引号和转义 字符串操作

Bash 变量

网道（WangDoc.com），互联网文档计划

目录 [隐藏]

1. 简介

Bash 变量分成环境变量和自定义变量两类。

1.1 环境变量

环境变量是 Bash 环境自带的变量，进入 Shell 时已经定义好了，可以直接使用。它们通常是系统定义好的，也可以由用户从父 Shell 传入子 Shell。

`env` 命令或 `printenv` 命令，可以显示所有环境变量。

```
$ env
# 或者
$ printenv
```

下面是一些常见的环境变量。

- `BASHPID` ： Bash 进程的进程 ID。
- `BASHOPTS` ： 当前 Shell 的参数，可以用 `shopt` 命令修改。
- `DISPLAY` ： 图形环境的显示器名字，通常是 `:0` ，表示 X Server 的第一个显示器。
- `EDITOR` ： 默认的文本编辑器。
- `HOME` ： 用户的主目录。
- `HOST` ： 当前主机的名称。
- `IFS` ： 词与词之间的分隔符，默认为空格。
- `LANG` ： 字符集以及语言编码，比如 `zh_CN.UTF-8` 。
- `PATH` ： 由冒号分开的目录列表，当输入可执行程序名后，会搜索这个目录列表。
- `PS1` ： Shell 提示符。
- `PS2` ： 输入多行命令时，次要的 Shell 提示符。
- `PWD` ： 当前工作目录。
- `RANDOM` ： 返回一个0到32767之间的随机数。
- `SHELL` ： Shell 的名字。
- `SHELLOPTS` ： 启动当前 Shell 的 `set` 命令的参数，参见《set 命令》一章。
- `TERM` ： 终端类型名，即终端仿真器所用的协议。
- `UID` ： 当前用户的 ID 编号。
- `USER` ： 当前用户的用户名。

很多环境变量很少发生变化，而且是只读的，可以视为常量。由于它们的变量名全部都是大写，所以传统上，如果用户要自己定义一个常量，也会使用全部大写的变量名。

链接

本文源码
代码仓库
反馈

注意，Bash 变量名区分大小写， `HOME` 和 `home` 是两个不同的变量。

查看单个环境变量的值，可以使用 `printenv` 命令或 `echo` 命令。

```
$ printenv PATH
# 或者
$ echo $PATH
```

注意， `printenv` 命令后面的变量名，不用加前缀 `$` 。

1.2 自定义变量

自定义变量是用户在当前 Shell 里面自己定义的变量，仅在当前 Shell 可用。一旦退出当前 Shell，该变量就不存在了。

`set` 命令可以显示所有变量（包括环境变量和自定义变量），以及所有的 Bash 函数。

```
$ set
```

2. 创建变量

用户创建变量的时候，变量名必须遵守下面的规则。

- 字母、数字和下划线字符组成。
- 第一个字符必须是一个字母或一个下划线，不能是数字。
- 不允许出现空格和标点符号。

变量声明的语法如下。

```
variable=value
```

上面命令中，等号左边是变量名，右边是变量。注意，等号两边不能有空格。

如果变量的值包含空格，则必须将值放在引号中。

```
myvar="hello world"
```

Bash 没有数据类型的概念，所有的变量值都是字符串。

下面是一些自定义变量的例子。

```
a=z                     # 变量 a 赋值为字符串 z
b="a string"            # 变量值包含空格，就必须放在引
c="a string and $b"     # 变量值可以引用其他变量的值
d="\t\ta string\n"      # 变量值可以使用转义字符
e=$(ls -l foo.txt)      # 变量值可以是命令的执行结果
f=$((5 * 7))            # 变量值可以是数学运算的结果
```

变量可以重复赋值，后面的赋值会覆盖前面的赋值。

```
$ foo=1
$ foo=2
$ echo $foo
2
```

上面例子中，变量 `foo` 的第二次赋值会覆盖第一次赋值。

如果同一行定义多个变量，必须使用分号（ `;` ）分隔。

```
$ foo=1;bar=2
```

上面例子中，同一行定义了 `foo` 和 `bar` 两个变量。

3. 读取变量

读取变量的时候，直接在变量名前加上 `$` 就可以了。

```
$ foo=bar
$ echo $foo
bar
```

每当 Shell 看到以 `$` 开头的单词时，就会尝试读取这个变量名对应的值。

如果变量不存在，Bash 不会报错，而会输出空字符。

由于 `$` 在 Bash 中有特殊含义，把它当作美元符号使用时，一定要非常小心，

```
$ echo The total is $100.00
The total is 00.00
```

上面命令的原意是输入 `$100` ，但是 Bash 将 `$1` 解释成了变量，该变量为空，因此输入就变成了 `00.00` 。所以，如果要使用 `$` 的原义，需要在 `$` 前面放上反斜杠，进行转义。

```
$ echo The total is \$100.00
The total is $100.00
```

读取变量的时候，变量名也可以使用花括号 `{}` 包围，比如 `$a` 也可以写成 `${a}` 。这种写法可以用于变量名与其他字符连用的情况。

```
$ a=foo
$ echo $a_file

$ echo ${a}_file
foo_file
```

上面代码中，变量名 `a_file` 不会有任何输出，因为 Bash 将其整个解释为变量，而这个变量是不存在的。只有用花括号区分 `$a` ，Bash 才能正确解读。

事实上，读取变量的语法 `$foo` ，可以看作是 `${foo}` 的简写形式。

如果变量的值本身也是变量，可以使用 `${!varname}` 的语法，读取最终的值。

```
$ myvar=USER
$ echo ${!myvar}
```

```
ruanyf
```

上面的例子中，变量 `myvar` 的值是 `USER` ， `${!myvar}` 的写法将其展开成最终的值。

如果变量值包含连续空格（或制表符和换行符），最好放在双引号里面读取。

```
$ a="1 2  3"
$ echo $a
1 2 3
$ echo "$a"
1 2  3
```

上面示例中，变量 `a` 的值包含两个连续空格。如果直接读取，Shell 会将连续空格合并成一个。只有放在双引号里面读取，才能保持原来的格式。

4. 删除变量

`unset` 命令用来删除一个变量。

```
unset NAME
```

这个命令不是很有用。因为不存在的 Bash 变量一律等于空字符串，所以即使 `unset` 命令删除了变量，还是可以读取这个变量，值为空字符串。

所以，删除一个变量，也可以将这个变量设成空字符串。

```
$ foo=''
$ foo=
```

上面两种写法，都是删除了变量 `foo` 。由于不存在的值默认为空字符串，所以后一种写法可以在等号右边不写任何值。

5. 输出变量，export 命令

用户创建的变量仅可用于当前 Shell，子 Shell 默认读取不到父 Shell 定义的变量。为了把变量传递给子 Shell，需要使用 `export` 命令。这样输出的变量，对于子 Shell 来说就是环境变量。

`export` 命令用来向子 Shell 输出变量。

```
NAME=foo
export NAME
```

上面命令输出了变量 `NAME` 。变量的赋值和输出也可以在一个步骤中完成。

```
export NAME=value
```

上面命令执行后，当前 Shell 及随后新建的子 Shell，都可以读取变量 `$NAME` 。

子 Shell 如果修改继承的变量，不会影响父 Shell。

```
# 输出变量 $foo
$ export foo=bar

# 新建子 Shell
$ bash

# 读取 $foo
$ echo $foo
bar

# 修改继承的变量
$ foo=baz

# 退出子 Shell
$ exit

# 读取 $foo
```

```
$ echo $foo
bar
```

上面例子中，子 Shell 修改了继承的变量 `$foo` ，对父 Shell 没有影响。

6. 特殊变量

Bash 提供一些特殊变量。这些变量的值由 Shell 提供，用户不能进行赋值。

（1） `$?`

`$?` 为上一个命令的退出码，用来判断上一个命令是否执行成功。返回值是 `0` ，表示上一个命令执行成功；如果不是零，表示上一个命令执行失败。

```
$ ls doesnotexist
ls: doesnotexist: No such file or directory

$ echo $?
1
```

上面例子中， `ls` 命令查看一个不存在的文件，导致报错。`$?` 为1，表示上一个命令执行失败。

（2） `$$`

`$$` 为当前 Shell 的进程 ID。

```
$ echo $$
10662
```

这个特殊变量可以用来命名临时文件。

```
LOGFILE=/tmp/output_log.$$
```

（3） `$_`

`$_` 为上一个命令的最后一个参数。

```
$ grep dictionary /usr/share/dict/words
dictionary

$ echo $_
/usr/share/dict/words
```

（4） `$!`

`$!` 为最近一个后台执行的异步命令的进程 ID。

```
$ firefox &
[1] 11064

$ echo $!
11064
```

上面例子中， `firefox` 是后台运行的命令， `$!` 返回该命令的进程 ID。

（5） `$0`

`$0` 为当前 Shell 的名称（在命令行直接执行时）或者脚本名（在脚本中执行时）。

```
$ echo $0
bash
```

上面例子中， `$0` 返回当前运行的是 Bash。

（6） `$-`

`$-` 为当前 Shell 的启动参数。

```
$ echo $-
himBHs
```

（7） `$@` 和 `$#`

`$#` 表示脚本的参数数量， `$@` 表示脚本的参数值，参见脚本一章。

7. 变量的默认值

Bash 提供四个特殊语法，跟变量的默认值有关，目的是保证变量不为空。

```
${varname:-word}
```

上面语法的含义是，如果变量 `varname` 存在且不为空，则返回它的值，否则返回 `word` 。它的目的是返回一个默认值，比如 `${count:-0}` 表示变量 `count` 不存在时返回 `0` 。

```
${varname:=word}
```

上面语法的含义是，如果变量 `varname` 存在且不为空，则返回它的值，否则将它设为 `word` ，并且返回 `word` 。它的目的是设置变量的默认值，比如 `${count:=0}` 表示变量 `count` 不存在时返回 `0` ，且将 `count` 设为 `0` 。

```
${varname:+word}
```

上面语法的含义是，如果变量名存在且不为空，则返回 `word` ，否则返回空值。它的目的是测试变量是否存在，比如 `${count:+1}` 表示变量 `count` 存在时返回 `1` （表示 `true` ），否则返回空值。

```
${varname:?message}
```

上面语法的含义是，如果变量 `varname` 存在且不为空，则返回它的值，否则打印出 `varname: message` ，并中断脚本的执行。如果省略了 `message` ，则输出默认的信息“parameter null or not set.”。它的目的是防止变量未定义，比如 `${count:?"undefined!"}` 表示变量 `count` 未定义时就中断执行，抛出错误，返回给定的报错信息 `undefined!` 。

上面四种语法如果用在脚本中，变量名的部分可以用数字 `1` 到 `9` ，表示脚本的参数。

```
filename=${1:?"filename missing."}
```

上面代码出现在脚本中， `1` 表示脚本的第一个参数。如果该参数不存在，就退出脚本并报错。

8. declare 命令

`declare` 命令可以声明一些特殊类型的变量，为变量设置一些限制，比如声明只读类型的变量和整数类型的变量。

它的语法形式如下。

```
declare OPTION VARIABLE=value
```

`declare` 命令的主要参数（OPTION）如下。

- `-a` ：声明数组变量。
- `-f` ：输出所有函数定义。
- `-F` ：输出所有函数名。
- `-i` ：声明整数变量。
- `-l` ：声明变量为小写字母。
- `-p` ：查看变量信息。
- `-r` ：声明只读变量。
- `-u` ：声明变量为大写字母。
- `-x` ：该变量输出为环境变量。

`declare` 命令如果用在函数中，声明的变量只在函数内部有效，等同于 `local` 命令。

不带任何参数时， `declare` 命令输出当前环境的所有变量，包括函数在内，等同于不带有任何参数的 `set` 命令。

```
$ declare
```

（1） `-i` 参数

`-i` 参数声明整数变量以后，可以直接进行数学运算。

```
$ declare -i val1=12 val2=5
$ declare -i result
$ result=val1*val2
$ echo $result
60
```

上面例子中，如果变量 `result` 不声明为整数， `val1*val2` 会被当作字面量，不会进行整数运算。另外， `val1` 和 `val2` 其实不需要声明为整数，因为只要 `result` 声明为整数，它的赋值就会自动解释为整数运算。

注意，一个变量声明为整数以后，依然可以被改写为字符串。

```
$ declare -i var=12
$ var=foo
$ echo $var
0
```

上面例子中，变量 `var` 声明为整数，覆盖以后，Bash 不会报错，但会赋以不确定的值，上面的例子中可能输出0，也可能输出的是3。

（2） `-x` 参数

`-x` 参数等同于 `export` 命令，可以输出一个变量为子 Shell 的环境变量。

```
$ declare -x foo
# 等同于
$ export foo
```

（3） `-r` 参数

`-r` 参数可以声明只读变量，无法改变变量值，也不能 `unset` 变量。

```
$ declare -r bar=1

$ bar=2
bash: bar: 只读变量
```

```
$ echo $?
1

$ unset bar
bash: bar: 只读变量
$ echo $?
1
```

上面例子中，后两个赋值语句都会报错，命令执行失败。

（4） -u 参数

-u 参数声明变量为大写字母，可以自动把变量值转成大写字母。

```
$ declare -u foo
$ foo=upper
$ echo $foo
UPPER
```

（5） -l 参数

-l 参数声明变量为小写字母，可以自动把变量值转成小写字母。

```
$ declare -l bar
$ bar=LOWER
$ echo $bar
lower
```

（6） -p 参数

-p 参数输出变量信息。

```
$ foo=hello
$ declare -p foo
declare -- foo="hello"
$ declare -p bar
bar: 未找到
```

上面例子中， declare -p 可以输出已定义变量的值，对于未定义的变量，会提示找不到。

如果不提供变量名， `declare -p` 输出所有变量的信息。

```
$ declare -p
```

(7) `-f` 参数

`-f` 参数输出当前环境的所有函数，包括它的定义。

```
$ declare -f
```

(8) `-F` 参数

`-F` 参数输出当前环境的所有函数名，不包含函数定义。

```
$ declare -F
```

9. readonly 命令

`readonly` 命令等同于 `declare -r` ，用来声明只读变量，不能改变变量值，也不能 `unset` 变量。

```
$ readonly foo=1
$ foo=2
bash: foo: 只读变量
$ echo $?
1
```

上面例子中，更改只读变量 `foo` 会报错，命令执行失败。

`readonly` 命令有三个参数。

- `-f` ：声明的变量为函数名。
- `-p` ：打印出所有的只读变量。
- `-a` ：声明的变量为数组。

10. let 命令

`let` 命令声明变量时，可以直接执行算术表达式。

```
$ let foo=1+2
$ echo $foo
3
```

上面例子中， `let` 命令可以直接计算 `1 + 2` 。

`let` 命令的参数表达式如果包含空格，就需要使用引号。

```
$ let "foo = 1 + 2"
```

`let` 可以同时对多个变量赋值，赋值表达式之间使用空格分隔。

```
$ let "v1 = 1" "v2 = v1++"
$ echo $v1,$v2
2,1
```

上面例子中， `let` 声明了两个变量 `v1` 和 `v2` ，其中 `v2` 等于 `v1++` ，表示先返回 `v1` 的值，然后 `v1` 自增。

这种语法支持的运算符，参考《Bash 的算术运算》一章。

引号和转义　　字符串操作

联系：contact@wangdoc.com