Shell脚本编程的常识

（这些往往是经常用到，但是各种网络上的材料都语焉不详的东西，个人认为比较有用）

七种文件类型

d 目录 l 符号链接

s 套接字文件 b 块设备文件

c 字符设备文件 p 命名管道文件

- 普通文件

正则表达式

从一个文件或命令输出中抽取或过滤文本时。可使用正则表达式（RE），正则表达式是一些特殊或不很特殊的字符串模式的集合。

基本的元字符集：

^ 只匹配行首。

$ 只匹配行尾。

\* 一个单字符后紧跟\*，匹配0个或多个此单字符。

[] 匹配[]内字符，可以是一个单字符，也可以是字符序列。可以使

用-来表示[]内范围，如[1-5]等价于[1,2,3,4,5]。

\ 屏蔽一个元字符的特殊含义，如\$表示字符$，而不表示匹配行

尾。

. 匹配任意单字符。

pattern\{n\} 匹配pattern出现的次数n

pattern\{n,\}m匹配pattern出现的次数，但表示次数最少为n

pattern\{n,m\} 匹配pattern出现的次数在n与m之间（n，m为0－255）

几个常见的例子：

显示可执行的文件：ls –l | grep …x...x..x

只显示文件夹：ls –l | grep ^d

匹配所有的空行：^$

匹配所有的单词：[A-Z a-z]\*

匹配任一非字母型字符：[^A-Z a-z]

包含八个字符的行：^……..$(8个.)

字符类描述

以下是可用字符类的相当完整的列表：

[:alnum:] 字母数字 [a-z A-Z 0-9]

[:alpha:] 字母 [a-z A-Z]

[:blank:] 空格或制表键

[:cntrl:] 任何控制字符

[:digit:] 数字 [0-9]

[:graph:] 任何可视字符（无空格）

[:lower:] 小写 [a-z]

[:print:] 非控制字符

[:punct:] 标点字符

[:space:] 空格

[:upper:] 大写 [A-Z]

[:xdigit:] 十六进制数字 [0-9 a-f A-F]

尽可能使用字符类是很有利的，因为它们可以更好地适应非英语 locale（包括某些必需的重音字符等等）.

shell的引号类型

shell共有四种引用类型：

“ ” 双引号

‘ ’ 单引号

` ` 反引号

\ 反斜线

l “ ” 可引用除$、` 、\ 、外的任意字符或字符串，“ ”中的变量能够正常显示变量值。

l ‘ ’与“ ”类似，不同在于shell会忽略任何的引用值。

例如： GIRL=‘girl’

echo “The ‘$GIRL’ did well”

则打印：The ‘girl’ did well

l ` `用于设置系统命令的输出到变量，shell会将` `中的内容作为一个系统命令并执行质。

例如：echo `date` 则打印当前的系统时间。

l \ 用来屏蔽特殊含义的字符：& \* + ^ $ ` “ | ?

例如：expr 12 \\* 12 将输出144

变量设置时的不同模式：

valiable\_name=value 设置实际值到 variable\_name中

valiable\_name+value 如果设置了variable\_name，则重设其值

valiable\_name:?value 如果未设置variable\_name，则先显示未定义用户错误信息

valiable\_name?value 如果未设置variable\_name，则显示系统错误信息

valiable\_name:=value 如果未设置variable\_name，则设置其值

valiable\_name-value 同上，但取值并不设置到variable\_name

条件测试

test命令用于测试字符串、文件状态和数字，expr测试和执行数值输出。

Test格式：test condition 或 [ condition ](需要特别注意的是condition的两边都要有一个空格，否则会报错)，test命令返回0表示成功。

l 下面将分别描述test的三种测试：

n 文件状态测试(常用的)

-d 测试是否文件夹

-f 测试是否一般文件

-L 测试是否链接文件

-r 测试文件是否可读

-w 测试文件是否可写

-x 测试文件是否可执行

-s 测试文件是否非空

n 字符串测试

五种格式： test “string”

test string\_operator “string”

test “string” string\_operator “string”

[ string\_operator “string” ]

[ “string” string\_operator “string” ]

其中string\_operator可以为： = 两字符串相等

!= 两字符串不等

-z 空串

-n 非空串

n 数值测试

两种格式： “number” number\_operator “number”

[ “number” number\_operator “number” ]

其中：number\_operator 可以为：-eq 、-ne、-gt、-lt、-ge

例如： NUMBER=130

[ “990” –le “995” –a “NUMBER” -gt “133” ]

(其中-a表示前后结果相“与”)

l expr命令一般用于整数值，但也可以用于字符串。

n 格式： expr srgument operator operator argument

例如： expr 10 + 10

expr 10 ^ 2 (10的平方)

expr $value + 10

n 增量计数――expr在循环中最基本的用法

例如： LOOP=0

LOOP=`expr $LOOP + 1`

n 模式匹配：通过指定的冒号选项计算字符串中的字符数

例如： value=account.doc

expr $value : `\(.\*\).doc`

输出 account

命令执行顺序

&& 成功执行一个命令后再执行下一个

|| 一个命令执行失败后再执行另一个命令

( ) 在当前shell中执行一组命令（格式：(命令1;命令2; ……)）

{ } 同( )

例如： comet mouth\_end || ( echo “hello” | mail dave ;exit )

如果没有( )，则shell将直接执行最后一个命令（exit）

脚本调试

最有用的调试脚本的工具是echo命令，可以随时打印有关变量或操作的信息，以帮助定位错误。也可使用打印最后状态($?) 命令来判断命令是否成功，这时要注意的是要在执行完要测试的命令后立即输出$?，否则$?将会改变。

Set命令也可以用来辅助脚本测试：

Set –n 读命令但是不执行

Set –v 显示读取的所有的行

Set –x 显示所有的命令及其参数

（要关闭set选项，只要把－换成+就可以了，这里有点特殊，要注意一下）

一些常用的小trick

打印一些头信息

command << dilimiter

……

……

dilimiter

以分界符号dilimiter中的内容作为命令的标准输入

常用在echo命令中，这样就避免了没输出一行就要使用一个echo命令，同时，输出格式的调整也相应变得简单了。

例如： echo << something\_message

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

hello, welcome to use my shell script

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

something\_message

将在屏幕上输出：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

hello, welcome to use my shell script

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

一、利用<<的分解符号性质还可以自动选择菜单或实现自动的ftp传输

也就是利用分解符号的性质自动选择菜单。

例如： ./menu\_choose >>output\_file 2>&1 <<Choose

2

3

Y

Choose

则自动在执行脚本的过程中一步步作出选择：2，3，Y

<<这种性质决定了它是理想的访问数据库的有用工具，可以用它来输入面对数据库提示时所作的各种选择。

创建一个长度为0的空文件

执行 > file\_name 命令或 touch file\_name 命令。

一些常用的shell变量

$# 传递到脚本的参数个数

$\* 以一个单字符串显示所有向脚本传递的参数（可大于9个）

$$ 脚本运行的当前进程的ID号

$! 后台运行的最后一个进程的ID号

$@ 与$#相同，但使用时加引号，并在引号中返回每个参数

$- 显示shell使用的当前选项

$? 显示最后命令的退出状态，0表示无错误（这个变量也常常用来打印输出，在脚本调试时标记某个shell命令或某个函数是否正确执行，但是要注意，$?记载的是最近的函数或命令的退出状态，因此打印时应该立即打印以获得正确的信息）

$0的使用

在变量中有一种位置变量$n，用来存放函数调用或脚本执行时传入的参数，其中$0表示函数名或脚本名，需要注意的是，这时的脚本名传递的是包含全路径的脚本名。从$1-$9表示传入的第一到第九个参数，这样的参数表示不能多于九个，如果多于九个，可以使用下面将要提到的shift指令来读取。

因为$0存放函数名或脚本名，因此我们可以通过echo $0来输出调用信息，但是，由于存放的是全路径名，我们可以利用一个shell命令来得到脚本名，basename $0 将得到$0中名字的部分，而与之相反的，dirname $0将得到$0中路径的部分。

Shift的运用

用head或tail指令指定查阅的行数

例如：查阅文件前20行： head –20 file\_name

查阅文件后10行： tail –10 file\_name

awk使用规则

awk 是一种很棒的语言。awk 适合于文本处理和报表生成，它还有许多精心设计的特性，允许进行需要特殊技巧程序设计。与某些语言不同，awk 的语法较为常见。它借鉴了某些语言的一些精华部分，如 C 语言、python 和 bash（虽然在技术上，awk 比 python 和 bash 早创建）。awk 是那种一旦学会了就会成为您战略编码库的主要部分的语言。

第一个 awk

让我们继续，开始使用 awk，以了解其工作原理。在命令行中输入以下命令：

$ awk '{ print }' /etc/passwd

您将会见到 /etc/passwd 文件的内容出现在眼前。现在，解释 awk 做了些什么。调用 awk 时，我们指定 /etc/passwd 作为输入文件。执行 awk 时，它依次对 /etc/passwd 中的每一行执行 print 命令。所有输出都发送到 stdout，所得到的结果与与执行catting /etc/passwd完全相同。

现在，解释 { print } 代码块。在 awk 中，花括号用于将几块代码组合到一起，这一点类似于 C 语言。在代码块中只有一条 print 命令。在 awk 中，如果只出现 print 命令，那么将打印当前行的全部内容。

这里是另一个 awk 示例，它的作用与上例完全相同：

$ awk '{ print $0 }' /etc/passwd

在 awk 中，$0 变量表示整个当前行，所以 print 和 print $0 的作用完全一样。

如果您愿意，可以创建一个 awk 程序，让它输出与输入数据完全无关的数据。以下是一个示例：

$ awk '{ print "" }' /etc/passwd

只要将 "" 字符串传递给 print 命令，它就会打印空白行。如果测试该脚本，将会发现对于 /etc/passwd 文件中的每一行，awk 都输出一个空白行。再次说明， awk 对输入文件中的每一行都执行这个脚本。以下是另一个示例：

$ awk '{ print "hiya" }' /etc/passwd

运行这个脚本将在您的屏幕上写满 hiya。:)

多个字段

awk 非常善于处理分成多个逻辑字段的文本，而且让您可以毫不费力地引用 awk 脚本中每个独立的字段。以下脚本将打印出您的系统上所有用户帐户的列表：

$ awk -F":" '{ print $1 }' /etc/passwd

上例中，在调用 awk 时，使用 -F 选项来指定 ":" 作为字段分隔符。awk 处理 print $1 命令时，它会打印出在输入文件中每一行中出现的第一个字段。以下是另一个示例：

$ awk -F":" '{ print $1 $3 }' /etc/passwd

以下是该脚本输出的摘录：

halt7

operator11

root0

shutdown6

sync5

bin1

....etc.

如您所见，awk 打印出 /etc/passwd 文件的第一和第三个字段，它们正好分别是用户名和用户标识字段。现在，当脚本运行时，它并不理想 -- 在两个输出字段之间没有空格！如果习惯于使用 bash 或 python 进行编程，那么您会指望 print $1 $3 命令在两个字段之间插入空格。然而，当两个字符串在 awk 程序中彼此相邻时，awk 会连接它们但不在它们之间添加空格。以下命令会在这两个字段中插入空格：

$ awk -F":" '{ print $1 " " $3 }' /etc/passwd

以这种方式调用 print 时，它将连接 $1、" " 和 $3，创建可读的输出。当然，如果需要的话，我们还可以插入一些文本标签：

$ awk -F":" '{ print "username: " $1 "\t\tuid:" $3" }' /etc/passwd

这将产生以下输出：

username: halt uid:7

username: operator uid:11

username: root uid:0

username: shutdown uid:6

username: sync uid:5

username: bin uid:1

....etc.

外部脚本

将脚本作为命令行自变量传递给 awk 对于小的单行程序来说是非常简单的，而对于多行程序，它就比较复杂。您肯定想要在外部文件中撰写脚本。然后可以向 awk 传递 -f 选项，以向它提供此脚本文件：

$ awk -f myscript.awk myfile.in

将脚本放入文本文件还可以让您使用附加 awk 功能。例如，这个多行脚本与前面的单行脚本的作用相同，它们都打印出 /etc/passwd 中每一行的第一个字段：

BEGIN {

FS=":"

}

{ print $1 }

这两个方法的差别在于如何设置字段分隔符。在这个脚本中，字段分隔符在代码自身中指定（通过设置 FS 变量），而在前一个示例中，通过在命令行上向 awk 传递 -F":" 选项来设置 FS。通常，最好在脚本自身中设置字段分隔符，只是因为这表示您可以少输入一个命令行自变量。我们将在本文的后面详细讨论 FS 变量。

BEGIN 和 END 块

通常，对于每个输入行，awk 都会执行每个脚本代码块一次。然而，在许多编程情况中，可能需要在 awk 开始处理输入文件中的文本之前执行初始化代码。对于这种情况，awk 允许您定义一个 BEGIN 块。我们在前一个示例中使用了 BEGIN 块。因为 awk 在开始处理输入文件之前会执行 BEGIN 块，因此它是初始化 FS（字段分隔符）变量、打印页眉或初始化其它在程序中以后会引用的全局变量的极佳位置。

awk 还提供了另一个特殊块，叫作 END 块。awk 在处理了输入文件中的所有行之后执行这个块。通常，END 块用于执行最终计算或打印应该出现在输出流结尾的摘要信息。

规则表达式和块

awk 允许使用规则表达式，根据规则表达式是否匹配当前行来选择执行独立代码块。以下示例脚本只输出包含字符序列 foo 的那些行：

/foo/ { print }

当然，可以使用更复杂的规则表达式。以下脚本将只打印包含浮点数的行：

/[0-9]+\.[0-9]\*/ { print }

还有许多其它方法可以选择执行代码块。我们可以将任意一种布尔表达式放在一个代码块之前，以控制何时执行某特定块。仅当对前面的布尔表达式求值为真时，awk 才执行代码块。以下示例脚本输出将输出其第一个字段等于 fred 的所有行中的第三个字段。如果当前行的第一个字段不等于 fred，awk 将继续处理文件而不对当前行执行 print 语句：

$1 == "fred" { print $3 }

awk 提供了完整的比较运算符集合，包括 "=="、"<"、">"、"<="、">=" 和 "!="。另外，awk 还提供了 "~" 和 "!~" 运算符，它们分别表示“匹配”和“不匹配”。它们的用法是在运算符左边指定变量，在右边指定规则表达式。如果某一行的第五个字段包含字符序列 root，那么以下示例将只打印这一行中的第三个字段：

$5 ~ /root/ { print $3 }

条件语句

awk 还提供了非常好的类似于 C 语言的 if 语句。如果您愿意，可以使用 if 语句重写前一个脚本：

{

if ( $5 ~ /root/ ) {

print $3

}

}

这两个脚本的功能完全一样。第一个示例中，布尔表达式放在代码块外面。而在第二个示例中，将对每一个输入行执行代码块，而且我们使用 if 语句来选择执行 print 命令。这两个方法都可以使用，可以选择最适合脚本其它部分的一种方法。

以下是更复杂的 awk if 语句示例。可以看到，尽管使用了复杂、嵌套的条件语句，if 语句看上去仍与相应的 C 语言 if 语句一样：

{

if ( $1 == "foo" ) {

if ( $2 == "foo" ) {

print "uno"

} else {

print "one"

}

} else if ($1 == "bar" ) {

print "two"

} else {

print "three"

}

}

使用 if 语句还可以将代码：

! /matchme/ { print $1 $3 $4 }

转换成：

{

if ( $0 !~ /matchme/ ) {

print $1 $3 $4

}

}

这两个脚本都只输出不包含 matchme 字符序列的那些行。此外，还可以选择最适合您的代码的方法。它们的功能完全相同。

awk 还允许使用布尔运算符 "||"（逻辑与）和 "&&"（逻辑或），以便创建更复杂的布尔表达式：

( $1 == "foo" ) && ( $2 == "bar" ) { print }

这个示例只打印第一个字段等于 foo 且第二个字段等于 bar 的那些行。

数值变量

至今，我们不是打印字符串、整行就是特定字段。然而，awk 还允许我们执行整数和浮点运算。通过使用数学表达式，可以很方便地编写计算文件中空白行数量的脚本。以下就是这样一个脚本：

BEGIN { x=0 }

/^$/ { x=x+1 }

END { print "I found " x " blank lines. :}" }

在 BEGIN 块中，将整数变量 x 初始化成零。然后，awk 每次遇到空白行时，awk 将执行 x=x+1 语句，递增 x。处理完所有行之后，执行 END 块，awk 将打印出最终摘要，指出它找到的空白行数量。

字符串化变量

awk 的优点之一就是“简单和字符串化”。我认为 awk 变量“字符串化”是因为所有 awk 变量在内部都是按字符串形式存储的。同时，awk 变量是“简单的”，因为可以对它执行数学操作，且只要变量包含有效数字字符串，awk 会自动处理字符串到数字的转换步骤。要理解我的观点，请研究以下这个示例：

x="1.01"

# We just set x to contain the \*string\* "1.01"

x=x+1

# We just added one to a \*string\*

print x

# Incidentally, these are comments :)

awk 将输出：

2.01

有趣吧！虽然将字符串值 1.01 赋值给变量 x，我们仍然可以对它加一。但在 bash 和 python 中却不能这样做。首先，bash 不支持浮点运算。而且，如果 bash 有“字符串化”变量，它们并不“简单”；要执行任何数学操作，bash 要求我们将数字放到丑陋的 $( ) ) 结构中。如果使用 python，则必须在对 1.01 字符串执行任何数学运算之前，将它转换成浮点值。虽然这并不困难，但它仍是附加的步骤。如果使用 awk，它是全自动的，而那会使我们的代码又好又整洁。如果想要对每个输入行的第一个字段乘方并加一，可以使用以下脚本：

{ print ($1^2)+1 }

如果做一个小实验，就可以发现如果某个特定变量不包含有效数字，awk 在对数学表达式求值时会将该变量当作数字零处理。

众多运算符

awk 的另一个优点是它有完整的数学运算符集合。除了标准的加、减、乘、除，awk 还允许使用前面演示过的指数运算符 "^"、模（余数）运算符 "%" 和其它许多从 C 语言中借入的易于使用的赋值操作符。

这些运算符包括前后加减（i++、--foo）、加／减／乘／除赋值运算符（ a+=3、b\*=2、c/=2.2、d-=6.2）。不仅如此 -- 我们还有易于使用的模／指数赋值运算符（a^=2、b%=4）。

字段分隔符

awk 有它自己的特殊变量集合。其中一些允许调整 awk 的运行方式，而其它变量可以被读取以收集关于输入的有用信息。我们已经接触过这些特殊变量中的一个，FS。前面已经提到过，这个变量让您可以设置 awk 要查找的字段之间的字符序列。我们使用 /etc/passwd 作为输入时，将 FS 设置成 ":"。当这样做有问题时，我们还可以更灵活地使用 FS。

FS 值并没有被限制为单一字符；可以通过指定任意长度的字符模式，将它设置成规则表达式。如果正在处理由一个或多个 tab 分隔的字段，您可能希望按以下方式设置 FS：

FS="\t+"

以上示例中，我们使用特殊 "+" 规则表达式字符，它表示“一个或多个前一字符”。

如果字段由空格分隔（一个或多个空格或 tab），您可能想要将 FS 设置成以下规则表达式：

FS="[[:space:]+]"

这个赋值表达式也有问题，它并非必要。为什么？因为缺省情况下，FS 设置成单一空格字符，awk 将这解释成表示“一个或多个空格或 tab”。在这个特殊示例中，缺省 FS 设置恰恰是您最想要的！

复杂的规则表达式也不成问题。即使您的记录由单词 "foo" 分隔，后面跟着三个数字，以下规则表达式仍允许对数据进行正确的分析：

FS="foo[0-9][0-9][0-9]"

字段数量

接着我们要讨论的两个变量通常并不是需要赋值的，而是用来读取以获取关于输入的有用信息。第一个是 NF 变量，也叫做“字段数量”变量。awk 会自动将该变量设置成当前记录中的字段数量。可以使用 NF 变量来只显示某些输入行：

NF == 3 { print "this particular record has three fields: " $0 }

当然，也可以在条件语句中使用 NF 变量，如下：

{

if ( NF > 2 ) {

print $1 " " $2 ":" $3

}

}

记录号

记录号 (NR) 是另一个方便的变量。它始终包含当前记录的编号（awk 将第一个记录算作记录号 1）。迄今为止，我们已经处理了每一行包含一个记录的输入文件。对于这些情况，NR 还会告诉您当前行号。然而，当我们在本系列以后部分中开始处理多行记录时，就不会再有这种情况，所以要注意！可以象使用 NF 变量一样使用 NR 来只打印某些输入行：

(NR < 10 ) || (NR > 100) { print "We are on record number 1-9 or 101+" }

另一个示例：

{

#skip header

if ( NR > 10 ) {

print "ok, now for the real information!"

}

}

awk 提供了适合各种用途的附加变量。我们将在以后的文章中讨论这些变量。

多行记录

awk 是一种用于读取和处理结构化数据（如系统的 /etc/passwd 文件）的极佳工具。/etc/passwd 是 UNIX 用户数据库，并且是用冒号定界的文本文件，它包含许多重要信息，包括所有现有用户帐户和用户标识，以及其它信息。在我的前一篇文章中，我演示了 awk 如何轻松地分析这个文件。我们只须将 FS（字段分隔符）变量设置成 ":"。

正确设置了 FS 变量之后，就可以将 awk 配置成分析几乎任何类型的结构化数据，只要这些数据是每行一个记录。然而，如果要分析占据多行的记录，仅仅依靠设置 FS 是不够的。在这些情况下，我们还需要修改 RS 记录分隔符变量。RS 变量告诉 awk 当前记录什么时候结束，新记录什么时候开始。

譬如，让我们讨论一下如何完成处理“联邦证人保护计划”所涉及人员的地址列表的任务：

Jimmy the Weasel

100 Pleasant Drive

San Francisco, CA 12345

Big Tony

200 Incognito Ave.

Suburbia, WA 67890

理论上，我们希望 awk 将每 3 行看作是一个独立的记录，而不是三个独立的记录。如果 awk 将地址的第一行看作是第一个字段 ($1)，街道地址看作是第二个字段 ($2)，城市、州和邮政编码看作是第三个字段 $3，那么这个代码就会变得很简单。以下就是我们想要得到的代码：

BEGIN {

FS="\n"

RS=""

}

在上面这段代码中，将 FS 设置成 "\n" 告诉 awk 每个字段都占据一行。通过将 RS 设置成 ""，还会告诉 awk 每个地址记录都由空白行分隔。一旦 awk 知道是如何格式化输入的，它就可以为我们执行所有分析工作，脚本的其余部分很简单。让我们研究一个完整的脚本，它将分析这个地址列表，并将每个记录打印在一行上，用逗号分隔每个字段。

address.awk BEGIN {

FS="\n"

RS=""

}

{

print $1 ", " $2 ", " $3

}

如果这个脚本保存为 address.awk，地址数据存储在文件 address.txt 中，可以通过输入 "awk -f address.awk address.txt" 来执行这个脚本。此代码将产生以下输出：

Jimmy the Weasel, 100 Pleasant Drive, San Francisco, CA 12345

Big Tony, 200 Incognito Ave., Suburbia, WA 67890

OFS 和 ORS

在 address.awk 的 print 语句中，可以看到 awk 会连接（合并）一行中彼此相邻的字符串。我们使用此功能在同一行上的三个字段之间插入一个逗号和空格 (", ")。这个方法虽然有用，但比较难看。与其在字段间插入 ", " 字符串，倒不如让通过设置一个特殊 awk 变量 OFS，让 awk 完成这件事。请参考下面这个代码片断。

print "Hello", "there", "Jim!"

这行代码中的逗号并不是实际文字字符串的一部分。事实上，它们告诉 awk "Hello"、"there" 和 "Jim!" 是单独的字段，并且应该在每个字符串之间打印 OFS 变量。缺省情况下，awk 产生以下输出：

Hello there Jim!

这是缺省情况下的输出结果，OFS 被设置成 " "，单个空格。不过，我们可以方便地重新定义 OFS，这样 awk 将插入我们中意的字段分隔符。以下是原始 address.awk 程序的修订版，它使用 OFS 来输出那些中间的 ", " 字符串：

address.awk 的修订版 BEGIN {

FS="\n"

RS=""

OFS=", "

}

{

print $1, $2, $3

}

awk 还有一个特殊变量 ORS，全称是“输出记录分隔符”。通过设置缺省为换行 ("\n") 的 OFS，我们可以控制在 print 语句结尾自动打印的字符。缺省 ORS 值会使 awk 在新行中输出每个新的 print 语句。如果想使输出的间隔翻倍，可以将 ORS 设置成 "\n\n"。或者，如果想要用单个空格分隔记录（而不换行），将 ORS 设置成 " "。

将多行转换成用 tab 分隔的格式

假设我们编写了一个脚本，它将地址列表转换成每个记录一行，且用 tab 定界的格式，以便导入电子表格。使用稍加修改的 address.awk 之后，就可以清楚地看到这个程序只适合于三行的地址。如果 awk 遇到以下地址，将丢掉第四行，并且不打印该行：

Cousin Vinnie

Vinnie's Auto Shop

300 City Alley

Sosueme, OR 76543

要处理这种情况，代码最好考虑每个字段的记录数量，并依次打印每个记录。现在，代码只打印地址的前三个字段。以下就是我们想要的一些代码：

适合具有任意多字段的地址的 address.awk 版本 BEGIN {

FS="\n"

RS=""

ORS=""

}

{

x=1

while ( x<NF ) {

print $x "\t"

x++

}

print $NF "\n"

}

首先，将字段分隔符 FS 设置成 "\n"，将记录分隔符 RS 设置成 ""，这样 awk 可以象以前一样正确分析多行地址。然后，将输出记录分隔符 ORS 设置成 ""，它将使 print 语句在每个调用结尾不输出新行。这意味着如果希望任何文本从新的一行开始，那么需要明确写入 print "\n"。

在主代码块中，创建了一个变量 x 来存储正在处理的当前字段的编号。起初，它被设置成 1。然后，我们使用 while 循环（一种 awk 循环结构，等同于 C 语言中的 while 循环），对于所有记录（最后一个记录除外）重复打印记录和 tab 字符。最后，打印最后一个记录和换行；此外，由于将 ORS 设置成 ""，print 将不输出换行。程序输出如下，这正是我们所期望的（不算漂亮，但用 tab 定界，以便于导入电子表格）：

Jimmy the Weasel 100 Pleasant Drive San Francisco, CA 12345

Big Tony 200 Incognito Ave. Suburbia, WA 67890

Cousin Vinnie Vinnie's Auto Shop 300 City Alley Sosueme, OR 76543

循环结构

我们已经看到了 awk 的 while 循环结构，它等同于相应的 C 语言 while 循环。awk 还有 "do...while" 循环，它在代码块结尾处对条件求值，而不象标准 while 循环那样在开始处求值。它类似于其它语言中的 "repeat...until" 循环。以下是一个示例：

do...while 示例 {

count=1

do {

print "I get printed at least once no matter what"

} while ( count != 1 )

}

与一般的 while 循环不同，由于在代码块之后对条件求值，"do...while" 循环永远都至少执行一次。换句话说，当第一次遇到普通 while 循环时，如果条件为假，将永远不执行该循环。

for 循环

awk 允许创建 for 循环，它就象 while 循环，也等同于 C 语言的 for 循环：

for ( initial assignment; comparison; increment ) {

code block

}

以下是一个简短示例：

for ( x = 1; x <= 4; x++ ) {

print "iteration",x

}

此段代码将打印：

iteration 1

iteration 2

iteration 3

iteration 4

break 和 continue

此外，如同 C 语言一样，awk 提供了 break 和 continue 语句。使用这些语句可以更好地控制 awk 的循环结构。以下是迫切需要 break 语句的代码片断：

while 死循环 while (1) {

print "forever and ever..."

}

因为 1 永远代表是真，这个 while 循环将永远运行下去。以下是一个只执行十次的循环：

break 语句示例 x=1

while(1) {

print "iteration",x

if ( x == 10 ) {

break

}

x++

}

这里，break 语句用于“逃出”最深层的循环。"break" 使循环立即终止，并继续执行循环代码块后面的语句。

continue 语句补充了 break，其作用如下：

x=1

while (1) {

if ( x == 4 ) {

x++

continue

}

print "iteration",x

if ( x > 20 ) {

break

}

x++

}

这段代码打印 "iteration 1" 到 "iteration 21"，"iteration 4" 除外。如果迭代等于 4，则增加 x 并调用 continue 语句，该语句立即使 awk 开始执行下一个循环迭代，而不执行代码块的其余部分。如同 break 一样，continue 语句适合各种 awk 迭代循环。在 for 循环主体中使用时，continue 将使循环控制变量自动增加。以下是一个等价循环：

for ( x=1; x<=21; x++ ) {

if ( x == 4 ) {

continue

}

print "iteration",x

}

在 while 循环中时，在调用 continue 之前没有必要增加 x，因为 for 循环会自动增加 x。

数组

如果您知道 awk 可以使用数组，您一定会感到高兴。然而，在 awk 中，数组下标通常从 1 开始，而不是 0：

myarray[1]="jim"

myarray[2]=456

awk 遇到第一个赋值语句时，它将创建 myarray，并将元素 myarray[1] 设置成 "jim"。执行了第二个赋值语句后，数组就有两个元素了。

数组迭代

定义之后，awk 有一个便利的机制来迭代数组元素，如下所示：

for ( x in myarray ) {

print myarray[x]

}

这段代码将打印数组 myarray 中的每一个元素。当对于 for 使用这种特殊的 "in" 形式时，awk 将 myarray 的每个现有下标依次赋值给 x（循环控制变量），每次赋值以后都循环一次循环代码。虽然这是一个非常方便的 awk 功能，但它有一个缺点 -- 当 awk 在数组下标之间轮转时，它不会依照任何特定的顺序。那就意味着我们不能知道以上代码的输出是：

jim

456

还是：

456

jim

套用 Forrest Gump 的话来说，迭代数组内容就像一盒巧克力 -- 您永远不知道将会得到什么。因此有必要使 awk 数组“字符串化”，我们现在就来研究这个问题。

数组下标字符串化

在我的前一篇文章中，我演示了 awk 实际上以字符串格式来存储数字值。虽然 awk 要执行必要的转换来完成这项工作，但它却可以使用某些看起来很奇怪的代码：

a="1"

b="2"

c=a+b+3

执行了这段代码后，c 等于 6。由于 awk 是“字符串化”的，添加字符串 "1" 和 "2" 在功能上并不比添加数字 1 和 2 难。这两种情况下，awk 都可以成功执行运算。awk 的“字符串化”性质非常可爱 -- 您可能想要知道如果使用数组的字符串下标会发生什么情况。例如，使用以下代码：

myarr["1"]="Mr. Whipple"

print myarr["1"]

可以预料，这段代码将打印 "Mr. Whipple"。但如果去掉第二个 "1" 下标中的引号，情况又会怎样呢？

myarr["1"]="Mr. Whipple"

print myarr[1]

猜想这个代码片断的结果比较难。awk 将 myarr["1"] 和 myarr[1] 看作数组的两个独立元素，还是它们是指同一个元素？答案是它们指的是同一个元素，awk 将打印 "Mr. Whipple"，如同第一个代码片断一样。虽然看上去可能有点怪，但 awk 在幕后却一直使用数组的字符串下标！

了解了这个奇怪的真相之后，我们中的一些人可能想要执行类似于以下的古怪代码：

myarr["name"]="Mr. Whipple"

print myarr["name"]

这段代码不仅不会产生错误，而且它的功能与前面的示例完全相同，也将打印 "Mr. Whipple"！可以看到，awk 并没有限制我们使用纯整数下标；如果我们愿意，可以使用字符串下标，而且不会产生任何问题。只要我们使用非整数数组下标，如 myarr["name"]，那么我们就在使用关联数组。从技术上讲，如果我们使用字符串下标，awk 的后台操作并没有什么不同（因为即便使用“整数”下标，awk 还是会将它看作是字符串）。但是，应该将它们称作关联数组 -- 它听起来很酷，而且会给您的上司留下印象。字符串化下标是我们的小秘密。;)

数组工具

谈到数组时，awk 给予我们许多灵活性。可以使用字符串下标，而且不需要连续的数字序列下标（例如，可以定义 myarr[1] 和 myarr[1000]，但不定义其它所有元素）。虽然这些都很有用，但在某些情况下，会产生混淆。幸好，awk 提供了一些实用功能有助于使数组变得更易于管理。

首先，可以删除数组元素。如果想要删除数组 fooarray 的元素 1，输入：

delete fooarray[1]

而且，如果想要查看是否存在某个特定数组元素，可以使用特殊的 "in" 布尔运算符，如下所示：

if ( 1 in fooarray ) {

print "Ayep! It's there."

} else {

print "Nope! Can't find it."

}

格式化输出

虽然大多数情况下 awk 的 print 语句可以完成任务，但有时我们还需要更多。在那些情况下，awk 提供了两个我们熟知的老朋友 printf() 和 sprintf()。是的，如同其它许多 awk 部件一样，这些函数等同于相应的 C 语言函数。printf() 会将格式化字符串打印到 stdout，而 sprintf() 则返回可以赋值给变量的格式化字符串。如果不熟悉 printf() 和 sprintf()，介绍 C 语言的文章可以让您迅速了解这两个基本打印函数。在 Linux 系统上，可以输入 "man 3 printf" 来查看 printf() 帮助页面。

以下是一些 awk sprintf() 和 printf() 的样本代码。可以看到，它们几乎与 C 语言完全相同。

x=1

b="foo"

printf("%s got a %d on the last test\n","Jim",83)

myout=("%s-%d",b,x)

print myout

此代码将打印：

Jim got a 83 on the last test

foo-1

字符串函数

awk 有许多字符串函数，这是件好事。在 awk 中，确实需要字符串函数，因为不能象在其它语言（如 C、C++ 和 Python）中那样将字符串看作是字符数组。例如，如果执行以下代码：

mystring="How are you doing today?"

print mystring[3]

将会接收到一个错误，如下所示：

awk: string.gawk:59: fatal: attempt to use scalar as array

噢，好吧。虽然不象 Python 的序列类型那样方便，但 awk 的字符串函数还是可以完成任务。让我们来看一下。

首先，有一个基本 length() 函数，它返回字符串的长度。以下是它的使用方法：

print length(mystring)

此代码将打印值：

24

好，继续。下一个字符串函数叫作 index，它将返回子字符串在另一个字符串中出现的位置，如果没有找到该字符串则返回 0。使用 mystring，可以按以下方法调用它：

print index(mystring,"you")

awk 会打印：

9

让我们继续讨论另外两个简单的函数，tolower() 和 toupper()。与您猜想的一样，这两个函数将返回字符串并且将所有字符分别转换成小写或大写。请注意，tolower() 和 toupper() 返回新的字符串，不会修改原来的字符串。这段代码：

print tolower(mystring)

print toupper(mystring)

print mystring

……将产生以下输出：

how are you doing today?

HOW ARE YOU DOING TODAY?

How are you doing today?

到现在为止一切不错，但我们究竟如何从字符串中选择子串，甚至单个字符？那就是使用 substr() 的原因。以下是 substr() 的调用方法：

mysub=substr(mystring,startpos,maxlen)

mystring 应该是要从中抽取子串的字符串变量或文字字符串。startpos 应该设置成起始字符位置，maxlen 应该包含要抽取的字符串的最大长度。请注意，我说的是最大长度；如果 length(mystring) 比 startpos+maxlen 短，那么得到的结果就会被截断。substr() 不会修改原始字符串，而是返回子串。以下是一个示例：

print substr(mystring,9,3)

awk 将打印：

you

如果您通常用于编程的语言使用数组下标访问部分字符串（以及不使用这种语言的人），请记住 substr() 是 awk 代替方法。需要使用它来抽取单个字符和子串；因为 awk 是基于字符串的语言，所以会经常用到它。

一些更耐人寻味的函数

首先是 match()。match() 与 index() 非常相似，它与 index() 的区别在于它并不搜索子串，它搜索的是规则表达式。match() 函数将返回匹配的起始位置，如果没有找到匹配，则返回 0。此外，match() 还将设置两个变量，叫作 RSTART 和 RLENGTH。RSTART 包含返回值（第一个匹配的位置），RLENGTH 指定它占据的字符跨度（如果没有找到匹配，则返回 -1）。通过使用 RSTART、RLENGTH、substr() 和一个小循环，可以轻松地迭代字符串中的每个匹配。以下是一个 match() 调用示例：

print match(mystring,/you/), RSTART, RLENGTH

awk 将打印：

9 9 3

字符串替换

现在，我们将研究两个字符串替换函数，sub() 和 gsub()。这些函数与目前已经讨论过的函数略有不同，因为它们确实修改原始字符串。以下是一个模板，显示了如何调用 sub()：

sub(regexp,replstring,mystring)

调用 sub() 时，它将在 mystring 中匹配 regexp 的第一个字符序列，并且用 replstring 替换该序列。sub() 和 gsub() 用相同的自变量；唯一的区别是 sub() 将替换第一个 regexp 匹配（如果有的话），gsub() 将执行全局替换，换出字符串中的所有匹配。以下是一个 sub() 和 gsub() 调用示例：

sub(/o/,"O",mystring)

print mystring

mystring="How are you doing today?"

gsub(/o/,"O",mystring)

print mystring

必须将 mystring 复位成其初始值，因为第一个 sub() 调用直接修改了 mystring。在执行时，此代码将使 awk 输出：

HOw are you doing today?

HOw are yOu dOing tOday?

当然，也可以是更复杂的规则表达式。我把测试一些复杂规则表达式的任务留给您来完成。

通过介绍函数 split()，我们来汇总一下已讨论过的函数。split() 的任务是“切开”字符串，并将各部分放到使用整数下标的数组中。以下是一个 split() 调用示例：

numelements=split("Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec",mymonths,",")

调用 split() 时，第一个自变量包含要切开文字字符串或字符串变量。在第二个自变量中，应该指定 split() 将填入片段部分的数组名称。在第三个元素中，指定用于切开字符串的分隔符。split() 返回时，它将返回分割的字符串元素的数量。split() 将每一个片段赋值给下标从 1 开始的数组，因此以下代码：

print mymonths[1],mymonths[numelements]

……将打印：

Jan Dec

特殊字符串形式

简短注释 -- 调用 length()、sub() 或 gsub() 时，可以去掉最后一个自变量，这样 awk 将对 $0（整个当前行）应用函数调用。要打印文件中每一行的长度，使用以下 awk 脚本：

{

print length()

}

sed使用规则

sed 是很有用（但常被遗忘）的 UNIX 流编辑器。sed是十分强大和小巧的文本流编辑器。使用sed 可以执行字符串替换、创建更大的 sed 脚本以及使用 sed 的附加、插入和更改行命令。在以批处理方式编辑文件或以有效方式创建 shell 脚本来修改现有文件方面，它是十分理想的工具。

sed 示例

sed 通过对输入数据执行任意数量用户指定的编辑操作（“命令”）来工作。sed 是基于行的，因此按顺序对每一行执行命令。然后，sed 将其结果写入标准输出 (stdout)，它不修改任何输入文件。

让我们看一些示例。头几个会有些奇怪，因为我要用它们演示 sed 如何工作，而不是执行任何有用的任务。然而，如果您是 sed 新手，那么理解它们是十分重要的。下面是第一个示例：

$ sed -e 'd' /etc/services

如果输入该命令，将得不到任何输出。那么，发生了什么？在该例中，用一个编辑命令 'd' 调用 sed。sed 打开 /etc/services 文件，将一行读入其模式缓冲区，执行编辑命令（“删除行”），然后打印模式缓冲区（缓冲区已为空）。然后，它对后面的每一行重复这些步骤。这不会产生输出，因为 "d" 命令除去了模式缓冲区中的每一行！

在该例中，还有几件事要注意。首先，根本没有修改 /etc/services。这还是因为 sed 只读取在命令行指定的文件，将其用作输入 -- 它不试图修改该文件。第二件要注意的事是 sed 是面向行的。'd' 命令不是简单地告诉 sed 一下子删除所有输入数据。相反，sed 逐行将 /etc/services 的每一行读入其称为模式缓冲区的内部缓冲区。一旦将一行读入模式缓冲区，它就执行 'd' 命令，然后打印模式缓冲区的内容（在本例中没有内容）。我将在后面为您演示如何使用地址范围来控制将命令应用到哪些行 -- 但是，如果不使用地址，命令将应用到所有行。第三件要注意的事是括起 'd' 命令的单引号的用法。养成使用单引号来括起 sed 命令的习惯是个好注意，这样可以禁用 shell 扩展。

另一个 sed 示例

下面是使用 sed 从输出流除去 /etc/services 文件第一行的示例：

$ sed -e '1d' /etc/services | more

如您所见，除了前面有 '1' 之外，该命令与第一个 'd' 命令十分类似。如果您猜到 '1' 指的是第一行，那您就猜对了。与第一个示例中只使用 'd' 不同的是，这一次使用的 'd' 前面有一个可选的数字地址。通过使用地址，可以告诉 sed 只对某一或某些特定行进行编辑。

地址范围

现在，让我们看一下如何指定地址范围。在本例中，sed 将删除输出的第 1 到 10 行：

$ sed -e '1,10d' /etc/services | more

当用逗号将两个地址分开时，sed 将把后面的命令应用到从第一个地址开始、到第二个地址结束的范围。在本例中，将 'd' 命令应用到第 1 到 10 行（包括这两行）。所有其它行都被忽略。

带规则表达式的地址

现在演示一个更有用的示例。假设要查看 /etc/services 文件的内容，但是对查看其中包括的注释部分不感兴趣。如您所知，可以通过以 '#' 字符开头的行在 /etc/services 文件中放置注释。为了避免注释，我们希望 sed 删除以 '#' 开始的行。以下是具体做法：

$ sed -e '/^#/d' /etc/services | more

试一下该例，看看发生了什么。您将注意到，sed 成功完成了预期任务。现在，让我们分析发生的情况：

要理解 '/^#/d' 命令，首先需要对其剖析。首先，让我们除去 'd' -- 这是我们前面所使用的同一个删除行命令。新增加的是 '/^#/' 部分，它是一种新的规则表达式地址。规则表达式地址总是由斜杠括起。它们指定一种 模式，紧跟在规则表达式地址之后的命令将仅适用于正好与该特定模式匹配的行。因此，'/^#/' 是一个规则表达式。（规则表达式的有关规定可以参见本文前面的内容）

例如：

$ sed -e '/regexp/d' /path/to/my/test/file | more

这将导致 sed 删除任何匹配的行。

对比如下的命令：

$ sed -n -e '/regexp/p' /path/to/my/test/file | more

请注意新的 '-n' 选项，该选项告诉 sed 除非明确要求打印模式空间，否则不这样做。您还会注意到，我们用 'p' 命令替换了 'd' 命令，如您所猜想的那样，这明确要求 sed 打印模式空间。就这样，将只打印匹配部分。

有关地址的更多内容

目前为止，我们已经看到了行地址、行范围地址和 regexp 地址。但是，还有更多的可能。我们可以指定两个用逗号分开的规则表达式，sed 将与所有从匹配第一个规则表达式的第一行开始，到匹配第二个规则表达式的行结束（包括该行）的所有行匹配。

例如，以下命令将打印从包含 "BEGIN" 的行开始，并且以包含 "END" 的行结束的文本块：

$ sed -n -e '/BEGIN/,/END/p' /my/test/file | more

如果没发现 "BEGIN"，那么将不打印数据。如果发现了 "BEGIN"，但是在这之后的所有行中都没发现 "END"，那么将打印所有后续行。发生这种情况是因为 sed 面向流的特性 -- 它不知道是否会出现 "END"。

C 源代码示例

如果只要打印 C 源文件中的 main() 函数，可输入：

$ sed -n -e '/main[[:space:]]\*(/,/^)/p' sourcefile.c | more

该命令有两个规则表达式 '/main[[:space:]]\*(/' 和 '/^}/'，以及一个命令 'p'。第一个规则表达式将与后面依次跟有任意数量的空格或制表键以及开始圆括号的字符串 "main" 匹配。这应该与一般 ANSI C main() 声明的开始匹配。

在这个特别的规则表达式中，出现了 '[[:space:]]' 字符类。这只是一个特殊的关键字，它告诉 sed 与 TAB 或空格匹配。如果愿意的话，可以不输入 '[[:space:]]'，而输入 '['，然后是空格字母，然后是 -V，然后再输入制表键字母和 ']' -- Control-V 告诉 bash 要插入“真正”的制表键，而不是执行命令扩展。使用 '[[:space:]]' 命令类（特别是在脚本中）会更清楚。

现在看一下第二个 regexp。'/^}' 将与任何出现在新行行首的 '}' 字符匹配。如果代码的格式很好，那么这将与 main() 函数的结束花括号匹配。如果格式不好，则不会正确匹配 -- 这是执行模式匹配任务的一件棘手之事。因为是处于 '-n' 安静方式，所以 'p' 命令还是完成其惯有任务，即明确告诉 sed 打印该行。试着对 C 源文件运行该命令 -- 它应该输出整个 main() { } 块，包括开始的 "main()" 和结束的 '}'。

替换

让我们看一下 sed 最有用的命令之一，替换命令。使用该命令，可以将特定字符串或匹配的规则表达式用另一个字符串替换。

下面是该命令最基本用法的示例：

$ sed -e 's/foo/bar/' myfile.txt

上面的命令将 myfile.txt 中每行第一次出现的 'foo'（如果有的话）用字符串 'bar' 替换，然后将该文件内容输出到标准输出。请注意，我说的是每行第一次出现，尽管这通常不是您想要的。在进行字符串替换时，通常想执行全局替换。也就是说，要替换每行中的所有出现，如下所示：

$ sed -e 's/foo/bar/g' myfile.txt

在最后一个斜杠之后附加的 'g' 选项告诉 sed 执行全局替换。

关于 's///' 替换命令，还有其它几件要了解的事。首先，它是一个命令，并且只是一个命令，在所有上例中都没有指定地址。这意味着，'s///' 还可以与地址一起使用来控制要将命令应用到哪些行，如下所示：

$ sed -e '1,10s/enchantment/entrapment/g' myfile2.txt

上例将导致用短语 'entrapment' 替换所有出现的短语 'enchantment'，但是只在第一到第十行（包括这两行）上这样做。

$ sed -e '/^$/,/^END/s/hills/mountains/g' myfile3.txt

该例将用 'mountains' 替换 'hills'，但是，只从空行开始，到以三个字符 'END' 开始的行结束（包括这两行）的文本块上这样做。

关于 's///' 命令的另一个妙处是 '/' 分隔符有许多替换选项。如果正在执行字符串替换，并且规则表达式或替换字符串中有许多斜杠，则可以通过在 's' 之后指定一个不同的字符来更改分隔符。例如，下例将把所有出现的 /usr/local 替换成 /usr：

$ sed -e 's:/usr/local:/usr:g' mylist.txt

在该例中，使用冒号作为分隔符。如果需要在规则表达式中指定分隔符字符，可以在它前面加入反斜杠。

规则表达式混乱

目前为止，我们只执行了简单的字符串替换。虽然这很方便，但是我们还可以匹配规则表达式。例如，以下 sed 命令将匹配从 '<' 开始、到 '>' 结束、并且在其中包含任意数量字符的短语。下例将删除该短语（用空字符串替换）：

$ sed -e 's/<.\*>//g' myfile.html

这是要从文件除去 HTML 标记的第一个很好的 sed 脚本尝试，但是由于规则表达式的特有规则，它不会很好地工作。原因何在？当 sed 试图在行中匹配规则表达式时，它要在行中查找最长的匹配。在我的前一篇 sed 文章中，这不成问题，因为我们使用的是 'd' 和 'p' 命令，这些命令总要删除或打印整行。但是，在使用 's///' 命令时，确实有很大不同，因为规则表达式匹配的整个部分将被目标字符串替换，或者，在本例中，被删除。这意味着，上例将把下行：

<b>This</b> is what <b>I</b> meant.

变成：meant.

我们要的不是这个，而是：This is what I meant.

幸运的是，有一种简便方法来纠正该问题。我们不输入“'<' 字符后面跟有一些字符并以 '>' 字符结束”的规则表达式，而只需输入一个“'<' 字符后面跟有任意数量非 '>' 字符并以 '>' 字符结束”的规则表达式。这将与最短、而不是最长的可能性匹配。新命令如下：

$ sed -e 's/<[^>]\*>//g' myfile.html

在上例中，'[^>]' 指定“非 '>'”字符，其后的 '\*' 完成该表达式以表示“零或多个非 '>' 字符”。对几个 html 文件测试该命令，将它们管道输出到 "more"，然后仔细查看其结果。

更多字符匹配

'[ ]' 规则表达式语法还有一些附加选项。要指定字符范围，只要字符不在第一个或最后一个位置，就可以使用 '-'，如下所示：

'[a-x]\*'

这将匹配零或多个全部为 'a'、'b'、'c'...'v'、'w'、'x' 的字符。另外，可以使用 '[:space:]' 字符类来匹配空格（字符类的相关信息可以参见本文前面部分内容）。

高级替换功能

我们已经看到如何执行简单甚至有些复杂的直接替换，但是 sed 还可以做更多的事。实际上可以引用匹配规则表达式的部分或全部，并使用这些部分来构造替换字符串。作为示例，假设您正在回复一条消息。下例将在每一行前面加上短语 "ralph said: "：

$ sed -e 's/.\*/ralph said: &/' origmsg.txt

输出如下：

ralph said: Hiya Jim, ralph said: ralph said:

I sure like this sed stuff! ralph said:

该例的替换字符串中使用了 '&' 字符，该字符告诉 sed 插入整个匹配的规则表达式。因此，可以将与 '.\*' 匹配的任何内容（行中的零或多个字符的最大组或整行）插入到替换字符串中的任何位置，甚至多次插入。这非常好，但 sed 甚至更强大。

那些极好的带反斜杠的圆括号

's///' 命令甚至比 '&' 更好，它允许我们在规则表达式中定义区域，然后可以在替换字符串中引用这些特定区域。作为示例，假设有一个包含以下文本的文件：

bar oni eeny meeny miny larry curly moe jimmy the weasel

现在假设要编写一个 sed 脚本，该脚本将把 "eeny meeny miny" 替换成 "Victor eeny-meeny Von miny" 等等。要这样做，首先要编写一个由空格分隔并与三个字符串匹配的规则表达式:

'.\* .\* .\*'

现在，将在其中每个感兴趣的区域两边插入带反斜杠的圆括号来定义区域：

'\(.\*\) \(.\*\) \(.\*\)'

除了要定义三个可在替换字符串中引用的逻辑区域以外，该规则表达式的工作原理将与第一个规则表达式相同。下面是最终脚本：

$ sed -e 's/\(.\*\) \(.\*\) \(.\*\)/Victor \1-\2 Von \3/' myfile.txt

如您所见，通过输入 '\x'（其中，x 是从 1 开始的区域号）来引用每个由圆括号定界的区域。输入如下：

Victor foo-bar Von oni Victor eeny-meeny Von miny Victor larry-curly Von moe Victor jimmy-the Von weasel

随着对 sed 越来越熟悉，您可以花最小力气来进行相当强大的文本处理。您可能想如何使用熟悉的脚本语言来处理这种问题 -- 能用一行代码轻易实现这样的解决方案吗？

组合使用

在开始创建更复杂的 sed 脚本时，需要有输入多个命令的能力。有几种方法这样做。首先，可以在命令之间使用分号。例如，以下命令系列使用 '=' 命令和 'p' 命令，'=' 命令告诉 sed 打印行号，'p' 命令明确告诉 sed 打印该行（因为处于 '-n' 模式）。

$ sed -n -e '=;p' myfile.txt

无论什么时候指定了两个或更多命令，都按顺序将每个命令应用到文件的每一行。在上例中，首先将 '=' 命令应用到第 1 行，然后应用 'p' 命令。接着，sed 继续处理第 2 行，并重复该过程。虽然分号很方便，但是在某些场合下，它不能正常工作。另一种替换方法是使用两个 -e 选项来指定两个不同的命令：

$ sed -n -e '=' -e 'p' myfile.txt

然而，在使用更为复杂的附加和插入命令时，甚至多个 '-e' 选项也不能帮我们的忙。对于复杂的多行脚本，最好的方法是将命令放入一个单独的文件中。然后，用 -f 选项引用该脚本文件：

$ sed -n -f mycommands.sed myfile.txt

这种方法虽然可能不太方便，但总是管用。

一个地址的多个命令

有时，可能要指定应用到一个地址的多个命令。这在执行许多 's///' 以变换源文件中的字和语法时特别方便。要对一个地址执行多个命令，可在文件中输入 sed 命令，然后使用 '{ }' 字符将这些命令分组，如下所示：

1,20{ s/[Ll]inux/GNU\/Linux/g s/samba/Samba/g s/posix/POSIX/g }

上例将把三个替换命令应用到第 1 行到第 20 行（包括这两行）。还可以使用规则表达式地址或者二者的组合：

1,/^END/{ s/[Ll]inux/GNU\/Linux/g s/samba/Samba/g s/posix/POSIX/g p }

该例将把 '{ }' 之间的所有命令应用到从第 1 行开始，到以字母 "END" 开始的行结束（如果在源文件中没发现 "END"，则到文件结束）的所有行。

附加、插入和更改行

既然在单独的文件中编写 sed 脚本，我们可以利用附加、插入和更改行命令。这些命令将在当前行之后插入一行，在当前行之前插入一行，或者替换模式空间中的当前行。它们也可以用来将多行插入到输出。插入行命令用法如下：

i\ This line will be inserted before each line

如果不为该命令指定地址，那么它将应用到每一行，并产生如下的输出：

This line will be inserted before each line line 1 here

This line will be inserted before each line line 2 here

This line will be inserted before each line line 3 here

This line will be inserted before each line line 4 here

如果要在当前行之前插入多行，可以通过在前一行之后附加一个反斜杠来添加附加行，如下所示：

i\ insert this line\ and this one\ and this one\ and, uh, this one too.

附加命令的用法与之类似，但是它将把一行或多行插入到模式空间中的当前行之后。其用法如下：

a\ insert this line after each line. Thanks! :)

另一方面，“更改行”命令将实际替换模式空间中的当前行，其用法如下：

c\ You're history, original line! Muhahaha!

因为附加、插入和更改行命令需要在多行输入，所以将把它们输入到一个文本 sed 脚本中，然后通过使用 '-f' 选项告诉 sed 执行它们。使用其它方法将命令传递给 sed 会出现问题。

使用 sed 的几个示例

这些示例不仅演示 sed 的能力，而且还做一些真正巧妙（和方便）的事。例如，在本文的后半部，将为您演示如何设计一个 sed 脚本来将 .QIF 文件从 Intuit 的 Quicken 金融程序转换成具有良好格式的文本文件。在那样做之前，我们将看一下不怎么复杂但却很有用的 sed 脚本。

l 文本转换

第一个实际脚本将 UNIX 风格的文本转换成 DOS/Windows 格式。您可能知道，基于 DOS/Windows 的文本文件在每一行末尾有一个 CR（回车）和 LF（换行），而 UNIX 文本只有一个换行。有时可能需要将某些 UNIX 文本移至 Windows 系统，该脚本将为您执行必需的格式转换。

$ sed -e 's/$/\r/' myunix.txt > mydos.txt

在该脚本中，'$' 规则表达式将与行的末尾匹配，而 '\r' 告诉 sed 在其之前插入一个回车。在换行之前插入回车，立即，每一行就以 CR/LF 结束。请注意，仅当使用 GNU sed 3.02.80 或以后的版本时，才会用 CR 替换 '\r'。如果还没有安装 GNU sed 3.02.80，请在我的第一篇 sed 文章中查看如何这样做的说明。

我已记不清有多少次在下载一些示例脚本或 C 代码之后，却发现它是 DOS/Windows 格式。虽然很多程序不在乎 DOS/Windows 格式的 CR/LF 文本文件，但是有几个程序却在乎 -- 最著名的是 bash，只要一遇到回车，它就会出问题。以下 sed 调用将把 DOS/Windows 格式的文本转换成可信赖的 UNIX 格式：

$ sed -e 's/.$//' mydos.txt > myunix.txt

该脚本的工作原理很简单：替代规则表达式与一行的最末字符匹配，而该字符恰好就是回车。我们用空字符替换它，从而将其从输出中彻底删除。如果使用该脚本并注意到已经删除了输出中每行的最末字符，那么，您就指定了已经是 UNIX 格式的文本文件。也就没必要那样做了！

l 反转行

下面是另一个方便的小脚本。与大多数 Linux 发行版中包括的 "tac" 命令一样，该脚本将反转文件中行的次序。"tac" 这个名称可能会给人以误导，因为 "tac" 不反转行中字符的位置（左和右），而是反转文件中行的位置（上和下）。用 "tac" 处理以下文件：

foo bar oni

....将产生以下输出：

oni bar foo

可以用以下 sed 脚本达到相同目的：

$ sed -e '1!G;h;$!d' forward.txt > backward.txt

如果登录到恰巧没有 "tac" 命令的 FreeBSD 系统，将发现该 sed 脚本很有用。虽然方便，但最好还是知道该脚本为什么那样做。让我们对它进行讨论。

反转解释：首先，该脚本包含三个由分号隔开的单独 sed 命令：'1!G'、'h' 和 '$!d'。现在，需要好好理解用于第一个和第三个命令的地址。如果第一个命令是 '1G'，则 'G' 命令将只应用第一行。然而，还有一个 '!' 字符 -- 该 '!' 字符忽略该地址，即，'G' 命令将应用到除第一行之外的所有行。'$!d' 命令与之类似。如果命令是 '$d'，则将只把 'd' 命令应用到文件中的最后一行（'$' 地址是指定最后一行的简单方式）。然而，有了 '!' 之后，'$!d' 将把 'd' 命令应用到除最后一行之外的所有行。现在，我们所要理解的是这些命令本身做什么。

当对上面的文本文件执行反转脚本时，首先执行的命令是 'h'。该命令告诉 sed 将模式空间（保存正在处理的当前行的缓冲区）的内容复制到保留空间（临时缓冲区）。然后，执行 'd' 命令，该命令从模式空间中删除 "foo"，以便在对这一行执行完所有命令之后不打印它。

现在，第二行。在将 "bar" 读入模式空间之后，执行 'G' 命令，该命令将保留空间的内容 ("foo\n") 附加到模式空间 ("bar\n")，使模式空间的内容为 "bar\n\foo\n"。'h' 命令将该内容放回保留空间保护起来，然后，'d' 从模式空间删除该行，以便不打印它。

对于最后的 "oni" 行，除了不删除模式空间的内容（由于 'd' 之前的 '$!'）以及将模式空间的内容（三行）打印到标准输出之外，重复同样的步骤。

linux常用脚本和函数

l #查找当前目录中是否存在指定目录，若不存在，则创建之

function mkdir\_1

{

if test ! -d $1

then

mkdir $1

fi

}

l #将指定文件中的"prefix = .\*"串替换为"prefix=\/home\/gnome-unicore-install2\/usr/"

#可以用来作为sed用法的参考

function modify\_prefix

{

chmod +w $1

cp $1 $1.bak

sed 's/prefix = .\*/prefix=\/home\/gnome-unicore-install2\/usr/g' $1.bak > $1

rm $1.bak

}

l #将指定文件中的"^LDFLAGS =.\*"串替换为"LDFLAGS = -rdynamic -lgdk\_pixbuf -lgtk -lgdk -lgmodule -lglib -ldl -lXext -lX11 -lm"

#change\_gnome-config FILENAME

function change\_gnome-config

{

cp $1 $1.bak

sed 's/^LDFLAGS =.\*/LDFLAGS = -rdynamic -lgdk\_pixbuf -lgtk -lgdk -lgmodule -lglib -ldl -lXext -lX11 -lm /g' $1.bak> $1

rm $1.bak

}

l #删除指定文件的含有指定字符的行

#格式：delete\_line filename "word\_contain"

function delete\_line

{

chmod +w $1

cp $1 $1.bak

cat $1.bak | grep -v -e "$2" >$1

}

l #用途：删除文件中包含line1或（和？）line2的行

#格式：delete\_line filename line1 line2

function delete\_line\_no

{

chmod +w $1

cp $1 $1.bak

sed $2,$3'd' $1.bak>$1

rm $1.bak

}

l #用途：在LINE\_NO指定的行插入字符串CONTENT

#可以用来作为sed用法的参考

#格式: add\_line FILENAME LINE\_NO CONTENT

function add\_line

{

chmod +w $1

cp $1 $1.bak

sed -e $2 'i\' "$3" '' $1.bak > $1

rm $1.bak

}

l #用途：检查含有"PC24"代码的程序并打印出来

#格式: check\_PC24 //after installation

function check\_PC24

{

echo "now comes the PC24 checking..."

. $COMMAND\_UNICORE/shell/shell\_PC24 >& /dev/null

if test -s $COMMAND\_UNICORE/PC24\_result

then :

echo "The following file contains PC24 problems: $COMMAND\_UNICORE/PC24\_result "

else

echo "No PC24 problem found"

fi

}

l #打印标题

displayheader() {

echo " \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

echo " \* IeeeCC754 testing tool \*"

echo " \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

echo " "

}

l #打印一个菜单的做法

displayplatformmenu() {

#clear the screen

clear

displayheader

echo " a) SunSparc "

echo " b) IntelPentium "

echo " c) AMD "

echo " d) Unicore32 "

echo " e) Unicore32(with FP2001) "

echo " "

echo -n " select a Platform > "

}

l #接收一个菜单输入

displayplatformmenu

read answer

case ${answer} in

a) TARGET="BasicOp";;

b) TARGET="Conversion";;

\*) badchoice;;

esac

l #查找当前目录下是否存在file\_name文件

#可以用来作为if用法的参考

detectfile\_name() {

if [ ! -f file\_name ]

then

echo "Error: file\_name does not exist. Please check"

exit 1;

else

echo "OK，the directy is exist"

fi

}

l #将参数指定的一个或多个目录项以及其下的多级子目录下的所有文件名和目录名转换为小写。

cvitem()

{

echo "mv $1 `dirname $1`/`basename $1 | tr \

'ABCDEFGHIJKLMNOPQRSTUVWXYZ' 'abcdefghijklmnopqrstuvwxyz'`"

}

[ $# = 0 ] && { echo "Usage: lcdir item1 item2 ..."; exit; }

for item in $\* #可以用来作为for用法的参考

do

[ "`dirname $item`" != "`basename $item`" ] && {

[ -d $item ] &&

{

for subitem in `ls $item`

do

cvlc $item/$subitem

done

}

cvitem $item

}

done

l #一个login的例子

if ($?path) then

set path=($HOME/bin $path)

else

set path=($HOME/bin /usr/bin .)

endif

if ( ! $ {?DT} ); then

stty dec new

tset -I -Q

endif

set mail=/usr/spool/mail/$USER

l #关于if使用的几个例子

n #执行一个命令或程序之前，先检查该命令是否存在，然後才执行

if [ -x /sbin/quotaon ] ; then

echo "Turning on Quota for root filesystem"

/sbin/quotaon /

fi

n #得到Hostname

#!/bin/sh

if [ -f /etc/HOSTNAME ] ; then

HOSTNAME=`cat /etc/HOSTNAME`

else

HOSTNAME=localhost

fi

n #如果某个设定档允许有好几个位置的话，例如crontab，可利用if then elif fi来找寻

if [ -f /etc/crontab ] ; then #[ -f /etc/crontab ]等价于test -f /etc/crontab

CRONTAB="/etc/crontab"

elif [ -f /var/spool/cron/crontabs/root ] ; then

CRONTAB="/var/spool/cron/crontabs/root"

elif [ -f /var/cron/tabs/root ] ; then

CRONTAB="/var/cron/tabs/root"

fi

export CRONTAB

n #利用uname来判断目前系统，并分别做各系统状况不同的事。

SYSTEM=`uname -s`

if [ $SYSTEM = "Linux" ] ; then

echo "Linux"

elif [ $SYSTEM = "FreeBSD" ] ; then

echo "FreeBSD"

elif [ $SYSTEM = "Solaris" ] ; then

echo "Solaris"

else

echo "What?"

fi

l #关于while使用的几个例子

n #无条件循环

while : ; do

echo "do something forever here"

sleep 5

done

linux常用命令

以下只说明各指令的基本用法, 若需详细说明, 请用 man 去读详细的 manual.

关于文件/目录处理的指令:

1. ls

这是最基本的文件指令。 ls 的意义为 "list"，也就是将某一个目录或是某一个文件的内容显示出来。

如果你在下 ls 指令后面没有跟任何的文件名，它将会显示出目前目录中所有文件。

也可以在 ls 后面加上所要察看的目录名称或文件的名称，如：

% ls /home2/X11R5

% ls first

ls 有一些特别的参数，可以给予使用者更多有关的信息，如下:

-a : 在 UNIX 中若一个目录或文件名字的第一个字符为 "." , 则使用 ls

将不会显示出这个文件的名字，我们称此类文件为隐藏文件。如果我们要察看这类文件，则必须加上参数 -a 。

-l : 这个参数代表使用 ls 的长( long )格式，可以显示更多的信息，如文件存取权，文件拥有者( owner )，文件大小，文件最后更新日期，甚而 symbolic link 的文件是 link 到那一个文件等等。例如：

% ls -l

drwx--x--x 2 jjtseng 512 Aug 8 05:08 18

drwx--x--x 2 jjtseng 512 Aug 8 22:00 19

-rw------- 1 jjtseng 566 Aug 8 05:28 makefile

2. cp

cp 这个指令的意义是复制("COPY") , 也就是将一个或多个文件复制成另一个文件或者是将其复制到另一个目录去。

cp 的用法如下:

cp f1 f2 : 将文件名为 f1 的文件复制一份为文件名为 f2 的文件。

cp f1 f2 f3 ... dir : 将文件 f1 f2 f3 ... 都以相同的文件名复制一份放到目录 dir 里面。

cp -r dir1 dir2 : 将 dir1 的全部内容全部复制到 dir2 里面。

cp 也有一些参数，如下:

-i : 此参数是当已经有文件名为 f2 的文件时，若迳自使用 cp 将会将原来 f2的内容覆盖，因此在要覆盖之前先询问使用者。如使用者的回答是y(yes)才执行复制的动作。

-r : 此参数是用来做递回复制用，可递归的将整个目录都复制到另一个目录中。

3. mv

mv 的意义为 move , 主要是将一文件改名或移动到另一个目录。与cp类似，它也有三种格式:

mv f1 f2 : 将文件名为 f1 的文件变更成文件名为 f2 的文件。

mv dir1 dir2 : 将文件名为 dir1 的目录变更成文件名为 dir2 的目录。

mv f1 f2 f3 ... dir : 将文件 f1 f2 f3 ... 都移至目录 dir 里面。

mv 的参数有两个，-f 和 -i , 其中 -i 的意义与 cp 中的相同，均是 interactive的意思。而 -f 为强迫( force ) , 就是不管有没有同名的文件，反正就是要执行，所有其他的参数遇到 -f 均会失效。

4. rm

rm 的意义是 remove ，也就是用来删除一个文件的指令。需要注意的是，在 UNIX 中一个被删除的文件除非是系统恰好做了备份，否则是无法像 DOS 里面一样还能够恢复的。所以在做 rm 动作的时候使用者应该要特别小心。

rm 的格式如下:

rm f1 f2 f3 .....

rm 的参数比较常用的有几个: -f , -i , 与 -r

-f : 将会使得系统在删除时，不提出任何警告讯息。

-i : 在删除文件之前均会询问是否真要删除。

-r : 递归的删除，可用于删除目录。

5. mkdir

mkdir 是一个让使用者建立一个目录的指令。你可以在一个目录底下使用 mkdir 建立一个子目录，使用的方法如下:

mkdir dirname1 [ dirname2 ... ]

8. pwd

pwd 会将当前目录的路径( path )显示出来

9. cat/more/less

以上三个指令均为察看文件内容的指令。cat 的意义是concatenate，实际就是把文件的内容显示出来的意思。

cat 有许多奇怪的参数，较常为人所使用的是 -n 参数，也就是把显示出来的内容加上行号。

cat 的用法如下:

cat [-n] :自标准输入读取内容，可以用 pipe 将别的指令的输出转向给 cat 。

cat [-n] filename : 将 filename 的内容读进来，显示在标准输出上。

问题在於 cat 它是不会停下来的，因此并不好用( 试想如果一个萤幕二十四行，而一个文件四百行，cat 一出来将会噼里啪啦不断的卷上去，使用者很难据此得到他们所需的信息。) 所以常常么使用 more 和less来帮助。

more 可以让文件根据控制台的模式一页页的显示出来，再根据使用者的要求换页或者换行。如果使用者要在某一个文件中搜寻一个特定的字符串，则按 / 然后跟着打所要搜寻的单字即可进行搜寻。

more 的使用法如下:

more filename

如果你在使用中觉得已经看到了所要看的部份，可以按'q'离开 more 的使用。

less 的用法与 more 极类似，原先它就是为了弥补 more 只能往前方卷页的缺点而设计。

Less 的用法如下:

less filename

它与 more 不同的是它可以按 y 来往上卷一行，并且可以用"?"来往回搜寻你所要找的字符。

10. chmod

chmod用来改变文件存取模式( change mode ) 。在linux中，一个文件有可读(r)可写(w)可执行(x)三种模式，分别针对该文件的拥有者( onwer )、同组用户( group member )(可以 ls -lg来观看某一文件的所属的 group )，以及其他人( other )。

一个文件如果改成可执行模式则系统就将其视为一个可执行文件，而一个目录的可执行模式代表使用者有进入该目录之权利。

chmod使用方式如下:

chmod [ -fR ] mode filename ...

其参数的意义如下:

-f Force. chmod 不会理会失败的动作。

-R Recurive. 会将目录下所有子目录及文件改为你所要改成的模式。

关于 Process 处理的指令:

1. ps

ps 是用来显示目前你的 process 或系统 processes 的状况。

其选项说明如下:

-a 列出包括其他 users 的 process 状况。

-u 显示 user - oriented 的 process 状况 。

-x 显示包括没有 terminal 控制的 process 状况 。

-w 使用较宽的显示模式来显示 process 状况 。

我们可以经由 ps 取得目前 processes 的状况，如 pid , running state 等。

2. kill

kill 指令的用途是送一个 signal 给某一个 process 。因为大部份送的都是用来杀掉 process 的 SIGKILL 或 SIGHUP，因此称为kill 。

kill 的用法为:

kill [ -SIGNAL ] pid ...

kill -l

SIGNAL 为一个 singal 的数字，从 0 到 31 ，其中 9 是 SIGKILL ，也就是一般用来杀掉一些无法正常 terminate 的信号。

也可以用 kill -l 来察看可代替 signal 号码的数字。

关于字符串处理的指令:

1. echo

echo 是用来显示一字符串在标准输出上。echo -n 则表示当显示完之后不执行换行操作。

2. grep

grep 为一过滤器，它可自一个或多个文件中过滤出具有某个字符串的行，或是从标准输入过滤出具有某个字符串的行。

grep的用法如下:

grep [-nv] match\_pattern file1 file2 ....

-n 把所找到的行在行前加上行号列出

-v 把不包含 match\_pattern 的行列出

match\_pattern 所要搜寻的字符串

-f 以 pattern\_file 存放所要搜寻的字符串

联机查询的指令:

1. man

man 是手册 ( manual ) 的意思。 UNIX 提供线上辅助( on-line help )的功能，man 就是用来让使用者在使用时查询指令、系统调用、标准库函数、各种表格等的使用所用的。

man 的用法如下:

man [-M path] [[section] title ] .....

man [-M path] -k keyword ...

-M path man 所需要的 manual database 的路径。我们也可以用设定环境变数 MANPATH 的方式来取代 -M 选项。

title 这是所要查询的目标。

section 用一个数字表示 manual 的分类，通常 1 代表可执行指令，2 代表系统调用( system call ) ，3 代表标准库函数，等等。

man 在 UNIX 上是一项非常重要的指令，我们在这里所描述的用法仅仅只是一个大家比较常用的用法以及简单的说明，真正详细的用法与说明还是要通过使用 man 来得到。

2. who

who 指令是用来查询目前有那些人在线上。

3. info

info的查询与man类似。

网络运用指令:

1. telnet

telnet 是一个提供 user 通过网络连到 remote host。

telnet 的 格式如下:

telnet [ hostname | ip-address ] [ port ]

hostname 为一个像 ccsun1 或是 ccsun1.cc.nctu.edu.tw 的 name address，ip-address 则为一个由四个小于 255 的数字组成的 ip address ，

2. ftp

ftp 的意义是 File Transfer Program ，是一个很常用的在网路文件传输的软件。

ftp 的格式如下:

ftp [ hostname | ip-address ]

其中 hostname | ip-address 的意义跟 telnet 中的相同。

在进入 ftp 之后，如果与 remote host 连接上了，它将会询问你 username 与密码，如果输入对了就可以开始进行文件传输。

在 ftp 中有许多的命令，这里仅列出较常用的 cd , lcd , mkdir , put , mput , get , mget , binary , ascii , prompt , help 与 quit 的使用方式。

ascii 将传输模式设为 ascii 模式。通常用於传送文字文件。

binary 将传输模式设为 binary 模式，通常用于传送执行文件，压缩文件与影像文件等。

cd remote-directory 将 remote host 上的工作目录改变。

lcd [ directory ] 更改 local host 的工作目录。

ls [ remote-directory ] [ local-file ] 列出 remote host 上的文件。

get remote-file [ local-file ] 取得登陆机的文件。

mget remote-files 可使用通用字符一次取得多个文件。

put local-file [ remote-file] 将 local host 的文件送到 remote host。

mput local-files 可使用通用字符一次将多个文件放到 remote host 上。

help [ command ] 线上辅助指令。

mkdir directory-name 在 remote host 新建一个目录。

prompt 更改交谈模式，若为 on 则在 mput 与 mget 时每做一个文件传输时均会询问。

Exit/quit离开ftp

3.rlogin命令

rlogin 是“remote login”（远程登录）的缩写。该命令与telnet命令很相似，允许用户启动远程系统上的交互命令会话。

rlogin 的一般格式是：

rlogin [ -8EKLdx ] [ -e char ] [-k realm ] [ - l username ] host

一般最常用的格式是：

rlogin host

该命令中各选项的含义为：

-8 此选项始终允许8位输入数据通道。该选项允许发送格式化的ANSI字符和其他的特殊代码。如果不用这个选项，除非远端的终止和启动字符不是或，否则就去掉奇偶校验位。

-E 停止把任何字符当作转义字符。当和-8选项一起使用时，它提供一个完全的透明连接。

-K 关闭所有的Kerberos确认。只有与使用Kerberos 确认协议的主机连接时才使用这个选项。

-L 允许rlogin会话在litout模式中运行。要了解更多信息，请查阅tty联机帮助。

-d 打开与远程主机进行通信的TCP sockets的socket调试。要了解更多信息，请查阅setsockopt的联机帮助。

-e 为rlogin会话设置转义字符，默认的转义字符是“~”，用户可以指定一个文字字符或一个\\nnn形式的八进制数。

-k 请求rlogin获得在指定区域内的远程主机的Kerberos许可，而不是获得由krb\_realmofhost（3）确定的远程主机区域内的远程主机的Kerberos 许可。

-x 为所有通过rlogin会话传送的数据打开DES加密。这会影响响应时间和CPU利用率，但是可以提高安全性。

4.rsh命令

rsh是“remote shell”（远程 shell）的缩写。 该命令在指定的远程主机上启动一个shell并执行用户在rsh命令行中指定的命令。如果用户没有给出要执行的命令，rsh就用rlogin命令使用户登录到远程机上。

rsh命令的一般格式是：

rsh [-Kdnx] [-k realm] [-l username] host [command]

一般常用的格式是：

rsh host [command ]

command可以是从shell提示符下键人的任何Linux命令。

rsh命令中各选项的含义如下：

-K 关闭所有的Kerbero确认。该选项只在与使用Kerbero确认的主机连接时才使用。

-d 打开与远程主机进行通信的TCP sockets的socket调试。要了解更多的信息，请查阅setsockopt的联机帮助。

-k 请求rsh获得在指定区域内的远程主机的Kerberos许可，而不是获得由krb\_relmofhost（3）确定的远程主机区域内的远程主机的Kerberos许可。

-l 缺省情况下，远程用户名与本地用户名相同。本选项允许指定远程用户名，如果指定了远程用户名，则使用Kerberos 确认，与在rlogin命令中一样。

-n 重定向来自特殊设备/dev/null的输入。

-x 为传送的所有数据打开DES加密。这会影响响应时间和CPU利用率，但是可以提高安全性。

Linux把标准输入放入rsh命令中,并把它拷贝到要远程执行的命令的标准输入中。它把远程命令的标准输出拷贝到rsh的标准输出中。它还把远程标准错误拷贝到本地标准错误文件中。任何退出、中止和中断信号都被送到远程命令中。当远程命令终止了，rsh也就终止了。

5.rcp命令

rcp代表“remote file copy”（远程文件拷贝）。该命令用于在计算机之间拷贝文件。

rcp命令有两种格式。第一种格式用于文件到文件的拷贝；第二种格式用于把文件或目录拷贝到另一个目录中。

rcp命令的一般格式是：

rcp [-px] [-k realm] file1 file2 rcp [-px] [-r] [-k realm] file

directory 每个文件或目录参数既可以是远程文件名也可以是本地文件名。远程文件名具有如下形式：rname@rhost：path，其中rname是远程用户名，rhost是远程计算机名，path是这个文件的路径。

rcp命令的各选项含义如下：

-r 递归地把源目录中的所有内容拷贝到目的目录中。要使用这个选项，目的必须是一个目录。

-p 试图保留源文件的修改时间和模式，忽略umask。

-k 请求rcp获得在指定区域内的远程主机的Kerberos 许可，而不是获得由krb\_relmofhost（3）确定的远程主机区域内的远程主机的Kerberos许可。

-x 为传送的所有数据打开DES加密。这会影响响应时间和CPU利用率，但是可以提高安全性。 如果在文件名中指定的路径不是完整的路径名，那么这个路径被解释为相对远程机上同名用户的主目录。如果没有给出远程用户名，就使用当前用户名。如果远程机上的路径包含特殊shell字符，需要用反斜线（\\）、双引号（”）或单引号（’）括起来，使所有的shell元字符都能被远程地解释。 需要说明的是，rcp不提示输入口令，它通过rsh命令来执行拷贝。

Vi常用技巧

l 取消命令

在vi中，只要没有把修改结果存入磁盘文件中，那么就可以通过“取消”来撤销最近的操作或对缓冲区的修改。

假设你无意删除了一行文本、改变了一些你不应该改变的内容或增加了一些不正确的文本，可以按<Esc>改变到命令模式中，然后按<u>，则文件内容恢复到修改前的样子。

l 保存到文件名为filename的文件中

发出写命令格式： :w filename

l 不使用小键盘来定位光标

vi用<h>、<j>、<k>、<l>键来定位光标。其中<h>、<l>键表示光标的左右移动，<j>、<k>键表示光标的上下移动，在某些没有或不能使用小键盘的情况下这四个键是很有用的。

下面是其他一些用于移动光标的键：

n 按空格键或<l>向右移动光标一个位置

n 按<Return>将光标移动到下一行的行首

n 使用<j>键将光标移动到下一行的当前位置或行末

n 按<->将光标移动到上一行行首

n 使用<k>键将光标移动到上一行的当前位置或行末

n 按<h>将光标向左移动一个字符

n 按<0>（零）将光标移动到一行的行首

n 按<$>将光标移动到一行的行末

l 大范围移动键

可快速定位光标到屏幕的顶部、中部和底部：

n 按<Shift-h>将光标移到屏幕的第一行，有时称为home位置

n 按<Shift-m>将光标移到现在屏幕显示的各行的中间一行

n 按<Shift-l>将光标移到屏幕的最后一行

n 按<Ctrl-f>向前移动一屏

n 按<Ctrl-b>向后移动一屏

n 要移动到缓冲区中指定的行中，在按<Shift-g>前键入行号（注意，这里的行号不是当前屏幕中的相对行号，而是绝对行号）

l 删除文本

n <x>删除光标处的字符

n <d> <w> 删除从当前字的光标处到下一个字的开始处之间的内容

n <d> <$> 删除从光标处到行尾之间的内容

n <Shift-d> 同<d> <$>，删除当前行的剩余部分

n <d> <d> 删除整行，不管光标在该行的位置

n 通过在上述命令之前键入一个整数，可将这些命令应用到几个对象中，例如：<4> <x>删除4个字符；<8> <d> <d> 删除8行

l 添加文本

n 使用<i>在光标位置前插入文本

n 使用<Shift-i>使你进入输入模式并且在当前行行首插入文本

n 使用<a>在光标位置后插入文本

n 使用<Shift-a>使你进入输入模式并且在当前行末尾插入文本

n 使用<o>在当前行的下面打开一行以添加文本

n 使用<Shift-o>在当前行的上面打开一行以添加文本

l 使vi显示行号

按<Esc>键以确保你在命令模式中，然后输入：se number。要关闭行号，输入：se nonumber

l 查找

n /string 在缓冲区中向前查找字符串string

n ?string 在缓冲区中向后查找字符串string

n <n> 以当前的方向再次查找

n <Shift-n>以相反的方向再次查找

n 注意，查找的字符串中若含有特殊字符的，要使用\来进行转意

l 修改和替换文本

n <r> 替换单个字符

n <Shift-r>替换一个字符序列

n <c> <w>修改当前字，从光标处到这个字的字尾

n <c> <e>修改当前字，从光标处到这个字的字尾（与<c> <w>相同）

n <c> <b>修改当前字，从该字的字头到光标以前的那些字符

n <c> <$>修改一行，从光标处到该行的行尾

n <Shift-c>修改一行，从光标处到该行的行尾（与<c> <$>相同）

n <c> <c>修改整行

n 注意，这些命令的每个命令都使之进入了输入模式。除使用<r>来替换单个字符外，必须按<Esc>键来完成所作的修改并返回命令模式

n 要修改几个字，在按<c> <w>之前使用一个整数

l 拷贝、剪切和粘贴

n <y> <w>拷贝从当前字的光标处到下一个字的开始处之间的内容

n <y> <$>拷贝从光标处到行尾之间的内容

n <Shift-y>拷贝当前行的剩余部分（与<y> <$>相同）

n <y> <y>拷贝整个当前行

n 通过在这些命令前键入整数，所有这些命令都可以用于多个对象。

n 当删除或剪切或拷贝时,删除或拷贝的对象被保存在通用缓冲区中,可以使用<p>或<Shift-p>命令将这个缓冲区中的内容粘贴到光标位置。

n <p>命令将对象粘贴到光标位置右边或光标位置后面

n <Shift-p>命令将对象粘贴到光标位置左边或光标位置前面

l 重复命令

可以按< . >来重复改变缓冲区的最后一个命令。