾部从而组成新的字符串。此命令对变量直接修改。

[语法]: append varName ?value?

例 4-1 append 命令的简单例子

```
%set var1 Hello
%set var2 World
%append var1 $var2
=>HelloWorld
%puts $var1
=>HelloWorld ;#var1 的内容被更改!
```

4.3 format 命令

4.3.1 format 命令说明

format 命令和 C 语言中的 printf 和 sprintf 命令类似。它根据一组格式说明来格式化字符串。此命令不会改变被操作字符串的内容。

[语法]: format spec value1 value2 ...

spec 变元包含了格式说明关键词和附加文字。使用%来引入一个关键词，后跟 0 个或者多个修饰符，然后使用一个转换格式符结尾。

关键词的基本格式是 “%aaaB”:aaa 是修饰符，B 代表一种格式转换符。例如%f 用于将对应位置的参数转化为浮点数。如果要使用“%”号，则可以使用%%来实现。否则会将%后的字符作为关键词来处理。

valueX 是变元。对每个变元来讲，其关键词可多达 6 部分：

- 位置说明符；
- 标志；
- 字段宽度；

- 精度;
- 长度;
- 转换符。

转换符之外的关键词都可以看作修饰符。

表 4-2 格式转换符

字符	说明
d	有符号整数
u	无符号整数
i	有符号整数。变元可以是十六进制 (0x) 或八进制 (0) 格式
o	无符号八进制数
x 或 X	无符号十六进制数
c	将整数映射到对应的 ASCII 字符
s	字符串
f	浮点数
e 或 E	科学记数法表示的浮点数
g 或 G	以 %f 或 %e 格式 (要短一些) 来表示的浮点数

表 4-3 格式标志符

标志	说明
-	使字段左对齐
+	字段右对齐
space	在数字前加一个空格, 除非数字带有前导符号。这在将许多数字排列在一起时非常有用
0	使用 0 作为补白
#	前导 0 表示八进制, 前导 0x 表示十六进制数。浮点数中总要带上小数点。不删除末尾的 0(%g)

位置说明符 i\$ 表示从第 i 个变元取数值而不是根据通常的位置对应关系对应的变元。位置记数从 1 开始。

4.3.2 format 举例

例 4-2 位置说明符的例子

```

;# 要取第 2 个变元值, 即 5。位置说明符的格式为 2$, 并用 \ 来引用符号 $:
% set res [format "%2\$$" 1 5 9]
=>5
%puts $res
=>5
% set str [format "%3\$$ %1\$$ %2\$$" "are" "right" "You"]
=> You are right

```

例 4-3 format 命令的简单例子

```

%format "%x" 20
=>14                ;# 将 20 转换为十六进制数
%format "%8x" 20
=> 14                ;# 将 20 转换为十六进制数, 8 位数据宽度, 右对齐
%format "%08x" 20

```

```

=>00000014           ;#与上一命令相似，但用 0 添齐
%format "%-8x" 20
=>14                 ;#宽度 8 位，左对齐
%format "%#08x" 20
=>0x000014           ;#将 20 转换为十六进制数，并添加前缀"0x"，
                     ;#总宽度为 8 为，右对齐（默认），前导空格用 0 补齐。
% set c [format %c%c 40 30000 41 ]
=> (3.000000e+004) ;#%c 将整数转换为对应的 ASCII 字符，40 和 41 分别对应 "("和")"
% set num [scan Aa %c var1 ]
=> 1                 ;#将字符串"Aa"的第一个字符"A"转换为对应整数，
                     ;#并赋值给 var1 变量，返回成功转换个数

% puts $var1
=> 65 ;#字符 A 对应的整数

```

上例最后一个 format 格式命令中，"%c"一个整数转换为对应 ASCII 字符输出，如果将一个字符转换为对应的整数，就需要"scan"命令来完成。"scan"命令见后面一节介绍。

4.4 scan 命令

scan 命令根据格式描述符来解析一个字符串并将对应值赋给后面的变量。返回成功转换的个数。

[语法] scan string format var? var?

scan 的格式描述几乎与 format 相同，但不用%u 格式。%c 的作用与 format 中的相反，是将一个 ASCII 字符转换为对应的整数值。

应该注意 scan 命令中位置顺序和 format 中的不一样。format 将多个目标变量转换成一个字符串，而 scan 则可将一个字符串分解为多个变量。

scan 格式包含有一种集合的概念。它使用方括弧来界定一组字符，这个集合匹配拷贝字符串的一个或多个字符到目的变量中去。这种集合的概念在以后的 regexp 正则表达式中应用更多。

如果 scan 命令中，没有指定输出变量，则它不返回成功转换个数，而是返回成功转换的结果，见下例说明。

例 4-4 scan 命令的简单例子

```

%set num [scan "abcABC" "%c%c" var1 var2]
=> 2
%puts "$var1 $var2"
=> 97 98
% set num [scan "abcABC" "%c%c" ]
=> 97 98
%puts $num
=> 97 98 ;#忘记指定输出变量的结果

;#格式描述说明要扫描小写的 a 到 z 区间的连续字符,用到了花括号表明是一个集合
% scan "abcABC" {%[a-z]} res
=> 1
% puts $res
=> abc
% scan "ABCabc" {%[A-b]} res
=> 1
% puts $res
=> ABCab

```

```
% scan "ABCa" {%[^a-b]} res
=>1                      ;#匹配非 a 到 b 的字符
% puts $res
=>ABC
% scan ABCa {%2[^a-b]} res
=>1                      ;#照样可以指定匹配字符数
% puts $res
=>AB
```

上例后面几个 scan 语句是一种字符集操作。方括号对 Tcl 有特殊的意义，用花括号或者反斜杠可以保护这种格式。字符 ^ 说明匹配所有不在集合中的字符。

4.5 binary 命令

Tcl8.0 之后增加了对二进制字符串的支持。

根据数据的存储组织形式，可以分为 ASCII 形式和二进制形式。按 ASCII 存放时，每个字节存放一个 ASCII 代码，代表一个数字字符。以二进制存放的时候，将存储数字的二进制值。比如一个整数 10000，用 ASCII 码表示时，每个数字字符用一个字节的 ASCII 码存储，共需要 5 个字节存储空间。而用二进制，则只需两个字节，来存储对应的二进制码

“10011100010000”（十六进制=2710H）。所以用二进制要比 ASCII 码节省存储空间。在内存中，都是用二进制存储数据。

binary format 就是将数值根据规定模式对 Tcl 的普通数据进行二进制压缩，而 binary scan 作用相反，是从二进制数值恢复 Tcl 普通数据。例 4-5 列举了这两个 binary 命令的作用。首先用 binary format 对数值字符串“25664”进行二进制压缩，然后从压缩了的二进制值用 binary scan 恢复。

例 4-5 体验 binary format 和 binary scan 的作用

```
%set b [binary format "s" 25664]
=> @d                      ;#整数 25664 以十六进制表示为 6440H。"s"为格式说明
                              ;#符，按照由低到高字节顺序存储。
% puts $b
=> @d                      ;#数值被压缩为两个字节，输出的结果为两个压缩字节对
                              ;#应的两个字符，@的 ASCII 码为 40H，d 的 ASCII 码为 64H
% binary scan $b "s" var
=> 1                      ;#返回转换成功的次数
% puts $var
var                          ;#binary scan 从二进制数 b 中将数值恢复并存入变量
=> 25664

% binary scan 1 c var
=>1                      ;#从字符 1 中恢复数值。字符 1 的 ASCII 码=31H，31H 对应十
                              ;#进制整数为 49
% puts $var
=> 49
```

[语法]: `binary format spec value1 ?value2 ...?`
`binary scan str spec var1 ?var2 ...?`

spec 为格式描述字符串。

格式描述模板包含类型关键字和记数值（type count）两个部分。下表总结了这些类型。表中，类型为跟在类型关键字字母后的可选记数。

表 4-4 二进制转换类型

类型	说明
a	包含 count 个字符的字符串。在 binary format 中以空字符作为补白
A	和 a 功能相同，只不过使用空格符而不是空字符作为补白。
b	长度为 count 的二进制字符串，以 0 和 1 组成，按照从低到高的 bit 位顺序排列
B	长度为 count 的二进制字符串，以 0 和 1 组成，按照从高到低的 bit 位顺序排列
h	长度为 count 的十六进制字符串，按照从低到高的字节顺序组成
H	长度为 count 的十六进制字符串，按照从高到低的字节顺序组成
c	一个 8 位字符编码。binary scan 中会从字符串中将字符转换为对应整数
s	字节顺序为 little-endian 的 16 位整数。count 用于指定重复特性
S	字节顺序为 big-endian 的 16 位整数。count 用于指定重复特性
i	字节顺序为 little-endian 的 32 位整数。count 用于指定重复特性
l	字节顺序为 big-endian 的 32 位整数。count 用于指定重复特性
f	本机格式的单精度浮点数。count 用于指定重复特性
d	本机格式的双精度浮点数。count 用于指定重复特性
x	使用 binary format 放置 count 个空字节。使用 binary scan 跳过 count 个字节
X	回退 count 个字节
@	跳到由 count 指定的绝对位置。如果 count 为*则跳到末尾

关于 binary 命令的详细介绍，请读者参考相关书籍和 Tcl 的帮助。下面给出了 binary 命令的另外一些例子。

例 4-6 binary 命令的简单应用

```
;(1) #获得一个字母对应的整数值
% binary scan "abc" "c" var1
=>1
% puts $var1
=> 97           ;#字母 a 对应的整数值
;(2) #获得字符串中多个字母对应的整数值，并保存到同一列表中或者分别保存
% binary scan "abcd" "c3" val
=>1             ;#根据模板成功完成一次转换
% puts $vala    ;#结果保存到一个变量中，以空格间隔
=>97 98 99
% binary scan "abcd" "ccc" vala valb valc
=>3             ;#分三次进行，一次一个字符
% puts "$vala $valb $valc"
=> 97 98 99

;#(3) 将一个 8 位二进制数（0，1 组成）转换成对应字符
% binary format B8 01001001      ;#十六进制为 49H
=> l
```

注意：用 binary format 压缩的字符串，需要在同一系统上用 binary scan 来恢复。不同系统上的 binary 字符串可能处理的方式有所不同。

4.6 subst 命令

subst 命令在字符串中搜索方括号、美元符号和反斜杠，并对其进行替换操作，而对其他数据不做处理。字符串内部的花括号对这种替换操作不阻止。

例 4-7 subst 命令的简单例子

```
% subst { a=$a sum=[expr 1 + 2]}
=> a=foo bar sum=3
% subst { a={ $a } sum={ [expr 1 + 2] }}
=> a={foo bar} sum={3} ;#数据内部的花括号不阻止替换
```

4.7 string 命令

字符串是 Tcl 中的基本数据类型，所以有大量的字符串操作命令。一个比较重要的问题就是模式匹配，通过模式匹配将字符串与指定的模式（格式）相匹配来进行字符串的比较、搜索等操作。本节的 string 命令提供了一些简单的模式匹配机制。而正则表达式则提供了更为复杂、更为强大的模式匹配机制。关于正则表达式见正则表达式一章介绍。

4.7.1 string 命令列表

下表给出了各种 string 命令语法格式和说明。

表 4-5 string 命令

命 令	说 明
string bytlength <i>str</i>	返回用于存储字符串的字节数。
string compare <i>?-nocase? ?-length len? str1 str2</i>	根据词典顺序比较字符串。 <i>-nocase</i> 选项表示大小写无关。 <i>-length</i> 选项表示只比较指定长度的开头部分字符。如果字符串相同就返回 0 (<i>str1=str2</i>)，如果 <i>str1</i> 的顺序比 <i>str2</i> 靠前就返回 -1 (<i>str1<str2</i>)，其他情况返回 1 (<i>str1>str2</i>)。
string equal <i>?-nocase? str1 str2</i>	比较字符串，相同返回 1，否则返回 0。
string first <i>str1 str2</i>	返回在 <i>str2</i> 中 <i>str1</i> 第一次出现的索引位置，如果没有找到则返回 -1
string index <i>str index</i>	返回指定位置的字符。 <i>index</i> 号从 0 开始。如果 <i>index</i> 为 <i>end</i> 则返回最后一个字符。
string is <i>class ?-strict? ?-failindex varname? str</i>	判断字符串的类型，如果是指定类型就返回 1。字符类型例如有整型、布尔型等等。如果使用了 <i>-strict</i> 则表示不匹配空字符，否则总是匹配的。如果指定了 <i>failindex</i> 则将 string 中非 <i>class</i> 的字符索引赋给 <i>varname</i> 变量。详细的字符类见后面的介绍（表 4-7）。
string last <i>str1 str2</i>	返回 <i>str2</i> 在 <i>str1</i> 最后一次出现的位置索引。没有搜索到就返回 -1。
string length <i>str</i>	返回 <i>str</i> 中的字符个数
string map <i>?-nocase? charMap str</i>	返回根据 <i>charMap</i> 中输入、输出列表将 <i>str</i> 中的字符进行映射后而产生的新字符串。参见本节“字符串映射”部分。
string match <i>pattern str</i>	如果 <i>str</i> 匹配 <i>pattern</i> 就返回 1，否则返回 0。使用的是通配风格的匹配。参见本节的“字符串的匹配”部分。
string range <i>str index1 index2</i>	返回 <i>str</i> 中从 <i>index1</i> 到 <i>index2</i> 之间的字符串。
string repeat <i>str count</i>	返回将 <i>str</i> 重复 <i>count</i> 次的字符串。
string replace <i>str first last ?newstr?</i>	将从 <i>first</i> 开始到 <i>last</i> 结束的一段字符串替换为 <i>newstr</i> 字符串。如果 <i>newstr</i> 没有，则这部分字符串内容会被删除。
string tolower <i>str ?first? ?last?</i>	将指定范围的字符转化为小写格式。
string totitle <i>str ?first? ?last?</i>	通过将第一个字符替换为 Unicode 的标题型字符或大写形式，而其余的替换为小写形式的方法将 <i>str</i> 转换为开始字母大写形式。可以用参数指定操作范围。
string toupper <i>string ?first? ?last?</i>	将指定范围的字符转化为大写形式。
string trim <i>str ?chars?</i>	从 <i>str</i> 两端删除 <i>chars</i> 中指定的字符。 <i>chars</i> 默认为空字符。
string trimleft <i>str ?chars?</i>	从 <i>str</i> 开头删除 <i>chars</i> 中指定的字符。 <i>chars</i> 默认为空字符。
string trimright <i>str ?chars?</i>	从 <i>str</i> 结尾删除 <i>chars</i> 中指定的字符。 <i>chars</i> 默认为空字符。
string wordend <i>str index</i>	返回 <i>str</i> 中在索引位置 <i>index</i> 包含字符的单词之后的字符的索引位置。
string wordstart <i>str index</i>	返回 <i>str</i> 中在索引位置 <i>index</i> 包含字符的单词中第一个字符的索引位置。

说明：命令中的两个问号之间的内容是任选项，表示根据实际需要可选的内容。

字符串的这些命令的使用方法比较相似。常用的字符串操作有：

- string match: 字符串匹配（或者比较）；
- 大小写转换: tolower 和 toupper 以及 totitle；
- equal 操作；
- string compare；
- string range；
- string replace 等。

4.7.2 字符串比较

我们在 expr 和控制语句如 if、while 中可用比较运算符"=="、"!="、"<"、"<="和">"等来进行字符串比较，但是如不注意的话就会产生问题。首先必须用双引号来将字符串值括起来，这样表达式语法分析器才能按照字符串类型来进行识别。然后必须用花括号将整个表达式括起来以阻止主解释器将双引号去掉：

```
if {$x == "true"} {puts ok}
```

然而，这样的直接比较还是会带来其他意想不到的问题。比较安全的方法是使用 string compare 和 string equal 来操作，而且这些 string 命令的执行速度也更快。

例 4-8 string compare 和 string equal 进行字符串比较的例子

```
%set s1 abc
=>abc
%set s2 abd
=>abd
%if { [string compare $s1 $s2] == 0 } {
    puts "s1 is same as s2"
} else {
    puts "s1 isn't same as s2"
}
=> s1 isn't same as s2
%if { [string equal $s1 $s2] } {
    puts "s1 is same as s2"
} else {
    puts "s1 isn't same as s2"
}
=> s1 isn't same as s2
```

注意命令返回值：string compare 在不同的情况下返回 1，并返回 0。而 string equal 和 string match 则恰恰相反，相等或者匹配时返回 1，不同返回 0。

4.7.3 string match 字符串匹配

string match 命令沿用了各类 UNIX shell 中所使用的文件名模式匹配机制。表 4-6 给出了匹配模式的三种结构。

表 4-6 使用 string match 来匹配字符的结构

字符	说明
*	通配符。匹配任意数量和值的任意字符
?	匹配一个字符
[chars]	匹配 chars 中的任意一个字符

为了使结果返回 1（匹配），pattern 和字符 str 必须相同，除非你是使用了匹配字符。

下面的命令匹配以 a 开头的字符串，使用了通配符"*"：


```
%string match a* alpha
=>1
```

下面的例子要用"?"来匹配有两个字符的字符串:

```
%string match ? XY
=>0 ;# 不匹配
string match ?? XY
=> 1 ;#ok
```

一个"?"对应一个字符, 为了匹配两个字符, 必须输入两个问号。
要匹配以 a 或者 b 开头的字符串:

```
%string match {[ab]*} bell
=>1
```

string match 支持匹配字符集。字符集要用花括号括起来以便 Tcl 正确理解方括号内是匹配模式而不是嵌套的命令。另一种方法可将模式放置在一个变量中:

```
%set pat {[ab]*x}
%string match $pat box
=> 1
```

也可以使用语法[x-y]来指定将要匹配的一个区段的字符。如[a-m]就表示所有从 a 到 m 的小写字母集。一个集合也不限定一个区段:

```
%string match {[a-z0-9]} 7
=>1
```

上面的匹配集合一个方括号只匹配一个字符。要匹配多个字符, 可以组合多个匹配格式:

```
%set st "apo"
%string match {[a-c][o-q]?} $st
=> 1
```

另外, 如果需要匹配字符串中的* 和?时, 就要在其前面用反斜杠标明:

```
%string match {*\?} "who are u?"
=>1
```

这种情况下, 要用花括号将模式括起来, 否则 Tcl 解释器还会进行反斜杠替换(花括号有阻止替换操作的作用), 如果不用花括号, 就要用两个反斜杠以便 Tcl 解释器进行反斜杠替换从而将两个反斜杠替换成一个:

```
%string match *\a "who a"
=> 0 ;#"\\a"被进行了替换操作(振铃)
%string match *\\a "who a"
=> 1
```

4.7.4 字符串替换

`string replace` 可以用新的字符串代替字符串中指定范围内的字符，如果没有指定新字符串，则指定范围内的字符都会被删除。另外注意：替换不改变原来字符串变量的值，只是返回更改后的新字符串。

例 4-9 字符串替换简单例子

```
%string replace aaaabbbb 1 3 ccc
=>acccbddd
%string replace aaaabbbb 1 3
=>abbbb
%set a aaabbb
%string replace $a 1 2
abbb
% puts $a
aaabbb ;#a 的值并没有改变
```

4.7.5 字符类别(class)测试

`string is` 命令用来测试一个字符串是否属于某个特定的类 (class)。它对于进行参数输入合法性检查非常有效。比如，要确保输入参数是整数，则可以这么做：

```
if {[string is integer $input]} {
    error "Invalid input parameter: $input. Please enter a integer number"
}
```

`if` 语句检查输入参数 `input` 的值是否为 `integer`，如果不是则报错。

命令 `string is` 当正确时返回 1，不正确返回 0，`!"` 是逻辑非操作。`error` 命令和 `puts` 作用相似，但 `error` 会终止程序。

类是按照 Unicode 字符集定义的，它们要比 ASCII 编码方式指定范围的字符集更通用。下表列举了这些类。

表 4-7 字符类名目

字符类	说明
<code>alnum</code>	任何字母或数字字符
<code>alpha</code>	任何字母字符
<code>ascii</code>	任何具有 7 位字符编码的字符（即，小于 128）
<code>boolean</code>	0, 1, true, false（不分大小写）
<code>control</code>	字符编码小于 32 而又不是 NULL 的字符
<code>digit</code>	任何数字字符
<code>double</code>	有效浮点数
<code>false</code>	0, false（不分大小写）
<code>graph</code>	不包含空格字符在内的任何打印字符
<code>integer</code>	有效整数
<code>lower</code>	全为小写的字符串
<code>print</code>	<code>alnum</code> 的同义词
<code>punct</code>	任何标点符号
<code>space</code>	空格符、制表符、换行符、回车、垂直制表、退格符
<code>true</code>	1, true（不分大小写）
<code>upper</code>	全为大写的字符串
<code>wordchar</code>	字母、数字和下划线

xdigit	有效的十六进制数字
--------	-----------

4.7.6 字符串映射

`string map` 命令根据字符映射对字符串进行转换。映射以输入、输出表的形式表示。凡是字符串中包含有输入序列的地方都使用相应的输出序列替换。输入、输出要成对使用：

```
%string map {f p d l} "food"  
=>pool
```

上例命令中 `{f p d l}` 为输入、输出表，输入、输出项成对出现：`f`、`d` 为输入，`p`、`l` 为输出，`f` 对应 `p`，而 `d` 对应 `l`。命令的结果是字符串 `food` 中的 `f` 被 `p` 替换，`d` 被 `l` 替换。

输入和输出项可不止一个字符而且不要求长度相同：

```
%string map {f pp d ll oo a} "food"  
=>ppall
```

`string map` 命令和 UNIX shell 的 `tr` 命令比较相似。

第 5 章. Tcl 列表操作

本章讲述 Tcl 列表操作。列表则是具有特殊解释的字符串。Tcl 中的列表操作和其它 Tcl 命令一样具有相同的结构。

列表可应用在诸如 `foreach` 这样的以列表为变元的循环命令中，也用于构建 `eval` 命令的延迟命令字符串。

5.1 列表命令集

表 5-1 列表相关命令

命令	说明
list <i>arg1 arg2 ...</i>	创建一个列表
lindex <i>list index</i>	返回列表 <i>list</i> 中的第 <i>index</i> 个元素 (element) 值
llength <i>list</i>	计算列表 <i>list</i> 元素个数
lrange <i>list index1 index2</i>	返回指定范围内 (从 <i>index1</i> 到 <i>index2</i>) 的元素
lappend <i>list arg1 arg2 ...</i>	将新元素追加到原来列表 <i>list</i> 后组成新的列表
linsert <i>list index arg1 arg2 ...</i>	将新元素插入到 <i>list</i> 中位于 <i>index</i> 元素之前的位置上
lreplace <i>list index1 index2 arg1 arg2 ...</i>	替换指定范围的元素
lsearch <i>?mode? list value</i>	根据匹配模式 <i>mode</i> ，查找 <i>list</i> 中与 <i>value</i> 匹配的元素位置索引。 <i>mode</i> 一般为 <code>-exact</code> 、 <code>-glob</code> 和 <code>-regexp</code> 。默认为 <code>-glob</code> 。找不到返回 -1。
lsort <i>?switches? list</i>	根据 开关选项对列表进行排序
concat <i>list1 list2 ...</i>	连接多个列表内容成一个列表
join <i>list joinChars</i>	以 <i>joinChars</i> 为分隔符将列表中的元素合并在一起
split <i>string splitChars</i>	以 <i>splitChars</i> 中的字符作为分隔符将字符串分解为列表元素。
foreach <i>var list {proc body}</i>	遍历列表各项，逐次将各元素值存入 <i>var</i> 中并执行 <i>proc body</i> 。相当于一个循环控制语句。

`foreach` 命令将在控制结构一章介绍。

5.2 list 命令

`list` 命令用来创建列表。

一个列表可以包含子列表，即列表可以嵌套。

例 5-1 使用 `list` 命令创建列表

```
% set l1 [list Sun Mon Tues]
=>Sun Mon Tues                ;#列表 l1 含有三个元素
% set l2 [list $l1 Wed]
=> {Sun Mon Tues} Wed          ;#列表 l2 中含有两个元素。第一个元素用花括号括起来。
% set str1 "Sun Mon Tues"
=>Sun Mon Tues
% set l2 [list $str1 Wed]
=>{Sun Mon Tues} Wed           ;#和上面的命令结果相同：“列表是特殊的字符串”。
%set l1 [list "Sun Mon Tues" "Wed"]
=>{Sun Mon Tues} Wed           ;#当元素是一个字符串时，被用花括号括起来了
%set l2 [list $l1 "Thur"]
=>{{Sun Mon Tues} Wed} Thur    ;#两个元素，理解花括号的作用。
```

```
%set b 10
% set l3 [list { a $b c} d]
=>{a $b c} d           ;#花括号阻止引用替换
% set l3 [list "a $b c" d]
=> {a 10 c} d
```

在创建 list 的时候比较灵活，有时比较难以理解和容易出错，特别是花括号。如何理解花括号的作用呢？可以这么简单认为：花括号内部代表的是子列表。当用 list 命令创建列表的时候，如果元素是单个的词，就不用大括弧括起来，但如果某个元素是以空格分割的字符串时，就将其看作一个子列表而用花括号括起来。注意实际的处理过程并不是这样的，要复杂一些。

5.3 concat 命令

concat 命令以空格为分隔符将多个列表拼装在一起形成新的列表。它和双引号的作用比较相似。

list 命令和 concat 命令都可以完成列表合并功能。list 和 lappend 命令保留每个列表的结构，将每个列表作为一个整体生成新列表的元素来完成。而 concat 命令则要先把各个列表的最外层列表结构去掉，将其中的所有元素取出来作为新列表的元素来完成合并，即新列表的每个元素也是合并前列表的元素。这个区别在后面动态建立 Tcl 命令的时候显得尤为重要。

例 5-2 concat 命令的例子

```
%set x {1 2}
=> 1 2
% set y "$x 3"           ;#$x 被替换后，作为列表结构的花括号被去掉，
                        ;#元素被提出来和 3 一起作为新列表的元素
=> 1 2 3
% set y "$x {3}"
=> 1 2 {3}
% set y [concat $x 3]    ;#结果同上面的双引号
=> 1 2 3
% set y [concat $x {3}]
=> 1 2 3
% set y [list $x 3]      ;#list 命令保留了每个列表的结构，将列表，而不是列表中的元素
=> {1 2} 3
% set y [list $x {3}]
=> {1 2} 3
% set y [lappend x 3]
=> {1 2} 3
```

5.4 lappend 命令

lappend 命令用来将新元素追加到列表末尾。

例 5-3 lappend 命令简单例子

```
% lappend new 1 2
=> 1 2
%lappend new {3 4} "5" {6} 7
=> 1 2 {3 4} 5 6 7      ;#单个词的元素的双引号和花括号被剥离了
% set new
=>1 2 {3 4} 5 6 7
```

由第一个命令可见，lappend 命令也可以用来创建一个列表。

5.5 llength 命令

llength 命令可以获得一个列表内元素的个数。

例 5-4 llength 命令的一个简单例子

```
% set l1 "1 2 3 4 5"
=>1 2 3 4 5           ;#定义了一个字符串
% set num [llength $l1]
=>5                   ;#这里 l1 被看作列表了
```

这个例子中，先前定义的一个字符串被看作一个 list！（Tcl 中的 list 只不过是一个有特殊解释的字符串）。

列表是用空格搁开的多个元素组成的字符串，而通过 list 相关命令得到了特殊解释。既然 list 是字符串，那么所有适用于字符串的操作命令也应该适用于列表。

5.6 lindex 命令

lindex 命令返回列表中指定位置的特定元素。列表索引从 0 开始记数！

例 5-5 lindex 命令的一个简单例子

```
%set x { 1 4 5 }
=> 1 4 5
% lindex $x 1
=>4
%lindex $x end
=>5
%lindex $x end-1
=>4           ;#获得倒数第二个元素
%lindex $x first
=>bad index "first": must be integer or end?-integer? ;#出错，可以通过这种方法获得帮助
```

5.7 lrange 命令

lrange 命令返回一个指定区段的列表元素，可以以 end 或者 end-n 作为索引（n 为正整数）。

```
% lrange {1 2 3 {4 5} 6} 2 end
=> 3 {4 5} 6
```

5.8 linsert 和 lreplace 命令

linsert 命令用来将元素插入到一个列表的由索引指定的位置。如果索引为 0 或者更小，则元素就会被添加到最前面。如果索引值大于或者等于列表长度，则元素被追加到列表尾部。其他情况元素被添加到指定位置之前。

lreplace 命令将一个指定区段的列表元素替换为新元素。如果没有指定新元素，则这个区域的元素就会被从列表中删除。

注意：这两个操作不会改变原来列表的内容，而是返回一个新列表。

例 5-6 linsert 和 lreplace 命令的例子

```
%set x {1 2}
=>1 2
% set new [linsert $x 0 he she]
=>he she 1 2
```

```
% set new [linsert $x end he she]
=>1 2 he she
() 372 % set new [linsert $x 1 he she]
=>1 he she 2
() 373 % puts $x
=>1 2           ;# x 的值没有改变

() 374 % puts $new
=>1 he she 2
() 375 % set y [lreplace $new 1 2 B C]
=>1 B C 2
() 376 % set y [lreplace $new 0 0]
=>he she 2
% puts $new
=>1 he she 2           ;#new 的内容并没有改变，这和 string replace 相同。
% set y [lreplace $new 1 2]
=>1 2           ;#将第 1、2 个元素删除
```

5.9 lsearch 命令

`lsearch` 命令在给定列表中搜索与匹配字符串匹配的元素，成功就返回正确的元素索引，否则返回-1。`lsearch` 支持通配符格式，但可以使用 `-exact` 选项将其屏蔽而进行精确匹配。

例 5-7 `lsearch` 命令的简单例子

```
% set l1 [list This is one list]
=> This is one list
% set index [lsearch $l1 l*]
=> 3
% set index [lsearch -exact $l1 l*]
-1
% set index [lsearch -exact $l1 list]
3
```

下面的例子是用 `lsearch` 和 `lreplace` 一起实现在 `list` 内删除所有和匹配值匹配的元素。`proc` 是过程定义命令，定义了一个过程 `ldel`。关于过程请参见后面的相关章节。`for` 循环语句实现循环操作，请参见控制结构一章。

例 5-8 `lsearch` 与 `lreplace` 结合删除列表元素

```
% proc ldel { list value } {
  set ix [lsearch -exact $list $value]
  for {} { $ix >= 0 } {} {
    set list [lreplace $list $ix $ix]
    set ix [lsearch -exact $list $value]
  }
  return $list
}
% set l1 [list 123 234 123 345 123 456]
% set val 123
% set l2 [ldel $l1 $val]
=> 234 345 456
```

5.10 lsort 命令

lsort 命令实现对列表的排序。排序操作不影响原表，而是返回排序之后的新表。

排序的方式有多种选择，可以通过 -ascii、-dictionary、-integer、-real 来指定基本排序类型，然后使用 -increasing、-decreasing 指定排列方式，默认为 -ascii、-increasing。要注意 ASCII 排序时使用字符编码；而 dictionary 排序方式整合大小写，并将包含的数字以数值大小来处理。

例 5-9 lsort 命令简单的例子

```
%set list "a Z z n100 n200 M p Hl hL m 1 20"
=>a Z z n100 n200 M p Hl hL m 1 20
% lsort -ascii $list
=>1 20 Hl M Z a hL m n100 n200 p z
% lsort -dictionary $list
=>1 20 a Hl hL M m n100 n200 p Z z
```

5.11 join 与 split 命令

join 命令接收一个列表，并用指定的分隔符将列表元素整合成一个字符串

```
%join {1 {2 3} {4 5 6}} :
=> 1:2 3:4 5 6
```

split 命令的作用与 join 的作用相反，它接收一个字符串，并根据给定的字符将其分割转换成一个列表。用于分割的字符应该在字符串中存在，否则 split 因为没有搜索到对应字符而将整个字符串作为唯一列表元素而返回，即返回原字符串。

例 5-10 split 命令例子

```
%set str cm8/auto/tools/aries/ASAM/NE/SNMP/IMPL/ne_create_board.tcl
=>cm8/auto/tools/aries/ASAM/NE/SNMP/IMPL/ne_create_board.tcl
% set s /
=>/
% set l1 [split $str $s]
=> cm8 auto tools aries ASAM NE SNMP IMPL ne_create_board.tcl
% set l2 [split $str "/."] ;#可以指定多个分割符
=> cm8 auto tools aries ASAM NE SNMP IMPL ne_create_board tcl
```

split 的默认分割符为空白符，包括空格符、制表符和换行符。如果分割符在字符串开始位置，或者有多个分割符相连，那么 split 命令就会产生空列表元素，并用 {} 表示，分割符并不被合并。

若打算将字符串的每个字符都区分开，即将每个字符都分割成列表元素，可以将分割符指定为空字符串 {}，这个方法对分析和处理字符串中的每个字符时比较有用。当遇到字符串内含有特殊的字符，如空格符时，split 也将其作为一个字符元素处理，为了利于区别起见，用花括号将空格元素括起来。

例 5-11 空元素与独立字符元素的 split 例子

```
%set str "/cm8 is a Vob directory, but /home/usr isn't one vob directory!"
=>/cm8 is a Vob directory, but /home/usr isn't one vob directory!
% set l3 [split $str "/" , '!']
=>{} cm8 is a Vob directory {} but {} home usr isn t one vob directory {}
% set l4 [split $str {}]
```

```

=>/cm8{ }is{ }a{ }Vob{ }directory,{ }but{ }/home/usr{ }isn't{ }o
ne{ }vob{ }directory!
% set str "hello"
=>hello
%set l5 [split $str {}]
=>h e l l o

```

上面的例子中，l3 和出现了空元素，l4 中出现了空格元素（注意区分一下）。即便只给人看也会让人不舒服，如果能将列表在整理一下，去掉空元素和空格元素该多好！该如何处理呢？可用例 5-8 lsearch 与 lreplace 结合删除列表元素的函数 ldel 来进一步处理：

例 5-12 用 ldel 删除指定元素

```

%ldel $l5 {} ;#元素含有一个字符，是空格符
=>/cm8isaVobdirectory,but/home/usrisn'tonevobdirectory!
%ldel $l3 {} ;#元素是空元素，花括号内没有字符
=>cm8 is a Vob directory but home usr isn t one vob directory

```

为了进一步说明列表操作命令的功能，下面再给一个 list 命令的例子。例 5-13 的函数 incr_mac 是对给定的一个点分十进制表示的 MAC 地址（格式 xxx.yyy.zzz.www.uuu.vvv）按照指定步长调整一次，步长也是一个点分十进制 MAC 地址值。

例 5-13 根据步长调整 MAC 地址值

```

proc incr_dotdec_mac_one_step { args } {
    array set myarr $args
    set mac $myarr(-mac)
    set step $myarr(-step)

    #分解 MAC 地址
    set mac_list [split $mac "."]
    set mac1 [lindex $mac_list 0]
    set mac2 [lindex $mac_list 1]
    set mac3 [lindex $mac_list 2]
    set mac4 [lindex $mac_list 3]
    set mac5 [lindex $mac_list 4]
    set mac6 [lindex $mac_list 5]

    #分解步长
    set step_list [split $step "."]
    set step1 [lindex $step_list 0]
    set step2 [lindex $step_list 1]
    set step3 [lindex $step_list 2]
    set step4 [lindex $step_list 3]
    set step5 [lindex $step_list 4]
    set step6 [lindex $step_list 5]

    set c_flag 0 ;#设置进位标志=0
    set sum6 [expr $mac6 + $step6 + $c_flag]
    if { $sum6 > 255 } {
        set c_flag 1 ;#值>255 需要进位
        set sum6 [expr fmod($sum6,256) ]
        set sum6 [expr int($sum6)]
    } else {
        set c_flag 0
    }
}

```

```
}

set sum5 [expr $mac5 + $step5 + $c_flag]
if { $sum5 > 255 } {
    set c_flag 1
    set sum5 [expr fmod($sum5,256) ]
    set sum5 [expr int($sum5)]
} else {
    set c_flag 0
}

set sum4 [expr $mac4 + $step4 + $c_flag]
if { $sum4 > 255 } {
    set c_flag 1
    set sum4 [expr fmod($sum4,256) ]
    set sum4 [expr int($sum4)]
} else {
    set c_flag 0
}

set sum3 [expr $mac3 + $step3 + $c_flag]
if { $sum3 > 255 } {
    set c_flag 1
    set sum3 [expr fmod($sum3,256) ]
    set sum3 [expr int($sum3)]
} else {
    set c_flag 0
}

set sum2 [expr $mac2 + $step2 + $c_flag]
if { $sum2 > 255 } {
    set c_flag 1
    set sum2 [expr fmod($sum2,256) ]
    set sum2 [expr int($sum2)] } else {
    set c_flag 0
}

set sum1 [expr $mac1 + $step1 + $c_flag]
if { $sum1 > 255 } {
    set c_flag 1
    set sum1 [expr fmod($sum1,256) ]
    set sum1 [expr int($sum1)]
} else {
    set c_flag 0
}

set sum $sum1.$sum2.$sum3.$sum4.$sum5.$sum6 ;#组合返回结果
return $sum

}
```

5.12 foreach 控制结构

foreach 命令/控制结构会遍历整个列表，逐次取出列表的每个元素的值放到指定变量中，使用者可以在跟随的过程体中添加必要的处理过程。关于控制结构将会在控制结构一章有详细的介绍。

下例 5-14 直接使用例 5-10 得到的列表 l1 作为处理对象说明 foreach 的作用。本例将每个列表元素依次打印出来：

例 5-14 foreach 的简单例子：依次打印列表元素

```
%foreach elem $l1 {  
  puts "---$elem---"  
}  
=> ---This---  
    ---is---  
    ---one---  
    ---list---
```

本例 foreach 在每次循环进行时，依次将列表 l1 中的一个元素值赋值给 elem 变量。

foreach 命令还可以同时对多个列表进行操作，而且还可以同时操作同一列表的多个元素。如果在最后一次循环之前就遍历完了某个列表，则与之对应的循环变量就会以空字符串来赋值。

例 5-15 具有多个值列表的 foreach 命令处理过程

```
foreach {x1 x2} {Orange Blue Red Green Black} x3 {Right Left Up Down} {  
  puts [format "x1=%8s x2=%8s x3=%8s" $x1 $x2 $x3]  
}  
=> x1= Orange x2=   Blue x3=   Right  
    x1=   Red x2=  Green x3=   Left  
    x1= Black x2=      x3=   Up  
    x1=      x2=      x3=  Down
```

第 6 章. 数组

Tcl 中的数组和其他高级语言的数组有些不同：Tcl 数组元素的索引，或称键值，可以是任意的字符串，而且其本身没有所谓多维数组的概念。数组的存取速度要比列表有优势，数组在内部使用散列表来存储，每个元素存取开销几乎相同，而列表的存取数据花费时间与其长度成正比。

数组比列表提供了更灵活的数据处理方法，其功能比较类似于 C 语言的结构。本章将对数组操作的一些命令进行探讨。

6.1 数组的定义与格式

数组索引是由圆括号 () 来指定的，每个数组元素变量名的格式是“数组名(索引值)”。数组元素使用 set 命令来定义和赋值：

[语法]: set arrName(index) value

也可以用 array 命令来定义一个数组：

[语法]: array set arrName { index1 value1 index2 value2 ... }

这个命令在定义数组的同时可以定义其元素和元素值。需要注意元素索引(index-n)与元素值(value-n)要成对输入，否则会出错。用命令 array set arrName "" 可以定义一个空数组。用普通变量值的获取方法——替换操作来获取数组元素值：

[语法]: set val \$arrName(index)

数组元素索引也支持替换操作，包括变量和命令替换，如：

**[语法]: set val \$arrName(\$index)
set val \$arrName([expr \$index + 1])**

Tcl 数组索引的值不象其他高级语言如 C 语言那样，要求一定是整数。Tcl 允许索引值为包括数字字符在内的所有合法字符组成的字符串。

例 6-1 认识数组

```
% array set arr1 ""           ;# 定义了一个空数组
% set array01(5) "Hello World"
=> Hello World
% puts $array01(5)
=>Hello World
% set array01(Hello) World
=> World
% puts $array01(Hello)
=> World
% array names array01         ;# array names 命令显示数组元素名
=> Hello 5
% array set arr2 {1 a 2 b 3 c 4 d}
% array names arr2
=> 4 1 2 3
% parray arr2                 ;# 输出数组全部内容
=> arr2(1) = a
    arr2(2) = b
```

```
arr2(3) = c
arr2(4) = d
```

可以使用 `unset` 命令来取消一个数组变量定义。

6.2 数组变量

可以象使用普通变量一样来使用数组变量元素。如使用 `info exist` 来检测它是否存在，使用 `incr` 来递增它的值，使用 `lappend` 列表操作来追加列表元素等。

```
%set arr(1) 10
%incr arr(1)
=>11
```

将一个已经定义的普通变量当作数组变量使用或者将已定义的数组变量当作普通变量赋值是错误的：

例 6-2 混淆普通变量和数组时的错误例子

```
% set arr 10                                ;#定义了一个普通变量
=>10
% set arr(a) 5                               ;#试图将普通变量当作数组使用时出错
=> can't set "arr(a)": variable isn't array
% unset arr                                  ;#取消变量定义
% set arr(a) 5                               ;#ok
=> 5
% set arr 10                                 ;#试图赋值给数组变量时出错
=> can't set "arr": variable is array
```

可以使用替换获得数组名，如：

例 6-3 通过替换间接使用数组变量

```
%set name Arr
=> Arr
%set ${name}(1) abc
=> abc
% puts $Arr(1)
=> abc
```

6.3 多维数组

在有些时候，可能需要象 C 语言这样：

```
int arr[2][2]
arr[0][0] = 100
```

来定义一个多维数组来处理数据。Tcl 并没有直接支持这种数组的格式，使用者自己可以定义所谓的多维数组，如：

```
% set arr(0,0) 100
% set arr(0,1) 200
% parray arr
=> arr(0,0) = 100
```

```
arr(0,1) = 200
```

由于 Tcl 数组索引的灵活性，使用时要小心，否则可能得不到预期的结果，如忘记了上面索引的逗号，就成了：

```
% set arr(00) 100
% parray arr
=> arr(0,0) = 100
    arr(0,1) = 200
    arr(00) = 100
```

6.4 数组操作命令

就象字符串和列表一样，数组也有一套专门的操作命令。表 6-1 给出了这些命令的语法和说明。

表 6-1 数组操作命令表

命令格式	说明
array exists arr	判断 arr 是否为数组变量，是返回 1
array get arr ?pattern?	返回一个包含交替出现索引、元素值的列表。pattern 选择匹配索引。如果不指定 pattern，返回所有的元素索引和值。
array names arr ?pattern?	返回索引
array set arr list	初始化数组
array size arr	数组大小
array startsearch arr	返回用于 arr 进行搜索的搜索标记
array nextelement arr index	返回下一个元素值，如果已在尾部的话，返回空串
array donesearch arr index	结束有 index 标识的搜索
parray arr	打印出 arr 的所有元素变量名和元素值

6.4.1 array get 命令

array get 命令提取数组索引、元素值对并将这些值对组织成一个列表。而 array set 命令则将一个列表（数据要成对）转换成一个数组。

例 6-4 数组和列表互换

```
% array set arr [list a AAA b BBB c CCC d DDD]
% array size arr           ;#数组元素个数
=> 4
% parray arr
=> arr(a) = AAA
    arr(b) = BBB
    arr(c) = CCC
    arr(d) = DDD
% set ll [array get arr]
=> d DDD a AAA b BBB c CCC
```

6.4.2 array names 命令

array names 返回所有元素索引名与模式 pattern 匹配的元素索引名列表。模式 pattern 和 string match 的模式格式相同。如果 pattern 没有指定，则返回所有数组元素索引名列表。

例 6-5 array names 的简单例子

```
%array set a [list "School,BUPT" "BUPT" "School,NJU" "NJU" "School,NJUA" "NJUA"]
% parray a
=>
    a(School,BUPT) = BUPT
    a(School,NJU)  = NJU
    a(School,NJUA) = NJUA
% array names a "School,*"
=>School,NJU School,NJUA School,BUPT
% array names a "School,N*"
=>School,NJU School,NJUA
% array names a
=>School,NJU School,NJUA School,BUPT
```

6.4.3 遍历数组

如何对未知数组的每个元素进行操作呢，这可有多种方法。下面的例子提供了一个途径：

例 6-6 遍历数组的一个方法举例

```
% array set a1 [list a AAA b BBB c CCC d DDD]
% set l1 [array names a1]
=> d a b c
% foreach id $l1 {
    puts "a1($id) = $a1($id)"
}
=> a1(d) = DDD
    a1(a) = AAA
    a1(b) = BBB
    a1(c) = CCC
```

注意：array names 返回的元素索引的时候，最后一个元素的索引被放置到了列表的第一个位置上。

6.4.4 用数组定义结构

Tcl 语言中没有结构类型，可以通过数组定义来实现相似的功能。

例 6-7 用数组来定义结构

```
% array set Struct_N [list -S_name "" -S_type "" -S_value ""]
% parray Struct_N
=> Struct_N(-S_name) =
    Struct_N(-S_type) =
    Struct_N(-S_value) =
```

第 7 章. 控制结构命令

控制结构允许程序根据不同的状态、条件和参数来选择不同的处理和执行路径，从而使代码具有更强的灵活性、健壮性和可读性。

Tcl 提供了 **if**、**if/else**、**if/elseif**、**foreach**、**for**、**while** 和 **switch** 命令来管理控制结构。这些命令和其他语言如 C 语言的条件语句的作用相同。需要区别的是在 Tcl 中所有控制结构都是由相应的**命令**来实现，而 C 语言中则是一条控制**语句**。

本章节除了主要讲述上述各种控制命令的使用方法，还讲述了在程序中如何通过 **catch** 命令来捕获命令执行状态、如何使用 **error** 命令来处理错误报告和如何用 **return** 语句从一个过程中返回。

控制结构通常要求带有一个延迟执行命令体或者过程体，这个命令体需要用花括号括起来以加以界定。

foreach 结构在列表一章中已经介绍过。

7.1 if/else 命令

if 命令根据表达式的结果来执行命令体：如果表达式结果为真，则执行命令体，否则会执行另外一个条件命令体（如果存在的话）。后面两个命令体(**elseif** 和 **else**)是可选的。

```
[语法]  if {test expr 测试表达式} {
            body 1
        } elseif {test expr} {
            body2
        } else {test expr} {
            body3
        }
```

说明：

1. 语法中用以界定过程体的花括号一定要和 **if** 命令在同一行上！因为对 Tcl 来讲，换行符就是命令结束符，所以如果在 **if** 表达式后直接换行，写成：

```
if {test expr}
{
...
}
```

就会出错。Tcl 遇到换行后就认为命令结束，但找不到执行命令体，返回错误。其他的控制命令，还有以后的过程定义命令等等都存在这个问题。

但情况并不全部如此。当在一个花括号体内或者一个双引号体内换行的时候，解释器不认为是命令的结束，所以上面的语法中，我们只将执行命令体的第一个花括号（左括号）留在了 **if** 命令行和 **else** 命令行，然后另起一行书写执行命令体的过程语句，右括号也被单独放到了一行上。这样做是为了提高可读性和便于查错。

2. 如果 **if** 后面还有 **else/elseif** 命令，则留意 **else/elseif** 的位置。**else/elseif** 要跟在 **if** 执行命令体的后面一个花括号后，不能分行，要有空格间隔花括号和 **else/elseif**。
3. 花括号括起的表达式、执行命令体或者其他内容相当于变量存在，所以前后与其他命令元素之前要有空格，否则 Tcl 会返回语法错误。
4. 可以使用多个 **elseif** 来创建一连串的条件命令控制结构。
5. 表达式支持变量替换和命令替换。
6. 表达式的计算结果如果是 **"true"**、**"yes"** 和非零值就判断为真，如果结果是 **"false"**、**"no"** 和零则判断为假。控制命令根据表达式结果来判断是否执行相应的执行命令体。

例 7-1 if/else 控制命令的简单例子

```
%set x hello
```

```
% if {[string compare $x hell]} {
    puts "String is hell"
} elseif ![string compare $x hel] {
    puts "String is hel"
} elseif ![string compare $x hello] {
    puts "String is hello"
} else { puts "Error input string!"}
=>String is hello
```

例子中 if 的表达式用花括号括了起来，而 elseif 的表达式却没有花括号，这两种表达方法都可以，但用花括号的时候条件命令语句执行得更有效率。

7.2 for 命令

for 命令和 C 语言的 for 语句相似。for 命令的语法格式为：

```
[ 语法 ] : for {start} {test expr} {next} {
                body
            }
```

for 命令有四个变元，**start** 是预置条件或者初始化命令，告诉 for 命令起始执行条件。**test expr** 是条件布尔表达式，以决定是否执行循环体 **body**，如果是真，则执行循环体，如果假则退出命令。如果表达式真，则在执行循环体后处理 **next** 命令，即 **next** 是一个后置命令执行体。

前三个变元可以选择置空，而将相应的处理放到循环体 body 中去。

例 7-2 一个 for 循环

```
%for { set i 0 } { $i < 10 } { incr i 2 } {
    if { $i == 4 } {
        continue ;#如果是 4，则不打印
    }

    puts "i = $i"

    if { $i >= 6 } {
        break
    }
}
=> i = 0
    i = 2
    i = 6
```

break 命令导致立刻从循环体内退出。而 continue 在 i 值为 4 的时候忽略后面的循环体的内容而执行下一个循环。

7.3 while 命令

while 命令格式为：

```
[ 语法 ] : while {test} {
                body
            }
```

`while` 命令和 `for` 命令非常相似。只要 **test** 为真，`while` 就执行循环体直到 **test** 变为假。`for` 命令和 `while` 命令的主要区别是，在 `while` 循环体内你必须更改被检测的测试体 **test** 的值，否则如果值一直没有改变成假时，`while` 将无限的执行循环体。而 `for` 命令你可以将这种处理过程在 **next** 变元中显式给出。

例 7-3 `while` 循环的例子

```
% set i 3
% while {$i > 0} {
    puts "Current index is $i."
    incr i -1
}
=> Current index is 3.
    Current index is 2.
    Current index is 1.
```

7.4 break 与 continue 命令

`break` 命令立即终止循环并退出循环体。而 `continue` 命令则会忽略后面的循环体内容而继续下一个循环处理过程。Tcl 中没有 `goto` 命令。`break` 和 `continue` 的用法见例 7-2。

7.5 switch 命令

`switch` 命令通过将给定字符串与不同的匹配模式进行匹配从而选择执行多分支命令体。`switch` 可基于模式匹配。命令格式为：

```
[语法]: switch [option] string {
    pattern-1 {body1}
    pattern-2 {body2}
    ...
    pattern-n {bodyn}
}
```

说明：

1. **option** 主要有：
 - exact 用精确匹配（默认）；
 - glob 用 glob 格式行模式匹配；
 - regexp 用 正则表达式模式匹配；
 - 标记选项结束或者说明不用选项。
2. 如果项邻的两个或者多个 `pattern-x` 的执行命令体是一样的，则可以只写出最后的一个执行命令体，而前面的执行命令体可以省略，并用“-”号来替代。
3. 最后一个 **option** 一定是 "--"，这个选项不可缺少！
4. 可以使用 `default` 匹配命令体来处理无法匹配模式。当其他模式都不匹配时，`default` 命令体就会被执行。需要注意，`default` 命令体要设置在最后，否则它就会被当作纯粹的字符串 `default` 而进行匹配。
5. 对于 `switch` 执行命令体内的注释一定要小心。Tcl 语法器处理的注释应该和命令处于同一层次，即一个注释要占用一个命令的位置。这样就限制了在 `switch` 体内注释的位置。比如你不能将一条注释放在和 `pattern-n` 同一级别的位置，那样 `switch` 命令就会将此条注释也当成一个匹配模式来解释，这有可能引起意想不到的错误。所以，如果打算在 `switch` 体内写注释的话，最好将注释放在相应的某个匹配模式的命令体 **body-n** 内。

例 7-4 `switch` 命令的简单例子

```
% set result TRUE
% switch -exact -- $result {
    # Comment 1: This comment will confuses switch command
```

```

"TRUE" {
    # Comment2: This comment is ok
    puts "TRUE"
}
"FALSE" {
    puts "FALSE"
}
"UNKNOWN" -
default { puts "UNKNOWN or unkown value"}
}
=>TRUE

```

“UNKNOWN 和 default 处理过程相同。上面的第一个注释语句（Comment1）被 switch 当成一个匹配项，匹配内容是“#”，而随后的注释语句会被当成命令体。switch 命令执行过程为：首先将输入字符“TRUE”和第一个匹配项“#”进行精确匹配，发现两者不同，然后与第二个匹配项“TRUE”匹配，两者相同，执行相应的命令过程体并结束 switch 命令。而象下面的例子，尽管 switch 语句和上例一样，但结果就没有那么庆幸了。

例 7-5 switch 语句中不当注释引起的错误

```

% set result #
% switch -exact -- $result {
    # Comment1: This comment will confuses switch command
    "TRUE" {
        # Comment2: This comment is ok
        puts "TRUE"
    }
    "FALSE" {
        puts "FALSE"
    }
    "UNKNOWN" -
    default { puts "UNKNOWN or unkown value"}
}
=> invalid command name "Comment1:."

```

7.6 catch 命令

在实际应用中，如果一条命令或函数在调用时使用了错误输入参数或者自身出现错误，它就会报错，而且终止当前脚本的执行。严重的错误也可能导致 Tcl shell 退出。catch 命令就可以用来捕获这种调用错误。

[语法]: catch {com_proc} ?res?

catch 命令有两个变元，**com_proc** 是命令体。**res** 用来保存命令返回结果，或是出错时的错误信息，此变元为可选项。如果有错，catch 会返回 1，无错就返回 0。命令体需要用花括号括起来。

例 7-6 用 catch 捕获命令错误信息

```

% set status [catch { puts "The value of y is $y"} res ]
=>1                ;# 因为用了未定义的变量 y 而出错
% if {$status} {
    puts "Command faild. Error Info: $res"
}

```

```
}
=>Command faild. Error Info: can't read "y": no such variable

% proc test {} {
    return OK
}
% set status [catch {test} res]
=> 0
% puts $res ;#catch 命令还可以捕获函数返回值
=> OK
```

注意使用 catch 命令来捕获命令错误并根据执行结果做进一步的处理!

7.7 error 命令

error 命令报告错误信息并终止脚本执行。

[语法]: `error message_string ?info? ?error_code?`

message_string 是错误信息字符串, info 变元用于初始化全局变量 `errorInfo`, 如果 info 没有提供, 则 error 自身初始化 `errorInfo`。变元 code 指定了一个机器可读的错误信息, 会被存储在全局变量 `errorCode` 中, 默认为 `NONE`。

例 7-7 error 命令的例子

```
#定义一个函数 foo, 返回一条错误:
%proc foo {} {
    error "1.Function foo report error" "2.Some error in function foo" 20
}
% foo                                ;#运行 foo 函数
=> 1.Function foo report error
% catch {foo} str
=>1                                ;#用 catch 捕获错误信息, error 输出的错误信息被保存在了 str
中
% puts $res
=> 1.Function foo report error
% puts $errorInfo                    ;#显示 errorInfo 内的内容
=> Some error in function foo
    (procedure "foo" line 1)
    invoked from within
    "foo"                                ;#errorInfo 中给出了详细的出错报告, 每条执行错误的
                                        ;#命令都会引起 errorInfo 的内容被改写
% puts $errorCode ;#显示 errorCode 的值, 为 error 报告的 code 值
=> 20

% puts $a                            ;#执行一条错误命令: 试图输出一个没有定义变量的值
=> can't read "a": no such variable ;#返回的错误信息
% puts $errorInfo
=>can't read "a": no such variable
    while executing
    "puts $a"
% p uts $errorCode
=> NONE
```

7.8 return 命令

在一个过程中，可以使用 `return` 来返回调用。`return` 的位置可以根据各种条件和需要进行安排，而且一个过程中可以包含多条 `return` 命令，但当遇到第一个可执行的 `return` 命令时就返回。

通常使用的 `return` 命令，或者不带变元，或者仅仅返回一个参数值。其实 `return` 命令的可选变元还有许多：

[语法]: `return ?-code cd? ?-errorinfo info? ?-errorcode errc? str`

（注：两个问号括起的变元为可选项。）

`-code` 的选项值是 `ok`、`error`、`return`、`break`、`continue`，也可以是一个整数。默认为 `ok`。`-code error` 选项使 `return` 命令功能和 `error` 非常相似。此时 `-errorcode` 选项设置全局变量 `errorCode`，而 `-errorinfo` 选项为 `errorInfo` 提供辅助信息。

例 7-8 用 `return` 命令从过程中返回

```
proc stub {} {
    return -code error -errorinfo "Function stub report error" -errorcode 30 "Some error in
function stub"
}
% stub
=>Some error in function stub
% puts $errorInfo
=>Function stub report error
    invoked from within
    "stub"
% set errorCode
=>30
```

7.9 exit 命令

`exit` 命令用来终止脚本的执行。`exit` 会终止并退出整个运行脚本的进程（退出 Tcl shell），使用时要小心。如果在退时提供了一个整数数值，则它代表退出状态。

第 8 章. 过程与作用域

一个过程创建好后, 就可以象 Tcl 内建命令一样直接使用。

过程可以有自己的内部变量, 而且起作用的范围仅限过程内部, 过程外部无法使用、获取这些变量的值。这就引出了作用域、局部变量和全局变量的概念。

Tcl8.0 以后的版本增加了名字空间(name space), 它为过程和变量提供了新的作用域。名字空间会在相关章节会介绍。本章将介绍过程和作用域的相关概念。

8.1 proc—过程定义命令

使用 proc 命令定义过程:

```
[语法]: proc procName {var1 var2 ...}{
    body
}
```

说明:

1. proc 命令有三个参数: procName 是定义的过程名字; {var1 var2 ...}是输入、输出参数列表; body 是过程执行命令体。body 的界定大括弧和 if 等命令的命令执行体遵循相同的规则和注意事项。
2. 可以使用 return 命令在需要的时候返回调用程序。
3. 使用过程的时候, 不一定输入所有的参数值。过程的输入参数可以有默认值。默认值由 {默认参数名 默认值}指定。如果调用过程时没有指定这些参数的值则会使用其默认值, 否则使用输入值来替代默认值。在使用默认参数的时候要注意, 如果默认参数之后还有非默认参数, 则在调用此过程的时候, 默认参数的值也要求输入, 否则会出错。这是因为 Tcl 调用过程的时候是根据位置来匹配参数和输入值的。

例 8-1 定义了一个带有默认参数的过程, 过程输出字符串信息, 并返回计算结果。

例 8-1 带有默认参数的过程定义

```
% proc Test { a {b 7} {str "Hello world"}} {
    puts "$str"
    return [expr $a * $b ]
}
% Test 10                                ;#只输入非默认参数 a 的值
=>Hello world
    70
% Test 10 5 "Call Test"                  ;#输入值替代默认值
=> Call Test
    50

%proc Test2 { {a 10} b} {                ;#默认参数后还有非默认参数
    puts "a=$a b=$b"
}
% Test2                                  ;#不提供输入
=> no value given for parameter "b" to "Test2"
% Test 2 5                                ;#没有输入非默认参数值
=> no value given for parameter "b" to "Test2"
% Test2 5 10                              ;#ok
=> a=5 b=10
```

如果参数列表中最后一个参数是 args 的话, 则过程可以接收可变数目的输入参数。当调用过程时, 除了指定参数以外的参数值都被 args 接收。如果参数列表中只有 args 一项, 则 args 接收所有输入参数值。

例 8-2 不定输入参数过程的例子

```
%proc vParam {args} {                                ;#定义一个只接收可变数目参数的过程。
    puts "input values are : $args"
}

% vParam                                                ;#没有给定任何参数值时
=> input values are :
% vParam 1 "Hello" 2 3 "End"                          ;#给定一组任意值
=> input values are : 1 Hello 2 3 End
```

一般在调用过程的时候，我们总希望能清楚输入参数值与哪个参数对应，希望能象下面的调用格式：

```
vParam -date "2003-11-21" -time "21:03:45" -person "John" -place "Yu Park"
```

这里清楚的表明，参数-date 的值为 2003-11-21，参数-time 的值为 21:03:45 等等。这样输入参数就不容易出错，而且可读性也好。

在过程内部，可以将输入参数名+值的列表转换成数组，则每个参数名就成了数组元素的所以，对应的参数值成了对应数组元素的值。这样对接下来的处理过程就比较容易使用。

例 8-3 参数名+参数值成对输入的过程定义

```
% proc config_sys {args} {
    array set inArr $args
    parray inArr
}
%config_sys -sysName "HUB100" -ipAddr 192.168.10.1 -date 2003-11-21 -time 21:03:45
=> inArr(-date) = 2003-11-21
    inArr(-ipAddr) = 192.168.10.1
    inArr(-sysName) = HUB100
    inArr(-time) = 21:03:45
```

用户定义的过程可以保存在单独的一个 Tcl 文件中，然后用 source 命令将此过程调用到脚本中。

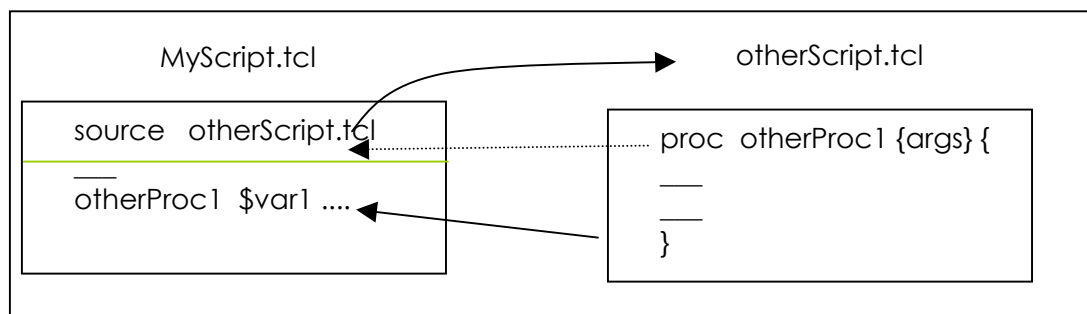


图 5 使用在另外一个 Tcl 文件中的过程

8.2 作用域

8.2.1 过程的作用域

默认情况下，过程名具有单一的作用域即全局作用域。所以可以在脚本的任何地方使用一个过程（使用之前，过程必须已经定义）。在后面的名字空间一章中，大家就会了解到，可以通过名字空间来限定过程的作用范围。如可以通过名字空间命令设置一个过程可以让外部调用，也可以设置过程不为外部所使用。这和 C++ 中的类(class)的作用比较相似。

过程的定义可以嵌套，低层定义的过程只有在上层过程被执行后才能生效。下面的例子是在一个过程 one 中定义了另外一个过程 two,看 two 是如何使用的。

例 8-4 过程的定义

```
% proc one {} {
    puts "I'm one"
    proc two {} {
        puts "I'm two"
    }

    puts "Call two in one:"
    two
}

% two
=>invalid command name "two"

% one
=>I'm one
    Call two in one:
    I'm two
% two
=>I'm two
```

8.2.2 变量的作用域

对于变量而言，根据其作用域可分为全局变量和局部变量。在所有过程之外定义的变量为外部变量，即全局变量，它的作用域为从开始定义到执行结束，除非中间有显式取消其定义。在一个过程体内定义的变量为内部变量，即局部变量，局部变量的作用域只限于过程内部使用，在此过程外面不能使用这些变量。在一个过程内部，即可以使用自身的局部变量，又可以使用全局变量，但是全局变量在过程内部不会自动可见，需要通过 global 命令来事先声明。因为作用域不同，所以过程中的变量可以与全局变量、其他过程中的变量有相同的名字。

设置必要的全局变量可以为过程之间提供数据联系的一个渠道。因为全局变量可以供所有过程使用，所以如果在一个过程中改变了全局变量的值，就能影响到其他过程。

在过程定义中的输入、输出参数列表中的参数为过程的内部参数。

例 8-5 变量的作用域

```
%set a 5
% set b -8
% set c 10
% proc P1 {a} {
    set b 42
    global c
    puts "Value of Input parameter a is $a"
```

```

    puts "My b is $b"
    puts "Global c is $c"
}
%P1 $b
=> Value of Input parameter a is -8
    My b is 42
    Global c is 10

```

global 命令使 全局变量 c 对过程可见。全局变量的定义不一定要在过程外完成，可以在任何一个过程中完成，其效果是将一个局部变量的作用域进行了扩展。但是在引用全局参数的值之前全局参数应首先已被赋值。另外当一个过程中要使用与局部变量相同名字的参数时，用 global 命令就要小心：你不能在定义完成一个局部变量后再用 global 命令使用同名的全局参数，这会出错。

例 8-6 全局变量与局部变量的关系

```

% proc P2 {} {
    global var1
    set var1 100
    puts "var1 = $var1"      ;#在 P2 内定义全局变量
}
% proc P3 {} {
    global var1
    puts "var1 = $var1"
}
% proc P4 {} {
    global var2
    puts $var2
}

% proc P5 {} {
    set a 30
    global a
    puts "a = $a"
}
% P2
=>var1 = 100
% P3\
=>var1 = 100
% P4                      ;#试图使用不存在的全局变量
=>can't read "var2": no such variable
% P5                      ;#在定义局部变量后试图引用同名全局变量
=>variable "a" already exists

```

还有另外一种方式来直接使用全局变量即用双冒号 (::)，见下例。

例 8-7 用 "::" 来声明全局变量

```

proc test {} {
    puts "The value of global variable var1 is $::var1"
}

% set var1 20
=>20

```

```
%test
=> The value of global variable var1 is 20
```

两种使用全局变量的方法都可以，但后种方式可以避免过程中局部变量和所要使用的全局变量同名的问题，也可以使代码更清晰。

8.3 upvar 命令

upvar 命令的作用与 global 的作用相似，upvar 通过“引用”来使用上层过程中的变量，它传递的是参数名而非值。

[语法]: `upvar ?level? otherVar1 myVar1 ?otherVar2 myVar2 ... ?`

upvar 命令将 myVar1 定义为 otherVar1 的一个引用 (reference)，otherVar 是有 level 指定的本过程调用栈中的向上 level 层的变量。当定义好之后，本过程就可以通过 myVar 参数来使用 otherVar 参数。

level 有两种表示方式。当 level 为一个整数值时，表示从当前作用域向全局作用域上溯到 level 层作用域。比如 level 为 1 时，代表调用过程作用域，level 为 2 时为上一层作用域。如果 level 是一个“#”跟一个整数，则表示从全局作用域向当前作用域下溯。比如 level 为 #0，则表示全局作用域，此时的作用同 global 命令。

一个过程 A 或可以被其他过程 B 所调用，而过程 B 也可被过程 C 调用，那么 A、B、C 三个过程组成了一个调用层次，A 为最低，C 为较高层 (global 全局过程为最高)。则 A 就可以通过 upvar 命令来引用过程 B、C 和全局过程的变量，而不象 global 命令只能使用全局变量。如果 level 为 1 (默认)，则表示引用相邻上一层的参数 (过程 B)，如果为 2，则表示引用的是向上第 2 层 (过程 C) 的参数。

还有一点，上面介绍 global 的时候提到，如果过程已经用 set 命令定义了一个参数，此时用 global 命令使用全局变量中相同名字的参数就会出错，报告参数已经存在。而通过 upvar 命令却可以作到这一点，upvar 命令通过一个不同的参数名来引用全局变量。

例 8-8 upvar 命令的例子

```

;# 步骤一、先编写一个.tcl 文本，保存代码 (upvar.tcl) :
;# -----保存程序代码开始-----
;# 本层是 global 层
;# (1). 一个简单的 upvar 的例子
;# 函数 SetPositive 将一个负数转换为对应整数.
proc SetPositive {varName varValue} {
    upvar $varName myvar           ;# 为输入参数名定义引用为 myvar
    if {$varValue < 0} { set myvar [expr -$varValue]; } else { set myvar $varValue; }
    return $myvar;
}

set x 5
set y -5
puts "Before call SetPositive:"
puts "X : $x   Y : $y\n"

SetPositive x $x                   ;# 调用转换函数处理变量 x
SetPositive y $y                   ;# 调用转换函数处理变量 y
puts "After call SetPositive:"
puts "X : $x   Y : $y\n"

```

;(2). 嵌套 upvar 命令的例子

过程 Second 将被用作第二层过程, 供第一层过程 First 调用

```
proc Second {varName} {  
    upvar 1 $varName z      ;# 将输入参数 varName 的值和变量 z 联系起来。  
                            ;# varName 的值应该是存在的上层变量名  
    upvar 2 x a             ;# 将向上两层的过程的参数 x( global x)与本过程参数 a  
                            ;#联系起来  
    puts "In Second: Z: $z A: $a";# Output the values, just to confirm  
    set z 1;                ;# 赋值给 z, 本例其实是赋值给向上一层的$varName;  
    set a 2;                ;# 赋值给 a, 本例其实是赋值给向上两层的参数 x  
}
```

第一层过程 First, 供 global 过程调用, 并调用第二层过程 Second.

```
proc First {varName} {  
    upvar $varName z      ;# 将输入参数 varName 的值和变量 z 联系起来。  
    puts "In First: Z: $z" ;# Output that value, to check it is 5  
    Second z;             ;# 调用过程 Second,来改变全局变量 x 和 y 的值  
}
```

```
First y      ;# Call First to perform nesting upvars, and output X and Y after the call.  
              ;# Function First will output value of y through reference z,  
              ;# Then First will call Second to change value of global variable y and x.  
puts "\nAfter call First: \nX : $x Y : $y\n"
```

;(3). 要使用的全局变量名和本过程的局部变量名相同的例子

```
proc SameName {} {  
    set x 20  
    upvar #0 x global_x  
    puts "In SameName:"  
    puts "My x is : $x"  
    puts "Global x is: $global_x"  
}
```

SameName ;#运行

##-----保存程序代码结束-----

步骤二、运行 tclsh 用 source 命令执行 upvar.tcl:

% source upvar.tcl

=> Before call SetPositive:

X : 5 Y : -5

After call SetPositive:

X : 5 Y : 5

In First: Z: 5

In Second: Z: 5 A: 5

After call First

X: 2 Y: 1

In SameName:

My x is : 20

Global x is: 2

如何理解 upvar 和引用的作用呢？可以这么认为，upvar 命令给指定的上层参数定义了一个别名变量，而别名变量和原上层参数变量指向同一内存空间。这和 C 语言中通过引用将多个指针指向同一内存空间相似。

通过 upvar 命令可以很容易的将一个数组传递给过程。

例 8-9 通过 upvar 命令来传递数组

```
proc operarr {arrName} {  
    upvar $arrName outArr  
    puts "The number of elements in array $arrName is: [array size outArr]"  
    puts "elements in array $arrName are:"  
    parray outArr  
  
    set outArr(-result) PASS  
    parray outArr  
}  
  
% array set arr [list --date 2003-11-23 -time 10:20:45 -place "Alcatel-sbell company"]  
% operarr arr  
=>The number of elements in array arr is: 3  
    elements in array arr are:  
    outArr(-date) = 2003-11-23  
    outArr(-place) = Alcatel-sbell company  
    outArr(-time) = 10:20:45  
% parray arr  
=> arr(-date) = 2003-11-23  
    arr(-place) = Alcatel-sbell company  
    arr(-result) = PASS ;#回传参数  
    arr(-time) = 10:20:45
```

8.4 rename 命令

rename 命令可以用来更改命令名，这些命令包括 TCL 自带的内建命令和读者自己定义的过程。

[语法]: `rename oldFuncName newFuncName`

如有一个命令名为 old，则通过命令

```
%rename old new
```

就将命令名改为了 new。如果以后要调用原来的命令就不能用 old，只能用 new。

如果新命令名 newFuncName 是空字符串 {}，则 rename 命令此时的作用是取消一个命令。

例 8-10 用 rename 命令来取消一个命令

```
% proc old {} { ;#定义一个过程  
    puts "This is old function."  
}  
% old
```

```

=>This is old function.
% rename old new
% old                      ;#试图再次调用 old 命令失效
=>invalid command name "old"
% new
=> This is old function.
% rename new {}            ;#通过 rename 命令取消 new 命令
% new                      ;#new 命令已经不存在，调用失效
invalid command name "new"

```

8.5 特殊变量

当 Tcl shell 被调用的时候，tclsh 或 wish 会设定一些变量，如命令行参数和环境变量数组。在编程时，可以使用命令行参数和环境变量数组传递参数信息。

8.5.1 命令行参数

命令行参数是在 Tcl shell 被调用时定义/初始化的。命令行参数有：

- argc 命令行参数的数目，不包括执行脚本的名字
- argv0 脚本名
- argv 命令行参数列表

读者可以通过下面的例子来认识命令行参数。本例中，先编写一个 Tcl 脚本文件 cmdline.tcl，然后调用 Tcl shell tclsh 来运行此脚本文件。第一次没有输入任何命令行参数，第二次输入了 5 个命令行参数。

例 8-11 打印命令行参数信息

```

;# STEP1: Create script:
;#please save follow script into cmdline.tcl
;# sample script : cmdline.tcl
puts "The number of command line arguments is: $argc"
puts "The name of the scriptis: $argv0"
puts "The command line arguments are: $argv"

;#STEP2: Call tclsh to run cmdline.tcl:
E:\> tclsh cmdline.tcl
=> The number of command line arguments is: 0
    The name of the scriptis: cmdline.tcl
    The command line arguments are:
E:\>tclsh cmdline.tcl a b c d e
=> The number of command line arguments is: 5
    The name of the scriptis: cmdline.tcl
    The command line arguments are: a b c d e

```

8.5.2 env--环境变量数组

TCL 提供了一事先就定义好的全局环境变量数组，这个数组叫做 env。读者可以通过环境变量数组元素的名字来使用各元素的值。下面的命令打印出 PATH 环境变量的内容：

```
%puts "$env(PATH)"
```

```
= > d:\software\Tcl\bin;D:\software\oracle\ora81\bin;C:\Program
Files\Oracle\jre\1.1.7\bin;C:\WINDOWS;C:\WINDOWS\system32;C:\WINDOWS\sys
tem32\WBEM;C:\PROGRA~1\COMMON~1\AUTODE~1;D:\software\TCLPRO1.4\wi
n32-
ix86\bin;d:\software\Rational\common;d:\software\Rational\ClearQuest;d:\soft
ware\Rational\ClearCase\bin;D:\software\TCLPRO1.3\win32-ix86\bin
```

用 `parray` 命令则可以显示所有环境变量和变量值:

```
%parray env
```

读者可以操作象一般的数组方式来操作环境变量数组。例如, 读者可以使用下面的命令加入一新的条目到 `PATH` 中:

```
%set env(PATH): "$env(PATH):/usr/sbin"
```

需要注意的是, 环境变量数组的任何修改都不影到原始行程 (例如, 读者启用 `TCL script` 的 `shell` 如 `unix` 的 `csh`, 就是 `TCL script` 原始行程) 的环境变量的内容 (当退出 `TCL shell` 之后, 这些更改也随之失效)。任何使用 `exec` 命令所建立出的 `script` 过程都继承修改後的所有环境变量。

8.6 eval 命令

一条被执行的 `Tcl` 命令被定义为一个字符串列表, 而这个列表的第一个字符串是一条 `Tcl` 内建命令或者过程。任何字符串或者列表, 只要符合这个标准, 都可以被计算和执行。而 `eval` 命令就可以完成计算和执行这种字符串或列表命令, 从而允许程序可以动态地构造命令。

[语法]: `eval var1 ?var2...varN?`

`eval`--Evaluate a `Tcl` script.

`eval` 命令按照 `concat` 风格将输入参数连接成命令字符串, 然后调用 `tcl_Eval` 来完成命令计算和执行。

下面的例子先创建一条符合上述标准的 `Tcl` 命令字符串或者列表, 然后调用 `eval` 命令对其进行计算并尝试执行此命令。如果成功, `eval` 命令就返回被执行命令的返回值, 如果命令字符串有错误, 则返回错误信息。注意用不同的命令组织形式可能导致不同的结果:

例 8-12 用 `eval` 创建新命令的简单例子

```
;(1) 将构造工作给 eval 完成
%eval puts "ok"
=>ok
```

```
;(2) 用花括号来构造命令
%set string "Hello World"
=>Hello World
%set cmd { puts stdout $string}
=> puts stdout $string
%eval $cmd
=> Hello World
```

;(2) 用花括号来组织命令来阻止替换

;(2) 调用 `eval` 来计算命令, `eval` 调用 `Tcl`
;(2) 解释器进行替换, 并执行命令

```
;(3) 用双引号构造命令导致出错
%set cmd "puts stdout $string"
```

;(3) 用双引号组织命令, 会导致替换提前发生

```
=> puts stdout Hello World
% eval $cmd
=> bad argument "World": should be "nonewline"
```

```
;/#(4)用 list 来构造命令
% set string "Hello World"
=>Hello world
% set cmd [list puts stdout $string]
=>puts stdout {Hello World}
%eval $cmd
=>Hello World
```

`eval` 命令完成两次替换：首先用 `cmd` 的值来替换 `$cmd`，然后在这个值上再次完成内部的替换，结果是一条 `puts` 命令被执行。

第三个 `eval` 命令出错，是因为在组织命令的时候使用了双引号，双引号不会阻止内部的替换操作，它破坏了列表的结构。解决这个问题的最好方法，是用 `list` 命令显式构造命令。

上面的例子给出了用 `eval` 和 `list` 来创建单一的命令。该命令一个强大的应用是动态创建一个过程。下面的例子中，在过程 `board` 中，通过 `eval` 命令动态地创建了一个过程 `board_{${t(board_instint)}}`，此过程名和其输入参数值取决于过程 `board` 的输入参数值。过程 `board` 的输入参数为：

```
-instance
-board_type
-s_rack_inst
-slot_pos
-lsm_pwr
-board_pres
```

例 8-13 动态定义过程

```
set l_boards [list]
proc board {args} {
    global l_boards
    set t(-lsm_pwr) UP
    set t(-board_pres) true
    array set t $args
    parray t

    set pname "board_${t(-board_inst)}"

    set cmd "proc $pname {} { return \"-instance ${t(-board_inst)} -board_type ${t(-board_type)} \
        -s_rack_inst ${t(-s_rack_inst)} -slot_pos ${t(-slot_pos)} -lsm_pwr ${t(-lsm_pwr)} -board_pres \
        ${t(-board_pres)}\" }"
    lappend l_boards ${t(-board_inst)}
    eval $cmd ;#嵌套定义了一个过程
}
```

;/#两次调用 `board` 过程，将分别创建两个新过程：`board_1` 和 `board_2`。

```
% board -board_inst 1 -board_type AACU-C -s_rack_inst 1 -slot_pos 1 -board_pres true
% board -board_inst 2 -board_type SANT-G -s_rack_inst 1 -slot_pos 1 -lsm_pwr UP -board_pres true
% puts $l_boards
=> 1 2
% board_1
=>-instance 1 -board_type AACU-C -s_rack_inst 1 -slot_pos 1 -lsm_pwr UP -board_pres true
% board_2
```

```
=>-instance 2 -board_type SANT-G -s_rack_inst 1 -slot_pos 1 -lsm_pwr UP -board_pres true
```

8.7 uplevel 命令

uplevel 命令和 eval 命令相似，不同的一点是，uplevel 命令不仅仅可以在当前的过程中计算 TCL 命令，而且可以在不同的作用域中计算 TCL 命令。举个简单的例子，如有一个过程需要在全局作用域下执行一条 TCL 命令，而这条命令使用的部分参数默认是全局变量。这时使用 "uplevel #0 command args" 在全局作用域下执行命令就很简单。

uplevel 命令的语法为：

[语法]: **uplevel** *?level?* *command* *?list1 list2 ...?*

level 的含义和 eval 命令相同，如果不指出的话则代表调用过程的作用域。command 为命令名，后跟参数列表。下面的例子中，过程 two 通过 uplevel 命令输出本地、上层调用过程和全局变量的值，给出的 uplevel 命令比较多，请仔细比较和体会 uplevel 的作用。有关 uplevel 命令的具体应用，请参考帮助文件。

例 8-14 uplevel 的简单例子

```
% set x "I'm global's x"
=>I'm global's x
% proc one {} {
    set x "I'm one's x"
    two
}
% proc two {} {
    set x "I'm two's x"

    uplevel {puts $x} ;#默认为调用者作用域

    uplevel [list puts $x] ;通过 list 命令

    uplevel 2 {puts $x}
    uplevel #0 {puts $x}

}
% one
=>I'm one's x
I'm two's x
I'm global's x
I'm global's x
% two
=>I'm global's x
I'm two's x
I'm global's x
bad level "2"
```

第 9 章. 正则表达式 (Regular Expressions)

为了与国内其他文献中称谓一致,从 1.2 版本起,使用“正则表达式”代替以前版本中的“正规表达式”。正规表达式是台湾地区的翻译版本。

使用 Tcl 离不开对字符串的操作,如字符串的匹配与搜索等。由 Henry Spencer 开发的正则表达式软件包提供了字符串处理的更为高效和强大的工具。

如果读者熟悉 perl 编程或者 UNIX 和 LINUX shell 编程的话,一定对正则表达式感到熟悉。即使对 UNIX 或 LINUX shell 的正则表达式不熟悉的话,读者也一定使用过下面的 shell 命令吧:

```
$ ls -la ~/ | grep ".profile"
```

上面的 unix 命令在 HOME 路径下列出名称中含有".profile"的文件或目录信息。又比如:

```
$ls -la ~/ | grep "^[d.]"
```

此命令列出 HOME 路径下所有是目录的文件信息。

上面两个 unix 命令中的 `grep` 就是使用正则表达式来过滤信息。`grep` 意为“全局正则表达式”。读者可以参考有关 UNIX 或 LINUX shell 编程的书籍来了解它们是如何使用正则表达式的。

Tcl 中的正则表达式要比 UNIX 或 LINUX 中的正则表达式更丰富和强大。

由 POSIX 定义的正则表达式有两类:扩展正则表达式 (extended REs, EREs) 和基本正则表达式 (basic REs, BREs)。从 Tcl8.0 起增加了高级正则表达式 (advanced REs, AREs)。本章只介绍基本正则表达式和高级正则表达式。

正则表达式的匹配器是用优化的 C 代码来实现的,因此进行模式匹配的速度很快。我们用前面介绍的字符串操作命令如 `string first`、`string match` 等也可以完成模式匹配工作,但是一条正则表达式可以经常可以替代多个的 `string` 命令,使得程序的性能得到提升。

有些 Tcl 命令支持正则表达式的操作,比如前面提到的 `lsearch` 命令和 `switch` 命令都可以支持一个由“-regexp”为标志的基于正则表达式匹配操作。此外,还有两个单独的命令来解析正则表达式,它们是 `regexp` 和 `regsub` 命令。本章首先介绍这两个命令的用法,然后再对正则表达式的语法进行分析。

9.1 regexp 命令

`regexp` 匹配正则表达式与字符串,它的语法为:

[语法]: `regexp ?switches? exp string ?matchvar? ?subMatchVar ... subMatchVar?`

说明:

1. `regexp` 命令比较字符串 `string` 是否与正则表达式 `exp` 部分或者全部匹配,并可以将字符串中的子字符串提取出来。如果字符串的某个子字符串和正则表达式匹配,则返回 1, 否则返回 0。
2. 如果在 `string` 变量后面,还有其它的变量(称之为匹配变量, match variables),则这些变量就保存了那些与正则表达式匹配的子字符串信息(如可能是子字符串的实际内容,或者是界定子字符串的起始、结束的索引数字)。`matchVar` 保存了匹配 `exp` 的字符串,而 `subMatchVar` 依次存放了和 `exp` 各个中单个括号语法(子模式)匹配的子字符串。如下图所示:

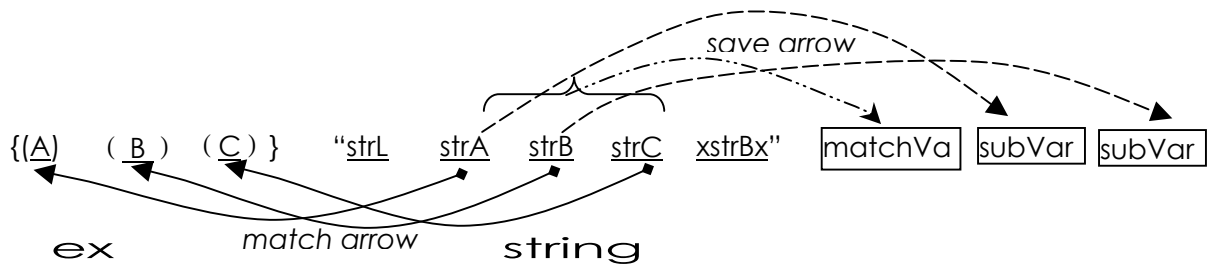


图 6 regexp 匹配过程示意图

上图 `exp` 中的 (A)、(B) 和 (C) 是正则表达式的子模式，`string` 中的 `strA` 等是与对应子模式匹配的子字符串。`matchVar` 中存放了与正则表达式匹配的字符串的信息。`subVar1` 存放了第一个匹配子字符串即 `strA` 的信息，`subVar2` 存放了 `strB` 的信息，由于后面没有其它的变量，就忽略存放第三个匹配字符串。匹配变量 `matchVar`、`subVarN` 中是存放实际字符串内容还是存放表明起始、结束范围的索引数值要 `regexp` 指明（用开关选项 `-indices`）。

3. `regexp` 中的 `switches` 是命令开关选项，这些选项主要有：

表 9-1 regexp 选项

选项	说明
<code>-nocase</code>	<code>exp</code> 中的小写字符可以匹配 <code>string</code> 中的大写和小写字符
<code>-indices</code>	返回界定 <code>string</code> 中匹配区间起始、结束的索引数值。否则返回匹配区间内字符串本身
<code>-expanded</code>	使用扩展语法
<code>-line</code>	等价于同时指定 <code>-lineanchor</code> 和 <code>-linestop</code>
<code>-lineanchor</code>	将 <code>^</code> 和 <code>\$</code> 的行为改为面向行的方式
<code>-linestop</code>	将匹配方式改变成和字符类不匹配换行符
<code>-about</code>	适用于调试，返回有关模式的信息而不是试图与输入进行匹配
<code>-all</code>	让正则表达式在 <code>string</code> 中匹配所有的匹配子字符串，返回匹配次数，而且将最后一次匹配结果存入匹配变量
<code>-inline</code>	将原来存放在匹配变量中的值以列表的形式返回，如果同时使用了 <code>-all</code> ，则返回所有满足匹配结果的值的列表。
<code>-start index</code>	用 <code>index</code> 指定 <code>exp</code> 在 <code>string</code> 中起始匹配位置。如果使用了 <code>-indices</code> ，返回的索引是从输入字符串 <code>string</code> 的绝对起始位置算起而不是从 <code>index</code> 指定位置算起
<code>--</code>	结束选项，如果表达式以 <code>-</code> 开始，则需先用此选项

例 9-1 regexp 的简单例子

```

;# ----- (1)
%set sample "Where there is a will, There is a way."
=>Where there is a will, There is a way.
%set result [regexp {will} $sample match var1]
=> 1
%puts "Result: $result.Match string is :match=$match.var1=$var1"
=> Result: 1.Match string is :match=will.var1=
% set result [regexp {[a-z]+} [a-z]+ [a-z]+] $sample mStr var1 var2 var3]
=>1
% if { $result } {
    puts "Result: $result match: $sample: mStr=$mStr; var1=$var1;var2=$var2;var3=$var3"
}

```

```

}
=>Result: 1 match: Where there is a will, There is a way.:
maStr=here there is;var1=here;var2=there;var3=is

;# ----- (2) 带有 nocase 选项的 regexp 命令
%set result [regexp -nocase {[a-z]+} ([a-z]+) ([a-z]+)] $sample matchStr var1 var2
var3]
=> 1
%if {$result} {
    puts "Result: $result match: $sample:\n matchStr=$matchStr;var1=$var1;var2=$var2;var3=$var3"
}
=>Result: 1 match: Where there is a will, There is a way.:
matchStr=Where there is;var1=Where;var2=there;var3=is

;#----- (3) 指明需要返回索引数值而不是字符串本身
%set result [regexp -indices {[a-z]+} ([a-z]+) ([a-z]+)] $sample mStr var1 var2 var3]
=>1
%if {$result} {
    puts "Result: $result match: $sample:\n mStr=$mStr;var1=$var1;var2=$var2;var3=$var3"
}
=> Result: 1 match: Where there is a will, There is a way.:
    mStr=1 13;var1=1 4;var2=6 10;var3=12 13    ;#返回的标识匹配区间的索引数值

;#----- (4) 使用 -inline 和 -all
%set result [regexp -inline -all -- {[a-z]+} ([a-z]+) ([a-z]+)] $sample ]
=> {here there is} here there is {here is a} here is a

```

9.2 regsub 命令

regsub 命令基于正则表达式完成字符串匹配和替换。

[语法]: **regsub** ?switches? exp string subSpec varName

说明:

1. *switches* 是命令开关选项，主要有：
 - 开关选项 -nocase、-expanded、-line、-linestop、-lineanchor、-start index 和 -- 与 regexp 命令开关选项作用相同
 - all 对所有满足匹配条件的字符串范围进行替换，返回匹配和替换的次数。没有此选项时，只匹配第一个满足匹配条件的匹配范围并用 *subSpec* 替换之。
 2. 替换模式 *subSpec* 可以是普通字符，也可以含有 & 和 \x (x 是一个数字)。当是这两个特殊字符时，& 和 \x 就会被替换成与 *exp* 中的对应模式匹配的 *string* 中的匹配范围内的字符。
- varName* 存放替换后的字符串。

例 9-2 用 regsub 进行字符串替换的简单例子

```

%set sample "Where there is a will, There is a way."
=> Where there is a will, There is a way.
%regsub "way" $sample "lawsuit" sample2
=>1

```

```
%puts "New: $sample2"
=> New: Where there is a will, There is a lawsuit.

;# (2) 使用&和\0的 subSpec:
% set reslut [regexp {[a-z]+} ([a-z]+)] $sample match var1 var2]
=>1
% puts "match=$match; var1=$var1; var2=$var2"
=>match=here there; var1=here; var2=there
% regsub {[a-z]+} ([a-z]+)] $sample {"-&- lawsuit -\1- -\2- "} sample2
%puts "New: $sample2"
=> New: W"-here there- lawsuit -here- -there- " is a will, There is a way.
```

9.3 正则表达式的语法

要深入理解正则表达式(Regular Expressions, REs)的作用和功能, 它的语法则让人感到头痛, 仅是理解这么多特殊的字符与组合就要花很大工夫。但是, 能好好理解正则表达式的语法, 会对正确、高效地使用这个工具起到很大作用。

正则表达式的语法很多, 本章仅讲述主要的语法。请读者参考相关文献或者 Tcl 帮助以获取更多有关正则表达式的知识。

Tcl8.1 之后增加了新的正则表达式语法, 支持 Unicode 和高级正则表达式 (AREs)。这里先介绍适用于所有版本的 Tcl 的基本语法。高级正则表达式的语法将在单独一节介绍。

9.3.1 分支(branch)和原子(atom)

正则表达式由一个或多个分支(branch)组成, 分支之间用符号 | 来相连。每个分支则由零个或者多个原子(atom)组成。这些原子的形式主要有 (标 AREs 为 AREs 支持功能):

- **(re)** 圆括号子表达式, 并报告捕获子串
- **(?:re)** 同上, 但屏蔽报告 (AREs)
- **()** 空圆括号子表达式, 捕获空字符串并报告
- **(?:)** 同上, 但屏蔽报告 (AREs)
- **[chars]** 方括号表达式, 匹配指定的字符集中任意一个字符
- **.** 匹配任意一个字符
- **\k** (k 是非字母数字字符)将字符看作普通字符 (关闭特殊解释), 如\\匹配反斜杠
- **\c** (c 是字母数字字符), AREs 的换码(escape)语法 (AREs)
- **{** 当后跟非数字字符时, 表示匹配左花括弧; 跟数字时, 表示边界 (bound,AREs)
- **x** x 是单个字符 (包含空格), 匹配字符本身

原子还可以使用限定字符, 也可以和量词结合使用。限定匹配和量词见下面的介绍。

9.3.2 基本语法

9.3.2.1 匹配字符

大多数字符可以用来直接作为原子和自身进行匹配, 如下例中的表达式匹配一个 a 和 b 字符组合:

```
% regexp {ab} "This text talks about China." match
=>1
% puts $match
=> ab
```

字符 “.” 用来匹配任意一个字符, 如下例匹配一个 a 和随后任意字符组成的组合:

```
%regexp {a.} "This text talks about China." match
=>1
% puts $match
=>al
```

9.3.2.2 限定匹配

字符匹配可以发生在字符串中的任何位置，一个模式不必分配整个字符串，在匹配地字符前面和后面都可以有未匹配的字符。可以使用定位符号（ $\wedge$ 和 $\$$ ）来指定匹配位置： $\wedge$ 将匹配限制在字符串起始位置， $\$$ 则限制在结尾。可以同时使用这两个符号来匹配整个字符串。如下例匹配所有以字母 T 或 M 开始的字符串：

例 9-3 匹配挂靠的简单例子

```
%regexp {^[TM]+} "This test talks about Chinese." match
=>1 ;#ok
%regexp {^[TM]+} "Man and Woman" match
=>1 ;#ok
% puts $match
=>M
%regexp {^[T]+} "Hello" match
=>0 ;# not match
```

定位符是相对整个输入字符串来说的，而忽略字符串内的换行符，ARE 支持将 $\wedge$ 和 $\$$ 设置为面向行的选项，并增加字符串起始和终止位置定位符 $\backslash A$ 和 $\backslash Z$ 。

对于 ARE，还有其他限定字符，即正前瞻(positive lookahead)和负前瞻(negative lookahead)。

9.3.2.3 方括号表达式与字符集

通过使用方括号括起多个字符的方式 $[xyz]$ ，来指定匹配字符的范围(1)。这一方法使正则表达式可以对字符集中的任意一个字符进行匹配。

可以使用语法 $[x-y]$ 来指定从字符 x 到字符 y 的字符组成的字符集(2)，这个时候要注意，不同的范围不能共享相同的端点，比如 $[a-e-z]$ 就是错误的语法。字符集范围对字符排序顺序有较强的依赖性，易于移植的程序要尽量避免使用这样的语法。

还可以用语法 $[\^xyz]$ 来指定字符集合的补集，即匹配字符为指定字符集以外的任意字符(3)。

例 9-4 字符集匹配的例子

```
;%#(1) 使用方括号括起多个字符
%regexp {[Hh]ello} "He said hello to me." match
=>1 ;#指定或者匹配 hello 或者匹配 Hello，返回 1 表示成功，否则返回 0
%puts $match
=>hello
%regexp {[Hh]ello} "He said Hello to me." match
=>1
% puts $match
=>Hello
;%#(2)
% regexp -indices {-[a-dA-D]} "Effort is applied" match
=>1 ;#匹配 a 到 d 和 A 到 D 之间的任意一个字符，将匹配位置放入 match
中
%puts $match
```

```

=>10 10          ;#在字符串的位置 10 的字符(a)和模式匹配
# (3)
%regexp -all -inline {[^a-zA-Z]} "Effort is applied" ;#匹配所有非 a 到 d 和 A 到 D 的字符
=>Effort{}is{}pplie          ;#单词间的空格被{}显式表示

# (4) 使用特殊字符
regexp -all -inline {[{}*?\\|]} {Th}is man is \an *unhappy ?person}
=> \}is {\an} *un ?pe          ;# 匹配

```

如果要在字符集中使用字符右方括号], 则注意将它放在左方括号之后。在字符集中包含左方括号和花括号则没有位置限制。如可表示为: “[{}]” 。另一个让方括号表达式支持字符[]和-的方法是把它们放在[.和.]之间, 从而使它们变成归整元素, 详见 AREs。

大多数正则表达式的语法字符被放在字符集中时便不再有特殊意义。这意味着不必用反斜杠引用来处理这些特殊字符(反斜杠自身除外)。如 “[+*?| \\\]” 是正确的语法(4)。

高级正则表达式增加了名字和反斜杠换码序列作为诸如空白字符、字母、字母数字等常用字符集的简单记号, 详见 AREs 一节。

9.3.2.4 匹配分支

用|可以连接多个匹配分支, 可以同时用来测试多个匹配模式。可以将|看作或运算, 即只要字符串与一个分支匹配即会进行匹配。如 Hello|hello 即表示或者与 hello 匹配, 或者与 Hello 匹配。

上面这个模式还等价于:

```
(h|H)ello
```

或者是:

```
[h|H]ello
```

9.3.2.5 量词 (Qualifier)

一个原子后面可以跟一个量词, 来指定进行多次匹配。这些量词及含义有:

- * 表示重复零次或多次
- + 表示重复一次或多次
- ? 表示重复零次或一次

例如:

- ba* —— 表示匹配一个 b 后跟有零个或者多个字符 a 的字符串;
- (ab)+ —— 表示匹配具有一个或者多个 ab 序列的字符串;
- . * —— 可以匹配任意字符串或者空字符串。

这几个量词具有贪婪 (greedy) 的特性: 它们尽可能多的匹配字符。ARE 中增加了非贪婪 (non-greedy) 匹配。

在使用量词*和?的时候要特别小心, 它们可以表示零个。因为零个的所有东西都可以匹配成功。如果匹配表达式设置为:

```
[a-z]* 和 [a-z]?
```

用这两个来匹配字符串"123abc"的话，结果可能并非所期望的那样是匹配字符串中的字母 abc 或 ab。这两个模式不是匹配字母，而是先匹配字符串最前端长度为零的子串。可以用 `regexp -indices` 来检查这种行为，改选项指定返回位置而非匹配子串。

例 9-5 使用量词*和?不当引起的错误

```
%regexp -indices {[a-z]?} "123abc" match
=>1
(bin) 187 % puts $match
=>0 -1
(bin) 188 % regexp -indices {[a-z]*} "123abc" match
=>1
(bin) 189 % puts $match
=>0 -1
```

9.3.2.6 子模式与匹配报告捕获

在正则表达式中，可以用圆括号来指定多个匹配子模式。如 `{(exp A) (exp B)}` 指定了两个匹配子模式（注意子模式之间还有一个空格，这个空格参与匹配）。与圆括号内部匹配的范围内的字符会被保存在匹配变量中。如果不用子模式的话，正则表达式只返回与整个正则表达式匹配范围内的字符串，而不能象子模式那样捕获更细致匹配子字符串。

例 9-6子模式捕获

```
%regexp {[a-z]+ [A-Z]+} "allow ln" match var1 var2
=>1
% puts "match=$match;var1=$var1;var2=$var2"
=>match=allow l;var1=;var2=           ;#没有指定子模式，只返回整个匹配子串，
                                     ;#匹配变量 var1 和 var2 为空

% regexp {[a-z]+} ([A-Z].) "allow ln" match var1 var2
=>1
% puts "match=$match;var1=$var1;var2=$var2"
=> match=allow ln;var1=allow;var2=ln   ;#用子模式指定子模式匹配，匹配子串被捕获
```

有时使用了圆括号形式的子模式，又不想记录（捕获）对应的匹配结果报告，则可以使用 ARE 中的非捕获圆括号来屏蔽（ARE 支持）：

`(?:exp)`

例 9-7屏蔽子模式报告

```
%regexp {[a-z]+} (?:[A-Z].) "allow ln" match var1 var2
=>1                                     ;#屏蔽第二个子模式
% puts "match=$match;var1=$var1;var2=$var2"
=> match=allow ln;var1=allow;var2=      ;#var2 为空
```

另外，空的圆括号 `()` 子模式表示捕获空字符串，并报告捕获的子串，`(?:)` 表示捕获子模式但不报告结果（ARE 支持）。

下面的例子给出了一个函数，它从一个给定的 Tcl 脚本文件中，搜索出脚本中定义的所有过程，并以列表形式存放所有过程名。

例 9-8子模式综合运用的例子：搜索脚本文件中定义的过程

```
set proc_list ""
```

```

proc get_proc_name_from_script { filename } {
    global proc_list
    set proc_list ""
    set status [catch {open $filename r} fd]
    if { $status } {
        puts "error to open file $filename"
        return 1
    }

    while {![eof $fd] } {
        set str [gets $fd]
        #puts $str
        if { [regexp {^([[:blank:]]+proc | proc)(?::[:blank:]]+)(^[[:blank:]]+)(?::[:blank:]]+)\} $str match m1 m2 ] } {
            lappend proc_list $m2
        }
    }
    close $fd
    foreach p $proc_list {
        puts $p
    }
} ;#End proc

```

一般的过程定义格式为：

```
proc procName {var1 ...} {...}
```

分析一下上面例子中的正则表达式。首先用"^"将匹配限定到每一行的开始，后跟四个子模式，最后是"{"。第一个子模式为"([[:blank:]]+proc | proc)"，使用了分支，匹配直接以"proc"或者以若干空格后跟"proc"，这是因为过程的定义都是用 proc 命令实现的，可能 proc 前面还有空格。第二个子模式和第四个子模式来匹配若干空格。第三个子模式匹配非空格符。

9.3.2.7 反斜杠引用

使用反斜杠\可以关闭特殊字符的特殊含义，从而将这些字符作为普通字符对待，这些特殊字符有“ . * ? + [] () ^ \$ | \”。

比如要匹配? 则需要写为：

```
\?
```

但是，如果这种特殊字符是用在方括号表达式内，就没有必要使用反斜杠处理（除反斜杠自身），如下面的两种方式效果相同：

```
(\+|\?)
[+|?]
```

另外在 ARE 中新增了反斜杠字符表示\B，所以表达式"\\"和"\B"都可以用来匹配反斜杠字符。

9.3.2.8 匹配优先级

如果一个模式可以匹配一个字符串的多个部分，则正则表达式将与字符串中最早出现的部分匹配。若又有匹配分支存在的话，则会进行最长分支匹配。法则就是：先匹配最早的，然后在最早位置匹配最长分支。使用执行交短匹配的非贪婪量词可以改变这种法则。

9.3.3 高级正则表达式 (AREs)

AREs 的语法向上兼容，即支持基本正则表达式语法。

AREs 的语法多是对基本语法的简化标记，另外 AREs 还增加了其他功能，如非贪婪(non-greedy)量词、换码(escape)、前瞻、归整元素、字符类、等价类等等。

9.3.3.1 反斜杠换码(escape)序列

在以前的正则表达式版本中，反斜杠只被用来屏蔽特殊字符的特殊含义而直接将其作为普通字符使用，如 * ? [] 等。对于其它情况则不予理会，如 \n 被解释为 n 而不是代表换行，在 AREs 中则被解释为换行符。

AREs 中增加了大量新的反斜杠序列，称为换码。这些换码在下表中给出。

表 9-2 AREs 的换码表

换码	说明
\a	告警或振铃(bell)符
\A	只匹配字符串的开始
\b	退格,\u0008
\B	反斜杠同义词，用来替代\\以减少反斜杠的数目
\cX	
\d	
\D	
\e	
\f	
\m	
\M	
\n	
\r	
\s	
\S	
\t	
\uXXXX	
\v	
\w	字母、数字、下划线，等价于[[:alnum:]]
\W	
\xhh	
\y	
\Y	
\Z	
\0	
\x	
\xy	
\xyz	

9.3.3.2 归整元素(collating element)

归整元素可以使字符集中能使用特殊字符或者字符的长名称。当前只有 Tcl 的 ASCII 标点符才有长名称，关于长名称，可在 Tcl 源代码 generic/regc_locale.c 中找到。在方括号表达式中用下面的格式来指定一个归整元素：

[.标识符.]

标识符可以是一个字符如"#"或者是一个长名称。归整元素在使用时，相当于单个字符。我们可以用归整元素来匹配特殊字符如左方括号]和-号，而不必象前面那样要有位置限制。

在将来的版本中，有可能支持将多个字符组成的字符串作为一个归整元素，这样多个字符串就相当于一个字符。比如[[.ch.]*c]可以匹配"chchc"字符串。但是目前的 Tcl 版本还不支持此功能，只是提出了这个概念。

例 9-9 归整元素匹配字符串

```
%regexp {ello[a-z[.].]a-z]} {string is H[ello]World} match
=> 1
%puts $match
=>ello]
```

9.3.3.3 等价类(equivalence class)

在方括号表达式中使用[=char=]来标识字符成为一个等价类，这里的 char 是一个字符。

等价类就是排序时位于相同位置的所有字符。一个等价类中只有一个字符。如过“o”或“ò”是同一等价类的成员，则[[=o=]]、[[=ò=]]和[oò]含义相同。

该语法只有在字符类定义中才有效。

9.3.3.4 字符类(character class)

字符类是各种字符集合的名字，有名字符类语法只有在方括号表达式中使用才有效。用[:和:]将字符类的名字括起来就代表所有属于此类的字符。

标准的字符类有：

- **alpha** A letter.
- **upper** An upper-case letter.
- **lower** A lower-case letter.
- **digit** A decimal digit.
- **xdigit** A hexadecimal digit.
- **alnum** An alphanumeric (letter or digit).
- **print** An alphanumeric (same as alnum).
- **blank** A space or tab character.
- **space** A character producing white space in displayed text.
- **punct** A punctuation character.
- **graph** A character with a visible representation.
- **cntrl** A control character.

例如，alpha 表示所有大写和小写字母集合的名字，则下面两个模式几乎相等：

```
[:alpha:]
[a-zA-Z]
```

它们的区别主要是 alpha 中还包括重音符。

下面的模式匹配诸如退格符、换行符等类似空白符的字符：

```
[:space:]
[\b\f\n\r\t\v]
\s
```

例 9-10 字符类的简单例子

```
%regexp {[[:digit:]]*} "123abc" match
```

```
=>1
% puts $match
=>123
```

有两个特殊的字符类，`[[:<:]]`和`[[:>:]]`。它们分别匹配单词的起始和终止位置。单词被定义为一个或多个匹配`\w`的字符。

9.3.3.5 非贪婪量词

量词`*`、`+`和`?`用来指定重复次数，在默认情况下，它们尽可能多的匹配字符，这被成为贪婪匹配。非贪婪匹配则是尽可能少的匹配字符。可以在这些量词之后加一个`?`号将它们设置为非贪婪匹配。如现在需要匹配“一个或多个非 B 字符后跟一个 B 字符”的模式。在使用贪婪量词的时候，必须明确指定这种非 B 字符模式：

```
%regexp -inline {[\^B]+B} "HelloBelloBello"
=>HelloB
```

否则，如果是：

```
%regexp -inline {.+B} "HelloBelloBello"
=>HelloBelloB
```

可以看出，“.”会贪婪地匹配所有字符直到最后一个 B 为止。而如果用非贪婪字符，遇到第一个 B 字符就会结束：

```
%regexp -inline {.+?B} "HelloBelloBello"
=>HelloB
```

通过使用非贪婪量词，上面的正则表达式从 6 个字符缩减为 4 个。

9.3.3.6 约束量词

语法`{m, n}`是一个量词，它表示前面的匹配项至少匹配 `m` 次，至多匹配 `n` 次。该语法有两个变种：`{m}`表示前面的匹配项要确切匹配 `m` 次；`{m,}`表示前面的匹配项要确切匹配 `m` 或者更多次。所有这些量词都可以在后面加上符号“`?`”来变成非贪婪的。如下面两命令分别重复两次和三次模式匹配：

```
%regexp -inline {(.+?B){2}} "HelloBelloBelloBalloBello"
=>HelloBelloB elloB
% regexp -inline {(.+?B){3}} "HelloBelloBelloBalloBello"
=>HelloBelloBelloB elloB
```

9.3.3.7 回退引用

正则表达式可以使用圆括号来匹配捕获子模式的值，而用回退引用可以再次使用先前的子模式。回退引用用`\1`、`\2`等来指代前面的圆括号子模式。

比如要匹配一个用单引号或者双引号括起来的字符串，需要使用一个包含两种匹配模式的分支匹配：

```
% regexp -inline {"[^\"]*"|'[^']*'} {The "One" means "1"}
=>{"One"} {"One"}
```

而如果用回退引用，则可以更简单一些：

```
%regexp -inline {"'|')[^"]*?\1} {The "One" means "1"}
=>{"One"} {}
```

9.3.3.8 前瞻(lookahead)

前瞻模式一些进行匹配但不消耗任何输入的子表达式，通常出现在模式的结尾。肯定的(positive)的前瞻如果匹配的话就使模式也匹配，否定的(negative)的前瞻在不匹配的情况下使模式匹配。

肯定前瞻表达式的格式为：(?=exp)，否定前瞻表达式的格式为：(?!exp)。exp 为子表达式。

如下面的模式匹配以 A 开头、以.txt 结尾的文件名：

```
^A.*\.txt$
```

下面的模式中使用了圆括号：

```
^A.*(\.txt$)
%regexp -inline {^A.*(\.txt)$} "Allow.txt"
=>Allow.txt .txt
```

上面的模式使用圆括号是为了和下面的前瞻模式作比较。上面模式对应的一个前瞻模式的版本为：

```
^A.*(?=\.txt)$
%regexp -inline {^A.*(?=\.txt)} Allow.txt
=>Allow
```

下面的否定前瞻匹配以 A 开头但不以.txt 结尾的文件名：

```
^A.*(?!\.txt)$
```

9.3.3.9 换行符敏感的匹配

略。

9.3.3.10 嵌入式选项

略。

9.3.3.11 扩展语法

略。

9.3.4 语法小结

略。

9.3.5 其它支持正则表达式的命令

Tcl 命令中，除了 regexp 和 regsub 直接使用正则表达式之外，lsearch 和 switch 命令也可以使用正则表达式。在 lsearch 命令和 switch 命令中，用 -regexp 选项可以标志用正则表达式模式进行匹配。

如下面的代码在列表中匹配以大写字母 M 开头的元素：

```
%set l [list Hello Mello Sello]
```

```
=> Hello Mello Sello
%set index [lsearch -regexp $l {\^M[a-zA-Z]*}]
=>1
```

下面过程中的 switch 语句则以正则表达式来设置匹配分支，各分支匹配以不同大写字母开头的字符串：

```
proc Check {var} {
    switch -exact -regexp -- $var {
        ^A[a-zA-Z]* { puts "Start by A"}
        ^B[a-zA-Z]* { puts "Start by B"}
        ^.[a-zA-Z]* -
        default      { puts "Start by other character"}
    }
}
```

```
%Check Aa
=> Start by A
%Check hello
=> Start by other character
```

Tk 文本组件、Tcl 扩展模块 Expect 也都使用正则表达式，关于这些请参考相关资料。

第 10 章. 名字空间

名字空间为命令和变量提供了新的作用域。一个名字空间就是一个变量和命令的集合。名字空间将变量和命令/过程封装起来以避免和其他名字空间的变量和命令冲突。Tcl 始终有一个默认的名字空间，我们称之为全局名字空间。全局名字空间包含了所有全局变量和命令。

10.1 创建名字空间

使用 `namespace eval` 命令可以让你创建新的名字空间。其语法为：

[语法]: namespace eval spaceName { body }

例 10-1 定义名字空间 Counter:

```
namespace eval Counter {
    namespace export bump
    variable num 0

    proc bump {} {
        variable num
        retrace [incr num]
    }
}
```

上例创建了一个含有变量 `num` 和过程 `bump` 的名字空间 `Counter`。名字空间中的变量和命令与同一程序中的相同名字的变量和命令相隔离。例如在全局名字空间中也会有一个 `bump` 过程的话，它与 `Counter` 内的 `bump` 被认为是不同的过程，可以分别使用。

名字空间的定义是动态的，你可以在需要的时候对名字空间的内容进行更改，也同样是使用 **namespace eval** 命令。如下面的两个命令和上面的名字空间定义命令效果相同(过程 `test` 首先被添加到名字空间，然后第二次的时候用 `rename` 将其删除)：

例 10-2 动态定义名字空间

```
namespace eval Counter {
    variable num 0
    proc test {args} {
        return $args
    }
}

namespace eval Counter {
    namespace export bump

    proc bump {} {
        variable num
        return [incr num]
    }

    rename test ""
}
```

名字空间可以嵌套，一个被嵌套在父名字空间的名字空间可以与外部隔离的。

10.2 用::限定符来使用变量和过程

可以使用限定符`::`来使用名字空间的变量和过程。如

Counter::test 就可以调用名字空间 Counter 的过程 test，而\$Counter::num 引用 Counter 中的变量 num。

对于使用全局变量和过程，可以有两种方法，一种是用 global 声明，另一种是用限定符::引导全局变量和过程，如::num 则引用全局变量 num，用\$::num 来获得其值。

直接以::作为前导的名字我们称之为“全限定名”，它指明在全局名字空间中对其进行解析。而没有::作为前导的名字为“部分限定名”，它指明从当前的名字空间中对其进行解析。如

```
::foo::x
```

表明 x 属于 foo 为全局名字空间的名字空间，而

```
foo::x
```

表明 x 属于 foo 为当前名字空间（不一定是全局名字空间）的名字空间。

如果是引用全局变量，最好使用 global 先声明再使用，这样效率会高些。

下面的例子给出了通过限定名来使用相应名字空间的过程是如何工作的：

例 10-3 限定名比较的简单例子

```
% namespace eval foo {
    namespace eval foo {                ;#嵌套的名字空间
        proc test {} {
            puts "In ::foo::foo::test"
        }
    }

    proc test {} {
        puts "Enter ::foo:test"
        puts "Call ::foo::foo::test"
        foo::test                        ;#调用::foo:foo::test
        puts "End ::foo:test"
    }

    foo::test                            ;#调用::foo::foo::test
    ::foo::test                          ;调用::foo::test
}

=>In ::foo::foo::test
Enter ::foo:test
Call ::foo::foo::test
In ::foo::foo::test
End ::foo:test
```

10.3 名字空间的变量

名字空间中的变量用 variable 定义，它类似与全局变量，名字空间中的过程要使用 variable 命令可以使用名字空间的变量。

[语法]: `variable name ?value? ?name value ? ?... name value?`

可以用 variable 先声明变量而将赋值操作留到后面。但如果先声明而在不赋值情况下引用变量的话，会出错。这因为在第一次用 set 命令对变量赋值时，变量才正式被创建。

在名字空间的代码中使用变量需要小心，如果用 `variable` 声明则显然他们是名字空间变量。但是，当使用时忘记声明它们，则要么使用同名的全局变量，要么自动被创建为名字空间变量。如下面的例子中，因为没有在名字空间内事先声明变量 `i` 和 `j`，则全局变量的 `i` 被使用，而且创建了名字空间的变量 `j`：

例 10-4 名字空间自动处理引用变量的归属

```
%set i 20
=> 20
%namespace eval test {
    ;#i 没有被事先声明为名字空间的变量，则会引用全局变量 i
    for {set i 1} { $i <=5} {incr i} {
        puts -nonewline "i=$i; "
    }

    puts "\n"
    for {set j 1} { $j <=5} {incr j} {
        puts -nonewline "j=$j; "
    }
    puts "\n"
}
% puts $i
=>6                ;#使用了全局变量 i
% puts $foo::j
=>6                ;#创建了新的变量::foo::j
```

10.4 过程的进口与出口

在名字空间内，可以用命令 **namespace export** 将命令出口，然后在名字空间之外就可以用 **namespace import** 命令将特定名字空间的命令进口。从名字空间进口命令后，该命令就作为本级名字空间的一个命令而存在，可不用使用 `::` 限定符而直接使用进口命令。

例 10-5 过程的进口与出口的简单例子

```
% namespace eval Counter {
    ;#将过程 dump 输出
    namespace export dump
    proc dump {} {
        puts "I'm Counter's dump"
    }

    proc test {} {
        puts "I'm Counter's test"
    }
    variable procList [info proc]                ;#保存本名字空间中的过程列表
}
% namespace import Counter::dump                ;#进口命令
% namespace import Counter::dump                ;#命令已经存在，不能重复进口
=> can't import command "dump": already exists
% namespace import -force Counter::dump ;#强制输入重复名称的过程，本名字
                                           ;#空间原来的过程被覆盖
% namespace import Counter::test
% test                                           ;#test 并没有被输出！
```



```

=> invalid command name "test"
% Counter::test                ;#使用::来调用 test
=> I'm Counter's test
% dump                          ;#dump 则可直接调用
=> I'm Counter's dump
% set index [lsearch [info proc] dump]
=> 5                            ;#dump 已经为全局过程
% puts $Counter::procList      ;#在 Counter 内能“看到”的过程
=> test dump
% namespace forget dump        ;#取消命令进口
% set index [lsearch [info command] dump]
=> -1                          ;#进口命令已经被取消

```

说明:

1. 只有过程才能用这两个命令，对变量无效。
2. 使用进口命令的时候要小心，避免不同名字空间的相同名称的命令冲突。如果一定进口相同名字的命令，可用开关选项-force 来强制输入，则原来的命令被覆盖。
3. 当用命令 **pkg_mkIndex** 来创建软件包索引的时候，应当清楚地知道，只有被出口的命令名才能出现在索引文件 pkgIndex.tcl 中。
4. 使用 **info proc** 和 **info command** 可以显示本名字空间中可见的过程或命令，这些命令包括自己定义的和进口的命令和过程。
5. 可以使用 **namespace forget** 来取消指定的进口命令

10.5 内省(introspection)

如上所述命令 **info command** 和 **info proc** 返回本名字空间可见命令或过程，而并不能反映出命令和过程在哪个名字空间中被定义的。而用命令 **namespace origin** 可以用来查看命令定义的名字空间。

例 10-6 namespace origin 命令例子

```

;#接上例
%namespace origin dump
::Counter::dump

```

10.6 名字空间命令集

下表给出了名字空间的操作子命令，这些子命令以 **namespace** 开始。

表 10-1 namespace 命令

命令	说明
namespace current	返回当前名字空间，全局变量为::
namespace children ?name? ?pat?	返回本名字空间中嵌套名字空间列表。 <i>pat</i> 为 string match 模式，用来限制返回内容
namespace code script	
namespace delete name ?name...?	删除名字空间的变量和命令
namespace eval name cmd ?args...?	如果名字空间 <i>name</i> 不存在，则创建之。如果有多个参数，则会象 eval 命令那样来处理。
namespace export ?-clear? ?pat1...patN?	将命令添加到输出列表。如果没有指定模式，就返回输出列表
namespace forget pat ?pat ... ?	取消名字进口

namespace import <i>?-force?</i> <i>pat ?pat...?</i>	进口命令
namespace inscope <i>name</i> <i>cmd ?args...?</i>	
namespace origin <i>cmd</i>	返回 <i>cmd</i> 的定义/原始名字空间
namespace parent <i>?name?</i>	返回父名字空间名
namespace qualifiers <i>name</i>	返回 <i>name</i> 中最后一个:: <i>之前的内容</i> ，如:: <i>a::b::c</i> 中的:: <i>a::b</i>
namespace which <i>?flag? name</i>	返回 <i>name</i> 的全限定路径。 <i>-flag</i> 为 <i>-command</i> 、 <i>-variable</i> 或 <i>-namespace</i> 中的一个
namespace tail <i>name</i>	返回 <i>name</i> 中最后:: <i>的组成</i> ，如:: <i>a::b</i> 中的 <i>b</i>

第 11 章. 跟踪与调试

11.1 clock 命令

用 clock 命令可获得当前系统时间，并能根据指定的格式处理时间字符串。

下表列出了 clock 相关命令，

表 11-1 clock 命令

命令	说明
clock clicks ?-milliseconds?	返回有赖于系统的高分辨率整型时间值。可以指定以毫秒为单位。
clock format value ?-format string? ?-gmt boolean?	将整型时间值转换为人们可读格式。这个整数值可能是 clock seconds、clock scan 或者是带 atime、mtime 或 ctime 选项的 file 命令返回的时间值。format 后面的 string 给出了具体的格式化格式。如果使用了 -gmt 选项，则其后跟一布尔型值。如果是 true，表明时间值格式化为格林威治标准时间，如果是 false，则格式化为本地时区时间。
clock scan dateString ?-base clockVal? ?-gmt boolean?	将给定的日期字符串转换为以秒为单位的时钟值（见 clock seconds）。如果指定 -base，则以整型时钟值 clockVal 为基准来计算。
clock seconds	返回以秒为单位的整型时钟值。

11.1.1 clock clicks 命令

clock clicks 返回高分辨率系统时钟计数器值，一般仅用于测量经过的时长。click 分辨率取决于系统本身。如果使用了 -milliseconds 选项，则分辨率以毫秒为粒度。如下面的代码计算出若干秒之内时钟滴答次数：

例 11-1 计算系统时钟滴答数

```

proc click {period} {
    set t1 [clock clicks]
    #Wait for $period seconds
    after [expr $period * 1000]
    set t2 [clock clicks]
    puts "[expr ($t2 - $t1)/$period] Clicks/Second"
}

%click 10
=>1000 Clicks/Second

```

11.1.2 clock seconds 命令

以秒为单位返回当前系统的时间。这个时间通常是从“epoch”算起。如：

```

%clock seconds
=>1071708320

```

11.1.3 clock format 命令

clock format 命令将整型时间值格式化为一个可读的日期字符串。这个时间值可以是 clock seconds、clock scan 命令，或者是带选项 atime、mtime 和 ctime 的 file 命令的返回值。日期字符串的格式可以由 -format 后的格式化字符串 *string* 确定。格式化字符串中使用一个或多个域描述符。域描述符由 % 后跟一个域描述符字符组成。有效的域描述符如下表所列：

表 11-2 clock format 的域描述符

域描述符	说明
%%	Insert a %.
%a	Abbreviated weekday name (Mon, Tue, etc.).
%A	Full weekday name (Monday, Tuesday, etc.).
%b	Abbreviated month name (Jan, Feb, etc.).
%B	Full month name.
%c	Locale specific date and time.
%d	Day of month (01 - 31).
%l	Hour in 12-hour format (00 - 12).
%j	Day of year (001 - 366).
%m	Month number (01 - 12).
%M	Minute (00 - 59).
%p	AM/PM indicator.
%S	Seconds (00 - 59).
%U	Week of year (00 - 52), Sunday is the first day of the week.
%w	Weekday number (Sunday = 0).
%W	Week of year (00 - 52), Monday is the first day of the week.
%x	Locale specific date format.
%X	Locale specific time format.
%y	Year without century (00 - 99).
%Y	Year with century (e.g. 1990)
%Z	Time zone name.

表 11-3 clock format 特定于 UNIX 系统的域描述符

%D	Date as %m/%d/%y.
%e	Day of month (1 - 31), no leading zeros.
%h	Abbreviated month name.
%n	Insert a newline.
%r	Time as %l:%M:%S %p.
%R	Time as %H:%M.
%t	Insert a tab.
%T	Time as %H:%M:%S.

NOTE: If the **-format** argument is not specified, the format string **"%a %b %d %H:%M:%S %Z %Y"** is used as default. If the **-gmt** argument is present the next argument must be a boolean which if true specifies that the time will be formatted as Greenwich Mean Time. If false then the local timezone will be used as defined by the operating environment.

例 11-2 clock format 在 windows 上的简单例子

```
#如果没有-format 选项, 则使用默认格式化字符串"%a %b %d %H:%M:%S %Z %Y"
%clock format [clock seconds]
=>Thu Dec 18 10:42:06 涓-鏃-癸未 2003
```

```
%clock format [clock seconds] -format \
    "WEEK=%A MONTH=%B DATE=%d TIME=%H:%M:%S in Year%Y"
=> WEEK=Thursday MONTH=December DATE=18 TIME=10:57:03 in Year2003
```

本例是在中文 Windows2000 上测试的。例子中的第一个命令使用了默认格式化字符串。**%Z** 要求输出时区名, 在输出的结果中, 这部分出现了乱码 (波浪线标出部分)。也许你很少遇到这个问题而根本不去关心它。此问题涉及到了系统的字符集和编码的问题。下面简要介绍一下字符集和编码。

不同的语言使用不同的字母或字符集, 为此定义了不同的编码(encoding)来支持不同的字符集。比如大家熟悉的 Unicode 编码、UTF-8 编码、Big5 编码、GB2312 (仿宋) 编码等。Tcl 内部默认使用 Unicode 16 位编码。

每个计算机系统对他们的文件设置了一种标准的系统编码方式。Tcl 会读文件并自动将它们从系统编码转换成 Unicode。输出的时候则相反, 将 Unicode 自动转换成系统编码。

你可以用 **"encoding names"** 列出 Tcl 所知道的编码格式, 这些编码被保留在 Tcl 脚本库的 encoding 目录的文件中。

而用 **"encoding system"** 命令可以查看当前系统正使用的编码方式。

用 **"encoding system encoding"** 可以改变系统的编码方式。不过在使用这个命令的时候要小心, 不恰当的编码方式可能使 Tcl 变的不可用, 即不再响应你的输入。

下面有关系统 encoding 例子是在 Windows2000 上 wish 界面内运行的:

例 11-3 Tcl 识别的系统编码方式

```
%encoding system
=>cp936

(bin) 2 %lsort [encoding names]
=>ascii big5 cp1250 cp1251 cp1252 cp1253 cp1254 cp1255 cp1256 cp1257 cp1258
cp437 cp737 cp775 cp850 cp852 cp855 cp857 cp860 cp861 cp862 cp863 cp864
cp865 cp866 cp869 cp874 cp932 cp936 cp949 cp950 dingbats ebcdic euc-cn euc-
jp euc-kr gb12345 gb1988 gb2312 identity iso2022 iso2022-jp iso2022-kr iso8859-1
iso8859-10 iso8859-13 iso8859-14 iso8859-15 iso8859-16 iso8859-2 iso8859-3 iso8859-4
iso8859-5 iso8859-6 iso8859-7 iso8859-8 iso8859-9 jis0201 jis0208 jis0212 koi8-r koi8-u
ksc5601 macCentEuro macCroatian macCyrillic macDingbats macGreek
macIceland macJapan macRoman macRomania macThai macTurkish macUkraine
shiftjis symbol tis-620 unicode utf-8

#更改系统编码为 UTF-8 方式
(bin) 3 %encoding system utf-8
```

编码的名称反映了它们的来源。“cp”代表“code pages”, 是 Windows 系统用的。“mac”用于 Macintosh 系统。“iso”、“euc”、“gb”和“jis”等来自各种标准化组织。

对于不同的文件也可能使用不同的编码方式。Tcl 中会自动转换成 Unicode 方式。不过你可以用 `fconfigure` 命令来设置这种编码方式。如你想读取一个以标准俄文编码(iso8859-7)的文件:

```
set in [open READM r]
fconfigure $in -encoding iso8859-7
```

用 `fconfigure` 命令也可以查看文件的编码格式, 如下面的命令查看标准输入、标准输出和标准错误输出的编码格式:

```
%puts "STDIN:[fconfigure stdin]\nSTDOUT:[fconfigure stdout]\nSTDERR:[fconfigure stderr]"
=>STDIN:-blocking 1 -buffering none -buffersize 4096 -encoding utf-8 -eofchar {} -translation lf
      STDOUT:-blocking 1 -buffering none -buffersize 4096 -encoding utf-8 -eofchar {} -translation lf
      STDERR:-blocking 1 -buffering none -buffersize 4096 -encoding utf-8 -eofchar {} -translation lf
```

返回到上面格式化输出时间信息的例 11-2, 出现乱码可能是由于系统使用 `cp936` 编码格式, 在转换到 Unicode 的时候有问题。我们不妨尝试将系统编码改成其它编码格式, 如 UTF-8 试一试。另外, 可以在 `clock format` 中使用 `"-gmt true"` 设置为格林威治标准来屏蔽乱码。

例 11-4 消除 `clock format` 输出中的乱码

```
#将系统编码设置为 UTF-8, 输出也为 UTF-8
%encoding system utf-8
%clock format [clock seconds]
=>Thu Dec 18 12:20:15 中国标准时间 2003
% clock format [clock seconds] -gmt true
=>Thu Dec 18 4:20:25 GMT 2003
```

```
#将系统和标准输出编码都设置为 Unicode
%encoding system unicode
%fconfigure stdout -encoding unicode
(clock format [clock seconds])
=>Thu Dec 18 12:25:16 中国标准时间 2003
```

有关其他字符集和编码问题, 请参考相关资料。

11.1.4 clock scan 命令

`clock scan` 命令用来解析一个日期字符串并返回对应的时间值(以秒为单位)。此命令可以处理各种格式的日期, 如果没有指明年份, 则采用当前年份。

可以用 `-base` 选项指明基准日期时间。如:

```
%encoding system utf-8
%clock format [clock scan "1 day" -base [clock scan 1999-10-31]]
=>Mon Nov 01 0:00:00 中国标准时间 1999
% clock format [clock scan "24 hours" -base [clock scan 1999-10-31]]
=>Mon Nov 01 0:00:00 中国标准时间 1999
```

`clock scan` 的日期字符串可以是任何可接受的形式, 如:

```
%clock format [clock scan "1 week ago"]
=>Thu Dec 11 12:43:01 中国标准时间 2003
%clock format [clock scan "2 days ago"]
```

```
=>Tue Dec 16 12:43:29 中国标准时间 2003
%clock format [clock seconds]
=>Thu Dec 18 12:48:52 中国标准时间 2003
% clock format [clock scan "next Monday"]
=>Mon Dec 29 0:00:00 中国标准时间 2003
```

也可以用 一个正或负的整数作为增量指示，如：

```
%clock format [clock scan "10:30:44 PM 1 week"]
=>Thu Dec 25 22:30:44 中国标准时间 2003
% clock format [clock scan "10:30:44 PM -1 week"]
=>Thu Dec 11 22:30:44 中国标准时间 2003
```

11.2 info 命令

info 命令允许 Tcl 程序从 Tcl 解释器获得有关当前解释器状态的信息。比如，通过 info 的子命令可以知道某个过程、变量或者命令是否在当前的解释器中存在，这样你就可以在使用一个变量或者调用一个过程的时候，先测试一下变量或者过程是否存在，从而避免操作不存在的变量、过程或命令而引起的错误。

例 11-5 用 info 测试变量是否存在

```
% set a [info exists b]
=> 1 ;#参数 b 存在
% set a [info exists q]
=> 0 ;#参数 q 没定义
```

info 的命令集见下表

表 11-4 info 命令集

命令	说明
info args <i>procedure</i>	过程 <i>procedure</i> 的参数名列表
info body <i>procedure</i>	过程 <i>procedure</i> 的（执行命令体的）内容
info cmdcount	已经执行的命令数
info commands <i>?pattern?</i>	列出所有命令，或与 <i>pattern</i> 匹配的命令。命令包括 Tcl 内建命令和过程
info complete <i>command</i>	测试 <i>command</i> 是否是一条完整命令，是则返回真
info default <i>proc arg var</i>	测试过程 <i>proc</i> 的参数 <i>arg</i> 是否有默认值，有则返回 1 并将默认值保存到 <i>var</i> 中
info exists <i>variable</i>	测试变量 <i>variable</i> 是否存在
info globals <i>?pattern?</i>	返回所有全局变量或者与 <i>pattern</i> 匹配的全局变量列表
info hostname	返回当前主机名
info level	当前过程调用的层次，全局作用域为 0
info level <i>number</i>	返回指定层次的命令及其参数的列表
info library	Tcl 库目录路径名
info loaded <i>?interp?</i>	加载到解释器 <i>interp</i> 中的库的列表
info locals <i>?pattern?</i>	返回全部局部变量或与 <i>pattern</i> 匹配的局部变量列表
info nameofexecutable	返回当前程序文件名（如 wish8.3, tclsh 等）
info patchlevel	Tcl 的补丁级别
info procs <i>?pattern?</i>	返回所有 Tcl 过程或与 <i>pattern</i> 匹配的过程名列表
info script	正在处理的脚本名，如一个 .tcl 文件正在被 source 命令

	处理，则文件内部的[info script]命令将返回此文件名。 如果没有正在处理的文件，则返回空
info sharedlibextension	共享库目录的扩展名(如.dll 在 windows 上，.so 在 unix 上)
info tclversion	Tcl 版本号
info vars ?pattern?	返回所有可见变量或与 pattern 匹配的可见变量列表

11.2.1 info level

用 info level 命令可以控制带有循环嵌套调用过程的循环层次数。例 11-6 中，fact 过程通过循环调用自身完成阶乘算法，通过将 info level 的返回值与阶乘值比较判断

例 11-6 用 info level 命令控制过程的循环嵌套

```

proc fact {val} {
    set level [info level]
    puts "Current level: $level val: $val"

    if {$level == $val} {return $val;}
    set num [expr $val - $level]           ;#将 val 的值减去当前 level 的值
    return [expr $num * [factorial $val]] ;#循环调用
}

% set res [fact 3]                        ;#3 的阶乘=3!
=> Current level: 1 val: 3
Current level: 2 - val: 3
Current level: 3 - val: 3
6
%puts "The result is: $res"
=> The result is: 6
%info args fact                          ;#输出 fact 的参数名
=>val
( % info body fact                       ;#输出过程 fact 的内容
=> set level [info level]
puts "Current level: $level val: $val"

if {$level == $val} {return $val;}
set num [expr $val - $level]             ;#将 val 的值减去当前 level 的值
return [expr $num * [factorial $val]]    ;#循环调用
%info library                            ;#库路径信息
=>D:/software/Tcl/lib/tcl8.3
% info sharedlibextension                ;#库扩展名信息
=> .dll

```

11.2.2 info exists

info exists 命令可以测试一个变量是否存在。在使用一个变量之前，用此命令先检测一下变量是否已经存在，从而避免因使用了未定义的变量引起的错误。

11.3 trace 命令

trace 命令用于变量操作跟踪，它注册一条命令到一个变量，只要这个变量发生指定的变化，如被读、写或者复位（unset）的时候，注册命令就会被调用来进行相关的处理。

11.3.1 trace variable

[语法] : trace variable varName operations command

说明：

1. operations 为变量操作选项，为下列选项的一个或者多个：
 - r 代表只读
 - w 代表只写
 - u 代表复位或 unset 操作
 operations 说明当变量发生这些动作时，命令就会被调用。
2. command 为注册命令，它必须能够接收三个参数（。当变量发生 operations 中的某一个动作的时候，command 就会执行：

command var1 var2 var3

其中，var1 代表变量名或者数组名。var2 是数组元素索引，如果跟踪的是普通变量（非数组变量），或者跟踪的数组被设置为复位跟踪且数组已经被复位，则此参数为空。var3 是跟踪的动作，即满足 options 定义的某个选项对应动作。

3. 可以多次调用 trace variable 为同一变量注册多条命令，这些命令会在指定条件满足时顺次执行。比如可以为一个变量的读、写和复位不同操作分别注册不同的命令，也可以为同一个操作注册多条命令。

例 11-7 用 trace variable 跟踪变量的简单例子

```

;# STEP1. Define two trace procedure
proc traceP1 {args} {
    puts "---Enter proc traceP1---"
    puts "There are [llength $args] input variables for trace command"
    puts "The input three variables' value are:"
    set varName [lindex $args 0]
    set index [lindex $args 1]
    set action [lindex $args 2]
    puts "varName: $varName \n index: $index \n action: $action"
    puts "---Proc traceP1 end---"
}

proc traceP2 {varName arrIndex op} {
    puts "---Enter proc traceP2---"
    switch -exact -- $op {
        w {set option "setted"; }
        r {set option "read"}
        u {set option "unsetted"}
    }

    puts "Three input variables's values are: "
    puts "varName: $varName \n index: $arrIndex \n action: $op"
    puts "Variable $varName was $option"

    puts "---Proc traceP2 end---"
}

```

```

}

% set a 1
=> 1
% array set b { one Hello two World }
% trace variable a rwu traceP1      ;#为变量 a 的读、写与复位注册过程 traceP1
% trace variable a r traceP2        ;#为变量 a 的读注册过程 traceP2
% trace variable b rwu traceP2      ;#为数组 b 的读、写与复位注册过程
traceP2
% trace variable b r traceP1        ;#为数组 b 的读注册过程 traceP1
% puts $a                          ;#变量 a 的读操作被注册了两个命令
=> ---Enter proc traceP2---
    Three input variables's values are:
        varName: a
        index:
        action: r
        Element of array a was read.
    ---Proc traceP2 end---
    ---Enter proc traceP1---
    There are 3 input variables for trace command
    The input three variables' value are:
        varName: a
        index:
        action: r
    ---Proc traceP1 end---
    1
%puts $b(one)                      ;# 变量 b 的读也被注册了两个命令
=>---Enter proc traceP1---
    There are 3 input variables for trace command

    The input three variables' value are:
        varName: b
        index: one
        action: r
    ---Proc traceP1 end---
    ---Enter proc traceP2---
    Three input variables's values are:
        varName: b
        index: one
        action: r
    Variable b was read
    ---Proc traceP2 end---
    Hello
%set a 10                          ;# 变量 a 的写只被注册了一个命令
=>---Enter proc traceP1---
    There are 3 input variables for trace command

    The input three variables' value are:
        varName: a
        index:
        action: w

```

```
---Proc traceP1 end---  
10
```

从上例可以看出，为变量注册命令后，先执行注册命令，然后在执行对参数的具体操作。用 `trace` 命令还可以作到对变量操作的限制，如可以限制变量为只读变量，当试图对变量进行其他操作时，注册命令就返回错误。

11.3.2 `trace vdelete`

`trace vdelete` 删除用 `trace variable` 为变量所做的一条注册命令。

[语法]: `trace vdelete varName operations command`

`trace vdelete` 的语法和 `trace variable` 的语法一致。

11.3.3 `trace vinfo`

`trace vinfo` 返回变量跟踪设置的信息

例 11-8 `trace vinfo` 的例子

```
% trace vinfo a  
=>{r traceP2} {rwu traceP1}
```

第 12 章. 脚本库与软件包

本节讲述如何在库中组织软件包(package)，并介绍如何使用 Tcl 提供的处理软件包命令。

一个 Tcl 程序库可能需要属于不同软件包的多个 Tcl 脚本文件组成。以一个用 Tcl 语言实现的集成自动测试软件为例，这个软件中，每个不同被测系统的可能有自己的命令软件包，比如用来配置 CISCO 路由器的软件包。除此之外，还可能有控制测试仪表如 ADTECH 的软件包等等。

这些不同的软件包可能由不同组织进行开发，而且可能存放在不同路径的库里。在测试的时候，可以编写一个主程序，让主程序来调用不同软件包中的命令。这时就要给主程序提供一种机制，告诉它到什么地方找到相应的软件包，如何能从软件包中调入需要的命令等等。有了这些机制，主程序就可以在需要的时候调用对应命令，而不需要将所有软件包都加载。

12.1 声明和使用软件包

12.1.1 软件包定位

Tcl 解释器有一个全局变量 `auto_path`，它在 `tclsh` 或 `wish` 启动的时候被初始化。`auto_path` 中包含了 Tcl 或 Tk 脚本文件的库目录的列表。可以通过 `lappend` 命令来设定这个变量来定位所需要的软件包路径。

下面的例子显示了 `wish` 初始化的 `auto_path` 变量的内容，并用 `lappend` 命令添加新的路径。

例 12-1 `auto_path` 的内容与操作

```
% set auto_path
=>D:/software/Tcl/lib/tcl8.3 D:/software/Tcl/lib D:/software/Tcl/lib/tk8.3
% lappend auto_path e:
=>D:/software/Tcl/lib/tcl8.3 D:/software/Tcl/lib D:/software/Tcl/lib/tk8.3 e:
```

把脚本存放到库目录下并设置了 `auto_path`，这还不一定能正确使用软件包，你必须在脚本中事先声明所属软件包，然后在使用时显式对其调用。

12.1.2 声明软件包命令

在脚本文件中用 `package provide` 命令来声明本脚本是属于或提供哪个软件包。

[语法]: `package provide pkgName version`

`pkgName` 是软件包的名字，`version` 是软件包的版本号，格式为“主版本号.子版本号”。如在一个脚本文件 `ax4k.tcl` 中有

```
package provide AX4000 1.0
```

则表示本脚本文件是属于版本为 1.0 的软件包 AX4000。

12.1.3 加载软件包命令

使用软件包中的内容之前，需要用 `package require` 命令将对应软件包加载到 Tcl 解释器中。

[语法] `package require ?-exact? pkgName ?version?`

`pkgName` 为要加载软件包的名字，`version` 为指定版本号（可选），`-exact` 选项进行版本精确匹配。如果不指定 `version`，则加载版本最高的那个软件包。

如在脚本文件 `ax4k.tcl` 中同时有如下几个命令：

```
package provide AX4000 1.0
package require AX_CTRL 1.2
package require -exact AX_CALC 1.0
```

则说明 ax4k.tcl 提供 AX40001.0 版本的软件包，同时需要加载 AX_CTRL 和 AX_CALC 两个软件包以使用这两个软件包的内容（如变量和命令）。

12.1.4 自动加载与软件包索引

package require 命令会遍历在 auto_path 列出的路径，并取相关软件包，但不是直接从库路径中的所有脚本文件或者库文件中搜索所需软件包，而是通过 source 库路径下的 pkgIndex.tcl 脚本完成的。pkgIndex.tcl 中存放了本路径下有那些软件包，及各个软件包在那个脚本文件中的信息。pkgIndex.tcl 脚本内容可以由命令 pkg_mkIndex.tcl 计算得出。pkg_mkIndex 的语法为：

[语法]: pkg_mkIndex ?switches? directoryName pattern ?pattern ...?

其中 directoryName 为目的目录名（相对或绝对路径）。pattern 为文件名匹配模式，这个模式和 glob 命令以及 UNIX 命令 ls 的模式相同。switches 为开关选项，主要有：

表 12-1 pkg_mkIndex 命令开关选项

选项	说明
-direct	产生带有 source 和 load 命令的索引，这会导致当执行 package require 时软件包的内容会被自动加载。
-load pattern	适用于从解释器。动态的将匹配 pattern 的软件包加载到从解释器，需要这个开关的原因是因为 tcbload 软件包需要加载用 TclPro 编译器编译的.tcb 文件
-verbose	将命令处理过程回显到显示终端

使用时，一般用 -direct 选项。

例 12-2 pkg_mkIndex 命令

```
% pkg_mkIndex ./ *.tcl    ;#在当前目录下，从*.tcl 文件中搜索软件包以生成索引文件
% dir                    ;#查看 pkgIndex.tcl 是否已经生成
=>:
:
  pkgIndex.tcl
% more pkgIndex.tcl      ;#pkgIndex.tcl 内容
=>
# Tcl package index file, version 1.1
# This file is generated by the "pkg_mkIndex -direct" command
# and sourced either when an application starts up or
# by a "package unknown" script. It invokes the
# "package ifneeded" command to set up package-related
# information so that packages will be loaded automatically
# in response to "package require" commands. When this
# script is sourced, the variable $dir must contain the
# full path name of this file's directory.
```

```

package ifneeded AX4000_ETH_STUB 1.0 [list source [file join $dir
ta_traffic_start_eth.tcl]]\n[list source [file join $dir
ta_bursty_traffic_start_eth.tcl]]\n[list source [file join $dir
ta_traffic_stop_eth.tcl]]\n[list source [file join $dir
ta_check_end2end_eth.tcl]]\n[list source [file join $dir ta_config_eth.tcl]]\n[list
source [file join $dir ta_start_measurement_eth.tcl]]\n[list source [file join $dir
ta_stop_measurement_eth.tcl]]\n[list source [file join $dir
ta_stop_multi_measurement_eth.tcl]]\n[list source [file join $dir
ta_start_multi_measurement_eth.tcl]]

```

有了 pkgIndex.tcl 索引文件，package require 命令就会根据索引在当前目录内加载软件包。但是如果当前目录中没有其它.tcl 脚本文件，加载过程并没有结束，而会继续在所有子目录内继续搜索 pkgIndex.tcl。如果子目录内有 pkgIndex.tcl 而且有需要的软件包，就会执行加载操作。如果此库目录下所有子目录内也找不到相关软件包，则会转到 auto_path 指定的下一个目录下搜索。这样一直持续下去，直到加载指定软件包所有内容为止。如果没有找到相应软件包，则 package require 命令报错。

从上例显示的索引文件的内容可以看出，对于脚本文件中定义的软件包，将用 source 命令进行加载。\$dir 会被替换成 pkgIndex.tcl 所在的实际目录名，list 将各个 source 命令组织成命令列表，最后会调用 eval 命令执行每个 source 列表命令。即加载过程为：

```
eval [[list source [file join $dir ta_traffic_start_eth.tcl]]
```

命令 package ifneeded AX4000_ETH_STUB 1.0 cmd1\n cmd2\....用来设定软件包对应的加载命令（即 source 命令列表）。

从上面对 pkgIndex.tcl 内容的分析知，我们完全可以不用 package require 命令而直接操作 pkgIndex.tcl 来加载软件包。下面的例子中的 load_pkg 过程就完成了这个功能。load_pkg 根据输入的软件包名称在 auto_path 各个路径下查找是否有 pkgIndex.tcl 索引文件，如果有，则打开该文件并搜索指定软件包，如果找到，就用 eval 命令计算对应行，从而完成软件包加载。在测试的时候，可以按以下步骤进行：

1. 创建一个子目录，在子目录下放置多个.tcl 脚本，并在脚本内输入"package provide pkgName"
 2. 然后用 pkg_mkIndex 命令生成 pkgIndex.tcl 索引文件
 3. 用 lappend 命令将子目录绝对路径添加到 auto_path 内
 4. 用 package names 显示软件包 pkgName 是否已经加载，如果是，则用 package forget 命令将其去掉
 5. 调用 load_pkg pkgName 加载软件包
 6. 用 package names 检查软件包 pkgName 是否已经加载
- 有关 package 的各条命令，请参见后面的 package 命令集。

例 12-3 简单的软件包加载的例子

```

proc load_pkg { pkgName } {
    global auto_path
    set count [llength $auto_path]
    for { set i 0 } { $i < $count } { incr i } {
        set dir [lindex $auto_path $i]
        puts "path=$dir"
        if [catch { open [file join $dir pkgIndex.tcl] } fd ] {
            # no pkgIndex.tcl
            puts "no pkgIndex.tcl under $dir"
            continue
        }
    }
}

```

```

        set str [read $fd]
        set index [lsearch -regexp $str "$pkgName"]
        puts "index=$index"
        if {$index == -1} {
            close $fd
            continue
        }

        set status [catch { eval $str } res]
        close $fd
    }
}

% lappend auto_path E:/docs/adsl/nbe/script/ta/stub
=>D:/software/Tcl/lib/tcl8.3 D:/software/Tcl/lib D:/software/Tcl/lib/tk8.3
E:/docs/adsl/nbe/script/ta/stub
%package names
=> Tk ActiveTcl Tcl
% load_pkg AX4000_ETH_STUB
=>path=D:/software/Tcl/lib/tcl8.3
    no pkgIndex.tcl under D:/software/Tcl/lib/tcl8.3
    path=D:/software/Tcl/lib
    no pkgIndex.tcl under D:/software/Tcl/lib
    path=D:/software/Tcl/lib/tk8.3
    index=-1
    path=E:/docs/adsl/nbe/script/ta/stub
    index=78
% package names
=> AX4000_ETH_STUB Tk ActiveTcl Tcl

```

另外还可以通过动态链接库加载软件包，但要用 `load` 命令而非 `source` 命令。

12.1.5 用链接库提供软件包

有些脚本是通过链接库提供的。比如有些公司为了对用户屏蔽产品控制的细节，将其接口软件包存放在一个链接库文件（是二进制文件，Windows 中为 .dll 文件，UNIX 中为 .so 文件）中。而加载这些库中的软件包则需要使用 `load` 命令，而非 `source` 命令。`load` 的语法为：

[语法]: `load fileName`
`load fileName packageName`
`load fileName packageName interp`

如我们需要使用 `ax4klib.so` 中提供的 `ax4kpkg` 软件包，则可以使用如下命令：
`%load directoryName ax4klib.so ax4kpkg`

也可以使用 `pkg_mkIndex` 命令为库中的软件包创建自动加载索引文件 `pkgIndex.tcl`。如：
`%pkg_mkIndex -direct /usr/local/lib/ *.so`
 即为指定目录下的所有 .so 库文件创建索引。在 `pkgIndex.tcl` 里面，相应的 `source` 命令语句都被 `load` 命令语句代替。形式为：

```
package ifneeded $axPkgname 3.0 [list load $axPath ax4kpkg]
```

有关 load 命令的相关信息，请参考 Tcl/Tk 的命令手册或者帮助文件。

12.2 package 命令集

package 命令提供了为当前解释器维护可用的数据包的简单数据库的方法和使用数据库的手段。一般的，在 Tcl 脚本中，只需要包含 **package require** 和 **package provide** 两条命令即可，其他命令则供系统脚本使用以维护数据包的数据库。下表给出了有关 package 的命令。注意，有些命令是 pkg_mkIndex 过程和自动加载设施使用的。

表 12-2 package 命令

命令	说明
package forget <i>package</i>	删除已经加载的软件包信息
package ifneeded <i>package</i> ? <i>scripts</i> ?	表明可以提供特定版本的特定数据包。通过执行 <i>scripts</i> 可以将数据包加载到解释器中
package names	返回当前加载的软件包列表
package present ?- exact ? <i>package</i> ? <i>version</i> ?	和 package require 命令相似，不过当软件包没有加载时它不加载软件包，而只是返回软件包未加载信息，如果软件包已经加载，它返回软件包的版本。
package provide <i>package</i> <i>version</i>	声明提供相应版本的软件包
package require <i>package</i> ? <i>version</i> ? ?- exact ?	声明需要使用特定软件包，如果有 -exact，则加载指定版本的软件包
package unknown ? <i>command</i> ?	查询或设置用来定位软件包的命令
package vcompare <i>v1</i> <i>v2</i>	比较版本 <i>v1</i> 和 <i>v2</i> 。相同返回 0， <i>v1</i> > <i>v2</i> 返回 1，否则返回 -1
package versions <i>package</i>	返回已经加载软件包的版本
package vsatisfies <i>v1</i> <i>v2</i>	<i>v1</i> 大于等于 <i>v2</i> 而且还有相同的主版本号的话返回 1，否则返回 0

12.3 小结

略。

第 13 章. 文件操作与程序调用

13.1 文件操作

13.1.1 文件 I/O

Tcl 支持缓存机制的文件 I/O 操作。最简单的文件操作是 `gets` 和 `puts`，但当有大量数据需要读取时，`read` 命令更有效，可以通过 `read` 命令将整个文件数据都读出来，然后用 `split` 命令将文件按行进行分割。

本节将介绍几个用于文件操作的命令：`open`、`close`、`puts`、`gets`、`read`、`seek`、`tell`、`eof` 和 `flush`。表 13-1 列出了这些基本命令。

表 13-1 文件操作命令

命令	说明
open <i>fileName ?access? ?permission?</i>	打开文件或者管道，返回文件描述符 <code>fileID</code>
puts <i>?-nonewline? fileID str</i>	向文件描述符写入字符串。
gets <i>fileID varName</i>	读取一行字符，丢弃行换行符
close <i>fileID</i>	关闭文件，将缓存的内容 <code>flush</code> 出
read <i>?nonewline? fileID</i>	读取剩余的字节并返回字符串，如果设置了 <code>-nonewline</code> ，则丢弃最后的换行符
read <i>fileID numBytes</i>	读取 <code>numBytes</code> 指定个数的字节，返回字符串
seek <i>fileID offset ?origin?</i>	设置读写定位偏移量。如果操作权限是 <code>"a"</code> ，则不能将写偏移设置到文件结尾之前，但可以将偏移设置到文件开始用于读。 <code>origin</code> 可以是 <code>"start"</code> 、 <code>"current"</code> 或 <code>"end"</code> 。
tell <i>fileID</i>	返回访问指针偏移量（10 进制字符串）
flush <i>fileID</i>	输出通道缓存中的输出数据
eof <i>fileID</i>	检查文件结束。如果返回 1 表示到了文件结尾，否则返回 0

open 命令用于打开文件。返回一个可以供其他文件操作命令操作文件的文件描述符 `fileID`。

[语法]: `open fileName ?access? ?permission?`

说明：

1. `fileName` 是用于打开的文件名
 2. `access` 是文件存取模式，默认为读操作。表 13-2 列出了 `access` 变量取值和含义
- 表 13-2 `open` 命令的 `access` 变量说明

变量值	说明
<code>r</code>	打开文件用于只读。文件必须存在
<code>r+</code>	打开文件用于读和写。文件必须存在
<code>w</code>	打开文件用于只写。文件存在时则覆盖原来的内容，否则先创建文件
<code>w+</code>	打开文件用于写和读。文件存在时则覆盖原来的内容，否则先创建文件
<code>a</code>	打开文件用于写。新输入数据被追加到文件末尾
<code>a+</code>	打开文件用于读和写。新输入数据被追加到文件末尾

3. `permission` 是一个八进制整数，用于设置文件的访问权限，默认为 `rw-rw-rw(0666)`。UNIX 系统将文件用户分成三类：属主（`master`，文件的创建者）、组用户（`group users`）和其他用户（`other users`）。每类用户设置三位文件访问权限控制标识，分别指定了读、写和执行权限。如果对应位置为“-”，则表示此类用户没有对应的访问权限。在 UNIX 系统中，可以通过 `chmod` 命令来更改文件的访问控制权限。

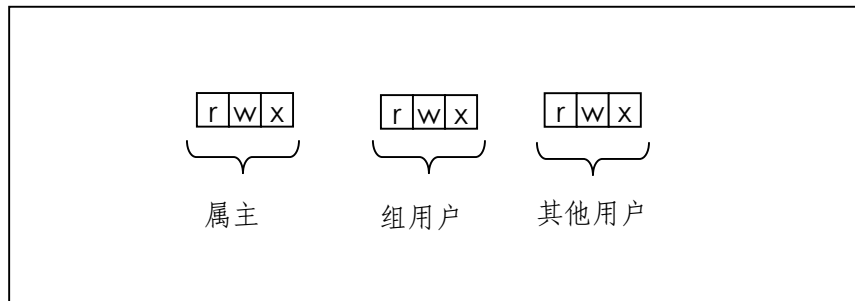


图 7 文件访问控制权限说明

4. 在使用 `open` 命令打开文件的时候，应该使用 `catch` 命令来捕获错误信息。这样会使代码更安全。当调用成功时，文件描述符被保存到 `catch` 的变量中，否则 `catch` 变量保存错误信息。

例 13-1 用 `open` 命令打开文件，并输入数据

```
%if [catch {open ~/data.tcl w+} res] {
    puts "Cannot open ~/data.tcl for write:$res"
} else {
    puts $res "This is one data file."
    flush $res
    close $res
}
% more ~/data.tcl
=>This is one data file.
```

使用这些文件命令时需要注意：

- 对 `gets` 命令，无法区分空行和文件结束 EOF，所以在使用此命令读文件时，需要用 `eof` 命令来判断文件 EOF
- 在 TCL 中，所有的数据都以 ASCII 字符串形式保存，这意味着当读取一个二进制文件时可能产生不可预料的结果。
- 文件 I/O 是有缓存机制的，当调用 `close` 命令后，会关闭指定文件并将相关通道缓存内容输出到文件。但是如果是非正常关闭，留在缓存中的数据可能无法保存到文件而丢失。
- `puts`、`gets`、`seek` 等命令中的变量 `fileID` 可以是如下几种：
 - a) 由 `open` 命令返回的文件描述符
 - b) 标准输入 `stdin`
 - c) 标准输出 `stdout`
 - d) 标准错误输出 `stderr`

13.1.2 文件系统信息命令

主要有两个命令用来提供关于文件系统的信息，`glob` 和 `file`。

13.1.2.1 glob 命令

glob 命令和 UNIX 系统的 ls 命令相似，用于文件的匹配搜索，并返回一个与搜索模式匹配的文件名列表。glob 支持通配符。

[语法]: **glob ?switches? pattern ?patternN?**

switches 选项有：

- -nocomplain 当返回空列表时，glob 不报错，不用此选项时，glob 会在返回空列表时报错
- -directory directory 在指定的目录中搜索。如 glob -directory e:\masm e1.tcl
- -path pathVar 在指定路径内搜索。不可以和-directory 同时使用。
- -- 结束 switches

glob 的匹配模式与 string match 命令的匹配规则相似：

- "*" 通配 0 或多个字符；
- {a,b,...} 匹配 a,b,... 中的任一个字符；
- "?" 通配单个字符；
- [abc] 匹配一组字符；
- 如果 pattern 开始两个字符是~/，则~将被用户路径环境变量值 HOME 替代，如果文件是以~开始，最好加一个./前导来避免这种扩展（如./~foo）。

13.1.2.2 file 命令集

Tcl 提供了一组 file 命令来检测文件系统中的文件信息和状态。表 13-3 列出了主要 file 命令及其选项。

表 13-3 file 命令集

命令	说明
file atime name	返回十进制字符串形式的最后一次访问(access)时间
file attributes name ? option ? ?value? ...	查询或设置文件属性
file copy ?-force? source destination	拷贝文件或目录
file delete ?-force? name	删除文件
file dirname name	返回文件所在目录
file executable name	测试文件可执行属性，可执行返回 1，否则返回 0
file exists name	测试文件是否存在，存在返回 1，否则返回 0
file extension name	返回文件扩展名
file isdirectory name	测试文件是否为目录，是返回 1，否则返回 0
file isfile name	如果 name 不是目录、符号连接和设备文件的话，返回 1，否则返回 0（测试是否为普通文件）
file join path path ...	将路径名的各部分连接起来形成一个新路径
file lstat name var	将链接（link）名的属性读入 var
file mkdir name	创建目录
file mtime name	返回文件最后一次修改时间（从 1970 年 1 月 1 日开始到被修改一刻经过的以十进制数表示的秒数）
file nativename name	返回 name 的本机平台版本
file owned name	测试是否为文件 master，如果是则返回 1，否则返回 0
file pathtype name	测试路径类别，为 relative、absolute 或 driverelative
file readable name	测试文件的可读权限，有则返回 1，否则返回 0

file readlink name	返回通过符号链接指向文件名。如果 name 不是链接标识或者不可读则返回错误。注意和硬链接区别。
file rename ? -force ? old new	更名
file rootname name	返回不带扩展名的文件名
file size name	返回文件字节数
file split name	将 name 分解成各个路径组成部分
file stat name var	将文件的属性读出，存入数组 var 中。
file tail name	返回最后一个路径组成部分
file type name	返回类型标识，有 - file: 普通文件 - directory: 目录 - characterSpecial: 面向字符的设备 - blockSpecial: 面向块的设备 - fifo: 有名管道 - link: 符号链接 - socket: 有名套接字
file writable name	如果有可写权限则返回 1，否则返回 0

lstat 和 stat 返回一组文件属性信息，并以数组形式保存。如果文件是一个符号链接，则 lstat 返回链接本身信息，stat 返回链接目标信息。符号连接(symbol link)只将链接指向目标文件存储域，而硬链接将目标文件内容也拷贝到自己文件内，相当也作了一次拷贝。

例 13-2 lstat 和 stat 命令举例

```

;#STEP1.在 UNIX 用户目录下创建一个文件，如 a.tcl
;#STEP2.用 ln 命令为 a.tcl 创建一个符号连接 "$ln -s a.tcl la": 文件 la 为 a.tcl 的符号链接
;#STEP3.用 lstat 和 stat 来获取 la 的属性:
% file lstat la var1
% file stat la var2
% parray var1
=> var1(ctime) = 1069660652
    var1(dev) = 61865988
    var1(gid) = 500
    var1(ino) = 2922500
    var1(mode) = 41471
    var1(mtime) = 1069660652
    var1(nlink) = 1
    var1(size) = 5
    var1(type) = link
    var1(uid) = 513
% parray var2
=> var2(ctime) = 1069118682
    var2(dev) = 61865988
    var2(gid) = 500
    var2(ino) = 2922505
    var2(mode) = 33261
    var2(mtime) = 1069119358
    var2(nlink) = 2
    var2(size) = 589

```

```
var2(type) = file
var2(uid) = 513
```

从例子可以看到，数组的元素除了 `type` 之外都是整数。表 13-4列出了这两个命令的返回数组内容。

表 13-4 file stat 命令数组元素

元素	说明
atime	最后一次访问时间，以秒计算，从 1970 年 1 月 1 日 0 时算起
ctime	最后一次属性修改时间
dev	设备标识符
gid	属组
ino	文件编号（即 i 节点号）
mode	权限位
mtime	最后一次修改时间
nlink	文件链接或目录引用记数
size	字节数
type	类型：file、directory、characterSpecial、blockSpecial、fifo、link 或 socket
uid	属主的用户 ID

13.2 程序调用

到现在为止，我们介绍的都是如何在 Tcl 解释器内部进行编程，其实 Tcl 是一种通用的粘连脚本语言，能其他的程序和软件包按照求组装在一起。为了完成这种功能，Tcl 提供了一些调用其他程序的方法。主要有两中方法在 Tcl 中调用一个程序：

- **open**.....打开一个连接到文件描述符的进程管道来运行其他程序
- **exec**将一个程序作为子进程运行

这些命令以前是为 UNIX 系统设计的，Tcl7.5 以后的 Windows 版本也可以实现他们。为了保证操作正确性，建议本章内 `exec` 命令和 `open` 命令的程序在 UNIX 系统内完成。

13.2.1 用 open 命令打开一个进程管道

此处的 `open` 调用和打开一个文件的命令相同，也返回一个文件描述符。如果文件名变量的第一个字符是 “|” 的话，`open` 命令则将此变量剩余部分当作是一个程序名，并将运行 `exec` 来处理，而输入、输出被重定向到了文件描述符。管道则可被子进程打开用以只读、只写或者读写。

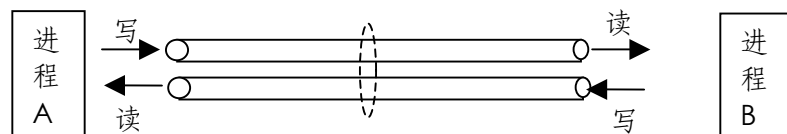


图 8 进程管道

如果一个文件被作为管道打开用于读和写，你必须清楚这个管道会被缓存的。`puts` 命令的输出将被缓存到一个 I/O 缓存中直到缓存被填满，或者执行 `flush` 命令强制其传输给子进程。而在子进程写满管道之前，子进程的输出也不会被 `read` 或 `gets` 命令获取，默认情况下 `read` 和 `gets` 命令将被阻塞以等待管道被子进程填满。

[语法] open |progName ?access?

progName 用双引号括起，可以包含变量，但开始一定是“|”。**access** 表明操作选择，有“r”、“w”等，可参见上节的 access 表。

例 13-3 用 open 命令打开只读进程管道

```
%set fd [open "| sort /etc/passwd" r]      ;#sort 命令的标准输出被重定向到文件描述符
$fd
=>file4
%set str [split [read $fd] \n]             ;#读取管道内的全部信息，并以换行符分割
=>adm:x:4:4:Admin:/var/adm: bin:x:2:2:/usr/bin: daemon:x:1:1::: {listen:x:37:4:Network
Admin:/usr/net/nls:} {lp:x:71:8:Line Printer Admin:/usr/spool/lp:}
{noaccess:x:60002:60002:No Access User:/:} {nobody4:x:65534:65534:SunOS 4.x
Nobody:/:} nobody:x:60001:60001:Nobody:/: {nuucp:x:9:9:uucp
Admin:/var/spool/uucppublic:/usr/lib/uucp/uucico} root:x:0:1:Super-User:/:/sbin/sh
sys:x:3:3:/: {uucp:x:5:5:uucp Admin:/usr/lib/uucp:} {}
%puts $str
=>adm:x:4:4:Admin:/var/adm: bin:x:2:2:/usr/bin: daemon:x:1:1::: {listen:x:37:4:Network
Admin:/usr/net/nls:} {lp:x:71:8:Line Printer Admin:/usr/spool/lp:}
{noaccess:x:60002:60002:No Access User:/:} {nobody4:x:65534:65534:SunOS 4.x
Nobody:/:} nobody:x:60001:60001:Nobody:/: {nuucp:x:9:9:uucp
Admin:/var/spool/uucppublic:/usr/lib/uucp/uucico} root:x:0:1:Super-User:/:/sbin/sh
sys:x:3:3:/: {uucp:x:5:5:uucp Admin:/usr/lib/uucp:} {}
%close $fd
```

进程管道创建好之后，就可以象操作文件那样对其进行读写操作。

下面的例子演示了另外一个进程管道的操作。首先编写一个.tcl 文件，它的代码是一个 while 无限循环，将从标准输入得到的字符串转换为大写字符串后输出到标准输出，当检测到输入是“end”后向标准输出输出跳出循环提示。open 命令将进程管道的标准输出和输入都重定向到文件描述符 io，这样我们就可以通过读写文件描述符和循环进程交互。注意进程管道读写是缓存的，输入较少的字符时，需要用 flush 命令输出缓存的内容。

例 13-4 用 open 命令打开进程管道又一例

```
;%STEP1.编写 a.tcl:
% set fd [open "~/a.tcl" w+]
=> file4
% puts $fd {
    while 1 {

        gets stdin line
        if [string match $line end] {
            puts stdout "EOF was detected,please close channel"      ;#检测到"end", 则
终止循环
            break
        }

        set upstr [string toupper $line]
        puts stdout $upstr
        flush stdout
    }
}
% flush $fd
% close $fd
```

```

;#STEP2. 创建进程管道，并测试管道读写
% set io [open "| wish a.tcl" r+]
=>file6
% puts $io helloworld
% flush $io
% gets $io
=>HELLOWORLD
% puts $io end           ;#输入 end 让进程跳出循环
% flush $io
% gets $io
=>EOF was detected,please close channel
%close $io

```

为了不至于 gets 命令被阻塞，可以先用 fconfigure 命令将进程管道设置为非阻塞的：

```
%fconfigure $io -blocking 0
```

这样，gets 命令在进程管道内没有输出数据的情况下，不等待而直接返回。

13.2.2 用 exec 命令调用程序

exec 命令从 Tcl 解释器中运行其它进程。

如下面通过执行 unix shell 的 date 命令来获得当前日期：

```

% exec date
=>Mon Nov 24 10:16:11 CST 2003

```

exec 命令和从 unix shell 命令行或者 unix shell 脚本中调用一个进程相似。它支持命令的输入、输出重定向，支持管道操作等。可将被调用程序的输出作为 exec 命令的值予以返回。

对于学习过 unix shell 中标准输入、输出的读者一定知道，每个进程通常有三个与之相关联 I/O 管道：标准输入、标准输出和标准错误输出。通过 I/O 重定向，可以更改这三个管道的传送方向，比如可以将进程的标准输出定位到一个文件，则进程的所有标准输出信息都会保存到那个文件内。而通过管道可以组织一个进程链：一个程序的标准输出被通过管道挂接到另外一个程序的标准输入，这样管道可以将任意数目的程序连接起来一起执行。

例 13-5 用 exec 处理管道与 I/O 重定向

```

set n [exec sort < /etc/passwd | wc -lw 2>~/errorInfo.tcl]
=> 12 21

```

例 9-1 使用 exec 来执行两个程序，第一个是 sort，它的标准输入被用输入重定向符"<"定向到了文件/etc/passwd，该命令从此文件中直接接收输入，对文件进行排序。sort 的标准输出通过管道（用"|"表示）传给了程序 wc，wc 来计算文件的行数和单词数（通过选项-lw 指定）。wc 的标准错误输出又被通过输出重定向符"2>"输出到用户目录下的 errorInfo.tcl 文件中。

[语法] exec ?switches? arg1 ?arg2? ... ?argN?

说明：

1. 如果 exec 的开始的参数是以"-开始，则这些参数被认为是命令的开关选项 switches。开关选项有：

□ -keepnewline 不要丢弃管道输出结果中尾部的换行符。通常换行符会被删除

- -- 标识开关选项的结束。下一个字符串将被认为是参数 `arg1`，即使它以“-”开始也不再被认为是开关选项
- 2. `arg1~argN` 可以是
 - 被执行程序名
 - 被执行程序的命令行参数
 - I/O 重定向指示

重定向指示有多种，表 13-5 概括了部分 `exec` 能识别的重定向指示标识。

表 13-5 重定向指示标识和说明

标识	说明
	将管道标识符之前的进程的标准输出作为跟随管道标识符后进程的标准输入
&	使标准输出和标准错误输出同时作为管道输出
<fileName	管道的第一个程序从指定文件读取输入
<@fileID	管道的第一个程序从指定文件描述符 <code>fileID</code> 中读取输入。 <code>fileID</code> 的值是命令 <code>open ... "r"</code> 的返回值
<<value	管道的第一个程序从 <code>value</code> 接收输入
>fileName	管道中最后一个程序的标准输出被保存到 <code>fileName</code> 指定的文件。文件中原来的内容将被覆盖
>>fileName	管道中最后一个程序的输出被追加到 <code>fileName</code> 中， <code>fileName</code> 原来的内容不会被覆盖
2>fileName	管道中所有程序的标准错误输出都被送到 <code>fileName</code> 中， <code>fileName</code> 原来的内容被覆盖
2>>fileName	管道中所有程序的标准错误输出都被追加到 <code>fileName</code> 中， <code>fileName</code> 原来的内容不会被覆盖
>&fileName	同时用标准错误输出和标准输出重写 <code>fileName</code>
>>&fileName	同时将标准错误输出和标准输出追加到 <code>fileName</code>
>@fileID	管道中最后一个程序的标准输出被定向到 <code>fileID</code> 通道
2>@fileID	将标准错误输出 定向到 <code>fileID</code> 通道
>&@fileID	同时将标准错误输出和标准输出定向到 <code>fileID</code> 通道
&	作为最后一个变量，表示将管道放到后台执行。 <code>exec</code> 命令返回进程标识

如果没有用 `&` 来将 `exec` 命令放到后台执行，则该命令在执行期间将被阻塞。

13.2.3 pid 命令

`pid` 命令返回当前进程的 ID。进程 ID 在每次进程调用时的值都会改变，所以可以用进程 ID 作为随机数生成的种子(seed)。也可以通过 `pid` 来查出与进程管道相关联的进程 ID

例 13-6 用 `pid` 命令检查进程 ID

```
%set fd [open "| sort /etc/passwd" r]      ;#sort 命令的标准输出被重定向到文件描述符
$fd
=>file5
%pid $fd                                   ;#与进程管道$fd 相关联的进程的进程号
=>23368
% pid
=>23318                                   ;#当前进程的进程号
```


第 14 章. 套接字与事件驱动编程简介

本章将对使用套接字进行网络客户与服务器编程进行简单的介绍, 然后再简单介绍对事件驱动编程相关的几个命令。在讲述套接字编程的时候, 可能会先用到有关事件驱动的命令, 读者可以参考事件驱动编程部分的介绍。

14.1 套接字编程

套接字(socket)是网络协议实体通信通道, 目前Tcl只支持基于TCP协议的套接字编程。**socket**命令可用来打开TCP连接通道, 并返回这个通道的标识符。这个通道标识符就象open命令返回的文件标识符一样可以进行诸如read、gets和puts、flush等文件传输操作, 还可以用close命令将其关闭。

大家知道, C/S 网络服务模型需要客户端和服务端。服务端是一个长期运行(循环)的资源维护和客户端命令处理与响应进程。如HTTP服务器可以在Internet上提供对网页的访问, FTP服务器可以响应ftp登录请求并处理文件存取请求等。而客户端则需要通过服务器实现对资源的有限访问。C/S编程需要对Client和Server端的程序分别编程。本节通过一个简单的C/S例子介绍一下C/S模型如何工作与编程。

[注]: scotty扩展模块可以支持许多网络协议, 如TCP/UDP、Tnm、RPC等。关于scotty模块Tnm接口的介绍, 大家可以参见《自动测试平台——核心与接口(SNMP代理部分)》。在自动测试小组的服务器172.24.213.247上安装了scotty模块, 通过模块的命令手册大家可以对其进行更深入的研究。更多的内容则可以访问scotty的主页:

<http://wwwsnmp.cs.utwente.nl/~schoenw/scotty/>

14.1.1 socket 命令

socket命令用来打开一个TCP网络连接, 命令格式为:

[语法]: (1) **socket ?options? host port**
(2) **socket -server command ?options? port**

第一个命令没有-server选项, 用于client端套接字操作, 第二个命令用于server套接字操作。

14.1.1.1 Client 端 socket 命令

Client端socket命令是“socket ?option? host port”。其中的选项option可以有如下几个:

- **myaddr addr** Addr是客户端的网络主机名或者IP地址。这个参数对有多个网络接口的客户端比较有用。如果不用这个选项, 则客户端的IP地址就由系统软件决定。
- **myport port** Port是客户端用来标识连接的TCP接口号, 如果不指定, 则此端口号由系统随机生成。
- **-async** 此选项表示客户端套接字异步进行连接。这意味着调用socket命令会立即返回, 套接字会被创建但不会立即与server连接。打开到服务器的连接可能会花费很长时间, 如果不用本选项, 则socket在执行的时候会被阻塞直到连接完成或失败。如果使用了本选项则socket命令不会被阻塞, 建立连接的过程会在后台进行。如果在连接建立之前就进行读写操作, 且套接字处于阻塞模式, 则操作会被阻塞。

下面的代码试图连接到一个无法访问server的时候, 未使用-async被阻塞直到连接失败:

```
% socket www.sina.com.cn 8080
=>couldn't open socket: connection timed out
```

host是服务器的主机名或者IP地址。

`port` 是服务器端的 TCP 端口号。
命令成功后返回通道标识符。

14.1.1.2 Server 端 socket 命令选项

Server 端的 socket 命令是 “**socket -server command ?options? port**”。

-server 表明是创建服务器套接字，接口为 `port`。此命令为 server 创建一个侦听套接字。Tcl 会自动接受所有客户端连接到本端口的连接(connect)请求，并为每个请求创建单独通信套接字。

在建立连接通道的时候，会自动调用 `command` 命令，并将三个额外参数传递给 `command`: 新的通道标识号、客户端主机名或 IP 地址、客户端端口号。

socket 命令成功时返回新通道的标识符。

options 为额外选项，有：

- **myaddr addr** server 端的网络主机名或者 IP 地址，对有多个网络端口的 server 比较有用。

`port` 为服务 TCP 端口号。

服务器侦听套接字通道不能被用来进行读写操作，它只用来接受新的客户端连接请求。当接收到 client 的连接请求时（在 client 主机上用 socket 命令），server 就会给此 client 创建新的套接字通道。而这些新的套接字可以进行读写操作。当侦听套接字通道被关闭(用 close 命令)时，server 就不能再接收新的连接请求，但先前已经建立的套接字不受影响。所以要断开与某一客户端的连接，就要单独进行一次关闭与客户端的那条套接字通道。

14.1.2 用 fconfigure 配置套接字

可以用 `fconfigure` 来配置套接字通道或者获取其信息。用 `fconfigure` 命令配置套接字通道的选项有：

- `-error` : 获取当前套接字的错误状态。这对判断一个异步套接字连接是否成功比较有用。
- `-socketname` : 获取指定套接字的信息，如主机 IP 地址、主机名和端口号等。
- `-peername` : server 端的套接字不支持本选项。对客户端返回对端主机名、IP 地址、端口号等。
- 其它 `fconfigure` 选项：是 `fconfigure` 的普通选项。如：
 - `-blocking 0/1` : 设置通道阻塞模式，0 为非阻塞，1 为阻塞
 - `-buffering newValue`: 设置通道缓存大小，如 `newValue` 可以为 `line` 或 `full` 等
 - 其它选项

详细的 `fconfigure` 选项见下面事件编程部分的介绍。

14.1.3 C/S 编程举例

本节设计了一个 C/S 通信模型。服务器端对每个客户端的连接请求进行处理，为每个请求建立一个新的套接字连接并将套接字连接标识符保存到全局数组变量 `sock_arr` 中。服务器侦听和每个客户端的连接，当有数据输入时，从连接缓存中读取并分析数据，如果接收的是 “quit”，则关闭与对应客户端的连接。如果接收的是 “end”，则将服务器套接字（侦听）删除，从而关闭服务器，否则返回相关信息。

客户端操作更为简单，只创建套接字连接，然后对套接字通道进行读、写操作。

服务器的代码最好单独放在一个.tcl 脚本文件中。

为了便于操作，server 和 client 运行在同一主机上，本脚本是在 windows 上测试的，主机地址就用 127.0.0.1 的环回地址。你当然可以将两者运行在不同主机上，这样会贴近实际应用环境。

运行的时候，最好打开两个 `tclsh` 或者 `wish`。一个运行 server，另一个用来运行 client。

例 14-1 基于 socket 实现的 C/S(Client/Server)简单模型

```

;#(1) server.tcl
;#-----START server.tcl-----
array set sock_arr ""
set SERVER_PORT 2540

proc Accept { newSock addr port } {
    global sock_arr
    puts "Accepted $newSock from $addr port $port"
    set sock_arr(addr,$newSock) [list $addr $port]
    puts "Now there are [expr [array size sock_arr] -1 ] client connects:\n"
    parray sock_arr

    fconfigure $newSock -blocking 0      ;#设置通道为非阻塞
    fconfigure $newSock -buffering line  ;#设置 buffering
    fileevent $newSock readable [list Echo $newSock] ;#设置事件驱动命令
}

proc Echo {sock} {
    if { [eof $sock] || [catch {gets $sock line}] } {
        return
    }

    ;#如果接收是 quit，则关闭对应通道套接字
    if {[string compare $line "quit"] == 0} {
        puts "Close $sock_arr(addr,$sock)"
        close $sock
        unset sock_arr(addr,$sock)
    }

    ;#如果接收是 end，则关闭 server 的套接字
    if {[string compare $line "end"] == 0} {
        if {[array size sock_arr] > 1} {
            puts "Still other clients using this server.Server's socket cannot be deleted."
        } else {
            puts "Close server's socket $sock_arr(main)"
            close $sock_arr(main)
            unset sock_arr(main)
        }
    }
}

puts stdout $line
puts $sock [string toupper $line]      ;#将客户端传来的字符串变为大写字符后回传
flush $sock
}

;#创建 server 套接字（用于侦听）
set status [catch { socket -server Accept $SERVER_PORT } ss]
if { !$status } {
    set sock_arr(main) $ss
    puts "Create server socket success. Server's socket is $ss"
} else {
    error "Create server's socket failed: $res"
}

```

```
}  
;#-----END server.tcl-----
```

```
;(2) client.tcl  
;#-----START client.tcl-----  
array set c_arr ""  
set SERVER_IP 127.0.0.1  
set SERVER_PORT 2540  
  
proc Create_Client { name } {  
    global c_arr SERVER_IP SERVER_PORT  
    set status [catch { socket $SERVER_IP $SERVER_PORT } res]  
    if { !$status } {  
        puts "Create client socket success. Client socket is $res"  
        set c_arr($name) $res  
        fconfigure $res -buffering line -blocking 0  
    } else {  
        error "Create client socket failed: $res"  
    }  
}  
  
proc Close_Client { name } {  
    global c_arr  
    if [info exists c_arr($name)] {  
        catch {close $c_arr($name)} res  
        unset c_arr($name)  
    }  
}  
;#-----END client.tcl-----
```

;(3) 打开两个 wish，如下图图 9. 下面 .S. 表示在 Server 界面内操作或显示，.C. 表示在 Client 界面操作或显示。

```
.S. %source server.tcl  
=>Create server socket success. Server's socket is sock172  
.C. %source client.tcl  
.C.% Create_Client c1  
=>Create client socket success. Client socket is sock172  
    sock172  
.S. =>Accepted sock204 from 127.0.0.1 port 1423  
    Now there are 1 client connects:  
  
        sock_arr(addr,sock204) = 127.0.0.1 1423  
        sock_arr(main)        = sock172  
.C.% set c1 $c_arr(c1)  
=>sock172  
.C.% puts $c1 "Hello,I'm Client c1. Please echo me."  
.C.% flush $c1  
.S.=> Hello,I'm Client c1. Please echo me.  
.C.% set res [gets $c1]  
=> HELLO,I'M CLIENT C1. PLEASE ECHO ME.
```

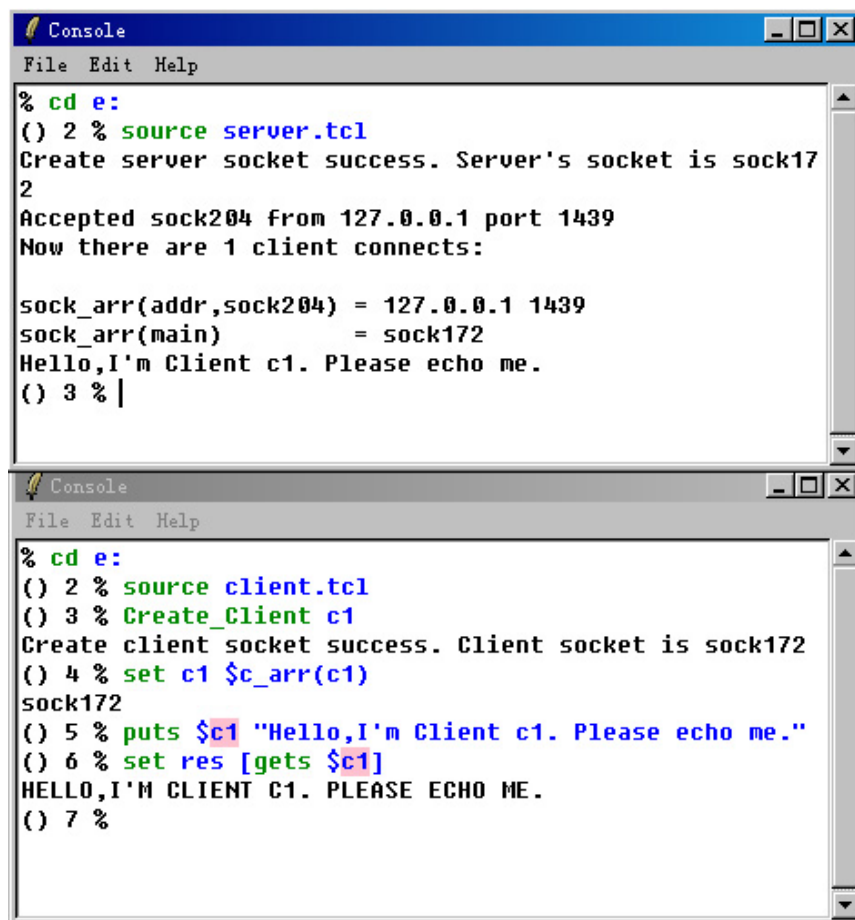


图 9 C/S 界面

14.2 事件驱动编程

事件驱动编程应用在象网络服务器和图形界面等待回应程序中。本节主要针对 Tcl 中的事件驱动编程。对于 Tk 中的图形用户界面事件编程不作介绍。

Tcl 提供了一种简便的事件驱动模型。当用这种模型注册了 Tcl 命令后，系统就会在发生了特定事件时调用注册命令。比如使用 `after` 命令可以注册一个命令，让它在指定经过多少时间后被调用；而使用 `fileevent` 命令可以为 I/O 接口如用 `open` 命令打开的文件 I/O、用 `socket` 命令注册的套接字等设置特定命令，这个特定命令会在 I/O 发生特定变化时被调用；`vwait` 则用来等待事件的发生，在等待期间 Tcl 会自动调用与不同事件关联的 Tcl 命令。

14.2.1 after 命令

`after` 命令用来设置在一段时间延迟之后执行的命令。各条 `after` 命令的语法在下表中列出：

表 14-1 after 命令

命令	说明
after <i>ms</i>	等待 <i>ms</i> 时间。 <i>ms</i> 单位是 millisecond 毫秒，下同。
after <i>ms</i> ? <i>script</i> <i>script</i> <i>script</i> ...?	将各 <i>script</i> 连接成一条命令并在 <i>ms</i> 毫秒后执行。连接 <i>script</i> 的处理方式同 <code>concat</code> 命令，这与 <code>eval</code> 命令相同。

	要注意 <code>concat</code> 和 <code>list</code> 的不同之处，每次遇到能将 <i>script</i> 连接成一条命令字符串的命令，最好用 <code>list</code> 命令来显式处理。 <code>after</code> 命令成功后，返回一个标识符，可以供 after cancel 使用。
after cancel <i>id</i>	将原来安排的延迟执行的命令取消。 <i>id</i> 为上条 <code>after</code> 命令返回的标识符，标识是那个命令要被取消。如果此时命令已经在执行，本命令则不起作用。
after cancel <i>script script script ...</i>	这取消原延迟执行命令。
after idle ? <i>script script script ...?</i>	用空格分割符将各 <i>script</i> 连成一条命令，等下次空闲的时刻执行。 <code>after</code> 命令返回命令标识符。
after info ? <i>id?</i>	返回待执行事件的信息。如果没有 <i>id</i> 参数，则返回所有命令句柄列表。

最简单的 `after` 命令应用是让程序暂停一段时间（毫秒），如下面的代码会等待 3 秒，在此期间程序被阻塞：

```
after 3000
```

`after` 命令也可以注册一条延迟执行命令，它会将多条变元 *script* 连接成一条命令。如果命令结构比较重要，最好使用 `list` 来显式创建命令而不要 `after` 用 `concat` 构建命令。下面的代码会在 3 秒后打印一条语句：

```
% set hd [after 3000 [list puts "This is one delay command."]]
=>after#0                ;#返回的命令标识符
This is one delay command. ;#等待 3 秒后输出
```

上面 `after` 的返回值是注册命令的标识符，可以用 `after cancel` 命令来取消注册的命令。这条命令不会阻塞当前程序。

`after idle` 命令用来注册一条在下次空闲时刻执行的命令。

举例：一个命令循环处理或者批处理操作，当按 `Run` 按钮时循环处理各条命令，拥护按 `STOP` 按钮时停止循环过程。试想如何用 `TCL/TK` 来设计这个过程？

如果仅仅用一个 `for` 或者 `while` 循环处理程序直接一条一条命令的循环处理，则 `Tcl/Tk` 程序会被阻塞在这个循环体中，直到命令都处理完，无法响应 `STOP` 按钮。如果能将任务按照时间进行划分，一段时间执行一条命令，然后再空出一段时间等待用户操作，这样问题就好处理了。这可以用 `after` 命令来实现。

下面例子中的程序就利用 `after` 命令实现了一种命令循环处理控制方法。`Run` 和 `Stop` 过程分别是 `RUN` 和 `STOP` 按钮的处理命令。全局变量 `switcher` 作为控制开关由 `Circle` 过程用于命令循环处理控制，当 `switcher` 为 `"true"` 时，处理下条命令，为 `"false"` 时，终止 `Circle` 过程。`Circle` 过程处理完一条命令后，用 `after` 命令注册延时 1 秒后运行自身，从而达到分时等待-处理命令的效果。`Repeat` 设置循环执行命令功能。`Exec_Cmd` 过程仅仅完成输出一条指定信息到标准输出。全局变量 `cmd_l` 中存放了需要输出的字符串列表，`pos` 指明了当前命令的位置。`button` 命令创建按钮，`pack` 命令将命令显示到界面上。这个程序需要使用 `wish` 来运行。

例 14-2 `after` 命令的简单应用

```
#!/bin/sh
#\
exec wish8.3 $0 ${1+"$@"}

button .run -text "RUN" -command [list Run]
button .stop -text "STOP" -command [list Stop]
```

```

button .repeat -text "REPEAT" -command [list Repeat]
pack .run .repeat .stop -side left -expand true

set cmd_l {1 2 3 4 5 6 7 8 9 10}
set pos 0
set max 10
set switcher off
set circle off
proc Run {} {
    global switcher pos circle
    if {$switcher == "on"} {
        error "Run task now, Try again later."
        return
    }
    set switcher on
    set circle off
    set pos 0
    Circle
}
proc Stop {} {
    global switcher circle pos
    set switcher off
    set circle off
    set pos 0
}

proc Circle {} {
    global switcher circle pos cmd_l max

    if { $switcher == "off" } {
        puts "Swither is off.Execution cancelled by user."
        return
    }

    if {$circle == "off" && $pos >= $max} {
        puts "Finish commands, no circle."
        set switcher off
        set pos 0
        return
    }

    if {$circle == "on" && $pos >= $max} {
        set pos 0
    }

    set str "Now run [lindex $cmd_l $pos]"
    set cmd [list puts $str]
    #You can replace Exec_Cmd by any other command
    Exec_Cmd $cmd
    incr pos

    after 1000 Circle
}

```

```

proc Exec_Cmd {cmd} {
    catch {uplevel #0 $cmd} err
}
proc Repeat {} {
    global switcher circle pos
    if {$switcher == "on"} {
        error "Run task now.Try again later."
        return
    }

    set switcher on
    set circle on
    set pos 0

    Circle
}

```

14.2.2 fileevent 命令

fileevent 命令为 I/O 通道注册一条命令，当通道变为可读或可写的时候该命令被执行。在套接字编程的例子里，我们已经使用了该命令为服务器端的套接字通道的注册了 Echo 命令，供套接字通道可读时执行。fileevent 命令格式为：

[语法]: fileevent channelId readable ?script? ;#注册 channelId 可读时的命令脚本
fileevent channelId writable ?script? ;#注册 channelId 可写时的命令脚本

当一个服务程序用 **gets** 或者 **read** 来读取一个阻塞通道的内容时，如果通道中没有数据，程序就会被阻塞在这个通道上，直到通道有可读数据到达。这样该程序在阻塞期间就不能为其他客户提供服务或处理别的事情，在其它客户看来该服务程序好象被“冻起来”。使用 **fileevent** 命令后，服务程序就可以在通道上有数据可读时才调用 **gets** 和 **read** 读取数据，从而避免被阻塞在该通道上。

channelId 是打开通道的标识符，象前面的 **open** 或 **socket** 命令返回值。当提供 *script* 时，**fileevent** 命令会返回事件句柄，*script* 也会在通道变为可读或者可写时被执行（这依赖于 **fileevent** 命令的第二个参数）。

对于一个 I/O 通道，至多有一个可读处理程序和可写处理程序，如果已经用 **fileevent** 命令注册了一个处理程序之后再次注册一个的话，原来的注册程序就会被覆盖。如果 **fileevent** 命令中没有 *script* 参数，则命令返回当前已经注册的命令，若没有注册命令则返回空字符串。

另外需要注意，用 fileevent 注册的命令只有在通道关闭的时候才被注销，而没有象 after delete 那样的 fileevent 命令来显式注销命令，所以及时关闭通道非常重要！

14.2.3 vwait 命令

vwait 命令用来等待某一变量被设置。其语法格式为：

[语法]: vwait varName

使用 **vwait** 将是 Tcl 程序进入到事件循环，在等待期间，程序会被阻塞。当变量 *varName* 被其他事件句柄设置时，**vwait** 返回，程序得以继续执行下面的步骤。这里的 *varName* 应该是一个全局变量。

下面例子中用 `fileevent` 为 `stdin` 设置了一个 `readable` 命令 `Read`，并用 `vwait` 等待变量 `x` 被改变。在 `Read` 内将读取的字符串变为大写后回显，如果是 “change” 则设置 `x` 值为 1。注意：请在 `tclsh` 上运行本例，在 `wish` 上看不出效果。

例 14-3vwait 命令的简单例子

```

;#(1)将脚本保存到 e:/test.tcl 内
-----START SCRIPT-----
set x ""

proc Read {} {
    global x
    set str [gets stdin]
    puts stdout [string toupper $str]
    if {[string compare $str change]} {
        set x 1
    }
}

fileevent stdin readable [list Read]

vwait x
exit
-----END SCRIPT-----

;#(2)执行脚本，用 tclsh
% cd e:
%source test.tcl
hello                ;#请输入
=>HELLO
change                ;#请输入
=>CHANGE              ;#输入为 change 时，vwait 返回
%

```

14.2.4 fconfigure 命令

14.2.4.1 fconfigure 语法

`fconfigure` 命令用来设置或者查询 I/O 通道的属性。通道的默认设置对大多数情况来说都是适用的。如果你是执行事件驱动的 I/O，则可能想将其设置为非阻塞模式；如果你是处理二进制数据，可能想关闭行结束符与字符集转换。`fconfigure` 的命令格式为：

[语法]： (1) `fconfigure channelId`
 (2) `fconfigure channelId name`
 (3) `fconfigure channelId name value ?name value ...?`

`channelId` 为 I/O 通道标识符，为 `open` 或 `socket` 命令返回的标识符。`name` 为属性名，`value` 为属性值。如果象 (1) 只提供 `channelId`，则返回通道的当前属性设置。如果是形式 (2)，只有一个属性名，则返回对应属性值。用形式 (3) 可进行属性设置。

如下面的命令返回 `stdin` 的属性：

```
%fconfigure stdin
=> -blocking 1 -buffering none -bufferSize 4096 -encoding utf-8 -eofchar {} -
translation lf
```

下表给出了 fconfigure 所控制的属性及说明。

表 14-2 fconfigure 控制的 I/O 通道属性

属性	说明
-blocking <i>boolean</i>	设置 I/O 通道阻塞模式：0 为非阻塞，1 为阻塞
-buffering <i>newValue</i>	设置缓冲模式：none、line 或 full。如果是 full，则 I/O 将缓冲输出数据直到 buffer 变满或者调用 flush 输出缓冲数据；如果是 line，则遇到换行符时自动调用 flush 输出缓冲数据；如果是 none，则每个输出操作的结果会被立即输出。
-bufferSize <i>newSize</i>	设置缓冲大小（字节）。newSize 为整数，范围为 10~100 万（字节）
-encoding <i>name</i>	字符编码格式。
-eofchar <i>char</i>	特殊输入文件结束符，DOS 为 Control-z(\0x1a)，其他为空。
-eofchar { <i>inChar outChar</i> }	同上。对读-写通道，指定双向的文件结束符
-translation <i>mode</i>	行结束符翻译。在 Tcl 中，一行的结束总是用 \n 标识，但对实际的文件和设备，不同的平台可能有不同的行结束标识。mode 包含：auto(自动)、lf(换行)、cr(回车)、crlf(回车换行)、binary(二进制)
-translation { <i>inMode outMode</i> }	同上。适用于读-写双向通道
-mode <i>mode</i>	只适用于串行设备，格式：baud、parity、data、stop
-peername	只适用于套接字，远端主机的 IP 地址
-peerport	只适用于套接字，远端主机的端口号

14.2.4.2 非阻塞 I/O

默认情况下，I/O 通道是阻塞的。gets 和 read 将会一直等待到有数据可用时才返回。如果 I/O 还没有准备好接收数据，则 puts 也会等待。

fconfigure 命令可以将通道设置为非阻塞模式，在没有数据可用时，gets 和 read 会立即返回。而 puts 命令将会接收所有的数据并放在缓存中，当低层设备准备好接收数据的时候，Tcl 会自动将输出缓存中的数据输出。设置 I/O 非阻塞模式的命令格式为：

```
fconfigure channelId -blocking 0
```

14.2.4.3 缓冲

在默认情况下，Tcl 会对数据进行缓冲，来提高 I/O 的效率。有时使用者想立即输出数据而不经缓冲，则可以用 fconfigure 命令来设置 I/O 通道的缓冲属性：

```
fconfigure channelId -buffering none
```

而 属性参数-bufferSize 可控制缓冲区的大小：

```
fconfigure channelId -buffering full -bufferSize 8192
```

默认在 `stdin` 和 `stdout` 上使用行缓冲，下面的命令打开行缓冲：

```
fconfigure channelId -buffering line
```

参考文献

- [1] Brent B.Welch, 《Tcl/Tk 编程权威指南（第 3 版）》中译本, 中国电力出版社 2002.6
- [2] Ray Johnson. Tcl Style Guide. Sun Microsystems,Inc., 1999.8
- [3] Brent B. Welch, Ken Jones, Jeffrey Hobbs. Practical Programming in Tcl and Tk, Fourth Edition. Prentice Hall Publication. 2003.7.10