



文本过滤

1

主要内容和学习要求

- q 能够熟练运用 `grep` 命令
- q 掌握 `sed` 流编辑器
- q 学会使用 `awk` 编程

2

grep 家族

q `grep` 是 Linux 下使用最广泛的命令之一，其作用是在一个或多个文件中查找某个字符模式所在的行，并将结果输出到屏幕上。

`grep` 命令不会对输入文件进行任何修改或影响

q `grep` 家族由 `grep`、`egrep` 和 `fgrep` 组成：

- u `grep`：标准 `grep` 命令，主要讨论此命令。
- u `egrep`：扩展 `grep`，支持基本及扩展的正则表达式。
- u `fgrep`：固定 `grep` (fixed `grep`)，也称快速 `grep` (fast `grep`)，按字面解释所有的字符，即正则表达式中的元字符不会被特殊处理。这里的“快速”并不是指速度快。

3

grep 的使用

q `grep` 命令的一般形式

```
grep [选项] pattern file1 file2 ...
```

- l `pattern`：可以是正则表达式（用单引号括起来）、或字符串（加双引号）、或一个单词。
- l `file1 file2 ...`：文件名列表，作为 `grep` 命令的输入；`grep` 的输入也可以来自标准输入或管道；

q 可以把匹配模式写入到一个文件中，每行写一个，然后使用 `-f` 选项，将该匹配模式传递给 `grep` 命令

```
grep -f patternfile file1 file2 ...
```

4

grep 常用选项

-c	只输出匹配的行的总数
-i	不区分大小写
-h	查询多个文件时，不显示文件名
-l	查询多个文件时，只输出包含匹配模式的文件的文件名
-n	显示匹配的行及行号
-v	反向查找，即只显示不包含匹配模式的行
-s	不显示错误信息

```
grep -i 'an*' datafile
```

5

grep 命令应用举例

u 查询多个文件,可以使用通配符“*”

```
grep "math2" *.txt
```

```
grep "12" *
```

u 反向匹配

```
ps aux | grep "ssh" | grep -v "grep"
```

u 匹配空行

```
grep -n '^$' datafile
```

```
grep -v '^$' datafile > datafile2
```

6

grep 命令应用举例

u 精确匹配单词: \`<` 和 \`>`

l 找出所有包含以 `north` 开头的单词的行

```
grep '\<north' datafile
```

l 找出所有包含以 `west` 结尾的单词的行

```
grep 'west\>' datafile
```

l 找出所有包含 `north` 单词的行

```
grep '\<north\>' datafile
```

7

grep 命令应用举例

u 递归搜索目录中的所有文件: `-r`

```
grep -r "north" datafile ~/Teaching/linux/
```

u 关于某个字符连续出现次数的匹配

```
grep 'o\{2,\}' helloworld
```

```
'o\{2,4\}', 'o\{2,4\}', 'lo\{2,4\}'
```

8

grep 命令应用举例

其它

```
grep '^n' datafile
```

```
grep 'y$' datafile
```

```
grep 'r\.' datafile
```

```
grep '^[we]' datafile
```

```
grep -i 'ss*' datafile
```

```
grep -n '[a-z]\{9\}' datafile
```

```
grep -c '\<[a-z].*n\>' datafile
```

9

grep 与管道

```
ls -l | grep '^d'
```

如果传递给 grep 的文件名参数中有目录的话，需使用“-d”选项

```
grep -d [ACTION] directory_name
```

其中 ACTION 可以是

read: 把目录文件当作普通文件来读取

skip: 目录将被忽略而跳过

recurse: 递归的方式读取目录下的每一个文件，可以用选项“-r”代替“-d recurse”

```
grep -rl "eth0" /etc
```

10

egrep 命令

使用 egrep 的主要好处是，它在使用 grep 提供的正则表达式元字符基础上增加了更多的元字符，见下表，但不能使用 \{ \}。

在 Linux 下: `egrep = grep -E`

egrep 增加的元字符

+	匹配一个或多个前一字符
?	匹配零个或一个前一字符
str1 str2	匹配 str1 或 str2
()	字符组

注意星号 * 和问号 ? 在 shell 通配符和正则表达式中的区别

11

egrep 举例与 fgrep

```
egrep 'WE+' datafile
```

```
egrep 'WE?' datafile
```

```
egrep 'S(h|u)' datafile
```

```
egrep 'Sh|u' datafile
```

fgrep 命令

fgrep 的使用方法与 grep 类似，但对正则表达式中的任何元字符都不做特殊处理。

```
fgrep '^n' datafile
```

12

流编辑器 sed

sed 是什么

sed 是一个精简的、非交互式的编辑器，它在命令行中输入编辑命令和指定文件名，然后在屏幕上查看输出。

sed 如何工作

sed 逐行处理文件（或输入），并将输出结果发送到屏幕。即：sed 从输入（可以是文件或其它标准输入）中读取一行，将之拷贝到一个编辑缓冲区，按指定的 sed 编辑命令进行处理，编辑完后将其发送到屏幕上，然后把这行从编辑缓冲区中删除，读取下面一行。重复此过程直到全部处理结束。

sed 只是对文件在内存中的副本进行操作，所以 sed 不会修改输入文件的内容。sed 总是输出到标准输出，可以使用重定向将 sed 的输出保存到文件中。

13

sed 的三种调用方式

在命令行中直接调用

```
sed [-n][-e] 'sed_cmd' input_file
```

! -n: 缺省情况下，sed 在将下一行读入缓冲区之前，自动输出缓冲区中的内容。此选项可以关闭自动输出。

! -e: 允许调用多条 sed 命令，如：

```
sed -e 'sed_cmd1' -e 'sed_cmd2' input_file
```

! sed_cmd: 使用格式: [address]sed_edit_cmd (通常用单引号括起来)，其中 address 为 sed 的行定位模式，用于指定将要被 sed 编辑的行。如果省略，sed 将编辑所有的行。sed_edit_cmd 为 sed 对被编辑行将要进行的编辑操作。

! input_file: sed 编辑的文件列表，若省略，sed 将从标准输入（重定向或管道）中读取输入。

14

sed 的三种调用方式

将 sed 命令插入脚本文件，然后调用

```
sed [选项] -f sed_script_file input_file
```

```
例: sed -n -f sedfile1 datafile
```

将 sed 命令插入脚本文件，生成 sed 可执行脚本文件，在命令行中直接键入脚本文件名来执行。

```
#!/bin/sed -f
sed_cmd1
... ..
```

```
例: ./sedfile2.sed -n datafile
```

15

定位方式

sed_cmd 中 address 的定位方式

n	表示第 n 行
\$	表示最后一行
m,n	表示从第 m 行到第 n 行
/pattern/	查询包含指定模式的行。如 /disk/、/[a-z]/
/pattern/,n	表示从包含指定模式的行到第 n 行
n,/pattern/	表示从第 n 行到包含指定模式的行
/模式1/,/模式2/	表示从包含模式1到包含模式2的行
!	反向选择，如 m,n! 的结果与 m,n 相反

16

常用 sed 编辑命令

q 常用的 `sed_edit_cmd`

u p: 打印匹配行

```
sed -n '1,3p' datafile // ('1,3!p')
```

```
sed -n '$p' datafile
```

```
sed -n '/north/p' datafile
```

u =: 显示匹配行的行号

```
sed -n '/north/= ' datafile
```

u d: 删除匹配的行

```
sed -n '/north/d' datafile
```

17

常用 sed 编辑命令

**u a **: 在指定行后面追加一行或多行文本, 并显示添加的新内容, 该命令主要用于 `sed` 脚本中。

```
sed -n '/eastern/a/newline1\
                        newline2\
                        newlineN' datafile
```

**u i **: 在指定行前追加一行或多行, 并显示添加的新内容, 使用格式同 `a \`

**u c **: 用新文本替换指定的行, 使用格式同 `a \`

u l: 显示指定行中所有字符, 包括控制字符(非打印字符)

```
sed -n '/west/l' datafile
```

18

常用 sed 编辑命令

u s: 替换命令, 使用格式为:

```
[address]s/old/new/[gpw]
```

l address: 如果省略, 表示编辑所有的行。

l g: 全局替换

l p: 打印被修改后的行

l w fname: 将被替换后的行内容写到指定的文件中

```
sed -n 's/west/east/gp' datafile
```

```
sed -n 's/Aanny/Anndy/w newdata' datafile
```

```
sed 's/[0-9][0-9]$/&.5/' datafile
```

& 符号用在替换字符串中时, 代表 **被替换的字符串**

19

常用 sed 编辑命令

u r: 读文件, 将另外一个文件中的内容附加到指定行后。

```
sed -n '$r newdata' datafile
```

u w: 写文件, 将指定行写入到另外一个文件中。

```
sed -n '/west/w newdata' datafile
```

u n: 将指定行的下面一行读入编辑缓冲区。

```
sed -n '/eastern/{n;s/AM/Archie/p}' datafile
```

对指定行同时使用多个 `sed` 编辑命令时, 需用大括号 “`{ }`” 括起来, 命令之间用分号 “`;`” 隔开。注意与 `-e` 选项的区别

20

常见的 sed 编辑命令小结

u q: 退出, 读取到指定行后退出 `sed`。

```
sed '/east/{s/east/west;q}' datafile
```

常见的 sed 编辑命令小结

命令	描述	命令	描述
<code>p</code>	打印匹配行	<code>s</code>	替换命令
<code>=</code>	显示匹配行的行号	<code>l</code>	显示指定行中所有字符
<code>d</code>	删除匹配的行	<code>r</code>	读文件
<code>a \</code>	在指定行后面追加文本	<code>w</code>	写文件
<code>i \</code>	在指定行前面追加文本	<code>n</code>	读取指定行的下面一行
<code>c \</code>	用新文本替换指定的行	<code>q</code>	退出 <code>sed</code>

21

shell 变量的使用

q `sed` 支持 `shell` 变量的使用

在 `sed_cmd` 中可以使用 `shell` 变量, 此时应使用双引号

```
myvar= "west"
```

```
sed -n "/${myvar}/p" datafile
```

q 如何输入控制字符, 如: 回车、Esc、F1 等

以输入 **回车** (`^M`) 为例:

先按 **Ctrl+v**, 释放按下的两个键后, 按下对应的功能键 (**Enter** 键) 即可。

22

一些 sed 行命令集

<code>'/north/p'</code>	打印所有包含 <code>north</code> 的行
<code>'/north!/p'</code>	打印所有不包含 <code>north</code> 的行
<code>'s/\.\$//g'</code>	删除以句点结尾的行中末尾的句点
<code>'s/^ */g'</code>	删除行首空格 (命令中 <code>^ *</code> 之间有两个空格)
<code>'s/ */ /g'</code>	将连续多个空格替换为一个空格 命令中 <code>*/</code> 前有三个空格, 后面是一个空格
<code>'/^\$/d'</code>	删除空行
<code>'s/^.//g'</code>	删除每行的第一个字符, 同 <code>'s/./'</code>
<code>'s/^/%g'</code>	在每行的最前面添加百分号 <code>%</code>
<code>'3,5s/d/D/'</code>	把第 3 行到第 5 行中每行的第一个 <code>d</code> 改成 <code>D</code>

23

awk 介绍

q `awk` 是什么

l `awk` 是一种用于处理数据和生成报告的编程语言

l `awk` 可以在命令中进行一些简单的操作, 也可以被写成脚本来处理较大的应用问题

l `awk` 与 `grep`、`sed` 结合使用, 将使 `shell` 编程更加容易

l `Linux` 下使用的 `awk` 是 `gawk`

q `awk` 如何工作

`awk` 逐行扫描输入 (可以是文件或管道等), 按给定的模式查找出匹配的行, 然后对这些行执行 `awk` 命令指定的操作。

q 与 `sed` 一样, `awk` 不会修改输入文件的内容。

可以使用 **重定向** 将 `awk` 的输出保存到文件中。

24

awk 的三种调用方式

U 在命令行键入命令:

```
awk [-F 字段分隔符] 'awk_script' input_file
```

若不指定字段分隔符, 则使用环境变量 `IFS` 的值 (通常为空格)

U 将 `awk` 命令插入脚本文件 `awd_script`, 然后调用:

```
awk -f awd_script input_file
```

U 将 `awk` 命令插入脚本文件, 生成 `awk` 可执行脚本文件, 然后在命令行中直接键入脚本文件名来执行。

```
#!/bin/awk -f
awk_cmd1
... ..
```

25

awk 的三种调用方式

U `awk_script` 可以由一条或多条 `awk_cmd` 组成, 每条 `awk_cmd` 各占一行。

U 每个 `awk_cmd` 由两部分组成: `/pattern/{actions}`

U `awk_cmd` 中的 `/pattern/` 和 `{actions}` 可以省略, 但不能同时省略: `/pattern/` 省略时表示对所有的输入行执行指定的 `actions`; `{actions}` 省略时表示打印匹配行。

U `awk` 命令的一般形式:

```
awk 'BEGIN {actions}
     /pattern1/{actions}
     .....
     /patternN/{actions}
     END {actions}' input_file
```

注意 BEGIN 和 END 都是大写字母。

其中 `BEGIN {actions}` 和 `END {actions}` 是可选的

26

awk 的执行过程

- ① 如果存在 `BEGIN`, `awk` 首先执行它指定的 `actions`
- ② `awk` 从输入中读取一行, 称为一条输入记录
- ③ `awk` 将读入的记录分割成数个字段, 并将第一个字段放入变量 `$1` 中, 第二个放入变量 `$2` 中, 以此类推; `$0` 表示整条记录; 字段分隔符可以通过选项 `-F` 指定, 否则使用缺省的分隔符。
- ④ 把当前输入记录依次与每一个 `awk_cmd` 中 `pattern` 比较: 如果相匹配, 就执行对应的 `actions`; 如果不匹配, 就跳过对应的 `actions`, 直到完成所有的 `awk_cmd`
- ⑤ 当一条输入记录处理完毕后, `awk` 读取输入的下一行, 重复上面的处理过程, 直到所有输入全部处理完毕。
- ⑥ 如果输入是文件列表, `awk` 将按顺序处理列表中的每个文件。
- ⑦ `awk` 处理完所有的输入后, 若存在 `END`, 执行相应的 `actions`。

27

awk 举例

```
awk '/Mar/{print $1,$3}' shipped
```

```
awk '{print $1,$3}' shipped
```

```
awk '/Mar/' shipped
```

```
awk 'BEGIN{print "Mon data"}/Mar/{print $1,$3}' shipped
```

```
awk '/Mar/{print $1,$3} END{print "OK"}' shipped
```

```
awk -F: -f awkfile1 employees2
```

28

模式匹配

Q `awk` 中的模式 (`pattern`) 匹配

① 使用正则表达式: `/rexp/`, 如 `/^A/`、`/A[0-9]*/`

`awk` 中正则表达式中常用到的元字符有:

<code>^</code>	只匹配行首 (可以看成是行首的标志)
<code>\$</code>	只匹配行尾 (可以看成是行尾的标志)
<code>*</code>	一个单字符后紧跟 <code>*</code> , 匹配 0 个或多个此字符
<code>[]</code>	匹配 <code>[]</code> 内的任意一个字符 (<code>[^]</code> 反向匹配)
<code>\</code>	用来屏蔽一个元字符的特殊含义
<code>.</code>	匹配任意单个字符
<code>str1 str2</code>	匹配 <code>str1</code> 或 <code>str2</code>
<code>+</code>	匹配一个或多个前一字符
<code>?</code>	匹配零个或一个前一字符
<code>()</code>	字符组

29

模式匹配

② 使用布尔 (比较) 表达式, 表达式的值为真时执行相应的操作 (`actions`)

| 表达式中可以使用变量 (如字段变量 `$1`, `$2` 等) 和 `/rexp/`

| 表达式中的运算符有

n 关系运算符: `<` `>` `<=` `>=` `==` `!=`

n 匹配运算符: `~` `!~`

`x ~ /rexp/` 如果 `x` 匹配 `/rexp/`, 则返回真;

`x !~ /rexp/` 如果 `x` 不匹配 `/rexp/`, 则返回真。

```
awk '$2 > 20 {print $0}' shipped
```

```
awk '$4 ~ /^6/ {print $0}' shipped
```

30

模式匹配

| 复合表达式: `&&` (逻辑与)、`||` (逻辑或)、`!` (逻辑非)

`expr1 && expr2` 两个表达式的值都为真时, 返回真

`expr1 || expr2` 两个表达式中有一个的值为真时, 返回真

`!expr` 表达式的值为假时, 返回真

```
awk '($2<20)&&($4~/^6/){print $0}' shipped
```

```
awk '($2<20)||($4~/^6/){print $0}' shipped
```

```
awk '!($4~/^6/){print $0}' shipped
```

```
awk '/^A/ && /0$/{print}' shipped
```

注: 表达式中有比较运算符时, 一般用圆括号括起来

31

字段分隔符、重定向和管道

Q 字段分隔符

`awk` 中的字段分隔符可以用 `-F` 选项指定, 缺省是空格。

```
awk '{print $1}' datafile2
```

```
awk -F: '{print $1}' datafile2
```

```
awk -F'[ :]' '{print $1}' datafile2
```

Q 重定向与管道

```
awk '{print $1, $2 > "output"}' datafile2
```

```
awk 'BEGIN{"date" | getline d; print d}'
```

32

AWK 中的操作 ACTIONS

q 操作由一条或多条语句或者命令组成，语句、命令之间用分号“;”隔开。操作中还可以使用流程控制结构的语句

q awk 命令

| **print** 输出列表：打印字符串、变量或表达式，输出列表中各参数之间用逗号隔开；若用空格隔开，打印时各输出之间没有空格

| **printf** ([格式控制符], 输出列表)：格式化打印，语法与 C 语言中的 **printf** 函数类似

| **next**：停止处理当前记录，开始读取和处理下一条记录

| **nextfile**：强迫 **awk** 停止处理当前的输入文件而处理输入文件列表中的下一个文件

| **exit**：使 **awk** 停止执行而跳出。若存在 **END** 语句，则执行 **END** 指定的 **actions**

33

AWK 语句

q awk 语句：主要是赋值语句

| 直接赋值：如果值是字符串，需加双引号。

```
awk 'BEGIN
     {x=1;y=x;z="OK";
     print "x=" x, "y=" y, "z=" z}'
```

| 用表达式赋值：

n 数值表达式：**num1 operator num2**

其中 **operator** 可以是 +, -, *, /, %, ^

当 **num1** 或 **num2** 是字符串时，**awk** 视其值为 0

n 条件表达式：**A?B:C** 当 A 为真时表达式的值为 B，否则为 C

n **awk** 也支持以下赋值操作符：

+=, -=, *=, /=, %=, ^=, ++, --

34

流控制

q awk 中的流控制结构（基本上是用 C 语言的语法）

```
| if (expr) {actions}
   [else if {actions}] (可以有多个 else if 语句)
   [else {actions}]
```

```
| while (expr) {actions}
```

```
| do {actions} while (expr)
```

```
| for (init_val;test_cond;incr_val) {actions}
```

| **break**：跳出 **for**, **while** 和 **do-while** 循环

| **continue**：跳过本次循环的剩余部分，直接进入下一轮循环

流控制结构举例：**awkfile2**

35

AWK 中的变量

q 在 **awk_script** 中的表达式中要经常使用变量。**awk** 的变量基本可以分为：字段变量，内置变量和自定义变量。

q 字段变量：**\$0, \$1, \$2, ...**

n 在 **awk** 执行过程中，字段变量的值是动态变化的。

但可以修改这些字段变量的值，被修改的字段值可以反映到 **awk** 的输出中。

n 可以创建新的输出字段，比如：当前输入记录被分割为 8 个字段，这时可以通过对变量 **\$9**（或 **\$9** 之后的字段变量）赋值而增加输出字段，**NF** 的值也将随之变化。

n 字段变量支持变量名替换。如 **\$NF** 表示最后一个字段

```
awk '{ $6=3*$2+$3; print }' shipped
```

36

内置变量

q 用于存储 **awk** 工作时的各种参数，这些变量的值会随着 **awk** 程序的运行而动态的变化，常见的有：

```
n ARGC：命令行参数个数（实际就是输入文件的数目加 1）
n ARGIND：当前被处理的文件在数组 ARGV 内的索引
n ARGV：命令行参数数组
n FILENAME：当前输入文件的文件名
n FNR：已经被 awk 读取过的记录(行)的总数目
n FS：输入记录的字段分隔符（缺省是空格和制表符）
n NF：当前行或记录的字段数
n NR：对当前输入文件而言，已被 awk 读取过的记录（行）的数目
n OFMT：数字的输出格式（缺省是 %.6g）
n OFS：输出记录的字段分隔符（缺省是空格）
n ORS：输出记录间的分隔符（缺省是 NEWLINE）
n RS：输入记录间的分隔符（缺省是 NEWLINE）
```

37

自定义变量

u 变量定义 **varname = value**

| 变量名由字母、数字和下划线组成，但不能以数字开头

| **awk** 变量无需声明，直接赋值即完成变量的定义和初始化

| **awk** 变量可以是数值变量或字符串变量

| **awk** 可以从表达式的上下文推导出变量的数据类型

u 在表达式中出现不带双引号的字符串都被视为变量

u 如果自定义变量在使用前没有被赋值，缺省值为 0 或空字符串

38

变量传递

q 如何向命令行 **awk** 程序传递变量的值

```
awk 'awk_script' var1=val1 var2=val2 ... files
```

| **var** 可以是 **awk** 内置变量或自定义变量。

| **var** 的值在 **awk** 开始对输入的第一条记录应用 **awk_script** 前传入。如果在 **awk_script** 中已经对某个变量赋值，那么命令行上的赋值无效。

| 在 **awk** 脚本程序中不能直接使用 **shell** 的变量。

| 可以向 **awk** 可执行脚本传递变量的值，与命令行类似，即

```
awk_ex_script var1=val1 var2=val2 ... files
```

```
awk '{if ($3 < ARG) print}' ARG=30 shipped
```

```
cat /etc/passwd | awk 'BEGIN {FS=":"} {if ($1==user) {print}}' user=$USER
```

39

AWK 内置函数

① 常见 **awk** 内置数值函数

n **int(x)**：取整数部份，朝 0 的方向做舍去。

n **sqr(x)**：正的平方根。

n **exp(x)**：以 **e** 为底的指数函数。

n **log(x)**：自然对数。

n **sin(x)**、**cos(x)**：正弦、余弦。

n **atan2(y,x)**：求 **y/x** 的 **arctan** 值，单位是弧度。

n **rand()**：得到一个随机数（平均分布在 0 和 1 之间）

n **srand(x)**：设定产生随机数的 **seed** 为 **x**

40

内置字符串函数

② 常见 awk 内置字符串函数

n index(str, substr): 返回子串 `substr` 在字符串 `str` 中第一次出现的位置, 若找不到, 则返回值为 0

```
awk 'BEGIN{print index("peanut","an")}'
```

n length(str): 返回字符串 `str` 的字符个数

n match(str, rexp): 返回模式 `rexp` 在字符串 `str` 中第一次出现的位置, 如果 `str` 中不包含 `rexp`, 则返回值 0

```
awk 'BEGIN{print match("hello",/l[^l]/)}'
```

n sprintf(format, expl, ...): 返回一个指定格式的表达式, 格式 `format` 与 `printf` 的打印格式类似 (不在屏幕上输出)

41

内置字符串函数

n sub(rexp, sub_str, target): 在目标串 `target` 中寻找第一个能够匹配正则表达式 `rexp` 的子串, 并用字符串 `sub_str` 替换该子串。若没有指定目标串, 则在整个记录中查找

```
awk 'BEGIN{str="water,water";sub(/at/, "ith", str);\nprint str}'
```

n gsub(rexp, sub_str, target): 与 `sub` 类似, 但 `gsub` 替换所有匹配的子串, 即全局替换。

n substr(str, start, len): 返回 `str` 的从指定位置 `start` 开始长度为 `len` 个字符的子串, 如果 `len` 省略, 则返回从 `start` 位置开始至结束位置的所有字符。

```
awk 'BEGIN{print substr("awk sed grep",5)}'
```

42

内置字符串函数

n split(str, array, fs): 使用由 `fs` 指定的分隔符将字符串 `str` 拆分成一个数组 `array`, 并返回数组的下标数

```
awk 'BEGIN{split("11/15/2005", date, "/"); \nprint date[2]}'
```

n tolower(str): 将字符串 `str` 中的大写字母改为小写字母

```
awk 'BEGIN{print tolower("MiXeD CaSe 123")}'
```

n toupper(str): 将字符串 `str` 中的小写字母改为大写字母

43

内置系统函数

③ 常见 awk 内置系统函数

n close(filename)

将输入或输出的文件 `filename` 关闭。

n system(command)

此函数允许调用操作系统的指令, 执行完毕后返回 `awk`

```
awk 'BEGIN {system("ls")}'
```

44

AWK 的自定义函数

```
function fun_name (parameter_list) {\n    body-of-function\n    // 函数体, 是 awk 语句块\n}
```

- ! `parameter_list` 是以逗号分隔的参数列表
- ! 自定义函数可以在 `awk` 程序的任何地方定义
- ! 函数名可包括字母、数字、下划线, 但**不可以数字开头**
- ! 调用自定义的函数与调用内置函数的方法一样

```
awk '{print "sum =", SquareSum($2,$3)} \nfunction SquareSum(x,y) {\ \ns    sum=x*x+y*y ; return sum \n}' shipped
```

45

AWK 中的数组

q 数组使用前, 无需预先定义, 也不必指定数组元素个数

q 访问数组的元素

经常使用循环来访问数组元素

```
for (element in array_name ) print \narray_name[element]
```

```
awk 'BEGIN{print \n    split("123#456#789", mya, "#"); \n    for (i in mya) {print mya[i]}}'
```

46

字符串屏蔽

q 使用字符串或正则表达式时, 有时需要在输出中加入一新行或一个特殊字符。这时就需要字符串屏蔽。

q `awk` 中常用的字符串屏蔽序列

<code>\b</code>	退格键	<code>\t</code>	<code>tab</code> 键
<code>\f</code>	走纸换页	<code>\n</code>	换行
<code>\r</code>	回车键	<code>\ddd</code>	八进制值 ASCII 码
<code>\c</code>	任意其他特殊字符。如: <code>\\</code> 为反斜线符号		

```
awk 'BEGIN{print \n    "\nMay\tDay\n\nMay\t\t104\t141\t171"}'
```

47

AWK 输出函数 PRINTF

q 基本上和 C 语言的语法类似

```
printf([格式控制符], 参数列表)
```

n 参数列表中可以有变量、数值数据或字符串, 用逗号隔开

n 格式控制符: `%[-][w][.p]fmt`

- ! `%`: 标识一个格式控制符的开始, 不可省略
- ! `-`: 表示参数输出时左对齐, 可省略
- ! `w`: 一个数字, 表示参数输出时占用域的宽度, 可省略
- ! `.p`: `p` 是一个数值, 表示最大字符串长度或小数位数, 可省略
- ! `fmt`: 一个小写字母, 表示输出参数的数据类型, 不可省略

48

AWK 输出函数 PRINTF

U 常见的 fmt

c	ASCII 字符	d	整数
f	浮点数, 如 12.3	e	浮点数, 科学记数法
g	自动决定用 e 或 f	s	字符串
o	八进制数	x	十六进制数

```
echo "65" | awk '{ printf "%c\n", $0 }'
```

```
awk 'BEGIN{printf "%.4f\n", 999}'
```

```
awk 'BEGIN{printf \
"2 number:%8.4f %8.2f", 999, 888}'
```

49

注意事项

Q 为了避免碰到 **awk** 错误, 要注意以下事项:

- n 确保整个 **awk_script** 用单引号括起来
- n 确保 **awk_script** 内所有引号都成对出现
- n 确保用花括号括起动作语句, 用圆括号括起条件语句
- n 如果使用字符串, 要保证字符串被双引号括起来 (在模式中除外)

Q **awk** 语言学起来可能有些复杂, 但使用它来编写一行命令或小脚本并不太难。**awk** 是 **shell** 编程的一个重要工具。在 **shell** 命令或编程中, 可以使用 **awk** 强大的文本处理能力。

50