

awk 手册

简体中文版由 bones7456 (bones7456@gmail.com) 整理.

原文:应该是 <http://phi.sinica.edu.tw/aspac/reports/94/94011/> 但是原文很乱.

说明:之前也是对 awk 几乎一无所知,无意中看到这篇文章,网上一搜,居然没有像样的简体中文版.有的也是不怎么完整,或者错误一大堆的.于是就顺手整理了下这篇文章.通过整理这篇文章,自己也渐渐掌握了 awk 的种种用法.

原文可能比较老,有些目前已经不适用的命令有所改动,文中所有命令均在 ubuntu7.04 下调试通过,用的 awk 是 mawk.

由于本人能力有限,错误和不妥之处在所难免,欢迎多多指正.

1. 前言

有关本手册 :

这是一本 awk 学习指引,其重点着重于 :

- awk 适于解决哪些问题 ?
- awk 常见的解题模式为何 ?

为使读者快速掌握 awk 解题的模式及特性,本手册系由一些较具代表性的范例及其题解所构成;各范例由浅入深,彼此间相互连贯,范例中并对所使用的 awk 语法及指令辅以必要的说明.有关 awk 的指令,函数,...等条列式的说明则收录于附录中,以利读者往后撰写程序时查阅.如此编排,可让读者在短时间内顺畅地学会使用 awk 来解决问题.建议读者循着范例上机实习,以加深学习效果.

读者宜先 具备下列背景 :

[a.] UNIX 环境下的简单操作及基本概念.

例如 : 文件编辑, 文件复制 及 管道, 输入/输出重定向 等概念

[b.] C 语言的基本语法及流程控制指令.

(awk 指令并不多,且其中之大部分与 C 语言中之用法一致,本手册中对该类指令之语法及特性不再加以繁冗的说明,读者若欲深究,可自行翻阅相关的 C 语言书籍)

2. awk 概述

为什么使用 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. 这种解题方式,使得这些小工具可因不同需求而被重复组合及重用(**reuse**); 也可藉此方式来先行测试大程序原型的可行性与正确性,将来若需要较高的执行速度时再用 **C** 语言来改写.这是 **awk** 最常被应用之处.若能常常如此处理问题,读者可以以更高的角度来思考抽象的问题,而不会被拘泥于细节的部份.

本手册为 **awk** 入门的学习指引,其内容将先强调如何撰写 **awk** 程序,未列入进一步解题方式的应用实例,这部分将留待 **UNIX** 进阶手册中再行讨论.

如何取得 **awk**

一般的 **UNIX** 操作系统,本身即附有 **awk**. 不同的 **UNIX** 操作系统所附的 **awk** 其版本亦不尽相同. 若读者所使用的系统上未附有 **awk**,可透过 **anonymous ftp** 到下列地方取得 :

[phi.sinica.edu.tw:/pub/gnu](ftp://phi.sinica.edu.tw/pub/gnu)
[ftp.edu.tw:/UNIX/gnu](ftp://ftp.edu.tw/UNIX/gnu)
[prep.ai.mit.edu:/pub/gnu](ftp://prep.ai.mit.edu/pub/gnu)

awk 如何工作

为便于解释 **awk** 程序架构,及有关术语(**terminology**),先以一个员工薪资档(**emp.dat**),来加以介绍.

```
A125 Jenny 100 210
A341 Dan 110 215
P158 Max 130 209
P148 John 125 220
A123 Linda 95 210
```

文件中各字段依次为 员工 ID, 姓名, 薪资率,及 实际工时. ID 中的第一码为部门识别码. "A","P"分别表示"组装"及"包装"部门.

本小节着重于说明 **awk** 程序的主要架构及工作原理,并对一些重要的名词辅以必要的解释. 由这部分内容,读者可体会出 **awk** 语言的主要精神及 **awk** 与其它语程序言的差异处. 为便于说明,以条列方式说明于后.

名词定义

- 数据行: **awk** 从数据文件上读取数据的基本单位.以上列文件 **emp.dat** 为例, **awk** 读入的第一笔数据行是 "A125 Jenny 100 210"
第二笔数据行是 "A341 Dan 110 215"
一般而言,一个 数据行 就相当于数据文件上的一行资料. (参考 :附录 B 内建变量"RS")
- 字段(Field) : 为数据行上被分隔开的子字符串.
以数据行"A125 Jenny 100 210"为例,
第一栏 第二栏 第三栏 第四栏 "A125" "Jenny" 100 210
一般是以空格符来分隔相邻的字段. (参考 :附录 D 内建变量"FS")

3.如何执行 awk

于 UNIX 的命令行上键入诸如下列格式的指令: ("\$"表 Shell 命令行上的提示符号)

```
$awk 'awk 程序' 数据文件文件名
```

则 awk 会先编译该程序, 然后执行该程序来处理所指定的数据文件.

(上列方式系直接把程序写在 UNIX 的命令行上)

awk 程序的主要结构:

awk 程序中主要语法是 Pattern { Actions}, 故常见之 awk 程序其型态如下:

```
Pattern1 { Actions1 }
```

```
Pattern2 { Actions2 }
```

```
.....
```

```
Pattern3 { Actions3 }
```

Pattern 是什么 ?

awk 可接受许多不同型态的 Pattern. 一般常使用 "关系表达式"(Relational expression) 来当成 Pattern.

例如:

$x > 34$ 是一个 Pattern, 判断变量 x 与 34 是否存在大于的关系.

$x == y$ 是一个 Pattern, 判断变量 x 与变量 y 是否存在等于的关系.

上式中 $x > 34$, $x == y$ 便是典型的 Pattern.

awk 提供 C 语言中常见的关系运算符(Relational Operators) 如

$>$, $<$, $>=$, $<=$, $==$, $!=$

此外, awk 还提供 \sim (match) 及 $!\sim$ (not match) 二个关系运算符(注一).

其用法与涵义如下:

若 A 为一字符串, B 为一正则表达式(Regular Expression)

$A \sim B$ 判断 字符串 A 中是否 包含 能匹配(match)B 表达式的子字符串.

$A !\sim B$ 判断 字符串 A 中是否 不包含 能匹配(match)B 表达式的子字符串.

例如:

"banana" \sim /an/ 整个是一个 Pattern.

因为"banana"中含有可以匹配 /an/ 的子字符串, 故此关系式成立(true), 整个 Pattern 的值也是 true.

相关细节请参考 附录 A Patterns, 附录 E Regular Expression

(注一:) 有少数 awk 论著, 把 \sim , $!\sim$ 当成另一类的 Operator, 并不视为一种 Relational Operator.

本手册中将这两个运算符当成一种 Relational Operator

Actions 是什么 ?

Actions 是由许多 awk 指令构成. 而 awk 的指令与 C 语言中的指令十分类似.

例如:

awk 的 I/O 指令: print, printf(), getline...

awk 的流程控制指令: if(...){...} else {...}, while(...){...}...

(请参考 附录 B --- "Actions")

awk 如何处理 Pattern { Actions } ?

awk 会先判断(Evaluate) 该 Pattern 的值, 若 Pattern 判断后的值为 true (或不为 0 的数字,或不是空的字符串), 则 awk 将执行该 Pattern 所对应的 Actions.反之, 若 Pattern 之值不为 true, 则 awk 将不执行该 Pattern 所对应的 Actions.

例如 : 若 awk 程序中有下列两指令

```
50 > 23 {print "Hello! The word!!" }
```

```
"banana" ~ /123/ { print "Good morning !" }
```

awk 会先判断 50 > 23 是否成立. 因为该式成立, 所以 awk 将印出 "Hello! The word!!". 而另一 Pattern 为 "banana" ~ /123/, 因为 "banana" 内未含有任何子字符串可 match /123/, 该 Pattern 之值为 false, 故 awk 将不会印出 "Good morning !"

awk 如何处理 { Actions } 的语法?(缺少 Pattern 部分)

有时语法 Pattern { Actions } 中, Pattern 部分被省略,只剩 {Actions}.这种情形表示 "无条件执行这个 Actions".

awk 的字段变量

awk 所内建的字段变量及其涵意如下 :

字段变量	含义
\$0	一字符串, 其内容为目前 awk 所读入的数据行.
\$1	\$0 上第一个字段的数据.
\$2	\$0 上第二个字段的数据.
...	其余类推

读入数据 行时, awk 如何更新 (update)这些内建的字段 变量?

当 awk 从数据文件中读取一个数据行时, awk 会使用内建变量\$0 予以记录.每当 \$0 被改动时 (例如 : 读入新的数据行 或 自行变更 \$0,...) awk 会立刻重新分析 \$0 的字段情况, 并将 \$0 上各字段的数据用 \$1, \$2, ..予以记录.

awk 的内建变 量(Built-in Variables)

awk 提供了许多内建变量, 使用者于程序中可使用这些变量来取得相关信息.常见的内建变量有 :

内建变量	含义
NF (Number of Fields)	为一整数, 其值表\$0 上所存在的字段数目.
NR (Number of Records)	为一整数, 其值表 awk 已读入的数据行数目.
FILENAMEawk	正在处理的数据文件文件名.

例如 : awk 从资料文件 emp.dat 中读入第一笔数据行

"A125 Jenny 100 210" 之后, 程序中:

\$0 之值将是 "A125 Jenny 100 210"

\$1 之值为 "A125"

\$2 之值为 "Jenny"

\$3 之值为 100

\$4 之值为 210

\$NF 之值为 4

\$NR 之值为 1

\$FILENAME 之值为 "emp.dat"

awk 的工作流程 :

执行 awk 时, 它会反复进行下列四步骤.

1. 自动从指定的数据文件中读取一个数据行.
2. 自动更新(Update)相关的内建变量之值. 如 : NF, NR, \$0...
3. 依次执行程序中 所有 的 Pattern { Actions } 指令.
4. 当执行完程序中所有 Pattern { Actions } 时, 若数据文件中还有未读取的数据, 则反复执行步骤 1 到步骤 4.

awk 会自动重复进行上述 4 个步骤, 使用者不须于程序中编写这个循环 (Loop).

打印文件中指定的字段 数据并加以计 算

awk 处理数据时, 它会自动从数据文件中一次读取一笔记录, 并会将该数据切分成一个个的字段; 程序中可使用 \$1, \$2,... 直接取得各个字段的内容. 这个特色让使用者易于用 awk 编写 reformatter 来改变数据格式.

[范例 :] 以文件 emp.dat 为例, 计算每人应发工资并打印报表.

[分析 :] awk 会自行一次读入一行数据, 故程序中仅需告诉

awk 如何处理所读入的数据行.

执行如下命令 : (\$ 表 UNIX 命令行上的提示符)

```
$ awk '{ print $2, $3 * $4 }' emp.dat
```

执行结果如下 :

屏幕出现 :

```
Jenny 21000
Dan 23650
Max 27170
John 27500
Linda 19950
```

[说明 :]

UNIX 命令行上, 执行 awk 的语法为:

```
$awk 'awk 程序' 欲处理的资料文件文件名
```

本范例中的 程序部分 为 {print \$2, \$3 * \$4}.

把程序置于命令行时, 程序之前后须以 ' 括住.
emp.dat 为指定给该程序处理的数据文件文件名.

本程序中使用 : Pattern { Actions } 语法.

Pattern 部分被省略, 表无任何限制条件. 故 awk 读入每笔数据行后都将无条件执行这个 Actions.

print 为 awk 所提供的输出指令, 会将数据输出到 stdout(屏幕).

print 的参数间彼此以 "," (逗号) 隔开, 印出数据时彼此间会以空白隔开. (参考 附录 D 内建变量 OFS)

将上述的 程序部分 储存于文件 pay1.awk 中. 执行命令时再指定 awk 程序文件 之文件名. 这是执行 awk 的另一种方式, 特别适用于程序较大的情况, 其语法如下:

```
$ awk -f awk 程序文件名 数据文件文件名
```

故执行下列两命令, 将产生同样的结果.

```
$ awk -f pay1.awk emp.dat  
$ awk '{ print $2, $3 * $4 }' emp.dat
```

读者可使用 "-f" 参数, 让 awk 主程序使用 “其它仅含 awk 函数 的文件中的函数 ” 其语法如下:

```
$ awk -f awk 主程序文件名 -f awk 函数文件名 数据文件文件名
```

(有关 awk 中函数的声明与使用于 7.4 中说明)

awk 中也提供与 C 语言中类似用法的 printf() 函数. 使用该函数可进一步控制数据的输出格式.

编辑另一个 awk 程序如下, 并取名为 pay2.awk

```
{ printf("%6s Work hours: %3d Pay: %5d\n", $2, $3, $3 * $4) }
```

执行下列命令

```
$awk -f pay2.awk emp.dat
```

执行结果屏幕出现:

```
Jenny Work hours: 100 Pay: 21000  
Dan Work hours: 110 Pay: 23650  
Max Work hours: 130 Pay: 27170  
John Work hours: 125 Pay: 27500  
Linda Work hours: 95 Pay: 19950
```

4. 选择符合指定 条件的记 录

Pattern { Action } 为 awk 中最主要的语法. 若某 Pattern 之值为真则执行它后方的 Action. awk 中常使用 "关系表达式" (Relational Expression) 来当成 Pattern.

awk 中除了 >, <, ==, !=, ... 等关系运算符 (Relational Operators) 外, 另外提供 ~(match), !~(Not Match) 二个关系运算符. 利用这两个运算符, 可判断某字符串是否包含能匹配所指定正则表达式的子字符串. 由于这些特性, 很容易使用 awk 来编写需要字符串比对, 判断的程序.

[范例 :] 承上例,
组装部门员工调薪 5%,(组装部门员工之 ID 以"A"开头)
所有员工最后之薪资率若仍低于 100,则以 100 计.
编写 awk 程序打印新的员工薪资率报表.

[分析]: 这个程序须先判断所读入的数据行是否合于指定条件, 再进行某些动作.awk 中 Pattern { Actions } 的语法已涵盖这种 " if (条件) { 动作} "的架构. 编写如下之程序, 并取名 adjust1.awk

```
$1 ~ /^A.*/ { $3 *= 1.05 } $3<100 { $3 = 100 }  
{ printf("%s %8s %d\n", $1, $2, $3)}
```

执行下列命令 :

```
$awk -f adjust1.awk emp.dat
```

结果如下 : 屏幕出现 :

```
A125    Jenny 105  
A341     Dan 115  
P158     Max 130  
P148     John 125  
A123     Linda 100
```

说 明 :

awk 的工作程序是: 从数据文件中每次读入一个数据行, 依序执行完程序中所有的 Pattern{ Action }指令:

```
$1~/^A.*/ { $3 *= 1.05 }  
$3 < 100 { $3 = 100 }  
{printf("%s %8s %d\n",$1,$2,$3)}
```

再从数据文件中读进下一笔记录继续进行处理.

第一个 Pattern { Action }是: \$1 ~ /^A.*/ { \$3 *= 1.05 }

\$1 ~ /^A.*/ 是一个 Pattern, 用来判断该笔数据行的第一栏是否包含以"A"开头的子字符串. 其中 /^A.*/ 是一个 Regular Expression, 用以表示任何以"A"开头的字符串. (有关 Regular Expression 之用法 参考 附录 E).

Actions 部分为 \$3 *= 1.05

\$3 *= 1.05 与 \$3 = \$3 * 1.05 意义相同. 运算符"*"之用法则与 C 语言中一样. 此后与 C 语言中用法相同的运算符或语法将不予赘述.

第二个 Pattern { Actions } 是: \$3 <100 { \$3 = 100 } 若第三栏的数据内容(表薪资率)小于 100, 则调整为 100.

第三个 Pattern { Actions } 是: {printf("%s %8s %d\n",\$1,\$2, \$3)} 省略了 Pattern(无条件执行 Actions), 故所有数据行调整后的数据都将被印出.

5.awk 中数组

awk 程序中允许使用字符串当做数组的下标(index). 利用这个特色十分有助于资料统计工作. (使用字符串当下标的数组称为 Associative Array)

首先建立一个数据文件, 并取名为 reg.dat. 此为一学生注册的资料文件; 第一栏为学生姓名,

其后为该生所修课程.

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

awk 中数组的特性

使用字符串当数组的下标(index).

使用数组前不须宣告数组名及其大小.

例如: 希望用数组来记录 reg.dat 中各门课程的修课人数.

这情况,有二项信息必须储存:

(a) 课程名称, 如: "O.S.", "Arch.".. ,共有哪些课程事先并不明确.

(b) 各课程的修课人数. 如: 有几个人修 "O.S."

在 awk 中只要用一个数组就可同时记录上列信息. 其方法如下:

使用一个数组 Number[] :

以课程名称当 Number[] 的下标.

以 Number[] 中不同下标所对映的元素代表修课人数.

例如:

有 2 个学生修 "O.S.", 则以 Number["O.S."] = 2 表之.

若修 "O.S." 的人数增加一人, 则 Number["O.S."] = Number["O.S."] + 1 或 Number["O.S."]++ .

如何取出 数组中储存的信息

以 C 语言为例, 声明 int Arr[100]; 之后, 若想得知 Arr[] 中所储存的数据, 只须用一个循环, 如 :

```
for(i=0; i<100; i++) printf("%d\n", Arr[i]);
```

即可. 上式中:

数组 Arr[] 的下标 : 0, 1, 2,..., 99

数组 Arr[] 中各下标所对应的值 : Arr[0], Arr[1],...Arr[99]

但 awk 中使用数组并不须事先宣告. 以刚才使用的 Number[] 而言, 程序执行前, 并不知将来有哪些课程名称可能被当成 Number[] 的下标.

awk 提供了一个指令, 藉由该指令 awk 会自动找寻数组中使用过的所有下标. 以 Number[] 为例, awk 将会找到 "O.S.", "Arch.",...

使用该指令时, 须指定所要找寻的数组, 及一个变量. awk 会使用该的变量来记录从数组中找到的每一个下标. 例如

```
for(course in Number){....}
```

指定用 course 来记录 awk 从 Number[] 中所找到的下标. awk 每找到一个下标时, 就用 course 记录该下标之值且执行 {...} 中之指令. 藉由这个方式便可取出数组中储存的信息. (详见下例)

[范例 :] 统计各科修课人数, 并印出结果.

建立如下程序, 并取名为 course.awk:

```
{ for( i=2; i <= NF; i++) Number[$i]++ }
END{for(course in Number) printf("%10s %d\n", course, Number[course] )}
```


执行下列命令：

```
$awk -f course.awk reg.dat
```

执行结果如下：

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

[说明:]

这程序包含二个 Pattern { Actions } 指令.

```
{ for( i=2; i <= NF; i++) Number[$i]++ }
END{for(course in Number) printf("%10s %d\n", course, Number[course] )}
```

第一个 Pattern { Actions } 指令中省略了 Pattern 部分. 故随着

每笔数据行的读入其 Actions 部分将逐次无条件被执行.

以 awk 读入第一笔资料 " Mary O.S. Arch. Discrete" 为例, 因为该笔数据 NF = 4(有 4 个字段), 故该 Action 的 for Loop 中 i = 2,3,4.

i \$i 最初 Number[\$i] Number[\$i]++ 之后

i=2 时 \$i="O.S." Number["O.S."]的值从默认的 0,变成了 1;

i=3 时 \$i="Arch." Number["Arch."]的值从默认的 0,变成了 1;

同理,i=4 时 \$i="Discrete" Number["Discrete"]的值从默认的 0,变成了 1;

第二个 Pattern { Actions } 指令中 END 为 awk 之保留字, 为 Pattern 的一种.

END 成立(其值为 true)的条件是: "awk 处理完所有数据, 即将离开程序时."

平常读入数据行时, END 并不成立, 故其后的 Actions 并不被执行;

唯有当 awk 读完所有数据时, 该 Actions 才会被执行 (注意, 不管数据行有多少笔, END 仅在最后才成立, 故该 Actions 仅被执行一次.)

BEGIN 与 END 有点类似, 是 awk 中另一个保留的 Pattern.

唯一不同的是: "以 BEGIN 为 Pattern 的 Actions 于程序一开始执行时, 被执行一次."

NF 为 awk 的内建变量, 用以表示 awk 正处理的数据行中, 所包含的字段个数.

awk 程序中若含有以 \$ 开头的自定变量, 都将以如下方式解释:

以 i=2 为例, \$i = \$2 表第二个字段数据. (实际上, \$ 在 awk 中为一运算符(Operator), 用以取得字段数据.)

6.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. 读者可利用这个特点来编写某些适用的系统工具.

[范例:] 写一个 awk 程序来打印出线上人数.

将下列程序建文件, 命名为 `count.awk`

```
BEGIN {  
while ( "who" | getline ) n++  
print n  
}
```

并执行下列命令：

```
awk -f count.awk
```

执行结果将会印出目前在线人数

[说明:]

`awk` 程序并不一定要处理数据文件. 以本例而言, 仅输入程序文件 `count.awk`, 未输入任何数据文件.

`BEGIN` 和 `END` 同为 `awk` 中的一种 `Pattern`. 以 `BEGIN` 为 `Pattern` 的 `Actions`, 只有在 `awk` 开始执行程序, 尚未开启任何输入文件前, 被执行一次.(注意: 只被执行一次)

"|" 为 `awk` 中表示管道的符号. `awk` 把 | 之前的字符串"who"当成 `Shell` 上的命令, 并将该命令送往 `Shell` 执行, 执行的结果(原先应于屏幕印出者)则藉由 `pipe` 送进 `awk` 程序中.

`getline` 为 `awk` 所提供的输入指令.

其语法如下：

语法	由何处读取数据	数据读入后置于
<code>getline var < file</code>	所指定的 <code>file</code>	变量 <code>var</code> (<code>var</code> 省略时, 表示置于 <code>\$0</code>)
<code>getline var</code>	<code>pipe</code> 变量	变量 <code>var</code> (<code>var</code> 省略时, 表示置于 <code>\$0</code>)
<code>getline var</code>	见 注一	变量 <code>var</code> (<code>var</code> 省略时, 表示置于 <code>\$0</code>)

注一：当 `Pattern` 为 `BEGIN` 或 `END` 时, `getline` 将由 `stdin` 读取数据, 否则由 `awk` 正处理的数据文件上读取数据.

`getline` 一次读取一行数据, 若读取成功则 `return 1`, 若读取失败则 `return -1`, 若遇到文件结束 (EOF), 则 `return 0`;

本程序使用 `getline` 所 `return` 的数据来做为 `while` 判断循环停止的条件, 某些 `awk` 版本较旧, 并不容许使用者改变 `$0` 之值. 这种版的 `awk` 执行本程序时会产生 `Error`, 读者可于 `getline` 之后置上一个变量 (如此, `getline` 读进来的数据便不会被置于 `$0`), 或直接改用 `gawk` 便可解决.

7.awk 程序的应 用实例

本节将示范一个统计上班到达时间及迟到次数的程序.

这程序每日被执行时将读入二个文件:

员工当日到班时间的数据文件 (如下列之 `arr.dat`)

存放员工当月迟到累计次数的文件.

当程序执行执完后将更新第二个文件的数据(迟到次数), 并打印当日的报表. 这程序将分成下列数小节逐步完成, 其大纲如下:

[7.1] 在到班资料文件 **arr.dat** 之前增加一行抬头

"ID Number Arrival Time", 并产生报表输出到文件 **today_rpt1** 中.

< 思考: 在 **awk** 中如何将数据输出到文件 >

[7.2] 将 **today_rpt1** 上的数据 按员工代号排序, 并加注执行当日日期 ; 产生文件 **today_rpt2**

< 思考 **awk** 中如何运用系统资源及 **awk** 中 Pipe 之特性 >

[7.3] 将 **awk** 程序包含 在一个 **shell script** 文件中

[7.4] 于 **today_rpt2** 每日报表上 , 迟到者之前加上 "*", 并加注当日平均 到班时间 ; 产生文件 **today_rpt3**

[7.5] 从文件中 读取当月迟到 次数, 并根据当 日出勤状况更新迟到累计数 .

< 思考 使用者在 **awk** 中如何读取文件数据 >

某公司其员工到勤时间档如下, 取名为 **arr.dat**. 文件中第一栏为员工代号, 第二栏为到达时间. 本范例中, 将使用该文件为数据文件.

```
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
```

➤ 重定向输出到文件

awk 中并未提供如 C 语言中之 **fopen()** 指令, 也未有 **fprintf()** 文件输出这样的指令. 但 **awk** 中任何输出函数之后皆可借助使用与 UNIX 中类似的 I/O 重定向符, 将输出的数据重定向到指定的文件; 其符号仍为 **>** (输出到一个新产生的文件) 或 **>>** (添加输出的数据到文件末尾).

[例 :] 在到班数据文件 **arr.dat** 之前增加一行抬头如下:

"ID Number Arrival Time", 并产生报表输出到文件 **today_rpt1** 中

建立如下文件并取名为 **reformat1.awk**

```
BEGIN { print " ID Number Arrival Time" > "today_rpt1"
print "======" > "today_rpt1"
}
{ printf(" %s %s\n", $1,$2 ) > "today_rpt1" }
```

执行:

```
$awk -f reformat1.awk arr.dat
```

执行后将产生文件 **today_rpt1**, 其内容如下 :

```
ID Number Arrival Time
```

```
=====
```

```
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
```

[说明:]

awk 程序中, 文件名称 `today_rpt1` 的前后须以" (双引号)括住, 表示 `today_rpt1` 为一字符串常量. 若未以"括住, 则 `today_rpt1` 将被 awk 解释为一个变量名称.

在 awk 中任何变量使用之前, 并不须事先声明. 其初始值为空字符串(Null string) 或 0. 因此程序中若未以 " 将 `today_rpt1` 括住, 则 `today_rpt1` 将是一变量, 其值将是空字符串, 这会在执行时造成错误(Unix 无法帮您开启一个以空字符串为文件名的文件).

因此在编辑 awk 程序时, 须格外留心. 因为若敲错变量名称, awk 在编译程序时会认为是一新的变量, 并不会察觉. 因此往往会造成运行时错误.

BEGIN 为 awk 的保留字, 是 Pattern 的一种.

以 BEGIN 为 Pattern 的 Actions 于 awk 程序刚被执行尚未读取数据文件时被执行一次, 此后便不再被执行.

读者或许觉得本程序中的 I/O 重定向符号应使用 "`>>`" (append)而非 "`>`".

本程序中若使用 "`>`" 将数据重导到 `today_rpt1`, awk 第一次执行该指令时会产生一个新档 `today_rpt1`, 其后再执行该指令时则把数据追加到 `today_rpt1` 文件末, 并非每执行一次就重开一个新文件.

若采用"`>>`"其差异仅在第一次执行该指令时, 若已存在 `today_rpt1` 则 awk 将直接把数据 append 在原文件之末尾. 这一点, 与 UNIX 中的用法不同.

➤awk 中如何利用系统资源

awk 程序中很容易使用系统资源. 这包括在程序中途调用 Shell 命令来处理程序中的部分数据; 或在调用 Shell 命令后将其产生的结果交回 awk 程序(不需将结果暂存于某个文件). 这一过程是借助 awk 所提供的管道 (虽然有些类似 Unix 中的管道, 但特性有些不同), 及一个从 awk 中呼叫 Unix 的 Shell 命令的语法来达成的.

[例:] 承上题, 将数据按员工 ID 排序后再输出到文件 `today_rpt2`, 并于表头附加执行时的日期.

[分析:]

awk 提供与 UNIX 用法近似的 pipe, 其记号亦为 "|". 其用法及含意如下:

awk 程序中可接受下列两种语法:

[a. 语法] awk output 指令 | "Shell 接受的命令"
(如 : print \$1,\$2 | "sort -k 1")

[b. 语法] "Shell 接受的命令" | awk input 指令
(如 : "ls " | getline)

注 : awk input 指令只有 getline 一个.

awk output 指令有 print, printf() 二个.

在 a 语法中, awk 所输出的数据将转送往 Shell, 由 Shell 的命令进行处理. 以上例而言, print 所输出的数据将经由 Shell 命令 "sort -k 1" 排序后再送往屏幕(stdout).

上列 awk 程序中, "print\$1, \$2" 可能反复执行很多次, 其输出的结果将先暂存于 pipe 中, 等到该程序结束时, 才会一并进行 "sort -k 1".

须注意二点 : 不论 print \$1, \$2 被执行几次, "sort -k 1" 的执行时间是 "awk 程序结束时", "sort -k 1" 的执行次数是 "一次".

在 b 语法中, awk 将先调用 Shell 命令. 其执行结果将通过 pipe 送入 awk 程序, 以上例而言, awk 先让 Shell 执行 "ls", Shell 执行后将结果存于 pipe, awk 指令 getline 再从 pipe 中读取数据.

使用本语法时应留心: 以上例而言, awk "立刻"调用 Shell 来执行 "ls", 执行次数是一次.

getline 则可能执行多次(若 pipe 中存在多行数据).

除上列 a, b 二中语法外, awk 程序中其它地方如出现像 "date", "cls", "ls"... 这样的字符串, awk 只把它当成一般字符串处理.

建立如下文件并取名为 reformat2.awk

```
# 程序 reformat2.awk
# 这程序用以练习 awk 中的 pipe
BEGIN {
    "date" | getline # Shell 执行 "date". getline 取得结果并以$0 记录
    print " Today is " , $2, $3 >"today_rpt2"
    print "===== " > "today_rpt2"
    print " ID Number Arrival Time" >"today_rpt2"
    close( "today_rpt2" )
}
{printf( "%s %s\n", $1 , $2 ) | "sort -k 1 >>today_rpt2"}
```

执行如下命令:

awk -f reformat2.awk arr.dat

执行后, 系统会自动将 sort 后的数据追加(Append; 因为使用 " >>") 到文件 today_rpt2 末端.
today_rpt2 内容如下 :

```
Today is 09月 21日
=====
ID Number Arrival Time
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
```

[说明:]

awk 程序由三个主要部分构成：

[i.] Pattern { Action} 指令

[ii.] 函数主体. 例如：`function double(x){ return 2*x }` (参考第 11 节 Recursive Program)

[iii.] Comment (以 # 开头识别之)

awk 的输入指令 `getline`, 每次读取一行数据. 若 `getline` 之后

未接任何变量, 则所读入之资料将以 `$0` 记录, 否则以所指定的变量储存之.

[以本例而言]:

执行 `"date"|getline` 后, `$0` 之值为 "2007 年 09 月 21 日 星期五 14:28:02 CST", 当 `$0` 之值被更新时, awk 将自动更新相关的内建变量, 如: `$1`, `$2`, ..., `NF`. 故 `$2` 之值将为 "09 月", `$3` 之值将为 "21 日".

(有少数旧版的 awk 不允许即使用者自行更新(update)`$0` 的值, 或者更新 `$0` 时, 它不会自动更新 `$1`, `$2`, ..., `NF`. 这情况下, 可改用 gawk 或 nawk. 否则使用者也可自行以 awk 字符串函数 `split()` 来分隔 `$0` 上的数据)

本程序中 `printf()` 指令会被执行 12 次(因为有 `arr.dat` 中有 12 行数据), 但读者不用担心数据被重复 sort 了 12 次. 当 awk 结束该程序时才会 `close` 这个 `pipe`, 此时才将这 12 行数据一次送往系统, 并呼叫 `"sort -k 1 >> today_rpt2"` 处理之.

awk 提供另一个调用 Shell 命令的方法, 即使用 awk 函数 `system("shell 命令")`

例如:

```
$ awk '
BEGIN{
system("date > date.dat")
getline < "date.dat"
print "Today is ", $2, $3
}
'
```

但使用 `system("shell 命令")` 时, awk 无法直接将执行中的部分数据输出给 Shell 命令. 且 Shell 命令执行的结果也无法直接输入到 awk 中.

➤ 执行 awk 程序的几种方式

本小节中描述如何将 awk 程序直接写在 shell script 之中. 此后使用者执行 awk 程序时, 就不需要每次都键入 `"awk -f program datafile"`.

script 中还可包含其它 Shell 命令, 如此更可增加执行过程的自动化.

建立一个简单的 awk 程序 mydump.awk, 如下:

```
{print}
```

这个程序执行时会把数据文件的内容 print 到屏幕上(与 cat 功用类似).

print 之后未接任何参数时, 表示 "print \$0".

若欲执行该 awk 程序, 来印出文件 today_rpt1 及 today_rpt2 的内容时, 必须于 UNIX 的命令行上执行下列命令:

方式一 `awk -f mydump.awk today_rpt1 today_rpt2`

方式二 `awk '{print}' today_rpt1 today_rpt2` 第二种方式系将 awk 程序直接写在 Shell 的命令行上, 这种方式仅适合较短的 awk 程序.

方式三 建立如下之 shell script, 并取名为 mydisplay,

```
#!/bin/sh

# 注意以下的 awk 与 ' 之间须有空白隔开
awk '
{print}
' $*
# 注意以上的 ' 与 $* 之间须有空白隔开
```

执行 mydisplay 之前, 须先将它改成可执行的文件(此步骤往后不再赘述). 请执行如下命令:

```
$ chmod +x mydisplay
```

往后使用者就可直接把 mydisplay 当成指令, 来 display 任何文件.

例如:

```
$ ./mydisplay today_rpt1 today_rpt2
```

[说明:]

在 script 文件 mydisplay 中, 指令"awk"与第一个 ' 之间须有空格(Shell 中并无 "awk'" 指令).

第一个 ' 用以通知 Shell 其后为 awk 程序.

第二个 ' 则表示 awk 程序结束.

故 awk 程序中一律以"括住字符串或字符, 而不使用 ', 以免 Shell 混淆.

\$* 为 shell script 中的用法, 它可用来代表命令行上 "mydisplay 之后的所有参数".

例如执行:

```
$ mydisplay today_rpt1 today_rpt2
```

事实上 Shell 已先把该指令转换成:

```
awk '
{ print}
' today_rpt1 today_rpt2
```

本例中, \$* 用以代表 "today_rpt1 today_rpt2". 在 Shell 的语法中, 可用 \$1 代表第一个参数, \$2 代表第二个参数. 当不确定命令行上的参数个数时, 可使用 \$* 表之.

awk 命令行上可同时指定多个数据文件.

以 `awk -f dump.awk today_rpt1 today_rpt2hf` 为例, awk 会先处理 today_rpt1, 再处理 today_rpt2.

此时若文件无法打开, 将造成错误.

例如: 不存在文件 "file_no_exist", 则执行:

```
$ awk -f dump.awk file_no_exit
```

将产生运行时错误(无法打开文件).

但某些 `awk` 程序 "仅" 包含以 `BEGIN` 为 `Pattern` 的指令. 执行这种 `awk` 程序时, `awk` 并不须开启任何数据文件. 此时命令行上若指定一个不存在的数据文件, 并不会产生 "无法打开文件" 的错误. (事实上 `awk` 并未打开该文件)

例如执行:

```
$ awk 'BEGIN {print "Hello,World!!"} ' file_no_exist
```

该程序中仅包含以 `BEGIN` 为 `Pattern` 的 `Pattern {actions}`, `awk` 执行时并不会开启任何数据文件; 所以不会因不存在文件 `file_no_exit` 而产生 "无法打开文件" 的错误.

`awk` 会将 `Shell` 命令行上 `awk` 程序(或 `-f` 程序文件名)之后的所有字符串, 视为将输入 `awk` 进行处理的数据文件文件名.

若执行 `awk` 的命令行上 "未指定任何数据文件文件名", 则将 `stdin` 视为输入之数据来源, 直到输入 `end of file` (`Ctrl-D`) 为止.

读者可以用下列程序自行测试, 执行如下命令:

```
$ awk -f mydump.awk #(未接任何数据文件文件名)
```

或

```
$ ./mydisplay #(未接任何数据文件文件名)
```

将会发现: 此后键入的任何数据将逐行复印一份于屏幕上. 这情况不是机器当机! 是因为 `awk` 程序正处于执行中. 它正按程序指示, 将读取数据并重新 `dump` 一次; 只因执行时未指定数据文件文件名, 故 `awk` 便以 `stdin` (键盘上的输入) 为数据来源. 读者可利用这个特点, 设计可与 `awk` 即时聊天的程序.

➤ 改变 `awk` 切割字段的方式 & 自定义函数

`awk` 不仅能自动分割字段, 也允许使用者改变其字段切割方式以适应各种格式之需要. 使用者也可自定义函数, 若有需要可将该函数单独写成一个文件, 以供其它 `awk` 程序调用.

[范例:] 承接 6.2 的例子, 若八点为上班时间, 请加注 "*" 于迟到记录之前, 并计算平均上班时间.

[分析:]

因八点整到达者, 不为迟到, 故仅以到达的小时数做判断是不够的; 仍应参考到达时的分钟数.

若 "将到达时间转换成以分钟为单位", 不仅易于判断是否迟到, 同时也易于计算到达平均时间.

到达时间(\$2)的格式为 `dd:dd` 或 `d:dd`; 数字当中含有一个 ":". 但文本数字交杂的数据 `awk` 无法直接做数学运算. (注: `awk` 中字符串 "26" 与数字 26, 并无差异, 可直接做字符串或数学运算, 这是 `awk` 重要特色之一. 但 `awk` 对文本数字交杂的字符串无法正确进行数学运算).

解决之方法:

[方法一]

对到达时间(\$2) `d:dd` 或 `dd:dd` 进行字符串运算, 分别取出到达的小时数及分钟数.

首先判断到达小时数为一位或两位字符, 再调用函数分别截取分钟数及小时数.

此解法需使用下列 `awk` 字符串函数:

length(字符串): 返回该字符串的长度.

substr(字符串,起始位置,长度): 返回从起始位置起, 指定长度之子字符串. 若未指定长度, 则返回从起始位置到字符串末尾的子字符串.

所以:

小时数 = substr(\$2, 1, length(\$2) - 3)

分钟数 = substr(\$2, length(\$2) - 2)

[方法二]

改变输入列字段的切割方式, 使 **awk** 切割字段后分别将小时数及分钟数隔开于二个不同的字段.

字段分隔字符 **FS (field separator)** 是 **awk** 的内建变量,其默认值是空白及 **tab**. **awk** 每次切割字段时都会先参考 **FS** 的内容. 若把 ":" 也当成分隔字符, 则 **awk** 便能自动把小时数及分钟数分隔成不同的字段.故令 **FS = "[\t:]"** (注: **[\t:]+** 为一 **Regular Expression**)

Regular Expression 中使用中括号 **[...]** 表示一个字符集合,用以表示任意一个位于两中括号间的字符.故可用 **"[\t:]"** 表示 一个 空白 , **tab** 或 ":"

Regular Expression 中使用 **"+"** 形容其前方的字符可出现一次或一次以上.

故 **"[\t:]"** 表示由一个或多个 "空白, **tab** 或 ":" 所组成的字符串.

设定 **FS = "[\t:]"** 后, 数据行如: "1034 7:26" 将被分割成 3 个字段

第一栏 第二栏 第三栏

\$1 \$2 \$3

1034 7 26

明显地, **awk** 程序中使用方法二比方法一更简洁方便. 本例子中采用方法二,也借此示范改变字段切割方式的用途.

编写 **awk** 程序 **reformat3**, 如下 :

```
#!/bin/sh

awk '
BEGIN {
FS= "[\t:]" #改变字段切割的方式
"date" | getline # Shell 执行 "date". getline 取得结果以$0 记录
print " Today is " , $2, $3 > "today_rpt3"
print "===== "> "today_rpt3"
print " ID Number Arrival Time" > "today_rpt3"
close( "today_rpt3" )
}
{
#已更改字段切割方式, $2 表到达小时数, $3 表分钟数
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
}
' $*

```

并执行如下指令：

```
$ ./reformat3 arr.dat
```

执行后,文件 `today_rpt3` 的内容如下:

```

Today is 09月 21日
=====
ID Number Arrival Time
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
Average arrival time : 7:49

```

[说明:]

awk 中亦允许使用者自定函数. 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. 使用者自定函数时, 还有许多细节须留心, 如 `data scope...` (请参考 第十节 Recursive Program)

awk 中亦提供与 C 语言中相同的 Conditional Operator. 上式 `printf()` 中使用 `arrival > 480 ? "*" :`
`" "` 即为一例若 `arrival` 大于 480 则 `return "*" , 否则 return " "`.

`%` 为 awk 的运算符(operator), 其作用与 C 语言中之 `%` 相同(取余数).

`NR`(Number of Record) 为 awk 的内建变量. 表示 awk 执行该程序后所读入的记录笔数.

awk 中提供的 `close()` 指令, 语法如下(有二种):

`close(filename)`

`close(置于 pipe 之前的 command)`

为何本程序使用了两个 `close()` 指令：

指令 `close("sort -k 1 >> today_rpt3")`, 其意思为 `close` 程序中置于 `"sort -k 1 >> today_rpt3 "` 之前的 Pipe, 并立刻调用 Shell 来执行 `"sort -k 1 >> today_rpt3"`. (若未执行这指令, awk 必须于结束该程序时才会进行上述动作;则这 12 笔 sort 后的数据将被 append 到文件 `today_rpt3` 中 "Average arrival time: ..." 的后方)

因为 Shell 排序后的数据也要写到 today_rpt3, 所以 awk 必须先关闭使用中的 today_rpt3 以使 Shell 正确将排序后的数据追加到 today_rpt3 否则 2 个不同的 process 同时打开一个文件进行输出将会产生不可预期的结果.

读者应留心上述两点,才可正确控制数据输出到文件中的顺序.

指令 close("sort -k 1 >> today_rpt3")中字符串 "sort +0n >> today_rpt3" 必须与 pipe | 后方的 Shell Command 名称一字不差, 否则 awk 将视为二个不同的 pipe.

读者可于 BEGIN{} 中先令变量 Sys_call = "sort +0n >> today_rpt3", 程序中再一律以 Sys_call 代替该字符串.

➤使用 getline 来读取数据

[范 例:] 承上题,从文件中读取当月迟到次数, 并根据当日出勤状况更新迟到累计数.(按不同的月份累计于不同的文件)

[分 析:]

程序中自动抓取系统日期的月份名称, 连接上 "late.dat", 形成累计迟到次数的文件名称(如 : 09 月 late.dat,...), 并以变量 late_file 记录该文件名.

累计迟到次数的文件中的数据格式为: 员工代号(ID) 迟到次数

例如, 执行本程序前文件 09 月 late.dat 的内容为 :

```
1012 0
1006 1
1052 2
1034 0
1005 0
1029 2
1042 0
1051 0
1008 0
1101 0
1025 1
1028 0
```

编写程序 reformat4 如下:

```
#!/bin/sh

awk '
BEGIN {
Sys_Sort = "sort -k 1 >> today_rpt4"
Result = "today_rpt4"
# 改变字段切割的方式
FS = "[\t:]+"
# 令 Shell 执行"date"; getline 读取结果,并以$0 记录
"date" | getline
print "Today is " , $2, $3 >Result
print "===== " > Result
print " ID Number Arrival Time" > Result
close( Result )
```

```

# 从文件按中读取迟到数据，并用数组 cnt[ ]记录. 数组 cnt[ ]中以
# 员工代号为下标，所对应的值为该员工之迟到次数.
late_file = $2"late.dat"
while( getline < late_file >0 ) cnt[$1] = $2
close( late_file )
}
{
# 已更改字段切割方式, $2 表小时数,$3 表分钟数
arrival = HM_to_M($2, $3)
if( arrival > 480 ){
mark = "*" # 若当天迟到,应再增加其迟到次数, 且令 mark 为"*.
cnt[$1]++ }
else mark = " "

# message 用以显示该员工的迟到累计数, 若未曾迟到 message 为空字符串
message = cnt[$1] ? cnt[$1] " times" : ""
printf("%s %2d:%2d %5s %s\n", $1, $2, $3, mark, message ) | Sys_Sort
total += arrival
}
END {
close( Result )
close( Sys_Sort )
printf(" Average arrival time : %d:%d\n", total/NR/60, (total/NR)%60 ) >> Result
#将数组 cnt[ ]中新的迟到数据写回文件中
for( any in cnt )
print any, cnt[any] > late_file
}
function HM_to_M( hour, min ){
return hour*60 + min
}
' $*

```

执行后, today_rpt4 之内容如下 :

```

Today is 09月 21日
=====
ID Number Arrival Time
1005 8:12 * 1 times
1006 7:45 1 times
1008 8: 1 * 1 times
1012 7:46
1025 7:27 1 times
1028 7:49
1029 7:57 2 times
1034 7:26
1042 7:59
1051 7:51
1052 8: 5 * 3 times
1101 7:32
Average arrival time : 7:49

```

09月 late.dat 文件被修改为如下:

```

1005 1

```

```
1012 0
1006 1
1008 1
1101 0
1025 1
1034 0
1042 0
1028 0
1029 2
1051 0
1052 3
```

说明：

`late_file` 是一变量, 用以记录迟到次数的文件的文件名.

`late_file` 之值由两部分构成, 前半部是当月月份名称(由调用"`date`"取得)后半部固定为"`late.dat`" 如: 09 月 `late.dat`.

指令 `getline < late_file` 表示从 `late_file` 所代表的文件中读取一笔记录, 并存放于 `$0`.

若使用者可自行把数据放入 `$0`, `awk` 会自动对这新置入 `$0` 的数据进行字段分割. 之后程序中可用 `$1`, `$2`,... 来表示该笔资料的第一栏, 第二栏,...

(注: 有少数 `awk` 版本不容许使用者自行将数据置于 `$0`, 遇此情况可改用 `gawk` 或 `nawk`)

执行 `getline` 指令时, 若成功读取记录, 它会返回 1. 若遇到文件结束, 它返回 0; 无法打开文件则返回 -1.

利用 `while(getline < filename > 0) {....}` 可读入文件中的每一笔数据并予处理. 这是 `awk` 中用户自行读取数据文件的一个重要模式.

数组 `cnt[]` 以员工 ID. 当下标(index), 其对应值表示其迟到的次数.

执行结束后, 利用 `for(Variable in array){...}` 的语法

`for(any in cnt) print any, cnt[any] > late_file`

将更新过的迟到数据重新写回记录迟到次数的文件. 该语法在前面曾有说明.

8. 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` 会把空白行当成来文件中 `Record` 的分隔符. 显然 `awk` 对 `RS = ""` 另有解释方式, 简略描述如下, 当 `RS = ""` 时: 数个并邻的空白行, `awk` 仅视成一个单一的 `Record Separator`. (`awk` 不会于两个紧并的空白行之间读取一笔空的 `Record`)

`awk` 会略过(skip)文件头或文件尾的空白行. 故不会因为这样的空白行, 造成 `awk` 多读入了二笔空的数据.

请观察下例, 首先建立一个数据文件 `week.rpt` 如下:

张长弓
GNUPLOT 入门

吴国强
Latex 简介
VAST-2 使用手册
mathematic 入门

李小华
awk Tutorial Guide
Regular Expression

该文件的开头有数行空白行, 各笔 **Record** 之间使用一个或数个空白行隔开. 读者请细心观察, 当 `RS = ""` 时, `awk` 读取该数据文件之方式.

编辑一个 `awk` 程序文件 `make_report` 如下:

```
#!/bin/sh

awk '
BEGIN {
FS = "\n"
RS = ""
split( "一. 二. 三. 四. 五. 六. 七. 八. 九.", C_Number, " " )
}
{
printf("\n%s 报告人 : %s \n",C_Number[NR],$1)
for( i=2; i <= NF; i++) printf(" %d. %s\n", i-1, $i)
} ' $*
```

执行

```
$ make_report week.rpt
```

屏幕产生结果如下:

```
一. 报告人 : 张长弓
  1. GNUPLOT 入门

二. 报告人 : 吴国强
  1. Latex 简介
  2. VAST-2 使用手册
  3. mathematic 入门

三. 报告人 : 李小华
  1. awk Tutorial Guide
  2. Regular Expression
```

[说 明:]

本程序同时也改变字段分隔字符(`FS= "\n"`), 如此一笔数据中的每一行都是一个 `field`. 例如:

`awk` 读入的第一笔 **Record** 为

```
张长弓
```

GNUPLOT 入门

其中 \$1 指的是"张长弓", \$2 指的是"GNUPLOT 入门"

上式中的 C_Number[] 是一个数组(array), 用以记录中文数字. 例如: C_Number[1] = "一.", C_Number[2] = "二." 这过程使用 awk 字符串函数 split() 来把中文数字放进数组 C_Number[] 中.

函数 split() 用法如下:

split(原字符串, 数组名, 分隔字符(field separator)): awk 将依所指定的分隔字符(field separator) 分隔原字符串成一个个的字段(field), 并以指定的 数组 记录各个被分隔的字段

9. 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. 本小节中将叙述如何在 awk 程序中取用这些参数.

建立文件如下, 命名为 see_arg :

```
#!/bin/sh

awk '
BEGIN {
for( i=0; i<ARGC ; i++)
print ARGV[i] # 依次印出 awk 所记录的参数
}
' $*
```

执行如下命令 :

```
$ ./see_arg first-arg second-arg
```

结果屏幕出现 :

```
awk
first-arg
second-arg
```

[说明 :]

ARGC, ARGV[] 为 awk 所提供的内建变量.

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.

例如, 当命令行为 :

```
$ awk -vx=36 -f program1 data1 data2
```

或

```
$ awk '{ print $1 , $2 }' data1 data2
```

其 ARGC 之值为 3

ARGV[0] 之值为 "awk"

ARGV[1] 之值为 "data1"

ARGV[2] 之值为 "data2"

命令行上的 "-f program1", "-vx=36", 或程序部分 '{ print \$1, \$2}' 都不会列入 ARGV 及 ARGV[] 中.

awk 利用 ARGV 来判断应开启的数据文件个数.

但使用者可强行改变 ARGV; 当 ARGV 之值被使用者设为 1 时;

awk 将被蒙骗, 误以为命令行上并无数据文件文件名, 故不会以 ARGV[1], ARGV[2], .. 为文件名来打开文件读取数据; 但在程序中仍可通过 ARGV[1], ARGV[2], .. 来取得命令行上的数据.

某一程序 test1.awk 如下 :

```
BEGIN{
  number = ARGV #先用 number 记住实际的参数个数.
  ARGV = 2 # 自行更改 ARGV=2, awk 将以为只有一个资料文件
  # 仍可藉由 ARGV[] 取得命令行上的资料.
  for( i=2; i<number; i++) data[i] = ARGV[i]
}
.....
```

于命令行上键入

```
$ awk -f test1.awk data_file apple orange
```

执行时 awk 会打开数据文件 data_file 以进行处理. 但不会打开以 apple, orange 为档名的文件(因为 ARGV 被改成 2). 但仍可通过 ARGV[2], ARGV[3] 取得命令行上的参数 apple, orange

也可以用下列命令来达成上例的效果.

```
$awk -f test2.awk -v data[2]="apple" -v data[3]="orange" data_file
```

10. 编写可与用户交互的 awk 程序

执行 awk 程序时, awk 会自动从文件中读取数据来进行处理, 直到文件结束. 只要将 awk 读取数据的来源改成键盘输入, 便可设计与 awk 交互的程序了.

本节将提供一个该类程序的范例.

[范例 :] 本节将编写一个英语生字测验的程序, 它将印出中文字意, 再由使用者回答其英语生字.

首先编辑一个数据档 test.dat (内容不限, 格式如下)

```
apple 苹果
orange 柳橙
banana 香蕉
pear 梨子
starfruit 杨桃
bellfruit 莲雾
kiwi 奇异果
pineapple 菠萝
watermelon 西瓜
```

编辑 awk 程序 "c2e" 如下:

```
#!/bin/sh
```



```

awk '
BEGIN {
while( getline < ARGV[1] ){ #由指定的文件中读取测验数据
English[++n] = $1 # 最后, n 将表示题目之题数
Chinese[n] = $2
}
ARGV[1] = "-" # "-"表示由 stdin(键盘输入)
srand() # 以系统时间为随机数启始的种子
question() #产生考题
}

{# awk 自动读入由键盘上输入的数据(使用者回答的答案)
if($1 != English[ind] )
print "Try again!"
else{
    print "\nYou are right !! Press Enter to Continue --- "
    getline
    question()#产生考题
}
}
function question(){
ind = int(rand()* n) + 1 #以随机数选取考题
system("clear")
print " Press \"ctrl-d\" to exit"
printf("\n%s ", Chinese[ind] " 的英文生字是: ")
}
' $*

```

执行时键入如下指令：

```
./c2e test.dat
```

屏幕将产生如下的画面:

```
Press "ctrl-d " to exit
```

莲雾 的英文生字是:

若输入 bellfruit

程序将产生

```
You are right !! Press Enter to Continue ---
```

[说明:]

参数 test.dat (ARGV[1])表示储存考题的数据文件文件名. awk 由该文件上取得考题资料后, 将 ARGV[1]改成 "-".

"-" 表示由 stdin(键盘输入)数据. 键盘输入数据的结束符号 (End of file)是 ctrl-d. 当 awk 读到 ctrl-d 时就停止由 stdin 读取数据.

awk 的数学函数中提供两个与随机数有关的函数.

srand(): 以当前的系统时间作为随机数的种子

rand(): 返回介于 0 与 1 之间的(近似)随机数值.

11.使用 awk 编写递归程序

awk 中除了函数的参数列(Argument List)上的参数(Arguments)外,所有变量不管于何处出现,全被视为全局变量.其生命持续至程序结束 --- 该变量不论在 function 外或 function 内皆可使用,只要变量名称相同所使用的就是同一个变量,直到程序结束.

因递归函数内部的变量,会因它调用子函数(本身)而重复使用,故编写该类函数时,应特别留心.

[例如:]执行

```
awk '
BEGIN {
x = 35
y = 45
test_variable( x )
printf("Return to main : arg1= %d, x= %d, y= %d, z= %d\n", arg1, x, y, z)
}
function test_variable( arg1 )
{
arg1++ # arg1 为参数列上的参数, 是 local variable. 离开此函数后将消失.
y ++ # 会改变主式中的变量 y
z = 55 # z 为该函数中新使用的变量, 主程序中变量 z 仍可被使用.
printf("Inside the function: arg1=%d,x=%d, y=%d, z=%d\n", arg1, x, y, z)
} '
```

结果屏幕印出

```
Inside the function: arg1=36,x=35, y=46, z=55
Return to main : arg1= 0, x= 35, y= 46, z= 55
```

由上可知:

函数内可任意使用主程序中的任何变量.函数内所启用的任何变量(除参数外),于该函数之外依然可以使用.此特性优劣参半,最大的坏处是式中的变量不易被保护,特别是递归调用本身,执行子函数时会破坏父函数内的变量.

一个变通的方法是:在函数的参数列中虚列一些参数.函数执行中使用这些虚列的参数来记录不想被破坏的数据,如此执行子函数时就不会破坏到这些数据.此外 awk 并不会检查调用函数时所传递的参数个数是否一致.

例如:定义递归函数如下:

```
function demo( arg1 ) { # 最常见的错误例子
.....
for(i=1; i< 20 ; i++){
demo(x)
# 又呼叫本身. 因为 i 是 global variable, 故执行完该子函数后
# 原函数中的 i 已经被坏, 故本函数无法正确执行.
.....
}
.....
}
```

可将上列函数中的 i 虚列在该函数的参数列上,如此 i 便是一个局部变量,不会因执行子函数而被破坏.

将上列函数修改如下:

```
function demo( arg1, i )
{
.....
for(i=1; i< 20; i++)
{
demo(x)#awk 不会检查呼叫函数时, 所传递的参数个数是否一致
.....
}
}
```

\$0, \$1,..., NF, NR,...也都是 global variable, 读者于递归函数中若有使用这些内建变量, 也应另外设立一些局部变量来保存, 以免被破坏.

[范例 :] 以下是一个常见的递归调用范例. 它要求使用者输入一串元素(各元素间用空白隔开) 然后印出这些元素所有可能的排列.

编辑如下的 awk 式, 取名为 permu

```
#!/bin/sh

awk '
BEGIN {
print "请输入排列的元素,各元素间请用空白隔开"
getline
permutation($0, "")
printf("\n 共 %d 种排列方式\n", counter)
}
function permutation( main_lst, buffer,      new_main_lst, nf, i, j )
{
    $0 = main_lst # 把 main_lst 指定给$0 之后 awk 将自动进行字段分割.
    nf = NF # 故可用 NF 表示 main_lst 上存在的元素个数.
    # BASE CASE : 当 main_lst 只有一个元素时.
    if( nf == 1){
        print buffer main_lst #buffer 的内容再加上 main_lst 就是完成
        counter++
        return
    }
    # General Case : 每次从 main_lst 中取出一个元素放到 buffer 中
    # 再用 main_lst 中剩下的元素 (new_main_lst) 往下进行排列
    else for( i=1; i<=nf ;i++)
    {
        $0 = main_lst # $0 为全局变量已被破坏, 故重新把 main_lst 赋
        给$0, 令 awk 再做一次字段分割
        new_main_lst = ""
        for(j=1; j<=nf; j++) # 连接 new_main_lst
            if( j != i ) new_main_lst = new_main_lst " " $j
        permutation( new_main_lst, buffer " " $i )
    }
}
' $*
```

执行

```
$ ./permu
```

屏幕上出现

请输入排列的元素,各元素间请用空白隔开

若输入 1 2 3 回车,结果印出

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

共 6 种排列方式

[说明:]

有些较旧版的 `awk`,并不容许使用者指定 `$0` 之值. 此时可改用 `gawk`, 或 `nawk`. 否则也可自行使用 `split()` 函数来分割 `main_lst`.

为避免执行子函数时破坏 `new_main_lst`, `nf`, `i`, `j` 故把这些变量也列于参数列上. 如此, `new_main_lst`, `nf`, `i`, `j` 将被当成局部变量,而不会受到子函数中同名的变量影响. 读者声明函数时,参数列上不妨将这些 "虚列的参数" 与真正用于传递信息的参数间以较长的空白隔开,以便于区别.

`awk` 中欲将字符串 `concatenation`(连接)时,直接将两字符串并置即可(Implicit Operator).

例如 :

```
awk '
BEGIN{
A = "This "
B = "is a "
C = A B "key." # 变量 A 与 B 之间应留空白,否则"AB"将代表另一新变量.
print C
}'
```

结果将印出

```
This is a key.
```

`awk` 使用者所编写的函数可再重用,并不需要每个 `awk` 式中都重新编写.

将函数部分单独编写于一文件中,当需要用到该函数时再以下列方式 `include` 进来.

```
$ awk -f 函数文件名 -f awk 主程序文件名 数据文件文件名
```

12.附录 A —— Pattern

`awk` 通过判断 `Pattern` 之值来决定是否执行其后所对应的 `Actions`. 这里列出几种常见的 `Pattern` :

➤BEGIN

`BEGIN` 为 `awk` 的保留字,是一种特殊的 `Pattern`.

`BEGIN` 成立(其值为 `true`)的时机是: "awk 程序一开始执行,尚未读取任何数据之前." 所以在

BEGIN { Actions } 语法中, 其 Actions 部份仅于程序一开始执行时被执行一次. 当 awk 从数据文件读入数据行后, BEGIN 便不再成立, 故不论有多少数据行, 该 Actions 部份仅被执行一次.

一般常把 "与数据文件内容无关" 与 "只需执行一次" 的部分置于该 Actions(以 BEGIN 为 Pattern)中.

例如:

```
BEGIN {  
  FS = "[\t:]" # 于程序一开始时, 改变 awk 切割字段的方式  
  RS = "" # 于程序一开始时, 改变 awk 分隔数据行的方式  
  count = 100 # 设定变量 count 的起始值  
  print " This is a title line " # 印出一行 title  
}  
..... # 其它 Pattern { Actions } .....
```

有些 awk 程序甚至"不需要读入任何数据行". 遇到这情况可把整个程序置于以 BEGIN 为 Pattern 的 Actions 中.

例如 :

```
BEGIN { print " Hello ! the Word ! " }
```

注意 : 执行该类仅含 BEGIN { Actions } 的程序时, awk 并不会开启任何数据文件进行处理.

➤END

END 为 awk 的保留字, 是另一种特殊的 Pattern.

END 成立(其值为 true)的时机与 BEGIN 恰好相反, 为:"awk 处理完所有数据, 即将离开程序时"平常读入数据行时, END 并不成立, 故其对应的 Actions 并不被执行; 唯有当 awk 读完所有数据时, 该 Actions 才会被执行

注意 : 不管数据行有多少笔, 该 Actions 仅被执行一次.

➤关系表达式

使用像 "A 关系运算符 B" 的表达式当成 Pattern.

当 A 与 B 存在所指定的关系(Relation)时, 该 Pattern 就算成立(true).

例如 :

```
length($0) <= 80 { print $0 }
```

上式中 length(\$0) <= 80 是一个 Pattern, 当 \$0(数据行)之长度小于等于 80 时该 Pattern 之值为 true, 将执行其后的 Action (打印该数据行).

awk 中提供下列 关系运算符(Relation Operator)

运算符 含意

> 大于

< 小于

>= 大于或等于

<= 小于或等于

== 等于

!= 不等于

~ match

!~ not match

上列关系运算符除~(match)与!~(not match)外与 C 语言中之含意一致.

~(match) 与 !~(match) 在 awk 之含意简述如下 :

若 A 为一字符串, B 为一正则表达式.

A ~B 判断 字符串 A 中是否 包含 能匹配(match)B 式样的子字符串.

A !~B 判断 字符串 A 中是否 未包含 能匹配(match)B 式样的子字符串.

例如 :

```
$0 ~ /program[0-9]+\./ { print $0 }
```

\$0 ~ /program[0-9]+\./ 整个是一个 Pattern, 用来判断\$0(数据行)中是否含有可 match /program[0-9]+\./ 的子字符串, 若\$0 中含有该类字符串, 则执行 print (打印该行数据).

Pattern 中被用来比对的字符串为\$0 时(如本例), 可仅以正则表达式部分表示整个 Pattern.

故本例的 Pattern 部分\$0 ~ /program[0-9]+\./ 可仅用 /program[0-9]+\./ 表之(有关匹配及正则表达式请参考 附录 E)

➤正则表达式

直接使用正则表达式当成 Pattern; 此为 \$0 ~ 正则表达式 的简写.

该 Pattern 用以判断 \$0(数据行) 中是否含有匹配该正则表达式的子字符串; 若含有该成立(true) 则执行其对应的 Actions.

例如 :

```
/^[0-9]*$/ { print "This line is a integer !" }
```

与 \$0 ~ /^[0-9]*\$/ { print "This line is a integer !" } 相同

➤混合 Pattern

之前所介绍的各种 Patterns, 其计算后结果为一逻辑值(True or False). awk 中逻辑值彼此间可通过&&(and), ||(or), !(not) 结合成一个新的逻辑值. 故不同 Patterns 彼此可通过上述结合符号来结合成一个新的 Pattern. 如此可进行复杂的条件判断.

例如 :

```
FNR >= 23 && FNR <= 28 { print " " $0 }
```

上式利用&& (and) 将两个 Pattern 求值的结果合并成一个逻辑值.

该式将数据文件中 第23行 到 28行 向右移 5 格(先输出 5 个空白字符)后输出.

(FNR 为 awk 的内建变量, 请参考 附录 D)

➤Pattern1 , Pattern2

遇到这种 Pattern, awk 会帮您设立一个 switch(或 flag).

当 awk 读入的数据行使得 Pattern1 成立时, awk 会打开(turn on)这 switch.

当 awk 读入的数据行使得 Pattern2 成立时, awk 会关上(turn off)这个 switch.

该 Pattern 成立的条件是 :

当这个 switch 被打开(turn on)时 (包括 Pattern1, 或 Pattern2 成立的情况)

例如：

```
FNR >= 23 && FNR <= 28 { print " " $0 }
```

可改写为

```
FNR == 23 , FNR == 28 { print " " $0 }
```

说明：

当 $FNR \geq 23$ 时, awk 就 turn on 这个 switch; 因为随着数据行的读入, awk 不停的累加 FNR. 当 $FNR = 28$ 时, Pattern2 ($FNR == 28$) 便成立, 这时 awk 会关上这个 switch.

当 switch 打开的期间, awk 会执行 `print " " $0`

(FNR 为 awk 的内建变量, 请参考 附录 D)

13.附录 B —— Actions

Actions 是由下列指令(statement)所组成：

- 表达式 (function calls, assignments..)
- print 表达式列表
- printf(格式化字符串, 表达式列表)
- if(表达式) 语句 [else 语句]
- while(表达式) 语句
- do 语句 while(表达式)
- for(表达式; 表达式; 表达式) 语句
- for(variable in array) 语句
- delete
- break
- continue
- next
- exit [表达式]
- 语句

awk 中大部分指令与 C 语言中的用法一致, 此处仅介绍较为常用或容易混淆的指令的用法.

➤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。

(c)第 5 节中有该指令的使用范例, 及解说.

- **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` 中的 I/O 指令

- **printf 指令**

该指令与 C 语言中的用法相同, 可借由该指令控制资料输出时的格式.

语法：

`printf("format", item1, item2, ...)`

范例：

```
id = "BE-2647"; ave = 89
printf("ID# : %s Ave Score : %d\n", id, ave)
```

(a)结果印出：

```
ID# : BE-2647 Ave Score : 89
```

(b)`format` 部分是由一般的字串(String Constant)及 格式控制字符(Format control letter, 其前会加上一个%字符)所构成. 以上式为例"`ID# :`"及 "`Ave Score :`"为一般字串. `%s` 及 `%d` 为格式控制字符.

(c)打印时, 一般字串将被原封不动地打印出来. 遇到格式控制字符时, 则依序把 `format` 后方之 `item` 转换成所指定的格式后进行打印.

(d)有关的细节, 读者可从介绍 C 语言的书籍上得到较完整的介绍.

(e)`print` 及 `printf` 两个指令, 其后可使用 `>` 或 `>>` 将输出到 `stdout` 的数据重定向到其它文件, 7.1 节中有完整的

- **print 指令**

范例：

```
id = "BE-267"; ave = 89
print "ID# :", id, "Ave Score :", ave
```

(a)结果印出：

```
ID# : BE-267 Ave Score :89
```

(b)`print` 之后可接上字串常数(Constant String)或变量. 它们彼此间可用`,"` 隔开.

(c)上式中, 字串 "ID#:" 与变量 id 之间使用","隔开, 打印时两者之间会以自动 OFS(请参考附录 D 内建变量 OFS) 隔开. OFS 之值一般内定为 "一个空格"

(d)上式中, 字串 "Ave Score:" 与变量 ave 之间并未以","隔开, awk 会将这两者先当成字串 concatenate 在一起(变成"Ave Score :89")后,再予打印

● **getline 指令**

语法

语法	由何处读取数据	数据读入后置于
getline var < file	所指定的 file	变量 var(var 省略时,表示置于\$0)
getline var	pipe 变量	变量 var(var 省略时,表示置于\$0)
getline var	见 注一	变量 var(var 省略时,表示置于\$0)

getline 一次读取一行资料, 若读取成功则 return 1,若读取失败则 return -1, 若遇到文件结束 (EOF), 则 return 0

● **close 指令**

该指令用以关闭一个打开的文件, 或 pipe (见下例)

范 例 :

```
BEGIN { print "ID #   Salary" > "data.rpt" }
{ print $1 , $2 * $3 | "sort -k 1 > data.rpt" }
END{ close( "data.rpt" )
      close( "sort -k 1 > data.rpt" )
      print " There are", NR, "records processed."
}
```

说 明 :

(a)上例中, 一开始执行 `print "ID # Salary">"data.rpt"` 指令来输出一行抬头. 它使用 I/O Redirection (>)将数据转输出到 data.rpt,故此时文件 data.rpt 是处于 Open 状态.

(b)指令 `print $1, $2 * $3` 不停的将输出的资料送往 pipe(), awk 在程序将结束时才会呼叫 shell 使用指令 `"sort -k 1 > data.rpt"` 来处理 pipe 中的数据; 并未立即执行, 这点与 Unix 中 pipe 的用法不尽相同.

(c)最后希望於文件 data.rpt 的末尾处加上一行 "There are.....".但此时, Shell 尚未执行 `"sort -k 1 > data.rpt"` 故各数据行排序后的 ID 及 Salary 等数据尚未写入 data.rpt. 所以得命令 awk 提前先通知 Shell 执行命令 `"sort -k 1 > data.rpt"` 来处理 pipe 中的资料. awk 中这个动作称为 close pipe. 是由执行 `close ("shell command")` 来完成. 需留心 close() 指令中的 shell command 需与|"后方的 shell command 完全相同(一字不差), 较佳的方法是先以该字串定义一个简短的变量, 程序中再以此变量代替该 shell command

(d)为什么执行 `close("data.rpt")`? 因为 sort 完后的资料也将写到 data.rpt,而该文件正为 awk 所打开使用(write)中, 故 awk 程式中应先关闭 data.rpt. 以免造成因二个 processes 同时打开一个文件进行输出(write)所产生的错误.

- **system 指令**

该指令用以执行 Shell 上的 `command`.

范 例 :

```
DataFile = "invent.rpt"
system( "rm " DataFile )
```

说明 :

(a)`system("字符串")`指令接受一个字符串当成 Shell 的命令. 上例中, 使用一个字符串常数"rm " 连接(concat)一个变量 `DataFile` 形成要求 Shell 执行的命令.Shell 实际执行的命令为 "rm invent.rpt".

- **"|" pipe 指令**

"|" 配合 `awk` 输出指令, 可把 `output` 到 `stdout` 的资料继续转送给 Shell 上的某一命令当成 `input` 的资料.

"|" 配合 `awk` `getline` 指令, 可呼叫 Shell 执行某一命令, 再以 `awk` 的 `getline` 指令将该命令的所产生的资料读进 `awk` 程序中.

范 例 :

```
{ print $1, $2 * $3 | "sort -k 1 > result" }
"date" | getline Date_data
```

读者请参考 7.2 节, 其中有完整的范例说明.

➤**awk 释放所占用的记忆体的指令**

`awk` 程式中常使用数组(Array)来记忆大量数据, `delete` 指令便是用来释放数组中的元素所占用的内存空间.

范 例 :

```
for( any in X_arr )
    delete X_arr[any]
```

读者请留心, `delete` 指令一次只能释放数组中的一个元素.

➤**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)

➤awk 中各运算符的运算级

按优先高低排列:

\$ (栏位运算元, 例如 : i=3; \$i 表示第 3 栏)

^ (指数运算)

+, -, ! (正, 负号, 及逻辑上的 not)

*, /, % (乘, 除, 余数)

+, - (加, 减)

>, >=, <, <=, ==, != (关系运算符)

~, !~ (match, not match)

&& (逻辑上的 and)

|| (逻辑上的 or)

? : (条件运算符)

=, +=, -=, *=, /=, %=, ^= (赋值运算符)

14.附录 C —— awk 的内建函数 (Built-in Functions)

➤(一). 字符串函数

- **index(原字符串, 找寻的子字符串):**

若原字符串中含有欲找寻的子字符串,则返回该子字符串在原字符串中第一次出现的位置,若未曾出现该子字符串则返回 0.

例如执行:

```
$ awk 'BEGIN{ print index("8-12-94","-") }'
```

结果印出

```
2
```

- **length(字符串):** 返回该字符串的长度.

例如执行:

```
$ awk 'BEGIN { print length("John") }'
```

结果印出

```
4
```

- **match(原字符串, 用以找寻比对的 正则表达式):**

awk 会在原字符串中找寻合乎正则表达式的子字符串. 若合乎条件的子字符串有多个, 则以原字符串中最左方的子字符串为准.

awk 找到该字符串后会依此字符串为依据进行下列动作:

设定 awk 内建变量 RSTART, RLENGTH:

RSTART = 合条件的子字符串在原字符串中的位置.

= 0; 若未找到合条件的子字符串.

RLENGTH = 合条件的子字符串长度.

= -1; 若未找到合条件的子字符串.

返回 RSTART 之值.

例如执行:

```
awk ' BEGIN {  
    match( "banana", /(an)+/ )  
    print RSTART, RLENGTH  
} '
```

执行结果输出

```
2 4
```

- **split(原字符串,数组名称,分隔字符):**

awk 将依所指定的分隔字符(field separator)来分隔原字符串成一个个的栏位(field),并以指定的数组记录各个被分隔的栏位.

例如 :

```
ArgLst = "5P12p89"  
split( ArgLst, Arr, /[Pp]/)
```

执行后 : Arr[1]=5, Arr[2]=12, Arr[3]=89

- **sprintf(格式字符串,项 1,项 2,...)**

该函数的用法与 awk 或 C 的输出函数 printf()相同. 所不同的是 sprintf()会将要求印出的结果当成一个字串返回. 一般最常使用 sprintf()来改变资料格式. 如: x 为一数值资料, 若欲将其变成一个含二位小数的资料,可执行如下指令:

```
x = 28  
x = sprintf("%.2f",x)
```

执行后 x = "28.00"

- **sub(比对用的 正则表达式,将替换的新字符串,原字符串)**

sub()将原字符串中第一个(最左边)合乎所指定的正则表达式的子字符串改以新字符串取代.

第二个参数"将替换的新字符串"中可用"&"来代表"合於条件的子字符串"

承上例,执行下列指令:

```
A = "a6b12anan212.45an6a"  
sub( /(an)+[0-9]*/, "[&]", A)  
print A
```

结果输出

```
ab12[anan212].45an6a
```

sub()不仅可执行替换(replacement)的功用,当第二个参数为空字符串("")时,sub()所执行的是"去除指定字符串"的功用.

通过 sub() 与 match() 的搭配使用,可逐次取出原字符串中合乎指定条件的所有子字符串.

例如执行下列程式:

```
awk '  
BEGIN {  
data = "p12-P34 P56-p61"  
while( match( data ,/[0-9]+/) > 0) {  
print substr(data, RSTART, RLENGTH )  
sub(/[0-9]+/, "",data)  
}  
'
```

结果输出 :

```
12  
34  
56
```

sub()中第三个参数(原字符串)若未指定,则其预设值为\$0.

可用 `sub(/[9-0]+/, "digital")` 表示 `sub(/[0-9]+/, "digital", $0)`

- **gsub(比对用的 正则表达式 , 将替换的新字符串 , 原字符串)**

这个函数与 sub()一样,同样是进行字符串取代的函数. 唯一不同点是

gsub()会取代所有符合条件的子字符串.

gsub()会返回被取代的子字符串个数.

请参考 sub().

- **substr(字符串,起始位置 [,长度]):**

返回从起始位置起,指定长度的子字符串. 若未指定长度,则返回起始位置到字符串末尾的子字符串.

执行下例

```
$ awk 'BEGIN { print substr("User:Wei-Lin Liu", 6) }'
```

结果印出

```
Wei-Lin Liu
```

➤(二). 数学函数

- **int(x): 返回 x 的整数部分(去掉小数).**

例如 :

int(7.8) 将返回 7

int(-7.8) 将返回 -7

- **sqrt(x): 返回 x 的平方根 .**

例如 :

sqrt(9) 将返回 3

若 x 为负数,则执行 sqrt(x)时将造成 Run Time Error [译者注: 我这里没有发生错误,返回的是"nan"]

- **exp(x): 将返回 e 的 x 次方 .**

例如 :

exp(1) 将返回 2.71828

- **log(x): 将返回 x 以 e 为底的对数值 .**

例如 :

log(exp(1)) 将返回 1

若 $x < 0$, 则执行 sqrt(x)时将造成 Run Time Error. [译者注: 我这里也没有发生错误,返回的

是"nan"]

- **sin(x)**: x 须以弧度为单位, sin(x) 将返回 x 的 sin 函数值.
- **cos(x)**: x 须以弧度 为单位, cos(x) 将返回 x 的 cos 函数值
- **atan2(y,x)**: 返回 y/x 的 tan 反函数之值, 返回值系 以弧度为单位 .
- **rand()**: 返回介于 0 与 1 之间的 (近似) 随机数值 ; $0 < \text{rand()} < 1$.

除非使用者自行指定 rand() 函数起始的种子, 否则每次执行 awk 程式时, rand() 函数都将使用同一个内定的种子, 来产生随机数.

- **srand([x])**: 指定以 x 为 rand() 函数起始 的种子.

若省略了 x, 则 awk 会以执行时的日期与时间为 rand() 函数起始的种子.

15. 附录 D —— awk 的内建变量 Built-in Variables

因内建变量的个数不多, 此处按其相关性分类说明, 并未按其字母顺序排列.

- **ARGC**

ARGC 表示命令行上除了选项 -F, -v, -f 及其所对应的参数之外的所有参数的个数. 若将 "awk 程式" 直接写於命令列上, 则 ARGC 亦不将该 "程式部分" 列入计算.

- **ARGV**

ARGV 数组用以记录命令列上的参数.

例 : 执行下列命令

```
$ awk -F\t -v a=8 -f prg.awk file1.dat file2.dat
```

或

```
$ awk -F\t -v a=8 '{ print $1 * a }' file1.dat file2.dat
```

执行上列任一程式后

```
ARGC = 3
```

```
ARGV[0] = "awk"
```

```
ARGV[1] = "file1.dat"
```

```
ARGV[2] = "file2.dat"
```

读者请留心 : 当 $\text{ARGC} = 3$ 时, 命令列上仅指定了 2 个文件.

注 :

-F\t 表示以 tab 为栏位分隔字符 FS(field separator).

-v a=8 是用以初始化程序中的变量 a.

- **FILENAME**

FILENAME 用以表示目前正在处理的文件档名.

- **FS**

栏位分隔字符.

- **\$0**

表示目前 awk 所读入的数据行.

- **\$1,\$2..**

分别表示所读入的数据行之第一栏, 第二栏,..

说明:

当 awk 读入一笔数据行 "A123 8:15" 时,会先以\$0 记录.

故 \$0 = "A123 8:15"

若程序中进一步使用了 \$1,\$2.. 或 NF 等内建变量时,awk 才会自动分割 \$0.

以便取得栏位相关的资料. 切割后各个栏位的资料会分别以\$1, \$2, \$3...予以记录.

awk 内定(default)的 栏位分隔字符(FS)为 空白字符(空格及 tab).

以本例而言, 读者若未改变 FS, 则分割后:

第一栏(\$1)="A123", 第二栏(\$2)="8:15".

使用者可用正则表达式自行定义 FS. awk 每次需要分割数据行时, 会参考目前 FS 的值.

例如 :

令 FS = "[:]+" 表示任何由 空白" " 或 冒号":" 所组成的字串都可当成分隔字符, 则分割后 :

第一栏(\$1) = "A123", 第二栏(\$2) = "8", 第三栏(\$3) = "15"

- **NR**

NR 表从 awk 开始执行该程序后所读取的数据行数.

- **FNR**

FNR 与 NR 功用类似. 不同的是 awk 每打开一个新的文件,FNR 便从 0 重新累计

- **NF**

NF 表目前的数据行所被切分的栏位数.

awk 每读入一笔资料后, 在程序中可以 NF 来得知该行数据包含的栏位个数.在下一笔资料被读入之前, NF 并不会改变. 但使用者若自行使用\$0 来记录数据,例如: 使用 `getline` , 此时 NF 将代表新的 \$0 上所记载的资料的栏位个数.

- **OFS**

OFS 输出时的栏位分隔字符. 预设值 " "(一个空白), 详见下面说明.

- **ORS**

ORS 输出时数据行的分隔字符. 预设值 "\n"(跳行), 见下面说明.

- **OFMT**

OFMT 数值资料的输出格式. 预设值 "%.6g"(若须要时最多印出 6 位小数)

当使用 `print` 指令一次印出多项资料时,

例如 : `print $1, $2`

输出时, awk 会自动在 \$1 与 \$2 之间补上一个 OFS 之值

每次使用 `print` 输出后, `awk` 会自动补上 `ORS` 之值.

使用 `print` 输出数值数据时, `awk` 将采用 `OFMT` 之值为输出格式.

例如 :

```
$ awk 'BEGIN { print 2/3,1; OFS=":"; OFMT="%.2g"; print 2/3,1 }'
```

输出:

```
0.666667 1
0.67:1
```

程序中通过改变 `OFS` 和 `OFMT` 的值, 改变了指令 `print` 的输出格式.

● RS

RS(Record Separator): `awk` 从文件上读取资料时, 将根据 **RS** 的定义把资料切割成许多 **Records**, 而 `awk` 一次仅读入一个 **Record**, 以进行处理.

RS 的预设值是 `"\n"`. 所以一般 `awk` 一次仅读入一行资料.

有时一个 **Record** 含括了几行资料(**Multi-line Record**). 这情况下不能再以 `"\n"`

来分隔相邻的 **Records**, 可改用 空白行 来分隔.

在 `awk` 程式中, 令 `RS = ""` 表示以 空白行 来分隔相邻的 **Records**.

● RSTART

RSTART 与使用字串函数 `match()` 有关的变量, 详见下面说明.

● RLENGTH

RLENGTH 与使用字串函数 `match()` 有关之变量.

当使用者使用 `match(...)` 函数后, `awk` 会将 `match(...)` 执行的结果以 **RSTART**, **RLENGTH** 记录.

请参考 附录 C `awk` 的内建函数 `match()`.

● SUBSEP

SUBSEP(Subscript Separator) 数组下标的分隔字符,

预设值为 `"\034"` 实际上, `awk` 中的 数组 只接受 字串 当它的下标, 如: `Arr["John"]`.

但使用者在 `awk` 中仍可使用 数字 当阵列的下标, 甚至可使用多维的数组(**Multi-dimensional Array**) 如: `Arr[2,79]`

事实上, `awk` 在接受 `Arr[2,79]` 之前, 就已先把其下标转换成字串 `"2\03479"`, 之后便以 `Arr["2\03479"]` 代替 `Arr[2,79]`.

可参考下例 :

```
awk 'BEGIN {
Arr[2,79] = 78
print Arr[2,79]
print Arr[ 2 , 79 ]
print Arr["2\03479"]
idx = 2 SUBSEP 79
print Arr[idx]
```

```
}  
, $*
```

执行结果输出:

```
78  
78  
78  
78
```

16.附录 E —— 正则表达式(Regular Expression) 简介

● 为什么要使用正则表达式

UNIX 中提供了许多 指令 和 tools, 它们具有在文件中 查找(Search)字串或替换(Replace)字串 的功能. 像 grep, vi, sed, awk,...

不论是查找字串或替换字串, 都得先告诉这些指令所要查找(被替换)的字串为何. 若未能预先明确知道所要查找(被替换)的字串为何, 只知该字串存在的范围或特征时, 例如 :

(一)找寻 "T0.c", "T1.c", "T2.c".... "T9.c" 当中的任一字串.

(二)找寻至少存在一个 "A"的任意字串.

这情况下, 如何告知执行查找字串的指令所要查找的字串为何.

例 (一) 中, 要查找任一在 "T" 与 ".c" 之间存在一个阿拉伯数字的字串;当然您可以列举的方式, 一把所要找寻的字串告诉执行命令的指令. 但例 (二) 中合乎该条件的字串有无限种可能, 势必无法一一列举. 此时, 便需要另一种字串表示的方法(协定).

● 什么是正则表达式

正则表达式(以下简称 Regexp)是一种字串表达的方式. 可用以指定具有某特征的所有字串.

注: 为区别于一般字串, 本附录中代表 Regexp 的字串之前皆加 "Regexp". awk 程式中常以/.... /括住 Regexp; 以区别于一般字串.

● 组成正则表达式的元素

普通字符 除了 . * [] + ? () \ ^ \$ 外之所有字符.

由普通字符所组成的 Regexp 其意义与原字串字面意义相同.

例如: Regexp "the" 与一般字串的 "the" 代表相同的意义.

. (Meta character) : 用以代表任意一字符.

须留心 UNIX Shell 中使用 "*"表示 Wild card, 可用以代表任意长度的字串. 而 Regexp 中使用 "." 来代表一个任意字符.(注意: 并非任意长度的字串)Regexp 中 "*" 另有其它涵意, 并不代表任意长度的字串.

^ 表示该字串必须出现于行首.

\$ 表示该字串必须出现于行末.

例如 :

Regexp /[^]The/ 用以表示所有出现于行首的字串 "The".

Regexp /The\$/ 用以表示所有出现于行末字串 "The".

\ 将特殊字符还原成字面意义的字符(Escape character)

Regexp 中特殊字符将被解释成特定的意义. 若要表示特殊字符的字面(literal meaning)意义时, 在特殊字符之前加上\"即可.

例如 :

使用 Regexp 来表示字串 "a.out"时, 不可写成 /a.out/.

因为 "."是特殊字符, 表任一字符. 可符合 Regexp /a.out/ 的字串将不只 "a.out" 一个; 字串 "a2out", "a3out", "aaout" ...都符合 Regexp /a.out/ 正确的用法为: /a\.out/

[...]字符集合, 用以表示两中括号间所有的字符当中的任一个.

例如:

Regexp /[Tt]/ 可用以表示字符 "T" 或 "t".故 Regexp /[Tt]he/ 表示 字串 "The" 或 "the".

字符集合 [...] 内不可随意留空白.

例如: Regexp /[Tt]/ 其中括号内有空白字符, 除表示"T", "t" 中任一字符, 也可代表一个 " "(空白字符)

- 字符集合中可使用 "-" 来指定字符的区间, 其用法如下:

Regexp /[0-9]/ 等于 /[0123456789]/ 用以表示任意一个阿拉伯数字.

同理 Regexp /[A-Z]/ 用以表示任意一个大写英文字母.

但应留心:

Regexp /[0-9a-z]/ 并不等于 /[0-9][a-z]/; 前者表示一个字符,后者表示二个字符.

Regexp /[-9]/ 或 /[9-]/ 只代表字符 "9"或 "-".

[^...]使用[^..] 产生字符集合的补集(complement set).

其用法如下 :

例如: 要指定 "T" 或 "t" 之外的任一字符, 可用 /^[^Tt]/ 表之.

同理 Regexp /^[^a-zA-Z]/ 表示英文字母之外的任一字符.

须留心 "^" 的位置 : "^"必须紧接於 "["之后, 才代表字符集合的补集

例如 :Regexp /[0-9^]/ 只是用以表示一个阿拉伯数字或字符"^".

* 形容字符重复次数的特殊字符.

"*" 形容它前方之字符可出现 1 次或多次, 或不出现(0 次).

例如:

Regex `/T[0-9]*\.c/` 中 `*` 形容其前 `[0-9]` (一个阿拉伯数字)出现的次数可为 0 次或 多次.故
Regex `/T[0-9]*\.c/` 可用以表示 "T.c", "T0.c", "T1.c"... "T19.c"

`+`形容其前的字符出现一次或一次以上.

例如:

Regex `/[0-9]+/` 用以表示一位或一位以上的数字.

`?` 形容其前的字符可出现一次或不出现.

例如:

Regex `/[+-]?[0-9]+/` 表示数字(一位以上)之前可出现正负号或不出现正负号.

(...)用以括住一群字符,且将之视成一个 group(见下面说明)

例如 :

Regex `/12+/` 表示字串 "12", "122", "1222", "12222",...

Regex `/(12)+/` 表示字串 "12", "1212", "121212", "12121212"....

上式中 12 以()括住,故 `+` 所形容的是 12, 重复出现的也是 12.

`|` 表示逻辑上的"或"(or)

例如:

Regex `/Oranges?|apples?|water/` 可用以表示 : 字串 "Orange", "Oranges" 或 "apple",
"apples" 或 "water"

● match 是什么 ?

讨论 Regex 时,经常遇到 "某字串匹配(match)某 Regex"的字眼. 其意思为 : "这个
Regex 可被解释成该字串".

[例如] :

字串 "the" 匹配(match) Regex `/[Tt]he/`.

因为 Regex `/[Tt]he/` 可解释成字串 "the" 或 "The", 故字串 "the" 或 "The"都匹配(match)
Regex `/[Tt]he/`.

● awk 中提供二个关系 运算符 (Relational Operator,见注一) `~` `!~`,

它们也称之为 match, not match.但函义与一般常称的 match 略有不同.

其定义如下:

A 表一字串, B 表一 Regular Expression

只要 A 字串中存在有子字串可 match(一般定义的 match) Regex B, 则 `A~B` 就算成立, 其
值为 true, 反之则为 false.

`!~` 的定义与`~`恰好相反.

例 如 :

"another" 中含有子字符串 "the" 可 match Regexp /[Tt]he/, 所以

"another" ~ /[Tt]he/ 之值为 true.

[注 一]: 有些论著不把这两个运算符(~, !~)与 Relational Operators归为一类.

● 应用 Regular Expression 解题的简 例

下面列出一些应用 Regular Expression 的简例, 部分范例中会更改\$0 之值, 若您使用的 awk 不允许用户更改 \$0 时, 请改用 gawk.

例 1:

将文件中所有的字符串 "Regular Expression" 或 "Regular expression" 换成 "Regexp"

```
awk '
{ gsub( /Regular[ \t]+[Ee]xpression/, "Regexp")
print
}
' $*
```

例 2:

去除文件中的空白行(或仅含空白字符或 tab 的行)

```
awk ' $0 !~ /^[ \t]*$/ { print } ' $*
```

例 3:

在文件中具有 ddd-dddd (电话号码型态, d 表 digital)的字符串前加上"TEL: "

```
awk '
{ gsub( /[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]/, "TEL : &" )
print
}
' $*
```

例 4:

从文件的 Fullname 中分离出 路径 与 档名

```
awk '
BEGIN{
Fullname = "/usr/local/bin/xdvi"
match( Fullname, /.*/ )
path = substr(Fullname, 1, RLENGTH-1)
name = substr(Fullname, RLENGTH+1)
print "path :", path, " name :", name
}
' $*
```

结果印出

```
path : /usr/local/bin  name : xdvi
```

例 5:

将某一数值改以现金表示法表示(整数部分每三位加一撇,且含二位小数)

```
awk '
BEGIN {
```

```

Number = 123456789
Number = sprintf("%.2f",Number)
while( match(Number,/ [0-9][0-9][0-9][0-9]/ ) )
    sub(/ [0-9][0-9][0-9][.]/, ",&", Number)
print Number
}
' $*

```

结果输出

```
$123,456,789.00
```

例 6:

把文件中所有具 "program 数字.f"形态的字串改为"[Ref : program 数字.c]"

```

awk '
{
while( match( $0, /program[0-9]+\f/ ) ){
    Replace = "[Ref : " substr( $0, RSTART, RLENGTH-2) ".c]"
    sub( /program[0-9]+\f/, Replace)
}
print
}
' $*

```