

Linux 操作系统



Shell 脚本编程



主要内容和学习要求

- ❑ 掌握创建 `shell` 脚本的基本步骤
- ❑ 学会使用条件测试
- ❑ 掌握 `if` 条件结构与 `case` 选择结构
- ❑ 掌握 `for` 循环、`while` 循环和 `until` 循环结构
- ❑ 学会 `shift` 命令的使用
- ❑ 学会 `shell` 脚本的调试



Shell 脚本

❑ Shell 脚本

当命令不在命令行中执行，而是从一个文件中执行时，该文件就称为 **shell 脚本**。**shell 脚本**按行解释。

❑ Shell 脚本的编写

- **shell 脚本**是纯文本文件，可以使用任何文本编辑器编写
- **shell 脚本**通常是以 `.sh` 作为后缀名

❑ Shell 脚本的执行

```
chmod +x script_name  
./script_name
```

```
sh script_name
```



Shell 脚本

□ Shell 脚本的格式

- ◆ 第一行：指定用哪个程序来编译和执行脚本。

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

```
#!/bin/sh
```

```
#!/bin/csh
```

- ◆ 可执行语句和 **shell** 控制结构
一个 **shell** 脚本通常由一组 **Linux** 命令、**shell** 命令、控制结构和注释语句构成。
- ◆ 注释：以“#”开头，可独占一行，或跟在语句的后面。

在脚本中多写注释语句是一个很好的编程习惯



Shell 脚本举例

```
#!/bin/bash
# This is the first Bash shell program
# Scriptname: greetings.sh
echo
echo -e "Hello $LOGNAME, \c"
echo "it's nice talking to you."
echo -e "Your present working directory is:"
pwd # Show the name of present directory
echo
echo "The time is `date +%T`!. \nBye"
echo
```

```
sh greetings
```

```
chmod +x greetings
greetings
```



Shell 脚本举例

```
#!/bin/bash
# This script is to test the usage of read
# Scriptname: ex4read.sh
echo "=== examples for testing read ==="
echo -e "What is your name? \c"
read name
echo "Hello $name"
echo
echo -n "Where do you work? "
read
echo "I guess $REPLY keeps you busy!"
echo
read -p "Enter your job title: "
echo "I thought you might be an $REPLY."
echo
echo "=== End of the script ==="
```



条件测试

- ◆ 条件测试可以根据某个特定条件是否满足，来选择执行相应的任务。
- ◆ **Bash** 中允许测试两种类型的条件：
命令成功或失败，表达式为真或假
- ◆ 任何一种测试中，都要有退出状态（返回值），退出状态为 0 表示命令成功或表达式为真，非0 则表示命令失败或表达式为假。
- ◆ 状态变量 `$?` 中保存命令退出状态的值

```
grep $USER /etc/passwd  
echo $?  
grep hello /etc/passwd; echo $?
```



测试表达式的值

- ❑ 表达式测试包括字符串测试、整数测试和文件测试。
- ❑ 内置测试命令 `test`
 - 通常用 `test` 命令来测试表达式的值

```
x=5; y=10  
test $x -gt $y  
echo $?
```

- `test` 命令可以用 **方括号** 来代替

```
x=5; y=10  
[ $x -gt $y ]  
echo $?
```

方括号前后要留空格!



测试表达式的值

- 2.x 版本以上的 **Bash** 中可以用**双方括号**来测试表达式的值，此时可以使用**通配符**进行**模式匹配**。

```
name=Tom  
[ $name = [Tt]?? ]  
echo $?
```

```
[[ $name = [Tt]?? ]]  
echo $?
```



字符串测试

❑ 字符串测试

操作符两边必须留空格!

<code>[-z str]</code>	如果字符串 <code>str</code> 长度为 0，返回真
<code>[-n str]</code>	如果字符串 <code>str</code> 长度不为 0，返回真
<code>[str1 = str2]</code>	两字符串相等
<code>[str1 != str2]</code>	两字符串不等

```
name=Tom; [ -z $name ]; echo $?
```

```
name2=Andy; [ $name = $name2 ] ; echo $?
```



整数测试

□ 整数测试，即比较大小

操作符两边必须留空格!

<code>[int1 -eq int2]</code>	<code>int1</code> 等于 <code>int2</code>
<code>[int1 -ne int2]</code>	<code>int1</code> 不等于 <code>int2</code>
<code>[int1 -gt int2]</code>	<code>int1</code> 大于 <code>int2</code>
<code>[int1 -ge int2]</code>	<code>int1</code> 大于或等于 <code>int2</code>
<code>[int1 -lt int2]</code>	<code>int1</code> 小于 <code>int2</code>
<code>[int1 -le int2]</code>	<code>int1</code> 小于或等于 <code>int2</code>

```
x=1; [ $x -eq 1 ]; echo $?
```

```
x=a; [ $x -eq 1 ]; echo $? X
```



整数测试

❑ 整数测试也可以使用 `let` 命令或双圆括号

- 相应的操作符为：

只能用于整数测试!

`==` 、 `!=` 、 `>` 、 `>=` 、 `<` 、 `<=`

- 例：

```
x=1; let "$x == 1"; echo $?
```

```
x=1; (( $x+1 >= 2 )); echo $?
```

❑ 两种测试方法的区别

- 使用的操作符不同
- `let` 和 双圆括号中可以使用算术表达式，而中括号不能
- `let` 和 双圆括号中，操作符两边可以不留空格



逻辑测试

□ 逻辑测试

<code>[expr1 -a expr2]</code>	逻辑与，都为真时，结果为真
<code>[expr1 -o expr2]</code>	逻辑或，有一个为真时，结果为真
<code>[! expr]</code>	逻辑非

```
x=1; name=Tom;
```

```
[ $x -eq 1 -a -n $name ]; echo $?
```

注：不能随便添加括号

```
[ ( $x -eq 1 ) -a ( -n $name ) ]; echo $?
```

X



逻辑测试

❑ 可以使用模式的逻辑测试

<code>[[pattern1 && pattern2]]</code>	逻辑与
<code>[[pattern1 pattern2]]</code>	逻辑或
<code>[[! pattern]]</code>	逻辑非

```
x=1; name=Tom;
```

```
[[ $x -eq 1 && $name = To? ]]; echo $?
```



文件测试

❑ 文件测试：文件是否存在，文件属性，访问权限等。

常见的文件测试操作符

-f <code>fname</code>	<code>fname</code> 存在且是普通文件时，返回真（即返回 0）
-L <code>fname</code>	<code>fname</code> 存在且是链接文件时，返回真
-d <code>fname</code>	<code>fname</code> 存在且是一个目录时，返回真
-e <code>fname</code>	<code>fname</code> （文件或目录）存在时，返回真
-s <code>fname</code>	<code>fname</code> 存在且大小大于 0 时，返回真
-r <code>fname</code>	<code>fname</code> （文件或目录）存在且可读时，返回真
-w <code>fname</code>	<code>fname</code> （文件或目录）存在且可写时，返回真
-x <code>fname</code>	<code>fname</code> （文件或目录）存在且可执行时，返回真

● 更多文件测试符参见 `test` 的在线帮助

`man test`



检查空值

□ 检查空值

```
[ "$name" = "" ]
```

```
[ ! "$name" ]
```

```
[ "X${name}" != "X" ]
```




if 条件语句

□ 语法结构

```
if expr1          # 如果expr1 为真(返回值为0)
then             # 那么
    commands1    # 执行语句块 commands1
elif expr2       # 若expr1 不真, 而expr2 为真
then            # 那么
    commands2    # 执行语句块 commands2
    ... ..      # 可以有多个 elif 语句
else            # else 最多只能有一个
    commands4    # 执行语句块 commands4
fi              # if 语句必须以单词 fi 终止
```



几点说明

- ◆ **elif** 可以有任意多个（0 个或多个）
- ◆ **else** 最多只能有一个（0 个或 1 个）
- ◆ **if** 语句必须以 **fi** 表示结束
- ◆ **expr** 通常为条件测试表达式；也可以是多个命令，以最后一个命令的退出状态为条件值。
- ◆ **commands** 为可执行语句块，如果为空，需使用 **shell** 提供的空命令“**:**”，即冒号。该命令不做任何事情，只返回一个退出状态 **0**
- ◆ **if** 语句可以嵌套使用

```
ex4if.sh, chkperm.sh, chkperm2.sh,  
name_grep, tellme, tellme2, idcheck.sh
```



case 选择语句

□ 语法结构

```
case expr in # expr 为表达式，关键词 in 不要忘！
  pattern1) # 若 expr 与 pattern1 匹配，注意括号
    commands1 # 执行语句块 commands1
  ;; # 跳出 case 结构
  pattern2) # 若 expr 与 pattern2 匹配
    commands2 # 执行语句块 commands2
  ;; # 跳出 case 结构
  ... .. # 可以有任意多个模式匹配
  *) # 若 expr 与上面的模式都不匹配
    commands # 执行语句块 commands
  ;; # 跳出 case 结构
esac # case 语句必须以 esac 终止
```



几点说明

- ◆ 表达式 `expr` 按顺序匹配每个模式，一旦有一个模式匹配成功，则执行该模式后面的所有命令，然后退出 `case`。
- ◆ 如果 `expr` 没有找到匹配的模式，则执行缺省值“`*`)”后面的命令块（类似于 `if` 中的 `else`）；“`*`)”可以不出现。
- ◆ 所给的匹配模式 `pattern` 中可以含有通配符和“`|`”。
- ◆ 每个命令块的最后必须有一个双分号，可以独占一行，或放在最后一个命令的后面。
- ◆ `case` 语句举例：`yes_no.sh`



for 循环语句

□ 语法结构

```
for variable in list
# 每一次循环，依次把列表 list 中的一个值赋给循环变量
do          # 循环开始的标志
    commands # 循环变量每取一次值，循环体就执行一遍
done       # 循环结束的标志
```

□ 几点说明

- 列表 `list` 可以是命令替换、变量名替换、字符串和文件名列表（可包含通配符）
- `for` 循环执行的次数取决于列表 `list` 中单词的个数
- `for` 循环体中一般要出现循环变量，但也可以不出现



for 循环执行过程

□ 循环执行过程

执行第一轮循环时，将 `list` 中的第一个词赋给循环变量，并把该词从 `list` 中删除，然后进入循环体，执行 `do` 和 `done` 之间的命令。下一次进入循环体时，则将第二个词赋给循环变量，并把该词从 `list` 中删除，再往后的循环也以此类推。当 `list` 中的词全部被移走后，循环就结束了。

```
forloop.sh, mybackup.sh
```

□ 位置参量的使用: `$*` 与 `$@` `greet.sh`

□ 可以省略 `in list`，此时使用位置参量

```
permx.sh tellme greet.sh / permx.sh *
```



while 循环语句

□ 语法结构

```
while expr # 执行 expr
do # 若 expr 的退出状态为 0，进入循环，否则退出 while
    commands # 循环体
done # 循环结束标志，返回循环顶部
```

□ 执行过程

先执行 `expr`，如果其退出状态为 `0`，就执行循环体。执行到关键字 `done` 后，回到循环的顶部，`while` 命令再次检查 `expr` 的退出状态。以此类推，循环将一直继续下去，直到 `expr` 的退出状态非 `0` 为止。



until 循环语句

□ 语法结构

```
until expr # 执行 expr
do # 若expr的退出状态非0，进入循环，否则退出until
    commands # 循环体
done # 循环结束标志，返回循环顶部
```

□ 执行过程

与 `while` 循环类似，只是当 `expr` 退出状态非 0 时才执行循环体，直到 `expr` 为 0 时退出循环。



break 和 continue

break [n]

- 用于强行退出当前循环。
- 如果是嵌套循环，则 `break` 命令后面可以跟一数字 `n`，表示退出第 `n` 重循环（最里面的为第一重循环）。

continue [n]

- 用于忽略本次循环的剩余部分，回到循环的顶部，继续下一次循环。
- 如果是嵌套循环，`continue` 命令后面也可跟一数字 `n`，表示回到第 `n` 重循环的顶部。

例: `months.sh`



exit 和 sleep

□ exit 命令

```
exit n
```

`exit` 命令用于退出脚本或当前进程。`n` 是一个从 0 到 255 的整数，0 表示成功退出，非零表示遇到某种失败而非正常退出。该整数被保存在状态变量 `$?` 中。

□ sleep 命令

```
sleep n
```

暂停 `n` 秒钟



select 循环与菜单

□ 语法结构

```
select variable in list
do          # 循环开始的标志
    commands # 循环变量每取一次值，循环体就执行一遍
done       # 循环结束的标志
```

□ 说明

- **select** 循环主要用于创建菜单，按数字顺序排列的菜单项将显示在标准错误上，并显示 **PS3** 提示符，等待用户输入
- 用户输入菜单列表中的某个数字，执行相应的命令
- 用户输入被保存在内置变量 **REPLY** 中。

例: `runit.sh`



select 与 case

❑ `select` 是个无限循环，因此要记住用 `break` 命令退出循环，或用 `exit` 命令终止脚本。也可以按 `ctrl+c` 退出循环。

❑ `select` 经常和 `case` 联合使用

例: `goodboy.sh`

❑ 与 `for` 循环类似，可以省略 `in list`，此时使用位置参量



循环控制 `shift` 命令

```
shift [n]
```

- 用于将参量列表 `list` 左移指定次数，缺省为左移一次。
- 参量列表 `list` 一旦被移动，最左端的那个参数就从列表中删除。`while` 循环遍历位置参量列表时，常用到 `shift`。

例:

```
doit.sh a b c d e f g h
```

```
shft.sh a b c d e f g h
```



随机数和 `expr` 命令

❑ 生成随机数的特殊变量

```
echo $RANDOM
```

❑ `expr`: 通用的表达式计算命令

表达式中参数与操作符必须以空格分开，表达式中的运算可以是算术运算，比较运算，字符串运算和逻辑运算。

```
expr 5 % 3
```

```
expr 5 \* 3 # 乘法符号必须被转义
```



字符串操作

□ 字符串操作

m 的取值从 0 到 $\${\#var}-1$

$\${\#var}$	返回字符串变量 var 的长度
$\${var:m}$	返回 $\${var}$ 中从第m个字符到最后的 部分
$\${var:m:len}$	返回 $\${var}$ 中从第m个字符开始，长度为len的 部分
$\${var\#pattern}$	删除 $\${var}$ 中开头部分与pattern匹配的 最小部分
$\${var##pattern}$	删除 $\${var}$ 中开头部分与pattern匹配的 最大部分
$\${var\%pattern}$	删除 $\${var}$ 中结尾部分与pattern匹配的 最小部分
$\${var%%pattern}$	删除 $\${var}$ 中结尾部分与pattern匹配的 最大部分
$\${var/old/new}$	用new替换 $\${var}$ 中第一次出现的old
$\${var//old/new}$	用new替换 $\${var}$ 中所有的old(全局替换)

注: pattern, old 中可以使用**通配符**。

例: `ex4str`



脚本调试

```
sh -x 脚本名
```

该选项可以使用户跟踪脚本的执行，此时 `shell` 对脚本中每条命令的处理过程为：先执行替换，然后显示，再执行它。`shell` 显示脚本中的行时，会在行首添加一个加号“+”。

```
sh -v 脚本名
```

在执行脚本之前，按输入的原样打印脚本中的各行。

```
sh -n 脚本名
```

对脚本进行语法检查，但不执行脚本。如果存在语法错误，`shell` 会报错，如果没有错误，则不显示任何内容。



编程小结：变量

❑ 局部变量、环境变量 (`export`、`declare -x`)

❑ 只读变量、整型变量

```
例: declare -i x; x="hello"; echo $x
```

0

❑ 位置参量 (`$0`、`$1`、`...`、`$*`、`$@`、`$#`、`$$`、`$?`)

❑ 变量的间接引用 (`eval`、`${!str}`)

```
例: name="hello"; x="name"; echo ${!x}
```

hello

❑ 命令替换 (``cmd``、`$(cmd)`)

❑ 整数运算

`declare` 定义的整型变量可以直接进行运算，
否则需用 `let` 命令或 `#[...]`、`$((...))` 进行整数运算。



编程小结：输入输出

□ 输入：read

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

```
read
```

→ REPLY

```
read -p "提示"
```

→ REPLY

□ 输出：printf

输出参数用空格隔开

```
printf "%-12.5f \t %d \n" 123.45 8
```

format
以%开头

flag

-:左对齐
+:输出符号
0:空白处添0
空格:前面加一空格

field width

字段宽度

precision

小数点后
输出位数

格式符

\b
\n
\r
\t
\v
\\
/"
%%
c
d
e
f
g
s
o
x



编程小结：条件测试

□ 字符串测试

操作符两边必须留空格！

<code>[-z string]</code>	如果字符串string长度为0，返回真
<code>[-n string]</code>	如果字符串string长度不为0，返回真
<code>[str1 = str2]</code>	两字符串相等（也可以使用 <code>==</code> ）
<code>[str1 != str2]</code>	两字符串不等

如果使用双方括号，可以使用 **通配符** 进行模式匹配。

<code>[[str1 = str2]]</code>	两字符串相等（也可以使用 <code>==</code> ）
<code>[[str1 != str2]]</code>	两字符串不等
<code>[[str1 > str2]]</code>	str1大于str2,按ASCII码比较
<code>[[str1 < str2]]</code>	str1小于str2,按ASCII码比较

例：`name=Tom; [[$name > Tom]]; echo $?`



编程小结：条件测试

❑ 整数测试

注意这两种方法的区别！

<code>[int1 -eq int2]</code>	<code>int1</code> 等于 <code>int2</code>
<code>[int1 -ne int2]</code>	<code>int1</code> 不等于 <code>int2</code>
<code>[int1 -gt int2]</code>	<code>int1</code> 大于 <code>int2</code>
<code>[int1 -ge int2]</code>	<code>int1</code> 大于或等于 <code>int2</code>
<code>[int1 -lt int2]</code>	<code>int1</code> 小于 <code>int2</code>
<code>[int1 -le int2]</code>	<code>int1</code> 小于或等于 <code>int2</code>

<code>((int1 == int2))</code>	<code>int1</code> 等于 <code>int2</code>
<code>((int1 != int2))</code>	<code>int1</code> 不等于 <code>int2</code>
<code>((int1 > int2))</code>	<code>int1</code> 大于 <code>int2</code>
<code>((int1 >= int2))</code>	<code>int1</code> 大于或等于 <code>int2</code>
<code>((int1 < int2))</code>	<code>int1</code> 小于 <code>int2</code>
<code>((int1 <= int2))</code>	<code>int1</code> 小于或等于 <code>int2</code>



编程小结：条件测试

□ 逻辑测试

[<code>expr1 -a expr2</code>]	逻辑与，都为真时，结果为真
[<code>expr1 -o expr2</code>]	逻辑或，有一个为真时，结果为真
[<code>! expr</code>]	逻辑非

如果使用双方括号，可以使用 **通配符** 进行模式匹配。

[[<code>pattern1 && pattern2</code>]]	逻辑与
[[<code>pattern1 pattern2</code>]]	逻辑或
[[<code>! pattern</code>]]	逻辑非



编程小结：条件测试

□ 文件测试

-f <code>fname</code>	<code>fname</code> 存在且是普通文件时，返回真（即返回 0）
-L <code>fname</code>	<code>fname</code> 存在且是链接文件时，返回真
-d <code>fname</code>	<code>fname</code> 存在且是一个目录时，返回真
-e <code>fname</code>	<code>fname</code> （文件或目录）存在时，返回真
-s <code>fname</code>	<code>fname</code> 存在且大小大于 0 时，返回真
-r <code>fname</code>	<code>fname</code> （文件或目录）存在且可读时，返回真
-w <code>fname</code>	<code>fname</code> （文件或目录）存在且可写时，返回真
-x <code>fname</code>	<code>fname</code> （文件或目录）存在且可执行时，返回真



编程小结：控制结构

- ❑ `if` 条件语句
- ❑ `case` 选择语句
- ❑ `for` 循环语句
- ❑ `while` 循环语句
- ❑ `until` 循环语句
- ❑ `break`、`continue`、`sleep` 命令
- ❑ `select` 循环与菜单
- ❑ `shift` 命令
- ❑ 各种括号的作用
 - `${...}`，`$(...)`，`$[...]`，`$((...))`
 - `[...]`，`[[...]]`，`((...))`



函数

- ❑ 和其它编程语言一样，**Bash** 也可以定义函数。
- ❑ 一个函数就是一个子程序，用于完成特定的任务，当有重复代码，或者一个任务只需要很少的修改就被重复几次执行时，这时你应考虑使用函数。
- ❑ 函数的一般格式

```
function function_name {  
    commands  
}
```

```
function_name () {  
    commands  
}
```




函数举例

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

```
fun1 () { echo "This is a function"; echo; }
```

一个函数可以写成一行，但命令之间必须用分号隔开

特别注意，最后一个命令后面也必须加分号

```
fun2 ()
```

```
{
```

```
    echo "This is fun2."
```

```
    echo "Now exiting fun2."
```

```
}
```



函数的调用

- ❑ 只需输入函数名即可调用该函数。
- ❑ 函数必须在调用之前定义

```
#!/bin/bash

fun2 ()
{
    echo "This is fun2."
    echo "Now exiting fun2."
}

fun2 # 调用函数 fun2
```

例: `ex4fun2.sh`, `ex4fun3.sh`



函数的调用

❑ 向函数传递参数

例: `ex4fun4.sh`

❑ 函数与命令行参数

例: `ex4fun5.sh`

❑ `return` 与 `exit`

例: `ex4fun6.sh`