Tcl 语言教程

声明：本教程由本人独立完成，唯一参考来源为 http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html 网站上的英文原文教程。翻译本教程纯粹为了大家学习使用，如用于其他目的请提前获得本人允许。尽管英文教程是在 BSD 许可下发布，但我还是征得了英文教程的原作者 David N. Welton 和另一个联合作者 Clif Flynt 的同意。

由于本人英文功底和时间所限，翻译过程中难免出现纰漏、错误或导致误解的地方，欢迎大家批评指正。（本文为第一版，以后会更改纰漏并上传新版本）

译作者：洛阳城主
学校：河南科技大学
邮箱：jingwei_5107@qq.com
完成时间：2011-11-10

目录

1 前言

欢迎来到 Tcl 教程。我们写这本教程目标是为了帮助你学习 Tcl 这门语言。它针对的是那些有一些编程知识的人，尽管你并不一定是一个专家。这个教程打算作为 Tcl 手册页面的副品，而它们提供了所有 Tcl 命令的一个参考。

它分为简短的几部分，每部分涉及这门语言的不同方面。根据你所使用的系统，你可以查询参考文档来寻找你感兴趣的命令。例如，在 Unix 系统下，“man while”会显示 while 命令的使用手册页。

每部分都附有相关的例子，让你明白如何使用相关的资料。

其他的资源：

Tcl 社区是非常友善的一个。如果自己可以尝试并弄明白那将是非常值得赞赏的。但是如果你遇到了挫折，我们也非常愿意提供帮助。下面是一些获得帮助的好去处：

1. Comp.lang.tcl 新闻组。通过新闻阅读器或者 Google Groups 访问。
2. Wiki 有很多关于 Tcl 用法的好点子的有用的代码，例子和讨论。
3. 如果你需要立即的帮助，irc.freenode.net #tcl 频道经常会有一些在线的家伙可以帮到你。但是请不要焦躁，如果没有人及时帮助你（如果你需要这样的支持，考虑雇一个顾问吧）。
4. 对于那些希望获得 Tcl 更深知识的人来说，有几本书可以推荐。Cilf Flynt，也是这本教程原作者，《Tcl/Tk:开发者指南》。其他畅销书：《Tcl 和 Tk 实践编程》。

致谢：

首先，感谢 Clif Flynt 他在 BSD 协议下发布了他的资料。下面的人也作出了贡献：

1. Neil Madden
2. Arjen Markus
3. David N.Welton

当然，我们也欢迎关于如何提高本教程的评论和建议。或者，如果本教程足够好，我们也不介意来一点感谢，呵呵。

2 运行 Tcl

当我们已经安装了 Tcl，你接下来要调用使用的程序是 tclsh。举例来说，如果你写了一些代码到 hello.tcl 文件，并且你想要执行它，你可以这样做：tclsh hello.tcl。根据你安装的 Tcl 的版本及你使用的操作系统发行版本，tclsh 程序可能只是一个真正的执行体的链接，在微软 Windows 下面，它可能链接到 tclsh8.6 或者 tclsh86.exe。

Tclsh 是一个简单的命令行交互式解释器。你可以要么在命令行用脚本启动它，这样它执行脚本到结束，然后退出；也可以不需要任何参数来启动它，这样它会呈现一个交互式的提示，即“%”用来提示输入。在交互模式下，你可以输入命令，这样 Tcl 然后会执行并显示结果，或者导致的任何错误信息。退出解释器，输入“exit”，然后按回车键。使用交互式解释器是一个非常好的学习如何使用 Tcl 的方法。如果输入命令而没有参数，大部分的 Tcl 的命令会产生一个有用的错误消息，它会解释这些命令是如何使用的。通过输入“info commands”，你会得到一个你的解释器知道的全部命令的列表。

Tclsh 执行体只是启动 Tcl 解释器方法之一，另一个常用的执行体，已经被安装到了你的系统里，是 wish，或者 WIndowing SHell。这个版本的 Tcl 自动加载创建图形用户界面（GUI）的 Tk 扩展。本教程不涉及 Tk，因此这里我们不使用 wish 解释器。其他的选项也是可以的，这样开发和调试代码时，可以提供比标准 tclsh 提供的功能更丰富的环境。一个非常流行的选择是 Jeff Hobbs 编写的 TkCon 强化版解释器。Eclipse IDE，以 DLTK 扩展的形式，提供了很好的 Tcl 支持。而且 Tcl 用户的 Wiki 提供一个提供 Tcl 支持的及一个 IDE 的列表及一个全面的 Tcl 源代码编辑器的目录。但是，不要慌。如果你不知道如何使用一个复杂的开发环境，通过简单文本编辑器来写 Tcl 代码也是非常容易的。

3 简单文本输出

传统的本教程的起始点是经典的“Hello World”程序。一旦你能够输出一个字符串，你就可以随意使用 Tcl，无论是自娱自乐还是为了挣钱。

输出字符串的命令是“puts”命令。

跟在 puts 后面的简单的文本单元会被输出到标准输出设备上。默认的行为是在输出文本结束之后，输出一个该系统下对应的换行符“return”。

如果字符串有多于一个的单词，你必须把这个字符串用双引号或大括号（{}）括起来。一组被双引号或大括号括起来的单词被当做一个单独的单位，而由空格分隔的多个单词被当做是命令的多个参数。双引号和大括号都可以被用来把几个单词组成一个单独的单元。然而，他们实际上处理并不相同。在下一课你会开始学习他们之间的一些不同。记住，在 Tcl 里面，单引号并没有它在其他编程语言如 C，Perl，Python 里面那样重要。

很多 Tcl 命令（包括 puts）可以接受多个参数。如果一个字符串没有被双引号或大括号括起来，Tcl 解释器会认为其中的每个单词都是一个分隔的参数，并且会将它单独传给 puts 命令。puts 命令会尽量把这些单词评估为可选的参数。

一条 Tcl 命令是由换行符或分号结束的一组单词。Tcl 注释符是“#”，它位于每行开头，或者每行的分号后面。

示例：

```
puts "Hello, World - In quotes"    ;# This is a comment after
the command.
# This is a comment at beginning of a line
puts {Hello, World - In Braces}
puts {Bad comment syntax example}   # *Error* - there is no
semicolon!

puts "This is line 1"; puts "this is line 2"

puts "Hello, World; - With  a semicolon inside the quotes"

# Words don't need to be quoted unless they contain white
space:
puts HelloWorld
```

4 变量赋值

Tcl 里面，任何东西都可以用字符串表示，尽管内部它可能被表示成 list，integer，double，或者其他类型，这样可以提高语言的效率。

Tcl 的赋值命令是 set。

当 set 被赋予两个参数进行调用时，例如：

```
set fruit Cauliflower
```

它把第二个参数的值（Cauliflower）放在被第一个参数（fruit）引用的内存空间。set 总是返回以第一个参数命名的变量的内容。因此，当 set 以两个参数调用的时候，它把第二个参数放在被第一个参数引用的内存空间，然后返回第二个参数。在上面的例子里，例如，它会返回“Cauliflower”，没有双引号。

set 命令的第一个参数可以是一个单独的单词，像 fruit，pi，或者数组的元素。数组会稍后更详细地讨论，现在只需要记住，通过一个单独的变量名很多数据可以被包括，每一个数据可以通过数组下标来访问。数组下标，在 Tcl 里面通过在变量名后面加上圆括号实现。

set 也可以只用一个参数调用。这时，返回那个参数的内容。

下面是 set 命令的概要：

set varName ?value?

1. 如果 value 被指定，那变量 varName 的内容被设置等于 value。
2. 如果 varName 只包含字母字符，没有圆括号，那它是标量变量。
3. 如果 varName 有 varName(index)的形式，那它是关联数组的元素。

如果你看一下示例代码，你会注意到，set 命令的第一个参数只是输入了它的名字，而 puts 语句的第一个参数却前缀了“$”符号。

美元符号告诉 Tcl 去使用变量的值，例如 *X*，*Y*。

Tcl 传递数据给子例程，要么通过**名称**，要么**值**。并不改变变量内容的命令通常用**值**传递参数，而确实改变数据的值的命令通常用**名称**传递数据。

示例：

```
set X "This is a string"

set Y 1.24

puts $X
puts $Y

puts "............................."

set label "The value in Y is: "
puts "$label $Y"
```

5 求值和置换（1）：用""给参数分组

这是讨论 Tcl 命令求值期间如何处理置换的三篇文章的第一篇。

在 Tcl 里面，命令求值在第二阶段完成。第一阶段是单纯的置换操作。第二阶段才是对合成的命令的求值。注意，只有一次置换操作被执行。因此，在命令

puts $varName

里面，自身变量的内容会置换$varName，然后命令才执行。假设我们设置 varName 为“Hello World”，顺序就像：puts $varName=>puts “Hello World”.

在置换阶段，会有几种类型的置换操作。

方括号（[]）括起来的命令被置换成其执行结果。（这会在“Result of a Command- Math101”一节做更详细解释）

双引号或大括号括起来的单词被组成一个单独的参数。然而，双引号和大括号会在置换阶段产生不同的行为。这节课，我们专注于双引用在置换阶段的作用。

用双引号组合单词使置换置换发生在引号内部，或者，用专业术语就是“插入”。被置换的分组然后作为单独的参数被求值。因此，在命令

puts "The current stock value is $varName"

里面，varName 当前的内容置换了$varName，然后整个字符串被打印到输出设备，就像上面的例子一样。

一般地，反斜杠（\）禁止了对于紧跟在它后面的单个字符的置换操作。任何紧跟在反斜杠后面的字符保持不变，不会发生置换。

然而，有一些特殊的“反斜杠搭配”字符串会在置换阶段被置换成指定的值。下面的反斜杠字符串会被置换如下所示：

String	Output	Hex Value
\a	Audible Bell(铃声)	0x07
\b	Backspace（退格）	0x08
\f	Form Feed (clear screen 清屏)	0x0c
\n	New Line（换行）	0x0a
\r	Carriage Return（回车）	0x0d
\t	Tab（制表）	0x09
\v	Vertical Tab（垂直制表）	0x0b
\0dd	Octal Value（八进制）	d is a digit from 0-7
\uHHHH	H is a hex digit 0-9,A-F,a-f. This represents a 16-bit	

String	Output	Hex Value
	Unicode character. （16 位 Unicode 字符）	
\xHH....	Hex Value（16 进制值）	H is a hex digit 0-9,A-F,a-f. **Note** that the \x substitution "keeps going" as long as it has hex digits, and only uses the last two, meaning that \xaa and \xaaaa are equal, and that \xaaAnd anyway will "eat" the A of "And". Using the \u notation is probably a better idea.

最后的例外是一行末尾的反斜杠。这导致解释器忽略回车，把文本当成单独一行。解释器会在末尾的反斜杠的位置插入空白。

示例:

```
set Z Albany
set Z_LABEL "The Capitol of New York is: "

puts "$Z_LABEL $Z"   ;# Prints the value of Z
puts "$Z_LABEL \$Z"  ;# Prints a literal $Z instead of the
value of Z

puts "\nBen Franklin is on the \$100.00 bill"

set a 100.00
puts "Washington is not on the $a bill"    ;# This is not
what you want
puts "Lincoln is not on the $$a bill"      ;# This is OK
puts "Hamilton is not on the \$a bill"     ;# This is not
what you want
puts "Ben Franklin is on the \$$a bill"    ;# But, this is
OK

puts "\n.................. examples of escape strings"
puts "Tab\tTab\tTab"
puts "This string prints out \non two lines"
puts "This string comes out\
on a single line"
```

6 求值和置换（2）：用{}给参数分组

在命令求值的置换阶段，两个用于分组的运算符，即花括号（{）和双引号（”），会被 Tcl 解释器区别对待。

在上一节课，你明白，用双引号分隔的词组允许置换发生在双引号内部。**相反，使用一对花括号不允许内部置换。花括号内部的字符毫不改变地直接被传递给命令。**特殊的，只有花括号内部的并且位于行尾的“反斜杠搭配”才会在括号内部被处理。这依然是一个续行符（一行代码扩展为两行或多行）。

注意，花括号只有在其用于单词分组的时候才有这种作用（例如，单词序列的开头或结尾）。如果字符串已经分组了，或者引号，或者花括号，并且花括号出现在分组的字符串中间（例如，”foo{bar”），那么，花括号被当做普通的字符对待，而不具有特殊的意义。**如果字符串被引号分组，置换会发生在引起来的字符串内部，甚至花括号里面。**

示例：

```
set Z Albany
set Z_LABEL "The Capitol of New York is: "

puts "\n.................. examples of differences between
\" and \{"
puts "$Z_LABEL $Z"//The Capitol of New York is: Albany
puts {$Z_LABEL $Z}//$Z_LABEL $Z

puts "\n....... examples of differences in nesting \{ and
\" "
puts "$Z_LABEL {$Z}"//The Capitol of New York is: {Albany}
puts {Who said, "What this country needs is a good $0.05
cigar!"?}// Who said, "What this country needs is a good
$0.05 cigar!"?，引号的内容没有评估成值。


puts "\n.................. examples of escape strings"
puts {There are no substitutions done within braces \n \r
\x0a \f \v}//不进行转义，直接输出字符串
puts {But, the escaped newline at the end of a\
string is still evaluated as a space}//一行输出，即续行符
```

7 求值和置换（3）：用[]给参数分组

把命令放到方括号（[]）里面，你就可以得到该命令的结果。它功能上等价于 sh 编程里面的反单引号（`），或者 C 语言里面的函数返回值。

当 Tcl 解释器读取一行的时候，它会把所有的$变量置换成他们的值。如果字符串的一部分是用方括号括起来的，那么方括号里面的字符串也会被解释器当做命令进行求值，然后用返回的值置换方括号里面的字符串。

让我们用下面的代码段来举例说明：

puts [readsensor [selectsensor]]

1. 解释器扫描整个命令，注意到，有一个命令置换要执行：readsensor [selectsensor]，解释器会把它发送给解释器进行求值。
2. 解释器再次发现了一个命令需要被求值和置换，即 selectsensor。
3. 虚构的 selectsensor 命令被求值，并假设它返回一个传感器去读。
4. 这时，readsensor 命令有了一个传感器去读，然后被求值。
5. 最后，readsensor 的值继续传递回给 puts 命令，它会打印输出到屏幕。

上述规则的例外如下：

1. 被“\”转义的方括号被当做字面值方括号。
2. 方括号里面的方括号在置换阶段不会被修改。

示例：

```
set x abc
puts "A simple substitution: $x\n"

set y [set x "def"]
puts "Remember that set returns the new value of the
variable: X: $x Y: $y\n"//X:def Y:def

set z {[set x "This is a string within quotes within braces"]}
puts "Note the curly braces: $z\n"//[set x "This is a string
within quotes within braces"],大括号的关系

set a "[set x {This is a string within braces within
quotes}]"//This is a string within braces within quotes
puts "See how the set is executed: $a"
puts "\$x is: $x\n"

set b "\[set y {This is a string within braces within
quotes}]"//[set y {This is a string within braces within
quotes}]
# Note the \ escapes the bracket, and must be doubled to
be a
# literal character in double quotes
```

```
puts "Note the \\ escapes the bracket:\n \$b is: $b"//这里的两个\\输出一个反斜杠。
puts "\$y is: $y"
```

8 命令返回值-Math 101

进行数学类型操作的 Tcl 命令 expr。接下来对 expr 命令的讨论取自 expr 手册页并做了修改。很多命令都内在地使用 expr，为了评估文本表达式，例如 if，while 和 for 循环，这些将在稍后部分讨论。这里给出的所有的 expr 的建议对这些其他命令也适用。

expr 取它全部的参数（例如"2+2"），然后把这个结果当做 Tcl "表达式"（而不是正常的命令）来评估，并返回结果值。Tcl 表达式允许的运算符包括所有的标准数学函数，逻辑运算符，按位运算符，以及像 random()，sqrt()，cosh()等数学函数等等。**表达式几乎总是产生数值结果（整型或浮点型）。**

性能提示：把 expr 命令的参数包括在花括号里面会提高代码的执行速度。所以，用 expr {$i * 10}，而不要简单地用 expr $i * 10。

提醒：当评估可能包含用户输入的表达式时，你应该总是使用括号，以避免产生可能的安全缺口。expr 命令执行自己的对变量和命令的置换循环，所以你应该使用括号来防止 Tcl 解释器做这些（导致双置换）。为了解释其危险性，考虑下面的交互回话：

```
% set userinput {[puts DANGER!]}
[puts DANGER!]
% expr $userinput == 1
DANGER!
0
% expr {$userinput == 1}
0
```

在第一个例子中，包含用户提供的输入代码被求值了。然而第二个例子里面，括号阻止了这种潜在的危险。作为一个基本规则，要总是用括号包括表达式，无论是直接使用 expr 还是一些其他接受表达式作为参数的命令（如 if 和 while）。

操作数：

一个 Tcl 表达式由操作数，操作符，和圆括号组合而成。空格也可以被使用于操作数，操作符和圆括号之间。它会被表达式处理器忽略。在可能的情况下，操作数也会被解释成整型值。整型值可以指定为十进制（正常的形式），或者八进制（操作数的第一个字符是 0），或者十六进制（操作数前两个字符是 0x）。

注意，八进制和十六进制转换在 expr 命令里面与 Tcl 置换阶段不同。在置换阶段，0x32 会转换成 ASCII 码 "2"，而 expr 会把它转换成十进制的 50.

如果一个操作数不是上面给出的整型格式，可能的话，那它会被当做浮点型数。浮点型数可以以遵循 ANSI C 编译器接受的任何形式指定。例如，下面的全部形式都是有效的浮点数：

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

```
.000001
```

如果没有可行的数值解释，那么操作数会被当做一个字符串（并且，只能在其上应用有限集合的操作数）。

小心：以 0 开头的 0700 不会被解释成十进制数 700（七百），而是被解释成八进制数 700=7*8*8=448（十进制）。

更糟的是，如果一个数包含数字 8 或 9，会导致错误：

```
% expr {0900+1}
expected integer but got "0900" (looks like invalid octal number)
```

八进制数实际上是过去的遗留产物，而当时这种数据格式更常用。

操作数可以以下面任何形式指定：

1. 一个数，要么是整型，要么是浮点型。
2. Tcl 变量，使用标准的$标记。变量的值会作为操作数。

运算符：

全部有效的运算符在下面列了出来，按运算优先级递减的顺序排列。

运算符	解释
- + ~ !	一元减，一元加，按位 NOT（按位取反），逻辑 NOT（非）。这些操作符都不可以用于字符串操作数，按位 NOT（按位取反）只可以用于整数。
**	求幂（浮点数和整型都有效）。
* / %	乘，除，求余。这些操作符不可以用于字符串。求余只可以用于整型。求余的符号总是与除数保持相同，而其绝对值小于除数。
+ -	加和减。对于任何数字操作数均有效。
<< >>	左移和右移（位）。仅对整型操作数有效。
< > <= >=	相关运算符：小于，大于，小于等于，大于等于。每个运算符如果条件为真返回 1，否则返回 0.这些运算符可以被应用于数值型操作数，也可以是字符串（当字符串比较的时候）。

eq ne in ni	比较两个字符串相等（eq）还是不相等（ne）。in 和 ni 用于检查字符串是包含在列表中（in）还是不在（ni）。这些运算符全部返回 1（true）或者 0（false）。使用这些运算符要确保操作数仅被当做字符串（及列表），而不是可能的数字：
	`% expr { "9" == "9.0"}`
	`1`
	`% expr { "9" eq "9.0"}`
	`0`

&	按位与。仅对整型有效。
^	按位异或 OR。仅对整型有效。
|	按位 OR。仅对整型有效。
&&	逻辑与 AND。如果两个参数都非 0，返回 1，否则返回 0.仅对数值型有效（整数，浮点数）。

||	逻辑或 OR 如果两个操作数都是 0，返回 0；否则返回 1. 仅对数值型有效（整数，浮点数）。

x?y:z	If-then-else。如果 x 求值为非 0，那么结果是 y，否则结果是 z。x 必须是数字值。
	`% set x 1`
	`% expr { $x>0? ($x+1) : ($x-1) }`
	`2`

数学函数：

Tcl 支持下面的数学函数在表达式里面使用：

abs	acos	asin	atan
atan2	bool	ceil	cos
cosh	double	entier	exp
floor	fmod	hypot	int
isqrt	log	log10	max
min	pow	rand	round
sin	sinh	sqrt	srand
tan	tanh	wide	

除了上面的函数，你也可以在表达式里面使用命令。例如：

```
% set x 1
% set w "Abcdef"
% expr { [string length $w]-2*$x }
4
```

类型转换：

Tcl 支持以下函数，来把一个数从一种形式转换成另一种形式：

1. double() 把一个数转换成浮点数。
2. int() 把一个数转换成通常的十进制整数数（截断小数部分）。
3. wide() 把一个数转换成所谓的宽整数（这些数有更大的范围）。
4. entire() 把一个数转换成一个空间大小适当的整数，而不需要进行截断。这可能和 int()和 wide()返回相同，或者是任意大小的一个整数。

下一课会更详细地解释各种数值类型。

示例：

```
set X 100
set Y 256
set Z [expr {$Y + $X}]
set Z_LABEL "$Y plus $X is "

puts "$Z_LABEL $Z"//256 plus 100 is  356
puts "The square root of $Y is [expr { sqrt($Y) }]\n"

puts "Because of the precedence rules \"5 + -3 * 4\"   is:
[expr {-3 * 4 + 5}]"//-7
```

```
puts "Because of the parentheses     \"(5 + -3) * 4\" is:
[expr {(5 + -3) * 4}]"//8

set A 3
set B 4
puts "The hypotenuse of a triangle: [expr {hypot($A,$B)}]"

#
# The trigonometric functions work with radians ...
#
set pi6 [expr {3.1415926/6.0}]
puts "The sine and cosine of pi/6: [expr {sin($pi6)}] [expr
{cos($pi6)}]"

#
# Working with arrays
#
set a(1) 10
set a(2) 7
set a(3) 17
set b   2
puts "Sum: [expr {$a(1)+$a($b)}]"
```

9 计算机和数字

如果你刚开始编程，那这一课可能包含一些惊喜的信息。但即使你熟悉写程序，计算机依然可以用数字做一些不可思议的事情。本课的目的是拨开一些秘密的云雾，并指引你绕过可能碰到的弯路。

这些秘密各自存在于编程语言里面，尽管其中一种编程语言可能使你比另一种语言更好地避开它们。问题是，计算机不能以我们的方式处理我们使用的数字。

例如（*）：

```
# Compute 1 million times 1 million
% puts [expr {1000000*1000000}]
-727379968
```

出于大部分人的意外，结果是负的。并不是 1000000000000，而是一个负数被返回了。

重要提示：我使用 Tcl8.4.1 完成所有的示例程序。在 Tcl8.5 里，结果希望是更直观的，即两个所谓的大整数的加法的和。然而，通常的主题依然是一样的。

现在考虑下面的示例,它基本上是相同的，除了一个十进制点。

```
# Compute 1 million times 1 million
% puts [expr {1000000*1000000.}]
1e+012
```

原因很简单，当然，如果你了解更多计算机算术背景的话。

1. 在第一个例子里面，我们让两个 integer 整数相乘，或者叫 short integer 短整型。当我们习惯于这些数的范围是正无穷到负无穷时，计算机不能处理这样的范围（至少不那么容易）。所以，作为替代方案，计算机只是处理实际数学整数的一个子集。他们处理从-2^{31} to 2^{31}-1（一般说来），即-2147483648 to 2147483647 的整数。

2. 在第二个例子里面，我们让两个浮点数相乘，统一的说法：一个实型数字或实数（尽管在数学上的实数和计算机实数之间有着巨大的差别，而计算机实数又位于另一组秘密和弯路的中心）。浮点数有更大的范围，他们可以用来处理像 1.2 和 4.1415026 这样的数。

浮点数典型的范围大体上是-1.0e+300 到 1.0e-300,0.0,以及 *1.0e-300* 到 *1.0e+300*。浮点数精度是有限的，通常是 12 位有效数字。

换一个实际的说法就是，浮点数可以是一个 1 后面跟着 300 个 0，或者小到 0.000…1(…代表 295 个 0)。

因为范围太大，在第二个例子里面，结果落在了范围内，因此我得到了我们预期的结果。

Tcl 策略：

Tcl 使用一个简单但有效地策略来决定使用哪种类型的数进行计算：

1. 如果你对整数进行加、减、乘、除，那结果是整数。如果结果在整肃范围内，你得到正确的结果；否则，你会得到一些看起来完全错误的东西。（注意，不久前，浮点数计算比整数计算花费多得多的时间。大部分的计算机并不提示整数结果溢出，因为那样太耗时了：一台计算机典型地使用很多这样的操作，而大

部分都落在预设的范围内。）

2. 如果你对一个整数和一个浮点数进行加、减、乘、除，那整数先被转换成等值的浮点数，然后再进行计算。结果是浮点数。

浮点数计算是相当复杂的，当前的（IEEE）标准在微小的细节上规定了应该发生什么。一个细节是如果结果溢出将会被报告。Tcl 捕获并显示这些警告：

```
# Compute 1.0e+300/1.0-300
% puts [expr {1.0e300/1.0e-300}]
floating-point value too large to represent
```

那些秘密和弯路是什么：

现在举一些你自己都会陷进去的秘密.执行下面的脚本：

```
#
# Division
#
puts "1/2 is [expr {1/2}]"
puts "-1/2 is [expr {-1/2}]"
puts "1/2 is [expr {1./2}]"
puts "1/3 is [expr {1./3}]"
puts "1/3 is [expr {double(1)/3}]"
```

前两个计算会得到令人惊奇的结果：0 和-1.那是因为结果是一个整数，1/2 和-1/2 的数学上的准确的值并不是整数。

如果你对 Tcl 工作的细节感兴趣，下面的输出 q 由如下的式子确定：

```
a = q * b + r
0 <= |r| < |b|
r has the same sign as q
```

下面是一些浮点数的例子：

```
set tcl_precision 17  ;# One of Tcl's few magic variables:
                      ;# Show all decimals needed to exactly
                      ;# reproduce a particular number
puts "1/2 is [expr {1./2}]"
puts "1/3 is [expr {1./3}]"

set a [expr {1.0/3.0}]
puts "3*(1/3) is [expr {3.0*$a}]"

set b [expr {10.0/3.0}]
puts "3*(10/3) is [expr {3.0*$b}]"

set c [expr {10.0/3.0}]
set d [expr {2.0/3.0}]
puts "(10.0/3.0) / (2.0/3.0) is [expr {$c/$d}]"
```

```
set e [expr {1.0/10.0}]
puts "1.2 / 0.1 is [expr {1.2/$e}]"
```

尽管上面的很多计算给出了你期望的结果，但是注意最后的数。最后的两个数并没有得到准确的 5 和 12.这是因为计算机只可以处理有限精度的数：浮点数并不是我们数学意义上的实数。

稍微有点意外的是，1/10 也出问题了。1.2/0.1 结果是 11.999999999999998，不是 12.这是一个在大部分当前计算机和编程语言里面非常令人不快的例子：他们对通常的十进制分数不起作用，而只对二进制分数起作用。所以，0.5 可以精确表示，但 0.1 却不可以。

一些实际影响：

浮点数并不是普通的十进制或实数的事实以及计算机实际处理浮点数的方式带来了很多后果和影响：

- 一台计算机上得到的结果可能和另一台得到的并不准确匹配。通常这个偏差很小，但如果你进行了大量计算，那他们可能会累加起来。
- 每当你从浮点数到整数进行转换时，比如说当确定一个图表的标签时（范围是 0 到 1.2，你想要步长是 0.1），你需要小心：

```
#
# The wrong way
#
set number [expr {int(1.2/0.1)}]  ;# Force an integer -
                                  ;# accidentally number = 11

for { set i 0 } { $i <= $number } { incr i } {
   set x [expr {$i*0.1}]
   ... create label $x
}



#
# A right way - note the limit
#
set x    0.0
set delta 0.1
while { $x < 1.2+0.5*$delta } {
   ... create label $x
   set x [expr {$x + $delta}]
}
```

- 如果你想要做金融计算，注意：做这样的计算有特定的标准，而不幸的是，这些标准取决于他们使用哪个国家的——美国标准和欧洲标准稍微有所不同。
- 超越函数，像 sin()，exp()，根本没有被标准化。各台计算机之间的输出可能相差一位或更多位。所以，如果你想要绝对确信∏（pi）是一个指定的值，

那就使用那个值，而不要指望像这样的公式：

```
#
# Two different estimates of "pi" - on Windows 98
#
set pi1 [expr {4.0*atan(1.0)}]
set pi2 [expr {6.0*asin(0.5)}]
puts [expr {$pi1-$pi2}]
-4.4408920985006262e-016
```

10 数值比较 101——if

跟绝大多数语言一样，Tcl 支持 if 命令。语法是：

- if *expr1* ?then? *body1* elseif *expr2* ?then? *body2*

  elseif ... ?else? *?bodyN?*

then 和 else 是可选的，尽管一般 then 被省略而 else 被使用。

if 后面的测试表达式应该返回一个值，它应该能够被解释成如下并相应表示 true 和 false 的形式：

	False	True
a numeric value	0	all others
yes/no	no	yes
true/false	false	true

如果测试表达式返回字符串"yes"/"no" 或者 "true"/"false"，不区分大小写。True/FALSE or YeS/nO 也是合法的返回值。

如果测试表达式求值为 True，那么 body1 会被执行。

如果测试表达式求子为 False，那 body1 后面的单词会被检查。如果下一个单词是 elseif，那么下一个测试表达式会被当做测试条件。如果下一个单词是 else，那最后的 body 会被当做命令来求值。

If 后面的测试表达式以 expr 命令一样的方式被求值。

If 后面的测试表达式可以包含在引号或括号内。如果是括号，它会在 if 命令内部求值，而如果是引号，它会在置换阶段被求值，然后另一轮置换会在 if 命令里面完成。

注意：额外的一轮置换可能会引起意料之外的问题，避免使用它。

示例：

```
set x 1

if {$x == 2} {puts "$x is 2"} else {puts "$x is not 2"}

if {$x != 1} {
    puts "$x is != 1"
} else {
    puts "$x is 1"
}

if $x==1 {puts "GOT 1"}
```

```
#
# Be careful, this is just an example
# Usually you should avoid such constructs,
# it is less than clear what is going on and it can be dangerous
#
set y x
if "$$y != 1" {
    puts "$$y is != 1"
} else {
    puts "$$y is 1"
}

#
# A dangerous example: due to the extra round of
substitution,
# the script stops
#
set y {[exit]}
if "$$y != 1" {
    puts "$$y is != 1"
} else {
    puts "$$y is 1"
}
```

11 文本比较——switch

Switch 命令允许你在你的代码中作出多个选择。它和 C 里面的相似，只是它更灵活。因为你可以在字符串上使用 switch，而不仅仅是整数。字符串会跟一组模式进行对比，当一个模式匹配这个字符串时，与那个模式相关的代码会被求值。

当你想要用一个变量跟几个可能的值对比时，使用 switch 是一个好主意，而不是使用很长的 if…elseif…elseif 语句。

该命令的语法是：

- switch **string** pattern1 body1 ?pattern2 body2? ... ?patternN bodyN?

或者

- switch ***string*** *{ pattern1 body1 ?pattern2 body2?...?patternN bodyN? }*

string 是一个你要测试的字符串，pattern1, pattern2,etc 是字符串将要对比的模式。如果 **string** 匹配一个模式，那么和那个模式关联的 body 内部的代码就会被执行。body 的返回值会被作为 switch 语句的返回值返回。并且，只有一个模式会被匹配。

如果最后一个模式参数是 default 字符串，那个模式会匹配任何字符串。这保证了无论 **string** 是什么，一些代码都将会被执行。

如果没有 default 参数，并且任何模式都不匹配 **string**，那么 switch 命令返回空的字符串。

如果你在这个命令上使用括号，那这些模式上不会发生置换。而命令的主体会被解析并求值，跟任何其他命令一样，所以置换的传递会在其上完成，就行第一种语法一样。第二种形式的好处是你可以分多行写命令，这样用括号括起来更易读。

注意：当使用 switch 和 if 命令的时候，你可以使用括号来对 body 参数分组。这是因为这些命令传递他们的 body 参数给 Tcl 解释器进行求值。这个求值过程包括置换的传递，和并不在 body 参数内的代码所做的一样。

示例：

```
set x "ONE"
set y 1
set z ONE

# This is probably the easiest and cleanest form of the
command
# to remember:
switch $x {
   "$z" {
      set y1 [expr {$y+1}]
      puts "MATCH \$z. $y + $z is $y1"
   }
```

```
    ONE {
        set y1 [expr {$y+1}]
        puts "MATCH ONE. $y + one is $y1"
    }
    TWO {
        set y1 [expr {$y+2}]
        puts "MATCH TWO. $y + two is $y1"
    }
    THREE {
        set y1 [expr {$y+3}]
        puts "MATCH THREE. $y + three is $y1"
    }
    default {
        puts "$x is NOT A MATCH"
    }
}


switch $x "$z" {
    set y1 [expr {$y+1}]
    puts "MATCH \$z. $y + $z is $y1"
} ONE {
    set y1 [expr {$y+1}]
    puts "MATCH ONE. $y + one is $y1"
} TWO {
    set y1 [expr {$y+2}]
    puts "MATCH TWO. $y + two is $y1"
} THREE {
    set y1 [expr {$y+3}]
    puts "MATCH THREE. $y + three is $y1"
} default {
    puts "$x does not match any of these choices"
}


switch $x "ONE" "puts ONE=1" "TWO" "puts TWO=2" "default"
"puts NO_MATCH"

switch $x \
"ONE"    "puts ONE=1"  \
"TWO"    "puts TWO=2" \
"default"    "puts NO_MATCH";
```

12 循环 101-while 循环

Tcl 包括两个循环命令，即 while 和 for 命令。就像 if 语句，他们求他们测试语句的值，跟 expr 命令的方式一样。这一节，我们讨论 while 命令，下一节讨论 for 命令。在这些命令其中一个大部分使用的情况下，另一个也可以使用。

- **while *test body***

while 命令把 test 当做一个表达式来求值。如果 test 是 true，那 body 里面的代码就被执行。Body 里面的代码执行完成，test 被再次求值。

Body 内部的 continue 语句会停止代码的执行，test 会被重新求值。Body 内部的 break 语句会中断整个 while 循环，执行会从 body 后面的下一行继续进行。

在 Tcl 里面，任何东西都是命令，任何东西都经历同样的置换阶段。由于这个原因，test 必须被放在括号里面。如果 test 被放在引号里面，置换阶段会把任何变量置换成它们当前的值，并传递这个 test 结果给 while 命令进行求值。因为 test 只含有数字，它的值总是相同的，很可能导致死循环。

看一下下面的两个循环。如果不是第二个循环里面的 break 命令，它会永远循环下去。

示例：

```
set x 1

# This is a normal way to write a Tcl while loop.

while {$x < 5} {//正常循环写法
    puts "x is $x"
    set x [expr {$x + 1}]
}

puts "exited first loop with X equal to $x\n"

# The next example shows the difference between ".." and
{...}
# How many times does the following loop run?  Why does it
not
# print on each pass?

set x 0
while "$x < 5" {//此处循环 3 次
    set x [expr {$x + 1}]
    if {$x > 7} break
    if "$x > 3" continue
    puts "x is $x"
}
puts "exited second loop with X equal to $x"
```

13 循环 102——for 和 incr

Tcl 支持迭代循环构造，和 C 里面的 for 循环相似。For 命令在 Tcl 里面接受 4 个参数。一个初始化，一个条件，一个增量，一个代码体（循环每次都求值）。

for 语法命令语法：

- ***for start test next body***

在对 for 命令求值期间，start 代码被求值一次，在任何其他参数被求值之前。Start 代码被求值之后，test 被求值。如果 test 求值为 true，那么再对 body 求值，最后，next 参数被求值。对 next 参数求值之后，解释器回到 test，重复这个过程。如果 test 求值为 false，那么循环会立即退出。

Start 是该命令的初始化部分。它通常被用来初始化迭代变量，但是也可以包含任何你希望在循环开始之前执行的代码。

Test 参数被当做表达式来求值，就像 expr，while 和 if 命令一样。

Next 通常是一个增量命令，但也可以包含 Tcl 解释器可以进行求值的任何命令。

Body 是要执行的代码体。

由于通常你并不想 Tcl 解释器在传递控制给 for 命令之前，在置换阶段就把变量换成他们的当前值，那么你通常要用花括号分组参数。当括号被使用来分组，换行符不会被当做 Tcl 命令的结束。这让写多行命令更容易。**然而，左大括号必须和 for 命令位于同一行，否则 Tcl 解释器会把 next 命令的右大括号当做命令的结束，这样你会得到一个错误。**这和其他像 C 或者 Perl 语言不同，这些语言不在乎你放置括号的位置。

在 body 体内，命令 break 和 continue 可以被使用，和它们跟 while 的使用情况一样。当碰到 break 时，循环立即退出。当遇到 continue 时，body 的求值停止，test 被再次求值。

由于增加迭代变量是如此常用，Tcl 为此有专门的命令：

- ***incr varName ?increment?***

这个命令把第二个参数的值加到第一个参数命名的变量上。如果没有值赋给第二个参数，那么默认赋给 1.

示例：

```
for {set i 0} {$i < 10} {incr i} {
    puts "I inside first loop: $i"
}

for {set i 3} {$i < 2} {incr i} {
    puts "I inside second loop: $i"
}

puts "Start"
set i 0
while {$i < 10} {
    puts "I inside third loop: $i"
    incr i
```

```
    puts "I after incr: $i"
}

set i 0
incr i
# This is equivalent to:
set i [expr {$i + 1}]
```

14 添加新的命令到 Tcl——proc

Tcl 中，实际上命令（经常被认为是其他语言里面的“函数”）和语法没有区别。也没有像 C，Java，Python，Perl 等语言中的保留关键字（像 if 和 while）。当 Tcl 解释器启动的时候，存在一个已知的命令的列表，解释器可以用其解析每一行。这些命令包括 while，for，set，puts 等等。然而，不管是内置的还是你自己通过 proc 命令创建的命令，只要他们遵守相同的所有 Tcl 命令都遵守的语法规则，他们都算是规则的 Tcl 命令。

Pro 命令用于创建新命令。语法如下：

- proc *name args body*

当 proc 被求值的时候，它创建一个新命令，名称是 name，参数是 args。当 name 过程被调用的时候，它就执行 body 内部的代码。

Args 是一个参数列表，它会被传递给 name。当 name 被调用时，具有这些名称的局部变量会被创建，并且这些传递给 name 的参数的值会被拷贝给这些变量。

Proc 命令中的 body 返回值可以用 return 命令定义。Return 命令会返回它的参数给调用它的程序。如果没有 return，当 body 最后一条命令执行完之后会返回到调用者。最后一条命令的返回值会成为过程的返回值。

示例：

```
proc sum {arg1 arg2} {
    set x [expr {$arg1 + $arg2}];
    return $x
}

puts " The sum of 2 + 3 is: [sum 2 3]\n\n"

proc for {a b c} {
    puts "The for command has been replaced by a puts";
    puts "The arguments were: $a\n$b\n$c\n"
}

for {set i 1} {$i < 10} {incr i}
```

15 proc 命令参数和返回值的变化

一个 proc 命令用一组必须的参数定义（就像上一个 sum 那样），或者它也可以有可变数量的参数。一个参数也可以在被定义的同时拥有一个默认值。

变量可以定义的同时赋予一个默认值，方法是把变量名称和默认值放在括号里面来作为 args 中的一个。**例如：**

```
proc justdoit {a {b 1} {c -1}} {

}
```

由于参数变量 b 和 c 存在默认值，你可以以三种方式调用这个过程：justdoit 10，这样会把 a 设置为 10，而 b 和 c 分别保持默认值 1 和-1；justdoit 10 20，这样同样地设置 b 为 20，而 c 还是默认值；或者设置全部三个参数，覆盖默认值。

如果最后声明的参数是单词 args，proc 命令可以接受可变数量的参数。如果 proc 命令参数列表最后的参数是 args，那么任何还未赋值的剩余参数都会被赋给 args。

下面的 example 过程定义了三个参数。当其被调用时，至少一个参数**必须**出现。第二个参数可以被忽略，这样它会被赋值为空串。通过声明 args 为最后一个参数，example 可以接受可变数量的参数。

注意：如果一个带默认值的变量后面却跟着一个非 args 的变量，那这个默认值绝不会被使用。例如，如果你这样声明一个 proc：proc function { a {b 1} c} {...}，你会总是必须以三个参数调用它。

Tcl 按照 proc 变量在命令里面列出的顺序为其赋值。如果你调用 function，提供两个参数，它们会被赋给 a 和 b，并且 Tcl 会产生一个错误因为 c 未被定义。

但是，你可以声明可能会没有值的其他参数，让他们跟在有默认值参数的后面。例如，下面是有效的：proc example {required {default1 a} {default2 b} args} {...}

在这个例子中，example 需要一个参数，并且会被赋给 required。如果有两个参数，第二个参数会被赋给 default1，如果有三个参数，第一个赋给 required，第二个赋给 default1，第三个赋给 default2. 如果例子的参数多于 3 个，第三个以后的所有的参数都会被赋给 args。

示例：

```
proc example {first {second ""} args} {
    if {$second eq ""} {
        puts "There is only one argument and it is: $first"
        return 1
    } else {
        if {$args eq ""} {
            puts "There are two arguments - $first and
$second"
            return 2
```

```
       } else {
          puts "There are many arguments - $first and
$second and $args"
          return "many"
       }
    }
}

set count1 [example ONE]
set count2 [example ONE TWO]
set count3 [example ONE TWO THREE ]
set count4 [example ONE TWO THREE FOUR]// ONE and TWO and
THREE FOUR

puts "The example was called with $count1, $count2, $count3,
and $count4 Arguments"
```

16 变量作用域——global 和 upvar

Tcl 对作用域内的变量进行求值，这些作用域可以被 procs，命名空间（看创建可重用的库-包和命名空间）或者最高层的 global 声明。

被求值的变量的作用域可以通过 global 和 upvar 命令改变。

Global 命令会导致一个局部作用域（过程内部）的变量引用那个名称的全局变量。

Upvar 命令也是相似的。它把当前作用域下的一个变量的名称关联到一个不同作用域的变量。这通常被用来模拟 procs 的“传引用”。

在其他 Tcl 代码里面，你可能也会碰到 variable 命令。它是命名空间系统的一部分，并且将会在那章详细讨论。

正常地，Tcl 使用一种叫做引用计数的“垃圾收集”，目的是自动清理那些不再被使用的变量。例如当他们在过程结束时会超出作用域，这样你就不必自己记录他们的情况。也可以用合适的名为 unset 的命令显式地复原他们。

Upvar 的语法是：

upvar *?level? otherVar1 myVar1 ?otherVar2 myVar2? ... ?otherVarN myVarN?*

Upvar 命令使 myVar1 变成了 otherVar1 的一个引用，myVar2 变成了 otherVar2 的一个引用，等等。**otherVar 变量的等级被声明为相对于当前过程的 level 等级。默认 level 是 1，下一级提高。**

如果 level 用的是数字，那么该层引用栈中位于当前层之上的许多层。

如果 level 前缀有“#”，那它引用全局以下的很多层。如果 level 是#0，那它引用的就是全局水平的变量。

如果你使用 upvar，但除了#0 和 1，你很可能在自找麻烦，除非你真地知道你在做什么。

你应该尽可能避免使用全局变量。如果你有很多全局变量，那么你应该考虑一下你的程序的设计。

注意：由于只有一个全局空间，如果你正在导入其他人的代码并且不够仔细，名称冲突会非常容易出现。建议你在每一个全局变量的前面加上一个唯一的前缀来避免意想不到的冲突。

示例：

```
proc SetPositive {variable value } {
    upvar $variable myvar
    if {$value < 0} {
        set myvar [expr {-$value}]
    } else {
        set myvar $value
    }
    return $myvar
}

SetPositive x 5
```

```
SetPositive y -5

puts "X : $x   Y: $y\n"

proc two {y} {
   upvar 1 $y z                  ;# tie the calling value
to variable z
   upvar 2 x a                  ;# Tie variable x two levels
up  to a
   puts "two: Z: $z A: $a"       ;# Output the values, just
to confirm
   set z 1                       ;# Set z, the passed
variable to 1;
   set a 2                      ;# Set x, two layers up to
2;
}

proc one {y} {
   upvar $y z                    ;# This ties the calling
value to variable z
   puts "one: Z: $z"             ;# Output that value, to
check it is 5
   two z                        ;# call proc two, which will
change the value
}

one y                           ;# Call one, and output X and
Y after the call.
puts "\nX: $x  Y: $y"


proc existence {variable} {
   upvar $variable testVar
   if { [info exists testVar] } {
      puts "$variable Exists"
   } else {
      puts "$variable Does Not Exist"
   }
}

set x 1
set y 2
for {set i 0} {$i < 5} {incr i} {
   set a($i) $i;
```

```
}

puts "\ntesting unsetting a simple variable"
# Confirm that x exists.
existence x
# Unset x
unset x
puts "x has been unset"
# Confirm that x no longer exists.
existence x
```

17 Tcl 数据结构 101-list（列表）

List 是一个基本的 Tcl 数据结构。一个 list 是一个简单的有序集合，可以包含数字、单词、字符串或其它 list。即使是 Tcl 命令也是 list，其中第一项是一个 proc 名称，其后的 list 成员是 proc 的参数。

List 可以用以下几种方式创建：

- **通过设置一个变量为由多个值组成的列表：**
  set lst {{item 1} {item 2} {item 3}}
- **用 split 命令：**
  set lst [split "item 1.item 2.item 3" "."]
- **用 list 命令：**
  set lst [list "item 1" "item 2" "item 3"]

一个独自的 list 成员可以通过 lindex 下标命令访问。

这些命令的简单描述：

list *?arg1? ?arg2? ... ?argN?*

构造一个这些参数组成的 list。

split *string ?splitChars?*

把 *string* 分成多个项组成的 list，分隔符是 *splitChars*。*splitChars* 默认是空格。**注意：**如果有多个 *splitChars*，那么每一个都会独立地执行拆分字符串操作。换句话说，split ″1234567″ ″36″会返回下面的列表{12 45 7}。

lindex *list index*

返回第 index 个列表项。**注意：**列表下标从 0 开始，而不是 1.所以第一项位于下标 0 处，以此类推。

llength *list*

返回李冰元素的个数。

列表项可以通过使用 foreach 命令进行枚举：

foreach *varname list body*

foreach 命令会对每一个 list 项执行 body 中的代码。每一次，***varname*** 会包含 list 下一项的值。

实际上，上面的 foreach 形式是一种简单的形式，但是这个命令却十分强大。它允许你同时接受多个来自列表的变量：foreach {a b} $listofpairs { ... }。你甚至可以同时接受来自多个列表的单个变量。例如：foreach a $listOfA b $listOfB { ... }。

示例：

```
set x "a b c"
puts "Item at index 2 of the list {$x} is: [lindex $x 2]\n"
```

```
set y [split 7/4/1776 "/"]
puts "We celebrate on the [lindex $y 1]'th day of the [lindex
$y 0]'th month\n"

set z [list puts "arg 2 is $y" ]
puts "A command resembles: $z\n"

set i 0
foreach j $x {
   puts "$j is item number $i in list x"
   incr i
}
```

18 增加&删除 list（列表）成员

增加和删除 list 成员的命令是：

concat *?arg1 arg2 ... argn?*

连接这些 args 参数到一个单独的列表。它同时去除了 args 中开头和结尾的空白，并在每一个参数中间添加一个单独的分隔符。Concat 的 args 参数可以是单个元素，也可以是 list 列表。如果一个 arg 已经是 list，那它的内容会被和其他 args 连接在一起。

lappend *listName ?arg1 arg2 ... argn?*

追加 args 到 listName 列表，每一个 arg 都被当做一个列表元素。

linsert *listName index arg1 ?arg2 ... argn?*

返回一个新的 list，即 listName 的第 index 个元素之前被插入了多个元素后形成的列表。每一个元素参数会变成新列表单独的元素。如果 index 小于等于 0，那这些新元素被插入到列表的开始位置。如果 index 是 end，或者它等于或超出了列表元素的总个数，那新元素被追加到列表末尾。

lreplace *listName first last ?arg1 ... argn?*

返回一个新列表，即 listName 的 N 个元素被 args 替换掉后的列表。如果 first 小于等于 0，那 lreplace 就从列表的第一个元素开始替换。如果 first 大于列表的结束位置（单词 **end**），那 lreplace 行为相当于 lappend。如果 args 的个数少于 first 和 last 的差值，那么没有 args 的位置会被删除。

lset *varName index newValue*

lset 命令可以被用来直接设置列表的元素，来代替 lreplace。

当你有任意数目的某种数据并且想要按序访问他们的时候，Tcl 中的列表是应该使用的正确的数据结构。在 C 里面，你可能会使用一个数组。在 Tcl 里面，数组是关联数组-哈希表，就像你在将来的部分会看到的。如果你想要有一个某种事物的集合，并且引用第 N 个事物（比如给出我一组数字中的第 10 个元素）或者通过 foreach 顺序遍历他们。

看一下示例代码，特别注意字符集被分成单独的列表元素的方式。

示例：

```
set b [list a b {c d e} {f {g h}}]
puts "Treated as a list: $b\n"

set b [split "a b {c d e} {f {g h}}"]
puts "Transformed by split: $b\n"

set a [concat a b {c d e} {f {g h}}]
puts "Concated: $a\n"

lappend a {ij K lm}                    ;# Note: {ij K lm}
is a single element
puts "After lappending: $a\n"

set b [linsert $a 3 "1 2 3"]           ;# "1 2 3" is a single
element
puts "After linsert at position 3: $b\n"

set b [lreplace $b 3 5 "AA" "BB"]
puts "After lreplacing 3 positions with 2 values at position
3: $b\n"
```

19 更多的的 list 命令-lsearch，lsort，lrange

列表可以用 lsearch 命令进行搜索，用 lsort 命令进行排序，用 lrange 命令提取部分列表项。

lsearch ***list pattern***

搜索一个匹配 pattern 的 list 项，返回第一个匹配项的下标，如果没有匹配项返回-1.默认情况下，lsearch 使用 glob 模式进行匹配。查看 globbing 章节。

lsort *list*

对 list 排序，返回经过排序的新的 list。默认情况下，按字母顺序排序。注意，这个命令返回经过排序的列表作为结果，而不是就地对列表排序。如果你有一个 list 变量，对它排序的方法应该这样：set lst [lsort $lst]。

lrange ***list first last***

返回由 first 和 last 之间的项组成的列表。如果 first 小于等于 0，它被当做第一个元素。如果 last 是 **end** 或者大于列表数目的值，它被当做末尾位置。如果 first 大于 last，那么返回空列表。

示例：

```
set list [list {Washington 1789} {Adams 1797} {Jefferson
1801} \
             {Madison 1809} {Monroe 1817} {Adams 1825} ]

set x [lsearch $list Washington*]
set y [lsearch $list Madison*]
incr x
incr y -1                    ;# Set range to be
not-inclusive

set subsetlist [lrange $list $x $y]

puts "The following presidents served between Washington
and Madison"
foreach item $subsetlist {
   puts "Starting in [lindex $item 1]: President [lindex
$item 0] "
}

set x [lsearch $list Madison*]
```

```
set srtlist [lsort $list]
set y [lsearch $srtlist Madison*]

puts "\n$x Presidents came before Madison chronologically"
puts "$y Presidents came before Madison alphabetically"
```

20 简单的模式匹配-“globbing”

默认情况下，lsearch 使用 globbing 方法来找到一个匹配项。Globbing 是一种通配符匹配技术，大部分 Unix shell 使用它。

Globbing 通配符是：

*****

匹配任意数量的任何字符

?

匹配任何一个字符

\X

反斜杠在 globbing 里面对一个特殊字符进行转意，和 Tcl 的置换的方式一样。使用反斜杠让你可以使用 glob 匹配一个*****或者**?**。

[...]

匹配括号内的任意一个字符。通过使用括号内的范围，一个范围内的字符可以被匹配。例如，[a-z]会匹配任何小写字母。

在后面的章节，你会看到 glob 命令用于目录（路径）的模式匹配，并返回匹配的文件的列表。

示例：

```
# Matches
string match f* foo

# Matches
string match f?? foo

# Doesn't match
string match f foo

# Returns a big list of files on my Debian system.
set bins [glob /usr/bin/*]
```

21 字符串子命令-长度，下标，范围

Tcl 命令经常会有“子命令”。String 命令就是这些命令中的一个。String 命令把它的第一个参数当做一个子命令。利用子命令是使一个命令做多种事情而不必使用模糊的名称的很好的方法。例如，Tcl 有 string length 命令而不是，比如 slength。

这一课包含这些 string 子命令：

`string length` ***`string`***

返回 `string` 的长度。

`string index` ***`string index`***

返回 `string` 的第 `index` 个字符。

`string range` *`string first last`*

返回 `first` 到 `last` 之间字符组成的字符串。

示例：

```
set string "this is my test string"

puts "There are [string length $string] characters in
\"$string\""

puts "[string index $string 1] is the second character in
\"$string\""

puts "\"[string range $string 5 10]\" are characters between
the 5'th and 10'th"
```

22 字符串比较-compare,match,first,last,wordend

有 6 个可以进模式和串匹配的字符串子命令。这些是相对快的操作，肯定比通常的表达式要快，即使没有表达式那么强大。

`string compare` ***string1 string2***

比较 ***string1*** 和 ***string2***，返回：

- -1，如果 ***string1*** 小于 ***string2***。
- 0，如果 ***string1*** 等于 ***string2***。
- 1，如果 ***string1*** 大于 ***string2***。

这些比较按照字母顺序，而不是数字顺序。换句话说，“a”小于“b”，“10”小于“2”。

`string first` *string1* `string2`

返回 *string1* 中第一次匹配 string2 的首字符的下标，如果没有匹配成功，返回-1.

`string last` ***string1 string2***

返回 ***string1*** 中最后一次匹配 ***string2*** 的首字符的下标，如果没有匹配成功，返回-1.

`string wordend` ***string index***

返回 ***string*** 中包含第 ***index*** 个字符的单词的最后一个字符后面一个字符的下标。一个单词是任何连续的字母、数字、下划线或其它单个字符的集合。

`string wordstart` ***string index***

返回 ***string*** 中包含第 ***index*** 个字符的单词的首字符的下标。一个单词是任何连续的字母、数字、下划线或其它单个字符的集合。

`string match` *pattern string*

如果 *pattern* 匹配了 *string*，返回 1。*Pattern* 是一个 glob 样式模式。

示例：

```
set fullpath "/usr/home/clif/TCL_STUFF/TclTutor/Lsn.17"
set relativepath "CVS/Entries"
```

```
set directorypath "/usr/bin/"

set paths [list $fullpath $relativepath $directorypath]

foreach path $paths  {
    set first [string first "/" $path]
    set last [string last "/" $path]

    # Report whether path is absolute or relative

    if {$first != 0} {
        puts "$path is a relative path"
    } else {
        puts "$path is an absolute path"
    }

    # If "/" is not the last character in $path, report the
last word.
    # else, remove the last "/", and find the next to last
"/", and
    #   report the last word.

    incr last
    if {$last != [string length $path]} {
        set name [string range $path $last end]
        puts "The file referenced in $path is $name"
    } else {
        incr last -2;
        set tmp [string range $path 0 $last]
        set last [string last "/" $tmp]
        incr last;
        set name [string range $tmp $last end]
        puts "The final directory in $path is $name"
    }

    # CVS is a directory created by the CVS source code control
system.
    #

    if {[string match "*CVS*" $path]} {
        puts "$path is part of the source code control tree"
    }
```

```
    # Compare to "a" to determine whether the first char is
upper or lower case
    set comparison [string  compare $name "a"]
    if {$comparison >= 0} {
        puts "$name starts with a lowercase letter\n"
    } else {
        puts "$name starts with an uppercase letter\n"
    }
}
```

23 修改字符串-tolower,toupper,trim,format

这些是修改字符串的命令。注意，所有这些命令都不是就地对字符串修改。任何情况下，都会返回一个新的串。

string tolower ***string***

返回 ***string*** 中所有大写字母被转换成小写后的字符串。

string toupper ***string***

返回 ***string*** 中所有小写字母被转换成大写后的字符串。

string trim ***string?trimChars?***

返回将 ***string*** 两端所有出现的 ***trimChars*** 删除后的新串。默认情况下，***trimChars*** 是空白符（空格，制表符，换行）。注意，这些字符不会被当做字符“块”对待。换句话说，string trim ″davidw″ dw 会返回 avi，而不是 davi。

string trimleft ***string ?trimChars?***

返回将 ***string*** 左端所有出现的 ***trimChars*** 删除后的新串。默认情况下，***trimChars*** 是空白符（空格，制表符，换行）。

string trimright ***string ?trimChars?***

返回将 ***string*** 右端所有出现的 ***trimChars*** 删除后的新串。默认情况下，***trimChars*** 是空白符（空格，制表符，换行）。

format ***formatString ?arg1 arg2 ... argN?***

返回格式化字符串，方式如同 ANSI 的 sprintf 过程。***formatString*** 是使用的格式的描述。这个协议的完整定义在 format 手册里面。定义的实际使用的子集是 ***formatString*** 包含字面字母，反斜杠序列和%域。

%域是一个%加上以下一个字符：

- s... 数据是字符串
- d... 数据是十进制整数
- x... 数据是十六进制整数
- o... 数据是八进制整数
- f... 数据是浮点数

%也可以接：

- -... 对域中数据进行左对齐
- +... 对域中数据进行右对齐

要对齐的值后面可以跟一个数，表示数据最少占用的空格数（译者注：数据占用界面宽度）。

示例：

```
set upper "THIS IS A STRING IN UPPER CASE LETTERS"
set lower "this is a string in lower case letters"
set trailer "This string has trailing dots ...."
set leader "....This string has leading dots"
set both  "((this string is nested in parens )))"

puts "tolower converts this: $upper"
puts "            to this: [string tolower $upper]\n"
puts "toupper converts this: $lower"
puts "            to this: [string toupper $lower]\n"
puts "trimright converts this: $trailer"
puts "              to this: [string trimright
$trailer .]\n"
puts "trimleft converts this: $leader"
puts "             to this: [string trimleft $leader .]\n"
puts "trim converts this: $both"
puts "          to this: [string trim $both "()"]\n"

set labels [format "%-20s %+10s " "Item" "Cost"]
set price1 [format "%-20s %10d Cents Each" "Tomatoes" "30"]
set price2 [format "%-20s %10d Cents Each" "Peppers" "20"]
set price3 [format "%-20s %10d Cents Each" "Onions" "10"]
set price4 [format "%-20s %10.2f per Lb." "Steak" "3.59997"]

puts "\n Example of format:\n"
puts "$labels"
puts "$price1"
puts "$price2"
puts "$price3"
puts "$price4"
```

24 正则表达式 101

Tcl 也支持被称为正则表达式的字符串操作。多个命令可以访问这些方法通过一个“-regexp”参数。可以查看手册页，看看那些命令支出正则表达式。

也有两个显式的命令用于解析正则表达式。

regexp ***?switches? exp string ?matchVar? ?subMatch1 ... subMatchN?***

搜索 ***string*** 中匹配正则式 exp 部分。如果参数 ***matchVar*** 已给出，那么匹配正则式的子串被拷贝到 ***matchVar***。如果 ***subMatchN*** 变量存在，那么进行匹配的串的其余部分（译者注：匹配剩下的多个子串）被拷贝到 ***subMatch*** 变量，从左到右进行。

regsub ***?switches? exp string subSpec varName***

搜索匹配正则式 ***exp*** 的子串，并将它替换为 ***subSpec***。结果字符串被拷贝到 ***varName***。

正则表达式有以下几种表示规则：

^ 匹配串的开始。

$ 匹配串的结尾。

. 匹配单个字符。

***** 匹配前面任意个数（0-n）的字符（译者注：表示前面的字符出现 0-n 次）。

+ 匹配前面至少一个的任意个数（1-n）的字符（译者注：表示前面的字符出现 1-n 次）。

[...] 匹配该字符集中的任意字符。

[^...] 匹配非^后面字符集中的任意字符。

(...) 把字符集组合成一个子规格。

正则表达式与 16 节和 18 节讨论的 globbing 是相似的。主要区别是匹配的字符集处理的方式。在 globbing 里面，选择未知文本集合的唯一方式是符号“*”。它匹配任意数量的字符。

在正则表达式解析的时候，当处理符号“*”时，它会立即匹配 0 或多个该字符。例如 a*会匹配 a,aaaaa,或者空串。如果符号“*”前面紧接的字符是用方括号包含的字符集，那么符号“*”会匹配任意数量的该字符集中的字符。例如，[a-c]*会匹配 aa,abc,aabcabc，或者上面的重复及空串。

符号“+”大致上和符号“*”的行为一样，除了它要求至少匹配一个字符。例如，[a-c]+会匹配 a,abc 或者 aabcabc，但不会是空串。

正则表达式解析比 globbing 更强大。通过 globbing，你可以使用方括号来包括一组字符集，而其中每一个都是一个匹配项。正则表达式解析也包括选择不在一个集合里面的字符的方法。如果[后面第一个字符是一个“^”，那么正则表达式解释器会匹配任意非方括号所包括的字符集中的字符。一个插入符“^”可以被包括在要匹配（或不匹配）的字符集里，可以放在除了开头的任意位置。

Regexp 命令和 string match 命令相似之处在于它匹配一个 exp 而不是字符串。不同的是，它可以匹配一个串的一部分，而不是整个串，并且会把匹配的字符放在 matchVar 变量里。

如果匹配规则是包括在圆括号内的正则表达式部分，那么 regexp 会拷贝匹配的子字符集串到 subSpec 参数里。这可以被用来解析简单的字符串。

Regsub 会拷贝字符串的内容到一个新的变量，置换匹配 exp 的字符为 subSpec 里面的字符。如果 subSpec 包含“&”或者“\0”，那么这些字符会被替换成匹配 exp 的字符。如果反斜杠后面跟着一个 1-9 的数字，那么下划线序列会被替换为圆括号内 exp 适当的部分。

注意，regexp 或 regsub 的 exp 参数被 Tcl 置换过程处理。因此，这个表达式经常包含在括号里，以防止 Tcl 任何特殊的处理。

示例：

```
set sample "Where there is a will, There is a way."

#
# Match the first substring with lowercase letters only
#
set result [regexp {[a-z]+} $sample match]
puts "Result: $result match: $match"

#
# Match the first two words, the first one allows uppercase
set result [regexp {([A-Za-z]+) +([a-z]+)} $sample match
sub1 sub2 ]
puts "Result: $result Match: $match 1: $sub1 2: $sub2"

#
# Replace a word
#
regsub "way" $sample "lawsuit" sample2
puts "New: $sample2"

#
# Use the -all option to count the number of "words"
#
puts "Number of words: [regexp -all {[^ ]+} $sample]"
```

25 更多正则表达式例子

正则表达式提供非常强大的方法来定义模式，但是理解和合理地使用它们也有一些不方便的地方。所以让我们详细地检查一些例子。

我们从一个简单但很有用的例子开始：在一行文本中找出浮点数。不要担心，我们简化这个问题，不去考虑它的全部的通用性。我仅仅考虑像 1.0 和 1.00e+01 这样的数。

我们怎么为这个问题设计我们的正则表达式呢？通过检查典型的字符串的例子，我们想要匹配：

- 有效的数字是：

```
1.0, .02, +0., 1, +1, -0.0120
```

- 无效的数字（即，我们不想识别为数字但表面上看又很像的字符串）：

```
-, +., 0.0.1, 0..2, ++1
```

- 不确定的数字：

```
+0000 and 0001
```

  我们会接受它们——因为它们正常情况下被接受，因为排出它们会使我们的模式变得更复杂化。

一个模式开始出现：

- 一个以符号（-或+）或数组开头的数。这可以被表达式[-+]?捕获，它匹配一个单独的”-”号，或一个单独的”+”号，或空。
- 一个单独的句号（英文中）之前可以有 0 或多个数字的数，句号后面可以有 0 或多个数字。或许，[0-9]*\.[0-9]*合适。
- 一个可能根本不包含句号（英文中）的数。所以，修正上面的表达式为：[0-9]*?\.[0-9]*。

整个表达式是：

```
[-+]?[0-9]*\.?[0-9]*
```

这时，我们可以做三件事：

1. 用一组像上面一样的例子试验这个表达式，看看是不是合适的字符串匹配了，而其他的字符串没有匹配。
2. 让它变得更漂亮，在我们开始测试它之前。例如，”[0-9]”这种字符太常用了，所以它有一个快捷方式，”\d”。所以，我们可以接受：

   ```
   [-+]?\d*\.?\d*
   ```

   作为替代。或者我们可以决定我们想要捕获句号之前或之后的数字来进行特殊处理：

   ```
   [-+]?([0-9])*\.?([0-9]*)
   ```

3. 或者，一般情况下，那可能是一个好的策略。即，我们可以在开始真正地使用模式之前仔细检查它。

你看，上面的模式有一个问题：所有的部分都是可选的，即，每一部分可以匹配空字符串——句号前面没有正负号，没有数字，没有句号，句号后面没有数字。换句话说，我们的模式可以匹配一个空字符串！

我们的不确定的数，像”+000”会全然地被接受，我们（勉强）接受。但是，更惊讶的是，”--1”和”A1B2”也会被接受。为什么？因为模式可以从字符串的任何位置开始，所以它会分别匹配子串”-1”和”1”。

我们需要重新考虑我们的模式——它太简单了，太随意：

- 一个减号或加号之前的字符，如果有，不能是另一个数字、句号、减号和加号。我们让它为一个空格、制表符或字符串的开头：^|[\t]。
  这可能看着有点奇怪，但是它表示的是：
  要么字符串的起始（方括号外面的^）、要么|（竖直条）、要么一个空格或制表符（记住：字符串”\t”表示 tab 制表符）。
- 句号（如果有的话）前面的任意数字序列都是允许的：[0-9]+\.?
- 句号前面的数字可能有数字 0，但后面必须有至少一个数字：\.[0-9]+
- 当然，句号前面和后面的数字是：[0-9]+\.[0-9]+
- 字符串（如果有）后面的字符不能是”+”，”-”，”.”。因为那样会让我们得到不可接受的和数相像的字符串：$|[^+-.]（美元符号表示字符串末尾）

在我们试图写下完全的正则表达式之前，让我们看看可能的不同的形式：

- 没有句号：[-+]?[0-9]+
- 句号前面没有数字：[-+]?\.[0-9]+
- 句号前面有数字，可能后面也有：[-+]?[0-9]+\.[0-9]*

现在合成是：

```
(^|[ \t])([-+]?([0-9]+|\.[0-9]+|[0-9]+\.[0-9]*))($|[^+-.])
```

或者：

```
(^|[ \t])([-+]?(\d+|\.\d+|\d+\.\d*))($|[^+-.])
```

小括号是必须的，因为要用它来区分竖直条分开的选项及捕获我们想要的子串。每一对小括号也定义了一个子串，这可以被放进一个单独的变量：

```
regexp {.....} $line whole char_before number nosign
char_after

#
# Or simply only the recognised number (x's as placeholders,
the
# last can be left out
#
regexp {.....} $line x x number
```

提示：要标识这些子串，只需从左到右计算左小括号的数目。

如果我们对它进行测试：

```
set pattern
{(^|[ \t])([-+]?(\d+|\.\d+|\d+\.\d*))($|[^+-.])}
set examples {"1.0"    " .02"  "  +0."
```

```
            "1"     "+1"    " -0.0120"
            "+0000"  " - "   "+."
            "0001"   "0..2"  "++1"
            "A1.0B"  "A1"}
foreach e $examples {
    if { [regexp $pattern $e whole \
            char_before number digits_before_period] } {
        puts ">>$e<<: $number ($whole)"
    } else {
        puts ">>$e<<: Does not contain a valid number"
    }
}
```

结果是：

```
>>1.0<<: 1.0 (1.0)
>> .02<<: .02 ( .02)
>>  +0.<<: +0. ( +0.)
>>1<<: 1 (1)
>>+1<<: +1 (+1)
>> -0.0120<<: -0.0120 ( -0.0120)
>>+0000<<: +0000 (+0000)
>> - <<: Does not contain a valid number
>>+.<<: Does not contain a valid number
>>0001<<: 0001 (0001)
>>0..2<<: Does not contain a valid number
>>++1<<: Does not contain a valid number
>>A1.0B<<: Does not contain a valid number
>>A1<<: Does not contain a valid number
```

所以，我们的模式正确地接受了我们希望被识别为数字的字符串并抛弃了其他格式的字符串。

现在我们转向一些其他模式：

- 字符串里面被包围的文本：*This is "quoted text"*.如果我们预先知道包围字符（这里是双引号”），那么”([^”])*”会捕获双引号里面的文本。
  假设我们不知道包围字符(它可以是”或’)，那么：

  ```
  regexp {(["'])[^"']*\1} $string enclosed_string
  ```

  会匹配。**\1** 是所谓的反向引用第一次捕获的子串。
- 你可以使用这个方法来看一个单词是否在同一行文本中出现两次：

  ```
  set string "Again and again and again ..."
  if { [regexp {(\y\w+\y).+\1} $string => word] } {
      puts "The word $word occurs at least twice"
  }
  ```

（模式\y 匹配一个单词的起始或结束，\w+指示我们想要至少一个字符）。

- 假设你需要在一些数学表达式里面检查小括号：比如，(1+a)/(1-b*x)。一个简单的检查就是计算左括号或右括号的个数：

```
#
# Use the return value of [regexp] to count the number
of
# parentheses ...
#
if { [regexp -all {(} $string] != [regexp -all {)}
$string] } {
    puts "Parentheses unbalanced!"
}
```

当然，这只是一个粗略的检查。一个更好地检查是查看当扫描字符串时，在任意一点是否右括号比左括号要多。我们可以简单的提取括号，然后把他们放进一个列表（**-inline** 选项做这个）：

```
set parens  [regexp -inline -all {[()]} $string]
set balance 0
set change("(")  1   ;# This technique saves an
if-block :)
set change(")") -1

foreach p $parens {
    incr balance $change($p)

    if { $balance < 0 } {
        puts "Parentheses unbalanced!"
    }
}
if { $balance != 0 } {
    puts "Parentheses unbalanced!"
}
```

最后：正则表达式是非常强大的，但是它们有一些理论上的限制。其中之一就是它们并不适合解析任意嵌套的文本。

你可以使用 VisualRegexp 或 Visual REGEXP 应用程序实验正则表达式。

更多的有关正则表达式（有很多）的理论背景和实际使用可以在 J.Friedl 的 Mastering Regular Expression 这本书上面找到。

26 更多引用陷阱-正则表达式 102

regexp ***?switches? exp string ?matchVar? ?subMatch1 ... subMatchN?***

用正则表达式 ***exp*** 搜索 ***string***。如果参数 ***matchVar*** 被给出，那么匹配正则式的子串被拷贝到它里面。如果 ***subMatchN*** 变量存在，那么被匹配字符串的小括号部分被拷贝到 ***subMatch*** 变量，从左到右。

regsub ***?switches? exp string subSpec varName***

用正则表达式 ***exp*** 搜索 ***string*** 中匹配的子串，并把它们替换为 ***subSpec***。结果串被拷贝到 ***varName***。

这两个正则式解析命令里面的正则式（exp）在 Tcl 置换阶段被 Tcl 解析器求值。这可以提供很多力量，也需要格外小心。

这些示例演示了正则表达式求值的一些复杂方面。这些每一个示例里面的字段讨论起来格外痛苦而冗长。

当你阅读示例时，需要记住的几点是：

- 左方括号对置换阶段和正则表达式解析器有意义。
- 一对小括号，一个加号，一个星号对正则表达式有意义，而对 Tcl 置换阶段没有意义。
- 反斜杠序列（\n，\t，等）对 Tcl 置换阶段有意义，而对正则表达式解析器没有意义。
- 反斜杠转义字符(\[)对 Tcl 置换阶段和正则表达式解析器没有特别意义。

字符在哪个阶段有意义影响匹配你希望匹配的字符所必需的转义的次数。一次转义可以是包括在花括号里，也可以在转义的字符前面加一个反斜杠。

传递一个左括号给正则表达式解析器来对一系列字符进行求值需要一次转义。让正则表达式解析器匹配一个字面左括号需要 2 次转义（一次在 Tcl 置换阶段对括号进行转义，一次在正则表达式解析里面对括号进行转义）。如果你的字符串被置于引号内，那么如果你想传给正则表达式解析器一个反斜杠，它也必须用一个反斜杠转义。

注意：你可以拷贝代码，在 tclsh 里面执行，看看效果。

示例：

```
#
# Examine an overview of UNIX/Linux disks
#
set list1 [list \
{/dev/wd0a       17086    10958     5272    68%    /}\
{/dev/wd0f      179824   127798    48428    73%    /news}\
```

```
{/dev/wd0h     1249244   967818   218962    82%    /usr}\
{/dev/wd0g       98190    32836    60444    35%    /var}]

foreach line $list1 {
    regexp {[^ ]* *([0-9]+)[^/]*(/[a-z]*)} $line match size
mounted;
    puts "$mounted is $size blocks"
}



#
# Extracting a hexadecimal value ...
#
set line {Interrupt Vector?    [32(0x20)]}
regexp "\[^\t]+\t\\\[\[0-9]+\\(0x(\[0-9a-fA-F]+)\\)]"
$line match hexval
puts "Hex Default is: 0x$hexval"

#
# Matching the special characters as if they were ordinary
#
set str2 "abc^def"
regexp "\[^a-f]*def" $str2 match
puts "using \[^a-f] the match is: $match"

regexp "\[a-f^]*def" $str2 match
puts "using \[a-f^] the match is: $match"

regsub {\^} $str2 " is followed by: " str3
puts "$str2 with the ^ substituted is: \"$str3\""

regsub "(\[a-f]+)\\^(\[a-f]+)" $str2 "\\2 follows \\1" str3
puts "$str2 is converted to \"$str3\""
```

27 关联数组

C、BASIC、FORTRAN 和 Java 这样的语言都支持数组，其下标的值是一个整数。Tcl 里面，就像大部分脚本语言（Perl，Python，PHP 等等）一样，支持关联数组（也称为“哈希”hash 表），而其下标值是字符串。

关联数组的语法是把下标放在圆括号里：

```
set name(first) "Mary"
set name(last)  "Poppins"

puts "Full name: $name(first) $name(last)"
```

除了简单的访问和创建数组的命令会在下一节讨论之外，这里有几个数组命令。

array exists ***arrayName***

如果 ***arrayName*** 存在，返回 1. 如果 ***arrayName*** 是一个标量，proc 过程或不存在，返回 0.

array names ***arrayName ?pattern***

返回关联数组 ***arrayName*** 索引的列表。如果 ***pattern*** 被提供了，那么只有匹配 ***pattern*** 的索引才会被返回。匹配不使用 globbing 的 string match 技术。

array size ***arrayName***

返回数组 ***arrayName*** 的元素个数。

array get ***arrayName***

返回一个列表，其中第奇数（1,3,5 等等）个成员是关联数组的索引，而名称后面跟着的就是该索引对应的数组成员的值。（译者注：2,4,6 等为 1,3,5 对应的数组元素的值）

array set ***arrayName dataList***

把一个列表转换成一个关联数组。***dataList*** 是一个 array get 返回格式的列表。即每第奇数个列表成员（1,3,5 等）是关联数组的索引，而每一个索引值后面跟的是该数组元素的值。

array unset ***arrayName ? pattern?***

复原数组所有元素。如果 ***pattern*** 存在，那只有匹配该模式的元素被复原。

当关联数组的名称作为 global 命令的参数给出时，那该关联数组所有的元素对该 proc 可用。出于这个原因，*Brent Welch* 建议（在 *Practical Programming in Tcl and Tk* 里面），在包里，使用关联数组来表示状态结构。

这个方法使在许多相互协作的 proc 过程间共享数据更容易，并且不会像对所有的共享数据项都使用独立的全局变量那样破坏全局命名空间。

另一个数组的常用的作用是存储表数据。下面的例子里面，我们会使用数组存储简单的名称数据库。

示例：

```
proc addname {first last} {
    global name

    # Create a new ID (stored in the name array too for easy
access)

    incr name(ID)
    set id $name(ID)

    set name($id,first) $first   ;# The index is simply a
string!
    set name($id,last)  $last    ;# So we can use both fixed
and
                               ;# varying parts
}

#
# Initialise the array and add a few names
#
global name
set name(ID) 0

addname Mary Poppins
addname Uriah Heep
addname Rene Descartes
addname Leonardo "da Vinci"

#
# Check the contents of our database
# The parray command is a quick way to
# print it
#
parray name
```

```
#
# Some array commands
#
array set array1 [list {123} {Abigail Aardvark} \
                   {234} {Bob Baboon} \
                   {345} {Cathy Coyote} \
                   {456} {Daniel Dog} ]

puts "Array1 has [array size array1] entries\n"

puts "Array1 has the following entries: \n [array names
array1] \n"

puts "ID Number 123 belongs to $array1(123)\n"

if {[array exist array1]} {
    puts "array1 is an array"
} else {
    puts "array1 is not an array"
}

if {[array exist array2]} {
    puts "array2 is an array"
} else {
    puts "array2 is not an array"
}

proc existence {variable} {
    upvar $variable testVar
    if { [info exists testVar] } {
      puts "$variable Exists"
    } else {
      puts "$variable Does Not Exist"
    }
}

# Create an array
for {set i 0} {$i < 5} {incr i} { set a($i) test }

puts "\ntesting unsetting a member of an array"
existence a(0)
puts "a0 has been unset"
unset a(0)
existence a(0)
```

```
puts "\ntesting unsetting several members of an array, with
an error"
existence a(3)
existence a(4)
catch {unset a(3) a(0) a(4)}
puts "\nAfter attempting to delete a(3), a(0) and a(4)"
existence a(3)
existence a(4)

puts "\nUnset all the array's elements"
existence a
array unset a *

puts "\ntesting unsetting an array"
existence a
puts "a has been unset"
unset a
existence a
```

28 更多数组命令-枚举及其过程中的用法

通常，你会想要循环遍历关联数组的内容-而不必明确地指定这些元素。从这一点来说，array names 和 array get 命令非常有用。你可以用它们两个通过模式（glob 样式）来选择你需要的元素。

```
foreach name [array names mydata] {
    puts "Data on \"$name\": $mydata($name)"
}

#
# Get names and values directly
#
foreach {name value} [array get mydata] {
    puts "Data on \"$name\": $value"
}
```

但是，注意，元素不会以任何可预知的顺序返回：这和底层的 hash 表有关系。如果你想要按照特定的顺序（比如，字母顺序），使用下面这样的代码：

```
foreach name [lsort [array names mydata]] {
    puts "Data on \"$name\": $mydata($name)"
}
```

尽管数组出于某些原因用作一个存储工具会非常棒，但当你把它们传给一个过程时，还是有点复杂的：它们实际上是变量的集合。下面的不会起作用：

```
proc print12 {a} {
   puts "$a(1), $a(2)"
}

set array(1) "A"
set array(2) "B"

print12 $array
```

原因很简单：数组并没有一个值。作为替代，上面的代码应该写成：

```
proc print12 {array} {
   upvar $array a
   puts "$a(1), $a(2)"
}

set array(1) "A"
set array(2) "B"
```

```
print12 array
```

所以，不应该传递一个“值”给数组，你应该传递名称。这样就另命名（通过 upvar 命令）了一个局部变量（行为和原始数组一样）。你可以用这种方式对原始的数组作出修改。

示例：

```
#
# The example of the previous lesson revisited - to get a
# more general "database"
#

proc addname {db first last} {
    upvar $db name

    # Create a new ID (stored in the name array too for easy
access)

    incr name(ID)
    set id $name(ID)

    set name($id,first) $first   ;# The index is simply a
string!
    set name($id,last)  $last    ;# So we can use both fixed
and
                               ;# varying parts
}

proc report {db} {
    upvar $db name

    # Loop over the last names: make a map from last name
to ID

    foreach n [array names name "*,last"] {
        #
        # Split the name to get the ID - the first part of
the name!
        #
        regexp {^[^,]+} $n id

        #
        # Store in a temporary array:
        # an "inverse" map of last name to ID)
        #
```

```
        set last      $name($n)
        set tmp($last) $id
    }

    #
    # Now we can easily print the names in the order we want!
    #
    foreach last [lsort [array names tmp]] {
        set id $tmp($last)
        puts "   $name($id,first) $name($id,last)"
    }
}

#
# Initialise the array and add a few names
#
set fictional_name(ID) 0
set historical_name(ID) 0

addname fictional_name Mary Poppins
addname fictional_name Uriah Heep
addname fictional_name Frodo Baggins

addname historical_name Rene Descartes
addname historical_name Richard Lionheart
addname historical_name Leonardo "da Vinci"
addname historical_name Charles Baudelaire
addname historical_name Julius Caesar

#
# Some simple reporting
#
puts "Fictional characters:"
report fictional_name
puts "Historical characters:"
report historical_name
```

29 字典-数组的替代方案

Tcl 数组是变量的集合，而不是值的集合。在一些条件下，这有优势（例如，你可以用在它们上面使用变量跟踪），但同时也有一些缺点：

- 它们不能直接以值得形式传递给一个过程。相反，你必须使用 array get 和 arrayset 命令来讲它们转换为值，然后再转换回来，或者使用 upvar 命令来创建该数组的别名。
- 多维数组（即，下标包含两个或多个部分的数组）必须仿效下面的构造：

```
set array(foo,2) 10
set array(bar,3) 11
```

  这里使用的逗号不是一种特别的语法，相反，只是字符串关键字的一部分。通过像”foo,2”和”bar,3”这样的关键字，我们正在使用一维的数组。这非常简单，但也可能变得非常笨拙（例如，没有可插入的空间）。
- 数组不可以被包含在其他数据结构里面，如列表，也不能不经过包装和解包进字符串就进行信道的发送。

在 Tcl8.5 里面，dict 命令被引入。这提供有效的方式来访问 key-value 对。就像数组一样，只不过字典全部是值。这意味着你可以传递它们给一个过程，以列表或字符串的形式，而不需要 dict 命令的帮助。Tcl 字典因此更像 Tcl 的列表，除了字典表示的是键到值得映射，而不是列表的有序的序列。

不像数组，你可以嵌套字典，这样某一个特定的键的值可以包含另一个字典。这样，你可以优美地（elegantly）创建复杂的数据结构，像层次数据库。你也可以把字典和其他 Tcl 数据结构联合起来。例如，你可以创建一个字典的列表，而每个字典又包含列表。

这是一个示例（改编自手册页面）：

```
#
# Create a dictionary:
# Two clients, known by their client number,
# with forenames, surname
#
dict set clients 1 forenames Joe
dict set clients 1 surname   Schmoe
dict set clients 2 forenames Anne
dict set clients 2 surname   Other

#
# Print a table
#
puts "Number of clients: [dict size $clients]"
dict for {id info} $clients {
    puts "Client $id:"
    dict with info {
```

```
        puts "   Name: $forenames $surname"
    }
}
```

示例中发生的是：

- 我们填充了一个字典，叫 *clients*，包含我们关于两个客户端的信息。字典有两个键，”1”和”2”，而且这些键的值本身又是一个（嵌套）字典——包含两个键”forenames”和”surname”。Dict set 命令接受一个键名称的列表来当做字典的路径（path）。命令最后的参数是我们想要设置的值。你可以提供无论多少的键参数给 dict set 命令，这会导致任意的复杂的嵌套数据结构。但是要小心。直截了当的数据结构设计对于大部分问题来说通常比嵌套的要好。
- dict for 命令然后循环遍历字典里的每一个键和值对（尽以最外层的水平）。Dict for 实际上是 foreach 的针对字典的特殊化版本。我们也可以如下这样写：

  ```
  foreach {id info} $clients { ... }
  ```

  这利用了这个事实，即，在 Tcl 中，每一个字典也是一个有效的 Tcl 列表，它包含表示字典内容的名称和值对的序列。但是，当处理字典的时候，dict for 命令还是首选的。因为它不仅更有效，而且使我们处理字典而不仅仅是列表的代码对于读者来说更清晰。
- 为了得到存储着 client ID 字典里的实际的值，我们使用 dict with 命令。这个命令获得这个字典，然后把它解压成一组当前作用域下的局部变量。例如，在我们的例子里，每次外层循环的 info 变量会包含一个含有两个键：”forenames”和”surname”的字典。Dict with 命令以与键和与字典中关联的值同名方式解压这些键到局部变量。这使我们当访问值的时候可以使用更方便的语法，而不必使用到处 dict get。一个相关的命令是 dict update 命令，它允许你准确指定你想要转化为变量的那些键。你要知道，你对这些变量做出的任何改变在 dict with 命令完成的时候都会被拷贝回字典。

Dict for 循环返回字典中元素的顺序被定义为键被添加进字典的时间顺序。如果你需要以特定顺序访问这些键，那么建议你首先对这些键进行显式地排序。例如，为了按字母顺序，基于键地检索字典所有的元素，我们可以使用 lsort 命令：

```
foreach name [lsort [dict keys $mydata]] {
    puts "Data on \"$name\": [dict get $mydata $name]"
}
```

示例：

在这个示例中，我们转换前面课程的简单数据库来处理字典，而不是数组。

```
#
# The example of the previous lesson revisited - using dicts.
#

proc addname {dbVar first last} {
```

```
    upvar 1 $dbVar db

    # Create a new ID (stored in the name array too for easy
access)
    dict incr db ID
    set id [dict get $db ID]

    # Create the new record
    dict set db $id first $first
    dict set db $id last  $last
}

proc report {db} {

    # Loop over the last names: make a map from last name
to ID

    dict for {id name} $db {
        # Create a temporary dictionary mapping from
        # last name to ID, for reverse lookup
        if {$id eq "ID"} { continue }
        set last      [dict get $name last]
        dict set tmp $last $id
    }

    #
    # Now we can easily print the names in the order we want!
    #
    foreach last [lsort [dict keys $tmp]] {
        set id [dict get $tmp $last]
        puts "   [dict get $db $id first] $last"
    }
}

#
# Initialise the array and add a few names
#
dict set fictional_name ID 0
dict set historical_name ID 0

addname fictional_name Mary Poppins
addname fictional_name Uriah Heep
addname fictional_name Frodo Baggins
```

```
addname historical_name Rene Descartes
addname historical_name Richard Lionheart
addname historical_name Leonardo "da Vinci"
addname historical_name Charles Baudelaire
addname historical_name Julius Caesar

#
# Some simple reporting
#
puts "Fictional characters:"
report $fictional_name
puts "Historical characters:"
report $historical_name
```

注意，在这个示例中，我们以两种不同的方式使用字典。在 addname 过程里面，我们传递字典变量名，使用 upvar 来连接它，正如我们前面对数组所做的那样。我们这样做，这样数据库的改变被反映在正在调用的作用域里面，而不需要返回一个新的字典值（试着改变代码，不使用 upvar）。但是，在 report 过程里面，我们以值传递字典，并直接使用它。比较字典和数组的示例代码（上一课），看看两种数据结构之间及用法的不同。

30 文件访问 101

Tcl 提供多个方法来对磁盘上的文件进行读和写。最简单的访问文件的方法就借助 gets 和 puts。但是，当有很多数据去读得时候，有时，使用 read 命令来加载整个文件，然后通过 split 命令将文件解析成行会更有效。

这些方法也可以用于 socket 和管道之间的通信。甚至可以，借助所谓的“虚拟文件系统”使用内存里面的文件而不是硬盘上的。Tcl 提供一个几乎统一的接口给这些不同的资源，这样一般来说你不需要自己关心细节。

`open` ***`fileName ?access? ?permission?`***

打开一个文件，并返回标记（token），当借助 gets，puts，close 等访问文件时使用。

- ***`fileName`*** 是要打开的文件名。
- ***`access`*** 是文件访问的模式
  - r…….打开文件读。文件必须已经存在。
  - r+…….打开文件读和写。文件必须已经存在。
  - w……..打开文件写。如果文件不存在就创建之，如果存在设置文件长度为 0.
  - w+……打开文件读写。如果文件不存在就创建之，如果存在设置文件长度为 0.
  - a……..打开文件写。文件必须已经存在。设置当前位置为文件末尾。
  - a+……打开文件写。文件不存在，创建之。设置当前位置为文件末尾。
- ***`permission`*** 是一个整数，用来设置文件的访问权限。默认是 `rw-rw-rw-(0666)`。你可以用它来设置权限的文件所有者、其所属的组以及其他用户。对于很多应用来说，默认是最好的。

`close` ***`fileID`***

关闭前面用 `open` 打开的文件，并从缓存里排出任何剩余的输出。

`gets` ***`fileID ?varName?`***

读入 ***`fileID`*** 该输入文件的一行，忽略结束换行符（`newline`）。

如果存在 ***`varName`*** 参数，`gets` 返回读取的字符数（如果文件结束，返回

-1），并把读取的输入行放在 ***varName*** 里面。

如果 ***varName*** 没有指定，gets 返回输入行。如果出现以下情况，返回空字符串：

- 文件中存在空行
- 当前位置是文件末尾。（EOF）

puts ?-nonewline? ***?fileID? string***

将 ***string*** 里面的字符写入 ***fileID*** 引用的流，***fileID*** 可以是：

- 带有写访问权限的 open 调用的返回值。
- Stdout
- Stderr

read ***?-nonewline? fileID***

读取 ***fileID*** 中全部剩余的字节，并返回这个字符串。如果***-nonewline*** 被设置，那么如果行尾时换行（newline），其将被忽略。任何存在的文件结束条件在 read 命令执行之前都会被清除。

read ***fileID numBytes***

从 ***fileID*** 读取至多 ***numBytes*** 个字节，并以 Tcl 字符串的形式返回输入。任何已存在的文件结束条件在 read 执行之前都会被清除。

seek ***fileID offset ?origin?***

改变 ***fileID*** 引用的文件的当前位置。注意，如果文件以 a 的访问权限打开，那么当前位置不能被设置为文件结束之前的位置进行写操作，但是可以被设置为文件的开始进行读操作。

- ***fileID*** 可以是：
    - 一个由 open 返回的文件标识符
    - Stdin
    - Stdout
    - Stderr
- ***Offset*** 是一个当前将要被设置的位置的偏移字节数。偏移起始测量位置默认为文件开始，但也可以从当前位置开始，或者通过适当设置 ***origin*** 为文件末尾。
- ***Origin*** 是 ***offset*** 开始测量的起始位置。它默认是文件开始。***Origin***

必须是：

* Start……. ***Offset*** 从文件开始进行测量。
* current……. ***Offset*** 从文件当前位置开始测量。
* end……. ***Offset*** 从文件末尾开始测量。

`tell` ***fileID***

以十进制字符串形式返回 ***fileID*** 文件访问指针的位置。

`flush` ***fileID***

排出任何 ***fileID*** 文件的缓存输出。

`eof` ***fileID***

如果 EOF 条件成立，返回 1，否则返回 0.

Tcl 文件访问需要记住的几点：

- 文件 I/O 会被缓存。尽管你期望输出被发送，但其实它可能没有被发送。当你的出现正常退出时，文件会全部被关闭并冲刷干净（缓存）。但如果出现被意外终止，可能只会被关闭（而不会被冲刷干净）。
- 可用的打开文件槽个数是有限的。如果你期望程序打开多个文件，记住操作完之后关闭它们。
- 空的行和 EOF 用以下命令是不可区分的：
  `set string [gets filename]`
  使用 eof 命令来确定文件是否位于末尾，或者使用 gets 的其他形式（参看示例）。
- 你不能重写任何以 a 访问权限打开的文件中的数据。但是，你可以找到文件的开始用于 gets 命令。
- 以 w+权限打开文件允许你重写数据，但会删除全部现有的文件中的数据。
- 以 r+权限打开文件允许你重写数据，同时保存文件现有的数据。
- 默认情况下，命令假设字符串表示“可读”的文本。如果你想要读取“二进制”文件，你必须使用 fconfigure 命令。
- 通常，尤其当你在你的程序里处理配置数据时，你可以使用 source 命令而不是这里展示的相对底层的命令。务必确信你的数据被当做 Tcl 命令解释，然后再 source 这个文件。

示例：

```
#
# Count the number of lines in a text file
#
set infile [open "myfile.txt" r]
set number 0
```

```
#
# gets with two arguments returns the length of the line,
# -1 if the end of the file is found
#
while { [gets $infile line] >= 0 } {
    incr number
}
close $infile

puts "Number of lines: $number"

#
# Also report it in an external file
#
set outfile [open "report.out" w]
puts $outfile "Number of lines: $number"
close $outfile
```

31 文件信息-file,glob

有两个命令提供文件系统的信息，glob 和 file。

Glob 提供一个路径下所有文件名称的访问。它使用一个类似于 UNIX **ls** 命令和 Windows（DOS）**dir** 命令的名称匹配机制，并返回符合模式匹配的名称列表。

File 提供三组功能:

- 适用于文件名称解析的字符串操作
  - Dirname …… 返回 path 的路径部分。
  - Extension…… 返回文件名称扩展。
  - Join ……. 连接路径和文件名为一个字符串。
  - Nativename… 任何文件或路径的本地名称。
  - Rootname …… 返回没有扩展名的文件名称。
  - Split ……. 将字符串拆成路径和文件名。
  - Tail …… 返回没有路径的文件名。
- 关于目录里面条目的信息
  - Atime ……. 返回上次访问时间。
  - Executable……. 如果文件可被用户执行，返回 1.
  - Exists …….. 如果文件存在，返回 1.
  - Isdirectory…….. 如果条目是一个目录，返回 1.
  - Isfile …….. 如果条目是正常的文件，返回 1.
  - Lstat …….. 发那好文件的状态信息的数组。
  - Mtime ……. 返回上次数据修改的时间。
  - Owned …….. 如果文件为该用户所有，返回 1.
  - Readable …….. 如果文件可被该用户读，返回 1.
  - Readlink …….. 返回符号链接指向的文件名称。

    - Size ..... 返回文件大小字节数。
    - Stat ..... 返回文件状态信息的数组。
    - Type ..... 返回文件类型。
    - Writable ..... 如果文件可被该用户写，返回 1.
- 操作文件和目录自身
    - Copy ..... 拷贝文件或路径
    - Delete ..... 删除文件或路径
    - Mkdir ..... 创建新目录
    - Rename ..... 重命名或移动一个文件或目录

通过这两个命令，一个程序可以获得大部分需要的和操作文件和路径的信息。

尽管检索文件呈现的和它们拥有的属性信息通常是一件平台高度依赖的事情，但是 Tcl 提供了一个隐藏几乎所有特定于平台（而与编程人员无关）的细节的接口。

为了利用这个特征，那就总是优先考虑通过 file join ，file split 和其他命令来操作文件名。

例如，引用上一个目录里面的文件：

```
set upfile [file join ".." "myfile.out"]
# upfile will have the value "../myfile.out"
```

“..”指示“父目录”。

因为外部命令可能不会很得体地处理 Tcl 使用的统一表示，所以 Tcl 也提供一个命令将字符串转化为本机字符串：

```
#
# On Windows the name becomes "..\myfile.out"
#
set newname [file nativename [file join ".." "myfile.out"]]
```

检索当前目录下所有扩展名为”.tcl”的文件：

```
set tclfiles [glob *.tcl]
puts "Name - date of last modification"
foreach f $tclfiles {
    puts "$f - [clock format [file mtime $f] -format %x]"
}
```

（clock 命令把 file time 命令返回的秒数转化为一个简单的日期字符串，如”12/22/04”）

glob ***?switches? pattern ?patternN?***

返回匹配 ***pattern*** 或 ***patternN*** 的文件名称列表。

Switches 可能的取值为（有更多的开关可用）：

- ***-nocomplain***
  允许 glob 返回空的列表而不会产生错误。没有这个标记，当返回空列表时会产生错误。
- ***-types typeList***

  选择命令应该返回的文件或路径的类型。***typeList*** 可以包含类型字母，

  像"d"表示路径，"f"表示普通文件，以及表示用户权限的字母和关键字

  （如，"r"表示文件或目录可读）。
- --

  标志开关的结束。这允许在模式(pattern)中使用"-"而不会引起 glob 解释器的误解。

Pattern 遵守和 sring 匹配 globbing 规则相同的匹配规则，除了下面这些情况：

- **{a,b,...}** 匹配任何字符串 a,b 等。
- 文件名开头的"."必须在文件名中匹配一个"."。如果不是名称的第一个字符，那这个"."只是一个通配符。
- 所有的"/"必须准确匹配
- 如果 ***Pattern*** 前两个字符是**~/**,那么~会替换成 **HOME** 环境变量的值。
- 如果 ***Pattern*** 第一个字符是~，跟着后面是登录 id，那么~loginid 会替换成登录账号的 home 目录。

注意，***Pattern*** 匹配的文件名列表会以任意顺序返回（即，比如，不要期望它们会按照字母排序）。

file ***atime name***

返回上次访问 ***name*** 文件的始于一些依赖于系统的起始日期的秒数，也被称为“epoch”（通常是 1/1/1970）。

file ***copy ?-force? name target***

拷贝文件或目录 ***name*** 到一个新文件 ***target***(或这个名称的目录)。***-force*** 开关允许你重写已经存在的文件。

file ***delete ?-force? name***

删除文件/目录 ***name***。***-force*** 开关允许你删除非空目录。

file ***dirname name***

返回路径/文件名的目录部分字符串。如果 ***name*** 不包含斜杠，file dirname 返回”.”。如果 name 中最后的”/”也是第一个字符，那么返回”/”。

file ***executable name***

如果 ***name*** 对于当前用户是可执行的，返回 1,；否则返回 0.

file ***exists name***

如果文件 ***name*** 存在，并且用户有所有路径下对该文件的搜索权限，返回 1；否则返回 0.

file ***extension name***

返回文件扩展名。

file ***isdirectory name***

如果文件是一个目录，返回 1；否则返回 0.

file ***isfile name***

如果文件名是正常的文件，返回 1；否则返回 0.

file ***lstat name varName***

这个命令返回和系统调用 lstat 相同的信息。结果被放在关联数组 ***varName*** 里面。***varName*** 数组的下标是：

- Atime …… 上次访问时间
- Ctime …… 上次文件状态改变时间
- Dev …… inode 设备
- Gid …… 文件组的组 id
- Ino …… inode 号码
- Mode …… inode 保护模式
- Mtime …… 上次数据修改时间
- Nlink …… 物理连接数
- Size …… 文件字节大小
- Type …… 文件类型
- Uid …… 文件所有者的用户 id

因为这会调用 **lstat**，如果 ***name*** 是一个符号链接，***varName*** 的值会指向这

个链接，而不是链接的文件。（查看 stat 子命令）

`file` ***mkdir name***

创建新目录 ***name***。

`file` ***mtime name***

不管使用的系统其实日期是什么，都返回自从 1970 年 1 月 1 日以来的上次修改时间。

`file` ***owned name***

如果文件被当前用户拥有，返回 1；否则返回 0.

`file` ***readable name***

如果当前用户对该文件拥有读权限，返回 1；否则返回 0.

`file` ***readlink name***

返回符号链接指向的文件名称。如果 ***name*** 不是符号链接，或者不可读，产生错误。

`file` ***rename ?-force? name target***

重命名文件/目录 ***name*** 为新的名称 ***target***(或者也可以是已经存在的目录名称)。开关***-force*** 允许你对已经存在的文件进行重写。

`file` ***rootname name***

返回 ***name*** 里面所有的字符，除了最后的”.”。如果 ***name*** 不包含”.”，返回***$name***。

`file` ***size name***

返回 ***name*** 的字节大小。

`file` ***stat name varName***

返回和系统调用 **stat** 相同的信息。结果被放置在关联数组 ***varName***。其中下标是：

- Atime …… 上次访问时间
- Ctime …… 上次文件状态改变时间
- Dev …… inode 设备
- Gid …… 文件组的组 id
- Ino …… inode 号码
- Mode …… inode 保护模式
- Mtime …… 上次数据修改时间
- Nlink …… 物理连接数

- Size ..... 文件字节大小
- Type ..... 文件类型
- Uid ..... 文件所有者的用户 id

file ***tail name***

返回 ***name*** 最后一个斜杠之前的全部字符。如果 ***name*** 不包含斜杠，返回 name。

file ***type name***

返回表示文件类型的字符串，包括：

- file ..... 普通的文件
- directory ..... 目录
- characterSpecial ..... 面向字符的设备
- blockSpecial ..... 面向块的设备
- fifo ..... 命名管道
- link ..... 符号链接
- socket ..... 命名 socket

file ***writable name***

如果当前用户对该文件拥有写权限，返回 1；否则返回 0.

注意：上面给出的概述并不包含各种子命令的全部细节，而且也没有列出全部的子命令。请查看它们的手册页面。

示例：

```
#
# Report all the files and subdirectories in the current
directory
# For files: show the size
# For directories: show that they _are_ directories
#

set dirs [glob -nocomplain -type d *]
if { [llength $dirs] > 0 } {
    puts "Directories:"
    foreach d [lsort $dirs] {
        puts "    $d"
    }
} else {
```

```
    puts "(no subdirectories)"
}

set files [glob -nocomplain -type f *]
if { [llength $files] > 0 } {
    puts "Files:"
    foreach f [lsort $files] {
        puts "    [file size $f] - $f"
    }
} else {
    puts "(no files)"
}
```

32 执行 Tcl 的其他程序-exec,open

截至目前，课程已经完成了在 Tcl 解释器里面的编程工作。然而，作为一种脚本语言，用 Tcl 来把其他包或程序捆绑在一起也是很有用的。为了完成这个功能，Tcl 有两种方法来启动另一个程序。

- open …… 运行一个带有连接到文件描述符的 I/O 的新的程序。
- exec …… 以子进程的身份运行一个新的程序。

Open 调用和打开文件是同一个调用。如果文件名参数中第一个字符是”pipe”符号（/），那么 open 会把参数剩余的部分当做一个程序名，并用连接到这个文件描述符的标准的输入输出方式运行这个程序。”pipe”连接可以被用来读取那一个程序的输出，或者往那个程序里写新的数据，或者同时进行两者。

如果”pipe”被打开进行读和写，你必须了解，管道是缓存的。Puts 命令的输出会被保存在一个 I/O 缓存里，直到缓存变满或者你执行 flush 命令强迫缓存被传输给那一个程序。那一个程序的输出对于 read 或 gets 或许不可用直到它的输出缓存被填充或显式地 flush 清空出去。

（注意：由于这对于另一个程序来说是内部操作，所以你的 Tcl 脚本没办法影响它。另一个程序只是简单地必须配合。但是，这也不是绝对正确：expect 扩展实际上解决了这个限制，通过利用深层的系统特征。）

Exec 调用和在一个交互式 shell、DOS 窗口或 UNIX/Linuxshell 脚本提示下面调用一个程序（或者一组通过管道连接在一起的程序）是相似的。它支持几种样式的输出重定向，或者它可以返回其他程序的输出，作为 exec 调用的返回值。

open ***|progName ?access?***

返回管道的文件描述符。***progName*** 参数必须以管道符号开始。如果 ***progName*** 被包括在引号或花括号内，它可以包含子进程的参数。

exec ***?switches? arg1 ?arg2? … ?argN?***

exec 把它的参数当作将要运行的一组程序的名称及参数。如果 ***args*** 以”-”起始，那么它们会被当作 exec 命令的 ***switches*** 选项，而不是当作子进程或子进程选项被调用。

Switches 是：

-keepnewline

在管道行输出中返回一个尾部的换行（newline）。通常，尾部的换行（newline）会被删除。

--

标志 switches 的结束。接下来的字符串会被当做 arg1，哪怕它以”-”开始。

arg1 . . . argN 可以是：

- 要执行的程序的名称
- 子进程的命令行参数
- I/O 重定向指令
- 使新的程序后台运行的指令：

```
exec myprog &
```

会在后台启动程序 myprog，并立即返回。在这个程序和 Tcl 脚本之间没有连接，它们都可以独立运行。
“&”必须是最后一个参数——你可以在其之前使用所有各种类型的参数。
[**NOTE:** add information on how to wait for the program to finish?]（译者注：似乎最终作者也没有将如何等待启动的程序完成的信息添加进来。）

有很多 I/O 重定向命令。这些命令的主要的子集是：

|

将管道符号前面的命令的标准输出与管道符号后面的命令的标准输入进行管道化连接。

< filename

管道里面的第一个程序会从 ***filename*** 读取输入。

<@ fileID

管道里面的第一个程序会从 Tcl 描述符 fileID 读取输入。fileID 是 open … “r”命令的返回值。

<< value

管道里面的第一个程序会读取 ***value*** 作为它的输入。

> filename

管道里最后一个程序的输出会被发送给 ***filename***。***Filename*** 任何之前的内容都会丢失。

>> filename

管道里最后一个程序的输出会被追加（append）到 ***filename***。

2> filename

管道里所有程序的标准错误（stderr）会被发送给 ***filename***。***filename*** 任何之前的内容都会丢失。

2>> filename

管道里所有程序的标准错误（stderr）会被追加（append）到 ***filename***。

>@ fileID

管道里最后一个程序的输出会被写进 ***fileID*** 里面。***fileID*** 是 open … "w"的返回值。

如果你熟悉 shell 编程，那么当你写 Tcl 脚本使用 exec 和 open 调用的时候有一些不同你需要了解。

- 你不需要引号把参数引起来来对它们转义，以防止 shell 展开它们。在示例中，sed 命令的参数并没有放在引号里面。如果它被写在引号里，引号会被传给 sed，而不是被去除掉（像 shell 那样），并且 sed 会报告一个错误。
- 如果你使用 open |*cmd "r+"* 句法，你必须在每一个 puts 后面跟一个 flush 来强制 Tcl 发送自身缓存中的命令给程序。程序自身的输出也可能被缓存在它的输出缓存里面。
  有时你可以强制外部程序的输出全部 flush 清空，通过给这个进程发送 exit 命令。
  你也可以 fconfigure 命令来取消连接的缓存。
  就像已经提过的，expect 的 Tcl 扩展提供一个更好的接口给其他程序，特别是处理缓存问题。
  [**NOTE:** add good reference to expect]（译者注：很明显，最后作者也没有添加 expect 的参考。）
- 如果 open /cmd 其中的一个命令失败，open 并不返回错误。然而，通过 gets $file 尝试从文件描述符里面读取输入会返回一个空的字符串。使用 gets $file input 句法会返回一个字符计数-1.
- Tcl 并不像 UNIX/Linux shell 那样展开文件名。所以：

```
exec ls *.tcl
```

会失败——基本不可能有字面值为"*.tcl"这样的文件。

如果你需要展开，你可以使用 glob 命令：

```
eval exec ls [glob *.tcl]
```

或者，自 Tcl8.5 之前：

```
exec ls {expand}[glob *.tcl]
```

{expand}前缀被用来强制列表作为独立的参数。

- 如果 exec 调用其中一个命令执行失败，exec 会返回一个错误，错误输出会包括最后的一行来描述这个错误。

  Exec 把任何标准错误的输出当做外部程序失败的指示。这是一个保守的假设：很多程序那样运行，它们设置错误代码是很不严谨的。

  然而，一些程序往标准错误（stderr）进行写入时并没有把它预计为一个错误的饿指示。你可以通过使用 catch 命令来防止你的脚本崩溃。

```
if { [catch { exec ls *.tcl } msg] } {
   puts "Something seems to have gone wrong but we will
ignore it"
}
```

为了监视程序的返回值，可能的失败原因，你可以使用全局的 errorInfo 变量：

```
if { [catch { exec ls *.tcl } msg] } {
   puts "Something seems to have gone wrong:"
   puts "Information about it: $::errorInfo"
}
```

示例：

```
#
# Write a Tcl script to get a platform-independent
program:
#
# Create a unique (mostly) file name for a Tcl program
```

```
set TMPDIR "/tmp"
if { [info exists ::env(TMP)] } {
    set TMPDIR $::env(TMP)
}
set tempFileName "$TMPDIR/invert_[pid].tcl"

# Open the output file, and
# write the program to it

set outfl [open $tempFileName w]

puts $outfl {
    set len [gets stdin line]
    if {$len < 5} {exit -1}

    for {set i [expr {$len-1}]} {$i >= 0} {incr i -1} {
        append l2 [string range $line $i $i]
    }
    puts $l2
    exit 0
}

# Flush and close the file
flush $outfl
close $outfl

#
# Run the new Tcl script:
#
# Open a pipe to the program (for both reading and writing:
r+)
#
set io [open "|[info nameofexecutable] $tempFileName"
r+]

#
# send a string to the new program
#    *MUST FLUSH*
puts $io "This will come back backwards."
flush $io

# Get the reply, and display it.
set len [gets $io line]
```

```
puts  "To reverse: 'This will come back backwards.'"
puts "Reversed is: $line"
puts "The line is $len characters long"

# Run the program with input defined in an exec call

set invert [exec [info nameofexecutable] $tempFileName
<< \
      "ABLE WAS I ERE I SAW ELBA"]

# display the results
puts "The inversion of 'ABLE WAS I ERE I SAW ELBA' is \n
$invert"

# Clean up
file delete $tempFileName
```

33 了解命令和变量的存在？-info

Tcl 提供很多命令来自我检测——描述你的程序正在发生什么，你的过程怎么实现的，哪些变量被设置了等等。

Info 命令允许 Tcl 程序获取 Tcl 解释器的信息。接下来的三节课会涉及 info 命令的多个方面。（其他允许自我检测的命令包括：traces，namespaces，通过 after 命令的调度而延迟执行的命令等等。）

这一课会涉及一些 info 子命令，它们会返回哪些过程，变量或命令存在于当前解释器实例中的信息。通过使用这些子命令，你可以确定一个变量或过程是否存在，然后你再试着去访问它。

下面的代码演示了如何使用 info exists 命令来做一个 incr 操作，而绝不会返回 no such variable 错误，因为它检查并确信变量存在，然后才增加它的值：

```
proc safeIncr {val {amount 1}} {
    upvar $val v
    if { [info exists v] } {
        incr v $amount
    } else {
        set v $amount
    }
}
```

返回现有的命令和变量的列表的那些 info 命令。

几乎所有的这些命令都接受一个模式参数，并遵守 string match 规则。如果 pattern 没有提供，全部项的列表被返回（犹如模式是“*”）。

info ***commands ?pattern?***

返回命令的列表，内部命令、过程，只要匹配 pattern。

info ***exists varName***

如果 ***varName*** 作为一个变量（或数组元素）存在与当前上下文中，返回 1，否则返回 0.

info ***functions*** *?**pattern**?*

返回匹配 ***pattern*** 的 expr 命令下可用的数学函数列表。

info ***globals ?pattern?***

返回匹配 ***pattern*** 的全局变量列表。

info ***locals*** *?**pattern**?*

返回匹配 ***pattern*** 的局部变量列表。

info ***procs ?pattern?***

返回匹配 ***pattern*** 的 Tcl 过程的列表。

info ***vars*** *?**pattern**?*

返回匹配 ***pattern*** 的局部和全局变量列表。

示例：

```
if {[info procs safeIncr] eq "safeIncr"} {
    safeIncr a
}

puts "After calling SafeIncr with a non existent variable:
$a"

set a 100
safeIncr a
puts "After calling SafeIncr with a variable with a value
of 100: $a"

safeIncr b -3
puts "After calling safeIncr with a non existent variable
by -3: $b"

set b 100
safeIncr b -3
puts "After calling safeIncr with a variable whose value
is 100 by -3: $b"

puts "\nThese variables have been defined: [lsort [info
vars]]"
puts "\nThese globals have been defined:  [lsort [info
globals]]"

set exist [info procs localproc]
if {$exist == ""} {
   puts "\nlocalproc does not exist at point 1"
}

proc localproc {} {
   global argv

   set loc1 1
   set loc2 2
   puts "\nLocal variables accessible in this proc are:
[lsort [info locals]]"
```

```
   puts "\nVariables accessible from this proc are:
[lsort [info vars]]"
   puts "\nGlobal variables visible from this proc are:
[lsort [info globals]]"
}

set exist [info procs localproc]
if {$exist != ""} {
   puts "localproc does exist at point 2"
}

localproc
```

34 解释器状态-info

有很多 info 命令的子命令可以提供解释器当前状态的子命令。这些命令提供访问信息，像当前版本、补丁等级，正在执行的是什么脚本，已经执行了多少命令，以及当前过程在调用树的深度。

Info tclversion 和 info patchlevel 可以用来查明执行你代码的解释器修订版本是否支持你正在使用的特征。如果你知道某些特征在某些版本的解释器中不可用，那你可以定义自己的过程来处理这个问题，或者简单地退出程序并返回错误消息。

Info cmdcount 和 info level 可以被使用，当优化 Tcl 脚本，查明需要多少等级和命令来完成一个功能。

注意，pid 命令并不属于 info 命令，其本身就是一个命令。

返回解释器当前状态信息的子命令（注意：有时其他几个子命令也是很有用的。）

info ***cmdcount***

返回解释器已经执行的命令的总数。

info ***level ?number?***

返回编译器当前对代码求值的栈等级。

如果 number 是一个正值，info level 返回那一级栈上这个过程的名称和参数。如果 info level 在正在被引用的过程里调用，number 值和 info level 的返回值一样。

如果 number 是一个负值，该命令引用当前 level 加上 number。因此，info level 返回那一级栈上这个过程的名称和参数。

info ***patchlevel***

返回全局变量 tcl_patchlevel 的值。这是一个三级版的数，标识 Tcl 版本，如：“8.4.6”。

info ***script***

如果有一个文件正在被求值的话，返回当前被求值的文件名称。如果没有，返回空串。

这可以被实例用来确定持有感兴趣的其他脚本或文件的路径（他们通常存在于相同或相关的路径里），而不必硬编码这些路径。

info ***tclversion***

返回全局变量 tcl_version 的值。它是解释器的修订号，如：“8.4”。

pid

返回当前进程号。

示例：

```
puts "This is how many commands have been executed: [info
cmdcount]"
```

```
puts "Now  *THIS* many commands have been executed: [info
cmdcount]"

puts "\nThis interpreter is revision level: [info
tclversion]"
puts "This interpreter is at patch level: [info patchlevel]"

puts "The process id for this program is [pid]"

proc factorial {val} {
    puts "Current level: [info level] - val: $val"
    set lvl [info level]
    if {$lvl == $val} {
        return $val
    }
    return [expr {($val-$lvl) * [factorial $val]}]
}

set count1 [info cmdcount]
set fact [factorial 3]
set count2 [info cmdcount]
puts "The factorial of 3 is $fact"
puts "Before calling the factorial proc, $count1 commands
had been executed"
puts "After calling the factorial proc, $count2 commands
had been executed"
puts "It took [expr $count2-$count1] commands to calculate
this factorial"

#
# Use [info script] to determine where the other files of
interest
# reside
#
set sysdir [file dirname [info script]]
source [file join $sysdir "utils.tcl"]
```

35 过程的相关信息-info

Info 命令包括一组提供一个过程的所有你想要的信息的子命令。这些子命令会返回一个过程的整体，参数及任何参数默认值。

这些子命令可以被用来：

- 访问调试器中的过程的内容。
- 从模板生成自定义过程。
- 提示输入（译者注：参数）时报告默认值。

返回过程信息的 Info 命令：

`info` ***args procname***

返回过程 ***procname*** 的参数名称的列表。

`info` ***body procname***

返回过程 ***procname*** 的主体。

`info` ***default procname arg varName***

如果过程 ***procname*** 中的参数 arg 有默认值，返回 1，并设置 ***varName*** 成默认值。否则，返回 0.

示例：

```
proc demo {argument1 {default "DefaultValue"} } {
    puts "This is a demo proc.  It is being called with
$argument1 and $default"
    #
    # We can use [info level] to find out if a value was given
for
    # the optional argument "default" ...
    #
    puts "Actual call: [info level [info level]]"
}

puts "The args for demo are: [info args demo]\n"
puts "The body for demo is:  [info body demo]\n"

set arglist [info args demo]
foreach arg $arglist {
    if {[info default demo $arg defaultval]} {
        puts "$arg has a default value of $defaultval"
    } else {
        puts "$arg has no default"
    }
}
```

36 模块化-source

Source 命令会加载一个文件并执行它。这允许一个程序被分解到多个文件，每一个文件定义功能的一个特定方面的过程和变量。例如，你可能有一个 database.tcl 的文件，它包含所有的处理数据库的过程；或者 gui.tcl 的文件，它处理用 Tk 创建图形用户界面的工作，那么主脚本可以使用 source 命令简单地引用每一个文件。程序模块化更强大的技术会在下一节的包部分讨论。

这个命令可以用来：

- 把一个程序分解到多个文件
- 制作包含一组特定功能的所有过程的库文件
- 配置程序
- 加载文件

source ***fileName***

读取脚本 ***fileName*** 文件，并执行它。如果脚本执行成功，source 返回脚本最后一个语句的值。

如果脚本中存在错误，source 返回那个错误。

如果有一个返回（不是 proc 里面的），那么 source 立即返回，不再执行剩余脚本。

如果 ***fileName*** 以波浪符号（~）开始，那么$env(HOME)会置换这个波浪符号，和 file 命令里面所做的一样。

示例：

sourcedata.tcl:

```
# Example data file to be sourced
set scr [info script]
proc testproc {} {
    global scr
    puts "testproc source file: $scr"
}
set abc 1
return
set aaaa 1
```

sourcemain.tcl:

```
set filename "sourcedata.tcl"
puts "Global variables visible before sourcing $filename:"
puts "[lsort [info globals]]\n"
```

```
if {[info procs testproc] eq ""} {
    puts "testproc does not exist.  sourcing $filename"
    source $filename
}

puts "\nNow executing testproc"
testproc

puts "Global variables visible after sourcing $filename:"
puts "[lsort [info globals]]\n"
```

37 创建可重用的库-packages 和 namespaces

上一课演示了 source 命令如何用来将一个程序分解为多个文件，每一个负责不同功能区块。这是一个实现模块化简单而有用的技术。然而，直接使用 source 命令也有很多弊端。Tcl 提供了一个更强大的机制来处理可重用代码单元，称作包（package）。一个包就是一组实现一个功能的一些文件的集合，以及标识这个包的名称和一个版本号，版本号用于同时使用同一个包得多个版本。一个包可以是一组 Tcl 脚本，二进制库和两者的联合。二进制库在这个教程里面不讨论。

使用包

package 命令提供使用包、比较包版本及在解释器上注册自己的包的能力。一个包通过 package require 命令并通过包的名称及可选的版本号来进行加载。脚本第一次请求一个包时，Tcl 建立一个可用的包及版本号的数据库。它通过搜索 tcl_pkgPath 和 auto_path 全局变量列出的所有目录及其任何子目录下的包索引文件来完成这个任务。每一个包提供一个叫做 pkgIndex.tcl 的文件来告诉 Tcl 这个目录下任何包的名称及版本，以及如何加载它们，在需要的时候。

好的风格应该是在你的每一个脚本文件的开始加上一组 package require 语句来加载任何需要的包。这服务于两个目的：1、确信任何任何丢失的需求被尽可能地标识出来；2、清晰地记录你的代码的全部依赖。Tcl 和 Tk 都作为包来发布，即使你已经加载了它们，最好也还是显式地引用它们。因为这使你的脚本更具有可移植性，并且它记录了你的脚本的版本需求。

创建包

创建包需要三步：

- 添加 package provide 语句到你的脚本里。
- 创建一个 pkgIndex.tcl 文件。
- 安装包到 Tcl 可以访问的地方。

第一步是添加 package provide 语句到你的脚本里。把这个语句放在你脚本的头部是一个好风格。package provide 命令告诉 Tcl 你的包的名称及提供的版本。

下一步是创建 pkgIndex.tcl 文件。这个文件告诉 Tcl 如何加载你的包。本质上，这个索引文件只是一个 Tcl 文件，它在 Tcl 搜索包得时候被加载进解释器。应该使用 package ifneeded 命令注册脚本，脚本会在需要的时候加载这个包。当 Tcl 搜索到任何包时，pkgIndex.tcl 文件在解释器里面全局性地被求值。由于这个原因，对于索引脚本文件而言，除了告诉 Tcl 如何加载这个包之外做别的任何事都是非常糟的风格。索引脚本里面不应该定义过程，引用包，或者执行任何其他的可能影响解释器状态的动作。

创建 pkgIndex.tcl 脚本最简单的方式是使用 pkg_mkIndex 命令。pkg_mkIndex 命令扫描 directory 目录下匹配给出的 pattern 的文件来寻找 package provide 命令。根据这个信息，它可以在该目录下生成适当的 pkgIndex.tcl 文件。

一旦一个包索引被创建，下一步就是把这个包移到一个 Tcl 可以找到的地方。Tcl_pkgPath 和 auto_path 全局变量包含 Tcl 搜索包的路径的列表。包索引和所有的包实现文件应该被安装到这些目录的子目录之一。或者，auto_path 变量可以在运行时被扩展来告诉 Tcl 这些新的地方来搜索包。

package require ?-exact? ***name ?version?***

加载 ***name*** 标识的包。如果-exact 开关连同 ***version*** 被给出，那么只有正确的包版本才会被接受。如果一个 ***version*** 被给出，而没有-exact ，那么任何等于或高于那个版本（但是要有相同的主版本号）的包都会被接受。如果没有指定版本号，那么任何版本都会被加载。如果一个匹配的包可以被找到，那么它被加载，命令返回实际的版本号；否则命令产生一个错误。

package provide ***name ?version?***

如果 ***version*** 被给出，这个命令告诉 Tcl ***name*** 指示的包的版本被加载了。如果同一个包的另一个版本已经被加载了，那么会产生错误。如果 ***version*** 参数被忽略，那么命令返回当前被加载的版本号，或者如果包还没有被加载的话返回空字符串。

pkg_mkIndex ?-direct? ?-lazy? ?-load ***pkgPat***? ?-verbose? ***dir ?pattern pattern ...?***

为一个或一组包创建 pkgIndex.tcl 文件。命令通过加载 ***dir*** 目录下与 ***pattern***S 匹配的文件并查看出现的新的包和命令实现。命令能够处理 Tcl 脚本文件和二进制文件（这里不讨论）。

命名空间

当使用包的时候，尤其当使用其他人写的代码时可能出现的一个问题是命名冲突。当两份代码试图定义以相同名称一个过程或变量时会发生。在 Tcl 里面，当这个问题发生时，老的过程或变量被简单地重写。这有时是很有用的特征，但更多情况下如果两个定义不兼容的话，却是一个产生众多 bug 的原因。为了解决这个问题，Tcl 提供了 namespace 命令来允许命令和变量被划分到独立的区块里面，叫做命名空间。每一个命名空间可以包含命令和变量，它们对于这个命名空间是局部的，而且不可以被其他命名空间里面的命令和变量重写。当一个命名空间里面的命令被调用时，它会看到该命名空间以及全局命名空间下所有其他的命令和变量。命名空间也可以包含其他命名空间。这允许一个层次化的命名空间被创建，方式类似于文静系统层次结构或者 Tk 部件层次结构。每一个命名空间自身有一个名称，它在其父命名空间可见。命名空间里面的项目可以通过创建一个到该项目的路径来访问。这通过将项目的名称和::连接来实现。举例来说，为了访问命名空间 foo 里面的变量 bar，你可以使用路径 foo::bar。这种路径被称作相对路径，因为 Tcl 会试图跟踪相对于当前命名空间的路径。如果失败，路径代表一个命令，那么 Tcl 也会查看相对于全局空间的路径。你可以通过描述它在全局空间（称作::）层次结构里面的准确位置来构造一个全修饰（fully-qualified）路径。例如，如果我们的 foo 空间是一个全局空间的子空间，那么 bar 的全限定名称是::foo::bar。当引用任何本空间之外的对象从而避免错误时，使用全修饰名称通常是一个好主意。

一个命名空间可以导出（export）其包含的一些或所有的命令名称。这些命

令然后可以被导入（import）到其他命名空间。这实际上在新的命名空间里面创建了一个局部的命令，当被调用时，会调用原始空间里面的原始命令。这是为位于外空间的频繁使用的命令创建快捷方式一项很有用的技术。一般地，一个命名空间应该小心导出命令时不要跟任何内置的 Tcl 命令或一个通用的名称重名。

当涉及命名空间时，一些用到的最重要的命令是：

`namespace eval` ***`path script`***

这个命令求命名空间下 ***`path`*** 指定的 ***`script`*** 的值。如果命名空间不存在那它会被创建。当脚本被执行时，这个命名空间变成当前的空间，而且任何非全修饰命令被相对于这个空间来解析。返回 ***`script`*** 最后命令的结果。

`namespace delete` ***`?namespace namespace ...?`***

删除每一个指定的命名空间，连同其包含的所有的变量，命令和子空间。

`namespace current`

返回当前空间的全修饰路径。

`namespace export ?-clear?` ***`?pattern pattern ...?`***

添加所有与 ***`pattern`***S 匹配的命令到当前空间导出的命令列表。如果-clear 开关被给出，那么导出列表在添加任何新命令之前被清空。如果没有参数给出，返回当前导出的命令名称。每一个模式都是一个 glob 风格的模式，像*,[a-z]*,或者*foo*等。

`namespace import ?-force?` ***`?pattern pattern ...?`***

导入与任意 ***`pattern`***S 匹配的所有命令到当前空间。每一个模式都是 glob 风格的模式，像 foo::*,或者 foo::bar 等。

使用包的命名空间

William Duquette 有一个很好的使用命名空间和包的指导，在 http://www.wjduquette.com/tcl/namespaces.html 。一般地，一个包应该提供一个命名空间最为全局空间的子空间，并把它自己所有的命令和变量放到该空间里面。一个包不应该按照默认方式把命令和变量放到全局空间里面。给你的包及其提供的空间相同的名称来避免冲突也是很好的风格。

示例：

这个示例创建了一个包，它提供了一个栈数据结构。

```
# Register the package
package provide tutstack 1.0
package require Tcl     8.5

# Create the namespace
namespace eval ::tutstack {
    # Export commands
    namespace export create destroy push pop peek empty

    # Set up state
```

```
    variable stack
    variable id 0
}

# Create a new stack
proc ::tutstack::create {} {
    variable stack
    variable id

    set token "stack[incr id]"
    set stack($token) [list]
    return $token
}

# Destroy a stack
proc ::tutstack::destroy {token} {
    variable stack

    unset stack($token)
}

# Push an element onto a stack
proc ::tutstack::push {token elem} {
    variable stack

    lappend stack($token) $elem
}

# Check if stack is empty
proc ::tutstack::empty {token} {
    variable stack

    set num [llength $stack($token)]
    return [expr {$num == 0}]
}

# See what is on top of the stack without removing it
proc ::tutstack::peek {token} {
    variable stack

    if {[empty $token]} {
      error "stack empty"
    }
```

```
    return [lindex $stack($token) end]
}

# Remove an element from the top of the stack
proc ::tutstack::pop {token} {
    variable stack

    set ret [peek $token]
    set stack($token) [lrange $stack($token) 0 end-1]
    return $ret
}
```

群组（ensemble）

结构化相关命令通用做法是将他们分组，每一组包括一个单独的命令及其子命令。这种命令被称作群组命令。Tcl 标准库里面有很多示例。例如，string 命令是一个群，其子命令是 length，index，match 等。Tcl8.5 引入了一个将一个命名空间转换成一个群组的方便的方法，通过 namespace ensemble 命令。这个命令是非常灵活，有很多选项可以准确指定子命令如何被映射到该空间下的命令。然而，最基本的用法是非常简单的，只是创建一个和命名空间名称相同的群组命令，所有的导出过程被注册为子命令。为了阐明这一点，我们把我们的栈数据结构转化为一个群组：

```
package require tutstack 1.0
package require Tcl     8.5

namespace eval ::tutstack {
    # Create the ensemble command
    namespace ensemble create
}

# Now we can use our stack through the ensemble command
set stack [tutstack create]
foreach num {1 2 3 4 5} { tutstack push $stack $num }

while { ![tutstack empty $stack] } {
    puts "[tutstack pop $stack]"
}

tutstack destroy $stack
```

除了在命名空间下提供一个更好的访问功能的语法，群组命令还帮助清晰地将一个包的公共接口及私有实现细节区分开来，因为只有导出命令被注册为子命令，群组会实施这个区分。熟悉面向对象程序设计的读者会意识到命名空间和群组机制提供了很多相同的封装优势。确实，Tcl 很多面向对象的扩展建立在强大

的命名空间机制之上。

38 创建命令-eval

Tcl 和其他大部分编译器的一个不同是 Tcl 允许正在执行的程序在运行时创建新的命令，并执行他们。

一条 Tcl 命令被定义成一组字符串列表，其首个字符串是一个命令或 proc。任何符合条件的字符串或列表都可以被求值和执行。

Eval 命令会对字符串列表求值，就好像它们是%后面输入的或来自文件的命令一样。Eval 命令正常情况下返回命令求得的最终结果。如果被求值的命令抛出一个错误（例如，如果其中一个字符串中存在语法错误），那么 eval 会抛出一个错误。

注意，不管 concat 还是 list 都可以用来创建命令字符串，但是这两个命令会创建稍微不同的命令字符串。

eval *arg1 ??arg2?? … ??argn??*

对 arg1-argn 求值，当做一个或多个 Tcl 命令那样。Args 参数拼接成一个字符串，并被传递给 tcl_Eval 来求值和执行。

Eval 返回所求的结果（或错误代码）。

示例：

```
set cmd {puts "Evaluating a puts"}
puts "CMD IS: $cmd"
eval $cmd   // Evaluating a puts

// 这里其实就是动态构造一个命令（过程），以字符串的形式。
if {[string match [info procs newProcA] ""] } {
    puts "\nDefining newProcA for this invocation"
    set num 0;
    set cmd "proc newProcA "
    set cmd [concat $cmd "{} {\n"]
    set cmd [concat $cmd "global num;\n"]
    set cmd [concat $cmd "incr num;\n"]
    set cmd [concat $cmd " return
\"/tmp/TMP.[pid].\$num\";\n"]
    set cmd [concat $cmd "}"]
    eval  $cmd
}

puts "\nThe body of newProcA is: \n[info body newProcA]\n"

puts "newProcA returns: [newProcA]"
puts "newProcA returns: [newProcA]"

#
# Define a proc using lists
```

```
#
// 可以发现，用 list 操作更清晰，简单易懂。不过总的来说，目的和效果都是在代码执行过程中定义新的命令（过程），并执行之。
if {[string match [info procs newProcB] ""] } {
    puts "\nDefining newProcB for this invocation"
    set cmd "proc newProcB "
    lappend cmd {}
    lappend cmd {global num; incr num; return $num;}

    eval  $cmd
}

puts "\nThe body of newProcB is: \n[info body newProcB]\n"
puts "newProcB returns: [newProcB]"
```

39 更多命令构造-format list

当你试着为 **eval** 构造命令字符串的时候，可能会有一些意外的结果。

例如，

```
eval puts OK
```

会打印字符串 OK。然而，

```
eval puts Not OK
```

会产生一个错误。

第二个命令产生错误的原因是 **eval** 使用 **concat** 来合并它的参数到一个命令字符串里面。这导致两个单词 **Not OK** 被当做两个参数传给 **puts**。如果 **puts** 的参数多于 1 个，第一个参数必须是一个文件指针。

第二个命令的正确的书写方式有：

```
eval [list puts {Not OK}]
eval [list puts "Not OK"]
set cmd "puts" ; lappend cmd {Not OK}; eval $cmd
```

只要你掌握传给 **eval** 的参数是如何分组的，你就可以用很多方法为 **eval** 创建字符串，包括 **string** 命令和 **format**。

推荐的为 **eval** 构造命令的方式是使用 **list** 和 **lappend** 命令。然而，如果你需要在命令里面加入括号，就像先前一课那样，这些命令会变得很难使用。

前一课的例子这里被 lappend 在示例代码里面重新实现。

命令的完成可以用 **info complete** 来检查。**info complete** 也可以用在交互式程序里面来确定正在输入的行是否是一个 complete 命令，或者用户只需要输入回车来更好地格式化命令。

info complete *string*

如果 ***string*** 没有不成对的方括号、花括号和圆括号，那值 1 被返回，否则返回 0.

示例：

```
set cmd "OK"
eval puts $cmd

set cmd "puts" ; lappend cmd {Also OK}; eval $cmd

set cmd "NOT OK"
eval puts $cmd

eval [format {%s "%s"} puts "Even This Works"]

set cmd "And even this can be made to work"

eval [format {%s "%s"} puts $cmd ]
```

```
set tmpFileNum 0;

set cmd {proc tempFileName }
lappend cmd ""
lappend cmd "global num; incr num; return
\"/tmp/TMP.[pid].\$num\""
eval  $cmd

puts "\nThis is the body of the proc definition:"
puts "[info body tempFileName]\n"

set cmd {puts "This is Cool!}

if {[info complete $cmd]} {
   eval $cmd
} else {
   puts "INCOMPLETE COMMAND: $cmd"
}
```

40 不经过求值的置换-format,subst

Tcl 解释器在命令置换阶段只做一次置换过程。在一些情况下，像把一个变量名放到一个变量里，就需要经过两次置换阶段。这样，subst 命令就用得上了。Subst 执行一次置换过程，而不需要做任何命令的执行，除了那些要求置换起作用的命令。举例来说：[]内的命令会被执行，结果会放在返回字符串里。

在示例代码：

```
puts "[subst $$c]\n"
```

里，它演示了一个例子，即把一个变量名放在变量了，并间接求值。

Format 命令也可以用于强制某些等级的置换发生。

subst ?-nobackslashes? ?-nocommands? ?-novariables? *string*

传递 ***string***，并返回反斜杠序列、命令和变量被替换成它们的等价体之后的原始字符串。

如果任何**-no…**参数出现，那么那组置换不会执行。

注意：**subst** 不接受方括号和引号。

示例：

```
set a "alpha"
set b a

puts {a and b with no substitution: $a $$b}
puts "a and b with one pass of substitution: $a $$b"
puts "a and b with subst in braces: [subst {$a $$b}]"
puts "a and b with subst in quotes: [subst "$a $$b"]\n"

puts "format with no subst [format {$%s} $b]"
puts "format with subst: [subst [format {$%s} $b]]"
eval "puts \"eval after format: [format {$%s} $b]\""

set num 0;
set cmd "proc tempFileName {} "
set cmd [format "%s {global num; incr num;" $cmd]
set cmd [format {%s return "/tmp/TMP.%s.$num"} $cmd [pid] ]
set cmd [format "%s }" $cmd ]
eval $cmd

puts "[info body tempFileName]"

set a arrayname
set b index
set c newvalue
eval [format "set %s(%s) %s" $a $b $c]
```

```
puts "Index: $b of $a was set to: $arrayname(index)"
```

41 改变工作目录-cd,pwd

Tcl 也支持改变和显示当前工作路径的命令。

它们是：

cd *?dirName?*

改变当前路径到 *dirName*（如果 *dirName* 被给出），如果 *dirName* 未被给出则转到**$HOME** 目录。如果 *dirName* 是波浪字符（~），cd 将当前工作目录转到用户 home 家目录。如果 *dirName* 以波浪字符（~）开始，那剩余的字符被当做一个登录 id，cd 将当前工作路径转到用户$HOME 目录。

Pwd

返回当前路径。

示例：

```
set dirs [list TEMPDIR]

puts "[format "%-15s  %-20s " "FILE" "DIRECTORY"]"

foreach dir $dirs {
    catch {cd $dir}
    set c_files [glob -nocomplain c*]

    foreach name $c_files {
        puts "[format "%-15s  %-20s " $name [pwd]]"
    }
}
```

42 调试和错误-errorInfo errorCode catch error return

在上一节课中，我们讨论了 return 命令如何用来从 proc 返回一个值。在 Tcl 中，一个 proc 可以返回一个值，但它总是返回一个状态。

当一个 Tcl 命令或过程在执行过程中遇到一个错误，全局变量 errorInfo 被设置，一个错误条件被生成。如果你有一个过程 a 调用过程 b，而 b 又调用过程 c，c 调用 d，如果 d 产生一个错误，那“调用堆栈”会展开。由于 d 产生了一个错误，c 不会彻底完成执行，它必须把错误向上传给 b，一次传给 a，每一个过程会添加一些关于这个问题的需要报告的信息。例如：

```
proc a {} {
    b
}
proc b {} {
    c
}
proc c {} {
    d
}
proc d {} {
    some_command
}

a
```

会产生如下输出：

```
invalid command name "some_command"
    while executing
"some_command"
    (procedure "d" line 2)
    invoked from within
"d"
    (procedure "c" line 2)
    invoked from within
"c"
    (procedure "b" line 2)
    invoked from within
"b"
    (procedure "a" line 2)
    invoked from within
"a"
    (file "errors.tcl" line 16)
```

实际上，当任何异常条件产生，包括 break 和 continue，上述情况都会发生。

Break 和 continue 命令通常出现在某些循环里面，循环命令会捕获异常并适当处理它，意味着它要么停止执行循环，要么继续下一次循环过程而不再执行循环体剩余部分。

用 catch 命令捕获错误和异常也是可以的。Catch 命令执行一些代码，捕获代码偶然产生的任何错误。程序员然后可以决定对这些错误做什么和如何相应地采取措施，而不需要让整个应用停止。

举例来说，如果 open 返回一个错误，用户可以被提示来提供另一个文件名。

Tcl 过程也可以产生一个错误状态条件。这可以通过为 return 命令指定错误返回选项实现，或者使用 error 命令。不管哪一种情况，一个消息都会被放到 errorInfo 里面，而且过程会产生一个错误。

error ***message ?info? ?code?***

产生一个错误条件，强迫 Tcl 调用堆栈展开，其中在每一步，错误信息都会被添加。

如果 info 或 code 被提供，errorInfo 和 errorCode 变量会被初始化成这些值。

catch ***script ?varName?***

求 ***script*** 的值，并执行之。Catch 的返回值是一个 Tcl 解释器执行完 ***script*** 后的状态返回值。如果 ***script*** 没有错误，值是 0，否则是 1。

如果 ***varName*** 被提供了，并且如果 ***script*** 成功执行，那 ***script*** 的返回值被放在 ***varName*** 里面。如果没有成功执行，***varName*** 存放的就是错误。

return ***?-code code? ?-errorinfo info? ?-errorcode errorcode? ?value?***

产生一个异常条件返回。可能的参数是：

-code ***code***

后面的值指定返回的状态。***Code*** 必须是：

- ok – 正常的状态返回
- error – 过程返回错误状态
- return – 正常 return
- break – 过程返回 break 状态
- continue – 过程返回 continue 状态

这些可以让你写出行为和内置 break，error 和 continue 命令一样的过程。

-errorinfo ***info***

info 会是 errorInfo 变量的第一个字符串。

-errorcode ***errorcode***

过程设置 errorCode 为 ***errorcode***。

value

value 字符串会被设置成过程的返回值。

errorInfo

errorInfo 是一个全局变量，它包含运行出错的命令的错误信息。

errorCode

errorCode 是一个全局变量，它包含运行出错的命令的错误代码。这是因为这种格式易于用脚本解析，这样 Tcl 脚本可以检查变量的内容，然后决定相应地做什么。

示例：

```
proc errorproc {x} {
    if {$x > 0} {
      error "Error generated by error" "Info String for
error" $x
    }
}

catch errorproc
puts "after bad proc call: ErrorCode: $errorCode"
puts "ERRORINFO:\n$errorInfo\n"

set errorInfo "";
catch {errorproc 0}
puts "after proc call with no error: ErrorCode: $errorCode"
puts "ERRORINFO:\n$errorInfo\n"

catch {errorproc 2}
puts "after error generated in proc: ErrorCode: $errorCode"
puts "ERRORINFO:\n$errorInfo\n"


proc returnErr { x } {
    return -code error -errorinfo "Return Generates This"
-errorcode "-999"
}

catch {returnErr 2}
puts "after proc that uses return to generate an error:
ErrorCode: $errorCode"
puts "ERRORINFO:\n$errorInfo\n"

proc withError {x} {
    set x $a
}
```

```
catch {withError 2}
puts "after proc with an error: ErrorCode: $errorCode"
puts "ERRORINFO:\n$errorInfo\n"

catch {open [file join no_such_directory no_such_file] r}
puts "after an error call to a nonexistent file:"
puts "ErrorCode: $errorCode"
puts "ERRORINFO:\n$errorInfo\n"
```

43 更多调试-trace

当你正在调试 Tcl 代码时，有时能够跟踪代码的执行或者当多种事情发生在一个变量上时简单地检查它的状态会是非常有用的。Trace 命令提供了这些功能。它是一个非常强大的命令，可以以很多有意思的方式使用。同时也要小心不要滥用，如果使用不当（例如，变量似乎正在神奇地改变），可能导致代码难以理解。所以你要小心使用它。

有三条基本操作可以用于 trace 命令的执行：

- add，一般形式：`trace add *type ops ?args?*`
- info，一般形式：`trace info *type name*`
- `remove`,一般形式：`trace remove *type name opList command*`

它们分别添加 trace，检索 traces 信息，移除 trace。Traces 可以被添加到三种“事物”：

- variable –当一些事件发生在这些变量上，像正在读写时，被添加到变量的 traces 被调用。
- command – 每当当命名的命令被重命名或删除时，添加到这些命令的 traces 就会被调用。
- execution – 每当命名的命令被执行时，execution 上的 traces 就会被调用。

变量上的 traces 在四种条件下会被调用—当变量由于 array 命令被访问或修改；当变量被读、写和重置。例如，为了在一个变量上设置 trace，这样当它被写入时，而值并不改变，你可以这样做：

```
proc vartrace {oldval varname element op} {
    upvar $varname localvar
    set localvar $oldval
}

set tracedvar 1
trace add variable tracedvar write [list vartrace
$tracedvar]

set tracedvar 2
puts "tracedvar is $tracedvar"
```

在上面的例子里面，我们创建了一个四个参数的过程。我们提供第一个参数，变量的原始值，因为变量的值已经改变之后 trace 被触发，所以我们需要自己保存原始值。另外三个参数是变量的名称，元素名称（如果变量是一个数组，例子中不是）及 trace 的操作（这里是 write）。当 trace 被调用的时候，我只是把变量的值设置为原始值。我可以做一些像生成一个错误，这样警告人们变量不应该被写入。实际上，这或许会更好。如果其他人正尝试理解你的程序，当他们发

现简单的 set 命令不再起作用时，他们可能会非常迷惑。

command 和 execution traces 是为了专业用户—可能那些用 Tcl 为 Tcl 写调试器的人，因此并没有包含在本教程里面。更多信息请查看 trace 手册页面。

示例：

```
proc traceproc {variableName arrayElement operation} {
    set op(write) Write
    set op(unset) Unset
    set op(read) Read

    set level [info level]
    incr level -1
    if {$level > 0} {
      set procid [info level $level]
    } else {
      set procid "main"
    }

    if {![string match $arrayElement ""]} {
      puts "TRACE: $op($operation)
$variableName($arrayElement) in $procid"
    } else {
      puts "TRACE: $op($operation) $variableName in
$procid"
    }
}

proc testProc {input1 input2} {
    upvar $input1 i
    upvar $input2 j

    set i 2
    set k $j
}

trace add variable i1 write traceproc
trace add variable i2 read traceproc
trace add variable i2 write traceproc

set i2 "testvalue"

puts "\ncall testProc"
testProc i1 i2

puts "\nTraces on i1: [trace info variable i1]"
```

```
puts "Traces on i2: [trace info variable i2]\n"

trace remove variable i2 read traceproc
puts "Traces on i2 after vdelete: [trace info variable i2]"

puts "\ncall testProc again"
testProc i1 i2
```

44 命令行参数和环境变量字符串

如果脚本被命令行调用时有不同的值，那么它会有用的多得多。

例如，可以写一个从一个文件提取一个特定值的脚本，这样它提示输入文件名称，读取文件名称，然后提取数据。或者，它也可以写成循环遍历命令行输入的文件名称，并从每一个文件提取数据，然后输出文件名称和数据。

第二个写程序的方法可以容易地被用于其他脚本。这使其更有用。

Tcl 脚本命令行参数的数量被传给全局变量 argc。Tcl 脚本的名称传给全局变量 argv0，剩余的命令行参数被以列表的形式传给 argv。运行脚本的可执行体名称，像 tclsh 通过命令 info nameofexecutable 给出。

另一个给脚本传递参数的方法是使用 **environment variables**。举例来说，假设你正在写一个程序，即用户提供一些评论才能进入一条记录。我们认为，允许用户用他们自己喜欢的编辑器编辑他们的评论会是很友好的。如果用户已经定义了 EDITOR 环境变量，那你可以调用那个编辑器让他们使用。

环境变量在 Tcl 脚本里面也是可用的，即全局关联数组 env。Env 的下标是环境变量的名称。命令 puts ″$env(PATH)″ 会输出 PATH 环境变量的内容。

示例：

```
puts "There are $argc arguments to this script"
puts "The name of this script is $argv0"
if {$argc > 0} {puts "The other arguments are: $argv" }

puts "You have these environment variables set:"
foreach index [array names env] {
    puts "$index: $env($index)"
}
```

45 计时脚本

让脚本允许更快的最简单的方法就是购买一个更快的处理器。不幸的是，这通常并不总是一个可选项。你可能需要优化你的脚本来使它运行更快。如果你不能测量执行部分脚本的时间，而这部分正是你试着优化的，那么这会很困难。

Time 命令就是这个问题的解决方案。Time 会测量执行脚本的时间长度。你可以修改脚本，重新执行 time，查看你提高了多少。

你执行示例之后，摆弄一下循环次数的大小，即 timetst1 和 timetst2。如果你把内层循环设成 5 或更少，那么执行 timetst2 会花费比 timetst1 更多的时间。这是因为需要时间对变量 i 进行计算和赋值。如果内层循环太小了，那么因为内层循环加法减少所节省的时间又会被消耗在外层循环的计算上。

time ***script ?count?***

返回执行 ***script*** 消耗的毫秒数。如果 ***count*** 被指定，那就会执行 ***count*** 次，并求出平均结果。这里的时间是流逝的时间，而不是 CPU 时间。

示例：

```
proc timetst1 {lst} {
    set x [lsearch $lst "5000"]
    return $x
}

proc timetst2 {array} {
    upvar $array a
    return $a(5000);
}

# Make a long list and a large array.
for {set i 0} {$i < 5001} {incr i} {
    set array($i) $i
    lappend list $i
}

puts "Time for list search: [ time {timetst1 $list} 10]"
puts "Time for array index: [ time {timetst2 array} 10]"
```

46 通道 I/O:socket,fileevent,vwait

Tcl I/O 基于通道的概念。一个通道概念上合 C 语言里面的 FILE*，或者 shell 编程里面的流相似。不同的是，一个通道可以要么是一个类似文件的流设备，也可以是类似 socket 的面向连接的构造。

一个基于通道的流用 open 命令创建，这在第 26 课讨论过。基于通道的 socket 用 socket 命令创建。Socket 可以被以客户端或服务器端模式打开。

如果一个 socket 通道被以服务器端打开，那么 tcl 程序会在这个通道上监听另一个尝试和它连接的任务。当上述情况发生时，一个新的通道被创建用于这个连接（server->new client），并且 tcl 程序继续在原端口号上监听连接。这样，一个单独的 Tcl 服务端可以同时地和多个客户端通话。

当一个通道存在时，并且可用于读写时，可以定义一个 handler（处理器）并调用它。Handler 用 fileevent 命令定义。当一个 tcl 过程对一个阻塞设备进行 gets 或 puts 操作，但设备没有准备好 I/O 时，程序会阻塞直到设备准备就绪。如果 I/O 通道的另一端已经断线时，这可能会很长时间。使用 fileevent 命令时，只有通道准备就绪传输数据时，程序才访问它。

最终，出现了一个命令来等待直到一个事件发生。Vwait 命令会等待直到一个变量被设置。这可以用于为客户端和服务端的交互创建一个信号量风格的功能，并让一个控制过程知道一个事件已经发生了。

看一下示例，你会看到 socket 命令正在被客户端和服务端同时使用，而 fileevent 和 vwait 命令正在被用于控制客户端和服务端的 I/O。

特别注意，flush 命令也正在使用。当一个通道作为一个命令的管道打开时，它并不发送数据，直到要么 flush 被调用，要么缓存满了。基于通道的 socket 不会自动发送数据。

`socket -server` ***command ?options? port***

带`-server` 标记的 socket 命令启动一个服务器端 socket，监听端口 ***port***。

当一个连接发生在 ***port*** 时，过程 ***command*** 被调用，参数是：

- channel - 新客户端的通道
- address - 客户端的 IP 地址
- port - 分配给客户端的端口

`socket` ***?options? host port***

不带`-server` 选项的 socket 命令打开一个到 IP 地址是 ***host***，端口地址为 ***port*** 的系统的客户端连接。IP 地址可以是数字字符串或完整格式的域地址。要连接到本地，使用地址 127.0.0.1（环回地址）。

`fileevent` ***channelID*** `readable` ***?script?***
`fileevent` ***channelID*** `writeable` ***?script?***

fileevent 命令定义一个 handler（处理器）用于条件发生时调用。条件

是 readable，当 ***channelID*** 上的数据准备好读取的时候调用脚本 ***script***；writeable，当 ***channelID*** 准备好接收数据时。注意， ***script*** 必须检查文件结束（EOF）。

vwait ***varName***

vwait 命令暂停脚本的执行，直到一些背景动作设置 ***varName*** 的值。背景动作可以是一个 fileevent 事件触发的过程调用，或者 socket 连接，或者 tk 部件引起的事件。

示例：

```
proc serverOpen {channel addr port} {
    global connected
    set connected 1
    fileevent $channel readable "readLine Server $channel"
    puts "OPENED"
}

proc readLine {who channel} {
    global didRead
    if { [gets $channel line] < 0} {
      fileevent $channel readable {}
      after idle "close $channel;set out 1"
    } else {
      puts "READ LINE: $line"
      puts $channel "This is a return"
      flush $channel;
      set didRead 1
    }
}

set connected 0
# catch {socket -server serverOpen 33000} server
set server [socket -server serverOpen 33000]

after 100 update

set sock [socket -async 127.0.0.1 33000]
vwait connected

puts $sock "A Test Line"
```

```
flush $sock
vwait didRead
set len [gets $sock line]
puts "Return line: $len -- $line"

catch {close $sock}
vwait out
close $server
```

47 Time and Date-clock

Clock 命令提供 Tcl 里面的时间和日期的访问。根据调用的子命令，它可以取得当前的时间，或者在 time 和 date 的不同表示之间进行转换。

Clock 命令是一个平台相关的方法，用于获取 unix **date** 命令的显示功能，并提供对 unix **gettime()**调用的返回值的访问。

clock seconds

clock seconds 命令以秒为单位返回自纪元以来的时间。纪元以来的日期随操作系统不同而变化，因此这个值对于比较目的或用于 **clock format** 命令的输入时会很有用。

clock format *clockValue* ?-gmt *boolean*? ?-format *string*?

format 子命令格式化一个 ***clockValue***（**clock clicks** 返回的那样）为人可读的字符串。

-gmt 开关接受一个布尔值作为第二个参数。如果布尔值是 **1** 或 **true**，那时间会被格式化为格林威治标准时间，否则它被格式化为本地时间。

-format 选项控制返回什么样的格式。要格式化的 ***String*** 参数的内容和第 19、33、34 节讨论的格式语句相似。另外，有几个**%***描述符可以用来描述输出。

他们包括：

- **%a** . . . . 简写的周日名称（Mon,Tue,等）
- **%A** . . . . 全称周日名称（Monday,Tuesday,等）
- **%b** . . . . 简写月名称（Jan,Feb,等）
- **%B** . . . . 全称月名称（January,February,等）
- **%d** . . . . 每月几号
- **%j** . . . . 儒历日
- **%m** . . . . 月号（01-12）
- **%y** . . . . 每世纪的年号
- **%Y** . . . . 4 位数年号

- **%H** . . . . 小时（00-23）
- **%I** . . . . 小时（00-12）
- **%M** . . . . 分钟（00-59）
- **%S** . . . . 秒（00-59）
- **%P** . . . . PM 或 AM（下午或上午）

- **%D** . . . . 日期 格式：%m/%d/%y

- **%r** . . . . 时间 格式：%I：%M：%S %p
- **%R**. . . . 时间 格式：%I：%M
- **%T**. . . . 时间 格式：%I：%M：%S
- **%Z**. . . . 时区名称

clock scan *dateString*

scan 子命令转换一个人可读的字符串为系统时钟值，和 **clock seconds** 返回一致。

dateString 参数的这样格式的字符串：

time

每天的时间，以如下所示的任一种格式。正午可以是 AM，也可以是 PM 或大写的变体。如果没有指定，那么小时被解释为 24 小时制。时区可以是三个字母描述，EST，PDT 等。

- hh:mm:ss ?meridian? ?zone?
- hhmm ?meridian? ?zone?

date

如下所示格式的日期：

- mm/dd/yy
- mm/dd
- monthname dd, yy
- monthname dd
- dd monthname yy
- dd monthname
- day, dd monthname yy

示例：

```
set systemTime [clock seconds]

puts "The time is: [clock format $systemTime
-format %H:%M:%S]"
puts "The date is: [clock format $systemTime -format %D]"
puts [clock format $systemTime -format {Today is: %A, the %d
of %B, %Y}]
puts "\n the default format for the time is: [clock format
$systemTime]\n"

set halBirthBook "Jan 12, 1997"
set halBirthMovie "Jan 12, 1992"
set bookSeconds [clock scan $halBirthBook]
set movieSeconds [clock scan $halBirthMovie]
```

```
puts "The book and movie versions of '2001, A Space Oddysey'
had a"
puts "discrepancy of [expr {$bookSeconds - $movieSeconds}]
seconds in how"
puts "soon we would have sentient computers like the HAL
9000"
```

48 更多通道 I/O-fblocked&fconfigure

上一节课演示了如何使用文件和阻塞 socket 通道。Tcl 也提供非阻塞读写，并且允许你配置 I/O 缓存的大小以及行如何结束。

非阻塞读写意味着，并不是调用 **gets** 并一直等待到数据可用，非阻塞会立即返回。如果有数据可用，它就会被读取，而如果没有可用数据，**gets** 会返回长度 0.

如果你有几个通道必须检查输入，那你可以用 **fileevent** 命令来触发这些通道的读操作，然后使用 **fblocked** 命令确定何时全部数据读取完成。

Fblocked 和 **fconfigure** 命令提供更多的对通道的行为的控制。

Fblocked 命令检查一个通道是否已经返回了所有的可用的输入。当你正在处理设置为非阻塞模式的通道的时候，它是很有用的。你需要确定是否需要有可用的数据，或者通道是否已经被另一端关闭了。

Fconfigure 命令有很多选项，它们允许你查询或很好地调整通道的行为，包括其是否处于阻塞或非阻塞状态，缓存的大小，行结束字符，等等。

fconfigure *channel ?param1? ?value1? ?param2? ?value2?*

配置通道的行为。如果没有参数 param 值提供，那么一组有效的配置参数及其值会被返回。

如果一或多个 param/value 对被给出，那么那些参数被设置为请求值。

可以被设置的参数包括：

- **-blocking** . . .确定当数据不能在通道上被传输时，任务是否会阻塞。（例如，如果没有可读的数据，或者写入端的缓存满了。）
- **-buffersize** . . .数据被发送前被缓存或者数据接收时能够缓存用于读取的字节数。值必须是一个 10 到 1000000 之间的整数。
- **-translation** . . .设置 Tcl 如何结束输出的一行。默认情况下，行被换行符、回车和回车/换行终止，而它们需要适合解释器所运行在的系统。

  这可以被配置为：

  - **auto . . .** 翻译换行（newlines），回车，或换行/回车为行的结束符。输出当前平台正确的行结束。
  - **binary . . .** 把换行（newlines）当做行的结束符。输出不添加任何的行结束。
  - **cr . . .** 把回车当做行结束符（并在内部把它们翻译成换行 newline）。输出行以回车结束。这是 Macintosh 标准。
  - **crlf . . .** 把 cr/lf 对当做行结束符，以回车/换行（linefeed）组合结束输出行。这是 Windows 标准，并且也只能用于所有的面向行的网络协议。
  - **lf . . .** 把换行（linefeed）当做行结束符，并以换行（linefeed）结束输出行。这是 Unix 标准。

这个例子和第 40 课的例子很相似，都是同样地脚本里面的一个 client 和 server socket。它演示了一个 server 通道，其正在被配置为非阻塞，并使用默认缓存形式——数据直到换行（newline）出现或者缓存已满才对脚本变得可用。

当首先写：

```
puts -nonewline $sock "A Test Line"
```

被完成，**fileevent** 触发读操作，但是 **gets** 并不能读取字符，因为没有出现换行(newline)。Gets 返回-1，fblocked 返回 1.当只有换行(newline)被发送，输入缓存中的数据会变得可用，gets 返回 18，fblocked 返回 0.

示例：

```
proc serverOpen {channel addr port} {
    puts "channel: $channel - from Address: $addr  Port:
$port"
    puts "The default state for blocking is: [fconfigure
$channel -blocking]"
    puts "The default buffer size is: [fconfigure $channel
-buffersize ]"

    # Set this channel to be non-blocking.
    fconfigure $channel -blocking 0
    set bl [fconfigure $channel -blocking]
    puts "After fconfigure the state for blocking is: $bl"

    # Change the buffer size to be smaller
    fconfigure $channel -buffersize 12
    puts "After Fconfigure buffer size is: [fconfigure
$channel -buffersize ]\n"

    # When input is available, read it.
    fileevent $channel readable "readLine Server $channel"
}

proc readLine {who channel} {
    global didRead
    global blocked

    puts "There is input for $who on $channel"

    set len [gets $channel line]
    set blocked [fblocked $channel]
    puts "Characters Read: $len  Fblocked: $blocked"

    if {$len < 0} {
        if {$blocked} {
            puts "Input is blocked"
        } else {
            puts "The socket was closed - closing my end"
```

```
            close $channel;
        }
    } else {
        puts "Read $len characters:  $line"
        puts $channel "This is a return"
        flush $channel;
    }
    incr didRead;
}

set server [socket -server serverOpen 33000]

after 120 update;  # This kicks MS-Windows machines for this
application

set sock [socket 127.0.0.1 33000]

set bl [fconfigure $sock -blocking]
set bu [fconfigure $sock -buffersize]
puts "Original setting for sock: Sock blocking: $bl
buffersize: $bu"

fconfigure $sock -blocking No
fconfigure $sock -buffersize 8;

set bl [fconfigure $sock -blocking]
set bu [fconfigure $sock -buffersize]
puts "Modified setting for sock: Sock blocking: $bl
buffersize: $bu\n"

# Send a line to the server -- NOTE flush

set didRead 0
puts -nonewline $sock "A Test Line"
flush $sock;

# Loop until two reads have been done.

while {$didRead < 2} {
    # Wait for didRead to be set
    vwait didRead
    if {$blocked} {
        puts $sock "Newline"
        flush $sock
```

```
        puts "SEND NEWLINE"
    }
}

set len [gets $sock line]
puts "Return line: $len -- $line"
close $sock
vwait didRead
catch {close $server}
```

49 子解释器

对于大部分应用来说，一个单独的解释器和子程序已经分成足够了。然而，如果你正在创建一个客户端——服务端系统（比如说），你可能需要有几个解释器与不同的客户端通话，维持它们的状态。你可以使用状态变量、命名规范和与硬盘交换状态来实现，但这样变得乱七八糟。

Interp 命令在已存在的解释器里创建新的子解释器。子解释器可用拥有自己的变量、命令和打开文件的集合，或者它们可以被允许访问父解释器的对象。

如果子解释器创建时带有-safe 选项，那它不能访问文件系统，否则会破坏你的系统。这个特征允许脚本对未知（和不信任）的地方的代码进行求值。

子解释器的名字是一个层次列表。如果解释器 foo 是 bar 解释器的子解释器，那么它可以被顶层的解释器访问，{bar foo}。

初始解释器（当你输入 tclsh 得到的）是空的列表{}。

Interp 命令有几个子命令和选项。关键的子集是：

interp create ***-safe name***

创建一个新的解释器，并返回其名称。如果-safe 选项被使用，那么新的解释器不可以访问某些危险的系统功能。

interp delete ***name***

删除对应名称的子解释器。

interp eval *args*

它和普通的 **eval** 命令相似，除了这个事实，即它求的是子解释器的脚本的值，而不是初始解释器的脚本的值。**Interp eval** 命令把参数拼接成一个字符串，并把那行运送给子解释器去求值。

interp alias ***srcPath srcCmd targetPath targetCmd arg arg***

interp alias 命令允许脚本在子解释器或子解释器和初始解释器之间共享过程。

注意，从属的解释器有独立的状态和命名空间。但是**没有**独立的事件循环。这些不是线程，它们不会独立执行。比如，如果一个从属解释器被一个阻塞的 I/O 请求停止，其他的解释器都不会被处理，知道这个解释器解除阻塞。

下面的例子演示了两个被创建在初始解释器{}下面的子解释器。每一个解释器都给出了一个变量 name，它包含对应解释器的名称。

注意，alias 命令使过程在该过程被定义的解释器里面被求值，而不是它要被求值的解释器。如果你需要一个过程存在于一个解释器里面，你必须在那个解释器里面执行 **interp eval** proc 过程命令。如果你想解释器能回调回初始解

释器（或其他解释器），你可以使用 **interp alias** 命令。

示例：

```
set i1 [interp create firstChild]
set i2 [interp create secondChild]

puts "first child interp:  $i1"
puts "second child interp: $i2\n"

# Set a variable "name" in each child interp, and
#  create a procedure within each interp
#  to return that value
foreach int [list $i1 $i2] {
    interp eval $int [list set name $int]
    interp eval $int {proc nameis {} {global name; return
"nameis: $name";} }
}

foreach int [list $i1 $i2] {
    interp eval $int "puts \"EVAL IN $int: name is \$name\""
    puts "Return from 'nameis' is: [interp eval $int nameis]"
}

#
# A short program to return the value of "name"
#
proc rtnName {} {
    global name
    return "rtnName is: $name"
}

#
# Alias that procedure to a proc in $i1
interp alias $i1 rtnName {} rtnName

puts ""

# This is an error.  The alias causes the evaluation
#  to happen in the {} interpreter, where name is
#  not defined.
puts "firstChild reports [interp eval $i1 rtnName]"
```