shell和makefile编程

李传煌

2015-03-05

shell编程

一、Shell编程简介
二、Shell中的变量
三、Shell变量中的操作
四、常见Shell命令
五、Shell程序设计

一、shell编程简介

- shell脚本以

  #!/bin/sh

  开始，它通知shell使用系统上的/bin/sh解释器，#符号是注释
- 编辑好脚本以后，要使脚本程序运行，需要改变脚本程序的权限：

  chmod +x filename

二、Shell 中的变量

1、变量赋值和引用

- shell变量类型:只有字符串型,变量可以被赋值,被修改,被引用
- 变量名命名方法:第一个字符为字母,其余字符可以是字母,数字或下划线.
- 变量定义:不需要事先定义,直接赋值定义新变量,直接赋值修改原变量的值
- 变量引用:在变量名前加$号,代表变量内容

例: addr=20.1.1.254

echo $addr

注:1.等号两侧不允许有多余的空格. 2.等号右侧的字符串中含有空格或者制表符,换行符时,要用引号将打算赋值的字符串括起 3.引用一个未定义的变量,变量值为空字符串.

二、Shell 中的变量

2、read:读用户的输入

内部命令read,可以从标准输入上读入一行,并将这行的内容赋值给一个变量 可以用在脚本文件中,接受用户的输入。

3、环境变量和局部变量

- 局部变量:所创建的shell变量,默认为局部变量
- 环境变量: (全局变量)在当前shell下启动的子进程只继承环境变量不继承局部变量

继承:指子进程有自己的一整套独立存储的环境变量,但是这些环境变量的初始状态是从父进程那里原封不动抄写下来的,从此以后,父子进程各保留一套,子进程对全局变量的修改,不影响父进程中的同名变量的值,子进程继续创建它自己的子进程时,这些值生效.

4、内置变量

- 在shell中已经定义的变量，可以在脚本文件中直接使用，且不允许对这些变量赋值

内置变量		含义
$?		最后一次执行的命令的返回码， 0成功,其它值失败
$$		shell进程自己的PID
$!		shell进程最近启动的后台进程的PID
位置变量	$#	命令行参数的个数(不包括脚本文件的名字在内)
	$0	脚本文件本身的名字
	$1 $2,…	第一个,第二个,….,命令行参数
	“$*”	“$1 $2 $3 $4…”,将所有命令行参数组织成一个整体,作为一个单词
	“$@”	“$1” “$2” “$3”…,将多个命令行参数看作是多个“单词”

三、shell中的变量的操作

1、显示变量：

$ HELLO=“Hello, World”

$ echo $ { HELLO }

2、清除变量：

unset variable-name

3、显示所有本地shell变量：

$ set

4、结合变量值：

echo $ { variable-name1 } $ { variable-name2 }

$ TMP_VAR1=“Hello, ”

$ TMP_VAR2=“World”

$ echo $ { TMP_VAR1 } $ { TMP_VAR2 }

Hello, World

三、shell中的变量操作

5、测试变量是否已经设置

- 有时要测试是否已设置或初始化变量。命令格式为：

  $ { variable: -value }

- 如果设置了变量值，则使用它；如果未设置，则取新值。

  $ COLOUR = blue

  $ echo “This colour is $ { COLOUR: -green }”

  This colour is blue

- 变量colour取值blue，echo打印变量colour，首先查看该变量是否已赋值，如果查到，则使用该值。现在清除该值，再来看看结果。

  $ unset COLOUR

  $ echo “This colour is $ { COLOUR: -green }”

  This colour is green

三、shell变量中的操作

6、使用变量保存系统命令参数的替换信息。

```
$ SRC_FILE=‘ /var /log/ messages ’
$ DST_FILE=‘ /var /log/ messages.1’
$ cp  $ { SRC_FILE }  $ { DST_FILE }
```

7、设置只读变量

- 如想设置变量后不再改变其值，可以将该变量设置成只读。

```
variable-name=value
readonly variable-name
```

- 把变量设置成一个常量并把它设为只读，任何改变该变量值的操作都将返回错误信息。

```
$ MAXVAL=65535
$ echo  $ { MAXVAL }
65535
$ readonly MAXVAL
$ MAXVAL=32767
```

三、shell变量中的操作

8、设置环境变量

VARIABLE-NAME=value;
export VARIABLE-NAME

- 环境变量配置有三种方法（分别是：修改/etc/profile文件，修改用户目录下的.bashrc文件，直接在shell下修改）
  比较常用的是修改/etc/profile文件

9、显示环境变量

- 显示环境变量与显示本地变量一样。
  $ MYSHELL='bash'
  $ export MYSHELL
  $ echo ${MYSHELL}
- 使用env命令可以查看所有的环境变量。
  $ env
- 改变PATH：
  $ echo $ PATH
  $ PATH=$ PATH":."

三、shell变量中的操作

10、清除环境变量

- 使用unset命令可以清除环境变量。

```
$ unset MYSHELL
$ echo  $ { MYSHELL }
```

四、常见Shell 命令

- echo “some text”: 将文字内容打印在屏幕上
- ls: 文件列表。
- wc –l file wc -w file wc -c file: 计算文件行数 计算文件中的单词数 计算文件中的字符数。
- cp sourcefile destfile: 文件拷贝。
- mv oldname newname : 重命名文件或移动文件。
- rm file: 删除文件。
- grep 'pattern' file: 在文件内搜索字符串比如：grep 'searchstring' file.txt
- cut -b colnum file: 指定欲显示的文件内容范围，并将它们输出到标准输出设备比如：输出每行第5个到第9个字符cut –b 5-9 file.txt千万不要和cat命令混淆，这是两个完全不同的命令。
- cat file.txt: 输出文件内容到标准输出设备（屏幕）上。
- file somefile: 得到文件类型。
- read var: 提示用户输入，并将输入赋值给变量。
- sort file.txt: 对file.txt文件中的行进行排序。
- Diff： Compare and select differences in two files

四、常见Shell 命令

- uniq: 删除文本文件中出现的行列比如： sort file.txt | uniq。
- expr: 进行数学运算Example: add 2 and 3 expr 2 "+" 3。
- find: 搜索文件比如：根据文件名搜索find . -name filename -print。
- tee: 将数据输出到标准输出设备(屏幕) 和文件比如：somecommand | tee outfile。
- basename file: 返回不包含路径的文件名比如： basename /bin/tux将返回 tux。
- dirname file: 返回文件所在路径比如：dirname /bin/tux将返回 /bin。
- head file: 打印文本文件开头几行。
- tail file : 打印文本文件末尾几行。
- sed: Sed是一个基本的查找替换程序。可以从标准输入（比如命令管道）读入文本，并将结果输出到标准输出（屏幕）。该命令采用正则表达式（见参考）进行搜索。不要和shell中的通配符相混淆。比如：将linuxfocus 替换为 LinuxFocus ： cat text.file | sed ‘s/linuxfocus/LinuxFocus/’ > newtext.file。
- awk: awk 用来从文本文件中提取字段。
- 管道 (|) grep "hello" file.txt | wc -l
- 重定向：将命令的结果输出到文件，而不是标准输出 >,>>

五、shell程序设计

主要的控制流结构

- for 循环
- until 循环
- while 循环
- 使用 break 和 continue 控制循环
- if then else 语句
- case 语句

五、shell程序设计——条件判断

1.条件

条件判断的唯一依据是判定一条命令是否执行成功. 判断方法是根据命令执行的返回码, 返回0, 就算条件成立, 返回非0任意值, 就算条件不成立.

✓内置变量$?:用于返回上个命令执行结束后的返回码的值用管道线连接的若干命令, shell仅采用最后一个命令执行的返回码

2.最简单的条件判断

仅含有一个分支, 条件成立或者不成立时执行相应的命令. 采用符号&&或||

➢若命令1执行成功(返回码为0), 则执行命令2, 否则不执行命令2

格式:　命令1 && 命令2

➢若命令1执行失败(返回码不为0), 则执行命令2, 否则不执行命令2

格式:命令1　|| 命令2

3.命令true与命令false(两个命令文件,均为两个简单的C语言程序)

五、shell程序设计—条件判断

4.命令test与命令[

命令test可以提供一些常用的条件判断，命令[和命令test功能等价,区别是前者要求其最后一个命令行参数必须为右方括号。

test命令主要提供了以下的判断功能：

1)对文件特性的测试

语法格式:test -[dfrwxs]file

-d file---文件file存在且为目录文件

-f file---文件file存在且为普通文件

-r file---文件file存在且为可读文件

-w file---文件file存在且为可写文件

-x file---文件file存在且为可执行文件

-s file---文件file存在且文件长度为非零

例:test -d /home/usera && echo”目录usera存在”

[-d /home/usera] || echo”目录usera不存在或没有此目录”

五、shell程序设计—条件判断

2)对字符串内容的测试

格式:test s----当字符串s为非空时测试结果为真值

s1 = s2---当字符串s1和s2相同时结果为真值

s1 != s2---当字符串s1和s2不相同时结果为真值

-z s1---s1串长度等于0

-n s1---s1串长度不等于0

注意:a.等号与不等号两侧的空格是必不可少的

b.调用某个变量时, $name的引号是必须的

3)用于对整数n的测试

语法:

n1 -eq n2 ---当整数n1与n2相等时,返回真值

n1 -ne n2 ---当整数n1与n2不相等时,返回真值

n1 -lt n2 ---当整数n1小于n2时,返回真值

n1 -le n2 ---当整数n1小于等于n2时,返回真值

n1 -gt n2---当整数n1大于n2时,返回真值

n1 -ge n2---当整数n1大于等于n2时,返回真值

五、shell程序设计—— 条件判断

4)逻辑运算

语法:

!	NOT(非)	-o	OR(或)	-a	AND(与)

五、shell程序设计—— if结构

多分支条件语句: if – then – elif – else- fi结构

```
if [condition_1]
   then
      command_1
 elif [condition_2]
   then
      command_2
 elif [condition_3]
 then
    command_3
    ……
    ……
 else
   command_n
fi
```

例如:

```
  if [ $# = 1 ]
    then
     cp $1 $HOME/user1
     vi $1
    else
echo”you must specify a
      filename.Try again”
 fi
 exit 0
```

五、shell程序设计—— if结构

例：用if 语句实现一个实际应用中的问题

假设有一个连续运行的系统,每当运行中遇到错误时,系统都创建一个错误记录文件errorfile并将错误信息写入其中.而且这个文件是按照错误出现的频率进行更新的,现在要求我们编写一段shell程序,根据errorfile文件产生的特性再生成一个定时错误日志文件,日志文件名为datelog.完成这一功能的shell程序名为checkerr.sh,让errorfile和checkerr.sh文件配合运行,完成每小时记录依次错误产生的情况.

```
#!/bin/sh
date>>datelog
  if test –r errorfile
     then cat errorfile>>datelog
             rm errorfile
    else echo ‘No error this hour’>>datelog
 fi
```

五、shell程序设计—— case结构

case-in结构是基于模式匹配基础上的多条件分支.
case语句的语法为：

```
case  word  in
      pattern –1) pat1-list1;;
      pattern –2) pat2-list2;;
          ……
      *)default-list;;
esac
```

说明:
a.模式描述时,使用shell的文件名匹配原则
b.;;是一个整体,不可使用空格等分隔
c.在右括号和;;之间可以夹着多个命令定义的程序块,本程序块可以有多个命令,也没必要用大括号或者括号括起.
d.可以使用竖线罗列出多个模式
e.当word可以与多个模式匹配时,只执行它所遇到的第一个命令表

五、shell程序设计—— case结构

例1:向指定的文件中添加信息

```
#!/bin/sh
#filename:append.sh
case “$#” in
        1) cat >> $1;;
        2) cat >> $2 < $1;;
        *) echo ’usage:append.sh [from] to’;;
esac
```

如果在引用脚本文件时没有携带任何参数,容易引起case语法错误,所以最好带上双引号

例2:编写一段shell程序,根据执行时获取的当前时间显示出不同的问候信息.

```
#!/bin/sh
#例题 wh.sh
# case 结构
hour=`date +%H`
case $hour in
0[1-9|1[01]]) echo “Good morning!”;;
1[234567]) echo “Good afternoon!”;;
        *) echo “Good evening!”;;
    esac
```

竖线罗列

五、shell程序设计—循环结构

1、while循环：while-do-done结构

语法结构为:

```
while[condition]
do
  commands
   ……
 last-command
done
```

说明:

a.关键字必须以独立命令行的首个单词的身份出现,以确保shell会有机会把它们处理成内部命令

b.do 和done之间的一段程序算作循环体,因此不需要在有多个命令时加大括号或括号

c.行与行合并时,应注意在行之间加上分号(;)

d.可以交互式使用while结构

e.冒号命令和true命令可以用于条件中作为永真的条件

例: while : ; do ls –l mydata; sleep 10; done

五、shell程序设计—循环结构

例：while循环

```
#!/bin/sh
      while [ -r abc.c ]
      do echo’Before sleep……’
         sleep 5
         echo’sleep done’
      done
```

2、 until循环:Until-do-done结构

until循环与while循环类似,所不同的是until循环只要循环条件为假(非0值),就执行循环体,其语句格式如下:

```
  until  [condition]
  do
     commands
     …….
     last-command
done
```

五、shell程序设计—循环结构

说明：

- 如果在第一次执行时,循环条件就为真,则循环可能会永远不执行
- 必须在程序中设置能使条件为真的因素(注意:对while来讲是判别条件为假,而until是判别条件为真),否则该循环会成为一个无限循环的死循环程序(若出现此问题用户可以使用kill命令杀死此进程)

例：编一个程序,查看指定用户是否登录到系统上.如果没有,则在他登录时进行报告

```
#程序uon:查询用户“***”是否在系统中
#
until who | grep ”$1” > /dev/null
do
   sleep 30
done
echo”\07\07 $l is logged on.”
exit 0
```

五、shell程序设计—循环结构

3、for循环：for-in –done结构

for 循环的语法格式为:

```
for variable
in list-of-values
do commands
……
    last-command
done
```

循环体被执行多次,给出一个由多个单词构成的表格,每次循环,循环控制变量取值是表格中的一个单词,若没有指定循环控制变量的取值表,系统就会用shell的位置变量中的$1,$2,…来作为循环控制变量的取值表.

```
for name                          for name in $1,$2….
    do list                           do list
done                              done
```

五、shell程序设计—循环结构

例：列出用户注册目录下的cc和work 子目录中所有C语言源程序文件

```
#!/bin/sh
# 显示*.C 文件
 cd $HOME
 for dir in cc work
 do echo”……in $dir ……”
 cd $dir
      for file in *.[c]
          do ls –l $file
          done
 cd ..
 done
```

五、shell程序设计—循环结构

例:给出一组程序名,终止这些程序文件启动的所有进程

```
$ cat k
  for name
  do
      echo “$name: \c ”
      PID= `ps –e | awk “/[0-9] : [0-9][0-9] $name \\$/{ printf(\\”%d\\”,\\$1) }” `
      if [ -n “$PID” ]
      then
          echo kill $PID
          kill    $PID
      else
          echo No process
      fi
  done
$ k myap findkey sortdat
 myap: kill 20608 27336 28072 29720
 findkey: kill 36994 37948
 sortdat:  No process
```

五、shell程序设计—循环结构

例：编写一段shell程序完成:根据从键盘输入的学生成绩,显示相应的成绩标准(分出及格和优秀等)

```
#!/bin/sh
  #score.sh
  echo –e “Please enter the score:\c”
  while read SCORE
  do
     case $SCORE IN
     ?|[1-5]?)  echo ‘Failed!’
        echo –e ‘Please enter the score:\c’;;
      6?)       echo “Passed!”
        echo –e”Please enter the score:\c”;;
      7?)       echo ‘Medium!’
        echo –e” Please enter the score:\c”;;
      8?)       echo”Good!”
     echo –e” Please enter the score:\c”;;
    9?|100)         echo”excellent”
   echo –e” Please enter the score:\c”;;
       *)      echo exit;;
     esac
done
```

makefile 编程

——实现大型工程的自动化编译功能的shell程序

Makefile简介

- Makefile文件保存了编译器和连接器的参数选项，还表述了所有源文件之间的关系(源代码文件需要的特定的包含文件,可执行文件要求包含的目标文件模块及库等)

- 创建程序(make程序)首先读取Makefile文件，然后再激活编译器、汇编器、资源编译器和连接器以便产生最后的输出，最后输出并生成的通常是可执行文件。

makefile规则

- target…:prerequisites…
  - target目标文件，obj或者exe文件，标签
  - prerequisites依赖文件或者目标
  - 如果prerequisites中的文件更新了，所有的command需要重新执行。
- command1
- command2…
  - 要执行的命令，任意shell命令

举例

- edit : main.o kbd.o command.o display.o \
  insert.o search.o files.o utils.o
  cc -o edit main.o kbd.o command.o display.o \
  insert.o search.o files.o utils.o
- main.o : main.c defs.h
  cc -c main.c
  clean :
  rm edit main.o kbd.o command.o display.o \
  insert.o search.o files.o utils.o
- 反斜杠（\）是换行符的意思。这样比较便于Makefile的易读

makefile工作过程

- 输入make命令
- 寻找makefile文件
- 寻找目标如edit
- 寻找依赖的.O文件
- 如果依赖文件不存在，则寻找此文件的依赖（堆栈）
- clean:需要运行make clean:强制执行

变量

* 使用变量可以减轻重复，可读性更好
* objects = main.o kbd.o command.o display.o \
  insert.o search.o files.o utils.o
* edit : $(objects)
  cc -o edit $(objects)
  main.o : main.c defs.h
  cc -c main.c
  clean :
  rm edit $(objects)

make 自动推导

- objects = main.o kbd.o command.o display.o \
  insert.o search.o files.o utils.o
- edit : $(objects)
  cc -o edit $(objects)
- main.o : defs.h

更加简洁的版本

- objects = main.o kbd.o command.o display.o \
  insert.o search.o files.o utils.o
- edit : $(objects)
  cc -o edit $(objects)
- $(objects) : defs.h
  kbd.o command.o files.o : command.h
  display.o insert.o search.o files.o : buffer.h

清空目标文件注意

- .PHONY : clean
  clean :
  -rm edit $(objects)
- .PHONY意思表示clean是一个“伪目标”，。而在rm命令前面加了一个小减号的意思是，也许某些文件出现问题，但不要管，继续做后面的事。
- clean的规则不要放在文件的开头，不然，这就会变成make的默认目标，相信谁也不愿意这样。不成文的规矩是——“clean从来都是放在文件的最后”。

一些make知识

- makefile文件名
  - Makefile, makefile
  - 其他名字，make –f Make.Linux
- 引用其他makefile
  - include <filename>: include foo.make
- 命令前加-号表示忽略错误，继续执行
- 命令开头需要tab开始
- 注释#
- 显示@echo 信息。make –s(--slient)禁止

makefile通配符

* “*”，“?”和“[...]，和shell一样
* “～”表示$HOME目录
* %，表示0个或者多个匹配
* $@表示目标集合，$<依赖目标集，数组？
  * objects = foo.o bar.o
  * all: $(objects)
  * $(objects): %.o: %.c
    $(CC) -c $(CFLAGS) $< -o $@
  * foo.o : foo.c
    $(CC) -c $(CFLAGS) foo.c -o foo.o
    bar.o : bar.c
    $(CC) -c $(CFLAGS) bar.c -o bar.o

自动化变量

- $%,仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是"foo.a(bar.o)"，那么，"$%"就是"bar.o“
- $?, 所有比目标新的依赖目标的集合
- $^,有的依赖目标的集合,去掉重复的
- $+，很像“$^”，也是所有依赖目标的集合。只是它不去除重复的依赖目标。
- $*，表示目标模式中"%"及其之前的部分。如果目标是"dir/a.foo.b"，并且目标的模式是"a.%.b"，那么，"$*"的值就是"dir/a.foo"

变量

- 前面加$使用，最好用() {}包括。真正$用$$表示。
- =，与:=赋值不同
- 变量替换
  - foo := a.o b.o c.o
    bar := $(foo:.o=.c)，bar := $(foo:%.o=%.c)
- 追加变量+=
- 多行变量define var ....endef

条件判断

- <conditional-directive>
  <text-if-true>
  else
  <text-if-false>
  endif
- ifeq...ifdef, ifndef

函数

- $(<function> <arguments>)
  - subst ,字符串替换
  - patsubst, 模式字符串替换
  - strip 去空格
  - findstring 查找字符串
  - filter， 过滤
  - filter_out, 反过滤
  - sort, 排序
  - word，取单词，wordlist，取单词串

文件操作函数

- dir 取目录函数（从一个字符串中）
- notdir取非目录部分
- suffix取文件后缀
- basename取前缀函数
- addsuffix加后缀函数
- addprefix加前缀函数
- join连接
- call创建参数化的函数
- shell