

Linux 操作系统及应用

第五章 — shell 脚本编程

李亦农 唐晓晟

hoplee@bupt.edu.cn txs@bupt.edu.cn

BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS (BUPT)
SCHOOL OF INFORMATION AND COMMUNICATION ENGINEERING



内容简介

- ① 基本概念
- ② shell 变量
- ③ shell 的内部命令及其组合方式
- ④ 流控
- ⑤ 函数
- ⑥ shell 历程



bash 介绍

- bash (Bourne Again SHell) 是自由软件基金会发布的 Bourne Shell 的兼容程序。它包含了许多其他优秀 shell 的良好特性，功能十分全面。很多 Linux 版本都提供 bash。
- 重要的命令行机制：
 - 变量
 - 命令补全
 - 特殊字符
 - 别名
 - 重定向
 - 管道
 - 历史表
 - 命令行编辑



进入 shell I

- 用户登录后系统会自动为用户运行一个 shell，称为**登录 shell**。以后用户提交的所有任务都是这个 shell 的子进程。
- 用户还可以通过运行命令 `/bin/sh` 来产生一个新的 shell。
- `bash` 的提示符为 `$`，超级用户的 shell 提示符为 `#`。
- shell 的主要任务是解释用户的输入，对其中的元字符以及 shell 变量进行代换，然后产生子进程并用指定命令文件的代码和数据去重新生成子进程的上下文；用户的命令执行完毕后 shell 返回一个提示符，将控制权重新交给用户。
- 当 shell 执行用户命令时，它有一些执行方式和环境，可以由用户定制。
- 借助 shell，用户可以：
 - 将多个命令组合成一个新命令；
 - 在后台执行命令；



进入 shell II

- 循环地执行命令；
- 根据不同的条件执行不同的命令；
- 改变命令的输入文件或输出文件。
- ...



shell 脚本

- 所谓**shell 脚本程序**就是将你平时在 shell 命令提示符后输入的若干个 UNIX 命令依次写到一个文本文件中；
- 其中还可以加上一些条件判断、人机交互、参数运用、函数调用等等技巧，以使 shell 脚本更加“**聪明**”地执行；
- 最简单的 shell 脚本只是依次执行事先写好的命令。
- shell 脚本程序设计的内容主要有：变量、内部命令及其组合、流控、函数等；
- **bash**程序的调试：用**bash -x bash-script**命令来查看一个出错的**bash**脚本到底错在什么地方，可帮助程序员进行脚本调试。
- **bash**脚本编程语言长期占据编程语言排行榜前列。



执行 shell 脚本 I

- 假设 shell 程序编制完成后，保存在 script 文件 `tt.sh` 中，其执行方式有以下几种：

```
1 $ sh ./tt.sh
2 $ sh < ./tt.sh
3 $ chmod +x tt.sh; ./tt.sh
4 $ source ./tt.sh # equal to '. ./tt.sh'
5 $ exec ./tt.sh
```

- 第一种方式表示执行一个新的 shell 作为当前 shell 的子 shell，并且将当前目录下的 `tt.sh` 文件作为这个子 shell 的执行参数；
- 第二种方式表示执行一个新的 shell 作为当前 shell 的子 shell，并且将此子 shell 的标准输入重定向到当前目录下的 `tt.sh` 文件；



执行 shell 脚本 II

- 第三种方式是增加文件`tt.sh`的执行权限，直接在当前 shell 中执行脚本文件`tt.sh`（其实是先产生一个子 shell，然后再由这个子 shell 通过`fork/exec`去依次执行脚本中的命令）；
- 第四种方式是在当前 shell 中执行此脚本；
- 第五种方式是在当前 shell 中执行此脚本，并且用此脚本覆盖当前进程；



执行 shell 脚本 III

- 范例:

1.sh

```
1  #!/bin/bash
2  A=B
3  echo "PID of 1.sh before exec/source/fork:$$"
4  export A
5  echo "1.sh:\$A is $A"
6  case $1 in
7      exec)
8          echo "using exec..."
9          exec ./2.sh;;
10     source)
11         echo "using source..."
12         . ./2.sh;;
```



执行 shell 脚本 IV

```
13      *)
14      echo "using fork by default..."
15      ./2.sh;;
16  esac
17  echo "PID of 1.sh after exec/source/fork:$$"
18  echo "1.sh:\$A is $A"
```

2.sh

```
1  #!/bin/bash
2  echo "PID of 2.sh:$$"
3  echo "2.sh get \$A=$A from 1.sh"
4  A=C
5  export A
6  echo "2.sh:\$A is $A"
```



执行 shell 脚本 V

- 运行结果:

```
1 [hop@Worm shell]$ ./1.sh fork
2 PID of 1.sh before exec/source/fork:3342
3 1.sh:$A is B
4 using fork by default...
5 PID of 2.sh:3343
6 2.sh get $A=B from 1.sh
7 2.sh:$A is C
8 PID of 1.sh after exec/source/fork:3342
9 1.sh:$A is B
10 [hop@Worm shell]$ ./1.sh source
11 PID of 1.sh before exec/source/fork:3344
12 1.sh:$A is B
13 using source...
```



执行 shell 脚本 VI

```
14 PID of 2.sh:3344
15 2.sh get $A=B from 1.sh
16 2.sh:$A is C
17 PID of 1.sh after exec/source/fork:3344
18 1.sh:$A is C
19 [hop@Worm shell]$ ./1.sh exec
20 PID of 1.sh before exec/source/fork:3345
21 1.sh:$A is B
22 using exec...
23 PID of 2.sh:3345
24 2.sh get $A=B from 1.sh
25 2.sh:$A is C
```



shell 变量

- shell 的变量分为四种类型：用户自定义变量、环境变量、位置参数和预定义变量。
- shell 变量的值共有三种可能：未设定、空值、非空值。



环境变量 I

- shell 的环境变量实际上就是具有某个特定值的一个名称。这个名称不能包括 \$ 和空格；
- 环境变量分为系统预定义的变量和用户自定义的变量两类；
- 环境变量的设定方法为：<variable>=<value>，等号左右不能有空格；
- 常见的系统环境变量及其含义如下表所示；
- 用户也可以修改这些系统环境变量。



环境变量 II

变量名	含义
HOME	用户主目录
HZ	时钟中断频率
LOGNAME	用户登录名
MAIL	用户的邮件目录
PATH	命令搜索路径序列
SHELL	用户 shell 类型
TERM	用户终端类型
SHLVL	Shell 级别，登录 shell 为 1
PWD	当前工作目录
USER	用户名
GROUP	用户所属组名
PS1	一级提示符（缺省为 \$）



环境变量 III

PS2	二级提示符（缺省为 >）
IFS	内部域分隔符
TZ	时区
HOSTNAME	主机名



位置参数

- 当用户执行一个 shell 命令时，shell 将创建 10 个位置参数，分别是：`$0`、`$1`、...、和`$9`；
- `$0`表示命令的文件名本身，而`$1`、`$2`、...、`$9`分别表示命令的第 1 个、第 2 个、...、第 9 个命令行参数；
- 如果一个命令的命令行参数多于 9 个，那么后面的参数就不能直接得到了，这时就需要使用 shell 的内部命令`shift`了；
- `shift`命令的作用是将位置参数左移，即：`$1=$2`；`$2=$3`；...，而`$9`此时就等于刚才没有出现的第 10 个命令行参数；
- 另外内部命令`set`用于给位置参数赋值。



预定义变量 I

变量名	含义
<code>\$*</code>	命令行中的所有参数，从 <code>\$1</code> 开始，用空格分开，不限于 9 个
<code>\$0</code>	与 <code>\$*</code> 类似，但是它的值是多个字符串，而不是一个
<code>\$#</code>	位置参数的总数，不包括 <code>\$0</code>
<code>\$?</code>	命令返回的值（十进制），不能被赋值命令修改
<code>\$\$</code>	当前命令的 <code>pid</code> 号，不能被赋值命令修改
<code>!</code>	在后台运行的最近一个进程的 <code>pid</code> 号



预定义变量 II

- 除非特别声明，shell 的变量是局部的，只在当前 shell 有效，而在其子 shell 中无法使用此变量。除非使用 **export** 命令将变量输出。并且子 shell 不能更改由其父 shell 设置的变量值，除非使用 **source** 命令将变量输出；
- 在 shell 脚本中调用 **cd** 来改变工作目录时，不会影响其父 shell 的当前目录



预定义变量 III

- 范例:

```
1 [Apple]$ cat teststarat
2 #!/bin/bash
3 #@(#) test 'for' loops with $* and $@
4 #
5 echo "Test star"
6 echo "We have $# arguments"
7 for arg in $*
8 do
9     echo $arg
10 done
11
12 echo "Test at"
13 echo "We have $# arguments"
```



预定义变量 IV

```
14 for arg in "$@"
15 do
16     echo $arg
17 done
18 # End of file
19
20 [Apple]$ ./teststarat "a b" c
21 Test star
22 We have 2 arguments
23 a
24 b
25 c
26 Test at
27 We have 2 arguments
28 a b
```

预定义变量 V

29

c



命令替换

- 根据第三章的“附录一”我们可知，反引号`是用来作命令替换的；
- 另外，`$()`也可以用于命令替换，尤其是在嵌套的命令替换场合，但是并不是所有的标准 `shell` 都支持。:(



变量替换 I

- 最简单的变量替换是在变量名前加上\$符号。
- 如果需要给变量的值加上后缀，那么就要用花括号把变量名括起来，下例将显示 Larger:

```
1 word="Large"  
2 echo ${word}r
```

- 下面列出几种使用花括号进行变量替换的情况：
- 记忆的方法为：
 - #是去掉左边（键盘上#在\$的左边）
 - %是去掉右边（键盘上%在\$的右边）
 - 单一符号是最短匹配；
 - 两个符号是最长匹配；
 - 带冒号是检查未设定和空值，不带冒号只检查未设定。



变量替换 II

替换方式	含义
<code>\${VAR}</code>	基本替换，花括号限定变量名的开始和结束
<code>\${VAR-WORD}</code>	若 <code>VAR</code> 未设定，则返回 <code>WORD</code> 的值。不改变变量的值
<code>\${VAR:-WORD}</code>	若 <code>VAR</code> 未设定或为空值，则返回 <code>WORD</code> 的值；否则返回 <code>VAR</code> 的值。不改变变量的值
<code>\${VAR=WORD}</code>	若 <code>VAR</code> 未设定，则返回 <code>WORD</code> 的值，同时将 <code>WORD</code> 的值赋给 <code>VAR</code>
<code>\${VAR:=WORD}</code>	若 <code>VAR</code> 未设定或为空值，则返回 <code>WORD</code> 的值，同时将 <code>WORD</code> 的值赋给 <code>VAR</code>
<code>\${VAR+WORD}</code>	若 <code>VAR</code> 已设定，则返回 <code>WORD</code> ；否则返回空。不改变变量的值



变量替换 III

`${VAR:+WORD}`

若 `VAR` 为非空值，则返回 `WORD`；否则返回空。不改变变量的值

`${VAR?MESSAGE}`

若 `VAR` 未设定，则将 `MESSAGE` 的值输出到标准错误和标准输出，同时 `shell` 也显示出 `VAR` 的名字；否则返回 `VAR` 的值

`${VAR:?MESSAGE}`

若 `VAR` 未设定或为空值，则将 `MESSAGE` 的值输出到标准错误和标准输出，同时 `shell` 也显示出 `VAR` 的名字；否则返回 `VAR` 的值

`${#VAR}`

返回 `VAR` 的长度。如果 `VAR` 是 `*` 或 `@`，则返回 `$@` 中元素的个数

`${VAR#WORD}`

返回删除掉 `WORD` 的最短匹配之后的字符串，从左到右

`${VAR##WORD}`

返回删除掉 `WORD` 的最长匹配之后的字符串，从左到右



变量替换 IV

<code>\${VAR%WORD}</code>	返回删除掉 <code>WORD</code> 的最短匹配之后的字符串，从右到左
<code>\${VAR%%WORD}</code>	返回删除掉 <code>WORD</code> 的最长匹配之后的字符串，从右到左
<code>\${VAR:offset:length}</code>	返回从第 <code>offset</code> 个字符开始的长度为 <code>length</code> 的子字符串
<code>\${VAR/s/t}</code>	将第一个 <code>s</code> 的最长匹配替换为 <code>t</code> 后返回
<code>\${VAR//s/t}</code>	将所有 <code>s</code> 的最长匹配替换为 <code>t</code> 后返回



变量替换 V

- 上述替换方式参见**bash**手册页的 `EXPANSION` 小节。
- 例: `examples/shell/varsub`



数组 I

- **bash**提供一维数组变量。任何变量都可以作为数组的元素，数组的大小没有限制，其下标从 0 开始；
- 声明数组用带有 **-a** 选项的 **declare** 命令：

```
1 declare -a VAR[SUB]
```

其中下标 **SUB** 可以省略；

- 数组变量可以进行单个元素的赋值，也可以进行一组元素的赋值：

```
1 VAR[ `SUB` ]=VALUE
```

其中，**SUB** 作为一个算术表达式必须得到一个大于或等于 0 的值；

```
1 VAR=(VALUE1 VALUE2 ... VALUEn)
```



数组 II

其中每个 **VALUE_i** 具有 `[[SUB]=] STRING` 的形式，如果提供了下标，则它就是赋值元素的索引，否则赋值元素的索引就是上一次赋值的索引加一；

- 在声明的同时也可以进行赋值；
- 数组的任何元素都可以通过 `${VAR[`SUB`]}` 来引用；
- 使用命令 **unset** 来清除数组的某个元素或整个数组。



数值运算 I

- shell 中的变量缺省时一律当作字符串来处理;
- 当需要进行数值运算时, 可以将算术表达式用`$(())`括起来;
- `$(())`中的变量可以不带`$`符号;
- **bash**中的运算种类如下表所示 (优先级由高到低):

<code>id++ id--</code>	后置增量运算和后置减量运算
<code>++id --id</code>	前置增量运算和前置减量运算
<code>- +</code>	单目运算: 取负号和取正号
<code>!</code>	逻辑“非”、比特“非”
<code>**</code>	幂
<code>* / %</code>	乘、除、取余
<code>+ -</code>	加、减
<code><< >></code>	左/右比特移位



数值运算 II

<code><= >= < ></code>	关系运算：大于和小于
<code>== !=</code>	关系运算：相等和不等
<code>&</code>	比特“与”
<code>^</code>	比特“异或”
<code> </code>	比特“或”
<code>&&</code>	逻辑“与”
<code> </code>	逻辑“或”
<code>expr?expr:expr</code>	条件三目运算
<code>= *= /= %=</code>	
<code>+= -= <<= >>=</code>	
<code>&= ^= =</code>	
<code>expr1,expr2</code>	顺序运算



数值运算 III

- 圆括号可以用于改变运算的次序；
- 以 0 开头的数字常量表示八进制数；以 0x 或 0X 开头的表示十六进制；
- 可以通过表达式 `[base#]n` 来表示 2 到 64 进制的数。其中 `base` 表示进制的基，`n` 为此进制的数；方括号表示可选，不是表达式的一部分；`base` 缺省时为 10；
- 大于 9 的数字依次用小写字母、大写字母、@ 和下划线 `_` 表示。如果进制小于等于 36，则大、小写字母含义相同；
- 计算的结果一律用十进制表示。



shell 的内部命令 I

命令名	含义
<code>;</code>	空命令，返回值为 0
<code>exec cmds_list</code>	在当前 shell 环境中执行 <code>cmds_list</code>
<code>read var</code>	从标准输入中读取变量 <code>var</code> 的值
<code>readonly var</code>	设定 <code>var</code> 为只读变量
<code>time cmd</code>	显示在当前 shell 中运行的所有进程累计的用户时间及系统时间
<code>ulimit [-f] N</code>	指定创建文件时文件的最大块数， <code>-f</code> 选项表示写文件时也受限制
<code>umask [[n]nnn]</code>	指定用户在创建文件时的缺省权限，若省略 <code>nnnn</code> ，则 shell 显示出当前的设置情况
<code>unset <name></code>	清除 <code>name</code> 的设置，变量或函数，但是不包括 shell 预先定义的变量



shell 的内部命令 II

`wait [N]`

等待子进程`N`的结束并返回子进程`N`的返回值。如果没有参数`N`，则等待其所有子进程结束

`set +|- <char>`

设置 shell 执行时的一些选项，+ 表示设置，- 表示取消



shell 的内部命令 III

- set 命令的选项有：

v verbose, 回显

x 预处理后的回显, 且在命令前有一 + 号

e 若 shell 的任一命令返回非零值, 则立即终止退出

n 并不真正执行 script 中的命令

t 指示 shell 在读取和执行完当前输入行剩余部分的命令后退出



shell 命令的组合方式 I

- 主要组合方式
 - 重定向机制
 - 管道
 - 命令序列
- 高级重定向：
 - 对于每个 shell 命令，系统自动打开三个文件：

0	标准输入
1	标准输出
2	标准错误输出



shell 命令的组合方式 II

- bash**的重定向操作总结如下：（当**noclobber**被设定时，输出重定向的文件如果已经存在且为普通文件，则输出重定向失败，若使用了**>|**，则不考虑**noclobber**选项。）

<code>> file</code>	重定向标准输出到 <code>file</code>
<code>< file</code>	重定向标准输入为 <code>file</code>
<code>>> file</code>	重定向标准输出到 <code>file</code> ，若 <code>file</code> 已存在，则追加
<code>> file</code>	在 noclobber 已设定的情况下强制将标准输出重定向到 <code>file</code>
<code>n> file</code>	在 noclobber 已设定的情况下强制将文件描述符 <code>n</code> 重定向到 <code>file</code>
<code><> file</code>	用 <code>file</code> 同时作为标准输入和标准输出
<code>n<> file</code>	用 <code>file</code> 同时作为文件描述符 <code>n</code> 的输入和输出



shell 命令的组合方式 III

<< label	Here-Document
n> file	重定向文件描述符n的输出到file
n< file	重定向文件描述符n的输入为file
n>> file	重定向文件描述符n的输出到file, 若file已存在, 则追加
n>&	复制标准输出到文件描述符n
n<&	复制标准输入到文件描述符n
n>&m	复制输出文件描述符m到文件描述符n
n<&m	复制输入文件描述符m到文件描述符n
&> file	重定向标准输出和标准错误到file
>&-	关闭标准输出
<&-	关闭标准输入
n>&-	关闭输出文件描述符n
n<&-	关闭输入文件描述符n
>&n	重定向标准输出到文件描述符n



shell 命令的组合方式 IV

<code><&n</code>	重定向标准输入为文件描述符 <code>n</code>
<code>2> file</code>	重定向标准错误至 <code>file</code>
<code>1>&2</code>	发送标准输出至标准错误
<code>2>&1</code>	发送标准错误至标准输出
<code>cmd1 cmd2</code>	管道线



shell 命令的组合方式 V

- 顺序组合：用分号将若干个命令分开写在一行上，各命令将顺序执行，整个组合的返回值为最后一个命令的返回值；
- 条件组合：

<code>cmd1 && cmd2</code>	只有在第一个命令的返回值为 0(True) 的情况下才执行第二个命令；
-----------------------------------	--

<code>cmd1 cmd2</code>	只有在第一个命令的返回值不为 0 的情况下才执行第二个命令；
---------------------------	--------------------------------



shell 命令的组合方式 VI

- 许多时候，我们在 shell 操作上，需要在一定条件下一次执行多个命令，也就是说，要么全部执行，要么全部不执行。此时就需要引入“**命令群组**”（command group）的概念。
- `()`和`{}`都可以用于将多个命令群组化，但二者略有区别：
 - `()`将产生一个子 shell 去执行其中的命令群组，称为“**nested sub-shell**”；
 - `{}`在本 shell 内执行命令群组并且要求命令群组必须以回车符或一个分号结束，称为“**non-named command group**”。



Here-Document I

- 另外**bash**中还提供另外一种称为**Here-Document**的结构，可以将用户需要通过键盘输入的字符串改为从程序体中直接读入，如密码等。下面的小程序演示了这个功能：

```
1  #!/bin/bash
2  passwd="mypasswd"
3  ftp -n localhost <<FTPFTP
4  user anonymous $passwd
5  binary
6  bye
7  FTPFTP
8  exit 0
9  # End of file
```



Here-Document II

- 这个程序在用户需要通过键盘敲入一些字符时，通过程序内部的动作来模拟键盘输入。请注意 here documents 的基本结构为：

```
1  command <<LABEL
2  statements
3  ...
4  LABEL
```

- 在需要键盘输入的命令后面直接加上<<符号，然后跟上一个特别的字符串作为标签，在该串后按顺序输入本来应该由键盘输入的所有字符，在所有需要输入的字符都结束后，重复一遍前面<<符号后的“标签”即表示该输入到此结束。
- Here documents 还有一种变体叫做**Here strings**，其格式为：
<<<word。其中word将被扩展，然后作为前导命令的标准输入使用。



测试 I

- 条件测试语句 **test**。其功能为根据后面的表达式的值返回 0/1。它有两种表达形式：

```
1 test expr
2 [ expr ]
```

- 条件测试表达式主要有以下几类：文件测试、字符串测试、数值测试、表达式组合。
 - 文件测试：

<code>-r filename</code>	若文件存在且用户可读则返回真
<code>-w filename</code>	若文件存在且用户可写则返回真
<code>-x filename</code>	若文件存在且用户可执行则返回真
<code>-s filename</code>	若文件存在且长度大于 0 则返回真
<code>-d filename</code>	若为一个目录则返回真
<code>-f filename</code>	若为一个普通文件则返回真



测试 II

- 字符串测试:

<code>string</code>	若 <code>string</code> 的值非空则返回真
<code>-z string</code>	若 <code>string</code> 的长度为 0 则返回真
<code>-n string</code>	若 <code>string</code> 的长度非 0 则返回真
<code>str1=str2</code>	若 <code>str1</code> 和 <code>str2</code> 的值相等则返回真
<code>str1!=str2</code>	若 <code>str1</code> 和 <code>str2</code> 的值不等则返回真
<code>str1<str2</code>	若 <code>str1</code> 按当前区域设置排在 <code>str2</code> 的前面则返回真
<code>str1>str2</code>	若 <code>str1</code> 按当前区域设置排在 <code>str2</code> 的后面则返回真



测试 III

- 数值（整数）测试:

<code>n1 -eq n2</code>	若n1和n2为数值且相等则返回真
<code>n1 -ne n2</code>	若n1和n2为数值且不等则返回真
<code>n1 -gt n2</code>	若n1和n2为数值且 $n1 > n2$ 则返回真
<code>n1 -ge n2</code>	若n1和n2为数值且 $n1 \geq n2$ 则返回真
<code>n1 -lt n2</code>	若n1和n2为数值且 $n1 < n2$ 则返回真
<code>n1 -le n2</code>	若n1和n2为数值且 $n1 \leq n2$ 则返回真



测试 IV

- 表达式组合:

<code>!expr</code>	非
--------------------	---

<code>expr1 -a expr2</code>	与
-----------------------------	---

<code>expr1 -o expr2</code>	或
-----------------------------	---

<code>\(expr\)</code>	圆括号用于改变优先级，反斜线用于取消圆括号的 shell 特殊含义： <code>(cmdlist)</code> 表示在子 shell 中运行 <code>cmdlist</code>
-----------------------	---



命令

- **exit**命令: **exit** [n] 强行使一个脚本终止, 将n的值返回给调用进程。
- **trap**命令: **trap** [-l] [[commands] signals] 让你的脚本接收信号, 并有选择地对它们起作用:
 - -l选项将列出所有信号码及其助记名;
 - 指定当 shell 收到signals中列出的信号后就去执行commands中的命令;
 - **trap**语句必须作为继#!/bin/bash后的第一句非注释代码。



分支 I

- **if**语句:

```
1  if expr1
2  then
3      cmds_list1
4  [[elif expr2
5  then
6      cmds_list2]
7  else
8      cmds_list3]
9  fi
```



分支 II

- case 语句:

```
1 case TestString in
2   pattern1)
3     cmds_list1;;
4   pattern2)
5     cmds_list2;;
6     .....
7   patternN)
8     cmds_listN;;
9 esac
```



循环 I

- **while**语句:

```
1 while expr
2 do
3     cmds_list
4 done
```

- 每次执行到**while**语句时都测试**expr**的值，为**True**时执行循环语句块，否则终止。
- **until**语句:

```
1 until expr
2 do
3     cmds_list
4 done
```

循环 II

- 每次执行到`until`语句时都先执行循环语句块，然后测试`expr`的值，为`False`时继续下一次循环，否则终止。
- `for`语句：

```
1 for var in value_list
2 do
3     cmds_list
4 done
```

或

```
1 for (( expr1 ; expr2 ; expr3 ))
2 do
3     cmds_list
4 done
```



循环 III

- 在值表上迭代，对值表中的每个值执行一次循环语句块。
- `select`语句：

```
1 select name [ in word ]
2 do
3     cmds_list
4 done
```

- 将值表中的每一项前面加上一个数字输出到标准错误上，然后输出一个三级提示符`PS3`。如果用户的输入是某个相应的数字，则将相应的值项赋给变量`name`；如果用户的输入为空，则再次显示值表中的每一项和前导数字；如果用户的输入为`EOF`，则结束`select`命令。
- 用户的整个输入将保存在一个名为`REPLY`的变量中。
- 对用户的每个合法输入都将执行一遍`cmds_list`直到遇到`break`命令为止。



循环 IV

- **break**语句:

```
1 break [n]
```

- 退出当前的命令块。可以指定退出**n**重嵌套。
- **continue**语句:

```
1 continue [n]
```

- 跳过命令块中的剩余命令。**n**指明受影响的循环层数。
- 只能用在**do-done**之间。



函数 I

- shell 函数由若干命令组成，基本格式为：

```
1 function FunctionName
2 {
3     cmds_list
4 }
```

或

```
1 FunctionName()
2 {
3     cmds_list
4 }
```

- 函数体中最后一条语句后必须加一个分号作为函数定义的结束；



函数 II

- 当 **FunctionName** 与已有的别名重复时，第二种格式会失效；
- 函数编写完毕后，需要用 **source** 命令对其进行处理之后才能在别处引用；
- 在函数内部，位置变量和 **\$#** 变量都是局部的，而其他普通变量是全局的；
- 为了防止变量作用域的混乱，请在使用变量时显式地指出其作用域；(**local**)
- 所有函数在使用前必须定义。这意味着必须将函数放在脚本开始部分，直至 **shell** 解释器首次发现它时，才可以使用。调用函数仅使用其函数名即可；
- 向函数传递参数就像在一般脚本中使用特殊变量 **\$1**, **\$2**, ..., **\$9** 一样，函数取得所传参数后，将原始参数传回 **shell** 脚本，因此最好先在函数内重新设置变量保存所传的参数。这样如果函数发生错误，就可以通过已经本地化的变量名迅速加以跟踪。



shell 历程 I

- ① 系统引导过程中显示在屏幕上的信息可以使用 `dmesg` 命令查看。
- ② 在终端准备好进行操作之前系统运行的步骤可归纳为下述四步：

```
init---getty---login---shell
```

- ③ `init` 是系统的根进程，所有其他的进程都是它的后代。`init` 在启动时要去读取 `/etc/inittab` 文件中的设置，以便决定应该选择哪一种运行模式，不同的运行模式有着不同的系统配置。



shell 历程 II

④ 大部分 Linux 拥有 0-6, S, s 八种运行模式:

0	挂起 (Debian 中为关闭系统)
---	--------------------

1	单用户模式
---	-------

2	多用户, 无 NFS
---	------------

3	多用户, 有 NFS
---	------------

4	未用
---	----

5	X11 (Debian 中 2-5 均为多用户模式)
---	----------------------------

6	重新引导
---	------

S,s	单用户模式, 不需要 <code>/etc/inittab</code> 文件
-----	---



shell 历程 III

- ⑤ 运行级别 1 仅允许超级用户登录，并且仅挂装根分区；
- ⑥ `init`运行的第一个脚本是`/etc/init.d/rcS`。
- ⑦ 检测安装了 `sysv-rc` 还是 `file-rc` 软件包，以下假设安装了 `sysv-rc` 软件包。
- ⑧ `/etc/init.d/rcS`运行`/etc/rcS.d/`目录下面的所有脚本来执行初始化，包括：检查并挂载文件系统、装载内核模块、启动网络服务、设置时钟等。
- ⑨ 系统启动进程完成后，`init`启动所有在默认级别配置中需要启动的服务，到 `/etc/rc.d/rc[0-6].d`中的一个目录去执行其中的脚本；
- ⑩ 当系统收到用户要求登录的请求之后将产生一个独立的子`init`进程，用于连接所有的终端，并为用户打开标准输入、输出和错误文件，然后`fork`并`exec` `getty`；
- ⑪ `getty`设置终端类型、速率和连接协议，显示`/etc/issue`文件的内容并提示用户输入登录名；



shell 历程 IV

- 12 `getty`读取用户的登录名并将其作为参数传递给`exec`出来的`login`进程;
- 13 `login`提示用户输入口令并根据`/etc/passwd`中的存储值判断口令是否一致;
- 14 不存在指定的用户或三次密码错误将导致整个登录失败;
- 15 如果最终登录失败, `login`终止, 控制权将回到`init`, `init`得知登录终止后将为终端启动一个新的`getty`;
- 16 网络登录中由`xinetd`代替`getty`;
- 17 若登录成功, 则由`login`执行下述任务:
- 18 更新系统记帐文件;
- 19 检查用户是否有未阅读的邮件;
- 20 显示`motd`(message of the day);
- 21 执行相应的启动文件用于设置各种环境变量;



shell 历程 V

- 22 查阅`/etc/default/login`文件来建立其他的环境变量 (`TZ`, `HZ`, `ALTSHELL`等);
- 23 移到用户的主目录;
- 24 从`/etc/passwd`中得到用户的 `uid` 及其 `gid` 以便判定用户对文件的权限;
- 25 设置`umask`为 `022` 或 `002`;
- 26 `exec`由`/etc/passwd`最后字段指定的 `shell`;
- 27 此时用户的登录过程结束, 用户应该已经看到了`PS1`, 这表示系统已经准备好接受用户的任务了。
- 28 当用户注销时实际上是终止了这个 `shell` 进程, 这个操作将唤醒此 `shell` 进程的父进程`init`, 因此屏幕上将再次出现登录提示;
- 29 当用户输入一行命令时, `shell` 的处理过程为:
 - 1 分析输入流来区别出命令名、选项和参数



shell 历程 VI

- ② 扩展任何通配符
- ③ 将变量名转换为它们的值
- ④ 找到由命令名指定的执行文件
- ⑤ 检查用户的执行权限
- ⑥ 如果可执行文件是一个二进制的文件，登录 shell 将生成一个子进程来执行这个命令，
- ⑦ 所有的选项和参数都会传递到这个进程。如果使用了后台操作符 `&`，则父进程（登录 shell）不等待子进程的结束便立即显示提示符并使子进程独立地在后台运行。如果没有 `&`，则父进程必须等待子进程的结束，然后再返回提示符。
- ⑧ 如果可执行文件是一个 ASCII 的 script，则登录 shell 将产生一个子 shell，这个子 shell 将直接从 script 里读取命令执行，而不是从标准输入读取命令。



The End

The End of Chapter V.

