# 第十一章：shell 基础

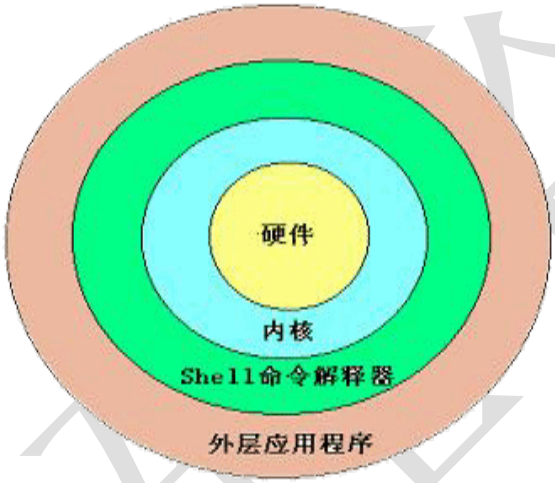
尚硅谷云计算 Linux 课程

版本：V1.0

讲师：沈超

## 一、Shell 概述

1、什么是 Shell




2、shell 的分类

| Shell 类别 | 易学性 | 可移植性 | 编辑性 | 快捷性 |
|---|---|---|---|---|
| Bourne Shell (sh) | 容易 | 好 | 较差 | 较差 |
| Korn Shell (ksh) | 较难 | 较好 | 好 | 较好 |
| Bourne Again (Bash) | 难 | 较好 | 好 | 好 |
| POSIX Shell (psh) | 较难 | 好 | 好 | 较好 |
| C Shell (csh) | 较难 | 差 | 较好 | 较好 |
| TC Shell (tcsh) | 难 | 差 | 好 | 好 |

Shell 的两种主要语法类型有 Bourne 和 C，这两种语法彼此不兼容。Bourne 家族主要包括 sh、ksh、Bash、psh、zsh；C 家族主要包括：csh、tcsh （Bash 和 zsh 在不同程度上支持 csh 的语法）。

我们可以通过/etc/shells 文件来查询 Linux 支持的 Shell。命令如下：

```
[root@localhost ~]# vi /etc/shells
/bin/sh
/bin/Bash
/sbin/nologin
/bin/tcsh
/bin/csh
```

## 二、Shell 脚本的执行方式

1、echo 命令

```
[root@localhost ~]# echo [选项] [输出内容]
选项：
    -e：      支持反斜线控制的字符转换（具体参见表 11-2）
    -n：      取消输出后行末的换行符号（就是内容输出后不换行）
```

例子 1：

```
[root@localhost ~]# echo "Mr. Shen Chao is the most honest man! "
#echo 的内容就会打印到屏幕上。
Mr. Shen Chao is the most honest man!
[root@localhost ~]#
```

例子 2：

```
[root@localhost ~]# echo -n "Mr. Shen Chao is the most honest man!"
Mr. Shen Chao is the most honest man![root@localhost ~]#
#如果加入了“-n”选项，输出内容结束后，不会换行直接显示新行的提示符。
```

在 echo 命令中如果使用了“-e”选项，则可以支持控制字符，如表 11-2 所示：

| 控制字符 | 作　　用 |
|---|---|
| \\ | 输出\本身 |
| \a | 输出警告音 |
| \b | 退格键，也就是向左删除键 |
| \c | 取消输出行末的换行符。和“-n”选项一致 |
| \e | ESCAPE 键 |
| \f | 换页符 |
| \n | 换行符 |
| \r | 回车键 |
| \t | 制表符，也就是 Tab 键 |
| \v | 垂直制表符 |
| \0nnn | 按照八进制 ASCII 码表输出字符。其中 0 为数字零，nnn 是三位八进制数 |
| \xhh | 按照十六进制 ASCII 码表输出字符。其中 hh 是两位十六进制数 |

例子 3：

```
[root@localhost ~]# echo -e "\\ \a"
\
#这个输出会输出\，同时会在系统音响中输出一声提示音
```

例子 4：

```
[root@localhost ~]# echo -e "ab\bc"
ac
#这个输出中，在 b 键左侧有“\b”，所以输出时只有 ac
```

例子 5：

```
[root@localhost ~]# echo -e "a\tb\tc\nd\te\tf"
a       b       c
```

```
d       e       f
#我们加入了制表符“\t”和换行符“\n”,所以会按照格式输出
```

例子 6：

```
[root@localhost ~]# echo -e "\0141\t\0142\t\0143\n\0144\t\0145\t\0146"
a       b       c
d       e       f
#还是会输出上面的内容，不过是按照八进制 ASCII 码输出的。
```

也就是说 141 这个八进制，在 ASCII 码中代表小写的“a”，其他的以此类推。

例子 7：

```
[root@localhost ~]# echo -e "\x61\t\x62\t\x63\n\x64\t\x65\t\x66"
a       b       c
d       e       f
#如果按照十六进制 ASCII 码也同样可以输出
```

echo 命令还可以进行一些比较有意思的东西，比如：

例子 8：

```
[root@localhost ~]# echo -e "\e[1;31m  abcd \e[0m"
```

这条命令会把 abcd 按照红色输出。解释下这个命令\e[1 是标准格式，代表颜色输出开始，\e[0m 代表颜色输出结束，31m 定义字体颜色是红色。echo 能够识别的颜色如下：30m=黑色，31m=红色，32m=绿色，33m=黄色，34m=蓝色，35m=洋红，36m=青色，37m=白色。

例子 9：

```
[root@localhost ~]# echo -e "\e[1;42m abcd \e[0m"
```

这条命令会给 abcd 加入一个绿色的背景。echo 可以使用的背景颜色如下：40m=黑色，41m=红色，42m=绿色，43m=黄色，44m=蓝色，45m=洋红，46m=青色，47m=白色。

2、Shell 脚本的执行

```
[root@localhost sh]# vi hello.sh
#!/bin/Bash
#The first program
# Author: shenchao （E-mail: shenchao@atguigu.com）

echo -e "Mr. Shen Chao is the most honest man. "
```

Shell 脚本写好了，那么这个脚本该如何运行呢？在 Linux 中脚本的执行主要有这样两种种方法：

✧ 赋予执行权限，直接运行

这种方法是最常用的 Shell 脚本运行方法，也最为直接简单。就是赋予执行权限之后，直接运行。当然运行时可以使用绝对路径，也可以使用相对路径运行。命令如下：

```
[root@localhost sh]# chmod 755 hello.sh
#赋予执行权限
[root@localhost sh]# /root/sh/hello.sh
Mr. Shen Chao is the most honest man.
#使用绝对路径运行
[root@localhost sh]# ./hello.sh
```

```
Mr. Shen Chao is the most honest man.
#因为我们已经在/root/sh 目录当中，所以也可以使用相对路径运行
```

✧ 通过 Bash 调用执行脚本

这种方法也非常简单，命令如下：

```
[root@localhost sh]# bash hello.sh
Mr. Shen Chao is the most honest man.
```

## 三、Bash 的基本功能

### 1 历史命令

1）历史命令的查看

```
[root@localhost ~]# history [选项] [历史命令保存文件]
选项：
    -c:     清空历史命令
    -w:     把缓存中的历史命令写入历史命令保存文件。如果不手工指定历史命令保存文
            件，则放入默认历史命令保存文件~/.bash_history 中
```

```
[root@localhost ~]# vi /etc/profile
…省略部分输出…
HISTSIZE=1000
…省略部分输出…
```

我们使用 history 命令查看的历史命令和~/.bash_history 文件中保存的历史命令是不同的。那是因为当前登录操作的命令并没有直接写入~/.bash_history 文件，而是保存在缓存当中的。需要等当前用户注销之后，缓存中的命令才会写入~/.bash_history 文件。如果我们需要把内存中的命令直接写入~/.bash_history 文件，而不等用户注销时再写入，就需要使用“-w”选项了。命令如下：

```
[root@localhost ~]# history -w
#把缓存中的历史命令直接写入~/.bash_history
```

这时再去查询~/.bash_history 文件，历史命令就和 history 命令查询的一致了。

如果需要清空历史命令，只需要执行：

```
[root@localhost ~]# history -c
#清空历史命令
```

2）、历史命令的调用

如果想要使用原先的历史命令有这样几种方法：

✧ 使用上、下箭头调用以前的历史命令

✧ 使用“!n”重复执行第 n 条历史命令

✧ 使用“!!”重复执行上一条命令

✧ 使用“!字串”重复执行最后一条以该字串开头的命令

✧ 使用“!$”重复上一条命令的最后一个参数

### 2、命令与文件的补全

3　命令别名

```
命令格式：
[root@localhost ~]# alias
#查询命令别名
[root@localhost ~]# alias 别名='原命令'
#设定命令别名

例如：
[root@localhost ~]# alias
#查询系统中已经定义好的别名
alias cp='cp -i'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias mv='mv -i'
alias rm='rm -i'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'

[root@localhost ~]# alias vi='vim'
#定义 vim 命令的别名是 vi
```

既然我们说别名的优先级比命令高，那么命令执行时具体的顺序是什么呢？命令执行时的顺序是这样的：

1、 第一顺位执行用绝对路径或相对路径执行的命令。

2、 第二顺位执行别名。

3、 第三顺位执行 Bash 的内部命令。

4　第四顺位执行按照$PATH 环境变量定义的目录查找顺序找到的第一个命令。

为了让这个别名永久生效，可以把别名写入环境变量配置文件“~/.bashrc”。命令如下：

```
[root@localhost ~]# vi /root/.bashrc
```

4　Bash 常用快捷键

| 快捷键 | 作　　用 |
|---|---|
| ctrl+A | 把光标移动到命令行开头。如果我们输入的命令过长，想要把光标移动到命令行开头时使用。 |
| ctrl+E | 把光标移动到命令行结尾。 |
| **ctrl+C** | 强制终止当前的命令。 |
| **ctrl+L** | 清屏，相当于 clear 命令。 |
| **ctrl+U** | 删除或剪切光标之前的命令。我输入了一行很长的命令，不用使用退格键一个一个字符的删除，使用这个快捷键会更加方便 |
| ctrl+K | 删除或剪切光标之后的内容。 |
| **ctrl+Y** | 粘贴 ctrl+U 或 ctrl+K 剪切的内容。 |
| **ctrl+R** | 在历史命令中搜索，按下 ctrl+R 之后，就会出现搜索界面，只要输入搜索内容，就会从历史命令中搜索。 |

| | |
|---|---|
| ctrl+D | 退出当前终端。 |
| ctrl+Z | 暂停，并放入后台。这个快捷键牵扯工作管理的内容，我们在系统管理章节详细介绍。 |
| ctrl+S | 暂停屏幕输出。 |
| ctrl+Q | 恢复屏幕输出。 |

5　输入输出重定向

1）、Bash 的标准输入输出

| 设备 | 设备文件名 | 文件描述符 | 类型 |
|---|---|---|---|
| 键盘 | /dev/stdin | 0 | 标准输入 |
| 显示器 | /dev/stdout | 1 | 标准输出 |
| 显示器 | /dev/stderr | 2 | 标准错误输出 |

2）、输出重定向

| 类　型 | 符　号 | 作用 |
|---|---|---|
| 标准输出重定向 | 命令 > 文件 | 以覆盖的方式，把命令的正确输出输出到指定的文件或设备当中。 |
| | 命令 >> 文件 | 以追加的方式，把命令的正确输出输出到指定的文件或设备当中。 |
| 标准错误输出重定向 | 错误命令 2>文件 | 以覆盖的方式，把命令的错误输出输出到指定的文件或设备当中。 |
| | 错误命令 2>>文件 | 以追加的方式，把命令的错误输出输出到指定的文件或设备当中。 |
| 正确输出和错误输出同时保存 | 命令 > 文件 2>&1 | 以覆盖的方式，把正确输出和错误输出都保存到同一个文件当中。 |
| | 命令 >> 文件 2>&1 | 以追加的方式，把正确输出和错误输出都保存到同一个文件当中。 |
| | 命令 &>文件 | 以覆盖的方式，把正确输出和错误输出都保存到同一个文件当中。 |
| | 命令 &>>文件 | 以追加的方式，把正确输出和错误输出都保存到同一个文件当中。 |
| | 命令>>文件 1　2>>文件 2 | 把正确的输出追加到文件 1 中，把错误的输出追加到文件 2 中。 |

3）、输入重定向

```
[root@localhost ~]# wc [选项] [文件名]
选项：
    -c      统计字节数
    -w      统计单词数
    -l      统计行数
```

6　多命令顺序执行

| 多命令执行符 | 格式 | 作　　用 |
| --- | --- | --- |
| ; | 命令 1 ; 命令 2 | 多个命令顺序执行，命令之间没有任何逻辑联系 |
| && | 命令 1 && 命令 2 | 当命令 1 正确执行（$?=0），则命令 2 才会执行<br>当命令 1 执行不正确（$?≠0），则命令 2 不会执行 |
| \|\| | 命令 1 \|\| 命令 2 | 当命令 1 执行不正确（$?≠0），则命令 2 才会执行<br>当命令 1 正确执行（$?=0），则命令 2 不会执行 |

7　管道符

1）、行提取命令 grep

```
[root@localhost ~]# grep [选项] "搜索内容" 文件名
选项：
    -A 数字：        列出符合条件的行，并列出后续的 n 行
    -B 数字：        列出符合条件的行，并列出前面的 n 行
    -c：             统计找到的符合条件的字符串的次数
    -i：             忽略大小写
    -n：             输出行号
    -v：             反向查找
    --color=auto     搜索出的关键字用颜色显示
```

举几个例子：

```
[root@localhost ~]# grep "/bin/bash" /etc/passwd
#查找用户信息文件/etc/passwd 中，有多少可以登录的用户
```

再举几个例子吧：

```
[root@localhost ~]# grep -A 3 "root" /etc/passwd
#查找包含有“root”的行，并列出后续的 3 行

[root@localhost ~]# grep -n "/bin/bash" /etc/passwd
#查找可以登录的用户，并显示行号

[root@localhost ~]# grep -v "/bin/bash" /etc/passwd
#查找不含有“/bin/bash”的行，其实就是列出所有的伪用户
```

2）find 和 grep 的区别

find 命令是在系统当中搜索符合条件的文件名，如果需要模糊查询，使用通配符（通配符我们下一小节进行介绍）进行匹配，搜索时文件名是完全匹配。

```
[root@localhost ~]# touch abc
#建立文件 abc
[root@localhost ~]# touch abcd
#建立文件 abcd
[root@localhost ~]# find . -name "abc"
./abc
#搜索文件名是 abc 的文件，只会找到 abc 文件，而不会找到文件 abcd
#虽然 abcd 文件名中包含 abc，但是 find 是完全匹配，只能和要搜索的数据完全一样，才能找到
```

注意：find 命令是可以通过-regex 选项识别正则表达式规则的，也就是说 find 命令可以按照正则表达式规则匹配，而正则表达式是模糊匹配。但是对于初学者而言，find 命令和 grep 命令本身就不好理解，所以我们这里只按照通配符规则来进行 find 查询。

grep 命令是在文件当中搜索符合条件的字符串，如果需要模糊查询，使用正则表达式进行匹配，搜索时字符串是包含匹配。

```
[root@localhost ~]# echo abc > test
#在 test 文件中写入 abc 数据
[root@localhost ~]# echo abcd >> test
#在 test 文件中再追加 abcd 数据
[root@localhost ~]# grep "abc" test
abc
abcd
#grep 命令查找时，只要数据行中包含有 abc，就会都列出
#所以 abc 和 abcd 都可以查询到
```

3）管道符

```
[root@localhost ~]# ll -a /etc/  | more
```

```
[root@localhost ~]# netstat -an | grep "ESTABLISHED"
#查询下本地所有网络连接，提取包含 ESTABLISHED（已建立连接）的行
#就可以知道我们的服务器上有多少已经成功连接的网络连接
```

```
[root@localhost ~]# netstat -an | grep "ESTABLISHED" | wc -l
#如果想知道具体的网络连接数量，就可以再使用 wc 命令统计行数
```

```
[root@localhost ~]# rpm -qa | grep httpd
```

8　通配符

| 通配符 | 作　　用 |
|---|---|
| ? | 匹配一个任意字符 |
| * | 匹配 0 个或任意多个任意字符，也就是可以匹配任何内容 |
| [] | 匹配中括号中任意一个字符。例如：[abc]代表一定匹配一个字符，或者是 a，或者是 b，或者是 c。 |
| [-] | 匹配中括号中任意一个字符，-代表一个范围。例如：[a-z]代表匹配一个小写字母。 |
| [^] | 逻辑非，表示匹配不是中括号内的一个字符。例如：[^0-9]代表匹配一个不是数字的字符。 |

```
[root@localhost tmp]# touch abc
[root@localhost tmp]# touch abcd
[root@localhost tmp]# touch 012
[root@localhost tmp]# touch 0abc
```

```
#建立几个测试文件
[root@localhost tmp]# ls *
012  0abc  abc  abcd
#“*”代表所有的文件
[root@localhost tmp]# ls ?abc
0abc
#“?”匹配任意一个字符，所以会匹配 0abc
#但是不能匹配 abc，因为“?”不能匹配空
[root@localhost tmp]# ls [0-9]*
012  0abc
#匹配任何以数字开头的文件
[root@localhost tmp]# ls [^0-9]*
abc  abcd
#匹配不已数字开头的文件
```

9　Bash 中其他特殊符号

| 符　号 | 作　用 |
|---|---|
| ,, | 单引号。在单引号中所有的特殊符号，如“$”和“`”(反引号)都没有特殊含义。 |
| ”” | 双引号。在双引号中特殊符号都没有特殊含义，但是“$”、“`”和“\”是例外，拥有“调用变量的值”、“引用命令”和“转义符”的特殊含义。 |
| `` | 反引号。反引号括起来的内容是系统命令，在 Bash 中会先执行它。和$()作用一样，不过推荐使用$()，因为反引号非常容易看错。 |
| $() | 和反引号作用一样，用来引用系统命令。 |
| () | 用于一串命令执行时，()中的命令会在子 Shell 中运行 |
| {} | 用于一串命令执行时，{}中的命令会在当前 Shell 中执行。也可以用于变量变形与替换。 |
| [] | 用于变量的测试。 |
| # | 在 Shell 脚本中，#开头的行代表注释。 |
| $ | 用于调用变量的值，如需要调用变量 name 的值时，需要用$name 的方式得到变量的值。 |
| \ | 转义符，跟在\之后的特殊符号将失去特殊含义，变为普通字符。如\$将输出“$”符号，而不当做是变量引用。 |

1）、单引号和双引号

```
[root@localhost ~]# name=sc
#定义变量 name 的值是 sc（就是最正直的人，超哥我了！）
[root@localhost ~]# echo '$name'
$name
#如果输出时使用单引号，则$name 原封不动的输出
[root@localhost ~]# echo "$name"
sc
#如果输出时使用双引号，则会输出变量 name 的值 sc
```

```
[root@localhost ~]# echo `date`
2018 年 10 月 21 日 星期一 18:16:33 CST
#反引号括起来的命令会正常执行
[root@localhost ~]# echo ’`date`’
`date`
#但是如果反引号命令被单引号括起来，那么这个命令不会执行，`date`会被当成普通字符输出
[root@localhost ~]# echo ″`date`″
2018 年 10 月 21 日 星期一 18:14:21 CST
#如果是双引号括起来，那么这个命令又会正常执行
```

2）反引号

```
[root@localhost ~]# echo ls
ls
#如果命令不用反引号包含，命令不会执行，而是直接输出
[root@localhost ~]# echo `ls`
anaconda-ks.cfg  install.log  install.log.syslog  sh  test  testfile
#只有用反引号包括命令，这个命令才会执行
[root@localhost ~]# echo $(date)
2018 年 10 月 21 日 星期一 18:25:09 CST
#使用$(命令)的方式也是可以的
```

3）、小括号、中括号和大括号

在介绍小括号和大括号的区别之前，我们先要解释一个概念，那就是父 Shell 和子 Shell。在我们的 Bash 中，是可以调用新的 Bash 的，比如：

```
[root@localhost ~]# bash
[root@localhost ~]#
```

这时，我们通过 pstree 命令查看一下进程数：

```
[root@localhost ~]# pstree
init─┬─abrt-dump-oops
…省略部分输出
     ├─sshd─┬─sshd───bash───bash───pstree
```

知道了父 Shell 和子 Shell，我们接着解释小括号和大括号的区别。如果是用于一串命令的执行，那么小括号和大括号的主要区别在于：

- ()执行一串命令时，需要重新开一个子 shell 进行执行
- {}执行一串命令时，是在当前 shell 执行；
- ()和{}都是把一串的命令放在括号里面，并且命令之间用;号隔开；
- ()最后一个命令可以不用分号；
- {}最后一个命令要用分号；
- {}的第一个命令和左括号之间必须要有一个空格；
- ()里的各命令不必和括号有空格；
- ()和{}中括号里面的某个命令的重定向只影响该命令，但括号外的重定向则影响到括号里的所有命令。

还是举几个例子来看看吧，这样写实在是太抽象了：

```
[root@localhost ~]# name=sc
#在父 Shell 中定义变量 name 的值是 sc
[root@localhost ~]# (name=liming;echo $name)
liming
#如果用()括起来一串命令，这些命令都可以执行
#给 name 变量重新赋值，但是这个值只在子 Shell 中生效
[root@localhost ~]# echo $name
sc
#父 Shell 中 name 的值还是 sc，而不是 liming

[root@localhost ~]# { name=liming;echo $name; }
liming
#但是用大括号来进行一串命令的执行时，name 变量的修改是直接在父 Shell 当中的
#注意大括号的格式
[root@localhost ~]# echo $name
liming
#所以 name 变量的值已经被修改了
```

## 四、Bash 的变量和运算符

1　什么是变量

在定义变量时，有一些规则需要遵守：

- ✧ 变量名称可以由字母、数字和下划线组成，但是不能以数字开头。如果变量名是“2name”则是错误的。
- ✧ 在 Bash 中，变量的默认类型都是字符串型，如果要进行数值运算，则必修指定变量类型为数值型。
- ✧ 变量用等号连接值，等号左右两侧不能有空格。
- ✧ 变量的值如果有空格，需要使用单引号或双引号包括。如：“test="hello world!"”。其中双引号括起来的内容“$”、“\”和反引号都拥有特殊含义，而单引号括起来的内容都是普通字符。
- ✧ 在变量的值中，可以使用“\”转义符。
- ✧ 如果需要增加变量的值，那么可以进行变量值的叠加。不过变量需要用双引号包含"$变量名"或用${变量名}包含变量名。例如：

```
[root@localhost ~]# test=123
[root@localhost ~]# test="$test"456
[root@localhost ~]# echo $test
123456
#叠加变量 test，变量值变成了 123456
[root@localhost ~]# test=${test}789
[root@localhost ~]# echo $test
123456789
#再叠加变量 test，变量值编程了 123456789
```

变量值的叠加可以使用两种格式：“$变量名”或${变量名}

✧ 如果是把命令的结果作为变量值赋予变量，则需要使用反引号或$()包含命令。例如：

```
[root@localhost ~]# test=$(date)
[root@localhost ~]# echo $test
2018 年 10 月 21 日 星期一 20:27:50 CST
```

✧ 环境变量名建议大写，便于区分。

2、变量的分类

✧ 用户自定义变量：这种变量是最常见的变量，由用户自由定义变量名和变量的值。

✧ 环境变量：这种变量中主要保存的是和系统操作环境相关的数据，比如当前登录用户，用户的家目录，命令的提示符等。不是太好理解吧，那么大家还记得在 Windows 中，同一台电脑可以有多个用户登录，而且每个用户都可以定义自己的桌面样式和分辨率，这些其实就是 Windows 的操作环境，可以当做是 Windows 的环境变量来理解。环境变量的变量名可以自由定义，但是一般对系统起作用的环境变量的变量名是系统预先设定好的。

✧ 位置参数变量：这种变量主要是用来向脚本当中传递参数或数据的，变量名不能自定义，变量作用是固定的。

✧ 预定义变量：是 Bash 中已经定义好的变量，变量名不能自定义，变量作用也是固定的。

3、用户自定义变量

1）、变量定义

```
[root@localhost ~]# 2name="shen chao"
-bash: 2name=shen chao: command not found
#变量名不能用数字开头
[root@localhost ~]# name = "shenchao"
-bash: name: command not found
#等号左右两侧不能有空格
[root@localhost ~]# name=shen chao
-bash: chao: command not found
#变量的值如果有空格，必须用引号包含
```

2）、 变量调用

```
[root@localhost ~]# name="shen chao"
#定义变量 name
[root@localhost ~]# echo $name
shen chao
#输出变量 name 的值
```

3）、 变量查看

```
[root@localhost ~]# set [选项]
选项：
    -u:     如果设定此选项，调用未声明变量时会报错（默认无任何提示）
    -x:     如果设定此选项，在命令执行之前，会把命令先输出一次
```

```
[root@localhost ~]# set
BASH=/bin/bash
…省略部分输出…
name='shen chao'
#直接使用 set 命令，会查询系统中所有的变量，包含用户自定义变量和环境变量

[root@localhost ~]# set -u
[root@localhost ~]# echo $file
-bash: file: unbound variable
#当设置了-u 选项后，如果调用没有设定的变量会有报错。默认是没有任何输出的。

[root@localhost ~]# set -x
[root@localhost ~]# ls
+ ls --color=auto
anaconda-ks.cfg  install.log  install.log.syslog  sh  tdir  test  testfile
#如果设定了-x 选项，会在每个命令执行之前，先把命令输出一次
```

4）、 变量删除

```
[root@localhost ~]# unset 变量名
```

4 环境变量

1）、环境变量设置

```
[root@localhost ~]# export age="18"
#使用 export 声明的变量即是环境变量
```

2）、环境变量查询和删除

env 命令和 set 命令的区别是，set 命令可以查看所有变量，而 env 命令只能查看环境变量。

```
[root@localhost ~]# unset gender
[root@localhost ~]# env | grep gender
#删除环境变量 gender
```

3）、系统默认环境变量

```
[root@localhost ~]# env
HOSTNAME=localhost.localdomain          ←主机名
SHELL=/bin/bash                         ←当前的 shell
TERM=linux                              ←终端环境
HISTSIZE=1000                           ←历史命令条数
SSH_CLIENT=192.168.4.159 4824 22        ←当前操作环境是用 ssh 连接的，这里记录客户端 ip
SSH_TTY=/dev/pts/1                      ←ssh 连接的终端时 pts/1
USER=root                               ←当前登录的用户
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;
01:or=40;31;01:mi=01;05;37;41:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=
01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.taz=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01
```

```
;31:*.txz=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lz=01;31:*.xz=0
1;31:*.bz2=01;31:*.tbz=01;31:*.tbz2=01;31:*.bz=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31
:*.jar=01;31:*.rar=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.j
pg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga
=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=
01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01
;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35
:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.
flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=0
1;35:*.axv=01;35:*.anx=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=01;36:*.au=01;36:*.flac=01;3
6:*.mid=01;36:*.midi=01;36:*.mka=01;36:*.mp3=01;36:*.mpc=01;36:*.ogg=01;36:*.ra=01;36:*
.wav=01;36:*.axa=01;36:*.oga=01;36:*.spx=01;36:*.xspf=01;36:
    ←定义颜色显示
age=18                                    ←我们刚刚定义的环境变量
PATH=/usr/lib/qt-3.3/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/
root/bin                          ←系统查找命令的路径
MAIL=/var/spool/mail/root                 ←用户邮箱
PWD=/root                                 ←当前所在目录
LANG=zh_CN.UTF-8                          ←语系
HOME=/root                                ←当前登录用户的家目录
SHLVL=2                                   ←当前在第二层子 shell 中。还记得我们刚刚进入了
                                          一个子 shell 吗？如果是第一层 shell，这里是 1
LOGNAME=root                              ←登录用户
_=/bin/env                                ←上次执行命令的最后一个参数或命令本身
```

env 命令可以查询到所有的环境变量，可是还有一些变量虽然不是环境变量，却是和 Bash 操作接口相关的变量，这些变量也对我们的 Bash 操作终端起到了重要的作用。这些变量就只能用 set 命令来查看了，我只列出重要的内容吧：

```
[root@localhost ~]# set
BASH=/bin/bash                          ←Bash 的位置
BASH_VERSINFO=([0]="4" [1]="1" [2]="2" [3]="1" [4]="release"
 [5]="i386-redhat-linux-gnu")           ←Bash 版本
BASH_VERSION='4.1.2(1)-release'         ←bash 的版本
COLORS=/etc/DIR_COLORS                  ←颜色记录文件
HISTFILE=/root/.bash_history            ←历史命令保存文件
HISTFILESIZE=1000                       ←在文件当中记录的历史命令最大条数
HISTSIZE=1000                           ←在缓存中记录的历史命令最大条数
LANG=zh_CN.UTF-8                        ←语系环境
MACHTYPE=i386-redhat-linux-gnu          ←软件类型是 i386 兼容类型
MAILCHECK=60                            ←每 60 秒去扫描新邮件
PPID=2166                               ←父 shell 的 PID。我们当前 Shell 是一个子 shell
PS1='[\u@\h \W]\$ '                     ←命令提示符
PS2='> '                                ←如果命令一行没有输入完成，第二行命令的提示符
UID=0                                   ←当前用户的 UID
```

✧ PATH 变量：系统查找命令的路径

先查询下 PATH 环境变量的值：

```
[root@localhost ~]# echo $PATH
/usr/lib/qt-3.3/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin
```

PATH 变量的值是用“：”分割的路径，这些路径就是系统查找命令的路径。也就是说当我们输入了一个程序名，如果没有写入路径，系统就会到 PATH 变量定义的路径中去寻找，是否有可以执行的程序。如果找到则执行，否则会报“命令没有发现”的错误。

那么是不是我们把自己的脚本拷贝到 PATH 变量定义的路径中，我们自己写的脚本也可以不输入路径而直接运行呢？

```
[root@localhost ~]# cp /root/sh/hello.sh  /bin/
#拷贝 hello.sh 到/bin 目录
[root@localhost ~]# hello.sh
Mr. Shen Chao is the most honest man.
#hello.sh 可以直接执行了
```

那么我们是不是可以修改 PATH 变量的值，而不是把程序脚本复制到/bin/目录中。当然是可以的，我们通过变量的叠加就可以实现了：

```
[root@localhost ~]# PATH="$PATH":/root/sh
#在变量 PATH 的后面，加入/root/sh 目录
[root@localhost ~]# echo $PATH
/usr/lib/qt-3.3/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin:/root/sh
#查询 PATH 的值，变量叠加生效了
```

当然我们这样定义的 PATH 变量只是临时生效，一旦重启或注销就会消失，如果想要永久生效，需要写入环境变量配置文件，我们在“环境变量配置文件”小节中再详细介绍。

✧ PS1 变量：命令提示符设置

PS1 是一个很有意思的变量（这可不是 SONY 的游戏机哦），是用来定义命令行的提示符的，可以安装我们自己的需求来定义自己喜欢的提示符。PS1 可以支持以下这些选项：

- \d：显示日期，格式为“星期 月 日”
- \H：显示完整的主机名。如默认主机名“localhost.localdomain”
- \h：显示简写主机名。如默认主机名“localhost”
- \t：显示 24 小时制时间，格式为“HH:MM:SS”
- \T：显示 12 小时制时间，格式为“HH:MM:SS”
- \A：显示 24 小时制时间，格式为“HH:MM”
- \@：显示 12 小时制时间，格式为“HH:MM am/pm”
- \u：显示当前用户名
- \v：显示 Bash 的版本信息
- \w：显示当前所在目录的完整名称
- \W：显示当前所在目录的最后一个目录
- \#：执行的第几个命令
- \$：提示符。如果是 root 用户会显示提示符为“#”，如果是普通用户会显示提示符为

“$”

这些选项该怎么用啊？我们先看看 PS1 变量的默认值吧：

```
[root@localhost ~]# echo $PS1
[\u@\h \W]\$
#默认的提示符是显示“[用户名@简写主机名 最后所在目录]提示符”
```

在 PS1 变量中，如果是可以解释的符号，如“\u”、“\h”等，则显示这个符号的作用。如果是不能解释的符号，如“@”或“空格”，则原符号输出。那么我们修改下 PS1 变量，看看会出现社么情况吧：

```
[root@localhost ~]# PS1='[\u@\t \w]\$ '
#修改提示符为‘[用户名@当前时间 当前所在完整目录]提示符’
[root@04:46:40 ~]#cd /usr/local/src/
#切换下当前所在目录，因为家目录是看不出来区别的
[root@04:47:29 /usr/local/src]#
#看到了吗？提示符按照我们的设计发生了变化
```

这里要小心，PS1 变量的值要用单引号包含，否则设置不生效。再举个例子吧：

```
[root@04:50:08 /usr/local/src]#PS1='[\u@\@ \h \# \W]\$'
[root@04:53 上午 localhost 31 src]#
#提示符又变了。\@：时间格式是 HH:MM am/pm；\#：会显示执行了多少个命令
```

PS1 变量可以自由定制，好像看到了点 Linux 可以自由定制和修改的影子，还是很有意思的。不过说实话一个提示符已经使用习惯了，如果换一个还是非常别扭的，还是改回默认的提示符吧：

```
[root@04:53 上午 localhost 31 src]#PS1='[\u@\h \W]\$ '
[root@localhost src]#
```

✧ LANG 语系变量

LANG 变量定义了 Linux 系统的主语系环境，这个变量的默认值是：

```
[root@localhost src]# echo $LANG
zh_CN.UTF-8
```

这是因为我们 Linux 安装时，选择的是中文安装，所以默认的主语系变量是“zh_CN.UTF-8”。那么 Linux 中到底支持多少语系呢？我们可以使用以下命令查询：

```
[root@localhost src]# locale -a | more
aa_DJ
aa_DJ.iso88591
aa_DJ.utf8
aa_ER
…省略部分输出…
#查询支持的语系
[root@localhost src]# locale -a | wc -l
735
#是在太多了，统计一下有多少个吧
```

我们支持这么多的语系，当前系统到底是什么语系呢？使用 locale 命令直接查询：

```
[root@localhost src]# locale
LANG=zh_CN.UTF-8
LC_CTYPE="zh_CN.UTF-8"
```

```
LC_NUMERIC="zh_CN.UTF-8"
LC_TIME="zh_CN.UTF-8"
LC_COLLATE="zh_CN.UTF-8"
LC_MONETARY="zh_CN.UTF-8"
LC_MESSAGES="zh_CN.UTF-8"
LC_PAPER="zh_CN.UTF-8"
LC_NAME="zh_CN.UTF-8"
LC_ADDRESS="zh_CN.UTF-8"
LC_TELEPHONE="zh_CN.UTF-8"
LC_MEASUREMENT="zh_CN.UTF-8"
LC_IDENTIFICATION="zh_CN.UTF-8"
LC_ALL=
```

我们还要通过文件/etc/sysconfig/i18n 定义系统的默认语系，查看下这个文件的内容：

```
[root@localhost src]# cat /etc/sysconfig/i18n
LANG="zh_CN.UTF-8"
```

这又是当前系统语系，又是默认语系，有没有快晕倒的感觉。解释下吧，我们可以这样理解，默认语系是下次重启之后系统所使用的语系，而当前系统语系是当前系统使用的语系。如果系统重启，会从默认语系配置文件/etc/sysconfig/i18n 中读出语系，然后赋予变量 LANG 让这个语系生效。也就是说，LANG 定义的语系只对当前系统生效，要想永久生效就要修改/etc/sysconfig/i18n 文件了。

说到这里，我们需要解释下 Linux 中文支持的问题。是不是我们只要定义了语系为中文语系，如 zh_CN.UTF-8 就可以正确显示中文了呢？这要分情况，如果我们是在图形界面中，或者是使用远程连接工具（如 SecureCRT），只要正确设置了语系，那么是可以正确显示中文的。当然远程连接工具也要配置正确的语系环境，具体配置方式可以参考 Linux 系统安装章节。

那么如果是纯字符界面（本地终端 tty1-tty6）是不能显示中文的，因为 Linux 的纯字符界面时不能显示中文这么复杂的编码的。如果我们非要在纯字符界面显示中文，那么只能安装中文插件，如 zhcon 等。我们举个例子吧：

```
[root@localhost src]# echo $LANG
zh_CN.UTF-8
#我当前使用远程工具连接，只要语系正确，则可以正确显示中文
[root@localhost src]# df
文件系统                1K-块      已用      可用 已用% 挂载点
/dev/sda3             19923216   1813532  17097616  10% /
tmpfs                   312672         0    312672   0% /dev/shm
/dev/sda1               198337     26359    161738  15% /boot
#df 命令可以看到中文是正常的
```

但如果是纯字符界面呢？虽然我们是中文安装的，但纯字符界面的语系可是“en_US.UTF-8”，如图 11-2 所示：

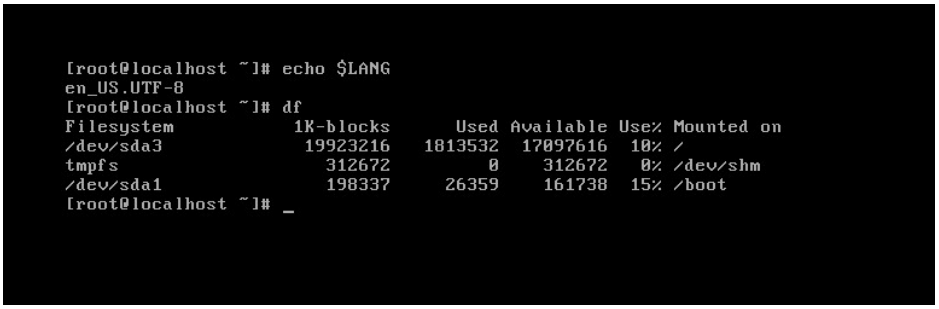



图 11-2 字符界面语系

那么我们更改下语系为中文，看看会出现什么情况吧：

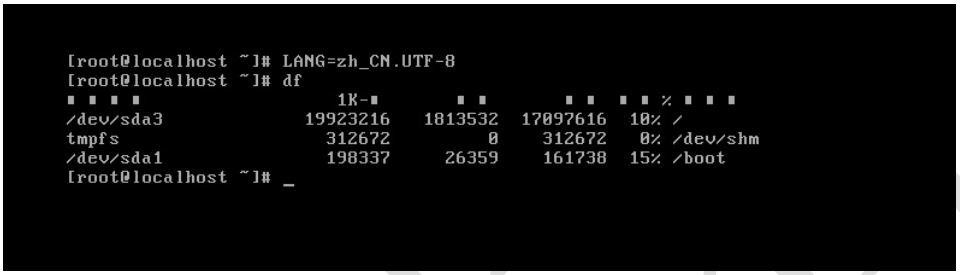



图 11-3 字符界面设置中文语系

如果我们非要在纯字符界面中设置中文语系，那么就会出现乱码。怎么解决呢？安装 zhcon 中文插件吧，安装并不复杂，查询下安装说明应该可以轻松安装。

5 位置参数变量

| 位置参数变量 | 作 用 |
|---|---|
| $n | n 为数字，$0 代表命令本身，$1-$9 代表第一到第九个参数，十以上的参数需要用大括号包含，如${10}. |
| $* | 这个变量代表命令行中所有的参数，$*把所有的参数看成一个整体 |
| $@ | 这个变量也代表命令行中所有的参数，不过$@把每个参数区分对待 |
| $# | 这个变量代表命令行中所有参数的个数 |

```
[root@localhost sh]# vi count.sh
#!/bin/bash
# Author: shenchao (E-mail: shenchao@atguigu.com)

num1=$1
#给 num1 变量赋值是第一个参数
num2=$2
#给 num2 变量赋值是第二个参数
sum=$(( $num1 + $num2))
#变量 sum 的和是 num1 加 num2
#Shell 当中的运算还是不太一样的，我们 Shell 运算符小节中详细介绍
echo $sum
#打印变量 sum 的值
```

那么还有几个位置参数变量是干嘛的呢？我们在写个脚本来说明下：

```
[root@localhost sh]# vi parameter.sh
#!/bin/bash
# Author: shenchao (E-mail: shenchao@atguigu.com)

echo "A total of $# parameters"
#使用$#代表所有参数的个数
echo "The parameters is: $*"
#使用$*代表所有的参数
echo "The parameters is: $@"
#使用$@也代表所有参数
```

那么“$*”和“$@”有区别吗？还是有区别的，$*会把接收的所有参数当成一个整体对待，而$@则会区分对待接收到的所有参数。还是举个例子：

```
[root@localhost sh]# vi parameter2.sh
#!/bin/bash
# Author: shenchao (E-mail: shenchao@atguigu.com)

for i in "$*"
#定义 for 循环，in 后面有几个值，for 会循环多少次，注意“$*”要用双引号括起来
#每次循环会把 in 后面的值赋予变量 i
#Shell 把$*中的所有参数看成是一个整体，所以这个 for 循环只会循环一次
        do
                echo "The parameters is: $i"
                #打印变量$i 的值
        done

x=1
#定义变量 x 的值为 1
for y in "$@"
#同样 in 后面的有几个值，for 循环几次，每次都把值赋予变量 y
#可是 Shell 中把“$@”中的每个参数都看成是独立的，所以“$@”中有几个参数，就会循环几次
        do
                echo "The parameter$x is: $y"
                #输出变量 y 的值
                x=$(( $x +1 ))
                #然变量 x 每次循环都加 1，为了输出时看的更清楚
        done
```

6 预定义变量

| 预定义变量 | 作　　用 |
|---|---|
| $? | 最后一次执行的命令的返回状态。如果这个变量的值为 0，证明上一个命令正确执行；如果这个变量的值为非 0（具体是哪个数，由命令自己来决定），则证明上一个命令执行不正确了。 |

| | |
|---|---|
| $$ | 当前进程的进程号（PID） |
| $! | 后台运行的最后一个进程的进程号（PID） |

我们先来看看“$?”这个变量，看起来不好理解，我们还是举个例子：

```
[root@localhost sh]# ls
count.sh  hello.sh  parameter2.sh  parameter.sh
#ls 命令正确执行
[root@localhost sh]# echo $?
0
#预定义变量“$?”的值是 0，证明上一个命令执行正确
[root@localhost sh]# ls install.log
ls: 无法访问 install.log: 没有那个文件或目录
#当前目录中没有 install.log 文件，所以 ls 命令报错了
[root@localhost sh]# echo $?
2
#变量“$?”返回一个非 0 的值，证明上一个命令没有正确执行
#至于错误的返回值到底是多少，是在编写 ls 命令时定义好的，如果碰到文件不存在就返回数值 2
```

接下来我们来说明下“$$”和“$!”这两个预定义变量，我们写个脚本吧：

```
[root@localhost sh]# vi variable.sh
#!/bin/bash
# Author: shenchao （E-mail: shenchao@atguigu.com）

echo "The current process is $$"
#输出当前进程的 PID。
#这个 PID 就是 variable.sh 这个脚本执行时，生成的进程的 PID

find /root -name hello.sh &
#使用 find 命令在 root 目录下查找 hello.sh 文件
#符号&的意思是把命令放入后台执行，工作管理我们在系统管理章节会详细介绍
echo "The last one Daemon process is $!"
#输出这个后台执行命令的进程的 PID，也就是输出 find 命令的 PID 号
```

7　接收键盘输入

```
[root@localhost ~]# read [选项] [变量名]
选项：
    -p “提示信息”：     在等待 read 输入时，输出提示信息
    -t 秒数：           read 命令会一直等待用户输入，使用此选项可以指定等待时间
    -n 字符数：         read 命令只接受指定的字符数，就会执行
    -s：                隐藏输入的数据，适用于机密信息的输入
变量名：
    变量名可以自定义，如果不指定变量名，会把输入保存入默认变量 REPLY
    如果只提供了一个变量名，则整个输入行赋予该变量
    如果提供了一个以上的变量名，则输入行分为若干字，一个接一个地赋予各个变量，而命令行上
```

的最后一个变量取得剩余的所有字

还是写个例子来解释下 read 命令：

```
[root@localhost sh]# vi read.sh
#!/bin/bash
# Author: shenchao （E-mail: shenchao@atguigu.com）

read -t 30 -p "Please input your name: " name
#提示“请输入姓名”并等待 30 秒，把用户的输入保存入变量 name 中
echo "Name is $name"
#看看变量“$name”中是否保存了你的输入

read -s -t 30 -p "Please enter your age: " age
#提示“请输入年龄”并等待 30 秒，把用户的输入保存入变量 age 中
#年龄是隐私，所以我们用“-s”选项隐藏输入
echo -e "\n"
#调整输出格式，如果不输出换行，一会的年龄输出不会换行
echo "Age is $age"


read -n 1 -t 30 -p "Please select your gender[M/F]: " gender
#提示“请选择性别”并等待 30 秒，把用户的输入保存入变量 gender
#使用“-n 1”选项只接收一个输入字符就会执行（都不用输入回车）
echo -e "\n"
echo "Sex is $gender"
```

8　Shell 的运算符

1）数值运算的方法

那如果我需要进行数值运算，可以采用以下三种方法中的任意一种：

✧　使用 declare 声明变量类型

既然所有变量的默认类型是字符串型，那么只要我们把变量声明为整数型不就可以运算了吗？使用 declare 命令就可以实现声明变量的类型。命令如下：

```
[root@localhost ~]# declare [+/-][选项] 变量名
选项：
    -:      给变量设定类型属性
    +:      取消变量的类型属性
    -a:     将变量声明为数组型
    -i:     将变量声明为整数型（integer）
    -r:     讲变量声明为只读变量。注意，一旦设置为只读变量，既不能修改变量的值，
            也不能删除变量，甚至不能通过+r 取消只读属性
    -x:     将变量声明为环境变量
    -p:     显示指定变量的被声明的类型
```

例子 1：数值运算

那我们只要把变量声明为整数型不就可以运算了吗？试试吧：

```
[root@localhost ~]# aa=11
[root@localhost ~]# bb=22
#给变量 aa 和 bb 赋值
[root@localhost ~]# declare -i cc=$aa+$bb
#声明变量 cc 的类型是整数型，它的值是 aa 和 bb 的和
[root@localhost ~]# echo $cc
33
#这下终于可以相加了
```

这样运算好麻烦啊，没有办法啊，Shell 在数值运算这里确实是比较麻烦，习惯就好了-_-!。

例子 2：数组变量类型

数组这个东东只有写一些较为复杂的程序才会用到，大家可以先不用着急学习数组，当有需要的时候再回来详细学习。那么数组是什么呢？所谓数组，就是相同数据类型的元素按一定顺序排列的集合，就是把有限个类型相同的变量用一个名字命名，然后用编号区分他们的变量的集合，这个名字称为数组名，编号称为下标。组成数组的各个变量成为数组的分量，也称为数组的元素，有时也称为下标变量。

一看定义就一头雾水，更加不明白数组是什么了。那么换个说法，变量和数组都是用来保存数据的，只是变量只能赋予一个数据值，一旦重复复制，后一个值就会覆盖前一个值。而数组是可以赋予一组相同类型的数据值。大家可以把变量想象成一个小办公室，这个办公室只能容纳一个人办公，办公室名就是变量名。而数组是一个大办公室，可以容纳很多人同时办公，在这个大办公室办公的每个人是通过不同的座位号来区分的，这个座位号就是数组的下标，而大办公室的名字就是数组名。

还是举个例子吧：

```
[root@localhost ~]# name[0]="shen chao"
#数组中第一个变量是沈超（大办公室第一个办公桌坐最高大威猛帅气的人）
[root@localhost ~]# name[1]="li ming"
#数组第二个变量是李明（大办公室第二个办公桌坐头发锃亮的人）
[root@localhost ~]# name[2]="tong gang"
#数组第三个变量是佟刚（大办公室第三个办公桌坐眼睛比超哥还小的老师）
[root@localhost ~]# echo ${name}
shen chao
#输出数组的内容，如果只写数组名，那么只会输出第一个下标变量
[root@localhost ~]# echo ${name[*]}
shen chao li ming tong gang
#输出数组所有的内容
```

注意数组的下标是从 0 开始的，在调用数组值时，需要使用${数组[下标]}的方式来读取。

不过好像在刚刚的例子中，我们并没有把 name 变量声明为数组型啊，其实只要我们在定义变量时采用了“变量名[下标]”的格式，这个变量就会被系统认为是数组型了，不用强制声明。

例子 3： 环境变量

我们其实也可以使用 declare 命令把变量声明为环境变量，和 export 命令的作用是一样的：

```
[root@localhost ~]# declare -x test=123
#把变量 test 声明为环境变量
```

例子 4：只读属性

注意一旦给变量设定了只读属性，那么这个变量既不能修改变量的值，也不能删除变量，甚至不能使用“+r”选项取消只读属性。命令如下：

```
[root@localhost ~]# declare -r test
#给 test 赋予只读属性
[root@localhost ~]# test=456
-bash: test: readonly variable
#test 变量的值就不能修改了
[root@localhost ~]# declare +r test
-bash: declare: test: readonly variable
#也不能取消只读属性
[root@localhost ~]# unset test
-bash: unset: test: cannot unset: readonly variable
#也不能删除变量
```

不过还好这个变量只是命令行声明的，所以只要重新登录或重启，这个变量就会消失了。

例子 5：查询变量属性和取消变量属性

变量属性的查询使用“-p”选项，变量属性的取消使用“+”选项。命令如下：

```
[root@localhost ~]# declare -p cc
declare -i cc="33"
#cc 变量是 int 型
[root@localhost ~]# declare -p name
declare -a name='([0]="shen chao" [1]="li ming" [2]="tong gang")'
#name 变量是数组型
[root@localhost ~]# declare -p test
declare -rx test="123"
#test 变量是环境变量和只读变量

[root@localhost ~]# declare +x test
#取消 test 变量的环境变量属性
[root@localhost ~]# declare -p test
declare -r test="123"
#注意，只读变量属性是不能取消的
```

✧ 使用 expr 或 let 数值运算工具

要想进行数值运算的第二种方法是使用 expr 命令，这种命令就没有 declare 命令复杂了。命令如下：

```
[root@localhost ~]# aa=11
[root@localhost ~]# bb=22
#给变量 aa 和变量 bb 赋值
[root@localhost ~]# dd=$(expr $aa + $bb)
#dd 的值是 aa 和 bb 的和。注意“+”号左右两侧必须有空格
```

```
[root@localhost ~]# echo $dd
33
```

使用 expr 命令进行运算时，要注意“+”号左右两侧必须有空格，否则运算不执行。

至于 let 命令和 expr 命令基本类似，都是 Linux 中的运算命令，命令格式如下：

```
[root@localhost ~]# aa=11
[root@localhost ~]# bb=22
#给变量 aa 和变量 bb 赋值
[root@localhost ~]# let ee=$aa+$bb
[root@localhost ~]# echo $ee
33
#变量 ee 的值是 aa 和 bb 的和

[root@localhost ~]# n=20
#定义变量 n
[root@localhost ~]# let n+=1
#变量 n 的值等于变量本身再加 1
[root@localhost ~]# echo $n
21
```

expr 命令和 let 命令大家可以按照习惯使用，不过 let 命令对格式要求要比 expr 命令宽松，所以推荐使用 let 命令进行数值运算。

✧ 使用“$((运算式))”或“$[运算式]”方式运算

其实这是一种方式“$(())”和“$[]”这两种括号按照个人习惯使用即可。命令如下：

```
[root@localhost ~]# aa=11
[root@localhost ~]# bb=22
[root@localhost ~]# ff=$(( $aa+$bb ))
[root@localhost ~]# echo $ff
33
#变量 ff 的值是 aa 和 bb 的和
[root@localhost ~]# gg=$[ $aa+$bb ]
[root@localhost ~]# echo $gg
33
#变量 gg 的值是 aa 和 bb 的和
```

这三种数值运算方式，大家可以按照自己的习惯来进行使用。不过我们推荐使用“$((运算式))”的方式

2）、 Shell 常用运算符

| 优先级 | 运算符 | 说明 |
|---|---|---|
| 13 | -, + | 单目负、单目正 |
| 12 | !, ~ | 逻辑非、按位取反或补码 |
| 11 | *, /, % | 乘、除、取模 |
| 10 | +, - | 加、减 |
| 9 | <<, >> | 按位左移、按位右移 |

| | | |
|---|---|---|
| 8 | < =, > =, < , > | 小于或等于、大于或等于、小于、大于 |
| 7 | == , != | 等于、不等于 |
| 6 | & | 按位与 |
| 5 | ^ | 按位异或 |
| 4 | \| | 按位或 |
| 3 | && | 逻辑与 |
| 2 | \|\| | 逻辑或 |
| 1 | =, +=, -=, *=, /=, %=, &=, ^=, \|=, <<=, >>= | 赋值、运算且赋值 |

运算符优先级表明在每个表达式或子表达式中哪一个运算对象首先被求值，数值越大优先级越高，具有较高优先级级别的运算符先于较低级别的运算符进行求值运算。

例子 1： 加减乘除

```
[root@localhost ~]# aa=$(( (11+3)*3/2 ))
#虽然乘和除的优先级高于加，但是通过小括号可以调整运算优先级
[root@localhost ~]# echo $aa
21
```

例子 2：取模运算

```
[root@localhost ~]# bb=$(( 14%3 ))
[root@localhost ~]# echo $bb
2
#14 不能被 3 整除，余数是 2
```

例子 3：逻辑与

```
[root@localhost ~]# cc=$(( 1 && 0 ))
[root@localhost ~]# echo $cc
0
#逻辑与运算只有想与的两边都是 1，与的结果才是 1，否则与的结果是 0
```

9 变量的测试与内容置换

| 变量置换方式 | 变量 y 没有设置 | 变量 y 为空值 | 变量 y 设置值 |
|---|---|---|---|
| x=${y-新值} | x=新值 | x 为空 | x=$y |
| x=${y:-新值} | x=新值 | x=新值 | x=$y |
| x=${y+新值} | x 为空 | x=新值 | x=新值 |
| x=${y:+新值} | x 为空 | x 为空 | x=新值 |
| x=${y=新值} | x=新值<br>y=新值 | x 为空<br>y 值不变 | x=$y<br>y 值不变 |
| x=${y:=新值} | x=新值<br>y=新值 | x=新值<br>y=新值 | x=$y<br>y 值不变 |
| x=${y?新值} | 新值输出到标准错误输出（就是屏幕） | x 为空 | x=$y |
| x=${y:?新值} | 新值输出到标准错误输出 | 新值输出到标准错误输出 | x=$y |

如果大括号内没有“：”，则变量 y 是为空，还是没有设置，处理方法是不同的；如果大括号内

有“：”，则变量 y 不论是为空，还是没有没有设置，处理方法是一样的。

如果大括号内是“-”或“+”，则在改变变量 x 值的时候，变量 y 是不改变的；如果大括号内是“=”，则在改变变量 x 值的同时，变量 y 的值也会改变。

如果大括号内是“？”，则当变量 y 不存在或为空时，会把“新值”当成报错输出到屏幕上。

看的头都晕了吧，举几个例子说明下吧：

例子 1：

```
[root@localhost ~]# unset y
#删除变量 y
[root@localhost ~]# x=${y-new}
#进行测试
[root@localhost ~]# echo $x
new
#因为变量 y 不存在，所以 x=new
[root@localhost ~]# echo $y

#但是变量 y 还是不存在的
```

和表 11-12 对比下，这个表是不是可以看懂了。这是变量 y 不存在的情况，那如果变量 y 的值是空呢？

```
[root@localhost ~]# y=""
#给变量 y 赋值为空
[root@localhost ~]# x=${y-new}
#进行测试
[root@localhost ~]# echo $x

[root@localhost ~]# echo $y

#变量 x 和变量 y 值都是空
```

那如果变量 y 有值呢？

```
[root@localhost ~]# y=old
#给变量 y 赋值
[root@localhost ~]# x=${y-new}
#进行测试
[root@localhost ~]# echo $x
old
[root@localhost ~]# echo $y
old
#变量 x 和变量 y 的值都是 old
```

例子 2：

那如果大括号内是“=”号，又该是什么情况呢？先测试下变量 y 没有设置的情况：

```
[root@localhost ~]# unset y
#删除变量 y
[root@localhost ~]# x=${y:=new}
#进行测试
```

```
[root@localhost ~]# echo $x
new
[root@localhost ~]# echo $y
new
#变量 x 和变量 y 的值都是 new
```

一旦使用“=”号，那么变量 y 和变量 x 都会同时进行处理，而不像例子 1 中只改变变量 x 的值。那如果变量 y 为空又是什么情况呢？

```
[root@localhost ~]# y=""
#设定变量 y 为空
[root@localhost ~]# x=${y:=new}
#进程测试
[root@localhost ~]# echo $x
new
[root@localhost ~]# echo $y
new
#变量 x 和变量 y 的值都是 new
```

一旦在大括号中使用“:”，那么变量 y 为空或者不设定，处理方式都是一样的了。那如果 y 已经赋值了，又是什么情况：

```
[root@localhost ~]# y=old
#给 y 赋值
[root@localhost ~]# x=${y:=new}
#进行测试
[root@localhost ~]# echo $x
old
[root@localhost ~]# echo $y
old
#原来变量 x 和变量 y 的值都是 old
```

例子 3：

再测试下大括号中是“？”的情况吧：

```
[root@localhost ~]# unset y
#删除变量 y
[root@localhost ~]# x=${y?new}
-bash: y: new
#会把值“new”输出到屏幕上
```

那如果变量 y 已经赋值了呢：

```
[root@localhost ~]# y=old
#给变量 y 赋值
[root@localhost ~]# x=${y?new}
#进行测试
[root@localhost ~]# echo $x
old
[root@localhost ~]# echo $y
old
```

```
#变量 x 和变量 y 的值都是 old
```

# 五 环境变量配置文件

1 source 命令

```
[root@localhost ~]# source 配置文件
或
[root@localhost ~]# . 配置文件
```

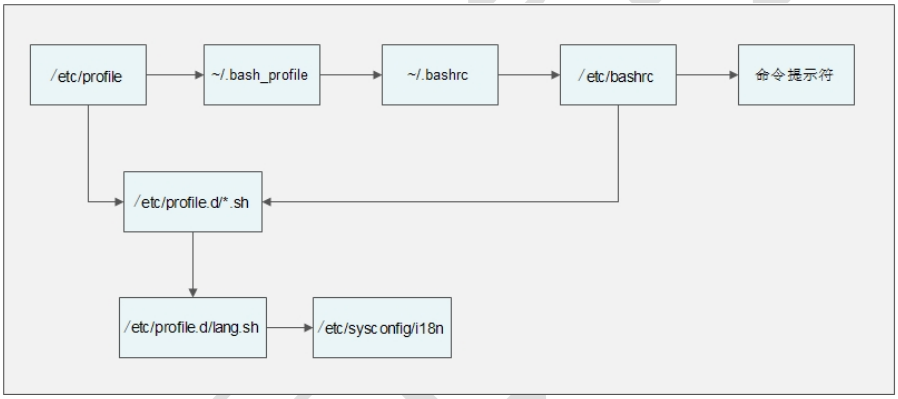
2 环境变量配置文件

1）、 登录时生效的环境变量配置文件

在 Linux 系统登录时主要生效的环境变量配置文件有以下五个：

- ✧ /etc/profile
- ✧ /etc/profile.d/*.sh
- ✧ ~/.bash_profile
- ✧ ~/.bashrc
- ✧ /etc/bashrc

环境变量配置文件调用过程




✧ 在用户登录过程先调用/etc/profile 文件

在这个环境变量配置文件中会定义这些默认环境变量：

- ➢ USER 变量：根据登录的用户，给这个变量赋值（就是让 USER 变量的值是当前用户）。
- ➢ LOGNAME 变量：根据 USER 变量的值，给这个变量赋值。
- ➢ MAIL 变量：根据登录的用户，定义用户的邮箱为/var/spool/mail/用户名。
- ➢ PATH 变量：根据登录用户的 UID 是否为 0，判断 PATH 变量是否包含/sbin、/usr/sbin 和/usr/local/sbin 这三个系统命令目录。
- ➢ HOSTNAME 变量：更加主机名，给这个变量赋值。
- ➢ HISTSIZE 变量：定义历史命令的保存条数。
- ➢ umask：定义 umask 默认权限。注意/etc/profile 文件中的 umask 权限是在“有用户登录过程（也就是输入了用户名和密码）”时才会生效。
- ➢ 调用/etc/profile.d/*.sh 文件，也就是调用/etc/profile.d/目录下所有以.sh 结尾的文件。

✧ 由/etc/profile 文件调用/etc/profile.d/*.sh 文件

这个目录中所有以.sh 结尾的文件都会被/etc/profile 文件调用，这里最常用的就是 lang.sh 文件，而这个文件又会调用/etc/sysconfig/i18n 文件。/etc/sysconfig/i18n 这个文件眼熟吗？就是我们前面讲过的默认语系配置文件啊。

✧ 由/etc/profile 文件调用~/.bash_profile 文件

~/.bash_profile 文件就没有那么复杂了，这个文件主要实现了两个功能：

- 调用了~/.bashrc 文件。
- 在 PATH 变量后面加入了“:$HOME/bin”这个目录。那也就是说，如果我们在自己的家目录中建立 bin 目录，然后把自己的脚本放入“~/bin”目录，就可以直接执行脚本，而不用通过目录执行了。

✧ 由~/.bash_profile 文件调用~/.bashrc 文件

在~/.bashrc 文件中主要实现了：

- 定义默认别名，所以超哥把自己定义的别名也放入了这个文件。
- 调用/etc/bashrc

✧ 由~/.bashrc 调用了/etc/bashrc 文件

在/etc/bashrc 文件中主要定义了这些内容：

- PS1 变量：也就是用户的提示符，如果我们想要永久修改提示符，就要在这个文件中修改
- umask：定义 umask 默认权限。这个文件中定义的 umask 是针对“没有登录过程（也就是不需要输入用户名和密码时，比如从一个终端切换到另一个终端，或进入子 Shell）”时生效的。如果是“有用户登录过程”，则是/etc/profile 文件中的 umask 生效。
- PATH 变量：会给 PATH 变量追加值，当然也是在“没有登录过程”时才生效。
- 调用/etc/profile.d/*.sh 文件，这也是在“没有用户登录过程”是才调用。在“有用户登录过程”时，/etc/profile.d/*.sh 文件已经被/etc/profile 文件调用过了。

这样这五个环境变量配置文件会被依次调用，那么如果是我们自己定义的环境变量应该放入哪个文件呢？如果你的修改是打算对所有用户生效的，那么可以放入/etc/profile 环境变量配置文件；如果你的修改只是给自己使用的，那么可以放入~/.bash_profile 或~/.bashrc 这两个配置文件中的任一个。

可是如果我们误删除了这些环境变量，比如删除了/etc/bashrc 文件，或删除了~/.bashrc 文件，那么这些文件中配置就会失效（~/.bashrc 文件会调用/etc/bashrc 文件）。那么我们的提示符就会变成：

```
-bash-4.1#
```

2）、 注销时生效的环境变量配置文件

在用户退出登录时，只会调用一个环境变量配置文件，就是~/.bash_logout。这个文件默认没有写入任何内容，可是如果我们希望再退出登录时执行一些操作，比如清除历史命令，备份某些数据，就可以把命令写入这个文件。

3）、 其他配置文件

还有一些环节变量配置文件，最常见的就是~/bash_history 文件，也就是历史命令保存文件。这

个文件已经讲过了，这里我们只是把它归入环境变量配置文件小节而已。

3　Shell 登录信息

1)、　/etc/issue

我们在登录 tty1-tty6 这六个本地终端时，会有几行的欢迎界面。这些欢迎信息是保存在哪里的？可以修改吗？当然可以修改，这些欢迎信息是保存在/etc/issue 文件中，我们查看下这个文件：

```
[root@localhost ~]# cat /etc/issue
CentOS release 6.8 (Final)
Kernel \r on an \m
```

可以支持的转义符我们可以通过 man agetty 命令查询，在表中我们列出常见的转义符作用：

| 转义符 | 作　　用 |
|---|---|
| \d | 显示当前系统日期 |
| \s | 显示操作系统名称 |
| \l | 显示登录的终端号，这个比较常用。 |
| \m | 显示硬件体系结构，如 i386、i686 等 |
| \n | 显示主机名 |
| \o | 显示域名 |
| \r | 显示内核版本 |
| \t | 显示当前系统时间 |
| \u | 显示当前登录用户的序列号 |

2)、　/etc/issue.net

/etc/issue 是在本地终端登录是显示欢迎信息的，如果是远程登录（如 ssh 远程登录，或 telnet 远程登录）需要显示欢迎信息，则需要配置/etc/issue.net 这个文件了。使用这个文件时由两点需要注意：

- ✧ 首先，在/etc/issue 文件中支持的转义符，在/etc/issue.net 文件中不能使用。
- ✧ 其次，ssh 远程登录是否显示/etc/issue.net 文件中的欢迎信息，是由 ssh 的配置文件决定的。

如果我们需要 ssh 远程登录可以查看/etc/issue.net 的欢迎信息，那么首先需要修改 ssh 的配置文件/etc/ssh/sshd_config，加入如下内容：

```
[root@localhost ~]# cat /etc/ssh/sshd_config
…省略部分输出…
# no default banner path
#Banner none
Banner /etc/issue.net
…省略部分输出…
```

这样在 ssh 远程登录时，也可以显示欢迎信息，只是不再可以识别“\d”和“\l”等信息了

3)、　/etc/motd

/etc/motd 文件中也是显示欢迎信息的，这个文件和/etc/issue 及/etc/issue.net 文件的区别是：/etc/issue 及/etc/issue.net 是在用户登录之前显示欢迎信息，而/etc/motd 是在用户输入用户名和密码，正确登录之后显示欢迎信息。在/etc/motd 文件中的欢迎信息，不论是本地登录，还是远程登录都可以显示。

4　定义 Bash 快捷键

```
[root@localhost ~]# stty -a
#查询所有的快捷键
```

那么这些快捷键可以更改吗？可以啊，只要执行：

```
[root@localhost ~]# stty 关键字 快捷键

例如：
[root@localhost ~]# stty intr ^p
#定义 ctrl+p 快捷键为强制终止，“^”字符只要手工输入即可
[root@localhost ~]# stty -a
speed 38400 baud; rows 21; columns 104; line = 0;
intr = ^P; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>; eol2 = <undef>; swtch
= <undef>;
start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O; min =
1; time = 0;
#强制终止变成了 ctrl+p 快捷键
```