执行脚本的几种方式：
1、sh scriptname 或bash scriptname 会开启新的shell进程执行脚本。
2、使脚本具有执行权限，chmod +x scripname，在当前目录，使用./scriptname执行。
3、source scriptname，在当前shel进程中执行脚本。

脚本首行的#!/bin/bash 表示以/bin/bash来解释执行脚本，在非首行出现#!会被当作普通的注释

```
root@ubuntu:~# cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
```

输出系统时间、当前登录用户、系统当前时间到日志文件，脚本

```
#!/bin/bash
#LOG_FILE=log

echo "current date and time: $(date)' " > $LOG_FILE
echo "currnet login users: $(who)" >> $LOG_FILE
echo "uptime: $(uptime)" >> $LOG_FILE
```

注：在字符串中执行命令 $(cmd)
    >重定向，覆盖模式
    >>重定向，追加模式
输出

```
current date and time: Tue Jan 16 05:10:27 PST 2018'
currnet login users: lg       tty7         2018-01-16 03:27 (:0)
uptime:  05:10:27 up  1:43,  1 user,  load average: 0.14, 0.08, 0.03
```

特殊符号

“”双引号，不影响变量的引用，单引号使变量的引用失效

```
a=375
```

```
echo "$a" # 375
echo '$a' # $a
```

“;” 命令分隔符,可以用来在一行中来写多个命令，前一个命令执行结果不影响后一个命令的执行

```
root@ubuntu:~# ls a ; pwd
ls: cannot access 'a': No such file or directory
/root
```

“;;”双分号用于在case语句中结束一个case

```
variable=abc
case "$variable" in
  abc)  echo "\$variable = abc" ;;
  xyz)  echo "\$variable = xyz" ;;
esac
```

“,”逗号运算符将一系列算术操作连接到一起，只返回最后一个

```
let "t2 = ((a = 9, 15 / 3))" # Set "a = 9" and "t2 = 15 / 3"
echo $t2 $a # 5 9
```

“`”命令替换符，在”``“内可以使用linux命令，命令替换符与$()功能等同（更推荐使用$()形式）

```
x="abc"
y="`cat cleanup.sh`" #变量y的值为cleanup.sh文件的内容，也可以不要外边的双引号
echo $y
```

“**$**”变量替换符，用于替换变量，变量在双引号内时会作为变量使用，在单引号内时作为普通字符

```
```

```
x="abc"
y="$x" # 将x变量的值赋给y
z='$x' # 将$x字符串赋给z
```

“**”指数运算符

```
let a=3**2
echo $a # 9
```

"?"测试符，判断一个字符是否被设置

```
${var?} #检查var变量是否被赋值，未赋值则输出错误，且后面的代码不会被执行
echo $var #这行代码不会执行
```

":" 作为占位符

```
: ${username=`ha`} # “:”作为占位符，username的值不是shell命令也不会报错
${username=`ha`} # 报错./test_command.sh: line 2: ha: command not found
${username=`whoami`} #不报错，因为whoami是shell命令
```

另外“:”与shell的builtin命令true作用相同

```
:
echo $? # 0 “:”执行返回结果为0
```

在无限循环中与true等同

```
while:
do
  ...
done
```

等价于

```
while true
do
  ...
done
```

"$"参数替换符；在正则表达式中作为行结束符

```
$var # =${var} 变量替换
```

${parameter-default} 如果变量未声明，则使用默认值，如果变量声明但未赋值，则使用空值

```
var1=1
var2=2
echo ${var1-$var2} # 1  var1声明且赋值，使用原值
echo ${var3-$var2} # 2  var3未声明，使用默认值var2
var4=  # var4声明但未赋值
echo ${var4-$var2} #  输出空
```

${parameter:-default} 与不加冒号的区别是，只要变量未赋值，就使用默认值（未赋值和赋值为null不同）

```
var1=1
var2=2
echo ${var1:-$var2} # 1  var1声明且赋值，使用原值
echo ${var3:-$var2} # 2  var3未声明，使用默认值var2
var4=  # var4声明但未赋值
echo ${var4:-$var2} # 2  var4未赋值，使用默认值var2
var5=null
echo ${var4:-$var2} # null
```

${parameter+value} 如果变量已声明，使用value值，否则使用空值

```
var1=1
var2=2
echo ${var1+$var2} # 2
echo ${var3+$var2} #空值
var3=
```

```
echo ${var3+$var2} # 7
```

${parameter:+value}如果变量已声明但不为空，则使用value值；如果变量未声明或者已声明但未赋值，使用空值

```
var1=1
var2=2
echo ${var1:+$var2} # 2
echo ${var3:+$var2} # 空
var3=
echo ${$var3:+$var2} # 空
```

“()” 命令组，在“()”中的命令将作为一个子shell来运行，对于脚本中剩下的部分是不可见的，括号中的变量可看作为局部变量

```
(a=123)
echo $a #为空
```

另外“()”也用作初始化数组

```
Array=(elem1 elem2 elem3)
```

“{x,y,z}”大括号，命令对大括号中的以“,”分隔的文件起分别作用

```
cat {file1,file2,file3} > combined_file
# 把 file1,file2,file3 连接在一起,并且重定向到 combined_file 中.
```

```
cp file22.{txt,backup}
# 拷贝"file22.txt" 到"file22.backup"中
```

“{a..z}”扩展的大括号，结果为a到z的连续字符

```
echo {a..z}
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

“{}”代码块，在代码块中声明的变量，在脚本剩下的部分可见

```
{
  b=2
)
echo $b #2
```

“[]”测试命令，等价于test命令

```
if [ 1 -eq 1 ] #注意“[”后和“]”前有空格，  等价于 if test 1 -eq 1
then
 echo "true"
fi
```

“[[]]”扩展测试命令，使用[[...]]条件判断结构, 而不是[...], 能够防止脚本中的许多逻辑错误. 比如,&&, ||, <,和> 操作符能够正常存在于[[ ]]条件判断结构中, 但是如果出现在[ ]结构中的话, 会报错。

```
if [ `ls` && `ls` ]   #./test_command.sh: line 3: [: missing `]'
then
 echo "true"
fi

if [[ `ls` && `ls` ]]  #正确
then
 echo "true"
fi
```

“((...))”双括号构造，与let命令功能类似，在“(())”可以进行算术运算，也可以进行C语言中的类似“a++”的运算

```
a=$((2 + 3))
echo $a #5
((a++))
echo $a #6
```

“**> &> >& >> < <>**”重定向符，stdin（标准输入），stdout（标准输出），stderr（标准错误输出），文件描述符分别为0,1,2 。1是标准输出的默认重定向文件描述符，0是标准输入的默认重定向文件描述符。

command >filename 重定向脚本的输出到文件中，覆盖文件原有内容。

command &>filename 重定向 stdout 和 stderr 到文件中，&在此代表标准输出和标准错误输出

```
command_test () { type "$1" &>/dev/null; }  #将函数的输出结果的标准输出和标准错误输出重定向到/dev/null中

cmd=rmdir            # Legitimate command.
command_test $cmd; echo $?   # 0

cmd=bogus_command    # Illegitimate command.
command_test $cmd; echo $?   # 1
```

i>&j 表示把文件标识符i重新定向到j

command >>filename 以附加方式重定向到文件

command < filename 从文件中接收输入

```
grep search-word <filename
```

j<>filename 打开filename用于读写，并将j作为filename的文件描述符，如果filename不存在则创建，如果j未指定，则将标准输入0作为j的值

```
 exec 3<> File             # Open "File" and assign fd 3 to it.
```

关闭文件描述符

```
n<&-          关闭输入文件标识符n
0<&-或<&-     关闭标准输入stdin
n>&-          关闭输出文件标识符n
1>&-或>&-     关闭标准输出stdout
```

“|”管道命令，将标准输入和标准输出用管道连接起来

```
cat a.txt | grep search-word
```

“&”在后台运行命令

```
sleep 10 & #在后台休眠10秒
```

“~+” 当前工作目录，相当于 $PWD

“~-”前一个工作目录，相当于 $OLDPWD

变量和参数

unset取消变量声明，使用set命令可以查看已声明的变量

```
root@ubuntu:~# set | less
a=123
BASH=/bin/bash
USER=root
...
```

对于一个空值变量（未声明的变量）在进行算术操作的时候，将其值作为0

```
let uninitialized+=5 # Add 5 to it.
echo $uninitialized #5
```

变量赋值时在“=”左右不允许有空格

```
a=375
b = 2 #报错
```

```
hello=$a
echo $hello
```

变量赋值时值不允许有空格，可以使用"\"转义空格，或者使用""

```
b=2 3  #错误
b=2\ 3 #正确
b="2 3" #正确
```

let命令赋值，let是bash中用于计算的工具，在变量的计算时不需要加上$来表示变量

```
let a=5+4 b=9-3
echo $a $b # 9 6
```

read命令赋值，从键盘读取输入

```
read a
echo $a
```

Bash不对变量区分类型，本质上,Bash 变量都是字符串，Bash允许对变量进行比较操作和算术运算，这取决于变量的值是否只包含数字。

```
a=x123
((a++))
echo $a #1 进行算术运算时，Bash将字符串变量的整型值设置为0

b=
((b++)) #1 空变量（值为""或''或null）进行算术运算时，被初始化为0
```

未声明的变量作为除数时不会被初始化为0，会报语法错误

```
((f /= $undef)) #-bash: ((: f /= : syntax error: operand expected (error token is "/= ")
```

特殊变量

本地变量：变量只在代码块或者函数中可见，使用关键字local修饰

```
func()
{
  local local_var=123
  b=234
}
#函数声明也可以定义在一行，但是方法体中的每条语句都要用“;”分隔开，最后一个语句结尾也不例外,单条语句的方法体同样在结尾需要“;”
#func{local loacl_var=123; b=234;}

func #调用函数，在函数调用之前，函数内的变量对外是不可见的
echo $local_var #不可见
echo $b #234
```

环境变量：影响shell运行的变量

在shell中设置了环境变量，需要用exported命令激活。脚本只能对它产生的子进程 export 变量。一个从命令行被调用的脚本 export 的变量，将不能影响调用这个脚本的那个命令行 shell 的环境.

位置变量：$N

$0表示shell文件的名称

$N（N>0）表示程序的第N个参数，大于9的位置变量需要用“{}”，${10}，${11}

特殊变量：只读的系统变量，不能被修改

```
$*  表示程序的所有参数
$#  表示程序的参数个数
$$  表示这个脚本进程的PID
$!  执行上一个后台程序的PID，执行后台程序加上“&”
$?  退出状态变量.$?保存一个命令、一个函数或者脚本本身的退出状态.
```

例：sp_var.sh

```
#!/bin/bash

echo "$* 是程序的所有参数"
```

```
echo "$# 是程序的参数个数"
touch /tmp a.txt
echo "$$ 表示这个程序的PID"
touch /tmp/b.txt & # &表示在后台执行命令
echo "$! 是上一个后台程序的PID"
echo "$? 是上一个命令的返回值"
echo "$$ 表示这个程序的PID"
```

执行结果：./sp_var/sh abc def

```
abc def 是程序的所有参数
2 是程序的参数个数
2370 表示这个程序的PID
2372 是上一个后台程序的PID
0 是上一个命令的返回值
2370 表示这个程序的PID
```

引号

双引号（""）在Bash中用于保留字面意思（$、\、和`除外）
单引号中所有的字符都当作字面意思
使用引号可以抑制echo命令的换行作用

```
echo $(ls -l) #去掉换行
total 28 -rw-r--r-- 1 root root 0 Jan 27 19:18 a.txt -rwxr-xr-x 1 root root 157 Jan 16 04:11 cleanup.sh -rw-r--r-- 1 root root 239 Jan 17 04:19 log

echo "$(ls -l)"
total 28
-rw-r--r-- 1 root root   0 Jan 27 19:18 a.txt
-rwxr-xr-x 1 root root 157 Jan 16 04:11 cleanup.sh
-rw-r--r-- 1 root root 239 Jan 17 04:19 log
```

退出和退出状态码

exit命令用于结束脚本，每个命令都会返回一个exit状态，成功返回0，失败返回非0值，在脚本或脚本函数中执行的最后的命令会决定退出状态，退出状态码范围为0-255，当一个脚本以不带参数 exit 来结束时（或者省略exit命令时）,脚本的退出状态就由脚本中最后执行命令来决定。

“!”逻辑非命令会影响退出状态码

```
true
echo $? #0, 正确执行
! true #"!"与命令之间有空格，没有空格表示执行前一个命令
echo $? #1
```

“!”不改变管道命令的执行，只影响退出状态码

```
! ls | bogus_command   # bash: bogus_command: command not found
echo $?               # 0
```

测试命令

if/then结构可以测试命令的返回值是否为0，如果是则执行更多命令

```
word=Linux
letter_sequence=inu
if echo "$word" | grep -q "$letter_sequence"
# The "-q" option to grep suppresses output. 抑制输出
then
  echo "$letter_sequence found in $word"
else
  echo "$letter_sequence not found in $word"
fi
```

if-then测试结构

```
if [ condition-true ] #"[" "]"前后有空格
then
   command 1
   command 2
   ...
else
   command 3
   command 4
   ...
fi
```

elif测试结构

```
if [ condition1 ]
then
   command1
   command2
   command3
elif [ condition2 ]
then
   command4
   command5
else
   default-command
fi
```

0是true

```
if [ 0 ]      # zero
then
  echo "0 is true."
else          # Or else ...
  echo "0 is false."
fi            # 0 is true.
```

1是true

```
if [ 1 ]      # one
then
  echo "1 is true."
else
  echo "1 is false."
fi            # 1 is true.
```

-1是true

```
if [ -1 ]     # minus one
then
```

```
  echo "-1 is true."
else
  echo "-1 is false."
fi            # -1 is true.
```

null是false

```
if [ ]        # NULL (empty condition)
then
  echo "NULL is true."
else
  echo "NULL is false."
fi            # NULL is false.
```

未声明的变量是false

```
if [ $xyz ]   # Tests if $xyz is null, but it's only an uninitialized
variable.
then
  echo "Uninitialized variable is true."
else
  echo "Uninitialized variable is false."
fi            # Uninitialized variable is false.
```

文件测试操作符

```
-e 文件是否存在 #用法 if [ -e a.txt]
-f 文件是否是普通文件（file, 非目录或设备文件）
-s 文件长度不为0
-d 文件是否是目录
-b 文件是否是块设备（硬盘驱动器、CD-ROM驱动器和闪存驱动器）
-c 文件是否是字符设备（键盘、调制解调器、声卡）
-p 文件是否是管道
-h 文件是否是符号链接
-S 文件是否是Socket
-r 文件是否有读权限
-w 文件是否有写权限
-x 文件是否有执行权限
-O 当前用户是否是文件的拥有者
```

```
-G 当前用户的group-id是否和文件的group-id相等
-N 文件从上一次阅读到现在是否被修改过
f1 -nt f2 文件1是否比文件2新（newer）
f1 -ot f2 文件1是否比文件2旧(older)
f1 -ef f2 文件1和文件2是否都硬链接到同一个文件
! 反转测试结果，如果!后面没有比较命令，则返回true
if [ ! ]
then echo true
fi #返回true
```

整型比较操作符

```
-eq 是否相等(equal) #用法 if [ $a -eq $b ] 变量a和变量b的值必须是全数字
-ne 是否不相等(not equal)
-gt 是否大于(greater then)
-ge 是否大于或等于(greater or equal)
-lt 是否小于(less then)
-le 是否小于或等于(less or equal)
< 是否小于，需要加双括号，使用C语言写法，#用法 if (($a < $b))
<= 是否小于等于，需要加双括号
> 是否大于，需要加双括号
>= 是否大于或等于，需要加双括号
```

字符串比较

```
= 是否相等 #用法 if [ $a = $b ] 注意等于号前后的空格
== 是否相等，与“=”相同
!= 是否不相等
< 是否小于，比较ASCII字母顺序 #用法 if [[$a < $b]] 或 if [$a \< $b]  “[]”中需要转义
> 是否大于
-z 字符串是否为空 (zero-length长度为0) #用法 if [ -z $str ]
-n 字符串是否不为空 (not null)
```

使用 -n判断字符串是否为空是，需要将变量用双引号括起来
未声明的变量和声明了但未赋值的变量都为null
变量str1未声明，判断非空时，不加双引号判断为非空，加双引号判断为空

```
#不使用双引号时
```

```
if [ -n $str1 ]
then
  echo "not null"
else
  echo "null"
fi # 打印not null，正确的结果应该是null
```

```
#不使用双引号时
if [ -n $str1 ]
then
  echo "not null"
else
  echo "null"
fi # 打印 null
```

如果不使用双引号，可以使用如下方式

```
if [ $str1 ] #不使用-n，但是与正常逻辑相悖
then
  echo "not null"
else
  echo "null"
fi # 打印null
```

使用此方式如果str1是一个表达式“a = b” 就会出现错误的结果

```
str2="a = b"
if [ $str2 ] #将变量str2的值代入，测试条件变为 if [ a = b ]，返回false
then
  echo "not null"
else
  echo "null"
fi #打印null
```

复合比较

```
-a 逻辑与 #用法 if[ exp1 -a exp2 ] 如果 exp1 和 exp2 都为 true 的话,这个表达式将返回 true
-o 逻辑或 #用法 if[ exp1 -o exp2 ] 如果 exp1 和 exp2 中有一个为 true 的话,那么这
```

```
个表达式就返回 true
```

与C语言中的&& 和 ||功能相同，&&在shell中的用法如下

```
if [[ exp1 && exp2 ]]
#或者
if exp1 && exp2
```

if-then嵌套结构

```
if [ condition1 ]
then
  if [ condition2 ]
  then
    do-something
  fi
fi
```

操作符

位操作符

```
<< #左移一位（乘以2）
<<= #左移n位（=号后边是几位就左移几位）let "var <<= 2" 左移2位，乘以4
>> #右移一位
>>= #右移n位
& #按位与 （都为1则为1，否则为0）
&= #按位与赋值，a&=b 等价于 a=a&b
| #按位或 （有一个为1就为1）
|= #按位或赋值
~ #按位非 （0变1，1变0）
^ #按位异或（相同为0，不同为1）
```

逻辑操作符

```
! #非
&& #且，两个表达式都为真则为真
|| #或，两个表达式有一个为真则为真
```

数字常量

shell 脚本默认都是将数字作为 10 进制数处理，除非这个数字使用某种特殊的标记法或前缀开头。以 0 开头是 8 进制，以 0x 开头是 16 进制数。

使用 BASE#NUMBER 这种形式可以表示其它进制。

```
let "b32 = 32#77" #32进制
echo "base-32 number = $b32" # 231
```

超出进制范围的变量会给出错误消息

```
let "bad_oct = 081" #-bash: let: bad_oct = 081: value too great for base (error token is "081")
```

变量扩展部分

内部变量（Builtin变量）

```
$BASH  #Bash的二进制执行文件的位置 /bin/bash
$BASH_VERSION #系统的Bash版本号 4.3.48(1)-release
$HOME #当前用户的home路径
$GROUPS #当前用户所属的组
$HOSTNAMR #主机名
$HOSTTYPE #主机类型 x86_64 与 $MACHTYPE类似 x86_64-pc-linux-gnu
$OSTYPE #系统类型 linux-gnu
$PPID #父进程PID
$SECONDS #此脚本已经运行的时间（单位为秒）
$UID #当前用户ID，在/etc/passwd中记录
$REPLY #使用read命令读入数据，但是未使用变量保存时，REPLY会保存最后一条read的数据，如果有变量保存，则REPLY不保存数据
```

```
read #不用变量保存，输入23
echo $REPLY #输出23

read a #使用变量保存，输入45
echo $REPLY #输出23，最后一次read但是未使用变量保存的值
```

声明变量类型

declare 或 typeset命令可以用于声明变量的类型

```
declare -r var1=1 #声明var1为只读变量
var1=2 #-bash: var1: readonly variable  修改只读变量的值会报错
```

```
declare -i var2=1 #声明var为整型变量
var2=str #将整型变量的值赋值为字符串
echo $var2 #0  整型变量的值被重置为0
```

将变量声明为整型后，可以直接进行某些算术运算，而不需要使用let命令或expr命令

```
n=6/3  #未声明变量n为整型变量，被当作字符串
echo "n = $n"       # n = 6/3

let n=6/3    #或者 n= $(expr 6 / 3)
echo "n = $n"       # n = 2

declare -i n  #声明变量n为整型变量
n=6/3
echo "n = $n"       # n = 2
```

```
declare -a array #声明变量array为数组
```

变量的间接引用

```
a=letter_of_alphabet
letter_of_alphabet=z
echo $a # letter_of_alphabet
#如果想要得到变量letter_of_alphabet的值
eval echo "\$$a" #eval命令将会首先扫描命令行进行所有的替换，然后再执行命令
```

获取字符串的长度

```
str=abc
echo ${#str} # 3
echo `expr length $str` #3
echo `expr "$str" : '.*'` #3
```

循环分支

for循环

适用于知道循环次数的情况

```
for arg in [list]
do
 command(s)...
done
```

举例

```
NUMBERS="9 7 3 8 37.53"

for number in `echo $NUMBERS`  # for number in 9 7 3 8 37.53
do
  echo -n "$number " #不换行打印
done
```

大括号形式

```
for a in {1..10}
```

seq命令形式

```
for a in `seq 10` #seq 10 命令产生1-10的连续整数
```

C形式for循环

```
LIMIT=10

for ((a=1; a <= LIMIT ; a++))  # Double parentheses, and naked "LIMIT"
do
```

```
  echo -n "$a "
done
```

可以使用大括号替代do 和 done关键字

```
for((n=1; n<=10; n++))
# No do!
{
  echo -n "* $n *"
}
# No done!

# Outputs:
# * 1 ** 2 ** 3 ** 4 ** 5 ** 6 ** 7 ** 8 ** 9 ** 10 *
```

while循环

适用于不知道循环次数的情况

```
while [ condition ]
do
 command(s)...
done
```

举例

```
var0=0
LIMIT=10

while [ "$var0" -lt "$LIMIT" ]
#      ^                    ^
# Spaces, because these are "test-brackets" . . .
do
  echo -n "$var0 "        # -n suppresses newline.
  #             ^          Space, to separate printed out numbers.

  var0=`expr $var0 + 1`   # var0=$(($var0+1))
                          # ((var0 ++))
                          # let "var0 += 1"
done
```

until循环

循环至少会执行一次，当条件成立时退出循环，与C语言中的do...until结构类似

```
until [ condition-is-true ]
do
 command(s)...
done
```

举例

```
END_CONDITION=end

until [ "$var1" = "$END_CONDITION" ] #当输入end时退出
do
  echo "Input variable #1 "
  echo "($END_CONDITION to exit)"
  read var1
  echo
done
```