Table of Contents *generated with DocToc*

SHELL变量

定义变量

```
variable=value
variable='value'
variable="value"
```

> - 赋值号`=`的周围不能有空格
> - 在 shell 中，每一个变量的值都是字符串，无论你给变量赋值时有没有使用引号，值都会以字符串的形式存储
> - shell 在默认情况下不会区分变量类型，即使你将整数和小数赋值给变量，它们也会被视为字符串
> - 如果 value 不包含任何空白符（例如空格、Tab 缩进等），那么可以不使用引号
> - 如果 value 包含了空白符，那么就必须使用引号包围起来

使用变量

> 使用一个定义过的变量，只要在变量名前面加`$`符号即可

```
$ name="wnavy"
$ echo $name
wnavy
$ echo ${name}
wnavy
```

修改变量的值

> - 第二次对变量赋值时不能在变量名前加`$`，只有在使用变量时才能加`$`

```
$ name="wnavy"
$ name="haijun"
$ echo "my name is $name"
my name is haijun
```

单引号和双引号

> - 以单引号`''`包围变量的值时，单引号里面是什么就输出什么，即使内容中有变量和命令（命令需要反引起来）也会把它们原样输出。这种方式比较适合定义显示纯字符串的情况，即不希望解析变量、命令等的场景
> - 以双引号`""`包围变量的值时，输出时会先解析里面的变量和命令，而不是把双引号中的变量名和命令原样输出。这种方式比较适合字符串中附带有变量和命令并且想将其解析后再输出的变量定义

```
$ name="wnavy"
$ name="haijun"
$ echo "my name is $name"
my name is haijun
$ echo 'my name is $name'
my name is $name
```

将命令的结果赋值给变量

```
$ kernel_release=`uname -r`
$ echo $kernel_release
4.15.0-45-generic

$ operating_system=$(uname -o)
$ echo $operating_system
GNU/Linux
```

只读变量

```
$ operating_system=$(uname -o)
$ echo $operating_system
GNU/Linux
$ readonly operating_system
$ operating_system=$(uname -a)
-bash: operating_system: readonly variable
```

删除变量

```
$ kernel_release=`uname -r`
$ echo $kernel_release
4.15.0-45-generic
$ unset kernel_release
$ echo $kernel_release

```

特殊变量

变量	含义
$0	当前脚本的文件名
$n（n≥1）	传递给脚本或函数的参数。n 是一个数字，表示第几个参数。例如，第一个参数是 $1，第二个参数是 $2
$#	传递给脚本或函数的参数个数
$*	传递给脚本或函数的所有参数
$@	传递给脚本或函数的所有参数。当被双引号 " " 包含时，$@ 与 $* 稍有不同
$?	上个命令的退出状态，或函数的返回值
$$	当前 Shell 进程 ID。对于 Shell 脚本，就是这些脚本所在的进程 ID

> test.sh

```bash
#!/bin/bash

test_func() {
    echo "First Parameter of <test_func> : $1"
    echo "Second Parameter of <test_func> : $2"
}

test_func "Hello" "World"

echo "Process ID: $$"
echo "File Name: $0"
echo "First Parameter : $1"
echo "Second Parameter : $2"
echo "All parameters 1: $@"
echo "All parameters 2: $*"
echo "Total: $#"
```

> output

```
First Parameter of <test_func> : Hello
Second Parameter of <test_func> : World
Process ID: 113026
File Name: ./test.sh
First Parameter :
Second Parameter :
All parameters 1:
All parameters 2:
Total: 0
```

字符串拼接

> test.sh

```
#!/bin/bash
name="Shell"
url="http://wnavy.com/"
str1=$name$url  #中间不能有空格
str2="$name $url"  #如果被双引号包围，那么中间可以有空格
str3=$name": "$url  #中间可以出现别的字符串
str4="$name: $url"  #这样写也可以
str5="${name}Script: ${url}index.html"  #这个时候需要给变量名加上大括号
echo $str1
echo $str2
echo $str3
echo $str4
echo $str5
```

> output

```
Shellhttp://wnavy.com/
Shell http://wnavy.com/
Shell: http://wnavy.com/
Shell: http://wnavy.com/
ShellScript: http://wnavy.com/index.html
```

> - 使用如下方式式可以获取字符串长度：

```
${#string_name}
```

```
$ str="Hello World!"
$ echo ${#str}
12
```

字符串截取

从指定位置开始截取

格式	说明
${string: start :length}	从 string 字符串的左边第 start 个字符开始，向右截取 length 个字符。
${string: start}	从 string 字符串的左边第 start 个字符开始截取，直到最后。
${string: 0-start :length}	从 string 字符串的右边第 start 个字符开始，向右截取 length 个字符。
${string: 0-start}	从 string 字符串的右边第 start 个字符开始截取，直到最后。
${string#*chars}	从 string 字符串第一次出现 *chars 的位置开始，截取 *chars 右边的所有字符。
${string##*chars}	从 string 字符串最后一次出现 *chars 的位置开始，截取 *chars 右边的所有字符。
${string%*chars}	从 string 字符串第一次出现 *chars 的位置开始，截取 *chars 左边的所有字符。
${string%%*chars}	从 string 字符串最后一次出现 *chars 的位置开始，截取 *chars 左边的所有字符。

从左边开始计数

> - ***如果想从字符串的左边开始计数，那么截取字符串的具体格式如下：***

```
${string: start: length}
```

> - 其中，string 是要截取的字符串，start 是起始位置（从左边开始，从 0 开始计数），length 是要截取的长度（省略的话表示直到字符串的末尾）。

```
$ url="http://www.wnavy.com/"
$ echo ${url: 7: 13}
www.wnavy.com
$ echo ${url: 7}
www.wnavy.com/
```

从右边开始计数

> - ***如果想从字符串的右边开始计数，那么截取字符串的具体格式如下：***

```
${string: 0-start :length}
```

> - 同上一种格式相比，这种格式仅仅多了`0-`，这是固定的写法，专门用来表示从字符串右边开始计数。
> - 从左边开始计数时，起始数字是 0（这符合程序员思维）；从右边开始计数时，起始数字是 1
> - 不管从哪边开始计数，截取方向都是从左到右。

```
$ url="http://www.wnavy.com/"
$ echo ${url: 0-10: 5}
wnavy
$ echo ${url: 0-10}
wnavy.com/
```

从指定字符（子字符串）开始截取

> - *这种截取方式无法指定字符串长度，只能从指定字符（子字符串）截取到字符串末尾*
> - *Shell 可以截取指定字符（子字符串）右边的所有字符，也可以截取左边的所有字符*

使用 # 号截取右边字符

> - ***使用`#`号可以截取指定字符（或者子字符串）右边的所有字符，具体格式如下：***

```
${string#*chars}
```

> - 其中，string 表示要截取的字符，chars 是指定的字符（或者子字符串），`*`是通配符的一种，表示任意长度的字符串。`*chars`连起来使用的意思是：忽略左边的所有字符，直到遇见 chars（chars 不会被截取）。

```
$ url="http://www.wnavy.com/index.html"
$ echo ${url#*:}
//www.wnavy.com/index.html
$ echo ${url#*//}
www.wnavy.com/index.html

# 遇到第一个匹配的字符（子字符串）就结束了
$ echo ${url#*/}
/www.wnavy.com/index.html

# 如果希望直到最后一个指定字符（子字符串）再匹配结束，可以使用##
$ echo ${url##*/}
index.html
```

使用 % 截取左边字符

> - ***使用`%`号可以截取指定字符（或者子字符串）左边的所有字符，具体格式如下：***

```
${string%chars*}
```

> - 请注意`*`的位置，因为要截取 chars 左边的字符，而忽略 chars 右边的字符，所以`*`应该位于 chars 的右侧。其他方面`%`和`#`的用法相同，这里不再赘述，仅举例说明：

```
$ url="http://www.wnavy.com/index.html"
$ echo ${url%.*}
http://www.wnavy.com/index
$ echo ${url%/*}
http://www.wnavy.com
$ echo ${url%%/*}
http:
```

SHELL 条件判断

IF的基本语法

```
if [ command ]; then
    # 符合该条件执行的语句
elif [ command ]; then
    # 符合该条件执行的语句
else
    # 符合该条件执行的语句
fi
```

文件/文件夹判断

```
# 常用的:
[ -a FILE ] # 如果 FILE 存在则为真。
[ -d FILE ] # 如果 FILE 存在且是一个目录则返回为真。
[ -e FILE ] # 如果 指定的文件或目录存在时返回为真。
[ -f FILE ] # 如果 FILE 存在且是一个普通文件则返回为真。
[ -r FILE ] # 如果 FILE 存在且是可读的则返回为真。
[ -w FILE ] # 如果 FILE 存在且是可写的则返回为真。（一个目录为了它的内容被访问必然是可执行的）
[ -x FILE ] # 如果 FILE 存在且是可执行的则返回为真。

# 不常用的:
[ -b FILE ] # 如果 FILE 存在且是一个块文件则返回为真。
[ -c FILE ] # 如果 FILE 存在且是一个字符文件则返回为真。
[ -g FILE ] # 如果 FILE 存在且设置了SGID则返回为真。
[ -h FILE ] # 如果 FILE 存在且是一个符号符号链接文件则返回为真。（该选项在一些老系统上无效）
[ -k FILE ] # 如果 FILE 存在且已经设置了冒险位则返回为真。
[ -p FILE ] # 如果 FILE 存并且是命令管道时返回为真。
[ -s FILE ] # 如果 FILE 存在且大小非0时为真则返回为真。
[ -u FILE ] # 如果 FILE 存在且设置了SUID位时返回为真。
[ -O FILE ] # 如果 FILE 存在且属有效用户ID则返回为真。
[ -G FILE ] # 如果 FILE 存在且默认组为当前组则返回为真。（只检查系统默认组）
[ -L FILE ] # 如果 FILE 存在且是一个符号连接则返回为真。
[ -N FILE ] # 如果 FILE 存在 and has been mod如果ied since it was last read则返回为真。
[ -S FILE ] # 如果 FILE 存在且是一个套接字则返回为真。
```

```
[ FILE1 -nt FILE2 ] # 如果 FILE1 比 FILE2 新，或者 FILE1 存在但是 FILE2 不存在则返回为真。
[ FILE1 -ot FILE2 ] # 如果 FILE1 比 FILE2 老，或者 FILE2 存在但是 FILE1 不存在则返回为真。
[ FILE1 -ef FILE2 ] # 如果 FILE1 和 FILE2 指向相同的设备和节点号则返回为真。
```

字符串判断

```
[ -z STRING ] # 如果STRING的长度为零则为真,即判断是否为空,空即是真;
[ -n STRING ] # 如果STRING的长度非零则为真,即判断是否为非空,非空即是真;
[ STRING1 ] # 如果字符串不为空则为真,与-n类似;
[ STRING1 = STRING2 ] # 如果两个字符串相同则为真;
[ STRING1 == STRING2 ] # 如果两个字符串相同则为真;
[ STRING1 != STRING2 ] # 如果字符串不相同则为真;
[ STRING1 < STRING2 ] # 如果 “STRING1”字典排序在“STRING2”前面则返回为真;
[ STRING1 > STRING2 ] # 如果 “STRING1”字典排序在“STRING2”后面则返回为真;
```

数值判断

```
[ INT1 -eq INT2 ] # INT1和INT2两数相等返回为真，=
[ INT1 -ne INT2 ] # INT1和INT2两数不等返回为真，<>
[ INT1 -gt INT2 ] # INT1大于INT2返回为真，>
[ INT1 -ge INT2 ] # INT1大于等于INT2返回为真，>=
[ INT1 -lt INT2 ] # INT1小于INT2返回为真，<
[ INT1 -le INT2 ] # INT1小于等于INT2返回为真，<=
```

逻辑判断

```
[ ! EXPR ] # 逻辑非，如果 EXPR 是false则返回为真。
[ EXPR1 -a EXPR2 ] # 逻辑与，如果 EXPR1 and EXPR2 全真则返回为真。
[ EXPR1 -o EXPR2 ] # 逻辑或，如果 EXPR1 或者 EXPR2 为真则返回为真。
[ ] || [ ] # 用OR来合并两个条件
[ ] && [ ] # 用AND来合并两个条件
```

其他判断

```
[ -t FD ] # 如果文件描述符 FD （默认值为1）打开且指向一个终端则返回为真
[ -o optionname ] # 如果shell选项optionname开启则返回为真
```

IF高级特性

双圆括号(())

- (()) 表示数学表达式

> 在判断命令中只允许在比较中进行简单的算术操作，而双圆括号提供更多的数学符号，而且在双圆括号里面的'>','<'号不需要转意。

```
echo $((1+1)) # 2
```

双方括号[[]]

- [[ ]] 表示高级字符串处理函数

> 双方括号中判断命令使用标准的字符串比较，还可以使用匹配模式，从而定义与字符串相匹配的正则表达式。

> 在shell中，[$a != 1 || $b = 2]是不允许出，要用[$a != 1] || [$b = 2]，而双括号就可以解决这个问题的，[[$a != 1 || $b = 2]]。又比如这个["$a" -lt "$b"]，也可以改成双括号的形式(("$a" < "$b"))

复杂逻辑判断

> - 对多个表达式进行逻辑运算

```
[ -z "$str1" ] || [ -z "$str2" ] # 正确
[ -z "$str1" -o -z "$str2" ] # 正确
[[ -z $str1 || -z $str2 ]] # 正确
[[ -z $str1 ]] || [[ -z $str2 ]] # 正确
[[ -z $str1 -o -z $str2 ]] # 错误! [[ ]] 剔除了 test 命令的-o和-a选项，只能使用 || 和 &&
```

数学计算

算术运算符

算术运算符	说明/含义
+、-	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
!、&&、||	逻辑非（取反）、逻辑与（and）、逻辑或（or）
<、<=、>、>=	比较符号（小于、小于等于、大于、大于等于）
==、!=、=	比较符号（相等、不相等；对于字符串，= 也可以表示相当于）
<<、>>	向左移位、向右移位
~、|、&、^	按位取反、按位或、按位与、按位异或
=、+=、-=、*=、/=、%=	赋值运算符，例如 a+=1 相当于 a=a+1，a-=1 相当于 a=a-1

数学计算命令

运算操作符/运算命令	说明
(( ))	用于整数运算，效率很高，**推荐使用。**
let	用于整数运算，和 (()) 类似。
$[]	用于整数运算，不如 (()) 灵活。
expr	可用于整数运算，也可以处理字符串。比较麻烦，需要注意各种细节，不推荐使用。
bc	Linux下的一个计算器程序，可以处理整数和小数。Shell 本身只支持整数运算，想计算小数就得使用 bc 这个外部的计算器。
declare -i	将变量定义为整数，然后再进行数学运算时就不会被当做字符串了。功能有限，仅支持最基本的数学运算（加减乘除和取余），不支持逻辑运算、自增自减等，所以在实际开发中很少使用。

举例

> - 默认情况下，Shell 不会直接进行算术运算，而是把+两边的数据（数值或者变量）当做字符串，把+当做字符串连接符
> - 在 (( )) 中使用变量无需加上`$`前缀，(( )) 会自动解析变量名
> - 可以在 (( )) 前面加上`$`符号获取 (( )) 命令的执行结果，也即获取整个表达式的值

```
$ a=23
$ b=$a+55
$ echo $b
23+55

$ a=23
$ let b=$a+55
$ echo $b
78

$ a=23
$ b=$((a+55))
$ echo $b
78
$ c=$((a+b))
$ echo $c
101
$ echo $((a+c))
124
$ echo $((d=a+c, e=b+d))
202
$ echo $d
124
$ echo $e
202
$ echo $((a>0 && b>0))
1
$ echo $((a>100 && b>100))
0
```

read

```
#!/bin/bash
read -p "please  input a score:"  score
echo  -e "your  score [$score] is judging by sys now"
if [ "$score" -ge "0" ]&&[ "$score" -lt "60" ];then
    echo  "sorry,you  are lost!"
elif [ "$score" -ge "60" ]&&[ "$score" -lt "85" ];then
    echo "just  soso!"
elif [ "$score" -le "100" ]&&[ "$score" -ge "85" ];then
     echo "good  job!"
else
     echo "input  score is wrong , the range is [0-100]!"
fi
```

case in

```
case expression in
    pattern1)
        statement1
        ;;
    pattern2)
        statement2
        ;;
    pattern3)
        statement3
        ;;
    ……
    *)
        statementn
esac
```

> case、in 和 esac 都是 Shell 关键字，expression 表示表达式，pattern 表示匹配模式。
>
> - expression 既可以是一个变量、一个数字、一个字符串，还可以是一个数学计算表达式，或者是 命令的执行结果，只要能够得到 expression 的值就可以。
> - pattern 可以是一个数字、一个字符串，甚至是一个简单的正则表达式。

> case 会将 expression 的值与 pattern1、pattern2、pattern3 逐个进行匹配：
>
> - 如果 expression 和某个模式（比如 pattern2）匹配成功，就会执行这模式（比如 pattern2）后面对应的所有语句（该语句可以有一条，也可以有多条），直到遇见双分号`;;`才停止；然后整个 case 语句就执行完了，程序会跳出整个 case 语句，执行 esac 后面的其它语句。
> - 如果 expression 没有匹配到任何一个模式，那么就执行`*)`后面的语句（`*`表示其它所有值），直到遇见双分号`;;`或者`esac`才结束。`*)`相当于多个 if 分支语句中最后的 else 部分。

while

> test.sh

```
#!/bin/bash
i=1
sum=0
while ((i <= 100))
do
    ((sum += i))
    ((i++))
done
echo "The sum is: $sum"
```

> output

```
The sum is: 5050
```

break

> test.sh

```
#!/bin/bash
i=0
sum=0
while true; do
    if ((i<=100)); then
        ((sum += i))
        ((i++))
    else
        break
    fi
done
echo "The sum is: $sum"
```

> output

```
The sum is: 5050
```

for

> test.sh

```
#!/bin/bash
sum=0
for ((i=1; i<=100; i++))
do
    ((sum += i))
done
echo "The sum is: $sum"
```

> output

```
The sum is: 5050
```

select in

- `#?` 用来提示用户输入菜单编号；`^D` 表示按下 Ctrl+D 组合键，它的作用是结束 select in 循环。
- select 是无限循环（死循环），输入空值，或者输入的值无效，都不会结束循环，只有遇到 break 语句，或者按下 Ctrl+D 组合键才能结束循环。

- select in 通常和 case in 一起使用，在用户输入不同的编号时可以做出不同的反应。

test.sh

```
#!/bin/bash
echo "What is your favourite OS?"
select name in "Linux" "Windows" "Mac OS" "UNIX" "Android"
do
    case $name in
        "Linux")
            echo "Linux是一个类UNIX操作系统，它开源免费，运行在各种服务器设备和嵌入式设备。"
            break
            ;;
        "Windows")
            echo "Windows是微软开发的个人电脑操作系统，它是闭源收费的。"
            break
            ;;
        "Mac OS")
            echo "Mac OS是苹果公司基于UNIX开发的一款图形界面操作系统，只能运行与苹果提供的硬件之上。"
            break
            ;;
        "UNIX")
            echo "UNIX是操作系统的开山鼻祖，现在已经逐渐退出历史舞台，只应用在特殊场合。"
            break
            ;;
        "Android")
            echo "Android是由Google开发的手机操作系统，目前已经占据了70%的市场份额。"
            break
            ;;
        *)
            echo "输入错误，请重新输入"
    esac
done
```

output

```
What is your favourite OS?
1) Linux
2) Windows
3) Mac OS
4) UNIX
5) Android
#? 1
Linux是一个类UNIX操作系统，它开源免费，运行在各种服务器设备和嵌入式设备。
```

SHELLE数组

- Shell 没有限制数组的大小，理论上可以存放无限量的数据。

> - Shell 是弱类型的，它并不要求所有数组元素的类型必须相同。
> - Shell 数组的长度不是固定的，定义之后还可以增加元素。
> - Shell 数组可以只给特定元素赋值。数组的长度是赋值的元素个数
> - Shell 数组元素的下标也是从 0 开始计数。
> - 获取数组中的元素要使用下标`[ ]`，下标可以是一个整数，也可以是一个结果为整数的表达式；下标必须大于等于 0。
> - 常用的 Bash Shell 只支持一维数组，不支持多维数组。

数组的定义和使用

> - 在 Shell 中，用括号`( )`来表示数组，数组元素之间用空格来分隔。
> - 定义数组的一般形式为：

```
array_name=(ele1  ele2  ele3 ... elen)
```

> *赋值号`=`两边不能有空格，必须紧挨着数组名和数组元素。*

获取数组元素的值，一般使用下面的格式：

```
${array_name[index]}
```

> *使用`@`或``可以获取数组中的所有元素。*

```
$ arry=(1 2 "3" "4" "Hello World")
$ echo ${arry[0]}
1
$ echo ${arry[2]}
3$ echo ${arry[4]}
Hello World
$ arry[5]="5"
$ echo ${arry[5]}
5
$ echo ${arry[*]}
1 2 3 4 Hello World 5
$ echo ${arry[@]}
1 2 3 4 Hello World 5

table=([2]=2 [5]=5 [100]=100) # 数组table长度为3
echo ${table[0]} # 元素0不存在

echo ${table[2]}
2
echo ${table[5]}
5
echo ${table[*]}
2 5 100
 echo ${table[@]}
```

```
2 5 100
```

获取数组长度

> 利用`@`或`*`，可以将数组扩展成列表，然后使用`#`来获取数组元素的个数，格式如下：

```
${#array_name[@]}
${#array_name[*]}
```

> 两种形式是等价的，选择其一即可。

```
$ table=([2]=2 [5]=5 [100]=100)
$ echo ${#table[*]}
3
$ table[0]=0
$ table[1]=1
$ table[2]=200
$ echo ${table[*]}
0 1 200 5 100
$ echo ${#table[*]}
5
```

> 如果某个元素是字符串，还可以通过指定下标的方式获得该元素的长度，如下所示：

```
${#arr[2]}
```

```
$ array=("Hello" "World" "!" "Hello World!")
$ echo ${#array[*]}
4
$ echo ${array[*]}
Hello World ! Hello World!
$ echo ${#array[0]}
5
$ echo ${#array[1]}
5
$ echo ${#array[2]}
1
$ echo ${#array[3]}
12
$ echo ${#array[4]} #元素4不存在
0
```

数组的拼接

> 先利用`@`或`*`，将数组扩展成列表，然后再合并到一起。具体格式如下：

```
array_new=(${array1[@]}  ${array2[@]})
array_new=(${array1[*]}  ${array2[*]})
```

> 两种方式是等价的，选择其一即可。

```
$ array1=("Hello" "World" "!")
$ array2=("Hello World!")
$ echo ${array1[*]}
Hello World !
$ echo ${#array1[*]}
3
$ echo ${array2[*]}
Hello World!
$ echo ${#array2[*]}
1
$ array=(${array1[*]} ${array2[*]})
$ echo ${array[*]}
Hello World ! Hello World!
$ echo ${#array[*]}
5
```

删除数组元素

> 使用 unset 关键字来删除数组元素，具体格式如下：

```
unset array_name[index]
```

如果不写下标，那么就是删除整个数组，所有元素都会消失。如下：

```
unset array_name
```

```
$ tbl=(1 2 3 4 5 6)
$ echo ${tbl[*]}
1 2 3 4 5 6
$ echo ${#tbl[*]}
6
$ unset tbl[0]
$ unset tbl[1]
$ echo ${tbl[*]}
3 4 5 6
$ echo ${#tbl[*]}
4
$ unset tbl
$ echo ${#tbl[*]}
0
```