Bash Learning

一. Introduction to LINUX Systems

1.1. The history of LINUX and its versions

> The LINUX system was developed in 1991 by a Finnish university student, Linus, and a succession of enthusiasts.

> LIUNX distributions: the redhat series (redhat, centOS) and the debian series (debian, ubuntu). Both distributions use the LINUX kernel, so the basic operations are the same, the main difference is that the software is installed differently.

1.2. Features as open source software

> Free, accessible software source code is more secure, distributable and improved.

二. Linux common commands

1. Directory processing commands

```
# list
ls  # /-a /-l /-d
```

```
# make directories
mkdir # /-p
```

```
# change directory
cd # /.. /- /~
```

```
# print working directory
pwd
```

```
# remove empoty directory
rmdir
```

```
# copy
cp # /-r /-p
```

```
# move
mv
```

```
# remove
rm # /-rf /-i
```

2. Document processing commands

```
touch
```

```
cat # /-n
$ cat -n /etc/services
```

```
more # /q /Enter /space
$ more /etc/services
```

```
less # / :/ser (find the specified character,press n to find the next one)
$ less /etc/services
```

```
head # /-n
$ head -n 20 /etc/services
```

```
tail # /-n
$ tail -n 20 /etc/services
```

3. Permission management commands

```
# change file mode bits
chmod # / [{ugo} {+-=} {rwx}] [document/directory] / [mode=421]
[document/directory]
# r----4, w-----2, x----1
$ chmod g+w testfile
$ chmod 777 testfile
$ chmod -R 777 testfile # recursive modification
```

4. Search commands

```
# search the directory where the command is located
which
$ which ls
```

```
# search within document
grep  # /-i /-v /-n
$ cat sonnet104.txt |grep -n -i -v "three"
```

5. Help commands

```
# manual
man
$ man ls
```

```
help
$ ls --help
```

```
whatis
$ whatis ls
```

6. Compression and decompression commands

```
# GUN zip
gzip
$ gzip testfile

# GUN unzip
gunzip
$ gunzip testfile
```

> gzip and gunzip do not retain the original file after compression and decompression

```
tar
$ tar -zcvf sonnet104.tar.gz sonnet104.txt
$ tar -zxvf sonnet104.tar.gz
```

```
zip  # /-r (compressed directory)
$ zip sonnet104.zip sonnet104.txt

unzip
$ unzip sonnet104.zip
```

> zip and unzip do not retain the original file after compression and decompression

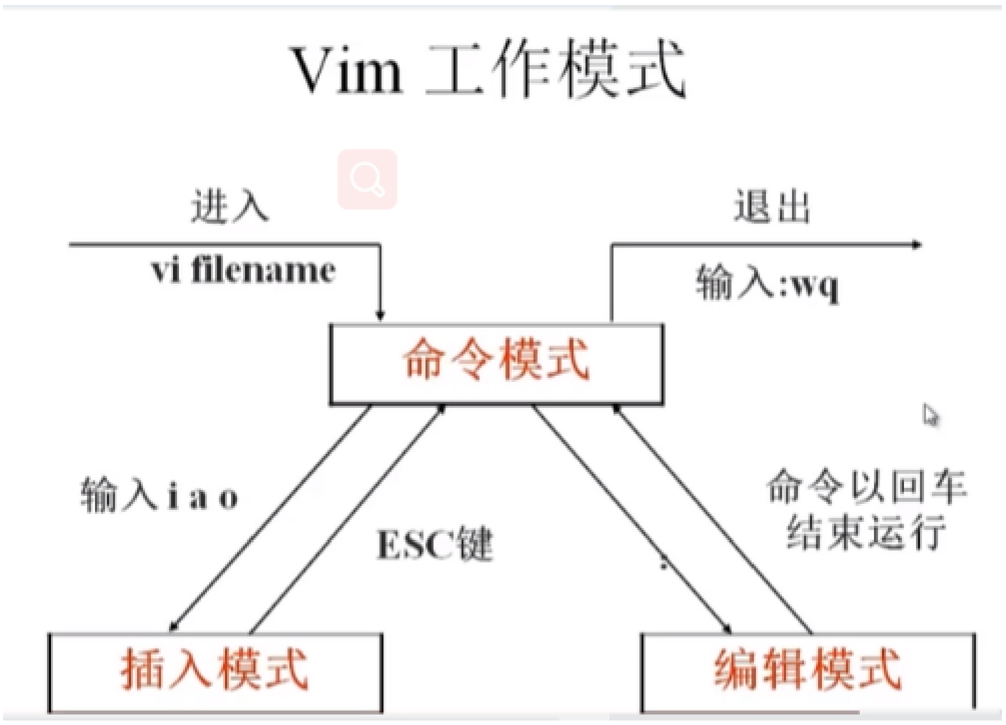
```
bzip2 # /-k ( retain the original file)
$ bzip2 sonnet104.txt

bunzip2
$ bunzip2 -k sonnet104.txt.bz2
```

> bzip2 is an upgraded version of gzip with a very impressive compression ratio

三. Text editor vim

> vim is a powerful full-screen text editor that creates, edits and displays text files



1. Insert commands

```
a  # Insert after the character where the cursor is located
A  # Insert at the end of the line where the cursor is located
i  # Insert before the character where the cursor is located
I  # Insert at the beginning of the line where the cursor is located
o  # Insert a new line under the cursor
O  # Insert a new line on the cursor
```

2. Locate commands

```
:set number     # Set the line number
:set nonumber   # Cancel line number
gg              # Go to first line
G               # Go to the last line
:n              # Go to nth row
$               # Move to end of line
0               # Move to beginning of line
```

3. Delete, copy, cut and paste commands

```
dd          # Delete the line where the cursor is
:n1,n2d     # Delete a specified range of rows

yy          # Copy the line where the cursor is
nyy         # Copy n lines below the cursor line

dd          # Cut the line where the cursor is
ndd         # Cut n lines below the line where the cursor is located

p/P         # Paste under or over the line where the cursor is located
```

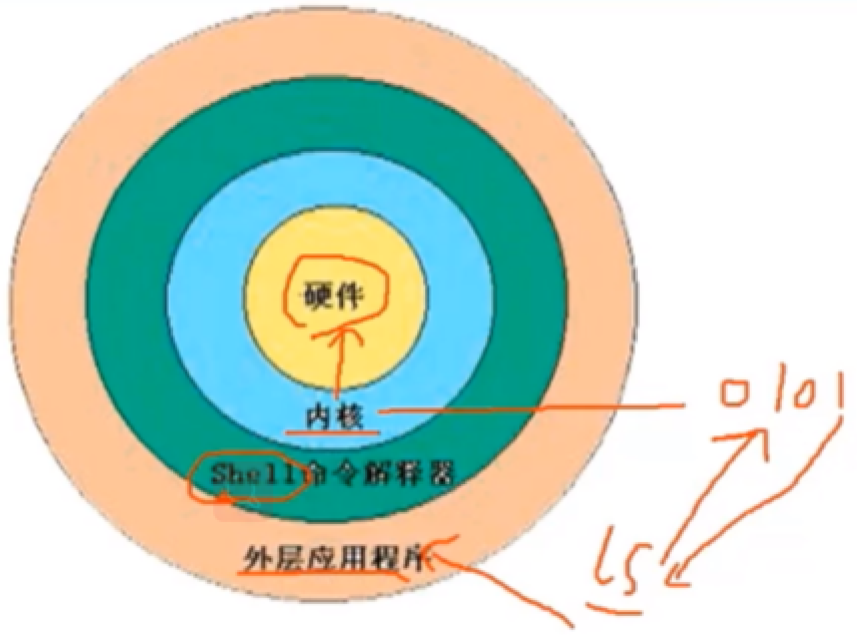
4. Replace command

```
:%s/old/new/g         # Replace the specified string with the full text

:n1,n2s/old/new/g     # Replace the specified string within a certain range
```

四. Shell

1. Overview of the shell



> The shell is a command line interpreter that provides an interface system level program for the user to send requests to the Linux kernel in order to run programs. The user can use the shell to start, hang, stop, and write a number of programs.

> The shell is also a powerful programming language that is easy to write, debug and very flexible. shell is an interpreted scripting language in which Linux system commands can be invoked directly.

2. Execution of shell scripts

2.1. The echo output command

```
echo  # /-e  # Support for backslash-controlled character conversion

$ echo -e \
> 'hello,
>  world!'
```

2.2. The first script

```
$ vim hello.sh
#!/bin/bash
#The first script

echo "Hello, world!"
```

> Before the script can be executed, it needs to be given execute permission

```
$ chmod 755 hello.sh
$ ./hello.sh

# Execution is possible without granting execution rights
$ bash hello.sh
```

2.3. Output redirection

```
0  # Standard input
1  # Standard output
2  # Standard error output

command > document   #Outputs the correct output to the specified file as an
overwrite
command >> document  #Output the correct output to the specified file as an
append
error command 2> document
error command 2>> document

command > document 2>&1  #Save both correct and incorrect output to the same
file as an overwrite
command >> document 2>&1
command >> document1 2>>document2 #Append the correct output to file 1 and the
incorrect output to file 2

# Commonly used example
$ nohup ./hello.sh >run.log 2>&1 &
```

2.4. Multi-command sequential execution and pipeline characters

```
;     command1; command2  #Sequential execution, no logical relationship
&&    command1 && command2 #Logical and, when command 1 is executed correctly, command 2 will be executed
||   command1 || command2  #Logical or, when command 1 is not executed correctly, command 2 will be executed
|    command1 |command2   #Pipeline character with the correct output of command 1 as the operation of command 2

$ ls && echo "yes" || echo "no"

$ cat sonnet104.txt |tr -cs A-Za-z "\n" |sort |uniq -c |sort -n -k 1,1 |awk '$1>2 {print $2}'
```

2.5. wildcard character

```
?     #Match a character
*     #Match 0 or any number of arbitrary characters
[]    #Match any of the characters in the brackets  [abc]
[-]   #Match any of the characters in the brackets  [a-z]
[^]   #Logical non, matches characters not in brackets [^0-9]

$ touch abc abcd 012 0abc
$ ls
$ ls ?abc
$ ls [^0-9]*
$ ls [0-9]*
```

2.6. Other special symbols

```
``    #Anti-quote. The content enclosed in backquotes is a system command, and serves the same purpose as $().
""    #Double inverted commas. None of the special symbols in double quotes have a special meaning, but $ and \ have a special meaning
''  #Single inverted commas. All special symbols in single inverted commas have no special meaning.
$()   #Referencing system commands
$    #Calling variables
\    #Escape character

$ name=lsw
$ echo '$name'
$ echo "$name"
$ echo '$(date)'
$ echo "$(date)"
```

3. Variables in Bash

> Variables are bits of computer memory in which the values stored can be changed. Each variable has a name, so it is easy to refer to it. The use of variables allows useful information or temporary information to be stored.

3.1. User-defined variables

> Effective only in the current shell

```
# Defining variable
$ name="Shengwei Li"

# Defining variable
$ aa=123
$ aa="$aa"456
$ aa=${aa}789

# Calling variable
$ echo $name

# View variable
$ set

# Delete variable
$ unset name
```

3.2. Environment variables

> This variable mainly stores data related to the operating environment of the system

> Environment variables can take effect in the current shell and all sub-shells. If written to a configuration file, it will take effect in all shells.

```
# Setting environment variable
$ export name="shengwei li"

# Query variable
$ env

# Delete variable
$ unset name
```

```
# Common environment variables - PATH
$ echo $PATH

$ PATH="$PATH":/home/bioinfo/sh
```

3.3. Position parameter variables

> This variable is mainly used to pass parameters or data into the script

```
$n   #$1-$9 for the first to ninth arguments, ${10}
$*   #Represents all parameters, looking at all parameters as a whole
$@   #Represents all parameters, but treats each parameter differently
$#   #Represents the number of all parameters
```

```
$ vim sum.sh
```

```
#!/bin/bash

num1=$1
num2=$2
num3=$3

sum=$(( $num1+$num2+$num3 ))
echo $sum

echo "A total of $# parameters"

echo "The parameters is $@."

echo "The parameter is $*."

$ chmod 755 sum.sh
$ ./sum.sh 1  2 3
```

3.5. Predefined variables

> This variable is one that is already defined in bash

```
# Receive keyboard input
read  # /-p /-t /-n /-s
```

```
$ vim read.sh

#!/bin/bash

read -t 30 -p "Pleace enter your name: " name

echo "Hello, $name."

read -t 30 -s -p "Pleace enter your age: " age

echo -e "\n"
echo "You are $age years old now."

read -t 30 -n 1 -p "Pleace enter your gender:[M/F] " gender
echo -e "\n"
echo "Your gender is $gender"
```

4. Shell programming

4.1. cut, awk, sed

```
# cut: field extraction command (intercept eligible columns, tab separated by
default)
cut  # /-f /-d

$ cut -f 2 student.txt
$ cut -f 2,3 student.txt
$ cut -d ":" -f 1,3 /etc/passwd

# Limitations of the cut command
$ df -h |cut -d "" -f 1,3
```

```
# awk command (reads by row, intercepts eligible columns; FS specifies
separator)
awk 'pattern1{action1} pattern2{action2}...'  document

## pattern
x>10
x>=10
x<=10
## action
1. Formatted output
2. flow control statements

$ cat student.txt |awk '$4>80 {print $2}'


## print command (Generally used in the awk command)
$ awk '{print $2 "\t" $3}' student.txt
$ df -h |awk '{print $1 "\t" $3}'

## BEGIN, FS, END
$ cat student.txt |awk 'BEGIN {print "Transcript \n"} {print $2 "\t" $4}'

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

$ cat student.txt |awk 'END {print "End \n"} {print $2 "\t" $4}'
```

```
# sed command (line operation, which differs from vim in that not only can you
modify the contents of a file, but also supports pipe operations, which can
modify the result of a command)

# sed is a lightweight stream editor for selecting, replacing, deleting and
adding commands to data

sed [option] '[action]' document

## option
-n  # Outputting command-processed data to the screen
-e  # Allows multiple sed commands to be applied to input data
-i  # Save the results of the changes directly to a file

## action
a  # add
c  # row replacement
i  # insert
d  # delete
```

```
p  # print
s  # string replacement

$ sed '2p' student.txt
$ sed -n '2p' student.txt
$ sed '2,4d' student.txt
$ sed '2a hello' student.txt
$ sed '2i hello' student.txt
$ sed '2c No such person' student.txt
$ sed '3s/Zhang/ZHANG/g' student.txt
$ sed -i '3s/Zhang/ZHANG/g' student.txt
$ cat student.txt
$ sed -e 's/Li/LI/g; s/si/SI/g' student.txt
```

4.2. sort, uniq, wc

```
sort [option] document

## option
-u  # remove duplicate rows
-r  # reverse sorting
-o  # write the results to a file  / sort -r number.txt -o number.txt
-n  # sorting by numeric type
-t  # specify the separator
-k  # sort by the specified field

$ sort /etc/passwd
$ sort -r /etc/passwd
$ sort -t ":" -k 3,3 /etc/passwd
$ sort -n -t ":" -k 3,3 /etc/passwd
```

```
uniq [option] document

## option
-c  # display the number of times the row recurs next to each column
-d  # show only recurring rows

$ cat sonnet104.txt |tr -cs A-Za-z '\n' |sort |uniq -c |sort -n -k 1,1 |awk
'$1>2 {print $2}'
$ cat sonnet104.txt |tr -cs A-Za-z '\n'|sort |uniq -c -d
```

```
wc [option] document

## option
-l
-w
-m
```

4.3. Process control

4.3.1 if

```
#-----------------------
if [ condition ]
    then
        process
fi

$ if [ $(ps -ef | grep -c "ssh") -gt 1 ]; then echo "true"; fi

#----------------------
if [ condition ]
    then
        process1
    else
        process2
fi


#----------------------
if [ condition ]
    then
        process1
elif [ condition2 ]
    then
        process2
elif [ condition3 ]
    then
        process3
else
    process4
fi


#----------------------
vim used.sh

#!/bin/bash
# counting root partition usage

rate=$(df -h |grep "sda3" |awk '{print $5}' |cut -d "%" -f 1)

echo $rate

if [ $rate -ge 10 ]
    then
        echo "Warning! /dev/sda3 is full.rate= $rate."
fi
```

4.3.2 case

```
case $variable in
    value1)
        process1;;
    value2)
        process2;;
    value3)
```

```
        process3;;
esac

#--------------------
vim case.sh

#/bin/bash

# determining user input

read -t 30 -p "Do you want to continue? [Y/N]"  i

case $i in
    Y | y)
        echo "ok, we will continue!";;
    N | n)
        echo "OK, Good By.";;
    *)
        echo "error choice.";;
esac
```

4.3.3 for

```
#------------
for variable in item1 item2 item3
    do
        process
    done

$ for var in item1 item2 ... itemN; do command1; command2… done;

#---------------
for((value;condition;change))
    do
        process
    done

#-------------------
vim for1.sh

#!/bin/bash
# Printing time

for time in morning noon afternoon evening
    do
        echo "This time is $time!"
    done

#----------------
vim for2.sh

#/bin/bash

#0+1+2+...+100

s=0
```

```
for (( i=1;i<101;i=i+1 ))
    do
        s=$(( $s+$i ))
    done

echo "The result of 1+2+...+100= $s."
```

4.3.4 while

```
while [ condition ]
    do
        process
    done

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

#0+1+2+...+100

i=1
s=0

while [ $i -le 100 ]
    do
        s=$(( $s+$i ))
        i=$(( $i+1 ))
    done

echo "The result of 1+2+...+100= $s."
```

4.3.5 until

```
# The until loop is the opposite of the while loop in that it loops if the
condition does not hold, and terminates once it does.

#----------------------
vim until.sh

#!/bin/bash

#0+1+2+...+100

i=1
s=0

until [ $i -gt 100 ]
    do
        s=$(( $s+$i ))
        i=$(( $i+1 ))
    done

echo "The result of 1+2+...+100= $s."
```