



## 第3讲 Shell编程



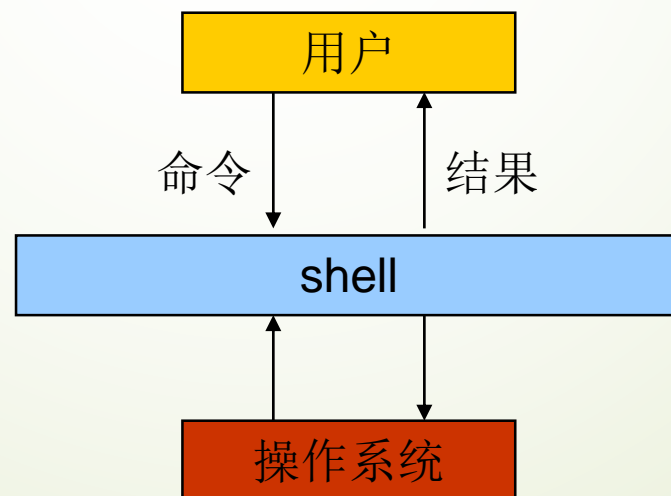
# 主要内容

- 3.1 Shell基础
- 3.2 Shell编程基础
- 3.3 Shell脚本中的特殊符号
- 3.4 Shell变量和表达式
- 3.5 Shell脚本控制流程
- 3.6 Shell脚本函数和数组
- 3.7 Bash调试

## 3.1 Shell基础—Shell概念

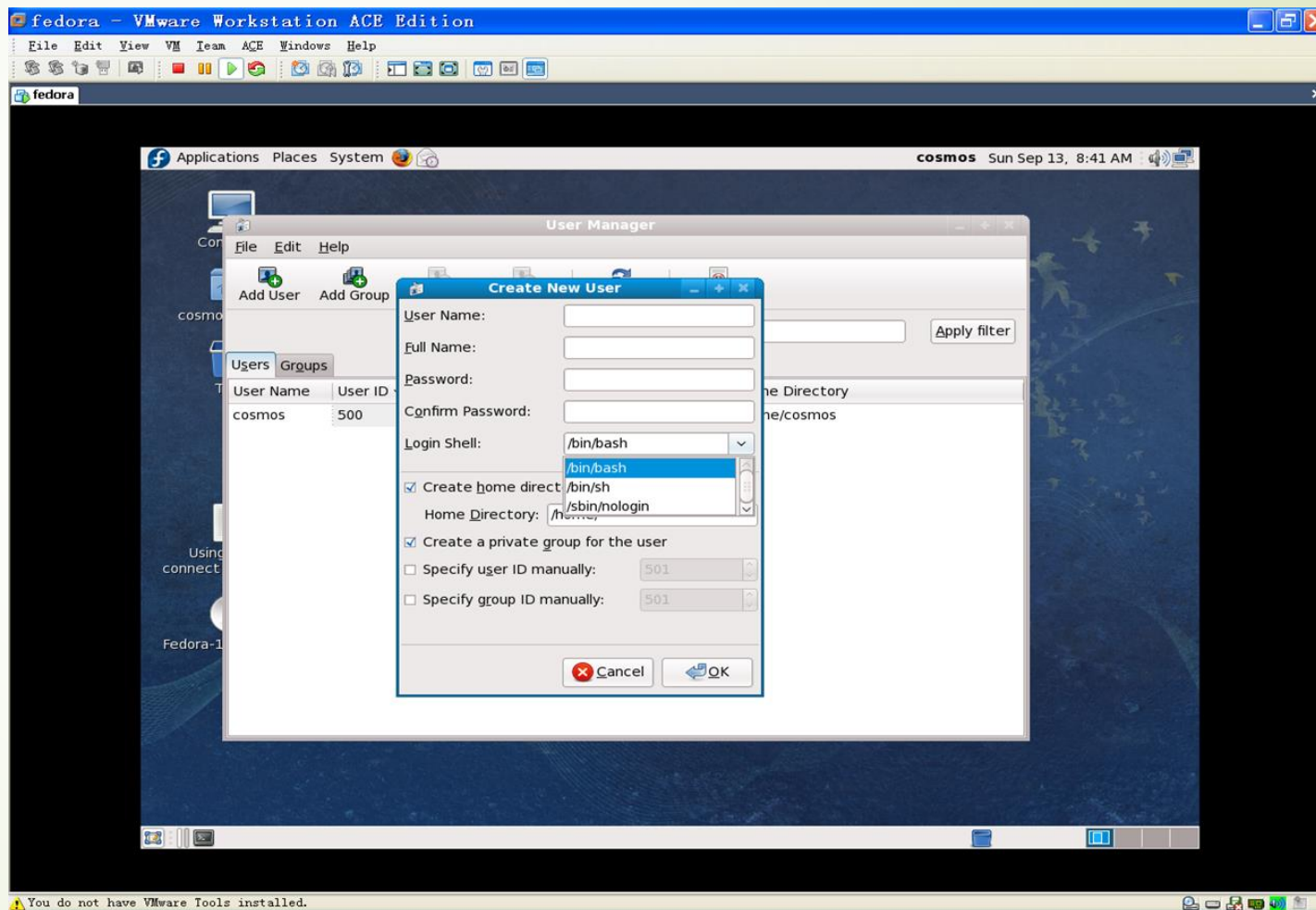
### ► 什么是Shell?

- ✓ Shell是一个命令解释器，可以用来启动、停止、编写程序。
- Shell是用户和UNIX/Linux操作系统内核程序间的一个接口。



# Shell简介—Shell种类及历史

- Bourne shell(sh):Stephen Bourne,1979
- Bourne Again Shell(bash)
- C shell (csh): Billjoy,70年代末期
- Korn Shell(ksh): DavidKorn,80年代中期



# Linux Shell — bash

- ➡ 支持Bourne shell、C shell和korn shell
- ➡ 标准shell为bash
  - ✓ 向下兼容bourne shell
  - ✓ 作业控制 (job control)
  - ✓ 别名功能 (aliases)
  - ✓ 命令历史(command history)
  - ✓ 命令行编辑功能
  - ✓ 提供更丰富的变量类型、命令与控制结构

# Shell简介—Shell作用

- 最主要的功用：解释用户在命令提示符下输入的指令
- 提供个性化的用户环境。
  - 通常在shell的初始化文件中完成。  
(.bash\_profile、.bash\_login、.bash\_cshrc等)这些特性包括：设置变量、搜寻路径、权限、提示符等
- 解释性的程序语言。
  - Shell程序→命令文件。由列在文件内的命令所构成：Linux命令和基本的程序结构



# Shell功能

## ■ 命令行解释功能

Shell试图解释命令行输入的一行字符。其基本格式：

`command arguments`

## ■ 启动程序

启动命令行中要求的程序。实际是内核执行该程序。

## ■ 输入/输出重定向

`$ ls -l >a.txt`

## ■ 管道连接

管道是输入输出重定向的特例，它将命令的输出直接连到另一个命令的输入。

# Shell简介—Shell功能

## ■ 变量维护

定义变量，使用变量等。

```
$LOOKUP=/usr/mydir  
$echo LOOKUP  
LOOKUP
```

## ■ 环境控制

用户个人环境的设置，包括用户的home目录、用户终端类型及PATH路径等。





# 主要内容

- 3.1 Shell基础
- 3.2 Shell编程基础
- 3.3 Shell脚本中的特殊符号
- 3.4 Shell变量和表达式
- 3.5 Shell脚本控制流程
- 3.6 Shell脚本函数和数组
- 3.7 Bash调试

# Shell编程基础

## ► Shell编程

将Linux命令与各种

流程控制和条件判断

来组合命令与变量，就可以进行Shell编程。

# Shell脚本的例子

- Shell 脚本是一个文本文件，可用vi编辑保存。

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

指明该脚本执行需要的  
命令解释器

```
LOG_DIR=/var/log
```

定义变量

```
# 如果使用变量,当然比把代码写死好.
```

```
cd $LOG_DIR
```

```
cat /dev/null > messages }
```

```
cat /dev/null > wtmp
```

```
echo "Logs cleaned up."
```

```
exit
```

退出Shell程  
序

执行UNIX内部命  
令，覆盖原文件内  
容

# Shell脚本的执行

- `#!/bin/bash`

在 Linux 系统中默认是 Bash

- `#!` 后边给出的路径名必须是正确的,否则将会出现一个错误消息,通常是 “Command not found”。

- `#!`也可以被忽略,不过这样脚本无法使用 shell 内建的指令。

- 如果在脚本行中加上`#!`,那么 bash 将把它认为是一个一般的注释行。

# Shell脚本的执行

## ■ 试一试

- 假如在脚本的第一行放入`#!/bin/rm`或者在普通文本文件中第一行放置`#!/bin/more`，然后将文件设为可执行权限执行，看看会发生什么？

# Shell脚本的执行

## ➤ Shell脚本的执行方法

sh scriptname

bash scriptname

➤ 不推荐使用 sh <scriptname>, 因为这禁用了脚本stdin中读数据的功能.

➤ 更方便的方法是让脚本通过chmod命令可以修改, 然后./scriptname测试它.

chmod 555 scriptname (允许任何人都具有可读和执行权限)

chmod +rx scriptname (允许任何人都具有可读和执行权限)

chmod u+rx scriptname (只给脚本可读和执行权限)

# Shell基本语法—Shell脚本的执行

- 为什么不直接使用 `scriptname` 来执行脚本？
- 如果你当前的目录下(`$PWD`)正好有你想要执行的脚本，为什么它运行不了呢？
- 失败的原因是，出于安全考虑,当前目录并没有被加在用户的`$PATH` 变量中。因此,在当前目录下调用脚本 `./scriptname` 这种形式。



# Shell脚本的退出及退出状态

- `exit` 命令被用来结束脚本。
- `exit n` .当n为0时表示执行成功，非0通常表示一个错误码。
- 脚本中将错误码n传递给BASH。
- 脚本中若无`exit`语句，则其返回状态由最后一条语句执行的状态决定。
- `$?`  读取最后执行命令的退出码
- 特定的退出码都有预定的含义，用户不应该在自己的脚本中使用它

# Shell脚本的退出及退出状态

```
#!/bin/bash
echo hello
echo $? # 返回0,因为执行成功
lskdf   # 不认识的命令.
echo $? # 返回非0 值,因为失败了.
echo
exit 113 # 将返回113 给 shell.
$ echo $?
```

# 退出码的含义

- ➡ 0表示成功，1-125用户可自定义具体含义

推出码	含义
126	文件不可执行
127	命令未找到
128及以上	收到一个信号



# 主要内容

- 3.1 Shell基础
- 3.2 Shell编程基础
- 3.3 Shell脚本中的特殊符号
- 3.4 Shell变量和表达式
- 3.5 Shell脚本控制流程
- 3.6 Shell脚本函数和数组
- 3.7 Bash调试

# Shell脚本的注释

## ➡ # 字符

注释,行首以#开头为注释(!是个例外)注释也可以存在于本行命令的后边。

echo 命令中被转义的#是不能作为注释的。

```
# This line is a comment.  
$echo #aabb bb 啥也不输出  
$  
$echo \#aabb bb  
$#aabb bb
```

## Shell一行中多个命令用 ; 分隔

- ➡ ; 命令分隔符,可以用来在一行中来写多个命令。

```
$echo hello; echo there
```

```
Hello
```

```
there
```

## , 逗号 和 \ 转义字符

```
$let "t2 = ((a = 9, 15 / 3))"  
$echo $t2;echo $a  
$5  
$9
```

, 逗号链接了一系列的算术操作,虽然里边所有的内容都被运行了,但只有最后一项被返回.

➡ \ 转义字符,如\X 等价于 "X" 或 'X'





# 主要内容

- 3.1 Shell基础
- 3.2 Shell编程基础
- 3.3 Shell脚本中的特殊符号
- 3.4 Shell变量和表达式
- 3.5 Shell脚本控制流程
- 3.6 Shell脚本函数和数组
- 3.7 Bash调试



# shell变量的类型

- 环境变量
- 用户定义变量
- 内部变量

# Shell基本语法—环境变量

- 它是定义和系统工作环境有关的变量，用户亦可重新定义该类变量。其包含：
  - **HOME** 用于保持注册目录的完全路径名
  - **PATH** shell按照该变量的顺序搜索与名称一致的可执行文件。
  - **TERM** 终端类型
  - **UID** 当前用户的标识
  - **PWD** 当前工作目录的绝对路径
  - **PS1** 主提示符，特权用户是#，普通用户是\$。

# Shell基本语法—用户定义变量

## ➤ 用户自定义变量规则：

变量名=变量值

### ■ 定义变量注意事项：

- 定义变量时，变量名前不需要加\$。

NAME=lyq

- 变量不需声明，可直接使用或者赋值
- 变量设为只读，使其不再改变

readonly 变量名

# Shell基本语法—用户定义变量

- 用户定义之变量名由字母和下划线组成，并且变量名第一个字母不能为数字。
- 使用变量时，在变量名字两边\$后面加上{ }

```
$SUN=sun
```

```
$echo ${SUN}day
```

```
$echo $SUNday #没有定义的变量输出为空
```

比较上述两条命令的输出结果。

- ✧ 命令行上同时对多个变量赋值，赋值语句之间用空格分开，变量赋值从右至左进行。

```
$X=x Y=y;echo x;echo y
```

```
$X=$Y Y=y;echo x;echo y
```

# 变量与引号

- " 双引号括起来的字符除 $\$$ 、 $\backslash\backslash$ 、 $'$ 、和 双引号之外都将作为普通字符对待。
- '单引号括起来的字符均作为普通字符出现。

```
$string='$PATH'
```

```
$echo $string
```

```
 $\$PATH$ 
```

```
$string="$PATH"
```

```
$echo $string
```

```
 $\$/usr/bin:/home/sxlyq$ 
```

# 变量与引号

- ` 后置引用,命令替换。其对应于键盘左上角的符号。其所括字符串在被Shell解释时, 首先执行其中的命令并将其结果代替该命令。

```
$pwd  
/home/sxlyq  
$string="current directory is  
`pwd`"  
$echo $string  
Current directory is  
/home/sxlyq
```

```
echo \ $A 显示为$A 如果  
不加\将显示为1234
```

```
echo ` 显示为`
```

```
echo \" 显示为双引号
```

```
echo \\ 显示为\
```





A=`date`

echo \$A #显示的不是date而是当时的时间串

如有一文件A的内容如下

ABCDEFGG

1234456

abcdefg

B=`cat A|grep 234`# 检索文件A中含有234的行

echo \$B 将显示为1234456

echo “\$B” 将显示什么?

echo ‘\$B’ 将显示什么?

# Shell基本语法—内部变量

## ■ 内部变量只能使用而无法修改或重定义

- ✓ \$# 传递给脚本参数的数量
- ✓ \$\* 所有传递给脚本的参数内容
- ✓ \$? 上条命令执行后返回的状态
- ✓ \$\$ 当前进程的进程号 → 最常见的用途是作为暂存文件的名称，以保证不会重复。
- ✓ \$! 后台运行的最后一个进程号
- ✓ \$0 当前执行的进程名
- ✓ @\$ 它是\$\*的另外一种形式，它不使用IFS。

## \$@与\$\*的区别

```
$ IFS=' '
```

```
$ set foo bar bam
```

```
$ echo "$@"
```

```
foo bar bam
```

```
$ echo "$*"
```

```
foobarbam
```

```
$ unset IFS
```

```
$ echo "$*"
```

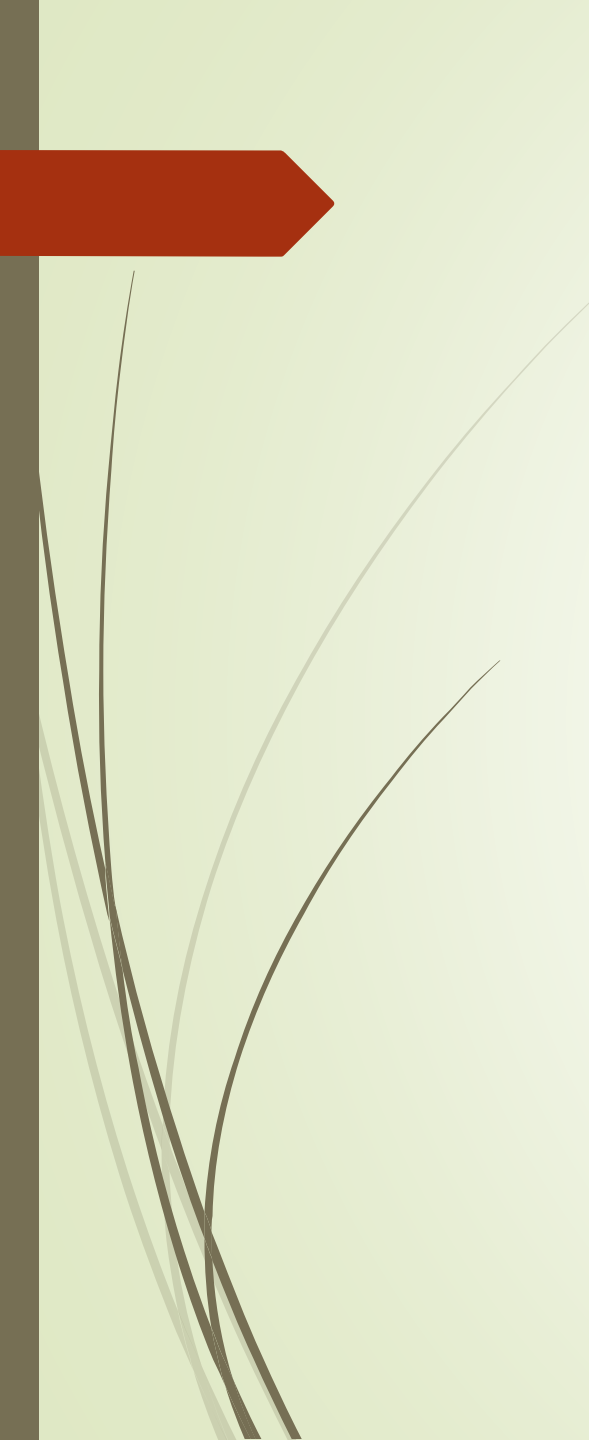
```
foo bar bam
```

# Shell位置参数变量

- 由shell在程序运行时设置。

在命令行中按照各自的位置决定的变量，程序名之后输入的参数，之间用空格分割，第一个参数可使用\$1取得，以此类推，\$0表示当前Shell程序的进程名。

- 共10个位置参数变量，从程序名开始依次为第0~9个
- Shift命令递归访问参数



```
#!/bin/bash
echo
echo "The name of this script is \"$0\"."
echo
if [ -n "$1" ]           # 测试变量
then
echo "Parameter #1 is $1"
fi
if [ -n "$2" ]
then
echo "Parameter #2 is $2"
fi
```

向这个脚本传递 10 个参数,如 **./scriptname 1 2  
3 4 5 6 7 8 9 10**

# Shell位置参数变量

- ➡ **shift** shift 命令重新分配位置参数,其实就是向左移动一个位置
- ➡ `$1 <--- $2, $2 <--- $3, $3 <--- $4, 等等`
- ➡ 之前的\$1 将消失,但是\$0(脚本名)是不会改变的。如果你使用了大量的位置参数,那么 shift 命令允许你存取超过 10 个参数。

# Shell位置参数变量

```
#!/bin/bash
until [ -z "$1" ] # 直到所有参数都用完
do
    echo -n "$1 "
    shift
done
echo
exit 0
```

使用#. /shift a b c def      执行该脚本，  
结果是什么呢？



# Shell基本语法—参数置换变量

- 根据不同条件给变量赋予不同的值。
- 变量=\${参数:-word}, \${parameter:-default}如果设置了参数，则用参数的值置换变量的值，否则用word置换。这两者大部分情况下相同。
- 变量=\${参数:=word}, \${parameter:=default}如果设置了参数，则用参数的值置换变量的值，否则把变量设置成word，然后再用word替换参数的值。
- 变量=\${参数:? Word}如果设置了参数，则用参数的值置换变量的值，否则就显示word并从shell中退出，如果省略了word，则显示标准信息。该种方式常用于出错指示。
- 变量=\${参数:+word} 如果设置了参数，则用word置换变量，否则不进行置换而使用 null字符串。

# Shell基本语法—参数置换变量

```
[cosmos@localhost ~]$ name=${username:-`whoami`}
[cosmos@localhost ~]$ echo ${name}
cosmos
[cosmos@localhost ~]$ username=aaa
[cosmos@localhost ~]$ name=${username:-`whoami`}
[cosmos@localhost ~]$ echo ${name}
aaa
[cosmos@localhost ~]$ unset username
[cosmos@localhost ~]$ name=${username:="Jerry"}
[cosmos@localhost ~]$ echo ${name}
Jerry
```

# Shell基本语法—参数置换变量

- 如果脚本中没有命令行参数 ,那么 default parameter将被使用.

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

```
DEFAULT_FILENAME=generic.data
```

```
filename=${1:-$DEFAULT_FILENAME}
```

```
echo ${filename}
```

- 将该脚本命名为：test.sh，分别执行./test.sh 及./test.sh abc

generic.data    和    abc

# Shell基本语法—变量的使用

- Bash 变量是不分数据类型的。
- Bash 并不区分变量但是依赖上下文,Bash也允许比较操作和算术操作.
- 决定这些的关键因素就是,变量中的值是否只有数字.

# 变量的使用风格

- `$(command)` 语法
- 它 ``command`` 功能相同，但更推荐使用，因为它易于使用，不会与双引号，单引号混淆。
- 另外，普通的变量使用推荐使用” `$variable`” 用法

# 数字型变量两种运算方法

- 对数字型变量进行运算的两种方法：
- (1) `$((...))`用法
- (2) `expr`表达式

# \$((...))用法

- 将需要求值的表达式包括在\$((...))中
- 例如:

```
#!/bin/sh
x=0
while [ "$x" -ne 10 ]; do
echo $x
x=$((x+1))
done
exit 0
```



## 与x= \$(..)用法的区别

- ➡ 两对圆括号用于算术替换
- ➡ 一对圆括号用于命令的执行和获取输出

## expr表达式

- expr 命令将它的参数当作一个表达式进行求值
- 例如: `x=`expr $x + 1`` 也可用 `x=$(expr $x +1)` 对x变量进行加一操作

# Shell变量表达式

➤ test命令：

test condition或者[condition] 命令进行条件测试

➤ 用在以下四种情况：

- ✓ 字符比较
- ✓ 数值的比较
- ✓ 文件操作
- ✓ 逻辑操作



## 变量表达式例子

```
#!/bin/sh
x=1
while [ "$x" -le 5 ]; do
echo $x
x=`expr $x \* 2`
done
exit 0
```

# Shell变量表达式

## ➡ 字符比较

test命令	含义	test命令	含义
str1=str2	当str1与str2相同时，返回真	-n str	当str的长度大于0时，返回真
str1!=str2	当str1与str2不同时，返回真	-z str	当str的长度是0是，返回真
str	当str不是空字符时，返回真		

# Shell变量表达式

## ➡ 整数操作符

test表达式	含义	test表达式	含义
int1 -eq int2	当int1等于int2时， 返回真	int1 -gt int2	当int1大于int2时， 返回真
int1 -ge int2	当int1大于/等于int2 时，返回真	int1 -ne int2	当int1不等于int2 时，返回真
int1 -le int2	当int1小于/等于int2 时，返回真		

# Shell变量表达式

## ■ 文件操作符

test表达式	含义	test表达式	含义
-d file	当file是一个 目录时，返回真	-s file	当file文件长度大于0时，返回真
-f file	当file是一个 普通文件时，返回真	-w file	当file是一个可写文件时，返回真
-r file	当file是一个 可读文件时，返回真	-x file	当file是一个可执行文件时，返回真



# Shell变量表达式

## ➡ 逻辑操作符

test表达式	含义
!expr	当expr的值是假时，返回真
expr1 -a expr2	当expr1和expr2值同为真时，返回真
expr1 -o expr2	当expr1和expr2的值至少有一个为真时，返回真

# Shell变量表达式

```
#!/bin/bash
a=4
b=5
echo
if [ "$a" -ne "$b" ]
then
    echo "$a is not equal to $b"
    echo "(arithmetic comparison)"
fi
if [ "$a" != "$b" ]
then
    echo "$a is not equal to $b."
    echo "(string comparison)"
fi
```

- # 变量 a 和b 既可以当作整型也可以当作是字符串.
- # 这里在算术比较和字符串比较之间有些混淆, 因为Bash变量并不是强类型的.

# Shell变量表达式

- `[cosmos@localhost ~]$ test -z ${name} && echo "name is null"`
- `name is null`
- `[cosmos@localhost ~]$ name=cosmos`
- `[cosmos@localhost ~]$ test -z ${name} && echo "name is null"`
- `[cosmos@localhost ~]$ test -n ${name} && echo "name is not null"`
- `name is not null`
  
- `[cosmos@localhost ~]$ test -1 -gt -2 && echo yes`
- `yes`

# Shell字符串操作

## ➡ 字符串长度

- ✓ `${#string}`
- ✓ `expr length $string`
- ✓ `expr "$string" : '.*'`

```
1 stringZ=abcABC123ABCabc
2
3 echo ${#stringZ}           # 15
4 echo `expr length $stringZ` # 15
5 echo `expr "$stringZ" : '.*'` # 15
```

# Shell字符串操作

## ➤ 从字符串开始的位置匹配子串的长度

✓ `expr match "$string" '$substring'`

✓ `$substring` 是一个正则表达式

✓ `expr "$string" : '$substring'`

✓ `$substring` 是一个正则表达式

```
stringZ=abcABC123ABCabc
```

```
#      |-----|
```

```
echo `expr match "$stringZ" 'abc[A-Z]*.2'` # 8  abcABC12
```

```
echo `expr "$stringZ" : 'abc[A-Z]*.2'`      # 8
```

```
echo `expr match "$stringZ" 'abc[A-Z]*'`    #6
```

# Shell字符串操作

## ➡ 索引

➡ `expr index $string $substring`

➡ 匹配到子串的第一个字符出现的位置。

```
stringZ=abcABC123ABCAbc
echo `expr index "$stringZ" C12`           # 6
echo `expr index "$stringZ" 1c`            # 3
echo `expr index "$stringZ" 2c`            # 3
echo `expr index "$stringZ" b2c`           # 2
echo `expr index "$stringZ" a45c`          # 1
```

# Shell字符串操作

## ■ 提取子串

■ `${string:position}` 在 `string` 中从位置 `$position` 开始提取子串. 如果 `$string` 为 “\*” 或 “@”, 那么将提取从位置 `$position` 开始的位置参数。

## ■ `${string:position:length}`

在 `string` 中从位置 `$position` 开始提取 `$length` 长度的子串.



# Shell字符串操作

```
stringZ=abcABC123ABCabc
```

```
echo ${stringZ:0} # abcABC123ABCabc
```

```
echo ${stringZ:1} # bcABC123ABCabc
```

```
echo ${stringZ:7} # 23ABCabc
```

```
echo ${stringZ:7:3} # 23A
```

# 有没有可能从字符结尾开始,反向提取子串?

```
echo ${stringZ:(-4)} # Cabc
```

# 使用圆括号或者添加一个空格来转义这个位置参数.

## Shell字符串操作

- `expr substr $string $position $length`
- 在 `string` 中从位置 `$position` 开始提取 `$length` 长度的子串

```
1 stringZ=abcABC123ABCabc
2 #      123456789.....
3 #      1-based indexing.
5 echo `expr substr $stringZ 1 2`      # ab
6 echo `expr substr $stringZ 4 3`      # ABC
```

# Shell字符串操作

➡ `expr match "$string" '\($substring\)`

从\$string 的开始位置提取\$substring,  
\$substring是一个正则表达式.

➡ `expr "$string" : '\($substring\)`

从\$string 的开始位置提取\$substring,  
\$substring是一个正则表达式.

# Shell字符串操作

```
stringZ=abcABC123ABCabc
```

```
echo `expr match "$stringZ" '\([b-c]*[A-Z]..[0-9]\)` #  
abcABC1
```

```
echo `expr "$stringZ" : '\([b-c]*[A-Z]..[0-9]\)` #  
abcABC1
```

```
echo `expr "$stringZ" : '\(.....\)` # abcABC1
```

# Shell字符串操作

- 子串消除
- `${string#substring}` 从\$string 的左边截掉第一个匹配的 \$substring
- `${string##substring}` 从\$string 的左边截掉最后一个匹配的 \$substring

```
stringZ=abcABC123ABCAbc
```

```
#      |----|
```

```
#      |-----|
```

```
echo ${stringZ#a*C}      # 123ABCAbc
```

```
echo ${stringZ##a*C}     # abc
```

# Shell字符串操作

- ➡ `${string%substring}` 从\$string 的右边截掉第一个匹配的 \$substring
- ➡ `${string%%substring}` 从\$string 的右边截掉最后一个匹配的\$substring

```
stringZ=abcABC123ABCabc
```

```
echo ${stringZ%b*c}    # abcABC123ABCa  
# 从$stringZ 的后边开始截掉'b'和'c'之间的最近的匹配
```

```
echo ${stringZ%%b*c}   # a  
# 从$stringZ 的后边开始截掉'b'和'c'之间的最远的匹配
```

# Shell字符串操作

- 子串替换
- `${string/substring/replacement}` 使用 `$replacement` 来替换第一个匹配的 `$substring`.
- `${string//substring/replacement}` 使用 `$replacement` 来替换所有匹配的 `$substring`

```
stringZ=abcABC123ABCabc
```

```
echo ${stringZ/abc/xyz}  
# xyzABC123ABCabc
```

```
echo ${stringZ//abc/xyz}  
# xyzABC123ABCxyz
```



# Shell字符串操作

`${string/#substring/replacement}`

如果\$substring 匹配\$string 的开头部分,那么就用  
\$replacement 来替换\$substring.

`${string/%substring/replacement}`

如果\$substring 匹配\$string 的结尾部分,那么就用  
\$replacement 来替换\$substring.

```
stringZ=abcABC123ABCabc  
echo ${stringZ/#abc/XYZ}  
# XYZABC123ABCabc  
echo ${stringZ/%abc/XYZ}  
# abcABC123ABCXYZ
```



# 主要内容

- 3.1 Shell基础
- 3.2 Shell编程基础
- 3.3 Shell脚本中的特殊符号
- 3.4 Shell变量和表达式
- 3.5 Shell脚本控制流程
- 3.6 Shell脚本函数和数组
- 3.7 Bash调试



## 3.5 Shell脚本控制流程

➤ 条件测试

➤ 流程控制

# 基本脚本编程—条件测试

## ► if then else语句

if 条件命令串

then

条件为真时的命令串

else

条件为假时的命令串

fi

# if 语句实例

```
➤ #!/bin/bash
➤ if [ $1 -le 10 ];then
➤     echo "a<=10"
➤ elif [ $1 -le 20 ];then
➤     echo "10<a<=20";
➤ else
➤     echo "a>20";
➤ fi
```

./testif.sh 30

./testif.sh 40

./testif.sh 15

# 基本脚本编程—流程控制

## ➡ case条件选择

case variable in

表达式1)

若干个命令行1

;;

表达式2)

若干个命令行2

;;

.....

\*) #用\*通配符来处理无匹配项情况


默认若干个命令行

esac

# case 实例1

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday #等待用户输入
case "$timeofday" in
yes | y | Yes | YES)
echo "Good Morning"
echo "Up bright and early this morning"
;;
[nN]*)
echo "Good Afternoon"
;;
*)
echo "Sorry, answer not recognized"
echo "Please answer yes or no"
exit 1
;;
esac
exit 0
```





## case 实例2

```
#!/bin/bash
read number
case $number in
  1|3|5|7|9) echo "odd number";;
  2|4|6|8|0) echo "even number";;
  *) echo "number is bigger than 9";;
esac
exit
```

# 脚本流程控制

## while循环

```
while condition do  
    command(s)  
done
```

## Until循环

```
until condition do  
    command(s)  
done
```

Shell还提供了true和false两条命令用于创建无限循环结构，它们的返回状态分别是总为0或总为非0

# 脚本流程控制

```
for arg in [list]
do
    command(s)...
done
```

- 注意:在循环的每次执行中,arg 将顺序的存取 list中列出的变量。
- list 中的参数允许包含通配符。

# while语句实例1


```
#!/bin/sh
foo=1
while [ "$foo" -le 20 ]
do
echo "Here we go again"
foo=$((foo+1))
done
exit 0
```

## while语句实例2

```
#!/bin/sh
echo "Enter password"
read pass
while [ "$pass" != "secret" ]; do
    echo "Sorry, try again"
    read pass
done
exit 0
```

# util实例


```
#!/bin/bash
echo -n "please input your name: "
read name
until ["${name}" = "cosmos" ] #如果名称不是cosmos, 则表达式
                                #返回为非0, 则继#续执行下列语句
do
    echo -n "the name you input is wrong,please input again: ";
    read name
done
echo "you have typed name:$name"
```



```
#!/bin/sh
for foo in bar fud 43
do
echo $foo
done
exit 0
```

```
#!/bin/sh
for file in $(ls f*.sh);do
    more $file | grep "abcd"
done
exit 0
```






# 逻辑运算

➤ && 逻辑与运算

➤ || 逻辑或运算



```
#!/bin/sh
touch file_one
rm -f file_two
if [ -f file_one ] && echo "hello" && [ -f file_two ]
    && echo "there"
then
echo "in if"
else
echo "in else"
fi
exit 0
```



hello  
in else



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

```
rm -f file_one
```

```
if [ -f file_one ] || echo "hello" || echo  
  "there"
```

```
then
```

```
echo "in if"
```

```
else
```

```
echo "in else"
```

```
fi
```

```
exit 0
```



hello  
in if



# 主要内容

- 3.1 Shell基础
- 3.2 Shell编程基础
- 3.3 Shell脚本中的特殊符号
- 3.4 Shell变量和表达式
- 3.5 Shell脚本控制流程
- 3.6 Shell脚本函数和数组
- 3.7 Bash调试

# 函数的基本用法

- (1)定义函数的语法如下：

[ function ] name () {函数体} [ 重定向 ]

其中function关键字及重定向命令是可选的

- (2)使用函数的方法：

funcname 参数列表。

函数名称加参数列表即可执行函数体中的命令。

函数通过位置参数\$1、\$2等访问传递给函数的参数，而\$0指的是函数名。

# 函数的基本用法

## ➤ (3) 函数返回值

函数通过return [n]语句返回值n。如果没有指定n，那么返回函数最后一条命令执行后所返回的状态。

## ➤ (4) 访问函数返回值

紧接着函数调用之后，通过\$?命令可访问函数返回值，注意，**\$?**与调用函数之间不能有其它语句。

# 函数的定义

```
function function_name {  
    command...
```

```
}
```


或

```
function_name () {  
    command...
```

```
}
```

- ➡ 函数被调用或被触发, 只需要简单地用函数名调用
- ➡ 函数定义必须在第一次调用前完成没有像 C 中的函数“声明”的方法。





```
#!/bin/bash
function max()
{
    if [ $# -ne 3 ];then
        echo "usage:max p1 p2 p3"
        exit 1
    fi
    max=$1
    if [ max -lt $2 ];then
        max=$2
    fi
    if [ max -lt $3 ];then
        max=$3
    fi
    return max
}
```




max 1 2 3

echo "the max number of 1 2 3 is : \$?"

exit

# 数组

- ➡ 数组元素可以用符号 `variable[xx]` 来初始化.
- ➡ 脚本可以用 `declare -a variable` 语句来清楚地指定一个数组.
- ➡ 要访问一个数组元素, 可以使用花括号来访问, 即 `${variable[xx]}`.




```
#!/bin/bash  
area[11]=23  
area[13]=37  
area[51]=UFOs
```

数组成员不必一定要连贯或连续的.

数组的一部分成员允许不被初始化.

数组中空缺元素是允许的.



```
echo -n "area[11] = "  
echo ${area[11]}    # {大括号}是需要的.  
echo -n "area[13] = "  
echo ${area[13]}  
echo "Contents of area[51] are ${area[51]}"
```



# 没有初始化内容的数组元素打印空值(NULL值).

```
echo -n "area[43] = "
```

```
echo ${area[43]}
```

```
echo "(area[43] unassigned)"
```

```
echo
```


# 两个数组元素和赋值给另一个数组元素

```
area[5]=`expr ${area[11]} + ${area[13]}`
```

```
echo "area[5] = area[11] + area[13]"
```

```
echo -n "area[5] = "
```

```
echo ${area[5]}
```



```
area[6]=`expr ${area[11]} + ${area[51]}`  
echo "area[6] = area[11] + area[51]"  
echo -n "area[6] = "  
echo ${area[6]}
```

# 这里会失败是因为整数和字符串相加是不允许的.



## 另一种指定数组元素的值的办法...

```
array_name=( XXX YYY ZZZ ... )
```

```
area2=( zero one two three four )
```

```
echo -n "area2[0] = "
```

```
echo ${area2[0]}
```

```
#数组下标从 0 开始计数
```


```
echo -n "area2[1] = "
```

```
echo ${area2[1]}
```

### 第三种指定数组元素值的 ...

`array_name=([xx]=XXX [yy]=YYY ...)`

```
area3=([17]=seventeen [24]=twenty-four)
echo -n "area3[17] = "
echo ${area3[17]}
echo -n "area3[24] = "
echo ${area3[24]}
exit 0
```



# 主要内容

- 3.1 Shell基础
- 3.2 Shell编程基础
- 3.3 Shell脚本中的特殊符号
- 3.4 Shell变量和表达式
- 3.5 Shell脚本控制流程
- 3.6 Shell脚本函数和数组
- 3.7 Bash调试

# 调试

- Bash shell 没有自带调试器, 甚至没有任何调试类型的命令或结构.
- 脚本里的语法错误或拼写错误会产生含糊的错误信息, 通常这些在调试非功能性的脚本时没有什么帮助。

# 错误的脚本

```
#!/bin/bash
# 这是一个错误的脚本.
# 哪里有错?
a=37
if [ $a -gt 27 ]
then
    echo $a
fi
exit 0
```

脚本：

```
./ex74.sh: [37:
command not found
上面的脚本有什么错误 (线索: 注意 if 的后面)?
```

空格

丢失关键字(keyword)

1 #!/bin/bash

2 # error.sh: 会产生什么样的错误信息?

3

4 for a in 1 2 3

5 do

6 echo "\$a"

7 # done # 第7 行的必需关键字 'done' 被注释掉.

8

9 exit 0

脚本：

error.sh: line 10: syntax error: unexpected end of file

# 用echo语句找错误位置

- 注意错误信息中说明的错误行不必一定要参考, 但那行是 bash 解释器最终认识到是个错误的地方.

- echo 语句

可用在脚本中有疑问的地方以跟踪变量的值。最好只在调试时才使用 echo 语句.

# 设置sh的调试选项

- 设置选项 -n -v -x
- sh -n scriptname 不会实际运行脚本,而只是检查脚本的语法错误,该方法不能检测所有的语法错误。
- sh -v scriptname 在实际执行一个命令前打印出这个命令。
- sh -x scriptname 打印每个命令的执行结果,但只用在某些小的方面。
- 使用一个“assert”(断言)函数在脚本中。
- 捕捉 exit.




## trap命令

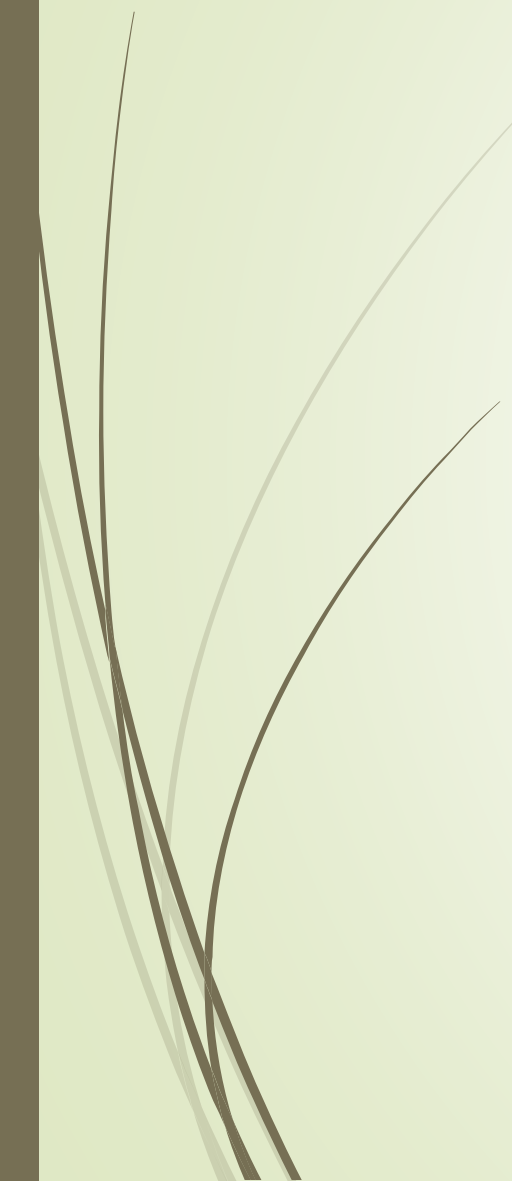

- 脚本的exit 命令会触发信号 0,终结进程,即脚本。
- 这常用来捕捉exit 命令做某事,如强制打印变量值.
- trap 命令必须是脚本中的第一个命令。

# trap命令

- 当收到一个信号时指定一个处理动作; 这在调试时也很有用.
- 信号是发往一个进程的非常简单的信息, 由内核或者由另一个进程发出, 以告诉接收进程采取一些指定的动作 (一般是中断)。例如, 按 Control-C, 发送一个用户中断( 即 INT 信号)到运行中的进程.
- trap的用法: `trap command signal` 表示接收到 signal信号后, 执行command命令



```
#!/bin/sh
trap 'rm -f /tmp/my_tmp_file_$$' INT
echo creating file /tmp/my_tmp_file_$$
date > /tmp/my_tmp_file_$$
echo "press interrupt (CTRL-C) to interrupt ...."
while [ -f /tmp/my_tmp_file_$$ ]; do
echo File exists
sleep 1
done
echo The file no longer exists
```



```
#!/bin/bash
trap "echo a=$a b=$b" EXIT
#EXIT信号是程序执
# 行exit命令时产生的信号
a=20
b=40
exit
```

# trap所能捕获的常用信号

➤ Signal	Description
➤ HUP	挂起
➤ INT	Ctrl+C引发的中断
➤ QUIT	Ctrl+\引发的退出
➤ ABRT	严重执行错误引发的中止
➤ ALRM	定时处理的报警信号
➤ TERM	终止，系统关机时发送

# 思考题

- 简述shell脚本中三个有特殊作用的字符.
- Shell脚本的变量有哪几类, 各是什么情况下使用?
- Shell脚本分支语句有几类, 各是什么情况下使用?
- Shell脚本循环语句有几类, 各是什么情况下使用?
- 简述shell脚本的调试方法。