**shell基础1.变量**

1.自定义变量

    定义方法：变量名=变量值，例如a=1，就是把1赋值给a这个变量。

    使用方法：$变量名，例如a=1;echo $a结果就是1了。

    删除方法：uset 变量名

2.环境变量

    定义方法：export 变量名，例如export a，就是定义一个环境变量a，环境变量一单定义，删除之前，在该计算机环境中有效，而自定义变量值在当前终端有效。

3.预定义变量：

    定义：预定义变量是系统中预制的变量，又叫内部变量。有其特殊含义，比如$?、$0、$1、$2、$\*、$#、$$...等等，$?的值若为0，则说明上个命令执行成功，若为1则说明上个命令执行错误，$1代表第一个参数，$2代表第二个参数...$\*代表所有的参数，$0返回当前shell，$#代表参数的个数，$$显示当前进程号。

**shell基础2.条件测试**

shell的条件测试使用test，常用[  ]替代test[  ]里面两边都有空格，不然报错。常用判断：-d（是否目录、-f(是否文件)、-e（是否存在）、-z（字符串长度是否为0）、-n（字符串长度是否不为0）等等。常用数字比较测试：-eq（是否相等）、-ne（是否不等）、-lt（是否小于）、-gt（是否大于）。常用字符串比较测试：=（是否相同）！=（是否不同）等等

条件测试多用于判断语句当中。  
  
例1：如要判断一个输入的文件是否存在，如果存在则删除它可以这样写：

#!/bin/bash

read filename

if [ -f ${filename} ]

then

    rm -f ${filename}

fi  
  
例2：输入两个参数，如果两个相等，则输出”$1=$2“否则输出”$1<>$2“  
  
#!/bin/bash  
if [ $1 -eq $2 ]  
then  
    echo "$1=$2"  
else  
    echo "$1<>$2"  
fi  
当然这里的例子写的比较简单，没有加入任何错误处理机制。

**shell基础3.字符串的截取**

首先说一下管道，管道符用|，管道的作用就是把前一个命令执行的结果送到后面，继续加工。字符串的截取当中管道很常用。

例1：假如在当前目录有一个文件file1，现在要截取出该文件中含有ip的行，常用以下命令

cat file1 | grep ip

当然也可以使用 grep ip file1

小技巧：

有时候我们在看一个注释行特别多的文件时，感觉有用行特别难找，这时就用到了grep -v '^#'，grep -v是不显示匹配行的意思，而^代表行首的意思，这里的意思就是不显示行首的#，而行尾用$代表。

另外有时候不但要找包含ip的行，还要找包含error的行，可以使用egrep来匹配，例1可以这样改写：cat file1 | egrep "ip|error"，当然这里不一定只匹配两个字符串，也可以使用此法匹配多个字符串。

例2：要搜索出系统进程中包含oracle字符的进程可以使用以下命令

ps -ef | grep oracle

下面来看awk，awk是一个编程器，功能比较强大，在shell编程中设计到行和列中字符截取的常用awk，awk截取时，列默认用空格或者table分隔。如果要用其他字符来分隔用-F字符，行用NR来表示，例如第一行用NR==1，前3行用NR\<=3，这里的\是转义字符，在shell中>代表输入重定向，加了\以后才表示小于。

例3：要截取/etc/passwd文件的前3行的用户id（以:分隔第3列）可以使用下面命令

cat /etc/passwd | awk NR\<=3 | awk -F: '{print $3}'

这里awk -F:代表以冒号分隔，'{print $3}'代表打印出来第三列。

上例用如下命令也可以实现

cat /etc/passwd | awk NR\<=3 | cut -d: -f3

cut -d类似于awk -F而'{print $3}'类似于-f3，这里明显看到取列的时候用cut -d比较简单。

今天有点累，感觉视力有点下降，先到这里。  
  
下面来点实用的东西，很多讲shell的书，里面语法讲的很多，但是看了以后还是不知道shell怎么用。  
  
例4：要查看/分区的磁盘使用率可以使用如下命令  
  
 df -h | awk NR==2'{print $5}'  
  
这样取到的第2行第5列是一个百分数，一般用来做磁盘报警的话，用整数比较方便，此例可以改写成下面命令  
  
df -h | awk NR==2'{print int($5)}'  
  
这里int($5)是把第5列转换为整数，下面是一个磁盘报警的例子  
  
例5：  
  
#!/bin/bash  
disk=`df -h | awk NR==2'{print int($5)}'`    #这里用的是但撇号，即esc键下面的键，shell中要把一个命令的输出赋值给一个变量，这个命令就要用但撇号扩起来

if [ ${disk} -ge 90 ]  
then  
    echo "/ is lager 90"

elif [ ${disk} -ge 70 ]  
then  
    echo "/ is lager 70"  
fi  
  
小知识：  
上面例子中，命令结果赋值给变量的时候用了但撇号，我们看到shell中有时候用单引号，有时候用双引号，那么这些有什么区别呢？单引号引起来的内容，会原样输出，双引号引起来，有特殊含义的会转换，比如双引号中的$变量名，输出的会是变量值。来看下面例子  
  
例6：  
1.a=5;echo a              输出结果为a，因为引用变量的时候前面要加$，这里没加，所以就输出了a，与前面定义的变量a没有任何关系，这也是常见错误引用变量的方法。  
2.a=5;echo $a             输出结果为5  
3.a=5;echo '$a'           输出结果为$a，单引号中的原样输出。  
4.a=5;echo "$a"          输出结果为5，双引号中的$a，会当成变量a来输出。  
  
这里2和4的结果相同，我们为什么还要使用双引号呢？应为很多时候输出的不是一个变量，而是一串，中间会有空格，这时候，双引号引起来的就会当成一个整体来处理，不用双引号，就会报错了。

**shell基础4.重定向**

重定向分为输入重定向和输出重定向。标准的输入用0表示，标准的正确输出用1表示，标准的错误输出用2表示。标准的正确输出和标准的错误输出加在一起用&表示。

例1：要将ls -l输出的结果写到一个文件/tmp/file1中，可以这样实现

ls -l >/tmp/file1

这里大于号是重定向的意思，使用“>”来重定向，如果目标文件不存在，会创建，如果目标文件存在且不为空，会把目标文件内容清空，在写入重定向的内容，常用echo "">文件名来清空一个文件，“>>”是追加重定向，如果目标文件存在，且不为空，使用双大于号，会在目标文件末尾继续追加重定向的内容。

在上例中，执行ls -l不会有错误出现，但是不是所有的命令执行结果都不会有错，看下面的例子

例2：假设在某shell脚本中有一个bug，在某个步骤需要读取一个文件/file2的内容，在有些情况下，/file2这个文件时不存在的，但是脚本中在cat /file2语句执行之前，没有判断/file2是否存在的语句，那么当/file2不存在的时候，执行cat /file2在屏幕会打印出“cat: /file2: No such file or directory”，那么我们的脚本在执行过程中，不允许屏幕出现类似的报错，该怎么处理？可以用下面命令代替cat /file2

cat /file2 2> /dev/null

这里的意思是把标准的错误输出放到/dev/null（俗称linux黑洞，重定向到这个里面的东西，就像进了黑洞一样消失了），这样，即使出现报错，也不会在屏幕打印出来了。

注意：常见命令如：cat /file2 2>&1 /dev/null意思是把cat /file2执行结果不管是正确输出还是错误输出，全部输出到/dev/null当中去，这样避免了在屏幕打印任何东西。

在来看输入重定向，输入重定向最常用的地方就是read line

例3：某日志文件/tmp/a.log中，部分错误信息是Error打头，部分错误是error打头，现在要把以Error打头的错误输出到/tmp/Error\_tmp中，把以error打头的输出到/tmp/error\_tmp中，可以用以下脚本实现

#!/bin/bash  
Error\_file="/tmp/Error\_bak"  
error\_file="/tmp/error\_bak"  
log=/tmp/a.log

while read line  
do  
test\_1=`echo ${line} | grep Error`  
test\_2=`echo ${line} | grep error`

if [ -n "${test\_1}" ]  
then  
    echo "${line}">>${Error\_file}  
elif [ -n "${test\_2}" ]  
then  
    echo "${line}">>${error\_file}  
else  
    echo "${line}">>/dev/null  
fi  
done<${log}

当然上面的功能可以用更简单的办法实现，这里只是为了说明输入重定向而已。

**shell基础5.判断和循环**

shell中判断一般使用if else语句，或者case语句，而循环常用for in循环或者while循环。

例1：判断一个输入的文件是否存在，如果存在，则输出“exist！”，如果不存在则输出“not exist！”，可以使用如下脚本实现

#!/bin/bash

echo -n "please input a path:"  
read file\_path  
if [ -f ${file\_path} ]  
then  
    echo "exist"  
else  
    echo "not exist"  
fi

上例中使用if else语句比较容易实现，而有所后使用case语句来判断比较容易实现。

例2：在写oracle服务的启动、关闭、重启脚本时，虽然可以使用if else语句实现，但是没有case语句好写，看下面脚本

脚本1：

#!/bin/bash  
a=$1

if [ $a = start ]  
then  
    echo "oracle service starting......"  
elif [ $a = stop ]  
then  
    echo "oracle service stopping......"  
elif [ $a = restart ]  
then  
    echo "oracle service restarting......"  
else  
    echo "Usage: $0 {start|stop|restart}"  
fi

脚本2：

#!/bin/bash  
a=$1  
case $a in  
    start)  
        echo "oracle service starting......"  
        ;;  
    stop)  
        echo "oracle service stoping......"  
        ;;  
    restart)  
        echo "oracle service restarting......"  
        ;;  
    \*)  
        echo "Usage: $0 {start|stop|restart}"  
        ;;  
esac

很显然脚本2比脚本1简单一些，但是不能说明case语句比if else语句好，只是case语句更善于处理这种定值判断，而如果是区间判断，还是必须要用if else的。没有最好的，只有更适合而已。

例3：假如要创建1-50共50个测试用户，可以用以下脚本实现

#!/bin/bash

for i in `seq 50`  
do  
    useradd test\_$i  
done

这里seq 50可以自动生成1到50共50个连续数字。

例4：上例中我们创建了50个测试用户，现在需要删除这50个用户，可以使用下面脚本实现

#!/bin/bash  
i=1

while [ $i -le 50 ]  
do  
    userdel test\_$i  
    i=`expr $i + 1`  
done

while循环中，只有while后面所带的条件为真就会一直执行，知道为假才跳出循环。

当然shell中判断、循环语句结构不止这些，这里只介绍比较常用的几个。

**shell基础6.函数**

函数，其实就是把一段可以实现某个功能的代码封装起来，到使用的时候引用一下，所以函数需要先定义后使用

函数的定义：      
法1：

function func\_name(){    
    函数体内容    
}    
法2：    
func\_name(){    
    函数体内容    
}

法2是常见shell函数定义方法。

调用函数的时候，直接写func\_name即可，不用写()，一般认为，把经常用的功能定义为函数，使用时在调用，这样可以避免写很多重复代码，但是在大多数规模较大的脚本中，几乎所有的功能都写成了函数，这也就满足了模块化编程思想，如果出现问题，只改问题模块即可，而且规模较大的脚本，如果不使用函数，很难读懂，使用函数，使代码变得清晰，提高了代码的可读性，当然，每个函数不要写的太长，定义的时候命名要易懂。

例1：要写一个简单的备份脚本，将制定目录（这里是/etc）的文件备份到/tmp，并且备份文件名以日期-时间来命名，可以用以下脚本实现

#!/bin/bash  
file\_sou=/etc/  
file\_des=/tmp/

test\_sou()  
{  
    [ ! -d ${file\_sou} ] && echo "bak souce is not exist!" && exit  
}

bak\_file()  
{  
    time\_all=`date "+%Y\_%m\_%d-%H:%M:%S"`  
    tar czvf ${file\_des}${time\_all}.tar.gz ${file\_sou} &> /dev/null  
    if [ $? -eq 0 ]  
    then  
        echo "backup is successful!"  
    else  
        echo "backup is failed!"  
    fi  
}

main()  
{  
    test\_sou  
    bak\_file  
}

main

这个脚本除了前面定义变量的部分外，是从main开始执行的，因为main以上部分除了定义变量部分以外，就是定义函数部分，函数只有在调用的时候才运行。该脚本如果将tar命令后面重定目标改为一个文件，那么就相当于给tar命令记录了日志。

《shell基础部分完》