Chapter4: bash变量和参数介绍

概述

- 变量可以让程序和脚本语言用来描述数据。一个变量仅仅是一个标签而已，被指定到计算机内存中存储着数据的某个位置或某些位置的标签。变量一般出现在算术运算操作和数量操纵及字符串解析中。

4.1. 变量替换(Variable Substitution)

- 变量的名称是其值的占位符，代表它所包含的数据。引用(检索)它的值称为变量替换。
- 举个例子来仔细区别一个变量名和其存储的值。例如：如果val1是某个变量的名称，那么使用$val1来引用该变量，即是其存储的数据。

```
[root@centos8 ~]val1=hello          # 将字符串"hello"赋值给变量val1
[root@centos8 ~]echo $val1          # 使用$符来引用该变量
hello
[root@centos8 ~]echo val1           # 不加$无法引用val所存储的数据
val1
```

- 只有在如下环境中，变量前的$符可以不出现：

(引用)变量不出现$前缀的情况
declared or assigned:声明其或者其被赋值[^1]
被unset或者export
在双圆括号((...))的算术运算操作结构中
变量代表某个信号的特殊情况[^2]

- 将引用值包括在两个双引号("$val")中不会影响其对变量值的引用(变量替换)。该引用方法称作"部分引用",或者叫做"弱引用"。但是使用两个单引号('$val')则会使变量替换失效，shell按照变量名的字面意思处理其。该引用方式称为"全引用",也叫作"强引用"。第五章详细介绍。
- **需要注意的是：$val实际上是${$val}的简写。某些场合前者$val会出错，使用长格式的后者兼容性更好。**
  -
- 例4-1. 变量赋值和变量替换：

```
#!/bin/bash
# Variables: 变量赋值和变量替换
a=375
hello=$a
#   ^ ^
#-----------------------------------------------------------------------
```

```
# 当进行赋值操作时，等号=两边不允许出现空格。
# 若出现空格会如何?
#  "VARIABLE =value"
#           ^
#% 上面脚本会试图运行"VARIABLE"命令，并且认为其参数为："=value"
#  "VARIABLE= value"
#            ^
#% 上面脚本会试图运行"value"命令，并且认为与其相关的环境变量"VARIABLE"的值是" "(空格)
#--------------------------------------------------------------------------
echo hello    # hello
# 此处 "hello" 不是变量引用，只是字符串
echo $hello   # 375
#    ^          这是变量引用
echo ${hello} # 375
#               同上，这是变量引用
# Quoting . . .
echo "$hello"    # 375  # 弱引用
echo "${hello}"  # 375  # 弱引用
echo
hello="A B  C   D"                                  # 给hello变量赋值，不用$
echo $hello   # A B C D                             # 变量引用结果未保留空格
echo "$hello" # A B  C   D                          # 变量引用结果保留空格
# 可以看到 echo $hello 和 echo "$hello" 结果不同
# ======================================
# 使用双引号引用可以保留变量的空白符
# ======================================
echo
echo '$hello'  # 输出：$hello  而不是："A B  C   D"
#    ^      ^
#  单引号使变量引用失效,导致$符号被按照字面值对待，而不认为其是引用变量

hello=    # 设置hello为空(null).
echo "\$hello (null value) = $hello"      # $hello (null value) =
#  Note that setting a variable to a null value is not the same as
#+ unsetting it, although the end result is the same (see below).
# ------------------------------------------------------------------
#  在同一行中可使用空格隔开来定义多个变量
#  值得注意的是，这样做会导致代码不易读，并且不利于移植。
var1=21  var2=22  var3=$V3
echo
echo "var1=$var1   var2=$var2   var3=$var3"
# 运行上面代码，老版本的shell可能会出错
# ------------------------------------------------------------------
echo; echo
numbers="one two three"
#          ^   ^
other_numbers="1 2 3"
#              ^ ^
#  如果某个变量里面含有空白符，则需要使用引号引用起来
#  other_numbers=1 2 3                  # 这样赋值会报错
echo "numbers = $numbers"
echo "other_numbers = $other_numbers"   # other_numbers = 1 2 3
#  使用反斜杠转义空白符亦可行
mixed_bag=2\ ---\ Whatever
```

```
#           ^    ^ Space after escape (\).
echo "$mixed_bag"         # 2 --- Whatever
echo; echo
echo "uninitialized_variable = $uninitialized_variable"
# 未初始化的变量具有空值 null(即是没有任何值)
uninitialized_variable=   #  声明但是没有赋值，和上面一样 --

echo "uninitialized_variable = $uninitialized_variable"
                          # It still has a null value.
uninitialized_variable=23       # 初始化变量
unset uninitialized_variable    # 删除该变量
echo "uninitialized_variable = $uninitialized_variable"
                                # uninitialized_variable =
                                # 任然为空
echo
exit 0
```

- 注意：一个未初始化的变量具有空值--没有赋值，不是**0**!
- 在还未赋值时使用某个变量可能会出现各种问题。但是未初始化的变量却可以使用在算术应算中。

```
[root@centos8 ~] echo "$uninitialized"            # 未初始化的变量

[root@centos8 ~] let "uninitialized += 5"         # 加5
[root@centos8 ~] echo "$uninitialized"
5                                                 # 结果为5

#  结论:
#  未初始化的变量没有值,但是在算术运算中视为0
```

4.2. 变量赋值(Variable Assignment)

> = 等号为赋值操作符(其前后不允许出现空白符)

- 注意：=号在不同的环境下既可以是赋值操作符，也可以是测试操作符。

> 例4-2. 一般变量赋值操作

```
#!/bin/bash
# Naked variables
echo
# When is a variable "naked", i.e., lacking the '$' in front?
# When it is being assigned, rather than referenced.
# Assignment
a=879
echo "The value of \"a\" is $a."
# 使用'let'关键字来做简单的运算赋值操作
let a=16+5
echo "The value of \"a\" is now $a."
```

```
echo
# 在for循环中(实际上是一种伪装的赋值操作):
echo -n "Values of \"a\" in the loop are: "
for a in 7 8 9 11
do
  echo -n "$a "
done
echo
echo
# In a 'read' statement (另一种赋值操作):
echo -n "Enter \"a\" "
read a
echo "The value of \"a\" is now $a."
echo
exit 0
```

例4-3. 一般变量赋值操作和特殊的赋值手法

```
#!/bin/bash
a=23              # Simple case
echo $a
b=$a
echo $b
# 使用(命令替换)来赋值
a=`echo Hello!`   # 将'echo'命令的输出结果赋值给 'a' ...
==================================
[root@centos8 /data]a=`echo Hello!`
-bash: !`: event not found
==================================
echo $a
#  Note that including an exclamation mark (!) within a
#+ command substitution construct will not work from the command-line,
#+ since this triggers the Bash "history mechanism."
#  Inside a script, however, the history functions are disabled by default.
a=`ls -l`         # 将命令'ls -l' 的运行结果 赋值给 'a'
echo $a           # 由于赋值没有使用双引号引起来，所以结果未保留格式
==================================
[root@centos8 /data] ls -l
100664000 drwxr-xr-x 2 root root 204 Oct  2 09:53 scripts
      131 drwxr-xr-x 2 root root  21 Oct  2 21:01 test
      171 drwxr-xr-x 4 root root  69 Oct  2 21:01 tmp
[root@centos8 /data] a=`ls -l`
[root@centos8 /data] echo $a
total 0 drwxr-xr-x 2 root root 204 Oct 2 09:53 scripts drwxr-xr-x 2 root root 21
Oct 2 21:01 test drwxr-xr-x 4 root root 69 Oct 2 21:01 tmp
==================================

echo
echo "$a"         # 引起来的变量保留了格式
                  # (See the chapter on "Quoting.")
==================================
[root@centos8 /data]echo "$a"
```

```
total 0
drwxr-xr-x 2 root root 204 Oct  2 09:53 scripts
drwxr-xr-x 2 root root  21 Oct  2 21:01 test
drwxr-xr-x 4 root root  69 Oct  2 21:01 tmp
==================================
exit 0
```

- 使用$(...)结构来变量赋值也是一种命令替换。（比反引号新的一种变量替换）

```
# From /etc/rc.d/rc.local
R=$(cat /etc/redhat-release)
arch=$(uname -m)
```

4.3. Bash变量是无强制类型要求的(Bash Variables Are Untyped)

- 并不像其他大多数编程语言，Bash并不以"类型"来区分其变量。本质上，Bash变量都是字符串，但是由于语境不同，Bash允许在变量上进行算术运算操作和比较操作。决定性因数是该变量是否只仅仅包含数字。

> 例4-4. 整数还是字符串?

```
#!/bin/bash
# int-or-string.sh
a=2334                   # 变量a是整数.
let "a += 1"
echo "a = $a "           # a = 2335
echo                     # 还是整数.
b=${a/23/BB}             # 将"23"替换为"BB".
                         # 该操作将 $b 转换为字符串.
echo "b = $b"            # b = BB35
declare -i b             # 即使声明其为整数也不行，还是字符串.
echo "b = $b"            # b = BB35
let "b += 1"             # BB35 + 1
echo "b = $b"            # b = 1
echo                     # Bash 将一个字符串的整数值视为0".
c=BB34
echo "c = $c"            # c = BB34
d=${c/BB/23}             # 将"BB"替换为"23".
                         # 该操作使得$d 成为整.
echo "d = $d"            # d = 2334
let "d += 1"             # 2334 + 1
echo "d = $d"            # d = 2335
echo
# 空(null)变量会怎样?
e=''                     # ... 或者 e="" ... 或者 e=
echo "e = $e"            # e =
let "e += 1"             # 允许对一个空变量执行算术运算?
echo "e = $e"            # e = 1
echo                     # 空变量转换为一个整数0.
```

```
# 未声明的变量呢?
echo "f = $f"            # f =
let "f += 1"             # Arithmetic operations allowed?
echo "f = $f"            # f = 1
echo                     # 未声明的变量转换为整数0.
#
# However ...
let "f /= $undecl_var"   # Divide by zero?
#  错误提示：-bash: let: f /= : syntax error: operand expected (error token is " ")
# Syntax error! 这里，变量 $undecl_var未被设置为0!
#
# But still ...
let "f /= 0"
#   let: f /= 0: division by 0 (error token is "0")
# Expected behavior.
#  Bash (usually) sets the "integer value" of null to zero
#+ when performing an arithmetic operation.
#  But, don't try this at home, folks!
#  It's undocumented and probably non-portable behavior.
# Conclusion: Variables in Bash are untyped,
#+ with all attendant consequences.
exit $?
```

- 无强制变量类型是把双刃剑。这使得代码更加灵活并且很容易的写出很多行代码（同时给你设了足够多的坑！）。然而，这样的灵活性和便利性使得很多微小的失误混进你的代码形成疏忽的代码编写习惯。
- 为了便于追踪脚本中的变量类型，bash也允许声明变量。

4.4. 特殊变量类型(Special Variable Types)

1.局部变量(Local variables)

- 局部变量指：只属于某个代码块或者函数的变量。

2.环境变量(Environmental variables)

- 环境变量：影响shell行为和用户接口的变量。
-
  - 广义来说，每个进程都有自己的"环境",也就是该进程所引用的变量的集合。从这个意义上来说，shell的行为和其他任何进程都一样。
-
  - 每次新的shell启动，它就会创建相对应于它自己的环境变量的另一个变量集合。更新或者新增变量都会使得shell更新其自己的环境，其所有的子shell进程会继承其环境。

- 如果一个脚本设置了环境变量，这些变量需要被"导出（export）"，也就是报告给该脚本的本地环境。这就是export命令的功能。

- **注意注意注意：**一个脚本只能将其变量导出到其子进程[^3]（该脚本初始化的命令或者进程）。从命令行执行的脚本不能将变量导出到命令行环境。子进程无法将变量导出到产生其的父进程。

3.位置变量(Positional parameters)

- 位置变量：从命令行传给某个脚本的参数: $0, $1, $2, $3 . . .

- $0表示脚本的名称，$1表示脚本的第一个参数，$2表示第二个参数，依次只到$9；之后的参数引用必须使用花括号括起来如：${10},${13}...
- 特殊变量$*和$@都表示所有的位置参数。
  - $* 将所有位置参数视为单个字符串
  - $@ 将每个位置参数存储为单独引用的字符串，分开对待每个位置参数

> 例4-5. 位置参数

```
#!/bin/bash
# 至少使用 10 来调用该脚本
# ./scriptname 1 2 3 4 5 6 7 8 9 10
MINPARAMS=10
echo
echo "The name of this script is \"$0\"."
# Adds ./ for current directory
echo "The name of this script is \"`basename $0`\"."
# Strips out path name info (see 'basename')
echo
if [ -n "$1" ]              # 测试结构中变量需要引起来.
then
 echo "Parameter #1 is $1"  # Need quotes to escape #
fi
if [ -n "$2" ]
then
 echo "Parameter #2 is $2"
fi
if [ -n "$3" ]
then
 echo "Parameter #3 is $3"
fi
# ...
if [ -n "${10}" ]  # Parameters > $9 must be enclosed in {brackets}.
then
 echo "Parameter #10 is ${10}"
fi
echo "-----------------------------------"
echo "All the command-line parameters are: "$*""
if [ $# -lt "$MINPARAMS" ]
then
  echo
  echo "This script needs at least $MINPARAMS command-line arguments!"
fi
echo
exit 0

# 脚本运行结果
==============================================================================
## 不带参数
[root@centos8 /data/test] ./ttt.sh

The name of this script is "./ttt.sh".
The name of this script is "ttt.sh".
```

```
---------------------------------
All the command-line parameters are:

This script needs at least 10 command-line arguments!

## 带10个参数
[root@centos8 /data/test] ./ttt.sh 1 2 3 4 5 6 7 8 9 10

The name of this script is "./ttt.sh".
The name of this script is "ttt.sh".

Parameter #1 is 1
Parameter #2 is 2
Parameter #3 is 3
Parameter #10 is 10
---------------------------------
All the command-line parameters are: 1 2 3 4 5 6 7 8 9 10
================================================================================
```

- 下面花括号内的位置变量（使用${!val}的形式）使得引用最后一个从命令行传给脚本的位置参数非常方便，这需要使用感叹号！来间接引用。

```
#!/bin/bash

args=$#
echo $args
echo "last=${!args}"
echo "last=${!#}"

# 脚本运行结果
=====================================================
[root@centos8 /data/test] ./ttt.sh 1 2 3 a b c hello
7
last=hello
last=hello
=====================================================
```

- 某些脚本由于调用它们的名字不同，会表现出不同的行为。要使得这种用法可行，脚本需要检查**$0**这个位置参数，即调用它的名字。这些可选的名字必须是不同的符号链接（**symbolic links**）**[^4]**。 如下面的例子，"origin.sh"有两个软链接分别为hello和hi，使用hello和hi来运行"origin.sh"，其表现出不同的行为。

```
[root@centos8 /data/test]ll
total 4
140 lrwxrwxrwx 1 root root   9 Oct  4 09:19 hello -> origin.sh
144 lrwxrwxrwx 1 root root   9 Oct  4 09:20 hi -> origin.sh
142 -rwxr-xr-x 1 root root 468 Oct  4 09:18 origin.sh
[root@centos8 /data/test] cat origin.sh
```

```
#!/bin/bash
#
#********************************************************************************
#Author:                        steveli
#QQ:                            1049103823
#Data:                      2019-10-04
#FileName:                  origin.sh
#URL:                   https://blog.csdn.net/YouOops
#Description:           Test scrpting.
#Copyright (C):     2019 All rights reserved
#********************************************************************************

case `basename $0` in                           # `basename $0`取出运行该脚本的软连接
名称
hello)
echo hello
;;
hi)
echo hi
;;
*)
echo error
;;
esac

[root@centos8 /data/test]./hello            # 使用软连接hello运行origin.sh
hello
[root@centos8 /data/test]./hi               # 使用软连接hi运行origin.sh
hi
```

- 如果一个脚本期望调用它时需要一个参数而实际却没有，这可能会导致一个无法预测的结果。避免这种错误的一个方法是在使用该位置参数的赋值语句的两边都追加一个额外的字符。

```
variable1_=$1_  # 而不是 variable1=$1
# 即使脚本未带参数，也不会报错
critical_argument01=$variable1_
# 多余的字符可以用下面的写法剥离.
variable1=${variable1_/_/}
# Side effects only if $variable1_ begins with an underscore.
# This uses one of the parameter substitution templates discussed later.
# (Leaving out the replacement pattern results in a deletion.)
# 更加直接的方法是测试所期望的位置参数是不是存在。
if [ -z $1 ]
then
  exit $E_MISSING_POS_PARAM
fi
#  然而，Fabian Kreutz 这个大佬指出这种用测试的方法
# 来确认是否有带位置参数的方法可能有副作用
# 更加合适的方法是:
#         ${1:-$DefaultVal}    # 该结构表示如果位置参数$1如果没有设置或者值为空则给$1赋
值为                                    # $DefaultVal；如果$1已被设置且有值，则使用$1的
```

```
原来的值
# 后面参数替换小节具体讨论
```

例4-6. wh, whois 域名查看

```
#!/bin/bash
# ex18.sh
# Does a 'whois domain-name' lookup on any of 3 alternate servers:
#                    ripe.net, cw.net, radb.net
# Place this script -- renamed 'wh' -- in /usr/local/bin
# Requires symbolic links:
# ln -s /usr/local/bin/wh /usr/local/bin/wh-ripe
# ln -s /usr/local/bin/wh /usr/local/bin/wh-apnic
# ln -s /usr/local/bin/wh /usr/local/bin/wh-tucows
E_NOARGS=75
if [ -z "$1" ]
then
  echo "Usage: `basename $0` [domain-name]"
  exit $E_NOARGS
fi
# 检查脚本名称并调用不同的服务.
case `basename $0` in    # Or:    case ${0##*/} in
    "wh"       ) whois $1@whois.tucows.com;;
    "wh-ripe"  ) whois $1@whois.ripe.net;;
    "wh-apnic" ) whois $1@whois.apnic.net;;
    "wh-cw"    ) whois $1@whois.cw.net;;
    *          ) echo "Usage: `basename $0` [domain-name]";;
esac
exit $?
```

- shift命令会再次指定脚本的位置参数，将位置变量依次向左移。如：$1 <--- $2, $2 <--- $3, $3 <--- $4, 等。在脚本中执行一次shift后原来的位置变量$1会被清除，此后的$1实际上是shift未运行时的$2；但是值得注意的是$0(脚本名称)不变。多次执行shift后可以访问${10}以后的位置变量。

例4-7. 使用shift

```
#!/bin/bash
# shft.sh: 使用 'shift' 逐步访问每个位置参数.
#  Name this script something like shft.sh,
#+ 带多个参数调用该脚本.
#+ 例如:
#      bash shft.sh a b c def 83 barndoor
until [ -z "$1" ]  # 只到所有位置变量都已经被访问. . .
do
  echo -n "$1 "
  shift
done
echo               # Extra linefeed.
echo "$2"
```

```
#  Nothing echoes!
#  当 $2 移动到 $1 (并且没有 $3 移动到 $2)
#+ 那么 $2 此时为空.
#  所以是参数'移动', 而不是'拷贝'.
exit

# 运行结果
============================================================
[root@centos8 /data/test]./shift_para.sh 1 2 3 as ds asd ffg
1 2 3 as ds asd ffg

============================================================
```

- shift命令可以跟一个数字n表示位置参数被左移n次

```
#!/bin/bash
# shift-past.sh
shift 3    # 左移3个位置.
#  n=3; shift $n
#  上面的写法也行.
echo "$1"
exit 0

# 运行结果
# ======================== #
$ sh shift-past.sh 1 2 3 4 5
4
# ======================== #
#  然而， Eleni Fragkiadaki 大哥指出,试图位移超过脚本所带位置参数个数($#)的次数,
#+ 会导致shift命令的退出状态为1，脚本的位置参数不变，这意味着脚本可能会陷入死循环。

#  例如:
#      until [ -z "$1" ]
#      do
#         echo -n "$1 "
#         shift 20    #  如果位置参数个数少于 20,
#      done           #+ 那么循环永远不会停!
#
# 当有疑问时，添加一个完整性检查比较好. . . .
#           shift 20 || break
#                    ^^^^^^^^
```

- shift命令移动位置参数的工作方式和传送到某个函数的参数相似。

注脚

[^1]: 赋值(assignment)有多种形式：使用等号 = (如：var1=27)；read语句中 ；循环的头部(如：for var2 in 1 2 3) [^2]: Linux中的信号指的是某个进程或者内核发送给另一个进程的消息，告诉其采取一些具体动作（通常是结束进程）。例如：按下Ctrl+C键发送一个用户中断信号给某个正在运行的进程，即一个INT信号 [^3]:子进程

(subprocess): 一个子进程是被另一个进程所产生的进程，后者叫该进程的父进程。 [^4]:符号链接(symbolic links)又叫软链接(soft links):在UNIX/linux系统中用来访问某文件或则某文件夹的另一个方法，软链接类似windows的快捷方式；在linux使用ln命令为某个文件或文件夹创建软链接。ln通过在文件系统的文件分配表中添加一个条目的附加名称来实现这一点。这样就可以使用附加的名称和原名称来访问某文或文件夹。这哥们介绍的不错Frank Hofmann