第一章 操作系统及UNIX Shell

- 1.1 什么是操作系统
- 1.2 操作系统的分类
- 1.3 UNIX操作系统的发展史
- 1.4 UNIX操作系统的功能模块
- 1.5 Unix/Linux Shell概述
- 1.6 Unix/Linux Shell 命令
- 1.7 Unix/Linux Shell 命令进阶
- 1.8 Unix/Linux Shell编程
  - 1.8.1 Shell程序
  - 1.8.2 Shell程序的特殊变量
  - 1.8.3 Shell语句
  - 1.8.4 函数

➢1.8.1 shell程序

➢什么是shell程序?

由shell命令、shell控制结构和注释、UNIX实用程序以及用户命令(由某种程序编译形成的可执行文件)组成的命令文件称为shell程序，也可称为shell script。

例如，my_shell是一个shell程序，文件内容为：

```
#! /bin/sh
if  test -f $1.c
then
   #  compile
   cc -o $1 $1.c
   $1
else
   echo $1 not found
fi
```

表示本程序采用B shell语法解释

if...then...else...fi是shell的if 结构

由#引起注释行

cc是UNIX的C语言编译命令

$1是shell程序的输入变量，这里作为用户命令使用

echo是shell命令

➢1.8.1 shell程序

➢shell程序的执行方式

(1) 先令需要执行的shell文件成为可执行，然后执行。

例如，采用以下形式：

$ chmod +x *shell文件*

$ ./*shell文件*

例如：

$ chmod +x my_shell

$ ./my_shell my_cat

从shell程序my_shell的内部来看，输入变量$1的值为my_cat，$1.c 的值为my_cat.c。

(2) 调用shell解释器执行shell文件。

形式为：

$ sh *shell文件*

例如：

$ sh my_shell my_cat

```
#!/bin/sh
if  test -f $1.c
then
   #  compile
   cc -o $1 $1.c
   $1
else
   echo $1 not found
fi
```

➢1.8.1 shell程序

➢ shell程序的识别

运行一个shell程序，必须用某个shell解释器(B shell、Bash或C shell)来解释这个shell程序。在UNIX中，不是通过文件的后缀来确定shell程序的种类，而是由程序文件的首行进行识别。

B shell的首行：	#! /bin/sh	则按B shell来解释程序
C shell的首行：	#! /bin/csh	则按C shell来解释程序
Bash的首行：	#! /bin/bash	则按Bash来解释程序

➢187.1 shell程序

➢ shell程序的识别

启动某个shell解释器运行一个shell程序，也仅仅表示在该程序运行时用该shell解释器来解释程序中的各种shell语句和控制结构。一旦程序结束，将仍然回到原来的shell环境。

在B shell环境下运行C shell程序，与执行/bin/csh转到C shell环境是不一样的。一旦这个C shell程序结束，将依然处在B shell环境之下而不会改为C shell环境。例如，csh_file是C shell程序：

```
$ csh_file          在B shell下运行C shell程序
$                   B shell环境
$ /bin/csh          在B shell下转到C shell
%                   C shell环境
```

第一章 操作系统及UNIX Shell

- 1.1 什么是操作系统
- 1.2 操作系统的分类
- 1.3 UNIX操作系统的发展史
- 1.4 UNIX操作系统的功能模块
- 1.5 Unix/Linux Shell概述
- 1.6 Unix/Linux Shell 命令
- 1.7 Unix/Linux Shell 命令进阶
- 1.8 Unix/Linux Shell编程
  - 1.8.1 Shell程序
  - 1.8.2 Shell程序的特殊变量
  - 1.8.3 Shell语句
  - 1.8.4 函数

➢1.8.2 shell程序的特殊变量

➢**shell程序的特殊变量**

- **shell命令的一般形式**

$ command [*option* ...] [*variable* ...]

如：

$ ls –l a*

$ mv a.out ../tmp

command 命令名

option 可选项参数，以“-”开头

variable 命令变量

option**和*****variable*****统称为命令行参数(parameter) 或者命令行变量。**

和shell命令类似，shell程序支持命令行参数，这些参数都可以传递到shell程序内部。

➢1.8.2 shell程序的特殊变量

➢**shell程序的特殊变量**

- **shell程序命令行参数：**

$ ./sh4 a.c b.c

- 特殊变量

从**shell**程序的内部看命令行参数，**B shell/Bash**用下列特殊变量表示：

$0	命令名本身
$1	命令行的第一个参数
$2	命令行的第二个参数
$3	命令行的第三个参数
$#	命令行参数的个数
$*	命令行的全体参数

➢1.8.2 shell程序的特殊变量

➢ 特殊变量

	B shell	Bash	C shell
命令行第n个参数	$n	$n	$n 或 $argv[n]
命令行参数个数（不含命令名 $0）	$#	$#	$#argv
命令行的全部参数（不含命令名 $0 ）	$*	$*	$* 或 $argv

➢1.8.2 shell程序的特殊变量

- **shell程序的特殊变量**
  - C程序中的命令行参数：

$ ./a.out hello world

```
int main(int argc, char* argv[])
{
   int i;
   for (i=0; i<argc; i++)
      printf("Argument %d is %s.\n", i, argv[i]);
   return 0;
}
```

➢1.8.2 shell程序的特殊变量

➢特殊变量示例 （B shell/Bash）

【例1-4】文件sh4 的内容为

```
echo shell file is [$0]          $0代表命令名
echo parameters are  [$*]        $*代表命令行(全体)参数
```

执行结果

```
$ sh4  a.c  b.c                  命令行参数为“a.c  b.c”
shell file is [sh4]              第一个echo的结果
parameters are [a.c b.c]         第二个echo的结果
```

➢1.8.2 shell程序的特殊变量

➢特殊变量示例 （B shell/Bash）

【例1-5】文件sh5 的内容为

echo [$*]□□:□“[$2]□□[$1]” **$2和$1为第2、第1个变量**

执行结果

$ sh5 36a1□□36a1.c **两个变量间有两个空格**

[36a1□36a1.c]□:□[36a1.c]□□[36a1]

shell对echo输出的空格采用以下原则：

输出命令中，无论在变量或者字符串之间有多少空格，输出时一律只取一个空格。

例如在以上冒号的两边空格数不相同，但是输出时都只有一个。在命令行的两个变量间有两个空格，输出时只产生一个，如**[36a1□36a1.c]**。

如果需要输出多个空格，应该用引号括起。例如在**“[$2]□□[$1]”**中有两个空格，则将产生全部空格，输出**[36a1.c]□□[36a1]** 。

➢1.8.2 shell程序的特殊变量

➢特殊变量示例 （B shell/Bash）

【例1-6】文件sh6 的内容为

```
echo  compile  $1.c
cc  -o  $1  $1.c
```

执行结果

```
$ sh6  34a1
compile 34a1.c
```

显示编译结果

【例1-7】文件sh7 的内容为

```
echo  $#  files  to  be  compiled
```

$#代表命令行变量的数目

执行结果

```
$ sh7  36a1  36a2  36a3
3 files to be compiled
```

第一章 操作系统及UNIX Shell

- 1.1 什么是操作系统
- 1.2 操作系统的分类
- 1.3 UNIX操作系统的发展史
- 1.4 UNIX操作系统的功能模块
- 1.5 Unix/Linux Shell概述
- 1.6 Unix/Linux Shell 命令
- 1.7 Unix/Linux Shell 命令进阶
- 1.8 Unix/Linux Shell编程
  - 1.8.1 Shell程序
  - 1.8.2 Shell程序的特殊变量
  - 1.8.3 Shell语句
  - 1.8.4 函数

➢1.8.3 shell语句

在shell程序或者shell命令中可以使用的shell语句包括：

- 命令语句
- 赋值语句
- 结束语句
- 输入/输出语句
- 运算语句
- 移位命令
- 各种控制结构语句
  - 测试表达式
  - **if** 结构
  - **for** 结构
  - **case** 结构
  - **while** 结构
  - **until**结构（**Bash**）

➢1.8.3 shell语句

➢命令语句

任何**unix**的**shell**命令或用户命令

➢赋值语句（注意不同**shell**）

给变量赋值：

变量=*值*　　或　**set** *变量*=*值*

撤销变量赋值：

变量=　　或　**unset** *变量*

➢结束语句

exit

在**shell**命令行中表示撤消注册，或者可以使用**logout**命令。在**shell**程序中表示结束程序的执行。

➢输入输出语句

➢ 输入语句

⊙**B shell和Bash的输入语句**

read *变量*　　从stdin读取*变量*

⊙**C shell的输入语句**

set *变量*=$<　　从stdin读取*变量*，$<表示标准输入

➢输出语句echo

- **B shell的echo语句**

  echo *变量*或*字符串*

- **Bash/C shell的echo语句**

  echo [*option*] *变量*或*字符串*

  ***option*的取值为：**

  -n　禁止换行(不换行)

  -e　允许用反斜杠处理特殊字符（\c, \n等）

➢输入输出语句

➢输入语句read示例

read *变量*　　从stdin读取*变量*

在**B shell/Bash**中，可以看到**read**语句的以下效果：

```
$ read w1
How□are□you          用户输入的字符串有三个字段
$ echo $w1
How are you           w1获得全部字段的值
$ read w1 w2
How□are□you          用户输入的字符串有三个字段
$ echo $w1
How                   w1获得第一个字段的值
$ echo $w2
are you               w2获得其余字段的值
```

可见，如果输入的字符串中有多个字段，而只指定了一个变量，那么**read**将把字段赋给该变量。如果输入多个变量，而字段数超过变量数，则在最后的字段全部赋予最后一个变量。

➢输入输出语句

➢输出语句示例

【例1-8】文件sh8（B shell）的内容为

```
echo enter word:
read  word
echo \\nword=$word
```

用\n产生换行

执行结果

```
$ sh8
enter word:
hello
```

用户输入数据

由\n产生的空行

```
word=hello
```

程序输出数据

➢输入输出语句

➢ 输出语句

echo语句的禁止换行问题，与操作系统以及shell的类型有关。在遵循System V的Linux中，使用\c表示禁止换行，而在遵循Bsd的Linux中，使用-n表示禁止换行。而实际上，在UNIX或者Linux的B shell，Bash以及C shell都有所不同。

⊙B shell输出语句的禁止换行

echo *变量*或*字符串* **\\c （-n有时候也可以用）**

⊙Bash输出语句的禁止换行

echo -n *变量*或*字符串*

⊙C shell输出语句的禁止换行

echo -n *变量*或*字符串*

➢输入输出语句

➢ 输出语句示例（B shell）

【例1-9】echo语句的功能将产生换行。如果禁止换行，需要在行尾加\c。又因为\是特殊字符，需用反斜杠或者单引号转义。

文件sh9为

```
echo  enter 2 word:□\\c        用反斜杠\对\c转义
read  w1 w2
echo  word1 is [${w1}], \      用反斜杠\表示续行
      word2 is [$w2]           本例${w1}和$w1的作用相同
```

执行结果

```
$ sh9
enter 2 words:□abc  def
word1 is [abc],  word2 is [def]
```

➢输入输出语句

➢输出语句echo

	B shell	Bash	
不产生换行： **nl?$**	echo nl?\\c echo “nl?\c” echo ‘nl?\c’	echo -e nl?\\c echo -e “nl?\c” echo -e ‘nl?\c’	echo -n nl?
多换一行： **nl?** **$**	echo nl?\\n echo “nl?\n” echo ‘nl?\n’	echo -e nl?\\n echo -e “nl?\n” echo -e ‘nl?\n’	

- **Bash/C shell缺少-e选项时，\c和\n将被解释为正常字符输出：**

```
$ echo nl?\\c
$ echo “nl?\c”
$ echo ‘nl?\c’
```
nl?\c
$

```
$ echo nl?\\n
$ echo “nl?\n”
$ echo ‘nl?\n’
```
nl?\n
$

- （注：**Bash的echo语句中-e选项允许反斜杠特殊转义：\\, \a, \b, \c, \e, \f, \n, \r, \t, \v, \0NNN, \xHH**）

➢运算语句

➢B shell的运算语句

语句格式：

变量=`expr□*表达式*□*运算符*□*表达式*`

其中，　*表达式*由变量、常量和*运算符*组成

*运算符*包括+、-、*、/(整除)、%(取余)等。

如有特殊字符，须加反斜杠\转义。

``是一对反引号。表示执行expr的结果将作为*变量*的值。

➢运算语句

➢运算语句示例（B shell）

【例1-11】文件sh11

```
echo enter 2 factors:  \\c
read f1 f2
p=`expr□$f1□\*□$f2`     ``是一对反引号，\*将特殊字符*转义
echo $f1□“*”□$f2□=□$p          用“*”将特殊字符*转义
```

其中，命令expr是计算变量f1乘f2，由于一对反引号``(命令结果替换符)的作用，其计算结果将被作为数据赋值给变量p。

如果不将特殊字符*转义，其数值将是当前目录下的所有文件。

执行结果

```
$ sh11
enter 2 factors: 3   5
3 * 5 = 15
```

➢运算语句

- **B shell的运算语句**
  变量=`expr□*表达式*□*运算符*□*表达式*`
- **Bash增加的运算语句**
  let□*变量*=*表达式 运算符 表达式*
  不能加空格，或者
  let□“*变量*□=□*表达式*□*运算符*□*表达式*”
  引号中空格可加可不加

例如：

```
let n=n+1                  或者
let “n=n+1”                或者
let “n□=□n□+□1”            都等价于
n=`expr□ $n□+□1`
```

显然第三种let的表达方式可读性最好，值得推荐。

➢运算语句

➢运算符

- **B shell的运算符：**
  算术运算符：+、-、*、/、%
- **Bash增加的运算符：**
  算术运算符：++(自加)、--(自减)、(乘幂)**
  移位运算符：<<(左移)、>>(右移)
  按位运算符：&(按位与)、|(按位或)、~(按位反)、^(按位异或)
  复合赋值符：+=、-=、*=、/=、%=、<<=、>>=、&=、^=、|=
  条件运算符：？ ：

【例1-25】运算语句示例

```
$ m=7 ; n=5
$ echo m = $m , n = $n
m = 7 , n = 5
$ let “s = m ^ n”                                    按位异或运算
$ echo s = $s
s = 2
$ let “s = (m & n) ? (m | n) : (m ^ n)”        条件运算
                                                （非零取前者，零取后者)
$ echo s = $s
s = 7
```

➢运算语句

⊙**B shell的运算语句格式：**

变量=`**expr**□*表达式*□*运算符*□*表达式*`

⊙**Bash新增运算语句格式：**

let□“*变量*□=□*表达式*□*运算符*□*表达式*”

⊙**C shell的运算语句格式：**

@□*变量*□=□*表达式*□*运算符*□*表达式*

例如，以下运算语句是等价的：

B shell：

p=`expr□$p□+□1`

Bash：

let□“p□=□p□+□1”

C shell：

@□p□=□$p□*□2

@□p++ 或者 @□p□++

@□p□+=□1

➢运算语句

➢运算符

➢ 浮点运算

运算语句中不能做浮点运算。

例如（**Bash**）：

$ n=1.3 n被视为字符串而不是浮点数

$ echo n = $n

n=1.3

$ let “n = n + 1” 无法执行浮点运算而显示出错信息

bash: let: 1.3: syntax error in expression ...

➢移位命令

- 移位命令的形式为：

shift

表示将传递到**shell**程序内部的命令行变量左移一位。

- 例如，执行以下命令：

$ cmd v1 v2 v3 v4

则该命令传递到**shell**程序内部的命令行参数为

$#	$0	$1	$2	$3	$4	$*
4	cmd	v1	v2	v3	v4	v1 v2 v3 v4

若执行**shift**，则命令行参数变为

$#	$0	$1	$2	$3	$*
3	cmd	v2	v3	v4	v2 v3 v4

➢移位命令

【例1-12】移位命令示例（B shell）。

文件sh12 的内容为

```
echo Num=$#  Var=$*
shift
echo Num=$#  Var=$*
shift
shift
echo Num=$#  Var=$*
```

执行结果：

```
$ sh12  a   bc  d  efg
Num=4  Var=a bc d efg
Num=3  Var=bc d efg
Num=1  Var=efg
```

- 控制结构
  - 测试表达式
  - **if** 结构
  - **for** 结构
  - **case** 结构
  - **while** 结构

➢控制结构

➢B shell测试表达式

在**if, while**等控制结构中，需要使用测试表达式，以便确定执行不同的语句

- 测试表达式的格式

[□*表达式*□**]**　　在方括号与*表达式*之间必须要用空格分开

或者　**test**□*表达式*　　在**test**与*表达式*之间必须用空格分开

如果*表达式*成立(*表达式*为真)，则测试的结果为真，否则为假。

- 字符串测试表达式

s1 □ **=** □ ***s2***	字符串***s1***与***s2***相同时为真
s1 □**!=** □ ***s2***	字符串***s1***与***s2***不相同时为真
-z □ ***s***	***s***是空字符串时为真
-n □ ***s***	***s***是非空字符串时为真

➢控制结构

➢测试表达式

- 关系运算表达式

  s1 关系运算符 s2

可使用的*关系运算符*如下：

-eq	等于	**-gt**	大于	**-ge**	大于等于
-ne	不等于	**-lt**	小于	**-le**	小于等于

- 文件测试表达式

-r □ *file*	文件***file***存在且可读
-w □ *file*	文件***file***存在且可写
-x □ *file*	文件***file***存在且可执行
-s □ *file*	文件***file***存在且非空
-d □ *dir*	***dir***是一个目录

- 逻辑运算表达式

! □ *s1*	非运算，若***s1***为真则“! ***s1***”为假
s1* □ -a □ *s2	与运算，若***s1***和***s2***都为真则为真
s1* □ -o □ *s2	或运算，若***s1***和***s2***都为假则为假

➢控制结构

➢测试表达式

- **B shell**的测试表达式的应用

测试表达式必须应用在**if, for , while**等控制结构中。

- **Bash**增加的测试表达式的应用

测试表达式还可以作为命令单独使用，可与命令状态值结合使用

【例1-26】测试表达式应用示例

```
$ [□“$TERM”□=□`echo□$TERM`□]      该测试必然成立
$ echo $?
0                                  前一命令运行成功，测试为真
$ [□$TERM□!=□`echo□$TERM`□]      该测试必然不成立
$ echo $?
1                                  前一命令运行失败，测试为假
```

又如：

```
$ n=130
$ [□$n□-gt□“200”□]
$ echo $?
1                                  前一命令运行失败，测试为假
```

➢控制结构

➢测试表达式

C shell测试表达式采用类似C语言的格式：

（□*表达式*□）

其中表达式示例：

s1 □ == □ ***s2***	字符串***s1***与***s2***相同时为真
s1 □!= □ ***s2***	字符串***s1***与***s2***不相同时为真
a □ >= □ ***b***	***a***大于等于***b***时为真

比较内容	B Shell (UNIX)	Bash (Linux)	C Shell (UNIX)
⊙ 关系运算符(数值比较)	-lt、-gt、-le、-ge、-eq、-ne	-lt、-gt、-le、-ge、-eq、-ne	<, >, <=, >=, ==, !=
⊙ 逻辑运算符	-a、-o、!	-a、-o、!	&&、||、!
⊙ 按位运算符	不提供	&、|、~、^	&、|、~、^
⊙ 复合赋值符	不提供	+=、-=、*=、/=、%= <<=、>>=、&=、^=、|=	+=、-=、*=、/=、%=
⊙ 条件运算符	不提供	? :	不提供
⊙ 字符串测试表达式	s1□=□s2	s1□=□s2	s1□==□s2
	s1□!=□s2	s1□!=□s2	s1□!=□s2
	-z□s	-z□s	-z□s
	-n□s	-n□s	-n□s

➢ 文件测试表达式

-r *file*	**文件*file*存在且可读**
-w *file*	**文件*file*存在且可写**
-x *file*	**文件*file*存在且可执行**
-s *file*	**文件*file*存在且非空(C shell不可用)**
-d *dir*	***dir*是一个目录**

C shell 新增：

-f *file*	**文件*file*存在并且是普通文件(plain file)**
-e *file*	**文件*file*存在(existance)**
-o *file*	**文件*file*存在而且用户是该文件的主人(owner)**

➢控制结构

➢if 结构

- **B shell/Bash if 结构的格式为：**

```
if [ 表达式 ]
then
    语句组
else          可以省略else结构
    语句组
fi
```

- **C shell if 结构的格式为：**

```
if (□表达式□)□then
    语句组
else if(□表达式□)□then
    语句组
else
  /* else if和else可以省略    */
    语句组
endif
```

➢控制结构

➢if 结构示例（B shell）

【例1-15】 文件sh15

```
if [ $# -eq 0 ]
then
   echo no args
else
   echo args=$*
fi
```

执行：

```
$ sh15
```

以及

```
$ sh15 a bc def
```

结果分别是什么？

【例1-16】文件sh16

```
if [ ! -d $1 ]
then
   echo $1 is not a directory
else
   echo $1 is a directory
fi
```

假定只存在目录src和bin，执行：

```
$ sh16 ab src efg  bin
```

结果是什么？

➢控制结构

请比较以下**shell**程序：

⊙B shell	⊙Bash	⊙C shell

#! /bin/sh
if [$# -eq 0]
  then
  echo no args
  else
  echo args=$*
  fi
	#! /bin/bash
	if test $# -eq 0
	  then
	  echo no args
	  else
	  echo args=$*
	  fi
		#! /bin/csh
		if ($#argv == 0) then
		  echo no args
		  else
		  echo args=$argv
		  endif

➢控制结构

➢B shell/Bash for 结构

for结构的格式为：

```
for 索引变量 in 变量表
do
    语句组
done
```

*变量表*是一个参数的枚举表
从*变量表*中逐个获得*索引变量*的值

【例1-17】文件sh17

```
for i in $*
do
   if [ ! -s $i ]
   then
      echo file $i is empty
   fi
done
```

执行：

```
$ >a
$ cp $HOME/.profile profile
$ sh17 a profile
```

>*file*的作用是生成一个空文件*file*

结果是什么？

➢控制结构

➢for 结构示例（B shell）

【例1-18】 文件sh18

```
for i in 0 1 2 3 4 5 6 7 8 9
do
   for j in 0 1 2 3 4 5 6 7 8 9
   do
      > $i$j
   done
done
```

执行：

```
$ sh18
```

结果是什么？

C shell for结构

foreach□*索引变量*□(□*变量表*□)
　　语句组
end

请比较以下**shell**程序：

⊙B shell	⊙Bash	⊙C shell

for i in $*
do
　if [! –s $i]
　then
　　echo file $i
　　echo is empty
　fi
done
	for i in $*
	do
	　if test ! –s $i
	　then
	　　echo file $i
	　　echo is empty
	　fi
	done
		foreach i ($argv)
		　if (! -s $i) then
		　　echo file $i
		　　echo is empty
		　endif
		end

➢控制结构

➢switch-case 结构

B shell/Bash 格式为：

```
case  变量  in
   情况1 )    语句组 ;;
   情况2 )    语句组 ;;
   …
   * )        语句组 ;;
esac
```

【例1-19】文件sh19

```
   for  i  in  $*
   do
      case  $i  in
         *.c)    echo  $i is C file ;;
         *.o)    echo  $i is OBJ file ;;
         *)      echo  $i is unkown file ;;
      esac
   done
```

执行：

```
   $ sh19  36a1.o  36a1 36a1.c
```

结果是什么？

C shell的switch-case结构格式为：

switch□(□*变量*□)
case *情况1*:
***语句组*; breaksw**
case *情况2*:
***语句组*; breaksw**
…….
default: *语句组*
endsw

请比较以下**shell**程序：

⊙B shell和Bash

```
case $i in
   *.c)  echo $i  is a C file ;;
   *.o) echo $i  is an OBJ file ;;
   *)    echo $i  is an other file;;
esac
```

⊙C shell

```
switch ( $i )
   case *.c:
            echo $i is C file
              breaksw
   case *.o:
             echo $i is OBJ file
              breaksw
   default:
            echo $i is other file
endsw
```

➢控制结构

➢while结构

B shell/Bash的while结构

格式为：

```
while [ 表达式 ]
do
   语句组
done
```

【例2-20】改写sh19为sh20

```
while [ $# -gt 0 ]
do
   case  $1 in
      *.c)    echo  $1 is C file ;;
      *.o)    echo  $1 is OBJ file ;;
      *)      echo  $1 is unkown file ;;
   esac
   shift
done
```

文件sh19

```
for  i in  $*
do
   case  $i  in
      *.c)    echo  $i is C file ;;
      *.o)    echo  $i is OBJ file;;
      *)      echo  $i is unkown file;;
   esac
done
```

书中[例1-8]的改写(第47页)

```
#! /bin/sh
# File: makec
# Func : compile C source files
# Usage : makec name1 ...
# Files to be compiled: file ...
if test $# -eq 0
then
    echo Usage: …
    exit
fi
echo There are $# files …
while [ $#  -gt  0 ]
do
    cc  -c  $1.c  2> $1.err
    if  [ -s $1.err ]
    then
        echo see error in file $1.err
    else
        echo $1.c is OK!
    fi
    shift
done
```

改写结果

```
#! /bin/sh
# File: makec
# Func : compile C source files
# Usage : makec name1 name2 …
# Files to be compiled: file =name.c
if test $# -eq 0
then
    echo Usage: makec name1 name2…
    exit
fi
echo There are $# files to be compiled
for  i  in  $*
do
    cc  -c $i.c 2> $i.err
    if  [ -s $i.err ]
    then
        echo see error in file $i.err
    else
        echo $i.c is OK!
    fi
done
```

C shell的while结构格式为

while□(□*表达式*□)
　　语句组
end

请比较以下**shell**程序：

⊙B shell	⊙Bash	⊙C shell
n=0 while [$# -gt 0] do 　n=`expr $n + 1` 　shift done echo n = $n	n=0 while test $# -gt 0 do 　let “n = n + 1” 　shift done echo n = $n	set n=0 while ($#argv > 0) 　@ n = $n + 1 　shift end echo n = $n

➢控制结构

➢Until 结构

until是Bash增加的控制结构，格式为：

until [*表达式*]
do
语句组
done

```
while [ $# -gt 0 ]
do
  echo $1
  shift
done
```

```
until [ $# -le 0 ]
do
  echo $1
  shift
done
```

第一章 操作系统及UNIX Shell

- 1.1 什么是操作系统
- 1.2 操作系统的分类
- 1.3 UNIX操作系统的发展史
- 1.4 UNIX操作系统的功能模块
- 1.5 Unix/Linux Shell概述
- 1.6 Unix/Linux Shell 命令
- 1.7 Unix/Linux Shell 命令进阶
- 1.8 Unix/Linux Shell编程
  - 1.8.1 Shell程序
  - 1.8.2 Shell程序的特殊变量
  - 1.8.3 Shell语句
  - 1.8.4 函数

➢1.8.4 函数

【注】函数是**B shell**、**Bash shell**和**C shell**都提供的功能。

➢函数的定义和调用

- 函数的定义格式

可采取以下任何一种方式。**function**是关键字，可以省略。

(1) function *函数名* ()
{
语句组
}
	(3) *函数名* ()
	{
	语句组
	}

(2) function *函数名* (){
语句组
}
	(4) *函数名* (){
	语句组
	}

- 函数的调用格式

函数名 [*实参*]

【例1-27】函数调用示例：sh27

```
thank () {        定义函数
        echo “Thank  you!”
 }
thank             调用函数
```

执行结果：

```
$ sh27
Thank you!
```

➢函数的外部调用

➢ 函数的生效和调用

除了在**Bash**程序中，可以调用已定义的函数之外，**Bash**还允许在**shell**命令行或者其他**shell**程序中调用已定义的函数。

当某个含有函数的**Bash**文件编制完成后，用**.(dot)**命令“运行”该文件。操作为：

$. ***shell文件***

“运行”***shell文件***，类似于“运行”配置文件，除了当场运行***shell文件***，还会**shell**环境中添加***shell文件***中定义的函数，从而可以在命令行或者程序中调用该函数。

➢函数的外部调用

【例1-28】函数外部调用示例：文件sh28

```
function SeeYou() {
   echo “Seeing you at my office!”
}
```

在shell中添加函数SeeYou，操作为：

```
$ . ./sh28                    “运行”shell文件sh28
$ SeeYou                      调用函数SeeYou
Seeing you at my office!
```

➢显示函数和删除函数

“运行”**shell**文件定义函数，与设置环境变量的作用相当，因此显示和删除函数与显示和删除环境变量的操作也类似。即

删除函数：

unset *函数*

显示函数：

set 将在显示所有的环境变量之后显示函数

例如，显示**SeeYou**函数的操作为：

```
$ set
... 显示环境变量 ...
function SeeYou()
{
   echo “Seeing you at my office!”
}
```

➢显示函数和删除函数

修改函数

如果需要修改函数，则在函数所在的**shell**文件修改之后，再“运行”该**shell**文件。即：

$. ***shell文件***

➢函数的参数传递

函数的形参遵循Bash特殊变量的用法，使用$*，$#，$1，$2等实现函数参数的传递。

【例1-29】函数参数传递示例1：sh29

```
prtDay()  {
   echo “today is $2”
   echo “yesterday was $1”
   echo “tomorrow will be $3”
}
prtDay 3-14  3-15  3-16
```

注意：　prtDay “3-14 3-15 3-16”　　不同。

执行结果

```
$ sh29
today is 3-15                  调用$2
yesterday was 3-14             调用$1
tomorrow will be 3-16          调用$3
```

➢函数的参数传递

【例1-30】函数参数传递示例2：

将sh27改为sh30

```
thank () {
   echo “Thank  $1!”
}
thanx () {
   thank  $1
   thank  $2
}
```

令函数生效的操作为：

```
$ . sh30
```

调用函数的操作为：

```
$ thanx  Wang Liu
Thank Wang!
Thank Liu!
```

【例1-27】函数调用示例：sh27

```
thank () {              定义函数
   echo “Thank  you!”
}
thank                   调用函数
```

➢附件B：B shell、Bash和C shell的比较

本课程要求通过上机实习以及编程练习，能够基本掌握B shell、Bash和C shell的使用。

因此，在附录中提供了B shell、Bash和C shell的比较一览表，供同学们查阅。

※基本特点 ※变量 ※特殊变量 ※Shell环境的识别 ※输入输出语句 ※运算语句 ※输入变量和程序变量	※表达式 ※控制结构及相关语句 ※命令行编辑 ※假名(别名)机制(C shell) ※命令史机制(C shell和Bash) ※工作目录栈(C shell和Bash) ※函数(Bash)

➢作　　业

- 上机及习题，将运行结果写在作业本上

(B shell程序)　　运行sh15（if结构）, sh18（for结构）,
sh19（switch-case结构）,
sh20（while结构）
E-3: 1-20.1（read）, 1-20.2（运算语句）

(Bash程序)　　sh30(Bash函数)，my_passwd(输入密码)
E-8: 1-31.1, 1-31.2,（变量与echo）
E-9: 1-31.4（特殊变量与通配符）,
1-31.5（for结构）

(C shell 程序)　　E-5：1-22.1（变量输入），1-22.2（运算表达式），
1-22.4（shell程序参数），1-22.5（for结构），
1-22.6（for结构）

- C语言编程练习E-16：3-4