尚观 ULA UNIX/Linux 集群架构师教材

Shell Script 教程

（Shell 脚本编程）

By 尚观 Linux 研究室 Kevin Zou
kevin@uplooking.com

Kevinzou (kissingwolf@gmail.com)

全书目录

第1章 shell 简介

第1.1节 什么是 shell

Shell 是一个命令解释器，是人与操作系统之间的桥梁。

我们平时无论任何操作，最终都要操作硬件，比如输入一个字符“ a ”，那么信号首先会从键盘传递到主板，通过主板总线传递到内存，ＣＰＵ，显卡等，最终经过显卡的运算完成后在屏幕的某个位置，显示一个特定字体的字符“ a ”，这一整个过程可以说是不断的和硬件打交道了，但是如果让人去发送这些硬件操作码显然不适合，因为这不是人干的事，所以我们有了操作系统，操作系统通过加载一定的硬件驱动，从而控制硬件，操作硬件，那剩下的事就是如何和操作系统通信了，对于普通的系统管理员来说，这也是一件非常困难的事，为了方便人和操作系统沟通，我们开发了 shell。

Shell 可以将我们平时运行的一些指令解释给操作系统执行，方便管理员操作系统。

而 Shell 的脚本其实是一种命令的堆积，我们将所有需要执行的命令，以从上至下的方式写在一个文件当中，交给 shell 去自动解释执行。

第1.2节 shell 历史

在 AT&T 的 Dennis Ritchie 和 Ken Thompson 设计 UNIX™ 的时候，他们想要为用户创建一种与他们的新系统交流的方法。

那时的操作系统带有命令解释器。命令解释器接受用户的命令，然后解释它们，因而计算机可以使用这些命令。

但是 Ritchie 和 Thompson 想要的不只是这些功能，他们想提供比当时的命令解释器具备更优异功能的工具。这导致了 Bourne shell（通称为 sh）的开发，由 S.R. Bourne 创建。自从 Bourne shell 的创建，其它 shell 也被一一开发，如 C shell（csh）和 Korn shell（ksh）。

当自由软件基金会想寻求一种免费的 shell，开发者们开始致力于 Bourne shell 以及当时其它 shell 中某些很受欢迎的功能背后的语言。

这个开发结果是 Bourne Again Shell，或称 bash。虽然你的 Red Hat Linux 包括几种不同的 shell，bash 是为互动用户提供的默认 shell 。

第1.3节　常见的 shell

Bourne shell 即 sh：AT&T 贝尔实验室编写的一个交换式的命令解释器。

C Shell ：Bill Joy 于 20 世纪 80 年代早期开发。为了让用户更容易的使用，他把语法结构变成了 C 语言风格。它新增了命令历史、别名、文件名替换、作业控制等功能。

korn shell (ksh) 是一个 Unix shell 。它由贝尔实验室的 David Korn 在二十世纪八十年代早期编写。它完全向上兼容 Bourne shell 并包含了 C shell 的很多特性。

Bourne-Again SHell： bash 是一个为 GNU 项目编写的 Unix shell。它的名字是一系列缩写：Bourne-Again SHell — 这是关于 Bourne shell（sh）的一个双关语（Bourne again / born again）。Bourne shell 是一个早期的重要 shell，由 Stephen Bourne 在 1978 年前后编写，并同 Version 7 Unix 一起发布。bash 则在 1987 年由 Brian Fox 创造。在 1990 年，Chet Ramey 成为了主要的维护者。

bash 是大多数 Linux 系统以及 Mac OS X v10.4 默认的 shell，它能运行于大多数 Unix 风格的操作系统之上，甚至被移植到了 Microsoft Windows 上的 Cygwin 和 MSYS 系统中，以实现 windows 的 POSIX 虚拟接口。此外，它也被 DJGPP 项目移植到了 MS-DOS 上。

POSIX shell ：POSIX shell 与 Korn shell 非常的相似，当前提供 POSIX shell 的最大卖主是 Hewlett-Packard。

第1.4节　为什么 Shell

解决重复操作的作业。

节约时间,提高工作效率。

功能强大，使用简单。

第2章　Shell 的基本使用

第2.1节　编程理念和流程图画法

算法的性质：　1）算法是一个有穷的解题序列。

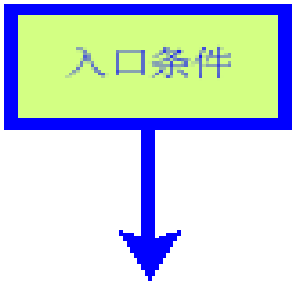
2）动作序列仅有一个初始动作。

3）序列中每一个动作仅有一个初始动作。

4）序列的终止表示问题得到解答或者没有解答。

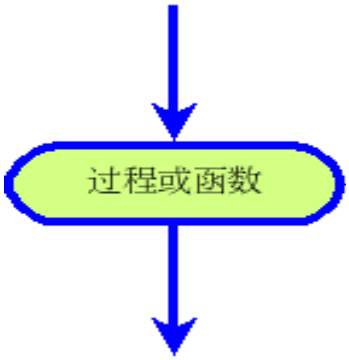
第2.2节　流程图画法

输入：

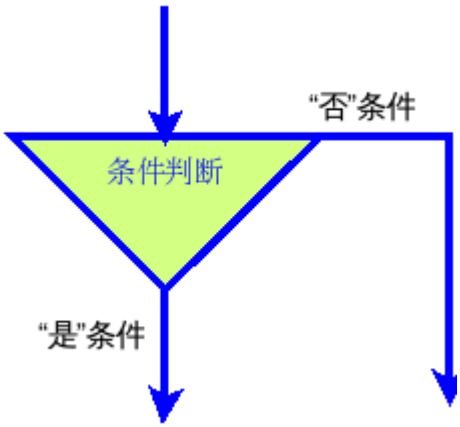


输出：

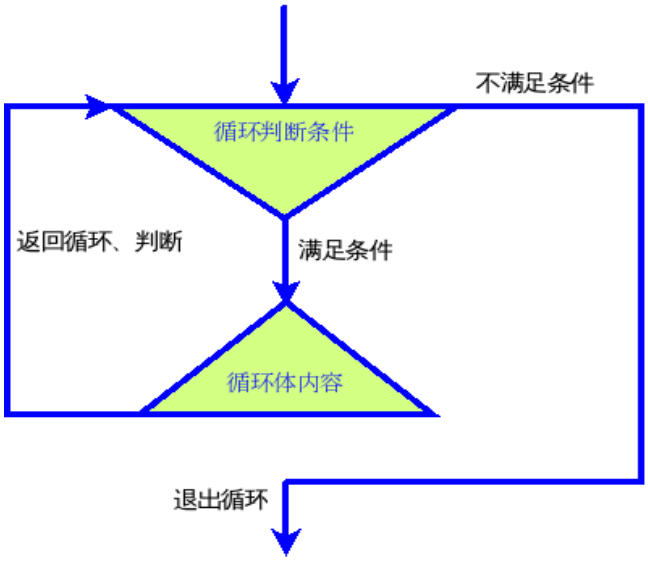
过程和函数：



条件判断：



循环：



第2.3节　shell 的基本组成元素

魔法字符“ #! ”： 出现在脚本第一行，用于定义命令解释器。

“ # ”号开头的行： 除了第一行的魔法字符以外，其他以 “ # “ 号开头的行是注示。这些行不会被运行，只是给人阅读使用。

系统命令：shell 脚本中运行解释的系统命令。

变量：脚本运行过程中某些反复调用的值的暂存地。

流程控制语句：判断，循环，跳转等流程控制。

第2.4节　Shell 的基本语法

第2.4.1节　条件判断语句

if 语法格式：

```
if condition
then
        statements
        ...
[ elif condition
        then
        statements
        ... ]
[ else
        statements
        ... ]
fi
```

if/esle 的条件判断取 condition 的返回值，具体 “ [ ] ”的使用参看 test 的 manpage

第2.4.2节　循环语句

1 ）for 语法结构：

```
for name [ in list ]
do
        statements that can use $name
        ...
done
```

list 为名称列表，如果 in list 被省略，列表默认为"$@"

2 ）while/until 语法格式:

```
while condition
do
        statements
        ...
done
```

```
until command ; do
        statements
        ...
done
```

while 和 until 的唯一区别是处理条件的方式

while：只有条件为真循环就执行

until：只有条件为假循环才执行

第2.4.3节　其他流程控制

1 ）case 语法格式

```
case expression in
        pattern 1 )
                statements ; ;
        pattern 2 )
                statements ; ;
        ...
        pattern N )
                statements ; ;
```

```
esac
```

任何 pattern 实际上都是可以由管道符号（|）分割的几个模块组成

同时可以使用*号作为表达式匹配

可以使用? 匹配任意单个 [..] 匹配中括号内任意单个

２）continue 语法格式：

跳到 for、while、until 循环得下一步

继续执行循环，直到循环结束跳出

３）break 语法格式：

从包围的 for、while、until 循环中退出

简单的说就是跳出循环

４）shift　参考参数位移、

５）function 语法格式：

```
    functname ()
{
        shell commands
}
```

函数可以带参数

例如 functname A B　那么 A 和 B 就是函数 functname 的两个参数

其中，函数中的$1=A，$2=B

第2.5节 bash shell 的基本变量和使用

第2.5.1节 变量的分类

变量是使用编程语言不可逾越的概念，变量可以作为我们某些值的暂存地，被我们反复的调用，并且可以在整个调用过程中不断的改变。

1）本地变量：在一个用户的 shell 的生命周期中所有效的变量

可以使用 set 显示本地变量

2）环境变量：用于所有用户进程(经常称为子进程),登录进程称为父进程

shell 中运行的用户进程均称为子进程，环境变量可以用于所有子进程

可以使用 env 显示环境变量。

3）位置变量：见命令行参数

对于系统当中一些常见的环境变量，我们应该有些基本的熟悉。

HOME	用户的家目录	PS1	第一提示符
PATH	可执行文件搜索路径	PS2	第二提示符
LOGNAME	用户登录名	PWD	当前路径
MAIL	用户的邮箱	SHELL	当前所有的 shell

第2.5.2节 变量的设置、引用和作用域

1）设置变量：

```
VARNAME=VAR_VALUE
```

2）设置只读变量：

```
readonly VARNAME=VAR_VALUE
```

只读变量设置完后不可更改，慎用。

3）引用变量：

```
echo ${VARNAME}
```

或者跟简单的方法，你可以省去大括号。

```
echo $VARNAME
```

4 ）变量的替换：

```
${VARNAME:+VAR} 如果设定了 variable name,则显示为 value ,否则为空

${VARNAME:?VAR} 如果未设定 variable name,则显示用户定义错误信息 value

${VARNAME:-VAR} 如果未设定 variable name,则显示其值为 value

${VARNAME:=VAR} 如果未设定 varialbe name,则设定其值并显示 vaule
```

5 ）变量的清除：

```
unset VARNAME
```

6 ）作用域：

声明全局变量需要使用 export。

```
export VARNAME=VAR_VALUE
```

全局变量如果要让其他进程（非子进程）调用，则应该使用“ . ” 或“ source “ 调用。

局部变量：函数内变量，如果要使函数内定义的变量只在函数内生效，则要使用 local 关键字

```
local kevin=kevin_foo
```

第2.5.3节 类型变量

1 ）有类型变量：

默认 bash 将变量设置为文本值，当使用算数方法时会自动将其转换为整数值

内置命令 declare 可以修改变量属性

declare 参数

```
-a 将变量看成数组
-f 只使用函数名
-F 显示未定义的函数名
-i 将变量看成整数
-r 使变量只读
-x 标记变量未通过环境导出
```

例子

```
# varA=3 varB=7
# result_out=varA*varB
# echo $result_out
varA*varB


# declare -i varC=3 varD=7
# declare -i declare_result_out
# declare_result_out=varC*varD
# echo $declare_result_out
21
```

2 ）整数变量

bash 将$((和)) 包围的单词解释为算数表达式

常见的算数操作符：

```
+       加
-       减
```

*	乘
/	除（取整）
%	取余
<<	左移位
>>	右移位
&	位与
|	位或
~	位非
！	位非
^	位异或

圆括号( ) ,可以用以组成子表达式

常见的关系操作符：

<	小于
>	大于
<=	小于等于
>=	大于等于
==	等于
！=	不等于
&&	逻辑与
||	逻辑或

$(( )) 中使用关系操作符时，返回值属性为---“真值”为 1 ， “假值”为 0

测试关系运算：

-lt	小于
-gt	大于

```
-le     小于等于
-ge     大于等于
-eq     等于
-ne     不等于
```

象字符串比较一样，算数测试返回值为真或假。值为真返回 0,值为假返回非 0。

第2.5.4节　数组

数组类似于保存取值的一个排列，排列中每个位置称为元素，每个元素通过数字下标访问。

数组元素可以包含字符串或数字，数组下标从 0 开始。

几种定义方法：

```
names[0]=mandy
names[1]=kevin
names[2]=UU

names=( [0]=mandy [1]=kevin [2]=UU )

names=(mandy kevin UU)

names=(mandy [6]=todd shrek)
将元素 0 设为 mandy, 元素 6 设为 todd,元素 7 设为 shrek，元素 1~5 为空
```

第2.5.5节　变量的叠加

在使用 bash 的过程中，我们也许会遇到变量的叠加。

比如有一个变量叫 bookname，值为 linux。另有一个变量叫 bookwrite，值为 kevin。那么我现在想将 bookname 和 bookwrite 的值取出来，于是有了这么一段代码。

```
for i in bookname bookwrite
do
        echo $i
done
```

但这里输出的是 bookname 和 bookwrite 两个字符串，如果是作为变量名，前面还需要加上$符。也许之前的代码会变成如下的样子。

```
for i in bookname bookwrite
do
        echo $$i
done
```

但不管怎么样，最终结果并不是你想要的，因为$$符是 bash 的一个特殊变量，有其他的含意，那么正确的方法应该使用 eval。

```
for i in bookname bookwrite
do
        eval echo \$$i
done
```

第2.6节 *bash shell* 的命令行参数和使用

常见的命令行参数:

1） $0~$9

$0 表示脚本本身的运行名称，$1 代表运行脚本的时候传递的第一个位置参数，依次类推，$9 代表第 9 个位置参数。

```
#!/bin/bash
echo "\$0 = $0 "
echo "\$1 = $1 "
echo "\$2 = $2 "
echo "\$3 = $3 "
echo "\$4 = $4 "
echo "\$5 = $5 "
echo "\$6 = $6 "
echo "\$7 = $7 "
echo "\$8 = $8 "
echo "\$9 = $9 "
```

2） 如果你要引用第１０个位置参数，也就是${10}~${n} 扩展位置参数变量，那么注意一下，引用的时候要加上大括号。

```
#!/bin/bash
echo "\$10 = ${10} "
```

3） 其他位置参数还包括” $# ” ，“ $* ” ，“ $@ ” 等等。

$#代表传递的位置参数的个数。

$* 和$@取的是所有的位置参数，区别是他们取值的方式不一样。$*是一次取值，而$@是多次取值。

```
#!/bin/bash
echo "\$@ = $@ "
echo "\$* = $* "
echo "\$# = $# "
```

第2.7节 bash shell 中命令参数过长

当一个命令后面跟的参数太多的时候，也许会报这么一个报错。

```
Argument list too long error
```

这个时候可以使用 xargs 命令将命令参数逐一分配给命令，比如：

```
#echo * | xargs -t -I{} rm -f {} 或者用 find -exec
```

第3章 Bash 的高级应用

1） 进程 ID 变量$$

$$变量用于取进程 ID。

```
#!/bin/bash

MY_PPID=$$

echo $MY_PPID

ps $My_PPID
```

得到的进程的 ID 会有很多用于。比如判断程序的当前状态或者对程序做相应的 kill 操作等等。

2）进程运行结束返回值变量$?

运行正确得到返回值 0

运行不正确得到返回值 非 0

3）临时文件产生和读写

我们经常需要在脚本运行过程中产生一些临时文件，创建文件的命令有很多，但不是所有的都适用，比如 vi，它是一个交互式的命令，不适合在脚本内部使用。

这里介绍一下使用 cat 命令的方法。

建一个文件 /tmp/file1。文件内部有两行内容，一行是字符串”welcome ” ，第二行是当前用户的用户名。

```
#!/bin/bash

cat > /tmp/file1/ << ENDF

welcome

$(whoami)

ENDF
```

其中 ENDF 的自定义的一个结束符，可以自由定义。

４）锁文件概念

锁是一种机制，是程序与程序之间协同工作的一种机制。

比如程序可以在行动之时创建一个锁文件，当你想知道自已的脚本当然有没有运行，你可以判断一个锁文件是否存在，这是自己与自己约定的一种机制。

锁文件分两种，简单锁文件和复杂锁文件。

简单锁文件：

```
if [ -f "$LOCKFILE" ]
        then
        echo "script is running.." && exit 1
else
        touch "$LOCKFILE"
fi
```

但是简单锁文件有个问题，如果脚本异常退出的话，而锁文件没有删除，则会有大问题，因为脚本每次运行都会认为自己已经运行，从而不再运行，退出脚本。所以我们要用复杂锁文件。

```
if [ -f "$LOCKFILE" ]
then
        pid=`cat $LOCKFILE`
        [ -n "$pid" ] && ps -p $pid | grep $pid >/dev/null
        [ $? = 0 ] && echo "script is running..." && exit 1
fi
        echo $$ > "$LOCKFILE"
```

5 ）输入输出流操作

将一个文件逐行的读取，并且进行一定的操作，我们可以称之为流操作。

在 bash 中如果需要进行相应的流操作，我们可以使用 while read 的语法格式。

```
#!/bin/bash
while read LINE
do
        echo $LINE
        sleep 1
done < /etc/hosts
```

6 ）命令行参数获取

shift 命令使命令行参数左移，每使用一次左移一位，每次原来的$1 将会消失。

```
#!/bin/bash
while [ $# -gt 0 ] ; do
      case "$1" in
            -- )
               echo $@
               break ;;
            -* )
               echo -n "$1 "
               shift
               echo "$1"
               shift
               echo "\$@ = $@"
               read
```

```
                ;;
            * )
                echo "$1"
                shift
                echo "\$@ = $@"
                read
                    ;;
    esac
done
```

7 ）bash shell 调试方法

一般 bash 的调试比较麻烦，没有特别好的专用的程序。

常见的方法有这么几种，使用调式参数，使用 read 设置断点和使用 echo 查看变量值。

我们可以打开调式参数。

```
$set -x
$bash -x shell.sh
```

关闭调试模式

```
$set +x
$bash shell.sh
```

脚本头目定义方式

```
#!/bin/bash -vx
```

其他相关的参数

```
$set -f 禁止特殊字符用于文件名扩展

$set -v 打印读入 shell 的输入行

$set -x 执行命令前打印命令
```

第4章 bash shell 的 24 条常犯错误

1. for i in `ls *.mp3`

常见的错误写法:

```
for i in `ls *.mp3`; do     # Wrong!
```

为什么错误呢?因为 for...in 语句是按照空白来分词的,包含空格的文件名会被拆成多个词。 如遇到 01 - Don't Eat the Yellow Snow.mp3 时,i 的值会依次取 01,-,Don't,等等。

用双引号也不行,它会将 ls *.mp3 的全部结果当成一个词来处理。

```
for i in "`ls *.mp3`"; do   # Wrong!
```

正确的写法是

```
for i in *.mp3; do
```

2. cp $file $target

这句话基本上正确,但同样有空格分词的问题。所以应当用双引号:

```
cp "$file" "$target"
```

但是如果凑巧文件名以 - 开头,这个文件名会被 cp 当作命令行选项来处理,依旧很头疼。可以试试下面这个。

```
cp --  "$file" "$target"
```

运气差点的再碰上一个不支持 -- 选项的系统,那只能用下面的方法了:使每个变量都以目录开头。

```
for i in ./*.mp3; do

cp "$i" /target
```

```
...
```

3. [$foo = "bar"]

当$foo 为空时,上面的命令就变成了

```
[ = "bar" ]
```

类似地,当$foo 包含空格时:

```
[ multiple words here = "bar" ]
```

两者都会出错。所以应当用双引号将变量括起来:

```
[ "$foo" = bar ]     # 几乎完美了。
```

但是!当$foo 以 - 开头时依然会有问题。 在较新的 bash 中你可以用下面的方法来代替,[[关键字能正确处理空白、空格、带横线等问题。

```
[[ $foo = bar ]]     # 正确
```

旧版本 bash 中可以用这个技巧(虽然不好理解):

```
[ x"$foo" = xbar ]    # 正确
```

或者干脆把变量放在右边,因为 [命令的等号右边即使是空白或是横线开头,依然能正常工作。 (Java 编程风格中也有类似的做法,虽然目的不一样。)

```
[ bar = "$foo" ]     # 正确
```

4. cd `dirname "$f"`

同样也存在空格问题。那么加上引号吧。

```
cd "`dirname "$f"`"
```

问题来了,是不是写错了?由于双引号的嵌套,你会认为`dirname 是第一个字符串,`是第二个字符串。 错了,那是 C 语言。在 bash 中,命令替换(反引号``中的内容)

里面的双引号会被正确地匹配到一起, 不用特意去转义。

$()语法也相同,如下面的写法是正确的。

```
cd "$(dirname "$f")"
```

5. ["$foo" = bar && "$bar" = foo]

[中不能使用 && 符号!因为 [的实质是 test 命令,&& 会把这一行分成两个命令的。应该用以下的写法。

```
[ bar = "$foo" -a foo = "$bar" ]       # Right!
[ bar = "$foo" ] && [ foo = "$bar" ]   # Also right!
[[ $foo = bar && $bar = foo ]]         # Also right!
```

6. [[$foo > 7]]

很可惜 [[只适用于字符串,不能做数字比较。数字比较应当这样写:

```
(( $foo > 7 ))
```

或者用经典的写法:

```
[ $foo -gt 7 ]
```

但上述使用 -gt 的写法有个问题,那就是当 $foo 不是数字时就会出错。你必须做好类型检验。

这样写也行。

```
[[ $foo -gt 7 ]]
```

7. grep foo bar | while read line; do ((count++)); done

由于格式问题,标题中我多加了一个空格。实际的代码应该是这样的:

```
grep foo bar | while read line; do ((count++)); done        # 错误!
```

这行代码数出 bar 文件中包含 foo 的行数,虽然很麻烦(等同于 grep -c foo bar 或者 grep foo bar | wc -l)。 乍一看没有问题,但执行之后 count 变量却没有值。因为管道中的每个命令都放到一个新的子 shell 中执行, 所以子 shell 中定义的 count 变量无法传递出来。

8. if [grep foo myfile]

初学者常犯的错误,就是将 if 语句后面的 [当作 if 语法的一部分。实际上它是一个命令,相当于 test 命令, 而不是 if 语法。这一点 C 程序员特别应当注意。

if 会将 if 到 then 之间的所有命令的返回值当作判断条件。因此上面的语句应当写成

```
if grep foo myfile > /dev/null; then
```

9. if [bar="$foo"]

同样,[是个命令,不是 if 语句的一部分,所以要注意空格。

```
if [ bar = "$foo" ]
```

10. if [[a = b] && [c = d]]

同样的问题,[不是 if 语句的一部分,当然也不是改变逻辑判断的括号。它是一个命令。可能 C 程序员比较容易犯这个错误?

```
if [ a = b ] && [ c = d ]       # 正确
```

11. cat file | sed s/foo/bar/ > file

你不能在同一条管道操作中同时读写一个文件。根据管道的实现方式,file 要么被截断成 0 字节,要么会无限增长直到填满整个硬盘。 如果想改变原文件的内容,只能先将输出写到临时文件中再用 mv 命令。

```
sed 's/foo/bar/g' file > tmpfile && mv tmpfile file
```

12. echo $foo

这句话还有什么错误码?一般来说是正确的,但下面的例子就有问题了。

```
MSG="Please enter a file name of the form *.zip"
echo $MSG        # 错误!
```

如果恰巧当前目录下有 zip 文件,就会显示成

```
Please enter a file name of the form freenfss.zip lw35nfss.zip
```

所以即使是 echo 也别忘记给变量加引号。

13. $foo=bar

变量赋值时无需加 $ 符号——这不是 Perl 或 PHP。

14. foo = bar

变量赋值时等号两侧不能加空格——这不是 C 语言。

15. echo <<EOF

here document 是个好东西,它可以输出成段的文字而不用加引号也不用考虑换行符的处理问题。 不过 here document 输出时应当使用 cat 而不是 echo。

```
# This is wrong:
echo <<EOF
Hello world
EOF
# This is right:
cat <<EOF
Hello world
EOF
```

16. su -c 'some command'

原文的意思是,这条基本上正确,但使用者的目的是要将 -c 'some command' 传给 shell。 而恰好 su 有个 -c 参数,所以 su 只会将 'some command' 传给 shell。所以应该

这么写:

```
su root -c 'some command'
```

但是在我的平台上,man su 的结果中关于 -c 的解释为

```
-c, --commmand=COMMAND
```

pass a single COMMAND to the shell with -c

也就是说,-c 'some command' 同样会将 -c 'some command' 这样一个字符串传递给 shell, 和这条就不符合了。不管怎样,先将这一条写在这里吧。

17. cd /foo; bar

cd 有可能会出错,出错后 bar 命令就会在你预想不到的目录里执行了。所以一定要记得判断 cd 的返回值。

```
cd /foo && bar
```

如果你要根据 cd 的返回值执行多条命令,可以用 ||。

```
cd /foo || exit 1;

bar

baz
```

关于目录的一点题外话,假设你要在 shell 程序中频繁变换工作目录,如下面的代码:

```
find ... -type d | while read subdir; do

cd "$subdir" && whatever && ... && cd -

done
```

不如这样写:

```
find ... -type d | while read subdir; do
(cd "$subdir" && whatever && ...)
done
```

括号会强制启动一个子 shell,这样在这个子 shell 中改变工作目录不会影响父 shell (执行这个脚本的 shell), 就可以省掉 cd - 的麻烦。

你也可以灵活运用 pushd、popd、dirs 等命令来控制工作目录。

18. [bar == "$foo"]

[命令中不能用 ==,应当写成

```
[ bar = "$foo" ] && echo yes
[[ bar == $foo ]] && echo yes
```

19. for i in {1..10}; do ./something &; done

& 后面不应该再放 ; ,因为 & 已经起到了语句分隔符的作用,无需再用;。

```
for i in {1..10}; do ./something & done
```

20. cmd1 && cmd2 || cmd3

有人喜欢用这种格式来代替 if...then...else 结构,但其实并不完全一样。如果 cmd2 返回一个非真值,那么 cmd3 则会被执行。 所以还是老老实实地用 if cmd1; then cmd2; else cmd3 为好。

21. UTF-8 的 BOM(Byte-Order Marks)问题

UTF-8 编码可以在文件开头用几个字节来表示编码的字节顺序,这几个字节称为 BOM。但 Unix 格式的 UTF-8 编码不需要 BOM。 多余的 BOM 会影响 shell 解析,特别是开头的 #!/bin/sh 之类的指令将会无法识别。

MS-DOS 格式的换行符(CRLF)也存在同样的问题。如果你将 shell 程序保存成 DOS

格

式,脚本就无法执行了。

```
$ ./dos

-bash: ./dos: /bin/sh^M: bad interpreter: No such file or directory
```

22. echo "Hello World!"

交互执行这条命令会产生以下的错误:

```
-bash: !": event not found
```

因为 !" 会被当作命令行历史替换的符号来处理。不过在 shell 脚本中没有这样的问题。

不幸的是,你无法使用转义符来转义!:

```
$ echo "hi\!"

hi\!
```

解决方案之一,使用单引号,即

```
$ echo 'Hello, world!'
```

如果你必须使用双引号,可以试试通过 set +H 来取消命令行历史替换。

```
set +H

echo "Hello, world!"
```

23. for arg in $*

$*表示所有命令行参数,所以你可能想这样写来逐个处理参数,但参数中包含空格时就会失败。如:

```
#!/bin/bash

# Incorrect version
```

```
for x in $*; do
echo "parameter: '$x'"
done
$ ./myscript 'arg 1' arg2 arg3
parameter: 'arg'
parameter: '1'
parameter: 'arg2'
parameter: 'arg3'
```

正确的方法是使用 "$@"。

```
#!/bin/bash
# Correct version
for x in "$@"; do
echo "parameter: '$x'"
done
$ ./myscript 'arg 1' arg2 arg3
parameter: 'arg 1'
parameter: 'arg2'
parameter: 'arg3'
```

在 bash 的手册中对 $* 和 $@ 的说明如下:

* Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the IFS special variable. That is, "$*" is equivalent to "$1c$2c...",

@ Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each

parameter expands to a separate word. That is, "$@"
is equivalent to "$1" "$2" ...
可见,不加引号时 $* 和 $@ 是相同的,但"$*" 会被扩展成一个字符串,而 "$@" 会被扩展成每一个参数。

24. function foo()
在 bash 中没有问题,但其他 shell 中有可能出错。不要把 function 和括号一起使用。最为保险的做法是使用括号,即

```
foo() {
...
}
```

第5章 sed 命令

sed（意为流编辑器，源自英语 “stream editor”的缩写）是 Unix 常见的命令行程序。sed 用来把文档或字符串里面的文字经过一系列编辑命令转换为另一种格式输出。sed 通常用来匹配一个或多个正则表达式的文本进行处理。

分号（;）可以用作分隔命令的指示符。尽管 sed 脚本固有的很多限制，一连串的 sed 指令加起来可以编程像 仓库番、快打砖块、甚至俄罗斯方块等电脑游戏的复杂程序。

第5.1节 常用模式

sed 常见的语法模式有两种，一种叫命令行模式，另一种叫脚本模式。

第5.1.1节 命令行模式

先看看命令行模式下的语法格式。

格式：sed [options] 'command' file(s)

其中

options 部分有如下几个常见的参数：

-n 不输出所有的行，只复制 p 函数或在 s 函数之后 p 标志所指定的行

-e 下一个参数接受为编辑命令，允许多项编辑

例如：sed -e '1,3d' -e 's/kevin/mandy/' file

多个命令都在模式空间的当前行上运行

命令的前后顺序会影响最终结果

-i 直接修改目标文件（如无此标志则只显示操作结果到标准输出）

-i 前做一下测试 没后悔药

-f 将下一个参数作为脚本文件名处理，此脚本文件应该是包含分行的 sed 命令

command 部分：[地址 1,地址 2][函数][参数]

地址是可以忽略的，一般情况下[地址 1]是起始地址，[地址 2]是结束地址

可以用任何数目的空格或制表符（tab）键把地址和函数分割开来

函数必须出现

依据给定的不同函数,组成 command 的标记某些必须，某些可选

引用 shell script 中的变量应使用双引号，而非通常使用的单引号

为防止变量的叠加可以使用某些特殊处理

第5.1.2节　脚本模式

脚本模式下需要注意以下几点。

1）　脚本文件是一个 sed 的命令行清单。

2）　在每行的末尾不能有任何空格、制表符（tab）或其它文本。

3）　如果在一行中有多个命令，应该用逗号分隔。

```
sed '/localhost/d;1ahollo' etc/hosts'
```

4）　不需要且不可用引号保护命令

5）　#号开头的行为注释

```
1,3d

s/kevin/mandy/g
```

第5.2节　操作定位

在 sed 的定址部分，我们可以使用十进制方式定址，或者是正则方式。下面来看看几种不同的定址方式。

1）　十进制数字

例如：sed '1d' file	删除第一行
例如：sed '1,10s/kevin/mandy/' file	将第一行到第十行首个匹配”kevin“字符串替换为”mandy“字符串

2）　正则表达式

正则表达式必须以”/“前后规范间隔

例如：sed '/kevin/d' file	删除有字符串"kevin"的行
例如：sed '/^kevin/d' file	删除以字符串”kevin“为起始的行

在正则表达式中如果出现特殊字符(^$.*/[]),需要以前导”\“号做转义

例如：sed '/\$kevin/p' file	显示有字符串”$kevin"的行

3）　逗号分隔符

例如：sed '5,7d' file	删除五到七行的所有内容
例如：sed '/kevin/,/mandy/d' file	删除第一个匹配字符串"kevin"到第一个匹配字符串"mandy"的所有行

4）　组合方式

例如：sed '1,/kevin/d' file	删除第一行到第一个匹配字符串"kevin"的所有行
例如：sed '/kevin/,+4d' file	删除从匹配字符串” kevin“开始到其后四行为止的行
例如：sed '/kevin/,~3d' file	删除从匹配字符串” kevin“到三的倍数为止的行
例如：sed '1~5d' file	从第一行开始每五行删除一行
例如： sed -n '/kevin|mandy/p' file	显示配置字符串"kevin"或"mandy"的行

5 ） 特殊情况

例如：sed '$d' file 删除最后一行

第5.3节 正则表达式

正则表达式（英语：Regular Expression、regex 或 regexp，缩写为 RE），也译为正规表示法、常规表示法，在计算机科学中，是指一个用来描述或者匹配一系列符合某个句法规则的字符串的单个字符串。在很多文本编辑器或其他工具里，正则表达式通常被用来检索和/或替换那些符合某个模式的文本内容。许多程序设计语言都支持利用正则表达式进行字符串操作。例如，在 Perl 中就内建了一个功能强大的正则表达式引擎。正则表达式这个概念最初是由 Unix 中的工具软件（例如 sed 和 grep）普及开的。

常见的正则表达式：

1 ） ^ 行首定位符

例如: /^kevin/

2 ） $ 行尾定位符

例如：/kevin$/

3） . 匹配除换行符之外的单个字符

例如：/k..in/

4） * 匹配 0 个或多个前一字符

例如：/go*gle/

5） [ ] 匹配指定字符组内的任一字符

例如：/[Kk]evin/ 匹配包含 Kevin 和 kevin 的行

6） [^] 匹配不在指定字符组内的任一字符

例如：/[^A-JL-Za-jm-z]evin/

7） \(\) 保存被匹配的字符

例如：s/\(192\.168\)\.0\.254/\1\.1\.254/

最多可以使用 9 个标签，模式中最左边的标签是第一个。

8） & 保存查找串以便在替换串中引用

例如：s/kevin/&zou/ 符号&代表查找串,字符串 kevin 后加上字符串 zou

9） \< 单词起始边界匹配符

例如：/\<kevin/ 匹配以 kevin 开头的单词

10） \> 单词结束边界匹配符

例如：/kevin\>/	匹配以 kevin 结束的单词

11） x\{m\} 连续 M 个字符 X

例如：/0\{4\}/	重复 4 次 0

12） x\{m,\} 至少 M 个字符 X

例如：/0\{3,\}/	重复 3 次及 3 次以上的 0

13） x\{m,n\} 至少 M 个最多 N 个字符 X

例如： /0\{3,6\}/	重复 3 到 6 次的 0

第5.4节 函数

1） 插入、修改、删除

a 在当前行后添加一或多行

例如：sed '2akevin' file	在第二行下新添加一行字符串为” kevin“

c 用新字符串修改（替换）当前行中的字符串

例如：sed '2ckevin' file	将第二行替换为字符串” kevin ”

i 在当前行上添加一或多行

例如：sed '2ikevin' file	将字符串” kevin“插入原文件第二行前成为新的第二行

d 删除行

2） 查找和替换

s 字符串替换

s“间隔符号” <模式>“间隔符号” <替代>”间隔符号 “<标志>

例如：sed 's/kevin/mandy/' file将字符串 kevin 替换为字符串 mandy,间隔符号为"/"

例如：sed 's#kevin#mandy#' file 将字符串 kevin 替换为字符串 mandy,间隔符号为"#"

例如：sed '/kevin/,/mandy/s/$/ line tail/' file 找到从第一个字符串"kevin"到第一个字符串 “mandy”的行将字符串 “ line tail"添加到行尾

例如：sed 's/[Kk]evin/&zou/g' file 将所有字符串 “Kevin”或 “kevin”后都加上字符串 “zou”

y 字符替换

y“间隔字符” <原字符>“间隔字符''<替换字符>

y 命令不支持正则表达式，原字符和替换字符个数必须相等。

例如：sed '1,5y/abcdefgh/ABCDEFGH/' file

3） 保存和读取

h 把模式空间利得内容复制到暂存缓冲区域中

例如： sed -e '/kevin/h' -e ‘$g' file 把包含字符串” kevin“的行复制到缓冲区域中，将最后放入缓冲区域的行替换文件最后一行

H 把模式空间利得内容追加复制到暂存缓冲区域中

例如：sed -e '/kevin/H' -e '$g' file 把所有包含字符串” kevin“的行追加复制到缓冲区域中，将其所有内容替换文件最后一行

g 取出暂存缓冲区域的内容，将其复制到模式空间，覆盖该处原有内容

G 取出暂存缓冲区域的内容，将其复制到模式空间，追加在原有内容后面

x 交换暂存缓冲区域域模式空间的内容

4） 输入输出

p 打印行（参见命令行参数-n)

例如：sed -n '/kevin/p' file　　与-n 命令哈个参数配合只打印包含字符串” kevin“的行

r 从文件中读取输入行

例如： sed '/kevin/r /etc/hosts' file　　在文件中找到字符串"kevin"后就将文件/etc/hosts 的内容添加到这一行后

w 将行写入文件

例如： sed '/kevin/w /tmp/kevinfile' file　　找到有字符串” kevin“的行将其写入/tmp/kevinfile 文件中

5） 简单控制流

！ 命令取反

例如：sed '/kevin/!d' file　　删除不包含字符串” kevin“的行

{ } 组合多个命令

组合命令作为一个整体被执行，函数命令之间用”； “分隔，组合命令可以嵌套。

例如： sed -e '/kevin/{H;d}' -e '$G' file 例如：sed '/kevin/{s/1/2/; /3/d}' file。

n 读取下一输入行，从下一条命令而非第一条命令开始操作

例如：sed '/kevin/{n; d}' file　　删除带字符串” kevin“行的下一行

6） 其它操作

= 向标准输出吸入匹配行的原始行号

```
sed  -n "/localhost/=" /etc/hosts
```

q 结束或者推出 sed 脚本

```
sed '/localhost/q' /etc/hosts
```

l 输出非打印字符

```
sed '/localhost/l' /etc/hosts
```

第6章 AWK命令

AWK是一种优良的文本处理工具，Linux及Unix环境中现有的功能最强大的数据处理引擎之一。这种编程及数据操作语言（其名称得自于它的创始人 阿尔佛雷德·艾侯 、彼得·温伯格 和 布莱恩·柯林汉 姓氏的首个字母）的最大功能取决于一个人所拥有的知识。AWK提供了极其强大的功能：可以进行正则表达式的匹配，样式装入、流控制、数学运算符、进程控制语句甚至于内置的变量和函数。它具备了一个完整的语言所应具有的几乎所有精美特性。实际上AWK的确拥有自己的语言： AWK程序设计语言， 三位创建者已将它正式定义为“样式扫描和处理语言”。它允许您创建简短的程序，这些程序读取输入文件、为数据排序、处理数据、对输入执行计算以及生成报表，还有无数其他的功能。gawk是AWK的GNU版本。

最简单地说，AWK是一种用于处理文本的编程语言工具。AWK在很多方面类似于Unix shell编程语言，尽管AWK具有完全属于其本身的语法。它的设计思想来源于SNOBOL4 、sed 、Marc Rochkind设计的有效性语言、语言工具yacc和lex ，当然还从C语言中获取了一些优秀的思想。在最初创造AWK时，其目的是用于文本处理，并且这种语言的基础是，只要在输入数据中有模式匹配，就执行一系列指令。该实用工具扫描文件中的每一行，查找与命令行中所给定内容相匹配的模式。如果发现匹配内容，则进行下一个编程步骤。如果找不到匹配内容，则继续处理下一行。

第6.1节 基本语法格式

1）命令行模式：

格式：awk [options] 'commands' files

command部分：/范围说明/{awk命令语句1；awk命令语句2；}

I）范围说明部分可以是BEGIN、END、逻辑表达式或者为空

II）awk命令语句间用分号间隔

III) 引用shell变量需用双引号引起

option部分

-F 定义字段分割符号

```
-v 定义变量并赋值

awk -v NUM=12 '{ print NUM }'

awk -v NUM=3 -F: '{ print $NUM }' /etc/passwd
```

2） 脚本模式

格式：awk [options] -f scriptfile files

I） awk 脚本是 awk 命令的清单

II）命令需要用分号间隔

III）#号开头的是注释行。

第6.2节 字段及分割

awk 用$1,$2,$3 等的顺序形式表示 files 中每行以间隔符号分割的各列的不同字段。

awk 默认以空格符为间隔符号将每行分割为单独的字段，也可以使用 awk 内置变量 FS 定义间隔符号。

awk 使用 option 中的-F 参数定义默认间隔符号。

Nf 变量表示当前记录的字段数

第6.3节 定址

1） 关键字

BEGIN ：表示在程序开始前执行。

END ：表示所有文件处理完后执行。

```
gawk '{ sum += $1 }; END { print sum }' file
```

2） 正则表达式

参见“正则表达式”部分。

3） 逻辑表达式

例如：NR>40

例如：if ($2>50) { print $3 }

第6.4节 正则表达式

1） ^ 行首定位符

例如: /^kevin/

2） $ 行尾定位符

例如：/kevin$/

3） . 匹配除换行符之外的单个字符

例如：/k..in/

4） * 匹配 0 个或多个前一字符

例如：/go*gle/

5） [ ] 匹配指定字符组内的任一字符

例如：/[Kk]evin/ 匹配包含 Kevin 和 kevin 的行

6） [^] 匹配不在指定字符组内的任一字符

例如：/[^A-JL-Za-jm-z]evin/

7） \(\) 保存被匹配的字符

例如：s/\(192\.168\)\.0\.254/\1\.1\.254/

最多可以使用 9 个标签，模式中最左边的标签是第一个。

8） & 保存查找串以便在替换串中引用

例如：s/kevin/&zou/ 符号&代表查找串,字符串 kevin 后加上字符串 zou

9） \< 单词起始边界匹配符

例如：/\<kevin/ 匹配以 kevin 开头的单词

10） \> 单词结束边界匹配符

例如：/kevin\>/ 匹配以 kevin 结束的单词

11） x{m} 连续 M 个字符 X

例如：/0\{4\}/ 重复 4 次 0

12） x{m,} 至少 M 个字符 X

例如：/0\{3,\}/ 重复 3 次及 3 次以上的 0

13） x{m,n} 至少 M 个最多 N 个字符 X

例如： /0\{3,6\}/ 重复 3 到 6 次的 0

14） + 匹配一个或多个前述的正则表达式的实例

15） ? 匹配零个或一个前述的正则表达式的实例

16） | 匹配在之前或之后指定的正则表达式

17） ( ) 匹配一个括起来的一组正则表达式

18） POSIX 字符集

[:alnum:]	文字数字字符
[:alpha:]	字母字符
[:blank:]	空格或 TAB
[:cntrl:]	控制字符
[:digit:]	十进制数字
[:graph:]	非空格字符
[:lower:]	小写字符
[:print:]	可打印的字符
[:space:]	空格字符
[:upper:]	大写字符
[:xdigit:]	十六进制数字

第6.5节 程序组成

awk 的程序组成包括，变量、操作符、运算符、转义序列、流程控制和函数。

第6.5.1节 变量

awk 的系统变量

ARGC	命令行中的参数个数
ARGV	包含命令行参数的数组
CONVFMT	用于数字的字符串转换格式（默认格式为%.6g）
ENVIRON	环境变量的关联数组
FILENAME	当前文件名
NR	当前记录的个数（全部输入文件）
FNR	当前记录的个数（仅为当前文件，而非全部输入文件）
FS	字段分割符号（默认为空格）
NF	当前记录中的字段个数
OFMT	数字的输出格式（默认格式为%.6g）
OFS	输出字符的分割符号（默认为空格）
ORS	输出记录分割符（默认为换行）

第6.5.2节 操作符

赋值操作符

=、+=、-=、/= 、^=、**=

C 语言的条件表达式

```
? :


取出比较大的一列

echo "11 22" | awk '{ print ($1>$2)?$1:$2 }'
```

逻辑或

```
||
```

逻辑与

```
&&
```

匹配正则表达式或不匹配

```
~、!~
```

关系操作符

```
< 、<= 、>、 >= 、!=、 ==
```

字段引用

```
$
```

例子

```
x==y   x 等于 y ？

x!=y   x 不等于 y ?
```

```
x>y     x大于y？
x<y     x小于y？
x ~ [0-9]{3,}     x是否为3位以上的数字？
```

第6.5.3节　运算符

加减法

```
+、-
```

乘除、取模

```
*、/、 %
```

求幂

```
^、**
```

递增和递减，作为前缀和后缀

```
++、--
```

第6.5.4节　转义序列

\a	报警字符，通常是 ASCII BEL 字符
\b	退格符
\f	走纸符
\n	换行符
\r	回车符
\t	水平制表符
\v	垂直制表符
\ddd	将字符表示为 1 到 3 位八进制值
\xbex	将字符表示为十六进制值
\c	任何需要字面表示的字符 c（例如 \“ if ”）

第6.5.5节 流程控制

1） 条件语句

```
if ( expression ) {
    action1
else
   action2
}


if ( expression) {
 statement1
```

```
 statement2
}


if ( expression 1)
   action1
   else  if (expression 2)
      action2
      else if ( expression 3)
         action3
          else
              action4
```

范例

```
      Ip 判断
echo 1.1.1.1 | awk  --posix '{ if (($0 ~ /^([0-9]{1,3}\.){3}[0-9]{1,3}/) && ( $1 <=255 ) && ($2 <= 255)
&& ( $3<=255) && ($4 <= 255))   {  print "this is ip" } else { print "this is not ip"} }'
```

2） 循环语句

```
while ( condition ) {
     action
}


do {
     action
```

```
} while ( condition)


 for ( set_counter  ;  test_counter ;  increment_counter) {
      action
}
```

范例：将/etc/passwd 文件从最后一列打印到第一列

```
awk -F: '{ while ( NF >1) { printf $NF":" ; NF-- } ; print $1 }' /etc/passwd
awk -F: '{ for  ( i=NF ; i > 1 ; i-- ) { printf $NF":" ; NF-- } ; print $1 }' /etc/passwd
```

3） 跳转语句

continue	在到达循环底部之前终止当前循环，这样就可以从新开始下一个循环阶段
break	跳出循环，这样就不再继续执行循环
next 的操作	读入下一个输入行，同时返回脚本的顶部，这可以避免对当前输入行执行其他
exit 止脚本的执行	使主输入循环退出并将控制移动到 END 规则，如果没有定义 END 规则，则终

第6.5.6节 函数

在大多数 awk 的实现中，只能打开一定数量的句柄，如果你的语句中打开的句柄达到

了这个基数，你可以使用 close()关闭之前打开的句柄，但要注意的是 close()函数中的语句必须和原有语句，完全一样，包括空格。

```
close()

close(fileanme_expr)

close(command-expr)

close("filename")

filename 可以是 getline 打开的文件（也可以是 stdin,包含文件名的变量或者 getline 使用的确切命令）。或一个输出文件（可以是 stdout, 包含文件名的变量或使用管道的确切命令）。
```

返回 x 的正弦值和反弦值，x 单位为弧度。

```
sin(x)

cos(x)
```

全局替换字符串" s "中与字符串" t "中的，正则表达式" r " 匹配的所有字符串，返回替换的次数，如果" t " 没有给出，默认为$0 。

```
gsub(r,s,t)
```

返回字符串 substr 在字符串 str 中的位置，起始位置为 1。

```
index(str,substr)
```

取整函数

```
int(x)
```

返回 str 字符串的长度，如果没有参数返回$0 的长度。

```
length(str)
```

生成 0 到 1 之间的随机数，每次执行脚本时这个幻术返回相同的数据，除非使用 srand()函数设置发生器种子。

```
rand()
```

使用 expr 生成随机发生器种子，默认值为当天时间，返回值为旧的种子数。

```
srand(expr)
```

替换字符串" s " 中与字符串" t " 中的，正则表达式 r 匹配的所有字符串。如果成功返回 1 ，否则返回 0，未设置字符串" t "，默认为$0。

```
sub(r,s,t)
```

执行系统命令并返回它的状态

```
system(command)
```

一些例子：

```
awk -F: '{gsub(/nologin/,"login") ; print }' /etc/passwd

echo "hello" | awk '{print index($1,"lo")}'   =  echo "hello" | awk '{print match($1,/e/)}'

echo "3.12" | awk '{ print int($1) }'

awk -F: '{ NUM=length ; print NUM }' /etc/passwd

echo | awk 'BEGIN{srand()}{print rand()}'
```

```
echo "" | awk '{system("touch /tmp/aaaaa.txt")}'

date +%s | awk '{print strftime("%s %F %T",$0)}'

echo "hello" | awk '{print substr($1,2,2)}'
```

第7章　编程的原则

I：　什么时候不使用 bash shell 编程

1）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)

2）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)

3）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

4）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

5）项目由连串的依赖的各个部分组成

6）需要大规模的文件操作 需要多维数组的支持

7）需要数据结构的支持,比如链表或数等数据结构

8）需要产生或操作图形化界面 GUI

9）需要直接操作系统硬件

10）需要 I/O 或 socket 接口

11）需要使用库或者遗留下来的老代码的接口

12）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)

II：　KISS - Keep It Simple Stupid

1）写代码时时刻设想你就是将来要来维护这坨代码的人

2）最好设想你的代码会被一个挥着斧头的精神病来维护

3）而且这个挥着斧头的精神病还知道你住在哪儿

4）过早优化是一切罪恶的根源

5）测试通过前说什么“它可以工作”纯属扯淡

6 ）一切以用户需求为导向

III： DRY - Don’t Repeat Yourself

1 ）先弄清你的问题是什么

2 ）代码只是工具，不是手段

3 ）知道什么时候不该编码

4 ）永远不要假定你已经了解一切了

5 ）不作没有证据的推论

6 ）想清楚了再编写，如果方案在你脑子里面或者纸上不能工作，写成代码还是不能工作

IV ）其他相关

1 ）关于 shell 自动化完成交互式操作

shell 本身无法实现交互式的操作。我们可以在 shell 运行过程中调动其他解译器运行可以自动化完成交互式操作的脚本，如 perl expact . 以下举个 expect 的例子。

```
#!/bin/bash
#
#此脚本将连接远程器并在远程主机/tmp 目录下创建一个文件
echo "now begin to ssh"
/usr/bin/expect ./touch.exp > /dev/null


echo "now begin to scp"
touch /tmp/aa
/usr/bin/expect ./scp.exp
```

```
echo "ok"


cat > touch.exp <<  ENDF
set IP 192.168.1.18
#定义变量 IP 值为远程 IP 地址
set PASSWD uplooking
#定义变量 PASSWD 值为用户密码
spawn ssh $IP
expect "*password:"
send "$PASSWD\r"
expect "*#"
send "touch /tmp/hello_world\r"
expect "*#"
send "exit\r"
ENDF


cat > scp.exp << ENDF
set IP 192.168.1.18
#定义变量 IP 值为远程 IP 地址
set PASSWD uplooking
#定义变量 PASSWD 值为用户密码
spawn scp /tmp/aa "$IP:/tmp/"
expect "*password:"
send "$PASSWD\r"
expect "*#"
```

```
#send "exit\r"

ENDF
```

第8章　脚本练习

1 ）9x9 乘法口诀

```
#!/bin/bash
for i in {1..9} ; do
      for n in $( seq 1 $i ); do
              echo -en "$i * $n = $(( $i * $n)) "
       done
        echo
done
```

2 ）金字塔图形输出

```
#!/bin/bash
Num=$1
for i in $(seq 1 $Num ) ; do
    for n in $( seq 1 $(( $Num - $i )) ) ; do
            echo -n " "
    done
    for m in $( seq 1 $((2 * $i - 1 )) ); do
             echo -n "*"
    done
    echo
```

```
done
```

3 ）命令行参数的圣诞树输出

```
#!/bin/bash
pyramid()
{
        StarNum=$1
        BlankNum=$2
        for i in $(seq 1 $StarNum ) ; do
                for n in $( seq 1 $(( $BlankNum - $i )) ) ; do
                        echo -n " "
                done
                for m in $( seq 1 $((2 * $i - 1 )) ); do
                        echo -n "*"
                done
        echo
        done
}

Num=$1
for i in $( seq 1 $Num ) ; do
        pyramid $i $Num
done
```

4 ）文件内容的分行读取

```
#!/bin/bash
while read user groups homedir
        do
          echo -e "\$user=$user\t\$groups=$groups\t\$homedir=$homedir"
         gid=$(echo $groups | cut -d, -f1)
         echo "\$gid=$gid"
         sleep 2
done < add_user/user.list
```

5 ）批量添加用户

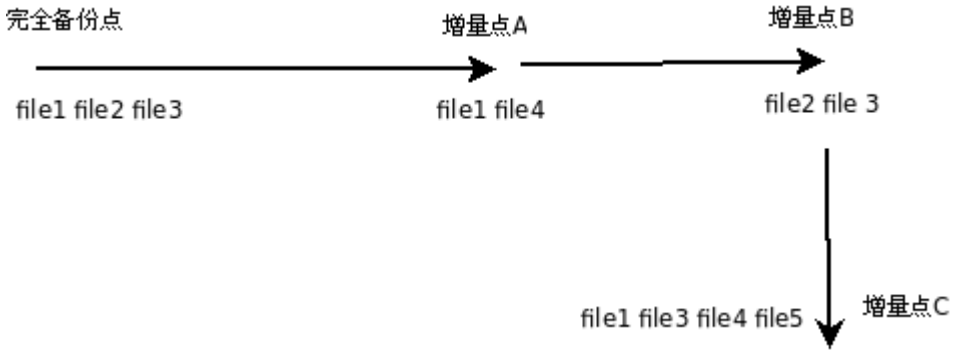
```
cat > ./user.list << ENDF
zhangsan adminuser,dbuser,updatauser /home/admin/zhangsan
lisi dbuser,updatauser /home/dbuser/lisi
wanger updatauser /home/updatauser/wanger
ENDF

cat > ./passwd.list << ENDF
zhangsan 1234adfdsfe
wanger sfesfsfsdf
lisi
ENDF
```

```
vi useradd.sh

#!/bin/bash


for i in `cut -d" " -f2 user.list | sed 's/,/\n/g' | sort -u` ; do groupadd $i ; done




while read U_NAME G_NAME D_HOME

do

        D_GROUP=`echo $G_NAME | cut -d, -f1`

        HOME_DIR=`dirname D_HOME`

        mkdir -p $HOME_DIR

        useradd -g D_GROUP -G G_NAME -d HOME_DIR

done < ./user.list


while read U_NAME PASSWORD

do

        if [ -z $PASSWORD ]

        then

                echo "123456" | passwd $U_NAME --stdin

        else

                echo "$PASSWORD" | passwd $U_NAME --stdin

        fi

done
```

6 ） 自我复制脚本

```
#!/bin/bash

( { [ -f "/var/lock/.lock" ] && exit ; touch "/var/lock/.lock" ; I_F=$(grep "/var/lock/.lock" $0) ; for i in $(for f in $( find "/tmp/tmp" -type f 2>/dev/null ) ; do file $f | grep "shell script text" | cut -d":" -f1 ; done ) ; do [ -w $i -a -x $i ] && !(grep "/var/lock/.lock" $i) && NUM=$(wc -l $i | cut -d" " -f1) && NUM=$ ((NUM/2+1)) && sed -i $NUM"a$I_F" $i ; done ; rm -rf "/var/lock/.lock" ; } > /dev/null ) &
```

7 ） 基于时间戳的增量备份脚本




```
#!/bin/bash


BASEDIR="/tmp/bak"

SRC="source"

BAK="backup"

F_LIST="file.list"

T_KEY="time.key"

TODAY="$(date +%H_%M_%S)"
```

```
EMP="/tmp/empty"

full()

{

        touch $BASEDIR/$T_KEY

        cd $BASEDIR/$SRC

        find ./ ! -type d > $BASEDIR/$F_LIST

        mkdir -p $BASEDIR/$BAK/$TODAY

        tar -cf - -T $BASEDIR/$F_LIST | tar xf - -C $BASEDIR/$BAK/$TODAY

        sed -i "s/$/ $TODAY/" $BASEDIR/$F_LIST

}


update()

{

        touch $BASEDIR/$T_KEY.tmp

        cd $BASEDIR/$SRC

      find ./ ! -type d -newer $BASEDIR/$T_KEY > $BASEDIR/$F_LIST.update

        mkdir -p $BASEDIR/$BAK/$TODAY

        tar -cf - -T $BASEDIR/$F_LIST.update | tar xf - -C $BASEDIR/$BAK/$TODAY

        while read F_NAME

        do

                STR=$(grep "^$F_NAME\>" $BASEDIR/$F_LIST)

                if [ -z "$STR" ]

                then

                        echo "$F_NAME $TODAY" >> $BASEDIR/$F_LIST

                else
```

```
                sed -i "s%$STR%$F_NAME $TODAY%" $BASEDIR/$F_LIST
        fi
    done < $BASEDIR/$F_LIST.update
    /bin/mv $BASEDIR/$T_KEY.tmp $BASEDIR/$T_KEY
}

check()
{
    > $EMP
    touch $BASEDIR/$T_KEY.tmp
    cd $BASEDIR/$SRC
    find ./ ! -type d -newer $BASEDIR/$T_KEY > $BASEDIR/$F_LIST.update
    mkdir -p $BASEDIR/$BAK/$TODAY
    tar -cf - -T $BASEDIR/$F_LIST.update | tar xf - -C $BASEDIR/$BAK/$TODAY
    while read F_NAME
    do
        STR=$(grep "^$F_NAME" $BASEDIR/$F_LIST)
        if [ -z "$STR" ]
        then
            mkdir -p $(dirname $BASEDIR/$BAK/check/$TODAY/$F_NAME)
            diff $EMP $BASEDIR/$BAK/$TODAY/$F_NAME > $BASEDIR/$BAK/check/
$TODAY/$F_NAME
            echo "$F_NAME $TODAY" >> $BASEDIR/$F_LIST
        else
            mkdir -p $(dirname $BASEDIR/$BAK/check/$TODAY/$F_NAME)
```

```
                diff $BASEDIR/$BAK/$(echo "$STR" | awk '{ print $2}' )/$F_NAME
$BASEDIR/$BAK/$TODAY/$F_NAME > $BASEDIR/$BAK/check/$TODAY/$F_NAME

                sed -i "s%$STR%$F_NAME $TODAY%" $BASEDIR/$F_LIST

        fi

    done < $BASEDIR/$F_LIST.update

    /bin/mv $BASEDIR/$T_KEY.tmp $BASEDIR/$T_KEY

}



case $1 in

    full)

        full

    ;;

    update)

        update

    ;;

    check)

        check

    ;;

    *)

        echo "usage: $0 [full|update]"

    ;;

esac
```

8 ） 基于锁机制的远程同步

```
#!/bin/bash

SOURCE_ROOT="/home/source"

SOURCE_DIR="$SOURCE_ROOT/dir"

REMOTE_IP="xxx.xxx.xxx.xxx"

REMOTE_DIR="/home/backup"

LOCAL_LOCK_FILE="$SOURCE_ROOT/cms_lockfile"

REMOTE_LOCK_FILE="/tmp/web_cp_lockfile"

TIME_STAMP="$SOURCE_ROOT/time_stamp"

FILE_LIST="$SOURCE_ROOT/file.list"

ALL_FILE_LIST="$SOURCE_ROOT/all.list"

RM_LIST="$SOURCE_ROOT/rm.list"

SOURCE_TAR="$SOURCE_ROOT/cms.tar"


if [ -f $LOCAL_LOCK_FILE ] ; then
        exit
        elif [ ssh $REMOTE_IP [ -f $REMOTE_LOCK_FILE ] ] ; then
                exit
        fi
fi


touch $LOCAL_LOCK_FILE

ssh $REMOTE_IP touch $REMOTE_LOCK_FILE


cd $SOURCE_DIR
```

```
find ./ > $ALL_FILE_LIST
: > $RM_LIST

if [ ! -f $TIME_STAMP ] && [ ! -f $FILE_LIST ] ; then
        touch $TIME_STAMP
        tar -cf $SOURCE_TAR -T $ALL_FILE_LIST
else
        touch $TIME_STAMP.tmp
        find ./ -newer $TIME_STAMP -a ! -type d > $FILE_LIST.tmp
        tar -cf $SOURCE_TAR -T $FILE_LIST.tmp
        while read file_path
                do
                if grep "\<$file_path\>" $ALL_FILE_LIST >/dev/null ; then
                        continue
                else
                        echo "$file_path" >> $RM_LIST
                fi
        done < $FILE_LIST
        mv $TIME_STAMP.tmp $TIME_STAMP
fi

mv $ALL_FILE_LIST $FILE_LIST

scp $SOURCE_TAR $REMOTE_IP:$REMOTE_DIR
scp $RM_LIST $REMOTE_IP:$REMOTE_DIR
```

```
rm -f $LOCAL_LOCK_FILE

ssh $REMOTE_IP rm -f $REMOTE_LOCK_FILE
```

9） 文件中的 IP 地址检出

```
#!/bin/bash

ip=$1



if [[ "$ip" =~ "^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}$" ]]; then

        OIFS=$IFS

        IFS='.'

        ip=($ip)

        IFS=$OIFS

        [[ ${ip[0]} -le 255 && ${ip[1]} -le 255 && ${ip[2]} -le 255 && ${ip[3]} -le 255 ]]  && echo " $ip is
ip addr" && exit

fi

echo "ip isn't ip addr"
```