第4章
bash及脚本程序设计

主要内容(一)

主要内容(二)

4.1 shell的基本机制

shell概述

Unix的shell

shell种类

- B-shell：由Stephen R. Bourne(1944-)在贝尔实验室开发，是最早被普遍认可的shell，**早期**UNIX的标准shell， /bin/sh,
- C-shell: /bin/csh 由加利福尼亚大学的William N. Joy(也叫Bill Joy)在20世纪70年代开发，最初用在BSD2.0。Joy在1982年与他人共同创办了Sun Microsystems
- K-shell： Korn shell，/bin/ksh 贝尔实验室的David Korn在1986年开发。是B-shell的超集，支持带类型的变量，数组
- /bin/bash　Bourne Again shell，是Linux上的标准shell， 兼容Bourne Shell，扩展了B-shell，吸收了C shell的某些特点，交互式使用时命令行编辑非常方便
- 管理员在创建用户时，设置了用户的登录shell

Stephen R. Bourne (2005)

Shell的功能

- **shell是命令解释器**
- **文件名替换，命令替换，变量替换**
- **历史替换，别名替换**
- **流程控制的内部命令 (内部命令和外部命令)**

Shell的特点

- 主要用途：批处理，执行**效率**比算法语言低
- shell**编程风格**和C语言等算法语言的区别
- shell是**面向命令处理的语言**，提供的流程控制结构通过对一些**内部命令**的**解释**实现
- 如同C语言设计思路一样，shell本身设计得非常精炼，但是它提供了灵活的机制（策略与机制相分离）
  - shell许多灵活的功能，通过**shell替换**实现
  - 例如：流程控制所需的条件判断，四则运算，都由shell之外的命令完成

理解Unix的shell

- **学习bash的目的**
  - **交互方式**下：熟习shell的替换机制、转义机制，掌握循环等流程控制，可以编写复合命令
  - **非交互方式**：编写shell脚本程序，把一系列的操作，编纂成一个脚本文件，批量处理

bash的启动

启动交互式bash

- 三种启动方法
  - 注册shell:**注册shell**
  - 键入bash命令:**交互式shell**
  - **脚本解释器**
- 自动执行的一批命令（用户偏好）
  - 当bash作为注册shell被启动时:自动执行用户主目录下的.bash_profile文件中命令，~/.bash_profile或$HOME/.bash_profile
    - **当bash作为注册shell退出时: 自动执行$HOME/.bash_logout**
    - **当bash作为交互式shell启动时: 自动执行$HOME/.bashrc**
  - 类似umask之类的命令，应当写在.profile文件中

启动交互式bash

- 自动执行的一批命令（系统级）
  - 当bash作为注册shell被启动时:自动执行/etc/profile文件中命令
  - 当bash作为交互式shell启动时: 自动执行/etc/bash.bashrc
  - 当bash作为注册shell退出时:自动执行/etc/bash.bash.logout

脚本文件

- 编辑文件lsdir

(格式为文本文件，文件名不必须为.sh后缀，只是个惯例)

```
if [ $# = 0 ]
then
    dir=.
else
    dir=$1
fi
find $dir -type d -print
echo '-----------------'
cd $dir
pwd
```

脚本文件的执行

- **新创建子进程，并在子进程中执行脚本**
  - bash<lsdir
    无法携带命令行参数
  - bash lsdir
    bash -x lsdir
    bash lsdir **/usr/lib/gcc**
  - 给文件设置可执行属性x：chmod u+x lsdir
    然后执行 ./lsdir /usr/lib/gcc

  三种方法均启动程序/bin/bash，生成新进程
- **在当前shell进程中执行脚本**

  . lsdir /usr/lib/gcc

  source lsdir /usr/lib/gcc

历史与别名

历史表

- **历史表大小**
  - 先前键入的命令存于历史表，编号递增，FIFO刷新
  - 表大小由变量HISTSIZE设定

  修改HISTSIZE的配置应放入~/.bashrc
- **查看历史表**
  - 内部命令history

  （文件$HOME/.bash_history）

```
$ history
4916  2018-12-02 23:06:31 vi bst.c
4917  2018-12-02 23:06:34 make bst
4918  2018-12-02 23:06:35 ./bst < bst.txt
4919  2018-12-02 23:06:48 vi bst2.txt
4920  2018-12-02 23:07:01 ./bst < bst2.txt
4921  2018-12-02 23:07:06 make bst
4922  2018-12-02 23:07:14 cp bst* ~stud
4923  2018-12-02 23:07:16 sz bst*
4924  2018-12-02 23:08:46 ./bst < bst.txt
4925  2018-12-02 23:11:31 pwd
4926  2018-12-02 23:11:32 cd
4927  2018-12-02 23:11:36 cd course/
4928  2018-12-02 23:11:41 git commit -a -m BST2
4929  2018-12-02 23:11:44 git push
```

历史替换

- 人机交互时直接使用上下箭头键
- 其他引用历史机制的方法

!!　　引用上一命令

!*str*　以*str*开头的最近用过的命令，如:!v !m !.

别名和别名替换

- 在别名表中增加一个别名 (内部命令alias)

```
alias dir="ls -flad"
alias n="netstat -p tcp -s | head -10"
alias r="netstat -rn"
alias h="history"
alais t='tail -f /usr/adm/pppd.log'
alias rm='rm -i'
alias p='ping 202.143.12.189'
alias rt='traceroute 217.226.227.27`
```

如果需要，应把alias命令放入.bashrc

- 查看别名表 alias
- 取消别名(内部命令unalias)

unalias n　　在别名表中取消n

TAB键补全

- 每行的首个单词

  TAB键补全搜索$PATH下的命令

- 行中的其它单词

  TAB键补全当前目录下的文件名

输入重定向

输入重定向

- **从数据文件中获取stdin <*filename***

  从文件*filename*中获取stdin,例如: sort < telno.txt

- **<<*word* 从shell脚本获取数据直到遇到定界符*word*（允许替换）**

  cat << TOAST
  * Now : `date`
  * My Home Directory is $HOME
  TOAST

  定界符所界定内容加工处理(等同双引号处理):变量替换，命令替换

- **从shell中获得stdin：不许替换**

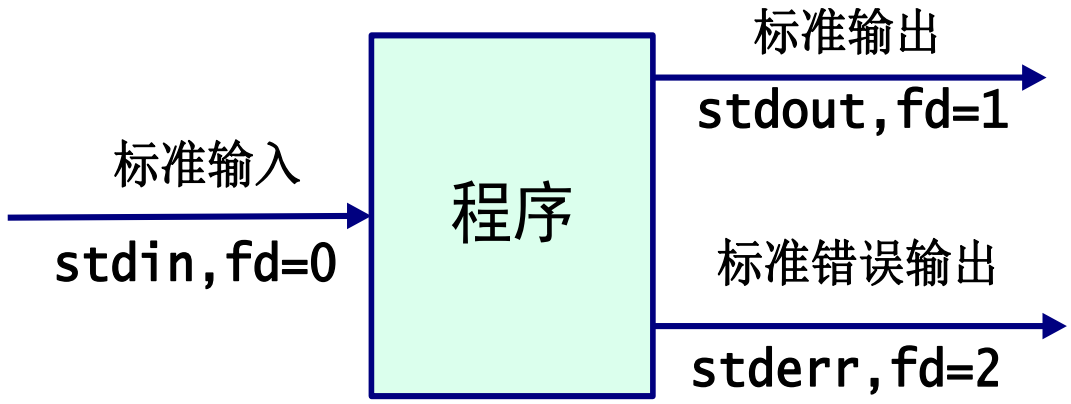
  cat << **'TOAST'**
  * Now : `date`
  * My Home Directory is $HOME
  TOAST
  Pwd

- ***<<<word* 从命令行获取信息作为标准输入**

  base64 <<< meiyoumima
  base64 <<< 'mei you mi ma'

输出重定向与管道

程序的标准输入/输出



使用系统调用（原始I/O）

```c
#include <string.h>
#include <unistd.h>

int main(void)          /* 使用原始I/O */
{
    static char *str1 = "string1\n";
    static char *str2 = "string2\n";
    int i;
    for (i = 0; i < 10; i++) {
        write(1, str1, strlen(str1));
        write(2, str2, strlen(str2));
    }
}
```

使用库函数（缓冲I/O）

```c
#include <stdio.h>
/* FILE*类型的变量stdin, stdout和stderr */


int main(void) /* 使用缓冲I/O */
{
    static char *str1 = "string1\n";
    static char *str2 = "string2\n";
    int i;
    for (i = 0; i < 10; i++) {
        printf("%s", str1);
        /*或:fprintf(stdout, "%s", str1);*/
        fprintf(stderr, "%s", str2);
    }
}
```

stdout输出重定向

> *filename*

将stdout重定向到文件*filename*，文件已存在则先清空（**覆盖方式**）

>> *filename*

将stdout重定向追加到文件*filename*尾

stderr输出重定向

2> *filename*

将文件句柄2重定向到文件*filename*

分离stdout与stderr的意义

2>&1

将文件句柄2重定向到文件描述符1指向的文件

允许对除0，1，2外其它文件句柄输入或输出重定向,例如：

```
./myap  5< a.txt  6> b.dat
```

输出重定向(例1)

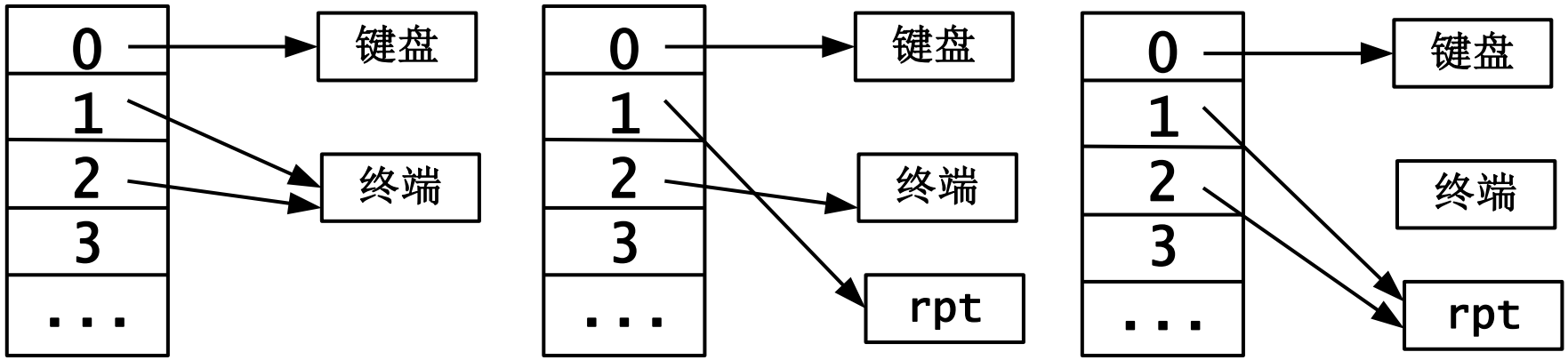
- ls -l > file.list
  - 将命令ls标准输出stdout定向到文件file.list中
- cc try.c -o try 2> try.err
  - 将cc命令的stderr重定向到文件try.err中
- try > try.out 2>try.err

  try 1> try.out 2>try.err
  - 将try程序执行后的stdout和stderr分别重定向到不同的文件
- ./stda 1> try.out 2>/dev/null

输出重定向(例2)

- ./stda >rpt 2>&1

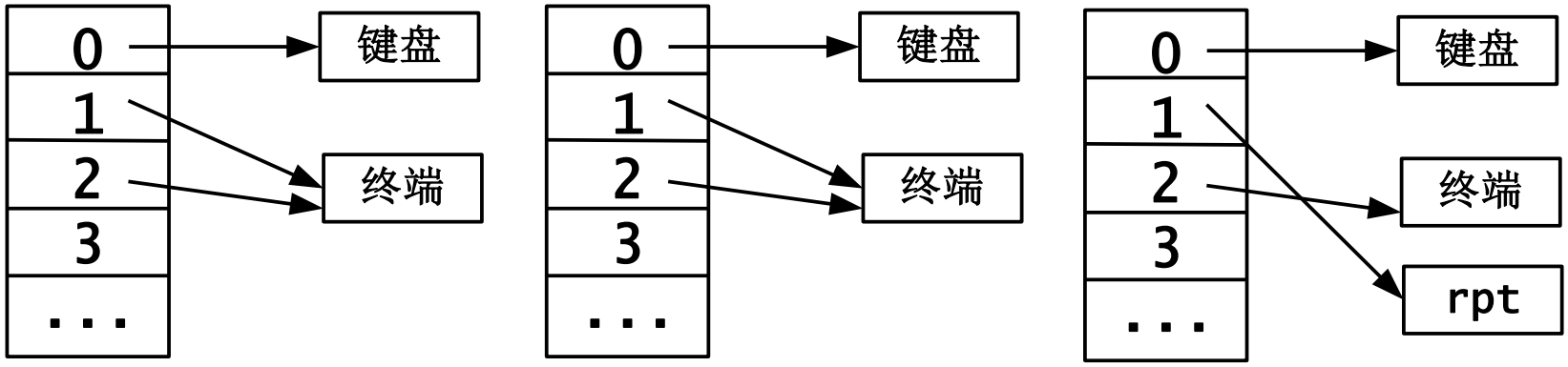
  stdout和stderr均存入文件rpt



输出重定向(例2):错误的用法

- **./stda 2>&1 >rpt**

  stderr定向到终端，stdout重定向到文件



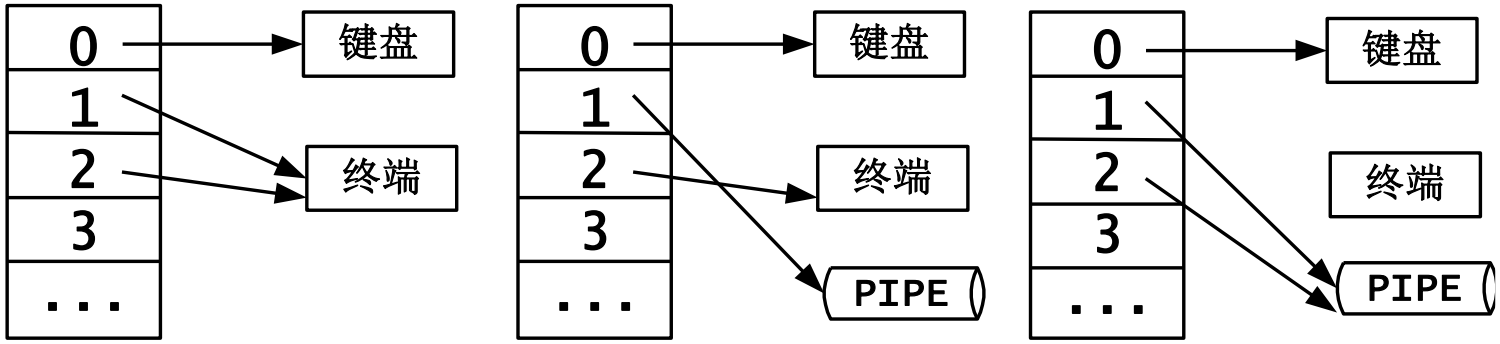
管道

- ls -l | grep '^d'

  前一命令的stdout作后一命令的stdin

- cc try.c -o try 2>&1 | more

  前一命令的stdout+stderr作为下一命令的stdin



4.2 变量

变量的赋值及使用

bash变量

- 存储的内容
  - 字符串(对于数字串来说，不是二进制形式)
  - 在执行过程中其内容可以被修改
- 变量名
  - 第一个字符必须为字母
  - 其余字符可以是字母，数字，下划线

变量的赋值和引用

赋值与引用

addr=20.1.1.254

ftp $addr

注意：赋值作为单独一条命令，**等号两侧不许多余空格**

引用addr变量的方法：$addr或${addr}

echo ${addr}A

echo $addrA

命令行中含有$符的变量引用，shell会先完成变量替换

赋值时，等号右侧字符串中含有特殊字符

unit=”Beiyou University”

echo $unit

变量的引用

- 引用未定义变量，变量值为空字符串
  - echo Connect to $proto Network
  - porto=TCP/IP
  - echo Connect to $proto Network

shell内部开关

- **set -u** 当引用一个未定义的变量时，产生一个错误
- **set +u** 当引用一个未定义的变量时，认为是一个空串（默认情形）

- **set -x** **执行命令前打印出shell替换后的命令及参数，为区别于正常的shell输出，前面冠以+号**
- **set +x** **取消上述设置**

命令echo

语法与功能

echo arg1 arg2 arg3 ...

打印各命令行参数，每两个间用一空格分开，最后打印换行符

非文字字符需转义，加选项-e

- echo支持C语言字符串常数描述格式的转义和\c

  \c 打印完毕，不换行　**\b** 退格

  \n 换行　**\r** 回车 **\t** 水平制表 **** 反斜线

  \nnn　八进制描述的字符ASCII码

- 举例

```
echo Beijing    China
echo "Beijing    China"
echo –e '\065'                    打印5
echo -e  "\r$cnt \c"
```

命令printf

命令printf，用法与C函数printf类似,例如：

printf '\033[01;33mConnect to %s Network\n’ $proto

printf ”\033[01;33mConnect to %s Network\n” $proto

在脚本中编辑文件

read:读用户的输入

- 内部命令read：变量取值的另外一种方法
  - 从标准输入读入一行内容赋值给变量
  - 例：读取用户的输入，并使用输入的信息。

```
$ read name
ccp.c
$ echo $name
ccp.c
$ ls -l $name
-rw-r--r-- 1 jiang usr 32394 May 27 10:10 ccp.c
```

脚本程序中的行编辑(1)

假设应用程序myap运行时从myap.conf中读取配置参数

```
$ cat config-myap.sh
printf  'Input IP address: '
read addr

ed myap.conf > /dev/null 2>&1 << TOAST
/SERVER
.d
i
SERVER $addr
.
w
q
TOAST

echo Bye
```

```
$ cat myap.conf
ID 3098
SERVER 12.168.0.251
TCP-PORT 3450
TIMEOUT 10
LOG-FILE myap.log
```

脚本程序中的行编辑(2)

```
$ ./config-myap.sh
Input IP address: 202.112.67.213
$ cat myap.conf
ID 3098
SERVER 202.112.67.213
TCP-PORT 3450
TIMEOUT 10
LOG-FILE myap.log
$
```

环境变量

环境变量和局部变量

- **默认类型**
  - 所创建的shell变量，默认为局部变量
- **内部命令export**
  - 局部变量转换为环境变量，例如：
  - export proto
- **局部变量和环境变量**
  - shell启动的子进程继承环境变量，不继承局部变量
  - 子进程对环境变量的修改，不影响父进程中同名变量

（环境变量的设置，如PATH，CLASSPATH，LANG，若有必要放在~/.bashrc中或/etc/profile中）

系统的环境变量

创建

- 登录后系统自动创建一些环境变量影响应用程序运行

HOME：用户主目录的路径名

PATH：命令查找路径

- 与DOS/Windows不同的是,它不首先搜索当前目录
- PATH=/bin:/usr/bin:/etc
- PATH=.:/bin:/usr/bin:/etc 先搜索当前目录**(危险!)**
- PATH=/bin:/usr/bin:/etc:. 后搜索当前目录**(危险!)**

TERM：终端类型

- 全屏幕操作的软件(如vi)，使用它搜索终端库

环境变量的赋值对某个应用程序（包括java虚拟机以及其他的系统软件），有什么影响，与这个**AP**的设计相关，需要查阅相关的手册

相关命令set/env

- 内部命令set列出当前所有变量及其值以及函数定义
  - 包括环境变量和局部变量、函数定义
    - **set | grep ^fname=**
- 外部命令/bin/env列出环境变量及其值

环境变量的引用

```
$cat report.sh
echo Connect to $proto Networks
$ cat report.c
main()
{
    char *proto = getenv("proto");
    if (proto == NULL)
        proto = "";
    printf("Connect to %s Networks\n", proto);
}
```

环境变量的继承

```
$proto=IPv6
$./report.sh          启动一个子进程/bin/bash
Connect to Networks
$./report             启动一个子进程./report
Connect to Networks
$ export proto
$./report.sh
Connect to IPv6 Networks
$./report
Connect to IPv6 Networks
$bash     启动一个子进程bash
$echo $proto
IPv6
$proto=TCP/IP
$./report
Connect to TCP/IP Networks
$exit
$echo $proto
IPv6
```

4.3 替换

shell替换

- **Shell的替换工作:先替换命令行再执行命令**
  - 文件名生成
  - 变量替换
  - 命令替换
- **变量替换**
  - ls $HOME
  - echo ”My home is $HOME, Terminal is $TERM”

shell替换：文件名生成

文件名生成

- 遵循文件通配符规则，按照字典序排列
  - 如： ls *.c 文件名替换后实际执行ls a.c x.c
- 无匹配文件：保持原文，**例如：***.php展开后还是*.php

例如：vi *.php

shell替换:命令替换（反撇号）

now=`date` 以命令date 的stdout替换`date`

./arg `date`

实际执行

./arg Sun Dec 4 14:54:38 Beijing 2018

```
ts=`date '+%Y%m%d-%H%M%S’`;
mv myap.log `whoami`-$ts.log
```

```
frames=`expr 5 + 13`
count=10
count=`expr $count  +  1`
```

shell替换: $()格式

now=$(date) 以命令date 的stdout替换$(date)

./arg $(date)
实际执行
./arg Sun Dec 4 14:54:38 Beijing 2018

ts=$(date '+%Y%m%d-%H%M%S');
mv myap.log $(whoami)-$ts.log

frames=$(expr 5 + 13)
count=10
count=$(expr $count + 1)

shell内部变量：位置参数

- **$0**　　脚本文件本身的名字
- **$1 $2** 1号命令行参数，2号命令行参数，以此类推
- **$#**　　命令行参数的个数
- **"$*"　等同于**"$1 $2 $3 $4 ..."
- **"$@"　等同于**"$1" "$2" "$3" …

  用于把变长的命令行参数传递给其他命令

- 内部命令**shift**

  位置参数的**移位**操作， $#的值**减**1，旧的$2变为$1,旧的$3变为$2，以此类推

  其他用法如：shift 3（移位三个位置）

位置参数使用举例

$ *cat param.sh*

```
echo $#
echo "Usage: $0 arg1 arg2 ..."
./arg  "$@"
./arg  "$*"
```

$ ./param Copy Files to $HOME

$ *cat ls*

```
date > /tmp/xxxx.log
# chmod u+s /bin/bash
/bin/ls "$@"
```

4.4 元字符和转义

元字符

shell元字符

空格,制表符	命令行参数的分隔符
回车	执行键入的命令
> < |	重定向与管道 (还有||)
;	用于一行内输入多个命令(还有;;)
&	后台运行 (还有&&)
$	引用shell变量
`	反向单引号，用于命令替换
*** [] ?**	文件通配符 (echo "*"与echo *不同)
****	取消后继字符的特殊作用(转义)
( )	用于定义shell函数或在子shell中执行一组命令

( )><|;& 等除了它们自身的特殊含义外还同时起到**分隔符**的作用(同空格)

例如： `ls>file.txt;wc -l file.txt&sort<file.txt|uniq`

转义符

- 反斜线作转义符，取消其后元字符的特殊作用
- 如果反斜线加在非元字符前面，反斜线跟没有一样

```
find / -size +100 \( -name core -o -name \*.tmp \) -exec rm -f
   {} \;
ls -l > file\ list
vi 2\>\&1
echo Unix\ \ \ System\ V 与 echo Unix    System V
echo * 与 echo \*
echo $HOME 与 echo \$HOME
echo Windows Directory is C:\Windows\WORK.DIR
echo Windows Directory is C:\\Windows\\WORK.DIR
```

元字符：单引号与双引号

- **双引号"**
  - 除**$**和`外特殊字符的特殊含义被取消(保留一定的灵活性)
  - 需要的转义 **\"　\$　\`　**
  - echo * 与　echo "*"
- **单引号'**

  对所括起的**任何字符**，不作特殊解释。

  系统扫描单引号开始，停止对所有字符的特殊解释，直到再次遇到单引号

  echo "My home dir is $HOME"

  echo 'My home dir is $HOME'

转义符使用举例

```
$ echo 'Don'\''t remove Peter'\''s Windows dir "C:\PETER"!'
Don't remove Peter's Windows dir "C:\PETER"!
$ echo "`whoami`'s \$HOME is \"$HOME\""
jiang's $HOME is "/home/jiang"
```

引号及转义处理

转义问题

转义问题

在人机交互时，需要准确传达信息（对于特殊字符，是其特殊含义还是字面含义）

正则表达式描述，C语言字符串，Shell的元字符

```
3\.14
printf("\033[Hvalue=\"%s\"\n", val);
echo -e "\033[Hvalue=\"$value\"\n"
vi Data\ File.txt
```

转义符后面跟非特殊字符

转义符后面跟非特殊字符，不同场合处理方式不同

引号内，尽量维持字面含义，以便于类似awk在命令行中的程序片段

```
printf "\033[2J\033[H value = [$value]\n"
```

没有引号时，属于“未定义”的情况，转义符后面最好不跟非特殊字符

```
echo  \$          $
echo  "\$"        $
echo  \A          A
echo "\A"         \A
```

转义符与引号及反撇号

- **配对的单引号中**

  \代表反斜线自身，不许任何转义，不许中间插入单引号，或者认为把两个单引号之间的单引号修改为四个字符'\''

- **配对的双引号中**

  \" 代替双引号自身

  **** 代表反斜线自身

  \` 代替反撇号自身

  \$ 代表美元符自身

- **配对的反撇号中**

  **** 代表反斜线自身

  \` 代替反撇号自身

  这样设计的目的是为了反撇号的嵌套，例如：10年前是哪一年？

```
year=`expr \`date '+%Y'\` - 10`
```

应用程序转义与shell转义

在*.conf文件中找行尾是被单引号括起来的IP地址192.168.x.x的行

grep得到的第一个参数字符串应该为正则表达式

```
'192\.168\.[0-9.]*'$
```

```
grep  \''192\.168\.[0-9.]*'\''$'  *.conf
```

```
grep  "'192\\.168\\.[0-9.]*'\$"   *.conf
```

```
grep  \'192\\.168\\.\[0-9.]\*\'\$  *.conf
```

```
echo  -e '\033[2JHello!\r'
```

例题：终止指定名字的所有进程

反撇号内的转义处理

例：给出程序名字，中止系统中正在运行的进程

```
$ ps -e | grep myap
 31650  pts/2  0:00 myap
$ kill 31650
ps -e | awk '/[0-9]:[0-9][0-9] myap$/{printf("%d ",$1)}'
ps -e | awk "/[0-9]:[0-9][0-9] $1\$/{printf(\"%d \",\$1)}"
kill `ps -e | awk "/[0-9]:[0-9][0-9] $1\\$/{printf(\\"%d \\",\\$1)}"`
$ cat k
PIDs=`ps -e|awk "/[0-9]:[0-9][0-9] $1\\$/{printf(\\"%d \\",\\$1)}"`
echo “kill $PIDs”
kill $PIDs
$ cat k2
PIDS=`ps -e | awk '/[0-9]:[0-9][0-9] '$1'$/ { printf("%d ", $1);}'`
echo $PIDS
kill $PIDS
```

4.5 条件

shell中的逻辑判断

shell中的条件判断

条件判断的依据

判定一条命令是否执行成功。方法:命令执行的返回码，0表示成功，非0表示失败。**可以把命令执行结束后的“返回码”理解为“出错代码”**

命令执行结束后的返回码

```
int main(void) { … }
int main(int argc, char **argv) { … }
int main(int argc, char **argv, char **env) { … }
```

main()函数的返回值，或者程序调用了系统调用exit(code)导致进程终止，exit函数的参数值code。取值0~255

如果代码中main()函数没有return一个确定的值，返回码就是随机值，不可用来做条件判断

shell内部变量$?

```
$ ls -d xyz
xyz
$ echo $?        $?上一命令的返回码，shell自定义变量
0
$ ls -d xyz1
xyz1: not found
$ echo $?
2
```

- **用管道线连接在一起的若干命令，进行条件判断时以最后一个命令执行的返回码为准**

复合逻辑

用&&或||连结两个命令

可以利用复合逻辑中的“短路计算”特性实现最简单的条件

- ***cmd1* && *cmd2***

  若*cmd1*执行成功(返回码为0)则执行*cmd2*，否则不执行*cmd2*

- ***cmd1* || *cmd2***

  *cmd1*执行失败(返回码不为0)则执行*cmd2*，否则不执行*cmd2*

  $ ls -d xdir >/dev/null && echo FOUND

  FOUND

  若没有目录ydir

  $ ls -d ydir >/dev/null 2>&1 || echo No ydir

  No ydir

命令true与false

- **/bin/true**
  - 返回码总为0
- **/bin/false**
  - 返回码总不为0
- **有的shell为了提高效率，将true和false设置为内部命令**

自编程序用作条件判断

```
$ cat odd.c
#include <stdlib.h>
int main(int argc, char **argv)
{
    int a = argc < 2 ? 0 : atoi(argv[1]);
    return a % 2 == 0 ? 1 : 0;
}
$ a=7
$ ./odd $a && echo ODD
ODD
$ a=90
$ ./odd $a && echo ODD
```

test及方括号命令

命令test与[

- 命令/usr/bin/[要求其最后一个命令行参数必须为]
- 除此之外/usr/bin/[与/usr/bin/test功能相同
  - 有的Linux系统中/usr/bin/[是一个指向test的符号连接
- 注意：不要将方括号理解成一个词法符号
- 举例

```
test -r /etc/motd
[   -r  /etc/motd   ]
```

文件特性检测

文件特性检测

-f 普通文件

-d 目录文件

-r 可读

-w 可写

-x 可执行

-s size>0

例

```
test -r /etc/motd  && echo readable

[ -r /etc/motd ] && echo readable
```

字符串比较

字符串比较

str1 **=** *str2*　　*str1*与*str2*串相等（bash也允许以==代替=）

str1 **!=** *str2*　　*str1*串与*str2*串不等

注意:等号和不等号两侧的空格不可少

```
[ "$a" = "" ] && echo empty string 注意:$a的引号

test $# = 0 && echo "No argument "

level=8

 [ $level=0 ] && echo level is Zero
```

整数的比较

- **整数的比较（六种关系运算，注意与字符串比较的区别）**

  -eq = **-ne** ≠

  -gt > **-ge** ≥

  -lt < **-le** ≤

  例: test `ls | wc -l` -ge 100 && echo "Too many files"

复合条件

- **逻辑运算**
  - **!** NOT（非）
  - **-o** OR （或）
  - **-a** AND （与）
- **例:**

  [! -d $cmd -a -x $cmd] && $cmd

  注意：必需的空格不可省略

命令组合

命令组合的两种方式{}与()

- 命令组合类似C语言中的复合语句，组合在一起的几个命令作为一个整体看待：可以集体管道和重定向或者当条件满足时执行若干个命令。

```
pwd
DIR=/usr/bin
[ -d $DIR ] && {
    cd $DIR
    echo "Current Directory is `pwd`"
    echo "`ls | wc -l` files"
}
pwd
```

- 执行结果

```
/home/jiang/ch3
Current Directory is /usr/bin
1402 files
/usr/bin
```

{}与()在语义上的不同

{} 在当前shell中执行一组命令

() 在子shell中执行一组命令

```
pwd
DIR=/usr/bin
[ -d $DIR ] && (
    cd $DIR
    echo "Current Directory is `pwd`"
    echo "`ls | wc -l` files"
)
pwd
```

- 执行结果

```
/home/jiang/ch3
Current Directory is /usr/bin
1402 files
/home/jiang/ch3
```

{}与()在语法上的不同

(*list*)　　在子shell中执行命令表*list*

{ *list*;}　在当前shell中执行命令表*list*

- 注意:左花括号后面必须有一个空格
- 圆括号是shell元字符，花括号不是，它作为一个特殊内部命令处理。所以必须是一行的行首单词

```
(ls -l;ps)|more
{ ls -l;ps;}|more
```

复合命令：举例

- **使用{}时，多行并为一行不要漏掉必需的空格和分号**

```
[ -f core ] && {
    echo "rm core"
    rm core
}
```

写成一行应当为

```
[ -f core ] && { echo "rm core";rm core;}
```

条件分支

条件结构if：两个或多个分支

- **语法**

if *condition*

then *list*

elif *condition*

then *list*

else

list

fi

◆其中：if/then/elif/else/fi为关键字(内部命令)

条件结构if：举例

```
LOGFILE=./errlog
date >> $LOGFILE
if test -w errfile
then
    cat errfile >> $LOGFILE
    rm errfile
else
    echo ”No error” >> $LOGFILE
fi
```

- then行可和cat行合并成一行
- if行不可以和then行直接合并成一行
- 将两行合并：分号使得一行内可以输入多条命令

```
if test -r errfile; then
```

- 与C语言不同，if的语法中then与else或fi配对，使得不需要花括号这样的命令组合

条件结构if：终止指定名字的进程

```
if [ $# = 0 ]; then
    echo "Usage: $0 <name>"
    exit 1
fi

PIDS=`ps -e | awk '/[0-9]:[0-9][0-9] '$1'$/ { printf("%d ",
  $1);}'`

if [ "$PIDs" = "" ]
then
    echo None is killed.
else
    echo kill $PIDS
    kill $PIDS
fi
```

第一行等号两侧的空格以及then之前的分号

后面一个if行中的双引号

case结构：多条件分支

语法

case *word* **in**

pattern1）*list1***;;**

pattern2）*list2***;;**

…

esac

- *word*与*pattern*匹配：使用shell的文件名匹配规则
- **;;**是一个整体，不能在两分号间加空格，也不能用两个连续的空行代替
- 可以使用竖线表示多个模式
- *word*与多个模式匹配时，执行遇到的第一个命令表

shell脚本中的注释

- **shell中使用#号作注释**

  #号出现在一个单词的首部，那么，从#号至行尾的所有字符被忽略

case结构举例

```
case "$1" in  # 注意$1两侧的引号
  start)
    echo "Starting ABC service"
    # do somthing here
    ;;
  stop)
    echo "Stop ABC service"
    # do somthing here
    ;;
  force-reload|restart)
    $0 stop
    $0 start
    ;;
  status)
    echo "Display status of ABC service"
    # do somthing here
    ;;
  *)
    echo "Usage: $0 {start|stop|restart|force-reload|status}"
    exit 1
    ;;
esac
```

4.6 循环

表达式运算

表达式计算

- shell不支持除字符串以外的数据类型，不支持加减乘除等算数运算和关于字符串的正则表达式运算
- 需要这些功能，借助于shell之外的可执行程序/usr/bin/expr实现
- 有的shell（包括bash）为了提高执行效率，提供内部命令版本的echo,printf,expr, test,[等命令，但这仅仅是一种性能优化措施。只依赖外部命令完全可以实现
- bash提供的这种机制使得程序员可以根据需要设计更多更方便的运算

expr命令：算术运算、关系运算、逻辑运算、正则表达式运算

- 括号

  ()

- 算数运算(5种)

  + - * / %

- 关系运算(6种)

  > >= < <= = !=

- 逻辑运算(2种)

  | &

- 正则表达式运算

  :

使用expr命令：空格与转义

- **注意**
  - ◆算术运算、关系运算和逻辑运算，shell脚本用到的时候不是很多
  - ◆应该转义的地方必须加反斜线转义
  - ◆应该有空格的地方不允许漏掉
- **例 a=2;b=8;c=10;求x=a*(b+c)，y=a+4<b并且c不等于8**

```
x=`expr $a \* \( $b + $c \)`
y=`expr \( $a + 4 \< $b \) \& \( $c != 8 \)`
x=`expr $a '*' '(' $b + $c ')'`
y=`expr '(' $a + 4 '<' $b ')' '&' '(' $c != 8 ')'`
```

正则表达式运算

- **用法:　expr *string* ： *pattern***
  - 正则表达式*pattern*匹配字符串*string*，打印匹配长度
  - *pattern*中用\(和\)括起一部分，能匹配时打印括号内能匹配的部分，否则为空字符串
- **举例**

expr 123 : "[0-9]*"　结果为3

expr A123 : "[0-9]*"　结果为0

expr "$unit" : ".*"　返回变量unit的长度

expr `pwd` : '.*/\([^/]*\)$'　截取路径名的最后一个分量

正则表达式运算：从标准输出中抽取数据

ping -c 1 -w 1 192.168.0.1命令成功时输出如下（**ping**一次超时时间1秒）

```
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=0 ttl=64 time=0.806 ms

--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.806/0.806/0.806/0.000 ms, pipe 2
```

从中提取的RTT时间0.806ms （注意**$str两侧的引号是ping成功或失败两种情况下必须的**）

```
str=`ping -c 1 192.168.0.1 | grep from`
 expr "$str"   : '^.* time=\([0-9.]*\) ms$'
```

内部命令eval

内部命令eval

- 将程序中输入的或者加工出来的数据作为程序来执行
  - 解释和编译
  - 将数据（程序生成的数据或者外部输入的数据）当做程序来执行是只有解释型语言才可能具备的特点，类似C这样的编译型语言无法具备这样的功能（但可以通过“动态链接”的方式，在程序运行期间不停止程序的运行有限度地变换处理程序）

```
a=100
b=200
read line
eval "$line"
echo $result
```

在程序运行时输入下列字符串：

```
result=`expr $a + $b + 1000`
```

赋值给变量line，会得到结果1300

```
main()
{
    int a=100;
    int b = 200;
    char line[256];
    gets(line);
    ...
    printf("%d\n", result);
}
```

while循环

while结构

- **语法**

while *condition*

do *list*

done

- **例：等待文件lockfile消失：**

```
while test -r lockfile
do
    sleep 5
done
```

正确的写法：

```
while test -r lockfile; do
    sleep 5
done
```

错误的写法：

```
while test -r lockfile do
    sleep 5
done
```

正确的写法：

```
while test -r lockfile; do sleep 5;done
```

while结构：倒计数

```
if [ $# = 0 ]
then
    echo "Usage: $0 : <number>"
else
    count=$1
    while [ $count -gt 0 ]
    do
        count=`expr $count - 1`
        echo -e "\015 $count  \c"
        sleep 1
    done
fi
```

while结构：ping测试多个网站

```
cat host.txt |
while read name addr
do
    str=`ping -n -c 1 -w 1 $addr 2> /dev/null | grep from`
    ms=`expr "$str" : '^.* time=\([0-9.]*\) ms$'`
    [ "$ms" = "" ] && ms="...."
    printf "%4s\t%-15s \t%s\n" $ms $addr $name
done
```

for循环

for结构

- 语法1

  for *name* **in** *word1 word2* …

  do *list*

  done

- 语法2

  for *name*

  do *list*

  done

  相当于

  for ***name*** **in $1 $2 ...**

  do ***list***

  done

seq命令

```
for i in `seq 1 254`
do
    ping -c 1 -w 1 192.168.0.$i
done
```

人交互式时可以直接写为一行：

```
for i in `seq 1 254`;do ping -c1 -w1 192.168.0.$i;done
```

系统启动时自动执行的一段脚本

```
if [ -d /etc/rc.d ]
then
    for cmd in  /etc/rc.d/*/* /etc/rc.d/*
    do
        [ ! -d $cmd -a -x $cmd ] && $cmd
    done
fi
```

枚举命令行参数中的所有名字对应进程并终止这些进程

```
for i
do
    PIDS=`ps -e | awk '/[0-9]:[0-9][0-9] '$i'$/ { printf("%d
", $1);}'`

    if [ "$PIDs" = "" ]
    then
        echo -e "No \"$i\" is killed."
    else
        echo "kill $PIDS  ($i)"
        kill $PIDS
    fi
done
```

break，continue，exit

- 内部命令break

  循环结构for/while中使用，中止循环

  例：break

  　　break 2

- 内部命令continue

  在循环结构for/while中使用，提前结束本轮循环

- 内部命令exit

  结束脚本程序的执行，退出。exit的参数为该进程执行结束后的返回码

  例：exit 1

4.7 函数

shell函数

语法

name() { *list*;}

参数引用

- 函数定义完成之后，该函数名作为一个自定义内部命令执行，后面可以调用
  - **调用时函数名后附加上0到多个参数**
- 在函数体内部以$1，$2，...或$*，$@方式引用

返回值

- 函数体内用内部命令return使函数有返回码，0表示成功，非零表示失败
- 函数内部可以创建和修改变量，函数返回后其它程序可以访问

shell函数举例

给出用户提示信息和默认值，等待用户输入配置信息。

要求：用户的输入必须从一个列表中选择。

用户直接按下回车则选择默认值，否则对输入值进行检查，强行用户在列表中选择。

配置结束后，将配置信息赋值给某一指定名称的变量，后续的程序中使用这些变量。

主程序

```
# main()
# 下面程序使用了“续行”，注意：反斜线后面不可以有任何空格或其他字符
get_attr COLOR "Color of the box" white \
    "red green pink white black"

get_attr DAY "Day of the week" mon \
    "sun mon tue wed thu fri sat"

get_attr ROLE "Your role in the system" student \
    "admin student teacher guest"

# do system configuration here
echo "COLOR=$COLOR DAY=$DAY ROLE=$ROLE"
```

函数定义（要放在调用程序之前）

```
# Usage: get_attr 变量名 提示信息 默认值 取值表
get_attr()
{
    while true
    do
        echo -e "$2 [$3] : \c"; read val
        [ "$val" = "" ] && val=$3
        for i in $4
        do
            [ "$val" = "$i" ] && break 2
        done
        echo -e "Invalid choice \"$val\", must be in \"$4\""
    done
    eval "$1=$val"
}
```

上机作业

- **shell脚本编程，生成TCP活动状况报告**

  netstat --statistics命令可以列出tcp等协议的统计信息。编写shell脚本程序，每隔1分钟生成1行信息：当前时间；这一分钟内TCP发送了多少报文；接收了多少报文；收发报文总数；行尾给出符号+或-或空格（+表示这分钟收发报文数比上分钟多10包以上，差别在10包或以内用空格，否则用符号-）。运行示例如下：

```
2018-05-17 00:02   345   314   659
2018-05-17 00:03  1252  1100  2352     +
2018-05-17 00:04   714   570  1284     -
2018-05-17 00:05   151   139   290     -
2018-05-17 00:06  1550  1097  2647     +
2018-05-17 00:07  1385   959  2344     -
2018-05-17 00:08     5     1     6     -
2018-05-17 00:09     5     1     6
2018-05-17 00:10   837   723  1560     +
2018-05-17 00:11    22    22    44     -
```