linux 系统下给命令指定别名 alias 命令用法:
在 linux 系统中如果命令太长又不符合用户的习惯，那么我们可以为它指定一个别名。虽然可以为命令建立“链接”解决长文件名的问题，但对于带命令行参数的命令，链接就无能为力了。而指定别名则可以解决此类所有问题。只要举一些例子就可以了：
alias l='ls -l' ;用 l 代替 ls -l 命令(Xenix 下就有类似的 l 命令)
alias cd..='cd ..' ;用 cd.. 代替 cd .. 命令(对在 DOS 下使用惯了 cd.. 的人帮助很大)
alias md='mkdir' ;用 md 代替 mkdir 命令(对在 DOS 下…)
alias c:='mount /dev/hda1 /mnt/c & cd /mnt/c' ;用 c: 命令代替命令序列：安装 DOS 分区，再进入。
通常我们可以将以上命令放到自己的 home 目录下的.bash_prifle 文件中,在使用 source .bash_profile 命令.即可使用.

Shell 编程基础

我们可以使用任意一种文字编辑器，比如 gedit、kedit、emacs、vi 等来编写 shell 脚本，它必须以如下行开始（必须放在文件的第一行）：

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

注意：最好使用“!/bin/bash”而不是“!/bin/sh”，如果使用 tc shell 改为 tcsh，其他类似。
符号#!用来告诉系统执行该脚本的程序，本例使用/bin/sh。编辑结束并保存后，如果要执行该脚本，必须先使其可执行：

```
chmod +x filename
```

此后在该脚本所在目录下，输入 ./filename 即可执行该脚本。

目录

变量赋值和引用

Shell 编程中，使用变量无需事先声明，同时变量名的命名须遵循如下规则：

- 首个字符必须为字母（a-z，A-Z）
- 中间不能有空格，可以使用下划线（_）
- 不能使用标点符号
- 不能使用 bash 里的关键字（可用 help 命令查看保留关键字）

需要给变量赋值时，可以这么写：

```
变量名=值
```

要取用一个变量的值，只需在变量名前面加一个$ (注意：给变量赋值的时候，不能在"="两边留空格)

```
#!/bin/sh
# 对变量赋值：
a="hello world" #等号两边均不能有空格存在
# 打印变量 a 的值：
echo "A is:" $a
```

挑个自己喜欢的编辑器，输入上述内容，并保存为文件 first，然后执行 chmod +x first 使其可执行，最后输入 ./first 执行该脚本。其输出结果如下：

```
A is: hello world
```

有时候变量名可能会和其它文字混淆，比如：

```
num=2
echo "this is the $numnd"
```

上述脚本并不会输出"this is the 2nd"而是"this is the "；这是由于 shell 会去搜索变量 numnd 的值，而实际上这个变量此时并没有值。这时，我们可以用花括号来告诉 shell 要打印的是 num 变量：

```
num=2
echo "this is the ${num}nd"
```

其输出结果为：this is the 2nd

需要注意 shell 的默认赋值是字符串赋值。比如：

```
var=1
var=$var+1
echo $var
```

打印出来的不是2而是1＋1。为了达到我们想要的效果有以下几种表达方式：

```
let "var+=1"
var=$[$var+1]
var=`expr $var + 1`#注意加号两边的空格，否则还是按照字符串的方式赋值。
```

注意：前两种方式在 bash 下有效，在 sh 下会出错。

let 表示数学运算，expr 用于整数值运算，每一项用空格隔开，$[]将中括号内的表达式作为数学运算先计算结果再输出。

Shell 脚本中有许多变量是系统自动设定的，我们将在用到这些变量时再作说明。除了只在脚本内有效的普通 shell 变量外，还有环境变量，即那些由 export 关键字处理过的变量。本文不讨论环境变量，因为它们一般只在登录脚本中用到。

Shell 里的流程控制

if 语 句

"if"表达式如果条件为真，则执行 then 后的部分:

```
if ....; then
  ....
elif ....; then
  ....
else
  ....
fi
```

大多数情况下，可以使用测试命令来对条件进行测试，比如可以比较字符串、判断文件是否存在及是否可读等等……通常用**" [ ] "来表示条件测试**，注意这里的空格很重要，要确保方括号前后的空格。

[-f "somefile"] ：判断是否是一个文件

[-x "/bin/ls"] ：判断/bin/ls 是否存在并有可执行权限

[-n "$var"] ：判断$var 变量是否有值

["$a" = "$b"] ：判断$a 和$b 是否相等

执行 man test 可以查看所有测试表达式可以比较和判断的类型。下面是一个简单的 if 语句：

```
#!/bin/sh

if [ ${SHELL} = "/bin/bash" ]; then
  echo "your login shell is the bash (bourne again shell)"
else
  echo "your login shell is not bash but ${SHELL}"
fi
```

变量$SHELL 包含有登录 shell 的名称，我们拿它和/bin/bash 进行比较以判断当前使用的 shell 是否为 bash。

还可以使用 test 选项 文件名 来测试，而测试结果使用 echo $?来查看

选项有： -d -f -w -r -x -L

数值测试的选项有： -eq = -ne -qt > -lt < -le <= -ge >=

&& 和 || 操作符

熟悉 C 语言的朋友可能会喜欢下面的表达式：

```
[ -f "/etc/shadow" ] && echo "This computer uses shadow passwords"
```

这 里的 && 就是一个快捷操作符，如果左边的表达式为真则执行右边的语句，你也可以把它看作逻辑运算里的与操作。上述脚本表示如果/etc/shadow 文件存在，则 打印"This computer uses shadow passwords"。同样 shell 编程中还可以用或操作(||)，例如：

```
#!/bin/sh

mailfolder=/var/spool/mail/james
[ -r "$mailfolder" ] || { echo "Can not read $mailfolder" ; exit 1; }
echo "$mailfolder has mail from:"
grep "^From " $mailfolder
```

该脚本首先判断 mailfolder 是否可读，如果可读则打印该文件中的"From" 一行。如果不可读则或操作生效，打印错误信息后脚本退出。需要注意的是，这里我们必须使用如下两个命令：

-打印错误信息

-退出程序

我们使用花括号以匿名函数的形式将两个命令放到一起作为一个命令使用；普通函数稍后再作说明。即使不用与和或操作符，我们也可以用 if 表达式完成任何事情，但是使用与或操作符会更便利很多。

case 语句

case 表达式可以用来匹配一个给定的字符串，而不是数字（可别和 C 语言里的 switch...case 混淆）。

```
case ... in
  ...) do something here
    ;;
 ...
    ;;
esac
```

file 命令可以辨别出一个给定文件的文件类型，如：file lf.gz，其输出结果为：

```
lf.gz: gzip compressed data, deflated, original filename,
last modified: Mon Aug 27 23:09:18 2001, os: Unix
```

我们利用这点写了一个名为 smartzip 的脚本，该脚本可以自动解压 bzip2, gzip 和 zip 类型的压缩文件：

```
#!/bin/sh

ftype=`file "$1"`   # Note ' and ` is different
case "$ftype" in
"$1: Zip archive"*)
  unzip "$1" ;;
"$1: gzip compressed"*)
  gunzip "$1" ;;
"$1: bzip2 compressed"*)
  bunzip2 "$1" ;;
*) echo "File $1 can not be uncompressed with smartzip";;
esac
```

你可能注意到上面使用了一个特殊变量$1，该变量包含有传递给该脚本的第一个参数值。也就是说，当我们运行：

```
smartzip articles.zip
```

$1 就是字符串 articles.zip。

select 语句

select 表达式是 bash 的一种扩展应用，擅长于交互式场合。用户可以从一组不同的值中进行选择：

```
select var in ... ; do
break;
done
.... now $var can be used ....
```

下面是一个简单的示例：

```
#!/bin/sh

echo "What is your favourite OS?"
select var in "Linux" "Gnu Hurd" "Free BSD" "Other"; do
  break;
done
echo "You have selected $var"
```

如果以上脚本运行出现 select ：NOT FOUND 将 #!/bin/sh 改为 #!/bin/bash 该脚本的运行结果如下：

```
What is your favourite OS?
1) Linux
2) Gnu Hurd
3) Free BSD
4) Other
#? 1
You have selected Linux
```

while/for 循环

在 shell 中，可以使用如下循环：

```
while ...; do
   ....
done
```

只要测试表达式条件为真，则 while 循环将一直运行。关键字"break"用来跳出循环，而关键字”continue”则可以跳过一个循环的余下部分，直接跳到下一次循环中。

for 循环会查看一个字符串行表（字符串用空格分隔），并将其赋给一个变量：

```
for var in ....; do
```

```
  ....
done
```

下面的示例会把 A B C 分别打印到屏幕上：

```
#!/bin/sh

for var in A B C ; do
  echo "var is $var"
done
```

下面是一个实用的脚本 showrpm，其功能是打印一些 RPM 包的统计信息：

```
#!/bin/sh

# list a content summary of a number of RPM packages
# USAGE: showrpm rpmfile1 rpmfile2 ...
# EXAMPLE: showrpm /cdrom/RedHat/RPMS/*.rpm
for rpmpackage in $*; do
  if [ -r "$rpmpackage" ];then
     echo "================ $rpmpackage ==============="
     rpm -qi -p $rpmpackage
  else
     echo "ERROR: cannot read file $rpmpackage"
  fi
done
```

这里出现了第二个特殊**变量$*，该变量包含有输入的所有命令行参数值**。如果你运行 showrpm openssh.rpm w3m.rpm webgrep.rpm，那么 $* 就包含有 3 个字符串，即 openssh.rpm, w3m.rpm 和 webgrep.rpm。

Shell 里的一些特殊符号

引号

在向程序传递任何参数之前，程序会扩展通配符和变量。这里所谓的扩展是指程序会把通配符（比如*）替换成适当的文件名，把变量替换成变量值。我们可以使用引号来防止这种扩展，先来看一个例子，假设在当前目录下有两个 jpg 文件：mail.jpg 和 tux.jpg。

```
#!/bin/sh

echo *.jpg
```

运行结果为：

```
mail.jpg tux.jpg
```

引号（单引号和双引号）可以防止通配符*的扩展：

```
#!/bin/sh

echo "*.jpg"
echo '*.jpg'
```

其运行结果为：

```
*.jpg
*.jpg
```

其中单引号更严格一些，它可以防止任何变量扩展；而双引号可以防止通配符扩展但允许变量扩展：

```
#!/bin/sh

echo $SHELL
echo "$SHELL"
echo '$SHELL'
```

运行结果为：

```
/bin/bash
/bin/bash
$SHELL
```

此外还有一种防止这种扩展的方法，即使用转义字符——反斜杆:\：

```
echo \*.jpg
echo \$SHELL
```

输出结果为：

```
*.jpg
$SHELL
```

Here Document

当 要将几行文字传递给一个命令时，用 here documents 是一种不错的方法。对每个脚本写一段帮助性的文字是很有用的，此时如果使用 here documents 就不必用 echo 函数一行行输出。Here document 以 << 开头，后面接上一个字符串，这个字符串还必须出现在 here document 的末尾。下面是一个例子，在该例子中，我们对多个文件进行重命名，并且使用 here documents 打印帮助：

```
#!/bin/sh

# we have less than 3 arguments. Print the help text:
if [ $# -lt 3 ] ; then
cat << HELP

ren -- renames a number of files using sed regular expressions USAGE: ren 'regexp'
'replacement' files...
```

```
EXAMPLE: rename all *.HTM files in *.html:
  ren 'HTM$' 'html' *.HTM

HELP
  exit 0
fi
OLD="$1"
NEW="$2"
# The shift command removes one argument from the list of
# command line arguments.
shift
shift
# $* contains now all the files:
for file in $*; do
  if [ -f "$file" ] ; then
      newfile=`echo "$file" | sed "s/${OLD}/${NEW}/g"`
      if [ -f "$newfile" ]; then
  echo "ERROR: $newfile exists already"
      else
          echo "renaming $file to $newfile ..."
          mv "$file" "$newfile"
      fi
  fi
done
```

这 个示例有点复杂，我们需要多花点时间来说明一番。第一个 if 表达式判断输入命令行参数是否小于3个(特殊变量$# 表示包含参数的个数) 。如果输入参数小于3个，则将帮助文字传递给 cat 命令，然后由 cat 命令将其打印在屏幕上。打印帮助文字后程序退出。如果输入参数等于或大于3个，我们 就将第一个参数赋值给变量 OLD，第二个参数赋值给变量 NEW。下一步，我们使用 shift 命令将第一个和第二个参数从参数列表中删除，这样原来的第三个 参数就成为参数列表$*的第一个参数。然后我们开始循环，命令行参数列表被一个接一个地被赋值给变量$file。接着我们判断该文件是否存在，如果存在则 通过 sed 命令搜索和替换来产生新的文件名。然后将反短斜线内命令结果赋值给 newfile。这样我们就达到了目的：得到了旧文件名和新文件名。然后使用 mv 命令进行重命名

Shell 里的函数

如果你写过比较复杂的脚本，就会发现可能在几个地方使用了相同的代码，这时如果用上函数，会方便很多。函数的大致样子如下：

```
functionname()
{
# inside the body $1 is the first argument given to the function
# $2 the second ...
body
```

```
}
```

你需要在每个脚本的开始对函数进行声明。

下面是一个名为 xtitlebar 的脚本，它可以改变终端窗口的名称。这里使用了一个名为 help 的函数，该函数在脚本中使用了两次：

```
#!/bin/sh
# vim: set sw=4 ts=4 et:
help()
{
cat << HELP
xtitlebar -- change the name of an xterm, gnome-terminal or kde konsole
USAGE: xtitlebar [-h] "string_for_titelbar"
OPTIONS: -h help text
EXAMPLE: xtitlebar "cvs"
HELP
exit 0
}
# in case of error or if -h is given we call the function help:
[ -z "$1" ] && help
[ "$1" = "-h" ] && help
# send the escape sequence to change the xterm titelbar:
echo -e "33]0;$107"
#
```

在脚本中提供帮助是一种很好的编程习惯，可以方便其他用户（和自己）使用和理解脚本。

== 命令行参数 == XXXXXXXXXXXXXXXXXXXXXXXXX

我 们已经见过$* 和 $1, $2 ... $9 等特殊变量，这些特殊变量包含了用户从命令行输入的参数。迄今为止，我们仅仅了解了一些简单的命令行语法（比如一些强制性的参数和查看帮助的-h 选项）。 但是在编写更复杂的程序时，您可能会发现您需要更多的自定义的选项。通常的惯例是在所有可选的参数之前加一个减号，后面再加上参数值 (比如文件名)。

有好多方法可以实现对输入参数的分析，但是下面的使用 case 表达式的例子无疑是一个不错的方法。

```
#!/bin/sh

help()
{
  cat << HELP
  This is a generic command line parser demo.
  USAGE EXAMPLE: cmdparser -l hello -f -- -somefile1 somefile2
HELP
  exit 0
}
```

```
while [ -n "$1" ]; do
case $1 in
  -h) help;shift 1;; # function help is called
  -f) opt_f=1;shift 1;; # variable opt_f is set
  -l) opt_l=$2;shift 2;; # -l takes an argument -> shift by 2
  --) shift;break;; # end of options
  -*) echo "error: no such option $1. -h for help";exit 1;;
  *) break;;
esac
done

echo "opt_f is $opt_f"
echo "opt_l is $opt_l"
echo "first arg is $1"
echo "2nd arg is $2"
```

你可以这样运行该脚本：

```
cmdparser -l hello -f -- -somefile1 somefile2
```

返回结果如下：

```
opt_f is 1
opt_l is hello
first arg is -somefile1
2nd arg is somefile2
```

这个脚本是如何工作的呢？脚本首先在所有输入命令行参数中进行循环，将输入参数与 case 表达式进行比较，如果匹配则设置一个变量并且移除该参数。根据 unix 系统的惯例，首先输入的应该是包含减号的参数。

Shell 脚本示例

=== 一般编程步骤=== xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
现在我们来讨论编写一个脚本的一般步骤。任何优秀的脚本都应该具有帮助和输入参数。写一个框架脚本（framework.sh），该脚本包含了大多数脚本需要的框架结构，是一个非常不错的主意。这样一来，当我们开始编写新脚本时，可以先执行如下命令：

```
cp framework.sh myscript
```

然后再插入自己的函数。
让我们来看看如下两个示例。

二进制到十进制的转换

脚本 b2d 将二进制数 (比如 1101) 转换为相应的十进制数。这也是一个用 expr 命令进行数学运算的例子：

```
#!/bin/sh
# vim: set sw=4 ts=4 et:
help()
{
  cat << HELP

b2d -- convert binary to decimal

USAGE: b2d [-h] binarynum

OPTIONS: -h help text

EXAMPLE: b2d 111010
will return 58
HELP
  exit 0
}

error()
{
  # print an error and exit
  echo "$1"
  exit 1
}

lastchar()
{
  # return the last character of a string in $rval
  if [ -z "$1" ]; then
      # empty string
      rval=""
      return
  fi
  # wc puts some space behind the output this is why we need sed:
  numofchar=`echo -n "$1" | wc -c | sed 's/ //g' `
  # now cut out the last char
  rval=`echo -n "$1" | cut -b $numofchar`
}

chop()
{
```

```
  # remove the last character in string and return it in $rval
  if [ -z "$1" ]; then
     # empty string
     rval=""
     return
  fi
  # wc puts some space behind the output this is why we need sed:
  numofchar=`echo -n "$1" | wc -c | sed 's/ //g' `
  if [ "$numofchar" = "1" ]; then
     # only one char in string
     rval=""
     return
  fi
  numofcharminus1=`expr $numofchar "-" 1`
  # now cut all but the last char:
  rval=`echo -n "$1" | cut -b -$numofcharminus1`
  #原来的 rval=`echo -n "$1" | cut -b 0-${numofcharminus1}`运行时出错.
  #原因是 cut 从1开始计数，应该是 cut -b 1-${numofcharminus1}
}

while [ -n "$1" ]; do
case $1 in
  -h) help;shift 1;; # function help is called
  --) shift;break;; # end of options
  -*) error "error: no such option $1. -h for help";;
  *) break;;
esac
done

# The main program
sum=0
weight=1
# one arg must be given:
[ -z "$1" ] && help
binnum="$1"
binnumorig="$1"

while [ -n "$binnum" ]; do
  lastchar "$binnum"
  if [ "$rval" = "1" ]; then
     sum=`expr "$weight" "+" "$sum"`
  fi
  # remove the last position in $binnum
  chop "$binnum"
```

```
  binnum="$rval"
  weight=`expr "$weight" "*" 2`
done

echo "binary $binnumorig is decimal $sum"
#
```

该脚本使用的算法是利用十进制和二进制数权值 (1,2,4,8,16,..)，比如二进制"10"可以这样转换成十进制：

0 * 1 + 1 * 2 = 2

为了得到单个的二进制数我们是用了 lastchar 函数。该函数使用 wc –c 计算字符个数，然后使用 cut 命令取出末尾一个字符。Chop 函数的功能则是移除最后一个字符。

文件循环拷贝

你 可能有这样的需求并一直都这么做：将所有发出邮件保存到一个文件中。但是过了几个月之后，这个文件可能会变得很大以至于该文件的访问速度变慢；下面的脚本 rotatefile 可以解决这个问题。这个脚本可以重命名邮件保存文件（假设为 outmail）为 outmail.1，而原来的 outmail.1就变成了 outmail.2 等等...

```
#!/bin/sh
# vim: set sw=4 ts=4 et:

ver="0.1"
help()
{
  cat << HELP
  rotatefile -- rotate the file name
  USAGE: rotatefile [-h] filename
  OPTIONS: -h help text
  EXAMPLE: rotatefile out

  This will e.g rename out.2 to out.3, out.1 to out.2, out to out.1[BR]
  and create an empty out-file

  The max number is 10

  version $ver
  HELP

  exit 0
}
```

```
error()
{
  echo "$1"
  exit 1
}

while [ -n "$1" ]; do
  case $1 in
      -h) help;shift 1;;
      --) break;;
      -*) echo "error: no such option $1. -h for help";exit 1;;
      *) break;;
  esac
done

# input check:
if [ -z "$1" ] ; then
  error "ERROR: you must specify a file, use -h for help"
fi

filen="$1"
# rename any .1 , .2 etc file:
for n in 9 8 7 6 5 4 3 2 1; do
  if [ -f "$filen.$n" ]; then
      p=`expr $n + 1`
      echo "mv $filen.$n $filen.$p"
      mv $filen.$n $filen.$p
  fi
done

# rename the original file:
if [ -f "$filen" ]; then
  echo "mv $filen $filen.1"
  mv $filen $filen.1
fi

echo touch $filen
touch $filen
```

这 个脚本是如何工作的呢？在检测到用户提供了一个文件名之后，首先进行一个9到1的循环；文件名.9重命名为文件名.10，文件名.8重命名为文件名. 9.....等等。循环结束之后，把原始文件命名为文件名.1，同时创建一个和原始文件同名的空文件（touch $filen）

脚本调试

最简单的调试方法当然是使用 echo 命令。你可以在任何怀疑出错的地方用 echo 打印变量值，这也是大部分 shell 程序员花费80%的时间用于调试的原因。Shell 脚本的好处在于无需重新编译，而插入一个 echo 命令也不需要多少时间。

shell 也有一个真正的调试模式，如果脚本"strangescript"出错，可以使用如下命令进行调试：

```
sh -x strangescript
```

上述命令会执行该脚本，同时显示所有变量的值。

shell 还有一个不执行脚本只检查语法的模式，命令如下：

```
sh -n your_script
```

这个命令会返回所有语法错误。

我们希望你现在已经可以开始编写自己的 shell 脚本了，尽情享受这份乐趣吧！ :)