

AWK

awk 是一种程序语言. 它具有一般程序语言常见的功能.

因**awk**语言具有某些特点, 如: 使用直译器(Interpreter)不需先行编译; 变量无类型之分(Typeless), 可使用文字当数组的下标(Associative Array)...等特色. 因此, 使用**awk**撰写程序比起使用其它语言更简洁便利且节省时间. **awk**还具有一些内建功能, 使得**awk**擅于处理具数据行(Record), 字段(Field)型态的资料; 此外, **awk**内建有pipe的功能, 可将处理中的数据传送给外部的 Shell命令加以处理, 再将Shell命令处理后的数据传回**awk**程序, 这个特点也使得**awk**程序很容易使用系统资源.

由于**awk**具有上述特色, 在问题处理的过程中, 可轻易使用**awk**来撰写一些小工具; 这些小工具并非用来解决整个大问题, 它们只扮演解决个别问题过程的某些角色, 可藉由Shell所提供的pipe将数据按需要传送给不同的小工具进行处理, 以解决整个大问题.

模式和动作

单引号

输入文件

任何 `awk` 语句都由模式和动作组成。

模式可以是任何条件语句或复合语句或正则表达式。模式包括两个特殊字段 `BEGIN` 和 `END`。动作即对数据的操作如打印、计算等

```
[songdy@login-0-0 songdy] $ awk '/chrX/{print} example.txt
chrX      152067      A      27      1.00000
chrX      548568      G      24      1.00000
chrX      1084630     A      99      1.00000
chrX      1318414     C      99      1.00000
```

提示符

域和记录

模式

动作

`awk` 执行时，其浏览域标记为 `$1`, `$2`...`$n`。这种方法称为域标识。`$0` 代表整条记录。

```
chr10      2085      T      64      0.00207313
chr10      2094      C      44      0.0550898
chr10      37435     G      76      0.189062
chr10      38639     G      45      0.474901
chr11      1102      T      33      1.00000
```

域

记录

调用awk

第一种是命令行方式：

awk **【-F filed-separator】** 'commands' input-files

awk 'BEGIN{print "Good afternoon "} {total += \$1} END{print total}' input-file

第二种方法是将所有a w k命令插入一个文件，并使a w k程序可执行，然后用a w k命令解释器作为脚本的首行，以便通过键入脚本名称来调用它。

chmod u+x print.awk

./print.awk input-file

```
#!/bin/awk -f
{print $0}
```

第三种方式是将所有的a w k命令插入一个单独文件，然后调用：

awk -f print.awk input-file

```
{print $0}
```

保存a w k输出

有两种方式保存s h e l l提示符下a w k脚本的输出。

A、使用输出重定向符号>文件名，下面的例子重定向输出到文件，使用这种方法要注意，显示屏上不会显示输出结果。

```
awk '{print $0}' chr10.snp >chr10.newsnp
```

B、第二种方法是使用t e e命令，在输出到文件的同时输出到屏幕。

```
awk '{print $0}' chr10.snp | tee chr10.newsnp
```

常见应用

打印所有记录和打印指定域

awk '{print \$0}' example.txt

```
chr13 1053 C 99 0.188428
chr13 1329 T 99 0.730084
chr13 1902 T 41 0.637185
chr13 1911 G 68 0.0432504
chr14 10362 A 99 0.0405247
chr14 19442 G 99 0.347385
chr14 19456 T 99 0.481283
chr14 46375 A 30 0.474725
chr15 504 G 99 0.286869
chr15 1207 G 87 0.344410
chr15 1289 A 37 0.175097
chr15 1417 G 92 0.707854
chr16 14876 G 58 1.00000
chr16 38736 A 24 0.286713
chr16 58784 T 26 0.664835
```

awk '{print \$1,\$4}' example.txt

```
chr2 99
chr2 73
chr3 25
chr3 22
chr3 99
chr3 99
chr4 41
chr4 26
chr4 32
chr4 29
chr5 99
chr5 99
chr5 31
chr5 26
```

\$1 \$4

打印报告头和信息尾

```
awk 'BEGIN{print "chr\tpos\tbase\tnum\tp-value"} /chrX/ {print}  
END{print "end of the data"}' example.txt
```

报告头

chr	pos	base	num	p-value
chrX	152067	A	27	1.00000
chrX	548568	G	24	1.00000
chrX	1084630	A	99	1.00000
chrX	1318414	C	99	1.00000
end of the data				

信息尾

BEGIN和END

BEGIN和END部分在awk中都仅执行一次且有各自的用途

如BEGIN用在程序一开始时, 改变awk切割字段的方式、程序一开始时, 改变awk分隔数据行的方式、设定变量的起始值、印出一行 title、不需要读入任何数据行, 而END用来打印结尾信息或者用来输出统计信息等

```
awk '{total+=$6} END{print total}' *.single >single
```

```
awk '{total+=$6} END{print total}' *.soap >soap
```

```
awk 'BEGIN{total=100} {total += $5} END{print "total:",total}' example.txt
```

```
total: 151.39
```

awk 中的数学运算符(Arithmetic Operators)

+(加), -(减), *(乘), /(除), %(求余数), ^(指数) 与 C 语言中用法相同

awk 中的赋值运算符(Assignment Operators)

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

x += 5 的意思为 $x = x + 5$, 其余类推.

awk 中的条件运算符(Conditional Operator)

语 法: 判断条件 ? value1 : value2

若 判断条件 成立(true) 则返回 value1, 否则返回 value2.

awk 中的逻辑运算符(Logical Operators)

&&(and), ||(or), !(not)

Extended Regular Expression 中使用 "|" 表示 or 请勿混淆.

awk 中的关系运算符(Relational Operators)

>, >=, <, <=, ==, !=, ~, !~

awk 中其它的运算符

+(正号), -(负号), ++(Increment Operator), --(Decrement Operator)

正则表达式

正则表达式同perl中的正则表达式

例：计算文件的大小（正则式的应用）

```
[songdy@login-0-0 songdy]$ ll
total 1011
-rw-r--r-- 1 songdy pap 322738 Aug 13 09:28 bash.ref
-rw-r--r-- 1 songdy pap 21285 Aug 13 09:28 chromosome_graph_wb.pl
drwxr-xr-x 6 songdy pap 60 Aug 13 09:30 drawingsoft
-rwxr-xr-x 1 songdy pap 1563 Aug 19 10:05 dup.pl
-rw-r--r-- 1 songdy pap 2986 Aug 18 15:19 example.txt
-rw-r--r-- 1 songdy pap 637 Aug 17 10:24 gcPercentage.pl
-rw-r--r-- 1 songdy pap 68 Aug 16 19:17 gcPercentage.sh
drwxr-xr-x 5 songdy pap 43 Aug 18 10:06 HUM
-rw-r--r-- 1 songdy pap 2344 Aug 16 15:36 paixu.out
-rw-r--r-- 1 songdy pap 647 Aug 16 18:07 paixu.pl
drwxr-xr-x 3 songdy pap 950 Aug 17 17:17 programs
drwxr-xr-x 13 songdy pap 388 Aug 17 15:50 Resequencing
-rw-r--r-- 1 songdy pap 2352 Aug 16 14:51 result.txt
-rw-r--r-- 1 songdy pap 1480 Aug 13 10:21 soap2.20.ref
-rw-r--r-- 1 songdy pap 4101 Aug 16 10:01 staticsAll.pl
-rw-r--r-- 1 songdy pap 5023 Aug 16 12:16 staticsAll_V1.0.pl
drwxr-xr-x 2 songdy pap 244 Aug 13 09:29 Study_only
drwxr-xr-x 2 songdy pap 317 Aug 17 18:04 tmp
[songdy@login-0-0 songdy]$ ll | awk '/^[^d]/ {print $9"\t"$5}(total+=$5)END{print "total KB:"total}'

bash.ref          322738
chromosome_graph_wb.pl 21285
dup.pl 1563
example.txt       2986
gcPercentage.pl   637
gcPercentage.sh   68
paixu.out         2344
paixu.pl          647
result.txt        2352
soap2.20.ref      1480
staticsAll.pl     4101
staticsAll_V1.0.pl 5023
total KB:367226
```

匹配非目录文件

末尾输出统计信息

复合模式或复合操作符用于形成复杂的逻辑操作，复杂程度取决于编程者本人

&& AND : 语句两边必须同时匹配为真。

|| OR: 语句两边同时或其中一边匹配为真。

! 非求逆

awk '{if(\$4 > 90 && \$5 > 0.5) print}'

example.txt

awk '{if(\$4 > 90 || \$5 > 0.5) print}'

example.txt

```
chr11 369195 C 99 0.624061
chr13 1399 T 99 0.730084
chr15 1417 G 92 0.707854
chr1 223244 C 99 0.576241
chr1 223274 G 99 0.710428
chr1 223288 C 99 0.519978
chr2 46926 T 99 0.555855
chr3 827 C 99 0.625830
chrMT 9461 T 99 1.00000
chrX 1084630 A 99 1.00000
chrX 1318414 C 99 1.00000
chrY 902644 C 99 1.00000
chrY 1155322 G 99 1.00000
```

```
chr9 105991 G 20 1.00000
chr9 121306 C 99 0.389979
chr9 124859 A 20 0.545455
chrMT 9461 T 99 1.00000
chrUn 587388 T 26 1.00000
chrUn 647716 A 73 0.737429
chrX 152067 A 27 1.00000
chrX 548568 G 24 1.00000
chrX 1084630 A 99 1.00000
chrX 1318414 C 99 1.00000
chrY 713894 A 27 1.00000
chrY 902644 C 99 1.00000
chrY 1041140 G 27 1.00000
chrY 1155322 G 99 1.00000
```

显示、修改文本域、显示修改记录、增加新的域和列

```
[songdy@login-0-0 songdy]$ awk '(if($1~/chrX/){OFS="\t";print})' example.txt
```

```
chrX 152067 A 27 1.00000
chrX 548568 G 24 1.00000
chrX 1084630 A 99 1.00000
chrX 1318414 C 99 1.00000
```

正则匹配，选取文本

```
[songdy@login-0-0 songdy]$ awk '(if($1~/chrX/){OFS="\t";$4*=$4;print})' example.txt
```

```
chrX 152067 A 729 1.00000
chrX 548568 G 576 1.00000
chrX 1084630 A 9801 1.00000
chrX 1318414 C 9801 1.00000
```

修改域4 注意：没有修改源文件的内容

```
[songdy@login-0-0 songdy]$ awk '(if($1~/chrX/){OFS="\t";$6=$2+$4;print})' example.txt
```

```
chrX 152067 A 27 1.00000 152094
chrX 548568 G 24 1.00000 548592
chrX 1084630 A 99 1.00000 1084729
chrX 1318414 C 99 1.00000 1318513
```

增加一个域6

```
[songdy@login-0-0 songdy]$ awk '(if($1~/chrX/){OFS="\t";$6=$2+$4;print})' example.txt|awk '{total+=$6}END{print total}'
3103928
```

```
[songdy@login-0-0 songdy]$ awk '(if($1~/chrX/){OFS="\t";$6=$2+$4;print})' example.txt|awk '{total+=$6;print}END{print total}'
```

```
chrX 152067 A 27 1.00000 152094
chrX 548568 G 24 1.00000 548592
chrX 1084630 A 99 1.00000 1084729
chrX 1318414 C 99 1.00000 1318513
```

增加一行信息

```
3103928
```

awk内置变量

ARGC 命令行参数个数 ARGV 命令行参数排列

ENVIRON 支持队列中系统环境变量的使用

FILENAME 浏览的文件名 FNR 浏览文件的记录数

FS 设置输入域分隔符，等价于命令行-F选项

NF 浏览记录的域个数 NR 已读的记录数

OFS 输出域分隔符 ORS 输出记录分隔符

RS 控制记录分隔符

awk 'END{print FILENAME,FNR,NR,NF}' example.txt

连续读入多个文件的时候FNR和NR会有区别为什么？

```
example.txt 89 89 5
```

```
[songdy@login-0-0 soap]$ pwd
```

```
/ifs2/PAP/songdy/Resequencing/work/soap
```

```
[songdy@login-0-0 soap]$ echo $PWD | awk -F/ '{print $NF}'
```

```
soap
```

```
[songdy@compute-0-34 20] $ ls
B10snJ92_chr10.cns.Q20 B10snJ92_chr15.cns.Q20 B10snJ92_chr1.cns.Q20 B10snJ92_chr6.cns.Q20 B10snJ92_chrUn.cns.Q20
B10snJ92_chr11.cns.Q20 B10snJ92_chr16.cns.Q20 B10snJ92_chr2.cns.Q20 B10snJ92_chr7.cns.Q20 B10snJ92_chrX.cns.Q20
B10snJ92_chr12.cns.Q20 B10snJ92_chr17.cns.Q20 B10snJ92_chr3.cns.Q20 B10snJ92_chr8.cns.Q20 B10snJ92_chrY.cns.Q20
B10snJ92_chr13.cns.Q20 B10snJ92_chr18.cns.Q20 B10snJ92_chr4.cns.Q20 B10snJ92_chr9.cns.Q20
B10snJ92_chr14.cns.Q20 B10snJ92_chr19.cns.Q20 B10snJ92_chr5.cns.Q20 B10snJ92_chrMT.cns.Q20
```

```
[songdy@compute-0-34 20] $ head -3 B10snJ92_chr10.cns.Q20 | awk 'BEGIN(OFS=":") {print $1,$2,$3,$4,$5}'
```

```
chr10:2085:T:Y:64
chr10:2094:C:Y:44
chr10:37435:G:R:76
```

改变输出域的分割符

```
[songdy@compute-0-34 20] $ head -3 B10snJ92_chr10.cns.Q20 | awk 'BEGIN(OFS=":";ORS="\t"){print $1,$2,$3,$4,$5}'
```

```
chr10:2085:T:Y:64 chr10:2094:C:Y:44 chr10:37435:G:R:76 [songdy@compute-0-34 20] $
```

```
[songdy@compute-0-34 20] $ for f in *.Q20;do awk 'BEGIN(OFS="\t";ORS="\n")END{print FILENAME,NR,FNR}' $f;done
```

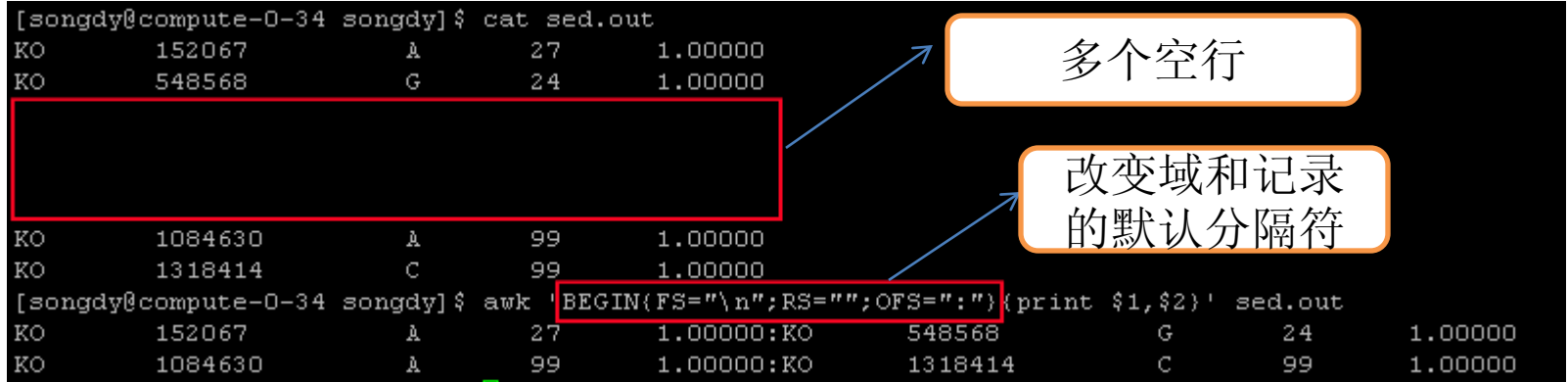
```
B10snJ92_chr10.cns.Q20 3141 3141
B10snJ92_chr11.cns.Q20 52522 52522
B10snJ92_chr12.cns.Q20 3072 3072
B10snJ92_chr13.cns.Q20 17996 17996
B10snJ92_chr14.cns.Q20 6767 6767
B10snJ92_chr15.cns.Q20 7108 7108
B10snJ92_chr16.cns.Q20 3173 3173
B10snJ92_chr17.cns.Q20 2816 2816
B10snJ92_chr18.cns.Q20 2616 2616
B10snJ92_chr19.cns.Q20 1318 1318
B10snJ92_chr1.cns.Q20 9554 9554
B10snJ92_chr2.cns.Q20 9729 9729
B10snJ92_chr3.cns.Q20 4125 4125
B10snJ92_chr4.cns.Q20 50352 50352
B10snJ92_chr5.cns.Q20 5274 5274
B10snJ92_chr6.cns.Q20 4419 4419
B10snJ92_chr7.cns.Q20 4796 4796
B10snJ92_chr8.cns.Q20 36272 36272
B10snJ92_chr9.cns.Q20 7196 7196
B10snJ92_chrMT.cns.Q20 1 1
B10snJ92_chrUn.cns.Q20 1956 1956
B10snJ92_chrX.cns.Q20 1905 1905
B10snJ92_chrY.cns.Q20 561 561
```

改变输出域的分隔符，同时改变输出记录的分隔符

处理多行的数据

awk 每次从数据文件中只读取一数据进行处理.awk是依照其内建变量 RS(Record Separator) 的定义将文件中的数据分隔成一行一行的Record. RS 的默认值是 “\n”(跳行符号), 故平常awk中一行数据就是一笔 Record. 但有些文件中一笔Record涵盖了多行数据, 这种情况下不能再以 “\n” 来分隔Records. 最常使用的方法是相邻的Records之间改以一个空白行 来隔开. 在awk程序中, 令 RS = ""(空字符串)后,awk把会空白行当成来文件中Record的分隔符.

```
[songdy@compute-0-34 songdy]$ cat sed.out
KO      152067      A      27      1.00000
KO      548568      G      24      1.00000
KO      1084630     A      99      1.00000
KO      1318414     C      99      1.00000
[songdy@compute-0-34 songdy]$ awk 'BEGIN{FS="\n";RS="";OFS=":"}(print $1,$2)' sed.out
KO      152067      A      27      1.00000:KO      548568      G      24      1.00000
KO      1084630     A      99      1.00000:KO      1318414     C      99      1.00000
```



多个空行

改变域和记录的默认分隔符

如何读取命令行上的参数

大部分的应用程序都允许使用者在命令之后增加一些选择性的参数.执行awk时这些参数大部分用于指定数据文件文件名,有时希望在程序中能从命令行上得到一些其它用途的数据.

ARGC: 为一整数. 代表命令行上, 除了选项-v, -f 及其对应的参数之外所有参数的数目; **ARGV[]**: 为一字符串数组. ARGV[0],ARGV[1],...ARGV[ARGC-1].分别代表命令行上相对应的参数.

```
[songdy@compute-0-34 songdy]$ awk 'BEGIN{for(i=0;i<ARGC;i++)printf("ARGV[%d]\t%-s\n"),i,ARGV[i]}' generateStatics.V2.pl examplebck.txt bash.ref gcPercentage.sh paixu.out soap2.20.ref today_rpt3 sed.out
ARGV[0] awk
ARGV[1] generateStatics.V2.pl
ARGV[2] examplebck.txt
ARGV[3] bash.ref
ARGV[4] gcPercentage.sh
ARGV[5] paixu.out
ARGV[6] soap2.20.ref
ARGV[7] today_rpt3
ARGV[8] sed.out
```

awk强大的内置字符串函数

<code>gsub(r,s)</code>	在整个\$0中用s替代r
<code>gsub(r,s,t)</code>	在整个t中用s替代r
<code>index(s,t)</code>	返回s中字符串t的第一位置
<code>length(s)</code>	返回s长度
<code>match(s,r)</code>	测试s是否包含匹配r的字符串
<code>split(s,a,fs)</code>	在fs上将s分成序列a
<code>sprintf(fmt,exp)</code>	返回经fmt格式化后的exp
<code>sub(r,s)</code>	用\$0中最左边最长的子串代替s
<code>substr(s,p)</code>	返回字符串s中从p开始的后缀部分
<code>substr(s,p,n)</code>	返回字符串s中从p开始长度为n的后缀部分

```
[songdy@login-0-0 songdy]$ awk 'gsub(/chrX/, "KO")' example.txt
KO      152067      A      27      1.00000
KO      548568      G      24      1.00000
KO      1084630     A      99      1.00000
KO      1318414     C      99      1.00000
[songdy@login-0-0 songdy]$ awk 'BEGIN{print index("sounds good!", "oo")}'
9
[songdy@login-0-0 songdy]$ awk '$2=="1084630" {print length($2) "\t" $2}' example.txt
7      1084630
```


格式化输出

1、awk printf修饰符

- 左对齐

Width 域的步长，用0表示0步长

.prec 最大字符串长度，或小数点右边的位数

2、awk printf格式

%c ASCII字符 %d 整数

%e 浮点数，科学记数法

%f 浮点数，例如（123.44）

%g awk 决定使用哪种浮点数转换e或者f

%o 八进制数 %s 字符串 %x 十六进制数

```
[songdy@login-0-0 songdy]$ echo "65" | awk '{printf "%c\n", $0}'  
A  
[songdy@login-0-0 songdy]$ awk 'BEGIN{printf "%c\n", 65}'  
A  
[songdy@login-0-0 songdy]$ awk 'BEGIN{printf "%f\n", 65}'  
65.000000  
[songdy@login-0-0 songdy]$ awk '/chrX/{printf "%s\t%-10s\n", $1, $2}' example.txt  
chrX      152067  
chrX      548568  
chrX      1084630  
chrX      1318414
```

向一行a w k命令传值

awk 命令变量=输入文件值

```
[songdy@login-0-0 songdy]$ awk '{if($2>POS) print}' POS=1041140 example.txt
```

chrX	1084630	A	99	1.00000
chrX	1318414	C	99	1.00000
chrY	1155322	G	99	1.00000

变量赋值

```
[songdy@login-0-0 songdy]$ who | awk '{print $1"\tis logged on"}'
```

liurmn	is logged on
liurmn	is logged on
liurmn	is logged on
liurmn	is logged on
chensong	is logged on
liuke	is logged on
wuxiaole	is logged on
shaolibi	is logged on
zhuying	is logged on
chenzhsh	is logged on
yanzhixi	is logged on
liuyanbo	is logged on
sunrenji	is logged on
chenwm	is logged on
songdy	is logged on
liurmn	is logged on
lixia	is logged on
sunrenji	is logged on
liuke	is logged on

管道传值

awk数组和awk脚本

`split`函数，使用它将元素划分进一个数组，例如

`split`返回数组`myarray`下标数。

数组使用前，不必定义，也不必指定数组元素个数，而且可以使用字符串当数组的下标（`index`）。经常使用循环来访问数组。下面是一种循环类型的基本结构：

```
[songdy@login-0-0 songdy]$ awk 'BEGIN{print split("123#456#789",myarray,"#")}'  
3  
[songdy@login-0-0 songdy]$ awk 'BEGIN{print split("123#456#789",myarray,"#")}END  
{for(i in myarray){printf("myarray[%d],%d\n"),i,myarray[i]}}' sed.out  
3  
myarray[1],123  
myarray[2],456  
myarray[3],789
```

遍历数组，逐个输出数组中的元素

返回数组`myarray`，数组中有三个元素

awk脚本就是把命令行上的单引号之间的部分写到脚本中就可以了
例如 `awk '{print $0}' input-file`

写到脚本中如下

```
#!/bin/sh -f  
{print $0}
```

这样就可以调用了
详见下例

字符串下标

```
[songdy@compute-O-34 Study_only]$ cat course.txt
```

```
Mary O.S. Arch. Discrete
Steve D.S. Algorithm Arch.
Wang Discrete Graphics O.S.
Lisa Graphics A.I.
Lily Discrete Algorithm
```

```
[songdy@compute-O-34 Study_only]$ cat course.awk
```

```
#!/bin/awk -f
BEGIN{printf("%10s\t%s\n", "Course", "numbers")}
{for(i=2;i<=NF;i++) Number[$i]++}
END{for(course in Number)printf("%10s\t%d\n", course, Number[course])}
```

```
[songdy@compute-O-34 Study_only]$ ll course.awk
```

```
-rw-r--r-- 1 songdy pap 167 Aug 19 13:58 course.awk
```

```
[songdy@compute-O-34 Study_only]$ awk -f course.awk course.txt
```

```
Course      numbers
Discrete    3
  O.S.      2
  A.I.      1
  D.S.      1
Graphics    2
Algorithm   2
  Arch.     2
```

加可执行权限

```
[songdy@compute-O-34 Study_only]$ ./course.awk course.txt
```

```
-bash: ./course.awk: Permission denied
```

```
[songdy@compute-O-34 Study_only]$ chmod u+x course.awk
```

```
[songdy@compute-O-34 Study_only]$ ll course.awk
```

```
-rwxr--r-- 1 songdy pap 167 Aug 19 13:58 course.awk
```

```
[songdy@compute-O-34 Study_only]$ ./course.awk course.txt
```

```
Course      numbers
Discrete    3
  O.S.      2
  A.I.      1
  D.S.      1
Graphics    2
Algorithm   2
  Arch.     2
```

awk 程序中使用 Shell 命令

awk程序中允许呼叫Shell指令. 并提供管道解决awk与系统间数据传递的问题. 所以awk很容易使用系统资源.

```
[songdy@compute-0-34 songdy]$ awk 'BEGIN {while ( "who" | getline ) n++;print n}'  
132  
[songdy@compute-0-34 songdy]$ cat >count.awk  
BEGIN {  
while ( "who" | getline )  
n++  
print n  
}  
  
[songdy@compute-0-34 songdy]$ awk -f count.awk  
132
```

自定义函数

awk 中亦允许使用者自定义函数. 函数定义方式请参考本程序,
function 为 awk 的保留字.HM_to_M() 这函数负责将所传入之小时
及分钟数转换成以分钟为单位. 使用者自定义函数时,有许多细节须留
心

```

[songdy@compute-0-35 songdy]$ cat testdata.txt
1034 7:26
1025 7:27
1101 7:32
1006 7:45
1012 7:46
1028 7:49
1051 7:51
1029 7:57
1042 7:59
1008 8:01
1052 8:05
1005 8:12

[songdy@compute-0-35 songdy]$ cat testdata.awk
#!/bin/sh
awk '
BEGIN {
FS= "[ \t:]+"
"date" | getline
print " Today is " , $2, $3 > "today_rpt3"
print "=====>" > "today_rpt3"
print " ID Number Arrival Time" > "today_rpt3"
close( "today_rpt3" )
}
{
arrival = HM_to_M($2, $3)
printf(" %s %s:%s %s\n", $1, $2, $3, arrival > 480 ? "*" : " ") | "sort -k 1 >>today_rpt3"
total += arrival
}
END {
close("today_rpt3")
close("sort -k 1 >> today_rpt3")
printf(" Average arrival time : %d:%d\n", total/NR/60, (total/NR)%60 ) >> "today_rpt3"
}
function HM_to_M( hour, min ){
return hour*60 + min
}
' $*

[songdy@compute-0-35 songdy]$ ./testdata.awk testdata.txt
[songdy@compute-0-35 songdy]$ cat today_rpt3
Today is Aug 24
=====
ID Number Arrival Time
Average arrival time : 7:49
1005 8:12 *
1006 7:45
1008 8:01 *
1012 7:46
1025 7:27
1028 7:49
1029 7:57
1034 7:26
1042 7:59
1051 7:51
1052 8:05 *
1101 7:32

```

修改字段分隔符

调用自定义函数

三目操作符

自定义函数

sort排序


```
[songdy@login-0-0 soap]$ head -10 100614_I112_FC201MDABXX_L2_MOUmftRBBDIAAPE_1.fq.soap | awk '{print $NF}'
99T
100
100
100
100
99G
44A55
0A99
100
99T
100
```

打印soap文件的最后一列

```
[songdy@login-0-0 soap]$ head -10 100614_I112_FC201MDABXX_L3_MOUmftRBBDIABPE_1.fq.soap | awk '{print $NF}' | awk 'gsub(/[^ATCG]/,"") {print}'
A
```

去除所有的非ATCG的字符

```
A
C
A
```

统计所有的ATCG字符

```
CG

[songdy@login-0-0 soap]$ head -10 100614_I112_FC201MDABXX_L3_MOUmftRBBDIABPE_1.fq.soap | awk '{print $NF}' | awk 'gsub(/[^ATCG]/,"") {print}' | awk '{total+=length($0)}END{print total}'
6
```

```
[songdy@login-0-0 soap]$ for f in *.soap;do head -10000 $f | awk '{print $NF}' |
awk 'gsub(/[^ATCG]/,"") {print}' | awk '{total+=length($0)}END{print total}';echo
$f;done
3854
100614_I112_FC201MDABXX_L2_MOUmftRBBDIABPE_1.fq.soap
3830
100614_I112_FC201MDABXX_L3_MOUmftRBBDIABPE_1.fq.soap
7320
100704_I137_FC200T7ABXX_L6_MOUmftRBBDIABPE_1.fq.soap
7488
100704_I137_FC200T7ABXX_L7_MOUmftRBBDIABPE_1.fq.soap
7230
100704_I137_FC200T7ABXX_L8_MOUmftRBBDIABPE_1.fq.soap
```

for循环实现多文件处理

流程控制指令

if 指令

语法

if (表达式) 语句1 [else 语句2]

范例：

```
if( $1 > 25 )print "The 1st field is larger than 25"
```

```
else print "The 1st field is not larger than 25"
```

(a)与 C 语言中相同, 若 表达式 计算(evaluate)后之值不为 0 或空字符串, 则执行 语句1; 否则执行 语句2.

(b)进行逻辑判断的表达式所返回的值有两种, 若最后的逻辑值为true, 则返回1, 否则返回0.

(c)语法中else 语句2 以[] 前后括住表示该部分可视需要而予加入或省略.

while 指令

语法：

while(表达式) 语句

范例：

```
while( match(buffer,/ [0-9]+\ .c/ ) ){  
  print "Find :" substr( buffer,RSTART, RLENGTH)  
  buff = substr( buffer, RSTART + RLENGTH)  
}
```

上列范例找出 **buffer** 中所有能匹配 `/[0-9]+.c/`(数字之后接上 ".c"的所有子字符串).

范例中 **while** 以函数 `match()`所返回的值做为判断条件. 若**buffer** 中还含有匹配指定条件的子字符串(`match`成功), 则 `match()`函数返回1,**while** 将持续进行其后的语句.

do-while 指令

语法：

do 语句 while(表达式)

范例：

```
do{
```

```
print "Enter y or n ! "
```

```
getline data
```

```
} while( data !~ /^[YyNn]$/)
```

(a) 上例要求用户从键盘上输入一个字符, 若该字符不是Y, y, N, 或 n则会不停执行该循环,直到读取正确字符为止.

(b)do-while 指令与 while 指令 最大的差异是 : do-while 指令会先执行statement 而后再判断是否应继续执行. 所以, 无论如何其 statement 部分至少会执行一次.

for Statement 指令(一)

语法：

for(variable in array) statement

范例：执行下列命令

```
awk      'BEGIN{ X[1]= 50; X[2]= 60; X["last"]= 70
```

```
for( any in X )printf("X[%s] = %d\n", any, X[any] ) }'
```

结果输出：

```
X[last] = 70
```

```
X[1] = 50
```

```
X[2] = 60
```

(a)这个 for 指令, 专用以查找数组中所有的下标值, 并依次使用所指定的变量予以记录. 以本例而言, 变量 **any** 将逐次代表 "last", 1 及 2 .

(b)以这个 for 指令, 所查找出的下标之值彼此间并无任何次续关系.

for Statement 指令(二)

语法：

```
for(expression1; expression2; expression3) statement
```

范例：

```
for( i =1; i< =10; i++) sum = sum + i
```

说明：

(a)上列范例用以计算 1 加到 10 的总和.

(b)expression1 常用于设定该 for 循环的起始条件, 如上例中的 i=1

expression2 用于设定该循环的停止条件, 如上例中的 i <= 10

expression3 常用于改变 counter 之值, 如上例中的 i++

break 指令

break 指令用以强迫中断(跳离) **for**, **while**, **do-while** 等循环.

范例 :

```
while( getline < "datafile" > 0 )  
{if( $1 == 0 )      break  
else      print $2 / $1  
}
```

上例中, **awk** 不断地从文件 **datafile** 中读取资料, 当**\$1**等于0时,就停止该执行循环.

continue 指令

循环中的 statement 进行到一半时, 执行 continue 指令来略过循环中尚未执行的 statement.

范例 :

```
for( index in X_array)
{if( index !~ /[0-9]+/ ) continue
print "There is a digital index", index
}
```

上例中若 index 不为数字则执行 continue, 故将略过(不执行)其后的指令. 需留心 continue 与 break 的差异 : 执行 continue 只是掠过其后未执行的 statement, 但并未跳离开该循环.

next 指令

执行 next 指令时, awk 将掠过位于该指令(next)之后的所有指令(包括其后的所有Pattern{ Actions }), 接著读取下一笔数据行,继续从第一个 Pattern Actions} 执行起.

范例 :

```
/^[ \t]*$/ { print "This is a blank line! Do nothing here !"    next }
```

```
$2 != 0 { print $1, $1/$2 }
```

上例中, 当 awk 读入的数据行为空白行时(match /^[\t]*\$/),除打印消息外只执行 next, 故awk 将略过其后的指令, 继续读取下一笔资料, 从头(第一个 Pattern{ Actions })执行起.

exit 指令

执行 exit 指令时, awk 将立刻跳离(停止执行)该awk程序.

`awk`语言学起来可能有些复杂，尤其是不太容易看懂的错误提示，但使用它来编写一行命令或小脚本并不太难。这里只讲述了`awk`的最基本功能，相信大家已经掌握了`awk`的基本用法。`awk`是`shell`编程的一个重要工具。在`shell`命令或编程中，虽然可以使用`awk`强大的文本处理能力，但是并不要求我们成为这方面的专家。

