Linux 系统概论

天津医科大学
生物医学工程与技术学院

2018-2019 学年下学期（春）
2017 级生信班

第八章　shell 编程

伊现富（Yi Xianfu）

天津医科大学（TIJMU）
生物医学工程与技术学院

2019 年 6 月

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

```
#!/bin/bash
read -p "Input number of lines: " var
for i in $(seq $var $((2*$var-1))); do
  for j in $(seq $i); do
    if [ $j -lt $var ]; then
      echo -e " \c"
    else
      k=$((RANDOM%10)); m=$((RANDOM%7)); n=$((RANDOM%7))
      case $k in
        3)
          echo -e "\033[3$m;5m□\033[0m\c"
          ;;
        *)
          echo -e "\033[4$m;3$n;6m♥\033[0m\c"
          ;;
      esac
    fi
  done
  ((var--))
  echo
done
```

Input number of lines: 30

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

引言 | shell 的职责

shell 脚本

shell 程序通常称为脚本（script），是存储在单个文件中的一系列系统命令。
每次调用 shell 脚本时，其中的命令就会自动地依次执行。
shell 脚本可用于很多方面，特别是用于系统管理。通过 shell 脚本，可以利用少数的简单脚本自动完成大量的工作。
shell 脚本实质上是一组按顺序执行的命令。

```
#!/bin/bash
# This script will test if we're in a leap year or not.

year=`date +%Y`

if (( ("$year" % 400) == "0" )) || (( ("$year" % 4 == "0") && ("$year" % 100 !=
"0") )); then
  echo "This is a leap year.  Don't forget to charge the extra day!"
else
  echo "This is not a leap year."
fi
```

```
#!/bin/bash
#This script can crreate user1 to user10 auth.
#I will show you how to set passwd in script.
#Data:2013/12/14
#BY:atiger77

for i in `seq 1 10`
do
   useradd user$i
   echo "redhat" |passwd --stdin user$i
done
```

引言 | shell 脚本

Scripts are a sequence of statements and commands stored in a file that can be executed by a shell. The most commonly used shell in Linux is **bash**.

Shell scripts are used to execute sequences of commands and other types of statements. Commands can be divided into the following categories:

- Compiled applications
- Built-in **bash** commands
- Other scripts

Compiled applications are binary executable files that you can find on the filesystem. The shell script always has access to compiled applications such as **rm**, **ls** , **df** , **vi** , and **gzip**.

bash has many **built-in** commands which can only be used to display the output within a terminal shell or shell script. Sometimes these commands have the same name as executable programs on the system, such as **echo** which can lead to subtle problems. **bash** built-in commands include and `cd`, `pwd`, `echo`, `read`, `logout`, `printf`, `let`, and `ulimit`.

A complete list of **bash** built-in commands can be found in the **bash man** page, or by simply typing `help`.

引言 | shell 脚本 | 用途

```
#!/bin/bash
#This script can crreate user1 to user10 auth.
#I will show you how to set passwd in script.
#Data:2013/12/14
#BY:atiger77

for i in `seq 1 10`
do
   useradd user$i
   echo "redhat" |passwd --stdin user$i
done
```

```
#!/bin/bash
if
        [ "$1" = "kernel" ];then
        echo 'user'
elif
        [ "$1" = "user" ];then
        echo 'kernel'
else
        echo 'usage:/root/program kernel | user'
fi
```

```
#!/bin/bash
# This script will test if we're in a leap year or not.

year=`date +%Y`

if (( ("$year" % 400) == "0" )) || (( ("$year" % 4 == "0") && ("$year" % 100 !=
"0") )); then
  echo "This is a leap year.  Don't forget to charge the extra day!"
else
  echo "This is not a leap year."
fi
```

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

```
#!/bin/bash

#This is to show what a example looks like.

echo "Our first example!"
echo # This inserts an empty line in output.
echo "We are currently in the following directory:"
pwd
echo
echo "This directory contains the following files:"
ls
```

shell 脚本的结构

- #! 调用 shell（指定执行脚本的 shell）。
- # 进行注释。
- 命令和控制结构。
- shell 命令，可以按照分号分隔，也可以按照换行符分隔。
- 如果想在一行中写入多个命令，可以通过分号（;）分隔。

创建 shell 脚本的步骤

1. 创建一个包含命令和控制结构的文件。`vim script.sh`
2. 修改文件的权限使它可执行。`chmod u+x script.sh`
3. 运行文件。`./script.sh` 或 `bash script.sh`（不需要可执行权限）

shell 脚本的结构

- #! 调用 shell（指定执行脚本的 shell）。
- # 进行注释。
- 命令和控制结构。
- shell 命令，可以按照分号分隔，也可以按照换行符分隔。
- 如果想在一行中写入多个命令，可以通过分号（;）分隔。

创建 shell 脚本的步骤

1. 创建一个包含命令和控制结构的文件。`vim script.sh`
2. 修改文件的权限使它可执行。`chmod u+x script.sh`
3. 运行文件。`./script.sh` 或 `bash script.sh`（不需要可执行权限）

shell脚本运行方式

例子:helloworld.sh

```
#!/bin/bash
#这是一个打印"hello world"的shell 脚本
echo "hello world"
```

脚本学堂 www.jbxue.com

- chmod u+x helloword.sh
- ./helloword

编程起步 | 脚本运行

1. vi /root/welcome

2.
```
#!/bin/bash
#filename:welcome
first ()
{
echo "==================================="
echo "Hello! Everyone! Welcome to the Linux World!"
echo "==================================="
}
second ()
{
echo "*****************************************"
}

first
second
second
first
```

3. chmod 777 /root/welcome
4. ./root/welcome

source /root/welcome

教学提纲

1 引言
2 编程起步
- 脚本结构与运行
- 调用 shell
- 特殊字符
- 变量
- 从键盘读取输入
- 特殊变量
- 退出状态

3 流程控制
- 条件流程控制
- 选择流程控制
- 迭代流程控制

4 高级概念
- 输入输出重定向
- 并行
- 命令替换
- 通配

5 shell 函数
- 声明与调用
- 返回值
- 嵌套和递归
- 作用域
- 文件处理
- 数组
- 关联数组

6 脚本调试
7 回顾与总结
- 总结
- 思考题

编程起步 | 调用 shell

- 在向脚本中添加任何其他内容之前，用户需要告诉系统将要启动一个 shell 脚本。
- 使用 shebang 结构，如：`#!/bin/bash` 告诉系统接下来的命令由 bash shell 执行。
- 要创建一个脚本，用户应该首先添加 shebang 行，然后添加命令。
- shebang 是因为 # 称为 hash，！称为 bang。

```
$cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
/bin/tcsh
/usr/bin/tcsh
/usr/bin/tmux
/bin/ksh93
/usr/bin/screen
/bin/zsh
/usr/bin/zsh
```

编程起步 | 调用 shell | 实例

```
#!/bin/bash

# /home/yixf/dirinfo
# 2014年 06月 24日
# A really stupid script to give some basic info about the current directory

pwd
ls
```

教学提纲

1 引言
2 编程起步
- 脚本结构与运行
- 调用 shell
- 特殊字符
- 变量
- 从键盘读取输入
- 特殊变量
- 退出状态

3 流程控制
- 条件流程控制
- 选择流程控制
- 迭代流程控制

4 高级概念
- 输入输出重定向
- 并行
- 命令替换
- 通配

5 shell 函数
- 声明与调用
- 返回值
- 嵌套和递归
- 作用域
- 文件处理
- 数组
- 关联数组

6 脚本调试
7 回顾与总结
- 总结
- 思考题

编程起步 | 特殊字符

Character	Description
#	Used to add a comment, **except** when used as \#, or as #! when starting a script
\	Used at the end of a line to indicate continuation on to the next line
;	Used to interpret what follows as a new command
$	Indicates what follows is a variable

好的注释对好的程序而言非常关键：为什么编写它、脚本中各部分的作用、如何使用脚本、……

shell 脚本是独立的文档，因此注释它们的最简单也是最好的方法是使用脚本自己的注释。

注释方法

- 注释行以散列符号（#）开头，在 # 之后出现的所有内容都认为是注释，不解释为命令。
- 占用多行的注释在每行的开头都有一个 #。
- 可以在一行的中间插入 #，# 右边的所有内容都将看作注释。

```
# Here is a comment that
# spans multipe lines.
command # This is comment explaining the cmd
```

注释方法

- 注释行以散列符号（#）开头，在 # 之后出现的所有内容都认为是注释，不解释为命令。
- 占用多行的注释在每行的开头都有一个 #。
- 可以在一行的中间插入 #，# 右边的所有内容都将看作注释。

```
# Here is a comment that
# spans multipe lines.
command # This is comment explaining the cmd
```

- 在脚本的顶部放置一些基本信息。
  包括：脚本的名称、日期、脚本的总体作用、命令行参数的正确语法、脚本的不同寻常之处、……
- 保存脚本的修改日志。
  将每个修改添加到修改日志的顶部。
- 注释脚本的每个部分。
  说明每个部分的作用或目的。
- 识别必须由用户添加的数据。
  解释需要添加的信息以及该信息需要的格式。

编程起步 | 特殊字符 | Splitting Long Commands Over Multiple Lines

Users sometimes need to combine several commands and statements and even conditionally execute them based on the behaviour of operators used in between them. This method is called **chaining of commands**.

The **concatenation operator** (\) is used to concatenate large commands over several lines in the shell.

The command is divided into multiple lines to make it look readable and easier to understand. The \ operator at the end of each line combines the commands from multiple lines and executes it as one single command.

Line 1 : Command 1 starts here	`scp abc@server1.linux.com:\`
Line 2 : Command 1 continues	`/var/ftp/pub/userdata/custdata/read \`
Line 3 : Command 1 continues	`abc@server3.linux.co.in\`
Line 4 : Command 1 ends	`:/opt/oradba/master/abc/`

编程起步 | 特殊字符 | Putting Multiple Commands on a Single Line

Sometimes you may want to group multiple commands on a single line. The `;` (semicolon) character is used to separate these commands and execute them sequentially as if they had been typed on separate lines.

The three commands in the following example will all execute even if the ones preceding them fail: `make; make install; make clean`

However, you may want to abort subsequent commands if one fails. You can do this using the `&&` (and) operator as in: `make && make install && make clean`. If the first command fails the second one will never be executed.

A final refinement is to use the `||` (or) operator as in: `cat file1 || cat file2 || cat file3`. In this case, you proceed until something succeeds and then you stop executing any further steps.

教学提纲

1 引言
2 编程起步
- 脚本结构与运行
- 调用 shell
- 特殊字符
- 变量
- 从键盘读取输入
- 特殊变量
- 退出状态

3 流程控制
- 条件流程控制
- 选择流程控制
- 迭代流程控制

4 高级概念
- 输入输出重定向
- 并行
- 命令替换
- 通配

5 shell 函数
- 声明与调用
- 返回值
- 嵌套和递归
- 作用域
- 文件处理
- 数组
- 关联数组

6 脚本调试
7 回顾与总结
- 总结
- 思考题

简介

shell 传递数据的一种方法，用来代表每个取值的符号名。

分类

- 临时变量：shell 程序内部定义的，其使用范围仅限于定义它的程序，对其它程序不可见。包括：用户自定义变量、位置变量。
- 永久变量：环境变量，其值不随 shell 脚本的执行结束而消失。

简介

shell 传递数据的一种方法，用来代表每个取值的符号名。

分类

- 临时变量：shell 程序内部定义的，其使用范围仅限于定义它的程序，对其它程序不可见。包括：用户自定义变量、位置变量。
- 永久变量：环境变量，其值不随 shell 脚本的执行结束而消失。

编程起步 | 变量 | 使用

- 使用赋值运算符 = 对变量进行赋值，= 两边不能有空格。
- 如果要给变量赋空值，可以在 = 后面跟一个换行符。
- 在 bash shell 中，默认情况下将变量视作文本字符串。
- 通过在变量名前加一个美元符号 $ 来访问该变量的值。
- 如果变量名的前面有美元符号 $，那么可以将该变量替换成一个表达式。
- 变量名对大小写敏感。按照惯例，Linux 变量用大写字母表示。
- 变量名长度没有限制，由字母、数字或下划线序列组成。
- 变量名必须以字母或下划线开头，不能用数字。
- 通常情况下，变量名最好使用字符和数字，并且以数字结尾。
- 使用 set 命令列出所有的变量，使用 unset 命令删除变量。

编程起步 | 变量 | Introduction

Almost all scripts use **variables** containing a value, which can be used anywhere in the script. These variables can either be user or system defined. Many applications use such **environment variables** for supplying inputs, validation, and controlling behaviour.

Some examples of standard environment variables are `HOME`, `PATH`, and `HOST`. When referenced, environment variables must be prefixed with the `$` symbol as in `$HOME`. You can view and set the value of environment variables. For example, the following command displays the value stored in the `PATH` variable: `echo $PATH`.

However, no prefix is required when setting or modifying the variable value. For example, the following command sets the value of the `MYCOLOR` variable to blue: `MYCOLOR=blue`.

You can get a list of environment variables with the **env**, **set**, or **printenv** commands.

编程起步 | 变量 | Exporting

By default, the variables created within a script are available only to the subsequent steps of that script. Any child processes (sub-shells) do not have automatic access to the values of these variables. To make them available to child processes, they must be promoted to environment variables using the **export** statement as in: `export VAR=value`, or `VAR=value; export VAR`.

While child processes are allowed to modify the value of exported variables, the parent will not see any changes; exported variables are not shared, but only copied.

```
# 变量的赋值与访问
EDITOR="vim"
echo $EDITOR # 输出: vim

num=2
echo "this is the ${num}nd" # this is the 2nd

# 默认将变量视作文本字符串
VAR=1
VAR=$VAR+1
echo $VAR # 输出文本字符串“1+1”，而不是数字2

# 变量替换（双引号）
PERSON="Fred"
echo "Hello, $PERSON" # 输出: Hello, Fred
echo 'Hello, $PERSON' # 输出: Hello, $PERSON
```

引号

在向程序传递任何参数之前，程序会扩展通配符和变量。这里所谓的扩展是指程序会把通配符（比如 *）替换成适当的文件名，把变量替换成变量值。我们可以使用引号来防止这种扩展。

- 引号（单引号和双引号）可以防止通配符 * 的扩展
- 单引号更严格一些，它可以防止任何变量扩展
- 双引号可以防止通配符扩展但允许变量扩展
- 还有一种防止这种扩展的方法，即使用转义字符——反斜线（\）

```
#!/bin/bash

# 通配符扩展
echo *.jpg
# 防止通配符扩展
echo "*.jpg" # 输出: *.jpg
echo '*.jpg' # 同上
echo \*.jpg  # 同上

# 变量展开
echo $SHELL    # 输出: /bin/bash
echo "$SHELL" # 同上
# 防止变量展开
echo '$SHELL' # 输出: $SHELL
echo \$SHELL  # 同上
```

教学提纲

1 引言
2 编程起步
- 脚本结构与运行
- 调用 shell
- 特殊字符
- 变量
- 从键盘读取输入
- 特殊变量
- 退出状态

3 流程控制
- 条件流程控制
- 选择流程控制
- 迭代流程控制

4 高级概念
- 输入输出重定向
- 并行
- 命令替换
- 通配

5 shell 函数
- 声明与调用
- 返回值
- 嵌套和递归
- 作用域
- 文件处理
- 数组
- 关联数组

6 脚本调试
7 回顾与总结
- 总结
- 思考题

编程起步 | 从键盘读取输入

用户可以通过 read 命令读取键盘的输入值为变量赋值。

```
# read读取键盘输入，并赋值给变量PERSON
echo "What is your name?"
read PERSON

echo "Hello, $PERSON"
```

```
# read读取键盘输入，并赋值给变量PERSON
read -p "What is your name?" PERSON

echo "Hello, $PERSON"
```

用户可以通过 read 命令读取键盘的输入值为变量赋值。

```
# read读取键盘输入，并赋值给变量PERSON
echo "What is your name?"
read PERSON

echo "Hello, $PERSON"
```

```
# read读取键盘输入，并赋值给变量PERSON
read -p "What is your name?" PERSON

echo "Hello, $PERSON"
```

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

变量	使用	功能
?	$?	上一个命令的退出状态
$	$$	当前 shell 进程的 PID
!	$!	后台运行的上一个命令的 PID
0	$0	当前脚本的文件名
#	$#	传递给脚本的参数个数
1-9	$1	传递给脚本的第 1（2,3,...,9）个参数
*	$*	传递给脚本的所有参数（作为一个单词）
@	$@	传递给脚本的所有参数的列表
_	$_	上一个命令的最后一个参数

编程起步 | 特殊变量

内置变量		含义
$?		最后一次执行的命令的返回码
$$		shell进程自己的PID
$!		shell进程最近启动的后台进程的PID
位置变量	$#	命令行参数的个数(不包括脚本文件的名字在内)
	$0	脚本文件本身的名字
	$1 $2,...	第一个,第二个,....,命令行参数
	"$*"	"$1 $2 $3 $4...",将所有命令行参数组织成一个整体,作为一个单词
	"$@"	"$1" "$2" "$3"...,将多个命令行参数看作是多个"单词"

编程起步 | 特殊变量 | 脚本参数

Users often need to pass parameter values to a script, such as a filename, date, etc. Scripts will take different paths or arrive at different values according to the parameters (command arguments) that are passed to them. These values can be text or numbers. Within a script, the parameter or an argument is represented with a $ and a number.

Parameter	Meaning
$0	Script name
$1	First parameter
$2, $3, etc.	Second, third parameter, etc.
$*	All parameters
$#	Number of arguments

```
#!/bin/bash

echo The name of this program is: $0

echo The number of arguments is: $#

echo The first argument is: $1
echo The second argument is: $2

echo All of the arguments are : $*
```

Objective of the script: To display the arguments passed on the command prompt.

Steps:

1. Display the name of the script, which is in $0
2. Display the value of first argument/parameter passed from terminal, which is in $1
3. Display the value of second argument/parameter passed from terminal, which is in $2
4. Display the value of third argument/parameter passed from terminal, which is in $3

```
1 #!/bin/bash
2 echo "The name of the script being executed is:" $0
3 echo "The first parameter passed is : " $1
4 echo "The second parameter passed is : " $2
5 echo "The third parameter passed is : " $3
6
```

```
[root@pc05 ~]# cat test.sh
#!/bin/bash
echo "本程序名：$0"
echo "执行时一共输入 $# 个位置参数"
echo "其中第一个参数是：$1"
echo "所有的参数如下：$*"
[root@pc05 ~]# ./test.sh Hello Everybody!
本程序名：./test.sh
执行时一共输入 2 个位置参数
其中第一个参数是：Hello
所有的参数如下：Hello Everybody!
```

教学提纲

1 引言
2 编程起步
- 脚本结构与运行
- 调用 shell
- 特殊字符
- 变量
- 从键盘读取输入
- 特殊变量
- 退出状态

3 流程控制
- 条件流程控制
- 选择流程控制
- 迭代流程控制

4 高级概念
- 输入输出重定向
- 并行
- 命令替换
- 通配

5 shell 函数
- 声明与调用
- 返回值
- 嵌套和递归
- 作用域
- 文件处理
- 数组
- 关联数组

6 脚本调试
7 回顾与总结
- 总结
- 思考题

退出状态

- 退出状态（exit status）是每个命令在完成时返回的一个数值。
- 通常情况下，如果命令执行成功，返回退出状态 0；如果不成功，则返回 1（非零的数值，最大是 255）。
- 有些命令由于特殊原因会返回额外的退出状态。
- `exit 0`

```
[root@pc05 ~]# mkdir /mulua
[root@pc05 ~]# echo $?
0
[root@pc05 ~]# mkdir /mulu/a
mkdir: 无法创建目录 "/mulu/a" : 没有那个文件或目录
[root@pc05 ~]# echo $?
1
```

```
[test3@CentOS ~]$ ls /etc/passwd
/etc/passwd
[test3@CentOS ~]$ echo $?
0
[test3@CentOS ~]$ ls filethatdoesnotexist
ls: cannot access filethatdoesnotexist: No such file or directory
[test3@CentOS ~]$ echo $?
2
```

状态码	描述
0	命令成功结束
1	一般性未知错误
2	不适合的 shell 命令
126	命令无法执行
127	没有找到命令
128	无效的退出参数
128+x	与 Linux 信号 x 相关的严重错误
130	通过 Ctrl+C 终止的命令
255	正常范围之外的退出状态码

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

流程控制 | 简介

流程控制

流程控制（flow control）允许程序做出判断：程序计算条件的值，并根据这些条件执行相应的操作。

分类

- 条件流程控制（conditional flow control）：根据特定的约束是否满足来决定是否执行某个代码段
- 迭代流程控制（iterative flow control）：代码块重复或迭代，直到满足某个条件为止

流程控制 | 简介

流程控制

流程控制（flow control）允许程序做出判断：程序计算条件的值，并根据这些条件执行相应的操作。

分类

1. 条件流程控制（conditional flow control）：根据特定的约束是否满足来决定是否执行某个代码段
2. 迭代流程控制（iterative flow control）：代码块重复或迭代，直到满足某个条件为止

流程控制 | 条件 vs. 迭代

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

Conditional decision making using an **if** statement, is a basic construct that any useful programming or scripting language must have.

When an **if** statement is used, the ensuing actions depend on the evaluation of specified conditions such as:

- Numerical or string comparisons
- Return value of a command (0 for success)
- File existence or permissions

```
# compact form
if TEST-COMMANDS; then CONSEQUENT-COMMANDS;
  fi

# more general definition
if condition
then
  statements
else
  statements
fi

# example
if [ -f /etc/passwd  ] ; then
  cat /etc/passwd
fi
```

Objective of the script: To accept a file name from command prompt and display its existence.

Steps:

1. Accept the command line argument and store in 'file1'
2. Check for its existence and if found true
   1. Display the *"File exist"*
3. Else
   1. Display the *"File doesn't exist"*

```
#!/bin/bash
file=$1
if [ -f $file ]
then
 echo -e "The $file exist"
else
 echo -e "The $file does not exist"
fi
```

Objective of the script: To accept a number and display whether it is equal to or greater or less than 100.

Steps:
1. Display the message to enter a number
2. Read the number
3. If given number=100
    1. Display "Count is 100"
4. Else If number>100
    1. Display "Count is greater than 100"
5. Else
    1. Display "Count is less than 100"

```bash
#!/bin/bash
echo "enter a number"
read count
if [ $count -eq 100 ]
then
  echo "Example-3: Count is 100"
elif [ $count -gt 100 ]
then
  echo "Example-3: Count is greater than 100"
else
  echo "Example-3: Count is less than 100"
fi
```

Objective of the script: To accept 2 numbers, 1 operator and display the calculated value based on the operator.

Steps:

1. Display a message to enter first number
2. Read the first number
3. Display a message to enter second number
4. Read the second number
5. Display a message to enter the option for the operation
6. Read the option number
7. If operator is 1
   1. Display the sum of 2 numbers
8. Else if operator is 2
   1. Display the difference of 2 numbers
9. Else if operator is 3
   1. Display the product of 2 numbers
10. Else
    1. Display Invalid input

```
1 #!/bin/bash
2 echo "Enter the first number"
3 read inp1
4 echo "Enter the second number"
5 read inp2
6 echo "1. Addition"
7 echo "2. Subtraction"
8 echo "3. Multiplication"
9 echo -n "Please choose a word [1,2 or 3]? "
10 read oper
11 if [ $oper -eq 1 ]
12 then
13         echo "Addition Result " $(($inp1 + $inp2))
14 else
15         if [ $oper -eq 2 ]
16         then
17                 echo "Subtraction Result " $(($inp1 - $inp2))
18         else
19                 if [ $oper -eq 3 ]
20                 then
21                         echo "Multiplication Result " $(($inp1 * $inp2))
22                 else
23                         echo "Invalid input"
24                 fi
25         fi
26 fi
```

流程控制 | 条件流程控制 | if-then | Example

```bash
#!/bin/bash
# Check if Test1.txt exists
if [ -e test1.txt ]; then
   echo "test1.txt exists. Continue processing."
else

 # else print the second statement and check the next validation.
   echo "test1.txt does not exist. Check temp directory."

# Check if the temp directory exists and  print first statement if TRUE
   if [ -d temp ]; then
      echo "temp directory exists. Create test1.txt in temp."

# create temporary file named Test1.txt
      touch temp/test1.txt
   else

 # else print second statement and perform next validation.
      echo "temp directory does not exist. Is sym link there"
      if [ -s sym-link ]; then
         echo "sym-link exists. All is good."
      else
         echo "sym-link does not exist. Bail out."
         exit 2
      fi
   fi
fi
```

条件为真 then
命令序列...
if 条件测试命令
fi 结束判断
条件为假

```
if some_condition
then
  something happens
fi
```

流程控制 | 条件流程控制 | if-then | 实例

```
#!/bin/bash
echo "Guess the secret color"
read COLOR
if [ $COLOR = "purple" ]
then
  echo "You are correct."
fi
```

```
#!/bin/bash
if [ ${SHELL} = "/bin/bash" ]; then
  echo "your login shell is the bash"
else
  echo "your login shell is ${SHELL}"
fi
```

流程控制 | 条件流程控制 | if-then | 实例

```
#!/bin/bash
echo "Guess the secret color"
read COLOR
if [ $COLOR = "purple" ]
then
  echo "You are correct."
fi
```

```
#!/bin/bash
if [ ${SHELL} = "/bin/bash" ]; then
  echo "your login shell is the bash"
else
  echo "your login shell is ${SHELL}"
fi
```

条件为真 then
命令序列 1...
if 条件测试命令
fi 结束判断
条件为假 else
命令序列 2...

```
if some_condition
then
  something happens
else
  something happens
fi
```

```
#!/bin/bash

echo "Guess the secret color"
read COLOR

if [ $COLOR = "purple" ]
then
  echo "You are correct."
else
  echo "Your guess was incorrect."
fi
```

条件为真 then
命令序列 1...
if 条件测试命令 1
条件为真 then
命令序列 2...
elif 条件测试命令 2
fi 结束判断
.....
.....
条件为假 else
命令序列 n...

```
if some_condition
then
  something happens
elif other_condition
then
  something happens
else
  something happens
fi
```

流程控制 | 条件流程控制 | if-then | 实例

```
#!/bin/bash

echo "Guess the secret color"
read COLOR

if [ $COLOR = "purple" ]
then
  echo "You are correct."
elif [ $COLOR = "blue" ]
then
  echo "You are close."
else
  echo "Your guess was incorrect."
fi
```

```
[root@pc05 ~]# cat chkdir.sh
#!/bin/bash
BACKUP_DIR="/opt/mrepo/"
if [ ! -d $BACKUP_DIR ]
then
    mkdir -p $BACKUP_DIR
fi
```

```
[root@pc05 ~]# cat chkhost.sh
#!/bin/bash
ping -c 3 -i 0.2 -W 3 $1 &> /dev/null
if [ $? -eq 0 ]
then
    echo "Host $1 is up."
else
    echo "Host $1 is down."
fi
```

```
[root@pc05 ~]# cat gradediv.sh
#!/bin/bash
read -p "请输入您的分数（0-100）：" GRADE
if [ $GRADE -ge 85 ] && [ $GRADE -le 100 ] ; then
    echo "$GRADE 分！优秀"
elif [ $GRADE -ge 70 ] && [ $GRADE -le 84 ] ; then
    echo "$GRADE 分，合格"
else
    echo "$GRADE 分？不合格"
fi
```

但使用 [ ] 要特别注意，在上述每个组件中间，都要用空格分隔，假设空格使用“□”来表示：

```
[  "$HOME"  ==  "$MAIL"  ]
[□"$HOME"□==□"$MAIL"□]
 ↑        ↑  ↑        ↑
```

上面的例子表示两个字符串 $HOME 与 $MAIL 是否相同，相当于 test $HOME = $MAIL。如果没有空格分隔，例如 [$HOME==$MAIL]，bash 就会显示错误信息，要特别注意。所以，最好要注意：

- 中括号 [ ] 内的每个组件都需要用空格来分隔。
- 中括号内的变量，最好用双引号来设置。
- 中括号内的常量，最好用单引号或双引号来设置。

- 使用test命令
  - 语法：test 条件表达式
- 使用括号[ ]
  - 语法：[空格 条件表达式 空格]
- 条件表达式的值为真返回零，为假时返回非零值

test

- test 命令用于计算条件的值。
- 方括号在语法上与 test 命令等价。
- 方括号中的空格很重要，要确保方括号前后的空格。

```
if [ $COLOR = "purple" ]
if (test $COLOR = "purple")

# 测试文件是否存在
if [ -e filename ]
if (test -e filename)
```

test

- test 命令用于计算条件的值。
- 方括号在语法上与 test 命令等价。
- 方括号中的空格很重要，要确保方括号前后的空格。

```
if [ $COLOR = "purple" ]
if (test $COLOR = "purple")

# 测试文件是否存在
if [ -e filename ]
if (test -e filename)
```

测试	助记	功能
-d	Directory	文件存在并且是目录
-e	Exist	指定的文件存在
-f	File	文在存在并且是普通文件
-G	Group	执行命令的用户属于文件所属组的成员
-nt	Newer Than	file1 -nt file2, 前一个文件比后一个文件新
-ot	Older Than	file1 -ot file2, 前一个文件比后一个文件老
-O	Owner	执行命令的用户是文件的所有者
-r	Read	执行命令的用户拥有对文件的读取权限
-s	Size	文在存在并且不为空
-w	Write	执行命令的用户拥有对文件的写入权限
-x	eXecute	执行命令的用户拥有对文件的执行权限

You can view the full list of file conditions using the command `man 1 test`.

流程控制 | 条件流程控制 | test | 文件测试

Condition	Meaning
`-e file`	Check if the file exists.
`-d file`	Check if the file is a directory.
`-f file`	Check if the file is a regular file (i.e., not a symbolic link, device node, directory, etc.)
`-s file`	Check if the file is of non-zero size.
`-g file`	Check if the file has *sgid* set.
`-u file`	Check if the file has *suid* set.
`-r file`	Check if the file is readable.
`-w file`	Check if the file is writable.
`-x file`	Check if the file is executable.

存在及识别

- -e：目标是否存在（Exist）
- -d：是否为目录（Directory）
- -f：是否为文件（File）

示 例

```
[root@pc05 ~]# [ -f "/etc/fstab" ] && echo YES
YES
[root@pc05 ~]# [ -f "/etc" ] && echo YES
[root@pc05 ~]#
```

权限的检测

- -r：是否有读取（Read）权限
- -w：是否有写入（Write）权限
- -x：是否有可执行（eXcute）权限

示例1

```
[root@pc05 ~]# ls -l /etc/shadow
-r--------- 1 root root 1386 11-08 18:43 /etc/shadow
[root@pc05 ~]# [ -x "/etc/shadow" ] && echo YES
[root@pc05 ~]# [ -r "/etc/shadow" ] && echo YES
YES
```

运算符	示例	功能
=	stringA = stringB	stringA 与 stringB 相同，与 == 等价
!=	stringA != stringB	stringA 与 stringB 不同
>	stringA > stringB	stringA 大于 stringB
<	stringA < stringB	stringA 小于 stringB
-z	-z string	string 为空
-n	-n string	string 非空

字符串匹配

- =：两个字符串相同
- !=：两个字符串不相同

示 例

```
[root@pc05 ~]# echo $USER
root
[root@pc05 ~]# [ $USER = "root" ] && echo YES
YES
[root@pc05 ~]# [ $USER != "nobody" ] && echo YES
YES
```

```
#!/bin/bash
# Section that reads the input
echo " Enter any color code [R OR Y OR G] :"
read COLOR
echo $COLOR
# Section that compares the entry and display a message
if [ "$COLOR" == "R" ]
then
echo "STOP! LEAVE WAY FOR OTHERS"
elif [ "$COLOR" == "Y" ]
then
echo "GET READY YOUR WAY WILL BE OPEN SHORTLY"
elif [ "$COLOR" == "G" ]
then
echo "MOVE.. IT IS UR TURN TO GO"
else
echo "INCORRECT COLOR CODE"
fi
```

A **string variable** contains a sequence of text characters. It can include letters, numbers, symbols and punctuation marks. Some examples: `abcde`, `123`, `abcde 123`, `abcde-123`, `&acbde=%123`.

String **operators** include those that do comparison, sorting, and finding the length.

Operator	Meaning
`[ string1 > string2 ]`	Compares the sorting order of string1 and string2.
`[ string1 = string2 ]`	Compares the characters in string1 with the characters in string2.
`myLen1=${#string1}`	Saves the length of string1 in the variable myLen1.

At times, you may not need to compare or use an entire string.

To extract the first character of a string we can specify: `${string:0:1}`. Here 0 is the offset in the string (i.e., which character to begin from) where the extraction needs to start and 1 is the number of characters to be extracted.

To extract all characters in a string after a dot (.), use the following expression: `${string#*.}`.

```
[test1@localhost ~]$ export name="bagend.hobbiton.com"
[test1@localhost ~]$ loco=${name:0:6}; echo $loco
bagend
[test1@localhost ~]$ moto=${name#*.}; echo $moto
hobbiton.com
[test1@localhost ~]$
```

Objective of the script: To display the status of ping operation

Steps:

1. Display a message asking for the IP Address
2. Accept the IP address
3. If given IP is not null, then
   1. Ping the IP address
   2. If return value of the *ping* command is 0, then
      1. Display a "positive message"

      else
      1. Display a "negative message"

   else
   1. Display a message saying "IP address is empty"

```
1 #!/bin/bash
2 echo "Enter the Ipaddress"
3 read ip
4
5 if [ ! -z $ip ]
6 then
7         ping -c 1 $ip
8         if [ $? -eq 0 ] ; then
9                 echo "Machine is giving ping response"
10        else
11                echo "Machine is not pinging"
12        fi
13 else
14        echo "IP Address is empty"
15 fi
```

Objective of the script: To accept the string/data passed from keyboard, manipulate them and display on the terminal.

Steps:

1. Ask the user to type a string
2. Accept the message typed by the user
3. Ask the user to type another string
4. Accept the message typed by the user
5. Display the two names
6. Compute the length of the first and second string
7. Display the length of the first and second string

```
echo "Enter the first string"
read str1
echo "Enter the second string"
read str2
echo $str1
echo $str2
myLen1=${#str1}
myLen2=${#str2}
echo Length of the first string is: $myLen1
echo Length of the second string is: $myLen2
```

```
[mgnanda@localhost ~]$ chmod a+x sc2.sh
[mgnanda@localhost ~]$ bash sc2.sh
Enter the first string
Robert
Enter the second string
USA
Robert
USA
Length of the first string is: 6
Length of the second string is: 3
```

运算符	助记	功能
-eq	EQual	等于
-ne	Not Equal	不等于
-gt	Greater Than	大于
-lt	Lesser Than	小于
-ge	Greater or Equal	大于等于
-le	Lesser or Equal	小于等于

```
#!/bin/bash
# Prompt for a user name...
echo "Please enter your age:"
read AGE
if [ "$AGE" -lt 20 ] || [ "$AGE" -ge 50 ] ; then
echo "Sorry, you are out of the age range."
elif [ "$AGE" -ge 20 ] && [ "$AGE" -lt 30 ] ; then
echo "You are in your 20s"
elif [ "$AGE" -ge 30 ] && [ "$AGE" -lt 40 ] ; then
echo "You are in your 30s"
elif [ "$AGE" -ge 40 ] && [ "$AGE" -lt 50 ] ; then
echo "You are in your 40s"
fi
```

Objective of the script: To display whether two numbers are zeroes or equal or larger or smaller

Steps:

1. Ask the user to enter the first number
2. Read the first number
3. Ask the user to enter the second number
4. Read the second number
5. Compare the first number and second number are equal to zero, if found true
   1. Display *Both number are zero*
6. If not then compare whether first number is equal to second number, if found true
   1. Display *both numbers are equal*
7. If not then compare whether first number is greater than second number, if found true
   1. Display *first number is greater than second number*

   else
   1. Display the *first number is less than second number*

```bash
#!/bin/bash
echo "Please enter first number"
read first
echo "Please enter second number"
read second

if [ $first -eq 0 ] && [ $second -eq 0 ]
then
        echo "Num1 and Num2 are zero"
elif [ $first -eq $second ]
then
        echo "Both Values are equal"
elif [ $first -gt $second ]
then
        echo "$first is greater than $second"
else
        echo "$first is lesser than $second"
fi
```

$(( )) shell 内置的算数运算格式

$[ ] 将中括号内的表达式作为数学运算先计算结果再输出

let 表示数学运算

expr 用于整数值运算，每一项用空格隔开

Arithmetic expressions can be evaluated in the following three ways (spaces are important!):

- Using the **expr** utility: **expr** is a standard but somewhat deprecated program. The syntax is as follows: `expr 8 + 8; echo $(expr 8 + 8)`
- Using the `$((...))` syntax: This is the built-in shell format. The syntax is as follows: `echo $((x+1))`
- Using the built-in shell command `let`. The syntax is as follows: `let x=( 1 + 2 ); echo $x`

In modern shell scripts the use of **expr** is better replaced with `var=$((...))`.

```
# 默认将变量作为文本字符串
MYVAR=1
MYVAR=$MYVAR+1
echo $MYVAR # 输出: 1+1

# 需要用特定的方法进行算数运算
MYVAR=`expr $MYVAR + 1`
echo $MYVAR # 输出: 2
```

```
var=1 # var: 1
# 第一种方法: $(( ))
var=$(($var+1)) # var: 2
((var++)) # var: 3
# 第二种方法: $[ ]
var=$[$var+1] # var: 4
var=$[var+1] # var: 5
# 第三种方法: let
let var+=1 # var: 6
let var=var+1 # var: 7
let var=$var+1 # var: 8
# 第四种方法: expr (注意加号两边的空格, 否则还是按照字符串的方式赋值)
var=$(expr $var + 1) # var: 9, 不建议使用
var=`expr $var + 1` # var: 10, 强烈不建议使用
```

示例1

```
[root@pc05 ~]# expr 45 + 21
66
[root@pc05 ~]# expr 45 - 21
24
[root@pc05 ~]# expr 45 \* 21
945
```

示例2

```
[root@pc05 ~]# expr 45 / 21
2
[root@pc05 ~]# expr 45 % 21
3
[root@pc05 ~]# X=45; Y=21; expr $X - $Y
24
```

通过$变量名引用

```
#!/bin/sh
_int=3
var=`expr $_int + 10`
echo $var
var=`expr $var - 1`
echo $var
var=`expr $var / 3`
echo $var
```

Himi — himi — 53×14

```
Last login: Sat Apr 14 15:28:18 on ttys000
mac:~ Himi$ /Users/Himi/Desktop/himi ; exit;
13
12
4
```

- shell 提供三种逻辑操作完成此功能。
  - -a ：逻辑与，操作符两边均为真，结果为真，否则为假。
  - -o ：逻辑或，操作符两边一边为真，结果为真，否则为假。
  - ！ ：逻辑否，条件为假，结果为真。
- 其优先级为："！"最高，"-a"次之，"-o"最低。

例：if [-x file1 –a –x file2]

if [! –d file1]

http://blog.csdn.net/caoyicheng1

```
# 逻辑运算符and (&&, -a)
if [ condition1 -a condition2 ]
# if [ condition1 ] && [ condition2 ]
then
  some action
fi

# 等价于
if [ condition1 ]
then
  if [ condition2 ]
  then
    some action
  fi
fi
```

```
# 逻辑运算符or (||, -o)
if [ condition1 -o condition2 ]
# if [ condition1 ] || [ condition2 ]
then
  some action
fi

# 等价于
if [ condition1 ]
then
  some action
elif [ condition2 ]
then
  the same action
fi
```

Boolean expressions evaluate to either **TRUE** or **FALSE**, and results are obtained using the various Boolean operators.

Operator	Operation	Meaning
&&	AND	The action will be performed only if both the conditions evaluate to true.
||	OR	The action will be performed if any one of the conditions evaluate to true.
!	NOT	The action will be performed only if the condition evaluates to false.

Note that if you have multiple conditions strung together with the `&&` operator processing stops as soon as a condition evaluates to false. For example if you have `A && B && C` and A is true but B is false, C will never be executed.

Likewise if you are using the `||` operator, processing stops as soon as anything is true. For example if you have `A || B || C` and A is false and B is true, you will also never execute C.

- 在 [ ] 表达式中，不直接支持 >、< 运算符，需要加转义字符，表示字符串大小比较，以 ASCII 码位置进行比较。
- 在 [ ] 表达式中，逻辑运算符 ||、&& 需要分别用-o 和-a 表示。
- [[ ]] 运算符只是 [ ] 运算符的扩充。能够支持 >、> 符号运算，不再需要转义符，它还是以字符串比较大小。
- [[ ]] 运算符里面支持逻辑运算符 || 和 &&。

In modern scripts you may see doubled brackets as in `[[ -f /etc/passwd ]]`. This is not an error. It is never wrong to do so and it avoids some subtle problems such as referring to an empty environment variable without surrounding it in double quotes.

逻辑卷标	表示意思
1.	**关于档案与目录的侦测逻辑卷标！**
-f	常用！侦测『档案』是否存在 eg: if [-f filename]
-d	常用！侦测『目录』是否存在
-b	侦测是否为一个『 block 档案』
-c	侦测是否为一个『 character 档案』
-S	侦测是否为一个『 socket 标签档案』
-L	侦测是否为一个『 symbolic link 的档案』
-e	侦测『某个东西』是否存在！
2.	**关于程序的逻辑卷标！**
-G	侦测是否由 GID 所执行的程序所拥有
-O	侦测是否由 UID 所执行的程序所拥有
-p	侦测是否为程序间传送信息的 name pipe 或是 FIFO（老实说，这个不太懂！）

3.	**关于档案的属性侦测！**
-r	侦测是否为可读的属性
-w	侦测是否为可以写入的属性
-x	侦测是否为可执行的属性
-s	侦测是否为『非空白档案』
-u	侦测是否具有『 SUID 』的属性
-g	侦测是否具有『 SGID 』的属性
-k	侦测是否具有『 sticky bit 』的属性
4.	**两个档案之间的判断与比较；** 例如[test file1 -nt file2]
-nt	第一个档案比第二个档案新
-ot	第一个档案比第二个档案旧
-ef	第一个档案与第二个档案为同一个档案（ link 之类的档案）
5.	逻辑的『和(and)』『或(or)』
&&	逻辑的 AND 的意思
||	逻辑的 OR 的意思

流程控制 | 条件流程控制 | 运算符 | 汇总（3/3）

运算符号	代表意义
=	等于 应用于：整型或字符串比较 如果在[] 中，只能是字符串
!=	不等于 应用于：整型或字符串比较 如果在[] 中，只能是字符串
<	小于 应用于：整型比较 在[] 中，不能使用 表示字符串
>	大于 应用于：整型比较 在[] 中，不能使用 表示字符串
-eq	等于 应用于：整型比较
-ne	不等于 应用于：整型比较
-lt	小于 应用于：整型比较
-gt	大于 应用于：整型比较
-le	小于或等于 应用于：整型比较
-ge	大于或等于 应用于：整型比较
-a	双方都成立（and）逻辑表达式 –a 逻辑表达式
-o	单方成立（or）逻辑表达式 –o 逻辑表达式
-z	空字符串
-n	非空字符串

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

作用

select 表达式是 bash 的一种扩展应用，擅长于交互式场合。它把关键字中的每一项做成表单，用户可以从中进行选择。

1. 自动用 1、2、3、4 列出菜单（没有 echo 指令，自动显示菜单）
2. 自动 read 输入选择（没有 read 指令，自动输入）
3. 赋值给变量（没有赋值指令，自动输入数字后，赋值字符串给变量）

语法

```
select var in WORDS
do
  command(s)
done
# ... now $var can be used ...
```

作用

select 表达式是 bash 的一种扩展应用，擅长于交互式场合。它把关键字中的每一项做成表单，用户可以从中进行选择。

1. 自动用 1、2、3、4 列出菜单（没有 echo 指令，自动显示菜单）
2. 自动 read 输入选择（没有 read 指令，自动输入）
3. 赋值给变量（没有赋值指令，自动输入数字后，赋值字符串给变量）

语法

```
select var in WORDS
do
  command(s)
done
# ... now $var can be used ...
```

```
#!/bin/bash
echo "What is your favourite OS?"
select var in "Linux" "Free BSD" "Other"
do
  break
done
echo "You have selected $var"

# 运行结果
What is your favourite OS?
1) Linux
2) Free BSD
3) Other
#? 1
You have selected Linux
```

- select 本身就是一个循环，break 是当选择后就跳出循环
- select 输入选择是数字，但变量值却是字符串
- 一般与 case 语句配合使用

The `case` statement is used in scenarios where the actual value of a variable can lead to different execution paths. `case` statements are often used to handle command-line options.

Case Statement
Expression
Case 1
True
Execute comands and exit
False
Case 2
True
Execute comands and exit
False
. . .
True
Execute comands and exit
False
Case n
True
Execute comands and exit
False
Execute default commands and exit

```
1 case expression in
2   pattern1) execute commands;;
3   pattern2) execute commands;;
4   pattern3) execute commands;;
5   pattern4) execute commands;;
6   * )       execute some default commands or
    nothing ;;
7 esac
```

```
1  #!/bin/sh
2  # Prompt user to enter a character
3  echo "Please enter a letter:"
4  read charac
5  case "$charac" in
6      "a"|"A") echo "You have typed a vowel!" ;;
7      "e"|"E") echo "You have typed a vowel!" ;;
8      "i"|"I") echo "You have typed a vowel!" ;;
9      "o"|"O") echo "You have typed a vowel!" ;;
10     "u"|"U") echo "You have typed a vowel!" ;;
11     *)         echo "You have typed a consonant!" ;;
12 esac
13 exit 0
```

流程控制 | 选择流程控制 | case | 逻辑流程

作用

case 表达式可以用来匹配一个给定的字符串，而不是数字。

```
case expression in
  pattern1)
    action1
    ;;
  pattern2)
    action2
    ;;
  *)
    default action
    ;;
esac
```

作用

case 表达式可以用来匹配一个给定的字符串，而不是数字。

```
case expression in
  pattern1)
    action1
    ;;
  pattern2)
    action2
    ;;
  *)
    default action
    ;;
esac
```

pattern

pattern 是正则表达式，可以用下面的字符：

*	任意字符串
?	任意单个字符
[abc]	a、b 或 c 单个字符其中之一
[a-n]	从 a 到 n 的任意一个字符
|	多重选择

流程控制 | 选择流程控制 | case | 实例

```
#!/bin/bash
case "$1" in
  start)
    sleep 3600 &
    ;;
  stop)
    pkill -x "sleep"
    ;;
  *)
    echo "Usage: $0 {start|stop}"
    ;;
esac
```

```
1  #!/bin/bash
2  read -p "请输入一个字符，并按Enter键确认：" KEY
3
4  case "$KEY" in
5    [a-z]|[A-Z])
6      echo "您输入的是一字母。"
7      ;;
8    [0-9])
9      echo "您输入的是一数字。"
10     ;;
11   *)
12     echo "您输入的是一空格、功能键或其他控制字符。"
13     ;;
14 esac
```

```
# smartzip, 自动解压bzip2、gzip和zip类型的压缩文件
#!/bin/bash

ftype="$(file "$1")"
case "$ftype" in
  "$1: Zip archive"*)
    unzip "$1" ;;
  "$1: gzip compressed"*)
    gunzip "$1" ;;
  "$1: bzip2 compressed"*)
    bunzip2 "$1" ;;
  *)
    echo "File $1 can not be uncompressed with smartzip";;
esac
```

```bash
#!/bin/bash

select DRINK in tea cofee water juice appe all none
do
  case $DRINK in
    tea|cofee|water|all)
      echo "Go to canteen"
      ;;
    juice|appe)
      echo "Available at home"
      ;;
    none)
      break
      ;;
    *)
      echo "ERROR: Invalid selection"
      ;;
  esac
done
```

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

By using **looping constructs**, you can execute one or more lines of code repetitively. Usually you do this until a conditional test returns either true or false as is required.

Three type of loops are often used in most programming languages:

- `for`
- `while`
- `until`

All these loops are easily used for repeating a set of statements until the exit condition is true.

The `while` loop repeats a set of statements as long as the control command returns true.

```
while condition is true
do
  Commands for execution
done
```

The set of commands that need to be repeated should be enclosed between `do` and `done`. You can use any command or operator as the condition. Often it is enclosed within square brackets (`[]`).

```
#!/bin/bash
#
echo "Enter the number"
read no
fact=1
i=1
while [ $i -le $no ]
do
        fact=$(($fact * $i))
        i=$(($i + 1))
done
echo "The factorial of $no is $fact"
```

Objective of the script: To display the contents of an existing file.

Steps:

1. Display a message, asking for a *filename*
2. Accept the name of a file
3. Redirect the 'stdin' to a file
4. While a line of text exists, do the following
   5. Display the line being read

```
#! /bin/bash
echo -e "Enter absolute path of the file name you want to read"
read file
exec <$file # redirects stdin to a file
while read line
do
echo $line
done
```

```
while condition
do
  action
done
```

```
#!/bin/bash

read -p "Guess the secret color (red, blue, yellow, purple, or orange):" COLOR

while [ $COLOR != "purple" ]
do
  read -p "Incorrect. Guess again:" COLOR
done

echo "Correct."
```

```
#!/bin/bash

guess_color () {
  read -p "Guess the secret color (red, blue, yellow, purple, or orange):" COLOR
  while [ $COLOR != "purple" ]
  do
    echo "Incorrect. Guess again!"
    guess_color
  done
}

guess_color
echo "Correct."
```

The `until` loop repeats a set of statements as long as the control command is false. Thus it is essentially the opposite of the `while` loop.

```
until condition is false
do
  Commands for execution
done
```

Similar to the while loop, the set of commands that need to be repeated should be enclosed between `do` and `done`. You can use any command or operator as the condition.

```
#Until Loop - Example-1
echo "NUMBER"
mn=1
mx=10
until [ $mn -gt $mx ]
do
        echo "$mn"
        mn=$(($mn + 2))
done
```

Objective of the script: To display the contents of a variable until it meets a condition, using 'do' loop

Steps:

1. Initialize a variable '*number*' to 0
2. Repeat the following steps until number >= 10
   1. Display the *number*
   2. Increase the *number* by an unit

```
#!/bin/bash

number=0
until [ $number -ge 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

```
until condition
do
  action
done
```

```
#!/bin/bash

read -p "Guess the secret color(red, blue,
  yellow, purple, or orange):" COLOR

until [ $COLOR = "purple" ]
do
  read -p "Incorrect. Guess again:" COLOR
done

echo "Correct."
```

The `for` loop operates on each element of a list of items.

```
for variable-name in list
do
  execute one iteration for each item in the list until the list is finished
done
```

In this case, `variable-name` and `list` are substituted by you as appropriate (see examples). As with other looping constructs, the statements that are repeated should be enclosed by `do` and `done`.

```
#!/bin/sh
#
sum=0
for i in 1 2 3 4
do
  sum=$(($sum+$i))
done
echo "The sum of $i numbers is: $sum"
```

流程控制 | 迭代流程控制 | for | Example

Objective of the script: To display the type of files, based on the extension

Steps:

1. Repeat the following steps until the output of 'ls' command exists
   2. Retrieve the extension of the first file read from the ls command
   3. If the ext is 'c' then display a message "C Source File'
   4. If the ext is 'o' then display a message "Object File'
   5. If the ext is 'sh' then display a message "Shell script'
   6. If the ext is 'txt' then display a message "Text File'
   7. If the ext is "any other" value, then display a message "Unknown file"

```bash
#!/bin/bash
for filename in $(ls)
do
        # Take extension available in a filename
        ext=${filename##*\.}
        case "$ext" in
        c) echo "$filename : C source file"
           ;;
        o) echo "$filename : Object file"
           ;;
        sh) echo "$filename : Shell script"
            ;;
        txt) echo "$filename : Text file"
             ;;
        *) echo "$filename : Unknown file type/Not processed"
           ;;
esac
done
```

```
for VAR in LIST
do
  action
done
```

```
1 #!/bin/bash
2
3 # LIST字符串只要用空格字符进行分隔，每次for…in读取
    时，就会按顺序将读值赋给前面的变量
4 for DAY in Sunday Monday Tuesday Wednesday
    Thursday Friday Saturday
5 do
6   echo "The day is: $DAY"
7 done
```

```
#!/bin/bash

# list a content summary of a number of RPM packages
# USAGE: showrpm rpmfile1 rpmfile2 ...
# EXAMPLE: showrpm /cdrom/RedHat/RPMS/*.rpm
for rpmpackage in "$@"; do
  if [ -r "$rpmpackage" ]; then
    echo "======== $rpmpackage ========="
    rpm -qi -p $rpmpackage
  else
    echo "ERROR: cannot read file $rpmpackage"
  fi
done
```

流程控制 | 迭代流程控制 | for | 实例

```
[root@pc05 ~]# cat test1.sh
#!/bin/bash
for i in "1st." "2nd." "3rd."
do
    echo $i
done
[root@pc05 ~]# ./test1.sh
1st.
2nd.
3rd.
```

```
[root@pc05 ~]# cat /etc/host.conf
order hosts,bind
[root@pc05 ~]# cat test2.sh
#!/bin/bash
for i in $(cat /etc/host.conf)
do
    echo $i
done
[root@pc05 ~]# ./test2.sh
order
hosts,bind
```

```
[root@pc05 ~]# cat uad.sh
#!/bin/bash
for i in $(cat /root/users.txt)
do
        useradd $i
        echo "123456" | passwd --stdin $i
        chage -d 0 $i
done
```

```
for ((赋值; 条件; 运算语句))
do
  actions
done
```

```
#!/bin/bash

for ((i=1; i<=10; i++))
do
  echo $i
done
```

```
for ((赋值; 条件; 运算语句))
do
  actions
done
```

```
#!/bin/bash

for ((i=1; i<=10; i++))
do
  echo $i
done
```

跳出循环

- break：跳出整个循环
- continue：跳过本次循环（的余下部分），直接进入下次循环

结束

- break：结束循环（loop）
- return：结束函数（function）
- exit：结束脚本（script）或者 shell

参数控制

- shift：参数左移。每执行一次，参数序列顺次左移一个位置，$#的值减 1。用于分别处理每个参数，移出去的参数不再可用。

跳出循环

- break：跳出整个循环
- continue：跳过本次循环（的余下部分），直接进入下次循环

结束

- break：结束循环（loop）
- return：结束函数（function）
- exit：结束脚本（script）或者 shell

参数控制

- shift：参数左移。每执行一次，参数序列顺次左移一个位置，$#的值减 1。用于分别处理每个参数，移出去的参数不再可用。

跳出循环

- break：跳出整个循环
- continue：跳过本次循环（的余下部分），直接进入下次循环

结束

- break：结束循环（loop）
- return：结束函数（function）
- exit：结束脚本（script）或者 shell

参数控制

- shift：参数左移。每执行一次，参数序列顺次左移一个位置，$#的值减 1。用于分别处理每个参数，移出去的参数不再可用。

流程控制 | 迭代流程控制 | 补充 | shift

```
#!/bin/sh

if [ $# -le 0 ]
then
  echo "Not enough parameters"
  exit 0
fi

sum=0
while [ $# -gt 0 ]
do
  sum=$(($sum+$1))
  shift
done
echo $sum
```

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

高级概念 | 输入输出重定向

Most operating systems accept input from the keyboard and display the output on the terminal. However, in shell scripting you can send the output to a file. The process of diverting the output to a file is called output **redirection**.

- The > character is used to write output to a file.
- Two > characters (>>) will append output to a file if it exists, and act just like > if the file does not already exist.

Just as the output can be redirected **to** a file, the input of a command can be read **from** a file. The process of reading input from a file is called input redirection and uses the < character.

编号	符号	含义	默认
0	STDIN	标准输入	终端的键盘
1	STDOUT	标准输出	终端的显示器
2	STDERR	标准错误输出	终端的显示器

操作符	含义
< FILE	等价于 0< FILE，把 STDIN 重定向到文件
> FILE	等价于 1> FILE，把 STDOUT 重定向到文件
2> FILE	把 STDERR 重定向到文件
>> FILE	把 STDOUT 添加到文件末尾
&> FILE	对 STDOUT 和 STDERR 同时进行重定向
2>&1	把 STDERR 重定向到 STDOUT
1>&2	把 STDOUT 重定向到 STDERR
|	把一个程序的输出发送给另一个程序
<<delimiter	使用 here 文档中的内容重定向 STDIN

编号	符号	含义	默认
0	STDIN	标准输入	终端的键盘
1	STDOUT	标准输出	终端的显示器
2	STDERR	标准错误输出	终端的显示器

操作符	含义
< FILE	等价于 0< FILE，把 STDIN 重定向到文件
> FILE	等价于 1> FILE，把 STDOUT 重定向到文件
2> FILE	把 STDERR 重定向到文件
>> FILE	把 STDOUT 添加到文件末尾
&> FILE	对 STDOUT 和 STDERR 同时进行重定向
2>&1	把 STDERR 重定向到 STDOUT
1>&2	把 STDOUT 重定向到 STDERR
|	把一个程序的输出发送给另一个程序
<<delimiter	使用 here 文档中的内容重定向 STDIN

高级概念 | 输入输出重定向

名称	英文缩写	内容	默认绑定位置	文件路径	Shell中代号
标准输入流	`stdin`	程序读取的用户输入	键盘输入	`/dev/stdin`	`0`
标准输出流	`stdout`	程序的打印的正常信息	终端(`terminal`), 即显示器	`/dev/stdin`	`1`
标准错误流	`stderr`	程序的错误信息	终端(`terminal`) , , 即显示器	`/dev/stderr`	`2`

高级概念 | 输入输出重定向 | 重定向方式一览表

操作	含义
cmd > file	把 stdout 重定向到 file
cmd >> file	把 stdout 追加到 file
cmd 2> file	把 stderr 重定向到 file
cmd 2>> file	把 stderr 追加到 file
cmd &> file	把 stdout 和 stderr 重定向到 file
cmd > file 2>&1	把 stdout 和 stderr 重定向到 file
cmd >> file 2>&1	把 stdout 和 stderr 追加到 file
cmd file2 cmd	cmd 以 file 作为 stdin，以 file2 作为 stdout
cat <>file	以读写的方式打开 file
cmd < file cmd	cmd 命令以 file 文件作为 stdin
cmd << delimiter Here document	从 stdin 中读入，直至遇到 delimiter 分界符

高级概念 | 输入输出重定向

类别	操作符	说明
输入重定向	<	输入重定向是将命令中接收输入的途径由默认的键盘更改（重定向）为指定的文件
输出重定向	>	将命令的执行结果重定向输出到指定的文件中，命令进行输出重定向后执行结果将不显示在屏幕上
	>>	将命令执行的结果重定向并追加到指定文件的末尾保存
错误重定向	2>	清空指定文件的内容，并保存标准错误输出的内容到指定文件中
	2>>	向指定文件中追加命令的错误输出，而不覆盖文件中的原有内容
输出与错误组合重定向	&>	将标准输出与错误输出的内容全部重定向到指定文件

http://blog.csdn.net/caoyicheng1

- 管道操作符 |
- "|"符用于连接左右两个命令，将"|"左边的命令执行结果（输出）作为"|"右边命令的输入

  cmd1 | cmd2
- 在同一条命令中可以使用多个"|"符连接多条命令 cmd1 | cmd2 | ... | cmdn

```
tr a-z A-Z <<END_TEXT
one two three
END_TEXT
#ONE TWO THREE

# 默认会进行变量替换和命令替换
cat << EOF
Working dir $PWD
EOF
#Working dir /home/user

# 通过使用引号包裹标识符来禁用变量替换和命令替换
cat << "EOF"
Working dir $PWD
EOF
#Working dir $PWD
```

高级概念 | 输入输出重定向 | 将变量中内容作为程序输入

```
EXP="1+1"
bc -q <<< "${EXP}"
```

```
for LINE in $(cat test.txt)
do
  echo $LINE
done

cat test.txt | while read LINE
do
  echo $LINE
done

While read LINE
do
  echo $LINE
done  < test.txt
```

```
1  LINE_NO=0
2  cat test.txt |
3  while read LINE
4  do
5     (( LINE_NO++ ))
6     echo "${LINE_NO} ${LINE}"
7  done
8
9  echo "${LINE_NO}"
10 # 输出0，原因是管道中的while语句是执行在子进程中的，
     不会改变父进程中LINE_NO变量的值
```

```
# 将当前目录及子目录所有后缀名为.log的文件删除
find . -type f -name *.log | xargs rm

# 将当前目录及子目录中所有后缀名为.log的文件移动到/data/log中
find . -type f -name *.log | xargs -i mv {} /data/log
```

```
# xargs可以从标准输入读取内容，以之构建并执行另一个命令行
# xargs直接接命令名称，则将从标准输入读取的所有内容合并为一行构建命令行并执行
# xargs加上-i参数，则会每读取一行就构建并执行一个命令行，构建命令行时会将{}替换为该行的内容
cat test.txt
#a
#b
#c
cat test.txt | xargs echo rm
#rm a b c
cat test.txt | xargs -i echo rm {}
#rm a
#rm b
#rm c
```

教学提纲

1 引言
2 编程起步
- 脚本结构与运行
- 调用 shell
- 特殊字符
- 变量
- 从键盘读取输入
- 特殊变量
- 退出状态

3 流程控制
- 条件流程控制
- 选择流程控制
- 迭代流程控制

4 高级概念
- 输入输出重定向
- 并行
- 命令替换
- 通配

5 shell 函数
- 声明与调用
- 返回值
- 嵌套和递归
- 作用域
- 文件处理
- 数组
- 关联数组

6 脚本调试
7 回顾与总结
- 总结
- 思考题

```
#!/bin/bash

# 输出1到10: 需要10秒
for i in $(seq 1 10)
do
  echo $i; sleep 2
done

# 输出1到10: 只需要2秒
for i in $(seq 1 10)
do
  {
    echo $i; sleep 2
  } &
done
wait
```

高级概念 | 并行 | 控制并行数量

```bash
#!/bin/bash

# 输出1到100：每次输出10个数，每次输出后间隔2秒继续下一轮的输出；只需要20秒
n=10
for  ((i=0; $n*i<100; i++))
do
  for ((j=$n*i+1; j<$n*i+($n+1); j++))
  do
    {
      echo $j
      sleep 2
    } &
  done
  wait
done
```

```
#!/bin/bash

while read LINE
do
  {
    echo $LINE
  } &
done < test.txt
wait
```

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

高级概念 | 命令替换

At times, you may need to substitute the result of a command as a portion of another command. It can be done in two ways:

- By enclosing the inner command with backticks (`)
- By enclosing the inner command in `$( )`

No matter the method, the innermost command will be executed in a newly launched shell environment, and the standard output of the shell will be inserted where the command substitution was done.

Virtually any command can be executed this way. Both of these methods enable command substitution; however, the `$( )` method allows command nesting. New scripts should always use this more modern method.

```
L=`wc -l textfile` # 使用反引号``
L=$(wc -l textfile) # 使用$()
echo "Next year will be 20$(expr $(date +%y) + 1)" # 嵌套
echo "Next year will be 20$(($(date +%y)+1))"
```

```
[test3@CentOS ~]$ uname -r
2.6.32-431.20.5.el6.x86_64
[test3@CentOS ~]$ echo cd /lib/modules/`uname -r`/
cd /lib/modules/2.6.32-431.20.5.el6.x86_64/
[test3@CentOS ~]$ echo cd /lib/modules/$(uname -r)/
cd /lib/modules/2.6.32-431.20.5.el6.x86_64/
[test3@CentOS ~]$ pwd
/home/test3
[test3@CentOS ~]$ $(echo cd /lib/modules/$(uname -r)/)
[test3@CentOS 2.6.32-431.20.5.el6.x86_64]$ pwd
/lib/modules/2.6.32-431.20.5.el6.x86_64
[test3@CentOS 2.6.32-431.20.5.el6.x86_64]$
```

```
# 如果命令执行结果有多行内容，存入变量并打印时换行符会丢失:
cat test.txt
#a
#b
#c
CONTENT=`cat test.txt`
echo ${CONTENT}
#a b c

#若需要保留换行符，则在打印时必须加上"":
CONTENT=`cat test.txt`
echo "${CONTENT}"
#a
#b
#c
```

```
# ``: 命令替换
ls `pwd`
# $( ): 命令替换
ls $(pwd)

# $(( )): 算数运算
var=1 #var: 1
var=$(($var+1)) #var: 2
# $[ ]: 算术运算
var=$[$var+1] #var: 3

# ${ }: 变量保护（确切说是变量替换）
num=1
echo "The ${num}st." #The 1st.
echo "The $numst." #The .
```

高级概念 | Creating Temporary Files and Directories

Consider a situation where you want to retrieve 100 records from a file with 10,000 records. You will need a place to store the extracted information, perhaps in a **temporary file**, while you do further processing on it.

Temporary files (and directories) are meant to store data for a short time. Usually one arranges it so that these files disappear when the program using them terminates. While you can also use **touch** to create a temporary file, this may make it easy for hackers to gain access to your data.

The best practice is to create random and unpredictable filenames for temporary storage. One way to do this is with the **mktemp** utility.

Command	Usage
`TEMP=$(mktemp /tmp/tempfile.XXXXXXXX)`	To create a temporary file
`TEMPDIR=$(mktemp -d /tmp/tempdir.XXXXXXXX)`	To create a temporary directory

The `XXXXXXXX` is replaced by the **mktemp** utility with random characters to ensure the name of the temporary file cannot be easily predicted and is only known within your program.

高级概念 | Discarding Output with /dev/null

Certain commands like **find** will produce voluminous amounts of output which can overwhelm the console. To avoid this, we can redirect the large output to a special file (a device node) called **/dev/null**. This file is also called the **bit bucket** or **black hole**.

It discards all data that gets written to it and never returns a failure on write operations. Using the proper redirection operators, it can make the output disappear from commands that would normally generate output to **stdout** and/or **stderr**.

```
find / > /dev/null
```

In the above command, the entire standard output stream is ignored, but any errors will still appear on the console.

高级概念 | Random Numbers and Data

It is often useful to generate random numbers and other random data when performing tasks such as:

- Performing security-related tasks.
- Reinitializing storage devices.
- Erasing and/or obscuring existing data.
- Generating meaningless data to be used for tests.

Such random numbers can be generated by using the `$RANDOM` environment variable (`echo $RANDOM`), which is derived from the Linux kernel's built-in random number generator, or by the **OpenSSL** library function, which uses the FIPS140 algorithm to generate random numbers for encryption.

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

高级概念 | 通配

疑问

如果你的当前目录中有 1.txt 2.txt 3.txt 三个文件，那么当你执行 ls *.txt 这条命令，shell 究竟为你做了什么？

解答

其实 shell 会先读取当前目录，然后按 *.txt 的通配条件过滤得到 1.txt 2.txt 3.txt 这个文件列表，然后将这个列表作为参数传给 ls，即 ls *.txt 相当于 ls 1.txt 2.txt 3.txt，ls 命令本身并不会得到 *.txt 这样的参数。

补充

仅当目录中没有符合 *.txt 通配的文件，shell 才会将 *.txt 这个字符串当作参数直接传给 ls 命令。
如果需要列出当前目录中所有的 txt 文件，使用 echo *.txt 也同样可以达到目的。

高级概念 | 通配

疑问

如果你的当前目录中有 1.txt 2.txt 3.txt 三个文件，那么当你执行 ls *.txt 这条命令，shell 究竟为你做了什么？

解答

其实 shell 会先读取当前目录，然后按 *.txt 的通配条件过滤得到 1.txt 2.txt 3.txt 这个文件列表，然后将这个列表作为参数传给 ls，即 ls *.txt 相当于 ls 1.txt 2.txt 3.txt，ls 命令本身并不会得到 *.txt 这样的参数。

补充

仅当目录中没有符合 *.txt 通配的文件，shell 才会将 *.txt 这个字符串当作参数直接传给 ls 命令。
如果需要列出当前目录中所有的 txt 文件，使用 echo *.txt 也同样可以达到目的。

高级概念 | 通配

疑问

如果你的当前目录中有 1.txt 2.txt 3.txt 三个文件，那么当你执行 ls *.txt 这条命令，shell 究竟为你做了什么？

解答

其实 shell 会先读取当前目录，然后按 *.txt 的通配条件过滤得到 1.txt 2.txt 3.txt 这个文件列表，然后将这个列表作为参数传给 ls，即 ls *.txt 相当于 ls 1.txt 2.txt 3.txt，ls 命令本身并不会得到 *.txt 这样的参数。

补充

仅当目录中没有符合 *.txt 通配的文件，shell 才会将 *.txt 这个字符串当作参数直接传给 ls 命令。
如果需要列出当前目录中所有的 txt 文件，使用 echo *.txt 也同样可以达到目的。

高级概念 | 通配 | 通配符一览表

字符	含义	实例
*	匹配 0 或多个字符	`a*b` a与b之间可以有任意长度的任意字符，也可以一个也没有，如aabcb, axyzb, a012b, ab。
?	匹配任意一个字符	`a?b` a与b之间必须也只能有一个字符，可以是任意字符，如aab, abb, acb, a0b。
`[list]`	匹配 list 中的任意单一字符	`a[xyz]b` a与b之间必须也只能有一个字符，但只能是 x 或 y 或 z，如: axb, ayb, azb。
`[!list]` `[^list]`	匹配 除list 中的任意单一字符	`a[!0-9]b` a与b之间必须也只能有一个字符，但不能是阿拉伯数字，如axb, aab, a-b。
`[c1-c2]`	匹配 c1-c2 中的任意单一字符 如：[0-9] [a-z]	`a[0-9]b` 0与9之间必须也只能有一个字符 如a0b, a1b... a9b。
`{string1,string2,...}`	枚举sring1或string2(或更多)其一字符串	`a{abc,xyz,123}b` 展开成 `aabcb axyzb a123b`
`{c1..c2}` `{n1..n2}`	枚举c1-c2中所有字符或n1-n2中所有数字	`{a..f}` 展开成 `a b c d e f` `a{1..5}` 展开成 `a1 a2 a3 a4 a5`

```
# *用于通配文件名或目录名中某一部分为任意内容
rm *.log                # 删除当前目录中所有后缀名为.log的文件
rm */log/*.log          # 删除所有二级log目录中后缀名为.log的文件

# [ ]用于通配文件名或目录名中某个字符为限定范围内或限定范围外的值
rm Program[1-9]*.log    # 删除当前目录中以Program跟着一个1到9的数字开头，并以.log为后缀名的文件
du -sh /[^udp]*         # 对根目录中所有不以u、d、p开头的文件夹求取总大小

# ?用于通配文件名中某处一个任意值的字符
rm L????.txt            # 通配一个文件名以L开头，后面跟着4个字符，并以.txt结尾的文件：如LAB01.txt
```

```
# { }用于通配在它枚举范围内的值。它是直接展开的，可以将它用于批量创建目录或文件，也可以用于生成序列

# 批量生成目录
ls; mkdir dir{0..2}{0..2}
ls #dir00  dir01  dir02  dir10  dir11  dir12  dir20  dir21  dir22

# 生成序列；{ }生成的序列常用于for循环
for ip in 192.168.234.{1..255}
do #查找192.168.234.1~192.168.234.255整个网段能ping通的所有ip
  ping ${ip} -w 1 &> /dev/null && echo ${ip} is Alive
done
```

高级概念 | 通配 | 补充

- *、?、[ ] 的通配都会按读取目录并过滤的方式展开通配项目
- { } 则不会有读取目录的过程，它是通过枚举所有符合条件的通配项直接展开的
- * 通配的结果与目录中存在哪些文件有关系，而 { } 的通配结果与目录中存在哪些文件无关。若用 { } 进行通配，则有可能将不存在的文件路径作为命令行参数传给程序。

高级概念 | 通配 | 补充

```
ls
#1.txt  2.txt  3.txt

echo *.txt
#1.txt 2.txt 3.txt

echo {1..5}.txt
#1.txt 2.txt 3.txt 4.txt 5.txt

ls {1..5}.txt
#ls: cannot access 4.txt: No such file or directory
#ls: cannot access 5.txt: No such file or directory
#1.txt  2.txt  3.txt
```

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

shell 函数 | Introduction

A **function** is a code block that implements a set of operations. Functions are useful for executing procedures multiple times perhaps with varying input variables. Functions are also often called **subroutines**. Using functions in scripts requires two steps:

- Declaring a function
- Calling a function

The function can be as long as desired and have many statements. Once defined, the function can be called later as many times as necessary.

```
#!bin/bash
display(){
        echo "This is the message from the function"
        echo "The parameter passed from calling process is" $1
}
display "Bob"
display "Stuart"
display "Rambo"
display "Bond"
```

shell 函数 | 简介

函数

函数使您可以把一个脚本程序的整体功能分割成比较小的、逻辑的子功能，在需要的时候可以调用它们执行各自单独的任务。使用函数执行重复性的任务是实现代码重用的一种很好的方式。

函数的优势

- 不必重复编写完成同样功能的代码
- 使脚本更加容易维护
- 使代码保持良好的结构

shell 函数 | 简介

函数

函数使您可以把一个脚本程序的整体功能分割成比较小的、逻辑的子功能，在需要的时候可以调用它们执行各自单独的任务。使用函数执行重复性的任务是实现代码重用的一种很好的方式。

函数的优势

- 不必重复编写完成同样功能的代码
- 使脚本更加容易维护
- 使代码保持良好的结构

变量

- 变量均为全局变量，没有局部变量
- 使用 local 定义局部变量

参数

- 调用函数时，可以传递参数
- 在函数中用 $1、$2、……来引用参数

注意

- $0：在脚本中，$0 是脚本的名字；在函数中，$0 还是脚本的名字，而非函数的名字
- $#：在脚本和函数中，分别是脚本和函数读入的参数个数

变量

- 变量均为全局变量，没有局部变量
- 使用 local 定义局部变量

参数

- 调用函数时，可以传递参数
- 在函数中用 $1、$2、……来引用参数

注意

- $0：在脚本中，$0 是脚本的名字；在函数中，$0 还是脚本的名字，而非函数的名字
- $#：在脚本和函数中，分别是脚本和函数读入的参数个数

变量

- 变量均为全局变量，没有局部变量
- 使用 local 定义局部变量

参数

- 调用函数时，可以传递参数
- 在函数中用 $1、$2、……来引用参数

注意

- $0：在脚本中，$0 是脚本的名字；在函数中，$0 还是脚本的名字，而非函数的名字
- $#：在脚本和函数中，分别是脚本和函数读入的参数个数

```
#!/bin/bash
# test with: sVSf.sh s1 s2

echo "\$0 in script: $0"      # sVSf.sh
echo "\$1 in script: $1"      # s1
echo "\$# in script: $#"      # 2
echo "\$@ in script: $@"      # s1 s2

repeat () {
  echo "\$0 in function: $0" # sVSf.sh
  echo "\$1 in function: $1" # f1
  echo "\$# in function: $#" # 3
  echo "\$@ in function: $@" # f1 f2 f3
}

repeat f1 f2 f3
```

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

```
# 函数声明
name () {
  commands
}

# 函数调用，不带()
name
name parameter(s)
```

shell 函数 | 声明与调用

```
1 #!/bin/bash
2
3 # A simple function
4 repeat () {
5   echo "I don't know $1 $2"
6 }
7
8 repeat Your Name
9 # I don't know Your Name
```

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

默认

函数的返回值默认是最后一条语句的状态。

显式

```
return code
return 0
return 1
```

默认

函数的返回值默认是最后一条语句的状态。

显式

```
return code
return 0
return 1
```

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

函数嵌套

在一个函数中调用包含在另一个函数中的功能，这种情况叫做函数嵌套。

递归函数

函数可以调用自己，就像调用其他函数一样。这种调用自己的函数称为递归函数。

函数嵌套

在一个函数中调用包含在另一个函数中的功能，这种情况叫做函数嵌套。

递归函数

函数可以调用自己，就像调用其他函数一样。这种调用自己的函数称为递归函数。

shell 函数 | 嵌套和递归

```
#!/bin/bash

number_one () {
  echo "This is the first function speaking ..."
  number_two
}

number_two () {
  echo "This is now the second function speaking..."
}

number_one
# This is the first function speaking...
# This is now the second function speaking...
```

shell 函数 | 嵌套和递归 | Fork 炸弹（切勿尝试！）

```
# 经典的只有13个字符的Linux Fork炸弹
:(){ :|:& };:
# 格式化后
:()
{
:|:&
}
;
:
# 更好理解的格式
bomb()
{
bomb|bomb&
}
;
bomb
```

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

作用域

- 全局作用域：在脚本的任何地方都可以访问变量
- 局部作用域：只能在声明变量的作用域内访问它们

定义局部变量

```
# 使用local关键字定义局部变量
local localVariable=1
```

shell 函数 | 作用域

```
#!/bin/bash

# Dealing with local and global variables

scope () {
  local localVar=1
  globalVar=2
  echo "localVar in function: $localVar"
  echo "globalVar in function: $globalVar"
}

scope

echo "localVar outside function: $localVar"
echo "globalVar outside function: $globalVar"
```

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

测试	助记	功能
-d	Directory	文件存在并且是目录
-e	Exist	指定的文件存在
-f	File	文在存在并且是普通文件
-G	Group	执行命令的用户属于文件所属组的成员
-nt	Newer Than	file1 -nt file2, 前一个文件比后一个文件新
-ot	Older Than	file1 -ot file2, 前一个文件比后一个文件老
-O	Owner	执行命令的用户是文件的所有者
-r	Read	执行命令的用户拥有对文件的读取权限
-s	Size	文在存在并且不为空
-w	Write	执行命令的用户拥有对文件的写入权限
-x	eXecute	执行命令的用户拥有对文件的执行权限

```
if [ -r File -a -x File ]
# if [ -r File ] && [ -x File ]
then
  echo "File exists, is readable, and executable!"
fi

if [ -r File -o -w File ]
# if [ -r File ] || [ -w File ]
then
  echo "File exists and is readable or writable!"
fi
```

```
#!/bin/bash

tmpFile=/tmp/sigtrap$$
cat > $tmpFile

function removeTemp () {
  if [ -f "$tmpFile" ]
  then
    echo "Soring out the temp file..."
    rm -f "$tmpFile"
  fi
}

trp removeTemp 1 2
exit 0
```

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

```
# 第一种方法: array[index]=value
day[0]=Sunday

# 第二种方法: array2=(value1 value2 value3 ...)
day=(Sunday Monday Tuesday Wednesday Thursday
    Friday Saturday)

# 第三种方法: array3=([0]=value1 [13]=value2
   [7]=value3)
day=([0]=Sunday [6]=Saturday [1]=Monday)
```

```
# 获得数组中某个特定索引位置的数据: ${array[index]}
# 索引从0开始
firstDay=${day[1]}
echo "$firstDay"

# 获得数组中的所有数据: arrayElements=${array[@]}
allDay=${day[@]}
#allDay=${day[*]}

# 获得数组包含的元素数目: arrayLength=${#array[@]}
numberDay=${#day[@]}
#numberDay=${#day[*]}

# 获得数组中包含的索引值: arrayIndex=${!array[@]}
indexDay=${!day[@]}
```

```
# 获得数组中某个范围内的数据

# 获得第四个元素（含）之后的所有元素的数据
afterTue=${day[@]:3}

# 获得第四个元素（含）后面两个元素的数据
# 获得第四个和第五个元素的数据
WedThu=${day[@]:3:2}
```

```
# 删除索引位置为1的元素的数据
unset array[1]
unset day[1]

# 删除数组中所有元素的数据
unset array[@]
unset day[@]
```

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

```
# 关联数组需要bash 4.0以上版本才支持，选用需慎重
# 查看bash版本用bash --version
# 关联数组必须用declare -A显示声明类型，否则数值会出错

# 声明关联数组（必须有此句）
declare -A a

# 初始化关联数组
a=(["apple"]="a1" ["banana"]="b2" ["carrot"]="c3")
```

```
# 获取元素个数
echo ${#a[*]}
# 获取元素值
echo ${a["carrot"]}
# 插入或修改元素
a["durian"]="d4"
# 获取所有的key
echo ${!a[*]}
# 删除元素
unset a["banana"]
# 清空数组
unset a
```

```
# 遍历数组(仅value)
for i in ${a[*]}
do
  echo ${i}
done

# 遍历数组(key和value)
for key in ${!a[*]}
do
  echo "${key} ==> ${a[${key}]}"
done
```

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

脚本调试 | Introduction

While working with scripts and commands, you may run into errors. These may be due to an error in the script, such as incorrect syntax, or other ingredients such as a missing file or insufficient permission to do an operation. These errors may be reported with a specific error code, but often just yield incorrect or confusing output. So how do you go about identifying and fixing an error?

Debugging helps you troubleshoot and resolve such errors, and is one of the most important tasks a system administrator performs.

脚本调试 | More

Before fixing an error (or bug), it is vital to know its source.

In **bash** shell scripting, you can run a script in **debug mode** by doing `bash -x ./script_file`. Debug mode helps identify the error because:

- It traces and prefixes each command with the + character.
- It displays each command before executing it.
- It can debug only selected parts of a script (if desired) with:

```
set -x      # turns on debugging
...
set +x      # turns off debugging
```

脚本调试 | Redirecting Errors

In UNIX/Linux, all programs that run are given three open file streams when they are started.

File stream	Description	File Descriptor
stdin	Standard Input, by default the keyboard/terminal for programs run from the command line	0
stdout	Standard output, by default the screen for programs run from the command line	1
stderr	Standard error, where output error messages are shown or saved	2

Using redirection we can save the **stdout** and **stderr** output streams to one file or two separate files for later analysis after a program or command is executed.

脚本调试

在命令行中

- `sh -n script.sh`。检查语法，不执行脚本。
- `sh -x script.sh`。执行脚本并显示所有变量的值。

在 shell 脚本中

- `set -x; command; set +x`。脚本局部调试。
- `echo`。打印变量的值、提示信息等来调试脚本。

脚本调试

在命令行中

- `sh -n script.sh`。检查语法，不执行脚本。
- `sh -x script.sh`。执行脚本并显示所有变量的值。

在 shell 脚本中

- `set -x; command; set +x`。脚本局部调试。
- `echo`。打印变量的值、提示信息等来调试脚本。

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

shell 编程 | 总结

知识点

- shell 脚本编程起步：脚本结构、运行、注释，调用 shell，变量、特殊变量，从键盘读取输入，退出状态
- 条件流程控制：if-then，test
- 选择流程控制：select，case
- 迭代流程控制：while，until，for
- shell 编程高级概念：输入输出重定向，命令替换，通配
- shell 函数：声明与调用，返回值，嵌套和递归，作用域，文件处理，数组，关联数组
- shell 脚本的调试

技能

- 掌握 shell 编程的基本语法和调试命令
- 使用 shell 编写应用脚本

教学提纲

1. 引言
2. 编程起步
   - 脚本结构与运行
   - 调用 shell
   - 特殊字符
   - 变量
   - 从键盘读取输入
   - 特殊变量
   - 退出状态
3. 流程控制
   - 条件流程控制
   - 选择流程控制
   - 迭代流程控制
4. 高级概念
   - 输入输出重定向
   - 并行
   - 命令替换
   - 通配
5. shell 函数
   - 声明与调用
   - 返回值
   - 嵌套和递归
   - 作用域
   - 文件处理
   - 数组
   - 关联数组
6. 脚本调试
7. 回顾与总结
   - 总结
   - 思考题

shell 编程 | 思考题

- 如何调用 shell？什么是 shebang 结构？
- 如何给变量赋值、访问变量的值？
- 使用什么命令读取键盘输入？
- 常用的特殊变量有哪些，举例并解释其含义。
- 命令执行成功和不成功的退出状态分别是什么？
- 哪些语句用于条件流程控制，语法是怎样的？
- 哪些语句用于迭代流程控制，语法是怎样的？
- 条件测试的语法有哪两种？
- 常见的文件测试有哪些，举例并解释其含义。
- 字符串和整数值比较的运算符有哪些？
- 如何进行算术运算，语法结构是怎样的？
- 多个条件时使用的逻辑运算符有哪些？
- 如何进行输入输出的重定向、命令替换？
- 如何声明并调用 shell 函数？
- 常见的数组操作有哪些？
- 如何对 shell 脚本进行调试？

下节预告

还记得学习过的 C 语言吗？C 语言中如何处理变量？如何进行条件判断？如何进行迭代循环？

Powered by

TEX LATEX XETEX Beamer