嵌入式技术

脚本语言程序设计

邢超

西北工业大学航天学院

脚本语言发展

- Shell
    - Bash
    - Ksh
- 快速开发
    - Tcl
    - VB
- 高阶编程
    - Lua
    - Guile

脚本语言编程

- 扩展
  - 速度
  - 系统调用
- 嵌入
  - 灵活
  - 方便

Shell

- Bourne shell (sh)
- Korn Shell (ksh)
- C Shell (csh)
- Bourne-Again SHell (bash)
- zsh

csh

- 由 Bill Joy 所写
- 语法和 C 语言的很相似

ksh

- Dave Korn 所写
- 集合了 C shell 和 Bourne shell 的优点
- 和 Bourne shell 兼容

Bash

- Bourne shell（sh）的一个双关语（Bourne again / born again）
- Stephen Bourne 在 1978 年前后编写 Bourne shell，并同 Version 7 Unix 一起发布。
- Bash 则在 1987 年由 Brian Fox 创造
- 在 1990 年，Chet Ramey 成为了主要的维护者
- POSTIX 2 shell specifications

Bash's Configuration Files

default: /etc/profile
home directory:

- .bash_profile: read and the commands in it executed by Bash every time you log in to the system
- .bashrc: read and executed by Bash every time you start a subshell
- .bash_logout: read and executed by Bash every time a login shell exits

Hello world in bash

```
#!/bin/bash
STR="Hello␣World!"
echo $STR
```

tar

```
#!/bin/bash
OF=/var/my-backup-$(date +%Y%m%d).tgz
tar -cZf $OF /home/me/
```

Local variables

Local variables can be created by using the keyword local.

```
#!/bin/bash
HELLO=Hello
function hello {
        local HELLO=World
        echo $HELLO
}
echo $HELLO
hello
echo $HELLO
```

Local variables

```
#!/bin/bash
T1="foo"
T2="bar"
if [ "$T1" = "$T2" ]; then
    echo expression evaluated as true
else
    echo expression evaluated as false
fi
```

for

```
#!/bin/bash
for i in $( ls ); do
    echo item: $i
done
```

C-like for

```
#!/bin/bash
for i in `seq 1 10`;
do
        echo $i
done
```

While

```
#!/bin/bash
COUNTER=0
while [  $COUNTER -lt 10 ]; do
    echo The counter is $COUNTER
    let COUNTER=COUNTER+1
done
```

Untile

```
#!/bin/bash
COUNTER=20
until [  $COUNTER -lt 10 ]; do
    echo COUNTER $COUNTER
    let COUNTER-=1
done
```

Functions with parameters

```
#!/bin/bash
function quit {
   exit
}
function e {
    echo $1
}
e Hello
e World
quit
echo foo
```

Using the command line

```
#!/bin/bash
if [ -z "$1" ]; then
     echo usage: $0 directory
     exit
fi
SRCD=$1
TGTD="/var/backups/"
OF=home-$(date +%Y%m%d).tgz
tar -cZf $TGTD$OF $SRCD
```

User input

```
#!/bin/bash
echo Please, enter your firstname and lastname
read FN LN
echo "Hi! $LN, $FN !"
```

File renamer (simple)

```
#!/bin/bash
# renames.sh
# basic file renamer

criteria=$1
re_match=$2
replace=$3

for i in $( ls *$criteria* );
do
    src=$i
    tgt=$(echo $i | sed -e "s/$re_match/$replace/")
    mv $src $tgt
done
```

Tcl/Tk

- Creator: John Ousterhout
- Tool command language (tickle)
- Everything Is A String (EIAS)
- http://www.tcl.tk

Math

```
% set result [expr (4+6)/4]
2
% set result [expr (4.0+6)/4]
2.5
% set variable 255
% puts "The␣number␣$variable"
The number 255
% puts [format "The␣number␣%d␣is␣equal␣to␣0x%02X" \
  $variable $variable]
The number 255 is equal to 0xFF
```

if

```
if {$c == "Hell"} {
    puts "Oh god !"
} else {
    puts "Peace !"
}
```

while

```
% while {$i<4} {
> puts "$i*$i␣is␣[expr␣$i*$i]"
> incr i
> }
0*0 is 0
1*1 is 1
2*2 is 4
3*3 is 9
```

for

```
% for {set i 0} {$i<4} {incr i} {
> puts "$i*$i␣is␣[expr␣$i*$i]"
> }
0*0 is 0
1*1 is 1
2*2 is 4
3*3 is 9
```

foreach

```
% set observations \
  {Bruxelles 15 22 London 12 19 Paris 18 27}
Bruxelles 15 22 London 12 19 Paris 18 27
% foreach {town Tmin Tmax} $observations {
> set Tavg [expr ($Tmin+$Tmax)/2.0]
> puts "$town␣$Tavg"
> }
Bruxelles 18.5
London 15.5
Paris 22.5
```

Array

```
% set observations \
  {Bruxelles 15 22 London 12 19 Paris 18 27}
Bruxelles 15 22 London 12 19 Paris 18 27
% foreach {town Tmin Tmax} $observations {
set obs($town-min) $Tmin
set obs($town-max) $Tmax
}
% parray obs
obs(Bruxelles-max) = 22
obs(Bruxelles-min) = 15
obs(London-max)    = 19
obs(London-min)    = 12
obs(Paris-max)     = 27
obs(Paris-min)     = 18
```

Procedures

```
% proc sum2 {a b} {
>  return [expr $a+$b]
> }
```

if a procedure does not contain any ’return’ statement, the default return value is the return value of the last evaluated function in this procedure. So the following script is perfectly equivalent :

```
% proc sum2 {a b} {
>    expr $a + $b
> }
```

To call the ’sum2’ function, we do the following :

```
% sum2 12 5
17
```

```
% proc sum {args} {
>     set result 0
>     foreach n $args {
>         set result [expr $result+$n]
>     }
>     return $result
> }
% sum 12 9 6 4
31
```

```
% proc count {start end {step 1}} {
>   for {set i $start} {$i<=$end} {incr i $step} {
>     puts $i
>   }
> }
% count 1 3
1
2
3
% count 1 5 2
1
3
5
```

```
% set global_counter 3
% proc incr_counter {} {
>     global global_counter
>     incr global_counter
> }
% incr_counter
 4
% set global_counter
 4
```

```
% set counter(value) 3
% set counter(active) 1
% proc incr_counter {} {
>     global counter
>     if {$counter(active)} {
>         incr counter(value)
>     }
> }
% incr_counter
4
% set counter(active) 0
0
% incr_counter
4
```

Eval

- concatenate all its arguments in one string
- splits this string using spaces as separators
- evaluate the command sentence formed by all the substrings

```
%  proc average {args} {
>      return [expr [eval sum $args] / [llength $args]]
>  }
% average 45.0 65.0 78.0 55.0
60.75
```

upvar

With the 'upvar' command, you can access a variable which belongs to a higher level of the procedure call stack.

```
% proc decr {n steps} {
>   upvar $n upa
>   set upa [expr $upa - $steps]
> }
% set nb 12
12
% decr nb 3
9
% puts $nb
9
```

uplevel

With the 'uplevel' command, you can evaluate something on higher level in the stack.

```
% proc do {todo condition} {
>     set ok 1
>     while {$ok} {
>        uplevel $todo
>        if {[uplevel "expr␣$condition"]==0} {set ok 0}
>     }
> }
% set i 0
0
% do {
puts $i
incr i
} {$i<4}
0
1
2
3
```

Perl

- Larry Wall
- Practical Extraction and Report Language(实用摘录和报告语言)
- Pathologically Eclectic Rubbish Lister(病态折衷垃圾列表器)

Operations and Assignment

```
#Perl uses all the usual C arithmetic operators:
$a = 1 + 2;       # Add 1 and 2 and store in $a
$a = 3 - 4;       # Subtract 4 from 3 and store in $a
$a = 5 * 6;       # Multiply 5 and 6
$a = 7 / 8;       # Divide 7 by 8 to give 0.875
$a = 9 ** 10;     # Nine to the power of 10
$a = 5 % 2;       # Remainder of 5 divided by 2
++$a;             # Increment $a and then return it
$a++;             # Return $a and then increment it
--$a;             # Decrement $a and then return it
$a--;             # Return $a and then decrement it
#and for strings Perl has the following among others:
$a = $b . $c;     # Concatenate $b and $c
$a = $b x $c;     # $b repeated $c times
#To assign values Perl includes
$a = $b;          # Assign $b to $a
$a += $b;         # Add $b to $a
$a -= $b;         # Subtract $b from $a
$a .= $b;         # Append $b onto $a
```

```
# print apples and pears using concatenation:
$a = 'apples';
$b = 'pears';
print $a.'␣and␣'.$b;
#It would be nicer to include only one string
# in the final print statement, but the line
print '$a␣and␣$b';
#prints literally $a and $b which isn't very helpful.
# Instead we can use the double quotes
# in place of the single quotes:
print "$a␣and␣$b";
#The double quotes force interpolation of any codes,
# including interpreting variables.
# This is a much nicer than our original statement.
# Other codes that are interpolated include
# special characters such as newline and tab.
# The code \n is a newline and \t is a tab.
```

Array

```
#The statement
@food  = ("apples", "pears", "eels");
@music = ("whistle", "flute");
# assigns a list to the array variable @food
# and a list to the array variable @music.
# Array is accessed by using indices starting from 0,
# and square brackets are used to specify the index.
# The expression
$food[2]
# returns eels. Notice that the @ has changed to a $
# because eels is a scalar.
```

push

```
# The first assignment below explodes the @music
# variable so that it is equivalent to the second.
@moremusic = ("organ", @music, "harp");
@moremusic = ("organ", "whistle", "flute", "harp");
# A neater way of adding elements is to use:
push(@food, "eggs");
# which pushes eggs onto the end of the array @food.
# To push two or more items onto the array use
# one of the following forms:
push(@food, "eggs", "lard");
push(@food, ("eggs", "lard"));
push(@food, @morefood);
# "push" function returns the length of the new list.
```

pop

```
# To remove the last item from a list
# and return it use the pop function.
# From our original list "pop" function returns eels
# and @food now has two elements:
$grub = pop(@food);      # Now $grub = "eels"
# It is also possible to assign an array to a scalar.
# As usual context is important. The line
$f = @food;
# assigns the length of @food, but
$f = "@food";
# turns the list into a string with a space
# between each element.
# Arrays can also be used to
# make multiple assignments to scalar variables:
($a, $b) = ($c, $d);# Same as $a=$c; $b=$d;
($a, $b) = @food;# $a and $b are the first two
                 # items of @food.
($a, @somefood) = @food;# $a is the first item of
                        # @food, @somefood is a
                        # list of the others.
(@somefood, $a) = @food;# @somefood is @food and
                        # $a is undefined.
# The last assignment occurs
# because arrays are greedy,
# and @somefood will swallow up
# as much of @food as it can.
# Therefore that form is best avoided.
# Finally, you may want to find the index of
# the last element of a list.
# To do this for the @food array use:
$#food
```

Associative arrays

```
%ages = ("Michael␣Caine", 39,
         "Dirty␣Den", 34,
         "Angie", 27,
         "Willy", "21␣in␣dog␣years",
         "The␣Queen␣Mother", 108);
$ages{"Michael␣Caine"}; # Returns 39
$ages{"Dirty␣Den"}; # Returns 34
$ages{"Angie"}; # Returns 27
$ages{"Willy"}; # Returns "21 in dog years"
$ages{"The␣Queen␣Mother"};# Returns 108

@info = %ages;   # @info is a list array. It
                 # now has 10 elements
$info[5];        # Returns the value 27 from
                 # the list array @info
%moreages = @info;# %moreages is an associative
                  # array. It is the same as %ages
```

Testing

```
Testing
$a == $b # Is $a numerically equal to $b?
         # Beware: Don't use the = operator.
$a != $b # Is $a numerically unequal to $b?
$a eq $b # Is $a string-equal to $b?
$a ne $b # Is $a string-unequal to $b?

#You can also use logical and, or and not:

($a && $b) # Is $a and $b true?
($a || $b) # Is either $a or $b true?
!($a)      # is $a false?
```

Conditionals

```
if (!$a) # The ! is the not operator
{
  print "The␣string␣is␣empty\n";
}
elsif (length($a) == 1) # If above fails, try this
{
  print "The␣string␣has␣one␣character\n";
}
elsif (length($a) == 2) # If that fails, try this
{
  print "The␣string␣has␣two␣characters\n";
}
else  # Now, everything has failed
{
  print "The␣string␣has␣lots␣of␣characters\n";
}
```

foreach

```
foreach $morsel (@food) # Visit each item in turn
                        # and call it $morsel
{
        print "$morsel\n";# Print the item
        print "Yum␣yum\n";# That was nice
}
```

for

First of all the statement initialise is executed. Then while test is true the block of actions is executed. After each time the block is executed inc takes place. Here is an example for loop to print out the numbers 0 to 9.

```
for ($i = 0; $i < 10; ++$i)
# Start with $i = 1
# Do it while $i < 10
# Increment $i before repeating
{
        print "$i\n";
}
```

while

```
#!/usr/local/bin/perl
print "Password?␣";                # Ask for input
$a = <STDIN>;                      # Get input
chop $a;                           # Remove the newline at end
while ($a ne "fred")               # While input is wrong...
{
    print "sorry.␣Again?␣";        # Ask again
    $a = <STDIN>;                  # Get input again
    chop $a;                       # Chop off newline again
}
```

while

```
#!/usr/local/bin/perl
do
{
        "Password?␣";           # Ask for input
        $a = <STDIN>;           # Get input
        chop $a;                # Chop off newline
}
while ($a ne "fred")            # Redo while wrong input
```

Subroutines

```
sub mysubroutine
{
        print "Not␣a␣very␣interesting␣routine\n";
        print "This␣does␣the␣same␣thing␣every␣time\n";
}

&mysubroutine;          # Call the subroutine
&mysubroutine($_);      # Call it with a parameter
&mysubroutine(1+2, $_); # Call it with two parameters
```

Parameters

```
sub printargs
{
        print "@_\n";
}

&printargs("perly", "king");
# Example prints "perly king"
&printargs("frog", "and", "toad");
# Prints "frog and toad"

sub printfirsttwo
{
  print "Your␣first␣argument␣was␣$_[0]\n";
  print "and␣$_[1]␣was␣your␣second\n";
}
```

Returning values

```
sub maximum
{
        if ($_[0] > $_[1])
        {
                $_[0];
        }
        else
        {
                $_[1];
        }
}

$biggest = &maximum(37, 24);# Now $biggest is 37
```

Local variables

```
$a=1;
$b=1;
sub local_test
{
   local($a, $b); # Make local variables
   ($a, $b) = ($_[0], $_[1]);# Assign values
}
&local_test(2,2);
```

In fact, it can even be tidied up by replacing the first two lines with

```
local($a, $b) = ($_[0], $_[1]);
```

Python

- Guido van Rossum
- Monty Python's Flying Circus
- Indentation

```
# Fibonacci numbers, imperative style
N=10

 first = 0     # seed value fibonacci(0)
 second = 1    # seed value fibonacci(1)
 fib_number = first + second
# calculate fibonacci(2)
 for position in range(N-2):
# iterate N-2 times to give Fibonacci number
     first = second
# update the value of the 'previous' variables
     second = fib_number
     fib_number = first + second
# update the result value to fibonacci(position)
 print fib_number
```

```
# Fibonacci numbers, functional style
N=10

# Fibonacci numbers, functional style
def fibonacci(position):
# Fibonacci number N (for N >= 0)
    if position == 0: return 0
# seed value fibonacci(0)
    elif position == 1: return 1
# seed value fibonacci(1)
    else: return fibonacci(position-1)
                 + fibonacci(position-2)
# calculate fibonacci(position)

fib_number = fibonacci(N)
print fib_number
```

思考

- 当前有哪些脚本语言，它们的特点是什么？