今天任务

1.shell介绍

2.shell的运行环境和方式

3.学习变量,read命令,数组,运算,测试

4.学习条件控制,循环,方法

5.介绍awk,sed命令

6.介绍定时器

教学目标

1.掌握shell的基本语法

2.能够进行简单的shell脚本的编写

3.了解awk,sed命令

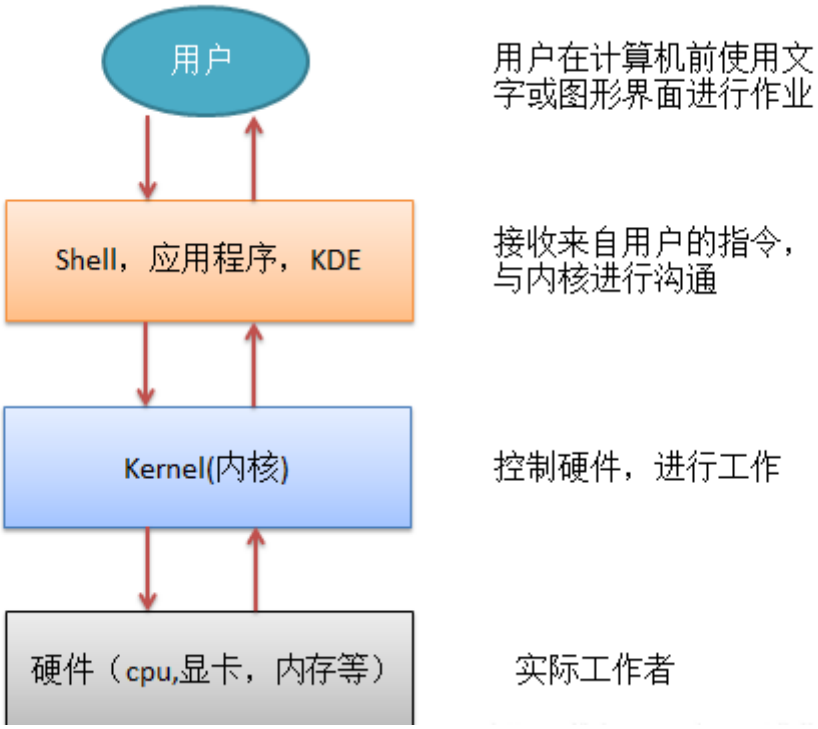
4.能够设置简单的定时器

第一章：shell编程

1.1 shell的概念介绍

1.1.1 命令解释器

Shell是命令解释器(command interpreter)，是Unix操作系统的用户接口，程序从用户接口得到输入信息，shell将用户程序及其输入翻译成操作系统内核（kernel）能够识别的指令，并且操作系统内核执行完将返回的输出通过shell再呈现给用户，下图所示用户、shell和操作系统的关系：



一个系统可以存在多个shell，可以通过cat /etc/shells命令查看系统中安装的shell。

```
[root@qianfeng01 ~]# cat /etc/shells
/bin/sh
/bin/bash
/usr/bin/sh
/usr/bin/bash
```

操作系统内核（kernel）与shell是独立的套件，而且都可被替换；不同的操作系统使用不同的shell； 同一个kernel之上可以使用不同的shell。 也可以查看当前shell环境是哪种：echo $SHELL

1.1.2 shell脚本

Shell也是一门编程语言，即shell脚本。在此脚本中，我们可以使用一些编程语法来进行一些任务操作。如：变量、类型、分支结构、循环结构、数组、函数等语法。 在shell脚本里，必须指定一种shell命令解释器。

1.2 shell编程规范

1.2.1 脚本文件的结构

```
1. 文件的扩展名必须是.sh
2. 文件的首行必须使用#!       指定script的运行shell环境(即脚本解释器)
   如：#!/bin/bash
3. 脚本里的行注释符号为#
4. 指令、选项、参数之间即使有多个空格仍会被视为一个空格。
5. tab键形成的空白也被视为一个空格键
6. 空白行会被忽略
```

1.2.2 脚本文件的执行

第一种执行方式： 使用bash程序来调用执行，只需要有读(r)权限即可

```
[hadoop@master ~]$  sh  *.sh
或者
[hadoop@master ~]$  bash   *.sh
```

第二种执行方式：直接写script，必须要有rx权限才行

```
[hadoop@master ~]$  ./*.sh
绝对路径写法：  /home/hadoop/*.sh
相对路径写法：  ./*.sh
```

第三种：借助变量PATH功能：

```
将*.sh放入~/bin目录下，因为PATH里拼接了~/bin目录。
注意：~/bin目录必须自行创建
```

1.2.3 shell中的变量用法

```
1. 变量的命名规则

   - 命名只能使用英文字母，数字和下划线。首个字符不能以数字开头。
   - 字母习惯使用大写。
```

```
    - 中间不能有空格。
    - 不能使用标点符号。
    - 不能使用bash里的关键字（可用help命令查看保留关键字）

2. 变量的使用规则

    - 直接定义变量名称，没有类型需要强调（类似于数学中:x=1,y=2,z=x+y）
    - 赋值时，"="前后不能有空格
    - 命令的执行结果赋值给变量时，使用反单引号        如：TIME=`date`
    - 调用变量时，必须使用$        格式： $变量名    或    ${变量名}
```

1.3 变量分类

Linux Shell中的变量可以分为三种变量：局部变量，环境变量，特殊变量。可以通过set命令查看系统中存在的所有变量

```
局部变量：也就是用户自定义的变量，在脚本中或命令中定义，仅在当前shell实例中有效，其他shell启动的程序不能访问局部变量。
环境变量：保存和系统操作环境相关的数据。\$HOME、\$PWD、\$SHELL、$USER等等
特殊变量：
    一种是位置参数变量：主要用来向脚本中传递参数或数据，变量名不能自定义，变量作用固定。
    一种是预定义变量：是Bash中已经定义好的变量，变量名不能自定义，变量作用也是固定的。
```

1.3.1 局部变量

用户自定义的变量由字母或下划线开头，由字母，数字或下划线序列组成，并且大小写字母意义不同，变量名长度没有限制。

1) 设置变量

习惯上用大写字母来命名变量。变量名以字母表示的字符开头，不能用数字。

2) 变量调用

在使用变量时，要在变量名前加上前缀“$”.

使用echo 命令查看变量值

eg: echo $A

3) 变量赋值

第一种:定义时赋值

变量 = 值

等号两侧不能有空格

eg: STR="hello world"

eg: A=9

第二种:将一个命令的执行结果赋给变量

eg: A=`ls -la` 反引号，运行里面的命令，并把结果返回给变量A

eg: A=$(ls -la) 等价于反引号

eg: aa=$((4+5))

eg: bb=`expr 4 + 5 `

第三种:将一个变量赋给另一个变量

eg : A=$STR

4) 变量叠加

eg: aa=123

eg: cc="$aa"456

eg: dd=${aa}789

单引号和双引号的区别

现象：单引号里的内容会全部输出，而双引号里的内容会有变化

原因：单引号会将所有特殊字符脱意

eg: NUM=10

SUM="$NUM hehe" echo $SUM 输出10 hehe

SUM2='$NUM hehe' echo $SUM2 输出$NUM hehe

5) 列出所有的变量

set

6) 删除变量

eg: unset NAME

eg: unset A 撤销变量 A

eg: readonly B=2 声明静态的变量 B=2 ， 不能 unset

```
[root@node1 ~]# readonly b=cc
[root@node1 ~]# unset b
-bash: unset: b: cannot unset: readonly variable
[root@node1 ~]#
```

用户自定义的变量，作用域为当前的shell环境。

1.3.2 环境变量

用户自定义变量只在当前的shell中生效，而环境变量会在当前shell和其所有子shell中生效。如果把环境变量写入相应的配置文件，那么这个环境变量就会在所有的shell中生效。

export 变量名=变量值 声明变量

作用域：当前shell以及所有的子shell

1.3.3 位置参数变量

$n | n为数字，**$0**代表命令本身，**$1-$9**代表第一到第9个参数，十以上的参数需要用大括号包含，如**${10}**。
$* | 代表命令行中所有的参数，把所有的参数看成一个整体。以"$1 $2 … $n"的形式输出所有参数
$@ | 代表命令行中的所有参数，把每个参数区分对待。以"$1" "$2" … "$n" 的形式输出所有参数
$# | 参数的个数

shift指令：参数左移，每执行一次，参数序列顺次左移一个位置，$# 的值减1，用于分别处理每个参数，移出去的参数不再可用

$* 和 $@的区别

$* 和 $@ 都表示传递给函数或脚本的所有参数，不被双引号" "包含时，都以"$1" "$2" … "$n" 的形式输出所有参数

当它们被双引号" "包含时，"$*" 会将所有的参数作为一个整体，以"$1 $2 … $n"的形式输出所有参数；"$@" 会将各个参数分开，以"$1" "$2" … "$n" 的形式输出所有参数

shell脚本中执行测试：

```
echo "test \$*"
for i in $*
do
 echo $i
done
echo "test \$@"
for i in $@
do
 echo $i
done

echo "test \"\$*\""
for i in "$*"
do
 echo $i
done
echo "test \"\$@\""
for i in "$@"
do
echo $i
done
```

输出结果：

```
[root@node1 ~]# sh test1.sh a b
test $*
a
b
test $@
a
b
test "$*"
a b
test "$@"
a
b
```

1.3.4 预定义变量

$?	执行上一个命令的返回值 执行成功，返回**0**，执行失败，返回非**0**（具体数字由命令决定）
$$	当前进程的进程号（PID），即当前脚本执行时生成的进程号
$!	后台运行的最后一个进程的进程号（PID），最近一个被放入后台执行的进程 &

测试$$:

编写shell脚本 vi pre.sh

pwd >/dev/null 执行

echo $$ 执行

测试$!:

执行下面的语句

ls /etc >/dev/null & 执行

echo $! 执行

测试$?:

./pre.sh ; echo $?

分析：这里的意思是一次顺序执行两个命令

如果pre.sh可以执行，$?会返回0.否则返回非零的一个数字

1.4 read命令

1.4.1 命令说明

read [选项] 值

read -p(提示语句) -n(字符个数) -t(等待时间，单位为秒) –s(隐藏输入)

1.4.2 实例

read –t 30 –p “please input your name: ” NAME

echo $NAME

read –s –p “please input your age : ” AGE

echo $AGE 注意：如果隐藏输入，这里的结果是看不到的

read –n 1 –p “please input your sex [M/F]: ” GENDER

echo $GENDER

注意：

1.在输入时，如果输错了要删除要执行control+delete

2.不要输入中文

3.NAME与"之间要有空格

1.5 运算

1.5.1 expr

格式 :expr m + n 或$((m+n)) 注意expr与运算符和变量间要有空格

sum=$((m+n)) 中=与$之间没有空格

expr命令：对整数型变量进行算术运算

(注意：运算符前后必须要有空格)

expr 3 + 5 expr 3 – 5

echo `expr 10 / 3`

10/3的结果为3，因为是取整

expr 3 * 10

\ 是转义符

1.5.2 实例

计算（2 +3 ）×4 的值

1 .分步计算

```
S= `expr 2 + 3`

expr $S  \\* 4
```

2.一步完成计算

```
expr  \`expr 2 + 3 \` \\* 4
S=`expr  \`  expr 2 + 3  \`  \\* 4`
echo $S
```

或

echo $(((2 + 3) * 4))

1.5.3 $()与${}

$()与${}的区别

$( )的用途和反引号``一样，用来表示优先执行的命令

eg: echo $(ls /root)

${ } 就是取变量了

eg：echo ${PATH}

$((运算内容)) 适用于数值运算

eg: echo $((3+1*4))

1.6 字符串

1.6.1 字符串的基本用法

```
- 字符串不能单独，必须要配合变量。
- 字符串可以使用单引号['  ']，也可以使用双引号[" "]，也可以不用引号
- 单引号内的任何字符没有任何意义，都会原样输出，
- 单引号内使用变量是无效的，单引号内不能出现单引号
- 双引号内可以使用变量
- 双引号内可以使用转义字符
- 在字符串拼接操作时，我们可以进行无缝拼接，或者是在双引号里使用变量
```

1.6.2 字符串的长度

```
可以使用${#variable} 或者 expr length "${variable}"。因为expr是指令，所以别忘记使用反单引号``或者是$()

直接看案例：

]vim test3.sh
#!/bin/bash
var='welcome to china'
length1=${#var}
length2=$(expr length "${var}")  <==$()写法
length3=`expr length "$var"`     <==反单引号写法
```

1.7 shell数组

1.7.1 Array的使用规则

```
- 在/bin/bash这个shell中，只有一维数组的概念，并且不限定数组的长度。
- 数组的元素下标是从0开始的，
- 获取数组的元素要使用下标
- 下标使用不当，会报错
```

1.7.2 Array的定义

```
定义格式： variable=(值1  值2    …  值n)
注意：元素之间除了使用空格作为分隔符，还可以使用换行符。
```

或者

```
name[0]=值1
name[1]=值2
…
name[n]=值n
```

1.7.3 读取数组

```
${variable[index]}： 读取index索引上的元素
${variable[*]}或者${variable[@]}：读取所有元素
${#variable[*]}或者${#variable[@]} ： 读取数组的长度
```

案例：

```
[michael@master ~]$ vim test3.sh
#!/bin/bash
citys=(cc sh bj sd hlj)
hobby[0]=book
hobby[1]=film
hobby[2]=music
echo ${citys[0]}      <==cc
echo ${hobby[*]}      <==book film music
echo ${#hobby[@]}     <==3
```

1.8 test测试命令

文件类型检测	
test -e filename	检测filename是否存在
test -d filename	检测filename是不是目录
test -f filename	检测filename是不是普通文件
test -L filename	检测filename是不是软链接文件
文件属性检测	
test -r filename	检测filename是否有可读权限
test -w filename	检测filename是否有可写权限
test -x filename	检测filename是否有执行权限
两个文件的比较	
test file1 -nt file2	检测 file1 是否比 file2 新
test file1 -ot file2	检测 file1 是否比 file2 旧
test file1 -ef file2	检测 file1和 file2 是否为同一文件
两个整数的比较	
test n1 -eq n2	检测 n1 与 n2 是否相等
test n1 -ne n2	检测n1 与 n2 是否不等
test n1 -gt n2	检测n1 是否大于 n2 （greater than）
test n1 -lt n2	检测n1 是否小于 n2 （less than）
test n1 -ge n2	检测n1 是否大于等于n2
test n1 -le n2	检测n1 是否小于等于n2
字符串的数据	
test -z string	检测string 是否为空
test -n string	检测string 是否不为空 -n 也可以省略
test string1 == string2	检测string1与string2是否相等
test string1 != string2	检测string1与string2 是否不想等
多条件连接 test -r filename -a -x filename	
-a	and，两个检测都成立，返回true
-o	or，只要有一个检测成立，就返回true
!	非

- 通常test命令不单独使用，而是与if语句连用，或者是放在循环结构中。
- 判断符号[] 除了好用的test外，我们还可以使用中括号来进行检测条件是否成立。

  举例说明

```
[  -r  filename ]    :   检测filename是否有可读权限
[  -f  filename -a  -r  filename ]   :   检测filename是不是普通文件并且有可读权限
```

1.9 条件控制

1.9.1 if 条件语句-单分支

if/else命令

单分支if条件语句

```
if [ 条件判断式 ]
then
    程序
fi
```

或者

```
if [ 条件判断式 ] ; then
    程序
fi
```

eg:

```
!/bin/sh
if  [ -x  /etc/rc.d/init.d/httpd ]
then
    /etc/rc.d/init.d/httpd restart
fi
```

单分支条件语句需要注意几个点

```
if语句使用fi结尾，和一般语言使用大括号结尾不同。

[ 条件判断式 ] 就是使用test命令判断，所以中括号和条件判断式之间必须有空格

then后面跟符号条件之后执行的程序，可以放在[]之后，用“;”分割，也可以换行写入，就不需要”; "了。

if与中括号之间必须要有空格
```

1.9.2 if 条件语句-多分支

```
 if [ 条件判断式1]
 then
     当条件判断式1成立时，执行程序1
 elif  [ 条件判断式2 ]
 then
          当条件判断式2成立时，执行程序2
     ...省略更多条件
 else
     当所有条件都不成立时，最后执行此程序
 fi
```

eg1:

```
#!/bin/bash
read -p "please input your name:"  NAME
echo  $NAME
if [ “$NAME” == root ]
then
    echo "hello ${NAME},  welcome !"
elif [ $NAME == tom ]
then
        echo "hello ${NAME},  welcome !"
else
          echo "SB, get out here !"
fi
```

eg2:

编写一个坐车脚本

要求:脚本:home.sh,从外面传入一个参数,根据参数判断1.坐飞机 2.坐火车 3.坐火箭 4.不回了

1.9.3 case

case命令是一个多分支的if/else命令，case变量的值用来匹配value1,value2,value3等等。

匹配到后则执行跟在后面的命令直到遇到双分号为止(;;)

case命令以esac作为终止符。

case行尾必须为单词 in 每个模式必须以右括号 ） 结束

匹配模式中可使用方括号表示一个连续的范围，如[0-9]；使用竖杠符号“|”表示或。

最后的“*）”表示默认模式，当使用前面的各种模式均无法匹配该变量时，将执行“*）”后的命令序列。

格式

```
CMD=$1
case $CMD in
start)
    echo "starting"
    ;;
stop)
    echo "stoping"
;;
test)
    echo  “I’m testing”
;;
*)
    echo "Usage: {start|stop} "
;;
esac
```

1.10 循环

1.10.1 for循环

for循环命令用来在一个列表条目中执行有限次数的命令。

比如，你可能会在一个姓名列表或文件列表中循环执行同个命令。

for命令后紧跟一个自定义变量、一个关键字in和一个字符串列表（可以是变量）。

第一次执行for循环时，字符串列表中的第一个字符串会赋值给自定义变量，然后执行循环命令，直到遇到done语句；

第二次执行for循环时，会右推字符串列表中的第二个字符串给自定义变量

依次类推，直到字符串列表遍历完。

遍历结构:

第一种：

```
for N in 1 2 3

do

echo $N

done
```

或

```
for N in 1 2 3; do echo $N; done
```

或

```
for N in {1..3}; do echo $N; done
```

或

```
for N in {1,2,3}; do echo $N; done
```

注意：{}中的数字之间不能有空格

第二种：

```
for ((i = 0; i <= 5; i++))

do

echo "welcome $i times"

done
```

或

```
for ((i = 0; i <= 5; i++)); do echo "welcome $i times"; done
```

练习：计算从1到100的和。

```
s=0
for((i=1;i<=100;i++))
do
        s=$(($s + $i))
done
echo "sum="$s
```

1.10.2 while循环

注意：until循环与while正好相反，即：while是条件成立循环执行；until是条件不成立循环执行

while命令根据紧跟其后的命令(command)来判断是否执行while循环，当command执行后的返回值(exit status)为0时，则执行while循环语句块，直到遇到done语句，然后再返回到while命令，判断command的返回值，当得打返回值为非0时，则终止while循环。

第一种

while [expression]

do

command

…

done

练习：求1-10 各个数的平方和

```
num=1
while [ $num -le 10 ]
do
    SUM=`expr $num \* $num`
    echo $SUM
    num=`expr $num + 1`
done

num=1
while [ $num -le 10 ]
do
        sum=$(( $num * $num ))
        echo $sum
        num=$(( $num + 1 ))
done
```

第二种方式：

while ((expression))

do

command

…

done

```
i=1
while((i<=10))
do
        sum=$(( $i * $i ))
        echo $sum
        i=$(($i + 1))
done

i=1
while((i<=10))
do
        sum=$(( $i * $i  ))
        echo $sum
        let i++
done
```

1.11 函数

函数代表着一个或一组命令的集合，表示一个功能模块，常用于模块化编程。

以下是关于函数的一些重要说明：

在shell中，函数必须先定义，再调用

使用return value来获取函数的返回值

函数在当前shell中执行，可以使用脚本中的变量。

函数的格式如下：

函数名()

{

命令1…..

命令2….

return 返回值变量

}

结构:

[function] funname [()]

{

action;

[return int;]

}

function start() / function start / start()

eg:

```
function start() {
        echo "starting"
}
function stop {
        echo "stopping"
}
restart() {
        echo "restarting"
}

$1
```

注意：

如果函数名后没有（），在函数名和{ 之间，必须要有空格以示区分。

函数返回值，只能通过$? 系统变量获得，可以显示加：return 返回值

如果不加return，将以最后一条命令的运行结果，作为返回值。

return后的内容以字符串的形式写入，但是执行时会自动转成数值型，范围：数值n(0-255)

1.12 脚本调试

1) 编写脚本:test.sh

```
1 #!/bin/bash
2
3 a=$1
4 set -x    这里是添加的set -x
5 b=3
6 echo "b:"+$b
7 c=$a
8 echo $a
```

2) 执行 bash -x test.sh

说明:这将执行该脚本并显示所有变量的值。

3) 在shell脚本里添加set -x ,对部分脚本调试

执行 bash test.sh 1

```
显示:
+ b=3
+ echo b:+3
b:+3
+ c=1
+ echo 1
1
说明只对set -x以下的脚本进行调试
```

4) bash -n script 不执行脚本只是检查语法的模式，将返回所有语法错误。如:函数没有正确的闭合

```
编写test.sh 脚本
#!/bin/bash

for N in 1 2 3
do
        echo $N
这里忘记写done,for没有正确的闭合
function start() {
        echo "haha"
}
start
```

执行 bash -n test.sh

显示:test.sh: line 20: syntax error: unexpected end of file

5) bash –v script 执行并显示脚本内容

1.13 awk介绍(选讲)

1.13.1 cut命令

cut [选项] 文件名 默认分割符是制表符，一个制表符代表一列

选项:

-f 列号： 提取第几列

-d 分隔符： 按照指定分隔符分割列

eg: cut -f 2 aa.txt 提取第二列

eg: cut -d ":" -f 1,3 /etc/passwd 以:分割，提取第1和第3列

eg: cat /etc/passwd | grep /bin/bash | grep -v root | cut -d ":" -f 1 获取所有可登陆的普通用户用户名

cut的局限性 不能分割不定长度的空格

比如:df -h 不能使用cut分割

df -h | grep sda1 | cut -f 5

1.13.2 awk介绍

一个强大的文本分析工具

把文件逐行的读入，以空格为默认分隔符将每行切片，切开的部分再进行各种分析处理。

语法：awk '条件1{动作1}条件2{动作2}...'文件名

条件（Pattern）：

一般使用关系表达式作为条件： > >= <=等

动作（Action）：

格式化输出

流程控制语句

eg: df -h | awk '{print $1 "\t" $3}' 显示第一列和第三列

1.13.3 FS内置变量

eg: 输出可登陆用户的用户名和UID

cat /etc/passwd | grep "/bin/bash" | awk 'BEGIN {FS=":"} {print $1 "\t"$3 }'

这里使用FS内置变量指定分隔符为：,而且使用BEGIN保证第一行也操作，因为awk命令会在读取第一行后再执行条件

注意:指定分隔符用-F更简单

eg: cat /etc/passwd | grep "/bin/bash" | awk -F: '{print $1 "\t"$3 }' 效果同上

eg: 判断一下根目录的使用情况

df -h |grep sda1 | awk '{print $5}' | awk -F% '{print $1} $1<80{print "info"}$1>80{print "warning"}'

BEGIN 在所有数据读取之前执行

eg: awk 'BEGIN {printf "first Line \n"} {printf $2 }' aa.txt 在输出之前使用BEGIN输出内容

END 在所有数据执行之后执行

eg: awk 'END {printf "The End \n"} {print $2}' aa.txt 所有命令执行完后，输出一句"The End"

df -h | grep sda2 | awk '{print $5}' | awk -F% '{print $1}'

df -h | grep sda2 | awk '{print $5}' | cut -d"%" -f 1

eg: 获取所有用户信息里的用户名：

cat /etc/passwd | awk -F: '{print $1}'

awk -F: '{print $1}' /etc/passwd

eg: 获取当前机器的ip地址： ifconfig eth0

ifconfig eth0 | grep 'inet addr' | awk -F: '{print $2}' | awk '{print $1}'

1.14 sed介绍(选讲)

1.14.1 sed命令介绍

sed: stream editor

sed是一个非交互性文本流编辑器。它编辑文件或标准输入导出的文本拷贝。标准输入可能是来自键盘、文件重定向、字符串或变量，或者是一个管道的文本。

注意： sed并不与初始化文件打交道， 它操作的只是一个拷贝，然后所有的改动如果没有重定向到一个文件，将输出到屏幕。

语法：sed [选项]’[动作]’ 文件名

常用选项:

-n 使用安静（silent）模式。显示经过sed特殊处理的数据。

-e 允许多点编辑。

-i 直接修改读取的档案内容，而不是由屏幕输出。

命令	功能描述
a	新增， a 的后面可以接字符串，在下一行出现 注意：最好动作使用单引号，可以自动换行
c	替换
d	删除
i	插入， i 的后面可以接字符串
p	打印
s	查找并替换，例如 1,20s/old/new/g

1.14.2 sed命令的使用

eg: sed '2p' sed.txt 显示第二行和所有数据

eg: sed -n '2,3p' sed.txt 只显示第二和第三行

eg: df -h | sed -n '1p' 接收命令结果数据

eg: sed ‘2a bing’ sed.txt 在第二行后面添加数据

eg: sed ‘4i fengjie \ chenchen’ sed.txt 在第4行之前添加两行数据

eg: sed ‘2c this is replace’ sed.txt 替换第二行数据

eg: sed ‘s/it/edu360/g’ sed.txt 把sed.txt文件中的it替换为edu360,并输出

eg: sed -e '1s/1/34/g;3s/yangmi//g' sed.txt 同时进行多个替换

eg: sed –i ‘s/it/edu360/g’ sed.txt 要想真正替换，需要使用-i参数

```
[root@node2 ~]# more sed.txt
sldx
it spark
hadoop edu
it hadoop
it scala

[root@node2 ~]#
[root@node2 ~]# sed -e 's/it/edu360/g' sed.txt
sldx
edu360 spark
hadoop edu
edu360 hadoop
edu360 scala
```

eg: 使用sed获取机器的ip地址

注意:在对文件进行匹配的时候,^是一个文件的开始 $是一个文件的结束

^.*addr:意思是说从开始到addr全部的内容

ifconfig eth0 | grep 'inet addr'| sed 's/^.*addr://g' | sed 's/ Bcast.*$//g'