

Shell字符串变量：格式介绍

目标

能够使用字符串的3种方式

掌握3种方式的区别

能够获取字符串的长度

介绍

字符串（String）就是一系列字符的组合。字符串是 Shell 编程中最常用的数据类型之一（除了数字和字符串，也没有其他类型了）

字符串的3种格式

1. 单引号方式
2. 双引号方式, 推荐
3. 不用引号方式

```
[root@itheima ~]# val='abc'
[root@itheima ~]# var3="abc"
[root@itheima ~]# var2=abc
[root@itheima ~]#
```

字符串的3种格式区别

1. 使用单引号 `' '` 的字符串：

任何字符都会原样输出，在拼接字符串中使用变量是无效的。

```
[root@itheima ~]# var1=abc
[root@itheima ~]# echo 'abc${var1}'
abc${var1}
[root@itheima ~]#
```

2. 由双引号 `" "` 包围的字符串：

其中包含了变量，那么该变量会被解析得到值，而不是原样输出。

字符串中还可以出现双引号的子字符串，但是需要转义。

```
[root@itheima ~]# var1=itheima
[root@itheima ~]# echo "you are welcome,\"${var1}\""
you are welcome,"itheima"
[root@itheima ~]#
```

3. 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命令解析。

```
[root@itheima ~]# var4=abc bcd dd
-bash: bcd: 未找到命令
```

```
[root@itheima ~]# var1=itheima
[root@itheima ~]# echo you are welcome,${var1}
you are welcome,itheima
[root@itheima ~]#
```

获取字符串的长度

语法

```
${#字符串变量名}
```

含义

获取字符串的字符长度

演示

```
[root@itheima ~]# var1=itheima
[root@itheima ~]# echo ${#var1}
7
[root@itheima ~]#
```

小结

能够使用字符串的3种方式

单引号方式

双引号方式[推荐]

不使用引号方式

掌握3种方式的区别

单引号, 原样输出, 不会解析里面的变量

双引号, 会解析里面的变量, 并且可以使用子双引号, 需要转义

不使用引号, 也会解析里面的变量, 但是不能含有空格, 空格后面的字符串会作为命令去执行

能够获取字符串的长度

```
${#var_name}
```

Shell字符串变量：拼接

目标

掌握Shell字符串拼接

字符串拼接方式

1. 无符号拼接
2. 双引号拼接
3. 混合拼接

演示

```
[root@itheima ~]# var1=itheima
[root@itheima ~]# var2=itcast
[root@itheima ~]# echo ${var1}${var2} ← 无符号拼接
itheima itcast
[root@itheima ~]# echo "${var1}${var2}" ← 双引号拼接
itheima itcast
[root@itheima ~]# echo ${var1}"&"${var2} ← 混合拼接
itheima&itcast
[root@itheima ~]# echo ${var1}'& '${var2} ← 混合拼接
itheima&itcast
```

小结

1. 无符号拼接
2. 双引号拼接
3. 混合拼接

Shell字符串变量：字符串截取

目标

掌握shell字符串截取的常用格式

语法

格式	说明
<code>\${变量名:start:length}</code>	从 string 字符串的左边第 start 个字符开始，向右截取 length 个字符。start从0开始
<code>\${变量名:start}</code>	从 string 字符串的左边第 start 个字符开始截取，直到最后。
<code>\${变量名:0-start:length}</code>	从 string 字符串的右边第 start 个字符开始，向右截取 length 个字符。start从1开始, 代表右侧第一个字符
<code>\${变量名:0-start}</code>	从 string 字符串的右边第 start 个字符开始截取，直到最后。
<code>\${变量名#*chars}</code>	从 string 字符串左边第一次出现 *chars 的位置开始，截取 *chars 右边的所有字符。
<code>\${变量名##*chars}</code>	从 string 字符串左边最后一次出现 *chars 的位置开始，截取 *chars 右边的所有字符。
<code>\${变量名%chars*}</code>	从 string 字符串右边第一次出现 chars* 的位置开始，截取 chars* 左边的所有字符。
<code>\${变量名%%chars*}</code>	从 string 字符串右边最后一次出现 chars* 的位置开始，截取 chars* 左边的所有字符

示例要求

字符串"welcome to itheima"

- # 从左侧第0个开始,向左截取2个字符
- # 从左侧第11个开始,向左截取所有字符
- # 从右侧第5个开始,向右截取2个字符
- # 截取左边第一次出现字符e右边的所有字符
- # 截取左边最后一次出现字符e右边的所有字符
- # 截取右边第一次出现字符e左边的所有字符
- # 截取右边最后一次出现字符e左边的所有字符

截取演示

从左侧第0个开始,向左截取2个字符

```
[root@itheima ~]# var1="welcome to itheima"
[root@itheima ~]# echo ${var1:0:2}
we
[root@itheima ~]#
```

从左侧第11个开始,向左截取所有字符

```
[root@itheima ~]# var1="welcome to itheima"
[root@itheima ~]# echo ${var1:11}
itheima
[root@itheima ~]#
```

从右侧第5个开始,向右截取2个字符

```
[root@itheima ~]# var1="welcome to itheima"
[root@itheima ~]# echo ${var1:0-5:2}
he
[root@itheima ~]#
```

截取左边第一次出现字符e右边的所有字符

```
[root@itheima ~]# echo ${var1#*e}
lcome to itheima
[root@itheima ~]#
```

截取左边最后一次出现字符e右边的所有字符

```
[root@itheima ~]# echo ${var1##*e}
ima
[root@itheima ~]#
```

截取右边第一次出现字符e左边的所有字符

```
[root@itheima ~]# echo ${var1%e*}
welcome to ith
[root@itheima ~]#
```

截取右边最后一次出现字符e左边的所有字符

```
[root@itheima ~]# echo ${var1%%e*}
w
```

小结

格式	说明
<code>\${变量名: start :length}</code>	从左边start位置开始截取length个字符长度, 从0开始
<code>\${变量名: start}</code>	从左边start位置开始截取到最后所有字符

Shell索引数组变量：定义-获取-拼接-删除

目标

1. 能够定义Shell数组
2. 能够获取Shell数组数据
3. 能够进行Shell数组的拼接, 删除

介绍

Shell 支持数组（Array），数组是若干数据的集合，其中的每一份数据都称为数组的元素。

注意Bash Shell 只支持一维数组，不支持多维数组。

数组的定义

语法

在 Shell 中，用括号（ ）来表示数组，数组元素之间用空格来分隔。语法为：

```
array_name=(item1 item2 ...) # 方式1
array_name=([索引下标1]=item1 [索引下标2]=item2 ...) # 方式2
```

注意，赋值号 = 两边不能有空格

演示

1.定义数字存储100,3,22,58,77,17,20

```
nums=(29 100 13 8 91 44)
```

2.Shell 是弱类型的，它并不要求所有数组元素的类型必须相同

```
arr=(20 56 "http://www.itcast.cn/")
```

Shell数组元素定义后不是固定的, 定义后还可以赋值

```
arr[6]=100
```

3.可以给指定元素赋值初始化

```
arr2=([0]=1 [2]=100 [4]=aa)
```

由于上面只赋值了3个元素, 所以数组的长度是3

```
[root@itheima ~]# nums=(29 100 13 8 91 44)
[root@itheima ~]# arr=(20 56 "http://www.itcast.cn/")
[root@itheima ~]# arr[6]=100
[root@itheima ~]# arr2=([0]=1 [2]=100 [4]=aa)
[root@itheima ~]#
```

数组的获取

语法

- 1.通过下标获取元素值,index从0开始

```
${arr[index]}
```

注意使用 `{ }`

- 2.获取值同时复制给其他变量

```
item=${arr[index]}
```

- 3.使用 `@` 或 `*` 可以获取数组中的所有元素

```
${arr[@]}  
${arr[*]}
```

- 4.获取数组的长度或个数

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

- 5.获取数组指定元素的字符长度

```
${#arr[索引]}
```

演示

```
[root@itheima ~]# nums=(29 100 13 8 91 44)
[root@itheima ~]# arr=(20 56 "http://www.itcast.cn/")
[root@itheima ~]# arr[6]=100
[root@itheima ~]# arr2=([0]=1 [2]=100 [4]=aa)
[root@itheima ~]# echo ${nums[0]}
29
[root@itheima ~]# echo ${nums[1]}
100
[root@itheima ~]# echo ${arr[1]}
56
[root@itheima ~]# echo ${arr[2]}
http://www.itcast.cn/
[root@itheima ~]# item=${arr[2]}
[root@itheima ~]# echo $item
http://www.itcast.cn/
[root@itheima ~]# echo ${arr[*]}
20 56 http://www.itcast.cn/ 100
[root@itheima ~]# echo ${arr[@]}
20 56 http://www.itcast.cn/ 100
[root@itheima ~]# echo ${arr2[@]}
1 100 aa
[root@itheima ~]# █
```

```
[root@itheima ~]# echo ${arr2[@]}
1 100 aa
[root@itheima ~]# echo ${#arr2[@]}
3
```

```
[root@itheima ~]# echo ${#arr2[*]}
3
```

```
[root@itheima ~]# echo ${arr[2]}
http://www.itcast.cn/
[root@itheima ~]# echo ${#arr[2]}
21
[root@itheima ~]# █
```

数组的拼接

介绍

所谓 Shell 数组拼接（数组合并），就是将两个数组连接成一个数组

语法

使用 `@` 和 `*` 获取数组所有元素之后进行拼接

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

演示


```
[root@itheima ~]# array1=(1 aa bb)
[root@itheima ~]# array2=(hello 123)
[root@itheima ~]# array_new=(${array1[@]} ${array2[@]})
[root@itheima ~]# array_new2=(${array1[*]} ${array2[*]})
[root@itheima ~]# echo ${array_new[@]}
1 aa bb hello 123
[root@itheima ~]# echo ${array_new2[*]}
1 aa bb hello 123
[root@itheima ~]#
```

数组的删除

介绍

删除数组指定元素数据和删除整个数组数据

语法

删除数组指定元素数据

```
unset array_name[index]
```

删除整个数组

```
unset array_name
```

演示

```
[root@itheima ~]# echo ${array_new[@]}
1 aa bb hello 123
[root@itheima ~]# unset ${array_new[1]}
[root@itheima ~]# echo ${array_new[@]}
1 aa bb hello 123
[root@itheima ~]# echo ${array_new[@]}
1 aa bb hello 123
[root@itheima ~]# unset array_new[1]
[root@itheima ~]# echo ${array_new[@]}
1 bb hello 123
[root@itheima ~]# unset array_new
[root@itheima ~]# echo ${array_new[@]}
```

小结

1.能够定义Shell数组

```
array_name=(item1 item2 ...)  
array_name=([索引1]=item1 [索引2]=item2 ...)
```

2.能够获取Shell数组数据

```
${array_name[索引下标]}  
${array_name[*]}  
${array_name[@]}
```

3.数组的拼接

```
array_newname=(${array_name1[*]} ${array_name2[*]} ...)
```

4.数组的删除

```
unset array_name[索引]  
unset array_name
```

Shell内置命令：alias设置别名

目标

- 1.理解内置命令的含义
- 2.能够使用alias内置命令进行给命令定义别名

内置命令介绍

Shell 内置命令，就是由 Bash Shell 自身提供的命令，而不是文件系统中的可执行脚本文件。

使用type 来确定一个命令是否是内置命令：

```
type 命令
```

```
[root@itheima ~]# type cd  
cd 是 shell 内嵌  
[root@itheima ~]# type ifconfig  
ifconfig 是 /usr/sbin/ifconfig  
[root@itheima ~]# type crontab  
crontab 是 /usr/bin/crontab  
[root@itheima ~]#
```

cd是内嵌命令
ifconfig是可执行文件
crontab是可执行文件

通常来说，内置命令会比外部命令执行得更快，执行外部命令时不但会触发磁盘 I/O，还需要 fork 出一个单独的进程来执行，执行完成后再退出。而执行内置命令相当于调用当前 Shell 进程的一个函数，还是在当前 Shell 环境进程内，减少了上下文切换。

内置命令列表

命令	说明
:	扩展参数列表，执行重定向操作
.	读取并执行指定文件中的命令（在当前 shell 环境中）
==alias==	为指定命令定义一个别名
bg	将作业以后台模式运行
bind	将键盘序列绑定到一个 readline 函数或宏
break	退出 for、while、select 或 until 循环
builtin	执行指定的 shell 内建命令
caller	返回活动子函数调用的上下文
cd	将当前目录切换为指定的目录
command	执行指定的命令，无需进行通常的 shell 查找
compgen	为指定单词生成可能的补全匹配
complete	显示指定的单词是如何补全的
compgpt	修改指定单词的补全选项
continue	继续执行 for、while、select 或 until 循环的下一迭代
==declare==	声明一个变量或变量类型。
dirs	显示当前存储目录的列表
disown	从进程作业表中删除指定的作业
==echo==	将指定字符串输出到 STDOUT
enable	启用或禁用指定的内建shell命令
eval	将指定的参数拼接成一个命令，然后执行该命令
exec	用指定命令替换 shell 进程
==exit==	强制 shell 以指定的退出状态码退出
export	设置子 shell 进程可用的变量
fc	从历史记录中选择命令列表
fg	将作业以前台模式运行
getopts	分析指定的位置参数
hash	查找并记住指定命令的全路径名
help	显示帮助文件

命令	说明
history	显示命令历史记录
jobs	列出活动作业
kill	向指定的进程 ID(PID) 发送一个系统信号
let	计算一个数学表达式中的每个参数
local	在函数中创建一个作用域受限的变量
logout	退出登录 shell
mapfile	从 STDIN 读取数据行，并将其加入索引数组
popd	从目录栈中删除记录
printf	使用格式化字符串显示文本
pushd	向目录栈添加一个目录
pwd	显示当前工作目录的路径名
==read==	从 STDIN 读取一行数据并将其赋给一个变量
readarray	从 STDIN 读取数据行并将其放入索引数组
readonly	从 STDIN 读取一行数据并将其赋给一个不可修改的变量
return	强制函数以某个值退出，这个值可以被调用脚本提取
set	设置并显示环境变量的值和 shell 属性
shift	将位置参数依次向下降一个位置
shopt	打开/关闭控制 shell 可选行为的变量值
source	读取并执行指定文件中的命令（在当前 shell 环境中）
suspend	暂停 Shell 的执行，直到收到一个 SIGCONT 信号
test	基于指定条件返回退出状态码 0 或 1
times	显示累计的用户和系统时间
trap	如果收到了指定的系统信号，执行指定的命令
type	显示指定的单词如果作为命令将会如何被解释
typeset	声明一个变量或变量类型。
ulimit	为系统用户设置指定的资源的上限
umask	为新建的文件和目录设置默认权限
unalias	删除指定的别名

命令	说明
unset	删除指定的环境变量或 shell 属性
wait	等待指定的进程完成，并返回退出状态码

alisa 介绍

alisa 用于给命令创建别名。

好处: 可以将经常操作比较复杂的命令进行设置别名, 通过别名的操作提高工作效率

若该命令且不带任何参数, 则显示当前 Shell 进程中的所有别名列表。

```
[root@itheima ~]# alias
alias cp='cp -i'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias mv='mv -i'
alias rm='rm -i'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
[root@itheima ~]#
```

上面是系统为了方便命令操作默认将部分命令创建为别名 `ll` 的命令与 `ls -l` 的效果一样, 就是因为 `ll` 是别名

alias 别名定义语法

```
alias 别名="命令"
```

这里使用单引号或双引号都可以

unalias 别名删除语法

删除指定的别名

```
unalias 别名
```

删除当前Shell环境中所有的别名

```
unalias -a
```

注意: 以上2种方式删除都是临时删除当前Shell的别名, 如果想永久删除必须去配置文件中手动删除

alias 演示

输出所有进程信息

```
ps -aux
```

给上面的命令起别名为psList, 并使用psList输出

```
[root@itheima ~]# alias psList='ps -aux'
[root@itheima ~]# psList
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.6 128164 6820 ?        Ss   08:25   0:01 /usr/lib/systemd/systemd --switched-root --system --deserialize 21
root         2  0.0  0.0      0     0 ?        S    08:25   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    08:25   0:00 [ksoftirqd/0]
root         5  0.0  0.0      0     0 ?        S<   08:25   0:00 [kworker/0:0H]
root         6  0.0  0.0      0     0 ?        S    08:25   0:00 [kworker/u256:0]
root         7  0.0  0.0      0     0 ?        S    08:25   0:00 [migration/0]
root         8  0.0  0.0      0     0 ?        S    08:25   0:00 [rcu_bh]
root         9  0.0  0.0      0     0 ?        R    08:25   0:00 [rcu_sched]
root        10  0.0  0.0      0     0 ?        S    08:25   0:00 [watchdog/0]
root        12  0.0  0.0      0     0 ?        S    08:25   0:00 [kdevtmpfs]
root        13  0.0  0.0      0     0 ?        S<   08:25   0:00 [netns]
root        14  0.0  0.0      0     0 ?        S    08:25   0:00 [khubd]
```

删除psList别名

```
[root@itheima ~]# unalias psList
[root@itheima ~]# psList
-bash: psList: 未找到命令
[root@itheima ~]#
```

小结

1.理解内置命令的含义

就是Shell内部提供的命令, 可以直接使用

内置命令与外部的脚本文件执行区别

内置命令: 执行速度快, 内置命令执行是当前Shell环境的一部分, 就在当前进程内执行, 减少了上下文切换

外部的脚本文件: 执行速度慢, 需要磁盘I/O操作, 并且是fork一个新的进程运行脚本文件, 导致进程之间上下文切换

2.能够使用alias内置命令

定义别名语法: `alias 别名="命令"`

删除别名: `unalias 别名`

Shell内置命令: echo输出字符串

目标

1.能够echo输出不换行的字符串

2.能够使用echo输出转义字符

echo命令介绍

echo 是一个 Shell 内置命令，用于在终端输出字符串，并在最后默认加上换行符

echo输出语法

默认输出换行语法

```
echo 字符串
```

输出不换行语法

```
echo -n 字符串
```

输出演示

1.创建echo1.sh

```
touch echo1.sh
```

2.编辑文件内容并保存

```
vim echo1.sh
```

文件内容

```
#!/bin/bash
echo "hello"
echo "world"
echo -n "itheima "
echo -n "shell "
```

3.运行echo1.sh文件观察效果

```
[root@itheima ~]# sh echo1.sh
hello
world
itheima shell [root@itheima ~]#
```

itheima shell输出为一行

echo输出转义字符

\n 转义字符

用于echo输出字符串非结尾处的换行, 但是默认echo无法解析 \n 转义字符

演示

```
[root@itheima ~]# echo "hello \nworld"
hello \nworld
[root@itheima ~]#
```

-e 参数

-e 参数用于解析转义字符

```
echo -e '字符串中含有转义字符'
```

演示

```
[root@itheima ~]# echo -e "hello \nworld"
hello
world
[root@itheima ~]#
```

/c 转义字符

/c 用于强制清除echo的结尾换行输出

演示

```
[root@itheima ~]# echo "hello world"
hello world
[root@itheima ~]# echo -e "hello world\c"
hello world[root@itheima ~]#
```

← echo默认输出后会换行

← echo使用-e参数与转义字符串\c进行清除输出最后的换行

小结

1.echo输出不换行的字符串

```
echo -n 输出的数据
```

2.echo输出转义字符有哪些

`\n` 换行字符

`\c` 清除echo结尾的换行

`echo -e` 含有转义字符的数据

Shell内置命令：read读取控制台输入

目标

理解read命令的作用

使用read给多个变量赋值

使用read读取1个字符

使用read限制时间输入

介绍

`read` 是 Shell 内置命令，用于从标准输入中读取数据并赋值给变量。如果没有进行重定向，默认就是从终端控制台读取用户输入的数据；如果进行了重定向，那么可以从文件中读取数据。

后续讲解Shell重定向

语法

```
read [-options] [var1 var2 ...]
```

`options` 表示选项，如下表所示；`var` 表示用来存储数据的变量，可以有一个，也可以有多个。

`options` 和 `var` 都是可选的，如果没有提供变量名，那么读取的数据将存放到环境变量 `REPLY` 变量中。

`$REPLY`保存read最后一个读入命令的数据

options支持的参数

选项	说明
-a array	把读取的数据赋值给数组 array，从下标 0 开始。
-d delimiter	用字符串 delimiter 指定读取结束的位置，而不是一个换行符（读取到的数据不包括 delimiter）。
-e	在获取用户输入的时候，对功能键进行编码转换，不会直接显示功能键对应的字符。
== -n num ==	读取 num 个字符，而不是整行字符。
== -p prompt ==	显示提示信息，提示内容为 prompt。
-r	原样读取（Raw mode），不把反斜杠字符解释为转义字符。
== -s ==	静默模式（Silent mode），不会在屏幕上显示输入的字符。 当输入密码和其它确认信息的时候，这是很有必要的。
== -t seconds ==	设置超时时间，单位为秒。如果用户没有在指定时间内输入完成，那么 read 将会返回一个非 0 的退出状态，表示读取失败。
-u fd	使用文件描述符 fd 作为输入源，而不是标准输入，类似于重定向。

示例1：多个变量赋值

需求

使用 read 命令给多个变量赋值

步骤

编辑文件 read1.sh 脚本文件

```
# 使用read命令读取数据,要有提示信息"请输入姓名,年龄,爱好:" 将数据赋值给多个变量
# 打印每一个变量的值
```

演示

创建文件

```
touch read1.sh
```

编辑文件

```
vim read1.sh
```

read1.sh文件内容

```
#!/bin/bash
read -p "请输入姓名,年龄,爱好: " name age hobby
echo "姓名: $name"
echo "年龄: $age"
echo "爱好: $hobby"
```

运行效果

```
[root@itheima ~]# sh read1.sh
请输入姓名,年龄,爱好: itheima 14 Shell
姓名: itheima
年龄: 14
爱好: Shell
[root@itheima ~]#
```

示例2：读取一个字符

需求

从终端控制台只读取一个字符

步骤

编辑文件read2.sh脚本文件

```
# 使用read命令读取数据,要有提示信息"您确定要删除数据吗?(请输入y/n):" 并且设置读取一个字符
# 打印这个字符输出
```

演示

创建文件

```
touch read2.sh
```

编辑文件

```
vim read2.sh
```

read2.sh文件内容

```
#!/bin/bash
read -n 1 -p '请输入一个字符: ' char
printf '\n'
echo '你输入的字符为:$char'
```

注意

`-n 1` 只读取一个字符。运行脚本后，只要输入的一个字符，立即读取结束，不用等待用户按下回车键。

`printf '\n'` 语句用于换行，否则 `echo` 的输出结果会和用户输入的内容位于同一行，不容易区分

运行效果

```
[root@itheima ~]# sh read2.sh
请输入一个字符: A
你输入的字符为:A
[root@itheima ~]#
```

示例3：限制时间输入

需求

在终端控制台输入时, 设置指定时间内输入密码

步骤

编辑文件read3.sh脚本文件

```
# 使用read命令读取数据,要有提示信息"请输入密码(20秒内):" 并且设置限制时间20秒
# 输出一个换行
# 使用read命令读取数据,要有提示信息"请再次输入密码(20秒内):" 并且设置限制时间20秒
# 输出一个换行
# 校验密码与再次输入的密码是否一致
```

演示

创建文件

```
touch read3.sh
```

编辑文件

```
vim read3.sh
```

read3.sh文件内容

```
#!/bin/bash
read -t 20 -sp '请输入密码(20秒内):' pwd1
printf '\n'
read -t 20 -sp '请再次输入密码(20秒内):' pwd2
printf '\n'
if
    [ $pwd1 == $pwd2 ]
then
    echo '密码与确认密码一致, 验证通过'
else
    echo '密码与确认密码不一致, 验证失败'
fi
```

执行文件运行效果

```
[root@itheima ~]# sh read3.sh
请输入密码(20秒内):
请再次输入密码(20秒内):
密码与确认密码一致, 验证通过
[root@itheima ~]# sh read3.sh
请输入密码(20秒内):
请再次输入密码(20秒内):
密码与确认密码不一致, 验证失败
[root@itheima ~]#
```

小结

read命令是什么

用于读取标准输入(终端控制台用户输入的数据)

read给多个变量赋值

```
read var1 var2 ...
```

read读取1个字符

```
read -n 1 var1
```

read限制时间输入

```
read -t 秒数 var1
```

Shell内置命令：exit退出

目标

掌握exit的应用场景

介绍

`exit` 用于退出当前Shell环境进程结束运行, 并且可以返回一个状态码. 一般使用 `$?` 可以获取退出状态码

语法

正确退出语法

```
exit # 默认返回状态码0, 一般代表命令执行成功
```

错误退出语法

```
exit 非0数字 # 数字建议的范围0~255, 一般代表命令执行失败
```

exit应用场景

1. 结束当前Shell进程
2. 当Shell 进程执行出错退出时, 可以返回不同的状态值代表不同的错误

比如执行一个脚本文件里面操作一个文件时, 可以返回1 表示文件不存在, 2 表示文件没有读取权限, 3 表示文件类型不对。

示例：Shell脚本文件中使用exit退出

需求

编写Shell脚本使用exit 退出, 退出时返回一个非0数字状态值, 执行脚本文件并打印返回状态值

步骤

1. 创建exit.sh文件
2. 编辑exit.sh文件, 使用 `exit 数字` 退出结束当前Shell
3. 执行文件, 打印返回状态值

演示

创建文件

```
touch exit.sh
```

编辑文件

```
vim exit.sh
```

exit.sh文件内容: 使用 `exit 数字` 退出结束当前Shell

```
#!/bin/bash
echo 'one'
exit 6
echo 'two'
```

运行效果

```
[root@itheima ~]# sh exit.sh
one
[root@itheima ~]# echo $?
6
[root@itheima ~]#
```

小结

exit的应用场景

- 1.结束当前Shell进程
- 2.可以返回不同的状态码, 用于不同的业务处理

Shell内置命令：declare设置变量

目标

- 1.能够使用declare设置变量的属性
- 2.能够使用declare查看全部Shell变量与函数
- 3.能够使用declare实现索引数组与关联数组变量

介绍

declare命令用于声明 shell 变量。可用来声明变量并设置变量的属性，也可用来显示shell函数。若不加上任何参数，则会显示全部的shell变量与函数(与执行set指令的效果相同)。

declare命令作用

1. declare设置变量的属性[重要]
2. 查看全部Shell变量与函数
3. 实现关联数组变量[重要]

declare设置变量的属性语法

```
declare [+/-][aArxi f][变量名称 = 设置值]
```

+/- "-"可用来指定变量的属性，"+"则是取消变量所设的属性。

a array, 设置为普通索引数组

A Array, 设置为key-value关联数组

r readonly, 将变量设置为只读, 也可以使用readonly

x export, 设置变量成为全局变量, 也可以使用export

i int, 设置为整型变量。

f function, 设置为一个函数变量

示例1：设置变量属性

需求

操作一个变量属性, 设置为整型\ 取消整型\ 设置为只读等操作

步骤

1. 使用declare设置一个整型变量并赋值
2. 取消变量的整型类型, 尝试设置其他类型值尝试
3. 设置变量只读, 尝试赋值测试

演示

```
[root@itheima ~]# declare -i age=20
[root@itheima ~]# echo $age
20
[root@itheima ~]# age=abc
[root@itheima ~]# echo $age
0
[root@itheima ~]# age=14
[root@itheima ~]# echo $age
14
[root@itheima ~]# declare +i age
[root@itheima ~]# age=abc
[root@itheima ~]# echo $age
abc
[root@itheima ~]# declare -r age
[root@itheima ~]# age=123
-bash: age: 只读变量
[root@itheima ~]#
```

定义一个整型变量age, 赋值为20

给整型变量age赋值为abc, 读取数据为0, 说明赋值非整型失败

赋值整型就会成功

设置age变量取消整型类型限制, 并赋值为abc成功

设置变量age为只读

只读变量不可以修改值

查看全部Shell变量与函数定义语法

```
declare [-fF]
```

declare 后无参数, 查询全部Shell变量与函数定义, 与set功能一模一样

-f 查询仅显示函数定义。

-F 查询仅显示函数名字

示例2：查看当前Shell所有函数名

查看所有变量与所有函数

```
declare
```

```
[root@itheima ~]# declare
ABRT_DEBUG_LOG=/dev/null
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extglob:extquote:force_ignore:histappend:interactive_comment
ptvars:sourcepath
BASHRC_VAR1=/etc/bashrc
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_COMPLETION_COMPAT_DIR=/etc/bash_completion.d
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSIONINFO=([0]="4" [1]="2" [2]="46" [3]="2" [4]="release" [5]="x86_64-redhat-linux-gnu")
BASH_VERSION='4.2.46(2)-release'
CLASSPATH=.:usr/local/jdk1.8.0_162/lib
COLUMNS=133
COMP_WORDBREAKS=$' \t\n\"><=;|&(:'
DIRSTACK=()
EUID=0
```

查看所有函数与定义

```
declare -f
```

```
[root@itheima ~]# declare -f
__expand_tilde_by_ref ()
{
    if [[ ${!1} == \~* ]]; then
        if [[ ${!1} == */* ]]; then
            eval $1="${!1/%\~*}/${!1#*/}";
        else
            eval $1="${!1}";
        fi;
    fi
}
__get_cword_at_cursor_by_ref ()
{
    local cword words=();
    __reassemble_comp_words_by_ref "$1" words cword;
    local i cur index=$COMP_POINT lead=${COMP_LINE:0:$COMP_POINT};
    if [[ $index -gt 0 && ( -n $lead && -n ${lead//[[:space:]]} ) ]]; then
        cur=$COMP_LINE;
        for ((i = 0; i <= cword; ++i ))
        do
            while [[ $#cur -ge ${#words[i]} && "${cur:0:${#words[i]}}" != "${words[i]}" ]]; do
                cur="${cur:1}";
                ((index--));
            done;
            if [[ $i -lt $cword ]]; then
                local old_size=${#cur};
                cur="${cur}${words[i]}";
                local new_size=${#cur};
                index=$(( index - old_size + new_size ));
            fi;
        done;
    fi
}
```

查询所有函数名列表

```
declare -F
```

```
[root@itheima ~]# declare -F
declare -f __expand_tilde_by_ref
declare -f __get_cword_at_cursor_by_ref
declare -f __git_aliased_command
declare -f __git_aliases
declare -f __git_commands
declare -f __git_complete
declare -f __git_complete_diff_index_file
declare -f __git_complete_file
declare -f __git_complete_index_file
declare -f __git_complete_remote_or_refspec
declare -f __git_complete_revlist
declare -f __git_complete_revlist_file
declare -f __git_complete_strategy
declare -f __git_compute_all_commands
declare -f __git_compute_merge_strategies
declare -f __git_compute_porcelain_commands
declare -f __git_config_get_set_variables
declare -f __git_count_arguments
declare -f __git_diff_index_files
declare -f __git_diff_index_helper
declare -f __git_find_on_cmdline
declare -f __git_func_wrap
declare -f __git_has_doubledash
```

实现key-value关联数组变量语法

关联数组也称为“键值对 (key-value)”数组，键 (key) 也即字符串形式的数组下标，值 (value) 也即元素值。

```
declare -A 关联数组变量名=([字符串key1]=值1 [字符串key2]=值2 ...)
```

declare也可以用于定义普通索引数组, `-a` 参数创建普通或索引数组 `-A` 创建关联数组

```
declare -a 关联数组变量名=(值1 值2 ...)
```

```
declare -a 关联数组变量名=([0]=值1 [1]=值2 ...)
```

获取指定key的值

```
${关联数组变量名[key]}
```

获取所有值

```
${关联数组变量名[*]} # 方式1  
${关联数组变量名[@]} # 方式2
```

示例3：实现普通索引数组

需求

使用declare定义一普通数组,并获取打印元素数据的值

步骤

- 1.创建declare1.sh文件
- 2.编辑declare1.sh文件内容, 使用declare定义普通数组数据与打印
- 2.执行文件

演示

创建文件

```
touch declare1.sh
```

编辑文件

```
vim declare1.sh
```

declare1.sh文件内容

```
#!/bin/bash
declare -a arr1=(1 2 3 aa)
echo ${arr1[1]}
echo ${arr1[*]}

declare -a arr2=([0]=1 [1]=2 [2]=3 [4]=aa)
echo ${arr2[1]}
echo ${arr2[*]}

declare -a arr2[3]=4
echo ${arr2[*]}
```

运行结果

```
[root@itheima ~]# sh declare1.sh
2
1 2 3 aa
2
1 2 3 aa
1 2 3 4 aa
[root@itheima ~]#
```

示例4：实现key-value关联数组

需求

使用declare定义关联数组并打印输出

演示

```
[root@itheima ~]# declare -A arr4=( [name]="itheima" [age]=20 ["hobby"]="Shell" )
[root@itheima ~]# echo ${arr4[name]}
itheima
[root@itheima ~]# echo ${arr4[age]}
20
[root@itheima ~]# echo ${arr4[*]}
Shell itheima 20
[root@itheima ~]#
```

注意: 使用参数 `-A` 才是关联数组，关联数组无法使用索引获取

```
[root@itheima ~]# array3=(["abc"]=100 ["itcast"]="itheima" )
[root@itheima ~]# echo ${array3["itcast"]}
itheima
[root@itheima ~]# echo ${array3[*]}
itheima
```

不用declare实现关联数组会有问题: 无法获取数组所有数据

小结

1.使用declare设置变量的属性(重点)

```
declare [+/-][aArxi f] 变量名=变量值
```

2.使用declare查看全部Shell变量与函数

```
declare
```

declare -f 查询所有函数的定义

declare -F 查询所欲函数的名称列表

3.使用declare实现普通索引数组和关联数组变量(重点)

索引数组: `declare -a array_name=(item1 item2 ...)`

关联数组: `declare -A array_name=([字符串key1]=item1 [字符串key2]=item2 ...)`

Shell运算符：算术运算符

目标

掌握算术运算符的使用

掌握expr求值表达式命令

expr命令

介绍

expr 是 evaluate expressions 的缩写，译为“求值表达式”。Shell expr 是一个功能强大，并且比较复杂的命令，它除了可以实现整数计算，还可以结合一些选项对字符串进行处理，例如计算字符串长度、字符串比较、字符串匹配、字符串提取等, 后续讲解。

expr语法

计算语法

expr 算术运算符表达式

注意: 运算表达式

获取计算结果赋值给新变量语法

`result=`expr 算术运算符表达式``

演示

```
[root@itheima ~]# expr 1 + 2
3
[root@itheima ~]# result=`expr 1 + 2`
[root@itheima ~]# echo $result
3
[root@itheima ~]#
```

注意: 运算符表达式中每个数字与符号之间要有空格

算术运算符介绍

下表列出了常用的算术运算符，假定变量 a 为 1，变量 b 为 2：

运算符	说明	举例
+	加法	<code>expr \$a + \$b</code> 结果为 3
-	减法	<code>expr \$a - \$b</code> 结果为 -1
*	乘法	<code>expr \$a * \$b</code> 结果为 2
/	除法	<code>expr \$b / \$a</code> 结果为 2
%	取余	<code>expr \$b % \$a</code> 结果为 0
=	赋值	<code>a=\$b</code> 将把变量 b 的值赋给 a

四则运算中如果使用了(), 也需要转义 `\(1 + 1 \)`

算术运算符演示

operation1.sh脚本代码

```
#!/bin/bash
a=1 b=2          # 声明变量a=1和b=2
echo "a=${a} b=${b}"
echo "a + b = `expr $a + $b`"
echo "a * b = `expr $a \* $b`"
echo "a - b = `expr $a - $b`"
echo "a / b = `expr $a / $b`"
echo "b / a = `expr $b / $a`"
echo "b % a = `expr $b % $a`"
```

```
[root@itheima ~]# sh operation1.sh
a=1 b=2
a + b = 3
a * b = 2
a - b = -1
a / b = 2
b / a = 2
b % a = 0
```

小结

常见算术运算符的使用

```
+ - \* / %
```

expr执行运算命令

```
expr 表达式
```

变量名=`expr 表达式`

Shell运算符： 比较运算符

目标

1. 了解有哪些比较运算符
2. 掌握推荐使用哪个比较运算符

整数比较运算符

语法

下表列出了常用的比较运算符，假定变量 a 为 1，变量 b 为 2：

运算符	说明	举例
<code>-eq</code>	equals 检测两个数是否相等，相等返回 0, 否则返回1。	<code>[\$a -eq \$b]</code> 返回 1。
<code>-ne</code>	not equals检测两个数是否不相等，不相等返回 true。	<code>[\$a -ne \$b]</code> 返回 0。
<code>-gt</code>	greater than检测左边的数是否大于右边的，是返回0, 否则1	<code>[\$a -gt \$b]</code> 返回 1。
<code>-lt</code>	lower than检测左边的数是否小于右边的，是返回0, 否则1	<code>[\$a -lt \$b]</code> 返回 0。
<code>-ge</code>	greater equals检测左边的数是否大于等于右边的，是返回0, 否则1	<code>[\$a -ge \$b]</code> 返回 1。
<code>-le</code>	lower equals检测左边的数是否小于等于右边的，是返回0, 否则1	<code>[\$a -le \$b]</code> 返回 0。
<code><</code>	检测左边的数是否小于右边的，是返回0, 否则1	<code>((\$a < \$b))</code> 返回0
<code><=</code>	检测左边的数是否小于等于右边的，是返回0, 否则1	<code>((\$a <= \$b))</code> 返回0
<code>></code>	检测左边的数是否大于右边的，是返回0, 否则1	<code>((\$a > \$b))</code> 返回1
<code>>=</code>	检测左边的数是否大于等于右边的，是返回0, 否则1	<code>((\$a >= \$b))</code> 返回1
<code>==</code>	检测左边的数是否等于右边的，是返回0, 否则1	<code>((\$a == \$b))</code> 返回1
<code>!=</code>	检测左边的数是否不等于右边的，是返回0, 否则1	<code>((\$a != \$b))</code> 返回0

注意:

整数比较运算符只支持整数，不支持小数与字符串(字符串比较后续讲解)，除非字符串的值是整数数字。

每个命令都有返回值, 这个后面我们会讲解退出状态再具体说明, 返回0代表成功, 返回1代表失败

演示

operation2.sh脚本代码

```
#!/bin/bash
a=1 b=2
echo "a=${a} b=${b}"
if [ $a -eq $b ]
then
    echo "$a -eq $b : a 等于 b"
```

```
else
    echo "$a -eq $b: a 不等于 b" # 输出这个
fi
if [ $a -ne $b ]
then
    echo "$a -ne $b: a 不等于 b" # 输出这个
else
    echo "$a -ne $b : a 等于 b"
fi
if [ $a -gt $b ]
then
    echo "$a -gt $b: a 大于 b"
else
    echo "$a -gt $b: a 不大于 b" # 输出这个
fi
if [ $a -lt $b ]
then
    echo "$a -lt $b: a 小于 b" # 输出这个
else
    echo "$a -lt $b: a 不小于 b"
fi
if [ $a -ge $b ]
then
    echo "$a -ge $b: a 大于或等于 b"
else
    echo "$a -ge $b: a 小于 b" # 输出这个
fi
if [ $a -le $b ]
then
    echo "$a -le $b: a 小于或等于 b" # 输出这个
else
    echo "$a -le $b: a 大于 b"
fi

if (($a > $b))
then
    echo "$a > $b: a 大于 b"
else
    echo "$a > $b: a 不大于 b"
fi
if (($a < $b))
then
    echo "$a < $b: a 小于 b"
else
    echo "$a < $b: a 不小于 b"
fi
if (($a >= $b))
then
    echo "$a >= $b: a 大于或等于 b"
else
    echo "$a >= $b: a 小于 b"
fi
if (($a <= $b))
```

```
then
    echo "$a <= $b: a 小于或等于 b"
else
    echo "$a <= $b: a 大于 b"
fi
```

运行效果

```
[root@itheima ~]# sh operation2.sh
a=1 b=2
1 -eq 2: a 不等于 b
1 -ne 2: a 不等于 b
1 -gt 2: a 不大于 b
1 -lt 2: a 小于 b
1 -ge 2: a 小于 b
1 -le 2: a 小于或等于 b
1 > 2: a 不大于 b
1 < 2: a 小于 b
1 >= 2: a 小于 b
1 <= 2: a 小于或等于 b
[root@itheima ~]#
```

字符串比较运算符

介绍

可以比较2个变量, 变量的类型可以为数字（整数, 小数）与字符串

语法

下表列出了常用的字符串运算符, 假定变量 a 为 "abc", 变量 b 为 "efg":

字符串比较可以使用 `[]` 和 `[]` 2种方式

运算符	说明	举例
<code>==</code> 或 <code>=</code>	相等。用于比较两个字符串或数字，相同则返回 0。可以使用 <code>=</code>	<code>[\$a == \$b]</code> 返回1 <code>[\$a = \$b]</code> 返回 1 <code>[[\$a == \$b]]</code> 返回1 <code>[[\$a = \$b]]</code> 返回1
<code>!=</code>	不相等。用于比较两个字符串或数字，不相同则返回 0。	<code>[\$a != \$b]</code> 返回 0 <code>[[\$a != \$b]]</code> 返回 0
<code><</code>	小于, 用于比较两个字符串或数字， 小于返回0， 否则返回1	<code>[\$a \< \$b]</code> 返回 0 <code>[[\$a < \$b]]</code> 返回 0
<code>></code>	大于, 用于比较两个字符串或数字， 大于返回0， 否则返回1	<code>[\$a \> \$b]</code> 返回 1 <code>[[\$a > \$b]]</code> 返回 1
<code>-z</code>	检测字符串长度是否为0， 如果长度为0返回则返回0, 否则返回1。	<code>[-z \$a]</code> 返回 false。
<code>-n</code>	检测字符串长度是否不为 0， 如果长度不为 0 则返回0, 否则返回 1。	<code>[-n "\$a"]</code> 返回 true。
<code>\$</code>	检测字符串是否不为空， 不为空返回0, 为空返回1。	<code>[\$a]</code> 返回 true。

字符串比较没有 `<=` 可以通过 `[["a" < "b" && "a" == "b"]]`

演示

operation6.sh脚本代码

```
#!/bin/bash

a="itheima" b="itcast" c=1 d=2
echo "a=${a},b=${b},c=${c},d=${d}"

if [ $a = $b ]
then
    echo "$a = $b : a 等于 b"
else
    echo "$a = $b: a 不等于 b"
fi

if [ $a != $b ]
then
    echo "$a != $b : a 不等于 b"
else
    echo "$a != $b: a 等于 b"
fi

if [[ $a > $b ]]
then
    echo "$a > $b : a 大于 b"
```

```

else
    echo "$a > $b: a 不大于 b"
fi

if [ $a \> $b ]
then
    echo "$a > $b : a 大于 b"
else
    echo "$a > $b: a 不大于 b"
fi

if [[ $c > $d ]]
then
    echo "$c > $d : c 大于 d"
else
    echo "$c > $d: c 不大于 d"
fi

if [ -z $a ]
then
    echo "-z $a : 字符串长度为 0"
else
    echo "-z $a : 字符串长度不为 0"
fi

if [ -n "$a" ]
then
    echo "-n $a : 字符串长度不为 0"
else
    echo "-n $a : 字符串长度为 0"
fi

if [ $a ]
then
    echo "$a : 字符串不为空"
else
    echo "$a : 字符串为空"
fi

```

运行效果

```

[root@itheima ~]# sh operation6.sh
a=itheima,b=itcast,c=1,d=2
itheima = itcast: a 不等于 b
itheima != itcast : a 不等于 b
itheima > itcast : a 大于 b
itheima > itcast : a 大于 b
1 > 2: c 不大于 d
-z itheima : 字符串长度不为 0
-n itheima : 字符串长度不为 0
itheima : 字符串不为空

```

[[]] 和 [] 的区别

区别1: word splitting的发生

区别说明

`[[]]` 不会有word splitting发生

`[]` 会有word splitting发生

word splitting介绍

会将含有空格字符串进行分拆分割后比较

示例

```
[root@itheima ~]# a="a"
[root@itheima ~]# b="a b c"
[root@itheima ~]# [[ $a == $b ]]
[root@itheima ~]# echo $?
1
[root@itheima ~]# [ $a == $b ]
-bash: [: 参数太多
```

b变量时含有空格字符串的

使用[[]]比较正常, 返回1, 代表不相等

使用[]比较失败, 这是由于b变量进行以空格分隔分拆后进行比较导致的失败

通过 `$?` 获取上一个命令的退出状态, 0代表成功, 1代表失败

区别2: 转义字符

区别说明

`[[]]` 对 `<` 不需要转义, 格式为 `[[字符串1 < 字符串2]]`

`[]` 需要对 `<`, `>` 转义, 格式为 `[字符串1 \< 字符串2]`

示例

`[]` 执行效果

```
[root@itheima ~]# [ "a" > "b" ]
[root@itheima ~]# echo $?
0
[root@itheima ~]# [ "a" < "b" ]
[root@itheima ~]# echo $?
0
[root@itheima ~]# [ "a" \> "b" ]
[root@itheima ~]# echo $?
1
[root@itheima ~]# [ "a" \< "b" ]
[root@itheima ~]# echo $?
0
[root@itheima ~]#
```

[]不转义比较字符串, 怎么比较都返回0, 说明无效

[]中对>转义后比较字符串, 返回1结果正确

[]中对<转义后比较字符串, 返回0结果正确

`[[]]` 执行效果, 不需要转义执行结果正确

```
[root@itheima ~]# [[ "a" > "b" ]]
[root@itheima ~]# echo $?
1
[root@itheima ~]# [[ "a" < "b" ]]
[root@itheima ~]# echo $?
0
[root@itheima ~]#
```

小结

1. 了解有哪些比较运算符

整数比较运算符

比较方式: `[]` 和 `((()))`

方式1: `[整数1 options 整数2]`

options: `-eq -gt -ge -lt -le -ne`

方式2: `((整数1 options 整数2))`

options: `< <= > >= == !=` 注意判断等于不可以使用 `=`, 等于在 `(())` 里面是赋值的含义

字符串,数字比较运算符

介绍: 可以比较2个变量, 变量的类型可以为字符串, 数字(整数,小数)

比较方式: `[]` 和 `[[[]]]`

方式1: `[变量1 options 整数2]`

options: `<, >, ==, =, !=`

`<, >` 需要转义

会发生word splitting

方式2: `[[变量1 options 变量2]]`

options: `<, >, ==, =, !=`

`<, >` 不需要转义

不会发生word splitting

2. 推荐使用哪个比较运算符?

`[[表达式]]` 即可以实现数字和字符串比较, 并且不需要转义, 不会word splitting

Shell运算符： 布尔运算符

目标

掌握布尔运算符的使用

介绍

运算符	说明	举例
!	非运算，取反, 表达式为 true 则返回 false, 否则返回 true。	[! 表达式] 取反。
-o	or 或运算，有一个表达式为 true 则返回 true。	[表达式1 -o 表达式2]
-a	and 与运算，两个表达式都为 true 才返回 true。	[表达式1 -a 表达式2]

注意布尔运算符放在 [] 或与test命令配合使用才有效

布尔运算符常与与test命令配合使用, 后续讲解

演示

operation4.sh脚本代码脚本代码

```
#!/bin/bash
a=1 b=2

if [ $a -lt 2 -a $b -gt 10 ]
then
    echo "$a 小于 2 且 $b 大于 10 : 返回 true"
else
    echo "$a 小于 2 且 $b 大于 10 : 返回 false" # $b -gt 10不成立，输出这个表达式
fi

if [ $a -lt 10 -o $b -gt 10 ]
then
    echo "$a 小于 10 或 $b 大于 10 : 返回 true" # $a -lt 10 成立，输出这个表达式
else
    echo "$a 小于 10 或 $b 大于 10 : 返回 false"
fi

if [ ! $a -gt $b ]
then
    echo "$a 大于 $b 取反 : 返回 true"
else
    echo "$a 大于 $b 取反 : 返回 false" # $a -gt $b 为true ，取反为false，输出这个表达式
fi
```

运行效果

```
[root@itheima ~]# sh operation4.sh
1 小于 2 且 2 大于 10 : 返回 false
1 小于 10 或 2 大于 10 : 返回 true
1 大于 2 取反 : 返回 true
[root@itheima ~]#
```


小结

布尔运算符的使用

- `!` 取反
 - `-o` or, 或者, 只要有一个表达式成立返回0, 那么就返回0
 - `-a` and, 并且, 需要多个表达式必须全部成立才返回0, 否则返回1
- 布尔运算符必须与`[]`使用 或者 与`test`命令一起使用, 后续讲解`test`命令

Shell运算符： 逻辑运算符

目标

- 1. 能够使用逻辑运算符
- 2. 能够理解逻辑运算符与布尔运算符区别

介绍

运算符	说明	举例
<code>&&</code>	逻辑的 AND	<code>[[表达式1 && 表达式2]]</code>
<code> </code>	逻辑的 OR	<code>[[表达式1 表达式2]]</code>
<code>!</code>	逻辑非	<code>[[! 表达式]]</code>

注意

使用 `&&` 和 `||` 的运算符必须放在 `[]` 或 `(())` 中才有效, 否则报错

`!` 可以用在 `[]`, `[]` 中, 不可以在`(())`

演示

operation5.sh脚本代码

```
#!/bin/bash

a=1 b=2

if [[ $a -lt 10 && $b -gt 10 ]]
then
    echo "返回 true"
else
    echo "返回 false" # $b -gt 10 不成立, 输出false
```

```
fi

if [[ $a -lt 10 || $b -gt 10 ]]
then
    echo "返回 true"    # $a -lt 10 成立， 输出true
else
    echo "返回 false"
fi
```

运行效果

```
[root@itheima ~]# sh operation5.sh
返回 false
返回 true
[root@itheima ~]#
```

小结

1. 使用逻辑运算符

- ! 逻辑非
- && 逻辑与
- || 逻辑或

2. 理解逻辑运算符与布尔运算符区别

布尔运算符: ! -o -a

必须有[] 或test命令执行

逻辑运算符: ! || &&

|| && 必须有[][] 或 (()) 执行

! 可以用在 [] 或 [[]] , 不可以用在(())

Shell运算符： 文件测试运算符

目标

能够使用常用文件测试运算符检查文件

例如: 文件是否存在\是否可读\是否可执行\是否为空\是否可写\是否是目录\是否是普通文件

linux系统文件类型介绍

-: 普通文件

d: 目录文件

l: 链接文件

b: 块设备文件

c: 字符设备文件

块设备文件: 比如计算机硬盘/dev/sda

```
[root@itheima dev]# ll /dev | grep sda
brw-rw----. 1 root disk      8,  0 7月  8 08:49 sda
brw-rw----. 1 root disk      8,  1 7月  8 08:49 sda1
brw-rw----. 1 root disk      8,  2 7月  8 08:49 sda2
```

字符设备文件: 比如计算机的USB文件/dev/usb

```
[root@itheima dev]# ll /dev | grep usb
crw-----. 1 root root    249,  0 7月  8 08:49 usbmon0
crw-----. 1 root root    249,  1 7月  8 08:49 usbmon1
crw-----. 1 root root    249,  2 7月  8 08:49 usbmon2
```

设备文件都是对应计算机硬件的, 不同的设备文件代表不同的传输数据方式

p: 管道文件

介绍

文件测试运算符用于检测文件的各种属性。

属性检测描述如下:

操作符	说明	举例
-b file	检测文件是否是块设备文件，如果是，则返回 true。	[-b \$file] 返回 false。
-c file	检测文件是否是字符设备文件，如果是，则返回 true。	[-c \$file] 返回 false。
==-d file==	directory, 检测文件是否是目录，如果是，则返回 true。	[-d \$file] 返回 false。
==-f file==	file, 检测文件是否是普通文件（既不是目录，也不是设备文件），如果是，则返回 true。	[-f \$file] 返回 true。
-g file	检测文件是否设置了 SGID 位，如果是，则返回 true。	[-g \$file] 返回 false。
-k file	检测文件是否设置了粘着位(Sticky Bit)，如果是，则返回 true。	[-k \$file] 返回 false。
-p file	检测文件是否是有名管道文件，如果是，则返回 true。	[-p \$file] 返回 false。
-u file	检测文件是否设置了 SUID 位，如果是，则返回 true。	[-u \$file] 返回 false。
==-r file==	read,检测文件是否可读，如果是，则返回 true。	[-r \$file] 返回 true。
==-w file==	write,检测文件是否可写，如果是，则返回 true。	[-w \$file] 返回 true。
==-x file==	execute, 检测文件是否可执行，如果是，则返回 true。	[-x \$file] 返回 true。
==-s file==	size, 检测文件是否为空（文件大小是否大于0），不为空返回 true。	[-s \$file] 返回 true。
==-e file==	exists, 检测文件（包括目录）是否存在，如果是，则返回 true。	[-e \$file] 返回 true。
file1 -nt file2	new than(nt), file1是否比file2新	[file1 -nt file2]
file1 -ot file2	old than(ot), file1是否比file2旧	[file1 -ot file2]

其他检查符：

- -S: 判断某文件是否 socket。
- -L: link, 检测文件是否存在并且是一个符号链接。

语法

```
[ options 文件路径字符串 ]  
或  
[[ options 文件路径字符串 ]]
```

演示

operation6.sh脚本代码

```
#!/bin/bash
```

```
file="/root/operation1.sh"
if [ -w $file ]
then
    echo "文件可写"
else
    echo "文件不可写"
fi
if [ -r $file ]
then
    echo "文件可读"
else
    echo "文件不可读"
fi
if [ -x $file ]
then
    echo "文件可执行"
else
    echo "文件不可执行"
fi
if [ -f $file ]
then
    echo "文件是普通文件"
else
    echo "文件是特殊文件"
fi
if [ -s $file ]
then
    echo "文件不是空"
else
    echo "文件是空"
fi
if [ -e $file ]
then
    echo "文件存在"
else
    echo "文件不存在"
fi
if [ -d $file ]
then
    echo "文件是目录"
else
    echo "文件不是目录"
fi

file2="/root/operation2.sh"
if [ file -nt file2 ]
then
    echo "operation1.sh文件比operation2.sh文件新"
else
    echo "operation1.sh文件不比operation2.sh文件新"
fi
```

运行效果

```
[root@itheima ~]# sh operation6.sh
文件可写
文件可读
文件不可执行
文件是普通文件
文件不是空
文件存在
文件不是目录
operation1.sh文件不比operation2.sh文件新
[root@itheima ~]#
```

operation1.sh文件不可执行, 因为没有可执行权限

查看operation1.sh文件权限

```
[root@itheima ~]# ll | grep operation1.sh
-rw-r--r--. 1 root root 549 7月 1 15:16 operation1.sh
[root@itheima ~]#
```

给operation1.sh添加执行权限

```
[root@itheima ~]# chmod a+x operation1.sh
[root@itheima ~]# ll | grep operation1.sh
-rwxr-xr-x. 1 root root 549 7月 1 15:16 operation1.sh
[root@itheima ~]#
```

重新执行operation6.sh

```
[root@itheima ~]# sh operation6.sh
文件可写
文件可读
文件可执行 ←
文件是普通文件
文件不是空
文件存在
文件不是目录
operation1.sh文件不比operation2.sh文件新
[root@itheima ~]#
```

小结

常用文件测试运算符

文件是否存在: -e

是否可读: -r

是否可执行: -x

是否为空: -s

是否可写: -w

是否是目录: -d

是否是普通文件: -f