Shell 笔记

郑华

2019 年 1 月 15 日

目录

第一章　Bash 基本功能

1.1　基本

history　查看历史命令

- `-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`~/.bash_history`

历史命令默认保存条数可以在文件`/etc/profile` 中进行设置

alias　定义别名 `alias 别名='原命令'`

删除别名`unalias`

命令执行时顺序

1. 首先执行用绝对路径或相对路径执行的命令
2. 其次执行别名 (alias) 关联的命令
3. 其次执行 Bash 的内部命令
4. 最后执行按照`$PATH` 环境变量定义的目录查找顺序找到的第一个命令

Bash 快捷键　如表1.1所示：

表 1.1: bash 快捷键

快捷键	作用含义
ctr+A	把光标移动到命令行开头
ctr+E	把光标移动到命令行结尾
ctr+C	强制终止当前命令
ctr+L	清屏，相当于 clear 命令
ctr+U	剪切 (dd) 光标之前的命令
ctr+K	剪切光标之后的命令
ctr+Y	粘贴 ctr+U 或 ctr+K 剪切的内容
ctr+R	在历史命令中搜索
ctr+D	退出当前终端
ctr+Z	暂停，并放入后台
ctr+S	暂停屏幕输出
ctr+Q	恢复屏幕输出

环境变量配置文件 为使得环境变量永久生效的文件，主要是定义对系统的操作环境生效的默认环境变量，比如PATH、HISTSIZE、PS1、HOMENAME 等默认变量。

保存在/etc/目录下的配置文件是对所有用户都生效的。而保存在用户当前目录~/则只对当前用户生效

环境变量叠加方式 PATH='$PATH':/root 将/root 添加到环境变量中，但是这样更改后，下次重启会消失。如果想要长久生效需要更改配置文件。

执行顺序

1. /etc/profile
2. ~/.bash_profile | ~/.bash_login | ~/.profile
3. ~/.bashrc
4. /etc/bashrc
5. ~/.bash_logout

执行过程如下所图1.1示->

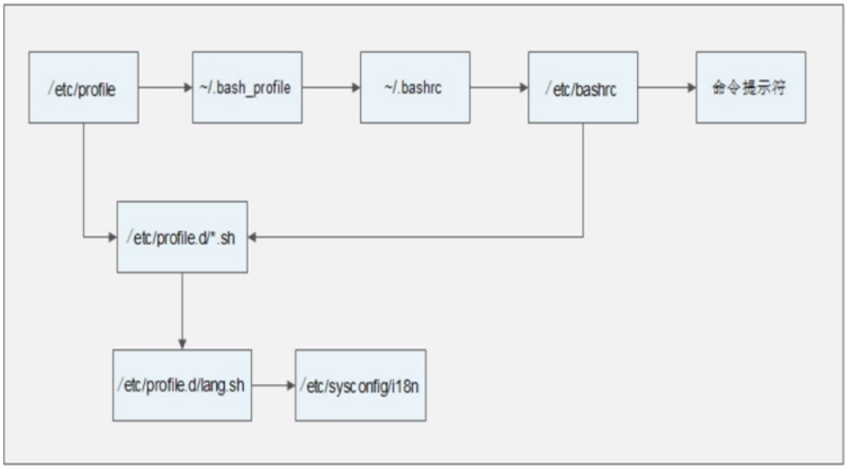


图 1.1: 执行顺序

- /etc/profile

  当用户第一次登录时, 该文件被执行. 此文件为系统的每个用户设置环境信息, 定义PATH、LOGNAME、HISTSIZE、HOSTNAME 等，调用/etc/profile.d/目录下的所有脚本文件.sh 并执行。然后进入用户目录,调用~/.bash_profile,追加PATH 并生效,并间接调用~/.bashrc，完成别名定义，并简介调用/etc/bashrc 配置文件完成不用登陆的 shell 配置并生效。

  ```
  此文件为系统的每个用户设置环境信息,
  /*--当用户第一次登录时,该文件被执行.*/
  并从/etc/profile.d目录的配置文件中搜集shell的设置.
  # /etc/profile

  # System wide environment and startup programs, for login setup
  # Functions and aliases go in /etc/bashrc
  ```

- /etc/bashrc

  为每一个运行 bash shell 的用户执行此文件. 当 bash shell 被打开时, 该文件被读取. 设置未登录的 BASH 环境变量

  ```
  /*为每一个运行bash shell的用户执行此文件*/.当bash shell被打开时,该文件被读取,每次用
      户打开一个终端时，即执行此文件
  # /etc/bashrc

  # System wide functions and aliases
  # Environment stuff goes in /etc/profile
  ```

- ~/.bash_profile

  每个用户都可使用该文件输入专用于自己使用的 shell 信息, **当用户登录时, 该文件仅仅执行一次!** 默认情况下, 他设置一些环境变量, 执行用户的.bashrc 文件.

- ~/.profile

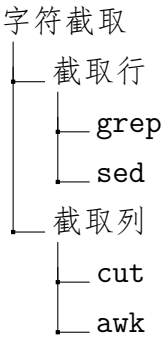
在 Debian 中使用.profile 文件代替.bash_profile 文件

- ~/.bashrc
  设置登录用户的**别名设置**等私人定义，并调用配置/etc/bashrc
- ~/.bash_logout
  当每次退出系统 (退出 bash shell) 时, 执行该文件.

参考 https://yq.aliyun.com/articles/47501#comment

source source 配置文件 => . 配置文件 使配置文件生效。

1.2 字符截取



grep grep 选项 字符串 文件名

cut cut 选项 文件名，有局限，仅支持制表符、和常用分隔符（,:。-），不支持空格。

- -f列号 提取第几列
- -d分隔符 按照指定分隔符分割列

cut -f 2 student.txt 提取文件第 2 列

cut -f 2,3 student.txt 提取文件第 2 列和第三列

cut -d ":" -f 1,3 /etc/passwd 截取第一列和第三列，并且分隔符号为:

1.3 awk

1.3.1 基本使用

使用格式 `awk '条件1{动作1} 条件2{动作2}...' 文件名`，先读入一行，然后逐行执行操作。

条件 Pattern 一般使用关系表达式作为条件

- `x > 10`
- `x >= 10`
- `x <= 10`

动作 Action

- 格式化输出
- 流程控制语句

例子

- `awk ' {printf $2 "\t" $6 "\n"}'  student.txt` :patten 省略，表示对每行都执行，显式文件中的第 2 列和第 6 列。
- `df -h|awk '{print $1 "\t" $3}'` : 截取 df 输出的第 1 列和第三列，与 cut 不同的是，awk 无须指定分割符，并且需要注意的是，print 会自动分行，但是格式控制与 printf 相同。

1.3.2 awk 编程

使用格式

BEGIN 在其他所有命令逐行执行前，执行一次 BEGIN 后的动作

```
awk 'BEGIN{print "Test awk BEGIN"} {print $2 "\t" $5}'  student.txt
```

这条命令首先执行 BEGIN 后的 print Anction, 然后逐行读取数据执行之后的条件与动作。

FS 内置变量 指定分隔符

```
awk '{FS=":"}{print $1"\t"$3}'  /etc/passwd
```

这条命令意思是指定分割符为:，但是在读入第一行后，再执行后面 Action 时，指定分隔符已经来不及了，所以这种情况第一行时不会正确使用分割符分割的，而是从第 2 行开始的，为了解决这个问题，可以利用前面提到的 BEGIN 解决。

```
awk 'BEGIN{FS=":"}{xx}'  xx
```

END 在所有命令执行完后，执行一次 END 后的动作

```
awk '{print $1"\t"$3}  END{print"The End"}'  student.txt
```

关系运算 `cat fileName.txt | grep -v Name | awk '$6 >= 87 {print $2}'`

1.4 sed

轻量**流** (与 vi 相比支持管道操作) 编辑器，主要用来将数据进行选取、替换、删除、新增的命令。

使用格式 `sed [选项] '[动作]'  文件名`

- `-n` 一般 sed 命令会将所有数据都输出到屏幕，而如果使用-n 选项，则只会将经过 sed 命令处理的行输出到屏幕
- `-e` 允许对输入数据应用多条 sed 命令编辑
- `-i` 用 sed 的修改结果直接修改读取数据的文件，而不是由屏幕输出。

动作

- `a` 追加，在当前行后添加一行或多行。添加多行时，除最后一行外，每行末尾需要用“\”代表数据未完结。
- `c` 行替换，用 c 后面的字符串替换原数据行，替换多行时，除最后一行外，每行末尾需用“\” 代表数据未完结。
- `i` 插入，在当前行前插入一行或多行，插入多行...

- d 删除，删除指定行
- p 打印，输出指定行
- s 字符替换，用一个字符串替换另一个字符串。格式为"行范围s/旧字符串/新字符串/g"

第二章　Shell 基本知识

2.1　Shell

Shell 就是一个命令行解释器，它的作用是解释执行用户的命令, 用户输入一条命令,Shell 就解释执行一条, 这种方式称为交互式 (Interactive)

Shell 还有一种执行命令的方式称为**批处理 (Batch)**, *用户事先写一个 Shell 脚本 (Script)*, **其中有很多条命令, 让 Shell 一次把这些命令执行完, 而不必一条一条地敲命令**。Shell 脚本和编程语言很相似, 也有变量和流程控制语句, 包括循环和分支。但 **Shell 脚本是解释执行的, 不需要编译,Shell 程序从脚本中一行一行读取并执行这些命令, 相当于一个用户把脚本中的命令一行一行敲到 Shell 提示符下执行**。作为程序设计语言，它虽然不是 Linux 系统内核的一部分，但它调用了系统内核的大部分功能来执行程序、创建文档并以并行的方式协调各个程序的运行

2.2　Shell 执行脚本

shell 执行脚本是一门解释性语言、批量化处理语言，大大的节省了工作成本，shell **脚本第一行必须以#! 开头**，*它表示该脚本使用后面的解释器解释执行*。

Example:　//script.sh 注：这是一个文本文件

```
#!/bin/bash
#注意 echo 默认为换行输出，如果不换行+\c
echo "this␣is␣a␣test"
ls
ls -l
echo "there␣are␣all␣files"
```

执行方式　：

```
//第一种执行方式:
```

```
[admin@localhost Shell]$ chmod +x script.sh
[admin@localhost Shell]$ ./script.sh

//第二种执行方式：
[admin@localhost Shell]$ /bin/bash script.sh
//等价于：
[admin@localhost Shell]$ shell script.sh
```

执行过程 对于非内置命令, 父 shell 先 fork 一个子 shell，然后子进程去执行 bash 的代码，从文本中一行一行的读取命令，然后再派生一个孙子进程去执行这些命令。

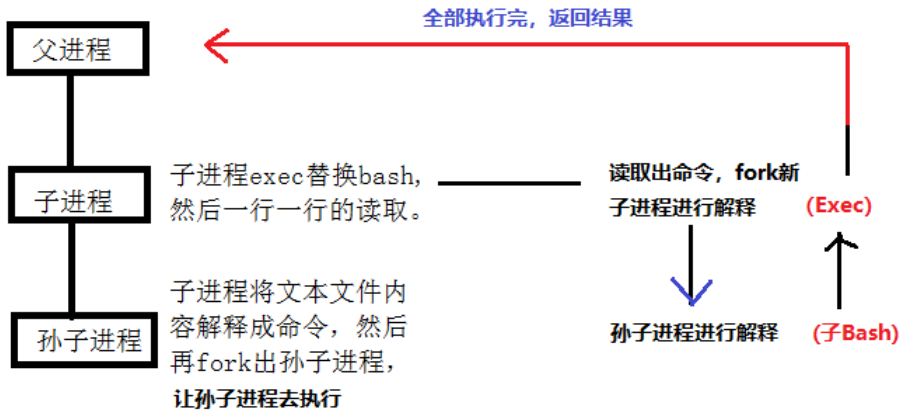


图 2.1: Shell Process

1. 交互式进程（父进程）创建一个子进程用于执行脚本，父进程等待子进程终止
2. 子进程程序替换 bash 解释器
3. 读取 shell 脚本的命令，将其以参数传递的方式传递给 bash 解释器
4. 子 bash 对 shell 脚本传入的参数进行读取，读一行识别到它是一个命令，则再创建一个子进程，子 bash 等待该新进程终止
5. 新进程执行该命令，执行完后将结果交给子进程
6. 子进程继续读取命令，创建新进程，新进程执行该命令，将结果返回给子进程，直到执行完最后一条命令
7. 子进程终止，将结果返回给交互式父进程

注意 注意：像 export、cd、env、set 这些内置命令，在键入命令行后，交互式进程不会创建子进程，而是调用 bash 内部的函数执行这些命令，改变的是交互式进程。

如果在命令行下，将多个命令用括号'()' 括起来，并用分号隔开来执行，交互式进程依然会创建一个子 shell 执行括号中的命令, 但是用'{}' 则不会创建子进程:

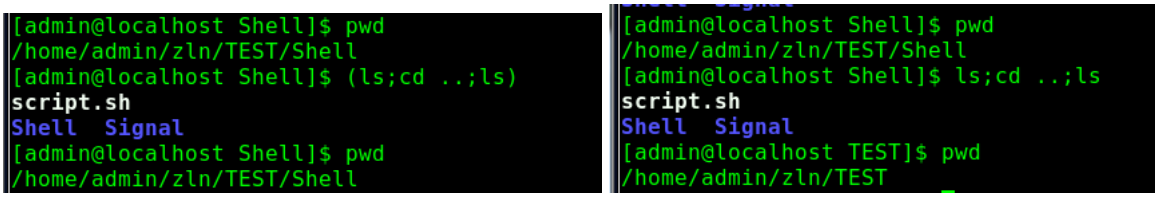


图 2.2: (comands) VS Comands

source . 或者 **source** 这两个命令是 Shell 的内建命令, 这种方式不会创建子 **Shell**, 而是直接在交互式 **Shell** 下逐行执行脚本中的命令

Example :

```
#!/bin/bash
ls
echo "#################"
cd ..
ls
```

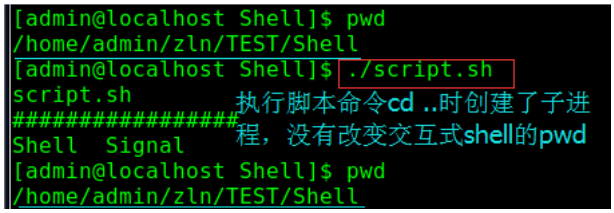


图 2.3: without use . Or source Command

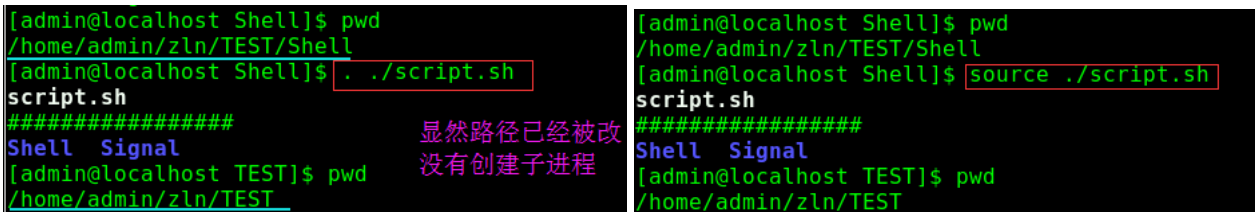


图 2.4: Use . Or source Command

shell 子进程：https://blog.csdn.net/sosodream/article/details/5683515

export 从这种意义上来说，用户可以有许多 shell，**每个 shell 都是由某个 shell（称为父 shell）派生的**。

在子 shell 中定义的变量只在该子 shell 内有效。如果在一个 shell 脚本程序中定义了一个变量，当该脚本程序运行时，这个定义的变量只是该脚本程序内的一个局部变量，其他的 shell 不能引用它，**要使某个变量的值可以在其他 shell 中被改变**，*可以使用* **export 命令**对已定义的变量进行输出。

export 命令将使系统在创建每一个新的 shell 时，**定义这个变量的一个拷贝。这个过程称之为变量输出**。

- 一个 shell 中的**系统环境变量**会被复制到子 shell 中（用 **export 定义的变量**）
- 一个 shell 中的**系统环境变量**只对 该 *shell 或者它的子 shell* 有效，该 shell 结束时变量消失（并不能返回到父 shell 中）。
- **不用 export 定义的变量只对该 shell 有效，对子 shell 也是无效的**

2.3 Shell 变量

shell 变量不需要进行任何声明，直接定义即可，因为 shell 变量的值实际上都是字符串（对于没有定义的变量默认是一个空串）。定义的时候 shell 变量由大写字母加下划线组成，并且定义的时候等号两边不能存在空格，否则会被认为是命令！

shell 变量的种类

环境变量： shell 进程的环境变量可以从当前 shell 进程传给 fork 出来的子进程

本地变量： 只存在于当前 shell 进程

利用 printenv 可以显示当前 shell 进程的环境变量；利用 set 命令可以显示当前 shell 进程中的定义的所有变量（包括环境变量和本地变量）和函数。

一个 shell 变量定义后仅存在于当前 Shell 进程, 是一个本地变量。用export 命令可以把本地变量导出为环境变量。用 unset 命令可以删除已定义的环境变量或本地变量。

```
//分步 先定义后导出
COUNT=5
export COUNT
```

```
//一步完成定义和导出环境变量
export COUNT=5

//删除已经定义的环境变量
unset COUNT
```

变量引用： 引用 shell 变量要用到 $ 符号，加{} 可以防止歧义。

```
COUNT=5
echo $COUNT
echo ${COUNT}911
```

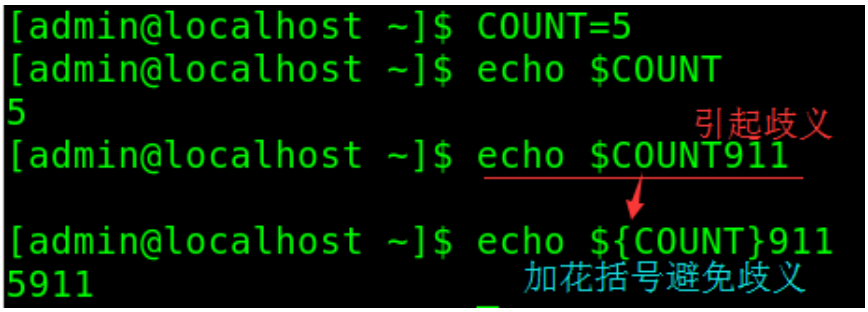


图 2.5: Shell Variable Use

只读变量 使用 readonly 命令可以将变量定义为只读变量，只读变量的值不能被改变。

下面的例子尝试更改只读变量，结果报错：

```
#!/bin/bash
myUrl="http://www.w3cschool.cc"
readonly myUrl
myUrl="http://www.runoob.com"

//运行脚本，结果如下：
/bin/sh: NAME: This variable is read only.
```

删除变量 使用 unset 命令可以删除变量。语法：unset variable_name, 变量被删除后不能再次使用。unset 命令不能删除只读变量。

```
#!/bin/sh
myUrl="http://www.runoob.com"
unset myUrl
echo $myUrl

// 以上没输出
```

Shell 字符串 字符串是 shell 编程中最常用最有用的数据类型（除了数字和字符串，也没啥其它类型好用了），字符串可以用单引号，也可以用双引号，也可以不用引号。

- **单引号字符串的限制:**

  1. 单引号里的任何字符都会原样输出，单引号字符串中的变量是无效的；
  2. 单引号字串中不能出现单引号（对单引号使用转义符后也不行）。

- **双引号字符串的好处:**

  1. **双引号里可以有变量**
  2. **双引号里可以出现转义字符**

```
str='this␣is␣a␣string'
your_name='qinjx'
str="Hello,␣I␣know␣your␣are␣\"$your_name\"!␣\n"

// 拼接字符串
your_name="qinjx"
greeting="hello,␣"$your_name"␣!"
greeting_1="hello,␣${your_name}␣!"
echo $greeting $greeting_1

// 获取字符串长度
string="abcd"
echo ${#string} #输出 4

// 提取子字符串：以下实例从字符串第 2 个字符开始截取 4 个字符
string="runoob␣is␣a␣great␣site"
echo ${string:1:4} # 输出 unoo

// 查找子字符串:查找字符 "i 或 s" 的位置:
string="runoob␣is␣a␣great␣company"
echo 'expr␣index␣"${string}"␣is' //# 输出 8
```

Shell 数组 bash **支持一维数组**（不支持多维数组），并且没有限定数组的大小。类似与 C 语言，数组元素的下标由 0 开始编号。获取数组中的元素要利用下标，下标可以是整数或算术表达式，其值应大于或等于 0

- **定义数组**：用括号来表示数组，数组元素用” 空格” 符号分割开。定义数组的一般形式为：数组名 =(值 1 值 2 ... 值 n)，还可以单独定义数组的各个分量

- **读取数组：**${数组名[下标]}
- **求数组长度:**获取数组长度的方法与获取字符串长度的方法相同,length=${#array_name[@]}

```
// 定义数组
array_name=(value0 value1 value2 value3)
array_name=(
    value0
    value1
    value2
    value3
)
// 单独定义数组的各个分量
array_name[0]=value0
array_name[1]=value1
array_name[n]=valuen

// 读取数组各元素和所有元素
valuei=${array_name[i]}
// 使用@符号可以获取数组中的所有元素
echo ${array_name[@]}

// 获取数组的长度
# 取得数组元素的个数
length=${#array_name[@]}
# 或者
length=${#array_name[*]}
# 取得数组单个元素的长度
lengthn=${#array_name[n]}
```

Shell 注释　以# 开头的行为注释行

2.4　Shell 通配符、命令代换、单引号、双引号

见基础相关拓展.

2.5　Shell 传递参数

我们可以在执行 Shell 脚本时，向脚本传递参数，脚本内获取参数的格式为：$n。n 代表一个数字，1 为执行脚本的第一个参数，2 为执行脚本的第二个参数，以此类推……

- $#：参数个数
- $$：当前 shell 的 PID 进程 ID
- $@和$* 均表示所有参数，形式有所不同。$@：`"$1" "$2" … "$n"`；$*：`"$1 $2 … $n"`。

```
//以下实例我们向脚本传递三个参数，并分别输出，其中 $0 为执行的文件名
#!/bin/bash
echo "Shell␣传递参数实例！";
echo "执行的文件名：$0";
echo "第一个参数为：$1";
echo "第二个参数为：$2";
echo "第三个参数为：$3";

$ chmod +x test.sh
$ ./test.sh 1 2 3
Shell 传递参数实例!
执行的文件名：./test.sh
第一个参数为：1
第二个参数为：2
第三个参数为：3
```

2.6 Shell 各种括号

http://blog.csdn.net/taiyang1987912/article/details/39551385

( )-小括号

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)

(( ))-双小括号

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. 只要括号中的运算符、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. **常用于算术运算比较，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。只要括号中的表达式符合C语言运算规则,比如可以直接使用for((i=0;i<5;i++)), 如果不使用双括号, 则为for i in `seq 0 4`或者for i in {0..4}。再如可以直接使用if ((3i<5)), 如果不使用双括号, 则为if [$i -lt 5]

[ ]-中括号

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**

[[ ]]-双中括号

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。比如可以直接使用if [[$a != 1 && $a != 2]], 如果不适用双括号, 则为if [$a -ne 1] && [$a != 2] 或者if [$a -ne 1 -a $a != 2]

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

```
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
```

{ }-大括号

{} 常规用法

1. 大括号拓展 (通配 (globbing)) 将对大括号中的文件名做扩展

   - 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*
   - 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。

{} 特殊的替换用法 ${var:-string},${var:+string},${var:=string},${var:?string}

1. ${var:-string} 和${var:=string}:

   若变量 var 为空,则用在命令行中用 string 来替换${var:-string},否则变量 var 不为空时,则

用变量 var 的值来替换${var:-string};对于${var:=string}的替换规则和${var:-string}是一样的，所不同之处是${var:=string}若 var 为空时，用 string 替换${var:=string}的同时，把 string 赋给变量 var： ${var:=string}很常用的一种用法是，判断某个变量是否赋值，没有的话则给它赋上一个默认值。

2. ${var:+string}的替换规则和上面的相反:

   即只有当 var 不是空的时候才替换成 string，若 var 为空时则不替换或者说是替换成变量 var 的值，即空值。(因为变量 var 此时为空，所以这两种说法是等价的)

3. ${var:?string}替换规则为:若变量 var 不为空,则用变量 var 的值来替换${var:?string};若变量 var 为空，则把 string 输出到标准错误中，并从脚本中退出。我们可利用此特性来检查是否设置了变量的值。

{} 模式匹配替换用法 模式匹配记忆方法：

是去掉左边(在键盘上#在$之左边)

% 是去掉右边(在键盘上%在$之右边)

#和%中的单一符号是最小匹配，两个相同符号是最大匹配。

${var%pattern},${var%%pattern},${var#pattern},${var##pattern}

1. 第一种模式：${variable%pattern}，这种模式时，shell 在 variable 中查找，看它是否一给的模式 pattern 结尾，如果是，就从命令行把 variable 中的内容去掉右边最短的匹配模式
2. 第二种模式：${variable%%pattern}，这种模式时，shell 在 variable 中查找，看它是否一给的模式 pattern 结尾，如果是，就从命令行把 variable 中的内容去掉右边最长的匹配模式
3. 第三种模式：${variable#pattern}这种模式时，shell 在 variable 中查找，看它是否一给的模式 pattern 开始，如果是，就从命令行把 variable 中的内容去掉左边最短的匹配模式
4. 第四种模式：${variable##pattern}这种模式时，shell 在 variable 中查找，看它是否一给的模式 pattern 结尾，如果是，就从命令行把 variable 中的内容去掉右边最长的匹配模式
5. 注：这四种模式中都不会改变 variable 的值，其中，只有在 pattern 中使用了 * 匹配符号时，% 和%%，#和## 才有区别。

```
# var=testcase
# echo $var
testcase
# echo ${var%s*e}
testca
```

```
# echo $var
testcase
# echo ${var%%s*e}
te
# echo ${var#?e}
stcase
# echo ${var##?e}
stcase
# echo ${var##*e}

# echo ${var##*s}
e
# echo ${var##test}
case
```

{} 字符串提取和替换 ：

`${var:num},${var:num1:num2},${var/pattern/pattern},${var//pattern/pattern}`

1. 第一种模式：`${var:num}`，这种模式时，shell 在 var 中提取第 num 个字符到末尾的所有字符。若 num 为正数，从左边 0 处开始；若 num 为负数，从右边开始提取字串，但必须使用在冒号后面加空格或一个数字或整个 num 加上括号，如`${var: -2}`、`${var:1-3}` 或`${var:(-2)}`。

2. 第二种模式：`${var:num1:num2}`，num1 是位置，num2 是长度。表示从`$var` 字符串的第`$num1` 个位置开始提取长度为`$num2` 的子串。不能为负数。

3. `${var/pattern/pattern}` 表示将 var 字符串的第一个匹配的 pattern 替换为另一个 pattern

4. `${var//pattern/pattern}` 表示将 var 字符串中的所有能匹配的 pattern 替换为另一个 pattern

2.7 参考

基本使用参考博客http://www.cnblogs.com/Lynn-Zhang/p/5758287.html。

使用教程：http://c.biancheng.net/cpp/view/6998.html

菜鸟教程-使用教程 2：http://www.runoob.com/linux/linux-shell.html

Bash 在线运行网址:http://www.runoob.com/try/runcode.php?filename=helloworld&type=bash

第三章　Shell 运算操作

3.1　算数运算

```
#!/bin/bash

a=10
b=20

val=`expr $a + $b`
echo "a␣+␣b␣:␣$val"

val=`expr $a - $b`
echo "a␣-␣b␣:␣$val"

//乘号(*)前边必须加反斜杠(\)才能实现乘法运算
val=`expr $a \* $b`
echo "a␣*␣b␣:␣$val"

val=`expr $b / $a`
echo "b␣/␣a␣:␣$val"

val=`expr $b % $a`
echo "b␣%␣a␣:␣$val"

if [ $a == $b ]
then
   echo "a␣等于␣b"
fi
if [ $a != $b ]
then
   echo "a␣不等于␣b"
fi

//////////////// Result -->//////////////////////
```

```
a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
a 不等于 b
```

3.2 关系运算

```
#!/bin/bash

a=10
b=20

if [ $a -eq $b ]
then
   echo "$a -eq $b: a 等于 b"
else
   echo "$a -eq $b: a 不等于 b"
fi
if [ $a -ne $b ]
then
   echo "$a -ne $b: a 不等于 b"
else
   echo "$a -ne $b: a 等于 b"
fi
if [ $a -gt $b ]
then
   echo "$a -gt $b: a 大于 b"
else
   echo "$a -gt $b: a 不大于 b"
fi
if [ $a -lt $b ]
then
   echo "$a -lt $b: a 小于 b"
else
   echo "$a -lt $b: a 不小于 b"
fi
if [ $a -ge $b ]
then
   echo "$a -ge $b: a 大于或等于 b"
else
   echo "$a -ge $b: a 小于 b"
```

```
fi
if [ $a -le $b ]
then
   echo "$a -le $b: a 小于或等于 b"
else
   echo "$a -le $b: a 大于 b"
fi

//////////////////////Result ->////////////////////////////////
10 -eq 20: a 不等于 b
10 -ne 20: a 不等于 b
10 -gt 20: a 不大于 b
10 -lt 20: a 小于 b
10 -ge 20: a 小于 b
10 -le 20: a 小于或等于 b
```

3.3 布尔运算

```
#!/bin/bash

a=10
b=20

if [ $a != $b ]
then
   echo "$a != $b: a 不等于 b"
else
   echo "$a != $b: a 等于 b"
fi
if [ $a -lt 100 -a $b -gt 15 ]
then
   echo "$a -lt 100 -a $b -gt 15 : 返回 true"
else
   echo "$a -lt 100 -a $b -gt 15 : 返回 false"
fi
if [ $a -lt 100 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : 返回 true"
else
   echo "$a -lt 100 -o $b -gt 100 : 返回 false"
fi
if [ $a -lt 5 -o $b -gt 100 ]
then
```

```
    echo "$a -lt 100 -o $b -gt 100 : 返回 true"
else
    echo "$a -lt 100 -o $b -gt 100 : 返回 false"
fi
//////////////////////Result ->///////////////////////////////////
10 != 20 : a 不等于 b
10 -lt 100 -a 20 -gt 15 : 返回 true
10 -lt 100 -o 20 -gt 100 : 返回 true
10 -lt 100 -o 20 -gt 100 : 返回 false
```

3.4 逻辑运算

```
#!/bin/bash

a=10
b=20

if [[ $a -lt 100 && $b -gt 100 ]]
then
    echo "返回 true"
else
    echo "返回 false"
fi

if [[ $a -lt 100 || $b -gt 100 ]]
then
    echo "返回 true"
else
    echo "返回 false"
fi

//////////////////////////Result->////////////////////////////////
返回 false
返回 true
```

3.5 字符串运算

```
#!/bin/bash
a="abc"
b="efg"
```

```
if [ $a = $b ]
then
   echo "$a = $b : a 等于 b"
else
   echo "$a = $b: a 不等于 b"
fi
if [ $a != $b ]
then
   echo "$a != $b : a 不等于 b"
else
   echo "$a != $b: a 等于 b"
fi
if [ -z $a ]
then
   echo "-z $a : 字符串长度为 0"
else
   echo "-z $a : 字符串长度不为 0"
fi
if [ -n $a ]
then
   echo "-n $a : 字符串长度不为 0"
else
   echo "-n $a : 字符串长度为 0"
fi
if [ $a ]
then
   echo "$a : 字符串不为空"
else
   echo "$a : 字符串为空"
fi
////////////////////////////Result-->//////////////////////////
abc = efg: a 不等于 b
abc != efg : a 不等于 b
-z abc : 字符串长度不为 0
-n abc : 字符串长度不为 0
abc : 字符串不为空
```

3.6 文件测试运算

```
#!/bin/bash

file="/var/www/runoob/test.sh"
if [ -r $file ]
```

```
then
    echo "文件可读"
else
    echo "文件不可读"
fi
if [ -w $file ]
then
    echo "文件可写"
else
    echo "文件不可写"
fi
if [ -x $file ]
then
    echo "文件可执行"
else
    echo "文件不可执行"
fi
if [ -f $file ]
then
    echo "文件为普通文件"
else
    echo "文件为特殊文件"
fi
if [ -d $file ]
then
    echo "文件是个目录"
else
    echo "文件不是个目录"
fi
if [ -s $file ]
then
    echo "文件不为空"
else
    echo "文件为空"
fi
if [ -e $file ]
then
    echo "文件存在"
else
    echo "文件不存在"
fi


////////////////////Result ////////////////////////////////
```

文件可读
文件可写
文件可执行
文件为普通文件
文件不是个目录
文件不为空
文件存在

第四章　Shell 脚本控制结构

4.1　两种判断格式

test 选项　`test -e /root/install.log`

[空格条件空格]　`[ -e /root/install.log ]`

4.2　if-else

```
a=10
b=20
if [ $a == $b ]
then
    echo "a␣等于␣b"
elif [ $a -gt $b ]
then
    echo "a␣大于␣b"
elif [ $a -lt $b ]
then
    echo "a␣小于␣b"
else
    echo "没有符合的条件"
fi
```

4.3　while

let 命令　let 命令是 BASH 中用于计算的工具，用于执行一个或多个表达式，变量计算中不需要加上 $ 来表示变量。如果表达式中包含了空格或其他特殊字符，则必须引起来

```
#!/bin/sh
```

```
int=1
while(( $int<=5 ))
do
    echo $int
    let "int++"
done

# let 使用方法
let a=5+4
let b=9-3
let no--
let ++no
let no+=10
let no-=20
```

4.4 for

```
for loop in 1 2 3 4 5
do
    echo "The value is: $loop"
done
//////////////////////////Result is ....
The value is: 1
The value is: 2
The value is: 3
The value is: 4
The value is: 5

for str in 'This is a string'
do
    echo $str
done
//////////////////////Result is ......
This is a string
```

4.5 until

```
until condition
do
    command
done
```

4.6 case

```
case 值 in
模式1)
    command1
    command2
    ...
    commandN
    ;;
模式2)
    command1
    command2
    ...
    commandN
    ;;
esac

echo '输入␣1␣到␣4␣之间的数字:'
echo '你输入的数字为:'
read aNum

case $aNum in
    1) echo '你选择了␣1'
    ;;
    2) echo '你选择了␣2'
    ;;
    3) echo '你选择了␣3'
    ;;
    4) echo '你选择了␣4'
    ;;
    *) echo '你没有输入␣1␣到␣4␣之间的数字'
    ;;
esac
```

4.7 continue,break

第五章　Shell 函数

5.1　函数定义

shell 中函数的定义格式如下：

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

注意

1. 可以带 function fun() 定义，也可以直接 fun() 定义, 不带任何参数。
2. 参数返回，可以显示加：return 返回，如果不加，将以最后一条命令运行结果，作为返回值。return 后跟数值 n(0-255)
3. 函数返回值在调用该函数后通过 $? 来获得

```
#!/bin/bash
// Without Return
demoFun(){
    echo "这是我的第一个␣shell␣函数!"
}
echo "-----函数开始执行-----"
demoFun
echo "-----函数执行完毕-----"

---->Result:

-----函数开始执行-----
这是我的第一个 shell 函数!
-----函数执行完毕-----
```

```
// With Return
funWithReturn(){
    echo "这个函数会对输入的两个数字进行相加运算..."
    echo "输入第一个数字:␣"
    read aNum
    echo "输入第二个数字:␣"
    read anotherNum
    echo "两个数字分别为␣$aNum␣和␣$anotherNum␣!"
    return $(($aNum+$anotherNum))
}
funWithReturn
echo "输入的两个数字之和为␣$?␣!"

---->Reuslt:

这个函数会对输入的两个数字进行相加运算...
输入第一个数字:
1
输入第二个数字:
2
两个数字分别为 1 和 2 !
输入的两个数字之和为 3 !
```

5.2 函数参数

在 Shell 中，调用函数时可以向其传递参数。在函数体内部，通过 $n 的形式来获取参数的值，例如，$1 表示第一个参数，$2 表示第二个参数...

```
#!/bin/bash

funWithParam(){
    echo "第一个参数为␣$1␣!"
    echo "第二个参数为␣$2␣!"
    echo "第十个参数为␣$10␣!"
    echo "第十个参数为␣${10}␣!"
    echo "第十一个参数为␣${11}␣!"
    echo "参数总数有␣$#␣个!"
    echo "作为一个字符串输出所有参数␣$*␣!"
}
funWithParam 1 2 3 4 5 6 7 8 9 34 73

----------->Result:
```

```
第一个参数为 1 !
第二个参数为 2 !
第十个参数为 10 !
第十个参数为 34 !
第十一个参数为 73 !
参数总数有 11 个!
作为一个字符串输出所有参数 1 2 3 4 5 6 7 8 9 34 73 !
```

注意，$10 不能获取第十个参数，获取第十个参数需要${10}。当 n>=10 时，需要使用${n}来获取参数。

另外，还有几个特殊字符用来处理参数：

参数类型	说明
$#	传递到脚本的参数个数
$*	以一个单字符串显示所有向脚本传递的参数
$$	脚本运行的当前进程 ID 号
$!	后台运行的最后一个进程的 ID 号
$@	与$* 相同，但是使用时加引号，并在引号中返回每个参数。
$-	显示 Shell 使用的当前选项，与 set 命令功能相同
$?	显示最后命令的退出状态。0 表示没有错误，其他任何值表明有错误。

表 5.1: 参数说明

第六章　Shell 脚本调用已有脚本

和其他语言一样，**Shell 也可以包含外部脚本**。这样可以很方便的封装一些公用的代码作为一个独立的文件。

Shell 文件包含的语法格式如下：

```
. filename # 注意点号(.)和文件名中间有一空格
或
source filename
```

Example　->

```
//--->test1.sh
#!/bin/bash
url="http://www.runoob.com"

//--->test2.sh 引用test1.sh
#使用 . 号来引用test1.sh 文件
. ./test1.sh

# 或者使用以下包含文件代码
# source ./test1.sh

echo "菜鸟教程官网地址：$url"

--->Exec and Result:
$ chmod +x test2.sh
$ ./test2.sh
菜鸟教程官网地址：http://www.runoob.com
```

第七章　Shell 示例

7.1　LeetCode

How would you print just the 10th line of a file?　From LeetCode

```
#!/bin/bash
count=0

while read line
do
    let ++count;
    if [ $count -eq 10 ] // if [[$count <= 10]] if((count <= 10))
        then echo $line
        break
    fi
done < file.txt
```

7.2　课程

添加 10 个用户

```
// Method One!
#!/bin/bash
for i in {0..10}
do
    if /usr/bin/id user$i > /dev/null 2>&1 #将正确结果输出到空洞，返回的0，　对于查询不存
        在的用户,将输出到空洞，返回1
        then echo "this␣user$i␣have␣exists"
    esle
        useradd user$i && echo "set␣up␣user$i␣success"
    fi
done
```

```
// Method Two!
for i in `seq 1 10`
do
    cut -d: -f1 /etc/passwd |grep "user$i" 2>>/tmp/etc.err || useradd user$i
done
```

>/dev/null 2>&1 默认情况是 1，也就是等同于1>/dev/null 2>&1, 意思就是**把标准输出**重定向到“黑洞”，**还把错误输出 2** 重定向到**等同于** 1 的重定向，也就是标准输出和错误输出都进了“黑洞”

& 表示**等同于**的意思，2>&1，表示 2 的输出重定向**等同于** 1

重定向有严格的顺序要求！

cut -c 显示该行第几列的东西

cut -c n,m 显示第 n 列到第 m 列的东西

cut -f n [-d :] 用：进行切分改行, -f 表示显示第几列

第八章　参考文献

练习题：https://wenku.baidu.com/view/e6664cafbd64783e08122b4f.html

https://zhangge.net/4023.html

参考：http://www.cnblogs.com/90zeng/p/shellNotes.html

在线解释王者：https://explainshell.com