SHELL十三问

2008125163340

SHELL十三问之五：var=value？export 前后差在哪？
文章整理: 文章来源: 网络
这次让我们暂时丢开command line，先来了解一下bash变量(variable)吧...
所谓的变量，就是就是利用一个特定的"名称"(name)来存取一段可以变化的"值"(value)。
设定(set)
在bash中，你可以用"="来设定或重新定义变量的内容：

name=value

在设定变量的时侯，得遵守如下规则：

*等号左右两边不能使用区隔符号(IFS)，也应避免使用shell的保留字符(meta charactor)。
*变量名称不能使用$符号。
*变量名称的第一个字母不能是数字(number)。
*变量名称长度不可超过256个字母。
*变量名称及变量值之大小写是有区别的(case sensitive)。

如下是一些变量设定时常见的错误：

A= B　　：不能有IFS
1A=B　　：不能以数字开头
$A=B　　：名称不能有$
a=B　　：这跟a=b是不同的

如下则是可以接受的设定：

A=" B"　　：IFS被关闭了(请参考前面的quoting章节)
A1=B　　：并非以数字开头
A=$B　　：$可用在变量值内
This_Is_A_Long_Name=b　　：可用_连接较长的名称或值，且大小写有别。

变量替换(substitution)
Shell之所以强大，其中的一个因素是它可以在命令行中对变量作替换(substitution)处理。
在命令行中使用者可以使用$符号加上变量名称(除了在用=号定义变量名称之外),
将变量值给替换出来，然后再重新组建命令行。
比方：

$ A=ls
$ B=la
$ C=/tmp
$ $A -$B $C

(注意：以上命令行的第一个$是shell prompt，并不在命令行之内。)
必需强调的是，我们所提的变量替换，只发生在command line上面。(是的，让我们再回到command line吧　u)
仔细分析最后那行command line，不难发现在被执行之前(在输入CR字符之前),
$符号会对每一个变量作替换处理(将变量值替换出来再重组命令行)，最后会得出如下命令行：

ls -la /tmp

还记得第二章我请大家"务必理解"的那两句吗？若你忘了，那我这里再重贴一遍：
若从技术细节来看，shell会依据IFS(Internal Field Seperator)将command line所输入的文字给拆解为"字段"(word)。
然后再针对特殊字符(meta)先作处理，最后再重组整行command line。
这里的$就是command line中最经典的meta之一了，就是作变量替换的　u
在日常的shell操作中，我们常会使用echo命令来查看特定变量的值，例如：

$ echo $A -$B $C

我们已学过，echo命令只单纯将其argument送至"标准输出"(STDOUT，通常是我们的荧幕)。
所以上面的命令会在荧幕上得到如下结果：

ls -la /tmp

这是由于echo命令在执行时，会先将$A(ls)、$B(la)、跟$C(/tmp)给替换出来的结果。
利用shell对变量的替换处理能力，我们在设定变量时就更为灵活了：

A=B
B=$A

这样，B的变量值就可继承A变量"当时"的变量值了。
不过，不要以"数学罗辑"来套用变量的设定，比方说：

A=B
B=C

这样并不会让A的变量值变成C。
上面是单纯定义了两个不同名称的变量：A与 B，它们的值分别是B与 C。再如：

A=B
B=$A
A=C

同样也不会让B的值换成C。

若变量被重复定义的话，则原有旧值将被新值所取代。(这不正是"可变的量"吗？ ^_^)
当我们在设定变量的时侯，请记着这点：

*用一个名称储存一个数值

仅此而已。
此外，我们也可利用命令行的变量替换能力来"扩充"(append)变量值：

A=B:C:D
A=$A:E

这样，第一行我们设定A的值为"B:C:D"，然后，第二行再将值扩充为"A:B:C:E"。
上面的扩充范例，我们使用区隔符号(:)来达到扩充目的，
要是没有区隔符号的话，如下是有问题的：

A=BCD
A=$AE

因为第二次是将A的值继承$AE的提换结果，而非$A再加E u
要解决此问题，我们可用更严谨的替换处理：

A=BCD
A=${A}E

上例中，我们使用{}将变量名称的范围给明确定义出来，
如此一来，我们就可以将A的变量值从BCD给扩充为BCDE。
(提示：关于${name}事实上还可做到更多的变量处理能力，这些均属于比较进阶的变量处理，现阶段暂时不介绍了，请大家自行参考数据。如CU的贴子：
http://www.chinaunix.net/forum/viewtopic.php?t=201843)
* export *
严格来说，我们在当前shell中所定义的变量，均属于"本地变量"(local variable)，
只有经过export命令的"输出"处理，才能成为环境变量(environment variable)：

$ A=B
$ export A

或：

$ export A=B

经过export输出处理之后，变量A就能成为一个环境变量供其后的命令使用。
在使用export 的时侯，请别忘记shell在命令行对变量的"替换"(substitution)处理，
比方说：

$ A=B

```
$ B=C
$ export $A
```

上面的命令并未将A输出为环境变量，而是将B作输出，
这是因为在这个命令行中，$A会首先被提换出B然后再"塞回"作export的参数。
要理解这个export，事实上需要从process的角度来理解才能透彻。
我将于下一章为大家说明process的观念，敬请留意。
取消变量
要取消一个变量，在bash中可使用unset命令来处理：

```
unset A
```

与export一样，unset命令行也同样会作变量替换(这其实就是shell的功能之一),
因此：

```
$ A=B
$ B=C
$ unset $A
```

事实上所取消的变量是B而不是A。
此外，变量一旦经过unset取消之后，其结果是将整个变量拿掉，而不仅是取消其变量值。
如下两行其实是很不一样的：

```
$ A=
$ unset A
```

第一行只是将变量A设定为"空值"(null value)，但第二行则让变量A不在存在。
虽然用眼睛来看，这两种变量状态在如下命令结果中都是一样的：

```
$ A=
$ echo $A
$ unset A
$ echo $A
```

请学员务必能识别null value与unset的本质区别，这在一些进阶的变量处理上是很严格的。
比方说：

```
$ str=              #设为null
$ var=${str=expr}       #定义var
$ echo $var

$ echo $str

$ unset str       #取消
$ var=${str=expr}       #定义var
$ echo $var
expr
$ echo $str
expr
```

聪明的读者(yes, you!)，稍加思考的话，
应该不难发现为何同样的var=${str=expr}在null与unset之下的不同吧？
若你看不出来，那可能是如下原因之一：
a.你太笨了
b.不了解 var=${str=expr}　　这个进阶处理
c.对本篇说明还没来得及消化吸收
e.我讲得不好
不知，你选哪个呢？.... ^_^

再来解释一下var=${str=expr}：
首先，var=$str这个大家都可理解吧。
而接下来的思考方向是，究竟$str这个变量是如下哪一种情况呢：
1) unset
2) null
3) not null
1)假如是unset，那么var=${str=expr}的结果将是：
var=expr
str=expr
2)假如是null，那var=${str=expr}的结果是：
var=
str=
3)假如是not null (比方为xyz)，那var=${str=expr}之结果是：
var=xyz
str=xyz
测试如下：

```
$ showvar() {
>    var=${str=expr}
>    echo \$var is $var
>    echo \$str is $str
> }
$ unset str
$ showvar
$var is expr
$str is expr
$ str=
$ showvar
$var is
$str is
$ str=xyz
$ showvar
$var is xyz
$str is xyz
```

接下来，再来看看var=${str:=expr}好了：
1) $str为not set：
var=expr
str=expr
2) $str为null：
var=expr
str=expr
3) $str为not null (str=xyz)：
var=xyz
str=xyz
测试如下：

```
$ showvar() {
> var=${str:=expr}
> echo \$var is $var
```

```
> echo \$str is $str
> }
$ unset str
$ showvar
$var is expr
$str is expr
$ str=
$ showvar
$var is expr
$str is expr
$ str=xyz
$ showvar
$var is xyz
$str is xyz
```

最后比教一下${str=expr}与${str:=expr}：

*两者在not set与not null都一致

*但当null值时，${str=expr}会将$var与$str都设为null，但${str:=expr}则设为expr

从这个再延伸出其它模拟，不防请大家"实作"观查一下有何不同？

var=${str-expr} vs var=${str:-expr}

var=${str+expr} vs var=${str:+expr}

var=${str?expr} vs var=${str:?expr}

```
$ showvar() {
>    var=${str-expr}
>    echo \$var is $var
>    echo \$str is $str
> }
$ unset str
$ showvar
$var is expr
$str is
$ str=
$ showvar
$var is
$str is
$ str=xyz
$ showvar
$var is xyz
$str is xyz

$ showvar() {
>    var=${str:-expr}
>    echo \$var is $var
>    echo \$str is $str
> }
$ unset str
$ showvar
```

```
$var is expr
$str is
$ str=
$ showvar
$var is expr
$str is
$ str=xyz
$ showvar
$var is xyz
$str is xyz
```

可以看出来${str-expr}与${str:-expr}
*两者在$str为not set与not null都一致
*但当null值时，${str-expr}会将$var与$str都设为null，但${str:-expr}则将$var设为expr

```
$ showvar() {
>    var=${str+expr}
>    echo \$var is $var
>    echo \$str is $str
> }
$ unset str
$ showvar
$var is
$str is
$ str=
$ showvar
$var is expr
$str is
$ str=xyz
$ showvar
$var is expr
$str is xyz
$ showvar() {
>    var=${str:+expr}
>    echo \$var is $var
>    echo \$str is $str
> }
$ unset str
$ showvar
$var is
$str is
$ str=
$ showvar
$var is
$str is
$ str=xyz
$ showvar
$var is expr
```

$str is xyz
可以看出来${str+expr}与${str:+expr}
*两者在$str为not set与not null都一致
*但当null值时，${str+expr}会将$var与$str都设为expr，但${str:+expr}则将$var设为null

```
$ showvar() {
>    var=${str?expr}
>    echo \$var is $var
>    echo \$str is $str
> }
$ unset str
$ showvar
expr
$ str=
$ showvar
$var is
$str is
$ str=xyz
$ showvar
$var is xyz
$str is xyz
$ showvar() {
>    var=${str:?expr}
>    echo \$var is $var
>    echo \$str is $str
> }
$ unset str
$ showvar
expr
$ str=
$ showvar
expr
$ str=xyz
$ showvar
$var is xyz
$str is xyz
```

可以看出来${str?expr}与${str:?expr}
*两者在$str为not set与not null都一致
*但当null值时，${str?expr}会将$var与$str都设为null，但${str:?expr}则将$var设为expr
综上所述：
$var=${str=expr}与$var=${str:=expr}
$var=${str-expr}与$var=${str:-expr}
$var=${str+expr}与$var=${str:+expr}
$var=${str?expr}与$var=${str:?expr}
* 两者在 $str 为not set 与 not null 都一致
* 但当 null 值时不一致
分别描述如下：
$var=${str:-expr}的功能：

SHELL十三问

当str非空的时候var赋值为str，否则（包括not set和null）将var赋值为expr
$var=${str-expr}的功能：
当str非空的时候var赋值为str，为空的时候var设置为null，not set的时候将var赋值为expr

$var=${str:=expr}的功能：
当str非空的时候var赋值为str，否则（包括not set和null）将var和str均赋值为expr
$var=${str=expr}的功能：
当str非空的时候var赋值为str，为空的时候var设置为null，not set的时候将var和str均赋值为expr

$var=${str:?expr}的功能：
当str非空的时候var赋值为expr，否则将expr写入标准错误后退出
$var=${str?expr}的功能：
当str非空的时候var赋值为expr，为空的时候var设置为null，not set的时候将expr写入标准错误后退出

$var=${str:+expr}的功能：
当str非空的时候var赋值为expr，否则将不做任何操作
$var=${str+expr}的功能：
当str非空的时候var赋值为expr，为空的时候var设置为expr，not set的时候不做任何操作

看来还是带有冒号的简单，只有两种情况：非空和其他，呵呵
[]

SHELL十三问之六：exec 跟 source 差在哪？
文章整理: 文章来源: 网络
这次先让我们从CU Shell版的一个实例贴子来谈起吧：
例中的提问是：
cd /etc/aa/bb/cc可以执行，但是把这条命令写入shell时shell不执行！
这是什么原因呀！
我当时如何回答暂时别去深究，先让我们了解一下进程(process)的观念好了。
首先，我们所执行的任何程序，都是由父进程(parent process)所产生出来的一个子进程(child process)，子进程在结束后，将返回到父进程去。此一现像在Linux系统中被称为 fork。
当子进程被产生的时候，将会从父进程那里获得一定的资源分配、及(更重要的是)继承父进程的环境　u
让我们回到上一章所谈到的"环境变量"吧：
*所谓环境变量其实就是那些会传给子进程的变量。
简单而言，"遗传性"就是区分本地变量与环境变量的决定性指标。
然而，从遗传的角度来看，我们也不难发现环境变量的另一个重要特征：
*环境变量只能从父进程到子进程单向继承。换句话说：在子进程中的环境如何变更，均不会影响父进程的环境。
接下来，再让我们了解一下命令脚本(shell script)的概念。
所谓的shell script讲起来很简单，就是将你平时在shell prompt后所输入的多行command line依序写入一个文件去而已。
其中再加上一些条件判断、互动界面、参数运用、函数调用、等等技巧，得以让script更加"聪明"的执行，但若撇开这些技巧不谈，我们真的可以简单的看成script只不过依次执行预先写好的命令行而已。
再结合以上两个概念(process + script)，那应该就不难理解如下这句话的意思了：
*正常来说，当我们执行一个shell script时，其实是先产生一个sub-shell的子进程，然后sub-shell再去产生命令行的子进程。
然则，那让我们回到本章开始时所提到的例子再从新思考：
cd /etc/aa/bb/cc可以执行，但是把这条命令写入shell时shell不执行！
这是什么原因呀！
我当时的答案是这样的：
因为，一般我们跑的shell script是用subshell去执行的。

从process的观念来看，是parent process产生一个child process去执行，当child结束后，会返回parent，但parent的环境是不会因child的改变而改变的。
所谓的环境元数很多，凡举effective id, variable, workding dir等等...
其中的workding dir ($PWD)正是楼主的疑问所在：
当用subshell来跑script的话，sub shell的$PWD会因为cd而变更，
但当返回primary shell时，$PWD是不会变更的。
能够了解问题的原因及其原理是很好的，但是？如何解决问题恐怕是我们更感兴趣的　u是吧？^_^
那好，接下来，再让我们了解一下source命令好了。
当你有了fork的概念之后，要理解source就不难：
*所谓source就是让script在当前shell内执行、而不是产生一个sub-shell来执行。
由于所有执行结果均于当前shell内完成，若script的环境有所改变，当然也会改变当前环境了　u
因此，只要我们要将原本单独输入的script命令行变成source命令的参数，就可轻易解决前例提到的问题了。
比方说，原本我们是如此执行 script的：
./my.script
现在改成这样即可：
source ./my.script
或：
. ./my.script
说到这里，我想，各位有兴趣看看/etc底下的众多设定文件，应该不难理解它们被定义后，如何让其它script读取并继承了吧？
若然，日后你有机会写自己的script，应也不难专门指定一个设定文件以供不同的script一起"共享"了... ^_^
okay，到这里，若你搞得懂fork与source的不同，那接下来再接受一个挑战：
----那exec又与source/fork有何不同呢？
哦...要了解exec或许较为复杂，尤其扯上File Descriptor的话...
不过，简单来说：
* exec也是让script在同一个进程上执行，但是原有进程则被结束了。
也就是简而言之：原有进程会否终止，就是exec与source/fork的最大差异了。
嗯，光是从理论去理解，或许没那么好消化，不如动手"实作+思考"来的印像深刻哦。
下面让我们写两个简单的script，分别命令为1.sh及2.sh：
1.sh

```
#!/bin/sh
A=B
echo "PID for 1.sh before exec/source/fork:$$"
export A
echo "1.sh: \$A is $A"
case $1 in
        exec)
                echo "using exec..."
                exec ./2.sh ;;
        source)
                echo "using source..."
                . ./2.sh ;;
        *)
                echo "using fork by default..."
                ./2.sh ;;
esac
echo "PID for 1.sh after exec/source/fork:$$"
echo "1.sh: \$A is $A"
```

2.sh
#!/bin/sh
echo "PID for 2.sh: $$"
echo "2.sh get \$A=$A from 1.sh"
A=C
export A
echo "2.sh: \$A is $A"
然后，分别跑如下参数来观察结果：
$ ./1.sh fork
PID for 1.sh before exec/source/fork:531
1.sh: $A is B
using fork by default...
PID for 2.sh:532
2.sh get $A=B from 1.sh
2.sh: $A is C
PID for 1.sh after exec/source/fork:531
1.sh: $A is B

$ ./1.sh source
PID for 1.sh before exec/source/fork:533
1.sh: $A is B
using source...
PID for 2.sh:533
2.sh get $A=B from 1.sh
2.sh: $A is C
PID for 1.sh after exec/source/fork:533
1.sh: $A is C

$ ./1.sh exec
PID for 1.sh before exec/source/fork:537
1.sh: $A is B
using exec...
PID for 2.sh:537
2.sh get $A=B from 1.sh
2.sh: $A is C
############
echo "PID for 1.sh after exec/source/fork:$$"
echo "1.sh: \$A is $A"
已经不会执行了，1.sh的进程已经没了。
##############
[]

SHELL十三问之七：( ) 与 { } 差在哪？
文章整理: 文章来源: 网络
先说一下，为何要用( )或{ }好了。
许多时候，我们在shell操作上，需要在一定条件下一次执行多个命令，也就是说，要么不执行，要么就全执行，而不是每次依序的判断是否要执行下一个命令。或是，需要从一些命令执行优先次顺中得到豁免，如算术的2*(3+4)那样...
这时候，我们就可引入"命令群组"(command group)的概念：将多个命令集中处理。

SHELL十三问

在shell command line中，一般人或许不太计较( )与{ }这两对符号的差异，虽然两者都可将多个命令作群组化处理，但若从技术细节上，却是很不一样的：
( )将command group置于sub-shell去执行，也称nested sub-shell。
{ }则是在同一个shell内完成，也称为non-named command group。
如果你对上一章的fork与source的概念还记得的话，那就不难理解两者的差异了。
要是在command group中扯上变量及其它环境的修改，我们可以根据不同的需求来使用( )或{ }。通常而言，若所作的修改是临时的，且不想影响原有或以后的设定，那我们就nested sub-shell，反之，则用non-named command group。
是的，光从command line来看，( )与{ }的差别就讲完了，够轻松吧~~~ ^_^
然而，若这两个meta用在其它command meta或领域中(如Regular Expression)，还是有很多差别的。只是，我不打算再去说明了，留给读者自己慢慢发掘好了...
我这里只想补充一个概念，就是function。
所谓的function，就是用一个名字去命名一个 command group，然后再调用这个名字去执行command group。从non-named command group来推断，大概你也可以猜到我要说的是{ }了吧？
在bash中，function的定义方式有两种：
方式一：

```
function function_name {
  command1
  command2
  command3
  ....
}
```

方式二：

```
fuction_name () {
  command1
  command2
  command3
  ....
}
```

用哪一种方式无所谓，只是若碰到所定义的名称与现有的命令或别名(Alias)冲突的话，方式二或许会失败。
但方式二起码可以少打function这一串英文字母，对懒人来说(如我)，又何乐不为呢？... ^_^
function在某一程度来说，也可称为"函式"，但请不要与传统编程所使用的函式(library)搞混了，毕竟两者差异很大。
惟一相同的是，我们都可以随时用"已定义的名称"来调用它们...
若我们在shell操作中，需要不断的重复执行某些命令，我们首先想到的，或许是将命令写成命令稿(shell script)。不过，我们也可以写成function，然后在command line中打上function_name就可当一般的script来使用了。只是若你在shell中定义的function，除了可用unset function_name取消外，一旦退出shell，function也跟着取消。
然而，在script中使用function却有许多好处，除了可以提高整体script的执行效能外(因为已被加载)，还可以节省许多重复的代码...
简单而言，若你会将多个命令写成script以供调用的话，那，你可以将function看成是script中的script ... ^_^
而且，透过上一章介绍的source命令，我们可以自行定义许许多多好用的function，再集中写在特定文件中，然后，在其它的script中用source将它们加载并反复执行。
若你是RedHat Linux的使用者，或许，已经猜得出/etc/rc.d/init.d/functions这个文件是作啥用的了~~~ ^_^
[]

SHELL十三问之八：$(( )) 与 $( ) 还有${ } 差在哪？
文章整理: 文章来源: 网络
我们上一章介绍了( )与{ }的不同，这次让我们扩展一下，看看更多的变化：$( )与${ }又是啥玩意儿呢？
在bash shell中，$( )与`` ` `` (反引号)都是用来做命令替换用(command substitution)的。
所谓的命令替换与我们第五章学过的变量替换差不多，都是用来重组命令行：

*完成引号里的命令行，然后将其结果替换出来，再重组命令行。
例如：
$ echo the last sunday is $(date -d "last sunday" +%Y-%m-%d)
如此便可方便得到上一星期天的日期了...
上例是在linux下，在FreeBSD下应该用下面的：
echo the last Sunday is `date  Cv Sunday +%Y%m%d`
在操作上，用$( )或` `都无所谓，只是我"个人"比较喜欢用$( )，理由是：
1, ` `很容易与' ' (单引号)搞混乱，尤其对初学者来说。
有时在一些奇怪的字形显示中，两种符号是一模一样的(直竖两点)。
当然了，有经验的朋友还是一眼就能分辩两者。只是，若能更好的避免混乱，又何乐不为呢？
2,在多层次的复合替换中，` `须要额外的跳脱(\`)处理，而$( )则比较直观。例如：
这是错的：
command1 `command2 `command3` `
原本的意图是要在command2 `command3`先将command3提换出来给command 2处理，然后再将结果传给command1 `command2 ...`来处理。
然而，真正的结果在命令行中却是分成了`command2 `与``两段。
正确的输入应该如下：
command1 `command2 \`command3\` `
要不然，换成$( )就没问题了：
command1 $(command2 $(command3))
只要你喜欢，做多少层的替换都没问题啦~~~ ^_^
不过，$( )并不是没有弊端的...
首先，` `基本上可用在全部的unix shell中使用，若写成shell script，其移植性比较高。
而$( )并不见的每一种shell都能使用，我只能跟你说，若你用bash2的话，肯定没问题...
接下来，再让我们看${ }吧...它其实就是用来作变量替换用的啦。
一般情况下，$var与${var}并没有啥不一样。
但是用${ }会比较精确的界定变量名称的范围，比方说：
$ A=B
$ echo $AB
原本是打算先将$A的结果替换出来，然后再补一个B字母于其后，但在命令行上，真正的结果却是只会替换变量名称为AB的值出来 ...
若使用${ }就没问题了：
$ echo ${A}B
BB
不过，假如你只看到${ }只能用来界定变量名称的话，那你就实在太小看bash了 u
有兴趣的话，你可先参考一下cu本版的精华文章：
http://www.chinaunix.net/forum/viewtopic.php?t=201843
为了完整起见，我这里再用一些例子加以说明${ }的一些特异功能：
假设我们定义了一个变量为：
file=/dir1/dir2/dir3/my.file.txt
我们可以用${ }分别替换获得不同的值：
${file#*/}：拿掉第一条/及其左边的字符串：dir1/dir2/dir3/my.file.txt
${file##*/}：拿掉最后一条/及其左边的字符串：my.file.txt
${file#*.}：拿掉第一个. 及其左边的字符串：file.txt
${file##*.}：拿掉最后一个. 及其左边的字符串：txt
${file%/*}：拿掉最后条/及其右边的字符串：/dir1/dir2/dir3
${file%%/*}：拿掉第一条/及其右边的字符串：(空值)
${file%.*}：拿掉最后一个. 及其右边的字符串：/dir1/dir2/dir3/my.file

${file%%.*}：拿掉第一个. 及其右边的字符串：/dir1/dir2/dir3/my
记忆的方法为：
#是去掉左边(在鉴盘上#在$之左边)
%是去掉右边(在鉴盘上%在$之右边)
单一符号是最小匹配　r两个符号是最大匹配。
${file:0:5}：提取最左边的5个字节：/dir1
${file:5:5}：提取第5个字节右边的连续5个字节：/dir2
我们也可以对变量值里的字符串作替换：
${file/dir/path}：将第一个dir替换为path：/path1/dir2/dir3/my.file.txt
${file//dir/path}：将全部dir替换为path：/path1/path2/path3/my.file.txt
利用${ }还可针对不同的变量状态赋值(没设定、空值、非空值)：
${file-my.file.txt}：假如$file没有设定，则使用my.file.txt作传回值。(空值及非空值时不作处理)
${file:-my.file.txt}：假如$file没有设定或为空值，则使用my.file.txt作传回值。(非空值时不作处理)
${file+my.file.txt}：假如$file设为空值或非空值，均使用my.file.txt作传回值。(没设定时不作处理)
${file:+my.file.txt}：若$file为非空值，则使用my.file.txt作传回值。(没设定及空值时不作处理)
${file=my.file.txt}：若$file没设定，则使用my.file.txt作传回值，同时将$file赋值为my.file.txt。(空值及非空值时不作处理)
${file:=my.file.txt}：若$file没设定或为空值，则使用my.file.txt作传回值，同时将$file赋值为my.file.txt。(非空值时不作处理
)
${file?my.file.txt}：若$file没设定，则将my.file.txt输出至STDERR。(空值及非空值时不作处理)
${file:?my.file.txt}：若$file没设定或为空值，则将my.file.txt输出至STDERR。(非空值时不作处理)
tips:
以上的理解在于,你一定要分清楚unset与null及non-null这三种赋值状态.
一般而言, :与null有关,若不带:的话, null不受影响,若带:则连null也受影响.
还有哦，${#var}可计算出变量值的长度：
${#file}可得到27，因为/dir1/dir2/dir3/my.file.txt刚好是27个字节...
接下来，再为大家介稍一下bash的组数(array)处理方法。
一般而言，A="a b c def"这样的变量只是将$A替换为一个单一的字符串，
但是改为A=(a b c def)，则是将$A定义为组数...
bash的组数替换方法可参考如下方法：
${A[@]}或${A
· }可得到a b c def (全部组数)
${A[0]}可得到a (第一个组数)，${A[1]}则为第二个组数...
${#A[@]}或${#A
· }可得到4 (全部组数数量)
${#A[0]}可得到1 (即第一个组数(a)的长度)，${#A[3]}可得到3 (第四个组数(def)的长度)
A[3]=xyz则是将第四个组数重新定义为xyz ...
诸如此类的....
能够善用bash的$( )与${ }可大大提高及简化shell在变量上的处理能力哦
好了，最后为大家介绍$(( ))的用途吧：它是用来作整数运算的。
在bash中，$(( ))的整数运算符号大致有这些：
+ - * /：分别为"加、减、乘、除"。
%：余数运算
& | ^ !：分别为"AND、OR、XOR、NOT"运算。
例：
$ a=5; b=7; c=2
$ echo $((a+b*c))
19
$ echo $(((a+b)/c))

6
$ echo $(((a*b)%c))
1
在$(( ))中的变量名称，可于其前面加$符号来替换，也可以不用，如：
$(($a + $b * $c))也可得到19的结果
此外，$(( ))还可作不同进位(如二进制、八进位、十六进制)作运算呢，只是，输出结果皆为十进制而已：
echo $((16#2a))结果为42 (16进位转十进制)
以一个实用的例子来看看吧：
假如当前的 umask是022，那么新建文件的权限即为：
$ umask 022
$ echo "obase=8;$((8#666 & (8#777 ^ 8#$(umask))))" | bc
644
事实上，单纯用(( ))也可重定义变量值，或作testing：
a=5; ((a++))可将$a重定义为6
a=5; ((a--))则为a=4
a=5; b=7; ((a < b))会得到 0 (true)的返回值。
常见的用于(( ))的测试符号有如下这些：
<：小于
>：大于
<=：小于或等于
>=：大于或等于
==：等于
!=：不等于
不过，使用(( ))作整数测试时，请不要跟[ ]的整数测试搞混乱了。(更多的测试我将于第十章为大家介绍)
上面的介绍，并没有详列每一种可用的状态，更多的，就请读者参考手册文件啰...
[]

SHELL十三问之九：$@ 与 $* 差在哪？
文章整理: 文章来源: 网络
要说$@与$*之前，需得先从shell script的positional parameter谈起...我们都已经知道变量(variable)是如何定义及替换的，这个不用再多讲了。但是，我们还需要知道有些变量是shell内定的，且其名称是我们不能随意修改的，其中就有positional parameter在内。
在shell script中，我们可用$0, $1, $2, $3 ...这样的变量分别提取命令行中的如下部份：
script_name parameter1 parameter2 parameter3 ...
我们很容易就能猜出$0就是代表shell script名称(路径)本身，而$1就是其后的第一个参数，如此类推....
须得留意的是IFS的作用，也就是，若IFS被quoting处理后，那么positional parameter也会改变。
如下例：
my.sh p1 "p2 p3" p4
由于在p2与p3之间的空格键被soft quote所关闭了，因此my.sh中的$2是"p2 p3"而$3则是p4 ...
还记得前两章我们提到fucntion时，我不是说过它是script中的script吗？ ^_^
是的，function一样可以读取自己的(有别于script的) postitional parameter，惟一例外的是$0而已。
举例而言：假设my.sh里有一个fucntion叫my_fun ,若在script中跑my_fun fp1 fp2 fp3，那么，function内的$0是my.sh，而$1则是fp1而非p1了...
不如写个简单的my.sh script 看看吧：

```
#!/bin/bash
my_fun() {
  echo '$0 inside function is '$0
  echo '$1 inside function is '$1
```

```
    echo '$2 inside function is '$2
}
echo '$0 outside function is '$0
echo '$1 outside function is '$1
echo '$2 outside function is '$2
my_fun fp1 "fp2 fp3"
```

然后在command line中跑一下script就知道了：

```
chmod +x my.sh
./my.sh p1 "p2 p3"
$0 outside function is ./my.sh
$1 outside function is p1
$2 outside function is p2 p3
$0 inside function is ./my.sh
$1 inside function is fp1
$2 inside function is fp2 fp3
```

然而，在使用positional parameter的时候，我们要注意一些陷阱哦：

* $10不是替换第10个参数，而是替换第一个参数($1)然后再补一个0于其后！

也就是，my.sh one two three four five six seven eigth nine ten这样的command line，my.sh里的$10不是ten而是one0哦...小心小心！

要抓到ten的话，有两种方法：

方法一是使用我们上一章介绍的${ }，也就是用${10}即可。

方法二，就是shift了。

用通俗的说法来说，所谓的shift就是取消positional parameter中最左边的参数($0不受影响)。其默认值为1，也就是shift或shift 1 都是取消$1，而原本的$2则变成$1、$3变成$2 ...若shift 3则是取消前面三个参数，也就是原本的$4将变成$1 ...

那，亲爱的读者，你说要shift掉多少个参数，才可用$1取得${10}呢？^_^

okay，当我们对positional parameter有了基本概念之后，那再让我们看看其它相关变量吧。

首先是$#：它可抓出positional parameter的数量。

以前面的my.sh p1 "p2 p3"为例：

由于p2与p3之间的IFS是在soft quote中，因此$#可得到2的值。

但如果p2与p3没有置于quoting中话，那$#就可得到3的值了。

同样的道理在function中也是一样的...

因此，我们常在shell script里用如下方法测试script是否有读进参数：

```
[ $# = 0 ]
```

假如为0，那就表示script没有参数，否则就是有带参数...

接下来就是$@与$*：

精确来讲，两者只有在soft quote中才有差异，否则，都表示"全部参数"($0除外)。

举例来说好了：

若在command line上跑my.sh p1 "p2 p3" p4的话，

不管是$@还是$*，都可得到p1 p2 p3 p4就是了。

但是，如果置于soft quote中的话：

"$@"则可得到"p1" "p2 p3" "p4"这三个不同的词段(word)　r

"$*"则可得到"p1 p2 p3 p4"这一整串单一的词段。

我们可修改一下前面的my.sh，使之内容如下：

```
#!/bin/bash
my_fun() {
    echo "$#"
}
echo 'the number of parameter in "$@" is '$(my_fun "$@")
```

echo 'the number of parameter in "$*" is '$(my_fun "$*")
然后再执行./my.sh p1 "p2 p3" p4就知道$@与$*差在哪了...
the number of parameter in "$@" is3
the number of parameter in "$*" is1
[]

SHELL十三问之十一：> 与 < 差在哪？
文章整理: 文章来源: 网络
11.1
谈到 I/O redirection ， 不妨先让我们认识一下 File Descriptor (FD) 。
程序的运算，在大部份情况下都是进行数据(data)的处理， 这些数据从哪读进？又送出到哪里呢？这就是 file descriptor (FD) 的功用了。
在 shell 程序中，最常使用的 FD 大概有三个，分别为：
0: Standard Input (STDIN)
1: Standard Output (STDOUT)
2: Standard Error Output (STDERR)
在标准情况下，这些 FD 分别跟如下设备(device)关联：
stdin(0): keyboard
stdout(1): monitor
stderr(2): monitor
我们可以用如下下命令测试一下：

$ mail -s test root
this is a test mail.
please skip.
^d (同时按 crtl 跟 d 键)
很明显，mail 程序所读进的数据，就是从 stdin 也就是 keyboard 读进的。不过，不见得每个程序的 stdin 都跟 mail 一样从 keyboard 读进，因为程序作者可以从档案参数读进 stdin ，如：
$ cat /etc/passwd
但，要是cat 之后没有档案参数则又如何呢？
哦，请您自己玩玩看啰.... ^_^

$ cat
(请留意数据输出到哪里去了，最后别忘了按 ^d 离开...)
至于 stdout 与 stderr ，嗯... 等我有空再续吧... ^_^
还是，有哪位前辈要来玩接龙呢？
11.2
沿文再续，书接上一回... ^_^
相信，经过上一个练习后，你对 stdin 与 stdout 应该不难理解吧？ 然后，让我们继续看 stderr 好了。
事实上，stderr 没甚么难理解的：说穿了就是"错误信息"要往哪边送而已...
比方说，若读进的档案参数是不存在的，那我们在 monitor 上就看到了：

$ ls no.such.file
ls: no.such.file: No such file or directory
若，一个命令同时产生stdout 与 stderr 呢？
那还不简单，都送到 monitor 来就好了：

$ touch my.file

$ ls my.file no.such.file
ls: no.such.file: No such file or directory
my.file
okay，至此，关于 FD 及其名称、还有相关联的设备，相信你已经没问题了吧？
那好，接下来让我们看看如何改变这些 FD 的预设数据信道，我们可用 < 来改变读进的数据信道(stdin)，使之从指定的档案读进。我们可用 > 来改变送出的数据信道(stdout, stderr)，使之输出到指定的档案。
比方说：

$ cat < my.file
就是从my.file 读进数据

$ mail -s test root < /etc/passwd
则是从/etc/passwd 读进...
这样一来，stdin 将不再是从 keyboard 读进，而是从档案读进了...
严格来说，< 符号之前需要指定一个 FD 的(之间不能有空白)， 但因为 0 是 < 的默认值，因此 < 与 0< 是一样的 u
okay，这个好理解吧？
那，要是用两个 << 又是啥呢？
这是所谓的 HERE Document ，它可以让我们输入一段文本，直到读到 << 后指定的字符串。
比方说：

$ cat <<FINISH
first line here
second line there
third line nowhere
FINISH
这样的话，cat 会读进 3 行句子，而无需从 keyboard 读进数据且要等 ^d 结束输入。
至于 > 又如何呢？
且听下回分解....

11.3
okay，又到讲古时间~~~
当你搞懂了 0< 原来就是改变 stdin 的数据输入信道之后，相信要理解如下两个 redirection 就不难了：
* 1>
* 2>
前者是改变 stdout 的数据输出信道，后者是改变 stderr 的数据输出信道。
两者都是将原本要送出到 monitor 的数据转向输出到指定档案去。
由于 1 是 > 的默认值，因此，1> 与 > 是相同的，都是改 stdout 。
用上次的 ls 例子来说明一下好了：

$ ls my.file no.such.file 1>file.out
ls: no.such.file: No such file or directory
这样monitor 就只剩下 stderr 而已。因为 stdout 给写进 file.out 去了。

$ ls my.file no.such.file 2>file.err
my.file
这样monitor 就只剩下 stdout ，因为 stderr 写进了 file.err 。

$ ls my.file no.such.file 1>file.out 2>file.err

这样monitor 就啥也没有，因为 stdout 与 stderr 都给转到档案去了...
呵~~~ 看来要理解 > 一点也不难啦　u是不？没骗你吧？ ^_^
不过，有些地方还是要注意一下的。
首先，是同时写入的问题。比方如下这个例子：

$ ls my.file no.such.file 1>file.both 2>file.both
假如stdout(1) 与 stderr(2) 都同时在写入 file.both 的话， 则是采取 “ 覆盖 ” 方式：后来写入的覆盖前面的。
让我们假设一个 stdout 与 stderr 同时写入 file.out 的情形好了：
* 首先 stdout 写入10个字符
* 然后 stderr 写入 6 个字符
那么，这时候原本 stdout 的前面 6 个字符就被 stderr 覆盖掉了。
那，如何解决呢？所谓山不转路转、路不转人转嘛，
我们可以换一个思维：将 stderr 导进 stdout 或将 stdout 导进 sterr ， 而不是大家在抢同一份档案，不就行了　u
bingo　u就是这样啦：
* 2>&1 就是将 stderr 并进 stdout 作输出
* 1>&2 或 >&2 就是将 stdout 并进 stderr 作输出
于是，前面的错误操作可以改为：

$ ls my.file no.such.file 1>file.both 2>&1
或
$ ls my.file no.such.file 2>file.both >&2
这样，不就皆大欢喜了吗？ 呵~~~ ^_^
不过，光解决了同时写入的问题还不够，我们还有其它技巧需要了解的。
故事还没结束，别走开　u广告后，我们再回来...　u

11.4
okay，这次不讲 I/O Redirction ， 讲佛吧...
(有没搞错？　u网中人是否头壳烧坏了？...) 嘻~~~ ^_^
学佛的最高境界，就是"四大皆空"。至于是空哪四大块？我也不知，因为我还没到那境界...
但这个"空"字，却非常值得我们返复把玩的：
--- 色即是空、空即是色　u
好了，施主要是能够领会"空"的禅意，那离修成正果不远矣~~~
在 Linux 档案系统里，有个设备档位于 /dev/null 。
许多人都问过我那是甚么玩意儿？我跟你说好了：那就是"空"啦　u
没错　u空空如也的空就是 null 了.... 请问施主是否忽然有所顿误了呢？然则恭喜了~~~ ^_^
这个 null 在 I/O Redirection 中可有用得很呢：
* 若将 FD1 跟 FD2 转到 /dev/null 去，就可将 stdout 与 stderr 弄不见掉。
* 若将 FD0 接到 /dev/null 来，那就是读进 nothing 。
比方说，当我们在执行一个程序时，画面会同时送出 stdout 跟 stderr ， 假如你不想看到 stderr (也不想存到档案去)，那可以：

$ ls my.file no.such.file 2>/dev/null
my.file
若要相反：只想看到stderr 呢？还不简单　u将 stdout 弄到 null 就行：

$ ls my.file no.such.file >/dev/null
ls: no.such.file: No such file or directory
那接下来，假如单纯只跑程序，不想看到任何输出结果呢？

SHELL十三问

哦，这里留了一手上次节目没讲的法子，专门赠予有缘人　u... ^_^
除了用 >/dev/null 2>&1 之外，你还可以如此：

$ ls my.file no.such.file &>/dev/null
(提示：将 &> 换成 >& 也行啦~~!)
okay？讲完佛，接下来，再让我们看看如下情况：

$ echo "1" > file.out
$ cat file.out
1
$ echo "2" > file.out
$ cat file.out
2
看来，我们在重导stdout 或 stderr 进一份档案时，似乎永远只获得最后一次导入的结果。
那，之前的内容呢？
呵~~~ 要解决这个问提很简单啦，将 > 换成 >> 就好：

$ echo "3" >> file.out
$ cat file.out
2
3
如此一来，被重导的目标档案之内容并不会失去，而新的内容则一直增加在最后面去。
easy ？ 呵 ... ^_^
但，只要你再一次用回单一的 > 来重导的话，那么，旧的内容还是会被"洗"掉的　u
这时，你要如何避免呢？
----备份　u yes，我听到了　u不过.... 还有更好的吗？
既然与施主这么有缘份，老纳就送你一个锦囊妙法吧：

$ set -o noclobber
$ echo "4" > file.out
-bash: file: cannot overwrite existing file
那，要如何取消这个"限制"呢？
哦，将 set -o 换成 set +o 就行：

$ set +o noclobber
$ echo "5" > file.out
$ cat file.out
5
再问：那... 有办法不取消而又"临时"盖写目标档案吗？
哦，佛曰：不可告也　u
啊~~~ 开玩笑的、开玩笑的啦~~~ ^_^ 唉，早就料到人心是不足的了　u

$ set -o noclobber
$ echo "6" >| file.out
$ cat file.out
6
留意到没有：在> 后面再加个" | "就好(注意： > 与 | 之间不能有空白哦)....
呼.... (深呼吸吐纳一下吧)~~~ ^_^

再来还有一个难题要你去参透的呢：

$ echo "some text here" > file
$ cat < file
some text here
$ cat < file > file.bak
$ cat < file.bak
some text here
$ cat < file > file
$ cat < file
嗯？ u注意到没有？ u u
---- 怎么最后那个 cat 命令看到的 file 竟是空的？ u
why? why? why?
同学们：下节课不要迟到啰~~~!

11.5
当当当~~~ 上课啰~~~ ^_^
前面提到：$ cat < file > file 之后原本有内容的档案结果却被洗掉了 u
要理解这一现像其实不难，这只是 priority 的问题而已：
* 在 IO Redirection 中，stdout 与 stderr 的管道会先准备好，才会从 stdin 读进资料。
也就是说，在上例中，> file 会先将 file 清空，然后才读进 < file， 但这时候档案已经被清空了，因此就变成读不进任何数据了...
哦~~~ 原来如此~~~~ ^_^
那... 如下两例又如何呢？

$ cat <> file
$ cat < file >> file
嗯... 同学们，这两个答案就当练习题啰，下节课之前请交作业 u
好了，I/O Redirection 也快讲完了，sorry，因为我也只知道这么多而已啦~~~ 嘻~~ ^_^
不过，还有一样东东是一定要讲的，各位观众(请自行配乐~!#@!$%) ：
---- 就是 pipe line 也 u
谈到 pipe line，我相信不少人都不会陌生：
我们在很多 command line 上常看到的" | "符号就是 pipe line 了。
不过，究竟 pipe line 是甚么东东呢？
别急别急... 先查一下英汉字典，看看 pipe 是甚么意思？
没错 u它就是"水管"的意思...
那么，你能想象一下水管是怎么一根接着一根的吗？
又，每根水管之间的 input 跟 output 又如何呢？
嗯？？
灵光一闪：原来 pipe line 的 I/O 跟水管的 I/O 是一模一样的：
* 上一个命令的 stdout 接到下一个命令的 stdin 去了 u
的确如此... 不管在 command line 上你使用了多少个 pipe line，
前后两个 command 的 I/O 都是彼此连接的 u(恭喜：你终于开窍了 u ^_^)
不过... 然而... 但是... ... stderr 呢？
好问题 u不过也容易理解：
* 若水管漏水怎么办？
也就是说：在 pipe line 之间，前一个命令的 stderr 是不会接进下一命令的 stdin 的， 其输出，若不用 2> 导到 file 去的话，它还是送到监视器上面来 u

这点请你在 pipe line 运用上务必要注意的。
那，或许你又会问：
* 有办法将 stderr 也喂进下一个命令的 stdin 去吗？
(贪得无厌的家伙　u)
方法当然是有，而且你早已学过了　u ^_^
我提示一下就好：
* 请问你如何将 stderr 合并进 stdout 一同输出呢？
若你答不出来，下课之后再来问我吧... (如果你脸皮真够厚的话...)
或许，你仍意尤未尽　u或许，你曾经碰到过下面的问题：
* 在 cm1 | cm2 | cm3 ... 这段 pipe line 中，若要将 cm2 的结果存到某一档案呢？
若你写成 cm1 | cm2 > file | cm3 的话，
那你肯定会发现 cm3 的 stdin 是空的　u(当然啦，你都将水管接到别的水池了　u)
聪明的你或许会如此解决：

cm1 | cm2 > file ; cm3 < file
是的，你的确可以这样做，但最大的坏处是：这样一来，file I/O 会变双倍　u
在 command 执行的整个过程中，file I/O 是最常见的最大效能杀手。
凡是有经验的 shell 操作者，都会尽量避免或降低 file I/O 的频率。
那，上面问题还有更好方法吗？
有的，那就是 tee 命令了。
*所谓 tee 命令是在不影响原本 I/O 的情况下，将 stdout复制一份到档案去。
因此，上面的命令行可以如此打：

cm1 | cm2 | tee file | cm3
在预设上，tee 会改写目标档案，若你要改为增加内容的话，那可用 -a 参数达成。
基本上，pipe line 的应用在 shell 操作上是非常广泛的，尤其是在 text filtering 方面， 凡举 cat, more, head, tail, wc, expand, tr, grep, sed, awk, ... 等等文字处理工具， 搭配起 pipe line 来使用，你会惊觉 command line 原来是活得如此精彩的　u
常让人有"众里寻他千百度，蓦然回首，那人却在灯火阑珊处　u"之感... ^_^
....
好了，关于 I/O Redirection 的介绍就到此告一段落。
若日后有空的话，再为大家介绍其它在 shell 上好玩的东西　ubye... ^_^
[]

SHELL十三问之十二：你要 if 还是 case 呢？
文章整理: 文章来源: 网络
还记得我们在第10 章所介绍的 return value 吗？
若你记得 return value ，我想你也应该记得了 && 与 || 是甚么意思吧？
用这两个符号再配搭 command group 的话，我们可让 shell script 变得更加聪明哦。
比方说：

```
comd1 && {
   comd2
   comd3
} || {
   comd4
   comd5
}
```

意思是说：
假如comd1 的 return value 为 true 的话，

然则执行 comd2 与 comd3 ,
否则执行 comd4 与 comd5 。
事实上，我们在写 shell script 的时候，经常需要用到这样那样的条件以作出不同的处理动作。
用 && 与 || 的确可以达成条件执行的效果，然而，从"人类语言"上来理解，却不是那么直观。
更多时候，我们还是喜欢用 if .... then ... else ... 这样的 keyword 来表达条件执行。
在 bash shell 中，我们可以如此修改上一段代码：

```
if comd1
then
   comd2
   comd3
else
   comd4
   comd5
fi
```

这也是我们在shell script 中最常用到的 if 判断式：
只要if 后面的 command line 返回 true 的 return value (我们最常用 test 命令来送出 return value)，然则就执行 then 后面的命令，否则执行 else 后的命令 rfi 则是用来结束判断式的 keyword 。
在 if 判断式中，else 部份可以不用，但 then 是必需的。
(若 then 后不想跑任何 command ，可用" ： " 这个 null command 代替)。
当然，then 或 else 后面，也可以再使用更进一层的条件判断式，这在 shell script 设计上很常见。
若有多项条件需要"依序"进行判断的话，那我们则可使用 elif 这样的 keyword ：

```
if comd1; then
   comd2
elif comd3; then
   comd4
else
   comd5
fi
```

意思是说：
若comd1 为 true ，然则执行 comd2 r
否则再测试 comd3 ，然则执行 comd4 r
倘若 comd1 与 comd3 均不成立，那就执行 comd5 。
if 判断式的例子很常见，你可从很多 shell script 中看得到，我这里就不再举例子了...
接下来要为大家介绍的是 case 判断式。
虽然 if 判断式已可应付大部份的条件执行了，然而，在某些场合中，却不够灵活，尤其是在 string 式样的判断上，比方如下：

```
QQ () {
   echo -n "Do you want to continue? (Yes/No): "
   read YN
   if [ "$YN" = Y -o "$YN" = y -o "$YN" = "Yes" -o "$YN" = "yes" -o "$YN" = "YES" ]
   then
      QQ
   else
      exit 0
   fi
}
QQ
```

从例中，我们看得出来，最麻烦的部份是在于判断YN 的值可能有好几种式样。

聪明的你或许会如此修改：

```
...
if echo "$YN" | grep -q '^[Yy]\([Ee][Ss]\)*$'
...
```

也就是用Regular Expression 来简化代码。(我们有机会再来介绍 RE)
只是... 是否有其它更方便的方法呢？
有的，就是用 case 判断式即可：

```
QQ () {
    echo -n "Do you want to continue? (Yes/No): "
    read YN
   case "$YN" in
       [Yy]|[Yy][Ee][Ss])
           QQ
           ;;
       *)
           exit 0
           ;;
    esac
}
QQ
```

我们常case 的判断式来判断某一变量在同的值(通常是 string)时作出不同的处理，
比方说，判断 script 参数以执行不同的命令。
若你有兴趣、且用 Linux 系统的话，不妨挖一挖 /etc/init.d/* 里那堆 script 中的 case 用法。
如下就是一例：

```
case "$1" in
  start)
        start
        ;;
  stop)
        stop
        ;;
  status)
        rhstatus
        ;;
  restart|reload)
        restart
        ;;
  condrestart)
        [ -f /var/lock/subsys/syslog ] && restart || :
        ;;
  *)
        echo $"Usage: $0 {start|stop|status|restart|condrestart}"
        exit 1
esac
```

(若你对 positional parameter 的印像已经模糊了，请重看第 9 章吧。)
[]

SHELL十三问之十三：for、while 与 until 差在哪？

SHELL十三问

文章整理: 文章来源: 网络
终于，来到shell 十三问的最后一问了... 长长吐一口气~~~~
最后要介绍的是 shell script 设计中常见的"循环"(loop)。
所谓的 loop 就是 script 中的一段在一定条件下反复执行的代码。
bash shell 中常用的 loop 有如下三种：
* for
* while
* until
for loop 是从一个清单列表中读进变量值，并"依次"的循环执行 do 到 done 之间的命令行。
例：

```
for var in one two three four five
do
   echo ----------
   echo '$var is '$var
   echo
done
```

上例的执行结果将会是：
1) for 会定义一个叫 var 的变量，其值依次是 one two three four five 。
2) 因为有 5 个变量值，因此 do 与 done 之间的命令行会被循环执行 5 次。
3) 每次循环均用 echo 产生三行句子。
而第二行中不在 hard quote 之内的 $var 会依次被替换为 one two three four five 。
4) 当最后一个变量值处理完毕，循环结束。
我们不难看出，在 for loop 中，变量值的多寡，决定循环的次数。
然而，变量在循环中是否使用则不一定，得视设计需求而定。
倘若 for loop 没有使用 in 这个 keyword 来指定变量值清单的话，其值将从 $@ (或 $*)中继承：

```
for var; do
   ....
done
```

(若你忘记了 positional parameter ， 请温习第 9 章...)
for loop 用于处理"清单"(list)项目非常方便，其清单除了可明确指定或从 positional parameter 取得之外，也可从变量替换或命令替换取得... (再一次提醒：别忘了命令行的"重组"特性 u)
然而，对于一些"累计变化"的项目(如整数加减)，for 亦能处理：

```
for ((i=1;i<=10;i++))
do
  echo "num is $i"
done
```

除了for loop ， 上面的例子我们也可改用 while loop 来做到：

```
num=1
while [ "$num" -le 10 ]; do
   echo "num is $num"
   num=$(($num + 1))
done
```

while loop 的原理与 for loop 稍有不同：
它不是逐次处理清单中的变量值，而是取决于 while 后面的命令行之 return value ：
* 若为 ture ， 则执行 do 与 done 之间的命令，然后重新判断 while 后的 return value 。
* 若为 false ， 则不再执行 do 与 done 之间的命令而结束循环。
分析上例：
1) 在 while 之前，定义变量 num=1 。

2) 然后测试(test) $num 是否小于或等于 10 。
3) 结果为 true ， 于是执行 echo 并将 num 的值加一。
4) 再作第二轮测试，其时 num 的值为 1+1=2 ， 依然小于或等于 10，因此为 true ， 继续循环。
5) 直到 num 为 10+1=11 时，测试才会失败... 于是结束循环。
我们不难发现：
* 若 while 的测试结果永远为 true 的话，那循环将一直永久执行下去：

```
while :; do
   echo looping...
done
```

上例的" : "是 bash 的 null command ， 不做任何动作，除了送回 true 的 return value 。
因此这个循环不会结束，称作死循环。
死循环的产生有可能是故意设计的(如跑 daemon)，也可能是设计错误。
若要结束死寻环，可透过 signal 来终止(如按下 ctrl-c)。
(关于 process 与 signal ， 等日后有机会再补充，十三问暂时略过。)
一旦你能够理解 while loop 的话，那，就能理解 until loop ：
* 与 while 相反，until 是在 return value 为 false 时进入循环，否则结束。
因此，前面的例子我们也可以轻松的用 until 来写：

```
num=1
until [ ! "$num" -le 10 ]; do
   echo "num is $num"
   num=$(($num + 1))
done
```

或是：

```
num=1
until [ "$num" -gt 10 ]; do
   echo "num is $num"
   num=$(($num + 1))
done
```

okay ， 关于 bash 的三个常用的 loop 暂时介绍到这里。
在结束本章之前，再跟大家补充两个与 loop 有关的命令：
* break
* continue
这两个命令常用在复合式循环里，也就是在 do ... done 之间又有更进一层的 loop ，
当然，用在单一循环中也未尝不可啦... ^_^
break 是用来打断循环，也就是"强迫结束" 循环。
若 break 后面指定一个数值 n 的话，则"从里向外"打断第 n 个循环，默认值为 break 1 ， 也就是打断当前的循环。
在使用 break 时需要注意的是， 它与 return 及 exit 是不同的：
* break 是结束 loop
* return 是结束 function
* exit 是结束 script/shell
而 continue 则与 break 相反：强迫进入下一次循环动作。
若你理解不来的话，那你可简单的看成：在 continue 到 done 之间的句子略过而返回循环顶端...
与 break 相同的是：continue 后面也可指定一个数值 n ， 以决定继续哪一层(从里向外计算)的循环，
默认值为 continue 1 ， 也就是继续当前的循环。
在 shell script 设计中，若能善用 loop ， 将能大幅度提高 script 在复杂条件下的处理能力。
请多加练习吧....

好了，该是到了结束的时候了。

SHELL十三问

婆婆妈妈的跟大家啰唆了一堆关于 shell 的基础概念，目的不是要告诉大家"答案"，而是要带给大家"启发"...
在日后关于 shell 的讨论中，我或许会经常用"链接"方式指引回来十三问中的内容，以便我们在进行技术探讨时彼此能有一些讨论基础，而不至于各说各话、徒费时力。但，更希望十三问能带给你更多的思考与乐趣，至为重要的是透过实作来加深理解。
是的，我很重视"实作"与"独立思考"这两项学习要素，若你能够掌握其中真义，那请容我说声：
--- 恭喜　u十三问你没白看了　u ^_^
p.s.
至于补充问题部份，我暂时不写了。而是希望：
1) 大家扩充题目。
2) 一起来写心得。
Good luck and happy studying!
[]

《Unix Shell 实例精解》学习笔记之grep篇
文章整理: 文章来源: 网络
1．grep的含义是“ 全局搜索正则表达式(RE)并打印该行 ”
2.grep 的选项
选项　功能
-b 在各行之前放置它发现的块号。有时在根据上下文定位磁盘字块时有用
-c 显示匹配行数而不是内容
-h 不显示文件名
-I 在座比较时忽略字母大小写
-n 文件中每行之前给出它的相关行号
-s 无声操作。即除了错误消息外不做任何显示。用于检查退出状态
-v 把搜索翻转为只显示不匹配的行
-w 把表达式当作一个次来搜索，相当于用\<和\>括起来
3．grep命令的退出状态
如果grep操作成功，则状态是0，如果模式没找到，状态是1，如果文件没找到，状态是2。如果操作被取消，则状态是130。
查看状态的方法：
在csh中用echo $status。
在sh和ksh中用echo $? 。
例如
$ echo $? 0
5
4.带正则表达式的grep举例：
用于这些例子的文本文件叫datafile，位于chap03目录(你也可以放在别的目录下)。datafile内容如下：
northwest NW Charles Main　3.0 .98 3 34
western WE Sharon Gray　5.3 .97 5 23
southwest SW Lewis Dalsass　2.7 .8 2 18
southern SO Suan Chin　5.1 .95 4 15
southeast SE　Patricia Hemenway　4.0 .7 4 17
eastern EA TB Savage　4.4 .84 5 20
northeast NE AM Main Jr.　5.1 .94 3 13
north NO Margot Weber　4.5 .89 5 9
central CT Ann Stephens　5.7 .94 5 13
1) % grep NW datafile
解释：打印datafile中包含NW的行

2) grep NW d*
解释：打印所有以d开头的文件中含有NW的文件。
3) % grep '^n' datafile
解释：打印文件datafile中所有以字母n开头的行
4) % grep TB Savage datafile
解释：在Savage和datafile文件中查找有TB的行
5) % grep 'TB Savage' datafile
解释：在datafile文件中查找含有TB Savage的行并打印。
6) % grep '^[we]' datafile
解释：打印datafile中以w或者e开头的行
7) % grep 'ss* ' datafile
northwest NW Charles Main 3.0 .98 3 34
southwest SW Lewis Dalsass 2.7 .8 2 18
解释：打印所有包含一个s并跟0个或者多个s，然后跟一个空格的行
6．用管道的grep.
grep 可以从管道得到输入。
% ls Cl
drwxr-xr-x 6 oracle dba 512 4月 3 21:49 chap10
drwxr-xr-x 2 oracle dba 512 4月 10 22:23 exam
-rwxr--r-- 1 oracle dba 1842 4月 3 21:51 readme.txt
-rwxr--r-- 1 oracle dba 1801 4月 3 21:51 unix_readme.txt
% ls Cl | grep '^d'
drwxr-xr-x 6 oracle dba 512 4月 3 21:49 chap10
drwxr-xr-x 2 oracle dba 512 4月 10 22:23 exam
% ls -l |grep '^[^d]'
-rwxr--r-- 1 oracle dba 1842 4月 3 21:51 readme.txt
-rwxr--r-- 1 oracle dba 1801 4月 3 21:51 unix_readme.txt
7．带选项的grep举例：
% grep Cc ‘ west ’ datafile
3
解释:计算datafile中含有west的总数。
8．egrep (扩展的grep)
egrep可以使用额外的正则表达式，如下表。
元字符 功能 例子 解释
+ 匹配一个或多个前驱字符 '[a-z]+ove' 匹配一个或多个小写字母，后跟ove
? 匹配0个或者1个前驱字符 'lo?ve' 将找到love或love
a|b 匹配a或者b 'love|hate' 与love或hate匹配
() 组字符 'lov(ely|able) 与lovely或lovable匹配
9．egrep 举例：
% egrep '2\.?[0-9]' datafile
解释：打印所有这样的行：它包含一个2，后跟0个或者一个句号，然后跟一个数字。
10． fgrep
fgrep 把所有的元字符都当作字符本身，只代表自己。
11． UNIX 工具试验参考答案（内容参考datebook）
(1) 打印包含San的行
% grep 'San' datebook
(2) 打印所有以J开头的人名所在的行
% grep '^J' datebook

(3) 打印以700结尾的行
% grep '700$' datebook
(4) 打印所有不包含834的行
% grep -v '834' datebook
(5) 打印出生在12月(December)的行
% grep '/12' datebook
(6) 打印工资是6位数的行，并给出行号
% grep -n '[0-9]\{6,\}$' datebook
[]

《Unix Shell 实例精解》学习笔记之sed篇
文章整理: 文章来源: 网络
1．sed命令简介
sed是流线型、非交互式编辑器。它允许你执行与vi和ex编辑器里一样的编辑任务。Sed 程序不是与编辑器交互式工作的，而是让你在命令行里敲入编辑的命令，给文件命名，然后在屏幕上查看命令输出结果。
2．sed工作原理
sed编辑器按一次处理一行的方式来处理文件，并把输出送到屏幕上。
3．sed 可以用寻址的方式来决定想要编辑哪一行。
4．sed 命令和选项
命令 功能
a\ 在当前行上添加一个文本行或者多个文本行
c\ 用新闻本改变（取代）当前行里的文本
d 删除行
i\ 在当前行之前插入文本
h 把模式空间内容复制到一个固定缓存
H 把模式空间内容添加到一个固定缓存
g 得到固定缓存里所有的禀复制到模式缓存，重写其内容
G 得到固定缓存的内容并复制到模式缓存，添加到里面
I 列出不打印的字符
p 打印行
n 读下一输入行，并开始用下一个命令处理换行符，而不是用第一个命令
q 结束或退出sed
r 从一个文件读如行
! 把命令应用到除了选出的行以外的其他所有行
s 把一个字串替换成另一个
替换标志
g 在一行上进行全局替换
p 打印行
w 把行写到一个文件中
x 用模式空间的内容交换固定缓存的内容
y 把一个字符转换成另一个(不能和整则表达式元字符一起使用)
5．sed元字符
　基本上，grep和vi使用的元字符都可以用在sed中。下表列出了一些特别的sed元字符：
元字符 &
功能 保存搜索串以便可以记在替换串里
例子　s/love/**&**/
解释　&号代表搜索串。串love将被星号包围的自身所替代；即love变成**love**
6.　sed 的实例(使用datafile)

SHELL十三问

datafile内容如下：
northwest NW Charles Main 3.0 .98 3 34
western WE Sharon Gray 5.3 .97 5 23
southwest SW Lewis Dalsass 2.7 .8 2 18
southern SO Suan Chin 5.1 .95 4 15
southeast SE Patricia Hemenway 4.0 .7 4 17
eastern EA TB Savage 4.4 .84 5 20
northeast NE AM Main Jr. 5.1 .94 3 13
north NO Margot Weber 4.5 .89 5 9
central CT Ann Stephens 5.7 .94 5 13
1> 打印：p命令
sed '/north/p' datafile
默认输出所有行，找到north的行重复打印
sed Cn '/north/p' datafile
禁止默认输出，只打印找到north的行
2> 删除：d命令
sed '3d' datafile
删除第三行，其余行输出到屏幕
sed '3,$d' datafile
从第3行到最后一行都删除，将剩余部分输出到屏幕
sed '/north/d' datafile
将含有north的行删除，其余输出到屏幕
3> 替换：s命令
sed 's/west/north/g' datafile
解释：找到datafile中的所有west并替换成north，将替换后的内容输出到屏幕。
sed 's/[0-9][0-9]$/&.5/' datafile
解释：在替代串里的&字符代表在搜索串中真正找到的。每个以两个数字结尾的行都被它自己取代，且要在后面加上.5
sed -n 's/Hemenway/Jones/gp' datafile
解释：所有的Hemenway所在的位置都用Jones来取代，而且只有改变的行被打印。-n与p命令选项相结合来禁止默认输出。g代表全局替换
sed -n 's/\(Mar\)got/\1ianne/p' datafile
解释：模式Mar被封装在括弧里且在一个专用寄存器里存为标记1。在替换串里它将被引用做\1。然后用Marianne替代Margot。
sed 's#3#88#g' datafile
解释：s命令后面的字符是搜索串和替换串之间的分界符。默认的分界符是一个正斜杠，但也可以改变（只有使用s命令时）。无论s命令后面跟什么字符，它都是新的串分界符。当搜索包含一个正斜杠的模式，如路径或生日时，这种技巧可能有用的 ^V^
4> 被选中的行的范围：逗号
sed -n '/west/,/east/p' datafile
解释：打印在west和east之间的模式范围内所有行。如果west出现在east之后，则打印从west到下一个east或者到文件末尾的行，无论哪种情况先出现都可以。
sed '/west/,/east/s/$/**VACA**/' datafile
解释：对于在模式west到east范围内的行，行末尾将用**VACA**来取代。
5> 多次编辑 -e 选项
sed -e '1,3d' -e 's/Hemenway/Jones/' datafile
-e选项允许多次编辑。不同的编辑顺序可能导致不同的结果。
例如，如果两个命令都执行了替换，第一次替换可能影响第二次替换。
6> 从文件中读取：r命令

sed '/Suan/r newfile' datafile

解释：r命令从newfile中读取内容，将内容输出到Suan的后面。如果datafile中Suan出现的次数不只一次，则分别放到Suan的后面。

7> 写入文件：w命令

sed -n '/north/w newfile' datafile

解释：w命令把指定的行写入到一个文件。本例中所有的包含north的行写入到newfile中。

等同于 sed -n '/north/p' datafile > newfile

8> 添加：a命令

```
$ sed '/north/a\
 > ---->THE NORTH SALES DISTRICT HAS MOVED<-----' datafile
northwest NW Charles Main      3.0 .98 3 34
 ---->THE NORTH SALES DISTRICT HAS MOVED<-----
western WE Sharon Gray      5.3 .97 5 23
southwest SW Lewis Dalsass      2.7 .8 2 18
southern SO Suan Chin      5.1  .95 4 15
southeast SE      Patricia Hemenway    4.0  .7  4  17
eastern  EA  TB Savage       4.4  .84  5  20
northeast  NE  AM Main Jr.       5.1  .94  3  13
---->THE NORTH SALES DISTRICT HAS MOVED<-----
north   NO  Margot Weber       4.5  .89 5  9
---->THE NORTH SALES DISTRICT HAS MOVED<-----
central  CT  Ann Stephens        5.7  .94   5  13
```

解释：红颜色的内容是要输入的内容。a\命令后面跟要添加的内容。奇怪的是a\后面必须另起一行，在输入要添加的内容，否则会提示命令错乱，真是搞不懂。

9> 插入：i命令

```
$ sed '/north/i\
 > ---->THE NORTH SALES DISTRICT HAS MOVED<-----' datafile
 ---->THE NORTH SALES DISTRICT HAS MOVED<-----
northwest NW Charles Main      3.0 .98 3 34
 ---->THE NORTH SALES DISTRICT HAS MOVED<-----
western WE Sharon Gray      5.3 .97 5 23
southwest SW Lewis Dalsass      2.7 .8 2 18
southern SO Suan Chin      5.1  .95 4 15
southeast SE      Patricia Hemenway    4.0  .7  4  17
eastern  EA  TB Savage       4.4  .84  5  20
---->THE NORTH SALES DISTRICT HAS MOVED<-----
northeast  NE  AM Main Jr.       5.1  .94  3  13
---->THE NORTH SALES DISTRICT HAS MOVED<-----
north   NO  Margot Weber       4.5  .89 5  9
central  CT  Ann Stephens        5.7  .94   5  13
```

10>下一个：n命令

```
$ sed '/eastern/{n;s/AM/Archie/;}' datafile
 northwest NW Charles Main      3.0 .98 3 34
 western WE Sharon Gray      5.3 .97 5 23
 southwest SW Lewis Dalsass      2.7 .8 2 18
 southern SO Suan Chin      5.1  .95 4 15
 southeast SE      Patricia Hemenway    4.0  .7  4  17
 eastern  EA  TB Savage       4.4  .84  5  20
```

northeast NE AM Main Jr. 5.1 .94 3 13
.....
解释：如果在某一行里模式eastern被匹配，n命令使sed区的下一行，用该行带换模式空间，用Archie替换AM，打印并继续。
11>变换：y 命令
% sed '1,3y/abcdefghijklmnopqrst/ABCDEFGHIJKLMNOPQRST/' datafile
解释:将对应字母进行转换。
12>退出：q命令
% sed '5q' datafile
解释：在打印了5行之后，用q命令退出sed程序。
13>保存和取得：h和G命令
$ sed -e '/southeast/h' -e '$G' datafile
northwest NW Charles Main 3.0 .98 3 34
western WE Sharon Gray 5.3 .97 5 23
southwest SW Lewis Dalsass 2.7 .8 2 18
southern SO Suan Chin 5.1 .95 4 15
southeast SE Patricia Hemenway 4.0 .7 4 17
eastern EA TB Savage 4.4 .84 5 20
northeast NE AM Main Jr. 5.1 .94 3 13
north NO Margot Weber 4.5 .89 5 9
central CT Ann Stephens 5.7 .94 5 13
southeast SE Patricia Hemenway 4.0 .7 4 17
解释：当sed处理文件时，每行都存在模式空间(pattern space)的临时缓存中。除非行被禁止打印或删除,否则行将在处理完后被打印到屏幕，然后请模式空间并把下一输入行保存在那里等待处理。在这个例子中，在找到模式之后，把它放在模式空间里，而且h命令复制它并把它存到另一个叫做保存缓存(holding buffer)中。
第二个sed指令里，当读入最后一行($)时,G命令告诉sed从包存缓存中取得该行并放回模式空间缓存，添加到当前存在那里的行中。本例子就是最后一行。
$ sed -e '/WE/{h;d;}' -e '/CT/G' datafile
northwest NW Charles Main 3.0 .98 3 34
southwest SW Lewis Dalsass 2.7 .8 2 18
southern SO Suan Chin 5.1 .95 4 15
southeast SE Patricia Hemenway 4.0 .7 4 17
eastern EA TB Savage 4.4 .84 5 20
northeast NE AM Main Jr. 5.1 .94 3 13
north NO Margot Weber 4.5 .89 5 9
central CT Ann Stephens 5.7 .94 5 13
western WE Sharon Gray 5.3 .97 5 23
解释：第一个命令h将找到了WE的行放到保存缓存中，然后删除该行；第二个命令/CT/G就是在找到了CT的行的后面加入保存缓存的内容。
14>G和g的区别
G命令在符合的条件行后面添加保存缓存中的内容；g命令用保存缓存中的内容覆盖符合条件的行。
15>sed 命令的花括号{}的作用
花括号{}中可以放入多个命令，每个命令后面要用分号;。
16>保存和交换：h 和 x命令。
$ sed -e '/Patricia/h' -e '/Margot/x' datafile
northwest NW Charles Main 3.0 .98 3 34
western WE Sharon Gray 5.3 .97 5 23
southwest SW Lewis Dalsass 2.7 .8 2 18

southern SO Suan Chin 5.1 .95 4 15
southeast SE Patricia Hemenway 4.0 .7 4 17
eastern EA TB Savage 4.4 .84 5 20
northeast NE AM Main Jr. 5.1 .94 3 13
southeast SE Patricia Hemenway 4.0 .7 4 17
central CT Ann Stephens 5.7 .94 5 13
解释：x命令将找到的行用保存缓存中的内容替换。

7. 用sed来编写命令表

sed 命令表(script)是文件里的一个sed命令列表。用-f选项来引用一个命令表文件。编辑sed命令表有特殊要求：命令末尾不能有任何为岁的空白符或者文本。如果命令不是自成一行，就必须用分号结束。在源代码chap4目录下有两个编辑好的命令表文件(sedding1和sedding2)可以参考。

下面是使用sed命令表的例子。

```
$ sed -f sedding1 datafile
 EMPLOYEE DATABASE
 ---------------------
 northwest NW Charles Main 3.0 .98 3 34
 western WE Sharon Gray 5.3 .97 5 23
 southwest SW Lewis Dalsass 2.7 .8 2 18
Lewis is the TOP Salesperson for April!!
Lewis is moving to the southern district next month.
CONGRATULATIONS!
 southern SO Suan Chin 5.1 .95 4 15
 southeast SE Patricia Hemenway 4.0 .7 4 17
 eastern EA TB Savage 4.4 .84 5 20
 northeast NE AM Main Jr. 5.1 .94 3 13
********************
     MARGOT HAS RETIRED
********************
```

8. Sed练习参考答案

练习内容参考databook文件

1〉 把Jon改成Jonathan

```
sed 's/Jon/Jonathan/' datebook
```

2〉 删除头3行

```
sed '1,3d' datebook
```

3〉 打印5—10行

```
sed -n '5,10p' datebook
```

4〉 删除包含Lane的行

```
sed '/Lane/d' datebook
```

5〉 打印所有生日是在Noverber到December之间的行

```
sed -n '/:1[12]\/p' datebook
```

6〉 把三个星添加到以Fred开头的行尾

```
sed '/^Fred/s/$/***/' datebook
```

7〉 用JOSE HAS RETIRED取代包含Jose的行

```
sed 's/^Jose[0-9]*[a-z]*[A-Z]* *.*$/JOSE HAS LEFT/' datebook
```

8〉 把Popeye的生日改成11/14/46

```
sed '/Popeye/s/:[0-9]*[0-9]*\/[0-9]*[0-9]*\/[0-9]*[0-9]*/:11\/14\/46/' datebook
```

9〉 删除所有空白行 sed '/^$/d' datebook 10〉 写一个sed命令表，将:

a. 在第1行之前插入标题PERSONNEL FILE

b. 删除以500结尾的工资
c. 打印文件内容，把姓和名颠倒
d. 在文件末尾添加THE END
答案放在chap04/a10文件中，内容如下：

```
# My first sed script by Wangzhh.
1i\
 PERSONNEL FILE
/500/d
s/\([A-Z][a-z]*\) \([A-Z][a-z]*\):/\2 \1:/
$a\
 THE END
```

[]

sed 实例讲解(上篇)
文章整理: 文章来源: 网络
前面的几句罗嗦话
在 UNIX/Linux 世界中有很多文本编辑器(如：vi、emacs 和 jed 以及很多其它工具)可供我们选择。我们都有自己熟悉并且喜爱的编辑器(和我们喜爱的组合键)。有了可信赖的编辑器，我们可以轻松处理任何数量与 UNIX 有关的管理或编程任务。
虽然交互式编辑器很棒，但却有其限制。尽管其交互式特性可以成为强项，但也有其不足之处。考虑一下需要对一组文件执行类似更改的情形。您可能会本能地运行自己所喜爱的编辑器，然后手工执行一组烦琐、重复和耗时的编辑任务。然而，有一种更好的方法。
开始说sed了，注意好好听
如果可以使编辑文件的过程自动化，以便用“ 批处理 ”方式编辑文件，甚至编写可以对现有文件进行复杂更改的脚本，那将太好了。幸运的是，对于这种情况，有一种更好的方法 -- 这种更好的方法称为 "sed"。
sed 是一种几乎包括在所有 UNIX/Linux 平台的轻量级流编辑器。sed 有许多很好的特性。首先，它相当小巧，通常要比您所喜爱的脚本语言小很多倍。其次，因为 sed 是一种流编辑器，所以，它可以对从如管道这样的标准输入接收的数据进行编辑。因此，无需将要编辑的数据存储在磁盘上的文件中。因为可以轻易将数据管道输出到 sed，所以，将 sed 用作强大的 shell 脚本中长而复杂的管道很容易。试一下用您所喜爱的编辑器去那样做。
GNU sed
对 Linux 用户来说幸运的是，最好的 sed 版本之一恰好是 GNU sed，其当前版本是 3.02。每一个 Linux 发行版都有（或至少应该有）GNU sed。GNU sed 之所以流行不仅因为可以自由分发其源代码，还因为它恰巧有许多对 POSIX sed 标准便利、省时的扩展。另外，GNU 没有 sed 早期专门版本的很多限制，如行长度限制 -- GNU 可以轻松处理任意长度的行。
最新的 GNU sed
在研究这篇文章之时我注意到：几个在线 sed 爱好者提到 GNU sed 3.02a。奇怪的是，在ftp.gnu.org上找不到sed 3.02a，所以，我只得在别处寻找。我在alpha.gnu.org 的 /pub/sed 中找到了它。于是我高兴地将其下载、编译然后安装，而几分钟后我发现最新的 sed 版本却是 3.02.80 -- 可在alpha.gnu.org 上 3.02a 源代码旁边找到其源代码。安装完 GNU sed 3.02.80 之后，我就完全准备好了。
alpha.gnu.org
alpha.gnu.org是新的和实验性 GNU 源代码的所在地。然而，您还会在那里发现许多优秀、稳定的源代码。出于某种原因，不是许多 GNU 开发人员忘记将稳定的源代码移至ftp.gnu.org，就是它们的 "beta" 期间格外长（2 年！）。例如，sed 3.02a 已有两年，甚至　3.02.80 也有一年。

正确的 sed
在本系列中，将使用 GNU sed 3.02.80。在即将出现的本系列后续文章中，某些（但非常少）最高级的示例将不能在 GNU sed 3.02 或 3.02a 中使用。如果您使用的不是 GNU sed，那么结果可能会不同。现在为什么不花些时间安装 GNU sed 3.02.80 呢？那样，不仅可以为本系列的余下部分作好准备，而且还可以使用可能是目前最好的 sed。
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，它对竺娴拿恳恍兄馗凑庑 ┏街琛U獠换岵　　涞觯　蛭　"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。但是，它做些什么呢？很明显，现在该复习规则表达式了。
规则表达式复习
可以使用规则表达式来表示可能会在文本中发现的模式。您在 shell 命令行中用过 '*' 字符吗？这种用法与规则表达式类似，但并不相同。下面是可以在规则表达式中使用的特殊字符：
字符　描述
% 与行首匹配
　与行末尾匹配
　与任一个字符匹配
　将与前一个字符的零或多个出现匹配
[ ] 与 [ ] 之内的所有字符匹配
感受规则表达式的最好方法可能是看几个示例。所有这些示例都将被 sed 作为合法地址接受，这些地址出现在命令的左边。下面是几个示例：
规则

表达式 描述
/./ 将与包含至少一个字符的任何行匹配
/../ 将与包含至少两个字符的任何行匹配
/^#/ 将与以 '#' 开始的任何行匹配
/^$/ 将与所有空行匹配
/}^/ 将与以 '}'（无空格）结束的任何行匹配
/} *^/ 将与以 '}' 后面跟有零或多个空格结束的任何行匹配
/[abc]/ 将与包含小写 'a'、'b' 或 'c' 的任何行匹配
/^[abc]/ 将与以 'a'、'b' 或 'c'开始的任何行匹配
在这些示例中，鼓励您尝试几个。花一些时间熟悉规则表达式，然后尝试几个自己创建的规则表达式。可以如下使用 regexp：
$ sed -e '/regexp/d' /path/to/my/test/file | more
这将导致 sed 删除任何匹配的行。然而，通过告诉 sed打印 regexp 匹配并删除不匹配的内容，而不是与之相反的方法，会更有利于熟悉规则表达式。可以用以下命令这样做：
$ 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[[]]*(/,/^}/p' sourcefile.c | more
该命令有两个规则表达式 '/main[[]]*(/' 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 '[[]]' 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 '[[]]'，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 '[[]]' 命令类（特别是在脚本中）会更清楚。
好，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 '}'。
[]

sed 实例讲解(中篇)
文章整理: 文章来源: 网络
sed 是很有用（但常被遗忘）的 UNIX 流编辑器。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。
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：

This is what I 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 '[]' 字符类来匹配空格。以下是可用字符类的相当完整的列表：

字符类　描述

[: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）.
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：

```
foo 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。
[]

sed 实例讲解(下篇)
文章整理: 文章来源: 网络
在上一篇sed介绍文章中，我们演示 sed 的工作原理，但是它们当中很少有示例能实际做特别有用的事。在这篇 sed 系列的最后文章中，我要改变那种方式，并使用 sed 来做实际的事。我将为您显示几个示例，它们不仅演示 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。
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！
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。

现在，要用 sed 执行一些强大的数据转换。

sed QIF 魔法

过去几个星期，我一直想买一份 Quicken来结算我的银行帐户。Quicken 是一个非常好的金融程序，当然会成功地完成这项工作。但是，经过考虑之后，我觉得自己可以轻易编写某个软件来结算我的支票簿。我想，毕竟，我是个软件开发人员！我开发了一个很好的小型支票簿结算程序（使用 awk），它通过分析包含我的所有交易的文本文件的语法来计算余额。略微调整之后，我将其改进，以便可以象 Quicken 那样跟踪不同的贷款和借款类别。但是，我还要添加一个特性。最近，我将帐户转移到一家有联机 Web 帐户界面的银行。有一天，我注意到，这家银行的 Web 站点允许以 Quicken 的 .QIF 格式下载我的帐户信息。我马上觉得，如果可以将该信息转换成文本格式，那就太棒了。

两种格式的故事

在查看 QIF 格式之前，先看一下我的 checkbook.txt 格式：

28 Aug 2000 food - - Y Supermarket 30.94 25 Aug 2000 watr - 103 Y Check 103 52.86

在我的文件中，所有字段都由一个或多个制表符分开，每个交易占据一行。日期之后的下一个字段列出支出类型（如果是收入项，则为 "-"）。第三个字段列出收入类型（如果是支出项，则为 "-"）。然后，是一个支票号字段（如果为空，则还是 "-"），一个交易完成字段（"Y" 或 "N"），一个注释和一个美元金额字段。现在，让我们看一下 QIF 格式。当用文本查看器查看下载的 QIF 文件时，它看起来如下：

!Type:Bank D08/28/2000 T-8.15 N PCHECKCARD SUPERMARKET ^ D08/28/2000 T-8.25 N PCHECKCARD PUNJAB RESTAURANT ^ D08/28/2000 T-17.17 N PCHECKCARD SUPERMARKET

浏览过文件之后，不难猜出其格式 -- 忽略第一行，其余的格式如下：

D<数据>

T<交易量>

N<支票号>

P<描述>

^ （这是字段分隔符）

开始处理

在处理象这样重要的 sed 项目时，不要气馁 -- sed 允许您将数据逐渐修改成最终形式。在进行当中，可以继续细化 sed 脚本，直到输出与预期的完全一样为止。无需在试第一次时就保证其完全正确。

要开始，首先创建一个名为 "qiftrans.sed" 的文件，然后开始修改数据：

```
1d /^^/d s/[[:cntrl:]]//g
```

第一个 '1d' 命令删除第一行，第二个命令从输出除去那些讨厌的 '^' 字符。最后一行除去文件中可能存在的任何控制字符。既然在处理外来文件格式，我想消除在中途遇到任何控制字符的风险。到目前为止，一切顺利。现在，要向该基本脚本中添加一些处理功能：

```
1d /^^/d s/[[:cntrl:]]//g /^D/ {
s/^D\(.*\)/\1\tOUTY\tINNY\t/
s/^01/Jan/        s/^02/Feb/
s/^03/Mar/        s/^04/Apr/
s/^05/May/        s/^06/Jun/
s/^07/Jul/        s/^08/Aug/
s/^09/Sep/        s/^10/Oct/
s/^11/Nov/        s/^12/Dec/
s:^\(.*\)/\(.*\)/\(.*\):\2 \1 \3:  }
```

首先，添加一个 '/^D/' 地址，以便 sed 只在遇到 QIF 数据字段的第一个字符 'D' 时才开始处理。当 sed 将这样一行读入其模式空间时，将按顺序执行花括号中的所有命令。

花括号中的第一个命令将把如下行：

```
D08/28/2000
```

变换成：

```
08/28/2000 OUTY INNY
```

当然，现在的格式还不完美，但没关系。我们将在进行过程中逐渐细化模式空间的内容。后面 12 行的最后效果是将数据变换成三个字母的格式，最后一行从数据中除去三个斜杠。最后得到这一行：

```
Aug 28 2000 OUTY INNY
```

OUTY 和 INNY 字段是占位符，以后将被替换。现在还不能确定它们，因为如果美元金额为负，将把 OUTY 和 INNY 设置成 "misc" 和 "-"，但是，如果美元金额为正，将分别把它们更改成 "-" 和 "inco"。既然还没有读入美元金额，所以，需要暂时使用占位符。

细化

现在进一步细化：

```
1d  /^^/d s/[[:cntrl:]]//g  /^D/ {
s/^D\(.*\)/\1\tOUTY\tINNY\t/
s/^01/Jan/        s/^02/Feb/
      s/^03/Mar/        s/^04/Apr/
  s/^05/May/        s/^06/Jun/
  s/^07/Jul/        s/^08/Aug/
  s/^09/Sep/        s/^10/Oct/
  s/^11/Nov/        s/^12/Dec/
  s:^\(.*\)/\(.*\)/\(.*\):\2 \1 \3:
  N        N        N
  s/\nT\(.*\)\nN\(.*\)\nP\(.*\)/NUM\2NUM\t\tY\t\t\3\tAMT\1AMT/
  s/NUMNUM/-/        s/NUM\([0-9]*\)NUM/\1/
  s/\([0-9]\),/\1/  }
```

后七行有些复杂，所以将详细讨论它们。首先，连续使用三个 'N' 命令。'N' 命令告诉 sed 将 下一行读入输入中，然后将其附加到当前模式空间。这三个 'N' 命令导致将下三行附加到当前模式空间缓冲区，现在这一行看起来如下：

```
28 Aug 2000 OUTY INNY \nT-8.15\nN\nPCHECKCARD SUPERMARKET
```

sed 的模式空间变得很难看 -- 需要除去额外的新行，并执行某些附加的格式化。要这样做，将使用替代命令。要匹配的模

式为：

'\nT.*\nN.*\nP.*'

这将与后面依次跟有 'T'、零或多个字符、新行、'N'、任何数量的字符、新行、'P'、以及任何数量字符的新行匹配。呀！这个规则表达式将与刚刚附加到模式空间的三行的全部内容匹配。但我们要重新格式化该区域，而不是整个替换它。美元金额、支票号（如果有的话）和描述需要出现在替换字符串中。要这样做，我们用带有反斜杠的圆括号括起那些“ 感兴趣部分 ”，以便可以在替换字符串中引用它们（使用 '\1'、'\2\ 和 '\3' 来告诉 sed 将它们插入到何处）。以下是最后的命令：

s/\nT\(.*\)\nN\(.*\)\nP\(.*\)/NUM\2NUM\t\tY\t\t\3\tAMT\1AMT/

该命令将我们的行变换成：

28 Aug 2000 OUTY INNY NUMNUM Y CHECKCARD SUPERMARKET AMT-8.15AMT

虽然该行正变得好一些，但是，有几件事一看就有点...啊...有趣。首先是那个愚蠢的 "NUMNUM" 字符串 -- 其目的何在？如果查看 sed 脚本的后两行，就会发现其目的，后两行将把 "NUMNUM" 替换成 "-"，而把 "NUM"<number>"NUM" 替换成 <number>。如您所见，用愚蠢的标记括起支票号允许我们在该字段为空时方便地插入一个 "-"。

结束尝试

最后一行除去数字后的逗号。它把如 "3,231.00" 这样的美元金额转换成我使用的格式 "3231.00"。现在，让我们看一下最终脚本：

最终的“ QIF 到文本 ”脚本

```
1d /^^/d s/[[:cntrl:]]//g /^D/ { s/^D\(.*\)/\1\tOUTY\tINNY\t/
s/^01/Jan/ s/^02/Feb/ s/^03/Mar/ s/^04/Apr/ s/^05/May/
s/^06/Jun/ s/^07/Jul/ s/^08/Aug/ s/^09/Sep/ s/^10/Oct/
s/^11/Nov/ s/^12/Dec/ s:^\(.*\)/\(.*\)/\(.*\):\2 \1 \3:
N N N s/\nT\(.*\)\nN\(.*\)\nP\(.*\)/NUM\2NUM\t\tY\t\t\3\tAMT\1AMT/
s/NUMNUM/-/ s/NUM\([0-9]*\)NUM/\1/ s/\([0-9]\),/\1/
/AMT-[0-9]*.[0-9]*AMT/b fixnegs
s/AMT\(.*\)AMT/\1/ s/OUTY/-/ s/INNY/inco/
b done :fixnegs s/AMT-\(.*\)AMT/\1/ s/OUTY/misc/
s/INNY/-/ :done }
```

附加的十一行使用替代和一些分支功能来美化输出。首先看一下这行：

```
/AMT-[0-9]*.[0-9]*AMT/b fixnegs
```

该行包含一个格式为 "/regexp/b label" 的分支命令。如果模式空间与规则表达式匹配，sed 将分支到 fixnegs 标号。您应该可以轻易找到该标号，它在代码中为 ":fixnegs"。如果规则表达式不匹配，则以常规方式继续处理下一个命令。

既然您理解该命令本身的工作原理，让我们看一下分支。如果看一下分支规则表达式，将看到它与后面依次跟有 '-'、任意数量的数字、一个 '.'、任意数量的数字和 'AMT' 的字符串 'AMT' 匹配。就象我确信您已猜到一样，该规则表达式专门处理负的美元金额。在这之前，用 'ATM' 括起美元金额，以便以后可以轻易找到它。因为规则表达式只与以 '-' 开始的美元金额匹配，所以，该分支只在恰巧处理借款时才发生。如果正处理贷款，应该将 OUTY 设置成 'misc'，将 INNY 设置成 '-'，并且应该除去贷款数量前面的负号。如果跟踪代码的流程，将看到实际情况正是这样。如果不执行分支，则用 '-' 替换 OUTY，用 'inco' 替换 INNY。完成了！现在输出行是完美的：

28 Aug 2000 misc - - Y CHECKCARD SUPERMARKET -8.15

别犯糊涂

如您所见，只要循序渐进地解决问题，使用 sed 转换数据就没有那么难。不要试图使用一个 sed 命令或一下子解决所有问题。相反，要朝着目标逐步进行，并不断改进 sed 脚本，直到其输出正如您希望那样为止。sed 有许多功能，希望您已非常熟悉其内部工作原理并继续努力以进一步掌握它！

[]

sort实例应用

文章整理: 文章来源: 网络

通过使用 sort 和 tsort，而不是采取使用 Perl 或 Awk 的较复杂的解决方案，可以节省时间，同时还能避免令人头疼的问题。Jacek Artymiak 将向您说明如何做到这一点。

尽管可以用 Perl 或 Awk 编写高级排序应用程序，但并非总是有此必要，而且这样的工作也常常令人感到头疼。使用 sort 命令，您同样可以实现您所需的大多数功能，而且更容易，它可以对多个文件中的行进行排序、合并文件甚至可以查看

欠裼斜匾 6 运　墙　信判颉 D　梢灾付ㄋ判蚣　ㄐ糜诒冉系男兄械囊徊糠郑　　部刹恢付ǎ　笠恢智榭鲅　sort 就比较所有行。

因此，如果您想对密码文件进行排序，就可以使用下列命令（请注意，您不能将输出直接发送到输入文件，因为这会破坏输入文件。这就是为何您需要将它发送到临时文件中，然后将该文件重命名为 /etc/passwd 的原因，如下所示）。

1、清单 1. 简单排序

$ su -

sort /etc/passwd > /etc/passwd-new

mv /etc/passwd-new /etc/passwd

2、有关 sort 和 tsort 的更多信息

通过打开有关排序操作的 GNU 手册页来学习手册页中的内容，或者通过在命令行中输入 man sort 或 man tsort 在新的终端窗口的手册页或信息页中查看这些选项。

如果您想倒转排序的次序，则应当使用 -r 选项。您还可以用 -u 选项来禁止打印相同的行。

3、sort 的一个非常实用的特性是它用字段键进行排序的能力。字段是一个文本字符串，通过某个字符与其它字段分隔开。例如，/etc/passwd 中的字段是用冒号（:）分隔的。因此，如果愿意的话，您可以按照用户标识、组标识、注释字段、主目录或 shell 对 /etc/passwd 进行排序。要做到这一点，请使用 -t 选项，其后跟着用作分隔符的字符，接着是用作排序键的字段编号，再跟作为键的最后一个字段的编号；

例如，

sort -t : -k 5,5 /etc/passwd

按照注释字段对密码文件进行排序，该字段中存储了完整的用户名（如“ John Smith ”）。

而

sort -t : -k 3,4 /etc/passwd

同时使用用户标识和组标识对同一个文件进行排序。如果您省略了第二个数字，那么 sort 会假定键是从给定的字段开始，一直到每一行的末尾。动手试一试，并观察其中的区别（当数字排序看上去有错时，请添加 -g 选项）。

还要注意的是，空白过渡是缺省的分隔符，因此，如果字段已经用空白字符分隔了，那么您可以省略分隔符，只使用 -t（另注：字段的编号是从 1 开始的）。

5、为了更好地进行控制，您可以使用键和偏移量。偏移量是用点与键相分隔的，比如在 -k 1.3,5.7 中，表示排序键应当从第 1 个字段的第 3 个字符开始，到第 5 个字段的第 7 个字符结束（偏移量也是从 1 开始编号的）。何时会用得着偏移量呢？嗯，我时常用它来对 Apache 日志进行排序；键和偏移量表示法让我跳过了日期字段。

6、另一个要关注的选项是 -b，它告知 sort 忽略空白字符（空格、跳格等等）并将行中的第一个非空白字符当做是排序键的开始。还有，如果您使用该选项，那么将从第一个非空白字符开始计算偏移量（当字段分隔符不是空白字符，且字段可能包含以空白字符开头的字符串时，这非常有用）。

可以用下面这些选项来进一步修改排序算法：

-d（只将字母、数字和空白用作排序键）、

-f（关闭大小写区分，认为小写和大写字符是一样的）、

-i（忽略非打印的 ASCII 字符）、

-M（使用三个字母的月份名称缩写：JAN、FEB、MAR … 来对行进行排序）和

-n（只用数字、- 和逗号或另外一个千位分隔符对行进行排序）。

这些选项以及 -b 和 -r 选项可以用作键编号的一部分，

在这种情况下，它们只适用于该键而非全局，其作用就跟在键定义外使用它时一样。

以键编号的用法为例，请考虑：

sort -t: -k 4g,4 -k 3gr,3 /etc/passwd

这条命令将按照组标识对 passwd 文件进行排序，而在组内按照用户标识进行逆向排序。

7、如果您所使用的键不能用来确定哪一行是在先，那么它也可以解决这类平局问题。增加一个解决平局问题的提示，请添加另一个 -k 选项，让它跟在字段和（可选的）偏移量后面，使用与前面用于定义键相同的表示法；

例如，

sort -k 3.4,4.5 -k 7.3,9.4 /etc/passwd

对行进行排序时，使用从第 3 个键的第 4 个字符开始到第 4 个键的第 5 个字符结束的键，然后再采用从第 7 个字段的第 3 个字符到第　9 个字段的第 4 个字符结束的键来解决上述难题。

8、最后一组选项处理输入、输出和临时文件。例如，-c 选项，当它用于 sort -c < file 中时，它检查输入文件是否已进行了排序（您也可以使用其它选项），如果已进行了排序，则报告一个错误。这样，在处理可能需要花很长时间进行排序的大型文件之前，可以很方便地对其进行检查。当您将 -u 选项和 -c 选项一起使用时，会被解释为一个请求：检查输入文件中不存在两个相同的行。
9、当您处理大型文件时还有一个很重要的 -T 选项，它用于为临时文件（这些临时文件在 sort 完成工作之后会被除去）指定其它目录，而不是缺省的 /tmp 目录。
10、您可以使用 sort 来同时处理多个文件，这样做的方式基本上有两种：首先可以使用 cat 来并置它们，如下所示：
cat file1 file2 file3 | sort > outfile
或者，可以使用下面这个命令：
sort -m file1 file2 file3 > outfile
第二种情况有个条件：在将所有输入文件一起进行 sort -m 之前，每个文件都必须经过排序。这看起来似乎是个不必要的负担，但事实上这加快了工作速度并节约了宝贵的系统资源。对了，别忘了 -m 选项。在这里您可以使用 -u 选项来禁止打印相同的行。
11、如果需要某种更深奥的排序方法，您可能要查看 tsort 命令，该命令对文件执行拓扑排序。拓扑排序和标准 sort 之间的差别如清单 2 所示。
清单 2. 拓扑排序和标准排序之间的差别
$ cat happybirthday.txt
Happy Birthday to You!
Happy Birthday to You!
Happy Birthday Dear Tux!
Happy Birthday to You!
$ sort happybirthday.txt
Happy Birthday Dear Tux!
Happy Birthday to You!
Happy Birthday to You!
Happy Birthday to You!
$ tsort happybirthday.txt
Dear
Happy
to
Tux!
Birthday
You!
当然，对于 tsort 的使用来说，这并非一个非常有用的演示，只是举例说明了这两个命令输出的不同。
tsort 通常用于解决一种逻辑问题，即必须通过观察到的部分次序预测出整个次序；例如（来自 tsort 信息页中）：
tsort <<EOF a b c d e f b c d e EOF
会产生这样的输出

a
b
c
d
e
f

tsort的没试过，最后一个我不行耶，不知道咋回事
[]

高级Bash脚本编程指南(一)
文章整理: 文章来源: 网络

SHELL十三问

高级Bash脚本编程指南(一)

目录

++++

SHELL十三问

14-1. 愚蠢的脚本策略
14-2. 从循环的输出中产生一个变量
14-3. 找anagram(回文构词法, 可以将一个有意义的单词, 变换为1个或多个有意义的单词, 但是还是原来的子母集合)
16-1. 使用exec重定向标准输入
16-2. 使用exec来重定向stdout
16-3. 使用exec在同一脚本中重定向stdin和stdout
16-4. 避免子shell
16-5. while循环的重定向
16-6. 另一种while循环的重定向
16-7. until循环重定向
16-8. for循环重定向
16-9. for循环重定向 loop (将标准输入和标准输出都重定向了)
16-10. 重定向if/then测试结构
16-11. 用于上面例子的"names.data"数据文件
16-12. 记录日志事件
17-1. 广播: 发送消息给每个登录上的用户
17-2. 仿造文件: 创建一个两行的仿造文件
17-3. 使用cat的多行消息
17-4. 带有抑制tab功能的多行消息
17-5. 使用参数替换的here document
17-6. 上传一个文件对到"Sunsite"的incoming目录
17-7. 关闭参数替换
17-8. 一个产生另外一个脚本的脚本
17-9. Here documents与函数
17-10. "匿名" here Document
17-11. 注释掉一段代码块
17-12. 一个自文档化(self-documenting)的脚本
17-13. 在一个文件的开头添加文本
20-1. 子shell中的变量作用域
20-2. 列出用户的配置文件
20-3. 在子shell里进行串行处理
21-1. 在受限的情况下运行脚本
23-1. 简单函数
23-2. 带着参数的函数
23-3. 函数和被传给脚本的命令行参数
23-4. 传递间接引用给函数
23-5. 解除传递给函数的参数引用
23-6. 再次尝试解除传递给函数的参数引用
23-7. 两个数中的最大者
23-8. 把数字转化成罗马数字
23-9. 测试函数最大的返回值
23-10. 比较两个大整数
23-11. 用户名的真实名
23-12. 局部变量的可见范围
23-13. 用局部变量来递归
23-14. 汉诺塔
24-1. 脚本中的别名
24-2. unalias: 设置和删除别名

SHELL十三问

25-1. 使用"与列表(and list)"来测试命令行参数
25-2. 用"与列表"的另一个命令行参数测试
25-3. "或列表"和"与列表"的结合使用
26-1. 简单的数组用法
26-2. 格式化一首诗
26-3. 多种数组操作
26-4. 用于数组的字符串操作符
26-5. 将脚本的内容传给数组
26-6. 一些数组专用的工具
26-7. 关于空数组和空数组元素
26-8. 初始化数组
26-9. 复制和连接数组
26-10. 关于连接数组的更多信息
26-11. 一位老朋友: 冒泡排序
26-12. 内嵌数组和间接引用
26-13. 复杂数组应用: 埃拉托色尼素数筛子
26-14. 模拟下推的堆栈
26-15. 复杂的数组应用: 列出一种怪异的数学序列
26-16. 模拟二维数组,并使它倾斜
27-1. 利用/dev/tcp 来检修故障
27-2. 搜索与一个PID相关的进程
27-3. 网络连接状态
28-1. 隐藏cookie而不再使用
28-2. 用/dev/zero创建一个交换临时文件
28-3. 创建ramdisk
29-1. 一个错误的脚本
29-2. 丢失关键字(keyword)
29-3. 另一个错误脚本
29-4. 用"assert"测试条件
29-5. 捕捉 exit
29-6. 在Control-C后清除垃圾
29-7. 跟踪变量
29-8. 运行多进程 (在多处理器的机器里)
31-1. 数字和字符串比较是不相等同的
31-2. 子SHELL缺陷
31-3. 把echo的输出用管道输送给read命令
33-1. shell 包装
33-2. 稍微复杂一些的shell包装
33-3. 写到日志文件的shell包装
33-4. 包装awk的脚本
33-5. 另一个包装awk的脚本
33-6. 把Perl嵌入Bash脚本
33-7. Bash 和 Perl 脚本联合使用
33-8. 递归调用自己本身的(无用)脚本
33-9. 递归调用自己本身的(有用)脚本
33-10. 另一个递归调用自己本身的(有用)脚本
33-11. 一个 "彩色的" 地址资料库
33-12. 画盒子

SHELL十三问

第一部分 热身

++++++++++++++++

shell是一个命令解释器.是介于操作系统kernel与用户之间的一个绝缘层.准确地说,它也是一一种强力的计算机语言.一个shell程序,被称为一个脚本,是一种很容易使用的工具,它可以通过将系统调用,公共程序,工具,和编译过的二进制程序粘合在一起来建立应用.事实上,所有的UNIX命令和工具再加上公共程序,对于shell脚本来说,都是可调用的.如果这些你还觉得不够,那么shell内建命令,比如test与循环结构,也会给脚本添加强力的支持和增加灵活性.Shell脚本对于管理系统任务和其它的重复工作的例程来说,表现的非常好,根本不需要那些华而不实的成熟紧凑的程序语言.

第1章 为什么使用shell编程

==========

没有程序语言是完美的.甚至没有一个唯一最好的语言,只有对于特定目的,比较适合和不适合的程序语言.

Herbert Mayer

对于任何想适当精通一些系统管理知识的人来说,掌握shell脚本知识都是最基本的,即使这些人可能并不打算真正的编写一些脚本.想一下Linux机器的启动过程,在这个过程中,必将运行/etc/rc.d目录下的脚本来存储系统配置和建立服务.详细的理解这些启动脚本对于分析系统的行为是非常重要的,并且有时候可能必须修改它.

学习如何编写shell脚本并不是一件很困难的事,因为脚本可以分为很小的块,并且相对于shell特性的操作和选项[1]部分,只需要学习很小的一部分就可以了.语法是简单并且直观的,编写脚本很像是在命令行上把一些相关命令和工具连接起来,并且只有很少的一部分规则需要学习.绝大部分脚本第一次就可以正常的工作,而且即使调试一个长一些的脚本也是很直观的.

一个shell脚本是一个类似于小吃店的(quick and dirty)方法,在你使用原型设计一个复杂的应用的时候.在工程开发的第一阶段,即使从功能中取得很有限的一个子集放到shell脚本中来完成往往都是非常有用的.使用这种方法,程序的结果可以被测试和尝试运行,并且在处理使用诸如C/C++,Java或者Perl语言编写的最终代码前,主要的缺陷和陷阱往往就被发现了.

Shell脚本遵循典型的UNIX哲学,就是把大的复杂的工程分成小规模的子任务,并且把这些部件和工具组合起来.许多人认为这种办法更好一些,至少这种办法比使用那种高\大\全的语言更美,更愉悦,更适合解决问题.比如Perl就是这种能干任何事能适合任何人的语言,但是代价就是你需要强迫自己使用这种语言来思考解决问题的办法.

什么时候不使用Shell脚本

资源密集型的任务,尤其在需要考虑效率时(比如,排序,hash等等)

需要处理大任务的数学操作,尤其是浮点运算,精确运算,或者复杂的算术运算

(这种情况一般使用C++或FORTRAN来处理)

有跨平台移植需求(一般使用C或Java)

复杂的应用,在必须使用结构化编程的时候(需要变量的类型检查,函数原型,等等)

对于影响系统全局性的关键任务应用。

对于安全有很高要求的任务,比如你需要一个健壮的系统来防止入侵,破解,恶意破坏等等.

项目由连串的依赖的各个部分组成。

需要大规模的文件操作

需要多维数组的支持

需要数据结构的支持,比如链表或数等数据结构

需要产生或操作图形化界面GUI

需要直接操作系统硬件

需要I/O或socket接口

需要使用库或者遗留下来的老代码的接口

私人的,闭源的应用(shell脚本把代码就放在文本文件中,全世界都能看到)

如果你的应用符合上边的任意一条,那么就考虑一下更强大的语言吧--或许是Perl,Tcl,Python, Ruby -- 或者是更高层次的编译语言比如C/C++,或者是Java.即使如此,你会发现,使用shell来原型开发你的应用,在开发步骤中也是非常有用的.

我们将开始使用Bash,Bash是"Bourne-Again shell"首字母的缩写,也是Stephen Bourne的经典的Bourne shell的一个双关语,(译者:说实话,我一直搞不清这个双关语是什么意思,为什么叫"Bourn-Again shell",这其中应该有个什么典故吧,哪位好心,告诉我一下^^).Bash已经成为了所有UNIX中shell脚本的事实上的标准了.同时这本书也覆盖了绝大部分的其他一些shell的原则,比如Korn Shell,Bash从ksh中继承了一部分特性,[2]C Shell和它的变种.(注意:C Shell编程是不被推荐的,因为一些特定的内在问题,Tom Christiansen在1993年10月指出了这个问题请在http://www.etext.org/Quartz/computer/unix/csh.harmful.gz中查看具体内容.)

接下来是脚本的一些说明.在展示shell不同的特征之前,它可以减轻一些阅读书中例子的负担.本书中的例子脚本,都在尽可能的范围内进行了测试,并且其中的一些将使用在真实的生活中.读者可以运行这些例子脚本(使用scriptname.sh或者scriptname.bash的形式),[3]并给这些脚本执行权限(chmod u+rx scriptname),然后执行它们,看看发生了什么.如果存档的脚本不可用,那么就从本书的HTML,pdf或者text的发行版本中把它们拷贝粘贴出来.考虑到这些脚本中的内容在我们还没解释它之前就被列在这里,可能会影响读者的理解,这就需要读者暂时忽略这些内容.

除非特别注明,本书作者编写了本书中的绝大部分例子脚本.

注意事项:

[1] 这些在builtins章节被引用,这些是shell的内部特征.

[2] ksh88的许多特性,甚至是一些ksh93的特性都被合并到Bash中了.

[3] 根据惯例,用户编写的Bourne shell脚本应该在脚本的名字后边加上.sh扩展名.
一些系统脚本,比如那些在/etc/rc.d中的脚本,则不遵循这种命名习惯.

第2章 带着一个Sha-Bang出发(Sha-Bang指的是#!)

在一个最简单的例子中,一个shell脚本其实就是将一堆系统命令列在一个文件中.它的最基本的用处就是,在你每次输入这些特定顺序的命令时可以少敲一些字.

Example 2-1 清除:清除/var/log下的log文件

```
################################Start Script#########################################
1 # Cleanup
2 # 当然要使用root身份来运行这个脚本
3
4 cd /var/log
5 cat /dev/null > messages
6 cat /dev/null > wtmp
7 echo "Logs cleaned up."
################################End Script###########################################
```

这根本就没什么稀奇的, 只不过是命令的堆积, 来让从console或者xterm中一个一个的输入命令更方便一些.好处就是把所有命令都放在一个脚本中,不用每次都敲它们.这样的话,对于特定的应用来说,这个脚本就很容易被修改或定制.

Example 2-2 清除:一个改良的清除脚本

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # 一个Bash脚本的正确的开头部分.
 3
 4 # Cleanup, 版本 2
 5
```

```
 6 # 当然要使用root身份来运行.
 7 # 在此处插入代码,来打印错误消息,并且在不是root身份的时候退出.
 8
 9 LOG_DIR=/var/log
10 # 如果使用变量,当然比把代码写死的好.
11 cd $LOG_DIR
12
13 cat /dev/null > messages
14 cat /dev/null > wtmp
15
16
17 echo "Logs cleaned up."
18
19 exit # 这个命令是一种正确并且合适的退出脚本的方法.
```

##################################End Script##

现在,让我们看一下一个真正意义的脚本.而且我们可以走得更远...

Example 2-3. cleanup:一个增强的和广义的删除logfile的脚本

##################################Start Script##

```
 1 #!/bin/bash
 2 # 清除, 版本 3
 3
 4 #  Warning:
 5 #  -------
 6 #  这个脚本有好多特征,这些特征是在后边章节进行解释的,大概是进行到本书的一半的
 7 #  时候,
 8 #  你就会觉得它没有什么神秘的了.
 9 #
10
11
12
13 LOG_DIR=/var/log
14 ROOT_UID=0     # $UID为0的时候,用户才具有根用户的权限
15 LINES=50       # 默认的保存行数
16 E_XCD=66       # 不能修改目录?
17 E_NOTROOT=67   # 非根用户将以error退出
18
19
20 # 当然要使用根用户来运行
21 if [ "$UID" -ne "$ROOT_UID" ]
22 then
23   echo "Must be root to run this script."
24   exit $E_NOTROOT
25 fi
26
27 if [ -n "$1" ]
28 # 测试是否有命令行参数(非空).
29 then
30   lines=$1
```

```
31 else
32   lines=$LINES # 默认,如果不在命令行中指定
33 fi
34
35
36 #  Stephane Chazelas 建议使用下边
37 #+ 的更好方法来检测命令行参数.
38 #+ 但对于这章来说还是有点超前.
39 #
40 #    E_WRONGARGS=65  # 非数值参数(错误的参数格式)
41 #
42 #    case "$1" in
43 #    ""      ) lines=50;;
44 #    *[!0-9]*) echo "Usage: `basename $0` file-to-cleanup"; exit $E_WRONGARGS;;
45 #    *       ) lines=$1;;
46 #    esac
47 #
48 #* 直到"Loops"的章节才会对上边的内容进行详细的描述.
49
50
51 cd $LOG_DIR
52
53 if [ `pwd` != "$LOG_DIR" ]  # 或者 if[ "$PWD" != "$LOG_DIR" ]
54                             # 不在 /var/log中?
55 then
56   echo "Can't change to $LOG_DIR."
57   exit $E_XCD
58 fi  # 在处理log file之前,再确认一遍当前目录是否正确.
59
60 # 更有效率的做法是
61 #
62 # cd /var/log || {
63 #   echo "Cannot change to necessary directory." >&2
64 #   exit $E_XCD;
65 # }
66
67
68
69
70 tail -$lines messages > mesg.temp # 保存log file消息的最后部分.
71 mv mesg.temp messages             # 变为新的log目录.
72
73
74 # cat /dev/null > messages
75 #* 不再需要了,使用上边的方法更安全.
76
77 cat /dev/null > wtmp  #  ': > wtmp' 和 '> wtmp'具有相同的作用
78 echo "Logs cleaned up."
```

```
79
80 exit 0
81 #  退出之前返回0,返回0表示成功.
82 #
```

################################End Script###

因为你可能希望将系统log全部消灭,这个版本留下了log消息最后的部分.你将不断地找到新的方法来完善这个脚本,并提高效率.

要注意,在每个脚本的开头都使用"#!",这意味着告诉你的系统这个文件的执行需要指定一个解释器.#!实际上是一个2字节[1]的魔法数字,这是指定一个文件类型的特殊标记, 换句话说, 在这种情况下,指的就是一个可执行的脚本(键入man magic来获得关于这个迷人话题的更多详细信息).在#!之后接着是一个路径名.这个路径名指定了一个解释脚本中命令的程序,这个程序可以是shell,程序语言或者是任意一个通用程序.这个指定的程序从头开始解释并且执行脚本中的命令(从#!行下边的一行开始),忽略注释.[2]

如:

```
1 #!/bin/sh
2 #!/bin/bash
3 #!/usr/bin/perl
4 #!/usr/bin/tcl
5 #!/bin/sed -f
6 #!/usr/awk -f
```

上边每一个脚本头的行都指定了一个不同的命令解释器,如果是/bin/sh,那么就是默认shell (在Linux系统中默认是Bash).[3]使用#!/bin/sh,在大多数商业发行的UNIX上,默认是Bourne shell,这将让你的脚本可以正常的运行在非Linux机器上,虽然这将会牺牲Bash一些独特的特征. 脚本将与POSIX[4] 的sh标准相一致.

注意: #! 后边给出的路径名必须是正确的,否则将会出现一个错误消息,通常是 "Command not found",这将是你运行这个脚本时所得到的唯一结果.

当然"#!"也可以被忽略,不过这样你的脚本文件就只能是一些命令的集合,不能够使用shell内建的指令了,如果不能使用变量的话,当然这也就失去了脚本编程的意义了.

注意:这个例子鼓励你使用模块化的方式来编写脚本,平时也要注意收集一些零碎的代码, 这些零碎的代码可能用在你将来编写的脚本中.这样你就可以通过这些代码片段来构造一个较大的工程用例. 以下边脚本作为序,来测试脚本被调用的参数是否正确.

################################Start Script###

```
 1 E_WRONG_ARGS=65
 2 script_parameters="-a -h -m -z"
 3 #                    -a = all, -h = help, 等等.
 4
 5 if [ $# -ne $Number_of_expected_args ]
 6 then
 7   echo "Usage: `basename $0` $script_parameters"
 8   # `basename $0`是这个脚本的文件名
 9   exit $E_WRONG_ARGS
10 fi
```

################################End Script###

大多数情况下,你需要编写一个脚本来执行一个特定的任务,在本章中第一个脚本就是一个这样的例子, 然后你会修改它来完成一个不同的,但比较相似的任务.用变量来代替写死的常量,就是一个好方法,将重复的代码放到一个函数中,也是一种好习惯.

2.1 调用一个脚本

编写完脚本之后,你可以使用sh scriptname,[5]或者bash scriptname来调用它.
(不推荐使用sh <scriptname,因为这禁用了脚本从stdin中读数据的功能.)
更方便的方法是让脚本本身就具有可执行权限,通过chmod命令可以修改.
比如:
chmod 555 scriptname (允许任何人都具有 可读和执行权限) [6]
或:
chmod +rx scriptname (允许任何人都具有 可读和执行权限)
chmod u+rx scriptname (只给脚本的所有者 可读和执行权限)
既然脚本已经具有了可执行权限,现在你可以使用./scriptname.[7]来测试它了.如果这个脚本以一个"#!"行开头,那么脚本将会调用合适的命令解释器来运行.
最后一步,在脚本被测试和debug之后,你可能想把它移动到/usr/local/bin(当然是以root身份),来让你的脚本对所有用户都有用.这样用户就可以直接敲脚本名字来运行了.
注意事项:
[1] 那些具有UNIX味道的脚本(基于4.2BSD)需要一个4字节的魔法数字,在#!后边需要一个空格#! /bin/sh.
[2] 脚本中的#!行的最重要的任务就是命令解释器(sh或者bash).因为这行是以#开始的,当命令解释器执行这个脚本的时候,会把它作为一个注释行.当然,在这之前,这行语句已经完成了它的任务,就是调用命令解释器.
如果在脚本的里边还有一个#!行,那么bash将把它认为是一个一般的注释行.

```
1 #!/bin/bash
2
3 echo "Part 1 of script."
4 a=1
5
6 #!/bin/bash
7 # 这将不会开始一个新脚本.
8
9 echo "Part 2 of script."
10 echo $a  # Value of $a stays at 1.
```

[3] 这里可以玩一些小技巧.

```
1 #!/bin/rm
2 # 自删除脚本.
3
4 # 当你运行这个脚本时,基本上什么都不会发生...除非这个文件消失不见.
5
6 WHATEVER=65
7
8 echo "This line will never print (betcha!)."
9
10 exit $WHATEVER  # 没关系,脚本是不会在这退出的.
```

当然,你还可以试试在一个README文件的开头加上#!/bin/more,并让它具有执行权限.结果将是文档自动列出自己的内容.(一个使用cat命令的here document可能是一个更好的选则,--见Example 17-3).
[4] 可移植的操作系统接口,标准化类UNIX操作系统的一种尝试.POSIX规范可以在http://www.opengroup.org/onlinepubs/007904975/toc.htm中查阅.
[5] 小心:使用sh scriptname来调用脚本的时候将会关闭一些Bash特定的扩展,脚本可能因此而调用失败.
[6] 脚本需要读和执行权限,因为shell需要读这个脚本.

[7] 为什么不直接使用scriptname来调用脚本?如果你当前的目录下($PWD)正好有你想要
执行的脚本,为什么它运行不了呢?失败的原因是,出于安全考虑,当前目录并没有被
加在用户的$PATH变量中.因此,在当前目录下调用脚本必须使用./scriptname这种
形式.

2.2 初步的练习

---

1. 系统管理员经常会为了自动化一些常用的任务而编写脚本.举出几个这种有用的脚本的实例.
2. 编写一个脚本,显示时间和日期,列出所有的登录用户,显示系统的更新时间.然后这个脚本
将会把这些内容保存到一个log file中.

第二部分 基本

++++++++++++++++

第3章 特殊字符

==========

注释,行首以#开头为注释(#!是个例外).

```
1 # This line is a comment.
```

注释也可以存在于本行命令的后边.

```
1 echo "A comment will follow." # 注释在这里
2 #                           ^ 注意#前边的空白
```

注释也可以在本行空白的后边.

```
1  # A tab precedes this comment.
```

注意:命令是不能跟在同一行上注释的后边的,没有办法,在同一行上,注释的后边想
要再使用命令,只能另起一行.

当然,在echo命令中被转义的#是不能作为注释的.

同样的,#也可以出现在特定的参数替换结构中或者是数字常量表达式中.

```
1 echo "The # here does not begin a comment."
2 echo 'The # here does not begin a comment.'
3 echo The \# here does not begin a comment.
4 echo The # 这里开始一个注释
5
6 echo ${PATH#*:}       # 参数替换,不是一个注释
7 echo $(( 2#101011 ))  # 数制转换,不是一个注释
8
9 # Thanks, S.C.
```

标准的引用和转义字符('"\)可以用来转义#

; 命令分隔符,可以用来在一行中来写多个命令.

```
1 echo hello; echo there
2
3
4 if [ -x "$filename" ]; then    # 注意:"if"和"then"需要分隔
5                               # 为啥?
6   echo "File $filename exists."; cp $filename $filename.bak
7 else
8   echo "File $filename not found."; touch $filename
9 fi; echo "File test complete."
```

有时候需要转义

;; 终止"case"选项.

```
1 case "$variable" in
2 abc)  echo "\$variable = abc" ;;
```

3 xyz) echo "\$variable = xyz" ;;
4 esac

. .命令等价于source命令(见Example 11-20).这是一个bash的内建命令.

. .作为文件名的一部分.如果作为文件名的前缀的话,那么这个文件将成为隐藏文件.
将不被ls命令列出.

```
bash$ touch .hidden-file
bash$ ls -l
total 10
-rw-r--r--    1 bozo      4034 Jul 18 22:04 data1.addressbook
-rw-r--r--    1 bozo      4602 May 25 13:58 data1.addressbook.bak
-rw-r--r--    1 bozo       877 Dec 17  2000 employment.addressbook


bash$ ls -al
total 14
drwxrwxr-x    2 bozo  bozo      1024 Aug 29 20:54 ./
drwx------   52 bozo  bozo      3072 Aug 29 20:51 ../
-rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.addressbook
-rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.addressbook.bak
-rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.addressbook
-rw-rw-r--    1 bozo  bozo         0 Aug 29 20:54 .hidden-file
```

.命令如果作为目录名的一部分的话,那么.表达的是当前目录.".."表示上一级目录.

```
bash$ pwd
/home/bozo/projects
bash$ cd .
bash$ pwd
/home/bozo/projects
bash$ cd ..
bash$ pwd
/home/bozo/
```

.命令经常作为一个文件移动命令的目的地.

```
bash$ cp /home/bozo/current_work/junk/* .
```

. .字符匹配,这是作为正则表达是的一部分,用来匹配任何的单个字符.

" 部分引用."STRING"阻止了一部分特殊字符,具体见第5章.

' 全引用. 'STRING' 阻止了全部特殊字符,具体见第5章.

, 逗号链接了一系列的算术操作,虽然里边所有的内容都被运行了,但只有最后一项被返回.
如:

```
1 let "t2 = ((a = 9, 15 / 3))"  # Set "a = 9" and "t2 = 15 / 3"
```

\ 转义字符,如\X等价于"X"或'X',具体见第5章.

/ 文件名路径分隔符.或用来做除法操作.

` 后置引用,命令替换,具体见第14章

: 空命令,等价于"NOP"(no op,一个什么也不干的命令).也可以被认为与shell的内建命令(true)作用相同.":"命令是一个bash的内建命令,它的返回值为0,就是shell返回的true.
如:

```
1 :
2 echo $?   # 0
```

死循环,如:

```
1 while :
2 do
3    operation-1
4    operation-2
5    ...
6    operation-n
7 done
8
9 # 与下边相同:
10 #    while true
11 #    do
12 #      ...
13 #    done
```

在if/then中的占位符,如:

```
1 if condition
2 then :   # 什么都不做,引出分支.
3 else
4    take-some-action
5 fi
```

在一个2元命令中提供一个占位符,具体见Example 8-2,和"默认参数".如:

```
1 : ${username=`whoami`}
2 # ${username=`whoami`}  如果没有":"的话,将给出一个错误,除非"username"是
3 #                       个命令
```

在here document中提供一个占位符,见Example 17-10.

使用"参数替换"来评估字符串变量(见Example 9-14).如:

```
1 : ${HOSTNAME?} ${USER?} ${MAIL?}
2 #  如果一个或多个必要的环境变量没被设置的话,
3 #+ 就打印错误信息.
```

"变量扩展/子串替换"

在和 > (重定向操作符)结合使用时,把一个文件截断到0长度,没有修改它的权限.

如果文件在之前并不存在,那么就创建它.如:

```
1 : > data.xxx  #文件"data.xxx"现在被清空了.
2
3 #与 cat /dev/null >data.xxx 的作用相同
4 #然而,这不会产生一个新的进程,因为":"是一个内建命令.
```

具体参见Example 12-14.

在和>>重定向操作符结合使用时,将不会对想要附加的文件产生任何影响.

如果文件不存在,将创建.

注意: 这只适用于正规文件,而不是管道,符号连接,和某些特殊文件.

也可能用来作为注释行,虽然我们不推荐这么做.使用#来注释的话,将关闭剩余行的错误检查,所以可以在注释行中写任何东西.然而,使用:的话将不会这样.如:

```
1 : This is a comment thar generates an error,(if [ $x -eq 3] ).
```

":"还用来在/etc/passwd和$PATH变量中用来做分隔符.

```
bash$ echo $PATH
/usr/local/bin:/bin:/usr/X11R6/bin:/sbin:/usr/sbin:/usr/games
```

! 取反操作符,将反转"退出状态"结果,(见Example 6-2).也会反转test操作符的意义.比如修改=为!=.!操作是Bash的一个关键字.

在一个不同的上下文中,!也会出现在"间接变量引用"见Example 9-22.

在另一种上下文中,!还能反转bash的"history mechanism"(见附录J 历史命令)
需要注意的是,在一个脚本中,"history mechanism"是被禁用的.

* 万能匹配字符,用于文件名匹配(这个东西有个专有名词叫file globbing),或者是正则表达式中.注意:在正则表达式匹配中的作用和在文件名匹配中的作用是不同的.

```
bash$ echo *
abs-book.sgml add-drive.sh agram.sh alias.sh
```

* 数学乘法.
**是幂运算.
? 测试操作.在一个确定的表达式中,用?来测试结果.
(())结构可以用来做数学计算或者是写c代码,那?就是c语言的3元操作符的一个.
在"参数替换"中,?测试一个变量是否被set了.
? 在file globbing中和在正则表达式中一样匹配任意的单个字符.
$ 变量替换

```
1 var1=5
2 var2=23skidoo
3
4 echo $var1     # 5
5 echo $var2     # 23skidoo
```

$ 在正则表达式中作为行结束符.
${} 参数替换,见9.3节.
$*,$@ 位置参数
$? 退出状态变量.$?保存一个命令/一个函数或者脚本本身的退出状态.
$$ 进程ID变量.这个$$变量保存运行脚本进程ID
() 命令组.如:

```
1 (a=hello;echo $a)
```

注意:在()中的命令列表,将作为一个子shell来运行.
在()中的变量,由于是在子shell中,所以对于脚本剩下的部分是不可用的.
如:

```
1 a=123
2 ( a=321; )
3
4 echo "a = $a"   # a = 123
5 # 在圆括号中a变量,更像是一个局部变量.
```

用在数组初始化,如:

```
1 Array=(element1,element2,element3)
```

{xxx,yyy,zzz...}
大括号扩展,如:

```
1 cat {file1,file2,file3} > combined_file
2 # 把file1,file2,file3连接在一起,并且重定向到combined_file中.
3
4
5 cp file22.{txt,backup}
6 # 拷贝"file22.txt" 到"file22.backup"中
```

一个命令可能会对大括号中的以逗号分割的文件列表起作用[1]. file globbing将对大括号中的文件名作扩展.
注意: 在大括号中,不允许有空白,除非这个空白是有意义的.

```
echo {file1,file2}\ :{\ A," B",' C'}
```

file1 : A file1 : B file1 : C file2 : A file2 : B file2 : C

{} 代码块.又被称为内部组.事实上,这个结构创建了一个匿名的函数.但是与函数不同的是,在其中声明的变量,对于脚本其他部分的代码来说还是可见的.如:

```
bash$
{
 local a;
 a= 123;
}
```

bash中的local申请的变量只能够用在函数中.

```
1 a=123
2 { a=321; }
3 echo "a = $a"   # a = 321   (说明在代码块中对变量a所作的修改,影响了外边的变量a)
4
5 # Thanks, S.C.
```

下边的代码展示了在{}结构中代码的I/O重定向.

Example 3-1. 代码块和I/O重定向

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # 从 /etc/fstab中读行
 3
 4 File=/etc/fstab
 5
 6 {
 7 read line1
 8 read line2
 9 } < $File
10
11 echo "First line in $File is:"
12 echo "$line1"
13 echo
14 echo "Second line in $File is:"
15 echo "$line2"
16
17 exit 0
18
19 # 现在,你怎么分析每行的分割域
20 # 暗示: 使用 awk.
################################End Script###########################################
```

Example 3-2. 将一个代码块的结果保存到文件

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # rpm-check.sh
 3
 4 # 这个脚本的目的是为了描述,列表,和确定是否可以安装一个rpm包.
 5 # 在一个文件中保存输出.
 6 #
 7 # 这个脚本使用一个代码块来展示
 8
```

```
 9 SUCCESS=0
10 E_NOARGS=65
11
12 if [ -z "$1" ]
13 then
14   echo "Usage: `basename $0` rpm-file"
15   exit $E_NOARGS
16 fi
17
18 {
19   echo
20   echo "Archive Description:"
21   rpm -qpi $1       # 查询说明
22   echo
23   echo "Archive Listing:"
24   rpm -qpl $1       # 查询列表
25   echo
26   rpm -i --test $1  # 查询rpm包是否可以被安装
27   if [ "$?" -eq $SUCCESS ]
28   then
29     echo "$1 can be installed."
30   else
31     echo "$1 cannot be installed."
32   fi
33   echo
34 } > "$1.test"       # 把代码块中的所有输出都重定向到文件中
35
36 echo "Results of rpm test in file $1.test"
37
38 # 查看rpm的man页来查看rpm的选项
39
40 exit 0
```

################################End Script###

注意: 与()中的命令不同的是,{}中的代码块将不能正常地开启一个新shell.[2]

{} \; 路径名.一般都在find命令中使用.这不是一个shell内建命令.

注意: ";"用来结束find命令序列的-exec选项.

[] test.

test的表达式将在[]中.

值得注意的是[是shell内建test命令的一部分,并不是/usr/bin/test中的扩展命令的一个连接.

[[]] test.

test表达式放在[[]]中.(shell关键字)

具体查看[[]]结构的讨论.

[] 数组元素

```
Array[1]=slot_1
echo ${Array[1]}
```

[] 字符范围

在正则表达式中使用,作为字符匹配的一个范围

SHELL十三问

(()) 数学计算的扩展
在(())结构中可以使用一些数字计算.
具体参阅((...))结构.
>&>>&>><
重定向.
scriptname >filename 重定向脚本的输出到文件中.覆盖文件原有内容.
command &>filename 重定向stdout和stderr到文件中
command >&2 重定向command的stdout到stderr
scriptname >>filename 重定向脚本的输出到文件中.添加到文件尾端,如果没有文件,
则创建这个文件.
进程替换,具体见"进程替换部分",跟命令替换极其类似.
(command)>
<(command)
<和> 可用来做字符串比较
<和> 可用在数学计算比较
<< 重定向,用在"here document"
<<< 重定向,用在"here string"
<,> ASCII比较

```
 1 veg1=carrots
 2 veg2=tomatoes
 3
 4 if [[ "$veg1" < "$veg2" ]]
 5 then
 6   echo "Although $veg1 precede $veg2 in the dictionary,"
 7   echo "this implies nothing about my culinary preferences."
 8 else
 9   echo "What kind of dictionary are you using, anyhow?"
10 fi
```

\<,\> 正则表达式中的单词边界.如:
bash$grep '\<the\>' textfile

| 管道.分析前边命令的输出,并将输出作为后边命令的输入.这是一种产生命令链的
好方法.

```
1 echo ls -l | sh
2 #  传递"echo ls -l"的输出到shell中,
3 #+ 与一个简单的"ls -l"结果相同.
4
5
6 cat *.lst | sort | uniq
7 # 合并和排序所有的".lst"文件,然后删除所有重复的行.
```

管道是进程间通讯的一个典型办法,将一个进程的stdout放到另一个进程的stdin中.
标准的方法是将一个一般命令的输出,比如cat或echo,传递到一个过滤命令中(在这个
过滤命令中将处理输入),得到结果,如:
cat $filename1 | $filename2 | grep $search_word
当然输出的命令也可以传递到脚本中.如:
################################Start Script###################################

```
1 #!/bin/bash
```

```
2 # uppercase.sh : 修改输出,全部转换为大写
3
4 tr 'a-z' 'A-Z'
5 #  字符范围必须被""引用起来
6 #+ 来阻止产生单字符的文件名.
7
8 exit 0
```

################################End Script###

现在让我们输送ls -l的输出到一个脚本中.

```
bash$ ls -l | ./uppercase.sh
-RW-RW-R--    1 BOZO  BOZO      109 APR  7 19:49 1.TXT
-RW-RW-R--    1 BOZO  BOZO      109 APR 14 16:48 2.TXT
-RW-R--R--    1 BOZO  BOZO      725 APR 20 20:56 DATA-FILE
```

注意:管道中的一个进程的stdout必须被下一个进程作为stdin读入.否则,数据流会阻塞,并且管道将产生非预期的行为.

如:

```
1 cat file1 file2 | ls -l | sort
2 #从"cat file1 file2"中的输出并没出现
```

作为子进程的运行的管道,不能够改变脚本的变量.

```
1 variable="initial_value"
2 echo "new_value" | read variable
3 echo "variable = $variable"   #variable = initial_value
```

如果管道中的某个命令产生了一个异常,并中途失败,那么这个管道将过早的终止.

这种行为被叫做a broken pipe,并且这种状态下将发送一个SIGPIPE信号.

>| 强制重定向(即使设置了noclobber选项--就是-C选项).这将强制的覆盖一个现存文件.

|| 或-逻辑操作.

& 后台运行命令.一个命令后边跟一个&,将表示在后台运行.

```
bash$sleep 10 &
[1] 850
[1]+ Done   sleep 10
```

在一个脚本中,命令和循环都可能运行在后台.

Example 3-3. 在后台运行一个循环

################################Start Script###

```
1 #!/bin/bash
2 #background-loop.sh
3
4 for i in 1 2 3 4 5 6 7 8 9 10    #第一个循环
5 do
6 echo -n "$i"
7 done&      #在后台运行这个循环
8       #在第2个循环之后,将在某些时候执行.
9
10 echo #这个'echo'某些时候将不会显示.
11
12 for i in 11 12 13 14 15 16 17 18 19 20  #第二个循环
13 do
14  echo -n "$i"
```

```
15 done
16
17 echo #这个'echo'某些时候将不会显示.
18
19 #-------------------------------------------------------
20
21 #期望的输出应该是
22 #1 2 3 4 5 6 7 8 9 10
23 #11 12 13 14 15 16 17 18 19 20
24
25 #然而实际的结果有可能是
26 #11 12 13 14 15 16 17 18 19 20
27 #1 2 3 4 5 6 7 8 9 10 bozo $
28 #(第2个'echo'没执行,为什么?)
29
30 #也可能是
31 #1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
32 #(第1个'echo'没执行,为什么?)
33
34 #非常少见的执行结果,也有可能是:
35 #11 12 13 1 2 3 4 5 6 7 8 9 10 14 15 16 17 18 19 20
36 #前台的循环先于后台的执行
37
38 exit 0
39
40 # Nasimuddin Ansari 建议加一句 sleep 1
41 #+ 在 6行和14行的 echo -n "$i"之后加
42 #+ 将看到一些乐趣
```

################################End Script###

注意:在一个脚本内后台运行一个命令,有可能造成这个脚本的挂起,等待一个按键响应.幸运的是,我们可以在Example 11-24附近,看到这个问题的解决办法.

&& 与-逻辑操作.

- 选项,前缀.在所有的命令内如果想使用选项参数的话,前边都要加上"-".

```
COMMAND -[Option1][Option2][...]
ls -al
sort -dfu $filename
set -- $variable
 1 if [ $file1 -ot $file2 ]
 2 then
 3   echo "File $file1 is older than $file2."
 4 fi
 5
 6 if [ "$a" -eq "$b" ]
 7 then
 8   echo "$a is equal to $b."
 9 fi
10
11 if [ "$c" -eq 24 -a "$d" -eq 47 ]
```

```
 12 then
 13   echo "$c equals 24 and $d equals 47."
 14 fi
```

- 用于重定向 stdin 或 stdout.

```
################################Start Script#########################################
 1 (cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
 2 # 从一个目录移动整个目录树到另一个目录
 3 # [courtesy Alan Cox <a.cox@swansea.ac.uk>, with a minor change]
 4
 5 # 1) cd /source/directory    源目录
 6 # 2) &&                     与操作,如果cd命令成功了,那么就执行下边的命令
 7 # 3) tar cf - .             'c'创建一个新文档,'f'后边跟'-'指定目标文件作为stdout
 8 #                          '-'后边的'f'(file)选项,指明作为stdout的目标文件.
 9 #                          并且在当前目录('.')执行.
10 # 4) |                      管道...
11 # 5) ( ... )                一个子shell
12 # 6) cd /dest/directory      改变当前目录到目标目录.
13 # 7) &&                     与操作,同上.
14 # 8) tar xpvf -             'x'解档,'p'保证所有权和文件属性,
15 #      'v'发完整消息到stdout
16 #                          'f'后边跟'-',从stdin读取数据
17 #
18 #                          注意:'x' 是一个命令, 'p', 'v', 'f' 是选项.
19 # Whew!
20
21
22
23 # 更优雅的写法应该是
24 #   cd source/directory
25 #   tar cf - . | (cd ../dest/directory; tar xpvf -)
26 #
27 #    当然也可以这么写:
28 # cp -a /source/directory/* /dest/directory
29 #    或者:
30 # cp -a /source/directory/* /source/directory/.[^.]* /dest/directory
31 #    如果在/source/directory中有隐藏文件的话.
################################End Script###########################################
################################Start Script#########################################
1 bunzip2 linux-2.6.13.tar.bz2 | tar xvf -
2 # --未解压的tar文件--   | --然后把它传递到"tar"中--
3 # 如果 "tar" 没能够正常的处理"bunzip2",
4 # 这就需要使用管道来执行2个单独的步骤来完成它.
5 # 这个练习的目的是解档"bzipped"的kernel源文件.
################################End Script###########################################
```

注意:在上边这个例子中'-'不太象是bash的操作符,而更像是tar的参数.

```
bash$echo "whatever" | cat -
whatever
```

在需要一个文件名的地方,-重定向输出到stdout(如在tar和cf命令中),或者从

stdin中接受输入,而不是从一个文件中接受输入.这是在管道中作为一个过滤器,来使用文件定位工具的一种办法.

```
bash$file
用法: file [-bciknvzl] [-f namefile] [-m magicfiles] file...
```

上边这个例子file将会出错,提示你如何使用file命令.

添加一个"-"将得到一个更有用的结果.这将使得shell等待用户输入.

```
bash$file -
abc
standard input:     ASCII text
bash$file -
#!/bin/bash
standard input:     Bourn-Again shell script tesxt executable
```

现在命令从stdin中接受了输入,并分析它.

"-"常用于管道后边的命令,具体参看33.7节,来看使用技巧.

使用diff命令来和另一个文件的一部分进行比较.

```
grep Linux file1 | diff file2 -
```

最后,一个真实世界的使用tar命令的例子.

Example 3-4. 备份最后一天所有修改的文件.

```
################################Start Script#########################################
 1 #!/bin/bash
 2
 3 #  在一个"tarball"中(经过tar和gzip处理过的文件)
 4 #+ 备份最后24小时当前目录下d所有修改的文件.
 5
 6 BACKUPFILE=backup-$(date +%m-%d-%Y)
 7 #               在备份文件中嵌入时间.
 8 #               Thanks, Joshua Tschida, for the idea.
 9 archive=${1:-$BACKUPFILE}
10 #  如果在命令行中没有指定备份文件的文件名,
11 #+ 那么将默认使用"backup-MM-DD-YYYY.tar.gz".
12
13 tar cvf - `find . -mtime -1 -type f -print` > $archive.tar
14 gzip $archive.tar
15 echo "Directory $PWD backed up in archive file \"$archive.tar.gz\"."
16
17
18 #  Stephane Chazelas指出上边代码,
19 #+ 如果在发现太多的文件的时候,或者是如果文件
20 #+ 名包括空格的时候,将执行失败.
21
22 # Stephane Chazelas建议使用下边的两种代码之一
23 # --------------------------------------------------------------------
24 #   find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$archive.tar"
25 #      使用gnu版本的find.
26
27
28 #   find . -mtime -1 -type f -exec tar rvf "$archive.tar" '{}' \;
29 #        对于其他风格的UNIX便于移植,但是比较慢.
```

```
30 # ---------------------------------------------------------------
31
32
33 exit 0
################################End Script#########################################
```

注意:以"-"开头的文件名在使用"-"作为重定向操作符的时候,可能会产生问题.
应该写一个脚本来检查这个问题,并给这个文件加上合适的前缀.如:
./-FILENAME, $PWD/-FILENAME,或$PATHNAME/-FILENAME.
如果变量的值以"-"开头,可能也会引起问题.

```
1 var="-n"
2 echo $var
3 #具有"echo -n"的效果了,这样什么都不会输出的.
```

- 之前工作的目录."cd -"将回到之前的工作目录,具体请参考"$OLDPWD"环境变量.
注意:一定要和之前讨论的重定向功能分开,但是只能依赖上下文区分.
- 算术减号.
= 算术等号,有时也用来比较字符串.

```
1 a=28
2 echo $a   # 28
```

+ 算术加号,也用在正则表达式中.
+ 选项,对于特定的命令来说使用"+"来打开特定的选项,用"-"来关闭特定的选项.
% 算术取模运算.也用在正则表达式中.
~ home目录.相当于$HOME变量.~bozo是bozo的home目录,并且ls ~bozo将列出其中的
内容. ~/就是当前用户的home目录,并且ls ~/将列出其中的内容,如:

```
bash$ echo ~bozo
/home/bozo
bash$ echo ~
/home/bozo
bash$ echo ~/
/home/bozo/
bash$ echo ~:
/home/bozo:
bash$ echo ~nonexistent-user
~nonexistent-user
```

~+ 当前工作目录,相当于$PWD变量.
~- 之前的工作目录,相当于$OLDPWD内部变量.
=~ 用于正则表达式,这个操作将在正则表达式匹配部分讲解,只有version3才支持.
^ 行首,正则表达式中表示行首."^"定位到行首.
控制字符
修改终端或文本显示的行为.控制字符以CONTROL + key组合.
控制字符在脚本中不能正常使用.
Ctl-B 光标后退,这应该依赖于bash输入的风格,默认是emacs风格的.
Ctl-C Break,终止前台工作.
Ctl-D 从当前shell登出(和exit很像)
"EOF"(文件结束符).这也能从stdin中终止输入.
在console或者在xterm window中输入的时候,Ctl-D将删除光标下字符.
当没有字符时,Ctrl-D将退出当前会话.在xterm window也有关闭窗口
的效果.
Ctl-G beep.在一些老的终端,将响铃.

Ctl-H backspace,删除光标前边的字符.如:

```
1 #!/bin/bash
2 # 在一个变量中插入Ctl-H
3
4 a="^H^H"                # 两个 Ctl-H (backspaces).
5 echo "abcdef"           # abcdef
6 echo -n "abcdef$a "      # abcd f
7 # 注意结尾的空格 ^            ^ 两个 twice.
8 echo -n "abcdef$a"       # abcdef
9 # 结尾没有空格            没有 backspace 的效果了(why?).
10                        # 结果并不像期望的那样
11 echo; echo
```

Ctl-I 就是tab键.

Ctl-J 新行.

Ctl-K 垂直tab.(垂直tab?新颖,没听过)

作用就是删除光标到行尾的字符.

Ctl-L clear,清屏.

Ctl-M 回车

##################################Start Script###

```
1 #!/bin/bash
2 # Thank you, Lee Maschmeyer, for this example.
3
4 read -n 1 -s -p $'Control-M leaves cursor at beginning of this line. Press Enter. \x0d'
5                          #当然,'0d'就是二进制的回车.
6 echo >&2   #  '-s'参数使得任何输入都不将回显出来
7          #+ 所以,明确的重起一行是必要的.
8
9 read -n 1 -s -p $'Control-J leaves cursor on next line. \x0a'
10 echo >&2   #  Control-J 是换行.
11
12 ###
13
14 read -n 1 -s -p $'And Control-K\x0bgoes straight down.'
15 echo >&2   #  Control-K 是垂直制表符.
16
17 # 关于垂直制表符效果的一个更好的例子见下边:
18
19 var=$'\x0aThis is the bottom line\x0bThis is the top line\x0a'
20 echo "$var"
21 #  这句与上边的例子使用的是同样的办法,然而:
22 echo "$var" | col
23 #  这将造成垂直制表符右边的部分在左边部分的上边.
24 #  这也解释了为什么我们要在行首和行尾加上一个换行符--
25 #+ 来避免一个混乱的屏幕输出.
26
27 # Lee Maschmeyer的解释:
28 # ----------------------
29 #  In the [first vertical tab example] . . . the vertical tab
```

```
29 #  在这里[第一个垂直制表符的例子中] . . . 这个垂直制表符
30 #+ makes the printing go straight down without a carriage return.
31 #  This is true only on devices, such as the Linux console,
32 #+ that can't go "backward."
33 #  The real purpose of VT is to go straight UP, not down.
34 #  It can be used to print superscripts on a printer.
34 #  它可以用来在一个打印机上打印上标.
35 #  col的作用,可以用来模仿VT的合适的行为.
36
37 exit 0
##################################End Script#############################################
```

Ctl-Q 继续(等价于XON字符),这个继续的标准输入在一个终端里
Ctl-S 挂起(等价于XOFF字符),这个被挂起的stdin在一个终端里,用Ctl-Q恢复
Ctl-U 删除光标到行首的所有字符,在某些设置下,删除全行.
Ctl-V 当输入字符时,Ctl-V允许插入控制字符.比如,下边2个例子是等价的
  echo -e '\x0a'
  echo <Ctl-V><Ctl-J>
  Ctl-V在文本编辑器中十分有用,在vim中一样.
Ctl-W 删除当前光标到前边的最近一个空格之间的字符.
  在某些设置下,删除到第一个非字母或数字的字符.
Ctl-Z 终止前台工作.

空白部分
分割命令或者是变量.包括空格,tab,空行,或任何它们的组合.
在一些特殊情况下,空白是不允许的,如变量赋值时,会引起语法错误.
空白行在脚本中没有效果.
"$IFS",对于某些命令输入的特殊变量分割域,默认使用的是空白.
如果想保留空白,使用引用.

注意事项:

[1] shell做大括号的命令扩展.但是命令本身需要对扩展的结果作处理.
[2] 例外:在pipe中的一个大括号中的代码段可能运行在一个子shell中.

```
1 ls | { read firstline; read secondline; }
2 #  错误,在打括号中的代码段,将运行到子shell中.
3 #+ 所以ls的输出将不能传递到代码块中.
4 echo "First line is $firstline; second line is $secondline"  # 不能工作
5
6 # Thanks, S.C.
```

[3] 换行符也被认为是空白.这也解释了为什么一个空行也会被认为是空白.

第4章 变量和参数的介绍

==========

4.1 变量替换

$ 变量替换操作符
只有在变量被声明,赋值,unset或exported或者是在变量代表一个signal的时候,
变量才会是以本来的面目出现在脚本里.变量在被赋值的时候,可能需要使用"=",
read状态或者是在循环的头部.
在""中还是会发生变量替换,这被叫做部分引用,或叫弱引用.而在''中就不会发生变

量替换,这叫做全引用,也叫强引用.具体见第5章的讨论.
注意:$var与${var}的区别,不加{},在某些上下文将引起错误,为了安全,使用2.
具体见9.3节 参数替换.

Example 4-1. 变量赋值和替换

```
################################Start Script#########################################
 1 #!/bin/bash
 2
 3 # 变量赋值和替换
 4
 5 a=375
 6 hello=$a
 7
 8 #-------------------------------------------------------------------------
 9 # 强烈注意,在赋值的前后一定不要有空格.
10 # 如果有空格会发生什么?
11
12 #  如果"VARIABLE =value",
13 #             ^
14 #+ 脚本将尝试运行一个"VARIABLE"的命令,带着一个"=value"参数.
15
16 #  如果"VARIABLE= value",
17 #              ^
18 #+ script tries to run "value" command with
18 #+ 脚本将尝试运行一个"value"的命令,带着
19 #+ the environmental variable "VARIABLE" set to "".
19 #+ 一个被赋成""值的环境变量"VARIABLE".
20 #-------------------------------------------------------------------------
21
22
23 echo hello    # 没有变量引用,不过是个hello字符串
24
25 echo $hello
26 echo ${hello} # 同上
27
28 echo "$hello"
29 echo "${hello}"
30
31 echo
32
33 hello="A B  C   D"
34 echo $hello   # A B C D
35 echo "$hello" # A B  C   D
36 # 就象你看到的echo $hello   和    echo "$hello"   将给出不同的结果.
37 #                              ^     ^
38 # Quoting a variable preserves whitespace.
38 # 引用一个变量将保留其中的空白,当然,如果是变量替换就不会保留了.
39
40 echo
```

```
41
42 echo '$hello'  # $hello
43 #    ^      ^
44 # 全引用的作用
45 #+ 将导致"$"变成一个单独的字符.
46
47 # 注意两种引用不同的效果
48
49
50 hello=    # 设置为空值
51 echo "\$hello (null value) = $hello"
52 #  注意设置一个变量为空,与unset它,不是一回事,虽然看起来一样
53 #
54
55 # ------------------------------------------------------------
56
57 #  可以在同一行上设置多个变量.
58 #+ 要以空白分隔
59 #  小心,这会降低可读性,和可移植性.
60
61 var1=21  var2=22  var3=$V3
62 echo
63 echo "var1=$var1   var2=$var2   var3=$var3"
64
65 # 在老版本的"sh"上,可能会有问题.
66
67 # ------------------------------------------------------------
68
69 echo; echo
70
71 numbers="one two three"
72 #          ^   ^
73 other_numbers="1 2 3"
74 #              ^ ^
75 #  如果变量中有空白,那么引用就必要了.
76 #
77 echo "numbers = $numbers"
78 echo "other_numbers = $other_numbers"   # other_numbers = 1 2 3
79 echo
80
81 echo "uninitialized_variable = $uninitialized_variable"
82 # Uninitialized变量为空值(根本就没赋值).
83 uninitialized_variable=   #  声明,但是没被初始化
84                           #+ 其实和前边设置为空值得作用是一样的.
85 echo "uninitialized_variable = $uninitialized_variable"
86                           # 还是一个空值
87
88 uninitialized_variable=23       # 赋值
```

```
89 unset uninitialized_variable    # Unset it.
90 echo "uninitialized_variable = $uninitialized_variable"
91                          # 还是空值
92 echo
93
94 exit 0
```

################################End Script###

注意: 一个空值变量,或者是根本就没声明的变量,在赋值之前使用它可能会引起问题.
 但是还是可以用来做算术运算

################################Start Script###

```
1 echo "$uninitialized"                  # (blank line)
2 let "uninitialized += 5"               # Add 5 to it.
3 echo "$uninitialized"                  # 5
4
5 #  结论:
6 #  对于一个空值变量在做算术操作的时候,就好像它的值为0一样.
8 #  This is undocumented (and probably non-portable) behavior.
7 #  这并没被文档化(可能是不可移植)的行为.
```

################################End Script###

具体参考 Example 11-21

4.2 变量赋值

= 赋值操作符(前后都不能有空白)

 不要与-eq混淆,那个是test,并不是赋值.

 注意,=也可被用来做test操作,这依赖于上下文.

Example 4-2. 一般的变量赋值

################################Start Script###

```
 1 #!/bin/bash
 2 # "裸体"变量
 3
 4 echo
 5
 6 # 变量什么时候是"裸体"的,比如前边少了$的时候.
 7 # 当它被赋值的时候,而不是被引用的时候.
 8
 9 # 赋值
10 a=879
11 echo "The value of \"a\" is $a."
12
13 # 使用let赋值
14 let a=16+5
15 echo "The value of \"a\" is now $a."
16
17 echo
18
19 # 在for循环中
20 echo -n "Values of \"a\" in the loop are: "
21 for a in 7 8 9 11
```

```
22 do
23   echo -n "$a "
24 done
25
26 echo
27 echo
28
29 # 在read命令状态中
30 echo -n "Enter \"a\" "
31 read a
32 echo "The value of \"a\" is now $a."
33
34 echo
35
36 exit 0
```

################################End Script###

Example 4-3. 变量赋值,一般的和比较特殊的

################################Start Script###

```
 1 #!/bin/bash
 2
 3 a=23             # Simple case
 4 echo $a
 5 b=$a
 6 echo $b
 7
 8 # 现在让我们来点小变化
 9
10 a=`echo Hello!`   # 把echo命令的结果传给变量a
11 echo $a
12 #  注意,如果在命令扩展结构中使用一个(!)的话,在命令行中将不能工作
13 #+ 因为这触发了Bash的"历史机制".
14 #  但是,在校本里边使用的话,历史功能是被关闭的,所以就能够正常运行.
15
16
17 a=`ls -l`         # 把ls -l的结果给a
18 echo $a          # 别忘了,这么引用的话,ls的结果中的所有空白部分都没了(包括换行)
19 echo
20 echo "$a"        # 这么引用就正常了,保留了空白
21                  # (具体参阅章节"引用")
22
23 exit 0
```

################################End Script###

使用$(...)机制进行的变量赋值(除去使用``来赋值的另外一种新方法).事实上这两种方法都是命令替换的一种形式.

```
# 来自于/ect/rc.d/rc.local
R=$(cat /ect/redhat-release)
arch=$(uname -m)
```

4.3 Bash变量是不分类型的

不像其他程序语言一样,Bash并不对变量区分"类型".本质上,Bash变量都是字符串.
但是依赖于上下文,Bash也允许比较操作和算术操作.决定这些的关键因素就是,变量中的值
是否只有数字.

Example 4-4 整型还是string?

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # int-or-string.sh: 整形还是string?
 3
 4 a=2334                # 整型
 5 let "a += 1"
 6 echo "a = $a "         # a = 2335
 7 echo                  # 还是整型
 8
 9
10 b=${a/23/BB}           # 将23替换成BB
11                       # 这将把b变量从整型变为string
12 echo "b = $b"          # b = BB35
13 declare -i b           # 即使使用declare命令也不会对此有任何帮助,9.4节有解释
14 echo "b = $b"          # b = BB35
15
16 let "b += 1"           # BB35 + 1 =
17 echo "b = $b"          # b = 1
18 echo
19
20 c=BB34
21 echo "c = $c"          # c = BB34
22 d=${c/BB/23}           # S将BB替换成23
23                       # 这使得$d变为一个整形
24 echo "d = $d"          # d = 2334
25 let "d += 1"           # 2334 + 1 =
26 echo "d = $d"          # d = 2335
27 echo
28
29 # 关于空变量怎么样?
30 e=""
31 echo "e = $e"          # e =
32 let "e += 1"           # 算术操作允许一个空变量?
33 echo "e = $e"          # e = 1
34 echo                  # 空变量将转换成一个整型变量
35
36 # 关于未声明的变量怎么样?
37 echo "f = $f"          # f =
38 let "f += 1"           # 算术操作允许么?
39 echo "f = $f"          # f = 1
40 echo                  # 未声明的变量将转换成一个整型变量
41
42
```

```
43
44 # 所以说Bash中的变量都是无类型的.
45
46 exit 0
```

################################End Script###

4.4 特殊的变量类型

---

local variables

这种变量只有在代码块或者是函数中才可见(具体见23.2和23章)

environmental variables

这种变量将改变用户接口和shell的行为.

在一般的上下文中,每个进程都有自己的环境,就是一组保持进程可能引用的信息的变量.这种情况下,shell于一个一般进程是相同的.

每次当shell启动时,它都将创建自己的环境变量.更新或者添加新的环境变量,将导致shell更新它的环境,同时也会影响所有继承自这个环境的所有子进程(由这个命令导致的).

注意:分配给环境变量的空间是受限的.创建太多的环境变量将引起空间溢出,这会引起问题.

关于eval命令,具体见第11章

```
bash$ eval "`seq 10000 | sed -e 's/.*/export var&=ZZZZZZZZZZZZZ/'`"
bash$ du
bash: /usr/bin/du: Argument list too long
```

如果一个脚本设置了环境变量,需要export它,来通知本脚本的环境,这是export命令的功能,关于export命令,具体见11章.

脚本只能对它产生的子进程export变量.一个从命令行被调用的脚本export的变量,将不能影响调用这个脚本的那个命令行shell的环境.

positional parameters

就是从命令行中传进来的参数,$0, $1, $2, $3...

$0就是脚本文件的名字,$1是第一个参数,$2为第2个...,参见[1](有$0的说明),$9以后就需要打括号了,如${10},${11},${12}...

两个值得注意的变量$*和$@(第9章有具体的描述),表示所有的位置参数.

Example 4-5 位置参数

################################Start Script###

```
 1 #!/bin/bash
 2
 3 # 作为用例,调用这个脚本至少需要10个参数,如
 4 # ./scriptname 1 2 3 4 5 6 7 8 9 10
 5 MINPARAMS=10
 6
 7 echo
 8
 9 echo "The name of this script is \"$0\"."
10 # 添加./是为了当前目录
11 echo "The name of this script is \"`basename $0`\"."
12 # 去掉目录信息,具体见'basename'命令
13
14 echo
```

```
15
16 if [ -n "$1" ]            # 测试变量被被引用
17 then
18  echo "Parameter #1 is $1"  # "#"没被转义
19 fi
20
21 if [ -n "$2" ]
22 then
23  echo "Parameter #2 is $2"
24 fi
25
26 if [ -n "$3" ]
27 then
28  echo "Parameter #3 is $3"
29 fi
30
31 # ...
32
33
34 if [ -n "${10}" ] # 大于9的参数必须出现在{}中.
35 then
36  echo "Parameter #10 is ${10}"
37 fi
38
39 echo "-----------------------------------"
40 echo "All the command-line parameters are: "$*""
41
42 if [ $# -lt "$MINPARAMS" ] #$#是传到脚本里的位置参数的个数
43 then
44   echo
45   echo "This script needs at least $MINPARAMS command-line arguments!"
46 fi
47
48 echo
49
50 exit 0
```

################################End Script###

{}标记法是一种很好的使用位置参数的方法.这也需要间接引用(见Example 34-2)

```
1 args=$#          # 位置参数的个数
2 lastarg=${!args}
3 # 或:       lastarg=${!#}
4 # 注意 lastarg=${!$#} 将报错
```

一些脚本可能会依赖于使用不同的调用名字,而表现出不同的行为,这样一般都需要判断$0,而其他的名字都是通过ln命令产生的链接.(具体参见Example 12-2)

如果脚本需要一个命令行参数,而调用的时候,没用这个参数,这就有可能造成分配一个空变量,这样估计就会引起问题.一种解决办法就是在这个位置参数,和相关的变量后边,都添加一个额外的字符.具体见下边的例子.

################################Start Script###

```
 1 variable1_=$1_  # 而不是 variable1=$1
 2 # 这将阻止一个错误,即使在调用时没使用这个位置参数.
 3
 4 critical_argument01=$variable1_
 5
 6 # 这个扩展的字符是可以被消除掉的,就像这样.
 7 variable1=${variable1_/_/}
 8 # 副作用就是$variable1_多了一个下划线
 9 # 这里使用了一个参数替换模版(后边会有具体的讨论)
10 # (Leaving out the replacement pattern results in a deletion.)
10 # (在一个删除动作中,节省了一个替换模式)
11
12
13 # 一个解决这种问题的更简单的做法就是,判断一下这个位置参数是否传递下来了
14 if [ -z $1 ]
15 then
16   exit $E_MISSING_POS_PARAM
17 fi
18
19
20 #  但是上边的方法将可能产生一个意外的副作用
21 #  参数替换的更好的办法应该是:
22 #        ${1:-$DefaultVal}
23 #  具体察看"Parameter Substition"节
24 #+ 在第9章
```

##################################End Script##

Example 4-6 wh,whois节点名字查询

##################################Start Script##

```
 1 #!/bin/bash
 2 # ex18.sh
 3
 4 # Does a 'whois domain-name' lookup on any of 3 alternate servers:
 5 #                 ripe.net, cw.net, radb.net
 6
 7 # 把这个脚本重命名为'wh',然后放到/usr/local/bin下
 8
 9 # 需要3个符号链接
10 # ln -s /usr/local/bin/wh /usr/local/bin/wh-ripe
11 # ln -s /usr/local/bin/wh /usr/local/bin/wh-cw
12 # ln -s /usr/local/bin/wh /usr/local/bin/wh-radb
13
14 E_NOARGS=65
15
16
17 if [ -z "$1" ]
18 then
19   echo "Usage: `basename $0` [domain-name]"
```

```
20   exit $E_NOARGS
21 fi
22
23 # Check script name and call proper server.
23 # 检查脚本名字,然后调用合适的服务器
24 case `basename $0` in    # Or:    case ${0##*/} in
25     "wh"     ) whois $1@whois.ripe.net;;
26     "wh-ripe") whois $1@whois.ripe.net;;
27     "wh-radb") whois $1@whois.radb.net;;
28     "wh-cw"  ) whois $1@whois.cw.net;;
29     *        ) echo "Usage: `basename $0` [domain-name]";;
30 esac
31
32 exit $?
```

################################End Script###

shift shift命令重新分配位置参数,其实就是向左移动一个位置.

$1 <--- $2, $2 <--- $3, $3 <--- $4, 等等.

老的$1将消失,但是$0(脚本名)是不会改变的.如果你使用了大量的位置参数,那么

shift命令允许你存取超过10个参数.虽然{}表示法也允许这样.

Example 4-7 使用shift

################################Start Script###

```
 1 #!/bin/bash
 2 # 使用'shift'来穿过所有的位置参数.
 3
 4 #  把这个脚本命名为shft,
 5 #+ 并且使用一些参数来调用它,如:
 6 #          ./shft a b c def 23 skidoo
 7
 8 until [ -z "$1" ]  # 知道所有参数都用光
 9 do
10   echo -n "$1 "
11   shift
12 done
13
14 echo              # 额外的换行.
15
16 exit 0
```

################################End Script###

在将参数传递到函数中时,shift的工作方式也基本差不多.具体见Example 33-15

注意事项:

[1] 进程调用设置$0参数的脚本.一般的,这个参数就是脚本名字.具体察看execv的man页.

第5章 引用(翻译的可能有问题,特指引号)

==========

引号的特殊效果就是,保护字符串中的特殊字符不被shell或者是shell脚本重新解释或者扩展.

(我们这里所说的"特殊"指的是一些字符在shell中具有的特殊意义,比如*)

如:

```
bash$ ls -l [Vv]*
```

-rw-rw-r-- 1 bozo bozo 324 Apr 2 15:05 VIEWDATA.BAT
-rw-rw-r-- 1 bozo bozo 507 May 4 14:25 vartrace.sh
-rw-rw-r-- 1 bozo bozo 539 Apr 14 17:11 viewdata.sh

bash$ ls -l '[Vv]*'
ls: [Vv]*: No such file or directory
在我们一般的生活中,引号内的内容往往有特殊的含义,而在Bash中,当我们引用一个字符串,
我们是保护它的字面含义.
特定的程序和工具能够重新解释或扩展特殊的字符.引用的一个重要的作用就是保护命令行中
的参数,但还是允许正在调用的程序来扩展它.
bash$ grep '[Ff]irst' *.txt
file1.txt:This is the first line of file1.txt.
file2.txt:This is the First line of file2.txt.
注意 grep [Ff]irst *.txt在Bash下的行为(其实就是正则表达式么),[1]
引用还可以抑制echo命令的换行作用.
bash$ echo $(ls -l)
total 8 -rw-rw-r-- 1 bozo bozo 130 Aug 21 12:57 t222.sh -rw-rw-r-- 1 bozo bozo 78 Aug 21 12:57 t71.sh
bash$ echo "$(ls -l)"
total 8
-rw-rw-r-- 1 bozo bozo 130 Aug 21 12:57 t222.sh
-rw-rw-r-- 1 bozo bozo 78 Aug 21 12:57 t71.sh
5.1 引用变量

在一个双引号中直接使用变量名,一般都是没有问题的.它阻止了所有在引号中的特殊字符的
重新解释--包括变量名[2]--但是$,`和\除外.[3]保留$,作为特殊字符的意义,是为了能够在双
引号中也能够正常地引用变量("$var").这样在""中可以使用变量所表达的值(Example 4-1).
使用""来防止单词分割.[4]如果在参数列表中使用双引号,将使得双引号中的参数作为一个参
数.即使双引号中的字符串包含多个单词(也就是包含空白部分),也不会变为多个参数,如:
```
1 variable1="a variable containing five words"
2 COMMAND This is $variable1    # COMMAND将以7个参数来执行
3 # "This" "is" "a" "variable" "containing" "five" "words"
4
5 COMMAND "This is $variable1"  # COMMAND将以1个参数来执行
6 # "This is a variable containing five words"
7
8
9 variable2=""    # 空值
10
11 COMMAND $variable2 $variable2 $variable2        # COMMAND将不带参数执行
12 COMMAND "$variable2" "$variable2" "$variable2"  # COMMAND将以3个空参数来执行
13 COMMAND "$variable2 $variable2 $variable2"      # COMMAND将以1个参数来执行(2空格)
```
用双引号把参数封到echo中是很有必要的,只有在单词分隔或时保留空白时的时候可能
有些问题.
Example 5-1 echo一些诡异的变量
```
################################Start Script#########################################
1 #!/bin/bash
2 # weirdvars.sh: echo诡异的变量
3
```

```
 4 var="'(]\\{}\$\""
 5 echo $var        # '(]\{}$"
 6 echo "$var"      # '(]\{}$"    并没有什么不同
 7
 8 echo
 9
10 IFS='\'
11 echo $var        # '(] {}$"     \ 转换成空格了?明显和IFS有关系么!又不傻!
12 echo "$var"      # '(]\{}$"
13
14 exit 0
```

################################End Script##

单引号操作总体上和""很像,但不允许引用变量.因为$的特殊含义被关闭了.在"中除了',其他
字符都没有特殊的含义了.所以单引号比双引号严格.

因为即使是\,在"中都被关闭了,所以你想在"中显示'的含义,将得不到预期的效果.

```
 1 echo "Why can't I write 's between single quotes"
 2
 3 echo
 4
 5 # 一种绕弯的方法
 6 echo 'Why can'\''t I write '''''s between single quotes'
 7 #    |-------|  |----------|   |------------------------|
 8 # 包含了2个单引号字符,原书好像有错误
```

注意事项:

[1] 除非当前目录下,正好有个叫first的文件.

[2] 即使是变量的值也是有副作用的(见下边)

[3] 如果在""中包含"!"的话,在命令行中将会出现错误.因为这个"!"被当作历史命令来解释了.
在一个脚本中,这种情况是不会发生的,因为在脚本中,Bash历史记录被关闭了.
下边是一些关于"\"一些不协调的行为.

```
bash$ echo hello\!
hello!
bash$ echo "hello\!"
hello\!
bash$ echo -e x\ty
xty
bash$ echo -e "x\ty"
x       y
```

[4] "单词分隔",在这个上下文中意味着,将一个字符串分隔为一些分离的参数.

5.2 转义(\)

转义是一种引用单个字符的方法.一个具有特殊含义的字符前边放上一个转义符(\)就告诉shell
这个字符失去了特殊的含义.

值得注意的是,在某些特定的命令和工具中,比如echo和sed,转义符往往会起到相反的效果,
它反倒有可能引发出这个字符特殊的含义.

对于特定的转义符的特殊的含义
在echo和sed中所使用的

\n 意味着新的一行

\r 回车

\t tab键
\v vertical tab(垂直tab),查前边的Ctl-K
\b backspace,查前边的Ctl-H
\a "alert"(如beep或flash)
\0xx 转换成8进制ASCII解码,等价于oxx
Example 5-2 转义符
##################################Start Script###

```
 1 #!/bin/bash
 2 # escaped.sh: 转义符
 3
 4 echo; echo
 5
 6 echo "\v\v\v\v"      # 逐字的打印\v\v\v\v .
 7 # 使用-e选项的echo命令来打印转义符
 8 echo "=============="
 9 echo "VERTICAL TABS"
10 echo -e "\v\v\v\v"   # Prints 4 vertical tabs.
11 echo "=============="
12
13 echo "QUOTATION MARK"
14 echo -e "\042"       # 打印" (引号, 8进制的ASCII 码就是42).
15 echo "=============="
16
17 # The $'\X' construct makes the -e option unnecessary.
17 # 如果使用$'\X'结构,那-e选项就不必要了
18 echo; echo "NEWLINE AND BEEP"
19 echo $'\n'          # 新行.
20 echo $'\a'          # Alert (beep).
21
22 echo "==============="
23 echo "QUOTATION MARKS"
24 # 版本2以后Bash允许使用$'\nnn'结构
25 # 注意这种情况,'\nnn\是8进制
26 echo $'\t \042 \t'   # Quote (") framed by tabs.
27
28 # 当然,也可以使用16进制的值,使用$'\xhhh' 结构
29 echo $'\t \x22 \t'  # Quote (") framed by tabs.
30
31 # 早一点的Bash版本允许'\x022'这种形式
32 echo "==============="
33 echo
34
35
36 # 分配ASCII字符到变量中
37 # ---------------------
38 quote=$'\042'       # \042是",分配到变量中
39 echo "$quote This is a quoted string, $quote and this lies outside the quotes."
40
```

```
41 echo
42
43 # Concatenating ASCII chars in a variable.
43 # 变量中的连续的ASCII char.
44 triple_underline=$'\137\137\137'  # 137 是8进制的ASCII 码'_'.
45 echo "$triple_underline UNDERLINE $triple_underline"
46
47 echo
48
49 ABC=$'\101\102\103\010'          # 101, 102, 103 是8进制的码A, B, C.
50 echo $ABC
51
52 echo; echo
53
54 escape=$'\033'                   # 033 是8进制码for escape.
55 echo "\"escape\" echoes as $escape"
56 #"escape" echoes as               没有变量被输出
57
58 echo; echo
59
60 exit 0
```

################################End Script###

另一个关于$"字符串扩展结果的例子见Example 34-1

\" 表达引号本身

```
 1 echo "Hello"               # Hello
 2 echo "\"Hello\", he said."    # "Hello", he said.
```

\$ $号本身,跟在\$后的变量名,将不能扩展

```
 1 echo "\$variable01"  # 结果是$variable01
```

\\ \号本身.

```
 1 echo "\\"  # 结果是\
 2
 3 # 相反的 . . .
 4
 5 echo "\"   # 这会出现第2个命令提示符,说白了就是提示你命令不全,你再补个"就
 6   # 好了.如果是在脚本里,就会给出一个错误.
```

注意:\的行为依赖于它是否被转义,被"",或者是否在"命令替换"和"here document"中.

################################Start Script###

```
 1                  #  简单的转义和""
 2 echo \z           #  z
 3 echo \\z          # \z
 4 echo '\z'         # \z
 5 echo '\\z'        # \\z
 6 echo "\z"         # \z
 7 echo "\\z"        # \z
 8
 9                  #  命令替换
10 echo `echo \z`      #  z
11 echo `echo \\z`     #  z
```

```
12 echo `echo \\\z`      # \z
13 echo `echo \\\\z`     # \z
14 echo `echo \\\\\\z`   # \z
15 echo `echo \\\\\\\z`  # \\z
16 echo `echo "\z"`      # \z
17 echo `echo "\\z"`     # \z
18
19                  # Here document
20 cat <<EOF
21 \z
22 EOF               # \z
23
24 cat <<EOF
25 \\z
26 EOF               # \z
```

##################################End Script##

分配给变量的字符串的元素也会被转义,但是只把一个转义符分配给变量将会报错.

##################################Start Script##

```
 1 variable=\
 2 echo "$variable"
 3 # Will not work - gives an error message:
 3 # 将不能正常工作- 将给出一个错误消息:
 4 # test.sh: : command not found
 5 # 一个"裸体的" 转义符将不能够安全的分配给变量.
 6 #
 7 #  What actually happens here is that the "\" escapes the newline and
 7 #  这里其实真正发生的是variable=\,这句被shell认为是没有完成,\被认为是一个续行符
 8 #+ 这样,下边的这句echo,也被认为是上一行的补充.所以,总的来说就是一个非法变量分配
 9
10 variable=\
11 23skidoo
12 echo "$variable"       #  23skidoo
13                   #  这句就可以使用,因为这是一个合法的变量分配
14
15 variable=\
16 #       \^   转义一个空格
17 echo "$variable"       # 显示空格
18
19 variable=\\
20 echo "$variable"       # \
21
22 variable=\\\
23 echo "$variable"
24 # 不能正常工作,给出一个错误
25 # test.sh: \: command not found
26 #
27 #  第一个转义符把第2个\转义了,但是第3个又变成"裸体的"了,
28 #+ 与上边的例子的原因相同
```

```
29
30 variable=\\\\
31 echo "$variable"       # \\
32                        # 转了两个\
33                        # 没问题
```

##################################End Script##

转义一个空格,在命令行参数列表中将会阻止单词分隔问题.

##################################Start Script##

```
 1 file_list="/bin/cat /bin/gzip /bin/more /usr/bin/less /usr/bin/emacs-20.7"
 2 # 列出的文件都作为命令的参数.
 3
 4 # Add two files to the list, and list all.
 4 # 加2个文件到list中,并且列出全部.
 5 ls -l /usr/X11R6/bin/xsetroot /sbin/dump $file_list
 6
 7 echo "------------------------------------------------------------------------"
 8
 9 # 如果我们转义2个空格,会发生什么?
10 ls -l /usr/X11R6/bin/xsetroot\ /sbin/dump\ $file_list
11 # 错误: 因为前3个路径名被合并成一个参数传给了'ls -l'
12 #       因为2个转义符阻止了参数(单词)分离
```

##################################End Script##

转义符也提供续行功能.一般,每一行都包含一个不同的命令,但如果在行尾加上\,那就会接受新行的输入,作为这一行的补充.

```
1 (cd /source/directory && tar cf - . ) | \
2 (cd /dest/directory && tar xpvf -)
3 # 重复了 Alan Cox的目录树拷贝命令
4 # 为了增加可读性分成2行.
5
6 # 也可以使用如下方式:
7 tar cf - -C /source/directory . |
8 tar xpvf - -C /dest/directory
9 # 察看下边的注意事项
```

注意:如果一个脚本以|(管道字符)结束.那么一个\(转义符),就不用非加上不可了.

但是一个好的shell脚本编写风格,还是应该在行尾加上\,以增加可读性.

##################################Start Script##

```
 1 echo "foo
 2 bar"
 3 #foo
 4 #bar
 5
 6 echo
 7
 8 echo 'foo
 9 bar'    # 没区别
10 #foo
11 #bar
12
```

```
13 echo
14
15 echo foo\
16 bar    # 续行
17 #foobar
18
19 echo
20
21 echo "foo\
22 bar"    # 与上边一样,\还是作为续行符
23 #foobar
24
25 echo
26
27 echo 'foo\
28 bar'    # 由于是强引用,所以\没被解释成续行符
29 #foo\
30 #bar
```

################################End Script###

第6章 退出和退出状态

==========

exit命令被用来结束脚本,就像C语言一样.他也会返回一个值来传给父进程,父进程会判断是否可用.

每个命令都会返回一个exit状态(有时候也叫return状态).成功返回0,如果返回一个非0值,通常情况下都会被认为是一个错误码.一个编写良好的UNIX命令,程序,和工具都会返回一个0作为退出码来表示成功,虽然偶尔也会有例外.

同样的,脚本中的函数和脚本本身都会返回退出状态.在脚本或者是脚本函数中执行的最后的命令会决定退出状态.在脚本中,exit nnn命令将会把nnn退出码传递给shell(nnn必须是10进制数0-255).

当一个脚本以不带参数exit来结束时,脚本的退出状态就由脚本中最后执行命令来决定.

```
 1 #!/bin/bash
 2
 3 COMMAND_1
 4
 5 . . .
 6
 7 # 将以最后的命令来决定退出状态
 8 COMMAND_LAST
 9
10 exit $?
```

```
1 #!/bin/bash
2
3 COMMAND1
4
5 . . .
6
7 # 将以最后的命令来决定退出状态
```

8 COMMAND_LAST
$?读取最后执行命令的退出码.函数返回后,$?给出函数最后执行的那条命令的退出码.这种给函数返回值的方法是Bash的方法.对于脚本来说也一样.总之,一般情况下,0为成功,非0失败W.
Example 6-1 exit/exit状态
################################Start Script###
```
 1 #!/bin/bash
 2
 3 echo hello
 4 echo $?    # 返回0,因为执行成功
 5
 6 lskdf      # 不认识的命令.
 7 echo $?    # 返回非0值,因为失败了.
 8
 9 echo
10
11 exit 113   # 将返回113给shell.
12           # To verify this, type "echo $?" after script terminates.
12           # 为了验证这个,在脚本结束的地方使用"echo $?"
```
################################End Script###
$?对于测试脚本中的命令的结果特别有用(见Example 12-32和Example 12-17).
注意: !逻辑非操作,将会反转test命令的结果,并且这会影响exit状态.
Example 6-2 否定一个条件使用!
################################Start Script###
```
 1 true  # true是shell内建命令,什么事都不做,就是shell返回0
 2 echo "exit status of \"true\" = $?"     # 0
 3
 4 ! true
 5 echo "exit status of \"! true\" = $?"   # 1
 6 # 注意:"!"需要一个空格
 7 #    !true   将导致一个"command not found"错误
 8 #
 9 # 如果一个命令以'!'开头,那么将使用Bash的历史机制.就是显示这个命令被使用的历史.
10
11 true
12 !true
13 # 这次就没有错误了.
14 # 他不过是重复了之前的命令(true).
```
################################End Script###
注意事项:
特定的退出码都有预定的含义(见附录D),用户不应该在自己的脚本中指定他.

第7章 Tests
==========
每个完整的合理的编程语言都具有条件判断的功能.Bash具有test命令,不同的[]和()操作,和if/then结构.
7.1 Test结构

一个if/then结构可以测试命令的返回值是否为0(因为0表示成功),如果是的话,执行更多命令.

有一个专用命令"["(左中括号,特殊字符).这个命令与test命令等价,但是出于效率上的考虑,
它是一个内建命令.这个命令把它的参数作为比较表达式或是文件测试,并且根据比较的结果,
返回一个退出码.
在版本2.02的Bash中,推出了一个新的[[...]]扩展test命令.因为这种表现形式可能对某些语
言的程序员来说更加熟悉.注意"[["是一个关键字,并不是一个命令.
Bash把[[$a -lt $b]]看作一个单独的元素,并且返回一个退出码.
((...))和let...结果也能够返回一个退出码,当它们所测试的算术表达式的结果为非0的时候,
他们的退出码将返回0.这些算术扩展(见第15章)结构被用来做算术比较.

```
1 let "1<2" returns 0 (as "1<2" expands to "1")
2 (( 0 && 1 )) returns 1 (as "0 && 1" expands to "0")
```

if命令可以测试任何命令,不仅仅是括号中的条件.

```
 1 if cmp a b &> /dev/null  # 阻止输出.
 2 then echo "Files a and b are identical."
 3 else echo "Files a and b differ."
 4 fi
 5
 6 # 非常有用的"if-grep" 结构:
 7 # ------------------------
 8 if grep -q Bash file
 9 then echo "File contains at least one occurrence of Bash."
10 fi
11
12 word=Linux
13 letter_sequence=inu
14 if echo "$word" | grep -q "$letter_sequence"
15 # "-q"选项是用来阻止输出
16 then
17   echo "$letter_sequence found in $word"
18 else
19   echo "$letter_sequence not found in $word"
20 fi
21
22
23 if COMMAND_WHOSE_EXIT_STATUS_IS_0_UNLESS_ERROR_OCCURRED
24 then echo "Command succeeded."
25 else echo "Command failed."
26 fi
```

一个if/then结构可以包含多级比较和tests.

```
 1 if echo "Next *if* is part of the comparison for the first *if*."
 2
 3   if [[ $comparison = "integer" ]]
 4     then (( a < b ))
 5   else
 6     [[ $a < $b ]]
 7   fi
 8
 9 then
10   echo '$a is less than $b'
```

11 fi

Example 7-1 什么情况下为真?

################################Start Script###

```
 1 #!/bin/bash
 2
 3 #  技巧:
 4 #  如果你不确定一个特定的条件如何判断.
 5 #+ 在一个if-test结构中测试它.
 6
 7 echo
 8
 9 echo "Testing \"0\""
10 if [ 0 ]     # zero
11 then
12   echo "0 is true."
13 else
14   echo "0 is false."
15 fi           # 0 is true.
16
17 echo
18
19 echo "Testing \"1\""
20 if [ 1 ]     # one
21 then
22   echo "1 is true."
23 else
24   echo "1 is false."
25 fi           # 1 is true.
26
27 echo
28
29 echo "Testing \"-1\""
30 if [ -1 ]    # -1
31 then
32   echo "-1 is true."
33 else
34   echo "-1 is false."
35 fi           # -1 is true.
36
37 echo
38
39 echo "Testing \"NULL\""
40 if [ ]       # NULL (控状态)
41 then
42   echo "NULL is true."
43 else
44   echo "NULL is false."
45 fi           # NULL is false.
```

```
46
47 echo
48
49 echo "Testing \"xyz\""
50 if [ xyz ]   # 字符串
51 then
52   echo "Random string is true."
53 else
54   echo "Random string is false."
55 fi          # Random string is true.
56
57 echo
58
59 echo "Testing \"\$xyz\""
60 if [ $xyz ]  # 测试$xyz是否为null,但是...(明显没人定义么!)
61            # 只不过是一个未定义的变量
62 then
63   echo "Uninitialized variable is true."
64 else
65   echo "Uninitialized variable is false."
66 fi          # Uninitialized variable is false.
67
68 echo
69
70 echo "Testing \"-n \$xyz\""
71 if [ -n "$xyz" ]          # 更学究的的检查
72 then
73   echo "Uninitialized variable is true."
74 else
75   echo "Uninitialized variable is false."
76 fi          # Uninitialized variable is false.
77
78 echo
79
80
81 xyz=        # 初始化了,但是将其设为空值
82
83 echo "Testing \"-n \$xyz\""
84 if [ -n "$xyz" ]
85 then
86   echo "Null variable is true."
87 else
88   echo "Null variable is false."
89 fi          # Null variable is false.
90
91
92 echo
93
```

```
 94
 95 # 什么时候"flase"为true?
 96
 97 echo "Testing \"false\""
 98 if [ "false" ]             #  看起来"false"只不过是个字符串而已.
 99 then
100   echo "\"false\" is true." #+ 并且它test的结果就是true.
101 else
102   echo "\"false\" is false."
103 fi            # "false" is true.
104
105 echo
106
107 echo "Testing \"\$false\""  # 再来一个,未声明的变量
108 if [ "$false" ]
109 then
110   echo "\"\$false\" is true."
111 else
112   echo "\"\$false\" is false."
113 fi            # "$false" is false.
114               # 现在我们终于得到了期望的结果
115
116 #  如果我们test这个变量"$true"会发生什么结果?答案是和"$flase"一样,都为空,因为我
117 #+ 们并没有定义它.
118 echo
119
120 exit 0
```

##################################End Script##

练习.解释上边例子的行为(我想我解释的已经够清楚了)

```
 1 if [ condition-true ]
 2 then
 3    command 1
 4    command 2
 5    ...
 6 else
 7    # 可选的(如果不需要可以省去)
 8    # 如果原始的条件测试结果是false,那么添加默认的代码来执行.
 9    command 3
10    command 4
11    ...
12 fi
```

注意:当if和then在一个条件测试的同一行中的话,必须使用";"来终止if表达式.if和then都是关键字.关键字(或者命令)作为一个表达式的开头,并且在一个新的表达式开始之前,必须结束上一个表达式.

```
 1 if [ -x "$filename" ]; then
```

Else if和elif

elif

elif是else if的缩减形式.

```
 1 if [ condition1 ]
 2 then
 3    command1
 4    command2
 5    command3
 6 elif [ condition2 ]
 7 # Same as else if
 8 then
 9    command4
10    command5
11 else
12    default-command
13 fi
```

使用if test condition-true这种形式和if[condition-true]这种形式是等价的.向我们前边所说的"["是test的标记.并且以"]"结束.在if/test中并不应该这么严厉,但是新版本的Bash需要它.

注意:test命令是Bash的内建命令,用来测试文件类型和比较字符串.因此,在Bash脚本中,test并不调用/usr/bin/test的二进制版本(这是sh-utils工具包的一部分).同样的,[并不调用/usr/bin/[,被连接到/usr/bin/test.

```
bash$ type test
test is a shell builtin
bash$ type '['
[ is a shell builtin
bash$ type '[['
[[ is a shell keyword
bash$ type ']]'
]] is a shell keyword
bash$ type ']'
bash: type: ]: not found
```

Example 7-2 几个等效命令test,/usr/bin/test,[],和/usr/bin/[

```
################################Start Script#########################################
 1 #!/bin/bash
 2
 3 echo
 4
 5 if test -z "$1"
 6 then
 7   echo "No command-line arguments."
 8 else
 9   echo "First command-line argument is $1."
10 fi
11
12 echo
13
14 if /usr/bin/test -z "$1"      # 与内建的test结果相同
15 then
16   echo "No command-line arguments."
17 else
```

```
 18   echo "First command-line argument is $1."
 19 fi
 20
 21 echo
 22
 23 if [ -z "$1" ]              # 与上边代码的作用相同
 24 #   if [ -z "$1"              应该工作,但是...
 25 #+  Bash相应一个缺少关闭中括号的错误消息.
 26 then
 27   echo "No command-line arguments."
 28 else
 29   echo "First command-line argument is $1."
 30 fi
 31
 32 echo
 33
 34
 35 if /usr/bin/[ -z "$1" ]      # 再来一个,与上边代码的作用相同
 36 # if /usr/bin/[ -z "$1"       # 工作,但是给个错误消息
 37 #                           # 注意:
 38 #                             This has been fixed in Bash, version 3.x.
 38 #                             在ver 3.x上,这个bug已经被Bash修正了.
 39 then
 40   echo "No command-line arguments."
 41 else
 42   echo "First command-line argument is $1."
 43 fi
 44
 45 echo
 46
 47 exit 0
```

################################End Script###

[[]]结构比Bash的[]更加灵活,这是一个扩展的test命令,从ksh88继承过来的.

注意:在[[]]结构中,将没有文件扩展或者是单词分离,但是会发生参数扩展和命令替换.

```
 1 file=/etc/passwd
 2
 3 if [[ -e $file ]]
 4 then
 5   echo "Password file exists."
 6 fi
```

注意:使用[[]],而不是[],能够阻止脚本中的许多逻辑错误.比如,尽管在[]中将给出一个错误,
但是&&,||,<>操作还是能够工作在一个[[]]test之中.

注意:在if后边,test命令和[]或[[]]都不是必须的.如下:

```
 1 dir=/home/bozo
 2
 3 if cd "$dir" 2>/dev/null; then   # "2>/dev/null" hides error message.
 4   echo "Now in $dir."
 5 else
```

```
 6   echo "Can't change to $dir."
 7 fi
```

if命令将返回if后边的命令的退出码.

与此相似,当在一个在使用与或列表结构的时候,test或中括号的使用,也并不一定非的有if不可

```
 1 var1=20
 2 var2=22
 3 [ "$var1" -ne "$var2" ] && echo "$var1 is not equal to $var2"
 4
 5 home=/home/bozo
 6 [ -d "$home" ] || echo "$home directory does not exist."
```

(())结构扩展并计算一个算术表达式的结果.如果表达式的结果为0,它将返回1作为退出码,或者是"false".而一个非0表达式的结果将返回0作为退出码,或者是"true".

Example 7-3 算数测试使用(( ))

##################################Start Script###

```
 1 #!/bin/bash
 2 # 算数测试
 3
 4 # The (( ... )) construct evaluates and tests numerical expressions.
 4 # (( ... ))结构计算并测试算数表达式的结果.
 5 # 退出码将与[ ... ]结构相反!
 6
 7 (( 0 ))
 8 echo "Exit status of \"(( 0 ))\" is $?."        # 1
 9
10 (( 1 ))
11 echo "Exit status of \"(( 1 ))\" is $?."        # 0
12
13 (( 5 > 4 ))                                     # true
14 echo "Exit status of \"(( 5 > 4 ))\" is $?."    # 0
15
16 (( 5 > 9 ))                                     # false
17 echo "Exit status of \"(( 5 > 9 ))\" is $?."    # 1
18
19 (( 5 - 5 ))                                     # 0
20 echo "Exit status of \"(( 5 - 5 ))\" is $?."    # 1
21
22 (( 5 / 4 ))                                     # 除法也行
23 echo "Exit status of \"(( 5 / 4 ))\" is $?."    # 0
24
25 (( 1 / 2 ))                                     # 出发结果<1
26 echo "Exit status of \"(( 1 / 2 ))\" is $?."    # 结果将为0
27                                                 # 1
28
29 (( 1 / 0 )) 2>/dev/null                         # 除数为0的错误
30 #           ^^^^^^^^^^^
31 echo "Exit status of \"(( 1 / 0 ))\" is $?."    # 1
32
33 # What effect does the "2>/dev/null" have?
```

33 # "2>/dev/null"的作用是什么?
34 # 如果删除"2>dev/null"将会发生什么?
35 # Try removing it, then rerunning the script.
35 # 尝试删除它,然后再运行脚本.
36
37 exit 0
################################End Script###
7.2 文件测试操作

返回true如果...
-e 文件存在
-a 文件存在
这个选项的效果与-e相同.但是它已经被弃用了,并且不鼓励使用
-f file是一个regular文件(不是目录或者设备文件)
-s 文件长度不为0
-d 文件是个目录
-b 文件是个块设备(软盘,cdrom等等)
-c 文件是个字符设备(键盘,modem,声卡等等)
-p 文件是个管道
-h 文件是个符号链接
-L 文件是个符号链接
-S 文件是个socket
-t 关联到一个终端设备的文件描述符
这个选项一般都用来检测是否在一个给定脚本中的stdin[-t0]或[-t1]是一个终端
-r 文件具有读权限(对于用户运行这个test)
-w 文件具有写权限(对于用户运行这个test)
-x 文件具有执行权限(对于用户运行这个test)
-g set-group-id(sgid)标志到文件或目录上
如果一个目录具有sgid标志,那么一个被创建在这个目录里的文件,这个目录属于创建
这个目录的用户组,并不一定与创建这个文件的用户的组相同.对于workgroup的目录
共享来说,这非常有用.见<<UNIX环境高级编程中文版>>第58页.
-u set-user-id(suid)标志到文件上
如果运行一个具有root权限的文件,那么运行进程将取得root权限,即使你是一个普通
用户.[1]这对于需要存取系统硬件的执行操作(比如pppd和cdrecord)非常有用.如果
没有suid标志的话,那么普通用户(没有root权限)将无法运行这种程序.
见<<UNIX环境高级编程中文版>>第58页.
-rwsr-xr-t 1 root 178236 Oct 2 2000 /usr/sbin/pppd
对于设置了suid的文件,在它的权限标志中有"s".
-k 设置粘贴位,见<<UNIX环境高级编程中文版>>第65页.
对于"sticky bit",save-text-mode标志是一个文件权限的特殊类型.如果设置了这
个标志,那么这个文件将被保存在交换区,为了达到快速存取的目的.如果设置在目录
中,它将限制写权限.对于设置了sticky bit位的文件或目录,权限标志中有"t".
drwxrwxrwt 7 root 1024 May 19 21:26 tmp/
如果一个用户并不时具有stick bit位的目录的拥有者,但是具有写权限,那么用户只
能在这个目录下删除自己所拥有的文件.这将防止用户在一个公开的目录中不慎覆盖
或者删除别人的文件,比如/tmp(当然root或者是目录的所有者可以随便删除或重命名
其中的文件).
-O 你是文件的所有者.

-G 文件的group-id和你的相同.
-N 从文件最后被阅读到现在,是否被修改.
f1 -nt f2
文件f1比f2新
f1 -ot f2
f1比f2老
f1 -ef f2
f1和f2都硬连接到同一个文件.
! 非--反转上边测试的结果(如果条件缺席,将返回true)
Example 7-4 test死的链接文件
################################Start Script###

```
 1 #!/bin/bash
 2 # broken-link.sh
 3 # Written by Lee bigelow <ligelowbee@yahoo.com>
 4 # Used with permission.
 5
 6 #一个真正有用的shell脚本来找出死链接文件并且输出它们的引用
 7 #以便于它们可以被输入到xargs命令中进行处理 :)
 8 #比如: broken-link.sh /somedir /someotherdir|xargs rm
 9 #
10 #这里,不管怎么说,是一种更好的方法
11 #
12 #find "somedir" -type l -print0|\
13 #xargs -r0 file|\
14 #grep "broken symbolic"|
15 #sed -e 's/^\|: *broken symbolic.*$/"/g'
16 #
17 #但这不是一个纯粹的bash,最起码现在不是.
18 #小心:小心/proc文件系统和任何的循环链接文件.
19 ######################################################################
20
21
22 #如果没对这个脚本传递参数,那么就使用当前目录.
23 #否则就使用传递进来的参数作为目录来搜索.
24 #
25 ####################
26 [ $# -eq 0 ] && directorys=`pwd` || directorys=$@
27
28 #建立函数linkchk来检查传进来的目录或文件是否是链接和是否存在,
29 #并且打印出它们的引用
30 #如果传进来的目录有子目录,
31 #那么把子目录也发送到linkchk函数中处理,就是递归目录.
32 ##########
33 linkchk () {
34     for element in $1/*; do
35     [ -h "$element" -a ! -e "$element" ] && echo \"$element\"
36     [ -d "$element" ] && linkchk $element
37     # Of course, '-h' tests for symbolic link, '-d' for directory.
```

```
37    # 当然'-h'是测试链接,'-d'是测试目录.
38    done
39 }
40
41 #如果是个可用目录,那就把每个从脚本传递进来的参数都送到linkche函数中.
42 #如果不是,那就打印出错误消息和使用信息.
43 #
44 #################
45 for directory in $directorys; do
46    if [ -d $directory ]
47  then linkchk $directory
48  else
49     echo "$directory is not a directory"
50     echo "Usage: $0 dir1 dir2 ..."
51    fi
52 done
53
54 exit 0
```

################################End Script###

Example 28-1, Example 10-7, Example 10-3, Example 28-3, 和Example A-1 也会说明文件测试操作的使用过程.

注意事项:

[1] 小心suid,可能引起安全漏洞,但是不会影响shell脚本.

[2] 在当代UNIX系统中,已经不使用sticky bit了,只在目录中使用.

7.3 其他比较操作

二元比较操作符,比较变量或者比较数字.注意数字与字符串的区别.

整数比较

-eq 等于,如:if ["$a" -eq "$b"]

-ne 不等于,如:if ["$a" -ne "$b"]

-gt 大于,如:if ["$a" -gt "$b"]

-ge 大于等于,如:if ["$a" -ge "$b"]

-lt 小于,如:if ["$a" -lt "$b"]

-le 小于等于,如:if ["$a" -le "$b"]

< 小于(需要双括号),如:(("$a" < "$b"))

<= 小于等于(需要双括号),如:(("$a" <= "$b"))

> 大于(需要双括号),如:(("$a" > "$b"))

>= 大于等于(需要双括号),如:(("$a" >= "$b"))

字符串比较

= 等于,如:if ["$a" = "$b"]

== 等于,如:if ["$a" == "$b"],与=等价

注意:==的功能在[[]]和[]中的行为是不同的,如下:

```
1 [[ $a == z* ]]   # 如果$a以"z"开头(模式匹配)那么将为true
2 [[ $a == "z*" ]]  # 如果$a等于z*(字符匹配),那么结果为true
3
4 [ $a == z* ]     # File globbing 和word splitting将会发生
5 [ "$a" == "z*" ]  # 如果$a等于z*(字符匹配),那么结果为true
```

一点解释,关于File globbing是一种关于文件的速记法,比如"*.c"就是,再如~也是.

但是file globbing并不是严格的正则表达式,虽然绝大多数情况下结构比较像.

!= 不等于,如:if ["$a" != "$b"]

这个操作符将在[[]]结构中使用模式匹配.

< 小于,在ASCII字母顺序下.如:

if [["$a" < "$b"]]

if ["$a" \< "$b"]

注意:在[]结构中"<"需要被转义.

> 大于,在ASCII字母顺序下.如:

if [["$a" > "$b"]]

if ["$a" \> "$b"]

注意:在[]结构中">"需要被转义.

具体参考Example 26-11来查看这个操作符应用的例子.

-z 字符串为"null".就是长度为0.

-n 字符串不为"null"

注意:

使用-n在[]结构中测试必须要用""把变量引起来.使用一个未被""的字符串来使用! -z

或者就是未用""引用的字符串本身,放到[]结构中(见Example 7-6)虽然一般情况下可

以工作,但这是不安全的.习惯于使用""来测试字符串是一种好习惯.[1]

Example 7-5 数字和字符串比较

```
#################################Start Script#########################################
 1 #!/bin/bash
 2
 3 a=4
 4 b=5
 5
 6 #  这里的变量a和b既可以当作整型也可以当作是字符串.
 7 #  这里在算术比较和字符串比较之间有些混淆,
 8 #+ 因为Bash变量并不是强类型的.
 9
10 #  Bash允许对整型变量操作和比较
11 #+ 当然变量中只包含数字字符.
12 #  但是还是要考虑清楚再做.
13
14 echo
15
16 if [ "$a" -ne "$b" ]
17 then
18   echo "$a is not equal to $b"
19   echo "(arithmetic comparison)"
20 fi
21
22 echo
23
24 if [ "$a" != "$b" ]
25 then
26   echo "$a is not equal to $b."
27   echo "(string comparison)"
28   #     "4"  != "5"
```

```
29   # ASCII 52 != ASCII 53
30 fi
31
32 # 在这个特定的例子中,"-ne"和"!="都可以.
33
34 echo
35
36 exit 0
```

################################End Script###

Example 7-6 测试字符串是否为null

################################Start Script###

```
 1 #!/bin/bash
 2 #  str-test.sh: 测试null字符串和非引用字符串,
 3 #+ but not strings and sealing wax, not to mention cabbages and kings . . .
 4 #+ 上边这句没看懂
 5 # Using   if [ ... ]
 6
 7
 8 # 如果一个字符串没被初始化,那么它就没有定义的值(像这种话,总感觉像屁话)
 9 # 这种状态叫做"null"(与zero不同)
10
11 if [ -n $string1 ]    # $string1 没被声明和初始化
12 then
13   echo "String \"string1\" is not null."
14 else
15   echo "String \"string1\" is null."
16 fi
17 # 错误的结果.
18 # 显示$string1为非空,虽然他没被初始化.
19
20
21 echo
22
23
24 # 让我们再试一下.
25
26 if [ -n "$string1" ]  # 这次$string1被引用了.
27 then
28   echo "String \"string1\" is not null."
29 else
30   echo "String \"string1\" is null."
31 fi                    # ""的字符串在[]结构中
32
33
34 echo
35
36
37 if [ $string1 ]       # 这次$string1变成"裸体"的了
```

```
38 then
39   echo "String \"string1\" is not null."
40 else
41   echo "String \"string1\" is null."
42 fi
43 # 这工作得很好.
44 # 这个[]test操作检测string是否为null.
45 # 然而,使用("$string1")是一种很好的习惯
46 #
47 # As Stephane Chazelas points out,
48 #    if [ $string1 ]    有1个参数 "]"
49 #    if [ "$string1" ]  有2个参数,空的"$string1"和"]"
50
51
52
53 echo
54
55
56
57 string1=initialized
58
59 if [ $string1 ]       # 再来,$string1"裸体了"
60 then
61   echo "String \"string1\" is not null."
62 else
63   echo "String \"string1\" is null."
64 fi
65 # 再来,给出了正确的结果.
66 # 不过怎么说("$string1")还是好很多,因为. . .
67
68
69 string1="a = b"
70
71 if [ $string1 ]       # 再来,$string1 再次裸体了.
72 then
73   echo "String \"string1\" is not null."
74 else
75   echo "String \"string1\" is null."
76 fi
77 # 非引用的"$string1"现在给出了一个错误的结果!
78
79 exit 0
80 # Thank you, also, Florian Wisser, for the "heads-up".
```

################################End Script###

Example 7-7 zmore

################################Start Script###

```
 1 #!/bin/bash
 2 # zmore
```

```
 3
 4 #使用'more'来查看gzip文件
 5
 6 NOARGS=65
 7 NOTFOUND=66
 8 NOTGZIP=67
 9
10 if [ $# -eq 0 ] # 与 if [ -z "$1" ]同样的效果
11 # 应该是说前边的那句注释有问题,$1是可以存在的,比如:zmore "" arg2 arg3
12 then
13   echo "Usage: `basename $0` filename" >&2
14   # 错误消息到stderr
15   exit $NOARGS
16   # 脚本返回65作为退出码.
17 fi
18
19 filename=$1
20
21 if [ ! -f "$filename" ]   # 将$filename ""起来,来允许可能的空白
22 then
23   echo "File $filename not found!" >&2
24    # 错误消息到stderr
25   exit $NOTFOUND
26 fi
27
28 if [ ${filename##*.} != "gz" ]
29 # 在变量替换中使用中括号
30 then
31   echo "File $1 is not a gzipped file!"
32   exit $NOTGZIP
33 fi
34
35 zcat $1 | more
36
37 # 使用过滤命令'more'
38 # 如果你想的话也可使用'less'
39
40
41 exit $?   # 脚本将返回pipe的结果作为退出码
42 # 事实上,不用非的有"exit $?",但是不管怎么说,有了这句,能正规一些
43 # 将最后一句命令的执行状态作为退出码返回
```

################################End Script###

混合比较

-a 逻辑与

exp1 -a exp2 如果exp1和exp2都为true的话,这个表达式将返回true

-o 逻辑或

exp1 -o exp2 如果exp1和exp2中有一个为true的话,那么这个表达式就返回true

这与Bash的比较操作符&&和||很相像.在[[]]中使用它.

```
 1 [[ condition1 && condition2 ]]
```

-o和-a一般都是和test命令或者是[]一起工作.

```
 1 if [ "$exp1" -a "$exp2" ]
```

请参考Example 8-3,Example 26-16和Example A-28来查看混合比较操作的行为.

注意事项:

[1] S.C.(这家伙是个人名)指出,在使用混合比较的时候即使"$var"也可能会产生问题.
如果$string为空的话,[-n "$string" -o "$a" = "$b"]可能在某些版本的Bash中
会有问题.为了附加一个额外的字符到可能的空变量中的一种安全的办法是,
["x$string" != x -o "x$a" = "x$b"](the "x's" cancel out)(没看懂).
cancel out是抵消的意思.

7.4 嵌套的if/then条件test

可以使用if/then来进行嵌套的条件test.最终的结果和上边的使用&&混合比较操作是相同的.

```
 1 if [ condition1 ]
 2 then
 3   if [ condition2 ]
 4   then
 5     do-something  # 这里只有在condition1和condition2都可用的时候才行.
 6   fi
 7 fi
```

具体请查看Example 34-4.

7.5 检查你的test知识

系统范围的xinitrc文件可以用来启动X server.这个文件中包含了相当多的if/then test,
就像下边的节选一样:

```
 1 if [ -f $HOME/.Xclients ]; then
 2   exec $HOME/.Xclients
 3 elif [ -f /etc/X11/xinit/Xclients ]; then
 4   exec /etc/X11/xinit/Xclients
 5 else
 6     # 故障保险设置,虽然我们永远都不会走到这来.
 7     # (我们在Xclients中也提供了相同的机制)它不会受伤的.
 8     xclock -geometry 100x100-5+5 &
 9     xterm -geometry 80x50-50+150 &
10     if [ -f /usr/bin/netscape -a -f /usr/share/doc/HTML/index.html ]; then
11           netscape /usr/share/doc/HTML/index.html &
12     fi
13 fi
```

对上边的"test"结构进行解释,然后检查整个文件,/etc/X11/xinit/xinitrc,并分析if/then
test结构.你可能需要查看一下后边才能讲解到的grep,sed和正则表达式的知识.

第8章 操作符和相关的主题

==========

8.1 操作符

等号操作符

变量赋值

初始化或者修改变量的值

=

无论在算术运算还是字符串运算中,都是赋值语句.

```
1 var=27
2 category=minerals  # No spaces allowed after the "=".
```

注意:不要和"="test操作符混淆.

```
1 #    = as a test operator
2
3 if [ "$string1" = "$string2" ]
4 # if [ "X$string1" = "X$string2" ] is safer,
5 # to prevent an error message should one of the variables be empty.
6 # (The prepended "X" characters cancel out.)
7 then
8    command
9 fi
```

算术操作符

+ 加法

- 减法

* 乘法

/ 除法

** 幂运算

```
1 # Bash, version 2.02, introduced the "**" exponentiation operator.
2
3 let "z=5**3"
4 echo "z = $z"   # z = 125
```

% 取模

```
bash$ expr 5 % 3
2
```

5/3=1余2

模运算经常用在其它的事情中,比如产生特定的范围的数字(Example 9-24, Example 9-27)和格式化程序的输出(Example 26-15,Example A-6).它甚至可以用来产生质数,(Example A-16).事实上取模运算在算术运算中使用的频率惊人的高.

Example 8-1 最大公约数

##################################Start Script###

```
1 #!/bin/bash
2 # gcd.sh: 最大公约数
3 #         使用Euclid's 算法
4
5 #  最大公约数,就是2个数能够同时整除的最大的数.
6 #
7
8 #  Euclid's算法采用连续除法.
9 #  在每个循环中
10 #+ 被除数 <---  除数
11 #+ 除数 <---  余数
12 #+ 直到余数= 0.
13 #+ 在最后的循环中The gcd = 被除数
14 #
15 #  关于这个算法更精彩的讨论
```

```
16 #  见Jim Loy's site, http://www.jimloy.com/number/euclids.htm.
17
18
19 # ------------------------------------------------------
20 # 参数检查
21 ARGS=2
22 E_BADARGS=65
23
24 if [ $# -ne "$ARGS" ]
25 then
26   echo "Usage: `basename $0` first-number second-number"
27   exit $E_BADARGS
28 fi
29 # ------------------------------------------------------
30
31
32 gcd ()
33 {
34
35   dividend=$1                #  随便给值
36   divisor=$2                 #+ 即使$2大,也没关系.
37                              #  Why not?
38
39   remainder=1                #  如果再循环中使用为初始化的变量.
40                              #+ 那将在第一次循环中产生一个错误消息.
41
42
43   until [ "$remainder" -eq 0 ]
44   do
45     let "remainder = $dividend % $divisor"
46     dividend=$divisor          # 现在使用2个最小的数重复.
47     divisor=$remainder
48   done                       # Euclid's algorithm
49
50 }                            # Last $dividend is the gcd.
50 }                            # 最后的$dividend就是gcd.
51
52
53 gcd $1 $2
54
55 echo; echo "GCD of $1 and $2 = $dividend"; echo
56
57
58 # 练习:
59 # --------
60 #  检查命令行参数来确定它们都是整数,
61 #+ and exit the script with an appropriate error message if not.
61 #+ 否则就选择合适的错误消息退出.
```

```
62
63 exit 0
################################End Script#########################################
```

+= 加等于(通过常量增加变量)

```
 let "var += 5"  #var将在本身值的基础上增加5
```

-= 减等于

*= 乘等于

```
 let "var *= 4"
```

/= 除等于

%= 取模赋值,算术操作经常使用expr或者let表达式.

Example 8-2 使用算术操作符

```
################################Start Script#######################################
 1 #!/bin/bash
 2 # Counting to 11 in 10 different ways.
 3
 4 n=1; echo -n "$n "
 5
 6 let "n = $n + 1"   # let "n = n + 1"  这么写也行
 7 echo -n "$n "
 8
 9
10 : $((n = $n + 1))
11 #  ":" 是必须的,这是因为,如果没有":"的话,Bash将
12 #+ 尝试把"$((n = $n + 1))"解释成一个命令
13 echo -n "$n "
14
15 (( n = n + 1 ))
16 #  对于上边的方法的一个更简单的选则.
17 #  Thanks, David Lombard, for pointing this out.
18 echo -n "$n "
19
20 n=$(($n + 1))
21 echo -n "$n "
22
23 : $[ n = $n + 1 ]
24 #  ":" 是必须的,这是因为,如果没有":"的话,Bash将
25 #+ 尝试把"$[ n = $n + 1 ]" 解释成一个命令
26 #  即使"n"被初始化成为一个字符串,这句也能工作.
27 echo -n "$n "
28
29 n=$[ $n + 1 ]
30 #  即使"n"被初始化成为一个字符串,这句也能工作.
31 #* Avoid this type of construct, since it is obsolete and nonportable.
31 #* 尽量避免这种类型的结果,因为这已经被废弃了,并且不具可移植性.
32 #  Thanks, Stephane Chazelas.
33 echo -n "$n "
34
35 # 现在来个C风格的增量操作.
```

```
36 # Thanks, Frank Wang, for pointing this out.
37
38 let "n++"          # let "++n"  also works.
39 echo -n "$n "
40
41 (( n++ ))         # (( ++n )  also works.
42 echo -n "$n "
43
44 : $(( n++ ))      # : $(( ++n )) also works.
45 echo -n "$n "
46
47 : $[ n++ ]        # : $[ ++n ]] also works
48 echo -n "$n "
49
50 echo
51
52 exit 0
```

##################################End Script##

注意:在Bash中的整型变量事实上是32位的,范围是 -2147483648 到2147483647.如果超过这个范围进行算术操作,将不会得到你期望的结果(就是溢出么).

```
1 a=2147483646
2 echo "a = $a"      # a = 2147483646
3 let "a+=1"         # 加1 "a".
4 echo "a = $a"      # a = 2147483647
5 let "a+=1"         # 再加1 "a" ,将超过上限了.
6 echo "a = $a"      # a = -2147483648
7                    #     错误 (溢出了)
```

在Bash 2.05b版本中,Bash支持64位整型了.

注意:Bash并不能理解浮点运算.它把包含的小数点看作字符串.

```
1 a=1.5
2
3 let "b = $a + 1.3"  # 错误.
4 # t2.sh: let: b = 1.5 + 1.3: 表达式的语义错误(错误标志为".5 + 1.3")
5
6 echo "b = $b"       # b=1
```

如果真想做浮点运算的话,使用bc(见12.8节),bc可以进行浮点运算或调用数学库函数.

位操作符.

(晕,有点强大过分了吧,位级操作都支持.)

位操作符在shell脚本中极少使用.它们最主要的用途看起来就是操作和test从sockets中读出的变量."Bit flipping"与编译语言的联系很紧密,比如c/c++,在这种语言中它可以运行得足够快.(原文有处on the fly,我查了一下,好像是没事干的意思,没理解)

<< 左移1位(每次左移都将乘2)

<<= 左移几位,=号后边将给出左移几位

 let "var <<= 2"就是左移2位(就是乘4)

>> 右移1位(每次右移都将除2)

>>= 右移几位

& 按位与

&= 按位与赋值

| 按位或
|= 按位或赋值
~ 按位非
! 按位否?(没理解和上边的~有什么区别?),感觉是应该放到下边的逻辑操作中
^ 按位异或XOR
^= 异或赋值
逻辑操作:
&& 逻辑与

```
1 if [ $condition1 ] && [ $condition2 ]
2 # 与:  if [ $condition1 -a $condition2 ] 相同
3 # 如果condition1和condition2都为true,那结果就为true.
4
5 if [[ $condition1 && $condition2 ]]   # 也可以.
6 # 注意&&不允许出现在[ ... ]中.
```

注意:&&也可以用在and list中(见25章),但是使用的时候需要依赖上下文.
|| 逻辑或

```
1 if [ $condition1 ] || [ $condition2 ]
2 # 与:  if [ $condition1 -o $condition2 ] 相同
3 # 如果condition1或condition2为true,那结果就为true.
4
5 if [[ $condition1 || $condition2 ]]   # 也可以
6 # 注意||不允许出现在[ ... ]中.
```

注意:Bash将test每个连接到逻辑操作的状态的退出状态(见第6章).
Example 8-3 使用&&和||进行混合状态的test
##################################Start Script###

```
1 #!/bin/bash
2
3 a=24
4 b=47
5
6 if [ "$a" -eq 24 ] && [ "$b" -eq 47 ]
7 then
8   echo "Test #1 succeeds."
9 else
10   echo "Test #1 fails."
11 fi
12
13 # 错误:   if [ "$a" -eq 24 && "$b" -eq 47 ]
14 #+        尝试执行' [ "$a" -eq 24 '
15 #+        因为没找到']'所以失败了.
16 #
17 #  注意:  如果 [[ $a -eq 24 && $b -eq 24 ]]  能够工作.
18 #  那这个[[]]的test结构就比[]结构更灵活了.
19 #
20 #    (在17行的"&&"与第6行的"&&"意义不同)
21 #    Thanks, Stephane Chazelas, for pointing this out.
22
23
```

```
24 if [ "$a" -eq 98 ] || [ "$b" -eq 47 ]
25 then
26   echo "Test #2 succeeds."
27 else
28   echo "Test #2 fails."
29 fi
30
31
32 #  -a和-o选项提供了
33 #+ 一种可选的混合test方法.
34 #  Thanks to Patrick Callahan for pointing this out.
35
36
37 if [ "$a" -eq 24 -a "$b" -eq 47 ]
38 then
39   echo "Test #3 succeeds."
40 else
41   echo "Test #3 fails."
42 fi
43
44
45 if [ "$a" -eq 98 -o "$b" -eq 47 ]
46 then
47   echo "Test #4 succeeds."
48 else
49   echo "Test #4 fails."
50 fi
51
52
53 a=rhino
54 b=crocodile
55 if [ "$a" = rhino ] && [ "$b" = crocodile ]
56 then
57   echo "Test #5 succeeds."
58 else
59   echo "Test #5 fails."
60 fi
61
62 exit 0
```

################################End Script###

&&和||操作也能在算术运算的上下文中找到.

```
bash$ echo $(( 1 && 2 )) $((3 && 0)) $((4 || 0)) $((0 || 0))
1 0 1 0
```

混杂操作:

, 逗号操作符

逗号操作符可以连接2个或多个算术运算.所有的操作都会被执行,但是只有最后一个操作作为结果.

```
1 let "t1 = ((5 + 3, 7 - 1, 15 - 4))"
```

```
 2 echo "t1 = $t1"              # t1 = 11
 3
 4 let "t2 = ((a = 9, 15 / 3))"  # Set "a" and calculate "t2".
 5 echo "t2 = $t2    a = $a"     # t2 = 5    a = 9
```

","主要用在for循环中,具体见Example 10-12.

8.2 数字常量

---

shell脚本默认都是将数字作为10进制数处理,除非这个数字某种特殊的标记法或前缀开头.
以0开头就是8进制.以0x开头就是16进制数.使用BASE#NUMBER这种形式可以表示其它进制表示法

Example 8-4 数字常量的处理

###################################Start Script###

```
 1 #!/bin/bash
 2 # numbers.sh: 数字常量的几种不同的表示法
 3
 4 # 10进制: 默认
 5 let "dec = 32"
 6 echo "decimal number = $dec"            # 32
 7 # 一切都很正常
 8
 9
10 # 8进制: 以'0'(零)开头
11 let "oct = 032"
12 echo "octal number = $oct"              # 26
13 # 表达式的结果用10进制表示.
14 #
15
16 # 16进制表示:数字以'0x'或者'0X'开头
17 let "hex = 0x32"
18 echo "hexadecimal number = $hex"        # 50
19 # 表达式的结果用10进制表示.
20
21 # 其它进制: BASE#NUMBER
22 # BASE between 2 and 64.
22 # 2到64进制都可以.
23 # NUMBER必须在BASE的范围内,具体见下边.
24
25
26 let "bin = 2#111100111001101"
27 echo "binary number = $bin"             # 31181
28
29 let "b32 = 32#77"
30 echo "base-32 number = $b32"            # 231
31
32 let "b64 = 64#@_"
33 echo "base-64 number = $b64"            # 4031
34 # 这种64进制的表示法中的每位数字都必须在64进制表示法的限制字符内.
35 # 10 个数字+ 26 个小写字母+ 26 个大写字母+ @ + _
```

```
36
37
38 echo
39
40 echo $((36#zz)) $((2#10101010)) $((16#AF16)) $((53#1aA))
41                                    # 1295 170 44822 3375
42
43
44 #  重要的注意事项:
45 #  ---------------
46 #  如果使用的每位数字超出了这个进制表示法规定字符的范围的话,
47 #+ 将给出一个错误消息.
48
49 let "bad_oct = 081"
50 # (部分的) 错误消息输出:
51 #  bad_oct = 081: too great for base (error token is "081")
52 #             Octal numbers use only digits in the range 0 - 7.
53
54 exit 0       # Thanks, Rich Bartell and Stephane Chazelas, for clarification.
```

```
##################################End Script#############################################
[]
```

高级Bash脚本编程指南(二)
文章整理: 文章来源: 网络
高级Bash脚本编程指南(二)

第三部分 超越基本
++++++++++++++++++++
第9章 变量重游
================
如果变量使用恰当,将会增加脚本的能量和灵活性.但是前提是这需要仔细学习变量的细节知识.
9.1 内部变量

Builtin variable
 这些内建的变量,将影响bash脚本的行为.
 $BASH
  这个变量将指向Bash的二进制执行文件的位置.
  bash$ echo $BASH
  /bin/bash
 $BASH_ENV
  这个环境变量将指向一个Bash启动文件,这个启动文件将在调用一个脚本时被读取.
 $BASH_SUBSHELL
  这个变量将提醒subshell的层次,这是一个在version3才被添加到Bash中的新特性.
  见Example 20-1.
 $BASH_VERSINFO[n]
  记录Bash安装信息的一个6元素的数组.与下边的$BASH_VERSION很像,但这个更加详细.
   1 # Bash version info:

```
 2
 3 for n in 0 1 2 3 4 5
 4 do
 5   echo "BASH_VERSINFO[$n] = ${BASH_VERSINFO[$n]}"
 6 done
 7
 8 # BASH_VERSINFO[0] = 3                  # 主版本号
 9 # BASH_VERSINFO[1] = 00                 # 次版本号
10 # BASH_VERSINFO[2] = 14                 # Patch 次数.
11 # BASH_VERSINFO[3] = 1                  # Build version.
12 # BASH_VERSINFO[4] = release            # Release status.
13 # BASH_VERSINFO[5] = i386-redhat-linux-gnu  # Architecture
```

$BASH_VERSION

安装在系统上的Bash的版本号.

```
bash$ echo $BASH_VERSION
3.00.14(1)-release
tcsh% echo $BASH_VERSION
BASH_VERSION: Undefined variable.
```

使用这个变量对于判断系统上到底运行的是那个shll来说是一种非常好的办法.$SHELL有时将不能给出正确的答案.

$DIRSTACK

在目录栈中最上边的值(将受到pushd和popd的影响).

这个内建的变量与dirs命令是保持一致的,但是dirs命令将显示目录栈的整个内容.

$EDITOR

脚本调用的默认编辑器,一般是vi或者是emacs.

$EUID

"effective"用户ID号.

当前用户被假定的任何id号.可能在su命令中使用.

注意:$EUID并不一定与$UID相同.

$FUNCNAME

当前函数的名字.

```
1 xyz23 ()
2 {
3   echo "$FUNCNAME now executing."  # xyz23 现在正在被执行.
4 }
5
6 xyz23
7
8 echo "FUNCNAME = $FUNCNAME"       # FUNCNAME =
9                                   # 出了函数就变为Null值了.
```

$GLOBIGNORE

一个文件名的模式匹配列表,如果在file globbing中匹配到的文件包含这个列表中的某个文件,那么这个文件将被从匹配到的文件中去掉.

$GROUPS

当前用户属于的组.

这是一个当前用户的组id列表(数组),就像在/etc/passwd中记录的一样.

```
root# echo $GROUPS
0
```

```
root# echo ${GROUPS[1]}
1
root# echo ${GROUPS[5]}
6
```

$HOME
用户的home目录,一般都是/home/username(见Example 9-14)
$HOSTNAME
hostname命令将在一个init脚本中,在启动的时候分配一个系统名字.
gethostname()函数将用来设置这个$HOSTNAME内部变量.(见Example 9-14)
$HOSTTYPE
主机类型
就像$MACHTYPE,识别系统的硬件.

```
bash$ echo $HOSTTYPE
i686
```

$IFS
内部域分隔符.
这个变量用来决定Bash在解释字符串时如何识别域,或者单词边界.
$IFS默认为空白(空格,tab,和新行),但可以修改,比如在分析逗号分隔的数据文件时.
注意:$*使用$IFS中的第一个字符,具体见Example 5-1.

```
bash$ echo $IFS | cat -vte
$
bash$ bash -c 'set w x y z; IFS=":-;"; echo "$*"'
w:x:y:o
```

注意:$IFS并不像它处理其它字符一样处理空白.

Example 9-1 $IFS和空白

```
#################################Start Script#########################################
#!/bin/bash
# $IFS 处理空白的方法,与处理其它字符不同.

output_args_one_per_line()
{
  for arg
  do echo "[$arg]"
  done
}

echo; echo "IFS=\" \""
echo "-------"

IFS=" "
var=" a  b c   "
output_args_one_per_line $var  # output_args_one_per_line `echo " a  b c   "`
#
# [a]
# [b]
# [c]

```

```
22
23 echo; echo "IFS=:"
24 echo "-----"
25
26 IFS=:
27 var=":a::b:c:::"              # 与上边的一样,但是用" "替换了":"
28 output_args_one_per_line $var
29 #
30 # []
31 # [a]
32 # []
33 # [b]
34 # [c]
35 # []
36 # []
37 # []
38
39 # 同样的事情也会发生在awk中的"FS"域分隔符.
40
41 # Thank you, Stephane Chazelas.
42
43 echo
44
45 exit 0
```

##################################End Script###

Example 12-37也是使用$IFS的另一个启发性的例子.

$IGNOREEOF

忽略EOF: 告诉shell在log out之前要忽略多少文件结束符(control-D).

$LC_COLLATE

常在.bashrc或/etc/profile中设置,这个变量用来在文件名扩展和模式匹配校对顺序.

如果$LC_COLLATE被错误的设置,那么将会在filename globbing中引起错误的结果.

注意:在2.05以后的Bash版本中,filename globbing将不在对[]中的字符区分大小写.

比如:ls [A-M]* 将即匹配File1.txt也会匹配file1.txt.为了恢复[]的习惯用法,

设置$LC_COLLATE的值为c,使用export LC_COLLATE=c 在/etc/profile或者是

~/.bashrc中.

$LC_CTYPE

这个内部变量用来控制globbing和模式匹配的字符串解释.

$LINENO

这个变量记录它所在的shell脚本中它所在行的行号.这个变量一般用于调试目的.

```
1 # *** BEGIN DEBUG BLOCK ***
2 last_cmd_arg=$_  # Save it.
3
4 echo "At line number $LINENO, variable \"v1\" = $v1"
5 echo "Last command argument processed = $last_cmd_arg"
6 # *** END DEBUG BLOCK ***
```

$MACHTYPE

系统类型

SHELL十三问

提示系统硬件
bash$ echo $MACHTYPE
i686
$OLDPWD
老的工作目录("OLD-print-working-directory",你所在的之前的目录)
$OSTYPE
操作系统类型.
bash$ echo $OSTYPE
linux
$PATH
指向Bash外部命令所在的位置,一般为/usr/bin,/usr/X11R6/bin,/usr/local/bin等.
当给出一个命令时,Bash将自动对$PATH中的目录做一张hash表.$PATH中以":"分隔的
目录列表将被存储在环境变量中.一般的,系统存储的$PATH定义在/ect/processed或
~/.bashrc中(见Appendix G).
bash$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:/sbin:/usr/sbin
PATH=${PATH}:/opt/bin将把/opt/bin目录附加到$PATH变量中.在脚本中,这是一个
添加目录到$PATH中的便捷方法.这样在这个脚本退出的时候,$PATH将会恢复(因为这个
shell是个子进程,像这样的一个脚本是不会将它的父进程的环境变量修改的)
注意:当前的工作目录"./"一般都在$PATH中被省去.
$PIPESTATUS
数组变量将保存最后一个运行的前台管道的退出码.有趣的是,这个退出码和最后一个命令
运行的退出码并不一定相同.
bash$ echo $PIPESTATUS
0
bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo $PIPESTATUS
141
bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo $?
127
$PIPESTATUS数组的每个成员都会保存一个管道命令的退出码,$PIPESTATUS[0]保存第
一个管道命令的退出码,$PIPESTATUS[1]保存第2个,以此类推.
注意:$PIPESTATUS变量在一个login shell中可能会包含一个错误的0值(3.0以下版本)
tcsh% bash
bash$ who | grep nobody | sort
bash$ echo ${PIPESTATUS[*]}
0
包含在脚本中的上边这行将会产生一个期望的输出0 1 0.
注意:在某些上下文$PIPESTATUS可能不会给出正确的结果.
bash$ echo $BASH_VERSION
3.00.14(1)-release
bash$ $ ls | bogus_command | wc
bash: bogus_command: command not found
0   0   0
bash$ echo ${PIPESTATUS[@]}

141 127 0

Chet Ramey把上边输出不成确原因归咎于ls的行为.因为如果把ls的结果放到管道上,
并且这个输出没被读取,那么SIGPIPE将会kill掉它,并且退出码变为141,而不是我们期
望的0.这种情况也会发生在tr命令中.
注意:$PIPESTATUS是一个"volatile"变量.在任何命令插入之前,并且在pipe询问之后,
这个变量需要立即被捕捉.

```
bash$ $ ls | bogus_command | wc
bash: bogus_command: command not found
0       0       0
bash$ echo ${PIPESTATUS[@]}
0 127 0
bash$ echo ${PIPESTATUS[@]}
0
```

$PPID
一个进程的$PPID就是它的父进程的进程id(pid).[1]
使用pidof命令对比一下.
$PROMPT_COMMAND
这个变量保存一个在主提示符($PS1)显示之前需要执行的命令.
$PS1
主提示符,具体见命令行上的显示.
$PS2
第2提示符,当你需要额外的输入的时候将会显示,默认为">".
$PS3
第3提示符,在一个select循环中显示(见Example 10-29).
$PS4
第4提示符,当使用-x选项调用脚本时,这个提示符将出现在每行的输出前边.
默认为"+".
$PWD
工作目录(你当前所在的目录).
与pwd内建命令作用相同.

##################################Start Script###

```
1 #!/bin/bash
2
3 E_WRONG_DIRECTORY=73
4
5 clear # 清屏.
6
7 TargetDirectory=/home/bozo/projects/GreatAmericanNovel
8
9 cd $TargetDirectory
10 echo "Deleting stale files in $TargetDirectory."
11
12 if [ "$PWD" != "$TargetDirectory" ]
13 then    # 防止偶然删除错误的目录
14   echo "Wrong directory!"
15   echo "In $PWD, rather than $TargetDirectory!"
16   echo "Bailing out!"
```

```
17   exit $E_WRONG_DIRECTORY
18 fi
19
20 rm -rf *
21 rm .[A-Za-z0-9]*    # Delete dotfiles.
21 rm .[A-Za-z0-9]*    # 删除"."文件(隐含文件).
22 # rm -f .[^.]* ..?*   为了删除以多个"."开头的文件.
23 # (shopt -s dotglob; rm -f *)   也行.
24 # Thanks, S.C. for pointing this out.
25
26 # 文件名能够包含0-255范围的所有字符,除了"/".
27 # 删除以各种诡异字符开头的文件将作为一个练习留给大家.
28
29 # 这里预留给其他的必要操作.
30
31 echo
32 echo "Done."
33 echo "Old files deleted in $TargetDirectory."
34 echo
35
36
37 exit 0
```

##################################End Script##

$REPLY

read命令如果没有给变量,那么输入将保存在$REPLY中.在select菜单中也可用,但是只
提供选择的变量的项数,而不是变量本身的值.

##################################Start Script##

```
 1 #!/bin/bash
 2 # reply.sh
 3
 4 # REPLY是'read'命令结果保存的默认变量.
 5
 6 echo
 7 echo -n "What is your favorite vegetable? "
 8 read
 9
10 echo "Your favorite vegetable is $REPLY."
11 #  当且仅当在没有变量提供给"read"命令时,
12 #+ REPLY才保存最后一个"read"命令读入的值.
13
14 echo
15 echo -n "What is your favorite fruit? "
16 read fruit
17 echo "Your favorite fruit is $fruit."
18 echo "but..."
19 echo "Value of \$REPLY is still $REPLY."
20 #  $REPLY还是保存着上一个read命令的值,
21 #+ 因为变量$fruit被传入到了这个新的"read"命令中.
```

```
22
23 echo
24
25 exit 0
```

################################End Script###

$SECONDS

这个脚本已经运行的时间(单位为秒).

################################Start Script###

```
 1 #!/bin/bash
 2
 3 TIME_LIMIT=10
 4 INTERVAL=1
 5
 6 echo
 7 echo "Hit Control-C to exit before $TIME_LIMIT seconds."
 8 echo
 9
10 while [ "$SECONDS" -le "$TIME_LIMIT" ]
11 do
12   if [ "$SECONDS" -eq 1 ]
13   then
14     units=second
15   else
16     units=seconds
17   fi
18
19   echo "This script has been running $SECONDS $units."
20   #  在一台比较慢的或者是负载很大的机器上,这个脚本可能会跳过几次循环
21   #+ 在一个while循环中.
22   sleep $INTERVAL
23 done
24
25 echo -e "\a"  # Beep!
26
27 exit 0
```

################################End Script###

$SHELLOPTS

这个变量里保存shell允许的选项,这个变量是只读的.

```
bash$ echo $SHELLOPTS
braceexpand:hashall:histexpand:monitor:history:interactive-comments:emacs
```

$SHLVL

Shell层次,就是shell层叠的层次,如果是命令行那$SHLVL就是1,如果命令行执行的脚本中,$SHLVL就是2,以此类推.

$TMOUT

如果$TMOUT环境变量被设置为一个非零的时间值,那么在过了这个指定的时间之后,shell提示符将会超时,这会引起一个logout.

在2.05b版本的Bash中,已经支持在一个带有read命令的脚本中使用$TMOUT变量.

```
 1 # 需要使用Bash v2.05b或者以后的版本上
```

```
 2
 3 TMOUT=3    # Prompt times out at three seconds.
 3 TMOUT=3    # 设置超时的时间为3秒
 4
 5 echo "What is your favorite song?"
 6 echo "Quickly now, you only have $TMOUT seconds to answer!"
 7 read song
 8
 9 if [ -z "$song" ]
10 then
11   song="(no answer)"
12   # 默认响应.
13 fi
14
15 echo "Your favorite song is $song."
```

这里有一个更复杂的方法来在一个脚本中实现超时功能.一种办法就是建立一个时间循环,在超时的时候通知脚本.不过,这也需要一个信号处理机制,在超时的时候来产生中断.

(参见Example 29-5)

Example 9-2 时间输入

```
##################################Start Script#########################################
 1 #!/bin/bash
 2 # timed-input.sh
 3
 4 # TMOUT=3    在新版本的Bash上也能工作.
 5
 6
 7 TIMELIMIT=3  # 在这个例子上是3秒,也可以设其他的值.
 8
 9 PrintAnswer()
10 {
11   if [ "$answer" = TIMEOUT ]
12   then
13     echo $answer
14   else       # 别想混合着两个例子.
15     echo "Your favorite veggie is $answer"
16     kill $!  # kill将不再需要TimerOn函数运行在后台.
17              # $! 是运行在后台的最后一个工作的PID.
18   fi
19
20 }
21
22
23
24 TimerOn()
25 {
26   sleep $TIMELIMIT && kill -s 14 $$ &
27   # 等待3秒,然后发送一个信号给脚本.
```

```
 28 }
 29
 30 Int14Vector()
 31 {
 32   answer="TIMEOUT"
 33   PrintAnswer
 34   exit 14
 35 }
 36
 37 trap Int14Vector 14   # 为了我们的目的,时间中断(14)被破坏了.
 38
 39 echo "What is your favorite vegetable "
 40 TimerOn
 41 read answer
 42 PrintAnswer
 43
 44
 45 #  很明显的,这是一个拼凑的实现.
 46 #+ 然而使用"-t"选项来"read"的话,将会简化这个任务.
 47 #  见"t-out.sh",在下边.
 48
 49 #  如果你需要一个真正的幽雅的写法...
 50 #+ 建议你使用c/c++来写这个应用,
 51 #+ 使用合适的库来完成这个任务,比如'alarm'和'setitimer'.
 52
 53 exit 0
```

##################################End Script##

使用stty也是一种选择.

Example 9-3 再来一个时间输入

##################################Start Script##

```
 1 #!/bin/bash
 2 # timeout.sh
 3
 4 #  Stephane Chazelas编写,
 5 #+ 本书作者进行了一些修改.
 6
 7 INTERVAL=5              # timeout间隔
 8
 9 timedout_read() {
10   timeout=$1
11   varname=$2
12   old_tty_settings=`stty -g`
13   stty -icanon min 0 time ${timeout}0
14   eval read $varname      # 或者就是 read $varname
15   stty "$old_tty_settings"
16   # 察看"stty"的man页.
17 }
18
```

```
19 echo; echo -n "What's your name? Quick! "
20 timedout_read $INTERVAL your_name
21
22 #  这种方法可能不是每个终端类型都可以正常使用的.
23 #  最大的timeout依赖于具体的终端.
24 #+ (一般都是25.5秒).
25
26 echo
27
28 if [ ! -z "$your_name" ]  # If name input before timeout...
29 then
30   echo "Your name is $your_name."
31 else
32   echo "Timed out."
33 fi
34
35 echo
36
37 # 这个脚本的行为可能与"timed-input.sh"有点不同.
38 # 在每次按键的时候,计数器都会重置.
39
40 exit 0
```

################################End Script###

或许,最简单的办法就是使用-t选项来read了.

Example 9-4 Timed read

################################Start Script###

```
 1 #!/bin/bash
 2 # t-out.sh
 3 # "syngin seven"的一个很好的提议 (thanks).
 4
 5
 6 TIMELIMIT=4         # 4 seconds
 7
 8 read -t $TIMELIMIT variable <&1
 9 #                        ^^^
10 #  在这个例子中,对于Bash 1.x和2.x就需要使用"<&1"
11 #  但对于Bash 3.x就不需要.
12
13 echo
14
15 if [ -z "$variable" ]  # Is null?
16 then
17   echo "Timed out, variable still unset."
18 else
19   echo "variable = $variable"
20 fi
21
22 exit 0
```

################################End Script###

$UID

用户ID号.

当前用户的id号,在/etc/passwd中记录.

这个值不会因为用户使用了su命令而改变.$UID是只读变量,不容易在命令行或者是脚本中被修改,并且和内建的id命令很相像.

Example 9-5 我是root?

################################Start Script###

```
 1 #!/bin/bash
 2 # am-i-root.sh:   我是不是root用户?
 3
 4 ROOT_UID=0   # Root的$UID是0.
 5
 6 if [ "$UID" -eq "$ROOT_UID" ]  # 是否是root用户,请站出来.
 7 then
 8   echo "You are root."
 9 else
10   echo "You are just an ordinary user (but mom loves you just the same)."
11 fi
12
13 exit 0
14
15
16 # ============================================================== #
17 # 下边的代码将不被执行,因为脚本已经退出了.
18
19 # 检验是root用户的一种可选方法:
20
21 ROOTUSER_NAME=root
22
23 username=`id -nu`             # Or...   username=`whoami`
24 if [ "$username" = "$ROOTUSER_NAME" ]
25 then
26   echo "Rooty, toot, toot. You are root."
27 else
28   echo "You are just a regular fella."
29 fi
```

################################End Script###

见例子Example 2-3

注意:变量$ENV,$LOGNAME,$MAIL,$TERM,$USER,和$USERNAME并不是Bash的内建变量.它们经常被设置成环境变量,它们一般都放在Bash的安装文件中.$SHELL,用户登录的shell的名字,可能是从/etc/passwd设置的,也可能是在一个"init"脚本中设置的,同样的,它也不是Bash的内建变量.

```
tcsh% echo $LOGNAME
bozo
tcsh% echo $SHELL
/bin/tcsh
tcsh% echo $TERM
```

rxvt
bash$ echo $LOGNAME
bozo
bash$ echo $SHELL
/bin/tcsh
bash$ echo $TERM
rxvt

位置参数

$0, $1, $2,等等...
位置参数,从命令行传递给脚本,或者是传递给函数.或者赋职给一个变量.
(具体见Example 4-5和Example 11-15)

$#
命令行或者是位置参数的个数.(见Example 33-2)

$*
所有的位置参数,被作为一个单词.
注意:"$*"必须被""引用.

$@
与$*同义,但是每个参数都是一个独立的""引用字串,这就意味着参数被完整地传递,
并没有被解释和扩展.这也意味着,每个参数列表中的每个参数都被当成一个独立的
单词.
注意:"$@"必须被引用.

Example 9-6 arglist:通过$*和$@列出所有的参数

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # arglist.sh
 3 # 多使用几个参数来调用这个脚本,比如"one tow three".
 4
 5 E_BADARGS=65
 6
 7 if [ ! -n "$1" ]
 8 then
 9   echo "Usage: `basename $0` argument1 argument2 etc."
10   exit $E_BADARGS
11 fi
12
13 echo
14
15 index=1          # 初始化数量.
16
17 echo "Listing args with \"\$*\":"
18 for arg in "$*"  # 如果"$*"不被""引用,那么将不能正常地工作
19 do
20   echo "Arg #$index = $arg"
21   let "index+=1"
22 done             # $* sees all arguments as single word.
22 done             # $* 认为所有的参数为一个单词
23 echo "Entire arg list seen as single word."
```

```
24
25 echo
26
27 index=1        # 重置数量.
28               # 如果你忘了这句会发生什么?
29
30 echo "Listing args with \"\$@\":"
31 for arg in "$@"
32 do
33   echo "Arg #$index = $arg"
34   let "index+=1"
35 done           # $@ 认为每个参数都一个单独的单词.
36 echo "Arg list seen as separate words."
37
38 echo
39
40 index=1        # 重置数量.
41
42 echo "Listing args with \$* (unquoted):"
43 for arg in $*
44 do
45   echo "Arg #$index = $arg"
46   let "index+=1"
47 done           # 未""引用的$*把参数作为独立的单词.
48 echo "Arg list seen as separate words."
49
50 exit 0
```

##################################End Script###

在shift命令后边,$@将保存命令行中剩余的参数,而$1被丢掉了.

```
 1 #!/bin/bash
 2 # 使用 ./scriptname 1 2 3 4 5 来调用这个脚本
 3
 4 echo "$@"    # 1 2 3 4 5
 5 shift
 6 echo "$@"    # 2 3 4 5
 7 shift
 8 echo "$@"    # 3 4 5
 9
10 # 每个"shift"都丢弃$1.
11 # "$@" 将包含剩下的参数.
```

$@也作为为工具使用,用来过滤传给脚本的输入.

cat "$@"结构接受从stdin传来的输入,也接受从参数中指定的文件传来的输入.

具体见Example 12-21和Example 12-22.

注意:$*和$@的参数有时会不一致,发生令人迷惑的行为,这依赖于$IFS的设置.

Example 9-7 不一致的$*和$@行为

##################################Start Script###

```
 1 #!/bin/bash
 2
```

```
3 #  "$*"和"$@"的古怪行为,
4 #+ 依赖于它们是否被""引用.
5 #  单词拆分和换行的不一致处理.
6
7
8 set -- "First one" "second" "third:one" "" "Fifth: :one"
9 # 设置这个脚本参数,$1,$2,等等.
10
11 echo
12
13 echo 'IFS unchanged, using "$*"'
14 c=0
15 for i in "$*"              # 引用
16 do echo "$((c+=1)): [$i]"   # 这行在下边的每个例子中都一样.
17                            # Echo参数.
18 done
19 echo ---
20
21 echo 'IFS unchanged, using $*'
22 c=0
23 for i in $*                # 未引用
24 do echo "$((c+=1)): [$i]"
25 done
26 echo ---
27
28 echo 'IFS unchanged, using "$@"'
29 c=0
30 for i in "$@"
31 do echo "$((c+=1)): [$i]"
32 done
33 echo ---
34
35 echo 'IFS unchanged, using $@'
36 c=0
37 for i in $@
38 do echo "$((c+=1)): [$i]"
39 done
40 echo ---
41
42 IFS=:
43 echo 'IFS=":", using "$*"'
44 c=0
45 for i in "$*"
46 do echo "$((c+=1)): [$i]"
47 done
48 echo ---
49
50 echo 'IFS=":", using $*'
```

```
51 c=0
52 for i in $*
53 do echo "$((c+=1)): [$i]"
54 done
55 echo ---
56
57 var=$*
58 echo 'IFS=":", using "$var" (var=$*)'
59 c=0
60 for i in "$var"
61 do echo "$((c+=1)): [$i]"
62 done
63 echo ---
64
65 echo 'IFS=":", using $var (var=$*)'
66 c=0
67 for i in $var
68 do echo "$((c+=1)): [$i]"
69 done
70 echo ---
71
72 var="$*"
73 echo 'IFS=":", using $var (var="$*")'
74 c=0
75 for i in $var
76 do echo "$((c+=1)): [$i]"
77 done
78 echo ---
79
80 echo 'IFS=":", using "$var" (var="$*")'
81 c=0
82 for i in "$var"
83 do echo "$((c+=1)): [$i]"
84 done
85 echo ---
86
87 echo 'IFS=":", using "$@"'
88 c=0
89 for i in "$@"
90 do echo "$((c+=1)): [$i]"
91 done
92 echo ---
93
94 echo 'IFS=":", using $@'
95 c=0
96 for i in $@
97 do echo "$((c+=1)): [$i]"
98 done
```

```
 99 echo ---
100
101 var=$@
102 echo 'IFS=":", using $var (var=$@)'
103 c=0
104 for i in $var
105 do echo "$((c+=1)): [$i]"
106 done
107 echo ---
108
109 echo 'IFS=":", using "$var" (var=$@)'
110 c=0
111 for i in "$var"
112 do echo "$((c+=1)): [$i]"
113 done
114 echo ---
115
116 var="$@"
117 echo 'IFS=":", using "$var" (var="$@")'
118 c=0
119 for i in "$var"
120 do echo "$((c+=1)): [$i]"
121 done
122 echo ---
123
124 echo 'IFS=":", using $var (var="$@")'
125 c=0
126 for i in $var
127 do echo "$((c+=1)): [$i]"
128 done
129
130 echo
131
132 # 用ksh或者zsh -y来试试这个脚本.
133
134 exit 0
135
136 # This example script by Stephane Chazelas,
137 # and slightly modified by the document author.
```

##################################End Script###

注意:$@和$*中的参数只有在""中才会不同.

Example 9-8 当$IFS为空时的$*和$@

##################################Start Script###

```
 1 #!/bin/bash
 2
 3 #  如果$IFS被设置为空时,
 4 #+ 那么"$*" 和"$@" 将不会象期望那样echo出位置参数.
 5
```

```
 6 mecho ()      # Echo 位置参数.
 7 {
 8 echo "$1,$2,$3";
 9 }
10
11
12 IFS=""        # 设置为空.
13 set a b c     # 位置参数.
14
15 mecho "$*"    # abc,,
16 mecho $*      # a,b,c
17
18 mecho $@      # a,b,c
19 mecho "$@"    # a,b,c
20
21 #  当$IFS设置为空时,$* 和$@ 的行为依赖于
22 #+ 正在运行的Bash或者sh的版本.
23 #  所以在脚本中使用这种"feature"不是明智的行为.
24
25
26 # Thanks, Stephane Chazelas.
27
28 exit 0
```

################################End Script###

其他的特殊参数

$-

传递给脚本的falg(使用set命令).参考Example 11-15.

注意:这起初是ksh的特征,后来被引进到Bash中,但不幸的是,在Bash中它看上去也不能可靠的工作.使用它的一个可能的方法就是让这个脚本进行自我测试(查看是否是交互的).

$!

在后台运行的最后的工作的PID(进程ID).

```
 1 LOG=$0.log
 2
 3 COMMAND1="sleep 100"
 4
 5 echo "Logging PIDs background commands for script: $0" >> "$LOG"
 6 # 所以它们可以被监控,并且在必要的时候kill掉.
 7 echo >> "$LOG"
 8
 9 # Logging 命令.
10
11 echo -n "PID of \"$COMMAND1\":  " >> "$LOG"
12 ${COMMAND1} &
13 echo $! >> "$LOG"
14 # PID of "sleep 100":  1506
15
16 # Thank you, Jacques Lederer, for suggesting this.
```

```
 1 possibly_hanging_job & { sleep ${TIMEOUT}; eval 'kill -9 $!' &> /dev/null; }
 2 # 强制结束一个品行不良的程序.
 3 # 很有用,比如在init脚本中.
 4
 5 # Thank you,Sylvain Fourmanoit,for this creative use of the "!" variable.
```

$_
保存之前执行的命令的最后一个参数.
Example 9-9 下划线变量

```
################################Start Script#########################################
 1 #!/bin/bash
 2
 3 echo $_            # /bin/bash
 4                   # 只是调用/bin/bash来运行这个脚本.
 5
 6 du >/dev/null       # 将没有命令的输出
 7 echo $_            # du
 8
 9 ls -al >/dev/null    # 没有命令输出
10 echo $_            # -al  (最后的参数)
11
12 :
13 echo $_            # :
################################End Script###########################################
```

$?
命令,函数或者脚本本身的退出状态(见Example 23-7)
$$
脚本自身的进程ID.这个变量经常用来构造一个"unique"的临时文件名.
(参考Example A-13,Example 29-6,Example 12-28和Example 11-25).
这通常比调用mktemp来得简单.
注意事项:
[1] 当前运行的脚本的PID为$$.
[2] "argument"和"parameter"这两个单词经常不加区分的使用.在这整本书中,这两个
单词的意思完全相同.(在翻译的时候就未加区分,统统翻译成参数)
9.2 操作字符串

Bash支持超多的字符串操作,操作的种类和数量令人惊异.但不幸的是,这些工具缺乏集中性.
一些是参数替换的子集,但是另一些则属于UNIX的expr命令.这就导致了命令语法的不一致和
功能的重叠,当然也会引起混乱.
字符串长度

```
${#string}
expr length $string
expr "$string" : '.*'
 1 stringZ=abcABC123ABCabc
 2
 3 echo ${#stringZ}                 # 15
 4 echo `expr length $stringZ`      # 15
 5 echo `expr "$stringZ" : '.*'`    # 15
```

Example 9-10 在一个文本文件的段间插入空行

SHELL十三问

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # paragraph-space.sh
 3
 4 # 在一个不空行的文本文件的段间插入空行.
 5 # Usage: $0 <FILENAME
 6
 7 MINLEN=45        # 可能需要修改这个值.
 8 #  假定行的长度小于$MINLEN指定的长度
 9 #+ $MINLEN中的值用来描述多少个字符结束一个段.
10
11 while read line  # 对于需要多行输入的文件基本都是这个样子
12 do
13   echo "$line"   # 输出line.
14
15   len=${#line}
16   if [ "$len" -lt "$MINLEN" ]
17     then echo    # 在短行后边添加一个空行
18   fi
19 done
20
21 exit 0
################################End Script#########################################
从字符串开始的位置匹配子串的长度
 expr match "$string" '$substring'
  $substring是一个正则表达式

 expr "$string" : '$substring'
  $substring是一个正则表达式
 1 stringZ=abcABC123ABCabc
 2 #       |------|
 3
 4 echo `expr match "$stringZ" 'abc[A-Z]*.2'`   # 8
 5 echo `expr "$stringZ" : 'abc[A-Z]*.2'`       # 8
索引|
 expr index $string $substring
  匹配到子串的第一个字符的位置.
 1 stringZ=abcABC123ABCabc
 2 echo `expr index "$stringZ" C12`            # 6
 3                                             # C position.
 4
 5 echo `expr index "$stringZ" 1c`             # 3
 6 # 'c' (in #3 position) matches before '1'.
 在C语言中最近的等价函数为strchr().
提取子串

 ${string:position}
  在string中从位置$position开始提取子串.
```

如果$string为"*"或"@",那么将提取从位置$position开始的位置参数,[1]
${string:position:length}
在string中从位置$position开始提取$length长度的子串.

```
################################Start Script#########################################
1 stringZ=abcABC123ABCabc
2 #       0123456789.....
3 #       0-based indexing.
4
5 echo ${stringZ:0}                    # abcABC123ABCabc
6 echo ${stringZ:1}                    # bcABC123ABCabc
7 echo ${stringZ:7}                    # 23ABCabc
8
9 echo ${stringZ:7:3}                   # 23A
10                                  # 3个字符长度的子串.
11
12
13
14 # 有没有可能从字符结尾开始,反向提取子串?
15
16 echo ${stringZ:-4}                    # abcABC123ABCabc
17 # 以${parameter:-default}方式,默认是提取完整地字符串.
18 # 然而 . . .
19
20 echo ${stringZ:(-4)}                   # Cabc
21 echo ${stringZ: -4}                    # Cabc
22 # 现在,它可以工作了.
23 # 使用圆括号或者添加一个空格来转义这个位置参数.
24
25 # Thank you, Dan Jacobson, for pointing this out.
################################End Script###########################################
```

如果$string参数为"*"或"@",那将最大的提取从$position开始的$length个位置参数.

```
1 echo ${*:2}          # Echo出第2个和后边所有的位置参数.
2 echo ${@:2}          # 与前边相同.
3
4 echo ${*:2:3}        # 从第2个开始,Echo出后边3个位置参数.
```

expr substr $string $position $length
在string中从位置$position开始提取$length长度的子串.

```
1 stringZ=abcABC123ABCabc
2 #       123456789......
3 #       1-based indexing.
4
5 echo `expr substr $stringZ 1 2`            # ab
6 echo `expr substr $stringZ 4 3`            # ABC
```

expr match "$string" '\($substring\)'
从$string的开始位置提取$substring,$substring是一个正则表达式.
expr "$string" : '\($substring\)'
从$string的开始位置提取$substring,$substring是一个正则表达式.

```
1 stringZ=abcABC123ABCabc
```

```
2 #       =======
3
4 echo `expr match "$stringZ" '\(.[b-c]*[A-Z]..[0-9]\)'`   # abcABC1
5 echo `expr "$stringZ" : '\(.[b-c]*[A-Z]..[0-9]\)'`       # abcABC1
6 echo `expr "$stringZ" : '\(.......\)'`                   # abcABC1
7 # All of the above forms give an identical result.
```

子串削除

${string#substring}

从$string的左边截掉第一个匹配的$substring

${string##substring}

从$string的左边截掉最后一个个匹配的$substring

```
1 stringZ=abcABC123ABCabc
2 #       |----|
3 #       |----------|
4
5 echo ${stringZ#a*C}      # 123ABCabc
6 # 截掉'a'和'C'之间最近的匹配.
7
8 echo ${stringZ##a*C}     # abc
9 # 截掉'a'和'C'之间最远的匹配.
```

${string%substring}

从$string的右边截掉第一个匹配的$substring

${string%%substring}

从$string的右边截掉最后一个匹配的$substring

```
1 stringZ=abcABC123ABCabc
2 #                 ||
3 #       |------------|
4
5 echo ${stringZ%b*c}      # abcABC123ABCa
6 # 从$stringZ的后边开始截掉'b'和'c'之间的最近的匹配
7
8 echo ${stringZ%%b*c}     # a
9 # 从$stringZ的后边开始截掉'b'和'c'之间的最远的匹配
```

Example 9-11 利用修改文件名,来转换图片格式

```
################################Start Script#########################################
1 #!/bin/bash
2 #  cvt.sh:
3 #  把一个目录下的所有MacPaint格式的图片文件都转换为"pbm"格式的图片文件.
4
5 #  使用来自"netpbm"包的"macptopbm"程序,
6 #+ 这个程序主要是由Brian Henderson(bryanh@giraffe-data.com)来维护的.
7 #  Netpbm是大多数Linux发行版的标准部分.
8
9 OPERATION=macptopbm
10 SUFFIX=pbm          # 新的文件名后缀
11
```

```
12 if [ -n "$1" ]
13 then
14   directory=$1      # 如果目录名作为第1个参数给出...
15 else
16   directory=$PWD    # 否则使用当前的工作目录.
17 fi
18
19 #  假设在目标目录中的所有文件都是MacPaint格式的图片文件,
20 #+ 以".mac"为文件名的后缀.
21
22 for file in $directory/*    # Filename globbing.
23 do
24   filename=${file%.*c}      #  去掉文件名的".mac"后缀
25                        #+ ('.*c' matches everything
25                        #+ ('.*c' 将匹配'.'和'c'之间的任何字符串).
26
27   $OPERATION $file > "$filename.$SUFFIX"
28                        # 转换为新的文件名.
29   rm -f $file             # 转换完毕后删除原有的文件.
30   echo "$filename.$SUFFIX"  # 从stdout输出反馈.
31 done
32
33 exit 0
34
35 # 练习:
36 # --------
37 #  就像它现在这个样子,这个脚本把当前目录的所有文件都转换了.
38 #
39 #  修改这个脚本,让他只转换以".mac"为后缀的文件.
```

################################End Script###

一个简单的模拟getopt命令的办法就是使用子串提取结构.

Example 9-12 模仿getopt命令

################################Start Script###

```
 1 #!/bin/bash
 2 # getopt-simple.sh
 3 # Author: Chris Morgan
 4 # 授权使用在ABS Guide中.
 5
 6
 7 getopt_simple()
 8 {
 9    echo "getopt_simple()"
10    echo "Parameters are '$*'"
11    until [ -z "$1" ]
12    do
13      echo "Processing parameter of: '$1'"
14      if [ ${1:0:1} = '/' ]
15      then
```

```
16          tmp=${1:1}             # 去掉开头的'/' . . .
17          parameter=${tmp%%=*}     # 提取名字.
18          value=${tmp##*=}         # 提取值.
19          echo "Parameter: '$parameter', value: '$value'"
20          eval $parameter=$value
21        fi
22        shift
23      done
24 }
25
26 # 传递所有的选项到getopt_simple().
27 getopt_simple $*
28
29 echo "test is '$test'"
30 echo "test2 is '$test2'"
31
32 exit 0
33
34 ---
35
36 sh getopt_example.sh /test=value1 /test2=value2
37
38 Parameters are '/test=value1 /test2=value2'
39 Processing parameter of: '/test=value1'
40 Parameter: 'test', value: 'value1'
41 Processing parameter of: '/test2=value2'
42 Parameter: 'test2', value: 'value2'
43 test is 'value1'
44 test2 is 'value2'
```

################################End Script###

子串替换

${string/substring/replacement}

使用$replacement来替换第一个匹配的$substring.

${string//substring/replacement}

使用$replacement来替换所有匹配的$substring.

```
1 stringZ=abcABC123ABCabc
2
3 echo ${stringZ/abc/xyz}          # xyzABC123ABCabc
4                                  # 用'xyz'来替换第一个匹配的'abc'.
5
6 echo ${stringZ//abc/xyz}         # xyzABC123ABCxyz
7                                  # 用'xyz'来替换所有匹配的'abc'.
```

${string/#substring/replacement}

如果$substring匹配$string的开头部分,那么就用$replacement来替换$substring.

${string/%substring/replacement}

如果$substring匹配$string的结尾部分,那么就用$replacement来替换$substring.

```
1 stringZ=abcABC123ABCabc
2
```

```
 3 echo ${stringZ/#abc/XYZ}         # XYZABC123ABCabc
 4                                  # 用'XYZ'替换开头的'abc'
 5
 6 echo ${stringZ/%abc/XYZ}         # abcABC123ABCXYZ
 7                                  # 用'XYZ'替换结尾的'abc'
```

9.2.1 使用awk来操作字符串

~~~~~~~~~~~~~~~~~~~~~~~~~

Bash脚本也可以使用awk来操作字符串.

Example 9-13 提取字符串的一种可选的方法

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # substring-extraction.sh
 3
 4 String=23skidoo1
 5 #      012345678    Bash
 6 #      123456789    awk
 7 # 注意,对于awk和Bash来说,它们使用的是不同的string索引系统:
 8 # Bash的第一个字符是从'0'开始记录的.
 9 # Awk的第一个字符是从'1'开始记录的.
10
11 echo ${String:2:4} # 位置3 (0-1-2), 4 个字符长
12                                  # skid
13
14 # awk中等价于${string:pos:length}的命令是substr(string,pos,length).
15 echo | awk '
16 { print substr("${String}",3,4)      # skid
17 }
18 '
19 #  使用一个空的"echo"通过管道给awk一个假的输入,
20 #+ 这样可以不用提供一个文件名.
21
22 exit 0
################################End Script###########################################
```

9.2.2 更深的讨论

~~~~~~~~~~~~~~~~~~

关于在脚本中使用字符串更深的讨论,请参考 9.3节,h和expr命令列表的相关章节.

关于脚本的例子,见:

1 Example 12-9

2 Example 9-16

3 Example 9-17

4 Example 9-18

5 Example 9-20

注意事项:

[1] 这适用于命令行参数和函数参数.

9.3 参数替换

操作和扩展变量

SHELL十三问

${parameter}
 与$parameter相同,就是parameter的值.在特定的上下文中,只有少部分会产生
 ${parameter}的混淆.可以组合起来一起赋指给字符串变量.

```
 1 your_id=${USER}-on-${HOSTNAME}
 2 echo "$your_id"
 3 #
 4 echo "Old \$PATH = $PATH"
 5 PATH=${PATH}:/opt/bin  #Add /opt/bin to $PATH for duration of script.
 6 echo "New \$PATH = $PATH"
```

${parameter-default},${parameter:-default}
 如果parameter没被set,那么就使用default.

```
 1 echo ${username-`whoami`}
 2 # echo `whoami`的结果,如果没set username变量的话.
```

 注意:${parameter-default}和${parameter:-default}大部分时候是相同的.
  额外的":"在parameter被声明的时候(而且被赋空值),会有一些不同.

##################################Start Script###

```
 1 #!/bin/bash
 2 # param-sub.sh
 3
 4 #  一个变量是否被声明
 5 #+ 将会影响默认选项的触发
 6 #+ 甚至于这个变量被设为空.
 7
 8 username0=
 9 echo "username0 has been declared, but is set to null."
10 echo "username0 = ${username0-`whoami`}"
11 # 将不会echo.
12
13 echo
14
15 echo username1 has not been declared.
16 echo "username1 = ${username1-`whoami`}"
17 # 将会echo.
18
19 username2=
20 echo "username2 has been declared, but is set to null."
21 echo "username2 = ${username2:-`whoami`}"
22 #                        ^
23 # 将会echo因为使用的是:-而不是 -.
24 # 和前边的第一个例子好好比较一下.
25
26
27 #
28
29 # 再来一个:
30
```

```
31 variable=
32 # 变量已经被声明了,但是被设置为空.
33
34 echo "${variable-0}"    # (no output)
35 echo "${variable:-1}"   # 1
36 #              ^
37
38 unset variable
39
40 echo "${variable-2}"    # 2
41 echo "${variable:-3}"   # 3
42
43 exit 0
```

##################################End Script###

如果脚本中并没有传入命令行参数,那么default parameter将被使用.

```
1 DEFAULT_FILENAME=generic.data
2 filename=${1:-$DEFAULT_FILENAME}
3 #  如果没有参数被传递进来,那么下边的命令快将操作
4 #+ 文件"generic.data"
5 #
6 #  后续命令.
```

另外参见Example 3-4,Example 28-2,和Example A-6.

与"使用一个与列表来支持一个默认的命令行参数"的方法相比较.

${parameter=default},${parameter:=default}

如果parameter未设置,那么就设置为default.

这两种办法绝大多数时候用法都一样,只有在$parameter被声明并设置为空的时候,才会有区别,[1]和上边的行为一样.

```
1 echo ${username=`whoami`}
2 # Variable "username" is now set to `whoami`.
2 # 变量"username"被赋值为`whoami`.
```

${parameter+alt_value},${parameter:+alt_value}

如果parameter被set了,那就使用alt_value,否则就使用null字符串.

这两种办法绝大多数时候用法都一样,只有在$parameter被声明并设置为空的时候,会有区别,见下.

##################################Start Script###

```
1 echo "###### \${parameter+alt_value} ########"
2 echo
3
4 a=${param1+xyz}
5 echo "a = $a"      # a =
6
7 param2=
8 a=${param2+xyz}
9 echo "a = $a"      # a = xyz
10
11 param3=123
12 a=${param3+xyz}
```

```
13 echo "a = $a"      # a = xyz
14
15 echo
16 echo "###### \${parameter:+alt_value} ########"
17 echo
18
19 a=${param4:+xyz}
20 echo "a = $a"      # a =
21
22 param5=
23 a=${param5:+xyz}
24 echo "a = $a"      # a =
25 # 与a=${param5+xyz}有不同的结果.
26
27 param6=123
28 a=${param6+xyz}
29 echo "a = $a"      # a = xyz
```

################################End Script###

${parameter?err_msg}, ${parameter:?err_msg}

如果parameter被set,那就是用set的值,否则print err_msg.

这两种办法绝大多数时候用法都一样,只有在$parameter被声明并设置为空的时候,

会有区别,见上.

Example 9-14 使用参数替换和error messages

################################Start Script###

```
1 #!/bin/bash
2
3 #  检查一些系统的环境变量.
4 #  这是个好习惯.
5 #  比如,如果$USER(在console上的用户名)没被set,
6 #+ 那么系统就不会认你.
7
8 : ${HOSTNAME?} ${USER?} ${HOME?} ${MAIL?}
9   echo
10   echo "Name of the machine is $HOSTNAME."
11   echo "You are $USER."
12   echo "Your home directory is $HOME."
13   echo "Your mail INBOX is located in $MAIL."
14   echo
15   echo "If you are reading this message,"
16   echo "critical environmental variables have been set."
17   echo
18   echo
19
20 # ------------------------------------------------------
21
22 #  ${variablename?} 结果也可以用来
23 #+ 在一个脚本中检查变量是否被set.
24
```

```
25 ThisVariable=Value-of-ThisVariable
26 #  注意,顺便提一下,这个字符串变量可能在它们的名字中会被设置
27 #+ 非法字符
28 : ${ThisVariable?}
29 echo "Value of ThisVariable is $ThisVariable".
30 echo
31 echo
32
33
34 : ${ZZXy23AB?"ZZXy23AB has not been set."}
35 #  如果ZZXy23AB没被set,
36 #+ 那么这个脚本将以一个error message终止.
37
38 # 你可以指定错误消息.
39 # : ${variablename?"ERROR MESSAGE"}
40
41
42 # 同样的结果:    dummy_variable=${ZZXy23AB?}
43 #                dummy_variable=${ZZXy23AB?"ZXy23AB has not been set."}
44 #
45 #                echo ${ZZXy23AB?} >/dev/null
46
47 #  同"set -u"命令来比较这些检查变量是否被set的方法.
48 #
49
50
51
52 echo "You will not see this message, because script already terminated."
53
54 HERE=0
55 exit $HERE   # Will NOT exit here.
56
57 # 事实上,这个脚本将返回值1作为退出状态(echo $?).
```

################################End Script###

Example 9-15 参数替换和"usage"messages

################################Start Script###

```
 1 #!/bin/bash
 2 # usage-message.sh
 3
 4 : ${1?"Usage: $0 ARGUMENT"}
 5 #  如果没有命令行参数,那么脚本将在此处退出.
 6 #+ 并且打出如下的错误消息.
 7 #    usage-message.sh: 1: Usage: usage-message.sh ARGUMENT
 8
 9 echo "These two lines echo only if command-line parameter given."
10 echo "command line parameter = \"$1\""
11
12 exit 0  # 如果有命令行参数,那么将在此处退出.
```

```
13
14 # 测试这个脚本,第1次测试带参数,第2次测试不带参数.
15 # 如果有参数,那么"$?"就是0.
16 # 如果没有,那么"$?"就是1.
```

################################End Script###

参数替换和扩展

下边的表达式是使用expr字符串匹配操作的补充(见Example 12-9).

这些特定的使用方法绝大多数情况都是用来分析文件目录名.

变量长度/子串删除

${#var}

字符串长度($var的字符数量).对于一个数组,${#array}是数组中第一个元素的长度.

一些例外:

${#*}和${#@}将给出位置参数的个数.

对于数组来说${#array[*]}和${$#array[@]}将给出数组元素的个数.

Example 9-16 变量长度

################################Start Script###

```
 1 #!/bin/bash
 2 # length.sh
 3
 4 E_NO_ARGS=65
 5
 6 if [ $# -eq 0 ]  # 这个demo脚本必须有命令行参数.
 7 then
 8   echo "Please invoke this script with one or more command-line arguments."
 9   exit $E_NO_ARGS
10 fi
11
12 var01=abcdEFGH28ij
13 echo "var01 = ${var01}"
14 echo "Length of var01 = ${#var01}"
15 # 现在,让我们试试在里边嵌入一个空格.
16 var02="abcd EFGH28ij"
17 echo "var02 = ${var02}"
18 echo "Length of var02 = ${#var02}"
19
20 echo "Number of command-line arguments passed to script = ${#@}"
21 echo "Number of command-line arguments passed to script = ${#*}"
22
23 exit 0
```

################################End Script###

${var#Pattern}, ${var##Pattern}

从$var开头删除最近或最远匹配$Pattern的子串.

来自Example A-7例子的一部分.

```
 1 # 来自"days-between.sh"例子的一个函数.
 2 # 去掉传递进来的参数开头的0.
 3
 4 strip_leading_zero () #  去掉开头的0
 5 {                     #+ 从传递进来的参数中.
```

```
 6   return=${1#0}      #  "1"指的是"$1" -- 传进来的参数.
 7 }                    #  "0"就是我们想从"$1"中删除的子串.
```

下边是Manfred Schwarb's对上边函数的一个改版.

```
 1 strip_leading_zero2 () # 去掉开头的0,因为如果不去掉的话
 2 {                     # Bash将会把这个值作为8进制解释.
 3   shopt -s extglob    # 打开扩展globbing.
 4   local val=${1##+(0)} # 使用局部变量,匹配最长的连续的0.
 5   shopt -u extglob    # 打开扩展globbing.
 6   _strip_leading_zero2=${val:-0}
 7                       # 如果输入为0,那么返回0来代替"".
 8 }
```

另一个例子

```
 1 echo `basename $PWD`       # 当前工作目录的basename.
 2 echo "${PWD##*/}"          # 当前工作目录的basename.
 3 echo
 4 echo `basename $0`         # 脚本名字.
 5 echo $0                   # 脚本名字.
 6 echo "${0##*/}"            # 脚本名字.
 7 echo
 8 filename=test.data
 9 echo "${filename##*.}"     # data
```

${var%Pattern}, ${var%%Pattern}

从$var结尾删除最近或最远匹配$Pattern的子串.

Bash version2 添加了额外的选项.

Example 9-17 参数替换中的模式匹配

###################################Start Script###

```
 1 #!/bin/bash
 2 # patt-matching.sh
 3
 4 # 使用# ## % %%来进行参数替换操作的模式匹配.
 5
 6 var1=abcd12345abc6789
 7 pattern1=a*c  # * (通配符) 匹配a - c之间的任何字符.
 8
 9 echo
10 echo "var1 = $var1"          # abcd12345abc6789
11 echo "var1 = ${var1}"        # abcd12345abc6789
12                              # (alternate form)
13 echo "Number of characters in ${var1} = ${#var1}"
14 echo
15
16 echo "pattern1 = $pattern1"   # a*c  (everything between 'a' and 'c')
17 echo "--------------"
18 echo '${var1#$pattern1}  =' "${var1#$pattern1}"    #        d12345abc6789
19 # 最短的可能匹配, 去掉abcd12345abc6789的前3个字符
20 #                  |-|            ^^^
21 echo '${var1##$pattern1} =' "${var1##$pattern1}"   #                6789
22 # 最远的匹配,去掉abcd12345abc6789的前12个字符.
```

```
23 #               |----------|     ^^^^
24
25 echo; echo; echo
26
27 pattern2=b*9           # 'b' 到'9'之间的任何字符
28 echo "var1 = $var1"     # 还是 abcd12345abc6789
29 echo
30 echo "pattern2 = $pattern2"
31 echo "--------------"
32 echo '${var1%pattern2}  =' "${var1%$pattern2}"     #     abcd12345a
33 # 最近的匹配, 去掉abcd12345abc6789的最后6个字符
34 #                       |----|  ^^^^
35 echo '${var1%%pattern2} =' "${var1%%$pattern2}"    #     a
36 # 最远匹配, 去掉abcd12345abc6789的最后12个字符
37 #               |-------------|  ^^^^^^
38
39 # 记住, # 和## 从字符串的左边开始,并且去掉左边的字符串,
40 #          % 和 %% 从字符串的右边开始,并且去掉右边的子串.
41
42 echo
43
44 exit 0
```

################################End Script###

Example 9-18 重命名文件扩展名

################################Start Script###

```
 1 #!/bin/bash
 2 # rfe.sh: 重命名文件扩展名.
 3 #
 4 # 用法:   rfe old_extension new_extension
 5 #
 6 # 例子:
 7 # 将指定目录的所有 *.gif 文件都重命名为 *.jpg,
 8 # 用法:   rfe gif jpg
 9
10
11 E_BADARGS=65
12
13 case $# in
14   0|1)           # "|" 在这里的意思是或操作.
15   echo "Usage: `basename $0` old_file_suffix new_file_suffix"
16   exit $E_BADARGS  # 如果只有0个或1个参数,那么就退出.
17   ;;
18 esac
19
20
21 for filename in *.$1
22 # 以第一个参数为扩展名的全部文件的列表
23 do
```

```
24   mv $filename ${filename%$1}$2
25   #  从筛选出的文件中先去掉以第一参数结尾的扩展名部门,
26   #+ 然后作为扩展名把第2个参数添加上.
27 done
28
29 exit 0
```

##################################End Script###

变量扩展/子串替换

这些结构都是从ksh中吸收来的.

${var:pos}

变量var从位置pos开始扩展.

${var:pos:len}

从位置pos开始,并扩展len长度个字符.见Example A-14(这个例子里有这种操作的一个创造性用法)

${var/Pattern/Replacement}

使用Replacement来替换var中的第一个Pattern的匹配.

${var//Pattern/Replacement}

全局替换.在var中所有的匹配,都会用Replacement来替换.

向上边所说,如果Replacement被忽略的话,那么所有匹配到的Pattern都会被删除.

Example 9-19 使用模式匹配来分析比较特殊的字符串

##################################Start Script###

```
 1 #!/bin/bash
 2
 3 var1=abcd-1234-defg
 4 echo "var1 = $var1"
 5
 6 t=${var1#*-*}
 7 echo "var1 (with everything, up to and including first - stripped out) = $t"
 8 #  t=${var1#*-}  在这个例子中作用是一样的,
 9 #+ 因为 # 匹配这个最近的字符串,
10 #+ 并且 * 匹配前边的任何字符串,包括一个空字符.
11 # (Thanks, Stephane Chazelas, for pointing this out.)
12
13 t=${var1##*-*}
14 echo "If var1 contains a \"-\", returns empty string...   var1 = $t"
15
16
17 t=${var1%*-*}
18 echo "var1 (with everything from the last - on stripped out) = $t"
19
20 echo
21
22 # ------------------------------------------
23 path_name=/home/bozo/ideas/thoughts.for.today
24 # ------------------------------------------
25 echo "path_name = $path_name"
26 t=${path_name##/*/}
27 echo "path_name, stripped of prefixes = $t"
```

```
28 # 在这个特定的例子中,与 t=`basename $path_name` 的作用一致.
29 #  t=${path_name%/}; t=${t##*/}   是一个更一般的解决办法,
30 #+ 但有时还是不行.
31 #  如果 $path_name 以一个新行结束, 那么`basename $path_name` 将不能工作,
32 #+ 但是上边这个表达式可以.
33 # (Thanks, S.C.)
34
35 t=${path_name%/*.*}
36 # 与 t=`dirname $path_name` 效果相同.
37 echo "path_name, stripped of suffixes = $t"
38 # 在某些情况下将失效,比如 "../", "/foo////", # "foo/", "/".
39 #  删除后缀,尤其是在basename没有后缀的时候,
40 #+ 但是dirname还是会使问题复杂化.
41 # (Thanks, S.C.)
42
43 echo
44
45 t=${path_name:11}
46 echo "$path_name, with first 11 chars stripped off = $t"
47 t=${path_name:11:5}
48 echo "$path_name, with first 11 chars stripped off, length 5 = $t"
49
50 echo
51
52 t=${path_name/bozo/clown}
53 echo "$path_name with \"bozo\" replaced  by \"clown\" = $t"
54 t=${path_name/today/}
55 echo "$path_name with \"today\" deleted = $t"
56 t=${path_name//o/O}
57 echo "$path_name with all o's capitalized = $t"
58 t=${path_name//o/}
59 echo "$path_name with all o's deleted = $t"
60
61 exit 0
```

################################End Script###

${var/#Pattern/Replacement}

如果var的前缀匹配到了Pattern,那么就用Replacement来替换Pattern.

${var/%Pattern/Replacement}

如果var的后缀匹配到了Pattern,那么就用Replacement来替换Pattern.

Example 9-20 对字符串的前缀或后缀使用匹配模式

################################Start Script###

```
1 #!/bin/bash
2 # var-match.sh:
3 # 对字符串的前后缀使用匹配替换的一个样本
4
5 v0=abc1234zip1234abc    # 原始变量.
6 echo "v0 = $v0"         # abc1234zip1234abc
7 echo
```

```
 8
 9 # 匹配字符串的前缀
10 v1=${v0/#abc/ABCDEF}    # abc1234zip1234abc
11                         # |-|
12 echo "v1 = $v1"         # ABCDEF1234zip1234abc
13                         # |----|
14
15 # 匹配字符串的后缀
16 v2=${v0/%abc/ABCDEF}    # abc1234zip123abc
17                         #            |-|
18 echo "v2 = $v2"         # abc1234zip1234ABCDEF
19                         #             |----|
20
21 echo
22
23 #  ----------------------------------------------------
24 #  必须在开头或结尾匹配,否则,
25 #+ 将不会产生替换结果.
26 #  ----------------------------------------------------
27 v3=${v0/#123/000}       # 匹配上了,但不是在字符串的开头
28 echo "v3 = $v3"         # abc1234zip1234abc
29                         # 没替换.
30 v4=${v0/%123/000}       # 匹配上了,但不是在字符串结尾.
31 echo "v4 = $v4"         # abc1234zip1234abc
32                         # 没替换.
33
34 exit 0
```

##################################End Script###

${!varprefix*}, ${!varprefix@}

使用变量的前缀来匹配前边所有声明过的变量.

```
 1 xyz23=whatever
 2 xyz24=
 3
 4 a=${!xyz*}      # 以"xyz"作为前缀,匹配所有前边声明过的变量.
 5 echo "a = $a"   # a = xyz23 xyz24
 6 a=${!xyz@}      # 同上.
 7 echo "a = $a"   # a = xyz23 xyz24
 8
 9 # Bash, version 2.04, 添加了这个特征.
```

注意事项:

[1] 如果在一个非交互脚本中,$parameter为空的话,那么这个脚本将以127返回.
(127退出码对应的Bash错误码为"command not found").

9.4 指定类型的变量:declare或者typeset

---

declare或者typeset内建命令(这两个命令是完全一样的)允许指定变量的具体类型.在某些特定的语言中,这是一种指定类型的很弱的形式.declare命令是在Bash版本2或之后的版本才被加入的.typeset命令也可以工作在ksh脚本中.

declare/typeset 选项

-r 只读

```
1 declare -r var1
```

(declare -r var1与readonly var1是完全一样的)

这和C语言中的const关键字一样,都是强制指定只读.如果你尝试修改一个只读变量的值,那么你将得到一个错误消息.

-i 整形

```
1 declare -i number
2 # 这个脚本将把变量"number"后边的赋值视为一个整形.
3
4 number=3
5 echo "Number = $number"     # Number = 3
6
7 number=three
8 echo "Number = $number"     # Number = 0
9 # 尝试把"three"解释为整形.
```

如果把一个变量指定为整形,那么即使没有expr和let命令,也允许使用特定的算术运算

```
1 n=6/3
2 echo "n = $n"       # n = 6/3
3
4 declare -i n
5 n=6/3
6 echo "n = $n"       # n = 2
```

-a 数组

```
1 declae -a indices
```

变量indices将被视为数组.

-f 函数

```
1 declare -f
```

如果使用declare -f而不带参数的话,将会列出这个脚本中之前定义的所有函数.

```
1 declare -f function_name
```

如果使用declare -f function_name这种形式的话,将只会列出这个函数的名字.

-x export

```
1 declare -x var3
```

这种使用方式,将会把var3 export出来.

Example 9-21 使用declare来指定变量的类型

```
##################################Start Script#########################################
1 #!/bin/bash
2
3 func1 ()
4 {
5 echo This is a function.
6 }
7
8 declare -f        # 列出之前的所有函数.
9
10 echo
11
12 declare -i var1   # var1 是个整形.
```

```
13 var1=2367
14 echo "var1 declared as $var1"
15 var1=var1+1       # 变量声明不需使用'let'命令.
16 echo "var1 incremented by 1 is $var1."
17 # 尝试将变量修改为整形.
18 echo "Attempting to change var1 to floating point value, 2367.1."
19 var1=2367.1       # 结果将是一个错误消息,并且变量并没有被修改.
20 echo "var1 is still $var1"
21
22 echo
23
24 declare -r var2=13.36        # 'declare' 允许设置变量的属性,
25                              #+ 并且同时分配变量的值.
26 echo "var2 declared as $var2" # 尝试修改只读变量.
27 var2=13.37                   # 产生一个错误消息,并且从脚本退出了.
28
29 echo "var2 is still $var2"    # 这行将不会被执行.
30
31 exit 0                       # 脚本将不会在此处退出.
#################################End Script#########################################
```

注意:使用declare内建命令将会限制变量的作用域.

```
 1 foo ()
 2 {
 3 FOO="bar"
 4 }
 5
 6 bar ()
 7 {
 8 foo
 9 echo $FOO
10 }
11
12 bar   # Prints bar.
```

然而...

```
 1 foo (){
 2 declare FOO="bar"
 3 }
 4
 5 bar ()
 6 {
 7 foo
 8 echo $FOO
 9 }
10
11 bar  # Prints nothing.
12
13
14 # Thank you, Michael Iatrou, for pointing this out.
```

9.5 变量的间接引用

假设一个变量的值是另一个变量的名字.我们有可能从第一个变量中取得第2个变量的值么?
比如,如果a=letter_of_alphabet接着letter_of_alphabet=z,那么我们能从a中得到z么?
答案是:当然可以,并且这被称为间接引用.它使用一个不常用的符号eval var1=\$$var2.
Example 9-22 间接引用

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # ind-ref.sh: 间接变量引用
 3 # 存取一个变量的值的值(这里翻译得有点拗口,不过凑合吧)
 4
 5 a=letter_of_alphabet   # 变量"a"的值是另一个变量的名字.
 6 letter_of_alphabet=z
 7
 8 echo
 9
10 # 直接引用.
11 echo "a = $a"          # a = letter_of_alphabet
12
13 # 间接引用.
14 eval a=\$$a
15 echo "Now a = $a"      # Now a = z
16
17 echo
18
19
20 # 现在,让我们试试修改第2个引用的值.
21
22 t=table_cell_3
23 table_cell_3=24
24 echo "\"table_cell_3\" = $table_cell_3"           # "table_cell_3" = 24
25 echo -n "dereferenced \"t\" = "; eval echo \$$t    # 解引用 "t" = 24
26 # 在这个简单的例子中,下边的表达式也能正常工作(为什么?).
27 #         eval t=\$$t; echo "\"t\" = $t"
28
29 echo
30
31 t=table_cell_3
32 NEW_VAL=387
33 table_cell_3=$NEW_VAL
34 echo "Changing value of \"table_cell_3\" to $NEW_VAL."
35 echo "\"table_cell_3\" now $table_cell_3"
36 echo -n "dereferenced \"t\" now "; eval echo \$$t
37 # "eval" 将获得两个参数 "echo" 和 "\$$t" (与$table_cell_3等价)
38
39 echo
40
```

```
41 # (Thanks, Stephane Chazelas, 澄清了上边的行为.)
42
43
44 # 另一个方法是使用${!t}符号,见"Bash, 版本2"小节.
45 # 也请参阅ex78.sh.
46
47 exit 0
```

################################End Script###

间接应用到底有什么应用价值?它给Bash添加了一种类似于C语言指针的功能,在Example 34-3中有例子.并且,还有一些其它的有趣的应用....

Nils Radtke展示了如何建立一个"dynamic"变量名字并且取出其中的值.当sourcing(包含)配置文件时,这很有用.

################################Start Script###

```
 1 #!/bin/bash
 2
 3
 4 # ----------------------------------------------
 5 # 这部分内容可能来自于单独的文件.
 6 isdnMyProviderRemoteNet=172.16.0.100
 7 isdnYourProviderRemoteNet=10.0.0.10
 8 isdnOnlineService="MyProvider"
 9 # ----------------------------------------------
10
11
12 remoteNet=$(eval "echo \$$(echo isdn${isdnOnlineService}RemoteNet)")
13 remoteNet=$(eval "echo \$$(echo isdnMyProviderRemoteNet)")
14 remoteNet=$(eval "echo \$isdnMyProviderRemoteNet")
15 remoteNet=$(eval "echo $isdnMyProviderRemoteNet")
16
17 echo "$remoteNet"    # 172.16.0.100
18
19 # =================================================================
20
21 #  同时,它甚至能更好.
21 #
22
23 #  考虑下边的脚本,给出了一个变量getSparc,
24 #+ 但是没给出变量getIa64:
25
26 chkMirrorArchs () {
27   arch="$1";
28   if [ "$(eval "echo \${$(echo get$(echo -ne $arch |
29         sed 's/^\(.\).*/\1/g' | tr 'a-z' 'A-Z'; echo $arch |
30         sed 's/^.\(.*\)/\1/g')):-false}")" = true ]
31   then
32     return 0;
33   else
34     return 1;
```

```
35   fi;
36 }
37
38 getSparc="true"
39 unset getIa64
40 chkMirrorArchs sparc
41 echo $?       # 0
42              # True
43
44 chkMirrorArchs Ia64
45 echo $?       # 1
46              # False
47
48 # 注意:
49 # -----<rojy bug>
50 # Even the to-be-substituted variable name part is built explicitly.
51 # The parameters to the chkMirrorArchs calls are all lower case.
52 # The variable name is composed of two parts: "get" and "Sparc" . . .
```

################################End Script###

Example 9-23 传递一个间接引用给awk

################################Start Script###

```
 1 #!/bin/bash
 2
 3 #  "column totaler"脚本的另一个版本
 4 #+ 这个版本在目标文件中添加了一个特殊的列(数字的).
 5 #  这个脚本使用了间接引用.
 6
 7 ARGS=2
 8 E_WRONGARGS=65
 9
10 if [ $# -ne "$ARGS" ] # 检查命令行参数是否是合适的个数.
11 then
12   echo "Usage: `basename $0` filename column-number"
13   exit $E_WRONGARGS
14 fi
15
16 filename=$1
17 column_number=$2
18
19 #===== 上边的这部分,与原来的脚本一样 =====#
20
21
22 # 一个多行的awk脚本被调用,通过 ' ..... '
23
24
25 # awk 脚本开始.
26 # ------------------------------------------------
27 awk "
```

```
28
29 { total += \$${column_number} # 间接引用.
30 }
31 END {
32     print total
33     }
34
35     " "$filename"
36 # ---------------------------------------------
37 # awk 脚本结束.
38
39 #  间接的变量引用避免了在一个内嵌的awk脚本中引用
40 #+ 一个shell变量的问题.
41 #  Thanks, Stephane Chazelas.
42
43
44 exit 0
```

################################End Script##

注意: 这个脚本有些狡猾.如果第2个变量修改了它的值,那么第一个变量必须被适当的解引用(像上边的例子一样).幸运的是,在Bash版本2中引入的${!variable}(参见Example 34-2)是的间接引用更加直观了.

注意: Bash并不支持指针的算术运算,并且这严格的限制了间接引用的使用.事实上,在脚本语言中,间接引用本来就是丑陋的部分.

9.6 $RANDOM: 产生随机整数

$RANDOM是Bash的内部函数(并不是常量),这个函数将返回一个范围在0 - 32767之间的一个伪随机整数.它不应该被用来产生密匙.

Example 9-24 产生随机数

################################Start Script##

```
 1 #!/bin/bash
 2
 3 # $RANDOM 在每次调用的时候,返回一个不同的随机整数.
 4 # 指定的范围是: 0 - 32767 (有符号的16-bit 整数).
 5
 6 MAXCOUNT=10
 7 count=1
 8
 9 echo
10 echo "$MAXCOUNT random numbers:"
11 echo "-----------------"
12 while [ "$count" -le $MAXCOUNT ]      # 产生10 ($MAXCOUNT) 个随机整数.
13 do
14   number=$RANDOM
15   echo $number
16   let "count += 1"  # 数量加1.
17 done
18 echo "-----------------"
```

```
19
20 # 如果你需要在一个特定范围内产生一个随机int,那么使用'modulo'(模)操作.
21 # 这将返回一个除法操作的余数.
22
23 RANGE=500
24
25 echo
26
27 number=$RANDOM
28 let "number %= $RANGE"
29 #           ^^
30 echo "Random number less than $RANGE  ---  $number"
31
32 echo
33
34
35
36 #  如果你需要产生一个比你指定的最小边界大的随机数,
37 #+ 那么建立一个test循环,来丢弃所有产生对比这个数小的随机数.
38
39 FLOOR=200
40
41 number=0   #initialize
42 while [ "$number" -le $FLOOR ]
43 do
44   number=$RANDOM
45 done
46 echo "Random number greater than $FLOOR ---  $number"
47 echo
48
49   # 让我们对上边的循环尝试一个小改动,也就是
50   #       让"number = $RANDOM + $FLOOR"
51   # 这将不再需要那个while循环,并且能够运行得更快.
52   # 但是, 这可能会产生一个问题.那么这个问题是什么呢?(译者:这很简单,有可能溢出)
53
54
55
56 # 结合上边两个例子的技术,来达到获得在指定的上下限之间来产生随机数.
57 number=0   #initialize
58 while [ "$number" -le $FLOOR ]
59 do
60   number=$RANDOM
61   let "number %= $RANGE"  # 让$number依比例落在$RANGE范围内.
62 done
63 echo "Random number between $FLOOR and $RANGE ---  $number"
64 echo
65
66
```

```
67
68 # 产生一个二元选择,就是"true"和"false"两个值.
69 BINARY=2
70 T=1
71 number=$RANDOM
72
73 let "number %= $BINARY"
74 #  注意,让"number >>= 14" 将给出一个更好的随机分配
75 #+ (右移14位将把所有为全部清空,除了第15位,因为有符号,所以第16位是符号位).
76 if [ "$number" -eq $T ]
77 then
78   echo "TRUE"
79 else
80   echo "FALSE"
81 fi
82
83 echo
84
85
86 # 抛骰子
87 SPOTS=6   # 模6给出的范围就是0-5.
88         # 加1就会得到期望的范围1 - 6.
89         # Thanks, Paulo Marcel Coelho Aragao, for the simplification.
90 die1=0
91 die2=0
92 # 是否让SPOTS=7比加1更好呢?解释行或者不行的原因?
93
94 # 每次抛骰子,都会给出均等的机会.
95
96    let "die1 = $RANDOM % $SPOTS +1" # 抛第一次.
97    let "die2 = $RANDOM % $SPOTS +1" # 抛第二次.
98    #  上边的那个算术操作,具有更高的优先级呢 --
99    #+ 模操作(%)还是加法操作(+)?
100
101
102 let "throw = $die1 + $die2"
103 echo "Throw of the dice = $throw"
104 echo
105
106
107 exit 0
```

################################End Script###

Example 9-25 从一副扑克牌中取出一张随机的牌

################################Start Script###

```
1 #!/bin/bash
2 # pick-card.sh
3
4 # 这是一个从数组中取出随机元素的一个例子.
```

```
 5
 6
 7 # 取出一张牌,任何一张.
 8
 9 Suites="Clubs
10 Diamonds
11 Hearts
12 Spades"
13
14 Denominations="2
15 3
16 4
17 5
18 6
19 7
20 8
21 9
22 10
23 Jack
24 Queen
25 King
26 Ace"
27
28 # 注意变量的多行展开.
29
30
31 suite=($Suites)              # 读到数组变量中.
32 denomination=($Denominations)
33
34 num_suites=${#suite[*]}       # 计算有多少个元素.
35 num_denominations=${#denomination[*]}
36
37 echo -n "${denomination[$((RANDOM%num_denominations))]} of "
38 echo ${suite[$((RANDOM%num_suites))]}
39
40
41 # $bozo sh pick-cards.sh
42 # Jack of Clubs
43
44
45 # Thank you, "jipe," for pointing out this use of $RANDOM.
46 exit 0
```

################################End Script###

Jipe展示了一系列的在一定范围中产生随机数的方法.

```
 1 #  在6到30之间产生随机数.
 2    rnumber=$((RANDOM%25+6))
 3
 4 #  还是产生6-30之间的随机数,
```

```
 5 #+ 但是这个数字必须被3均分.
 6    rnumber=$(((RANDOM%30/3+1)*3))
 7
 8 #  注意,这可能不会在所有时候都能正常地运行.
 9 #  It fails if $RANDOM returns 0.
10
11 #  Frank Wang 建议用下班的方法来取代:
12    rnumber=$(( RANDOM%27/3*3+6 ))
```

Bill Gradwohl 提出了一个重要的规则来产生正数.

```
1 rnumber=$(((RANDOM%(max-min+divisibleBy))/divisibleBy*divisibleBy+min))
```

这里Bill给出了一个通用函数,这个函数返回一个在两个指定值之间的随机数

Example 9-26 两个指定值之间的随机数

##################################Start Script###

```
 1 #!/bin/bash
 2 # random-between.sh
 3 # 在两个指定值之间的随机数.
 4 # Bill Gradwohl编写的本脚本,本文作者作了较小的修改.
 5 # 允许使用.
 6
 7
 8 randomBetween() {
 9   # 产生一个正的或者负的随机数.
10    #+ 在$max和$max之间
11    #+ 并且可被$divisibleBy整除的.
12    #  给出一个合理的随机分配的返回值.
13    #
14    #  Bill Gradwohl - Oct 1, 2003
15
16    syntax() {
17    # 在函数中内嵌函数
18      echo
19      echo    "Syntax: randomBetween [min] [max] [multiple]"
20      echo
21      echo    "Expects up to 3 passed parameters, but all are completely optional."
22      echo    "min is the minimum value"
23      echo    "max is the maximum value"
24      echo    "multiple specifies that the answer must be a multiple of this value."
25      echo    "    i.e. answer must be evenly divisible by this number."
26      echo
27      echo    "If any value is missing, defaults area supplied as: 0 32767 1"
28      echo    "Successful completion returns 0, unsuccessful completion returns"
29      echo    "function syntax and 1."
30      echo    "The answer is returned in the global variable randomBetweenAnswer"
31      echo    "Negative values for any passed parameter are handled correctly."
32    }
33
34    local min=${1:-0}
35    local max=${2:-32767}
```

```
36  local divisibleBy=${3:-1}
37  # 默认值分配,用来处理没有参数传递进来的时候.
38
39  local x
40  local spread
41
42  # 确认divisibleBy是正值.
43  [ ${divisibleBy} -lt 0 ] && divisibleBy=$((0-divisibleBy))
44
45  # 完整性检查.
46  if [ $# -gt 3 -o ${divisibleBy} -eq 0 -o  ${min} -eq ${max} ]; then
47    syntax
48    return 1
49  fi
50
51  # 察看是否min和max颠倒了.
52  if [ ${min} -gt ${max} ]; then
53    # 交换它们.
54    x=${min}
55    min=${max}
56    max=${x}
57  fi
58
59  #  如果min自己并不能够被$divisibleBy整除,
60  #+ 那么就调整min的值,使其能够被$divisibleBy整除,前提是不能放大范围.
61  if [ $((min/divisibleBy*divisibleBy)) -ne ${min} ]; then
62    if [ ${min} -lt 0 ]; then
63      min=$((min/divisibleBy*divisibleBy))
64    else
65      min=$((((min/divisibleBy)+1)*divisibleBy))
66    fi
67  fi
68
69  #  如果min自己并不能够被$divisibleBy整除,
70  #+ 那么就调整max的值,使其能够被$divisibleBy整除,前提是不能放大范围.
71  if [ $((max/divisibleBy*divisibleBy)) -ne ${max} ]; then
72    if [ ${max} -lt 0 ]; then
73      max=$((((max/divisibleBy)-1)*divisibleBy))
74    else
75      max=$((max/divisibleBy*divisibleBy))
76    fi
77  fi
78
79  #  ----------------------------------------------------------------
80  #  现在,来做真正的工作.
81
82  #  注意,为了得到对于端点来说合适的分配,
83  #+ 随机值的范围不得不落在
```

```
84   #+ 0 和 abs(max-min)+divisibleBy之间, 而不是 abs(max-min)+1.
85
86   #  对于端点来说,
87   #+ 这个少量的增加将会产生合适的分配.
88
89   #  修改这个公式,使用abs(max-min)+1来代替abs(max-min)+divisibleBy的话,
90   #+ 也能够产生正确的答案, 但是在这种情况下生成的随机值对于正好为端点倍数
91   #+ 的这种情况来说将是不完美的,因为在正好为端点倍数的情况的随机率比较低,
92   #+ 因为你才加1而已,这比正常的公式所产生的机率要小得多(正常为加divisibleBy)
93   #  ----------------------------------------------------------------
94
95   spread=$((max-min))
96   [ ${spread} -lt 0 ] && spread=$((0-spread))
97   let spread+=divisibleBy
98   randomBetweenAnswer=$(((RANDOM%spread)/divisibleBy*divisibleBy+min))
99
100   return 0
101
102   #  然而,Paulo Marcel Coelho Aragao指出
103   #+ 当$max和$min不能被$divisibleBy整除时,
104   #+ 这个公式将会失败.
105   #
106   #  他建议使用如下的公式:
107   #    rnumber = $(((RANDOM%(max-min+1)+min)/divisibleBy*divisibleBy))
108
109 }
110
111 # 让我们测试一下这个函数.
112 min=-14
113 max=20
114 divisibleBy=3
115
116
117 #  产生一个数组answers,answers的下标用来表示在范围内可能出现的值,
118 #+ 而内容记录的是对于这个值出现的次数,如果我们循环足够多次,一定会得到
119 #+ 一次出现机会.
120 declare -a answer
121 minimum=${min}
122 maximum=${max}
123   if [ $((minimum/divisibleBy*divisibleBy)) -ne ${minimum} ]; then
124     if [ ${minimum} -lt 0 ]; then
125       minimum=$((minimum/divisibleBy*divisibleBy))
126     else
127       minimum=$((((minimum/divisibleBy)+1)*divisibleBy))
128     fi
129   fi
130
131
```

```
132   #  如果maximum自己并不能够被$divisibleBy整除,
133   #+ 那么就调整maximum的值,使其能够被$divisibleBy整除,前提是不能放大范围.
134
135   if [ $((maximum/divisibleBy*divisibleBy)) -ne ${maximum} ]; then
136     if [ ${maximum} -lt 0 ]; then
137       maximum=$((((maximum/divisibleBy)-1)*divisibleBy))
138     else
139       maximum=$((maximum/divisibleBy*divisibleBy))
140     fi
141   fi
142
143
144 #  我们需要产生一个下标全为正的数组,
145 #+ 所以我们需要一个displacement来保正都为正的结果.
146
147
148 displacement=$((0-minimum))
149 for ((i=${minimum}; i<=${maximum}; i+=divisibleBy)); do
150   answer[i+displacement]=0
151 done
152
153
154 # 现在我们循环足够多的次数来得到我们想要的答案.
155 loopIt=1000   #  脚本作者建议 100000,
156             #+ 但是这实在是需要太长的时间了.
157
158 for ((i=0; i<${loopIt}; ++i)); do
159
160   #  注意,我们在这里调用randomBetween函数时,故意将min和max颠倒顺序
161   #+ 我们是为了测试在这种情况下,此函数是否还能得到正确的结果.
162
163   randomBetween ${max} ${min} ${divisibleBy}
164
165   # 如果答案不是我们所预期的,那么就报告一个错误.
166   [ ${randomBetweenAnswer} -lt ${min} -o ${randomBetweenAnswer} -gt ${max} ] && echo MIN or MAX error -
${randomBetweenAnswer}!
167   [ $((randomBetweenAnswer%${divisibleBy})) -ne 0 ] && echo DIVISIBLE BY error - ${randomBetweenAnswer}!
168
169   # 将统计值存到answer之中.
170   answer[randomBetweenAnswer+displacement]=$((answer[randomBetweenAnswer+displacement]+1))
171 done
172
173
174
175 # 让我们察看一下结果
176
177 for ((i=${minimum}; i<=${maximum}; i+=divisibleBy)); do
178   [ ${answer[i+displacement]} -eq 0 ] && echo "We never got an answer of $i." || echo "${i} occurred
```

```
${answer[i+displacement]} times."
179 done
180
181
182 exit 0
```

################################End Script###

$RANDOM到底有多随机?最好的办法就是写个脚本来测试一下.跟踪随机数的分配情况.

让我们用随机数摇一个骰子.

Example 9-27 使用随机数来摇一个骰子

################################Start Script###

```
 1 #!/bin/bash
 2 # RANDOM到底有多random?
 3
 4 RANDOM=$$      # 使用脚本的进程ID来作为随机数的产生种子.
 5
 6 PIPS=6        # 一个骰子有6面.
 7 MAXTHROWS=600   # 如果你没别的事干,那么可以增加这个数值.
 8 throw=0        # 抛骰子的次数.
 9
10 ones=0        #  必须把所有count都初始化为0,
11 twos=0        #+ 因为未初始化的变量为null,不是0.
12 threes=0
13 fours=0
14 fives=0
15 sixes=0
16
17 print_result ()
18 {
19 echo
20 echo "ones =   $ones"
21 echo "twos =   $twos"
22 echo "threes = $threes"
23 echo "fours =  $fours"
24 echo "fives =  $fives"
25 echo "sixes =  $sixes"
26 echo
27 }
28
29 update_count()
30 {
31 case "$1" in
32   0) let "ones += 1";;   # 因为骰子没有0,所以给1.
33   1) let "twos += 1";;   # 对tows做同样的事.
34   2) let "threes += 1";;
35   3) let "fours += 1";;
36   4) let "fives += 1";;
37   5) let "sixes += 1";;
38 esac
```

```
39 }
40
41 echo
42
43
44 while [ "$throw" -lt "$MAXTHROWS" ]
45 do
46   let "die1 = RANDOM % $PIPS"
47   update_count $die1
48   let "throw += 1"
49 done
50
51 print_result
52
53 exit 0
54
55 #  如果RANDOM是真正的随机,那么摇出来结果应该平均的.
56 #  $MAXTHROWS设为600,那么每面都应该为100,上下的出入不应该超过20.
57 #
58 #  记住RANDOM毕竟只是一个伪随机数,
59 #+ 并且不是十分完美的.
60
61 #  随机数的产生是一个深奥并复杂的问题.
62 #  足够长的随机序列,不但会展现杂乱无章的一面,
63 #+ 而且会展现机会均等的一面.
64
65 # 一个很简单的练习:
66 # -----------------
67 # 重写这个例子,做成抛1000次硬币的形式.
68 # 分为正反两面.
```

################################End Script###

像我们在上边的例子中看到的,最好在每次随机数产生时都使用新的种子.应为如果使用同样的种子的话,那么随机数将产生相同的序列.[2](C中random()函数也会有这样的行为)

Example 9-28 重新分配随机数种子

################################Start Script###

```
 1 #!/bin/bash
 2 # seeding-random.sh: 设置RANDOM变量作为种子.
 3
 4 MAXCOUNT=25       # 决定产生多少个随机数.
 5
 6 random_numbers ()
 7 {
 8 count=0
 9 while [ "$count" -lt "$MAXCOUNT" ]
10 do
11   number=$RANDOM
12   echo -n "$number "
13   let "count += 1"
```

```
14 done
15 }
16
17 echo; echo
18
19 RANDOM=1        # 为随机数的产生设置RANDOM种子.
20 random_numbers
21
22 echo; echo
23
24 RANDOM=1        # 设置同样的种子...
25 random_numbers  # ...将会和上边产生的随机数列相同.
26               #
27               # 复制一个相同的随机数序列在什么时候有用呢?
28
29 echo; echo
30
31 RANDOM=2        # 再试一下,但这次使用不同的种子...
32 random_numbers  # 将给出一个不同的随机数序列.
33
34 echo; echo
35
36 # RANDOM=$$  使用脚本的进程id 作为随机数的种子.
37 # 从'time'或'date'命令中取得RANDOM作为种子也是很常用的办法.
38
39 # 一个有想象力的方法...
40 SEED=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }')
41 #  首先从/dev/urandom(系统伪随机设备文件)中取出1行,
42 #+ 然后着这个可打印行转换为(8进制)数,通过使用"od"命令,
43 #+ 最后使用"awk"来获得一个数,
44 #+ 这个数将作为随机数产生的种子.
45 RANDOM=$SEED
46 random_numbers
47
48 echo; echo
49
50 exit 0
##################################End Script#########################################
```

注意:/dev/urandom设备文件提供了一种比单独使用$RANDOM更好的,能产生更"随机"的随机数
的方法.
dd if=/dev/urandom of=targetfile bs=1 count=XX能够产生一个很分散的为随机数.
然而,将这个数赋值到一个脚本文件的变量中,还需要可操作性,比如使用"od"命令
(就像上边的例子,见Example 12-13),或者使用dd命令(见Example 12-55),或者管道到
"md5sum"命令中(见Example 33-14).

当然还有其它的产生伪随机数的方法.Awk就可以提供一个方便的方法.
Example 9-29 使用awk产生伪随机数

```
##################################Start Script#######################################
```

```
 1 #!/bin/bash
 2 # random2.sh: 产生一个范围0 - 1的为随机数.
 3 # 使用awk的rand()函数.
 4
 5 AWKSCRIPT=' { srand(); print rand() } '
 6 #           Command(s) / 传到awk中的参数
 7 # 注意,srand()函数用来产生awk的随机数种子.
 8
 9
10 echo -n "Random number between 0 and 1 = "
11
12 echo | awk "$AWKSCRIPT"
13 # 如果你省去'echo'那么将发生什么?
14
15 exit 0
16
17
18 # Exercises:
18 # 练习:
19 # -----
20
21 # 1) 使用循环结构,打印出10个不同的随机数.
22 #      (提示: 在循环的每次执行过程中,你必须使用"srand()"函数来生成不同的
23 #+     种子.如果你没做这件事那么将发生什么?
24
25 # 2) 使用一个整数乘法作为一个放缩因子,在10到100的范围之间,
26 #+   来产生随机数.
27
28 # 3) 同上边的练习 #2,但这次产生随机整数.
```

################################End Script###

"data"命令也可以用来产生伪随机整数序列.

注意事项:

[1] 真正的随机事件(在它存在的范围内),只发生在特定的几个未知的自然界现象中,比如放射性衰变.计算机只能产生模拟的随机事件,并且计算机产生的"随机"数因此只能称为伪随机数.

[2] 计算机产生的伪随机数序列用的种子可以被看成是一种标识标签.比如,使用种子23所产生的伪随机数序列就被称作序列#23.

一个伪随机序列的特点就是在这个序列开始重复之前的所有元素的个数的和,也就是这个序列的长度.一个好的伪随机产生算法将可以产生一个非常长的不重复的序列.

9.7 双圆括号结构

((...))与let命令很像,允许算术扩展和赋值.举个简单的例子a=$((5 + 3)),将把a设为"5+3"或者8.然而,双圆括号也是一种在Bash中允许使用C风格的变量处理的机制. Example 9-30 C风格的变量处理

################################Start Script###

```
 1 #!/bin/bash
 2 # 处理一个变量,C风格,使用((...))结构.
 3
```

```
4
5 echo
6
7 (( a = 23 ))  # 给一个变量赋值,从"="两边的空格就能看出这是c风格的处理.
8 echo "a (initial value) = $a"
9
10 (( a++ ))     # 变量'a'后加1,C风格.
11 echo "a (after a++) = $a"
12
13 (( a-- ))     # 变量'a'后减1,C风格.
14 echo "a (after a--) = $a"
15
16
17 (( ++a ))     # 变量'a'预加1,C风格.
18 echo "a (after ++a) = $a"
19
20 (( --a ))     # 变量'a'预减1,C风格.
21 echo "a (after --a) = $a"
22
23 echo
24
25 ############################################################
26 #  注意:在C语言中,预减和后减操作
27 #+ 会有些不同的副作用.
28
29 n=1; let --n && echo "True" || echo "False"  # False
30 n=1; let n-- && echo "True" || echo "False"  # True
31
32 #  Thanks, Jeroen Domburg.
33 ############################################################
34
35 echo
36
37 (( t = a<45?7:11 ))   # C风格的3元操作.
38 echo "If a < 45, then t = 7, else t = 11."
39 echo "t = $t "        # Yes!
40
41 echo
42
43
44 # ----------------
45 # 复活节彩蛋注意!
46 # ----------------
47 #  Chet Ramey 显然的偷偷摸摸的做了一些未公开的C风格的结构
48 #+ 放在Bash中(准确地说是根据ksh来改写的,这更接近些)
49 #  在Bash文档中,Ramey调用((...))shell算法,
50 #+ 但是它可以走得更远.
51 #  对不起, Chet, 现在秘密被公开了.
```

```
 52
 53 # See also "for" and "while" loops using the ((...)) construct.
 53 # 也参考一些"for"和"while"循环中使用((...))结构的例子.
 54
 55 # 这些只能工作在2.04或者更高版本的Bash中.
 56
 57 exit 0
```

################################End Script###

见Example 10-12.

[]

高级Bash脚本编程指南(三)（上）

文章整理: 文章来源: 网络

高级Bash脚本编程指南(三)

第10章 循环和分支

==========

对代码块进行操作是有组织的结构化的shell脚本的关键.为了达到这个目的,循环和分支提供帮助.

10.1 循环

循环就是重复一些命令的代码块,如果条件不满足就退出循环.

for loops

for arg in [list]

这是一个基本的循环结构.它与C的相似结构有很大不同.

```
for arg in [list]
do
 command(s)...
done
```

注意:在循环的每次执行中,arg将顺序的存取list中列出的变量.

```
1 for arg in "$var1" "$var2" "$var3" ... "$varN"
2 # 在第1次循环中, arg = $var1
3 # 在第2次循环中, arg = $var2
4 # 在第3次循环中, arg = $var3
5 # ...
6 # 在第n次循环中, arg = $varN
7
8 # 在[list]中的参数加上双引号是为了阻止单词分离.
```

list中的参数允许包含通配符.

如果do和for想在同一行出现,那么在它们之间需要添加一个";".

```
for arg in [list]; do
```

Example 10-1 循环的一个简单例子

################################Start Script###

```
1 #!/bin/bash
2 # 列出所有行星.
3
4 for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
5 do
6   echo $planet  # Each planet on a separate line.
```

```
 7 done
 8
 9 echo
10
11 for planet in "Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto"
12 # 所有的行星都在同一行上.
13 # 完整的'list'作为一个变量都封在""中
14 do
15   echo $planet
16 done
17
18 exit 0
```

################################End Script###

注意:每个[list]中的元素都可能包含多个参数.在处理参数组时,这是非常有用的.
在这种情况下,使用set命令(见Example 11-15)来强制解析每个[list]中的元素,
并且分配每个解析出来的部分到一个位置参数中.

Example 10-2 每个[list]元素带两个参数的for循环

################################Start Script###

```
 1 #!/bin/bash
 2 # 还是行星.
 3
 4 # 分配行星的名字和它距太阳的距离.
 5
 6 for planet in "Mercury 36" "Venus 67" "Earth 93"  "Mars 142" "Jupiter 483"
 7 do
 8   set -- $planet  # 解析变量"planet"并且设置位置参数.
 9   # "--" 将防止$planet为空,或者是以一个破折号开头.
10
11   # 可能需要保存原始的位置参数,因为它们被覆盖了.
12   # 一种方法就是使用数组,
13   #       original_params=("$@")
14
15   echo "$1  $2,000,000 miles from the sun"
16   #-------two  tabs---把后边的0和$2连接起来
17 done
18
19 # (Thanks, S.C., for additional clarification.)
20
21 exit 0
```

################################End Script###

可以在for循环中的[list]位置放入一个变量

Example 10-3 文件信息:对包含在变量中的文件列表进行操作

################################Start Script###

```
 1 #!/bin/bash
 2 # fileinfo.sh
 3
 4 FILES="/usr/sbin/accept
 5 /usr/sbin/pwck
```

```
 6 /usr/sbin/chroot
 7 /usr/bin/fakefile
 8 /sbin/badblocks
 9 /sbin/ypbind"     # 你关心的文件列表.
10                 # 扔进去一个假文件, /usr/bin/fakefile.
11
12 echo
13
14 for file in $FILES
15 do
16
17   if [ ! -e "$file" ]      # 检查文件是否存在.
18   then
19     echo "$file does not exist."; echo
20     continue              # 继续下一个.
21    fi
22
23   ls -l $file | awk '{ print $9 "        file size: " $5 }'  # 打印2个域.
24   whatis `basename $file`   # 文件信息.
25   # 注意whatis数据库需要提前建立好.
26   # 要想达到这个目的, 以root身份运行/usr/bin/makewhatis.
27   echo
28 done
29
30 exit 0
```

################################End Script###

如果在for循环的[list]中有通配符(*和?),那将会产生文件名扩展,也就是file globbing.

Example 10-4 在for循环中操作文件

################################Start Script###

```
 1 #!/bin/bash
 2 # list-glob.sh: 产生 [list] 在for循环中, 使用 "globbing"
 3
 4 echo
 5
 6 for file in *
 7 #          ^ 在表达式中识别file globbing时,
 8 #+           Bash 将执行文件名扩展
 9 do
10   ls -l "$file"  # 列出所有在$PWD(当前目录)中的所有文件.
11   #  回想一下,通配符"*"能够匹配所有文件,
12   #+ 然而,在"globbing"中,是不能比配"."文件的.
13
14   #  If the pattern matches no file, it is expanded to itself.
14   #  如果没匹配到任何文件,那它将扩展成自己.
15   #  为了不让这种情况发生,那就设置nullglob选项
16   #+   (shopt -s nullglob).
17   #  Thanks, S.C.
18 done
```

```
19
20 echo; echo
21
22 for file in [jx]*
23 do
24   rm -f $file    # 只删除当前目录下以"j"或"x"开头的文件.
25   echo "Removed file \"$file\"".
26 done
27
28 echo
29
30 exit 0
```

################################End Script###

在一个for循环中忽略[list]的话,将会使循环操作$@(从命令行传递给脚本的参数列表).

一个非常好的例子,见Example A-16.

Example 10-5 在for循环中省略[list]

################################Start Script###

```
 1 #!/bin/bash
 2
 3 #  使用两种方法来调用这个脚本,一种是带参数的情况,另一种不带参数.
 4 #+ 观察此脚本的行为各是什么样的?
 5
 6 for a
 7 do
 8  echo -n "$a "
 9 done
10
11 #  The 'in list' missing, therefore the loop operates on '$@'
11 #  没有[list],所以循环将操作'$@'
12 #+ (包括空白的命令参数列表).
13
14 echo
15
16 exit 0
```

################################End Script###

也可以使用命令替换来产生for循环的[list].具体见Example 12-49,Example 10-10,

和Example 12-43.

Example 10-6 使用命令替换来产生for循环的[list]

################################Start Script###

```
 1 #!/bin/bash
 2 #  for-loopcmd.sh: 带[list]的for循环
 3 #+ [list]是由命令替换产生的.
 4
 5 NUMBERS="9 7 3 8 37.53"
 6
 7 for number in `echo $NUMBERS`  # for number in 9 7 3 8 37.53
 8 do
 9   echo -n "$number "
```

```
10 done
11
12 echo
13 exit 0
```

################################End Script###

下边是一个用命令替换来产生[list]的更复杂的例子.

Example 10-7 对于二进制文件的一个grep替换

################################Start Script###

```
 1 #!/bin/bash
 2 # bin-grep.sh: 在一个二进制文件中定位匹配字串.
 3
 4 # 对于二进制文件的一个grep替换
 5 # 与"grep -a"的效果相似
 6
 7 E_BADARGS=65
 8 E_NOFILE=66
 9
10 if [ $# -ne 2 ]
11 then
12   echo "Usage: `basename $0` search_string filename"
13   exit $E_BADARGS
14 fi
15
16 if [ ! -f "$2" ]
17 then
18   echo "File \"$2\" does not exist."
19   exit $E_NOFILE
20 fi
21
22
23 IFS="\n"        # 由Paulo Marcel Coelho Aragao提出的建议.
24 for word in $( strings "$2" | grep "$1" )
25 # "strings" 命令列出二进制文件中的所有字符串.
26 # 输出到管道交给"grep",然后由grep命令来过滤字符串.
27 do
28   echo $word
29 done
30
31 # S.C. 指出, 行23 - 29 可以被下边的这行来代替,
32 #    strings "$2" | grep "$1" | tr -s "$IFS" '[\n*]'
33
34
35 # 试试用"./bin-grep.sh mem /bin/ls"来运行这个脚本.
36
37 exit 0
```

################################End Script###

大部分相同.

Example 10-8 列出系统上的所有用户

################################Start Script###

```
 1 #!/bin/bash
 2 # userlist.sh
 3
 4 PASSWORD_FILE=/etc/passwd
 5 n=1           # User number
 6
 7 for name in $(awk 'BEGIN{FS=":"}{print $1}' < "$PASSWORD_FILE" )
 8 # 域分隔   = :          ^^^^^^
 9 # 打印出第一个域               ^^^^^^^^
10 # 从password文件中取得输入                ^^^^^^^^^^^^^^^^
11 do
12   echo "USER #$n = $name"
13   let "n += 1"
14 done
15
16
17 # USER #1 = root
18 # USER #2 = bin
19 # USER #3 = daemon
20 # ...
21 # USER #30 = bozo
22
23 exit 0
24
25 #  练习 :
26 #  ------
27 #  一个普通用户(或者是一个普通用户运行的脚本)
28 #+ 怎么能读取/etc/password呢?
29 #  这是否是一个安全漏洞? 为什么是?为什么不是?
```

################################End Script###

关于用命令替换来产生[list]的最后的例子.

Example 10-9 在目录的所有文件中查找源字串

################################Start Script###

```
 1 #!/bin/bash
 2 # findstring.sh:
 3 # 在一个指定目录的所有文件中查找一个特定的字符串.
 4
 5 directory=/usr/bin/
 6 fstring="Free Software Foundation"  # 查看那个文件中包含FSF.
 7
 8 for file in $( find $directory -type f -name '*' | sort )
 9 do
10   strings -f $file | grep "$fstring" | sed -e "s%$directory%%"
11   #  在"sed"表达式中,
12   #+ 我们必须替换掉正常的替换分隔符"/",
13   #+ 因为"/"碰巧是我们需要过滤的字串之一.
14   #  如果不用"%"代替"/"作为分隔符,那么这个操作将失败,并给出一个错误消息.(试试)
```

```
15 done
16
17 exit 0
18
19 #  练习 (easy):
20 #  ------------
21 #  将内部用的$directory和$fstring变量,用从
22 #+ 命令行参数代替.
```

################################End Script###

for循环的输出也可以通过管道传递到一个或多个命令中.

Example 10-10 列出目录中所有的符号连接文件

################################Start Script###

```
 1 #!/bin/bash
 2 # symlinks.sh: 列出目录中所有的符号连接文件.
 3
 4
 5 directory=${1-`pwd`}
 6 #  如果没有其他的特殊指定,
 7 #+ 默认为当前工作目录.
 8 #  下边的代码块,和上边这句等价.
 9 # ------------------------------------------------------------
10 # ARGS=1                # 需要一个命令行参数.
11 #
12 # if [ $# -ne "$ARGS" ]  # 如果不是一个参数的话...
13 # then
14 #   directory=`pwd`      # 当前工作目录
15 # else
16 #   directory=$1
17 # fi
18 # ------------------------------------------------------------
19
20 echo "symbolic links in directory \"$directory\""
21
22 for file in "$( find $directory -type l )"   # -type l 就是符号连接文件
23 do
24   echo "$file"
25 done | sort                          # 否则列出的文件将是未排序的
26 #  严格上说,此处并不一定非要一个循环不可,
27 #+ 因为"find"命令的结果将被扩展成一个单词.
28 #  然而,这种方式很容易理解和说明.
29
30 #  Dominik 'Aeneas' Schnitzer 指出,
31 #+ 如果没将 $( find $directory -type l )用""引用起来的话
32 #+ 那么将会把一个带有空白部分的文件名拆成以空白分隔的两部分(文件名中允许有空白).
33 #  即使这只将取出每个参数的第一个域.
34
35 exit 0
36
```

```
37
38 # Jean Helou 建议使用下边的方法:
39
40 echo "symbolic links in directory \"$directory\""
41 # 当前IFS的备份.要小心使用这个值.
42 OLDIFS=$IFS
43 IFS=:
44
45 for file in $(find $directory -type l -printf "%p$IFS")
46 do    #                          ^^^^^^^^^^^^^^^^^
47       echo "$file"
48       done|sort
```

################################End Script###

循环的输出可以重定向到文件中,我们对上边的例子做了一点修改.

Example 10-11 将目录中的符号连接文件名保存到一个文件中

################################Start Script###

```
 1 #!/bin/bash
 2 # symlinks.sh: 列出目录中所有的符号连接文件.
 3
 4 OUTFILE=symlinks.list                    # 保存的文件
 5
 6 directory=${1-`pwd`}
 7 #  如果没有其他的特殊指定,
 8 #+ 默认为当前工作目录.
 9
10
11 echo "symbolic links in directory \"$directory\"" > "$OUTFILE"
12 echo "----------------------------" >> "$OUTFILE"
13
14 for file in "$( find $directory -type l )"    # -type l 为符号链接
15 do
16   echo "$file"
17 done | sort >> "$OUTFILE"                  # 循环的输出
18 #          ^^^^^^^^^^^^^                  重定向到一个文件中
19
20 exit 0
```

################################End Script###

有一种非常像C语言的for循环的语法形式.这需要使用(()).

Example 10-12 一个C风格的for循环

################################Start Script###

```
 1 #!/bin/bash
 2 # 两种循环到10的方法.
 3
 4 echo
 5
 6 # 标准语法.
 7 for a in 1 2 3 4 5 6 7 8 9 10
 8 do
```

```
 9   echo -n "$a "
10 done
11
12 echo; echo
13
14 # +==========================================+
15
16 # 现在, 让我们用C风格的语法做同样的事.
17
18 LIMIT=10
19
20 for ((a=1; a <= LIMIT ; a++))  # Double parentheses, and "LIMIT" with no "$".
20 for ((a=1; a <= LIMIT ; a++))  # 双圆括号, 并且"LIMIT"变量前边没有 "$".
21 do
22   echo -n "$a "
23 done                          # 这是一个借用'ksh93'的结构.
24
25 echo; echo
26
27 # +========================================================================+
28
29 # 让我们使用C的逗号操作符,来同时增加两个变量的值.
30
31 for ((a=1, b=1; a <= LIMIT ; a++, b++))  # 逗号将同时进行2条操作.
32 do
33   echo -n "$a-$b "
34 done
35
36 echo; echo
37
38 exit 0
```

################################End Script###

参考Example 26-15,Example 26-16,和Example A-6.

现在来一个现实生活中使用的for循环.

Example 10-13 在batch mode中使用efax

################################Start Script###

```
 1 #!/bin/bash
 2 # Faxing ('fax' 必须已经被安装过了).
 3
 4 EXPECTED_ARGS=2
 5 E_BADARGS=65
 6
 7 if [ $# -ne $EXPECTED_ARGS ]
 8 # 检查命令行参数的个数是否正确.
 9 then
10    echo "Usage: `basename $0` phone# text-file"
11    exit $E_BADARGS
```

```
12 fi
13
14
15 if [ ! -f "$2" ]
16 then
17   echo "File $2 is not a text file"
18   exit $E_BADARGS
19 fi
20
21
22 fax make $2            # 从文本文件中创建传真格式的文件.
23
24 for file in $(ls $2.0*)  # 连接转换过的文件.
25                         # 在变量列表中使用通配符.
26 do
27   fil="$fil $file"
28 done
29
30 efax -d /dev/ttyS3 -o1 -t "T$1" $fil   # 干活的地方.
31
32
33 # S.C. 指出, 通过下边的命令可以省去for循环.
34 #    efax -d /dev/ttyS3 -o1 -t "T$1" $2.0*
35 # 但这并不十分有讲解意义[嘿嘿].
36
37 exit 0
```

################################End Script###

while

这种结构在循环的开头判断条件是否满足,如果条件一直满足,那就一直循环下去(0为退出码).与for循环的区别是,这种结构适合用在循环次数未知的情况下.

```
while [condition]
do
 command...
done
```

和for循环一样,如果想把do和条件放到同一行上还是需要一个";".

```
while [condition]; do
```

注意一下某种特定的while循环,比如getopts结构,好像和这里所介绍的模版有点脱节.

Example 10-14 简单的while循环

################################Start Script###

```
1 #!/bin/bash
2
3 var0=0
4 LIMIT=10
5
6 while [ "$var0" -lt "$LIMIT" ]
7 do
8   echo -n "$var0 "       # -n 将会阻止产生新行.
9   #           ^          空格,数字之间的分隔.
```

```
10
11  var0=`expr $var0 + 1`   # var0=$(($var0+1))  也可以.
12                          # var0=$((var0 + 1)) 也可以.
13                          # let "var0 += 1"    也可以.
14 done                     # 使用其他的方法也行.
15
16 echo
17
18 exit 0
```

################################End Script###

Example 10-15 另一个while循环

################################Start Script###

```
 1 #!/bin/bash
 2
 3 echo
 4                          # 等价于:
 5 while [ "$var1" != "end" ]     # while test "$var1" != "end"
 6 do
 7   echo "Input variable #1 (end to exit) "
 8   read var1                  # 为什么不使用'read $var1'?
 9   echo "variable #1 = $var1"   # 因为包含"#"字符,所以需要""
10   # 如果输入为'end',那么就在这里echo.
11   # 不在这里判断结束,在循环顶判断.
12   echo
13 done
14
15 exit 0
```

################################End Script###

一个while循环可以有多个判断条件,但是只有最后一个才能决定是否退出循环.然而这需要一种有点不同的循环语法.

Example 10-16 多条件的while循环

################################Start Script###

```
 1 #!/bin/bash
 2
 3 var1=unset
 4 previous=$var1
 5
 6 while echo "previous-variable = $previous"
 7       echo
 8       previous=$var1
 9       [ "$var1" != end ] # 记录之前的$var1.
10       # 这个"while"循环中有4个条件, 但是只有最后一个能控制循环.
11       # 退出状态由第4个条件决定.
12 do
13 echo "Input variable #1 (end to exit) "
14   read var1
15   echo "variable #1 = $var1"
16 done
```

```
17
18 # 尝试理解这个脚本的运行过程.
19 # 这里还是有点小技巧的.
20
21 exit 0
```

################################End Script###

与for循环一样,while循环也可通过(())来使用C风格语法.(见Example 9-30)

Example 10-17 C风格的while循环

################################Start Script###

```
 1 #!/bin/bash
 2 # wh-loopc.sh: 循环10次的while循环.
 3
 4 LIMIT=10
 5 a=1
 6
 7 while [ "$a" -le $LIMIT ]
 8 do
 9   echo -n "$a "
10   let "a+=1"
11 done          # 到目前为止都没什么令人惊奇的地方.
12
13 echo; echo
14
15 # +==========================================================================+
16
17 # 现在, 重复C风格的语法.
18
19 ((a = 1))      # a=1
20 # 双圆括号允许赋值两边的空格,就像C语言一样.
21
22 while (( a <= LIMIT ))   # 双圆括号, 变量前边没有"$".
23 do
24   echo -n "$a "
25   ((a += 1))   # let "a+=1"
26   # Yes, 看到了吧.
27   # 双圆括号允许像C风格的语法一样增加变量的值.
28 done
29
30 echo
31
32 # 现在,C程序员可以在Bash中找到回家的感觉了吧.
33
34 exit 0
```

################################End Script###

注意:while循环的stdin可以用<来重定向到文件.

whild循环的stdin支持管道.

until

这个结构在循环的顶部判断条件,并且如果条件一直为false那就一直循环下去.(与while

相反)
until [condition-is-true]
do
 command...
done
注意: until循环的判断在循环的顶部,这与某些编程语言是不同的.
与for循环一样,如果想把do和条件放在一行里,就使用";".
until [condition-is-true]; do
Example 10-18 until循环
################################Start Script###
```
1 #!/bin/bash
2
3 END_CONDITION=end
4
5 until [ "$var1" = "$END_CONDITION" ]
6 # 在循环的顶部判断条件.
7 do
8   echo "Input variable #1 "
 9   echo "($END_CONDITION to exit)"
10   read var1
11   echo "variable #1 = $var1"
12   echo
13 done
14
15 exit 0
```
################################End Script###

10.2 嵌套循环

嵌套循环就是在一个循环中还有一个循环,内部循环在外部循环体中.在外部循环的每次执行过程中都会触发内部循环,直到内部循环执行结束.外部循环执行了多少次,内部循环就完成多少次.当然,不论是外部循环或内部循环的break语句都会打断处理过程.
Example 10-19 嵌套循环
################################Start Script###
```
 1 #!/bin/bash
 2 # nested-loop.sh: 嵌套的"for" 循环.
 3
 4 outer=1            # 设置外部循环计数.
 5
 6 # 开始外部循环.
 7 for a in 1 2 3 4 5
 8 do
 9   echo "Pass $outer in outer loop."
10   echo "---------------------"
11   inner=1          # 重设内部循环的计数.
12
13   # ===============================================
14   # 开始内部循环.
```

```
15  for b in 1 2 3 4 5
16  do
17    echo "Pass $inner in inner loop."
18    let "inner+=1"  # 增加内部循环计数.
19  done
20  # 内部循环结束.
21  # ============================================
22
23  let "outer+=1"    # 增加外部循环的计数.
24  echo              # 每次外部循环之间的间隔.
25 done
26 # 外部循环结束.
27
28 exit 0
```

################################End Script###

10.3 循环控制

影响循环行为的命令

break,continue

break和continue这两个循环控制命令[1]与其它语言的类似命令的行为是相同的.break 命令将会跳出循环,continue命令将会跳过本次循环下边的语句,直接进入下次循环.

Example 10-20 break和continue命令在循环中的效果

################################Start Script###

```
 1 #!/bin/bash
 2
 3 LIMIT=19  # 上限
 4
 5 echo
 6 echo "Printing Numbers 1 through 20 (but not 3 and 11)."
 7
 8 a=0
 9
10 while [ $a -le "$LIMIT" ]
11 do
12  a=$(($a+1))
13
14  if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]  # 除了3和11.
15  then
16    continue      # 跳过本次循环剩下的语句.
17  fi
18
19  echo -n "$a "   # 在$a等于3和11的时候,这句将不会执行.
20 done
21
22 # 练习:
23 # 为什么循环会打印出20?
24
```

```
25 echo; echo
26
27 echo Printing Numbers 1 through 20, but something happens after 2.
28
29 ##############################################################
30
31 # Same loop, but substituting 'break' for 'continue'.
31 # 同样的循环, 但是用'break'来代替'continue'.
32
33 a=0
34
35 while [ "$a" -le "$LIMIT" ]
36 do
37  a=$(($a+1))
38
39  if [ "$a" -gt 2 ]
40  then
41    break  # 将会跳出整个循环.
42  fi
43
44  echo -n "$a "
45 done
46
47 echo; echo; echo
48
49 exit 0
```

################################End Script###

break命令可以带一个参数.一个不带参数的break循环只能退出最内层的循环,而break N可以退出N层循环.

Example 10-21 多层循环的退出

################################Start Script###

```
 1 #!/bin/bash
 2 # break-levels.sh: 退出循环.
 3
 4 # "break N" 退出N层循环.
 5
 6 for outerloop in 1 2 3 4 5
 7 do
 8   echo -n "Group $outerloop:   "
 9
10   # ----------------------------------------------------
11   for innerloop in 1 2 3 4 5
12   do
13     echo -n "$innerloop "
14
15     if [ "$innerloop" -eq 3 ]
16     then
17       break  # 试试 break 2 来看看发生什么.
```

```
18            # (内部循环和外部循环都被退出了.)
19     fi
20   done
21   # ----------------------------------------------------------
22
23   echo
24 done
25
26 echo
27
28 exit 0
```

################################End Script###

continue命令也可以带一个参数.一个不带参数的continue命令只去掉本次循环的剩余代码.而continue N将会把N层循环剩余的代码都去掉,但是循环的次数不变.

Example 10-22 多层循环的continue

################################Start Script###

```
 1 #!/bin/bash
 2 # "continue N" 命令, 将让N层的循环全部被continue.
 3
 4 for outer in I II III IV V          # 外部循环
 5 do
 6   echo; echo -n "Group $outer: "
 7
 8   # ----------------------------------------------------------------
 9   for inner in 1 2 3 4 5 6 7 8 9 10  # 内部循环
10   do
11
12     if [ "$inner" -eq 7 ]
13     then
14       continue 2  # continue 2层, 也就是到outer循环上.
15                   # 将"continue 2"替换为一个单独的"continue"
16                   # 来看一下一个正常循环的行为.
17     fi
18
19     echo -n "$inner "  # 7 8 9 10 将不会被echo
20   done
21   # ----------------------------------------------------------------
22 #译者注:如果在此处添加echo的话,当然也不会输出.
23 done
24
25 echo; echo
26
27 # 练习:
28 # 准备一个有意义的"continue N"的使用,放在脚本中.
29
30 exit 0
```

################################End Script###

Example 10-23 在实际的任务中使用"continue N"

SHELL十三问

################################Start Script###

```
 1 # Albert Reiner 给出了一个关于使用"continue N"的例子:
 2 # ------------------------------------------------
 3
 4 #  Suppose I have a large number of jobs that need to be run, with
 5 #+ any data that is to be treated in files of a given name pattern in a
 6 #+ directory. There are several machines that access this directory, and
 7 #+ I want to distribute the work over these different boxen. Then I
 8 #+ usually nohup something like the following on every box:
 9
10 while true
11 do
12   for n in .iso.*
13   do
14     [ "$n" = ".iso.opts" ] && continue
15     beta=${n#.iso.}
16     [ -r .Iso.$beta ] && continue
17     [ -r .lock.$beta ] && sleep 10 && continue
18     lockfile -r0 .lock.$beta || continue
19     echo -n "$beta: " `date`
20     run-isotherm $beta
21     date
22     ls -alF .Iso.$beta
23     [ -r .Iso.$beta ] && rm -f .lock.$beta
24     continue 2
25   done
26   break
27 done
28
29 #  The details, in particular the sleep N, are particular to my
30 #+ application, but the general pattern is:
31
32 while true
33 do
34   for job in {pattern}
35   do
36     {job already done or running} && continue
37     {mark job as running, do job, mark job as done}
38     continue 2
39   done
40   break       # Or something like `sleep 600' to avoid termination.
41 done
42
43 #  This way the script will stop only when there are no more jobs to do
44 #+ (including jobs that were added during runtime). Through the use
45 #+ of appropriate lockfiles it can be run on several machines
46 #+ concurrently without duplication of calculations [which run a couple
47 #+ of hours in my case, so I really want to avoid this]. Also, as search
```

```
48 #+ always starts again from the beginning, one can encode priorities in
49 #+ the file names. Of course, one could also do this without `continue 2',
50 #+ but then one would have to actually check whether or not some job
51 #+ was done (so that we should immediately look for the next job) or not
52 #+ (in which case we terminate or sleep for a long time before checking
53 #+ for a new job).
```

##################################End Script###

注意:continue N结构如果被用在一个有意义的上下文中的话,往往都很难理解,并且技巧性很高.所以最好的方法就是尽量避免它.

注意事项:

[1] 这两个命令是shell的内建命令,而不像其它的循环命令那样,比如while和case,这两个是关键字.

10.4 测试与分支(case和select结构)

---

case和select结构在技术上说不是循环,因为它们并不对可执行的代码块进行迭代.但是和循环相似的是,它们也依靠在代码块的顶部或底部的条件判断来决定程序的分支.

在代码块中控制程序分支

case (in) / esac

在shell中的case同C/C++中的switch结构是相同的.它允许通过判断来选择代码块中多条路径中的一条.

```
case "$variable" in
"$condition1")
command...
;;
"$condition1")
command...
;;
esac
```

注意: 对变量使用""并不是强制的,因为不会发生单词分离.

每句测试行,都以右小括号)结尾.

每个条件块都以两个分号结尾;;.

case块的结束以esac(case的反向拼写)结尾.

Example 10-24 使用case

##################################Start Script###

```
1 #!/bin/bash
2 # 测试字符串范围
3
4 echo; echo "Hit a key, then hit return."
5 read Keypress
6
7 case "$Keypress" in
8   [[:lower:]]   ) echo "Lowercase letter";;
9   [[:upper:]]   ) echo "Uppercase letter";;
10  [0-9]         ) echo "Digit";;
11  *             ) echo "Punctuation, whitespace, or other";;
12 esac      #  Allows ranges of characters in [square brackets],
12 esac      #  允许字符串的范围出现在[]中,
```

```
13          #+ or POSIX ranges in [[double square brackets.
13          #+ 或者POSIX范围在[[中.
14
15 #  在这个例子的第一个版本中,
16 #+ 测试大写和小写字符串使用的是
17 #+ [a-z] 和 [A-Z].
18 #  这种用法将不会在某些特定的场合或Linux发行版中正常工作.
19 #  POSIX 更具可移植性.
20 #  感谢Frank Wang 指出这点.
21
22 #  练习:
23 #  -----
24 #  就像这个脚本所表现的,它只允许单个按键,然后就结束了.
25 #  修改这个脚本,让它能够接受重复输入,
26 #+ 报告每个按键,并且只有在"X"被键入时才结束.
27 #  暗示: 将这些代码都用"while"循环圈起来.
28
29 exit 0
```

################################End Script###

Example 10-25 使用case来创建菜单

################################Start Script###

```
 1 #!/bin/bash
 2
 3 # 未经处理的地址资料
 4
 5 clear # 清屏.
 6
 7 echo "          Contact List"
 8 echo "          ------- ----"
 9 echo "Choose one of the following persons:"
10 echo
11 echo "[E]vans, Roland"
12 echo "[J]ones, Mildred"
13 echo "[S]mith, Julie"
14 echo "[Z]ane, Morris"
15 echo
16
17 read person
18
19 case "$person" in
20 # 注意,变量是被引用的.
21
22   "E" | "e" )
23   # 接受大写或小写输入.
24   echo
25   echo "Roland Evans"
26   echo "4321 Floppy Dr."
27   echo "Hardscrabble, CO 80753"
```

```
28   echo "(303) 734-9874"
29   echo "(303) 734-9892 fax"
30   echo "revans@zzy.net"
31   echo "Business partner & old friend"
32   ;;
33 # 注意,在每个选项后边都需要以;;结尾.
34
35   "J" | "j" )
36   echo
37   echo "Mildred Jones"
38   echo "249 E. 7th St., Apt. 19"
39   echo "New York, NY 10009"
40   echo "(212) 533-2814"
41   echo "(212) 533-9972 fax"
42   echo "milliej@loisaida.com"
43   echo "Ex-girlfriend"
44   echo "Birthday: Feb. 11"
45   ;;
46
47 # 后边的Smith和Zane的信息在这里就省略了.
48
49         * )
50    # 默认选项.
51    # 空输入(敲RETURN).
52    echo
53    echo "Not yet in database."
54   ;;
55
56 esac
57
58 echo
59
60 #  练习:
61 #  -----
62 #  修改这个脚本,让它能够接受多输入,
63 #+ 并且能够显示多个地址.
64
65 exit 0
##################################End Script#############################################
 一个case的特殊用法,用来测试命令行参数.
##################################Start Script###########################################
 1 #! /bin/bash
 2
 3 case "$1" in
 4 "") echo "Usage: ${0##*/} <filename>"; exit $E_PARAM;;  # 没有命令行参数,
 5                                        # 或者第一个参数为空.
 6 # Note that ${0##*/} is ${var##pattern} param substitution. Net result is $0.
 6 # 注意:${0##*/} 是${var##pattern} 这种模式的替换. 得到的结果是$0.
```

```
 7
 8 -*) FILENAME=./$1;;   #  如果传递进来的文件名参数($1)以一个破折号开头,
 9                       #+ 那么用./$1来代替
10                       #+ 这样后边的命令将不会把它作为一个选项来解释.
11
12 * ) FILENAME=$1;;     # 否则, $1.
13 esac
```

################################End Script###

这是一个更容易懂的命令行参数处理的一个例子.

################################Start Script###

```
 1 #! /bin/bash
 2
 3
 4 while [ $# -gt 0 ]; do    # 直到你用完所有的参数...
 5   case "$1" in
 6     -d|--debug)
 7               # "-d" or "--debug" parameter?
 8               DEBUG=1
 9               ;;
10     -c|--conf)
11               CONFFILE="$2"
12               shift
13               if [ ! -f $CONFFILE ]; then
14                 echo "Error: Supplied file doesn't exist!"
15                 exit $E_CONFFILE     # 文件没发现错误.
16               fi
17               ;;
18   esac
19   shift       # 检查剩下的参数.
20 done
21
22 #  来自Stefano Falsetto的 "Log2Rot" 脚本,
23 #+ 他的"rottlog" 包的一部分.
24 #  授权使用.
```

################################End Script###

Example 10-26 使用命令替换来产生case变量

################################Start Script###

```
 1 #!/bin/bash
 2 # case-cmd.sh: 使用命令替换来产生"case"变量
 3
 4 case $( arch ) in   # "arch" 返回机器的类型.
 5                    # 等价于 'uname -m' ...
 6 i386 ) echo "80386-based machine";;
 7 i486 ) echo "80486-based machine";;
 8 i586 ) echo "Pentium-based machine";;
 9 i686 ) echo "Pentium2+-based machine";;
10 *    ) echo "Other type of machine";;
11 esac
```

```
12
13 exit 0
```

################################End Script###

case结构也可以过滤globbing模式的字符串.

Example 10-27 简单字符串匹配

################################Start Script###

```
 1 #!/bin/bash
 2 # match-string.sh: 简单字符串匹配
 3
 4 match_string ()
 5 {
 6   MATCH=0
 7   NOMATCH=90
 8   PARAMS=2     # 函数需要2个参数.
 9   BAD_PARAMS=91
10
11   [ $# -eq $PARAMS ] || return $BAD_PARAMS
12
13   case "$1" in
14   "$2") return $MATCH;;
15   *   ) return $NOMATCH;;
16   esac
17
18 }
19
20
21 a=one
22 b=two
23 c=three
24 d=two
25
26
27 match_string $a     # 参数个数错误.
28 echo $?            # 91
29
30 match_string $a $b  # 不匹配
31 echo $?            # 90
32
33 match_string $b $d  # 匹配
34 echo $?            # 0
35
36
37 exit 0
```

################################End Script###

Example 10-28 检查是否是字母输入

################################Start Script###

```
 1 #!/bin/bash
 2 # isalpha.sh: 使用"case"结构来过滤字符串.
```

```
3
4 SUCCESS=0
5 FAILURE=-1
6
7 isalpha ()  # 检查输入的*第一个字符*是不是字母表上的字符.
8 {
9 if [ -z "$1" ]              # 没有参数传进来?
10 then
11   return $FAILURE
12 fi
13
14 case "$1" in
15 [a-zA-Z]*) return $SUCCESS;;  # 以一个字母开头?
16 *        ) return $FAILURE;;
17 esac
18 }            # 同C语言的"isalpha()"函数相比较.
19
20
21 isalpha2 ()   # 测试是否*整个字符串*为字母表字符.
22 {
23   [ $# -eq 1 ] || return $FAILURE
24
25   case $1 in
26   *[!a-zA-Z]*|"") return $FAILURE;;
27               *) return $SUCCESS;;
28   esac
29 }
30
31 isdigit ()    # 测试是否*整个字符串*都是数字.
32 {             # 换句话说就是测试是否是整数变量.
33   [ $# -eq 1 ] || return $FAILURE
34
35   case $1 in
36   *[!0-9]*|"") return $FAILURE;;
37             *) return $SUCCESS;;
38   esac
39 }
40
41
42
43 check_var ()  # 测试 isalpha ().
44 {
45 if isalpha "$@"
46 then
47   echo "\"$*\" begins with an alpha character."
48   if isalpha2 "$@"
49   then       # 不需要测试第一个字符是否是non-alpha.
50     echo "\"$*\" contains only alpha characters."
```

```
51   else
52     echo "\"$*\" contains at least one non-alpha character."
53   fi
54 else
55   echo "\"$*\" begins with a non-alpha character."
56              # 如果没有参数传递进来,也是"non-alpha".
57 fi
58
59 echo
60
61 }
62
63 digit_check ()  # 测试 isdigit ().
64 {
65 if isdigit "$@"
66 then
67   echo "\"$*\" contains only digits [0 - 9]."
68 else
69   echo "\"$*\" has at least one non-digit character."
70 fi
71
72 echo
73
74 }
75
76 a=23skidoo
77 b=H3llo
78 c=-What?
79 d=What?
80 e=`echo $b`   # 命令替换.
81 f=AbcDef
82 g=27234
83 h=27a34
84 i=27.34
85
86 check_var $a
87 check_var $b
88 check_var $c
89 check_var $d
90 check_var $e
91 check_var $f
92 check_var     # 没有参数传进来,将发生什么?
93 #
94 digit_check $g
95 digit_check $h
96 digit_check $i
97
98
```

```
 99 exit 0       # S.C改进过这个脚本.
100
101 # Exercise:
102 # --------
103 #  编写一个 'isfloat ()'函数来测试浮点数.
104 #  暗示: 这个函数基本上与'isdigit ()'一样,
105 #+ 但是要添加一部分小数点的处理.
```

##################################End Script##

select

select结构是建立菜单的另一种工具,这种结构是从ksh中引入的.

```
select variable [in list]
do
command...
break
done
```

提示用户选择的内容比如放在变量列表中.注意:select命令使用PS3提示符[默认为(#?)]

但是可以修改PS3.

Example 10-29 用select来创建菜单

##################################Start Script##

```
 1 #!/bin/bash
 2
 3 PS3='Choose your favorite vegetable: ' # 设置提示符字串.
 4
 5 echo
 6
 7 select vegetable in "beans" "carrots" "potatoes" "onions" "rutabagas"
 8 do
 9   echo
10   echo "Your favorite veggie is $vegetable."
11   echo "Yuck!"
12   echo
13   break  # 如果这里没有'break'会发生什么?
14 done
15
16 exit 0
```

##################################End Script##

如果忽略了in list列表,那么select命令将使用传递到脚本的命令行参数,或者是函数参数

前提是将select写到这个函数中.

与for variable [in list]结构在忽略[in list]时的行为相比较.

Example 10-30 用函数中select结构来创建菜单

##################################Start Script##

```
 1 #!/bin/bash
 2
 3 PS3='Choose your favorite vegetable: '
 4
 5 echo
 6
 7 choice_of()
```

```
 8 {
 9 select vegetable
10 # [in list] 被忽略, 所以'select'用传递给函数的参数.
11 do
12   echo
13   echo "Your favorite veggie is $vegetable."
14   echo "Yuck!"
15   echo
16   break
17 done
18 }
19
20 choice_of beans rice carrots radishes tomatoes spinach
21 #        $1    $2   $3      $4       $5       $6
22 #        传递给choice_of() 函数的参数
23
24 exit 0
```

################################End Script###

参见Example 34-3.

第11章 内部命令与内建

==========

内建命令指的就是包含在Bash工具集中的命令.这主要是考虑到执行效率的问题--内建命令将比外部命令的执行得更快,外部命令通常需要fork出一个单独的进程来执行.另外一部分原因是特定的内建命令需要直接存取shell内核部分.

当一个命令或者是shell本身需要初始化(或者创建)一个新的子进程来执行一个任务的时候,这种行为被称为forking.这个新产生的进程被叫做子进程,并且这个进程是从父进程中分离出来的.当子进程执行它的任务时,同时父进程也在运行.

注意:当父进程取得子进程的进程ID的时候,父进程可以传递给子进程参数,而反过来则不行.这将产生不可思议的并且很难追踪的问题.

Example 11-1 一个fork出多个自己实例的脚本

################################Start Script###

```
 1 #!/bin/bash
 2 # spawn.sh
 3
 4
 5 PIDS=$(pidof sh $0)  # 这个脚本不同实例的进程ID.
 6 P_array=( $PIDS )    # 把它们放到数组里(为什么?).
 7 echo $PIDS           # 显示父进程和子进程的进程ID.
 8 let "instances = ${#P_array[*]} - 1"  # 计算元素个数,至少为1.
 9                                       # 为什么减1?
10 echo "$instances instance(s) of this script running."
11 echo "[Hit Ctl-C to exit.]"; echo
12
13
14 sleep 1             # 等.
15 sh $0               # 再来一次.
16
17 exit 0              # 没必要: 脚本永远不会走到这里.
```

```
18                # 为什么走不到这里?
19
20 #  在使用Ctl-C退出之后,
21 #+ 是否所有产生的进程都会被kill掉?
22 #  如果是这样的话, 为什么?
23
24 # 注意:
25 # ----
26 # 小心,不要让这个脚本运行太长时间.
27 # 它最后将吃掉你大部分的系统资源.
28
29 #  对于用脚本产生大量的自身实例来说,
30 #+ 是否有适当的脚本技术.
31 #  为什么是为什么不是?
```

################################End Script###

一般的,脚本中的内建命令在执行时将不会fork出一个子进程.但是脚本中的外部或过滤命令通常会fork一个子进程.

一个内建命令通常与一个系统命令同名,但是Bash在内部重新实现了这些命令.比如,Bash的echo命令与/bin/echo就不尽相同,虽然它们的行为绝大多数情况下是一样的.

```
1 #!/bin/bash
2
3 echo "This line uses the \"echo\" builtin."
4 /bin/echo "This line uses the /bin/echo system command."
```

关键字的意思就是保留字.对于shell来说关键字有特殊的含义,并且用来构建shell的语法结构.比如,"for","while","do"和"!"都是关键字.与内建命令相同的是,关键字也是Bash的骨干部分,但是与内建命令不同的是,关键字自身并不是命令,而是一个比较大的命令结构的一部分.[1]

I/O类

echo

打印(到stdout)一个表达式或变量(见Example 4-1).

```
1 echo Hello
2 echo $a
```

echo需要使用-e参数来打印转移字符.见Example 5-2.

一般的每个echo命令都会在终端上新起一行,但是-n选项将会阻止新起一行.

注意:echo命令可以用来作为一系列命令的管道输入.

```
1 if echo "$VAR" | grep -q txt   # if [[ $VAR = *txt* ]]
2 then
3   echo "$VAR contains the substring sequence \"txt\""
4 fi
```

注意:echo命令与命令替换相组合可以用来设置一个变量.

```
a=`echo "HELLO" | tr A-Z a-z`
```

参见Example 12-19,Example 12-3,Example 12-42,和Example 12-43.

注意:echo `command`将会删除任何有命令产生的换行符.

$IFS(内部域分隔符)一般都会将\n(换行符)包含在它的空白字符集合中.Bash因此会根据参数中的换行来分离命令的输出.然后echo将以空格代替换行来输出这些参数.

```
bash$ ls -l /usr/share/apps/kjezz/sounds
-rw-r--r--    1 root     root         1407 Nov  7  2000 reflect.au
-rw-r--r--    1 root     root          362 Nov  7  2000 seconds.au
```

bash$ echo `ls -l /usr/share/apps/kjezz/sounds`
total 40 -rw-r--r-- 1 root root 716 Nov 7 2000 reflect.au -rw-r--r-- 1 root root 362 Nov 7 2000 seconds.au

所以,我们怎么才能在一个需要echo出来的字符串中嵌入换行呢?

##################################Start Script###

```
1 # 嵌入一个换行?
2 echo "Why doesn't this string \n split on two lines?"
3 # 上边这句的\n将被打印出来.达不到换行的目的.
4
5 # 让我们在试试其他方法.
6
7 echo
8
9 echo $"A line of text containing
10 a linefeed."
11 # 打印出2个独立的行,(潜入换行成功了).
12 # 但是,"$"前缀是否是必要的?
13
14 echo
15
16 echo "This string splits
17 on two lines."
18 # 不用非得有"$"前缀.
19
20 echo
21 echo "---------------"
22 echo
23
24 echo -n $"Another line of text containing
25 a linefeed."
26 # 打印出2个独立的行,(潜入换行成功了).
27 # 即使-n选项,也没能阻止换行(译者:-n 阻止了第2个换行)
28
29 echo
30 echo
31 echo "---------------"
32 echo
33 echo
34
35 # 然而,下边的代码就没能像期望的那样运行.
36 # Why not? Hint: Assignment to a variable.
36 # 为什么失败? 提示: 因为分配到了变量.
37 string1=$"Yet another line of text containing
38 a linefeed (maybe)."
39
40 echo $string1
41 # Yet another line of text containing a linefeed (maybe).
42 #                              ^
```

```
43 # 换行变成了空格.
44
45 # Thanks, Steve Parker, for pointing this out.
```

################################End Script###

注意: 这个命令是shell的一个内建命令,与/bin/echo不同,虽然行为相似.

```
bash$ type -a echo
echo is a shell builtin
echo is /bin/echo
```

printf

printf命令,格式化输出,是echo命令的增强.它是C语言printf()库函数的一个有限的变形,
并且在语法上有些不同.

```
printf format-string... parameter...
```

这是Bash的内建版本,与/bin/printf或/usr/bin/printf命令不同.想更深入的了解,请
察看printf(系统命令)的man页.

注意:老版本的Bash可能不支持printf.

Example 11-2 printf

################################Start Script###

```
 1 #!/bin/bash
 2 # printf demo
 3
 4 PI=3.14159265358979
 5 DecimalConstant=31373
 6 Message1="Greetings,"
 7 Message2="Earthling."
 8
 9 echo
10
11 printf "Pi to 2 decimal places = %1.2f" $PI
12 echo
13 printf "Pi to 9 decimal places = %1.9f" $PI  # 都能正确地结束.
14
15 printf "\n"                          # 打印一个换行,
16                                     # 等价于 'echo' . . .
17
18 printf "Constant = \t%d\n" $DecimalConstant  # 插入一个 tab (\t).
19
20 printf "%s %s \n" $Message1 $Message2
21
22 echo
23
24 # ============================================#
25 # 模仿C函数, sprintf().
26 # 使用一个格式化的字符串来加载一个变量.
27
28 echo
29
30 Pi12=$(printf "%1.12f" $PI)
31 echo "Pi to 12 decimal places = $Pi12"
```

```
32
33 Msg=`printf "%s %s \n" $Message1 $Message2`
34 echo $Msg; echo $Msg
35
36 #  向我们看到的一样,现在'sprintf'函数可以
37 #+ 作为一个可被加载的模块
38 #+ 但这是不可移植的.
39
40 exit 0
```

################################End Script###

使用printf的最主要的应用就是格式化错误消息.

```
 1 E_BADDIR=65
 2
 3 var=nonexistent_directory
 4
 5 error()
 6 {
 7   printf "$@" >&2
 8   # 格式化传递进来的位置参数,并把它们送到stderr.
 9   echo
10   exit $E_BADDIR
11 }
12
13 cd $var || error $"Can't cd to %s." "$var"
14
15 # Thanks, S.C.
```

read

从stdin中读取一个变量的值,也就是与键盘交互取得变量的值.使用-a参数可以取得数组变量(见Example 26-6).

Example 11-3 使用read,变量分配

################################Start Script###

```
 1 #!/bin/bash
 2 # "Reading" 变量.
 3
 4 echo -n "Enter the value of variable 'var1': "
 5 # -n选项,阻止换行.
 6
 7 read var1
 8 # 注意在var1前面没有'$',因为变量正在被设置.
 9
10 echo "var1 = $var1"
11
12
13 echo
14
15 # 一个'read'命令可以设置多个变量.
16 echo -n "Enter the values of variables 'var2' and 'var3' (separated by a space or tab): "
17 read var2 var3
```

```
18 echo "var2 = $var2      var3 = $var3"
19 # 如果你只输入了一个值,那么其他的变量还是未设置(null).
20
21 exit 0
```

################################End Script###

一个不带变量参数的read命令,将把来自键盘的输入存入到专用变量$REPLY中.

Example 11-4 当使用一个不带变量参数的read命令时,将会发生什么?

################################Start Script###

```
 1 #!/bin/bash
 2 # read-novar.sh
 3
 4 echo
 5
 6 # ------------------------- #
 7 echo -n "Enter a value: "
 8 read var
 9 echo "\"var\" = "$var""
10 # 到这里为止,都与期望的相同.
11 # ------------------------- #
12
13 echo
14
15 # ------------------------------------------------------------------ #
16 echo -n "Enter another value: "
17 read          #  没有变量分配给'read'命令,因此...
18              #+ 输入将分配给默认变量,$REPLY.
19 var="$REPLY"
20 echo "\"var\" = "$var""
21 # 这部分代码和上边的代码等价.
22 # ------------------------------------------------------------------ #
23
24 echo
25
26 exit 0
```

################################End Script###

通常情况下,在使用read命令时,输入一个\然后回车,将会阻止产生一个新行.-r选项将会让\转义.

Example 11-5 read命令的多行输入

################################Start Script###

```
 1 #!/bin/bash
 2
 3 echo
 4
 5 echo "Enter a string terminated by a \\, then press <ENTER>."
 6 echo "Then, enter a second string, and again press <ENTER>."
 7 read var1     # "\"将会阻止产生新行,当read $var1时.
 8              #     first line \
 9              #     second line
```

```
10
11 echo "var1 = $var1"
12 #     var1 = first line second line
13
14 #  For each line terminated by a "\"
14 #  对于每个一个"\"结尾的行
15 #+ 你都会看到一个下一行的提示符,让你继续向var1输入内容.
16
17 echo; echo
18
19 echo "Enter another string terminated by a \\ , then press <ENTER>."
20 read -r var2  # -r选项将会让"\"转义.
21              #     first line \
22
23 echo "var2 = $var2"
24 #     var2 = first line \
25
26 # 第一个<ENTER>就会结束var2变量的录入.
27
28 echo
29
30 exit 0
```

################################End Script###

read命令有些有趣的选项,这些选项允许打印出一个提示符,然后在不输入<ENTER>的情况下,可以读入你的按键字符.

```
 1 # Read a keypress without hitting ENTER.
 1 # 不敲回车,读取一个按键字符.
 2
 3 read -s -n1 -p "Hit a key " keypress
 4 echo; echo "Keypress was "\"$keypress\""."
 5
 6 # -s 选项意味着不打印输入.
 7 # -n N 选项意味着直接受N个字符的输入.
 8 # -p 选项意味着在读取输入之前打印出后边的提示符.
 9
10 # 使用这些选项是有技巧的,因为你需要使用正确的循序来使用它们.
```

read的-n选项也可以检测方向键,和一些控制按键.

Example 11-6 检测方向键

################################Start Script###

```
 1 #!/bin/bash
 2 # arrow-detect.sh: 检测方向键,和一些非打印字符的按键.
 3 # Thank you, Sandro Magi告诉了我怎么做.
 4
 5 # --------------------------------------------
 6 # 按键产生的字符编码.
 7 arrowup='\[A'
 8 arrowdown='\[B'
 9 arrowrt='\[C'
```

```
10 arrowleft='\[D'
11 insert='\[2'
12 delete='\[3'
13 # --------------------------------------------
14
15 SUCCESS=0
16 OTHER=65
17
18 echo -n "Press a key...  "
19 # 如果不是上边列表所列出的按键,可能还是需要按回车.(译者:因为一般按键是一个字符)
20 read -n3 key                    # 读3个字符.
21
22 echo -n "$key" | grep "$arrowup"  #检查输入字符是否匹配.
23 if [ "$?" -eq $SUCCESS ]
24 then
25   echo "Up-arrow key pressed."
26   exit $SUCCESS
27 fi
28
29 echo -n "$key" | grep "$arrowdown"
30 if [ "$?" -eq $SUCCESS ]
31 then
32   echo "Down-arrow key pressed."
33   exit $SUCCESS
34 fi
35
36 echo -n "$key" | grep "$arrowrt"
37 if [ "$?" -eq $SUCCESS ]
38 then
39   echo "Right-arrow key pressed."
40   exit $SUCCESS
41 fi
42
43 echo -n "$key" | grep "$arrowleft"
44 if [ "$?" -eq $SUCCESS ]
45 then
46   echo "Left-arrow key pressed."
47   exit $SUCCESS
48 fi
49
50 echo -n "$key" | grep "$insert"
51 if [ "$?" -eq $SUCCESS ]
52 then
53   echo "\"Insert\" key pressed."
54   exit $SUCCESS
55 fi
56
57 echo -n "$key" | grep "$delete"
```

```
58 if [ "$?" -eq $SUCCESS ]
59 then
60   echo "\"Delete\" key pressed."
61   exit $SUCCESS
62 fi
63
64
65 echo " Some other key pressed."
66
67 exit $OTHER
68
69 #  练习:
70 #  -----
71 #  1) 通过使用'case'结构来代替'if'结构
72 #+    来简化这个脚本.
73 #  2) Add detection of the "Home," "End," "PgUp," and "PgDn" keys.
73 #  2) 添加"Home," "End," "PgUp," 和 "PgDn"这些按键的检查.
```

##################################End Script##

注意: 对read命令来说,-n 选项将不会检测ENTER(新行)键.

read命令的-t选项允许时间输入(见Example 9-4).

read命令也可以从重定向的文件中读入变量的值.如果文件中的内容超过一行,那么只有第一行被分配到这个变量中.如果read命令有超过一个参数,那么每个变量都会从文件中取得以定义的空白分隔的字符串作为变量的值.小心!

Example 11-7 通过文件重定向来使用read

##################################Start Script##

```
 1 #!/bin/bash
 2
 3 read var1 <data-file
 4 echo "var1 = $var1"
 5 # var1将会把data-file的第一行的全部内容都作为它的值.
 6
 7 read var2 var3 <data-file
 8 echo "var2 = $var2   var3 = $var3"
 9 # 注意,这里"read"命令将会产生一种不直观的行为.
10 # 1) 重新从文件的开头开始读入变量.
11 # 2) 每个变量都设置成了以空白分割的字符串,
12 #    而不是之前的以整行的内容作为变量的值.
13 # 3) 而最后一个变量将会取得第一行剩余的全部部分(不管是否以空白分割).
14 # 4) 如果需要赋值的变量的个数比文件中第一行一空白分割的字符串的个数多的话,
15 #    那么这些变量将会被赋空值.
16
17 echo "-----------------------------------------------"
18
19 # 如何用循环来解决上边所提到的问题:
20 while read line
21 do
22   echo "$line"
23 done <data-file
```

```
24 # Thanks, Heiner Steven for pointing this out.
25
26 echo "------------------------------------------------"
27
28 # 使用$IFS (内部域分隔变量)来将每行的输入单独的放到"read"中,
29 # 如果你不想使用默认空白的话.
30
31 echo "List of all users:"
32 OIFS=$IFS; IFS=:       # /etc/passwd 使用 ":" 作为域分隔符.
33 while read name passwd uid gid fullname ignore
34 do
35   echo "$name ($fullname)"
36 done </etc/passwd   # I/O 重定向.
37 IFS=$OIFS             # 恢复原始的 $IFS.
38 # 这个代码片段也是Heiner Steven写的.
39
40
41
42 #  在循环内部设置$IFS变量
43 #+ 而不用把原始的$IFS
44 #+ 保存到临时变量中.
45 #  Thanks, Dim Segebart, for pointing this out.
46 echo "------------------------------------------------"
47 echo "List of all users:"
48
49 while IFS=: read name passwd uid gid fullname ignore
50 do
51   echo "$name ($fullname)"
52 done </etc/passwd   # I/O 重定向.
53
54 echo
55 echo "\$IFS still $IFS"
56
57 exit 0
```

##################################End Script###

注意:管道输出到一个read命令中,使用管道echo输出到read会失败.

然而使用管道cat输出看起来能够正常运行.

```
1 cat file1 file2 |
2 while read line
3 do
4 echo $line
5 done
```

但是,像Bjon Eriksson指出的:

Example 11-8 管道输出到read中的问题

##################################Start Script###

```
1 #!/bin/sh
2 # readpipe.sh
3 # 这个例子是Bjon Eriksson捐献的.
```

```
 4
 5 last="(null)"
 6 cat $0 |
 7 while read line
 8 do
 9     echo "{$line}"
10     last=$line
11 done
12 printf "\nAll done, last:$last\n"
13
14 exit 0  # 代码结束.
15         # 下边是这个脚本的部分输出.
16         # 打印出了多余的大括号.
17
18 ################################################
19
20 ./readpipe.sh
21
22 {#!/bin/sh}
23 {last="(null)"}
24 {cat $0 |}
25 {while read line}
26 {do}
27 {echo "{$line}"}
28 {last=$line}
29 {done}
30 {printf "nAll done, last:$lastn"}
31
32
33 All done, last:(null)
34
35 变量(last)是设置在子shell中的而没设在外边.
#################################End Script#############################################
 在许多linux发行版上,gendiff脚本通常在/usr/bin下,将find的输出使用管道传递到一个
 while循环中.
 1 find $1 \( -name "*$2" -o -name ".*$2" \) -print |
 2 while read f; do
 3 . . .
文件系统类
cd
 cd,修改目录命令,在脚本中用得最多的时候就是,命令需要在指定目录下运行时,需要用cd
 修改当前工作目录.
 1 (cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
 [之前有个例子,Alan Cox写的]
 -P(physical)选项的作用是忽略符号连接.
 cd - 将把工作目录改为$OLDPWD,就是之前的工作目录.
 注意:当我们用两个/来作为cd命令的参数时,结果却出乎我们的意料.
 bash$ cd //
```

bash$ pwd
//
输出应该,并且当然是/.无论在命令行下还是在脚本中,这都是个问题.
pwd
打印当前的工作目录.这将给用户(或脚本)当前的工作目录(见Example 11-9).使用这个
命令的结果和从内键变量$PWD中读取的值是相同的.
pushd, popd, dirs
这几个命令可以使得工作目录书签化,就是可以按顺序向前或向后移动工作目录.
压栈的动作可以保存工作目录列表.选项可以允许对目录栈作不同的操作.
pushd dir-name 把路径dir-name压入目录栈,同时修改当前目录到dir-name.
popd 将目录栈中最上边的目录弹出,同时修改当前目录到弹出来的那个目录.
dirs 列出所有目录栈的内容(与$DIRSTACK便两相比较).一个成功的pushd或者popd将会
自动的调用dirs命令.
对于那些并没有对当前工作目录做硬编码,并且需要对当前工作目录做灵活修改的脚本来说
,使用这些命令是再好不过的了.注意内建$DIRSTACK数组变量,这个变量可以在脚本内存取,
并且它们保存了目录栈的内容.
Example 11-9 修改当前的工作目录
################################Start Script###
```
1 #!/bin/bash
2
3 dir1=/usr/local
4 dir2=/var/spool
5
6 pushd $dir1
7 # 将会自动运行一个 'dirs' (把目录栈的内容列到stdout上).
8 echo "Now in directory `pwd`." # Uses back-quoted 'pwd'.
9
10 # 现在对'dir1'做一些操作.
11 pushd $dir2
12 echo "Now in directory `pwd`."
13
14 # 现在对'dir2'做一些操作.
15 echo "The top entry in the DIRSTACK array is $DIRSTACK."
16 popd
17 echo "Now back in directory `pwd`."
18
19 # 现在,对'dir1'做更多的操作.
20 popd
21 echo "Now back in original working directory `pwd`."
22
23 exit 0
24
25 # 如果你不使用 'popd'将会发生什么 -- 然后退出这个脚本?
26 # 你最后将落在那个目录中?为什么?
```
################################End Script###
变量类
let
let命令将执行变量的算术操作.在许多情况下,它被看作是复杂的expr版本的一个简化版.

Example 11-10 用"let"命令来作算术操作.

```
################################Start Script#########################################
 1 #!/bin/bash
 2
 3 echo
 4
 5 let a=11           # 与 'a=11' 相同
 6 let a=a+5          # 等价于let "a = a + 5"
 7                   # (双引号和空格是这句话更具可读性.)
 8 echo "11 + 5 = $a"  # 16
 9
10 let "a <<= 3"       # 等价于let "a = a << 3"
11 echo "\"\$a\" (=16) left-shifted 3 places = $a"
12                    # 128
13
14 let "a /= 4"        # 等价于let "a = a / 4"
15 echo "128 / 4 = $a" # 32
16
17 let "a -= 5"        # 等价于let "a = a - 5"
18 echo "32 - 5 = $a"  # 27
19
20 let "a *=  10"      # 等价于let "a = a * 10"
21 echo "27 * 10 = $a" # 270
22
23 let "a %= 8"        # 等价于let "a = a % 8"
24 echo "270 modulo 8 = $a  (270 / 8 = 33, remainder $a)"
25                    # 6
26
27 echo
28
29 exit 0
################################End Script###########################################
```

eval

```
 eval arg1 [arg2] ... [argN]
 将表达式中的参数,或者表达式列表,组合起来,并且评估它们.包含在表达式中的任何变量
 都将被扩展.结果将会被转化到命令中.这对于从命令行或者脚本中产生代码是很有用的.
 bash$ process=xterm
 bash$ show_process="eval ps ax | grep $process"
 bash$ $show_process
 1867 tty1     S      0:02 xterm
 2779 tty1     S      0:00 xterm
 2886 pts/1    S      0:00 grep xterm
```

Example 11-11 显示eval命令的效果

```
################################Start Script#########################################
 1 #!/bin/bash
 2
 3 y=`eval ls -l`  #  与 y=`ls -l` 很相似
 4 echo $y        #+ 但是换行符将被删除,因为echo的变量未被""引用.
```

```
 5 echo
 6 echo "$y"      #  用""将变量引用,换行符就不会被空格替换了.
 7
 8 echo; echo
 9
10 y=`eval df`     #  与 y=`df` 很相似
11 echo $y        #+ 换行符又被空格替换了.
12
13 #  当没有LF(换行符)出现时,对于使用"awk"这样的工具来说,
14 #+ 可能分析输出的结果更容易一些.
15
16 echo
17 echo "============================================================="
18 echo
19
20 # Now, showing how to "expand" a variable using "eval" . . .
20 # 现在,来看一下怎么用"eval"命令来扩展一个变量. . .
21
22 for i in 1 2 3 4 5; do
23   eval value=$i
24   #  value=$i 将具有同样的效果. "eval"并不非得在这里使用.
25   #  一个缺乏特殊含义的变量将被评估为自身 --
26   #+ 也就是说,这个变量除了能够被扩展成自身所表示的字符,不能扩展成任何其他的含义.
27   echo $value
28 done
29
30 echo
31 echo "---"
32 echo
33
34 for i in ls df; do
35   value=eval $i
36   #  value=$i has an entirely different effect here.
36   #  value=$i 在这里就与上边这句有了本质上的区别.
37   #  "eval" 将会评估命令 "ls" 和 "df" . . .
38   #  术语 "ls" 和 "df" 就具有特殊含义,
39   #+ 因为它们被解释成命令,
40   #+ 而不是字符串本身.
41   echo $value
42 done
43
44
45 exit 0
```

##################################End Script###

Example 11-12 强制登出(log-off)

##################################Start Script###

```
 1 #!/bin/bash
 2 # 结束ppp进程来强制登出log-off.
```

```
 3
 4 # 脚本应该以根用户的身份来运行.
 5
 6 killppp="eval kill -9 `ps ax | awk '/ppp/ { print $1 }'`"
 7 #                 -------- ppp 的进程ID       -------
 8
 9 $killppp                # 这个变量现在成为了一个命令.
10
11
12 # 下边的命令必须以根用户的身份来运行.
13
14 chmod 666 /dev/ttyS3      # 恢复读写权限,否则什么?
15 #  因为在ppp上执行一个SIGKILL将会修改串口的权限,
16 #+ 我们把权限恢复到之前的状态.
17
18 rm /var/lock/LCK..ttyS3   # 删除串口琐文件.为什么?
19
20 exit 0
21
22 # 练习:
23 # -----
24 # 1) 编写一个脚本来验证是否跟用户正在运行它.
25 # 2) 做一个检查,检查一下将要杀掉的进程
26 #+   再杀掉这个进程之前,它是否正在运行.
27 # 3) 基于'fuser'来编写达到这个目的的另一个版本的脚本
28 #+      if [ fuser -s /dev/modem ]; then . . .
```

##################################End Script###

Example 11-13 另一个"rot13"的版本

##################################Start Script###

```
 1 #!/bin/bash
 2 # 使用'eval'的一个"rot13"的版本,(译者:rot13就是把26个字母,从中间分为2瓣,各13个)
 3 # 与脚本"rot13.sh" 比较一下.
 4
 5 setvar_rot_13()             # "rot13" 函数
 6 {
 7   local varname=$1 varvalue=$2
 8   eval $varname='$(echo "$varvalue" | tr a-z n-za-m)'
 9 }
10
11
12 setvar_rot_13 var "foobar"   # 用"foobar" 传递到rot13函数中.
13 echo $var                   # 结果是sbbone
14
15 setvar_rot_13 var "$var"     # 传递"sbbone" 到rot13函数中.
16                             # 又变成了原始值.
17 echo $var                   # foobar
18
19 # 这个例子是Segebart Chazelas编写的.
```

```
20 # 作者又修改了一下.
21
22 exit 0
```

################################End Script###

Rory Winston捐献了下编的脚本,关于使用eval命令.

Example 11-14 在Perl脚本中使用eval命令来强制变量替换

################################Start Script###

```
 1 In the Perl script "test.pl":
 2        ...
 3        my $WEBROOT = <WEBROOT_PATH>;
 4        ...
 5
 6 To force variable substitution try:
 7        $export WEBROOT_PATH=/usr/local/webroot
 8        $sed 's/<WEBROOT_PATH>/$WEBROOT_PATH/' < test.pl > out
 9
10 But this just gives:
11         my $WEBROOT = $WEBROOT_PATH;
12
13 However:
14        $export WEBROOT_PATH=/usr/local/webroot
15        $eval sed 's%\<WEBROOT_PATH\>%$WEBROOT_PATH%' < test.pl > out
16 #       ====
17
18 That works fine, and gives the expected substitution:
19        my $WEBROOT = /usr/local/webroot;
20
21
22 ### Correction applied to original example by Paulo Marcel Coelho Aragao.
```

################################End Script###

eval命令是有风险的,如果有更合适的方法来实现功能的话,尽量要避免使用它.

eval命令将执行命令的内容,如果命令中有rm -rf*这种东西,可能就不是你想要的了.

如果在一个不熟悉的人编写的脚本中使用eval命令将是危险的.

set

set命令用来修改内部脚本变量的值.一个作用就是触发选项标志位来帮助决定脚本的行为.另一个应用就是以一个命令的结果(set `command`)来重新设置脚本的位置参数.脚本将会从命令的输出中重新分析出位置参数.

Example 11-15 使用set来改变脚本的位置参数

################################Start Script###

```
 1 #!/bin/bash
 2
 3 # script "set-test"
 4
 5 # 使用3个命令行参数来调用这个脚本,
 6 # 比如, "./set-test one two three".
 7
 8 echo
 9 echo "Positional parameters before  set \`uname -a\` :" #uname命令打印操作系统名
```

```
10 echo "Command-line argument #1 = $1"
11 echo "Command-line argument #2 = $2"
12 echo "Command-line argument #3 = $3"
13
14
15 set `uname -a` # 把`uname -a`的命令输出设置
16              # 为新的位置参数.
17
18 echo $_       # 这要看你的unmae -a输出了,这句打印出的就是输出的最后一个单词.
19 # 在脚本中设置标志.
20
21 echo "Positional parameters after  set \`uname -a\` :"
22 # $1, $2, $3, 等等. 这些位置参数将被重新初始化为`uname -a`的结果
23 echo "Field #1 of 'uname -a' = $1"
24 echo "Field #2 of 'uname -a' = $2"
25 echo "Field #3 of 'uname -a' = $3"
26 echo ---
27 echo $_       # ---
28 echo
29
30 exit 0
```

```
################################End Script#########################################
 不使用任何选项或参数来调用set命令的话,将会列出所有的环境变量和其他所有的已经
 初始化过的命令.
 bash$ set
 AUTHORCOPY=/home/bozo/posts
 BASH=/bin/bash
 BASH_VERSION=$'2.05.8(1)-release'
 ...
 XAUTHORITY=/home/bozo/.Xauthority
 _=/etc/bashrc
 variable22=abc
 variable23=xzy
 使用参数--来调用set命令的话,将会明确的分配位置参数.如果--选项后边没有跟变量名
 的话,那么结果就使所有位置参数都比unset了.
Example 11-16 重新分配位置参数
################################Start Script#######################################
 1 #!/bin/bash
 2
 3 variable="one two three four five"
 4
 5 set -- $variable
 6 # 将位置参数的内容设为变量"$variable"的内容.
 7
 8 first_param=$1
 9 second_param=$2
10 shift; shift       # Shift past first two positional params.
11 remaining_params="$*"
```

```
12
13 echo
14 echo "first parameter = $first_param"            # one
15 echo "second parameter = $second_param"          # two
16 echo "remaining parameters = $remaining_params"   # three four five
17
18 echo; echo
19
20 # 再来一次.
21 set -- $variable
22 first_param=$1
23 second_param=$2
24 echo "first parameter = $first_param"            # one
25 echo "second parameter = $second_param"          # two
26
27 # ==============================================
28
29 set --
30 # Unsets positional parameters if no variable specified.
30 # 如果没指定变量,那么将会unset所有的位置参数.
31
32 first_param=$1
33 second_param=$2
34 echo "first parameter = $first_param"            # (null value)
35 echo "second parameter = $second_param"          # (null value)
36
37 exit 0
```

################################End Script###

见Example 10-2,和Example 12-51.

unset

unset命令用来删除一个shell变量,效果就是把这个变量设为null.注意:这个命令对位置参数无效.

```
bash$ unset PATH
bash$ echo $PATH
bash$
```

Example 11-17 Unset一个变量

################################Start Script###

```
1 #!/bin/bash
2 # unset.sh: Unset一个变量.
3
4 variable=hello                    # 初始化.
5 echo "variable = $variable"
6
7 unset variable                   # Unset.
8                                   # 与 variable= 的效果相同.
9 echo "(unset) variable = $variable"  # $variable 设为 null.
10
11 exit 0
```

################################End Script###
export
export命令将会使得被export的变量在运行的脚本(或shell)的所有的子进程中都可用.
不幸的是,没有办法将变量export到父进程(就是调用这个脚本或shell的进程)中.
关于export命令的一个重要的使用就是用在启动文件中,启动文件是用来初始化并且
设置环境变量,让用户进程可以存取环境变量.
Example 11-18 使用export命令传递一个变量到一个内嵌awk的脚本中
################################Start Script###

```
 1 #!/bin/bash
 2
 3 #  这是"求列的和"脚本的另外一个版本(col-totaler.sh)
 4 #+ 那个脚本可以把目标文件中的指定的列上的所有数字全部累加起来,求和.
 5 #  这个版本将把一个变量通过export的形式传递到'awk'中 . . .
 6 #+ 并且把awk脚本放到一个变量中.
 7
 8
 9 ARGS=2
10 E_WRONGARGS=65
11
12 if [ $# -ne "$ARGS" ] # 检查命令行参数的个数.
13 then
14   echo "Usage: `basename $0` filename column-number"
15   exit $E_WRONGARGS
16 fi
17
18 filename=$1
19 column_number=$2
20
21 #===== 上边的这部分,与原始脚本完全一样 =====#
22
23 export column_number
24 # 将列号通过export出来,这样后边的进程就可用了.
25
26
27 # ----------------------------------------------
28 awkscript='{ total += $ENVIRON["column_number"] }
29 END { print total }'
30 # 是的,一个变量可以保存一个awk脚本.
31 # ----------------------------------------------
32
33 # 现在,运行awk脚本.
34 awk "$awkscript" "$filename"
35
36 # Thanks, Stephane Chazelas.
37
38 exit 0
```

################################End Script###
注意:可以在一个操作中同时赋值和export变量,如: export var1=xxx.

然而,像Greg Keraunen指出的,在某些情况下使用上边这种形式,将与先设置变量,然后
export变量效果不同.
bash$ export var=(a b); echo ${var[0]}
(a b)

bash$ var=(a b); export var; echo ${var[0]}
a
declare, typeset
declare和typeset命令被用来指定或限制变量的属性.
readonly
与declare -r作用相同,设置变量的只读属性,也可以认为是设置常量.设置了这种属性之后
如果你还要修改它,那么你将得到一个错误消息.这种情况与C语言中的const常量类型的情
况是相同的.
getopts
可以说这是分析传递到脚本的命令行参数的最强力工具.这个命令与getopt外部命令,和
C语言中的库函数getopt的作用是相同的.它允许传递和连接多个选项[2]到脚本中,并能分
配多个参数到脚本中.
getopts结构使用两个隐含变量.$OPTIND是参数指针(选项索引),和$OPTARG(选项参数)
(可选的)可以在选项后边附加一个参数.在声明标签中,选项名后边的冒号用来提示
这个选项名已经分配了一个参数.
getopts结构通常都组成一组放在一个while循环中,循环过程中每次处理一个选项和参数,
然后增加隐含变量$OPTIND的值,再进行下一次的处理.
注意: 1.通过命令行传递到脚本中的参数前边必须加上一个减号(-).这是一个前缀,这样
getopts命令将会认为这个参数是一个选项.事实上,getopts不会处理不带"-"前缀
的参数,如果第一个参数就没有"-",那么将结束选项的处理.
2.使用getopts的while循环模版还是与标准的while循环模版有些不同.没有标准
while循环中的[]判断条件.
3.getopts结构将会取代getopt外部命令.
##################################Start Script###

```
1 while getopts ":abcde:fg" Option
2 # Initial declaration.
2 # 开始的声明.
3 # a, b, c, d, e, f, 和 g 被认为是选项(标志).
4 # e选项后边的:提示,这个选项带一个参数.
5 do
6   case $Option in
7     a ) # Do something with variable 'a'.
7     a ) # 对选项'a'作些操作.
8     b ) # 对选项'b'作些操作.
9     ...
10     e)  # Do something with 'e', and also with $OPTARG,
10     e)  # 对选项'e'作些操作, 同时处理一下$OPTARG,
11         # which is the associated argument passed with option 'e'.
11         # 这个变量里边将保存传递给选项"e"的参数.
12     ...
13     g ) # 对选项'g'作些操作.
14   esac
15 done
```

```
16 shift $(($OPTIND - 1))
17 # 将参数指针向下移动.
18
19 # 所有这些远没有它看起来的那么复杂.<嘿嘿>
20
```

################################End Script###

Example 11-19 使用getopts命令来读取传递给脚本的选项/参数.

(我测试的结果与说明不同,我使用 ./scriptname -mnp,但是$OPTIND的值居然是1 1 2)

################################Start Script###

```
 1 #!/bin/bash
 2 # 练习 getopts 和 OPTIND
 3 # 在Bill Gradwohl的建议下,这个脚本于 10/09/03 被修改.
 4
 5
 6 # 这里我们将学习 'getopts'如何处理脚本的命令行参数.
 7 # 参数被作为"选项"(标志)被解析,并且分配参数.
 8
 9 # 试一下通过如下方法来调用这个脚本
10 # 'scriptname -mn'
11 # 'scriptname -oq qOption' (qOption 可以是任意的哪怕有些诡异字符的字符串.)
12 # 'scriptname -qXXX -r'
13 #
14 # 'scriptname -qr'    - 意外的结果, "r" 将被看成是选项 "q" 的参数.
15 # 'scriptname -q -r'  - 意外的结果, 同上.
16 # 'scriptname -mnop -mnop'  - 意外的结果
17 # (OPTIND is unreliable at stating where an option came from).
18 #
19 #  如果一个选项需要一个参数("flag:"),那么它应该
20 #+ 取得在命令行上挨在它后边的任何字符.
21
22 NO_ARGS=0
23 E_OPTERROR=65
24
25 if [ $# -eq "$NO_ARGS" ]  # 不带命令行参数就调用脚本?
26 then
27   echo "Usage: `basename $0` options (-mnopqrs)"
28   exit $E_OPTERROR        # 如果没有参数传进来,那就退出,并解释用法.
29 fi
30 # 用法: 脚本名 -选项名
31 # 注意: 破折号(-)是必须的
32
33
34 while getopts ":mnopq:rs" Option
35 do
36   case $Option in
37     m     ) echo "Scenario #1: option -m-   [OPTIND=${OPTIND}]";;
38     n | o ) echo "Scenario #2: option -$Option-   [OPTIND=${OPTIND}]";;
39     p     ) echo "Scenario #3: option -p-   [OPTIND=${OPTIND}]";;
```

```
40     q     ) echo "Scenario #4: option -q-\
41  with argument \"$OPTARG\"   [OPTIND=${OPTIND}]";;
42     #  注意,选项'q'必须分配一个参数,
43     #+ 否则默认将失败.
44     r | s ) echo "Scenario #5: option -$Option-";;
45     *     ) echo "Unimplemented option chosen.";;   # DEFAULT
46   esac
47 done
48
49 shift $(($OPTIND - 1))
50 #  将参数指针减1,这样它将指向下一个参数.
51 #  $1 现在引用的是命令行上的第一个非选项参数
52 #+ 如果有一个这样的参数存在的话.
53
54 exit 0
55
56 #   像 Bill Gradwohl 所说,<rojy bug>
57 #  "The getopts mechanism allows one to specify:  scriptname -mnop -mnop
58 #+  but there is no reliable way to differentiate what came from where
59 #+  by using OPTIND."
```

##################################End Script###

脚本行为

source, . (点命令)

这个命令在命令行上执行的时候,将会执行一个脚本.在一个文件内一个source file-name将会加载file-name文件.source一个文件(或点命令)将会在脚本中引入代码,并附加到脚本中(与C语言中的#include指令的效果相同).最终的结果就像是在使用"sourced"行上插入了相应文件的内容.这在多个脚本需要引用相同的数据,或函数库时非常有用.

Example 11-20 "Including"一个数据文件

##################################Start Script###

```
 1 #!/bin/bash
 2
 3 . data-file    # 加载一个数据文件.
 4 # 与"source data-file"效果相同,但是更具可移植性.
 5
 6 #  文件"data-file"必须存在于当前工作目录,
 7 #+ 因为这个文件时使用'basename'来引用的.
 8
 9 # 现在,引用这个数据文件中的一些数据.
10
11 echo "variable1 (from data-file) = $variable1"
12 echo "variable3 (from data-file) = $variable3"
13
14 let "sum = $variable2 + $variable4"
15 echo "Sum of variable2 + variable4 (from data-file) = $sum"
16 echo "message1 (from data-file) is \"$message1\""
17 # 注意 :                        将双引号转义
18
19 print_message This is the message-print function in the data-file.
```

```
20
21
22 exit 0
```

################################End Script###

Example 11-20使用的data-file.见上边,这个文件必须和上边的脚本放在同一目录下.

################################Start Script###

```
 1 # 这是需要被脚本加载的data file.
 2 # 这种文件可以包含变量,函数,等等.
 3 # 在脚本中可以通过'source'或者'.'命令来加载.
 4
 5 # 让我们初始化一些变量.
 6
 7 variable1=22
 8 variable2=474
 9 variable3=5
10 variable4=97
11
12 message1="Hello, how are you?"
13 message2="Enough for now. Goodbye."
14
15 print_message ()
16 {
17 # Echo出传递进来的任何消息.
18
19   if [ -z "$1" ]
20   then
21     return 1
22     # 如果没有参数的话,出错.
23   fi
24
25   echo
26
27   until [ -z "$1" ]
28   do
29     # 循环处理传递到函数中的参数.
30     echo -n "$1"
31     # 每次Echo 一个参数, -n禁止换行.
32     echo -n " "
33     # 在参数间插入空格.
34     shift
35     # 下一个.
36   done
37
38   echo
39
40   return 0
41 }
```

################################End Script###

如果引入的文件本身就是一个可执行脚本的话,那么它将运行起来,当它return的时候,控制权又重新回到了引用它的脚本中.一个用source引入的脚本可以使用return 命令来达到这个目的.

也可以向需要source的脚本中传递参数.这些参数在source脚本中被认为是位置参数.

```
1 source $filename $arg1 arg2
```

你甚至可以在脚本文件中source脚本文件自身,虽然看不出有什么实际的应用价值.

Example 11-21 一个没什么用的,source自身的脚本

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # self-source.sh: 一个脚本递归的source自身.
 3 # 来自于"Stupid Script Tricks," 卷 II.
 4
 5 MAXPASSCNT=100    # source自身的最大数量.
 6
 7 echo -n  "$pass_count  "
 8 #  在第一次运行的时候,这句只不过echo出2个空格,
 9 #+ 因为$pass_count还没被初始化.
10
11 let "pass_count += 1"
12 #  假定这个为初始化的变量 $pass_count
13 #+ 可以在第一次运行的时候+1.
14 #  这句可以正常工作于Bash和pdksh,但是
15 #+ 它依赖于不可移植(并且可能危险)的行为.
16 #  更好的方法是在使用$pass_count之前,先把这个变量初始化为0.
17
18 while [ "$pass_count" -le $MAXPASSCNT ]
19 do
20   . $0   # 脚本"sources" 自身, 而不是调用自己.
21         # ./$0 (应该能够正常递归) 但是不能在这正常运行. 为什么?
22 done
23
24 #  这里发生的动作并不是真正的递归,
25 #+ 因为脚本成功的展开了自己,换句话说,
26 #+ 在每次循环的过程中
27 #+ 在每个'source'行(第20行)上
28 #  都产生了新的代码.
29 #
30 #  当然,脚本会把每个新'sourced'进来的文件的"#!"行
31 #+ 都解释成注释,而不会把它看成是一个新的脚本.
32
33 echo
34
35 exit 0   # 最终的效果就是从1数到100.
36         # 让人印象深刻.
37
38 # 练习:
39 # -----
40 # 使用这个小技巧编写一些真正能干些事情的脚本.
```

################################End Script###
exit
绝对的停止一个脚本的运行.exit命令有可以随便找一个整数变量作为退出脚本返回shell
时的退出码.使用exit 0对于退出一个简单脚本来说是种好习惯,表明成功运行.
注意: 如果不带参数的使用exit来退出,那么退出码将是脚本中最后一个命令的退出码.
等价于exit $?.
exec
这个shell内建命令将使用一个特定的命令来取代当前进程.一般的当shell遇到一个命令,
它会fork off一个子进程来真正的运行命令.使用exec内建命令,shell就不会fork了,并
且命令的执行将会替换掉当前shell.因此,当我们在脚本中使用它时,当命令实行完毕,
它就会强制退出脚本.[3]
Example 11-22 exec的效果
################################Start Script###

```
 1 #!/bin/bash
 2
 3 exec echo "Exiting \"$0\"."   # 脚本将在此退出.
 4
 5 # ----------------------------------
 6 # 下边的部分将执行不到.
 7
 8 echo "This echo will never echo."
 9
10 exit 99                  # 脚本不会在这退出.
11                         # 脚本退出后检查一下退出码
12                         #+ 使用'echo $?'命令.
13                         # 肯定不是99.
```

################################End Script###
Example 11-23 一个exec自身的脚本
################################Start Script###

```
 1 #!/bin/bash
 2 # self-exec.sh
 3
 4 echo
 5
 6 echo "This line appears ONCE in the script, yet it keeps echoing."
 7 echo "The PID of this instance of the script is still $$."
 8 #    上边这句用来根本没产生子进程.
 9
10 echo "=================== Hit Ctl-C to exit ==================="
11
12 sleep 1
13
14 exec $0   # 产生了本脚本的另一个实例,
15         #+ 并且这个实例代替了之前的那个.
16
17 echo "This line will never echo!"  # 当然会这样.
18
19 exit 0
```

################################End Script###
exec命令还能用于重新分配文件描述符.比如: exec <zzz-file将会用zzz-file来代替
stdin.
注意: find命令的 -exec选项与shell内建的exec命令是不同的.
shopt
这个命令允许shell在空闲时修改shell选项(见Example 24-1和Example 24-2).它经常出
现在启动脚本中,但是在一般脚本中也可用.需要Bash 2.0版本以上.

```
1 shopt -s cdspell
2 # Allows minor misspelling of directory names with 'cd'
2 # 使用'cd'命令时,允许产生少量的拼写错误.
3
4 cd /hpme  # 噢! 应该是'/home'.
5 pwd      # /home
6          # 拼写错误被纠正了.
```

caller
将caller命令放到函数中,将会在stdout上打印出函数调用者的信息.

```
 1 #!/bin/bash
 2
 3 function1 ()
 4 {
 5   # 在 function1 () 内部.
 6   caller 0   # 显示调用者信息.
 7 }
 8
 9 function1    # 脚本的第9行.
10
11 # 9 main test.sh
12 # ^              函数调用者所在的行号.
13 #   ^^^^           从脚本的"main"部分调用的.
14 #       ^^^^^^^    调用脚本的名字
15
16 caller 0     # 没效果,因为这个命令不再函数中.
```

caller命令也可以返回在一个脚本中被source的另一个脚本的信息.象函数一样,这是一个
"子例程调用",你会发现这个命令在调试的时候特别有用.
命令类
ture
一个返回成功(就是返回0)退出码的命令,但是除此之外什么事也不做.

```
1 # 死循环
2 while true   # 这里的true可以用":"替换
3 do
4    operation-1
5    operation-2
6    ...
7    operation-n
8    # 需要一种手段从循环中跳出来,或者是让这个脚本挂起.
9 done
```

flase
一个返回失败(非0)退出码的命令,但是除此之外什么事也不做.

```
 1 # 测试 "false"
 2 if false
 3 then
 4   echo "false evaluates \"true\""
 5 else
 6   echo "false evaluates \"false\""
 7 fi
 8 # 失败会显示"false"
 9
10
11 # while "false" 循环 (空循环)
12 while false
13 do
14   # 这里边的代码将不会走到.
15   operation-1
16   operation-2
17   ...
18   operation-n
19   # 什么事都没发生!
20 done
```

type[cmd]

与which扩展命令很相像,type cmd将给出"cmd"的完整路径.与which命令不同的是,type命令是Bash内建命令.一个很有用的选项是-a选项,使用这个选项可以鉴别所识别的参数是关键字还是内建命令,也可以定位同名的系统命令.

```
bash$ type '['
[ is a shell builtin
bash$ type -a '['
[ is a shell builtin
[ is /usr/bin/[
```

hash[cmds]

在shell的hash表中[4],记录指定命令的路径名,所以在shell或脚本中在调用这个命令的话,shell或脚本将不需要再在$PATH中重新搜索这个命令了.如果不带参数的调用hash命令,它将列出所有已经被hash的命令.-r选项会重新设置hash表.

bind

bind内建命令用来显示或修改readline[5]的键绑定.

help

获得shell内建命令的一个小的使用总结.这与whatis命令比较象,但是help是内建命令.

```
bash$ help exit
exit: exit [n]
    Exit the shell with a status of N.  If N is omitted, the exit status
    is that of the last command executed.
```

11.1 作业控制命令

下边的作业控制命令需要一个"作业标识符"作为参数.见这章结尾的表.

jobs

在后台列出所有正在运行的作业,给出作业号.

注意: 进程和作业的概念太容易混淆了.特定的内建命令,比如kill,disown和wait即可以接受一个作业号作为参数也可以接受一个作为参数.但是fg,bg和jobs命令只能接受

作业号作为参数.
bash$ sleep 100 &
[1] 1384
bash $ jobs
[1]+ Running sleep 100 &
注意: "1"是作业号(作业是被当前shell所维护的),而"1384"是进程号(进程是被系统维护的).为了kill掉作业/进程,或者使用 kill %1命令或者使用kill 1384命令,这两个命令都可以.
感谢,S.C.

disown
从shell的当前作业表中,删除作业.

fg,bg
fg命令可以把一个在后台运行的作业放到前台来运行.而bg命令将会重新启动一个挂起的作业,并且在后台运行它.如果使用fg或者bg命令的时候没指定作业号,那么默认将对当前正在运行的作业做操作.

wait
停止脚本的运行,直到后台运行的所有作业都结束为止,或者直到指定作业号或进程号为选项的作业结束为止.
你可以使用wait命令来防止在后台作业没完成(这会产生一个孤儿进程)之前退出脚本.

Example 11-24 在继续处理之前,等待一个进程的结束

```
##################################Start Script#############################################
 1 #!/bin/bash
 2
 3 ROOT_UID=0   # 只有$UID 为0的用户才拥有root权限.
 4 E_NOTROOT=65
 5 E_NOPARAMS=66
 6
 7 if [ "$UID" -ne "$ROOT_UID" ]
 8 then
 9   echo "Must be root to run this script."
10   # "Run along kid, it's past your bedtime."
11   exit $E_NOTROOT
12 fi
13
14 if [ -z "$1" ]
15 then
16   echo "Usage: `basename $0` find-string"
17   exit $E_NOPARAMS
18 fi
19
20
21 echo "Updating 'locate' database..."
22 echo "This may take a while."
23 updatedb /usr &     # 必须使用root身份来运行.
24
25 wait
26 # 将不会继续向下运行,除非 'updatedb'命令执行完成.
27 # 你希望在查找文件名之前更新database.
```

```
28
29 locate $1
30
31 #  如果没有'wait'命令的话,而且在比较糟的情况下,
32 #+ 脚本可能在'updatedb'命令还在运行的时候退出,
33 #+ 这将会导致'updatedb'成为一个孤儿进程.
34
35 exit 0
```

################################End Script###

当然,wait 也可以接受一个作业标识符作为参数,比如,wait %1或wait $PPID.见"作业标识符表".

注意: 在一个脚本中,使用一个后台运行的命令(使用&)可能会使这个脚本挂起,直到敲回车,挂起才会被恢复.看起来只有这个命令的结果需要输出到stdout的时候才会发生这种现象.这会是一个很烦人的现象.

```
 1 #!/bin/bash
 2 # test.sh
 3
 4 ls -l &
 5 echo "Done."
```

```
bash$ ./test.sh
Done.
[bozo@localhost test-scripts]$ total 1
-rwxr-xr-x    1 bozo     bozo          34 Oct 11 15:09 test.sh
```

–

看起来在这个后台运行命令的后边放上一个wait命令可能会解决这个问题.

```
 1 #!/bin/bash
 2 # test.sh
 3
 4 ls -l &
 5 echo "Done."
 6 wait
```

```
bash$ ./test.sh
Done.
[bozo@localhost test-scripts]$ total 1
-rwxr-xr-x    1 bozo     bozo          34 Oct 11 15:09 test.sh
```

如果把这个后台运行命令的输出重定向到文件中或者重定向到/dev/null中,也能解决这个问题.

suspend

这个命令的效果与Control-Z很相像,但是它挂起的是这个shell(这个shell的父进程应该在合适的时候重新恢复它).

logout

退出一个登陆的shell,也可以指定一个退出码.

times

给出执行命令所占的时间,使用如下形式输出:

```
0m0.020s 0m0.020s
```

这是一种很有限的能力,因为这不常出现于shell脚本中.

kill
通过发送一个适当的结束信号,来强制结束一个进程(见Example 13-6).
Example 11-25 一个结束自身的脚本.

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # self-destruct.sh
 3
 4 kill $$  # 脚本将在此处结束自己的进程.
 5         # Recall that "$$" is the script's PID.
 5         # 回忆一下,"$$"就是脚本的PID.
 6
 7 echo "This line will not echo."
 8 # 而且shell将会发送一个"Terminated"消息到stdout.
 9
10 exit 0
11
12 #  在脚本结束自身进程之后,
13 #+ 它返回的退出码是什么?
14 #
15 # sh self-destruct.sh
16 # echo $?
17 # 143
18 #
19 # 143 = 128 + 15
20 #            结束信号
################################End Script#########################################
```

注意: kill -l将列出所有信号. kill -9 是"必杀"命令,这个命令将会结束哪些顽固的
不想被kill掉的进程.有时候kill -15也可以干这个活.一个僵尸进程不能被登陆的
用户kill掉, -- 因为你不能杀掉一些已经死了的东西 -- ,但是init进程迟早会
把它清除干净.僵尸进程就是子进程已经结束掉,而父进程却没kill掉这个子进程,
那么这个子进程就成为僵尸进程.

command
command命令会禁用别名和函数的查找.它只查找内部命令以及搜索路径中找到的脚本或可
执行程序.(译者,只在要执行的命令与函数或别名同名时使用,因为函数的优先级比内建命
令的优先级高)
(译者:注意一下bash执行命令的优先级:
1.别名
2.关键字
3.函数
4.内置命令
5.脚本或可执行程序($PATH)
)
注意: 当象运行的命令或函数与内建命令同名时,由于内建命令比外部命令的优先级高,而
函数比内建命令优先级高,所以bash将总会执行优先级比较高的命令.这样你就没有选
择的余地了.所以Bash提供了3个命令来让你有选择的机会.command命令就是这3个命
令之一.
另外两个是builtin和enable.

builtin

在"builtin"后边的命令将只调用内建命令.暂时的禁用同名的函数或者是同名的扩展命令.

enable

这个命令或者禁用内建命令或者恢复内建命令.如: enable -n kill将禁用kill内建命令,
所以当我们调用kill时,使用的将是/bin/kill外部命令.
-a选项将会恢复相应的内建命令,如果不带参数的话,将会恢复所有的内建命令.
选项-f filename将会从适当的编译过的目标文件[6]中以共享库(DLL)的形式来加载一个
内建命令.

autoload

这是从ksh的autoloader命令移植过来的.一个带有"autoload"声明的函数,在它第一次被
调用的时候才会被加载.[7] 这样做会节省系统资源.
注意: autoload命令并不是Bash安装时候的核心命令的一部分.这个命令需要使用命令
enable -f(见上边enable命令)来加载.

Table 11-1 作业标识符

记法	含义
%N	作业号[N]
%S	以字符串S开头的被(命令行)调用的作业
%?S	包含字符串S的被(命令行)调用的作业
%%	当前作业(前台最后结束的作业,或后台最后启动的作业)
%+	当前作业(前台最后结束的作业,或后台最后启动的作业)
%-	最后的作业
$!	最后的后台进程

注意事项:

[1] 一个例外就是time命令,Bash官方文档说这个命令是一个关键字.

[2] 一个选项就是一个行为上比较象标志位的参数,可以用来打开或关闭脚本的某些行为.
而和某个特定选项相关的参数就是用来控制这个选项功能是开启还是关闭的.

[3] 除非exec被用来重新分配文件描述符.

[4] hash是一种处理存储在表中数据的方法,这种方法就是为表中的数据建立查找键.
而数据项本身是不规则的,这样就可以通过一个简单的数学算法来产生一个数字,
这个数字被用来作为查找键.
使用hash的一个最有利的地方就是提高了速度.而缺点就是会产生"冲撞" -- 也就是
说,可能会有多个数据元素使用同一个主键.
关于hash的例子见 Example A-21 和 Example A-22.

[5] 在一个交互的shell中,readline库就是Bash用来读取输入的.
(译者: 比如默认的Emacs风格的输入,当然也可以改为vi风格的输入)

[6] 一些可加载的内建命令的C源代码都放在/usr/share/doc/bash-?.??/functions下.
注意: enable命令的-f选项并不是对所有系统都支持的(看移没移植上).

[7] typeset -fu可以达到和autoload命令相同的作用.

第12章 外部过滤器,程序和命令

==========

标准的 UNIX 命令使得 shell 脚本更加灵活.通过简单的编程结构把shell指令和系统命令结合起来,这才是脚本能力的所在.

12.1 基本命令

新手必须要掌握的初级命令

ls

基本的列出所有文件的命令.但是往往就是因为这个命令太简单,所以我们总是低估它.比如,用 -R 选项,这是递归选项,ls 将会以目录树的形式列出所有文件, 另一个很有用的选项是 -S ,将会按照文件尺寸列出所有文件, -t, 将会按照修改时间来列出文件,-i 选项会显示文件的inode(见 Example 12-4).

Example 12-1 使用ls命令来创建一个烧录CDR的内容列表

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # ex40.sh (burn-cd.sh)
 3 # 自动刻录CDR的脚本.
 4
 5
 6 SPEED=2         # 如果你的硬件支持的话,你可以选用更高的速度.
 7 IMAGEFILE=cdimage.iso
 8 CONTENTSFILE=contents
 9 DEVICE=cdrom
10 # DEVICE="0,0"     为了使用老版本的CDR
11 DEFAULTDIR=/opt  # 这是包含需要被刻录内容的目录.
12                  # 必须保证目录存在.
13                  # 小练习: 测试一下目录是否存在.
14
15 # Uses Joerg Schilling's "cdrecord" package:
15 # 使用 Joerg Schilling 的 "cdrecord"包:
16 # http://www.fokus.fhg.de/usr/schilling/cdrecord.html
17
18 #  如果一般用户调用这个脚本的话,可能需要root身份
19 #+ chmod u+s /usr/bin/cdrecord
20 #  当然, 这会产生安全漏洞, 虽然这是一个比较小的安全漏洞.
21
22 if [ -z "$1" ]
23 then
24   IMAGE_DIRECTORY=$DEFAULTDIR
25   # 如果命令行没指定的话, 那么这个就是默认目录.
26 else
27     IMAGE_DIRECTORY=$1
28 fi
29
30 # 创建一个内容列表文件.
31 ls -lRF $IMAGE_DIRECTORY > $IMAGE_DIRECTORY/$CONTENTSFILE
32 #  "l" 选项将给出一个"长"文件列表.
33 #  "R" 选项将使这个列表递归.
```

```
34 #  "F" 选项将标记出文件类型 (比如: 目录是以 /结尾, 而可执行文件以 *结尾).
35 echo "Creating table of contents."
36
37 # 在烧录到CDR之前创建一个镜像文件.
38 mkisofs -r -o $IMAGEFILE $IMAGE_DIRECTORY
39 echo "Creating ISO9660 file system image ($IMAGEFILE)."
40
41 # 烧录CDR.
42 echo "Burning the disk."
43 echo "Please be patient, this will take a while."
44 cdrecord -v -isosize speed=$SPEED dev=$DEVICE $IMAGEFILE
45
46 exit $?
##################################End Script##########################################
```

cat, tac

cat, 是单词 concatenate的缩写, 把文件的内容输出到stdout. 当与重定向操作符 (> 或 >>)结合使用时, 一般都是用来将多个文件连接起来.

```
1 # Uses of 'cat'
2 cat filename                    # 打印出文件内容.
3
4 cat file.1 file.2 file.3 > file.123   # 把3个文件连接到一个文件中.
```

cat 命令的 -n 选项是为了在目标文件中的所有行前边插入行号. -b 选项 与 -n 选项一样, 区别是不对空行进行编号. -v 选项可以使用 ^ 标记法 来echo 出不可打印字符.-s选项可以把多个空行压缩成一个空行.

见 Example 12-25 和 Example 12-21.

注意: 在一个 管道 中, 可能有一种把stdin 重定向 到一个文件中的更有效的办法, 这种方法比 cat文件的方法更有效率.

```
1 cat filename | tr a-z A-Z
2
3 tr a-z A-Z < filename   #  效果相同,但是处理更少,
4                         #+ 并且连管道都省掉了.
```

tac 命令, 就是 cat的反转, 将从文件的结尾列出文件.

rev

把每一行中的内容反转, 并且输出到 stdout上. 这个命令与 tac命令的效果是不同的, 因为它并不反转行序, 而是把每行的内容反转.

```
bash$ cat file1.txt
This is line 1.
This is line 2.
bash$ tac file1.txt
This is line 2.
This is line 1.
bash$ rev file1.txt
.1 enil si sihT
.2 enil si sihT
```

cp

这是文件拷贝命令. cp file1 file2 把 file1 拷贝到 file2, 如果存在 file2 的话,那 file2 将被覆盖 (见 Example 12-6).

注意: 特别有用的选项就是 -a 归档 选项 (为了copy一个完整的目录树), -u 是更新选

项, 和 -r 与 -R 递归选项.

```
1 cp -u source_dir/* dest_dir
2 #  "Synchronize" dest_dir to source_dir把源目录"同步"到目标目录上,
3 #+  也就是拷贝所有更新的文件和之前不存在的文件.
```

mv

这是文件移动命令. 它等价于 cp 与 rm 命令的组合. 它可以把多个文件移动到目录中,甚至将目录重命名. 想查看 mv 在脚本中使用的例子, 见 Example 9-18 和 Example A-2.

注意: 当使用非交互脚本时,可以使用 mv 的-f (强制) 选项来避免用户的输入.

当一个目录被移动到一个已存在的目录时,那么它将成为目标目录的子目录.

```
bash$ mv source_directory target_directory
bash$ ls -lF target_directory
total 1
drwxrwxr-x    2 bozo  bozo      1024 May 28 19:20 source_directory/
```

rm

删除(清除)一个或多个文件. -f 选项将强制删除文件,即使这个文件是只读的.并且可以用来避免用户输入(在非交互脚本中使用).

注意: rm 将无法删除以破折号开头的文件.

```
bash$ rm -badname
rm: invalid option -- b
Try `rm --help' for more information.
```

解决这个问题的一个方法就是在要删除的文件的前边加上"./".

```
bash$ rm ./-badname
```

另一种解决的方法是 在文件名前边加上 " -- ".

```
bash$ rm -- -badname
```

注意: 当使用递归参数 -r时, rm 命令将会删除整个目录树. 如果不慎使用 rm -rf *那整个目录树就真的完了.

rmdir

删除目录. 但是只有这个目录中没有文件 -- 当然会包含不可见的 点文件 [1] -- 的时候这个命令才会成功.

mkdir

生成目录, 创建一个空目录. 比如, mkdir -p project/programs/December 将会创建出这个指定的目录, 即使project目录和programs目录都不存在. -p 选项将会自动产生必要的父目录, 这样也就同时创建了多个目录.

chmod

修改一个现存文件的属性 (见 Example 11-12).

```
1 chmod +x filename
2 # 使得文件filename对所有用户都可执行.
3
4 chmod u+s filename
5 # 设置"filename"文件的"suid"位.
6 # 这样一般用户就可以执行"filename", 他将拥有和文件宿主相同的权限.
7 # (这并不适用于shell 脚本)
1 chmod 644 filename
2 # Makes "filename" readable/writable to owner, readable to
3 # 设置文件宿主的 r/w 权限,并对一般用户
3 # 设置读权限.
4 # (8进制模式).
1 chmod 1777 directory-name
```

2 # 对这个目录设置r/w 和可执行权限, 并开放给所有人.
3 # 同时设置 "粘贴位".
4 # 这意味着, 只有目录宿主,
5 # 文件宿主, 当然, 还有root
6 # 可以删除这个目录中的任何特定的文件.

chattr

修改文件属性. 这个命令与上边的 chmod 命令相类似, 但是有不同的选项和不同的调用语法, 并且这个命令只能工作在ext2文件系统中.

chattr 命令的一个特别有趣的选项是i. chattr +i filename 将使得这个文件被标记为永远不变. 这个文件将不能被修改, 连接, 或删除, 即使是root也不行. 这个文件属性只能被root设置和删除. 类似的, a 选项将会把文件标记为只能追加数据.

root# chattr +i file1.txt

root# rm file1.txt

rm: remove write-protected regular file `file1.txt'? y

rm: cannot remove `file1.txt': Operation not permitted

如果文件设置了s(安全)属性, 那么当这个文件被删除时,这个文件所在磁盘的块将全部被0填充.

如果文件设置了u(不可删除)属性, 那么当这个文件被删除后, 这个文件的内容还可以被恢复(不可删除).

如果文件设置了c(压缩)属性, 那么当这个文件在进行写操作时,它将自动被压缩,并且在读的时候, 自动解压.

注意: 使用命令chattr do设置的属性, 将不会显示在文件列表中(ls -l).

ln

创建文件链接, 前提是这个文件是存在的. "链接" 就是一个文件的引用, 也就是这个文件的另一个名字. ln 命令允许对同一个文件引用多个链接,并且是避免混淆的一个很好的方法 (见 Example 4-6).

ln 对于文件来说只不过是创建了一个引用, 一个指针而已, 因为创建出来的连接文件只有几个字节.

绝大多数使用ln 命令时使用是 -s 选项, 可以称为符号链接, 或软链接.使用 -s 选项的一个优点是它可以穿越文件系统来链接目录.

关于使用这个命令的语法还是有点小技巧的. 比如: ln -s oldfile newfile 将对老文件产生一个新的文件链接.

注意: 如果之前就存在newfile的话, 那么将会产生一个错误消息.

使用链接中的哪种类型?

就像 John Macdonald 解释的那样:

不论是那种类型的链接, 都提供了一种双向引用的手段 -- 也就是说, 不管你用文件的那个名字对文件内容进行修改, 你修改的效果都即会反映到原始名字的文件, 也会反映到链接名字的文件.当你工作在更高层次的时候, 才会发生软硬链接的不同. 硬链接的优点是, 原始文件与链接文件之间是相互独立的 -- 如果你删除或者重命名老文件, 那么这种操作将不会影响硬链接的文件, 硬链接的文件讲还是原来文件的内容. 然而如果你使用软链接的, 当你把老文件删除或重命名后, 软链接将再也找不到原来文件的内容了. 而软链接的优点是它可以跨越文件系统(因为它只不过是文件名的一个引用, 而并不是真正的数据). 与硬链接的另一个不同是, 一个符号链接可以指向一个目录.

链接给出了一种可以用多个名字来调用脚本的能力(当然这也适用于任何可执行的类型), 并且脚本的行为将依赖于脚本是如何被调用的.

Example 12-2 Hello or Good-bye

#################################Start Script###

```
1 #!/bin/bash
2 # hello.sh: 显示"hello" 还是 "goodbye"
3 #+         依赖于脚本是如何被调用的.
4
5 # 在当前目录下($PWD)为这个脚本创建一个链接:
6 #    ln -s hello.sh goodbye
7 # 现在, 通过如下两种方法来调用这个脚本:
8 # ./hello.sh
9 # ./goodbye
10
11
12 HELLO_CALL=65
13 GOODBYE_CALL=66
14
15 if [ $0 = "./goodbye" ]
16 then
17   echo "Good-bye!"
18   # 当然, 在这里你也可以添加一些其他的 goodbye类型的命令.Some other goodbye-type commands, as appropriate.
19   exit $GOODBYE_CALL
20 fi
21
22 echo "Hello!"
23 # 当然, 在这里你也可以添加一些其他的 hello类型的命令.
24 exit $HELLO_CALL
```

##################################End Script##

man, info

These 这两个命令用来查看系统命令或安装工具的手册和信息.当两者都可用时, info 页一般比 man也会包含更多的细节描述.

注意事项:

[1] Dotfiles 就是文件名以"."开头的文件, 比如 ~/.Xdefaults. 这样的文件在一般的 ls 命令使用中将不会被显示出来 (当然 ls -a 将会显示它们), 并且它们也不会被一个意外的 rm -rf *删除. 在用户的home目录中,Dotfiles 一般被用来当作安装和配置文件.

12.2 复杂命令

更高级的用户命令

find

-exec COMMAND \;

在每一个find 匹配到的文件执行 COMMAND 命令. 命令序列以 ; 结束(";" 是 转义符 以保证 shell 传递到find命令中的字符不会被解释为其他的特殊字符).

```
bash$ find ~/ -name '*.txt'
/home/bozo/.kde/share/apps/karm/karmdata.txt
/home/bozo/misc/irmeyc.txt
/home/bozo/test-scripts/1.txt
```

如果 COMMAND 中包含 {}, 那么 find 命令将会用所有匹配文件的路径名来替换 "{}" .

```
1 find ~/ -name 'core*' -exec rm {} \;
2 # 从用户的 home 目录中删除所有的 core dump文件.
 1 find /home/bozo/projects -mtime 1
```

```
 2 #  列出最后一天被修改的
 3 #+ 在/home/bozo/projects目录树下的所有文件.
 4 #
 5 #  mtime = last modification time of the target file
 6 #  ctime = last status change time (via 'chmod' or otherwise)
 7 #  atime = last access time
 8
 9 DIR=/home/bozo/junk_files
10 find "$DIR" -type f -atime +5 -exec rm {} \;
11 #                             ^^
12 #  大括号就是"find"命令用来替换目录的地方.
13 #
14 #  删除至少5天内没被存取过的
15 #+ "/home/bozo/junk_files" 中的所有文件.
16 #
17 #  "-type filetype", where
18 #  f = regular file
19 #  d = directory, etc.
20 #  ('find' 命令的 man页有完整的选项列表.)
 1 find /etc -exec grep '[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*' {} \;
 2
 3 # 在/etc 目录中的文件找到所所有包含 IP 地址(xxx.xxx.xxx.xxx) 的文件.
 4 # 可能会查找到一些多余的匹配. 我们如何去掉它们呢?
 5
 6 # 或许可以使用如下方法:
 7
 8 find /etc -type f -exec cat '{}' \; | tr -c '.[:digit:]' '\n' \
 9 | grep '^[^.][^.]*\.[^.][^.]*\.[^.][^.]*\.[^.][^.]*$'
10 #
11 #  [:digit:] 是一种字符类.is one of the character classes
12 #+ 关于字符类的介绍见 POSIX 1003.2 标准化文档.
13
14 # Thanks, St é phane Chazelas.
```

注意: find 命令的 -exec 选项不应该与shell中的内建命令 exec 相混淆.

Example 12-3 删除当前目录下文件名中包含一些特殊字符(包括空白)的文件..

```
###################################Start Script#########################################
 1 #!/bin/bash
 2 # badname.sh
 3 # 删除当前目录下文件名中包含一些特殊字符的文件.
 4
 5 for filename in *
 6 do
 7   badname=`echo "$filename" | sed -n /[\+\{\;\"\\\=\?~\(\)\<\>\&\*\|\$]/p`
 8 # badname=`echo "$filename" | sed -n '/[+{;"\=?~()<>&*|$]/p'`  这句也行.
 9 # 删除文件名包含这些字符的文件:    + { ; " \ = ? ~ ( ) < > & * | $
10 #
11   rm $badname 2>/dev/null
12 #           ^^^^^^^^^^ 错误消息将被抛弃.
```

```
13 done
14
15 # 现在, 处理文件名中以任何方式包含空白的文件.
16 find . -name "* *" -exec rm -f {} \;
17 # "find"命令匹配到的目录名将替换到{}的位置.
18 # '\' 是为了保证 ';'被正确的转义, 并且放到命令的结尾.
19
20 exit 0
21
22 #-----------------------------------------------------------------
23 # 这行下边的命令将不会运行, 因为 "exit" 命令.
24
25 # 这句是上边脚本的一个可选方法:
26 find . -name '*[+{;"\\=?~()<>&*|$ ]*' -exec rm -f '{}' \;
27 # (Thanks, S.C.)
```

##################################End Script##

Example 12-4 通过文件的 inode 号来删除文件

##################################Start Script##

```
 1 #!/bin/bash
 2 # idelete.sh: 通过文件的inode号来删除文件.
 3
 4 #  当文件名以一个非法字符开头的时候, 这就非常有用了,
 5 #+ 比如 ? 或 -.
 6
 7 ARGCOUNT=1                # 文件名参数必须被传递到脚本中.
 8 E_WRONGARGS=70
 9 E_FILE_NOT_EXIST=71
10 E_CHANGED_MIND=72
11
12 if [ $# -ne "$ARGCOUNT" ]
13 then
14   echo "Usage: `basename $0` filename"
15   exit $E_WRONGARGS
16 fi
17
18 if [ ! -e "$1" ]
19 then
20   echo "File \""$1"\" does not exist."
21   exit $E_FILE_NOT_EXIST
22 fi
23
24 inum=`ls -i | grep "$1" | awk '{print $1}'`
25 # inum = inode (索引节点) 号.
26 # ----------------------------------------------------------
27 # 每个文件都有一个inode号, 这个号用来记录文件物理地址信息.
28 # ----------------------------------------------------------
29
30 echo; echo -n "Are you absolutely sure you want to delete \"$1\" (y/n)? "
```

```
31 # 'rm' 命令的 '-v' 选项也会问这句话.
32 read answer
33 case "$answer" in
34 [nN]) echo "Changed your mind, huh?"
35       exit $E_CHANGED_MIND
36       ;;
37 *)    echo "Deleting file \"$1\".";;
38 esac
39
40 find . -inum $inum -exec rm {} \;
41 #                         ^^
42 #       大括号就是"find"命令
43 #+      用来替换文本输出的地方.
44 echo "File "\"$1"\" deleted!"
45
46 exit 0
```

################################End Script###

见 Example 12-27, Example 3-4, 和 Example 10-9 这些例子展示了使用 find 命令. 对于这个复杂而有强大的命令来说, 查看man页可以获得更多的细节.

xargs

这是给命令传递参数的一个过滤器, 也是组合多个命令的一个工具.它把一个数据流分割为一些足够小的块, 以方便过滤器和命令进行处理. 由此这个命令也是后置引用的一个强有力的替换. 在一般使用过多参数的命令替换失败的时候,用xargs 来替换它一般都能成功.

[1] 通常情况下, xargs 从管道或者stdin中读取数据, 但是它也能够从文件的输出中读取数据.

xargs的默认命令是 echo. 这意味着通过管道传递给xargs的输入将会包含换行和空白, 不过通过xargs的处理, 换行和空白将被空格取代.

```
bash$ ls -l
total 0
-rw-rw-r--    1 bozo  bozo        0 Jan 29 23:58 file1
-rw-rw-r--    1 bozo  bozo        0 Jan 29 23:58 file2

bash$ ls -l | xargs
total 0 -rw-rw-r-- 1 bozo bozo 0 Jan 29 23:58 file1 -rw-rw-r-- 1 bozo bozo 0 Jan 29 23:58 file2

bash$ find ~/mail -type f | xargs grep "Linux"
./misc:User-Agent: slrn/0.9.8.1 (Linux)
./sent-mail-jul-2005: hosted by the Linux Documentation Project.
./sent-mail-jul-2005: (Linux Documentation Project Site, rtf version)
./sent-mail-jul-2005: Subject: Criticism of Bozo's Windows/Linux article
./sent-mail-jul-2005: while mentioning that the Linux ext2/ext3 filesystem
. . .
```

ls | xargs -p -l gzip 使用gzips 压缩当前目录下的每个文件, 一次压缩一个, 并且在每次压缩前都提示用户.

注意: 一个有趣的 xargs 选项是 -n NN, NN 是限制每次传递进来参数的个数.

ls | xargs -n 8 echo 以每行8列的形式列出当前目录下的所有文件.

注意: 另一个有用的选项是 -0, 使用 find -print0 或 grep -lZ 这两种组合方式. 这允许处理包含空白或引号的参数.

```
find / -type f -print0 | xargs -0 grep -liwZ GUI | xargs -0 rm -f
grep -rliwZ GUI / | xargs -0 rm -f
```

上边两行都可用来删除任何包含 "GUI" 的文件. (Thanks, S.C.)

Example 12-5 Logfile: 使用 xargs 来监控系统 log

```
################################Start Script#########################################
 1 #!/bin/bash
 2
 3 # 从 /var/log/messagesGenerates 的尾部开始
 4 # 产生当前目录下的一个lof 文件.
 5
 6 # 注意: 如果这个脚本被一个一般用户调用的话,
 7 # /var/log/messages 必须是全部可读的.
 8 #        #root chmod 644 /var/log/messages
 9
10 LINES=5
11
12 ( date; uname -a ) >>logfile
13 # 时间和机器名
14 echo ------------------------------------------------------------------- >>logfile
15 tail -$LINES /var/log/messages | xargs |  fmt -s >>logfile
16 echo >>logfile
17 echo >>logfile
18
19 exit 0
20
21 #  注意:
22 #  -----
23 #  像 Frank Wang 所指出,
24 #+ 在原文件中的任何不匹配的引号(包括单引号和双引号)
25 #+ 都会给xargs造成麻烦.
26 #
27 #  他建议使用下边的这行来替换上边的第15行:
28 #     tail -$LINES /var/log/messages | tr -d "\"" | xargs | fmt -s >>logfile
29
30
31
32 #  练习:
33 #  -----
34 #  修改这个脚本, 使得这个脚本每个20分钟
35 #+ 就跟踪一下 /var/log/messages 的修改记录.
36 #  提示: 使用 "watch" 命令.
################################End Script###########################################
```

在find命令中, 一对大括号就一个文本替换的位置.

Example 12-6 把当前目录下的文件拷贝到另一个文件中

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # copydir.sh
 3
```

```
 4 #  拷贝 (verbose) 当前目录($PWD)下的所有文件到
 5 #+ 命令行中指定的另一个目录下.
 6
 7 E_NOARGS=65
 8
 9 if [ -z "$1" ]   # 如果没有参数传递进来那就退出.
10 then
11   echo "Usage: `basename $0` directory-to-copy-to"
12   exit $E_NOARGS
13 fi
14
15 ls . | xargs -i -t cp ./{} $1
16 #            ^^ ^^      ^^
17 #  -t 是 "verbose" (输出命令行到stderr) 选项.
18 #  -i 是"替换字符串"选项.
19 #  {} 是输出文本的替换点.
20 #  这与在"find"命令中使用{}的情况很相像.
21 #
22 #  列出当前目录下的所有文件(ls .),
23 #+ 将 "ls" 的输出作为参数传递到 "xargs"(-i -t 选项) 中,
24 #+ 然后拷贝(cp)这些参数({})到一个新目录中($1).
25 #
26 #  最终的结果和下边的命令等价,
27 #+   cp * $1
28 #+ 除非有文件名中嵌入了"空白"字符.
29
30 exit 0
```

################################End Script###

Example 12-7 通过名字Kill进程

################################Start Script###

```
 1 #!/bin/bash
 2 # kill-byname.sh: 通过名字kill进程.
 3 # 与脚本kill-process.sh相比较.
 4
 5 #  例如,
 6 #+ 试一下 "./kill-byname.sh xterm" --
 7 #+ 并且查看你系统上的所有xterm都将消失.
 8
 9 #  警告:
10 #  -----
11 #  这是一个非常危险的脚本.
12 #  运行它的时候一定要小心. (尤其是以root身份运行时)
13 #+ 因为运行这个脚本可能会引起数据丢失或产生其他一些不好的效果.
14
15 E_BADARGS=66
16
17 if test -z "$1"  # 没有参数传递进来?
18 then
```

```
19   echo "Usage: `basename $0` Process(es)_to_kill"
20   exit $E_BADARGS
21 fi
22
23
24 PROCESS_NAME="$1"
25 ps ax | grep "$PROCESS_NAME" | awk '{print $1}' | xargs -i kill {} 2&>/dev/null
26 #                                          ^^      ^^
27
28 # -----------------------------------------------------------
29 # 注意:
30 # -i 参数是xargs命令的"替换字符串"选项.
31 # 大括号对的地方就是替换点.
32 # 2&>/dev/null 将会丢弃不需要的错误消息.
33 # -----------------------------------------------------------
34
35 exit $?
```

##################################End Script##

Example 12-8 使用xargs分析单词出现的频率

##################################Start Script##

```
 1 #!/bin/bash
 2 # wf2.sh: Crude word frequency analysis on a text file.
 3
 4 # 使用 'xargs' 将文本行分解为单词.
 5 # 于后边的 "wf.sh" 脚本相比较.
 6
 7
 8 # 检查命令行上输入的文件.
 9 ARGS=1
10 E_BADARGS=65
11 E_NOFILE=66
12
13 if [ $# -ne "$ARGS" ]
14 # 纠正传递到脚本中的参数个数?
15 then
16   echo "Usage: `basename $0` filename"
17   exit $E_BADARGS
18 fi
19
20 if [ ! -f "$1" ]       # 检查文件是否存在.
21 then
22   echo "File \"$1\" does not exist."
23   exit $E_NOFILE
24 fi
25
26
27
28 ######################################################################
```

```
29 cat "$1" | xargs -n1 | \
30 #  列出文件, 每行一个单词.
31 tr A-Z a-z | \
32 #  将字符转换为小写.
33 sed -e 's/\.//g'  -e 's/\,//g' -e 's/ /\
34 /g' | \
35 #  过滤掉句号和逗号,
36 #+ 并且将单词间的空格修改为换行,
37 sort | uniq -c | sort -nr
38 #  最后统计出现次数,把数字显示在第一列,然后显示单词,并按数字排序.
39 ######################################################################
40
41 #  这个例子的作用与"wf.sh"的作用是一样的,
42 #+ 但是这个例子比较臃肿, 并且运行起来更慢一些(为什么?).
43
44 exit 0
```

################################End Script###

expr

通用求值表达式: 通过给定的操作(参数必须以空格分开)连接参数,并对参数求值.可以使
算术操作, 比较操作, 字符串操作或者是逻辑操作.

```
expr 3 + 5
 返回 8
expr 5 % 3
 返回 2
expr 1 / 0
 返回错误消息, expr: division by zero
 不允许非法的算术操作.
expr 5 \* 3
 返回 15
 在算术表达式expr中使用乘法操作时, 乘法符号必须被转义.
y=`expr $y + 1`
 增加变量的值, 与 let y=y+1 和 y=$(($y+1)) 的效果相同. 这是使用算术表达式的
 一个例子.
z=`expr substr $string $position $length`
 在位置$position上提取$length长度的子串.
```

Example 12-9 使用 expr

################################Start Script#######################################

```
 1 #!/bin/bash
 2
 3 # 展示一些 'expr'的使用
 4 # ======================
 5
 6 echo
 7
 8 # 算术 操作
 9 # ---- ----
10
11 echo "Arithmetic Operators"
```

```
12 echo
13 a=`expr 5 + 3`
14 echo "5 + 3 = $a"
15
16 a=`expr $a + 1`
17 echo
18 echo "a + 1 = $a"
19 echo "(incrementing a variable)"
20
21 a=`expr 5 % 3`
22 # 取模操作
23 echo
24 echo "5 mod 3 = $a"
25
26 echo
27 echo
28
29 # 逻辑 操作
30 # ---- ----
31
32 #  true返回 1 ,false 返回 0 ,
33 #+ 而Bash的使用惯例则相反.
34
35 echo "Logical Operators"
36 echo
37
38 x=24
39 y=25
40 b=`expr $x = $y`         # 测试相等.
41 echo "b = $b"           # 0  ( $x -ne $y )
42 echo
43
44 a=3
45 b=`expr $a \> 10`
46 echo 'b=`expr $a \> 10`, therefore...'
47 echo "If a > 10, b = 0 (false)"
48 echo "b = $b"           # 0  ( 3 ! -gt 10 )
49 echo
50
51 b=`expr $a \< 10`
52 echo "If a < 10, b = 1 (true)"
53 echo "b = $b"           # 1  ( 3 -lt 10 )
54 echo
55 # Note escaping of operators.
56
57 b=`expr $a \<= 3`
58 echo "If a <= 3, b = 1 (true)"
59 echo "b = $b"           # 1  ( 3 -le 3 )
```

```
60 # 也有 "\>=" 操作 (大于等于).
61
62
63 echo
64 echo
65
66
67
68 # 字符串 操作
69 # ------ ----
70
71 echo "String Operators"
72 echo
73
74 a=1234zipper43231
75 echo "The string being operated upon is \"$a\"."
76
77 # 长度: 字符串长度
78 b=`expr length $a`
79 echo "Length of \"$a\" is $b."
80
81 # 索引: 从字符串的开头查找匹配的子串,
82 #       并取得第一个匹配子串的位置.
83 b=`expr index $a 23`
84 echo "Numerical position of first \"2\" in \"$a\" is \"$b\"."
85
86 # substr: 从指定位置提取指定长度的字串.
87 b=`expr substr $a 2 6`
88 echo "Substring of \"$a\", starting at position 2,\
89 and 6 chars long is \"$b\"."
90
91
92 #  'match' 操作的默认行为就是
93 #+ 从字符串的开始进行搜索,并匹配第一个匹配的字符串.
94 #
95 #       使用正则表达式
96 b=`expr match "$a" '[0-9]*'`              #  数字的个数.
97 echo Number of digits at the beginning of \"$a\" is $b.
98 b=`expr match "$a" '\([0-9]*\)'`          #  注意需要转义括号
99 #                  ==     ==            + 这样才能触发子串的匹配.
100 echo "The digits at the beginning of \"$a\" are \"$b\"."
101
102 echo
103
104 exit 0
##################################End Script##########################################
```

注意: ":" 操作可以替换 match. 比如, b=`expr $a : [0-9]*`与上边所使用的 b=`expr match $a [0-9]*` 完全等价.

################################Start Script###

```
 1 #!/bin/bash
 2
 3 echo
 4 echo "String operations using \"expr \$string : \" construct"
 5 echo "==================================================="
 6 echo
 7
 8 a=1234zipper5FLIPPER43231
 9
10 echo "The string being operated upon is \"`expr "$a" : '\(.*\)'`\"."
11 #    转义括号对操作.                          ==  ==
12
13 #      ***************************
14 #+           转移括号对
15 #+         用来匹配一个子串
16 #      ***************************
17
18
19 #  如果不转义括号的话...
20 #+ 那么 'expr' 将把string操作转换为一个整数.
21
22 echo "Length of \"$a\" is `expr "$a" : '.*'`."   # 字符串长度
23
24 echo "Number of digits at the beginning of \"$a\" is `expr "$a" : '[0-9]*'`."
25
26 # ------------------------------------------------------------------------ #
27
28 echo
29
30 echo "The digits at the beginning of \"$a\" are `expr "$a" : '\([0-9]*\)'`."
31 #                                          ==     ==
32 echo "The first 7 characters of \"$a\" are `expr "$a" : '\(.......\)'`."
33 #       =====                               ==      ==
34 # 再来一个, 转义括号对强制一个子串匹配.
35 #
36 echo "The last 7 characters of \"$a\" are `expr "$a" : '.*\(.......\)'`."
37 #       ====              end of string operator  ^^
38 #  (最后这个模式的意思是忽略前边的任何字符,直到最后7个字符,
39 #+  最后7个点就是需要匹配的任意7个字符的字串)
40
41 echo
42
43 exit 0
```

################################End Script###

上边的脚本展示了expr是如何使用转义的括号对 -- \(... \) -- 和 正则表达式 一起来分析和匹配子串. 下边是另外一个例子, 这次的例子是真正的应用用例.

```
1 # 去掉字符串开头和结尾的空白.
```

```
2 LRFDATE=`expr "$LRFDATE" : '[[:space:]]*\(.*\)[[:space:]]*$'`
3
4 #  来自于 Peter Knowle的 "booklistgen.sh" 脚本
5 #+ 用来将文件转换为Sony Librie格式.
6 #  (http://booklistgensh.peterknowles.com)
```

Perl, sed, 和 awk 是更强大的字符串分析工具. 在脚本中嵌入一段比较短的 sed 或 awk 操作 (见 Section 33.2) 比使用 expr 更加有吸引力.

见 Section 9.2 将会有更多使用 expr 进行字符串操作的例子.

注意事项:

[1] 即使在不必非得强制使用 xargs 的时候, 使用 xargs 也可以明显地提高多文件批处理执行命令的速度.

[]

高级Bash脚本编程指南(三)（下）

文章整理: 文章来源: 网络

12.3 时间/日期 命令

时间/日期 和计时

date

直接调用, date 就会打印日期和时间到 stdout 上. 这个命令有趣的地方在于它的格式化和分析选项上.

Example 12-10 使用 date 命令

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # 练习 'date' 命令
 3
 4 echo "The number of days since the year's beginning is `date +%j`."
 5 # 需要在调用格式的前边加上一个 '+' 号.
 6 # %j 给出今天是本年度的第几天.
 7
 8 echo "The number of seconds elapsed since 01/01/1970 is `date +%s`."
 9 #  %s 将产生从 "UNIX 元年" 到现在为止的秒数,yields number of seconds since "UNIX epoch" began,
10 #+ 但是这东西有用么?
11
12 prefix=temp
13 suffix=$(date +%s)  # 'date'命令的 "+%s" 选项是 GNU-特性.
14 filename=$prefix.$suffix
15 echo $filename
16 #  这是一种非常好的产生 "唯一" 的临时文件的办法,
17 #+ 甚至比使用 $$ 都强.
18
19 # 如果想了解 'date' 命令的更多选项, 请查阅这个命令的 man 页.
20
21 exit 0
################################End Script#########################################
 -u 选项将给出 UTC (译者: UTC 是协调世界时英文缩写) 时间(Universal Coordinated T
 ime).
 bash$ date
```

Fri Mar 29 21:07:39 MST 2002

bash$ date -u
Sat Mar 30 04:07:42 UTC 2002
date 命令有许多的输出选项. 比如 %N 将以10亿分之一为单位表示当前时间. 这个选项的
一个有趣的用法就是用来产生一个6位的随机数.
1 date +%N | sed -e 's/000$//' -e 's/^0//'
2 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
3 # 去掉开头和结尾的0.
当然,还有许多其它的选项 (请查看 man date).
1 date +%j
2 # 显示今天是本年度的第几天(从1月1日开始计算).
3
4 date +%k%M
5 # 显示当前小时数和分钟数.
6
7
8
9 # 'TZ' 参数允许改变当前的默认时区.
10 date # Mon Mar 28 21:42:16 MST 2005
11 TZ=EST date # Mon Mar 28 23:42:16 EST 2005
12 # Thanks, Frank Kannemann and Pete Sjoberg, for the tip.
13
14
15 SixDaysAgo=$(date --date='6 days ago')
16 OneMonthAgo=$(date --date='1 month ago') # 4周前(不是一个月).
17 OneYearAgo=$(date --date='1 year ago')
参见 Example 3-4.
zdump
查看特定时区的当前时间.
bash$ zdump EST
EST Tue Sep 18 22:09:22 2001 EST
time
输出统计出来的命令执行的时间.
time ls -l / 给出的输出大概是如下格式:

0.00user 0.01system 0:00.05elapsed 16%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (149major+27minor)pagefaults 0swaps
参见前边章节所讲的一个类似的命令 times .
注意: 在Bash的 2.0版本 中, time 成为了shell的一个保留字, 并且在一个带有管道的
命令行中,这个命令的行为有些小的变化.
touch
这是一个用来更新文件被存取或修改的时间的工具,这个时间可以是当前系统的时间,也可
以是指定的时间, 这个命令也用来产生一个新文件.命令 touch zzz 将产生一个以zzz为名
字的0字节长度文件, 当然前提是zzz文件不存在. 为了存储时间信息, 就需要一个时间戳
为空的文件, 比如当你想跟踪一个工程的修改时间的时候,这就非常有用了.
注意: touch 命令等价于 : >> newfile 或 >> newfile (对于一个普通文件).
at

at 命令是一个作业控制命令, 用来在指定时间执行给定的命令集合.它有点像 cron 命令, 然而, at 命令主要还是用来执行那种一次性执行的命令集合.

at 2pm January 15 将会提示让你输入需要在这个时间你要执行的命令序列. 这些命令应该是可以和shell脚本兼容的,因为, 实际上, 在一个可执行的脚本中, 用户每次只能敲一行. 输入以 Ctl-D 结束.

你可以使用-f选项或者使用 (<)重定向操作符, 来让 at 命令从一个文件中读取命令集合. 这个文件其实就一个可执行的的脚本, 虽然它是一个不可交互的脚本. 在文件中包含一个 run-parts 命令, 对于执行一套不同的脚本来说是非常聪明的做法.

bash$ at 2:30 am Friday < at-jobs.list

job 2 at 2000-10-27 02:30

batch

batch 作业控制命令与 at 命令的行为很相像, 但 batch 命令被用来在系统平均载量降到 0.8 以下时执行一次性的任务. 与 at 命令相似的是, 它也可以使用 -f 选项来从文件中读取命令.

cal

从stdout中输出一个格式比较整齐的日历. 也可以指定年和月来显示那个月的日历.

sleep

这个命令与一个等待循环的效果一样. 你可以指定需要暂停的秒数, 这段时间将什么都不干.当一个后台运行的进程需要偶尔检测一个事件时,这个功能很有用. 也可用于计时. 参见 Example 29-6.

1 sleep 3 # Pauses 3 seconds.

注意: sleep 命令默认为秒, 但是你也可以指定天数, 小时数或分钟数.

1 sleep 3 h # Pauses 3 hours!

注意: 如果你想每隔一段时间来运行一个命令的话, 那么 watch 命令将比 sleep 命令好得多.

usleep

Microsleep 睡眠微秒("u" 会被希腊人读成 "mu", 或者是 micro- 前缀). 与上边的 sleep 命令作用相同, 但这个命令是以百万分之一秒为单位的. 当需要精确计时, 或者需要非常频繁的监控一个正在运行的进程的时候, 这个命令非常有用.

1 usleep 30 # 暂停 30 microseconds.

这个命令是 Red Hat initscripts / rc-scripts 包的一部分.

注意: 事实上 usleep 命令并不能提供非常精确的计时, 所以如果你需要一个实时的任务的话, 这个命令并不适合.

hwclock, clock

hwclock 命令可以存取或调整硬件时钟. 这个命令的一些选项需要 root 权限. 在系统启动的时候, /etc/rc.d/rc.sysinit 这个启动文件,会使用 hwclock 来从硬件时钟中读取并设置系统时间.clock at bootup.

clock 命令与 hwclock命令完全相同.

12.4 文本处理命令

处理文本和文本文件的命令

sort

文件排序, 通常用在管道中当过滤器来使用. 这个命令可以依据指定的关键字或指定的字符位置, 对文件行进行排序. 使用 -m 选项, 它将会合并预排序的输入文件. 想了解这个命令的全部参数请参考这个命令的 info 页. 见 Example 10-9, Example 10-10, 和 Example A-8.

tsort

拓扑排序 ,读取以空格分隔的有序对, 并且依靠输入模式进行排序.

uniq
这个过滤器将会删除一个已排序文件中的重复行.这个命令经常出现在 sort命令的管道后边.

```
1 cat list-1 list-2 list-3 | sort | uniq > final.list
2 # 将3个文件连接起来,
3 # 将它们排序,
4 # 删除其中重复的行,
5 # 最后将结果重定向到一个文件中.
```

-c选项的意思是在输出行前面加上每行在输入文件中出现的次数.

```
bash$ cat testfile
This line occurs only once.
This line occurs twice.
This line occurs twice.
This line occurs three times.
This line occurs three times.
This line occurs three times.
bash$ uniq -c testfile
   1 This line occurs only once.
   2 This line occurs twice.
   3 This line occurs three times.
bash$ sort testfile | uniq -c | sort -nr
   3 This line occurs three times.
   2 This line occurs twice.
   1 This line occurs only once.
```

sort INPUTFILE | uniq -c | sort -nr 命令 先对 INPUTFILE 排序, 然后统计 每行出现的次数, 最后的(-nr 选项将会产生一个数字的反转排序). 这种命令模版一般都用来分析 log 文件或者用来分析字典列表, 或者用在那些需要检查文本词汇结构的地方.

Example 12-11 分析单词出现的频率

```
##################################Start Script#########################################
1 #!/bin/bash
2 # wf.sh: 分析文本文件中自然词汇出现的频率.
3 # "wf2.sh" 是一个效率更高的版本.
4
5
6 # 从命令行中检查输入的文件.
7 ARGS=1
8 E_BADARGS=65
9 E_NOFILE=66
10
11 if [ $# -ne "$ARGS" ]  # 检验传递到脚本中参数的个数.
12 then
13   echo "Usage: `basename $0` filename"
14   exit $E_BADARGS
15 fi
16
17 if [ ! -f "$1" ]       # 检查传入的文件参数是否存在.
18 then
19   echo "File \"$1\" does not exist."
```

```
20   exit $E_NOFILE
21 fi
22
23
24
25 ###############################################################
26 # main ()
27 sed -e 's/\.//g'  -e 's/\,//g' -e 's/ /\
28 /g' "$1" | tr 'A-Z' 'a-z' | sort | uniq -c | sort -nr
29 #                     ========================
30 #                       检查单词出现的频率
31
32 #  过滤掉句号和逗号,
33 #+ 并且把单词间的空格转化为换行,
34 #+ 然后转化为小写,
35 #+ 最后统计出现的频率并按频率排序.
36
37 #  Arun Giridhar 建议将上边的代码修改为:
38 #  . . . | sort | uniq -c | sort +1 [-f] | sort +0 -nr
39 #  这句添加了第2个排序主键, 所以
40 #+ 这个与上边等价的例子将按照字母顺序进行排序.
41 #  就像他所解释的:
42 #  "这是一个有效的根排序, 首先对频率最少的
43 #+ 列进行排序
44 #+ (单词或者字符串, 忽略大小写)
45 #+ 然后对频率最高的列进行排序."
46 #
47 #  像 Frank Wang 所解释的那样, 上边的代码等价于:
48 #+       . . . | sort | uniq -c | sort +0 -nr
49 #+ 用下边这行也行:
50 #+       . . . | sort | uniq -c | sort -k1nr -k
51 ###############################################################
52
53 exit 0
54
55 # 练习:
56 # -----
57 # 1) 使用 'sed' 命令来过滤其他的标点符号,
58 #+   比如分号.
59 # 2) 修改这个脚本, 添加能够过滤多个空格或者
60 #    空白的能力.
################################End Script#############################################
 bash$ cat testfile
 This line occurs only once.
 This line occurs twice.
 This line occurs twice.
 This line occurs three times.
 This line occurs three times.
```

This line occurs three times.
bash$ ./wf.sh testfile
6 this
6 occurs
6 line
3 times
3 three
2 twice
1 only
1 once

expand, unexpand

expand 将会把每个tab转化为一个空格.这个命令经常用在管道中.
unexpand 将会把每个空格转化为一个tab.效果与 expand 相反.

cut

一个从文件中提取特定域的工具. 这个命令与 awk 中使用的 print $N命令很相似, 但是更受限. 在脚本中使用cut命令会比使用 awk 命令来得容易一些. 最重要的选项就是 -d (字段定界符) 和 -f (域分隔符) 选项.

使用 cut 来获得所有mount上的文件系统的列表:

```
1 cut -d ' ' -f1,2 /etc/mtab
```

使用 cut 命令列出 OS 和 kernel的版本:

```
1 uname -a | cut -d" " -f1,3,11,12
```

使用 cut 命令从 e-mail 中提取消息头:

```
bash$ grep '^Subject:' read-messages | cut -c10-80
Re: Linux suitable for mission-critical apps?
MAKE MILLIONS WORKING AT HOME!!!
Spam complaint
Re: Spam complaint
```

使用 cut 命令来分析一个文件:

```
 1 # 列出所有在/etc/passwd中的用户.
 2
 3 FILENAME=/etc/passwd
 4
 5 for user in $(cut -d: -f1 $FILENAME)
 6 do
 7   echo $user
 8 done
 9
10 # Thanks, Oleg Philon for suggesting this.
```

cut -d ' ' -f2,3 filename 等价于 awk -F'[ ]' '{ print $2, $3 }' filename

注意:

你甚至可以指定换行符作为字段定界符. 这个小伎俩实际上就是在命令行上插入一个换行(RETURN).(译者: linux使用lf作为换行符的).

```
bash$ cut -d'
' -f3,7,19 testfile
This is line 3 of testfile.
This is line 7 of testfile.
This is line 19 of testfile.
Thank you, Jaka Kranjc, for pointing this out.
```

参见 Example 12-43.

paste

将多个文件,以每个文件一列的形式合并到一个文件中, 合并后的文件没列就是原来的一个文件.对于创建系统log文件来说, 使用 cut 命令与 paste 命令相结合是非常有用的.

join

这个命令与 paste 命令属于同类命令, 但是它能够完成某些特殊的目地. 这个强力工具能够以一种特殊的形式来合并2个文件, 这种特殊的形式本质上就是一个关联数据库的简单版本.

join 命令只能够操作2个文件, 它可以将那些具有特定标记域(通常是一个数字标签)的行合并起来, 并且将结果输出到stdout. 被加入的文件应该事先根据标记域进行排序以便于能够正确的匹配.

```
1 File: 1.data
2
3 100 Shoes
4 200 Laces
5 300 Socks
1 File: 2.data
2
3 100 $40.00
4 200 $1.00
5 300 $2.00
bash$ join 1.data 2.data
File: 1.data 2.data
100 Shoes $40.00
200 Laces $1.00
300 Socks $2.00
```

注意: 在输出中标记域将只会出现一次.

head

将一个文件的头打印到stdout上 (默认为10行, 可以自己修改). 这个命令也有一些有趣的选项.

Example 12-12 那个文件是脚本?

```
##################################Start Script#########################################
1 #!/bin/bash
2 # script-detector.sh: 在一个目录中检查所有的脚本文件.
3
4 TESTCHARS=2    # 测试前两个字节.
5 SHABANG='#!'   # 脚本都是以 "sha-bang." 开头的.
6
7 for file in *  # 遍历当前目录下的所有文件.
8 do
9   if [[ `head -c$TESTCHARS "$file"` = "$SHABANG" ]]
10  #      head -c2                    #!
11  #  '-c' 选项将从文件头输出指定个数的字符,
12  #+ 而不是默认的行数.
13  then
14    echo "File \"$file\" is a script."
15  else
16    echo "File \"$file\" is *not* a script."
```

```
17   fi
18 done
19
20 exit 0
21
22 #  练习:
23 #  -----
24 #  1) 将这个脚本修改为可以指定目录
25 #+    来扫描目录下的脚本.
26 #+    (而不是只搜索当前目录).
27 #
28 #  2) 就目前看来, 这个脚本将不能正确识别出
29 #+    Perl, awk, 和其他一些脚本语言的脚本文件.
30 #     修正这个问题.
```

################################End Script###

Example 12-13 产生10进制随机数

################################Start Script###

```
 1 #!/bin/bash
 2 # rnd.sh: 输出一个10进制随机数
 3
 4 # Script by Stephane Chazelas.
 5
 6 head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'
 7
 8
 9 # ============================================================== #
10
11 # 分析
12 # ----
13
14 # head:
15 # -c4 选项将取得前4个字节.
16
17 # od:
18 # -N4 选项将限制输出为4个字节.
19 # -tu4 选项将使用无符号10进制格式来输出.
20
21 # sed:
22 # -n 选项, 使用 "s" 命令与 "p" 标志组合的方式,
23 # 将会只输出匹配的行.
24
25
26
27 # 本脚本作者解释 'sed' 命令的行为如下.
28
29 # head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'
30 # ----------------------------------> |
31
```

```
32 # 假设一直处理到 "sed"命令时的输出--> |
33 # 为 0000000 1198195154\n
34
35 #  sed 命令开始读取字串: 0000000 1198195154\n.
36 #  这里它发现一个换行符,
37 #+ 所以 sed 准备处理第一行 (0000000 1198195154).
38 #  sed命令开始匹配它的 <range> 和 <action>. 第一个匹配的并且只有这一个匹配的:
39
40 #   range     action
41 #   1         s/.* //p
42
43 #  因为行号在range中, 所以 sed 开始执行 action:
44 #+ 替换掉以空格结束的最长的字符串, 在这行中这个字符串是
45 #  ("0000000 ") ,用空字符串(//)将这个匹配到的字串替换掉, 如果成功, 那就打印出结果
46 #  ("p" 在这里是 "s" 命令的标志, 这与单独的 "p" 命令是不同的).
47
48 #  sed 命令现在开始继续读取输入. (注意在继续之前,
49 #+ continuing, 如果没使用 -n 选项的话, sed 命令将再次
50 #+ 将这行打印一遍).
51
52 # 现在, sed 命令读取剩余的字符串, 并且找到文件的结尾.
53 # sed 命令开始处理第2行(这行也被标记为 '$'
54 # 因为这已经是最后一行).
55 # 所以这行没被匹配到 <range> 中, 这样sed命令就结束了.
56
57 #  这个 sed 命令的简短的解释是:
58 #  "在第一行中删除第一个空格左边全部的字符,
59 #+ 然后打印出来."
60
61 # 一个更好的来达到这个目的的方法是:
62 #         sed -e 's/.* //;q'
63
64 # 这里, <range> 和 <action> 分别是 (也可以写成
65 #         sed -e 's/.* //' -e q):
66
67 #   range                 action
68 #   nothing (matches line)   s/.* //
69 #   nothing (matches line)   q (quit)
70
71 #  这里, sed 命令只会读取第一行的输入.
72 #  将会执行2个命令, 并且会在退出之前打印出(已经替换过的)这行(因为 "q" action),
73 #+ 因为没使用 "-n" 选项.
74
75 # ============================================================== #
76
77 # 也可以使用如下一个更简单的语句来代替:
78 #         head -c4 /dev/urandom| od -An -tu4
79
```

```
80 exit 0
################################End Script#########################################
```

参见 Example 12-35.

tail

将一个文件的结尾输出到 stdout 中(默认为 10 行). 通常用来跟踪一个系统 logfile 的修改状况, 使用 -f 选项的话, tail 命令将会继续显示添加到文件中的行.

Example 12-14 使用 tail 命令来监控系统log

```
################################Start Script#######################################
 1 #!/bin/bash
 2
 3 filename=sys.log
 4
 5 cat /dev/null > $filename; echo "Creating / cleaning out file."
 6 #  如果文件不存在的话就创建文件,
 7 #+ 然后将这个文件清空.
 8 #  : > filename   和   > filename 也可以完成这个工作.
 9
10 tail /var/log/messages > $filename
11 # /var/log/messages 必须具有全局可读权限才行.
12
13 echo "$filename contains tail end of system log."
14
15 exit 0
################################End Script#########################################
```

注意:

为了列出一个文本文件中的指定行数, 可以将 head 命令的输出通过 管道 传递到 tail -1 中 . 比如 head -8 database.txt | tail -1 将会列出 database.txt 文件的第8行.

下边是将一个文本文件中指定范围的所有行都保存到一个变量中:

```
 1 var=$(head -$m $filename | tail -$n)
 2
 3 # filename = 文件名
 4 # m = 从文件开头到想取得的指定范围的行数的最后一行
 5 # n = 取得指定范围的行数 (从块结尾开始截断)
```

参见 Example 12-5, Example 12-35 和 Example 29-6.

grep

使用 正则表达式 的一个多用途文本搜索工具. 这个命令本来是 ed 行编辑器中的一个命令/过滤器: g/re/p -- global - regular expression - print.

grep pattern [file...]

在文件中搜索所有 pattern 出现的位置, pattern 既可以是要搜索的字符串,也可以是一个正则表达式.

```
bash$ grep '[rst]ystem.$' osinfo.txt
The GPL governs the distribution of the Linux operating system.
```

如果没有指定文件参数, grep 通常用在管道中对 stdout 进行过滤.

```
bash$ ps ax | grep clock
765 tty1     S      0:00 xclock
901 pts/1    S      0:00 grep clock
```

-i 选项在搜索时忽略大小写.

-w 选项用来匹配整词.
-l 选项仅列出符合匹配的文件, 而不列出匹配行.
-r (递归) 选项不仅在当前工作目录下搜索匹配, 而且搜索子目录.
-n 选项列出所有匹配行, 并显示行号.

```
bash$ grep -n Linux osinfo.txt
2:This is a file containing information about Linux.
6:The GPL governs the distribution of the Linux operating system.
```

-v (或者--invert-match) 选项将会显示所有不匹配的行.

```
1 grep pattern1 *.txt | grep -v pattern2
2
3 # 匹配在"*.txt"中所有包含 "pattern1"的行,
4 # 而不显示匹配包含 "pattern2"的行.
```

-c (--count) 选项将只会显示匹配到的行数的总数,而不会列出具体的匹配.

```
 1 grep -c txt *.sgml   # (在 "*.sgml" 文件中, 匹配"txt"的行数的总数.)
 2
 3
 4 #   grep -cz .
 5 #          ^ 点
 6 # 意思是计数 (-c) 所有以空字符分割(-z) 的匹配 "."的项
 7 # "."是正则表达式的一个符号, 表达匹配任意一个非空字符(至少要包含一个字符).
 8 #
 9 printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz .     # 3
10 printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '$'   # 5
11 printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '^'   # 5
12 #
13 printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -c '$'    # 9
14 # 默认情况下, 是使用换行符(\n)来分隔匹配项.
15
16 # 注意  -z 选项是 GNU "grep" 特定的选项.
17
18
19 # Thanks, S.C.
```

当有多个文件参数的时候, grep 将会指出哪个文件中包含具体的匹配.

```
bash$ grep Linux osinfo.txt misc.txt
osinfo.txt:This is a file containing information about Linux.
osinfo.txt:The GPL governs the distribution of the Linux operating system.
misc.txt:The Linux operating system is steadily gaining in popularity.
```

注意: 如果在 grep 命令只搜索一个文件的时候, 那么可以简单的把 /dev/null 作为第2个文件参数传给 grep .

```
 bash$ grep Linux osinfo.txt /dev/null
 osinfo.txt:This is a file containing information about Linux.
 osinfo.txt:The GPL governs the distribution of the Linux operating system.
```

如果存在一个成功的匹配, 那么 grep 命令将会返回 0 作为 退出状态 ,这样就可以将 grep 命令的结果放在脚本的条件测试中来使用, 尤其和 -q (禁止输出)选项组合时特别有用.

```
 1 SUCCESS=0                  # 如果 grep 匹配成功
 2 word=Linux
 3 filename=data.file
```

```
 4
 5 grep -q "$word" "$filename"    # "-q" 选项将使得什么都不输出到 stdout 上.
 6
 7 if [ $? -eq $SUCCESS ]
 8 # if grep -q "$word" "$filename"   这句话可以代替行 5 - 7.
 9 then
10   echo "$word found in $filename"
11 else
12   echo "$word not found in $filename"
13 fi
```

Example 29-6 展示了如何使用 grep 命令来在一个系统 logfile 中进行一个单词的模式匹配.

Example 12-15 在一个脚本中模仿 "grep" 的行为

##################################Start Script###

```
 1 #!/bin/bash
 2 # grp.sh: 一个非常粗糙的 'grep' 的实现.
 3
 4 E_BADARGS=65
 5
 6 if [ -z "$1" ]    # 检查传递给脚本的参数.
 7 then
 8   echo "Usage: `basename $0` pattern"
 9   exit $E_BADARGS
10 fi
11
12 echo
13
14 for file in *     # 遍历 $PWD 下的所有文件.
15 do
16   output=$(sed -n /"$1"/p $file)  # 命令替换.
17
18   if [ ! -z "$output" ]          # 如果"$output" 不加双引号将会发生什么?
19   then
20     echo -n "$file: "
21     echo $output
22   fi             #  sed -ne "/$1/s|^|${file}: |p"  这句与上边这段等价.
23
24   echo
25 done
26
27 echo
28
29 exit 0
30
31 # 练习:
32 # -----
33 # 1) 在任何给定的文件中,如果有超过一个匹配的话, 在输出中添加新行.
34 # 2) 添加一些特征.
```

################################End Script###
如何使用 grep 命令来搜索两个(或两个以上)独立的模式? 如果你想显示在一个或多个文
件中既匹配"pattern1" 又匹配 "pattern2"的所有匹配行又该如何做呢?(译者: 这是取交
集的情况, 如果取并集该怎么办呢?)
一个方法是通过 管道 来将 grep pattern1 的结果传递到 grep pattern2 中 .
例如, 给定如下文件:

```
1 # Filename: tstfile
2
3 This is a sample file.
4 This is an ordinary text file.
5 This file does not contain any unusual text.
6 This file is not unusual.
7 Here is some text.
```

现在, 让我们在这个文件中搜索既包含 "file" 又包含 "text" 的所有行

```
bash$ grep file tstfile
# Filename: tstfile
This is a sample file.
This is an ordinary text file.
This file does not contain any unusual text.
This file is not unusual.
bash$ grep file tstfile | grep text
This is an ordinary text file.
This file does not contain any unusual text.
```

--
egrep - 扩展的 grep - 这个命令与 grep -E 等价. 这个命令用起来有些不同, 由于正
则表达式扩展, 将会使得搜索更具灵活性.
fgrep - 快速的 grep - 这个命令与 grep -F 等价. 这是一种按照字符串字面意思进行
的搜索(即不允许使用正则表达式), 这样有时候会使搜索变得容易一些.
注意: 在某些linux发行版中, egrep 和 fgrep 都是 grep 命令的符号连接或者是别名,
 只不过调用的时候分别使用 -E 和 -F 选项罢了.
Example 12-16 在1913年的韦氏词典中查找定义
################################Start Script###

```
1 #!/bin/bash
2 # dict-lookup.sh
3
4 #  这个脚本在1913年的韦氏词典中查找定义.
5 #  这本公共词典可以通过不同的
6 #+ 站点来下载,包括
7 #+ Project Gutenberg (http://www.gutenberg.org/etext/247).
8 #
9 #  在通过本脚本使用之前,
10 #+ 先要将这本字典由 DOS 格式转换为 UNIX格式(只以 LF 作为行结束符).
11 #  将这个文件存储为纯文本形式, 并且保证是未压缩的 ASCII 格式.
12 #  将DEFAULT_DICTFILE 变量以 path/filename 形式设置好.
13
14
15 E_BADARGS=65
16 MAXCONTEXTLINES=50                  # 显示的最大行数.
```

```
17 DEFAULT_DICTFILE="/usr/share/dict/webster1913-dict.txt"
18                                  # 默认的路径和文件名.
19                                  # 在必要的时候可以进行修改.
20 #  注意:
21 #  -----
22 #  这个特定的1913年版的韦氏词典
23 #+ 在每个入口都是以大写字母开头的
24 #+ (剩余的字符都是小写).
25 #  只有每部分的第一行是以这种形式开始的,
26 #+ 这也就是为什么搜索算法是下边的这个样子.
27
28
29
30 if [[ -z $(echo "$1" | sed -n '/^[A-Z]/p') ]]
31 #  必须指定一个要查找的单词,
32 #+ 并且这个单词必须以大写字母开头.
33 then
34   echo "Usage: `basename $0` Word-to-define [dictionary-file]"
35   echo
36   echo "Note: Word to look up must start with capital letter,"
37   echo "with the rest of the word in lowercase."
38   echo "--------------------------------------------"
39   echo "Examples: Abandon, Dictionary, Marking, etc."
40   exit $E_BADARGS
41 fi
42
43
44 if [ -z "$2" ]                      #  也可以指定不同的词典
45                                  #+ 作为这个脚本的第2个参数传递进来.
46 then
47   dictfile=$DEFAULT_DICTFILE
48 else
49   dictfile="$2"
50 fi
51
52 # ------------------------------------------------------------
53 Definition=$(fgrep -A $MAXCONTEXTLINES "$1 \\" "$dictfile")
54 #                               以 "Word \..." 这种形式定义
55 #
56 #  当然, 即使搜索一个特别大的文本文件的时候
57 #+ "fgrep" 也是足够快的.
58
59
60 # 现在, 剪掉定义块.
61
62 echo "$Definition" |
63 sed -n '1,/^[A-Z]/p' |
64 #  从输出的第一行
```

```
65 #+ 打印到下一部分的第一行.
66 sed '$d' | sed '$d'
67 #  删除输出的最后两行Delete last two lines of output
68 #+ (空行和下一部分的第一行).
69 # ------------------------------------------------------
70
71 exit 0
72
73 # 练习:
74 # -----
75 # 1)  修改这个脚本, 让它具备能够处理任何字符形式的输入
76 #   + (大写, 小写, 或大小写混合), 然后将其转换为
77 #   + 能够处理的统一形式.
78 #
79 # 2)  将这个脚本转化为一个 GUI 应用,
80 #   + 使用一些比如像 "gdialog"的东西 . . .
81 #     这样的话, 脚本将不再从命令行中
82 #   + 取得这些参数.
83 #
84 # 3)  修改这个脚本让它具备能够分析另外一个
85 #   + 公共词典的能力,比如 U.S. Census Bureau Gazetteer.
```

##################################End Script##

agrep (近似 grep) 扩展了 grep 近似匹配的能力. 搜索的字符串可能会与最终匹配结果所找到字符串有些不同.这个工具并不是核心 Linux 发行版的一部分.

注意: 为了搜索压缩文件, 应使用 zgrep, zegrep, 或 zfgrep. 这些命令也可以对未压缩的文件进行搜索, 只不过会比一般的 grep, egrep, 和 fgrep 慢上一些. 当然, 在你要搜索的文件中如果混合了压缩和未压缩的文件的话, 那么使用这些命令是非常方便的.

如果要搜索 bzipped 类型的文件, 使用 bzgrep.

look

命令 look 与命令 grep 很相似, 但是这个命令只能做字典查询, 也就是它所搜索的文件必须已经排过序的单词列表. 默认情况下, 如果没有指定搜索那个文件, 那就默认搜索/usr/dict/words文件(译者: 感觉好像应该是/usr/share/dict/words), 当然也可以指定其他目录下的文件进行搜索.

Example 12-17 检查列表中单词的正确性

##################################Start Script##

```
1 #!/bin/bash
2 # lookup: 对指定数据文件中的每个单词都做一遍字典查询..
3
4 file=words.data  # 指定的要搜索的数据文件.
5
6 echo
7
8 while [ "$word" != end ]  # 数据文件中最后一个单词.
9 do
10   read word      # 从数据文件中读, 因为在循环的后边重定向了.
11   look $word > /dev/null  # 不想将字典文件中的行显示出来.
12   lookup=$?      #  'look' 命令的退出状态.
```

```
13
14  if [ "$lookup" -eq 0 ]
15  then
16    echo "\"$word\" is valid."
17  else
18    echo "\"$word\" is invalid."
19  fi
20
21 done <"$file"    # 将 stdin 重定向到 $file, 所以 "reads" 来自于 $file.
22
23 echo
24
25 exit 0
26
27 # ----------------------------------------------------
28 # 下边的代码行将不会执行, 因为上边已经有 "exit"命令了.
29
30
31 # Stephane Chazelas 建议使用下边更简洁的方法:
32
33 while read word && [[ $word != end ]]
34 do if look "$word" > /dev/null
35    then echo "\"$word\" is valid."
36    else echo "\"$word\" is invalid."
37    fi
38 done <"$file"
39
40 exit 0
```

##################################End Script##

sed, awk

这个两个命令都是独立的脚本语言, 尤其适合分析文本文件和命令输出. 既可以单独使用, 也可以结合管道和在shell脚本中使用.

sed

非交互式的 "流编辑器", 在批量模式下, 允许使用许多 ex 命令.你会发现它在shell脚本中非常有用.

awk

可编程的文件提取器和文件格式化工具, 在结构化的文本文件中,处理或提取特定域(特定列)具有非常好的表现.它的语法与 C 语言很类似.

wc

wc 可以统计文件或 I/O 流中的单词数量.

bash $ wc /usr/share/doc/sed-4.1.2/README

13 70 447 README

[13 lines 70 words 447 characters]

wc -w 统计单词数量.

wc -l 统计行数量.

wc -c 统计字节数量.

wc -m 统计字符数量.

wc -L 给出文件中最长行的长度.

使用 wc 命令来统计当前工作目录下有多少个 .txt 文件.

```
1 $ ls *.txt | wc -l
2 # 因为列出的文件名都是以换行符区分的,所以使用 -l 来统计.
3
4 # 另一种达到这个目的的方法:
5 #      find . -maxdepth 1 -name \*.txt -print0 | grep -cz .
6 #      (shopt -s nullglob; set -- *.txt; echo $#)
7
8 # Thanks, S.C.
```

使用 wc 命令来统计所有以 d - h 开头的文件的大小.

```
bash$ wc [d-h]* | grep total | awk '{print $3}'
71832
```

使用 wc 命令来查看指定文件中包含 "Linux" 的行一共有多少.

```
bash$ grep Linux abs-book.sgml | wc -l
50
```

参见 Example 12-35 和 Example 16-8.

某些命令的某些选项其实已经包含了 wc 命令的部分功能.

```
1 ... | grep foo | wc -l
2 # 这个命令使用得非常频繁, 但事实上它有更简便的写法.
3
4 ... | grep -c foo
5 # 只要使用 grep 命令的 "-c" (或 "--count")选项就能达到同样的目的.
6
7 # Thanks, S.C.
```

tr

字符转换过滤器.

注意: 必须使用引用或中括号, 这样做才是合理的. 引用可以阻止 shell 重新解释出现在 tr 命令序列中的特殊字符.中括号应该被引用起来防止被shell扩展.

无论 tr "A-Z" "*" <filename 还是 tr A-Z * <filename 都可以将 filename 中的大写字符修改为星号(写到 stdout).但是在某些系统上可能就不能正常工作了, 而 tr A-Z '[**]' 在任何系统上都可以正常工作.

-d 选项删除指定范围的字符.

```
1 echo "abcdef"                # abcdef
2 echo "abcdef" | tr -d b-d     # aef
3
4
5 tr -d 0-9 <filename
6 # 删除 "filename" 中所有的数字.
```

--squeeze-repeats (或 -s) 选项用来在重复字符序列中除去除第一个字符以外的所有字符. 这个选项在删除多余的whitespace 的时候非常有用.

```
bash$ echo "XXXXX" | tr --squeeze-repeats 'X'
X
```

-c "complement" 选项将会 反转 匹配的字符集. 通过这个选项, tr 将只会对那些 不匹配的字符起作用.

```
bash$ echo "acfdeb123" | tr -c b-d +
+c+d+b++++
```

注意 tr 命令支持 POSIX 字符类. [1]

```
bash$ echo "abcd2ef1" | tr '[:alpha:]' -
```

----2--1

Example 12-18 转换大写: 把一个文件的内容全部转换为大写.

################################Start Script###

```
 1 #!/bin/bash
 2 # 把一个文件的内容全部转换为大写.
 3
 4 E_BADARGS=65
 5
 6 if [ -z "$1" ]  # 检查命令行参数.
 7 then
 8   echo "Usage: `basename $0` filename"
 9   exit $E_BADARGS
10 fi
11
12 tr a-z A-Z <"$1"
13
14 # 与上边的作用相同, 但是使用了 POSIX 字符集标记方法:
15 #       tr '[:lower:]' '[:upper:]' <"$1"
16 # Thanks, S.C.
17
18 exit 0
19
20 #  练习:
21 #  重写这个脚本, 通过选项可以控制脚本或者
22 #+ 转换为大写或者转换为小写.
```

################################End Script###

Example 12-19 转换小写: 将当前目录下的所有文全部转换为小写.

################################Start Script###

```
 1 #!/bin/bash
 2 #
 3 #  将当前目录下的所有文全部转换为小写.
 4 #
 5 #  灵感来自于 John Dubois 的脚本,
 6 #+ 转换为 Bash 脚本,
 7 #+ 然后被本书作者精简了一下.
 8
 9
10 for filename in *               # 遍历当前目录下的所有文件.
11 do
12    fname=`basename $filename`
13    n=`echo $fname | tr A-Z a-z`  # 将名字修改为小写.
14    if [ "$fname" != "$n" ]      # 只对那些文件名不是小写的文件进行重命名.
15    then
16      mv $fname $n
17    fi
18 done
19
20 exit $?
```

```
21
22
23 # 下边的代码将不会被执行, 因为上边的 "exit".
24 #---------------------------------------#
25 # 删除上边的内容,来运行下边的内容.
26
27 # 对于那些文件名中包含空白和新行的文件, 上边的脚本就不能工作了.
28 # Stephane Chazelas 因此建议使用下边的方法:
29
30
31 for filename in *    # 不必非得使用 basename 命令,
32                      # 因为 "*" 不会返回任何包含 "/" 的文件.
33 do n=`echo "$filename/" | tr '[:upper:]' '[:lower:]'`
34 #                          POSIX 字符集标记法.
35 #                  添加的斜线是为了在文件名结尾换行不会被
36 #                  命令替换删掉.
37   # 变量替换:
38   n=${n%/}         # 从文件名中将上边添加在结尾的斜线删除掉.
39   [[ $filename == $n ]] || mv "$filename" "$n"
40                     # 检查文件名是否已经是小写.
41 done
42
43 exit $?
```

################################End Script###

Example 12-20 Du: DOS 到 UNIX 文本文件的转换.

################################Start Script###

```
 1 #!/bin/bash
 2 # Du.sh: DOS 到 UNIX 文本文件的转换.
 3
 4 E_WRONGARGS=65
 5
 6 if [ -z "$1" ]
 7 then
 8   echo "Usage: `basename $0` filename-to-convert"
 9   exit $E_WRONGARGS
10 fi
11
12 NEWFILENAME=$1.unx
13
14 CR='\015'  # 回车Carriage return.
15          # 015 是 8 进制的 ASCII 码的回车.
16          # DOS 中文本文件的行结束符是 CR-LF.
17          # UNIX 中文本文件的行结束符只是 LF.
18
19 tr -d $CR < $1 > $NEWFILENAME
20 # 删除回车并且写到新文件中.
21
22 echo "Original DOS text file is \"$1\"."
```

```
23 echo "Converted UNIX text file is \"$NEWFILENAME\"."
24
25 exit 0
26
27 # 练习:
28 # -----
29 # 修改上边的脚本完成从UNIX 到 DOS 的转换.
```

##################################End Script##

Example 12-21 rot13: rot13, 弱智加密.

##################################Start Script##

```
 1 #!/bin/bash
 2 # rot13.sh: 典型的 rot13 算法,
 3 #           使用这种方法加密可能可以愚弄一下3岁小孩.
 4
 5 # 用法: ./rot13.sh filename
 6 # 或    ./rot13.sh <filename
 7 # 或    ./rot13.sh and supply keyboard input (stdin)
 8
 9 cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M'   # "a" 变为 "n", "b" 变为 "o", 等等.
10 #  'cat "$@"' 结构
11 #+ 允许从stdin或者从文件中获得输入.
12
13 exit 0
```

##################################End Script##

Example 12-22 Generating "Crypto-Quote" Puzzles

##################################Start Script##

```
 1 #!/bin/bash
 2 # crypto-quote.sh: 加密
 3
 4 #  使用单码替换(单一字母替换法)来进行加密.
 5 #  The result is similar to the "Crypto Quote" puzzles
 6 #+ seen in the Op Ed pages of the Sunday paper. <rojy bug>(不太了解这句的内容, 应该是有特定的含义)
 7
 8
 9 key=ETAOINSHRDLUBCFGJMQPVWZYXK
10 # "key" 不过是一个乱序的字母表.
11 # 修改 "key" 就会修改加密的结果.
12
13 # The 'cat "$@"' construction gets input either from stdin or from files.
14 # 如果使用stdin, 那么要想结束输入就使用 Control-D.
15 # 否则就要在命令行上指定文件名.
16
17 cat "$@" | tr "a-z" "A-Z" | tr "A-Z" "$key"
18 #       |  转化为大写  |    加密
19 # 小写, 大写, 或混合大小写, 都可以正常工作.
20 # 但是传递进来的非字母字符将不会起任何变化.
21
22
```

```
23 # 用下边的语句试试这个脚本:
24 # "Nothing so needs reforming as other people's habits."
25 # --Mark Twain
26 #
27 # 输出为:
28 # "CFPHRCS QF CIIOQ MINFMBRCS EQ FPHIM GIFGUI'Q HETRPQ."
29 # --BEML PZERC
30
31 # 解密:
32 # cat "$@" | tr "$key" "A-Z"
33
34
35 #  这个简单的密码可以轻易的被一个12岁的小孩
36 #+ 用铅笔和纸破解.
37
38 exit 0
39
40 #  练习:
41 #  -----
42 #  修改这个脚本, 让它可以用命令行参数
43 #+ 来决定加密或解密.
```

################################End Script###

注意: tr 的不同版本

tr 工具在历史上有2个重要版本. BSD 版本不需要使用中括号 (tr a-z A-Z), 但是 SysV 版本则需要中括号 (tr '[a-z]' '[A-Z]'). GNU 版本的 tr 命令与 BSD 版本比较相像, 所以使用中括号来引用字符范围是强制性的(译者: 感觉这句说反了, 读者可自行参照原文).

fold

将输入按照指定宽度进行折行. 这里有一个非常有用的选项 -s ,这个选项可以使用空格进行断行.(译者: 事实上只有外文才需要使用空格断行, 中文是不需要的) (参见 Example 12-23 和 Example A-1).

fmt

一个简单的文件格式器, 通常用在管道中, 将一个比较长的文本行输出进行折行.

Example 12-23 格式化文件列表.

```
################################Start Script#######################################
 1 #!/bin/bash
 2
 3 WIDTH=40                 # 设为 40 列宽.
 4
 5 b=`ls /usr/local/bin`      # 取得文件列表...
 6
 7 echo $b | fmt -w $WIDTH
 8
 9 # 也可以使用如下方法,作用相同
10 #    echo $b | fold - -s -w $WIDTH
11
12 exit 0
```

################################End Script###

参见 Example 12-5.
注意: 如果想找到一个更强力的 fmt 工具可以选择 Kamil Toman 的 par 工具, 这个工具可以从后边的这个网址取得http://www.cs.berkeley.edu/~amc/Par/.

col

这个命令用来滤除标准输入的反向换行符号. 这个工具还可以将空白用等价的 tab 来替换. col 工具最主要的应用还是从特定的文本处理工具中过滤输出, 比如 groff 和 tbl.
(译者: 主要用来将man页转化为文本)

column

列格式化工具. 这个过滤工具将会将列类型的文本转化为"易于打印"的表格式进行输出, 通过在合适的位置插入tab.

Example 12-24 使用 column 来格式化目录列表

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # 这是"column" man页中的一个例子, 作者对这个例子做了很小的修改.
 3
 4
 5 (printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \
 6 ; ls -l | sed 1d) | column -t
 7
 8 #  管道中的 "sed 1d" 删除输出的第一行,
 9 #+ 第一行将是 "total       N",
10 #+ 其中 "N" 是 "ls -l" 找到的文件总数.
11
12 # "column" 中的 -t 选项用来转化为易于打印的表形式.
13
14 exit 0
################################End Script###########################################
```

colrm

列删除过滤器. 这个工具将会从文件中删除指定的列(列中的字符串)并且写到文件中, 如果指定的列不存在,那么就回到 stdout. colrm 2 4 <filename 将会在filename文件中对每行删除第2到第4列之间的所有字符.
注意: 如果这个文件包含tab和不可打印字符, 那将会引起不可预期的行为. 在这种情况下, 应该通过管道的手段使用 expand 和 unexpand 命令来预处理 colrm.

nl

计算行号过滤器. nl filename 将会在 stdout 中列出文件的所有内容, 但是会在每个非空行的前面加上连续的行号. 如果没有 filename 参数, 那么就操作 stdin.
nl 命令的输出与 cat -n 非常相似, 然而, 默认情况下 nl 不会列出空行.

Example 12-25 nl: 一个自己计算行号的脚本.

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # line-number.sh
 3
 4 # 这个脚本将会 echo 自身两次, 并显示行号.
 5
 6 # 'nl' 命令显示的时候你将会看到, 本行是第4行, 因为它不计空行.
 7 # 'cat -n' 命令显示的时候你将会看到, 本行是第6行.
 8
 9 nl `basename $0`
```

```
10
11 echo; echo  # 下边, 让我们试试 'cat -n'
12
13 cat -n `basename $0`
14 # 区别就是 'cat -n' 对空行也进行计数.
15 # 注意 'nl -ba' 也会这么做.
16
17 exit 0
18 # ------------------------------------------------------------
```

################################End Script###

pr

格式化打印过滤器. 这个命令会将文件(或stdout)分页, 将它们分成合适的小块以便于硬拷贝打印或者在屏幕上浏览.使用这个命令的不同的参数可以完成好多任务, 比如对行和列的操作,加入行, 设置页边, 计算行号, 添加页眉, 合并文件等等. pr 命令集合了许多命令的功能, 比如 nl, paste, fold, column, 和 expand.

pr -o 5 --width=65 fileZZZ | more 这个命令对fileZZZ进行了比较好的分页,并且打印到屏幕上.文件的缩进被设置为5, 总宽度设置为65.

一个特定的使用选项 -d, 强制隔行打印 (与 sed -G 效果相同).

gettext

GNU gettext 包是专门用来将程序的输出翻译或者本地化为不同国家语言的工具集.在最开始的时候仅仅支持C 语言, 现在已经支持了相当数量的其它程序语言和脚本语言.

要想查看 gettext 程序 如何在shell脚本中工作. 参见 info 页.

msgfmt

一个产生2进制消息目录的程序. 这个命令主要用来 本地化.

iconv

一个可以将文件转化为不同编码格式(字符集)的工具. 这个命令主要用来 本地化.

```
 1 # 将字符符串由 UTF-8 格式转换为 UTF-16 并且打印到 BookList 中
 2 function write_utf8_string {
 3     STRING=$1
 4     BOOKLIST=$2
 5     echo -n "$STRING" | iconv -f UTF8 -t UTF16 | cut -b 3- | tr -d \\n >> "$BOOKLIST"
 6 }
 7
 8 #  来自于 Peter Knowles' "booklistgen.sh" 脚本
 9 #+ 目的是把文件转换为 Sony Librie 格式.
10 #  (http://booklistgensh.peterknowles.com)
```

recode

可以认为这个命令时上边 iconv 命令的一个空想家版本. 这个非常灵活的并可以把整个文件都转换为不同编码格式的工具并不是Linux 标准安装的一部分.

TeX, gs

TeX 和 Postscript 都是文本标记语言, 用来对打印和格式化的视频显示进行预拷贝.

TeX 是 Donald Knuth 精心制作的排版系统. 通常情况下, 通过编写脚本的手段来把所有的选项和参数封装起来一起传到标记语言中是一件很方便的事情.

Ghostscript (gs) 是一个 遵循 GPL 的Postscript 解释器.

enscript

将纯文本文件转换为 PostScript 的工具

比如, enscript filename.txt -p filename.ps 产生一个 PostScript 输出文件 filename.ps.

groff, tbl, eqn
另一种文本标记和显示格式化语言是 groff. 这是一个对传统 UNIX roff/troff 显示和排版包的 GNU 增强版本.Man页 使用的就是 groff.
tbl 表处理工具可以认为是 groff 的一部分, 它的功能就是将表标记转化到 groff 命令中.
eqn 等式处理工具也是 groff 的一部分, 它的功能是将等式标记转化到 groff 命令中.
Example 12-26 manview: 查看格式化的man页
################################Start Script###
```
 1 #!/bin/bash
 2 # manview.sh: 将man页源文件格式化以方便查看.
 3
 4 #  当你想阅读man页的时候, 这个脚本就有用了.
 5 #  它允许你在运行的时候查看
 6 #+ 中间结果.
 7
 8 E_WRONGARGS=65
 9
10 if [ -z "$1" ]
11 then
12   echo "Usage: `basename $0` filename"
13   exit $E_WRONGARGS
14 fi
15
16 # ---------------------------
17 groff -Tascii -man $1 | less
18 # 来自于 groff man页.
19 # ---------------------------
20
21 #  如果man业中包括表或者等式,
22 #+ 那么上边的代码就够呛了.
23 #  下边的这行代码可以解决上边的这个问题.
24 #
25 #   gtbl < "$1" | geqn -Tlatin1 | groff -Tlatin1 -mtty-char -man
26 #
27 #   Thanks, S.C.
28
29 exit 0
```
################################End Script###
lex, yacc
lex 是用于模式匹配的词汇分析产生程序. 在Linux系统上这个命令已经被 flex 取代了.
yacc 工具基于一系列的语法规范生成语法分析程序. 在Linux系统上这个命令已经被 bison 取代了.
注意事项:
[1] 对于 GNU 版本的 tr 命令来说这是唯一一处比那些商业 UNIX 系统上的一般版本合适的地方.
12.5 文件与归档命令

归档命令

tar

标准的 UNIX 归档工具. [1] 起初这只是一个 磁带 归档 程序, 而现在这个工具已经被开发为通用打包程序, 它能够处理所有设备的所有类型的归档文件, 包括磁带设备, 正常文件, 甚至是 stdout (参见Example 3-4). GNU 的tar工具现在可以接受不同种类的压缩过滤器, 比如tar czvf archive_name.tar.gz *, 并且可以递归的处理归档文件, 还可以用 gzip 压缩目录下的所有文件, 除了当前目录下($PWD)的 点文件 . [2]

一些有用的 tar 命令选项:

1. -c 创建 (一个新的归档文件)
2. -x 解压文件 (从存在的归档文件中)
3. --delete 删除文件 (从存在的归档文件中)

注意: 这个选项不能用于磁带类型设备.

4. -r 将文件添加到现存的归档文件的尾部
5. -A 将 tar 文件添加到现存的归档文件的尾部
6. -t 列出现存的归档文件中包含的内容
7. -u 更新归档文件
8. -d 使用指定的文件系统 比较归档文件
9. -z 用 gzip 压缩归档文件

(压缩还是解压, 依赖于是否组合了 -c 或 -x)选项

10. -j 用 bzip2 压缩归档文件

注意: 如果想从损坏的用 gzip 压缩过的 tar 文件中取得数据, 那将是很困难的. 所有当我们归档重要的文件的时候, 一定要保留多个备份.

shar

Shell 归档工具. 存在于 shell 归档文件中的所有文件都是未经压缩的, 并且本质上是一个shell 脚本,以 #!/bin/sh 开头, 并且包含所有必要的解档命令. Shar 归档文件 至今还在 Internet 新闻组中使用, 否则的话 shar早就被 tar/gzip 所取代了. unshar 命令用来解档 shar 归档文件.

ar

创建和操作归档文件的工具, 主要在对2进制目标文件打包成库时才会用到.

rpm

Red Hat 包管理器, 或者说 rpm 工具提供了一种对源文件或2进制文件进行打包的方法. 除此之外, 它还包括安装命令, 并且还检查包的完整性.

一个简单的 rpm -i package_name.rpm 命令对于安装一个包来说就足够了, 虽然这个命令还有好多其它的选项.

注意: rpm -qf 列出一个文件属于那个包.

```
bash$ rpm -qf /bin/ls
coreutils-5.2.1-3
```

注意: rpm -qa 将会列出给定系统上所有安装了的 rpm 包. rpm -qa package_name 命令将会列出于给定名字匹配的包.

```
bash$ rpm -qa
redhat-logos-1.1.3-1
glibc-2.2.4-13
cracklib-2.7-12
dosfstools-2.7-1
gdbm-1.8.0-10
ksymoops-2.4.1-1
mktemp-1.5-11
perl-5.6.0-17
reiserfs-utils-3.x.0j-2
```

...

bash$ rpm -qa docbook-utils
docbook-utils-0.6.9-2

bash$ rpm -qa docbook | grep docbook
docbook-dtd31-sgml-1.0-10
docbook-style-dsssl-1.64-3
docbook-dtd30-sgml-1.0-10
docbook-dtd40-sgml-1.0-11
docbook-utils-pdf-0.6.9-2
docbook-dtd41-sgml-1.0-10
docbook-utils-0.6.9-2

cpio

这个特殊的归档拷贝命令(拷贝输入和输出)现在已经很少能见到了, 因为它已经被 tar/gz ip 所替代了.现在这个命令只在一些比较特殊的地方还在使用,比如拷贝一个目录树.

Example 12-27 使用 cpio 来拷贝一个目录树

```
###################################Start Script#########################################
 1 #!/bin/bash
 2
 3 # 使用 'cpio' 拷贝目录树.
 4
 5 # 使用 'cpio' 的优点:
 6 #   加速拷贝. 比通过管道使用 'tar' 命令快一些.
 7 #   很适合拷贝一些 'cp' 命令
 8 #+  搞不定的的特殊文件(比如名字叫 pipes 的文件, 等等)
 9
10 ARGS=2
11 E_BADARGS=65
12
13 if [ $# -ne "$ARGS" ]
14 then
15   echo "Usage: `basename $0` source destination"
16   exit $E_BADARGS
17 fi
18
19 source=$1
20 destination=$2
21
22 find "$source" -depth | cpio -admvp "$destination"
23 #              ^^^^^         ^^^^^
24 # 阅读 'find' 和 'cpio' 的man 页来了解这些选项的意义.
25
26
27 # 练习:
28 # -----
```

```
29
30 #  添加一些代码来检查 'find | cpio' 管道命令的退出码($?)
31 #+ 并且如果出现错误的时候输出合适的错误码.
32
33 exit 0
```

################################End Script###

rpm2cpio

这个命令可以从 rpm 归档文件中解出一个 cpio 归档文件.

Example 12-28 解包一个 rpm 归档文件

################################Start Script###

```
 1 #!/bin/bash
 2 # de-rpm.sh: 解包一个 'rpm' 归档文件
 3
 4 : ${1?"Usage: `basename $0` target-file"}
 5 # 必须指定 'rpm' 归档文件名作为参数.
 6
 7
 8 TEMPFILE=$$.cpio                  # Tempfile 必须是一个"唯一"的名字.
 9                                   # $$ 是这个脚本的进程 ID.
10
11 rpm2cpio < $1 > $TEMPFILE         # 将 rpm 归档文件转换为 cpio 归档文件.
12 cpio --make-directories -F $TEMPFILE -i  # 解包 cpio 归档文件.
13 rm -f $TEMPFILE                   # 删除 cpio 归档文件.
14
15 exit 0
16
17 #  练习:
18 #  添加一些代码来检查    1) "target-file" 是否存在
19 #+                      2) 这个文件是否是一个 rpm 归档文件.
20 #  暗示:                    分析 'file' 命令的输出.
```

################################End Script###

压缩命令

gzip

标准的 GNU/UNIX 压缩工具, 取代了比较差的 compress 命令. 相应的解压命令是gunzip, gzip -d 是等价的.

zcat 过滤器可以将一个 gzip 文件解压到 stdout, 所以尽可能的使用管道和重定向. 这个命令事实上就是一个可以工作于压缩文件(包括一些的使用老的 compress 工具压缩的文件)的 cat 命令. zcat 命令等价于 gzip -dc.

注意: 在某些商业的 UNIX 系统上, zcat 与 uncompress -c 等价, 并且不能工作于 gzip 文件.

参见 Example 7-7.

bzip2

用来压缩的一个可选的工具, 通常比 gzip 命令压缩率更高(所以更慢), 适用于比较大的文件. 相应的解压命令是 bunzip2.

注意: 新版本的 tar 命令已经直接支持 bzip2 了.

compress, uncompress

这是一个老的, 私有的压缩工具, 一般的商业 UNIX 发行版都会有这个工具. 更有效率的 gzip 工具早就把这个工具替换掉了. Linux 发行版一般也会包含一个兼容的 compress 命

令, 虽然 gunzip 也可以加压用 compress 工具压缩的文件.
注意: znew 命令可以将 compress 压缩的文件转换为 gzip 压缩的文件.

sq

另一种压缩工具, 一个只能工作于排过序的 ASCII 单词列表的过滤器.这个命令使用过滤器标准的调用语法, sq < input-file > output-file. 速度很快, 但是效率远不及 gzip. 相应的解压命令为 unsq, 调用方法与 sq 相同.
注意: sq 的输出可以通过管道传递给 gzip 以便于进一步的压缩.

zip, unzip

跨平台的文件归档和压缩工具, 与 DOS 下的 pkzip.exe 兼容. zip 归档文件看起来在互联网上比 tar 包更流行.

unarc, unarj, unrar

这些 Linux 工具可以用来解档那些用 DOS 下的 arc.exe, arj.exe, 和 rar.exe 程序进行归档的文件.

文件信息

file

确定文件类型的工具. 命令 file file-name 将会用 ascii 文本或数据的形式返回 file-name 文件的详细描述. 这个命令会使用 /usr/share/magic, /etc/magic, 或 /usr/lib/magic 中定义的 魔法数字 来标识包含某种魔法数字的文件, 上边所举出的这3个文件需要依赖于具体的 Linux/UNIX 发行版.
-f 选项将会让 file 命令运行于批处理模式, 也就是说它会分析 -f 后边所指定的文件, 从中读取需要处理的文件列表, 然后依次执行 file 命令. -z 选项, 当对压缩过的目标文件使用时, 将会强制分析压缩的文件类型.

```
bash$ file test.tar.gz
test.tar.gz: gzip compressed data, deflated, last modified: Sun Sep 16 13:34:51 2001, os: Unix

bash file -z test.tar.gz
test.tar.gz: GNU tar archive (gzip compressed data, deflated, last modified: Sun Sep 16 13:34:51 2001, os: Unix)
```

```
1 # 在给定的目录中找出sh和Bash脚本文件:
2
3 DIRECTORY=/usr/local/bin
4 KEYWORD=Bourne
5 # Bourne 和 Bourne-Again shell 脚本
6
7 file $DIRECTORY/* | fgrep $KEYWORD
8
9 # 输出:
10
11 # /usr/local/bin/burn-cd:         Bourne-Again shell script text executable
12 # /usr/local/bin/burnit:          Bourne-Again shell script text executable
13 # /usr/local/bin/cassette.sh:     Bourne shell script text executable
14 # /usr/local/bin/copy-cd:         Bourne-Again shell script text executable
15 # . . .
```

Example 12-29 从 C 文件中去掉注释

```
##################################Start Script#########################################
1 #!/bin/bash
2 # strip-comment.sh: 去掉C 程序中的注释 (/* 注释 */)
3
4 E_NOARGS=0
```

```
E_ARGERROR=66
E_WRONG_FILE_TYPE=67

if [ $# -eq "$E_NOARGS" ]
then
  echo "Usage: `basename $0` C-program-file" >&2 # 将错误消息发到 stderr.
  exit $E_ARGERROR
fi

# 检查文件类型是否正确.
type=`file $1 | awk '{ print $2, $3, $4, $5 }'`
# "file $1" echoe 出文件类型 . . .
# 然后 awk 会删掉第一个域,  就是文件名 . . .
# 然后结果将会传递到变量 "type" 中.
correct_type="ASCII C program text"

if [ "$type" != "$correct_type" ]
then
  echo
  echo "This script works on C program files only."
  echo
  exit $E_WRONG_FILE_TYPE
fi


# 相当隐秘的 sed 脚本:
#--------
sed '
/^\/\*/d
/.*\*\//d
' $1
#--------
# 如果你花上几个小时来学习 sed 语法的话, 上边这个命令还是很好理解的.


#  如果注释和代码在同一行上, 上边的脚本就不行了.
#+ 所以需要添加一些代码来处理这种情况.
#  这是一个很重要的练习.

#  当然, 上边的代码也会删除带有 "*/" 的非注释行 --
#+ 这也不是一个令人满意的结果.

exit 0


# ------------------------------------------------------------------
# 下边的代码不会执行, 因为上边已经 'exit 0' 了.

```

```
53 # Stephane Chazelas 建议使用下边的方法:
54
55 usage() {
56   echo "Usage: `basename $0` C-program-file" >&2
57   exit 1
58 }
59
60 WEIRD=`echo -n -e '\377'`    # or WEIRD=$'\377'
61 [[ $# -eq 1 ]] || usage
62 case `file "$1"` in
63   *"C program text"*) sed -e "s%/\*%${WEIRD}%g;s%\*/%${WEIRD}%g" "$1" \
64      | tr '\377\n' '\n\377' \
65      | sed -ne 'p;n' \
66      | tr -d '\n' | tr '\377' '\n';;
67   *) usage;;
68 esac
69
70 #  如果是下列的这些情况, 还是很糟糕:
71 #  printf("/*");
72 #  or
73 #  /*  /* buggy embedded comment */
74 #
75 #  为了处理上边所有这些特殊情况(字符串中的注释, 含有 \", \\" ...
76 #+ 的字符串中的注释) 唯一的方法还是写一个 C 分析器
77 #+ (或许可以使用lex 或者 yacc ?).
78
79 exit 0
```

################################End Script###

which

which command-xxx 将会给出 "command-xxx" 的完整路径. 当你想在系统中准确定位一个
特定的命令或工具的时候, 这个命令就非常有用了.

```
$bash which rm
 /usr/bin/rm
```

whereis

与上边的 which 很相似, whereis command-xxx 不只会给出 "command-xxx" 的完整路径,
而且还会给出这个命令的 man页 的完整路径.

```
$bash whereis rm
 rm: /bin/rm /usr/share/man/man1/rm.1.bz2
```

whatis

whatis filexxx 将会在 whatis 数据库中查询 "filexxx". 当你想确认系统命令和重要的
配置文件的时候, 这个命令就非常重要了. 可以把这个命令认为是一个简单的 man 命令.

```
$bash whatis whatis
 whatis              (1)  - search the whatis database for complete words
```

Example 12-30 Exploring /usr/X11R6/bin

################################Start Script###

```
1 #!/bin/bash
2
3 # 在 /usr/X11R6/bin 中的所有神秘的2进制文件都是什么东西?
```

```
 4
 5 DIRECTORY="/usr/X11R6/bin"
 6 # 也试试 "/bin", "/usr/bin", "/usr/local/bin", 等等.
 7
 8 for file in $DIRECTORY/*
 9 do
10   whatis `basename $file`  # 将会 echo 出这个2进制文件的信息.
11 done
12
13 exit 0
14
15 # 你可能希望将这个脚本的输出重定向, 像这样:
16 # ./what.sh >>whatis.db
17 # 或者一页一页的在 stdout 上查看,
18 # ./what.sh | less
```

##################################End Script###

参见 Example 10-3.

vdir

显示详细的目录列表. 与 ls -l 的效果类似.

这是一个 GNU fileutils.

```
 bash$ vdir
 total 10
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.xrolo
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.xrolo.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.xrolo

 bash ls -l
 total 10
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.xrolo
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.xrolo.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.xrolo
```

locate, slocate

locate 命令将会在预先建立好的档案数据库中查询文件. slocate 命令是 locate 的安全版本(locate 命令可能已经被关联到 slocate 命令上了).

```
$bash locate hickson
 /usr/lib/xephem/catalogs/hickson.edb
```

readlink

显示符号连接所指向的文件.

```
 bash$ readlink /usr/bin/awk
 ../../bin/gawk
```

strings

使用 strings 命令在二进制或数据文件中找出可打印字符. 它将在目标文件中列出所有找到的可打印字符的序列. 这个命令对于想进行快速查找一个 n 个字符的打印检查来说是很方便的,也可以用来检查一个未知格式的图片文件 (strings image-file | more 可能会搜索出像 JFIF 这样的字符串, 那么这就意味着这个文件是一个 jpeg 格式的图片文件).

在脚本中, 你可能会使用 grep 或 sed 命令来分析 strings 命令的输出. 参见 Example 10-7 和 Example 10-9.

Example 12-31 一个"改进过"的 strings 命令

################################Start Script###

```
 1 #!/bin/bash
 2 # wstrings.sh: "word-strings" (增强的 "strings" 命令)
 3 #
 4 #  这个脚本将会过滤 "strings" 命令的输出.
 5 #+ 通过排除标准单词列表的形式检查来过滤输出.
 6 #  这将有效的过滤掉无意义的字符,
 7 #+ 并且指挥输出可以识别的字符.
 8
 9 # ============================================================
10 #               脚本参数的标准检查
11 ARGS=1
12 E_BADARGS=65
13 E_NOFILE=66
14
15 if [ $# -ne $ARGS ]
16 then
17   echo "Usage: `basename $0` filename"
18   exit $E_BADARGS
19 fi
20
21 if [ ! -f "$1" ]                    # 检查文件是否存在.
22 then
23    echo "File \"$1\" does not exist."
24    exit $E_NOFILE
25 fi
26 # ============================================================
27
28
29 MINSTRLEN=3                         #  最小的字符串长度.
30 WORDFILE=/usr/share/dict/linux.words  #  字典文件.
31                                     #  也可以指定一个不同的
32                                     #+ 单词列表文件,
33                                     #+ 但这种文件必须是以每个单词一行的方式进行保存.
34
35
36 wlist=`strings "$1" | tr A-Z a-z | tr '[:space:]' Z | \
37 tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`
38
39 # 将'strings' 命令的输出通过管道传递到多个 'tr' 命令中.
40 #  "tr A-Z a-z"  全部转换为小写字符.
41 #  "tr '[:space:]'"  转换空白字符为多个 Z.
42 #  "tr -cs '[:alpha:]' Z"  将非字母表字符转换为多个 Z,
43 #+ 然后去除多个连续的 Z.
44 #  "tr -s '\173-\377' Z"  把所有z后边的字符都转换为 Z.
45 #+ 并且去除多余重复的Z.(注意173(123 ascii "{")和377(255 ascii 最后一个字符)都是8进制)
46 #+ 这样处理之后, 我们所有之前需要处理的令我们头痛的字符
47 #+ 就全都转换为字符 Z 了.
```

```
48 #  最后"tr Z ' '" 将把所有的 Z 都转换为空格,
49 #+ 这样我们在下边循环中用到的变量 wlist 中的内容就全部以空格分隔了.
50
51 #  ***************************************************************
52 #  注意, 我们使用管道来将多个 'tr' 的输出传递到下一个 'tr' 时
53 #+ 每次都使用了不同的参数.
54 #  ***************************************************************
55
56
57 for word in $wlist                  # 重要:
58                                     # $wlist 这里不能使用双引号.
59                                     # "$wlist" 不能正常工作.
60                                     # 为什么不行?
61 do
62
63   strlen=${#word}                   # 字符串长度.
64   if [ "$strlen" -lt "$MINSTRLEN" ]   # 跳过短的字符串.
65   then
66     continue
67   fi
68
69   grep -Fw $word "$WORDFILE"         #  只匹配整个单词.
70 #      ^^^                          #  "固定字符串" 和
71                                     #+ "整个单词" 选项.
72
73 done
74
75
76 exit $?
```

################################End Script###

比较命令

diff, patch

diff: 一个非常灵活的文件比较工具. 这个工具将会以一行接一行的形式来比较目标文件. 在某些应用中, 比如说比较单词词典, 在通过管道将结果传递给 diff 命令之前, 使用诸如 sort 和 uniq 命令来对文件进行过滤将是非常有用的.diff file-1 file-2 将会输出2个文件不同的行,并会通过符号标识出每个不同行所属的文件.

diff 命令的 --side-by-side 选项将会把2个比较中的文件全部输出, 按照左右分隔的形式, 并会把不同的行标记出来. -c 和 -u 选项也会使得 diff 命令的输出变得容易解释一些.

还有一些 diff 命令的变种, 比如 sdiff, wdiff, xdiff, 和 mgdiff.

注意: 如果比较的两个文件是完全一样的话, 那么 diff 命令会返回 0 作为退出码, 如果不同的话就返回 1 作为退出码. 这样 diff 命令就可以用在 shell 脚本的测试结构中了. (见下边)

diff 命令的一个重要用法就是产生区别文件, 这个文件将用作 patch 命令的 -e 选项的参数, -e 选项接受 ed 或 ex 脚本.

patch: 灵活的版本工具.给出一个用 diff 命令产生的区别文件, patch 命令可以将一个老版本的包更新为一个新版本的包. 因为你发布一个小的区别文件远比重新发布一个大的软件包来的容易得多.对于频繁更新的 Linux 内核来说, 使用补丁包的形式来发布将是一

种很好的方法.

```
1 patch -p1 <patch-file
2 # 在'patch-file'中取得所有的修改列表
3 # 然后把它们应用于其中索引到的文件上.
4 # 那么这个包就被更新为新版本了.
```

更新 kernel:

```
1 cd /usr/src
2 gzip -cd patchXX.gz | patch -p0
3 #  使用'patch'来更新内核源文件.
4 # 来自于匿名作者(Alan Cox?)的
5 # Linux 内核文档 "README".
```

注意: diff 命令也可以递归的比较目录下的所有文件(包含子目录).

```
bash$ diff -r ~/notes1 ~/notes2
Only in /home/bozo/notes1: file02
Only in /home/bozo/notes1: file03
Only in /home/bozo/notes2: file04
```

注意: 使用 zdiff 来比较 gzip 文件.

diff3

一个 diff 命令的扩展版本, 可以同时比较3个文件. 如果成功执行那么这个命令就返回0, 但是不幸的是这个命令不给出比较结果的信息.

```
bash$ diff3 file-1 file-2 file-3
====
1:1c
  This is line 1 of "file-1".
2:1c
  This is line 1 of "file-2".
3:1c
  This is line 1 of "file-3"
```

sdiff

比较 和/或 编辑2个文件, 将它们合并到一个输出文件中. 因为这个命令的交互特性, 所以在脚本中很少使用这个命令.

cmp

cmp 命令是上边 diff 命令的一个简单版本. diff 命令会报告两个文件的不同之处, 而 cmp 命令仅仅指出那些位置有不同, 而不会显示不同的具体细节.

注意: 与 diff 一样,如果两个文件相同 cmp 返回0作为退出码, 如果不同返回1. 这样就可以用在 shell 脚本的测试结构中了.

Example 12-32 在一个脚本中使用 cmp 来比较2个文件.

```
################################Start Script#######################################
1 #!/bin/bash
2
3 ARGS=2  # 脚本需要2个参数.
4 E_BADARGS=65
5 E_UNREADABLE=66
6
7 if [ $# -ne "$ARGS" ]
8 then
9   echo "Usage: `basename $0` file1 file2"
10   exit $E_BADARGS
```

```
11 fi
12
13 if [[ ! -r "$1" || ! -r "$2" ]]
14 then
15   echo "Both files to be compared must exist and be readable."
16   exit $E_UNREADABLE
17 fi
18
19 cmp $1 $2 &> /dev/null  # /dev/null 将会禁止 "cmp" 命令的输出.
20 #   cmp -s $1 $2  与上边这句结果相同 ("-s" 选项是安静标志)
21 #   Thank you  Anders Gustavsson for pointing this out.
22 #
23 # 用 'diff' 命令也可以, 比如,   diff $1 $2 &> /dev/null
24
25 if [ $? -eq 0 ]        # 测试 "cmp" 命令的退出码.
26 then
27   echo "File \"$1\" is identical to file \"$2\"."
28 else
29   echo "File \"$1\" differs from file \"$2\"."
30 fi
31
32 exit 0
```

################################End Script###

注意: 用 zcmp 处理 gzip 文件.

comm

多功能的文件比较工具. 使用这个命令之前必须先排序.

comm -options first-file second-file

comm file-1 file-2 将会输出3列:

* 第 1 列 = 只在 file-1 中存在的行
* 第 2 列 = 只在 file-2 中存在的行
* 第 2 列 = 两边相同的行.

下列选项可以禁止1列或多列的输出.

* -1 禁止显示第一栏 (译者: 在 File1 中的行)
* -2 禁止显示第二栏 (译者: 在 File2 中的行)
* -3 禁止显示第三栏 (译者: File1 和 File2 公共的行)
* -12 禁止第一列和第二列, (就是说选项可以组合).

一般工具

basename

从文件名中去掉路径信息, 只打印出文件名. 结构 basename $0 可以让脚本知道它自己的名字, 也就是, 它被调用的名字. 可以用来显示用法信息, 比如如果你调用脚本的时候缺少参数, 可以使用如下语句:

```
  1 echo "Usage: `basename $0` arg1 arg2 ... argn"
```

dirname

从带路径的文件名中去掉文件名, 只打印出路径信息.

注意: basename 和 dirname 可以操作任意字符串. 参数可以不是一个真正存在的文件, 甚至可以不是一个文件名.(参见 Example A-7).

Example 12-33 basename 和 dirname

################################Start Script###

```
 1 #!/bin/bash
 2
 3 a=/home/bozo/daily-journal.txt
 4
 5 echo "Basename of /home/bozo/daily-journal.txt = `basename $a`"
 6 echo "Dirname of /home/bozo/daily-journal.txt = `dirname $a`"
 7 echo
 8 echo "My own home is `basename ~/`."        # `basename ~` also works.
 9 echo "The home of my home is `dirname ~/`."  # `dirname ~`  also works.
10
11 exit 0
```

################################End Script##

split, csplit

将一个文件分割为几个小段的工具. 这些命令通常用来将大的文件分割, 并备份到软盘上, 或者是为了切成合适的尺寸用 email 上传.

csplit 根据 上下文 来切割文件, 切割的位置将会发生在模式匹配的地方.

sum, cksum, md5sum, sha1sum

这些都是用来产生 checksum 的工具. checksum 的目的是用来检验文件的完整性, 是对文件的内容进行数学计算而得到的. 出于安全目的一个脚本可能会有一个 checksum 列表, 这样可以确保关键系统文件的内容不会被修改或损坏. 对于需要安全性的应用来说, 应该使用 md5sum (message digest 5 checksum) 命令, 或者更好的更新的 sha1sum (安全 Hash 算法).

```
bash$ cksum /boot/vmlinuz
1670054224 804083 /boot/vmlinuz

bash$ echo -n "Top Secret" | cksum
3391003827 10



bash$ md5sum /boot/vmlinuz
0f43eccea8f09e0a0b2b5cf1dcf333ba  /boot/vmlinuz

bash$ echo -n "Top Secret" | md5sum
8babc97a6f62a4649716f4df8d61728f  -
```

注意: cksum 命令将会显示目标的尺寸(字节), 目标可以使文件或 stdout.

md5sum 和 sha1sum 命令在它们收到 stdout 的输入时候, 显示一个 dash .

Example 12-34 检查文件完整性

################################Start Script##

```
 1 #!/bin/bash
 2 # file-integrity.sh: 检查一个给定目录下的文件
 3 #                   是否被改动了.
 4
 5 E_DIR_NOMATCH=70
 6 E_BAD_DBFILE=71
 7
 8 dbfile=File_record.md5
 9 # 存储记录的文件名 (数据库文件).
```

```
10
11
12 set_up_database ()
13 {
14   echo ""$directory"" > "$dbfile"
15   # 把目录名写到文件的第一行.
16   md5sum "$directory"/* >> "$dbfile"
17   # 在文件中附上  md5 checksums 和 filenames.
18 }
19
20 check_database ()
21 {
22   local n=0
23   local filename
24   local checksum
25
26   # ---------------------------------------- #
27   #  这个文件检查其实是不必要的,
28   #+ 但是能安全一些.
29
30   if [ ! -r "$dbfile" ]
31   then
32     echo "Unable to read checksum database file!"
33     exit $E_BAD_DBFILE
34   fi
35   # ---------------------------------------- #
36
37   while read record[n]
38   do
39
40     directory_checked="${record[0]}"
41     if [ "$directory_checked" != "$directory" ]
42     then
43       echo "Directories do not match up!"
44       # 换个目录试一下.
45       exit $E_DIR_NOMATCH
46     fi
47
48     if [ "$n" -gt 0 ]   # 不是目录名.
49     then
50       filename[n]=$( echo ${record[$n]} | awk '{ print $2 }' )
51       #  md5sum 向后写记录,
52       #+ 先写 checksum, 然后写 filename.
53       checksum[n]=$( md5sum "${filename[n]}" )
54
55
56       if [ "${record[n]}" = "${checksum[n]}" ]
57       then
```

```
58        echo "${filename[n]} unchanged."
59
60      elif [ "`basename ${filename[n]}`" != "$dbfile" ]
61           #  跳过checksum 数据库文件,
62           #+ 因为在每次调用脚本它都会被修改.
63      #  ---
64      #  这不幸的意味着当我们在 $PWD中运行这个脚本
65      #+ 时, 修改这个 checksum 数
66      #+ 据库文件将不会被检测出来.
67      #  练习: 修复这个问题.
68  then
69          echo "${filename[n]} : CHECKSUM ERROR!"
70       # 因为最后的检查, 文件已经被修改.
71      fi
72
73      fi
74
75
76
77     let "n+=1"
78   done <"$dbfile"       # 从 checksum 数据库文件中读.
79
80 }
81
82 # ============================================================ #
83 # main ()
84
85 if [ -z  "$1" ]
86 then
87   directory="$PWD"      #  如果没制定参数,
88 else                    #+ 那么就使用当前的工作目录.
89   directory="$1"
90 fi
91
92 clear                   # 清屏.
93 echo " Running file integrity check on $directory"
94 echo
95
96 # ------------------------------------------------------------------ #
97   if [ ! -r "$dbfile" ] # 是否需要建立数据库文件?
98   then
99     echo "Setting up database file, \""$directory"/"$dbfile"\"."; echo
100     set_up_database
101   fi
102 # ------------------------------------------------------------------ #
103
104 check_database          # 调用主要处理函数.
105
```

```
106 echo
107
108 #  你可能想把这个脚本的输出重定向到文件中,
109 #+ 尤其在这个目录中有很多文件的时候.
110
111 exit 0
112
113 #  如果要对数量非常多的文件做完整性检查,
114 #+ 可以考虑一下 "Tripwire" 包,
115 #+ http://sourceforge.net/projects/tripwire/.
116
################################End Script#########################################
```

参见 Example A-19 和 Example 33-14 , 这两个例子展示了 md5sum 命令的用法.

注意: 已经有 128-bit md5sum 被破解的报告了,所以现在更安全的 160-bit sha1sum 是非常受欢迎的, 并且已经被加入到 checksum 工具包中.

一些安全顾问认为即使是 sha1sum 也是会被泄漏的. 所以, 下一个工具是什么呢?

-- 512-bit 的 checksum 工具?

```
bash$ md5sum testfile
e181e2c8720c60522c4c4c981108e367  testfile


bash$ sha1sum testfile
5d7425a9c08a66c3177f1e31286fa40986ffc996  testfile
```

shred

用随机字符填充文件, 使得文件无法恢复, 这样就可以保证文件安全的被删除. 这个命令的效果与 Example 12-55 一样, 但是使用这个命令是一种更优雅更彻底的方法.

这是一个 GNU fileutils.

注意: 即使使用了 shred 命令, 高级的(forensic)辩论技术还是能够恢复文件的内容.

编码和解码

uuencode

这个工具用来把二进制文件编码成 ASCII 字符串,这个工具适用于编码e-mail消息体,或者新闻组消息.

uudecode

这个工具用来把 uuencode 后的 ASCII 字符串恢复为二进制文件.

Example 12-35 Uudecod 编码后的文件

```
################################Start Script#########################################
1 #!/bin/bash
2 # 在当前目录下 uudecode 所有用 uuencode 编码的文件.
3
4 lines=35       # 允许读头部的 35 行(范围很宽).
5
6 for File in *   # Test 所有 $PWD 下的文件.
7 do
8   search1=`head -$lines $File | grep begin | wc -w`
9   search2=`tail -$lines $File | grep end | wc -w`
10  #  Uuencode 过的文件在文件开始的地方有个 "begin",
11  #+ 在文件结尾的地方有个 "end".
12  if [ "$search1" -gt 0 ]
```

```
13   then
14     if [ "$search2" -gt 0 ]
15     then
16       echo "uudecoding - $File -"
17       uudecode $File
18     fi
19   fi
20 done
21
22 #  小心不要让这个脚本运行自己,
23 #+ 因为它也会把自身也认为是一个 uuencoded 文件,
24 #+ 这都是因为这个脚本自身也包含 "begin" 和 "end".
25
26 #  练习:
27 #  -----
28 #  修改这个脚本, 让它可以检查一个新闻组的每个文件,
29 #+ 并且如果下一个没找的话就跳过.
30
31 exit 0
##################################End Script#############################################
```

注意: fold -s 命令在处理从 Usenet 新闻组下载下来的长的uudecode 文本消息的时候可能会有用(可能在管道中).

mimencode, mmencode

mimencode 和 mmencode 命令处理多媒体编码的 email 附件. 虽然 mail 用户代理(比如 pine 或 kmail) 通常情况下都会自动处理, 但是这些特定的工具允许从命令行或 shell脚本中来手动操作这些附件.

crypt

这个工具曾经是标准的 UNIX 文件加密工具. [3] 政府由于政策上的动机规定禁止加密软件的输出, 这样导致了 crypt 命令从 UNIX 世界消失, 并且在大多数的 Linux 发行版中也没有这个命令. 幸运的是, 程序员们想出了一些替代它的方法, 在这些方法中有作者自己的 cruft (参见 Example A-4).

一些杂项工具

mktemp

使用一个"唯一"的文件名来创建一个 临时文件 [4] . 如果不带参数的在命令行下调用这个命令时, 将会在 /tmp 目录下产生一个零长度的文件.

```
bash$ mktemp
/tmp/tmp.zzsvql3154
```

```
1 PREFIX=filename
2 tempfile=`mktemp $PREFIX.XXXXXX`
3 #                  ^^^^^^ 在这个临时的文件名中
4 #+                       至少需要6个占位符.
5 #  如果没有指定临时文件的文件名,
6 #+ 那么默认就是 "tmp.XXXXXXXXXX".
7
8 echo "tempfile name = $tempfile"
9 # tempfile name = filename.QA2ZpY
10 #             或者一些其他的相似的名字...
11
```

```
  12 #  使用 600 为文件权限
  13 #+ 来在当前工作目录下创建一个这样的文件.
  14 #  这样就不需要 "umask 177" 了.
  15 #  但不管怎么说, 这也是一个好的编程风格.
```

make

build 和 compile 二进制包的工具. 当源文件被增加或修改时就会触发一些操作, 这个工具用来控制这些操作.

make 命令将会检查 Makefile, makefile 是文件的依赖和操作列表.

install

特殊目的的文件拷贝命令, 与 cp 命令相似, 但是具有设置拷贝文件的权限和属性的能力. 这个命令看起来是为了安装软件包所定制的, 而且就其本身而言, 这个命令经常出现在 Makefile 中(在 make install : 区中). 在安装脚本中也会看到这个命令的使用.

dos2unix

这个工具是由 Benjamin Lin 和其同事编写的, 目的是将 DOS 格式的文本文件 (以 CR-LF 为行结束符) 转换为 UNIX 格式 (以 LF 为行结束符), 反过来也一样.

ptx

ptx [targetfile] 命令将会输出目标文件的序列改变的索引(交叉引用列表). 如果必要的话, 这个命令可以在管道中进行更深层次的过滤和格式化.

more, less

分页显示文本文件或 stdout, 一次一屏.可以用来过滤 stdout 的输出 . . . 或一个脚本的输出.

more 命令的一个有趣的应用就是测试一个命令序列的执行, 来避免可能发生的糟糕的结果.

```
  1 ls /home/bozo | awk '{print "rm -rf " $1}' | more
  2 #                                  ^^^^
  3
  4 # 检测下边(灾难性的)命令行的效果:
  5 #     ls /home/bozo | awk '{print "rm -rf " $1}' | sh
  6 #     推入 shell 中执行 . . .
```

注意事项:

[1] 在这里所讨论的一个归档文件, 只不过是存储在一个单一位置上的一些相关文件的集合.

[2] tar czvf archive_name.tar.gz * 可以 包含当前工作目录下的点文件. 这是一个未文档化的 GNU tar 的"特征".

[3] 这是一个对称的块密码, 过去曾在单系统或本地网络中用来加密文件, 用来对抗 "public key" 密码类, pgp 就是一个众所周知的例子.

[4] 使用 -d 选项可以创建一个临时的目录.

12.6 通讯命令

下边命令中的某几个命令你会在 "追踪垃圾邮件" 练习中找到其用法, 用来进行网络数据的转换和分析.

信息与统计

host

通过名字或 IP 地址来搜索一个互联网主机的信息, 使用 DNS.

```
bash$ host surfacemail.com
surfacemail.com. has address 202.92.42.236
```

ipcalc

显示一个主机 IP 信息. 使用 -h 选项, ipcalc 将会做一个 DNS 的反向查询, 通过 IP 地址找到主机(服务器)名.

```
bash$ ipcalc -h 202.92.42.236
HOSTNAME=surfacemail.com
```

nslookup

通过 IP 地址在一个主机上做一个互联网的 "名字服务查询". 事实上这与 ipcalc -h 或 dig -x 等价. 这个命令既可以交互运行也可以非交互运行, 换句话说, 就是在脚本中运行.

nslookup 命令据说已经慢慢被"忽视"了, 但是它还是有它的用处.

```
bash$ nslookup -sil 66.97.104.180
nslookup kuhleersparnis.ch
Server:         135.116.137.2
Address:        135.116.137.2#53
Non-authoritative answer:
Name:   kuhleersparnis.ch
```

dig

域信息查询. 与 nslookup 很相似, dig 在一个主机上做一个互联网的 "名字服务查询". 这个命令既可以交互运行也可以非交互运行, 换句话说, 就是在脚本中运行.

下边是一些 dig 命令有趣的选项, +time=N 选项用来设置查询超时为 N 秒, +nofail 选项用来持续查询服务器直到收到一个响应, -x 选项会做反向地址查询.

比较下边这3个命令的输出, dig -x , ipcalc -h 和 nslookup.

```
bash$ dig -x 81.9.6.2
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 11649
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 0
;; QUESTION SECTION:
;2.6.9.81.in-addr.arpa.          IN      PTR
;; AUTHORITY SECTION:
6.9.81.in-addr.arpa.    3600    IN      SOA     ns.eltel.net. noc.eltel.net.
2002031705 900 600 86400 3600
;; Query time: 537 msec
;; SERVER: 135.116.137.2#53(135.116.137.2)
;; WHEN: Wed Jun 26 08:35:24 2002
;; MSG SIZE  rcvd: 91
```

Example 12-36 查找滥用的连接来报告垃圾邮件发送者

```
##################################Start Script#########################################
 1 #!/bin/bash
 2 # spam-lookup.sh: 查找滥用的连接来报告垃圾邮件发送者.
 3 # 感谢 Michael Zick.
 4
 5 # 检查命令行参数.
 6 ARGCOUNT=1
 7 E_WRONGARGS=65
 8 if [ $# -ne "$ARGCOUNT" ]
 9 then
10   echo "Usage: `basename $0` domain-name"
11   exit $E_WRONGARGS
12 fi
```

```
13
14
15 dig +short $1.contacts.abuse.net -c in -t txt
16 # 也试试:
17 #    dig +nssearch $1
18 #    尽量找到 "可信赖的名字服务器" 并且显示 SOA 记录.
19
20 # 下边这句也可以:
21 #    whois -h whois.abuse.net $1
22 #         ^^ ^^^^^^^^^^^^^^^^  指定主机.
23 #    使用这个命令也可以查找多个垃圾邮件发送者, 比如:"
24 #    whois -h whois.abuse.net $spamdomain1 $spamdomain2 . . .
25
26
27 #  练习:
28 #  -----
29 #  扩展这个脚本的功能,
30 #+ 让它可以自动发送 e-mail 来通知
31 #+ 需要对此负责的 ISP 的联系地址.
32 #  暗示: 使用 "mail" 命令.
33
34 exit $?
35
36 # spam-lookup.sh chinatietong.com
37 #              一个已知的垃圾邮件域.(译者: 中国铁通. . .)
38
39 # "crnet_mgr@chinatietong.com"
40 # "crnet_tec@chinatietong.com"
41 # "postmaster@chinatietong.com"
42
43
44 #  如果想找到这个脚本的一个更详尽的版本,
45 #+ 请访问 SpamViz 的主页, http://www.spamviz.net/index.html.
```

##################################End Script###

Example 12-37 分析一个垃圾邮件域<rojy bug>

##################################Start Script###

```
 1 #! /bin/bash
 2 # is-spammer.sh: 鉴别一个垃圾邮件域
 3
 4 # $Id: is-spammer, v 1.4 2004/09/01 19:37:52 mszick Exp $
 5 # 上边这行是 RCS ID 信息.
 6 #
 7 #  这是附件中捐献脚本 is_spammer.bash
 8 #+ 的一个简单版本.
 9
10 # is-spammer <domain.name>
11
12 # 使用外部程序: 'dig'
```

```
13 # 测试版本: 9.2.4rc5
14
15 # 使用函数.
16 # 使用 IFS 来分析分配在数组中的字符串.
17 # 检查 e-mail 黑名单.
18
19 # 使用来自文本体中的 domain.name:
20 # http://www.good_stuff.spammer.biz/just_ignore_everything_else
21 #                    ^^^^^^^^^^
22 # 或者使用来自任意 e-mail 地址的 domain.name:
23 # Really_Good_Offer@spammer.biz
24 #
25 # 并将其作为这个脚本的唯一参数.
26 #(另: 你的 Inet 连接应该保证连接)
27 #
28 # 这样, 在上边两个实例中调用这个脚本:
29 #       is-spammer.sh spammer.biz
30
31
32 # Whitespace == :Space:Tab:Line Feed:Carriage Return:
33 WSP_IFS=$'\x20'$'\x09'$'\x0A'$'\x0D'
34
35 # No Whitespace == Line Feed:Carriage Return
36 No_WSP=$'\x0A'$'\x0D'
37
38 # 域分隔符为点分10进制 ip 地址
39 ADR_IFS=${No_WSP}'.'
40
41 # 取得 dns 文本资源记录.
42 # get_txt <error_code> <list_query>
43 get_txt() {
44
45     # 分析在"."中分配的 $1.
46     local -a dns
47     IFS=$ADR_IFS
48     dns=( $1 )
49     IFS=$WSP_IFS
50     if [ "${dns[0]}" == '127' ]
51     then
52         # 查看此处是否有原因.
53         echo $(dig +short $2 -t txt)
54     fi
55 }
56
57 # 取得 dns 地址资源记录.
58 # chk_adr <rev_dns> <list_server>
59 chk_adr() {
60     local reply
```

```
61    local server
62    local reason
63
64    server=${1}${2}
65    reply=$( dig +short ${server} )
66
67    # 假设应答可能是一个错误码 . . .
68    if [ ${#reply} -gt 6 ]
69    then
70        reason=$(get_txt ${reply} ${server} )
71        reason=${reason:-${reply}}
72    fi
73    echo ${reason:-' not blacklisted.'}
74 }
75
76 # 需要从名字中取得 IP 地址.
77 echo 'Get address of: '$1
78 ip_adr=$(dig +short $1)
79 dns_reply=${ip_adr:-' no answer '}
80 echo ' Found address: '${dns_reply}
81
82 # 一个可用的应答至少是4个数字加上3个点.
83 if [ ${#ip_adr} -gt 6 ]
84 then
85    echo
86    declare query
87
88    # 分析点中的分配.
89    declare -a dns
90    IFS=$ADR_IFS
91    dns=( ${ip_adr} )
92    IFS=$WSP_IFS
93
94    # Reorder octets into dns query order.
95    rev_dns="${dns[3]}"."${dns[2]}"."${dns[1]}"."${dns[0]}"'.'
96
97 # 参见: http://www.spamhaus.org (Conservative, well maintained)
98    echo -n 'spamhaus.org says: '
99    echo $(chk_adr ${rev_dns} 'sbl-xbl.spamhaus.org')
100
101 # 参见: http://ordb.org (Open mail relays)
102    echo -n '   ordb.org  says: '
103    echo $(chk_adr ${rev_dns} 'relays.ordb.org')
104
105 # 参见: http://www.spamcop.net/ (你可以在这里报告 spammer)
106    echo -n ' spamcop.net says: '
107    echo $(chk_adr ${rev_dns} 'bl.spamcop.net')
108
```

```
109 # # # 其他的黑名单操作 # # #
110
111 # 参见: http://cbl.abuseat.org.
112     echo -n ' abuseat.org says: '
113     echo $(chk_adr ${rev_dns} 'cbl.abuseat.org')
114
115 # 参见: http://dsbl.org/usage (Various mail relays)
116     echo
117     echo 'Distributed Server Listings'
118     echo -n '       list.dsbl.org says: '
119     echo $(chk_adr ${rev_dns} 'list.dsbl.org')
120
121     echo -n '   multihop.dsbl.org says: '
122     echo $(chk_adr ${rev_dns} 'multihop.dsbl.org')
123
124     echo -n 'unconfirmed.dsbl.org says: '
125     echo $(chk_adr ${rev_dns} 'unconfirmed.dsbl.org')
126
127 else
128     echo
129     echo 'Could not use that address.'
130 fi
131
132 exit 0
133
134 # 练习:
135 # -----
136
137 # 1) 检查脚本的参数,
138 #    并且如果必要的话使用合适的错误消息退出.
139
140 # 2) 检查调用这个脚本的时候是否在线,
141 #    并且如果必要的话使用合适的错误消息退出.
142
143 # 3) Substitute generic variables for "hard-coded" BHL domains.
144
145 # 4) 通过对 'dig' 命令使用 "+time=" 选项
146     来给这个脚本设置一个暂停.
##################################End Script#############################################
```

想获得比上边这个脚本更详细的版本, 参见 Example A-27.

traceroute

跟踪包发送到远端主机过程中的路由信息. 这个命令在 LAN, WAN, 或者在 Internet 上都可以正常工作. 远端主机可以通过 IP 地址来指定. 这个命令的输出也可以通过管道中的 grep 或 sed 命令来过滤.

```
bash$ traceroute 81.9.6.2
traceroute to 81.9.6.2 (81.9.6.2), 30 hops max, 38 byte packets
1  tc43.xjbnnbrb.com (136.30.178.8)  191.303 ms  179.400 ms  179.767 ms
2  or0.xjbnnbrb.com (136.30.178.1)  179.536 ms  179.534 ms  169.685 ms
```

3 192.168.11.101 (192.168.11.101) 189.471 ms 189.556 ms *
...

ping

广播一个 "ICMP ECHO_REQUEST" 包到其他主机上, 既可以是本地网络也可以使远端网络.

这是一个测试网络连接的诊断工具, 应该小心使用.

一个成功的 ping 返回的 退出码 为 0. 可以用在脚本的测试语句中.

```
bash$ ping localhost
PING localhost.localdomain (127.0.0.1) from 127.0.0.1 : 56(84) bytes of data.
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=0 ttl=255 time=709 usec
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=1 ttl=255 time=286 usec
--- localhost.localdomain ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/mdev = 0.286/0.497/0.709/0.212 ms
```

whois

执行DNS (域名系统) 查询lookup. -h 选项允许指定需要查询的特定的 whois 服务器.

参见 Example 4-6 和 Example 12-36.

finger

取得网络上的用户信息. 另外这个命令可以显示一个用户的~/.plan, ~/.project, 和 ~/.forward 文件, 如果存在的话.

```
bash$ finger
Login  Name          Tty      Idle  Login Time   Office     Office Phone
bozo   Bozo Bozeman   tty1       8  Jun 25 16:59
bozo   Bozo Bozeman   ttyp0         Jun 25 16:59
bozo   Bozo Bozeman   ttyp1         Jun 25 17:07



bash$ finger bozo
Login: bozo                         Name: Bozo Bozeman
Directory: /home/bozo               Shell: /bin/bash
Office: 2355 Clown St., 543-1234
On since Fri Aug 31 20:13 (MST) on tty1    1 hour 38 minutes idle
On since Fri Aug 31 20:13 (MST) on pts/0   12 seconds idle
On since Fri Aug 31 20:13 (MST) on pts/1
On since Fri Aug 31 20:31 (MST) on pts/2   1 hour 16 minutes idle
No mail.
No Plan.
```

处于安全上的考虑, 许多网络都禁用了 finger 以及和它相关的幽灵进程. [1]

chfn

修改 finger 命令所显示出来的用户信息.

vrfy

验证一个互联网的 e-mail 地址.

远端主机接入

sx, rx

sx 和 rx 命令使用 xmodem 协议, 设置服务来向远端主机传输文件和接收文件. 这些都是通讯安装包的一般部分, 比如 minicom.

sz, rz

sz 和 rz 命令使用 zmodem 协议, 设置服务来向远端主机传输文件和接收文件. zmodem

协议在某些方面比 xmodem强, 比如使用更快的的传输波特率, 并且可以对中断的文件进行续传.与 sx 一样 rx, 这些都是通讯安装包的一般部分.

ftp

向远端服务器上传或下载的工具和协议. 一个ftp会话可以写到脚本中自动运行. (见 Example 17-6, Example A-4, 和 Example A-13).

uucp, uux, cu

uucp: UNIX 到 UNIX 拷贝. 这是一个通讯安装包, 目的是为了在 UNIX 服务器之间传输文件. 使用 shell 脚本来处理 uucp 命令序列是一种有效的方法.

因为互联网和电子邮件的出现, uucp 现在看起来已经很落伍了, 但是这个命令在互联网连接不可用或者不适合使用的地方, 这个命令还是可以完美的运行. uucp 的优点就是它的容错性, 即使有一个服务将拷贝操作中断了, 那么当连接恢复的时候, 这个命令还是可以在中断的地方续传.

---

uux: UNIX 到 UNIX 执行. 在远端系统上执行一个命令.这个命令是 uucp 包的一部分.

---

cu: Call Up 一个远端系统并且作为一个简单终端进行连接. 这是一个 telnet 的缩减版本. 这个命令是 uucp 包的一部分.

telnet

连接远端主机的工具和协议.

注意:telnet 协议本身包含安全漏洞, 因此我们应该适当的避免使用.

wget

wget 工具使用非交互的形式从 web 或 ftp 站点上取得或下载文件. 在脚本中使用正好.

```
1 wget -p http://www.xyz23.com/file01.html
2 #  The -p or --page-requisite 选项将会使得 wget 取得显示指定页时
3 #+ 所需要的所有文件.(译者: 比如内嵌图片和样式表等).
4
5 wget -r ftp://ftp.xyz24.net/~bozo/project_files/ -O $SAVEFILE
6 #  -r 选项将会递归的从指定站点
7 #+ 上下载所有连接.
```

Example 12-38 获得一份股票报价

```
################################Start Script#########################################
1 #!/bin/bash
2 # quote-fetch.sh: 下载一份股票报价.
3
4
5 E_NOPARAMS=66
6
7 if [ -z "$1" ]  # 必须指定需要获取的股票(代号).
8   then echo "Usage: `basename $0` stock-symbol"
9   exit $E_NOPARAMS
10 fi
11
12 stock_symbol=$1
13
14 file_suffix=.html
15 # 获得一个 HTML 文件, 所以要正确命名它.
16 URL='http://finance.yahoo.com/q?s='
17 # Yahoo 金融板块, 后缀是股票查询.
```

```
18
19 # ------------------------------------------------------------
20 wget -O ${stock_symbol}${file_suffix} "${URL}${stock_symbol}"
21 # ------------------------------------------------------------
22
23
24 # 在 http://search.yahoo.com 上查询相关材料:
25 # ------------------------------------------------------------
26 # URL="http://search.yahoo.com/search?fr=ush-news&p=${query}"
27 # wget -O "$savefilename" "${URL}"
28 # ------------------------------------------------------------
29 # 保存相关 URL 的列表.
30
31 exit $?
32
33 # 练习:
34 # -----
35 #
36 # 1) 添加一个测试来验证用户正在线.
37 #    (暗示: 对 "ppp" 或 "connect" 来分析 'ps -ax' 的输出.
38 #
39 # 2) 修改这个脚本, 让这个脚本具有获得本地天气预报的能力,
40 #+   将用户的 zip code 作为参数.
```

################################End Script###

参见 Example A-29 和 Example A-30.

lynx

lynx 是一个网页浏览器, 也是一个文件浏览器. 它可以(通过使用 -dump 选项)在脚本中使用. 它的作用是可以从 Web 或 ftp 站点上非交互的获得文件.

```
1 lynx -dump http://www.xyz23.com/file01.html >$SAVEFILE
```

使用 -traversal 选项, lynx 将从参数中指定的 HTTP URL 开始, 遍历指定服务器上的所有链接. 如果与 -crawl 选项一起用的话, 将会把每个输出的页面文本都放到一个 log 文件中.

rlogin

远端登陆, 在远端的主机上开启一个会话. 这个命令存在安全隐患, 所以要使用 ssh 来代替.

rsh

远端 shell, 在远端的主机上执行命令. 这个命令存在安全隐患, 所以要使用 ssh 来代替.

rcp

远端拷贝, 在网络上的不同主机间拷贝文件.

rsync

远端同步, 在网络上的不同主机间(同步)更新文件.

```
bash$ rsync -a ~/sourcedir/*txt /node1/subdirectory/
```

Example 12-39 更新 Fedora 4 <rojy bug>

################################Start Script###

```
1 #!/bin/bash
2 # fc4upd.sh
3
```

```
4 # 脚本作者: Frank Wang.
5 # 本书作者作了少量修改.
6 # 授权在本书中使用.
7
8
9 #  使用 rsync 命令从镜像站点上下载 Fedora 4 的更新.
10 #  为了节省空间, 如果有多个版本存在的话,
11 #+ 只下载最新的包.
12
13 URL=rsync://distro.ibiblio.org/fedora-linux-core/updates/
14 # URL=rsync://ftp.kddilabs.jp/fedora/core/updates/
15 # URL=rsync://rsync.planetmirror.com/fedora-linux-core/updates/
16
17 DEST=${1:-/var/www/html/fedora/updates/}
18 LOG=/tmp/repo-update-$(/bin/date +%Y-%m-%d).txt
19 PID_FILE=/var/run/${0##*/}.pid
20
21 E_RETURN=65       # 某些意想不到的错误.
22
23
24 # 一搬 rsync 选项
25 # -r: 递归下载
26 # -t: 保存时间
27 # -v: verbose
28
29 OPTS="-rtv --delete-excluded --delete-after --partial"
30
31 # rsync include 模式
32 # Leading slash causes absolute path name match.
33 INCLUDE=(
34     "/4/i386/kde-i18n-Chinese*"
35 #   ^                     ^
36 # 双引号是必须的, 用来防止file globbing.
37 )
38
39
40 # rsync exclude 模式
41 # 使用 "#" 临时注释掉一些不需要的包.
42 EXCLUDE=(
43     /1
44     /2
45     /3
46     /testing
47     /4/SRPMS
48     /4/ppc
49     /4/x86_64
50     /4/i386/debug
51    "/4/i386/kde-i18n-*"
```

```
52    "/4/i386/openoffice.org-langpack-*"
53    "/4/i386/*i586.rpm"
54    "/4/i386/GFS-*"
55    "/4/i386/cman-*"
56    "/4/i386/dlm-*"
57    "/4/i386/gnbd-*"
58    "/4/i386/kernel-smp*"
59 #  "/4/i386/kernel-xen*"
60 #  "/4/i386/xen-*"
61 )
62
63
64 init () {
65    # 让管道命令返回可能的 rsync 错误, 比如, 网络延时(stalled network).
66    set -o pipefail
67
68    TMP=${TMPDIR:-/tmp}/${0##*/}.$$     # 保存精炼的下载列表.
69    trap "{
70       rm -f $TMP 2>/dev/null
71    }" EXIT                          # 删除存在的临时文件.
72 }
73
74
75 check_pid () {
76 # 检查进程是否存在.
77    if [ -s "$PID_FILE" ]; then
78       echo "PID file exists. Checking ..."
79       PID=$(/bin/egrep -o "^[[:digit:]]+" $PID_FILE)
80       if /bin/ps --pid $PID &>/dev/null; then
81          echo "Process $PID found. ${0##*/} seems to be running!"
82          /usr/bin/logger -t ${0##*/} \
83             "Process $PID found. ${0##*/} seems to be running!"
84          exit $E_RETURN
85       fi
86       echo "Process $PID not found. Start new process . . ."
87    fi
88 }
89
90
91 #  根据上边的模式,
92 #+ 设置整个文件的更新范围, 从 root 或 $URL 开始.
93 set_range () {
94    include=
95    exclude=
96    for p in "${INCLUDE[@]}"; do
97       include="$include --include \"$p\""
98    done
99
```

```
100     for p in "${EXCLUDE[@]}"; do
101         exclude="$exclude --exclude \"$p\""
102     done
103 }
104
105
106 # 获得并提炼 rsync 更新列表.
107 get_list () {
108     echo $$ > $PID_FILE || {
109         echo "Can't write to pid file $PID_FILE"
110         exit $E_RETURN
111     }
112
113     echo -n "Retrieving and refining update list . . ."
114
115     # 获得列表 -- 为了作为单个命令来运行 rsync 需要 'eval'.
116     # $3 和 $4 是文件创建的日期和时间.
117     # $5 是完整的包名字.
118     previous=
119     pre_file=
120     pre_date=0
121     eval /bin/nice /usr/bin/rsync \
122         -r $include $exclude $URL | \
123         egrep '^dr.x|^-r' | \
124         awk '{print $3, $4, $5}' | \
125         sort -k3 | \
126         { while read line; do
127             # 获得这段运行的秒数, 过滤掉不用的包.
128             cur_date=$(date -d "$(echo $line | awk '{print $1, $2}')" +%s)
129             #  echo $cur_date
130
131             # 取得文件名.
132             cur_file=$(echo $line | awk '{print $3}')
133             #  echo $cur_file
134
135             # 如果可能的话, 从文件名中取得 rpm 的包名字.
136             if [[ $cur_file == *rpm ]]; then
137                 pkg_name=$(echo $cur_file | sed -r -e \
138                     's/(^([^_-]+[_-])+)[[:digit:]]+\..*[_-].*$/\1/')
139             else
140                 pkg_name=
141             fi
142             # echo $pkg_name
143
144             if [ -z "$pkg_name" ]; then   #  如果不是一个 rpm 文件,
145                 echo $cur_file >> $TMP    #+ 然后添加到下载列表里.
146             elif [ "$pkg_name" != "$previous" ]; then   # 发现一个新包.
147                 echo $pre_file >> $TMP                # 输出最新的文件.
```

```
148             previous=$pkg_name                  # 保存当前状态.
149             pre_date=$cur_date
150             pre_file=$cur_file
151         elif [ "$cur_date" -gt "$pre_date" ]; then  #  如果是相同的包, 但是更新一些,
152             pre_date=$cur_date                  #+ 那么就更新最新的.
153             pre_file=$cur_file
154         fi
155         done
156         echo $pre_file >> $TMP                  #  TMP 现在包含所有
157                                                 #+ 提炼过的列表.
158         # echo "subshell=$BASH_SUBSHELL"
159
160     }     # 这里的打括号是为了让最后这句"echo $pre_file >> $TMP"
161         # 也能与整个循环一起放到同一个子 shell ( 1 )中.
162
163     RET=$?  # 取得管道命令的返回码.
164
165     [ "$RET" -ne 0 ] && {
166       echo "List retrieving failed with code $RET"
167       exit $E_RETURN
168     }
169
170     echo "done"; echo
171 }
172
173 # 真正的 rsync 的下载部分.
174 get_file () {
175
176     echo "Downloading..."
177     /bin/nice /usr/bin/rsync \
178       $OPTS \
179       --filter "merge,+/ $TMP" \
180       --exclude '*'  \
181       $URL $DEST     \
182       | /usr/bin/tee $LOG
183
184     RET=$?
185
186       #  --filter merge,+/ is crucial for the intention.
187       #  + modifier means include and / means absolute path.
188       #  Then sorted list in $TMP will contain ascending dir name and
189       #+ prevent the following --exclude '*' from "shortcutting the circuit."
190
191     echo "Done"
192
193     rm -f $PID_FILE 2>/dev/null
194
195     return $RET
```

```
196 }
197
198 # -------
199 # Main
200 init
201 check_pid
202 set_range
203 get_list
204 get_file
205 RET=$?
206 # -------
207
208 if [ "$RET" -eq 0 ]; then
209     /usr/bin/logger -t ${0##*/} "Fedora update mirrored successfully."
210 else
211     /usr/bin/logger -t ${0##*/} "Fedora update mirrored with failure code: $RET"
212 fi
213
214 exit $RET
```

##################################End Script##

使用 rcp, rsync, 和其他一些有安全问题的类似工具, 并将这些工具用在 shell 脚本中是不明智的. 应该考虑使用 ssh, scp, 或者一个 expect 脚本来代替这些不安全的工具.

ssh

安全 shell, 登陆远端主机并在其上运行命令. 这个工具具有身份认证和加密的功能, 可以安全的替换 telnet, rlogin, rcp, 和 rsh 等工具. 参见 man页 来获取详细信息.

Example 12-40 使用 ssh

##################################Start Script##

```
 1 #!/bin/bash
 2 # remote.bash: 使用 ssh.
 3
 4 # 这个例子是 Michael Zick 编写的.
 5 # 授权使用.
 6
 7
 8 #   假设:
 9 #   -----
10 #   fd-2(文件描述符2) 并没有被抛弃 ( '2>/dev/null' ).
11 #   ssh/sshd 假设 stderr ('2') 将会被显示给用户.
12 #
13 #   sshd 正运行在你的机器上.
14 #   对于大多数 '标准' 的发行版, 是应该有的,
15 #+  并且没有一些稀奇古怪的 ssh-keygen.
16
17 # 在你的机器上从命令行中试一下 ssh:
18 #
19 # $ ssh $HOSTNAME
20 # 不同特殊的准备, 你将被要求输入你的密码.
21 #   输入密码
```

```
22 #   完成后,  $ exit
23 #
24 # 好使了么? 如果好使了, 你可以做好准备来获取更多的乐趣了.
25
26 # 在你的机器上用 'root'身份来试试 ssh:
27 #
28 #   $  ssh -l root $HOSTNAME
29 #   当询问密码时, 输入 root 的密码, 别输入你的密码.
30 #         Last login: Tue Aug 10 20:25:49 2004 from localhost.localdomain
31 #   完成后键入 'exit'.
32
33 #  上边的动作将会给你一个交互的shell.
34 #  在 'single command' 模式下建立 sshd 是可能的, <rojy bug>
35 #+ 不过这已经超出本例的范围了.
36 #  唯一需要注意的事情是下面都可以工作在
37 #+ 'single command' 模式.
38
39
40 # 一个基本的写输出(本地)命令.
41
42 ls -l
43
44 # 现在在远端机器上使用同样的基本命令.
45 # 使用一套不同的 'USERNAME' 和 'HOSTNAME' :
46 USER=${USERNAME:-$(whoami)}
47 HOST=${HOSTNAME:-$(hostname)}
48
49 #  现在在远端主机上运行上边的命令行命令,
50 #+ 当然, 所有的传输都被加密了.
51
52 ssh -l ${USER} ${HOST} " ls -l "
53
54 #  期望的结果就是在远端主机上列出你的
55 #+ username 主目录的所有文件.
56 #  如果想看点不一样的, 那就
57 #+ 在别的地方运行这个脚本, 别再你的主目录上运行这个脚本.
58
59 #  换句话说, Bash 命令已经作为一个引用行
60 #+ 被传递到远端的shell 中了,这样就可以在远端的机器上运行它了.
61 #  在这种情况下, sshd 代表你运行了 ' bash -c "ls -l" '.
62
63 #  对于每个命令行如果想不输入密码的话,
64 #+ 对于这种类似的议题, 可以参阅
65 #+    man ssh
66 #+    man ssh-keygen
67 #+    man sshd_config.
68
69 exit 0
```

################################End Script###
注意: 在循环中, ssh 可能会引起意想不到的异常行为. 根据comp.unix 上的shell文档
Usenet post , ssh 继承了循环的标准输入.为了解决这个问题, 使用 ssh 的 -n
或者 -f 选项.
感谢 Jason Bechtel, 指出这点.

scp

安全拷贝, 在功能上与 rcp 很相似, 就是在2个不同的网络主机之间拷贝文件, 但是要通
过鉴权的方式, 并且使用与 ssh 类似的安全层.

Local Network

write

这是一个端到端通讯的工具. 这个工具可以从你的终端上(console 或者 xterm)发送整行
到另一个用户的终端上. mesg 命令当然也可以用来对于一个终端的写权限
因为 write 是需要交互的, 所以这个命令通常不使用在脚本中.

netconfig

用来配置网络适配器(使用 DHCP)的命令行工具. 这个命令对于红帽发行版来说是内置的.

Mail

mail

发送或读取 e-mail 消息.
如果把这个命令行的 mail 客户端当成一个脚本中的命令来使用的话, 效果非常好.

Example 12-41 一个可以mail自己的脚本

################################Start Script###

```
 1 #!/bin/sh
 2 # self-mailer.sh: mail自己的脚本
 3
 4 adr=${1:-`whoami`}     # 如果不指定的话, 默认是当前用户.
 5 #  键入 'self-mailer.sh wiseguy@superdupergenius.com'
 6 #+ 发送这个脚本到这个地址.
 7 #  如果只键入 'self-mailer.sh' (不给参数) 的话, 那么这脚本就会被发送给
 8 #+ 调用者, 比如 bozo@localhost.localdomain.
 9 #
10 #  如果想了解 ${parameter:-default} 结构的更多细节,
11 #+ 请参见第9章 变量重游中的
12 #+ 第3节 参数替换.
13
14 # ============================================================================
15   cat $0 | mail -s "Script \"`basename $0`\" has mailed itself to you." "$adr"
16 # ============================================================================
17
18 # --------------------------------------------
19 #  来自 self-mailing 脚本的一份祝福.
20 #  一个喜欢恶搞的家伙运行了这个脚本,
21 #+ 这导致了他自己收到了这份mail.
22 #  显然的, 有些人确实没什么事好做,
23 #+ 就只能浪费他们自己的时间玩了.
24 # --------------------------------------------
25
26 echo "At `date`, script \"`basename $0`\" mailed to "$adr"."
```

27
28 exit 0
################################End Script###
mailto
 与 mail 命令很相似, mailto 命令可以使用命令行或在脚本中发送 e-mail 消息. 然而,
 mailto 命令也允许发送 MIME (多媒体) 消息.
vacation
 这个工具可以自动回复 e-mail 给发送者, 表示邮件的接受者正在度假暂时无法收到邮件.
 这个工具与 sendmail 一起运行于网络上, 并且这个工具不支持拨号的 POPmail 帐号.
注意事项:
[1] 一个幽灵进程指的是并未附加在终端会话中的后台进程. 幽灵进程 在指定的时间执
 行指定的服务, 或者由特定的事件出发来执行指定的服务.
12.7 终端控制命令

影响控制台或终端的命令
tput
 初始化终端或者从 terminfo data 中取得终端信息. 不同的选项允许特定的终端操作.
 tput clear 与下边的 clear 等价. tput reset 与下边的 reset 等价. tput sgr0 也可
 以重置终端, 但是并不清除屏幕.
  bash$ tput longname
  xterm terminal emulator (XFree86 4.0 Window System)
 使用 tput cup X Y 将会把光标移动到当前终端的(X,Y)坐标上. 使用这个命令之前一边
 都要先使用一下 clear 命令, 把屏幕清除一下.
 注意: stty 提供了一个更强力的命令专门用来设置如何控制终端.
infocmp
 这个命令会打印出大量的当前终端的信息. 事实上它是引用了 terminfo 数据库.
  bash$ infocmp
  #   通过来自于文件的 infocmp 显示出来:
  /usr/share/terminfo/r/rxvt
  rxvt|rxvt terminal emulator (X Window System),
    am, bce, eo, km, mir, msgr, xenl, xon,
    colors#8, cols#80, it#8, lines#24, pairs#64,
    acsc=``aaffggjjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}~~,
    bel=^G, blink=\E[5m, bold=\E[1m,
    civis=\E[?25l,
    clear=\E[H\E[2J, cnorm=\E[?25h, cr=^M,
    ...
reset
 重置终端参数并且清除屏幕. 与 clear 命令一样, 光标和提示符将会重新出现在终端的
 左上角.
clear
 clear 命令只不过是简单的清除控制台或者 xterm 的屏幕. 光标和提示符将会重新出现
 在屏幕或者 xterm window 的左上角. 这个命令既可以用在命令行中也可以用在脚本中.
 参见 Example 10-25.
script
 这个工具将会记录(保存到一个文件中)所有的用户在控制台下的或在 xterm window下的
 按键信息. 这其实就是创建了一个会话记录.
12.8 数学计算命令

"Doing the numbers"

factor

将一个正数分解为多个素数.

```
bash$ factor 27417
27417: 3 13 19 37
```

bc

Bash 不能处理浮点运算, 并且缺乏特定的一些操作,这些操作都是一些重要的计算功能.

幸运的是, bc 可以解决这个问题.

bc 不仅仅是个多功能灵活的精确的工具, 而且它还提供许多编程语言才具备的一些方便的功能.

bc 比较类似于 C 语言的语法.

因为它是一个完整的 UNIX 工具, 所以它可以用在管道中, bc 在脚本中也是很常用的.

这里有一个简单的使用 bc 命令的模版可以用来在计算脚本中的变量. 用在命令替换中.

```
variable=$(echo "OPTIONS; OPERATIONS" | bc)
```

Example 12-42 按月偿还贷款

```
################################Start Script#########################################
#!/bin/bash
# monthlypmt.sh: 计算按月偿还贷款的数量.

#  这份代码是一份修改版本, 原始版本在 "mcalc" (贷款计算)包中,
#+ 这个包的作者是 Jeff Schmidt 和 Mendel Cooper (本书作者).
#   http://www.ibiblio.org/pub/Linux/apps/financial/mcalc-1.6.tar.gz  [15k]

echo
echo "Given the principal, interest rate, and term of a mortgage,"
echo "calculate the monthly payment."

bottom=1.0

echo
echo -n "Enter principal (no commas) "
read principal
echo -n "Enter interest rate (percent) "  # 如果是 12%, 那就键入 "12", 别输入 ".12".
read interest_r
echo -n "Enter term (months) "
read term


interest_r=$(echo "scale=9; $interest_r/100.0" | bc) # 转换成小数.
                # "scale" 指定了有效数字的个数.


interest_rate=$(echo "scale=9; $interest_r/12 + 1.0" | bc)


top=$(echo "scale=9; $principal*$interest_rate^$term" | bc)
```

```
32
33  echo; echo "Please be patient. This may take a while."
34
35  let "months = $term - 1"
36 # ============================================================================
37  for ((x=$months; x > 0; x--))
38  do
39    bot=$(echo "scale=9; $interest_rate^$x" | bc)
40    bottom=$(echo "scale=9; $bottom+$bot" | bc)
41 #  bottom = $(($bottom + $bot"))
42  done
43 # ============================================================================
44
45 # ----------------------------------------------------------------------
46 #  Rick Boivie 给出了一个对上边循环的修改,
47 #+ 这个修改更加有效率, 将会节省大概 2/3 的时间.
48
49 # for ((x=1; x <= $months; x++))
50 # do
51 #   bottom=$(echo "scale=9; $bottom * $interest_rate + 1" | bc)
52 # done
53
54
55 #  然后他又想出了一个更加有效率的版本,
56 #+ 将会节省 95% 的时间!
57
58 # bottom=`{
59 #     echo "scale=9; bottom=$bottom; interest_rate=$interest_rate"
60 #     for ((x=1; x <= $months; x++))
61 #     do
62 #         echo 'bottom = bottom * interest_rate + 1'
63 #     done
64 #     echo 'bottom'
65 #     } | bc`      # 在命令替换中嵌入一个 'for 循环'.
66 # ----------------------------------------------------------------------
67 #  On the other hand, Frank Wang suggests:
68 #  bottom=$(echo "scale=9; ($interest_rate^$term-1)/($interest_rate-1)" | bc)
69
70 #  因为 . . .
71 #  在循环后边的算法
72 #+ 事实上是一个等比数列的求和公式.
73 #  求和公式是 e0(1-q^n)/(1-q),
74 #+ e0 是第一个元素 并且 q=e(n+1)/e(n)
75 #+ 和 n 是元素的数量.
76 # ----------------------------------------------------------------------
77
78
79  # let "payment = $top/$bottom"
```

```
payment=$(echo "scale=2; $top/$bottom" | bc)
# 使用2位有效数字来表示美元和美分.

echo
echo "monthly payment = \$$payment"  # 在总和的前边显示美元符号.
echo


exit 0


# 练习:
#   1) 处理输入允许本金总数中的逗号.
#   2) 处理输入允许按照百分号和小数点的形式输入利率.
#   3) 如果你真正想好好编写这个脚本,
#      那么就扩展这个脚本让它能够打印出完整的分期付款表.
```

################################End Script###

Example 12-43 数制转换

################################Start Script###

```
#!/bin/bash
################################################################################
# 脚本      : base.sh - 用不同的数值来打印数字 (Bourne Shell)
# 作者      : Heiner Steven (heiner.steven@odn.de)
# 日期      : 07-03-95
# 类型      : 桌面
# $Id: base.sh,v 1.2 2000/02/06 19:55:35 heiner Exp $
# ==> 上边这行是 RCS ID 信息.
################################################################################
# 描述
#
# Changes
# 21-03-95 stv fixed error occuring with 0xb as input (0.2)
################################################################################

# ==> 在本书中使用这个脚本通过了作者的授权.
# ==> 注释是本书作者添加的.

NOARGS=65
PN=`basename "$0"`          # 程序名
VER=`echo '$Revision: 1.2 $' | cut -d' ' -f2`  # ==> VER=1.2

Usage () {
    echo "$PN - print number to different bases, $VER (stv '95)
usage: $PN [number ...]

If no number is given, the numbers are read from standard input.
A number may be
    binary (base 2)  starting with 0b (i.e. 0b1100)
```

```
30    octal (base 8)  starting with 0  (i.e. 014)
31    hexadecimal (base 16) starting with 0x (i.e. 0xc)
32    decimal   otherwise (i.e. 12)" >&2
33    exit $NOARGS
34 }  # ==> 打印出用法信息的函数.
35
36 Msg () {
37    for i   # ==> 省略 [list] .
38    do echo "$PN: $i" >&2
39    done
40 }
41
42 Fatal () { Msg "$@"; exit 66; }
43
44 PrintBases () {
45    # 决定数值的数制
46    for i      # ==> 省略 [list]...
47    do         # ==> 所以是对命令行参数进行操作.
48  case "$i" in
49     0b*)  ibase=2;; # 2进制
50     0x*|[a-f]*|[A-F]*) ibase=16;; # 16进制
51     0*)   ibase=8;; # 8进制
52     [1-9]*)  ibase=10;; # 10进制
53     *)
54   Msg "illegal number $i - ignored"
55   continue;;
56  esac
57
58  # 去掉前缀, 将16进制数字转换为大写(bc需要大写)
59  number=`echo "$i" | sed -e 's:^0[bBxX]::' | tr '[a-f]' '[A-F]'`
60  # ==>使用":" 作为sed分隔符, 而不使用"/".
61
62  # 将数字转换为10进制
63  dec=`echo "ibase=$ibase; $number" | bc`  # ==> 'bc' 是个计算工具.
64  case "$dec" in
65     [0-9]*) ;;    # 数字没问题
66     *)  continue;;   # 错误: 忽略
67  esac
68
69  # 在一行上打印所有的转换后的数字.
70  # ==> 'here document' 提供命令列表给'bc'.
71  echo `bc <<!
72     obase=16; "hex="; $dec
73     obase=10; "dec="; $dec
74     obase=8;  "oct="; $dec
75     obase=2;  "bin="; $dec
76 !
77    ` | sed -e 's: : :g'
```

```
78
79    done
80 }
81
82 while [ $# -gt 0 ]
83 # ==>  这里必须使用一个 "while 循环",
84 # ==>+ 因为所有的 case 都可能退出循环或者
85 # ==>+ 结束脚本.
86 # ==> (感谢, Paulo Marcel Coelho Aragao.)
87 do
88    case "$1" in
89  --)    shift; break;;
90  -h)    Usage;;              # ==> 帮助信息.
91  -*)    Usage;;
92        *)    break;;   # 第一个数字
93    esac   # ==> 对于非法输入更严格检查是非常有用的.
94    shift
95 done
96
97 if [ $# -gt 0 ]
98 then
99    PrintBases "$@"
100 else    # 从标准输入中读取
101    while read line
102    do
103  PrintBases $line
104    done
105 fi
106
107
108 exit 0
##################################End Script##########################################
```

调用 bc 的另一种可选的方法就是使用 here document ,并把它嵌入到 命令替换 块中.

当一个脚本需要将一个选项列表和多个命令传递到 bc 中时, 这种方法就显得非常合适.

```
 1 variable=`bc << LIMIT_STRING
 2 options
 3 statements
 4 operations
 5 LIMIT_STRING
 6 `
 7
 8 ...or...
 9
10
11 variable=$(bc << LIMIT_STRING
12 options
13 statements
14 operations
```

```
 15 LIMIT_STRING
 16 )
```

Example 12-44 使用 "here document" 来调用 bc

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # 使用命令替换来调用 'bc'
 3 # 并与 'here document' 相结合.
 4
 5
 6 var1=`bc << EOF
 7 18.33 * 19.78
 8 EOF
 9 `
10 echo $var1       # 362.56
11
12
13 #  $( ... ) 这种标记法也可以.
14 v1=23.53
15 v2=17.881
16 v3=83.501
17 v4=171.63
18
19 var2=$(bc << EOF
20 scale = 4
21 a = ( $v1 + $v2 )
22 b = ( $v3 * $v4 )
23 a * b + 15.35
24 EOF
25 )
26 echo $var2       # 593487.8452
27
28
29 var3=$(bc -l << EOF
30 scale = 9
31 s ( 1.7 )
32 EOF
33 )
34 # 返回弧度为1.7的正弦.
35 # "-l" 选项将会调用 'bc' 算数库.
36 echo $var3       # .991664810
37
38
39 # 现在, 在函数中试一下...
40 hyp=            # 声明全局变量.
41 hypotenuse ()   # 计算直角三角形的斜边.
42 {
43 hyp=$(bc -l << EOF
44 scale = 9
```

```
45 sqrt ( $1 * $1 + $2 * $2 )
46 EOF
47 )
48 # 不幸的是, 不能从bash 函数中返回浮点值.
49 }
50
51 hypotenuse 3.68 7.31
52 echo "hypotenuse = $hyp"    # 8.184039344
53
54
55 exit 0
```

################################End Script###

Example 12-45 计算圆周率

################################Start Script###

```
 1 #!/bin/bash
 2 # cannon.sh: 通过开炮来取得近似的圆周率值.
 3
 4 # 这事实上是一个"Monte Carlo"蒙特卡洛模拟的非常简单的实例:
 5 #+ 蒙特卡洛模拟是一种由现实事件抽象出来的数学模型,
 6 #+ 由于要使用随机抽样统计来估算数学函数, 所以使用伪随机数来模拟真正的随机.
 7
 8 #  想象有一个完美的正方形土地, 边长为10000个单位.
 9 #  在这块土地的中间有一个完美的圆形湖,
10 #+ 这个湖的直径是10000个单位.
11 #  这块土地的绝大多数面积都是水, 当然只有4个角上有一些土地.
12 #  (可以把这个湖想象成为使这个正方形的内接圆.)
13 #
14 #  我们将使用老式的大炮和铁炮弹
15 #+ 向这块正方形的土地上开炮.
16 #  所有的炮弹都会击中这块正方形土地的某个地方.
17 #+ 或者是打到湖上, 或者是打到4个角的土地上.
18 #  因为这个湖占据了这个区域大部分地方,
19 #+ 所以大部分的炮弹都会"扑通"一声落到水里.
20 #  而只有很少的炮弹会"砰"的一声落到4个
21 #+ 角的土地上.
22 #
23 #  如果我们发出的炮弹足够随机的落到这块正方形区域中的话,
24 #+ 那么落到水里的炮弹与打出炮弹的总数的比率,
25 #+ 大概非常接近于 PI/4.
26 #
27 #  原因是所有的炮弹事实上都
28 #+ 打在了这个土地的右上角,
29 #+ 也就是, 笛卡尔坐标系的第一象限.
30 #  (之前的解释只是一个简化.)
31 #
32 #  理论上来说, 如果打出的炮弹越多, 就越接近这个数字.
33 #  然而, 对于shell 脚本来说一定会作些让步的,
34 #+ 因为它肯定不能和那些内建就支持浮点运算的编译语言相比.
```

```
35 #  当然就会降低精度.
36
37
38 DIMENSION=10000  # 这块土地的边长.
39               # 这也是所产生的随机整数的上限.
40
41 MAXSHOTS=1000    # 开炮次数.
42               # 10000 或更多次的话, 效果应该更好, 但有点太浪费时间了.
43 PMULTIPLIER=4.0  # 接近于 PI 的比例因子.
44
45 get_random ()
46 {
47 SEED=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }')
48 RANDOM=$SEED                              #  来自于 "seeding-random.sh"
49                                      #+ 的例子脚本.
50 let "rnum = $RANDOM % $DIMENSION"           #  范围小于 10000.
51 echo $rnum
52 }
53
54 distance=       # 声明全局变量.
55 hypotenuse ()   # 从 "alt-bc.sh" 例子来的,
56 {               # 计算直角三角形的斜边的函数.
57 distance=$(bc -l << EOF
58 scale = 0
59 sqrt ( $1 * $1 + $2 * $2 )
60 EOF
61 )
62 #  设置 "scale" 为 0 , 好让结果四舍五入为整数值,
63 #+ 这是这个脚本中必须折中的一个地方.
64 #  不幸的是, 这将降低模拟的精度.
65 }
66
67
68 # main() {
69
70 # 初始化变量.
71 shots=0
72 splashes=0
73 thuds=0
74 Pi=0
75
76 while [ "$shots" -lt  "$MAXSHOTS" ]         # 主循环.
77 do
78
79   xCoord=$(get_random)                    # 取得随机的 X 与 Y 坐标.
80   yCoord=$(get_random)
81   hypotenuse $xCoord $yCoord              #  直角三角形斜边 =
82                                      #+ distance.
```

```
 83   ((shots++))
 84
 85   printf "#%4d   " $shots
 86   printf "Xc = %4d  " $xCoord
 87   printf "Yc = %4d  " $yCoord
 88   printf "Distance = %5d  " $distance        #  到湖中心的
 89                                    #+ 距离 --
 90                                    #  起始坐标点 --
 91                                    #+  (0,0).
 92
 93   if [ "$distance" -le "$DIMENSION" ]
 94   then
 95     echo -n "SPLASH!  "
 96     ((splashes++))
 97   else
 98     echo -n "THUD!    "
 99     ((thuds++))
100   fi
101
102   Pi=$(echo "scale=9; $PMULTIPLIER*$splashes/$shots" | bc)
103   # 将比例乘以 4.0.
104   echo -n "PI ~ $Pi"
105   echo
106
107 done
108
109 echo
110 echo "After $shots shots, PI looks like approximately $Pi."
111 # 如果不太准的话, 那么就提高一下运行的次数. . .
112 # 可能是由于运行错误和随机数随机程度不高造成的.
113 echo
114
115 # }
116
117 exit 0
118
119 #  要想知道一个shell脚本到底适不适合作为
120 #+ 一种需要对复杂和精度都有要求的计算应用的模拟的话.
121 #
122 #  一般至少需要两个判断条件.
123 #  1) 作为一种概念的验证: 来显示它可以做到.
124 #  2) 在使用真正的编译语言来实现一个算法之前,
125 #+    使用脚本来测试和验证这个算法.
################################End Script#########################################
```

dc

dc (桌面计算器desk calculator) 工具是面向栈的并且使用 RPN (逆波兰表达式 "Reverse Polish Notation" 又叫"后缀表达式"). 与 bc 命令很相像 , 但是这个工具具备好多只有编程语言才具备的能力.

(译者注: 正常表达式 逆波兰表达式
a+b a,b,+
a+(b-c) a,b,c,-,+
a+(b-c)*d a,d,b,c,-,*,+
)

绝大多数人都避免使用这个工具, 因为它需要非直觉的 RPN 输入. 但是, 它却有特定的用途.

Example 12-46 将10进制数字转换为16进制数字

```
##################################Start Script#########################################
 1 #!/bin/bash
 2 # hexconvert.sh: 将10进制数字转换为16进制数字
 3
 4 E_NOARGS=65 # 缺命令行参数错误.
 5 BASE=16     # 16进制.
 6
 7 if [ -z "$1" ]
 8 then
 9   echo "Usage: $0 number"
10   exit $E_NOARGS
11   # 需要一个命令行参数.
12 fi
13 # 练习: 添加命令行参数检查.
14
15
16 hexcvt ()
17 {
18 if [ -z "$1" ]
19 then
20   echo 0
21   return    # 如果没有参数传递到这个函数中就 "return" 0.
22 fi
23
24 echo ""$1" "$BASE" o p" | dc
25 #               "o" 设置输出的基数(数制).
26 #                "p" 打印栈顶.
27 # 察看 dc 的 man 页来了解其他的选项.
28 return
29 }
30
31 hexcvt "$1"
32
33 exit 0
##################################End Script#########################################
```

通过仔细学习 dc 命令的 info 页, 可以更深入的理解这个复杂的命令. 但是, 有一些精通 dc巫术 的小组经常会炫耀他们使用这个强大而又晦涩难懂的工具时的一些技巧, 并以此为乐.

```
bash$ echo "16i[q]sa[ln0=aln100%Pln100/snlbx]sbA0D68736142snlbxq" | dc"
Bash
```

Example 12-47 因子分解

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # factr.sh: 分解约数
 3
 4 MIN=2      # 如果比这个数小就不行了.
 5 E_NOARGS=65
 6 E_TOOSMALL=66
 7
 8 if [ -z $1 ]
 9 then
10   echo "Usage: $0 number"
11   exit $E_NOARGS
12 fi
13
14 if [ "$1" -lt "$MIN" ]
15 then
16   echo "Number to factor must be $MIN or greater."
17   exit $E_TOOSMALL
18 fi
19
20 # 练习: 添加类型检查 (防止非整型的参数).
21
22 echo "Factors of $1:"
23 # ---------------------------------------------------------------------------
24 echo "$1[p]s2[lip/dli%0=1dvsr]s12sid2%0=13sidvsr[dli%0=1lrli2+dsi!>.]ds.xd1<2" | dc
25 # ---------------------------------------------------------------------------
26 # 上边这行代码是 Michel Charpentier 编写的<charpov@cs.unh.edu>.
27 # 在此使用经过授权 (thanks).
28
29  exit 0
################################End Script#########################################
```

awk

在脚本中使用浮点运算的另一种方法是使用 awk 内建的数学运算函数, 可以用在shell wrapper中.

Example 12-48 计算直角三角形的斜边

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # hypotenuse.sh: 返回直角三角形的斜边.
 3 #              ( 直角边长的平方和,然后对和取平方根)
 4
 5 ARGS=2              # 需要将2个直角边作为参数传递进来.
 6 E_BADARGS=65        # 错误的参数值.
 7
 8 if [ $# -ne "$ARGS" ] # 测试传递到脚本中的参数值.
 9 then
10   echo "Usage: `basename $0` side_1 side_2"
11   exit $E_BADARGS
```

```
12 fi
13
14
15 AWKSCRIPT=' { printf( "%3.7f\n", sqrt($1*$1 + $2*$2) ) } '
16 #             命令 / 传递给awk的参数
17
18
19 # 现在, 将参数通过管道传递给awk.
20 echo -n "Hypotenuse of $1 and $2 = "
21 echo $1 $2 | awk "$AWKSCRIPT"
22
23 exit 0
```

##################################End Script##

12.9 混杂命令

一些不好归类的命令

jot, seq

这些工具通过用户指定的范围和增量来产生一系列的整数.

每个产生出来的整数一般都占一行, 但是可以使用 -s 选项来改变这种设置.

```
bash$ seq 5
1
2
3
4
5
```

```
bash$ seq -s : 5
1:2:3:4:5
```

jot 和 seq 命令都经常用在 for 循环中.

Example 12-49 使用 seq 来产生循环参数

##################################Start Script###

```
1 #!/bin/bash
2 # 使用 "seq"
3
4 echo
5
6 for a in `seq 80`  # 或者   for a in $( seq 80 )
7 # 与  " for a in 1 2 3 4 5 ... 80 "相同  (少敲了好多字!).
8 # 也可以使用 'jot' (如果系统上有的话).
9 do
10   echo -n "$a "
11 done      # 1 2 3 4 5 ... 80
12 # 这也是一个通过使用命令的输出
13 # 来产生 "for"循环中 [list] 列表的例子.
14
15 echo; echo
```

```
16
17
18 COUNT=80  # 当然, 'seq' 也可以使用一个可替换的参数.
19
20 for a in `seq $COUNT`  # 或者   for a in $( seq $COUNT )
21 do
22   echo -n "$a "
23 done      # 1 2 3 4 5 ... 80
24
25 echo; echo
26
27 BEGIN=75
28 END=80
29
30 for a in `seq $BEGIN $END`
31 #  传给 "seq" 两个参数, 从第一个参数开始增长,
32 #+ 一直增长到第二个参数为止.
33 do
34   echo -n "$a "
35 done      # 75 76 77 78 79 80
36
37 echo; echo
38
39 BEGIN=45
40 INTERVAL=5
41 END=80
42
43 for a in `seq $BEGIN $INTERVAL $END`
44 #  传给 "seq" 三个参数从第一个参数开始增长,
45 #+ 并以第二个参数作为增量,
46 #+ 一直增长到第三个参数为止.
47 do
48   echo -n "$a "
49 done      # 45 50 55 60 65 70 75 80
50
51 echo; echo
52
53 exit 0
```

################################End Script###

一个简单些的例子:

```
 1 #  产生10个连续扩展名的文件,
 2 #+ 名字分别是 file.1, file.2 . . . file.10.
 3 COUNT=10
 4 PREFIX=file
 5
 6 for filename in `seq $COUNT`
 7 do
 8   touch $PREFIX.$filename
```

```
 9  # 或者, 你可以做一些其他的操作,
10  #+ 比如 rm, grep, 等等.
11 done
```

Example 12-50 字母统计

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # letter-count.sh: 统计一个文本文件中字母出现的次数.
 3 # 由 Stefano Palmeri 编写.
 4 # 经过授权使用在本书中.
 5 # 本书作者做了少许修改.
 6
 7 MINARGS=2         # 本脚本至少需要2个参数.
 8 E_BADARGS=65
 9 FILE=$1
10
11 let LETTERS=$#-1   # 制定了多少个字母 (作为命令行参数).
12                   # (从命令行参数的个数中减1.)
13
14
15 show_help(){
16     echo
17           echo Usage: `basename $0` file letters
18           echo Note: `basename $0` arguments are case sensitive.
19           echo Example: `basename $0` foobar.txt G n U L i N U x.
20     echo
21 }
22
23 # 检查参数个数.
24 if [ $# -lt $MINARGS ]; then
25    echo
26    echo "Not enough arguments."
27    echo
28    show_help
29    exit $E_BADARGS
30 fi
31
32
33 # 检查文件是否存在.
34 if [ ! -f $FILE ]; then
35     echo "File \"$FILE\" does not exist."
36     exit $E_BADARGS
37 fi
38
39
40
41 # 统计字母出现的次数.
42 for n in `seq $LETTERS`; do
43       shift
```

```
44      if [[ `echo -n "$1" | wc -c` -eq 1 ]]; then            # 检查参数.
45            echo "$1" -\> `cat $FILE | tr -cd  "$1" | wc -c` #  统计.
46      else
47            echo "$1 is not a  single char."
48      fi
49 done
50
51 exit $?
52
53 #  这个脚本在功能上与 letter-count2.sh 完全相同,
54 #+ 但是运行得更快.
55 #  为什么?
```

##################################End Script##

getopt

getopt 命令将会分析以破折号开头的命令行选项. 这个外部命令与Bash的内建命令 getopts 作用相同. 通过使用 -l 标志, getopt 可以处理长(多字符)选项, 并且也允许参数重置.

Example 12-51 使用getopt来分析命令行选项

##################################Start Script##

```
 1 #!/bin/bash
 2 # 使用 getopt.
 3
 4 # 尝试使用下边的不同的方法来调用这脚本:
 5 #   sh ex33a.sh -a
 6 #   sh ex33a.sh -abc
 7 #   sh ex33a.sh -a -b -c
 8 #   sh ex33a.sh -d
 9 #   sh ex33a.sh -dXYZ
10 #   sh ex33a.sh -d XYZ
11 #   sh ex33a.sh -abcd
12 #   sh ex33a.sh -abcdZ
13 #   sh ex33a.sh -z
14 #   sh ex33a.sh a
15 # 解释上面每一次调用的结果.
16
17 E_OPTERR=65
18
19 if [ "$#" -eq 0 ]
20 then   # 脚本需要至少一个命令行参数.
21   echo "Usage $0 -[options a,b,c]"
22   exit $E_OPTERR
23 fi
24
25 set -- `getopt "abcd:" "$@"`
26 # 为命令行参数设置位置参数.
27 # 如果使用 "$*" 来代替 "$@" 的话会发生什么?
28
29 while [ ! -z "$1" ]
```

```
30 do
31   case "$1" in
32     -a) echo "Option \"a\"";;
33     -b) echo "Option \"b\"";;
34     -c) echo "Option \"c\"";;
35     -d) echo "Option \"d\" $2";;
36      *) break;;
37   esac
38
39   shift
40 done
41
42 #  通常来说在脚本中使用内建的 'getopts' 命令,
43 #+ 会比使用 'getopt' 好一些.
44 #  参见 "ex33.sh".
45
46 exit 0
```

################################End Script###

参见 Example 9-12 , 这是对 getopt 命令的一个简单模拟.

run-parts

run-parts 命令 [1] 将会执行目标目录中所有的脚本, 这些将本会以 ASCII 的循序进行排列. 当然, 这些脚本都需要具有可执行权限.

cron 幽灵进程 会调用 run-parts 来运行 /etc/cron.* 下的所有脚本.

yes

yes 命令的默认行为是向 stdout 中连续不断的输出字符 y,每个y占一行.使用control-c来结束运行. 如果想换一个输出字符的话, 可以使用 yes 其他的字符串, 这样就会连续不同的输出你指定的字符串. 那么这样的命令究竟能做什么呢? 在命令行或者脚本中, yes的输出可以通过重定向或管道来传递给一些需要用户输入进行交互的命令. 事实上, 这个命令可以说是 expect 命令(译者注: 这个命令本书未介绍, 一个自动实现交互的命令)的一个简化版本.

yes | fsck /dev/hda1 将会以非交互的形式运行fsck(因为需要用户输入的 y 全由yes命令搞定了)(小心使用!).

yes | rm -r dirname 与 rm -rf dirname 效果相同(小心使用!).

注意: 当用 yes 的管道形式来使用一些可能具有潜在危险的系统命令的时候一定要深思熟虑, 比如 fsck 或 fdisk. 可能会产生一些意外的副作用.

banner

将会把字符串用一个 ASCII 字符(默认是 '#')来画出来(就是将多个'#'拼出一副字符的图形).可以作为硬拷贝重定向到打印机上(译者注: 可以使用-w 选项设置宽度).

printenv

对于某个特定的用户, 显示出所有的 环境变量.

```
bash$ printenv | grep HOME
HOME=/home/bozo
```

lp

lp 和 lpr 命令将会把文件发送到打印队列中, 并且作为硬拷贝来打印. [2] 这些命令会纪录它们名字的起始位置并传递到行打印机的另一个位置.<rojy bug>

bash$ lp file1.txt 或者 bash lp <file1.txt

通常情况下都是将pr的格式化的输出传递到 lp.

bash$ pr -options file1.txt | lp

格式化的包, 比如 groff 和 Ghostscript 就可以将它们的输出直接发送给 lp.

```
bash$ groff -Tascii file.tr | lp
bash$ gs -options | lp file.ps
```

还有一些相关的命令, 比如 lpq, 可以查看打印队列, lprm, 可以用来从打印队列中删除作业.

tee

[UNIX 从管道行业借来的主意.]

这是一个重定向操作, 但是有些不同. 就像管道中的"三通"一样, 这个命令可以将命令或者管道命令的输出抽出到一个文件中,而且并不影响结果. 当你想将一个正在运行的进程的输出保存到文件中时, 或者为了debug而保存输出记录的时候, 这个命令就非常有用了.

```
                    (重定向)
                   |----> to file
                   |
  =========================|====================
  command ---> command ---> |tee ---> command ---> ---> output of pipe
  ============================================
```

```
  1 cat listfile* | sort | tee check.file | uniq > result.file
```

(在对排序的结果进行 uniq (去掉重复行) 之前,文件 check.file 中保存了排过序的 "listfiles".)

mkfifo

这个不大引人注意的命令可以创建一个命名管道, 并产生一个临时的先进先出的buffer用来在两个进程间传输数据. [3] 典型的使用是一个进程向FIFO中写数据, 另一个进程读出来. 参见 Example A-15.

pathchk

这个命令用来检查文件名的有效性. 如果文件名超过了最大允许长度(255 个字符), 或者它所在的一个或多个路径搜索不到, 那么就会产生一个错误结果.

不幸的是,并不能够返回一个可识别的错误码, 因此它在脚本中几乎没有什么用. 一般都使用文件测试操作.

dd

这也是一个不太出名的工具, 但却是一个令人恐惧的 "数据复制" 命令. 最开始, 这个命令是被用来在UNIX 微机和IBM大型机之间通过磁带来交换数据, 这个命令现在仍然有它的用途. dd 命令只不过是简单的拷贝一个文件 (或者 stdin/stdout), 但是它会做一些转换. 下边是一些可能的转换, 比如 ASCII/EBCDIC, [4] 大写/小写, 在输入和输出之间的字节对的交换, 还有对输入文件做一些截头去尾的工作. dd --help 列出了所有转换, 还有这个强力工具的一些其他选项.

```
  1 # 将一个文件转换为大写:
  2
  3 dd if=$filename conv=ucase > $filename.uppercase
  4 #                 lcase   # 转换为小写
```

Example 12-52 一个拷贝自身的脚本

```
##################################Start Script#########################################
 1 #!/bin/bash
 2 # self-copy.sh
 3
 4 # 这个脚本将会拷贝自身.
 5
 6 file_subscript=copy
 7
```

```
 8 dd if=$0 of=$0.$file_subscript 2>/dev/null
 9 # 阻止dd产生的消息:           ^^^^^^^^^^
10
11 exit $?
```

################################End Script###

Example 12-53 练习dd

################################Start Script###

```
 1 #!/bin/bash
 2 # exercising-dd.sh
 3
 4 # 由Stephane Chazelas编写.
 5 # 本文作者做了少量修改.
 6
 7 input_file=$0   # 脚本本身.
 8 output_file=log.txt
 9 n=3
10 p=5
11
12 dd if=$input_file of=$output_file bs=1 skip=$((n-1)) count=$((p-n+1)) 2> /dev/null
13 # 从脚本中把位置n到p的字符提取出来.
14
15 # ------------------------------------------------------
16
17 echo -n "hello world" | dd cbs=1 conv=unblock 2> /dev/null
18 # 垂直的 echo "hello world" .
19
20 exit 0
```

################################End Script###

为了展示dd的多种用途, 让我们使用它来记录按键.

Example 12-54 记录按键

################################Start Script###

```
 1 #!/bin/bash
 2 # dd-keypress.sh: 记录按键, 不需要按回车.
 3
 4
 5 keypresses=4                 # 记录按键的个数.
 6
 7
 8 old_tty_setting=$(stty -g)       # 保存老的终端设置.
 9
10 echo "Press $keypresses keys."
11 stty -icanon -echo              # 禁用标准模式.
12                                 # 禁用本地 echo.
13 keys=$(dd bs=1 count=$keypresses 2> /dev/null)
14 # 如果不指定输入文件的话, 'dd' 使用标准输入.
15
16 stty "$old_tty_setting"          # 恢复老的终端设置.
17
```

```
18 echo "You pressed the \"$keys\" keys."
19
20 # 感谢 Stephane Chazelas, 演示了这种方法.
21 exit 0
```

################################End Script###

dd 命令可以在数据流上做随即存取.

```
  1 echo -n . | dd bs=1 seek=4 of=file conv=notrunc
  2 # "conv=notrunc" 选项意味着输出文件不能被截短.
  3
  4 # Thanks, S.C.
```

dd 命令可以将数据或磁盘镜像拷贝到设备中, 也可以从设备中拷贝数据或磁盘镜像, 比如说磁盘或磁带设备都可以 (Example A-5). 通常用来创建启动盘.

dd if=kernel-image of=/dev/fd0H1440

同样的, dd 可以拷贝软盘的整个内容(甚至是其他操作系统的磁盘格式) 到硬盘驱动器上(以镜像文件的形式).

dd if=/dev/fd0 of=/home/bozo/projects/floppy.img

dd 命令还有一些其他用途, 包括可以初始化临时交换文件 (Example 28-2) 和 ramdisks (内存虚拟硬盘) (Example 28-3). 它甚至可以做一些对整个硬盘分区的底层拷贝, 虽然不建议这么做.

一些(可能是比较无聊的)人总会想一些关于 dd 命令的有趣的应用.

Example 12-55 安全的删除一个文件

################################Start Script###

```
 1 #!/bin/bash
 2 # blot-out.sh: 删除一个文件所有的记录.
 3
 4 #  这个脚本会使用随即字节交替的覆盖
 5 #+ 目标文件, 并且在最终删除这个文件之前清零.
 6 #  这么做之后, 即使你通过传统手段来检查磁盘扇区
 7 #+ 也不能把文件原始数据重新恢复.
 8
 9 PASSES=7        #  破坏文件的次数.
10               #  提高这个数字会减慢脚本运行的速度,
11               #+ 尤其是对尺寸比较大的目标文件进行操作的时候.
12 BLOCKSIZE=1     #  带有 /dev/urandom 的 I/O 需要单位块尺寸,
13               #+ 否则你可能会获得奇怪的结果.
14 E_BADARGS=70    #  不同的错误退出码.
15 E_NOT_FOUND=71
16 E_CHANGED_MIND=72
17
18 if [ -z "$1" ]   # 没指定文件名.
19 then
20   echo "Usage: `basename $0` filename"
21   exit $E_BADARGS
22 fi
23
24 file=$1
25
26 if [ ! -e "$file" ]
```

```
27 then
28   echo "File \"$file\" not found."
29   exit $E_NOT_FOUND
30 fi
31
32 echo; echo -n "Are you absolutely sure you want to blot out \"$file\" (y/n)? "
33 read answer
34 case "$answer" in
35 [nN]) echo "Changed your mind, huh?"
36       exit $E_CHANGED_MIND
37       ;;
38 *)    echo "Blotting out file \"$file\".";;
39 esac
40
41
42 flength=$(ls -l "$file" | awk '{print $5}')  # 5 是文件长度.
43 pass_count=1
44
45 chmod u+w "$file"   # Allow overwriting/deleting the file.
46
47 echo
48
49 while [ "$pass_count" -le "$PASSES" ]
50 do
51   echo "Pass #$pass_count"
52   sync         # 刷新buffer.
53   dd if=/dev/urandom of=$file bs=$BLOCKSIZE count=$flength
54               # 使用随机字节进行填充.
55   sync         # 再刷新buffer.
56   dd if=/dev/zero of=$file bs=$BLOCKSIZE count=$flength
57               # 用0填充.
58   sync         # 再刷新buffer.
59   let "pass_count += 1"
60   echo
61 done
62
63
64 rm -f $file    # 最后, 删除这个已经被破坏得不成样子的文件.
65 sync           # 最后一次刷新buffer.
66
67 echo "File \"$file\" blotted out and deleted."; echo
68
69
70 exit 0
71
72 #  这是一种真正安全的删除文件的办法,
73 #+ 但是效率比较低, 运行比较慢.
74 #  GNU 的文件工具包中的 "shred" 命令,
```

75 #+ 也可以完成相同的工作, 不过更有效率.
76
77 # 使用普通的方法是不可能重新恢复这个文件了.
78 # 然而 . . .
79 #+ 这个简单的例子是不能够抵抗
80 #+ 那些经验丰富并且正规的分析.
81
82 # 这个脚本可能不会很好的运行在日志文件系统上.(译者注: JFS)
83 # 练习 (很难): 像它做的那样修正这个问题.
84
85
86
87 # Tom Vier的文件删除包可以更加彻底
88 #+ 的删除文件, 比这个简单的例子厉害得多.
89 # http://www.ibiblio.org/pub/Linux/utils/file/wipe-2.0.0.tar.bz2
90
91 # 如果想对安全删除文件这一论题进行深度的分析,
92 #+ 可以参见Peter Gutmann的页面,
93 #+ "Secure Deletion of Data From Magnetic and Solid-State Memory".
94 # http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html
##################################End Script###

od
od(octal dump)过滤器, 将会把输入(或文件)转换为8进制或者其他进制. 在你需要查看
或处理一些二进制数据文件或者一个不可读的系统设备文件的时候, 这个命令非常有用,
比如/dev/urandom,或者是一个二进制数据过滤器. 参见 Example 9-28 和
Example 12-13.

hexdump
对二进制文件进行 16进制, 8进制, 10进制, 或者 ASCII 码的查阅动作. 这个命令大体
上与上边的od命令作用相同, 但是远不及 od 命令有用.

objdump
显示编译后的2进制文件或2进制可执行文件的信息, 以16进制的形式显示, 或者显示反汇
编列表(使用-d选项).

```
bash$ objdump -d /bin/ls
/bin/ls:     file format elf32-i386
Disassembly of section .init:
080490bc <.init>:
 80490bc:       55                      push   %ebp
 80490bd:       89 e5                   mov    %esp,%ebp
 . . .
```

mcookie
这个命令会产生一个"magic cookie", 这是一个128-bit (32-字符) 的伪随机16进制数字,
这个数字一般都用来作为X server的鉴权"签名". 这个命令还可以用来在脚本中作为一
种生成随机数的手段, 当然这是一种"小吃店"(虽然不太正统, 但是很方便)的风格.

```
1 random000=$(mcookie)
```

当然, 完成同样的目的还可以使用 md5 命令.

```
1 # 产生关于脚本本身的 md5 checksum.
2 random001=`md5sum $0 | awk '{print $1}'`
3 # 使用 'awk' 来去掉文件名.
```

mcookie 还给出了产生"唯一"文件名的另一种方法.

Example 12-56 文件名产生器

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # tempfile-name.sh:  临时文件名产生器
 3
 4 BASE_STR=`mcookie`   # 32-字符的 magic cookie.
 5 POS=11              # 字符串中随便的一个位置.
 6 LEN=5               # 取得 $LEN 长度连续的字符串.
 7
 8 prefix=temp         #  最终的一个临时文件.
 9                     #  如果想让这个文件更加唯一,
10                     #+ 可以对这个前缀也使用下边的方法来生成.
11
12 suffix=${BASE_STR:POS:LEN}
13                     # 提取从第11个字符之后的长度为5的字符串.
14
15 temp_filename=$prefix.$suffix
16                     # 构造文件名.
17
18 echo "Temp filename = "$temp_filename""
19
20 # sh tempfile-name.sh
21 # Temp filename = temp.e19ea
22
23 #  与使用 'date' 命令(参考 ex51.sh)来创建唯一文件名
24 #+ 的方法相比较.
25
26 exit 0
################################End Script###########################################
```

units

这个工具用来在不同的计量单位之间互相转换. 当你在交互模式下正常调用时, 会发现在脚本中 units 也很有用.

Example 12-57 将米转换为英里

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # unit-conversion.sh
 3
 4
 5 convert_units ()  # 通过参数取得需要转换的单位.
 6 {
 7   cf=$(units "$1" "$2" | sed --silent -e '1p' | awk '{print $2}')
 8   # 除了真正需要转换的部分保留下来外,其他的部分都去掉.
 9   echo "$cf"
10 }
11
12 Unit1=miles
13 Unit2=meters
```

```
14 cfactor=`convert_units $Unit1 $Unit2`
15 quantity=3.73
16
17 result=$(echo $quantity*$cfactor | bc)
18
19 echo "There are $result $Unit2 in $quantity $Unit1."
20
21 #  如果你传递了两个不匹配的单位会发生什么?
22 #+ 比如分别传入英亩和英里?
23
24 exit 0
```

################################End Script###

m4

一个隐藏的财宝, m4 是一个强力的宏处理过滤器, [5] 差不多可以说是一种语言了. 虽然最开始这个工具是用来作为 RatFor 的预处理器而编写的, 但是后来证明 m4 作为独立的工具也是非常有用的. 事实上, m4 结合了许多工具的功能, 比如 eval, tr, 和 awk, 除此之外, 它还使得宏扩展变得容易.

在 2004年4月的 Linux Journal 的问题列表中有一篇关于 m4 命令用法得非常好的文章.

Example 12-58 使用 m4

################################Start Script###

```
 1 #!/bin/bash
 2 # m4.sh: 使用 m4 宏处理器
 3
 4 # 字符操作
 5 string=abcdA01
 6 echo "len($string)" | m4                    # 7
 7 echo "substr($string,4)" | m4                 # A01
 8 echo "regexp($string,[0-1][0-1],\&Z)" | m4        # 01Z
 9
10 # 算术操作
11 echo "incr(22)" | m4                        # 23
12 echo "eval(99 / 3)" | m4                     # 33
13
14 exit 0
```

################################End Script###

doexec

doexec 命令允许将一个随便的参数列表传递到一个二进制可执行文件中. 特别的, 甚至可以传递 arg[0] (相当于脚本中的 $0), 这样可以使用不同的名字来调用这个可执行文件, 并且通过不同的调用的名字, 可以让这个可执行文件执行不同的动作. 这也可以说是一种将参数传递到可执行文件中的比较绕圈子的做法.

比如, /usr/local/bin 目录可能包含一个 "aaa" 的二进制文件. 使用 doexec /usr/local/bin/aaa list 可以 列出 当前工作目录下所有以 "a" 开头的的文件, 而使用 doexec /usr/local/bin/aaa delete 将会删除这些文件.

注意: 可执行文件的不同行为必须定义在可执行文件自身的代码中, 可以使用如下的 shell脚本作类比:

```
 1 case `basename $0` in
 2 "name1" ) do_something;;
 3 "name2" ) do_something_else;;
```

```
  4 "name3" ) do_yet_another_thing;;
  5 *       ) bail_out;;
  6 esac
```

dialog

dialog 工具集提供了一种从脚本中调用交互对话框的方法. dialog 的更好的变种版本是 -- gdialog, Xdialog, 和 kdialog -- 事实上是调用的 X-Windows 的界面工具集. 参见 Example 33-19.

sox

sox 命令, "sound exchange" (声音转换)命令, 可以进行声音文件的转换. 事实上,可执行文件 /usr/bin/play (现在不建议使用) 只不过是 sox 的一个 shell 包装器而已. 举个例子, sox soundfile.wav soundfile.au 将会把一个 WAV 声音文件转换成一个 (Sun 音频格式) AU 声音文件.

Shell 脚本非常适合于使用 sox 的声音操作来批处理声音文件. 比如, 参见 Linux Radio Timeshift HOWTO 和 MP3do Project. 注意事项:

[1] 这个工具事实上是从 Debian Linux 发行版中的一个脚本借鉴过来的.

[2] 打印队列 就是"在线等待"打印的作业组.

[3] 对于本话题的一个完美的介绍, 请参见 Andy Vaught 的文章, 命名管道的介绍, (http://www2.linuxjournal.com/lj-issues/issue41/2156.html), 这是 Linux Journal (http://www.linuxjournal.com/)1997年9月的一个问题.

[4] EBCDIC (发音是 "ebb-sid-ick") 是单词 (Extended Binary Coded Decimal Interchange Code) 的首字母缩写. 这是 IBM 的数据格式, 现在已经不常见了. dd 命令的 conv=ebcdic 选项的一个比较奇异的使用方法是对一个文件进行快速而且容易但不太安全的编码.

```
  1 cat $file | dd conv=swab,ebcdic > $file_encrypted
  2 # 编码 (看起来好像没什么用).
  3 # 应该交换字节(swab), 有点晦涩.
  4
  5 cat $file_encrypted | dd conv=swab,ascii > $file_plaintext
  6 # 解码.
```

[5] 宏 是一个符号常量, 将会被扩展成一个命令字符串或者一系列的参数操作.

[]

高级Bash脚本编程指南(四)
文章整理: 文章来源: 网络
高级Bash脚本编程指南(四)

第13章 系统与管理命令

==========

在/etc/rc.d目录中的启动和关机脚本中包含了好多有用的(和没用的)这些系统管理命令. 这些命令通常总是被root用户使用, 用与系统维护或者是紧急文件系统修复.一定要小心使用这些工具, 因为如果滥用的话, 它们会损坏你的系统.

Users 和 Groups 类命令

users

显示所有的登录的用户. 这个命令与 who -q 基本一致.

groups

列出当前用户和他所属于的组. 这相当于 $GROUPS 内部变量, 但是这个命令将会给出组名字, 而不是数字.

```
bash$ groups
bozita cdrom cdwriter audio xgrp

bash$ echo $GROUPS
501
```

chown, chgrp

chown 命令将会修改一个或多个文件的所有权. 对于root来说这是一种非常好的将文件的所有权从一个用户换到另一个用户的方法. 一个普通用户不能修改文件的所有权, 即使他是文件的宿主也不行. [1]

```
root# chown bozo *.txt
```

chgrp 将会修改一个或个文件党组所有权. 你必须是这些文件的宿主, 并且是目的组的成员(或者root), 这样才能使用这个操作.

```
1 chgrp --recursive dunderheads *.data
2 #  "dunderheads"(译者: 晕,蠢才...) 组现在拥有了所有的"*.data"文件.
3 #+ 包括所有$PWD目录下的子目录中的文件(--recursive的作用就是包含子目录).
```

useradd, userdel

useradd 管理命令将会在系统上添加一个用户帐号, 并且如果指定的话, 还会为特定的用户创建home目录. 相应的userdel 命令将会从系统上删除一个用户帐号, [2] 并且删除相应的文件.

注意: adduser命令与useradd是相同的, adduser通常都是一个符号链接.

usermod

修改用户帐号. 可以修改密码, 组身份, 截止日期, 或者给定用户帐号的其他的属性. 使用这个命令, 用户的密码可能会被锁定, 因为密码会影响到帐号的有效性.

groupmod

修改指定组. 组名字或者ID号都可以使用这个命令来修改.

id

id 将会列出当前进程的真实和有效用户ID, 还有用户的组ID. 这与Bash的内部变量 $UID, $EUID, 和 $GROUPS 很相像.

```
bash$ id
uid=501(bozo) gid=501(bozo) groups=501(bozo),22(cdrom),80(cdwriter),81(audio)

bash$ echo $UID
501
```

注意: id 命令只有在有效ID与真实ID不符时才会显示有效id.

参见 Example 9-5.

who

显示系统上所有已经登录的用户.

```
bash$ who
bozo  tty1     Apr 27 17:45
bozo  pts/0    Apr 27 17:46
bozo  pts/1    Apr 27 17:47
bozo  pts/2    Apr 27 17:49
```

-m 选项将会只给出当前用户的详细信息. 将任意两个参数传递到who中 都等价于who -m, 就像 who am i 或者 who The Man.

```
bash$ who -m
localhost.localdomain!bozo  pts/2    Apr 27 17:49
```

whoami 与who -m 很相似, 但是只列出用户名.

```
bash$ whoami
```

bozo

w

显示所有的登录的用户和属于它们的进程. 这是一个who的扩展版本. w的输出可以通过管道传递到grep中, 这样就可以查找指定的用户或进程.

```
bash$ w | grep startx
bozo  tty1     -                4:22pm  6:41   4.47s  0.45s  startx
```

logname

显示当前用户的登录名(可以在/var/run/utmp中找到). 这与上边的whoami很相近.

```
bash$ logname
bozo

bash$ whoami
bozo
```

然而...

```
bash$ su
Password: ......

bash# whoami
root
bash# logname
bozo
```

注意: logname只会打印出登录的用户名, 而whoami 将会给出附着到当前进程的用户名. 就像我们上边看到的那样, 这两个名字有时会不同.

su

使用一个代替的用户来运行一个程序或脚本. su rjones 将会以 rjones 来启动一个 shell. 一个不加参数的su默认就是root. 参见 Example A-15.

sudo

以root(或其他用户)的身份来运行一个命令. 这个命令可以运行在脚本中, 这样就允许以正规的用户身份来运行脚本.

```
1 #!/bin/bash
2
3 # 一些命令.
4 sudo cp /root/secretfile /home/bozo/secret
5 # 一些命令.
```

文件 /etc/sudoers 持有允许调用sudo的用户名.

passwd

设置, 修改, 或者管理用户的密码.

passwd 命令可以用在脚本中, 但可能你不想这么用.

Example 13-1 设置一个新密码

```
#################################Start Script#########################################
1 #!/bin/bash
2 #  setnew-password.sh: 只用于说明目的.
3 #                  如果真正运行这个脚本并不是一个好主意.
4 #  这个脚本必须以root身份运行.
5
6 ROOT_UID=0        # Root 的 $UID 0.
7 E_WRONG_USER=65    # 不是 root?
8
```

```
 9 E_NOSUCHUSER=70
10 SUCCESS=0
11
12
13 if [ "$UID" -ne "$ROOT_UID" ]
14 then
15   echo; echo "Only root can run this script."; echo
16   exit $E_WRONG_USER
17 else
18   echo
19   echo "You should know better than to run this script, root."
20   echo "Even root users get the blues... "
21   echo
22 fi
23
24
25 username=bozo
26 NEWPASSWORD=security_violation
27
28 # 检查bozo是否在这里.
29 grep -q "$username" /etc/passwd
30 if [ $? -ne $SUCCESS ]
31 then
32   echo "User $username does not exist."
33   echo "No password changed."
34   exit $E_NOSUCHUSER
35 fi
36
37 echo "$NEWPASSWORD" | passwd --stdin "$username"
38 #  'passwd'命令 '--stdin' 选项允许
39 #+ 从stdin(或者管道)中获得一个新的密码.
40
41 echo; echo "User $username's password changed!"
42
43 # 在脚本中使用'passwd'命令是很危险的.
44
45 exit 0
```

################################End Script###

passwd 命令的 -l, -u, 和 -d 选项允许锁定, 解锁,和删除一个用户的密码. 只有root
用户可以使用这些选项.

ac

显示用户登录的连接时间, 就像从 /var/log/wtmp 中读取一样. 这是GNU的一个统计工具.

```
 bash$ ac
   total       68.08
```

last

用户最后登录的信息, 就像从/var/log/wtmp中读出来一样. 这个命令也可以用来显示远
端登录.
比如, 显示最后几次系统的重启信息:

```
bash$ last reboot
reboot   system boot  2.6.9-1.667     Fri Feb  4 18:18         (00:02)
reboot   system boot  2.6.9-1.667     Fri Feb  4 15:20         (01:27)
reboot   system boot  2.6.9-1.667     Fri Feb  4 12:56         (00:49)
reboot   system boot  2.6.9-1.667     Thu Feb  3 21:08          (02:17)
. . .
wtmp begins Tue Feb  1 12:50:09 2005
```

newgrp

不用登出就可以修改用户的组ID. 并且允许存取新组的文件. 因为用户可能同时属于多个组, 这个命令很少被使用.

终端类命令

tty

显示当前用户终端的名字. 注意每一个单独的xterm窗口都被算作一个不同的终端.

```
bash$ tty
/dev/pts/1
```

stty

显示并(或)修改终端设置. 这个复杂命令可以用在脚本中, 并可以用来控制终端的行为和其显示输出的方法. 参见这个命令的info页, 并仔细学习它.

Example 13-2 设置一个擦除字符

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # erase.sh: 在读取输入时使用"stty"来设置一个擦除字符.
 3
 4 echo -n "What is your name? "
 5 read name                  #  试试退格键
 6                           #+ 来删除输入的字符.
 7                           #  有什么问题?
 8 echo "Your name is $name."
 9
10 stty erase '#'             #  将 "hashmark" (#) 设置为退格字符.
11 echo -n "What is your name? "
12 read name                  #  使用#来删除最后键入的字符.
13 echo "Your name is $name."
14
15 # 警告: 即使在脚本退出后, 新的键值还是保持设置.(译者: 使用stty erase '^?' 恢复)
16
17 exit 0
################################End Script#########################################
```

Example 13-3 关掉终端对于密码的echo

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # secret-pw.sh: 保护密码不被显示
 3
 4 echo
 5 echo -n "Enter password "
 6 read passwd
 7 echo "password is $passwd"
 8 echo -n "If someone had been looking over your shoulder, "
```

```
 9 echo "your password would have been compromised."
10
11 echo && echo  # 在一个"与列表"中产生2个换行.
12
13
14 stty -echo    # 关闭屏幕的echo.
15
16 echo -n "Enter password again "
17 read passwd
18 echo
19 echo "password is $passwd"
20 echo
21
22 stty echo     # 恢复屏幕的echo.
23
24 exit 0
25
26 # 详细的阅读stty命令的info页, 以便于更好的掌握这个有用并且狡猾的工具.
```

################################End Script###

一个具有创造性的stty命令的用法, 检测用户所按的键(不用敲回车).

Example 13-4 按键检测

################################Start Script###

```
 1 #!/bin/bash
 2 # keypress.sh: 检测用户按键 ("hot keys").
 3
 4 echo
 5
 6 old_tty_settings=$(stty -g)   # 保存老的设置(为什么?).
 7 stty -icanon
 8 Keypress=$(head -c1)         # 或者 $(dd bs=1 count=1 2> /dev/null)
 9                              # 在非GNU的系统上
10
11 echo
12 echo "Key pressed was \""$Keypress"\"."
13 echo
14
15 stty "$old_tty_settings"      # 恢复老的设置.
16
17 # 感谢, Stephane Chazelas.
18
19 exit 0
```

################################End Script###

参见 Example 9-3.

注意: 终端与模式terminals and modes

一般情况下, 一个终端都是工作在canonical(标准)模式下. 当用户按键后, 事实上所产生的字符并没有马上传递到运行在当前终端上的程序. 终端上的一个本地的缓存保存了这些按键. 当用按下ENTER键的时候, 才会将所有保存的按键信息传递到运行的程序中. 这就意味着在终端内部存在一个基本的行编辑器.

bash$ stty -a
speed 9600 baud; rows 36; columns 96; line = 0;
intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D; eol = <undef>; eol2 = <undef>;
start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O;
...
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt

在使用canonical模式的时候, 可以对本地终端行编辑器所定义的特殊按键进行重新定义.

bash$ cat > filexxx
wha<ctl-W>I<ctl-H>foo bar<ctl-U>hello world<ENTER>
<ctl-D>
bash$ cat filexxx
hello world
bash$ wc -c < filexxx
12

控制终端的进程只保存了12个字符(11个字母加上一个换行), 虽然用户敲了26个按键.

在 non-canonical ("raw") 模式, 每次按键(包括特殊定义的按键, 比如 ctl-H)将会立即发送一个字符到控制进程.

Bash提示符禁用了icanon和echo, 因为它用自己的更好的行编辑器代替了终端的基本行编辑器. 比如, 当你在Bash提示符下敲ctl-A的时候, 终端将不会显示 ^A, 但是Bash将会获得\1字符, 然后解释这个字符, 这样光标就移动到行首了.

St é phane Chazelas

setterm

设置特定的终端属性. 这个命令将向它的终端的stdout写一个字符串, 这个字符串将修改终端的行为.

bash$ setterm -cursor off
bash$

setterm 命令可以被用在脚本中来修改写到stdout的文本的外观, 虽然如果你仅仅只想完成这个目的, 还有特定的更好的工具可以用.

```
1 setterm -bold on
2 echo bold hello
3
4 setterm -bold off
5 echo normal hello
```

tset

显示或初始化终端设置. 可以说这是stty的功能比较弱的版本.

bash$ tset -r
Terminal type is xterm-xfree86.
Kill is control-U (^U).
Interrupt is control-C (^C).

setserial

设置或者显示串口参数. 这个脚本只能被root用户来运行, 并且通常都在系统安装脚本中使用.

```
1 # 来自于 /etc/pcmcia/serial 脚本:
2
3 IRQ=`setserial /dev/$DEVICE | sed -e 's/.*IRQ: //'`
```

4 setserial /dev/$DEVICE irq 0 ; setserial /dev/$DEVICE irq $IRQ

getty, agetty

一个终端的初始化过程通常都是使用getty或agetty来建立, 这样才能让用户登录. 这些命令并不用在用户的shell脚本中. 它们的行为与stty很相似.

mesg

使能或禁用当前用户终端的存取权限. 禁用存取权限将会阻止网络上的另一用户向这个终端写消息.

注意: 当你正在编写文本文件的时候, 在文本中间突然来了一个莫名其妙的消息, 这对你来说是非常烦人的. 在多用户的网络环境下, 当你不想被打断的时候, 你可能因此希望禁用对你终端的写权限.

wall

这是一个缩写单词 "write all", 也就是, 向登录到网络上的任何终端的所有用户都发送一个消息. 最早这是一个管理员的工具, 很有用, 比如, 当系统有问题的时候, 管理可以警告系统上的所有人暂时离开 (参见 Example 17-1).

```
bash$ wall System going down for maintenance in 5 minutes!
Broadcast message from bozo (pts/1) Sun Jul  8 13:53:27 2001...
System going down for maintenance in 5 minutes!
```

注意: 如果某个特定终端使用mesg来禁止了写权限, 那么wall将不会给它发消息.

信息与统计类

uname

输出系统的说明(OS, 内核版本, 等等.)到stdout. 使用 -a 选项, 将会给出详细的信息(参见 Example 12-5). 使用-s选项只会输出OS类型.

```
bash$ uname -a
Linux localhost.localdomain 2.2.15-2.5.0 #1 Sat Feb 5 00:13:43 EST 2000 i686 unknown

bash$ uname -s
Linux
```

arch

显示系统的硬件体系结构. 等价于 uname -m. 参见 Example 10-26.

```
bash$ arch
i686

bash$ uname -m
i686
```

lastcomm

给出前一个命令的信息, 存储在/var/account/pacct文件中. 命令名字与用户名字都可以使用选项来指定. 这是GNU的一个统计工具.

lastlog

列出系统上所有用户最后登录的时间. 存在/var/log/lastlog文件中.

```
bash$ lastlog
root         tty1                 Fri Dec  7 18:43:21 -0700 2001
bin                               **Never logged in**
daemon                            **Never logged in**
...
bozo         tty1                 Sat Dec  8 21:14:29 -0700 2001
```

```
bash$ lastlog | grep root
root         tty1                      Fri Dec  7 18:43:21 -0700 2001
```

注意: 如果用户对于/var/log/lastlog文件没有读权限的话, 那么调用这个命令就会失败.

lsof

列出打开的文件. 这个命令将会把所有当前打开的文件列出一份详细的表格, 包括文件的所有者信息, 尺寸, 与它们相关的信息等等. 当然, lsof也可以管道输出到 grep 和(或) awk来分析它的结果.

```
bash$ lsof
COMMAND    PID    USER   FD   TYPE     DEVICE    SIZE     NODE NAME
init         1    root  mem    REG       3,5   30748    30303 /sbin/init
init         1    root  mem    REG       3,5   73120     8069 /lib/ld-2.1.3.so
init         1    root  mem    REG       3,5  931668     8075 /lib/libc-2.1.3.so
cardmgr    213    root  mem    REG       3,5   36956    30357 /sbin/cardmgr
...
```

strace

为了跟踪系统和信号的诊断和调试工具. 调用它最简单的方法就是strace COMMAND.

```
bash$ strace df
execve("/bin/df", ["df"], [/* 45 vars */]) = 0
uname({sys="Linux", node="bozo.localdomain", ...}) = 0
brk(0)                                  = 0x804f5e4
...
```

这是 Solaris truss命令的Linux的等价工具.

nmap

网络端口扫描器. 这个命令将会扫描一个服务器来定位打开的端口, 并且定位这些端口相关的服务. 这是一个防止网络被黑客入侵的一个重要的安全工具.

```
  1 #!/bin/bash
  2
  3 SERVER=$HOST                     # localhost.localdomain (127.0.0.1).
  4 PORT_NUMBER=25                    # SMTP 端口.
  5
  6 nmap $SERVER | grep -w "$PORT_NUMBER"  # 这个指定端口打开了么?
  7 #             grep -w 匹配整个单词,
  8 #+            这样就不会匹配类似于1025这种含有25的端口了.
  9
 10 exit 0
 11
 12 # 25/tcp     open        smtp
```

nc

nc(netcat)工具是一个完整的工具包, 可以使用它来连接和监听TCP和UDP端口. 它可以用来作为诊断和测试工具, 也可以用来作为基于脚本的HTTP客户端和服务器的组件.

```
bash$ nc localhost.localdomain 25
220 localhost.localdomain ESMTP Sendmail 8.13.1/8.13.1; Thu, 31 Mar 2005 15:41:35 -0700
```

Example 13-5 Checking a remote server for identd<rojy bug>

```
################################Start Script#########################################
 1 #! /bin/sh
 2 ## Duplicate DaveG's ident-scan thingie using netcat. Oooh, he'll be p*ssed.
 3 ## Args: target port [port port port ...]
 4 ## Hose stdout _and_ stderr together.
```

```
 5 ##
 6 ##  优点: runs slower than ident-scan, giving remote inetd less cause
 7 ##+ for alarm, and only hits the few known daemon ports you specify.
 8 ##  缺点: requires numeric-only port args, the output sleazitude,
 9 ##+ and won't work for r-services when coming from high source ports.
10 # 脚本作者: Hobbit <hobbit@avian.org>
11 # 授权使用在本书中.
12
13 # ------------------------------------------------
14 E_BADARGS=65      # 至少需要两个参数.
15 TWO_WINKS=2       # 需要睡多长时间.
16 THREE_WINKS=3
17 IDPORT=113        # Authentication "tap ident" port.
18 RAND1=999
19 RAND2=31337
20 TIMEOUT0=9
21 TIMEOUT1=8
22 TIMEOUT2=4
23 # ------------------------------------------------
24
25 case "${2}" in
26   "" ) echo "Need HOST and at least one PORT." ; exit $E_BADARGS ;;
27 esac
28
29 # Ping 'em once and see if they *are* running identd.
30 nc -z -w $TIMEOUT0 "$1" $IDPORT || { echo "Oops, $1 isn't running identd." ; exit 0 ; }
31 #  -z scans for listening daemons.
32 #     -w $TIMEOUT = How long to try to connect.
33
34 # Generate a randomish base port.
35 RP=`expr $$ % $RAND1 + $RAND2`
36
37 TRG="$1"
38 shift
39
40 while test "$1" ; do
41   nc -v -w $TIMEOUT1 -p ${RP} "$TRG" ${1} < /dev/null > /dev/null &
42   PROC=$!
43   sleep $THREE_WINKS
44   echo "${1},${RP}" | nc -w $TIMEOUT2 -r "$TRG" $IDPORT 2>&1
45   sleep $TWO_WINKS
46
47 # 这个脚本看起来是不是一个瘸腿脚本, 或者其它更差的什么东西?
48 # ABS Guide 作者注释: "并不是真的那么差,
49 #+                    事实上相当清楚."
50
51   kill -HUP $PROC
52   RP=`expr ${RP} + 1`
```

```
53   shift
54 done
55
56 exit $?
57
58 #  注意事项:
59 #  ---------
60
61 #  尝试注释一下第30行的程序, 并且使用"localhost.localdomain 25"
62 #+ 作为参数来运行这个脚本.
63
64 #  For more of Hobbit's 'nc' example scripts,
65 #+ look in the documentation:
66 #+ the /usr/share/doc/nc-X.XX/scripts directory.
```

##################################End Script###

并且, 当然, 这里还有Dr. Andrew Tridgell在BistKeeper事件中臭名卓著的一行脚本:

```
   1 echo clone | nc thunk.org 5000 > e2fsprogs.dat
```

free

使用表格形式来显示内存和缓存的使用情况. 这个命令的输出非常适合于使用 grep, awk 或者Perl来分析. procinfo命令将会显示free命令所能显示的所有信息, 而且更多.

```
 bash$ free
     total       used       free     shared    buffers     cached
   Mem:         30504      28624       1880      15820       1608      16376
   -/+ buffers/cache:      10640      19864
   Swap:        68540       3128      65412
```

显示未使用的RAM内存:

```
 bash$ free | grep Mem | awk '{ print $4 }'
 1880
```

procinfo

从/proc pseudo-filesystem中提取和显示所有信息和统计资料. 这个命令将给出更详细的信息.

```
 bash$ procinfo | grep Bootup
 Bootup: Wed Mar 21 15:15:50 2001    Load average: 0.04 0.21 0.34 3/47 6829
```

lsdev

显示设备, 也就是显示安装的硬件.

```
 bash$ lsdev
 Device           DMA   IRQ  I/O Ports
 ------------------------------------------------
 cascade           4     2
 dma                      0080-008f
 dma1                     0000-001f
 dma2                     00c0-00df
 fpu                    00f0-00ff
 ide0                14  01f0-01f7 03f6-03f6
 ...
```

du

递归的显示(磁盘)文件的使用状况. 除非指定, 默认是当前工作目录.

```
 bash$ du -ach
```

```
  1.0k    ./wi.sh
  1.0k    ./tst.sh
  1.0k    ./random.file
  6.0k    .
  6.0k    total
```

df

使用列表的形式显示文件系统的使用状况.

```
  bash$ df
  Filesystem           1k-blocks      Used Available Use% Mounted on
  /dev/hda5              273262     92607    166547  36% /
  /dev/hda8              222525    123951     87085  59% /home
  /dev/hda7             1408796   1075744    261488  80% /usr
```

dmesg

将所有的系统启动消息输出到stdout上. 方便出错,并且可以查出安装了哪些设备驱动和察看使用了哪些系统中断. dmesg命令的输出当然也可以在脚本中使用 grep, sed, 或 awk 来进行分析.

```
  bash$ dmesg | grep hda
  Kernel command line: ro root=/dev/hda2
  hda: IBM-DLGA-23080, ATA DISK drive
  hda: 6015744 sectors (3080 MB) w/96KiB Cache, CHS=746/128/63
  hda: hda1 hda2 hda3 < hda5 hda6 hda7 > hda4
```

stat

显示一个或多个给定文件(也可以是目录文件或设备文件)的详细的统计信息.

```
  bash$ stat test.cru
    File: "test.cru"
    Size: 49970       Allocated Blocks: 100         Filetype: Regular File
    Mode: (0664/-rw-rw-r--)       Uid: (  501/ bozo)  Gid: (  501/ bozo)
  Device:  3,8   Inode: 18185     Links: 1
  Access: Sat Jun  2 16:40:24 2001
  Modify: Sat Jun  2 16:40:24 2001
  Change: Sat Jun  2 16:40:24 2001
```

如果目标文件不存在, stat 将会返回一个错误信息.

```
  bash$ stat nonexistent-file
  nonexistent-file: No such file or directory
```

vmstat

显示虚拟内存的统计信息.

```
  bash$ vmstat
  procs                  memory    swap         io system         cpu
  r  b  w   swpd   free   buff  cache  si  so    bi    bo   in    cs  us  sy id
  0  0  0      0  11040   2636  38952   0   0    33     7  271    88   8   3 89
```

netstat

显示当前网络的统计和信息, 比如路由表和激活的连接. 这个工具存取/proc/net(第27章)中的信息. 参见 Example 27-3.

netstat -r 等价于 route 命令.

```
  bash$ netstat
  Active Internet connections (w/o servers)
  Proto Recv-Q Send-Q Local Address           Foreign Address         State
  Active UNIX domain sockets (w/o servers)
```

```
Proto RefCnt Flags       Type       State         I-Node Path
unix  11     [ ]         DGRAM                    906    /dev/log
unix  3      [ ]         STREAM     CONNECTED     4514   /tmp/.X11-unix/X0
unix  3      [ ]         STREAM     CONNECTED     4513
. . .
```

uptime

显示系统运行的时间, 还有其他一些统计信息.

```
bash$ uptime
10:28pm  up  1:57,  3 users,  load average: 0.17, 0.34, 0.27
```

注意: load average 如果小于或等于1, 那么就意味着系统会马上处理. 如果 load average大于1, 那么就意味着进程需要排队. 如果load average大于3, 那么就意味着, 系统性能已经显著下降了.

hostname

显示系统的主机名字. 这个命令在 /etc/rc.d 安装脚本(/etc/rc.d/rc.sysinit 或类似的)中设置主机名. 等价于uname -n, 并且与$HOSTNAME内部变量很相像.

```
bash$ hostname
localhost.localdomain

bash$ echo $HOSTNAME
localhost.localdomain
```

与 hostname 命令很相像的命令还有 domainname, dnsdomainname, nisdomainname, 和 ypdomainname 命令. 使用这些来显示或设置系统DNS 或者 NIS/YP 域名. 对于hostname 命令来说使用不同的选项一样可以达到上边这些命令的目的.

hostid

显示主机的32位的16进制ID.

```
bash$ hostid
7f0100
```

注意: 这个命令据说对于特定系统可以获得一个"唯一"的序号. 某些产品的注册过程可能会需要这个序号来作为用户的许可证. 不幸的是, hostid 只会使用字节转换的方法来用16进制显示机器的网络地址.

一个没有网络的Linux机器的典型的网络地址设置在/ect/hosts中.

```
bash$ cat /etc/hosts
127.0.0.1               localhost.localdomain localhost
```

碰巧, 通过对127.0.0.1进行字节转换, 我们获得了 0.127.1.0, 用16进制表示就是 007f0100, 这就是上边hostid返回的结果. 这样几乎所有的无网络的Linux机器都会得到这个hostid.

sar

sar (System Activity Reporter系统活动报告) 命令将会给出系统统计的一个非常详细的概要. Santa Cruz Operation("老" SCO)公司在1999年4月份以开源软件的形式发布了sar. 这个命令并不是基本Linux发行版的一部分, 但是你可以从Sebastien Godard 写的 sysstat utilities 包中获得这个工具.

```
bash$ sar
Linux 2.4.9 (brooks.seringas.fr)  09/26/03
10:30:00        CPU     %user     %nice   %system   %iowait     %idle
10:40:00        all      2.21     10.90     65.48      0.00     21.41
10:50:00        all      3.36      0.00     72.36      0.00     24.28
11:00:00        all      1.12      0.00     80.77      0.00     18.11
Average:        all      2.23      3.63     72.87      0.00     21.27
```

```
14:32:30        LINUX RESTART
15:00:00        CPU     %user     %nice   %system   %iowait     %idle
15:10:00        all      8.59      2.40     17.47      0.00     71.54
15:20:00        all      4.07      1.00     11.95      0.00     82.98
15:30:00        all      0.79      2.94      7.56      0.00     88.71
Average:        all      6.33      1.70     14.71      0.00     77.26
```

readelf

显示指定的 elf 格式的2进制文件的统计信息. 这个工具是binutils工具包的一部分.

```
bash$ readelf -h /bin/bash
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  . . .
```

size

size [/path/to/binary] 命令可以显示2进制可执行文件或归档文件每部分的尺寸. 这个工具主要是程序员使用.

```
bash$ size /bin/bash
text    data     bss     dec     hex filename
 495971   22496   17392  535859   82d33 /bin/bash
```

系统日志类

logger

附加一个用户产生的消息到系统日之中 (/var/log/messages). 不是root用户也可以调用 logger.

```
  1 logger Experiencing instability in network connection at 23:10, 05/21.
  2 # 现在, 运行 'tail /var/log/messages'.
```

通过在脚本中调用一个logger命令, 就可以将调试信息写到/var/log/messages中.

```
  1 logger -t $0 -i Logging at line "$LINENO".
  2 # "-t" 选项可以为长的入口指定标签.
  3 # "-i" 选项记录进程ID.
  4
  5 # tail /var/log/message
  6 # ...
  7 # Jul  7 20:48:58 localhost ./test.sh[1712]: Logging at line 3.
```

logrotate

这个工具用来管理系统的log文件, 可以在合适的时候轮换, 压缩, 删除, 和(或)e-mail 它们. 这个工具将从老的log文件中取得一些杂乱的记录保存在/var/log中. 通常使用 cron 来每天运行logrotate.

在/etc/logrotate.conf中添加合适的入口就可以管理自己的log文件了, 就像管理系统 log文件一样.

注意: Stefano Falsetto 创造了rottlog, 他认为这是logrotate的改进版本.

作业控制

ps

进程统计: 通过进程所有者和PID(进程ID)来列出当前执行的进程. 通常都是使用ax选项

来调用这个命令, 并且结果可以通过管道传递到 grep 或 sed 中来搜索特定的进程 (参见 Example 11-12 和 Example 27-2).

```
bash$  ps ax | grep sendmail
295 ?    S    0:00 sendmail: accepting connections on port 25
```

如果想使用"树"的形式来显示系统进程: ps afjx 或者 ps ax --forest.

pgrep, pkill

ps 命令与grep或kill结合使用.

```
bash$ ps a | grep mingetty
 2212 tty2     Ss+    0:00 /sbin/mingetty tty2
 2213 tty3     Ss+    0:00 /sbin/mingetty tty3
 2214 tty4     Ss+    0:00 /sbin/mingetty tty4
 2215 tty5     Ss+    0:00 /sbin/mingetty tty5
 2216 tty6     Ss+    0:00 /sbin/mingetty tty6
 4849 pts/2    S+     0:00 grep mingetty
 bash$ pgrep mingetty
 2212 mingetty
 2213 mingetty
 2214 mingetty
 2215 mingetty
 2216 mingetty
```

pstree

使用"树"形式列出当前执行的进程. -p选项显示PID,和进程名字.

top

连续不断的显示cpu使用率最高的进程. -b 选项将会以文本方式显示, 以便于可以在脚本中分析或存取.

```
bash$ top -b
  8:30pm  up 3 min,  3 users,  load average: 0.49, 0.32, 0.13
 45 processes: 44 sleeping, 1 running, 0 zombie, 0 stopped
 CPU states: 13.6% user,  7.3% system,  0.0% nice, 78.9% idle
 Mem:    78396K av,   65468K used,   12928K free,       0K shrd,    2352K buff
 Swap:  157208K av,       0K used,  157208K free                37244K cached
  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
  848 bozo      17   0   996  996   800 R     5.6  1.2   0:00 top
  1 root       8   0   512  512   444 S     0.0  0.6   0:04 init
  2 root       9   0     0    0     0 SW    0.0  0.0   0:00 keventd
  ...
```

nice

使用修改后的优先级来运行一个后台作业. 优先级从19(最低)到-20(最高). 只有root用户可以设置负的(比较高的)优先级. 相关的命令是renice, snice, 和skill.

nohup

保持一个命令的运行, 即使用户登出系统. 这个命令做为前台进程来运行, 除非前边加 &. 如果你在脚本中使用nohup命令, 最好和wait 命令一起使用, 这样可以避免创建一个孤儿进程或僵尸进程.

pidof

取得一个正在运行的作业的进程ID(PID). 因为一些作业控制命令, 比如kill和renice只能使用进程的PID(而不是它的名字), 所以有时候必须的取得PID. pidof命令与$PPID内部变量非常相似.

```
bash$ pidof xclock
```

880

Example 13-6 pidof 帮助杀掉一个进程

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # kill-process.sh
 3
 4 NOPROCESS=2
 5
 6 process=xxxyyyzzz  # 使用不存在的进程.
 7 # 只不过是为了演示...
 8 # ... 并不想在这个脚本中杀掉任何真正的进程.
 9 #
10 # 如果, 举个例子, 你想使用这个脚本来断线Internet,
11 #     process=pppd
12
13 t=`pidof $process`       # 取得$process的pid(进程id).
14 # 'kill'必须使用pid(不能用程序名).
15
16 if [ -z "$t" ]          # 如果没这个进程, 'pidof' 返回空.
17 then
18   echo "Process $process was not running."
19   echo "Nothing killed."
20   exit $NOPROCESS
21 fi
22
23 kill $t                 # 对于顽固的进程可能需要'kill -9'.
24
25 # 这里需要做一个检查, 看看进程是否允许自身被kill.
26 # 或许另一个 " t=`pidof $process` " 或者 ...
27
28
29 # 整个脚本都可以使用下边这句来替换:
30 #    kill $(pidof -x process_name)
31 # 但是这就没有教育意义了.
32
33 exit 0
################################End Script###########################################
```

fuser

取得一个正在存取某个或某些文件(或目录)的进程ID. 使用-k选项将会杀掉这些进程. 对于系统安全来说, 尤其是在脚本中想阻止未被授权的用户存取系统服务的时候, 这个命令就显得很有用了.

```
bash$ fuser -u /usr/bin/vim
/usr/bin/vim:        3207e(bozo)

bash$ fuser -u /dev/null
/dev/null:           3009(bozo)  3010(bozo)  3197(bozo)  3199(bozo)
```

当正常的插入或删除保存的媒体, 比如CD ROM或者USB闪存设备的时候, fuser的应用也显得特别重要. 有时候当你想umount一个设备失败的时候(出现设备忙的错误消息), 这意味

着某些用户或进程正在存取这个设备. 使用fuser -um /dev/device_name可以搞定这些, 这样你就可以杀掉所有相关的进程.

```
bash$ umount /mnt/usbdrive
umount: /mnt/usbdrive: device is busy

bash$ fuser -um /dev/usbdrive
/mnt/usbdrive:        1772c(bozo)

bash$ kill -9 1772
bash$ umount /mnt/usbdrive
```

fuser 的-n选项可以获得正在存取某一端口的进程. 当和nmap命令组合使用的时候尤其有用.

```
root# nmap localhost.localdomain
PORT     STATE SERVICE
25/tcp   open  smtp

root# fuser -un tcp 25
25/tcp:              2095(root)

root# ps ax | grep 2095 | grep -v grep
2095 ?        Ss     0:00 sendmail: accepting connections
```

cron

管理程序调度器, 执行一些日常任务, 比如清除和删除系统log文件, 或者更新slocate命令的数据库. 这是at命令的超级用户版本(虽然每个用户都可以有自己的crontab文件, 并且这个文件可以使用crontab命令来修改). 它以幽灵进程T的身份来运行, 并且从/ect/crontab中获得执行的调度入口.

注意: 一些Linux的风格都使用crond, Matthew Dillon的cron.

进程控制和启动类

init

init 命令是所有进程的父进程. 在系统启动的最后一步调用, init 将会依据/etc/inittab来决定系统的运行级别. 只能使用root身份来运行它的别名telinit.

telinit

init命令的符号链接, 这是一种修改系统运行级别的一个手段, 通常在系统维护或者紧急的文件系统修复的时候才用. 只能使用root身份调用. 调用这个命令是非常危险的 - 在你使用之前确定你已经很好地了解它.

runlevel

显示当前和最后的运行级别, 也就是, 确定你的系统是否终止(runlevel 为0), 还是运行在单用户模式(1), 多用户模式(2), 或者是运行在X Windows(5), 还是正在重启(6). 这个命令将会存取/var/run/utmp文件.

halt, shutdown, reboot

设置系统关机的命令, 通常比电源关机的优先级高.

service

开启或停止一个系统服务. 启动脚本在/etc/init.d中, 并且/etc/rc.d在系统启动的时候使用这个命令来启动服务.

```
root# /sbin/service iptables stop
Flushing firewall rules:                   [  OK  ]
Setting chains to policy ACCEPT: filter    [  OK  ]
Unloading iptables modules:                [  OK  ]
```

网络类

ifconfig

网络的接口配置和调试工具.

```
bash$ ifconfig -a
lo        Link encap:Local Loopback
      inet addr:127.0.0.1  Mask:255.0.0.0
      UP LOOPBACK RUNNING  MTU:16436  Metric:1
      RX packets:10 errors:0 dropped:0 overruns:0 frame:0
      TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:0
      RX bytes:700 (700.0 b)  TX bytes:700 (700.0 b)
```

ifconfig 命令绝大多数情况都是在启动时候设置接口, 或者在重启的时候关闭它们.

```
  1 # 来自于 /etc/rc.d/init.d/network 的代码片段
  2
  3 # ...
  4
  5 # 检查网络是否启动.
  6 [ ${NETWORKING} = "no" ] && exit 0
  7
  8 [ -x /sbin/ifconfig ] || exit 0
  9
 10 # ...
 11
 12 for i in $interfaces ; do
 13   if ifconfig $i 2>/dev/null | grep -q "UP" >/dev/null 2>&1 ; then
 14     action "Shutting down interface $i: " ./ifdown $i boot
 15   fi
 16 # grep命令的GNU指定的 "-q" 的意思是"安静", 也就是不产生输出.
 17 # 这样, 后边重定向到/dev/null的操作就有点重复了.
 18
 19 # ...
 20
 21 echo "Currently active devices:"
 22 echo `/sbin/ifconfig | grep ^[a-z] | awk '{print $1}'`
 23 #                           ^^^^^  应该被引用防止globbing.
 24 #  下边这段也能工作.
 25 #    echo $(/sbin/ifconfig | awk '/^[a-z]/ { print $1 })'
 26 #    echo $(/sbin/ifconfig | sed -e 's/ .*//')
 27 #  Thanks, S.C.做了额外的注释.
```

参见 Example 29-6.

iwconfig

这是为了配置无线网络的命令集合. 可以说是上边的ifconfig的无线版本.

route

显示内核路由表信息, 或者查看内核路由表的修改.

```
bash$ route
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
pm3-67.bozosisp *               255.255.255.255 UH      40 0          0 ppp0
127.0.0.0       *               255.0.0.0       U       40 0          0 lo
```

default pm3-67.bozosisp 0.0.0.0 UG 40 0 0 ppp0

chkconfig

检查网络配置. 这个命令负责显示和管理在启动过程中所开启的网络服务(这些服务都是从/etc/rc?.d目录中开启的).

最开始是从IRIX到Red Hat Linux的一个接口, chkconfig在某些Linux发行版中并不是核心安装的一部分.

```
bash$ chkconfig --list
atd             0:off   1:off   2:off   3:on    4:on    5:on    6:off
rwhod           0:off   1:off   2:off   3:off   4:off   5:off   6:off
...
```

tcpdump

网络包的"嗅探器". 这是一个用来分析和调试网络上传输情况的工具, 它所使用的手段是把匹配指定规则的包头都显示出来.

显示主机bozoville和主机caduceus之间所有传输的ip包.

```
bash$ tcpdump ip host bozoville and caduceus
```

当然,tcpdump的输出可以被分析, 可以用我们之前讨论的文本处理工具来分析结果.

文件系统类

mount

加载一个文件系统, 通常都用来安装外部设备, 比如软盘或CDROM. 文件/etc/fstab 将会提供一个方便的列表, 这个列表列出了所有可用的文件系统, 分区和设备, 另外还包括某些选项, 比如是否可以自动或者手动的mount. 文件/etc/mtab 显示了当前已经mount的文件系统和分区(包括虚拟的, 比如/proc).

mount -a 将会mount所有列在/ect/fstab中的文件系统和分区, 除了那些标记有非自动选项的. 在启动的时候, 在/etc/rc.d中的一个启动脚本(rc.sysinit或者一些相似的脚本)将会这么调用, mount所有可用的文件系统和分区.

```
1 mount -t iso9660 /dev/cdrom /mnt/cdrom
2 # 加载 CDROM
3 mount /mnt/cdrom
4 # 方便的方法, 如果 /mnt/cdrom 包含在 /etc/fstab 中
```

这个多功能的命令甚至可以将一个普通文件mount到块设备中, 并且这个文件就好像一个文件系统一样. mount可以将文件与一个loopback设备相关联来达到这个目的.

ccomplishes that by associating the file with a loopback device. 这种应用通常都是用来mount和检查一个ISO9660镜像,在这个镜像被烧录到CDR之前. [3]

Example 13-7 检查一个CD镜像

```
################################Start Script#########################################
1 # 以root身份...
2
3 mkdir /mnt/cdtest  # 如果没有的话,准备一个mount点.
4
5 mount -r -t iso9660 -o loop cd-image.iso /mnt/cdtest   # mount这个镜像.
6 #                 "-o loop" option equivalent to "losetup /dev/loop0"
7 cd /mnt/cdtest     # 现在检查这个镜像.
8 ls -alR           # 列出目录树中的文件.
9                   # 等等.
################################End Script###########################################
```

umount

卸除一个当前已经mount的文件系统. 在正常删除之前已经mount的软盘和CDROM之前, 这个设备必须被unmount, 否则文件系统将会损坏.

```
  1 umount /mnt/cdrom
  2 # 现在你可以按下退出按钮(指的是cdrom或软盘驱动器上的退出钮), 并安全的退出光盘.
```

sync

强制写入所有需要更新的buffer上的数据到硬盘上(同步带有buffer的驱动器). 如果不是严格必要的话,一个sync就可以保证系统管理员或者用户刚刚修改的数据会安全的在突然的断点中幸存下来. 在比较早以前, 在系统重启前都是使用 sync; sync (两次, 这样保证绝对可靠), 这是一种很有用的小心的方法.

有时候, 比如当你想安全删除一个文件的时候(参见 Example 12-55), 或者当磁盘灯开始闪烁的时候, 你可能需要强制马上进行buffer刷新.

losetup

建立和配置loopback设备.

Example 13-8 在一个文件中创建文件系统

```
###################################Start Script#############################################
1 SIZE=1000000  # 1M
2
3 head -c $SIZE < /dev/zero > file  # 建立指定尺寸的文件.
4 losetup /dev/loop0 file          # 作为loopback设备来建立.
5 mke2fs /dev/loop0                # 创建文件系统.
6 mount -o loop /dev/loop0 /mnt     # Mount它.
7
8 # Thanks, S.C.
###################################End Script###############################################
```

mkswap

创建一个交换分区或文件. 交换区域随后必须马上使用swapon来使能.

swapon, swapoff

使能/禁用 交换分区或文件. 这两个命令通常在启动和关机的时候才有效.

mke2fs

创建Linux ext2 文件系统. 这个命令必须以root身份调用.

Example 13-9 添加一个新的硬盘驱动器

```
###################################Start Script#############################################
 1 #!/bin/bash
 2
 3 # 在系统上添加第二块硬盘驱动器.
 4 # 软件配置. 假设硬件已经安装了.
 5 # 来自于本书作者的一篇文章.
 6 # 在"Linux Gazette"的问题#38上, http://www.linuxgazette.com.
 7
 8 ROOT_UID=0     # 这个脚本必须以root身份运行.
 9 E_NOTROOT=67   # 非root用户将会产生这个错误.
10
11 if [ "$UID" -ne "$ROOT_UID" ]
12 then
13   echo "Must be root to run this script."
14   exit $E_NOTROOT
15 fi
16
17 # 要非常谨慎的小心使用!
18 # 如果某步错了, 可能会彻底摧毁你当前的文件系统.
```

```
19
20
21 NEWDISK=/dev/hdb        # 假设/dev/hdb空白. 检查一下!
22 MOUNTPOINT=/mnt/newdisk  # 或者选择另外的mount点.
23
24
25 fdisk $NEWDISK
26 mke2fs -cv $NEWDISK1   # 检查坏块, 详细输出.
27 #  注意:    /dev/hdb1, *不是* /dev/hdb!
28 mkdir $MOUNTPOINT
29 chmod 777 $MOUNTPOINT  # 让所有用户都具有全部权限.
30
31
32 # 现在, 测试一下...
33 # mount -t ext2 /dev/hdb1 /mnt/newdisk
34 # 尝试创建一个目录.
35 # 如果工作起来了, umount它, 然后继续.
36
37 # 最后一步:
38 # 将下边这行添加到/etc/fstab.
39 # /dev/hdb1  /mnt/newdisk  ext2  defaults  1 1
40
41 exit 0
################################End Script#########################################
```

参见 Example 13-8 和 Example 28-3.

tune2fs

调整ext2文件系统. 可以用来修改文件系统参数, 比如mount的最大数量. 必须以root身份调用.

注意: 这是一个非常危险的命令. 如果坏了, 你需要自己负责, 因为它可能会破坏你的文件系统.

dumpe2fs

打印(输出到stdout上)非常详细的文件系统信息. 必须以root身份调用.

```
root# dumpe2fs /dev/hda7 | grep 'ount count'
dumpe2fs 1.19, 13-Jul-2000 for EXT2 FS 0.5b, 95/08/09
Mount count:              6
Maximum mount count:      20
```

hdparm

列出或修改硬盘参数. 这个命令必须以root身份调用, 如果滥用的话会有危险.

fdisk

在存储设备上(通常都是硬盘)创建和修改一个分区表. 必须以root身份使用.

注意: 谨慎使用这个命令. 如果出错, 会破坏你现存的文件系统.

fsck, e2fsck, debugfs

文件系统的检查, 修复, 和除错命令集合.

fsck: 检查UNIX文件系统的前端工具(也可以调用其它的工具). 文件系统的类型一般都是默认的ext2.

e2fsck: ext2文件系统检查器.

debugfs: ext2文件系统除错器. 这个多功能但是危险的工具的用处之一就是(尝试)恢复删除的文件. 只有高级用户才能用.

上边的这几个命令都必须以root身份调用, 这些命令都很危险, 如果滥用的话会破坏文件系统.

badblocks

检查存储设备的坏块(物理损坏). 这个命令在格式化新安装的硬盘时或者测试备份的完整性的时候会被用到. [4] 举个例子, badblocks /dev/fd0 测试一个软盘.

badblocks可能会引起比较糟糕的结果(覆盖所有数据), 在只读模式下就不会发生这种情况.如果root用户拥有需要测试的设备(通常都是这种情况), 那么root用户必须调用这个命令.

lsusb, usbmodules

lsusb 命令会列出所有USB(Universal Serial Bus通用串行总线)总线和使用USB的设备.

usbmodules 命令会输出连接USB设备的驱动模块的信息.

```
root# lsusb
Bus 001 Device 001: ID 0000:0000
Device Descriptor:
  bLength             18
  bDescriptorType      1
  bcdUSB            1.00
  bDeviceClass         9 Hub
  bDeviceSubClass      0
  bDeviceProtocol      0
  bMaxPacketSize0      8
  idVendor          0x0000
  idProduct         0x0000
  . . .
```

mkbootdisk

创建启动软盘, 启动盘可以唤醒系统, 比如当MBR(master boot record主启动记录)坏掉的时候. mkbootdisk 命令其实是一个Bash脚本, 由Erik Troan所编写, 放在/sbin目录中.

chroot

修改ROOT目录. 一般的命令都是从$PATH中获得的, 相对的默认的根目录是 /. 这个命令将会把根目录修改为另一个目录(并且也将把工作目录修改到那). 出于安全目的, 这个命令时非常有用的, 举个例子, 当系统管理员希望限制一些特定的用户, 比如telnet上来的用户, 将他们限定到文件系统上一个安全的地方(这有时候被称为将一个guest用户限制在"chroot 监牢"中). 注意, 在使用chroot之后, 系统的二进制可执行文件的目录将不再可用了.

chroot /opt 将会使得原来的/usr/bin目录变为/opt/usr/bin. 同样,

chroot /aaa/bbb /bin/ls 将会使得ls命令以/aaa/bbb作为根目录, 而不是以前的/.

如果使用alias XX 'chroot /aaa/bbb ls', 并把这句放到用户的~/.bashrc文件中的话, 这将可以有效地限制运行命令"XX"时, 命令"XX"可以使用文件系统的范围.

当从启动盘恢复的时候(chroot 到 /dev/fd0), 或者当系统从死机状态恢复过来并作为进入lilo的选择手段的时候, chroot命令都是非常方便的. 其它的应用还包括从不同的文件系统进行安装(一个rpm选项)或者从CDROM上运行一个只读文件系统. 只能以root身份调用, 小心使用.

注意: 由于正常的$PATH将不再被关联了, 所以可能需要将一些特定的系统文件拷贝到chrooted目录中.

lockfile

这个工具是procmail包的一部分(www.procmail.org). 它可以创建一个锁定文件, 锁定文件是一种用来控制存取文件, 设备或资源的标记文件. 锁定文件就像一个标记一样被使用, 如果特定的文件, 设备, 或资源正在被一个特定的进程所使用("busy"), 那么对于其它进

程来说, 就只能受限进行存取(或者不能存取).

```
1 lockfile /home/bozo/lockfiles/$0.lock
2 # 创建一个以脚本名字为前缀的写保护锁定文件.
```

锁定文件用在一些特定的场合, 比如说保护系统的mail目录以防止多个用户同时修改, 或者提示一个modem端口正在被存取, 或者显示Netscape的一个实例正在使用它的缓存. 脚本可以做一些检查工作, 比如说一个特定的进程可以创建一个锁定文件, 那么只要检查这个特定的进程是否在运行, 就可以判断出锁定文件是否存在了. 注意如果脚本尝试创建一个已经存在的锁定文件的话, 那么脚本很可能被挂起.

一般情况下, 应用创建或检查锁定文件都放在/var/lock目录中. [5] 脚本可以使用下面的方法来检测锁定文件是否存在.

```
1 appname=xyzip
2 # 应用 "xyzip" 创建锁定文件 "/var/lock/xyzip.lock".
3
4 if [ -e "/var/lock/$appname.lock" ]
5 then
6   ...
```

flock<rojy bug>

flock命令比lockfile命令用得少得多.Much less useful than the lockfile command is flock. It sets an "advisory" lock on a file and then executes a command while the lock is on. This is to prevent any other process from setting a lock on that file until completion of the specified command.

```
1 flock $0 cat $0 > lockfile__$0
2 #  Set a lock on the script the above line appears in,
3 #+ while listing the script to stdout.
```

注意: 与lockfile不同, flock不会自动创建一个锁定文件.

mknod

创建块或者字符设备文件(当在系统上安装新硬盘时可能是必要的). MAKEDEV工具事实上具有nknod的全部功能, 而且更容易使用.

MAKEDEV

创建设备文件的工具. 必须在/dev目录下, 并且以root身份使用.

```
root# ./MAKEDEV
```

这是mknod的高级版本.

tmpwatch

自动删除在指定时间内未被存取过的文件. 通常都是被cron调用, 用来删掉老的log文件.

备份类

dump, restore

dump 命令是一个精巧的文件系统备份工具, 通常都用在比较大的安装和网络上. [6] 它读取原始的磁盘分区并且以二进制形式来写备份文件. 需要备份的文件可以保存到各种各样的存储设备上, 包括磁盘和磁带. restore命令用来恢复dump所产生的备份.

fdformat

对软盘进行低级格式化.

系统资源类

ulimit

设置使用系统资源的上限. 通常情况下都是使用-f选项来调用, -f用来设置文件尺寸的限制(ulimit -f 1000就是将文件大小限制为1M). -c(译者注: 这里应该是作者笔误, 作者写的是-t)选项来限制coredump(译者注: 核心转储, 程序崩溃时的内存状态写入文件)尺寸(ulimit -c 0 就是不要coredumps). 一般情况下, ulimit的值应该设置在/etc/profile 和(或)~/.bash_profile中(参见 Appendix G).

注意: Judicious 使用ulimit 可以保护系统免受可怕的fork炸弹的迫害.

```
1 #!/bin/bash
2 # 这个脚本只是为了展示用.
3 # 你要自己为运行这个脚本的后果负责 -- 它*将*凝固你的系统.
4
5 while true  #  死循环.
6 do
7   $0 &     #  这个脚本调用自身 . . .
8           #+ fork无限次 . . .
9           #+ 直道系统完全不动, 因为所有的资源都耗尽了.
10 done       #  这就是臭名卓著的 "sorcerer's appentice" 剧情.<rojy bug>(译者注:巫师的厢房?没看懂)
11
12 exit 0     #  这里不会真正的推出, 因为这个脚本不会终止.
```

当这个脚本超过预先设置的限制时, 在/etc/profile中的 ulimit -Hu XX (XX 就是需要限制的用户进程) 可以终止这个脚本的运行.

quota

显示用户或组的磁盘配额.

setquota

从命令行中设置用户或组的磁盘配额.

umask

设定用户创建文件时权限的缺省mask(掩码). 也可以用来限制特定用户的默认文件属性.

所有用户创建的文件属性都是由umask所指定的. The (octal) 传递给umask的8进制的值定义了文件的权限. 比如, umask 022将会使得新文件的权限最多为755(777 与非 022) [7]

当然, 用户可以随后使用chmod来修改指定文件的属性. 用户一般都是将umask设置值的地方放在/etc/profile 和(或) ~/.bash_profile中 (参见 Appendix G).

Example 13-10 使用umask来将输出文件隐藏起来

```
##################################Start Script#########################################
1 #!/bin/bash
2 # rot13a.sh: 与"rot13.sh"脚本相同, 但是会将输出写道"安全"文件中.
3
4 # 用法: ./rot13a.sh filename
5 # 或     ./rot13a.sh <filename
6 # 或     ./rot13a.sh 同时提供键盘输入(stdin)
7
8 umask 177              #  文件创建掩码.
9                       #  被这个脚本所创建的文件
10                      #+ 将具有600权限.
11
12 OUTFILE=decrypted.txt   #  结果保存在"decrypted.txt"中
13                      #+ 这个文件只能够被
14                      #  这个脚本的调用者(or root)所读写.
15
16 cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M' > $OUTFILE
17 #    ^^ 从stdin 或文件中输入.        ^^^^^^^^^^ 输出重定向到文件中.
18
19 exit 0
##################################End Script#########################################
```

rdev

取得root device, swap space, 或 video mode的相关信息, 或者对它们进行修改. 通常说来rdev都是被lilo所使用, 但是在建立一个ram disk的时候, 这个命令也很有用. 小心使用, 这是一个危险的命令.

模块类

lsmod

列出所有安装的内核模块.

```
bash$ lsmod
Module                  Size  Used by
autofs                  9456   2 (autoclean)
opl3                   11376   0
serial_cs               5456   0 (unused)
sb                     34752   0
uart401                 6384   0 [sb]
sound                  58368   0 [opl3 sb uart401]
soundlow                 464   0 [sound]
soundcore               2800   6 [sb sound]
ds                      6448   2 [serial_cs]
i82365                 22928   2
pcmcia_core            45984   0 [serial_cs ds i82365]
```

注意: 使用cat /proc/modules可以得到同样的结果.

insmod

强制一个内核模块的安装(如果可能的话, 使用modprobe来代替) 必须以root身份调用.

rmmod

强制卸载一个内核模块. 必须以root身份调用.

modprobe

模块装载器, 一般情况下都是在启动脚本中自动调用. 必须以root身份调用.

depmod

创建模块依赖文件, 一般都是在启动脚本中调用.

modinfo

输出一个可装载模块的信息.

```
bash$ modinfo hid
filename:    /lib/modules/2.4.20-6/kernel/drivers/usb/hid.o
description: "USB HID support drivers"
author:      "Andreas Gal, Vojtech Pavlik <vojtech@suse.cz>"
license:     "GPL"
```

杂项类

env

使用设置过的或修改过(并不是修改整个系统环境)的环境变量来运行一个程序或脚本. 使用 [varname=xxx] 形式可以在脚本中修改环境变量. 如果没有指定参数, 那么这个命令将会列出所有设置的环境变量.

注意: 在Bash和其它的Bourne shell 衍生物中, 是可以在单一命令行上设置多个变量的.

```
1 var1=value1 var2=value2 commandXXX
2 # $var1 和 $var2 只设置在'commandXXX'的环境中.
```

注意: 当不知道shell或解释器的路径的时候, 脚本的第一行(#!行)可以使用env.

```
1 #! /usr/bin/env perl
2
3 print "This Perl script will run,\n";
4 print "even when I don't know where to find Perl.\n";
```

```
    5
    6 # 便于跨平台移植,
    7 # Perl程序可能没在期望的地方.
    8 # Thanks, S.C.
```

ldd

显示一个可执行文件的共享库的依赖关系.

```
 bash$ ldd /bin/ls
 libc.so.6 => /lib/libc.so.6 (0x4000c000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

watch

以指定的时间间隔来重复运行一个命令.

默认的时间间隔是2秒, 但时刻以使用-n选项来修改.

```
   1 watch -n 5 tail /var/log/messages
   2 # 每隔5秒钟显示系统log文件的结尾, /var/log/messages.
```

strip

从可执行文件中去掉调试符号引用. 这样做可以减小尺寸, 但是就不能调试了.

这个命令一般都用在Makefile中, 但是很少用在shell脚本中.

nm

列出未strip过的编译后的2进制文件的符号.

rdist

远程文件分布客户机程序: 在远端服务器上同步, 克隆, 或者备份一个文件系统.

13.1 分析一个系统脚本

利用我们所学到的关于管理命令的知识, 让我们一起来练习分析一个系统脚本. 最简单并且最短的系统脚本之一是killall, 这个脚本被用来在系统关机时挂起运行的脚本.

Example 13-11 killall, 来自于 /etc/rc.d/init.d

```
##################################Start Script#########################################
 1 #!/bin/sh
 2
 3 # --> 本书作者所作的注释全部以"# -->"开头.
 4
 5 # --> 这是由Miquel van Smoorenburg所编写的
 6 # --> 'rc'脚本包的一部分, <miquels@drinkel.nl.mugnet.org>.
 7
 8 # --> 这个特殊的脚本看起来是是为Red Hat / FC所特定的,
 9 # --> (在其它的发行版中可能不会出现).
10
11 #  停止所有正在运行的不必要的服务
12 #+ (there shouldn't be any, so this is just a sanity check)
13
14 for i in /var/lock/subsys/*; do
15       # --> 标准的for/in循环, 但是由于"do"在同一行上,
16       # --> 所以必须添加";".
17       # 检查脚本是否在那.
18       [ ! -f $i ] && continue
19       # --> 这是一种使用"与列表"的聪明的方法, 等价于:
20       # --> if [ ! -f "$i" ]; then continue
21
```

```
22      # 取得子系统的名字.
23      subsys=${i#/var/lock/subsys/}
24      # --> 匹配变量名, 在这里就是文件名.
25      # --> 与subsys=`basename $i`完全等价.
26
27      # -->  从锁定文件名中获得
28      # -->+ (如果那里有锁定文件的话,
29      # -->+ 那就证明进程正在运行).
30      # -->  参考一下上边所讲的"锁定文件"的内容.
31
32
33      # 终止子系统.
34      if [ -f /etc/rc.d/init.d/$subsys.init ]; then
35        /etc/rc.d/init.d/$subsys.init stop
36      else
37        /etc/rc.d/init.d/$subsys stop
38      # -->  挂起运行的作业和幽灵进程.
39      # -->  注意"stop"只是一个位置参数,
40      # -->+ 并不是shell内建命令.
41      fi
42 done
```

################################End Script###

这个没有那么糟. 除了在变量匹配的地方玩了一点花样, 其它也没有别的材料了.

练习 1. 在/etc/rc.d/init.d中, 分析halt脚本. 比脚本killall长一些, 但是概念上很相近.
对这个脚本做一个拷贝, 放到你的home目录下并且用它练习一下(不要以root身份运行它). 使用-vn标志来模拟运行一下(sh -vn scriptname). 添加详细的注释. 将"action"命令修改为"echos".

练习 2. 察看/etc/rc.d/init.d下的更多更复杂的脚本. 看看你是不是能够理解其中的一些脚本. 使用上边的过程来分析这些脚本. 为了更详细的理解, 你可能也需要分析在usr/share/doc/initscripts-?.??目录下的文件sysvinitfile, 这些都是"initscript"文件的一部分.

注意事项:

[1] 这是在Linux机器上或者在带有磁盘配额的UNIX系统上的真实情况.

[2] 如果正在被删除的特定的用户已经登录了主机, 那么 userdel 命令将会失败.

[3] 对于烧录CDR的更多的细节, 可以参见Alex Withers的文章, 创建CD, 在Linux Journal 的1999年的10月文章列表中.

[4] mke2fs的-c选项也会进行坏块检查.

[5] 因为只有root用户才具有对/var/lock目录的写权限, 一般的用户脚本是不能在那里设置一个锁定文件的.

[6] 单用户的Linux系统的操作更倾向于使用简单的备份工具, 比如tar.

[7] NAND(与非)是一种逻辑操作. 这种操作的效果和减法很相像.

第14章 命令替换

命令替换将会重新分配一个命令[1]甚至是多个命令的输出; 它会将命令的输出如实地添加到另一个上下文中. [2]

使用命令替换的典型形式是使用后置引用(`...`). 后置引用形式的命令(就是被反引号括起来)将会产生命令行文本.

SHELL十三问

```
1 script_name=`basename $0`
2 echo "The name of this script is $script_name."
```

这样的话, 命令的输出可以被当成传递到另一个命令的参数, 或者保存到变量中, 甚至可以用来产生for循环的参数列表.

```
1 rm `cat filename`  # "filename" 包含了需要被删除的文件列表.
2 #
3 # S. C. 指出使用这种形式, 可能会产生"参数列表太长"的错误.
4 # 更好的方法是            xargs rm -- < filename
5 # ( -- 同时覆盖了那些以"-"开头的文件所产生的特殊情况 )
6
7 textfile_listing=`ls *.txt`
8 # 变量中包含了当前工作目录下所有的*.txt文件.
9 echo $textfile_listing
10
11 textfile_listing2=$(ls *.txt)  # 这是命令替换的另一种形式.
12 echo $textfile_listing2
13 # 同样的结果.
14
15 # 将文件列表放入到一个字符串中的一个可能的问题就是
16 # 可能会混进一个新行.
17 #
18 # 一个安全的将文件列表传递到参数中的方法就是使用数组.
19 #     shopt -s nullglob    # 如果不匹配, 那就不进行文件名扩展.
20 #     textfile_listing=( *.txt )
21 #
22 # Thanks, S.C.
```

注意: 命令替换将会调用一个subshell.

注意: 命令替换可能会引起word splitting.

```
1 COMMAND `echo a b`    # 2个参数: a and b
2
3 COMMAND "`echo a b`"  # 1个参数: "a b"
4
5 COMMAND `echo`        # 无参数
6
7 COMMAND "`echo`"      # 一个空的参数
8
9
10 # Thanks, S.C.
```

即使没有引起word splitting, 命令替换也会去掉多余的新行.

```
1 # cd "`pwd`"  # 这句总会正常的工作.
2 # 然而...
3
4 mkdir 'dir with trailing newline
5 '
6
7 cd 'dir with trailing newline
8 '
9
```

```
cd "`pwd`"  # 错误消息:
# bash: cd: /tmp/file with trailing newline: No such file or directory

cd "$PWD"   # 运行良好.




old_tty_setting=$(stty -g)   # 保存老的终端设置.
echo "Hit a key "
stty -icanon -echo          # 对终端禁用"canonical"模式.
                            # 这样的话, 也会禁用了*本地*的echo.
key=$(dd bs=1 count=1 2> /dev/null)   # 使用'dd'命令来取得一个按键.
stty "$old_tty_setting"     # 保存老的设置.
echo "You hit ${#key} key."  # ${#variable} = number of characters in $variable
#
# 按键任何键除了回车, 那么输出就是"You hit 1 key."
# 按下回车, 那么输出就是"You hit 0 key."
# 新行已经被命令替换吃掉了.

Thanks, S.C.
```

注意: 当一个变量是使用命令替换的结果做为值的时候, 然后使用echo命令来输出这个变量(并且不引用这个变量, 就是不用引号括起来), 那么命令替换将会从最终的输出中删掉换行符. 这可能会引起一些异常情况.

```
dir_listing=`ls -l`
echo $dir_listing     # 未引用, 就是没用引号括起来

# 想打出来一个有序的目录列表.Expecting a nicely ordered directory listing.

# 可惜, 下边将是我们所获得的:
# total 3 -rw-rw-r-- 1 bozo bozo 30 May 13 17:15 1.txt -rw-rw-r-- 1 bozo
# bozo 51 May 15 20:57 t2.sh -rwxr-xr-x 1 bozo bozo 217 Mar 5 21:13 wi.sh

# 新行消失了.



echo "$dir_listing"   # 用引号括起来
# -rw-rw-r--    1 bozo       30 May 13 17:15 1.txt
# -rw-rw-r--    1 bozo       51 May 15 20:57 t2.sh
# -rwxr-xr-x    1 bozo      217 Mar  5 21:13 wi.sh
```

命令替换甚至允许将整个文件的内容放到变量中, 可以使用重定向或者cat命令.

```
variable1=`<file1`     #  将"file1"的内容放到"variable1"中.
variable2=`cat file2`  #  将"file2"的内容放到"variable2"中.
                       #  但是这行将会fork一个新进程, This, however, forks a new process,
                       #+ 所以这行代码将会比第一行代码执行得慢.

#  注意:
```

```
 7 #  变量中是可以包含空白的,
 8 #+ 甚至是 (厌恶至极的), 控制字符.
 1 #  摘录自系统文件, /etc/rc.d/rc.sysinit
 2 #+ (这是红帽安装中使用的)
 3
 4
 5 if [ -f /fsckoptions ]; then
 6         fsckoptions=`cat /fsckoptions`
 7 ...
 8 fi
 9 #
10 #
11 if [ -e "/proc/ide/${disk[$device]}/media" ] ; then
12               hdmedia=`cat /proc/ide/${disk[$device]}/media`
13 ...
14 fi
15 #
16 #
17 if [ ! -n "`uname -r | grep -- "-"`" ]; then
18         ktag="`cat /proc/version`"
19 ...
20 fi
21 #
22 #
23 if [ $usb = "1" ]; then
24     sleep 5
25     mouseoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E "^I.*Cls=03.*Prot=02"`
26     kbdoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E "^I.*Cls=03.*Prot=01"`
27 ...
28 fi
```

注意: 不要将一个非常长的文本文件的内容设置到一个变量中, 除非你有一个非常好的原因非要这么做不可. 不要将2进制文件的内容保存到变量中.

Example 14-1 愚蠢的脚本策略

```
###################################Start Script#########################################
 1 #!/bin/bash
 2 # stupid-script-tricks.sh: 朋友, 别在家这么做.
 3 # 来自于"Stupid Script Tricks," 卷I.
 4
 5
 6 dangerous_variable=`cat /boot/vmlinuz`  # 这是压缩过的Linux内核本身.
 7
 8 echo "string-length of \$dangerous_variable = ${#dangerous_variable}"
 9 # 这个字符串变量的长度是 $dangerous_variable = 794151
10 # (不要使用'wc -c /boot/vmlinuz'来计算长度.)
11
12 # echo "$dangerous_variable"
13 # 千万别尝试这么做! 这样将挂起这个脚本.
14
```

```
15
16 #  文档作者已经意识到将二进制文件设置到
17 #+ 变量中是一个没用的应用.
18
19 exit 0
```

################################End Script###

注意, 在这里是不会发生缓冲区溢出错误. 因为这是一个解释型语言的实例, Bash就是一种解释型语言, 解释型语言会比编译型语言提供更多的对程序错误的保护措施.

变量替换允许将一个循环的输出放入到一个变量中.这么做的关键就是将循环中echo命令的输出全部截取.

Example 14-2 从循环的输出中产生一个变量

################################Start Script#######################################

```
 1 #!/bin/bash
 2 # csubloop.sh: 从循环的输出中产生一个变量.
 3
 4 variable1=`for i in 1 2 3 4 5
 5 do
 6   echo -n "$i"              #  对于这里的命令替换来说
 7 done`                      #+ 这个'echo'命令是非常关键的.
 8
 9 echo "variable1 = $variable1"  # variable1 = 12345
10
11
12 i=0
13 variable2=`while [ "$i" -lt 10 ]
14 do
15   echo -n "$i"              # 再来一个, 'echo'是必须的.
16   let "i += 1"              # 递增.
17 done`
18
19 echo "variable2 = $variable2"  # variable2 = 0123456789
20
21 #  这就证明了在一个变量声明中
22 #+ 嵌入一个循环是可行的.
23
24 exit 0
```

################################End Script###

注意: 命令替换使得扩展有效的Bash工具集变为可能. 这样, 写一段小程序或者一段脚本就可以达到目的, 因为程序或脚本的输出会传到stdout上(就像一个标准的工具所做的那样), 然后重新将这些输出保存到变量中.(译者: 作者的意思就是在这种情况下写脚本和写程序作用是一样的.)

```
 1 #include <stdio.h>
 2
 3 /*  "Hello, world." C program  */
 4
 5 int main()
 6 {
 7   printf( "Hello, world." );
```

```
  8   return (0);
  9 }
bash$ gcc -o hello hello.c
  1 #!/bin/bash
  2 # hello.sh
  3
  4 greeting=`./hello`
  5 echo $greeting
bash$ sh hello.sh
Hello, world.
```

注意: 对于命令替换来说,$(COMMAND) 形式已经取代了反引号"`".

```
  1 output=$(sed -n /"$1"/p $file)   # 来自于 "grp.sh"例子.
  2
  3 # 将一个文本的内容保存到变量中.
  4 File_contents1=$(cat $file1)
  5 File_contents2=$(<$file2)        # Bash 也允许这么做.
```

$(...) 形式的命令替换在处理双反斜线(\\)时与`...`形式不同.

```
bash$ echo `echo \\`

bash$ echo $(echo \\)
```

$(...) 形式的命令替换是允许嵌套的. [3]

```
  1 word_count=$( wc -w $(ls -l | awk '{print $9}') )
```

或者, 可以更加灵活. . .

Example 14-3 找anagram(回文构词法, 可以将一个有意义的单词, 变换为1个或多个有意义的单词, 但是还是原来的子母集合)

##################################Start Script###

```
 1 #!/bin/bash
 2 # agram2.sh
 3 # 关于命令替换嵌套的例子.
 4
 5 #  使用"anagram"工具
 6 #+ 这是作者的"yawl"文字表包中的一部分.
 7 #  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz
 8 #  http://personal.riverusers.com/~thegrendel/yawl-0.3.2.tar.gz
 9
10 E_NOARGS=66
11 E_BADARG=67
12 MINLEN=7
13
14 if [ -z "$1" ]
15 then
16   echo "Usage $0 LETTERSET"
17   exit $E_NOARGS        # 脚本需要一个命令行参数.
18 elif [ ${#1} -lt $MINLEN ]
19 then
20   echo "Argument must have at least $MINLEN letters."
21   exit $E_BADARG
22 fi
```

```
23
24
25
26 FILTER='.......'        # 必须至少有7个字符.
27 #       1234567
28 Anagrams=( $(echo $(anagram $1 | grep $FILTER) ) )
29 #          |    |    嵌套的命令替换        ||
30 #       (             数组分配                )
31
32 echo
33 echo "${#Anagrams[*]}  7+ letter anagrams found"
34 echo
35 echo ${Anagrams[0]}      # 第一个anagram.
36 echo ${Anagrams[1]}      # 第二个anagram.
37                         # 等等.
38
39 # echo "${Anagrams[*]}"  # 在一行上列出所有的anagram . . .
40
41 #  考虑到后边还有"数组"作为单独的一章进行讲解,
42 #+ 这里就不深入了.
43
44 # 可以参阅agram.sh脚本, 这也是一个找出anagram的例子.
45
46 exit $?
```

##################################End Script##

命令替换在脚本中使用的例子:

1. Example 10-7
2. Example 10-26
3. Example 9-28
4. Example 12-3
5. Example 12-19
6. Example 12-15
7. Example 12-49
8. Example 10-13
9. Example 10-10
10. Example 12-29
11. Example 16-8
12. Example A-17
13. Example 27-2
14. Example 12-42
15. Example 12-43
16. Example 12-44

注意事项:

[1] 对于命令替换来说, 这个命令可以是外部的系统命令, 也可以是内部脚本的内建命令, 甚至是一个脚本函数.

[2] 从技术的角度来讲, 命令替换将会抽取出一个命令的输出, 然后使用=操作赋值到一个变量中.

[3] 事实上, 对于反引号的嵌套是可行的, 但是只能将内部的反引号转义才行, 就像

John默认指出的那样.

```
1 word_count=` wc -w \`ls -l | awk '{print $9}'\` `
```

第15章 算术扩展

==========

算术扩展提供了一种强力的工具, 可以在脚本中执行(整型)算法操作. 可以使用backticks, double parentheses, 或 let来将字符串转换为数字表达式.

一些变化

使用反引号的算术扩展(通常都是和expr一起使用)

```
1 z=`expr $z + 3`         # 'expr'命令将会执行这个扩展.
```

使用双括号, 和let形式的算术扩展

反引号形式的算术扩展已经被双括号形式所替代了 -- ((...)) 和 $((...)) -- 当然也可以使用非常方便的let形式.

```
1 z=$(($z+3))
2 z=$((z+3))                    #  也正确.
3                               #  使用双括号的形式,
4                               #+ 参数解引用
5                               #+ 是可选的.
6
7 # $((EXPRESSION)) is arithmetic expansion.  #  不要与命令
8                               #+ 替换相混淆.
9
10
11
12 # 使用双括号的形式也可以不用给变量赋值.
13
14  n=0
15  echo "n = $n"                  # n = 0
16
17  (( n += 1 ))                  # 递增.
18 # (( $n += 1 )) is incorrect!
19  echo "n = $n"                  # n = 1
20
21
22 let z=z+3
23 let "z += 3"  #  使用引用的形式, 允许在变量赋值的时候存在空格.
24            #  'let'操作事实上执行得的是算术赋值,
25            #+ 而不是算术扩展.
```

下边是一些在脚本中使用算术扩展的例子:

1. Example 12-9
2. Example 10-14
3. Example 26-1
4. Example 26-11
5. Example A-17

第16章 I/O 重定向

==========

默认情况下始终有3个"文件"处于打开状态, stdin (键盘), stdout (屏幕), and stderr (错误消息输出到屏幕上). 这3个文件和其他打开的文件都可以被重定向. 对于重定向简单的

解释就是捕捉一个文件, 命令, 程序, 脚本, 或者甚至是脚本中的代码块(参见 Example 3-1 和 Example 3-2)的输出, 然后将这些输出作为输入发送到另一个文件, 命令, 程序, 或脚本中.

每个打开的文件都会被分配一个文件描述符.[1]stdin, stdout, 和stderr的文件描述符分别是0, 1, 和 2. 对于正在打开的额外文件, 保留了描述符3到9. 在某些时候将这些格外的文件描述符分配给stdin, stdout, 或者是stderr作为临时的副本链接是非常有用的. [2] 在经过复杂的重定向和刷新之后需要把它们恢复成正常的样子 (参见 Example 16-1).

```
COMMAND_OUTPUT >
  # 重定向stdout到一个文件.
  # 如果没有这个文件就创建, 否则就覆盖.

  ls -lR > dir-tree.list
  # 创建一个包含目录树列表的文件.

: > filename
  # > 会把文件"filename"截断为0长度.
  # 如果文件不存在, 那么就创建一个0长度的文件(与'touch'的效果相同).
  # : 是一个占位符, 不产生任何输出.

> filename
  # > 会把文件"filename"截断为0长度.
  # 如果文件不存在, 那么就创建一个0长度的文件(与'touch'的效果相同).
  # (与上边的": >"效果相同, 但是在某些shell下可能不能工作.)

COMMAND_OUTPUT >>
  # 重定向stdout到一个文件.
  # 如果文件不存在, 那么就创建它, 如果存在, 那么就追加到文件后边.


  # 单行重定向命令(只会影响它们所在的行):
  # -------------------------------------------------------------------

1>filename
  # 重定向stdout到文件"filename".
1>>filename
  # 重定向并追加stdout到文件"filename".
2>filename
  # 重定向stderr到文件"filename".
2>>filename
  # 重定向并追加stderr到文件"filename".
&>filename
  # 将stdout和stderr都重定向到文件"filename".

  #==============================================================================
  # 重定向stdout, 一次一行.
  LOGFILE=script.log

  echo "This statement is sent to the log file, \"$LOGFILE\"." 1>$LOGFILE
```

```
42    echo "This statement is appended to \"$LOGFILE\"." 1>>$LOGFILE
43    echo "This statement is also appended to \"$LOGFILE\"." 1>>$LOGFILE
44    echo "This statement is echoed to stdout, and will not appear in \"$LOGFILE\"."
45    # 每行过后, 这些重定向命令会自动"reset".
46
47
48
49    # 重定向stderr, 一次一行.
50    ERRORFILE=script.errors
51
52    bad_command1 2>$ERRORFILE       #  错误消息发到$ERRORFILE中.
53    bad_command2 2>>$ERRORFILE      #  错误消息添加到$ERRORFILE中.
54    bad_command3                    #  错误消息echo到stderr,
55                                    #+ 并且不出现在$ERRORFILE中.
56    # 每行过后, 这些重定向命令也会自动"reset".
57    #=============================================================================
58
59
60
61  2>&1
62    # 重定向stderr到stdout.
63    # 得到的错误消息与stdout一样, 发送到一个地方.
64
65  i>&j
66    # 重定向文件描述符i 到 j.
67    # 指向i文件的所有输出都发送到j中去.
68
69  >&j
70    # 默认的, 重定向文件描述符1(stdout)到 j.
71    # 所有传递到stdout的输出都送到j中去.
72
73  0< FILENAME
74   < FILENAME
75    # 从文件中接受输入.
76    # 与">"是成对命令, 并且通常都是结合使用.
77    #
78    # grep search-word <filename
79
80
81  [j]<>filename
82    # 为了读写"filename", 把文件"filename"打开, 并且分配文件描述符"j"给它.
83    # 如果文件"filename"不存在, 那么就创建它.
84    # 如果文件描述符"j"没指定, 那默认是fd 0, stdin.
85    #
86    # 这种应用通常是为了写到一个文件中指定的地方.
87    echo 1234567890 > File    # 写字符串到"File".
88    exec 3<> File             # 打开"File"并且给它分配fd 3.
89    read -n 4 <&3             # 只读4个字符.
```

```
 90     echo -n . >&3          # 写一个小数点.
 91     exec 3>&-              # 关闭fd 3.
 92     cat File               # ==> 1234.67890
 93     # 随机存储.
 94
 95
 96
 97   |
 98     # 管道.
 99     # 通用目的的处理和命令链工具.
100     # 与">"很相似, 但是实际上更通用.
101     # 对于想将命令, 脚本, 文件和程序串连起来的时候很有用.
102     cat *.txt | sort | uniq > result-file
103     # 对所有的.txt文件的输出进行排序, 并且删除重复行,
104     # 最后将结果保存到"result-file"中.
```

可以将输入输出重定向和(或)管道的多个实例结合到一起写在一行上.

```
 1 command < input-file > output-file
 2
 3 command1 | command2 | command3 > output-file
```

参见 Example 12-28 和 Example A-15.

可以将多个输出流重定向到一个文件上.

```
 1 ls -yz >> command.log 2>&1
 2 #  将错误选项"yz"的结果放到文件"command.log"中.
 3 #  因为stderr被重定向到这个文件中,
 4 #+ 所有的错误消息也就都指向那里了.
 5
 6 #  注意, 下边这个例子就不会给出相同的结果.
 7 ls -yz 2>&1 >> command.log
 8 #  输出一个错误消息, 但是并不写到文件中.
 9
10 #  如果将stdout和stderr都重定向,
11 #+ 命令的顺序会有些不同.
```

关闭文件描述符

n<&- 关闭输入文件描述符n.

0<&-, <&- 关闭stdin.

n>&- 关闭输出文件描述符n.

1>&-, >&- 关闭stdout.

子进程继承了打开的文件描述符. 这就是为什么管道可以工作. 如果想阻止fd被继承, 那么可以关掉它.

```
 1 # 只重定向stderr到一个管道.
 2
 3 exec 3>&1                     # 保存当前stdout的"值".
 4 ls -l 2>&1 >&3 3>&- | grep bad 3>&-    # 对'grep'关闭fd 3(但不关闭'ls').
 5 #            ^^^^   ^^^^
 6 exec 3>&-                    # 现在对于剩余的脚本关闭它.
 7
 8 # Thanks, S.C.
```

如果想了解关于I/O重定向更多的细节参见 附录 E.

16.1. 使用exec

---

exec <filename 命令会将stdin重定向到文件中. 从这句开始, 后边的输入就都来自于这个文件了, 而不是标准输入了(通常都是键盘输入). 这样就提供了一种按行读取文件的方法, 并且可以使用sed 和/或 awk来对每一行进行分析.

Example 16-1 使用exec重定向标准输入

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # 使用'exec'重定向标准输入.
 3
 4
 5 exec 6<&0          # 将文件描述符#6与stdin链接起来.
 6                   # 保存了stdin.
 7
 8 exec < data-file   # stdin被文件"data-file"所代替.
 9
10 read a1           # 读取文件"data-file"的第一行.
11 read a2           # 读取文件"data-file"的第二行.
12
13 echo
14 echo "Following lines read from file."
15 echo "-------------------------------"
16 echo $a1
17 echo $a2
18
19 echo; echo; echo
20
21 exec 0<&6 6<&-
22 #  现在将stdin从fd #6中恢复, 因为刚才我们把stdin重定向到#6了,
23 #+ 然后关闭fd #6 ( 6<&- ), 好让这个描述符继续被其他进程所使用.
24 #
25 # <&6 6<&-    这么做也可以.
26
27 echo -n "Enter data  "
28 read b1  # 现在"read"已经恢复正常了, 就是从stdin中读取.
29 echo "Input read from stdin."
30 echo "----------------------"
31 echo "b1 = $b1"
32
33 echo
34
35 exit 0
################################End Script###########################################
```

同样的, exec >filename 命令将会把stdout重定向到一个指定的文件中. 这样所有的命令输出就都会发向那个指定的文件, 而不是stdout.

Example 16-2 使用exec来重定向stdout

```
################################Start Script#########################################
 1 #!/bin/bash
```

```
 2 # reassign-stdout.sh
 3
 4 LOGFILE=logfile.txt
 5
 6 exec 6>&1          # 将fd #6与stdout相连接.
 7                    # 保存stdout.
 8
 9 exec > $LOGFILE     # stdout就被文件"logfile.txt"所代替了.
10
11 # ------------------------------------------------------------ #
12 # 在这块中所有命令的输出就都发向文件 $LOGFILE.
13
14 echo -n "Logfile: "
15 date
16 echo "-------------------------------------"
17 echo
18
19 echo "Output of \"ls -al\" command"
20 echo
21 ls -al
22 echo; echo
23 echo "Output of \"df\" command"
24 echo
25 df
26
27 # ------------------------------------------------------------ #
28
29 exec 1>&6 6>&-      # 恢复stdout, 然后关闭文件描述符#6.
30
31 echo
32 echo "== stdout now restored to default == "
33 echo
34 ls -al
35 echo
36
37 exit 0
```

################################End Script###

Example 16-3 使用exec在同一脚本中重定向stdin和stdout

################################Start Script###

```
 1 #!/bin/bash
 2 # upperconv.sh
 3 # 将一个指定的输入文件转换为大写.
 4
 5 E_FILE_ACCESS=70
 6 E_WRONG_ARGS=71
 7
 8 if [ ! -r "$1" ]    # 判断指定的输入文件是否可读?
 9 then
```

```
10   echo "Can't read from input file!"
11   echo "Usage: $0 input-file output-file"
12   exit $E_FILE_ACCESS
13 fi                 #  即使输入文件($1)没被指定
14                    #+ 也还是会以相同的错误退出(为什么?).
15
16 if [ -z "$2" ]
17 then
18   echo "Need to specify output file."
19   echo "Usage: $0 input-file output-file"
20   exit $E_WRONG_ARGS
21 fi
22
23
24 exec 4<&0
25 exec < $1           # 将会从输入文件中读取.
26
27 exec 7>&1
28 exec > $2           # 将写到输出文件中.
29                    # 假设输出文件是可写的(添加检查?).
30
31 # -----------------------------------------------
32    cat - | tr a-z A-Z   # 转换为大写.
33 #   ^^^^^              # 从stdin中读取.Reads from stdin.
34 #         ^^^^^^^^^^   # 写到stdout上.
35 # 然而, stdin和stdout都被重定向了.
36 # -----------------------------------------------
37
38 exec 1>&7 7>&-      # 恢复 stout.
39 exec 0<&4 4<&-      # 恢复 stdin.
40
41 # 恢复之后, 下边这行代码将会如期望的一样打印到stdout上.
42 echo "File \"$1\" written to \"$2\" as uppercase conversion."
43
44 exit 0
```

################################End Script###

I/O重定向是一种避免可怕的子shell中不可存取变量问题的方法.

Example 16-4 避免子shell

################################Start Script###

```
 1 #!/bin/bash
 2 # avoid-subshell.sh
 3 # Matthew Walker提出的建议.
 4
 5 Lines=0
 6
 7 echo
 8
 9 cat myfile.txt | while read line;  #  (译者注: 管道会产生子shell)
```

```
10              do {
11                echo $line
12                (( Lines++ ));  #  增加这个变量的值
13                              #+ 但是外部循环却不能存取.
14                              #  子shell问题.
15              }
16              done
17
18 echo "Number of lines read = $Lines"     # 0
19                                  # 错误!
20
21 echo "------------------------"
22
23
24 exec 3<> myfile.txt
25 while read line <&3
26 do {
27   echo "$line"
28   (( Lines++ ));                 #  增加这个变量的值
29                              #+ 现在外部循环就可以存取了.
30                              #  没有子shell, 现在就没问题了.
31 }
32 done
33 exec 3>&-
34
35 echo "Number of lines read = $Lines"     # 8
36
37 echo
38
39 exit 0
40
41 # 下边这些行是脚本的结果, 脚本是不会走到这里的.
42
43 $ cat myfile.txt
44
45 Line 1.
46 Line 2.
47 Line 3.
48 Line 4.
49 Line 5.
50 Line 6.
51 Line 7.
52 Line 8.
```

################################End Script##

注意事项:

[1] 一个文件描述符说白了就是文件系统为了跟踪这个打开的文件而分配给它的一个数字.
也可以的将其理解为文件指针的一个简单版本. 与C中的文件句柄的概念相似.

[2] 使用文件描述符5可能会引起问题. 当Bash使用exec创建一个子进程的时候, 子进程

会继承fd5(参见Chet Ramey的归档e-mail, SUBJECT: RE: File descriptor 5 is held open). 最好还是不要去招惹这个特定的fd.

第17章 Here Documents

==========

here document 就是一段特殊目的的代码块. 他使用I/O 重定向的形式来将一个命令序列传递到一个交互程序或者命令中, 比如ftp, cat, 或者ex文本编辑器.

```
1 COMMAND <<InputComesFromHERE
2 ...
3 InputComesFromHERE
```

limit string 用来划定命令序列的范围(译者注: 两个相同的limit string之间就是命令序列). 特殊符号 << 用来表识limit string. 这个符号具有重定向文件的输出到程序或命令的输入的作用. 与 interactive-program < command-file 很相象, command-file包含:

```
1 command #1
2 command #2
3 ...
```

而here document 的形式看上去是如下的样子:

```
1 #!/bin/bash
2 interactive-program <<LimitString
3 command #1
4 command #2
5 ...
6 LimitString
```

选择一个名字非常诡异的limit string将会避免命令列表和limit string重名的问题.

注意,某些时候here document 用在非交互工具和命令上的时候也会有好的效果, 比如, wall.

Example 17-1 广播: 发送消息给每个登录上的用户

```
################################Start Script#########################################
1 #!/bin/bash
2
3 wall <<zzz23EndOfMessagezzz23
4 E-mail your noontime orders for pizza to the system administrator.
5     (Add an extra dollar for anchovy or mushroom topping.)
6 # 额外的消息文本写在这里.
7 # 注意: 'wall' 会打印注释行.
8 zzz23EndOfMessagezzz23
9
10 # 可以使用更有效率的做法
11 #         wall <message-file
12 #  然而将消息模版嵌入到脚本中
13 #+ 是一种"小吃店"(快速但是比较脏)的只能使用一次的解决办法.
14
15 exit 0
################################End Script#########################################
```

即使是某些不大可能的工具, 如vi也可以使用here document.

Example 17-2 仿造文件: 创建一个两行的仿造文件

```
################################Start Script#########################################
1 #!/bin/bash
2
3 # 用非交互的方式来使用'vi'编辑一个文件.
```

```
4 # 模仿'sed'.
5
6 E_BADARGS=65
7
8 if [ -z "$1" ]
9 then
10   echo "Usage: `basename $0` filename"
11   exit $E_BADARGS
12 fi
13
14 TARGETFILE=$1
15
16 # 在文件中插入两行, 然后保存.
17 #--------Begin here document-----------#
18 vi $TARGETFILE <<x23LimitStringx23
19 i
20 This is line 1 of the example file.
21 This is line 2 of the example file.
22 ^[
23 ZZ
24 x23LimitStringx23
25 #----------End here document-----------#
26
27 #  注意上边^[是一个转义符,键入Ctrl+v <Esc>就行,
28 #+ 事实上它是<Esc>键.
29
30 #  Bram Moolenaar指出这种方法不能正常地用在'vim'上, (译者注: Bram Moolenaar是vim作者)
31 #+ 因为可能会有终端的相互影响问题.
32
33 exit 0
```

################################End Script###

上边的脚本也可以不用vi而用ex来实现. Here document 包含ex命令列表的做法足够形成自己的类别了, 叫ex scripts.

```
1 #!/bin/bash
2 #  把所有后缀为".txt"文件
3 #+ 中的"Smith"都替换成"Jones".
4
5 ORIGINAL=Smith
6 REPLACEMENT=Jones
7
8 for word in $(fgrep -l $ORIGINAL *.txt)
9 do
10   # --------------------------------------
11   ex $word <<EOF
12   :%s/$ORIGINAL/$REPLACEMENT/g
13   :wq
14 EOF
15   # :%s 是"ex"的替换命令.
```

```
 16   # :wq 是保存并退出的意思.
 17   # ---------------------------------
 18 done
```

与"ex scripts"相似的是cat scripts.

Example 17-3 使用cat的多行消息

```
###################################Start Script#########################################
 1 #!/bin/bash
 2
 3 #  'echo' 对于打印单行消息是非常好的,
 4 #+  但是在打印消息块时可能就有点问题了.
 5 #   'cat' here document可以解决这个限制.
 6
 7 cat <<End-of-message
 8 -----------------------------------
 9 This is line 1 of the message.
10 This is line 2 of the message.
11 This is line 3 of the message.
12 This is line 4 of the message.
13 This is the last line of the message.
14 -----------------------------------
15 End-of-message
16
17 #  用下边这行代替上边的第7行
18 #+   cat > $Newfile <<End-of-message
19 #+       ^^^^^^^^^^
20 #+ 那么就会把输出写到文件$Newfile中, 而不是stdout.
21
22 exit 0
23
24
25 #-------------------------------------------
26 # 下边的代码不会运行, 因为上边的"exit 0".
27
28 # S.C. 指出下边代码也可以运行.
29 echo "-----------------------------------
30 This is line 1 of the message.
31 This is line 2 of the message.
32 This is line 3 of the message.
33 This is line 4 of the message.
34 This is the last line of the message.
35 -----------------------------------"
36 # 然而, 文本可能不包含双引号, 除非它们被转义.
###################################End Script###########################################
```

- 选项用来标记here document的limit string (<<-LimitString), 可以抑制输出时前边的tab (不是空格). 这可以增加一个脚本的可读性.

Example 17-4 带有抑制tab功能的多行消息

```
###################################Start Script#########################################
 1 #!/bin/bash
```

```
 2 # 与之前的例子相同, 但是...
 3
 4 #  - 选项对于here docutment来说,<<-
 5 #+ 可以抑制文档体前边的tab,
 6 #+ 而*不*是空格 *not* spaces.
 7
 8 cat <<-ENDOFMESSAGE
 9  This is line 1 of the message.
10  This is line 2 of the message.
11  This is line 3 of the message.
12  This is line 4 of the message.
13  This is the last line of the message.
14 ENDOFMESSAGE
15 # 脚本在输出的时候左边将被刷掉.
16 # 就是说每行前边的tab将不会显示.
17
18 # 上边5行"消息"的前边都是tab, 不是空格.
19 # 空格是不受<<-影响的.
20
21 # 注意, 这个选项对于*嵌在*中间的tab没作用.
22
23 exit 0
```

##################################End Script##

here document 支持参数和命令替换. 所以也可以给here document的消息体传递不同的参数,
这样相应的也会修改输出.

Example 17-5 使用参数替换的here document

##################################Start Script##

```
 1 #!/bin/bash
 2 # 一个使用'cat'命令的here document, 使用了参数替换
 3
 4 # 不传命令行参数给它,  ./scriptname
 5 # 传一个命令行参数给它,  ./scriptname Mortimer
 6 # 传一个2个单词(用引号括起来)的命令行参数给它,
 7 #                       ./scriptname "Mortimer Jones"
 8
 9 CMDLINEPARAM=1     # 所期望的最少的命令行参数的个数.
10
11 if [ $# -ge $CMDLINEPARAM ]
12 then
13   NAME=$1         # 如果命令行参数超过1个,
14                   #+ 那么就只取第一个参数.
15 else
16   NAME="John Doe"  # 默认情况下, 如果没有命令行参数的话.
17 fi
18
19 RESPONDENT="the author of this fine script"
20
21
```

```
22 cat <<Endofmessage
23
24 Hello, there, $NAME.
25 Greetings to you, $NAME, from $RESPONDENT.
26
27 # This comment shows up in the output (why?).
28
29 Endofmessage
30
31 # 注意上边的空行也打印到输出,
32 # 而上边那行"注释"当然也会打印到输出.
33 # (译者注: 这就是为什么不翻译那行注释的原因, 尽量保持原代码的原样)
34 exit 0
```

################################End Script###

这是一个包含参数替换的here document的有用的脚本.

Example 17-6 上传一个文件对到"Sunsite"的incoming目录

################################Start Script###

```
 1 #!/bin/bash
 2 # upload.sh
 3
 4 #  上传文件对(Filename.lsm, Filename.tar.gz)
 5 #+ 到Sunsite/UNC (ibiblio.org)的incoming目录.
 6 #  Filename.tar.gz是自身的tar包.
 7 #  Filename.lsm是描述文件.
 8 #  Sunsite需要"lsm"文件, 否则就拒绝贡献.
 9
10
11 E_ARGERROR=65
12
13 if [ -z "$1" ]
14 then
15   echo "Usage: `basename $0` Filename-to-upload"
16   exit $E_ARGERROR
17 fi
18
19
20 Filename=`basename $1`          # 从文件名中去掉目录字符串.
21
22 Server="ibiblio.org"
23 Directory="/incoming/Linux"
24 #  在这里也不一定非得将上边的参数写死在这个脚本中,
25 #+ 可以使用命令行参数的方法来替换.
26
27 Password="your.e-mail.address"   # 可以修改成相匹配的密码.
28
29 ftp -n $Server <<End-Of-Session
30 # -n 选项禁用自动登录.
31
```

```
32 user anonymous "$Password"
33 binary
34 bell                       # 在每个文件传输后, 响铃.
35 cd $Directory
36 put "$Filename.lsm"
37 put "$Filename.tar.gz"
38 bye
39 End-Of-Session
40
41 exit 0
```

################################End Script###

在here document的开头引用或转义"limit string"会使得here document的消息体中的参数替换被禁用.

Example 17-7 关闭参数替换

################################Start Script###

```
 1 #!/bin/bash
 2 #  一个使用'cat'的here document, 但是禁用了参数替换.
 3
 4 NAME="John Doe"
 5 RESPONDENT="the author of this fine script"
 6
 7 cat <<'Endofmessage'
 8
 9 Hello, there, $NAME.
10 Greetings to you, $NAME, from $RESPONDENT.
11
12 Endofmessage
13
14 #  当"limit string"被引用或转义那么就禁用了参数替换.
15 #  下边的两种方式具有相同的效果.
16 #  cat <<"Endofmessage"
17 #  cat <<\Endofmessage
18
19 exit 0
```

################################End Script###

禁用了参数替换后, 将允许输出文本本身(译者注: 就是未转义的原文). 产生脚本甚至是程序代码就是这种用法的用途之一.

Example 17-8 一个产生另外一个脚本的脚本

################################Start Script###

```
 1 #!/bin/bash
 2 # generate-script.sh
 3 # 基于Albert Reiner的一个主意.
 4
 5 OUTFILE=generated.sh        # 所产生文件的名字.
 6
 7
 8 # ------------------------------------------------------------
 9 # 'Here document包含了需要产生的脚本的代码.
```

```
10 (
11 cat <<'EOF'
12 #!/bin/bash
13
14 echo "This is a generated shell script."
15 #  Note that since we are inside a subshell,
16 #+ we can't access variables in the "outside" script.
17
18 echo "Generated file will be named: $OUTFILE"
19 #  Above line will not work as normally expected
20 #+ because parameter expansion has been disabled.
21 #  Instead, the result is literal output.
22
23 a=7
24 b=3
25
26 let "c = $a * $b"
27 echo "c = $c"
28
29 exit 0
30 EOF
31 ) > $OUTFILE
32 # ------------------------------------------------------------
33
34 #  将'limit string'引用起来将会阻止上边
35 #+ here document的消息体中的变量扩展.
36 #  这会使得输出文件中的内容保持here document消息体中的原文.
37
38 if [ -f "$OUTFILE" ]
39 then
40   chmod 755 $OUTFILE
41   # 让所产生的文件具有可执行权限.
42 else
43   echo "Problem in creating file: \"$OUTFILE\""
44 fi
45
46 #  这个方法也用来产生
47 #+ C程序代码, Perl程序代码, Python程序代码, makefile,
48 #+ 和其他的一些类似的代码.
49 #  (译者注: 中间一段没译的注释将会被here document打印出来)
50 exit 0
```

################################End Script###

也可以将here document的输出保存到变量中.

```
 1 variable=$(cat <<SETVAR
 2 This variable
 3 runs over multiple lines.
 4 SETVAR)
 5
```

```
  6 echo "$variable"
```

同一脚本中的函数也可以接受here document的输出作为自身的参数.

Example 17-9 Here documents与函数

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # here-function.sh
 3
 4 GetPersonalData ()
 5 {
 6   read firstname
 7   read lastname
 8   read address
 9   read city
10   read state
11   read zipcode
12 } # 这个函数无疑的看起来就一个交互函数, 但是...
13
14
15 # 给上边的函数提供输入.
16 GetPersonalData <<RECORD001
17 Bozo
18 Bozeman
19 2726 Nondescript Dr.
20 Baltimore
21 MD
22 21226
23 RECORD001
24
25
26 echo
27 echo "$firstname $lastname"
28 echo "$address"
29 echo "$city, $state $zipcode"
30 echo
31
32 exit 0
################################End Script###########################################
```

也可以这么使用: 做一个假命令来从一个here document中接收输出. 这么做事实上就是创建了一个"匿名"的here document.

Example 17-10 "匿名" here Document

```
################################Start Script#########################################
1 #!/bin/bash
2
3 : <<TESTVARIABLES
4 ${HOSTNAME?}${USER?}${MAIL?}  # 如果其中一个变量没被设置, 那么就打印错误信息.
5 TESTVARIABLES
6
7 exit 0
```

```
################################End Script#########################################
```

注意: 上边所示技术的一种变化可以用来"注释"掉代码块.

Example 17-11 注释掉一段代码块

```
################################Start Script#######################################
 1 #!/bin/bash
 2 # commentblock.sh
 3
 4 : <<COMMENTBLOCK
 5 echo "This line will not echo."
 6 This is a comment line missing the "#" prefix.
 7 This is another comment line missing the "#" prefix.
 8
 9 &*@!!++=
10 The above line will cause no error message,
11 because the Bash interpreter will ignore it.
12 COMMENTBLOCK
13
14 echo "Exit value of above \"COMMENTBLOCK\" is $?."   # 0
15 # 这里将不会显示任何错误.
16
17
18 #  上边的这种技术当然也可以用来注释掉
19 #+ 一段正在使用的代码, 如果你有某些特定调试要求的话.
20 #  这将比对每行都敲入"#"来得方便的多,
21 #+ 而且如果你想恢复的话, 还得将添加上的"#"删除掉.
22
23 : <<DEBUGXXX
24 for file in *
25 do
26  cat "$file"
27 done
28 DEBUGXXX
29
30 exit 0
################################End Script#########################################
```

注意: 关于这种小技巧的另一个应用就是能够产生自文档化(self-documenting)的脚本.

Example 17-12 一个自文档化(self-documenting)的脚本

```
################################Start Script#######################################
 1 #!/bin/bash
 2 # self-document.sh: 自文档化(self-documenting)的脚本
 3 # Modification of "colm.sh".
 4
 5 DOC_REQUEST=70
 6
 7 if [ "$1" = "-h"  -o "$1" = "--help" ]     # 请求帮助.
 8 then
 9   echo; echo "Usage: $0 [directory-name]"; echo
10   sed --silent -e '/DOCUMENTATIONXX$/,/^DOCUMENTATIONXX$/p' "$0" |
```

```
11   sed -e '/DOCUMENTATIONXX$/d'; exit $DOC_REQUEST; fi
12
13
14 : <<DOCUMENTATIONXX
15 List the statistics of a specified directory in tabular format.
16 ---------------------------------------------------------------
17 The command line parameter gives the directory to be listed.
18 If no directory specified or directory specified cannot be read,
19 then list the current working directory.
20
21 DOCUMENTATIONXX
22
23 if [ -z "$1" -o ! -r "$1" ]
24 then
25   directory=.
26 else
27   directory="$1"
28 fi
29
30 echo "Listing of "$directory":"; echo
31 (printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \
32 ; ls -l "$directory" | sed 1d) | column -t
33
34 exit 0
```

##################################End Script###

使用cat 脚本 也能够完成相同的目的.

```
  1 DOC_REQUEST=70
  2
  3 if [ "$1" = "-h"  -o "$1" = "--help" ]     # 请求帮助.
  4 then                                 # 使用"cat 脚本" . . .
  5   cat <<DOCUMENTATIONXX
  6 List the statistics of a specified directory in tabular format.
  7 ---------------------------------------------------------------
  8 The command line parameter gives the directory to be listed.
  9 If no directory specified or directory specified cannot be read,
 10 then list the current working directory.
 11
 12 DOCUMENTATIONXX
 13 exit $DOC_REQUEST
 14 fi
```

参见 Example A-27 可以了解更多关于自文档化脚本的好例子.

注意: Here document创建临时文件, 但是这些文件将在打开后被删除, 并且不能够被任何其他进程所存取.

```
bash$ bash -c 'lsof -a -p $$ -d0' << EOF
> EOF
lsof    1213 bozo    0r   REG    3,5    0 30386 /tmp/t1213-0-sh (deleted)
```

注意: 某些工具是不能工作在here document中的.

警告: 结束的limit string, 就是here document最后一行的limit string, 必须开始于第一

个字符位置. 它的前面不能够有任何前置的空白. 而在这个limit string后边的空白也会
引起异常问题. 空白将会阻止limit string的识别.(译者注: 下边这个脚本由于结束
limit string的问题, 造成脚本无法结束, 所有内容全部被打印出来, 所以注释就不译了,
保持例子脚本的原样.)

```
 1 #!/bin/bash
 2
 3 echo "---------------------------------------------------------------------"
 4
 5 cat <<LimitString
 6 echo "This is line 1 of the message inside the here document."
 7 echo "This is line 2 of the message inside the here document."
 8 echo "This is the final line of the message inside the here document."
 9     LimitString
10 #^^^^Indented limit string. Error! This script will not behave as expected.
11
12 echo "---------------------------------------------------------------------"
13
14 #  These comments are outside the 'here document',
15 #+ and should not echo.
16
17 echo "Outside the here document."
18
19 exit 0
20
21 echo "This line had better not echo."  # Follows an 'exit' command.
```

对于那些使用"here document"得非常复杂的任务, 最好考虑使用expect脚本语言, 这种语言
就是为了达到向交互程序添加输入的目的而量身定做的.

17.1. Here Strings

---

here string 可以被认为是here document的一种定制形式. 除了COMMAND <<<$WORD 就什么都
没有了, $WORD将被扩展并且被送入COMMAND的stdin中.

Example 17-13 在一个文件的开头添加文本

##################################Start Script###

```
 1 #!/bin/bash
 2 # prepend.sh: 在文件的开头添加文本.
 3 #
 4 #  Kenny Stauffer所捐助的脚本例子,
 5 #+ 被本文作者作了少量的修改.
 6
 7
 8 E_NOSUCHFILE=65
 9
10 read -p "File: " file   # 'read'命令的 -p 参数显示提示符.
11 if [ ! -e "$file" ]
12 then   # 如果没有这个文件那就进来.
13   echo "File $file not found."
14   exit $E_NOSUCHFILE
15 fi
```

```
16
17 read -p "Title: " title
18 cat - $file <<<$title > $file.new
19
20 echo "Modified file is $file.new"
21
22 exit 0
23
24 # 下边是'man bash'中的一段:
25 # Here Strings
26 #  here document的一种变形,形式如下:
27 #
28 #   <<<word
29 #
30 #   word被扩展并且提供到command的标准输入中.
```

##################################End Script##

练习: 找出here string的其他用法.

第18章 休息时间

==========

这个神奇的暂停可以给读者一个休息的机会, 可能读者到了这里也会会心一笑吧.

Linux同志们, 向你们致敬! 你正在阅读的这些东西, 将会给你们带来好运. 把这份文档发给你的10个朋友. 在拷贝这份文档之前, 在信的结尾写上一个100行的Bash脚本发送给列表上的第一个人. 然后在信的底部删除它们的名字并添加你自己的名字.

不要打断这个链条! 并且在48小时之内完成它.

Brooklyn的Wilfred?P.没有成功的发送他的10个拷贝, 当他第2天早上醒来发现他的工作变成了"COBOL 程序员". Newport?News的Howard?L.在一个月内才发出了他的10个拷贝, 这个时间足够建立一个100个节点的Beowulf cluster来玩Tuxracer了. Chicago的Amelia?V.对这封信付之一笑并且打断了这个链条, 不久之后, 她的终端爆炸了, 她现在花了好多天时间为MS Windows写文档.

千万不要打断这个链条! 今天就把10个拷贝发出去!

[]

高级Bash脚本编程指南(五)（上）

文章整理: 文章来源: 网络

高级Bash脚本编程指南(五)

第四部分 高级

++++++++++++++++

到了这儿,我们将要准备深入脚本编程中一些难的,不寻常的话题.随着话题的展开,我们会以多种方法和检测边界条件的方式来“ 打开信封 ”,看个明白.(当我们涉足未知领域时会发生什么?).

目录

23. 函数
24. 别名(Aliases)
25. 列表结构
26. 数组
27. /dev和/proc
28. 关于Zeros和Nulls
29. 调试
30. 选项
31. 检查遗漏(Gotchas)
32. 脚本编程风格
33. 杂项
34. Bash,版本2和3

第19章 正则表达式

==========

为了充分发挥shell编程的威力, 你需要精通正则表达式. 一些命令和软件包普遍在脚本编程中使用正则表达式,例如grep, expr, sed和awk.

19.1 一个简要的正则表达式介绍

一个正式表达式是一个字符串.字符串里的字符被称为元字符,它们可能表示了比它们字面上看起来的意思更丰富的含义.例如,一个引用符号可能表示引用一个人演讲中的话,或者表示下面将要讲到的引申表示的意思.正则表达式是一个字符或/和元字符组合成的字符集,它们匹配(或指定)一个模式.

一个正则表达式包含下面一个或多个项:

1. 一个字符集.
   这里的字符集里的字符表示的就是它们字面上的意思.正则表达式最简单的情况就是仅仅由字符集组成,而没有其他的元字符.
2. 锚.
   一个锚指明了正则表达式在一行文本中要匹配的位置,例如^和$就是锚.
3. 修饰符
   它们用于展开或缩小(即是修改了)正则表达式匹配文本行的范围.修饰符包括了星号.
   括号和反斜杠符号.

正则表达是的主要作用是用来文本搜索和字串操作.一个正则表达式匹配一个字符或是一串字符--完整的一串字符或是另外一个字符串的子串.

星号 -- * -- 匹配前一个字符的任意多次(包括零次).
  "1133*"匹配11 + 一个或更多的3 + 可能的其他字符: 113, 1133, 111312, 等等.

点 -- . -- 匹配除了新行符之外的任意一个字符. [1]
  "13." 匹配13 + 至少一个任意字符(包括空格): 1133, 11333, 但不匹配 13
  (因为少了附加的至少一个任意字符).

脱字符 -- ^ -- 匹配一行的开头,但依赖于上下文环境,可能在正则表达式中表示否定一个字符集的意思.

美元符 -- $ -- 在正则表达式中匹配行尾.
  "^$" 匹配空行.

方括号 -- [...] -- 在正则表达式中表示匹配括号中的一个字符.
  "[xyz]" 匹配字符x, y, 或z.
  "[c-n]" 匹配从字符c到n之间的任意一个字符.
  "[B-Pk-y]" 匹配从B到P 或从k到y的任意一个字符.
  "[a-z0-9]" 匹配任意小写字母或数字.
  "[^b-d]" 匹配除了从b到d范围内所有的字符. 这是正则表达式中反转意思或取否

的一 个例子.(就好像在别的情形中!字符所扮演的角色).
多个方括号字符集组合使用可以匹配一般的单词和数字模式."[Yy][Ee][Ss]" 匹配yes, Yes, YES, yEs, 等等.
"[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]"匹配社会安全码(Social Security number).

反斜杠字符 -- \ -- 转义(escapes) 一个特殊的字符,使这个字符表示原来字面上的意思.
"\$"表示了原来的字面意思"$",而不是在正则表达式中表达的匹配行尾的意思.
同样,"\\"也被解释成了字面上的意思"\".

转义(escape)"尖角号" -- \<...\> -- 用于表示单词的边界.
尖角号必须被转义,因为不这样做的话它们就表示单纯的字面意思而已.
"\<the\>" 匹配单词"the",但不匹配"them", "there", "other", 等等.

```
bash$ cat textfile
This is line 1, of which there is only one instance.
This is the only instance of line 2.
This is line 3, another line.
This is line 4.


bash$ grep 'the' textfile
This is line 1, of which there is only one instance.
This is the only instance of line 2.
This is line 3, another line.


bash$ grep '\<the\>' textfile
This is the only instance of line 2.
```

确定正则表达式能否工作的唯一办法是测试它.

```
 1 TEST FILE: tstfile                    # 不匹配.
 2                                       # 不匹配.
 3 Run   grep "1133*"  on this file.     # 匹配.
 4                                       # 不匹配.
 5                                       # 不匹配.
 6 This line contains the number 113.    # 匹配.
 7 This line contains the number 13.     # 不匹配.
 8 This line contains the number 133.    # 不匹配.
 9 This line contains the number 1133.   # 匹配.
10 This line contains the number 113312. # 匹配.
11 This line contains the number 1112.   # 不匹配.
12 This line contains the number 113312312.  # 匹配.
13 This line contains no numbers at all.     # 不匹配.
bash$ grep "1133*" tstfile
Run   grep "1133*"  on this file.       # 匹配.
This line contains the number 113.      # 匹配.
This line contains the number 1133.     # 匹配.
This line contains the number 113312.   # 匹配.
This line contains the number 113312312.  # 匹配.
```

扩展的正则表达式. 增加了一些元字符到上面提到的基本的元字符集合里. 它们在egrep, awk,和Perl中使用.

问号 -- ? -- 匹配零或一个前面的字符. 它一般用于匹配单个字符.

加号 -- + -- 匹配一个或多个前面的字符.它的作用和*很相似,但唯一的区别是它不匹配零个字符的情况.

```
1 # GNU 版本的 sed 和 awk 可以使用"+",
2 # 但它应该转义一下.
3
4 echo a111b | sed -ne '/a1\+b/p'
5 echo a111b | grep 'a1\+b'
6 echo a111b | gawk '/a1+b/'
7 # 上面三句都是等价的效果.
8
9 # 多谢, S.C.
```

转义"大括号" -- \{ \} -- 指示前面正则表达式匹配的次数.

要转义是因为不转义的话大括号只是表示他们字面上的意思.这个用法只是技巧上的而不是基本正则表达式的内容.

"[0-9]\{5\}" 精确匹配5个数字 (从 0 到 9的数字).

注意: 大括号不能在 “ 经典 ” (不是POSIX兼容)的正则表达式版本的awk中使用. 然而, gawk 有一个选项--re-interval来允许使用大括号(不必转义).

```
bash$ echo 2222 | gawk --re-interval '/2{3}/'
2222
```

Perl和一些egrep版本不要求转义大括号.

圆括号 -- ( ) -- 括起一组正则表达式. 它和下面要讲的"|"操作符或在用expr进行子字符串提取(substring extraction)一起使用很有用.

竖线 -- | -- "或"正则操作符用于匹配一组可选的字符.

```
bash$ egrep 're(a|e)d' misc.txt
People who read seem to be better informed than those who do not.
The clarinet produces sound by the vibration of its reed.
```

注意: 一些sed, ed, 和ex的版本像GNU的软件版本一样支持上面描述的扩展正则表达式的版本.

POSIX字符类. [:class:]

这是另外一个可选的用于指定匹配字符范围的方法.

[:alnum:] 匹配字母和数字.等同于A-Za-z0-9.

[:alpha:] 匹配字母. 等同于A-Za-z.

[:blank:] 匹配一个空格或是一个制表符(tab).

[:cntrl:] 匹配控制字符.

[:digit:] 匹配(十进制)数字. 等同于0-9.

[:graph:] (可打印的图形字符). 匹配 ASCII 码值的33 - 126之间的字符. 这和下面提到的[:print:]一样,但是不包括空格字符.

[:lower:] 匹配小写字母. 等同于a-z.

[:print:] (可打印字符). 匹配 ASCII码值 32 - 126之间的字符. 这和上面提到的一样[:graph:],但是增多一个空格字符.

[:space:] 匹配空白字符 (空格符和水平制表符).

[:upper:] 匹配大写字母. 等同于A-Z.

[:xdigit:] 匹配十六进制数字. 等同于0-9A-Fa-f.

注意: POSIX字符类一般都要求用引号或是双方括号double brackets ([[ ]])引起来.

```
bash$ grep [[:digit:]] test.file
abc=723
```

这些字符类在一个受限的范围内甚至可能用在能用在通配(globbing)中.

```
bash$ ls -l ?[[:digit:]][[:digit:]]?
-rw-rw-r--    1 bozo  bozo         0 Aug 21 14:47 a33b
```

为了理解POSIX字符类在脚本中的使用,请参考例子 12-18 和 例子 12-19.

Sed, awk, 和Perl在脚本中被用作过滤器, "过滤"或转换文件/IO流的时候以正则表达式作为参数.参考例子 A-12和例子 A-17 来理解这种用法.

在正则表达式这个复杂主题的标准参考是Friedl的Mastering Regular Expressions.由Dougherty和Robbins写的 Sed & Awk也给出了一个清晰的正则表达式论述. 查看参考书目找到这个主题更多的信息.

注意事项:

[1] 因为sed, awk, 和 grep 通常处理单行,而不能匹配一个新行符. 在要处理多行的一个输入时,可以使用点操作符,它可以匹配新行符.

```
 1 #!/bin/bash
 2
 3 sed -e 'N;s/.*/[&]/' << EOF   # Here Document
 4 line1
 5 line2
 6 EOF
 7 # 输出:
 8 # [line1
 9 # line2]
10
11
12
13 echo
14
15 awk '{ $0=$1 "\n" $2; if (/line.1/) {print}}' << EOF
16 line 1
17 line 2
18 EOF
19 # 输出:
20 # line
21 # 1
22
23
24 # 多谢, S.C.
25
26 exit 0
```

19.1 通配

Bash本身没有正则表达式的功能.在脚本里,使用正则表达式的是命令和软件包 -- 例如sed和awk -- 它们可以解释正则表达式.

Bash所做的是展开文件名扩展 [1] -- 这就是所谓的通配(globbing) -- 但它不是使用标准的正则表达式. 而是使用通配符. 通配解释标准的通配符：*和?, 方括号括起来的字符,还有其他的一些特殊的字符(比如说^用来表示取反匹配).然而通配机制的通配符有很大的局限性. 包含有*号的字符串将不会匹配以点开头的文件,例如.bashrc. [2] 另外,通配机制的? 字符和正则

表达式中表示的意思不一样.

```
bash$ ls -l
total 2
-rw-rw-r--    1 bozo  bozo        0 Aug  6 18:42 a.1
-rw-rw-r--    1 bozo  bozo        0 Aug  6 18:42 b.1
-rw-rw-r--    1 bozo  bozo        0 Aug  6 18:42 c.1
-rw-rw-r--    1 bozo  bozo      466 Aug  6 17:48 t2.sh
-rw-rw-r--    1 bozo  bozo      758 Jul 30 09:02 test1.txt

bash$ ls -l t?.sh
-rw-rw-r--    1 bozo  bozo      466 Aug  6 17:48 t2.sh

bash$ ls -l [ab]*
-rw-rw-r--    1 bozo  bozo        0 Aug  6 18:42 a.1
-rw-rw-r--    1 bozo  bozo        0 Aug  6 18:42 b.1

bash$ ls -l [a-c]*
-rw-rw-r--    1 bozo  bozo        0 Aug  6 18:42 a.1
-rw-rw-r--    1 bozo  bozo        0 Aug  6 18:42 b.1
-rw-rw-r--    1 bozo  bozo        0 Aug  6 18:42 c.1

bash$ ls -l [^ab]*
-rw-rw-r--    1 bozo  bozo        0 Aug  6 18:42 c.1
-rw-rw-r--    1 bozo  bozo      466 Aug  6 17:48 t2.sh
-rw-rw-r--    1 bozo  bozo      758 Jul 30 09:02 test1.txt

bash$ ls -l {b*,c*,*est*}
-rw-rw-r--    1 bozo  bozo        0 Aug  6 18:42 b.1
-rw-rw-r--    1 bozo  bozo        0 Aug  6 18:42 c.1
-rw-rw-r--    1 bozo  bozo      758 Jul 30 09:02 test1.txt
```

Bash会对命令行中没有引号引起来的字符尝试文件名扩展. echo 命令可以印证这一点.

```
bash$ echo *
a.1 b.1 c.1 t2.sh test1.txt

bash$ echo t*
t2.sh test1.txt
```

注意: 可以改变Bash对通配字符进行解释的行为. set -f 命令可以禁止通配机制, 并且 shopt的选项nocaseglob和nullglob 能改变通配的行为.

参考例子 10-4.

注意事项:

[1] 文件名扩展意思是扩展包含有特殊字符的文件名模式和模板. 例如,example.???可能扩展成example.001和/或example.txt.

[2] 文件名扩展能匹配点开头的文件,但仅在模式字串明确地包含字面意思的点(.)时才扩展.

```
1 ~/[.]bashrc    #  不会扩展成 ~/.bashrc
2 ~/?bashrc      #  也不会扩展.
3                #  通配机制中的通配符和元字符不会扩展点文件
4                #
```

```
 5
 6 ~/.[b]ashrc    #  会扩展成 ~/.bashrc
 7 ~/.ba?hrc      #  也会.
 8 ~/.bashr*      #  也会.
 9
10 # 可以使用"dotglob"选项把这个特性禁用.
11
12 # 多谢, S.C.
```

第20章 子shell(Subshells)

==========

运行一个shell脚本时会启动另一个命令解释器. 就好像你的命令是在命令行提示下被解释的一样, 类似于批处理文件里的一系列命令.每个shell脚本有效地运行在父shell(parent shell)的一个子进程里.这个父shell是指在一个控制终端或在一个xterm窗口中给你命令指示符的进程. shell脚本也能启动他自己的子进程. 这些子shell(即子进程)使脚本因为效率而同时进行多个子任务执行时能做串行处理.

一般来说,脚本里的一个外部命令(external command)能生成(forks)出一个子进程,然而Bash内建(builtin)的命令却不这样做,因此,内建命令比起外部的等价命令执行起来更快.

圆括号里的命令列表

(命令1; 命令2; 命令3; ...)

嵌在圆括号里的一列命令在一个子shell里运行.

注意: 在子shell里的变量不能被这段子shell代码块之外外面的脚本访问.这些变量是不能被产生这个子shell的父进程(parent process)存取的,实际上它们是局部变量(local variables).

Example 20-1 子shell中的变量作用域

##################################Start Script###

```
 1 #!/bin/bash
 2 # subshell.sh
 3
 4 echo
 5
 6 echo "Subshell level OUTSIDE subshell = $BASH_SUBSHELL"
 7 # Bash, 版本 3, 增加了新的            $BASH_SUBSHELL 变量.
 8 echo
 9
10 outer_variable=Outer
11
12 (
13 echo "Subshell level INSIDE subshell = $BASH_SUBSHELL"
14 inner_variable=Inner
15
16 echo "From subshell, \"inner_variable\" = $inner_variable"
17 echo "From subshell, \"outer\" = $outer_variable"
18 )
19
20 echo
21 echo "Subshell level OUTSIDE subshell = $BASH_SUBSHELL"
22 echo
23
```

```
24 if [ -z "$inner_variable" ]
25 then
26   echo "inner_variable undefined in main body of shell"
27 else
28   echo "inner_variable defined in main body of shell"
29 fi
30
31 echo "From main body of shell, \"inner_variable\" = $inner_variable"
32 #  $inner_variable 会以没有初始化的变量来打印
33 #+ 因为变量是在子shell里定义的"局部变量".
34 #  这个有办法补救的吗?
35
36 echo
37
38 exit 0
```

################################End Script###

参考例子 31-2.

+

在子shell中的目录更改不会影响到父shell.

Example 20-2 列出用户的配置文件

################################Start Script###

```
 1 #!/bin/bash
 2 # allprofs.sh: 打印所有用户的配置文件
 3
 4 # 由 Heiner Steven编写, 并由本书作者修改.
 5
 6 FILE=.bashrc  #  在一般的脚本里,包含用户配置的文件是".profile".
 7             #
 8
 9 for home in `awk -F: '{print $6}' /etc/passwd`
10 do
11   [ -d "$home" ] || continue    # 如果没有家目录,跳过此次循环.
12   [ -r "$home" ] || continue    # 如果目录没有读权限,跳过此次循环.
13   (cd $home; [ -e $FILE ] && less $FILE)
14 done
15
16 #  当脚本终止时,不必用'cd'命令返回原来的目录,
17 #+ 因为'cd $home'是在子shell中发生的,不影响父shell.
18
19 exit 0
```

################################End Script###

子shell可用于为一组命令设定临时的环境变量.

```
  1 COMMAND1
  2 COMMAND2
  3 COMMAND3
  4 (
  5   IFS=:
  6   PATH=/bin
```

```
  7   unset TERMINFO
  8   set -C
  9   shift 5
 10   COMMAND4
 11   COMMAND5
 12   exit 3 # 只是从子shell退出.
 13 )
 14 # 父shell不受影响,变量值没有更改.
 15 COMMAND6
 16 COMMAND7
```

它的一个应用是测试是否一个变量被定义了.

```
  1 if (set -u; : $variable) 2> /dev/null
  2 then
  3   echo "Variable is set."
  4 fi    #  变量已经在当前脚本中被设置,
  5       #+ 或是Bash的一个内部变量,
  6       #+ 或是可见环境变量(指已经被导出的环境变量).
  7
  8 # 也可以写成          [[ ${variable-x} != x || ${variable-y} != y ]]
  9 # 或                 [[ ${variable-x} != x$variable ]]
 10 # 或                  [[ ${variable+x} = x ]]
 11 # 或                  [[ ${variable-x} != x ]]
```

另一个应用是检查一个加锁的文件:

```
  1 if (set -C; : > lock_file) 2> /dev/null
  2 then
  3   :  # lock_file 不存在,还没有用户运行这个脚本
  4 else
  5   echo "Another user is already running that script."
  6 exit 65
  7 fi
  8
  9 #  由St    phane Chazelas编程
 10 #+ 由Paulo Marcel Coelho Aragao修改.
```

进程在不同的子shell中可以串行地执行.这样就允许把一个复杂的任务分成几个小的子问题来同时地处理.

Example 20-3 在子shell里进行串行处理

```
################################Start Script#########################################
 1  (cat list1 list2 list3 | sort | uniq > list123) &
 2  (cat list4 list5 list6 | sort | uniq > list456) &
 3  #列表的合并和排序同时进.
 4  #放到后台运行可以确保能够串行执行.
 5  #
 6  #和下面的有相同的作用：
 7  #   cat list1 list2 list3 | sort | uniq > list123 &
 8  #   cat list4 list5 list6 | sort | uniq > list456 &
 9
10  wait   #在所有的子shell执行完成前不再执行后面的命令.
11
```

12 diff list123 list456
################################End Script###
用"|"管道操作把I/O流重定向到子shell,例如ls -al | (command).
注意: 在一个花括号内的代码块不会运行一个子shell.
 { command1; command2; command3; ... }
第21章 受限shell(Restricted Shells)
==================================
在受限shell中禁用的命令
   在受限shell中运行的脚本或脚本的个代码断会禁用一些正常shell中可以执行的命令.这是
限制脚本用户的权限和最小化运行脚本导致的破坏的安全措施.
   使用cd 命令更改工作目录.
   更改环境变量$PATH, $SHELL, $BASH_ENV,或$ENV 的值.
   读或更改shell环境选项变量$SHELLOPTS的值.
   输出重定向.
   调用的命令路径中包括有一个或更多个/字符.
   调用exec来把当前的受限shell替换成另外一个不同的进程.
   脚本中许多其他无意中能破坏或捣乱的命令.
   在脚本中企图脱离受限shell模式的操作.
Example 21-1 在受限的情况下运行脚本
################################Start Script###
```
 1 #!/bin/bash
 2
 3 #  脚本开头以"#!/bin/bash -r"来调用
 4 #+ 会使整个脚本在受限模式下运行.
 5
 6 echo
 7
 8 echo "Changing directory."
 9 cd /usr/local
10 echo "Now in `pwd`"
11 echo "Coming back home."
12 cd
13 echo "Now in `pwd`"
14 echo
15
16 # 不受限的模式下,所有操作都能正常成功.
17
18 set -r
19 # set --restricted    也能起相同的作用.
20 echo "==> Now in restricted mode. <=="
21
22 echo
23 echo
24
25 echo "Attempting directory change in restricted mode."
26 cd ..
27 echo "Still in `pwd`"
28
```

```
29 echo
30 echo
31
32 echo "\$SHELL = $SHELL"
33 echo "Attempting to change shell in restricted mode."
34 SHELL="/bin/ash"
35 echo
36 echo "\$SHELL= $SHELL"
37
38 echo
39 echo
40
41 echo "Attempting to redirect output in restricted mode."
42 ls -l /usr/bin > bin.files
43 ls -l bin.files    # Try to list attempted file creation effort.
44
45 echo
46
47 exit 0
```

################################End Script###

第22章 进程替换

==========

进程替换与命令替换(command substitution)很相似. 命令替换把一个命令的结果赋给一个变量,例如 dir_contents=`ls -al`或xref=$(grep word datafile). 进程替换则是把一个进程的输出回馈给另一个进程 (换句话说,它把一个命令的结果发送给另一个命令).

命令替换的一般形式

由圆括号括起的命令

>(command)

<(command)

启动进程替换. 它是用/dev/fd/<n>文件把在圆括号内的进程的处理结果发送给另外一个进程. [1] (译者注：实际上现代的UNIX类操作系统提供的/dev/fd/n文件是与文件描述相关的,整数n指的就是在进程运行时对应数字的文件描述符)

注意: 在"<" 或or ">" 与圆括号之间是没有空格的. 如果加了空格将会引起错误信息.

```
bash$ echo >(true)
/dev/fd/63

bash$ echo <(true)
/dev/fd/63
```

Bash在两个文件描述符(file descriptors)之间创建了一个管道, --fIn 和 fOut--. true命令的标准输入被连接到fOut(dup2(fOut, 0)), 然后Bash把/dev/fd/fIn作为参数传给echo.

如果系统的/dev/fd/<n>文件不够时,Bash会使用临时文件. (Thanks, S.C.)

进程替换能比较两个不同命令之间的输出,或者甚至相同命令不同选项的输出.

```
bash$ comm <(ls -l) <(ls -al)
total 12
-rw-rw-r--    1 bozo bozo       78 Mar 10 12:58 File0
-rw-rw-r--    1 bozo bozo       42 Mar 10 12:58 File2
-rw-rw-r--    1 bozo bozo      103 Mar 10 12:58 t2.sh
      total 20
```

```
drwxrwxrwx    2 bozo bozo     4096 Mar 10 18:10 .
drwx------   72 bozo bozo     4096 Mar 10 17:58 ..
-rw-rw-r--    1 bozo bozo       78 Mar 10 12:58 File0
-rw-rw-r--    1 bozo bozo       42 Mar 10 12:58 File2
-rw-rw-r--    1 bozo bozo      103 Mar 10 12:58 t2.sh
```

用进程替换来比较两个不同目录的内容 (考察哪些文件名是相同的,哪些是不同的):

```
1 diff <(ls $first_directory) <(ls $second_directory)
```

其他一些进程替换的用法和技巧:

```
1 cat <(ls -l)
2 # 等同于     ls -l | cat
3
4 sort -k 9 <(ls -l /bin) <(ls -l /usr/bin) <(ls -l /usr/X11R6/bin)
5 # 列出系统中3个主要的'bin'目录的所有文件,并且按文件名排序.
6 # 注意是三个明显不同的命令输出回馈给'sort'.
7
8
9 diff <(command1) <(command2)    # 给出两个命令输出的不同之处.
10
11 tar cf >(bzip2 -c > file.tar.bz2) $directory_name
12 # 调用"tar cf /dev/fd/?? $directory_name",和"bzip2 -c > file.tar.bz2".
13 #
14 # 因为/dev/fd/<n>的系统属性,
15 # 所以两个命令之间的管道不必是命名的.
16 #
17 # 这种效果可以模仿出来.
18 #
19 bzip2 -c < pipe > file.tar.bz2&
20 tar cf pipe $directory_name
21 rm pipe
22 #       或者
23 exec 3>&1
24 tar cf /dev/fd/4 $directory_name 4>&1 >&3 3>&- | bzip2 -c > file.tar.bz2 3>&-
25 exec 3>&-
26
27
28 # Thanks, St`phane Chazelas
```

有个读者给我发来下面关于进程替换的有趣例子A.

```
1 # 摘自SuSE发行版中的代码片断:
2
3 while read  des what mask iface; do
4 # 这里省略了一些命令 ...
5 done < <(route -n)
6
7
8 # 为了测试它,我们来做些动作.
9 while read  des what mask iface; do
10   echo $des $what $mask $iface
11 done < <(route -n)
```

```
12
13 # 输出:
14 # Kernel IP routing table
15 # Destination Gateway Genmask Flags Metric Ref Use Iface
16 # 127.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 lo
17
18
19
20 # 由 St    phane Chazelas给出的,一个更容易理解的等价代码是:
21 route -n |
22   while read des what mask iface; do   # 管道的输出被赋给了变量.
23     echo $des $what $mask $iface
24   done  #  这样就取回了和上面一样的输出.
25         #  但是, Ulrich Gayer指出 . . .
26         #+ 这个简单版本的等价代码在while循环中使用了一个子shell,
27         #+ 因此当管道结束后变量会被毁掉.
28
29
30
31 #  更进一步, Filip Moritz解释了上面两个例子之间有一个细微的不同之处
32 #+ 如下所示.
33
34 (
35 route -n | while read x; do ((y++)); done
36 echo $y # $y 仍然没有被声明或设置
37
38 while read x; do ((y++)); done < <(route -n)
39 echo $y # $y的值为 route -n 输出的行数
40 )
41
42 # 一般来说
43 (
44 : | x=x
45 # 看上去是启动了一个子shell
46 : | ( x=x )
47 # 但
48 x=x < <(:)
49 # 实际上不是
50 )
51
52 # 当解析csv或类似的东西时非常有用.
53 # 事实上,这就是SuSE原本的代码片断所要实现的功能.
```

注意事项:

[1] 这与命名管道(named pipe)(临时文件)有相同的作用, 事实上命名管道同样在进程替换中被使用.

第23章 函数

==========

和"真正的"编程语言一样, Bash也有函数,虽然在某些实现方面稍有些限制. 一个函数是一个

子程序,用于实现一串操作的代码块(code block),它是完成特定任务的"黑盒子". 当有重复代码, 当一个任务只需要很少的修改就被重复几次执行时, 这时你应考虑使用函数.

```
function function_name {
command...
}
```

或

```
function_name () {
command...
}
```

第二种格式的写法更深得C程序员的喜欢(并且也是更可移植的).

因为在C中,函数的左花括号也可以写在下一行中.

```
function_name ()
{
command...
}
```

函数被调用或被触发, 只需要简单地用函数名调用.

Example 23-1 简单函数

```
##################################Start Script#############################################
 1 #!/bin/bash
 2
 3 JUST_A_SECOND=1
 4
 5 funky ()
 6 { # 这是一个最简单的函数.
 7   echo "This is a funky function."
 8   echo "Now exiting funky function."
 9 } # 函数必须在调用前声明.
10
11
12 fun ()
13 { # 一个稍复杂的函数.
14   i=0
15   REPEATS=30
16
17   echo
18   echo "And now the fun really begins."
19   echo
20
21   sleep $JUST_A_SECOND    # 嘿, 暂停一秒!
22   while [ $i -lt $REPEATS ]
23   do
24     echo "----------FUNCTIONS---------->"
25     echo "<------------ARE-------------"
26     echo "<------------FUN------------>"
27     echo
28     let "i+=1"
29   done
30 }
```

```
31
32  # 现在,调用两个函数.
33
34 funky
35 fun
36
37 exit 0
```

################################End Script###

函数定义必须在第一次调用函数前完成.没有像C中的函数“ 声明 ”方法.

```
 1 f1
 2 # 因为函数"f1"还没有定义,这会引起错误信息.
 3
 4 declare -f f1      # 这样也没用.
 5 f1                 # 仍然会引起错误.
 6
 7 # 然而...
 8
 9
10 f1 ()
11 {
12   echo "Calling function \"f2\" from within function \"f1\"."
13   f2
14 }
15
16 f2 ()
17 {
18   echo "Function \"f2\"."
19 }
20
21 f1  #  虽然在它定义前被引用过,
22     #+ 函数"f2"实际到这儿才被调用.
23     #  这样是允许的.
24
25     # Thanks, S.C.
```

在一个函数内嵌套另一个函数也是可以的,但是不常用.

```
 1 f1 ()
 2 {
 3
 4   f2 () # nested
 5   {
 6     echo "Function \"f2\", inside \"f1\"."
 7   }
 8
 9 }
10
11 f2  #  引起错误.
12     #  就是你先"declare -f f2"了也没用.
13
```

```
14 echo
15
16 f1  #  什么也不做,因为调用"f1"不会自动调用"f2".
17 f2  #  现在,可以正确的调用"f2"了,
18     #+ 因为之前调用"f1"使"f2"在脚本中变得可见了.
19
20     # Thanks, S.C.
```

函数声明可以出现在看上去不可能出现的地方,那些不可能的地方本该由一个命令出现的地方.

```
1 ls -l | foo() { echo "foo"; }  # 允许,但没什么用.
2
3
4
5 if [ "$USER" = bozo ]
6 then
7   bozo_greet ()   # 在if/then结构中定义了函数.
8   {
9     echo "Hello, Bozo."
10  }
11 fi
12
13 bozo_greet       # 只能由Bozo运行, 其他用户会引起错误.
14
15
16
17 # 在某些上下文,像这样可能会有用.
18 NO_EXIT=1   # 将会打开下面的函数定义.
19
20 [[ $NO_EXIT -eq 1 ]] && exit() { true; }     # 在"and-list"(and列表)中定义函数.
21 # 如果 $NO_EXIT 是 1,声明函数"exit ()".
22 # 把"exit"取别名为"true"将会禁用内建的"exit".
23
24 exit  # 调用"exit ()"函数, 而不是内建的"exit".
25
26 # Thanks, S.C.
```

23.1. 复杂函数和函数复杂性

函数可以处理传递给它的参数并且能返回它的退出状态码(exit status)给脚本后续使用.

```
1 function_name $arg1 $arg2
```

函数以位置来引用传递过来的参数(就好像他们是位置参数(positional parameters)), 例如 $1, $2,以此类推.

Example 23-2 带着参数的函数

```
##################################Start Script#########################################
1 #!/bin/bash
2 # 函数和参数
3
4 DEFAULT=default                        # 默认的参数值.
5
6 func2 () {
```

```
 7  if [ -z "$1" ]                   # 第一个参数是否长度为零?
 8  then
 9    echo "-Parameter #1 is zero length.-"  # 则没有参数传递进来.
10  else
11    echo "-Param #1 is \"$1\".-"
12  fi
13
14  variable=${1-$DEFAULT}                #
15  echo "variable = $variable"           # 参数替换会表现出什么?
16                                        # --------------------------
17                                        # 它用于分辨没有参数和一个只有NULL值的参数.
18                                        #
19
20  if [ "$2" ]
21  then
22    echo "-Parameter #2 is \"$2\".-"
23  fi
24
25  return 0
26 }
27
28 echo
29
30 echo "Nothing passed."
31 func2                      # 没有参数来调用
32 echo
33
34
35 echo "Zero-length parameter passed."
36 func2 ""                    # 以一个长度为零的参数调用
37 echo
38
39 echo "Null parameter passed."
40 func2 "$uninitialized_param"   # 以未初始化的参数来调用
41 echo
42
43 echo "One parameter passed."
44 func2 first          # 用一个参数来调用
45 echo
46
47 echo "Two parameters passed."
48 func2 first second    # 以二个参数来调用
49 echo
50
51 echo "\"\" \"second\" passed."
52 func2 "" second      # 以第一个参数为零长度,而第二个参数是一个ASCII码组成的字符串来调用.
53 echo                 #
54
```

```
55 exit 0
```

################################End Script###

注意: shift命令可以工作在传递给函数的参数 (参考例子 33-15).

但是,传给脚本的命令行参数怎么办?在函数内部可以看到它们吗?好,让我们来弄清楚.

Example 23-3 函数和被传给脚本的命令行参数

################################Start Script###

```
 1 #!/bin/bash
 2 # func-cmdlinearg.sh
 3 #  以一个命令行参数来调用这个脚本,
 4 #+ 类似 $0 arg1来调用.
 5
 6
 7 func ()
 8
 9 {
10 echo "$1"
11 }
12
13 echo "First call to function: no arg passed."
14 echo "See if command-line arg is seen."
15 func
16 # 不!命令行参数看不到.
17
18 echo "============================================================"
19 echo
20 echo "Second call to function: command-line arg passed explicitly."
21 func $1
22 # 现在可以看到了!
23
24 exit 0
```

################################End Script###

与别的编程语言相比,shell脚本一般只传递值给函数,变量名(实现上是指针)如果作为参数传递给函数会被看成是字面上字符串的意思.函数解释参数是以字面上的意思来解释的.

间接变量引用(Indirect variable references) (参考例子 34-2)提供了传递变量指针给函数的一个笨拙的机制.

Example 23-4 传递间接引用给函数

################################Start Script###

```
 1 #!/bin/bash
 2 # ind-func.sh: 传递间接引用给函数.
 3
 4 echo_var ()
 5 {
 6 echo "$1"
 7 }
 8
 9 message=Hello
10 Hello=Goodbye
11
12 echo_var "$message"        # Hello
```

```
13 # 现在,让我们传递一个间接引用给函数.
14 echo_var "${!message}"     # Goodbye
15
16 echo "-------------"
17
18 # 如果我们改变"hello"变量的值会发生什么?
19 Hello="Hello, again!"
20 echo_var "$message"        # Hello
21 echo_var "${!message}"     # Hello, again!
22
23 exit 0
```

################################End Script###

下一个逻辑问题是：在传递参数给函数之后是否能解除参数的引用.

Example 23-5 解除传递给函数的参数引用

################################Start Script###

```
 1 #!/bin/bash
 2 # dereference.sh
 3 # 给函数传递不同的参数.
 4 # Bruce W. Clare编写.
 5
 6 dereference ()
 7 {
 8     y=\$"$1"   # 变量名.
 9     echo $y    # $Junk
10
11     x=`eval "expr \"$y\" "`
12     echo $1=$x
13     eval "$1=\"Some Different Text \""  # 赋新值.
14 }
15
16 Junk="Some Text"
17 echo $Junk "before"    # Some Text before
18
19 dereference Junk
20 echo $Junk "after"     # Some Different Text after
21
22 exit 0
```

################################End Script###

Example 23-6 再次尝试解除传递给函数的参数引用

################################Start Script###

```
 1 #!/bin/bash
 2 # ref-params.sh: 解除传递给函数的参数引用.
 3 #              (复杂例子)
 4
 5 ITERATIONS=3  # 取得输入的次数.
 6 icount=1
 7
 8 my_read () {
```

```
 9  #  用my_read varname来调用,
10  #+ 输出用括号括起的先前的值作为默认值,
11  #+ 然后要求输入一个新值.
12
13  local local_var
14
15  echo -n "Enter a value "
16  eval 'echo -n "[$'$1'] "'  #  先前的值.
17 # eval echo -n "[\$$1] "     #  更好理解,
18                          #+ 但会丢失用户输入在尾部的空格.
19  read local_var
20  [ -n "$local_var" ] && eval $1=\$local_var
21
22  # "and列表(And-list)": 如果变量"local_var"测试成功则把变量"$1"的值赋给它.
23 }
24
25 echo
26
27 while [ "$icount" -le "$ITERATIONS" ]
28 do
29  my_read var
30  echo "Entry #$icount = $var"
31  let "icount += 1"
32  echo
33 done
34
35
36 # 多谢Stephane Chazelas提供的示范例子.
37
38 exit 0
```

################################End Script###

退出和返回

退出状态(exit status)

函数返回一个被称为退出状态的值. 退出状态可以由return来指定statement, 否则函数的退出状态是函数最后一个执行命令的退出状态(0表示成功,非0表示出错代码). 退出状态(exit status)可以在脚本中由$? 引用. 这个机制使脚本函数也可以像C函数一样有一个"返回值".

return

终止一个函数.return 命令[1]可选地带一个整数参数,这个整数作为函数的"返回值"返回给调用此函数的脚本,并且这个值也被赋给变量$?.

Example 23-7 两个数中的最大者

################################Start Script###

```
 1 #!/bin/bash
 2 # max.sh: 两个整数中的最大者.
 3
 4 E_PARAM_ERR=-198    # 如果传给函数的参数少于2个时的返回值.
 5 EQUAL=-199         # 如果两个整数值相等的返回值.
 6 #  任一个传给函数的参数值溢出
```

```
 7 #
 8
 9 max2 ()            # 返回两个整数的较大值.
10 {                 # 注意: 参与比较的数必须小于257.
11 if [ -z "$2" ]
12 then
13   return $E_PARAM_ERR
14 fi
15
16 if [ "$1" -eq "$2" ]
17 then
18   return $EQUAL
19 else
20   if [ "$1" -gt "$2" ]
21   then
22     return $1
23   else
24     return $2
25   fi
26 fi
27 }
28
29 max2 33 34
30 return_val=$?
31
32 if [ "$return_val" -eq $E_PARAM_ERR ]
33 then
34   echo "Need to pass two parameters to the function."
35 elif [ "$return_val" -eq $EQUAL ]
36   then
37     echo "The two numbers are equal."
38 else
39     echo "The larger of the two numbers is $return_val."
40 fi
41
42
43 exit 0
44
45 #  练习 (容易):
46 #  ----------------
47 #  把这个脚本转化成交互式的脚本,
48 #+ 也就是说,让脚本可以要求调用者输入两个整数.
################################End Script#############################################
 注意: 为了函数可以返回字符串或是数组,用一个可在函数外可见的变量.
   1 count_lines_in_etc_passwd()
   2 {
   3   [[ -r /etc/passwd ]] && REPLY=$(echo $(wc -l < /etc/passwd))
   4   #  如果/etc/passwd可读,则把REPLY设置成文件的行数.
```

```
  5  # 返回一个参数值和状态信息.
  6  # 'echo'好像没有必要,但 . . .
  7  #+ 它的作用是删除输出中的多余空白字符.
  8 }
  9
 10 if count_lines_in_etc_passwd
 11 then
 12   echo "There are $REPLY lines in /etc/passwd."
 13 else
 14   echo "Cannot count lines in /etc/passwd."
 15 fi
 16
 17 # Thanks, S.C.
```

Example 23-8 把数字转化成罗马数字

```
##################################Start Script#########################################
 1 #!/bin/bash
 2
 3 # 阿拉伯数字转化为罗马数字
 4 # 转化范围: 0 - 200
 5 # 这是比较粗糙的,但可以工作.
 6
 7 # 扩展可接受的范围来作为脚本功能的扩充,这个作为练习完成.
 8
 9 # 用法: roman number-to-convert
10
11 LIMIT=200
12 E_ARG_ERR=65
13 E_OUT_OF_RANGE=66
14
15 if [ -z "$1" ]
16 then
17   echo "Usage: `basename $0` number-to-convert"
18   exit $E_ARG_ERR
19 fi
20
21 num=$1
22 if [ "$num" -gt $LIMIT ]
23 then
24   echo "Out of range!"
25   exit $E_OUT_OF_RANGE
26 fi
27
28 to_roman ()   # 在第一次调用函数前必须先定义.
29 {
30 number=$1
31 factor=$2
32 rchar=$3
33 let "remainder = number - factor"
```

```
34 while [ "$remainder" -ge 0 ]
35 do
36   echo -n $rchar
37   let "number -= factor"
38   let "remainder = number - factor"
39 done
40
41 return $number
42       # 练习:
43       # --------
44       # 解释这个函数是怎么工作的.
45       # 提示: 靠不断地除来分割数字.
46 }
47
48
49 to_roman $num 100 C
50 num=$?
51 to_roman $num 90 LXXXX
52 num=$?
53 to_roman $num 50 L
54 num=$?
55 to_roman $num 40 XL
56 num=$?
57 to_roman $num 10 X
58 num=$?
59 to_roman $num 9 IX
60 num=$?
61 to_roman $num 5 V
62 num=$?
63 to_roman $num 4 IV
64 num=$?
65 to_roman $num 1 I
66
67 echo
68
69 exit 0
```

################################End Script###

请参考例子 10-28.

注意: 函数最大可返回的正整数为255. return 命令与退出状态(exit status)的概念联系很紧密,而退出状态的值受此限制.幸运地是有多种(工作区workarounds)来对付这种要求函数返回大整数的情况.

Example 23-9 测试函数最大的返回值

################################Start Script###

```
1 #!/bin/bash
2 # return-test.sh
3
4 # 一个函数最大可能返回的值是255.
5
```

```
return_test ()        # 无论传给函数什么都返回它.
{
  return $1
}

return_test 27        # o.k.
echo $?               # 返回 27.

return_test 255       # 仍然 o.k.
echo $?               # 返回 255.

return_test 257       # 错误!
echo $?               # 返回 1 (返回代码指示错误).

# ======================================================
return_test -151896   # 能够返回这个非常大的负数么?
echo $?               # 会返回-151896?
                      # 不! 它将返回168.
#  2.05b版本之前的Bash是允许
#+ 超大负整数作为返回值的.
#  但是比它更新一点的版本修正了这个漏洞.
#  这将破坏比较老的脚本.
#  慎用!
# ======================================================

exit 0
```

##################################End Script##

如果你非常想使用超大整数作为"返回值"的话, 那么只能通过将你想返回的返回值直接的传递到一个全局变量中的手段来达到目的.

```
Return_Val=   # 全局变量, 用来保存函数中需要返回的超大整数.

alt_return_test ()
{
  fvar=$1
  Return_Val=$fvar
  return   # Returns 0 (success).
}

alt_return_test 1
echo $?                       # 0
echo "return value = $Return_Val"    # 1

alt_return_test 256
echo "return value = $Return_Val"    # 256

alt_return_test 257
echo "return value = $Return_Val"    # 257
```

```
  20 alt_return_test 25701
  21 echo "return value = $Return_Val"    #25701
```

一种更优雅的方法是让函数echo出它的返回值, 输出到stdout上, 然后再通过"命令替换"的手段来捕获它. 参考Section 33.7关于这个问题的讨论.

Example 23-10 比较两个大整数

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # max2.sh: 取两个超大整数中最大的.
 3
 4 #  这个脚本与前面的"max.sh"例子作用相同,
 5 #+ 经过修改可以适用于比较超大整数.
 6
 7 EQUAL=0            # 如果两个参数相同的返回值.
 8 E_PARAM_ERR=-99999  # 没有足够的参数传递到函数中.
 9 #          ^^^^^^    也可能是传递到函数中的某个参数超出范围了.
10
11 max2 ()           # 从这两个数中"返回"更大一些的.
12 {
13 if [ -z "$2" ]
14 then
15   echo $E_PARAM_ERR
16   return
17 fi
18
19 if [ "$1" -eq "$2" ]
20 then
21   echo $EQUAL
22   return
23 else
24   if [ "$1" -gt "$2" ]
25   then
26     retval=$1
27   else
28     retval=$2
29   fi
30 fi
31
32 echo $retval       # echo(到stdout), 而不是使用返回值.
33                    # 为什么?
34 }
35
36
37 return_val=$(max2 33001 33997)
38 #            ^^^^            函数名
39 #                 ^^^^^ ^^^^^ 这是传递进来的参数
40 #  这事实上是一个命令替换的形式:
41 #+ 会把这个函数当作一个命令来处理,
42 #+ 并且分配这个函数的stdout到变量"return_val"中.
```

```
43
44
45 # ========================= OUTPUT ========================
46 if [ "$return_val" -eq "$E_PARAM_ERR" ]
47   then
48   echo "Error in parameters passed to comparison function!"
49 elif [ "$return_val" -eq "$EQUAL" ]
50   then
51    echo "The two numbers are equal."
52 else
53    echo "The larger of the two numbers is $return_val."
54 fi
55 # =========================================================
56
57 exit 0
58
59 #  练习:
60 #  -----
61 #  1) 找出一种更优雅的方法来测试
62 #+    传递到函数中的参数.
63 #  2) 在"OUTPUT"的时候简化if/then结构.
64 #  3) 重写这个脚本使其能够从命令行参数中来获取输入.
##################################End Script##########################################
```

下边是获得一个函数的"返回值"的另一个例子. 想要了解这个例子需要一些awk的知识.

```
  1 month_length ()  # 以月份数作为参数.
  2 {                # 返回这个月有几天.
  3 monthD="31 28 31 30 31 30 31 31 30 31 30 31"  # 作为局部变量来声明?
  4 echo "$monthD" | awk '{ print $'"${1}"' }'    # 有技巧的.
  5 #                          ^^^^^^^^^
  6 # 先将参数传递到函数中  ($1 -- 月份号), 然后就到awk了.
  7 # Awk将会根据传递进来的月份号来决定打印"print $1 . . . print $12"中的哪个 (依赖于月份号)
  8 # 传递参数到内嵌awk脚本的模版:
  9 #                              $'"${script_parameter}"'
 10
 11 #  需要错误检查来修正参数的范围(1-12)
 12 #+ 并且要处理闰年的特殊的2月.
 13 }
 14
 15 # ---------------------------------------------
 16 # 用例:
 17 month=4       # 拿4月来举个例子.
 18 days_in=$(month_length $month)
 19 echo $days_in  # 30
 20 # ---------------------------------------------
```

也参考例子 A-7.

练习: 用我们已经学到的扩展先前罗马数字那个例子脚本能接受任意大的输入.

重定向

重定向函数的标准输入

函数本质上是一个代码块(code block), 这样意思着它的标准输入可以被重定向
(就像在例子 3-1中显示的).

Example 23-11 用户名的真实名

```
#################################Start Script#########################################
 1 #!/bin/bash
 2 # realname.sh
 3 #
 4 # 由用户名而从/etc/passwd取得"真实名".
 5
 6
 7 ARGCOUNT=1       # 需要一个参数.
 8 E_WRONGARGS=65
 9
10 file=/etc/passwd
11 pattern=$1
12
13 if [ $# -ne "$ARGCOUNT" ]
14 then
15   echo "Usage: `basename $0` USERNAME"
16   exit $E_WRONGARGS
17 fi
18
19 file_excerpt ()  # 以要求的模式来扫描文件,然后打印文件相关的部分.
20 {
21 while read line  # "while" does not necessarily need "[ condition ]"
22 do
23   echo "$line" | grep $1 | awk -F":" '{ print $5 }'  # awk指定使用":"为界定符.
24 done
25 } <$file  # 重定向函数的标准输入.
26
27 file_excerpt $pattern
28
29 # Yes, this entire script could be reduced to
30 #       grep PATTERN /etc/passwd | awk -F":" '{ print $5 }'
31 # or
32 #       awk -F: '/PATTERN/ {print $5}'
33 # or
34 #       awk -F: '($1 == "username") { print $5 }' # real name from username
35 # 但是,这些可能起不到示例的作用.
36
37 exit 0
#################################End Script#########################################
```

还有一个办法,可能是更好理解的重定向函数标准输入方法.它为函数内的一个括号内的
代码块调用标准输入重定向.

```
 1 # 用下面的代替:
 2 Function ()
 3 {
 4  ...
```

```
 5  } < file
 6
 7 # 也试一下这个:
 8 Function ()
 9 {
10   {
11     ...
12    } < file
13 }
14
15 # 同样,
16
17 Function ()  # 可以工作.
18 {
19   {
20    echo $*
21   } | tr a b
22 }
23
24 Function ()  # 这个不会工作
25 {
26   echo $*
27 } | tr a b   # 这儿的内嵌代码块是强制的.
28
29
30 # Thanks, S.C.
```

注意事项:

[1] return命令是Bash内建(builtin)的.

23.2. 局部变量

怎么样使一个变量变成局部的?

局部变量

如果变量用local来声明,那么它只能在该变量声明的代码块(block of code)中可见.

这个代码块就是局部"范围". 在一个函数内,局部变量意味着只能在函数代码块内它才有意义.

Example 23-12 局部变量的可见范围

```
###################################Start Script#########################################
 1 #!/bin/bash
 2 # 在函数内部的全局和局部变量.
 3
 4 func ()
 5 {
 6   local loc_var=23       # 声明为局部变量.
 7   echo                   # 使用内建的'local'关键字.
 8   echo "\"loc_var\" in function = $loc_var"
 9   global_var=999         # 没有声明为局部变量.
10                          # 默认为全局变量.
11   echo "\"global_var\" in function = $global_var"
```

```
12 }
13
14 func
15
16 # 现在,来看看是否局部变量"loc_var"能否在函数外面可见.
17
18 echo
19 echo "\"loc_var\" outside function = $loc_var"
20                              # $loc_var outside function =
21                              # 不, $loc_var不是全局可访问的.
22 echo "\"global_var\" outside function = $global_var"
23                              # $global_var outside function = 999
24                              # $global_var 是全局可访问的.
25 echo
26
27 exit 0
28 #  与In contrast to C相比, 在函数内声明的Bash变量只有在
29 #+ 它被明确声明成局部的变量时才是局部的.
```

################################End Script##

注意: 在函数调用之前,所有在函数内声明且没有明确声明为local的变量都可在函数体外可见.

```
 1 #!/bin/bash
 2
 3 func ()
 4 {
 5 global_var=37    #  在函数还没有被调用前
 6               #+ 变量只在函数内可见.
 7 }              #  函数结束
 8
 9 echo "global_var = $global_var"  # global_var =
10                              #  函数"func"还没有被调用,
11                              #+ 所以变量$global_var还不能被访问.
12
13 func
14 echo "global_var = $global_var"  # global_var = 37
15                              # 已经在函数调用时设置了值.
```

23.2.1. 局部变量使递归变得可能.

局部变量可以递归, [1] 但这个办法会产生大量的计算,因此它在shell脚本中是被明确表明不推荐的. [2]

Example 23-13 用局部变量来递归

################################Start Script##

```
1 #!/bin/bash
2
3 #             阶乘
4 #             ---------
5
6
```

```
 7 # bash允许递归吗?
 8 # 嗯, 允许, 但是...
 9 # 它太慢以致你难以忍受.
10
11
12 MAX_ARG=5
13 E_WRONG_ARGS=65
14 E_RANGE_ERR=66
15
16
17 if [ -z "$1" ]
18 then
19   echo "Usage: `basename $0` number"
20   exit $E_WRONG_ARGS
21 fi
22
23 if [ "$1" -gt $MAX_ARG ]
24 then
25   echo "Out of range (5 is maximum)."
26   #  现在让我们来了解实际情况.
27   #  如果你想求比这个更大的范围的阶乘,
28   #+ 应该重新用一个真正的编程语言来写.
29   exit $E_RANGE_ERR
30 fi
31
32 fact ()
33 {
34   local number=$1
35   #  变量"number"必须声明为局部,
36   #+ 否则它不会工作.
37   if [ "$number" -eq 0 ]
38   then
39     factorial=1    # 0的阶乘为1.
40   else
41     let "decrnum = number - 1"
42     fact $decrnum  # 递归调用(函数内部调用自己本身).
43     let "factorial = $number * $?"
44   fi
45
46   return $factorial
47 }
48
49 fact $1
50 echo "Factorial of $1 is $?."
51
52 exit 0
```

################################End Script###

也请参考例子 A-16的脚本递归的例子. 必须意识到递归也意味着巨大的资源消耗和缓慢的运

行,因此它不适合在脚本中使用.
注意事项:
[1] Herbert Mayer 给递归下的定义是". . . expressing an algorithm by using a simpler version of that same algorithm(用一个相同算法的版本来表示一个算法) . . ." 递归函数是调用它自己本身的函数.
[2] 太多层的递归可能会引起脚本段错误而崩溃.

```
 1 #!/bin/bash
 2
 3 #  警告: 运行这个脚本可能使你的系统失去响应!
 4 #  如果你运气不错,在它使用完所有可用内存之前会段错误而退出.
 5
 6 recursive_function ()
 7 {
 8 echo "$1"     # 使函数做些事情以加速产生段错误.
 9 (( $1 < $2 )) && recursive_function $(( $1 + 1 )) $2;
10 #  当第一个参数比第二个参数少时,
11 #+ 把第1个参数增1再次递归.
12 }
13
14 recursive_function 1 50000  # 递归 50,000 次!
15 #  非常可能段错误 (依赖于栈的大小,它由ulimit -m设置).
16
17 #  这种深度的递归甚至可能由于耗尽栈的内存大小而引起C程序的段错误.
18 #
19
20
21 echo "This will probably not print."
22 exit 0  # 这个脚本将不会从这儿正常退出.
23
24 #  多谢, St`phane Chazelas.
```

23.3. 不使用局部变量的递归

函数甚至可以不使用局部变量来调用自己.
Example 23-14 汉诺塔

```
##################################Start Script#########################################
 1 #! /bin/bash
 2 #
 3 # 汉诺塔(The Towers Of Hanoi)
 4 # Bash script
 5 # Copyright (C) 2000 Amit Singh. All Rights Reserved.
 6 # http://hanoi.kernelthread.com
 7 #
 8 # 在bash version 2.05b.0(13)-release下测试通过
 9 #
10 #  经过作者同意后在"Advanced Bash Scripting Guide"书中使用
11 #
12 #  由ABS的作者做了少许修改.
13
```

```
#==========================================================================#
#  汉诺塔是由Edouard Lucas提出的数学谜题 ,
#+ 他是19世纪的法国数学家.
#
#  有三个直立的柱子竖在地面上.
#  第一个柱子有一组的盘子套在上面.
#  这些盘子是平整的,中间带着孔,
#+ 因此它们才能套在柱子上面.
#  这组盘子有不同的直径,它们是依照直径从小到大来从高到低放置.
#
#  最小的盘在最高,最大的盘在最底部.
#
#  现在的任务是要把这一组的盘子从一个柱子全部地搬到另一个柱子上.
#
#  你只能一次从一个柱子上移动一个盘子到另一个柱子.
#  允许把盘子重新移回到它原来的最初位置.
#  你可以把一个小的盘子放在大的盘子上面,
#+ 但不能把大的盘子放在小的盘子上面.
#  请注意这一点.
#
#  对于这一组盘子,数量少时,只需要移动很少的次数就能达到要求.
#+ 但随着这组盘子的数量的增加,
#+ 移动的次数几乎成倍增长的,
#+ 而移动的策略变得愈加复杂.
#
#  想了解更多的信息, 请访问 http://hanoi.kernelthread.com.
#
#
#        ...                ...                ...
#       | |              | |              | |
#      _|_|_             | |              | |
#     |_____|            | |              | |
#    |_______|           | |              | |
#   |_________|          | |              | |
#  |___________|         | |              | |
#  |           |         | |              | |
# .-------------------------------------------------------------.
# |*************************************************************|
#        #1               #2                #3
#
#==========================================================================#


E_NOPARAM=66  # 没有参数传给脚本.
E_BADPARAM=67 # 传给脚本的盘子数不合法.
Moves=        # 保存移动次数的全局变量.
              # 这儿修改了原脚本.

```

```
62 dohanoi() {   # 递归函数.
63     case $1 in
64     0)
65         ;;
66     *)
67         dohanoi "$(($1-1))" $2 $4 $3
68         echo move $2 "-->" $3
69  let "Moves += 1"  # 这儿修改了原脚本.
70         dohanoi "$(($1-1))" $4 $3 $2
71         ;;
72     esac
73 }
74
75 case $# in
76 1)
77     case $(($1>0)) in     # 至少要有一个盘子.
78     1)
79         dohanoi $1 1 3 2
80         echo "Total moves = $Moves"
81         exit 0;
82         ;;
83     *)
84         echo "$0: illegal value for number of disks";
85         exit $E_BADPARAM;
86         ;;
87     esac
88     ;;
89 *)
90     echo "usage: $0 N"
91     echo "       Where \"N\" is the number of disks."
92     exit $E_NOPARAM;
93     ;;
94 esac
95
96 # 练习:
97 # ---------
98 # 1) 从现在这个位置以下的命令会不会总是被执行?
99 #    为什么? (容易)
100 # 2) 解释这个可运行的"dohanoi"函数的原理.
101 #    (难)
```

##################################End Script###

第24章 别名(Aliases)

==========

Bash别名本质上是一个简称, 缩写, 这可避免键入过长的命令序列. 例如,如果我们添加
alias lm="ls -l | more" 这一行到文件~/.bashrc file里, 然后每次在命令行键入lm 将会
自动被替换成ls -l | more. 这使用户在命令行不必键冗长的命令序列也避免了记忆复杂的命
令及众多选项. 设置alias rm="rm -i" (交互式删除)可以使你犯下错误时不必过度悲伤,它
能避免你不小心删除重要文件.

在脚本里,别名机制不是非常的有用. 如果把别名机制想像成C预处理器的某些功能将会非常好,比如宏扩展,但是,不幸的是Bash不能在别名中扩展参数. [1] 而且,别名不能在 “ 混合型的结构 ” 中使用,比如if/then语句, 循环, 和函数. 还有一个限制是别名不能递归地扩展. 大多数情况Almost invariably, 我们想让别名完成的工作都能被函数更高效地完成.

Example 24-1 脚本中的别名

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # alias.sh
 3
 4 shopt -s expand_aliases
 5 # 必须设置这个选项,否则脚本不会扩展别名功能.
 6
 7
 8 # 首先, 来点有趣的.
 9 alias Jesse_James='echo "\"Alias Jesse James\" was a 1959 comedy starring Bob Hope."'
10 Jesse_James
11
12 echo; echo; echo;
13
14 alias ll="ls -l"
15 # 可以使用单引号(')或双引号(")来定义一个别名.
16
17 echo "Trying aliased \"ll\":"
18 ll /usr/X11R6/bin/mk*   #* 别名工作了.
19
20 echo
21
22 directory=/usr/X11R6/bin/
23 prefix=mk*  # 看通配符会不会引起麻烦.
24 echo "Variables \"directory\" + \"prefix\" = $directory$prefix"
25 echo
26
27 alias lll="ls -l $directory$prefix"
28
29 echo "Trying aliased \"lll\":"
30 lll         # 详细列出在/usr/X11R6/bin目录下所有以mk开头的文件.
31 # 别名能处理连接变量 -- 包括通配符 -- o.k.
32
33
34
35
36 TRUE=1
37
38 echo
39
40 if [ TRUE ]
41 then
42   alias rr="ls -l"
```

```
43   echo "Trying aliased \"rr\" within if/then statement:"
44   rr /usr/X11R6/bin/mk*   #* 引起错误信息!
45   # 别名不能在混合结构中使用.
46   echo "However, previously expanded alias still recognized:"
47   ll /usr/X11R6/bin/mk*
48 fi
49
50 echo
51
52 count=0
53 while [ $count -lt 3 ]
54 do
55   alias rrr="ls -l"
56   echo "Trying aliased \"rrr\" within \"while\" loop:"
57   rrr /usr/X11R6/bin/mk*   #* 在这儿,别名也不会扩展.
58                         #  alias.sh: line 57: rrr: command not found
59   let count+=1
60 done
61
62 echo; echo
63
64 alias xyz='cat $0'   # 脚本打印自身内容.
65                    # 注意是单引号(强引用).
66 xyz
67 #  虽然Bash的文档它是不会工作的,但好像它是可以工作的.
68 #
69 #
70 #  然而,就像 Steve Jacobson指出,
71 #+ 参数"$0"立即扩展成了这个别名的声明.
72
73 exit 0
```

################################End Script###

unalias 命令删除先前设置的别名.

Example 24-2 unalias: 设置和删除别名

################################Start Script###

```
 1 #!/bin/bash
 2 # unalias.sh
 3
 4 shopt -s expand_aliases  # 打开别名功能扩展.
 5
 6 alias llm='ls -al | more'
 7 llm
 8
 9 echo
10
11 unalias llm              # 删除别名.
12 llm
13 # 引起错误信息,因为'llm'已经不再有效了.
```

```
14
15 exit 0
################################End Script#########################################
 bash$ ./unalias.sh
 total 6
drwxrwxr-x    2 bozo     bozo         3072 Feb  6 14:04 .
drwxr-xr-x   40 bozo     bozo         2048 Feb  6 14:04 ..
-rwxr-xr-x    1 bozo     bozo          199 Feb  6 14:04 unalias.sh
./unalias.sh: llm: command not found
```

注意事项:

[1] 但是, 别名好像能扩展位置参数.

第25章 列表结构

==========

"与列表(and list)"和"或列表(or list)" 结构提供一种处理一串连续命令的方法. 它们能有效地替代复杂的嵌套if/then语句甚至可以代替case语句.

连接命令

与列表(and list)

```
   1 command-1 && command-2 && command-3 && ... command-n
```

如果每个命令都返回真值(0)将会依次执行下去. 当某个命令返回假值(非零值), 整个命令链就会结束执行(第一个返回假的命令将会是最后一个执行的命令,后面的都不再执行).

Example 25-1 使用"与列表(and list)"来测试命令行参数

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # "and list"
 3
 4 if [ ! -z "$1" ] && echo "Argument #1 = $1" && [ ! -z "$2" ] && echo "Argument #2 = $2"
 5 then
 6   echo "At least 2 arguments passed to script."
 7   # 所有连接起来的命令都返回真.
 8 else
 9   echo "Less than 2 arguments passed to script."
10   # 整个命令列表中至少有一个命令返回假值.
11 fi
12 # 注意"if [ ! -z $1 ]" 可以工作,但它是有所假定的等价物,
13 #   if [ -n $1 ] 不会工作.
14 #     但是, 加引用可以让它工作.
15 #  if [ -n "$1" ] 就可以了.
16 #     小心!
17 # 最好总是引起要测试的变量.
18
19
20 # 这是使用"纯粹"的 if/then 语句完成的同等功能.
21 if [ ! -z "$1" ]
22 then
23   echo "Argument #1 = $1"
24 fi
25 if [ ! -z "$2" ]
26 then
```

```
27   echo "Argument #2 = $2"
28   echo "At least 2 arguments passed to script."
29 else
30   echo "Less than 2 arguments passed to script."
31 fi
32 # 这会更长且不如"与列表"精致.
33
34
35 exit 0
```

################################End Script###

Example 25-2 用"与列表"的另一个命令行参数测试

################################Start Script###

```
 1 #!/bin/bash
 2
 3 ARGS=1       # 期望的参数个数.
 4 E_BADARGS=65  # 如果用户给出不正确的参数个数的退出码.
 5
 6 test $# -ne $ARGS && echo "Usage: `basename $0` $ARGS argument(s)" && exit $E_BADARGS
 7 #  如果 条件1 测试为真(表示传给脚本的参数不对),
 8 #+ 则余下的命令会被执行,并且脚本结束运行.
 9
10 # 下面的代码只有当上面的测试失败时才会执行.
11 echo "Correct number of arguments passed to this script."
12
13 exit 0
14
15 # 为了检查退出码,脚本结束后用"echo $?"来查看退出码.
```

################################End Script###

当然,一个与列表也能给变量设置默认值.

```
  1 arg1=$@      # 不管怎样,设置变量$arg1为命令行参数.
  2
  3 [ -z "$arg1" ] && arg1=DEFAULT
  4              # 如果没有在命令行上指定参数则把$arg1设置为DEFAULT.
```

或列表(or list)

```
  1 command-1 || command-2 || command-3 || ... command-n
```

只要前一个命令返回假命令链就会依次执行下去. 一旦有一个命令返回真, 命令链就会结束(第一个返回真的命令将会是最后一个执行的命令). 这显然和"与列表"正好相反.

Example 25-3 "或列表"和"与列表"的结合使用

################################Start Script###

```
 1 #!/bin/bash
 2
 3 #  delete.sh, 不是很聪明的文件删除功能.
 4 #  用法: delete filename
 5
 6 E_BADARGS=65
 7
 8 if [ -z "$1" ]
 9 then
```

```
10   echo "Usage: `basename $0` filename"
11   exit $E_BADARGS  # 没有参数? 跳出脚本.
12 else
13   file=$1          # 设置文件名.
14 fi
15
16
17 [ ! -f "$file" ] && echo "File \"$file\" not found. \
18 Cowardly refusing to delete a nonexistent file."
19 # 与列表, 用于文件不存在时给出一个错误信息.
20 # 注意 echo 命令的参数用了一个转义符继续使第二行也是这个命令的参数.
21
22 [ ! -f "$file" ] || (rm -f $file; echo "File \"$file\" deleted.")
23 # 或列表, 用于存在文件时删除此文件.
24
25 # 注意上面两个相反的逻辑.
26 # 与列表为真时才执行, 或列表为假时执行.
27
28 exit 0
##################################End Script##########################################
```

注意: 如果在与列表的第一个命令返回真时,它会执行.

```
 1 # ==> 下面的片断摘自Miquel van Smoorenburg写的 /etc/rc.d/init.d/single 脚本
 2 #+==> 示例与和或列表的使用.
 3 # ==> "箭头"的注释由本书作者添加.
 4
 5 [ -x /usr/bin/clear ] && /usr/bin/clear
 6  # ==> 如果 /usr/bin/clear 存在, 则调用它.
 7  # ==> 在调用一个命令前检查它是否存在,
 8  #+==> 以避免产生错误信息和其他难读懂的结果.
 9
10  # ==> . . .
11
12 # 如果他们想在单用户模式下运行某些程序, 可能也会运行这个...
13 for i in /etc/rc1.d/S[0-9][0-9]* ; do
14       # 检查脚本是否可执行.
15       [ -x "$i" ] || continue
16  # ==> 如果在目录$PWD中相应的文件没有发现,
17  #+==> 则会跳过此次循环.
18
19       # 不接受备份文件和由rpm产生的文件.
20       case "$1" in
21             *.rpmsave|*.rpmorig|*.rpmnew|*~|*.orig)
22                   continue;;
23       esac
24       [ "$i" = "/etc/rc1.d/S00single" ] && continue
25  # ==> 设置脚本名,但还不执行它.
26       $i start
27 done
```

```
 28
 29  # ==> . . .
```

注意: 与列表或是或列表的退出状态是最后一个执行命令的退出状态.

灵活地组合"与"和"或"列表是允许的,但这样逻辑会很容易变得费解并且需要较多的测试.

```
  1 false && true || echo false        # false
  2
  3 # 结果等同
  4 ( false && true ) || echo false    # false
  5 # 但不同与
  6 false && ( true || echo false )    # (没有输出)
  7
  8 #  注意是从左到右来分组并求值的,
  9 #+ 因为逻辑操作符"&&"和"||"有相同的优先处理权.
 10
 11 #  最好避免这种复杂,除非你确实知道你在做什么.
 12
 13 #  Thanks, S.C.
```

参考例子 A-7和例子 7-4 演示的使用与/或列表测试变量的例子.

第26章 数组

==========

较新的Bash版本支持一维数组. 数组元素可以用符号variable[xx]来初始化. 另外,脚本可以用declare -a variable语句来清楚地指定一个数组. 要访问一个数组元素,可以使用花括号来访问,即${variable[xx]}.

Example 26-1 简单的数组用法

```
###################################Start Script#############################################
 1 #!/bin/bash
 2
 3
 4 area[11]=23
 5 area[13]=37
 6 area[51]=UFOs
 7
 8 #  数组成员不必一定要连贯或连续的.
 9
10 #  数组的一部分成员允许不被初始化.
11 #  数组中空缺元素是允许的.
12 #  实际上,保存着稀疏数据的数组(“稀疏数组”)在电子表格处理软件中非常有用.
13 #
14
15
16 echo -n "area[11] = "
17 echo ${area[11]}    #  {大括号}是需要的.
18
19 echo -n "area[13] = "
20 echo ${area[13]}
21
22 echo "Contents of area[51] are ${area[51]}."
23
```

```
24 # 没有初始化内容的数组元素打印空值(NULL值).
25 echo -n "area[43] = "
26 echo ${area[43]}
27 echo "(area[43] unassigned)"
28
29 echo
30
31 # 两个数组元素的和被赋值给另一个数组元素
32 area[5]=`expr ${area[11]} + ${area[13]}`
33 echo "area[5] = area[11] + area[13]"
34 echo -n "area[5] = "
35 echo ${area[5]}
36
37 area[6]=`expr ${area[11]} + ${area[51]}`
38 echo "area[6] = area[11] + area[51]"
39 echo -n "area[6] = "
40 echo ${area[6]}
41 # 这里会失败是因为整数和字符串相加是不允许的.
42
43 echo; echo; echo
44
45 # --------------------------------------------------------------------
46 # 另一个数组, "area2".
47 # 另一种指定数组元素的值的办法...
48 # array_name=( XXX YYY ZZZ ... )
49
50 area2=( zero one two three four )
51
52 echo -n "area2[0] = "
53 echo ${area2[0]}
54 # 啊哈, 从0开始计数(即数组的第一个元素是[0], 而不是 [1]).
55
56 echo -n "area2[1] = "
57 echo ${area2[1]}    # [1] 是数组的第二个元素.
58 # --------------------------------------------------------------------
59
60 echo; echo; echo
61
62 # ----------------------------------------------
63 # 第三种数组, "area3".
64 # 第三种指定数组元素值的办法...
65 # array_name=([xx]=XXX [yy]=YYY ...)
66
67 area3=([17]=seventeen [24]=twenty-four)
68
69 echo -n "area3[17] = "
70 echo ${area3[17]}
71
```

```
72 echo -n "area3[24] = "
73 echo ${area3[24]}
74 # -----------------------------------------------
75
76 exit 0
```

################################End Script###

注意: Bash 允许把变量当成数组来操作,即使这个变量没有明确地被声明为数组.

```
 1 string=abcABC123ABCabc
 2 echo ${string[@]}            # abcABC123ABCabc
 3 echo ${string[*]}            # abcABC123ABCabc
 4 echo ${string[0]}            # abcABC123ABCabc
 5 echo ${string[1]}            # 没有输出!
 6                              # 为什么?
 7 echo ${#string[@]}           # 1
 8                              # 数组中只有一个元素.
 9                              # 且是这个字符串本身.
10
11 # Thank you, Michael Zick, for pointing this out.
```

类似的示范请参考Bash variables are untyped.

Example 26-2 格式化一首诗

################################Start Script###

```
 1 #!/bin/bash
 2 # poem.sh: 排印出作者喜欢的一首诗.
 3
 4 # 诗的行数 (一小节诗).
 5 Line[1]="I do not know which to prefer,"
 6 Line[2]="The beauty of inflections"
 7 Line[3]="Or the beauty of innuendoes,"
 8 Line[4]="The blackbird whistling"
 9 Line[5]="Or just after."
10
11 # 出处.
12 Attrib[1]=" Wallace Stevens"
13 Attrib[2]="\"Thirteen Ways of Looking at a Blackbird\""
14 # 此诗是公众的 (版权期已经到期了).
15
16 echo
17
18 for index in 1 2 3 4 5    # 5行.
19 do
20   printf "     %s\n" "${Line[index]}"
21 done
22
23 for index in 1 2          # 打印两行出处行.
24 do
25   printf "          %s\n" "${Attrib[index]}"
26 done
27
```

```
28 echo
29
30 exit 0
31
32 # 练习:
33 # --------
34 # 修改这个脚本使其从一个文本文件中提取内容打印一首行.
```

################################End Script###

数组元素有它们独有的语法, 并且甚至Bash命令和操作符有特殊的选项可以支持数组使用.

Example 26-3 多种数组操作

################################Start Script###

```
 1 #!/bin/bash
 2 # array-ops.sh: 数组更多有趣的用法.
 3
 4
 5 array=( zero one two three four five )
 6 # 元素    0   1   2    3     4    5
 7
 8 echo ${array[0]}       #  zero
 9 echo ${array:0}        #  zero
10                        #  第一个元素的参数扩展,
11                        #+ 从位置0开始 (即第一个字符).
12 echo ${array:1}        #  ero
13                        #  第一个元素的参数扩展,
14                        #+ 从位置1开始 (即第二个字符).
15
16 echo "--------------"
17
18 echo ${#array[0]}      #  4
19                        #  数组第一个元素的长度.
20 echo ${#array}         #  4
21                        #  数组第一个元素的长度.
22                        #  (另一种写法)
23
24 echo ${#array[1]}      #  3
25                        #  数组第二个元素的长度.
26                        #  Bash的数组是0开始索引的.
27
28 echo ${#array[*]}      #  6
29                        #  数组中元素的个数.
30 echo ${#array[@]}      #  6
31                        #  数组中元素的个数.
32
33 echo "--------------"
34
35 array2=( [0]="first element" [1]="second element" [3]="fourth element" )
36
37 echo ${array2[0]}      # 第一个元素
```

```
38 echo ${array2[1]}      # 第二个元素
39 echo ${array2[2]}      #
40                        # 因为初始化时没有指定,因此值为空(null).
41 echo ${array2[3]}      # 第四个元素
42
43
44 exit 0
```

################################End Script###

大部分标准的字符串操作符 可以用于数组操作.

Example 26-4 用于数组的字符串操作符

################################Start Script###

```
 1 #!/bin/bash
 2 # array-strops.sh: 用于数组的字符串操作符.
 3 # 由Michael Zick编码.
 4 # 已征得作者的同意.
 5
 6 #  一般来说,任何类似 ${name ... } 写法的字符串操作符
 7 #+ 都能在一个数组的所有字符串元素中使用
 8 #+ 像${name[@] ... } 或 ${name[*] ...} 的写法.
 9
10
11 arrayZ=( one two three four five five )
12
13 echo
14
15 # 提取尾部的子串
16 echo ${arrayZ[@]:0}     # one two three four five five
17                         # 所有的元素.
18
19 echo ${arrayZ[@]:1}     # two three four five five
20                         # 在第一个元素 element[0]后面的所有元素.
21
22 echo ${arrayZ[@]:1:2}   # two three
23                         # 只提取在元素 element[0]后面的两个元素.
24
25 echo "-----------------------"
26
27 #  子串删除
28 #  从字符串的前部删除最短的匹配,
29 #+ 匹配字串是一个正则表达式.
30
31 echo ${arrayZ[@]#f*r}   # one two three five five
32                         # 匹配表达式作用于数组所有元素.
33                         # 匹配了"four"并把它删除.
34
35 # 字符串前部最长的匹配
36 echo ${arrayZ[@]##t*e}  # one two four five five
37                         # 匹配表达式作用于数组所有元素.
```

```
38                  # 匹配"three"并把它删除.
39
40 # 字符串尾部的最短匹配
41 echo ${arrayZ[@]%h*e}  # one two t four five five
42                  # 匹配表达式作用于数组所有元素.
43                  # 匹配"hree"并把它删除.
44
45 # 字符串尾部的最长匹配
46 echo ${arrayZ[@]%%t*e}  # one two four five five
47                  # 匹配表达式作用于数组所有元素.
48                  # 匹配"three"并把它删除.
49
50 echo "-----------------------"
51
52 # 子串替换
53
54 # 第一个匹配的子串会被替换
55 echo ${arrayZ[@]/fiv/XYZ}  # one two three four XYZe XYZe
56                      # 匹配表达式作用于数组所有元素.
57
58 # 所有匹配的子串会被替换
59 echo ${arrayZ[@]//iv/YY}   # one two three four fYYe fYYe
60                      # 匹配表达式作用于数组所有元素.
61
62 # 删除所有的匹配子串
63 # 没有指定代替字串意味着删除
64 echo ${arrayZ[@]//fi/}     # one two three four ve ve
65                      # 匹配表达式作用于数组所有元素.
66
67 # 替换最前部出现的字串
68 echo ${arrayZ[@]/#fi/XY}   # one two three four XYve XYve
69                      # 匹配表达式作用于数组所有元素.
70
71 # 替换最后部出现的字串
72 echo ${arrayZ[@]/%ve/ZZ}   # one two three four fiZZ fiZZ
73                      # 匹配表达式作用于数组所有元素.
74
75 echo ${arrayZ[@]/%o/XX}    # one twXX three four five five
76                      # 为什么?
77
78 echo "-----------------------"
79
80
81 # 在从awk(或其他的工具)取得数据之前 --
82 # 记得:
83 #   $( ... ) 是命令替换.
84 #   函数以子进程运行.
85 #   函数将输出打印到标准输出.
```

```
 86 #   用read来读取函数的标准输出.
 87 #   name[@]的写法指定了一个"for-each"的操作.
 88
 89 newstr() {
 90     echo -n "!!!"
 91 }
 92
 93 echo ${arrayZ[@]/%e/$(newstr)}
 94 # on!!! two thre!!! four fiv!!! fiv!!!
 95 # Q.E.D: 替换部分的动作实际上是一个'赋值'.
 96
 97 #  使用"For-Each"型的
 98 echo ${arrayZ[@]//*/$(newstr optional_arguments)}
 99 #  现在Now, 如果if Bash只传递匹配$0的字符串给要调用的函数. . .
100 #
101
102 echo
103
104 exit 0
```

################################End Script###

命令替换能创建数组的新的单个元素.

Example 26-5 将脚本的内容传给数组

################################Start Script###

```
 1 #!/bin/bash
 2 # script-array.sh: 把此脚本的内容传进数组.
 3 # 从Chris Martin的e-mail中得到灵感 (多谢!).
 4
 5 script_contents=( $(cat "$0") )  #  把这个脚本($0)的内容存进数组.
 6                                  #
 7
 8 for element in $(seq 0 $((${#script_contents[@]} - 1)))
 9   do              #  ${#script_contents[@]}
10                   #+ 表示数组中元素的个数.
11                   #
12                   #  问题:
13                   #  为什么需要  seq 0  ?
14                   #  试试更改成 seq 1.
15   echo -n "${script_contents[$element]}"
16                   # 将脚本的每行列成一个域.
17   echo -n " -- "    # 使用" -- "作为域分隔符.
18 done
19
20 echo
21
22 exit 0
23
24 # 练习:
25 # --------
```

```
26 #  修改这个脚本使它能按照它原本的格式输出,
27 #+ 连同空白符,换行,等等.
28 #
```

################################End Script###

在数组的环境里, 一些 Bash 内建的命令 含义有一些轻微的改变. 例如, unset 会删除数组元素, 或甚至删除整个数组.

Example 26-6 一些数组专用的工具

################################Start Script###

```
 1 #!/bin/bash
 2
 3 declare -a colors
 4 #  所有脚本后面的命令都会把
 5 #+ 变量"colors"作为数组对待.
 6
 7 echo "Enter your favorite colors (separated from each other by a space)."
 8
 9 read -a colors    # 键入至少3种颜色以用于下面的示例.
10 #  指定'read'命令的选项,
11 #+ 允许指定数组元素.
12
13 echo
14
15 element_count=${#colors[@]}
16 # 专用语法来提取数组元素的个数.
17 #     element_count=${#colors[*]} 也可以.
18 #
19 #  "@"变量允许分割引号内的单词
20 #+ (依靠空白字符来分隔变量).
21 #
22 #  这就像"$@" 和"$*"在位置参数中表现出来的一样.
23 #
24
25 index=0
26
27 while [ "$index" -lt "$element_count" ]
28 do    # List all the elements in the array.
29   echo ${colors[$index]}
30   let "index = $index + 1"
31 done
32 # 每个数组元素被列为单独的一行.
33 # 如果这个没有要求, 可以用  echo -n "${colors[$index]} "
34 #
35 # 可以用一个"for"循环来做:
36 #   for i in "${colors[@]}"
37 #   do
38 #     echo "$i"
39 #   done
40 # (Thanks, S.C.)
```

```
41
42 echo
43
44 # 再次列出数组中所有的元素, 但使用更优雅的做法.
45   echo ${colors[@]}         # echo ${colors[*]} 也可以.
46
47 echo
48
49 # "unset"命令删除一个数组元素或是整个数组.
50 unset colors[1]             # 删除数组的第二个元素.
51                            # 作用等同于   colors[1]=
52 echo  ${colors[@]}          # 再列出数组,第二个元素没有了.
53
54 unset colors                # 删除整个数组.
55                            #  unset colors[*] 或
56                            #+ unset colors[@] 都可以.
57 echo; echo -n "Colors gone."
58 echo ${colors[@]}           # 再列出数组, 则为空了.
59
60 exit 0
```

##################################End Script###

正如在前面的例子中看到的, ${array_name[@]}和${array_name[*]} 都与数组的所有元素相关. 同样地, 为了计算数组的元素个数, 可以用${#array_name[@]} 或${#array_name[*]}. ${#array_name} 是数组第一个元素${array_name[0]}的长度(字符数) .

Example 26-7 关于空数组和空数组元素

##################################Start Script###

```
 1 #!/bin/bash
 2 # empty-array.sh
 3
 4 #  多谢 Stephane Chazelas 制作这个例子最初的版本,
 5 #+ 并由 Michael Zick 扩展了.
 6
 7
 8 # 空数组不同与含有空值元素的数组.
 9
10 array0=( first second third )
11 array1=( '' )   # "array1" 由一个空元素组成.
12 array2=( )     # 没有元素 . . . "array2" 是空的.
13
14 echo
15 ListArray()
16 {
17 echo
18 echo "Elements in array0:  ${array0[@]}"
19 echo "Elements in array1:  ${array1[@]}"
20 echo "Elements in array2:  ${array2[@]}"
21 echo
22 echo "Length of first element in array0 = ${#array0}"
```

```
23 echo "Length of first element in array1 = ${#array1}"
24 echo "Length of first element in array2 = ${#array2}"
25 echo
26 echo "Number of elements in array0 = ${#array0[*]}"  # 3
27 echo "Number of elements in array1 = ${#array1[*]}"  # 1  (惊奇!)
28 echo "Number of elements in array2 = ${#array2[*]}"  # 0
29 }
30
31 # ==============================================================
32
33 ListArray
34
35 # 尝试扩展这些数组.
36
37 # 增加一个元素到数组.
38 array0=( "${array0[@]}" "new1" )
39 array1=( "${array1[@]}" "new1" )
40 array2=( "${array2[@]}" "new1" )
41
42 ListArray
43
44 # 或
45 array0[${#array0[*]}]="new2"
46 array1[${#array1[*]}]="new2"
47 array2[${#array2[*]}]="new2"
48
49 ListArray
50
51 # 当像上面的做法增加数组时,数组像 '栈'
52 # 上面的做法是 'push(压栈)'
53 # 栈高是:
54 height=${#array2[@]}
55 echo
56 echo "Stack height for array2 = $height"
57
58 # 'pop(出栈)' 是:
59 unset array2[${#array2[@]}-1]   # 数组是以0开始索引的,
60 height=${#array2[@]}           #+ 这就意味着第一个元素下标是 0.
61 echo
62 echo "POP"
63 echo "New stack height for array2 = $height"
64
65 ListArray
66
67 # 只列出数组array0的第二和第三个元素.
68 from=1   #是以0开始的数字
69 to=2  #
70 array3=( ${array0[@]:1:2} )
```

```
71 echo
72 echo "Elements in array3:  ${array3[@]}"
73
74 # 像一个字符串一样处理(字符的数组).
75 # 试试其他的字符串格式.
76
77 # 替换:
78 array4=( ${array0[@]/second/2nd} )
79 echo
80 echo "Elements in array4:  ${array4[@]}"
81
82 # 替换所有匹配通配符的字符串.
83 array5=( ${array0[@]//new?/old} )
84 echo
85 echo "Elements in array5:  ${array5[@]}"
86
87 # 当你开始觉得对此有把握的时候 . . .
88 array6=( ${array0[@]#*new} )
89 echo # 这个可能会使你感到惊奇.
90 echo "Elements in array6:  ${array6[@]}"
91
92 array7=( ${array0[@]#new1} )
93 echo # 数组array6之后就没有惊奇了.
94 echo "Elements in array7:  ${array7[@]}"
95
96 # 这看起来非常像 . . .
97 array8=( ${array0[@]/new1/} )
98 echo
99 echo "Elements in array8:  ${array8[@]}"
100
101 #  那么我们怎么总结它呢So what can one say about this?
102
103 #  字符串操作在数组var[@]的每一个元素中执行.
104 #
105 #  因此Therefore : 如果结果是一个零长度的字符串,
106 #+ Bash支持字符串向量操作,
107 #+ 元素会在结果赋值中消失不见.
108
109 #  提问, 这些字符串是强还是弱引用?
110
111 zap='new*'
112 array9=( ${array0[@]/$zap/} )
113 echo
114 echo "Elements in array9:  ${array9[@]}"
115
116 # 当你还在想你在Kansas州的何处时 . . .
117 array10=( ${array0[@]#$zap} )
118 echo
```

```
119 echo "Elements in array10:  ${array10[@]}"
120
121 # 把 array7 和 array10比较.
122 # 把 array8 和 array9比较.
123
124 # 答案: 必须用弱引用.
125
126 exit 0
```

##################################End Script##

${array_name[@]}和${array_name[*]} 的关系类似于$@ and $*. 这种数组用法非常有用.

```
  1 # 复制一个数组.
  2 array2=( "${array1[@]}" )
  3 # 或
  4 array2="${array1[@]}"
  5
  6 # 给数组增加一个元素.
  7 array=( "${array[@]}" "new element" )
  8 # 或
  9 array[${#array[*]}]="new element"
 10
 11 # Thanks, S.C.
```

注意: array=(element1 element2 ... elementN) 初始化操作, 依赖于命令替换 (command substitution)使将一个文本内容加载进数组成为可能.

```
  1 #!/bin/bash
  2
  3 filename=sample_file
  4
  5 #           cat sample_file
  6 #
  7 #           1 a b c
  8 #           2 d e fg
  9
 10
 11 declare -a array1
 12
 13 array1=( `cat "$filename"`)              #  加载$filename文件的内容进数组array1.
 14 #        打印文件到标准输出              #
 15 #
 16 #  array1=( `cat "$filename" | tr '\n' ' '`)
 17 #                        把文件里的换行变为空格.
 18 #  这是没必要的,因为Bash做单词分割时会把换行变为空格.
 19 #
 20
 21 echo ${array1[@]}          # 打印数组.
 22 #                          1 a b c 2 d e fg
 23 #
 24 #  文件中每个由空白符分隔开的“ 词 ”都被存在数组的一个元素里
 25 #
```

```
 26
 27 element_count=${#array1[*]}
 28 echo $element_count          # 8
```

出色的技巧使数组的操作技术又多了一种.

Example 26-8 初始化数组

```
################################Start Script#########################################
 1 #! /bin/bash
 2 # array-assign.bash
 3
 4 #  数组操作是Bash特有的,
 5 #+ 因此脚本名用".bash"结尾.
 6
 7 # Copyright (c) Michael S. Zick, 2003, All rights reserved.
 8 # 许可证: 没有任何限制,可以用于任何目的的反复使用.
 9 # Version: $ID$
10 #
11 # 由William Park添加注释.
12
13 #  基于Stephane Chazelas提供在本书中的一个例子
14 #
15
16 # 'times' 命令的输出格式:
17 # User CPU <空格> System CPU
18 # User CPU of dead children <空格> System CPU of dead children
19
20 #  Bash赋一个数组的所有元素给新的数组变量有两种办法.
21 #
22 #  在Bash版本2.04, 2.05a 和 2.05b,
23 #+ 这两种办法都对NULL的值的元素全部丢弃.
24 #  另一种数组赋值办法是维护[下标]=值之间的关系将会在新版本的Bash支持.
25 #
26
27 #  可以用外部命令来构造一个大数组,
28 #+ 但几千个元素的数组如下就可以构造了.
29 #
30
31 declare -a bigOne=( /dev/* )
32 echo
33 echo 'Conditions: Unquoted, default IFS, All-Elements-Of'
34 echo "Number of elements in array is ${#bigOne[@]}"
35
36 # set -vx
37
38
39
40 echo
41 echo '- - testing: =( ${array[@]} ) - -'
42 times
```

```
43 declare -a bigTwo=( ${bigOne[@]} )
44 #              ^           ^
45 times
46
47 echo
48 echo '- - testing: =${array[@]} - -'
49 times
50 declare -a bigThree=${bigOne[@]}
51 # 这次没有用括号.
52 times
53
54 #  正如Stephane Chazelas指出的那样比较输出的数组可以了解第二种格式的赋值比第三和第四的times的更快
55 #
56 #
57 #  William Park 解释explains:
58 #+ bigTwo 数组是被赋值了一个单字符串,
59 #+ bigThree 则赋值时一个一个元素的赋值.
60 #  所以, 实际上的情况是:
61 #                bigTwo=( [0]="... ... ..." )
62 #                bigThree=( [0]="..." [1]="..." [2]="..." ... )
63
64
65 #  我在本书的例子中仍然会继续用第一种格式,
66 #+ 因为我认为这会对说明清楚更有帮助.
67
68 #  我的例子中的可复用的部分实际上还是会使用第二种格式,
69 #+ 因为这种格式更快一些.
70
71 # MSZ: 很抱歉早先的失误(应是指本书的先前版本).
72
73
74 #  注:
75 #  ----
76 #  在31和43行的"declare -a"语句不是必须的,
77 #+ 因为会在使用Array=( ... )赋值格式时暗示它是数组.
78 #
79 #  但是, 省略这些声明会导致后面脚本的相关操作更慢一些.
80 #
81 #  试一下, 看有什么变化.
82
83 exit 0
```

################################End Script###

注意: 对变量增加 declare -a 语句声明可以加速后面的数组操作速度.

Example 26-9 复制和连接数组

################################Start Script###

```
 1 #! /bin/bash
 2 # CopyArray.sh
 3 #
```

```
4 # 由 Michael Zick编写.
5 # 在本书中使用已得到许可.
6
7 #  怎么传递变量名和值处理,返回就用使用该变量,
8 #+ 或说"创建你自己的赋值语句".
9
10
11 CpArray_Mac() {
12
13 # 创建赋值命令语句
14
15     echo -n 'eval '
16     echo -n "$2"                 # 目的变量名
17     echo -n '=( ${'
18     echo -n "$1"                 # 源名字
19     echo -n '[@]} )'
20
21 # 上面的全部会合成单个命令.
22 # 这就是函数所有的功能.
23 }
24
25 declare -f CopyArray              # 函数"指针"
26 CopyArray=CpArray_Mac             # 建立命令
27
28 Hype()
29 {
30
31 # 要复制的数组名为 $1.
32 # (接合数组,并包含尾部的字符串"Really Rocks".)
33 # 返回结果的数组名为 $2.
34
35     local -a TMP
36     local -a hype=( Really Rocks )
37
38     $($CopyArray $1 TMP)
39     TMP=( ${TMP[@]} ${hype[@]} )
40     $($CopyArray TMP $2)
41 }
42
43 declare -a before=( Advanced Bash Scripting )
44 declare -a after
45
46 echo "Array Before = ${before[@]}"
47
48 Hype before after
49
50 echo "Array After = ${after[@]}"
51
```

```
52 # 有多余的字符串?
53
54 echo "What ${after[@]:3:2}?"
55
56 declare -a modest=( ${after[@]:2:1} ${after[@]:3:2} )
57 #                   ----     子串提取      ----
58
59 echo "Array Modest = ${modest[@]}"
60
61 # 'before'变量变成什么了？
62
63 echo "Array Before = ${before[@]}"
64
65 exit 0
```

################################End Script###

Example 26-10 关于连接数组的更多信息

################################Start Script###

```
 1 #! /bin/bash
 2 # array-append.bash
 3
 4 # Copyright (c) Michael S. Zick, 2003, All rights reserved.
 5 # 许可: 可以无限制的以任何目的任何格式重复使用.
 6 # 版本: $ID$
 7 #
 8 # 格式上由M.C做了轻微的修改.
 9
10
11 # 数组操作是Bash特有的属性.
12 # 原来的 UNIX /bin/sh 没有类似的功能.
13
14
15 #  把此脚本的输出管道输送给 'more'
16 #+ 以便输出不会滚过终端屏幕.
17
18
19 # 下标依次使用.
20 declare -a array1=( zero1 one1 two1 )
21 # 下标有未使用的 ([1] 没有被定义).
22 declare -a array2=( [0]=zero2 [2]=two2 [3]=three2 )
23
24 echo
25 echo '- Confirm that the array is really subscript sparse. -'
26 echo "Number of elements: 4"       # 这儿是举例子就用硬编码.
27 for (( i = 0 ; i < 4 ; i++ ))
28 do
29     echo "Element [$i]: ${array2[$i]}"
30 done
31 # 也可以参考basics-reviewed.bash更多的常见代码.
```

```
32
33
34 declare -a dest
35
36 # 组合 (添加) 两个数组到第三个数组.
37 echo
38 echo 'Conditions: Unquoted, default IFS, All-Elements-Of operator'
39 echo '- Undefined elements not present, subscripts not maintained. -'
40 # # 那些未定义的元素不存在; 组合时会丢弃这些元素.
41
42 dest=( ${array1[@]} ${array2[@]} )
43 # dest=${array1[@]}${array2[@]}     # 奇怪的结果, 或者叫臭虫.
44
45 # 现在, 打印出结果.
46 echo
47 echo '- - Testing Array Append - -'
48 cnt=${#dest[@]}
49
50 echo "Number of elements: $cnt"
51 for (( i = 0 ; i < cnt ; i++ ))
52 do
53     echo "Element [$i]: ${dest[$i]}"
54 done
55
56 # 把一个数组赋值给另一个数组的单个元素 (两次).
57 dest[0]=${array1[@]}
58 dest[1]=${array2[@]}
59
60 # 列出结果.
61 echo
62 echo '- - Testing modified array - -'
63 cnt=${#dest[@]}
64
65 echo "Number of elements: $cnt"
66 for (( i = 0 ; i < cnt ; i++ ))
67 do
68     echo "Element [$i]: ${dest[$i]}"
69 done
70
71 # 检测第二个元素的改变.
72 echo
73 echo '- - Reassign and list second element - -'
74
75 declare -a subArray=${dest[1]}
76 cnt=${#subArray[@]}
77
78 echo "Number of elements: $cnt"
79 for (( i = 0 ; i < cnt ; i++ ))
```

```
 80 do
 81     echo "Element [$i]: ${subArray[$i]}"
 82 done
 83
 84 #  用 '=${ ... }' 把整个数组的值赋给另一个数组的单个元素
 85 #+ 使数组所有元素值被转换成了一个字符串,各元素的值由一个空格分开(其实是IFS的第一个字符).
 86 #
 87 #
 88
 89 # 如果原先的元素没有包含空白符 . . .
 90 # 如果原先的数组下标都是连续的 . . .
 91 # 我们就能取回最初的数组结构.
 92
 93 # 恢复第二个元素的修改回元素.
 94 echo
 95 echo '- - Listing restored element - -'
 96
 97 declare -a subArray=( ${dest[1]} )
 98 cnt=${#subArray[@]}
 99
100 echo "Number of elements: $cnt"
101 for (( i = 0 ; i < cnt ; i++ ))
102 do
103     echo "Element [$i]: ${subArray[$i]}"
104 done
105 echo '- - Do not depend on this behavior. - -'
106 echo '- - This behavior is subject to change - -'
107 echo '- - in versions of Bash newer than version 2.05b - -'
108
109 # MSZ: 很抱歉早先时混淆的几个要点(译者注：应该是指本书早先的版本).
110
111 exit 0
```

################################End Script###

--

数组允许在脚本中实现一些常见的熟悉算法.这是否是必要的好想法在此不讨论,留给读者自行判断.

Example 26-11 一位老朋友: 冒泡排序

################################Start Script###

```
 1 #!/bin/bash
 2 # bubble.sh: 排序法之冒泡排序.
 3
 4 # 回忆冒泡排序法. 在这个版本中要实现它...
 5
 6 #  靠连续地多次比较数组元素来排序,
 7 #+ 比较两个相邻的元素,如果排序顺序不对,则交换两者的顺序.
 8 #  当第一轮比较结束后,最"重"的元素就被排到了最底部.
 9 #  当第二轮比较结束后,第二"重"的元素就被排到了次底部的位置.
10 #  以此类推.
```

```
11 #  这意味着每轮的比较不需要比较先前已"沉淀"好的数据.
12 #  因此你会注意到后面数据的打印会比较快一些.
13
14
15 exchange()
16 {
17   # 交换数组的两个元素.
18   local temp=${Countries[$1]} #  临时保存要交换的一个元素.
19                               #
20   Countries[$1]=${Countries[$2]}
21   Countries[$2]=$temp
22
23   return
24 }
25
26 declare -a Countries  #  声明数组,
27                       #+ 在此是可选的,因为下面它会被按数组来初始化.
28
29 #  是否允许用转义符(\)将数组的各变量值放到几行上?
30 #
31 #  是的.
32
33 Countries=(Netherlands Ukraine Zaire Turkey Russia Yemen Syria \
34 Brazil Argentina Nicaragua Japan Mexico Venezuela Greece England \
35 Israel Peru Canada Oman Denmark Wales France Kenya \
36 Xanadu Qatar Liechtenstein Hungary)
37
38 # "Xanadu" 是个虚拟的充满美好的神话之地.
39 #
40
41
42 clear                 # 开始之前清除屏幕.
43
44 echo "0: ${Countries[*]}"  # 从0索引的元素开始列出整个数组.
45
46 number_of_elements=${#Countries[@]}
47 let "comparisons = $number_of_elements - 1"
48
49 count=1 # 传递数字.
50
51 while [ "$comparisons" -gt 0 ]         # 开始外部的循环
52 do
53
54   index=0  # 每轮开始前重设索引值为0.
55
56   while [ "$index" -lt "$comparisons" ] # 开始内部循环
57   do
58     if [ ${Countries[$index]} \> ${Countries[`expr $index + 1`]} ]
```

```
59    # 如果原来的排序次序不对...
60    # 回想一下 \> 在单方括号里是is ASCII 码的比较操作符.
61    #
62
63    # if [[ ${Countries[$index]} > ${Countries[`expr $index + 1`]} ]]
64    #+ 也可以.
65    then
66      exchange $index `expr $index + 1`  # 交换.
67    fi
68    let "index += 1"
69   done # 内部循环结束
70
71 # ----------------------------------------------------------------------
72 # Paulo Marcel Coelho Aragao 建议使用更简单的for-loops.
73 #
74 # for (( last = $number_of_elements - 1 ; last > 1 ; last-- ))
75 # do
76 #     for (( i = 0 ; i < last ; i++ ))
77 #     do
78 #         [[ "${Countries[$i]}" > "${Countries[$((i+1))]}" ]] \
79 #             && exchange $i $((i+1))
80 #     done
81 # done
82 # ----------------------------------------------------------------------
83
84
85 let "comparisons -= 1" #  因为最"重"的元素冒到了最底部,
86                        #+ 我们可以每轮少做一些比较.
87
88 echo
89 echo "$count: ${Countries[@]}"  # 每轮结束后,打印一次数组.
90 echo
91 let "count += 1"                # 增加传递计数.
92
93 done                            # 外部循环结束
94                                 # 完成.
95
96 exit 0
################################End Script#############################################
--
在数组内嵌一个数组有可能做到吗?
  1 #!/bin/bash
  2 # "内嵌" 数组.
  3
  4 #  Michael Zick 提供这个例子,
  5 #+ 由William Park作了些纠正和解释.
  6
  7 AnArray=( $(ls --inode --ignore-backups --almost-all \
```

```
  8  --directory --full-time --color=none --time=status \
  9  --sort=time -l ${PWD} ) )  # 命令及选项.
 10
 11 # 空格是有意义的 . . . 不要在上面引号引用任何东西.
 12
 13 SubArray=( ${AnArray[@]:11:1}  ${AnArray[@]:6:5} )
 14 #  这个数组有6个元素:
 15 #+     SubArray=( [0]=${AnArray[11]} [1]=${AnArray[6]} [2]=${AnArray[7]}
 16 #      [3]=${AnArray[8]} [4]=${AnArray[9]} [5]=${AnArray[10]} )
 17 #
 18 #  Bash中的数组像是字符串(char *)型的(循环)链表.
 19 #
 20 #  因此, 这实际上不是内嵌的数组,
 21 #+ 但它的功能是相似的.
 22
 23 echo "Current directory and date of last status change:"
 24 echo "${SubArray[@]}"
 25
 26 exit 0
```

--

内嵌数组和间接引用(indirect references) 的组合使用产生了一些有趣的用法.

Example 26-12 内嵌数组和间接引用

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # embedded-arrays.sh
 3 # 内嵌数组和间接引用.
 4
 5 # 由Dennis Leeuw编写.
 6 # 已获使用许可.
 7 # 由本文作者修改.
 8
 9
10 ARRAY1=(
11         VAR1_1=value11
12         VAR1_2=value12
13         VAR1_3=value13
14 )
15
16 ARRAY2=(
17         VARIABLE="test"
18         STRING="VAR1=value1 VAR2=value2 VAR3=value3"
19         ARRAY21=${ARRAY1[*]}
20 )      # 把ARRAY1数组嵌到这个数组里.
21
22 function print () {
23         OLD_IFS="$IFS"
24         IFS=$'\n'      #  这是为了在每个行打印一个数组元素.
25                        #
```

```
26      TEST1="ARRAY2[*]"
27      local ${!TEST1} # 试下删除这行会发生什么.
28      # 间接引用.
29 # 这使 $TEST1只在函数内存取.
30 #
31
32
33      # 我们看看还能干点什么.
34      echo
35      echo "\$TEST1 = $TEST1"      # 变量的名称.
36      echo; echo
37      echo "{\$TEST1} = ${!TEST1}" # 变量的内容.
38                              # 这就是间接引用的作用.
39                              #
40      echo
41      echo "---------------------------------------------"; echo
42      echo
43
44
45      # 打印变量
46      echo "Variable VARIABLE: $VARIABLE"
47
48      # 打印一个字符串元素
49      IFS="$OLD_IFS"
50      TEST2="STRING[*]"
51      local ${!TEST2}     # 间接引用 (像上面一样).
52      echo "String element VAR2: $VAR2 from STRING"
53
54      # 打印一个字符串元素
55      TEST2="ARRAY21[*]"
56      local ${!TEST2}     # 间接引用 (像上面一样).
57      echo "Array element VAR1_1: $VAR1_1 from ARRAY21"
58 }
59
60 print
61 echo
62
63 exit 0
64
65 #   脚本作者注,
66 #+ "你可以很容易地将其扩展成Bash的一个能创建hash的脚本."
67 #   (难) 留给读者的练习: 实现它.
##################################End Script#############################################
--
数组使埃拉托色尼素数筛子有了shell脚本的实现. 当然, 如果是追求效率的应用自然应该用
一种编译型的语言,例如用C. 这种脚本运行实在是太慢.
Example 26-13 复杂数组应用: 埃拉托色尼素数筛子
##################################Start Script###########################################
```

SHELL十三问

```
1 #!/bin/bash
2 # sieve.sh (ex68.sh)
3
4 # 埃拉托色尼素数筛子
5 # 找素数的经典算法.
6
7 #  在同等数量的数值内这个脚本比用C写的版本慢很多.
8 #
9
10 LOWER_LIMIT=1       # 从1开始.
11 UPPER_LIMIT=1000    # 到 1000.
12 # (如果你很有时间的话,你可以把它设得更高 . . . )
13
14 PRIME=1
15 NON_PRIME=0
16
17 let SPLIT=UPPER_LIMIT/2
18 # 优化:
19 # 只需要测试中间到最大之间的值 (为什么?).
20
21
22 declare -a Primes
23 # Primes[] 是一个数组.
24
25
26 initialize ()
27 {
28 # 初始化数组.
29
30 i=$LOWER_LIMIT
31 until [ "$i" -gt "$UPPER_LIMIT" ]
32 do
33   Primes[i]=$PRIME
34   let "i += 1"
35 done
36 #  假定所有的数组成员都是需要检查的 (素数)
37 #+ 一直到检查完成前.
38 }
39
40 print_primes ()
41 {
42 # 打印出所有Primes[]数组中被标记为素数的元素.
43
44 i=$LOWER_LIMIT
45
46 until [ "$i" -gt "$UPPER_LIMIT" ]
47 do
48
```

```
49   if [ "${Primes[i]}" -eq "$PRIME" ]
50   then
51     printf "%8d" $i
52     # 每个数字打印前先打印8个空格, 数字是在偶数列打印的.
53   fi
54
55   let "i += 1"
56
57 done
58
59 }
60
61 sift () # 查出非素数.
62 {
63
64 let i=$LOWER_LIMIT+1
65 # 我们都知道1是素数, 所以我们从2开始.
66
67 until [ "$i" -gt "$UPPER_LIMIT" ]
68 do
69
70 if [ "${Primes[i]}" -eq "$PRIME" ]
71 # 不要处理已经过滤过的数字 (被标识为非素数).
72 then
73
74   t=$i
75
76   while [ "$t" -le "$UPPER_LIMIT" ]
77   do
78     let "t += $i "
79     Primes[t]=$NON_PRIME
80     # 标识为非素数.
81   done
82
83 fi
84
85   let "i += 1"
86 done
87
88
89 }
90
91
92 # ============================================
93 # main ()
94 # 继续调用函数.
95 initialize
96 sift
```

```
 97 print_primes
 98 # 这就是被称为结构化编程的东西了.
 99 # ==============================================
100
101 echo
102
103 exit 0
104
105
106
107 # ------------------------------------------------------- #
108 # 因为前面的一个'exit',所以下面的代码不会被执行.
109
110 #  下面是Stephane Chazelas写的一个埃拉托色尼素数筛子的改进版本,
111 #+ 运行会稍微快一点.
112
113 # 必须在命令行上指定参数(寻找素数的限制范围).
114
115 UPPER_LIMIT=$1                # 值来自命令行.
116 let SPLIT=UPPER_LIMIT/2       # 从中间值到最大值.
117
118 Primes=( '' $(seq $UPPER_LIMIT) )
119
120 i=1
121 until (( ( i += 1 ) > SPLIT ))  # 仅需要从中间值检查.
122 do
123   if [[ -n $Primes[i] ]]
124   then
125     t=$i
126     until (( ( t += i ) > UPPER_LIMIT ))
127     do
128       Primes[t]=
129     done
130   fi
131 done
132 echo ${Primes[*]}
133
134 exit 0
```

################################End Script###

比较这个用数组的素数产生器和另一种不用数组的例子 A-16.

--

数组可以做一定程度的扩展,以模拟支持Bash原本不支持的数据结构.

Example 26-14 模拟下推的堆栈

################################Start Script###

```
 1 #!/bin/bash
 2 # stack.sh: 下推的堆栈模拟
 3
 4 #  类似于CPU栈, 下推的堆栈依次保存数据项,
```

```
5 #+ 但取出时则反序进行, 后进先出.
6
7 BP=100          #  栈数组的基点指针.
8                 #  从元素100开始.
9
10 SP=$BP          #  栈指针.
11                #  初始化栈底.
12
13 Data=           #  当前栈的内容.
14                #  必须定义成全局变量,
15                #+ 因为函数的返回整数有范围限制.
16
17 declare -a stack
18
19
20 push()          # 把一个数据项压入栈.
21 {
22 if [ -z "$1" ]    # 没有可压入的?
23 then
24   return
25 fi
26
27 let "SP -= 1"     # 更新堆栈指针.
28 stack[$SP]=$1
29
30 return
31 }
32
33 pop()                # 从栈中弹出一个数据项.
34 {
35 Data=                # 清空保存数据项中间变量.
36
37 if [ "$SP" -eq "$BP" ]   # 已经没有数据可弹出?
38 then
39   return
40 fi                   #  这使SP不会超过100,
41                      #+ 例如, 这可保护一个失控的堆栈.
42
43 Data=${stack[$SP]}
44 let "SP += 1"          # 更新堆栈指针.
45 return
46 }
47
48 status_report()        # 打印堆栈的当前状态.
49 {
50 echo "-----------------------------------"
51 echo "REPORT"
52 echo "Stack Pointer = $SP"
```

```
53 echo "Just popped \""$Data"\" off the stack."
54 echo "-----------------------------------"
55 echo
56 }
57
58
59 # =============================================================
60 # 现在,来点乐子.
61
62 echo
63
64 # 看你是否能从空栈里弹出数据项来.
65 pop
66 status_report
67
68 echo
69
70 push garbage
71 pop
72 status_report     # 压入garbage, 弹出garbage.
73
74 value1=23; push $value1
75 value2=skidoo; push $value2
76 value3=FINAL; push $value3
77
78 pop               # FINAL
79 status_report
80 pop               # skidoo
81 status_report
82 pop               # 23
83 status_report    # 后进, 先出!
84
85 #  注意堆栈指针每次压栈时减,
86 #+ 每次弹出时加一.
87
88 echo
89
90 exit 0
91
92 # =============================================================
93
94
95 # 练习:
96 # ---------
97
98 # 1)  修改"push()"函数,使其调用一次就能够压入多个数据项.
99 #
100
```

```
101 # 2)  修改"pop()"函数,使其调用一次就能弹出多个数据项.
102 #
103
104 # 3)  给那些有临界操作的函数增加出错检查.
105 #     即是指是否一次完成操作或没有完成操作返回相应的代码,
106 #   + 没有完成要启动合适的处理动作.
107 #
108
109 # 4)  这个脚本为基础,
110 #   + 写一个栈实现的四则运算计算器.
################################End Script#########################################
--
```

要想操作数组的下标需要中间变量. 如果确实要这么做, 可以考虑使用一种更强功能的编程语言, 例如 Perl 或 C.

Example 26-15 复杂的数组应用: 列出一种怪异的数学序列

```
################################Start Script#########################################
 1 #!/bin/bash
 2
 3 # Douglas Hofstadter的有名的"Q-series":
 4
 5 # Q(1) = Q(2) = 1
 6 # Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2)), 当 n>2 时
 7
 8 # 这是令人感到陌生的也是没有规律的"乱序"整数序列.
 9 # 序列的头20个如下所示:
10 # 1 1 2 3 3 4 5 5 6 6 6 8 8 8 10 9 10 11 11 12
11
12 #  参考Hofstadter的书, "Goedel, Escher, Bach: An Eternal Golden Braid",
13 #+ 页码 137.
14
15
16 LIMIT=100     # 计算数的个数.
17 LINEWIDTH=20  # 很行要打印的数的个数.
18
19 Q[1]=1       # 序列的头2个是 1.
20 Q[2]=1
21
22 echo
23 echo "Q-series [$LIMIT terms]:"
24 echo -n "${Q[1]} "             # 打印头2个数.
25 echo -n "${Q[2]} "
26
27 for ((n=3; n <= $LIMIT; n++))  # C风格的循环条件.
28 do   # Q[n] = Q[n - Q[n-1]] + Q[n - Q[n-2]]  当 n>2 时
29 #  需要将表达式分步计算,
30 #+ 因为Bash不擅长处理此类复杂计算.
31
32   let "n1 = $n - 1"       # n-1
```

```
33   let "n2 = $n - 2"       # n-2
34
35   t0=`expr $n - ${Q[n1]}`  # n - Q[n-1]
36   t1=`expr $n - ${Q[n2]}`  # n - Q[n-2]
37
38   T0=${Q[t0]}              # Q[n - Q[n-1]]
39   T1=${Q[t1]}              # Q[n - Q[n-2]]
40
41 Q[n]=`expr $T0 + $T1`     # Q[n - Q[n-1]] + Q[n - Q[n-2]]
42 echo -n "${Q[n]} "
43
44 if [ `expr $n % $LINEWIDTH` -eq 0 ]   # 格式化输出.
45 then   #     ^ 取模操作
46   echo # 把行分成内部的块.
47 fi
48
49 done
50
51 echo
52
53 exit 0
54
55 # 这是Q-series问题的迭代实现.
56 # 更直接明了的递归实现留给读者完成.
57 # 警告: 递归地计算这个序列会花很长的时间.
```

################################End Script###

--

Bash 只支持一维数组,但有一些技巧可用来模拟多维数组.

Example 26-16 模拟二维数组,并使它倾斜

################################Start Script###

```
 1 #!/bin/bash
 2 # twodim.sh: 模拟二维数组.
 3
 4 # 一维数组由单行组成.
 5 # 二维数组由连续的行组成.
 6
 7 Rows=5
 8 Columns=5
 9 # 5 X 5 的数组Array.
10
11 declare -a alpha     # char alpha [Rows] [Columns];
12                      # 不必要的声明. 为什么?
13
14 load_alpha ()
15 {
16 local rc=0
17 local index
18
```

```
19 for i in A B C D E F G H I J K L M N O P Q R S T U V W X Y
20 do     # 如果你高兴,可以使用不同的符号.
21   local row=`expr $rc / $Columns`
22   local column=`expr $rc % $Rows`
23   let "index = $row * $Rows + $column"
24   alpha[$index]=$i
25 # alpha[$row][$column]
26   let "rc += 1"
27 done
28
29 #  更简单的办法
30 #+   declare -a alpha=( A B C D E F G H I J K L M N O P Q R S T U V W X Y )
31 #+ 但这就缺少了二维数组的感觉了.
32 }
33
34 print_alpha ()
35 {
36 local row=0
37 local index
38
39 echo
40
41 while [ "$row" -lt "$Rows" ]   #  以行顺序为索引打印行的各元素:
42 do                              #+ 即数组列值变化快,
43                                 #+ 行值变化慢.
44   local column=0
45
46   echo -n "       "            #  依行倾斜打印正方形的数组.
47
48   while [ "$column" -lt "$Columns" ]
49   do
50     let "index = $row * $Rows + $column"
51     echo -n "${alpha[index]} "  # alpha[$row][$column]
52     let "column += 1"
53   done
54
55   let "row += 1"
56   echo
57
58 done
59
60 # 等同于
61 #     echo ${alpha[*]} | xargs -n $Columns
62
63 echo
64 }
65
66 filter ()     # 过滤出负数的数组索引.
```

```
67 {
68
69 echo -n "  "  # 产生倾斜角度.
70              # 解释怎么办到的.
71
72 if [[ "$1" -ge 0 &&  "$1" -lt "$Rows" && "$2" -ge 0 && "$2" -lt "$Columns" ]]
73 then
74    let "index = $1 * $Rows + $2"
75    # Now, print it rotated现在,打印旋转角度.
76    echo -n " ${alpha[index]}"
77    #           alpha[$row][$column]
78 fi
79
80 }
81
82
83
84
85 rotate ()  #  旋转数组 45 度 --
86 {         #+ 在左下角"平衡"图形.
87 local row
88 local column
89
90 for (( row = Rows; row > -Rows; row-- ))
91   do      # 从后面步进数组. 为什么?
92
93   for (( column = 0; column < Columns; column++ ))
94   do
95
96    if [ "$row" -ge 0 ]
97    then
98      let "t1 = $column - $row"
99      let "t2 = $column"
100    else
101      let "t1 = $column"
102      let "t2 = $column + $row"
103    fi
104
105    filter $t1 $t2   # 过滤出负数数组索引.
106                     # 如果你不这样做会怎么样?
107   done
108
109   echo; echo
110
111 done
112
113 #  数组旋转灵感源于Herbert Mayer写的
114 #+ "Advanced C Programming on the IBM PC," 的例子 (页码. 143-146)
```

```
115 #+ (看参考书目附录).
116 #  这也能看出C能做的事情有多少能用shell脚本做到.
117 #
118
119 }
120
121
122 #---------------   现在, 可以开始了.    ------------#
123 load_alpha     # 加载数组.
124 print_alpha    # 打印数组.
125 rotate         # 反时钟旋转数组45度.
126 #-----------------------------------------------------#
127
128 exit 0
129
130 # 这是有点做作,不太优雅.
131
132 # 练习:
133 # ---------
134 # 1)  重写数组加载和打印函数,
135 #     使其更直观和容易了解.
136 #
137 # 2)  指出数组旋转函数是什么原理.
138 #     Hint索引: 思考数组从尾向前索引的实现.
139 #
140 # 3)  重写脚本使其可以处理非方形数组Rewrite this script to handle a non-square array,
141 #     例如 6 X 4 的数组.
142 #     尝试旋转数组时做到最小"失真".
```

##################################End Script##

二维数组本质上等同于一维数组, 而只增加了使用行和列的位置来引用和操作元素的寻址模式.

关于二维数组更好的例子, 请参考例子 A-10.

--

另一个有趣的使用数组的脚本:

* 例子 14-3

[]

高级Bash脚本编程指南(五)（下）

文章整理: 文章来源: 网络

第27章 /dev 和 /proc

======================

Linux 或 UNIX 机器都带有/dev和/proc目录用于特殊目的.

27.1. /dev

在 /dev 目录内包含以或不以硬件形式出现的物理设备条目. [1] 包含被挂载的文件系统的硬设备分区在/dev目录下都有对应的条目, 就像df命令所展示的.

```
bash$ df
Filesystem           1k-blocks      Used Available Use%
Mounted on
```

```
/dev/hda6             495876    222748    247527  48% /
/dev/hda1              50755      3887     44248   9% /boot
/dev/hda8             367013     13262    334803   4% /home
/dev/hda5            1714416   1123624    503704  70% /usr
```

在其他方面, /dev 目录也包含环回设备(loopback devices) , 例如/dev/loop0. 环回设备是一个使普通文件能被像对待块设备一样来进行存取的机制. [2] 这使我们可以将一个大文件内的整个文件系统挂载到系统目录下. 参考例子 13-8和例子 13-7.

/dev还有少量的伪设备用于特殊的用途, 例如/dev/null, /dev/zero, /dev/urandom, /dev/sda1, /dev/udp, 和/dev/tcp.

例如:

为了挂载(mount) 一个USB闪盘设备, 将下面一行添加到/etc/fstab. [3]

```
  1 /dev/sda1    /mnt/flashdrive    auto    noauto,user,noatime    0 0
```

(也请参考例子 A-23.)

当对/dev/tcp/$host/$port 伪设备文件执行一个命令时, Bash会打开一个相关的TCP的socket. [4]

从nist.gov得到时间:

```
bash$ cat </dev/tcp/time.nist.gov/13
53082 04-03-18 04:26:54 68 0 0 502.3 UTC(NIST) *
```

[Mark贡献了上面的例子.]

下载一个 URL:

```
bash$ exec 5<>/dev/tcp/www.net.cn/80
bash$ echo -e "GET / HTTP/1.0\n" >&5
bash$ cat <&5
```

[Thanks, Mark 和 Mihai Maties.]

Example 27-1 利用/dev/tcp 来检修故障

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # dev-tcp.sh: 用/dev/tcp 重定向来检查Internet连接.
 3
 4 # Troy Engel编写.
 5 # 已得到作者允许.
 6
 7 TCP_HOST=www.dns-diy.com   # 一个已知的 ISP.
 8 TCP_PORT=80                # http的端口是80 .
 9
10 # 尝试连接. (有些像 'ping' . . .)
11 echo "HEAD / HTTP/1.0" >/dev/tcp/${TCP_HOST}/${TCP_PORT}
12 MYEXIT=$?
13
14 : <<EXPLANATION
15 If bash was compiled with --enable-net-redirections, it has the capability of
16 using a special character device for both TCP and UDP redirections. These
17 redirections are used identically as STDIN/STDOUT/STDERR. The device entries
18 are 30,36 for /dev/tcp:
19
20   mknod /dev/tcp c 30 36
```

```
21
22 >From the bash reference:
23 /dev/tcp/host/port
24     If host is a valid hostname or Internet address, and port is an integer
25 port number or service name, Bash attempts to open a TCP connection to the
26 corresponding socket.
27 EXPLANATION
28
29
30 if [ "X$MYEXIT" = "X0" ]; then
31   echo "Connection successful. Exit code: $MYEXIT"
32 else
33   echo "Connection unsuccessful. Exit code: $MYEXIT"
34 fi
35
36 exit $MYEXIT
```

################################End Script###

译者补充上面这个例子输出的解释(EXPLANATION)译文：

如果bash以--enable-net- redirections选项来编译,它就拥有了使用一个特殊字符设备来完成TCP和UDP重定向功能的能力.这种重定向能力就像 STDIN/STDOUT/STDERR一样被标识.该字符设备/dev/tcp的主次设备号是30,36:

mknod /dev/tcp c 30 36

>摘自bash参考手册：

/dev/tcp/host/port

如果host是一个有效的主机名或因特网有效地址,并且port是一个整数的端口号或是服务名称,Bash会尝试打开一个相对应的TCP连接socket.

注意事项:

[1] /dev目录中的条目是为各种物理设备和虚拟设备提供的挂载点. 这些条目使用非常少的设备空间.
一些像/dev/null, /dev/zero, 和 /dev/urandom的设备是虚拟的. 它们不是真正的物理设备,而只是存在于软件的虚拟设备.

[2] 块设备读或写(或两者兼之)数据都是以块为单位的进行的, 与之相对应的字符设备则使用字符为单位来进行存取.块设备典型的有硬盘和CD-ROM设备,字符设备典型的例子如键盘.

[3] 当然,挂载点/mnt/flashdrive必须存在,如果不存在,以root用户来执行
mkdir /mnt/flashdrive.
为了最终能挂载设备,用下面的命令: mount /mnt/flashdrive
较新的Linux发行版自动把闪盘设备挂载到/media目录.

[4] socket是一种特殊的用于通信的I/O端口. 它允许同一台主机内不同硬件设备间的数据传输,允许在相同网络中的主机间的数据传输,也允许穿越不同网络的主机间的数据传输,当然,也允许在Internet上不同位置主机间的数据传输.

27.2. /proc

/proc目录实际上是一个伪文件系统 . 在 /proc 目录里的文件是当前运行系统和内核进程及它们的相关信息和统计.

```
bash$ cat /proc/devices
Character devices:
  1 mem
```

```
 2 pty
 3 ttyp
 4 ttyS
 5 cua
 7 vcs
10 misc
14 sound
29 fb
36 netlink
128 ptm
136 pts
162 raw
254 pcmcia
Block devices:
  1 ramdisk
  2 fd
  3 ide0
  9 md


bash$ cat /proc/interrupts
           CPU0
  0:      84505          XT-PIC  timer
  1:       3375          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  5:          1          XT-PIC  soundblaster
  8:          1          XT-PIC  rtc
 12:       4231          XT-PIC  PS/2 Mouse
 14:     109373          XT-PIC  ide0
NMI:          0
ERR:          0


bash$ cat /proc/partitions
major minor  #blocks  name     rio rmerge rsect ruse wio wmerge wsect wuse running use aveq
   3     0    3007872 hda 4472 22260 114520 94240 3551 18703 50384 549710 0 111550 644030
   3     1      52416 hda1 27 395 844 960 4 2 14 180 0 800 1140
   3     2          1 hda2 0 0 0 0 0 0 0 0 0 0 0
   3     4     165280 hda4 10 0 20 210 0 0 0 0 0 210 210
   ...


bash$ cat /proc/loadavg
0.13 0.42 0.27 2/44 1119


bash$ cat /proc/apm
1.16 1.2 0x03 0x01 0xff 0x80 -1% -1 ?
```

Shell 脚本可以从/proc目录中的一些文件里提取数据. [1]

```
 1 FS=iso                    # ISO 文件系统是否被内核支持?
 2
 3 grep $FS /proc/filesystems   # iso9660
 1 kernel_version=$( awk '{ print $3 }' /proc/version )
 1 CPU=$( awk '/model name/ {print $4}' < /proc/cpuinfo )
 2
 3 if [ $CPU = Pentium ]
 4 then
 5   run_some_commands
 6   ...
 7 else
 8   run_different_commands
 9   ...
10 fi
 1 devfile="/proc/bus/usb/devices"
 2 USB1="Spd=12"
 3 USB2="Spd=480"
 4
 5
 6 bus_speed=$(grep Spd $devfile | awk '{print $9}')
 7
 8 if [ "$bus_speed" = "$USB1" ]
 9 then
10   echo "USB 1.1 port found."
11   # 这儿开始操作USB 1.1相关的动作.
12 fi
```

/proc目录下有许多不相同的数字命名的子目录. 这些子目录的数字名字都映射对应的当前正在运行的进程的进程号(process ID) . 这些子目录里面有许多文件用于保存对应进程的信息. 文件 stat 和 status 保存着进程运行时的各项统计, the cmdline文件保存该进程的被调用时的命令行参数, 而and the exe 文件是该运行进程完整路径名的符号链接. 还有其他一些文件,但从脚本的观点来看它们都非常的有意思.

Example 27-2 搜索与一个PID相关的进程

```
###################################Start Script#########################################
 1 #!/bin/bash
 2 # pid-identifier.sh: 给出指定PID的进程的程序全路径.
 3
 4 ARGNO=1  # 此脚本期望的参数个数.
 5 E_WRONGARGS=65
 6 E_BADPID=66
 7 E_NOSUCHPROCESS=67
 8 E_NOPERMISSION=68
 9 PROCFILE=exe
10
11 if [ $# -ne $ARGNO ]
12 then
13   echo "Usage: `basename $0` PID-number" >&2  # 帮助信息重定向到标准出错.
14   exit $E_WRONGARGS
```

```
15 fi
16
17 pidno=$( ps ax | grep $1 | awk '{ print $1 }' | grep $1 )
18 # 搜索命令"ps"输出的第一列.
19 # 然后再次确认是真正我们要寻找的进程,而不是这个脚本调用而产生的进程.
20 # 后一个"grep $1"会滤掉这个可能产生的进程.
21 #
22 #    pidno=$( ps ax | awk '{ print $1 }' | grep $1 )
23 #    也可以, 由 Teemu Huovila指出.
24
25 if [ -z "$pidno" ]  # 如果过滤完后结果是一个空字符串,
26 then                # 没有对应的PID进程在运行.
27   echo "No such process running."
28   exit $E_NOSUCHPROCESS
29 fi
30
31 # 也可以用:
32 #   if ! ps $1 > /dev/null 2>&1
33 #   then              # 没有对应的PID进程在运行.
34 #     echo "No such process running."
35 #     exit $E_NOSUCHPROCESS
36 #    fi
37
38 # 为了简化整个进程,使用"pidof".
39
40
41 if [ ! -r "/proc/$1/$PROCFILE" ]  # 检查读权限.
42 then
43   echo "Process $1 running, but..."
44   echo "Can't get read permission on /proc/$1/$PROCFILE."
45   exit $E_NOPERMISSION  # 普通用户不能存取/proc目录的某些文件.
46 fi
47
48 # 最后两个测试可以用下面的代替:
49 #    if ! kill -0 $1 > /dev/null 2>&1 # '0'不是一个信号,
50                                       # 但这样可以测试是否可以
51                                       # 向该进程发送信号.
52 #    then echo "PID doesn't exist or you're not its owner" >&2
53 #    exit $E_BADPID
54 #    fi
55
56
57
58 exe_file=$( ls -l /proc/$1 | grep "exe" | awk '{ print $11 }' )
59 # 或       exe_file=$( ls -l /proc/$1/exe | awk '{print $11}' )
60 #
61 # /proc/pid-number/exe 是进程程序全路径的符号链接.
62 #
```

```
63
64 if [ -e "$exe_file" ] # 如果 /proc/pid-number/exe 存在 ...
65 then               # 则相应的进程存在.
66   echo "Process #$1 invoked by $exe_file."
67 else
68   echo "No such process running."
69 fi
70
71
72 # 这个被详细讲解的脚本几乎可以用下面的命令代替：
73 # ps ax | grep $1 | awk '{ print $5 }'
74 # 然而, 这样并不会工作...
75 # 因为'ps'输出的第5列是进程的argv[0](即命令行第一个参数,调用时程序用的程序路径本身),
76 # 但不是可执行文件.
77 #
78 # 然而, 下面的两个都可以工作.
79 #       find /proc/$1/exe -printf '%l\n'
80 #       lsof -aFn -p $1 -d txt | sed -ne 's/^n//p'
81
82 # 由Stephane Chazelas附加注释.
83
84 exit 0
```

################################End Script##

Example 27-3 网络连接状态

################################Start Script##

```
 1 #!/bin/bash
 2
 3 PROCNAME=pppd       # ppp 守护进程
 4 PROCFILENAME=status  # 在这儿寻找信息.
 5 NOTCONNECTED=65
 6 INTERVAL=2          # 两秒刷新一次.
 7
 8 pidno=$( ps ax | grep -v "ps ax" | grep -v grep | grep $PROCNAME | awk '{ print $1 }' )
 9 # 搜索ppp守护进程'pppd'的进程号.
10 # 一定要过滤掉由搜索进程产生的该行进程.
11 #
12 #  正如Oleg Philon指出的那样,
13 #+ 使用"pidof"命令会相当的简单.
14 #  pidno=$( pidof $PROCNAME )
15 #
16 #  颇有良心的建议:
17 #+ 当命令序列变得复杂的时候,去寻找更简洁的办法. .
18
19
20 if [ -z "$pidno" ]  # 如果没有找到此进程号,则进程没有运行.
21 then
22   echo "Not connected."
23   exit $NOTCONNECTED
```

```
24 else
25   echo "Connected."; echo
26 fi
27
28 while [ true ]      # 死循环,这儿可以有所改进.
29 do
30
31   if [ ! -e "/proc/$pidno/$PROCFILENAME" ]
32   # 进程运行时,对应的"status"文件会存在.
33   then
34     echo "Disconnected."
35     exit $NOTCONNECTED
36   fi
37
38 netstat -s | grep "packets received"  # 取得一些连接统计.
39 netstat -s | grep "packets delivered"
40
41
42   sleep $INTERVAL
43   echo; echo
44
45 done
46
47 exit 0
48
49 # 当要停止它时,可以用Control-C终止.
50
51 #    练习:
52 #    ---------
53 #    改进这个脚本,使它能按"q"键退出.
54 #    给脚本更友好的界面.
```

##################################End Script###

注意: 一般来说, 写/proc目录里的文件是危险 ,因为这样会破坏这个文件系统或摧毁机器.

注意事项:

[1] 一些系统命令, 例如 procinfo, free, vmstat, lsdev, 和uptime也能做类似的事情.

第28章 关于Zeros和Nulls

==========

/dev/zero和/dev/null

使用/dev/null

把/dev/null看作"黑洞". 它非常等价于一个只写文件. 所有写入它的内容都会永远丢失.

而尝试从它那儿读取内容则什么也读不到. 然而, /dev/null对命令行和脚本都非常的有用.

禁止标准输出.

```
1 cat $filename >/dev/null
2 # 文件内容丢失,而不会输出到标准输出.
```

禁止标准错误 (来自例子 12-3).

```
1 rm $badname 2>/dev/null
2 #           这样错误信息[标准错误]就被丢到太平洋去了.
```

禁止标准输出和标准错误的输出.

```
 1 cat $filename 2>/dev/null >/dev/null
 2 # 如果"$filename"不存在,将不会有任何错误信息提示.
 3 # 如果"$filename"存在, 文件的内容不会打印到标准输出.
 4 # 因此Therefore, 上面的代码根本不会输出任何信息.
 5 #
 6 #  当只想测试命令的退出码而不想有任何输出时非常有用.
 7 #
 8 #
 9 # cat $filename &>/dev/null
10 #     也可以, 由 Baris Cicek 指出.
```

删除一个文件的内容, 但是保留文件本身, 和所有的文件权限(来自于Example 2-1和 Example 2-3):

```
 1 cat /dev/null > /var/log/messages
 2 #  : > /var/log/messages   有同样的效果, 但不会产生新的进程.(因为:是内建的)
 3
 4 cat /dev/null > /var/log/wtmp
```

自动清空日志文件的内容 (特别适合处理这些由商业Web站点发送的讨厌的"cookies"):

Example 28-1 隐藏cookie而不再使用

```
################################Start Script#########################################
1 if [ -f ~/.netscape/cookies ]  # 如果存在则删除.
2 then
3   rm -f ~/.netscape/cookies
4 fi
5
6 ln -s /dev/null ~/.netscape/cookies
7 # 现在所有的cookies都会丢入黑洞而不会保存在磁盘上了.
################################End Script###########################################
```

使用/dev/zero

像/dev/null一样, /dev/zero也是一个伪文件, 但它实际上产生连续不断的null的流 (二进制的零流,而不是ASCII型的). 写入它的输出会丢失不见, 而从/dev/zero读出一 连串的null也比较困难, 虽然这也能通过od或一个十六进制编辑器来做到. /dev/zero主 要的用处是用来创建一个指定长度用于初始化的空文件,就像临时交换文件.

Example 28-2 用/dev/zero创建一个交换临时文件

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # 创建一个交换文件.
 3
 4 ROOT_UID=0        # Root 用户的 $UID 是 0.
 5 E_WRONG_USER=65   # 不是 root?
 6
 7 FILE=/swap
 8 BLOCKSIZE=1024
 9 MINBLOCKS=40
10 SUCCESS=0
11
12
13 # 这个脚本必须用root来运行.
14 if [ "$UID" -ne "$ROOT_UID" ]
```

```
15 then
16   echo; echo "You must be root to run this script."; echo
17   exit $E_WRONG_USER
18 fi
19
20
21 blocks=${1:-$MINBLOCKS}         #  如果命令行没有指定,
22                                  #+ 则设置为默认的40块.
23 # 上面这句等同如：
24 # ----------------------------------------------------
25 # if [ -n "$1" ]
26 # then
27 #   blocks=$1
28 # else
29 #   blocks=$MINBLOCKS
30 # fi
31 # ----------------------------------------------------
32
33
34 if [ "$blocks" -lt $MINBLOCKS ]
35 then
36   blocks=$MINBLOCKS             # 最少要有 40 个块长.
37 fi
38
39
40 echo "Creating swap file of size $blocks blocks (KB)."
41 dd if=/dev/zero of=$FILE bs=$BLOCKSIZE count=$blocks  # 把零写入文件.
42
43 mkswap $FILE $blocks            # 将此文件建为交换文件(或称交换分区).
44 swapon $FILE                    # 激活交换文件.
45
46 echo "Swap file created and activated."
47
48 exit $SUCCESS
```

##################################End Script###

关于 /dev/zero 的另一个应用是为特定的目的而用零去填充一个指定大小的文件, 如挂载一个文件系统到环回设备 (loopback device) (参考例子 13-8) 或"安全地" 删除一个文件 (参考例子 12-55).

Example 28-3 创建ramdisk

##################################Start Script###

```
 1 #!/bin/bash
 2 # ramdisk.sh
 3
 4 #  "ramdisk"是系统RAM内存的一段,
 5 #+ 它可以被当成是一个文件系统来操作.
 6 #  它的优点是存取速度非常快 (包括读和写).
 7 #  缺点: 易失性, 当计算机重启或关机时会丢失数据.
 8 #+       会减少系统可用的RAM.
```

```
 9 #
10 #  那么ramdisk有什么作用呢?
11 #  保存一个较大的数据集在ramdisk, 比如一张表或字典,
12 #+ 这样可以加速数据查询, 因为在内存里查找比在磁盘里查找快得多.
13
14
15 E_NON_ROOT_USER=70           # 必须用root来运行.
16 ROOTUSER_NAME=root
17
18 MOUNTPT=/mnt/ramdisk
19 SIZE=2000                   # 2K 个块 (可以合适的做修改)
20 BLOCKSIZE=1024              # 每块有1K (1024 byte) 的大小
21 DEVICE=/dev/ram0            # 第一个 ram 设备
22
23 username=`id -nu`
24 if [ "$username" != "$ROOTUSER_NAME" ]
25 then
26   echo "Must be root to run \"`basename $0`\"."
27   exit $E_NON_ROOT_USER
28 fi
29
30 if [ ! -d "$MOUNTPT" ]       #  测试挂载点是否已经存在了,
31 then                         #+ 如果这个脚本已经运行了好几次了就不会再建这个目录了
32   mkdir $MOUNTPT             #+ 因为前面已经建立了.
33 fi
34
35 dd if=/dev/zero of=$DEVICE count=$SIZE bs=$BLOCKSIZE  # 把RAM设备的内容用零填充.
36                                    # 为何需要这么做?
37 mke2fs $DEVICE               # 在RAM设备上创建一个ext2文件系统.
38 mount $DEVICE $MOUNTPT       # 挂载设备.
39 chmod 777 $MOUNTPT           # 使普通用户也可以存取这个ramdisk.
40                             # 但是, 只能由root来卸载它.
41
42 echo "\"$MOUNTPT\" now available for use."
43 # 现在 ramdisk 即使普通用户也可以用来存取文件了.
44
45 #  注意, ramdisk是易失的, 所以当计算机系统重启或关机时ramdisk里的内容会消失.
46 #
47 #  拷贝所有你想保存文件到一个常规的磁盘目录下.
48
49 # 重启之后, 运行这个脚本再次建立起一个 ramdisk.
50 # 仅重新加载 /mnt/ramdisk 而没有其他的步骤将不会正确工作.
51
52 #  如果加以改进, 这个脚本可以放在 /etc/rc.d/rc.local,
53 #+ 以使系统启动时能自动设立一个ramdisk.
54 #  这样很合适速度要求高的数据库服务器.
55
56 exit 0
```

################################End Script###
最后值得一提的是, ELF二进制文件利用了/dev/zero.
第29章 调试
==========
Debugging is twice as hard as writing the code in the first place. Therefore,
if you write the code as cleverly as possible, you are, by definition, not smart
enough to debug it.
Brian Kernighan
Bash shell 没有自带调试器, 甚至没有任何调试类型的命令或结构. [1] 脚本里的语法错误或拼写错误会产生含糊的错误信息,通常这些在调试非功能性的脚本时没什么帮助.
Example 29-1 一个错误的脚本
################################Start Script###
```
1 #!/bin/bash
2 # ex74.sh
3
4 # 这是一个错误的脚本.
5 # 哪里有错?
6
7 a=37
8
9 if [$a -gt 27 ]
10 then
11   echo $a
12 fi
13
14 exit 0
```
################################End Script###
脚本的输出:
./ex74.sh: [37: command not found
上面的脚本有什么错误(线索: 注意if的后面)?
Example 29-2 丢失关键字(keyword)
################################Start Script###
```
1 #!/bin/bash
2 # missing-keyword.sh: 会产生什么样的错误信息?
3
4 for a in 1 2 3
5 do
6   echo "$a"
7 # done    # 第7行的必需的关键字 'done' 被注释掉了.
8
9 exit 0
```
################################End Script###
脚本的输出:
missing-keyword.sh: line 10: syntax error: unexpected end of file
注意错误信息中说明的错误行不必一定要参考, 但那行是Bash解释器最终认识到是个错误的地方.
出错信息可能在报告语法错误的行号时会忽略脚本的注释行.
如果脚本可以执行,但不是你所期望的那样工作怎么办? 这大多是由于常见的逻辑错误产生的.

Example 29-3 另一个错误脚本

```
################################Start Script#########################################
 1 #!/bin/bash
 2
 3 #  这个脚本目的是为了删除当前目录下的所有文件,包括文件名含有空格的文件.
 4 #
 5 #  但不能工作.
 6 #  为什么?
 7
 8
 9 badname=`ls | grep ' '`
10
11 # 试试这个:
12 # echo "$badname"
13
14 rm "$badname"
15
16 exit 0
################################End Script###########################################
```

为了找出 例子 29-3 的错误可以把echo "$badname" 行的注释去掉. echo 出来的信息对你判断是否脚本以你希望的方式运行时很有帮助.

在这个实际的例子里, rm "$badname" 不会达到想要的结果,因为$badname 没有引用起来.

加上引号以保证rm 命令只有一个参数(这就只能匹配一个文件名). 一个不完善的解决办法是

删除A partial fix is to remove to quotes from $badname and to reset $IFS to contain only a newline, IFS=$'\n'. 不过, 存在更简单的办法.<rojy bug>

```
 1 # 修正删除包含空格文件名时出错的办法.
 2 rm *\ *
 3 rm *" "*
 4 rm *' '*
 5 # Thank you. S.C.
```

总结该脚本的症状,

1. 终止于一个"syntax error"(语法错误)的信息, 或
2. 它能运行, 但不是按期望的那样运行(逻辑错误).
3. 它能运行,运行的和期望的一样, 但有讨厌的副作用 (逻辑炸弹).

用来调试不能工作的脚本的工具包括

1. echo 语句可用在脚本中的有疑问的点上以跟踪了解变量的值, 并且也可以了解后续脚本的动作.

注意: 最好只在调试时才使用echo语句.

```
 1 ### debecho (debug-echo), by Stefano Falsetto ###
 2 ### 只有变量 DEBUG 设置了值时才会打印传递进来的变量值. ###
 3 debecho () {
 4   if [ ! -z "$DEBUG" ]; then
 5     echo "$1" >&2
 6     #       ^^^ 打印到标准出错
 7   fi
 8 }
 9
10 DEBUG=on
```

```
 11 Whatever=whatnot
 12 debecho $Whatever   # whatnot
 13
 14 DEBUG=
 15 Whatever=notwhat
 16 debecho $Whatever   # (这儿就不会打印了.)
```

2. 使用 tee 过滤器来检查临界点的进程或数据流.

3. 设置选项 -n -v -x

sh -n scriptname 不会实际运行脚本,而只是检查脚本的语法错误. 这等同于把 set -n 或 set -o noexec 插入脚本中. 注意还是有一些语法错误不能被这种检查找出来.

sh -v scriptname 在实际执行一个命令前打印出这个命令. 这也等同于在脚本里设置 set -v 或 set -o verbose.

选项 -n 和 -v 可以一块使用. sh -nv scriptname 会打印详细的语法检查.

sh -x scriptname 打印每个命令的执行结果, 但只用在某些小的方面. 它等同于脚本中插入 set -x 或 set -o xtrace.

把 set -u 或 set -o nounset 插入到脚本里并运行它, 就会在每个试图使用没有申明过的变量的地方打印出一个错误信息.

4. 使用一个"assert"(断言) 函数在脚本的临界点上测试变量或条件. (这是从C语言中借用来的.)

Example 29-4 用"assert"测试条件

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # assert.sh
 3
 4 assert ()                 #  如果条件测试失败,
 5 {                         #+ 则打印错误信息并退出脚本.
 6   E_PARAM_ERR=98
 7   E_ASSERT_FAILED=99
 8
 9
10   if [ -z "$2" ]         # 没有传递足够的参数.
11   then
12     return $E_PARAM_ERR   # 什么也不做就返回.
13   fi
14
15   lineno=$2
16
17   if [ ! $1 ]
18   then
19     echo "Assertion failed:  \"$1\""
20     echo "File \"$0\", line $lineno"
21     exit $E_ASSERT_FAILED
22   # else
23   #   return
24   #   返回并继续执行脚本后面的代码.
25   fi
26 }
```

```
27
28
29 a=5
30 b=4
31 condition="$a -lt $b"     #  会错误信息并从脚本退出.
32                          #  把这个 “ 条件 ” 放在某个地方,
33                          #+ 然后看看有什么现象.
34
35 assert "$condition" $LINENO
36 # 脚本以下的代码只有当"assert"成功时才会继续执行.
37
38
39 # 其他的命令.
40 # ...
41 echo "This statement echoes only if the \"assert\" does not fail."
42 # ...
43 # 余下的其他命令.
44
45 exit 0
```

################################End Script###

5. 用变量$LINENO和内建的caller.

6. 捕捉exit.

脚本中的The exit 命令会触发信号0,终结进程,即脚本本身. [2] 这常用来捕捉exit命令做某事, 如强制打印变量值. trap 命令必须是脚本中第一个命令.

捕捉信号

trap

当收到一个信号时指定一个处理动作; 这在调试时也很有用.

注意: 信号是发往一个进程的非常简单的信息, 要么是由内核发出要么是由另一个进程, 以告诉接收进程采取一些指定的动作 (一般是中止). 例如, 按Control-C, 发送一个用户中断(即 INT 信号)到运行中的进程.

```
1 trap " 2
2 # 忽略信号 2 (Control-C), 没有指定处理动作.
3
4 trap 'echo "Control-C disabled."' 2
5 # 当按 Control-C 时显示一行信息.
```

Example 29-5 捕捉 exit

################################Start Script###

```
1 #!/bin/bash
2 # 用trap捕捉变量值.
3
4 trap 'echo Variable Listing --- a = $a  b = $b' EXIT
5 #  EXIT 是脚本中exit命令产生的信号的信号名.
6 #
7 #  由"trap"指定的命令不会被马上执行,只有当发送了一个适应的信号时才会执行.
8 #
9
10 echo "This prints before the \"trap\" --"
11 echo "even though the script sees the \"trap\" first."
```

```
12 echo
13
14 a=39
15
16 b=36
17
18 exit 0
19 #  注意到注释掉上面一行的'exit'命令也没有什么不同,
20 #+ 这是因为执行完所有的命令脚本都会退出.
```

################################End Script###

Example 29-6 在Control-C后清除垃圾

################################Start Script###

```
 1 #!/bin/bash
 2 # logon.sh: 简陋的检查你是否还处于连线的脚本.
 3
 4 umask 177  # 确定临时文件不是全部用户都可读的.
 5
 6
 7 TRUE=1
 8 LOGFILE=/var/log/messages
 9 #  注意 $LOGFILE 必须是可读的
10 #+ (用 root来做：chmod 644 /var/log/messages).
11 TEMPFILE=temp.$$
12 #  创建一个"唯一的"临时文件名, 使用脚本的进程ID.
13 #     用 'mktemp' 是另一个可行的办法.
14 #     举例:
15 #     TEMPFILE=`mktemp temp.XXXXXX`
16 KEYWORD=address
17 #  上网时, 把"remote IP address xxx.xxx.xxx.xxx"这行
18 #                  加到 /var/log/messages.
19 ONLINE=22
20 USER_INTERRUPT=13
21 CHECK_LINES=100
22 #  日志文件中有多少行要检查.
23
24 trap 'rm -f $TEMPFILE; exit $USER_INTERRUPT' TERM INT
25 #  如果脚本被control-c中断了,则清除临时文件.
26
27 echo
28
29 while [ $TRUE ]  #死循环.
30 do
31   tail -$CHECK_LINES $LOGFILE> $TEMPFILE
32   #  保存系统日志文件的最后100行到临时文件.
33   #  这是需要的, 因为新版本的内核在登录网络时产生许多日志文件信息.
34   search=`grep $KEYWORD $TEMPFILE`
35   #  检查"IP address" 短语是不是存在,
36   #+ 它指示了一次成功的网络登录.
```

```
37
38  if [ ! -z "$search" ] #  引号是必须的,因为变量可能会有一些空白符.
39  then
40    echo "On-line"
41    rm -f $TEMPFILE    #  清除临时文件.
42    exit $ONLINE
43  else
44    echo -n "."       #  -n 选项使echo不会产生新行符,
45                      #+ 这样你可以从该行的继续打印.
46  fi
47
48  sleep 1
49 done
50
51
52 #  注: 如果你更改KEYWORD变量的值为"Exit",
53 #+ 这个脚本就能用来在网络登录后检查掉线
54 #
55
56 # 练习: 修改脚本,像上面所说的那样,并修正得更好
57 #
58
59 exit 0
60
61
62 # Nick Drage 建议用另一种方法:
63
64 while true
65  do ifconfig ppp0 | grep UP 1> /dev/null && echo "connected" && exit 0
66  echo -n "."   # 在连接上之前打印点 (.....).
67  sleep 2
68 done
69
70 # 问题: 用 Control-C来终止这个进程可能是不够的.
71 #+        (点可能会继续被打印.)
72 # 练习: 修复这个问题.
73
74
75
76 # Stephane Chazelas 也提出了另一个办法:
77
78 CHECK_INTERVAL=1
79
80 while ! tail -1 "$LOGFILE" | grep -q "$KEYWORD"
81 do echo -n .
82    sleep $CHECK_INTERVAL
83 done
84 echo "On-line"
```

85
86 # 练习: 讨论这几个方法的优缺点.
87 #
################################End Script###
注意: trap 的DEBUG参数在每个命令执行完后都会引起一个指定的执行动作,例如,这可用来
 跟踪变量.
Example 29-7 跟踪变量
################################Start Script###
```
 1 #!/bin/bash
 2
 3 trap 'echo "VARIABLE-TRACE> \$variable = \"$variable\"" DEBUG
 4 # 在每个命令行显示变量$variable 的值.
 5
 6 variable=29
 7
 8 echo "Just initialized \"\$variable\" to $variable."
 9
10 let "variable *= 3"
11 echo "Just multiplied \"\$variable\" by 3."
12
13 exit $?
14
15 #  "trap 'command1 . . . command2 . . .' DEBUG" 的结构适合复杂脚本的环境
16 #+ 在这种情况下多次"echo $variable"比较没有技巧并且也耗时.
17 #
18 #
19
20 # Thanks, Stephane Chazelas 指出这一点.
21
22
23 脚本的输出:
24
25 VARIABLE-TRACE> $variable = ""
26 VARIABLE-TRACE> $variable = "29"
27 Just initialized "$variable" to 29.
28 VARIABLE-TRACE> $variable = "29"
29 VARIABLE-TRACE> $variable = "87"
30 Just multiplied "$variable" by 3.
31 VARIABLE-TRACE> $variable = "87"
```
################################End Script###
当然, trap 命令除了调试还有其他的用处.
Example 29-8 运行多进程 (在多处理器的机器里)
################################Start Script###
```
 1 #!/bin/bash
 2 # parent.sh
 3 # 在多处理器的机器里运行多进程.
 4 # 作者: Tedman Eng
 5
```

```
6 #  这是要介绍的两个脚本的第一个,
7 #+ 这两个脚本都在要在相同的工作目录下.
8
9
10
11
12 LIMIT=$1        # 要启动的进程总数
13 NUMPROC=4       # 当前进程数 (forks?)
14 PROCID=1        # 启动的进程ID
15 echo "My PID is $$"
16
17 function start_thread() {
18        if [ $PROCID -le $LIMIT ] ; then
19                ./child.sh $PROCID&
20                let "PROCID++"
21        else
22           echo "Limit reached."
23           wait
24           exit
25        fi
26 }
27
28 while [ "$NUMPROC" -gt 0 ]; do
29        start_thread;
30        let "NUMPROC--"
31 done
32
33
34 while true
35 do
36
37 trap "start_thread" SIGRTMIN
38
39 done
40
41 exit 0
42
43
44
45 # ======== 下面是第二个脚本 ========
46
47
48 #!/bin/bash
49 # child.sh
50 # 在多处理器的机器里运行多进程.
51 # 这个脚本由parent.sh脚本调用(即上面的脚本).
52 # 作者: Tedman Eng
53
```

```
54 temp=$RANDOM
55 index=$1
56 shift
57 let "temp %= 5"
58 let "temp += 4"
59 echo "Starting $index  Time:$temp" "$@"
60 sleep ${temp}
61 echo "Ending $index"
62 kill -s SIGRTMIN $PPID
63
64 exit 0
65
66
67 # ======================== 脚本作者注 ======================== #
68 #  这不是完全没有bug的脚本.
69 #  我运行LIMIT = 500 ,在过了开头的一二百个循环后,
70 #+ 这些进程有一个消失了!
71 #  不能确定是不是因为捕捉信号产生碰撞还是其他的原因.
72 #  一但信号捕捉到,在下一个信号设置之前,
73 #+ 会有一个短暂的时间来执行信号处理程序,
74 #+ 这段时间内很可能会丢失一个信号捕捉,因此失去生成一个子进程的机会.
75
76 #  毫无疑问会有人能找出这个bug的原因,并且修复它
77 #+ . . . 在将来的某个时候.
78
79
80
81 # ============================================================================ #
82
83
84
85 # ---------------------------------------------------------------------#
86
87
88
89 ######################################################################
90 # 下面的脚本由Vernia Damiano原创.
91 # 不幸地是, 它不能正确工作.
92 ######################################################################
93
94 #!/bin/bash
95
96 #  必须以最少一个整数参数来调用这个脚本
97 #+ (这个整数是协作进程的数目).
98 #  所有的其他参数被传给要启动的进程.
99
100
101 INDICE=8        # 要启动的进程数目
```

```
102 TEMPO=5        # 每个进程最大的睡眼时间
103 E_BADARGS=65   # 没有参数传给脚本的错误值.
104
105 if [ $# -eq 0 ] # 检查是否至少传了一个参数给脚本.
106 then
107   echo "Usage: `basename $0` number_of_processes [passed params]"
108   exit $E_BADARGS
109 fi
110
111 NUMPROC=$1            # 协作进程的数目
112 shift
113 PARAMETRI=( "$@" )     # 每个进程的参数
114
115 function avvia() {
116         local temp
117         local index
118         temp=$RANDOM
119         index=$1
120         shift
121         let "temp %= $TEMPO"
122         let "temp += 1"
123         echo "Starting $index Time:$temp" "$@"
124         sleep ${temp}
125         echo "Ending $index"
126         kill -s SIGRTMIN $$
127 }
128
129 function parti() {
130         if [ $INDICE -gt 0 ] ; then
131             avvia $INDICE "${PARAMETRI[@]}" &
132               let "INDICE--"
133         else
134               trap : SIGRTMIN
135         fi
136 }
137
138 trap parti SIGRTMIN
139
140 while [ "$NUMPROC" -gt 0 ]; do
141         parti;
142         let "NUMPROC--"
143 done
144
145 wait
146 trap - SIGRTMIN
147
148 exit $?
149
```

150 : <<SCRIPT_AUTHOR_COMMENTS
151 我需要运行能指定选项的一个程序,
152 能接受许多不同的文件,并在一个多处理器的机器上运行
153 所以我想(我也将会)使指定数目的进程运行,并且每个进程终止后都能启动一个新的
154
155
156 "wait"命令没什么帮助, 因为它是等候一个指定的或所有的后台进程.
157 所以我写了这个使用了trap指令的bash脚本来做这个任务.
158
159 --Vernia Damiano
160 SCRIPT_AUTHOR_COMMENTS
################################End Script##
注意: trap " SIGNAL (两个引号引空) 在脚本中禁用了 SIGNAL 信号的动作(即忽略了).
trap SIGNAL 则恢复了 SIGNAL 信号前次的处理动作. 这在保护脚本的某些临界点的
位置不受意外的中断影响时很有用.

```
1 trap " 2 # 信号 2是 Control-C, 现在被忽略了.
2 command
3 command
4 command
5 trap 2 # 再启用Control-C
6
```

Bash的版本 3 增加了下面的特殊变量用于调试.

1. $BASH_ARGC
2. $BASH_ARGV
3. $BASH_COMMAND
4. $BASH_EXECUTION_STRING
5. $BASH_LINENO
6. $BASH_SOURCE
7. $BASH_SUBSHELL

注意事项:
[1] Rocky Bernstein的 Bash debugger 实际上填补了这个空白.
[2] 依据惯例,信号0 被指定为退出(exit).
第30章 选项
==========
选项用来更改shell或/和脚本行为的机制.
set 命令用来在脚本里激活各种选项. 在脚本中任何你想让选项生效的地方,插入
set -o option-name 或, 用更简短的格式, set -option-abbrev. 这两种格式都是等价的.

```
1 #!/bin/bash
2
3 set -o verbose
4 # 执行前打印命令.
1 #!/bin/bash
2
3 set -v
4 # 和上面的有完全相同的效果.
```

注意: 为了在脚本里停用一个选项, 插入 set +o option-name 或 set +option-abbrev.

```
1 #!/bin/bash
2
```

```
 3      set -o verbose
 4      # 激活命令回显.
 5      command
 6      ...
 7      command
 8
 9      set +o verbose
10      # 停用命令回显.
11      command
12      # 没有回显命令了.
13
14
15      set -v
16      # 激活命令回显.
17      command
18      ...
19      command
20
21      set +v
22      # 停用命令回显.
23      command
24
25      exit 0
26
```

另一个在脚本里启用选项的方法是在脚本头部的#!后面指定选项.

```
 1      #!/bin/bash -x
 2      #
 3      # 下面是脚本的主要内容.
 4
```

从命令行来激活脚本的选项也是可以办到的. 一些不能和set一起用的选项可以用在命令行指定. -i是其中之一, 可以使脚本以交互方式运行.

bash -v script-name

bash -o verbose script-name

下面的表格列举了一些有用的选项. 它们都可以用简短格式来指定(以一个短横线开头)也可以用完整的名字来指定(用双短横线开头或用-o来指定).

table 30-1 Bash 选项

缩写	名称	作用
-C	noclobber	防止重定向时覆盖文件 (此作用会被>|覆盖)
-D	(none)	列出双引号引起的含有$前缀的字符串,但不执行脚本中的命令
-a	allexport	导出所有定义的变量到环境变量中
-b	notify	当后台任务终止时给出通知 (在脚本中用的不多)

| -c... | (none) | 从...读命令
==========
| -e | errexit | 脚本发生第一个错误时就中止脚本运行,即当一个命令
| | | 返回非零值时退出脚本 (除了until 或 while loops,
| | | if-tests, list constructs)
==========
| -f | noglob | 文件名替换停用(指像*这样的符号不能替换为文件名了)
==========
| -i | interactive | 使脚本以交互式方式运行
==========
| -n | noexec | 从脚本里读命令但不执行它们(语法检查)
==========
| -o Option-Name | (none) | 调用Option-Name 选项
==========
| -o posix | POSIX | 更改Bash或脚本的行为,使之符合POSIX标准.
==========
| -p | privileged | 脚本作为"suid"程序来运行 (小心!)
==========
| -r | restricted | 脚本在受限模式中运行 (参考第21章).
==========
| -s | stdin | 从标准输入读命令
==========
| -t | (none) | 第一个命令后就退出
==========
| -u | nounset | 当使用一个未定义的变量时产生一个错误信息,并强制
| | | 退出脚本.
==========
| -v | verbose | 执行命令之前打印命令到标准输出
==========
| -x | xtrace | 与-v相似, 但打印完整的命令
==========
| - | (none) | 选项列表结束的标志. 后面的参数是位置参数.
==========
| -- | (none) | 释放位置参数. 如果参数列表被指定了(-- arg1 arg2),
| | | 则位置参数被依次设置为参数列表中的值.
==========
第31章 Gotchas
==========
Turandot: Gli enigmi sono tre, la morte una!
Caleph: No, no! Gli enigmi sono tre, una la vita!
Puccini
将保留字和字符声明为变量名.

```
1 case=value0      # 引发错误.
2 23skidoo=value1  # 也会有错误.
3 # 以数字开头的变量名是由shell保留使用的.
4 # 试试 _23skidoo=value1. 用下划线开头的变量名是允许的.
5
6 # 但是 . . .  仅使用下划线来用做变量名也是不行的.
```

```
 7 _=25
 8 echo $_          # $_ 是一个特殊的变量,被设置为最后命令的最后一个参数.
 9
10 xyz((!*=value2    # 引起严重的错误.
11 # 在第三版的Bash, 标点不能在变量名中出现.
```

用连字符或其他保留字符当做变量名(或函数名).

```
 1 var-1=23
 2 # 用 'var_1' 代替.
 3
 4 function-whatever ()   # 错误
 5 # 用 'function_whatever ()' 代替.
 6
 7
 8 # 在第三版的 Bash, 标点不能在函数名中使用.
 9 function.whatever ()   # 错误
10 # 用 'functionWhatever ()' 代替.
```

给变量和函数使用相同的名字. 这会使脚本不能分辨两者.

```
 1 do_something ()
 2 {
 3   echo "This function does something with \"$1\"."
 4 }
 5
 6 do_something=do_something
 7
 8 do_something do_something
 9
10 # 这些都是合法的,但让人混淆.
```

不适当地使用宽白符(whitespace). 和其它的编程语言相比,Bash非常讲究空白字符的使用.

```
 1 var1 = 23   # 'var1=23' 正确.
 2 # 上面一行,Bash试图执行命令"var1"
 3 # 并且它的参数是"="和"23".
 4
 5 let c = $a - $b   # 'let c=$a-$b' 或 'let "c = $a - $b"'是正确的.
 6
 7 if [ $a -le 5]    # if [ $a -le 5 ]   是正确的.
 8 # if [ "$a" -le 5 ]   会更好.
 9 # [[ $a -le 5 ]] 也可以.
```

未初始化的变量(指赋值前的变量)被认为是NULL值的,而不是有零值.

```
 1 #!/bin/bash
 2
 3 echo "uninitialized_var = $uninitialized_var"
 4 # uninitialized_var =
```

混淆测试里的= 和 -eq 操作符. 请记住, = 是比较字符变量而 -eq 比较整数.

```
 1 if [ "$a" = 273 ]      # $a 是一个整数还是一个字符串?
 2 if [ "$a" -eq 273 ]    # 如果$a 是一个整数,用这个表达式.
 3
 4 # 有时你能混用 -eq 和 = 而没有不利的结果.
 5 # 然而 . . .
```

```
 6
 7
 8 a=273.0   # 不是一个整数.
 9
10 if [ "$a" = 273 ]
11 then
12   echo "Comparison works."
13 else
14   echo "Comparison does not work."
15 fi    # Comparison does not work.
16
17 # 与   a=" 273"  和 a="0273" 一样.
18
19
20 # 同样, 问题仍然是试图对非整数值使用 "-eq" 测试.
21
22 if [ "$a" -eq 273.0 ]
23 then
24   echo "a = $a"
25 fi  # 因错误信息而中断.
26 # test.sh: [: 273.0: integer expression expected
```

误用字符串比较操作符.

Example 31-1 数字和字符串比较是不相等同的

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # bad-op.sh: 在整数比较中使用字符串比较.
 3
 4 echo
 5 number=1
 6
 7 # 下面的 "while" 循环有两个错误:
 8 #+ 一个很明显,另一个比较隐蔽.
 9
10 while [ "$number" < 5 ]    # 错误! 应该是:  while [ "$number" -lt 5 ]
11 do
12   echo -n "$number "
13   let "number += 1"
14 done
15 #  尝试运行时会收到错误信息而退出:
16 #+ bad-op.sh: line 10: 5: No such file or directory
17 #  在单括号里, "<" 需要转义,
18 #+ 而即使是如此, 对此整数比较它仍然是错的.
19
20
21 echo "----------------------"
22
23
24 while [ "$number" \< 5 ]    #  1 2 3 4
```

```
25 do                        #
26   echo -n "$number "       #  看起来好像是能工作的, 但 . . .
27   let "number += 1"        #+ 它其实是在对 ASCII 码的比较,
28 done                      #+ 而非是对数值的比较.
29
30 echo; echo "----------------------"
31
32 # 下面这样便会引起问题了. 例如:
33
34 lesser=5
35 greater=105
36
37 if [ "$greater" \< "$lesser" ]
38 then
39   echo "$greater is less than $lesser"
40 fi                           # 105 is less than 5
41 #  事实上, "105" 小于 "5"
42 #+ 是因为使用了字符串比较 (以ASCII码的排序顺序比较).
43
44 echo
45
46 exit 0
```

##################################End Script##

有时在测试时的方括号([ ])里的变量需要引用起来(双引号). 如果没有这么做可能会引起不可预料的结果. 参考例子 7-6, 例子 16-5, 和 例子 9-6.

在脚本里的命令可能会因为脚本没有运行权限而导致运行失败. 如果用户不能在命令行里调用一个命令,即使把这个命令加到一个脚本中也一样会失败. 这时可以尝试更改访命令的属性, 甚至可能给它设置suid位(当然是以root来设置).

试图用 - 来做重定向操作(事实上它不是操作符)会导致令人讨厌的意外.

```
1 command1 2> - | command2  # 试图把command1的错误重定向到一个管道里...
2 #    ...不会工作.
3
4 command1 2>& - | command2  # 也没有效果.
5
6 Thanks, S.C.
```

用 Bash 版本 2+ 的功能可以当有错误信息时引发修复动作. 老一些的 Linux机器可能默认的安装是 1.XX 版本的Bash.

```
1 #!/bin/bash
2
3 minimum_version=2
4 # 因为 Chet Ramey 经常给Bash增加新的特性,
5 # 你把 $minimum_version 设为 2.XX比较合适,或者是其他合适的值.
6 E_BAD_VERSION=80
7
8 if [ "$BASH_VERSION" \< "$minimum_version" ]
9 then
10   echo "This script works only with Bash, version $minimum or greater."
11   echo "Upgrade strongly recommended."
```

```
 12   exit $E_BAD_VERSION
 13 fi
 14
 15 ...
```

在非Linux的机器上使用Bourne shell脚本(#!/bin/sh)的Bash专有功能可能会引起不可预料的行为. Linux系统通常都把sh 取别名为 bash, 但在其他的常见的UNIX系统却不一定是这样.
使用Bash中没有文档化的属性是危险的尝试. 在这本书的前几版中有几个脚本依赖于exit或return的值没有限制不能用负整数(虽然限制了exit或return 的最大值是255). 不幸地是, 在版本 2.05b 以上这种情况就消失了. 参考See 例子 23-9.
一个带有DOS风格新行符 (\r\n) 的脚本会执行失败, 因为#!/bin/bash\r\n 不是合法的,不同于合法的#!/bin/bash\n. 解决办法就是把脚本转换成UNIX风格的新行符.

```
  1 #!/bin/bash
  2
  3 echo "Here"
  4
  5 unix2dos $0    # 脚本先把自己改成DOS格式.
  6 chmod 755 $0   # 更改回执行权限.
  7               # 'unix2dos'命令会删除执行权限.
  8
  9 ./$0          # 脚本尝试再次运行自己本身.
 10               # 但它是一个DOS文件而不会正常工作了.
 11
 12 echo "There"
 13
 14 exit 0
```

shell脚本以 #!/bin/sh 行开头将不会在Bash兼容的模式下运行. 一些Bash专有的功能可能会被禁用掉. 那些需要完全使用Bash专有扩展特性的脚本应该用#!/bin/bash开头.
脚本里在 here document 的终结输入的字符串前加入空白字符会引起不可预料的结果.
脚本不能export(导出)变量到它的父进程(parent process),或父进程的环境里. 就像我们学的生物一样,一个子进程可以从父进程里继承但不能去影响父进程.

```
  1 WHATEVER=/home/bozo
  2 export WHATEVER
  3 exit 0
 bash$ echo $WHATEVER

 bash$
```

可以确定, 回到命令提示符, $WHATEVER 变量仍然没有设置.
在子SHELL(subshell)设置和操作变量 , 然后尝试在子SHELL的作用范围外使用相同名的变量将会导致非期望的结果.
Example 31-2 子SHELL缺陷
##################################Start Script###

```
 1 #!/bin/bash
 2 # 在子SHELL中的变量缺陷.
 3
 4 outer_variable=outer
 5 echo
 6 echo "outer_variable = $outer_variable"
 7 echo
```

```
 8
 9 (
10 # 子SHELL开始
11
12 echo "outer_variable inside subshell = $outer_variable"
13 inner_variable=inner  # Set
14 echo "inner_variable inside subshell = $inner_variable"
15 outer_variable=inner  # Will value change globally?
16 echo "outer_variable inside subshell = $outer_variable"
17
18 # 导出变量会有什么不同吗?
19 #    export inner_variable
20 #    export outer_variable
21 # 试试看.
22
23 # 子SHELL结束
24 )
25
26 echo
27 echo "inner_variable outside subshell = $inner_variable"  # Unset.
28 echo "outer_variable outside subshell = $outer_variable"  # Unchanged.
29 echo
30
31 exit 0
32
33 # 如果你没有注释第 19 和 20行会怎么样?
34 # 会有什么不同吗?
```

################################End Script###

把 echo 的输出用管道(Piping)输送给read命令可能会产生不可预料的结果. 在这个情况下, read 表现地好像它是在一个子SHELL里一样. 可用set 命令代替 (就像在例子 11-16里的一样).

Example 31-3 把echo的输出用管道输送给read命令

################################Start Script###

```
 1 #!/bin/bash
 2 #  badread.sh:
 3 #  尝试用 'echo 和 'read'
 4 #+ 来达到不用交互地给变量赋值的目的.
 5
 6 a=aaa
 7 b=bbb
 8 c=ccc
 9
10 echo "one two three" | read a b c
11 # 试图重新给 a, b, 和 c赋值.
12
13 echo
14 echo "a = $a"  # a = aaa
15 echo "b = $b"  # b = bbb
```

```
16 echo "c = $c"  # c = ccc
17 # 重新赋值失败.
18
19 # ------------------------------
20
21 # 用下面的另一种方法.
22
23 var=`echo "one two three"`
24 set -- $var
25 a=$1; b=$2; c=$3
26
27 echo "-------"
28 echo "a = $a"  # a = one
29 echo "b = $b"  # b = two
30 echo "c = $c"  # c = three
31 # 重新赋值成功.
32
33 # ------------------------------
34
35 #  也请注意echo值到'read'命令里是在一个子SHELL里起作用的.
36 #  所以,变量的值只在子SHELL里被改变了.
37
38 a=aaa          # 从头开始.
39 b=bbb
40 c=ccc
41
42 echo; echo
43 echo "one two three" | ( read a b c;
44 echo "Inside subshell: "; echo "a = $a"; echo "b = $b"; echo "c = $c" )
45 # a = one
46 # b = two
47 # c = three
48 echo "-----------------"
49 echo "Outside subshell: "
50 echo "a = $a"  # a = aaa
51 echo "b = $b"  # b = bbb
52 echo "c = $c"  # c = ccc
53 echo
54
55 exit 0
#################################End Script#############################################
```

事实上, 也正如 Anthony Richardson 指出的那样, 管道任何的数据到循环里都会引起相似的问题.

```
 1 # 循环管道问题.
 2 #  Anthony Richardson编写此例,
 3 #+ Wilbert Berendsen补遗此例.
 4
 5
```

```
 6 foundone=false
 7 find $HOME -type f -atime +30 -size 100k |
 8 while true
 9 do
10    read f
11    echo "$f is over 100KB and has not been accessed in over 30 days"
12    echo "Consider moving the file to archives."
13    foundone=true
14    # -----------------------------------
15    echo "Subshell level = $BASH_SUBSHELL"
16    # Subshell level = 1
17    # 没错, 现在是在子shell里头运行.
18    # -----------------------------------
19 done
20
21 #  foundone 变量在此总是有false值
22 #+ 因此它是在子SHELL里被设为true值的
23 if [ $foundone = false ]
24 then
25    echo "No files need archiving."
26 fi
27
28 # =====================现在, 使用正确的方法:=================
29
30 foundone=false
31 for f in $(find $HOME -type f -atime +30 -size 100k)  # 没有使用管道.
32 do
33    echo "$f is over 100KB and has not been accessed in over 30 days"
34    echo "Consider moving the file to archives."
35    foundone=true
36 done
37
38 if [ $foundone = false ]
39 then
40    echo "No files need archiving."
41 fi
42
43 # ==================另一种方法==================
44
45 #  脚本中读变量值的相应部分替换在代码块里头读变量,
46 #+ 这使变量能在相同的子SHELL里共享了.
47 #  Thank you, W.B.
48
49 find $HOME -type f -atime +30 -size 100k | {
50     foundone=false
51     while read f
52     do
53       echo "$f is over 100KB and has not been accessed in over 30 days"
```

```
 54       echo "Consider moving the file to archives."
 55       foundone=true
 56     done
 57
 58     if ! $foundone
 59     then
 60       echo "No files need archiving."
 61     fi
 62 }
```

相关的问题是：当尝试写 tail -f 的输出给管道并传递给grep时会发生问题.

```
  1 tail -f /var/log/messages | grep "$ERROR_MSG" >> error.log
  2 # "error.log"文件里将不会写入任何东西.
```

--

在脚本中使用"suid" 的命令是危险的, 因为这会危及系统安全. [1]

用shell编写CGI程序是值得商榷的. Shell脚本的变量不是"类型安全的", 这样它用于CGI连接使用时会引发不希望的结果. 其次, 它很难防范骇客的攻击.

Bash 不能正确处理双斜线 (//) 字符串.

在Linux 或 BSD上写的Bash脚本可能需要修正以使它们也能在商业的UNIX (或 Apple OSX)上运行. 这些脚本常使用比一般的UNIX系统上的同类工具更强大功能的GNU 命令和过滤工具. 这方面一个明显的例子是文本处理工具tr.

Danger is near thee --
Beware, beware, beware, beware.
Many brave hearts are asleep in the deep.
So beware --
Beware.
A.J. Lamb and H.W. Petrie

注意事项:

[1] 给脚本设置suid 权限是没有用的.

第32章 脚本编程风格

==========

写脚本时养成结构化和系统方法的习惯. 即使你在信封背后随便做一下草稿也是有益的,要养成在写代码前花几分钟来规划和组织你的想法.

这儿是一些风格的指南. 注意这节文档不是想成为一个官方Shell编程风格.

32.1. 非官方的Shell脚本风格

---

* 注释你的代码.这会使你的代码更容易让别人理解和赏识,同时也便于你维护.

```
  1 PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"
  2 # 当你去年写下这句代码时非常的了解它在干什么事,但现在它完全是一个谜.
  3 # (摘自 Antek Sawicki的"pw.sh" 脚本.)
```

给脚本和函数加上描述性的头部信息.

```
  1 #!/bin/bash
  2
  3 #***********************************************#
  4 #               xyz.sh                  #
  5 #         written by Bozo Bozeman           #
  6 #             July 05, 2001               #
  7 #                                 #
  8 #               清除项目文件.             #
```

```
 9 #************************************************#
10
11 E_BADDIR=65                    # 没有那样的目录.
12 projectdir=/home/bozo/projects    # 要清除的目录.
13
14 # ------------------------------------------------------ #
15 # cleanup_pfiles ()                                      #
16 # 删除指定目录里的所有文件.                              #
17 # 参数: $target_directory                                #
18 # 返回: 成功返回0 , 失败返回$E_BADDIR值.                 #
19 # ------------------------------------------------------ #
20 cleanup_pfiles ()
21 {
22   if [ ! -d "$1" ] # 测试目标目录是否存在.
23   then
24     echo "$1 is not a directory."
25     return $E_BADDIR
26   fi
27
28   rm -f "$1"/*
29   return 0   # 成功.
30 }
31
32 cleanup_pfiles $projectdir
33
34 exit 0
```

确认 #!/bin/bash 在脚本的第一行,在任何头部注释行之前.

* 避免使用 "魔数," [1] 它是硬编码的字符常量. 用有意义的变量名来代替. 这使脚本更容易理解并允许在不破坏应用的情况下做改变和更新.

```
 1 if [ -f /var/log/messages ]
 2 then
 3   ...
 4 fi
 5 # 一年以后,你决定让脚本改为检查 /var/log/syslog.
 6 # 那么现在就需要你手动修改脚本里每一处的要改动的代码,
 7 # 希望不要有你疏漏的地方.
 8
 9 # 更好的办法是:
10 LOGFILE=/var/log/messages  # 只需要改动一行.
11 if [ -f "$LOGFILE" ]
12 then
13   ...
14 fi
```

* 为变量和函数选择描述性的名字.

```
 1 fl=`ls -al $dirname`              # 含义含糊.
 2 file_listing=`ls -al $dirname`    # 更好的名字.
 3
 4
```

```
 5 MAXVAL=10  # 同一个脚本所有程序代码使用脚本常量.
 6 while [ "$index" -le "$MAXVAL" ]
 7 ...
 8
 9
10 E_NOTFOUND=75                  #  把错误代码的代表的变量名大写U,
11                                 # +并以"E_"开头.
12 if [ ! -e "$filename" ]
13 then
14   echo "File $filename not found."
15   exit $E_NOTFOUND
16 fi
17
18
19 MAIL_DIRECTORY=/var/spool/mail/bozo  # 环境变量名用大写.
20 export MAIL_DIRECTORY
21
22
23 GetAnswer ()                    # 函数名用适当的大小写混合组成.
24 {
25   prompt=$1
26   echo -n $prompt
27   read answer
28   return $answer
29 }
30
31 GetAnswer "What is your favorite number? "
32 favorite_number=$?
33 echo $favorite_number
34
35
36 _uservariable=23                # 语法允许, 但不推荐.
37 # 用户定义的变量最好不要用下划线开头.
38 # 把这个留给系统变量使用更好.
```

* 用有含义和系统的方法来使用退出代码(exit codes).

```
 1 E_WRONG_ARGS=65
 2 ...
 3 ...
 4 exit $E_WRONG_ARGS
```

也参考附录 D.

最后 建议在脚本中使用/usr/include/sysexits.h的退出码, 虽然它们主要由 C 和 C++ 语言编程时使用.

* 使用标准的参数选项. 最后 建议使用下面一组参数标志.

```
 1 -a      All: Return all information (including hidden file info).
 2 -b      Brief: Short version, usually for other scripts.
 3 -c      Copy, concatenate, etc.
 4 -d      Daily: Use information from the whole day, and not merely
 5         information for a specific instance/user.
```

```
  6 -e      Extended/Elaborate: (often does not include hidden file info).
  7 -h      Help: Verbose usage w/descs, aux info, discussion, help.
  8         See also -V.
  9 -l      Log output of script.
 10 -m      Manual: Launch man-page for base command.
 11 -n      Numbers: Numerical data only.
 12 -r      Recursive: All files in a directory (and/or all sub-dirs).
 13 -s      Setup & File Maintenance: Config files for this script.
 14 -u      Usage: List of invocation flags for the script.
 15 -v      Verbose: Human readable output, more or less formatted.
 16 -V      Version / License / Copy(right|left) / Contribs (email too).
```

也参考附录 F.

* 把复杂的脚本分割成简单一些的模块. 用合适的函数来实现各个功能. 参考例子 34-4.
* 如果有简单的结构可以使用,不要使用复杂的结构.

```
  1 COMMAND
  2 if [ $? -eq 0 ]
  3 ...
  4 # 多余的并且也不直接明了.
  5
  6 if COMMAND
  7 ...
  8 # 更简练 (或者可能会损失一些可读性).
```

```
   ... reading the UNIX source code to the Bourne shell
   (/bin/sh). I was shocked at how much simple algorithms could
   be made cryptic, and therefore useless, by a poor choice of
   code style. I asked myself, "Could someone be proud of this
   code?"
            Landon Noll
```

注意事项:

[1] 在上下文, "魔数" 和用来指明文件类型的 魔数(magic numbers)有完全不同的意思.

第33章 杂项

==========

```
   Nobody really knows what the Bourne shell's grammar is. Even
   examination of the source code is little help.
          Tom Duff
```

33.1. 交互式和非交互式的shells和脚本

交互式的shell在 tty终端从用户的输入中读取命令. 另一方面, shell能在启动时读取启动文件,显示一个提示符并默认激活作业控制. 用户能交互地使用shell.

运行脚本的shell一般都是非交互的shell. 但脚本仍然可以存取它拥有的终端. 脚本里甚至可以仿效成可交互的shell.

```
  1 #!/bin/bash
  2 MY_PROMPT='$ '
  3 while :
  4 do
  5   echo -n "$MY_PROMPT"
  6   read line
  7   eval "$line"
```

```
  8  done
  9
 10 exit 0
 11
 12 # 这个例子脚本, 和上面的解释由
 13 # St é phane Chazelas 提供(再次感谢).
```

让我们考虑一个要求用户交互式输入的脚本,通常用read语句 (参考例子 11-3). 真正的情况可能有些混乱.以现在假设的情况来说,交互式脚本被限制在一个tty设备上,它本身已经是从一个控制终端或一个中被用户调用的.

初始化和启动脚本不必是非交互式的,因为它们必须不需要人为地干预地运行.许多管理和系统维护脚本也同样是非交互式的.不多变的重复性的任务可以自动地由非交互式脚本完成.

非交互式的脚本可以在后台运行,但交互脚本在后台运行则会被挂起,等待永远不会到达的输入.解决这个难点的办法可以写预料这种情况的脚本或是内嵌here document 的脚本来获取脚本期望的输入,这样就可作为后台任务运行了.在最简单的情况,重定向一个文件给一个read语句提供输入(read variable <file). 这就可能适应交互和非交互的工作环境下都能达成脚本运行的目的.

如果脚本需要测试当前是否运行在交互shell中,一个简单的办法是找一下是否有提示符变量,即$PS1是否设置了. (如果脚本需要用户输入数据,则脚本会显示一个提示符.)

```
 1 if [ -z $PS1 ] # 没有提示符?
 2 then
 3  # 非交互式
 4  ...
 5 else
 6  # 交互式
 7  ...
 8 fi
```

另一个办法是脚本可以测试是否在变量$-中出现了选项"i".

```
 1 case $- in
 2 *i*)    # 交互式 shell
 3 ;;
 4 *)      # 非交互式 shell
 5 ;;
 6 # (Courtesy of "UNIX F.A.Q.," 1993)
```

注意: 脚本可以使用-i选项强制在交互式模式下运行或脚本头用#!/bin/bash -i. 注意这样可能会引起脚本古怪的行为或当没有错误出现时也会显示错误信息.

33.2. Shell 包装

包装脚本是指嵌有一个系统命令和程序的脚本,也保存了一组传给该命令的参数. [1] 包装脚本使原本很复杂的命令行简单化. 这对 sed 和 awk 特别有用.

sed 和 awk 命令一般从命令行上以 sed -e 'commands' 和 awk 'commands' 来调用. 把sed和awk的命令嵌入到Bash脚本里使调用变得更简单, 并且也可多次使用. 也可以综合地利用 sed 和 awk 的功能, 例如管道(piping)连接sed 命令的输出到awk命令中. 保存为可执行的文件, 你可以用脚本编写的或修改的调用格式多次的调用它, 而不必在命令行上重复键入复杂的命令行.

Example 33-1 shell 包装

```
################################Start Script#########################################
 1 #!/bin/bash
 2
```

```
 3 # 这是一个把文件中的空行删除的简单脚本.
 4 # 没有参数检查.
 5 #
 6 # 你可能想增加类似下面的代码:
 7 #
 8 # E_NOARGS=65
 9 # if [ -z "$1" ]
10 # then
11 #  echo "Usage: `basename $0` target-file"
12 #  exit $E_NOARGS
13 # fi
14
15
16 # 就像从命令行调用下面的命令：
17 #    sed -e '/^$/d' filename
18 #
19
20 sed -e /^$/d "$1"
21 #  The '-e' 意味着后面跟的是编辑命令 (这是可选的).
22 #  '^' 匹配行的开头, '$' 则是行的结尾.
23 #  这个表达式匹配行首和行尾之间什么也没有的行,
24 #+ 即空白行.
25 #  'd'是删除命令.
26
27 #  引号引起命令行参数就允许在文件名中使用空白字符和特殊字符
28 #
29
30 #  注意这个脚本不能真正的修改目标文件.
31 #  如果你需要保存修改,就要重定向到某个输出文件里.
32
33 exit 0
```

##################################End Script###

Example 33-2 稍微复杂一些的shell包装

##################################Start Script###

```
 1 #!/bin/bash
 2
 3 #  "subst", 把一个文件中的一个模式替换成一个模式的脚本
 4 #
 5 #  例如, "subst Smith Jones letter.txt".
 6
 7 ARGS=3        # 脚本要求三个参数.
 8 E_BADARGS=65   # 传递了错误的参数个数给脚本.
 9
10 if [ $# -ne "$ARGS" ]
11 # 测试脚本参数的个数 (这是好办法).
12 then
13   echo "Usage: `basename $0` old-pattern new-pattern filename"
14   exit $E_BADARGS
```

```
15 fi
16
17 old_pattern=$1
18 new_pattern=$2
19
20 if [ -f "$3" ]
21 then
22     file_name=$3
23 else
24     echo "File \"$3\" does not exist."
25     exit $E_BADARGS
26 fi
27
28
29 #  这儿是实现功能的代码.
30
31 # ------------------------------------------------
32 sed -e "s/$old_pattern/$new_pattern/g" $file_name
33 # ------------------------------------------------
34
35 #  's' 在sed命令里表示替换,
36 #+ /pattern/表示匹配地址.
37 #  The "g"也叫全局标志使sed会在每一行有$old_pattern模式出现的所有地方替换,
38 #+ 而不只是匹配第一个出现的地方.
39 #  参考'sed'的有关书籍了解更深入的解释.
40
41 exit 0    # 脚本成功调用会返回 0.
```

##################################End Script##

Example 33-3 写到日志文件的shell包装

##################################Start Script##

```
 1 #!/bin/bash
 2 #  普通的shell包装,执行一个操作并记录在日志里
 3 #
 4
 5 # 需要设置下面的两个变量.
 6 OPERATION=
 7 #         可以是一个复杂的命令链,
 8 #+        例如awk脚本或是管道 . . .
 9 LOGFILE=
10 #         不管怎么样,命令行参数还是要提供给操作的.
11
12
13 OPTIONS="$@"
14
15
16 # 记录操作.
17 echo "`date` + `whoami` + $OPERATION "$@"" >> $LOGFILE
18 # 现在, 执行操作.
```

```
19 exec $OPERATION "$@"
20
21 # 在操作之前记录日志是必须的.
22 # 为什么?
```

################################End Script###

Example 33-4 包装awk的脚本

################################Start Script###

```
 1 #!/bin/bash
 2 # pr-ascii.sh: 打印 ASCII 码的字符表.
 3
 4 START=33   # 可打印的 ASCII 字符的范围 (十进制).
 5 END=125
 6
 7 echo " Decimal   Hex     Character"   # 表头.
 8 echo " -------   ---     ---------"
 9
10 for ((i=START; i<=END; i++))
11 do
12   echo $i | awk '{printf("  %3d       %2x         %c\n", $1, $1, $1)}'
13 # 在这个上下文,不会运行Bash的内建printf命令:
14 #     printf "%c" "$i"
15 done
16
17 exit 0
18
19
20 #  Decimal   Hex     Character
21 #  -------   ---     ---------
22 #    33       21        !
23 #    34       22        "
24 #    35       23        #
25 #    36       24        $
26 #
27 #    . . .
28 #
29 #   122       7a        z
30 #   123       7b        {
31 #   124       7c        |
32 #   125       7d        }
33
34
35 #  把脚本的输出重定向到一个文件或是管道给more命令来查看:
36 #+   sh pr-asc.sh | more
```

################################End Script###

Example 33-5 另一个包装awk的脚本

################################Start Script###

```
 1 #!/bin/bash
 2
```

```
3 # 给目标文件增加一列由数字指定的列.
4
5 ARGS=2
6 E_WRONGARGS=65
7
8 if [ $# -ne "$ARGS" ] # 检查命令行参数个数是否正确.
9 then
10   echo "Usage: `basename $0` filename column-number"
11   exit $E_WRONGARGS
12 fi
13
14 filename=$1
15 column_number=$2
16
17 #  传递shell变量给脚本的awk部分需要一点技巧.
18 #  方法之一是在awk脚本中使用强引用来引起bash脚本的变量
19 #
20 #    $'$BASH_SCRIPT_VAR'
21 #     ^              ^
22 #  这个方法在下面的内嵌的awk脚本中出现.
23 #  参考awk文档了解更多的细节.
24
25 # 多行的awk脚本调用格式为:  awk ' ..... '
26
27
28 # 开始 awk 脚本.
29 # ------------------------------
30 awk '
31
32 { total += $'"${column_number}"'
33 }
34 END {
35     print total
36 }
37
38 ' "$filename"
39 # ------------------------------
40 # awk脚本结束.
41
42
43 #   把shell变量传递给awk变量可能是不安全的,
44 #+  因此Stephane Chazelas提出了下面另外一种方法:
45 #   ---------------------------------------
46 #   awk -v column_number="$column_number" '
47 #   { total += $column_number
48 #   }
49 #   END {
50 #       print total
```

```
51 #   }' "$filename"
52 #   ---------------------------------------
53
54
55 exit 0
```

################################End Script###

对于要实现这些功能而只用一种多合一的瑞士军刀应该用Perl. Perl兼有sed和awk的能力, 并且具有C的一个很大的子集. 它是标准的并支持面向对象编程的方方面面,甚至是很琐碎的东西. 短的Perl脚本也可以嵌入到shell脚本中去,以至于有些人宣称Perl能够完全地代替shell编程(本文作者对此持怀疑态度).

Example 33-6 把Perl嵌入Bash脚本

################################Start Script#######################################

```
 1 #!/bin/bash
 2
 3 # Shell命令可以包含 Perl 脚本.
 4 echo "This precedes the embedded Perl script within \"$0\"."
 5 echo "============================================================="
 6
 7 perl -e 'print "This is an embedded Perl script.\n";'
 8 # 像sed脚本, Perl 也使用"-e"选项.
 9
10 echo "============================================================="
11 echo "However, the script may also contain shell and system commands."
12
13 exit 0
```

################################End Script###

把Bash脚本和Perl脚本放在同一个文件是可能的. 依赖于脚本如何被调用, 要么是Bash部分被执行,要么是Perl部分被执行.

Example 33-7 Bash 和 Perl 脚本联合使用

################################Start Script#######################################

```
 1 #!/bin/bash
 2 # bashandperl.sh
 3
 4 echo "Greetings from the Bash part of the script."
 5 # 下面可以有更多的Bash命令.
 6
 7 exit 0
 8 # 脚本的Bash部分结束.
 9
10 # ====================================================
11
12 #!/usr/bin/perl
13 # 脚本的这个部分必须用-x选项来调用.
14
15 print "Greetings from the Perl part of the script.\n";
16 # 下面可以有更多的Perl命令.
17
18 # 脚本的Perl部分结束.
```

```
################################End Script#########################################
 bash$ bash bashandperl.sh
 Greetings from the Bash part of the script.

 bash$ perl -x bashandperl.sh
 Greetings from the Perl part of the script.
```

注意事项:

[1] 事实上,相当数量的Linux软件工具包是shell包装脚本. 例如/usr/bin/pdf2ps, /usr/bin/batch, 和 /usr/X11R6/bin/xmkmf.

33.3. 测试和比较: 另一种方法

对于测试,[[ ]]结构可能比[ ]更合适.同样地,算术比较可能用(( ))结构更有用.

```
  1 a=8
  2
  3 # 下面所有的比较是等价的.
  4 test "$a" -lt 16 && echo "yes, $a < 16"        # "与列表"
  5 /bin/test "$a" -lt 16 && echo "yes, $a < 16"
  6 [ "$a" -lt 16 ] && echo "yes, $a < 16"
  7 [[ $a -lt 16 ]] && echo "yes, $a < 16"         # 在[[ ]]和(( ))中不必用引号引起变量
  8 (( a < 16 )) && echo "yes, $a < 16"            #
  9
 10 city="New York"
 11 # 同样,下面的所有比较都是等价的.
 12 test "$city" \< Paris && echo "Yes, Paris is greater than $city"  # 产生 ASCII 顺序.
 13 /bin/test "$city" \< Paris && echo "Yes, Paris is greater than $city"
 14 [ "$city" \< Paris ] && echo "Yes, Paris is greater than $city"
 15 [[ $city < Paris ]] && echo "Yes, Paris is greater than $city"    # 不需要用引号引起$city.
 16
 17 # 多谢, S.C.
```

33.4. 递归

脚本是否能 递归地 调用自己本身? 当然可以.

Example 33-8 递归调用自己本身的(无用)脚本

```
################################Start Script#######################################
 1 #!/bin/bash
 2 # recurse.sh
 3
 4 #  脚本能否递归地调用自己?
 5 #  是的, 但这有什么实际的用处吗?
 6 #  (看下面的.)
 7
 8 RANGE=10
 9 MAXVAL=9
10
11 i=$RANDOM
12 let "i %= $RANGE"  # 产生一个从 0 到 $RANGE - 1 之间的随机数.
13
14 if [ "$i" -lt "$MAXVAL" ]
```

```
15 then
16   echo "i = $i"
17   ./$0           # 脚本递归地调用再生成一个和自己一样的实例.
18 fi              # 每个子脚本做的事都一样,
19                 #+ 直到产生的变量 $i 和变量 $MAXVAL 相等.
20
21 # 用"while"循环代替"if/then"测试会引起错误.
22 # 解释为什么会这样.
23
24 exit 0
25
26 # 注:
27 # ----
28 # 脚本要正确地工作必须有执行权限.
29 # 这是指用"sh"命令来调用这个脚本而没有设置正确权限导致的问题.
30 # 请解释原因.
```

################################End Script###

Example 33-9 递归调用自己本身的(有用)脚本

################################Start Script###

```
 1 #!/bin/bash
 2 # pb.sh: 电话本(phone book)
 3
 4 # 由Rick Boivie编写,已得到使用许可.
 5 # 由ABS文档作者修改.
 6
 7 MINARGS=1     # 脚本需要至少一个参数.
 8 DATAFILE=./phonebook
 9             # 在当前目录下名为"phonebook"的数据文件必须存在
10             #
11 PROGNAME=$0
12 E_NOARGS=70   # 没有参数的错误值.
13
14 if [ $# -lt $MINARGS ]; then
15       echo "Usage: "$PROGNAME" data"
16       exit $E_NOARGS
17 fi
18
19
20 if [ $# -eq $MINARGS ]; then
21       grep $1 "$DATAFILE"
22       # 如果$DATAFILE文件不存在,'grep' 会打印一个错误信息.
23 else
24       ( shift; "$PROGNAME" $* ) | grep $1
25       # 脚本递归调用本身.
26 fi
27
28 exit 0       # 脚本在这儿退出.
29             # 因此Therefore, 从这行开始可以写没有#开头的的注释行
```

```
30       #
31
32 # ---------------------------------------------------------------------
33 "phonebook"文件的例子:
34
35 John Doe        1555 Main St., Baltimore, MD 21228         (410) 222-3333
36 Mary Moe        9899 Jones Blvd., Warren, NH 03787         (603) 898-3232
37 Richard Roe     856 E. 7th St., New York, NY 10009         (212) 333-4567
38 Sam Roe         956 E. 8th St., New York, NY 10009         (212) 444-5678
39 Zoe Zenobia     4481 N. Baker St., San Francisco, SF 94338  (415) 501-1631
40 # ---------------------------------------------------------------------
41
42 $bash pb.sh Roe
43 Richard Roe     856 E. 7th St., New York, NY 10009         (212) 333-4567
44 Sam Roe         956 E. 8th St., New York, NY 10009         (212) 444-5678
45
46 $bash pb.sh Roe Sam
47 Sam Roe         956 E. 8th St., New York, NY 10009         (212) 444-5678
48
49 #  当超过一个参数传给这个脚本时,
50 #+ 它只打印包含所有参数的行.
```

##################################End Script###

Example 33-10 另一个递归调用自己本身的(有用)脚本

##################################Start Script###

```
 1 #!/bin/bash
 2 # usrmnt.sh, 由Anthony Richardson编写
 3 # 得到允许在此使用.
 4
 5 # usage:       usrmnt.sh
 6 # 描述: 挂载设备, 调用者必须列在/etc/sudoers文件的MNTUSERS组里
 7 #
 8
 9 # ------------------------------------------------------------
10 #  这是一个用户挂载设备的脚本,它用sudo来调用自己.
11 #  只有拥有合适权限的用户才能用
12
13 #   usermount /dev/fd0 /mnt/floppy
14
15 # 来代替
16
17 #   sudo usermount /dev/fd0 /mnt/floppy
18
19 #  我使用相同的技术来处理我所有的sudo脚本,
20 #+ 因为我觉得它很方便.
21 # ------------------------------------------------------------
22
23 #  如果 SUDO_COMMAND 变量没有设置,我们不能通过sudo来运行脚本本身.
24 #+ 传递用户的真实ID和组ID . . .
```

```
25
26 if [ -z "$SUDO_COMMAND" ]
27 then
28    mntusr=$(id -u) grpusr=$(id -g) sudo $0 $*
29    exit 0
30 fi
31
32 # 如果我们以sudo来调用运行,就会运行这儿.
33 /bin/mount $* -o uid=$mntusr,gid=$grpusr
34
35 exit 0
36
37 # 附注 (由脚本作者加注):
38 # ------------------------------------------------
39
40 # 1) Linux允许在/etc/fstab文件中使用"users"选项
41 #    以使任何用户能挂载可移动的介质.
42 #    但是, 在一个服务器上,
43 #    我只想有限的几个用户可以存取可移动介质.
44 #    我发现使用sudo可以有更多的控制.
45
46 # 2) 我也发现sudo能通过组更方便地达成目的.
47 #
48
49 # 3) 这个方法使给予任何想给合适权限的人使用mount命令
50 #    所以要小心使用.
51 #    你也可以开发类似的脚本mntfloppy, mntcdrom,和 mntsamba来使mount命令得到更好的控制
52 #
53 #
54 #
```

################################End Script###

注意: 过多层次的递归调用会耗尽脚本的堆栈空间,会引起段错误.

33.5. 彩色脚本

ANSI [1] 定义了屏幕属性的转义序列集合,例如粗体文本,背景和前景颜色. DOS批处理文件(batch files) 一般使用ANSI的转义代码来控制色彩输出,Bash脚本也是这么做的.

Example 33-11 一个 "彩色的" 地址资料库

################################Start Script###

```
 1 #!/bin/bash
 2 # ex30a.sh: ex30.sh的"彩色" 版本.
 3 #           没有加工处理的地址资料库
 4
 5
 6 clear                          # 清除屏幕.
 7
 8 echo -n "          "
 9 echo -e '\E[37;44m'"\033[1mContact List\033[0m"
10                                # 白色为前景色,蓝色为背景色
```

```
11 echo; echo
12 echo -e "\033[1mChoose one of the following persons:\033[0m"
13                                  # 粗体
14 tput sgr0
15 echo "(Enter only the first letter of name.)"
16 echo
17 echo -en '\E[47;34m'"\033[1mE\033[0m"   # 蓝色
18 tput sgr0                            # 把色彩设置为"常规"
19 echo "vans, Roland"                   # "[E]vans, Roland"
20 echo -en '\E[47;35m'"\033[1mJ\033[0m"   # 红紫色
21 tput sgr0
22 echo "ones, Mildred"
23 echo -en '\E[47;32m'"\033[1mS\033[0m"   # 绿色
24 tput sgr0
25 echo "mith, Julie"
26 echo -en '\E[47;31m'"\033[1mZ\033[0m"   # 红色
27 tput sgr0
28 echo "ane, Morris"
29 echo
30
31 read person
32
33 case "$person" in
34 # 注意变量被引起来了.
35
36   "E" | "e" )
37   # 接受大小写的输入.
38   echo
39   echo "Roland Evans"
40   echo "4321 Floppy Dr."
41   echo "Hardscrabble, CO 80753"
42   echo "(303) 734-9874"
43   echo "(303) 734-9892 fax"
44   echo "revans@zzy.net"
45   echo "Business partner & old friend"
46   ;;
47
48   "J" | "j" )
49   echo
50   echo "Mildred Jones"
51   echo "249 E. 7th St., Apt. 19"
52   echo "New York, NY 10009"
53   echo "(212) 533-2814"
54   echo "(212) 533-9972 fax"
55   echo "milliej@loisaida.com"
56   echo "Girlfriend"
57   echo "Birthday: Feb. 11"
58   ;;
```

```
59
60 # 稍后为 Smith 和 Zane 增加信息.
61
62         * )
63   # 默认选项Default option.
64   # 空的输入(直接按了回车) 也会匹配这儿.
65   echo
66   echo "Not yet in database."
67   ;;
68
69 esac
70
71 tput sgr0                       # 把色彩重设为"常规".
72
73 echo
74
75 exit 0
```

##################################End Script###

Example 33-12 画盒子

##################################Start Script###

```
 1 #!/bin/bash
 2 # Draw-box.sh: 用ASCII字符画一个盒子.
 3
 4 # Stefano Palmeri编写,文档作者作了少量编辑.
 5 # 征得作者同意在本书使用.
 6
 7
 8 ###############################################################################
 9 ###  draw_box 函数的注释  ###
10
11 #  "draw_box" 函数使用户可以在终端上画一个盒子.
12 #
13 #
14 #  用法: draw_box ROW COLUMN HEIGHT WIDTH [COLOR]
15 #  ROW 和 COLUMN 定位要画的盒子的左上角.
16 #
17 #  ROW 和 COLUMN 必须要大于0且小于目前终端的尺寸.
18 #
19 #  HEIGHT 是盒子的行数,必须 > 0.
20 #  HEIGHT + ROW 必须 <= 终端的高度.
21 #  WIDTH 是盒子的列数,必须 > 0.
22 #  WIDTH + COLUMN 必须 <= 终端的宽度.
23 #
24 #  例如: 如果你当前终端的尺寸是 20x80,
25 #  draw_box 2 3 10 45 是合法的
26 #  draw_box 2 3 19 45 的 HEIGHT 值是错的 (19+2 > 20)
27 #  draw_box 2 3 18 78 的 WIDTH 值是错的 (78+3 > 80)
28 #
```

```
29 #  COLOR 是盒子边框的颜色.
30 #  它是第5个参数,并且它是可选的.
31 #  0=黑色 1=红色 2=绿色 3=棕褐色 4=蓝色 5=紫色 6=青色 7=白色.
32 #  如果你传给这个函数错的参数,
33 #+ 它就会以代码65退出,
34 #+ 没有其他的信息打印到标准出错上.
35 #
36 #  在画盒子之前要清屏.
37 #  函数内不包含有清屏命令.
38 #  这使用户可以画多个盒子,甚至叠接多个盒子.
39
40 ###  draw_box 函数注释结束  ###
41 ################################################################################
42
43 draw_box(){
44
45 #=============#
46 HORZ="-"
47 VERT="|"
48 CORNER_CHAR="+"
49
50 MINARGS=4
51 E_BADARGS=65
52 #=============#
53
54
55 if [ $# -lt "$MINARGS" ]; then              # 如果参数小于4,退出.
56    exit $E_BADARGS
57 fi
58
59 # 搜寻参数中的非数字的字符.
60 # 能用其他更好的办法吗 (留给读者的练习?).
61 if echo $@ | tr -d [:blank:] | tr -d [:digit:] | grep . &> /dev/null; then
62   exit $E_BADARGS
63 fi
64
65 BOX_HEIGHT=`expr $3 - 1`   #  -1 是需要的,因为因为边角的"+"是高和宽共有的部分.
66 BOX_WIDTH=`expr $4 - 1`    #
67 T_ROWS=`tput lines`       #  定义当前终端长和宽的尺寸,
68 T_COLS=`tput cols`        #
69
70 if [ $1 -lt 1 ] || [ $1 -gt $T_ROWS ]; then    #  如果参数是数字就开始检查有效性.
71    exit $E_BADARGS                          #
72 fi
73 if [ $2 -lt 1 ] || [ $2 -gt $T_COLS ]; then
74    exit $E_BADARGS
75 fi
76 if [ `expr $1 + $BOX_HEIGHT + 1` -gt $T_ROWS ]; then
```

```
77    exit $E_BADARGS
78 fi
79 if [ `expr $2 + $BOX_WIDTH + 1` -gt $T_COLS ]; then
80    exit $E_BADARGS
81 fi
82 if [ $3 -lt 1 ] || [ $4 -lt 1 ]; then
83    exit $E_BADARGS
84 fi                              # 参数检查完毕.
85
86 plot_char(){                    # 函数内的函数.
87    echo -e "\E[${1};${2}H"$3
88 }
89
90 echo -ne "\E[3${5}m"            # 如果传递了盒子边框颜色参数,则设置它.
91
92 # start drawing the box
93
94 count=1                                 #  用plot_char函数画垂直线
95 for (( r=$1; count<=$BOX_HEIGHT; r++)); do      #
96   plot_char $r $2 $VERT
97   let count=count+1
98 done
99
100 count=1
101 c=`expr $2 + $BOX_WIDTH`
102 for (( r=$1; count<=$BOX_HEIGHT; r++)); do
103   plot_char $r $c $VERT
104   let count=count+1
105 done
106
107 count=1                                 #  用plot_char函数画水平线
108 for (( c=$2; count<=$BOX_WIDTH; c++)); do      #
109   plot_char $1 $c $HORZ
110   let count=count+1
111 done
112
113 count=1
114 r=`expr $1 + $BOX_HEIGHT`
115 for (( c=$2; count<=$BOX_WIDTH; c++)); do
116   plot_char $r $c $HORZ
117   let count=count+1
118 done
119
120 plot_char $1 $2 $CORNER_CHAR                # 画盒子的角.
121 plot_char $1 `expr $2 + $BOX_WIDTH` +
122 plot_char `expr $1 + $BOX_HEIGHT` $2 +
123 plot_char `expr $1 + $BOX_HEIGHT` `expr $2 + $BOX_WIDTH` +
124
```

```
125 echo -ne "\E[0m"           #  恢复最初的颜色.
126
127 P_ROWS=`expr $T_ROWS - 1`   #  在终端的底部打印提示符.
128
129 echo -e "\E[${P_ROWS};1H"
130 }
131
132
133 # 现在, 让我们来画一个盒子.
134 clear                    # 清屏.
135 R=2     # 行
136 C=3     # 列
137 H=10    # 高
138 W=45    # 宽
139 col=1   # 颜色(红)
140 draw_box $R $C $H $W $col   # 画盒子.
141
142 exit 0
143
144 # 练习:
145 # --------
146 # 增加可以在盒子里打印文本的选项
```

##################################End Script##

最简单也可能是最有用的ANSI转义序列是加粗文本, \033[1m ... \033[0m. \033 触发转义序列, 而 "[1" 启用加粗属性, 而"[0" 表示切换回禁用加粗状态. "m"则表示终止一个转义序列.

```
 bash$ echo -e "\033[1mThis is bold text.\033[0m"
```

一种相似的转义序列可切换下划线效果 (在 rxvt 和 aterm 上).

```
 bash$ echo -e "\033[4mThis is underlined text.\033[0m"
```

注意: echo使用-e选项可以启用转义序列.

其他的转义序列可用于更改文本或/和背景色彩.

```
 bash$ echo -e '\E[34;47mThis prints in blue.'; tput sgr0

 bash$ echo -e '\E[33;44m'"yellow text on blue background"; tput sgr0

 bash$ echo -e '\E[1;33;44m'"BOLD yellow text on blue background"; tput sgr0
```

注意: 通常为淡色的前景色文本设置粗体效果是较好的.

tput sgr0 把终端设置恢复为原样. 如果省略这一句会使后续在该终端的输出仍为蓝色.

注意: 因为tput sgr0 在某些环境下不能恢复终端设置, echo -ne \E[0m 会是更好的选择.

可以在有色的背景上用下面的模板写有色彩的文本.

```
echo -e '\E[COLOR1;COLOR2mSome text goes here.'
```

"\E[" 开始转义序列. 分号分隔的数值"COLOR1" 和 "COLOR2" 指定前景色和背景色, 数值和色彩的对应参见下面的表格. (数值的顺序不是有关系的,因为前景色和背景色数值都落在不重叠的范围里.) "m"终止该转义序列, 然后文本以结束的转义指定的属性显示.

也要注意到用单引号引用了echo -e后面的余下命令序列.

下表的数值是在 rxvt 终端运行的结果. 具体效果可能在其他的各种终端上不一样.

table 33-1. 转义序列中数值和彩色的对应

==========
色彩	前景色	背景色
黑	30	40
红	31	41
绿	32	42
黄	33	43
蓝	34	44
洋红	35	45
青	36	46
白	37	47
==========
Example 33-13 显示彩色文本
################################Start Script###

```
 1 #!/bin/bash
 2 # color-echo.sh: 用彩色来显示文本.
 3
 4 # 依照需要修改这个脚本.
 5 # 这比手写彩色的代码更容易一些.
 6
 7 black='\E[30;47m'
 8 red='\E[31;47m'
 9 green='\E[32;47m'
10 yellow='\E[33;47m'
11 blue='\E[34;47m'
12 magenta='\E[35;47m'
13 cyan='\E[36;47m'
14 white='\E[37;47m'
15
16
17 alias Reset="tput sgr0"      #  把文本属性重设回原来没有清屏前的
18                             #
19
20
21 cecho ()                     # Color-echo.
22                             # 参数 $1 = 要显示的信息
23                             # 参数 $2 = 颜色
24 {
25 local default_msg="No message passed."
26                             # 不是非要一个本地变量.
27
28 message=${1:-$default_msg}   # 默认的信息.
29 color=${2:-$black}           # 如果没有指定,默认使用黑色.
30
31   echo -e "$color"
32   echo "$message"
33   Reset                     # 重设文本属性.
34
35   return
```

```
36 }
37
38
39 # 现在,让我们试试.
40 # ----------------------------------------------------
41 cecho "Feeling blue..." $blue
42 cecho "Magenta looks more like purple." $magenta
43 cecho "Green with envy." $green
44 cecho "Seeing red?" $red
45 cecho "Cyan, more familiarly known as aqua." $cyan
46 cecho "No color passed (defaults to black)."
47       # 缺失 $color (色彩)参数.
48 cecho "\"Empty\" color passed (defaults to black)." ""
49       # 空的 $color (色彩)参数.
50 cecho
51       # $message(信息) 和 $color (色彩)参数都缺失.
52 cecho "" ""
53       # 空的 $message (信息)和 $color (色彩)参数.
54 # ----------------------------------------------------
55
56 echo
57
58 exit 0
59
60 # 练习:
61 # ---------
62 # 1) 为'cecho ()'函数增加粗体的效果.
63 # 2) 增加可选的彩色背景.
```

################################End Script###

Example 33-14 "赛马" 游戏

################################Start Script###

```
 1 #!/bin/bash
 2 # horserace.sh: 非常简单的赛马模拟.
 3 # 作者: Stefano Palmeri
 4 # 已取得使用许可.
 5
 6 ######################################################################
 7 #  脚本目的:
 8 #  使用转义字符和终端颜色.
 9 #
10 #  练习:
11 #  编辑脚本使其更具有随机性,
12 #+ 设置一个假的赌场 . . .
13 #  嗯 . . . 嗯 . . . 这个开始使我想起了一部电影 . . .
14 #
15 #  脚本给每匹马一个随机的障碍.
16 #  不均等会以障碍来计算
17 #+ 并且用一种欧洲风格表达出来.
```

```
18 #  例如: 机率(odds)=3.75 意味着如果你押1美元赢,
19 #+ 你可以赢得3.75美元.
20 #
21 #  脚本已经在GNU/Linux操作系统上测试过 OS,
22 #+ 测试终端有xterm 和 rxvt, 及 konsole.
23 #  测试机器有AMD 900 MHz 的处理器,
24 #+ 平均比赛时间是75秒.
25 #  在更快的计算机上比赛时间应该会更低.
26 #  所以, 如果你想有更多的悬念,重设USLEEP_ARG 变量的值.
27 #
28 #  由Stefano Palmeri编写.
29 ################################################################
30
31 E_RUNERR=65
32
33 # 检查 md5sum 和 bc 是不是安装了.
34 if ! which bc &> /dev/null; then
35    echo bc is not installed.
36    echo "Can\'t run . . . "
37    exit $E_RUNERR
38 fi
39 if ! which md5sum &> /dev/null; then
40    echo md5sum is not installed.
41    echo "Can\'t run . . . "
42    exit $E_RUNERR
43 fi
44
45 #  更改下面的变量值可以使脚本执行的更慢.
46 #  它会作为usleep的参数 (man usleep)
47 #+ 并且它的单位是微秒 (500000微秒 = 半秒).
48 USLEEP_ARG=0
49
50 #  如果脚本接收到ctrl-c中断,清除临时目录, 恢复终端光标和颜色
51 #
52 trap 'echo -en "\E[?25h"; echo -en "\E[0m"; stty echo;\
53 tput cup 20 0; rm -fr  $HORSE_RACE_TMP_DIR'  TERM EXIT
54 #  参考调试的章节了解'trap'的更多解释
55
56 # 给脚本设置一个唯一(实际不是绝对唯一的)的临时目录名.
57 HORSE_RACE_TMP_DIR=$HOME/.horserace-`date +%s`-`head -c10 /dev/urandom | md5sum | head -c30`
58
59 # 创建临时目录,并切换到该目录下.
60 mkdir $HORSE_RACE_TMP_DIR
61 cd $HORSE_RACE_TMP_DIR
62
63
64 #  这个函数把光标移动到行为 $1 列为 $2 然后打印 $3.
65 #  例如: "move_and_echo 5 10 linux" 等同于
```

```
66 #+ "tput cup 4 9; echo linux", 但是用一个命令代替了两个.
67 #  注: "tput cup" 表示在终端左上角的 0 0 位置,
68 #+ echo 是在终端的左上角的 1 1 位置.
69 move_and_echo() {
70           echo -ne "\E[${1};${2}H""$3"
71 }
72
73 # 产生1-9之间伪随机数的函数.
74 random_1_9 () {
75                head -c10 /dev/urandom | md5sum | tr -d [a-z] | tr -d 0 | cut -c1
76 }
77
78 #  画马时模拟运动的两个函数.
79 draw_horse_one() {
80               echo -n " "//$MOVE_HORSE//
81 }
82 draw_horse_two(){
83              echo -n " "\\\\$MOVE_HORSE\\\\
84 }
85
86
87 # 取得当前的终端尺寸.
88 N_COLS=`tput cols`
89 N_LINES=`tput lines`
90
91 # 至少需要 20-行 X 80-列 的终端尺寸. 检查一下.
92 if [ $N_COLS -lt 80 ] || [ $N_LINES -lt 20 ]; then
93    echo "`basename $0` needs a 80-cols X 20-lines terminal."
94    echo "Your terminal is ${N_COLS}-cols X ${N_LINES}-lines."
95    exit $E_RUNERR
96 fi
97
98
99 # 开始画赛场.
100
101 # 需要一个80个字符的字符串,看下面的.
102 BLANK80=`seq -s "" 100 | head -c80`
103
104 clear
105
106 # 把前景和背景颜色设置成白色的.
107 echo -ne '\E[37;47m'
108
109 # 把光标移到终端的左上角.
110 tput cup 0 0
111
112 # 画六条白线.
113 for n in `seq 5`; do
```

```
114     echo $BLANK80       # 线是用80个字符组成的字符串.
115 done
116
117 # 把前景色设置成黑色.
118 echo -ne '\E[30m'
119
120 move_and_echo 3 1 "START  1"
121 move_and_echo 3 75 FINISH
122 move_and_echo 1 5 "|"
123 move_and_echo 1 80 "|"
124 move_and_echo 2 5 "|"
125 move_and_echo 2 80 "|"
126 move_and_echo 4 5 "|  2"
127 move_and_echo 4 80 "|"
128 move_and_echo 5 5 "V  3"
129 move_and_echo 5 80 "V"
130
131 # 把前景色设置成红色.
132 echo -ne '\E[31m'
133
134 # 一些ASCII艺术.
135 move_and_echo 1 8 "..@@@..@@@@@...@@@@@.@...@..@@@@..."
136 move_and_echo 2 8 ".@...@...@.......@...@...@.@......."
137 move_and_echo 3 8 ".@@@@@...@.......@...@@@@@.@@@@...."
138 move_and_echo 4 8 ".@...@...@.......@...@...@.@......."
139 move_and_echo 5 8 ".@...@...@.......@...@...@..@@@@..."
140 move_and_echo 1 43 "@@@@...@@@...@@@@..@@@@..@@@@."
141 move_and_echo 2 43 "@...@.@...@.@.....@.....@....."
142 move_and_echo 3 43 "@@@@..@@@@@.@.....@@@@...@@@.."
143 move_and_echo 4 43 "@..@..@...@.@.....@.........@."
144 move_and_echo 5 43 "@...@.@...@..@@@@..@@@@.@@@@.."
145
146
147 # 把前景和背景颜色设为绿色.
148 echo -ne '\E[32;42m'
149
150 # 画11行绿线.
151 tput cup 5 0
152 for n in `seq 11`; do
153     echo $BLANK80
154 done
155
156 # 把前景色设为黑色.
157 echo -ne '\E[30m'
158 tput cup 5 0
159
160 # 画栅栏.
161 echo "++++++++++++++++++++++++++++++++++++++\
```

```
162 ++++++++++++++++++++++++++++++++++++++++++"
163
164 tput cup 15 0
165 echo "++++++++++++++++++++++++++++++++++++\
166 ++++++++++++++++++++++++++++++++++++++++++"
167
168 # 把前景和背景色设回白色.
169 echo -ne '\E[37;47m'
170
171 # 画3条白线.
172 for n in `seq 3`; do
173     echo $BLANK80
174 done
175
176 # 把前景色设为黑色.
177 echo -ne '\E[30m'
178
179 # 创建9个文件来保存障碍物.
180 for n in `seq 10 7 68`; do
181     touch $n
182 done
183
184 # 设置脚本要画的马的类型为第一种类型.
185 HORSE_TYPE=2
186
187 #  为每匹马创建位置文件和机率文件.
188 #+ 在这些文件里保存了该匹马当前的位置,
189 #+ 类型和机率.
190 for HN in `seq 9`; do
191     touch horse_${HN}_position
192     touch odds_${HN}
193     echo \-1 > horse_${HN}_position
194     echo $HORSE_TYPE >>  horse_${HN}_position
195     # 给马定义随机的障碍物.
196      HANDICAP=`random_1_9`
197     # 检查random_1_9函数是否返回了有效值.
198     while ! echo $HANDICAP | grep [1-9] &> /dev/null; do
199           HANDICAP=`random_1_9`
200     done
201     # 给马定义最后的障碍的位置.
202     LHP=`expr $HANDICAP \* 7 + 3`
203     for FILE in `seq 10 7 $LHP`; do
204         echo $HN >> $FILE
205     done
206
207     # 计算机率.
208     case $HANDICAP in
209         1) ODDS=`echo $HANDICAP \* 0.25 + 1.25 | bc`
```

```
210                         echo $ODDS > odds_${HN}
211             ;;
212             2 | 3) ODDS=`echo $HANDICAP \* 0.40 + 1.25 | bc`
213                             echo $ODDS > odds_${HN}
214             ;;
215             4 | 5 | 6) ODDS=`echo $HANDICAP \* 0.55 + 1.25 | bc`
216                                 echo $ODDS > odds_${HN}
217             ;;
218             7 | 8) ODDS=`echo $HANDICAP \* 0.75 + 1.25 | bc`
219                             echo $ODDS > odds_${HN}
220             ;;
221             9) ODDS=`echo $HANDICAP \* 0.90 + 1.25 | bc`
222                             echo $ODDS > odds_${HN}
223     esac
224
225
226 done
227
228
229 # 打印机率.
230 print_odds() {
231 tput cup 6 0
232 echo -ne '\E[30;42m'
233 for HN in `seq 9`; do
234     echo "#$HN odds->" `cat odds_${HN}`
235 done
236 }
237
238 # 在起跑线上画马.
239 draw_horses() {
240 tput cup 6 0
241 echo -ne '\E[30;42m'
242 for HN in `seq 9`; do
243     echo /\\$HN/\\"                         "
244 done
245 }
246
247 print_odds
248
249 echo -ne '\E[47m'
250 # 等待回车按键开始赛马.
251 # 转义序列'\E[?25l'禁显了光标.
252 tput cup 17 0
253 echo -e '\E[?25l'Press [enter] key to start the race...
254 read -s
255
256 #  禁用了终端的常规显示功能.
257 #  这避免了赛跑时不小心按了按键键入显示字符而弄乱了屏幕.
```

```
258 #
259 stty -echo
260
261 # ----------------------------------------------------------
262 # 开始赛跑.
263
264 draw_horses
265 echo -ne '\E[37;47m'
266 move_and_echo 18 1 $BLANK80
267 echo -ne '\E[30m'
268 move_and_echo 18 1 Starting...
269 sleep 1
270
271 # 设置终点线的列数.
272 WINNING_POS=74
273
274 # 记录赛跑开始的时间.
275 START_TIME=`date +%s`
276
277 # COL 是由下面的"while"结构使用的.
278 COL=0
279
280 while [ $COL -lt $WINNING_POS ]; do
281
282         MOVE_HORSE=0
283
284         # 检查random_1_9函数是否返回了有效值.
285         while ! echo $MOVE_HORSE | grep [1-9] &> /dev/null; do
286             MOVE_HORSE=`random_1_9`
287         done
288
289         # 取得随机取得的马的类型和当前位置.
290         HORSE_TYPE=`cat  horse_${MOVE_HORSE}_position | tail -1`
291         COL=$(expr `cat  horse_${MOVE_HORSE}_position | head -1`)
292
293         ADD_POS=1
294         # 检查当前的位置是否是障碍物的位置.
295         if seq 10 7 68 | grep -w $COL &> /dev/null; then
296             if grep -w $MOVE_HORSE $COL &> /dev/null; then
297                 ADD_POS=0
298                 grep -v -w  $MOVE_HORSE $COL > ${COL}_new
299                 rm -f $COL
300                 mv -f ${COL}_new $COL
301                 else ADD_POS=1
302             fi
303         else ADD_POS=1
304         fi
305         COL=`expr $COL + $ADD_POS`
```

```
306         echo $COL >  horse_${MOVE_HORSE}_position  # 保存新位置.
307
308        # 选择要画的马的类型.
309         case $HORSE_TYPE in
310             1) HORSE_TYPE=2; DRAW_HORSE=draw_horse_two
311             ;;
312             2) HORSE_TYPE=1; DRAW_HORSE=draw_horse_one
313         esac
314         echo $HORSE_TYPE >>  horse_${MOVE_HORSE}_position # 保存当前类型.
315
316         # 把前景色设为黑,背景色设为绿.
317         echo -ne '\E[30;42m'
318
319         # 把光标位置移到新的马的位置.
320         tput cup `expr $MOVE_HORSE + 5`  `cat  horse_${MOVE_HORSE}_position | head -1`
321
322         # 画马.
323         $DRAW_HORSE
324          usleep $USLEEP_ARG
325
326          # 当所有的马都越过15行的之后,再次打印机率.
327          touch fieldline15
328          if [ $COL = 15 ]; then
329            echo $MOVE_HORSE >> fieldline15
330          fi
331          if [ `wc -l fieldline15 | cut -f1 -d " "` = 9 ]; then
332             print_odds
333             : > fieldline15
334          fi
335
336         # 取得领头的马.
337         HIGHEST_POS=`cat *position | sort -n | tail -1`
338
339         # 把背景色重设为白色.
340         echo -ne '\E[47m'
341         tput cup 17 0
342         echo -n Current leader: `grep -w $HIGHEST_POS *position | cut -c7`"                          "
343
344 done
345
346 # 取得赛马结束的时间.
347 FINISH_TIME=`date +%s`
348
349 # 背景色设为绿色并且启用闪动的功能.
350 echo -ne '\E[30;42m'
351 echo -en '\E[5m'
352
353 # 使获胜的马闪动.
```

```
354 tput cup `expr $MOVE_HORSE + 5` `cat  horse_${MOVE_HORSE}_position | head -1`
355 $DRAW_HORSE
356
357 # 禁用闪动文本.
358 echo -en '\E[25m'
359
360 # 把前景和背景色设为白色.
361 echo -ne '\E[37;47m'
362 move_and_echo 18 1 $BLANK80
363
364 # 前景色设为黑色.
365 echo -ne '\E[30m'
366
367 # 闪动获胜的马.
368 tput cup 17 0
369 echo -e "\E[5mWINNER: $MOVE_HORSE\E[25m""  Odds: `cat odds_${MOVE_HORSE}`"\
370 "  Race time: `expr $FINISH_TIME - $START_TIME` secs"
371
372 # 恢复光标和最初的颜色.
373 echo -en "\E[?25h"
374 echo -en "\E[0m"
375
376 # 恢复回显功能.
377 stty echo
378
379 # 删除赛跑的临时文件.
380 rm -rf $HORSE_RACE_TMP_DIR
381
382 tput cup 19 0
383
384 exit 0
```

################################End Script###

参考 例子 A-22.

注意: 然而,有一个主要的问题,那就是ANSI 转义序列是不可移植的. 在一些终端运行的
很好的代码可能在另外一些终端上可能运行地很糟糕. 在彩色脚本作者终端上运行的
很好的脚本可能在另外一些终端上就产生不可阅读的输出了. 这给彩色脚本的用处大
大打了个折扣,而很可能使这些技术变成一个暗机关或只是一个玩具而已.

Moshe Jacobson的颜色工具(http://runslinux.net/projects.html#color)能相当容易地使用
ANSI转义序列. 它用清晰和较有逻辑的语法来代替刚才讨论的难用的结构.
Henry/teikedvl 也同样开发了一个软件包来简化彩色脚本的一些操作
(http://scriptechocolor.sourceforge.net/).

注意事项:

[1] 当然,ANSI是American National Standards Institute(美国国家标准组织)的缩
写. 这个令人敬畏的组织建立和维护着许多技术和工业的标准.

33.6. 优化

大多数shell脚本处理不复杂的问题时会有很快的解决办法. 正因为这样,优化脚本速度不是
一个问题. 考虑这样的情况, 一个脚本处理很重要的任务, 虽然它确实运行的很好很正确,但

是处理速度太慢. 用一种可编译的语言重写它可能不是非常好的选择. 最简单的办法是重写使这个脚本效率低下的部分. 这个代码优化的原理是否同样适用于效率低下的shell脚本?
检查脚本中的循环. 反复执行操作的时间消耗增长非常的快. 如果可能, 可以从循环中删除时间消耗的操作.
优先使用内建(builtin)命令而不是系统命令. 内建命令执行起来更快并且一般调用时不会产生新的子shell.
避免不需要的命令, 特别是管道(pipe).

```
1 cat "$file" | grep "$word"
2
3 grep "$word" "$file"
4
5 #  上面的命令行有同样的效果,
6 #+ 但第二个运行的更有效率,因为它不产生新的子进程.
```

cat 命令似乎特别常在脚本中被滥用.
用time和times工具去了解计算花费的时间. 考虑用C甚至是汇编重写关键的消耗时间的部分.
尝试最小化文件I/O. Bash在文件处理上不是特别地有效率, 所以要考虑在脚本中使用更合适地工具来处理, 比如说awk 或 Perl.
采用结构化的思想来写脚本, 使各个模块能够依据需要组织和合并起来.一些适用于高级语言的优化技术也可以用在脚本上 , 但有些技术, 比如说循环优化, 几乎是不相关的. 上面的讨论, 依据经验来判断.
怎样优化减少执行时间的优秀脚本示例, 请参考例子 12-42.
33.7. 各种小技巧

---

* 为了记录在一个实际的会话期或多个会话期内运行的用户脚本,可以加下面的代码到每个你想追踪记录的脚本里. 这会记录下连续的脚本名记录和调用的次数.

```
1 # 添加(>>)下面几行到你想追踪记录的脚本末尾处.
2
3 whoami>> $SAVE_FILE    # 记录调用脚本的用户.
4 echo $0>> $SAVE_FILE   # 记录脚本名.
5 date>> $SAVE_FILE      # 记录日期和时间.
6 echo>> $SAVE_FILE      # 空行作为分隔行.
7
8 #  当然, SAVE_FILE 变量应在~/.bashrc中定义并导出(export)
9 #+ (变量值类似如 ~/.scripts-run)
```

* >> 操作符可以在文件尾添加内容. 如果你想在文件头添加内容,那应该怎么办?

```
1 file=data.txt
2 title="***This is the title line of data text file***"
3
4 echo $title | cat - $file >$file.new
5 # "cat -" 连接标准输出的内容和$file的内容.
6 #  最后的结果就是生成了一个新文件,
7 #+ 文件的头添加了 $title 的值,后跟$file的内容.
```

这是早先例子 17-13中的简化变体. 当然, , sed 也可以办到.
* 脚本也可以像内嵌到另一个shell脚本的普通命令一样调用, 如 Tcl 或 wish 脚本, 甚至可以是Makefile. 它们可以作为外部shell命令用C语言的 system() 函数调用, 例如.,
system("script_name");.
* 把内嵌的 sed 或 awk 脚本的内容赋值给一个变量可以增加包装脚本(shell wrapper)的可读性. 参考 例子 A-1 和 例子 11-18.

SHELL十三问

* 把你最喜欢和最有用的定义和函数放在一些文件中. 当需要的使用的时候, 在脚本中使用 dot (.) 或 source 命令来"包含(include)"这些"库文件"的一个或多个.

```
 1 # 脚本库
 2 # ------ -------
 3
 4 # 注:
 5 # 本文件没有"#!"开头.
 6 # 也没有真正做执行动作的代码.
 7
 8
 9 # 有用的变量定义
10
11 ROOT_UID=0           # Root用户的 $UID 值是0.
12 E_NOTROOT=101        # 非root用户出错代码.
13 MAXRETVAL=255        # 函数最大的的返回值(正值).
14 SUCCESS=0
15 FAILURE=-1
16
17
18
19 # 函数
20
21 Usage ()             # "Usage:" 信息(即帮助信息).
22 {
23   if [ -z "$1" ]     # 没有传递参数.
24   then
25     msg=filename
26   else
27     msg=$@
28   fi
29
30   echo "Usage: `basename $0` "$msg""
31 }
32
33
34 Check_if_root ()     # 检查是不是root在运行脚本.
35 {                    # 取自例子"ex39.sh".
36   if [ "$UID" -ne "$ROOT_UID" ]
37   then
38     echo "Must be root to run this script."
39     exit $E_NOTROOT
40   fi
41 }
42
43
44 CreateTempfileName ()  # 创建一个"唯一"的临时文件.
45 {                    # 取自例子"ex51.sh".
46   prefix=temp
```

```
47  suffix=`eval date +%s`
48  Tempfilename=$prefix.$suffix
49 }
50
51
52 isalpha2 ()         # 测试字符串是不是都是字母组成的.
53 {                   # 取自例子"isalpha.sh".
54  [ $# -eq 1 ] || return $FAILURE
55
56  case $1 in
57  *[!a-zA-Z]*|"") return $FAILURE;;
58  *) return $SUCCESS;;
59  esac                # Thanks, S.C.
60 }
61
62
63 abs ()                  # 绝对值.
64 {                       # 注意: 最大的返回值 = 255.
65  E_ARGERR=-999999
66
67  if [ -z "$1" ]             # 要传递参数.
68  then
69    return $E_ARGERR         # 返回错误.
70  fi
71
72  if [ "$1" -ge 0 ]          # 如果非负的值,
73  then                    #
74    absval=$1                # 绝对值是本身.
75  else                    # 否则,
76    let "absval = (( 0 - $1 ))"  # 改变它的符号.
77  fi
78
79  return $absval
80 }
81
82
83 tolower ()          #  把传递的字符串转为小写
84 {                   #
85
86  if [ -z "$1" ]     #  如果没有传递参数,
87  then               #+ 打印错误信息
88    echo "(null)"     #+ (C风格的void指针的错误信息)
89    return           #+ 然后从函数中返回.
90  fi
91
92  echo "$@" | tr A-Z a-z
93  # 转换传递过来的所有参数($@).
94
```

```
 95   return
 96
 97 # 用命令替换功能把函数的输出赋给变量.
 98 # 例如:
 99 #    oldvar="A seT of miXed-caSe LEtTerS"
100 #    newvar=`tolower "$oldvar"`
101 #    echo "$newvar"    # 一串混合大小写的字符转换成了全部小写字符
102 #
103 # 练习: 重写这个函数把传递的参数变为大写
104 #       ... toupper()  [容易].
105 }
```

* 在脚本中添加特殊种类的注释开头标识有助于条理清晰和可读性.

```
 1 ## 表示注意.
 2 rm -rf *.zzy   ##  "rm"命令的"-rf"组合选项非常的危险,
 3              ##+ 尤其是对通配符而言.
 4
 5 #+ 表示继续上一行.
 6 #  这是第一行
 7 #+ 这是多行的注释,
 8 #+ 这里是最后一行.
 9
10 #* 表示标注.
11
12 #o 表示列表项.
13
14 #> 表示另一个观点.
15 while [ "$var1" != "end" ]    #> while test "$var1" != "end"
```

* if-test 结构的一种聪明用法是用来注释一块代码块.

```
 1 #!/bin/bash
 2
 3 COMMENT_BLOCK=
 4 #  给上面的变量设置某个值就会产生讨厌的结果
 5 #
 6
 7 if [ $COMMENT_BLOCK ]; then
 8
 9 Comment block --
10 ================================
11 This is a comment line.
12 This is another comment line.
13 This is yet another comment line.
14 ================================
15
16 echo "This will not echo."
17
18 Comment blocks are error-free! Whee!
19
20 fi
```

```
21
22 echo "No more comments, please."
23
24 exit 0
```

把这种方法和使用here documents来注释代码块作一个比较.

* 测试$? 退出状态变量, 因为一个脚本可能想要测试一个参数是否只包含数字,以便后面可以把它当作一个整数.

```
 1 #!/bin/bash
 2
 3 SUCCESS=0
 4 E_BADINPUT=65
 5
 6 test "$1" -ne 0 -o "$1" -eq 0 2>/dev/null
 7 # 整数要么等于零要么不等于零.
 8 # 2>/dev/null 可以抑制错误信息.
 9
10 if [ $? -ne "$SUCCESS" ]
11 then
12   echo "Usage: `basename $0` integer-input"
13   exit $E_BADINPUT
14 fi
15
16 let "sum = $1 + 25"            # 如果$1不是整数就会产生错误.
17 echo "Sum = $sum"
18
19 # 任何变量,而不仅仅命令行参数可用这种方法来测试.
20
21 exit 0
```

* 0 - 255 范围的函数返回值是个严格的限制. 用全局变量和其他方法常常出问题. 函数内返回值给脚本主体的另一个办法是让函数写值到标准输出(通常是用echo) 作为"返回值", 并且将其赋给一个变量. 这实际是命令替换(command substitution)的变体.

Example 33-15 返回值技巧

```
################################Start Script#######################################
 1 #!/bin/bash
 2 # multiplication.sh
 3
 4 multiply ()                  # 传递乘数.
 5 {                           # 能接受多个参数.
 6
 7   local product=1
 8
 9   until [ -z "$1" ]          # 直到所有参数都处理完毕...
10   do
11     let "product *= $1"
12     shift
13   done
14
15   echo $product               #  不会打印到标准输出,
```

```
16 }                        #+ 因为要把它赋给一个变量.
17
18 mult1=15383; mult2=25211
19 val1=`multiply $mult1 $mult2`
20 echo "$mult1 X $mult2 = $val1"
21                          # 387820813
22
23 mult1=25; mult2=5; mult3=20
24 val2=`multiply $mult1 $mult2 $mult3`
25 echo "$mult1 X $mult2 X $mult3 = $val2"
26                          # 2500
27
28 mult1=188; mult2=37; mult3=25; mult4=47
29 val3=`multiply $mult1 $mult2 $mult3 $mult4`
30 echo "$mult1 X $mult2 X $mult3 X $mult4 = $val3"
31                          # 8173300
32
33 exit 0
```

################################End Script##

相同的技术也可用在字符串中. 这意味着函数可以"返回"一个非数字的值.

```
 1 capitalize_ichar ()        #  把传递来的参数字符串的第一个字母大写
 2 {                         #
 3
 4   string0="$@"            # 能接受多个参数.
 5
 6   firstchar=${string0:0:1}   # 第一个字符.
 7   string1=${string0:1}       # 余下的字符.
 8
 9   FirstChar=`echo "$firstchar" | tr a-z A-Z`
10                          # 第一个字符转换成大写字符.
11
12   echo "$FirstChar$string1"  # 打印到标准输出.
13
14 }
15
16 newstring=`capitalize_ichar "every sentence should start with a capital letter."`
17 echo "$newstring"          # Every sentence should start with a capital letter.
```

用这种办法甚至可能"返回" 多个值.

Example 33-16 整型还是string?

################################Start Script##

```
1 #!/bin/bash
2 # sum-product.sh
3 # 函数可以"返回"多个值.
4
5 sum_and_product ()   # 计算所传参数的总和与乘积.
6 {
7   echo $(( $1 + $2 )) $(( $1 * $2 ))
8 # 打印每个计算的值到标准输出,各值用空格分隔开.
```

```
 9 }
10
11 echo
12 echo "Enter first number "
13 read first
14
15 echo
16 echo "Enter second number "
17 read second
18 echo
19
20 retval=`sum_and_product $first $second`     # 把函数的输出赋值给变量.
21 sum=`echo "$retval" | awk '{print $1}'`     # 把第一个域的值赋给sum变量.
22 product=`echo "$retval" | awk '{print $2}'` # 把第二个域的值赋给product变量.
23
24 echo "$first + $second = $sum"
25 echo "$first * $second = $product"
26 echo
27
28 exit 0
```

##################################End Script##

* 下一个技巧是传递数组给函数的技术, 然后 "返回" 一个数组给脚本.
 用 变量替换(command substitution)把数组的所有元素用空格分隔开来并赋给一个变量
 就可以实现给函数传递数组. 用先前介绍的方法函数内echo一个数组并"返回此值",然后
 调用命令替换用 (...) 操作符赋值给一个数组.

Example 33-17 传递和返回数组

##################################Start Script###

```
 1 #!/bin/bash
 2 # array-function.sh: 传递一个数组给函数并且...
 3 #                  从函数"返回"一个数组
 4
 5
 6 Pass_Array ()
 7 {
 8   local passed_array   # 局部变量.
 9   passed_array=( `echo "$1"` )
10   echo "${passed_array[@]}"
11   #  列出新数组中的所有元素
12   #+ 新数组是在函数内声明和赋值的.
13 }
14
15
16 original_array=( element1 element2 element3 element4 element5 )
17
18 echo
19 echo "original_array = ${original_array[@]}"
20 #                   列出最初的数组元素.
21
```

```
22
23 # 下面是传递数组给函数的技巧.
24 # ***********************************
25 argument=`echo ${original_array[@]}`
26 # ***********************************
27 #  把原数组的所有元素用空格分隔开合成一个字符串并赋给一个变量
28 #
29 #
30 # 注意：只是把数组本身传给函数是不会工作的.
31
32
33 # 下面是允许数组作为"返回值"的技巧.
34 # ****************************************
35 returned_array=( `Pass_Array "$argument"` )
36 # ****************************************
37 # 把函数的输出赋给数组变量.
38
39 echo "returned_array = ${returned_array[@]}"
40
41 echo "============================================================="
42
43 #  现在,再试一次Now, try it again,
44 #+ 尝试在函数外存取(列出)数组.
45 Pass_Array "$argument"
46
47 # 函数本身可以列出数组,但...
48 #+ 函数外存取数组被禁止.
49 echo "Passed array (within function) = ${passed_array[@]}"
50 # 因为变量是函数内的局部变量,所以只有NULL值.
51
52 echo
53
54 exit 0
################################End Script############################################
 在例子 A-10中有一个更精心制作的给函数传递数组的例子.
* 利用双括号结构,使在for 和 while 循环中可以使用C风格的语法来设置和增加变量. 参
 考例子 10-12 和 例子 10-17.
* 在脚本开头设置 path 和 umask 增加脚本的"可移植性" -- 在某些把 $PATH 和 umask
 弄乱的系统里也可以运行.
   1 #!/bin/bash
   2 PATH=/bin:/usr/bin:/usr/local/bin ; export PATH
   3 umask 022   # 脚本的创建的文件有 755 的权限设置.
   4
   5 # 多谢Ian D. Allen提出这个技巧.
* 一个有用的脚本技术是：重复地把一个过滤器的输出回馈(用管道)给另一个相同过滤器,
 但过滤器有不同的参数和/或选项. 尤其对 tr 和 grep 更合适.
   1 # 取自例子"wstrings.sh".
   2
```

```
  3 wlist=`strings "$1" | tr A-Z a-z | tr '[:space:]' Z | \
  4 tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`
```

Example 33-18 anagrams游戏

```
################################Start Script#########################################
 1 #!/bin/bash
 2 # agram.sh: 用anagrams玩游戏.
 3
 4 # 寻找 anagrams ...
 5 LETTERSET=etaoinshrdlu
 6 FILTER='.......'      # 最小有多少个字母?
 7 #       1234567
 8
 9 anagram "$LETTERSET" | # 找出这串字符中所有的 anagrams ...
10 grep "$FILTER" |      # 至少7个字符,
11 grep '^is' |          # 以'is'开头
12 grep -v 's$' |        # 不是复数的(指英文单词复数)
13 grep -v 'ed$'         # 不是过去式的(当然也是英文单词)
14 # 可以加许多组合条件和过滤器.
15
16 #  使用 "anagram" 软件
17 #+ 它是作者 "yawl" 单词列表软件包的一部分.
18 #  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz
19 #  http://personal.riverusers.com/~thegrendel/yawl-0.3.2.tar.gz
20
21 exit 0               # 代码结束.
22
23
24 bash$ sh agram.sh
25 islander
26 isolate
27 isolead
28 isotheral
29
30
31
32 #  练习:
33 #  ---------
34 #  修改这个脚本使 LETTERSET 能作为命令行参数来接受.
35 #  能够传递参数给第 11 - 13 行的过滤器(就像 $FILTER),
36 #+ 以便能靠传递参数来指定一种功能.
37
38 #  参考agram2.sh了解些微不同的anagram的一种方法
39 #
################################End Script###########################################
```

See also Example 27-3, Example 12-22, and Example A-9.

* 使用"匿名的 here documents" 来注释代码块,这样避免了对代码块的每一块单独用#来注释了. 参考例子 17-11.
* 当依赖某个命令脚本在一台没有安装该命令的机器上运行时会出错. 使用 whatis 命令可

以避免此问题.

```
 1 CMD=command1              # 第一选择First choice.
 2 PlanB=command2            # 第二选择Fallback option.
 3
 4 command_test=$(whatis "$CMD" | grep 'nothing appropriate')
 5 #  如果'command1'没有在系统里发现 , 'whatis'会返回:
 6 #+ "command1: nothing appropriate."
 7 #
 8 #  另一种更安全的办法是:
 9 #     command_test=$(whereis "$CMD" | grep \/)
10 #  但后面的测试判断应该翻转过来,
11 #+ 因为$command_test只有当系统存在$CMD命令时才有内容.
12 #
13 #     (Thanks, bojster.)
14
15
16 if [[ -z "$command_test" ]]  # 检查命令是否存在.
17 then
18   $CMD option1 option2       #  调用command1.
19 else                        #  否则,
20   $PlanB                     #+ 调用command2.
21 fi
```

* 在发生错误的情况下 if-grep test 可能不会返回期望的结果,因为文本是打印在标准出错而不是标准输出上.

```
 1 if ls -l nonexistent_filename | grep -q 'No such file or directory'
 2   then echo "File \"nonexistent_filename\" does not exist."
 3 fi
```

把标准出错重定向到标准输出上可以修改这个.

```
 1 if ls -l nonexistent_filename 2>&1 | grep -q 'No such file or directory'
 2 #                             ^^^^
 3   then echo "File \"nonexistent_filename\" does not exist."
 4 fi
 5
 6 # 多谢Chris Martin指出.
```

* The run-parts 命令很容易依次运行一组命令脚本,特别是和 cron 或 at 组合起来.
* 在shell脚本里能调用 X-Windows 的窗口小部件将多么美好. 已经存在有几种工具包实现这个了, 它们称为Xscript, Xmenu, 和 widtools. 头两个已经不再维护. 幸运地是仍然可以从这儿下载widtools.

注意: widtools (widget tools) 工具包要求安装了 XForms 库. 另外, 它的 Makefile 在典型的Linux系统上安装前需要做一些合适的编辑. 最后, 提供的6个部件有3个不能工作 (事实上会发生段错误).

dialog 工具集提供了shell脚本使用一种称为"对话框"的窗口部件. 原始的 dialog 软件包工作在文本模式的控制台下, 但它的后续软件 gdialog, Xdialog, 和 kdialog 使用基于X-Windows的窗口小部件集.

Example 33-19 在shell脚本中调用的窗口部件

```
################################Start Script#########################################
1 #!/bin/bash
2 # dialog.sh: 使用 'gdialog' 窗口部件.
```

```
3 # 必须在你的系统里安装'gdialog'才能运行此脚本.
4 # 版本 1.1 (04/05/05 修正)
5
6 # 这个脚本的灵感源自下面的文章.
7 #     "Scripting for X Productivity," by Marco Fioretti,
8 #      LINUX JOURNAL, Issue 113, September 2003, pp. 86-9.
9 # Thank you, all you good people at LJ.
10
11
12 # 在窗口中的输入错误.
13 E_INPUT=65
14 # 输入窗口显示的尺寸.
15 HEIGHT=50
16 WIDTH=60
17
18 # 输出文件名 (由脚本名构建而来).
19 OUTFILE=$0.output
20
21 # 把这个脚本的内容显示在窗口中.
22 gdialog --title "Displaying: $0" --textbox $0 $HEIGHT $WIDTH
23
24
25
26 # 现在,保存输入到输出文件中.
27 echo -n "VARIABLE=" > $OUTFILE
28 gdialog --title "User Input" --inputbox "Enter variable, please:" \
29 $HEIGHT $WIDTH 2>> $OUTFILE
30
31
32 if [ "$?" -eq 0 ]
33 # 检查退出状态是一个好习惯.
34 then
35   echo "Executed \"dialog box\" without errors."
36 else
37   echo "Error(s) in \"dialog box\" execution."
38        # 或者, 点击"Cancel", 而不是"OK" 按钮.
39   rm $OUTFILE
40   exit $E_INPUT
41 fi
42
43
44
45 # 现在,我们重新取得并显示保存的变量.
46 . $OUTFILE   # 'Source' 保存的文件(即执行).
47 echo "The variable input in the \"input box\" was: "$VARIABLE""
48
49
50 rm $OUTFILE  # 清除临时文件.
```

51 # 有些应用可能需要保留这些文件.
52
53 exit $?
################################End Script###
其他的在脚本中使用窗口的工具还有 Tk 或 wish (Tcl 派生物), PerlTk (Perl 的 Tk 扩展), tksh (ksh 的 Tk 扩展), XForms4Perl (Perl 的 XForms 扩展), Gtk-Perl (Perl 的 Gtk 扩展), 或 PyQt (Python 的 Qt 扩展).
* 为了对复杂的脚本做多次的版本修订管理, 可以使用 rcs 软件包.
使用这个软件包的好处之一是会自动地升级ID头标识.在 rcs 的co命令处理一些预定义的关键字参数替换,例如,代替脚本里头#Id的,如类似下面的行:

```
1 #$Id: hello-world.sh,v 1.1 2004/10/16 02:43:05 bozo Exp $
```

33.8. 安全话题

33.8.1. 被感染的脚本

有一个简短的关于脚本安全的介绍是适当的. 脚本程序可能会包含蠕虫病毒,特洛伊木马, 或是其他的病毒. 由于这些原因, 决不要以root身份运行脚本 (或允许它被插入到系统的/etc/rc.d里的启动脚本中) 除非你确定这是值得信赖的源码或你已经很小心地分析过了脚本并确信它不会有什么危害.
Bell实验室及其他地方的病毒研究人员, 包括 M. Douglas McIlroy, Tom Duff, 和 Fred Cohen 已经调查过了shell脚本病毒的实现. 他们认为写脚本病毒甚至对于新手来说也是很容易的,就像 “ 脚本小子(script kiddie) ” 也能容易地写出. [1]
这也是学习脚本编程的原因之一：学习读懂脚本和理解脚本可以保护你的系统免受骇客攻击或破坏.
33.8.2. 隐藏Shell脚本源码

为了安全, 使脚本不可读是有必要的. 如果有软件可以把脚本转化成相应的二进制执行文件就好了. Francisco Rosales的 shc - 通用的Shell脚本编译器(generic shell script compiler) 可以出色地完成目标.
不幸地, 依照发表在2005年十月的Linux Journal杂志上的一篇文章, 二进制文件可以,至少在某些情况下能被恢复回脚本的原始代码. 但不管怎么说,这对大多数技术不高超的骇客来说仍然是一个保持脚本安全的有效的方法.
注意事项:
[1] 参考 Marius van Oers 的文章(Unix Shell Scripting Malware),和列在参考书目 (bibliography)的参考资料.
33.9. 移植话题

这本书是关于在GNU/Linux系统下的Bash编程. 但同样,sh 和 ksh 用户也能在这儿得到许多有用的价值.
以现在的情况来看,许多种shell和脚本语言都尽力使自己符合 POSIX 1003.2 标准. 用 --posix 选项调用Bash或在脚本开头插入 set -o posix 就能使Bash能以很接近这个标准的方式运行. 在脚本开头用

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

比用

```
1 #!/bin/bash
```

会更好.注意在Linux和一些UNIX风格的系统里/bin/sh是/bin/bash的一个链接(link), 并且如果脚本以/bin/sh调用时会禁用Bash的扩展功能.
大多数的Bash脚本能不作修改就能运行在 ksh下, 反之亦然, 因为 Chet Ramey 辛勤地把 ksh

的属性移植到了最新的Bash版本.
在商业的 UNIX 机器上, 使用了GNU扩展属性的标准命令的脚本可能不会工作. 这个问题在最近几年已经有所改观了, 因为GNU软件包完美地代替了在这些"大块头的"UNIX运行的相应工具. 源码分发 给传统UNIX上加快了这种趋势.
Bash 有传统的 Bourne shell 缺乏的一些属性. 下面是其中一些:
* 一些扩展的 调用选项(invocation options)
* 使用 $( ) 结构来完成命令替换(Command substitution)
* 一些 字符串处理(string manipulation) 操作符
* 进程替换(Process substitution)
* Bash的 内建(builtins) 命令
参考 Bash F.A.Q. 查看完整的列表.
33.10. 在Windows下进行Shell编程

使用其他操作系统用户希望能运行UNIX类型的脚本能在他们的系统上运行, 因此也希望能在这本书里能学到这方面的知识. 来自Cygnus的 Cygwin 软件结合来自Mortice Kern的MKS软件包(MKS utilities)可以给Windows添加shell脚本的兼容.
已经有正式宣布Windows的将来版本会包含Bash风格的命令行和脚本能力,但目前为止还没有结果.
第34章 Bash, 版本 2 和 3
==========
34.1. Bash, 版本2

当前运行在你的机器里的Bash版本号是版本 2.xx.y 或 3.xx.y.
```
bash$ echo $BASH_VERSION
2.05.b.0(1)-release
```
经典的Bash版本2编程语言升级版增加了数组变量, [1] 字符串和参数扩展, 和间接变量引用的更好的方法,及其他的属性.
Example 34-1 字符串扩展
```
#################################Start Script#########################################
 1 #!/bin/bash
 2
 3 # 字符串扩展.
 4 # Bash版本2引入的特性.
 5
 6 #  具有$'xxx'格式的字符串
 7 #+ 将会解释里面的标准的转义字符.
 8
 9 echo $'Ringing bell 3 times \a \a \a'
10     # 可能在一些终端只能响铃一次.
11 echo $'Three form feeds \f \f \f'
12 echo $'10 newlines \n\n\n\n\n\n\n\n\n\n'
13 echo $'\102\141\163\150'   # Bash
14                          # 八进制相等的字符.
15
16 exit 0
#################################End Script###########################################
```
Example 34-2 间接变量引用 - 新方法
```
#################################Start Script#########################################
```

```
 1 #!/bin/bash
 2
 3 # 间接变量引用.
 4 # 这有点像C++的引用属性.
 5
 6
 7 a=letter_of_alphabet
 8 letter_of_alphabet=z
 9
10 echo "a = $a"          # 直接引用.
11
12 echo "Now a = ${!a}"   # 间接引用.
13 # ${!variable} 形式比老的"eval var1=\$$var2"更高级
14
15 echo
16
17 t=table_cell_3
18 table_cell_3=24
19 echo "t = ${!t}"                  # t = 24
20 table_cell_3=387
21 echo "Value of t changed to ${!t}"   # 387
22
23 #  这在用来引用数组或表格的成员时非常有用,
24 #+ 或用来模拟多维数组.
25 #  如果有可索引的选项 (类似于指针运算)
26 #+ 会更好. 唉.
27
28 exit 0
```

################################End Script###

Example 34-3 使用间接变量引用的简单数据库应用

################################Start Script###

```
 1 #!/bin/bash
 2 # resistor-inventory.sh
 3 # 使用间接变量引用的简单数据库应用.
 4
 5 # ============================================================== #
 6 # 数据
 7
 8 B1723_value=470                    # 值
 9 B1723_powerdissip=.25               # 是什么
10 B1723_colorcode="yellow-violet-brown"      # 色彩带宽
11 B1723_loc=173                      # 它们存在哪儿
12 B1723_inventory=78                 # 有多少
13
14 B1724_value=1000
15 B1724_powerdissip=.25
16 B1724_colorcode="brown-black-red"
17 B1724_loc=24N
```

```
18 B1724_inventory=243
19
20 B1725_value=10000
21 B1725_powerdissip=.25
22 B1725_colorcode="brown-black-orange"
23 B1725_loc=24N
24 B1725_inventory=89
25
26 # ============================================================== #
27
28
29 echo
30
31 PS3='Enter catalog number: '
32
33 echo
34
35 select catalog_number in "B1723" "B1724" "B1725"
36 do
37   Inv=${catalog_number}_inventory
38   Val=${catalog_number}_value
39   Pdissip=${catalog_number}_powerdissip
40   Loc=${catalog_number}_loc
41   Ccode=${catalog_number}_colorcode
42
43   echo
44   echo "Catalog number $catalog_number:"
45   echo "There are ${!Inv} of [${!Val} ohm / ${!Pdissip} watt] resistors in stock."
46   echo "These are located in bin # ${!Loc}."
47   echo "Their color code is \"${!Ccode}\"."
48
49   break
50 done
51
52 echo; echo
53
54 # 练习:
55 # ---------
56 # 1) 重写脚本,使其从外部文件里读数据.
57 # 2) 重写脚本,用数组代替间接变量引用
58 #
59 #    用数组会更简单明了
60
61
62 # 注:
63 # -----
64 #  Shell脚本除了最简单的数据应用,其实并不合适数据库应用,
65 #+ 它过多地依赖实际工作的环境和命令.
```

```
66 #  写数据库应用更好的还是用一门自然支持数据结构的语言,
67 #+ 如 C++ 或 Java (或甚至是 Perl).
68
69 exit 0
```

################################End Script###

Example 34-4 用数组和其他的小技巧来处理四人随机打牌

################################Start Script###

```
 1 #!/bin/bash
 2
 3 # Cards:
 4 # 处理四人打牌.
 5
 6 UNPICKED=0
 7 PICKED=1
 8
 9 DUPE_CARD=99
10
11 LOWER_LIMIT=0
12 UPPER_LIMIT=51
13 CARDS_IN_SUIT=13
14 CARDS=52
15
16 declare -a Deck
17 declare -a Suits
18 declare -a Cards
19 #  用一个三维数据来描述数据会更容易实现也更明了一些.
20 #
21 #  可能Bash将来的版本会支持多维数组.
22
23
24 initialize_Deck ()
25 {
26 i=$LOWER_LIMIT
27 until [ "$i" -gt $UPPER_LIMIT ]
28 do
29   Deck[i]=$UNPICKED   # 把整副牌的每张牌都设为没人持牌.
30   let "i += 1"
31 done
32 echo
33 }
34
35 initialize_Suits ()
36 {
37 Suits[0]=C #梅花
38 Suits[1]=D #方块
39 Suits[2]=H #红心
40 Suits[3]=S #黑桃
41 }
```

```
42
43 initialize_Cards ()
44 {
45 Cards=(2 3 4 5 6 7 8 9 10 J Q K A)
46 # 另一种初始化数组的方法.
47 }
48
49 pick_a_card ()
50 {
51 card_number=$RANDOM
52 let "card_number %= $CARDS"
53 if [ "${Deck[card_number]}" -eq $UNPICKED ]
54 then
55   Deck[card_number]=$PICKED
56   return $card_number
57 else
58   return $DUPE_CARD
59 fi
60 }
61
62 parse_card ()
63 {
64 number=$1
65 let "suit_number = number / CARDS_IN_SUIT"
66 suit=${Suits[suit_number]}
67 echo -n "$suit-"
68 let "card_no = number % CARDS_IN_SUIT"
69 Card=${Cards[card_no]}
70 printf %-4s $Card
71 # 优雅地打印各张牌.
72 }
73
74 seed_random ()  # 随机产生牌上数值的种子.
75 {               # 如果你没有这么做会有什么发生?
76 seed=`eval date +%s`
77 let "seed %= 32766"
78 RANDOM=$seed
79 #  其他的产生随机用的种子的方法还有什么W?
80 #
81 }
82
83 deal_cards ()
84 {
85 echo
86
87 cards_picked=0
88 while [ "$cards_picked" -le $UPPER_LIMIT ]
89 do
```

```
  pick_a_card
  t=$?

  if [ "$t" -ne $DUPE_CARD ]
  then
    parse_card $t

    u=$cards_picked+1
    # 改回1步进的索引(临时的). 为什么?
    let "u %= $CARDS_IN_SUIT"
     if [ "$u" -eq 0 ]   # 内嵌的 if/then 条件测试.
     then
      echo
      echo
     fi
    # Separate hands.

    let "cards_picked += 1"
   fi
done

echo

return 0
}


# 结构化编程:
# 整个程序逻辑模块化.

#================
seed_random
initialize_Deck
initialize_Suits
initialize_Cards
deal_cards
#================

exit 0



# 练习 1:
# 把这个脚本完整地做注释.

# 练习 2:
# 增加一个处理例程 (函数) 来以花色排序打印出每个人手中的牌.
# 如果你高兴,可增加你喜欢的各种酷的代码.
```

138
139 # 练习 3:
140 # 简化和理顺脚本的逻辑.
################################End Script###
注意事项:
[1] Chet Ramey 承诺会在Bash的未来版本中实现关联数组(associative arrays)
(一个Perl特性). 到了版本3,这个特性还没有实现.
34.2. Bash版本3

在2004年7月27日, Chet Ramey 发布了Bash的第三版本. 它修复了许多bug并加入了一些新的
属性.
增加的一些属性有:
* 新的,更特别的不可移植的 {a..z} 花括号扩展(brace expansion) 操作符.

```
 1 #!/bin/bash
 2
 3 for i in {1..10}
 4 #  比下面的更简单并且更易于理解
 5 #+ for i in $(seq 10)
 6 do
 7   echo -n "$i "
 8 done
 9
10 echo
11
12 # 1 2 3 4 5 6 7 8 9 10
```

* ${!array[@]} 操作符, 它扩展给定的数组(array)的所有元素下标.

```
 1 #!/bin/bash
 2
 3 Array=(element-zero element-one element-two element-three)
 4
 5 echo ${Array[0]}   # 元素0
 6                   # 数组的第一个元素.
 7
 8 echo ${!Array[@]}  # 0 1 2 3
 9                   # 数组所有的下标.
10
11 for i in ${!Array[@]}
12 do
13   echo ${Array[i]} # element-zero
14                    # element-one
15                    # element-two
16                    # element-three
17                    #
18                    # 在数组里的所有元素.
19 done
```

* =~ 正则表达式(Regular Expression) 匹配操作符在双方括号(double brackets)
测试表达式中使用. (Perl也有一个相似的操作符.)

```
 1 #!/bin/bash
```

```
  2
  3 variable="This is a fine mess."
  4
  5 echo "$variable"
  6
  7 if [[ "$variable" =~ "T*fin*es*" ]]
  8 # 在双方括号([[]])里用=~操作符进行正则匹配.
  9 then
 10   echo "match found"
 11       # match found
 12 fi
```

或, 更有用的用法:

```
  1 #!/bin/bash
  2
  3 input=$1
  4
  5
  6 if [[ "$input" =~ "[1-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]" ]]
  7 # NNN-NN-NNNN
  8 # 每个N是一个数字.
  9 # 但, 开头的第一个数字不能是 0.
 10 then
 11   echo "Social Security number."
 12   # 处理 SSN.
 13 else
 14   echo "Not a Social Security number!"
 15   # 或者, 要求正确的输入.
 16 fi
```

还有一个使用 =~ 操作符的例子, 参考例子 A-28.

注意: 升级到Bash版本3使原来在早先版本可以工作的少部分脚本不能工作了. 要重新测试
原来的脚本看是否它们仍然可以工作!
确实发生不能工作的情况, 在 Advanced Bash Scripting Guide 里的部分脚本代码
不得不修复 (例如,例子 A-20 和 例子 9-4).

第35章 后记

==========

35.1. 作者后记

---

doce ut discas

(Teach, that you yourself may learn.)

我怎么会写这本脚本编程的书?这有一个很奇怪的故事.时间回到前几年,那时我准备学习shell脚本编程－－这最好的办法莫过于读一本这方面的好书了.我一直在找一本能覆盖这个主题方方面面的指南参考书.我也在找一本能把难点说得清楚容易并能用实际的代码和代码注释解释这些难以理解的细节. [1] 事实上,我在找一本非常满意的书, 或者是类似的东西.

不幸的是,这是不存在的.如果我想要一本,那我不得不自己写一本.于是,它就写出来了.

这使我想起一个关于疯子教授的故事.下面的事像笨蛋那样荒谬.当这本书上架之前,对于所有的在图书馆的书使他有了想写一本书的主意.他就这样做了,开始了这项好处多多的工作.

他争分夺秒地开始完成和现在这本书标题很像的书,他每天都很快地奔跑回家来做这件事.当几年之后他死掉了,他有了写几千本书保存下来,可能会被和其他的破书一块放在书架上.可

能他写的书不是那么的好,但这有什么关系呢?这是一个活在幻想里的一个朋友.即使他没有被幻想给迷惑驱使...但我忍不住地钦佩这个老笨蛋.
注意事项:
[1] 这是声名狼藉使人郁闷到死的技术.
35.2. 关于作者

这家伙到底是谁?
作者没有外交特权,不是被强迫写作的. [1] 这本书有点违背他的其他的主要工作, HOW-2 Meet Women: The Shy Man's Guide to Relationships. 他另外也写了 Software-Building HOWTO. 近来, 他尝试写一些虚构的短篇小说.
自1995年成为一个Linux用户以来(Slackware 2.2, kernel 1.2.1), 作者已经发表了一些软件包,包括 cruft 一次性加密软件包(one-time pad encryption utility), 软件 mcalc 可用做计算器, 软件judge是Scrabble拼字游戏的自动求解包, 和软件包yawl 一起组成猜词表.
他从一台CDC 3800上使用FORTRAN IV开始他的编程之旅, 但那种日子一点也不值得怀念.
作者和他的妻子还有他们的狗隐居在一个偏远的地方,他幻想着人性是善良的.
注意事项:
[1] 这些谁可以做,谁不可以做...拿到一个MCSE证书.
35.3. 哪里可以取得帮助?

作者不是太忙(并且心情不错)的话,会回答一般性的脚本编程问题.但是,如果你的特定应用的脚本不工作,建议你最好把问题发到 comp.os.unix.shell 新闻组去.
35.4. 制作这本书的工具

35.4.1 硬件

一个运行着Red Hat 7.1/7.3的IBM Thinkpad, model 760XL(P166, 104 meg RAM)的笔记本.
是的,它非常的缓慢并且还有一个更人胆战心惊的键盘,但它总比一根铅笔加一个巨大的写字板好多了.
升级： 升级到了运行着FC3的 770Z Thinkpad (P2-366, 192 meg RAM)笔记本.谁想捐赠一个更新的一点笔记本给这个快饿死的作者 <g>?
35.4.2 软件和排版软件
---
1. Bram Moolenaar的功能强大的SGML软件 vim文本编辑器.
2. OpenJade, 一个把SGML文档转换为其他格式的DSSSL翻译引擎.
3. Norman Walsh的DSSSL排版框架.
4. 由Norman Walsh和Leonard Muellner (O'Reilly, ISBN 1-56592-580-7)写的最权威的指南：DocBook. 它仍然是任何一个想写Docbook SGML格式的书的标准参考书.

todo
====
12.22 第2部分翻译结束.阶段性总结.
12.02 本打算自己看,做做笔记就算了,如果要大家看,就不行了.补全前3章所有未译的地方.
01.06 所有的没翻译或翻译不对的地方标记<rojy bug>.
01.08 一些朋友提出想要,可惜没译完,最近效率比较低,手里还有些其它的活:(.
丑媳妇总要见公婆,还是放了吧.
01.13 11章结束
01.14 12章开始, 翻译html版本.
01.19 开始12.4节.

SHELL十三问

02.16 开始12.6节.
03.23 12章终于结束了...
04.15 前3部分终于结束了, 历时4个多月:(, 后边的部分就交给
05.15 终于完成了, 没有黄毅兄的帮助, 真是不敢想象, 恐怕就要流产了
接下来就要搞sgml版本的了, 唉... 到底走弯路了...
[]

awk基础入门(1)
文章整理: 文章来源: 网络
Awk是一种非常好的语言，同时有一个非常奇怪的名称。在本系列文章中，DanielRobbins 将使您迅速掌握 awk编程技巧。随着本系列的进展，将讨论更高级的主题，最后将演示一个真正的高级awk 演示程序。
捍卫 awk
在本系列文章中，我将使您成为精通 awk 的编码人员。我承认，awk 并没有一个非常好听且又非常“ 时髦 ”的名字。awk 的 GNU 版本（叫作 gawk）听起来非常怪异。那些不熟悉这种语言的人可能听说过 "awk"，并可能认为它是一组落伍且过时的混乱代码。它甚至会使最博学的 UNIX 权威陷于错乱的边缘（使他不断地发出 "kill -9!" 命令，就象使用咖啡机一样）。
的确，awk 没有一个动听的名字。但它是一种很棒的语言。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
[]

awk基础入门(2)
文章整理: 文章来源: 网络
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
[]

awk基础入门(3)
文章整理: 文章来源: 网络
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 }}
[]

制作工具：小说下载阅读器 http://www.mybook66.com<PIXTEL_MMI_EBOOK_2005>2
</PIXTEL_MMI_EBOOK_2005>