

特殊字符

笨办法学 

概述

- | | |
|-------|------|
| ▶ # | ▶ () |
| ▶ ; | ▶ {} |
| ▶ . | ▶ [] |
| ▶ "" | ▶ > |
| ▶ '' | ▶ |
| ▶ , | ▶ |
| ▶ / \ | ▶ & |
| ▶ ` | ▶ - |
| ▶ : | ▶ = |
| ▶ ! | ▶ + |
| ▶ * | ▶ % |
| ▶ ? | ▶ ~ |
| ▶ \$ | ▶ ^ |

当做注释的多，但也有例外

```
root@tomlab1:~  
1 #!/bin/bash  
2  
3 # This is a comment.  
4  
5 echo "A commnet will follow " # 注释在这里  
6 # 注意上一条语句中，#号前面要有一个空格  
7  
8         # A tab precede this comment  
9 # 注意上一条语句中，注释是放在本行 行首空白的后面  
10  
11 # echo中被引号的#，是不能被当作的注释的  
12 # echo中被转义的#，是不能被当作的注释的  
13 # 反斜线是转义字符  
14  
15 echo "111-双引号 The # here does not begin a comment."  
16 echo '222-单引号 The # here does not begin a comment.'  
17 echo 333-无引导且有转义 The \# here does not begin a comment.  
18 echo 444-无引号无转义 The # here begin a comment.  
19  
20 # 在特定的参数替换结构中，#号不是注释  
21 echo ${PATH#*:} # 效果：将第一个冒号之前的删除  
22  
23 # 在数字常量表达式中，#号不是注释  
24 echo $((2#101011)) # 数制转换，2进制转换为10进制，这不是注释
```

24,80-62 All

;
是命令分隔符，在同一行上写两个或两个以上的命令

```
root@tomlab1:~  
1 #!/bin/bash  
2  
3 # 分号(separator [semicolon])测试  
4 echo hello; echo there  
5  
6 filename='mytestfile.txt'  
7  
8 if [ -e "$filename" ]; then # 注意: if 和 then需要分隔  
9     echo "File $filename exists." ; cp $filename $filename.bak  
10 else  
11     echo "File $filename not found." ; touch $filename  
12 fi; echo "File test complete."  
~  
~  
"03-002separator.sh" 12L, 315C written 12,30 All
```

:: 双分号 double semicolon , 是case代码块的结束符

```
root@tomlab1:~  
1 #!/bin/bash  
2 # 双分号 double semicolon 示例  
3  
4 variable='abc'  
5  
6 case "$variable" in  
7 abc) echo "\$variable = abc";;  
8 xyz) echo "\$variable = xyz";;  
9 esac  
10  
11 exit  
~  
~  
-- INSERT --
```

• 点 (句点) 命令 dot command (period)

```
root@tomlab1:~  
1 #!/bin/bash  
2 # 点 命令测试, 点命令与source命令效果相同  
3  
4 . 03-004dot.txt # 加载一个数据文件  
5 # 这与 source 03-004dot.txt 效果相同  
6 # 03-004dot.txt 必须存在于当前的工作目录  
7  
8 # 下面, 引用数据文件中定义的一个变量  
9 echo D1  
10  
11 exit  
~  
~  
"03-004dot.sh" 11L, 281C 11,1 All
```

cat 03-004dot.txt
D1=111
D2=222
D3=333

茴 茴 茴 茴

• 点作为文件名的一部分

component of a filename

▶ 隐藏文件

```
# touch .hidden-file

# ls -l
total 0

# ls -la
total 4
drwxr-xr-x  2 root root   26 Oct 13 18:36 .
dr-xr-x---  6 root root 4096 Oct 13 18:36 ..
-rw-r--r--  1 root root    0 Oct 13 18:36 .hidden-file
```

▶ 目录名

```
# pwd
/root/tmp

# cd .
# pwd
/root/tmp

# cd ..
# pwd
/root
```


• 点是正则表达式中的匹配字符 regular expression

"13." 匹配: 13 + 至少一个任意字符 (包括空格):

示例:

匹配: 1133, 11333

不匹配: 13 (因为缺少"."所能匹配的至少一个任意字符)



引用 quoting

- ▶ " 部分引用 partial quoting
 - ▶ 阻止STRING中大部分特殊的字符的解释
 - ▶ 但是会发生变量替换
 - ▶ 弱引用 weak quoting
- ▶ ' 全引用 full quoting
 - ▶ 阻止STRING中所有特殊字符的解释
 - ▶ 包括不会发生变量替换
 - ▶ 强引用 strong quoting

```
# ls -l [Mm]*
-rw-r--r-- 1 root root 0 Oct 11 15:51 message
-rw-r--r-- 1 root root 0 Oct 12 18:11 mytestfile.txt

# ls -l "[Mm]*"
ls: cannot access [Mm]*: No such file or directory

# ls -l '[Mm]*'
ls: cannot access [Mm]*: No such file or directory
```

```
# echo $PWD
/root

# echo "$PWD"
/root

# echo '$PWD'
$PWD
```

示例：引用与赋值

```
root@tomlab1:~  
1 #!/bin/bash  
2 # 引用练习 + 变量赋值练习  
3  
4 a=123  
5 hello=$a  
6  
7 # -----  
8 # 强烈注意：赋值时，等号前后一定要不要空格  
9 #  
10 # 1、如果等号前面有空格？  
11 #     VARIABLE =value  
12 #     将执行带一个参数=value的命令 VARIABLE  
13 #  
14 # 2、如果等号后面有空格？  
15 #     VARIABLE= value  
16 #     将执行后面这个小写的value命令，并且带一个赋值为""的变量VARIABLE  
17 # -----  
18  
19 echo hello # 这不是一个变量，所以只会输出hello  
20  
21 echo $hello  
22 echo ${hello} # 结果同上一行  
23  
24 echo "$hello"  
25 echo '$hello'  
26  
27 echo  
28 exit  
29  
"03-005quoting.sh" 29L, 685C 28,4 All
```

逗号操作符 comma operator

- ▶ 用于连接多个的算术操作，并返回最后一项
- ▶ 示例1

```
let "t1 = ((a = 9, 15 / 3)) "  
echo "t1 = $t1"
```

- ▶ 示例2

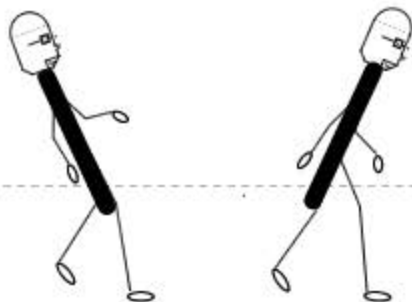
```
let "t2 = ((5 + 3, 7 - 1, 15 - 4)) "  
echo "t2 = $t2"
```

- ▶ 示例3

```
let "t3 = ((a = 9, 15 / 3)) "  
echo "t3 = $t3      a = $a"
```



斜线



\ 反斜线 backslash 转义符

- ▶ “怪→正常”

```
\$      \"      \'
```

- ▶ “正常→怪”

\n	新的一行
\r	回车
\t	水平制表符
\v	垂直制表符
\b	后退符
\a	"alert"(蜂鸣或者闪烁)
\0xx	转换为八进制的ASCII码, 等价于0xx

/ 正斜线 forward slash

- ▶ 文件名路径分隔符

```
# cd /etc/sysconfig/network-scripts/
```

- ▶ 除法算术操作符

反引号 backquote, 命令替换 command substitution

```
root@tomlab1:~  
1 #!/bin/bash  
2 # 本脚本的名称为: 03-006backquote.sh  
3 # 用于测试反引号backquote的命令替换command substitution功能  
4  
5 echo $0  
6  
7 script_name=`basename $0`  
8 echo $script_name  
9  
10 script_name=`basename -s .sh $0`  
11 echo $script_name  
12 exit
```

"03-006backquote.sh" 12 lines --100%-- 12,4 All

```
# /root/03-006backquote.sh  
/root/03-006backquote.sh  
03-006backquote.sh  
03-006backquote
```

```
# ./03-006backquote.sh  
./03-006backquote.sh  
03-006backquote.sh  
03-006backquote
```


: 冒号colon , 空命令 null command

```
root@tomlab1:~  
1 #!/bin/bash  
2 # 示例代码: 冒号 colon, 空命令 null command  
3  
4 # 例1: 死循环 Endless loop  
5  
6 while : # 本行等同于 while true  
7 do  
8     operation-1  
9     operation-2  
10    operation-3  
11 done  
12  
13 # 例2: if/then的占用符 placeholder  
14  
15 if condition  
16 then : # 什么都不做, 引出分支, 有可能以后再补充  
17 else  
18     take-some-action  
19 fi  
20  
21 # 例3: 清空一个文件, 但不会修改该文件的权限, 也cat /dev/null类似  
22 #     由于空命令是一个内建的命令, 所以不会生产一个新进程  
23  
24 : > testdata.txt  
25  
"03-007colon.sh" 25L, 532C written 25,0-1 All
```

! 取反操作符 reverse

- ▶ 反转命令的退出状态码
- ▶ 反转测试操作符的意义

```
root@tomlab1:~  
1 #!/bin/bash  
2 # 叹号! 取反操作符 与 退出状态码 的练习  
3  
4 # true 与 冒号都是什么都不做的命令  
5 true  
6 echo "Exit status of \"true\" = $?"  
7  
8 ! true    # 注意 感叹号! 与 命令之间空格  
9 echo "Exit status of \"! true\" = $?"  
10  
11  
12 # 退出状态码必须是十进制数, 范围是0 - 255  
13 exit 88  
1,1 All
```

```
# ./NegatingCondition.sh  
Exit status of "true" = 0  
Exit status of "! true" = 1  
  
# echo $?  
88
```


* 星号 asterisk

- ▶ 文件名通配符 wild card、globbing

```
bash$ ls -l *  
bash$ ls -l [ab]*  
bash$ ls -l [a-c]*  
bash$ ls -l [^ab]*  
bash$ ls -l {b*,c*,*est*}
```

- ▶ 正则表达式, metacharacter、regular expression
 - ▶ 匹配任意个数(包含0个)的字符
 - ▶ 示例: 1133*
- ▶ 乘法 multiplication
 - ▶ ** 幂 exponentiation



? 问号 question mark

- ▶ 文件名通配符 wild card、globbing

```
bash$ ls -l t?.sh
```

- ▶ 正则表达式, metacharacter、regular expression
 - ▶ 匹配它前面的字符, 1次或0次。通常用来匹配单个字符
- ▶ 测试操作符 test operator
 - ▶ 在双圆括号结构表达式中, 用来测试一个条件的结果 (三元操作符)

```
(( var0 = var1<98?9:21 ))
```

=

```
if [ "$var1" -lt 98 ]  
then  
    var0=9  
else  
    var0=21  
fi
```

- ▶ 在参数替换表达式中, 用来测试一个变量是否被设置

((...)) 双圆括号结构

- ▶ 与let命令很相似, 允许算术扩展和赋值.

```
a=$(( 5 + 3 ))
```

- ▶ 在Bash中, 使用C语言风格变量操作的一种处理机制

```
(( a = 23 )) # 变量赋值, "="两边允许有空格.  
(( a++ ))    # 后置自加  
(( a-- ))    # 后置自减  
(( ++a ))    # 前置自加  
(( --a ))    # 前置自减
```



`{ }` 参数替换

Parameter substitution

- ▶ 如果变量未被声明或赋值，那么就输出错误信息
 - ▶ `${parameter?err_msg}` 如果变量未声明.....
 - ▶ `${parameter:?err_msg}` 如果变量声明了，但未赋值.....

正常地进行变量定义与赋值

```
bash$ var1=123
```

```
bash$ echo $var1  
123
```

```
bash$ echo ${var1}  
123
```

①

变量未定义

```
bash$ echo ${var2}  
无输出
```

```
bash$ echo ${var2?err_msg}  
-bash: var2: err_msg
```

```
bash$ echo ${var2:?err_msg}  
-bash: var2: err_msg
```

②

仅定义了变量，但未赋值

```
bash$ var3=  
bash$ echo ${var3}
```

```
bash$ echo ${var3?err_msg}  
无输出
```

```
bash$ echo ${var3:?err_msg}  
-bash: var3: err_msg
```

③

如果一个环境变量或自定义变量没有被定义....

▶ 问题：

正常地进行变量定义与赋值

```
bash $ ORA_HOME=""  
bash $ if [ $ORA_HOME = "" ]; then echo "111"; fi  
111
```

如果没有定义变量

~~bash \$ ORA_HOME=""~~

```
bash $ if [ $ORA_HOME = "" ]; then echo "111"; fi  
111
```

▶ 对策：

- ▶ 使用问号?来测试一个变量是否被设置
- ▶ 如果没有被设置，就输出一个错误退出脚本
- ▶ 这个错误信息是可以自定义的



\$ 有多种用途

- | | |
|---------------------|-------------------------|
| ▶ \$ 变量替换(引用变量的内容) | Variable substitution |
| ▶ \$ 正则表达式中的行结束符 | Regular expression |
| ▶ \${ } 参数替换 | Parameter substitution |
| ▶ \$' ... ' 引用字符串扩展 | Quoted string expansion |
| ▶ \$*, @\$ 位置参数 | Positional parameters |
| ▶ \$? 退出状态码变量 | Exit status variable |
| ▶ \$\$ 进程ID变量 | Process ID variable |



\$ 变量替换(引用变量的内容). Variable substitution

- ▶ 变量的名字就是变量保存值的地方
- ▶ 引用变量的值就叫做变量替换
- ▶ 在以下情况下，变量名没有前缀名\$
 - ▶ 变量被声明或被赋值
 - ▶ 变量被**unset**
 - ▶ 变量被**export**
 - ▶ 变量代表一种信号
- ▶ 在引用时，\$的变化：
 - ▶ 双引号 (" ")，弱引用，发生变量替换
 - ▶ 单引号 (' ')，强引用，保持字面意思

```
1 bash$ variable1=23

2 bash$ echo variable1
variable1

3 bash$ echo $variable1
23

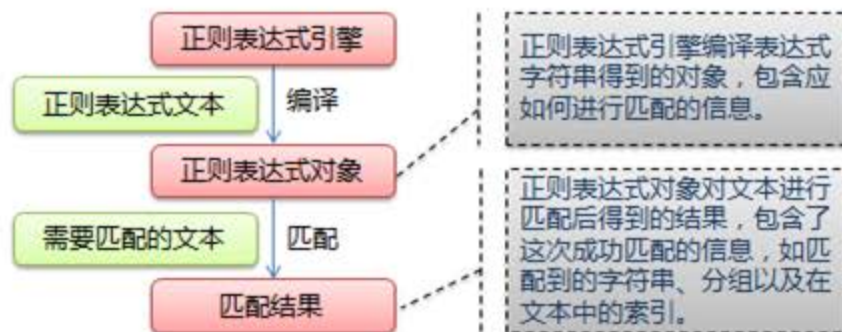
4 bash$ echo ${variable1}
23
```


\$ 正则表达式中的行结束符

Regular expression

在RE中用来匹配行尾，例如：

- ▶ `"abc$"` 匹配行尾的abc
- ▶ `"^$"` 匹配空行



`${}` 参数替换

Parameter substitution

▶ 如果变量未被声明或赋值，那么就输出错误信息

▶ `${parameter?err_msg}`

▶ `${parameter:?err_msg}`

▶ 如果变量未被声明或赋值，那么就替换为默认值

▶ `${parameter-default}`

▶ `${parameter:-default}`

```
echo ${username-`whoami`}
```

```
DEFAULT_FILENAME=generic.data  
filename=${1:-$DEFAULT_FILENAME}
```

\$ 有多种用途

- ▶ ~~\$~~ 变量替换(引用变量的内容). ~~Variable substitution~~
- ▶ ~~\$~~ 正则表达式中的行结束符 ~~Regular expression~~
- ▶ ~~\${}~~ 参数替换. ~~Parameter substitution~~
- ▶ \$' ... ' 引用字符串扩展 Quoted string expansion
- ▶ \$# , \$* , @\$ 位置参数 Positional parameters
- ▶ \$? 退出状态码变量 Exit status variable
- ▶ \$\$ 进程ID变量 Process ID variable

```
bash$ quote=$'\042'  
  
bash$ echo $quote  
"
```



\$#, \$*, \$@, ... 位置参数

- ▶ \$0, \$1, \$2, 等等
 - ▶ 位置参数, 从命令行传递到脚本, 或者传递给函数
- ▶ \$#
 - ▶ 命令行参数或者位置参数的个数
- ▶ \$*
 - ▶ 所有的位置参数都被看作为一个单词
- ▶ \$@
 - ▶ 与\$*相同, 但是每个参数都是一个独立的引用字符串

./ScriptName



\$ 有多种用途

- ▶ ~~\$ — 变量替换(引用变量的内容). — Variable substitution~~
- ▶ ~~\$ — 正则表达式中的行结束符 — Regular expression~~
- ▶ ~~\${} 参数替换. — Parameter substitution~~
- ▶ ~~\$' ... ' 引用字符串扩展 — Quoted string expansion~~
- ▶ ~~\$#, \$*, @\$ 位置参数 — Positional parameters~~
- ▶ \$? 退出状态码变量
Exit status variable
- ▶ \$\$ 进程ID变量
Process ID variable



() 圆括号 parenthesis

▶ 命令组

```
bash$ a=123
```

```
bash$ (echo "a = $a"; a=321; echo "a = $a" )
```

```
a = 123
```

```
a = 321
```

```
bash$ echo "a = $a"
```

```
a = 123
```



▶ 数组初始化

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

{ } 大括号 花括号 brace

▶ 扩展 expansion

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

# 拷贝"file1.txt"到"file1.backup"中
cp file1.{txt,backup}

# 输出26个英文字母、阿拉伯数字
echo {a..z} {0..9}
```

▶ 代码块 Block of code

- ▶ 变称为inline group, 相当于创建了一个匿名函数(没有名字的函数)
- ▶ 与“标准”函数不同, 在其中声明的变量, 对于脚本其他部分的来说是可见的

▶ {} \: 路径名 pathname

示例：代码块与IO重定向

```
root@tomlab1:~  
1 #!/bin/bash  
2 # 大括号代码块 和 IO重定向  
3  
4 # 从/etc/fstab中读行  
5  
6 File=/etc/fstab  
7  
8 {  
9 read line1  
10 read line2  
11 } <$File  
12  
13 echo "First line if $File is:"  
14 echo "$line1"  
15 echo  
16  
17 echo "Second line if $File is:"  
18 echo "$line2"  
19  
20 exit 0  
~  
"03-01CodeBlockIORedir.sh" 20 lines --100%-- 20,6 All
```

[] 中括号 方括号 bracket

▶ 条件测试 Test

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

- ▶ 扩展测试[[]] Test
- ▶ 数组元素 Array element
- ▶ 字符范围 Range of characters



[] 中括号 方括号 bracket

▶ ~~条件测试~~ ——— Test

▶ 扩展测试[[]] Test

```
1 file=/etc/passwd  
  
2 if [[ -e $file ]]  
3 then  
4   echo "Password file exists."  
5 fi
```

▶ 数组元素 Array element

▶ 字符范围 Range of characters



[] 中括号 方括号 bracket

- ▶ 条件测试 ————— Test
- ▶ 扩展测试[[]] ——— Test
- ▶ 数组元素 Array element

```
Array[1]=slot_1  
echo ${Array[1]}
```

- ▶ 字符范围 Range of characters



[] 中括号 方括号 bracket

- ▶ 条件测试 ——— Test
- ▶ 扩展测试[[]] ——— Test
- ▶ 数组元素. ——— Array element
- ▶ 字符范围. Range of characters

[xyz]	匹配字符x, y, 或z
[c-n]	匹配字符c到字符n之间的任意一个字符
[B-Pk-y]	匹配从B到P, 或者从k到y之间的任意一个字符
[a-z0-9]	匹配任意小写字母或数字
[^b-d]	将会匹配范围在b到d之外的任意一个字符
[Yy][Ee][Ss]	能够匹配yes, Yes, YES, yEs, 等等
[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]	匹配美国的社保码



> &> >& >> < <>

▶ 重定向 redirection

重定向操作符	功能
< 文件名	重定向输入
> 文件名	重定向输出
>> 文件名	追加输出
2> 文件名	重定向标准错误输出
2>> 文件名	重定向和追加标准错误输出
&> 文件名	重定向标准输出和标准错误输出（首选方式*）
>& 文件名	重定向标准输出和标准错误输出
2>&1	将标准错误输出重定向到标准输出的去处
1>&2	将输出重定向到标准错误输出的去处
>	重定向输出时忽略noclobber
<> 文件名	如果是一个设备文件，使用文件作为标准输入和标准输出

- 1、`stdin`、`stdout`和`stderr`的文件描述符分别是0、1、2
- 2、文件描述符与>之间没有空格
- 3、文件描述符默认值是1，1>可以简写为 >
- 4、> 后面的文件描述符前面一定要有&，否则会当作普通文件
- 5、&> file 与 >& file意思完全相同，都等价于 >file 2>&1

TIP!

▶ 进程替换 process substitution

▶ 比较操作符 comparison operator



> &> >& >> < <>

▶ 重定向 redirection

示例：

```
1 scriptname > filename
2 scriptname >> filename
3 scriptname 2>&1
4 command &> filename
5 command 1>&2 简写为 command >&2
6 ls 2>a1 >&2 等同 ls >a1 2>&1
```

- ▶ 进程替换 process substitution
- ▶ 比较操作符 comparison operator



> &> >& >> < <>

▶ 重定向 ~~redirection~~

▶ 进程替换 process substitution

▶ 命令替换：把一个命令的结果赋值给一个变量

```
dir_contents=`ls -al`
```

▶ 进程替换：把一个进程的输出提供给另一个进程

```
>(command)
```

```
<(command)
```

▶ 比较操作符 comparison operator



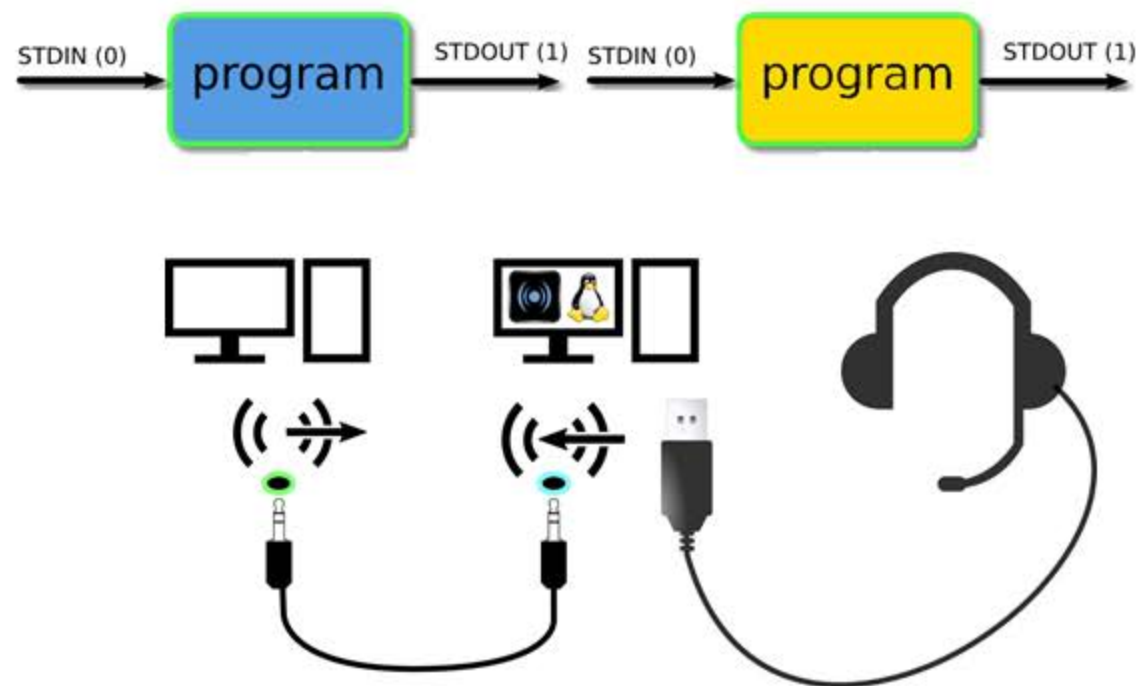
> &> >& >> < <>

- ▶ 重定向 — redirection
- ▶ 进程替换 — process substitution
- ▶ 比较操作符 — comparison operator
 - ▶ 字符串比较
 - ▶ 按ASCII字符进行排序来比较
 - ▶ 整数比较



```
root@tomlab1:~  
1 #!/bin/bash  
2 # ASCII 字符<>比较  
3  
4 veg1=carrots  
5 veg2=tomatoes  
6  
7 if [[ "$veg1" < "$veg2" ]]  
8 then  
9     echo "$veg1 < $veg2"  
10 else  
11     echo "What kind of dictionary are you using, anywho?"  
12 fi  
13  
14 exit 0  
~  
~  
~  
"03-008ASCIIComparison.sh" 14L, 192C 14,1 All
```

| 管道 pipe



示例：

```
1 cat *.lst | sort | uniq
```

```
2 cat file1 file2 | ls -l | sort
```

|| 或-逻辑操作 OR logical operator

- 在条件测试结构中，如果条件测试结构两边中的任意一边结果为true的话，||操作就会返回0(代表执行成功)。

示例：

```
1 if [ $condition1 ] || [ $condition2 ]  
2 if [ $condition1 -o $condition2 ]  
3 if [ $condition1 || $condition2 ] ❌  
4 if [[ $condition1 || $condition2 ]]
```

& 后台运行命令 Run job in background

- 命令后边跟一个&，表示在后台运行

```
root@tomlab1:~  
1 #!/bin/bash  
2 # 03-03backgroud-loop.sh  
3  
4 for i in 1 2 3 4 5 6 7 8 9 10 # 第一个循环  
5 do  
6     echo -n "$i "  
7 done &    # 在后台执行这个循环  
8  
9 echo  
10  
11 for i in 11 12 13 14 15 16 17 18 19 20 # 第二个循环  
12 do  
13     echo -n "$i "  
14 done     # 正常地在前台执行这个循环  
15  
16 echo  
17 exit 0  
~  
~  
"03-03backgroud-loop.sh" 17L, 289C      17,6      All
```


&& 与-逻辑操作 AND logical operator

- 在条件测试结构的两边结果都为true时，&&才返回0(代表success)

示例：

```
1 if [ $condition1 ] && [ $condition2 ]  
2 if [ $condition1 -a $condition2 ]  
3 if [ $condition1 && $condition2 ] ❌  
4 if [[ $condition1 && $condition2 ]]
```

- 连字符Hyphen、破折号Dash、减号 Minus

▶ 选项, 前缀

```
1 ls -al
2 sort -dfu $filename
3
4 sort --ignore-leading-blanks
5
6 set --$variable
```

Option, Prefix

```
1 if [ $file1 -ot $file2 ]
2 then
3     echo "File $file1 is older than $file2."
4 fi
5
6 param2=${param1:-$DEFAULTVAL}
```

- ▶ 重定向stdin或stdout
- ▶ 先前的工作目录
- ▶ 减号

Redirection

Previous working directory

Minus

- 连字符Hyphen、破折号Dash、减号 Minus

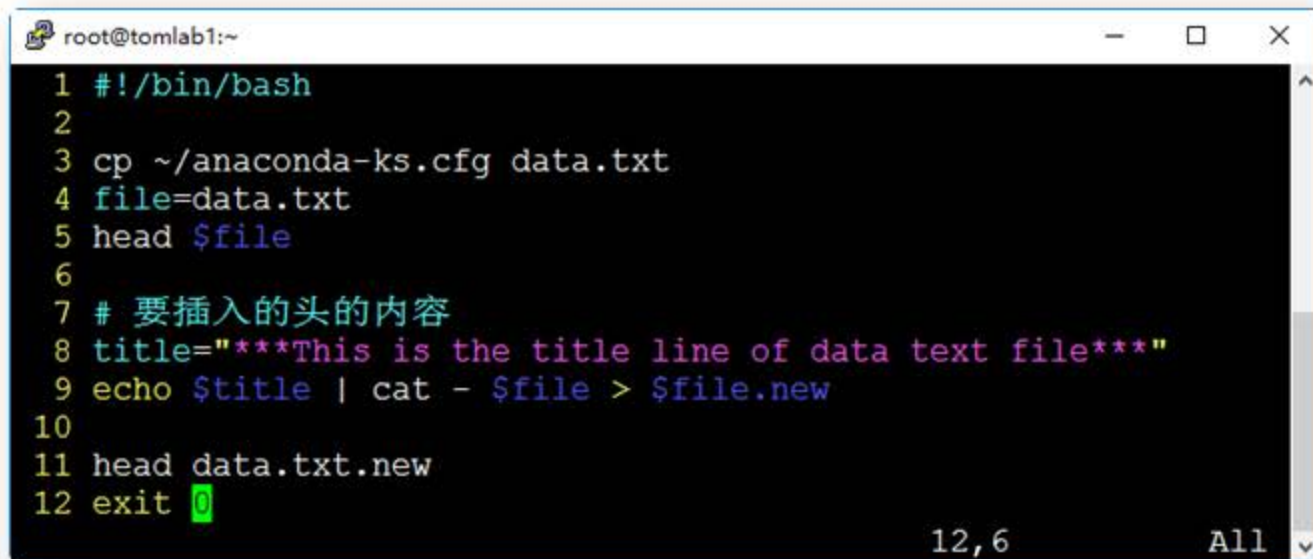
- ▶ ~~选项, 前缀~~ Option, Prefix
- ▶ 重定向stdin或stdout Redirection

```
1 echo "whatever" | cat -  
  
2 bunzip2 -c linux-2.6.16.tar.bz2 | tar xvf -  
  
3 (cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)  
  
4 cp -a /source/directory/* /dest/directory  
  
5 cp -a /source/directory/* /source/directory/.[^.]* /dest/directory
```

- ▶ 先前的工作目录 Previous working directory
- ▶ 减号 Minus

- 连字符Hyphen、破折号Dash、减号 Minus

- ▶ 选项, 前缀 Option, Prefix
- ▶ 重定向stdin或stdout Redirection



```
root@tomlab1:~  
1 #!/bin/bash  
2  
3 cp ~/anaconda-ks.cfg data.txt  
4 file=data.txt  
5 head $file  
6  
7 # 要插入的头的内容  
8 title="***This is the title line of data text file***"  
9 echo $title | cat - $file > $file.new  
10  
11 head data.txt.new  
12 exit 0
```

- ▶ 先前的工作目录 Previous working directory
- ▶ 减号 Minus

示例：备份最近一天当前目录下所有修改的文件

```
root@tomlab1:~  
1 #!/bin/bash  
2  
3 # 备份最近一天当前目录下所有修改的文件  
4 # 使用-的stdin、stdout, 及tar+gzip的手段  
5  
6 # 默认的备份文件名, 嵌入当前的时间  
7 BACKUPFILE=backup-$(date +%Y-%m-%d)  
8  
9 # 如果没有传递参数给此脚本, 则使用默认的文件名  
10 archive=${1:-$BACKUPFILE}  
11  
12 tar cvf - `find . -mtime -1 -type f -print` > $archive.tar  
13 gzip $archive.tar  
14  
15 # 如果当前目录中文件过多、或文件名包括空格时, 有可能执行失败  
16 # 可以考虑修改为:  
17 # find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$archive.tar"  
18  
19 echo "Directory $PWD backup up in archive file \"$archive.tar.gz\"."  
20  
21 exit 0  
~  
~  
"03-04BackFiles.sh" 21L, 647C 17,1 All
```


- 连字符Hyphen、破折号Dash、减号 Minus

- ▶ ~~选项, 前缀~~ ~~Option, Prefix~~
- ▶ ~~重定向stdin或stdout~~ ~~Redirection~~
- ▶ 先前的工作目录 Previous working directory
 - ▶ `cd -` 回到先前的工作目录
 - ▶ 其实是使用环境变量 `$OLDPWD`
- ▶ 减号 Minus

= 等号 Equal

▶ 等号 Equal

```
1 a=28  
2 echo $a
```

▶ 字符串比较操作 String Comparison Operator

```
1 if [ "$a" = "$b" ]
```

+ 加号 Plus

- ▶ 加法算术操作 Addition arithmetic operator
- ▶ 正则表达式 Regular Expression

```
1 echo a111b | sed -ne '/a1\+b/p'
2 echo a111b | grep 'a1\+b'
3 echo a111b | gawk '/a1+b/'
```

- ▶ 选项 Option

```
# echo $BADARG
无输出
# set -u
# echo $BADARG
-bash: BADARG: unbound variable
# set +u
# echo $BADARG
无输出
```

七加一，七减一，加完减完等于几？

七加一，七减一，加完减完还是七。

% 百分号 Percent

▶ 取模 Modulo

```
bash$ expr 5 % 3  
2
```

▶ 模式匹配 Pattern matching

```
1 echo ${var%Pattern}  
  
2 echo ${var%%Pattern}
```

~ 波浪号 Tilde

- ▶ ~ 家目录 Home
 - ▶ \$HOME
- ▶ ~+ 当前工作目录 Current working directory
 - ▶ \$PWD
- ▶ ~- 先前的工作目录 Previous working directory
 - ▶ \$OLDPWD
- ▶ =~ 正则表达式匹配 Regular expression match

~ 波浪号 Tilde

- ▶ ~ 家目录 Home
 - ▶ \$HOME
- ▶ ~+ 当前工作目录 Current working directory
 - ▶ \$PWD
- ▶ ~- 先前的工作目录 Previous working directory
 - ▶ \$OLDPWD
- ▶ =~ 正则表达式匹配 Regular expression match

^ 脱字号，补字号 Caret

- ▶ 行首 Beginning of line
- ▶ 大写转换 Uppercase conversion

```
$ bash --version
GNU bash, version 4.2.46(2)-release
```

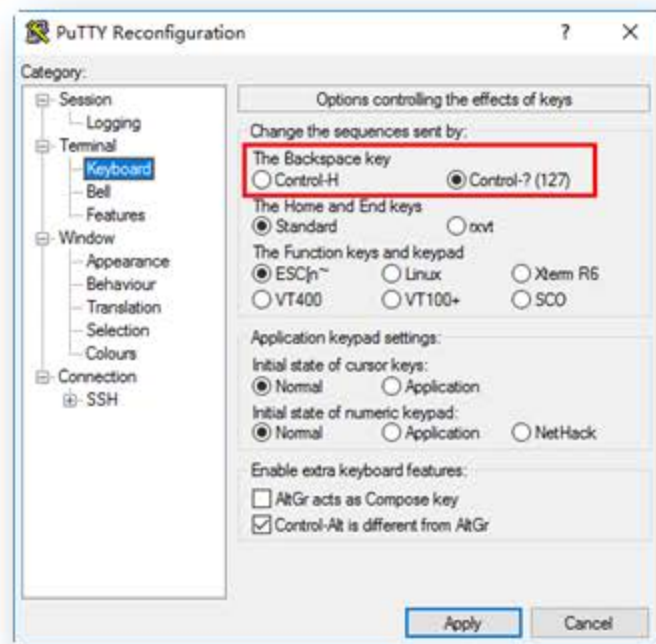
```
$ var=veryMixedUpVariable
```

```
$ echo ${var}
veryMixedUpVariable
```

```
$ echo ${var^}
VeryMixedUpVariable
```

```
$ echo ${var^^}
VERYMIXEDUPVARIABLE
```

- ▶ 控制字符 Control Characters



小测试

- | | |
|-------|------|
| ▶ # | ▶ () |
| ▶ ; | ▶ {} |
| ▶ . | ▶ [] |
| ▶ "" | ▶ > |
| ▶ '' | ▶ |
| ▶ , | ▶ |
| ▶ / \ | ▶ & |
| ▶ ` | ▶ - |
| ▶ : | ▶ = |
| ▶ ! | ▶ + |
| ▶ * | ▶ % |
| ▶ ? | ▶ ~ |
| ▶ \$ | ▶ ^ |

其它课程

