项目八　Bash Shell 简介

【任务描述】

Bash Shell 是当今 GNU/Linux 操作系统最为广泛使用的 Shell 发行版之一，这是一个开源的 GNU 项目，于 1989 年发布，可用于编程和交互使用。它也包括了命令行补全、命令行编辑、键绑定、无限制大小的命令历史记录等。

【学习目标】

1. 了解什么是 Bash Shell；
2. 了解 Bash Shell 的发展历史；
3. 了解 Bash Shell 的语法与特性。

任务一　认识 Bash Shell

Bash 是一种 Unix shell 和命令语言，由 Brian Fox 为 GNU 项目编写，作为 Bourne shell 的自由软件替代品。它于 1989 年首次发布，已被用作大多数 Linux 发行版的默认登录 shell。Bash 是 Linus Torvalds 移植到 Linux 的第一个程序之一，还有 GCC。Windows 10 和 Windows 11 也可以通过 Windows 子系统 for Linux 获得一个版本。它也是 Solaris 11 中的默认用户 shell。Bash 也是 Apple macOS 版本的默认 shell，从 10.3(最初，默认 shell 是 tcsh)到 2019 年发布的 macOS Catalina，该版本将默认 shell 更改为 zsh，尽管 Bash 仍然可用作为替代 shell。

Bash 是一个命令处理程序，它通常运行在一个文本窗口中，用户可以在其中键入导致操作的命令。Bash 还可以从一个称为 shell 脚本的文件中读取和执行命令。像大多数 Unix shell 一样，它支持文件名通配符(通配符匹配)、管道、文档、命令替换、变量和用于条件测试和迭代的控制结构。关键字、语法、动态作用域变量和语言的其他基本特性都是从 sh 复制的。其他特性，例如历史，都是从 csh 和 ksh 复制的。Bash 是一个 posix 兼容的 shell，但是有许多扩展。

这个 shell 的名字是 Bourne Again shell 的首字母缩写，它取代了 Bourne shell 的名字和“重生”的概念。

Bash 1.03 版本(1989 年 8 月)的一个安全漏洞被称为 Shellshock，在 2014 年 9 月初被发现，并迅速导致了互联网上的一系列攻击。在发现这些漏洞后，很快就发布了修复这些漏洞的补丁。

任务二　Bash Shell 的发展历史

1988 年 1 月 10 日，在 Richard Stallman 对之前的开发人员缺乏进展感到不满之后，Brian Fox 开始编写 Bash。Stallman 和自由软件基金会(FSF)考虑了一个自由的 shell，它可以运行现有的 shell 脚本，对于一个由 BSD 和 GNU 代码构建的完全自由的系统来说是非常具有战略意义的，这是他们自己资助的少数项目之一，Fox 作为 FSF 的雇员承担了这项工作。Fox 在 1989 年 6 月 8 日发布了 Bash 的 beta 版本。99，并一直是主要的维护者，直到 1992 年中期到 1994 年中期之间的某个时候，他被 FSF 解雇了，他的责任转移到了另一个早期贡献者 Chet Ramey 身上。

从那时起，Bash 已成为迄今为止 Linux 用户中最受欢迎的 shell，成为该操作系统的各种发行版(尽管 Almquist shell 可能是默认的脚本 shell)和苹果在 2019 年 10 月 Catalina 之前发布的 macOS 上的默认交互 shell。Bash 也被移植到 Microsoft Windows 并与 Cygwin 和 MinGW 一起发布，通过 DJGPP 项目移植到 DOS，通过 Novell NetWare 移植到 OpenVMS，通过 GNV 项目移植到 ArcaOS，并通过各种终端仿真应用程序移植到 Android。

2014 年 9 月，Unix/Linux 专家 Stéphane Chazelas 在该程序中发现了一个安全漏洞。该漏洞于 9 月 24 日首次披露，名为 Shellshock，编号为 CVE-2014-6271、CVE-2014-6277 和 CVE-2014-7169。这个错误被认为是严重的，因为使用 Bash 的 CGI 脚本可能是脆弱的，允许任意代码执行。该错误与 Bash 如何通过环境变量将函数定义传递给子 shell 有关。

任务三　Bash Shell 的语法与特性

bash 的命令语法是 Bourne shell 命令语法的超集。数量庞大的 Bourne shell

脚本大多不经修改即可以在 bash 中执行，只有那些引用了 Bourne 特殊变量或使用了 Bourne 的内置命令的脚本才需要修改。bash 的命令语法很多来自 Korn shell（ksh）和 C shell（csh），例如命令行编辑，命令历史，目录栈，$RANDOM 和 $PPID 变量，以及 POSIX 的命令置换语法：$(...)。作为一个交互式的 shell，按下 TAB 键即可自动补全已部分输入的程序名，文件名，变量名等等。

使用'function'关键字时，Bash 的函数声明与 Bourne/Korn/POSIX 脚本不兼容（Korn shell 有同样的问题）。不过 Bash 也接受 Bourne/Korn/POSIX 的函数声明语法。因为许多不同，Bash 脚本很少能在 Bourne 或 Korn 解释器中运行，除非编写脚本时刻意保持兼容性。然而，随着 Linux 的普及，这种方式正变得越来越少。不过在 POSIX 模式下，Bash 更加符合 POSIX。

bash 的语法针对 Bourne shell 的不足做了很多扩展。这里将其中的一些进行列举：

- 花括号扩展

花括号扩展是一个从 C shell 借鉴而来的特性，它产生一系列指定的字符串（按照原先从左到右的顺序）。这些字符串不需要是已经存在的文件。

```
$ echo a{p,c,d,b}eape ace ade abe$ echo {a,b,c}{d,e,f}ad ae af bd be bf cd c
e cf
```

花括号扩展不应该被用在可移植的 shell 脚本中，因为 Bourne shell 产生的结果不同。

```
#! /bin/sh
# 传统的 shell 并不产生相同结果 echo a{p,c,d,b}e
# a{p,c,d,b}e
```

当花括号扩展和通配符一起使用时，花括号扩展首先被解析，然后正常解析通配符。因此，可以用这种方法获得当前目录的一系列 JPEG 和 PEG 文件。

```
ls *.{jpg,jpeg,png}     # 首先扩展为*.jpg *.jpeg *.png，然后解析通配符
echo *.{png,jp{e,}g}     # echo 显示扩展结果；花括号扩展可以嵌套。
```

除了列举备选项，还可以用“..”在花括号扩展中指定字符或数字范围。较新的 Bash 版本接受一个整数作为第三个参数，指定增量。

```
$ echo {1..10}1 2 3 4 5 6 7 8 9 10
$ echo file{1..4}.txtfile1.txt file2.txt file3.txt file4.txt$ echo {a..e}a b c d
e$ echo {1..10..3}1 4 7 10
$ echo {a..j..3}a d g j
```

当花括号扩展和变量扩展一起使用时，变量扩展解析于花括号扩展之后。有时有必要使用内置的 eval 函数。

```
$ start=1; end=10
$ echo {$start..$end}   # 由于解析顺序，无法得到想要的结果
{1..10}
$ eval echo {$start..$end} # 首先进行变量扩展的解析
1 2 3 4 5 6 7 8 9 10
```

- 使用整数

与 Bourne shell 不同的是 bash 不用另外生成进程即能进行整数运算。bash 使用((...))命令和$[...]变量语法来达到这个目的：

```
VAR=55            # 将整数 55 赋值给变量 VAR
((VAR = VAR + 1))      # 变量 VAR 加 1。注意这里没有'$'
((++VAR))      # 另一种方法给 VAR 加 1。使用 C 语言风格的前缀自增
((VAR++))      # 另一种方法给 VAR 加 1。使用 C 语言风格的后缀自增
echo $((VAR * 22))     # VAR 乘以 22 并将结果送入命令
echo $[VAR * 22]       # 同上，但为过时用法
```

((...))命令可以用于条件语句，因为它的退出状态是 0 或者非 0（大多数情况下是 1），可以用于是与非的条件判断：

```
if ((VAR == Y * 3 + X * 2))
then
      echo Yes
fi
((Z > 23)) && echo Yes
```

((...))命令支持下列比较操作符：'==', '!=', '>', '<', '>=', 和'<='。

bash 不能在自身进程内进行浮点数运算。当前有这个能力的 unix shell 只有 Korn shell 和 Z shell。

- 输入输出重定向

bash 拥有传统 Bourne shell 缺乏的 I/O 重定向语法。bash 可以同时重定向标准输出和标准错误，这需要使用下面的语法：

```
command &> file
```

这比等价的 Bourne shell 语法"command > file 2>&1"来的简单。2.05b 版本以后，bash 可以用下列语法重定向标准输入至字符串（称为 here string）：

```
command <<< "string to be read as standard input"
```

如果字符串包括空格就需要用引号包裹字符串。

- 进程内的正则表达式

bash 3.0 支持进程内的正则表达式，使用下面的语法：

```
[[ string =~ regex ]]
```

正则表达式语法同 regex(7) man page 所描述的一致。正则表达式匹配字符串时上述命令的退出状态为 0，不匹配为 1。正则表达式中用圆括号括起的子表达式可以访问 shell 变量 BASH_REMATCH，如下：

```
if [[ abcfoobarbletch =~ 'foo(bar)bl(.*)' ]] then                echo The regex matches!
                echo $BASH_REMATCH          -- outputs: foobarbletch
                echo ${BASH_REMATCH[1]} -- outputs: bar
                echo ${BASH_REMATCH[2]} -- outputs: etch
 fi
```

使用这个语法的性能要比生成一个新的进程来运行 grep 命令优越，因为正则表达式匹配在 bash 进程内完成。如果正则表达式或者字符串包括空格或者 shell 关键字，（诸如'*'或者'?'），就需要用引号包裹。Bash 4 开始的版本已经不需要这

么做了。

- 转移字符

$'string'形式的字符串会被特殊处理。字符串会被展开成 string，并像 C 语言那样将反斜杠及紧跟的字符进行替换。反斜杠转义序列的转换方式如下：

操作	定义
\a	响铃符
\b	退格符
\e	ANSI 转义符，等价于\033
\f	馈页符
\n	换行符
\r	回车符
\t	水平制表符
\v	垂直制表符
\\	反斜杠
\'	单引号
\nnn	十进制值为 nnn 的 8-bit 字符(1-3 位)
\xHH	十六进制为 HH 的 8-bit 字符(1 或 2 位)
\cx	Control-X 字符

扩展后的结果将被单引号包裹，就好像美元符号一直就不存在一样。

双引号包裹的字符串前若有一个美元符号（$"..."）将会使得字符串被翻译成符合当前 locale 的语言。如果当前 locale 是 C 或者 POSIX，美元符号会被忽略。如果字符串被翻译并替换，替换后的字符串仍被双引号包裹。

- 关联数组

Bash 4.0 开始支持关联数组，通过类似 AWK 的方式，对于多维数组提供了伪支持。

```
$ declare -A a              # 声明一个名为 a 的伪二位数组
$ i=1; j=2
$ a[$i,$j]=5                # 将键 "$i,$j" 与值 5 对应
$ echo ${a[$i,$j]}
```

- 移植性

调用 Bash 时指定--posix 或者在脚本中声明 set -o posix，可以使得 Bash 几乎遵循 POSIX 1003.2 标准。若要保证一个 Bash 脚本的移植性，至少需要考虑到 Bourne shell，即 Bash 取代的 shell。Bash 有一些传统的 Bourne shell 所没有的特性，包括以下这些：

- 某些扩展的调用选项
- 命令替换（即$()）（尽管这是 POSIX 1003.2 标准的一部分）
- 花括号扩展
- 某些数组操作、关联数组
- 扩展的双层方括号判断语句
- 某些字符串生成操作
- 进程替换
- 正则表达式匹配符
- Bash 特有的内置工具
- 协进程

- 键盘快捷键

Bash 默认使用 Emacs 的快捷键，可以通过 set -o vi 让它使用 Vi 的快捷键。

- 进程管理

Bash 有两种执行命令的模式：批处理模式、并发模式。 要以批处理模式执行命令（即按照顺序），必须用;分隔

```
command1 ; command2
```

在这个例子中，当 command1 执行完毕，即执行 command2 要并发执行两个命令，它们必须用&分隔。

```
command1 & command2
```

在这种情况下，command1 在后台执行（通过&），从而立即将控制返回到 shell，以执行 command2

- 管道

管道用于将一系列命令联系起来，也就是将一个命令的输出通过一个无形的"管道"作为另外一个命令的输人。管道命令是"|"，例如：

```
[root@echo root]#cat dir.out | grep"test" | wc -l
```

管道将 cat 命令的输出(列出 dirout 文件的内容)输送给 grep 命令，grep 在内容里查找出单词 test，grep 的输出则是过滤出所有包含单词 test 的行，过滤出的内容又被输送给 wc 命令，wc 命令最后统计内容总的行数。

项目九　Bash Shell 基础

【任务描述】

【学习目标】

1.

2.

3.

任务一　Shell 脚本

(一)　什么是 Shell 脚本？

系统命令 + 变量 + 函数 + 正则表达式 + 流程控制 + 终端输入输出 = 自动批量处理系统任务

(二)　Shell 脚本的通用格式

```
#!/bin/bash                          //  申明是在 bash 环境下执行的脚本
# some comment                       //  写明一些注释，方便以后调试，阅读，与他人协作。理解代码
var1=value1                          //  定义变量
var2=value2
fun(){                               //  定义函数
    cmd0
}
shell keyword          //  Bash 的关键字，比如 if  for  while .....
cmd1                                 //  通过关键字组合起来的 Bash 的命令
```

cmd2

cmd3

在 Linux 下推荐使用 vim 编辑器编写脚本；

给脚本赋予可执行的权限：

chmod +x example.sh

示例：

[root@localhost shell]# vim example.sh

#!/bin/bash

This is a test script.

huangdaojin

2015-08-12

v0.1

cd /boot/

ls

cd -

[root@localhost shell]# chmod +x example.sh

(三) 脚本的执行方式

1) ./example.sh = /path/example.sh
2) sh example.sh = bash example.sh
3) . example.sh
4) source example.sh

说明：

方式 1，2 会打开一个 BASH 的子进程来进行执行。（一定需要执行权限）

方式 3，4 会在当前 BASH 环境中执行。 (不需要执行权限)

大多数情况，使用第 1，2 种方式执行； 当脚本的工作任务就是初始化当前的 shell 环境时，使用第 3，4 种。

> 问：
>
> 为什么不能像系统命令如 ls , mount 直接用[root@localhost shell]# example.sh 这样方式执行。
>
> 原因：系统的命令都是放在 bin 或 sbin 目录下的可执行程序，而这些放系统命令的目录系统通过一个环境变量 PATH 来搜索。
>
> [root@localhost shell]# echo $PATH
>
> /usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:/root/bin

如果需要将一个 shell 脚本变成一个系统命令，处理方式：

1. 将 example.sh 复制到 PATH 定义的目录中的任一目录;

2. 将 example.sh 所在的目录/shell 加入到 PATH 变量中去。（推荐）

[root@localhost shell]# PATH=$PATH:/shell

[root@localhost shell]# export PATH

如果需要永久保存/shell 目录的 PATH 变量中，则需要将 PATH=$PATH:/shell 写到 bash_profile 中。

（四） 调试脚本

1) sh -xv script.sh (常用) sh -n script.sh
2) 在脚本中使用 set -x 和 set +x 来限定调试的范围
3) 使用#!/bin/bash -xv

任务二　输出与输入

(一)　终端打印输出

1)　echo 用法：

```
[root@localhost shell]# echo "Welcome"
Welcome

[root@localhost shell]# echo "123" ; echo "456"
123
456

[root@localhost shell]# echo -n    "123" ; echo "456"
123456

[root@localhost shell]# echo -e    "123\t456\n789\t222"
123       456
789       222
[root@localhost shell]# echo -e    "123\t456"; echo -e "654\t321"
123       456
654       321

[root@localhost shell]#    echo "$PWD"
/home/demo
[root@localhost shell]#    echo '$PWD'
$PWD

[root@localhost shell]#    echo "hello world !"
bash: !": event not found
[root@localhost shell]# echo 'hello world !'
```

```
hello world !
[root@localhost shell]# echo hello\ world\ !
hello world !
```

2) printf 的用法：

```
[root@localhost shell]# vim    2.sh
#!/bin/bash
printf "%5s %-10s %-4s\n" num name source
printf    "%5s %-10s %-4.2f\n" 1 zhangsan 95.679
printf    "%5s %-10s %-4.2f\n" 2 lisi 98.787
printf    "%5s %-10s %-4.2f\n" 3 wangwu 87.334


 [root@localhost shell]#    ./2.sh
  num name            source
        1 zhangsan       95.68
        2 lisi                    98.79
         3 wangwu          87.33
```

（二） Shell 的变量

变量的类型：

- 环境变量（全局变量）
- 自定义变量
- 预定义变量

示例：

1) 环境变量

```
[demo@teacher ~]$ env
```

```
[root@teacher ~]# echo $PATH
/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin
[root@teacher ~]# PATH="$PATH:/shell"
[root@teacher ~]# echo $PATH
/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin:/shell
```

2) 自定义变量

- 变量赋值

shell 中变量不需要声明，可直接使用。但是在自定义变量名的时候，要避免变量名冲突（和环境变量或预定义变量或命令冲突）

```
[root@teacher ~]# var="123456"
[root@teacher ~]# echo $var
123456

默认情况下，定义的变量不会在子 shell 进程继承
[root@teacher ~]# bash
[root@teacher ~]# echo $var

[root@teacher ~]# export var="abcdefg"
设置为当前 BASH 的子 shell 进程中可以继承 (非真正的全局变量，只能够向下继承)。

[root@teacher ~]# echo $var
abcdefg
[root@teacher ~]# bash
[root@teacher ~]# echo $var
abcdefg
```

- 输出变量值

```
[demo@365linux ~]$ a=123;b=456
[demo@365linux ~]$ echo $ab

[demo@365linux ~]$ echo $a$b
123456
[demo@365linux ~]$ echo ${a}b
123b
[demo@365linux ~]$ echo "$a"b
123b
[demo@365linux ~]$ echo $a\b
123b
[demo@365linux ~]$ echo $a'b'
123b

[root@teacher ~]# echo $var
abcdefg
[root@teacher ~]# echo ${#var}
7
```

- 数组

```
[root@teacher ~]# var=(1 2 3 4 5)     #默认索引为 01234
[root@teacher ~]# echo $var
1
[root@teacher ~]# echo ${var[1]}
2
[root@teacher ~]# echo ${var[0]}
[root@teacher ~]# echo ${var[*]}
1 2 3 4 5
```

```
[root@teacher ~]# echo ${#var[*]}
5
[root@localhost shell]# a[0]=1
[root@localhost shell]# a[1]=2
[root@localhost shell]# echo ${a}
1
[root@localhost shell]# echo ${a[1]}
2
[root@localhost shell]# echo ${a[*]}

关联数组，可以自定义索引
[root@teacher ~]# declare -A fruits_value
[root@teacher ~]# fruits_value=([apple]='100dollars'
[orange]='200dollars')
[root@teacher ~]# echo "apple costs ${fruits_value[apple]}"
apple costs 100dollars

[root@teacher ~]#    echo ${!fruits_value[*]}
```

3) 预定义变量

实际上是一种全局变量，专为可执行的程序或脚本设计,变量值随脚本执行的情况而自动变化

```
$0 $1 $2 $3 .....
```

脚本本身 脚本的第一个参数 脚本的第二个参数 脚本的第三参数 。。。。。。

```
$#
```

脚本的参数的总个数

```
$*   $@
```

列出脚本的所有参数

两者的区别：

$*列出所有的参数作为一个整体输出

$@列出所有的参数是一个一个单独输出

```
$$
```

PID

```
$?
```

脚本执行后的返回值，若上一命令执行成功，则返回 0（真），否返回非 0（假）

示例：

```
#!/bin/bash
echo -e   "$0"   "\tThis is value of \$0"
echo -e   "$1"   "\tThis is value of \$1"
echo -e   "$2"   "\tThis is value of \$2"
echo -e   "$3"   "\tThis is value of \$3"
echo -e   "$#"   "\tThis is value of \$#"
echo -e   "$*"   "\tThis is value of \$*"
echo -e   "$@"    "\tThis is value of \$@"
echo -e   "$$"   "\tThis is value of \$\$"
echo -e   "$?"   "\tThis is value of \$?"
ls   /xxx
echo -e   "$?"   "\tThis is value of \$?"
```

```
［root@teacher 02_example]# ./02.sh     -a    -b -not    --help    -cd --file
./02.sh  This is value of $0
-a   This is value of $1
-b   This is value of $2
-not      This is value of $3
6    This is value of $#
-a -b -not --help -cd --file       This is value of $*
-a -b -not --help -cd --file       This is value of $@
29952   This is value of $$
0    This is value of $?
ls: 无法访问/xxx: 没有那个文件或目录
2    This is value of $?
```

关于返回值 $?

0 为真 ，即当前的命令执行成功。

非 0 为假， 即当前 的命令执行不成功。

示例：

```
[root@localhost shell]# ls
1.sh    2.sh    example.sh
[root@localhost shell]# echo $?
0
[root@localhost shell]# ls +
ls: 无法访问+: 没有那个文件或目录
[root@localhost shell]# echo $?
2
```

(三) 从标准输入

文件或键盘交互输入获取变量的值

```
[root@teacher ~]# read a
123456
[root@teacher ~]# echo $a
123456

[root@localhost shell]# echo "test" > 1.txt
[root@localhost shell]# read b <1.txt
[root@localhost shell]# echo $b
test

-p ： 输出提示字符串
[root@teacher ~]# read -p "Please input passwd ：  " pw

-s ：输入过程中终端不回显
[root@teacher ~]# read -s -p "Please input passwd: " pw

-t ：输入等待的超时时间
[root@teacher ~]# read -t 10 -s -p "Please input passwd: " pw
:/shells
-n ： 限制输入 n 个字符为变量值
[root@localhost shell]# read -n 5 test


-d ： 定义输入结束符，比如：
[root@localhost shell]# read -d + ab
qwerty＋
[root@localhost shell]# echo $ab
qwerty
```

（四） 输入输出重定向

返回值介绍

```
0 stdin
1 stdout
2 stderr
```

输出重定向

```
>  >>
2> 2>>
&>
```

重定向到特殊设备

```
> /dev/null
> /dev/stdout
> /dev/stdin
> /dev/pts/4
```

输入重定向

```
<  <<
```

使用技巧：用 cat 命令和<<配合输出一个格式化到文本内容到屏幕或文件，EOF 是指定的自定义结束符。

输出到屏幕：

```
[root@qianyun 02_example]# cat <<EOF
> xuehao    xingming    fenshu
> 1         zhangsan    90
```

```
> 2            lisi            100
> EOF
```

输出到文件：

```
cat   <<EOF>log.txt
1        LOG FILE HEADER
2        this is a test log file
EOF
```

或者 cat > log.txt <<EOF

tee 输出重定向的命令

-a, --append

-i 忽略中断信号

与> 和>>不同的是，tee 是进行复制传递， 即可以输出到屏幕，同时输出到文件或管道。

If a FILE is -, copy again to standard output

示例：

```
[root@localhost shell]# grep "root" /etc/passwd |tee 1.txt
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
[root@localhost shell]# cat 1.txt
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin



[root@LEP01 ~]# grep "root" /etc/passwd |tee -
root:x:0:0:root:/root:/bin/bash
```

```
operator:x:11:0:operator:/root:/sbin/nologin
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin

[root@LEP01 ~]# grep "root" /etc/passwd |tee -a 1.txt

[root@LEP01 ~]# grep "root" /etc/passwd |tee $SSH_TTY |wc -l
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
 2

 [root@LEP01 ~]#cat a* |tee -a out.txt | cat -n
```

任务三　Shell 的关键字

(一)　数学运算

1)　算数运算

```
[root@LEP01 ~]# expr 1 + 3
注意数字两边有空格，支持　＋　- \* / %

[root@LEP01 ~]# expr 3 - 5
-2

[root@LEP01 ~]# echo $((6/3))　　#常用

[root@LEP01 ~]# echo $[5*2]

[root@LEP01 ~]# let a=5-2;echo $a　#最常用
```

let a++ ==> let a=a+1

let a-- ==> let a=a-1

let a+=2 ==> let a=a+2

let a-=2 ==> let a=a-2

[root@LEP01 ~]# declare -i a=5-2;echo $a

[root@LEP01 ~]# bc

bc 1.06.95

Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.

This is free software with ABSOLUTELY NO WARRANTY.

For details type `warranty'.

6/3

2

6/4

1

2-5

-3

[root@LEP01 ~]# echo "6+3" |bc

9

[root@LEP01 ~]# echo "6.3-3.2" |bc

3.1

[root@LEP01 ~]# echo "6.8/2" |bc

3

[root@LEP01 ~]# echo "scale=2;6.8/2" |bc

3.40

[root@LEP01 ~]# echo "4>3" |bc 与$?的返回值相反。

```
1
[root@LEP01 ~]# echo "3>4" |bc
0

[root@LEP01 ~]# echo "3.21<4.66" |bc
1
[root@LEP01 ~]# echo "3.21>4.66" |bc
0
```

2) 进制转换

```
#!/bin/bash
no=100
echo "obase=2;$no" |bc

no=1100100
echo "obase=10;ibase=2;$no" |bc

计算平方以及平方根：
echo "sqrt(100)" | bc
echo "10^10" |bc
```

(二) 获取时间

```
date
date +%s
date --date "2013-08-17 09:10:11" +%s
date -d "@1279592730"
date -d "1970-01-01 utc 1279592730 seconds"
date -d "1970-01-01 14781 days" "+%Y/%m/%d"
```

```
date --date "Jan 20 2009" +%A
date
date -s "2013/12/11 11:22:33"
```

(三) 函数

定义

```
fun ()
{
    statements;
}
```

引用

```
fun
```

导出

```
export -f fun
```

示例：

```
#!/bin/bash
# 这一个脚本演示了函数的使用场景。
# 写一个脚本，让用户输入用户名，如果输入匹配到了内置的用户名，就输出 yes 和用户名，如果不匹配，就报错，并输出当前的工作路径。

read -p  "Please input your name : "  username
case  $username in
        root)
            echo "YES"
            echo "$username"
            ;;
```

```
        zhangsan)
            echo "YES"
            echo "$username"
            ;;
        lisi)
            echo "YES"
            echo "$username"
            ;;
        *)
            echo "NO"
            echo "$PWD"
esac
```

在脚本中，echo 在匹配到用户名后重复了很多的代码，用函数改进一下：

```
fun1(){
        echo "YES"
        echo "$username"
}
fun2(){
        echo "NO"
        echo "$PWD"
}

read -p  "Please input your name : "  username
case  $username in
        root)
            fun1
            ;;
        zhangsan)
            fun1
```

```
        ;;
    lisi)
        fun1
        ;;
    *)
    fun2
esac
```

（四） 命令执行

```
$ cmd1 | cmd2| cmd3          管道
```

在执行命令的时候打开一个新的子进程：

```
$ (cmd)
```

示例：

```
pwd ; cd /boot ; ls ; pwd
cd -
pwd;(cd /boot; ls);pwd


$ cmd_output=`commands`
$ cmd_output=$(ls |cat -n)
```

后面的命令先执行；建议使用 cmd_output=$(commands), 常用于与 date 命令配合，获得当前的时间。

```
$ cmd1;cmd2;cmd3   前一个命令执行完，就执行后面一个。
$ cmd1 && cmd2  前一个命令执行成功，才执行后一个。
$ cmd1 || cmd2  前一个命令执行不成功，才执行后一个。
```

（五） 流程控制

1) 判断

test 命令

```
[root@teacher ~]# test -d /etc
[root@teacher ~]# echo $?
0
[root@teacher ~]# test -d /etccc
[root@teacher ~]# echo $?
1
[root@teacher ~]# test -d /etc && echo "directory"
directory
[root@teacher ~]# test -d /etccc && echo "directory"

[root@teacher ~]# test -d /etccc || echo "Not directory"
Not directory

[root@teacher ~]# path=/etc
[root@teacher ~]# test -d "$path" && echo "dir" || echo "Not dir"
dir
[root@teacher ~]# path=/etccc
[root@teacher ~]# test -d "$path" && echo "dir" || echo "Not dir"
Not dir
[root@teacher ~]# test -d "$path" || echo "Not dir" && echo "dir"
这是错误的写法，存在逻辑问题。

[root@teacher ~]# [ -d "$path" ] || echo "Not dir"     推荐使用[     ]来替代 test 命令，使用代码更加易读。
Not dir
```

注意 [空格 -d "$var" 空格]

文件类型的检测

-e -f -d -b -c -S -P -L

文件属性的检测

-s -r -w -x -u -g -k

文件比较

-nt -ot -ef

整数的运算比较

-eq -ne -gt -lt -ge -le

变量检测

-z -n = !=

多重判断条件的连接

-a -o !

解释：

文件类型的判断

-e 是否存在

-d 是否目录

-f 是否文件

-b 是否块设备

-c 是否字符设备

-S 是否 socket

-P 是否管道文件

-L 是否链接文件

文件属性的判断

-s 文件大小大于 0

-r 是否具有 r 权限

-w 是否具有 w 权限

-x 是否具有 x 权限

-u 是否设置了 suid

-g 是否设置了 sgid

-k 是否设置了 t 位

文件比较

-nt 前一个文件比后一个文件新

-ot 前比后旧

-ef 有相同的设备和 i n o d e 号

整数运算的比较

-eq 两个数值相等

-ne 不相等

-gt 前比后大

-lt 前比后小

-ge 大于等于

-le 小于等于

变量检测

-z 变量为空

-n 变量不为空

= 变量等于某个值

!= 变量不等于某个值

多重判断的连接。

-a 两个条件同时成立

-o 两个条件满足一个即可

! 条件取反

2) 条件判断

- if 关键字的用法

```
if 条件表达式 ; then
    条件成立时执行的命令
elif 条件表达式二; then
    条件二成立时执行的命令
else
    不匹配以上条件时执行的命令
fi
```

示例：

```
#!/bin/bash
read -p "Please input (Y/N): " yn
if [ "$yn" = "y" -o "$yn" = "Y" ];then
        echo "OK"
elif [ "$yn" = "n" -o "$yn" = "N" ];then
        echo "exit"
else
        echo "sorry"
fi
```

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

（六） 循环语句

- for 关键字

```
for 变量 in 循环体
        do
            命令
        done
```

循环体可以是多个字符串，连续的数值，文本的每一行，命令的结果的标准输出等。

示例：

```
#!/bin/bash
for i in {1..10}
        do
            echo $i
        done
```

示例：

```
#!/bin/bash
for ((i=10;i>=0;i--))
        do
            echo $i
            sleep 1
        done
```

- while 关键字

```
while 条件成立时
        do
            命令
```

```
done
```

示例：

```
#!/bin/bash
i=10
while [ "$i" -ge 0 ]
        do
                echo "$i"
                let i--
        done
```

可以使用 ：或 true 或 ［ 1 －gt 0 ］等条件形成死循环。写永久循环脚本时注意脚本的暂停（sleep）

- until 关键字

```
until 条件不成立时
        do
            命令
        done
```

示例：

```
#!/bin/bash
echo -n "倒计时："
tput sc

i=10
until [ $i -eq 0 ]
        do
                tput rc
                tput ed
```

```
                echo -n    $i
                let i--
                sleep 1
        done
echo
```

- select 关键字

```
select  变量    in    多个变量值
        do
            命令
        done
```

最常用在和 case 进行配合。

- 中断   break   continue

示例：

```
#!/bin/bash
for i in {1..10}
        do
                echo "$i"
                [ $i -eq 5 ] && break
        done
```

示例：

```
#!/bin/bash
for i in {1..10}
        do
                [ $i -eq 5 ] && continue
                echo "$i"
        done
```

- shift 关键字

作用：拿掉脚本的参数

示例：

```
#!/bin/bash
until [ $# -le 0 ]
        do
                echo "The frist arg:    $1 counts: $#"
                shift 1
        done
```

任务四　shell 脚本常用命令使用技巧

(一)　cat 命令

1)　例出文件内容

```
[root@teacher shell]# cat 1.sh
```

-n 显示行号

-s 不输出多行空行

-T 将跳格字符显示为^I

2)　合并文件的内容

```
[root@teacher shell]# cat 1.sh 2.sh > 3.sh
```

3)　配合<<在脚本中进行格式输出

```
#!/bin/bash
cat <<EOF
    ###############
```

```
    #       ^ ^       #
    #         -       #
    ###############
EOF

cat > file.txt << eof

[root@teacher shell]# echo "Test stdin"|cat - file.txt
```

对于-的使用技巧：

```
# tar -zcf - test01 |tar -zxf - -C /smbdir
```

（二） head 命令

```
[root@teacher shell]# head /etc/passwd
-c,  --bytes=[-]K     显示每个文件的前 K 字节内容；
                如果附加"-"参数，则除了每个文件的最后 K 字节数据外
                显示剩余全部内容
  -n, --lines=[-]K    显示每个文件的前 K 行内容；
                如果附加"-"参数，则除了每个文件的最后 K 行外显示
                剩余全部内容
  -q, --quiet, --silent 不显示包含给定文件名的文件头
  -v, --verbose       总是显示包含给定文件名的文件头

[root@teacher shell]# cat -n /etc/passwd |head -n 5
[root@teacher shell]# cat -n /etc/passwd |head -n -5
[root@teacher shell]# cat -n /etc/passwd |head -n 5 |tac
```

（三） tail 命令

```
[root@teacher shell]# cat -n /etc/passwd |tail

-c, --bytes=K          输出最后 K 字节；另外，使用-c +K 从每个文件的
                       第 K 字节输出
   -f, --follow[={name|descriptor}]
          即时输出文件变化后追加的数据。
-n, --lines=K

[root@teacher shell]# cat -n /etc/passwd |tail -n ＋5
[root@teacher shell]# tail -f 1.sh
```

（四） find 命令

find 命令选项非常多，详请参考 man find

find [-H] [-L] [-P] [-D debugopts] [-Olevel] [path...] [expression]

-P Never follow symbolic links. default

-L Follow symbolic links. 跟踪目录的链接

-H Do not follow symbolic links, except while processing the command line arguments.

-P -L -H 是用来控制符号连接的处理

find /path 不加任何条件时，结果会显示出/path 目录下的所有文件，与 ls 不同的是，find 的结果带有路径。

```
[root@teacher ~]# find  /var/mail
```

```
/var/mail
[root@teacher ~]# find -L /var/mail
/var/mail
/var/mail/demo
/var/mail/rpc
```

OPTIONS ： 查找选项

-mindepth 最小的搜索的目录深度

-maxdepth 最大的搜索的目录深度

```
[root@teacher shell]# find . -name 1.sh
./1/2/3/1.sh
./1/2/1.sh
./1.sh
[root@teacher shell]# find . -maxdepth 1 -name 1.sh
./1.sh
[root@teacher shell]# find . -maxdepth 2 -name 1.sh
./1.sh
[root@teacher shell]# find . -maxdepth 3 -name 1.sh
./1/2/1.sh
./1.sh
```

TESTS ： 查找条件

+n for greater than n, 大于 n

-n for less than n, 小于 n

n for exactly n. 匹配 n

-amin n 文件在 n 分钟之前被访问过

-atime n 文件在 n 天之前被访问过 （当 n=0，指的是 1 天之内，当 n=1,

指的是 1 天（24h）以前 2 天之内，当 n=+1,指的是大于 1 天之前的所有。

-cmin n

-ctime n　文件在 n 天之前属性被修改过

-mmin n

-mtime n　文件内容 n 天之前被修改过

-newer file

-anewer file

-cnewer file

通过修改时间，访问时间，属性修改时间来和 file 进行比较更新的文件。

-empty　查找空文件

-executable 查找可执行文件

-fstype type 指定查找的文件系统类型

-uid n　通过用户 uid 查找文件

-gid n　通过组 gid 查找文件

find / -fstype ext4 -gid 500

-user uname

-group gname

通过用户名和组名去查找

-nouser

-nogroup

查找没有用户和组的文件

-name　通过名字查找

-iname

-lname

-ilname

通过链接名查找

其中 i 开头的选项表示忽略文件名中的大小写，下面的类似。

-inum n

-samefile name

通过比较 inode 来查找相同的文件。

-regex pattern　通过正则表达式去匹配文件名，这个匹配是要包含整个路径的，而不仅仅是匹配文件名。

-iregex pattern

-links n 根据文件的链接数查找

-lname pattern 通过名字查找链接

-wholename pattern

-iwholename pattern

-path pattern 通过路径去查找

-perm mode　通过权限去查找

-perm -mode

-perm /mode

-perm +mode　（等同于/mode，　过时的选项）

-readable

-writable

-size n[cwbkMG]

通过文件东西查找。可以使用（n +n -n）

-type c

通过文件类型去查找

-xtype c

结果中包含链接文件

```
[root@teacher 1]# find /dev/ -type b -ls
[root@teacher 1]# find /dev/ -xtype b -ls
```

-context pattern

通过 SELINUX 上下文查找

ACTIONS ： 查找到匹配项后的操作

-delete 删除 一定要使用匹配条件，并不能和-prune 一起使用，自动打开－depth

-exec command ; 执行命令

```
[root@teacher 1]# find /etc  -name "grub*" -exec basename {} \;  > 1.txt
```

-exec command {} +

-execdir command ;

-execdir command {} +

-ok command ; 每次询问

-okdir command ;

-print　将匹配到的结果打印到屏幕，并使用换行，默认操作。

-print0 将匹配到的结果打印到屏幕，并使用 null 字符替代换行。可以配合 xargs

-printf format

-ls　将匹配到的文件用 ls -l 方式列出。

-fls file 将列出的内容写入到一个文件

-fprint file　同上，区别是这个只打印文件名，没有 ls -l。

-fprint0 file

-fprintf file format

-prune　查找过程中搜索到的文件是一个目录，那么不进入该目录查找。当使用 depth 失效，也不和 delete 一起使用

-quit

举例：

```
find .  \( -name "*.txt" -o -name "*.pdf" \)
find . -type f -name "*.swp" -delete
find . -type f -user root -exec chown webuser {} \;
find path -type f -name "*.mp3" -exec mv {} target_dir \;
find /tmp -name core -type f -print | xargs /bin/rm -f
```

（五） xargs 命令

将标准输入转换成其他命令的命令行参数

示例：

```
[root@teacher 1]# cat test.txt
1234
2 3 4
5678
9 1

[root@teacher 1]# cat test.txt |xargs
1234 2 3 4 5678 9 1

[root@teacher 1]# cat test.txt |xargs -n 3
1234 2 3
4 5678 9
1

[root@teacher 1]# echo "123x456x789x9876" |xargs -d "x" -n 2
123 456
789 9876
```

调试：

```
[root@teacher 1]# cat arg.sh
#!/bin/bash
echo "$*""#'

[root@teacher 1]# ./arg.sh arg1
arg1#
[root@teacher 1]# ./arg.sh arg1 arg2
arg1 arg2#
[root@teacher 1]# ./arg.sh arg1 arg2 arg3
```

```
arg1 arg2 arg3#

[root@teacher 1]# cat arg.txt
arg1
arg2
arg3

[root@teacher 1]# cat arg.txt |xargs ./arg.sh
arg1 arg2 arg3#

[root@teacher 1]# cat arg.txt |xargs -n 1 ./arg.sh
arg1#
arg2#
arg3#

[root@teacher 1]# ls |cp /tmp
cp: 在"/tmp" 后缺少了要操作的目标文件
请尝试执行"cp --help"来获取更多信息。

[root@teacher 1]# ls |xargs -I {}    cp    {} /tmp

{}可以是自定义的任意字符。
```

主要用来和 find 命令进行配合。详见 find 的示例。

(六) tr 命令

从标准输入中替换或缩减和/或删除字符，并将结果写到标准输出。

tr [选项]... ste1 [ste2]

基本用法举例：

```
[root@teacher 1]# echo abc|tr 'a-z' 'A-Z'
ABC

[root@teacher 1]# echo abc|tr 'abc' '12'
122

[root@teacher 1]# echo abc|tr 'abc' '1'
111



[root@teacher 1]# echo abc|tr 'abc' '12345'
123

[root@teacher 1]# echo abc|tr 'a' '12345'
1bc
[root@teacher 1]# echo abc|tr 'ab' '12345'
12c
[root@teacher 1]# echo abc|tr 'abcde' '12345'
123
[root@teacher 1]# echo abc|tr 'bcde' '12345'
a12

[root@foundation0 shells]# echo abc    | tr    'acb' '123'
132
```

选项：

-c, -C, --complement　　首先保留 SET1

-d, --delete　　删除匹配 SET1 的内容，并不作替换

-s, --squeeze-repeats　如果匹配于 SET1 的字符在输入序列中存在连续的重复，在替换时会被统一缩为一个字符的长度

-t, --truncate-set1　　先将 SET1 的长度截为和 SET2 相等

```
[root@teacher 1]# echo abc|tr -c 'bcde' '12345'
5bc5
```

解释：bcde 和 1234 匹配替换，而加了-c， bc 会保留下来，a 和\n 被替换为 5，得到结果 5bc5

```
[root@teacher 1]# echo "hello 1 char 2 next 4"|tr -d -c '0-9 \n'
 1  2  4

[root@teacher 1]# echo abc | tr -d 'b'
ac

[root@teacher 1]# echo "aaabacdcc" |tr -s 'abc'
abacdc
[root@teacher 1]# echo "aaabacdcc" |tr -s 'abc' '123'
1213d3

[root@teacher 1]# echo "aaabacdcc" |tr -t 'abcd' '123'
111213d33
```

思考：

```
cat sum.txt
1
2
32
24
5
```

16

使用 tr 命令计算出它们的和。

```
[root@teacher 1]# cat sum.txt |echo $((`tr '\n' '+'`0))
[root@teacher 1]# cat sum.txt |echo $[ $(tr '\n' '+') 0 ]
```

tr 可以使用字符类

比如：

\NNN　八进制值为 NNN 的字符(1 至 3 个数位)

\\　反斜杠

\a　终端鸣响

\b　退格

\f　换页

\n　换行

\r　回车

\t　水平制表符

\v　垂直制表符

字符 1-字符 2　从字符 1 到字符 2 的升序递增过程中经历的所有字符

[字符*] 在 SET2 中适用，指定字符会被连续复制直到吻合 SET1 的长度

[字符*次数] 对字符执行指定次数的复制，若次数以 0 开头则被视为八进制数

[:alnum:] 所有的字母和数字

[:alpha:] 所有的字母

[:blank:] 所有呈水平排列的空白字符

[:cntrl:] 所有的控制字符

[:digit:] 所有的数字

[:graph:] 所有的可打印字符，不包括空格

[:lower:] 所有的小写字母

[:print:] 所有的可打印字符，包括空格

[:punct:] 所有的标点字符

[:space:] 所有呈水平或垂直排列的空白字符

[:upper:] 所有的大写字母

[:xdigit:] 所有的十六进制数

[=字符=] 所有和指定字符相等的字符

仅在 SET1 和 SET2 都给出，同时没有-d 选项的时候才会进行替换。

仅在替换时才可能用到-t 选项。如果需要 SET2 将被通过在末尾添加原来的末字符的方式

补充到同 SET1 等长。SET2 中多余的字符将被省略。只有[:lower:] 和[:upper:]

以升序展开字符；在用于替换时的 SET2 中以成对表示大小写转换。-s 作用于 SET1，既不

替换也不删除，否则在替换或展开后使用 SET2 缩减。

（七） cut 命令

基本用法举例

```
[root@teacher 1]# cat /etc/passwd |cut -b 5
[root@teacher 1]# cat /etc/passwd |cut -b 1-5
```

用法：cut [选项]... [文件]...

从每个文件中输出指定部分到标准输出。

长选项必须使用的参数对于短选项时也是必需使用的。

-b, --bytes=列表 只选中指定的这些字节

-c, --characters=列表 只选中指定的这些字符

-d, --delimiter=分界符使用指定分界符代替制表符作为区域分界

-f, --fields=LIST select only these fields; also print any line

that contains no delimiter character, unless
the -s option is specified

-n with -b: don't split multibyte characters

--complement 补全选中的字节、字符或域

-s, --only-delimited 不打印没有包含分界符的行

--output-delimiter=字符串使用指定的字符串作为输出分界符，默认采用输入的分界符

--help 显示此帮助信息并退出

--version 显示版本信息并退出

仅使用 f -b, -c 或-f 中的一个。每一个列表都是专门为一个类别作出的，或者您可以用逗号隔

开要同时显示的不同类别。您的输入顺序将作为读取顺序，每个仅能输入一次。

每种参数格式表示范围如下：

N 从第 1 个开始数的第 N 个字节、字符或域

N- 从第 N 个开始到所在行结束的所有字符、字节或域

N-M 从第 N 个开始到第 M 个之间(包括第 M 个)的所有字符、字节或域

-M从第 1 个开始到第 M 个之间(包括第 M 个)的所有字符、字节或域

当没有文件参数，或者文件不存在时，从标准输入读取

```
[root@teacher 1]# head -3  /etc/passwd |cut -d ":" -f 1
root
bin
daemon
[root@teacher 1]# head -3  /etc/passwd |cut -d ":" -f 2
x
```

```
x
x
[root@teacher 1]# head -3    /etc/passwd |cut -d ":" -f 1,3
root:0
bin:1
daemon:2
[root@teacher 1]# head -3    /etc/passwd |cut -d ":" -f 1-3
root:x:0
bin:x:1
daemon:x:2



[root@teacher 1]# cat 1.txt
1243 456 78 ab
ab cc 687 cc
abcdef
cc aa cc
aa cc
[root@teacher 1]# cat 1.txt |cut -d " " -f 4
ab
cc
abcdef



[root@teacher 1]# cat 1.txt |cut -d " " -f 4 -s
ab
cc
```

（八） sort 命令

基本用法举例：

```
[root@teacher 1]# cat sum.txt
1
2
32
24
5
16
[root@teacher 1]# cat sum.txt |sort
1
16
2
24
32
5
[root@teacher 1]# cat sum.txt |sort -n
1
2
5
16
24
32
```

用法：sort [选项]... [文件]...

或：sort [选项]... --files0-from=F

串联排序所有指定文件并将结果写到标准输出。

长选项必须使用的参数对于短选项时也是必需使用的。

排序选项：

-b, --ignore-leading-blanks 忽略前导的空白区域

-f, --ignore-case 忽略字母大小写

-g, --general-numeric-sort 按照常规数值排序

-i, --ignore-nonprinting 只排序可打印字符

-M, --month-sort 比较 (未知) < "一月" < ... < "十二月"

-h, --human-numeric-sort 使用易读性数字(例如： 2K 1G)

-n, --numeric-sort 根据字符串数值比较

-R, --random-sort 根据随机 hash 排序

--random-source=文件 从指定文件中获得随机字节

-r, --reverse 逆序输出排序结果

--sort=WORD 按照 WORD 指定的格式排序：

一般数字-g，高可读性-h，月份-M，数字-n，

随机-R，版本-V

-V, --version-sort 在文本内进行自然版本排序

-k, --key=位置 1[,位置 2] 在位置 1 开始一个 key，在位置 2 终止(默认为行尾)

-o, --output=文件 将结果写入到文件而非标准输出

-t, --field-separator=分隔符 使用指定的分隔符代替非空格到空格的转换

-u, --unique 配合-c，严格校验排序；不配合-c，则只输出一次排序结果

-z, --zero-terminated 以 0 字节而非新行作为行尾标志

```
[root@teacher 1]# cat /etc/passwd |cut -d":" -f1,3|sort -t ":" -k 2 -n
[root@teacher 1]# cat /etc/passwd |cut -d":" -f7|sort -u
```

```
/bin/bash
/bin/sync
/sbin/halt
/sbin/nologin
/sbin/shutdown
```

（九） uniq 命令

作用：去重

基本用法举例：

```
[root@teacher 1]# cat /etc/passwd |cut -d":" -f7|sort|uniq -c|sort -nr
      31 /sbin/nologin
       2 /bin/bash
       1 /sbin/shutdown
       1 /sbin/halt
       1 /bin/sync
```

用法：uniq [选项]... [文件]

从输入文件或者标准输入中筛选相邻的匹配行并写入到输出文件或标准输出。

不附加任何选项时匹配行将在首次出现处被合并。

长选项必须使用的参数对于短选项时也是必需使用的。

-c, --count 在每行前加上表示相应行目出现次数的前缀编号

-d, --repeated 只输出重复的行

-D, --all-repeated[=delimit-method 显示所有重复的行

delimit-method={none(default),prepend,separate}

以空行为界限

-f, --skip-fields=N 比较时跳过前 N 列

-i, --ignore-case 在比较的时候不区分大小写

-s, --skip-chars=N 比较时跳过前 N 个字符

-u, --unique 只显示唯一的行

-z, --zero-terminated 使用'\0'作为行结束符，而不是新换行

-w, --check-chars=N 对每行第 N 个字符以后的内容不作对照

--help 显示此帮助信息并退出

--version 显示版本信息并退出

提示：uniq 不会检查重复的行，除非它们是相邻的行。

（十） wc 命令

用法：wc [选项]... [文件]...

或：wc [选项]... --files0-from=F

输出每个指定文件的行数、单词计数和字节数，如果指定了

多于一个文件，继续给出所有相关数据的总计。如果没有指定

文件，或者文件为"-"，则从标准输入读取数据。

-c, --bytes 输出字节数统计

-m, --chars 输出字符数统计

-l, --lines 输出行数统计

--files0-from=文件 从指定文件读取以 NUL 终止的名称，如果该文件被

指定为"-"则从标准输入读文件名

-L, --max-line-length 显示最长行的长度

-w, --words 显示单词计数

基本用法举例：

```
[root@teacher 1]# cat /etc/passwd |wc -l
36
```

（十一）grep 命令

基本用法示例：

```
[root@teacher 1]# cat /etc/passwd |grep "root"
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
```

用法: grep [选项]... PATTERN [FILE]...

在每个 FILE 或是标准输入中查找 PATTERN。

默认的 PATTERN 是一个基本正则表达式(缩写为 BRE)。

例如: grep -i 'hello world' menu.h main.c

当 PATTERN 使用正则表达式的时候，要用单引号。

示例：

```
[root@teacher 1]# cat /etc/passwd |grep '^root'
root:x:0:0:root:/root:/bin/bash
[root@teacher 1]# cat /etc/passwd |grep '\<bin\>'
```

正则表达式详见下一项目内容。

正则表达式选择与解释:

-E, --extended-regexp PATTERN 是一个可扩展的正则表达式(缩写为 ERE)

-F, --fixed-strings PATTERN 是一组由断行符分隔的定长字符串。

-e, --regexp=PATTERN 用 PATTERN 来进行匹配操作

-f, --file=FILE 从 FILE 中取得 PATTERN

-i, --ignore-case 忽略大小写

-w, --word-regexp 强制 PATTERN 仅完全匹配字词

-x, --line-regexp 强制 PATTERN 仅完全匹配一行

-z, --null-data 一个 0 字节的数据行，但不是空行

Miscellaneous:

-s, --no-messages　不显示错误信息

-v, --invert-match 取反

Output control:

-m, --max-count=NUM　达到 NUM 次匹配时停止

-n, --line-number　显示行号

-H, --with-filename　显示匹配结果的文件名

-h, --no-filename 不显示匹配结果的文件名

-o, --only-matching 显示匹配的 PATTERN

-q, --quiet, --silent 安静模式，没有任何一般输出

-a, --text --binary-files=text

-d, --directories=ACTION 如何操作目录，ACTION is `read', `recurse', or `skip'

-D, --devices=ACTION 如何操作设备

-R, -r, --recursive　目录递归，等同于 --directories=recurse

--include=FILE_PATTERN　只匹配表达式表示的文件

--exclude=FILE_PATTERN　排除表达式表示的文件

--exclude-from=FILE　从文件读取要排除的文件或目录

--exclude-dir=PATTERN 从表达式获取要排除的目录

-L, --files-without-match　打印出内容不匹配的文件名

-l, --files-with-matches　打印出内容匹配的文件名

-c, --count 打印出每个文件匹配到内容的次数

-Z, --null　print 0 byte after FILE name

Context control:

-B, --before-context=NUM 打印出匹配内容行前 NUM 行

-A, --after-context=NUM　打印出匹配内容行后 NUM 行

-C, --context=NUM 打印出多少行

--color[=WHEN], 颜色高亮显示

--colour[=WHEN] use markers to highlight the matching strings;
WHEN is `always', `never', or `auto'

‘egrep’即‘grep -E’。‘fgrep’即‘grep -F’。

举例：

```
[root@teacher shell]# grep -l 'hostname' /etc/*

[root@teacher 1]# grep    'root' /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
[root@teacher 1]# grep -e 'root' /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
[root@teacher 1]# grep -e    'root' -e 'demo' /etc/passwd
demo:x:500:500:demo:/home/demo:/bin/bash
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin

[root@teacher 1]# grep -q 'root' /etc/passwd
[root@teacher 1]# echo $?
0
[root@teacher 1]# grep -q 'rootxx' /etc/passwd
[root@teacher 1]# echo $?
1

[root@teacher 1]# grep -c 'root' /etc/passwd
```

```
2


[root@teacher 1]# grep -w 'bin' /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
demo:x:500:500:demo:/home/demo:/bin/bash


[root@teacher 1]# grep -R -i    -H 'ServerName' /etc
```