shell

有一个shells 的PDF

man case
修改用户支持的shell

vim /etc/shells -------系统支持的shells
/bin/sh
/bin/bash
/sbin/nologin
/usr/bin/sh
/usr/bin/bash
/usr/sbin/nologin
/bin/tcsh
/bin/csh
修改：
usermod -s /bin/tcsh test1
交互式修改：
chsh test1
Changing shell for test1.
New shell [/bin/tcsh]: /bin/bash
Shell changed.
• 内部命令和外部命令
• Shell不需要启动一个单独的进程来运行内部命令
• Shell需要创建（fork）和执行（exec）一个新的子进程来运行外部命令
• 尽量使用内部命令有助于性能的提升
type cd history pwd help
cd is a shell builtin ——内部命令
内部命令
type history cd help alias pwd time exit：
type ls
ls is aliased to `ls --color=tty' ——外部命令

连续：
[1-5] 1到5的其中一个
{1..10} 1到10
或者关系：
{1，10}1或者10
[12345]或者
非关系：
[^12345] [^]取反
[!12345] [!]取反

转义
touch test*\?\[\].sh \ 脱义符 续行符

bash 特殊符号：
'' ---单引号 视为一个整体，但是不解析特殊含义 脱义
"" ---双引号 视为一个整体，解析特殊含义 变量一般价格双引号
`` ---优先执行
$() ---同上 执行一条命令
$(()) ---运算
$[] ---运算
expr ---运算 符号左右要有空格 expr 1 + 1
; ---可对一行命令进行分割，在执行过程中不考虑上一个命令执行是否是正确的 分隔两条命令

&&　　---可对一行命令进行分割,在执行中需要前一条命令执行成功，再执行下一条命令
||　　---或关系，前面执行不成功，再执行后一条
!　　---命令历史
!!　　---最后一条命令
!$　　---最后一条命令的参数

大写	[[:upper:]]	[A－Z]
小写	[[:lower:]]	[a-z]
字母	[[:alpha:]]	[a-Z]
字母数字	[[:alnum:]]	
空格或者制表符	[[:blank:]]	
纯数字	[[:digit:]]	[0-9]
标点符号	[[:punct:]]	

```
[root@rhel7 ~]# echo today is `date +%F`
today is 2002-01-07
[root@rhel7 ~]# echo today is $(date +%F)
today is 2002-01-07
```

用户登陆之后读取环境变量顺序　　读取后面面会把前面的覆盖 所以修改后面的

```
[kiosk@foundation0 Desktop]$ ssh root@rhel7
Last login: Sun Jan  6 23:58:30 2002 from 172.25.0.250
this is /etc/profile
this is $HOME/.bash_profile
this is $HOME/.bashrc
this is /etc/bashrc
```

user01-->login-->bash-->/etc/profile-->$HOME/.bash_profile-->$HOME/.bashrc-->/etc/bashrc

自定义变量：　　有空格要要用“”不能用数数字开头

VARNAME=value

```
[root@rhel7 ~]# ABC=123
```

ABC--变量名称
123--对变量赋值

```
[root@rhel7 ~]# echo $ABC
123
```

--调用变量　变量加个$

取消变量：

```
[root@rhel7 ~]# unset ABC
```

只读变量

```
[root@rhel7 ~]# readonly ABC
[root@rhel7 ~]# ABC=456
-bash: ABC: readonly variable
[root@rhel7 ~]# unset ABC
-bash: unset: ABC: cannot unset: readonly variable
```

---不能修改

---不能删除

${}　　----界定变量范围

ABC=123

```
[root@rhel7 ~]# echo ${ABC}D
123D
[root@rhel7 ~]# echo ${ABC:1:2}D
23D
[root@rhel7 ~]# echo ${ABC:0:2}D
12D
```

从第二个变量起，输出两个

从第一个变量起，输出两个

set　　----查看自定义变量

```
[root@rhel7 ~]# set |grep ABC
ABC=123

env       ----系统变量

[root@rhel7 ~]# echo $PATH                          输出环境变量
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
[root@rhel7 ~]# echo $HOME
/root
[root@rhel7 ~]# echo $PWD                           当前路径
/root
[root@rhel7 ~]# echo $SHELL
/bin/bash

[root@rhel7 ~]# echo $PS1
[\u@\h \W]\$
```

PS1和PS2

提示符变量，用于设置提示符格式

PS1是用于设置一级Shell提示符的环境变量，也称为主提示符字符串，即改变： [root @jselab ~]#

PS1变量是[\u@\h \W]\$，\u、\h、\W和\$表示了特定含义，\u表示当前用户名，\h表示表示主机名，\W表示当前目录名，如果是root用户，\$表示#号，其他用户，\$则表示$号

PS2是用于设置二级Shell提示符的环境变量

\d	以"周 月 日"格式显示的日期
\H	主机名和域名
\h	主机名
\s	Shell的类型名称
\T	以12小时制显示时间，格式为：HH:MM:SS
\t	以24小时制显示时间，格式为：HH:MM:SS
\@	以12小时制显示时间，格式为：am/pm
\u	当前的用户名
\v	bash Shell的版本号
\V	bash Shell的版本号和补丁号
\w	当前工作目录的完整路径
\W	当前工作目录名字
\#	当前命令的序列号
\$	如果UID为0，打印#；否则，打印$

定义一级命令提示符

```
[root@-bash rhel712:49:53] $PS1="[\u@\s \H\T]\\$"
```

定义二级命令提示符

```
echo $PS2

[root@-bash rhel712:50:20]#echo $PS2
>

[root@-bash rhel712:51:31]#PS2="##"       ---修改二级命令提示符
```

新的PATH

PATH=$PATH：新的路径

```
vim abc.sh
#!/bin/bash

echo "$1 is 第一个参数"
echo "$2 is 第二个参数"
echo "$# 命令传递参数的个数"
```

```
echo "$* 命令行传递的所有参数"          整体打印出来
echo "$@ 命令行的所有参数"              单独打印出来
echo "$$ 进程pid"
echo "$! 后台运行的最后一个进程号"
echo "$0 当前进程名称"
```

添加执行权限
chmod +x abc.sh

```
[root@rhel7 ~]# ./abc.sh 1 2 3 4 5
1 is 第一个参数
2 is 第二个参数
5 命令传递参数的个数
1 2 3 4 5 命令行传递的所有参数
1 2 3 4 5 命令行的所有参数
2640 进程pid
 后台运行的最后一个进程号
./abc.sh 当前进程名称
```

```
[root@rhel7 ~]# declare -i A=3                    数值定义类型
[root@rhel7 ~]# declare -i B=7
[root@rhel7 ~]# declare -i RESULT=$A*$B
[root@rhel7 ~]# echo $RESULT
21
```

有类型变量

- 默认bash将变量设置为文本值，当使用算数方法时会自动将其转换为整数值
- 内置命令declare可以修改变量属性
- **declare参数**
  - -a 将变量看成数组
  - -f 只使用函数名
  - -F 显示未定义的函数名
  - -i 将变量看成整数
  - -r 使变量只读
  - -x 标记变量通过环境导出

export　和decllare-x　同意

数组就是一堆变量，但是在内存空间是连续的，这样就可以用一个名字（下标）引用他们，就管他们叫数组

- 数组
  - 数组类似于保存取值的一个排列
    排列中每个位置称为元素
    每个元素通过数字下标访问
  - 数组元素可以包含字符串或数字
  - 数组下标从0开始

```
[root@rhel7 test]# for i in "${A[*]}" ; do mkdir "$i" ;done
drwxr-xr-x. 2 root root 6 Jan  7 01:41 a b c d e f
[root@rhel7 test]# for i in "${A[@]}" ; do mkdir "$i" ;done
drwxr-xr-x. 2 root root 6 Jan  7 01:42 a
drwxr-xr-x. 2 root root 6 Jan  7 01:41 a b c d e f
drwxr-xr-x. 2 root root 6 Jan  7 01:42 b
drwxr-xr-x. 2 root root 6 Jan  7 01:42 c
drwxr-xr-x. 2 root root 6 Jan  7 01:42 d
drwxr-xr-x. 2 root root 6 Jan  7 01:42 e
drwxr-xr-x. 2 root root 6 Jan  7 01:42 f
```

```
[root@rhel7 test]# AB=([1]=a b c d [6]=7 8 9 [11]=m)
[root@rhel7 test]# for i in {0..15} ;do echo \${AB[$i]}=${AB[i]};done
${AB[0]}=
${AB[1]}=a
${AB[2]}=b
${AB[3]}=c
${AB[4]}=d
${AB[5]}=
${AB[6]}=7
${AB[7]}=8
${AB[8]}=9
${AB[9]}=
${AB[10]}=
${AB[11]}=m
```

##########有时间看一下##########

利用 ${ } 还可针对不同的变量状态赋值(没设定、空值、非空值)：

${file-my.file.txt} ：假如 $file 没有设定，则使用 my.file.txt(临时给一次) 作传回值。(空值及非空值时不作处理)

${file:-my.file.txt} ：假如 $file 没有设定或为空值，则使用 my.file.txt 作传回值。 (非空值时不作处理)

${file+my.file.txt} ：假如 $file 设为空值或非空值，均使用 my.file.txt 作传回值。(没设定时不作处理)

${file:+my.file.txt} ：若 $file 为非空值，则使用 my.file.txt 作传回值。 (没设定及空值时不作处理)

${file=my.file.txt} ：若 $file 没设定，则使用 my.file.txt 作传回值，同时将 $file 赋值为 my.file.txt 。 (空值及非空值时不作处理)

${file:=my.file.txt} ：若 $file 没设定或为空值，则使用 my.file.txt 作传回值，同时将 $file 赋值为 my.file.txt 。 (非空值时不作处理)

${file?my.file.txt} ：若 $file 没设定，则将 my.file.txt 输出至 STDERR。 (空值及非空值时不作处理)

${file:?my.file.txt} ：若 $file 没设定或为空值，则将 my.file.txt 输出至 STDERR。 (非空值时不作处理)

当file变量为没设定值

```
echo ${file-file.txt}          ---将file.txt 作为值输出一次
echo ${file=file.txt}          ---将file.txt 作为file变量的赋值定义
echo ${file+file.txt}          ---不做处理
```

当file为空值：

```
[root@rhel7 test]# echo ${file:+file.txt}

[root@rhel7 test]# file=
[root@rhel7 test]# echo ${file:=file.txt}       ---将file赋值为file.txt
file.txt

echo ${file-file.txt}                            ----不做处理
```

非空值

```
echo ${file-file.txt}          ----不处理
echo ${file+file.txt}          ----返回file.txt
echo ${file=file.txt}          ----不处理
```

##########

当file为没设定时，做错误输出

```
[root@rhel7 test]# ${file?value is wrong}
-bash: file: value is wrong
```

变量切片：
${file:0:5}：提取最左边的 5 个字节：/dir1
${file:5:5}：提取第 5 个字节右边的连续 5 个字节：/dir2

判断参数&语句

r h e l 6

/etc/init.d/

read

-p 提示字符
-n 数字
-s 密文
-t 指定时间

算术操作（expr）　　——在bash中只能做整数的运算

+ 加
- 减
* 乘
/ 除（取整）
% 取余

$(())
$[]

expr $RANDOM % 10　　一个随机变量

let "v=2**16"
echo $v
65536

[root@i ~]# let "v=((((((1 + 2) * 10)) / 3))) * 2"

退出状态

在Linux系统中，每当命令执行完成后，系统都会返回一个退出状态。该退出状态用一整数值表示，用于判断命令运行正确与否。
若退出状态值为0，表示命令运行成功
若退出状态值不为0时，则表示命令运行失败
最后一次执行的命令的退出状态值被保存在内置变量"$?"中，所以可以通过echo语句进行测试命令是否运行成功
0 表示运行成功，程序执行未遇到任何问题
1~125 表示运行失败，脚本命令、系统命令错误或参数传递错误
126 找到了该命令但无法执行
>128 命令被系统强制结束

计算小数：

bc (计算时间最慢)

echo 0.33 + 0.44 |bc

time echo 1+1 |bc

time $((1 + 1))

time 一条命令的执行时间

条件表达式：
test命令
用途：测试特定的表达式是否成立，当条件成立时，命令执行后的返回值$?为0，否则为其他数值
格式：　test　条件表达式
　　[　条件表达式　]

test可以测试表示有哪些：
　　1、文件状态
　　2、字符串的比对
　　3、整数的比对
　　4、多条件组合（|| && !）(-a -o !)

#man test

-a	两个条件都成立	并
&&	前一条成立执行下一条	
-o	有一个条件成立	或
||	前一条不成功才执行下一条	
!	非　取反	

比较数字

-eq	equal等于
-ne	not equal不等于
-lt	lesser than小于
-le	lesser equal小于等于
-gt	granter then大于
-ge	granter equal大于等于

比较字符

-z string　　字符串true，如果字符串是空的。

-n string　　字符串true，如果字符串不为空。

string1 == string2

string1 != string2

string1 < string2

string1 > string2

比较文件的类型

man test

-e file　　文件的真实文件是否存在。

-s 文件存在切不为空

-a file 文件是否存在

-b file 文件存在，而且是设备文件

-c file 文件存在，而且字符文件

-d file 文件存在，而且类型是目录

-f file 文件存在，而且常规文件

-g file 文件存在，而且设置强制位

find / -perm -2000 查找系统中哪些文件拥有强制位

-h file 文件存在，而且是硬链接文件 ln-d 1.txt 2.txt

-L file 文件存在而且是软链接 ln -s 1.txt 2.txt

True if file exists and is a symbolic link.

-k file 拥有粘滞位目录

-p file 测试管道文件

find / -type p 查找系统中有哪些管道文件

-r file 文件存在，而且当前用户对此拥有读的权限

-s file 文件存在，而且文件内容非空

-u file 拥有suid的文件

find / -perm -4000
[-u /bin/mount] && echo YES

-w file 测试文件写权限

-x file 测试是否拥有执行权限

-O file 测试文件拥有者是否是当前用户

-G file 对此文件属组（主组）

-S file 测试对象是否是socket 软连接 ln -S

[-S /dev/log] && echo YES

file1 -nt file2 ——file1是否比file2新 -n 比较哪个文件新（和t在一起）

file1 -ot file2 ——测试filt1是否老于file2

file1 -ef file2 --可以用于测试两个文件是否是硬链接的关系

判断语句if

简单if结构

简单的if结构是：

```
if    expression  [  条件  ]
then
    command
    command
    ...
fi
```

在使用这种简单if结构时，要特别注意测试条件后如果没有“；”，则then语句要换行，否则会产生不必要的错误。如果if和then可以处于同一行，则必须用“;”

if/else结构

命令是双向选择语句，当用户执行脚本时如果不满足if后的表达式也会执行else后的命令，所以有很好的交互性。其结构为：

```
if      expression1
then
        command
        ...
        command
else
        command
        ...
        command
fi
```

if/elif/else结构

if/elif/else结构针对某一事件的多种情况进行处理。通常表现为“如果满足某种条件，则进行某种处理，否则接着判断另一个条件，直到找到满足的条件，然后执行相应的处理”。其语法格式为：

```
if      expression1
  then
      command

  elif  expression2
  then
      command
elif    expressionN
then
  command

else
  command
fi
```

case结构

和if/elif/else结构一样，case结构同样可用于多分支选择语句，常用来根据表达式的值选择要执行的语句，该命令的一般格式为：

```
case Variable  in
value1)

command;;

*)
    command;;
```

esac

for循环

列表for循环

列表for循环语句用于将一组命令执行已知的次数，下面给出了for循环语句的基本格式：

```
for variable in {list}
do
    command
    command
    …
done
```

其中do和done之间的命令称为循环体，执行次数和list列表中常数或字符串的个数相同。
当执行for循环时，首先将in后的list列表的第一个常数或字符串赋值给循环变量，然后执行循环体；接着将list列表中的第二个常数或字符串赋值给循环变量，再次执行循环体，这个过程将一直持续到list列表中无其他的常数或字符串，然后执行done命令后的命令序列。

for i in {N M K}

N是起始点
K是步长
M是结束点

不带列表for循环

for i in {1 10 2}

不带列表的for循环执行时由用户指定参数和参数的个数，下面给出了不带列表的for循环的基本格式：

```
for variable
do
    command
    command
    …
done
```

其中do和done之间的命令称为循环体，Shell会自动的将命令行键入的所有参数依次组织成列表，每次将一个命令行键入的参数显示给用户，直至所有的命令行中的参数都显示给用户。

类C风格的for循环（1）

类C风格的for循环也可被称为计次循环，一般用于循环次数已知的情况，下面给出了类C风格的for循环的语法格式：

```
for(( expr1; expr2; expr3 ))
do
    command
    command
    …
done
```

类C风格的for循环（2）

对类C风格的for循环结构的解释：
其中表达式expr1为循环变量赋初值的语句
表达式expr2决定是否进行循环的表达式，当判断expr2退出状态为0执行do和done之间的循环体，当退出状态为非0时将退出for循环执行done后的命令

表达式expr3用于改变循环变量的语句
类C风格的for循环结构中，循环体也是一个块语句，要么是单条命令，要么是多条命令，但必须包裹在do和done之间。

演示例8-10和8-11

for 循环语句

```
for  var    in  1 2 3 4 5
                    do
                            command
                    done

for (( var=1 ; var < 6 ; var++ ))
                    do
                            command
                    done

for (( var=5; var >0 ; var-- ))
                    do
                            command
                    done
```

实例1：批量新建10个用户
1、所有用户的初始化密码是uplooking
2、所有用户每30天需要改一次密码，如果不改密码，在密码过期后三天账号不可用
3、所有的账号第一登录时都需要强制修改密码

```
for i in {1..10}
do
        useradd login$i
        echo "uplooking" | passwd --stdin login$i
        chage -M 30 -I 3 login$i
        chage -d 0 login$i
done
```

实例2：根据花名册来新建用户

```
# cat user.txt                  --花名册
工号    姓名    所在部门      入职时间       离职时间       薪水
001     zhang3  rs            2012-1-1                      5000
002     li04        cw          2000-1-1       2005-12-12     4000
003     wang5   sc            2010-10-10                        3500
```

脚本如下：

```
#!/bin/bash
users=$(awk 'NR>1 {print $3"_"$2}' user.txt)

for i in $users
do
        useradd $i
        echo "uplooking" | passwd --stdin $i
        chage -M 30 -I 3 $i
        chage -d 0 $i

done
```

```
# for a in $(awk -F: '{print $1}' /etc/passwd | tail -20); do userdel -r $a; done


vim for1.sh
A=$1
B=${A:=10}
for  (( i=1;i<=$B;i++ ))
do
   for (( j=1 ;j<=$B;j++))
   do
      echo $i $j
      a=$((a+1))

    done

 done
 echo $a

 sh for1.sh 10
```

实例3：批量设置磁盘配额（普通用户和老板的配额需要不一样)

```
useradd u01
useradd boss

#!/bin/bash

users="u01 u02 boss"
block_soft=$((1*1024*1024))
block_hard=$((2*1024*1024))
inode_soft=0
inode_hard=0

mountpoint=/home

for i in $users
do
        if [ "$i" = boss ]
        then
                setquota -u $i 20000000 30000000 $inode_soft $inode_hard $mountpoint
        else
                setquota -u $i $block_soft $block_hard $inode_soft $inode_hard $mountpoint
        fi
done
```

批量删除可以登录系统的用户除了root：

```
#!/bin/bash
del_users=$(awk -F':' '/bash$/ && !/root/ {print $1}' /etc/passwd)
for i in $del_users
do
            userdel -r $i
            rm -rf /home/$i
            rm -rf /var/spool/mail/$i
done
```

1)通过脚本完成启动磁盘配额的功能（自己完成）
2)批量新建samba用户（/sbin/nologin），samba用户初始密码为uplooking
3)为每个samba用户批量设置配额为：软:2G 硬为：3G

```
#!/bin/bash

block_soft=$((2*1024*1024))
block_hard=$((3*1024*1024))

inode_soft=0
inode_hard=0

mountpoint=/mnt

for i in {1..10}
do
#       useradd u0$i -s /sbin/nologin
        if [ "u0$i" = boss ]
        then
                setquota -u u0$i 20000000 30000000 $inode_soft $inode_hard $mountpoint
        else
                setquota -u u0$i $block_soft $block_hard $inode_soft $inode_hard $mountpoint
        fi

        (echo uplooking;echo uplooking) | smbpasswd -s -a u0$i

done
```

1)使用for求出1+2+3+..+100的总和

```
#!/bin/bash

sum=0
for i in {1..100}
do
            sum=$(($sum+$i))
done

echo $sum
```

2)求偶数之和

```
#!/bin/bash

sum=0
for ((i=1;i<=100;i+=2))                                    \\$(seq 1 2 100 ) {1..100..2}
do
            sum=$(($sum+$i+1))
done

echo $sum
```

while循环

while循环语句也称前测试循环语句，它的循环重复执行次数，是利用一个条件来控制是否继续重复执行这个语句。while语句与for循环语句相比，无论是语法还是执行的流程，都比较简明易懂。while循环格式如下：

```
while expression
do
    command
    command
    …
done
```

while循环语句之所以命名为前测试循环，是因为它要先判断此循环的条件是否成立，然后才作重复执行的操作。也就是说，while循环语句执行过程是：先判断expression的退出状态，如果退出状态为0，则执行循环体，并且在执行完循环体后，进行下一次循环，否则退出循环执行done后的命令。
为了避免死循环，必须保证在循环体中包含循环出口条件，即存在expression的退出状态为非0的情况。

计数器控制的while循环

假定该种情形是在已经准确知道要输入的数据或字符串的数目，在这种情况下可采用计数器控制的while循环结构来处理。这种情形下while循环的格式如下所示：

```
counter = 1
while expression
do
    command
    …

    command
    …
done
```

vim while_exam5.sh

```
#while_exam5.sh
#!/bin/bash

echo "Please input the num:"
read num

signal=0

while [[ "$signal" != 1 ]]
do
  if [ "$num" -lt 4 ]
  then
      echo "Too small. Try again!"
      read num
  elif [ "$num" -gt 4 ]
  then
       echo "To high. Try again"
       read num
   else
       signal=1
       echo " Congratulation, you are right! "
  fi

done
```

结束标记控制的while循环

在Linux Shell编程中很多时候不知道读入数据的个数，但是可以设置一个特殊的数据值来结束while循环，该特殊数据值称为结束标记，其通过提示用户输入特殊字符或数字来操作。当用户输入该标记后结束while循环，执行done后的命令。在该情形下，while循环的形式如下所示：

```
read variable
while [[ "$variable" ！ = sentinel ]]
do
    read variable
done
```

实例8：

```
read -p "请输入：" num
while [ ！  "$num" = "exit" ]
do
   read  -p "请输入：" num
done
```

标志控制的while循环

标志控制的while循环使用用户输入的标志的值来控制循环的结束，这样避免了用户不知到循环结束标记的麻烦。在该情形下，while循环的形式如下所示：

```
signal=0
while (( signal != 1 ))
do
        ...
        if expression
        then
            signal=1
        fi
        ...
  done
```

演示例8-19

vim while.sh

```
a=0
while [ 10 -gt $a ]
do

echo $a
sleep 1
a=$((a+1))
done
echo $a
```

命令行控制的while循环

有时需要使用命令行来指定输出参数和参数个数，这时用其他的三种形式的while循环是无法实现的，所以需要使用命令行控制的while循环。该形式下，while循环通常与shift结合起来使用，其中shift命令使位置变量下移一位（即$2代替$1，$3代替$2），并且使$#变量递减，当最后一个参数也显示给用户后，$#就会等于0，同时$*也等于空，下面是该情形下，while循环的形式为：

```
while [[ "$*" != "" ]]
do
    echo "$1"
```

```
                shift
        done
```

演示例8-20

while read line

vim whitl2.sh

```
while read line
do
    echo "line=$line"
    sleep 1
done < /etc/passwd
```

shift命令使位置变量下移一位（即$2代替$1，$3代替$2），并且使$#变量递减，当最后一个参数也显示给用户后，$#就会等于0，同时$*也等于空

vim shift.sh

```
#!/bin/bash

while [ $# -ne 0 ]
do
        echo $1
        shift
done
```

```
# chmod +x shift.sh
# sh -x ./shift.sh 1 2 3
+ '[' 3 -ne 0 ']'
+ echo 1
1
+ shift
+ '[' 2 -ne 0 ']'
+ echo 2
2
+ shift
+ '[' 1 -ne 0 ']'
+ echo 3
3
+ shift
+ '[' 0 -ne 0 ']'
==========
# vim user_passwd.txt
uu01    123
uu02    456
uu03    789
uu04    012
uu05    345
```

```
# vim while_useradd_shift.sh                    --每次给脚本传递两个变量，然后再把变量迁移掉
#!/bin/bash


while [ $# -ne 0 ]
do
        useradd $1
        echo $2 | passwd --stdin $1
        shift 2
done
```

until循环

在执行while循环时，只要是expression的退出状态为0将一直执行循环体。until命令和while命令类似，但区别是until循环中expression的退出状态不为0将循环体一直执行下去，直到退出状态为0，下面给出了until循环的结构：

```
until expression
do
    command
    command
    …
done
```

演示例8-21

```
while true
do
echo A
sleep 1
done

until false
do
echo A
sleep 1
done

vim until_exam1.sh
#until_exam1.sh
#!/bin/bash

i=0

until [[ "$i" -gt 5 ]]
do
   let "square=i*i"
   echo "$i * $i = $square"
   let "i++"
done
```

++++++++++++++++++++++++++

```
vim until_exam2.sh
#while_exam2.sh
#!/bin/bash

echo "Please input the num(1-10) "
```

```
read num

until [[ "$num" = 4 ]]
do
   if [ "$num" -lt 4 ]
   then
        echo "Too small. Try again!"
        read num
   elif [ "$num" -gt 4 ]
   then
         echo "To high. Try again"
         read num
    else
       exit 0
   fi
done

echo "Congratulation, you are right! "
```

break循环控制符

break语句可以应用在for、while和until循环语句中，用于强行退出循环，也就是忽略循环体中任何其他语句和循环条件的限制。

```
vim break_exam1.sh
#break_exam1.sh
#!/bin/bash


for (( i=1; i <= 9; i++))
do
       for (( j=1; j<= i; j++ ))
       do


           let "temp=i*j"

           if [ "$temp" -eq 7 ]
           then
               break
           fi

           echo -n "$i*$j=$temp  "
       done
       echo ""
  done
```

++

```
vim break_exam2.sh
#break_exam2.sh
#!/bin/bash


for (( i=1; i <= 9; i++))
do
      for (( j=1; j<= i; j++ ))
      do
          let "temp=i*j"
          echo -n "$i*$j=$temp  "
      done

      if [ "$i" -eq 6 ]
      then
          break
      fi

      echo ""
  done
```

continue循环控制符

continue循环控制符应用在for、while和until语句中，用于让脚本跳过其后面的语句，执行下一次循环。

```
vim continue_exam1.sh
#continue_exam1.sh
#!/bin/bash


for (( i=1; i <= 9; i++))
do
    if [ "$i" -eq 7 -o "$i" -eq 6 ]
    then
        continue
    fi


    for (( j=1; j<= i; j++ ))
    do

        let "temp=i*j"
        echo -n "$i*$j=$temp  "
    done


    echo ""
done
```

嵌套循环
一个循环体内又包含另一个完整的循环结构，称为循环的嵌套。在外部循环的每次执行过程中都会触发内部循环，直至内部完成一次循环，才接着执行下一次的外部循环。for循环、while循环和until循环可以相互嵌套。

循环嵌套

打印出
1
12
123
1234
12345

```
#!/bin/bash
for (( j=1;j<=5;j++))
do
    for ((m=1;m<=j;m++))
    do
        echo -n $m
    done

 echo ""
done
```

5
54
543
5432
54321

```
for ((i=5;i>=1;i--))
    do

        for ((j=5;j>=i;j--))
        do
            echo -n "$j"

        done
            echo " "
done
```

*
**
***
****
*****
******

```
#!/bin/bash
for (( j=1;j<=6;j++))
do
    for ((m=1;m<=j;m++))
    do
        echo -n '*'
    done

 echo ""
done
```

select结构

Select是bash的扩展结构，用于交互式菜单显示，用于从一组不同的值中进行选择，功能有点类似于case结构，但其交互性要比case好的多，其基本结构为：

```
select variable in {list}
do
    command
    ...
     break
done
```

```
vim select_exam2.sh
#select_exam2.sh
#!/bin/bash

echo "What is your favourite color? "

select color
do
   break
done

echo "You have selected $color"
```

```
vim os.sh
PS3="please choose what operation system you are using:"      --PS3是select命令用来加入提示字符串的符号，（默认会使用#?)

echo
select os in  xp   vista   windows7 linux unix
do
       echo
       echo "your operation system is $os"
       echo
       break         --这里不加break的话，就会一直循环让你选择
done
```

函数是一串命令的集合，函数可以把大的命令集合分解成若干较小的任务。编程人员可以基于函数进一步的构造更复杂的Shell 程序，而不需要重复编写相同的代码。下面给出了Linux Shell中函数的基本形式

```
function_name()
{
    command1
    command2
    ...
    commandN
}
```

```
vim function1.sh

#function1.sh
#!/bin/bash

hello()
{
   echo "Hello, today is `date`"
}

echo "Now going to run function hello()"
hello

echo "end the function hello()"
```

判断de 优先级

&& > || > !

-a > -o > !

; || &&
以上三个都可以用来分割命令，区别在于：

;	不管前的	命令	是否执行成功都执行
||	前面的	命令或者条件	为假才执行后面的内容
&&	前面的	命令或者条件	为真才执行后面的内容

多个条件判断的时候先看优先级 然后条件的真假 最后看前面的命令有没有执行成功
注意：如果多重条件判断的化，那么需要考虑优先级

man test 文件属性

-z STRING	STRING的长度为零
STRING1 = STRING2	字符串相等
STRING1 ! = STRING2	字符串不相等
INTEGER1 -eq INTEGER2	INTEGER1等于INTEGER2
INTEGER1 -ge INTEGER2	INTEGER1大于或等于INTEGER2
INTEGER1 -gt INTEGER2	INTEGER1大于INTEGER2
INTEGER1 -LE INTEGER2	INTEGER1小于等于INTEGER2
INTEGER1 -lt INTEGER2	INTEGER1小于INTEGER2
INTEGER1 -ne INTEGER2	INTEGER1不等于INTEGER2
FILE1 -ef FILE2	FILE1和FILE2具有相同的设备和inode编号
FILE1 -nt FILE2	FILE1比FILE2更新（修改日期）
FILE1 -ot FILE2	FILE1比FILE2更旧（修改日期）
-b文件	FILE存在并且是特殊的块
-c文件	文件存在且是特殊字符
-d FILE	FILE存在，是一个目录
- 文件	文件已存在

-f文件　　FILE存在并且是常规文件

-g文件　　FILE存在且为set-group-ID

-G文件　　FILE存在且由有效组ID所有

-h文件　　FILE存在，是一个符号链接硬（与-L相同）

-k FILE　　FILE存在并且其粘性位设置　　o+t　目录 1777

-L文件　　FILE存在，是一个符号链接软（与-h相同）

-O文件　　FILE存在且由有效的用户ID拥有

-p文件　　FILE存在，是一个命名管道

-r FILE　　FILE存在并且授予读取权限

-s文件　　FILE存在且大小大于零

-S文件　　FILE存在并且是一个套接字

-t　　FD文件描述符FD在终端上打开

-u文件　　FILE存在，其设置用户ID位置1

-w文件　　FILE存在并且授予写入权限

-x FILE　　FILE存在且执行（或搜索）权限被授予

if 判断语句

```
if  [   ] ;then
        command
elif [   ] ;then
         command
else
        command
fi
```

case 语句

```
read -p “请输入xxxx：”  i
菜单语句
case  i in
            )            -----匹配的条件
                ;;       ------执行匹配条件的命令
             )
                ;;
            *)           -----除了上面的情况外
                 ;;
```

esac

2） case 关键字的用法：

```
case  $var   in
   条件1）
      条件1成立时执行命令;;
   条件2）
      条件2成立时执行命令;;
   条件3）
      条件3成立时执行命令;;
    ...)
          ....;;
   *)
      其他情况下执行的命令;;
esac
```

示例：

```
#!/bin/bash
read -p "Username: " u
read -s -p "Password: " p
echo
up=$u:$p

case $up in
       root:123456)
          echo "administrator."
          ;;
       chuyue:654321)
          echo "welcome user."
          ;;
       zhangsan:abcefg)
          echo "$u"
          ;;
       *)
          echo "who are you ?"
          ;;
esac
```

练习：
1）自行用shell脚本编写一个命令cmd.sh，当你输入cmd.sh -t的时候，显示当前的时间，-c 显示当前月的日历（cal），-i 显示IP地址，-v 显示命令的版本; -h, 显示命令的帮助文档（cat << EOF）。

2） 增加点难度，多个选项混合使用呢。（可能要用到循环，后面再写）

for

{1..10..2} 1-10隔两位取数 1 3 5 7 9
{0..10..2}   0 2 4 6 8
`seq 100` seq 序列命令   1..100

类c风格　类C风格的for循环也可被称为计次循环　类C风格的for循环结构中，循环体也是一个块语句，要

么是单条命令，要么是多条命令

```
for ((x=0;x<=5;x++))
        do
                echo $x
done
```

列表循环　列表for循环语句用于将一组命令执行已知的次数

```
for i  in {1..10}
        do
                echo $i
 done

 for i in `seq 100`

 for i in $(cat /etc/passwd)
        do
                echo $i
 done
```

非列表循环　　不带列表的for循环执行时由用户指定参数和参数的个数

for i in 可以直接 多个变量

找目录下的空文件

```
dir=/shells
cd $dir
for i in *
        do
        [ ! -s $i ]  && rm -rf $i
done
```

i属于该目录下的所有文件中的一个

-s 文件存在且不为空　！ 取反　文件存在但为空

until

until [ ]　当条件为假的时候执行 为真的时候跳出循环

```
        do
             command
 done

 false 假
 死循环
 until false
        do
                echo A
           sellp  10
 done
```

while

while [ ]　　当条件为真的时候执行循环 为假的时候退出循环

```
        do
```

```
        command
done

ture  :   都是真的意思
while  true  或者 :
      do
          echo  A
        sellp   10
done

file=/etc/passwd              read   行读取
while  read  files          从/etc/passwd中读取一行 内容给 files 这个变量
          do
                  echo $files
done < $file
```

while 可以一次性定义多个变量

```
while read  ip  user   passwd     以空格分割
      do

done <1.txt
 done < 1.txt
```

select 选择

```
a=1
a=2
a=3

select i in $(a[*])                  当i输入数组内
      do
          echo   $i
done
```

功能在脚本中把需要调用的函数放在数组里面 然后用select 选择执行要执行的函数

expect

自动应答
格式

```
      expect << END                                          ---------------开
始
          spawn  ssh root@ip                                 ---------------执行的命令
            expect {
                  "*(yes/no)*" {  send "yes\r";exp_continue}      -------捕捉的最后的关键字
                  "password:"  { send "$passwd\r";exp_continue}   -----捕捉关键字
```

eof{exit}　　--------结束
expect
}
END　　-------结束标签

脚本命令

continue　当条件满足时执行到这条命令的时候不管上一次命令继续执行下面的内容
跳过满足条件的循环 继续执行下一个循环

break　　中断当前的循环 执行下面的内容

exit　　退出循环

clear　　清屏

$RANDOM 随机变量

$RANDOM 系统自带的随机变量

随机数：
默认是0~32767。使用set |grep RANDOM　　查看上一次产生的随机数

产生0~1之间的随机数
echo $[$RANDOM%2]　0 1　　最小是0　　最大是100

产生0~2之间的随机数　0 1 2　　最小是0　　最大是2
echo $[$RANDOM%3]

产生0~3之间的随机数　0 1 2 3　　最小是0　　最大是3
echo $[$RANDOM%4]
。。。。
产生0~9内的随机数　0 1 2 3 4 5 6 7 8 9　　最小是0　　最大是9
echo $[$RANDOM%10]

产生0~100内的随机数　　最小是0　　最大是 100
echo $[$RANDOM%101]

产生50-100之内的随机数
echo $[$RANDOM%51+50]

产生三位数的随机数
echo $[$RANDOM%900+100]

随机数取余　　最小永远是0，最大是余数n（n-1 ） 总共有n个数
确保产生随机数的位数　个位数　直接 %10
十位数　%90+10　　0　0+10=10　　89　89+10=99
百位数　%900+100　　0　0+100=100　　899　899+100=999

千位数 %9000+1000 0 0+1000=1000 8999 8999 +1000=9999

script

```
#!/bin/bash
#随机数RANDOM 最小 最大
no(){
for i in `seq 1000`                                  ----------------产生1-1000的序列  i的取值范围
        do
        n1=$(echo $[RANDOM%10])                 ----------------产生一位数（个位数）
        n2=$(echo $[RANDOM%10])
        n3=$(echo $[RANDOM%10])
        n4=$(echo $[RANDOM%10])
        n5=$(echo $[RANDOM%10])
        n6=$(echo $[RANDOM%10])
        n7=$(echo $[RANDOM%10])
        n8=$(echo $[RANDOM%10])
        echo "139${n1}${n2}${n3}${n4}${n5}${n6}${n7}${n8}" >> phone.txt
done
}
no (){
max=1000
while true                                           ---------------------------和上面for循环的出的结果一样
        do
       now=$(cat phone.txt |wc -l)                ---------------------------查看文本统计数量并赋值给变量
        if [ $now -lt $max ];then
                n1=$(echo $[RANDOM%10])
                n2=$(echo $[RANDOM%10])
                n3=$(echo $[RANDOM%10])
                n4=$(echo $[RANDOM%10])
                n5=$(echo $[RANDOM%10])
                n6=$(echo $[RANDOM%10])
                n7=$(echo $[RANDOM%10])
                n8=$(echo $[RANDOM%10])
        echo  "139${n1}${n2}${n3}${n4}${n5}${n6}${n7}${n8}" >> phone.txt
        else
                exit
        fi
done
}

抽奖
 一  从文本中抽取5个
       file=/shells/shells1/phone.txt
       line=$(cat $file |wc -l)
       luckline=$(echo $[$RANDOM%$line+1])
       luckphone=$(head -$luckline $file |tail -1)
       i=1
       while [ $i -le 5 ]
       do
       echo "幸运观众是:139****${luckphone:7:4}"
       let i++
```

done
二 抽过一次的把它从文本中删除

fun _函 数

函数调用：
source function.sh 或者. function.sh
1、在命令行进行调用(当前终端当前进程生效)
2、写到用户环境变量里
profile 、~/.bash_profile
1》针对所有用户和进程生效
vim /etc/profile

2> 针对某个用户和紫禁城生效
vim ~/.bash_profile

3、直接写到脚本里

/etc/profile	系统和用户的全局的环境变量
/etc/bashrc	函数和别名的全局配置（bash）
~/.bash_profile	用户局部的环境变量
~/.bashrc	用户局部的bash信息
~/.bash_logout	用户退出当前bash时读取的文件

/etc/profile——>~/.bash_profile——>~/.bashrc——>/etc/bashrc——>~/.bash_logout

注意：
函数名不要和当前系统的命令或者符号相同

在一个脚本 里面写好的的函数 假如是/shells/auto_init.sh

那么我在 另外的脚本上要调用函数 如vsftp.sh
#!/bin/bash
source /shells/auto_init.sh 调用脚本
server_repo 调用参数