shelloldboy笔记

基础必备:

1.vi/vim编辑器的熟练使用,SSH终端以及.vimrc的设置等等

2.命令基础: Linux150个常用命令

3.常见Linux网络服务部署及排错.如:crond,nfs,rsync,inotify,lanmp,sersync.ssh等.

# 第1章 shell脚本编程应用指南

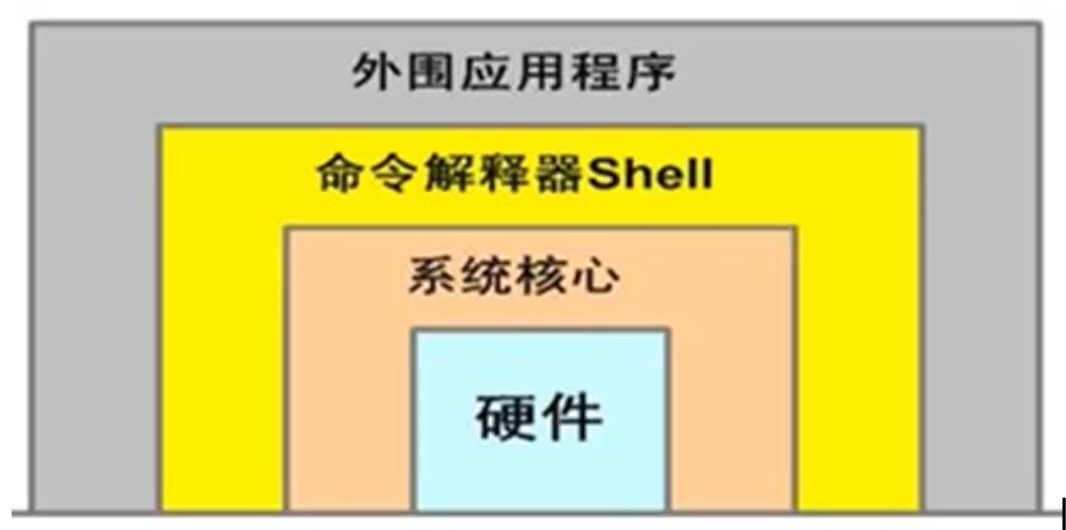
# 1.1 shell脚本简介

在解释shell脚本这个名词之前,我们先说下什么事shell?

### 1.1.1 设么是shell

shell是一个**命令解释器**,他在操作系统的最外层,负责直接与用户对话,把用户的输入解释给操作系统,并处理各种各样的操作系统的输出结果,输出到屏幕返回给用户.这种对话方式可以是交互的方式(键盘输入命令,可以立即得到shell的回应),或交互式(脚本)的方式.

下图中的黄色部分就是命令解释器,shell处于操作系统的位置

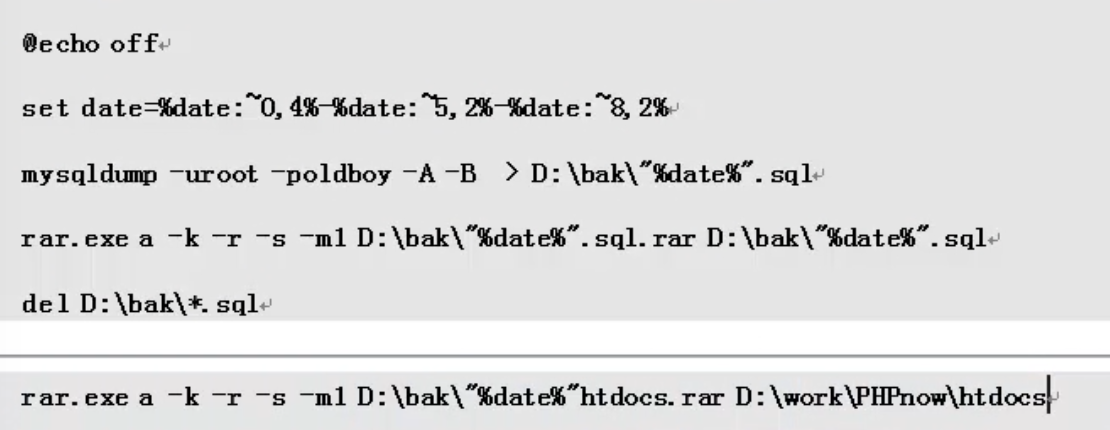


提示: shell英文为贝壳的意思,命令解释器shell就像一个贝壳一样包住了系统核心

### 1.1.2什么是shell脚本

当Linux命令或语句不在命令下执行(严格说,命令行也是shell),而是通过一个程序文件执行时,该程序就被称为shell脚本或shell程序,shell程序很类似DOS系统下的批处理程序(.bat).用户可以在shell脚本中敲入一系列的命令及命令语句组合.这些命令,变量和流程控制语句等有机的结合起来就形成了一个功能强大的shell脚本.

下面是在windows下利用批处理程序bat开发的备份网站及数据库的脚本.



范例1:.清楚/var/log下messages日志文件的简单命令脚本

|  |
| --- |
| # version 0.1  cd /var/log  cat /dev/null > messages  echo "Logs cleaned up." |

缺点:

1.root用户操作;2.没有流程控制

|  |
| --- |
| #!/bin/bash  # version 0.2 升级版  LOGDIR=/var/log  ROOT\_UID=0  # root用户执行,友好提示  if [ "$UID" -ne "$ROOT\_UID" ]  then  echo "Must be root to run this script."  exit 1  fi  # 不成功执行后面的  cd $LOG\_DIR || {  echo "Cannot change to necessary directory." >&2  exit 1  }  cat /dev/null > messages && echo "Logs cleaned up."  exit 0  # 0表示成功,1表示失败 |

拓展:清空日志及文件内容的三种方法

echo >test.log

> test.log

cat /dev/null > test.log

### 1.1.3shell脚本在运维工作中的作用地位

shell脚本擅长处理**纯文本类型**的数据.儿Linux中几乎所有的配置文件,日志文件都是纯文本类型的文件.因此,如果学号shell脚本语言,就可以利用他在Linux系统中发挥巨大的作用.

下面是shell脚本在运维工作中的作用地位形象图:



### 1.1.4 脚本语言的种类

1.1.4.1 shell脚本语言的种类

在UNIX/LINUX中主要有两大类Shell

\*\* Burner shell(包括sh,ksh,and bash)

\*\* C shell(包括 csh and tcsh)

shell脚本语言是**弱类型**语言,较为通用的shell又标准的Bourne shell(sh)和C shell(csh).其中Bourne shell已经被**bash shell**取代

查看系统的SHELL:

|  |
| --- |
| [root@localhost /]# cat /etc/shells #CentOS 7.3的版本  /bin/sh  /bin/bash  /sbin/nologin  /usr/bin/sh  /usr/bin/bash  /usr/sbin/nologin |

1.1.4.2 其他运维人员常用的脚本语言

php,perl,python

1.1.4.3 shell与php/perl/python语言的区别和优势

shell的优势在于处理操作系统底层的业务,因为有大量的系统命令为它做支撑,2000多个命令都是shell编程的有力支撑,特别是grep,awk,sed等.例如: 一键软件安装,优化,监控报警脚本.,常规的业务应用,shell开发简单快速,符合运维的大原则.

php,python优势在于开发运维工具,web界面的管理工具,以及web业务的开发等.处理一键软件安装,优化,报警脚本.常规的业务应用等php/python也是能都坐到的.但是开发小路和复杂度比用shell就差很多了.

### 1.1.5 常用操作系统的默认shell

Solaris是FreeBSD缺省的Bourne shell(sh).

AIX是Korn shell(ksh)

HP-UX是POSIX shell(sh)

CentOS是bash

查看方法: echo $SHELL 或者 grep root /etc/passwd

### 1.1.6 shell脚本的创建和执行

1.1.6.1 shell脚本的建立

在Linux系统中,shell脚本通常是在编辑器(vi/vim)中编写,由Unix/Linux命令,bash shell命令,程序结构控制语句和注释等内容组成.

**\*\* 脚本开头**

一个规范的shell脚本的第一行会**指出由哪个程序(解释器)来执行脚本中的内容**,在 linux bash编程一般为:

|  |
| --- |
| #!/bin/bash  或  #!/bin/sh |

其中开头的"#!"又称为幻数,在执行bash脚本的时候,内核会根据"#!"后的解释器来确定该由哪个程序解释脚本中的内容.注意: 这一行必须在每个**脚本顶端的第一行**,如果不是第一行则为脚本注释,例如:

|  |
| --- |
| #!/bin/bash  echo "oldboy start"  #!/bin/bash  #!/bin/sh # 这是注释了  echo "oldboy end" |

sh和bash的区别:

|  |
| --- |
| ls -l /bin/sh  ll /bin/sh |

sh为bash的软链接,推荐标准写法#!/bin/bash

Bash是GNU/Linux默认的shell,和Bourne Shell(sh)兼容,Bash采用了Korn shell(ksh)和Cshell(csh)的特色.符合IEEE POSIX P1003.2/ISO9945.2 Shell and Tools标准.

Centos'和RedHat Linux下默认的shell均为bash.因此,在写shell脚本的时候,我们的脚本的开头也可以不加"#!/bin/bash".但如果当前系统默认的shell非bash shell时,比如tcsh,那么就有必要写#!了.否则脚本文件执行的结果可能不是你想要的.建议养成好的编程习惯,不管什么脚本都加上开头语言标识.

如果脚本的开头的第一行不指定解释器,那么,就要用对应的解释器来解释执行.例如:如果是shell脚本就用bash test.sh执行.如果是python脚本,就用python test.py执行.

\*\* 脚本注释

#后面的都是注释.养成好的习惯一定要写注释

1.1.6.2 shell脚本的执行

当shell脚本以非交互的方式(文件)运行时,它会查找环境变量ENV,该变量指定了一个环境文件(通常是.bashrc, .bash\_profile, /etc/bashrc, /etc/profile等),然后从该环境变量文件开始执行,当读取了ENV文件后,shell才开始执行shell脚本中的内容.

**crond任务是把系统环境变量在脚本中重新定义.**

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 不需要给x权限,可以执行,推荐

②.path/script-name 或 ./script-name 必须有执行权限

**③source script-name或 . script-name**

执行说明:

第一种方法是当前脚本文件本身没有可执行权限(即文件x位为-号)时常用的方法,这里推荐用bash执行,或者文件开头没有制定解释器

第二种方法需要先将脚本文件的权限改为可执行,具体方法;chmod u+x script-name 或 chmod 755 script-name.然后通过脚本路径就可以直接执行脚本了

在生产环境中,不少读者在写完shell脚本后,由于疏忽忘记给予该脚本执行权限,就直接应用,结果导致脚本没有按自己的意愿手动或定时执行,所以最好用第一种方式执行脚本.

第三种方法通常是使用source或"." 点号读入或加载制定的shell脚本文件,然后,以此执行制定shell脚本文件中的所有语句.这些语句将作为当前父shell脚本进程的一部分运行.因此,使用source或者"."可以将自身脚本中的变量的值或函数等的返回值传递到当前的父shell脚本中使用.这是第三种方法和前两种方法的最大区别.注意!!!

source或者"."点号命令的功能是在当前shell中执行source或者"."点号加载并执行的相关的脚本文件中的命令及语句.而不是产生一个子shell来执行命令文件中的命令.

如果脚本中有需求引用其他脚本的内容或者配置文件时,最好用"."或source加载该脚本或配置文件.

举例:

|  |
| --- |
| [root@localhost ~]# cat test.sh  user=`whoami`  [root@localhost ~]# sh test.sh  [root@localhost ~]# echo $user  执行结果????  空 |

1.1.6.3 shell脚本开发基本规范及习惯

1) 开头指定脚本解释器

#!/bin/sh 或 #!/bin/bash

2) 开头加版本版权等信息

#Date: 2019-07-26 12:02:44

#Author: Created by pipi

#Mail: pizhiui313@gmail.com

#Functions: This script function is ....

#Version: 1.1

提示:可配置编辑文件自动加上以上信息,修改~/.vimrc配置文件

3) 脚本中不用中文注释

尽量用英文注释

4) 脚本以.sh扩展名

5) 代码书写习惯

1.成对内容一次性写完 {} () "" `` ''

2.[]中括号两端要有空格,[ ]

3.流程控制语句一次性写完.

if 条件内容

then

fi

for

do

内容

done

6) 通过缩进让代码易读

1.1.6.4 shell脚本开发制度及规范

1.1.6.5 shell帮住与资料推荐

http://www.gnu.org/software/bash/manual/bash.html

man bash

unix shell范例精解

shell-abs-3.9.1-cn

# 1.2 shell注释

单行注释使用 # 符号

多行注释使用如下规则：

|  |
| --- |
| :<<BLOCK'  ....注释内容  'BLOCK |
| :<<'BLOCK  ....注释内容  BLOCK' |
| :<<'  ....注释内容  ' |

# 1.2 shell变量基础及深入

## 1.2.0 什么是变量

略,程序猿都懂

## 1.2.1 变量类型

变量可分为两类: 环境变量(全局变量)和局部变量

环境变量也可称为全局变量,可以在创建他们的shell及其派生出来的任意子进程shell中使用.局部变量只能在创建他们的shell函数或脚本中使用.还有一些变量是用户创建的,其他的则是专用shell变量.

## 1.2.2 环境变量

环境变量用于定义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,命令目录经,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(**crond任务要注意**).

环境变量可以在命令行中设置,但用户退出时这些变量值也会丢失,因此最好在用户家目录下的.bash\_profile文件中或全局配置/etc/profile,/etc/bashrc文件或者/etc/profile.d中定义.将环境变量让入profile文件中,每次用户登录时这些变量值都将被初始化.

传统上,所有**环境变量均为大写**.环境变量应用于用户进程前,都应该用**export命令导出**

环境变量可用在创建他们的shell和从该shell派生的任意子shell或进程中.他们通常被称为全局变量以区别局部变量.通常,环境变量应该大写.

有一些环境变量,比如HOME,PATH,SEHLL,UID,USER等,在用户登录之前就已经被/bin/login程序设置好了.通常环境变量定义并保存在用户家目录下的.bash\_profile文件中.具体的环境变量说明参考下表:

通过**env**命令即可看见系统所有的环境变量

## 1.2.3 自定义环境变量(全局变量)

设置环境变量

如果想设置环境变量,就要在给变量赋值之后或设置变量是使用export命令.

格式:

①.export 变量名=value

②.变量名=value; export 变量名

③.declare -x 变量名=value

**自定义全局环境变量实例:**

\*\* 全局环境变量配置: /etc/profile /etc/bashrc

\*\* 用户的环境变量: ~/.bash\_profile ~/.bashrc 都可以

|  |
| --- |
| /etc/profile.d目录: 登录显示加载内容放在此目录,一般是脚本  /etc/motd文件: 登录显示加载内容,一般是文本 |

## 1.2.4 显示与取消环境变量

\* 通过echo命令打印环境变量(printf也可以) **printf和echo的区别**

$HOME 用户登录时进入的目录

$UID 当前用户的UID(用户标识)

$PWD 当前工作目录的绝对路径名

$SHELL 当前SHELL

$USER 当前用户

.....

\* env(print env) 或set 显示默认的环境变量

\* unset 消除本地变量和环境变量

unset PATH 取消系统环&境变量,一般不要

## 1.2.5 局部变量

#### 1.2.5.1 定义本地变量

本地变量在用户当前的shell生命周期的脚本中使用.如,本地变量OLDBOY全会为ett098,政治只在用户当前shell生存期中有意义.如果在shell中启动另一个进程或退出,本地变量OLDBOY值将无效

1. 普通字符串变量定义

变量名=value

变量名='value'

变量名="value"

shell中变量名的要求: 一般字母,数字,下划线.字母开头

题目1.:

a=192.168.1.2

b='192.168.1.2'

c='192.168.1.2'

echo "a=$a"

echo "b=$b"

echo "c=$c"

结果: 都显示ip**,""会解释变量的内容,''不会解释变量的内容**

题目2:

a=192.168.1.2-$a

b='192.168.1.2-$a'

c="192.168.1.2-$a"

echo "a=$a"

echo "b=$b"

echo "c=$c"

答案:

a=192.168.1.2-192.168.1.2

b=192.168.1.2-$a

c=192.168.1.2-192.168.1.2-192.168.1.2

第一种变量定义a的方式直接定义变量内容,内容一般为简单连续的数字,字符串,路径名等.

第二种变量定义b的方式通过单引号定义变量.特点:输出变量是单引号是什么就输出什么.即使内容中有变量也会把变量名原样输出.此方法适合定义显示纯字符串.

第三种变量定义c的方式通过双引号定义变量.特点:输出变量是引号里的变量会经过解析后输出该变量的内容.适合与字符串中附带有变量的内容的定义.

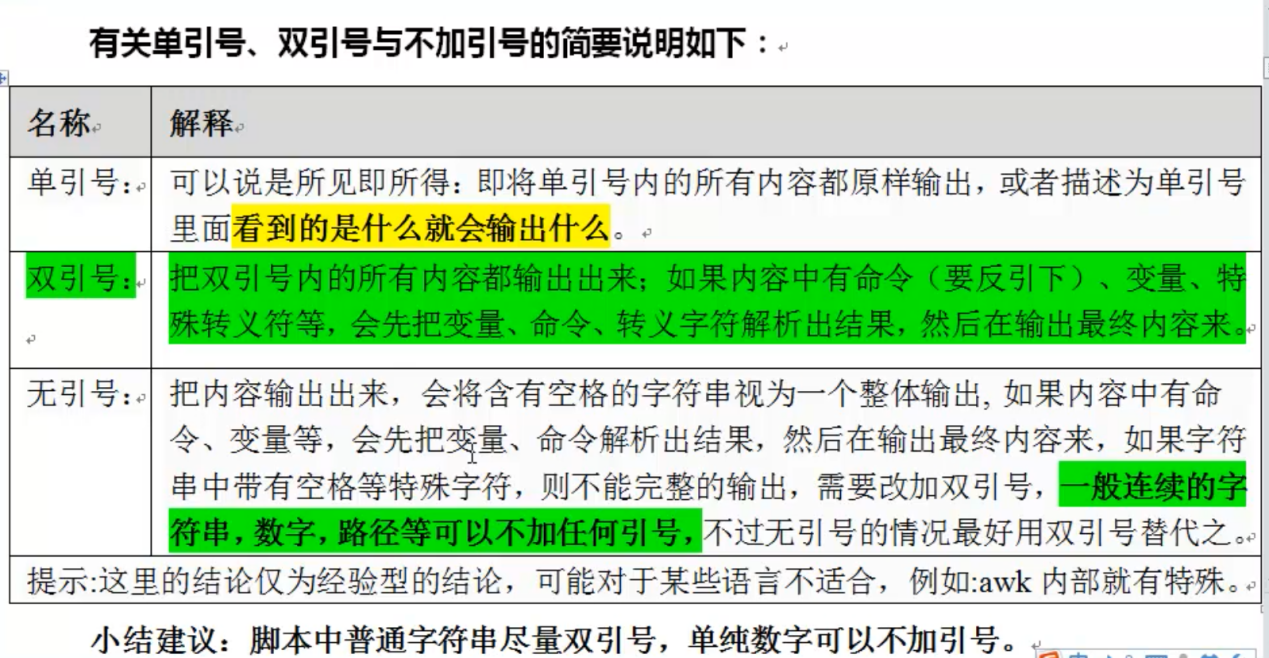
**习惯: 数字不加引号, 其他默认加双引号.**

#### 1.2.5.2 定义变量单引号,双引号与不加引号

单引号: 将单引号的内容原样输出,或者描述为单引号里面看到的什么就会输出什么'

双引号: 能够解析变量,把变量的内容解释出来再显示.

无引号: 一般连续的字符串,数字,路径 ,也会解析变量,命令结果



**实践1:**

|  |
| --- |
| [root@localhost ~]# echo 'today is `date`'  today is `date`  [root@localhost ~]# echo "today is `date`"  today is Wed Jul 26 15:37:48 CST 2017  [root@localhost ~]# echo "today is date"  today is date  [root@localhost ~]# echo "today is $(date)"  today is Wed Jul 26 15:38:22 CST 2017  [root@localhost ~]# echo today is $(date) #echo特殊性,如果是变量就会有问题(空格)  today is Wed Jul 26 15:42:06 CST 2017 |

**实践2:**

|  |
| --- |
| [root@localhost ~]# OLDBOY=testchars  [root@localhost ~]# echo $OLDBOY  testchars  [root@localhost ~]# echo '$OLDBOY'  $OLDBOY  [root@localhost ~]# echo "$OLDBOY"  testchars |

**实践3:**

|  |
| --- |
| [root@localhost ~]# cat > greo.log  testcharsl^H  oldboy  testchars  ^C  [root@localhost ~]# grep "$OLDBOY" greo.log # grep也可解释变量  testcharsl  testchars  [root@localhost ~]# grep '$OLDBOY' greo.log  [root@localhost ~]# grep $OLDBOY greo.log  testcharsl  testchars |

**实践3: awk正好是相反的,注意设置特殊用法**

|  |
| --- |
| [root@localhost ~]# **ETT=123**  [root@localhost ~]# awk 'BEGIN {print "$ETT"}'  $ETT  [root@localhost ~]# awk 'BEGIN {print '$ETT'}'  123  [root@localhost ~]# awk 'BEGIN {print $ETT}'  [root@localhost ~]# **ETT='abc'**  [root@localhost ~]# awk 'BEGIN {print "$ETT"}'  $ETT  [root@localhost ~]# awk 'BEGIN {print '$ETT'}'  [root@localhost ~]# awk 'BEGIN {print $ETT}'  [root@localhost ~]# awk 'BEGIN {print "'$ETT'"}' **# 特殊情况**  abc |

**实践4: sed问题**

|  |
| --- |
| [root@localhost ~]# sed -e /$OLDBOY/p greo.log  testcharsl  testcharsl  oldboy  testchars  testchars  [root@localhost ~]# sed /$OLDBOY/p greo.log  testcharsl  testcharsl  oldboy  testchars  testchars  [root@localhost ~]# cat greo.log  testcharsl  oldboy  testchars  [root@localhost ~]# sed -n /$OLDBOY/p greo.log  testcharsl  testchars  [root@localhost ~]# sed -n /"$OLDBOY"/p greo.log  testcharsl  testchars  [root@localhost ~]# sed -n '/'$OLDBOY'/p' greo.log  testcharsl  testchars  [root@localhost ~]# sed -n "/'$OLDBOY'/p" greo.log #没结果 |

**练习题**

**http://oldboy.blog.51cto.com/2561410/760192**

自定义变量的建议:

(1).内容为纯数字,可以不加引号

(2).没特殊情况.默认用双引号

(3).需要原样输出的时候,用单引号

变量的命名规范

1).变量命名要统一,使用全部大写字母,语义要清晰,能都正确表达变量内容的含义.过长的英文单词可采用前几个字符代替.多个单词用"\_"下划线号连接,**最好以${}或"${}"这种外面加大括号的方式来引用变量.**

2).避免无含义字符或数字

如COUNT这种没有意义的,尽量少用

3).全局变量和局部变量命名

a.脚本中的全局变量定义,如OLDBOY\_HOME或OLDBOYHOME,在变量使用时,使用{}将变量括起括"${OLDBOY\_HOME}". ${金庸}新著作

b.脚本中局部变量定义: 存在于脚本函数(function)中的变量称为局部变量,要以local方式进行声明,使之只在本函数作用域内有效,防止变量在函数中的命名与变量外部程序中变量重命名造成程序异常.

c.变量合并: 当某些变量或配置项要组合起来才有意义是,如文件的路径和文件名称,建议将要组合的变量合并到一起赋值给一个新的变量,这样既方便之后的调用,也为以后进行修改提供了方便.

**变量定义总结**: 多学习多模仿/etc/init.d/functions函数库脚本的定义思路

2.把一个命令作为变量定义的方法

两种方法: 1.``符号; 2. $(pwd)

生产环境常见应用

1.对站点按天打包生成不同的文件名

|  |
| --- |
| [root@localhost ~]# CMD=$(date +%F)  [root@localhost ~]# echo $CMD  2017-07-26  [root@localhost ~]# echo $(date +%F).tar.gz  2017-07-26.tar.gz  [root@localhost ~]# echo `date +%F`.tar.gz  2017-07-26.tar.gz |

2. 把/oldboy目录及其子目录下所有以扩展名.sh结尾的文件中包含/hostlist.txt的字符串全部替换为../idctest\_iplist.

|  |
| --- |
| [root@localhost oldboy]# **find ./ -name "oldboy.log"**  ./xiaodong/oldboy.log  ./xiaofan/oldboy.log  ./oldboy.log  [root@localhost oldboy]# **sed -i 's#oldboy#oldgirl#g' `find ./ -name "oldboy.log"`**  [root@localhost oldboy]# ll  total 8  -rw-r--r-- 1 root root 8 Jul 26 17:04 oldboy.log  -rw-r--r-- 1 root root 7 Jul 26 16:54 xiao  drwxr-xr-x 2 root root 24 Jul 26 17:04 xiaodong  drwxr-xr-x 2 root root 24 Jul 26 17:04 xiaofan  [root@localhost oldboy]# cat oldboy.log  oldgirl  [root@localhost oldboy]# **find ./ -name "oldboy.log" | xargs cat**  oldgirl  oldgirl  oldgirl |

#### 1.2.5.3 shell特殊变量 $0 $n $# $@

1. 位置变量

|  |
| --- |
| **$0** 获取当前shell脚本的文件名,如果执行有路径就包括脚本路径  **$n** 获取当前shell脚本的第n个参数值,n=1...9,当n为0时表示脚本的文件名,如果n大于9,用大括号括起来${10}  $\* 获取当前shell的所有参数,将所有的命令行参数视为单个字符串,相当于"$1$2$3" 注意与$#的区别  **$#** 获取当前执行的shell脚本后面接的参数的总个数  $@ 这个程序的所有参数 "$1" "$2" "$3" "...",这是将参数传递给其他程序的最佳方式,因为他会保留所有内嵌在每个参数里的任何空白. |

提示,$\*和$@的区别?了解

2. 进程状态变量

|  |
| --- |
| $$ 当前shell的进程号(PID)  $! 执行上一个指令的PID  **$?** 获取执行上一个指令的返回值(0为成功,非零为失败) #很常用  $\_ 在此之前执行的命令或脚本的最有个参数 |

**$0实例:**

(1) $0获取脚本的名称

|  |
| --- |
| [root@localhost ~]# cat n.sh  **#!/bin/bash**  **echo $0**  [root@localhost ~]# sh n.sh  n.sh  [root@localhost ~]# sh /root/n.sh  /root/n.sh  [root@localhost ~]# sh `pwd`/n.sh  /root/n.sh |

(2) 如果需要获取路径和文件名

|  |
| --- |
| [root@localhost ~]# cat n.sh  **#!/bin/bash**  **dirname $0**  **basename $0**  [root@localhost ~]# sh /root/n.sh  /root  n.sh  [root@localhost ~]# sh n.sh  .  n.sh  [root@localhost ~]# sh `pwd`/n.sh  /root  n.sh |

(3) 查看系统服务portmap(c5.8)脚本,摘录如下: 后来改为rpcbind 或者crond

|  |
| --- |
| 最后的帮助那里会有$0的使用 |

**$n实例:**

(1) $1 $2 ${10}

|  |
| --- |
| [root@localhost ~]# cat n.sh  **#!/bin/bash**  **echo $1**  [root@localhost ~]# sh n.sh oldboy zhangyang  oldboy  [root@localhost ~]# sh n.sh "oldboy zhangyang" # 引号表示一个参数  oldboy zhangyang  [root@localhost ~]# seq -s " " 15  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  [root@localhost ~]# seq 15|sed 's#^#$#g' |tr "\n" " " #参数准备  $1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11 $12 $13 $14 $15 [root@localhost ~]#  [root@localhost ~]# seq -s " " 15|sed 's# # $#g'  1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11 $12 $13 $14 $15  [root@localhost ~]# cat q.sh  **echo $1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11 $12 $13 $14 $15**  [root@localhost ~]# sh q.sh {a..z}  a b c d e f g h i a0 a1 a2 a3 a4 a5  [root@localhost ~]# cat q.sh  echo $1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ${11} # 超过9以后需要{}括起来  [root@localhost ~]# sh q.sh {a..z}  a b c d e f g h i j k |

**$#实例:**

|  |
| --- |
| [root@localhost ~]# cat q.sh  **echo $1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ${11}**  **echo $#**  [root@localhost ~]# sh q.sh {a..z}  a b c d e f g h i j k  26  # 控制用户传的参数个数  [root@localhost ~]# cat q.sh  **[ $# -ne 2 ] && {**  **echo "muse two"**  **exit 1**  **}**  echo "oldgirl"  [root@localhost ~]# sh q.sh tangying  muse two  [root@localhost ~]# sh q.sh tangying haha  oldgirl |

**$?实例:**

|  |
| --- |
| # 编译软件的时候,判断成功与否  # 根据返回值判断命令是否成功 |

**$\*和$@的区别**

区别仅在于加双引号的时候,即"$\*"和"$@"

|  |
| --- |
| [root@localhost ~]# set -- "I am" handsome oldboy.  [root@localhost ~]# echo $#  3  [root@localhost ~]# **for i in "$\*";do echo $i;done**  I am handsome oldboy.  [root@localhost ~]# **for i in "$@";do echo $i;done # 双引号情况下,分别输出**  I am  handsome  oldboy.  [root@localhost ~]# for i in $\*;do echo $i;done # 去掉双引号,情况特殊,把所有参数输出  I  am  handsome  oldboy.  [root@localhost ~]# for i in $@;do echo $i;done # 去掉双引号,情况特殊,把所有参数输出  I  am  handsome  oldboy.  # **shift命令解释 ssh-copy-id脚本有用到这个命令**  [root@localhost ~]# for i in "$@";do echo $i;done  I am  handsome  oldboy.  [root@localhost ~]# shift  [root@localhost ~]# for i in "$@";do echo $i;done  handsome  oldboy.  [root@localhost ~]# shift  [root@localhost ~]# for i in "$@";do echo $i;done  oldboy. |

学习系统脚本:

/etc/init.d/rpcbind

#### 1.2.5.4 bash内部变量

bash命令解释套装程序包含了一些内部命令.有些内部命令在目录列表时是看不见的.他们由shell本身提供.常用内部的命令有: echo, eval, exec, export, readonly, read, shift, wait和点(.). 下面简单介绍其命令格式和功能.

(1) echo

echo -Display messge on screen

命令格式: echo args <== 可以是字符串和变量的组合

功能: 将echo命令后面的args指定的字符串及变量等显示出标准输出

(2)eval

命令格式: eval args

功能: 当shell程序执行到eval语句时,shell读入参数args,并将它们组合成一个新的命令.然后执行

(3)exec

命令格式: exec命令参数

工功能: 当shell执行exec语句时, 不会去创建新的子进程, 而是转去执行指定的命令, 当指定的命令执行完毕时, 该进程(也就是中的进程)就终止了.所以shell程序中exec后面的语句将不再执行.

(4)export

命令格式: export 变量名 或: export 变量名=变量值

功能: shell可以用export把它的变量向下带入子shell, 从而让子进程继承父进程中的环境变量. 但子shell不能用export把它的变量向上带入父shell

注意: 不带任何变量名的export语句将显示出当前所有的export变量

(5) readonly

readonly - Mark variables/functions as readonly

命令格式: readonly 变量名

功能: 将一个用户定义的sehll变量标识为不可变. 不带任何参数的readonly命令将显示出所有的只读的sehll变量

(6) read

命令格式: read 变量名表

功能: 从标准输入读字符串等信息, 传给shell程序内部定义的变量.

可以在函数中用local变量名的方式申明局部变量.

(7) shift语句

shift - Shift positional parameters

功能: shift语句按如下方式重新命令所有的位置参数变量.即$2称为$1,$3称为$2...,在程序中每使用一次shift语句,都使所有的位置参数依次向左移动一个位置, 并使位置参数$#减1, 知道减为0为止.

系统案例: ssh-copy-id -i /root/.ssh/id\_dsa.pub

(8) wait

功能: 使shell等待在后台启动的所有子进程结束. wait返回的值总是真.

(9) exit

功能: 退出shell程序.在exit之后可有选择的指定一个数位作为返回状态

(10) "."

命令格式: . shell程序文件名

**内部变量实践:**

#### 1.2.5.5 变量子串的常用操作(了解)

常用操作如下: man bash找资料

|  |  |
| --- | --- |
| ${#string} | 返回$string的长度 |
| ${string:position} | 在$string中,从位置$position之后开始提取子串 |
| ${string:position:length} | 在$string中,从位置$position之后开始提取长度为$length的子串 |
| ${string#substring} | 从变量$string开头开始删除最短匹配$substring子串 |
| ${string##subingstring} | 从变量$string开头开始删除最长配置$substring子串 |
| ${string%substring} | 从变量$string结尾开始删除最短匹配$substring子串 |
| ${string%%substring} | 从变量$string结尾开始删除最长匹配$substring子串 |
| ${string/substring/replace} | 使用$replace,来代替第一个匹配的$substring |
| ${string/#substring/replace} | 如果$string前缀匹配#substring,就用$replace来代替匹配$substring |
| ${string/%substring/replace} | 如果$string后缀匹配#substring,就用$replace来代替匹配$substring |

定义OLDBOY变量,内容为"I am oldboy"

|  |
| --- |
| [root@orac122 ~]# OLDBOY="I am oldboy"  echo ${OLDBOY}  1) 返回字符差OLDBOY变量字符串的长度  echo $OLDBOY|wc -L  ehco ${#OLDBOY}  2) 截取OLDBOY变量字符串从第2个字符之后开始取,默认去后面字符的全部  echo ${OLDBOY:2} am oldboy  3) 截取OLDBOY变量字符串从第2个字符之后开始取, 取两个字符.  echo ${OLDBOY:2:2} am  echo ${OLDBOY}|cut -c 1-4 I am  echo ${OLDBOY}|cut -c 3-4 am  4) 从变量$OLDBOY开头开始删除最短匹配"a\*C"子串 OLDBOY="abcABC123ABCabc"  echo ${OLDBOY#a\*C} 123ABCabc  5) 从变量$OLDBOY开头开始删除最长匹配" a\*C "子串  echo ${OLDBOY## a\*C } abc  6) 从变量$OLDBOY结尾开始删除最短匹配a\*C子串  echo ${OLDBOY% a\*C } abcABC123ABC  echo ${OLDBOY%%a\*c} 没有了  echo ${OLDBOY/abc/mmm} mmmABC123ABCabc  echo ${OLDBOY/#abc/mmm} mmmABC123ABCabc  echo ${OLDBOY/%abc/mmm} abcABC123ABCmmm |

生产场景用法实例:

1) 变量结尾删除生产实践

功能描述如下:

${string%string} 从变量$string结尾开始删除最短匹配

# cat file.log

stu\_102999\_1\_finished.jpg

stu\_102999\_2\_finished.jpg

stu\_102999\_3\_finished.jpg

stu\_102999\_4\_finished.jpg

stu\_102999\_5\_finished.jpg

stu\_102999\_6\_finished.jpg

# touch `cat files.log |tr "\n" " "` 创建多个文件名

f= stu\_102999\_1\_finished.jpg

# echo $f

stu\_102999\_1\_finished.jpg

# echo ${f%finish\*.jpg}

f= stu\_102999\_1\_

# mv $f `echo ${f%finish\*.jpg}`.jpg

# for f in `ls \*fin\*` ; do mv $f `echo ${f%finish\*.jpg}`.jpg; done

2) 变量结尾替换生产实践

${string/%substring/replace} 如果$string后缀匹配$substring,就用$replace来代替匹配$substring

问题: 把下面文件的大写扩展名边小写扩展名:

实际处理脚本

for file in `ls ./\*.htm`

do

#mv $file `echo $file|sed 's/HTML/html/g'`

mv $file `echo ${file/%HTML/htm}`

done

for f in `ls \*jpg` ; do mv $f ${f/%jpg/html};done

**变量替换表**: 了解,man bash可以查到资料

|  |  |
| --- | --- |
| 运算符号 | 替换 |
| ${value:-word} | 如果变量名存在且非null,则返回值变量的值.否则,返回word字符串.  用途: 如果变量未定义,则返回默认值  范例: ${value:-word},如果value未定义,则表达式的值为word |
| ${value:=word} | 如果变量名存在切非null,则返回变量值.否则,设置这个变量值为word,并返回其值.  用途: 如果变量未定义,则设置变量为默认值,并返回默认值  范例: ${value:=word},如果value未定义,则设置value值为word,返回表达式的值也为word. |
| ${value:?"not defined"} | 如果变量名存在且非null,则返回变量的值.否则显示变量名:mesage,并退出当前的命令或者脚本  用途: 用于捕捉由于变量未定义而导致的错误,并退出程序.  范例: ${value:?"not defined"},如果value未定义,则显示-bash: value: not define,并**退出** |
| ${value: +word} | 如果变量名存在且非null,则返回word.否则返回null.  用途: 测试变量是否存在.  范例: ${value:+word} 如果value已经定义,返回word(也就是真) |
| 更过内容,请执行man bash查看帮助. | |

每个运算符内的冒号都是可选的.如果省略冒号,则将每个定义中的  
"存在且非null"部分改为"存在",也就是说,运算符仅用于测试变量是否存在.

下面一次举例说明:

(1)${value:-word}

|  |
| --- |
| [root@localhost ~]# result=${test:-UNSET}  [root@localhost ~]# echo $result  UNSET  [root@localhost ~]# test='oldboy'  [root@localhost ~]# result=${test:-UNSET}  [root@localhost ~]# echo $result  oldboy |

(2)${value:=word}

提示: 这个变量功能可以解决变量没有定义的问题,确保变量始终有值

|  |
| --- |
| [root@localhost ~]# unset result  [root@localhost ~]# echo $result  [root@localhost ~]# unset test  [root@localhost ~]# echo $test  [root@localhost ~]# result=${test:=UNSET}  [root@localhost ~]# echo $result  UNSET |

(3)${test:?"word"}

捕捉由于变量未定义而导致的错误.

|  |
| --- |
| [root@localhost ~]# echo ${value:? "not defined"}  -bash: value: not defined  [root@localhost ~]# value=1  [root@localhost ~]# echo ${value:? "not defined"}  1  [root@localhost ~]# unset value  [root@localhost ~]# echo ${value:? "not defined"}  -bash: value: not defined |

(4)${value:+word}

此功能可用于测试变量是否存在

|  |
| --- |
| [root@localhost ~]# r=${value:+1}  [root@localhost ~]# echo $r  [root@localhost ~]# value=oldboy  [root@localhost ~]# r=${value:+1}  [root@localhost ~]# echo $r  1 |

(5)${value-word}去掉冒号

变量没有定义就用后面的替代

|  |
| --- |
| [root@localhost ~]# httpd=${HTTPD-/usr/bin/httpd}  [root@localhost ~]# pidfile=${PIDFILE-/var/run/httpd.pid}  [root@localhost ~]# echo $httpd $pidfile  /usr/bin/httpd /var/run/httpd.pid |

(6)实战例子/etc/init.d/httpd

|  |
| --- |
| 1. /etc/init.d/httpd  2. /etc/init.d/crond  3. 删除操作  [root@localhost ~]# cat del.sh  path=/server/backup  find ${path:=/tmp/} -name "\*.tar.gz" -type f | xargs rm -rf  + path=/server/backup  + find /server/backup -name '\*.tar.gz' -type f  find: `/server/backup': No such file or directory |

(7)变量的名字长度计算时间对比

|  |
| --- |
| [root@localhost ~]# chars=`seq -s " " 100`  [root@localhost ~]# echo **${#chars} ## 效率最高**  291  [root@localhost ~]# echo **$(expr length "$chars")**  291  [root@localhost ~]# echo **${chars}|wc -L**  291  [root@localhost ~]# time for i in $(seq 11111);do count=${#chars};done;  real 0m0.373s  user 0m0.372s  sys 0m0.000s  [root@localhost ~]# time for i in $(seq 11111);do count=`echo expr length "${chars}"`;done;  real 0m4.229s  user 0m0.270s  sys 0m0.859s  [root@localhost ~]# time for i in $(seq 11111);do count=`echo ${chars}|wc -L`;done;  real 0m14.599s  user 0m0.340s  sys 0m0.824s |

变量替换的一些参考资料

(1)man bash 变量处理大全

(2) http://www.cnblogs.com/chengmo/archive/2010/10/02/1841355.html

#### 1.2.5.6 变量的数值计算(小技巧)

变量的数值计算常见的又如下几个命令:

(()) let expr bc(小数) $[] 其他都是整数

(1) (())的用法(很常用)

如果要执行简单的整数运算,只需将特定的算数表达式用"$(())"包括起来就行.

shell的算数运算符号常置于"$((""))"的语法中.这一语法如同双引号功能,除了内嵌双引号无需转义.

|  |  |
| --- | --- |
| ++ -- | 加加 减减 |
| + - ! ~ |  |
| \* / % | 乘 除法 取余 |
| + - |  |
| < <= > >= | 比较符号 |
| == != | 等等 不等,一个等号为赋值 |
| << >> | 左位移 右位移 |
| & | 位的AND |
| ^ | 位的异或 |
| | | 位的或 |
| && | 逻辑的AND(make && make install) |
| || | 逻辑的OR |
| ?: | 条件表达式 |
| = += -= \*= /= %= &= ^= <<= >>= |= |  |

范例1: shell的算术运算实例如下:

|  |
| --- |
| [root@localhost ~]# ((a=1+2\*\*3-4%3))  [root@localhost ~]# echo $a  8  [root@localhost ~]# b=$((1+2\*\*3-4%3))  [root@localhost ~]# echo $b  8  [root@localhost ~]# echo $((1+2\*\*3-4%3))  8 |

范例2: 定义变量计算

|  |
| --- |
| [root@localhost ~]# myvar=99  [root@localhost ~]# echo $(($myvar + 1))  100 |

范例3: 各种(())运算的例子

|  |
| --- |
| [root@localhost ~]# cat test.sh  #!/bin/bash  a=6  b=7  echo "a-b =$(( $a - $b))"  echo "a+b =$(( $a + $b))"  echo "a\*b =$(( $a \* $b))"  echo "a/b =$(( $a / $b))"  echo "a\*\*b =$(( $a \*\* $b))"`  ## 传参实现  [root@localhost ~]# cat test.sh  #!/bin/bash  a=$1  b=$2  echo "a-b =$(( $a - $b))"  echo "a+b =$(( $a + $b))"  echo "a\*b =$(( $a \* $b))"  echo "a/b =$(( $a / $b))"  echo "a\*\*b =$(( $a \*\* $b))"` |

范例4: 实现一个加减乘除功能的计算器.

|  |
| --- |
| [root@localhost ~]# cat bc.sh  #!/bin/bash  echo $(($1$2$3)) # 分开传参,容错比较好  [root@localhost ~]# sh bc.sh 10\*2  20  [root@localhost ~]# sh bc.sh 23+12  35  [root@localhost ~]# sh bc.sh 7/3  2  **升华版**  [root@localhost ~]# sh bc.sh 7/3  2  [root@localhost ~]# sh bc.sh 23+12  35  [root@localhost ~]# sh bc.sh 23 + 12  23  高级版  [root@localhost ~]# cat sp.sh  #!/bin/bash  print\_usage(){  printf "please enter an integer\n"  exit 1  }  read -p "Please input first number: " firstnum  if [ -n "`echo $firstnum|sed 's/[0-9]//g'|sed 's/-//g'`" ];then  print\_usage  fi  read -p "Please input the operators: " operators  if [ "${operators}" != "+" ]&&[ "${operators}" != "-" ]&&[ "${operators}" != "\*" ]&&[ "${operators}" != "/" ];then  echo "please use +-\*/"  exit 1  fi  read -p "Please input the second number: " secondnum  if [ -n "`echo $secondnum|sed 's/[0-9]//g'|sed 's/-//g'`" ];then  print\_usage  fi  echo "${firstnum}${operators}${secondnum}=$((${firstnum}${operators}${secondnum}))"  [root@localhost ~]# sh sp.sh  Please input first number: 23  Please input the operators: \*  Please input the second number: 12  23\*12=276 |

(2) let的用法

格式: let 赋值表达式

注意: let赋值表达式功能等同于: ((赋值表达式))

范例1: 给自变量i加8

|  |
| --- |
| [root@localhost ~]# i=2  [root@localhost ~]# let i=i+8  [root@localhost ~]# echo $i  10  [root@localhost ~]# i=i+8  [root@localhost ~]# echo $i  i+8 |

**范例2**: **利用let计算监控web服务状态(守护进程)**

|  |
| --- |
| ServerMonitor(){  timeout=10  fails=0  success=0  while true  do  /usr/bin/wget --timeout=$timeout --tries=1 http://192.168.20.84/ -q -0  0 /dev/null  if [ $? -ne 0 ]  then  let fails=fails+1  success=0  else  fails=0  let success=1  fi  if [ $success -ge 1 ]  then  exit 0  fi  if [ $fails -ge 2 ]  then  Critical="TMS应用服务出现故障,请紧急处理"  echo $Critical | mutt -s "服务down" pizh@yonyou.com  exit 1  fi  done  } |

(2) expr的用法

语法: expr Expression

expr命令一般用于整数值,但也可用于字符串,用来求表达式变量的值,同时expr是一个手工命令行计算器

|  |
| --- |
| [root@localhost ~]# expr 2 + 2  4  [root@localhost ~]# expr 3 - 2  1  [root@localhost ~]# expr 2-1  2-1  [root@localhost ~]# expr 2 -1  expr: syntax error  [root@localhost ~]# expr 2 - 1  1  [root@localhost ~]# expr 2 \\* 2 #乘号需要转义  4 |

注意:

1. 运算符左右都有空格

2. 使用乘号时,必须用反斜线屏蔽其特定含义.

**增量计数**

expr在循环中可用于增量计算.首先,循环初始化为0, 然后循环值加1,反引号的用法为命令替代.最基本的一种是从expr命令接受输出并将之放入循环变量

如: 给自变量i加1

|  |
| --- |
| [root@localhost ~]# i=0  [root@localhost ~]# i=`expr $i + 1`  [root@localhost ~]# echo $i  1 |

**expr [$a+$b]**表达式形式,其中$a$b可为整数值.

|  |
| --- |
| [root@localhost ~]# expr $[2+3]  5  [root@localhost ~]# expr $[2\*\*3]  8 |

其他特殊用法:

expr用法: ssh-copy-id脚本

if expr "$1" : ".\*\.pub" ; then

expr id\_dsa.pub : '.\*\.pub', 匹配\*.pub格式的文件如果是则为真.

expr "id\_dsa.pub" : ".\*\.pub"

**技巧: 判断文字或字符串的扩展名小案例**

|  |
| --- |
| [root@localhost ~]# expr "test.pub" : ".\*\.pub" && echo 1 || echo 0  8  1  [root@localhost ~]# expr "test.x" : ".\*\.pub" && echo 1 || echo 0  0  0  [root@localhost ~]# cat expr1.sh  #!/bin/bash  if expr "$1" : ".\*\.pub" &>/dev/null  then  echo "you are using $1"  else  echo "Please use \*.pub file"  fi |

**技巧: 判断一个变量是否为整数**

|  |
| --- |
| [root@localhost ~]# cat judge\_int.sh  #!/bin/bash  expr 1 + $1 &>/dev/null  if [ $? -eq 0 ]  then  echo "shi zheng shu"  else  echo "NONONO"  fi  # 加强版  [root@localhost ~]# cat judge\_int.sh  #!/bin/bash  while true  do  read -p "pls input: " a  expr 1 + $a &>/dev/null  [ $? -eq 0 ] && echo int || echo chars  done |

**技巧:** 通过expr计算字符的长度

|  |
| --- |
| [root@localhost ~]# chars=`seq -s " " 100`  [root@localhost ~]# echo ${#chars}  291  [root@localhost ~]# echo $(expr length "$chars")  291 |

(4) **bc命令**的用法

特点: bc的独有特点是支持小数计算, 当然也可以支持整数计算

bc是UNIX下的计算器,它可以用在命令行下面:

例: 给自变量i加1

i=2

i=`echo $i+1|bc` ---效率低

\*\* 因为bc支持科学计算, 所以这种方法功能非常强大

|  |
| --- |
| *## 可直接命令行进入,直接计算*  [root@localhost ~]# bc  bc 1.06.95  Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.  This is free software with ABSOLUTELY NO WARRANTY.  For details type `warranty'.  2+3  5  3\*\*3  (standard\_in) 2: syntax error  3\*3  9  5/2  2  5%2  1  quit  *## 一般不如上这样做,一般用下面的管道计算*  [root@localhost ~]# echo 1+1|bc  2  [root@localhost ~]# echo 3.5+5|bc #计算小数特有  8.5  [root@localhost ~]# echo 6-1.5|bc  4.5  [root@localhost ~]# echo "scale=1;5.23 \* 3.13"|bc  16.36  [root@localhost ~]# echo "obase=2;8"|bc #十进制的8转转换为二进制的  1000 |

**范例:** **通过命令输出1+2+3...+10的表达式,并计算出结果**

**1+2+3+4+5+6+7+8+9+10=**

|  |
| --- |
| [root@localhost ~]# echo `seq -s '+' 10`=`seq -s "+" 10|bc` ## 法1  1+2+3+4+5+6+7+8+9+10=55  [root@localhost ~]# echo `seq -s '+' 10`=$((`seq -s "+" 10`)) ## 法2  1+2+3+4+5+6+7+8+9+10=55 |

(5) **typeset**用法总结

使用整数变量直接计算

如:

|  |
| --- |
| [root@localhost ~]# typeset -i A=1 B=3  [root@localhost ~]# A=A+B  [root@localhost ~]# echo $A  4 |

(5)**$[]**的用法

如: 容错性比较好,expr用于判断整数

|  |
| --- |
| [root@localhost ~]# echo $[2+3]  5  [root@localhost ~]# echo $[2\*3]  6  [root@localhost ~]# echo $[ 2 + 3 ]  5 |

实际解决问题例子: 打印数学杨辉三角

http://oldboy.blog.51cto.com/2561410/756234

#### 1.2.5.7 shell变量的输入(read)

shell变量除了可以直接赋值或脚本传参外,还可以使用read命令从标准输入获得.read为内置命令,通过help read查看帮助.

语法格式: **read [参数] [变量名]**

常用参数:

-p promt: 设置提示信息

-t timeout: 设置输入等待的时间, 单位默认为秒

**范例1: read的基本读入**

|  |
| --- |
| [root@localhost ~]# **read -p "please input two number:" a1 a2** # 中间要有空格  please input two number:23 45  [root@localhost ~]# echo $a1 # 变量赋值给a1  23  [root@localhost ~]# echo $a2  45  [root@localhost ~]# **read -t 5 -p "please input:" a** # -t 5 5s后没有输入,退出  please input: |

提示: read -p的功能可以用echo和read实现

|  |
| --- |
| [root@localhost ~]# **echo -n "please input two number:"; read a1 a2**  please input two number:23 45  [root@localhost ~]# echo $a1  23 |

**范例2: 实践题**

**将前面的加减乘除计算的脚本改成通过read方式读入整数变量:**

|  |
| --- |
| ## 原始脚本  [root@localhost ~]# cat test.sh  #!/bin/bash  a=6  b=7  echo "a-b =$(( $a - $b))"  echo "a+b =$(( $a + $b))"  echo "a\*b =$(( $a \* $b))"  echo "a/b =$(( $a / $b))"  echo "a\*\*b =$(( $a \*\* $b))"  ## 修改后  [root@localhost ~]# cat test.sh  #!/bin/bash  echo -n "please input two numbers:"  read a b  echo "a-b =$(( $a - $b))"  echo "a+b =$(( $a + $b))"  echo "a\*b =$(( $a \* $b))"  echo "a/b =$(( $a / $b))"  echo "a\*\*b =$(( $a \*\* $b))" |

**范例4: 如果读入(read)的不是整数, 或者参数个数不是2个,以上shell脚本的执行会有什么结果? 如何解决?**

|  |
| --- |
| #!/bin/bash  read -p "Please input two numbers:" a b  expr 1 + $a &>/dev/null  A=$?  expr 1 + $b &>/dev/null  B=$?  if [ ! -n "$a" ] || [ ! -n "$b" ];then  echo "Please insert two values"  exit 1  elif [ "$A" -ne 0 ] || [ "$B" -ne 0 ];then  echo "Please input two int"  exit 2  else  echo "$a-$b=$(($a-$b))"  exit 0  fi  #传参方法  [ $# -ne 2 ] && {  echo "USAGE:$0 num1 num2"  exit 1  }  expr $1 + $2 &>/dev/null  if [ $? -ne 0 ];then  echo "error:Please input two numbers"  else  echo "$1-$2=$(($1-$2))"  fi |

## 综合例子:

开发shell脚本分别实现以**定义变量**,**脚本传参**以及**read读入**的方式比较2个整数大小.用条件表达式(禁止if)进行判断并以屏幕输出方式提醒用户比较结果.注意: 一共是开发3个脚本.当用脚本传参以及read读入的方式需要对变量是否为数字做判断.

解答:

|  |
| --- |
| #!/bin/bash  read -p "Pls input two num:" a b  [ -z "$a" ] || [ -z "$b" ] && {  echo "Pls input two num again."  exit 1  }  expr $a + 0 &>/dev/null  RETVAL1=$?  expr $b + 0 &>/dev/null  RETVAL2=$?  test $RETVAL1 -eq 0 -a $RETVAL2 -eq 0 || {  echo "Pls input two "two" numer"  exit 2  }  [ $a -lt $b ] && {  echo "$a < $b"  exit 0  }  [ $a -eq $b ] && {  echo "$a = $b"  exit 0  }  [ $a -gt $b ] && {  echo "$a > $b"  exit 0  } |

**例子:** 利用read和cat打印菜单,**lic\_menu.sh**

|  |
| --- |
|  |

# 1.3 条件测试

## 1.3.1 测试语句

1.3.1.1 条件测试语法

在bash的各种流程控制结构中通常要进行各种测试,然后根据测试结果执行不容的操作,有时候也会通过与if等条件语句相结合,让我们可以方便的完成判断.

格式1: test <测试表达式>

格式2: [ <测试表达式> ]

格式3: [[ <测试表达式> ]]

说明: 格式1和格式2是等价的.格式3是扩展的test命令.

**在[[]]中可以使用通配符进行模式匹配.&&,||,>,<等操作符可以用与[[]]中.**

**可以 man test查看帮助**

* **Test判断**

范例1: test 测试文件

|  |
| --- |
| [root@localhost ~]# test -f file && echo false || echo true  true  [root@localhost ~]# touch file  [root@localhost ~]# test -f file && echo false || echo true  false |

范例2: test ! 语法

|  |
| --- |
| [root@localhost ~]# test ! -f file && echo false || echo true  true |

范例3: test 字符串

|  |
| --- |
| [root@localhost ~]# test -n "$arg" && echo 1 || echo 0  0  [root@localhost ~]# test -z "$arg" && echo 1 || echo 0  1 |

* [] 判断.注意: 括号里面两边必须有空格

范例:

|  |
| --- |
| [root@localhost ~]# rm -rf file  [root@localhost ~]# [ -f file ] && echo 1 || echo 0  0  [root@localhost ~]# touch file  [root@localhost ~]# [ -f file ] && echo 1 || echo 0  1  [root@localhost ~]# [ ! -f file ] && echo 1 || echo 0  0  [root@localhost ~]# [ -f file ] && echo 1  1  [root@localhost ~]# [ -f file ] || echo 0 |

* [[]]用法

范例:

|  |
| --- |
| [root@localhost ~]# [[ -f file ]] && echo 1 || echo 0  1  [root@localhost ~]# rm -rf file  [root@localhost ~]# [[ -f file ]] && echo 1 || echo 0  0  [root@localhost ~]# [[ -f file && -d dir ]] && echo 1 || echo 0  0  [root@localhost ~]# mkdir dir  [root@localhost ~]# [[ -f file && -d dir ]] && echo 1 || echo 0  0  [root@localhost ~]# touch file  [root@localhost ~]# [[ -f file && -d dir ]] && echo 1 || echo 0  1 |

1.3.1.2 **文件**测试操作符

在书写测试表达式时,可以下表中的文件测试操作符.

|  |  |
| --- | --- |
| 常用文件测试操作符 | 说明 |
| -f 文件 file | 若文件存在且为普通文件则真 |
| -d 文件 directory | 若文件存在且为目录文件则真 |
| -s 文件 size | 若文件存在且不为空(文件大小非0)则真 |
| -e 文件 exist | 若文件存在且是普通文件则真,要区别-f |
| -r 文件 read | 若文件存在且可读则真 |
| -w 文件 write | 若文件存在且可写则真 |
| -x 文件 execute | 若文件存在且可执行则真 |
| -L文件 link | 若文件存在且为链接文件则真 |
| f1 -nt f2 newer than | 若文件f1比文件f2新则真 |
| f1 -ot f2 older than | 若文件f1比文件f2旧则真 |

演示:

|  |
| --- |
| [root@localhost ~]# [ -e file ] && echo 1 || echo 0  1  [root@localhost ~]# [ -f file ] && echo 1 || echo 0  1  [root@localhost ~]# mkdir dir  mkdir: cannot create directory `dir': File exists  [root@localhost ~]# [ -f dir ] && echo 1 || echo 0  0  [root@localhost ~]# [ -d dir ] && echo 1 || echo 0  1  [root@localhost ~]# [ -e dir ] && echo 1 || echo 0  1  [root@localhost ~]# [ -r dir ] && echo 1 || echo 0 必须是普通用户才能判断r,w,x  1  [root@localhost ~]# [ -w dir ] && echo 1 || echo 0  1 |

1.3.1.3 **字符串**测试操作符

字符串测试操作符的作用: 比较两个字符串是否相同,字符串长度是否为零,字符串是否为null(注: bash区分零长度字符串和空字符串)等.

"="比较两个字符串是否相同,与**==**等价.如if[ "$a" = "$b" ],其中$a这样的变量最好用""括起来,因为如果中间有空格,\*等符号就可能出错了,当然更好的方法是:[ "${a}"="${b}" ]."!="比较两个字符串是否相同,不同则为"是"

在书写测试表达式时,可以使用下表中的字符串测试操作符.

|  |  |
| --- | --- |
| 常用字符串测试操作符 | 说明 |
| -z "字符串" | 若串长度为0则真,-z可以理解为zero |
| -n "字符串" | 若串长度不为0则真,-n可以理解为nonezero |
| "串1" = "串2" | 若串1等于串2则真,可使用"=="代替"=" |
| "串1" != "串2" | 若串1不等于串2则真.但不能用"!=="代替"!=" |
| **特别注意:**  **1.字符串测试操作符号务必要用""引起来**  **2.字符串或字符串变量比较,等号两端最好都有空格,多参考系统脚本** | |

例子演示:

|  |
| --- |
| [root@localhost ~]# [ -n "abc" ] && echo 1 || echo 0  1  [root@localhost ~]# test="hello"  [root@localhost ~]# [ -n "$test" ] && echo 1 || echo 0  1  [root@localhost ~]# test=""  [root@localhost ~]# [ -n "$test" ] && echo 1 || echo 0  0  [root@localhost ~]# [ -z "$test" ] && echo 1 || echo 0  1  [root@localhost ~]# [ "abc" = "abd" ] && echo 1 || echo 0  0  [root@localhost ~]# [ "abc" == "abd" ] && echo 1 || echo 0  0  [root@localhost ~]# [ "abc" == "abc" ] && echo 1 || echo 0  1  [root@localhost ~]# [ "$test" == "" ] && echo 1 || echo 0  1  [root@localhost ~]# test="hello"  [root@localhost ~]# [ "$test" == "" ] && echo 1 || echo 0  0  [root@localhost ~]# [ "$test" == "hello" ] && echo 1 || echo 0  1 |

1.3.1.4 整数二元比较操作符

在书写测试表达式时,可以使用下表中的整数二元比较操作符

|  |  |  |
| --- | --- | --- |
| 在[]中使用的比较符 | 在(())和[[]]中使用的比较符 | 说明 |
| -eq | == | equal的缩写,相等 |
| -ne | != | not equal,不相等 |
| -gt | > | geater than 大于 |
| -ge | >= | geater equal 大于等于 |
| -lt | < | less than 小于 |
| -le | <= | less equal 小于等于 |

提示:

1). "<"符号意思是小于.if[[ "$a" < "$b" ]],if [ "$a" \< "$b" ].在单[]中需要转义,因为shell也用<和>重定向.对于">"符合同样适用

2)."="符号意思是等于,if[[ "$a" = "$b" ]] if[ "$a" = "$b" ]在单[]中不需要转义

特别提示:

经过实践,"="和"!="在[]中使用不需要转义,包含"<"和">"的符号在[]中使用需要转义,对于数字不转义的结果未必会报错,但是结果可能会不对

范例1: 二元数字比较

|  |
| --- |
| [root@localhost ~]# [ 2 < 1 ] && echo 1 || echo 0  -bash: 1: No such file or directory  0  [root@localhost ~]# [ 2 lt 1 ] && echo 1 || echo 0  -bash: [: lt: binary operator expected  0  [root@localhost ~]# [ 12 -eq 13 ] && echo 1 || echo 0  0  [root@localhost ~]# [ 12 -gt 13 ] && echo 1 || echo 0  0  [root@localhost ~]# [ 12 -lt 13 ] && echo 1 || echo 0  1  [root@localhost ~]# [ 12 < 13 ] && echo 1 || echo 0  -bash: 13: No such file or directory  0  [root@localhost ~]# [[ 12 < 13 ]] && echo 1 || echo 0  1  [root@localhost ~]# [ 12 \< 13 ] && echo 1 || echo 0  1  [root@localhost ~]# [ 12 = 13 ] && echo 1 || echo 0  0  [root@localhost ~]# [ 12 = 12 ] && echo 1 || echo 0  1  [root@localhost ~]# (( 12 = 12 )) && echo 1 || echo 0  -bash: ((: 12 = 12 : attempted assignment to non-variable (error token is "= 12 ")  0  [root@localhost ~]# (( 12 == 12 )) && echo 1 || echo 0  1 |

1.3.1.5 逻辑操作符

在书写测试表达式时,可以使用下表中的逻辑操作符实现复杂的条件测试

|  |  |  |
| --- | --- | --- |
| 在[]中使用的逻辑操作符 | 在[[]]中使用的逻辑操作符 | 说明 |
| -a | && | and 与,两端都为真,则真 |
| -o | || | or 或,有真则为真 |
| ! | ! | not 非,相反则为真 |

演示:

|  |
| --- |
| [root@localhost ~]# ll /etc/rc.local  lrwxrwxrwx. 1 root root 13 Aug 2 05:50 /etc/rc.local -> rc.d/rc.local  [root@localhost ~]# ll /etc/services  -rw-r--r--. 1 root root 641020 Mar 22 05:56 /etc/services  [root@localhost ~]# f1=/etc/rc.local  [root@localhost ~]# f2=/etc/services  [root@localhost ~]# [ -f "$f1" -a -f "$f2" ] && echo 1 || echo 0  1  [root@localhost ~]# [ -f "$f1" && -f "$f2" ] && echo 1 || echo 0  -bash: [: missing `]'  0  [root@localhost ~]# [[ -f "$f1" && -f "$f2" ]] && echo 1 || echo 0  1  [root@localhost ~]#  [root@localhost ~]#  [root@localhost ~]# [ -n "$f1" -a -z "$f2" ] && echo 1 || echo 0  0  [root@localhost ~]# [ -n "$f1" -o "$f1" = "$f2" ] && echo 1 || echo 0  1  [root@localhost ~]# [ -n "$f1" -a "$f1" = "$f2" ] && echo 1 || echo 0  0  [root@localhost ~]# echo $f1  /etc/rc.local  [root@localhost ~]# echo $f2  /etc/services  [root@localhost ~]# [ -n "$f1" -a "${#f1}" = "${#f2}" ] && echo 1 || echo 0  1  [root@localhost ~]# n1=12  [root@localhost ~]# n2=13  [root@localhost ~]# [[ $n1 -eq $n2 && -z "$n1" ]] && echo 1 || echo 0  0  [root@localhost ~]# [[ $n1 = $n2 && -z "$n1" ]] && echo 1 || echo 0  0  [root@localhost ~]# [[ ! $n1 = $n2 && ! -z "$n1" ]] && echo 1 || echo 0  1 |

## 1.3.2 条件测试举例

下面举一些条件测试的例子.为了清晰的显示测试结果,我们在屏幕上输出1(真)或0(假)表示测试结果.

系统中的脚本

1.3.2.4 test测试用例

|  |
| --- |
| [root@localhost ~]# echo $f1  /etc/rc.local  [root@localhost ~]# echo $f2  /etc/services  [root@localhost ~]# test -f $f1 && echo 1 || echo 0  1  [root@localhost ~]# test -n "$f1" && echo 1 || echo 0  1  [root@localhost ~]# test -z "$f1" && echo 1 || echo 0  0  [root@localhost ~]# test $f1 > $f2 && echo 1 || echo 0  1  [root@localhost ~]# echo $n1  12  [root@localhost ~]# echo $n2  13  [root@localhost ~]# test $n1 -gt $n2 && echo 1 || echo 0  0  [root@localhost ~]# test $n1 -eq $n2 -a -z "$2" && echo 1 || echo 0  0  [root@localhost ~]# touch a  [root@localhost ~]# touch b  [root@localhost ~]# test a -ot b && echo 1 || echo 0  1  [root@localhost ~]# test a -nt b && echo 1 || echo 0  0  [root@localhost ~]# test a -ef b && echo 1 || echo 0  0 |

# 1.4 分支与循环结构

## 1.4.1 if条件语句

1.4.1.1 if条件语句语法

* **单分支结构**

语法:

|  |
| --- |
| if[条件]  then  指令  fi  或  if[条件];then  指令  fi  特殊写法: if[ -f "$file1" ];then echo 1;fi 相当于: [ -f "file1" ] && echo 1 |

* **双分支结构**

语法:

|  |
| --- |
| if[条件]  then  指令  else  指令集  fi  [ -f "$file1" ] && echo 1 || echo0 |

* **多分支结构**

|  |
| --- |
| if 条件  then  指令  elif 条件  then  指令  else  指令  fi |

1.4.1.2 if条件举例

范例1: 但分支if条件比较两个整数的大小

|  |
| --- |
| #!/bin/bash  read -p "Please input two number:" a b  if [ $a -lt $b ];then  echo "$a < $b"  exit  fi  if [ $a -eq $b ];then  echo "$a = $b"  exit  fi  if [ $a -gt $b ];then  echo "$a > $b"  exit  fi  缺点: 重复,可以用多分支if实现 |

范例2: 开发shell脚本实现如果/server/scripts下面,如果存在就把if3.sh脚本的内容输出到屏幕,如果不存在就自动创建

|  |
| --- |
| #!/bin/bash  PATH=/server/scripts  FILE=if3.sh  if [ ! -d $PATH ];then  #mkdir -p $PATH  echo "$PATH is not exist,alreay create."  fi  if [ ! -f $PATH/$FILE ];then  touch $PATH/$FILE  echo "$FILE is not exist,already create."  exit  if  #ls -l $PATH/$FILE  echo "end....." |

范例3: 开发脚本判断系统剩余内存大小,如果低于100M就邮件报警.测试报警成功后加入系统时任务每3分钟执行一次检查

|  |
| --- |
| 1.如何取内存  [root@localhost ~]# **free -m|grep buffers/|awk '{print $NF}'**  830  [root@localhost ~]# **free -m|awk 'NR==3 {print $NF}'**  830  2.发邮件mail,mutt.sendmail服务要开启  /etc/init.d/sendmail start  3.正式脚本内容  #!/bin/bash  used\_mem=`free -m|awk 'NR==3 {print $NF}'`  if [ $used\_mem -lt 900 ];then  echo "mem is not enough,$used\_mem."  #echo "mem is not enough,$used\_mem."|mail -s "mem warning $(date +%F)" pizhihui@133.com  fi |

范例4: 双分支if条件语句命令行传参方式比较两个整数,前面范例1的升级

|  |
| --- |
| #!/bin/bash  a=$1  b=$2  if [ $a -lt $b ];then  echo "a < b"  exit 0  elif [ $a -gt $b ];then  echo "a > b"  exit 0  else  echo "a = b"  fi  exit 0 |

**扩展:判断字符串是否为数字的多种方法**

法一: sed加正则表达式

命令行传参:

组合语法判断1:

[ -n "`echo $num|sed 's/[0-9]//g'` -a -n "`echo $2|sed "s/[0-9]//g"`" ] && echo两个都是数字 && exit 1

单个判断语法:

[ -n "`echo $1|sed 's/[0-9]//g'`" ] && echo "必须为数字"

发二: 变量的子串替换加正则表达式

[ -z "`echo "${num//[0-9]/}"`" ] && echo 1 || echo 0

法三: 变量的子串替换加正则表达式(特殊判断思路)

[ -n "$num" -a "$num" = "${num//[^0-9]/}" ] && echo "it is num."

法四: expr计算判断

expr $1 + 0 >/dev/null 2>&1

[ $? -ne 0 ] && echo int

法五: 利用"=~"符号判断man bash帮助里[[]] 里介绍了

[[ $1 =~[0-9] ]] && echo int || echo char

法六:bc判断(有点小问题)

**范例5: 生产环境监控MySQL服务的实战例子**

问题描述: 监控MySQL服务是正常启动,如果未正常启动,就启动MySQL

|  |
| --- |
| 1.判断mysql服务是否正常  ①. netstat -nltp|grep 3306  ②. ps -ef|grep mysql  ③. mysql -uroot -proot -e "select version();" echo $?  ④. echo "PID,socket" 通过pid和socket文件判断 |

法一: 端口监控

|  |
| --- |
| #!/bin/bash  #!/bin/bash  port=`netstat -lnt|grep 3306 |awk -F '[: ]+' '{print $5}'`  if [ "$port" != "3306" ];then 推荐用字符串比较  ~~if [ $port -ne 3306 ];then 如果没有启动mysql服务,整数判断是错误的~~  /data/3306/mysql start  fi  #!/bin/bash  PORT=`netstat -lntup|grep 3306|wc -l`  if[ $PORT -ne 1 ];then  echo "Starting mysql....."  else  echo "MySQL is running"  fi |

法二: 利用进程

|  |
| --- |
| ps -ef|grep mysql|grep -v grep|wc -l  问题: ps自身也会有一个进程显示在本命令的结果中,所以grep -v过滤掉自己  #!/bin/bash  PROCESS=`ps -ef|grep mysql|grep -v grep|wc -l`  if[ $PROCESS -ne 2 ];then  echo "Starting mysql....."  else  echo "MySQL is running"  fi |

法三: 连接看返回值

|  |
| --- |
| mysql -uroot -proot -e "select version();" &>/dev/null  #!/bin/bash  mysql -uroot -proot -e "select version();" &>/dev/null  if[ $? -ne 0 ];then  echo "Starting mysql....."  else  echo "MySQL is running"  fi |

法四: 更专业的监控脚本

通过PIDFILE文件来判断是否正常启动

|  |
| --- |
| #!/bin/bash  PIDFILE="/application/mysql/data/mysql.pid"  MYSQL\_PATY="/application/mysql/bin"  DATADIR="/application/mysql/data"  if [ ! -f $PIDFILE ];then  /bin/sh $MYSQL\_PATH/mysqld-safe --datadir=$DATADIR --pid-file=$PIDFILE  touch /var/lock/subways/mysql  else  echo "MySQL is running"  fi |

回顾监控MySQL服务是否异常的多种方法:

1.根据MySQL端口号监控Mysql(本地)

此处是本地监控,端口在,服务可能不正常.如: 负载很高,CPU很高,连接数满了

本地:

netstat -lntup|grep 3306

lsof -i:3306|wc -l

远程: http://oldboy.blog.51cto.com/2561410/942530

telnet/nc/nmap

2.根据MySQL进程监控MySQL(本地)

只能本地监控,远程在服务可能不正常,如: 负载很高,CPU很高

ps -ef|grep mysql|grep -v grep|wc -l

注意: 脚本名字里不能含有过滤的字符串mysql,否则不准.

3.通过MySQL客户端命令及用户账户连接MySQL,然后根据返回命令的状态确定是否mysql正常(本地和远程)

4.通过java/php程序url方式监控mysql

**小结: web服务监控手段**

1.端口

本地: ss,netstat,lsof

远程: telnet,nmap,nc

nmap 10.10.3.120 -p 80|grep open|wc -l 等于1

ps -ef|grep nginx|wc -l

curl -I -s 10.10.3.120|head -l

curl -I 10.10.3.120>/dev/null|head -l

curl -o /dev/null -I -s -w "${http\_code}" 10.10.4.230 等于200

wget --spider --timeout=5 10.10.4.230 &>/dev/null echo $?

lsof -i : 80|wc -l 大户等于1

ps -ef|grep nginx|wc -l 大户等于2

http://oldboy.blog.51cto.com/2561410/942530

http://oldboy.blog.51cto.com/2561410/1196298

2.本地进程数

3.header(http code) curl -I 返回200就ok

4.URL(wget,curl),模拟用户的方式

5.php,java写监控的程序, 模拟用户的方式(开发提供)

6.mysql登录

lsof -i:80|wc -l (大于等于1)

nmap 10.0.0.7 -p 80|grep open|wc -l (等于1)

ps -ef|grep nginx|wc -l (大于等于2)

curl -I -s -w "%{http\_code}" -o /dev/null 10.0.0.7 (等于200)

wget --spider --timeout=10 --tries=2 10.0.0.7 &>/dev/null

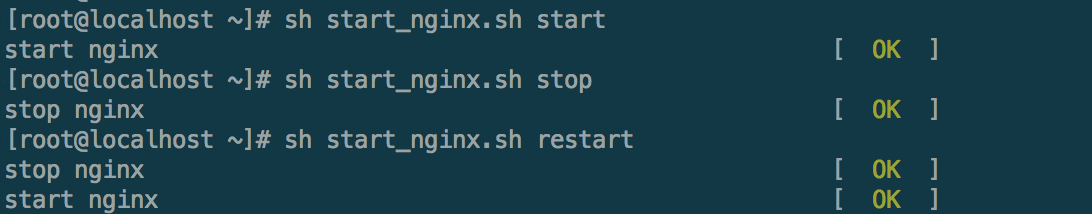
echo $?

0

|  |
| --- |
| [root@localhost ~]# cat check\_web.sh  #!/bin/bash  http\_code=`curl -I -s -w "%{http\_code}" -o /dev/null 10.0.0.7`  if [ $http\_code -ne 200 ];then  echo "web is error"  else  echo "web is ok."  if |

**范例: 利用系统函数模拟实现脚本启动的特殊颜色效果**

/start\_nginx.sh {start|stop|restart},用if语句实现.如下效果.



|  |
| --- |
| [root@localhost ~]# cat start\_nginx.sh  #!/bin/bash  . /etc/init.d/functions  if [ $# -ne 1 ];then  echo "USAGE: $0 {start|stop|restart}"  exit 1;  fi  if [ "$1" == "start" ];then  action "start nginx" /bin/true  elif [ "$1" == "stop" ];then  action "stop nginx" /bin/true  elif [ "$1" == "restart" ];then  action "stop nginx" /bin/true  action "start nginx" /bin/true  else  echo "USAGE: $0 {start|stop|restart}"  exit 1  fi |

**实战题: 监控web站点目录下所有文件是否被恶意篡改(文件内容被改了),如果有就打印改动的文件(发邮件),定时任务每3分钟执行一次.**

特征:a.大小可能变化

b.修改时间可能变化(ot,nt)

c.文件内容变化,md5指纹识别.

d.增加或则删除文件.

法一: 根据文件内容长度比较

|  |
| --- |
|  |

法二:

|  |
| --- |
|  |

## 1.4.2 case结构条件句

14.2.1 case结构条件语法

case "字符串变量" in

值1) 指令1....

;;

值2) 指令2...

;;

\*)指令

esac

1.4.2.2. case结构条件范例

**范例1: 根据用户的选择输入判断是哪种水果并加上不同颜色.**

执行脚本打印一个水果菜单如下:

1.apple

2.pear

3.banana

4.cherry

当用户选择水果的时候,打印告诉他选择的水果是什么,并给水果单词加上一种颜色,要求用case语句实现

|  |
| --- |
| [root@localhost ~]# cat fruit\_menu.sh  #!/bin/bash  RED\_COLOR='\E[1;31m'  GREEN\_COLOR='\E[1;32m'  YELLOW\_COLOR='\E[1;33m'  BLUE\_COLOR='\E[1;34m'  RES='\E[0m'  menu() {  cat <<END  1.apple  2.pear  3.banana  3.cherry  END  }  menu  read -p "Please input your choice:" fruit  case "$fruit" in  1)  echo -e "$RED\_COLOR apple $RES"  ;;  2)  echo -e "$GREEN\_COLOR pear $RES"  ;;  3)  echo -e "$YELLOW\_COLOR banana $RES"  ;;  \*)  echo "no fruit,you choose ."  esac |

**拓展: 让echo输出字符串显示不同颜色案例**

**范例1: 直接加颜色**

|  |
| --- |
| 1.字颜色范围: 30-37  echo -e "\033[30m 黑色字 hello trainning \033[0m"  echo -e "\033[31m 红色字 hello trainning \033[0m"  echo -e "\033[32m 绿色字 hello trainning \033[0m"  echo -e "\033[33m 黄色字 hello trainning \033[0m"  echo -e "\033[34m 蓝色字 hello trainning \033[0m"  echo -e "\033[35m 紫色字 hello trainning \033[0m"  echo -e "\033[36m 天蓝字 hello trainning \033[0m"  echo -e "\033[37m 白色字 hello trainning \033[0m" |

**范例2: 通过变量方式给字体加颜色**

|  |
| --- |
| #!/bin/bash  RED\_COLOR='\E[1;31m'  GREEN\_COLOR='\E[1;32m'  YELLOW\_COLOR='\E[1;33m'  BLUE\_COLOR='\E[1;34m'  RES='\E[0m'  echo -e "${ RED\_COLOR } ====red color=== ${RES}" |

**范例3: 利用case语句开发类似系统启动rsync服务的脚本(可参考系统的rpcbind/nfs/crond脚本)**

例如: /etc/init.d/rsyncd {start|stop|restart}

|  |
| --- |
| #!/bin/bash  PIDFILE="/var/run/rsyncd.pid"  start\_rsync() {  rsync --daemon  if [ -f "$PIDFILE" ];then  echi "rsync is runing"  else  rsync --deemon  echo "rsync is started"  fi  }  stop\_rsync() {  if [ -f $PIDFILE -a -n "$PIDFILE" ];then  kill -USR2 `cat $PIDFILE`  rm -f ${PIDFILE}  echo "rsync is stopped"  else  echo "rsync no running"  fi  }  case "$1" in  start)  start\_rsync  RETVAL=$?  ;;  stop)  stop\_rsync  RETVAL=$?  ;;  restart)  start\_rsync  sleep 2  stop\_rsync  RETVAL=$?  ;;  \*)  echo "USAGE: $0 {start|stop|restart}"  exit 1  esac  exit $RETVAL |

**练习: 利用case语句开发nginx服务系统启动的脚本,并且可以chkconfig设置自启动**

## 1.4.3 当型循环和直到型循环

1.4.3.1 当型和直到型循环语法

1.while条件句

语法:

**while 条件**

**do**

**指令**

**done**

2.until条件句

语法:

**until条件**

**do**

**指令**

**done**

1.4.3.2 当型和直到型循环基本范例

**范例1: 每隔2s记录一次系统的负载情况**

|  |
| --- |
| [root@localhost ~]# cat while01.sh  #!/bin/bash  while true  do  uptime  sleep 2  done  sh while01.sh & |

===============================

**脚本在后台执行知识扩展:**

|  |  |
| --- | --- |
| 功能 | 用途 |
| sh while.sh & | 把脚本while.sh放在**后台**执行 |
| ctl+c | **停止**执行当前脚本任务 |
| ctl+z | **暂停**执行当前脚本任务 |
| bg | 把当前脚本或任务放到后台执行 |
| fg | 当前脚本或任务拿到前台执行,如果有过个任务,可以通过fg加编号调出,如fg 1 |
| jobs | 查看执行的脚本或任务 |

防止脚本执行中断的方法

1) sh while.sh &

2) screen 总结此命令

3) nohup /server/scripts/uptime &

进程管理命令知识扩展

\*\* 进程管理:

bg: 后台运行 fg: 挂起程序 jobs: 显示后台程序 kill,killall,pkil: 杀掉进程

crontab: 设置定时 ps: 查看进程 pstree: 显示进程状态树

top: 显示进程 nice: 改变优先权 nohup: 用户退出系统之后继续工作

pgrep: 查找匹配条件的进程 **strace: 跟踪一个进程的系统调用**

ltrace: 跟踪进程调用库函数的情况 vmstat: 报告虚拟内存统计信息

扩展: Linux技巧,让进程在后台可靠运行的集中方法

http://www.ibm.com/developerworks/cn/linux/1-cn-nohup/

\*\* nohup用法:

[root@localhost ~]# nohup /server/scripts/uptime &

[1] 19699  
===============================

**范例2: 通过while语句计算从1加到100之和**

|  |
| --- |
| [root@localhost ~]# cat while02.sh  # 法一:  #!/bin/bash  i=1  sum=0  while [ $i -le 100 ]  do  let sum=sum+i  let i=i+1  done  echo $sum  # 法二:  #!/bin/bash  i=1  sum=0  while ((i <= 100))  do  ((sum=sum+i))  ((i++))  done  echo $sum  # 法三: 用公式  Sn=(a1+an)n/2  #!/bin/bash  ((sum=1000000\*(1+1000000)/2))  echo $sum |

参考资料: http://oldboy.blog.51cto.com/2561410/767862

**范例3: 使用while循环竖向打印10,9,8,...,1**

|  |
| --- |
| [root@localhost ~]# seq 10|tac  10  9  8  7  6  5  4  3  2  1  [root@localhost ~]# cat while03.sh  #!/bin/bash  i=10  while (($i>0))  do  echo $i  ((i--))  done |

**范例4: 手机充值10元, 每发一次短信(输出当前余额)花费1角5分钱,当余额低于一角不能发短信,提示余额不足,请充值,请用while语句实现.**

**解答: 单位换算,统一单位,统一成整数**

**10元=1000分**

**1角5分=15分**

|  |
| --- |
| [root@localhost ~]# cat while05.sh  #!/bin/bash  sum=1000  i=15  while (( sum>=i ))  do  ((sum=sum-i))  echo "send message, $sum"  done  echo "money is not enough:$sum" |

**范例5:**

**计算apache一天的日志access\_2010-12-8.log中所有行的日志各元素的访问字节数的总和.给出实现程序.请用while循环实现.**

|  |
| --- |
| 法1:  cat a.log|while read line  do  done  法2:  [root@localhost ~]# cat while06.sh  #!/bin/bash  sum=0  i=0  while read line  do  i=`echo $line|awk '{print $10}'`  if expr $i + 0 &>/dev/null  then  ((sum=sum+i))  fi  done < access\_2010-12-8.log  echo $sum  法3:  exec <a.log  while read line  do  done  其他解决办法:  echo `awk '{print $10}' access\_2010-12-8.log|grep -v -|tr "\n" "+"|sed 's#117+#117#g'`|bc  awk '{print $10}' access\_2010-12-8.log|grep -v "-"|awk '{sum+=$1}END{print sum}' |

**范例6: 每10s做一次rsync binlog推送,守护进程方式,写完脚本后台执行**

|  |
| --- |
| [root@localhost ~]# vim rsync\_binlog.sh  #!/bin/bash  while true  do  rsync -az /data/3306/mysql-bin.0\* rsync\_backup@10.0.0.9:backup --password-file=/etc/rsync.password &  sleep 10  done  [root@localhost ~]# sh rsync\_binlog.sh & |

**抓阄题目:**

要求:1.执行脚本后,想去的同学输入英文名字全拼,产生随机数01-99之间的数字,数字越大就去参加项目实践,前面已经抓住的数字,下次不能在出现相同数字.

2.第一个输入名字后,屏幕输出信息,并将名字和数字记录到文件里,程序不能退出继续等别的学生输入

实现网址: http://oldboy.blog.51cto.com/2561410/1308647

**while循环小结:**

1). while循环的特长是执行守护进程以及我们希望循环不退出,用于频率小于1分钟循环处理(crond),其他的while循环几乎都可以被for循环替代

2). if语句可以替换case语句,一般在系统启动脚本存入少量固定规则字符串,用case,其他判断都用if

3). if,for最常用,其次while(守护进程),case(服务启动脚本)

## 1.4.4 for循环

1.4.4.1 for循环结构语法

1.for循环结构

语法:

|  |
| --- |
| **for 变量名 in 变量取值列表**  **do**  **指令**  **done** |

提示: 在此结构中"in 变量取值列表"可省略,省略时相当于in "$@",使用for i就相当于使用for i in "$@"

1.4.4.2 for循环结构基础例子

**范例1:直接列出变量列表所有元素,打印5,4,3,2,1**

|  |
| --- |
| [root@localhost ~]# cat for1.sh  #!/bin/bash  for n in 5 4 3 2 1  do  echo $n  done  [root@localhost ~]# cat for2.sh  #!/bin/bash  for n in `seq 5`  do  echo $n  done |

**范例2: 获取当前目录下的目录名作为变量列表打印输出**

|  |
| --- |
| [root@localhost ~]# cat for2.sh  #!/bin/bash  for filename in `ls`  do  echo $filename  done  如果是目录  #!/bin/bash  for num in `ls -F|grep /`  do  echo $num  done |

范例3: 用for循环批量修改图片扩展名(请把jpg改成gif)

|  |
| --- |
| [root@localhost ~]# cat for-change-ext.sh  #!/bin/bash  for filename in `ls \*.jpg`  do  mv $filename `echo $filename cut -d . -f1`./gif  done  #rename命令直接实现  [root@localhost ~]# rename .gif .jpg \*.gif |

1.4.4.3 循环结构深入例子

范例4: Linux下批量修改文件名的生产案例

参考: http://oldboy.blog.51cto.com/2561410/711342

范例9: 开发脚本实现仅设置sshd rsyslog crond network sysstat开启自启动

|  |
| --- |
| [root@localhost ~]# cat for-change-ext.sh  #!/bin/bash  LANG=en  for name in `chkconfig --list|grep 3:on|awk '{print $1}'`  do  chkconfig $name off  done  for name in rsyslog nework crond sshd sysstat  do  chkconfig $name on  done |

题目1: 使用for循环在/hello目录下批量创建10个文件.名称依次为:

hello-1

hello-1

hello-1

hello-1

|  |
| --- |
| [root@localhost ~]# cat 01.sh  #!/bin/bash  [ ! -d /hello ] && mkdir -p /hello  for n in `seq 10`  do  touch /hello/hello-${n}.html  done |

升级: 文件前缀也随机字母

题目2: 将以上文件名中的hello全部改成Linux(用for循环实现),并且扩展名改成大写.要求: for循环里不能出现hello字样

|  |
| --- |
| [root@localhost ~]# cat 03.sh  #!/bin/bash  cd /hello  for file in `ls \*.html`  do  mv $file `echo $file|sed 's#hello#linux#g'|sed 's#html#HTML#g'`  done |

参考资料: http://oldboy.blog.51cto.com/2561410/711342

题目3: 批量创建10个系统账号,并设置密码(密码不同,为随机8位字符串)

数字加0的思路: http://oldboy.blog.51cto.com/2561410/788422

|  |
| --- |
| [root@localhost ~]# cat 04.sh  #!/bin/bash  for n in `seq -w 10`  do  useradd pipi$n&&\  echo "root $n"|passwd --stdin pipi$n #无交互设置密码  done  *## 升级版本,密码为随机的8位字符串(包含数字和字符)*  [root@localhost ~]# cat 06.sh  #!/bin/bash  for n in `seq -w 10`  do  useradd pipi3$n&&\  pwd=`echo "$RANDOM"|md5sum|cut -c 1-8`  echo "$pwd"|passwd --stdin pipi3$n  echo -e "pipi3$n\t $pwd " >>/hello/pass.txt  done |

**企业精品shell面试题案例及专家解答精讲**

http://edu.51cto.com/course/course\_id-1511.html

**拓展: Linux系统产生随机的方法**

法一: echo $RANDOM

法二: openssl rand -base64 8

法三: date +%s%N

法四: head /dev/urandom|cksum

法五: cat /proc/sys/kernel/random/uuid

法六: mkpasswd -l 8 (需要安装expect)

# 1.5 BREAK CONTINUE EXIT

1.5.1 break continue exit对比

|  |  |
| --- | --- |
| 命令 | 说明 |
| break n | n表示跳出循环的层数,如果省略n表示跳出整个循环 |
| continue n | n表示退出第n层继续循环,如果省略n表示跳出本次循环,忽略本次循环的剩余代码,进入循环的下一次循环 |
| exit n | 退出当前shell程序,n为返回值.n也可以省略,再下一次shell里通过$?接收这个n值 |
| return n | 用户在函数里,作为函数的返回值,用于判断函数执行是否正确. |

1.5.2 break continue exit范例

**范例1: break跳出整个循环.**

|  |
| --- |
| [root@localhost ~]# cat break1.sh  #!/bin/bash  for((i=0;i<=5;i++))  do  if [ $i -eq 3 ];then  # break;  continue;  fi  echo $i  done |

**范例: 开发shell脚本实现给服务器临时配置多个别名IP,并可以随时撤销配置的所有IP.IP地址为: 10.0.2.1-10.0.2.16.其中10.0.2.10不能配置.**

**配置IP命令(ifconfig/ip)提示: ifconfig ech0:0 10.0.2.10/24 up**

|  |
| --- |
| [root@localhost ~]# cat ip.sh  #!/bin/bash  for ((i=1;i<=16;i++))  do  if [ $i -eq 10 ];then  continue  fi  ifconfig eth0:$i 10.0.2.$i/24 up  done |

**企业实战题:**

**写一个脚本解决DOS攻击生产案例**

提示: 根据web日志或者网络连接数,监控当某个IP并发连接或者短时内PV达到100,及调用命令封掉对应的IP,监控频率每隔3分钟.防火墙命令为: iptables -A INPUT -s 10.0.1.10 -j DROP

解答:

web,也可以分析日志,把单IP PV数高的封掉.按天定义PV=1000即封掉

方法1:

|  |
| --- |
| #!/bin/bash  while true  do  awk '{print $1}' access.log|grep -v "^$"|sort|uniq -c >/tmp/tmp.log  exec </tmp/tmp.log  while read line  do  ip=`echo $line|awk '{print $2}'`  count=`echo $line|awk '{print $1}'`  if [ $count -gt 3 ] && [ `iptables -L -n|grep "$ip"|wc -l` -lt 1 ];then  iptables -I INPUT -s $ip -j DROP  echo "$line is droppes" >>/tmp/droplist.log  fi  done  sleep 5  done |

方法2: 网络连接数

|  |
| --- |
| #!/bin/bash  while true  do  grep EST a.log|awk -F '[ :]+' '{print $6}' |sort |uniq -c >/tmp/tmp.log  #netstat -an|grep EST|awk -F '[ :]' '{print $6}'|sort|uniq -c >/tmp/tmp.log  exec </tmp/tmp.log  while read line  do  ip=`echo $line|awk '{print $2}'`  count=`echo $line|awk '{print $1}'`  if [ $count -gt 3 ] && [ `iptables -L -n|grep "$ip"|wc -l` -lt 1 ];then  iptables -I INPUT -s $ip -j DROP  echo "$line is droppes" >>/tmp/droplist.log  fi  done  sleep 5  done |

**范例7: 编写shell开发Linux系统一键优化脚本**

01.安装系统时精简安装包(最小化安装)

02.配置国内高速yum源

03.禁用开机不需要启动的服务

04.优化系统内核参数/etc/sysctl.conf

05.增加系统文件描述符,堆栈等配置

06.禁止root远程登录,修改ssh端口为特殊端口,禁止DNS,空密码

07.有外网IP的机器需要开启配置防火墙,仅对外开启需要提供服务的端口,配置或关闭SELinux

08.清楚无用的默认系统账户或组(非必须)(添加运维成员的用户)

09.锁定敏感文件,如/etc/passwd(非必须)

10.配置服务器和互联网时间同步

11.配置sudo对普通用户权限精细控制

12.把以上11点写成一键优化脚本

|  |
| --- |
| #!/bin/bash |

# 1.6 SHELL函数

## 1.6.0 为什么要使用shell函数

简单地说函数的作用就是把程序里多次调用的相同的代码,定义成一份,然后起个名字,所有的调用都只用这个名字就可以了.修改代码时,只需要改变函数体内的代码即可.

优势:

1. 把相同的程序定义成函数,可以减少程序代码量

2. 增加程序的可读,易读性

3. 实现程序的功能模块化.

## 1.6.1 shell函数语法

语法格式:

简单的语法:

**函数名() {**

**指令**

**return n**

**}**

规范的语法

**function 函数名() {**

**指令**

**return n**

**}**

## 1.6.2 shell函数执行

调用函数:

1). 直接执行函数名即可.

函数名

注意:

a. 函数体必须在要执行的函数名的前面定义.shell的执行时从上到下按行执行的.

b. 不要带小括号

2). 带参数的函数执行方法

函数名 参数1 参数2

[函数带参数的说明]

a. 在函数体中位置参数**($1,$2,$3,$@)**都可以是函数的参数

b. 父脚本的参数临时地被函数所掩盖或隐藏.

c. **$0比较特殊,他仍然是父脚本的名称**.

d. 当函数完成时,原来的命令行参数会恢复.

e. 在shell函数里,return命令的功能与工作方式与exit相同,用于跳出函数

f. 在shell函数体里使用exit会终止整个shell脚本

g. return语句会返回一个退出值给调用的程序

## 1.6.3 shell函数范例

范例1: 开发脚本建立两个简单函数并调用执行.

|  |
| --- |
| [root@localhost ~]# cat f1.sh  #!/bin/bash  print\_hello  print\_hello() {  echo "hello world, my name is $1"  }  [root@localhost ~]# sh f1.sh #函数调用必须在函数的后面,shell是从上而下执行的  f1.sh: line 2: print\_hello: command not found  --------  [root@localhost ~]# cat f1.sh  #!/bin/bash  print\_hello() {  echo "hello world, my name is $1"  }  print\_hello  [root@localhost ~]# sh f1.sh #函数里的$1是针对函数的  hello world, my name is  --------  [root@localhost ~]# cat f1.sh  **#!/bin/bash**  **print\_hello() {**  **echo "hello world, my name is $1"**  **}**  print\_hello $1 #添加脚本的$1参数为函数的入参  [root@localhost ~]# sh f1.sh hello #必须  hello world, my name is hello  [root@localhost ~]# sh f1.sh nimei  hello world, my name is nimei |

范例2: 把函数体和执行的脚本分离(规范的方法)

|  |
| --- |
| (1) 首先建立函数库脚本  [root@localhost ~]# cat functions.sh  #!/bin/bash  #print string  print\_hello() {  echo "hello world, my name is $1"  }  (2) 执行脚本  [root@localhost ~]# cat f1.sh  #!/bin/bash  [ -f /root/functions.sh ] && . /root/functions.sh # 文件存在则加载该文件  print\_hello $1 |

**综合范例: 使用函数及函数传参,脚本传参实现对指定url的判断异常.**

|  |
| --- |
| #!/bin/bash  [ $# -ne 1 ] && {  echo "USAGE: $0 {url}"  exit 1  }  function check\_url() {  curl -I -s $1|head -1 && return 0 || return 1  }  check\_url $1 |

**企业案例: 开发mysql单实例或多实例启动脚本**

已知mysql多实例启动命令为:

mysqld\_safe --defaults-file=/data/3306/my.cnf &

停止命令为:

mysqladmin -uroot -proot -S /data/3306/mysql.sokc shutdown

请完成mysql多实例启动脚本的编写

要求: 用函数,case语句,if语句等实现

解答: 单实例

1.启动: mysqld\_safe --user=mysql &

停止: mysqladmin -uroot -proot shutdown

2. 开发脚本

|  |
| --- |
| [root@localhost ~]# cat start\_db1.sh  #!/bin/bash  . /etc/init.d/functions  path="/application/mysql/bin"  user=root  password=root  function usage() {  echo "$0 {start|stop|restart}"  exit 1  }  [ $# -ne 1 ] && usage  function start\_mysql() {  $path/mysqld\_safe --user=mysql --pid-file=/application/mysql/mysql/pid >/dev/null 2>&1 &  if [ $? -eq 0 ];then  action "start mysql" /bin/true  else  action "start mysql" /bin/false  fi  }  function stop\_mysql() {  $path/mysqladmin -u$user -p$password shutdown >/dev/null 2>&1  if [ $? -eq 0 ];then  action "stop mysql" /bin/true  else  action "stop mysql" /bin/false  fi  }  if [ "$1" == "start" ];then  start\_mysql  elif [ "$1" == "stop" ];then  stop\_mysql  elif [ "$1" == "restart" ];then  stop\_mysql  start\_mysql  else  usage  fi |

3.将上述脚本加入chkconfig开机启动(man chkconfig)

cp start\_db1.sh /etc/init.d/mysqld01

chmod +x /etc/init.d/mysqld01

#脚本开头加入如下两行

|  |
| --- |
| #!/bin/bash  # chkconfig: 2345 21 60  # description: start mysql and stop mysql scripts.  ... |

chkconfig --add mysqld01

chkconfig myqld01 on

4.用case实现

|  |
| --- |
| [root@localhost ~]# cat start\_db1.sh  #!/bin/bash  . /etc/init.d/functions  path="/application/mysql/bin"  user=root  password=root  function usage() {  echo "$0 {start|stop|restart}"  exit 1  }  [ $# -ne 1 ] && usage  function start\_mysql() {  $path/mysqld\_safe --user=mysql --pid-file=/application/mysql/mysql/pid >/dev/null 2>&1 &  if [ $? -eq 0 ];then  action "start mysql" /bin/true  else  action "start mysql" /bin/false  fi  }  function stop\_mysql() {  $path/mysqladmin -u$user -p$password shutdown >/dev/null 2>&1  if [ $? -eq 0 ];then  action "stop mysql" /bin/true  else  action "stop mysql" /bin/false  fi  }  case "$1" in  start)  start\_mysql  ;;  stop)  stop\_mysql  ;;  restart)  start\_mysql  stop\_mysql  ;;  \*)  echo "USAGE: $0 {start|stop|restart}"  exit 1  esac |

# 1.7 SHELL数组

数组重要知识小结:

定义: array=(1 2 3) array=($(ls))

打印: ${#array[@]} 数组长度

${array[@]} 数组所有元素

${array[i]}i 数组的单个元素

取数组的部分

## 1.7.1 数组介绍

简单的说,数组就是相同数据类型的元素按一定顺序排列的集合.

## 1.7.2 数组定义与增删改查

方法1: array=(value1 value2 value3 ...)

1) 数组定义

[root@localhost ~]# array=(1 2 3)

2) 获取数组的长度

|  |
| --- |
| [root@localhost ~]# echo ${#array[@]}  3  [root@localhost ~]# echo ${#array[\*]}  3 |

3)打印数组元素

|  |
| --- |
| [root@localhost ~]# echo ${array[0]}  1  [root@localhost ~]# echo ${array[1]}  2  [root@localhost ~]# echo ${array[\*]}  1 2 3  [root@localhost ~]# echo ${array[@]}  1 2 3 |

4) 数组赋值

|  |
| --- |
| [root@localhost ~]# array[3]=5  [root@localhost ~]# echo ${array[@]}  1 2 3 5  [root@localhost ~]# array[0]=hello  [root@localhost ~]# echo ${array[@]}  hello 2 3 5 |

5) 数组删除

|  |
| --- |
| [root@localhost ~]# echo ${array[@]}  hello 2 3 5  [root@localhost ~]# unset array  [root@localhost ~]# echo ${array[@]}  [root@localhost ~]# array=(1 2 3)  [root@localhost ~]# unset array[0]  [root@localhost ~]# echo ${array[@]}  2 3 |

6)数组内容的截取和替换

|  |
| --- |
| [root@localhost ~]# array=(1 2 3 4 5)  [root@localhost ~]# echo ${array[@]}  1 2 3 4 5  [root@localhost ~]# echo ${array[@]:1:3}  2 3 4  [root@localhost ~]# echo ${array[@]:3:2}  4 5  [root@localhost ~]# echo ${array[@]/5/6}  1 2 3 4 6 |

**常用语法:**

[root@localhost ~]# array=($(ls))

## 1.7.3 数组实践实战例子

**范例1: 通过列举法打印数组元素**

|  |
| --- |
| [root@localhost ~]# cat arr1.sh  #!/bin/bash  array=(  hello  zhangsan  lisi  )  for ((i=0;i<${#array[\*]};i++))  do  echo "This is num $i,then content is ${array[$i]}"  done  for name in ${array[@]}  do  echo "$name"  done |

**范例2: 把系统命令结果作为数组元素**

|  |
| --- |
| [root@localhost ~]# cd /hello/  [root@localhost hello]# ls  linux-10.HTML linux-2.HTML linux-4.HTML linux-6.HTML linux-8.HTML pass.txt  linux-1.HTML linux-3.HTML linux-5.HTML linux-7.HTML linux-9.HTML  [root@localhost hello]# dir=($(ls))  [root@localhost hello]# echo ${dir[@]}  linux-10.HTML linux-1.HTML linux-2.HTML linux-3.HTML linux-4.HTML linux-5.HTML linux-6.HTML linux-7.HTML linux-8.HTML linux-9.HTML pass.txt |

**范例3: 比较专业的生产检查URL地址的脚本(shell数组方法)**

|  |
| --- |
| function Check\_Url() {  curl -o /dev/null -s -w "%{http\_code}" $1  }  [root@localhost ~]# cat check\_url.sh  #!/bin/bash  . /etc/init.d/functions  url\_list(  http://etiantian.org  http://www..inuxpeixun.com  http://oldboy.blog.51cto.com  http://10.0.0.19  )  function wait() {  echo -n "3秒后,执行该操作"  for ((i=0;i<3;i++))  do  echo -n ".";sleep 1  done  echo  }  function check\_url() {  wait  echo "check url..."  for ((i=0;i<${#url\_list[\*]};i++))  do  # HTTP/1.1 200 OK  #judge=($(curl -I -s --connect-timeout 2 & ${url\_list[$i]}|head -l|tr "\r" "\n"))  judge=($(curl -o /dev/null --connect-timeout 2 -s -w "%{http\_code}" "${url\_list[$i]}"))  if [[ "${judge[1]}" == '200' && "${judge[2]}" == 'ok' ]];then  action "${url\_list[$i]}" /bin/true  else  action "${url\_list[$i]}" /bin/false  fi  done  }  check\_url |

**范例:**

**监控mysql主从同步是否异常,如果异常,则发送短信或者邮件给管理员.**提示:如果没主从同步环境,可以用下面文本放到文件读取来模拟:

阶段1: 开发一个守护进程脚本每30s实现检测一次

阶段2: 如果同步出现如下错误号(1158,1159,1008,1007,1062),请跳过错误

阶段3: 请使用数组技术实现上述脚本(获取主从判断及错误号部分)

|  |
| --- |
| #!/bin/bash  while true  do  array=($(egrep "\_Running|Behind\_Master" slave.log|awk '{print $NF}'))  if [ "${array[0]}" == "Yes" -a "${array[1]}" == "Yes" -a "${array[2]}" == "0" ];then  echo "MySQL slave is ok"  else  char="MySQL slave is not ok"  fi  sleep 30  done  #!/bin/bash  errorno=(1158 1159 1008 1007 1062)  mysql\_cmd="mysql -uroot -proot -stop slave -S /data/3307/mysql.sock"  while true  do  array=($($mysql\_cmd -e "show slave status\G"|egrep '\_Running|Behind\_Master|Last\_Errno'|awk '{print $NF}'))  if [ "${array[0]}" == "Yes" -a "${array[1]}" == "Yes" -a "${array[2]}" == "0" ];then  echo "MySQL slave is ok"  else  for((i=0;i<${#array[\*]};i++))  do  if [ "${array[3]}" -ne "${errorno[$i]}" ];then  $mysql\_cmd -e "stop slave && set global sql\_slave\_skpip\_counter=1;start slave;"  fi  done  char="MySQL slave is not ok"  fi  sleep 30  done |

**范例: 输出一段字符串中小于6个长度的单词**

|  |
| --- |
| 法1:  #!/bin/bash  arr=(I am Piter teacher,welcome to our class.)  for word in ${arr[@]}  do  if [ ${#word} -le 6 ];then  echo $word  fi  done  法2:  #!/bin/bash  arr=(I am Piter teacher,welcome to our class.)  for((i=0;i<${#arr[\*]};i++))  do  if [ ${#arr[$i]} -le 6 ];then  echo ${arr[$i]}  fi  done  法3:  #!/bin/bash  chars="I am Piter teacher,welcome to our class."  for n in $chars  do  if [ ${#n} -le 6 ];then  echo $n  fi  done  法4:  echo " I am Piter teacher,welcome to our class ."|awk '{for(i=1;i<=NF;i++)if(length($i)<=6)print $i}' |

**范例: 监控web站点目录(/var/html/www)下所有文件是否被恶意篡改(文件内容被改了),如果有就打印改动的文件名(发邮件),定时任务每3分钟执行一次(10分钟)**

解答: 文件改动的特征:

a.大小可能会变化

b.修改时间变化

c.文件内容会变化,md5sum指纹

d.增加或者删除文件

|  |
| --- |
| 最简单实用的方法: md5sum  #!/bin/bash  path=/server/script/07  n=`cat $path/check\_site.log`  [ ! -f "$n" ] && touch $path/check\_site.log  while true  do  f=`md5sum -c $path/md5sum.db 2>/dev/null | grep FAILED|wc -l`  find /application/nginx/html/bbs -type f > $path/new\_site.log  log=/tmp/check.log  [ ! -f $log ] && touch $log  if [ $f -ne 0 ] || [ `cat $path/new\_site.log |wc -l` -ne $n ];then  echo " `md5sum -c md5sum.db 2>/dev/null | grep FAILED` " > $log  diff $path/check\_site.log $path/new\_site.log >> $log  mail -s "site is cuangai $(date)" pizh@yonyou.com <$log  fi  sleep 5  done  事先需要建立好的库  find /application/nginx/html/bbs/ -type f|xargs md5sum >/server/scripts/07/md5sum.db  find /application/nginx/html/bbs/ -type f > /server/scritps/07/check\_site.log |

**脚本开发要求**

1.各类监控脚本,内存,磁盘,端口,URL监控报警

2.如何监控网站目录文件是否被篡改,以及站点目录批量被篡改后如何恢复

3.如何开发各类服务rsync,nginx,mysql等的启动及停止专业脚本

4.如何开发mysql主从同步监控报警以及自动处理不同步的脚本

5.一键配置mysql多实例,一键配置mysql主从,N多一键部署脚本

6.监控http/mysql/rsync/nfs等服务是否正常的生产脚本

7.一键软件安装及优化,lanmp,Linux一键优化,一键数据库安装,优化,配置主从

8.mysql多实例启动脚本,分库,分表自动备份脚本

9.根据网络连接数以及根据web日志PV封iP的脚本

10.监控网站的pv以及流量,并且对流量信息进行统计

11.检查web服务器URL地址的脚本,要可以批量及通用.

12.系统的基础优化一键优化的脚本

13.清理系统垃圾(过期备份,clientmquene目录等)脚本

14.tcp连接状态统计

15.批量创建用户并设置随机密码

16.批量获取服务器信息,批量分发文件

# 1.8 SHELL脚本的调试

## 1.8.1 常见错误范例

## 1.8.2 shell脚本调试技巧

使用dos2unix命令处理来自windows脚本

|  |
| --- |
| [root@localhost hello]# **dos2unix** test.sh |

-n: 不执行脚本,仅查询脚本语法是否有问题,并给出错误提示

-v: 在执行脚本时,先将脚本的内容输出到屏幕上然后执行脚本,如果有错误,也会给出错误提示

**-x: 将执行的脚本内容输出显示到屏幕上,对调试很有用的参数**

set -n

set -v

set -x 显示所有命令及参数

减少作用域调试

bashdb专用调试器

1) 直接执行脚本根据报错来调试

2) 要记得首先用dos2unix处理脚本

3) sh -x调试整个脚本,显示执行过程

4) set -x和set +x调试部分脚本

5) echo输出变量及相关内容,然后紧跟着exit退出不执行后面程序的方式,一步步跟踪脚本,对于逻辑错误比较好用

6)最关键的是编码习惯,将错误哦扼杀在萌芽之中,降低减轻调试负担

# 1.9 SHELL脚本开发习惯,规范,制度

1.9.1 命名规范

1.9.2 代码风格规范

1.9.3 注释规范

1.9.4 引用符号使用规范

1.9.5 日志规范

1.9.6 接口文件规范

1.9.7 配置规范

1.9.8 脚本存放规范

1.9.9 补充规范19条

1.9.10 优秀的习惯

# 1.10 SHELL开发环境的配置调整和优化

1.10.1 SecureCRT配置

1.10.2 vim shell的配置

1.10.3 .vimrc实现代码及参数配置

1.10.4 vim shell自动安装配置实现代码

1.105 .virmrc vim配置代码

1.10.6 普通用户.bashrc环境配置