

4 Shell变量进阶

Shell变量进阶

一、Shell变量数据的输入输出

1.1 Shell变量数据的输出

在Bash Shell中常用 `echo` 命令实现数据的输出功能。

功能描述：`echo` 命令向标准输出输出字符串并在行末默认加上换行符。可以实现更复杂的输出格式控制。

`echo` 命令的语法格式为：`echo` [选项] 参数

注意：这里的参数可以是字符串和变量的组合。参数个数不限。

选项	说明
<code>-n</code>	不换行输出内容
<code>-e</code>	解析转义字符

常用转义字符如下。

转义字符	含义
<code>\n</code>	换行
<code>\r</code>	回车
<code>\t</code>	制表符
<code>\b</code>	退格
<code>\v</code>	纵向制表符

```
1 # -n选项使echo命令输出不换行
2 #; 为命令分隔符, 可在一行输入多个命令
3 [root@Shell ~]# echo a;echo b
4 a
5 b
6 [root@Shell ~]# echo -n a;echo b
7 ab
8 [root@Shell ~]# echo -n a;echo -n b
9 ab[root@Shell ~]#
10 # -e选项支持输出转义字符
11 [root@Shell ~]# echo a\tb
12 atb
13 [root@Shell ~]# echo "a\tb"
14 a\tb
15 [root@Shell ~]# echo -e "a\tb"
16 a      b
17 #字符与变量组合时, 注意变量尽量用${}引用。
18 [root@Shell ~]# c=123
19 [root@Shell ~]# echo a$c
20 a123
21 [root@Shell ~]# echo a$cb
22 a
23 [root@Shell ~]# echo a${c}b
24 a123b
```

⚠ **注意：** `;` 为命令分隔符，可在一行输入多个命令。

💬 **扩展：** 使用 `printf` 命令也可以实现复杂的输出控制。

1.2 Shell变量的引用（难点）

字符串可以由单引号 `' '` 包围，也可以由双引号 `" "` 包围，也可以不用引号。它们之间是有区别的：

- 由单引号 `' '` 包围的字符串，也被称为**全引用**，所有在单引号中的字符都只能代表其作为字符的**字面意义**。

任何字符都会**原样输出**，在其中**使用变量是无效的**。

字符串中不能出现单引号，即使对单引号进行转义也不行。

- 由双引号 `" "` 包围的字符串，也被称为**部分引用**。在这种引用方式中，`$` 符

号、反引号 ```、转义符 (`\`) 这3种特殊字符依然会被解析为特殊意义。如果其中包含了某个变量，那么该变量会被解析（得到该变量的值），而不是原样输出。

字符串中可以出现双引号，只要它被转义了就行。

- 不被引号包围的字符串

不被引号包围的字符串中出现变量时也会被解析，这一点和双引号 `" "` 包围的字符串一样。

字符串中不能出现空格，否则空格后边的字符串会作为其他变量或者命令解析。

Shell | 复制代码

```
1 [root@Shell ~]# c=123
2 [root@Shell ~]# echo ac
3 ac
4 [root@Shell ~]# echo a$c
5 a123
6 [root@Shell ~]# echo a$c d
7 a123 d
8 [root@Shell ~]# echo 'a$c d'
9 a$c d
10 [root@Shell ~]# echo "a$c d"
11 a123 d
12
13 [root@shell ~]# echo '\t'
14 \t
15 [root@shell ~]# echo '`ls`'
16 `ls`
17 [root@shell ~]# echo `ls`
18 anaconda-ks.cfg
19 [root@shell ~]# echo "`ls`"
20 anaconda-ks.cfg
21
```

⚠ 总结： ``` 原样输出； `"` 既可以解析变量，也可以包含空格；不带引号功能类似双引号，但是不能有空格。

1.3 Shell变量数据的输入

在Shell脚本中，可以通过 `read` 命令从标准输入读取单行数据，并为变量赋值。

命令格式为: `read [选项] 变量名`

选项	说明
<code>-p</code>	显示提示信息
<code>-t</code>	设置读入数据的超时时间
<code>-n</code>	设置读取n个字符后结束。默认会读取标准输入的一整行内容。
<code>-r</code>	支持读取 <code>\</code> 。默认理解 <code>\</code> 为特殊符号。
<code>-s</code>	静默模式。即不显示标准输入的内容。

案例：从键盘读入赋值

▼

Bash | 复制代码

```
1 [root@Shell ~]# read -p "请为变量赋值，回车结束" a
2 请为变量赋值，回车结束abc
3 [root@Shell ~]# echo $a
4 abc
5 [root@Shell ~]# read b
6 adfad
7 [root@Shell ~]# echo $b
8 adfad
9 [root@Shell ~]# read -s c
10 [root@Shell ~]# echo $c
11 ok
12 [root@Shell ~]# read -n1 a
13 a[root@Shell ~]# echo $a
14 a
15 [root@Shell ~]# read -t1 b
```

案例：用户登录

```

1 [root@Shell ~]# vi user.sh
2 read -p "username:" user
3 read -s -p "password:" password
4 echo
5 echo $user $password
6 [root@Shell ~]# bash ./user.sh
7 username:root
8 password:
9 root abc

```

1.4 Shell变量的替换

命令替换是指将**命令的标准输出**作为值**赋给某个变量**。这种赋值方法可以直接处理上一个命令产生的数据，这也是Shell编程中使用非常频繁的功能。

Shell中有两种方式可以完成命令替换。

- 使用一对反引号包围命令。格式为：`命令`
- 使用括号包围命令，并在前面加上\$。格式为：\$(命令)

案例：按日期命名文件

在生产环境中，把命令的结果作为变量的内容进行赋值的方法，在脚本开发时很常见，如按天打包网站的站点目录程序，生成不同的文件名。

```

1 # 生成日期字符串
2 [root@Shell ~]# date +%F
3 2021-01-03
4 [root@Shell ~]# cmd1=`date +%F`
5 [root@Shell ~]# cmd2=$(date +%F)
6 [root@Shell ~]# echo ${cmd1}.tar.gz
7 2021-01-03.tar.gz
8 [root@Shell ~]# echo ${cmd2}.tar.gz
9 2021-01-03.tar.gz

```

二、Shell变量子串操作

Shell提供了一些可以直接对变量进行操作的表达式（**参数扩展**：Parameter Expansion）。通过这些表达式，可以**删除、替换和替代变量中的部分内容**。这种操作的优势在于可以简化代码并提高可读性。

2.1 Shell变量切片

Shell变量支持切片操作(按位置提取子串)，字符串中最左边的位置为 0。

切片时需要两个参数：起始位置，截取长度。

格式	说明
<code>\${变量名}</code>	返回变量值。
<code>\${#变量名}</code>	返回变量值的长度。
<code>\${变量名:起始位置}</code>	从变量值的左侧 起始位置 处开始截取子串，直到最后。
<code>\${变量名:起始位置:子串长度}</code>	从变量值的左侧 起始位置 处开始截取子串，长度为 子串长度 。
<code>\${变量名:(-起始位置)}</code>	从变量值的右侧 起始位置 处开始截取子串，直到最后。
<code>\${变量名:(-起始位置):子串长度}</code>	从变量值的右侧 起始位置 处开始截取子串，长度为 子串长度 。

```
1 [root@Shell ~]# a="Practice makes perfect"
2 [root@Shell ~]# echo ${a}
3 Practice makes perfect
4 # 显示字符串长度
5 [root@Shell ~]# echo ${#a}
6 22
7 # 从左边位置为2的字符开始截取
8 [root@Shell ~]# echo ${a:2}
9 actice makes perfect
10 # 从左边位置为2的字符开始截取2个字符
11 [root@Shell ~]# echo ${a:2:2}
12 ac
13 # 从左边位置为2的字符开始截取5个字符
14 [root@Shell ~]# echo ${a:2:5}
15 actic
16 # 先从右边截取2个字符，从左边第0位置截取
17 [root@Shell ~]# echo ${a:-2}
18 ct
19 [root@Shell ~]# echo ${a:(-2)}
20 ct
21 [root@Shell ~]# echo ${a:0-2}
22 ct
23 # 先从右边截取10个字符，从左边第0位置截取
24 [root@Shell ~]# echo ${a:0-10}
25 es perfect
26 [root@Shell ~]# echo ${a:1-10}
27 s perfect
28 [root@Shell ~]# echo ${a:2-10}
29 perfect
30 [root@Shell ~]# echo ${a:3-10}
31 perfect
32 [root@Shell ~]# echo ${a:4-10}
33 erfect
34 [root@Shell ~]# echo ${a:5-10}
35 rfect
36 # 从右边截取7个字符，然后再截取最左边的5个字符
37 [root@Shell ~]# echo ${a:0-7:5}
38 perfe
39 # 从右边截取7个字符，然后再从子串的1位置截取5个字符
40 [root@Shell ~]# echo ${a:1-7:5}
41 erfec
```

这里需要强调两点：

- 从左边开始截取时，起始位置是 `0`；从右边开始截取时，起始位置是 `1`。

截取方向不同，起始位置也不同。

- 不管从哪边开始截取，**截取方向都是从左到右**。

2.2 删除Shell变量子串

删除Shell变量子串的表达式如下表所示。

格式	说明
<code>\${变量名#模式}</code>	从左向右搜索变量值中符合 <code>模式</code> 的子串，并删除符合模式的最短子串。
<code>\${变量名##模式}</code>	从左向右搜索变量值中符合 <code>模式</code> 的子串，并删除符合模式的最长子串。
<code>\${变量名%模式}</code>	从右向左搜索变量值中符合 <code>模式</code> 的子串，并删除符合模式的最短子串。
<code>\${变量名%%模式}</code>	从右向左搜索变量值中符合 <code>模式</code> 的子串，并删除符合模式的最长子串。

⚠ 在删除、替换等搜索操作中经常用到通配符。

通配符 `*` 表示匹配任意多个字符，通配符 `?` 表示匹配任意1个字符。


```

1  # 定义变量
2  [root@Shell ~]# file=ab/ac/ad
3  [root@Shell ~]# echo $file
4  ab/ac/ad
5  # 从左向右删除符合条件的最短子串, 此处为ab/
6  [root@Shell ~]# echo ${file#*/}
7  ac/ad
8  # 从左向右删除符合条件的最长子串, 此处为ab/ac/
9  [root@Shell ~]# echo ${file##*/}
10 ad
11 # 从右向左删除符合条件的最短子串, 此处为/ad
12 [root@Shell ~]# echo ${file%/*}
13 ab/ac
14 # 从右向左删除符合条件的最长子串, 此处为/ac/ad
15 [root@Shell ~]# echo ${file%%/*}
16 ab
17
18 [root@Shell ~]# echo ${file#*a}
19 b/ac/ad
20 [root@Shell ~]# echo ${file##*a}
21 d

```

总结

- `#` 表示从左向右匹配, `%` 表示从右向左匹配。
- 1个符号表示匹配最短子串, 2个符号表示匹配最长子串。

2.3 替换Shell变量子串

替换Shell变量子串的表达式如下表所示。

格式	说明
<code>\${变量名/模式/新字符串}</code>	若变量内容匹配【模式】则第一个匹配的内容会被【新字符串】替换
<code>\${变量名//模式/新字符串}</code>	若变量内容匹配【模式】则全部匹配的内容会被【新字符串】替换

```

1 ▾ [root@Shell ~]# var=abca
2 ▾ [root@Shell ~]# echo $var
3  abca
4  #替换第一个a
5 ▾ [root@shell ~]# echo ${var/a/x}
6  xbca
7  #使用通配符*, 匹配a前的所有字符
8 ▾ [root@shell ~]# echo ${var/*a/x}
9  x
10 #使用通配符?, 匹配a前的一个字符
11 ▾ [root@shell ~]# echo ${var/?a/x}
12  abx
13 #使用通配符?, 匹配a前的两个字符
14 ▾ [root@shell ~]# echo ${var/??a/x}
15  ax
16 #替换所有a
17 ▾ [root@shell ~]# echo ${var//a/x}
18  xbcx
19 #将二级域名baidu替换为sina
20 ▾ [root@Shell ~]# url=www.baidu.com
21 ▾ [root@Shell ~]# echo ${url/baidu/sina}
22  www.sina.com

```

2.4 为Shell变量设置默认值（扩展）

在某些情况下，给一些变量设置默认值是比较有意义的。

例如，在连接数据库时，需要使用端口，这个端口可以是预先设置的具体端口，也可以是用户输入的端口。假如用户没有输入具体的端口号，脚本中就使用预先设置的端口。

格式	说明
<code>\${变量名:-新字符串}</code>	<p>若变量值为空或未赋值，则输出【新字符串】。</p> <p>若变量值不为空，则输出变量的值。</p> <p>注意：原变量的值不变。</p>

<code>\${变量名-新字符串}</code>	<p>若变量未赋值，则输出【新字符串】。</p> <p>若变量值不为空，则输出变量的值。</p> <p>注意：原变量的值不变。</p>
<code>\${变量名:=新字符串}</code>	<p>若变量值为空或未赋值，则先将【新字符串】赋值给变量，再返回变量值。</p> <p>若变量值不为空，则输出变量的值。</p>
<code>\${变量名=新字符串}</code>	<p>若变量未赋值，则先将【新字符串】赋值给变量，再返回变量值。</p> <p>若变量值不为空，则输出变量的值。</p>
<code>\${变量名:?新字符串}</code>	<p>若变量值为空或未赋值，则【新字符串】作为标准错误输出。</p> <p>若变量值不为空，则输出变量的值。</p> <p>注意：原变量的值不变。</p>
<code>\${变量名?新字符串}</code>	<p>若变量未赋值，则【新字符串】作为标准错误输出，否则输出变量的值。</p> <p>若变量值不为空，则输出变量的值。</p> <p>注意：原变量的值不变。</p>
<code>\${变量名:+新字符串}</code>	<p>若变量值为空或未赋值，则什么都不做。</p> <p>若变量值不为空，则输出变量的值。</p> <p>注意：原变量的值不变。</p>
<code>\${变量名+新字符串}</code>	<p>若变量未赋值，则什么都不做。</p> <p>若变量值为空，则输出【新字符串】</p> <p>若变量值不为空，则输出【新字符串】。</p> <p>注意：原变量的值不变。</p>

案例：演示 `${变量名:-新字符串}` 和 `${变量名-新字符串}`

```

1  # port变量未赋值
2  [root@Shell ~]# unset port
3  # port未赋值, 输出新字符串
4  [root@Shell ~]# echo ${port:-3306}
5  3306
6  # port变量值不变
7  [root@Shell ~]# echo ${port}
8
9  # port变量未赋值, 因此输出新字符串
10 [root@Shell ~]# echo ${port-3306}
11 3306
12
13 # port变量赋值为空
14 [root@shell ~]# port=""
15 # 无输出
16 [root@shell ~]# echo ${port-3306}
17
18 # 输出新字符串
19 [root@shell ~]# echo ${port:-3306}
20 3306
21
22 # port变量值不为空, 输出port变量值
23 [root@Shell ~]# port=3307
24 [root@Shell ~]# echo ${port:-3306}
25 3307
26 [root@shell ~]# echo ${port-3306}
27 3307
28 [root@shell ~]# echo ${port}
29 3307

```

这个变量的功能可以解决变量没有定义的问题，并确保没有定义的变量始终有值。

案例：演示 `${变量名:=新字符串}` 和 `${变量名=新字符串}`

```

1  # port变量未赋值
2  [root@Shell ~]# unset port
3  [root@Shell ~]# echo ${port:=3306}
4  3306
5  # 注意! port的值发生了变化
6  [root@Shell ~]# echo ${port}
7  3306
8  # port变量未赋值
9  [root@shell ~]# unset port
10 [root@shell ~]# echo ${port=3306}
11 3306
12 [root@shell ~]# echo ${port}
13 3306
14
15 # port变量为空
16 [root@shell ~]# port=""
17 [root@shell ~]# echo ${port:=3306}
18 3306
19 [root@shell ~]# echo ${port}
20 3306
21 [root@shell ~]# port=""
22 [root@shell ~]# echo ${port=3306}
23
24 [root@shell ~]# echo ${port}
25
26 # port变量值不为空
27 [root@Shell ~]# port=3307
28 [root@Shell ~]# echo ${port:=3306}
29 3307
30 [root@Shell ~]# echo ${port=3306}
31 3307
32 [root@Shell ~]# echo ${port}
33 3307
34

```

这个变量的功能可以解决变量没有定义的问题，并确保没有定义的变量始终有值。

案例：演示 `${变量名:?新字符串}` 和 `${变量名?新字符串}`

```

1  # 取消变量，确保port变量未赋值
2  [root@Shell ~]# unset port
3  [root@Shell ~]# echo ${port}
4
5  # 注意，输出错误
6  [root@Shell ~]# echo ${port:?3306}
7  -bash: port: 3306
8  # port变量值不变
9  [root@Shell ~]# echo ${port}
10
11 # port变量未赋值
12 [root@shell ~]# unset port
13 [root@shell ~]# echo ${port?3306}
14 -bash: port: 3306
15 # port变量为空
16 [root@shell ~]# port=""
17 [root@shell ~]# echo ${port?3306}
18
19
20 [root@Shell ~]# port=3307
21 [root@Shell ~]# echo ${port:?3306}
22 3307
23 [root@Shell ~]# echo ${port?3306}
24 3307
25 [root@Shell ~]# echo ${port}
26 3307

```

本例的用法可以用于设定由于变量未定义而报错的具体内容。

案例：演示 `${变量名:+新字符串}` 和 `${变量名+新字符串}`

```


1  # port变量未赋值
2  [root@shell ~]# unset port
3  [root@shell ~]# echo ${port:+3306}
4
5  [root@shell ~]# echo ${port+3306}
6
7  [root@shell ~]# echo ${port}
8  # port变量为空值
9  [root@shell ~]# port=""
10 [root@shell ~]# echo ${port:+3306}
11
12 [root@shell ~]# echo ${port+3306}
13 3306
14 [root@shell ~]# echo ${port}
15
16 # port变量值不为空
17 [root@shell ~]# port=3307
18 [root@shell ~]# echo ${port:+3306}
19 3306
20 [root@shell ~]# echo ${port+3306}
21 3306
22 [root@shell ~]# echo ${port}
23 3307

```

 **思考：想想这四种形式有什么差异？**

变量为空值或未赋值时有什么行为？

变量值不会空时，输出什么值？原变量有什么变化？

 可以使用 `man bash` 命令查看帮助文件，`Parameter Expansion` 章节详细介绍了参数扩展的语法。

三、Shell变量的算术运算

在Linux的Shell中，**变量值的类型默认是字符串**，**不能直接进行运算**。

如果需要对Shell变量进行运算，需要使用特殊方法。

算术运算符	说明
-------	----

+, -	加法（或正号）、减法（或负号）
*, /, %	乘法、除法、取余（取模）
**	幂运算
++, --	自增和自减，可以放在变量的前面也可以放在变量的后面
=, +=, -=, *=, /=, %=	赋值运算符，例如 a+=1 相当于 a=a+1, a-=1 相当于 a=a-1

3.1 expr 数值运算命令

`expr` 命令既可以用于**整数运算**，也可以用于相关**字符串长度、匹配等**运算处理。

`expr` 命令的语法格式为 `expr 表达式`。

▼

Shell | 复制代码

```

1 ▾ [root@Shell ~]# n1=10
2 ▾ [root@Shell ~]# n2=20
3 ▾ [root@Shell ~]# expr $n1 + $n2
4   30
5 ▾ [root@Shell ~]# expr $n1+$n2
6   10+20
7 ▾ [root@Shell ~]# expr $n1 \* $n2
8   200

```

在使用 `expr` 时，需要注意运算符及用于计算的数字**两边必须有空格**，否则会执行失败。

另外，`expr` 也支持**乘号运算**，在**使用乘号运算时必须用反斜线转义**，因为Shell可能将其误解为 `*` 号。

3.2 (()) 算术扩展

算术扩展 `(())` 的作用是进行**整数**运算和数值比较，其效率很高，用法也非常灵活。

算术扩展格式为 `((表达式))`。

括号内部两侧可以有空格，也可省略空格。

需要直接输出运算表达式的运算结果时，可以在 `((表达式))` 前加 `$` 符。

```
Shell | 复制代码
1 [root@Shell ~]# n1=10
2 [root@Shell ~]# n2=20
3 [root@Shell ~]# sum=$((n1+n2))
4 [root@Shell ~]# echo $sum
5 30
```

3.3 `let` 命令执行表达式

`let` 命令用于执行一个或多个表达式，计算中变量前不需要加上 `$`。

`let` 命令的功能等价于算术扩展。运算符两端如果有空格需要用双引号括起来。

`let` 命令的语法格式为：`let 赋值表达式`。

```
Bash | 复制代码
1 [root@Shell ~]# n1=10
2 [root@Shell ~]# n2=20
3 [root@Shell ~]# let sum2=n1+n2
4 [root@Shell ~]# echo $sum2
5 30
6 [root@shell ~]# let sum3=n1 + n2
7 -bash: let: +: syntax error: operand expected (error token is "+")
8 [root@shell ~]# let "sum3=n1 + n2"
9 [root@shell ~]# echo $sum3
10 30
```

3.4 `bc` 命令执行浮点数运算

`Bash` 内置了对整数四则运算的支持，但是并不支持浮点运算，而 `bc` 命令可以很方便的进行浮点运算。

`bc` 命令在很多Linux发行版中需要自己安装 `yum install bc -y`。

```
Shell | 复制代码
1
2 [root@Shell ~]# echo "1.212*3" | bc
3 3.636
```

`bc` 命令还是一个简易的交互式计算器，具体帮助见 `man bc`。

算术运算总结：

- 用于整数运算的方法有 `expr`、`(())` 和 `let`，其中 `let` 效率较高，推荐使用。
- 使用 `bc` 命令对小数进行运算。

小结

- Shell变量的输入输出：`echo`、`read`、变量引用 `""` `' '`、变量替换 `$()` 或 ```
- Shell变量的子串操作：切片、删除、替换
- Shell变量的算术运算：`let`、`bc`

课程目标

- 知识目标：掌握shell变量输入输出和算术运算的基础知识。
- 技能目标：能够根据实际需求通过 `read` 命令、变量引用、变量替换完成变量操作。

课外拓展

- 进一步了解Shell变量输入输出和算术运算的其他方法

参考资料

- bash帮助：`man bash`