Shell 从入门到精通

关于本文档

文档名称	Shell 从入门到精通
作者	李振良
腾讯课堂直播	http://opsdev.ke.qq.com
博客	http://lizhenliang.blog.51cto.com
QQ 技术群	323779636（Shell/Python 运维开发群）
说明	本文档均为个人经验总结，转发请保留出处，抵制不道德行为。 文档会不定期修改或新增知识点，请关注群状态。
最后更新时间	2017-02-26

学习目标

熟悉 Linux 系统常用命令与工具，掌握 Shell 脚本语言语法结构，能独立编写 Shell 脚本，完成自动化运维常规任务，提高工作效率，为以后学习其他语言打下坚实的基础。

目标人群

运维工程师、开发工程师、Linux 系统爱好者或已经具备其他编程语言的人群。

操作系统

本文档实验均采用 CentOS7_X64 系统。需要注意的是，与 CentOS6 或者 Ubuntu 相比，个别命令使用方法会有点不同。

待更新章节

第十章 Shell 编程时常用的系统文件
第十一章 Shell 常用命令与工具
第十二章 Shell 脚本编写实战

目录

第一章　Shell 基础知识

1.1 Shell 简介

Shell 是一个 C 语言编写的脚本语言，它是用户与 Linux 的桥梁，用户输入命令交给 Shell 处理，Shell 将相应的操作传递给内核（Kernel），内核把处理的结果输出给用户。
下面是处理流程示意图：

Shell 既然是工作在 Linux 内核之上，那我们也有必要知道下 Linux 相关知识。
Linux 是一套免费试用和自由传播的类 Unix 操作系统，是一个基于 POSIX 和 UNIX 的多用户、多任务、支持多线程和多 CPU 的操作系统。
1983 年 9 月 27 日，Richard Stallman（理查德-马修-斯托曼）发起 GNU 计划，它的目标是创建一套完全自由的操作系统。为保证 GNU 软件可以自由的使用、复制、修改和发布，所有的 GNU 软件都有一份在禁止其他人添加任何限制的情况下授权所有权利给任何人的协议条款，GNU 通用公共许可证（GNU General Plubic License，GPL），说白了就是不能做商业用途。
GNU 是"GNU is Not Unix"的递归缩写。UNIX 是一种广泛使用的商业操作系统的名称。
1985 年，Richard Stallman 又创立了自由软件基金会（Free Software Foundation，FSF）来为 GNU 计划提供技术、法律以及财政支持。
1990 年，GNU 计划开发主要项目有 Emacs（文本编辑器）、GCC（GNU Compiler Collection，GNU 编译器集合）、Bash 等，GCC 是一套 GNU 开发的编程语言编译器。还有开发一些 UNIX 系统的程序库和工具。
1991 年，Linuxs Torvalds（林纳斯- 托瓦兹）开发出了与 UNIX 兼容的 Linux 操作系统内核并在 GPL 条款下发布。
1992 年，Linux 与其他 GUN 软件结合，完全自由的 GUN/Linux 操作系统正式诞生，简称 Linux。
1995 年 1 月，Bob Young 创办 ACC 公司，以 GNU/Linux 为核心，开发出了 RedHat Linux 商业版。
Linux 基本思想有两点：第一，一切都是文件；第二，每个软件都有确定的用途。与 Unix 思想十分相近。

1.2 Shell 基本分两大类

1.2.1 图形界面 Shell（GUI Shell）
GUI 为 Unix 或者类 Unix 操作系统构造一个功能完善、操作简单以及界面友好的桌面环境。主流桌面环境有 KDE，Gnome 等。
1.2.2 命令行界面 Shell（CLI Shell）
CLI 是在用户提示符下键入可执行指令的界面，用户通过键盘输入指令，完成一系列操作。
在 Linux 系统上主流的 CLI 实现是 Bash，是许多 Linux 发行版默认的 Shell。还有许多 Unix 上 Shell，例如 tcsh、csh、ash、bsh、ksh 等。

1.3 第一个 Shell 脚本

本教程主要讲解在大多 Linux 发行版下默认 Bash Shell。Linux 系统是 RedHat 下的 CentOS 操作系统，完全免费。与其商业版 RHEL（Red Hat Enterprise Linux）出自同样的源代码，不同的是 CentOS 并不包含封闭源代码软件和售后支持。
用 vi 打开 test.sh，编写：

```
# vi test.sh
#!/bin/bash
echo "Hello world!"
```

第一行指定解释器，第二行打印 Hello world!
写好后，开始执行，执行 Shell 脚本有三种方法：
方法 1：直接用 bash 解释器执行

```
# bash test.sh
Hello world!
```

方法 2：添加可执行权限

```
# ll test.sh
-rw-r--r--. 1 root root 32 Aug 18 01:07 test.sh
# chmod +x test.sh
# ./test.sh
-bash: ./test.sh: Permission denied
# chmod +x test.sh
# ./test.sh # ./在当前目录
Hello world!
```

这种方式默认根据脚本第一行指定的解释器处理，如果没写以当前默认 Shell 解释器执行。
方法 3：source 命令执行，以当前默认 Shell 解释器执行

```
# source test.sh
Hello world!
```

1.4 Shell 变量

1.4.1 系统变量
在命令行提示符直接执行 env、set 查看系统或环境变量。env 显示用户环境变量，set 显示 Shell 预先定义好的变量以及用户变量。可以通过 export 导出成用户变量。

一些写 Shell 脚本时常用的系统变量：

$SHELL	默认 Shell
$HOME	当前用户家目录
$IFS	内部字段分隔符
$LANG	默认语言
$PATH	默认可执行程序路径
$PWD	当前目录
$UID	当前用户 ID
$USER	当前用户
$HISTSIZE	历史命令大小，可通过 HISTTIMEFORMAT 变量设置命令执行时间
$RANDOM	随机生成一个 0 至 32767 的整数
$HOSTNAME	主机名

1.4.2 普通变量与临时环境变量

普通变量定义：VAR=value

临时环境变量定义：export VAR=value

变量引用：$VAR

下面看下他们之间区别：

Shell 进程的环境变量作用域是 Shell 进程，当 export 导入到系统变量时，则作用域是 Shell 进程及其 Shell 子进程。

```
[root@localhost ~]# ps axjf  |grep pts
  1580  78902  78902  78902 ?           -1 Ss       0   0:00  \_ sshd: root@pts/0
 78902  78904  78904  78904 pts/0     79092 Ss       0   0:00      \_ -bash
 78904  79092  79092  78904 pts/0     79092 R+       0   0:00          \_ ps axjf
 78904  79093  79092  78904 pts/0     79092 S+       0   0:00          \_ grep --color=auto pts
[root@localhost ~]# echo $$
78904
[root@localhost ~]# VAR=123
[root@localhost ~]# echo $VAR
123
[root@localhost ~]# bash
[root@localhost ~]# echo $$
79136
[root@localhost ~]# ps axjf  |grep pts
  1580  78902  78902  78902 ?           -1 Ss       0   0:00  \_ sshd: root@pts/0
 78902  78904  78904  78904 pts/0     79152 Ss       0   0:00      \_ -bash
 78904  79136  79136  78904 pts/0     79152 S        0   0:00          \_ bash
 79136  79152  79152  78904 pts/0     79152 R+       0   0:00              \_ ps axjf
 79136  79153  79152  78904 pts/0     79152 S+       0   0:00              \_ grep --color=auto pts
[root@localhost ~]# echo $VAR

[root@localhost ~]# exit
exit
You have new mail in /var/spool/mail/root
[root@localhost ~]# echo $VAR
123
[root@localhost ~]# export VAR
You have new mail in /var/spool/mail/root
[root@localhost ~]# bash
[root@localhost ~]# echo $$
79242
[root@localhost ~]# ps axjf  |grep pts
  1580  78902  78902  78902 ?           -1 Ss       0   0:00  \_ sshd: root@pts/0
 78902  78904  78904  78904 pts/0     79258 Ss       0   0:00      \_ -bash
 78904  79242  79242  78904 pts/0     79258 S        0   0:00          \_ bash
 79242  79258  79258  78904 pts/0     79258 R+       0   0:00              \_ ps axjf
 79242  79259  79258  78904 pts/0     79258 S+       0   0:00              \_ grep --color=auto pts
[root@localhost ~]# echo $VAR
123
```

```
[root@localhost ~]# ps -ef |grep ssh
root        1580       1  0 Jan01 ?        00:00:00 /usr/sbin/sshd -D
```

ps axjf 输出的第一列是 PPID（父进程 ID），第二列是 PID（子进程 ID）

当 SSH 连接 Shell 时，当前终端 PPID（-bash）是 sshd 守护程序的 PID（root@pts/0），因此在当前终端下的所有进程的 PPID 都是-bash 的 PID，比如执行命令、运行脚本。

所以当在-bash 下设置的变量，只在-bash 进程下有效，而-bash 下的子进程 bash 是无效的，当 export 后才有效。

进一步说明：再重新连接 SSH，去除上面定义的变量测试下

```
[root@localhost ~]# ps -axjf |grep pts
  1580  79887  79887  79887 ?           -1 Ss       0   0:00  \_ sshd: root@pts/0
 79887  79891  79891  79891 pts/0     79934 Ss       0   0:00      \_ -bash
 79891  79934  79934  79891 pts/0     79934 R+       0   0:00          \_ ps -axjf
 79891  79935  79934  79891 pts/0     79934 S+       0   0:00          \_ grep --color=auto pts
[root@localhost ~]# echo $$
79891
[root@localhost ~]# VAR=123
[root@localhost ~]# cat test.sh
#!/bin/bash
ps -axjf |grep pts
echo $$
echo $VAR
[root@localhost ~]# bash test.sh
  1580  79887  79887  79887 ?           -1 Ss       0   0:00  \_ sshd: root@pts/0
 79887  79891  79891  79891 pts/0     79950 Ss       0   0:00      \_ -bash
 79891  79950  79950  79891 pts/0     79950 S+       0   0:00          \_ bash test.sh
 79950  79951  79950  79891 pts/0     79950 R+       0   0:00              \_ ps -axjf
 79950  79952  79950  79891 pts/0     79950 S+       0   0:00              \_ grep pts
79950

[root@localhost ~]# export VAR
[root@localhost ~]# bash test.sh
  1580  79887  79887  79887 ?           -1 Ss       0   0:00  \_ sshd: root@pts/0
 79887  79891  79891  79891 pts/0     79955 Ss       0   0:00      \_ -bash
 79891  79955  79955  79891 pts/0     79955 S+       0   0:00          \_ bash test.sh
 79955  79956  79955  79891 pts/0     79955 R+       0   0:00              \_ ps -axjf
 79955  79957  79955  79891 pts/0     79955 S+       0   0:00              \_ grep pts
79955
123
```

所以在当前 shell 定义的变量一定要 export，否则在写脚本时，会引用不到。

还需要注意的是退出终端后，所有用户定义的变量都会清除。
在/etc/profile 下定义的变量就是这个原理，后面有章节会讲解 Linux 常用变量文件。
1.4.3 位置变量
位置变量指的是函数或脚本后跟的第 n 个参数。
$1-$n，需要注意的是从第 10 个开始要用花括号调用，例如${10}
shift 可对位置变量控制，例如：

```
#!/bin/bash
echo "1: $1"
shift
echo "2: $2"
shift
echo "3: $3"
# bash test.sh a b c
1: a
2: c
3:
```

每执行一次 shift 命令，位置变量个数就会减一，而变量值则提前一位。shift n，可设置向前移动 n 位。
1.4.4 特殊变量

$0	脚本自身名字
$?	返回上一条命令是否执行成功，0 为执行成功，非 0 则为执行失败
$#	位置参数总数
$*	所有的位置参数被看做一个字符串
$@	每个位置参数被看做独立的字符串
$$	当前进程 PID
$!	上一条运行后台进程的 PID

1.5 变量引用

赋值运算符	示例
=	变量赋值
+=	两个变量相加

1.5.1 自定义变量与引用

```
# VAR=123
# echo $VAR
123
# VAR+=456
```

```
# echo $VAR
123456
```

Shell 中所有变量引用使用$符，后跟变量名。
有时个别特殊字符会影响正常引用，那么需要使用${VAR}，例如：

```
# VAR=123
# echo $VAR
123
# echo $VAR_    #  Shell 允许 VAR_为变量名，所以此引用认为这是一个有效的变量名，故此返回空

# echo ${VAR}
123
```

还有时候变量名与其他字符串紧碍着，也会误认为是整个变量：

```
# echo $VAR456

# echo ${VAR}456
123456
```

1.5.2 将命令结果作为变量值

```
# VAR=`echo 123`
# echo $VAR
123
# VAR=$(echo 123)
# echo $VAR
123
```

这里的反撇号等效于$()，都是用于执行 Shell 命令。

1.6 双引号和单引号

在变量赋值时，如果值有空格，Shell 会把空格后面的字符串解释为命令：

```
# VAR=1 2 3
-bash: 2: command not found
# VAR="1 2 3"
# echo $VAR
1 2 3
# VAR='1 2 3'
# echo $VAR
1 2 3
```

看不出什么区别，再举个说明：

```
# N=3
# VAR="1 2 $N"
# echo $VAR
1 2 3
```

```
# VAR='1 2 $N'
# echo $VAR
1 2 $N
```

单引号是告诉 Shell 忽略特殊字符，而双引号则解释特殊符号原有的意义，比如$、！。

1.7 注释

Shell 注释也很简单，只要在每行前面加个#号，即表示 Shell 忽略解释。

第二章　Shell 字符串处理之${}

上一章节讲解了为什么用${}引用变量，${}还有一个重要的功能，就是文本处理，单行文本基本上可以满足你所有需求。

2.1 获取字符串长度

```
# VAR='hello world!'
# echo $VAR
hello world!
# echo ${#VAR}
12
```

2.2 字符串切片

格式：
${parameter:offset}
${parameter:offset:length}
截取从 offset 个字符开始，向后 length 个字符。

```
截取 hello 字符串：
# VAR='hello world!'
# echo ${VAR:0:5}
hello
截取 wo 字符：
# echo ${VAR:6:2}
wo
截取 world 字符串：
# echo ${VAR:5:-1}
world
截取最后一个字符：
# echo ${VAR:(-1)}
!
截取最后二个字符：
# echo ${VAR:(-2)}
```

```
d!
截取从倒数第 3 个字符后的 2 个字符：
# echo ${VAR:(-3):2}
ld
```

2.3 替换字符串

格式：${parameter/pattern/string}

```
# VAR='hello world world!'
将第一个 world 字符串替换为 WORLD：
# echo ${VAR/world/WORLD}
hello WORLD world!
将全部 world 字符串替换为 WORLD：
# echo ${VAR//world/WORLD}
hello WORLD WORLD!
```

2.4 字符串截取

格式：

```
${parameter#word}    # 删除匹配前缀
${parameter##word}
${parameter%word}    # 删除匹配后缀
${parameter%%word}
# 去掉左边，最短匹配模式，##最长匹配模式。
% 去掉右边，最短匹配模式，%%最长匹配模式。
```

```
# URL="http://www.baidu.com/baike/user.html"
以//为分隔符截取右边字符串：
# echo ${URL#*//}
www.baidu.com/baike/user.html
以/为分隔符截取右边字符串：
# echo ${URL##*/}
user.html
以//为分隔符截取左边字符串：
# echo ${URL%%//*}
http:
以/为分隔符截取左边字符串：
# echo ${URL%/*}
http://www.baidu.com/baike
以.为分隔符截取左边：
# echo ${URL%.*}
http://www.baidu.com/baike/user
以.为分隔符截取右边：
# echo ${URL##*.}
html
```

2.5 变量状态赋值

```
${VAR:-string}    如果 VAR 变量为空则返回 string
${VAR:+string}    如果 VAR 变量不为空则返回 string
${VAR:=string}    如果 VAR 变量为空则重新赋值 VAR 变量值为 string
${VAR:?string}    如果 VAR 变量为空则将 string 输出到 stderr
```

```
如果变量为空就返回 hello world!:
# VAR=
# echo ${VAR:-'hello world!'}
hello world!
如果变量不为空就返回 hello world!:
# VAR="hello"
# echo ${VAR:+'hello world!'}
hello world!
如果变量为空就重新赋值:
# VAR=
# echo ${VAR:=hello}
hello
# echo $VAR
hello
如果变量为空就将信息输出 stderr:
# VAR=
# echo ${VAR:?value is null}
-bash: VAR: value is null
```

${}主要用途大概就这么多了，另外还可以获取数组元素，在后面章节会讲到。

2.6 字符串颜色

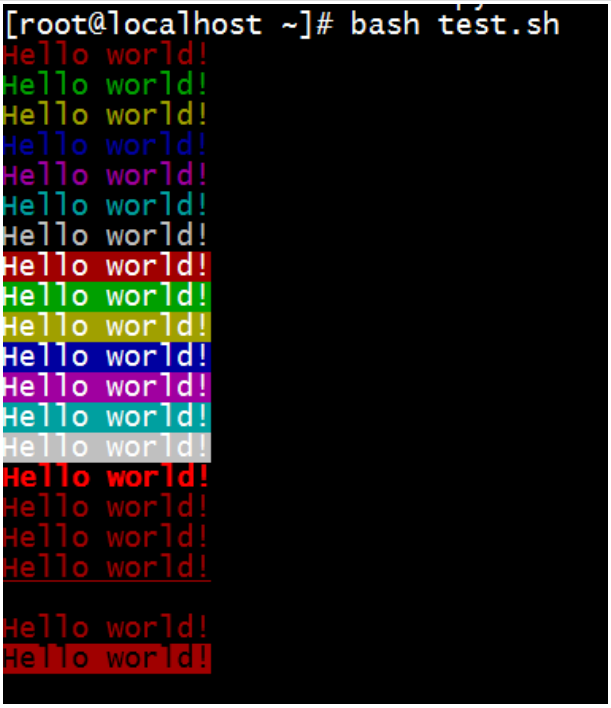
再介绍下字符串输出颜色，有时候关键地方需要醒目，颜色是最好的方式：

字体颜色	字体背景颜色	显示方式
30：黑 31：红 32：绿 33：黄 34：蓝色 35：紫色 36：深绿 37：白色	40：黑 41：深红 42：绿 43：黄色 44：蓝色 45：紫色 46：深绿 47：白色	0：终端默认设置 1：高亮显示 4：下划线 5：闪烁 7：反白显示 8：隐藏
格式： \033[1;31;40m # 1 是显示方式，可选。31 是字体颜色。40m 是字体背景颜色。 \033[0m # 恢复终端默认颜色，即取消颜色设置。		

示例：

```
#!/bin/bash
# 字体颜色
for i in {31..37}; do
```

```
      echo -e "\033[$i;40mHello world!\033[0m"
done
# 背景颜色
for i in {41..47}; do
      echo -e "\033[47;${i}mHello world!\033[0m"
done
# 显示方式
for i in {1..8}; do
      echo -e "\033[$i;31;40mHello world!\033[0m"
done
```



第三章　Shell 表达式与运算符

3.1 条件表达式

表达式	示例
[expression]	[1 -eq 1]
[[expression]]	[[1 -eq 1]]
test expression	test 1 -eq 1 ,等同于[]

3.2 整数比较符

比较符	描述	示例

-eq，equal	等于	[1 -eq 1]为 true
-ne，not equal	不等于	[1 -ne 1]为 false
-gt，greater than	大于	[2 -gt 1]为 true
-lt，lesser than	小于	[2 -lt 1]为 false
-ge，greater or equal	大于或等于	[2 -ge 1]为 true
-le，lesser or equal	小于或等于	[2 -le 1]为 false

3.3 字符串比较符

运算符	描述	示例
==	等于	["a" == "a"]为 true
!=	不等于	["a" != "a"]为 false
>	大于，判断字符串时根据 ASCII 码表顺序，不常用	在[]表达式中：[2 \> 1]为 true 在[[]]表达式中：[[2 > 1]]为 true 在(())表达式中：((3 > 2))为 true
<	小于，判断字符串时根据 ASCII 码表顺序，不常用	在[]表达式中：[2 \< 1]为 false 在[[]]表达式中：[[2 < 1]]为 false 在(())表达式中：((3 < 2))为 false
>=	大于等于	在(())表达式中：((3 >= 2))为 true
<=	小于等于	在(())表达式中：((3 <= 2))为 false
-n	字符串长度不等于 0 为真	VAR1=1;VAR2="" [-n "$VAR1"]为 true [-n "$VAR2"]为 false
-z	字符串长度等于 0 为真	VAR1=1;VAR2="" [-z "$VAR1"]为 false [-z "$VAR2"]为 true
str	字符串存在为真	VAR1=1;VAR2="" [$VAR1]为 true [$VAR2]为 false

需要注意的是，使用-z 或-n 判断字符串长度时，变量要加双引号。
举例说明：

```
# [ -z $a ] && echo yes || echo no
```

```
yes
# [ -n $a ] && echo yes || echo no
yes
# 加了双引号才能正常判断是否为空
# [ -z "$a" ] && echo yes || echo no
yes
# [ -n "$a" ] && echo yes || echo no
no
# 使用了双中括号就不用了双引号
# [[ -n $a ]] && echo yes || echo no
no
# [[ -z $a ]] && echo yes || echo no
yes
```

3.4 文件测试

测试符	描述	示例
-e	文件或目录存在为真	[-e path] path 存在为 true
-f	文件存在为真	[-f file_path] 文件存在为 true
-d	目录存在为真	[-d dir_path] 目录存在为 true
-r	有读权限为真	[-r file_path] file_path 有读权限为 true
-w	有写权限为真	[-w file_path] file_path 有写权限为 true
-x	有执行权限为真	[-x file_path] file_path 有执行权限为 true
-s	文件存在并且大小大于 0 为真	[-s file_path] file_path 存在并且大小大于 0 为 true

3.5 布尔运算符

运算符	描述	示例
!	非关系，条件结果取反	[! 1 -eq 2]为 true
-a	和关系，在[]表达式中使用	[1 -eq 1 -a 2 -eq 2]为 true
-o	或关系，在[]表达式中使用	[1 -eq 1 -o 2 -eq 1]为 true

3.6 逻辑判断符

判断符	描述	示例
&&	逻辑和，在[[]]和(())表达式中或判断表达式是否为真时使用	[[1 -eq 1 && 2 -eq 2]]为 true ((1 == 1 && 2 == 2))为 true [1 -eq 1] && echo yes 如果&&前面表达式为 true 则执行后面的
||	逻辑或，在[[]]和(())表达式中或判断表达式是否为真时使用	[[1 -eq 1 || 2 -eq 1]]为 true ((1 == 1 || 2 == 2))为 true [1 -eq 2] || echo yes 如果||前面表达式为 false 则执行后面的

3.7 整数运算

运算符	描述
+	加法
-	减法
*	乘法
/	除法
%	取余

运算表达式	示例
$(())	$((1+1))
$[]	$[1+1]

上面两个都不支持浮点运算。
$(())表达式还有一个用途，三目运算：

```
# 如果条件为真返回 1，否则返回 0
# echo $((1<0))
0
# echo $((1>0))
1
指定输出数字：
# echo $((1>0?1:2))
1
# echo $((1<0?1:2))
```

```
2
注意：返回值不支持字符串
```

3.8 其他运算工具（let/expr/bc）

除了 Shell 本身的算数运算表达式，还有几个命令支持复杂的算数运算：

命令	描述	示例
let	赋值并运算，支持++、--	let VAR=(1+2)*3 ; echo $VAR x=10 ; y=5 let x++;echo $x 每执行一次 x 加 1 let y--;echo $y 每执行一次 y 减 1 let x+=2 每执行一次 x 加 2 let x-=2 每执行一次 x 减 2
expr	乘法*需要加反斜杠转义*	expr 1 * 2　运算符两边必须有空格 expr \(1 + 2 \) * 2　使用双括号时要转义
bc	计算器，支持浮点运算、平方等	bc 本身就是一个计算器，可直接输入命令，进入解释器。 echo 1 + 2 |bc 将管道符前面标准输出作为 bc 的标准输入 echo "1.2+2" |bc echo "10^10" |bc echo 'scale=2;10/3' |bc　用 scale 保留两位小数点

由于 Shell 不支持浮点数比较，可以借助 bc 来完成需求：

```
# echo "1.2 < 2" |bc
1
# echo "1.2 > 2" |bc
0
# echo "1.2 == 2.2" |bc
0
# echo "1.2 != 2.2" |bc
1
```

看出规律了嘛？运算如果为真返回 1，否则返回 0，写一个例子：

```
# [ $(echo "2.2 > 2" |bc) -eq 1 ] && echo yes || echo no
yes
# [ $(echo "2.2 < 2" |bc) -eq 1 ] && echo yes || echo no
no
```

expr 还可以对字符串操作：

获取字符串长度：

```
# expr length "string"
6
```

截取字符串：

```
# expr substr "string" 4 6
ing
```

获取字符在字符串中出现的位置：

```
# expr index "string" str
1
# expr index "string" i
4
```

获取字符串开始字符出现的长度：

```
# expr match "string" s.*
6
# expr match "string" str
3
```

3.9 Shell 括号用途总结

看到这里，想一想里面所讲的小括号、中括号的用途，是不是有点懵逼了。那我们总结一下！

( )	用途 1：在运算中，先计算小括号里面的内容 用途 2：数组 用途 3：匹配分组
(( ))	用途 1：表达式，不支持-eq 这类的运算符。不支持-a 和-o，支持<=、>=、<、>这类比较符和&&、|| 用途 2：C 语言风格的 for(())表达式
$( )	执行 Shell 命令，与反撇号等效
$(( ))	用途 1：简单算数运算 用途 2：支持三目运算符　$((表达式?数字:数字　))
[ ]	条件表达式，里面不支持逻辑判断符
[[ ]]	条件表达式，里面不支持-a 和-o，不支持<=和>=比较符，支持-eq、<、>这类比较符。支持=~模式匹配，也可以不用双引号也不会影响原意，比[]更加通用
$[ ]	简单算数运算
{ }	对逗号（,）和点点（...）起作用，比如 touch {1,2}创建 1 和 2 文件，touch {1..3}创建 1、2 和 3 文件
${ }	用途 1：引用变量 用途 2：字符串处理

第四章　Shell 流程控制

流程控制是改变程序运行顺序的指令。

4.1 if 语句

格式：if list; then list; [elif list; then list;] ... [else list;] fi

4.1.1 单分支

```
if 条件表达式; then
      命令
fi
```

示例：

```
#!/bin/bash
N=10
if [ $N -gt 5 ]; then
      echo yes
fi
# bash test.sh
yes
```

4.1.2 双分支

```
if 条件表达式; then
      命令
else
      命令
fi
```

示例 1：

```
#!/bin/bash
N=10
if [ $N -lt 5 ]; then
      echo yes
else
      echo no
fi
# bash test.sh
no
```

示例 2：判断 crond 进程是否运行

```
#!/bin/bash
NAME=crond
NUM=$(ps -ef |grep $NAME |grep -vc grep)
if [ $NUM -eq 1 ]; then
     echo "$NAME running."
else
     echo "$NAME is not running!"
fi
```

示例 3：检查主机是否存活

```
#!/bin/bash
```

```
if ping -c 1 192.168.1.1 >/dev/null; then
     echo "OK."
else
     echo "NO!"
fi
```

if 语句可以直接对命令状态进行判断，就省去了获取$?这一步！

4.1.3 多分支

```
if 条件表达式; then
     命令
elif 条件表达式; then
    命令
else
    命令
fi
```

当不确定条件符合哪一个时，就可以把已知条件判断写出来，做相应的处理。

示例 1：

```
#!/bin/bash
N=$1
if [ $N -eq 3 ]; then
    echo "eq 3"
elif [ $N -eq 5 ]; then
    echo "eq 5"
elif [ $N -eq 8 ]; then
    echo "eq 8"
else
    echo "no"
fi
```

如果第一个条件符合就不再向下匹配。

示例 2：根据 Linux 不同发行版使用不同的命令安装软件

```
#!/bin/bash
if [ -e /etc/redhat-release ]; then
    yum install wget -y
elif [ $(cat /etc/issue |cut -d' ' -f1) == "Ubuntu" ]; then
    apt-get install wget -y
else
    Operating system does not support.
    exit
fi
```

4.2 for 语句

格式：for name [[in [word ...]] ;] do list ; done

```
for 变量名 in 取值列表; do
```

```
      命令
done
```

示例：

```
#!/bin/bash
for i in {1..3}; do
      echo $i
done
# bash test.sh
1
2
3
```

for 的语法也可以这么写：

```
#!/bin/bash
for i in "$@"; {      # $@是将位置参数作为单个来处理
      echo $i
}
# bash test.sh 1 2 3
1
2
3
```

默认 for 循环的取值列表是以空白符分隔，也就是第一章讲系统变量里的$IFS:

```
#!/bin/bash
for i in 12 34; do
      echo $i
done
# bash test.sh
12
34
```

如果想指定分隔符，可以重新赋值$IFS 变量：

```
#!/bin/bash
OLD_IFS=$IFS
IFS=":"
for i in $(head -1 /etc/passwd); do
    echo $i
done
IFS=$OLD_IFS   # 恢复默认值
# bash test.sh
root
x
0
0
root
/root
```

```
/bin/bash
```

for 循环还有一种 C 语言风格的语法，常用于计数、打印数字序列：

```
for (( expr1 ; expr2 ; expr3 )) ; do list ; done
```

```
#!/bin/bash
for ((i=1;i<=5;i++)); do   # 也可以 i--
      echo $i
done
```

示例 1：检查多个主机是否存活

```
#!/bin/bash
for ip in 192.168.1.{1..254}; do
    if ping -c 1 $ip >/dev/null; then
        echo "$ip OK."
    else
        echo "$ip NO!"
    fi
done
```

示例 2：检查多个域名是否可以访问

```
#!/bin/bash
URL="www.baidu.com www.sina.com www.jd.com"
for url in $URL; do
    HTTP_CODE=$(curl -o /dev/null -s -w %{http_code} http://$url)
    if [ $HTTP_CODE -eq 200 -o $HTTP_CODE -eq 301 ]; then
        echo "$url OK."
    else
        echo "$url NO!"
    fi
done
```

4.3 while 语句

格式：while list; do list; done

```
while 条件表达式; do
      命令
done
```

示例 1：

```
#!/bin/bash
N=0
while [ $N -lt 5 ]; do
      let N++
      echo $N
done
# bash test.sh
```

```
1
2
3
4
5
```

当条件表达式为 false 时，终止循环。
示例 2：条件表达式为 true，将会产生死循环

```
#!/bin/bash
while [ 1 -eq 1 ]; do
      echo "yes"
done
```

也可以条件表达式直接用 true：

```
#!/bin/bash
while true; do
      echo "yes"
done
```

还可以条件表达式用冒号，冒号在 Shell 中的意思是不做任何操作。但状态是 0，因此为 true：

```
#!/bin/bash
while :; do
      echo "yes"
done
```

示例 3：逐行处理文本
文本内容：

```
# cat a.txt
a b c
1 2 3
x y z
```

要想使用 while 循环逐行读取 a.txt 文件，有三种方式：
方式 1：

```
#!/bin/bash
cat ./a.txt | while read LINE; do
      echo $LINE
done
```

方式 2：

```
#!/bin/bash
while read LINE; do
      echo $LINE
done < ./a.txt
```

方式 3：

```
#!/bin/bash
```

```
exec < ./a.txt   # 读取文件作为标准输出
while read LINE; do
      echo $LINE
done
```

与 while 关联的还有一个 until 语句，它与 while 不同之处在于，是当条件表达式为 false 时才循环，实际使用中比较少，这里不再讲解。

4.4 break 和 continue 语句

break 是终止循环。
continue 是跳出当前循环。
示例 1：在死循环中，满足条件终止循环

```
#!/bin/bash
N=0
while true; do
      let N++
      if [ $N -eq 5 ]; then
            break
      fi
      echo $N
done
# bash test.sh
1
2
3
4
```

里面用了 if 判断，并用了 break 语句，它是跳出循环。与其关联的还有一个 continue 语句，它是跳出本次循环。
示例 2：举例子说明 continue 用法

```
#!/bin/bash
N=0
while [ $N -lt 5 ]; do
      let N++
      if [ $N -eq 3 ]; then
            continue
      fi
      echo $N
done
# bash test.sh
1
2
4
5
```

当变量 N 等于 3 时，continue 跳过了当前循环，没有执行下面的 echo。
注意：continue 与 break 语句只能循环语句中使用。

4.5 case 语句

case 语句一般用于选择性来执行对应部分块命令。
格式：case word in [[(] pattern [| pattern] ...) list ;;] ... esac

```
case 模式名  in
      模式 1)
            命令
            ;;
      模式 2)
            命令
            ;;
      *)
            不符合以上模式执行的命令
esac
```

每个模式必须以右括号结束，命令结尾以双分号结束。
示例：根据位置参数匹配不同的模式

```
#!/bin/bash
case $1 in
      start)
            echo "start."
            ;;
      stop)
            echo "stop."
            ;;
      restart)
            echo "restart."
            ;;
      *)
            echo "Usage: $0 {start|stop|restart}"
esac

# bash test.sh
Usage: test.sh {start|stop|restart}
# bash test.sh start
start.
# bash test.sh stop
stop.
# bash test.sh restart
restart.
```

上面例子是不是有点眼熟，在 Linux 下有一部分服务启动脚本都是这么写的。
模式也支持正则，匹配哪个模式就执行那个：

```
#!/bin/bash
case $1 in
    [0-9])
        echo "match number."
```

```
        ;;
    [a-z])
        echo "match letter."
        ;;
    '-h'|'--help')
        echo "help"
        ;;
    *)
        echo "Input error!"
        exit
esac

# bash test.sh 1
match number.
# bash test.sh a
match letter.
# bash test.sh -h
help
# bash test.sh --help
help
```

模式支持的正则有：*、?、[ ]、[.-.]、|。后面有章节单独讲解 Shell 正则表达式。

4.6 select 语句

select 是一个类似于 for 循环的语句。
格式：select name [in word] ; do list ; done

```
select 变量 in 选项1 选项2; do
      break
done
```

示例：

```
#!/bin/bash
select mysql_version in 5.1 5.6; do
      echo $mysql_version
done
# bash test.sh
1) 5.1
2) 5.6
#? 1
5.1
#? 2
5.6
```

用户输入编号会直接赋值给变量 mysql_version。作为菜单用的话，循环第二次后就不再显示菜单了，并不能满足需求。
在外面加个死循环，每次执行一次 select 就 break 一次，这样就能每次显示菜单了：

```
#!/bin/bash
```

```
while true; do
      select mysql_version in 5.1 5.6; do
            echo $mysql_version
            break
      done
done
# bash test.sh
1) 5.1
2) 5.6
#? 1
5.1
1) 5.1
2) 5.6
#? 2
5.6
1) 5.1
2) 5.6
```

如果再判断对用户输入的编号执行相应的命令，如果用 if 语句多分支的话要复杂许多，用 case 语句就简单多了。

```
#!/bin/bash
PS3="Select a number: "
while true; do
      select mysql_version in 5.1 5.6 quit; do
            case $mysql_version in
                  5.1)
                        echo "mysql 5.1"
                        break
                        ;;
                  5.6)
                        echo "mysql 5.6"
                        break
                        ;;
                  quit)
                        exit
                        ;;
                  *)
                        echo "Input error, Please enter again!"
                        break
            esac
      done
done
# bash test.sh
1) 5.1
2) 5.6
3) quit
Select a number: 1
mysql 5.1
```

```
1) 5.1
2) 5.6
3) quit
Select a number: 2
mysql 5.6
1) 5.1
2) 5.6
3) quit
Select a number: 3
```

如果不想用默认的提示符，可以通过重新赋值变量 PS3 来自定义。这下就比较完美了！

第五章　Shell 函数与数组

5.1 函数

格式：

```
func() {
      command
}
```

function 关键字可写，也可不写。
示例 1：

```
#!/bin/bash
func() {
      echo "This is a function."
}
func
# bash test.sh
This is a function.
```

Shell 函数很简单，函数名后跟双括号，再跟双大括号。通过函数名直接调用，不加小括号。
示例 2：函数返回值

```
#!/bin/bash
func() {
      VAR=$((1+1))
      return $VAR
      echo "This is a function."
}
func
echo $?
# bash test.sh
2
```

return 在函数中定义状态返回值，返回并终止函数，但返回的只能是 0-255 的数字，类似于 exit。
示例 3：函数传参

```
#!/bin/bash
func() {
      echo "Hello $1"
}
func world
# bash test.sh
Hello world
```

通过 Shell 位置参数给函数传参。
函数也支持递归调用，也就是自己调用自己。
例如：

```
#!/bin/bash
test() {
      echo $1
      sleep 1
      test hello
}
test
```

执行会一直在调用本身打印 hello，这就形成了闭环。
像经典的 fork 炸弹就是函数递归调用：
:(){ :|:& };: 或 .(){.|.&};.
分析下：
:(){ } 定义一个函数，函数名是冒号。
: 调用自身函数
| 管道符
: 再一次递归调用自身函数
:|: 表示每次调用函数":"的时候就会生成两份拷贝。
& 放到后台
; 分号是继续执行下一个命令，可以理解为换行。
: 最后一个冒号是调用函数。
因此不断生成新进程，直到系统资源崩溃。
一般递归函数用的也少，了解下即可！

5.2 数组

数组是相同类型的元素按一定顺序排列的集合。
格式：
array=(元素 1 元素 2 元素 3 ...)
用小括号初始化数组，元素之间用空格分隔。
定义方法 1：初始化数组
array=(a b c)
定义方法 2：新建数组并添加元素
array[下标]=元素
定义方法 3：将命令输出作为数组元素
array=($(command))
数组操作：

获取所有元素：

```
# echo ${array[*]}   # *和@ 都是代表所有元素
a b c
获取元素下标：
# echo ${!a[@]}
0 1 2
获取数组长度：
# echo ${#array[*]}
3
获取第一个元素：
# echo ${array[0]}
a
获取第二个元素：
# echo ${array[1]}
b
获取第三个元素：
# echo ${array[2]}
c
添加元素：
# array[3]=d
# echo ${array[*]}
a b c d
添加多个元素：
# array+=(e f g)
# echo ${array[*]}
a b c d e f g
删除第一个元素：
# unset array[0]     # 删除会保留元素下标
# echo ${array[*]}
b c d e f g
删除数组：
# unset array
```

数组下标从 0 开始。

示例 1：讲 seq 生成的数字序列循环放到数组里面

```
#!/bin/bash
for i in $(seq 1 10); do
      array[a]=$i
      let a++
done
echo ${array[*]}
# bash test.sh
1 2 3 4 5 6 7 8 9 10
```

示例 2：遍历数组元素

```
方法 1：
#!/bin/bash
IP=(192.168.1.1 192.168.1.2 192.168.1.3)
for ((i=0;i<${#IP[*]};i++)); do
```

```
      echo ${IP[$i]}
done
# bash test.sh
192.168.1.1
192.168.1.2
192.168.1.3
方法 2：
#!/bin/bash
IP=(192.168.1.1 192.168.1.2 192.168.1.3)
for   IP   in ${IP[*]}; do
        echo $IP
done
```

第六章　Shell 正则表达式

正则表达式在每种语言中都会有，功能就是匹配符合你预期要求的字符串。
Shell 正则表达式分为两种：
基础正则表达式：BRE（basic regular express）
扩展正则表达式：ERE（extend regular express），扩展的表达式有+、?、|和()
下面是一些常用的正则表达式符号，我们先拿 grep 工具举例说明。

符号	描述	示例
.	匹配除换行符(\n)之外的任意单个字符	匹配 123： echo -e "123\n456" |grep '1.3'
^	匹配前面字符串开头	匹配以 abc 开头的行： echo -e "abc\nxyz" |grep ^abc
$	匹配前面字符串结尾	匹配以 xyz 结尾的行： echo -e "abc\nxyz" |grep xyz$
*	匹配前一个字符零个或多个	匹配 x、xo 和 xoo： echo -e "x\nxo\nxoo\no\noo" |grep "xo*" x 是必须的，批量了 0 零个或多个
+	匹配前面字符 1 个或多个	匹配 abc 和 abcc： echo -e "abc\nabcc\nadd" |grep -E 'ab+' 匹配单个数字：echo "113" |grep -o '[0-9]' 连续匹配多个数字：echo "113" |grep -E -o '[0-9]+'
?	匹配前面字符 0 个或 1 个	匹配 ac 或 abc： echo -e "ac\nabc\nadd" |grep -E 'a?c'

[ ]	匹配中括号之中的任意一个字符	匹配 a 或 c： echo -e "a\nb\nc" |grep '[ac]'
[.-.]	匹配中括号中范围内的任意一个字符	匹配所有字母： echo -e "a\nb\nc" |grep '[a-z]'
[^]	匹配[^字符]之外的任意一个字符	匹配 a 或 b： echo -e "a\nb\nc" |grep '[^c-z]' 匹配末尾数字：echo "abc:cde;123" |grep -E '[^;]+$'
^[^]	匹配不是中括号内任意一个字符开头的行	匹配不是#开头的行： grep '^[^#]' /etc/httpd/conf/httpd.conf
{n}或{n,}	匹配花括号前面字符至少 n 个字符	匹配 abc 字符串（至少三个字符以上字符串）： echo -e "a\nabc\nc" |grep -E '[a-z]{3}'
{n,m}	匹配花括号前面字符至少 n 个字符，最多 m 个字符	匹配 12 和 123（不加边界符会匹配单个字符）： echo -e "1\n12\n123\n1234" |grep -E -w -o '[0-9]{2,3}'
\<	边界符，匹配字符串开始	匹配开始是 123 和 1234： echo -e "1\n12\n123\n1234" |grep '\<123'
\>	边界符，匹配字符串结束	匹配结束是 1234： echo -e "1\n12\n123\n1234" |grep '4\>'
( )	单元或组合：将小括号里面作为一个组合 分组：匹配小括号中正则表达式或字符。\n 反向引用，n 是数字，从 1 开始编号，表示引用第 n 个分组匹配的内容	单元：匹配 123a 字符串 echo "123abc" |grep -E -o '([0-9a-z]){4}' 分组：匹配 11 echo "113abc" |grep -E -o '(1)\1' 匹配出现 xo 出现零次或多次： echo -e "x\nxo\nxoo\no\noo" |egrep "(xo)*"
|	匹配竖杠两边的任意一个	匹配 12 和 123： echo -e "1\n12\n123\n1234" |grep -E '12\>|123\>'
\	转义符，将特殊符号转成原有意义	1.2，匹配 1.2：1\.2，否则 112 也会匹配到

Posix 字符	描述
[:alnum:]	等效[a-zA-Z0-9]
[:alpha:]	等效[a-zA-Z]

[:lower:]	等效[a-z]
[:upper:]	等效[A-Z]
[:digit:]	等效[0-9]
[:space:]	匹配任意空白字符，等效[\t\n\r\f\v]
[:graph:]	非空白字符
[:blank:]	空格与定位字符
[:cntrl:]	控制字符
[:print:]	可显示的字符
[:punct:]	标点符号字符
[:xdigit:]	十六进制

示例：

```
echo -e "1\n12\n123\n1234a" |grep '[[:digit:]]'
```

在 Shell 下使用这些正则表达式处理文本最多的命令有下面几个工具：

命令	描述
grep	默认不支持扩展表达式，加-E 选项开启 ERE。如果不加-E 使用花括号要加转义符\{\}
egrep	支持基础和扩展表达式
awk	支持 egrep 所有的正则表达式
sed	默认不支持扩展表达式，加-r 选项开启 ERE。如果不加-r 使用花括号要加转义符\{\}

支持的特殊字符	描述
\w	匹配任意数字和字母，等效[a-zA-Z0-9_]
\W	与\w 相反，等效[^a-zA-Z0-9_]
\b	匹配字符串开始或结束，等效\<和\>

\s	匹配任意的空白字符
\S	匹配非空白字符

空白符	描述
\n	换行符
\r	回车符
\t	水平制表符
\v	垂直制表符
\0	空值符
\b	退后一格

第七章　Shell 文本处理三剑客

7.1 grep

过滤来自一个文件或标准输入匹配模式内容。
除了 grep 外，还有 egrep、fgrep。egrep 是 grep 的扩展，相当于 grep -E。fgrep 相当于 grep -f，用的少。
Usage: grep [OPTION]... PATTERN [FILE]...

支持的正则	描述
-E，--extended-regexp	模式是扩展正则表达式（ERE）
-P，--perl-regexp	模式是 Perl 正则表达式。 与 Shell 正则字符使用方式一样，这里不过多讲解
-e，--regexp=PATTERN	使用模式匹配，可指定多个模式匹配
-f，--file=FILE	从文件每一行获取匹配模式
-i，--ignore-case	忽略大小写
-w，--word-regexp	模式匹配整个单词
-x，--line-regexp	模式匹配整行

-v，--invert-match	打印不匹配的行

输出控制	描述
-m，--max-count=NUM	输出匹配的结果 num 数
-n，--line-number	打印行号
-H，--with-filename	打印每个匹配的文件名
-h，--no-filename	不输出文件名
-o，--only-matching	只打印匹配的内容
-q，--quiet	不输出正常信息
-s，--no-messages	不输出错误信息
-r，--recursive --include=FILE_PATTERN --exclude=FILE_PATTERN --exclude-from=FILE --exclude-dir=PATTERN	递归目录 只搜索匹配的文件 跳过匹配的文件 跳过匹配的文件，来自文件模式 跳过匹配的目录
-c，--count	只打印每个文件匹配的行数

内容行控制	描述
-B，--before-context=NUM	打印匹配的前几行
-A，--after-context=NUM	打印匹配的后几行
-C，--context=NUM	打印匹配的前后几行
--color[=WHEN],	匹配的字体颜色

示例：

1）输出 b 文件中在 a 文件相同的行

```
# grep -f a b
```

2）输出 b 文件中在 a 文件不同的行

```
# grep -v -f a b
```

3） 匹配多个模式

```
# echo ″a bc de″ |xargs -n1 |grep -e 'a' -e 'bc'
a
```

```
bc
```

4）去除空格 http.conf 文件空行或开头#号的行

```
# grep -E -v "^$|^#" /etc/httpd/conf/httpd.conf
```

5） 匹配开头不分大小写的单词

```
# echo "A a b c" |xargs -n1 |grep -i a
或
# echo "A a b c" |xargs -n1 |grep  '[Aa]'
A
a
```

6）只显示匹配的字符串

```
# echo "this is a test" |grep -o 'is'
is
is
```

7）输出匹配的前五个结果

```
# seq 1 20 |grep -m 5 -E '[0-9]{2}'
10
11
12
13
14
```

8）统计匹配多少行

```
# seq 1 20 |grep -c -E '[0-9]{2}'
11
```

9） 匹配 b 字符开头的行

```
# echo "a bc de" |xargs -n1 |grep '^b'
bc
```

10） 匹配 de 字符结尾的行并输出匹配的行

```
# echo "a ab abc abcd abcde" |xargs -n1 |grep -n 'de$'
5:abcde
```

11） 递归搜索/etc 目录下包含 ip 的 conf 后缀文件

```
# grep -r '192.167.1.1' /etc --include *.conf
```

12） 排除搜索 bak 后缀的文件

```
# grep -r '192.167.1.1' /opt --exclude *.bak
```

13） 排除来自 file 中的文件

```
# grep -r '192.167.1.1' /opt --exclude-from file
```

14） 匹配 41 或 42 的数字

```
# seq 41 45 |grep -E '4[12]'
41
42
```

15） 匹配至少 2 个字符

```
# seq 13 |grep -E '[0-9]{2}'
10
11
12
13
```

16） 匹配至少 2 个字符的单词，最多 3 个字符的单词

```
# echo "a ab abc abcd abcde" |xargs -n1 |grep -E -w -o '[a-z]{2,3}'
ab
abc
```

17） 匹配所有 IP

```
# ifconfig |grep -E -o "[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}"
```

18） 打印匹配结果及后 3 行

```
# seq 1 10 |grep 5 -A 3
5
6
7
8
```

19） 打印匹配结果及前 3 行

```
# seq 1 10 |grep 5 -B 3
2
3
4
5
```

20） 打印匹配结果及前后 3 行

```
# seq 1 10 |grep 5 -C 3
2
3
4
5
6
7
8
```

21） 不显示输出

```
不显示错误输出：
# grep 'a' abc
grep: abc: No such file or directory
```

```
# grep -s 'a' abc
# echo $?
2
不显示正常输出：
# grep -q 'a' a.txt
```

grep 支持上一章的基础和扩展正则表达式字符。

7.2 sed

流编辑器，过滤和替换文本。
工作原理：sed 命令将当前处理的行读入模式空间进行处理，处理完把结果输出，并清空模式空间。然后再将下一行读入模式空间进行处理输出，以此类推，直到最后一行。还有一个空间叫保持空间，又称暂存空间，可以暂时存放一些处理的数据，但不能直接输出，只能放到模式空间输出。这两个空间其实就是在内存中初始化的一个内存区域，存放正在处理的数据和临时存放的数据。
Usage:
sed [OPTION]... {script-only-if-no-other-script} [input-file]...
sed [选项] '地址 命令' file

选项	描述
-n	不打印模式空间
-e	执行脚本、表达式来处理
-f	执行动作从文件读取执行
-i	修改原文件
-r	使用扩展正则表达式

命令	描述
s/regexp/replacement/	替换字符串
p	打印当前模式空间
P	打印模式空间的第一行
d	删除模式空间，开始下一个循环
D	删除模式空间的第一行，开始下一个循环
=	打印当前行号
a \text	当前行追加文本

i \text	当前行上面插入文本
c \text	所选行替换新文本
q	立即退出 sed 脚本
r	追加文本来自文件
: label	label 为 b 和 t 命令
b label	分支到脚本中带有标签的位置，如果分支不存在则分支到脚本的末尾
t label	如果 s///是一个成功的替换，才跳转到标签
h H	复制/追加模式空间到保持空间
g G	复制/追加保持空间到模式空间
x	交换模式空间和保持空间内容
l	打印模式空间的行，并显示控制字符$
n N	读取/追加下一行输入到模式空间
w filename	写入当前模式空间到文件
!	取反、否定
&	引用已匹配字符串

地址	描述
first~step	步长，每 step 行，从第 first 开始
$	匹配最后一行
/regexp/	正则表达式匹配行
number	只匹配指定行
addr1,addr2	开始匹配 addr1 行开始，直接 addr2 行结束
addr1,+N	从 addr1 行开始，向后的 N 行
addr1,~N	从 addr1 行开始，到 N 行结束

借助以下文本内容作为示例讲解：

```
# tail /etc/services
nimgtw              48003/udp                   # Nimbus Gateway
3gpp-cbsp           48049/tcp                   # 3GPP Cell Broadcast Service Protocol
isnetserv           48128/tcp                   # Image Systems Network Services
isnetserv           48128/udp                   # Image Systems Network Services
blp5                48129/tcp                   # Bloomberg locator
blp5                48129/udp                   # Bloomberg locator
com-bardac-dw       48556/tcp                   # com-bardac-dw
com-bardac-dw       48556/udp                   # com-bardac-dw
iqobject            48619/tcp                   # iqobject
iqobject            48619/udp                   # iqobject
```

7.2.1 匹配打印（p）

1）打印匹配 blp5 开头的行

```
# tail /etc/services |sed -n '/^blp5/p'
blp5                 48129/tcp                       # Bloomberg locator
blp5                 48129/udp                       # Bloomberg locator
```

2）打印第一行

```
# tail /etc/services |sed -n '1p'
nimgtw               48003/udp                       # Nimbus Gateway
```

3）打印第一行至第三行

```
# tail /etc/services |sed -n '1,3p'
nimgtw              48003/udp                   # Nimbus Gateway
3gpp-cbsp           48049/tcp                   # 3GPP Cell Broadcast Service Protocol
isnetserv           48128/tcp                   # Image Systems Network Services
```

4）打印奇数行

```
# seq 10 |sed -n '1~2p'
1
3
5
7
9
```

5）打印匹配行及后一行

```
# tail /etc/services |sed -n '/blp5/,+1p'
blp5                 48129/tcp                       # Bloomberg locator
blp5                 48129/udp                       # Bloomberg locator
```

6）打印最后一行

```
# tail /etc/services |sed -n '$p'
iqobject             48619/udp                       # iqobject
```

7）不打印最后一行

```
# tail /etc/services |sed -n '$!p'
3gpp-cbsp             48049/tcp                       # 3GPP Cell Broadcast Service
Protocol
isnetserv             48128/tcp                       # Image Systems Network Services
isnetserv             48128/udp                       # Image Systems Network Services
blp5                  48129/tcp                       # Bloomberg locator
blp5                  48129/udp                       # Bloomberg locator
com-bardac-dw         48556/tcp                       # com-bardac-dw
com-bardac-dw         48556/udp                       # com-bardac-dw
iqobject              48619/tcp                       # iqobject
iqobject              48619/udp                       # iqobject
```

感叹号也就是对后面的命令取反。

8）匹配范围

```
# tail /etc/services   |sed -n '/^blp5/,/^com/p'
blp5                  48129/tcp                        # Bloomberg locator
blp5                  48129/udp                        # Bloomberg locator
com-bardac-dw         48556/tcp                        # com-bardac-dw
```

匹配开头行到最后一行：

```
# tail /etc/services |sed -n '/blp5/,$p'
blp5                  48129/tcp                        # Bloomberg locator
blp5                  48129/udp                        # Bloomberg locator
com-bardac-dw         48556/tcp                        # com-bardac-dw
com-bardac-dw         48556/udp                        # com-bardac-dw
iqobject              48619/tcp                        # iqobject
iqobject              48619/udp                        # iqobject
```

以逗号分开两个样式选择某个范围。

9）引用系统变量，用引号

```
# a=1
# tail /etc/services |sed -n ''$a',3p'
或
# tail /etc/services |sed -n "$a,3p"
```

sed 命令用单引号时，里面变量用单引号引起来，或者 sed 命令用双引号，因为双引号解释特殊符号原有意义。

7.2.2 匹配删除（d）

删除与打印使用方法类似，简单举几个例子。

```
# tail /etc/services |sed '/blp5/d'
nimgtw                48003/udp                       # Nimbus Gateway
3gpp-cbsp             48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv             48128/tcp                       # Image Systems Network Services
isnetserv             48128/udp                       # Image Systems Network Services
```

```
com-bardac-dw         48556/tcp                       # com-bardac-dw
com-bardac-dw         48556/udp                       # com-bardac-dw
iqobject              48619/tcp                       # iqobject
iqobject              48619/udp                       # iqobject
# tail /etc/services |sed '1d'
3gpp-cbsp             48049/tcp                       # 3GPP Cell Broadcast Service
Protocol
isnetserv             48128/tcp                       # Image Systems Network Services
isnetserv             48128/udp                       # Image Systems Network Services
blp5                  48129/tcp                       # Bloomberg locator
blp5                  48129/udp                       # Bloomberg locator
com-bardac-dw         48556/tcp                       # com-bardac-dw
com-bardac-dw         48556/udp                       # com-bardac-dw
iqobject              48619/tcp                       # iqobject
iqobject              48619/udp                       # iqobject
# tail /etc/services |sed '1~2d'
3gpp-cbsp             48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv             48128/udp                       # Image Systems Network Services
blp5                  48129/udp                       # Bloomberg locator
com-bardac-dw         48556/udp                       # com-bardac-dw
iqobject              48619/udp                       # iqobject
# tail /etc/services |sed '1,3d'
isnetserv             48128/udp                       # Image Systems Network Services
blp5                  48129/tcp                       # Bloomberg locator
blp5                  48129/udp                       # Bloomberg locator
com-bardac-dw         48556/tcp                       # com-bardac-dw
com-bardac-dw         48556/udp                       # com-bardac-dw
iqobject              48619/tcp                       # iqobject
iqobject              48619/udp                       # iqobject
```

去除空格 http.conf 文件空行或开头#号的行：

```
# sed '/^#/d;/^$/d' /etc/httpd/conf/httpd.conf
```

打印是把匹配的打印出来，删除是把匹配的删除，删除只是不用-n 选项。

7.2.3 替换（s///）

1）替换 blp5 字符串为 test

```
# tail /etc/services |sed 's/blp5/test/'
3gpp-cbsp           48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
isnetserv           48128/udp                       # Image Systems Network Services
test                48129/tcp                       # Bloomberg locator
test                48129/udp                       # Bloomberg locator
com-bardac-dw       48556/tcp                       # com-bardac-dw
com-bardac-dw       48556/udp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
```

```
iqobject            48619/udp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```

全局替换加 g：

```
# tail /etc/services |sed 's/blp5/test/g'
```

2）替换开头是 blp5 的字符串并打印

```
# tail /etc/services |sed -n 's/^blp5/test/p'
test                48129/tcp                       # Bloomberg locator
test                48129/udp                       # Bloomberg locator
```

3）使用&命令引用匹配内容并替换

```
# tail /etc/services |sed 's/48049/&.0/'
3gpp-cbsp           48049.0/tcp                     # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
isnetserv           48128/udp                       # Image Systems Network Services
blp5                48129/tcp                       # Bloomberg locator
blp5                48129/udp                       # Bloomberg locator
com-bardac-dw       48556/tcp                       # com-bardac-dw
com-bardac-dw       48556/udp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
iqobject            48619/udp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```

IP 加单引号：

```
# echo '10.10.10.1 10.10.10.2 10.10.10.3' |sed -r 's/[^ ]+/"&"/g'
"10.10.10.1" "10.10.10.2" "10.10.10.3"
```

4）对 1-4 行的 blp5 进行替换

```
# tail /etc/services | sed '1,4s/blp5/test/'
3gpp-cbsp           48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
isnetserv           48128/udp                       # Image Systems Network Services
test                48129/tcp                       # Bloomberg locator
blp5                48129/udp                       # Bloomberg locator
com-bardac-dw       48556/tcp                       # com-bardac-dw
com-bardac-dw       48556/udp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
iqobject            48619/udp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```

5）对匹配行进行替换

```
# tail /etc/services | sed '/48129\/tcp/s/blp5/test/'
3gpp-cbsp           48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
isnetserv           48128/udp                       # Image Systems Network Services
test                48129/tcp                       # Bloomberg locator
blp5                48129/udp                       # Bloomberg locator
com-bardac-dw       48556/tcp                       # com-bardac-dw
```

```
com-bardac-dw       48556/udp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
iqobject            48619/udp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```

6）二次匹配替换

```
# tail /etc/services   |sed 's/blp5/test/;s/3g/4g/'
4gpp-cbsp           48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
isnetserv           48128/udp                       # Image Systems Network Services
test                48129/tcp                       # Bloomberg locator
test                48129/udp                       # Bloomberg locator
com-bardac-dw       48556/tcp                       # com-bardac-dw
com-bardac-dw       48556/udp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
iqobject            48619/udp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```

7）分组使用，在每个字符串后面添加 123

```
# tail /etc/services |sed -r 's/(.*) (.*)(#.*)/\1\2test \3/'
3gpp-cbsp           48049/tcp                     test # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                     test # Image Systems Network Services
isnetserv           48128/udp                     test # Image Systems Network Services
blp5                48129/tcp                     test # Bloomberg locator
blp5                48129/udp                     test # Bloomberg locator
com-bardac-dw       48556/tcp                     test # com-bardac-dw
com-bardac-dw       48556/udp                     test # com-bardac-dw
iqobject            48619/tcp                     test # iqobject
iqobject            48619/udp                     test # iqobject
matahari            49000/tcp                     test # Matahari Broker
```

第一列是第一个小括号匹配，第二列第二个小括号匹配，第三列一样。将不变的字符串匹配分组，再通过\数字按分组顺序反向引用。

8）将协议与端口号位置调换

```
# tail /etc/services |sed -r 's/(.*)(\<[0-9]+\>)\/(tcp|udp)(.*)/\1\3\/\2\4/'
3gpp-cbsp           tcp/48049                       # 3GPP Cell Broadcast Service
isnetserv           tcp/48128                       # Image Systems Network Services
isnetserv           udp/48128                       # Image Systems Network Services
blp5                tcp/48129                       # Bloomberg locator
blp5                udp/48129                       # Bloomberg locator
com-bardac-dw       tcp/48556                       # com-bardac-dw
com-bardac-dw       udp/48556                       # com-bardac-dw
iqobject            tcp/48619                       # iqobject
iqobject            udp/48619                       # iqobject
matahari            tcp/49000                       # Matahari Broker
```

9）位置调换

```
# echo "abc:cde;123:456" |sed -r 's/([^:]+)(;.*:)([^:]+$)/\3\2\1/'
```

```
abc:456;123:cde
```

10）注释匹配行后的多少行

```
# seq 10 |sed '/5/,+3s/^/#/'
1
2
3
4
#5
#6
#7
#8
9
10
```

11）去除开头和结尾空格或制表符

```
# echo "   1 2 3   " |sed 's/^[ \t]*//;s/[ \t]*$//'
1 2 3
```

7.2.4 多重编辑（-e）

```
# tail /etc/services |sed -e '1,2d' -e 's/blp5/test/'
isnetserv          48128/udp                       # Image Systems Network Services
test               48129/tcp                       # Bloomberg locator
test               48129/udp                       # Bloomberg locator
com-bardac-dw      48556/tcp                       # com-bardac-dw
com-bardac-dw      48556/udp                       # com-bardac-dw
iqobject           48619/tcp                       # iqobject
iqobject           48619/udp                       # iqobject
matahari           49000/tcp                       # Matahari Broker
```

也可以使用分号分隔：

```
# tail /etc/services |sed '1,2d;s/blp5/test/'
```

7.2.5 添加新内容（a、i 和 c）

1）在 blp5 上一行添加 test

```
# tail /etc/services |sed '/blp5/i \test'
3gpp-cbsp          48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv          48128/tcp                       # Image Systems Network Services
isnetserv          48128/udp                       # Image Systems Network Services
test
blp5               48129/tcp                       # Bloomberg locator
test
blp5               48129/udp                       # Bloomberg locator
```

```
com-bardac-dw       48556/tcp                       # com-bardac-dw
com-bardac-dw       48556/udp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
iqobject            48619/udp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```

2）在 blp5 下一行添加 test

```
# tail /etc/services |sed '/blp5/a \test'
3gpp-cbsp           48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
isnetserv           48128/udp                       # Image Systems Network Services
blp5                48129/tcp                       # Bloomberg locator
test
blp5                48129/udp                       # Bloomberg locator
test
com-bardac-dw       48556/tcp                       # com-bardac-dw
com-bardac-dw       48556/udp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
iqobject            48619/udp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```

3）将 blp5 替换新行

```
# tail /etc/services |sed '/blp5/c \test'
3gpp-cbsp           48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
isnetserv           48128/udp                       # Image Systems Network Services
test
test
com-bardac-dw       48556/tcp                       # com-bardac-dw
com-bardac-dw       48556/udp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
iqobject            48619/udp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```

4）在指定行下一行添加一行

```
# tail /etc/services |sed '2a \test'
3gpp-cbsp           48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
test
isnetserv           48128/udp                       # Image Systems Network Services
blp5                48129/tcp                       # Bloomberg locator
blp5                48129/udp                       # Bloomberg locator
com-bardac-dw       48556/tcp                       # com-bardac-dw
com-bardac-dw       48556/udp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
iqobject            48619/udp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```

5）在指定行前面和后面添加一行

```
# seq 5 |sed '3s/.*/txt\n&/'
1
2
txt
3
4
5
# seq 5 |sed '3s/.*/&\ntxt/'
1
2
3
txt
4
5
```

7.2.6 读取文件并追加到匹配行后（r）

```
# cat a.txt
123
456
# tail /etc/services |sed '/blp5/r a.txt'
3gpp-cbsp           48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
isnetserv           48128/udp                       # Image Systems Network Services
blp5                48129/tcp                       # Bloomberg locator
123
456
blp5                48129/udp                       # Bloomberg locator
123
456
com-bardac-dw       48556/tcp                       # com-bardac-dw
com-bardac-dw       48556/udp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
iqobject            48619/udp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```

7.2.7 将匹配行写到文件（w）

```
# tail /etc/services |sed '/blp5/w b.txt'
3gpp-cbsp           48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
isnetserv           48128/udp                       # Image Systems Network Services
blp5                48129/tcp                       # Bloomberg locator
blp5                48129/udp                       # Bloomberg locator
```

```
com-bardac-dw       48556/tcp                       # com-bardac-dw
com-bardac-dw       48556/udp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
iqobject            48619/udp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
# cat b.txt
blp5                48129/tcp                       # Bloomberg locator
blp5                48129/udp                       # Bloomberg locator
```

7.2.8 读取下一行（n 和 N）

n 命令的作用是读取下一行到模式空间。
N 命令的作用是追加下一行内容到模式空间，并以换行符\n 分隔。
1）打印匹配的下一行

```
# seq 5 |sed -n '/3/{n;p}'
4
```

2）打印偶数

```
# seq 6 |sed -n 'n;p'
2
4
6
```

sed 先读取第一行 1，执行 n 命令，获取下一行 2，此时模式空间是 2，执行 p 命令，打印模式空间。 现在模式空间是 2，sed 再读取 3，执行 n 命令，获取下一行 4，此时模式空间为 4，执行 p 命令，以此类推。
3）打印奇数

```
# seq 6 |sed 'n;d'
1
3
5
```

sed 先读取第一行 1，此时模式空间是 1，并打印模式空间 1，执行 n 命令，获取下一行 2，执行 d 命令，删除模式空间的 2，sed 再读取 3，此时模式空间是 3，并打印模式空间，再执行 n 命令，获取下一行 4，执行 d 命令，删除模式空间的 3，以此类推。

```
# seq 6 |sed -n 'p;n'
1
3
5
```

4）每三行执行一次 p 命令

```
# seq 6 |sed 'n;n;p'
1
2
3
```

```
3
4
5
6
6
```

sed 先读取第一行 1，并打印模式空间 1，执行 n 命令，获取下一行 2，并打印模式空间 2，再执行 n 命令，获取下一行 3，执行 p 命令，打印模式空间 3。sed 读取下一行 3，并打印模式空间 3,以此类推。

5）每三行替换一次

方法 1：

```
# seq 6 |sed 'n;n;s/^/=/;s/$/=/'
1
2
=3=
4
5
=6=
```

我们只是把 p 命令改成了替换命令。

方法 2：

这次用到了地址匹配，来实现上面的效果：

```
# seq 6 |sed '3~3{s/^/=/;s/$/=/}'
1
2
=3=
4
5
=6=
```

当执行多个 sed 命令时，有时相互会产生影响，我们可以用大括号{}把他们括起来。

6）再看下 N 命令的功能

```
# seq 6 |sed 'N;q'
1
2
```

将两行合并一行：

```
# seq 6 |sed 'N;s/\n//'
12
34
56
```

第一个命令：sed 读取第一行 1，N 命令读取下一行 2，并以\n2 追加，此时模式空间是 1\n2，再执行 q 退出。

为了进一步说明 N 的功能，看第二个命令：执行 N 命令后，此时模式空间是 1\n2，再执行把\n 替换为空，此时模式空间是 12，并打印。

```
# seq 5 |sed -n 'N;p'
1
2
```

```
3
4
# seq 6 |sed -n 'N;p'
1
2
3
4
5
6
```

为什么第一个不打印 5 呢？
因为 N 命令是读取下一行追加到 sed 读取的当前行，当 N 读取下一行没有内容时，则退出，也不会执行 p 命令打印当前行。
当行数为偶数时，N 始终就能读到下一行，所以也会执行 p 命令。
7）打印奇数行数时的最后一行

```
# seq 5 |sed -n '$!N;p'
1
2
3
4
5
```

加一个满足条件，当 sed 执行到最后一行时，用感叹号不去执行 N 命令，随后执行 p 命令。

7.2.9 打印和删除模式空间第一行（P 和 D）

P 命令作用是打印模式空间的第一行。
D 命令作用是删除模式空间的第一行。
1）打印奇数

```
# seq 6 |sed -n 'N;P'
1
3
5
```

2）保留最后一行

```
# seq 6 |sed 'N;D'
6
```

读取第一行 1，执行 N 命令读取下一行并追加到模式空间，此时模式空间是 1\n2，执行 D 命令删除模式空间第一行 1，剩余 2。
读取第二行，执行 N 命令，此时模式空间是 3\n4，执行 D 命令删除模式空间第一行 3，剩余 4。
以此类推，读取最后一行打印时，而 N 获取不到下一行则退出，不再执行 D，因此模式空间只剩余 6 就打印。

7.2.10 保持空间操作（h 与 H、g 与 G 和 x）

h 命令作用是复制模式空间内容到保持空间（覆盖）。
H 命令作用是复制模式空间内容追加到保持空间。

g 命令作用是复制保持空间内容到模式空间（覆盖）。
G 命令作用是复制保持空间内容追加到模式空间。
x 命令作用是模式空间与保持空间内容互换
1）将匹配的内容覆盖到另一个匹配

```
# seq 6 |sed -e '/3/{h;d}' -e '/5/g'
1
2
4
3
6
```

h 命令把匹配的 3 复制到保持空间，d 命令删除模式空间的 3。后面命令再对模式空间匹配 5，并用 g 命令把保持空间 3 覆盖模式空间 5。
2）将匹配的内容放到最后

```
# seq 6 |sed -e '/3/{h;d}' -e '$G'
1
2
4
5
6
3
```

3）交换模式空间和保持空间

```
# seq 6 |sed -e '/3/{h;d}' -e '/5/x' -e '$G'
1
2
4
3
6
5
```

看后面命令，在模式空间匹配 5 并将保持空间的 3 与 5 交换，5 就变成了 3,。最后把保持空间的 5 追加到模式空间的。
4）倒叙输出

```
# seq 5 |sed '1!G;h;$!d'
5
4
3
2
1
```

分析下：
1!G 第一行不执行把保持空间内容追加到模式空间，因为现在保持空间还没有数据。
h 将模式空间放到保持空间暂存。
$!d 最后一行不执行删除模式空间的内容。
读取第一行 1 时，跳过 G 命令，执行 h 命令将模式空间 1 复制到保持空间，执行 d 命令删除模式空间的 1。

读取第二行 2 时，模式空间是 2，执行 G 命令，将保持空间 1 追加到模式空间，此时模式空间是 2\n1，执行 h 命令将 2\n1 覆盖到保持空间，d 删除模式空间。
读取第三行 3 时，模式空间是 3，执行 G 命令，将保持空间 2\n1 追加到模式空间，此时模式空间是 3\n2\n1，执行 h 命令将模式空间内容复制到保持空间，d 删除模式空间。
以此类推，读到第 5 行时，模式空间是 5，执行 G 命令，将保持空间的 4\n3\n2\n1 追加模式空间，然后复制到模式空间，5\n4\n3\n2\n1，不执行 d，模式空间保留，输出。
由此可见，每次读取的行先放到模式空间，再复制到保持空间，d 命令删除模式空间内容，防止输出，再追加到模式空间，因为追加到模式空间，会追加到新读取的一行的后面，循环这样操作， 就把所有行一行行追加到新读取行的后面，就形成了倒叙。
5）每行后面添加新空行

```
# seq 10 |sed G
1

2

3

4

5
```

6）打印匹配行的上一行内容

```
# seq 5 |sed -n '/3/{x;p};h'
2
```

读取第一行 1，没有匹配到 3，不执行{x;p}，执行 h 命令将模式空间内容 1 覆盖到保持空间。
读取第二行 2，没有匹配到 3，不执行{x;p}，执行 h 命令将模式空间内容 2 覆盖到保持空间。
读取第三行 3，匹配到 3，执行 x 命令把模式空间 3 与保持空间 2 交换，再执行 p 打印模式空间 2.
以此类推。
7）打印匹配行到最后一行或下一行到最后一行

```
# seq 5 |sed -n '/3/,$p'
3
4
5
# seq 5 |sed -n '/3/,${h;x;p}'
3
4
5
# seq 5 |sed -n '/3/{:a;N;$!ba;p}'
3
4
5
# seq 5 |sed -n '/3/{n;:a;N;$!ba;p}'
4
5
```

匹配到 3 时，n 读取下一行 4，此时模式空间是 4，执行 N 命令读取下一行并追加到模式空间，此时模式空间是 4\n5，标签循环完成后打印模式空间 4\n5。

7.2.11 标签（:、b 和 t）

标签可以控制流，实现分支判断。
: lable name　定义标签
b lable　跳转到指定标签，如果没有标签则到脚本末尾
t lable　跳转到指定标签，前提是 s///命令执行成功
1）将换行符替换成逗号
方法 1：

```
# seq 6 |sed 'N;s/\n/,/'
1,2
3,4
5,6
```

这种方式并不能满足我们的需求，每次 sed 读取到模式空间再打印是新行，替换\n 也只能对 N 命令追加后的 1\n2 这样替换。
这时就可以用到标签了：

```
# seq 6 |sed ':a;N;s/\n/,/;b a'
1,2,3,4,5,6
```

看看这里的标签使用，:a 是定义的标签名，b a 是跳转到 a 位置。
sed 读取第一行 1，N 命令读取下一行 2，此时模式空间是 1\n2$，执行替换，此时模式空间是 1,2$，执行 b 命令再跳转到标签 a 位置继续执行 N 命令，读取下一行 3 追加到模式空间，此时模式空间是 1,2\n3$，再替换，以此类推，不断追加替换，直到最后一行 N 读不到下一行内容退出。
方法 2：

```
# seq 6 |sed ':a;N;$!b a;s/\n/,/g'
1,2,3,4,5,6
```

先将每行读入到模式空间，最后再执行全局替换。$!是如果是最后一行，则不执行 b a 跳转，最后执行全局替换。

```
# seq 6 |sed  ':a;N;b a;s/\n/,/g'
1
2
3
4
5
6
```

可以看到，不加$!是没有替换，因为循环到 N 命令没有读到行就退出了，后面的替换也就没执行。
2）每三个数字加个一个逗号

```
# echo "123456789" |sed -r 's/([0-9]+)([0-9]+{3})/\1,\2/'
123456,789
# echo "123456789" |sed -r ':a;s/([0-9]+)([0-9]+{3})/\1,\2/;t a'
123,456,789
# echo "123456789" |sed -r ':a;s/([0-9]+)([0-9]+{2})/\1,\2/;t a'
1,23,45,67,89
```

执行第一次时，替换最后一个，跳转后，再对 123456 匹配替换，直到匹配替换不成功，不执行 t 命令。

7.2.12 忽略大小写匹配（I）

```
# echo -e "a\nA\nb\nc" |sed 's/a/1/Ig'
1
1
b
c
```

7.2.13 获取总行数（#）

```
# seq 10 |sed -n '$='
```

8.3 awk

awk 是一个处理文本的编程语言工具，能用简短的程序处理标准输入或文件、数据排序、计算以及生成报表等等。
在 Linux 系统下默认 awk 是 gawk，它是 awk 的 GNU 版本。可以通过命令查看应用的版本：ls -l /bin/awk
基本的命令语法：awk option 'pattern {action}' file
其中 pattern 表示 AWK 在数据中查找的内容，而 action 是在找到匹配内容时所执行的一系列命令。花括号用于根据特定的模式对一系列指令进行分组。
awk 处理的工作方式与数据库类似，支持对记录和字段处理，这也是 grep 和 sed 不能实现的。
在 awk 中，缺省的情况下将文本文件中的一行视为一个记录，逐行放到内存中处理，而将一行中的某一部分作为记录中的一个字段。用 1,2,3...数字的方式顺序的表示行（记录）中的不同字段。用 $后跟数字，引用对应的字段，以逗号分隔，0 表示整个行。

8.3.1 选项

选项	描述
-f program-file	从文件中读取 awk 程序源文件
-F fs	指定 fs 为输入字段分隔符
-v var=value	变量赋值
--posix	兼容 POSIX 正则表达式
--dump-variables=[file]	把 awk 命令时的全局变量写入文件， 默认文件是 awkvars.out
--profile=[file]	格式化 awk 语句到文件，默认是 awkprof.out

8.3.2 模式

常用模式有：

Pattern	Description
BEGIN{ }	给程序赋予初始状态，先执行的工作
END{ }	程序结束之后执行的一些扫尾工作
/regular expression/	为每个输入记录匹配正则表达式
pattern && pattern	逻辑 and，满足两个模式
pattern || pattern	逻辑 or，满足其中一个模式
! pattern	逻辑 not，不满足模式
pattern1, pattern2	范围模式，匹配所有模式 1 的记录，直到匹配到模式 2

而动作呢，就是下面所讲的 print、流程控制、I/O 语句等。

示例：

1）从文件读取 awk 程序处理文件

```
# vi test.awk
{print $2}
# tail -n3 /etc/services |awk -f test.awk
48049/tcp
48128/tcp
49000/tcp
```

2）指定分隔符，打印指定字段

```
打印第二字段，默认以空格分隔：
# tail -n3 /etc/services |awk '{print $2}'
48049/tcp
48128/tcp
48128/udp
指定冒号为分隔符打印第一字段：
# awk -F ':' '{print $1}' /etc/passwd
root
bin
daemon
adm
lp
sync
......
```

还可以指定多个分隔符，作为同一个分隔符处理：

```
# tail -n3 /etc/services |awk -F'[/#]' '{print $3}'
```

```
  iqobject
  iqobject
  Matahari Broker
# tail -n3 /etc/services |awk -F'[/#]' '{print $1}'
iqobject            48619
iqobject            48619
matahari            49000
# tail -n3 /etc/services |awk -F'[/#]' '{print $2}'
tcp
udp
tcp
# tail -n3 /etc/services |awk -F'[/#]' '{print $3}'
  iqobject
  iqobject
  Matahari Broker
# tail -n3 /etc/services |awk -F'[ /]+' '{print $2}'
48619
48619
49000
```

[]元字符的意思是符号其中任意一个字符，也就是说每遇到一个/或#时就分隔一个字段，当用多个分隔符时，就能更方面处理字段了。

3）变量赋值

```
# awk -v a=123 'BEGIN{print a}'
123
```

系统变量作为 awk 变量的值：

```
# a=123
# awk -v a=$a 'BEGIN{print a}'
123
```

或使用单引号

```
# awk 'BEGIN{print '$a'}'
123
```

4）输出 awk 全局变量到文件

```
# seq 5 |awk --dump-variables '{print $0}'
1
2
3
4
5
# cat awkvars.out
ARGC: number (1)
ARGIND: number (0)
ARGV: array, 1 elements
BINMODE: number (0)
CONVFMT: string ("%.6g")
ERRNO: number (0)
FIELDWIDTHS: string ("")
```

```
FILENAME: string ("-")
FNR: number (5)
FS: string (" ")
IGNORECASE: number (0)
LINT: number (0)
NF: number (1)
NR: number (5)
OFMT: string ("%.6g")
OFS: string (" ")
ORS: string ("\n")
RLENGTH: number (0)
RS: string ("\n")
RSTART: number (0)
RT: string ("\n")
SUBSEP: string ("\034")
TEXTDOMAIN: string ("messages")
```

5）BEGIN 和 END

BEGIN 模式是在处理文件之前执行该操作，常用于修改内置变量、变量赋值和打印输出的页眉或标题。

例如：打印页眉

```
# tail /etc/services |awk 'BEGIN{print "Service\t\tPort\t\t\tDescription\n==="}{print $0}'
Service              Port                              Description
===
3gpp-cbsp           48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
isnetserv           48128/udp                       # Image Systems Network Services
blp5                48129/tcp                       # Bloomberg locator
blp5                48129/udp                       # Bloomberg locator
com-bardac-dw       48556/tcp                       # com-bardac-dw
com-bardac-dw       48556/udp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
iqobject            48619/udp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```

END 模式是在程序处理完才会执行。

例如：打印页尾

```
# tail /etc/services |awk '{print $0}END{print "===\nEND......"}'
3gpp-cbsp           48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
isnetserv           48128/udp                       # Image Systems Network Services
blp5                48129/tcp                       # Bloomberg locator
blp5                48129/udp                       # Bloomberg locator
com-bardac-dw       48556/tcp                       # com-bardac-dw
com-bardac-dw       48556/udp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
iqobject            48619/udp                       # iqobject
```

```
matahari            49000/tcp                       # Matahari Broker
===
END......
```

6）格式化输出 awk 命令到文件

```
# tail /etc/services |awk --profile 'BEGIN{print
"Service\t\tPort\t\t\tDescription\n==="}{print $0}END{print "===\nEND......"}'
Service         Port                    Description
===
nimgtw          48003/udp               # Nimbus Gateway
3gpp-cbsp       48049/tcp               # 3GPP Cell Broadcast Service Protocol
isnetserv       48128/tcp               # Image Systems Network Services
isnetserv       48128/udp               # Image Systems Network Services
blp5            48129/tcp               # Bloomberg locator
blp5            48129/udp               # Bloomberg locator
com-bardac-dw   48556/tcp               # com-bardac-dw
com-bardac-dw   48556/udp               # com-bardac-dw
iqobject        48619/tcp               # iqobject
iqobject        48619/udp               # iqobject
===
END......
# cat awkprof.out
        # gawk profile, created Sat Jan  7 19:45:22 2017

        # BEGIN block(s)

        BEGIN {
                print "Service\t\tPort\t\t\tDescription\n==="
        }

        # Rule(s)

        {
                print $0
        }

        # END block(s)

        END {
                print "===\nEND......"
        }
```

7）/re/正则匹配

```
匹配包含 tcp 的行：
# tail /etc/services |awk '/tcp/{print $0}'
3gpp-cbsp           48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
blp5                48129/tcp                       # Bloomberg locator
```

```
com-bardac-dw       48556/tcp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```
匹配开头是 blp5 的行：
```
# tail /etc/services |awk '/^blp5/{print $0}'
blp5                48129/tcp                       # Bloomberg locator
blp5                48129/udp                       # Bloomberg locator
```
匹配第一个字段是 8 个字符的行：
```
# tail /etc/services |awk '/^[a-z0-9]{8} /{print $0}'
iqobject            48619/tcp                       # iqobject
iqobject            48619/udp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```
如果没有匹配到，请查看你的 awk 版本（awk --version）是不是 3，因为 4 才支持{}

8）逻辑 and、or 和 not

匹配记录中包含 blp5 和 tcp 的行：
```
# tail /etc/services |awk '/blp5/ && /tcp/{print $0}'
blp5                48129/tcp                       # Bloomberg locator
```
匹配记录中包含 blp5 或 tcp 的行：
```
# tail /etc/services |awk '/blp5/ || /tcp/{print $0}'
3gpp-cbsp           48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
blp5                48129/tcp                       # Bloomberg locator
blp5                48129/udp                       # Bloomberg locator
com-bardac-dw       48556/tcp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```
不匹配开头是#和空行：
```
# awk '! /^#/ && ! /^$/{print $0}' /etc/httpd/conf/httpd.conf
```
或
```
# awk '! /^#|^$/' /etc/httpd/conf/httpd.conf
```
或
```
# awk '/^[^#]|"^$"/' /etc/httpd/conf/httpd.conf
```

9）匹配范围

```
# tail /etc/services |awk '/^blp5/,/^com/'
blp5                 48129/tcp                      # Bloomberg locator
blp5                 48129/udp                      # Bloomberg locator
com-bardac-dw        48556/tcp                      # com-bardac-dw
```
对匹配范围后记录再次处理，例如匹配关键字下一行到最后一行：
```
# seq 5 |awk '/3/,/^$/{printf /3/?"":$0"\n"}'
4
5
```
另一种判断真假的方式实现：
```
# seq 5 |awk '/3/{t=1;next}t'
4
5
```

1 和 2 都不匹配 3，不执行后面{}，执行 t，t 变量还没赋值，为空，空在 awk 中就为假，就不打印当前行。匹配到 3，执行 t=1，next 跳出，不执行 t。4 也不匹配 3，执行 t，t 的值上次赋值的 1，为真，打印当前行，以此类推。（非 0 的数字都为真，所以 t 可以写任意非 0 数字）
如果想打印匹配行都最后一行，就可以这样了：

```
# seq 5 |awk '/3/{t=1}t'
3
4
5
```

8.3.3 内置变量

变量名	描述
FS	输入字段分隔符，默认是空格或制表符
OFS	输出字段分隔符，默认是空格
RS	输入记录分隔符，默认是换行符\n
ORS	输出记录分隔符，默认是换行符\n
NF	统计当前记录中字段个数
NR	统计记录编号，每处理一行记录，编号就会+1
FNR	统计记录编号，每处理一行记录，编号也会+1，与 NR 不同的是，处理第二个文件时，编号会重新计数。
ARGC	命令行参数数量
ARGV	命令行参数数组序列数组，下标从 0 开始，ARGV[0]是 awk
ARGIND	当前正在处理的文件索引值。第一个文件是 1，第二个文件是 2，以此类推
ENVIRON	当前系统的环境变量
FILENAME	输出当前处理的文件名
IGNORECASE	忽略大小写
SUBSEP	数组中下标的分隔符，默认为"\034"

示例：
1）FS 和 OFS
在程序开始前重新赋值 FS 变量，改变默认分隔符为冒号，与-F 一样。

```
# awk 'BEGIN{FS=":"}{print $1,$2}' /etc/passwd |head -n5
root x
```

```
bin x
daemon x
adm x
lp x
```
也可以使用-v 来重新赋值这个变量：
```
# awk -vFS=':' '{print $1,$2}' /etc/passwd |head -n5          # 中间逗号被换成了 OFS 的默认值
root x
bin x
daemon x
adm x
lp x
```
由于 OFS 默认以空格分隔，反向引用多个字段分隔的也是空格，如果想指定输出分隔符这样：
```
# awk 'BEGIN{FS=":";OFS=":"}{print $1,$2}' /etc/passwd |head -n5
root:x
bin:x
daemon:x
adm:x
lp:x
```
也可以通过字符串拼接实现分隔：
```
# awk 'BEGIN{FS=":"}{print $1"#"$2}' /etc/passwd |head -n5
root#x
bin#x
daemon#x
adm#x
lp#x
```
2）RS 和 ORS

RS 默认是\n 分隔每行，如果想指定以某个字符作为分隔符来处理记录：
```
# echo "www.baidu.com/user/test.html" |awk 'BEGIN{RS="/"}{print $0}'
www.baidu.com
user
test.html

RS 也支持正则，简单演示下：
# seq -f "str%02g" 10 |sed 'n;n;a\-----' |awk 'BEGIN{RS="-+"}{print $1}'
str01
str04
str07
str10
```
将输出的换行符替换为+号：
```
# seq 10 |awk 'BEGIN{ORS="+"}{print $0}'
1+2+3+4+5+6+7+8+9+10+
```
替换某个字符：
```
# tail -n2 /etc/services |awk 'BEGIN{RS="/";ORS="#"}{print $0}'
iqobject            48619#udp                       # iqobject
matahari            49000#tcp                       # Matahari Broker
```
3）NF

NF 是字段个数。

```
# echo "a b c d e f" |awk '{print NF}'
6
打印最后一个字段：
# echo "a b c d e f" |awk '{print $NF}'
f
打印倒数第二个字段：
# echo "a b c d e f" |awk '{print $(NF-1)}'
e
排除最后两个字段：
# echo "a b c d e f" |awk '{$NF="";$(NF-1)="";print $0}'
a b c d
排除第一个字段：
# echo "a b c d e f" |awk '{$1="";print $0}'
  b c d e f
```

4）NR 和 FNR

NR 统计记录编号，每处理一行记录，编号就会+1，FNR 不同的是在统计第二个文件时会重新计数。

```
打印行数：
# tail -n5 /etc/services |awk '{print NR,$0}'
1 com-bardac-dw        48556/tcp                       # com-bardac-dw
2 com-bardac-dw        48556/udp                       # com-bardac-dw
3 iqobject             48619/tcp                       # iqobject
4 iqobject             48619/udp                       # iqobject
5 matahari             49000/tcp                       # Matahari Broker
打印总行数：
# tail -n5 /etc/services |awk 'END{print NR}'
5
打印第三行：
# tail -n5 /etc/services |awk 'NR==3'
iqobject             48619/tcp                       # iqobject
打印第三行第二个字段：
# tail -n5 /etc/services |awk 'NR==3{print $2}'
48619/tcp
打印前三行：
# tail -n5 /etc/services |awk 'NR<=3{print NR,$0}'
1 com-bardac-dw      48556/tcp                       # com-bardac-dw
2 com-bardac-dw      48556/udp                       # com-bardac-dw
3 iqobject           48619/tcp                       # iqobject
```

看下 NR 和 FNR 的区别：

```
# cat a
a
b
c
# cat b
c
d
```

```
e
# awk '{print NR,FNR,$0}' a b
1 1 a
2 2 b
3 3 c
4 1 c
5 2 d
6 3 e
```

可以看出 NR 每处理一行就会+1，而 FNR 在处理第二个文件时，编号重新计数。同时也知道 awk 处理两个文件时，是合并到一起处理。

```
# awk 'FNR==NR{print $0"1"}FNR!=NR{print $0"2"}' a b
a1
b1
c1
c2
d2
e2
```

当 FNR==NR 时，说明在处理第一个文件内容，不等于时说明在处理第二个文件内容。
一般 FNR 在处理多个文件时会用到，下面会讲解。
5）ARGC 和 ARGV
ARGC 是命令行参数数量
ARGV 是将命令行参数存到数组，元素由 ARGC 指定，数组下标从 0 开始

```
# awk 'BEGIN{print ARGC}' 1 2 3
4
# awk 'BEGIN{print ARGV[0]}'
awk
# awk 'BEGIN{print ARGV[1]}' 1 2
1
# awk 'BEGIN{print ARGV[2]}' 1 2
2
```

6）ARGIND
ARGIND 是当前正在处理的文件索引值，第一个文件是 1，第二个文件是 2，以此类推，从而可以通过这种方式判断正在处理哪个文件。

```
# awk '{print ARGIND,$0}' a b
1 a
1 b
1 c
2 c
2 d
2 e
# awk 'ARGIND==1{print "a->"$0}ARGIND==2{print "b->"$0}' a   b
a->a
a->b
a->c
b->c
```

```
b->d
b->e
```

7）ENVIRON
ENVIRON 调用系统变量。

```
# awk 'BEGIN{print ENVIRON["HOME"]}'
/root
```

如果是设置的环境变量，还需要用 export 导入到系统变量才可以调用：

```
# awk 'BEGIN{print ENVIRON["a"]}'

# export a
# awk 'BEGIN{print ENVIRON["a"]}'
123
```

8）FILENAME
FILENAME 是当前处理文件的文件名。

```
# awk 'FNR==NR{print FILENAME"->"$0}FNR!=NR{print FILENAME"->"$0}' a b
a->a
a->b
a->c
b->c
b->d
b->e
```

9）忽略大小写

```
# echo "A a b c" |xargs -n1 |awk 'BEGIN{IGNORECASE=1}/a/'
A
a
```

等于 1 代表忽略大小写。

8.3.4 操作符

运算符	描述
（....）	分组
$	字段引用
++ --	递增和递减
+ - ！	加号，减号，和逻辑否定
* / %	乘，除和取余
+ -	加法，减法
| |&	管道，用于 getline，print 和 printf

< > <= >= != ==	关系运算符
~ !~	正则表达式匹配，否定正则表达式匹配
in	数组成员
&& ||	逻辑 and，逻辑 or
?:	简写条件表达式： expr1 ? expr2 : expr3 第一个表达式为真，执行 expr2，否则执行 expr3
= += -= *= /= %= ^=	变量赋值运算符

须知：
在 awk 中，有 3 种情况表达式为假：数字是 0，空字符串和未定义的值。
数值运算，未定义变量初始值为 0。字符运算，未定义变量初始值为空。
举例测试：

```
# awk 'BEGIN{n=0;if(n)print "true";else print "false"}'
false
# awk 'BEGIN{s="";if(s)print "true";else print "false"}'
false
# awk 'BEGIN{if(s)print "true";else print "false"}'
false
```

示例：
1）截取整数

```
# echo "123abc abc123 123abc123" |xargs -n1 | awk '{print +$0}'
123
0
123
# echo "123abc abc123 123abc123" |xargs -n1 | awk '{print -$0}'
-123
0
-123
```

2）感叹号

```
打印奇数行：
# seq 6 |awk 'i=!i'
1
3
5
打印偶数行：
# seq 6 |awk '!(i=!i)'
2
4
6
```

读取第一行：i 是未定义变量，也就是 i=!0，!取反意思。感叹号右边是个布尔值，0 或空字符串为假，非 0 或非空字符串为真，!0 就是真，因此 i=1，条件为真打印当前记录。
没有 print 为什么会打印呢？因为模式后面没有动作，默认会打印整条记录。
读取第二行：因为上次 i 的值由 0 变成了 1，此时就是 i=!1，条件为假不打印。
读取第三行：上次条件又为假，i 恢复初始值 0，取反，继续打印。以此类推...
可以看出，运算时并没有判断行内容，而是利用布尔值真假判断输出当前行。

2）不匹配某行

```
# tail /etc/services |awk '!/blp5/{print $0}'
3gpp-cbsp           48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
isnetserv           48128/udp                       # Image Systems Network Services
com-bardac-dw       48556/tcp                       # com-bardac-dw
com-bardac-dw       48556/udp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
iqobject            48619/udp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```

3）乘法和除法

```
# seq 5 |awk '{print $0*2}'
2
4
6
8
10
# seq 5 |awk '{print $0%2}'
1
0
1
0
1
```

打印偶数行：

```
# seq 5 |awk '$0%2==0{print $0}'
2
4
```

打印奇数行：

```
# seq 5 |awk '$0%2!=0{print $0}'
1
3
5
```

4）管道符使用

```
# seq 5 |shuf |awk '{print $0|"sort"}'
1
2
3
4
```

```
5
```

5）正则表达式匹配

```
# seq 5 |awk '$0~3{print $0}'
3
# seq 5 |awk '$0!~3{print $0}'
1
2
4
5
# seq 5 |awk '$0~/[34]/{print $0}'
3
4
# seq 5 |awk '$0!~/[34]/{print $0}'
1
2
5
# seq 5 |awk '$0~/[^34]/{print $0}'
1
2
5
```

6）判断数组成员

```
# awk 'BEGIN{a["a"]=123}END{if("a" in a)print "yes"}' </dev/null
yes
```

7）三目运算符

```
# awk 'BEGIN{print 1==1?"yes":"no"}'   # 三目运算作为一个表达式，里面不允许写 print
yes
# seq 3 |awk '{print $0==2?"yes":"no"}'
no
yes
no
```

替换换行符为逗号：

```
# seq 5 |awk '{print n=(n?n",":$0)}'
1
1,2
1,2,3
1,2,3,4
1,2,3,4,5
# seq 5 |awk '{n=(n?n",":$0)}END{print n}'
1,2,3,4,5
```

说明：读取第一行时，n 没有变量，为假输出$0 也就是 1，并赋值变量 n，读取第二行时，n 是 1 为真，输出 1,2 以此类推，后面会一直为真。

每三行后面添加新一行：

```
# seq 10 |awk '{print NR%3?$0:$0 "\ntxt"}'
1
2
```

```
3
txt
4
5
6
txt
7
8
9
txt
10
在
两行合并一行：
# seq 6 |awk '{printf NR%2!=0?$0" ":$0" \n"}'
1 2
3 4
5 6
# seq 6 |awk 'ORS=NR%2?" ":"\n"'
1 2
3 4
5 6
# seq 6 |awk '{if(NR%2)ORS=" ";else ORS="\n";print}'
```

8）变量赋值

```
字段求和：
# seq 5 |awk '{sum+=1}END{print sum}'
5
# seq 5 |awk '{sum+=$0}END{print sum}'
15
```

8.3.5 流程控制

1）if 语句

格式：if (condition) statement [else statement]

```
单分支：
# seq 5 |awk '{if($0==3)print $0}'
3
双分支：
# seq 5 |awk '{if($0==3)print $0;else print "no"}'
no
no
3
no
no
多分支：
# cat file
1 2 3
```

```
4 5 6
7 8 9
# awk '{if($1==4){print "1"} else if($2==5){print "2"} else if($3==6){print "3"} else
{print "no"}}' file
no
1
no
```

2）while 语句

格式：while (condition) statement

```
遍历打印所有字段：
# awk '{i=1;while(i<=NF){print $i;i++}}' file
1
2
3
4
5
6
7
8
9
```

awk 是按行处理的，每次读取一行，并遍历打印每个字段。

3）for 语句 C 语言风格

格式：for (expr1; expr2; expr3) statement

```
遍历打印所有字段：
# cat file
1 2 3
4 5 6
7 8 9
# awk '{for(i=1;i<=NF;i++)print $i}' file
1
2
3
4
5
6
7
8
9
倒叙打印文本：
# awk '{for(i=NF;i>=1;i--)print $i}' file
3
2
1
6
5
4
```

```
9
8
7
```
都换行了，这并不是我们要的结果。怎么改进呢？
```
# awk '{for(i=NF;i>=1;i--){printf $i" "};print ""}' file   # print 本身就会新打印一行
3 2 1
6 5 4
9 8 7
```
或
```
# awk '{for(i=NF;i>=1;i--)if(i==1)printf $i"\n";else printf $i" "}' file
3 2 1
6 5 4
6 5 4
9 8 7
```
在这种情况下，是不是就排除第一行和倒数第一行呢？我们正序打印看下
排除第一行：
```
# awk '{for(i=2;i<=NF;i++){printf $i" "};print ""}' file
2 3
5 6
8 9
```
排除第二行：
```
# awk '{for(i=1;i<=NF-1;i++){printf $i" "};print ""}' file
1 2
4 5
7 8
```
IP 加单引号：
```
# echo '10.10.10.1 10.10.10.2 10.10.10.3' |awk '{for(i=1;i<=NF;i++)printf
"\047"$i"\047"}
'10.10.10.1'   '10.10.10.2'   '10.10.10.3'
```
\047 是 ASCII 码，可以通过 showkey -a 命令查看。

4）for 语句遍历数组

格式：for (var in array) statement

```
# seq -f "str%.g" 5 |awk '{a[NR]=$0}END{for(v in a)print v,a[v]}'
4 str4
5 str5
1 str1
2 str2
3 str3
```

5）break 和 continue 语句

break 跳过所有循环，continue 跳过当前循环。

```
# awk 'BEGIN{for(i=1;i<=5;i++){if(i==3){break};print i}}'
1
2
# awk 'BEGIN{for(i=1;i<=5;i++){if(i==3){continue};print i}}'
1
2
```

```
4
5
```

6）删除数组和元素

格式：

delete array[index]　删除数组元素

delete array　删除数组

```
# seq -f "str%.g" 5 |awk '{a[NR]=$0}END{delete a;for(v in a)print v,a[v]}'
空的…
# seq -f "str%.g" 5 |awk '{a[NR]=$0}END{delete a[3];for(v in a)print v,a[v]}'
4 str4
5 str5
1 str1
2 str2
```

7）exit 语句

格式：exit [expression]

exit 退出程序，与 shell 的 exit 一样。[expr]是 0-255 之间的数字。

```
# seq 5 |awk '{if($0~/3/)exit (123)}'
# echo $?
123
```

8.3.6 数组

数组：存储一系列相同类型的元素，键/值方式存储，通过下标（键）来访问值。

awk 中数组称为关联数组，不仅可以使用数字作为下标，还可以使用字符串作为下标。

数组元素的键和值存储在 awk 程序内部的一个表中，该表采用散列算法，因此数组元素是随机排序。

数组格式：array[index]=value

1）自定义数组

```
# awk 'BEGIN{a[0]="test";print a[0]}'
test
```

2）通过 NR 设置记录下标，下标从 1 开始

```
# tail -n3 /etc/passwd |awk -F: '{a[NR]=$1}END{print a[1]}'
systemd-network
# tail -n3 /etc/passwd |awk -F: '{a[NR]=$1}END{print a[2]}'
zabbix
# tail -n3 /etc/passwd |awk -F: '{a[NR]=$1}END{print a[3]}'
user
```

3）通过 for 循环遍历数组

```
# tail -n5 /etc/passwd |awk -F: '{a[NR]=$1}END{for(v in a)print a[v],v}'
zabbix 4
user 5
admin 1
systemd-bus-proxy 2
```

```
systemd-network 3
# tail -n5 /etc/passwd |awk -F: '{a[NR]=$1}END{for(i=1;i<=NR;i++)print a[i],i}'
admin 1
systemd-bus-proxy 2
systemd-network 3
zabbix 4
user 5
```

上面打印的 i 是数组的下标。
第一种 for 循环的结果是乱序的，刚说过，数组是无序存储。
第二种 for 循环通过下标获取的情况是排序正常。
所以当下标是数字序列时，还是用 for(expr1;expr2;expr3)循环表达式比较好，保持顺序不变。
4）通过++方式作为下标

```
# tail -n5 /etc/passwd |awk -F: '{a[x++]=$1}END{for(i=0;i<=x-1;i++)print a[i],i}'
admin 0
systemd-bus-proxy 1
systemd-network 2
zabbix 3
user 4
```

x 被 awk 初始化值是 0，没循环一次+1
5）使用字段作为下标

```
# tail -n5 /etc/passwd |awk -F: '{a[$1]=$7}END{for(v in a)print a[v],v}'
/sbin/nologin admin
/bin/bash user
/sbin/nologin systemd-network
/sbin/nologin systemd-bus-proxy
/sbin/nologin zabbix
```

6）统计相同字段出现次数

```
# tail /etc/services |awk '{a[$1]++}END{for(v in a)print a[v],v}'
2 com-bardac-dw
1 3gpp-cbsp
2 iqobject
1 matahari
2 isnetserv
2 blp5
# tail /etc/services |awk '{a[$1]+=1}END{for(v in a)print a[v],v}'
2 com-bardac-dw
1 3gpp-cbsp
2 iqobject
1 matahari
2 isnetserv
2 blp5
# tail /etc/services |awk '/blp5/{a[$1]++}END{for(v in a)print a[v],v}'
2 blp5
```

第一个字段作为下标，值被++初始化是 0，每次遇到下标（第一个字段）一样时，对应的值就会被+1，因此实现了统计出现次数。
想要实现去重的的话就简单了，只要打印下标即可。
7）统计 TCP 连接状态

```
# netstat -antp |awk '/^tcp/{a[$6]++}END{for(v in a)print a[v],v}'
9 LISTEN
6 ESTABLISHED
6 TIME_WAIT
```

8）只打印出现次数大于等于 2 的

```
# tail /etc/services |awk '{a[$1]++}END{for(v in a) if(a[v]>=2){print a[v],v}}'
2 com-bardac-dw
2 iqobject
2 isnetserv
2 blp5
```

9）去重

```
只打印重复的行：
# tail /etc/services |awk 'a[$1]++'
isnetserv           48128/udp                       # Image Systems Network Services
blp5                48129/udp                       # Bloomberg locator
com-bardac-dw       48556/udp                       # com-bardac-dw
iqobject            48619/udp                       # iqobject
不打印重复的行：
# tail /etc/services |awk '!a[$1]++'
3gpp-cbsp           48049/tcp                       # 3GPP Cell Broadcast Service
isnetserv           48128/tcp                       # Image Systems Network Services
blp5                48129/tcp                       # Bloomberg locator
com-bardac-dw       48556/tcp                       # com-bardac-dw
iqobject            48619/tcp                       # iqobject
matahari            49000/tcp                       # Matahari Broker
```

先明白一个情况，当值是 0 是为假，非 0 整数为真，知道这点就不难理解了。
只打印重复的行说明：当处理第一条记录时，执行了++，初始值是 0 为假，就不打印，如果再遇到相同的记录，值就会+1，不为 0，则打印。
不打印重复的行说明：当处理第一条记录时，执行了++，初始值是 0 为假，感叹号取反为真，打印，如果再遇到相同的记录，值就会+1，不为 0 为真，取反为假就不打印。

```
# tail /etc/services |awk '{if(a[$1]++)print $1}'
isnetserv
blp5
com-bardac-dw
iqobject
使用三目运算：
# tail /etc/services |awk '{print a[$1]++?$1:"no"}'
no
no
isnetserv
no
```

```
blp5
no
com-bardac-dw
no
iqobject
no
# tail /etc/services |awk '{if(!a[$1]++)print $1}'
3gpp-cbsp
isnetserv
blp5
com-bardac-dw
iqobject
matahari
```

10）统计每个相同字段的某字段总数：

```
# tail /etc/services |awk -F'[ /]+' '{a[$1]+=$2}END{for(v in a)print v, a[v]}'
com-bardac-dw 97112
3gpp-cbsp 48049
iqobject 97238
matahari 49000
isnetserv 96256
blp5 96258
```

11）多维数组

awk 的多维数组，实际上 awk 并不支持多维数组，而是逻辑上模拟二维数组的访问方式，比如 a[a,b]=1，使用 SUBSEP（默认\034）作为分隔下标字段，存储后是这样 a\034b。

示例：

```
# awk 'BEGIN{a["x","y"]=123;for(v in a) print v,a[v]}'
xy 123
```

我们可以重新复制 SUBSEP 变量，改变下标默认分隔符：

```
# awk 'BEGIN{SUBSEP=":";a["x","y"]=123;for(v in a) print v,a[v]}'
x:y 123
```

根据指定的字段统计出现次数：

```
# cat a
A 192.168.1.1 HTTP
B 192.168.1.2 HTTP
B 192.168.1.2 MYSQL
C 192.168.1.1 MYSQL
C 192.168.1.1 MQ
D 192.168.1.4 NGINX
# awk 'BEGIN{SUBSEP="-"}{a[$1,$2]++}END{for(v in a)print a[v],v}' a
1 D-192.168.1.4
1 A-192.168.1.1
2 C-192.168.1.1
2 B-192.168.1.2
```

8.3.7 内置函数

函数	描述
int(expr)	截断为整数
sqrt(expr)	平方根
rand()	返回一个随机数 N，0 和 1 范围，0 < N < 1
srand([expr])	使用 expr 生成随机数，如果不指定，默认使用当前时间为种子，如果前面有种子则使用生成随机数
asort(a, b)	对数组 a 的值进行排序，把排序后的值存到新的数组 b 中，新排序的数组下标从 1 开始
asorti(a,b)	对数组 a 的下标进行排序，同上
sub(r, s [, t])	对输入的记录用 s 替换 r，t 可选针对某字段替换 ，但只替换第一个字符串
gsub(r,s [, t])	对输入的记录用 s 替换 r，t 可选针对某字段替换，替换所有字符串
index(s, t)	返回 s 中字符串 t 的索引位置，0 为不存在
length([s])	返回 s 的长度
match(s, r [, a])	测试字符串 s 是否包含匹配 r 的字符串
split(s, a [, r [, seps]])	根据分隔符 seps 将 s 分成数组 a
substr(s, i [, n])	截取字符串 s 从 i 开始到长度 n，如果 n 没指定则是剩余部分
tolower(str)	str 中的所有大写转换成小写
toupper(str)	str 中的所有小写转换成大写
systime()	当前时间戳
strftime([format [, timestamp[, utc-flag]]])	格式化输出时间，将时间戳转为字符串

示例：

1）int()

```
# echo "123abc abc123 123abc123" |xargs -n1 | awk '{print int($0)}'
123
```

```
0
123
# awk 'BEGIN{print int(10/3)}'
3
```

2）sqrt()
获取 9 的平方根：

```
# awk 'BEGIN{print sqrt(9)}'
3
```

3）rand()和 srand()

```
rand()并不是每次运行就是一个随机数，会一直保持一个不变：
# awk 'BEGIN{print rand()}'
0.237788
当执行 srand()函数后，rand()才会发生变化，所以一般在 awk 着两个函数结合生成随机数，但是也有很大几率生成一样：
# awk 'BEGIN{srand();print rand()}'
0.31687
如果想生成 1-10 的随机数可以这样：
# awk 'BEGIN{srand();print int(rand()*10)}'
4
```

如果想更完美生成随机数，还得做相应的处理！
4）asort()和 asorti()

```
# seq -f "str%.g" 5 |awk '{a[x++]=$0}END{s=asort(a,b);for(i=1;i<=s;i++)print
b[i],i}'
str1 1
str2 2
str3 3
str4 4
str5 5
# seq -f "str%.g" 5 |awk '{a[x++]=$0}END{s=asorti(a,b);for(i=1;i<=s;i++)print
b[i],i}'
0 1
1 2
2 3
3 4
4 5
```

asort 将 a 数组的值放到数组 b，a 下标丢弃，并将数组 b 的总行号赋值给 s，新数组 b 下标从 1 开始，然后遍历。
5）sub()和 gsub()

```
# tail /etc/services |awk '/blp5/{sub(/tcp/,"icmp");print $0}'
blp5                48129/icmp                      # Bloomberg locator
blp5                48129/udp                       # Bloomberg locator
# tail /etc/services |awk '/blp5/{gsub(/c/,"9");print $0}'
blp5                48129/t9p                       # Bloomberg lo9ator
blp5                48129/udp                       # Bloomberg lo9ator
```

```
# echo "1 2 2 3 4 5" |awk 'gsub(2,7,$2){print $0}'
1 7 2 3 4 5
# echo "1 2 3 a b c" |awk 'gsub(/[0-9]/, '0'){print $0}'
0 0 0 a b c
```

在指定行前后加一行：

```
# seq 5 | awk 'NR==2{sub('/.*/',"txt\n&")}{print}'
1
txt
2
3
4
5
# seq 5 | awk 'NR==2{sub('/.*/',"&\ntxt")}{print}'
1
2
txt
3
4
5
```

6）index()

```
# tail -n 5 /etc/services |awk '{print index($2,"tcp")}'
7
0
7
0
7
```

7）length()

```
# tail -n 5 /etc/services |awk '{print length($2)}'
9
9
9
9
9
```

统计数组的长度：

```
# tail -n 5 /etc/services |awk '{a[$1]=$2}END{print length(a)}'
3
```

8）split()

```
# echo -e "123#456#789\nabc#cde#fgh" |awk '{split($0,a);for(v in a)print a[v],v}'
123#456#789 1
abc#cde#fgh 1
# echo -e "123#456#789\nabc#cde#fgh" |awk '{split($0,a,"#");for(v in a)print a[v],v}'
123 1
456 2
789 3
abc 1
cde 2
```

```
fgh 3
```

9）substr()

```
# echo -e "123#456#789\nabc#cde#fgh" |awk '{print
substr($0,4)}'
#456#789
#cde#fgh
# echo -e "123#456#789\nabc#cde#fgh" |awk '{print substr($0,4,5)}'
#456#
#cde#
```

10）tolower()和 toupper()

```
# echo -e "123#456#789\nABC#cde#fgh" |awk '{print tolower($0)}'
123#456#789
abc#cde#fgh
# echo -e "123#456#789\nabc#cde#fgh" |awk '{print toupper($0)}'
123#456#789
ABC#CDE#FGH
```

11)时间处理

```
返回当前时间戳:
# awk 'BEGIN{print systime()}'
1483297766
将时间戳转为日期和时间
# echo "1483297766" |awk '{print strftime("%Y-%m-%d %H:%M:%S",$0)}'
2017-01-01 14:09:26
```

8.3.8 I/O 语句

语句	描述
getline	读取下一个输入记录设置给$0
getline var	读取下一个输入记录并赋值给变量 var
command | getline [var]	运行 Shell 命令管道输出到$0 或 var
next	停止当前处理的输入记录后面动作
print	打印当前记录
printf fmt, expr-list	格式化输出
printf fmt, expr-list >file	格式输出和写到文件
system(cmd-line)	执行命令和返回状态

print ... >> file	追加输出到文件
print ... | command	打印输出作为命令输入

示例：

1）getline

```
获取匹配的下一行：
# seq 5 |awk '/3/{getline;print}'
4
# seq 5 |awk '/3/{print;getline;print}'
3
4
在匹配的下一行加个星号：
# seq 5 |awk '/3/{getline;sub(".*","&*");print}'
4*
# seq 5 |awk '/3/{print;getline;sub(".*","&*")}{print}'
1
2
3
4*
5
```

2）getline var

```
把 a 文件的行追加到 b 文件的行尾：
# cat a
a
b
c
# cat b
1 one
2 two
3 three
# awk '{getline line<"a";print $0,line}' b
1 one a
2 two b
3 three c
把 a 文件的行替换 b 文件的指定字段：
# awk '{getline line<"a";gsub($2,line,$2);print}' b
1 a
2 b
3 c
把 a 文件的行替换 b 文件的对应字段：
# awk '{getline line<"a";gsub("two",line,$2);print}' b
1 one
2 b
3 three
```

3）command | getline [var]

获取执行 shell 命令后结果的第一行：

```
# awk 'BEGIN{"seq 5"|getline var;print var}'
1
```

循环输出执行 shell 命令后的结果：

```
# awk 'BEGIN{while("seq 5"|getline)print}'
1
2
3
4
5
```

4）next

不打印匹配行：

```
# seq 5 |awk '{if($0==3){next}else{print}}'
1
2
4
5
```

删除指定行：

```
# seq 5 |awk 'NR==1{next}{print $0}'
2
3
4
5
```

如果前面动作成功，就遇到 next，后面的动作不再执行，跳过。

或者：

```
# seq 5 |awk 'NR!=1{print}'
2
3
4
5
```

把第一行内容放到每行的前面：

```
# cat a
hello
1 a
2 b
3 c
# awk 'NR==1{s=$0;next}{print s,$0}' a
hello  1 a
hello  2 b
hello  3 c
# awk 'NR==1{s=$0}NF!=1{print s,$0}' a
hello  1 a
hello  2 b
hello  3 c
```

5）system()

执行 shell 命令判断返回值：

```
# awk 'BEGIN{if(system("grep root /etc/passwd &>/dev/null")==0)print "yes";else print "no"}'
yes
```

6）打印结果写到文件

```
# tail -n5 /etc/services |awk '{print $2 > "a.txt"}'
# cat a.txt
48049/tcp
48128/tcp
48128/udp
48129/tcp
48129/udp
```

7）管道连接 shell 命令

```
将结果通过 grep 命令过滤：
# tail -n5 /etc/services |awk '{print $2|"grep tcp"}'
48556/tcp
48619/tcp
49000/tcp
```

8.3.9 printf 语句

格式化输出，默认打印字符串不换行。
格式：printf [format] arguments

Format	描述
%s	一个字符串
%d，%i	一个小数
%f	一个浮点数
%.ns	输出字符串，n 是输出几个字符
%ni	输出整数，n 是输出几个数字
%m.nf	输出浮点数，m 是输出整数位数，n 是输出的小数位数
%x	不带正负号的十六进制，使用 a 至 f 表示 10 到 15
%X	不带正负号的十六进制，使用 A 至 F 表示 10 至 15
%%	输出单个%
%-5s	左对齐，对参数每个字段左对齐,宽度为 5

%-4.2f	左对齐，宽度为 4，保留两位小数
%5s	右对齐，不加横线表示右对齐

示例：

```
将换行符换成逗号：
# seq 5 |awk '{if($0!=5)printf "%s,",$0;else print $0}'
1,2,3,4,5
小括号中的 5 是最后一个数字。
输出一个字符：
# awk 'BEGIN{printf "%.1s\n","abc"}'
a
保留一个小数点：
# awk 'BEGIN{printf "%.2f\n",10/3}'
3.33
格式化输出：
# awk 'BEGIN{printf "user:%s\tpass:%d\n","abc",123}'
user:abc            pass:123
左对齐宽度 10：
# awk 'BEGIN{printf "%-10s %-10s %-10s\n","ID","Name","Passwd"}'
ID              Name            Passwd
右对齐宽度 10：
# awk 'BEGIN{printf "%10s %10s %10s\n","ID","Name","Passwd"}'
            ID           Name        Passwd
打印表格：
# vi test.awk
BEGIN{
print "+--------------------+--------------------+";
printf "|%-20s|%-20s|\n","Name","Number";
print "+--------------------+--------------------+";
}
# awk -f test.awk
+--------------------+--------------------+
|Name                |Number              |
+--------------------+--------------------+
格式化输出：
# awk -F: 'BEGIN{printf "UserName\t\tShell\n----------------------------\n"}{printf
"%-20s %-20s\n",$1,$7}END{print "END...\n"}' /etc/passwd
打印十六进制：
# awk 'BEGIN{printf "%x %X",123,123}'
7b 7B
```

8.3.10 自定义函数

格式：function name(parameter list) { statements }

示例：

```
# awk 'function myfunc(a,b){return a+b}BEGIN{print myfunc(1,2)}'
```

```
3
```

8.3.11 需求案例

1）分析 Nginx 日志
日志格式：
'$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$http_referer" "$http_user_agent" "$http_x_forwarded_for"'

```
统计访问 IP 次数：
# awk '{a[$1]++}END{for(v in a)print v,a[v]}' access.log
统计访问访问大于 100 次的 IP：
# awk '{a[$1]++}END{for(v in a){if(a[v]>100)print v,a[v]}}' access.log
统计访问 IP 次数并排序取前 10：
# awk '{a[$1]++}END{for(v in a)print v,a[v] |"sort -k2 -nr |head -10"}' access.log
统计时间段访问最多的 IP：
# awk '$4>="[02/Jan/2017:00:02:00" && $4<="[02/Jan/2017:00:03:00"{a[$1]++}END{for(v in a)print v,a[v]}' access.log
统计上一分钟访问量：
#  date=$(date -d '-1 minute' +%d/%d/%Y:%H:%M)
# awk -vdate=$date '$4~date{c++}END{print c}' access.log
统计访问最多的 10 个页面：
#  awk  '{a[$7]++}END{for(v in  a)print v,a[v] |"sort  -k1  -nr|head  -n10"}'  access.log
统计每个 URL 数量和返回内容总大小：
# awk '{a[$7]++;size[$7]+=$10}END{for(v in a)print a[v],v,size[v]}' access.log
统计每个 IP 访问状态码数量：
# awk '{a[$1" "$9]++}END{for(v in a)print v,a[v]}' access.log
统计访问 IP 是 404 状态次数：
# awk  '{if($9~/404/)a[$1"  "$9]++}END{for(i  in  a)print v,a[v]}'  access.log
```

2）两个文件对比
找出 b 文件在 a 文件相同记录：

```
# seq 1 5 > a
# seq 3 7 > b
方法 1：
# awk 'FNR==NR{a[$0];next}{if($0 in a)print $0}' a b
3
4
5
# awk 'FNR==NR{a[$0];next}{if($0 in a)print FILENAME,$0}' a b
b 3
b 4
b 5
# awk 'FNR==NR{a[$0]}NR>FNR{if($0 in a)print $0}' a b
3
4
5
```

```
# awk 'FNR==NR{a[$0]=1;next}(a[$0]==1)' a b    # a[$0]是通过b文件每行获取值，如果是1说明有
# awk 'FNR==NR{a[$0]=1;next}{if(a[$0]==1)print}' a b
3
4
5
方法2：
# awk 'FILENAME=="a"{a[$0]}FILENAME=="b"{if($0 in a)print $0}' a b
3
4
5
方法3：
# awk 'ARGIND==1{a[$0]=1}ARGIND==2 && a[$0]==1' a b
3
4
5
```

找出b文件在a文件不同记录：

```
方法1：
# awk 'FNR==NR{a[$0];next}!($0 in a)' a b
6
7
# awk 'FNR==NR{a[$0]=1;next}(a[$0]!=1)' a b
# awk 'FNR==NR{a[$0]=1;next}{if(a[$0]!=1)print}' a b
6
7
方法2：
# awk 'FILENAME=="a"{a[$0]=1}FILENAME=="b" && a[$0]!=1' a b
方法3：
#  awk 'ARGIND==1{a[$0]=1}ARGIND==2 && a[$0]!=1' a b
```

3）合并两个文件

将a文件合并到b文件：

```
# cat a
zhangsan 20
lisi 23
wangwu 29
# cat b
zhangsan man
lisi woman
wangwu man
# awk 'FNR==NR{a[$1]=$0;next}{print a[$1],$2}' a b
zhangsan 20 man
lisi 23 woman
wangwu 29 man
# awk 'FNR==NR{a[$1]=$0}NR>FNR{print a[$1],$2}' a b
zhangsan 20 man
lisi 23 woman
```

```
wangwu 29 man
```

将 a 文件相同 IP 的服务名合并：

```
# cat a
192.168.1.1:  httpd
192.168.1.1:  tomcat
192.168.1.2:  httpd
192.168.1.2:  postfix
192.168.1.3:  mysqld
192.168.1.4:  httpd
# awk 'BEGIN{FS=":";OFS=":"}{a[$1]=a[$1] $2}END{for(v in a)print v,a[v]}' a
192.168.1.4:  httpd
192.168.1.1:  httpd  tomcat
192.168.1.2:  httpd  postfix
192.168.1.3:  mysqld
```

说明：数组 a 存储是$1=a[$1] $2，第一个 a[$1]是以第一个字段为下标，值是 a[$1] $2，也就是 $1=a[$1] $2，值的 a[$1]是用第一个字段为下标获取对应的值，但第一次数组 a 还没有元素，那么 a[$1]是空值，此时数组存储是 192.168.1.1=httpd，再遇到 192.168.1.1 时，a[$1]通过第一字段下标获得上次数组的 httpd，把当前处理的行第二个字段放到上一次同下标的值后面，作为下标 192.168.1.1 的新值。此时数组存储是 192.168.1.1=httpd tomcat。每次遇到相同的下标（第一个字段）就会获取上次这个下标对应的值与当前字段并作为此下标的新值。

4）将第一列合并到一行

```
# cat file
1 2 3
4 5 6
7 8 9
# awk '{for(i=1;i<=NF;i++)a[i]=a[i]$i" "}END{for(v in a)print a[v]}' file
1 4 7
2 5 8
3 6 9
```

说明：

for 循环是遍历每行的字段，NF 等于 3，循环 3 次。

读取第一行时：

第一个字段：a[1]=a[1]1" " 值 a[1]还未定义数组，下标也获取不到对应的值，所以为空，因此 a[1]=1 。

第二个字段：a[2]=a[2]2" " 值 a[2]数组 a 已经定义，但没有 2 这个下标，也获取不到对应的值，为空，因此 a[2]=2 。

第三个字段：a[3]=a[3]3" " 值 a[2]与上面一样，为空,a[3]=3 。

读取第二行时：

第一个字段：a[1]=a[1]4" " 值 a[2]获取数组 a 的 2 为下标对应的值，上面已经有这个下标了，对应的值是 1，因此 a[1]=1 4

第二个字段：a[2]=a[2]5" " 同上，a[2]=2 5

第三个字段：a[3]=a[3]6" " 同上，a[2]=3 6

读取第三行时处理方式同上，数组最后还是三个下标，分别是 1=1 4 7，2=2 5 8，3=3 6 9。最后 for 循环输出所有下标值。

5）字符串拆分，统计出现的次数

字符串拆分：

```
方法 1：
# echo "hello world" |awk -F '' '{print $1}'
h
# echo "hello" |awk -F '' '{for(i=1;i<=NF;i++)print $i}'
h
e
l
l
o
方法 2：
# echo "hello" |awk '{split($0,a,"");for(v in a)print a[v]}'
l
o
h
e
l
```

统计字符串中每个字母出现的次数：

```
# echo "a.b.c,c.d.e" |awk -F '[.,]' '{for(i=1;i<=NF;i++)a[$i]++}END{for(v in a)print
v,a[v]}'
a 1
b 1
c 2
d 1
e 1
```

5）费用统计

```
# cat a
zhangsan 8000 1
zhangsan 5000 1
lisi 1000 1
lisi 2000 1
wangwu 1500 1
zhaoliu 6000 1
zhaoliu 2000 1
zhaoliu 3000 1
# awk '{name[$1]++;cost[$1]+=$2;number[$1]+=$3}END{for(v in name)print
v,cost[v],number[v]}' a
zhangsan 5000 1
lisi 3000 2
wangwu 1500 1
zhaoliu 11000 3
```

6）获取数字字段最大值

```
# cat a
a b 1
c d 2
```

```
e f 3
g h 3
i j 2
```
获取第三字段最大值：
```
# awk 'BEGIN{max=0}{if($3>max)max=$3}END{print max}' a
3
```
打印第三字段最大行：
```
# awk 'BEGIN{max=0}{a[$0]=$3;if($3>max)max=$3}END{for(v in a)print v,a[v],max}' a
g h 3 3 3
e f 3 3 3
c d 2 2 3
a b 1 1 3
i j 2 2 3
# awk 'BEGIN{max=0}{a[$0]=$3;if($3>max)max=$3}END{for(v in a)if(a[v]==max)print v}' a
g h 3
e f 3
```
7）去除第一行和最后一行
```
# seq 5 |awk 'NR>2{print s}{s=$0}'
2
3
4
```
读取第一行，NR=1，不执行 print s，s=1
读取第二行，NR=2，不执行 print s，s=2 （大于为真）
读取第三行，NR=3，执行 print s，此时 s 是上一次 p 赋值内容 2，s=3
最后一行，执行 print s，打印倒数第二行，s=最后一行
获取 Nginx 负载均衡配置端 IP 和端口：
```
# cat nginx.conf
upstream example-servers1 {
   server 127.0.0.1:80 weight=1 max_fails=2 fail_timeout=30s;
}
upstream example-servers2 {
   server 127.0.0.1:80 weight=1 max_fails=2 fail_timeout=30s;
   server 127.0.0.1:82 backup;
}
# awk '/example-servers1/,/}/{if(NR>2){print s}{s=$2}}' nginx.conf
127.0.0.1:80
# awk '/example-servers1/,/}/{if(i>1)print s;s=$2;i++}' nginx.conf
# awk '/example-servers1/,/}/{if(i>1){print s}{s=$2;i++}}' nginx.conf
127.0.0.1:80
```
读取第一行，i 初始值为 0，0>1 为假，不执行 print s，x=example-servers1，i=1
读取第二行，i=1，1>1 为假，不执行 print s，s=127.0.0.1:80,i=2
读取第三行，i=2，2>1 为真，执行 print s，此时 s 是上一次 s 赋值内容 127.0.0.1:80，i=3
最后一行，执行 print s，打印倒数第二行，s=最后一行。
这种方式与上面一样，只是用 i++作为计数器。
8）知道上述方式，就可以实现这种需求了，打印匹配行的上一行

```
# seq 5 |awk '/3/{print s}{s=$0}'
2
```

其他参考资料：http://www.gnu.org/software/gawk/manual/gawk.html

第八章　Shell 标准输入、输出和错误

文件描述符（fd）：文件描述符是一个非负整数，在打开现存文件或新建文件时，内核会返回一个文件描述符，读写文件也需要使用文件描述符来访问文件。
内核为每个进程维护该进程打开的文件记录表。文件描述符只适于 Unix、Linux 操作系统。

8.1 标准输入、输出和错误

文件描述符	描述	映射关系
0	标准输入，键盘	/dev/stdin -> /proc/self/fd/0
1	标准输出，屏幕	/dev/stdout -> /proc/self/fd/1
2	标准错误，屏幕	/dev/stderr -> /proc/self/fd/2

8.2 重定向符号

符号	描述
>	符号左边输出作为右边输入（标准输出）
>>	符号左边输出追加右边输入
<	符号右边输出作为左边输入（标准输入）
<<	符号右边输出追加左边输入
&	重定向绑定符号

输入和输出可以被重定向符号解释到 shell。
shell 命令是从左到右依次执行命令。

下面 n 字母是文件描述符。

8.3 重定向输出

1）覆盖输出
一般格式：[n]>word

如果 n 没有指定，默认是 1
示例：

```
打印结果写到文件：
# echo "test" > a.txt
当没有安装 bc 计算器时，错误输出结果写到文件：
# echo "1 + 1" |bc 2 > error.log
```

2）追加重定向输出
一般格式：[n]>>word
如果 n 没有指定，默认是 1
示例：

```
打印结果追加到文件：
# echo "test" >> a.txt
当没有安装 bc 计算器时，错误输出结果追加文件：
# echo "1 + 1" |bc 2 > error.log
```

8.4 重定向输入

一般格式：[n]<word
如果 n 没有指定，默认是 0
示例：

```
a.txt 内容作为 grep 输入：
# grep "test" --color < a.txt
```

8.5 重定向标准输出和标准错误

1）覆盖重定向标准输出和标准错误
&>word 和>&word 等价于 >word 2>&1
&将标准输出和标准输入绑定到一起，重定向 word 文件。
示例：

```
当不确定执行对错时都覆盖到文件：
# echo "1 + 1" |bc &> error.log
当不确定执行对错时都覆盖到文件：
# echo "1 + 1" |bc > error.log  2>&1
```

2）追加重定向标准输出和标准错误
&>>word 等价于>>word 2>&1
示例：

```
当不确定执行对错时都追加文件：
# echo "1 + 1" |bc &>> error.log
```

将标准输出和标准输入追加重定向到 delimiter：

```
<< delimiter
     here-document
```

```
delimiter
```

从当前 shell 读取输入源，直到遇到一行只包含 delimiter 终止，内容作为标准输入。
将 eof 标准输入作为 cat 标准输出再写到 a.txt：

```
# cat << eof
123
abc
eof

123
abc
# cat > a.txt << eof
> 123
> abc
> eof
```

8.6 重定向到空设备

/dev/null 是一个空设备，向它写入的数组都会丢弃，但返回状态是成功的。与其对应的还有一个 /dev/zero 设备，提供无限的 0 数据流。
在写 Shell 脚本时我们经常会用到/dev/null 设备，将 stdout、stderr 输出给它，也就是我们不想要这些输出的数据。
通过重定向到/dev/null 忽略输出，比如我们没有安装 bc 计算器，正常会抛出没有发现命令：

```
# echo "1 + 1" |bc >/dev/null 2>&1
```

这就让标准和错误输出到了空设备。
忽略标准输出：

```
# echo "test" >/dev/null
```

忽略错误输出：

```
# echo "1 + 1" |bc 2>/dev/null
```

8.7 read 命令

read 命令从标准输入读取，并把输入的内容复制给变量。
命令格式： read [-ers] [-a array] [-d delim] [-i text] [-n nchars] [-N nchars] [-p prompt] [-t timeout] [-u fd] [name ...]

-e	在一个交互 shell 中使用 readline 获取行
-r	不允许反斜杠转义任何字符
-s	隐藏输入
-a array	保存为数组，元素以空格分隔

-d delimiter	持续读取直到遇到 delimiter 第一个字符退出
-n nchars	读取 nchars 个字符返回，而不是等到换行符
-p prompt	提示信息
-t timeout	等待超时时间，秒
-u fd	指定文件描述符号码作为输入，默认是 0
name	变量名

示例：

```
获取用户输入保存到变量：
# read -p "Please input your name: " VAR
Please input your name: lizhenliang
# echo $VAR
lizhenliang
用户输入保存为数组：
# read -p "Please input your name: " -a ARRAY
Please input your name: a b c
# echo ${ARRAY[*]}
a b c
遇到 e 字符返回：
# read -d e VAR
123
456
e
# echo $VAR
123 456
从文件作为 read 标准输入：
# cat a.txt
adfasfd
# read VAR < a.txt
# echo $VAR
adfasfd
while 循环读取每一行作为 read 的标准输入：
# cat a.txt |while read LINE; do echo $LINE; done
123
abc
分别变量赋值：
# read a b c
1 2 3
# echo $a
1
# echo $b
2
# echo $c
```

```
3
# echo 1 2 3 | while read a b c;do echo "$a $b $c"; done
1 2 3
```

第九章　Shell 信号发送与捕捉

9.1 Linux 信号类型

信号（Signal）：信号是在软件层次上对中断机制的一种模拟，通过给一个进程发送信号，执行相应的处理函数。
进程可以通过三种方式来响应一个信号：
1）忽略信号，即对信号不做任何处理，其中有两个信号不能忽略：SIGKILL 及 SIGSTOP。
2）捕捉信号。
3）执行缺省操作，Linux 对每种信号都规定了默认操作。
Linux 究竟采用上述三种方式的哪一个来响应信号呢？取决于传递给响应的 API 函数。
Linux 支持的信号有：

编号	信号名称	缺省动作	描述
1	SIGHUP	终止	终止进程，挂起
2	SIGINT	终止	键盘输入中断命令，一般是 CTRL+C
3	SIGQUIT	CoreDump	键盘输入退出命令，一般是 CTRL+\
4	SIGILL	CoreDump	非法指令
5	SIGTRAP	CoreDump	trap 指令发出，一般调试用
6	SIGABRT	CoreDump	abort(3)发出的终止信号
7	SIGBUS	CoreDump	非法地址
8	SIGFPE	CoreDump	浮点数异常
9	SIGKILL	终止	立即停止进程，不能捕获，不能忽略
10	SIGUSR1	终止	用户自定义信号 1，像 Nginx 就支持 USR1 信号，用于重载配置，重新打开日志
11	SIGSEGV	CoreDump	无效内存引用
12	SIGUSR2	终止	用户自定义信号 2
13	SIGPIPE	终止	管道不能访问

14	SIGALRM	终止	时钟信号，alrm(2)发出的终止信号
15	SIGTERM	终止	终止信号，进程会先关闭正在运行的任务或打开的文件再终止，有时间进程在有运行的任务而忽略此信号。不能捕捉
16	SIGSTKFLT	终止	处理器栈错误
17	SIGCHLD	可忽略	子进程结束时，父进程收到的信号
18	SIGCONT	可忽略	让终止的进程继续执行
19	SIGSTOP	停止	停止进程，不能忽略，不能捕获
20	SIGSTP	停止	停止进程，一般是 CTRL+Z
21	SIGTTIN	停止	后台进程从终端读数据
22	SIGTTOU	停止	后台进程从终端写数据
23	SIGURG	可忽略	紧急数组是否到达 socket
24	SIGXCPU	CoreDump	超出 CPU 占用资源限制
25	SIGXFSZ	CoreDump	超出文件大小资源限制
26	SIGVTALRM	终止	虚拟时钟信号，类似于 SIGALRM，但计算的是进程占用的时间
27	SIGPROF	终止	类似与 SIGALRM，但计算的是进程占用 CPU 的时间
28	SIGWINCH	可忽略	窗口大小改变发出的信号
29	SIGIO	终止	文件描述符准备就绪，可以输入/输出操作了
30	SIGPWR	终止	电源失败
31	SIGSYS	CoreDump	非法系统调用

CoreDump（核心转储）：当程序运行过程中异常退出时，内核把当前程序在内存状况存储在一个 core 文件中，以便调试。

Linux 支持两种信号：
一种是标准信号，编号 1-31，称为非可靠信号（非实时），不支持队列，信号可能会丢失，比如发送多次相同的信号，进程只能收到一次，如果第一个信号没有处理完，第二个信号将会丢弃。
另一种是扩展信号，编号 32-64，称为可靠信号（实时），支持队列，发多少次进程就可以收到多少次。
信号类型比较多，我们只要了解下，记住几个常用信号就行了，红色标记的我觉得需要记下。

发送信号一般有两种情况：
一种是内核检测到系统事件，比如键盘输入 CTRL+C 会发送 SIGINT 信号。
另一种是通过系统调用 kill 命令来向一个进程发送信号。

9.2 kill 命令

kill 命令发送信号给进程。
命令格式：kill [-s sigspec | -n signum | -sigspec] pid | jobspec ...
          kill -l [sigspec]
-s # 信号名称
-n # 信号编号
-l # 打印编号 1-31 信号名称
示例：

```
给一个进程发送终止信号：
kill -s SIGTERM pid
或
kill -n 15 pid
或
kill -15 pid
或
kill -TREM pid
```

9.3 trap 命令

trap 命令定义 shell 脚本在运行时根据接收的信号做相应的处理。
命令格式：trap [-lp] [[arg] signal_spec ...]
-l   # 打印编号 1-64 编号信号名称
arg   # 捕获信号后执行的命令或者函数
signal_spec # 信号名或编号
一般捕捉信号后，做以下几个动作：
1）清除临时文件
2）忽略该信号
3）询问用户是否终止脚本执行
示例 1：按 CTRL+C 不退出循环

```
#!/bin/bash
trap "" 2      # 不指定 arg 就不做任何操作，后面也可以写多个信号，以空格分隔
for i in {1..10}; do
     echo $i
     sleep 1
done
# bash a.sh
1
2
3
^C4
5
```

```
6
^C7
8
9
10
```

示例 2：循环打印数字，按 CTRL+C 退出，并打印退出提示

```
#!/bin/bash
trap "echo 'exit...';exit" 2
for i in {1..10}; do
     echo $i
     sleep 1
done
# bash test.sh
1
2
3
^Cexit...
```

示例 3：让用户选择是否终止循环

```
#!/bin/bash
trap "func" 2
func() {
     read -p "Terminate the process? (Y/N): " input
     if [ $input == "Y" ]; then
            exit
     fi
}
for i in {1..10}; do
     echo $i
     sleep 1
done

# bash a.sh
1
2
3
^CTerminate the process? (Y/N): Y
# bash a.sh
1
2
3
^CTerminate the process? (Y/N): N
4
5
6
...
```