第十二章：shell 编程

尚硅谷云计算 Linux 课程

版本：V1.0

讲师：沈超

一、正则表达式

1、概述

还记得我们在上一章说过正则表达式和通配符的区别（正则表达式用来在文件中匹配符合条件的字符串，通配符用来匹配符合条件的文件名）吗？其实这种区别只在 Shell 当中适用，因为用来在文件当中搜索字符串的命令，如 grep、awk、sed 等命令可以支持正则表达式，而在系统当中搜索文件的命令，如 ls、find、cp 这些命令不支持正则表达式，所以只能使用 shell 自己的通配符来进行匹配了。

2 基础正则表达式

元字符	作 用
*	前一个字符匹配 0 次或任意多次。
.	匹配除了换行符外任意一个字符。
^	匹配行首。例如：^hello 会匹配以 hello 开头的行。
$	匹配行尾。例如：hello&会匹配以 hello 结尾的行。
[]	匹配中括号中指定的任意一个字符，只匹配一个字符。 例如：[aoeiu] 匹配任意一个元音字母，[0-9] 匹配任意一位数字，[a-z][0-9]匹配小写字和一位数字构成的两位字符。
[^]	匹配除中括号的字符以外的任意一个字符。例如：[^0-9] 匹配任意一位非数字字符，[^a-z] 表示任意一位非小写字母。
\	转义符。用于取消讲特殊符号的含义取消。
\{n\}	表示其前面的字符恰好出现 n 次。例如：[0-9]\{4\} 匹配 4 位数字，[1][3-8][0-9]\{9\} 匹配手机号码。
\{n,\}	表示其前面的字符出现不小于 n 次。例如： [0-9]\{2,\} 表示两位及以上的数字。
\{n,m\}	表示其前面的字符至少出现 n 次，最多出现 m 次。例如：[a-z]\{6,8\} 匹配 6 到 8 位的小写字母。

在~/.bashrc 文件中建立这个别名：

```
[root@localhost ~]# vi /root/.bashrc
alias grep='grep --color=auto'
```

1）、 练习文件建立

```
[root@localhost ~]# vi test_rule.txt
Mr. Li Ming said:
he was the most honest man.
123despise him.
```

```
But since Mr. shen Chao came,
he never saaaid those words.
5555nice!

because,actuaaaally,
Mr. Shen Chao is the most honest man

Later,Mr. Li ming soid his hot body.
```

2）、 “*”前一个字符匹配 0 次，或任意多次

```
[root@localhost ~]# grep "a*" test_rule.txt
Mr. Li Ming said:
he was the most honest man.
123despise him.

But since Mr. shen Chao came,
he never saaaid those words.
5555nice!

because,actuaaaally,
Mr. Shen Chao is the most honest man

Later,Mr. Li ming soid his hot body.
```

如果这样写正则表达式“aa*”代表这行字符串一定要有一个 a，但是后面有没有 a 都可以。也就是说会匹配至少包含有一个 a 的行：

```
[root@localhost ~]# grep "aa*" test_rule.txt
Mr. Li Ming said:
he was the most honest man.
But since Mr. shen Chao came,
he never saaaid those words.
because,actuaaaally,
Mr. Shen Chao is the most honest man
Later,Mr. Li ming soid his hot body
```

如果正则表达式是“aaa*”，则会匹配最少包含两个连续 a 的字符串，如：

```
[root@localhost ~]# grep "aaa*" test_rule.txt
he never saaaid those words.
because,actuaaaally,
```

如果正则表达式是“aaaaa*”，则会匹配最少包含四个个连续 a 的字符串，如：

```
[root@localhost ~]# grep "aaaaa*" test_rule.txt
because,actuaaaally,
```

当然如果再多写一个 a，如“aaaaaa*”就不能从这篇文档中匹配任何内容了，因为我们这篇文档中 a 最多的单词“actuaaaally”只有四个个连续的 a，而“aaaaaa*”会匹配最少五个连续的 a。

3）、 “.” 匹配除了换行符外任意一个字符

正则表达式“.”只能匹配一个字符，这个字符可以是任意字符，举个例子：

```
[root@localhost ~]# grep  "s..d" test_rule.txt
Mr. Li Ming said:
Later,Mr. Li ming soid his hot body.
#“s..d”会匹配在 s 和 d 这两个字母之间一定有两个字符的单词

[root@localhost ~]# grep "s.*d" test_rule.txt
Mr. Li Ming said:
he never saaaid those words.
Later,Mr. Li ming soid his hot body.
#最后一句话比较有意思，匹配的是“soid his hot bod”

[root@localhost ~]# grep ".*" test_rule.txt
```

4)、 “^”匹配行首，“$”匹配行尾

“^”代表匹配行首，比如“^M”会匹配以大写“M”开头的行：

```
[root@localhost ~]# grep "^M" test_rule.txt
Mr. Li Ming said:
Mr. Shen Chao is the most honest man
```

“$”代表匹配行尾，如果“n$”会匹配以小写“n”结尾的行：

```
[root@localhost ~]# grep "n$" test_rule.txt
Mr. Shen Chao is the most honest man
```

而“^$”则会匹配空白行：

```
[root@localhost ~]# grep -n "^$" test_rule.txt
```

5）、 “[]” 匹配中括号中指定的任意一个字符，只匹配一个字符

“[]”会匹配中括号中指定任意一个字符，注意只能匹配一个字符。比如[ao]要不会匹配一个 a 字符，要不会匹配一个 o 字符：

```
[root@localhost ~]# grep "s[ao]id" test_rule.txt
```

而“[0-9]”会匹配任意一个数字，如：

```
[root@localhost ~]# grep "[0-9]" test_rule.txt
```

而“[A-Z]”则会匹配一个大写字母，如：

```
[root@localhost ~]# grep "[A-Z]" test_rule.txt
```

如果正则是“^[a-z]”代表匹配用小写字母开头的行：

```
[root@localhost ~]# grep "^[a-z]" test_rule.txt
```

6）“[^]” 匹配除中括号的字符以外的任意一个字符

```
[root@localhost ~]# grep "^[^a-z]" test_rule.txt
```

而“^[^a-zA-Z]”则会匹配不用字母开头的行：

```
[root@localhost ~]# grep "^[^a-zA-Z]" test_rule.txt
```

7）、 “\” 转义符

```
[root@localhost ~]# grep "\.$" test_rule.txt
```

8）、 “\{n\}”表示其前面的字符恰好出现 n 次

```
[root@localhost ~]# grep "a\{3\}" test_rule.txt
```

上面的两行都包含三个连续的 a，所以都会匹配。但是如果先要只显示三个连续的 a，可以这样来写正则：

```
[root@localhost ~]# grep "[su]a\{3\}[il]" test_rule.txt
```

如果正则是“[0-9]\{3\}”则会匹配包含连续的三个数字的字符串：

```
[root@localhost ~]# grep "[0-9]\{3\}" test_rule.txt
```

虽然“5555”有四个连续的数字，但是包含三个连续的数字，所以也是可以列出的。可是这样不能体现出来“[0-9]\{3\}”只能匹配三个连续的数字，而不能匹配四个连续的数字。那么正则就应该这样来写“^[0-9]\{3\}[a-z]”：

```
[root@localhost ~]# grep "^[0-9]\{3\}[a-z]" test_rule.txt
123despise him.
#只匹配用连续三个数字开头的行
```

9）、 “\{n,\}”表示其前面的字符出现不小于 n 次

“\{n,\}”会匹配前面的字符出现最少 n 次。比如“zo\{3,\}m”这个正则就会匹配用 z 开头，m 结尾，中间最少有三个 o 的字符串。那么“^[0-9]\{3,\}[a-z]”这个正则就能匹配最少用连续三个数字开头的字符串：

```
[root@localhost ~]# grep "^[0-9]\{3,\}[a-z]" test_rule.txt
```

而“[su]a\{3，\}[il]”正则则会匹配在字母 s 或 u 和 i 或 l 之间，最少出现三个连续的 a 的字符串：

```
[root@localhost ~]# grep "[su]a\{3,\}[il]" test_rule.txt
```

10）“\{n,m\}”匹配其前面的字符至少出现 n 次，最多出现 m 次

```
[root@localhost ~]# grep "sa\{1,3\}i" test_rule.txt
#匹配在字母 s 和字母 i 之间有最少一个 a，最多三个 a
[root@localhost ~]# grep "sa\{2,3\}i" test_rule.txt
#匹配在字母 s 和字母 i 之间有最少两个 a，最多三个 a
```

3 扩展正则表达式

熟悉正则表达式的童鞋应该很疑惑，在正则表达式中应该还可以支持一些元字符，比如“+”“？”“|”“（）”。其实 Linux 是支持这些元字符的，只是 grep 命令默认不支持而已。如果要想支持这些元字符，必须使用 egrep 命令或 grep -E 选项，所以我们又把这些元字符称作扩展元字符。

如果查询 grep 的帮助，对 egrep 的说明就是和 grep -E 选项一样的命令，所以我们可以把两个命令当做别名来对待。通过表 12-2 来看看 Shell 中支持的扩展元字符：

扩展元字符	作 用
+	前一个字符匹配 1 次或任意多次。 如“go+gle”会匹配“gogle”、“google”或“gooogle”，当然如果“o”有更多个，也能匹配。
?	前一个字符匹配 0 次或 1 次。

	如“colou?r”可以匹配“colour”或“color”。
|	匹配两个或多个分支选择。 如“was|his”会匹配既包含“was”的行，也匹配包含“his”的行。
（）	匹配其整体为一个字符，即模式单元。可以理解为由多个单个字符组成的大字符。 如“(dog)+”会匹配“dog”、“dogdog”、“dogdogdog”等，因为被（）包含的字符会当成一个整体。但“hello （world|earth）”会匹配“hello world”及“hello earth”。

二、 字符截取和替换命令

1 cut 列提取命令

```
[root@localhost ~]# cut [选项] 文件名
选项：
    -f 列号：       提取第几列
    -d 分隔符：     按照指定分隔符分割列
    -c 字符范围：   不依赖分隔符来区分列，而是通过字符范围（行首为 0）来进行字段
                    提取。“n-”表示从第 n 个字符到行尾；“n-m”从第 n 个字符到第 m
                    个字符；“-m”表示从第 1 个字符到第 m 个字符。
```

cut 命令的默认分隔符是制表符，也就是“tab”键，不过对空格符可是支持的不怎么好啊。我们先建立一个测试文件，然后看看 cut 命令的作用吧：

```
[root@localhost ~]# vi student.txt
ID      Name    gender  Mark
1       Liming  M       86
2       Sc      M       90
3       Tg      M       83
```

```
[root@localhost ~]# cut -f 2 student.txt
#提取第二列内容
```

那如果想要提取多列呢？只要列号直接用“，”分开，命令如下：

```
[root@localhost ~]# cut -f 2,3 student.txt
```

cut 可以按照字符进行提取，需要注意“8-”代表的是提取所有行的第十个字符开始到行尾，而“10-20”代表提取所有行的第十个字符到第二十个字符，而“-8”代表提取所有行从行首到第八个字符：

```
[root@localhost ~]# cut -c 8- student.txt
#提取第八个字符开始到行尾，好像很乱啊，那是因为每行的字符个数不相等啊
```

```
[root@localhost ~]# cut -d ":" -f 1,3 /etc/passwd
#以“：”作为分隔符，提取/etc/passwd 文件的第一列和第三列
```

如果我想用 cut 命令截取 df 命令的第一列和第三列，就会出现这样的情况：

```
[root@localhost ~]# df -h | cut -d " " -f 1,3
```

2　awk 编程

1）、　概述

2）、　printf 格式化输出

```
[root@localhost ~]# printf ‘输出类型输出格式’ 输出内容
输出类型：
    %ns:        输出字符串。n 是数字指代输出几个字符
    %ni:        输出整数。n 是数字指代输出几个数字
    %m.nf:      输出浮点数。m 和 n 是数字，指代输出的整数位数和小数位数。如%8.2f
                代表共输出 8 位数，其中 2 位是小数，6 位是整数。
输出格式：
    \a:         输出警告声音
    \b:         输出退格键，也就是 Backspace 键
    \f:         清除屏幕
    \n:         换行
    \r:         回车，也就是 Enter 键
    \t:         水平输出退格键，也就是 Tab 键
    \v:         垂直输出退格键，也就是 Tab 键
```

为了演示 printf 命令，我们需要修改下刚刚 cut 命令使用的 student.txt 文件，文件内容如下：

```
[root@localhost ~]# vi student.txt
ID      Name      PHP      Linux    MySQL    Average
1       Liming    82       95       86       87.66
2       Sc        74       96       87       85.66
3       Tg        99       83       93       91.66
```

我们使用 printf 命令输出下这个文件的内容：

```
[root@localhost ~]# printf '%s' $(cat student.txt)
IDNamegenderPHPLinuxMySQLAverage1LimingM82958687.662ScM74968785.663TgM99839391.66[root@
localhost ~]#
```

晕菜，全变乱了一锅粥。这就是 printf 命令，如果不指定输出格式，则会把所有输出内容连在一起输出。其实文本的输出本身就是这样的，cat 等文本输出命令之所以可以按照格式漂亮的输出，那是因为 cat 命令已经设定了输出格式。那么为了用 printf 输出合理的格式，应该这样做：

```
[root@localhost ~]# printf '%s\t %s\t %s\t %s\t %s\t %s\t \n' $(cat student.txt)
#注意在 printf 命令的单引号中，只能识别格式输出符号，而手工输入的空格是无效的
ID      Name    PHP     Linux   MySQL   Average
1       Liming  82      95      86      87.66
2       Sc      74      96      87      85.66
3       Tg      99      83      93      91.66
```

如果不想把成绩当成字符串输出，而是按照整型和浮点型输出，则要这样：

```
[root@localhost ~]# printf '%i\t %s\t %i\t %i\t %i\t %8.2f\t \n' \
$(cat student.txt | grep -v Name)
```

3)、 awk 基本使用

```
[root@localhost ~]# awk ‘条件 1{动作 1} 条件 2{动作 2}…’ 文件名
```

条件（Pattern）：

一般使用关系表达式作为条件。这些关系表达式非常多，具体参考表 12-3 所示，例如：

x > 10 判断变量 x 是否大于 10

x == y 判断变量 x 是否等于变量 y

A ~ B 判断字符串 A 中是否包含能匹配 B 表达式的子字符串

A !~ B 判断字符串 A 中是否不包含能匹配 B 表达式的子字符串

动作（Action）：

格式化输出

流程控制语句

我们这里先来学习 awk 基本用法，也就是只看看格式化输出动作是干什么的。至于条件类型和流程控制语句我们在后面再详细介绍。那看看这个例子吧：

```
[root@localhost ~]# awk '{printf $2 "\t" $6 "\n"}' student.txt
#输出第二列和第六列
```

比如刚刚截取 df 命令的结果时，cut 命令已经力不从心了，我们来看看 awk 命令：

```
[root@localhost ~]# df -h | awk '{print $1 "\t" $3}'
```

4）、 awk 的条件

条件的类型	条 件	说 明
awk 保留字	BEGIN	在 awk 程序一开始时，尚未读取任何数据之前执行。BEGIN 后的动作只在程序开始时执行一次
	END	在 awk 程序处理完所有数据，即将结束时执行。END 后的动作只在程序结束时执行一次
关系运算符	>	大于
	<	小于
	>=	大于等于
	<=	小于等于
	==	等于。用于判断两个值是否相等，如果是给变量赋值，请使用“=”号
	!=	不等于
	A~B	判断字符串 A 中是否包含能匹配 B 表达式的子字符串
	A!~B	判断字符串 A 中是否不包含能匹配 B 表达式的子字符串
正则表达式	/正则/	如果在“//”中可以写入字符，也可以支持正则表达式

✧ BEGIN

BEGIN 是 awk 的保留字，是一种特殊的条件类型。BEGIN 的执行时机是“在 awk 程序一开始时，尚未读取任何数据之前执行”。一旦 BEGIN 后的动作执行一次，当 awk 开始从文件中读入数据，BEGIN 的条件就不再成立，所以 BEGIN 定义的动作只能被执行一次。例如：

```
[root@localhost ~]# awk 'BEGIN{printf "This is a transcript \n" }
{printf $2 "\t" $6 "\n"}' student.txt
#awk 命令只要检测不到完整的单引号不会执行，所以这个命令的换行不用加入“\”，就是一行命令
#这里定义了两个动作
#第一个动作使用 BEGIN 条件，所以会在读入文件数据前打印“这是一张成绩单”（只会执行一次）
```

```
#第二个动作会打印文件的第二字段和第六字段
```

✧ END

END 也是 awk 保留字，不过刚好和 BEGIN 相反。END 是在 awk 程序处理完所有数据，即将结束时执行。END 后的动作只在程序结束时执行一次。例如：

```
[root@localhost ~]# awk 'END{printf "The End \n" }
{printf $2 "\t" $6 "\n"}' student.txt
#在输出结尾输入“The End”，这并不是文档本身的内容，而且只会执行一次
```

✧ 关系运算符

举几个例子看看关系运算符。假设我想看看平均成绩大于等于 87 分的学员是谁，就可以这样输入命令：

```
例子 1：
[root@localhost ~]# cat student.txt | grep -v Name |  \
 awk '$6 >= 87 {printf $2 "\n" }'
#使用 cat 输出文件内容，用 grep 取反包含“Name”的行
#判断第六字段（平均成绩）大于等于 87 分的行，如果判断式成立，则打第六列（学员名）
```

加入了条件之后，只有条件成立动作才会执行，如果条件不满足，则动作则不运行。通过这个实验，大家可以发现，虽然 awk 是列提取命令，但是也要按行来读入的。这个命令的执行过程是这样的：

1） 如果有 BEGIN 条件，则先执行 BEGIN 定义的动作

2） 如果没有 BEGIN 条件，则读入第一行，把第一行的数据依次赋予$0、$1、$2 等变量。其中$0 代表此行的整体数据，$1 代表第一字段，$2 代表第二字段。

2） 依据条件类型判断动作是否执行。如果条件符合，则执行动作，否则读入下一行数据。如果没有条件，则每行都执行动作。

3） 读入下一行数据，重复执行以上步骤。

再举个例子，如果我想看看 Sc 用户的平均成绩呢：

```
例子 2：
[root@localhost ~]#  awk '$2 ~ /Sc/ {printf $6 "\n"}' student.txt
#如果第二字段中输入包含有“Sc”字符，则打印第六字段数据
85.66
```

这里要注意在 awk 中，使用“//”包含的字符串，awk 命令才会查找。也就是说字符串必须用“//”包含，awk 命令才能正确识别。

✧ 正则表达式

如果要想让 awk 识别字符串，必须使用“//”包含，例如：

```
例子 1：
[root@localhost ~]# awk '/Liming/ {print}' student.txt
#打印 Liming 的成绩
```

当使用 df 命令查看分区使用情况是，如果我只想查看真正的系统分区的使用状况，而不想查看光盘和临时分区的使用状况，则可以：

```
例子 2：
[root@localhost ~]# df -h | awk '/sda[0-9]/ {printf $1 "\t" $5 "\n"} '
#查询包含有 sda 数字的行，并打印第一字段和第五字段
```

5）、 awk 内置变量

awk 内置变量	作 用
$0	代表目前 awk 所读入的整行数据。我们已知 awk 是一行一行读入数据的，$0 就代表当前读入行的整行数据。
$n	代表目前读入行的第 n 个字段。
NF	当前行拥有的字段（列）总数。
NR	当前 awk 所处理的行，是总数据的第几行。
FS	用户定义分隔符。awk 的默认分隔符是任何空格，如果想要使用其他分隔符（如“：”），就需要 FS 变量定义。
ARGC	命令行参数个数。
ARGV	命令行参数数组。
FNR	当前文件中的当前记录数（对输入文件起始为 1）。
OFMT	数值的输出格式（默认为%.6g）。
OFS	输出字段的分隔符（默认为空格）。
ORS	输出记录分隔符（默认为换行符）。
RS	输入记录分隔符（默认为换行符）。

```
[root@localhost ~]# cat /etc/passwd | grep "/bin/bash" | \
 awk '{FS=":"} {printf $1 "\t" $3 "\n"}'
#查询可以登录的用户的用户名和 UID
```

这里“：”分隔符生效了，可是第一行却没有起作用，原来我们忘记了“BEGIN”条件，那么再来试试：

```
[root@localhost ~]# cat /etc/passwd | grep "/bin/bash" | \
awk 'BEGIN {FS=":"} {printf $1 "\t" $3 "\n"}'
```

```
[root@localhost ~]# cat /etc/passwd | grep "/bin/bash" | \
awk 'BEGIN {FS=":"} {printf $1 "\t" $3 "\t 行号：" NR "\t 字段数：" NF "\n"}'
#解释下 awk 命令
#开始执行{分隔符是“：”} {输出第一字段和第三字段 输出行号（NR 值） 字段数（NF 值）}
root    0       行号：1         字段数：7
user1   501     行号：2         字段数：7
```

如果我只想看看 sshd 这个伪用户的相关信息，则可以这样使用：

```
[root@localhost ~]# cat /etc/passwd | \
awk 'BEGIN {FS=":"} $1=="sshd" {printf $1 "\t" $3 "\t 行号："NR "\t 字 段数："F "\n"}'
#可以看到 sshd 伪用户的 UID 是 74，是/etc/passwd 文件的第 28 行，此行有 7 个字段
```

6）、 awk 流程控制

我们再来利用下 student.txt 文件做个练习，后面的使用比较复杂，我们再看看这个文件的内容：

```
[root@localhost ~]# cat student.txt
ID      Name    PHP     Linux   MySQL   Average
1       Liming  82      95      86      87.66
```

```
2       Sc      74      96      87      85.66
3       Tg      99      83      93      91.66
```

我们先来看看该如何在 awk 中定义变量与调用变量的值。假设我想统计 PHP 成绩的总分，那么就应该这样：

```
[root@localhost ~]# awk 'NR==2{php1=$3}
NR==3{php2=$3}
NR==4{php3=$3;totle=php1+php2+php3;print "totle php is " totle}' student.txt
#统计 PHP 成绩的总分
```

我们解释下这个命令。“NR==2{php1=$3}”（条件是 NR==2，动作是 php1=$3）这句话是指如果输入数据是第二行（第一行是标题行），就把第二行的第三字段的值赋予变量“php1”。“NR==3{php2=$3}”这句话是指如果输入数据是第三行，就把第三行的第三字段的值赋予变量“php2”。“NR==4{php3=$3;totle=php1+php2+php3;print "totle php is " totle}”（“NR==4”是条件，后面{}中的都是动作)这句话是指如果输入数据是第四行，就把第四行的第三字段的值赋予变量“php3”；然后定义变量 totle 的值是“php1+php2+php3”；然后输出“totle php is”关键字，后面加变量 totle 的值。

在 awk 编程中，因为命令语句非常长，在输入格式时需要注意以下内容：

- ✧ 多个条件{动作}可以用空格分割，也可以用回车分割。
- ✧ 在一个动作中，如果需要执行多个命令，需要用“；”分割，或用回车分割。
- ✧ 在 awk 中，变量的赋值与调用都不需要加入“$”符。
- ✧ 条件中判断两个值是否相同，请使用“==”，以便和变量赋值进行区分。

在看看该如何实现流程控制，假设如果 Linux 成绩大于 90，就是一个好男人（学 PHP 的表示压力很大！）：

```
[root@localhost ~]# awk '{if (NR>=2)
{if ($4>90) printf $2 " is a good man!\n"}}' student.txt
#程序中有两个 if 判断，第一个判断行号大于 2，第二个判断 Linux 成绩大于 90 分
Liming is a good man!
Sc is a good man!
```

其实在 awk 中 if 判断语句，完全可以直接利用 awk 自带的条件来取代，刚刚的脚本可以改写成这样：

```
[root@localhost ~]# awk ' NR>=2 {test=$4}
test>90 {printf $2 " is a good man!\n"}' student.txt
#先判断行号如果大于 2，就把第四字段赋予变量 test
#在判断如果 test 的值大于 90 分，就打印好男人
Liming is a good man!
Sc is a good man!
```

7)、 awk 函数

awk 编程也允许在编程时使用函数，在本小节我们讲讲 awk 的自定义函数。awk 函数的定义方法如下：

```
function 函数名（参数列表）{
函数体
}
```

我们定义一个简单的函数，使用函数来打印 student.txt 的学员姓名和平均成绩，应该这样来写

函数：

```
[root@localhost ~]# awk 'function test(a,b) { printf a "\t" b "\n" }
#定义函数 test，包含两个参数，函数体的内容是输出这两个参数的值
{ test($2,$6) } ' student.txt
#调用函数 test，并向两个参数传递值。
Name    Average
Liming  87.66
Sc      85.66
Tg      91.66
```

8）、 awk 中调用脚本

对于小的单行程序来说，将脚本作为命令行自变量传递给 awk 是非常简单的，而对于多行程序就比较难处理。当程序是多行的时候，使用外部脚本是很适合的。首先在外部文件中写好脚本，然后可以使用 awk 的-f 选项，使其读入脚本并且执行。

例如，我们可以先编写一个 awk 脚本：

```
[root@localhost ~]# vi pass.awk
BEGIN   {FS=":"}
{ print  $1  "\t"  $3}
```

然后可以使用“-f”选项来调用这个脚本：

```
[root@localhost ~]# awk -f pass.awk /etc/passwd
root    0
bin     1
daemon  2
…省略部分输出…
```

3 sed 命令

sed 主要是用来将数据进行选取、替换、删除、新增的命令，我们看看命令的语法：

```
[root@localhost ~]# sed [选项] '[动作]' 文件名
选项：
    -n:             一般 sed 命令会把所有数据都输出到屏幕，如果加入此选择，则只会
                    把经过 sed 命令处理的行输出到屏幕。
    -e:             允许对输入数据应用多条 sed 命令编辑。
    -f 脚本文件名:  从 sed 脚本中读入 sed 操作。和 awk 命令的-f 非常类似。
    -r:             在 sed 中支持扩展正则表达式。
    -i:             用 sed 的修改结果直接修改读取数据的文件，而不是由屏幕输出
动作：
    a \:            追加，在当前行后添加一行或多行。添加多行时，除最后 一行外，
                    每行末尾需要用“\”代表数据未完结。
    c \:            行替换，用 c 后面的字符串替换原数据行，替换多行时，除最后一行
                    外，每行末尾需用“\”代表数据未完结。
    i \:            插入，在当期行前插入一行或多行。插入多行时，除最后 一行外，
                    每行末尾需要用“\”代表数据未完结。
    d:              删除，删除指定的行。
```

p：	打印，输出指定的行。
s：	字串替换，用一个字符串替换另外一个字符串。格式为“行范围 s/旧字串/新字串/g”（和 vim 中的替换格式类似）。

对 sed 命令大家要注意，sed 所做的修改并不会直接改变文件的内容（如果是用管道符接收的命令的输出，这种情况连文件都没有），而是把修改结果只显示到屏幕上，除非使用“-i”选项才会直接修改文件。

✧ 行数据操作

闲话少叙，直奔主题，我们举几个例子来看看 sed 命令到底是干嘛的。假设我想查看下 student.txt 的第二行，那么就可以利用“p”动作了：

```
[root@localhost ~]# sed '2p' student.txt
ID      Name    PHP     Linux   MySQL   Average
1       Liming  82      95      86      87.66
1       Liming  82      95      86      87.66
2       Sc      74      96      87      85.66
3       Tg      99      83      93      91.66
```

好像看着不怎么顺眼啊！“p”命令确实输出了第二行数据，但是 sed 命令还会把所有数据都输出一次，这时就会看到这个比较奇怪的结果。那如果我想指定输出某行数据，就需要“-n”选项的帮助了：

```
[root@localhost ~]# sed -n '2p' student.txt
1       Liming  82      95      86      87.66
```

再来看看如何删除文件的数据：

```
[root@localhost ~]# sed '2,4d' student.txt
#删除第二行到第四行的数据
ID      Name    PHP     Linux   MySQL   Average

[root@localhost ~]# cat student.txt
#但是文件本身并没有修改
ID      Name    PHP     Linux   MySQL   Average
1       Liming  82      95      86      87.66
2       Sc      74      96      87      85.66
3       Tg      99      83      93      91.66
```

再来看看如何追加和插入行数据：

```
[root@localhost ~]# sed '2a hello' student.txt
#在第二行后加入 hello
```

“a”会在指定行后面追加入数据，如果想要在指定行前面插入数据，则需要使用“i”动作：

```
[root@localhost ~]# sed '2i hello \
> world' student.txt
#在第二行前插入两行数据
```

如果是想追加或插入多行数据，除最后一行外，每行的末尾都要加入“\”代表数据未完结。再来看看“-n”选项的作用：

```
[root@localhost ~]# sed -n '2i hello \
#只查看 sed 命令操作的数据
```

```
> world' student.txt
```

“-n”只查看 sed 命令操作的数据，而不是查看所有数据。

再来看看如何实现行数据替换，假设李明老师的成绩太好了，我实在是不想看到他的成绩刺激我，那我就可以这样：

```
[root@localhost ~]# cat student.txt | sed '2c No such person'
```

sed 命令默认情况是不会修改文件内容的，如果我确定需要让 sed 命令直接处理文件的内容，可以使用“-i”选项。不过要小心啊，这样非常容易误操作，在操作系统文件时请小心谨慎。可以使用这样的命令：

```
[root@localhost ~]# sed -i '2c No such person' student.txt
```

✧ 字符串替换

“c”动作是进行整行替换的，如果仅仅想替换行中的部分数据，就要使用“s”动作了。s 动作的格式是：

```
[root@localhost ~]# sed ‘s/旧字串/新字串/g’ 文件名
```

替换的格式和 vim 非常类似，假设我觉得我自己的 PHP 成绩太低了，想作弊给他改高点，就可以这样来做：

```
[root@localhost ~]# sed '3s/74/99/g' student.txt
#在第三行中，把 74 换成 99
```

这样看起来就比较爽了吧。如果我想把 Tg 老师的成绩注释掉，让他不再生效（没有成绩了吧？补考去吧？）。可以这样做：

```
[root@localhost ~]# sed '4s/^/#/g' student.txt
#这里使用正则表达式，“^”代表行首
```

在 sed 中只能指定行范围，所以很遗憾我在他们两个的中间，不能只把他们两个注释掉，那么我们可以这样：

```
[root@localhost ~]# sed -e 's/Liming//g ; s/Tg//g' student.txt
#同时把“Liming”和“Tg”替换为空
```

“-e”选项可以同时执行多个 sed 动作，当然如果只是执行一个动作也可以使用“-e”选项，但是这时没有什么意义。还要注意，多个动作之间要用“；”号或回车分割，例如上一个命令也可以这样写：

```
[root@localhost ~]# sed -e 's/Liming//g
> s/Tg//g' student.txt
```

三 字符处理命令

1 排序命令 sort

```
[root@localhost ~]# sort [选项] 文件名
选项：
    -f:         忽略大小写
    -b:         忽略每行前面的空白部分
    -n:         以数值型进行排序，默认使用字符串型排序
    -r:         反向排序
    -u:         删除重复行。就是 uniq 命令
    -t:         指定分隔符，默认是分隔符是制表符
```

```
    -k n[,m]:    按照指定的字段范围排序。从第 n 字段开始，m 字段结束（默认到行尾）
```

sort 命令默认是用每行开头第一个字符来进行排序的，比如：

```
[root@localhost ~]# sort /etc/passwd
#排序用户信息文件
```

如果想要反向排序，请使用“-r”选项：

```
[root@localhost ~]# sort -r /etc/passwd
#反向排序
```

如果想要指定排序的字段，需要使用“-t”选项指定分隔符，并使用“-k”选项指定字段号。加入我想要按照 UID 字段排序/etc/passwd 文件：

```
[root@localhost ~]# sort -t ":" -k 3,3 /etc/passwd
#指定分隔符是“：”，用第三字段开头，第三字段结尾排序，就是只用第三字段排序
```

看起来好像很美，可是如果仔细看看，怎么 daemon 用户的 UID 是 2，反而排在了下面？这是因为 sort 默认是按照字符排序，前面用户的 UID 的第一个字符都是 1，所以这么排序。要想按照数字排序，请使用“-n”选项：

```
[root@localhost ~]# sort -n -t ":" -k 3,3 /etc/passwd
```

当然“-k”选项可以直接使用“-k 3”，代表从第三字段到行尾都排序（第一个字符先排序，如果一致，第二个字符再排序，知道行尾）。

2　uniq

uniq 命令是用来取消重复行的命令，其实和“sort -u”选项是一样的。命令格式如下：

```
[root@localhost ~]# uniq [选项] 文件名
选项：
    -i:     忽略大小写
```

3　统计命令 wc

```
[root@localhost ~]# wc [选项] 文件名
选项：
    -l:     只统计行数
    -w:     只统计单词数
    -m:     只统计字符数
```

四 条件判断

1　按照文件类型进行判断

根据表 12-5，我们先来看看 test 可以进行哪些文件类型的判断：

测试选项	作　　用
-b 文件	判断该文件是否存在，并且是否为块设备文件（是块设备文件为真）
-c 文件	判断该文件是否存在，并且是否为字符设备文件（是字符设备文件为真）
-d 文件	判断该文件是否存在，并且是否为目录文件（是目录为真）
-e 文件	判断该文件是否存在（存在为真）
-f 文件	判断该文件是否存在，并且是否为普通文件（是普通文件为真）
-L 文件	判断该文件是否存在，并且是否为符号链接文件（是符号链接文件为真）
-p 文件	判断该文件是否存在，并且是否为管道文件（是管道文件为真）

-s 文件	判断该文件是否存在，并且是否为非空（非空为真）
-S 文件	判断该文件是否存在，并且是否为套接字文件（是套接字文件为真）

```
[root@localhost ~]# [ -e /root/sh/ ]
[root@localhost ~]# echo $?
0
#判断结果为 0，/root/sh/目录是存在的

[root@localhost ~]# [ -e /root/test ]
[root@localhost ~]# echo $?
1
#在/root/下并没有 test 文件或目录，所以“$?”的返回值为非零
```

还记得多命令顺序执行的“&&”和“||”吗？我们可以再判断一下/root/sh/是否是目录：

```
[root@localhost ~]# [ -d /root/sh ] && echo "yes" || echo "no"
#第一个判断命令如果正确执行，则打印“yes”，否则打印“no”
```

2 按照文件权限进行判断

test 是非常完善的判断命令，还可以判断文件的权限，我们通过表 12-6 来看看：

测试选项	作 用
-r 文件	判断该文件是否存在，并且是否该文件拥有读权限（有读权限为真）
-w 文件	判断该文件是否存在，并且是否该文件拥有写权限（有写权限为真）
-x 文件	判断该文件是否存在，并且是否该文件拥有执行权限（有执行权限为真）
-u 文件	判断该文件是否存在，并且是否该文件拥有 SUID 权限（有 SUID 权限为真）
-g 文件	判断该文件是否存在，并且是否该文件拥有 SGID 权限（有 SGID 权限为真）
-k 文件	判断该文件是否存在，并且是否该文件拥有 SBit 权限（有 SBit 权限为真）

比如：

```
[root@localhost ~]# ll student.txt
-rw-r--r--. 1 root root 97 6 月   7 07:34 student.txt
[root@localhost ~]# [ -w student.txt ] && echo "yes" || echo "no"
yes
#判断文件是拥有写权限的
```

3 两个文件之间进行比较

通过表 12-7 来看看如何进行两个文件之间的比较：

测试选项	作 用
文件 1 -nt 文件 2	判断文件 1 的修改时间是否比文件 2 的新（如果新则为真）
文件 1 -ot 文件 2	判断文件 1 的修改时间是否比文件 2 的旧（如果旧则为真）
文件 1 -ef 文件 2	判断文件 1 是否和文件 2 的 Inode 号一致，可以理解为两个文件是否为同一个文件。这个判断用于判断硬链接是很好的方法

我们一直很苦恼，到底该如何判断两个文件是否是硬链接呢？这时 test 就派上用场了：

```
[root@localhost ~]# ln /root/student.txt /tmp/stu.txt
#创建个硬链接吧
[root@localhost ~]# [ /root/student.txt -ef /tmp/stu.txt ] && echo "yes" || echo "no"
```

```
yes
#用 test 测试下，果然很有用
```

4　两个整数之间比较

通过表 12-8 来学习下如何在两个整数之间进行比较：

测试选项	作　　用
整数 1 -eq 整数 2	判断整数 1 是否和整数 2 相等（相等为真）
整数 1 -ne 整数 2	判断整数 1 是否和整数 2 不相等（不相等位置）
整数 1 -gt 整数 2	判断整数 1 是否大于整数 2（大于为真）
整数 1 -lt 整数 2	判断整数 1 是否小于整数 2（小于位置）
整数 1 -ge 整数 2	判断整数 1 是否大于等于整数 2（大于等于为真）
整数 1 -le 整数 2	判断整数 1 是否小于等于整数 2（小于等于为真）

举个例子：

```
[root@localhost ~]# [ 23 -ge 22 ] && echo "yes" || echo "no"
yes
#判断 23 是否大于等于 22，当然是了
[root@localhost ~]# [ 23 -le 22 ] && echo "yes" || echo "no"
no
#判断 23 是否小于等于 22，当然不是了
```

5　字符串的判断

通过表 12-9，我们来学习下字符串的判断：

测试选项	作　　用
-z 字符串	判断字符串是否为空（为空返回真）
-n 字符串	判断字符串是否为非空（非空返回真）
字串 1 ==字串 2	判断字符串 1 是否和字符串 2 相等（相等返回真）
字串 1 != 字串 2	判断字符串 1 是否和字符串 2 不相等（不相等返回真）

举个例子：

```
[root@localhost ~]# name=sc
#给 name 变量赋值
[root@localhost ~]# [ -z "$name" ] && echo "yes" || echo "no"
no
#判断 name 变量是否为空，因为不为空，所以返回 no
```

再来看看如何判断两个字符串相等：

```
[root@localhost ~]# aa=11
[root@localhost ~]# bb=22
#给变量 aa 和变量 bb 赋值
[root@localhost ~]# [ "$aa" == "bb" ] && echo "yes" || echo "no"
no
#判断两个变量的值是否相等，明显不相等，所以返回 no
```

6　多重条件判断

通过表 12-10，来看看多重条件判断是什么样子的：

测试选项	作　　用
判断 1 -a 判断 2	逻辑与，判断 1 和判断 2 都成立，最终的结果才为真
判断 1 -o 判断 2	逻辑或，判断 1 和判断 2 有一个成立，最终的结果就为真
！判断	逻辑非，使原始的判断式取反

举个例子：

```
[root@localhost ~]# aa=11
#给变量 aa 赋值
[root@localhost ~]# [ -n "$aa" -a "$aa" -gt 23 ] && echo "yes" || echo "no"
no
#判断变量 aa 是否有值，同时判断变量 aa 的是否大于 23
#因为变量 aa 的值不大于 23，所以虽然第一个判断值为真，返回的结果也是假
```

要想让刚刚的判断式返回真，需要给变量 aa 重新赋个大于 23 的值：

```
[root@localhost ~]# aa=24
[root@localhost ~]# [ -n "$aa" -a "$aa" -gt 23 ] && echo "yes" || echo "no"
yes
```

再来看看逻辑非是什么样子的：

```
[root@localhost ~]# [ ! -n "$aa" ] && echo "yes" || echo "no"
no
#本来“-n”选项是变量 aa 不为空，返回值就是真。
#加入！之后，判断值就会取反，所以当变量 aa 有值时，返回值是假
```

注意：“！”和“-n”之间必须加入空格，否则会报错的。

五 流程控制

1　if 条件判断

1)、　单分支 if 条件语句

单分支条件语句最为简单，就是只有一个判断条件，如果符合条件则执行某个程序，否则什么事情都不做。语法如下：

```
if [ 条件判断式 ];then
    程序
fi
```

单分支条件语句需要注意几个点：

- ✧ if 语句使用 fi 结尾，和一般语言使用大括号结尾不同
- ✧ [条件判断式]就是使用 test 命令判断，所以中括号和条件判断式之间必须有空格
- ✧ then 后面跟符合条件之后执行的程序，可以放在[]之后，用“；”分割。也可以换行写入，就不需要“；”了，比如单分支 if 语句还可以这样写：

```
if [ 条件判断式 ]
    then
        程序
fi
```

```
[root@localhost ~]# vi sh/if1.sh
#!/bin/bash
```

```
#统计根分区使用率
# Author: shenchao （E-mail: shenchao@atguigu.com）

rate=$(df -h | grep "/dev/sda3" | awk '{print $5}' | cut -d "%" -f1)
#把根分区使用率作为变量值赋予变量 rate
if [ $rate -ge 80 ]
#判断 rate 的值如果大于等于 80，则执行 then 程序
    then
        echo "Warning! /dev/sda3 is full!!"
        #打印警告信息。在实际工作中，也可以向管理员发送邮件。
fi
```

2）、 双分支 if 条件语句

```
if [ 条件判断式 ]
    then
        条件成立时，执行的程序
    else
        条件不成立时，执行的另一个程序
fi
```

例子 1：

我们写一个数据备份的例子，来看看双分支 if 条件语句。

```
例子 1：备份 mysql 数据库
[root@localhost ~]# vi sh/bakmysql.sh
#!/bin/bash
#备份 mysql 数据库。
# Author: shenchao （E-mail: shenchao@atguigu.com）

ntpdate asia.pool.ntp.org &>/dev/null
#同步系统时间
date=$(date +%y%m%d)
#把当前系统时间按照“年月日”格式赋予变量 date
size=$(du -sh /var/lib/mysql)
#统计 mysql 数据库的大小，并把大小赋予 size 变量

if [ -d /tmp/dbbak ]
#判断备份目录是否存在，是否为目录
    then
        #如果判断为真，执行以下脚本
        echo "Date : $date!" > /tmp/dbbak/dbinfo.txt
        #把当前日期写入临时文件
        echo "Data size : $size" >> /tmp/dbbak/dbinfo.txt
        #把数据库大小写入临时文件
        cd /tmp/dbbak
```

```
        #进入备份目录
        tar -zcf mysql-lib-$date.tar.gz /var/lib/mysql dbinfo.txt &>/dev/null
        #打包压缩数据库与临时文件，把所有输出丢入垃圾箱（不想看到任何输出）
        rm -rf /tmp/dbbak/dbinfo.txt
        #删除临时文件
    else
        mkdir /tmp/dbbak
        #如果判断为假，则建立备份目录
        echo "Date : $date!" > /tmp/dbbak/dbinfo.txt
        echo "Data size : $size" >> /tmp/dbbak/dbinfo.txt
        #把日期和数据库大小保存如临时文件
        cd /tmp/dbbak
        tar -zcf mysql-lib-$date.tar.gz dbinfo.txt /var/lib/mysql &>/dev/null
        #压缩备份数据库与临时文件
        rm -rf /tmp/dbbak/dbinfo.txt
        #删除临时文件
fi
```

例子 2：

再举个例子，在工作当中，服务器上的服务经常会宕机。如果我们对服务器监控不好，就会造成服务器中服务宕机了，而管理员却不知道的情况，这时我们可以写一个脚本来监听本机的服务，如果服务停止或宕机了，可以自动重启这些服务。我们拿 apache 服务来举例：

```
例子 2：判断 apache 是否启动，如果没有启动则自动启动
[root@localhost ~]# vi sh/autostart.sh
#!/bin/bash
#判断 apache 是否启动，如果没有启动则自动启动
# Author: shenchao （E-mail: shenchao@atguigu.com）

port=$(nmap -sT 192.168.4.210 | grep tcp | grep http | awk '{print $2}')
#使用 nmap 命令扫描服务器，并截取 apache 服务的状态，赋予变量 port
if [ "$port" == "open" ]
#如果变量 port 的值是“open”
    then
        echo “$(date) httpd is ok!” >> /tmp/autostart-acc.log
        #则证明 apache 正常启动，在正常日志中写入一句话即可
    else
        /etc/rc.d/init.d/httpd start &>/dev/null
        #否则证明 apache 没有启动，自动启动 apache
        echo "$(date) restart httpd !!" >> /tmp/autostart-err.log
        #并在错误日志中记录自动启动 apche 的时间
fi
```

以我们使用 nmap 端口扫描命令，nmap 命令格式如下：

```
[root@localhost ~]# nmap -sT 域名或 IP
选项：
```

```
    -s      扫描
    -T      扫描所有开启的 TCP 端口
```

这条命令的执行结果如下：

```
[root@localhost ~]# nmap -sT 192.168.4.210
#可以看到这台服务器开启了如下的服务
Starting Nmap 5.51 ( http://nmap.org ) at 2018-11-25 15:11 CST
Nmap scan report for 192.168.4.210
Host is up (0.0010s latency).
Not shown: 994 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
80/tcp   open  http                    ←apache 的状态是 open
111/tcp  open  rpcbind
139/tcp  open  netbios-ssn
445/tcp  open  microsoft-ds
3306/tcp open  mysql

Nmap done: 1 IP address (1 host up) scanned in 0.49 seconds
```

知道了 nmap 命令的用法，我们在脚本中使用的命令就是为了截取 http 的状态，只要状态是“open”就证明 apache 启动正常，否则证明 apache 启动错误。来看看脚本中命令的结果：

```
[root@localhost ~]# nmap -sT 192.168.4.210 | grep tcp | grep http | awk '{print $2}'
#扫描指定计算机，提取包含 tcp 的行，在提取包含 httpd 的行，截取第二列
open
#把截取的值赋予变量 port
```

3）、 多分支 if 条件语句

```
if [ 条件判断式 1 ]
    then
        当条件判断式 1 成立时，执行程序 1
elif [ 条件判断式 2 ]
    then
        当条件判断式 2 成立时，执行程序 2
…省略更多条件…
else
    当所有条件都不成立时，最后执行此程序
fi
```

那我们再写一个例子，用 if 多分支条件语句来判断一下用户输入的是一个文件，还是一个目录：

```
例子：判断用户输入的是什么文件
[root@localhost ~]# vi sh/if-elif.sh
#!/bin/bash
#判断用户输入的是什么文件
# Author: shenchao （E-mail: shenchao@atguigu.com）
```

```
read -p "Please input a filename: " file
#接收键盘的输入，并赋予变量 file

if [ -z "$file" ]
#判断 file 变量是否为空
    then
        echo "Error,please input a filename"
        #如果为空，执行程序 1，也就是输出报错信息
        exit 1
        #退出程序，并返回值为 1（把返回值赋予变量$?）
elif [ ! -e "$file" ]
#判断 file 的值是否存在
    then
        echo "Your input is not a file!"
        #如果不存在，则执行程序 2
        exit 2
        #退出程序，把并定义返回值为 2
elif [ -f "$file" ]
#判断 file 的值是否为普通文件
    then
        echo "$file is a regulare file!"
        #如果是普通文件，则执行程序 3
elif [ -d "$file" ]
#判断 file 的值是否为目录文件
    then
        echo "$file is a directory!"
        #如果是目录文件，则执行程序 4
else
        echo "$file is an other file!"
        #如果以上判断都不是，则执行程序 5
fi
```

2　多分支 case 条件语句

case 语句和 if…elif…else 语句一样都是多分支条件语句，不过和 if 多分支条件语句不同的是，case 语句只能判断一种条件关系，而 if 语句可以判断多种条件关系。case 语句语法如下：

```
case $变量名 in
    "值 1"）
        如果变量的值等于值 1，则执行程序 1
        ;;
    "值 2"）
        如果变量的值等于值 2，则执行程序 2
        ;;
    …省略其他分支…
```

```
    *)
        如果变量的值都不是以上的值，则执行此程序
        ;;
esac
```

这个语句需要注意以下内容：

- case 语句，会取出变量中的值，然后与语句体中的值逐一比较。如果数值符合，则执行对应的程序，如果数值不符，则依次比较下一个值。如果所有的值都不符合，则执行“*）”（“*”代表所有其他值）中的程序。
- case 语句以“case”开头，以“esac”结尾。

每一个分支程序之后要通过“;;”双分号结尾，代表该程序段结束（千万不要忘记，超哥每次写 case 语句，都会忘记双分号，有点“囧”）。

我们写一个判断是“yes/no”的例子：

```
[root@localhost ~]# vi sh/case.sh
#!/bin/bash
#判断用户输入
# Author: shenchao （E-mail: shenchao@atguigu.com）

read -p "Please choose yes/no: " -t 30 cho
#在屏幕上输出“请选择 yes/no”，然后把用户选择赋予变量 cho
case $cho in
#判断变量 cho 的值
        "yes")
        #如果是 yes
                echo "Your choose is yes!"
                #执行程序 1
                ;;
        "no")
        #如果是 no
                echo "Your choose is no!"
                #执行程序 2
                ;;
        *)
        #如果既不是 yes，也不是 no
                echo "Your choose is error!"
                #则执行此程序
                ;;
esac
```

3　for 循环

for 循环是固定循环，也就是在循环时已经知道需要进行几次的循环，有时也把 for 循环称为计数循环。for 的语法有如下两种：

```
语法一：
```

```
for 变量 in 值1 值2 值3…
    do
        程序
    done
```

这种语法中 for 循环的次数，取决于 in 后面值的个数（空格分隔），有几个值就循环几次，并且每次循环都把值赋予变量。也就是说，假设 in 后面有三个值，for 会循环三次，第一次循环会把值 1 赋予变量，第二次循环会把值 2 赋予变量，以此类推。

```
语法二：
for (( 初始值;循环控制条件;变量变化 ))
    do
        程序
    done
```

语法二中需要注意：

- ✧ 初始值：在循环开始时，需要给某个变量赋予初始值，如 i=1；
- ✧ 循环控制条件：用于指定变量循环的次数，如 i<=100，则只要 i 的值小于等于 100，循环就会继续；
- ✧ 变量变化：每次循环之后，变量该如何变化，如 i=i+1。代表每次循环之后，变量 i 的值都加 1。

1)、 语法一举例：

我们先看看语法一是什么样子的：

例子 1：打印时间

```
[root@localhost ~]# vi sh/for.sh
#!/bin/bash
#打印时间
# Author: shenchao (E-mail: shenchao@atguigu.com)

for time in morning noon afternoon evening
        do
                echo "This time is $time!"
        done
```

批量解压缩脚本就应该这样写：

例子 2：批量解压缩

```
[root@localhost ~]# vi sh/auto-tar.sh
#!/bin/bash
#批量解压缩脚本
# Author: shenchao (E-mail: shenchao@atguigu.com)

cd /lamp
#进入压缩包目录
ls *.tar.gz > ls.log
#把所有.tar.gz 结尾的文件的文件覆盖到 ls.log 临时文件中
for i in $(cat ls.log)
```

```
#读取 ls.log 文件的内容，文件中有多少个值，就会循环多少次，每次循环把文件名赋予变量 i
        do
                tar -zxf $i &>/dev/null
                #加压缩，并把所有输出都丢弃
        done
rm -rf /lamp/ls.log
#删除临时文件 ls.log
```

2)、 语法二举例

那语法二就和其他语言中的 for 循环更加类似了，也就是事先决定循环次数的固定循环了。先举个简单的例子：

```
例子 1：从 1 加到 100
#!/bin/bash
#从 1 加到 100
# Author: shenchao （E-mail: shenchao@atguigu.com）

s=0
for (( i=1;i<=100;i=i+1 ))
#定义循环 100 次
        do
                s=$(( $s+$i ))
                每次循环给变量 s 赋值
        done
echo "The sum of 1+2+...+100 is : $s"
#输出 1 加到 100 的和
```

```
例子 2：批量添加指定数量的用户
[root@localhost ~]# vi useradd.sh
#!/bin/bash
#批量添加指定数量的用户
# Author: shenchao （E-mail: shenchao@atguigu.com）

read -p "Please input user name: " -t 30 name
#让用户输入用户名，把输入保存入变量 name
read -p "Please input the number of users: " -t 30 num
#让用户输入添加用户的数量，把输入保存入变量 num
read -p "Please input the password of users: " -t 30 pass
#让用户输入初始密码，把输入保存如变量 pass

if [ ! -z "$name" -a ! -z "$num" -a ! -z "$pass" ]
#判断三个变量不为空
        then
        y=$(echo $num | sed 's/[0-9]//g')
```

```
        #定义变量的值为后续命令的结果
        #后续命令作用是，把变量 num 的值替换为空。如果能替换为空，证明 num 的值为数字
        #如果不能替换为空，证明 num 的值为非数字。我们使用这种方法判断变量 num 的值为数字
        if [ -z "$y" ]
        #如果变量 y 的值为空，证明 num 变量是数字
                then
                for (( i=1;i<=$num;i=i+1 ))
                #循环 num 变量指定的次数
                    do
                        /usr/sbin/useradd $name$i &>/dev/null
                        #添加用户，用户名为变量 name 的值加变量 i 的数字
                        echo $pass | /usr/bin/passwd --stdin $name$i &>/dev/null
                        #给用户设定初始密码为变量 pass 的值
                    done
        fi
fi
```

例子 3：批量删除用户

```
[root@localhost ~]# vi sh/userdel.sh
#!/bin/bash
#批量删除用户
# Author: shenchao  (E-mail: shenchao@atguigu.com)

user=$(cat /etc/passwd | grep "/bin/bash"|grep -v "root"|cut -d ":" -f 1)
#读取用户信息文件，提取可以登录用户，取消 root 用户，截取第一列用户名
for i in $user
#循环，有多少个普通用户，循环多少次
        do
                userdel -r $i
                #每次循环，删除指定普通用户
        done
```

4　while 循环

```
while [ 条件判断式 ]
    do
        程序
    done
```

对 while 循环来讲，只要条件判断式成立，循环就会一直继续，直到条件判断式不成立，循环才会停止。好吧，我们还是写个 1 加到 100 的例子吧，这种例子虽然对系统管理帮助不大，但是对理解循环非常有帮助：

```
例子：1 加到 100
#!/bin/bash
#从 1 加到 100
```

```
# Author: shenchao （E-mail: shenchao@atguigu.com）

i=1
s=0
#给变量 i 和变量 s 赋值
while [ $i -le 100 ]
#如果变量 i 的值小于等于 100，则执行循环
        do
                s=$(( $s+$i ))
                i=$(( $i+1 ))
        done
echo "The sum is: $s"
```

5　until 循环

再来看看 until 循环，和 while 循环相反，until 循环时只要条件判断式不成立则进行循环，并执行循环程序。一旦循环条件成立，则终止循环。语法如下：

```
until [ 条件判断式 ]
    do
        程序
    done
```

还是写从 1 加到 100 这个例子，注意和 while 循环的区别：

```
例子：从 1 加到 100
[root@localhost ~]# vi sh/until.sh
#!/bin/bash
#从 1 加到 100
# Author: shenchao （E-mail: shenchao@atguigu.com）

i=1
s=0
#给变量 i 和变量 s 赋值
until [ $i -gt 100 ]
#循环直到变量 i 的值大于 100，就停止循环
        do
                s=$(( $s+$i ))
                i=$(( $i+1 ))
        done
echo "The sum is: $s"
```

6　函数

```
function 函数名 ()  {
    程序
}
```

那我们写一个函数吧，还记得从 1 加到 100 这个循环吗？这次我们用函数来实现它，不过不再是

从 1 加到 100 了，我们让用户自己来决定加到多少吧：

例子：

```
[root@localhost ~]# vi sh/function.sh
#!/bin/bash
#接收用户输入的数字，然后从 1 加到这个数字
# Author: shenchao （E-mail: shenchao@atguigu.com）

function sum () {
#定义函数 sum
        s=0
        for (( i=0;i<=$1;i=i+1 ))
        #循环直到 i 大于$1 为止。$1 是函数 sum 的第一个参数
        #在函数中也可以使用位置参数变量，不过这里的$1 指的是函数的第一个参数
                do
                        s=$(( $i+$s ))
                done
        echo "The sum of 1+2+3...+$1 is :  $s"
        #输出 1 加到$1 的和
}

read -p "Please input a number: " -t 30 num
#接收用户输入的数字，并把值赋予变量 num
y=$(echo $num | sed 's/[0-9]//g')
#把变量 num 的值替换为空，并赋予变量 y
if [ -z "$y" ]
#判断变量 y 是否为空，以确定变量 num 中是否为数字
        then
            sum $num
            #调用 sum 函数，并把变量 num 的值作为第一个参数传递给 sum 函数
        else
            echo "Error!! Please input a number!"
            #如果变量 num 的值不是数字，则输出报错信息
fi
```

7　特殊流程控制语句

1、 exit 语句

系统是有 exit 命令的，用于退出当前用户的登录状态。可是在 Shell 脚本中，exit 语句是用来退出当前脚本的。也就是说，在 Shell 脚本中，只要碰到了 exit 语句，后续的程序就不再执行，而直接退出脚本。exit 的语法如下：

```
exit [返回值]
```

如果 exit 命令之后定义了返回值，那么这个脚本执行之后的返回值就是我们自己定义的返回值。可以通过查询$?这个变量，来查看返回值。如果 exit 之后没有定义返回值，脚本执行之后的返回值是执行 exit 语句之前，最后执行的一条命令的返回值。写一个 exit 的例子：

```
[root@localhost ~]# vi sh/exit.sh
```

```
#!/bin/bash
#演示 exit 的作用
# Author: shenchao （E-mail: shenchao@atguigu.com）

read -p "Please input a number: " -t 30 num
#接收用户的输入，并把输入赋予变量 num
y=$(echo $num | sed 's/[0-9]//g')
#如果变量 num 的值是数字，则把 num 的值替换为空，否则不替换
#把替换之后的值赋予变量 y
[ -n "$y" ] && echo "Error! Please input a number!" && exit 18
#判断变量 y 的值如果不为空，输出报错信息，退出脚本，退出返回值为 18
echo "The number is: $num"
#如果没有退出加班，则打印变量 num 中的数字
```

这个脚本中，大家需要思考，如果我输入的不是数字，那么“echo "The number is: $num"”这个脚本是否会执行？当然不会，因为如果输入的不是数字，“[-n "$y"] && echo "Error! Please input a number!" && exit 18”这句脚本会执行，exit 一旦执行脚本就会终止。执行下这个脚本：

```
[root@localhost ~]# chmod 755 sh/exit.sh
#给脚本服务执行权限
[root@localhost ~]# sh/exit.sh              ←执行脚本
Please input a number: test                 ←输入值不是数字，而是 test
Error! Please input a number!               ←输出报错信息，而不会输出 test
[root@localhost ~]# echo $?                 ←查看下返回值
18                                          ←返回值居然真是 18 啊

[root@localhost ~]# sh/exit.sh
Please input a number: 10                   ←输入数字 10
The number is: 10                           ←输出数字 10
```

2、 break 语句

再来看看特殊流程控制语句 break 的作用，当程序执行到 break 语句时，会结束整个当前循环。而 continue 语句也是结束循环的语句，不过 continue 语句单次当前循环，而下次循环会继续。有点晕菜吧，画个示意图解释下 break 语句，如图 12-1 所示：

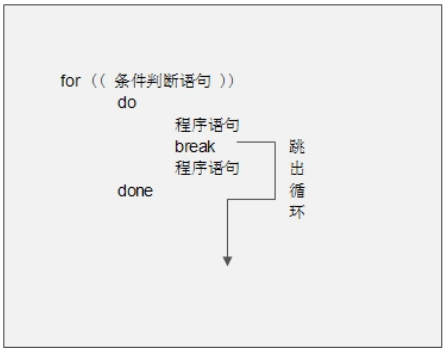

图 12-1 break 示意图

举个例子：

```
[root@localhost ~]# vi sh/break.sh
#!/bin/bash
#演示 break 跳出循环
# Author: shenchao (E-mail: shenchao@atguigu.com)

for (( i=1;i<=10;i=i+1 ))
#循环十次
        do
                if [ "$i" -eq 4 ]
                #如果变量 i 的值等于 4
                        then
                        break
                        #退出整个循环
                fi
                echo $i
                #输出变量 i 的值
        done
```

执行下这个脚本，因为一旦变量 i 的值等于 4，整个循环都会跳出，所以应该只能循环三次：

```
[root@localhost ~]# chmod 755 sh/break.sh
[root@localhost ~]# sh/break.sh
1
2
3
```

3、 continue 语句

再来看看 continue 语句，continue 也是结束流程控制的语句。如果在循环中，continue 语句只会结束单次当前循环，也画个示意图来说明下 continue 语句，如图 12-2 所示：

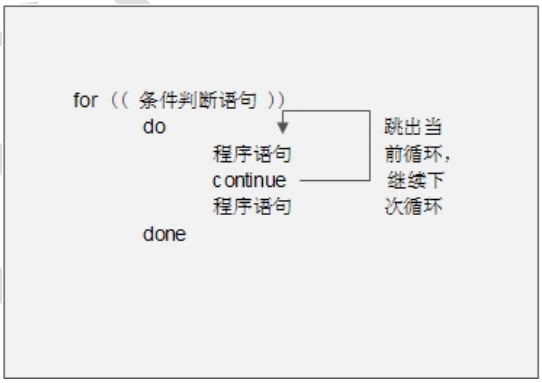

图 12-2 continue 示意图

还是用刚刚的脚本，不过退出语句换成 continue 语句，看看会发生什么情况：

```
[root@localhost ~]# vi sh/continue.sh
#!/bin/bash
#演示 continue 语句
# Author: shenchao (E-mail: shenchao@atguigu.com)
```

```
for (( i=1;i<=10;i=i+1 ))
        do
                if [ "$i" -eq 4 ]
                        then
                        continue
                        #退出语句换成 continue
                fi
                echo $i
        done
```

运行下这个脚本：

```
[root@localhost ~]# chmod 755 sh/continue.sh
#赋予执行权限
[root@localhost ~]# sh/continue.sh
1
2
3
5                                       ←少了 4 这个输出
6
7
8
9
10
```

continue 只会退出单次循环，所以并不影响后续的循环，所以只会少 4 的输出。这个例子和 break 的例子做个比较，应该可以更清楚的说明 break 和 continue 的区别。