Bash 环境及脚本编程

南开大学 物理学院生物物理系
吴爱平 M020120

声 明

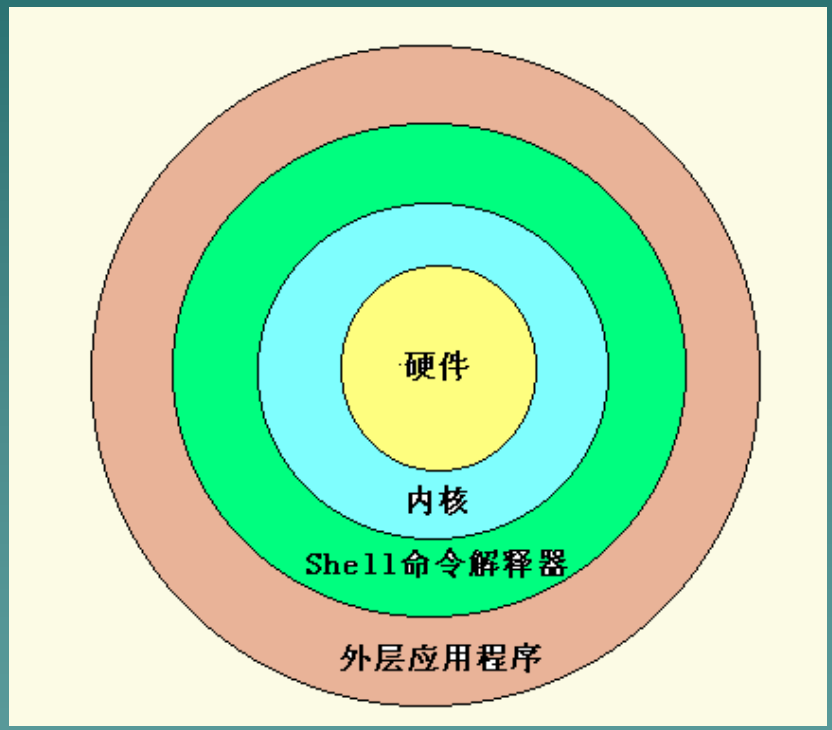
脚本语法部分材料来自清华大学计算机与信息管理中心，版权归其所有！

目录

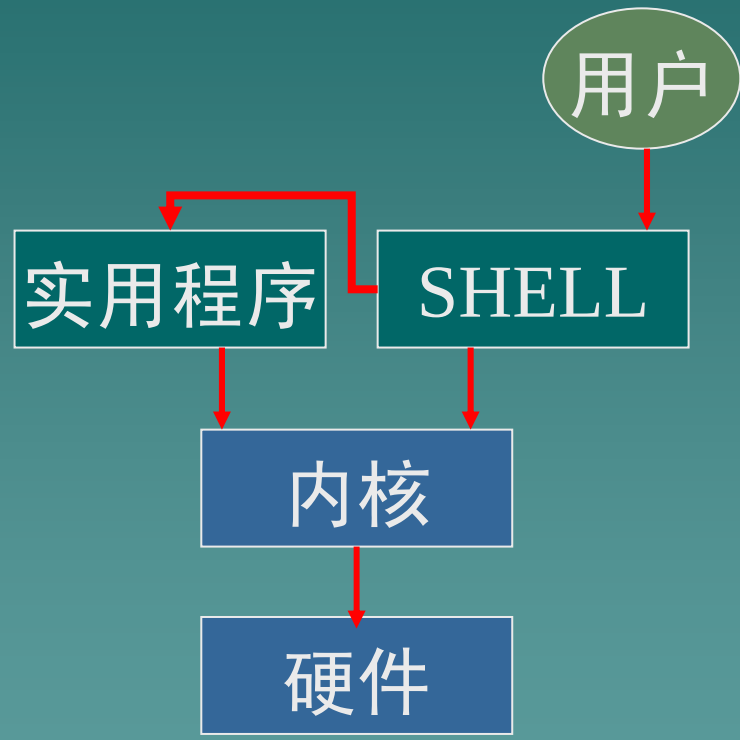
- Linux shell 概述
- Shell 脚本的执行分析
- 脚本语法细致分析
  - Shell命令总括
  - Shell命令的集成
  - Shell变量
  - Shell程序的控制结构
- 示例脚本深入分析

Shell 概述

Shell定位



a.有人认为



b.个人倾向

Shell是什么

- 字面理解：shell 是 “壳”，linux 是 kernel,是“核”
- Shell内涵：是一个特殊的应用程序，所以可以替换
  - Shell是用户一登录就开始运行的实用程序，即一登录即被调入内存；
  - 允许用户通过命令行或脚本的方式输入命令，并通过翻译解释这些命令完成用户与kernel 的交互

为什么要学习shell

Shell的使用：

- 翻译提示符后面输入的命令
- 设置shell初始化文件，使用户的工作环境个性化
- 作为解释型的程序语言

学习shell的什么

- Shell命令
- Shell的命令集成
- Shell程序的控制结构
- Shell的配置文件
- Shell用户环境定制
- Shell命令解析机制
- “freedom”的脚本使用

怎么学习shell

- 先易后难
- 先难后易

Shell的启动

BASH的启动次序

- 引导系统时，第一个运行的进程init
- 衍生出一个getty终端，该过程打开一个终端端口，提供一块空间给标准输入、标准输出和标准错误，把提示符显示在屏幕上
- 执行程序/bin/login，提示输入密码，加密并验证密码，建立一个初始环境
- 启动shell, /etc/passwd文件中/bin/bash
- BASH调用各种初始化文件，设置用户环境，出现

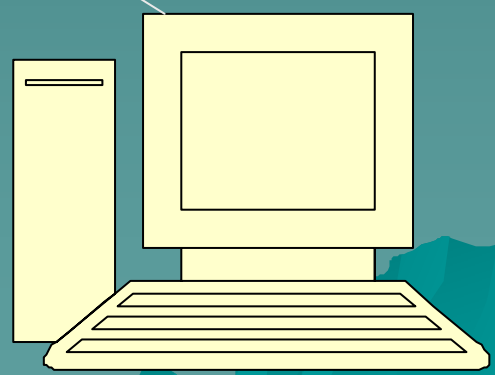
  [root @ nkstar2 root]$

init

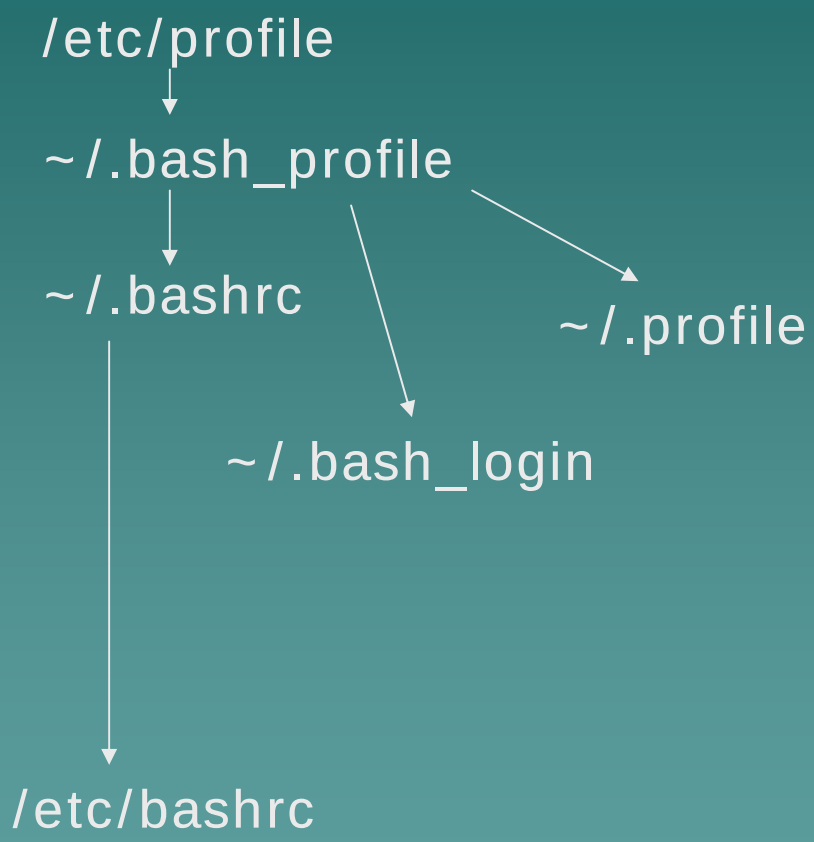
getty

login

Bash



BASH的初始化文件

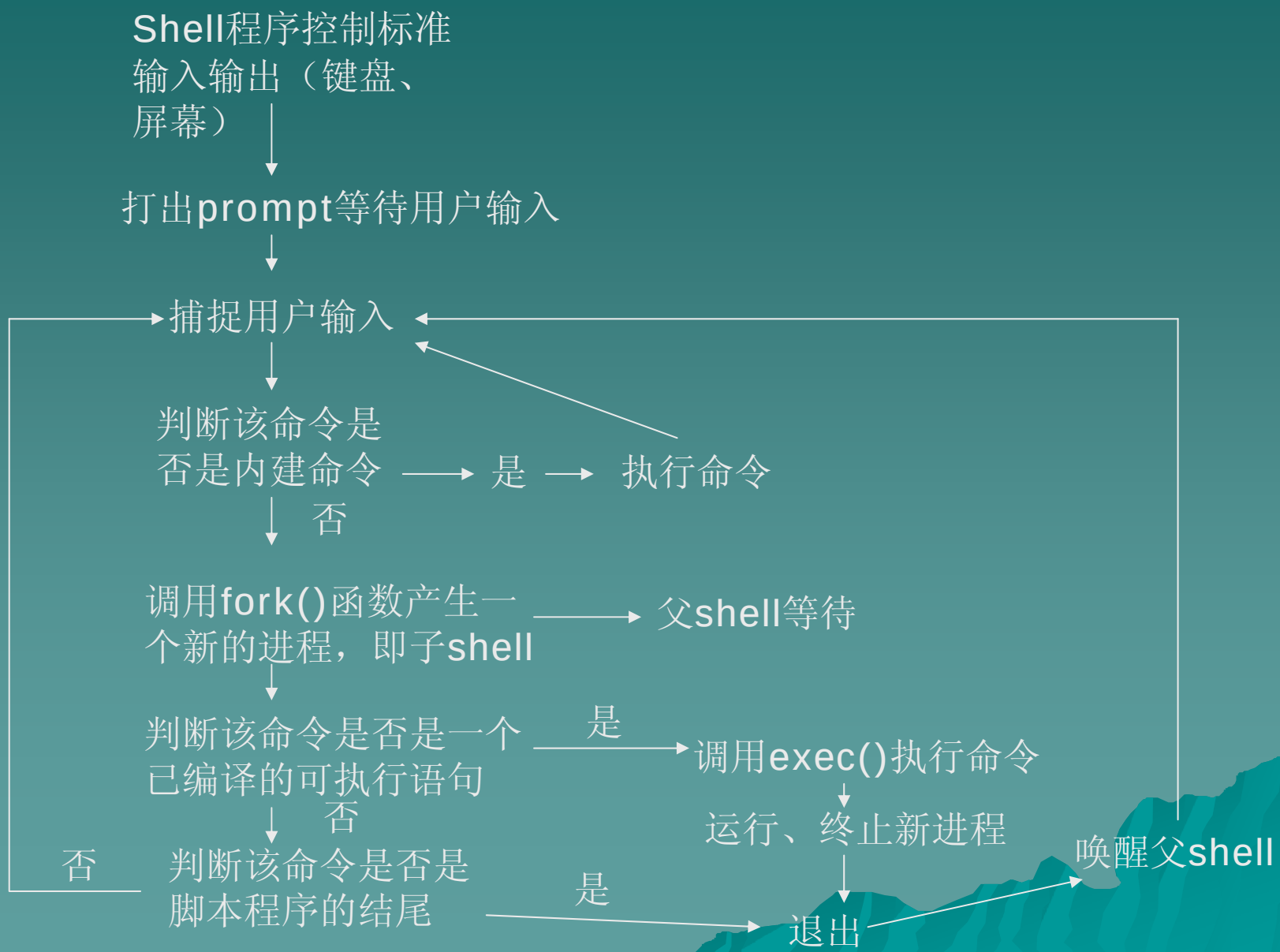

/etc/profile
~/.bash_profile
~/.bashrc
~/.profile
~/.bash_login
/etc/bashrc

环境配置文件查看

- /etc/profile
- ~/.bash_profile
- ~/.bashrc
- /etc/bashrc
- /etc/inputre：包含定义键击行为和设置的命令
- ~/.bash_history
- ~/.bash_logout
- ...

Shell命令及脚本的执行

Shell 命令的执行机制



Shell执行的环境与继承

- 登录系统时，shell启动并从启动它的/bin/login程序中继承多个变量、I/O流和进程特征
- Shell调用相应的初始化文件，对shell工作环境进行特定的设定
- 启动子shell,子shell从父shell处继承环境，包括进程的权限、工作目录、文件创建掩码、特殊变量、打开的文件和信号等

  为什么启动子shell?

  - 可能启动子shell的原因：
    - 后台处理
    - 处理整组的命令
    - 执行脚本
    - 。。。

执行shell命令的方法

- 在当前shell执行脚本
  - . test.bash
  - source test.bash
  - { test.bash; }
  - eval '. test.bash'
- 生成新shell即子shell再执行脚本
  - test.bash
  - /bin/bash test.bash
  - (. test.bash;)
  - exec test.bash

脚本语法细致分析

- shell命令总括
- shell命令组合集成
- shell变量
- shell程序控制结构
- shell程序调试

Shell命令总括

命令格式

一般格式：command [options] [arguments] [filenames]

- options 或 switches（常带有一个连字号“-”）：以此扩展命令的特性或功能
- Arguments：表示命令的自变量（参数）
- filenames：表示文件或目录的名字。

命令类型及执行顺序

- 别名：现存命令的缩写，可用alias查看
- 关键字：shell内部预先定义的一组特殊字符，如if, function, while及 until等
- 函数：由一组组织在一起的命令组成的一个例程
- 内建命令：已写入shell内部程序的例程，使用type查询，help帮助
- 可执行程序：

注意：别名和函数定义在shell内存里，内建命令是shell内部的例程，而可执行程序则储存在磁盘上。执行可执行程序时，shell搜索命令所在目录

Shell内部命令

出于运行效率的考虑，将一些命令构造在Shell的内部。这些命令比非内部命令执行速度快。

- 关键字（用于结构分支及控制）：

  if else for case while until continue break

- 内建命令：

: . alias bg bind builtin cd declare dirs disown echo enable eval exec exit export fc fg getopts hash help history jobs kill let local logout popd pushd pwd read readonly return set shift stop suspend test times trap typest ultimit umask unalias unset wait

Shell内常用命令

Shell命令大致划分为以下几类：

- 目录操作与管理
- 文件操作与管理命令
- 系统管理与维护
- 用户管理与维护
- 系统状态
- 进程管理
- 通讯命令
- 其它命令

Shell命令的组合集成

元字符和文件名生成

元字符（通配符）的定义

- *****　配任何字符串，包括空字符串
- **?**　匹配任何单个字符
- **[… , - , !]**　按照范围“-”、列表“…”或不匹配“!”等形式匹配指定的字符
- ****　转意符，使元字符失去其特殊的含义

例：**[a-d，x，y]**　匹配字符a、b、c、d、x、y；

z*　匹配以字符z开始的任何字符串；

x?y　匹配以x开始、以y结束、中间为任何单个字符的字符串；

[!Z]　匹配不为Z的单个字符。

元字符和文件名生成（续）

元字符作为文件扩展名的使用

例：**[a-f]*** 匹配字符a到字符f开头的文件名；
abc　d2　e3.c　f.dat

***z**　匹配以字符z结尾的任何字符串；
win.z　core.zz　a-c_5z

rc?.d　匹配以rc开始、以.d结束、中间为任何单个字符的文件名；
rc0.d　rc2.d　rcS.d

***[!o]**　匹配不以o结尾的文件名；

管道和命令表

- 管道：是一个命令的标准输出与另一个命令的标准输入之间的连接，不经过任何中间文件
- 管道线：是由管道操作符分隔的一个命令序列，最简单的管道线是一个简单命令
- 管道操作符：用符号“|”表示

例： who | wc -l
ps -ef | grep ftp
ls -l

管道和命令表（续1）

- 命令表：一串管道线（命令）构成了一个命令表，最简单的命令表是一个管道线（即一个简单命令）
- 管道线（命令）分隔符：分隔命令表元素，确定管道线执行的条件包括：
  - ；或 换行符 ：表示按顺序执行管道线
  - **&&** ： 表示根据条件（true），执行其后面的管道线
  - **||** ： 表示根据条件（false），执行其后面的管道线
  - **&** ： 表示前面的管道线在后台（异步）执行。

管道和命令表（续2）

例1：四个管道线构成一个命令表

```
ls  -l  /bin  /usr/bin
who | wc -l
a | b | c | d
ps
```

例2：与例1等价

```
ls -l /bin /usr/bin ； who | wc -l ； a | b | c | d； ps
```

例3： vi wuaiping_test &

例4：查询指定的文件是否存在，给出相应信息

```
test  -f  $1 &&  echo “$1 exists”
test  -f  $1  | |  echo “$1 not exists”
```

命令组合

- { 命令表 }

  由当前Shell来执行命令表

  例：{ cd mydoc ; rm junk; }

- (命令表)

  当前Shell派生一个子Shell进程来执行命令表

  例：（ cd mydoc ; rm junk; ）

命令替换

用命令的执行结果来替换这个字符串本身

例1： # now=‘date’ ⇦ 单引号

```
# echo $now

# now=`date`          ⇦ 单撇号
# echo $now
```

例2： # count=10

```
# count=`expr $count + 1`
# echo $count
11
```

注意：反撇号与单引号的区别

输入、输出重定向

- 使用标准改向符进行重定向(改向)

<	输入改向
<<	输入改向（here 文件）
>	输出改向
>>	追加输出改向

- 使用标准文件描述字进行重定向(改向)

使用文件描述字来定义输入、输出的标准文件，其中：

0：标准输入　1：标准输出　2：标准错误输出

输入、输出重定向（续1）

标准输入的改向（< 、 <<）

command < file

command << End Of Marker （结束标志符）

> .
> .
> .
> End Of Marker

例1：sort < myfile

例2：cat << !eof
> Hellow !
> ok !!
>!eof

输入、输出重定向（续2）

标准输出的改向（>、 >>）

- command > file
- command >> file

例1：date > myfile

例2：ls -l >> myfile

例3：cat file* > myfile

例4：ps -ef | grep ftp > myfile

输入、输出重定向(续3)

☞ 标准错误输出的改向（>、 >>）

- command 2 >file
- command 2 >>file

例1：将错误输出改向到err_file文件

```
myprog 2 > err_file
```

例2：将标准输出和错误输出改向out文件

```
myprog >out 2 >>out   (两种方法等价)
myprog >out 2 >>&1  (改向处理从左至右)
```

例3：错误输出被显示，标准输出改向out文件。

```
myprog 2 >&1 >> out
```

Shell 变量

Shell变量

- Shell变量
- 用户自定义变量
- 位置变量
- 环境变量
- 预定义的特殊变量
- 变量替换
- 特殊字符的引用

Shell变量概述

- Shell实际上是基于字符串的程序设计语言，也具有变量。变量的名字必须以字母或下划线开头，可以包括字母、数字和下划线
- Shell变量能够而且只能存储正文字符串，即它只有一种类型的变量——串变量
- 从赋值的形式上看，则可以分成四种类型的变量或变量形式
  - 用户自定义变量
  - 位置变量
  - 环境变量
  - 预定义的特殊变量

Shell变量概述（全局变量）

全局变量是一种特殊的变量，可以被任何运行的子Shell来引用。全局变量通过**export**命令来定义，格式如下：

export variables

其中 variables 是要变成全局变量的变量表名

- 一旦变量被定义为全局变量，则对于以后的所有子Shell来说这些都是全局变量
- 子Shell中无法改变全局变量的值
- 若在子Shell中改变全局变量的值，实际是对全局变量的副本进行更改，不影响全局变量值
- 子Shell中局部变量的使用优先于全局变量

Shell变量概述（局部变量）

在某一局部特定环境下使用的变量

- 注册Shell在接受到用户输入的命令（非内部命令）后，通常派生出一个子Shell，由此子Shell负责解释执行该命令
- 子Shell有自己的运行环境和变量，这些变量仅在子Shell的范围内的特定环境下才能使用
- 子Shell不能存取由父Shell设置的局部变量，也不能改变父Shell的变量值

用户自定义变量

- 语法格式：**name=string**
  赋值号“=”两边不允许有空白符；
  nodehost="beijing.UUCP ”
  path=/bin:/usr/bin:/etc/bin
  count=10

- 允许多个赋值操作，按从右到左的顺序进行
  # A=$B B=abc C="OK"
  # echo $A $B $C
  abc abc OK

- 当引用一个未设置的变量时，其隐含值为空
  # echo "$mail is path of mailbox”
  is path of mailbox

用户自定义变量（双引号）

☞ 如果用双引号“”将值括起来，则括起来的字符串允许出现空格、制表符和换行符的特殊字符，而且允许有变量替换

例1：
```
# MAIL=/var/mail/fk
# var="$MAIL is a path of mailbox"
# echo $var
/var/mail/fk is a path of mailbox
```

例2：
```
# str="This is \n a book"
# echo $str
This is
 a book
```

用户自定义变量（单引号）

- 如果用单引号‘’将值括起来，则括起来的字符串允许出现空格、制表符和换行符的特殊字符，但不允许有变量替换

例1
```
# BOOK="English book"
# MSG='$BOOK'
# echo $MSG
$BOOK
```

例2
```
# msg=' Today is \t Sunday'
# echo $msg
Today is \t Sunday
```

局部变量和全局变量作用域

任何没有用export命令定义过的变量是局部变量，子Shell不能存取父Shell的局部变量

- 子Shell中可以存取和修改父Shell的全局变量，但这种修改对于父Shell全局变量没有任何影响
- 在子Shell中用export命令定义的全局变量和对此变量的修改对父Shell变量没有影响
- 全局变量保持它的全局性，不仅能直接传递给它的子Shell，而且子Shell还能将它传递给子Shell的子Shell
- 在对变量赋值之前和之后的任何时候可以将该变量转换成全局变量

用户自定义变量（大括号）

- 引用变量的值时，可以用花括号{}将变量名称括起来，使变量名称与它的后续字符分隔开，如果紧跟在变量名称后面的字符是字母、数字或下划线时，必须要使用花括号

例： # str='This is a string'

echo "${str}ent test of variables"

This is a stringent test of variables

echo "$strent test of variables"

test of variables

位置变量

位置变量顾名思义是与位置有关的变量，这是一种特殊的变量。命令行的Shell过程名本身被指定为位置变量$0，依次参数为$1 …. $9

例：ls　/　/bin　/etc　/usr/bin　/dev　...

$0　$1　$2　$3　$4　$5　...

例：# cat finduser

who | grep $1

finduser fke

Shift命令(处理不定长参数)

- 使用shift可以将命令行位置参数依次移动位置， 即$2->$1, $3->$2. 在移位之前的第一个位置参数$1在移位后将不在存在
- 使用shift时，每进行一次移位，$#减1，使用这一特性可以用until循环对每个参 数进行处理
- 使用shift的另一个原因是Bourne Shell的位置参数变量为$1~$9, 因此通过位置变 量 只能访问前9个参数。但这并不等于在命令行上最多只能输入9个参数。此时如果想 访问 前9个参数之后的参数，就必须使用shift
- shift后可加整数进行一次多个移位，如：shift 3

环境变量

Shell执行环境由一系列环境变量组成，这些变量是由Shell维护和管理的。所有这些变量都可被用户重新设置，变量名由大写字母或数字组成。

CDPATH：执行cd命令时使用的搜索路径；

HOME ：用户的home目录

IFS ：内部的域分隔符，一般为空格符、制表符或换行符

MAIL ：指定特定文件(信箱)的路径，供邮件系统用

PATH ：寻找命令或可执行文件的搜索路径

PS1 ：主命令提示符，默认为“$”

PS2 ：从命令提示符，默认为“>”

TERM ：使用的终端类型

预定义的特殊变量

在Shell中有一组特殊的变量，其变量名和变量值只有Shell本身才可以设置

☞ **$#** — 记录传递给Shell的自变量个数

例1： # myprog a b c　　则 $#的值为3

例2：
```
if test $# -lt 2
   then
     echo “two or more args required”
     exit
fi
```

预定义的特殊变量（续1）

- **$?** — 取最近一次命令执行后的退出状态(返回码)：执行成功返回码为0， 执行失败返回码为1；

  例：# test -r my_file (假设my_file文件不可读)
  # echo $?
  1

- **$$** — 当前Shell的进程号

- **$!** — 取最后一个在后台运行的(使用“&”)， 进程的进程号

预定义的特殊变量（续2）

- **$*** — 匹配所有位置变量
  - $* 匹配 $1 $2 $3 …
  - “$*” 匹配 “$1 $2 $3 …”
- **$@** — 匹配所有位置变量
  - $@ 匹配 $1 $2 $3 …
  - “$@” 匹配 “$1” “ $2” “$3 ” …
- **$-** — Shell的标志位，既在Shell启动时使用的选项，或用set命令方式所提供的选项

变量替换

Shell在遇到未设置的变量时，将其值作为空串处理。而在实际应用中，对于未设置的变量，用户可以根据需要采用不同的处理方式，这可通过变量替换来实现

变量替换提供了三种功能：

- 允许替换未设置变量的隐含值
- 允许对未设置变量赋值
- 在访问未设置变量时，提示出错信息

变量替换格式

- ${var:-value} 如果var有值了那麼就用原本的值，不然用value的值
- ${var:+value} 如果var有值也用value的值
- ${var:?message} var有值那麼就用原本的值，不然就印出 message 值到荧幕並且跳出。
- ${var:%pattern} 如果pattern与var后面的部份吻合，传回剩下没有 吻合部份給var
- ${var:#pattern} 如果pattern与var前面的部份吻合，传回剩下没有 吻合部份給var
- ${var/pattern/substitute} 如果var有pattern吻合就代换成 substitute

变量的替换（示例）

例1：${var:-word} （假设$ PARM未设置）

```
# echo  " The value of  PARM is ${PARM:-undefined}"
The value of PARM is undefined
# echo $PARM
$ –
```

[注意] 此处的“ – ”表示空字符

例2：${var:-word} （假设$ PARM未设置）

```
# arg=${PARM:-“not defined”}
# echo ' $arg : '$arg

$arg : not defined
```

[注意] 双引号

特殊字符的引用

消除特殊字符的含义可用转义符、单引号和双引号

- 转义符（ \） 的引用

  消除紧跟在转义符后面的单个字符的特殊含义 。

  例　#count=`expr count \* 10`

- 单引号 （‘’）的引用

  消除被括在单引号中的所有特殊字符的含义。

  例：# echo ‘$count=’$count

- 双引号 （“”）的引用

  双引号能消除被括在双引号中大部分特殊字符的含义，

  但不能消除 $ 、 ` 、 " 、 \ 四个字符的其特殊含义：

  例： vdate=“`date` is system maintenance day !”

特殊字符的引用（逃逸字符）

☞ 特殊字符串引用的例外

引用双引号、单引号和转意符都不能消除对echo命令有特殊功能的控制字串（逃逸字符）的特殊含义。这些控制字串是：

\b Backspace
\c 显示后不换行
\f 在终端上屏幕的开始处显示
\n 换行
\r 回车
\t 制表符
\v 垂直制表符
\ 反斜框

Shell程序的控制结构

Shell的控制结构

- 条件和 **test** 命令
- **if** 结构
- **case** 结构
- **for** 结构
- **while** 结构
- **until** 结构
- 循环体中的其它命令

条件与 test 命令

- 简单条件

在高级语言中判断条件依赖于运算的结果，而Shell语言依赖条件是命令执行的“出口状态”

Shell命令的“出口状态”（$?）：

成功：0 、 true

失败：x 、 false (x 为非0数值)

例：判断指定目录是否存在， 并显示相应信息。

```
#  cat  checkdir
test -d $1 && echo “$1 is a dictory”&& exit 0
echo “$1 is not a dictroy”
exit 1
```

条件与 test 命令（续1）

- **test 命令**

  test 命令可用于对字符串、整数及文件进行各类测试

其命令格式如下：

```
test  expression
```

或 `[ expression ]`　　（注意 [ ] 中的空格）

expression 是测试的条件，计算结果：

为真，则返回“零”出口状态，

为假，否则返回“非零” 出口状态

例：判断当前上机用户人数是否多于10 ？

```
# test "`who | wc -l ` " -gt 10
# echo $?
```

test 字符串测试表达式

expression	满足下列条件时返回真值
string1 = string2	string1与string2相同
string1 != string2	string1与string2不相同
string	string不为空串
-n string	string不为空串
-z string	string为空串

字符串测试示例

例1：两个字符串进行比较

```
# user=smith
# test "$user"   =  smith
# echo $?
0
```

例2：查找指定的文件或目录

```
  # cat search
   test "$1" || { echo "err: no parameter" ;
\
                    exit 1; }
   find . -name "$1" -print
```

test 命令用于整数比较

首先要搞清楚整数比较的两个概念：

- Shell并不区分放在Shell变量中的值的类型，就变量本身而言，它存放的仅仅是一组字符串，既Shell只有一种类型的变量——串变量
- 当使用整数比较操作符时，是test命令来解释存放在变量中的整数值，而不是Shell。

test 整数测试表达式

expression	满足下列条件时返回真值
int1 -eq int2	两者为数值且int1等于int2
int1 -ge int2	两者为数值且int1大于或等于int2
int1 -gt int2	两者为数值且int1大于int2
int1 -le int2	两者为数值且int1小于或等于int2
int1 -lt int2	两者为数值且int1小于int2
int1 -ne int2	两者为数值且int1不等于int2

test 中常用的文件测试表达式

expression	满足下列条件时返回真值
-r FileName	FileName存在且为用户可读
-w FileName	FileName存在且为用户可写
-x FileName	FileName存在且为用户可执行
-s FileName	FileName存在且其长度大于0
-d FileName	FileName为一个目录
-f FileName	FileName为一个普通文件

test 文件测试示例

例1：检查指定的文件是否存在并且可读

```
test -f /usr/fk/message
```

例2：检查指定的文件是否为目录

```
test -d /usr/src/local/sendmail
```

例3：检查指定的出错文件是否为空，如不空则列出该文件的内容。

```
test -s $errfile && cat $errfile
```

条件与 test 命令（表达式的逻辑运算）

逻辑运算符包括：

! — 逻辑非单目运算符，可放置在任何其它test表达式之前，求得表达式运算结果得非值

-a — 逻辑与运算符，执行两个表达式的逻辑与运算，并且仅当两者都为真时，才返回真值

-o — 逻辑或运算符，执行两个表达式的逻辑或运算，并仅当两者之一为真时，就返回真值

条件与 test 命令（逻辑运算符的优先级）

逻辑运算符优先级(由高到低) 的排列顺序如下：

（ ） 🡅 ! 🡅 -a 🡅 -o

逻辑运算符优先级要比字符串操作符、数字比较操作符、文件操作符的优先级低。

条件与 test 命令（表达式的逻辑组合）

expression	满足下列条件时返回真值
! expr	expr返回值为假(Not)
expr1 –a expr2	expr1和expr2同时为真 (And)
expr1 –o expr2	expr1为真或expr2为真 (Or)
\(expr\)	expr为真时 [注意]左右括号前要加转义符 \

表达式的逻辑组合示例

例1：当指定的文件不可读时为真

```
test ! -r /usr/fk/message
```

例2：当指定的文件均存在，且message为可读、 $mailfile 指定的文件为普通文件时，返回真

```
test  -r /usr/fk/message  -a  -f  "$mailfile "
```

例3：当变量值大于等于0并且小于10时为真

```
test  "$count "  -ge 0  -a  "$count "  -lt 10
```

例4：

```
test  \( “$a” -eq 0  -o  "$b"  -gt 5 \)  -a  "$c"  -le 8
```

if 结构

if 的简单结构

格式

```
if command1
 then
  command2
  command3
  ...
fi
```

if 结构（续1）

if 的完整结构

格式

```
if command1
 then
   command2
   command3
   ...
 else
   command4
   command5
   ...
fi
```

if 的连用结构1

```
if command1
   then
      commands
   else
      if command2
      then
       commands
              :
          :
      fi
  fi
```

if 的连用结构2

格式2

```
if command1
then
   commands
elif command2
then
   commands
        :
          :
   commands
else
   commands
fi
```

case 结构

格式

```
case value in
        pattern1)   command11
                    …
                    command1n;;
        pattern2)   command21
                    …
                    command2n;;
          …           …
        patternn)   commandn1
                    …
                    commandnn;;
esac
```

for 结构

格式

```
for  variable in arg1 arg2 ... argn
do
    command
        ...
    command
done
```

while 结构

格式

```
while command
do
    command
        ...
    command
done
```

until 结构

格式

```
until command
do
    command
        …
    command
done
```

循环体中其它命令

- **break** 命令

  break是Shell的内部命令，用于在循环体中根据命令运行的返回条件，直接终止循环体内命令的执行。当执行**break**命令时，控制流从循环体中转移到**done**之后的第一条命令上。

- **continue** 命令

  continue是Shell的内部命令，用于在循环体中根据命令运行的返回条件，直接进入下一次循环命令的执行。 当执行**continue**命令时，控制流直接转到本循环体中第一条命令上。

函数的定义和使用

函数格式

```
FunctionName( ){
        command
            :
            :
        command
    }
```

函数的定义和使用（续）

例：
```
### The test codes for function definition
GetYesOrNo() {
  while echo "$*(Y/N)? \c" > &2
    do
      read reply RestData
      case  "$reply"in
         [yY] )  return 0 ;;
         [nN] )    return 1 ;;
         *     )    echo "Please enter Y or N !">&2 ;;
      esac
    done
```
执行 `# GetYesOrNo "Do you wish to continue" || exit`

显示 `Do you wish to continue(Y/ N)?`

Shell程序调试

程序调试

Shell提供了多种工具以便在调试Shell程序时使用，这些工具允许观察一个Shell程序的执行

常用的测试方式有：

Shell程序的详细跟踪；

Shell程序的跟踪执行；

Shell程序的详细跟踪

Shell提供的详细跟踪特性允许用户观察一个Shell程序的读入和执行，如果在读入命令行时发现语法错误，则终止程序的执。

命令行被读入后，Shell按读入时的形式在标准错误输出中显示该命令行，然后执行命令行。详细跟踪Shell程序的执行有两种方式：

- 整个程序的详细跟踪
- 局部程序的详细跟踪

Shell程序的详细跟踪（续）

- 整个程序的跟踪执行

  格式：**sh -v shprog**

  用来实现对整个文件的脚本进行跟踪

- 局部程序的跟踪执行

  格式：

  set -v —— 设置跟踪标志

  set +v —— 关闭跟踪标志

  用来实现对文件中的部分脚本进行跟踪

Shell程序的跟踪执行

Shell的跟踪执行功能允许用户观察一个Shell程序的执行，它使命令行在执行前完成所有替换之后，在标准错误输出中显示每一个被替换后的命令行，并且在行前加上前缀符号“+”（但变量赋值语句不加“+”符号），然后执行命令

同详细跟踪一样，对Shell程序的跟踪执行也有两种方式：

- 整个程序的跟踪执行
- 局部程序的跟踪执行。

Shell程序的跟踪执行（续）

- 整个程序的跟踪执行

  格式：**sh -x shprog**
  用来实现对整个文件脚本的跟踪执行

- 局部程序的跟踪执行
  格式：
  set -x —— 设置跟踪标志
  set +x —— 关闭跟踪标志
  用来实现对文件中部分脚本的跟踪执行

详细跟踪与跟踪执行的组合

- 整个程序的跟踪执行

  格式：**sh -vx shprog**

- 局部程序的跟踪执行

  格式

  set -vx —— 设置跟踪标志

  set +vx —— 关闭跟踪标志

Shell程序的应用

何时使用Shell程序设计语言

- 当一个问题的解决方法包含了许多UNIX系统的标准命令操作时，可使用Shell程序设计语言
- 如果一个问题能用在UNIX系统中已建立的基本操作所表示，则使用Shell程序设计语言能构成更强的功能
- 如果处理问题的基本数据是正文行或文件，则Shell可描述一个很好的解决方法；若基本数据是数字或字符，则使用Shell可能不是好办法
- 使用Shell程序的最后一个准则是程序的开发成本。

Shell 脚本深入分析

该下课了！该下课了！！