原文链接：https://blog.csdn.net/cxb14897/article/details/124951398

1 shell概述

1.1 什么是Shell

Shell是一个命令行解释器，它为用户提供了一个向Linux内核发送请求以便远程远程程序的界面系统级程序，用户可以用Shell来启动、挂起、停止甚至是编写一些程序。

Shell还是一个功能相当强大的编程语言，易编写，易调试，灵活性强。Shell是解释执行的脚本语言，在Shell中可以直接调用Linux系统命令。

shell脚本编程如同其他编程语言的一样，只要有一个能编写代码的文本编辑器和一个能解释执行的脚本解释器就可以运行了，而linux下的shell种类众多，常用的用：

Bourne Shell（/usr/bin/sh或/bin/sh）

Bourne Again Shell（/bin/bash）

C Shell（/usr/bin/csh）

K Shell（/usr/bin/ksh）

Shell for Root（/sbin/sh）

… …

在诸多linux发行版系统中，最常用的就是Bash，就是Bourne Again Shell，因为其能工提供环境变量以配置用户shell环境，支持历史记录、内置算数功能、支持通配符表达式等高效性能，将linux常用命令进行的简化，被广泛应用于Debian系列的linux发行版中。

2 Shell脚本的执行方式

运行shell脚本的方法有两种：

作为可执行程序运行

作为解释器参数运行

首先我们先学习一个输出命令echo

[root@localhost ~]# echo [选项] [输出内容]

选项：

-e 支持反斜线控制的字符转换

echo命令还可显示复杂的输出格式

显示普通的字符串

[root@localhost ~]# echo "Hello Word!"

显示转义字符

[root@localhost ~]# echo "\Hello Word!\"

显示变量

[root@localhost ~]# name="abc"

[root@localhost ~]# echo "you name is $name"

显示换行

[root@localhost ~]# echo -e "Right!\n " # -e 表示开启转义

显示结果定向重定向至文件

[root@localhost ~]# echo "this is a test" > testfile

显示command命令执行结果

[root@localhost ~]# echo `date`

echo命令还有其他使用规则，可以通过man echo 进行查询

2.1 编写第一个脚本

[root@localhost ~]# vim hello.sh

#!/bin/bash

echo -e "This is a test"

[root@localhost ~]# chmod +x hello.sh

[root@localhost ~]# ./hello.sh

3 Bash基本功能

3.1 历史命令与补全命令

3.1.1 历史命令

[root@localhost ~]# history [选项][历史保存文件]

选项：

-c 清空历史命令

-w 把缓存中的历史命令写入历史命令保存文件

~/.bash\_history

历史命令默认会保存1000条，可以在环境变量配置文件/etc/profile中修改

历史命令的调用

使用上、下箭头调用以前的历史命令

使用“!n”重复执行第n条历史命令

使用“!!”重复执行上一条命令

使用“!字串”重复执行最后一条以该字串开头的命令

3.1.2 命令与文件补全

在Bash中，命令与文件补全是非常方便与常用的功能，我们只要在输入命令或文件时，按“Tab”键就会自动进行补全

3.2 命令别名与常用快捷键

3.2.1 命令别名

[root@localhost ~]# alias 别名='原命令'

#设定命令别名

[root@localhost ~]# alias

#查询命令别名

让别名永久生效

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

删除别名

[root@localhost ~]# unalias 别名

3.2.2 Bash常用快捷键

3.3 输入输出重定向

3.3.1 输出重定向

3.3.2 输入重定向

[root@localhost ~]# wc [选项][文件名]

选项：

-c 统计字节数

-w 统计单词数

-l 统计行数

3.4 多命令顺序执行与管道符

3.4.1 多命令顺序执行

举例：

[root@localhost ~]# ls;date;pwd;

[root@localhost ~]# ls /root && echo yes #当第一条命令正确执行后，才会执行第二条

[root@localhost ~]# ls /root || echo yes #当第一条命令不正确执行，才会执行第二条

[root@localhost ~]# 命令 && echo yes || echo no

3.4.2 管道符

命令格式

[root@localhost ~]# 命令1 | 命令2

#命令1的正确输出作为命令2的操作对象

如：

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

3.4.3 grep命令

[root@localhost ~]# grep [选项] “搜索内容” 文件名

选项：

-i 忽略大小写

-n 输出行号

-v 方向查找

--color=auto 搜索出的关键字用颜色显示

举例：

[root@localhost ~]# grep -i 'user' /etc/passwd

3.5 通配符和其他特殊符号

3.5.1 通配符

举例：

[root@localhost ~]# ls a\* #匹配含有in的文件

[root@localhost ~]# rm -rf \* #删除当前目录下所有内容

[root@localhost ~]# ll \*abc #完全匹配文件名有abc

[root@localhost ~]# ll ?abc #匹配一个前面有一个字符的abc文件

[root@localhost ~]# ll [0-9]abc #匹配abc之前任意一个以数字开头

[root@localhost ~]#ll [^0-9]abc #匹配abc前面不是一个数字的字符文件名

3.5.2 Bash中其他特殊符号

单引号和双引号

[root@localhost ~]# name=abc

[root@localhost ~]# echo '$name'

[root@localhost ~]# echo "$name"

[root@localhost ~]# echo '$(date)'

[root@localhost ~]# echo "$(date)"

#单引号只输入里面的内容，作为字符串输出，没有什么意义

#双引号里面是可以执行里面的命令，有特殊含义

4 Bash变量

什么是变量？

变量是计算机内存的单元，其中存放的值可以改变。当Shell脚本需要保存一些信息时，如一个文件名或是一个数字，就把它存放在一个变量中。每个变量有一个名字，所以很容易引用它。使用变量可以保存有用信息，使系统获知用户相关设置，变量也可以用户保存暂时信息。

变量设置规则

变量名称可以由字母、数字和下划线组成，但不能以数字开头。

变量的默认类型都是字符串型

变量用等号连接值，等号左右两侧不能有空格。

变量的值如果有空格，需要使用单引号或者双引号苦括起来。

不能使用标点符号

4.1 用户自定义变量

本地变量

#变量定义

[root@localhost ~]# name=test

#变量调用

[root@localhost ~]# echo $name

#变量叠加

[root@localhost ~]# aaa=123

[root@localhost ~]# aaa="$aaa"456

[root@localhost ~]# aaa="${aaa}"789

#上面两个方式作用都是一样的

#变量查看

[root@localhost ~]# set

#变量删除

[root@localhost ~]# unset name

4.2 环境变量

这种变量主要保存的是和系统操作相关的数据。

设置环境变量

#申明变量

export 变量名=变量值

#查询环境变量

env

#删除变量

unset 变量名

#设置成可读变量，不可删除

readonly 变量名

export name=test #将name变量名设置成环境变量

4.3 位置参数变量

这种变量主要是用来向脚本中传递参数或者数据的，变量名不能自定义，变量作用是固定的，

举例：

[root@localhost ~]# vim canshu02.sh

#!/bin/bash

echo $0 #表示接受命令本身

echo $1 #接受第一个参数

echo $2 #接受第二个参数

echo $3 #接受第三个参数

[root@localhost ~]# ./canshu02.sh 111 222 333

./canshu02.sh

111

222

333

[root@localhost ~]# vim canshu03.sh

#!/bin/bash

sum=$(( $1+$2 ))

echo "sum is : $sum"

#举例说明$\*和$@的区别

[root@localhost ~]# vim canshu04.sh

#!/bin/bash

for i in "$\*"

do

echo $i

done

for y in "$@"

do

echo $y

done

[root@localhost ~]# ./canshu04.sh 111 222 333 444

111 222 333 444

111

222

333

444

#$\*将所有参数看成一个整体，$@传递的每个参数区分对待

4.4 预定义变量

是Bash中已经定义好的变量，变量名不能自定义，变量作用也是固定的。

举例：

#判断上一条命令是否正常执行，正常执行返回0，不正确执行返回1

[root@localhost ~]# echo $?

#!/bin/bash

#输出当前进程的PID,这个进程PID就是这个脚本执行时，生成的进程的PID

echo "The current process is $$"

#使用find命令在root目录下查找hello.sh文件

#符号&的意思是把命令放入后台执行

find /root -name hello.sh &

echo "The last one Daemon process is $!"

举例：接收键盘输入

[root@localhost ~]# read [选项][变量名]

选项：

-p "提示信息" 等待read输入时，输出提示信息

-t秒数 read命令一直等待用户输入，使用此选项可以指定等待时间

-n 字符数 read命令只接受指定的字符数，就会执行

-s 隐藏输入的数据，适用于机密信息的输入

#!/bin/bash

read -t 30 -p " Please input your name: " name

#提示后，并等待30秒，把用户的输入保存到变量name中

echo "$name"

read -s -t 30 -p " Please input your age: " age

#年龄是隐私，所以用-s 选项隐藏输入

echo -e "\n"

#-e表示隐藏输出的\n

echo " $age "

read -n 1 -t 30 -p "Please select your gender[M/F]:" gender

echo -e "\n"

echo "Sex is $gender"

5 Bash运算符

5.1 数值运算与运算符

5.1.1 数值运算

declare声明变量类型

[root@localhost ~]# declare [+/-][选项]变量名

选项：

- 给变量设定类型属性

+ 取消变量的类型属性

-i 将变量声明为整数型（integer）

-x 将变量声明为环境变量

-p 显示指定变量的被声明的类型

数值运算-方法1

#给变量aa和bb赋值

[root@localhost ~]# aa=111

[root@localhost ~]# bb=222

[root@localhost ~]# declare -i cc=$aa+$bb

expr或let数值运算工具-方法2

[root@localhost ~]# aa=111

[root@localhost ~]# bb=222

[root@localhost ~]# dd=$(expr $aa + $bb )

#dd的值是aa和bb的和。注意“+”号左右两侧必须有空格

$((运算式)) 或 “$[运算式]”-方法3

[root@localhost ~]# aa=111

[root@localhost ~]# bb=222

[root@localhost ~]# ff=$(( $aa+$bb ))

[root@localhost ~]# ee=$[ $aa+$bb ]

5.1.2 运算符

举例：

#虽然乘和除的优先级高于加，但是通过小括号可以调整运算优先级

[root@localhost ~]# gg=$(( (11+4)\*2/3))

#14不能被3整除，余数是2

[root@localhost ~]# bb=$(( 14%3 ))

#逻辑与运算只有想与的两边都是1，与的结果才是1，否则与的结果是0

[root@localhost ~]# dd=$(( 1 && 0 ))

5.2 变量测试与内容替换

举例：

测试x=${y-新值}

#删除变量y

[root@localhost ~]# unset y

#进行测试

[root@localhost ~]# x=${y-new}

[root@localhost ~]# echo $x

#因为变量y不存在，所有x=new

6 环境变量配置文件

6.1 环境变量配置文件简介

6.1.1 source命令

[root@localhost ~]# source 配置文件

或者

[root@localhost ~]# .配置文件

6.1.2 配置文件简介

环境变量配置文件中主要是定义对系统的操作环境生效的系统默认环境变量，比如PATH、HISTSIZE、PSI、HOSTNAME等默认环境变量。

[root@localhost ~]# echo $PATH

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin

[root@localhost ~]# PATH="$PATH":/root #这样设置只是临时生效，下次重启后就失效了

[root@localhost ~]# echo $PATH

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin:/root

我们通过set查看的配置文件

系统上面的配置文件目录

/etc/profile #其中的配置是对所有登录过系统的用户生效

/etc/profile.d/\*.sh #其中的配置是对所有登录过系统的用户生效

~/.bash\_profile #对当前用户配置生效

~/.bashrc #对当前用户配置生效

/etc/bashrc #其中的配置是对所有登录过系统的用户生效

6.2 环境变量配置文件作用

/etc/profile的作用：

USER变量

LOGNAME变量

MAIL变量

PATH变量

HOSTNAME变量

HISTSIZE变量

umask

调用/etc/profile.d/\*.sh文件

~/.bash\_profile的作用：

调用~/.bashrc文件

在PATH变量后加入了“:$HOME/bin”这个目录

6.3 其他配置文件和登录信息

6.3.1 注销时生效的环境变量配置文件

~/.bash\_logout

6.3.2 其他配置文件

~/bash\_history

6.3.3 Shell登录信息

本地终端欢迎信息：/etc/issue

远程终端欢迎信息：/etc/issue.net

转义符在/etc/issue.net文件中不能使用

是否显示此欢迎信息，是有ssh的配置文件/etc/ssh/sshd\_config决定，加入“Banner /etc/issue.net”行才能显示（记得重启SSH服务）

登陆后欢迎信息：/etc/motd

不管是本地登录，还是远程登录，都可以显示此欢迎信息

7 基础正则表达式

7.1 正则表达式与通配符

正则表达式用来在文件中匹配符合条件的字符串，正则是包含匹配。grep、awk、sed等命令可以支持正则表达式。

通配符用来匹配符合条件的文件名，通配符是完全匹配。ls、find、cp这些命令不支持正则表达式，所以只能使用shell自己的通配符来进行匹配了。

7.2 基础正则表达式

“\*”前一个字符匹配0次或者任意多次

#匹配所有的内容，包括空白行

[root@localhost ~]# grep "a\*" test.txt

#匹配至少包含一个a的行

[root@localhost ~]# grep "aa\*" test.txt

#匹配至少包含两个连续a的行

[root@localhost ~]# grep "aaa\*" test.txt

#匹配至少包含四个连续a的行

[root@localhost ~]# grep "aaaaa\*" test.txt

"."匹配除了换行符外任意一个字符

#s..d 会匹配在s和d这两个字母之间一定有两个字符的单词

[root@localhost ~]# grep "s..d" test.txt

#匹配在s和d字母之前有任意字符

[root@localhost ~]# grep "s.\*d" test.txt

#匹配所有内容

[root@localhost ~]# grep ".\*" test.txt

"^" 匹配行首，“$” 匹配行尾

#匹配以大写字母A开头的行

[root@localhost ~]# grep "^A" test.txt

#匹配以小写字母b开头的行

[root@localhost ~]# grep "b$" test.txt

#匹配空白行

[root@localhost ~]# grep "^$" test.txt

“[ ]”匹配中括号中指定的任意一个字符，只匹配一个字符

#匹配a和d字母中，要么是b、要么是c

[root@localhost ~]# grep "a[bc]de" test.txt

#匹配任意一个数字

[root@localhost ~]# grep "[0-9]" test.txt

#匹配用小写字母开头的行

[root@localhost ~]# grep "^[a-z]" test.txt

"[^]" 匹配除中括号的字符以外的任意一个字符

#匹配不用数字开头的行

[root@localhost ~]# grep "^[^0-9]" test.txt

#匹配不用字母开头的行

[root@localhost ~]# grep "^[^a-zA-Z]" test.txt

"\{n\}" 表示前面的字符恰好出现n次

#匹配a字母连续出现三次的字符串

[root@localhost ~]# grep "a\{3\}" test.txt

#匹配包含连续的三个数字的字符串

[root@localhost ~]# grep "[0-9]{3\}" test.txt

"\{n,\}" 表示其前面的字符出现不小于n次

#匹配最少用连续三个数字开头的行

[root@localhost ~]# grep "^[0-9]\{3,\}[a-z]" test.txt

"\{n,m\}" 匹配其前面的字符至少出现n次，最多出现m次

#匹配在字母a和g字母之前有最少一个b,最多三个b

[root@localhost ~]# grep "ab\{1,3\}g" test.txt

7.3 字符截取命令

7.3.1 cut字段提取命令

[root@localhost ~]# cut [选项] 文件名

选项：

-f 列号 提取第几列

-d 分隔符 按照指定分隔符分割列

举例：

#截取/etc/passwd下面，指定分隔符为“：”，第一列的所有用户名

[root@localhost ~]# cut -d ":" -f 1 /etc/passwd

注：cut不能作用于空格作为分隔符，并提取数据。只能用作制表符截取字符。

7.3.2 printf命令

printf '输出类型输出格式' 输出内容

输出类型：

%ns 输出字符串。n是数字指代输出的几个字符

%ni 输出整数。n是数字指代输出几个数字

%m.nf 输出浮点数。m和n是数字，指代输出的整数位数和小数位数。如%8.2f代表共输出8位 数，其中2位是小数，6位整数。

输出格式：

[root@localhost ~]# printf '%s %s %s\n' 1 2 3 4 5 6

7.3.3 awk命令

#打印提取第一列，第5列和第六列，并且以制表符分割开

[root@localhost ~]# df -h | awk '{print $1 "\t" $5 "\t" $6}'

#查看根分区的硬盘空间大小，用cut截取掉%，提取第一列

[root@localhost ~]# df -h | grep vda1 | awk '{print $5}' | cut -d "%" -f 1

FS内置变量

举例：

[root@localhost ~]# awk 'BEGIN{FS=":"}{print $1 "\t" $3}' /etc/passwd

root 0

bin 1

daemon 2

adm 3

lp 4

sync 5

shutdown 6

halt 7

mail 8

operator 11

games 12

ftp 14

nobody 99

systemd-network 192

dbus 81

polkitd 999

libstoragemgmt 998

rpc 32

ntp 38

abrt 173

sshd 74

postfix 89

chrony 997

tcpdump 72

syslog 996

lighthouse 1000

7.3.4 sed命令

sed是一种几乎包括在所有UNIX平台的轻量级流编辑器。sed主要是用来将数据进行选取、替换、删除、新增的命令。

[root@localhost ~]# sed [选项] ‘[动作]’ 文件名

选项：

-n 一般sed命令会把所有数据都输出到屏幕，如果加入此选项，则只会把经过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

举例：

#查看文件的第二行,它会打印所有的数据

[root@localhost ~]# sed '2p' test.txt

#只查看第二行的数据

[root@localhost ~]# sed -n '2p' test.txt

#删除第二行到三行的数据，但不修改文件本身

[root@localhost ~]# sed '2,4d' test.txt

#在第三行后追加hello

[root@localhost ~]# sed '3a hello' test.txt

字符串替换

sed 's/旧字符/新字符/g' 文件名

#在第四行中，把abc换成cde

[root@localhost ~]# sed '3s/abc/cde/g' test.txt

#直接将数据写入文件

[root@localhost ~]# sed -i '3s/abc/cde/g' test.txt

#同时将数据修改为空

[root@localhost ~]# sed -e 's/abc//g ; s/def//g' test.txt

8 条件判断

8.1 文件运算符

操作符

作用

-b 文件

判断该文件是否为块设备文件，如果是，则返回真。

-c 文件

判断该文件是否为字符设备文件，如果是，则返回真。

-d 文件

判断该文件是否为目录文件，如果是，则返回真。

-e 文件

判断该文件是否存在，如果存在，则返回真。

-f 文件

判断该文件是否为普通文件，如果是，则返回真。

-L 文件

判断该文件是否为符号链接文件，如果是，则返回真。

-p 文件

判断该文件是否为管道文件，如果是，则返回真。

-s 文件

判断该文件是否为非空，如果非空，则返回真。

-S 文件

判断该文件是否为套接字文件，如果是，则返回真。

-r 文件

判断该文件是否为可读，如果是，则返回真。

-w 文件

判断该文件是否为可写，如果是，则返回真。

-x 文件

判断该文件是否为可执行，如果是，则返回真。

-u 文件

判断该文件是否拥有SUID权限，如果是，则返回真。

-g 文件

判断该文件是否拥有SGID权限，如果是，则返回真。

-k 文件

判断该文件是否拥有SBit权限，如果是，则返回真。

两种判断格式

[root@localhost ~]# test -e /root/test.yaml

[root@localhost ~]# [ -e /root/test.yaml ]

举例：

#第一个判断命令如果正确执行，则打印yes,否则打印no

[root@localhost ~]# [ -d /root ] && echo "yes" || echo "no"

#判断文件是否有可写的权限

[root@localhost ~]# [ -w /root/dash.yaml ] && echo "yes" || echo "no"

yes

8.2 两个文件之间进行比较

测试选项

作用

文件1 -nt 文件2

判断文件1的修改时间是否比文件2的新，如果新则返回真。

文件1 -ot 文件2

判断文件1的修改时间是否比文件2的旧，如果旧则返回真。

文件1 -ef 文件2

判断文件1是否和文件2的Inode号一致，可以理解为两个文件是否为同一个文件，这个判断用于判断硬链接是很好的方法

举例：

[root@localhost ~]# [ /root/dash.yaml -ef /tmp/dash.yaml ] && echo yes || echo no

8.3 两个整数之间比较

测试选项

作用

整数1 -eq 整数2

判断整数1是否和整数2 相等，相等，则返回真。

整数1 -ne 整数2

判断整数1是否和整数2 不相等，不相等，则返回真。

整数1 -gt 整数2

判断整数1是否大于整数2 ，大于，则返回真。

整数1 -lt 整数2

判断整数1是否小于整数2 ，小于，则返回真。

整数1 -ge 整数2

判断整数1是否大于等于整数2 ，大于等于，则返回真。

整数1 -le 整数2

判断整数1是否小于等于整数2 ，小于等于，则返回真。

举例：

[root@localhost ~]# [ 23 -gt 22 ] && echo yes || echo no

8.4 字符串的判断

测试选项

作用

-z 字符串

判断字符串是否为空，为空则返回真。

-n 字符串

判断字符串是否为非空，非空则返回真。

字符串1 == 字符串2

判断字符串1是否和字符串2 相等，相等则返回真。

字符串1 != 字符串2

判断字符串1是否和字符串2不相等，不相等则返回真。

举例：

[root@localhost ~]# name=sc

[root@localhost ~]# [ -z "$name" ] && echo yes || echo no

no

[root@localhost ~]# [ -n "$name" ] && echo yes || echo no

yes

[root@localhost ~]# aa=abc

[root@localhost ~]# bb=abc

[root@localhost ~]# [ "$aa" == "$bb" ] && echo yes || echo no

yes

[root@localhost ~]# cc=ade

[root@localhost ~]# [ "$aa" == "$cc" ] && echo yes || echo no

no

8.5 多重条件判断

测试选项

作用

判断1 -a 判断2

逻辑与，判断1和判断2都成立，最终的结果才为真。

判断1 -o 判断2

逻辑或，判断1和判断2有一个成立，最终的结果就为真。

! 判断

逻辑非，使原始的判断式取反

举例:

[root@xiangbiao-docker ~]# aa=24

[root@xiangbiao-docker ~]# [ -n "$aa" -a "$aa" -gt 23 ] && echo yes || echo no

yes

[root@xiangbiao-docker ~]# [ -n "$aa" -a "$aa" -gt 24 ] && echo yes || echo no

no

9 流程控制

9.1 if语句

9.1.1 if条件语句

if [ 条件判断 ];then

程序

fi

或者

if [ 条件判断 ]

then

程序

fi

举例：

#统计根分区使用率

#!/bin/bash

#统计根分区使用率

rate=$(df -h | grep /dev/vda1 | awk '{print $5}' | cut -d "%" -f 1)

if [ $rate -ge 20 ]

then

echo "error 10"

fi

9.1.2 if else条件语句

if [ 条件判断 ]

then

条件成立时，执行此程序

else

条件不成立时，执行的另外一个程序

fi

举例1：

#！/bin/bash

#备份mysql数据库

#同步系统时间

ntpdate asia.pool.ntp.org &>/dev/null

#把当前系统时间按照“年月日”格式赋予变量date

date=$( date+%y%m%d )

#统计mysql的数据库大小，并把大小赋予size变量

size=$( du -sh /var/lib/mysql )

if [ -d /tmp/dbback ]

then

echo "Date: $date!" > /tmp/dbback/dbinfo.txt

echo "Date size: $size" >> /tmp/dbback/dbinfo.txt

cd /tmp/dbback

tar -zcf mysql-lib-$date.tar.gz /var/lib/mysql dbinfo.txt &>/dev/null

rm -rf /tmp/dbback/dbinfo.txt

else

mkdir /tmp/dbback

echo "Date: $date!" > /tmp/dbback/dbinfo.txt

echo "Date size: $size" >> /tmp/dbback/dbinfo.txt

cd /tmp/dbback

tar -zcf mysql-lib-$date.tar.gz /var/lib/mysql dbinfo.txt &>/dev/null

rm -rf /tmp/dbback/dbinfo.txt

fi

举例2：

#!/bin/bash

#全量备份etc目录

date=$(date +%y%m%d)

size=$(du -sh /home)

if [ -d /tmp/dbback ]

then

echo "Date is : $date" > /tmp/dbback/db.txt

echo "size is : $size" >> /tmp/dbback/db.txt

cd /tmp/dbback

tar -zcf home\_$date.tar.gz /home db.txt &>/dev/null

rm -rf /tmp/dbback/db.txt

else

mkdir /tmp/dbback

echo "Date is : $date" > /tmp/dbback/db.txt

echo "size is : $size" >> /tmp/dbback/db.txt

cd /tmp/dbback

tar -zcf home\_$date.tar.gz /home db.txt &>/dev/null

fi

举例3:

#!/bin/bash

port=$(nmap -sT 172.20.1.59 | grep tcp | grep http | awk '{print $2}')

if [ "$port" == "open" ]

then

echo "$(date) httpd is ok!" >> /tmp/httpd\_acc.log

else

/etc/rc.d/init.d/httpd restart &>/dev/null

echo "$(date) httpd reboot~!" >> /tmp/httpd\_err.log

fi

9.1.3 多分支if条件语句

if [ 条件判断式1 ]

then

当条件判断式1成立时，执行程序1

elif [ 条件判断式2 ]

then

当条件判断式2成立时，执行程序2

...省略更多条件

else

当所有条件都不成立时，最后执行此程序

fi

举例：

#!/bin/bash

#接收键盘输入，并赋予变量file

read -p "Please input a filename: " file

#判断file变量是否为空

if [ -z "$file" ]

then

echo "error,请输入文件名"

exit 1

#判断file的值是否存在

elif [ ! -e "$file" ]

then

echo " 你的文件不存在"

exit 2

#判断file的值是否为普通文件

elif [ -f "$file" ]

then

echo "文件是普通文件"

#判断file的值是否为目录文件

elif [ -d "$file" ]

then

echo "文件是目录文件"

else

echo "文件是其他类型的"

fi

9.2 case语句

case $变量名 in

"值1")

如果变量的值等值1，则执行程序1

;;

"值2")

如果变量的值等值2，则执行程序2

;;

……省略其他分支

\*)

如果变量的值都不是以上的值，则执行程序

;;

esac

举例：

#!/bin/bash

#测试case语句

echo "如果想去上班，请输入“1”"

echo "如果不想上班，请输入“2”"

echo "如果想居家办公，请输入“3”"

read -t 30 -p "请输入你的选择:" choose

case "$choose" in

"1")

echo "想去上班！"

;;

"2")

echo "不想去上班！"

;;

"3")

echo "居家办公！"

;;

\*)

echo "只能输入1或者2或者3"

;;

esac

9.3 for循环

语法一

for 变量 in 值1 值2 值3...

do

程序

done

举例：

#!/bin/bash

#打印1??6的循环

for i in 1 2 3 4 5 6

do

echo $i

done

语法二

for (( 初始值;循环控制条件;变量变化 ))

do

程序

done

举例1：

#!/bin/bash

#使用for循环计算1-100的和

s=0

for (( i=1;i<=100;i=i+1 ))

do

s=$(( $s+$i ))

done

echo "$s"

举例2：

#!/bin/bash

#批量添加指定数量的用户

read -t 30 -p "Please input user name:" name

read -t 30 -p "Please input the mumber of users:" num

read -t 30 -p "Please input the password of users:" pass

if [ ! -z "$name" -a ! -z "$num" -a ! -z "$pass" ]

then

y=$(echo $num | sed 's/[0-9]\*$'//g)

if [ -z "$y" ]

then

for (( i=1;i<=$num;i=i+1 ))

do

/usr/sbin/useradd $name$i &>/dev/null

echo $pass | /usr/bin/passwd --stdin $name$i &>/dev/null

done

fi

fi

9.4 while循环

while循环是不定循环，也称条件循环。只要条件判断式成立，循环就会一直继续，直到条件判断式不成立，循环才会停止，这就和for的固定循环就不太一样了。

while [ 条件判断式 ]

do

程序

done

举例：

#!/bin/bash

#从1加到100

i=1

s=0

while [ $i -le 100 ]

#如果变量i的值小于等于100，则执行循环

do

s=$(( $s+$i ))

i=$(( $i+1 ))

done

echo "The sum is: $s"

9.5 unit循环

until循环，和while循环相反，until循环时，只要条件判断式不成立，则进行循环，并执行循环程序。一旦循环条件成立，则终止循环。

#!/bin/bash

#从1加到100

i=1

s=0

until [ $i -gt 100 ]

#循环直到变量i的值大于100，循环就停止了

do

s=$(( $s+$i ))

i=$(( $i+1 ))

done

echo "The sum is : $s "