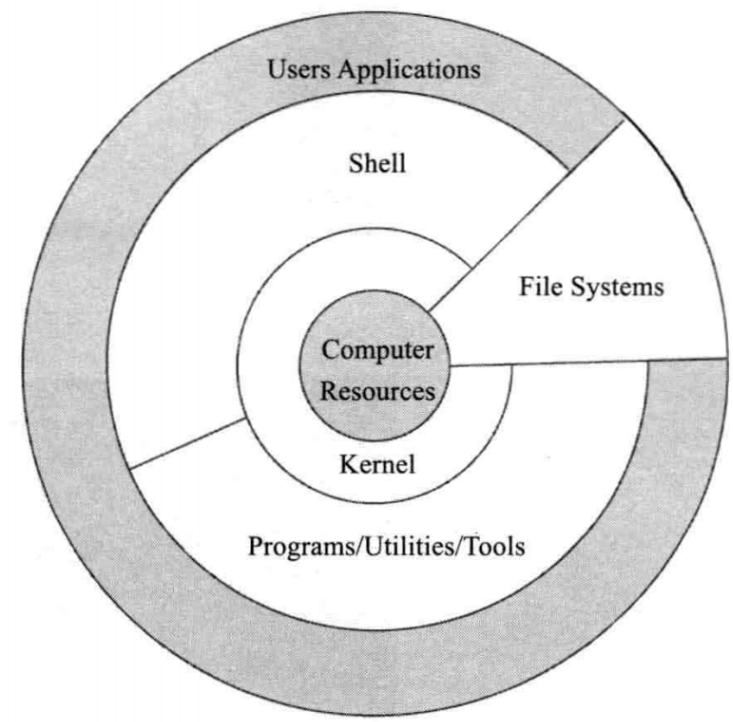
Shell是一个用C语言编写的程序，它是用户使用Linux的桥梁。Shell既是一种命令语言，又是一种程序设计语言。

Shell是指一种应用程序，这个应用程序提供了一个界面，用户通过这个界面访问操作系统内核的服务。

shell是一种解释型语言，这种语言经过编写后不经过任何编译就可以执行，是一种脚本语言。和编译型语言是执行前翻译不同，解释型语言的程序是执行时翻译，所以效率要差一些。



CentOS系统中支持很多shell,可以通过查看/etc/shells文件，查看所支持的shell,目前大多数的Linux基本都使用bash

```
[root@sunday-test ~]# cat /etc/shells
/bin/sh
/bin/bash
/sbin/nologin
/usr/bin/sh
/usr/bin/bash
/usr/sbin/nologin
/bin/tcsh
/bin/csh
/bin/zsh
```

bash shell有两种工作模式：互动模式和脚本模式。脚本模式效率更高，可以实现自动化。

编写第一个shell脚本：

```
[root@sunday-test shell-script]# cat HelloWorld.sh
#!/bin/bash                    //是一个标记，告诉系统执行这个文件需要的解释器
# this line is a comment    //  “#”号开头的行代表注释
echo "Hello World"
```

运行脚本有两种方法：

1、使用bash命令执行

```
[root@sunday-test shell-script]# bash HelloWorld.sh
Hello World
```

“.”号和source命令也都可以执行脚本，且不需要可执行权限

```
[root@sunday-test shell-script]# . HelloWorld.sh
Hello World
[root@sunday-test shell-script]# source HelloWorld.sh
Hello World
```

2、给脚本添加可执行权限，然后直接就可以执行了

```
[root@sunday-test shell-script]# chmod o+x HelloWorld.sh
[root@sunday-test shell-script]# ./HelloWorld.sh
Hello World
```

变量

顾名思义，变量就是其值可以变化的量。从变量的本质来说，变量名是指向一片用于存储数据的内存空间。变量有局部变量、环境变量之分。在脚本中，往往需要使用变量来存储有用信息，比如文件名、路径名、数值等，通过这些变量可以控制脚本的运行行为。

1、局部变量：是指在某个shell中生效的变量，对其他shell来说无效，局部变量的作用域被限定在声明它们的shell中，可以使用local内建命令来“显式”的声明局部变量，但仅限于函数内使用。

2、环境变量通常又称“全局变量”，以区别于局部变量。在shell脚本中，变量默认就是全局的，为了让子shell继承当前shell的变量，可以使用export命令将其定义为环境变量。

bash中默认包含有几十个预设的环境变量，下面介绍一些常用的：

BASH: Bash shell的全路径。

```
[root@sunday-test ~]# echo $BASH
/bin/bash
```

BASH_VERSION : bash shell的版本

```
[root@sunday-test ~]# echo $BASH_VERSION
4.2.46(1)-release
```

EUID：记录当前用户的UID。

```
[root@sunday-test ~]# echo $EUID
0
```

FUNCNAME:在用户函数体内部，记录当前函数体的函数名。

```
[root@sunday-test shell-script]# cat funcname.sh
```

```
#!/bin/bash
funcname() {
   echo $FUNCNAME
}
funcname
[root@sunday-test shell-script]# source funcname.sh
funcname
```

HISTCMD：记录下一条命令在history命令中的编号

HISTFILE：记录history命令记录文件的位置

HISTFILESIZE：设置HISTFILE文件记录命令的行数

HISTSIZE：设置命令缓冲区的大小

HOSTNAME：设置主机名

HOSTTYPE：展示主机的架构

```
[root@sunday-test shell-script]# echo $HOSTTYPE
x86_64
```

MACHTYPE：主机类型的GNU标识，这种标识有统一的结构。一般来说是“主机架构-公司-系统-gnu”

```
[root@sunday-test shell-script]# echo $MACHTYPE
x86_64-redhat-linux-gnu
```

LANG：设置当前系统的语言环境。

```
[root@sunday-test shell-script]# echo $LANG
zh_CN.UTF-8
[root@sunday-test shell-script]# LANG=en_US.UTF-8
[root@sunday-test shell-script]# echo $LANG
en_US.UTF-8
```

PWD：记录当前目录

OLDPWD：记录之前目录

PATH：代表命令的搜索路径，非常重要

表 13-2　一些预设变量

变量名	用　途
BASH_ENV	一般该值为空。如果该变量在调用脚本时已经设置，它的值将被展开，并用作在执行脚本前读取的启动文件名
BASH_VERSINFO	一个只读变量数组，保存 bash 的版本信息
COLUMNS	决定 select 内建命令打印选择列表时的宽度
COMP_LINE	当前命令行
COMP_POINT	相对于当前命令起点的当前光标位置
COMP_WORDS	由当前命令行中单个词组成的变量数组
DIRSTACK	保存当前目录栈内容的变量数组
FIGNORE	由冒号分隔的在补全文件名时要忽略的后缀列表
GLOBIGNORE	由冒号分隔的模板列表，定义在文件名展开时忽略的文件名
GROUPS	一个数组变量，包含当前用户作为成员组的列表
HISTCONTROL	定义一个命令是否加入历史列表中
IGNOREEOF	控制 Shell 接收 EOF 字符作为独立输入的行为
INPUTRC	readline 初始化文件的名称，取代默认值 /etc/inputrc
LC_ALL	如果该变量设置了，则这个变量将覆盖 LANG 的值
LC_CTYPE	决定在文件名展开和模板匹配里字符的解释和字符集的行为
LC_MESSAGES	该变量决定用于转换由 $ 引导的双引号字符串的区域
LC_NUMERIC	该变量决定数字格式化的本地类别
LINENO	当前执行的脚本或者 Shell 函数的行数
LINES	决定内建命令 select 打印选择列表的列长度
MAILCHECK	Shell 从 MAILPATH 或 MAIL 变量指定的文件中检查邮件的频率
OPTERR	如果设置成 1，bash 显示内建命令 getopts 生成的错误信息
PIPESTATUS	最近运行过的前台管道进程的退出状态值的列表
PPID	Shell 父进程的进程 ID
PS3	这个变量的值被用作 select 命令的提示符
PS4	在命令行前打印的提示符
RANDOM	生成一个 0~32767 的随机整数
REPLY	内建命令 read 的默认值
SECONDS	Shell 运行的秒数
SHELLOPTS	由冒号分隔的 Shell 已经启用的选项列表
SHLVL	每新增一个 Shell 进程，该值就增加 1
TMOUT	作为内建命令 read 的默认超时时间。当 Shell 处于交互状态时，这个值表示等待在基本提示串后输入的秒数
UID	当前用户的真实用户 ID

变量命名

shell中的变量必须以字母或者下划线开头，后面可以跟数字、字母和下划线，长度没有限制，区分大小写。

```
# 正确的变量命名
firstname
FIRSTNAME
_helloworld
big_data
Fullname
Person01
# 错误的变量命名
51play # 变量不能以数字开头
*badname # 变量不能以特殊字符开头
PS1 # 变量不能和 Shell 的预设变量名重名
for # 变量不能使用 Shell 的关键字
```

变量赋值和取值

定义变量：变量名=变量值

注意点1：变量名和变量值之间用等号紧紧相连，之间没有任何空格

[root@sunday-test ~]# name=john

[root@sunday-test ~]# name= john

bash: john: command not found...

注意点2：当变量值中有空格时必须用引号括起，否则会出现错误，可以是双引号，也可以是单引号

[root@sunday-test ~]# name="li si"

[root@sunday-test ~]# name=li si

bash: si: command not found...

Similar command is: 'ci'

变量的取值很简单，在变量名前加$号就可以了，严谨的方法是${}。建议用后者

[root@sunday-test ~]# echo $name

li si

[root@sunday-test ~]# echo ${name}

li si

取消变量

取消变量使用unset,后面跟变量名。函数也是可以被取消的，unset后面也是可以跟上函数名来取消函数的。

[root@sunday-test ~]# unset name

特殊变量

1、位置参数

shell中还有一些预先定义的特殊只读变量，它们的值只有在脚本运行时才能确定。

$0：代表脚本本身名字

$1----$9：第一个位置参数-------第9个位置参数

$#：脚本参数的个数总和

$@：表示脚本的所有参数

$* :表示脚本的所有参数

```
[root@sunday-test shell-script]# cat posion.sh
#!/bin/bash
echo "这个脚本的名字是：$0"
echo "参数一共有$#"
echo "参数的列表是: $@"
echo "参数的列表是: $*"
echo "第一个参数是: $1"
echo "第二个参数是: $2"
echo "第三个参数是：$3"
[root@sunday-test shell-script]# ./posion.sh a b c
这个脚本的名字是：./posion.sh
参数一共有3
参数的列表是: a b c
参数的列表是: a b c
第一个参数是: a
第二个参数是: b
第三个参数是：c
```

2、脚本或者命令返回值：$?

正常退出的命令和脚本应该返回值为0,任何非0的返回值都表示命令未正确退出或未正常执行

```
[root@sunday-test ~]# ifcofj
bash: ifcofj: 未找到命令...
[root@sunday-test ~]# echo $?
127
[root@sunday-test ~]# ping -c 1 192.168.5.23
PING 192.168.5.23 (192.168.5.23) 56(84) bytes of data.
From 192.168.5.141 icmp_seq=1 Destination Host Unreachable
--- 192.168.5.23 ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss, time 0ms
[root@sunday-test ~]# echo $?
1
```

数组

数组是一种特殊的数据结构，其中的每一项被称为一个元素，对于每个元素，都可以用索引的方式取出元素的值。使用数组的典型场景是一次性要记录很多类型相同的数据时（但不一定必须要相同）。比如，为了记录班级中所有人的数学成绩，如果不用数据来处理，那就只能定义所有人成绩的变量。shell中的数组对元素个数没有限制，但只支持一维数组。

1、数组定义

使用declare命令定义数组Array

```
[root@sunday-test ~]# declare -a Array
[root@sunday-test ~]# Array[0]=0
[root@sunday-test ~]# Array[1]=1
[root@sunday-test ~]# Array[2]="HelloWorld"
```

也可以在创建的同时赋值

```
[root@sunday-test ~]# declare -a Name=('john' 'sue')
[root@sunday-test ~]# Name[2]='wang'
```

更简单的方式来创建数组

```
[root@sunday-test ~]# Name=('john' 'sue')
```

还可以跳号赋值

```
[root@sunday-test ~]# Score=([3]=3 [5]=5 [7]=7)
```

2、数组操作

数组取值：格式为：${数组名[索引]}

```
[root@sunday-test ~]# echo ${Array[0]}
0
[root@sunday-test ~]# echo ${Array[2]}
HelloWorld
[root@sunday-test ~]# echo ${Name[1]}
sue
```

一次性取出数组所有的值：

```
[root@sunday-test ~]# echo ${Array[@]}
0 1 HelloWorld                  //得到的是以空格隔开的元素值
[root@sunday-test ~]# echo ${Array[*]}
0 1 HelloWorld              //输出的是一整个字符串
```

数组长度：即数组元素个数

利用“@”或“*”字符，可以将数组扩展成列表，然后使用“#”来获取数组元素的个数。

```
[root@sunday-test ~]# echo ${#Array[*]}
3
```

```
[root@sunday-test ~]# echo ${#Array[@]}
3
```

数组截取：可以截取某个元素的一部分，对象可以是整个数组或某个元素。

取出数组的第一个、第二个元素

```
[root@sunday-test ~]# echo ${Array[@]:1:2}
1 HelloWorld
```

取出第二个元素从第0个字符开始连续5个字符

```
[root@sunday-test ~]# echo ${Array[2]:0:5}
Hello
```

连接数组：将若干个数组进行拼接操作

```
[root@sunday-test ~]# Conn=(${Array[@]} ${Name[@]})
[root@sunday-test ~]# echo ${Conn[@]}
0 1 HelloWorld john sue
```

替换元素：将数组内某个元素的值替换成其他值。

```
[root@sunday-test ~]# Array=(${Array[@]/HelloWorld/HelloJohn})
[root@sunday-test ~]# echo ${Array[@]}
0 1 HelloJohn
```

取消数组或元素：使用unset命令

取消数组中的一个元素

```
[root@sunday-test ~]# unset Array[1]
[root@sunday-test ~]# echo ${Array[@]}
0 HelloJohn
```

取消整个数组

```
[root@sunday-test ~]# unset Array
[root@sunday-test ~]# echo ${Array[@]}
```

已经为空，数组已经不存在了

只读变量

只读变量又称常量，通过readonly内建命令创建，创建时需要赋值，并且之后无法修改。

```
[root@sunday-test ~]# readonly RO=100
[root@sunday-test ~]# RO=200
-bash: RO: 只读变量
```

转义和引用

shell中有很多特殊字符，会有特殊意义，但是有时候会造成麻烦，需要转义才可以使用，转义符号为“\”

```
[root@sunday-test ~]# echo 8 \* 8 =64
8 * 8 =64
[root@sunday-test ~]# echo \$Dollar
$Dollar
```

表 13-3 Shell 特殊字符

特殊字符	转义写法
'（单引号）	\'
"（双引号）	\"
*（星号）	*
%	\%
?	\?
\	\\
~	\~
`（反引号）	\`
+	\+
!	\!
#	\#
$	\$
&	\&
(	\(
)	\)
[	\[
]	\]
{	\{
}	\}
<	\<
>	\>
|	\\|
;	\;
/	\/

引用是指将字符串用某种符号括起来，以防止特殊字符被解析为其他意思。shell中一共有4种引用符，分别为双引号、单引号、反引号和转义符。双引号可以引用除$符号、反引号、转义符之外的所有字符；单引号可以引用所有字符；反引号则会将反引号中的内容解释为系统命令。

```
[root@sunday-test ~]# echo "current directory is $PWD"
current directory is /root
[root@sunday-test ~]# echo 'current directory is $PWD'
current directory is $PWD
```

命令替换：是指将命令的标准输出作为值赋给某个变量。

格式有反引号和$(),建议使用$()

```
[root@sunday-test ~]# TIME=`date +%F`
[root@sunday-test ~]# echo $TIME
2017-10-28
```

运算符：

shell中的运算符主要有比较运算符（用于整数比较）、字符串运算符（用于字符串测试）、文件操作运算符（用于文件测试）、逻辑运算符、算术运算符、位运算符、自增自减运算符等。

1、算术运算符

算术运算符指的是加、减、乘、除、余、幂等常见的算术运算，以及加等、减等、乘等、除等、余等复合算术运算。要特别注意的是，shell只支持整数计算，也就是说所有可能产生小数的运算都会舍去小数部分。

表 13-4　常规算术运算符

运算符	运算符举例	运算结果
+（加运算符）	1+1	2
-（减运算符）	2-1	1
*（乘运算符）	2*3	6
/（除运算符）	9/4	2
%（余运算符）	10%3	1
**（幂运算符）	2**3	8

表 13-5　复合算术运算符

运算符	运算符举例	变量 x 的运算结果
+=（加等运算符）	x=8;x+=2	10
-=（减等运算符）	x=8;x-=2	6
*=（乘等运算符）	x=8;x*=2	16
/=（除等运算符）	x=8;x/=2	4
%=（余等运算符）	x=8;x%=2	0

Bash shell 的算术运算有四种方式:

1：使用 expr 外部程式

```
加法 r=`expr 4 + 5`
echo $r
注意！’4’ ’+’ ’5’ 这三者之间要有空白
r=`expr 4 * 5` #错误
乘法 r=`expr 4 \* 5`
```

2：使用 $(( ))

```
r=$(( 4 + 5 ))
echo $r
```

3：使用 $[ ]

```
r=$[ 4 + 5 ]
echo $r
```

乘法

```
r=`expr 4 \* 5`
r=$(( 4 * 5 ))
r=$[ 4 * 5 ]
echo $r
```

除法

```
r=`expr 40 / 5`
r=$(( 40 / 5 ))
r=$[ 40 / 5 ]
echo $r
```

减法

```
r=`expr 40 - 5`
r=$(( 40 - 5 ))
r=$[ 40 - 5 ]
echo $r
```

求余数

```
r=$[ 100 % 43 ]
echo $r
```

乘幂 (如 2 的 3 次方)

```
r=$(( 2 ** 3 ))
r=$[ 2 ** 3 ]
echo $r
```

注：expr 沒有乘幂

4：使用let 命令

加法：

```
n=10
let n=n+1
echo $n
#n=11
```

乘法：

```
let m=n*10
echo $m
```

除法：

```
let r=m/10
echo $r
```

求余数：

```
let r=m%7
echo $r
```

乘幂：

```
let r=m**2
echo $r
```

虽然Bash shell 有四种算术运算方法，但并不是每一种都是跨平台的，建议使用expr。另外，我们在 script 中经常有加1操作，以下四法皆可：

```
m=$[ m + 1]
m=`expr $m + 1`
m=$(($m + 1))
let m=m+1
```

2、位运算符：

位运算符是基于内存中二进制数据的运算，也就是基于位的运算。

位运算的左移、右移元素其实就是整数在内存中的“左右移动”。其中左移运算符为<<,右移运算符为>>。

```
[root@sunday-test ~]# let "a=4<<2"       //4左移2位
[root@sunday-test ~]# echo $a
16
[root@sunday-test ~]# let "b=4>>2"        //4右移2位
[root@sunday-test ~]# echo $b
1
```

与运算（&）,也就是取小，只有对应的二进制值都为1的时候，结果才为1

```
[root@sunday-test ~]# let "c=192&255"
[root@sunday-test ~]# echo $c
192
```

按位或运算（|）,是将两个整数写成二进制的形式，然后同位置相比较，只要对应的位置有1，结果就是1。

```
[root@sunday-test ~]# let "d=8|4"
```

```
[root@sunday-test ~]# echo $d
12
```

按位异或运算（^）,是将两个整数写成二进制的形式，然后同位置比较，只要相同，结果为0，否则为1。

```
[root@sunday-test ~]# let "e=10^3"
[root@sunday-test ~]# echo $e
9
```

自增自减

自增自减运算主要包括前置自增、前置自减、后置自增、后置自减等。前置自增或自减操作会首先修改变量的值，然后再将变量的值传递出去；后置自增或后置自减则会首先将变量的值传递出去，然后再修改变量的值。自增符号为“++”，自减符号为“--”，操作对象只能是变量，不能是常数或表达式。

```
[root@localhost ~]# cat add_minus.sh
#!/bin/bash
Add_01=10
Add_02=10
#Add_01 前置自增
# 也就是先将 Add_01 自增 1 变为 11，然后赋值给 Add_03，即为 11
let "Add_03=(++Add_01)"

#Add_02 后置自增
# 也就是先将当前值赋给 Add_04，即 10，然后 Add_02 自增 1，即为 11
let "Add_04=(Add_02++)"
# 打印各变量的值
# 按照上面的计算方式，Add_01、Add_02、Add_03 为 11，Add_04 为 10
echo Add_01 is:$Add_01
echo Add_02 is:$Add_02
echo Add_03 is:$Add_03
echo Add_04 is:$Add_04

[root@localhost ~]# bash add_minus.sh
Add_01 is:11
Add_02 is:11
Add_03 is:11
Add_04 is:10
```

特殊字符

shell中除了普通字符外，还有很多具有特殊含义和功能的字符，在使用它们时要特别注意其含义和作用。

1、通配符

通配符用于模式匹配，常见的通配符有*、？和用[]括起来的字符序列。

*：代表任意长度的字符串。例如：a*可以匹配以a开头的任意长度的字符串，但是不包括点号和斜线号。也就是说a*不能匹配abc.txt。

？：用于匹配任一单个字符。

[]：代表匹配其中的任意一个字符，比如[abc]代表匹配a或则b或则c，[]中可以用“-”表明起止，比如[a-c]等同于[abc]。*和?在[]中则变成了普通字符，没有通配的功效。

2、引号：包括单引号和双引号。

双引号中的字符除了“$”、“\”、反引号依然保留其特殊用途外，其余字符都作为普通字符处理。

单引号中的字符都作为普通字符处理

3、注释符

"#"代表注释，除了”#！“,其表示某个解释器的路径，且必须在整个脚本的第一行。

4、大括号

变量扩展。引用变量，又叫变量扩展，例如变量VAR，可以使用${VAR}引用。

表 13-6　大括号的变量扩展

表达式	作　用
${VAR}	取出变量 VAR 的值
${VAR:-DEFAULT}	如果 VAR 没有定义，则以 $DEFAULT 作为其值
${VAR:=DEFAULT}	如果 VAR 没有定义，或者值为空，则以 $DEFAULT 作为其值
${VAR+VALUE}	如果定义了 VAR，则值为 $VALUE，否则为空字符串
${VAR:+VALUE}	如果定义了 VAR 并且不为空值，则值为 $VALUE，否则为空字符串
${VAR?MSG}	如果 VAR 没有被定义，则打印 $MSG
${VAR:?MSG}	如果 VAR 没有被定义或未赋值，则打印 $MSG
${!PREFIX*} ${!PREFIX@}	匹配所有以 PREFIX 开头的变量
${#STR}	返回 $STR 的长度
${STR:POSITION}	从位置 $POSITION 处提取子串
${STR:POSITION:LENGTH}	从位置 $POSITION 处提取长度为 $LENGTH 的子串
${STR#SUBSTR}	从变量 $STR 的开头处开始寻找，删除最短匹配 $SUBSTR 的子串
${STR##SUBSTR}	从变量 $STR 的开头处开始寻找，删除最长匹配 $SUBSTR 的子串
${STR%SUBSTR}	从变量 $STR 的结尾处开始寻找，删除最短匹配 $SUBSTR 的子串
${STR%%SUBSTR}	从变量 $STR 的结尾处开始寻找，删除最长匹配 $SUBSTR 的子串
${STR/SUBSTR/REPLACE}	使用 $REPLACE 替换第一个匹配的 $SUBSTR
${STR//SUBSTR/REPLACE}	使用 $REPLACE 替换所有匹配的 $SUBSTR
${STR/#SUBSTR/REPLACE}	如果 $STR 以 $SUBSTR 开始，则用 $REPLACE 来代替匹配到的 $SUBSTR
${STR/%SUBSTR/REPLACE}	如果 $STR 以 $SUBSTR 结束，则用 $REPLACE 来代替匹配到的 $SUBSTR

通配符扩展

用于匹配多个排列组合的可能。还可以用于匹配不同的文件。

```
[root@localhost ~]# echo {x1,x2,x3}{y1,y2,y3}
x1y1 x1y2 x1y3 x2y1 x2y2 x2y3 x3y1 x3y2 x3y3
[root@localhost ~]# echo file{1..6}
```

file1 file2 file3 file4 file5 file6

[root@localhost ~]# echo file{4,8,9}

file4 file8 file9

语句块

大括号还能用于构造语句块，语句之间使用回车隔开。使用语句块的场景一般是在自定义函数中。

5、控制字符

控制字符即ctrl+key组合键一起使用，用于修改终端或文本显示。控制字符是交互式使用的，不能用于脚本中。

表 13-7　控制字符

组合键	作　用
Ctrl+B	退格但是不删掉前面的字符
Ctrl+C	终结当前前台作业
Ctrl+D	结束符，可用于退出当前 Shell 或结束当前输入
Ctrl+G	系统输出一声鸣叫
Ctrl+H	退格且删掉前面的字符
Ctrl+L	清屏，和 clear 效果一样
Ctrl+I	水平制表符
Ctrl+K	垂直制表符
Ctrl+J	另起一行
Ctrl+M	回车
Ctrl+Z	暂停前台作业
Ctrl+V	在 vim 中操作 Visual Block
Ctrl+U	删除光标到行首的所有字符

6、感叹号

通常代表逻辑反，例如！=表示不等于