Tcl 教程

[Edit by roben_chen]

http://2316.vip.nease.net/scriptnet/ssdn/index.htm

目录

Tcl

TCL 语法

脚本、命令和单词符号

一个 TCL 脚本可以包含一个或多个命令。命令之间必须用换行符或分号隔开，下面的两个脚本都是合法的：

```
set a 1
set b 2
```

或

```
set a 1；set b 2
```

TCL 的每一个命令包含一个或几个单词，第一个单词代表命令名，另外的单词则是这个命令的参数，单词之间必须用空格或 TAB 键隔开。
TCL 解释器对一个命令的求值过程分为两部分：分析和执行。在分析阶段，TCL 解释器运用规则把命令分成一个个独立的单词，同时进行必要的置换(substitution)； 在执行阶段，TCL 解释器会把第一个单词当作命令名，并查看这个命令是否有定义，如果有定义就激活这个命令对应的 C/C++过程，并把所有的单词作为参数传递给该命令过程，让命令过程进行处理。

■置换(substitution)

注：在下面的所有章节的例子中，'%'为 TCL 的命令提示符，输入命令回车后，TCL 会在接着的一行输出命令执行结果。'//'后面是我自己加上的说明，不是例子的一部分。

TCL 解释器在分析命令时，把所有的命令参数都当作字符串看待，例如：

%set x 10 //定义变量 x,并把 x 的值赋为 10
10
%set y x+100 //y 的值是 x+100，而不是我们期望的 110
x+100

上例的第二个命令中，x 被看作字符串 x+100 的一部分，如果我们想使用 x 的值'10' ，就必须告诉 TCL 解释器：我们在这里期望的是变量 x 的值，而非字符'x'。怎么告诉 TCL 解释器呢，这就要用到 TCL 语言中提供的置换功能。

TCL 提供三种形式的置换：变量置换、命令置换和反斜杠置换。每种置换都会导致一个或多个单词本身被其他的值所代替。置换可以发生在包括命令名在内的每一个单词中，而且置换可以嵌套。

■变量置换(variable subtitution)

变量置换由一个 $ 符号标记，变量置换会导致变量的值插入一个单词中。例如：

%set y $x+100 //y 的值是 10+100，这里 x 被置换成它的值 10

10+100

这时，y 的值还不是我们想要的值 110，而是 10+100，因为 TCL 解释器把 10+100 看成是一个字符串而不是表达式，y 要想得到值 110，还必须用命令置换，使得 TCL 会把 10+100 看成一个表达式并求值。

■命令置换(command substitution)

命令置换是由[]括起来的 TCL 命令及其参数，命令置换会导致某一个命令的所有或部分单词被另一个命令的结果所代替。例如：

%set y [expr $x+100]

110

y 的值是 110，这里当 TCL 解释器遇到字符'['时，它就会把随后的 expr 作为一个命令名，从而激活与 expr 对应的 C/C++过程，并把'expr'和变量置换后得到的'10+110'传递给该命令过程进行处理。

如果在上例中我们去掉[]，那么 TCL 会报错。因为在正常情况下，TCL 解释器只把命令行中的第一个单词作为看作命令，其他的单词都作为普通字符串处理，看作是命令的参数。

注意，[]中必须是一个合法的 TCL 脚本，长度不限。[]中脚本的值为最后一个命令的返回值，例如：

%set y [expr $x+100;set b 300] //y 的值为 300，因为 set b 300 的返回值为 300
300

有了命令置换，实际上就表示命令之间是可以嵌套的，即一个命令的结果可以作为别的命令的参数。

■反斜杠置换(backslash substitution)

TCL 语言中的反斜杠置换类似于 C 语言中反斜杠的用法，主要用于在单词符号中插入诸如换行符、空格、[、$等被 TCL 解释器当作特殊符号对待的字符。例如：

set msg multiple\ space //msg 的值为 multiple space。

如果没有'\'的话，TCL 会报错，因为解释器会把这里最后两个单词之间的空格认为是分隔符，于是发现 set 命令有多于两个参数，从而报错。加入了'\'后，空格不被当作分隔符，'multiple space'被认为是一个单词(word)。又例如：

%set msg money\ \$3333\ \nArray\ a\[2]
//这个命令的执行结果为：money $3333
Array a[2]
这里的$不再被当作变量置换符。
TCL 支持以下的反斜杠置换:
Backslash Sequence Replaced By
\a Audible alert (0x7)
\b Backspace (0x8)
\f Form feed (0xc)
\n Newline (0xa)
\r Carriage return (0xd)
\t Tab (0x9)
\v Vertical tab (0xb)
\ddd Octal value given by ddd
(one, two, or three d's)
\xhh Hex value given by hh

(any number of h's)
\ newline space A single space character.

例如：

%set a \x48 //对应 \xhh
H //十六进制的 48 正好是 72，对应 H
% set a \110 //对应 \ddd
H //八进制的 110 正好是 72，对应 H
%set a [expr \ // 对应\newline space，一个命令可以用\newline 转到下一行继续
2+3]
5

■双引号和花括号

除了使用反斜杠外，TCL 提供另外两种方法来使得解释器把分隔符和置换符等特殊字符当作普通字符，而不作特殊处理，这就要使用双引号和花括号({})。

TCL 解释器对双引号中的各种分隔符将不作处理，但是对换行符 及 $ 和[]两种置换符会照常处理。例如：

%set x 100
100
%set y "$x ddd"
100 ddd

而在花括号中，所有特殊字符都将成为普通字符，失去其特殊意义，TCL 解释器不会对其作特殊处理。

%set y {/n$x [expr 10+100]}
/n$x [expr 10+100]

■注释

TCL 中的注释符是'#'，'#'和直到所在行结尾的所有字符都被 TCL 看作注释，TCL 解释器对注释将不作任何处理。不过，要注意的是，'#'必须出现在 TCL 解释器期望命令的第一个字符出现的地方，才被当作注释。

例如：

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

第二行中'#'就不被当作注释符，因为它出现在命令的中间，TCL 解释器把它和后面的字符当作命令的参数处理，从而导致错误。而第四行的'#'就被作为注释，因为前一个命令已经用一个分号结束，TCL 解释器期望下一个命令接着出现。现在在这个位置出现'#'，随后的字符就被当作注释了。

■变量

■简单变量

一个 TCL 的简单变量包含两个部分：名字和值。名字和值都可以是任意字符串。例如一个名为"1323 7&*: hdgg"的变量在 TCL 中都是合法的。不过为了更好的使用置换(substitution)，变量名最好按 C\C++语言中标识符的命名规则命名。 TCL 解释器在分析一个变量置换时，只把从 $ 符号往后直到第一个不是字母、数字或下划线的字符之间的单词符号作为要被置换的变量的名字。例如:

```
% set a 2
2
set a.1 4
4
% set b $a.1
2.1
```

在最后一个命令行，我们希望把变量 a.1 的值付给 b，但是 TCL 解释器在分析时只把$符号之后直到第一个不是字母、数字或下划线的字符(这里是'.')之间的单词符号(这里是'a')当作要被置换的变量的名字，所以 TCL 解释器把 a 置换成 2,然后把字符串"2.1"付给变量 b。这显然与我们的初衷不同。

当然，如果变量名中有不是字母、数字或下划线的字符，又要用置换，可以用花括号把变量名括起来。例如：

```
%set b ${a.1}
4
```

TCL 中的 set 命令能生成一个变量、也能读取或改变一个变量的值。例如：

```
% set a {kdfj kjdf}
kdfj kjdf
```

如果变量 a 还没有定义，这个命令将生成 变量 a，并将其值置为 kdfj kjdf，若 a 已定义，就简单的把 a 的值置为 kdfj kjdf。

```
%set a
kdfj kjdf
```

这个只有一个参数的 set 命令读取 a 的当前值 kdfj kjdf。

■数组

数组是一些元素的集合。TCL 的数组和普通计算机语言中的数组有很大的区别。在 TCL 中，不能单独声明一个数组，数组只能和数组元素一起声明。数组中，数组元素的名字包含两部分：数组名和数组中元素的名字，TCL 中数组元素的名字（下标）可以为任何字符串。例如：

```
set day(monday) 1
set day(tuesday) 2
```

第一个命令生成一个名为 day 的数组，同时在数组中生成一个名为 monday 的数组元素，并把值置为 1，第二个命令生成一个名为 tuesday 的数组元素，并把值置为 2。

简单变量的置换已经在前一节讨论过，这里讲一下数组元素的置换。除了有括号之外，数组元素的置换和简单变量类似。例：

```
set a monday
set day(monday) 1
set b $day(monday) //b 的值为 1，即 day(monday)的值。
set c $day($a) //c 的值为 1，即 day(monday)的值。
```

TCL 不能支持复杂的数据类型,这是一个很大的缺憾，也是 TCL 受指责很多的方面。但是 TCL 的一个扩展 ITCL 填补了这个缺憾

■相关命令

■set

这个命令在 3.1 已有详细介绍。

■unset

这个命令从解释器中删除变量，它后面可以有任意多个参数，每个参数是一个变量名,可以是简单变量，也可以是数组或数组元素。例如：

% unset a b day(monday)

上面的语句中删除了变量 a、b 和数组元素 day(monday)，但是数组 day 并没有删除，其他元素还存在，要删除整个数组，只需给出数组的名字。例如：

```
%puts $day(monday)
can't read "day(monday)": no such element in array
% puts $day(tuesday)
2
%unset day
% puts $day(tuesday)
can't read "day(tuesday)": no such variable
```

■append 和 incr

这两个命令提供了改变变量的值的简单手段。

append 命令把文本加到一个变量的后面，例如：

```
% set txt hello
hello
% append txt "! How are you"
hello! How are you
```

incr 命令把一个变量值加上一个整数。incr 要求变量原来的值和新加的值都必须是整数。

```
%set b a
a
% incr b
expected integer but got "a"
%set b 2
```

```
2
%incr b 3
5
```

表达式

■操作数

TCL 表达式的操作数通常是整数或实数。整数一般是十进制的， 但如果整数的第一个字符是 0(zero)，那么 TCL 将把这个整数看作八进制的，如果前两个字符是 0x 则这个整数被看作是十六进制的。TCL 的实数的写法与 ANSI C 中完全一样。如：

2.1
7.9e+12
6e4
3.

■运算符和优先级

下面的表格中列出了 TCL 中用到的运算符，它们的语法形式和用法跟 ANSI C 中很相似。这里就不一一介绍。下表中的运算符是按优先级从高到低往下排列的。同一格中的运算符优先级相同。

语法形式	结果	操作数类型
-a	负 a	int,float
!a	非 a	int,float
~a		int
a*b	乘	int,float
a/b	除	int,float
a%b	取模	int
a+b	加	int,float
a-b	减	int,float
a<<b	左移位	int
a>>b	右移位	int
a<b	小于	int,float,string
a>b	大于	int,float,string
a<=b	小于等于	int,float,string
a>=b	大于等于	int,float,string
a= =b	等于	int,float,string
a!=b	不等于	int,float,string
a&b	位操作与	int
a^b	位操作异或	int
a|b	位操作或	int
a&&b	逻辑与	int,float
a||b	逻辑或	int,float
a?b:c	选择运算	a:int,float

■数学函数

TCL 支持常用的数学函数，表达式中数学函数的写法类似于 C\C++语言的写法，数学函数的参数可以是任意表达式，多个参数之间用逗号隔开。例如：

%set x 2

2

% expr 2* sin($x<3)

1.68294196962

其中 expr 是 TCL 的一个命令，语法为： expr arg ?arg ...?

两个 ？之间的参数表示可省，后面介绍命令时对于可省参数都使用这种表示形式。 expr 可以有一个或多个参数，它把所有的参数组合到一起，作为一个表达式，然后求值：

%expr 1＋2*3

7

%expr 1 +2 *3

7

需要注意的坏闶牵 山 2.皇敲 睿 辉诒浆锸街谐鲂植庞幸庖濉?

TCL 中支持的数学函数如下

abs(x) Absolute value of x.

acos(x) Arc cosine of x, in the range 0 to p.

asin(x) Arc sine of x, in the range -p/2 to p/2.

atan(x) Arc tangent of x, in the range -p/2 to p/2.

atan2(x, y) Arc tangent of x/ y, in the range -p/2 to p/2.

ceil(x) Smallest integer not less than x.

cos(x) Cosine of x (x in radians).

cosh(x) Hyperbolic cosine of x.

double(i) Real value equal to integer i.

exp(x) e raised to the power x.

floor(x) Largest integer not greater than x.

fmod(x, y) Floating-point remainder of x divided by y.

hypot(x, y) Square root of (x 2 + y 2).

int(x) Integer value produced by truncating x.

log(x) Natural logarithm of x.

log10(x) Base 10 logarithm of x.

pow(x, y) x raised to the power y.

round(x) Integer value produced by rounding x.

sin(x) Sine of x (x in radians).

sinh(x) Hyperbolic sine of x.

sqrt(x) Square root of x.

tan(x) Tangent of x (x in radians).

tanh(x) Hyperbolic tangent of x.

TCL 中有很多命令都以表达式作为参数。最典型的是 expr 命令，另外 if、while、for 等循环控制命令的循环控制中也都使用表达式作为参数。

List

■list命令

list 这个概念在 TCL 中是用来表示集合的。TCL 中 list 是由一堆元素组成的有序集合，list 可以嵌套定义，list 每个元素可以是任意字符串，也可以是 list。下面都是 TCL 中的合法的 list：

```
{} //空 list
{a b c d}
{a {b c} d} //list 可以嵌套
```

list 是 TCL 中比较重要的一种数据结构，对于编写复杂的脚本有很大的帮助，TCL 提供了很多基本命令对 list 进行操作，下面一一介绍：

语法： list ? value value...?
这个命令生成一个 list，list 的元素就是所有的 value。例：

```
% list 1 2 {3 4}
1 2 {3 4}
```

■concat命令

语法:concat list ?list...?

这个命令把多个 list 合成一个 list，每个 list 变成新 list 的一个元素。

■lindex命令

语法：lindex list index
返回 list 的第 index 个(0-based)元素。例：

```
% lindex {1 2 {3 4}} 2
3 4
```

■llength命令

语法：llength list
返回 list 的元素个数。例

```
% llength {1 2 {3 4}}
3
```

■linsert命令

语法：linsert list index value ?value...?
返回一个新串，新串是把所有的 value 参数值插入 list 的第 index 个(0-based)元素之前得到。
例:

```
% linsert {1 2 {3 4}} 1 7 8 {9 10}
1 7 8 {9 10} 2 {3 4}
```

■lreplace命令

语法：lreplace list first last ?value value ...?
返回一个新串，新串是把 list 的第 firs (0-based)t 到第 last 个(0-based)元素用所有的 value 参数替换得到的。如果没有 value 参数，就表示删除第 first 到第 last 个元素。例：

```
% lreplace {1 7 8 {9 10} 2 {3 4}} 3 3
1 7 8 2 {3 4}
% lreplace {1 7 8 2 {3 4}} 4 4 4 5 6
1 7 8 2 4 5 6
```

■lrange 命令

语法:lrange list first last
返回 list 的第 first (0-based)到第 last (0-based)元素组成的串,如果 last 的值是 end。就是从第 first 个直到串的最后。

例:
% lrange {1 7 8 2 4 5 6} 3 end
2 4 5 6

■lappend命令

语法：lappend varname value ?value...?
把每个 value 的值作为一个元素附加到变量 varname 后面，并返回变量的新值，如果 varname 不存在，就生成这个变量。例：

```
% lappend a 1 2 3
1 2 3
% set a
1 2 3
```

■lsearch 命令

语法：lsearch ?-exact? ?-glob? ?-regexp? list pattern
返回 list 中第一个匹配模式 pattern 的元素的索引，如果找不到匹配就返回-1。-exact、-glob、-regexp 是三种模式匹配的技术。-exact 表示精确匹配；-glob 的匹配方式和 string match 命令的匹配方式相同，将在后面第八节介绍 string 命令时介绍；-regexp 表示正规表达式匹配，将在第八节介绍 regexp 命令时介绍。缺省时使用-glob 匹配。例:

```
% set a { how are you }
how are you
% lsearch $a y*
2
% lsearch $a y?
-1
```

■lsort命令

语法：lsort ?options? list
这个命令返回把 list 排序后的串。options 可以是如下值：

-ascii 按 ASCII 字符的顺序排序比较.这是缺省情况。

-dictionary 按字典排序,与-ascii 不同的地方是：
(1)不考虑大小写
(2)如果元素中有数字的话,数字被当作整数来排序.
因此：bigBoy 排在 bigbang 和 bigboy 之间, x10y 排在 x9y 和 x11y 之间.

-integer 把 list 的元素转换成整数,按整数排序.

-real 把 list 的元素转换成浮点数,按浮点数排序.

-increasing 升序(按 ASCII 字符比较)

-decreasing 降序(按 ASCII 字符比较)

-command command TCL 自动利用 command 命令把每两个元素一一比较,然后给出排序结果。

■split命令

语法：split string ?splitChars?
把字符串 string 按分隔符 splitChars 分成一个个单词，返回由这些单词组成的串。如果 splitChars
是一个空字符{}，string 被按字符分开。如果 splitChars 没有给出,以空格为分隔符。例：

```
% split "how.are.you" .
how are you
% split "how are you"
how are you
% split "how are you" {}
h o w { } a r e { } y o u
```

■join命令

语法:join list ?joinString?
join 命令是命令的逆。这个命令把 list 的所有元素合并到一个字符串中，中间以 joinString 分开。缺省的 joinString 是空格。例：

```
% join {h o w { } a r e { } y o u} {}
how are you
% join {how are you} .
how.are.you
```

控制流

■if命令

TCL 中的控制流和 C 语言类似，包括 if、while、for、foreach、switch、break、continue 等命令。

语法： if test1 body1 ?elseif test2 body2 elseif.... ? ?else bodyn?
TCL 先把 test1 当作一个表达式求值，如果值非 0，则把 body1 当作一个脚本执行并返回所得值，否则把 test2 当作一个表达式求值，如果值非 0，则把 body2 当作一个脚本执行并返回所得值……。例如：

```
if { $x>0 } {
.....
}elseif{ $x==1 } {
.....
}elseif { $x==2 } {
....
}else{
.....
}
```

注意，上例中'{'一定要写在上一行，因为如果不这样，TCL 解释器会认为 if 命令在换行符处已结束，下一行会被当成新的命令，从而导致错误的结果。在下面的循环命令的书写中也要注意这个问题。书写中还要注意的一个问题是 if 和{之间应该有一个空格，否则 TCL 解释器会把'if{'作为一个整体当作一个命令名，从而导致错误

■循环命令：while 、for、 foreach

循环命令包括 while、for、foreach 等。

■while 命令

语法为：while test body
参数 test 是一个表达式，body 是一个脚本，如果表达式的值非 0，就运行脚本，直到表达式为 0 才停止循环，此时 while 命令中断并返回一个空字符串。

例如：
假设变量 a 是一个链表，下面的脚本把 a 的值复制到 b：

```
set b " "
set i [expr [llength $a] -1]
while { $i>=0}{
lappend b [lindex $a $i]
incr i -1
}
```

■for 命令

语法为：for init test reinit body
参数 init 是一个初始化脚本，第二个参数 test 是一个表达式，用来决定循环什么时候中断，第三个参数 reinit 是一个重新初始化的脚本，第四个参数 body 也是脚本， 硌 诽濉O吕 肽侠 褰孟嗤 ?

```
set b " "
for {set i [expr [llength $a] -1]} {$i>=0} {incr i -1} {
lappend b [lindex $a $i] }
```

■foreach 命令

这个命令有两种语法形式

1、 foreach varName list body
第一个参数 varName 是一个变量，第二个参数 list 是一个表(有序集合)，第三个参数 body 是循环体。每次取得链表的一个元素，都会执行循环体一次。 下例与上例作用相同：

```
set b " "
foreach i $a{
```

```
set b [linsert $b 0 $i]
}
```

2、 foreach varlist1 list1 ?varlist2 list2 ...? Body

这种形式包含了第一种形式。第一个参数 varlist1 是一个循环变量列表，第二个参数是一个列表 list1，varlist1 中的变量会分别取 list1 中的值。body 参数是循环体。 ?varlist2 list2 ...? 表示可以有多个变量列表和列表对出现。例如：

```
set x {}
foreach {i j} {a b c d e f} {
lappend x $j $i
}
```

这时总共有三次循环，x 的值为"b a d c f e"。

```
set x {}
foreach i {a b c} j {d e f g} {
lappend x $i $j
}
```

这时总共有四次循环，x 的值为"a d b e c f {} g"。

```
set x {}
foreach i {a b c} {j k} {d e f g} {
lappend x $i $j $k
}
```

这时总共有三次循环，x 的值为"a d e b f g c {} {}"。

■break 和 continue 命令

在循环体中，可以用 break 和 continue 命令中断循环。其中 break 命令结束整个循环过程，并从循环中跳出，continue 只是结束本次循环。

■switch 命令

和 C 语言中 switch 语句一样，TCL 中的 switch 命令也可以由 if 命令实现。只是书写起来较为烦琐。 switch 命令的语法为: switch ? options? string { pattern body ? pattern body ...?}

第一个是可选参数 options，表示进行匹配的方式。TCL 支持三种匹配方式：-exact 方式，-glob 方式，-regexp 方式,缺省情况表示-glob 方式。-exact 方式表示的是精确匹配，-glob 方式的匹配方式和 string match 命令的匹配方式相同(第八节介绍)，-regexp 方式是正规表达式的

匹配方式(第八节介绍)。第二个参数 string 是要被用来作测试的值，第三个参数是括起来的一个或多个元素对，例：

```
switch $x {
a -
b {incr t1}
c {incr t2}
default {incr t3}
}
```

其中 a 的后面跟一个'－'表示使用和下一个模式相同的脚本。default 表示匹配任意值。一旦 switch 命令 找到一个模式匹配，就执行相应的脚本，并返回脚本的值，作为 switch 命令的返回值。

■eval命令

eval 命令是一个用来构造和执行 TCL 脚本的命令，其语法为：
eval arg ?arg ...?

它可以接收一个或多个参数，然后把所有的参数以空格隔开组合到一起成为一个脚本，然后对这个脚本进行求值。例如：

%eval set a 2 ;set b 4
4

■source命令

source 命令读一个文件并把这个文件的内容作为一个脚本进行求值。例如：

source e:/tcl&c/hello.tcl

注意路径的描述应该和 UNIX 相同，使用'/'而不是'\'。

过程(procedure)

■过程定义和返回值

TCL 支持过程的定义和调用，在 TCL 中,过程可以看作是用 TCL 脚本实现的命令，效果与 TCL 的固有命令相似。我们可以在任何时候使用 proc 命令定义自己的过程，TCL 中的过程类似于 C 中的函数。

TCL 中过程是由 proc 命令产生的：

例如：

```
% proc add {x y } {expr $x+$y}
```

proc 命令的第一个参数是你要定义的过程的名字，第二个参数是过程的参数列表，参数之间用空格隔开，第三个参数是一个 TCL 脚本，代表过程体。 proc 生成一个新的命令，可以象固有命令一样调用：

```
% add 1 2
3
```

在定义过程时，你可以利用 return 命令在任何地方返回你想要的值。 return 命令迅速中断过程，并把它的参数作为过程的结果。例如：

```
% proc abs {x} {
if {$x >= 0} { return $x }
return [expr -$x]
}
```

过程的返回值是过程体中最后执行的那条命令的返回值。

■局部变量和全局变量

对于在过程中定义的变量，因为它们只能在过程中被访问，并且当过程退出时会被自动删除，所以称为局部变量；在所有过程之外定义的变量我们称之为全局变量。TCL 中，局部变量和全局变量可以同名，两者的作用域的交集为空：局部变量的作用域是它所在的过程的内部；全局变量的作用域则不包括所有过程的内部。这一点和 C 语言有很大的不同.

如果我们想在过程内部引用一个全局变量的值，可以使用 global 命令。例如：

```
% set a 4
4
% proc sample { x } {
global a
incr a
return [expr $a+$x]
}
% sample 3
8
%set a
5
```

全局变量 a 在过程中被访问。在过程中对 a 的改变会直接反映到全局上。如果去掉语句 global a，TCL 会出错，因为它不认识变量 a。

■缺省参数和可变个数参数

TCL 还提供三种特殊的参数形式：
首先，你可以定义一个没有参数的过程，例如：

```
proc add {} { expr 2+3}
```

其次，可以定义具有缺省参数值的过程，我们可以为过程的部分或全部参数提供缺省值，如果调用过程时未提供那些参数的值，那么过程会自动使用缺省值赋给相应的参数。和 C\C++中具有缺省参数值的函数一样，有缺省值的参数只能位于参数列表的后部，即在第一个具有缺省值的参数后面的所有参数，都只能是具有缺省值的参数。

例如：

```
proc add {val1 {val2 2} {val3 3}}{
expr $val1+$val2+$val3
}
```

则：

```
add 1 //值为 6
add 2 20 //值为 25
add 4 5 6 //值为 15
```

另外，TCL 的过程定义还支持可变个数的参数，如果过程的最后一个参数是 args，那么就表示这个过程支持可变个数的参 饕谩 5 饕檬?位于 args 以前的参数象普通参数一样处理，但任何附加的参数都需要在过程体中作特殊处理，过程的局部变量 args 将会被设置为一个列表，其元素就是所有附加的变量。如果没有附加的变量，args 就设置成一个空串，下面是一个例子：

```
proc add { val1 args } {
set sum $val1
foreach i $args {
incr sum $i
}
return $sum
}
```

则：

```
add 2 //值为 2
add 2 3 4 5 6 //值为 20
```

■引用：upvar

命令语法:upvar ?level? otherVar myVar ?otherVar myVar ...?
upvar 命令使得用户可以在过程中对全局变量或其他过程中的局部变量进行访问。 upvar 命令的第一个参数 otherVar 是我们希望以引用方式访问的参数的名字，第二个参数 myVar 是这个过程中的局部变量的名字，一旦使用了 upvar 命令把 otherVar 和 myVar 绑定,那么在过程中对局部变量 myVar 的读写就相当于对这个过程的调用者中 otherVar 所代表的局部变量的读写。下面是一个例子：

```
% proc temp { arg } {
upvar $arg b
set b [expr $b+2]
}
% proc myexp { var } {
set a 4
temp a
return [expr $var+$a]
}
```

则：

```
% myexp 7
13
```

这个例子中，upvar 把$arg(实际上是过程 myexp 中的变量 a)和过程 temp 中的变量 b 绑定，对 b 的读写就相当于对 a 的读写。

upvar 命令语法中的 level 参数表示：调用 upvar 命令的过程相对于我们希望引用的变量 myVar 在调用栈中相对位置。例如：

```
upvar 2 other x
```

这个命令使得当前过程的调用者的调用者中的变量 other，可以在当前过程中利用 x 访问。缺省情况下，level 的值为 1，即当前过程(上例中的 temp)的调用者(上例中的 myexp)中的变量(上例中 myexp 的 a)可以在当前过程中利用局部变量(上例中 temp 的 b)访问。

如果要访问全局变量可以这样写：

```
upvar  #0 other x
```

那么，不管当前过程处于调用栈中的什么位置，都可以在当前过程中利用 x 访问全局变量 other。

字符串操作

■format命令

因为 TCL 把所有的输入都当作字符串看待，所以 TCL 提供了较强的字符串操作功能，TCL 中与字符串操作有关的命令有：string、format、regexp、regsub、scan 等。

format 命令

语法：format formatstring ?vlue value...?
format 命令类似于 ANSIC 中的 sprintf 函数和 MFC 中 CString 类提供的 Format 成员函数。它按 formatstring 提供的格式，把各个 value 的值组合到 formatstring 中形成一个新字符串，并返回。例如：

```
%set name john
John
%set age 20
20
%set msg [format "%s is %d years old" $name $age]
john is 20 years old
```

■scan命令

语法：scan string format varName ?varName ...?
scan 命令可以认为是 format 命令的逆，其功能类似于 ANSI C 中的 sscanf 函数。它按 format 提供的格式分析 string 字符串，然后把结果存到变量 varName 中,注意除了空格和 TAB 键之外，string 和 format 中的字符和'%'必须匹配。例如：

```
% scan "some 26 34" "some %d %d" a b
2
% set a
26
% set b
34
% scan "12.34.56.78" "%d.%d.%d.%d" c d e f
4
% puts [format "the value of c is %d,d is %d,e is %d ,f is %d" $c $d $e $f]
the value of c is 12,d is 34,e is 56 ,f is 78
```

scan 命令的返回值是匹配的变量个数。而且，我们发现，如果变量 varName 不存在的话，TCL 会自动声明该变量。

■regexp命令

语法：regexp ?switchs? ?--? exp string ?matchVar?\ ?subMatchVar subMatchVar...?
regexp 命令用于判断正规表达式 exp 是否全部或部分匹配字符串 string，匹配返回 1，否则 0。

在正规表达式中，一些字符具有特殊的含义，下表一一列出，并给予了解释。

字符	意义
.	匹配任意单个字符
^	表示从头进行匹配
$	表示从末尾进行匹配
\x	匹配字符 x，这可以抑制字符 x 的含义
[chars]	匹配字符集合 chars 中给出的任意字符，如果 chars 中的第一个字符是^，表示匹配任意不在 chars 中的字符，chars 的表示方法支持 a-z 之类的表示。
(regexp)	把 regexp 作为一个单项进行匹配
*	对*前面的项 0 进行次或多次匹配
+	对＋前面的项进行 1 次或多次匹配
?	对？前面的项进行 0 次或 1 次匹配
regexp1|regexp2	匹配 regexp1 或 regexp2 中的一项

下面的一个例子是从《Tcl and Tk ToolKit》中摘下来的，下面进行说明：

^((0x)?[0-9a-fA-F]+|[0-9]+)$

这个正规表达式匹配任何十六进制或十进制的整数。

两个正规表达式以|分开(0x)？[0-9a-fA-F]+和[0-9]+，表示可以匹配其中的任何一个，事实上前者匹配十六进制，后者匹配的十进制。

^表示必须从头进行匹配，从而上述正规表达式不匹配 jk12 之类不是以 0x 或数字开头的串。

＄表示必须从末尾开始匹配，从而上述正规表达式不匹配 12jk 之类不是数字或 a-fA-F 结尾的串。

下面以(0x)？[0-9a-fA-F]+ 进行说明，(0x)表示 0x 一起作为一项，?表示前一项(0x)可以出现 0 次或多次，[0-9a-fA-F]表示可以是任意 0 到 9 之间的单个数字或 a 到 f 或 A 到 F 之间的单个字母，＋表示象前面那样的单个数字或字母可以重复出现一次或多次。

```
% regexp {^((0x)?[0-9a-fA-F]+|[0-9]+)$} ab
1
```

```
% regexp {^((0x)?[0-9a-fA-F]+|[0-9]+)$} 0xabcd
1
% regexp {^((0x)?[0-9a-fA-F]+|[0-9]+)$} 12345
1
% regexp {^((0x)?[0-9a-fA-F]+|[0-9]+)$} 123j
0
```

如果 regexp 命令后面有参数 matchVar 和 subMatchVar，则所有的参数被当作变量名，如果变量不存在，就会被生成。 regexp 把匹配整个正规表达式的子字符串赋给第一个变量，匹配正规表达式的最左边的子表达式的子字符串赋给第二个变量，依次类推，例如：

```
% regexp { ([0-9]+) *([a-z]+)} " there is 100 apples" total num word
1
% puts " $total ,$num,$word"
100 apples ,100,apples
```

regexp 可以设置一些开关（switchs），来控制匹配结果：

开关	意义
-nocase	匹配时不考虑大小写
-indices	改变各个变量的值，这使各个变量的值变成了对应的匹配子串在整个字符串中所处位置的索引。例如： % regexp -indices { ([0-9]+) *([a-z]+)} " there is 100 apples" total num word 1 % puts " $total ,$num,$word" 9 20 ,10 12,15 20 正好子串“ 100 apples”的序号是 9-20,"100"的序号是 10-12,"apples"的序号是 15-20
-about	返回正则表达式本身的信息，而不是对缓冲区的解析。返回的是一个 list，第一个元素是子表达式的个数，第二个元素开始存放子表达式的信息
-expanded	启用扩展的规则，将空格和注释忽略掉，相当于使用内嵌语法(?x)
-line	启用行敏感匹配。正常情况下^和$只能匹配缓冲区起始和末尾，对于缓冲区内部新的行是不能匹配的，通过这个开关可以使缓冲区内部新的行也可以被匹配。它相当于同时使用**-linestop** 和**-lineanchor** 开关，或者使用内嵌语法(?n)
-linestop	启动行结束敏感开关。使^可以匹配缓冲区内部的新行。相当于内嵌语法(?p)
-lineanchor	改变^和$的匹配行为，使可以匹配缓冲区内部的新行。相当于内嵌语法(?w)
-all	进最大可能的匹配
-inline	Causes the command to return, as a list, the data that would

	otherwise be placed in match variables. When using -inline, match variables may not be specified. If used with -all, the list will be concatenated at each iteration, such that a flat list is always returned. For each match iteration, the command will append the overall match data, plus one element for each subexpression in the regular expression. Examples are: regexp -inline -- {\w(\w)} " inlined " => {in n} regexp -all -inline -- {\w(\w)} " inlined " => {in n li i ne e}
-**start** index	强制从偏移为 index 开始的位置进行匹配。使用这个开关之后，^将不能匹配行起始位置，\A 将匹配字符串的 index 偏移位置。如果使用了-**indices** 开关，则 indices 表示绝对位置，index 表示输入字符的相对位置。
--	表示这后面再没有开关（switchs）了，即使后面有以'-'开头的参数也被当作正规表达式的一部分。

【TCL 正则表达式规则详细说明】

◆DESCRIPTION（描述）

A regular expression describes strings of characters. It's a pattern that matches certain strings and doesn't match others.

◆DIFFERENT FLAVORS OF REs（和标准正则表达式的区别）

Regular expressions, as defined by POSIX, come in two flavors: extended REs and basic REs. EREs are roughly those of the traditional egrep, while BREs are roughly those of the traditional ed. This implementation adds a third flavor, advanced REs, basically EREs with some significant extensions.

This manual page primarily describes AREs. BREs mostly exist for backward compatibility in some old programs; they will be discussed at the end. POSIX EREs are almost an exact subset of AREs. Features of AREs that are not present in EREs will be indicated.

◆REGULAR EXPRESSION SYNTAX（语法）

Tcl regular expressions are implemented using the package written by Henry Spencer, based on the 1003.2 spec and some (not quite all) of the Perl5 extensions (thanks, Henry!). Much of the description of regular expressions below is copied verbatim from his manual entry.
An ARE is one or more branches, separated by `|', matching anything that matches any of the branches.
A branch is zero or more constraints or quantified atoms, concatenated. It matches a match for the first, followed by a match for the second, etc; an empty branch matches the empty string.

A quantified atom is an atom possibly followed by a single quantifier. Without a quantifier, it matches a match for the atom. The quantifiers, and what a so-quantified atom matches, are:

字符	意义
*	a sequence of 0 or more matches of the atom
+	a sequence of 1 or more matches of the atom
?	a sequence of 0 or 1 matches of the atom
{m}	a sequence of exactly m matches of the atom
{m,}	a sequence of m or more matches of the atom
{m,n}	a sequence of m through n (inclusive) matches of the atom; m may not exceed n
*? +? ?? {m}? {m,}? {m,n}?	non-greedy quantifiers, which match the same possibilities, but prefer the smallest number rather than the largest number of matches (see MATCHING)

The forms using { and } are known as bounds. The numbers m and n are unsigned decimal integers with permissible values from 0 to 255 inclusive.
An atom is one of:

字符	意义
(re)	(where re is any regular expression) matches a match for re, with the match noted for possible reporting
(?:re)	as previous, but does no reporting
()	matches an empty string, noted for possible reporting
(?:)	matches an empty string, without reporting
[chars]	a bracket expression, matching any one of the chars (see BRACKET EXPRESSIONS for more detail)

.	matches any single character
\k	where k is a non-alphanumeric character) matches that character taken as an ordinary character, e.g. \\ matches a backslash character
\c	where c is alphanumeric (possibly followed by other characters), an escape (AREs only), see ESCAPES below
{	when followed by a character other than a digit, matches the left-brace character `{'; when followed by a digit, it is the beginning of a bound (see above)
x	where x is a single character with no other significance, matches that character.

A constraint matches an empty string when specific conditions are met. A constraint may not be followed by a quantifier. The simple constraints are as follows; some more constraints are described later, under ESCAPES.

字符	意义
^	matches at the beginning of a line
$	matches at the end of a line
(?=re)	positive lookahead (AREs only), matches at any point where a substring matching re begins
(?!re)	negative lookahead (AREs only), matches at any point where no substring matching re begins

The lookahead constraints may not contain back references (see later), and all parentheses within them are considered non-capturing.
An RE may not end with `\'.

◆BRACKET EXPRESSIONS（预定义表达式）

A bracket expression is a list of characters enclosed in `[]'. It normally matches any single character from the list (but see below). If the list begins with `^', it matches any single character (but see below) not from the rest of the list.

If two characters in the list are separated by `-', this is shorthand for the full range of characters between those two (inclusive) in the collating sequence, e.g. [0-9] in ASCII matches any decimal digit. Two ranges may not share an endpoint, so e.g. a-c-e is illegal. Ranges are very collating-sequence-dependent, and portable programs should avoid relying on them.
To include a literal] or - in the list, the simplest method is to enclose it in [. and .] to make it a collating element (see below). Alternatively, make it the first character (following a possible `^'), or (AREs only) precede it with `\'. Alternatively, for `-',

make it the last character, or the second endpoint of a range. To use a literal - as the first endpoint of a range, make it a collating element or (AREs only) precede it with `\'. With the exception of these, some combinations using [(see next paragraphs), and escapes, all other special characters lose their special significance within a bracket expression.

Within a bracket expression, a collating element (a character, a multi-character sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in [. and .] stands for the sequence of characters of that collating element. The sequence is a single element of the bracket expression's list. A bracket expression in a locale that has multi-character collating elements can thus match more than one character. So (insidiously), a bracket expression that starts with ^ can match multi-character collating elements even if none of them appear in the bracket expression! (Note: Tcl currently has no multi-character collating elements. This information is only for illustration.)

For example, assume the collating sequence includes a ch multi-character collating element. Then the RE [[.ch.]]*c (zero or more ch's followed by c) matches the first five characters of `chchcc'. Also, the RE [^c]b matches all of `chb' (because [^c] matches the multi-character ch).
Within a bracket expression, a collating element enclosed in [= and =] is an equivalence class, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were `[.' and `.]'.) For example, if o and ?are the members of an equivalence class, then `[[=o=]]', `[[=?]]', and `[o' are all synonymous. An equivalence class may not be an endpoint of a range. (Note: Tcl currently implements only the Unicode locale. It doesn't define any equivalence classes. The examples above are just illustrations.)

Within a bracket expression, the name of a character class enclosed in [: and :] stands for the list of all characters (not all collating elements!) belonging to that class. Standard character classes are:

字符	意义
alpha	A letter.
upper	An upper-case letter.
lower	A lower-case letter.
digit	A decimal digit.
xdigit	A hexadecimal digit.
alnum	An alphanumeric (letter or digit).
print	An alphanumeric (same as alnum).
blank	A space or tab character.

space	A character producing white space in displayed text.
punct	A punctuation character.
graph	A character with a visible representation.
cntrl	A control character.

A locale may provide others. (Note that the current Tcl implementation has only one locale: the Unicode locale.) A character class may not be used as an endpoint of a range.
There are two special cases of bracket expressions: the bracket expressions [[:<:]] and [[:>:]] are constraints, matching empty strings at the beginning and end of a word respectively. A word is defined as a sequence of word characters that is neither preceded nor followed by word characters. A word character is an alnum character or an underscore (_). These special bracket expressions are deprecated; users of AREs should use constraint escapes instead (see below).

◆ESCAPES（转意字符）

Escapes (AREs only), which begin with a \ followed by an alphanumeric character, come in several varieties: character entry, class shorthands, constraint escapes, and back references. A \ followed by an alphanumeric character but not constituting a valid escape is illegal in AREs. In EREs, there are no escapes: outside a bracket expression, a \ followed by an alphanumeric character merely stands for that character as an ordinary character, and inside a bracket expression, \ is an ordinary character. (The latter is the one actual incompatibility between EREs and AREs.)

Character-entry escapes (AREs only) exist to make it easier to specify non-printing and otherwise inconvenient characters in REs:

字符	意义
\a	alert (bell) character, as in C
\b	backspace, as in C
\B	synonym for \ to help reduce backslash doubling in some applications where there are multiple levels of backslash processing
\cX	(where X is any character) the character whose low-order 5 bits are the same as those of X, and whose other bits are all zero
\e	the character whose collating-sequence name is `ESC', or failing that, the character with octal value 033
\f	formfeed, as in C
\n	newline, as in C
\r	carriage return, as in C

t	horizontal tab, as in C
uwxyz	(where wxyz is exactly four hexadecimal digits) the Unicode character U+wxyz in the local byte ordering
Ustuvwxyz	(where stuvwxyz is exactly eight hexadecimal digits) reserved for a somewhat-hypothetical Unicode extension to 32 bits
v	vertical tab, as in C are all available.
xhhh	(where hhh is any sequence of hexadecimal digits) the character whose hexadecimal value is 0xhhh (a single character no matter how many hexadecimal digits are used).
0	the character whose value is 0
\xy	(where xy is exactly two octal digits, and is not a back reference (see below)) the character whose octal value is 0xy
\xyz	(where xyz is exactly three octal digits, and is not a back reference (see below)) the character whose octal value is 0xyz

Hexadecimal digits are `0'-`9', `a'-`f', and `A'-`F'. Octal digits are `0'-`7'.
The character-entry escapes are always taken as ordinary characters. For example, \135 is] in ASCII, but \135 does not terminate a bracket expression. Beware, however, that some applications (e.g., C compilers) interpret such sequences themselves before the regular-expression package gets to see them, which may require doubling (quadrupling, etc.) the `\'.

Class-shorthand escapes (AREs only) provide shorthands for certain commonly-used character classes:

缩写	代表的完整表达式
\d	[[:digit:]]
\s	[[:space:]]
\w	[[:alnum:]_] (note underscore)
\D	[^[:digit:]]
\S	[^[:space:]]
\W	[^[:alnum:]_] (note underscore)

Within bracket expressions, `\d', `\s', and `\w' lose their outer brackets, and `\D', `\S', and `\W' are illegal. (So, for example, [a-c\d] is equivalent to [a-c[:digit:]]. Also, [a-c\D], which is equivalent to [a-c^[:digit:]], is illegal.)
A constraint escape (AREs only) is a constraint, matching the empty string if specific conditions are met, written as an escape:

字符	意义

\A	matches only at the beginning of the string (see MATCHING, below, for how this differs from `^')
\m	matches only at the beginning of a word
\M	matches only at the end of a word
\y	matches only at the beginning or end of a word
\Y	matches only at a point that is not the beginning or end of a word
\Z	matches only at the end of the string (see MATCHING, below, for how this differs from `$')
\m	(where m is a nonzero digit) a back reference, see below
\mnn	(where m is a nonzero digit, and nn is some more digits, and the decimal value mnn is not greater than the number of closing capturing parentheses seen so far) a back reference, see below

A word is defined as in the specification of [[:<:]] and [[:>:]] above. Constraint escapes are illegal within bracket expressions.
A back reference (AREs only) matches the same string matched by the parenthesized subexpression specified by the number, so that (e.g.) ([bc])\1 matches bb or cc but not `bc'. The subexpression must entirely precede the back reference in the RE. Subexpressions are numbered in the order of their leading parentheses. Non-capturing parentheses do not define subexpressions.

There is an inherent historical ambiguity between octal character-entry escapes and back references, which is resolved by heuristics, as hinted at above. A leading zero always indicates an octal escape. A single non-zero digit, not followed by another digit, is always taken as a back reference. A multi-digit sequence not starting with a zero is taken as a back reference if it comes after a suitable subexpression (i.e. the number is in the legal range for a back reference), and otherwise is taken as octal.

◆METASYNTAX（内嵌语法）

In addition to the main syntax described above, there are some special forms and miscellaneous syntactic facilities available.

Normally the flavor of RE being used is specified by application-dependent means. However, this can be overridden by a director. If an RE of any flavor begins with `***:', the rest of the RE is an ARE. If an RE of any flavor begins with `***=', the rest of the RE is taken to be a literal string, with all characters considered ordinary characters.
An ARE may begin with embedded options: a sequence (?xyz) (where xyz is one or more alphabetic characters) specifies options affecting the rest of the RE. These

supplement, and can override, any options specified by the application. The available option letters are:

字符	意义
b	rest of RE is a BRE
c	case-sensitive matching (usual default)
e	rest of RE is an ERE
i	case-insensitive matching (see MATCHING, below)
m	historical synonym for n
n	newline-sensitive matching (see MATCHING, below)
p	partial newline-sensitive matching (see MATCHING, below)
q	rest of RE is a literal string, all ordinary characters
s	non-newline-sensitive matching (usual default)
t	tight syntax (usual default; see below)
w	inverse partial newline-sensitive matching (see MATCHING, below)
x	expanded syntax (see below)

Embedded options take effect at the) terminating the sequence. They are available only at the start of an ARE, and may not be used later within it.
In addition to the usual (tight) RE syntax, in which all characters are significant, there is an expanded syntax, available in all flavors of RE with the -expanded switch, or in AREs with the embedded x option. In the expanded syntax, white-space characters are ignored and all characters between a # and the following newline (or the end of the RE) are ignored, permitting paragraphing and commenting a complex RE. There are three exceptions to that basic rule:

a white-space character or `#' preceded by `\' is retained
white space or `#' within a bracket expression is retained
white space and comments are illegal within multi-character symbols like the ARE `(?:' or the BRE `\('

Expanded-syntax white-space characters are blank, tab, newline, and any character that belongs to the space character class.
Finally, in an ARE, outside bracket expressions, the sequence `(?#ttt)' (where ttt is any text not containing a `)') is a comment, completely ignored. Again, this is not allowed between the characters of multi-character symbols like `(?:'. Such comments are more a historical artifact than a useful facility, and their use is deprecated; use the expanded syntax instead.

None of these metasyntax extensions is available if the application (or an initial ***= director) has specified that the user's input be treated as a literal string rather than as an RE.

◆MATCHING（匹配）

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, its choice is determined by its preference: either the longest substring, or the shortest.

Most atoms, and all constraints, have no preference. A parenthesized RE has the same preference (possibly none) as the RE. A quantified atom with quantifier {m} or {m}? has the same preference (possibly none) as the atom itself. A quantified atom with other normal quantifiers (including {m,n} with m equal to n) prefers longest match. A quantified atom with other non-greedy quantifiers (including {m,n}? with m equal to n) prefers shortest match. A branch has the same preference as the first quantified atom in it which has a preference. An RE consisting of two or more branches connected by the | operator prefers longest match.

Subject to the constraints imposed by the rules for matching the whole RE, subexpressions also match the longest or shortest possible substrings, based on their preferences, with subexpressions starting earlier in the RE taking priority over ones starting later. Note that outer subexpressions thus take priority over their component subexpressions.
Note that the quantifiers {1,1} and {1,1}? can be used to force longest and shortest preference, respectively, on a subexpression or a whole RE.

Match lengths are measured in characters, not collating elements. An empty string is considered longer than no match at all. For example, bb* matches the three middle characters of `abbbc', (week|wee)(night|knights) matches all ten characters of `
weeknights', when (.*).* is matched against abc the parenthesized subexpression matches all three characters, and when (a*)* is matched against bc both the whole RE and the parenthesized subexpression match an empty string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, so that x becomes `[xX]'. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, so that [x] becomes [xX] and [^x] becomes `[^xX]'.

If newline-sensitive matching is specified, . and bracket expressions using ^ will never match the newline character (so that matches will never cross newlines unless the RE explicitly arranges it) and ^ and $ will match the empty string after and before a newline respectively, in addition to matching at beginning and end of string respectively. ARE \A and \Z continue to match beginning or end of string only. If partial newline-sensitive matching is specified, this affects . and bracket expressions as with newline-sensitive matching, but not ^ and `$'.

If inverse partial newline-sensitive matching is specified, this affects ^ and $ as with newline-sensitive matching, but not . and bracket expressions. This isn't very useful but is provided for symmetry.

◆LIMITS AND COMPATIBILITY（限制和兼容性）

No particular limit is imposed on the length of REs. Programs intended to be highly portable should not employ REs longer than 256 bytes, as a POSIX-compliant implementation can refuse to accept such REs.

The only feature of AREs that is actually incompatible with POSIX EREs is that \ does not lose its special significance inside bracket expressions. All other ARE features use syntax which is illegal or has undefined or unspecified effects in POSIX EREs; the *** syntax of directors likewise is outside the POSIX syntax for both BREs and EREs.
Many of the ARE extensions are borrowed from Perl, but some have been changed to clean them up, and a few Perl extensions are not present. Incompatibilities of note include `\b', `\B', the lack of special treatment for a trailing newline, the addition of complemented bracket expressions to the things affected by newline-sensitive matching, the restrictions on parentheses and back references in lookahead constraints, and the longest/shortest-match (rather than first-match) matching semantics.

The matching rules for REs containing both normal and non-greedy quantifiers have changed since early beta-test versions of this package. (The new rules are much simpler and cleaner, but don't work as hard at guessing the user's real intentions.) Henry Spencer's original 1986 regexp package, still in widespread use (e.g., in pre-8.1 releases of Tcl), implemented an early version of today's EREs. There are four incompatibilities between regexp's near-EREs (`RREs' for short) and AREs. In roughly increasing order of significance:

In AREs, \ followed by an alphanumeric character is either an escape or an error, while in RREs, it was just another way of writing the alphanumeric. This should not be a problem because there was no reason to write such a sequence in RREs.

{ followed by a digit in an ARE is the beginning of a bound, while in RREs, { was always an ordinary character. Such sequences should be rare, and will often result in an error because following characters will not look like a valid bound.

In AREs, \ remains a special character within `[]', so a literal \ within [] must be written `\\'. \\ also gives a literal \ within [] in RREs, but only truly paranoid programmers routinely doubled the backslash.
AREs report the longest/shortest match for the RE, rather than the first found in a specified search order. This may affect some RREs which were written in the expectation that the first match would be reported. (The careful crafting of RREs to optimize the search order for fast matching is obsolete (AREs examine all possible matches in parallel, and their performance is largely insensitive to their complexity) but cases where the search order was exploited to deliberately find a match which was not the longest/shortest will need rewriting.)

◆BASIC REGULAR EXPRESSIONS（基本正则表达式）

BREs differ from EREs in several respects. `|', `+', and ? are ordinary characters and there is no equivalent for their functionality. The delimiters for bounds are \{ and `\}', with { and } by themselves ordinary characters. The parentheses for nested subexpressions are \( and `\)', with (and) by themselves ordinary characters. ^ is an ordinary character except at the beginning of the RE or the beginning of a parenthesized subexpression, $ is an ordinary character except at the end of the RE or the end of a parenthesized subexpression, and * is an ordinary character if it appears at the beginning of the RE or the beginning of a parenthesized subexpression (after a possible leading `^'). Finally, single-digit back references are available, and \< and \> are synonyms for [[:<:]] and [[:>:]] respectively; no other escapes are available.

■regsub命令

语法：regsub ?switchs? exp string subSpec varname
regsub 的第一个参数是一个整个表达式，第二个参数是一个输入字符串，这一点和 regexp 命令完全一样，也是当匹配时返回 1，否则返回 0。不过 regsub 用第三个参数的值来替换字符串 string 中和正规表达式匹配的部分，第四个参数被认为是一个变量，替换后的字符串存入这个变量中。例如：

```
% regsub there "They live there lives " their x
1
% puts $x
They live their lives
```

这里 there 被用 their 替换了。

regsub 命令也有几个开关(switchs):

-nocase 意义同 regexp 命令中。

-all 没有这个开关时，regsub 只替换第一个匹配，有了这个开关，regsub 将把所有匹配的地方全部替换。

-- 意义同 regexp 命令中。

■string命令

string 命令的语法：**string** option *arg* ?*arg*...?

string 命令具有强大的操作字符串的功能，其中的 option 选项多达 20 个。下面介绍其中常用的部分。

■1、string compare ?-nocase? ?-length int? string1 string2

把字符串 string1 和 string2 进行比较,返回值为-1、0 或 1,分别对应 string1 小于、等于或大于 string2。如果有 -length 参数，那么只比较前 int 个字符，如果 int 为负数，那么这个参数被忽略。 如果有 -nocase 参数，那么比较时不区分大小写。

■2、string equal ?-nocase? ?-length int? string1 string2

把字符串 string1 和 string2 进行比较，如果两者相同，返回值为 1，否则返回 0。其他参数与 8.5.1 同。

■3、string first string1 string2 ?startindex?

在 string2 中从头查找与 string1 匹配的字符序列，如果找到，那么就返回匹配的第一个字母所在的位置(0-based)。如果没有找到，那么返回-1。如果给出了 startindex 变量，那么将从 startindex 处开始查找。例如：

```
% string first ab defabc
3
% string first ab defabc 4
-1
```

■4、string index string charIndex

返回 string 中第 charIndex 个字符(0-based)。charIndex 可以是下面的值：

整数 n：字符串中第 n 个字符(0-based)

end ： 最后一个字符

end－整数 n：倒数第 n 个字符。string index "abcd" end-1 返回字符'c'

如果 charIndex 小于 0，或者大于字符串 string 的长度，那么返回空。

例如:

```
% string index abcdef 2
c
% string index abcdef end-2
d
```

■5、string last string1 string2 ?startindex?

参照 3.唯一的区别是从后往前查找

■6、string length string

返回字符串 string 的长度.

■7、string match ?-nocase? pattern string

如果 pattern 匹配 string,那么返回 1,否则返回 0.如果有-nocase 参数,那么就不区分大小写.

在 pattern 中可以使用通配符:

* 匹配 string 中的任意长的任意字符串,包括空字符串.

? 匹配 string 中任意单个字符

[chars] 匹配字符集合 chars 中给出的任意字符,其中可以使用 A-Z 这种形式

\x 匹配单个字符 x,使用'\'是为了让 x 可以为字符*,-,[,].

例子:

```
% string match * abcdef
1
% string match a* abcdef
1
string match a?cdef abcdef
1
% string match {a[b-f]cdef} abcdef //注意一定药用'{',否则 TCL 解释器会把 b-f 当作命令名
1 //从而导致错误
% string match {a[b-f]cdef} accdef
1
```

■8、string range string first last

返回字符串 string 中从第 first 个到第 last 个字符的子字符串(0-based)。如果 first<0，那么 first 被看作 0，如果 last 大于或等于字符串的长度，那么 last 被看作 end，如果 first 比 last 大，那么返回空。

■9、string repeat string count

返回值为：重复了 string 字符串 count 次的字符串。例如：

% string repeat "abc" 2

abcabc

■10、string replace string first last ?newstring?

返回值为：从字符串 string 中删除了第 first 到第 last 个字符(0-based)的字符串，如果给出了 newstring 变量，那么就用 newstring 替换从第 first 到第 last 个字符。如果 first<0，那么 first 被看作 0，如果 last 大于或等于字符串的长度，那么 last 被看作 end，如果 first 比 last 大或者大于字符串 string 的长度或者 last 小于 0，那么原封不动的返回 string 。

■11、string tolower string ?first? ?last?

返回值为：把字符串 string 转换成小写后的字符串，如果给出了 first 和 last 变量，就只转换 first 和 last 之间的字符。

■12、string toupper string ?first? ?last?

同 11。转换成大写。

■13、string trim string ?chars?

返回值为:从 string 字符串的首尾删除掉了字符集合 chars 中的字符后的字符串。如果没有给出 chars，那么将删除掉 spaces、tabs、newlines、carriage returns 这些字符。例如:

```
% string trim "abcde" {a d e}
bc
% string trim " def
> "
def
```

■14、string trimleft string ?chars?

同 13。不过只删除左边的字符。

■15、string trimright string ?chars?

同 13。不过只删除右边的字符。

文件访问

■文件名

TCL 提供了丰富的文件操作的命令。通过这些命令你可以对文件名进行操作(查找匹配某一模式的文件)、以顺序或随机方式读写文件、检索系统保留的文件信息（如最后访问时间)。

CL 中文件名和我们熟悉的 windows 表示文件的方法有一些区别：在表示文件的目录结构时它使用'/',而不是'\',这和 TCL 最初是在 UNIX 下实现有关。比如 C 盘 tcl 目录下的文件 sample.tcl 在 TCL 中这样表示：C:/tcl/sample.tcl。

■基本文件输入输出命令

这个名为 tgrep 的过程，可以说明 TCL 文件 I/O 的基本特点：

```
proc tgrep { pattern filename} {
set f [open $filename r]
while { [gets $f line ] } {
if {[regexp $pattern $line]} {
puts stdout $line
}
}
close $f
}
```

以上过程非常象 UNIX 的 grep 命令， 你可以用两个参数调用它，一个是模式，另一个是文件名，tgrep 将打印出文件中所有匹配该模式的行。

下面介绍上述过程中用到的几个基本的文件输入输出命令。

open *name* ?*access*?

open 命令 以 *access* 方式打开文件 *name*。返回供其他命令(gets,close 等)使用的文件标识。如果 *name* 的第一个字符是"|"，管道命令被触发，而不是打开文件。

文件的打开方式和我们熟悉的 C 语言类似，有以下方式：

r 只读方式打开。文件必须已经存在。这是默认方式。

r+ 读写方式打开，文件必须已经存在。

w 只写方式打开文件，如果文件存在则清空文件内容，否则创建一新的空文件。

w+ 读写方式打开文件，如文件存在则清空文件内容，否则创建新的空文件。

a 只写方式打开文件，文件必须存在，并把指针指向文件尾。

a+ 只读方式打开文件，并把指针指向文件尾。如文件不存在，创建新的空文件。

open 命令返回一个字符串用于表识打开的文件。当调用别的命令（如：gets,puts,close，）对打开的文件进行操作时，就可以使用这个文件标识符。TCL 有三个特定的文件标识：stdin,stdout 和 stderr ，分别对应标准输入、标准输出和错误通道，任何时候你都可以使用这三个文件标识。

gets *fileId* ?*varName*? 读 *fileId* 标识的文件的下一行，忽略换行符。如果命令中有 *varName* 就把该行赋给它，并返回该行的字符数（文件尾返回-1），如果没*有 varName 参数，*返回文件的下一行作为命令结果（如果到了文件尾，就返回空字符串）。

和 gets 类似的命令是 read，不过 read 不是以行为单位的，它有两种形式：

read ?**-nonewline**? *fileId* 读并返回 *fileId* 标识的文件中所有剩下的字节。如果没有 nonewline 开关，则在换行符处停止。

read *fileId numBytes* 在 *fileId* 标识的文件中读并返回下一个 numbytes 字节。

puts ?**-nonewline**? ?*fileId*? *string* puts 命令把 *string* 写到 *fileId* 中，如果没有 nonewline 开关的话，添加换行符。*fileId* 默认是 stdout。命令返回值为一空字符串。

puts 命令使用 C 的标准 I/O 库的缓冲区方案，这就意味着使用 puts 产生的信息不会立即出现在目标文件中。如果你想使数据立即出现在文件中，那你就调用 flush 命令：

flush *fileId* 把缓冲区内容写到 *fileId* 标识的文件中，命令返回值为空字符串。

flush 命令迫使缓冲区数据写到文件中。flush 直到数据被写完才返回。当文件关闭时缓冲区数据会自动 flush。

close ?*fileId*? 关闭标识为 *fileId* 的文件，命令返回值为一空字符串。

这里特别说明的一点是，TCL 中对串口、管道、socket 等的操作和对文件的操作类似，以上对文件的操作命令同样适用于它们。

■随机文件访问

默认文件输入输出方式是连续的：即每个 gets 或 read 命令返回的是上次 gets 或 read 访问位置后面的字节，每个 puts 命令写数据是接着上次 puts 写的位置接着写。TCL 提供了 seek,tell 和 eof 等命令使用户可以非连续访问文件。

每个打开的打开文件都有访问点，即下次读写开始的位置。文件打开时，访问点总是被设置为文件的开头或结尾，这取决于打开文件时使用的访问模式。每次读写后访问位置按访问的字节数后移相应的位数。

可以使用 seek 命令来改变文件的访问点：

seek *fileId offset* ?*origin*? 把 *fileId* 标识的文件的访问点设置为相对于 *origin* 偏移量为 *offset* 的位置。*origin* 可以是 start，current，end，默认是 start。命令的返回值是一空字符串。

例如：seek fileId 2000 改变 fieleId 标识的文件访问点，以便下次读写开始于文件的第 2000 个字节。

seek 的第三个参数说明偏移量从哪开始计算。第三个参数必为 start,current 或 end 中的一个。start 是默认值：即偏移量是相对文件开始处计算。current 是偏移量从当前访问位置计算。end 是偏移量从文件尾开始计算。

tell *fileId* 返回 fileId 标识的文件的当前访问位置。

eof *fileId* 如果到达 fileId 标识的文件的末尾返回 1，否则返回 0。

■当前工作目录

TCL 提供两个命令来管理当前工作目录：pwd 和 Cd。

pwd 和 UNIX 下的 pwd 命令完全一样， 没有参数，返回当前目录的完整路径。

cd 命令也和 UNIX 命令也一样，使用一个参数，可以把工作目录改变为参数提供的目录。如果 cd 没使用参数，UNIX 下，会把工作目录变为启动 TCL 脚本的用户的工作目录，WINDOWS 下会把工作目录变为 windows 操作系统的安装目录所在的盘的根目录(如：C:/)。值得注意的是，提供给 cd 的参数中路径中的应该用'/'而不是'\'。如 cd C:/TCL/lib。这是 UNIX 的风格。

■文件操作和获取文件信息

TCL 提供了两个命令进行文件名操作：glob 和 file，用来操作文件或获取文件信息。

glob 命令采用一种或多种模式作为参数，并返回匹配这个（些）模式的所有文件的列表，其语法为：

glob ?switches? pattern ?pattern ...?

其中 switches 可以取下面的值：

-nocomplain ：允许返回一个空串，没有-nocomplain 时，如果结果是空的，就返回错误。

-- ：表示 switches 结束，即后面以'-'开头的参数将不作为 switches。

glob 命令的模式采用 string match 命令(见 8.5.7 节)的匹配规则。例如：

```
%glob *.c *.h
main.c hash.c hash.h
```

返回当前目录中所有.c 或.h 的文件名。 glob 还允许模式中包含' 括在花括号中间以逗号分开的多种选择'，例如 ：

```
%glob {{src,backup}/*.[ch]}
src/main.c src/hash.c src/hash.h backup/hash.c
```

下面的命令和上面的命令等价：

```
glob {src/*.[ch]} {backup/*.[ch]}
```

注意：这些例子中模式周围的花括号是必须的，可以防止命令置换。在调用 **glob** 命令对应的 C 过程前这些括号会被 TCL 解释器去掉。

如果 **glob** 的模式以一斜线结束，那将只匹配目录名。例如：

```
glob */
```

只返回当前目录的所有子目录。

如果 **glob** 返回的文件名列表为空，通常会产生一个错误。但是 **glob** 的在样式参数之前的第一个参数是"-nocomplain"的话，这时即使结果为空，**glob** 也不会产生错误。

对文件名操作的第二个命令是 **file**。**file** 是有许多选项的常用命令，可以用来进行文件操作也可以检索文件信息。这节讨论与名字相关的选项，下一节描述其他选项。使用 file 命令时，我们会发现其中有很明显的 UNIX 痕迹。

file atime name 返回一个十进制的字符串，表示文件 name 最后被访问的时间。时间是以秒为单位从 1970 年 1 月 1 日 12：00AM 开始计算。如果文件 name 不存在或查询不到访问时间就返回错误。例：

```
% file atime license.txt
975945600
```

file copy ?**-force**? ?--? *source target*

file copy ?**-force**? ?--? *source* ?*source* ...? *targetDir*

这个命令把 source 中指明的文件或目录递归的拷贝到目的地址 targetDir，只有当存在-force 选项时，已经存在的文件才会被覆盖。试图覆盖一个非空的目录或以一个文件覆盖一个目录或以一个目录覆盖一个文件都会导致错误。--的含义和前面所说的一样。

file delete ?**-force**? ?--? *pathname* ?*pathname* ... ? 这个命令删除 pathname 指定的文件或目录，当指定了**-force** 时，非空的目录也会被删除。即使没有指定**-force**，只读文件也会被删除。删除一个不存在的文件不会引发错误。

file dirname *name* 返回 *name* 中最后一个"/"前的所有字符；如果 *name* 不包含"/"，返回"."；如果 *name* 中最后一个"/"是第 name 的第一个字符，返回"/"。

file executable *name* 如果 name 对当前用户是可以执行的，就返回 1，否则返回 0。

file exists *name* 如果 *name* 存在于当前用户拥有搜索权限的目录下返回 1，否则返回 0。

file extension *name* 返回 *name* 中最后的"."以后（包括这个小数点）的所有字符。如果 *name* 中没有"."或最后斜线后没有"."返回空字符。

file isdirectory *name* 如果 *name* 是目录返回 1，否则返回 0。

file isfile *name* 如果 *name* 是文件返回 1，否则返回 0。

file lstat name *arrayName* 除了利用 lstat 内核调用代理 stat 内核调用之外，和 **file stat** 命令一样，这意味着如果 name 是一个符号连接，那么这个命令返回的是这个符号连接的信息而不是这个符号连接指向的文件的信息。对于不支持符号连接的操作系统，这个命令和和 **file stat** 命令一样。

file mkdir dir ?dir ...? 这个命令和 UNIX 的 mkdir 命令类似，创建 dir 中指明的目录。如果 dir 已经存在，这个命令不作任何事情，也不返回错误。不过如果试图用一个目录覆盖已经存在的一个文件会导致错误。这个命令顺序处理各个参数，如果发生错误的话，马上退出。

file mtime *name* 返回十进制的字符串，表示文件 *name* 最后被修改的时间。时间是以秒为单位从 1970 年 1 月 1 日 12：00AM 开始计算。

file owned *name* 如果 *name* 被当前用户拥有，返回 1，否则返回 0。

file readable *name* 如果当前用户可*对 name* 进行读操作，返回 1，否则返回 0。

file readlink *name* 返回 *name* 代表的符号连接所指向的文件。如果 name 不是符号连接或者找不到符号连接，返回错误。在不支持符号连接的操作系统(如 windows)中选项 **readlink** 没有定义。

file rename ? **-force**? ?--? *source target*

file rename ?**-force**? ?--? *source* ?*source* ...? *targetDir*

这个命令同时具有重命名和移动文件(夹)的功能。把 source 指定的文件或目录改名或移动到 *targetDir* 下。 只有当存在-force 选项时，已经存在的文件才会被覆盖。 试图覆盖一个非空的目录或以一个文件覆盖一个目录或以一个目录覆盖一个文件都会导致错误。

file rootname *name* 返回 *name* 中最后"."以前（不包括这个小数点）的所有字符。如果 *name* 中没有"."返回 *Name*。

file size *name* 返回十进制字符串，以字节表示 *name* 的大小。如果文件不存在或得不到 name 的大小，返回错误。

file stat *name arrayName* 调用 stat 内核来访问 name，并设置 arrayName 参数来保存 stat 的返回信息。 arrayName 被当作一个数组，它将有以下元素：atime、ctime、dev、gid、ino、mode、mtime、nlink、size、type 和 uid。除了 type 以外，其他元素都是十进制的字符串，type 元素和 file type 命令的返回值一样。其它各个元素的含义如下：

atime 最后访问时间.

ctime 状态最后改变时间.

dev 包含文件的设备标识.

gid 文件组标识.

ino 设备中文件的序列号.

mode 文件的 mode 比特位.

mtime 最后修改时间.

nlink 到文件的连接的数目.

size 按字节表示的文件尺寸.

uid 文件所有者的标识.

这里的 atime、mtime、size 元素与前面讨论的 file 的选项有相同的值。要了解其他元素更多的信息，就查阅 stat 系统调用的文件；每个元都直接从相应 stat 返回的结构域中得到。 文件操作的 stat 选项提供了简单的方法使一次能获得一个文件的多条信息。这要比分多次调用 file 来获得相同的信息量要显著的快。

file tail *name* 返回 *name* 中最后一个斜线后的所有字符，如果没有斜线返回 *name*。

file type *name* 返回文件类型的字符串，返回值可能是下列中的一个： file、directory、characterspecial、blockSpecial、fifo、link 或 socket。

file writable *name*

如果当前用户对 *name* 可进行写操作，返回 1，否则返回 0。

错误和异常

错误

错误和异常处理机制是创建大而健壮的应用程序的必备条件之一，很多计算机语言都提供了错误和异常处理机制，TCL 也不例外。

错误(Errors)可以看作是异常(Exceptions)的特例。TCL 中，异常是导致脚本被终止的事件，除了错误还包括 break、continue 和 return 等命令。TCL 允许程序俘获异常，这样仅有程序的一部分工作被撤销。程序脚本俘获异常事件以后，可以忽略它，或者从异常中恢复。如果脚本无法恢复此异常，可以把它重新发布出去。下面是与异常有关的 TCL 命令：

catch *command* ?*varName*? 这个命令把 command 作为 TCL 脚本求值，返回一个整型值表明 command 结束的状态。如果提供 varName 参数，TCL 将生成变量 varName，用于保存 command 产生的错误消息。

error *message* ?*info*? ?*code*? 这个命令产生一个错误，并把 message 作为错误信息。如 崩¯¯nfo 参数，则被用于初始化全局变量 errorInfo。如果提供 code 参数，将被存储到全局变量 errorCode 中。

return -code *code* ?**-errorinfo** *info*? ?**-errorcode** *errorCode*? ?*string*? 这个命令使特定过程返回一个异常。code 指明异常的类型，必须是 ok,error,return,break,continue 或者是一个整数。-errorinfo 选项用于指定全局变量 errorInfo 的初始值，-errorcode 用于指定全局变量 errorCode 的初始值。 string 给出 return 的返回值或者是相关的错误信息，其默认值为空。

当发生一个 TCL 错误时，当前命令被终止。如果这个命令是一大段脚本的一部分，那么整个脚本被终止。如果一个 TCL 过程在运行中发生错误，那么过程被终止，同时调用它的过程，以至整个调用栈上的活动过程都被终止，并返回一个错误标识和一段错误描述信息。

举个例子，考虑下面脚本，它希望计算出列表元素的总和：

```
set list {44 16 123 98 57}
set sum 0
foreach el $list {
set sum [expr $sum+$element]
}
=> can't read "element": no such variable
```

这个脚本是错误的，因为没有 element 这个变量。TCL 分析 expr 命令时，会试图用 element 变量的值进行替换，但是找不到名字为 element 的变量，所以会报告一个错误。由于 foreach 命令利用 TCL 解释器解释循环体，所以错误标识被返回给 foreach。foreach 收到这个错误，会终止循环的执行，然后把同样的错误标识作为它自己的返回值返回给调用者。按这样的顺序，

将致使整个脚本终止。错误信息 can't read "element": no such variable 会被一路返回，并且很可能被显示给用户。

很多情况下，错误信息提供了足够的信息为你指出哪里以及为什么发生了错误。然而，如果错误发生在一组深层嵌套的过程调用中，仅仅给出错误信息还不能为指出哪里发生了错误提供足够信息。为了帮助我们指出错误的位置，当 TCL 撤销程序中运行的命令时，创建了一个跟踪栈，并且把这个跟踪栈存储到全局变量 errorInfo 中。跟踪栈中描述了每一层嵌套调用。例如发生上面的那个错误后，errorInfo 有如下的值：

```
can't read "element": no such variable
while executing
"expr $sum+$element"
("foreach" body line 2)
invoked from within
"foreach el $list {
set sum [expr $sum+$element]
}"
```

在全局变量 errorCode 中，TCL 还提供了一点额外的信息。errorCode 变量是包含了一个或若干元素的列表。第一个元素标示了错误类别，其他元素提供更详细的相关的信息。不过，errorCode 变量是 TCL 中相对较新的变量，只有一部分处理文件访问和子过程的命令会设置这个变量。如果一个命令产生的错误没有设置 errorCode 变量，TCL 会填一个 NONE 值。

当用户希望得到某一个错误的详细的信息，除了 命令返回值中的错误信息外，可以查看全局变量 errorInfo 和 errorCode 的值。

■从TCL脚本中产生错误

大多数 TCL 错误是由实现 TCL 解释器的 C 代码和内建命令的 C 代码产生的。然而，通过执行 TCL 命令 error 产生错误也是可以的，见下面的例子：

```
if {($x<0)||($x>100)} {
error "x is out of range ($x)"
}
```

error 命令产生了一个错误，并把它的参数作为错误消息。

作为一种编程的风格，你应该只在迫不得已终止程序时下才使用 error 命令。如果你认为错误很容易被恢复而不必终止整个脚本，那么使用通常的 return 机制声明成功或失败会更好（例如，命令成功返回某个值，失败返回另一个值，或者设置变量来表明成功或失败）。尽管从错误中恢复是可能的，但恢复机制比通常的 return 返回值机制要复杂。因此，最好是在你不想恢复的情况下才使用 error 命令。

■使用catch捕获错误

错误通常导致所有活动的 TCL 命令被终止，但是有些情况下，在错误发生后继续执行脚本是有用的。例如，你用 unset 取消变量 x 的定义，但执行 unset 时，x 可能不存在。如果你用 unset 取消不存在的变量，会产生一个错误：

```
% unset x
can't unset "x": no such variable
```

此时，你可以用 catch 命令忽略这个错误：

```
% catch {unset x}
1
```

catch 的参数是 TCL 脚本。如果脚本正常完成，catch 返回 0。如果脚本中发生错误，catch 会俘获错误（这样保证 catch 本身不被终止掉）然后返回 1 表示发生了错误。上面的例子忽略 unset 的任何错误，这样如果 x 存在则被取消，即使 x 以前不存在也对脚本没有任何影响。

catch 命令可以有第二个参数。如果提供这个参数，它应该是一个变量名，catch 把脚本的返回值或者是出错信息存入这个变量。

```
%catch {unset x} msg
1
%set msg
can't unset "x": no such variable
```

在这种情况下，unset 命令产生错误，所以 msg 被设置成包含了出错信息。如果变量 x 存在，那么 unset 会成功返回，这样 catch 的返回值为 0，msg 存放 unset 命令的返回值，这里是个空串。如果在命令正常返回时，你想访问脚本的返回值，这种形式很有用；如果你想在出错时利用错误信息做些什么，如产生 log 文件，这种形式也很有用。

■其他异常

错误不是导致运行中程序被终止的唯一形式。错误仅是被称为异常的一组事件的一个特例。除了 error，TCL 中还有三种形式的异常，他们是由 break、continue 和 return 命令产生的。所有的异常以相同的方式导致正在执行的活动脚本被终止，但有两点不同：首先，errorInfo 和 errorCode 只在错误异常中被设置；其次，除了错误之外的异常几乎总是被一个命令俘获，不会波及其他，而错误通常撤销整个程序中所有工作。例如，break 和 continue 通常是被引入到一个如 foreach 的循环命令中；foreach 将俘获 break 和 continue 异常，然后终止循环或者跳到下一次重复。类似地，return 通常只被包含在过程或者被 source 引入的文件中。过程实现和 source 命令将俘获 return 异常。

所有的异常伴随一个字符串值。在错误情况，这个串是错误信息，在 return 方式，串是过程或脚本的返回值，在 break 和 continue 方式，串是空的。

catch 命令其实可以俘获所有的异常，不仅是错误。catch 命令的返回值表明是那种情况的异常，catch 命令的第二个参数用来保存与异常相关的串。例纾?

```
%catch {return "all done"} string
2
%set string
all done
```

下表是对命令： **catch** *command* ?*varName*? 的说明。

catch 返回值	**描述**	**俘获者**
0	正常返回，*varName* 给出返回值	无异常
1	错误。 *varName* 给出错误信息	catch
2	执行了 return 命令，*varName* 包含过程返回值或者返回给 source 的结果	catch,source,过程调用
3	执行了 break 命令，*varName* 为空	catch,for,foreach,while,过程
4	执行了 continue 命令，*varName* 为空	catch,for,foreach,while,过程
其他值	用户或应用自定义	catch

与 catch 命令提供俘获所有异常的机制相对应，return 可以提供产生所有类型异常。

这里有一个 do 命令的实现，使用了 catch 和 return 来正确处理异常：

```
proc do {varName first last body} {
global errorInfo errorCode
upvar $varName v
for {set v $first} {$v <= $last} {incr v} {
switch [catch {uplevel $body} string] {
1 {return -code error -errorInfo $errorInfo \
```

```
-errorcode $errorCode $string}
2 {return -code return $string}
3 return
}
}
}
```

这个新的实现在 catch 命令中求循环体的值，然后检查循环体是如何结束的。如果没有发生异常(0)，或者异常是 continue(4)，那么 do 继续下一个循环。如果发生 error(1)或者 return(2)，那么 do 使用 return 把异常传递到调用者。如果发生了 break(3)异常，那么 do 正常返回到调用者，循环结束。

当 do 反射一个 error 到上层时，它使用了 return 的-errorInfo 选项，保证错误发生后能够得到一个正确的调用跟踪栈。-errorCode 选项用于类似的目的以传递由 catch 命令得到的初始 errorCode，作为 do 命令的 errorCode 返回。如果没有-errorCode 选项，errorCode 变量总是得到 NONE 值。

■深入 TCL

■查询数组中的元素

利用 **array** 命令可以查询一个数组变量中已经定义了的元素的信息。**array** 命令的形式如下：

array *option arrayName ?arg arg ...?*

由于 option 的不同，**array** 命令有多种形式。

如果我们打算开始对一个数组的元素进行查询，我们可以先启动一个搜索(search)，这可以由下面的命令做到：

array startserach *arrayName* 这个命令初始化一个对 name 数组的所有元素的搜索(search)，返回一个搜索标识(search identifier)，这个搜索标识将被用于命令 **array nextelement**、**array anymore** 和 **array donesearch**。

array nextelement *arrayName searchId* 这个命令返回 arrayName 的下一个元素，如果 arrayName 的所有元素在这一次搜索中都已经返回，那么返回一个空字符串。 搜索标识 *searchId* 必须是 **array startserach** 的返回值。注意：如果对 *arrayName* 的元素进行了添加或删除，那么所有的搜索都会自动结束，就象调用了命令 **array donesearch** 一样，这样会导致 **array nextelement** 操作失败。

array anymore *arrayName searchId* 如果在一个搜索中还有元素就返回 1，否则返回 0。*searchId* 同上。这个命令对具有名字为空的元素的数组尤其有用，因为这时从 **array nextelement** 中不能确定一个搜索是否完成。

array donesearch *arrayName searchId* 这个命令中止一个搜索，并销毁和这个搜索有关的所有状态。*searchId* 同上。命令返回值为一个空字符串。当一个搜索完成时一定要注意调用这个命令。

array 命令的其他 option 如下：

array exists *arrayName* 如果存在一个名为 *arrayName* 的数组，返回 1，否则返回 0。

array get *arrayName ?pattern?* 这个命令的返回值是一个元素个数为偶数的的 list。我们可以从前到后把相邻的两个元素分成一个个数据对，那么，每个数据对的第一个元素是 arrayName 中元素的名字，数据对的第二个元素是该数据元素的值。数据对的顺序没有规律。如果没有 pattern 参数，那么数组的所有元素都包含在结果中，如果有 pattern 参数，那么只有名字和 pattern 匹配(用 **string match** 的匹配规则)的元素包含在结果中。如果 *arrayName* 不是一个数组变量的名字或者数组中没有元素，那么返回一个空 list。例：

```
% set b(first) 1
1
% set b(second) 2
2
% array get b
second 2 first 1
```

array set *arrayName list* 设置数组 *arrayName* 的元素的值。 *list* 的 形式和 **array get** 的返回值的 list 形式一样。如果 *arrayName* 不存在，那么生成 *arrayName*。例：

```
% array set a {first 1 second 2}
% puts $a(first)
1
% array get a
second 2 first 1
```

array names *arrayName* ?*pattern*? 这个命令返回数组 *arrayName* 中和模式 pattern 匹配的元素的名字组成的一个 list。如果没有 pattern 参数，那么返回所有元素。如果数组中没有匹配的元素或者 *arrayName* 不是一个数组的名字，返回一个空字符串。

array size *arrayName* 返回代表数组元素个数的一个十进制的字符串，如果 arrayName 不是一个数组的名字，那么返回 0。

下面这个例子通过使用 **array names** 和 **foreach** 命令，枚举了数组所有的元素：

```
foreach i [array names a] {
puts "a($i)=$a($i)"
}
```

当然，我们也可以利用 **startsearch**、**anymore**、**nextelement**、和 **donesearch** 选项来遍历一个数组．这种方法比上面所给出的 foreach 方法的效率更高，不过要麻烦得多，因此不常用。

■info命令

info 命令提供了查看 TCL 解释器信息的手段，它有超过一打的选项，详细说明请参考下面几节。

■变量信息

info 命令的几个选项提供了查看变量信息的手段。

info exists *varName* 如果名为 *varName* 的变量在当前上下文(作为全局或局部变量)存在，返回 1，否则返回 0。

info globals ?*pattern*? 如果没有 pattern 参数，那么返回包含所有全局变量名字的一个 list。如果有 pattern 参数，就只返回那些和 pattern 匹配的全局变量(匹配的方式和 **string match** 相同)。

info locals ?*pattern*? 如果没有 pattern 参数，那么返回包含所有局部变量(包括当前过程的参数)名字的一个 list，global 和 upvar 命令定义的变量将不返回。如果有 pattern 参数，就只返回那些和 pattern 匹配的局部变量(匹配的方式和 **string match** 相同)。

info vars ?*pattern*? 如果没有 pattern 参数，那么返回包括局部变量和可见的全局变量的名字的一个 list。如果有 pattern 参数，就只返回和模式 pattern 匹配的局部变量和可见全局变量。模式中可以用 namespace 来限定范围，如:foo::option*，就只返回 namespace 中和 option*匹配的局部和全局变量。 (注：tcl80 以后引入了 namespace 概念，不过我们一般编写较小的 TCL 程序，可以对 namespace 不予理睬，用兴趣的话可以查找相关资料。)

下面针对上述命令举例，假设存在全局变量 global1 和 global2，并且有下列的过程存在:

```
proc test {arg1 arg2} {
global global1
set local1 1
set local2 2
...
}
```

然后在过程中执行下列命令:

```
% info vars
global1 arg1 arg2 local2 local1 //global2 不可见
% info globals
global2 global1
% info locals
arg1 arg2 local2 local1
```

```
% info vars *al*
global1 local2 local1
```

■过程信息

info 命令的另外的一些选项可以查看过程信息。

info procs ?*pattern*? 如果没有 pattern 参数，命令返回当前 namespace 中定义的所有过程的名字。如果有 pattern 参数，就只返回那些和 pattern 匹配的过程的名字(匹配的方式和 **string match** 相同)。

info body *procname* 返回过程 *procname* 的过程体。 *procname* 必须是一个 TCL 过程。

info args *procname* 返回包含过程 *procname* 的所有参数的名字的一个 list。 *procname* 必须是一个 TCL 过程。

info default *procname arg varname procname* 必须是一个 TCL 过程， *arg* 必须是这个过程的一个变量。如果 *arg* 没有缺省值，命令返回 0；否则返回 1，并且把 *arg* 的缺省值赋给变量 *varname*。

info level ?number? 如果没有 number 参数，这个命令返回当前过程在调用栈的位置。如果有 number 参数，那么返回的是包含 在调用栈的位置为 number 的过程的过程名及其参数的一个 list。

下面针对上述命令举例:

```
proc maybeprint {a b {c 24}} {
if {$a<$b} {
puts stdout "c is $c"
}
}
% info body maybeprint
if {$a<$b} {
puts stdout "c is $c"
}
% info args maybeprint
a b c
% info default maybeprint a x
0
% info default maybeprint a c
1
%set x
24
```

下面的过程打印出了当前的调用栈，并显示了每一个活动过程名字和参数：

```
proc printStack{}{
set level [info level]
for {set i 1} {$i<$level} {incr i} {
puts "Level $i:[info level $i]"
}
}
```

■命令信息

info 命令的另外选项可以查看命令信息。

info commands ?*pattern*? 如果没有参数 *pattern*，这个命令返回包含当前 namspace 中所有固有、扩展命令以及以 **proc** 命令定义的过程在内的所有命令的名字的一个 list。*pattern* 参数的含义和 **info procs** 一样。

info cmdcount 返回了一个十进制字符串，表明多少个命令曾在解释器中执行过。

info complete *command* 如果命令是 *command* 完整的，那么返回 1，否则返回 0。这里判断命令是否完整仅判断引号，括号和花括号是否配套。

info script 如果当前有脚本文件正在 Tcl 解释器中执行，则返回最内层处于激活状态的脚本文件名；否则将返回一个空的字符串。

■TCL 的版本和库

info tclversion 返回为 Tcl 解释器返回的版本号，形式为 major.minor，例如 8.3。

info library 返回 Tcl 库目录的完全路径。这个目录用于保存 Tcl 所使用的标准脚本，TCL 在初始化时会执行这个目录下的脚本。

■命令的执行时间

TCL 提供 time 命令来衡量 TCL 脚本的性能：

time script ?count? 这个命令重复执行 script 脚本 count 次。再把花费的总时间的用 count 除，返回一次的平均执行时间，单位为微秒。如果没有 count 参数，就取执行一次的时间。

■跟踪变量

TCL 提供了 **trace** 命令来跟踪一个或多个变量。如果已经建立对一个变量的跟踪，则不论什么时候对该变量进行了读、写、或删除操作，就会激活一个对应的 Tcl 命令，跟踪可以有很多的用途：

1.监视变量的用法（例如打印每一个读或写的操作）。

2.把变量的变化传递给系统的其他部分（例如一个 TK 程序中，在一个小图标上始终显示某个变量的当前值）。

3.限制对变量的某些操作(例如对任何试图用非十进制数的参数来改变变量的值的行为产生一个错误。）或重载某些操作（例如每次删除某个变量时，又重新创建它）。

trace 命令的语法为：

trace option ?arg arg ...?

其中 option 有以下几种形式：

trace variable *name ops command* 这个命令设置对变量 name 的一个跟踪：每次当对变量 *name* 作 ops 操作时，就会执行 *command* 命令。name 可以是一个简单变量，也可以是一个数组的元素或者整个数组。

ops 可以是以下几种操作的一个或几个的组合：

r 当变量被读时激活 command 命令。

w 当变量被写时激活 command 命令。

u 当变量被删除时激活 command 命令。通过用 unset 命令可以显式的删除一个变量，一个过程调用结束则会隐式的删除所有局部变量。当删除解释器时也会删除变量，不过这时跟踪已经不起作用了。

当对一个变量的跟踪被触发时，TCL 解释器会自动把三个参数添加到命令 command 的参数列表中。这样 command 实际上变成了

command name1 name2 op

其中 op 指明对变量作的什么操作。name1 和 name2 用于指明被操作的变量： 如果变量是一个标量，那么 name1 给出了变量的名字，而 name2 是一个空字符串；如果变量是一个数组的一个元素，那么 name1 给出数组的名字，而 name2 给出元素的名字；如果变量是整个数组，那么 name1 给出数组的名字而 name2 是一个空字符串。为了让你很好的理解上面的叙述，下面举一个例子：

```
trace variable color w pvar
trace variable a(length) w pvar
proc pvar {name element op} {
if {$element !=""} {
set name ${name}($element)
}
upvar $name x
puts "Variable $name set to $x"
}
```

上面的例子中，对标量变量 color 和数组元素 a(length)的写操作都会激活跟踪操作 pvar。我们看到过程 pvar 有三个参数，这三个参数 TCL 解释器会在跟踪操作被触发时自动传递给 pvar。比如如果我们对 color 的值作了改变，那么激活的就是 pvar color "" w。我们敲入：

```
% set color green
Variable color set to green
green
```

command 将在和触发跟踪操作的代码同样的上下文中执行：如果对被跟踪变量的访问是在一个过程中，那么 command 就可以访问这个过程的局部变量。比如：

```
proc Hello { } {
set a 2
trace variable b w { puts $a ;list }
set b 3
}
% Hello
2
3
```

对于被跟踪变量的读写操作，command 是在变量被读之后，而返回变量的值之前被执行的。因此，我们可以在 command 对变量的值进行改变，把新值作为读写的返回值。而且因为在执行 command 时，跟踪机制会临时失效，所以在 command 中对变量进行读写不会导致 command 被递归激活。例如：

```
% trace variable b r tmp
% proc tmp {var1 var2 var3 } {
upvar $var1 t
incr t 1
}
% set b 2
2
% puts $b
3
```

```
% puts $b
4
```

如果对读写操作的跟踪失败，即 command 失败，那么被跟踪的读写操作也会失败，并且返回和 command 同样的失败信息。利用这个机制可以实现只读变量。下面这个例子实现了一个值只能为正整数的变量：

```
trace variable size w forceInt
proc forceInt {name element op} {
upvar $name x
if ![regexp {^[0-9]*$} $x] {
error "value must b a postive integer"
}
}
```

如果一个变量有多个跟踪信息，那么各个跟踪被触发的先后原则是：最近添加的跟踪最先被触发，如果有一个跟踪发生错误，后面的跟踪就不会被触发。

trace vdelete *name ops command* 删除对变量 name 的 ops 操作的跟踪。返回值为一个空字符串。

trace vinfo *name* 这个命令返回对变量的跟踪信息。返回值是一个 list，list 的每个元素是一个子串，每个子串包括两个元素：跟踪的操作和与操作关联的命令。如果变量 name 不存在或没有跟踪信息，返回一个空字符串。

■命令的重命名和删除

rename 命令可以用来重命名或删除一个命令。

rename *oldName newName* 把命令 oldName 改名为 newName，如果 newName 为空，那么就从解释器中删除命令 oldName。

下面的脚本删除了文件 I/O 命令：

```
foreach cmd {open close read gets puts} {
rename $cmd {}
}
```

任何一个 Tcl 命令都可以被重命名或者删除，包括内建命令以及应用中定义的过程和命令。重命名一个内建命令可能会很有用，例如，exit 命令在 Tcl 中被定义为立即退出过程。如果某个应用希望在退出前获得清除它内部状态的机会，那么可以这样作：

```
rename exit exit.old
proc exit status {
application-specific cleanup
```

```
...
exit.old $status
}
```

在这个例子中，exit 命令被重命名为 exit.old，并且定义了新的 exit 命令，这个新命令作了应用必需的清除工作而后调用了改了名字的 exit 命令来结束进程。这样在已存在的描述程序中调用 exit 时就会有机会做清理应用状态的工作。

■unknown 命令

unknown 命令的语法为：

unknown *cmdName* ?*arg arg* ...? 当一个脚本试图执行一个不存在的命令时，TCL 解释器会激活 **unknown** 命令，并把那个不存在的命令的名字和参数传递给 **unknown** 命令。**unknown** 命令不是 TCL 的核心的一部分，它是由 TCL 脚本实现的，可以在 TCL 安装目录的 lib 子目录下的 init.tcl 文件中找到其定义。

unknown 命令具有以下功能：

1。如果命令是一个在 TCL 的某个库文件(这里的库文件指的是 TCL 目录的 lib 子目录下的 TCL 脚本文件)中定义的过程，则加载该库并重新执行命令，这叫做"auto-loading"（自动加载），关于它将在下一节描述。

2。如果存在一个程序的名字与未知命令一致，则调用 exec 命令来调用该程序，这项特性叫做"auto-exec"（自动执行）。例如你输入"dir"作为一个命令，unknown 会执行"exec dir"来列出当前目录的内容，如果这里的命令没有特别指明需要输入输出重定向，则自动执行功能会使用当前 Tcl 应用所拥有的标准输入输出流，以及标准错误流，这不同于直接调用 exec 命令，但是提供了在 Tcl 应用中直接执行其他应用程序的方法。

3。如果命令是一组特殊字符，将会产生一个新的调用，这个调用的内容是历史上已经执行过的命令。例如，如果命令时"!!"则上一条刚执行过的命令会再执行一遍。下一章将详细讲述该功能。

4。若命令是已知命令的唯一缩写，则调用对应的全名称的正确命令。在 TCL 中允许你使用命令名的缩写，只要缩写唯一即可。

如果你不喜欢 unknown 的缺省的行为，你也可以自己写一个新版本的 unknown 或者对库中已有 unknown 的命令进行扩展以增加某项功能。如果你不想对未知命令做任何处理，也可以删除 unknown，这样当调用到未知命令的时候就会产生错误。

■自动加载

在 unknown 过程中一项非常有用的功能就是自动加载，自动加载功能允许你编写一组 Tcl 过程放到一个脚本文件中，然后把该文件放到库目录之下，当程序调用这些过程的时候，第一次调用时由于命令还不存在就会进入 unknown 命令，而 unknown 则会找到在哪个库文件中包含了

这个过程的定义，接着会加载它，再去重新执行命令，而到下次使用刚才调用过的命令的时候，由于它已经存在了，从而会正常的执行命令，自动加载机制也就不会被再次启动。

自动加载提供了两个好处，首先，你可以把有用的过程建立为过程库，而你无需精确知道过程的定义到底在哪个源文件中，自动加载机制会自动替你寻找，第二个好处在于自动加载是非常有效率的，如果没有自动加载机制你将不得不在 TCL 应用的开头使用 source 命令来加载所有可能用到的库文件，而应用自动加载机制，应用启动时无需加载任何库文件，而且有些用不到的库文件永远都不会被加载，既缩短了启动时间又节省了内存。

使用自动加载只需简单的按下面三步来做：

第一，在一个目录下创建一组脚本文件作为库，一般这些文件都以".tcl"结尾。每个文件可以包含任意数量的过程定义。建议尽量减少各脚本文件之间的关联，让相互关联的过程位于同一个文件中。为了能够让自动加载功能正确运行，proc 命令定义一定要顶到最左边，并且与函数名用空格分开，过程名保持与 proc 在同一行上。

第二步，为自动加载建立索引。启动 Tcl 应用比如 tclsh，调用命令 **auto_mkindex** *dir pattern* , 第一个参数是目录名，第二个参数是一个模式。auto_**mkindex** 在目录 dir 中扫描文件名和模式 pattern 匹配的文件，并建立索引以指出哪些过程定义在哪些文件中，并把索引保存到目录 dir 下一个叫 tclindex 的文件中。如果修改了文件或者增减过程，需要重新生成索引。

第三步是在应用中设置变量 auto_path，把存放了希望使用到的库所在的目录赋给它。auto_path 变量包含了一个目录的列表，当自动加载被启动的时候，会搜索 auto_path 中所指的目录，检查各目录下的 tclindex 文件来确认过程被定义在哪个文件中。如果一个函数被定义在几个库中，则自动加载使用在 auto_path 中靠前的那个库。

例如，若一个应用使用目录/usr/local/tcl/lib/shapes 下的库，则在启动描述中应增加：

```
set auto_path [linsert $auto_path 0 /usr/local/tcl/lib/shapes]
```

这将把/usr/local/tcl/lib/shapes 作为起始搜索库的路径，同时保持所有的 Tcl/Tk 库不变，但是在/usr/local/tcl/lib/shapes 中定义的过程具有更高的优先级，一旦一个含有索引的目录加到了 auto_path 中，里面所有的过程都可以通过自动加载使用了。