TCL培训教程

深圳市华为技术有限公司

目　录

TCL培训教程

TCL培训教程(全)

关键词： TCL

摘　要： 本文是TCL教材的第三稿，前两稿分别是《TCL的使用》和《TCL培训教程》。这一稿加入了不少内容，是北研TCL兴趣小组共同努力的结果。本文详细介绍了TCL的各个方面，特别对利用C\C++语言扩展TCL命令作了详细论述。本文附有大量实例。

缩略语清单： TCL　　Tool　Command　Language　　一种脚本语言

参考资料清单： *请在表格中罗列本文档所引用的有关参考文献名称、作者、标题、编号、发布日期和出版单位等基本信息。*

参考资料清单					
名称	作者	编号	发布日期	查阅地点或渠道	出版单位*（若不为本公司发布的文献，请填写此列）*
Tcl and Tk ToolKit	John K.Ousterhout	981－235－951－6	1999	自己的图书	Addison Wesley Publishing Commpany
TCL的使用	陈旭盛			自写文档	
TCL培训教程	陈旭盛			自写文档	

1 引言

TCL(Tool Command Language)是一种解释执行的脚本语言(Scripting Language)。它提供了通用的编程能力：支持变量、过程和控制结构；同时TCL还拥有一个功能强大的固有的核心命令集。

由于TCL的解释器是用一个C\C++语言的过程库实现的，因此在某种意义上我们又可以把TCL看作一个C库，这个库中有丰富的用于扩展TCL命令的C\C++过程和函数，可以很容易就在C\C++应用程序中嵌入TCL，而且每个应用程序都可以根据自己的需要对TCL语言进行扩展。我们可以针对某一特定应用领域对TCL语言的核心命令集进行扩展，加入适合于自己的应用领域的扩展命令，如果需要，甚至可以加入新的控制结构，TCL解释器将把扩展命令和扩展控制结构与固有命令和固有控制结构同等看待。扩展后的TCL语言将可以继承TCL 核心部分的所有功能，包括核心命令、控制结构、数据类型、对过程的支持等。根据需要，我们甚至可以屏蔽掉TCL的某些固有命令和固有控制结构。通过对

TCL的扩展、继承或屏蔽，用户用不着象平时定义一种计算机语言那样对词法、语法、语义、语用等各方面加以定义，就可以方便的为自己的应用领域提供一种功能完备的脚本语言。

TCL良好的可扩展性使得它能很好地适应产品测试的需要，测试任务常常会由于设计和需求的改变而迅速改变，往往让测试人员疲于应付。利用TCL的可扩展性，测试人员就可以迅速继承多种新技术，并针对产品新特点迅速推出扩展TCL命令集，以用于产品的测试中，可以较容易跟上设计需求的变化。

另外，因为TCL是一种比C\C++ 语言有着更高抽象层次的语言，使用TCL可以在一种更高的层次上编写程序，它屏蔽掉了编写C\C++程序时必须涉及到的一些较为烦琐的细节，可以大大地提高开发测试例的速度。而且， 使用TCL语言写的测试例脚本，即使作了修改，也用不着重新编译就可以调用TCL解释器直接执行。可以省却不少时间。

TCL 目前已成为自动测试中事实上的标准。

2 语法

简单的讲，TCL语言的语法实际上是一些TCL解释器怎样对TCL命令进行分析的规则的集合。

2.1 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。

2.2 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。

2.2.1 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。

2.2.2 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。

2.2.3 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
```

2.2.4 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]
```

2.3 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。

3 变量

TCL支持两种类型的变量：简单变量和数组。

3.1 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。

3.2 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。

3.3 相关命令

3.3.1 set

这个命令在3.1已有详细介绍。

3.3.2 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
```

3.3.3 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
```

4 表达式

TCL中的表达式类似于ANSI C的表达式。表达式由操作数和操作符构成，下面分别介绍。

4.1 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.
```

4.2 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

1.1 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
```

需要注意的一点是，数学函数并不是命令，只在表达式中出现才有意义。

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。

2 List

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：

2.1 list命令

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}

2.2 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。

2.3 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

2.4 llength命令

语法：llength list

返回list的元素个数。例

% llength {1 2 {3 4}}

3

2.5 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}

2.6 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

2.7 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

2.8 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

2.9 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

2.10 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。

2.11 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
```

2.12 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
```

3 控制流

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

3.1 if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。

3.2 循环命令：while 、for、 foreach

3.2.1 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
}
```

3.2.2 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，代表循环体。下例与上例作用相同：

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

```
                    lappend b [lindex $a $i] }
```

3.2.3 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 {} {}"。

3.2.4 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。

3.2.5 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。

3.3 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
```

3.4 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'\'。

4 过程(procedure)

TCL培训教程

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。

4.1 过程定义和返回值

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。

4.2 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.

4.3 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数调用。调用时,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
```

4.4 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。

5 字符串操作

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。

5.1 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
```

5.2 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。

5.3 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：

-nocase 匹配时不考虑大小写

-indices 改变各个变量的值，这是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

-- 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。

1.1 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。

1.2 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。

1.2.1 **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。

1.2.2 **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。

1.2.3 **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

1.2.4 **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

1.2.5 **string last** string1 string2 ?startindex?

参照8.5.3.唯一的区别是从后往前查找

1.2.6 **string length** *string*

返回字符串string的长度.

1.2.7 **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
```

1.2.8 **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。

1.2.9 **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
```

1.2.10**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 。

1.2.11**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。

1.2.12**string toupper** *string* ?*first*? ?*last*?

同8.5.11。转换成大写。

1.2.13 **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
```

1.2.14**string trimleft** *string* ?*chars*?

同8.5.13。不过只删除左边的字符。

1.2.15**string trimright** *string* ?*chars*?

同8.5.13。不过只删除右边的字符。

2 文件访问

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)。

2.1 文件名

TCL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。

2.2 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+ 读写方式打开文件，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。

2.3 随机文件访问

默认文件输入输出方式是连续的：即每个gets或燎ead命令返回的是上次gets或燎ead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。

2.4 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。

2.5 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。

3 错误和异常

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。如果提供info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。

3.1 错误

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。

3.2 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。

3.3 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。

3.4 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。例如：

```
%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。

1 深入TCL

本章描述了一个允许您查询和操纵TCL解释器内部状态的命令集。例如，您可以通过这些命令看一个变量是否存在，可以查看数组有哪些入口(entry), 监控所有对变量的访问操作，可以重命名和删除一个命令或处理那些未定义命令的参考信息。

1.1 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。

1.2 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。

1.2.1 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
```

1.1.1 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]"
}
}
```

1.1.2 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。

1.1.3 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。

1.1.4 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。

1.1.5 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。

1.1.6 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。

1.1.7 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。

1.1.8 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。

2 历史记录

这部分内容主要描述TCL的历史机制，涉及到对以前执行过的命令的应用。历史机制在一个列表中保留了最近执行过的命令，使你不必重新敲入命令，还可以对以前的命令进行修改以创建新的命令而不必重新输入新的命令，特别是在命令较长时更加方便。

history命令的格式为：

history ?option? ?arg arg ...?

其中option 可为add, change, clear, event, info, keep, nextid, 或者redo。老版本中有substitute 和words，现在的版本（8.0以后）中被删掉了，增添了clear。下面一一介绍：

history 和**history info**相同，显示以前执行过的命令和序号，如果执行过的命名个数超过了历史记录列表允许的最大的数量，则只能按最大数量显示最近执行过的命令。

注意： history命令本身在历史记录列表中也占了一条，如：原来只有一条命令set a 123，现在输入history，则：

```
%history
   1  set a 123
   2  history
```

显示出两条历史记录。

history add *command* ?***exec***? 在历史记录列表中加一条命令，如果有**exec**选项，则执行该命令，并返回结果；如果没指定exec选项，则返回空字符串作为结果。其中添加的新命令要用双引号或花括号括上。如：

```
%history add "set b 100" exec
100
%history
1   history add "set b 100" exec
2  set b 100
3  history
```

history change *newValue* ?*event*?　　用newValue替代序号为event的命令，同时历史记录中的命令被改写。如没指定event则替换当前命令。如：

```
%history
   1  set a 123
   2  set b 23
   3  history
%history change "set a 100" 1
%history
  1  set a 100          //set a 123被set a 100替换了。
  2  set b 23
   3  history
   4  history "set a 100" 1
   5  history
%history change "set c 1"
%history
   1  set a 100
   2  set b 23
   3  history
   4  history "set a 100" 1
   5  history
   6  set c 1          //这里history change "set c 1"被set c 1替换了
   7  history
```

history clear　清除历史记录列表的内容，但记录列表允许的最大记录数这一属性仍然保留。例如：如果用history keep 50把最大记录数改变为50，history clear后记录内容空了，但最大记录数仍为50。

history event ?event?　其中event为历史事件的序号，返回该序号的命令行，如果没指定event则返回上一条命令行。如：

```
% history
  1  set a 123
  2  set b 23
```

```
  3  history
%history event 1
  set a 123
%history event
  history event 1
```

history info ?*count*?　没有count参数时和history命令一样，有count参数 时返回最近执行的*count*个命令。

history keep ?*count*?　把历史记录列表允许的最大数量设置为count。系统最初的最大记录为20。

history *nextid*　返回下一条将要添加到历史记录列表中的命令的序号。例：

```
%history
    set a 123
    2  set b 23
    3  history
%history nextid
    5        //因为history nextid的序号为4。
```

history redo ?event?　重新执行记录列表中序号为event的命令。event缺省为-1。

快捷键操作

!!　同命令 history redo相同。

！event　同命令 history redo event相同。

这两个快捷键操作对应上一节unknown命令的第3个功能。

3 TCL和C\C++

在阅读以下章节之前，假定你已经安装了TCL8.1或更高版本，同时有一个C或C＋＋的集成开发环境。注意安装TCL时一定要选择定制安装，把头文件和库文件都安装上。

TCL解释器是由一个C库实现的，在某种意义上TCL语言可以看作是一个C库。我们可以调用TCL提供的库函数来生成TCL解释器、求一个TCL脚本的值或扩展TCL固有命令。

TCL培训教程

TCL提供了一组有用的库函数来供用户访问TCL的变量、分析命令参数、操作TCL列表、求TCL表达式的值等。

3.1 生成自己的TCLSH

我们可以轻易的利用TCL库函数生成和Tclsh完全一样的程序：

```
#include "tcl.h"
#include "stdio.h"
#include "stdlib.h"
int  Tcl_AppInit(Tcl_Interp *interp);
main(int argc, char *argv[])
{
    Tcl_Main(argc, argv,Tcl_AppInit);
    exit(0);
}
int  Tcl_AppInit(Tcl_Interp *interp)
{
    Tcl_init(interp);
    return TCL_OK;
}
```

在工程中加入TCL对应的库文件，编译执行。在其中交互敲入命令，会发现和Tclsh完全一样。给出这个例子只是为了说明利用TCL提供的库函数是可以多么方便的生成自己的应用。但这个例子目前还没有加入我们自己的东西，下面一步我们就将加入我们自己扩展的命令。

3.2 扩展自己的命令：方法(一)

TCL提供了一整套供用户扩展自己的命令的函数（请参考文档《TCL库函数介绍(一)》和《TCL库函数介绍(二)》）。下面介绍怎样扩展自己的TCL命令。

TCL允许用户扩展命令，TCL解释器将把扩展命令和固有命令同等对待。TCL的扩展命令要求用C或C＋＋实现。扩展一个TCL命令大致可以分为两步：编写扩展命令对应的C/C++过程，注册命令。

3.2.1 编写扩展命令对应的C/C++过程

TCL要求所有的TCL扩展命令对应的C/C++过程具有同样的形式。在tcl.h文件中对TCL命令过程Tcl_CmdProc的定义是这样的：

```
typedef  int  Tcl_CmdProc(ClientData  clientdata,Tcl_Interp*  interp,int  argc,char*  argv[]);
```

这个函数中第一个参数clientdata暂时可以不予理会，在13.2.2将会介绍。

interp参数指向一个TCL解释器，TCL解释器是一个复杂的结构，不过让用户可见的部分比较简单，TCL中的定义如下：

```
Typedef struct Tcl_Interp {
char * result;
Tcl_freeProc * freeProc;
int  errorLine;
}Tcl_Interp;
```

argc和argv与我们编写C语言程序时main函数的两个参数的含义基本一样，argc代表命令的参数个数（包括命令名自身），argv是一个字符串数组，记录各个参数的字符形式。

命令过程的返回值为整型，将在13.2.3节单独讲解。

下面给出一个简单的扩展例子。假如我们想扩展一个命令 add ，形式为add int1 int2，实现两个整数相加。那么我们可以这样给出其实现：

```
int AddCmd(ClientData clientdata,Tcl_Interp* interp,int argc,char *  argv[])
{
    if(argc!=3)
    {
            interp->result="Useage Error! should be : add  int1 int2";
            return TCL_ERROR;
    }
    int i,j;
    if(TCL_OK!=Tcl_GetInt(interp,argv[1],&i))
    {
            sprintf(interp->result,"Expect integer but got %s",argv[1]);
            return TCL_ERROR;
    }
    if(TCL_OK!=Tcl_GetInt(interp,argv[2],&j))
    {
            sprintf(interp->result,"Expect integer but got %s",argv[2]);
            return TCL_ERROR;
    }
    i=i+j;
    sprintf(interp->result,"%d",i);
    return TCL_OK;
}
```

其中，Tcl_GetInt等函数可以参考文档《TCL库函数介绍(一)》和《TCL库函数介绍(二)》。

3.2.2 注册命令

TCL提供了一个库函数Tcl_CreateCommand负责把每一个使用C/C++语言编写的上述形式的TCL命令过程向TCL解释器注册,它的原型为:

```
Tcl_CreateCommand (Tcl_Interp* interp, char* cmdName, Tcl_CmdProc* cmdProc,
                   ClientData clientdata, Tcl_CmdDeleteProc* deleteProc)
```

第一个参数是一个解释器指针。

第二个参数是你想给自己的TCL命令取的名字。

第三个参数是一个函数指针,指向我们自己用C/C++语言编写的TCL命令过程。

一旦注册成功,TCL解释器将把命令过程cmdProc和命令名cmdName严格对应起来,一旦你在TCL中激活了命令名cmdName,TCL解释器就会自动调用命令过程cmdProc.例如:TCL解释器遇到命令add,就会调用AddCmd命令过程.

第四个参数是用来传递用户自己定义的数据对象的，TCL记录这个参数指向的对象,并把它传递给命令过程cmdProc的第一个参数。利用这个参数我们可以方便的实现在多个命令过程中对同一个数据对象进行操作。

ClientData类型在TCL的定义为 typedef void *ClientData; 它可以指向任何类型。

最后一个参数是一个函数指针，它指向一个回调函数,当我们从解释器interp中删除命令 cmdName 时， 函数 deleteProc 就会被激活， deleteProc 必须有同一的形式,在 TCL 中Tcl_CmdDeleteProc的定义为：

```
typedef void (Tcl_CmdDeleteProc) (ClientData clientData);
```

我们可以利用这个回调函数中释放clientData占用的内存。在较为复杂的TCL扩展中，我们将不可避免的要用到Tcl_CreateCommand的最后两个参数。

对于我们上面编写的AddCmd过程我们可以这样注册：

```
Tcl_CreateCommand(interp,"add",AddCmd,NULL,NULL);
```

把这个语句加到13.1节所举例子的Tcl_AppInit函数的Return TCL_OK语句之前，再把AddCmd函数的原型和实现加入那个例子，编译执行。就可以交互执行了：

```
%add 2 3
```

```
5
%add a 2
Expect integer but got 'a'
```

这样，就在TCL中扩展了一个命令add，TCL解释器遇到add命令时，就会把add命令及其后面的参数传递给对应的命令过程AddCmd处理。

3.2.3 命令返回值和命令对应的过程的返回值

TCL命令的返回值总是一个字符串，一个命令的返回值是由它对应的C/C++过程决定的。从13.2.2节所举的例子我们也可以看出来，TCL命令的返回值是由对应的C/C++过程中interp->result的值决定的。在AddCmd的例子中，如果我们把

```
Sprintf(interp->result,"%d",i);
```

修改为

```
Sprintf(interp->result,"the sum of  %s  and  %s  is :  %d",argv[1],argv[2],i);
```

重新编译执行，那么

```
%Add 2 3
the sum of  2  and  3  is :  5
```

需要注意的一点是，interp->result的预分配空间只有200个字节，如果命令的结果的长度超过200个字节，那么请使用函数Tcl_SetResult或Tcl_AppendResult来管理命令结果。

在13.2.1节中我们已经看到，一个命令对应的C/C++过程的返回值是一个整型值。TCL中命令对应的C/C++过程允许的返回值是 TCL_ERROR 、 TCL_OK 、 TCL_BREAK 、TCL_CONTINUE和TCL_RETURN，这些返回值是TCL为了方便实现控制流和对脚本文件的求值进行控制而引入的。通过设置命令过程的返回值，我们可以控制TCL脚本的执行过程，也可以实现自己的控制流。TCL中的while命令就是这样实现的。

```
int WhileCmd(ClientData clientData, Tcl_Interp *interp,
    int argc, char *argv[])
 {
    int bool;
    int code;
    if (argc != 3)
     {
            interp->result = "wrong # args";
            return TCL_ERROR;
    }
```

```
while (1)
{
Tcl_ResetResult(interp);
if (Tcl_ExprBoolean(interp, argv[1], &bool)!= TCL_OK)
        {
                return TCL_ERROR;
        }
if (bool == 0)
{
        return TCL_OK;
}
code = Tcl_Eval(interp, argv[2]);
if (code == TCL_CONTINUE)
{
        continue;
}
else if (code == TCL_BREAK)
{
        return TCL_OK;
}
else if (code != TCL_OK)
{
        return code;
}
}
```

我们在自己设计TCL扩展命令时，也应该注意设置TCL命令过程的返回值以控制TCL命令的执行过程。如果我们希望当某一个命令有错时，整个脚本还能继续往下执行，那么我们可以把那个命令过程的返回值总设为TCL_OK。例如，在扩展了add命令之后，我们可以编写如下脚本：

%eval set a 2; add a 3 ;set b 4

Expect integer but got a

%set b

Can't read "b":no such varible

这说明，由于在执行“add a 3”时出错，导致“set b 4”根本没有被执行。现在，我们在AddCmd过程中，所有的返回值全部改为TCL_OK,再编译执行：

%eval set a 2; add a 3 ;set b 4

4

%set b

4

这说明set b 4被执行了。

TCL解释器在执行脚本时，如果遇到脚本中某一个命令对应的命令过程的返回值为TCL_ERROR，那么解释器马上退出，不再对后面的脚本求值。由此可见，利用命令对应的过程的返回值，我们可以控制脚本的执行过程。根据实践，一般的原则是，如果有语法错误，应该返回TCL_ERROR，别的情况下可以根据需要决定。

3.3 扩展自己的命令：方法(二)

从上面的介绍来看，TCL从外部应用到内部实现都是基于字符串的，这样在涉及到类型转换时，需要在各种数据类型及其字符串表示之间来回转换，效率比较低，加上TCL是解释执行的，导致在利用TCL作较为大型的应用程序时执行速度不够理想，所以在TCL8.0以后的版本中，TCL在内部实现中引入了一个新的数据结构Tcl_Obj，代替了原来单纯的字符串表示。利用这个结构，可以一方面保持TCL库函数的前向兼容，另一方面可以提高TCL解释执行的速度。

由于这一结构的引入，TCL也提供了另外一套扩展TCL命令的函数，与13.2介绍的方法稍有不同。

下面先对Tcl_Obj结构作一介绍。

3.3.1 Tcl_Obj结构

在TCL中，Tcl_Obj结构定义如下：

```
typedef struct Tcl_Obj {
        int refCount;
        char *bytes;
        int length;
        Tcl_ObjType *typePtr;
        union {
                long longValue;
                double doubleValue;
                VOID *otherValuePtr;
                struct {
                        VOID *ptr1;
                        VOID *ptr2;
```

```
                    } twoPtrValue;
                } internalRep;
} Tcl_Obj;
```

在TCL的新的实现中，所有的对象(变量的值，命令参数，命令结果等等)都以Tcl_Obj对象保存：

1.分量bytes和length实际代表了对象的字符串表示，bytes必须以null结尾。如果bytes为null，表示对象的字符串表示无效。

2.typePtr 分量指向一个Tcl_ObjType，其中记录了对象的内部类型的名称、释放对象的内部表示的函数的指针等等。如果这个分量为null，那么表示internalRep分量无效。

3。当typePtr 分量不为空时，internalRep用来保存对象的内部表示，这可以是一个整数，也可以是一个双精度浮点数，或者一个指针，或者两个指针。

4。分量refCount记录对象被引用的次数，由此决定什么时候可以安全的释放这个对象占用的空间。对扩展TCL命令的用户来说，怎样维护refCount是很关键的。

采用了这种内部表示，TCL可以当需要字符串形式时，就输出Tcl_Obj的字符串形式，需要对象的实际类型时就输出实际类型的值。下面以一系列命令调用来说明TCL是怎么利用Tcl_Obj结构的：

set x 123

这个命令赋予变量x一个无类型的Tcl_Obj对象，其bytes分量包含"123"，而length分量的值为3。这时对象的字符串表示有效，而typePtr分量为空，因而内部表示internalRep无效。

puts $x

这时，因为x的字符串表示是有效的，所以直接取得x的字符串表示。

incr x

incr命令对应的过程调用 Tcl_GetIntFromObj函数从x对应的对象表示中取得一个整数。这个函数首先检查x的对应的Tcl_Obj对象是否已经是一个整数对象。因为现在typePtr为空，所以并非一个整数对象，因此它需要转换这个对象：从字符串表示中可以得到internalRep. longValue的值为123，同时把typePtr指向一个整数的Tcl_ObjType结构。这时，两种表示方法都成为有效的了。接着incr命令把对象的internalRep. longValue值加1，同时调用Tcl_InvalidateStringRep来使得x的字符串表示无效。因为此时字符串表示还是123，而实际应该为124。

puts $x

这时又需要x的字符串表示，而这时字符串表示已经无效，所以需要转换internalRep. longValue来获取x的字符串表示。这个命令调用后，两种表示又都成为有效的了。

从上面可以看出，TCL之所以采用Tcl_Obj来表示对象，就是为了尽量减少类型的转换次数。但这也给用户扩展带来了一定的额外工作量。所以如果不是对速度有特殊的要求的话。我们扩展TCL命令完全可以使用13.2介绍的方法，而不必使用下面介绍的方法。

3.3.2 编写扩展命令对应的C/C++过程

第二种TCL扩展命令对应的C/C++过程的形式和13.2介绍的稍有不同。在tcl.h文件中的定义是这样的:

```
typedef     int Tcl_ObjCmdProc( ClientData   clientData,  Tcl_Interp    *interp,
                                int   objc,  Tcl_Obj* const objv[]    );
```

前两个参数和13.2中介绍的完全一样。

objc代表Tcl_Obj对象的个数，因为命令的每个参数(包含命令名本身)都用一个对象表示，所以objc事实上相当于Argc。

第四个参数是一个对象数组，指向各个参数的对象表示。

命令过程的返回值和13.2完全一样。

下面我们把13.2中给出的扩展例子用第二种形式改写一下：

```
int AddCmd(ClientData clientdata,Tcl_Interp* interp,int objc,Tcl_Obj * const objv[])
{
    if(objc!=3)
    {
            interp->result="Useage Error!";
            return TCL_ERROR;
    }
    int i,j,l1,l2;
    if(TCL_OK!=Tcl_GetIntFromObj(interp,objv[1],&i))
    {
            sprintf(interp->result,"Expect integer but got %s",Tcl_GetStringFromObj(objv[1],&l1));
            return TCL_ERROR;
    }
    if(TCL_OK!=Tcl_GetIntFromObj(interp,objv[2],&j))
    {
            sprintf(interp->result,"Expect integer but got %s",Tcl_GetStringFromObj(objv[2],&l2));
            return TCL_ERROR;
    }
    i=i+j;
    sprintf(interp->result,"%d",i);
    return TCL_OK;
}
```

我们可以看到，除了Tcl_GetInt换成了Tcl_GetIntFromObj之外，基本没有多大的不同。

3.3.3 注册命令

相应的，第二种扩展方法也提供了自己命令注册函数,它的原型为:

```
void Tcl_CreateObjCommand  ( Tcl_Interp * interp,    char *  cmdName,
                             Tcl_ObjCmdProc *    cmdProc,
                             ClientData   clientData,  Tcl_CmdDeleteProc * deleteProc)
```

事实上这和13.2中介绍的Tcl_CreateCommand命令差不多，唯一的区别在于第三个参数，即命令过程必须是第二种形式的命令过程。

如果我们使用了第二种方法扩展了命令过程，那么必须使用第二种方法注册命令，否则会导致错误。

除了13.3.2和13.3.3中讲到的不同外，两种扩展方法没有太大的不同。不过如果我们在扩展过程中自己构造了Tcl_Obj对象的话，一定要注意维护其被引用的次数。另外也要注意，象13.3.1中的例子讲到的incr命令过程那样，如果需要的话，必须让对象的内部表示的一种失效。

3.4 利用clientData参数和deleteProc参数

在使用C＋＋扩展TCL命令时，我们不可避免的要用到类和自定义类型，这时有这种需求：几个或一组命令需要对同一个对象进行操作。这时我们除了利用全局变量外，还有其他办法吗？有！那就是利用clientData参数。

前面的例子中，我们已经看到，每一个命令函数的第一个参数是ClientData类型，而用来注册自定义命令的库函数Tcl_CreateCommand 和Tcl_CreateObjCommand，也有一个ClientData类型的参数。 这个ClientData 类型参数指向的就是自定义命令所要处理的对象，而且ClientData 被定义为void *类型，可以指向任何类型的对象。如果用户想在命令中使用自己定义的数据对象，就可以利用ClientData。

Tcl_CreateCommand和Tcl_CreateObjCommand函数还有一个参数是（Tcl_CmdDeleteProc *〕 类型，这个参数指向一个回叫(callback)函数，当删除某一个命令或程序退出时，TCL会调用对应的回叫函数来对该命令处理的数据对象进行善后处理。Tcl_CmdDeleteProc 的定义如下：

```
typedef void (Tcl_CmdDeleteProc) _ANSI_ARGS_((ClientData clientData))
```

可以看出，Tcl_CmdDeleteProc * 类型是指向一个带有一个ClientData类型参数，无返回值的函数的指针，我们可以设计一个这样的回调函数来释放自定义命令所处理的数据类型的对象申请的存储空间。

下面将以一个简单的类(class)来说明在扩展TCL命令时怎么使用自己定义的数据类型。

```
#include "Tcl.h"
```

```
class Sample
{
private:
    int x;
public:
    Sample(){ x=0;}
    Sample(Sample& sa){ x=sa.x;}
    void Set(int _x);
    int  Get();
};
int Sample::Get()
{
    return x;
}
void Sample::Set(int _x)
{
    x=_x;
}
int Tcl_InitApp(Tcl_Interp* interp);
int MyGetCmd(ClientData clientdata,Tcl_Interp* interp,int argc,char *  argv[]);
int MySetCmd(ClientData clientdata,Tcl_Interp* interp,int argc,char *  argv[]);
void DeleteProc(ClientData clientdata);
void main(int argc,char* argv[])
{
    Tcl_Main(argc,argv,Tcl_InitApp);
    return;
}
int Tcl_InitApp(Tcl_Interp* interp)
{
Tcl_Init(interp);
Sample* pSam=new Sample;
Tcl_CreateCommand(interp,   "myget",   MyGetCmd,      (ClientData)pSam,
                        (Tcl_CmdDeleteProc*)  DeleteProc);
Tcl_CreateCommand(interp,    "myset",     MySetCmd,       (ClientData)pSam,
                        (Tcl_CmdDeleteProc* )DeleteProc);
    return TCL_OK;
```

```
}
int MyGetCmd(ClientData clientdata,Tcl_Interp* interp,int argc,char *  argv[])
{
    if(argc!=1)
    {
            interp->result="Usage error!";
            return TCL_ERROR;
    }
    Sample* pSam=(Sample* )clientdata;
    if(!pSam)
    {
            return TCL_ERROR;
    }
    sprintf(interp->result,"%d",pSam->Get());
    return TCL_OK;
}
int MySetCmd(ClientData clientdata,Tcl_Interp* interp,int argc,char *  argv[])
{
    if(argc!=2)
    {
            interp->result="Usage error!";
            return TCL_ERROR;
    }
    Sample* pSam=(Sample* )clientdata;
    if(!pSam)
    {
            return TCL_ERROR;
    }
    int i;
    if(TCL_OK!=Tcl_GetInt(interp,argv[1],&i))
    {
            sprintf(interp->result,"Expect integer but got %s",argv[1]);
            return TCL_ERROR;
    }
    pSam->Set(i);
    sprintf(interp->result,"%d",i);
```

```
    return TCL_OK;
}
void DeleteProc(ClientData clientdata)
{
    int static i=0;
    if(i!=0)
            return;
    delete (char*)clientdata;
    i=1;
}
```

编译运行这个例子，交替敲入命令myget和myset可以反映出对数据对象pSam的操作导致的变化。 函数DeleteProc当我们删除myget或myset命令时会被激活，这时与这两个命令相关的对象*pSam的空间被释放.调试运行这个程序，在控制台中敲入

```
rename myget {}
```

这个命令从解释器中删除myget命令，我们会发现DeleteProc函数会被自动调用。敲入

```
rename myset {}
```

DeleteProc函数也会被自动调用。

事实上这个简单的利用类进行扩展的例子揭示了更多的含义：首先，说明我们可以在以前利用C++编写的代码的基础上进行TCL扩展，提高代码的可重用性，我们编写的代理的TCL扩展这样作的；另外，我们不但可以利用TCL对C函数作单元测试，也可以利用TCL对类的成员函数作单元测试。

3.5 在C/C++应用程序中嵌入TCL

在13.1中，我们扩展TCL的方法是利用Tcl_Main函数直接生成一个独立的TCL交互式程序。其实，我们还有更好的使用TCL的方法：在应用程序中嵌入TCL。

前面已经说过，TCL在某种意义上可以被认为是一个C函数/过程库。这决定了TCL可以和C/C++应用程序无缝集成-----我们可以轻易的把TCL嵌入到我们的一个C/C++应用程序中，这也是TCL被广泛应用的一个重要原因。TCL的可嵌入性使得我们可以为每一个应用程序提供一个支持变量、过程、控制流等编程要素并包括扩展命令和固有命令在内的功能完备的TCL扩展语言。

下面介绍在一个C/C++程序中嵌入TCL的方法.

假设我们已经用VC6生成了一个应用程序,并且加入一个菜单项TCL,OnTcl是菜单项TCL的消息响应函数:

```
void CMyExView::OnTcl()
{
    Tcl_Interp* interp=Tcl_CreateInterp();
    if(Tcl_InitApp(interp))
    {
            AfxMessageBox("TCL initialize Error!");
            return;
    }
    CFileDialog dlg(TRUE,NULL,NULL,OFN_HIDEREADONLY|OFN_FILEMUSTEXIST,
                    "Tcl Files(*.tcl)|*.tcl|All Files(*.*)|*.*||");
    if(IDOK!=dlg.DoModal())
    {
            return;
    }
    CString strFilePath=dlg.GetPathName();
    //evaluate the TCL description file
    if(TCL_OK!=Tcl_EvalFile(interp,(char*)(const char*)strFilePath))
    {
            AfxMessageBox("There are errors in your Tcl File");
            Tcl_DeleteInterp(interp);
            return;
    }
    Tcl_DeleteInterp(interp);
}
```

我们可以把前面的扩展命令add的实现部分集成到这个程序中，那么我们可以编写一个这样的脚本:

```
set a 2
set b 3
add $a $b
```

存成一个文件。点击菜单项TCL，就可以打开这个个编辑好的TCL文件并执行。遗憾的是我们不能看到执行结果。不过我们可以通过单步调试来跟踪命令的执行过程.在实际

应用中，如果想看到命令执行结果，可以在我们的扩展命令过程中加入对命令执行结果输出到某一个窗口的语句。

4 总结

这篇文章是《TCL的使用》和《TCL培训教程》的补充和修订，并加入了很多新内容。其中第9、10、11、12章是新加的内容，新加的各个章节是北研测试部TCL兴趣小组各个成员共同努力的结果：第9、12章由付剑仲完成，第10章杜祥宇完成，第11章由邓沈鸿完成，最后由我对各个章节进行了整理和修改，统一组稿。

本文基本上介绍了TCL的各个方面，特别对使用C\C++语言扩展TCL命令作了详尽的描述，这是所有的参考书上难以找到的内容。参照本文的例子，用户完全可以写出自己的TCL扩展命令。希望这篇文章能对推广在测试部使用TCL起一些推动作用。

学习一门计算机语言，从根本上来说还是要上机实习，希望测试部所有员工大家都能安装上TCL，加以实习，在应用的基础上才能进一步提高。如果需要一些本文中没有的内容，可以查阅TCL自带的帮助。