学习shell

书籍视频：《跟老男孩学Linux运维：Shell编程实战》

# Shell编程基础

前期需要掌握的基础知识：

1. vi/vim编辑器的命令，.vimrc设置等等搞清楚
2. 命令基础，100多个命令
3. 基础，高端的网络服务都要会。nfs，rsync，inotify，lanmp，sersync，ssh key批量分发管理。

### 什么叫Shell

shell是一个**命令解释器**，它在操作系统最外层，负责直接与用户对话，把用户输入的命令解释给操作系统并处理各种各样的操作的输出结果，输出到屏幕返回用户（交互式或者非交互式）。

**说明：**/var/log/message ，Linux的系统日志，很重要，系统有异常，看这个。

**案例1.简单清除/var/log下的messages日志脚本：**

#!bin/bash

cd /var/log

cat /dev/null > messages

echo "Logs cleaned up"

**案例2.包含变量、命令、流程控制语句清除/var/log下的messages日志简单脚本：**

#!bin/bash

LOG\_dir=/var/log

ROOT\_uid=0

if [ $UID -ne $ROOT\_uid ]

echo "You must be root to runing this script"

exit 1

fi

cd $LOG\_dir || {

echo "Cannot change to necessary directory" ##||前面命令不成功再实行中括号的命令

exit 1

}

cat /dev/null > messages && echo "Log cleaned up" ##&& 前面命令成功才实行后面命令

exit 0

### 脚本语言的种类

Unix/Linux中主要有两大类shell： Bourne Shell 和 C Shell

其他常用的脚本语言的种类： PHP语言、Perl语言 和 **Python**语言

### 常用操作系统默认的Shell

Linux：Bourne Again Shell（bash）

Solaris和FreeBSD：Bourne Shell（sh）

AIX：Korn Shell（ksh）

企业面试题Q：CentOS Linux系统默认的shell是什么？

A：bash

通过以下两种方法可以查看CentOS Linux系统默认的Shell。

1. echo $SHELL
2. grep root /etc/passwd

root:x:0:0:root:/root:/bin/bash

### Shell脚本语言的优势

Shell脚本语言的优势在于处理偏操作系统底层的业务，例如：Linux系统内部的很多应用（有的是应用的一部分）都是使用shell脚本语言开发的，因为有1000多个命令为它做支撑，特别是Linux正则表达式及三剑客（grep、awk、sed）等命令。

。。。。。。

### Shell脚本的建立和执行

##### 开头解释器语言标识

一个规范的Shell脚本的第一行必须是如下格式：

#!/bin/bash

或

#!/bin/sh #<==255个字符以内。

符号 #! 字符又称为幻数，用来指定该脚本文件的解析程序。

注意：这一行必须位于每个脚本顶端的第一行，如果不是第一行则为脚本注释。

如果在脚本开头的第一行不指定解释器，那么就要用对应的解释器来执行脚本：

如果是Shell脚本，就用bash test.sh执行test.sh。

如果是Python脚本，就用python test.sh执行test.py。

如果是expect脚本，就用expect test.exp执行test.exp。

##### 脚本的注释

跟在#后面的内容标识注释。

注释可自成一行，也可跟在命令后与命令同一行。

##### 基本执行流程

Centos：

当shell脚本运行时，它会先查找系统环境变量ENV，该变量指定了环境文件（加载顺序通常是/etc/profile、~/.bash\_profile、~/.bashrc、（家目录下，为用户环境变量）/etc/bashrc），在加载了上述环境变量文件后，shell就开始执行shell脚本中的内容。

只要在/etc/profile、/etc/profile.d/这几个文件或目录下中定义的环境变量，就是全局的，在脚本就能直接取到。如果在该用户下执行脚本，当然首先是在该用户下读取家目录中的全局环境变量。

Shell脚本是**从上至下、从左至右一次执行每一行的命令及语句**，即执行完了一个命令后再执行下一个，如果在shell脚本中遇到子脚本（即**脚本嵌套）时，就会先执行子脚本的内容，完成后再返回父脚本继续执行脚本内后续的命令及语句。**

通常情况下，在执行shell脚本时，会**向系统内核请求启动一个新的进程**，以便在该进程中执行脚本的命令及子shell脚本，如图：



Shell脚本的执行通常可以采用以下三种方式：

1. bash script-name或sh script-name（推荐使用的方式）
2. path/script-name或./ script-name（当前路径下执行脚本）

需要先将脚本文件的权限修改为可执行（chmod +x script-name），再在脚本绝对路径或相对路径下执行。

1. source script-name或. script-name（”**.**”后有空格）

使用source或“.”（点号）读入或加载指定的shell脚本文件script-name，然后，依次执行script-name中的所有语句。**这些语句将在当前父shell脚本father.sh进程中运行**（**其他几种模式都会启动新的进程执行子脚本**）。因此，使用source或“.”可以将script-name自身脚本中的变量值或函数等的返回值或函数等的返回值传递到当前父shell脚本father.sh中使用。这是它和其他几种方法的最大区别。

source或“.”命令的功能是：在当前shell中执行source或“.”加载并执行的相关脚本文件中的命令及语句，而**不是**产生一个子shell来执行文件中的命令。

czha@czha-virtual-machine:~/test\_shell$ cat 3

usercurdir='pwd'

czha@czha-virtual-machine:~/test\_shell$ source 3

czha@czha-virtual-machine:~/test\_shell$ pwd

/home/czha/test\_shell

czha@czha-virtual-machine:~/test\_shell$ $usercurdir

/home/czha/test\_shell

例：已经如下命令及返回结果，请问echo $user的返回的结果为（）

czha@czha-virtual-machine:~$ echo "user='whoami'" > test.sh

czha@czha-virtual-machine:~$ cat test.sh

user='whoami'

czha@czha-virtual-machine:~$ sh test.sh

参考的选择项如下：

1. 当前用户
2. czha
3. 空（无内容输出）

*czha@czha-virtual-machine:~$*

如果执行以下命令，则有：

czha@czha-virtual-machine:~$ source test.sh

czha@czha-virtual-machine:~$ $user

czha

通过以上面试题得出如下的结论：

1. 儿子shell脚本会直接继承父亲shell脚本的变量、函数（就好像是儿子随父亲姓，基因也会继承父亲的）等，反之则不可以。
2. 如果希望反过来继承（就好像是让父亲随儿子姓，让父亲的基因也继承儿子的），就要用source或者“. ”在父亲shell脚本中事先加载儿子shell脚本。

### Shell脚本开发规范

1）开头（第一行）指定脚本解释器，通常为：

#!/bin/bash 或 #！bin/sh

2）开头加版本版权信息

#Date: 21:30 2013-07-20

#Author: Created by xxx

#Mail: xxx@xxx.com

#Function: This scripts function is ......

# Version: GNU bash, version 3.2.25(1)-release

可配置vim编辑文件时自动加上以上信息的功能，方法是可修改“~/,vimrc”配置文件。

3）脚本中不用中文注释

尽量用英文注释，防止本机或切换系统环境后中文乱码的困扰。

4）脚本以.sh为扩展名

例如：script-name.sh

5）shell脚本应存放在固定的路径下

例如：/server/scripts

**代码书写良好习惯**

1. 成对的符号应尽量一次性写出来，然后退格在符号里增加内容，以防止遗漏。这些成对的符号包括：

**{ }**、**[ ]**、**‘ ’**、**` `**、**” ”**

1. 中括号（[]）两端至少要有一个空格，因此，键入中括号时即可留出[ ],然后再退格键入中间的内容，并确保两端都至少有一个空格，即先键入一对中括号，然后再退1格，输入2个空格，再退1格，双中括号（[ [ ] ]）的写法也是如此。

例如: [ abcde ]

1. 对于流程控制语句，应一次性将格式写完，再添加内容。

比如，一次性完成if语句的格式，应为：

if 条件内容

then

内容

fi

一次性完成for循环语句的格式，应为：

for

do

内容

done

1. 使用**缩进**让代码更易读

if 条件内容

then

内容

fi

1. 对于常规变量的字符串定义变量值应加双引号，并且等号前后不能有空格，需要强引用的（指所见即所得的字符引用），则用单引号（**’’**），如果是命令的引用，则用反引号（``）。例如：

OLDBOTY\_FILE=“test.txt”

1. 脚本中的单引号、双引号及反引号必须为应英文状态下的符号，其实所有的Linux字符及符号都应该是英文状态下的符号，这点需要特别注意

**良好的习惯可以让我们避免很多不必要的麻烦，提升很多的工作效率！**

# Shell变量基础

变量可分为两类：环境变量（全局变量）和普通变量（局部变量）。

* 环境变量：

也称为全局变量，可以在创建它们的shell及其派生出来的任意子进程shell中使用，环境变量又可分为自定义环境变量和bash内置的环境变量。

* 普通变量

也称为局部变量，只能在创建它们的shell函数或shell脚本中使用。普通变量一般由开发者在开发脚本程序时创建。

**Q：shell脚本中变量前加$表示什么意思?**

A1：就是一个规定的标识吧，告诉shell，这个时候该取$后面这个变量的值了。:加个$表示这引用这个变量的值。

A2：**$变量名： 表示输出变量，可以用$c和${c}两种方式**。

**Q：什么是子进程呢？**

A1：就是说在我目前这个shell的情况下，去打开另一个新的shell，新的那个shell就是子进程。

### 环境变量

环境变量一般是指用export内置命令导出的变量，用于**定义shell的运行环境，保证shell命令的正确执行**。

Shell通过环境变量来确定 登录用户名、命令路径、终端类型、登录目录等，所有的环境变量都是系统全局变量，可用于所有子进程中，这包括编辑器、shell脚本和各类应用。

环境变量可以在命令行中设置和创建，但用户退出命令行时这些变量值就会丢失，因此，如果希望永久保存环境变量，可在用户家目录的.bash\_profile或,bashrc（非用户登录模式特有，例如远程SSH）文件中，或者全局配置/etc/bashrc（非用户登录模式特有，例如远程SSH）、/etc/profile或者/etc/profile.d/文件中定义。在将环境变量放入上述文件后，**每次用户登录时这些变量都将被初始化。**

按照系统规范，所有**环境变量的名字均采用大写形式（虽然不是必须的，规范）**。在将环境变量应用于用户进程程序之前，**都应该用export命令导出定义**，例如：正确的环境变量定义方法为export OLDGIRL=1。

例如：查看/ect/profile脚本文件

……

HOSTNAME=`/usr/bin/hostname 2>/dev/null`

HISTSIZE=1000

if [ "$HISTCONTROL" = "ignorespace" ] ; then

export HISTCONTROL=ignoreboth

else

export HISTCONTROL=ignoredups

fi

***#用export导出 可以一起导，也可以一个个***

export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL

……

有一些环境变量，比如HOME、PATH、SHELL、UID、USER等，在用户登录之前就已经被/bin/login程序设置好了。通常环境变量被定义并保存在用户家目录下的.bash\_profile文件或全局的配置文件/etc/profile中。

例如：

[czha@localhost /]$ echo $PS1

[\u@\h \W]\$

u: 表示user用户

h: hostname主机名

W：路径

如果想让这个显示调整，可以设置

##### 关于环境变量与局部变量的继承

**子进程仅会继承父进程的环境变量，不会继承父进程的自定义变量（普通变量）。**

被export后的变量，我们可以称它为“环境变量”。环境变量可以被子进程所引用，但是其他的自定义变量内容就不会存在于子进程中。

例如：

[root@localhost scripts]# cat m2.sh

echo $OLDBOY

echo $TEST

[root@localhost scripts]# export TEST="命令行中export是否会继承"

[root@localhost scripts]# ./m2.sh

I am oldboy.

命令行中export是否会继承

[root@localhost scripts]#

从上可以看出，执行m2.sh脚本后，虽开启了新的进程，但是export后的TEST变量，还是继承到了m2.sh脚本中。

如果取消export的定义，仅仅将TEST定义成局部变量，则TEST不会输出，如下：

[root@localhost scripts]# unset TEST

[root@localhost scripts]# echo $TEST

[root@localhost scripts]# TEST="命令行中export是否会继承"

[root@localhost scripts]# ./m2.sh

I am oldboy.

[root@localhost scripts]#

为什么环境变量的数据可以被子进程所引用呢？这是因为内存配置的关系。理论上是这样的：

* 当启动一个shell，操作系统会分配一记忆块给shell使用，此内存内的变量可让子进程取用；
* 若在父进程利用export功能，可以让自定义变量的内容写到上述的记忆块当中（环境变量）；
* 当加载另一shell时（即启动子进程，而离开原本的父进程了），子shell可以将父shell的环境变量所在的的记忆块导入自己的环境变量当中。

##### 自定义环境变量

export命令 和 带-x选项的declare内置命令

1. export 变量名=value
2. 变量名=value; export 变量名
3. declare -x 变量名=value

[czha@localhost /]$ su root

Password:

[root@localhost /]# vim etc/profile

[root@localhost /]# tail -l /etc/profile

fi

done

#just for test

export OLDBOY="I am oldboy."

unset i

unset -f pathmunge

[root@localhost /]# echo $OLEBOY

[root@localhost /]# . /etc/profile （.后有空格 == source）

[root@localhost /]# echo $OLDBOY

I am oldboy.

[root@localhost /]# mkdir server

[root@localhost /]# mkdir /server/scripts

[root@localhost /]# cd server/scripts/

[root@localhost scripts]# vi m2.sh

[root@localhost scripts]# bash m2.sh

I am oldboy.

[root@localhost scripts]# sh m2.sh

I am oldboy.

[root@localhost scripts]# cat m2.sh

echo $oldboy

**设置环境变量**

1. 用户环境变量配置

.bash\_profile、

bashrc （推荐在此文件中优先设置）

1. 全局环境变量配置

/etc/profile

/etc/bashrc （推荐在此文件中优先设置）

/etc/profile.d/

设置登录提示的两种方式

（1）在/etc/motd文件中增加（原文件中无任何内容）提示的字符串，如下：

[root@localhost etc]# cat motd

Welcome to czha's linux for shell training!!

（2）在/etc/profile.d/下面增加如下脚本：

[root@localhost etc]# cat /etc/profile.d/oldboy.sh

echo “Here is oldboy training”

##### 显示与取消环境变量

在查看设置的变量时，**有3个命令可以显示变量的值：set、env和declare**（替代早期的typeset）。

* set命令输出所有的变量，包括全局变量和局部变量；
* env命令只显示全局变量；
* declare命令输出所有的变量、函数、整数和已经导出的变量。

set -o命令显示bash Shell的所有参数配置信息。

printenv也可查看

**取消变量的方法为使用“unset 变量名称”**，例如取消myname的设置：

unset myname

用set可以查看环境变量，用unset可以取消本地变量和环境变量

[czha@localhost ~]$ echo $OLDBOY

I am oldboy.

[czha@localhost ~]$ **unset OLDBOY**

[czha@localhost ~]$ echo $OLDBOY

#什么都没有显示

[czha@localhost ~]$ tail /etc/profile

fi

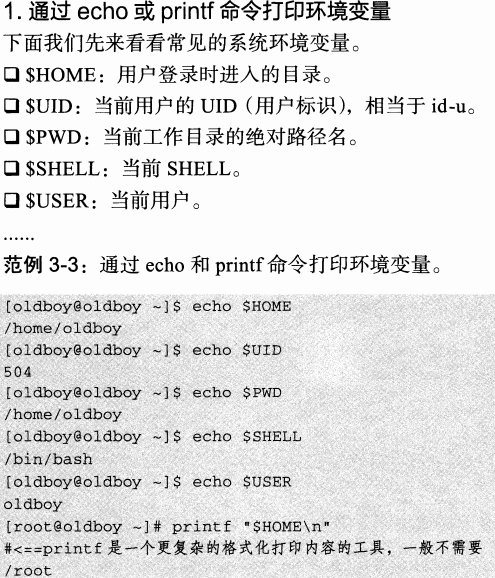
done

#just for test

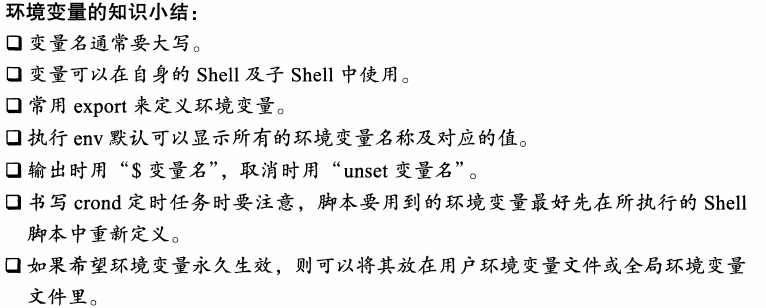
export OLDBOY="I am oldboy."

unset i

unset -f pathmunge



##### 总结



[root@localhost etc]# echo A="$USER"

A=czha

[root@localhost etc]# echo "A=$USER"

A=czha

[root@localhost etc]# echo A=$USER

A=czha

##### 环境变量初始化与对应文件的生效顺序

在登录Linux系统并启动一个bash shell时，默认情况下bash会在若干个文件中查找环境变量的设置。这些文件可统称为系统环境文件。Bash检查的环境变量文件的情况取决于系统运行shell的方式。系统运行shell的方式一般有3种：

1. 通过系统用户登录后默认运行的shell
2. 非登录交互式运行shell
3. 执行脚本运行非交互式shell

**用户登录Linux系统时启动shell**

/etc/profile ——> /etc/profile.d/\*.sh ——> ~/.bash\_profile -----> ~/.bashrc ——> /etc/bashrc

**用户的shell不是登录时启动的（非登录shell）**

~/.bashrc ——> /etc/bashrc

### 普通变量

本地变量在用户当前shell生存期的脚本中使用。例如，本地变量oldboy的取值为bingbing，这个值只在用户当前shell生存期中有意义。如果在shell中启动另一个进程或者退出，那么变量oldboy的值将会无效。

##### 普通变量的定义

为定义的普通变量赋值，一般有以下三种写法：

变量名=value #<==赋值时不加引号

变量名=‘value’ #<==赋值时加单引号

变量名=“value” #<==赋值时加双引号

##### 变量名规则以及变量内容赋值的要求

变量名一般由字母、数字、下划线组成，可由字母或下划线开头。

例如：oldboy、oldboy123、oldboy\_training

变量的内容可以用单引号或双引号引起来，也可不加引号，但是这三者的含义是不同的。

##### 普通变量的定义及输出示例

例1：在命令行中输入一下变量，并逐一打印：

a=192.168.1.2

b='192.168.1.2'

c="192.168.1.2"

echo "a=$a"

echo "b=$b"

echo "c=${c}"

在命令行中输入，我将其放入脚本文件中，一样的效果。

**[root@localhost scripts]# ls**

m2.sh varTmp\_1.sh varTmp\_2.sh

**[root@localhost scripts]# cat varTmp\_1.sh**

a=192.168.1.2

b='192.168.1.2'

c="192.168.1.2"

echo "a=$a" # echo a=$a也能达到相同效果

echo "b=$b"

echo "c=${c}"

**[root@localhost scripts]# sh varTmp\_1.sh**

a=192.168.1.2

b=192.168.1.2

c=192.168.1.2

可见，将连续的**普通字符串**的内容赋值给变量，不管用不用引号，或者不管用什么引号，它的内容是什么，打印变量时就会输出什么。

例2：

**[root@localhost scripts]# cat varTmp\_2.sh**

a=192.168.1.2

b='192.168.1.2'

c="192.168.1.2"

a=192.168.1.2-$a #如果改为a=192.168.1.2-a**（1）**

b='192.168.1.2-$a' **#单引号，不是反引号 此时的a为步骤（1）中的a**

c="192.168.1.2-$a"

echo "a=$a"

echo "b=$b"

echo "c=${c}"

**[root@localhost scripts]# ./varTmp\_2.sh**

a=192.168.1.2-192.168.1.2 #条件1修改，该句输出为a=192.168.1.2-a，没有$，则变量a不会被解析

b=192.168.1.2-$a

c=192.168.1.2-192.168.1.2-192.168.1.2

##### 变量定义的基本技巧总结

a=192.168.1.2 #无引号

b='192.168.1.2' #单引号

c="192.168.1.2" #双引号

1. **不加任何引号直接定义变量的内容**

例如：a=192.168.1.2-$a

定义变量a的方式是不加任何引号直接定义变量的内容，当内容为简单连续的数字、字符串、路径名时，可以这样用，例如：a=1, b=oldboy等。不加引号时，值里有变量的会被**解析后再输出**，上述变量定义中因为$a的值被解析为192.168.1.2，因此，新的a值就是192.168.1.2-192.168.1.2。

1. **单引号方式定义**

例如: b=’ 192.168.1.2-$a’

定义变量b的方式是通过单引号定义。这种定义方式的特点是：输出变量内容时，单引号里是什么就输出什么，即使内容中有变量和命令（命令需要反引起来）也会把它们原样输出。这种方式比较适合于定义显示纯字符串的情况，即不希望解析变量、命令等的场景。因此，对于这里变量b的值，定义时看到的是什么就输出什么，即192.168.1.2-$a。

1. **双引号方式定义**

例如: c=”192.168.1.2-$a”

定义变量c的方式是通过双引号定义。这种定义方式的特点是：输出变量内容时，引号里的变量及命令会经过**解析后再输出内容**，而不是把双引号中的变量名及命令（命令需要反引起来）原样输出。这种方式比较适合于字符串中附带有变量及命令且想将其解析后再输出的变量定义。

**老男孩经验：**

数字内容的变量定义可以不加引号，其他没有特别要求的字符串等定义最好都加上双引号，如果真的需要原样输出就加单引号，定义变量加双引号是最常见的使用场景。

##### 把一个命令的结果作为变量的内容赋值的方法

对需要获取命令结果的变量内容赋值的常见方法有两种：

（1）变量名=`ls` #<==把命令用反引号引起来，**不推荐**，因为容易与单引号混淆

（2）变量名=$(ls) #<==把命令用$()括起来，**推荐**

例：用两种方法把命令的结果赋值给变量。

[root@localhost scripts]# ls

m2.sh varTmp\_1.sh varTmp\_2.sh

[root@localhost scripts]# CMD=`ls` #<==``为反引号

[root@localhost scripts]# echo $CMD

m2.sh varTmp\_1.sh varTmp\_2.sh

[root@localhost scripts]# CMD1=$(pwd)

[root@localhost scripts]# echo $CMD1

/server/scripts #<==`打印当前用户所在的目录

**注：生产场景中把命令的结果作为变量的内容进行赋值的方法在脚本开发时很常见。**

例：按天打包网站的站点目录程序，生成不同的文件名（此为企业实战案例）

[root@localhost scripts]# CMD=$(date +%F) #<==将当前日期（格式为2017-09-07）赋值给CMD变量

[root@localhost scripts]# echo $CMD #<==输出变量的值

2017-09-07

[root@localhost scripts]# echo $(date +%F).tar.gz #<==直接输出时间命令的结果

2017-09-07.tar.gz

[root@localhost scripts]# tar zcf etc\_$(date +%F).tar.gz /etc #<==将时间作为压缩包名打包

tar: Removing leading `/' from member names

[root@localhost scripts]# ls

**etc\_2017-09-07.tar.gz**  m2.sh varTmp\_1.sh varTmp\_2.sh

RANDOM

这是随机数变量。在/dev/random这个文件中，可以通过这个随机数文件的变量（$RANDOM）来随机取得随机数值。在BASH环境下，这个RANDOM变量的内容介于0~32768之间。

[root@localhost czha]# declare -i number=$RANDOM\*10/32768; echo $number <==此时会随机取出0~9之间的值。

0

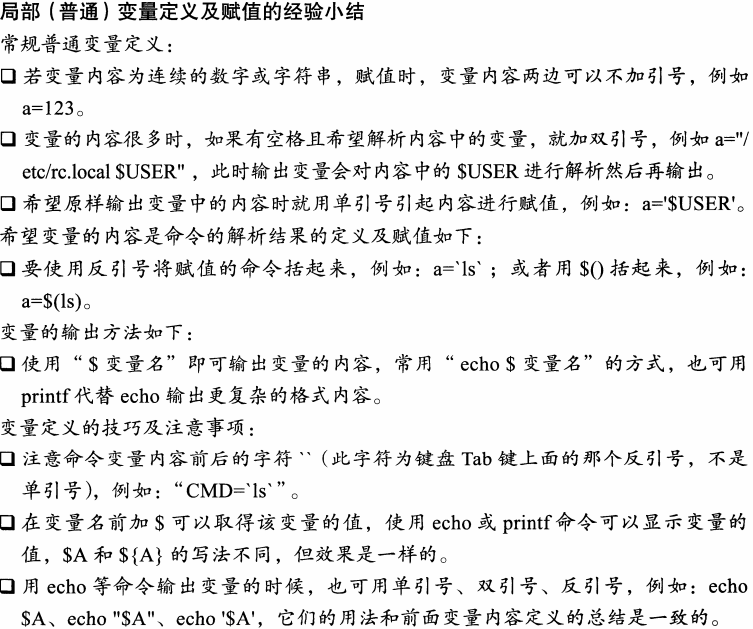
[root@localhost czha]# declare -i number=$RANDOM\*10/32768; echo $number

6

[root@localhost czha]# declare -i number=$RANDOM\*10/32768; echo $number

1

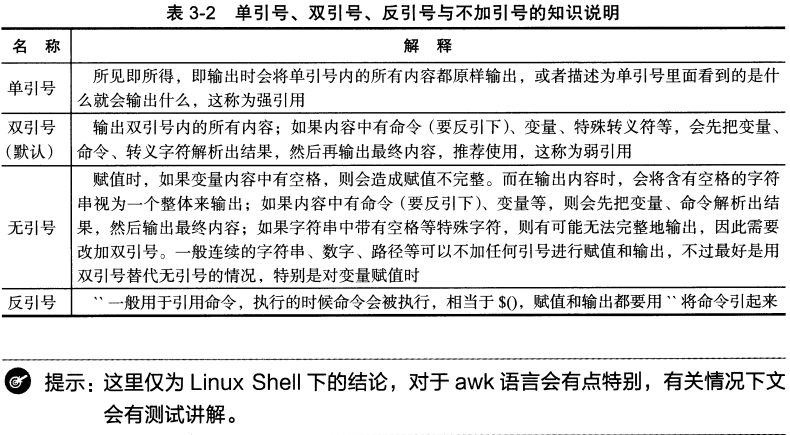
##### 总结





##### 变量定义及变量输出说明

有关shell变量定义、赋值及变量输出加单引号、双引号、反引号与不加引号的简要说明如下：



**老男孩的建议：**

* 在脚本中定义普通字符串变量时，应尽量把变量的内容用双引号括起来。
* 单纯数字的变量内容可以不加引号。
* 希望变量的内容原样输出时需要加单引号。
* 希望变量值引用命令并获取命令的结果时就用反引号或$()。

几个实例（**很重要**），见PDF

### 变量定义技巧总结

可以多学习和模仿操作系统自带的/etc/init.d/functions函数库脚本的定义思路，多学习Linux系统脚本中的定义，有经验的读者最终应形成一套适合自己的规范和习惯。

1. 变量名及变量内容定义小结
2. 变量名只能为字符、数字或下划线，只能以字母或下划线开发。
3. 变量名的定义要有一定规范，并且要见名之意。

示例：

OldboyAge=1

Oldboy\_age=1

OldboyAgeSex=1

OLDBOYAGE=1

1. 一般的变量定义、赋值常用双引号；简单连续的字符串可以不加引号；希望原样输出时使用单引号
2. 希望变量的内容时命令的解析结果时，要用反引号``，或者用$()把命令括起来再赋值。
3. shell定义变量时使用”=”的知识

“a=1”里等号是赋值的意思；比较变量是否相等时也可以用“=”或“==”。

1. 打印输出及使用变量的知识
2. 打印输出或使用变量时，变量名前要接$符号；变量名后面紧接其他字符的时候，要用大括号将变量部分单独括起来；在unset、export、(())等场景中使用但不打印变量时不加$，这个有些例外。
3. 打印输出或者使用变量时，一般加双双引号或不加引号；如果是字符串变量，最好加双引号；希望原样输出时使用单引号。

# Shell变量的功能

### echo——变量的显示

可以利用echo这个命令来显示变量，但是变量在被显示时，前面必须要加上字符“$”才行。

echo $xxx

或者

echo ${xxx}

### locale——影响显示结果的语系变量

查看我们的linux到底支持了多少的语系呢？这可以由locale这个命令查询到。

[root@localhost scripts]# locale

**LANG=zh\_CN.utf8**

LC\_CTYPE="zh\_CN.utf8"

LC\_NUMERIC="zh\_CN.utf8"

LC\_TIME="zh\_CN.utf8"

LC\_COLLATE="zh\_CN.utf8"

LC\_MONETARY="zh\_CN.utf8"

LC\_MESSAGES="zh\_CN.utf8"

LC\_PAPER="zh\_CN.utf8"

LC\_NAME="zh\_CN.utf8"

LC\_ADDRESS="zh\_CN.utf8"

LC\_TELEPHONE="zh\_CN.utf8"

LC\_MEASUREMENT="zh\_CN.utf8"

LC\_IDENTIFICATION="zh\_CN.utf8"

**LC\_ALL=**

[root@localhost scripts]#

当我们使用locale时，系统是列出目前linux主机内保有的语系文件，这些语系文件都放置在/usr/lib/locale/这个目录中。

### read——变量键盘读取

要读取来自键盘输入的变量，就是用read这个命令。这个命令最常被用在shell script的编写当中。

范例一：让用户由键盘输入内容，将该内容编程名为atest的变量

[root@localhost scripts]# read atest

keyboard input

[root@localhost scripts]# echo $atest

keyboard input

[root@localhost scripts]#

范例二：提示用户30秒内输入自己的大名，将该输入字符串作为名为named的变量内容

[root@localhost scripts]# read -p "Please keyin your name:" -t 30 named

Please keyin your name:zhajin

[root@localhost scripts]# echo $named

zhajin

[root@localhost scripts]#

### declare/typeset——声明变量的类型

declare [-aixr] varible

参数：

-a : 将后面名为varible的变量定义成为数组（array）类型

-i : 将后面名为varible的变量定义成为整数数字（integer）类型

-x : 用法与export一样，就是将后面的varible变成环境变量

-r : 将变量设置成为readonly类型，改变了不可被更改内容，也不能重设

范例一：让变量sum进行100+300+50累加结果

[czha@localhost scripts]$ sum=100+300+50

[czha@localhost scripts]$ echo $sum

100+300+50 <==怎么没有帮我计算加总？因为这是文字类型的变量属性

[czha@localhost scripts]$ declare -i sum=100+300+50 <==文字变量属性改为了整型数字类型

[czha@localhost scripts]$ echo $sum

450

[czha@localhost scripts]$

由于在默认的情况下，bash对于变量又几个基本的定义：

* 变量类型默认为“字符串”，所以若不指定变量类型，则1+2为一个“字符串”而不是“计算式”，所以上述第一个执行的结果才会出现那个情况的；
* bash环境中的数值运算，默认最多仅能到达整数类型，所以1/3结果是0.

范例二：将sum编程环境变量

[czha@localhost scripts]$ declare -x sum

[czha@localhost scripts]$ export | grep sum

declare -ix sum="450" <==果然出现了。包括i与x的声明。

[czha@localhost scripts]$

范例三：让sum变成只读属性，不可改动。

[czha@localhost scripts]$ declare -r sum

[czha@localhost scripts]$ sum=testing

bash: sum: 只读变量

[czha@localhost scripts]$

范例四：让sum变成非环境变量的自定义变量

[czha@localhost scripts]$ declare +x sum <== 将 – 变成 + 可以进行取消操作

[czha@localhost scripts]$ declare -p sum <== -p 可以单独列出变量的类型

declare -ir sum="450" <== 看吧！只剩下i，r的类型，不再有x了，不再试环境变量

[czha@localhost scripts]$

注意：比较有趣的是，如果你不小心将变量设置为“只读”，通常得要注销再登录才能复原该变量的类型了。

/\* 我尝试unset变量看看，只读的是否能取消，结果是不行 \*/

[czha@localhost scripts]$ echo $sum

450

[czha@localhost scripts]$ declare -p sum

declare -irx sum="450"

[czha@localhost scripts]$ unset sum

bash: unset: sum: 无法反设定: 只读 variable

[czha@localhost scripts]$

##### 数组array变量类型

设置数组的变量与内容：

var[index]=content

我有一个数组名为var，内容为var[1]=”small min”， var[2]=”big min”， var[3]=”nice min”。

[czha@localhost scripts]$ echo "${var[1]}, ${var[2]}, ${var[3]}"

small min, big min, nice min

[czha@localhost scripts]$ echo $var[1]$var[2 $var[3]

[1] [2 [3] <==注意，一定要加 { } ，否则不正确

数组的变量类型比较有趣的地方在于“读取”，一般来说，建议直接以${数组}的方式来读取，会比较正确无误。

### ulimit——与文件系统及程序的限制关系

想象一个状态：我的linux主机里面同时登陆了十个人，这十个人不知怎么搞得，同时打开了100个文件，每个文件的大小约为10MB，请问一下，我的linux主机的内存要有多大才够？10x100x10=10000MB=10G……天哪，这样系统不挂掉才怪。

为了要预防这个情况的发生，所以我们的bash是可以限制用户的某些系统资源的，包括可以打开的文件数量、可以使用的CPU时间可以使用的内存总量等。如何设置？用ulimit吧。

ulimit [-HSTabcdefilmnpqrstuvx [limit]] （书本上：ulimit [-SHacdfltu] [配额]）

参数：

-H : hard limit，严格的设置，必定不能超过这个设置的数值

-S ： soft limit，警告的设置，可以超过这个设置值，但是若超过则有警告信息。

在设置上，通常soft会比hard小，举例来说，soft可设置为80而hard

设置为100，那么你可以使用到90（因为没有超过100），但介于80~100

之间时，系统会有警告信息通知你。

-a ： 后面不接任何参数，可列出所有的限制额度。

-c ： 当某些进程发生错误时，系统可能会将该进程在内存中的信息写成文件（排错用），

这种文件就被称为内核文件（core file）。此为限制每个内核文件的最大容量。

-f ： 此shell可以常见的最大文件容量（一般可能设置为2GB）单位为KB。

-d ： 进程可使用的最大断裂内存（segment）容量。

-l ： 可用于锁定（lock）的内存量。

-t ： 可使用的最大CPU时间（单位为秒）。

-u ： 单一用户可以使用的最大进程数量。

范例一：列出你目前身份（假设为root）的所有限制数据数值

[czha@localhost scripts]$ ulimit -a

core file size (blocks, -c) 0 <== 只要是0就代表没有限制

data seg size (kbytes, -d) unlimited

scheduling priority (-e) 0

**file size (blocks, -f) unlimited** <== 可创建的单一文件的大小

pending signals (-i) 7171

max locked memory (kbytes, -l) 64

max memory size (kbytes, -m) unlimited

**open files (-n) 1024** <== 可同时打开的文件数量

pipe size (512 bytes, -p) 8

POSIX message queues (bytes, -q) 819200

real-time priority (-r) 0

stack size (kbytes, -s) 8192

cpu time (seconds, -t) unlimited

max user processes (-u) 4096

virtual memory (kbytes, -v) unlimited

file locks (-x) unlimited

范例二：限制用户仅能建立 10MB 以下容量的文件

[czha@localhost scripts]$ ulimit -f 10240

[czha@localhost scripts]$ ulimit -a

core file size (blocks, -c) 0

data seg size (kbytes, -d) unlimited

scheduling priority (-e) 0

**file size (blocks, -f) 10240**

……

[root@localhost scripts]# dd if=/dev/zero of=123 bs=1M count=20

文件大小超出限制(吐核) <==尝试创建20MB的文件，结果失败了

### 变量内容的删除、替代与替换

待整理

### alias、unalias——命令别名设置

给命令重新命名：alias 别名=’命令 参数’ (单引号)

例如：

[root@localhost scripts]# alias lm='ls -l' /\* 单引号 而不是反引号 \*/

[root@localhost scripts]# lm

总用量 20700

-rw-r--r--. 1 root root 10485760 9月 22 17:24 123

-rw-r--r--. 1 root root 10658961 9月 8 00:04 etc\_2017-09-07.tar.gz

-rw-r--r--. 1 root root 17 9月 12 14:56 grep.log

-rwxr-xr-x. 1 root root 24 9月 22 14:07 m2.sh

-rwxr-xr-x. 1 root root 139 9月 12 16:00 paraInput.sh

-rwxrwxr-x. 1 czha czha 13 9月 5 00:21 usrdir.sh

-rwxr-xr-x. 1 root root 86 9月 7 22:38 varTmp\_1.sh

-rwxr-xr-x. 1 root root 141 9月 7 23:01 varTmp\_2.sh

-rwxr-xr-x. 1 root root 41 9月 21 17:36 varTmp\_3.sh

-rwxr-xr-x. 1 root root 215 9月 15 00:07 yinhao.sh

-rwxr-xr-x. 1 root root 27 9月 14 21:12 ys\_para\_fast.sh

-rwxr-xr-x. 1 root root 146 9月 14 21:05 ys\_para\_new.sh

-rwxr-xr-x. 1 root root 156 9月 14 21:01 yunsuan\_para.sh

-rwxr-xr-x. 1 root root 154 9月 14 20:57 yunsuan.sh

如果要取消命令别名的话，那么久使用unalias，例如：

[root@localhost scripts]# unalias lm

[root@localhost scripts]# lm

bash: lm: 未找到命令...

# Shell变量知识进阶

### Shell中特殊且重要的变量

##### 特殊位置参数变量

* $0：获取档期那执行的shell脚本的文件名，如果执行脚本包含了路径，那么就包括路径；
* $n：获取当前执行的shell脚本的第n个参数值，n=1…9；

当n=0时，表示脚本的文件名；

当n大于9时，则用大括号括起来，例如${10}，接的参数以空格隔开；

* $#：获取当前执行的shell脚本后面接的参数的总个数

**$\*与$@的相同点：**

获取当前shell脚本所有传参的参数，不加引号时$\*与$@相同；

**$\*与$@的不同点：**

获取当前shell脚本所有传参的参数，都加双引号时$\*与$@是有区别的；

例如：

”$\*”，表示将所有的参数视为单个字符串，相当于”$1 $2 $3”

”$@”，表示将所有的参数视为不同的独立字符串，相当于”$1” “$2” “$3”，这是将多参数传递给其他程序的最佳方式，因为它会保留所有的内嵌在每个参数里的任何空白。

dirname 获取路径名，如果输入的是绝对路径，则显示绝对路径；若为当前路径，则显示当前路径。

basename 获取脚本文件的名称，与路径无关。

[root@localhost scripts]# cat paraInput.sh

echo $0 $1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ${11}

[root@localhost scripts]# sh paraInput.sh 1 2 3 4 5 6 7 8 9 10

pos\_0.sh 1 2 3 4 5 6 7 8 9 10

**范例：**根据用户在命令行的传参个数判断用户的输入，不合要求的给予提示并退出。

**范例：**利用set设置位置参数（同命令行脚本的传参）。

[root@localhost scripts]# set -- "I am" handsome boy. #通过set设置三字符串参数，”--”表示清除所有的参数变量，重新设置后面的参数变量。

[root@localhost scripts]# echo $#

3

[root@localhost scripts]# echo $1

I am

[root@localhost scripts]# echo $2

handsome

[root@localhost scripts]# echo $3

boy.

。。。。。。

（这个例子后续说明了$\*与$@的相同点和不同点）

##### Shell进程中的特殊状态变量

**$?** 获取执行上一个执行的执行状态返回值（0为成功，非0为失败），这个变量最常用。

$$ 获取当前执行的shell脚本的进程号（PID），这个变量不常用，了解即可。

$! 获取上一个在后台工作的进程的进程号（PID），这个变量不常用，了解即可。

$\_ 获取在此之前执行的命令或脚本的最后一个参数，这个变量不常用，了解即可。

Nginx Web服务软件下载、安装、编译实例

在企业场景下，**“$?”返回值的用法如下**：

1. 判断命令、脚本或函数等程序是否执行成功；
2. 若在脚本中调用执行“exit数字”，则会返回这个数字给“$?”变量。
3. 如果是在函数里，则通过“return 数字”把这个数字以函数返回值的形式传给“$?”。

##### Bash shell内置变量命令

Bash shell包含一些内置命令。这些命令在目录列表里是看不见的，它们由shell本身提供。

常用的内部命令有：echo、eval、exec、export、read、readonly，shift，wait，exit和点（.）。

shift常用语命令后带选项的shell脚本，shift后，参数1编成了文件名

### shell变量子串知识及实践

##### shell变量子串介绍

${parameter} 返回变量$parameter的内容

${#parameter} 返回变量$parameter内容的长度（按字符，包括空格），也适用于特殊变量

${parameter:offset} 在变量${parameter}中，从位置offset之后开始提取子串到结尾

${parameter:offset :length} 在变量${parameter}中，从位置offset之后开始提取长度为length的子串

${parameter#word} 从变量$parameter开头开始删除最短匹配的word子串

${parameter##word} 从变量$parameter开头开始删除最长匹配的word子串

${parameter/pattern/string } 使用string代替第一个匹配的pattern

${parameter//pattern/string } 使用string代替所有匹配的pattern

##### shell变量子串的实践和生产场景

[czha@localhost scripts]$ echo ${OLDBOY}

I am oldboy.

/\* 打印变量内容长度的几种方法 \*/

[czha@localhost scripts]$ echo ${#OLDBOY}

12

[czha@localhost scripts]$ echo ${OLDBOY}|wc -L #echo ${OLDBOY | wc -L #输出变量值，然后通过管道交给wc计算长度

12

[czha@localhost scripts]$ expr length "$OLDBOY" #利用expr的length函数计算变量长度

12

[czha@localhost scripts]$ echo "$OLDBOY"|awk '{print length($0)}' #利用awk的length函数计算变量长度，也可无”($0)”这几个字符

12

**几个例程，值得学习**

### Shell特殊扩展变量的知识与实践

##### Shell特殊扩展变量介绍

|  |  |
| --- | --- |
| ${parameter:-word} | 如果parameter的变量值为空或未赋值，则会返回word字符串并替代变量的值  **用途**：如果变量未定义，则返回备用的值，防止变量为空值或因未定义而导致异常 |
| ${parameter:=word} | 如果parameter的变量值为空或未赋值，则设置这个变量值为word，并返回其值。位置变量和特殊变量不适用  **用途**：基本同上一个${parameter:-word}，但该变量又额外给parameter变量赋值了 |
| ${parameter:?word} | 如果parameter的变量值为空或未赋值，那么word字符串将被作为标准错误输出，否则输出变量值  **用途**：用于捕捉由于变量未定义而导致的错误，并退出程序 |
| ${parameter:+word} | 如果parameter的变量值为空或未赋值，则什么都不做，否则word字符串将替代变量的值 |

表中，每个表达式内的冒号都是可选的。

##### Shell特殊扩展变量的实践和生产场景

[czha@localhost scripts]$ echo $test #变量未设置，所以输出时为空

[czha@localhost scripts]$ result=${test:-UNSET} #若test没值，则返回UNSET

[czha@localhost scripts]$ echo $result #打印result变量，返回UNSET，因为test没有赋值

UNSET

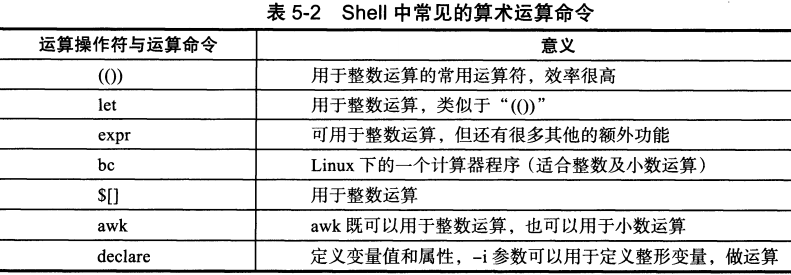
[czha@localhost scripts]$ echo ${test} #注意，此时打印test变量还是空的

[czha@localhost scripts]$

# 变量的数值计算实践

### 算术运算符





### 双小括号数值运算命令

##### 双小括号“(())”数值运算的基础语法

双小括号“(())”的作用是进行数值运算与数值比较，它的效率很高，用法灵活，企业运维经常采用。

|  |  |
| --- | --- |
| **运算操作符与运算命令** | **意义** |
| ((i=i+1)) | 此种书写方法为先运算后赋值法，即将i+1的结果赋值给变量i。注意，不能用“echo ((i=i+1))”的形式输出表达式的值，但可以用echo $((i=i+1))输出其值 |
| i=$((i+1)) | 可以在“(())”前加$符，表示将表达式运算后赋值给i |
| ((8>7&&5==5)) | 可以进行比较运算，还可以加入逻辑与逻辑或，用于条件判断 |
| echo $((2+1)) | 需要直接输出运算表达式的运算结果时，可以在 “(())”前加$符 |

##### 双小括号“(())”数值运算的实践

见实例

**值得反复学习**

# Shell脚本的条件测试与比较

### Shell脚本的条件测试

通常，在bash的各种条件结构和流程控制结构中都要进行各种测试，然后根据测试结果执行不同的操作，有时也会与if等条件语句相结合，来完成测试判断，以减少程序运行的错误。

执行条件测试表达式后通常会返回“真”或“假”，就像执行命令后的返回值为0表示真，非0表示假一样。

在bash编程里，条件测试常见的语法形式如下：

|  |  |
| --- | --- |
| 条件测试语法 | 说明 |
| 语法1：test <测试表达式> | 这是利用test命令进行条件测试表达式的方法。test命令和“<测试表达式>”之间至少有一个空格 |
| 语法2：[ <测试表达式> ] | 这是通过[]（单中括号）进行条件测试表达式的方法，和test命令的用法相同，这是**老男孩推荐**的方法。[]的边界和内容之间至少有一个空格。 |
| 语法3：[[ <测试表达式> ]] | 这是通过[[]]（双中括号）进行条件测试表达式的方法，是比test和[]更新的语法格式。[[]]的边界和内容之间至少有一个空格。 |
| 语法4：((<测试表达式>)) | 这是通过(())（双小括号）进行条件测试表达式的方法，一般用于if语句里。(())两端不需要空格。 |

# 正则表达式

简单地说，**正则表达式**就是处理字符串的方法，它是以行为单位来进行字符串的处理行为，正则表达式通过一些特殊符号的辅助，可以让用户轻易达到查找、删除、替换某特定字符串的处理程序。

### 语系对正则表达式的影响

[:alnum:] 代表英文大小写字符及数字，即0-9，A-Z，a-z

[:alpha:] 代表任何英文大小写字符，即A-Z，a-z

[:digit:] 代表数字而已，即0-9

[:lower:] 代表小写字符，即a-z

[:upper:] 代表大写字符，即A-Z

### grep

grep是分析一行信息，若当中有我们所需要的信息，就将该行列出来（关键字分析），简单的语法如下：

**grep [-acinv] [--color=auto] ‘查找字符串’ filename** (说明：单引号)

参数：

-a： 将binary文件以text文件的方式查找数据；

-c： 计算找到’查找字符串’的次数；

-i： 忽略大小写的不同，视为相同；

-n： 顺便输出行号；

-v： 反向选择，即显示出没有’查找字符串’内容的那一行；

--color=auto： 可以将找到的关键字部分加上颜色显示；

**范例一：将last当中有出现root的那一行就取出来。**

last | grep ‘root’

**范例二：与范例一相反，只要没有root的就取出来。**

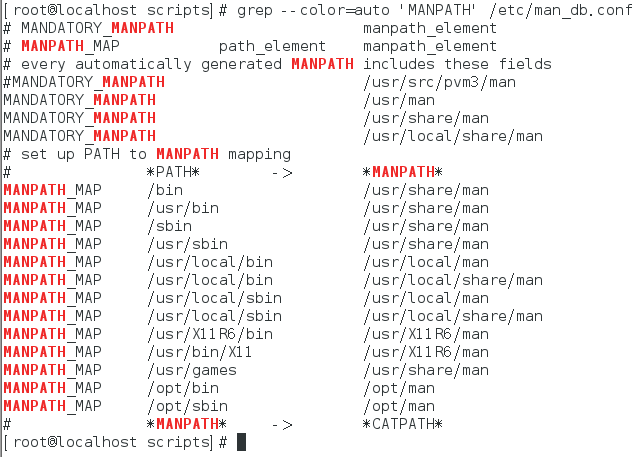
last | grep -v ‘root’

**范例三：在last的输出信息中，只要有root就取出，并且仅取第一列**

[root@localhost scripts]# last| grep 'root' |cut -d ' ' -f1

**范例四：取出/etc/man\_db.conf内含MANPATH的那几行**

grep --color=auto 'MANPATH' /etc/man\_db.conf <==同grep 'MANPATH' /etc/man\_db.conf



### 基础正则表达式的练习

##### 前提

* **语系已经使用”export LANG=C”的设置值；**
* **grep已经使用alias设置成为“grep –color=auto”；**（仅仅颜色高亮显示，更方便）

[czha@localhost ~]$ export LANG=C

[czha@localhost ~]$ locale

LANG=C

LC\_CTYPE="C"

LC\_NUMERIC="C"

LC\_TIME="C"

LC\_COLLATE="C"

LC\_MONETARY="C"

LC\_MESSAGES="C"

LC\_PAPER="C"

LC\_NAME="C"

LC\_ADDRESS="C"

LC\_TELEPHONE="C"

LC\_MEASUREMENT="C"

LC\_IDENTIFICATION="C"

LC\_ALL=

# Source global definitions

if [ -f /etc/bashrc ]; then

. /etc/bashrc

fi

alias grep='grep --color=auto'

# Uncomment the following line if you don't like systemctl's auto-paging feature:

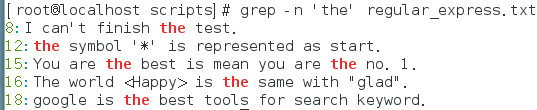
# export SYSTEMD\_PAGER=

# User specific aliases and functions

新建文件 regular\_express.txt，这个文件共22行，最下面一行为空白行。

##### 例题一：查找特定字符串

* **取the这个特定字符串**

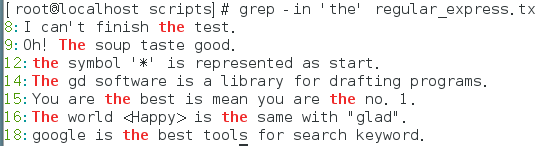


* **反向选择，没有“the”这个字符串时才显现在屏幕上**

[root@localhost scripts]# grep -nv 'the' regular\_express.txt

屏幕上除了8、12、15、16、18外都显示出来了

* **无论大小写的‘the’的这个字符串都显示**



9、14、16行都显示出来了

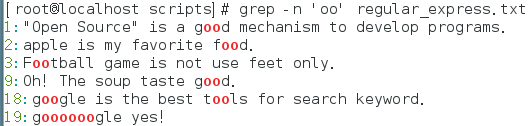
##### 例题二：利用中括号[]来查找集合字符

如果我想要查找test或taste这两个单词时，可以发现到，其实他们有共同的’t?st’存在。这时候，可以这样来查找：



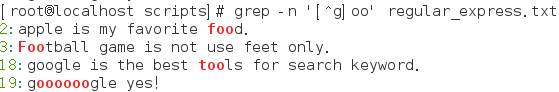
其实[ ]里面不论有几个字符，它都只代表一个字符。

而如果想要查找到有oo的字符时，则使用：



**反向选择[^]**

但是，如果我不想oo前面有g（所以[^g]放在oo前面）的话？可以利用集合字符的**反向选择[^]**来完成：



* 意思就是说，我需要的是oo，但是oo前面不能是g。仔细比较，上面两个代码，发现1、9行不见了，因为oo前面出现了g导致。2、3行没有疑问，因为foo与Foo均可以被接受，但是18行明明有google的goo啊！别忘记了，因为该行后者出现了tool的too。所以该行也被列出来。也就是说，18行里面虽然出现了我们所不需要的项目（goo），但是由于有需要的项目（too），因此，是符合字符串查找的。
* 至于19行，同样，因为goooooole里面的oo前面可能是o，例如：go(ooo)oogle，所以，这一行也符合需求。

再来，假设我**oo前面不想要有小写字符**，所以，我可以这样写[^abcd…z]oo，但是，这样似乎不怎么方便，由于小写字符的ASCII上编码的顺序是连续的（注意起那么的LANG=C）,因此，简化后如下：



当我们在一组集合字符中，如果该字符组是连续的，那么：

大写英文 [A-Z]

小写英文 [a-z]

数字 [0-9]

数字与英文 [a-zA-Z0-9]

如果我们要**取得有数字的那一行**，则：



但是，考虑到**语系对于编码顺序的影响**，因此除了连续编码使用减号“-”之外，你也可以使用如下的方法来取得前面两个测试的结果（见语系对正则表达式的影响）。

**[:lower:]即a-z，因此[a-z]等价于[[:lower:]]**



**[:digit:]即0-9 ，因此[0-9]等价于[[:digit:]]**



这样基本了解了[]、[^]以及[]中的-这几个特殊关键字。

##### 例题三：行首与行尾字符^$

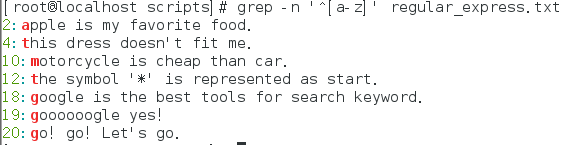
在例一中，可以查询到一行字符串里面有the的，那如果我想要**让the只在行首列出**呢？

这个时候就需要用到制表符了，如下：



此时，就只剩下第12行，因为只有第12行的行首是the开头。

此外，如果我想**开头是小写字符的那一行就列出**呢？可以这样：



等价于：[root@localhost scripts]# grep -n '^[[:lower:]]' regular\_express.txt

如果我**不想要开头是英文字母**，则可以是这样：



等价于：[root@localhost scripts]# grep -n '^[^[:alpha:]]' regular\_express.txt

**注意到了吧，那个^符号在字符集合符号（中括号[]）之内和之外是不同的；**

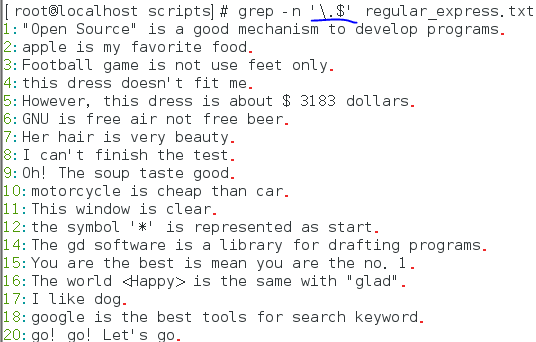
**^在[]之内表示“反向选择”；**

**^在[]之外表示定位在行首。**

**强调：[]是针对字符集合的（个人）**

###### 找行尾小数点

反过来，如果我想要找出**行尾结束为小数点(.)的那一行**，该如何处理？



特别注意到，因为小数点具有其他意义，所以必须要使用转义字符（\）来加以解除其特殊意义。

（不过，你也许会奇怪，5、9行最后也是.。怎么无法打印出来？因为windows平台和linux平台断行字符不一样，我这里是自己复制文本过来的，并不是拷贝文件过来的，与书本略有不同）

###### 找出哪一行是空白？



假设你已经知道在一个程序脚本（shell script）或者是配置文件中，空白行与开头为#的那一行是批注，因此如果你要将数据列出给别人参考时，可以将这些数据省略掉以节省宝贵的纸张，怎么做？以/etc/rsyslog.conf这个文件为例，查看输出结果：

[root@localhost scripts]# cat /etc/rsyslog.conf

[root@localhost scripts]# grep -v '^$' /etc/rsyslog.conf | grep -v '^#'

相对比：节省了很多版面

##### 例题四：任意一个字符.与重复字符\*

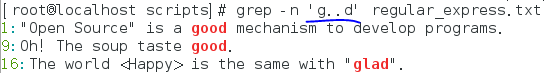
前面已经讲到，通配符\*可以用来表示任意（0或多个）字符；但正则表达式并不是通配符，两者之间是不相同的。

至于正则表达式当中的“.”则代表绝对有一个任意字符的意思，这两个字符意义如下：

**.（小数点）：代表一定有一个任意字符的意思；**

**\*（星号）：代表重复前一个0到无穷多次的意思，为组合形态。**

范例：假设我需要找出g??d的字符串，即共有四个字符，开头是g而结束时d，我可以这样做：

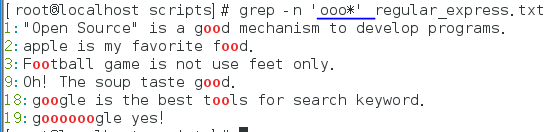


因为强调g与d之间一定要存在两个字符，因此，第13行的god和第14行的gd就不会被列出来了。

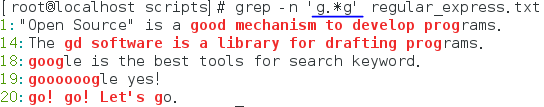
如果我想要列出有 oo, ooo, oooo 等等的数据， 也就是说，至少要有两个(含) o 以上，该如何是好？

因为 \* 代表的是『重复 0 个或多个前面的 **RE 字符**』的意义， 因此，『o\*』代表的是：『拥有空字节或一个 o 以上的字节』，因此，『 grep -n 'o\*' regular\_express.txt 』将会把所有的数据都列印出来终端上！

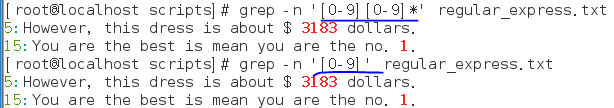
当我们需要『至少两个 o 以上的字串』时，就需要 ooo\* ，亦即是：



如果我想找出g开头与g结尾的字符串，当中的字符串可有可无，该怎么做？



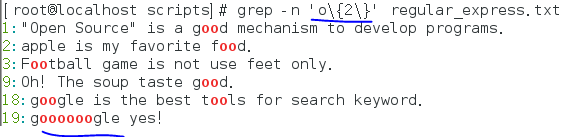
找出任意有数字的行？



两个都可以。

##### 例题五：限定连续RE字符范围{}

在上面，我们可以利用.和\*来设置0到无穷多个重复字符，那如果我们想要限制一个范围区间内的重复字符呢？比如，我们想得到2~5个o的连续字符串该如何做？此时就应该用到限定范围的字符{}了.但是因为{与}的符号在shell是由特殊意义的，因此，我们必须要使用转义字符\使它失去特殊意义才行。至于{}的语法是这样的，假设我要找两个o的字符串，可以是：



这样看似乎与 ooo\* 的字符没有什么差异啊？因为第 19 行有多个 o 依旧也出现了。好，那么换个查找的字符串，假设我们要找出 g 后面接 2 到 5 个 o ，然后再接一个 g 的字串，它会是这样：



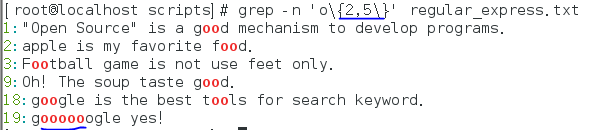
如果我想要的是 2 个 o 以上的 goooo....g 呢？除了可以是 gooo\*g ，也可以是：

[root@www ~]# grep -n 'go\{2,\}g' regular\_express.txt

18:google is the best tools for search keyword.

19:goooooogle yes!

Mine Test:



### 基础正则表达式字符（characters）

经过了上面的几个简单的范例，我们可以将基础的正规表示法特殊字符汇整如下：

|  |  |
| --- | --- |
| RE 字符 | 意义与范例 |
| ^word | 意义：待搜寻的字串(word)在行首！ 范例：搜寻行首为 # 开始的那一行，并列出行号  **grep -n '^#' regular\_express.txt** |
| word$ | 意义：待搜寻的字串(word)在行尾！ 范例：将行尾为 ! 的那一行列印出来，并列出行号  **grep -n '!$' regular\_express.txt** |
| . | 意义：代表『一定有一个任意字节』的字符！ 范例：搜寻的字串可以是 (eve) (eae) (eee) (e e)， 但不能仅有 (ee) ！亦即 e 与 e 中间『一定』仅有一个字节，而空白字节也是字节！  **grep -n 'e.e' regular\_express.txt** |
| \ | 意义：跳脱字符，将特殊符号的特殊意义去除！ 范例：搜寻含有单引号 ' 的那一行！  **grep -n \' regular\_express.txt** |
| \* | 意义：重复零个到无穷多个的前一个 RE 字符 范例：找出含有 (es) (ess) (esss) 等等的字串，注意，因为 \* 可以是 0 个，所以 es 也是符合带搜寻字串。另外，因为 \* 为重复『前一个 RE 字符』的符号， 因此，在 \* 之前必须要紧接著一个 RE 字符喔！例如任意字节则为 『.\*』 ！  **grep -n 'ess\*' regular\_express.txt** |
| [list] | 意义：字节集合的 RE 字符，里面列出想要撷取的字节！ 范例：搜寻含有 (gl) 或 (gd) 的那一行，需要特别留意的是，在 [] 当中『谨代表一个待搜寻的字节』，例如『 a[afl]y 』代表搜寻的字串可以是 aay, afy, aly 即 [afl] 代表 a 或 f 或 l 的意思！  **grep -n 'g[ld]' regular\_express.txt** |
| [n1-n2] | 意义：字节集合的 RE 字符，里面列出想要撷取的字节范围！ 范例：搜寻含有任意数字的那一行！需特别留意，在字节集合 [] 中的减号 - 是有特殊意义的，他代表两个字节之间的所有连续字节！但这个连续与否与 ASCII 编码有关，因此，你的编码需要配置正确(在 bash 当中，需要确定 LANG 与 LANGUAGE 的变量是否正确！) 例如所有大写字节则为 [A-Z]  **grep -n '[A-Z]' regular\_express.txt** |
| [^list] | 意义：字节集合的 RE 字符，里面列出不要的字串或范围！ 范例：搜寻的字串可以是 (oog) (ood) 但不能是 (oot) ，那个 ^ 在 [] 内时，代表的意义是『反向选择』的意思。例如，我不要大写字节，则为 [^A-Z]。但是，需要特别注意的是，如果以 grep -n [^A-Z] regular\_express.txt 来搜寻，却发现该文件内的所有行都被列出，为什么？因为这个 [^A-Z] 是『非大写字节』的意思，因为每一行均有非大写字节，例如第一行的 "Open Source" 就有 p,e,n,o.... 等等的小写字  **grep -n 'oo[^t]' regular\_express.txt** |
| \{n,m\} | 意义：连续 n 到 m 个的『前一个 RE 字符』 意义：若为 \{n\} 则是连续 n 个的前一个 RE 字符， 意义：若是 \{n,\} 则是连续 n 个以上的前一个 RE 字符！范例：在 g 与 g 之间有 2 个到 3 个的 o 存在的字串，亦即 (goog)(gooog)  **grep -n 'go\{2,3\}g' regular\_express.txt** |

再次强调：**『正规表示法的特殊字节』与一般在命令列输入命令的『万用字节』并不相同**，例如，在万用字节当中的 \* 代表的是『 0 ~ 无限多个字节』的意思，但是在正规表示法当中，\* 则是『重复 0 到无穷多个的前一个 RE 字符』的意思～使用的意义并不相同，不要搞混了！

举例来说，不支持正规表示法的 ls 这个工具中，若我们使用 『ls -l \* 』代表的是任意档名的文件，而 『ls -l a\* 』代表的是以 a 为开头的任何档名的文件，但在正规表示法中，我们要找到含有以 a 为开头的文件，则必须要这样：(需搭配支持正规表示法的工具)

ls | grep -n '^a.\*'

例题：

以 ls -l 配合 grep 找出 /etc/ 底下文件类型为连结档属性的档名

答：

由於 ls -l 列出连结档时标头会是『 lrwxrwxrwx 』，因此使用如下的命令即可找出结果：

ls -l /etc | grep '^l'

若仅想要列出几个文件，再以『 |wc -l 』 来累加处理即可。

### sed工具

sed本身也是一个管道命令，可以分析standard input的，而且sed还可以将数据进行替换、删除、新增、选取特定行等的功能。

[root@www ~]# sed [-nefr] [动作]

选项与参数：

-n ：使用安静(silent)模式。在一般 sed 的用法中，所有来自 STDIN

的数据一般都会被列出到萤幕上。但如果加上 -n 参数后，则只有经过

sed 特殊处理的那一行(或者动作)才会被列出来。

-e ：直接在命令列模式上进行 sed 的动作编辑；

-f ：直接将 sed 的动作写在一个文件内， -f filename 则可以运行 filename 内的

sed 动作；

-r ：sed 的动作支持的是延伸型正规表示法的语法。(默认是基础正规表示法语法)

-i ：直接修改读取的文件内容，而不是由萤幕输出。

动作说明： [n1[,n2]]function

n1, n2 ：不见得会存在，一般代表 选择进行动作的行数，举例来说，如果我的动作

是需要在 10 到 20 行之间进行的，则”10, 20[动作行为] “

function 有底下这些咚咚：

a ：新增， a 的后面可以接字串，而这些字串会在新的一行出现(目前的下一行)～

c ：取代， c 的后面可以接字串，这些字串可以取代 n1,n2 之间的行！

d ：删除，因为是删除啊，所以 d 后面通常不接任何咚咚；

i ：插入， i 的后面可以接字串，而这些字串会在新的一行出现(目前的上一行)；

p ：列印，亦即将某个选择的数据印出。通常 p 会与参数 sed -n 一起运行～

s ：取代，可以直接进行取代的工作哩！通常这个 s 的动作可以搭配

正规表示法！例如 1,20s/old/new/g 就是啦！

##### 以行为单位的新增/删除功能

###### 范例一：将/etc/passwd的内容列出并且打印行号，同时，请将2-5行删除。

[root@localhost scripts]# nl /etc/passwd | sed '2,5d'

1 root:x:0:0:root:/root:/bin/bash

6 sync:x:5:0:sync:/sbin:/bin/sync

7 shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown

省略……

看到了吧？sed 的动作为 '2,5d' ，那个 d 就是删除！因为 2-5 行给它删除了，所以显示的数据就没有 2-5 行。另外，注意一下，原本应该是要执行 sed -e 才对，没有 -e 也行。同时也要注意的是， **sed后面接的动作，请务必以 ' ' 两个单引号括住**！

###### 范例二：承上例，在第二行后（即是加在第三行）加上“drink tea?”字样！

**错误示范：**

[root@localhost scripts]# nl /etc/passwd | sed '2,3a drink tea?'

1 root:x:0:0:root:/root:/bin/bash

2 bin:x:1:1:bin:/bin:/sbin/nologin

drink tea?

3 daemon:x:2:2:daemon:/sbin:/sbin/nologin

drink tea?

4 adm:x:3:4:adm:/var/adm:/sbin/nologin

**正确示范：**

[root@localhost scripts]# nl /etc/passwd | sed '2a drink tea?'

1 root:x:0:0:root:/root:/bin/bash

2 bin:x:1:1:bin:/bin:/sbin/nologin

drink tea?

3 daemon:x:2:2:daemon:/sbin:/sbin/nologin

在 a 后面加上的字串就已将出现在第二行后面。那如果是要在第二行前呢？nl /etc/passwd | sed '2i drink tea' 就对啦！就是将 a 变成 i 即可。添加一行很简单，那如果是要增将两行以上呢？

###### 范例三：在第二行后面加入两行字，例如：“Drink tea or .....”与“drink beer?”

[root@localhost scripts]# nl /etc/passwd | sed '2a Drink tea or ......\

> drink beer ?'

1 root:x:0:0:root:/root:/bin/bash

2 bin:x:1:1:bin:/bin:/sbin/nologin

Drink tea or ......

drink beer ?

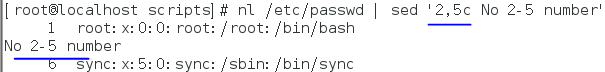
3 daemon:x:2:2:daemon:/sbin:/sbin/nologin

这个范例的重点是我们可以新增不只一行。可以新增好几行』但是每一行之间都必须要以反斜线 \ 来进行新行的添加。所以，上面的例子中，我们可以发现在第一行的最后面就有 \ 存在！那是一定要的。

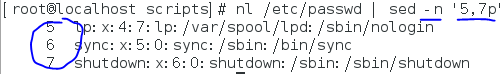
##### 以行为单位的取代与显示功能

刚刚是介绍如何新增与删除，那么如果要**整行取代**呢？看看底下的范例吧：

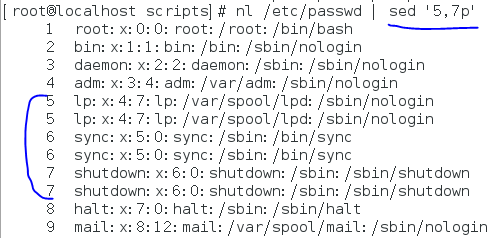
###### 范例四：我想将第2-5行的内容取代成为“No 2-5 number”？



透过这个方法我们就能够将数据整行替换了。非常容易吧！sed 还有更好用的工具。我们以前想要列出第 11~20 行，得要透过“head -n 20 | tail -n 10”之类的方法来处理，很麻烦。sed 则可以简单的直接取出你想要的那几行！是透过行号来找的。先看看底下的范例：



上述的命令中有个重要的选项『 -n 』，按照说明文件，这个 -n 代表的是『安静模式』！那么为什么要使用安静模式呢？你可以自行执行 sed '5,7p' 就知道了 (5-7 行会重复输出)！有没有加上 -n 的参数时，输出的数据可是差很多的喔！你可以透过这个 sed 的以行为单位的显示功能，就能够将某一个文件内的某些行号捉出来查阅！很棒的功能！不是吗？



##### 部分数据的搜寻并取代的功能

除了整行的处理模式之外， sed 还可以用行为单位进行部分数据的查找并替换的功能。基本上 sed 的查找并替换与 vi 相当的类似，它有点像这样：

**sed ‘s/**要被替换的字符串/新的字符串**/g’**

上表中**特殊字体的部分为关键字，请记下来**！至於三个斜线分成两栏就是新旧字串的替换。我们使用底下这个取得 IP 数据的范例，一段一段的来处理给您瞧瞧，让你了解一下什么是咱们所谓的查找并替换吧。

步骤一：先观察原始信息，利用 /sbin/ifconfig 查询 IP 为何？

[root@www ~]# /sbin/ifconfig eth0

eth0 Link encap:Ethernet HWaddr 00:90:CC:A6:34:84

inet addr:192.168.1.100 Bcast:192.168.1.255 Mask:255.255.255.0

inet6 addr: fe80::290:ccff:fea6:3484/64 Scope:Link

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

.....(以下省略).....

# 因为我们还没有讲到 IP ，这里你先有个概念即可啊！我们的重点在第二行，

# 也就是 192.168.1.100 那一行而已！先利用关键字捉出那一行！

步骤二：利用关键字配合 grep 撷取出关键的一行数据

[root@www ~]# /sbin/ifconfig eth0 | grep 'inet addr'

inet addr:192.168.1.100 Bcast:192.168.1.255 Mask:255.255.255.0

# 当场仅剩下一行！接下来，我们要将开始到 addr: 通通删除，就是像底下这样：

# ~~inet addr:~~192.168.1.100 Bcast:192.168.1.255 Mask:255.255.255.0

# 上面的删除关键在於『 ^.\*inet addr: 』啦！正规表示法出现！ ^\_^

步骤三：将 IP 前面的部分予以删除

[root@www ~]# /sbin/ifconfig eth0 | grep 'inet addr' | \

> sed 's/^.\*addr://g'

192.168.1.100 Bcast:192.168.1.255 Mask:255.255.255.0

# 仔细与上个步骤比较一下，前面的部分不见了！接下来则是删除后续的部分，亦即：

# 192.168.1.100 ~~Bcast:192.168.1.255 Mask:255.255.255.0~~

# 此时所需的正规表示法为：『 Bcast.\*$ 』就是啦！

步骤四：将 IP 后面的部分予以删除

[root@www ~]# /sbin/ifconfig eth0 | grep 'inet addr' | \

> sed 's/^.\*addr://g' | sed 's/Bcast.\*$//g'

192.168.1.100

让我们再来继续研究 sed 与正规表示法的配合练习！假设我只要 MAN 存在的那几行数据，但是含有 # 在内的注解我不想要，而且空白行我也不要！此时该如何处理呢？可以透过这几个步骤来实作看看：

……(自己上机操作)

##### 直接修改文件内容（危险操作）

你以为 sed 只有这样的能耐吗？那可不！ sed 甚至可以直接修改文件的内容呢！而不必使用管线命令或数据流重导向！不过，**由於这个动作会直接修改到原始的文件，所以请你千万不要随便拿系统配置档来测试。**我们还是使用你下载的 regular\_express.txt 文件来测试看看吧！

[root@localhost scripts]# sed -i 's/\.$/\!/g' regular\_express.txt

[root@localhost scripts]# cat regular\_express.txt

"Open Source" is a good mechanism to develop programs!

apple is my favorite food!

Football game is not use feet only!

this dress doesn't fit me!

However, this dress is about $ 3183 dollars!

GNU is free air not free beer!

Her hair is very beauty!

I can't finish the test!

Oh! The soup taste good!

motorcycle is cheap than car!

This window is clear!

the symbol '\*' is represented as start!

Oh! My god!

The gd software is a library for drafting programs!

You are the best is mean you are the no. 1!

The world <Happy> is the same with "glad"!

I like dog!

google is the best tools for search keyword!

goooooogle yes!

go! go! Let's go!

# I am VBird

[root@localhost scripts]#

sed 的『 -i 』选项可以直接修改文件内容，这功能非常有帮助！举例来说，如果你有一个 100 万行的文件，你要在第 100 行加某些文字，此时使用 vim 可能会疯掉！因为文件太大了！那怎办？就利用 sed 啊！透过 sed 直接修改/取代的功能，你甚至不需要使用 vim 去修订！很棒吧！

总之，这个 sed 不错用啦！而且很多的 shell script 都会使用到这个命令的功能～sed 可以帮助系统管理员管理好日常的工作喔！要仔细的学习呢！

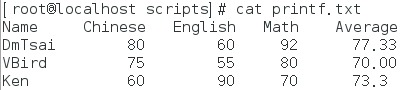
# 扩展正则表达式

后续再积累。

# 文件的格式化与相关处理

### 格式化打印：printf

在很多时候，我们可能需要将自己的数据给他格式化输出的！举例来说，考试卷分数的输出，姓名与科目及分数之间，总是可以稍微作个比较漂亮的版面配置吧？例如我想要输出底下的样式：



上表的数据主要分成五个栏位，各个栏位之间可使用 tab 或空白键进行分隔。请将上表的数据转存成为 printf.txt 档名，等一下我们会利用这个文件来进行几个小练习的。因为每个栏位的原始数据长度其实并非是如此固定的 (Chinese 长度就是比 Name 要多)，而我就是想要如此表示出这些数据，此时，就得需要列印格式管理员 printf 的帮忙了！printf 可以帮我们将数据输出的结果格式化，而且而支持一些特殊的字符～底下我们就来看看！

[root@www ~]# printf '列印格式' 实际内容

选项与参数：

关於格式方面的几个特殊样式：

\a 警告声音输出

\b 倒退键(backspace)

\f 清除萤幕 (form feed)

\n 输出新的一行

\r 亦即 Enter 按键

\t 水平的 [tab] 按键

\v 垂直的 [tab] 按键

\xNN NN 为两位数的数字，可以转换数字成为字节。

关於 C 程序语言内，常见的变量格式

%ns 那个 n 是数字， s 代表 string ，亦即多少个字节；

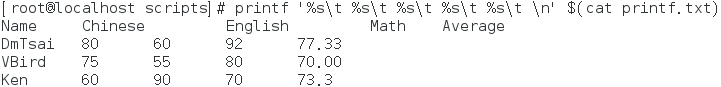
%ni 那个 n 是数字， i 代表 integer ，亦即多少整数码数；

%N.nf 那个 n 与 N 都是数字， f 代表 floating (浮点)，如果有小数码数，

假设我共要十个位数，但小数点有两位，即为 %10.2f 罗！

接下来我们来进行几个常见的练习。假设所有的数据都是一般文字 (这也是最常见的状态)，因此最常用来分隔数据的符号就是 [Tab] 啦！因为 [Tab] 按键可以将数据作个整齐的排列！那么如何利用 printf 呢？参考底下这个范例：

##### 范例一：将刚刚上头数据的文件 (printf.txt) 内容仅列出姓名与成绩：(用 [tab] 分隔)



省略

### awk：好用的数据处理工具

awk 也是一个非常棒的数据处理工具。相比于 sed 常常作用于一整个行的处理， awk 则比较倾向於一行当中分成数个“字段”来处理。因此，**awk 相当的适合对小型的数据进行处理**。awk通常运行的模式是这样的：

[root@www ~]# **awk** '条件类型1{动作1} 条件类型2{动作2} ...**'** **filename**

暂未弄明白

Shell script

# 专题

### 浅析linux下的/etc/profile、/etc/bashrc、~/.bash\_profile、~/.bashrc文件

* /etc/profile

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/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.

* ~/.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.

* ~/.bashrc

该文件包含专用于你的bash shell的bash信息,当登录时以及每次打开新的shell时,该文件被读取.

* ~/.bash\_logout

当每次退出系统(退出bash shell)时,执行该文件.

另外,/etc/profile中设定的变量(全局)的可以作用于任何用户,而~/.bashrc等中设定的变量(局部)只能**继承**/etc/profile中的变量,他们是"父子"关系.

~/.bash\_profile 是交互式、login 方式进入 bash 运行的

~/.bashrc 是交互式 non-login 方式进入 bash 运行的

通常二者设置大致相同，所以通常前者会调用后者。

每个用户都有属于自己的.bash\_history、.bash\_logout、.bash\_profile、.bashrc文件

根用户

[**root**@localhost ~]# ls -al

total 40

dr-xr-x---. 5 root root 241 Sep 7 00:21 .

dr-xr-xr-x. 18 root root 238 Sep 6 04:18 ..

-rw-------. 1 root root 1775 Feb 14 2017 anaconda-ks.cfg

-rw-------. 1 root root 1831 Sep 6 13:54 .bash\_history

-rw-r--r--. 1 root root 18 Dec 28 2013 .bash\_logout

-rw-r--r--. 1 root root 176 Dec 28 2013 .bash\_profile

-rw-r--r--. 1 root root 176 Dec 28 2013 .bashrc

czha这个用户

[**czha**@localhost ~]$ ls -al

total 5236

drwx------. 16 czha czha 4096 Sep 7 00:18 .

drwxr-xr-x. 4 root root 31 Sep 1 07:01 ..

-rw-------. 1 czha czha 2529 Sep 6 13:54 .bash\_history

-rw-r--r--. 1 czha czha 18 Aug 2 2016 .bash\_logout

-rw-r--r--. 1 czha czha 193 Aug 2 2016 .bash\_profile

-rw-r--r--. 1 czha czha 231 Aug 2 2016 .bashrc

bash的几个初始化文件 - [Ubuntu]

（1）/etc/profile

全局（公有）配置，不管是哪个用户，登录时都会读取该文件。

（2）/ect/bashrc

Ubuntu没有此文件，与之对应的是/ect/bash.bashrc，它也是全局（公有）的；bash执行时，不管是何种方式，都会读取此文件。

（3）~/.profile

若bash是以login方式执行时，读取~/.bash\_profile，若它不存在，则读取~/.bash\_login，若前两者不存在，读取~/.profile。

另外，图形模式登录时，此文件将被读取，即使存在~/.bash\_profile和~/.bash\_login。

（4）~/.bash\_login

若bash是以login方式执行时，读取~/.bash\_profile，若它不存在，则读取~/.bash\_login，若前两者不存在，读取~/.profile。

（5）~/.bash\_profile

Unbutu默认没有此文件，可新建。

只有bash是以login形式执行时，才会读取此文件。通常该配置文件还会配置成去读取~/.bashrc。

（6）~/.bashrc

当bash是以non-login形式执行时，读取此文件。若是以login形式执行，则不会读取此文件。

（7）~/.bash\_logout

注销时，且是longin形式，此文件才会读取。也就是说，在文本模式注销时，此文件会被读取，图形模式注销时，此文件不会被读取。

**下面是在本机的几个例子：**

1. 图形模式登录时，顺序读取：/etc/profile和~/.profile

2. 图形模式登录后，打开终端时，顺序读取：/etc/ bashrc（ubuntu下为/etc/ bash.bashrc）和~/.bashrc

3. 文本模式登录时，顺序读取：/etc/ bashrc （/etc/bash.bashrc），/etc/profile和~/.bash\_profile

4. 从其它用户su到该用户，则分两种情况：

（1）如果带-l参数（或-参数，--login参数），如：su -l username，则bash是lonin的，它将顺序读取以下配置文件：/etc/bash.bashrc，/etc/profile和~/.bash\_profile。

（2）如果没有带-l参数，则bash是non-login的，它将顺序读取：/etc/bash.bashrc和~/.bashrc

5. 注销时，或退出su登录的用户，如果是longin方式，那么bash会读取：~/.bash\_logout

6. 执行自定义的shell文件时，若使用“bash -l a.sh”的方式，则bash会读取行：/etc/profile和~/.bash\_profile，若使用其它方式，如：bash a.sh， ./a.sh，sh a.sh（这个不属于bash shell），则不会读取上面的任何文件。

7. 上面的例子凡是读取到~/.bash\_profile的，若该文件不存在，则读取~/.bash\_login，若前两者不存在，读取~/.profile。

**linux中profile与bashrc的区别**

要搞清bashrc与profile的区别，首先要弄明白**什么是交互式shell和非交互式shell，什么是login shell 和non-login shell**。

交互式模式就是shell等待你的输入，并且执行你提交的命令。这种模式被称作交互式是因为shell与用户进行交互。这种模式也是大多数用户非常熟悉的：登录、执行一些命令、签退。当你签退后，shell也终止了。 shell也可以运行在另外一种模式：非交互式模式。在这种模式下，shell不与你进行交互，而是读取存放在文件中的命令,并且执行它们。当它读到文件的结尾，shell也就终止了。

bashrc与profile都用于保存用户的环境信息，bashrc用于交互式non-loginshell，而profile用于交互式login shell。系统中存在许多bashrc和profile文件，下面逐一介绍：

/etc/profile此文件为系统的每个用户设置环境信息,当第一个用户登录时,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/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。有些linux版本中的/etc目录下已经没有了bashrc文件。

~/. 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,它设置一些环境变量,然后执行用户的.bashrc文件.

~/.bashrc:该文件包含专用于某个用户的bash shell的bash信息,当该用户登录时以及每次打开新的shell时,该文件被读取.

另外,/etc/profile中设定的变量(全局)的可以作用于任何用户,而~/.bashrc等中设定的变量(局部)只能继承/etc/profile中的变量,他们是"父子"关系.

某网友总结如下：

/etc/profile，/etc/bashrc 是系统全局环境变量设定

~/.profile，~/.bashrc用户家目录下的私有环境变量设定

当登入系统时候获得一个shell进程时，其读取环境设定档有三步

1首先读入的是全局环境变量设定档/etc/profile，然后根据其内容读取额外的设定的文档，如 /etc/profile.d和/etc/inputrc

2然后根据不同使用者帐号，去其家目录读取~/.bash\_profile，如果这读取不了就读取~/.bash\_login，这个也读取不了才会读取 ~/.profile，这三个文档设定基本上是一样的，读取有优先关系

3然后在根据用户帐号读取~/.bashrc ，至于~/.profile与~/.bashrc的不区别 都具有个性化定制功能

~/.profile可以设定本用户专有的路径，环境变量，等，它只能登入的时候执行一次

~/.bashrc也是某用户专有设定文档，可以设定路径，命令别名，每次shell script的执行都会使用它一次

个人认为：该专题部分观点有待验证……