Chapter3: bash shell 中的特殊字符详解

[sharp] # 井号

- 井号常用作注释符号

1.注释示例

```
# This line is a comment.
```

2.某命令后注释，**#**号前需要添加一个空格

```
echo "A comment will follow." # Comment here.
#                             ^ Note whitespace before #
```

3.注释前亦可跟空白字符

```
   # A tab precedes this comment.
```

4.注释符号还可以被嵌入到带管道的命令当中

```
initial=( `cat "$startfile" | sed -e '/#/d' | tr -d '\n' |\
          sed -e 's/\./\. /g' -e 's/_/_ /g'` )
# Delete lines containing '#' comment character.
# 该命令用于删除包含#号的行
```

5.当然，在**echo**命令中被引用或者被转义的**#**号不会成为注释,**#**号也会出现在特定的参数替换结构中及一些数值常量表达式中

```
echo "The # here does not begin a comment."
echo 'The # here does not begin a comment.'
echo The \# here does not begin a comment.
echo The # here begins a comment.
echo ${PATH#*:}       # 参数替换，不是注释
echo $(( 2#101011 ))  # 数制转换，不是注释

[root@centos7 /data/test]$echo ${PATH#*:}
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
```

```
[root@centos7 /data/test]$echo ${PATH}
/usr/lib64/qt-3.3/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
```

6.标准的单双引用符号和转义符号**("'/)**都能转义**#**号

7.某些特定的模式匹配操作也使用**#**号

[semicolon] ; 分号

- 分号一般用作命令分隔符，允许多个命令处于同一行

```
echo hello; echo there
```

```
if [ -x "$filename" ]; then    #  Note the space after the semicolon.
#+                    ^^
  echo "File $filename exists."; cp $filename $filename.bak
else   #                       ^^
  echo "File $filename not found."; touch $filename
fi; echo "File test complete."
```

[double semicolon] ;; 双分号

- 双分号用作case语句中的语句结束符
- bash4.0+的版本使用;;&或者;&作为结束符

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

[period] . 英文句号

- 通常，英文句号.为bash builtin命令，等同于source

1.作为文件名的一部分，当一个文件以**.**开头，则为隐藏文件，**ls**不会显示，使用**ls -a**

```
bash$ touch .hidden-file
bash$ ls -l
total 10
 -rw-r--r--    1 bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo      4602 May 25 13:58 data1.addressbook.bak
 -rw-r--r--    1 bozo       877 Dec 17  2000 employment.addressbook
bash$ ls -al
total 14
 drwxrwxr-x    2 bozo  bozo      1024 Aug 29 20:54 ./
 drwx------   52 bozo  bozo      3072 Aug 29 20:51 ../
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.addressbook.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.addressbook
 -rw-rw-r--    1 bozo  bozo         0 Aug 29 20:54 .hidden-file
```

2.句号在目录中，一个句号.表示当前目录，两个句号..表示当前目录的父目录

```
bash$ pwd
/home/bozo/projects
bash$ cd .
bash$ pwd
/home/bozo/projects
bash$ cd ..
bash$ pwd
/home/bozo/
```

3.句号在移动文件的命令中表示目标文件夹，此时的目标文件夹常常是当前目录

```
bash$ cp /home/bozo/current_work/junk/* .
# 拷贝junk/文件夹下的所有文件到当前目录
```

4.字符匹配时，句号作为正则表达式的一部分表示匹配任意一个字符

[double quote] ' " 单双引号

- **"STRING" preserves (from interpretation) most of the specia characters within STRING**
- **'STRING' preserves all special characters within STRING. This is a stronger form of quoting than "STRING"**
- 部分(弱)引用：**"STRING"** 这种写法表示 解释器会认为**STRING**中的小部分特殊字符有特殊意义
- 全(强)引用：**'STRING'** 这种写法表示 解释器会认为**STRING**中的所有特殊字符都无特殊意义

```
[root@centos7 /data]$var=jjjj
[root@centos7 /data]$cat file
```

```
test
ddd
[root@centos7 /data]$sed "s/test/${var}/" file #双引号表示部分特殊字符具备特殊意义,此处${var}表示jjjj
jjjj
ddd
[root@centos7 /data]$sed 's/test/${var}/' file #单引号表示所有特殊字符均无特殊意义,此处${var}表示包含6个字符的字符串
${var}
ddd
[root@centos7 /data]$
```

[comma operator] , 逗号操作符

- 逗号操作符将多个数学运算表达式链接在一起，所有表达式都会被计算，但是只有最后一个表达式的结果被返回

```
[root@centos7 ~]$let "t2 = ((a = 9, 15 / 3))"
[root@centos7 ~]$echo $t2   # 此处返回值为15/3=5
5
[root@centos7 ~]$echo $a
9
```

- 逗号操作符亦可和花括号{}配合以用来连接字符串

```
for file in /{,usr/}bin/*calc
#             ^    Find all executable files ending in "calc"
#+                 in /bin and /usr/bin directories.
do
        if [ -x "$file" ]
        then
          echo $file
        fi
done

# /bin/ipcalc
# /usr/bin/kcalc
# /usr/bin/oidcalc
# /usr/bin/oocalc
```

[backslash] \ 反斜杠

- 用于单个字符的引用机制

```
\X 该写法将转义字符X，等价于'X'，反斜杠也会用于转义'和"。
```

[forward slash] / 斜杠

- 斜杠一般用作文件名路径分隔符

```
[root@centos7 ~]$cat /home/bozo/projects/Makefile
```

- 斜杠也是除法运算符

[backquotes] ` 反引号

- 反引号用作命令替换，`command` 这种结构使得command的执行结果可以赋给新的变量

```
[root@centos7 ~]$num=`seq 1 10`
[root@centos7 ~]$echo $num
1 2 3 4 5 6 7 8 9 10
```

[colon] : 冒号

- 冒号在shell中表示"NOP" (no op, a do-nothing operation)，一个不做任何操作的命令；":"冒号命令属于bash builtin类型，其命令退出状态为True（0）

```
[root@centos7 ~]$:
[root@centos7 ~]$echo $?
0
```

1.冒号用于实现无限循环

```
while :
do
   operation-1
   operation-2
   ...
   operation-n
done
# 等同于:
#    while true
```

```
#    do
#      ...
#    done
```

2.在**if/then**语句中作为占位符

```
if condition
then :   # 什么也不做，分支继续
else     # Or else ...
   take-some-action
fi
```

3.在需要进行二进制操作的地方提供一个占位，

```
[root@centos7 ~] : ${username=`whoami`}
# ${username=`whoami`}   不以: 开头则会给出错误提示
#                        unless "username" is a command or builtin...
[root@centos7 ~] : ${1?"Usage: $0 ARGUMENT"}
```

4.用参数替换来确定某个变量是否已经存在

```
[root@centos7 ~]$: ${HOSTNAME?} ${USER?} ${MAIL?}
[root@centos7 ~]$echo $?
0
# 如果上面某个或多个必要的环境变量未设置，则会打印一条错误消息，此时我的电脑三个环境变量都已经存在，所以没有错误消息
```

5.和重定向符号**>**配套使用，清除某个文件的内容，不改变其原有的权限属性**-$: > file**

```
[root@centos7 /data/test]$cat hello
HELLO
[root@centos7 /data/test]$: > hello  # 此处清空hello文件，如果hello不存在，则创建之
[root@centos7 /data/test]$cat hello
[root@centos7 /data/test]$
```

注意：上面用法等同于**“cat /dev/null > hello”**

然而，使用上面的方法不会产生子进程，因为**“:”**是一个**builtin**类型

如果和追加重定向符**>>**配合使用，则不对文件产生任何影响；无对应文件则创建之

6.分号还可以用作注释，但是**bash**会在以分号开头的注释中检查错误；而以**#**号开头的注释是关闭错误检查的

```
[root@centos7 /data/test]$cat test.sh
#!/bin/bash
#
: This is a comment that generates an error, ( if [ $x -eq 3] )  #以分号开头的注释含有错误代码
[root@centos7 /data/test]$bash -n test.sh
test.sh: line 13: syntax error near unexpected token `('
test.sh: line 13: `: This is a comment that generates an error, ( if [ $x -eq 3] )' #bash检查出分号开头的注释存在错误
[root@centos7 /data/test]$
```

7.分号还可以作为域分隔符，如文件**:/etc/passwd**和环境变量**PATH**

```
[root@centos7 ~]$echo $PATH
/usr/lib64/qt-3.3/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
```

8.分号也可以作为函数名，但是不推荐这种做法，会使得代码不易读

```
:()             # 该函数的函数名为分号
{
  echo "The name of this function is "$FUNCNAME" "
  # Why use a colon as a function name?
  # It's a way of obfuscating your code.
}
```

注意：在比较新的**bash**版本中已经不允许这样做，但是可以使用_下划线作为函数名 **9.**冒号可以用作空函数中的占位符

```
not_empty ()
{
  :
} # Contains a : (null command), and so is not empty.
```

注：该函数不做任何动作，但是不是空函数

[bang]！感叹号

- 感叹号一般有取反和在bash命令行调用命令历史机制的作用

1.感叹号用来对一个**test**测试条件或者命令退出状态取反，感叹号属于**bash**关键字

2.在不同的上下文中，**!**的出现也意味着间接变量引用

3.在**bash**命令环境中，**!**调用**Bash**历史机制

```
 1092  cls
 1093  ll
 1094  history
[root@centos7 ~]$!l    # 该写法表示bash执行history命令记录中以l开头的最近一次执行过的
命令
ll
total 32
-rw-r--r--. 1 root root 4352 Sep 17 11:04 1
drwxr-xr-x. 3 root root   17 Aug 24 13:13 Desktop
drwxr-xr-x. 2 root root    6 Aug 24 13:06 Documents
drwxr-xr-x. 2 root root    6 Aug 24 13:06 Downloads
drwxr-xr-x. 2 root root    6 Aug 24 13:06 Music
drwxr-xr-x. 2 root root    6 Aug 24 13:06 Pictures
drwxr-xr-x. 2 root root    6 Aug 24 13:06 Public
drwxr-xr-x. 2 root root    6 Aug 24 13:06 Templates
drwxr-xr-x. 2 root root    6 Aug 24 13:06 Videos
-rw-------. 1 root root 2053 Aug 24 13:03 anaconda-ks.cfg
-rw-r--r--. 1 root root 2101 Aug 24 13:05 initial-setup-ks.cfg
-rwxr-xr-x. 1 root root 3121 Sep 12 00:04 reset.sh.bak0
-rw-r--r--. 1 root root    4 Sep  4 11:25 somaxconn~
-rw-r--r--. 1 root root    4 Sep  4 11:29 somaxcony~
-rw-r--r--. 1 root root    4 Sep  4 11:29 somaxconz~
[root@centos7 ~]$
```

[asterisk] * 星号通配符

1.星号在文件名通配操作中扮演通配符的角色，表示匹配某文件夹下的所有文件名；与其他具体字符结合表示匹配**0**个或者多个任意字符

```
[root@centos7 /data]$ echo *
c_program py_scripts rpmPacksges scripts test test_scripts ttt.sh
[root@centos7 ~]$ echo /data/*
/data/c_program /data/py_scripts /data/rpmPacksges /data/scripts /data/test
/data/test_scripts /data/ttt.sh
```

2.星号在正则表达式中表示：匹配其前面的字符任意次，包括**0**次
3.在算术运算操作中，星号表示乘法操作符
4.** 两个星号一起是运算符中的乘方操作符号，比如**2**8=256**
5.** 两个星号一起在**bash4.0**版本以上内核中表示**$\color{red}{扩展的文件通配操作}$，会递归通配文件

```
#!/bin/bash4
# filelist.bash4
shopt -s globstar  # 必须使能 globstar, 否则 ** 没用.
                   # globstar 是bash4.0中新增的shell选项
echo "Using *"; echo
for filename in *
```

```
do
  echo "$filename"
done   # 列出当前目录的文件 ($PWD).
echo; echo "--------------"; echo
echo "Using **"
for filename in **
do
  echo "$filename"
done   # 递归列出完整的文件树.
exit

# 列出当前目录输出如下
Using *
allmyfiles
filelist.bash4
--------------
# 递归列出完整的文件树输出如下
Using **
allmyfiles
allmyfiles/file.index.txt
allmyfiles/my_music
allmyfiles/my_music/me-singing-60s-folksongs.ogg
allmyfiles/my_music/me-singing-opera.ogg
allmyfiles/my_music/piano-lesson.1.ogg
allmyfiles/my_pictures
allmyfiles/my_pictures/at-beach-with-Jade.png
allmyfiles/my_pictures/picnic-with-Melissa.png
filelist.bash4
```

[test operator] ? 测试操作符

1.在特定的表达式中，问号表示对一个条件的测试
2.在双括号结构中，问号表现为**C**风格三元操作符的组成部分

```
# condition?result-if-true:result-if-false
# 条件?条件为真的值:条件为假的值
(( var0 = var1<98?9:21 ))
#                ^ ^
# if [ "$var1" -lt 98 ]
# then
#   var0=9
# else
#   var0=21
# fi
```

3.在参数替换表达式中，问号表示某变量是否已经存在

```
${paramseter?err_msg}, ${parameter:?err_msg}
# 如果parameter已经存在，就使用其；否则打印err_msg退出脚本，并且退出状态为1
# 上面两种写法几乎同等，后面写法中的冒号表示只有当parameter已经被声明且为空时(null)就使用
```

4.作为通配符，在文件名通配中表示匹配任何当个字符；在扩展正则表达式中表示匹配其前面的某单个字符。

[$] 变量替换符(获取一个变量所存储的内容)

```
[root@centos7 /data/globbing]$var1=123
[root@centos7 /data/globbing]$var2=hello
[root@centos7 /data/globbing]$echo $var1; echo $var2
123
hello
```

[$] end-of-line 在正则达式中表匹配文本的行结束位置，常用于锚定；在linux系统中文本文件的行结束符也是$

```
[root@centos7 /data/globbing]$ls |grep '.*[0-9]$'  # 表示匹配以数字结尾的文件名
10file.1
10file.2
1SdsflDSLFsdf677671
1SdsflDSLFsdf677672
{a-z}dsf3adsf1
DSdsflDSLFsdf677671
[root@centos7 /data/globbing]$ls |grep '.*[a-zA-Z]$'   # 表示匹配以字母结尾的文件名
10file.txt
1SdsflDSLFsdf67767A
1SdsflDSLFsdf67767B
a123321a
A123321A
a123321b
```

```
[root@centos7 /data/globbing]$echo hello > 10file.1
[root@centos7 /data/globbing]$cat -A 10file.1
hello$                                          # cat -A 查看文本的不可打印字符，包括tab键和行结束符$等。
```

[${}] 参数替换符,获取花括号中的变量所存储的内容；几乎和$等同，在某些情况下使用${}(例如：使用字符串连接不同的变量所存储的内容)

```
[root@centos7 /data/globbing]$echo $USER       # 环境变量USER，保存有当前用户的用户名，使用$USER获取
root
[root@centos7 /data/globbing]$echo ${USER}     # 此处功能同$
root
[root@centos7 /data/globbing]$your_id=${USER}-on-${HOSTNAME}     # 此处使用'-on-' 将USER和HOSTNAME存储的内容连接起来；获取它们的内容必须用${}
[root@centos7 /data/globbing]$echo "$your_id"
root-on-centos7.magedu.steve                    # 连接后的结果
```

[$' ... ']

- 该结构将展开单个或多个被转义的8进制或者16进制的值并转换为ASCII码或者Unicode字符：

```
[steve@centos7 ~]$echo $'\x21'
!
[steve@centos7 ~]$echo $'\x22'                  # 十六进制x22代表ASCII码中的双引号 "
"
[steve@centos7 ~]$echo $'\037'

```

[$*, $@] 位置参数，存储所有的位置参数，有区别

- $* 将所有位置参数视为单个字符串
- $@ 每个位置参数存储为单独引用的字符串，分开对待每个位置参数

[$?] 该环境变量存储退出状态；可以是命令、函数或者脚本的退出状态

- 脚本的退出状态为脚本中最后一条命令的退出状态
- 函数退出状态也为最后一条命令的退出状态
- 一般成功执行退出状态为0；命令执行失败退出状态为1-255之间的整数.

```
[root@centos7 /data/test]$cat exit_status.sh
#!/bin/bash
echo hello
```

```
echo $?    # 打印hello成功，放回的退出状态值为0.
lskdf      # Unrecognized command.
echo $?    # 无该命令命令，执行失败，退出状态非0.
echo
exit 113   # 脚本结束后使用echo $? 查看，脚本退出状态为113.
[root@centos7 /data/test]$bash exit_status.sh
hello
0
exit_status.sh: line 4: lskdf: command not found
127
[root@centos7 /data/test]$echo $?
113
```

[$$] 为PID变量，存储其出现在的脚本所属的进程的进程号

```
[root@centos7 ~]$pstree -p |grep sshd.*bash
           |-sshd(1164)---sshd(1842)---bash(1848)---bash(6007)-+-grep(6043)
[root@centos7 ~]$echo $$                          # 当前所在bash进程为6007
6007
[root@centos7 ~]$exit                             # 退出6007号bash进程
exit
[root@centos7 ~]$echo $$                          # 此时$$记录1848
1848
```

[()] 圆括号

1.圆括号可以用来执行其包括的一组命令，各个命令使用分号；隔开

```
[root@centos7 ~]$(a=hello; echo $a)
hello
```

- 注意；结构(command1;command2)中，shell会生成一个子shell进程来运行括号内的多个命令。括号内的变量(子shell中)不被括号外的命令读取，父进程(父shell)无法读取子进程的变量。

```
a=123
( a=321; )
echo "a = $a"   # a = 123
# "a" 可视为本地变量.
```

2.圆括号用于初始化数组

```
Array=(element1 element2 element3)
```

[{xxx,yyy,zzz,...}] 花括号展开

1.有如下用法

```
echo \"{These,words,are,quoted}\"   # " 添加前缀和后缀(prefix and suffix)
# "These" "words" "are" "quoted"
cat {file1,file2,file3} > combined_file
# 链接三个文件 file1, file2, and file3 成一个文件combined_file.
cp file22.{txt,backup}
# Copies "file22.txt" to "file22.backup"
```

2.使用花括号展开时如果包含空格需要转义

```
[root@centos7 /data/test]$echo {file1,file2}\ :{\ A," B",' C'}
file1 : A file1 : B file1 : C file2 : A file2 : B file2 : C
```

[{a..z}] 扩展的花括号展开

- 扩展的花括号展开是bash3.0中新介绍的特性

```
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
# Echoes characters between a and z.
echo {0..3} # 0 1 2 3
# Echoes characters between 0 and 3.
base64_charset=( {A..Z} {a..z} {0..9} + / = )
# Initializing an array, using extended brace expansion.
```

[{}] 花括号代码块

1.花括号中的代码块所包含的变量可被后续脚本代码识别

```
# 例子1
bash$ { local a;
              a=123; }
bash: local: 使用local关键字定义的变量为本地变量，只能在某个函数中使用

# 例子2
a=123
```

```
{ a=321; }
echo "a = $a"   # a = 321   (花括号中的变量2被打印)
# Thanks, S.C.
```

2.花括号所包含的代码块一般有**I/O**重定向

```
#!bin/bash

File=/etc/fstab
{
read line1
read line2
} < $File
echo "First line in $File is:"
echo "$line1"
echo
echo "Second line in $File is:"
echo "$line2"
exit
```

3.花括号内的代码块执行结果保存到某个文件中

```
#!/bin/bash
# rpm-check.sh
#  Queries an rpm file for description, listing,
#+ and whether it can be installed.
#  Saves output to a file.
#
#  This script illustrates using a code block.
SUCCESS=0
E_NOARGS=65
if [ -z "$1" ]
then
echo "Usage: `basename $0` rpm-file"
  exit $E_NOARGS
fi
{ # Begin code block.
  echo
  echo "Archive Description:"
  rpm -qpi $1       # Query description.
  echo
  echo "Archive Listing:"
  rpm -qpl $1       # Query listing.
  echo
  rpm -i --test $1  # Query whether rpm file can be installed.
  if [ "$?" -eq $SUCCESS ]
  then
    echo "$1 can be installed."
  else
```

```
    echo "$1 cannot be installed."
  fi
  echo              # End code block.
} > "$1.test"       # Redirects output of everything in block to file.
echo "Results of rpm test in file $1.test"
# See rpm man page for explanation of options.
exit 0
```

- 不像()圆括号中的命令组合会产生新的shell子进程，花括号{}中的代码通常不产生子shell。

4.不标准的for循环也可以遍历某个花括号中的代码块

```
for((n=1; n<=10; n++))    # 注意：圆括号后无内容，直接跟花括号代码块
# No do!
{
  echo -n "* $n *"
}
# No done!
# Outputs:
# * 1 ** 2 ** 3 ** 4 ** 5 ** 6 ** 7 ** 8 ** 9 ** 10 *
# And, echo $? returns 0, so Bash does not register an error.
```

5.花括号亦可用作文本占位符，一般用于xargs -i(替换字符串选项)后面；花括号是输出的文本的占位符

```
ls . | xargs -i -t cp ./{} $1
#            ^^          ^^
# From "ex42.sh" (copydir.sh) example.
```

[[]，[[]]] 测试结构，测试表达式位于[]之间

- 需要注意的是'[',左中括号是shell的built-in测试类型，而没有链接到外部测试命令/usr/bin/test
- [[]],双重的中括号测试结构比[]更加灵活和稳定，[[]]是shell的关键字，keyword; 也叫做扩展的测试命令，从ksh88版本中借鉴而来。
- 注意注意注意：在双中括号结构中，文件名展开和word splitting都失效；但是参数替换和命令替换有效。

```
file=/etc/passwd
if [[ -e $file ]]
then
  echo "Password file exists."
fi
```

- 在数组的使用中，[]用来标识数组中的某个元素

```
[root@centos8 ~]$Array[1]=slot_1
[root@centos8 ~]$echo ${Array[1]}
slot_1
```

- []还乐意用来表示某范围内的字符，作为正则表达式的一部分，中括号描述了某个范围内的可被匹配到的字符。

[$[...]] 整数表达式计算

- $[...]该结构计算中括号内的整数表达式

```
a=3
b=7
echo $[$a+$b]   # 10
echo $[$a*$b]   # 21
```

-

[(( ))] 整数表达式计算

- 展开并计算(())内的整数表达式

['>' '&>' '>&' '>>' '<>'] 各种重定向符

1. '>'

- scriptname >filename 重定向左边脚本的执行结果到右边文件中；如果文件已经存在则覆盖。

2. '&>'

- command &>filename 将command的标准输出和标准错误重定向到文件filename中。
- 在CLI中测试某个命令是否存在时使用该重定向符时很有帮助的。如：

```
bash$ type bogus_command &>/dev/null
bash$ echo $?
1
```

- 或是在脚本中

```
command_test () { type "$1" &>/dev/null; }
#                                     ^
cmd=rmdir            # Legitimate command.合法命令
```

```
command_test $cmd; echo $?   # 0
cmd=bogus_command    # Illegitimate command.不存在的命令
command_test $cmd; echo $?   # 1
```

3. '>&'

- command >&2 该结构将重定向command的标准输出到标准错误。

4. '>>'

- scriptname >>filename 将scriptname的输出结果追加到filename文件中。如果filename文件事先不存在则创建。

5. '<>'

[i]<>filename 该结构以可写可读的方式打开filename文件，并且添加文件描述符i到该文件。如果filename不存在，则创建。

[<<]

- 用于就地文本的重定向符号。

[<<<]

- 用于就地字符串的重定向符号。

[<,>]

- 用来比较ASCII字符

```
veg1=carrots
veg2=tomatoes
if [[ "$veg1" < "$veg2" ]]
then
  echo "Although $veg1 precede $veg2 in the dictionary,"
  echo -n "this does not necessarily imply anything "
  echo "about my culinary preferences."
else
  echo "What kind of dictionary are you using, anyhow?"
fi
```

[<, >]

- 在正则表达式中用于锚定单词的头部和尾部。

```
[root@centos8 ~]grep '\<root\>' /etc/passwd
root:x:0:0:steve,banzhuang,18800001111,1123443,:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
```

[|] 管道符

- linux中管道用于连接多个命令，将前一个命令的输出（标准输出）传给后一个命令的输入（标准输入）或者shell。

```
echo ls -l | sh
#  Passes the output of "echo ls -l" to the shell,
#+ with the same result as a simple "ls -l".
cat *.lst | sort | uniq
# Merges and sorts all ".lst" files, then deletes duplicate lines.
```

- 某个命令的输出也可以使用管道传给某个脚本。

```
#!/bin/bash
# uppercase.sh : Changes input to uppercase.
tr 'a-z' 'A-Z'
#  Letter ranges must be quoted
#+ to prevent filename generation from single-letter filenames.
exit 0
####################
bash$ ls -l | ./uppercase.sh
-RW-RW-R--    1 BOZO  BOZO       109 APR  7 19:49 1.TXT
-RW-RW-R--    1 BOZO  BOZO       109 APR 14 16:48 2.TXT
-RW-R--R--    1 BOZO  BOZO       725 APR 20 20:56 DATA-FILE
```

- 在管道两侧签个进程的标准输出必须作为下一个进程的标准输入被读取。否则，数据流将会被阻塞，管道将不会按照预想的工作。

```
cat file1 file2 | ls -l | sort
# The output from "cat file1 file2" disappears.命令cat file1 file2的输出会消失
```

- 管道是作为子进程运行的，因此无法更改脚本中的变量。

```
variable="initial_value"
echo "new_value" | read variable
echo "variable = $variable"     # variable = initial_value
```

- 如果在管道中的某个命令终止了，这将会提前结束该管道进程。叫做'断开的管道'，这种情况下会发送一个SIGPIPE信号。

[>|] 强制重定向符

- 强制重定向（即是noclobber选项已经被设置），使用该符号强制性的覆盖某个文件。
- noclobber是bash的选项之一，使用-C选项指定，意思是不让重定向符'>'覆盖文件，但使用'>|'强制覆盖。

[||] OR或逻辑操作符

- OR逻辑操作符。在test测试结构中，如果||两边的测试结构只有一个为真，则整体返回0（成功）。

[&] AND符号

- &使用该符号在后台运行某个作业。一个命令后面跟&符号将会在后台运行该命令。

```
[root@centos8 ~](sleep 5 ;echo -e "\ndone")&
[1] 15682
[root@centos8 ~]
done
```

- 在脚本中，命令或者循环也可以运行在后台。

```
#!/bin/bash
# background-loop.sh
for i in 1 2 3 4 5 6 7 8 9 10            # First loop.
do
  echo -n "$i "
done & # Run this loop in background.
       # Will sometimes execute after second loop.
echo   # This 'echo' sometimes will not display.
for i in 11 12 13 14 15 16 17 18 19 20   # Second loop.
do
  echo -n "$i "
done
echo   # This 'echo' sometimes will not display.
# ======================================================
# The expected output from the script:
# 1 2 3 4 5 6 7 8 9 10
# 11 12 13 14 15 16 17 18 19 20
# Sometimes, though, you get:
# 11 12 13 14 15 16 17 18 19 20
# 1 2 3 4 5 6 7 8 9 10 bozo $
# (The second 'echo' doesn't execute. Why?)
# Occasionally also:
```

```
# 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
# (The first 'echo' doesn't execute. Why?)
# Very rarely something like:
# 11 12 13 1 2 3 4 5 6 7 8 9 10 14 15 16 17 18 19 20
# The foreground loop preempts the background one.
exit 0
#  Nasimuddin Ansari suggests adding    sleep 1
#+ after the   echo -n "$i"   in lines 6 and 14,
#+ for some real fun.
```

[&&] AND逻辑操作符

- && AND逻辑操作符。在test测试结构中，如果操作符连接的测试条件都为真，则整个测试结果为0（真）。

选项 前缀

- 命令或者过滤器的选项标志。操作码的前缀。在参数替换中作为默认参数的前缀。
- 重定向（从标准输入或标准输出；到标准输入或标准输出）。下面是实际用到的例子：

```
（cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
# Move entire file tree from one directory to another
# [courtesy Alan Cox <a.cox@swansea.ac.uk>, with a minor change]
# 1) cd /source/directory
#    Source directory, where the files to be moved are.
# 2) &&
#   "And-list": if the 'cd' operation successful,
#    then execute the next command.
# 3) tar cf - .
#    The 'c' option 'tar' archiving command creates a new archive,
#    the 'f' (file) option, followed by '-' designates the target file
#    as stdout, and do it in current directory tree ('.').
# 4) |
#    Piped to ...
# 5) ( ... )
#    a subshell
# 6) cd /dest/directory
#    Change to the destination directory.
# 7) &&
#   "And-list", as above
# 8) tar xpvf -
#    Unarchive ('x'), preserve ownership and file permissions ('p'),
#    and send verbose messages to stdout ('v'),
#    reading data from stdin ('f' followed by '-').
#
#    Note that 'x' is a command, and 'p', 'v', 'f' are options.
#
# Whew!
```

```
# More elegant than, but equivalent to:
#   cd source/directory
#   tar cf - . | (cd ../dest/directory; tar xpvf -)
#
#     Also having same effect:
# cp -a /source/directory/* /dest/directory
#     Or:
# cp -a /source/directory/* /source/directory/.[^.]* /dest/directory
#     If there are hidden files in /source/directory.
bunzip2 -c linux-2.6.16.tar.bz2 | tar xvf -
#  --uncompress tar file--      | --then pass it to "tar"--
#  If "tar" has not been patched to handle "bunzip2",
#+ this needs to be done in two discrete steps, using a pipe.
#  The purpose of the exercise is to unarchive "bzipped" kernel source.
```

- 在下面的情况下"-"并不是一个bash操作符，而是被特定的UNIX工具如：tar，cat等识别的一个选项。

```
bash$ echo "whatever" | cat -
whatever
```

- 上面cat后一般跟文件，把文件换为"-"后，cat收到的内容将被重定向到标准输出。
- 加上"-"后使命令结果更加细节化。使用"-"让shell等待用户输入。
-

```
[root@centos8 /data]file -
123qwert
/dev/stdin: ASCII text
[root@centos8 /data]file -
#!/bin/bash
/dev/stdin: Bourne-Again shell script, ASCII text executable
```

- 下面是"-"和tar结合使用的一个例子：备份当前文件夹下24H内更改过的所有文件

```
#!/bin/bash
#  Backs up all files in current directory modified within last 24 hours
#+ in a "tarball" (tarred and gzipped file).
BACKUPFILE=backup-$(date +%m-%d-%Y)
#                 Embeds date in backup filename.
#                 Thanks, Joshua Tschida, for the idea.
archive=${1:-$BACKUPFILE}
#  If no backup-archive filename specified on command-line,
#+ it will default to "backup-MM-DD-YYYY.tar.gz."
tar cvf - `find . -mtime -1 -type f -print` > $archive.tar
gzip $archive.tar
echo "Directory $PWD backed up in archive file \"$archive.tar.g
```

```
#  Stephane Chazelas points out that the above code will fail
#+ if there are too many files found
#+ or if any filenames contain blank characters.
# He suggests the following alternatives:
# -------------------------------------------------------------------
#   find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$archive.tar"
#      using the GNU version of "find".
#   find . -mtime -1 -type f -exec tar rvf "$archive.tar" '{}' \;
#         portable to other UNIX flavors, but much slower.
# -------------------------------------------------------------------
exit 0
```

- 以"-"开头的文件名和重定向符"-"在一起使用时可能报错。脚本应该检查文件名是否以"-"开头，并加上合适的前缀。如：`./-FILENAME, $PWD/-FILENAME, 或者 $PATHNAME/-FILENAME`
- 如果某个变量的值以"-"开头，也可能造成错误。如：

```
var="-n"
echo $var
# bash认为上面的命令等同于 "echo -n", 不打印任何东西.
```

- "-"符号和cd命令使用时，表示前一个工作文件夹；此时使用到$OLDPWD这个环境变量。

```
[root@centos8 ~]pwd
/root
[root@centos8 ~]cd /etc/
[root@centos8 /etc]pwd
/etc
[root@centos8 /etc]cd -
/root
[root@centos8 ~]pwd
/root
```

- "-"在算术运算中做减法符号。

[--] 命令长选项

- 作为一个命令的长选项格式的一部分。
- 在和bash的builtin类型使用是表示命令选项的结束。可以用来删除某些以-开头的文件。如：

```
bash$ ls -l
-rw-r--r-- 1 bozo bozo 0 Nov 25 12:29 -badname #当前目录有个-开头的文件
bash$ rm -- -badname   # 使用--避免rm认为-badname为选项
bash$ ls -l
total 0
```

[=] 等号

- 赋值运算符

```
a=28
echo $a   # 28
```

- 等号也作为字符串比较操作符。

```
if [ "$a" = "$b" ] # 如果字符串$a和$b相同，则为真。注意等号两边空格
```

[+] 加号

- 加号，加法运算符。
- 正则表达式中加号表示匹配其前面的字符集一次或多次。类似*号，但是*号包括0次

[%] 百分号

- 百分号为取模运算符，即取除法的余数

```
let "z = 5 % 3"
echo $z  # 2
```

- 百分号也用于正则表达式中

[~] 波浪符(读：tilde)

- 波浪符代表家目录的意思，对应于$HOME环境变量。

```
[root@centos8 /etc/sysconfig/network-scripts] cd ~
[root@centos8 ~] pwd
/root

[root@centos8 /etc/sysconfig/network-scripts] ls ~
anaconda-ks.cfg      dead.letter  Documents  file1                lig    passwd
Public   Templates  Videos
anaconda-ks.cfg.bak  Desktop      Downloads  initial-setup-ks.cfg  Music  Pictures
scripts  tr
```

[~+] 当前工作目录$PWD

- "~+"表示当前工作目录，对应于内部变量$PWD。

```
[root@centos8 /etc/sysconfig/network-scripts] echo $PWD
/etc/sysconfig/network-scripts
[root@centos8 /etc/sysconfig/network-scripts] echo ~+
/etc/sysconfig/network-scripts
```

[~-] 前一个工作目录$OLDPWD

- "~-"表示前一个工作目录，对应于内部变量$OLDPWD。

```
[root@centos8 /etc/sysconfig/network-scripts] cd
[root@centos8 ~] echo ~+                          # 当前目录
/root
[root@centos8 ~] echo ~-                          # 前一个工作目录
/etc/sysconfig/network-scripts
[root@centos8 ~]
```

[=~] 正则表达式匹配符

- "=~"该正则表达式匹配符号用于两对中括号中。(Perl语言也有类似的操作符)。

```
#!/bin/bash
variable="This is a fine mess."
echo "$variable"
# Regex matching with =~ operator within [[ double brackets ]].
if [[ "$variable" =~ T.........fin*es* ]]
# NOTE: As of version 3.2 of Bash, expression to match no longer quoted.
then
  echo "match found"
      # match found
fi
```

或者更实用的例子：

```
#!/bin/bash
input=$1
if [[ "$input" =~ "[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]" ]]
#                 ^ NOTE: Quoting not necessary, as of version 3.2 of Bash.
# NNN-NN-NNNN (where each N is a digit).
```

```
then
  echo "Social Security number."
  # Process SSN.
else
  echo "Not a Social Security number!"
  # Or, ask for corrected input.
fi
```

[^] 行开始位置

- 在正则表达式中，"^"表示一行的开头。

```
#### 找出/etc/passwd文件中以r开头的行
[root@centos8 ~] grep "^r.*" /etc/passwd
root:x:0:0:steve,banzhuang,18800001111,1123443,:/root:/bin/bash
rtkit:x:172:172:RealtimeKit:/proc:/sbin/nologin
rpc:x:32:32:Rpcbind Daemon:/var/lib/rpcbind:/sbin/nologin
radvd:x:75:75:radvd user:/:/sbin/nologin
rpcuser:x:29:29:RPC Service User:/var/lib/nfs:/sbin/nologin
```

[^,^^]

- 在参数替换中起将字符串中的字母转换成大写的作用（在bash4版本中引进。）

```
#!/bin/bash4
var=veryMixedUpVariable
echo ${var}             # veryMixedUpVariable
echo ${var^}            # VeryMixedUpVariable
#         *              第一个字符大写
#         *                First char --> uppercase.
echo ${var^^}           # VERYMIXEDUPVARIABLE
#         **             所有字符大写
#         **               All chars  --> uppercase.
echo ${var,}            # veryMixedUpVariable
#         *              第一个字符小写
#         *                First char --> lowercase.
echo ${var,,}           # verymixedupvariable
#         **             所有字符小写
#         **               All chars  --> lowercase.
```

[Whitespace] 空白符

- 空白符一般作为命令或者变量之间的分割符。空白符包含空格、退格、空白行或者这些的组合。

- 在一些环境下不允许出现空白符，如变量赋值等。