# TCL 教程

> 原文： https://zetcode.com/lang/tcl/

这是 Tcl 教程。 在本教程中，您将学习 Tcl 语言。 本教程适合初学者。

## 目录


- Tcl 语言
- 词法结构
- 基本命令
- 表达式
- 控制流
- 字符串
- 列表
- 数组
- 过程
- 输入/输出


## Tcl

Tcl 是基于字符串的脚本语言。 源代码被编译成字节码，然后由 Tcl 解释器解释。 它由 John Osterhout 于 1988 年创建。该语言通常用于快速原型制作，脚本应用，GUI 和测试。 Tcl 代表工具命令语言，其中 Tcl 脚本的源代码由命令组成。

## 相关教程

您可能也对 Tcl / Tk 教程或 Tkinter 教程感兴趣。

# Tcl

> 原文： https://zetcode.com/lang/tcl/tcl/

在 Tcl 教程的这一部分中，我们将介绍 Tcl 编程语言。

## 目标

本教程的目标是使您开始使用 Tcl 编程语言。 本教程涵盖了 Tcl 语言的核心，即变量，列表，数组，控件结构和其他核心功能。 这不是该语言的完整介绍。 这是一种快速的入门数据。 该教程是在 Ubuntu Linux 上创建的。

## Tcl

Tcl 是基于字符串的脚本语言。 源代码被编译成字节码，然后由 Tcl 解释器解释。 它是由 John Osterhout 在 1988 年创建的。其目的是创建一种易于嵌入到应用中的语言。 但通常在其原始区域之外使用。 该语言通常用于快速原型制作，脚本应用，GUI 和测试。 Tcl 代表工具命令语言，其中 Tcl 脚本的源代码由命令组成。

Tcl 是一种过程语言。 它具有一些函数式特征。 在 Tcl 版本 8.6 中添加了 OOP。

Tcl 及其 Tk GUI 工具包的官方网站是 tcl.tk

## 人气

今天有数百种编程语言在使用。 Tcl 不属于最受欢迎的 Tcl。它使用的地方有它自己的利基。 例如，快速的原型制作，测试和数据库交互。

## 交互解释器

我们可以在脚本或交互式解释器中运行 Tcl 命令。 在本教程中，我们将使用交互式 Tcl 会话来演示一些较小的代码片段。 较大的代码示例将放在 Tcl 脚本中。

tclsh 是类似 shell 的应用，可从其标准输入或文件中读取 Tcl 命令并对其进行求值。 如果不带任何参数调用，它将以交互方式运行，从标准输入读取 Tcl 命令，并将命令结果和错误消息打印到标准输出。

```
$ tclsh
% puts $tcl_version
8.6
```

```
% puts $tcl_interactive
1
```

这是 Tcl 交互式会话的示例。

```
$ tclsh
```

我们使用tclsh命令启动交互式会话。

```
% puts $tcl_version
8.6
```

提示将更改为%字符。 我们将特殊的tcl_version变量的值打印到控制台。 设置为当前使用的 Tcl 的版本。

```
% puts $tcl_interactive
1
```

tcl_interactive变量告诉我们是否处于交互模式。

```
% exit
$
```

我们使用exit命令终止交互式会话。 也可以使用 Ctrl + C 组合键。

## Tcl 脚本

我们将有第一个简单的 Tcl 脚本示例。 Tcl 程序通常具有.tcl扩展名。

```
#!/usr/bin/tclsh

# first.tcl

puts "This is Tcl tutorial"
```

在此脚本中，我们将消息打印到控制台。

```
#!/usr/bin/tclsh
```

UNIX 中的每个脚本都以 shebang 开头。 shebang 是脚本中的前两个字符：#!。shebang 之后是解释器的路径，它将执行我们的脚本。 /usr/bin/是 Tcl 外壳最常见的位置。 它也可以位于/usr/local/bin/或其他位置。

```
# first.tcl
```

Tcl 中的注释前面带有#字符。

```
puts "This is Tcl tutorial"
```

puts命令将字符串输出到控制台。

```
$ which tclsh
/usr/bin/tclsh
```

可以使用which命令找到 Tcl 解释器的路径。

```
$ chmod +x first.tcl
$ ./first.tcl
This is Tcl tutorial
```

我们使用chmod命令使脚本可执行并执行。

## 数据来源

以下资源用于创建本教程：

- tcl.tk
- en.wikipedia.org/wiki/Tcl

在 Tcl 教程的这一部分中，我们介绍了 Tcl 语言。

# Tcl 语法结构

> 原文： https://zetcode.com/lang/tcl/lexis/

像人类语言一样，计算机语言也具有词汇结构。 Tcl 语言的词汇包括基本元素和适用于它们的规则。 单词是 Tcl 中的基本元素。 单词可以是命令，也可以是命令参数。 替换是 Tcl 语法的基本规则之一。

## 命令

Tcl 是基于字符串的解释命令语言。 Tcl 脚本由命令组成，这些命令由换行符或分号分隔。 命令是基本的执行元素。 命令后跟一个或多个单词（作为参数）。 每个参数都由空格分隔。

Tcl 命令的格式如下：`command arg1 arg2 arg3 ...` Tcl 解释器接受句子的每个单词并对其求值。 第一个单词被认为是命令。 大多数 Tcl 命令是可变的。 这意味着它们可以处理可变数量的参数。

解析 Tcl 脚本后，将求值命令。 每个命令在其自己的上下文中解释单词。

```
puts "Tcl language"
```

在上面的代码摘录中，我们具有`puts`命令。 此命令将消息打印到控制台。 `"Tcl language"`是正在打印的字符串。 与其他语言不同，字符串不能用双引号引起来； 除非有空白。

```
#!/usr/bin/tclsh

puts zetcode.com; puts androida.co

puts zetcode
puts androida
```

在第一种情况下，命令用分号`;`字符分隔。 在第二种情况下，它们之间用换行符分隔。

## 替换

Tcl 中存在三种替换。

- 命令替换
- 变量替换
- 反斜杠替换

方括号用于命令替换。

```
% puts [expr 1+2]
3
```

expr命令用于执行算术计算。 首先，求值方括号之间的命令，并将结果返回到puts命令。 然后，puts命令求值结果并将其打印到控制台。

如果单词包含美元符号$，则 Tcl 执行变量替换。 单词中的美元符号和以下字符由变量的值替换。

```
#!/usr/bin/tclsh

set name Jane

puts name
puts $name
```

我们创建一个名为name的变量，并为其设置一个值。

```
puts name
```

在这种情况下，我们将字符串name打印到控制台。

```
puts $name
```

在第二种情况下，参数以$字符开头。 name变量的值被打印到控制台。

```
$ ./name.tcl
name
Jane
```

这是示例的输出。

通过反斜杠替换，我们可以摆脱字符的原始含义。 例如，\n代表换行，\t是制表符。

```
% puts "This is \t Sparta"
This is          Sparta
```

在这里，\t序列被制表符代替。

```
#!/usr/bin/tclsh

puts "This was a \"great\" experience"
puts "The \\ character is the backslash character"

puts "20000\b\b miles"
```

如果要在引号字符中包含引号，请使用反斜杠替换。 另外，如果要打印\字符，则必须在其前面加上其他反斜杠。 \b替换为退格键。

```
$ ./backslash.tcl
This was a "great" experience
The \ character is the backslash character
200 miles
```

运行示例。

## 注释

人们使用注释来澄清源代码。 在 Tcl 中，注释以#字符开头。

```
# example of a puts command
puts "Tcl language"
```

tclsh会忽略#字符之后的所有字符。

```
puts "Tcl language" ; # example of a puts command
```

仅当使用分号时才可以进行内联注释。

## 空格

空白用于分隔 Tcl 源中的单词。 它还用于提高源代码的可读性。

```
set name Jane
```

set命令采用两个参数，两个参数之间用空格隔开。

```
set age        32
set name       Robert
set occupation programmer
```

如果要提高源代码的清晰度，我们可能会使用更多空间。

```
set vals { 1 2 3 4 5 6 7 }
puts $vals
```

空白用于分隔 Tcl 列表中的项目。 在基于 C 的语言中，我们将使用逗号字符。

## 变量

变量是保存值的标识符。 在编程中，我们说我们为变量分配或设置了一个值。 从技术上讲，变量是对存储值的计算机内存的引用。 变量名称区分大小写。 这意味着Name, name和NAME引用了三个不同的变量。

Tcl 中的变量是使用set命令创建的。 要获取变量的值，其名称前面带有$字符。

```
#!/usr/bin/tclsh

set name Jane
set Name Julia
set NAME Erika

puts $name
puts $Name
puts $NAME
```

在上面的脚本中，我们设置了三个变量。 变量名称相同，只是大小写不同。 但是，不建议这种做法。

```
$ ./case.tcl
Jane
Julia
Erika
```

这是case.tcl脚本的输出。

## 大括号

大括号{}在 Tcl 中具有特殊含义。 大括号内的单词替换已禁用。

```
#!/usr/bin/tclsh

set name {Julia Novak}
puts $name

puts "Her name is $name"
puts {Her name is $name}
```

这是一个小脚本，显示了 Tcl 中花括号的用法。

```
set name {Julia Novak}
```

可以使用大括号代替双引号来设置以空格分隔的字符串。

```
puts "Her name is $name"
```

在此变量被替换。

```
puts {Her name is $name}
```

使用花括号时，变量不会被替换。 一切都按字面意思打印。

```
$ ./braces.tcl
Julia Novak
Her name is Julia Novak
Her name is $name
```

`braces.tcl`脚本的输出。

```
#!/usr/bin/tclsh

set numbers { 1 2 3 4 5 6 7 }
puts $numbers

puts "Braces {} are reserved characters in Tcl"
puts {Braces {} are reserved characters in Tcl}
```

大括号用于创建列表。 列表是 Tcl 中的基本数据类型。

```
set numbers { 1 2 3 4 5 6 7 }
```

在此处创建数字列表。

```
puts "Braces {} are reserved characters in Tcl"
puts {Braces {} are reserved characters in Tcl}
```

双引号或其他大括号内的大括号被视为常规字符，没有特殊含义。

```
$ ./braces2.tcl
 1 2 3 4 5 6 7
Braces {} are reserved characters in Tcl
Braces {} are reserved characters in Tcl
```

`braces2.tcl`脚本的输出。

## 方括号

方括号[]用于创建嵌套命令。 这些嵌套命令在 Tcl 源代码行上的 main 命令之前执行。 它们用于将一个命令的结果作为参数传递给另一命令。

```
#!/usr/bin/tclsh

set cwd [pwd]
puts $cwd

puts [clock format [clock seconds] -format "%Y-%m-%d %T"]
```

在上面的代码示例中，我们显示了一些嵌套命令。

```
set cwd [pwd]
```

`pwd`命令返回脚本的当前工作目录。 它放在方括号之间，使其成为嵌套命令。 首先执行`pwd`命令，然后将命令的结果设置为`cwd`变量。

```
puts [clock format [clock seconds] -format "%Y-%m-%d %T"]
```

嵌套命令可以嵌套在其他嵌套命令中。 首先，执行clock seconds。它以秒为单位返回当前本地时间。 结果传递到clock format命令，该命令以可读形式格式化时间。 最后，格式化的时间返回到puts命令，该命令将其打印到控制台。

```
$ ./nested.tcl
/home/janbodnar/prog/tcl/lexis
2015-01-16 16:45:04
```

示例的输出。

## 双引号

双引号将单词组合为命令的单个参数。 美元符号，方括号和反斜杠在引号内解释。

```
#!/usr/bin/tclsh

set distro Ubuntu
puts "The Linux distribution name is $distro"

puts "The current working directory: [pwd]"
puts "2000000\b\b\b miles"
```

这是在 Tcl 中使用引号的实际示例。

```
puts "The Linux distribution name is $distro"
```

变量发行版在引号字符内求值。 $distro被替换为"Ubuntu"。

```
puts "The current working directory: [pwd]"
```

方括号内的命令也会被解释。 在这里，我们使用pwd命令获取当前的工作目录。

```
puts "2000000\b\b\b miles"
```

\b转义序列删除前一个字符。 在我们的例子中，三个零被删除。

```
$ ./quotes.tcl
The Linux distribution name is Ubuntu
The current working directory: /home/janbodnar/prog/tcl/lexis
2000 miles
```

quotes.tcl示例的输出。

## 反斜杠

反斜杠字符可以在 Tcl 中以三种不同的方式使用。 它引入了一些特殊字符，称为转义序列。 这些可以是换行符，选项卡，退格键等等。 它转义了特殊 Tcl 字符 ( $, {}, "", \, () ) 的含义。 最后，它可以用作换行符。

```
#!/usr/bin/tclsh

puts "0\t1"

set name Jane
puts \$name
puts \\$name

puts "He said: \"There are plenty of them\""
puts "There are currently many Linux\
distributions in the world"
```

上面的脚本显示了如何在 Tcl 中使用反斜杠字符。

```
puts "0\t1"
```

\t字符在 Tcl 中具有特殊含义。 它代表制表符空白字符。 当我们执行脚本时，在 0 和 1 内放置 8 个空格。

```
puts \$name
```

有了反斜杠，我们就可以摆脱美元符号的含义。 在我们的情况下，将$name字符打印到控制台。

```
puts \\$name
```

反斜杠也可以转义。 这里，反斜杠字符和name变量的值被打印到控制台。

```
puts "He said: \"There are plenty of them\""
```

我们可以通过转义内部双引号的含义来形成直接语音。

```
puts "There are currently many Linux\
distributions in the world"
```

如果源行太宽，我们可以使用反斜杠字符继续下一行，转义换行符。

```
$ ./backslash.tcl
0       1
$name
\Jane
He said: "There are plenty of them"
There are currently many Linux distributions in the world
```

运行示例。

## 圆括号

圆括号用于指示数组下标或更改expr命令的运算符的优先级。

```
#!/usr/bin/tclsh

set names(1) Jane
set names(2) Robert

puts $names(1)
puts $names(2)

puts [expr (1+3)*5]
```

这是一个简单的示例，在 Tcl 中带有圆括号。

```
puts $names(1)
```

我们使用圆括号通过圆括号内指定的键访问值。

```
puts [expr (1+3)*5]
```

在这里，我们更改了运算符的优先级。 首先将 1 和 3 相加，然后将结果乘以 5。

```
$ ./roundb.tcl
Jane
Robert
20
```

这是round.tcl脚本的输出。

在本章中，我们描述了 Tcl 语言的词汇。

# Tcl 中的基本命令

> 原文： https://zetcode.com/lang/tcl/basiccommands/

在 Tcl 教程的这一部分中，我们介绍了一些基本的 Tcl 命令。

涵盖的 Tcl 命令包括puts，open，gets，flush，incr，info，set和unset命令。

## puts命令

在第一个示例中，我们将提到puts命令。 puts命令用于将消息打印到控制台或文件等其他通道。 该命令具有以下语法：

```
puts ?-nonewline? ?channelId? string
```

puts是命令名称。 在问号之间指定了可选参数。 -nonewline开关禁止换行符。 默认情况下，该命令在每条消息中放置换行符。 channelId必须是开放通道（例如 Tcl 标准输入通道stdin）的标识符，该标识符是调用open或socket的返回值。 如果未指定，则默认为标准输出通道stdout。最后，string是要打印的消息。

```
#!/usr/bin/tclsh

puts "This is Tcl tutorial"
puts stdout "This is Tcl tutorial"
```

puts命令将消息打印到控制台。 这两个命令调用执行相同的操作。

## open命令

open命令打开文件，串行端口或命令管道，并返回通道标识符。 在下面的示例中，我们使用命令打开文件。

```
#!/usr/bin/tclsh

puts [open messages w] "This is Tcl tutorial"
```

puts命令用于写入文件，该文件通过open命令打开以进行写入。

```
$ cat messages
This is Tcl tutorial
```

我们将显示由上述 Tcl 脚本创建的消息文件的内容。

## gets和flush命令

gets命令从通道读取一条线，flush命令刷新通道的缓冲输出。 在下面的示例中，我们创建一个向用户打招呼的脚本。

```
#!/usr/bin/tclsh

puts -nonewline "What is your name? "
flush stdout
gets stdin name
puts "Hello $name"
```

在此示例中，我们请求用户输入并以自定义问候语打印输入。

```
puts -nonewline "What is your name? "
```

-nonewline选项禁止换行。 提示保持在同一行。

```
flush stdout
```

输出被缓冲。 要在命令运行后立即查看输出，我们可以使用flush命令。 stdout是标准输出。 在我们的例子中，它是一个终端。 它在 Tcl 中称为频道 ID。

```
gets stdin name
```

gets命令从标准输入读取一行。 结果存储在名称变量中。

```
puts "Hello $name"
```

最后，我们向用户致意。

```
$ ./name.tcl
What is your name? Jan
```

```
Hello Jan
```

运行示例。

## incr命令

incr递增变量的值。 它具有以下语法：

```
incr varName ?increment?
```

如果将参数传递给命令，则将其值添加到变量的值； 否则，该值将增加 1。

```
#!/usr/bin/tclsh

# incr_cmd.tcl

set x 5

incr x
puts $x

incr x 4
puts $x
```

该代码示例设置一个变量并将其递增两次。

```
set x 5
```

值 5 设置为x变量。

```
incr x
puts $x
```

x变量增加 1。数字 6 打印到控制台。

```
incr x 4
puts $x
```

数字 4 被添加到x变量中。 puts命令将 10 打印到控制台。

```
$ ./incr_cmd.tcl
6
10
```

这是incr_cmd.tcl脚本的输出。

## info命令

info命令返回有关 Tcl 解释器状态的信息。

```
#!/usr/bin/tclsh

puts [info tclversion]
puts [info host]
puts [info exists var]
```

info命令具有多个选项。 我们展示其中的三个。

```
puts [info tclversion]
```

在这里，我们打印 Tcl 解释器的版本。

```
puts [info host]
```

该行显示主机名。

```
puts [info exists var]
```

最后，我们检查变量var是否设置。

## set和unset设置命令

set命令用于创建和读取变量。 unset命令销毁变量。

```
#!/usr/bin/tclsh

set x 23
puts $x
puts [set x]

unset x
puts [info exists x]
```

显示set和unset命令的示例。

```
set x 23
```

我们创建一个x变量并为其分配值 23。

```
puts $x
```

我们打印x变量的值。

```
puts [set x]
```

此行还打印x变量的值。 具有一个参数的set命令读取变量的值。 该值将传递到puts命令并打印到终端。

```
unset x
```

变量x被破坏。

```
puts [info exists x]
```

我们使用info exists命令验证变量的存在。

## 命令行参数

像任何其他脚本一样，Tcl 脚本也可以使用命令行参数。 Tcl 具有三个预定义变量。

- $argc - 传递给脚本的参数数量
- $argv - 参数列表
- $argv0 - 脚本名称

```
#!/usr/bin/tclsh

puts "The script has $argc arguments"
puts "The list of arguments: $argv"
puts "The name of the script is $argv0"
```

我们在此脚本中使用所有预定义的变量。

```
$ ./args.tcl 1 2 3
The script has 3 arguments
The list of arguments: 1 2 3
The name of the script is ./args.tcl
```

运行示例。

本章介绍了 Tcl 语言的一些基础知识。

# Tcl 中的表达式

> 原文： https://zetcode.com/lang/tcl/expressions/

在 Tcl 教程的这一部分中，我们将讨论表达式。 在 Tcl 语言中，表达式未内置到核心语言中。 而是使用`expr`命令对表达式求值。

表达式是根据操作数和运算符构造的。 表达式的运算符指示将哪些运算应用于操作数。 表达式中运算符的求值顺序由运算符的优先级和关联性确定。

运算符是特殊符号，表示已执行某个过程。 编程语言的运算符来自数学。 程序员处理数据。 运算符用于处理数据。 操作数是运算符的输入（参数）之一。

下表显示了 Tcl 语言中使用的一组运算符：

| 类别 | 符号 |
| --- | --- |
| 按位，符号，逻辑非 | `- + ~ !` |
| 求幂 | `**` |
| 算术 | `+ - * / %` |
| 移位 | `<< >>` |
| 关系 | `== != < > <= >=` |
| 字符串比较 | `eq ne` |
| 列表 | `in ni` |
| 按位 | `& | ^` |
| 布尔 | `&& ||` |
| 三元 | `?:` |

一个运算符通常有一个或两个操作数。 那些仅使用一个操作数的运算符称为一元运算符。 那些使用两个操作数的对象称为二进制运算符。 还有一个三元运算符`?:`，它可以处理三个操作数。

## 基本运算符

基本运算符是常用的运算符。 这些是符号运算符，算术运算符，模和幂运算符。

```
#!/usr/bin/tclsh

puts [expr +2]
puts [expr -2]
puts [expr -(-2)]
puts [expr 2+2]
puts [expr 2-2]
puts [expr 2*2]
```

```
puts [expr 2/2]
puts [expr 2/2.0]
puts [expr 2 % 2]
puts [expr 2 ** 2]
```

上面的示例显示了 Tcl 中常用运算符的用法。

```
puts [expr +2]
```

在此代码行中，我们使用加号运算符。 它对数量没有影响。 它仅表示该数字为正。 可以将其省略，并且在大多数情况下可以将其省略。

```
puts [expr -2]
puts [expr -(-2)]
```

负号运算符是强制性的。 它说数字是负数。 减号运算符更改数字的符号。 在第二行中，负号运算符将 -2 更改为正 2。

```
puts [expr 2+2]
puts [expr 2-2]
puts [expr 2*2]
puts [expr 2/2]
```

上面的几行显示了常用的算术运算符。

```
puts [expr 2 % 2]
```

% 是取模或余数的运算符。 它找到一个数除以另一个的余数。 表达式2 % 2，2 模 2 为 0，因为 2 一次变成 2，余数为 0。因此代码行向控制台输出零。

```
puts [expr 2 ** 2]
```

这是求幂运算符。 代码行将 4 打印到控制台。

```
```

```
$ ./exp.tcl
2
-2
2
4
0
4
1
1.0
0
4
```

示例的输出。

## 除法运算符

刚入门的程序员通常会因除法运算而感到困惑。 在许多编程语言中，有两种除法运算：整数和非整数。 这也适用于 Tcl。

```
% expr 3/2
1
% expr 3/2.0
1.5
```

注意整数除法和浮点除法之间的区别。 当至少一个操作数是浮点数时，结果也就是浮点值。 结果更加准确。 如果两个操作数均为整数，则结果也为整数。

## 赋值和增量运算符

Tcl 中没有赋值运算符=，也没有增量和减量（ ++和-- ）运算符。 这些运算符在其他计算机语言中很常见。 取而代之的是，Tcl 具有命令。

```
% set a 5
5
% incr a
6
% incr a
7
% incr a -1
6
```

上面的代码显示了哪些命令用于实现缺少的运算符。

```
% set a 5
```

在 Python 中，我们将执行`a = 5`。在 Tcl 中，我们使用`set`命令将值设置为变量。

```
% incr a
6
```

在 C，Java 和许多其他语言中，我们将通过以下方式将变量增加一个：`a++;`。在 Tcl 中，我们使用`incr`命令。默认情况下，该值增加 1。

```
% incr a -1
6
```

上面的代码显示了如何将变量减 1，这由`--`减量运算符以基于 C 的语言完成。

## 布尔运算符

在 Tcl 中，我们具有以下逻辑运算符：

| 符号 | 名称 |
| --- | --- |
| && | 逻辑与 |
| \|\| | 逻辑或 |
| ! | 否定 |

布尔运算符也称为逻辑运算符。

```
#!/usr/bin/tclsh

set x 3
set y 8

puts [expr $x == $y]
puts [expr $y > $x]

if {$y > $x} {

    puts "y is greater than x"
}
```

许多表达式导致布尔值。 布尔值用于条件语句中。

```
puts [expr $x == $y]
puts [expr $y > $x]
```

关系运算符始终导致布尔值。 这两行显示 0 和 1。在 Tcl 中，0 为false，任何非零值为true。

```
if {$y > $x} {

    puts "y is greater than x"
}
```

仅当满足括号内的条件时，才执行if命令的主体。 $y > $x返回 true，因此消息"y大于x"被打印到终端。

```
#!/usr/bin/tclsh

puts [expr 0 && 0]
puts [expr 0 && 1]
puts [expr 1 && 0]
puts [expr 1 && 1]
```

此示例显示了逻辑和&&运算符。 仅当两个操作数均为true时，它的求值结果为true。

```
$ ./andoperator.tcl
0
0
0
1
```

如果两个操作数中的任何一个为true，则逻辑或||运算符的计算结果为true。

```
#!/usr/bin/tclsh

puts [expr 0 || 0]
puts [expr 0 || 1]
puts [expr 1 || 0]
puts [expr 1 || 1]
```

如果运算符的任一侧为真，则操作的结果为真。

```
$ ./oroperator.tcl
0
1
1
1
```

否定运算符!将true设为false，并将false设为false。

```
#!/usr/bin/tclsh

puts [expr ! 0]
puts [expr ! 1]
puts [expr ! (4<3)]
```

该示例显示了否定运算符的作用。

```
$ ./not.tcl
1
0
1
```

||和&&运算符经过短路求值。 短路求值意味着仅当第一个参数不足以确定表达式的值时才求值第二个参数：当逻辑的第一个参数的值等于false时，总值必须为false; 当逻辑或的第一个参数为true时，总值必须为true。短路求值主要用于提高性能。

一个例子可以使这一点更加清楚。

```
#!/usr/bin/tclsh

proc One {} {

    puts "Inside one"
    return false
}

proc Two {} {

    puts "Inside two"
    return true
}
```

```
puts "Short circuit"

if { [One] && [Two] } {

    puts "Pass"
}

puts "####################"

if { [Two] || [One] } {

    puts "Pass"
}
```

在示例中，我们有两个过程。（过程和条件将在后面描述。）它们在布尔表达式中用作操作数。我们将看看它们是否被调用。

```
if { [One] && [Two] } {

    puts "Pass"
}
```

One过程返回false。短路&&不求值第二步。没有必要。一旦操作数为假，逻辑结论的结果始终为假。控制台上仅打印"Inside one"。

```
puts "####################"

if { [Two] || [One] } {

    puts "Pass"
}
```

在第二种情况下，我们使用||运算符，并使用Two过程作为第一个操作数。在这种情况下，"Inside two"和"Pass"字符串将打印到终端。同样，也不必求值第二个操作数，因为一旦第一个操作数计算为true，则逻辑或始终为true。

```
$ ./shortcircuit.tcl
Short circuit
Inside one
####################
Inside two
Pass
```

shorcircuit.tcl脚本的结果。

## 关系运算符

关系运算符用于比较值。 这些运算符总是产生布尔值。 在 Tcl 中，0 代表false，1 代表true。关系运算符也称为比较运算符。

| 符号 | 含义 |
| --- | --- |
| < | 小于 |
| <= | 小于或等于 |
| > | 大于 |
| >= | 大于或等于 |
| == | 等于 |
| != | 不等于 |

该表显示了六个 Tcl 关系表达式。

```
#!/usr/bin/tclsh

puts [expr 3 < 4]
puts [expr 3 == 4]
puts [expr 4 >= 3]
puts [expr 4 != 3]
```

在 Tcl 中，我们使用==运算符比较数字。 某些语言（例如 Ada，Visual Basic 或 Pascal）使用=比较数字。

```
$ ./rel.tcl
1
0
1
1
```

该示例打印四个布尔值。

## 按位运算符

小数对人类是自然的。 二进制数是计算机固有的。 二进制，八进制，十进制和十六进制符号仅是相同数字的符号。 按位运算符使用二进制数的位。 像 Tcl 这样的高级语言很少使用按位运算符。

| 符号 | 含义 |
| --- | --- |

| 符号 | 含义 |
| --- | --- |
| ~ | 按位取反 |
| ^ | 按位异或 |
| & | 按位与 |
| \| | 按位或 |

按位取反运算符分别将 1 更改为 0，将 0 更改为 1。

```
% puts [expr ~7]
-8
% puts [expr ~-8]
7
```

运算符将 7 的所有位都还原。这些位之一还确定该数字是否为负。 如果我们再一次对所有位取反，我们将再次得到 7。

按位，运算符在两个数字之间进行逐位比较。 仅当操作数中的两个对应位均为 1 时，位位置的结果才为 1。

```
    00110
&   00011
=   00010
```

第一个数字是二进制表示法 6，第二个数字是 3，结果是 2。

```
% puts [expr 6 & 3]
2
% puts [expr 3 & 6]
2
```

按位或运算符在两个数字之间进行逐位比较。 如果操作数中的任何对应位为 1，则位位置的结果为 1。

```
    00110
|   00011
=   00111
```

结果为00110或十进制 7。

```
% puts [expr 6 | 3]
7
% puts [expr 3 | 6]
7
```

按位互斥或运算符在两个数字之间进行逐位比较。 如果操作数中对应位中的一个或另一个（但不是全部）为 1，则位位置的结果为 1。

```
    00110
^   00011
=   00101
```

结果为00101或十进制 5。

```
% puts [expr 6 ^ 3]
5
% puts [expr 3 ^ 6]
5
```

## 扩展运算符

扩展运算符{*}使列表中的每个项目成为当前命令的单独参数。 列表是基本的 Tcl 数据结构； 在下一章中将进行介绍。

```
#!/usr/bin/tclsh

set nums {1 2 3 4 5 6}
puts $nums
puts [tcl::mathfunc::max {*}$nums]
puts [tcl::mathfunc::min {*}$nums]
```

扩展运算符与两个数学函数一起使用。

```
set nums {1 2 3 4 5 6}
```

将创建一个名为nums的数字列表。 列表是值的有序集合。

```
puts $nums
```

列表的内容被打印到终端。

```
puts [tcl::mathfunc::max {*}$nums]
```

tcl::mathfunc::max是标准的数学函数。 它不处理列表。 数字应作为单独的参数传递。 扩展运算符将项目列表转换为单个项目。

```
$ ./expansion.tcl
1 2 3 4 5 6
6
1
```

示例输出。

## 运算符优先级

运算符优先级告诉我们首先求值哪个运算符。 优先级对于避免表达式中的歧义是必要的。

以下表达式 28 或 40 的结果是什么？

```
 3 + 5 * 5
```

像数学中一样，乘法运算符的优先级高于加法运算符。 结果是 28。

```
(3 + 5) * 5
```

我们使用括号来更改求值顺序。 括号内的表达式始终首先被求值。

下表显示了按优先级排序的常见 Tcl 运算符（最高优先级优先）：

| 类别 | 符号 | 关联性 |
| --- | --- | --- |
| 按位，逻辑非 | - + ~ ! | 左 |
| 求幂 | ** | 左 |

| 类别 | 符号 | 关联性 |
| --- | --- | --- |
| 算术 | + - * / % | 左 |
| 移位 | << >> | 左 |
| 关系 | == != < > <= >= | 左 |
| 字符串比较 | eq ne | 左 |
| 列表 | in ni | 左 |
| 按位 | & | ^ | 左 |
| 布尔 | && || | 左 |
| 三元 | ?: | 右 |

表的同一行上的运算符具有相同的优先级。

```
!/usr/bin/tclsh

puts [expr 3 + 5 * 5]
puts [expr (3 + 5) * 5]

puts [expr ! 1 || 1]
puts [expr ! (1 || 1)]
```

在此代码示例中，我们显示一些常见的表达式。 每个表达式的结果取决于优先级。

```
puts [expr 3 + 5 * 5]
```

该行打印 28。乘法运算符的优先级高于加法。 首先，计算5*5的乘积，然后加 3。

```
puts [expr (3 + 5) * 5]
```

圆括号可用于更改优先级。 在上面的表达式中，将 3 加到 5，然后将结果乘以 5。

```
puts [expr ! 1 || 1]
```

在这种情况下，否定运算符具有更高的优先级。 首先，第一个true（1）值取反为false（0），然后||运算符将false和true组合在一起，最后得到true。

```
$ ./precedence.tcl
28
40
1
0
```

输出。

## 关联性

有时，优先级不能令人满意地确定表达式的结果。 还有另一个规则称为关联性。 运算符的关联性确定优先级与相同的运算符的求值顺序。

```
9 / 3 * 3
```

此表达式的结果是 9 还是 1？ 乘法，删除和模运算符从左到右关联。 因此，该表达式的计算方式为：`(9 / 3) * 3`，结果为 9。

算术，布尔，关系和按位运算符都是从左到右关联的。

三元运算符是正确关联的。

## 三元运算符

三元运算符`?:`是条件运算符。 对于要根据条件表达式选择两个值之一的情况，它是一个方便的运算符。

```
cond-exp ? exp1 : exp2
```

如果`cond-exp`为`true`，则求值`exp1`并返回结果。 如果`cond-exp`为`false`，则求值`exp2`并返回其结果。

```
#!/usr/bin/tclsh

set age 32
set adult [expr $age >= 18 ? true : false]

puts "Adult: $adult"
```

在大多数国家/地区，成年取决于您的年龄。 如果您的年龄超过特定年龄，则您已经成年。 对于三元运算符，这是一种情况。

```
set adult [expr $age >= 18 ? true : false]
```

首先，求值赋值运算符右侧的表达式。 三元运算符的第一阶段是条件表达式求值。 因此，如果年龄大于或等于18，则返回?字符后的值。 如果不是，则返回:字符后的值。 然后将返回值分配给成人变量。

```
$ ./ternary.tcl
Adult: true
```

32 岁的成年人是成人。

## 计算素数

我们将计算素数。 本教程的稍后部分将介绍一些功能（列表，循环）。

```
#!/usr/bin/tclsh

# primes.tcl

set nums { 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
}

puts "Prime numbers:"

foreach num $nums {

    if { $num==1 } { continue }

    if { $num==2 || $num==3 } {

        puts -nonewline "$num "
        continue
    }

    set i [expr int(sqrt($num))]
    set isPrime true

    while { $i > 1 } {

        if { $num % $i == 0 } {

            set isPrime false
        }

        incr i -1
    }
```

```
    if { $isPrime } {

        puts -nonewline "$num "
    }
}

puts ""
```

在上面的示例中，我们处理了许多不同的运算符。 质数（或质数）是大于 1 的自然数，具有正好两个不同的自然数除数：1 和它本身。 我们拾取一个数字并将其除以数字，从 1 到拾取的数字。 实际上，我们不必尝试所有较小的数字，我们可以将数字除以所选数字的平方根。 该公式将起作用。 我们使用余数除法运算符。

```
set nums { 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
}
```

我们从该整数列表中计算素数。

```
if { $num==1 } { continue }
```

根据定义，1 不是质数。 continue命令跳到循环的下一个迭代。

```
if { $num==2 || $num==3 } {

    puts -nonewline "$num "
    continue
}
```

我们跳过 2 和 3 的计算。它们是质数，不需要进一步的计算。 请注意等式和条件或运算符的用法。 ==的优先级高于||运算符。 因此，我们不需要使用括号。

```
set i [expr int(sqrt($num))]
```

如果我们仅尝试小于所讨论数字的平方根的数字，那么我们可以。

```

```

```
while { $i > 1 } {

    if { $num % $i == 0 } {

        set isPrime false
    }

    incr i -1
}
```

在此while循环中，i是计算出的数字的平方根。 我们使用incr命令将每个循环周期将i减少 1。当i小于 1 时，循环结束。 例如，我们有 9。9 的平方根是 3。我们将 9 的数字除以 3 和 2。这对于我们的计算就足够了。

```
if { $isPrime } {

    puts -nonewline "$num "
}
```

如果余数除法运算符针对任何 i 值返回 0，则说明该数字不是质数。

```
$ ./primes.tcl
Prime numbers:
2 3 5 7 11 13 17 19 23 29 31
```

这是primes.tcl脚本的输出。

在 Tcl 教程的这一部分中，我们介绍了 Tcl 表达式。

# Tcl 中的控制流

> 原文： https://zetcode.com/lang/tcl/flowcontrol/

在 Tcl 教程的这一部分中，我们将讨论流控制。 我们将定义几个命令，这些命令使我们能够控制 Tcl 脚本的流程。

在 Tcl 语言中，有几个命令可用于更改程序流程。 运行程序时，其命令从源文件的顶部到底部执行。 逐一。 可以通过特定命令更改此流程。 命令可以多次执行。 有些命令是有条件的。 仅在满足特定条件时才执行它们。

## if命令

if命令具有以下一般形式：

```
if expr1 ?then? body1 elseif expr2 ?then? body2 elseif ... ?else? ?bodyN?
```

if命令用于检查表达式是否为真。 如果为真，则执行命令主体。 主体用花括号括起来。

if命令计算一个表达式。 该表达式必须返回一个布尔值。 在 Tcl 中，1, yes, true表示true, 0, no, false表示false。

```
!/usr/bin/tclsh

if yes {
    puts "This message is always shown"
}
```

在上面的示例中，始终执行由{ }字符包围的主体。

```
#!/usr/bin/tclsh

if true then {
    puts "This message is always shown"
}
```

then命令是可选的。 如果我们认为可以使用它，它将使代码更加清晰。

我们可以使用else命令创建一个简单的分支。 如果if命令后方括号内的表达式的计算结果为false，则将自动执行else命令后方的命令。

```

```

```
#!/usr/bin/tclsh

set sex female

if {$sex == "male"} {

    puts "It is a boy"
} else {

    puts "It is a girl"
}
```

我们有性别变量。 它具有"female"字符串。 布尔表达式的计算结果为false，我们在控制台中得到"It is a girl"。

```
$ ./girlboy.tcl
It is a girl
```

我们可以使用elseif命令创建多个分支。 仅当不满足先前条件时，elseif命令才会测试其他条件。 请注意，我们可以在测试中使用多个elseif命令。

```
#!/usr/bin/tclsh

# nums.tcl

puts -nonewline "Enter a number: "
flush stdout
set a [gets stdin]

if {$a < 0} {

    puts "the number is negative"
} elseif { $a == 0 } {

    puts "the numer is zero"
} else {

    puts "the number is positive"
}
```

在上面的脚本中，我们提示您输入一个值。 我们测试该值是否为负数或正数或等于零。 如果第一个表达式的计算结果为false，则对第二个表达式进行计算。 如果不满足先前的条件，则将执行else命令之后的主体。

```
$ ./nums.tcl
Enter a number: 2
the number is positive
$ ./nums.tcl
Enter a number: 0
the numer is zero
$ ./nums.tcl
Enter a number: -3
the number is negative
```

多次运行该示例。

## switch命令

switch命令将其字符串参数与每个模式参数按顺序进行匹配。 一旦找到与字符串匹配的模式，它就会通过将其递归传递给 Tcl 解释器来求值以下主体参数，并返回该求值结果。 如果最后一个模式参数为默认值，则它匹配任何内容。 如果没有任何模式参数与字符串匹配，并且没有给出默认值，那么switch命令将返回一个空字符串。

```
#!/usr/bin/tclsh

# switch_cmd.tcl

puts -nonewline "Select a top level domain name:"
flush stdout

gets stdin domain

switch $domain {

    us { puts "United States" }
    de { puts Germany }
    sk { puts Slovakia }
    hu { puts Hungary }
    default { puts "unknown" }
}
```

在脚本中，我们提示您输入域名。 有几种选择。 例如，如果该值等于我们，则将"United States"字符串打印到控制台。 如果该值与任何给定值都不匹配，那么将执行默认主体，并将unknown打印到控制台。

```
$ ./switch_cmd.tcl
Select a top level domain name:sk
Slovakia
```

我们已将sk字符串输入到控制台，该程序响应了斯洛伐克。

## while命令

while命令是一个控制流命令，它允许根据给定的布尔条件重复执行代码。

while命令在大括号括起来的块内执行命令。 每次将表达式求值为true时都将执行命令。

```
#!/usr/bin/tclsh

# whileloop.tcl

set i 0
set sum 0

while { $i < 10 } {

    incr i
    incr sum $i
}

puts $sum
```

在代码示例中，我们从一系列数字计算值的总和。

while循环包含三个部分：初始化，测试和更新。 该命令的每次执行都称为一个循环。

```
set i 0
```

我们启动i变量。 它用作计数器。

```
while { $i < 10 } {
...
}
```

while命令后面大括号内的表达式是第二阶段，即测试。 执行主体中的命令，直到表达式的计算结果为false。

```
incr i
```

while循环的最后第三阶段是更新。 计数器增加。 请注意，对while循环的不正确处理可能会导致循环不断。

## for命令

如果在启动循环之前知道周期数，则可以使用for命令。 在此构造中，我们声明一个计数器变量，该变量在每次循环重复期间都会自动增加或减少值。

```
#!/usr/bin/tclsh

for {set i 0} {$i < 10} {incr i} {
    puts $i
}
```

在此示例中，我们将数字0..9打印到控制台。

```
for {set i 0} {$i < 10} {incr i} {
    puts $i
}
```

分为三个阶段。 首先，我们将计数器i初始化为零。 此阶段仅完成一次。 接下来是条件。 如果满足条件，则执行for块中的命令。 然后进入第三阶段； 计数器增加。 现在，我们重复阶段 2 和 3，直到不满足条件并留下for循环。 在我们的情况下，当计数器i等于 10 时，for循环停止执行。

```
$ ./forloop.tcl
0
1
2
3
4
5
6
7
8
9
```

在这里，我们看到forloop.tcl脚本的输出。

## foreach命令

foreach命令简化了遍历数据集合的过程。 它没有明确的计数器。 它逐个元素地遍历一个列表，并且当前值被复制到构造中定义的变量中。

```
#!/usr/bin/tclsh
```

```
set planets { Mercury Venus Earth Mars Jupiter Saturn
    Uranus Neptune }

foreach planet $planets {
    puts $planet
}
```

在此示例中，我们使用foreach命令浏览行星列表。

```
foreach planet $planets {
    puts $planet
}
```

foreach命令的用法很简单。 planets是我们迭代的列表。 planet是具有列表中当前值的临时变量。 foreach命令遍历所有行星并将其打印到控制台。

```
$ ./planets.tcl
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
```

运行上面的 Tcl 脚本将给出此输出。

```
#!/usr/bin/tclsh

set actresses { Rachel Weiss Scarlett Johansson Jessica Alba \
    Marion Cotillard Jennifer Connelly}

foreach {first second} $actresses {
    puts "$first $second"
}
```

在此脚本中，我们迭代列表的值对。

```
foreach {first second} $actresses {
    puts "$first $second"
```

```
}
```

我们在每次迭代时从列表中选择两个值。

```
$ ./actresses.tcl
Rachel Weiss
Scarlett Johansson
Jessica Alba
Marion Cotillard
Jennifer Connelly
```

这是actresses.tcl脚本的输出。

```
#!/usr/bin/tclsh

foreach i { one two three } item {car coins rocks} {
    puts "$i $item"
}
```

我们可以并行遍历两个列表。

```
$ ./parallel.tcl
one car
two coins
three rocks
```

这是parallel.tcl脚本的输出。

## break和continue命令

break命令可用于终止由while, for或switch命令定义的块。

```
#!/usr/bin/tclsh

while true {

    set r [expr 1 + round(rand()*30)]
    puts -nonewline "$r "

    if {$r == 22} { break }
}
```

```
puts ""
```

我们定义了一个无限的while循环。 我们使用break命令退出此循环。 我们从 1 到 30 中选择一个随机值并打印出来。 如果该值等于 22，则结束无穷while循环。

```
set r [expr 1 + round(rand()*30)]
```

在这里，我们计算出 1..30 之间的随机数。 rand()是内置的 Tcl 过程。 它返回一个从 0 到 0.99999 的随机数。 rand()*30返回 0 到 29.99999 之间的随机数。 round()过程将最终数字四舍五入。

```
$ ./breakcommand.tcl
28 20 8 8 12 22
```

我们可能会得到这样的东西。

continue命令用于跳过循环的一部分，并继续循环的下一个迭代。 可以与for和while命令结合使用。

在下面的示例中，我们将打印一个数字列表，这些数字不能除以 2 而没有余数。

```
#!/usr/bin/tclsh

set num 0

while { $num < 100 } {

    incr num

    if {$num % 2 == 0} { continue }

    puts "$num "
}

puts ""
```

我们使用while循环遍历数字1..99。

```
if {$num % 2 == 0} { continue }
```

如果表达式`num % 2`返回 0，则所讨论的数字可以除以 2。执行`continue`命令，并跳过循环的其余部分。 在我们的情况下，循环的最后一个命令将被跳过，并且数字不会打印到控制台。 下一个迭代开始。

在 Tcl 教程的这一部分中，我们正在讨论控制流结构。

# Tcl 中的字符串

> 原文： https://zetcode.com/lang/tcl/strings/

在 Tcl 教程的这一部分中，我们将更详细地处理字符串数据。 字符串是计算机语言中的一种重要数据类型。

字符串是字符序列。 与其他语言不同，Tcl 中的字符串不必总是用双引号引起来。 仅当单词之间有空格时，它们才是必需的。 Tcl 是基于字符串的语言。 它提供了一组丰富的用于处理字符串的命令。

## 第一个例子

下面是一个显示一些字符串的简单示例。

```
#!/usr/bin/tclsh

puts Tcl
puts Java
puts Falcon

puts "Tcl language"
puts {Tcl language}
```

该脚本将一些字符串值打印到控制台。

```
puts Tcl
puts Java
puts Falcon
```

Tcl 中的字符串不必总是用引号引起来。

```
puts "Tcl language"
puts {Tcl language}
```

Tcl 中的字符串可以用双引号或大括号括起来。

```
$ ./simple_strings.tcl
Tcl
Java
Falcon
Tcl language
```

```
Tcl language
```

这是simple.tcl脚本的输出。

## 使用引号

如果我们想在直接演讲中显示报价该怎么办？ 在这种情况下，必须对内引号进行转义。

```
$ cat directspeech.tcl
#!/usr/bin/tclsh

puts "There are many stars"
puts "He said, \"Which one is your favourite?\""
```

我们使用\字符转义其他引号。

```
$ ./directspeech.tcl
There are many stars
He said, "Which one is your favourite?"
```

directspeech.tcl程序的输出。

## 多行字符串

在 Tcl 中创建多行字符串非常容易。 在创建多行字符串的许多其他语言中，我使用起来不那么方便。

```
#!/usr/bin/tclsh

set lyrics "I cheated myself
like I knew I would
I told ya, I was trouble
you know that I'm no good"

puts $lyrics
```

我们简单地继续下一行。 如果我们想显示经文，这很有用。

```
$ ./multiline.tcl
I cheated myself
like I knew I would
I told ya, I was trouble
```

```
you know that I'm no good
```

## 比较字符串

字符串的基本比较可以通过`string compare`命令完成。

```
#!/usr/bin/tclsh

puts [string compare 12 12]
puts [string compare Eagle Eagle]
puts [string compare Eagle eagle]
puts [string compare -nocase Eagle eagle]
```

`string compare`命令逐字符比较字符串。 如果发现两个字符串的第一个字符相等，则继续第二个字符，直到结尾。 如果字符串相等，则返回 0；如果第一个字符串中的不匹配字符位于 ASCII 表中的第二个字符串之前，则返回 -1。如果第一个字符串的不匹配字符位于第二个字符串的字符之后，则返回 1。

```
puts [string compare 12 12]
```

在这种情况下，12 是字符串。

```
puts [string compare Eagle Eagle]
```

两个字符串相等，将 0 打印到控制台。

```
puts [string compare Eagle eagle]
```

`E`位于`e`之前，因此返回-1。

```
puts [string compare -nocase Eagle eagle]
```

使用`-nocase`选项，我们将忽略大小写。 这两个字符串相等。

```

```

```
$ ./compare.tcl
0
0
-1
0
```

程序的输出。

`string equal`也可以用来比较字符串。 如果字符串相等，则命令返回 1，否则返回 0。

```
#!/usr/bin/tclsh

set str1 Tcl
set str2 "Tcl language"

puts [string compare $str1 $str2]
puts [string compare -length 3 $str1 $str2]

puts [string equal $str1 $str2]
puts [string equal -length 3 $str1 $str2]
```

该脚本显示了两个比较字符串的命令。

```
puts [string compare $str1 $str2]
```

该行打印 -1。前三个位置的字符相等。 在第四个位置，`string compare`命令将空格与`l`字符进行比较。 该空格位于 ASCII 表中的`l`字符之前。 字符串不相等。

```
puts [string compare -length 3 $str1 $str2]
```

在这种情况下，我们将比较限制为前三个字符。 它们在两个字符串中都相同，因此命令返回 0。

```
puts [string equal $str1 $str2]
```

这两个字符串不相同，因此`string equal`命令返回 0，表示为`false`。

```
```

```
puts [string equal -length 3 $str1 $str2]
```

将字符串限制为前三个字符，该命令返回 1。这意味着直到前三个字符为止它们都是相同的。

## 模式匹配

对于简单的模式匹配-球形-我们可以使用`string match`命令。 对于更强大的模式匹配，我们可以利用`regexp`命令。

```
#!/usr/bin/tclsh

puts [string match book???? bookcase]

puts [regexp {[a-z]{3}} "abc"]
puts [regexp {[^a-z]{3}} "abc"]
puts [regexp book(shelf|worm) bookworm]
```

该示例演示了`string match`和`regexp`命令的用法。 对于匹配项，它们返回 1，对于不匹配项，返回 0。

```
$ ./string_match.tcl
1
1
0
1
```

`string_match1.tcl`程序的输出。

## Unicode

我们可以在 Tcl 脚本中使用 Unicode 字符串。

```
#!/usr/bin/tclsh

puts "La femme vit par le sentiment, là où l'homme vit par l'action"
puts "Анна Каренина"
```

我们将两个消息打印到终端。 第一是法语，第二是俄语。

```
$ ./unicode.tcl
La femme vit par le sentiment, là où l'homme vit par l'action
```

```
Анна Каренина
```

输出。

## 字符串命令

Tcl 具有有用的内置命令，可用于处理字符串。

```
#!/usr/bin/tclsh

set str Eagle

puts [string length $str]

puts [string index $str 0]
puts [string index $str end]

puts [string range $str 1 3]
```

我们定义一个字符串变量并使用一些字符串命令。

```
puts [string length $str]
```

`string length`返回字符串中的字符数。

```
puts [string index $str 0]
puts [string index $str end]
```

`string index`命令在特定位置返回字符。

```
puts [string range $str 1 3]
```

`string range`返回由第一个和最后一个索引选择的字符范围。

```
$ ./strings1.tcl
5
E
e
```

```
agl
```

输出。

我们有一个split命令将字符串拆分为特定字符。 该命令返回单词列表。 可以使用join命令将这些单词组合成一个字符串。

```
#!/usr/bin/tclsh

set langs "Tcl,Java,C,C#,Ruby,Falcon"

puts [split $langs ,]
puts [join [split $langs ","] ":"]
```

在我们的程序中，我们将拆分并连接字符串。

```
set langs "Tcl,Java,C,C#,Ruby,Falcon"
```

这是我们将要拆分的字符串。 有几个单词，以逗号分隔。 逗号字符是用来分割字符串的字符。

```
puts [split $langs ,]
```

该行显示了我们从字符串中拆分出的所有单词。

```
puts [join [split $langs ","] ":"]
```

split命令返回字符串中的单词列表。 然后将这些词合并在一起。 单词现在将由冒号分隔。

```
$ ./splitjoin.tcl
Tcl Java C C# Ruby Falcon
Tcl:Java:C:C#:Ruby:Falcon
```

示例的输出。

接下来，我们将使用另一个带有几个字符串命令的示例。

```
#!/usr/bin/tclsh

set str "ZetCode"

puts [string toupper $str]
puts [string tolower $str]
puts [string totitle $str]
puts [string reverse $str]
```

我们介绍了四个字符串命令。 这些命令不会更改原始字符串。 它们返回一个新的，经过修改的字符串。

```
puts [string toupper $str]
```

我们将字符转换为大写。

```
puts [string tolower $str]
```

我们将字符串的字母转换为小写。

```
puts [string totitle $str]
```

`string totitle`返回一个字符串，其首字母大写； 其他字符均为小写。

```
puts [string reverse $str]
```

我们使用`string reverse`命令反转字符串的字符。

```
$ ./strings2.tcl
ZETCODE
zetcode
Zetcode
edoCteZ
```

运行程序。

# 格式化字符串

字符串的最基本格式在引号内完成。

```
#!/usr/bin/tclsh

set oranges 2
set apples 4
set bananas 3

puts "There are $oranges oranges, $apples apples and\
$bananas bananas. "
```

Tcl 用双引号求值变量。

```
puts "There are $oranges oranges, $apples apples and\
$bananas bananas. "
```

在此代码行中，我们将变量和字符串组合在一个句子中。

```
$ ./fruit.tcl
There are 2 oranges, 4 apples, and 3 bananas.
```

输出。

可以使用format命令完成更高级的格式化。 它具有以下概要：

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

formatString用于控制参数的显示方式。 该命令可以使用多个参数。

```
#!/usr/bin/tclsh

puts [format %s "Inception movie"]
puts [format "%d %s" 23 songs]
```

这是基本脚本，显示format命令的用法。

```
puts [format %s "Inception movie"]
```

该行仅将字符串输出到控制台。

```
puts [format "%d %s" 23 songs]
```

在这里，我们打印两个参数。 每个参数都有一个格式说明符，该说明符以%字符开头。

```
$ ./basicformat.tcl
Inception movie
23 songs
```

输出。

现在，我们显示format命令的一些基本转换说明符。 %s，%f，%d和%e是转换类型。 它们控制如何显示值。 转换类型是转换说明符的唯一必需部分。

```
#!/usr/bin/tclsh

puts [format "%s" "Tcl language"]
puts [format "%f" 212.432]
puts [format "%d" 20000]
puts [format "%e" 212.342]
```

我们将在终端上打印四则消息。

```
puts [format "%s" "Tcl language"]
```

%s是字符串的转换类型。

```
puts [format "%f" 212.432]
```

%f用于显示十进制数字。

```
puts [format "%d" 20000]
```

要打印整数值，我们使用%d转换类型。

```
puts [format "%e" 212.342]
```

%e用于以科学格式显示数字。

```
$ ./format.tcl
Tcl language
212.432000
20000
2.123420e+02
```

输出。

在下一个示例中，我们将以三种不同的数字格式设置数字格式。

```
#!/usr/bin/tclsh

puts [format "%-10s %-14s %s" Decimal Hexadecimal Octal]

puts [format "%-10d %-14x %o" 5000 5000 5000]
puts [format "%-10d %-14x %o" 344 344 344]
puts [format "%-10d %-14x %o" 55 55 55]
puts [format "%-10d %-14x %o" 9 9 9]
puts [format "%-10d %-14x %o" 15666 15666 15666]
```

我们以十进制，十六进制和八进制格式打印数字。 我们还将数字对齐三列。

```
puts [format "%-10d %-14x %o" 5000 5000 5000]
```

%-10d适用于第一个数字，%-14x适用于第二个数字，%o适用于第三个数字。 我们将描述第一个。 格式说明符以%字符开头。 负号-表示如果该值短于字段宽度，则将其对齐。 其余字段填充空白。 数字 10 指定字段宽度。 最后，d字符表明数字以十进制格式显示。 x代表十六进制，o代表八进制。

```

```

```
$ ./numbers.tcl
Decimal    Hexadecimal    Octal
5000       1388           11610
344        158            530
55         37             67
9          9              11
15666      3d32           36462
```

运行示例。

最后，我们将格式化日期和时间数据。 我们使用clock format命令。

```
#!/usr/bin/tclsh

set secs [clock seconds]

puts "Short date: [clock format $secs -format %D]"
puts "Long date: [clock format $secs -format "%A, %B %d, %Y"]"
puts "Short time: [clock format $secs -format %R]"
puts "Long time: [clock format $secs -format %r]"
puts "Month: [clock format $secs -format %B]"
puts "Year: [clock format $secs -format %Y]"
```

前面的示例演示了一些常见的日期和时间格式。

```
set secs [clock seconds]
```

我们以秒为单位获取当前时间戳。 此值随后传递给clock format命令，以使人类可以读取日期和时间。

```
puts "Short date: [clock format $secs -format %D]"
```

日期格式由-format选项控制。 有几种说明符。 %D以月/日/年格式返回日期。

```
$ ./clockformat.tcl
Short date: 04/11/2011
Long date: Monday, April 11, 2011
Short time: 11:30
Long time: 11:30:30 am
Month: April
```

```
Year: 2011
```

输出。

Tcl 教程的这一部分介绍了字符串。

{% raw %}

# Tcl 列表

> 原文： https://zetcode.com/lang/tcl/lists/

在 Tcl 教程的这一部分中，我们将讨论列表。

计算机程序可以处理数据。 处理数据组是一项基本的编程操作。 在 Tcl 中，列表是基本数据结构。 它是有价物品的有序集合。 列表中的项目用空格隔开。

列表中的每个项目均由其索引标识。 列表没有固定的长度。 列表元素可以是字符串，数字，变量，文件或其他列表。 我们可以将列表嵌套到其他任何深度的列表中。

## 创建列表

我们可以通过多种方式在 Tcl 中创建列表。

```
#!/usr/bin/tclsh

set l1 { 1 2 3 }
set l2 [list one two three]
set l3 [split "1.2.3.4" .]

puts $l1
puts $l2
puts $l3
```

创建了三个列表，并将它们的元素打印到控制台。

```
set l1 { 1 2 3 }
```

创建列表的基本方法是将列表的元素放在大括号内。 列表元素由空格分隔。

```
set l2 [list one two three]
```

创建列表的另一种方法是使用`list`命令。

```
set l3 [split "1.2.3.4" .]
```

一些 Tcl 命令作为结果返回一个列表。 在上面的代码行中，split命令返回从字符串生成的数字列表。

```
$ ./createlists.tcl
 1 2 3
one two three
1 2 3 4
```

createlists.tcl脚本的输出。

## llength命令

llength命令计算列表中的元素数。

```
#!/usr/bin/tclsh

puts [llength { 1 2 3 4 }]
puts [llength {}]
puts [llength { 1 2 {3 4} }]
puts [llength { 1 2 {} 3 4 }]
```

该脚本计算四个列表的长度。

```
puts [llength { 1 2 3 4 }]
```

该列表包含四个元素，因此，有 4 个被打印到控制台。

```
puts [llength {}]
```

此列表为空； llength命令返回 0。

```
puts [llength { 1 2 {3 4} }]
```

此列表包含一个内部列表– {3 4}。内部列表占一个元素。

```
puts [llength { 1 2 {} 3 4 }]
```

空列表也计入一个元素。

```
$ ./list_length.tcl
4
0
3
5
```

list_length.tcl示例的输出。

## 检索元素

列表元素检索有三个基本命令：lindex, lrange和lassign。

```
#!/usr/bin/tclsh

set vals { 2 4 6 8 10 12 14 }

puts [lindex $vals 0]
puts [lindex $vals 3]
puts [lindex $vals end]
puts [lindex $vals end-2]
```

该代码示例使用lindex命令从指定索引处的列表中检索元素。

```
puts [lindex $vals 0]
puts [lindex $vals 3]
```

Tcl 列表索引从 0 开始。上述命令在位置 1 和 4 处打印列表的元素。

```
puts [lindex $vals end]
puts [lindex $vals end-2]
```

end字符串表示最后一个元素的索引。 也可以从中减去一个整数。

```
$ ./retrieving.tcl
2
8
14
```

```
10
```

这是retrieving.tcl脚本的输出。

下一个代码示例说明lrange和lassign命令。

```
#!/usr/bin/tclsh

puts [lrange { a b c d e } 2 4]
puts [lrange { a b c d e } 1 end]

lassign { a b c } x y z
puts "$x $y $z"
```

lrange命令返回由两个索引指定的列表的一部分。 lassign命令将列表中的值分配给指定的变量。

```
puts [lrange { a b c d e } 2 4]
puts [lrange { a b c d e } 1 end]
```

在这里，我们打印列表的两个子列表。

```
lassign { a b c } x y z
puts "$x $y $z"
```

使用lassign命令，我们将列表元素分配给三个变量。

```
$ ./retrieving2.tcl
c d e
b c d e
a b c
```

这是retrieving2.tcl脚本的输出。

## 遍历列表

现在我们已经定义了列表和基本的列表操作，我们想遍历列表元素。 我们展示了几种浏览列表项的方法。

```
#!/usr/bin/tclsh

foreach item {1 2 3 4 5 6 7 8 9} {

    puts $item
}
```

我们使用foreach命令浏览列表元素。 item变量的每个循环周期都有数字列表中的下一个值。

```
$ ./traverse1.tcl
1
2
3
4
5
6
7
8
9
```

示例的输出。

在第二个示例中，我们将使用while循环查看日期名称。

```
#!/usr/bin/tclsh

set days [list Monday Tuesday Wednesday Thursday \
    Friday Saturday Sunday]
set n [llength $days]

set i 0

while {$i < $n} {

    puts [lindex $days $i]
    incr i
}
```

我们使用while循环遍历列表。 在使用while循环时，我们还需要一个计数器和列表中的项目数。

```
set days [list Monday Tuesday Wednesday Thursday \
    Friday Saturday Sunday]
```

list命令用于创建日期列表。

```
set n [llength $days]
```

列表的长度由llength命令确定。

```
set i 0
```

这是一个柜台。

```
while {$i < $n} {

    puts [lindex $days $i]
    incr i
}
```

while循环执行主体中的命令，直到计数器等于列表中的元素数为止。

```
puts [lindex $days $i]
```

lindex从计数器指向的列表中返回一个值。

```
incr i
```

计数器增加。

```
$ ./traverse2.tcl
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

示例的输出。

## lmap命令

可以使用lmap命令浏览列表中的元素。 这是一个函数式命令。 lmap命令遍历一个或多个列表中的所有元素并收集结果。

```
#!/usr/bin/tclsh

set vals { 1 2 3 4 5 6 }

puts [lmap a $vals {expr {$a ** 2}}]
```

该示例将lmap应用于整数列表。

```
puts [lmap a $vals {expr {$a ** 2}}]
```

函数式的lmap命令将表达式在其主体中应用于vals列表的每个元素。 返回包含新的平方整数列表的结果。

```
$ ./lmap_cmd.tcl
1 4 9 16 25 36
```

示例的输出。

## 插入元素

下一个示例将元素插入 Tcl 列表。 lappend命令将元素添加到列表的末尾； 它修改了原始列表。 linsert命令在指定的索引处插入元素； 它不会修改原始列表，但会返回一个新列表。

```
#!/usr/bin/tclsh

set nums {4 5 6}
puts $nums

lappend nums 7 8 9
puts $nums

puts [linsert $nums 0 1 2 3]
puts $nums
```

我们列出了三个数字。

```
lappend nums 7 8 9
```

lappend将数据附加到列表。 原始列表已更改。

```
puts [linsert $nums 0 1 2 3]
```

linsert在给定索引处插入元素。 第一个数字是索引。 其余值是要插入列表中的数字。 该命令创建一个新列表并返回它； 它不会修改原始列表。

```
$ ./inserting.tcl
4 5 6
4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
4 5 6 7 8 9
```

这是operations.tcl脚本的输出。

在下面的示例中，我们将连接列表，搜索项目并替换列表中的项目。

```
#!/usr/bin/tclsh

set animals1 { lion eagle elephant dog cat }
set animals2 { giraffe tiger horse dolphin }

set animals [concat $animals1 $animals2]

puts $animals

puts [lsearch -exact $animals eagle]
puts [lreplace $animals 3 4 buffalo crocodile]
```

我们定义了两个动物列表。 我们介绍了三个新命令。

```
set animals [concat $animals1 $animals2]
```

concat命令用于连接（添加）两个列表。 上一行连接了两个列表，新列表设置为animal变量。

```
puts [lsearch -exact $animals eagle]
```

使用lsearch命令，我们在列表中寻找鹰。 使用-exact选项，我们可以寻找完全匹配的内容。 该命令返回第一个匹配元素的索引，如果没有匹配，则返回 -1。

```
puts [lreplace $animals 3 4 buffalo crocodile]
```

lreplace命令用水牛和鳄鱼代替了狗和猫。

```
$ ./operations2.tcl
lion eagle elephant dog cat giraffe tiger horse dolphin
1
lion eagle elephant buffalo crocodile giraffe tiger horse dolphin
```

这是示例输出。

## 排序项目

在本节中，我们将展示如何对 Tcl 列表中的项目进行排序。

```
#!/usr/bin/tclsh

set names { John Mary Lenka Veronika Julia Robert }
set nums { 1 5 4 3 6 7 9 2 11 0 8 2 3 }

puts [lsort $names]
puts [lsort -ascii $names]
puts [lsort -ascii -decreasing $names]
puts [lsort -integer -increasing $nums]
puts [lsort -integer -decreasing $nums]
puts [lsort -integer -unique $nums]
```

要对列表元素进行排序，我们可以使用sort命令。 该命令不会修改原始列表。 它返回一个新的元素排序列表。

```
set names { John Mary Lenka Veronika Julia Robert }
set nums { 1 5 4 3 6 7 9 2 11 0 8 2 3 }
```

我们有两个列表。 在第一个数字中，我们有字符串，在第二个数字中。

```
puts [lsort $names]
puts [lsort -ascii $names]
```

默认排序是 ASCII 排序。 元素按其在 ASCII 表中的位置排序。

```
puts [lsort -integer -increasing $nums]
puts [lsort -integer -decreasing $nums]
```

我们将值视为整数，然后按升序和降序对其进行排序。

```
puts [lsort -integer -unique $nums]
```

我们在数字上下文中以递增顺序对列表中的元素进行排序。 重复项将被删除。

```
$ ./sorting.tcl
John Julia Lenka Mary Robert Veronika
John Julia Lenka Mary Robert Veronika
Veronika Robert Mary Lenka Julia John
0 1 2 2 3 3 4 5 6 7 8 9 11
11 9 8 7 6 5 4 3 3 2 2 1 0
0 1 2 3 4 5 6 7 8 9 11
```

这是`sorting.tcl`脚本的输出。

## 嵌套列表

在 Tcl 中可以有嵌套列表-其他列表中的列表。

```
#!/usr/bin/tclsh

set nums {1 2 {1 2 3 4} {{1 2} {3 4}} 3 4}

puts [llength $nums]
puts [llength [lindex $nums 2]]

puts [lindex $nums 0]
puts [lindex [lindex $nums 2] 1]
```

```
puts [lindex [lindex [lindex $nums 3] 1] 1]
```

这是一个在 Tcl 中具有嵌套列表的简单示例。

```
set nums {1 2 {1 2 3 4} {{1 2} {3 4}} 3 4}
```

nums是带有两个嵌套列表的列表。 第二个嵌套列表还有两个附加的内部嵌套列表。

```
puts [llength $nums]
```

我们确定列表的大小。 嵌套列表被视为一个元素。

```
puts [llength [lindex $nums 2]]
```

在此行中，我们确定第一个嵌套列表的大小，这是主列表的第三个元素。

```
puts [lindex $nums 0]
```

在这里，我们打印主列表的第一个元素。

```
puts [lindex [lindex $nums 2] 1]
```

在上一行中，我们获得了第一个嵌套列表的第二个元素。

```
puts [lindex [lindex [lindex $nums 3] 1] 1]
```

在这里，我们获得内部列表的第二个内部列表的第二个元素，它们位于主列表的第 4 个位置。 换句话说：最里面的命令首先执行。 `[lindex $nums 3]`返回`{{1 2} {3 4}}`。现在，第二个命令在此返回列表上运行。`[lindex {{1 2} {3 4}} 1]`表达式返回`{3 4}`。最后，最后一条命令`[lindex {3 4} 1]`返回 4，该命令被打印到终端上。

```
```

```
$ ./nestedlists.tcl
6
4
1
2
4
```

示例的输出。

可以使用更简单的语法来检索嵌套列表的元素。

```
#!/usr/bin/tclsh

set nums { 1 2 {1 2 3 {4 5}} 3 4 }

puts [lindex $nums 0]
puts [lindex $nums 2 1]
puts [lindex $nums 2 3 1]
```

索引跟随`lindex`命令的第一个参数，从最外面的列表的索引开始。

```
$ ./nestedlists2.tcl
1
2
5
```

示例的输出。

在 Tcl 教程的这一部分中，我们介绍了 Tcl 列表。

{% endraw %}

# Tcl 中的数组

> 原文： https://zetcode.com/lang/tcl/arrays/

在 Tcl 编程教程的这一部分中，我们将介绍数组。 我们将初始化数组并从中读取数据。

Tcl 数组是变量的集合。 每个变量可以保存任何值，并且该数组由任意值索引。 键值对是无序的。 Tcl 数组是关联数组。

## 创建数组

可以使用`set`或`array set`命令创建 Tcl 数组。

```
#!/usr/bin/tclsh

set names(1) Jane
set names(2) Tom
set names(3) Elisabeth
set names(4) Robert
set names(5) Julia
set names(6) Victoria

puts [array exists names]
puts [array size names]

puts $names(1)
puts $names(2)
puts $names(6)
```

我们创建一个名为`names`的数组。 数字是键，名称是数组的值。

```
set names(1) Jane
```

在这一行中，我们为数组键 1 设置了一个值`Jane`。我们以后可以通过键引用该值。

```
puts [array exists names]
```

`array exists`命令确定提供的参数是否为数组。

```
puts [array size names]
```

我们使用array size命令获得数组的大小。

```
puts $names(1)
```

我们通过键访问数组中的一个值。

```
$ ./names.tcl
1
6
Jane
Tom
Victoria
```

示例的输出。

在第二个示例中，使用array set命令创建一个数组。

```
#!/usr/bin/tclsh

array set days {
    1 Monday
    2 Tuesday
    3 Wednesday
    4 Thursday
    5 Friday
    6 Saturday
    7 Sunday
}

set n [array size days]

puts $days(1)
puts "array has $n elements"
```

我们创建一个days数组。 它具有 7 个键值对。

```
$ ./days.tcl
Monday
array has 7 elements
```

示例输出。

## 数组是变量的集合

与列表或字典不同，数组中的项目是变量。 这意味着我们可以引用它们。

```
#!/usr/bin/tclsh

array set days {
    1 Monday
    2 Tuesday
    3 Wednesday
    4 Thursday
    5 Friday
    6 Saturday
    7 Sunday
}

upvar #0 days(1) mon
upvar #0 days(2) tue
upvar #0 days(3) wed

puts $mon
puts $tue
puts $wed
```

在脚本中，upvar命令引用了days数组的三个变量。

```
upvar #0 days(1) mon
```

mon变量引用以 1 索引的变量。upvar命令的第一个参数是上层，其中#0表示顶层。 也就是说，days数组和mon变量都位于同一全局名称空间中。

```
puts $mon
```

在这里，我们通过mon变量引用days数组的项。

```
$ ./colvar.tcl
Monday
Tuesday
Wednesday
```

示例输出。

## 数组获取命令

`array get`命令返回一个包含数组元素对的列表。

```
#!/usr/bin/tclsh

array set days {
    Peter 34
    Jane 17
    Lucy 28
    Mark 43
    Anthony 36
}

puts [array get days]
```

该示例创建一个数组，并使用`array get`命令打印其键值对。

```
$ ./arrayget.tcl
Peter 34 Anthony 36 Lucy 28 Jane 17 Mark 43
```

这是`arrayget.tcl`脚本的输出。

## 遍历数组

在以下示例中，我们将展示如何遍历数组。

```
#!/usr/bin/tclsh

array set days {
    1 Monday
    2 Tuesday
    3 Wednesday
    4 Thursday
    5 Friday
    6 Saturday
    7 Sunday
}

foreach {n day} [array get days] {

    puts "$n -> $day"
```

```
}
```

该示例创建一个数组，并使用array get命令打印其键值对。

```
foreach {n day} [array get days] {
```

array get命令返回键，值元素的列表，可以使用foreach命令对其进行迭代。

```
$ ./days2.tcl
4 -> Thursday
5 -> Friday
1 -> Monday
6 -> Saturday
2 -> Tuesday
7 -> Sunday
3 -> Wednesday
```

这是days2.tcl脚本的输出。 请注意，成对的元素没有顺序。

以下脚本使用array names命令遍历数组。

```
#!/usr/bin/tclsh

array set nums { a 1 b 2 c 3 d 4 e 5 }

puts [array names nums]

foreach n [array names nums] {

    puts $nums($n)
}
```

我们创建一个简单的nums数组并循环遍历。

```
array set nums { a 1 b 2 c 3 d 4 e 5 }
```

我们定义一个简单的数组。

```
puts [array names nums]
```

`array names`返回一个列表，其中包含数组中所有元素的名称（键）。

```
foreach n [array names nums] {

    puts $nums($n)
}
```

我们使用键来获取值。

```
$ ./getnames.tcl
d e a b c
4
5
1
2
3
```

`getnames.tcl`脚本的输出。

前面的示例使用数组的副本，因此不太适合处理大型数组。 数组搜索工具效率更高。

```
#!/usr/bin/tclsh

array set days {
    1 Monday
    2 Tuesday
    3 Wednesday
    4 Thursday
    5 Friday
    6 Saturday
    7 Sunday
}

set start [array startsearch days]

while {[array anymore days $start]} {
    set key [array nextelement days $start]

    puts $days($key)
}
```

```
array donesearch days $start
```

我们使用数组搜索命令来迭代一个简单的数组。

```
set start [array startsearch days]
```

`array startsearch`命令引用数组的开头。

```
while {[array anymore days $start]} {
```

如果在数组搜索中还有剩余要处理的元素，则`array anymore`命令返回 1。

```
set key [array nextelement days $start]
```

`array nextelement`命令返回数组中下一个元素的名称。

```
array donesearch days $start
```

`array donesearch`命令终止数组搜索并破坏与该搜索关联的所有状态。

## 移除元素

在本章的最后一个示例中，我们将展示如何从数组中删除元素。

```
#!/usr/bin/tclsh

set names(1) Jane
set names(2) Tom
set names(3) Elisabeth
set names(4) Robert
set names(5) Julia
set names(6) Victoria

puts [array size names]
unset names(1)
unset names(2)
```

```
puts [array size names]
```

我们创建一个names数组。 我们使用unset命令从数组中删除项目。 我们在删除两项之前和之后检查数组的大小。

```
set names(1) Jane
```

set命令用于在数组中创建一个项目。

```
unset names(1)
```

我们使用unset命令从数组中删除键为 1 的元素。

```
$ ./removing.tcl
6
4
```

一开始，数组中有 6 个元素。 删除两个元素后，剩下 4 个元素。

在 Tcl 教程的这一部分中，我们使用了 Tcl 数组。

# Tcl 中的过程

> 原文： https://zetcode.com/lang/tcl/procedures/

在本教程的这一部分中，我们将介绍 Tcl 过程。

过程是包含一系列命令的代码块。 在许多编程语言中，过程都称为函数。 对于程序仅执行一个特定任务，这是一种良好的编程习惯。 程序为程序带来了模块化。 正确使用程序会带来以下优点：

- 减少代码重复
- 将复杂的问题分解成更简单的部分
- 提高代码的清晰度
- 重用代码
- 信息隐藏

过程有两种基本类型：内置过程和用户定义的过程。 内置过程是 Tcl 核心语言的一部分。 例如，`rand()`, `sin()`和`exp()`是内置程序。 用户定义的过程是使用`proc`关键字创建的过程。

`proc`关键字用于创建新的 Tcl 命令。 术语过程和命令通常可以互换使用。

我们从一个简单的例子开始。

```
#!/usr/bin/tclsh

proc tclver {} {

    set v [info tclversion]
    puts "This is Tcl version $v"
}

tclver
```

在此脚本中，我们创建一个简单的`tclver`过程。 该过程将打印 Tcl 语言的版本。

```
proc tclver {} {
```

使用`proc`命令创建新过程。 `{}`字符表明该过程没有参数。

```
    {

    set v [info tclversion]
    puts "This is Tcl version $v"
}
```

这是tclver过程的主体。 它在我们执行tclver命令时执行。 该命令的正文位于大括号之间。

```
tclver
```

通过指定其名称来调用该过程。

```
$ ./version.tcl
This is Tcl version 8.6
```

样本输出。

## 过程参数

参数是传递给过程的值。 过程可以采用一个或多个参数。 如果过程使用数据，则必须将数据传递给过程。

在下面的示例中，我们有一个采用一个参数的过程。

```
#!/usr/bin/tclsh

proc ftc {f} {

    return [expr $f * 9 / 5 + 32]
}

puts [ftc 100]
puts [ftc 0]
puts [ftc 30]
```

我们创建一个ftc程序，将华氏温度转换为摄氏温度。

```
proc ftc {f} {
```

该过程采用一个参数。 该程序的主体将使用其名称f。

```
return [expr $f * 9 / 5 + 32]
```

我们计算摄氏温度值。 return命令将值返回给调用方。 如果该过程未执行显式返回，则其返回值是在过程主体中执行的最后一条命令的值。

```
puts [ftc 100]
```

执行ftc过程。 它以 100 为参数。 这是华氏温度。 返回的值由puts命令使用，该命令将其打印到控制台。

```
$ ./fahrenheit.tcl
212
32
86
```

示例的输出。

接下来，我们将有一个接受两个参数的过程。

```
#!/usr/bin/tclsh

proc maximum {x y} {

    if {$x > $y} {

        return $x
    } else {

        return $y
    }
}

set a 23
set b 32

set val [maximum $a $b]
puts "The max of $a, $b is $val"
```

maximum过程将两个值的最大值相减。

```
proc maximum {x y} {
```

该方法有两个参数。

```
if {$x > $y} {

    return $x
} else {

    return $y
}
```

在这里，我们计算哪个更大。

```
set a 23
set b 32
```

我们定义了两个要比较的变量。

```
set val [maximum $a $b]
```

我们计算两个变量的最大值。

```
$ ./maximum.tcl
The max of 23, 32 is 32
```

这是maximum.tcl脚本的输出。

## 可变数量的参数

一个过程可以接受和处理可变数量的参数。 为此，我们使用特殊的args参数。

```
#!/usr/bin/tclsh

proc sum {args} {

    set s 0

    foreach arg $args {

        incr s $arg
    }

    return $s
```

```
}

puts [sum 1 2 3 4]
puts [sum 1 2]
puts [sum 4]
```

我们定义一个sum过程，将其所有参数加起来。

```
proc sum {args} {
```

sum过程具有一个特殊的args参数。 它具有传递给该过程的所有值的列表。

```
foreach arg $args {

    incr s $arg
}
```

我们遍历列表并计算总和。

```
puts [sum 1 2 3 4]
puts [sum 1 2]
puts [sum 4]
```

我们调用sum过程 3 次。 在第一种情况下，它采用 4 个参数，在第二种情况下为 2，在最后一种情况下为 4。

```
$ ./variable.tcl
10
3
4
```

variable.tcl脚本的输出。

## 隐式参数

Tcl 过程中的参数可能具有隐式值。 如果未提供显式值，则使用隐式值。

```
#!/usr/bin/tclsh
```

```
proc power {a {b 2}} {

    if {$b == 2} {
        return [expr $a * $a]
    }

    set value 1

    for {set i 0} {$i<$b} {incr i} {

        set value [expr $value * $a]
    }

    return $value
}

set v1 [power 5]
set v2 [power 5 4]

puts "5^2 is $v1"
puts "5^4 is $v2"
```

在这里，我们创建一个power过程。 该过程具有一个带有隐式值的参数。 我们可以使用一个和两个参数来调用该过程。

```
proc power {a {b 2}} {
```

第二个参数b具有隐式值 2。如果仅提供一个参数，则power过程会将a的值返回到幂 2。

```
set v1 [power 5]
set v2 [power 5 4]
```

我们用一个和两个参数调用幂过程。 第一行计算 5 的幂 2。第二行计算 5 的幂 4。

```
$ ./implicit.tcl
5^2 is 25
5^4 is 625
```

示例的输出。

## 返回多个值

return命令将一个值传递给调用方。 通常需要返回多个值。 在这种情况下，我们可以返回一个列表。

```
#!/usr/bin/tclsh

proc tworandoms {} {

    set r1 [expr round(rand()*10)]
    set r2 [expr round(rand()*10)]

    return [list $r1 $r2]
}

puts [tworandoms]
puts [tworandoms]
puts [tworandoms]
puts [tworandoms]
```

我们有一个tworandoms程序。 它返回 1 到 10 之间的两个随机整数。

```
set r1 [expr round(rand()*10)]
```

计算一个随机整数并将其设置为r1变量。

```
return [list $r1 $r2]
```

借助list命令返回两个值。

```
$ ./tworandoms.tcl
3 7
1 3
8 7
9 9
```

样本输出。

## 递归

在数学和计算机科学中，递归是一种定义函数的方法，其中所定义的函数在其自己的定义内应用。 换句话说，递归函数调用自身以完成其工作。 递归是解决许多编程任务的一种广泛使用的方法。 递归是诸如 Scheme, OCalm 或 Clojure 之类的函数式语言的基本方法。

递归调用在 Tcl 中有一个限制。 递归调用不能超过 1000 个。

递归的典型示例是阶乘的计算。 阶乘n!是所有小于或等于 n 的正整数的乘积。

```
#!/usr/bin/tclsh

proc factorial n {

    if {$n==0} {

        return 1
    } else {

        return [expr $n * [factorial [expr $n - 1]]]
    }
}

# Stack limit between 800 and 1000 levels

puts [factorial 4]
puts [factorial 10]
puts [factorial 18]
```

在此代码示例中，我们计算三个数字的阶乘。

```
return [expr $n * [factorial [expr $n - 1]]]
```

在factorial过程的主体内部，我们使用修改后的参数调用factorial过程。 该过程将自行调用。

```
$ ./recursion.tcl
24
3628800
6402373705728000
```

这些就是结果。 如果我们尝试计算 100 的阶乘，则会收到“嵌套求值过多”的错误。

## 作用域

在过程内部声明的变量具有过程作用域。 名称的作用域是程序文本的区域，在该区域内可以引用名称声明的实体而无需对该名称进行限定。 在过程内部声明的变量具有过程作用域； 它也称为本地作用域。 该变量仅在此特定过程中有效。

```
#!/usr/bin/tclsh

proc test {} {

    puts "inside procedure"
    #puts "x is $x"
    set x 4
    puts "x is $x"
}

set x 1

puts "outside procedure"
puts "x is $x"

test

puts "outside procedure"
puts "x is $x"
```

在前面的示例中，我们在测试过程的内部和外部定义了一个x变量。

```
set x 4
puts "x is $x"
```

在测试过程中，我们定义了x变量。 该变量具有局部作用域，仅在此过程内有效。

```
set x 1

puts "outside procedure"
puts "x is $x"
```

我们在过程外部定义x变量。 它具有全球作用域。 变量没有冲突，因为它们具有不同的作用域。

```
$ ./scope.tcl
outside procedure
x is 1
inside procedure
x is 4
outside procedure
x is 1
```

示例的输出。

可以在过程内部更改全局变量。

```
#!/usr/bin/tclsh

proc test {} {

    upvar x y
    puts "inside procedure"
    puts "y is $y"
    set y 4
    puts "y is $y"
}

set x 1

puts "outside procedure"
puts "x is $x"

test

puts "outside procedure"
puts "x is $x"
```

我们定义一个全局x变量。 我们在测试过程中更改变量。

```
upvar x y
```

我们使用upvar命令通过名称y引用全局x变量。

```
set y 4
```

我们为本地y变量分配一个值，并更改全局x变量的值。

```
$ ./scope2.tcl
outside procedure
x is 1
inside procedure
y is 1
y is 4
outside procedure
```

```
x is 4
```

从输出中我们可以看到测试过程更改了x变量。

使用global命令，我们可以从过程中引用全局变量。

```
#!/usr/bin/tclsh

proc test {} {

    global x
    puts "inside test procedure x is $x"

    proc nested {} {
        global x
        puts "inside nested x is $x"
    }
}

set x 1

test
nested

puts "outside x is $x"
```

在上面的示例中，我们有一个测试过程以及在该测试过程中定义的嵌套过程。 我们从这两个过程中都引用了全局x变量。

```
global x
puts "inside test procedure x is $x"
```

使用global命令，可以引用在测试过程之外定义的全局x变量。

```
proc nested {} {
    global x
    puts "inside nested x is $x"
}
```

可以创建嵌套过程。 这些是在其他过程中定义的过程。 我们使用global命令引用全局x变量。

```
test
nested
```

我们称之为测试过程及其嵌套过程。

```
$ ./scope3.tcl
inside test procedure x is 1
inside nested x is 1
outside x is 1
```

示例的输出。

在 Tcl 教程的这一部分中，我们介绍了 Tcl 过程。

# 输入&输出

> 原文： https://zetcode.com/lang/tcl/io/

在本章中，我们将使用 Tcl 中的输入和输出操作。 Tcl 有几个执行 io 的命令。 我们将介绍其中的一些。

Tcl 使用称为通道的对象读取和写入数据。 可以使用open或socket命令创建通道。 Tcl 脚本可使用三个标准通道，而无需显式创建它们。 操作系统会为每个新应用自动打开它们。 它们是stdin，stdout和stderr。脚本使用标准输入stdin来读取数据。 脚本使用标准输出stdout写入数据。 脚本使用标准错误stderr来编写错误消息。

在第一个示例中，我们将使用puts命令。 它具有以下概要：

```
puts ?-nonewline? ?channelId? string
```

channelId是我们要在其中写入文本的通道。 channelId是可选的。 如果未指定，则采用默认值stdout。

```
#!/usr/bin/tclsh

puts "Message 1"
puts stdout "Message 2"
puts stderr "Message 3"
```

puts命令将文本写入通道。

```
puts "Message 1"
```

如果未指定channelId，则默认情况下写入stdout。

```
puts stdout "Message 2"
```

此行与上一行相同。 我们仅明确指定了channelId。

```
puts stderr "Message 3"
```

我们写入标准错误通道。 错误消息默认情况下转到终端。

```
$ ./printing.tcl
Message 1
Message 2
Message 3
```

示例输出。

## read命令

read命令用于从通道读取数据。 可选参数指定要读取的字符数。 如果省略，该命令将从通道读取所有数据，直到最后。

```
#!/usr/bin/tclsh

set c [read stdin 1]

while {$c != "q"} {

    puts -nonewline "$c"
    set c [read stdin 1]
}
```

该脚本从标准输入通道读取一个字符，然后将其写入标准输出，直到遇到q字符为止。

```
set c [read stdin 1]
```

我们从标准输入通道（ stdin ）读取一个字符。

```
while {$c != "q"} {
```

我们继续读取字符，直到按q为止。

## gets命令

gets命令从通道读取下一行，返回行中直到（但不包括）行尾字符的所有内容。

```
#!/usr/bin/tclsh

puts -nonewline "Enter your name: "
flush stdout
set name [gets stdin]

puts "Hello $name"
```

该脚本要求用户输入，然后打印一条消息。

```
puts -nonewline "Enter your name: "
```

puts命令用于将消息打印到终端。 -nonewline选项禁止换行符。

```
flush stdout
```

Tcl 缓冲区在内部输出，因此用puts编写的字符可能不会立即出现在输出文件或设备上。 flush命令强制输出立即显示。

```
set name [gets stdin]
```

gets命令从通道读取一条线。

```
$ ./hello.tcl
Enter your name: Jan
Hello Jan
```

脚本的示例输出。

## pwd和cd命令

Tcl 具有pwd和cd命令，类似于 shell 命令。 pwd命令返回当前工作目录，cd命令用于更改工作目录。

```
#!/usr/bin/tclsh

set dir [pwd]
```

```
puts $dir

cd ..

set dir [pwd]
puts $dir
```

在此脚本中，我们将打印当前工作目录。 然后，我们更改工作目录并再次打印工作目录。

```
set dir [pwd]
```

pwd命令返回当前工作目录。

```
cd ..
```

我们将工作目录更改为当前目录的父目录。 我们使用cd命令。

```
$ ./cwd.tcl
/home/janbodnar/prog/tcl/io
/home/janbodnar/prog/tcl
```

样本输出。

## glob命令

Tcl 具有glob命令，该命令返回与模式匹配的文件的名称。

```
#!/usr/bin/tclsh

set files [glob *.tcl]

foreach file $files {

    puts $file
}
```

该脚本将所有带有.tcl扩展名的文件打印到控制台。

```
set files [glob *.tcl]
```

glob命令返回与*.tcl模式匹配的文件列表。

```
foreach file $files {

    puts $file
}
```

我们浏览文件列表，并将列表的每个项目打印到控制台。

```
$ ./globcmd.tcl
attributes.tcl
allfiles.tcl
printing.tcl
hello.tcl
read.tcl
files.tcl
globcmd.tcl
write2file.tcl
cwd.tcl
readfile.tcl
isfile.tcl
addnumbers.tcl
```

这是globcmd.tcl脚本的示例输出。

## 处理文件

file命令操纵文件名和属性。 它有很多选择。

```
#!/usr/bin/tclsh

puts [file volumes]
[file mkdir new]
```

该脚本将打印系统的已安装卷，并创建一个新目录。

```
puts [file volumes]
```

`file volumes`命令返回到系统上安装的卷的绝对路径。

```
[file mkdir new]
```

`file mkdir`创建一个名为`new`的目录。

```
$ ./voldir.tcl
/
$ ls -d */
doc/  new/  tmp/
```

在 Linux 系统上，有一个已安装的卷-根目录。 `ls`命令确认`new`目录的创建。

在下面的代码示例中，我们将检查文件名是常规文件还是目录。

```
#!/usr/bin/tclsh

set files [glob *]

foreach fl $files {

    if {[file isfile $fl]} {

        puts "$fl is a file"
    } elseif { [file isdirectory $fl]} {

        puts "$fl is a directory"
    }
}
```

我们遍历当前工作目录中的所有文件名，并打印它是文件还是目录。

```
set files [glob *]
```

使用`glob`命令，我们创建当前目录的文件和目录名称的列表。

```
if {[file isfile $fl]} {
```

如果有问题的文件名是文件，我们将执行if命令的主体。

```
} elseif { [file isdirectory $fl]} {
```

file isdirectory命令确定文件名是否为目录。 请注意，在 Unix 上，目录是文件的特例。

puts命令可用于写入文件。

```
#!/usr/bin/tclsh

set fp [open days w]

set days {Monday Tuesday Wednesday Thursday Friday Saturday Sunday}

puts $fp $days
close $fp
```

在脚本中，我们打开一个文件进行写入。 我们将一周中的几天写入文件。

```
set fp [open days w]
```

我们打开一个名为days的文件进行写入。 open命令返回一个通道 ID。

```
set days {Monday Tuesday Wednesday Thursday Friday Saturday Sunday}
```

该数据将被写入文件。

```
puts $fp $days
```

我们使用open命令返回的通道 ID 写入文件。

```
close $fp
```

我们关闭打开的频道。

```
$ ./write2file.tcl
$ cat days
Monday Tuesday Wednesday Thursday Friday Saturday Sunday
```

我们运行脚本并检查days文件的内容。

在下面的脚本中，我们将从文件中读取数据。

```
$ cat languages
Python
Tcl
Visual Basic
Perl
Java
C
C#
Ruby
Scheme
```

我们在目录中有一个简单的文件，称为语言。

```
#!/usr/bin/tclsh

set fp [open languages r]
set data [read $fp]

puts -nonewline $data

close $fp
```

我们从提供的文件中读取数据，读取其内容，然后将数据打印到终端。

```
set fp [open languages r]
```

我们通过以只读模式打开语言文件来创建频道。

```
set data [read $fp]
```

如果我们没有为read命令提供第二个参数，它将从文件中读取所有数据，直到文件结尾。

```
puts -nonewline $data
```

我们将数据打印到控制台。

```
$ ./readfile.tcl
Python
Tcl
Visual Basic
Perl
Java
C
C#
Ruby
Scheme
```

readfile.tcl脚本的示例运行。

eof命令检查所提供通道的行尾。

```
#!/usr/bin/tclsh

set fp [open languages]

while {![eof $fp]} {
    puts [gets $fp]
}

close $fp
```

我们使用eof命令读取文件的内容。

```
while {![eof $fp]} {
    puts [gets $fp]
}
```

循环继续进行，直到eof遇到文件结尾都返回true为止。 在体内，我们使用gets命令从文件中读取一行。

```
$ ./readfile2.tcl
Python
Tcl
Visual Basic
Perl
Java
C
C#
Ruby
Scheme
```

`readfile2.tcl`脚本的示例运行。

下一个脚本将执行一些其他文件操作。

```
#!/usr/bin/tclsh

set fp [open newfile w]

puts $fp "this is new file"
flush $fp

file copy newfile newfile2
file delete newfile

close $fp
```

我们打开一个文件，并在其中写入一些文本。 文件被复制。 然后删除原始文件。

```
file copy newfile newfile2
```

`file copy`命令复制文件。

```
file delete newfile
```

原始文件用`file delete`命令删除。

在最后一个示例中，我们将使用文件属性。

```

```

```
#!/usr/bin/tclsh

set files [glob *]

set mx 0

foreach fl $files {

    set len [string length $fl]

    if { $len > $mx} {

        set mx $len
    }
}

set fstr "%-$mx\s %-s"
puts [format $fstr Name Size]

set fstr "%-$mx\s %d bytes"
foreach fl $files {

    set size [file size $fl]

    puts [format $fstr $fl $size]

}
```

该脚本将创建两列。 在第一列中，我们有文件的名称。 在第二列中，我们显示文件的大小。

```
foreach fl $files {

    set len [string length $fl]

    if { $len > $mx} {

        set mx $len
    }
}
```

在此循环中，我们找出了最长的文件名。 格式化输出列时将使用此格式。

```
set fstr "%-$mx\s %-s"
puts [format $fstr Name Size]
```

在这里，我们打印列的标题。 要格式化数据，我们使用format命令。

```
set fstr "%-$mx\s %d bytes"
foreach fl $files {

    set size [file size $fl]

    puts [format $fstr $fl $size]

}
```

我们遍历文件列表并打印每个文件名及其大小。 file size命令确定文件的大小。

```
$ ./attributes.tcl
Name           Size
attributes.tcl 337 bytes
newfile2       17 bytes
allfiles.tcl   75 bytes
printing.tcl   83 bytes
languages      51 bytes
hello.tcl      109 bytes
days           57 bytes
read.tcl       113 bytes
files.tcl      140 bytes
globcmd.tcl    82 bytes
write2file.tcl 134 bytes
doc            4096 bytes
cwd.tcl        76 bytes
tmp            4096 bytes
readfile.tcl   98 bytes
isfile.tcl     219 bytes
```

样品运行。

在本章中，我们介绍了 Tcl 中的输入/输出操作。