Linux Shell简明教程

作者: schama

前言

写于: 2013/12

shell也叫做命令行界面，它是*nix操作系统下传统的用户和计算机的交互界面。用户直接输入命令来执行各种各样的任务。当然微软的Windows操作系统也提供了这样的功能，它们是Windows 95/98下的command.com、和基于Windows NT的各种系统下的cmd.exe。

*nix操作系统下的shell既是用户交互的界面，也是控制系统的脚本语言。当然在这点也有别于Windows下的命令行，虽然也提供了很简单的控制语句。在Windows操作系统下，可能有些用户从来都不会直接的使用shell，然而在Unix系列操作系统下，shell仍然是控制系统启动、X Window启动和很多其他实用工具的脚本解释程序。

bash是一个为GNU计划编写的Unix shell(本文主要基于bash)。它的名字是一系列缩写：Bourne-Again SHell—这是关于Bourne shell（sh）的一个双关语（Bourne again / born again）。Bourne shell是一个早期的重要shell，由史蒂夫·伯恩在1978年前后编写，并同Version 7 Unix一起发布。bash则在1987年由布莱恩·福克斯创造。在1990年，Chet Ramey成为了主要的维护者。

1. 变量

认识

shell变量是一种很"弱"的变量，默认情况下，一个变量保存一个串，shell不关心这个串是什么含义。所以若要进行数学运算，必须使用一些命令例如`let`、`declare`、`expr`、双括号等。

shell变量可分为两类：局部变量和环境变量。局部变量只在创建它们的shell中可用。而环境变量则可以在创建它们的shell及其派生出来的任意子进程中使用。

变量名必须以字母或下划线字符开头。其余的字符可以是字母、数字(0~9)或下划线字符。

任何其他的字符都标志着变量名的终止。名字是大小写敏感的。

给变量赋值时，等号周围不能有任何空白符。为了给变量赋空值，可以在等号后跟一个换行符。

用`set`命令可以查看所有的变量，`unset var`命令可以清除变量var，var相当于没有定义过。

`readonly var`可以把var变为只读变量，定义之后不能对var进行任何更改。

变量的引用

引用格式	返回值及用法举例
$var	返回变量，例如:var="Schama", 则$var既是Schama。注意""是定界符，而不是字符串
${var}	返回变量值， 推荐这种写法，因为代码的可读更好
${#var}	返回变量字符串长度
${var:start_index}	返回从start_index位置开始到末尾的字符
${var:start_index:length}	返回从start_index位置开始length个字符

引用格式	返回值及用法举例

${var#string}	返回从左边删除string后的字符串，尽量短的去匹配。 例如:
	`var="http://127.0.0.1/index.html"`
	`echo ${var#*/}`
	`/127.0.0.1/index.html`

${var##string}	返回从左边删除string后的字符串，尽量长的去匹配。 例如:
	`var="http://127.0.0.1/index.html"`
	`echo ${var##*/}`
	`index.html`

${var%string}	返回从右边删除string后的字符串，尽量短的去匹配。 例如:
	`var="http://127.0.0.1/index.html"`
	`echo ${var%/*}`
	`http://127.0.0.1`

${var%%string}	返回从右边删除string后的字符串，尽量长的去匹配。 例如:
	`var="http://127.0.0.1/index.html"`
	`echo ${var%%/*}`
	`http:`

${var:-newstring}	如果var为空值或者未定义，则返回newstring。如果var不为空，则返回原值
${var:=newstring}	如果var为空值或者未定义，则返回newstring，并把newstring复制给var。如果var不为空，则返回原值
${var:+newstring}	如果var不为空值，则返回newstring。如果var为空，则返回空值

引用格式	返回值及用法举例
${var:?newstring}	如果var为空值或者未定义，则将newstring写入标准错误流，本语句失败。如果var为空，则返回空值
${var:/substring/newstring}	返回var中第一个substring被替换成newstring字符串。 例如:`var=08880`，则`${var/0/Schama}`返回`Schama8880`
${var://substring/newstring}	返回var中所有的substring被替换成newstring字符串。 例如:`var=08880`，则`${var//0/Schama}`返回`Schama888Schama`
$(command)	返回command命令执行后所输出的结果。 例如`$(date)`返回`date`命令执行后的输出，相当于`date`
$((算术表达式))	返回双括号内算术运算结果。

环境变量

环境变量的定义方法如下：

```
$ var=value
$ export var
```

shell在初始化的时候会在执行profile等初始化脚本，脚本中定义了一些环境变量，这些变量会在创建子进程时传递给子进程。

用`env`命令可以查看当前的环境变量。

常用的系统环境变量如下：

环境变量	说明
_(下划线)	上一条命令的最后一个参数
BASH	展开为调用bash实例时使用的全路径名
CDPATH	cd命令的搜索路径。它是以冒号分隔的目录列表，shell通过它来搜索cd命令指定的目标目录。例如.:~:/usr
EDITOR	内置编辑器emacs、gmacs或vi的路径名
ENV	每一个新的bash shell(包括脚本)启动时执行的环境文件。通常赋予这个变量的文件名是.bashrc。
EUID	展开为在shell启动时被初始化的当前用户的有效ID
GROUPS	当前用户所属的组
HISTFILE	指定保存命令行历史的文件。默认值是~/.bash_history。如果被复位，交互式shell退出时将不保存命令行历史
HISTSIZE	记录在命令行历史文件中的命令数。默认是500
HOME	主目录。未指定目录时，cd命令将转向该目录
IFS	内部字段分隔符，一般是空格符、制表符和换行符，用于由命令替换，循环结构中的表和读取的输入产生的词的字段划分
LANG	用来为没有以LC_开头的变量明确选取的种类确定locale类
OLDPWD	前一个工作目录
PATH	命令搜索路径。一个由冒号分隔的目录列表，shell用它来搜索命令，一个普通值为 /usr/gnu/bin:/usr/local/bin:/usr/ucb:/usr/bin
PPID	父进程的进程ID
PS1	主提示符串，默认值是$

环境变量	说明
PS2	次提示符串，默认值是>
PS3	与select命令一起使用的选择提示符串，默认值是#？
PS4	当开启追踪时使用的调试提示符串，默认值是+。追踪可以用set –x开启
PWD	当前工作目录。由cd设置
RANDOM	每次引用该变量，就产生一个随机整数。随机数序列可以通过给RANDOM赋值来初始化。如果RANDOM被复位，即使随后再设置，它也将失去特定的属性
REPLY	当没有给read提供参数时设置
SHELL	当调用shell时，它扫描环境变量以寻找该名字。shell给PATH、PS1、PS2、MAILCHECK和IFS设置默认值。HOME和MAIL由login(1)设置
SHELLOPTS	包含一列开启的shell选项，比如braceexpand、hashall、monitor等
UID	展开为当前用户的用户ID，在shell启动时初始化

数值变量

shell中默认把变量值当作字符串，例如：

```
$ age=22
$ age=${age}+1
$ echo ${age}
22+1
$
```

输出结果为22+1，而不是23，因为shell将其解释为字符串，而不是数学运算。

可以用let命令使其进行数学运算，例如：

```
let age=${age}+1
```

也可以用declare把变量定义为整型。例如：

```
declare -i age=22
```

这里就用 -i 选项把age定义为整型的了。此后每次运算，都把age的右值识别为算术表达式或数字。

数组

在shell中可以使用数组，例如：

```
array[0]=0
array[1]=1
array[2]=2
```

则array就是一个数组，也可以这样给数组初始化：

```
array=(0 1 2) # 元素之间以空格分隔
```

可以通过

命令	备注
`${array[$i]}`	来访问array中某个元素
`${array[*]}`	返回值即数组的所有元素组成的串
`${#array[*]}`	的返回值即数组的元素个数
`${array[*]:0:2}`	返回第一个和第二个元素组成的串 0表示开始的位置， 2表示要返回的元素个数， 开始位置可以为0-2(0减去2)之类的， 表示从倒数第二个元素开始。

下面写个稍微复杂点的例子：

```
#!/usr/bin/env bash
for ((i=0; i<100; i++))
do
    array[$i]=$i
done

for ((i=0; i<100; i++))
do
    echo ${array[$i]}
done
```

如果要使用二维数组甚至三维数组该怎么实现呢，那就需要用`eval`命令来模拟数组的功能了。

eval

`eval`命令的作用是扫描命令两次再执行，如果不使用eval，只扫描一次，然后执行。

看个例子：

```
$ name=Schama
$ $name=hello
Schama=hello: command not found
```

为什么第二句给Schama变量赋值会出错呢？

从报错信息可以发现shell并没有识别这是个赋值语句，而是把`Schama=hello`当作一个命令来执行，当然会报错。

为什么不能识别这是赋值语句呢？

第一次扫描时，因为扫描到$符号，所以不能把这句当作赋值语句，赋值语句的左边总是一个变量名，而不应该是$开头的。所以第一次扫描仅仅识别了$name变量，并做了替换，而并没有认识到赋值语句。

如果使用`eval $name=hello`呢？

```
$ name=Schama
$ $name=hello
Schama=hello: command not found
$ eval $name=hello
$ echo $Schama
hello
```

可见使用了eval之后，对 $name=hello 第一次扫描替换了$name，没有识别赋值语句，第二次扫描识别是赋值语句，然后执行。现在大约可以想到怎样用eval实现二维数组了。

下面实现的二维数组每一行代表一个人的信息记录，包括姓名，年龄。

```
#!/usr/bin/env bash
for ((i=0; i<2; i++))
do
    for ((j=0; j<2; j++))
    do
        read man$i$j
    done
done

echo "next print:"
for ((i=0; i<2; i++))
do
    for ((j=0; j<2; j++))
    do
        eval echo -n "\$man$i$j:"
    done
printf "\n"
done
```

特殊变量

变量	说明
$0	当前脚本的文件名
$num	num为从1开始的数字，$1是第一个参数，$2是第二个参数，${10}是第十个参数
$#	传入脚本的参数的个数
$*	所有的位置参数(作为单个字符串)
$@	所有的位置参数(每个都作为独立的字符串)。
$?	当前shell进程中，上一个命令的返回值，如果上一个命令成功执行则$?的值为0，否则为其他非零值，常用做if语句条件
$$	当前shell进程的pid
$!	后台运行的最后一个进程的pid
$-	显示shell使用的当前选项
$_	之前命令的最后一个参数

单引号和双引号

```
$ a="hello world"
$ echo '$a'
$a
$ echo "$a"
hello world
```

从这个实例我们可以看出，单引号内的字符原样输出，双引号不会屏蔽变量和特殊符号的转义。

括号

- 单圆括号()

1. 命令组： 括号中的命令将会新开一个子shell顺序执行，所以括号中的变量不能够被脚本余下的部分使用。括号中多个命令之间用分号隔开，最后一个命令可以没有分号，各命令和括号之间不必有空格。
2. 命令替换： 等同于`cmd`，shell扫描一遍命令行，发现了$(cmd)结构，便将$(cmd)中的cmd执行一次，得到其标准输出，再将此输出放到原来命令。有些shell不支持，如tcsh。
3. 用于初始化数组。 如：array=(a b c d)

- 双圆括号(())

1. 整数扩展: 这种扩展计算是整数型的计算，不支持浮点型。((exp))结构扩展并计算一个算术表达式的值，如果表达式的结果为0，那么返回的退出状态码为1，或者是"假"，而一个非零值的表达式所返回的退出状态码将为0，或者是"true"。若是逻辑判断，表达式exp为真则为1,假则为0。
2. 要括号中的运算符、表达式符合C语言运算规则，都可用在$((exp))中，甚至是三目运算符。作不同进位(如二进制、八进制、十六进制)运算时，输出结果全都自动转化成了十进制。如：`echo $((16#5f))` 结果为95(16进位转十进制)
3. 单纯用(( )) 也可重定义变量值，比如 `a=5; ((a++))` 可将 `$a` 重定义为6
4. 双括号中的变量可以不使用$符号前缀。括号内支持多个表达式用逗号分开。

- 单方括号[]

1. bash 的内部命令，[和test是等同的。如果我们不用绝对路径指明，通常我们用的都是bash自带的命令。if/test结构中的左中括号是调用test的命令标识，右中括号是关闭条件判断的。这个命令把它的参数作为比较表达式或者作为文件测试，并且根据比较的结果来返回一个退出状态码。if/test结构中并不是必须右中括号，但是新版的Bash中要求必须这样。
2. Test和[]中可用的比较运算符只有==和!=，两者都是用于字符串比较的，不可用于整数比较，整数比较只能使用-eq，-gt这种形式。无论是字符串比较还是整数比较都不支持大于号小于号。如果实在想用，对于字符串比较可以使用转义形式，如果比较"ab"和"bc"：[ab \< bc]，结果为真，也就是返回状态为0。[ ]中的逻辑与和逻辑或使用-a 和-o 表示。
3. 字符范围。用作正则表达式的一部分，描述一个匹配的字符范围。作为test用途的中括号内不能使用正则。
4. 在一个array结构的上下文中，中括号用来引用数组中每个元素的编号。

5. bash测试和比较函数参考
   http://www.ibm.com/developerworks/cn/linux/l-bash-test.html

- 双方括号[[]]

1. [[是bash程序语言的关键字。并不是一个命令， 结构比[ ]结构更加通用。在和之间所有的字符都不会发生文件名扩展或者单词分割，但是会发生参数扩展和命令替换。
2. 支持字符串的模式匹配，使用 `=~` 操作符时甚至支持shell的正则表达式。字符串比较时可以把右边的作为一个模式，而不仅仅是一个字符串，比如 `[[ hello == hell? ]]` ，结果为真。 中匹配字符串或通配符，不需要引号。
3. 使用[[...]]条件判断结构，而不是[...]，能够防止脚本中的许多逻辑错误。比如，&&、||、 <和> 操作符能够正常存在于条件判断结构中，但是如果出现在[ ]结构中的话，会报错。
4. bash把双中括号中的表达式看作一个单独的元素，并返回一个退出状态码。

- 花括号{}

1. 大括号拓展： (通配(globbing))将对大括号中的文件名做扩展。在大括号中，不允许有空白，除非这个空白被引用或转义。第一种：对大括号中的以逗号分割的文件列表进行拓展。如 `touch {a,b}.txt` 结果为a.txt b.txt。第二种：对大括号中以点点（..）分割的顺序文件列表起拓展作用，如： `touch {a..d}.txt` 结果为a.txt b.txt c.txt d.txt
2. 代码块： 又被称为内部组，这个结构事实上创建了一个匿名函数 。与小括号中的命令不同，大括号内的命令不会新开一个子shell运行，即脚本余下部分仍可使用括号内变量。括号内的命令间用分号隔开，最后一个也必须有分号。{}的第一个命令和左括号之间必须要有一个空格。
3. 特殊用法： ${var:-string}、${var#/}、*${var%%.}*、${var:5:5}文档开头已经介绍。

- 实例

```
$ if ($i<5)
$ if [ $i -lt 5 ]
$ if [ $a -ne 1 -a $a != 2 ]
$ if [ $a -ne 1] && [ $a != 2 ]
$ if [[ $a != 1 && $a != 2 ]]

$ for i in $(seq 0 4);do echo $i;done
```

```
$ for i in `seq 0 4`;do echo $i;done
$ for ((i=0;i<5;i++));do echo $i;done
$ for i in{0..4};do echo $i;done
```

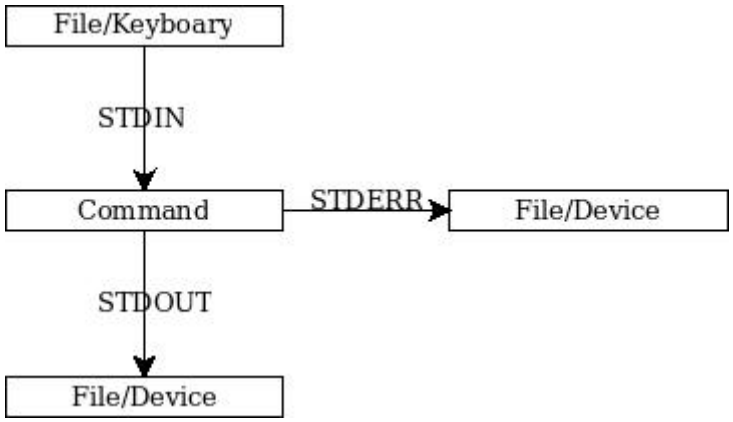
2. I/O

linux文件描述符，可以理解为linux跟踪打开文件，而分配的一个数字，这个数字有点类似c语言操作文件时候的句柄，通过句柄就可以实现文件的读写操作。 用户可以自定义文件描述符范围是：3-num,个最大数字，跟用户的：ulimit –n 定义数字有关系，不能超过最大值。

linux启动后，会默认打开3个文件描述符，分别是：
- 标准输入：standard input 0
- 正确输出：standard output 1
- 错误输出：error output 2

以后打开文件后。新增文件绑定描述符 可以依次增加。 一条shell命令执行，都会继承父进程的文件描述符。因此，所有运行的shell命令，都会有默认3个文件描述符。

对于任何一条linux 命令执行，它会是这样一个过程：



一个命令执行了：

- 先有一个输入：输入可以从键盘，也可以从文件得到
- 命令执行完成：成功了，会把成功结果输出到屏幕：standard output默认是屏幕
- 命令执行有错误：会把错误也输出到屏幕上面：standard error默认也是指的屏幕

文件输入输出由追踪为一个给定的进程所有打开文件的整数句柄来完成。这些数字值就是文件描述符。最为人们所知的文件描述符是 stdin, stdout 和 stderr，文件描述符的数字分别是0，1和2。这些数字和各自的设备是保留的。一个命令

执行前，先会准备好所有输入输出，默认分别绑定（stdin,stdout,stderr)，如果这个时候出现错误，命令将终止，不会执行。

这些默认的输出，输入都是linux系统内定的，我们在使用过程中，有时候并不希望执行结果输出到屏幕。我想输出到文件或其它设备。这个时候我们就需要进行输出重定向了。

3. 指令

echo

功能说明： 显示文字。
语　　法： echo [-ne][字符串]或 echo [--help][--version]
补充说明： echo会将输入的字符串送往标准输出。输出的字符串间以空白字符隔开, 并在最后加上换行号。
参　　数：

选项	说明
-n	不要在最后自动换行
-e	若字符串中出现以下字符，则特别加以处理，而不会将它当成一般文字输出：
	\a 发出警告声
	\b 删除前一个字符
	\c 最后不加上换行符号
	\f 换行但光标仍旧停留在原来的位置
	\n 换行且光标移至行首
	\r 光标移至行首，但不换行
	\t 插入tab
	\v 与\f相同
	\\ 插入\字符
	\nnn 插入nnn（八进制）所代表的ASCII字符
–help	显示帮助
–version	显示版本信息

read

功能说明： 接收标准输入

语　　法： read [-ers] [-a 数组] [-d 分隔符] [-i 缓冲区文字] [-n 读取字符数] [-N 读取字符数] [-p 提示符] [-t 超时] [-u 文件描述符] [名称 ...]

参　　数：

选项	说明
-a array	将词语赋值给 ARRAY 数组变量的序列下标成员，从零开始
-d delim	持续读取直到读入 DELIM 变量中的第一个字符，而不是换行符
-e	在一个交互式 shell 中使用 readline 获取行
-i text	使用 TEXT 文本作为 readline 的初始文字
-n nchars	读取 nchars 个字符之后返回，而不是等到读取换行符，但是分隔符仍然有效，如果遇到分隔符之前读取了不足 nchars 个字符
-N nchars	在准确读取了 nchars 个字符之后返回，除非遇到了文件结束符或者读超时，任何的分隔符都被忽略
-p prompt	在尝试读取之前输出 PROMPT 提示符并且不带换行符
-r	不允许反斜杠转义任何字符
-s	不显示终端的任何输入
-t timeout	如果在 TIMEOUT 秒内没有读取一个完整的行则超时并且返回失败。TMOUT 变量的值是默认的超时时间。TIMEOUT 可以是小数。如果 TIMEOUT 是0，那么仅当在指定的文件描述符上输入有效的时候，read 才返回成功。如果超过了超时时间，则返回状态码大于128
-u fd	从文件描述符 FD 中读取，而不是标准输入

cat

功能说明： 连接文件并在标准输出上输出
语　　法： cat [-AbeEnstTuv] [--help] [--version] fileName
参　　数：

选项	说明
-A, --show-all	等价于 -vET
-b, --number-nonblank	给非空输出行编号
-e	等价于 -vE
-E, --show-ends	在每行结束显示 $
-n, --number	给所有输出行编号
-s, --squeeze-blank	将所有的连续的多个空行替换为一个空行
-t	等价于 -vT
-T, --show-tabs	把 TAB 字符显示为 ^I
-v, --show-nonprinting	除了 LFD 和 TAB 之外所有控制符用 ^ 和 M - 记方式显示
--help	显示帮助并退出
--version	显示版本信息并退出

expr

功能说明： 计算表达式
语　　法： expr EXPRESSION
实　　例： 更多用法 `man expr`

```
$ expr length "this is string"
14
#计算字符串长度
```

```
$ expr 2 + 3
5
#整数运算 运算符号和参数之间要有空格分开;

$ expr `expr 2 + 3` \* 6
30
#通配符号(*),在作为乘法运算符时要用\、或两边用"、'符号修饰

$ LOOP=0
$ LOOP=`expr $LOOP + 1`
#增量计算
```

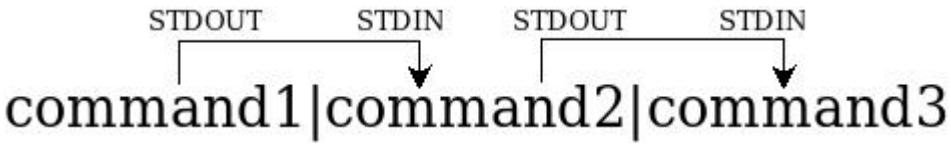
let

功能说明： 与expr命令相比，let命令更简洁直观
语　　法： let arg1 [arg2 ...]
附加说明：运算符与操作数据之间不必用空格分开，但表达式与表达式之间必须要用空格分开当运算符中有<、>、&、|等符号时，同样需要用引号（单引号、双引号）或者斜杠来修饰运算符。
实　　例：

```
$ let s=(2+3)*4
$ echo $s
20
```

管道

功能说明： 处理经由前面一个指令传出的正确输出信息(标准输出流)
语　　法： |
补充说明： **它只能处理标准输出流，没能力处理标准错误流。然后，传递给下一个命令，作为标准的输入。**
示　　例：




```
$ cat test.sh | grep -n 'echo'
5:    echo "very good!";
7:    echo "good!";
9:    echo "pass!";
11:    echo "no pass!";
#读出test.sh文件内容，通过管道转发给grep 作为输入内容


$ cat test.sh test1.sh | grep -n 'echo'
cat: test1.sh: 没有那个文件或目录
5:    echo "very good!";
7:    echo "good!";
9:    echo "pass!";
11:    echo "no pass!";
#cat test1.sh不存在，错误输出打印到屏幕，正确输出通过管道发送给grep


$ cat test.sh test1.sh 2>/dev/null | grep -n 'ec
ho'
5:    echo "very good!";
7:    echo "good!";
9:    echo "pass!";
11:    echo "no pass!";
#将test1.sh 没有找到错误输出重定向输出给/dev/null 文件，正确输出通过管道发送给grep


$ cat test.sh | ls
catfile      httprequest.txt  secure  test
      testfdread.sh  testpipe.sh    testsh.sh
   testwhile2.sh
```

```
envcron.txt  python           sh      testcase.s
h     testfor2.sh    testselect.sh  test.txt
   text.txt
env.txt      release          sms     testcronen
v.sh  testfor.sh     test.sh        testwhile1.s
h
#读取test.sh内容，通过管道发送给ls命令，由于ls 不支持标
准输入，因此数据被丢弃
```

重定向

- 输出重定向

格式： command-line1 [1-n] > file或文件操作符或设备
说明： 将一条命令执行结果（标准输出，或者错误输出，本来都要打印到屏幕上面的）重定向其它输出设备（文件，打开文件操作符，或打印机等等）1,2分别是标准输出，错误输出。
注意：

1. shell遇到”>”操作符，会判断右边文件是否存在，如果存在就先删除，并且创建新文件。不存在直接创建。 无论左边命令执行是否成功。右边文件都会变为空。
2. “>>”操作符，判断右边文件，如果不存在，先创建。以添加方式打开文件，会分配一个文件描述符[不特别指定，默认为1,2]然后，与左边的标准输出（1）或错误输出（2） 绑定。
3. 当命令：执行完，绑定文件的描述符也自动失效。0,1,2又会空闲。
4. 一条命令启动，命令的输入，正确输出，错误输出，默认分别绑定0,1,2文件描述符。
5. 一条命令在执行前，先会检查输出是否正确，如果输出设备错误，将不会进行命令执行

实例：

```
$ ls test.sh test1.sh
ls: test1.sh: 没有这个文件和目录
test.sh
#显示当前目录文件 test.sh test1.sh test1.sh实际不存
```

在

```
$ ls test.sh test1.sh 1>suc.txt
ls: test1.sh: 没有这个文件和目录
$ cat suc.txt
test.sh
#正确输出与错误输出都显示在屏幕了，现在需要把正确输出写入suc.txt
# 1>可以省略，不写，默认所至标准输出

$ ls test.sh test1.sh 1>suc.txt 2>err.txt
$ cat suc.txt err.txt
test.sh
ls: test1.sh: 没有这个文件和目录
#把错误输出，不输出到屏幕，输出到err.txt


$ ls test.sh test1.sh 1>>suc.txt 2>>err.txt
#继续追加把输出写入suc.txt err.txt  ">>"追加操作符


$ ls test.sh test1.sh 2>&-
test.sh
#将错误输出信息关闭掉


$ ls test.sh test1.sh 2>/dev/null
test.sh
#&[n] 代表是已经存在的文件描述符，&1 代表输出 &2代表错误输出 &-代表关闭与它绑定的描述符
#/dev/null 这个设备，是linux 中黑洞设备，什么信息只要输出给这个设备，都会给吃掉

#关闭所有输出
$ ls test.sh test1.sh  1>&- 2>&-
```

```
#关闭 1 ，2 文件描述符
$ ls test.sh test1.sh  2>/dev/null 1>/dev/null
#将1,2 输出转发给/dev/null设备
$ ls test.sh test1.sh >/dev/null 2>&1
#将错误输出2 绑定给 正确输出 1，然后将 正确输出 发送给 /dev/null设备  这种常用
$ ls test.sh test1.sh &>/dev/null
#& 代表标准输出 ，错误输出 将所有标准输出与错误输出 输入到/dev/null文件
```

- 输入重定向

格式： command-line [n] <file或文件描述符&设备
说明： 命令默认从键盘获得的输入，改成从文件，或者其它打开文件以及设备输入。执行这个命令，将标准输入0，与文件或设备绑定。将由它进行输入。
实例：

```
$ cat > catfile
testing
cat file test
#这里按下 [ctrl]+d 离开
#从标准输入【键盘】获得数据，然后输出给catfile文件


$ cat>catfile <test.sh
#cat 从test.sh 获得输入数据，然后输出给文件catfile


$ cat>catfile <<eof
test a file
test!
eof
#<< 这个连续两个小符号， 他代表的是『结束的输入字符』的意思。这样当空行输入eof字符，输入自动结束，不用ctrl+D
```

- exec绑定重定向

格式： exec 文件描述符[n] <或> file或文件描述符或设备
说明： 在上面讲的输入，输出重定向将输入，输出绑定文件或设备后。只对当前那条指令是有效的。如果需要在绑定之后，接下来的所有命令都支持的话。就需要用exec命令。
实例：

```
$ exec 6>&1
#将标准输出与fd 6绑定

$ ls  /proc/self/fd/
0  1  2  3  6
#出现文件描述符6

$ exec 1>suc.txt
#将接下来所有命令标准输出，绑定到suc.txt文件（输出到该文件）

$ ls -al
#执行命令，发现什么都不返回了，因为标准输出已经输出到suc.txt文件了

$ exec 1>&6
#恢复标准输出

$ exec 6>&-
#关闭fd 6描述符

$ ls /proc/self/fd/
0  1  2  3
```

使用前先将标准输入保存到文件描述符6，这里说明下，文件描述符默认会打开0,1,2还可以使用自定义描述符 。然后对标准输出绑定到文件，接下来所有输出都会发生到文件。使用完后，恢复标准的输出，关闭打开文件描述符6。

- 复杂一点实例

```
#!/usr/bin/env bash
exec 3<>test.sh;
#打开test.sh可读写操作，与文件描述符3绑定

while read line<&3
 do
    echo $line;
done
#循环读取文件描述符3（读取的是test.sh内容）
exec 3>&-
exec 3<&-
#关闭文件的，输入，输出绑定
```

等多的命令

命令可以是使用 `man command` 来查询详细文档

文件命令

命令	说明
ls	列出目录
cd dir	更改目录到dir
pwd	显示当前目录
mkdir dir	创建目录dir
rm file	删除file
rm -r dir	删除dir
cp file1 file2	复制file1到file2

命令	说明
cp -r dir1 dir2	复制dir1到dir2
mv file1 file2	将 file1 重命名或移动到 file2; 如果 file2 是一个存在的目录则将 file1 移动到目录 file2 中
ln -s file link	创建 file 的符号连接 link
touch file	创建 file
cat > file	将标准输入添加到 file
more file	查看 file 的内容
head file	查看 file 的前 10 行
tail file	查看 file 的后 10 行
tail -f file	从后 10 行开始查看 file 的内容

进程管理

命令	说明
ps	显示当前的活动进程
top	显示所有正在运行的进程
kill pid	杀掉进程 id pid
killall proc	杀掉所有名为 proc 的进程 *(小心使用)
bg	列出已停止或后台的作业
fg	将最近的作业带到前台
fg n	将作业 n 带到前台

文件权限

命令	说明
chmod octal file	更改file的权限 * 4 - 读(r) * 2 - 写(w) * 1 - 执行(x)
chown user file	更改file的所属用户
chgrp group file	更改file的所属组

搜索

命令	说明
grep pattern files	搜索 files 中匹配 pattern 的内容
grep -r pattern dir	递归搜索 dir 中匹配 pattern 的内容
command 管道 grep pattern	搜索 command 输出中匹配pattern的内容
find dir -name string	搜索dir目录下的string文件

系统信息

命令	说明
date	显示当前日期和时间
cal	显示当月的日历
uptime	显示系统从开机到现在所运行的时间
w	显示登录的用户
whoami	查看你的当前用户名
finger user	显示 user 的相关信息
uname -a	显示内核信息
cat /proc/cpuinfo	查看 cpu 信息

命令	说明
cat /proc/meminfo	查看内存信息
man command	显示 command 的说明手册
df	显示磁盘占用情况
du	显示目录空间占用情况
free	显示内存及交换区占用情况

压缩

命令	说明
tar cf file.tar files	创建包含 files 的 tar 文件 file.tar
tar xf file.tar	从 file.tar 提取文件
tar czf file.tar.gz files	使用 Gzip 压缩创建 tar 文件
tar xzf file.tar.gz	使用 Gzip 提取 tar 文件
tar cjf file.tar.bz2	使用 Bzip2 压缩创建 tar 文件
tar xjf file.tar.bz2	使用 Bzip2 提取 tar 文件
gzip file	压缩 file 并重命名为 file.gz
gzip -d file.gz	将 file.gz 解压缩为 file

网络

命令	说明
ping host	ping host 并输出结果
whois domain	获取 domain 的 whois 信息
dig domain	获取 domain 的 DNS 信息
dig -x host	逆向查询 host
wget file	下载 file

命令	说明
wget -c file	断点续传
ifconfig	设置或查看本地网卡的信息

快捷键

命令	说明
Ctrl+C	停止当前命令
Ctrl+Z	停止当前命令，并使用 fg 恢复
Ctrl+D	注销当前会话，与 exit 相似
Ctrl+W	删除当前行中的字
Ctrl+U	删除整行
!!	重复上次的命令
exit	注销当前会话

4. 通配符、元字符、转义符

通配符

通配符是由shell处理的（不是由所涉及到命令语句处理的，其实我们在shell各个命令中也没有发现有这些通配符介绍），它只会出现在 命令的“参数”里（它不用在命令名称里， 也不用在操作符上）。当shell在“参数”中遇到了通配符时，shell会将其当作路径或文件名去在磁盘上搜寻可能的匹配：若符合要求的匹配存在，则进行代换(路径扩展)；否则就将该通配符作为一个普通字符传递给“命令” ，然后再由命令进行处理。总之，通配符实际上就是一种shell实现的路径扩展功能。在通配符被处理后，shell会先完成该命令的重组，然后再继续处理重组后的命令，直至执行该命令。

字符	含义	实例
*	匹配 0 或多个字符	a*b a与b之间可以有任意长度的任意字符, 也可以一个也没有, 如aabcb, axyzb, a012b, ab。
?	匹配任意一个字符	a?b a与b之间必须也只能有一个字符, 可以是任意字符, 如aab, abb, acb, a0b。
[list]	匹配 list 中的任意单一字符	a[xyz]b a与b之间必须也只能有一个字符, 但只能是 x 或 y 或 z, 如: axb, ayb, azb。
[!list]	匹配 除list 中的任意单一字符	a[!0-9]b a与b之间必须也只能有一个字符, 但不能是阿拉伯数字, 如axb, aab, a-b。
[c1-c2]	匹配 c1-c2 中的任意单一字符 如：[0-9] [a-z]	a[0-9]b 0与9之间必须也只能有一个字符 如a0b, a1b... a9b。
{string1,string2,...}	匹配 sring1 或 string2 (或更多) 其一字符串	a{abc,xyz,123}b a与b之间只能是abc或xyz或123这三个字符串之一。

元字符（特殊字符 Meta）

shell 除了有通配符之外，由shell负责预先先解析后，将处理结果传给命令行之外，shell还有一系列自己的其他特殊字符。

字符	说明
IFS	由 或 或 三者之一组成(我们常用 space)。
CR	由 产生。
=	设定变量。
$	作变量或运算替换(请不要与 shell prompt 搞混了)。
>	重导向 stdout。 *
<	重导向 stdin。 *
&	重导向 file descriptor ，或将命令置于背境执行。 *
( )	将其内的命令置于 nested subshell 执行，或用于运算或命令替换。 *
{ }	将其内的命令置于 non-named function 中执行，或用在变量替换的界定范围。
;	在前一个命令结束时，而忽略其返回值，继续执行下一个命令。 *
&&	在前一个命令结束时，若返回值为 true，继续执行下一个命令。 *
!	执行 history 列表中的命令。*

加入”*” 都是作用在命令名直接。可以看到shell 元字符，基本是作用在命令上面，用作多命令分割（或者参数分割）。因此看到与通配符有相同的字符，但是实际上作用范围不同。所以不会出现混淆。

转义符

有时候，我们想让 通配符，或者元字符 变成普通字符，不需要使用它。那么这里我们就需要用到转义符了。 shell提供转义符有三种。

字符	说明
' '(单引号)	又叫硬转义，其内部所有的shell 元字符、通配符都会被关掉。注意，硬转义中不允许出现’ (单引号)。
" "(双引号)	又叫软转义，其内部只允许出现特定的shell 元字符：$用于参数代换`用于命令代替
\(反斜杠)	又叫转义，去除其后紧跟的元字符或通配符的特殊意义。

示例：

```
$ ls \*.txt
ls: 无法访问 *.txt: 没有那个文件或目录

$ ls '*.txt'
ls: 无法访问 *.txt: 没有那个文件或目录

$ ls 'a.txt'
a.txt

$ ls *.txt
a.txt  b.txt
```

5. 控制结构

&&

格式： statement1 && statement2 && statement3 && ...
说明： && 用法与C语言一致， 只有statement1执行成功才会执行一下条命令
实例：

```
#!/usr/bin/env bash

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
```

||

格式： statement1 || statement2 || statement3 || ...
说明： || 用法与C语言一致
实例：

```
#!/usr/bin/env bash

rm -f file_one

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

```
else
    echo "in else"
fi
exit 0
```

if/elif

格式：

```
if condition
then
    statements
else
    statements
fi
```

或

```
if condition; then
    statements
else
    statements
fi
```

或

```
if condition
then
    statements
elif condition; then
    statements
else
    statements
fi
```

实例：

```
#!/usr/bin/env bash

echo "Is it morning? Please answer yes or no"
read timeofday

if [ "$timeofday" = "yes" ]; then
    echo "Good morning"
else
    echo "Good afternoon"
fi
exit 0
```

while

说明： 循环反复直到条件为假
格式：

```
while condition; do
    statements
done
```

实例：

```
#!/usr/bin/env bash

echo "Enter password"
read trythis

while [ "$trythis" != "secret" ]; do
    echo "sorry, try again"
    read trythis
done
exit 0
```

until

说明： 循环反复直到条件为真

格式：

```
until condition
do
    statements
done
```

实例：

```
#!/usr/bin/env bash

until who | grep "$1" > /dev/null
do
    sleep 60
done

echo -e '\a' #响铃发出警报
echo "***$1 has just logged in****"
exit 0
```

shift

说明： 将变量的值依次向左传递，并形成一组新的参数值、

格式： shift[n]

实例：

```
#!/usr/bin/env bash

while [ -n "$*" ]
do
    echo $1 $2 $3 $4 $5 $6
    shift
```

```
done
exit 0
```

输出：

```
 $ ./b.sh 1 2 3 4 5 6 7
 1 2 3 4 5 6
 2 3 4 5 6 7
 3 4 5 6 7
 4 5 6 7
 5 6 7
 6 7
 7
```

for

格式：

```
for variable in values; do
    statements
done
```

或

```
for (( statements; condition ;statements )); do
    statements
done
```

实例1：

```
#! /usr/bin/env bash
for foo in bar fud 43
do
    echo $foo
done
```

```
exit 0
```

输出:

```
$ ./b.sh
bar
fud
43
```

实例2：

```
#! /usr/bin/env bash
for foo in "bar fud 43"
do
    echo $foo
done

exit 0
```

输出:

```
$ ./b.sh
bar fud 43
```

实例3：

```
#! /usr/bin/env bash
for (( i=0; i<3; i++ ))
do
    echo $foo
done

exit 0
```

输出:

```
$ b.sh
0
1
2
```

break && continue

说明： 用于立即终止当前循环的执行，break命令可以使用户从循环体中退出来。
格式： break[n]
参数： n表示要跳出几层循环，默认值为1

说明： 跳过循环体中在其之后的语句，会返回到本循环层的开头，进行下一次循环。
格式： continue[n]
参数： n表示从包含continue语句的最内层循环体向外跳到第几层循环，默认值为1，循环层数是由内向外编号。

case

格式：

```
case variable in
    pattern [ | pattern ] ...) statements;;
    pattern [ | pattern ] ...) statements;;
    ...
    *) #相当于default
    ;;
esac
```

实例：

```
#!/usr/bin/env bash

echo "Is it morning? Please answer yes or no"
```

```
read timeofday

case "$timeofday" in
    yes | y | Yes | YES )
            echo "Good Morning"
            echo "Up bright and early this morning"
            ;;
    [nN]*)
            echo "Good Afternoon"
            ;;
    *  )
            echo "sorry"
            exit 1
            ;;       #如果最后一个case模式是默认模式，可以省略最后一个双分号;;
                     #[yY] | [Yy] [Ee] [Ss])
esac

exit 0
```

select

说明： 制作一个选择表，在列表中选择一个选项执行命令行。如果选择的变量不在列表序列中，则返回一个空值。需要用break退出循环。*使用select可以更加简单的创建菜单！*

格式：

```
select var in list
do
    statements（通常用到循环变量）
done
```

实例：

```
#!/usr/bin/env bash
echo "a is 5 ,b is 3. Please select your method: "
a=5
b=3
select var in "a+b" "a-b" "a*b" "a/b"
do
    if [ $var ]; then
        echo "You select the choice '$var'"
        break   # 注：由于select是个循环，通过break来跳出
循环
    else
        echo "Invaild selection"
    fi
done
case $var in
"a+b")
    echo 'a+b= '`expr $a + $b`;;
"a-b")
    echo 'a-b= '`expr $a - $b`;;
"a*b")
    echo 'a*b= '`expr $a \* $b`;;
"a/b")
    echo 'a/b= '`expr $a / $b`;;
*)
    echo "input error"
esac

exit 0
```

6. 函数

函数的定义

说明： 函数有两部分组成，函数名和函数体。函数体是函数内的命令集合。函数名必须唯一。**方括号内为可选项！**
如果不加return，将以最后一条命令的结果作为返回值。**return后跟数值n (0~255)**

格式：

```
[ function ] funname [()] {
    action;
    [return int;]
}
```

实例：

```
#!/usr/bin/env bash

fSum 3 2;
function fSum()
{
    echo $1,$2;
    return $(($1+$2));
}
fSum 5 7;
total=$(fSum 3 2);
echo $total,$?;
exit 0
```

输出：

```
$ ./b.sh
b.sh: line 3: fSum: command not found
```

```
5,7
3,2,5
```

结论：

1. 必须在调用函数地方之前，声明函数，shell脚本是逐行运行。不会像其它语言一样先预编译。一次必须在使用函数前先声明函数
2. total=$(fSum 3 2); 通过这种调用方法，我们清楚知道，在shell 中单括号里面，可以是：命令语句。 因此，我们可以将shell中函数，看作是定义一个新的命令，它是命令，因此 各个输入参数直接用 空格分隔。 一次，命令里面获得参数方法可以通过：$0...$n得到。 $0代表函数本身。
3. 函数返回值，只能通过$? 系统变量获得，直接通过=,获得是空值。其实，我们按照上面一条理解，知道函数是一个命令，在shell获得命令返回值，都需要通过$?获得。

函数输出

为了返回大于255的数、浮点数和字符串值，最好用函数输出到变量

```
#!/usr/bin/env bash
function fun2 {
  read -p "enter a: " a
  echo -n "print 2a: "
  echo $[ $a * 2 ]
}
result=`fun2`    # 函数输出到变量
echo "return value $result"
exit 0
```

向函数传递参数(使用位置参数)

```
#!/usr/bin/env bash
if [ $# -ne 3 ]
then
    echo "usage: ${0##*/} a b c"
```

```
    exit
fi
fun3() {
    echo $[ $1 * $2 * $3 ]
}
result=`fun3 $1 $2 $3`
echo the result is $result
exit 0
```

全局变量与局部变量

默认条件下，在函数和shell主体中建立的变量都是全局变量，可以相互引用。当shell主体部分与函数部分拥有名字相同的变量时，可能会相互影响，在函数内部最好使用局部变量，消除影响，例如:

```
#!/usr/bin/env bash

echo $(uname);
declare num=1000;

uname()
{
    echo "test!";
    ((num++));
    return 100;
}
testvar()
{
    local num=10;
    ((num++));
    echo $num;

}

uname;
```

```
echo $?
echo $num;
testvar;
echo $num;
exit 0
```

输出：

```
b.sh
Linux
test!
100
1001
11
1001
```

结论：

1. 定义函数可以与系统命令相同，说明shell搜索命令时候，首先会在当前的shell文件定义好的地方查找，找到直接执行。
2. 需要获得函数值：通过$?获得
3. 如果需要传出其它类型函数值，可以在函数调用之前，定义变量（这个就是全局变量）。在函数内部就可以直接修改，然后在执行函数就可以读出修改过的值。
4. 如果需要定义自己变量，可以在函数中定义：`local 变量=值`，这时变量就是内部变量，它的修改，不会影响函数外部相同变量的值。

向函数传递数组变量

```
#!/usr/bin/env bash
a=(11 12 13 14 15)
echo ${a[*]}
function array(){
  echo parameters : "$@"
  local factorial=1
  for value in "$@"
```

```
  do
    factorial=$[ $factorial * $value ]
  done
  echo $factorial
}
array ${a[*]}
exit 0
```

函数返回数组变量

```
#!/usr/bin/env bash
a=(11 12 13 14 15)
function array(){
  echo parameters : "$@"
  local newarray=(`echo "$@"`)
  local element="$#"
  local i
  for (( i = 0; i < $element; i++ ))
  {
    newarray[$i]=$[ ${newarray[$i]} * 2 ]
  }
  echo  new value:${newarray[*]}
}
result=`array ${a[*]}`
echo ${result[*]}
exit 0
```