Bash+Sed+Awk

UNIT00 Preface

1、这本书假定你没有任何关于脚本或一般程序的编程知识，但是如果你具备相关的知识，那么你将很容易就能够达到中高级的水平．．．所有这些只是 Linux 浩瀚知识的一小部分.

你可以把本书作为教材，自学手册，或者是关于 shell 脚本技术的文档,也可作为教材来讲解一般的编程概念.

2、历史修订(文档名称:文档密级:)

Version	Last updated	修订版
1.0	20120710	第一次完成
1.1		重新编排并加入…
1.2		修订了… 这一段。感谢***!

3、贡献

感谢所有鼓励我让这个艰难工作变了可能的所有朋友；

4、任何信息，错误反馈及获得新版

请 mail:zcs0237@163.com;84066652@qq.com

5、版权申明(作者呕心沥血不易，请大家支持)

本书正式出版前读者可以仔细研读和自由传阅本书电子版，但不允许私自大量印刷和销售。若有人愿意整理成册且付印者，可通过邮件或博客留言联系作者商谈出版事宜。

任何单位或个人不得将本书的任何内容以任何形式进行任何商业目的使用!

对于非法盗印版，作者将本着愚公移山的精神，孜孜不倦的与盗版者周旋，直至法律做出公正的裁决。

5、I’ve learned that under everyone’s hard shell is someone who wants to be appreciated and loved.

6、前段时间在网上看到有人说“网上的 shell 技巧很多，但都只是告诉你怎么样，不告诉你为什么”，深以为然。

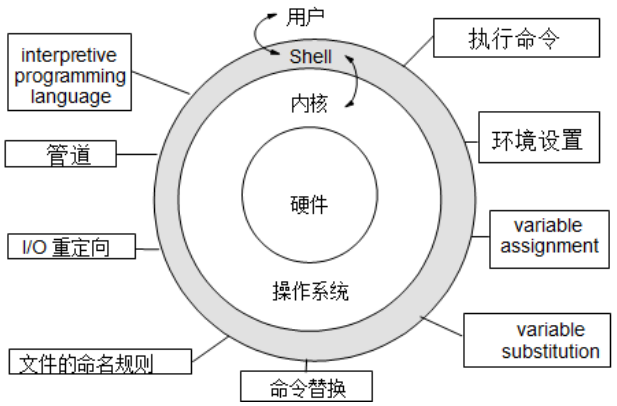
本文并不企图对 shell 作全面而系统的介绍，但也绝非零星地点到即止；而是希望通过介绍一些重要特性和提供相关参考信息，引起大家的兴趣，去深入挖掘其能力，真正把这一强大的自动化运维工具用好。

UNIT01 Shell Introduction

一、Linux 支持的 SHELL

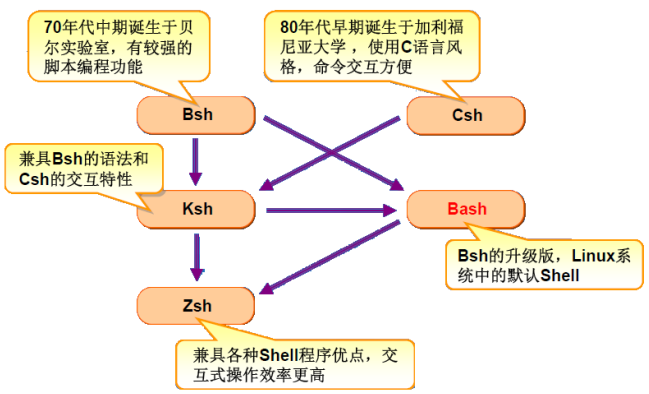
1、shell=命令解释器+脚本语言编程环境

shell 是系统维护的重要工具,便于自动批量处理大量任务。



BNU BASH 1998 年诞生, 兼容 Bourne Shell（sh）,2009 年 2 月推出 4.0,其官网为 www.gnu.org/software/bash。

2、显示支持的 shell(cat /etc/shells)



/bin/sh 已经被 bash 所取代

/bin/tcsh 整合 C Shell，提供更多的功能

3、各种 shell 对比

Shell 类别	易学性	可移植性	编辑性	快捷性
Bourne (sh)	容易	好	较差	较差

Korn (ksh)	较难	较好	好	较好
Bourne Again (bash)	难	较好	好	好
POSIX (psh)	较难	好	好	较好
C (csh)	较难	差	较好	较好
TC (tcsh)	难	差	好	好
Z (zsh)	难	差	好	好

4、不同脚本语言面向不同的任务，可能只用到其中一种或多种

通常学习 shell 做为脚本入门基础。
Perl 比 shell+sed+awk 更强大。
TCL 须专门学习 TCL 语法、功能相对强大;用的人比较少，代码比较复杂。
Expect 是 Unix 系统中用来进行自动化控制和测试的软件工具，作为 Tcl 脚本语言的一个扩展，应用在交互式软件中如 telnet，ftp，Passwd，fsck，rlogin，tip，ssh 等。
Perl 语言非常灵活，不易完全掌握；目前比较多人用，但代码公开性不强；
Python 语法简洁高效；须专门配置好运行环境；面向对象语言学习周期比较长；

5、交互式 script

一些脚本不需要来自用户的交互信息。非交互脚本的优势包括： 脚本每次都以可以预测的行为运行。 脚本可以在后台运行。
许多脚本在运行的时候给用户输出信息。交互脚本的优势在于： 可以建立更加灵活的脚本。 用户可自定义脚本使得其产生不同的行为。 脚本可以在运行过程中报告状态。
当编写交互脚本的时候，不要省略注释。用 echo 和 printf 打印适当的信息的脚本能变的更加友好且更加容易调试。 一个脚本可能做一件完美的工作，但是如果脚本不通知用户正在进行的工作，你将会得到许多来自用户的帮助请求。所以请把告诉用户等待计算完成的输出的提示信息包含进脚本。如果可能的话，尝试提醒下用户需要等待多长的时间。如果再执行某个特定任务的时候等待通常要持续很长时间，你可能会考虑把一些关于脚本输出进度的指示一起集成到脚本当中去。 当提示用户进行输入的时候，同样对输入数据的类型最好给出更多的相关信息。同样在检查参数的时候也采取同样的使用方法信息。

二、GNU Bourne Again Shell 的特点

1、可控制前台及后台运行

2、可定义许多选项及变量

shell 标记	set -f→ls /etc/*.conf——*失效 set +f→ls /etc/*.conf——*有效
shell 选项	shopt 内置命令；shopt -s 内置命令；shopt -u 内置命令 shopt -s cdspell→cd /ect——开启语法纠正 shopt -s expand_aliases——l.等别名有效 shopt -s extglob——ls *.log 中*或?有效 shopt -s nocaselob→ls install.LOG 中大写也成功

3、可支持命令 history 功能；可重新修改之前执行过的命令

三、历史记录命令

1、Fc 与 history

history(默认显示全部)	fc -l 100 110
history 9 显示最后 9 个	fc -l 9 显示>9 的命令
history -d 9 删除第 9 个	fc -ln 不带行号显示
history -c 为安全清 history	fc -lr 反序带行号
history -w 存入~/.bash_history	fc ssh 搜索 ssh 命令并编辑
history -r t.txt 读~/.bash_history	fc -s ls 搜索并执行

2、历史指令扩展的用法

!3——调用第 3 个命令	cd /tmp——!?tmp?等价于 cd /tmp
!-3——调用倒数第 3 个命令	ls /tmp/home——cd !!:$等价于 cd /home
!ssh——搜并执行 ssh 命令	ls /tmp/home——!ls :1 等价于 ls /tmp
!!——调用上一个命令	ls /var/www/index.php——!!:h 等价 ls /var/www
!!:p——调用上一命令但不执行	ls /var/www/index.php——!!:$等价 ls index.php
date;!#——等价于 date;date;	

3、可定义许变量（以 history 为例）

set|grep –i history

HISTFILE=/root/.bash_history

HISTSIZE=1000——/etc/profile 中定义~/.bash_history 的大小

HISTFILESIZE=1000

SHELLOPTS=braceexpand:emacs:hashall:histextend:history:interactive-comment:monitor

HISTSIZE=1000 /etc/profile:HISTSIZE=100 source /etc/profile 或. /etc/profile echo $HISTSIZE
histextend——历史指令扩展???????????? 调用历史指令更方便快速；调用先前的指令并安插参数；快速修正之前错误的命令。 Shopt -s history——历史指令扩展立即执行 Shopt -u history——历名指令扩展不立即执行
set +o history——关闭历史指令功能 Set -o history——打开历史指令的功能
export HISTCONTROL=ignoreboth|ignorespace|ignoredups Ignorespace——开头的空格不存入脚本 Ignoredups——连续重复的指令只存一个 Ignoreboth——结构以上历史功能
export HISTIGNORE=ls:t*:\&——不存入历史脚本中的命令 T*——t 打头的文件 &——连续重复的指令只存入一个

四、常用帮助

whatis passwd +#man [5|k|a] passwd
ls --help——用法或简短的语法总结。
info ls 比 man 更详细。可用易用的 pinfo 浏览 info page，可用箭头在链接间移动。
man page 通常提供参考信息，并不提供指导或一般使用。
/usr/share/doc:未被编入 man page 或 info page 中的新产品的文档。

五、Built-in Command List

echo $0 $$ 进程名、PID

:	空，永远返回为空　　:> list.txt
.	从当前 shell 中执行操作
break	退出 for,while,until,case 语句
cd	改变当前目录
continue	执行循环的下一步
echo	反馈信息到标准输出
eval	读取参数，执行结果命令
exec	执行命令，但不在当前 shell
exit	退出当前 shell
export	导出变量，使当前 shell 可利用它
pwd	显示当前目录
read	从标准输入读取一行文本
命令	命令解释
readonly	设置变量为只读
return	退出函数并带有返回值
set	控制各种参数到标准输出的显示
shift	命令行参数向左偏移一个
test	条件测试
times	显示用户脚本或任何系统命令的运行时间，第一行给出 shell 消耗时间，第二行给出运行命令消耗的时间
trap	当捕获信号时运行指定命令
ulimit	显示或设置 shell 资源
umask	显示或设置缺省文件创建模式
unset	从 shell 内存中删除变量或函数
wait	等待直到子进程运行完毕，报告终止
type	查询命令是否驻留系统以及该命令的类型

运行多个命令：

cd /etc;ls　　cd /etc&&ls　　cd /etc||ls

(cd /etc;ls)　　{cd /etc/X11;ls}

六、date

# date '+%a';; 星期几的缩写(Sun)	%A　星期几的完整写法(Sunday)
%b　月名的缩写(Oct)	%B　月名的完整写法(October)

# date '+%h';; Month abbreviation	# date '+%H';; 24 小时格式的小时
# date '+%d';; 十进制日期	# date '+%D';; # MM/DD/YY
%e 日期，如果只有一位会补上一个空格	%Z 时区(PDT)
%I 用十进制表示 12 小时格式的小时	# date '+%j';; 一年中的第几天
# date '+%m' 十进制表示的月份	# date '+%M' 十进制表示的分钟
%p 12 小时表示法(AM/PM)	# date '+%S';; 十进制 Second
# date '+%s';; 1970/01/01 second	# date '+%r';; # AM-PM time
# date '+%w';; 十进制星期几(sun 是 0)	%W 一年中的第几个星期(周一为第一天)
%x 重新设置本地日期(08/20/99)	%X 重新设置本地时间(12：00：00)
# date '+%y';; 两位数字表示的年(99)	%Y 四位数字表示的年
# date +'%F %T' 2012-06-23 12:53:07	# date +'%z-%Y-%M-%d %H:%M:%S' +0800-2012-06-23 12:06:34
# date +'%H:%M:%S' 13-06-49	# date +'%T' 21:20:08

date -d "last sunday" +%Y-%m-%d //上一星期天的日期

date 062421172012 //MMddHHMMYYYY

date -s 04/10/2002 //设定时间

date -s 02:02:00 //设定日期

七、ls -F

```
[root@station1 ~]# ls -F|grep \*$|cut -d* -f1
welcom.root
z.sh.bz2
[root@station1 ~]# ls -F|grep \*$
welcom.root*
z.sh.bz2*
```

无标识代表一般文件	ls -F|grep -v *$\|grep -v \@$|grep -v \=$\|grep -v \\|$|grep -v \/$ ls -F|egrep -v '*$\|\@$|\=$\|\\|$|\/$'

*代表可执行文件	ls -F /bin/bash*|grep *$
@代表 soft link	ls -F /bin/sh@|grep \@$
=代表 socket 文件	ls -F /dev/log|grep \=$
|代表 pipe 文件	ls -F /dev|grep \\|$

八、lsof 命令的使用案例

lsof 列出所有打开的文件 lsof | grep 'filename' 查看谁正在使用某个文件
lsof -c http 或 lsof | grep mysql 列出某个程序所打开的文件信息 lsof -c mysql -c apache 列出多个程序多打开的文件信息
lsof -u username 列出某个用户打开的文件信息 lsof -g 5555 某个用户组所打开的文件信息 lsof -u test -c mysql 列出某个用户以及某个程序所打开的文件信息 lsof -u ^root 列出除了某个用户外的被打开的文件信息
lsof -p 1 通过某个进程号显示该进行打开的文件 lsof -p 123,456,789 列出多个进程号对应的文件信息 lsof -p ^1 列出除了某个进程号，其他进程号所打开的文件信息
lsof –i 列出所有的网络连接 lsof -i tcp 列出所有 tcp 网络连接信息 lsof -i udp 列出所有 udp 网络连接信息 lsof -i :3306 列出谁在使用某个端口 lsof -i udp:55 列出谁在使用某个特定的 udp 端口 lsof -i tcp:80 特定的 tcp 端口 lsof -a -u test –i 列出某个用户的所有活跃的网络端口

UNIT02 File Finder

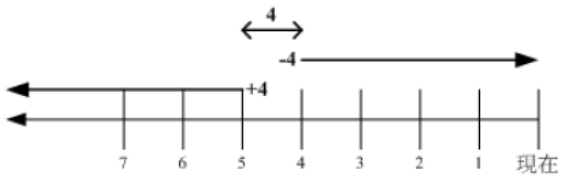
find 〈PATH〉 [-option] [action] 直接搜寻硬盘实时、递归、根据 i 节点信息查找文件时间花费很大！

whereis 与 locate 是利用数据库来搜寻数据而没有搜寻实际的硬盘文件，所以相当的快速，比较省时间！

所以通常使用 whereis 或者 locate 来检查，如果真的找不到了，才以 find 来搜寻。

一、与时间有关的参数(-atime,-ctime,-mtime)

-mtime n ：n 为数字，意义为在 n 天之前的『一天之内』被更动过的；
-mtime +n ：列出在 n 天之前(不含 n 天本身)被更动过内容的档案档名；
-mtime -n ：列出在 n 天之内(含 n 天本身)被更动过内容的档案档名。
-newer file ：file 为一个存在的档案，列出比 file 还要新的档案档名



find / -mtime 0　0 天=24 小时内	find / -mtime 4　3*24~4*24 小时内
find . -newer age.awk ! -newer belts.awk -exec ls -l {} \;	

二、与使用者或群组有关的参数

-uid n ：n 为数字，这个数字是用户的账号 ID，亦即 UID
-gid n ：n 为数字，这个数字是组名的 ID，亦即 GID
-user name ：name 为使用者账号名称喔！例如 dmtsai
-group name：name 为组名喔，例如 users ；
-nouser ：寻找档案的拥有者不存在 /etc/passwd 的人！
-nogroup ：寻找档案的拥有群组不存在于 /etc/group 的档案！
　　当你自行安装软件时，很可能该软件的属性当中并没有档案拥有者，
　　这是可能的！在这个时候，就可以使用 -nouser 与 -nogroup 搜寻。

```
#find /home -user zcs                    //找出属于某个用户的文件
```

#find / -nouser	//找出不属于系统任何人的档案

三、与文件名有关的参数

# find / -name t*.doc	# find |grep abc
# find -name tes?[1-5].doc	# find -name “test.doc”

四、与文件类型有关的参数(find /home -type f 找普通文件)

-type TYPE ：搜寻档案的类型为 TYPE 的，类型主要有：一般正规档案 (f)，装置档案 (b, c)，目录 (d)，连结档 (l)，socket (s)，及 FIFO (p) 等属性。

$ find /etc -type d -print	$ find . ! -type d -print
$ find /etc -type l -print	$ find /dev -type s -print|xargs file

在使用 find 命令的-exec 选项处理匹配到的文件时，find 命令将所有匹配到的文件一起传递给 exec 执行。不幸的是，有些系统对能够传递给 exec 的命令长度有限制，这样在 find 命令运行几分钟之后，就会出现溢出错误。错误信息通常是“参数列太长”或“参数列溢出”。这就是 xargs 命令的用处所在，特别是与 find 命令一起使用。Find 命令把匹配到的文件传递给 xargs 命令，而 xargs 命令每次只获取一部分文件而不是全部，不像-exec 选项那样。这样它可以先处理最先获取的一部分文件，然后是下一批，并如此继续下去。

在有些系统中，使用-exec 选项会为处理每一个匹配到的文件而发起一个相应的进程，并非将匹配到的文件全部作为参数一次执行；这样在有些情况下就会出现进程过多，系统性能下降的问题，因而效率不高；而使用 xargs 命令则只有一个进程。另外，在使用 xargs 命令时，究竟是一次获取所有的参数，还是分批取得参数，以及每一次获取参数的数目都会根据该命令的选项及系统内核中相应的可调参数来确定。

下面的例子查找系统中的每一个普通文件，然后使用 xargs 命令来测试它们分别属于哪类文件：

$find /dev -type s -print|xargs file

五、与文件大小有关的参数

-size [+-]SIZE：搜寻比 SIZE 还要大(+)或小(-)的档案。这个 SIZE 的规格有：c：代表 byte，
k：代表 1024bytes。所以，要找比 50KB 还要大的档案，就是『 -size +50k 』

-size +1000k 大于 1M	-size +8（默认 M）	-size +1000c	- size 0b

六、与档案权限有关的参数

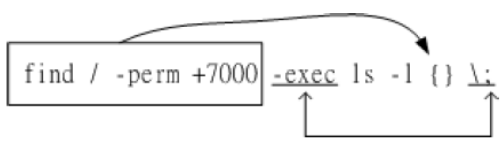
-perm mode ：搜寻档案属性『刚好等于』 mode 的档案，这个 mode 为类似 chmod
的属性值，举例来说， -rwsr-xr-x 的属性为 4755 ！
-perm -mode ：搜寻档案属性『必须要全部囊括 mode 的属性』的档案，举例来说，
我们要搜寻 -rwxr--r-- ，亦即 0744 的档案，使用 -perm -0744，
当一个档案的属性为 -rwsr-xr-x ，亦即 4755 时，也会被列出来，
因为 -rwsr-xr-x 的属性已经囊括了 -rwxr--r-- 的属性了。
-perm +mode ：搜寻档案属性『包含任一 mode 的属性』的档案，举例来说，我们搜寻
-rwxr-xr-x ，亦即 -perm +755 时，但一个档案属性为 -rw------
也会被列出来，因为他有 -rw.... 的属性存在！

#find / -perm - 220	find / - perm - g+w,u+w
#find / -perm /222 #find / -perm /u+w,g+w #find / -perm /u=w,g=w	u 有 w or g 有 w or o 有 w u 有 w or g 有 w u 有 w or g 有 w
#find / - perm -444 - perm /222 ! - perm /111 #find / - perm - a+r - perm /a+w ! - perm /a+x	

搜寻档案当中含有 SGID/SUID/SBIT 的属性
[root@linux ~]# find / -perm +7000
所谓的 7000 就是 ---s--s--t ，那么只要含有 s 或 t 的就列出，
所以当然要使用 +7000 ，使用 -7000 表示要含有 ---s--s--t 的所有三个权限，
因此，就是 +7000 ～瞭乎？

七、额外可进行的动作

-exec command ：command 为其它指令，-exec 后面可再接额外的指令来处理搜寻到
的结果。
-print ：将结果打印到屏幕上，这个动作是预设动作！



该范例中特殊的地方有 {} 以及 \; 还有 -exec 这个关键词，这些东西的意义为：

- {} 代表的是『由 find 找到的内容』，如上图所示，find 的结果会被放置到 {} 位置中；
- -exec 一直到 \; 是关键词，代表 find 额外动作的开始 (-exec) 到结束 (\;)，在这中间的就是 find 指令内的额外动作。 在本例中就是『 ls -l {} 』啰！
- 因为『;』在 bash 环境下是有特殊意义的，因此利用反斜杠来跳脱。

find path name-options [-print-exec -ok]

-ok 和-exec 的作用相同，只不过以一种更为安全的模式来执行该参数所给出的 shell 命令，在执行每一个命令之前，都会给出提示，让用户来确定是否执行。

#find / -exec grep "Hello" {} \;	查询所有保含字符串“Hello”的文件
#find / -name "*.tmp" -exec rm -f {} \;	删除所有临时文件
#find / -ctime +20 -exec rm -f {} \;	删除+20 天以前的文件
#find /etc -type f -links +1 -exec cp {} /links \;	

八、使用 depth 选项

在使用 find 命令时，可能希望先匹配所有的文件，再在子目录中查找。使用 depth 选项就可以使 find 命令这样做。这样做的一个原因就是，当在使用 find 命令向磁带上备份文件系统时，希望首先备份所有的文件，其次再备份子目录中的文件。

在下例中，find 从文件系统的根目录开始，查找 CON.FILE 的文件。它将首先匹配所有的文件然后再进入子目录中查找。

$find / -name"ZCS.FILE"-depth-print

九、-patch -prune 忽略某个目录

如果在查找文件时希望忽略某个目录，因为你知道那个目录中没有你所要查找的文件，那么可以使用-prune 选项来指出需要忽略的目录。在使用-prune 选项时要当心，因为如果你同时使用了-depth 选项，那么-prune 选项就会被 find 命令忽略。

如果希望在/apps 目录下查找文件，但不希望在/apps/bin 目录下查找，可以用：

比如要在/usr/sam 目录下查找不在 dir1 子目录之内的所有文件

find /usr/sam -path "/usr/sam/dir1" -prune -o -print

find [-path ..] [expression] 在路径列表的后面的是表达式

-path "/usr/sam" -prune -o -print

是-path "/usr/sam" -a -prune -o -print 简写

表达式按序求值，-a 和 -o 都是短路求值，与 && 和 || 类似

这个表达式组合特例可以用伪码写为

```
if -path "/usr/sam"  then
          -prune
else
          -print
```

避开多个文件夹

find /usr/sam \(-path /usr/sam/dir1 -o -path /usr/sam/file1 \) -prune -o -print

圆括号表示表达式的结合。

\ 表示引用，即指示 shell 不对后面的字符作特殊解释，而留给 find 去解释其意义。

查找某一确定文件，-name 等选项加在-o 之后

#find /usr/sam \(-path /usr/sam/dir1 -o -path /usr/sam/file1 \) -prune -o -name "temp" -print

十、除 find 外的其它常文件查找名称

1、locate passwd 在 updatedb 库中的文件名

/etc/updatedb.conf

```
PRUNEFS = "auto afs gfs gfs2 iso9660 sfs udf"
PRUNEPATHS = "/afs /media /net /sfs /tmp /udev /var/spool/cups /var/spool/squid /var/tmp"
```

/var/lib/mlocate/mlocate.db

数据库的建立是在每天执行一次(Redhat 默认)，所以当新建立起来档案，却还在数据更新之前搜寻档案，那么 locate 会报告找不到，此时可以手动执行 updatedb 命令来更新数据库。

```
[root@linux ~]# locate passwd
/lib/security/pam_passwdqc.so
/lib/security/pam_unix_passwd.so
/usr/lib/kde3/kded_kpasswdserver.so
.......中间省略.......
```

2、whereis useradd 在库中找文件

```
[root@linux ~]# whereis [-bmsu] 档案或目录名
-b    :只找 binary 的档案
-m    :只找在说明文件 manual 路径下的档案
-s    :只找 source 来源档案
-u    :没有说明档的档案!

[root@linux ~]# whereis -b passwd
passwd: /usr/bin/passwd /etc/passwd /etc/passwd.OLD
[root@linux ~]# whereis -m passwd
passwd: /usr/share/man/man1/passwd.1.gz /usr/share/man/man5/passwd.5.gz
```

3、which passwd 在$PATH 中找命令

-a ： 将所有可以找到的指令均列出，而不止第一个被找到的指令名称

```
[root@station1 ~]# which passwd
/bin/passwd
[root@station1 ~]# which passwd -a
/bin/passwd
/usr/bin/passwd
```

4、grep -ir root /etc 根据内容查找

5、ls -R|grep passwd根据文件名称

6、type -p grep

UNIT03 Basic Text Process

1、tr(-s 浓缩重复的字符)

```
#tr  'Installing'  'installing'  < /root/install.log
#tr -d Installing < install.log                    //-d 作用是删除
#cut -d: -f1-6 /etc/passwd|tr :  '+'
```

```
[root@station1 ~]# cat a
111
222
111
333
[root@station1 ~]# tr '1'  '2' < a
222
222
222
333
[root@station1 ~]# tr -s '1'  '2' < a
2
2
2
333


[root@station1 ~]# ps au|head -n2|tr ' ' ':'
USER:::::::PID:%CPU:%MEM::::VSZ:::RSS:TTY::::::STAT:START:::TIME:COMMAND
root::::::3436::0.0::0.0:::1676:::436:tty1:::::Ss+::05:47:::0:00:/sbin/mingetty:tty1
[root@station1 ~]# ps au|head -n2|tr -s ' ' ':'
USER:PID:%CPU:%MEM:VSZ:RSS:TTY:STAT:START:TIME:COMMAND
root:3436:0.0:0.0:1676:436:tty1:Ss+:05:47:0:00:/sbin/mingetty:tty1
[root@station1 ~]# ps au|head -n2|tr -s ' ' ':'|cut -d: -f2
PID
3436
```

2、cut

```
[root@station1 ~]# cut -d: -f1,5 /etc/passwd|grep -E '^root|^news'
root:root
news:news
[root@station1 ~]# cut -d: -f1,6 /etc/passwd|grep -E '^root|^news'
root:/root
news:/etc/news
[root@station1 ~]# cut -d: -f1,7 /etc/passwd|grep -E '^root|^news'
root:/bin/bash
news:
```

```
# ifconfig eth0 |grep "inet addr:" |awk '{print $2}'|cut -c 6-
# cut -c1-3,22-   f10              //抽出每行 1-3 及时性 22 个以后的字符
# cut -d: f3,4 /etc/passwd         //抽出各行的第 3,4 个字段
# free | tr -s '' | sed '/^Mem/!d' | cut -d" " -f2
```

3、sort+uniq

sort -r f10	降序排列
sort -n f10	显示行号
sort -k 2 f10	以第 2 列为依据排序
sort -k2 -t: passwd	以:为分隔符分列
sort dataf3|uniq	删除重复行

sort dataf3|uniq -d	挑出重复行
sort dataf3|uniq -c	计算重复次数

4、basename+dirname

tee foo.log tee -a foo.log	清空并在屏幕上显示及保存^d 追加并在屏幕上显示及保存^d
wc -l install.log wc -c install.log wc -w install.log	l=line=行 c=char=字符 w=word=单词 wc -lcw install.log
basename /usr/local/bin/sftp	dirname /usr/local/bin/sftp
strings /bin/ls	在二进制文件中查找可打印字符

5、tail

head -2 /etc/inittab.conf 看文件的前 2 行

tail notes	显示最后 10 行
tail -n 2 /etc/passwd	tail -2 /etc/passwd
tail -c +200 notes	从第 200 字节开始
tail -f accounts	实时地去读最新的后 10 行

6、cat install.log 不能翻页

-b, --number-nonblank:number nonblank output lines	
cat -A;	等价于--show-all
cat -E 末尾加$	等价于 cat --show-ends
cat -n 显示行号	等价于 cat--number
cat -t 用^I 代替 tab	等价于 cat --showtabs
cat -s 压缩多个空行为一个空行	等价于 cat --squeeze-blank

7、join

```
[root@station1 ~]# sort a1 >a1.o
[root@station1 ~]# sort a2 >a2.o
[root@station1 ~]# join a1.o a2.o
c 3 wo
s 2 hi
z 1 ok
```

8、expand -t 1 会将 TAB 改成 SPACE

9、touch -t 06261726 example

```
[root@station1 /]# touch -t 06261726 example
[root@station1 /]# stat example
  File: `example'
  Size: 0               Blocks: 8          IO Block: 4096   regular empty file
Device: fd00h/64768d    Inode: 98307       Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2012-06-26 17:26:00.000000000 +0800
Modify: 2012-06-26 17:26:00.000000000 +0800
Change: 2012-06-26 16:24:17.000000000 +0800
```

10、反序排列

# cat example	#echo "a b c"|rev

a c b # tac example b c a	c b a

UNIT03 Format Control Character

一、转义字符

转义字符	含义	转义字符	含义
\a	alert BEL	\b	退格键（Back Space）
\f	FormFeed 换页仅影响打印机	\n	Newline, 回车换行
\r	return，回车回行首	\t	Tab(水平 4,8 格)
\v	Vertical Tab 仅影响打印机	\o	空字符(NULL)
\xNN	转换 NN 数字成为字符	\d dd	八进制值
\c	取消行末之换行符号	\E	ESCAPE，跳脱键
\’	在双引号里只用’即可	\”	双引号
\\	反斜杠	\?	问号字符

1、echo

-e ： 启用(预设关闭-E)反斜线控制字符的转换(参考下表)

-n ： 取消行末之换行符号(与 -e 选项下的 \c 字符同意)

```
$ echo -n first line                    //-n 取消换行符
first line $                            //shell prompt 接在输出结果同一行
```

```
$ echo -e “a\tb\tc\nd\te\bf\a”          //\b：BACKSPACE ，也就是向左删除键
a b c                                   //\a： BELL (从系统喇叭送出铃声)
d f $                                   //e 后是删除键(\b)，因此输出就没有 e 了
```

由于使用了 -n 选项，因此 shell prompt 紧接在第二行之后。若你不用 -n 的话，那你在 \a 后再加个 \c ，也是同样的效果。

```
$ echo -e “\x61\x09\x62\x09\x63\x0a\x64\x09\x65\x09\x66”
a b c                                   //以 x 开首为十六进制
d e f                                   //\t：TAB，表格跳位键
```

2、awk

打印 May Day，中间夹 tab 键，后跟两个新行，再打印 May Day，但这次使用 ASCII 八进制数 104、141、171、分别代表 D、a、y。

```
# awk ’ BEGIN {print"\n\tDay\n\t\104\141\171"}’
```

Day Day

二、格式控制符(shell,c 及 awk 语言中的 printf 相近)

格式字符串由格式字符组成，%号后跟 n 个规定字符用来确定输出内容格式。有几个参数就要有几个格式控制符。

格式控制符（printf ” % [修饰符 [width] [.prec]] fmt”，参数,. . .)

1、% 标识一个格式控制符的开始，不可省略

2、修饰符，可省

- 左对齐，省略时右对齐 # 显示 8 进制整数时在前面加个 0 + 显示使用 d,e,f 和 g 转换的整数时，加上+号或－号 0 用 0 而不是空白符来填充所显示的值
echo "Linux" |nawk '{printf "|%-15s|\n", $1}' |Linux | //使用 '-' 时左对齐
awk -F ":" '{printf " %-15s %-15s %s\n",$1,$3,$7}' passwd rpcuser 29 /sbin/nologin

#printf("%+d %+d %+d\n",-5,0,5) //“+”号作用是输出符号位即数的正负号
如果不希望正数的前面出现‘+’号，只要在“% d”中间加个“ ”号（即空格）就行，
#printf("% d\n",-5,0,5); //注意%和 d 之间有一个空格
#awk ‘{ printf“%#0\n”,1 }’——01
#awk ‘{ printf “%#8.0f\n”,a }’ ——0. //浮点，即使不跟数字也打印一个小数点
#awk ‘{ printf “%#10.3g”,1 }’ ——1.00 //%g，%G，防止尾随 0 被删除

3、width，可省 字段宽度的最小值

如果实际数据宽度>m 则按实际宽度输出

4、.precision，精度,可省

%e、%E、%f-- 显示小数的位数	%g、%G - 有效数字的最大位数
%s -- 打印字符的最大数目	整数转换 -- 打印数字的最小位数

5、fmt，数据类型，不可省

%s	字符串	%q	将特殊字符用\转义
%c	字符	%d	十进制整数，同%i，%u
%f	浮点数，单精度小数，%lf 双精度	%e	显示科学计数法
%g	由 bash 选择用%f 或%e	%g	e 或 f 中较短的，并去掉无用的 0
%o	八进制	%p	指针

%X	由 A-F 表示 16 进制数	%x	由 a-f 表示 16 进制数
	%ld ， %lx ， %lo ， %lu　长整型		%hd ， %hx ， %ho ， %hu　短整型

6、用 Shell printf 显示变量值

%8.2f　全部的宽度仅有 8 个字符，整数部分占有 5 个字符，小数点本身 (.) 占一位，小数点下的位数则有两位。字符宽度： 12345678
%8.2f 意义：00000.00

printf “Hell world.\n” ——\n 有换行效果
printf “%s” “$str” ——显示字符串不换行
printf “%s\n” “$str” ——显示字符串并换行
printf “%c\n” “$str” ——显示第一个字符串并换行
printf “%f\n” 30——30.000000 浮点数
printf “%5s\n” yes——显示 5 个字符长度的字符串（靠右对齐），不足加空格
printf “%-5s\n” yes——显示 5 个字符长度的字符串（靠左对齐），不足加空格
printf “%5.1f\n” 30——表示含小数点 1 位共 5 位
printf “%s\n” “ABCDEFG”|tr ‘[A-Z]’ ‘[a-z]’
printf -v myvar “%q” “ABC 123 xyz” ——-v 不显到屏幕上，而给变量赋值
#awk ‘BEGIN{printf “num:%3.1f\n %15.2f\n”,999,888}’ num:9.0 8.00
#printf ‘\x45’ #echo 1.7 > f1 #awk ’{printf (″%d\n″,$1)} ’ f1　1 #awk ’{printf (″%f\n″,$1)}’ f1　1.700000 #awk ’{printf (″%6.1f\n″,$1)}’ f1 1.70　总 6=小 2＋整 6,小数点 1, 不够 6 右对齐 #awk ’{printf (″%04.1f\n″,$1)}’ f1　01.7　总宽为 4 场宽前有 0 则在输出值前加 0 #echo “65″ | awk ‘{ printf (” %c\n”,$0) }’　A #awk ‘BEGIN{printf “%.4f\n”,999}’ 999.0000

UNIT05 Script Execute Principle

一、父 Shell 和子 Shell

1、login shell 和父 shell

登陆主机后，在执行 Script 之前，我们所处的环境已经是在一个 Bash Shell 之中。这个 Shell 叫做 login Shell 是将来我们执行任何 Script 的上层环境，又叫做父 shell。

2、父 shell 和子 shell

执行某个 Script 时，父 shell 会根据 script 的第一行#!后指定的 shell 程序开启（fork）一个子 shell 环境，然后在子 shell 中执行此 script。一旦子 shell 中的 script 执行完毕，此子 shell 随即结束，仍然回到父 shell 中，不影响父 shell 原来的环境。

3、子 shell 会继承父 shell 的若干变量值的内容，这些变量称为环境变量。

4、子 shell 再开启子 shell——echo $SHLVL 用于看第几层

二、bash 的启动配置文件（在脚本中加入 echo 验证）

1、登陆

→/etc/profile.d/*.sh

→/etc/profile——bash login（提示符，7 个 shell 全同）

→/etc/bashrc——all shell（bash 提示符）

→~/.bashrc——all shell

→~/.bash_profile——all shell

→~/.bash_logout——bash（rm -rf /tmp/*）

2、注销

~/.bash_logout——举例 history -c;poweroff

3、执行新 shell

→/etc/bashrc

→~/.bashrc——末尾加 exit 然后 ctrl+shift+t

4、执行 script（使用!/bin/bash）

查 BASH_ENV 的内容

5、执行 script(使用!/bin/sh)

不调用启动文件，没有其它检查环境变量的操作。

三、子 shell 会继承环境变量，不需每次启动 profile

1、登陆 shell

Su －; login

2、非登陆 shell

su;gnome-terminal;exec-script;other

	上下文	登陆 shell	交互 shell
1	从 TTYn 登 shell	Ok	Ok
2	从网络登陆 shell	Ok	Ok
3	X 初始化 shell	Ok	Ok
4	X 终端 shell	X	Ok
5	执行 bash 命令的子 shell	X	ok
6	命令替换中的子 shell $(命令)	X	X
7	圆括号分组的命令的子 shell	X	X
8	执行脚本时使用的子 shell	X	X

四、source 和 script 的区别

1、script

script 是启动一个子 shell 来执行命令，在 script 中设的变量，无法改变当前 bash。

#vim auth.sh

echo “what is your name”

read name

echo “… … ok”

chmod +x auth.sh

. ./auth.sh————script

2、source

source 在当前 bash 环境下执行命令，所以通过文件设置环境变量是要用 source 命令。

```
#vim auth.sh
echo “what is your name”

read
echo ‘your name is:’ $REPLY

echo “what is your pass”
read pass
echo “… … ok”
chmod +x auth.sh
source auth.sh————source
```

五、环境信息配置文件 bashrc 与 profile 的区别

1、当登入系统时候获得一个 shell 进程时，其读取环境设定档有三步

首先读入的是全局环境变量设定档/etc/profile，然后根据其内容读取额外的设定的文档，如/etc/profile.d 和/etc/inputrc
然后根据不同使用者帐号，去其家目录读取~/.bash_profile，如果这读取不了就读取~/.bash_login，这个也读取不了才会读取~/.profile，这三个文档设定基本上是一样的，读取有优先关系
然后在根据用户帐号读取~/.bashrc

2、bashrc 用于 non-loginshell，而 profile 用于 login shell

/etc/profile:用户登陆时,该文件仅被执行一次，并从/etc/profile.d 下的文件中搜集 shell 的设置。
/etc/bashrc:用户打开新有 bash shell 时。有些 linux 版本中/etc 下已经没有该文件。
~/. profile:专用于某个用户登录时,该文件仅仅执行一次!然后执行用户的.bashrc。
~/.bashrc:专用于某个用户,当该用户登录时以及每次打开新的 shell 时该文件被读取。

六、BASH 分析命令行的方式

分隔管道	将指令切成个别的命令，以各个命令进行以下分析
取出 token	使用分隔字符——空格、tab 将命令分成 token
替换别名	检查命令的第一个 token 是不是别名，若是则把名名替换成真正的指令
括号扩展	如{sb,b}in 会扩展成 sbin、bin
~符号扩展	如~user2 替换成/home/user2
替换变量	如$HZ 替换成’Hello!’
替换命令	若 token 含$(which diff)的型式则把它替换成实际结果/usr/bin/diff
替换通配符	如*.sh 替换成 foo.sh
替换算术	若有算术式，则计算其结果并进行替换，如$((3+5))替换成 8
根据函数、内置命令、搜寻路径的顺序，找寻第一个 token 所代表的命令位于何处。	
执行已替换完成的指令	

UNIT06 Shell Mata-Characters

\回车,	续行符	\	跳脱符,将元字符还原成一般字符
>,<	输出重定向	>,<	输入重定向
!	调用历史记录;	!	逻辑运算中的 not
/	路径分割符	|	两个命令之间的管道
;	分割执行多个命令	$	置换，引用:
#	批注，常用在脚本中	~	家目录
&	将命令置为后台工作		

一、转义字符=跳脱符=逃逸字符——\

反斜线是使后续的字符恢复原来作为单纯字符的用途，换言之，这是要除去特殊字符的作用。下表的每一个字符都是 bash 的特殊字符。要显示这些字符本身，必须在其前面加上跳脱符。

\'	\"	*	\?	\\	\/
\#	\$	\<	\>	\&	\\|
\[	\]	\;	\!	\(	\)
\~	\`	\{	\}		

二、通配符元字符——万用字符

1、?一定有一个字符‘.’，不可为空，但不匹配‘.’打头的

#ls test?.dat　　#ls -la ???

2、*代表 0 或多个字符‘.’，可以是空字符串，‘.’打头的除外

zip——文件名中含 zip　　z*e——头是 z 并且尾是 e

ls???——ls 打头长度为 5　　/*/*.conf——某个下一级目录中的.conf

3、连字符“-”仅在方括号内有效，而*和？只在方括号外面是通配符

[a-dm]?　　以 a、b、c、d、m 开头且后面只跟一个字符的文件的名称

-a[*?]abc　*和？均为普通字符，它匹配的字符串只能是-a*abc 和-a?abc。

三、三种引号

echo Jack\`s book;echo “Jack\`s book”;echo ‘Jack\`s book’

1、‘’ 单引号逐个地取一串字符，不具有变量置换功能 #echo ‘echo $UID’ echo $UID mkdir ’ a b’
2、“” 双引号内容作字串允许变量置换 #echo “echo $UID” echo 0
3、``引用命令的结果,两反引号间为优先执行的命令 #echo `echo $UID` 0

四、三种括号的基用法

命令替换行=命令行扩展=bash 在解释命令前替换某些元字符，在操作上，用 $( ) 或 ` `″个人″比较喜欢用 $( ) ，理由是：

` ` 很容易与 ’ ’ (单引号)搞混乱。
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。

1、():将 command group 置于 nested sub-shell 去执行

$(ls /)	//命令替换
$((9**2))	//算术扩展
$var ， $(var)	//Shell 和环境变量

2、[]:任一字符

[]内任一字符	[a-z] [A-Z] [0-9] [a-Za-z0-9]* [wxy]
匹配列表外任意一个字符	[!abc]* [^abc]*
A1 b1 c1 ab1 ac1 ba1 bc1	

#ls [^ab]*只显示 c1 #rm [abc]*以上都会被删除 #cp -r /etc/sysconfig /bak/etc-$[$(date %Y%m%d) -1]

3、{}:枚举,字串集合，每个都分别匹配；在中间为区块的组合

echo s{ab,cd}y——saby scdy
/bin/z{[ef]gre,cm}p———/bin/zegrep /bin/zfgrep /bin/zcmp
ls /etc/*{conf,cf,org}
ls /etc/profile{,.bak}
ls /etc/sources.list{,.bak}
mkdir -p /root/tmp/{dir1,dir2,dir3}/{a,b,c}
echo {1,2,3,4,5,5,6,7,8,9}*{1,2,3,4,5,5,6,7,8,9}

UNIT07 BREs EREs PREs

正则表达式是一个包含运算符和运算字符串的表达式，用来匹配特定规则的文本正则表达式 简称为 表达式、匹配模式 或者 模式，它们可以互换。Shell 的正则表达式与 grep、sed、Awk 是不同的。

正规表示法与 Shell 在 Linux 当中的角色定位

说实在的，我们在学数学的时候，一个很重要、但是粉难的东西是一定要『背』的，那就是九九表，背成功了之后，未来在数学应用的路途上，真是一帆风顺啊！这个九九表我们在小学的时候几乎背了一整年才背下来，并不是这么好背的呢！但他却是基础当中的基础！你现在一定受惠相当的多呢 ^_^！

而我们谈到的这个正规表示法，与前一章的 BASH 就有点像是数学的九九表一样，是 Linux 基础当中的基础，虽然也是最难的部分，不过，如果学成了之后，一定是『大大的有帮助』的！这就好像是金庸小说里面的学武难关：任督二脉！打通任督二脉之后，武功立刻成倍成长！所以啦，不论是对于系统的认识与系统的管理部分，他都有很棒的辅助啊！请好好的学习这个基础吧！ ^_^

一、正则分类

许多程序设计语言都支持利用正则进行字符串操作。由于起源于 unix 系统，因此很多语法规则一样的。但是随着逐渐发展，后来扩展出以下几个类型。

1、PREs/Perl RE/POSIX RE

正则从 Perl 衍生出一个显赫的流派，叫做 Perl Compatible Regular Expression，\d、\w、\s 就是这个流派的特征。

下面表格列出基本的正则功能在常用 GNU 工具中的表示法。

PCRE 记法	vi/vim	grep	awk	sed
*	*	*	*	*
+	\+	\+	+	\+
?	\=	\?	?	\?
{m,n}	\{m,n}	\{m,n\}	{m,n}	\{m,n\}
\b *	\< \>	\< \>	\< \>	\y \< \>
(…|…)	\(…|…\)	\(…|…\)	(…|…)	(…|…)
(…)	\(…\)	\(…\)	(…)	(…)
\1 \2	\1 \2	\1 \2	不支持	\1 \2

注：PCRE 中\b 表示“单词起始或结束”，Linux 工具中\<匹配“单词起始”，\>匹配“单词结束”，sed 中\y 同时匹配这两个位置。

2、BREs/Basic Regular Expression

grep、vi、sed 都属于 BRE 这一派，是因为这些工具的诞生时间很早，之前这些元字符可能并没有特殊的含义;为保证向后兼容，元字符(、)、{、}必须\转义之后才具有特殊含义。

()可用来定义操作符的范围和优先度, (grand)?father 匹配 father 和 grandfather。
()+为扩展正则里的特殊符号,普通正则需要转义。 echo "abcabc" | grep "\(abc\)\+" echo "abcabc" | grep -E "(abc)+"
{}和|也是扩展的正则，sed，grep 使用时需加\转义，如 god|bon 匹配单词 god 或 bon
<>不属于扩展正则，但是通过使用\转义可匹配单词边界，sed、grep、awk 等都需转义。

3、EREs/Extended Regular Expression（不兼容 BRE）

egrep、awk 则属于 ERE 这一派（其实，现在的 BRE 和 ERE 在功能上并没有什么区别，主要的差异是在元字符的转义）。

流派	说明	工具
GNU BRE	(、)、{、}、+、?、|都必须转义使用	GNU grep、GNU sed
GNU ERE	元字符不必转义，+、?、(、)、{、}、|可直接使用，支持\1、\2	grep -E、GNU awk

二、Linux 中常用文本工具与正则的关系

额外的 GNU 正则表达式运算符 (GNU AWK 等工具支持)

\w	匹配任何单词组成字符，等同于[[:alnum:]_]
\W	匹配任何非单词组成字符，等同于[^[:alnum:]_]
\<\>	匹配单词的起始与结尾，如前文所述
\b	匹配单词的起始或结尾处所找到的空字符串。这是 \< 与 \> 运算符的结合 注意：由于 awk 使用 \b 表示后退字符，因此 GNU awk（gawk）使用 \y 表示此功能
\B	匹配两个单词组成字符之间的空字符串
\' \`	分别匹配 emacs 缓冲区的开始与结尾。GNU 程序（还有 emacs）通常将它们视为与 ^ 及 $ 同义

1、grep , egrep 正则特点(按行处理)

①grep 支持：BREs、EREs、PREs	②egrep 支持：EREs、PREs
grep 不跟参数表示使用 ”BREs“ grep ”-E" 要使用 “EREs“	egrep 不跟参数表示要使用 “EREs” egrep “-P" 表示要使用 “PREs"

grep “-P”表示使用 “PREs”	

2、sed 正则特点(按行处理): 默认 BRE, sed “-r” 使用 ERE

3、Awk（gawk）正则特点(对列进行操作) 支持 EREs

三、Regular Expression, RegEx, 描述某种规则的表达式

1、POSIX RE 元字符（用于方括号之外, *——*）

模式	是否匹配	匹配文本（粗体）/匹配失败的理由
ABC	是	居中的第 4、5 及 6 个字符：abc**ABC**defDEF
^ABC	否	限定匹配字符串的起始处
def	是	居中的第 7、8 及 9 个字符：abcABC**def**DEF
def$	否	限制匹配字符串的结尾处
[[:upper:]]\{3\}	是	居中的第 4、5 及 6 个字符：abc**ABC**defDEF
[[:upper:]]\{3\}$	是	结尾的第 10、11 及 12 个字符：abcDEFdef**DEF**
^[[:alpha:]]\{3\}	是	起始的第 1、2 及 3 个字符：**abc**ABCdefDEF

^Jack——以 Jack 开头	123$——$表示在尾部
. 1 个。	如 data\...代表 data.后接 2 个字符, .T.代表 3 个字符是间是 T
+ 1 个、多个	如 goo+gle 可匹配 google、gooogle、goooogle 等;
* 0 个、多个	如 xy*——x, x*(*左邻字符出现 0 个及 0 个以上)。
? 0 个、1 个	如 colou?r 可匹配 color 或者 colour; xy?——x, xy
^.*anonymous	.*等价于 1 乘于*等价于*等价于 0 个、多个字符
( )*$	匹配结尾处有一个或多个空格的行
$grep -c ’^$’ ch04	统计空格数
$grep -c ’^ *$’ ch04	匹配空行
sed -n ‘s/\(Ha\)/\1ha/p’ dataf3 \(...\)把符合的字符暂存起来\1, \n 调用	

2、POSIX RE 元字符（用于方括号之内）

[^0-9], [^A-Z], [^a-zA-Z], [^a-zA-Z0-9]——^代表“非/不是”之意
[Ss]ame, [A-Z], [a-z], [0-9]——代表字符串行中长度为 1 的一个字符
[\,] ——逗号

四、posix 字符类（用于 GNU BRE, GNU ERE, PRE）

POSIX 在 BRE 与 EERE 基础上加入了如下括号字符组的字符。

[:punct:]	标点符号	[][!"#$%&'()*+,./:;<=>?@\^_`{|}~-]
[:alnum:]	字母和数字	[a-zA-Z0-9]
[:lower:]	小写字母	[a-z]
[:upper:]	大写字母	[A-Z]
[:alpha:]	字母	[a-zA-Z]
[:digit:]	数字	[0-9]
[:blank:]	空格字符和制表符	[\t]
[:space:]	空白，含空格、制表、换页等	[\t\r\n\v\f]
[:graph:]	既能看见又能打印的字符	[\x21-\x7E]
[:cntrl:]	控制字符	[\x00-\x1F\x7F]
[:print:]	[:graph:]+空白,可打印字符	[\x20-\x7E]
[:xdigit:]	十六进制字符	[A-Fa-f0-9]
[:ascii:]	ASCII 字符	[\x00-\x7F]

以上这些字符类，如[:alnum:]是 A-Za-z0-9 的另一表达方式。为使用这种字符类，必须使用另一对括号进行将其标识为一正则。例如，A-Za-z0-9 本并不是正则，但[A-Za-z0-9]是。

同样的，[:alnum:]应写成[[:alnum:]]：形式一依赖于 ASCII 字符编码，形式二可在该类中表示来自其他语言的字符。

```
#cat file:   a1
             B222
             cccc
```

要在 awk 中开启支持 posix 类字符，需要多加一对中括号比如

```
awk '/[[:lower:]]/' file     //过滤掉了大写的 B 的行，awk 本身就支持扩展正则的
a1
cccc
```

如果要过滤一个小写字母开头后面是数字的行呢就可这样表达

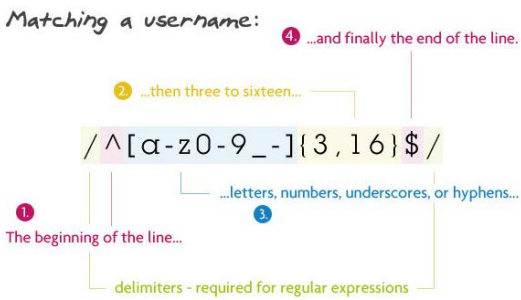
```
#awk '/[[:lower:]][0-9]+/' file
a1
```

精确范围类似 x\{m,n\}时需要开启 posix 支持

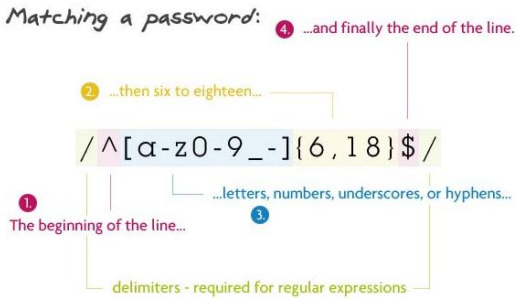
```
#awk --posix '/[[:lower:]][0-9]{1,3}$/' file
a1
#awk --re-interval '/[[:lower:]][0-9]{1,2}$/' file
a1
```

五、8 个你应该了解的正则表达式

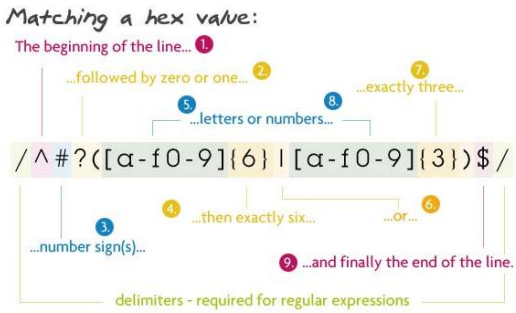
1、匹配用户名：/^[a-z0-9_-]{3,16}$/



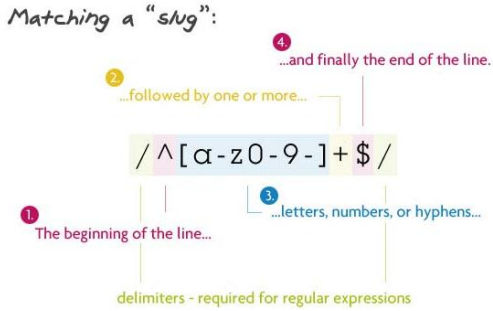
2、匹配密码：/^[a-z0-9_-]{6,18}$/



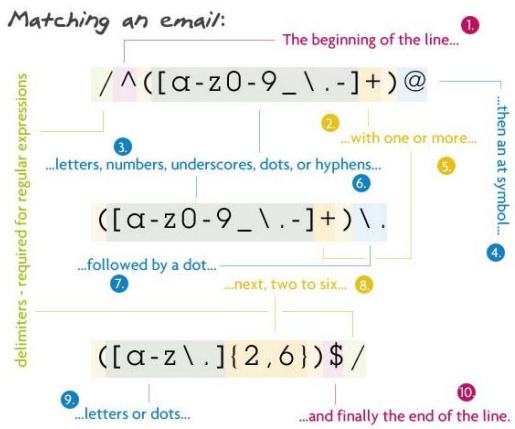
3、匹配一个 Hex 值：/^#?([a-f0-9]{6}|[a-f0-9]{3})$/



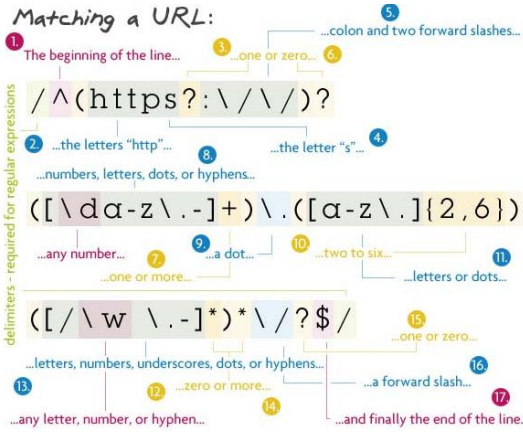
4、匹配一个 Slug：/^[a-z0-9-]+$/



5、匹配一个 Email：/^([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})$/

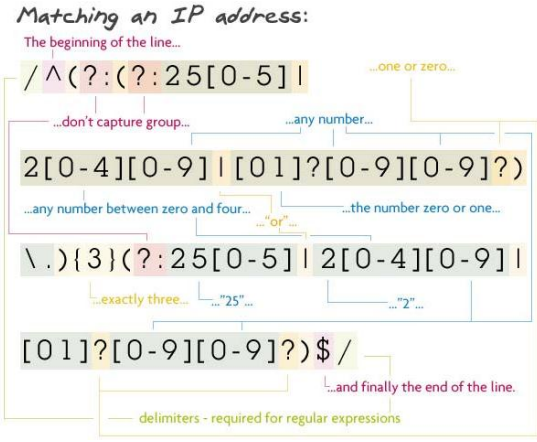


6、匹配一个 URL：/^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]*)*\/?$/

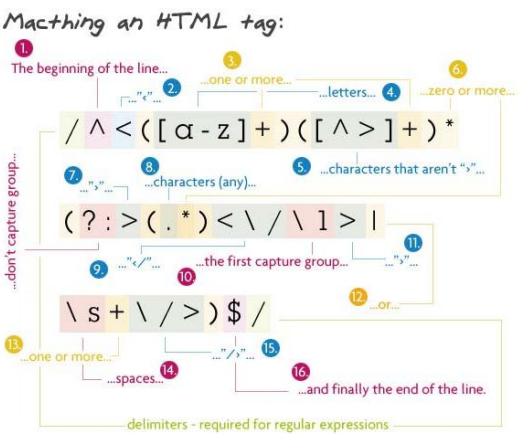


7、匹配 IP 地址：

/^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$/



8、匹配 HTML Tag： /^<([a-z]+)([^<]+)*(?:>(.*)<\/\1>|\s+\/>)$/


Macthing an HTML tag:
1. The beginning of the line...
2. ..."<"...
3. ...one or more...
4. ...letters...
5. ...characters that aren't ">"...
6. ...zero or more...
/^<([a-z]+)([^>]+)*
7. ...">"...
8. ...characters (any)...
...don't capture group...
(?:>(.*)<\/\1>|
9. ..."</"...
10. ...the first capture group...
11. ..."">"...
12. ...or...
\s+\/>)$/
13. ...one or more...
14. ...spaces...
15. ..."/>"...
16. ...and finally the end of the line.
delimiters - required for regular expressions

UNIT19 One hour Mastering RE

^ 和 $ 他们是分别用来匹配字符串的开始和结束，以下分别举例说明

"^The": 开头一定要有"The"字符串;

"of despair$": 结尾一定要有"of despair" 的字符串;

那么,

"^abc$": 就是要求以 abc 开头和以 abc 结尾的字符串，实际上是只有 abc 匹配

'*', '+',和 '?', 他们用来表示一个字符可以出现的次数或者顺序. 他们分别表示：

"zero or more"相当于{0,},

"one or more"相当于{1,},

"zero or one."相当于{0,1}, 这里是一些例子:

"ab*": 和 ab{0,}同义,匹配以 a 开头,后面可以接 0 个或者 N 个 b 组成的字符串("a", "ab", "abbb", 等);

"ab+": 和 ab{1,}同义,同上条一样，但最少要有一个 b 存在 ("ab", "abbb", 等.);

"ab?":和 ab{0,1}同义,可以没有或者只有一个 b;

"a?b+$": 匹配以一个或者 0 个 a 再加上一个以上的 b 结尾的字串.

要点, '*', '+',和 '?'只管它前面那个字符.

你也可以在大括号里面限制字符出现的个数，比如

"ab{2}": 要求 a 后面一定要跟两个 b（一个也不能少）("abb");

"ab{2,}": 要求 a 后面一定要有两个或者两个以上 b(如"abb", "abbbb", 等.);

"ab{3,5}": 要求 a 后面可以有 2－5 个 b("abbb", "abbbb", or "abbbbb").

现在我们把一定几个字符放到小括号里，比如：

"a(bc)*": 匹配 a 后面跟 0 个或者一个"bc";

"a(bc){1,5}": 一个到 5 个 "bc."

还有一个字符 '|', 相当于 OR 操作:

"hi | hello": 匹配含有"hi" 或者 "hello" 的 字符串;

"(b | cd)ef": 匹配含有 "bef" 或者 "cdef"的字符串;

"(a | b)*c": 匹配含有这样多个（包括 0 个）a 或 b，后跟一个 c;

一个点('.')可以代表所有的单一字符,不包括"\n"

如果,要匹配包括"\n"在内的所有单个字符,怎么办?

对了,用'[\n.]'这种模式.

"a.[0-9]": 一个 a 加一个字符再加一个 0 到 9 的数字

"^.{3}$": 三个任意字符结尾 .

中括号括住的内容只匹配一个单一的字符

"[ab]": 匹配单个的 a 或者 b (和 "a | b" 一样);

"[a-d]": 匹配'a' 到'd'的单个字符 (和"a | b | c | d" 还有 "[abcd]"效果一样); 一般我们都用[a-zA-Z]来指定字符为一个大小写英文

"^[a-zA-Z]": 匹配以大小写字母开头的字符串

"[0-9]%": 匹配含有 形如 x% 的字符串

",[a-zA-Z0-9]$": 匹配以逗号再加一个数字或字母结尾的字符串

你也可以把你不想要得字符列在中括号里，你只需要在总括号里面使用'^' 作为开头 "%[^a-zA-Z]%" 匹配含有两个百分号里面有一个非字母的字符串.

要点:^用在中括号开头的时候,就表示排除括号里的字符

在中括号里面, 所有的特殊字符，包括(''), 都将失去他们的特殊性质 "[*\+?{}.]"匹配含有这些字符的字符串.

还有,正如 regx 的手册告诉我们: "如果列表里含有 ']', 最好把它作为列表里的第一个字符(可能跟在'^'后面). 如果含有'-', 最好把它放在最前面或者最后面, or 或者一个范围的第二个结束点[a-d-0-9]中间的 '-' 将有效.

下面说说以\开头的

\b 书上说他是用来匹配一个单词边界,就是...比如've\b',可以匹配 love 里的 ve 而不匹配 very 里有 ve

\B 正好和上面的\b 相反.例子我就不举了

.....可以到 http://www.phpv.net/article.php/251 看看其它用\ 开头的语法

构建一个模式来匹配 货币数量 的输入

构建一个匹配模式去检查输入的信息是否为一个表示 money 的数字。我们认为一个表示 money 的数量有四种方式： "10000.00" 和 "10,000.00",或者没有小数部分, "10000" and "10,000". 现在让我们开始构建这个匹配模式:

^[1-9][0-9]*$

这是所变量必须以非 0 的数字开头.但这也意味着 单一的 "0" 也不能通过测试. 以下是解决的方法:

^(0|[1-9][0-9]*)$

"只有 0 和不以 0 开头的数字与之匹配"，我们也可以允许一个负号在数字之前:

^(0|-?[1-9][0-9]*)$

这就是: "0 或者 一个以 0 开头 且可能 有一个负号在前面的数字." 好了,现在让我们别那么严谨，允许以 0 开头.现在让我们放弃 负号 ，因为我们在表示钱币的时候并不需要用到. 我们现在指定 模式 用来匹配小数部分:

^[0-9]+(\.[0-9]+)?$

这暗示匹配的字符串必须最少以一个阿拉伯数字开头. 但是注意，在上面模式中 "10." 是不匹配的, 只有 "10" 和 "10.2" 才可以. （你知道为什么吗）

^[0-9]+(\.[0-9]{2})?$

我们上面指定小数点后面必须有两位小数.如果你认为这样太苛刻,你可以改成:

^[0-9]+(\.[0-9]{1,2})?$

这将允许小数点后面有一到两个字符. 现在我们加上用来增加可读性的逗号（每隔三位）, 我们可以这样表示:

^[0-9]{1,3}(,[0-9]{3})*(\.[0-9]{1,2})?$

不要忘记 '+' 可以被 '*' 替代 如果你想允许空白字符串被输入话 (为什么?). 也不要忘记反斜杆 '\' 在 php 字符串中可能会出现错误 (很普遍的错误).

现在，我们已经可以确认字符串了, 我们现在把所有逗号都去掉 str_replace(",", "", $money) 然后在把类型看成 double 然后我们就可以通过他做数学计算了.

现在，用户名的开始和结束都不能是句点．服务器也是这样．还有你不能有两个连续的句点他们之间至少存在一个字符，好现在我们来看一下怎么为用户名写一个匹配模式:

^[_a-zA-Z0-9-]+$

现在还不能允许句号的存在．我们把它加上:

^[_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-]+)*$

上面的意思就是说:"以至少一个规范字符（除了.）开头,后面跟着 0 个或者多个以点开始的字符串."

简单化一点，我们可以用 eregi()取代 ereg().eregi()对大小写不敏感，我们就不需要指定两个范围 "a-z" 和 "A-Z" ？只需要指定一个就可以了:

^[_a-z0-9-]+(\.[_a-z0-9-]+)*$

后面的服务器名字也是一样,但要去掉下划线:

^[a-z0-9-]+(\.[a-z0-9-]+)*$

好．现在只需要用”@”把两部分连接:

^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*$

这就是完整的 email 认证匹配模式了,只需要调用

eregi(‘^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*$ ’ ,$eamil)

就可以得到是否为 email 了

最后,我把另一串检查 EMAIL 的正则表达式让看文章的你来分析一下.

"^[-!#$%&\'*+\\./0-9=?A-Z^_`a-z{|}~]+'.'@'.'[-!#$%&\'*+\\/0-9=?A-Z^_`a-z{|}~]+\.'.'[-!#$%&\'*+\\./0-9=?A-Z^_`a-z{|}~]+$"

UNIT08 RE with reference to Table

字符	说明	Basic RegEx	Extended RegEx	python RegEx	Perl regEx
转义		\	\	\	\
^	匹配行首，例如'^dog'匹配以字符串 dog 开头的行（注意：awk 指令中，'^'则是匹配字符串的开始）	^	^	^	^
$	匹配行尾，例如：'^、dog$'匹配以字符串 dog 为结尾的行(注意:awk 指令中，'$'则是匹配字符串的结尾）	$	$	$	$
^$	匹配空行	^$	^$	^$	^$
^string$	匹配行，例如：'^dog$'匹配只含一个字符串 dog 的行	^string$	^string$	^string$	^string$
\<	匹配单词，例如：'\<frog' （等价于'\bfrog'），匹配以 frog 开头的单词	\<	\<	**N/A**	**N/A**（但可使用\b，例：'\bfrog'）
\>	匹配单词,例如:'frog\>'(等价于'frog\b')，匹配以 frog 结尾的单词	\>	\>	**N/A**	**N/A**（但可使用\b，例：'frog\b'）
\<x\>	匹配一个单词或者一个特定字符，例如：'\<frog\>'（等价于'\bfrog\b'）、'\<G\>'	\<x\>	\<x\>	**N/A**	**N/A**（但可使用\b，例：'\bfrog\b'
()	匹配表达式，例如：不支持'（frog）'	**N/A**（但可使用\(\)，如：\(dog\)	()	()	()
\(\)	匹配表达式，例如：不支持'（frog）'	\(\)	**不支持**（同())	**N/A**（同())	**N/A**（同())
?	匹配前面的子表达式 0 次或 1 次(等价于{0,1})，例如：where(is)?能匹配"where" 以及"whereis"	**N/A**（同\?)	?	?	?
\?	匹配前面的子表达式 0 次或 1 次(等价	\?	**N/A**（同?)	**不支持**（同?)	**N/A**（同?)

	于'\{0,1\}'），例如：'where\(is\)\? ' 能匹配 "where"以及"whereis"				
?	当该字符紧跟在任何一个其他限制符（*, +, ?, {n},{n,}, {n,m}） 后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串 "oooo"，'o+?' 将匹配单个"o"，而 'o+' 将匹配所有 'o'	**N/A**	**N/A**	**N/A**	**N/A**
.	匹配除换行符（'\n'）之外的任意单个字符（注意：awk 指令中的句点能匹配换行符）	.	.（如果要匹配包括"\n"在内的任何一个字符，请使用：'(^$)|(.)	.	.（如果要匹配包括"\n"在内的任何一个字符，请用：' [.\n] '
*	匹配前面的子表达式 0 次或多次（等价于{0, }），例如：zo* 能匹配 "z"以及"zoo"	*	*	*	*
\+	匹配前面的子表达式 1 次或多次（等价于'\{1, \}'），例如：'where\(is\)\+ ' 能匹配 "whereis"以及"whereisis"	\+	**不支持**（同+）	**N/A**（同+）	**N/A**（同+）
+	匹配前面的子表达式 1 次或多次（等价于{1, }），例如：zo+能匹配 "zo"以及"zoo"，但不能匹配 "z"	**N/A**（同\+）	+	+	+
{n}	n 必须是一个 0 或者正整数，匹配子表达式 n 次，例如：zo{2}能匹配	**N/A**（同\{n\}）	{n}	{n}	{n}
{n,}	"zooz"，但不能匹配 "Bob"n 必须是一个 0 或者正整数，匹配子表达式大于等于 n 次，例如：go{2,}	**N/A**（同\{n,\}）	{n,}	{n,}	{n,}
{n,m}	能匹配 "good"，但不能匹配 godm 和 n 均为非负整数，其中 n <= m，最少匹配 n 次且最多匹配 m 次 ，例如：o{1,3}将配"fooooood" 中的前三个 o（请注意在逗号和两个数之间不能有空格）	**N/A**（同\{n,m\}）	{n,m}	{n,m}	{n,m}

x|y	匹配 x 或 y，例如： 不支持'z|（food）' 能匹配 "z" 或"food"；'（z|f）ood' 则匹配"zood" 或 "food"	**N/A**（同 x|y)	x|y	x|y	x|y
[0-9]	匹配从 0 到 9 中的任意一个数字字符（注意：要写成递增）	[0-9]	[0-9]	[0-9]	[0-9]
[xyz]	字符集合，匹配所包含的任意一个字符，例如：'[abc]'可匹配"lay" 中的 'a'（注意：如果元字符，例如：.*等，它们被放在[ ]中，那么它们将变成一个普通字符）	[xyz]	[xyz]	[xyz]	[xyz]
[^xyz]	负值字符集合，匹配未包含的任意一个字符(注意：不包括换行符)，例如：'[^abc]' 可匹配 "Lay" 中的'L'（注意：[^xyz]在 awk 指令中则是匹配未包含的任意一个字符+换行符）	[^xyz]	[^xyz]	[^xyz]	[^xyz]
[A-Za-z]	匹配大写字母或者小写字母中的任意一个字符（注意：要写成递增）	[A-Za-z]	[A-Za-z]	[A-Za-z]	[A-Za-z]
[^A-Za-z]	匹配除了大写与小写字母之外的任意一个字符（注意：写成递增）	[^A-Za-z]	[^A-Za-z]	[^A-Za-z]	[^A-Za-z]
\d	匹配从 0 到 9 中的任意一个数字字符（等价于 [0-9]）	**N/A**	**N/A**	\d	\d
\D	匹配非数字字符（等价于 [^0-9]）	**N/A**	**N/A**	\D	\D
\S	匹配任何非空白字符（等价于 [^\f\n\r\t\v]）	**N/A**	**N/A**	\S	\S
\s	匹配任何空白字符，包括空格、制表符、换页符等等（等价于[\f\n\r\t\v]）	**N/A**	**N/A**	\s	\s
\W	匹配任何非单词字符 (等价于 [^A-Za-z0-9_])	\W	\W	\W	\W
\w	匹配包括下划线的任何单词字符（等价于 [A-Za-z0-9_]）	\w	\w	\w	\w
\B	匹配非单词边界，例如：'er\B' 能匹配 "verb" 中的'er'，但不能匹配"never" 中的'er'	\B	\B	\B	\B

\b	匹配一个单词边界，也就是指单词和空格间的位置，例如：'er\b' 可匹配"never" 中的 'er'，但不能匹配 "verb" 中的'er'	\b	\b	\b	\b
\t	匹配一个横向制表符（等价于 \x09 和 \cI）	**N/A**	**N/A**	\t	\t
\v	匹配一个垂直制表符（等价于 \x0b 和 \cK）	**N/A**	**N/A**	\v	\v
\n	匹配一个换行符（等价于 \x0a 和\cJ）	**N/A**	**N/A**	\n	\n
\f	匹配一个换页符（等价于\x0c 和\cL）	**N/A**	**N/A**	\f	\f
\r	匹配一个回车符（等价于 \x0d 和\cM）	**N/A**	**N/A**	\r	\r
\\	匹配转义字符本身"\"	\\	\\	\\	\\
\cx	匹配由 x 指明的控制字符，例如：\cM 匹配一个 Control-M 或回车符，x 的值必须为 A-Z 或 a-z 之一，否则，将 c 视为一个原义的 'c' 字符	**N/A**	**N/A**		\cx
\xn	匹配 n，其中 n 为十六进制转义值。十六进制转义值必须为确定的两个数字长，例如：'\x41' 匹配 "A"。'\x041' 则等价于'\x04' & "1"。正则中可使用 ASCII 编码	**N/A**	**N/A**		\xn
\num	匹配 num，其中 num 是一个正整数。表示对所获取的匹配的引用	**N/A**	\num	\num	
[:alnum:]	匹配任何一个字母或数字（[A-Za-z0-9]），例如：'[[:alnum:]]'	[:alnum:]	[:alnum:]	[:alnum:]	[:alnum:]
[:alpha:]	匹配任何一个字母([A－Za－z])，例如：' [[:alpha:]] '	[:alpha:]	[:alpha:]	[:alpha:]	[:alpha:]
[:digit:]	匹配任何一个数字（[0-9]），例如：'[[:digit:]] '	[:digit:]	[:digit:]	[:digit:]	[:digit:]
[:lower:]	匹配任何一个小写字母（[a-z]），例如：' [[:lower:]] '	[:lower:]	[:lower:]	[:lower:]	[:lower:]
[:upper:]	匹配任何一个大写字母（[A-Z]）	[:upper:]	[:upper:]	[:upper:]	[:upper:]
[:space:]	任何一个空白字符： 支持制表符、空格，	[:space:]	[:space:]	[:space:]	[:space:]

	例如：' [[:space:]] '				
[:blank:]	空格和制表符（横向和纵向），例如：'[[:blank:]]'ó'[\s\t\v]'	[:blank:]	[:blank:]	[:blank:]	[:blank:]
[:graph:]	任何一个可看得见的且可打印的字符（注意：不包括空格和换行符等），例如：'[[:graph:]] '	[:graph:]	[:graph:]	[:graph:]	[:graph:]
[:print:]	任何一个可打印的字符（注意：不包括：[:cntrl:]、字符串结束符'\0'、EOF 文件结束符（-1）， 但包括空格符号），例如：'[[:print:]] '	[:print:]	[:print:]	[:print:]	[:print:]
[:cntrl:]	任何一个控制字符(ASCII 字符集中的前 32 个字符，即：用十进制表示为从 0 到 31，例如：换行符、制表符等等)，例如：' [[:cntrl:]]'	[:cntrl:]	[:cntrl:]	[:cntrl:]	[:cntrl:]
[:punct:]	任何一个标点符号（不包括：[:alnum:]、[:cntrl:]、[:space:]这些字符集）	[:punct:]	[:punct:]	[:punct:]	[:punct:]
[:xdigit:]	任何一个十六进制数（即：0-9，a-f，A-F）	[:xdigit:]	[:xdigit:]	[:xdigit:]	[:xdigit:]

UNIT04 grep Family : grep、egrep、fgrep

一、grep 家族

1、grep 使用基本正则元字符集。标准的 grep 在默认情况下也支持以反斜杠开头的扩展正则元字符集。例如：\?,\+,\{,\|,\(,\)；前面没有反斜杠的扩展元字符集对于标准的 grep 无特别的含义。

2、egrep 是 grep 的扩展，用的是正则表达式元字符集的扩展集。

3、fgrep 就是 fixed grep 或者 fast grep，元字符只表示其自身的字面意义。

4、Linux 使用的是 Gnu grep，为了符合 POSIX 标准，grep 增加了-G、-E 和-F 选项，它们使你在拥有标准 grep 所提供的功能的同时还拥有 egrep 和 fgrep 的功能。Gnu grep 的使用格式如下：

grep 'pattern' filename(s)	基本正则表达式元字符集（默认）
grep -G 'pattern' filename(s)	基本正则表达式元字符集（默认）
grep -E 'pattern' filebname(s)	扩展正则表达式元字符集
grep -F 'pattern' filename(s)	无正则表达式元字符集

5、当使用 egrep 命令时 \ (或者 \ (匹配文本中的括弧，但是 (和)都是模式组部分中的特殊字符。在使用 grep 命令时，逆向也成立。

二、grep 命令(grep lm - -color=auto /proc/cpuinfo)

1、grep 的含义

grep 的名字可以追溯到 ex 编辑器。如果想启动 ex 编辑器并打印所有包含 pattern 的行，就需要输入：

: g/pattern/p

g 命令的意思是“文件中所有的行”或者“运行一个全文替代”。因为搜索模板被称为正则表达式，所以我们可以用 RE 来替换模板，命令读为：

: g/RE/p

你看到了，这就是 grep 命令的含义和名字的来源。它的意思是“全面搜索正则表达式并把找到的行打印出来（global search

regular expression (RE) and print out the line)。

使用 grep 的好处是不用启动编辑器就可以运行查找，也不需要用斜杠把正则括起来。因此它比使用 vi 和 ex 更加快捷和方便。

2、grep 怎样工作

grep 命令在一个或者多个文件中搜索字符串模板。如果模板包括空格，则必须用引号。

模板可以是一个被引用的字符串，其后面的所有字符串被看作文件名。grep 把搜索结果送到屏幕，但是不影响输入文件。

3、引号引用

在 grep 命令中输入字符串参数时，最好将其用引号括起来。这样做有两个原因，一是以防被误解为 shell 命令，二是可以用来查找多个单词组成的字符串，

例如：“jet plane”，如果不用双引号将其括起来，那么单词 plane 将被误认为是一个文件，查询结果将返回“文件不存在”的错误信息。

在调用变量时，也应该使用双引号，诸如：grep “$MYVAR” 文件名，如果不这样，将没有返回结果。

在调用模式匹配时，应使用单引号。

4、grep 的正则表达式元字符集（基本集）

基本集包括：^,$,.,*,[],[^],\<, \>。另外，Gnu 将\b,\w 和\W 作为 POSIX 新增的正则元字符予以识别。

^	^love	\(..\)b	\(love\)able
$	love$	x\{m\}	o\{5\}
.	l..e	x\{m,\}	o\{5,\}
*	*love	x\{m,n\}	o\{5,10\}
[]	[Ll]ove	\w	L\w*e
[^]	[^A-Z]ove	\W	love\W+
/<	\<love	\b	\blove\b
\>	love\>		

5、POSIX 字符类

为了处理不同的地区字符集，POSIX 加入了基本正则表达式和扩展正则表达式，表示这些类。

例如 A-Za-z0-9 本身并是正则表达式,但是[A-Za-z0-9]就是。

同样[：alnum：]只有被写成[[：alnum：]]时才是正则表达式。

这两种形式的区别在于，第一种依赖的是 ASCII 字符编码，第二种依赖的是类中的其他语言，例如瑞典语或者德语。

Bracketed Class/含义	Bracketed Class/含义
[:alnum:] 文字数字字符	[:print:] 跟非空字符一样，但是包括空格
[:alpha:] 文字字符	[:graph:] 非空字符（非空格、控制字符）
[:digit:] 数字字符	[:punct:] 标点符号
[:upper:] 大写字符	[:xdigit:] 十六进制数字（0-9、a-f、A-F）
[:lower:] 小写字符	[:space:] 所有白空格字符（新行、空格、制表符）
[:cntrl:] 控制字符	

#grep '[[:space:]]\.[[:digit:]][[:space:]]' datafile

#grep '[[:space:]]\.[[:digit:]][[:space:]]' datafile

#grep '[[:space:]]\.[[:digit:]][[:space:]]' datafile

#grep '5[[:upper:]][[:upper:]]' datafile

#grep '[[:upper:]]\.[[:upper:]][P,D]' datafile 以 P,D 结尾的

6、grep、grep-G 实例

grep -G NW file1	打印所有包含正则表达式 NW 的行。
grep NW d*	打印所有以 d 开头的文件中且包含正则表达式 NW 的行
grep '^n' file1	打印所有以 n 开头的行。^表示锚定行的开头。
grep '4$' file1	打印所有以 4 结束的行。$表示锚定行的结尾。
grep TB Savage file1	第一个参数是模板，其他的参数是文件名
grep 'TB Savage' file1	打印所有包含模板 TB Savage 的行。
grep '5\..' file1	第一个是 5，紧跟着一个点，再后是任意一个字符
grep '\.5' file1	打印所有包含字符串“.5”的行。
grep '^[we]' file1	打印所有以 w 或者 e 开头的行。
grep '[^0-9]' file1	括号内的^表示任意一个不在括号范围内的字符。
grep '[A-Z][A-Z][A-Z]' file1	打印所有包含前两个字符是大写字母，后面紧跟着一个空格及一个大写字母的字符串的行。例如，TB Savage 和 AM Main。

grep ’ss*’ file1 打印所有包含一个或者多个 s 且后面跟有一个空格的字符串的行。比如，Charles 和 Dalsass。
grep ’[a-z]\{9\}’ file1 打印所有包含每个字符串至少有 9 个连续小写字符串的行。
grep ’\(3\)\.[0-9].*\1 *\1 ’ file1 第一个字符是 3，紧跟着一个句点，然后是任意一个数字，然后是任意个数字，然后是一个 3，然后是任意个制表符，然后又是一个 3。因为 3 在一对圆括号中，它可以被后面的\1 引用。
grep ’\<north ’ file1 所有包含以 north 开始的单词的行。 grep ’\bnorth\b ’ file1 在所有 Gnu 版本的 grep 中，\b 是单词分界符
grep ’^n\w*\w ’ file1 第一个字符是 n，紧跟着是任意个字母或者数字字符，然后是一个非字母数字字符。在各种 Gnu 版本的 grep 中，\w 和\W 都是标准的单词匹配符。
grep ’\<[a-z].*n\> ’ file1 第一个字符是一个小写字母，紧跟着是任意个字符，然后以字符 n 结束。注意.*，它表示任意字符，包括空格。
#ls -l|grep ’^[^d]’ 不匹配行首

三、grep 与选项(grep 〈参数〉 〈正则〉 〈文件名〉)

1. -A NUM，--after-context=NUM 除了列出符合行之外，并且列出后 NUM 行。 $grep-A 1 panda file 从 file 中搜寻有 panda 样式的行，并显示该行的后 1 行
2. -a 或--text grep 原本是搜寻文字文件，若加上-a 参数则可将二进制档案视为文本文件搜寻，相当于--binary-files=text 这个参数。 $grep-a panda mv
3. -B NUM，--before-context=NUM 与 -A NUM 相对，同时输出匹配行的前 num 行。 $grep-B 1 panda file 从 file 中搜寻有 panda 样式的行，并显示该行的前 1 行
4. -b, --byte-offset 列出样式之前的内文总共有多少 byte .. 根据上下文定位磁盘块时有用 $grep-bpanda file 显示结果类似于: 66:pandahuang
5.-c 输出匹配行的计数只显示符合的总行数。加上-v,--invert-match 显示不符合的总行数。 #grep -c ’^ *$’ ch04 输出所有包含空行的行的数目
6. -C [NUM], -NUM, --context[=NUM] 列出符合行及上下各 NUM 行，默认值是 2。 $grep-C[NUM] panda file
7. -E, --extended-regexp 采用规则表示式去解释样式。
8. -f FILE, --file=FILE 档案的一行为一个样式。然后采用档案搜寻。 $grep-f newfile file //newfile 为搜寻样式文件
9. -G, --basic-regexp 将样式视为基本的规则表示式解释。(此为预设)
10. -i, --ignore-case 匹配时忽略大小写 $grep-i panda mv
11. -L, --files-without-match 显示出没有符合的文件名称。

12. -l, --files-with-matches 打印匹配模板的文件清单 #grep -l ‘\<PATH’ /etc/*输出所有包含词的文件名
13. -n, --line-number 在显示行前，标上行号。 #grep -n ‘\<root\> /etc/passwd 显示行号；严格匹配元字符：\<\> #grep -n /etc/passwd
14. -q, --quiet, --silent 只返回状态，0 则表示找到了匹配的行 #grep -q nstall install.log && echo $? 若找到 nstall 则传回真值
15. -r, --recursive-递归，到子目录中搜索，此相当于 -d recsuse 参数。 #grep – ir abc /usr 忽略大小写；递归查找
16. -v, --invert-match 显示除搜寻样式行之外的全部。 #grep -v -f file1 file2 && grep -v -f file2 file1 删除两个文件相同部分 #grep -v ^# /etc/initab|grep -v ^$
17. -w, --word-regexp 执行单词搜索，完全符合该"单词"的行才会被列出，相当于\<和\> #grep -w /hi/etc/fstab 输出所有包含词 reg-exp 的行 #grep “student\>” /ur/share/dict/words 精确匹配
18.-# 同时显示匹配行的上下#行;Grep -2 vim install.log

四、egrep 或 grep-E(行被限制在 2048 字节)

元字符	例子
+	重复至少“1 个”前一个 RE 字符 #egrep – n ’ go+d’ file1
?	重复至少“0 个”前一个 RE 字符 #egrep ’2\.?[0-9]’ datafile 一个 2 后跟 0 或一个.，再跟一个数
|	用“or”方式找出字符串 #egrep -v ’^$|^#’ file1 #who|egrep -v ’ ^(matty|pauline)’ ^符号排除字符串
()	找出“群组”字符串 #egrep ‘(shutdown|reboot) (s)?’ file1 #echo ’AxyzxyzxyzC’|egrep ’A(xyz)+C’
(..)(…)\1\2	\(love\)ing\1rs 保存为变量并替换
x{m,n}	o\{5\} o\{5,\} o\{5,10\} #egrep ‘[a-z]{3,5}’ file1 长度是 3～5 以小写字母组成的字串 #egrep ‘go{2,}gle’ some-file 前面的字符>2 个 #egrep ’^ [a-z]{5}’ file1 行首连续 5 个小写字母 grep -n ’[0-9]\{6,\}$’ datebook 打印工资是 6 位数的行并给出行号

UNIT09 Built-in Variables

env|grep SHELL,set|grep SHELL,echo ${SHELL}

一、SHELL 标准变量

LOGNAME	TERM	HOSTTYPE——i686
UID	SHELL	MACHTYPE——i686-redhat-linux-gnu
EUID	BASH	OSTYPE——linux_gnu
GROUPS	BASH_VERSION	HISTCMD 下一指令的编号
PATH	HOSTNAME	HISTSIZE
OLDPWD	PPID	HISTFILE
PWD	RANDOM 随机数	HISTFILESIZE
HOME	MAIL	SECONDS 当前 SHELL 时长
LANG	MAILCHECK	DISPLAY=X 服务器=:0.0

二、修改 shell 的分隔符

$ set | grep IFS IFS=$' \t\n'　　//shell 默认的分隔符
$ (IFS=:;a=Hello:World;echo $a)　　//結果是 Hello World，因为:现在起隔离作用

三、shell 提示符

PS1——主提示符——PS1=\S-\V$;echo $PS2——次提示符

①	\h——主机名(第一个点之前)	\H——完整主机名
②	\u——使用者账号名称	\s——shell 的名称
③	\d——周 月 日	\t——24 小时制的时间
④	\T——12 小时制的时间	\@——12 小时制的时间格式

④	\w——完整工作目录	\W——工作目录最后一节
⑥	\!——此命令的历史指令编号	\#——此命令为的编号
⑦	\v——bash 的版本号	\V——bash 的完整版本

四、特殊变量

$0	这个程序的执行名字
$n	这个程序的第 n 个参数值，n=1..9
$#	传递到脚本的参数个数
$*	传递到脚本的参数，与位置变量不同，此选项参数可超过 9 个
$$	脚本运行时当前进程的 ID 号，常用作临时变量的后缀，如 haison.$$
$!	后台运行的(&)最后一个进程的 ID 号
$@	与$#相同，使用时加引号，并在引号中返回参数个数
$-	上一个命令的最后一个参数
$?	最后命令的退出状态，0 表示没有错误，其他任何值表明有错误

```
echo "The script name is                    :`basename $0`"
echo "The first param of the script is      :$1"
echo "The second param of the script is     :$2"
echo "The tenth param of the script is      :$10"
echo "All the params you input are          :$*"
echo "The number of the params you input are:  $#"
echo "The process ID for this script is     :$$"
echo "The exit status of this script is     :$?"
```

UNIT10 Custom Variable

ApacheVersion=”httpd-2.2.p1”&&tar -xvzf $ApacheVersion.tar.gz	
echo Hi，${myname}mm…	变量有其它英、数、底线，用{}隔开$和变量名
dir2=lib&&echo /usr/$dir2/ntp	变量名后接的不是英、数、底线，不必用{}

一、变量分类

本地变量：用户自定义的变量，用set查看，(函数内部local A=c)。

环境变量：用于所有用户变量，必须用export命令导出，用env查看。

位置变量：$0(脚本名)，$1-$9:脚本参数。

特定变量：脚本运行时的一些相关信息。

1、临时变量，重启失效

shell 变量（某 shell 私有；set 验证） ABC=’/etc/sysconfig’ Bash——进入子 shell echo $ABC——无效 cd $ABC——无效	环境变最（子 shell 也有效；env 或 export 验证） export ABC=’/etc/sysconfig’ echo $ABC——有效 cd $ABC——有效

2、永久变量，重启生效；用 source 或.使之立即生效

用户变量 ~/.bashrc ~/.bash_profile（用户环境变量——path=$PATH:/eee:.）	全局变量 /etc/bashrc /etc/profile(全局环境变量——ABCD=1234) /etc/profile.d/*.sh（环境变量）

二、变量的配置(变量=; unset -f 函数名)

1、设定变量—— Linux 只有一种数据型态：字符串

等号两边有空格符可用双引号或单引号将变量内容结合起来；如 Yname=” Black Jk” ；
若要显示$等字符可用\或’’：echo \$I 或 echo ‘$I’；
变量名称中数字不能是开头字符；

在脚本开头注释部分要尽量详细的将本脚本设计功能、修改历史写清楚，最好将编写人员的联系方式也加入其中,在脚本的最初部分将环境变量设置好;
在编写 Script 时，为怕打错变量名称，造成排错上的困难，可规定变量一定要经过设定的程序才能使用 shopt -s -o nounset //nounset 变量设定过才可使用 echo $Infomix 报错 //-s 打开选项, -o 指用 set -o 设定选项 declare Infomix=50 && echo $Infomix

2、只读变量不可被 unset、值不可改、不能去掉 r 属性

b=a1 readonly b b=a2 Bash:b:readonly variable	delare -r b=a3 //设定 b 是只读变量 declare -p b //显示变量的属性 unset b //取消变量名 unset:b:cannot unset:readonly variable
declare -a ary 设定变量 ary 是一个数组 declare -I a 设定变量 a 是整数	declare -F 显示所有函数名及其属性 declare -x PATH 设定为环境变量

3、针对不同变量状态赋值

: 空值 测空值		- 负向 测不存在
= 设值 给空值		? 有问题 给空值变量设一个默认值
+ 正向 测试存在		
${待测变量:-默认值}	unset 或 null	传回“默认值”
${待测变量:=默认值}	unset 或 null	将变量值设为“默认值”
${待测变量:?提示信息}	unset 或 null	显示提示信息
${待测变量:+真值}	存在且非空	传回“真值”

三、$(( )) $( ) ${ }

1、 $( ) 与 ``

` ` 很容易与 ’ ’搞混乱;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：command1 `command2 `command3` `,换成 $( ) 就没问题了：command1 $(command2 $(command3))。

``` 基本上可用在全部的 unix shell 中使用，其 shell script 移植性比较高，而 $( ) 并不见的每一种 shell 都能使用。

**2、${ } bash 扩展替换获得不同的值(特 ${A}B)**

**(file=/dir1/dir2/my.file.txt)**

| |
|---|
| # 去左边($ 左边);% 是去右边(在 $ 右边);单一符号最小匹配；两个符号最大匹配。 |
| ${file#*/}：拿掉第一条/及其左边的字符串：dir1/dir2/my.file.txt<br>${file##*/}：拿掉最后一条/及其左边的字符串：my.file.txt<br>${file#*.}：拿掉第一个.及其左边的字符串：file.txt<br>${file##*.}：拿掉最后一个 . 及其左边的字符串：txt |
| ${file%/*}：拿掉最后条 / 及其右边的字符串：/dir1/dir2<br>${file%%/*}：拿掉第一条 / 及其右边的字符串：(空值)<br>${file%.*}：拿掉最后一个 . 及其右边的字符串：/dir1/dir2/my.file<br>${file%%.*}：拿掉第一个 . 及其右边的字符串：/dir1/dir2/my |
| ${file:0:5}：提取最左边的 5 个字节：/dir1<br>${file:5:5}：提取第 5 个字节右边的连续 5 个字节：/dir2 |
| ${file/dir/path}：将第一个 dir 提换为 path：/path1/dir2/my.file.txt<br>${file//dir/path}：将全部 dir 提换为 path：/path1/path2/my.file.txt<br>${#var}：可计算出变量值的长度：<br>${#file}：得到 27，/dir1/dir2/my.file.txt 是 27 字节... |

**3、$(( ))它是用来作整数运算的**

| | | | |
|---|---|---|---|
| $((12+i)) | $((12+$i)) | $((12+${i})) | $(($[j:-8]+2)) |
```

UNIT11 Here Document

```
# /bin/bash
: << REM-REM1
if [ $# -ne 1 ]; then
        echo "Usage: $0 PROCESS-NAME"
        exit 1
fi
REM-REM1
```

用Here Document 做多行批注

一、变量替换及抑制

Here Document 也支持变量替换——在输入的内容中如果有变量，bash 在转向前，会先替换变量值。

to=’ To:you@example.com.cn’ em=’ 20090310.txt’ Cat >$em << “HERE” //表示这个 Here Document 拥有和双引号一样的特性，即支持变量扩展。 $To HERE	#vim e.sho cat << - ‘HERE’ //-表示抑制各行首 TAB 的的作用 // ‘’ 即按原样输出，不做任何变量替换 --line1 --$wow HERE

二、利用 Here Document 打包 c 的原始码

Cat > ‘hello.c’ << EOF
#include <stdio.h>
Int main( )
{
Printf(“Hello world;\n”);
Return 0;
}
EOF
	gcc -o hello hello.c
	If [$? -eq 0];then
	Echo “执行 hello...”
	Echo
	./hello
	Else
	Echo ‘Complile ERROR:hello.c’
	fi

UNIT12 Redirection & PiPes

command > file	Write standard output of *command* to *file*
command 1> file	Write standard output of *command* to *file* (same as previous)
command 2> file	Write standard error of *command* to *file* **(OS/2 and NT)**
command > file 2>&1	Write both standard output *and* standard error of *command* to *file* **(OS/2 and NT)**
command >> file	Append standard output of *command* to *file*
command 1>> file	Append standard output of *command* to *file* (same as previous)
command 2>> file	Append standard error of *command* to *file* **(OS/2 and NT)**
command >> file 2>&1	Append both standard output *and* standard error of *command* to *file* **(OS/2 and NT)**
commandA | commandB	Redirect standard output of *commandA* to standard input of *commandB*
commandA 2>&1 | commandB	Redirect standard output *and* standard error of *commandA* to standard input of *commandB*
command < file	*command* gets standard input from *file*
command 2>&1	*command*'s standard error is redirected to standard output **(OS/2 and NT)**
command 1>&2	*command*'s standard output is redirected to standard error **(OS/2 and NT)**

一、重定向针对终端过滤器、不对交互工具和编辑器

Linux 命令:筛选器 ls 等;编辑器 vim 等;交互工具 mc 等

```
[root@station1 ~]# ll /dev/std*
lrwxrwxrwx 1 root root 15 Jun 12 01:47 /dev/stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root 15 Jun 12 01:47 /dev/stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root 15 Jun 12 01:47 /dev/stdout -> /proc/self/fd/1
```

流	描述符	缩写	转向	默认
标准输入	0	Stdin	<、<<	与 KB 相连
标准输出	1	Stdout	> 、1>; >>、1>>	与显示器相连
标准错误	2	Stderr	2>、2>>	与显示器相连
标准错误 定向到 标准输出			2>&1; &>	

1、终端程序一般从一个单一源以流的形式读取信息，如键盘。

在 linux 中，输入流被称作标准输入（Standard In=stdin），用>和>>重定向。

2、终端程序通常把信息作为流写入单一目的地，如显示器。

在 linux 中，输出流被称作标准输出（Standard Out=stdout）。

Mail -s “name” admin@a.com <a.txt

3、程序通常将出错状况信息报告给一个名叫标准错误（Standard Error=stderr）目的。

二、管道——Unix 哲学：联合使用多个简单命令

```
[root@station3 ~]# cat < /etc/termcap > foo1
[root@station3 ~]# cat   /etc/termcap > foo2
[root@station3 ~]# cp    /etc/termcap   foo3
[root@station3 ~]# cat foo1|wc -l
19092
[root@station3 ~]# cat foo2|wc -l
19092
[root@station3 ~]# cat foo3|wc -l
19092
```

```
# cat unsort.txt|sort
# cat unsort.txt|lp
# cat unsort.txt|sort>sorted.txt
# ls -l|grep ‘^d’ |wc -l
# ls -l|grep ‘^-’ |wc -l
# find /etc -size +100k 2> /dev/nul|grep
```

三、重定向举例

```
# ls  >  file      ls  >>  file
# ls 1>  file      ls 1>>  file

# ld 2>  file      ld 2>>  file
# lll >a 2>&1
# lll >a 2>&1   lll >>file 2>&1
# dir 1>&2  dir >a 1>&2   dir >>a 1>&2

# ls|tee   ls|tee >file 命令 a 的 stdout 变作命令 b 的输入
# ld|tee
# ld |tee >a——cat a：a 无内容
# ld 2>&1 | tee >a——cat a：a 有内容
```

```
[root@localhost ~]# ld|tee a
ld: no input files
[root@localhost ~]# cat a
[root@localhost ~]# ld|tee >a
ld: no input files
[root@localhost ~]# cat a
[root@localhost ~]# ld 2>&1|tee >a
[root@localhost ~]# cat a
ld: no input files
```

四、\；||；&&

1、&&成功符=成功则运行；||失败符=失败则运行

```
# ls /tmp/a||mkdir /tmp/a&&cd /tmp/a 运行两次
# ls /tmp/a||mkdir /tmp 第一次执行的路线
# ls /tmp/a &&cd /tmp/a 第二次执行的路线
```

```
# ping -c 100 -w 15 pc1 &> /dev/null&& echo “pc1 up”\
||echo “pc1 off;exit 1”
```

2、\在前叫跳脱符；\在后叫换行符

```
# echo “a \
b ”
```

```
# ./configure \
--with-apache=../apache-$ApacheVersion \
--with-mysql=$MYSQLHOME&&
```

3、单引号不转义与双引号四种会转义的情况$,\,!,`

```
# echo $PATH——显示变量值，等价于 echo “$PATH”
```

```
# echo \$PATH——显示‘$PATH’字符串，等价于 echo ‘$PATH’
```

UNIT13 Process Management

一、ps

1、应用举例

a 显示所有用户的所有进程	l 长格式输出
x 显示无控制终端的进程	u 按用户名和启动时间显示进程
r 显示运行中的进程	j 用任务格式来显示进程
ww 避免详细参数被截断	f 用树形格式来显示进程
# ps auxf |grep httpd	用 f 参数；父与子关系一目了然
#ps –C bash	看 bash 程序生成所有进程号
#ps –u root	看 root 用户启动的所有进程号
#ps -p 1 #ps -p $(/sbin/pidof init)	用 pid 查软件名
#ps –eo pid,%cpu,comm.,tty|grep ttys0	自定义格式的进程

2、ps aux

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   2072   632 ?        Ss   20:14   0:01 init [3]

root         2  0.0  0.0      0     0 ?        S<   20:14   0:00 [migration/0]
root         3  0.0  0.0      0     0 ?        SN   20:14   0:00 [ksoftirqd/0]
root         4  0.0  0.0      0     0 ?        S<   20:14   0:00 [watchdog/0]
root       245  0.0  0.0      0     0 ?        S    20:14   0:00 [khungtaskd]
```

```
[root@station24 ~]# ps axf|grep httpd|grep -v grep
 5336 ?        Ss     0:00 /usr/sbin/httpd
 5337 ?        S      0:00  \_ /usr/sbin/httpd
 5338 ?        S      0:00  \_ /usr/sbin/httpd
```

NI	进程的 NICE 值 NI	T	停止或被追踪
l	多进程的（使用 CLONE_THREAD）	W	进入内存交换;内存分页不足
<	优先级高的进程	N	优先级较低的进程
L	有些页被锁进内存	WCHAN	正在等待的进程资源;
+	位于后台的进程组	R	正在执行中
X	死掉的进程（从来没见过）;	s	进程的领导者，之下有子进程

二、pstree

```
[root@station24 ~]# pgrep "init"
1
[root@station24 ~]# pgrep -l "log"
3196 syslogd
3199 klogd
[root@station24 ~]# pgrep -l -U root -t tty1
4613 mingetty
[root@station24 ~]# pstree -aup
init,1
  ├─/usr/bin/sealer,4920 -E /usr/bin/sealert -s
  ├─acpid,3400
  ├─atd,3650
  ├─auditd,3168
  │   ├─audispd,3170
  │   │   └─{audispd},3171
  │   └─{auditd},3169
```

查看进程树（引导系统时，linux 内核的一个职责是启动第一个进程，一般是/sbin/init，因为一个业已存在的进程继续派生，所有其它进程得以启动，由于除了第一个进程之外每个进程都是由派生创建的，在进程之间存在着一个详细定义父子关系的家谱，那就是进程树，由内核启动的第一个进程位于进程树的根部）

-A	进程树之间以 ASCII 字符来连接；
-p	同时列出每个进程的 PID；
-u	显示用户名；
-a	显示每个程序的完整指令，包括路径，参数；
-c	不使用精简表示法；
-h	列出树状图时，特别标明现在执行的程序；
-l	采用长列格式显示树状图；
-n	用程序识别码排序，默认是以程序名称来排序；
-p	显示程序识别码；
-U	使用 UTF-8 列出绘图字符；

三、pgrep+pkill+killall+kill

Pkill,killall 这些工具在强行终止数据库服务器时，会让数据库产生更多的文件碎片，当碎片达到一定程度的时候，数据库就有崩溃的危险。 当然对占用资源过多的数据库子进程，我们可以用 kill 杀掉。

```
[root@server1 ~]# pgrep httpd|awk 'BEGIN{ORS=" "}{print}END{print "\n"}'
4200 4512 4513 4514 4515 4516 4517 4524 4525
 [root@server1 ~]# pidof httpd
4525 4524 4517 4516 4515 4514 4513 4512 4200
[root@server1 ~]# pgrep  -l httpd|awk 'BEGIN{ORS=" "}{print}END{print "\n"}'
4200 httpd 4512 httpd 4513 httpd 4514 httpd 4515 httpd 4516 httpd 4517 httpd 4524 httpd 4525 httpd
 [root@server1 ~]# pgrep -o httpd
4200
[root@server1 ~]# pgrep -n httpd
4525
[root@server1 ~]# pmap `pgrep httpd`|tail -n 1
 total    23120K
[root@server1 ~]# pkill httpd
[root@server1 ~]# pgrep httpd
```

kill（杀掉单个进程）

#kill -9 5901，5902,5903 //强制杀死多个进程

#kill -9 5901—5903 //强制杀死 5901—5903 三进程

#kill -9 0 //强制杀死所有后台程序

killall 程序名(一次性杀死所有对应程序的进程)

-i，--interactive：在给进程发送信号之前询问用户

-w，--wait：等到所有的进程都被取消后在返回

pkill 程序名(通过 ps 或 pgrep 来查看哪些程序在运行)

-l 列出程序名和进程 ID；

-o 进程起始的 ID；

-n 进程终止的 ID

xkill 是在桌面用的杀死图形界面的程序

比如当 firefox 出现崩溃不能退出时，用 xkill 点鼠标就能杀死 firefox 。当 xkill 运行时出来和个人脑骨的图标，哪个图形程序崩溃一点就 OK 了。如果想终止 xkill，就按右键取消。

UNIT14 Auto http&ftp&rsync

一、ftp 命令自动传输

1、手动 FTP

```
ftp ftp.kernel.org
anonymous
zcs@example.com
cd pub
get README
quit
```

2、半自动 FTP

```
#cat getreadme.ftp
ftp ftp.kernel.org
anonymous
zcs@example.com
cd pub
get README
quit
```

```
#ftp ftp.kernel.org < getreadme.ftp
ftp 等待键盘输入密码
读取到 zcs@example.com 时显示? Invalid command。
```

2、全自动 FTP

```
cat .netrc（提供用户名密码）
Default login anonymous password zcs@example.com
```

```
ftp ftp.example.com < getreadme.ftp
```

二、wget 用法详解

wget 是一个命令行工具，用于批量下载文件，支持 HTTP 和 FTP。

#wget http://server1/pub/dns.tar.gz -o /tmp/dns.tar.gz #wget -r http://place.your.url/here(下载整个站点,-r=--recursive)
只下载一类文件 #wget -m --reject=gif http://target.web.site/subdirectory
用 wget 下载整个网站 wget -r -p -np -k \ http://sources.redhat.com/gdb/current/onlinedocs/gdb_toc.html # 或者 wget -m http://www.tldp.org/LDP/abs/html/
断点续传是自动的 # wget -c http://the.url.of/incomplete/file (-c=--continue)
批量下载 #wget -F -i download.txt(-F=--force-html;-i file1 等价于--input-file=file1)

三、Rsync+SSH+cron 自动异地加密备份

1、首先要先对 192.168.0.3 把 Rsync 的 Server on 起来...

#chkconfig rsync on
#vi sync rsync -avlR --delete -e ssh 192.168.0.3:/var/lib/mysql /backup rsync -avlR --delete -e ssh 192.168.0.3:/var/www/html /backup
chmod 700 sync && ./sync receiving file list ... done...donewrote 16 bytes read 107 bytes 82.00 bytes/sectotal size is 0 speedup is 0.00receiving file list ... done...donewrote 16 bytes read 921 bytes 624.67 bytes/sectotal size is 308331 speedup is 329.06

2、ssh 参数意义如下

-a, --archive	It is a quick way of saying you want recursion and want to preserve almost everything.
-l, --links	When symlinks are encountered, recreate the symlink on the destination.

-R, --relative	Use relative paths. 保留相对路径...才不会让子目录跟 parent 挤在同一层...
--delete	是指如果 Server 端删除了一文件，那客户端也相应把这一文件删除，保持真正的一致。
-e ssh	建立起加密的连接。

四、RSYNC 同步/备份

rsync 服务器默认以 Xinetd 运行，还可常驻模式（daemon）运行，若一台主机以 daemon 模式运行 rsync，一般称其为 rsync 服务器。rsync 的 C/S 方式运行方式概述如下：用户验证由服务器负责，用户口令文件在默认配置文件/etc/rsyncd.conf 中指明。

1、服务器端配置# vi /etc/rsyncd.conf

uid = nobody	[root]
gid = nobody	path = /home/sites/compshop
use chroot = yes	auth users = root
max connections = 4	uid = root
pid file = /var/run/rsyncd.pid	gid = root
lock file = /var/run/rsync.lock	secrets file = /etc/rsyncd.secrets
log file = /var/log/rsyncd.log	read only = no

#chmod 600 /etc/rsyncd.conf

#vi /etc/rsyncd.secrets

root:123456

#chmod 600 /etc/rsyncd.secrets

#/usr/bin/rsync - -daemon

#echo "/usr/bin/rsync - -daemon" >> /etc/rc.local

#ps -ef | grep rsync 找出进程杀死可停止服务

2、客户端的配置

#vi /etc/rsyncd.secrets:123456（rsync 服务器登录密码）
#chmod 600 /etc/rsyncd.secrets
与服务器端同步 avqz

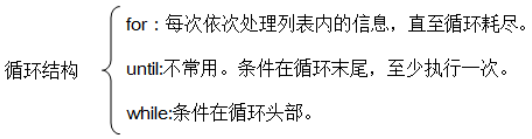
#rsync - vazu - progress - delete - - password - file=/etc/rsync.secret /tmp/old root@192.168.23.102::root

3、rsync 有六种不同的工作模式

拷贝本地文件。 如：rsync -a /data /backup
使用一个远程 shell 程序(如 rsh、ssh)来实现将本地机器的内容拷贝到远程机器。 如：rsync -avz *.c 192.168.0.3:dst
使用一个远程 shell 程序(如 rsh、ssh)来实现将远程机器的内容拷贝到本地机器。 如：rsync -avz 192.168.0.1:src/bar /data
从远程 rsync 服务器中拷贝文件到本地机。 如：rsync -av root@172.16.78.192::www /databack
从本地机器拷贝文件到远程 rsync 服务器中 如：rsync -av /databack root@172.16.78.192::www
列远程机的文件列表。 如：rsync -v rsync://172.16.78.192/www # rsync -vazu -progress /home terry@192.168.100.21:/terry/ #rsync - vazu - - progress - - password - file=/etc/rsync.secret terry@192.168.100.21:/terry/ /home

UNIT15 Loop structure

loop 就是 script 中的一段在一定条件下反复执行的代码。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。



一、for loop

for 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。

1、for var in one two three

```
do
echo ----------
echo '$var is '$var
done
```

上例的执行结果将会是：

→for 会定义一个叫 var 的变量，其值依次是 one two three。
→ 有 3 个变量值，因此 do 与 done 间的命令会被执行 3 次。
→每次循环均用 echo 产生三行句子。 第二行中不在 hard quote 内的$var 依次被替换为 one two three。
→当最后一个变量值处理完毕，循环结束。
不难看出，在 for loop 中，变量值的多寡，决定循环的次数。

2、for loop 用于处理"清单"(list)项目非常方便

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：

3、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
```

二、while loop

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，而取决于 while 后面命令行的 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。

1、num=1

```
while ((num<=10))
do
echo “num is $num”
num=$(($num + 1))
done
```

分析上例：

→在 while 之前，定义变量 num=1 。
→然后测试(test) $num 是否小于或等于 10 。
→结果为 true ，于是执行 echo 并将 num 的值加一。
→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。
→直到 num 为 10+1=11 时，测试才会失败... 于是结束循环。
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

2、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)。

三、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
```

四、select+break+continue

1、break 是结束 loop

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。

```
#! /bin/bash
#select1.sh
PS3='Please Select:*'
select f in *
do
   echo "You entered $REPLY refers to $f"
   break
done
```

```
#! /bin/bash
# select2.sh
PS3='Please select:'
menu="/ /root /etc /home"
select f in $menu
do
   echo "You entered $REPLY refers to  $f"
   break
```

2、return 是结束 function

3、exit 是结束 script/shell

4、continue 则与 break 相反：强迫进入下一次循环动作。

```
#! /bin/bash
#continue1.sh
for ((i=1;i<=10;i++))
do
   if [ $i -eq 6 ];then
      continue
   fi
      echo $i
done
```

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。

UNIT16 Select condition structure

选择结构
- if语句: if then else 提供条件测试
- case 语句 ：允许匹配模式、单词或值

一、if 语句: if then else 提供条件测试

1、格式

格式 1 if [条件] then 命令 fi	格式 2 if [条件] ; then 命令 fi
格式 3 if [条件] then 命令 1 else 命令 2 fi	格式 4 if [条件 1] then 命令 1 elif [条件 2] then 命令 2 else 命令 3 fi

2、示例

```
#!/bin/sh
#ifTest
echo -e "Enter the first integer:\c"
read FIRST
echo -n "Enter the second integer:"
```

```
read SECOND
if [ "$FIRST" -gt "$SECOND" ]
    then
       echo "$FIRST is greater than $SECOND"
    elif [ "$FIRST" -gt "$SECOND" ]
then
    echo "$FIRST is less than $SECOND"
else
    echo "$FIRST is equal to $SECOND"
fi
```

3、判断是文件还是目录

```
#!/bin/sh
file=./testing
if [ -d $file ]
then
   echo "$file is a directory"
elif [ -f $file ]
then
   if [  -r $file ]
   then
      echo "You have read permission on $file."
   fi
else
   echo  "$file is neither a file nor a directory. "
fi
```

二、case 语句：允许匹配模式、单词或值

1、格式

```
   shopt -s nocasematch
   read 值
   case $值 in
```

样式 1)命令 1;;

样式 2)命令 2;;

*)命令 n;;

esac

取值后面必须为单词 in,每一个模式必须以右括号结束。取值可以为变量或常数。取值检测匹配的每一个模式，一旦模式匹配，其间所有命令开始执行直至;;。执行完匹配模式相应命令后不再继续其他模式。如果无一匹配模式，使用*号捕获该值，再接受其他输入。

2、示例

```
#!/bin/bash
#
echo "Please choose install A,Z or Q:"
echo "[A]pache"
echo "[Z]abbix"
echo "[Q]uit"
read choose
case $choose in
a|A) echo  "apache configure...";;
z|Z) echo  "zabbix configure...";;
q|Q) exit;;
esac
```

3、高级

?（样式）	符合 0 个或 1 个括号里的样式，就算对比符合。
*（样式）	符合 0 个以上括号里的样式，就算对比符合。
+（样式）	符合 1 个以上括号里的样式，就算对比符合。
@（样式）	符合其中 1 个括号里的样式，就算对比符合。
!（样式）	不符合其中 1 个括号里的样式，就算对比符合。

```
#! /bin/bash
shopt  -s extglob
read yname
read $yname
case $yname in
j@(ac|ar)k|joe)echo  ‘Long time to see.’ ;;
*) echo  ‘Hi!’ ;;
esac
```

三、test 条件测试

test 命令可以和多种系统运算符一起使用。这些运算符可分为 4 类：整数运算符、字符串运算符、文件运算符和逻辑运算符。

1、整数运算符：用来判断数值表达式的真假

int1 -eq int2	INTEGER1 is equal to INTEGER2
int1 -ne int2	INTEGER1 is not equal to INTEGER2
int1 -ge int2	INTEGER1 is greater than or equal to INTEGER2
int1 -gt int2	INTEGER1 is greater than INTEGER2
int1 -lt int2	INTEGER1 is less than INTEGER2
int1 -le int2	INTEGER1 is less than or equal to INTEGER2

2、文件运算符：用来判断文件是否存在、类型及属性

①文件类型判断:如 test -e file1 表示普通文件 file1 存在否

-e FILE	FILE exists
-f FILE	FILE exists and is a regular file
-d FILE	FILE exists and is a directory
-b FILE	FILE exists and is block special
-c FILE	FILE exists and is character special
-S FILE	FILE exists and is a socket
-p FILE	FILE exists and is a named pipe
-L FILE	FILE exists and is a symbolic link (same as -h)

②文件权限侦测:如 test -r file1 测试 file1 可读否

-r FILE	FILE exists and read permission is granted
-w FILE	FILE exists and write permission is granted
-x FILE	FILE exists and execute (or search) permission is granted
-u FILE	FILE exists and its set-user-ID bit is set
-g FILE	FILE exists and is set-group-ID
-k FILE	FILE exists and has its sticky bit set
-s FILE	FILE exists and has a size greater than zero

③两个档案之间的比较，如 test file1 -nt file2

FILE1 -nt FILE2	FILE1 is newer (modification date) than FILE2
FILE1 -ot FILE2	FILE1 is older than FILE2
FILE1 -ef FILE2	FILE1 and FILE2 have the same device and inode numbers

3、字符串运算符：用来判断字符串表达式的真假

test -z STRING	the length of STRING is zero？若 string 为空字符串，则为 true
test -n STRING	the length of STRING is nonzero？ (-n 亦可省略)。
STRING1 = STRING2	the strings are equal，若相等，则回传 true
STRING1 != STRING2	the strings are not equal

4、逻辑操作符(多重条件判定):如 test -r filename -a -x file1

EXP1 -a EXP2	both EXPRESSION1 and EXPRESSION2 are true
EXP1 -o EXP2	either EXPRESSION1 or EXPRESSION2 is true
test ! -x file	当 file 不具有 x 时，回传 true

UNIT17 Shell Functions

1、函数定义 functionName(){命令序列; } 或 function functionName { COMMANDS; }
2、函数调用 functionName functionName 位置参数
3、函数返回 return 用函数中执行的上一个命令的退出码返回; return [value] 用给定的 value 值返回;
4、全局函数 export 命令可以将函数说明为全局函数，使其可以被子 shell 继承。
5、函数共享 把要共享的函数单独放在一个文件中，然后在要使用该函数的脚本中，在开始位置用以下格式的命令读取该文件。 .fileNme Source fileName
6、变量限制 local 变量名 限制函数只能本地用于当前函数。

```
#!/bin/sh
printmsg( ){
    prefix=“$1”
    shift
    echo “$prefix:$@”
}
printmsg “$@”        #调用函数
```

所有函数在使用前必须定义，这意味着必须将函数放在脚本开

始部分，直至 shell 解释器首次发现它，才可以使用。

调用函数仅使用其函数名即可，要传给函数的变量跟在函数后面。

函数里面定义的变量以下划线(_)开始。

函数可以放在同一个文件中作为一段代码，也可以放在只包含函数的单独文件中，文件也必须以#!/bin/sh 开头。

示例 1

```
#!/bin/sh
#funTest
#to test the function
DATE=`date`
Hello()
{
  echo “Hello,today is $DATE”
}
Hello
```

Shell 脚本也有自定义函数的功能。当脚本变得很大时，可将脚本文件中常用的程序写成函数，这样可以使脚本更小、更易于维护，定义函数的语法如下：

```
fname  ()   {
       Shellcommands
           }
```

例：求命令行中输入的数值组的最大的数，将文件存为 maxvalue。

```
#!/bin/bash
max ()
{
while test $1
do
        if test $maxvalue
        then
```

```
            if test $1  -gt $maxvalue
            then
                    maxvalue=$1
            fi
        else
             maxvalue=$1
        fi
        shift
    done
    return $maxvalue
    }
   max $*
   echo "Max Value is :$maxvalue"
   #end
```

执行结果：

```
$maxvalue 239 32 78 7 60 20 150 345 3
Max Value is :345
```

3. 填写执行结果

```
s=0;i=1
while test $i  -le 5
do
let s=$s+$i*$i
let i=$i+1
done
echo "s= $s"
```

一组命令集或语句形成的可用块称为 shell 函数

练习：

step1　在脚本程序中我们首先定义一个 yes_or_no 函数：

```
#!/bin/sh
yes_or_no() {
  echo "Is your name $* ?"
  while true
  do
    echo -n "Enter yes or no: "
    read x
    case "$x" in
      y | yes ) return 0;;
      n | no ) return 1;;
      *)       echo "Answer yes or no" ;;
    esac
  done
}
```

Step2:接下的是这个程序的主要部分：

```
echo "Original parameters are $*"
if yes_or_no
 then
  echo "Hi $1, nice name"
else
  echo "Never mind"
fi
exit 0
```

任务一:给下面的程序添加注释

```
 while true
do
echo "List Directory.....1"
echo "Change Directory.....2"
echo "Edit File.....3 "
echo "Remove File.....4"
echo "Exit Menu.....5"
echo -n "please input you choice:"

read choice


  case $choice in
    1) ls;;
    2) echo -e "\nEnter target directory"
    read dir
    cd $dir ;;
     3) echo -e "\nEnter file name"
    read file
    vi $file ;;
    4) echo  -e"\nEnter file name"
    read file
    rm $file ;;
    q|Q|5) echo -e "\nGoodbye"
     exit 1;;
    *) echo -e "\nillegal Option,please try again";;
     esac
done
```

任务二：设计一个打字训练游戏。要求：

1、主菜单功能为当用户输入“1”，练习输入数字； 输入 “2”，练习输入字母；输入 q 则退出；当有其他输入提醒用户重新输入。

2、给出待输入的字符串，待用户输入后判断其是否正确。

UNIT18 Array

一、创建和赋值

declare -a 数组名

如果不给出数组名，则显示当前定义的所有数组和数组的值。

例：$declare -a names

$names[0]=“David”

$names[1]=“Richard”

$names=(“David” ”Richard”)

二、引用和销毁

A=(a b c def) ，则是将 $A 定义为组数

1、bash 的 array 处理方法

${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[@]} 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 ...

2、销毁：

unset 数组名 销毁数组

unset 数组名[i] 收回第 i 个元素

UNIT19 VIM Introduction

由于被广泛移植，无论是 DOS，还是 AIX，都能见到 VI 的身影。作为开源世界最重要的编辑器之一（另一个是 Emacs），VI 以其强大的功能和无穷的魅力将使您终生受益。

vim 是一个『程序开发工具』，它加入了很多额外的功能，例如支持正则表示法的搜寻架构、多档案编辑、区块复制等等。

一、VI/vim 的历史

ed 是 Unix 上最古老的编辑器，它最初是 Unix 之父 Ken Thompson 编写的，他第一次在 ed 中应用了 regular expression，这个创举将 RE 理论带入实践，对 Unix 界造成了深远的影响。实际上 ed 是受伯克利大学的 QED 编辑器的影响，Ken 是从那里毕业的。
ex 是 ed 的扩展，实际上 vi 构建在 ex 之上，vi 引入了我们熟悉的全屏编辑模式。
vi：1976 年左右 Bill Joy 开发了 vi，他也是伯克利大学的毕业生，后来他跟其他人一起成了了 Sun Microsystems 公司并成为了 Sun 的首席科学家。 一开始 Bill 开发了 ex，尔后开发了 vi 作为 ex 的 visual interface，
vim 是 vi 的扩展，支持自动补全、彩色语法等功能。

二、为什么要学习 VI

1、所有 Unix Like 系统都会内建 vi 文本编辑器，其他的文本编辑器则不一定会存在；

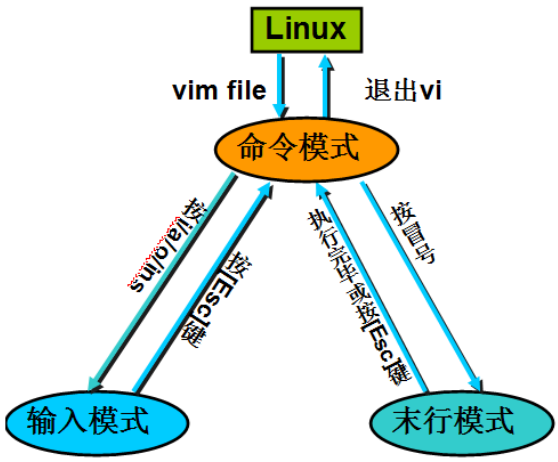
2、 很多个别软件的编辑接口都会主动呼叫 vi（例如未来会谈到的 crontab, visudo, edquota 等指令）；

3、 vim 具有程序编辑能力，可以主劢的以字体颜色辨别语法的正确性，方便程序设计；

4、 因为程序简单，编辑速度相当快速。

三、Vim 的 mode（可以简单地理解为“状态”）

与大部分其它编辑器不同，进入 Vim 后，缺省状态下键入的字符并不会插入到所编辑的文件之中。Vim 的模式概念非常重要。




```
1. Insert mode                          insert-index
2. Normal mode                          normal-index
3. Visual mode                          visual-index
index.txt.gz [Help][RO]                 6,9-65        Top
:help index
```

一般模式： vi 打开档案就直接进入一般模式了。在这个模式中， 你可以使用『上下左右』按键来移动光标，你可以使用『删除字符』和『删除整行』来处理档案内容， 也可以使用『复制、贴上』来处理你的文件数据。
编辑模式： 在一般模式中可以进行删除、复制、贴上等等的动作，但是却无法编辑文件内容！ 按下『i, I, o, O, a, A, r, R』等任何一个字母后会进入编辑模式。按下这些按键时，在画面左下方会出现『 INSERT 或 REPLACE 』的字样，此时才可以进行编辑。而如果要回到一般模式时， 则必须要按下『Esc』这个按键即可退出编辑模式。
指令列命令模式： 在一般模式当中，输入『 : / ? 』三个中的任何一个按钮，就可以将光标移动到最底下那一行。在这个模式当中， 可以提供你『搜寻』动作，读取、存盘、大量取代字符、离开 vi 、显示行号等等的动作也是在此模式中达成！
可视（visual）模式，用于选定文本块；可以在正常模式下输入“v”（小写）来按字符选定，输入“V”（大写）来按行选定，或输入“Ctrl-V”来按方块选定。
选择（select）模式，与 Windows 的编辑器较为接近的选择文本块的方式；很少使用。

四、vim 操作基础

1、输入模式命令

#vim + file1	将光标置于最后一行

#vim +n file1	将光标置于第 n 一行
#vim + /patten file1	将光标置于第一个 patten 一行
#vim -r file1	在上次用 vi 编辑时发生崩溃，恢复 file1
#vim -R file1	以只读方式编辑 file1,:write!强制保存
#view file1	浏览 file1 的内容而不进行编辑,:write!强制保存

#vim -M file 强制性避免对文件进行修改

试着修改文件时会看到这样的错误信息：

E21: Cannot make changes, modifiable is off

下面的命令还是可以去掉这层保护：

:set modifiable

:set write

2、进入 Insert mode 或 Replace mode (按 Esc 返回一般模式)

I,i	I 从第一个非空格字符处插入
A,a	A 从所在行的最后一个字符插入
O,o	O 从光标上一行插入一新行
R,r	R 会一直取代光标所在行直到按 ESC(nR/nr)

3、保存与退出命令说明

:q/: q!	退/强退	:w/:w! file	存/强存
:wq/:wq!	退存/强存强退	:n1,n2 w file	保存部分
:r [file]	合并	ZZ	保存退出
:!cmd	执行 cli	:r !cmd	读取 cmd 结果
:.write otherfile 只把当前行写入指定文件			

4、删除

d(	删除到句首	d4)	删至第四个句尾处
d}	删除到段尾	d{	删除到段首
db/dB	删至单词的开始	d5w	删除 5 个字符
dG	删光标到最后行	d1G	删光标到第一行
d0	删至行首	d^	删至第 1 个非空

5dd 或:5d,	删当 5 行	dM	屏幕中间行
10cj	向下删除 10 行	dL	屏幕底行
J	删换行符合并行	dH	屏幕首行

命令的快捷方式：有些操作符+位移命令使用率高以至于以单独的字符作为其快捷方式：

x 代表 dl(删 1 字符，3x 删 3 字符)　　X 代表 dh(删除当前光标左边的字符)

D 代表 d$(删至行尾)　　C 代表 c$(修改到行尾的内容)

s 代表 cl(修改一个字符)　　S 代表 cc(修改一整行)

5、复制、粘贴、撤消与重做

yy,nyy	复制行	y0	复制到行首
yw	拷贝一个单词	y$	拷贝当前光标到行尾的字符
y1G	复制到第一行	yG	复制到行尾
[ctrl]+r	撤消的撤消	.	重复前一个动作
P,p(paste)	P 贴在上一行	"U"	撤消对一行的全部操作
"u"撤消上次删除的内容，下个 u 将恢复倒数第二次被删字符。			

6、求助系统

- 使用了那么多软件，只有 vim 和 Emacs 的帮助系统给我方便快捷的感觉，大部分软件的帮助往往是摆设而已，而 vim 的帮助的确是考虑到了自己"help"的身份，利用它能很方便地找到想要的东西。
- vim 的帮助是超链接形式的，它使用的就是 tags，所以可以跟 ctags 功能一样按 Ctrl-] 跳转到链接所指处，按 Ctrl-o 返回。

```
:help           打开帮助首页,这个首页分类非常清楚
:help cmd       查找 normal mode 命令,比如 :help dd
:help i_cmd     查找 insert mode 命令,比如 :help i_Ctrl-y    这些信息都在 :help 打开的帮助
:help :cmd      查找 command-line mode 命令,比如 :help :s
:help 'option   查找选项,比如 :help 'tabstop
```

:help	:help -t	:help deleting
:help x	:help index	

五、进退之间

1、挂起与恢复

CTRL-Z 或:suspend　挂起 VIM 程序，回到启动 Vim 的 shell 中去
{执行任何 shell 命令}
fg　回 vim：别忘了等会再让回到 Vim，否则你会丢失所有的改动。

2、"!"用于在 vim 中运行程序的地方

执行 shell 命令，它会提示你按回车键继续，让你有机会看一下程序的输出是什么

:!ls 查看当前目录的内容，用于 Unix :!dir 查看当前目录的内容，用于 MS-Windows
:!{program} 执行{program}
:r !{program} 执行{program}并读取它的输出
:w !{program} 执行{program}并把当前缓冲区内容作为它的输入
:[range]!{program} 以{program}过滤指定的行
你可以这样打开一个新的 shell: :shell :!xterm&

六、灾难恢复

O 以只读方式打开:可能你知道别人正在编辑该文件，你不过想看一下它的内容而已。
E 还是要编辑:小心！如果该文件正被另一 Vim 编辑，你很可能会得到两个版本。
R 从交换文件中恢复:如果你确信交换文件中的内容正是你要找回的东西就那用这个。
Q 退出：如果有另一个 Vim 会话正在编辑最好是选择退让。
A 丢弃：退出同时会撤消对后续命令的执行，这在载入一个脚本编辑多个文件时比较有用,如打开一个多窗口的编辑会话时。
D 删除交换文件：有确信你已不再需要这个交换文件时才应做此选择。
Vim 并不总是能正确地检测到交换文件的存在。比如另一编辑该文件的会话将交换文件放入了另一目录或者不同机器对被编辑文件的路径的理解不同。
如果你实在是不想看到这样的警告信息，你可以在 shortmess 选项中加入"A"标志 1。

UNIT20 VIM cursor moving

1、以 Word 为单位的光标移动

w 往右移一个 word	3w 向右移动 3 个 word
b 往左一个 word	
"e"命令会将光标移动到下一个 word 的最后一个字符. "ge"命令会将光标移动到前一个 word 的最后一个字符上。	

2、将光标移到行首或行尾

"$"命令将光标移动到当前行行尾。<End>键的作用也一样。 "1$"会将光标移动到当前行行尾，"2$"则会移动到下一行的行尾，如此类推。
"^"命令将光标移动到当前行的第一个非空白字符上. "0"命令则总是把光标移动到当前行的第一个字符上。

3、将光标移动到指定的字符上(这 4 个命令不会使光标跑到其它行上)

"f"意为"find"：命令"fx"在当前行上查找下一个字符 x。 "F"命令向左方向搜索。
"t"意为"To"：命令"tx"是把光标停留在被搜索字符前的一字符上。 该命令的反方向版是"Tx":
这 4 个命令都可以用";"来重复。","也是重复同样的命令，但是方向与原命令向相反。

4、将光标移动到匹配的括号上

“%”跳转到与当前光标下的括号相匹配的那一个括号上去。 这对方括号[]和花括号 fg 同样适用。
如果当前光标在"("上，它就向前跳转到与它匹配")"上 如果当前在")"上，它就向后自动跳转到匹配的"("上去

5、将光标移动到指定的行上

"G"命令会把光标定位到文件尾。 "33G"把光标置于第 33 行上。
"50%"会把光标定位在文件中间. "90%"跳到接近文件尾。

"H"意为 Home，把光标定位在屏幕上的顶部 "M"为 Middle，把光标定位在屏幕上的中间 "L"为 Last，把光标定位在屏幕上的尾部

6、移动光标

3<space>	左移 3 行	$	移到当前行的行尾
:3	到第 3 行	0	移到当前行的行首
3z<enter>	将第 3 行滚至屏幕顶部	^	行首，非空格
+	移到下一行的行首	n+	上移 n 行
-	移到上一行的行首	n-	下移 n 行
gg	文件头	k，j，h，l	上、 下、左、右
Enter	换行	kkk/3k	上移 3 行
		%	{}()[]匹配

7、滚屏

CTRL-u 命令会向上翻半页 CTRL-d 命令会向下翻半页
CTRL-f 是向前滚动一整屏 CTRL-b 是向后滚动一整屏
CTRL-e 向上滚动一行，CTRL-E 意为 Extra CTRL-y 向下滚动一行(MS-Windows 兼容的映射键，CTRL-Y 可能被映射为重做)
"zz"命令会把当前行置为屏幕正中央： "zt"命令会把当前行置于屏幕顶端 "zb"则把当前行置于屏幕底端。

8、简单的搜索

/str1 或:/str1/	从光标开始处向文件尾搜,正搜
/str 或:?str?	从光标开始处向文件首搜索,反搜
n	在同一方向重复上一次搜索命令,正查
N	在反方向上重复上一次搜索命令,反查
*：读取光标处的字符串，并且移动光标到它再次出现的地方。	
#：和上面的类似，但是是往反方向寻找。	

简单的模式搜索

^ 字符匹配一行的开头。象"^root"就只匹配出现在一行开头的 root.

$字符匹配一行的末尾。象"was$"只匹配位于一行末尾的单词 was.

"/^the$"只会匹配到一行的内容仅包含"the"的情况。

9、使用标记

当你用"G"等命令从一个地方跳转到另一个地方时，Vim 会记得起跳的位置

``命令可以在两点之间来回跳转。

CTRL-O 命令是跳转到你更早些时间停置光标的位置(提示：O 意为 older).

CTRL-I 则是跳回到后来停置光标的更新的位置(提示：I 在键盘上位于 O 前面)。

备注: 使用 CTRL-I 与按下<Tab>键一样。

":jumps"命令会列出关于你曾经跳转过的位置的列表。最后一个跳转的位置以">"标记。

定义自己的标记，从 a 到 z 一共可以使用 26 个自定义的标记

命令"m{mark}"将当前光标下的位置名之为标记"{mark}"。

' mark(单引号)会使你跳转到 mark 所在行的行首。

`mark 会精准地把你带到你定义 mark 时所在的行和列。

可以使用下面这个命令来查看关于标记的列表：

:marks

在这个列表里你会看到一些特别的标记。象下面这些：

''进行此次跳转之前的起跳点

'" 上次编辑该文件时光标最后停留的位置

'[最后一次修改的起始位置

']最后一次修改的结束位置

UNIT21 VIM Substitute

在 vim 编辑器下使用正则：
1,$s/^\([a-z] [a-z]*\).*/\1/　　显示第一个单词
1,$s/^\([a-z] [a-z]*\)\([^a-z].*[^a-z]\)\([a-z]*[a-z]\)$/\3\2\1/
第一个单词与最后一个单词互换
1,$s/[0-9]//g　　把数字替换为空　g 全局
1,$s#:/[a-z] [a-z]*/[a-z] [a-z]*$##　　删掉 passwd 后面的字段
1,$s#.*:##　　只留路径

一、:[address]s/from/to/[flags]

1、基本范例

:s/^\<the\>/these/　用 these 替换当前行中第 1 个 the
:s/part1 /part2/g　用 part2 替换当前行中所有的 part1
:%s/part1/part2　用 part2 替换所有行中每行第 1 个 part1
:%s/part1/part2/g　用 part2 替换所有行中所有的 part1
:50s/part1 /part2　用 part2 替换第 50 行中的第 1 个 part1
:2,$s/part1/part2/g　用 part2 替换第 2 行到末行中所有的 part1
:.,+3s/part1/part2　用 part2 替换当前行以及当前行后面的三行中每行第 1 个 part1
:.,-3s/part1/part2/g 用 part2 替换当前行以及当前行前面的三行中所有的 part1
:.,$s/yes/no/
/str//w filea　删除后保存
:/str1/str2/w filea　替换后保存
:s+one/two+one or two+要替换的字串中包含/用+作为分隔符
/str/s/str1/str2　在首将出现 str 的行中出现的第一个 str1 替换为 str2
g/str/s/str1/str2　在所有行中将含 str 的行中出现的第一个 str1 替换为 str2
g/str/s/str1/str2/g 在所有行中将含 str 的行中出现的所有 str1 替换为 str2
:g/字串 1/s//字串 2/　用“字串 2”替换“字串 1”,每行只替换第一个

2、以下是采用替换操作进行常见注释格式操作的命令样式

:s/^/#　#用”#”注释当前行

:2,50s/^ /#	#在 2~50 行首添加”#”注释
:.,+3s/^/#	#用”#”注释当前行和当前行后面的三行
:%s/^/#	#用”#”注释所有行

3、替换一个 word

如果你是在写程序，你可能只想替换那些出现在注释中的"four"，代码中的留下。这可有点为难，"c"标志可以让每个目标被替换之前询问你的意见：

:%s/\<four\>/4/gc

replace with Teacher (y/n/a/q/l/^E/^Y)?

此时，你可以有几种答案:

y 好吧，替换吧

n 不，这个先留着

a 别问了，全部换掉吧(这群教授都不够格?? :-))

q 退出，剩下的也不要管了

l 把现在这个改完就退出吧

二、区块的复制、移动、替换、删除（与 sed 相同）

:n1 n2 m n3	move
:n1,n2 co $	copy
:g/字串/p	显示所有带有“字串”的行
:g!/字串/p	显示不带有字串的行
:n1,n2 g/字串/p	显示从 n1 到 n2 中，所有带有“字串”的行
:g/字串/d	删除所有带有“字串”的行
:g/字串/!d	删除所有不带有“字串”的行

：v/.xxxx./d 只保留含 xxxx 的行，其余都删掉

UNIT22 VIM Multi-window operation

在不同窗口中分别编辑不同的文件或同一文件的不同部分。窗口操作极大地方便了多文件操作，提高了文本处理的效率。

一、打开、创建、关闭多个窗口

1、窗口操作的快捷方式

Ctrl+W c 关闭分屏	Ctrl+W q 关闭分屏
Ctrl+W v 左右分割	Ctrl+W s 上下分割
ctrl+w+n 水平拆分编辑一空文件	ctrl+w+w 实现多个窗口这间的切换

2、窗口水平拆分

:split 或:sp 或或:new 或:split kk2.c

3、窗口垂直拆分(:vertical 可在任何分隔命令前)

:vsplit 或:vsplit two.c 或:vsp 或:vertical new
:diffs 分割视窗比较档案
vim -O one.txt two.txt three.txt"-O"可使打开的窗口都垂直排列。若已进入了 vim ":all"命令会为命令行上指定的所有文件各开一个窗口。 ":vertical all"则让打开的窗口都是垂直分隔。

4 关闭窗口

:q 关闭指定的窗口	:qall/:qall!关闭所有窗口
:only 关闭除当前窗口之外的窗口	:wall 针对所有窗口操作的命令
:wqall 针对所有窗口操作的命令	

5、切换窗口(用光标键来也同样可以)

CTRL-W h 到左边的窗口	CTRL-W j 到下面的窗口
CTRL-W k 到上面的窗口	CTRL-W l 到右边的窗口
CTRL-W t 到顶部窗口	CTRL-W b 到底部窗口

6、调整窗口大小(可以把鼠标移到窗口分隔上拖动它)

Ctrl+W <或是> 改变尺寸宽度	CTRL-W_让窗口达到它可能的最大高度
6CTRL-W +一次将窗口的高度增 6 行	6CTRL-W -一次将窗口的高度减 6 行
6CTRL-W _将窗口高度指定为 6 行	:3split alpha.c 打开高为 3 行的新窗

二、编辑多个文件

1、切换到另一文件

vim one.c two.c three.c Vim 将在启动后只显示第一个文件。
:args 查看文件列表,当前正编辑的那一个文件以方括号括起来。
:wnext 开始下一个文件的编辑,这个命令完成以下两个单独命令的工作: :write :next
:wprevious 要回到前一个文件
:last 要移到最后一个文件　　:first 到第一个 没有":wlast"或者":wfirst"这样的命令。
:2next
CTRL-^在两个文件间快速切换

2、自动存盘

当你在不同文件之间转移时，如果你确定自己每次都是要保存文件，就可以告诉 Vim 每当需要时就自动保存文件，不必过问：

:set autowrite 自动保存

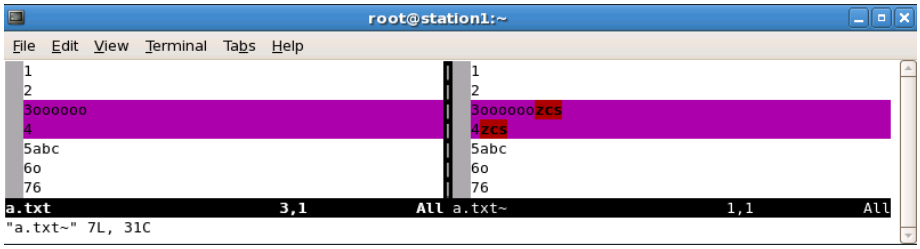
:set noautowrite 不自动保存

3、重新定义一个文件列表

:args five.c six.c seven.h

:args *.txt 或:args! *.txt

4、使用 vimdiff 查看不同



vimdiff main.c~ main.cVim 将会打开左右两个垂直分隔的窗口。

vim main.c

:vertical diffsplit main.c

UNIT23 VIM Visual mode Block operations

vim 列块操作就是在可是模式下操作选择的行中的某一些列的操作，比如常见的我们需要在很多行都需要做一些重复的工作，比如在行头和行尾加相应的标签，或者需要删除行中间某一些特殊的列，那么块操作就是很好的帮手了。

#vim→^v→I→#→ESC
#vim→set nu→^v→(100G:当前行到第 100 行)→I→#→ESC

一、进入和退出可视模式 visual-start

1、三种可视模式（:help blockwise-visual 关于列块操作的信息）

v *characterwise-visual*——v 进入普通可视模式，以字符为单位选择。
V *linewise-visual*——V 进入行可视模式，以行为单位选择。
CTRL-V *blockwise-visual*——CTRL-V (在 Windows 中为 CTRL-q)进入列块可视模式。

2、当键入 "v"、"CTRL-V" 和 "V" 时的模式转换:*

原有模式	″v″	″CTRL-V″	″V″
普通	可视	列块可视	行可视
可视	普通	列块可视	行可视
列块可视	可视	普通	行可视
行可视	可视	列块可视	普通

3、使用可视模式 visual-use 包含三个部骤

a.用 "v"、"V" 或 CTRL-V 标记文本的开始 当前光标下的字符将被作为标记的起始点。
b.把光标移到要标记的文本末尾 起始点和光标间的文本（含当前光标下的字符）将被高亮显示。
c.键入操作符命令 操作符命令将被应用到高亮显示的字符。

二、改变可视区域 visual-change

v_0 0 跳到高亮文本的另一端。同 ″o″ 命令相似，但是在列块模式下，光标移动到水平方向的另一个角。如果这个角上的字符占据了多个屏幕位置（例如一个<Tab>），那

么高亮的文本区域会被改变。
*v_$* 如果 ″$″ 命令和列块模式同时使用，那么高亮文本区域的右边界将取决于高亮区域中最长的行。如果遇到一个不是直上直下的移动命令，那么这个规则将终止。

三、操作可视区域 visual-operators

1、对可视区域可以使用的操作符包括:

d 删除所选中的部分。 D 没有把一行都选中，也把一行删除。	~ 切换大小写 |v_~| ! 通过外部命令过滤 |v_!|
Y yank 抽出 |v_Y| p put 放置 |v_p|	= 通过 'equalprg' 选项的命令过滤 |v_=| gq 按 'textwidth' 指定的宽度排版行|v_gq|
> 右移 1Tab |v_>| < 左移 1Tab |v_<|	x delete X delete J 连接 |v_J|
r 替换 |v_r| R 替换 |v_R|	s 修改 |v_s| S 修改 |v_S|
U 变成大写 u 变成小写 ~ 大小反转	^] 查找标签 |v_CTRL-]| : 为高亮的行启动 ex 命令 |v_:|
I(大写)+字符串+ESC block insert 会在每一行所选中内容的开始前添加”字符串” A(大写)+字符串+ESC block append 会在选中的块后面添加文本。 在选中块紧接每行末尾添加内容，则需用$来指定选择到行尾，而不是简单地用光标。	
c+字符串+ESC 删除并进入输入模式。只需要在选中的第一行输入一个字符串，当按下 esc 后所有行的选中部分都会和第一行一样。 C(大写) +字符串+ESC 会直接删除到行尾，其它与 c 一样。	

2、可用的标记方法包括(h、j、k、l 进行块选择合作):

o 移动光标至标记区域的另一边 O 移动光标至标记区域的另一端点	按:进入 ex，并给出选择的范围'<,'> 正则中选择中区域用\%V 表示
aw 一个单词（包括空格） |v_aw| aW 一个字串（包括空格） |v_aW|	ab 一个 () 块（包括小括号）|v_ab| ib 内含 () 块 |v_ib|
iw 内含单词 |v_iw|	aB 一个{} 块（包括大括号） |v_aB|

iW 内含字串 |v_iW|	iB 内含{} 块 |v_iB|
as a sentencewith white space is inner sentence 句子 |v_is|	a< 一个 <> 块 (包括<>) |v_a<| i< 内含 <> 块 |v_i<|
ap a paragraph |v_ap| ip inner paragraph |v_ip|	a[一个 [] 块 (包括 []) |v_a[| i[内含 [] 块 |v_i[|

四、列块操作 blockwise-operators

列块插入*v_b_I*

对一个列块可视模式下的选择区，I{string}<ESC> 命令将会从选择区开始的位置在每一行插入字符串{string}。

列块添加*v_b_A*

对一个列块可视模式下的选择区，A{string}<ESC> 命令将会从选择区结束处开始在每一行插入字符串{string}。当行的长短不同的时候，将导致选择区的右边界不是直线，这时候列块添加操作的行为就会有所不同:

列块是用<C-v>$产生的:这种情况下字符串被附加到每一行的结尾。

列块是用<C-v>{move-around}产生的:这种情况下字符串将被附加到每一行列块结尾。

列块修改 (c) *v_b_c*

选择区的所有文本将被相同的字符串代替。当使用 "c" 命令的时候选择区的文本将被删除，然后进入插入模式。这时候你可以键入不带回车的文本。当你按<Esc> 的时候所键入的文本将被插入选择区的每一行。

列块修改 (C) *v_b_C*

同 "c" 命令，但是选择区将扩展到每一行的结尾。

*v_b_<*或*v_b_>*列块平移

列块按照 'shiftwidth' 指定的单位移动。

列块替换*v_b_r*

高亮区域的每个字符被同一个字符代替。

UNIT24 VIM TIPS

一、更改文件名

1、复制文件

:edit copy.c

:saveas move.c

2、改变当前正在编辑的文件名，但不想保存该文件：

:edit copy.c

:file move.c

二、查找文件

1、文件浏览器

:edit .显示出来的窗口中将是当前目录下的内容。
<Enter> 在当前窗口中打开文件
o 打开一个水平分隔的窗口显示文件
v 打开一个垂直分隔的窗口显示文件
p 使用 jpreview-windowj 窗口
p 在 preview-window 窗口中编辑
t 在一个新标签页中打开文件
要回到刚才的文件浏览器再次用":edit ."即可。
<F1>会打开 netrw 的帮助文件

2、当前目录

所有的窗口都共享同一个工作目录。一旦在其中一个窗口中用":cd"改变了工作目录，其它窗口中的工作目录也将随之改变。
对一个窗口使用":lcd"后它的工作目录会被记录下来，这样其它窗口中的":cd"或":lcd"命令就不会再影响到该目录了。
在一个窗口中使用":cd"命令会重设它的工作目录为共享的工作目录。

:split

:lcd /etc

:pwd 查看 local directory.

/etc

CTRL-W w

:pwd

/home/zcs

3、用 path 选项中逗号分隔的目录名列表或当前目录来搜索该文件

:find *.txt

#include "inits.h"只需将光标置于"inits.h"文件上然后键入：

gf Vim 就会找到并编辑该文件。

path 选项的定义

:set path+=c:/prog/include 或:set path+=./proto

4、缓冲区列表

Active 出现在窗口中，内容被载入

Hidden 不显示在窗口中，但内容被载入

Inactive 不出现在窗口中，内容也未被载入

隐藏缓冲区：缓冲区中确有内容但你看不到它

:edit one.txt 改动

:hide edit two.txt 使 edit 工作于 hidden 选项被设置的状态。

缓冲区"one.txt"从屏幕上消失，但 Vim 保存了它的当前状态。

:buffers 或 :ls 显示缓冲区列表，结果形如：

1 #h "help.txt" line 62

2 %a+ "usr_21.txt" line 1

3 "usr_toc.txt" line 1

第一列是缓冲区编号。你可以在编辑该文件时以此代替文件名

u 未被列出的缓冲区|unlisted-buffer|

% 当前缓冲区

上一次的活动缓冲区 l 被载入并显示在某窗口中的缓冲区 h 被载入但隐藏的缓冲区。 = 只读的缓冲区 - 不可编辑的缓冲区，其中 modifiable 选项被关闭 + 有改动的缓冲区
编辑一个缓冲区：当然这一命令也可以使用文件名 :buffer 2　你可以以缓冲区编号指定要编辑的缓冲区 :buffer help 会根据键入的部分文件名选一最为相近的缓冲区。本例中是"help.txt"。 :sbuffer 3　要在一个新窗口中打开一个缓冲区使用命令
使用缓冲区列表 :bnext 跳转到下一个缓冲区 :bprevious 跳转到前一个缓冲区 :bfirst 跳转到第一个缓冲区 :blast 跳转到最后一个缓冲区 :bdelete 3 打入"unlisted"列表中，但":buffers!"还是会让它再度现身 :bwipe　要彻底清除一个缓冲区

三、vim 中大小写转化：~或 u 或 gU

1、整篇文章大写转化为小写

ggguG 无须进入命令行模式	gg=光标到文件第一个字符
gu=把选定范围全部小写	G=到文件结束

2、整篇文章小写转化为大写

gggUG 无须进入命令行模式	gg=光标到文件第一个字符
gU=把选定范围全部大写	G=到文件结束

3、只转化某个单词

guw 、gue	gu5w、gu5e 想转换 5 个单词
gUw、gUe	gU5w、gU5e 想转换 5 个单词

4、转换几行的大小写

10gU ：进从光标所在行和往下 10 行都进行小写到大写的转换

gU0 ：从光标所在位置到行首，都变为大写
gU$ ：从光标所在位置到行尾，都变为大写
gUG ：从光标所在位置到文章最后一个字符，都变为大写
gU1G ：从光标所在位置到文章第一个字符，都变为大写
gUU ：将当前行的字母改成大写
3gUU ：将从光标开始到下面 3 行字母改成大写
guu ：将当前行的字母全改成小写

5、将光标下的字母改变大小写：~

3~ 将光标位置开始的 3 个字母改变其大小写

g~~ 改变当前行字母的大小写

三、加速冒号命令

1、命令行补齐这个特性是很多人从 Vi 转到 Vim 的原因

:edit ~in<Tab> 上下文
:set isk<Tab> 这次补全出来的是： :set iskeyword 现在输入"="再按制表符键<Tab>: :set iskeyword=@,48-57,_,192-255
:set is 按 CTRL-D 列出所有匹配有众多的补全候选项时 incsearch isfname isident iskeyword isprint

2、命令行历史记录

:按 10 次<Up>找到 set 命令 :se<Up>更快的办法
:history 列出冒号的历史记录
:history /要查看搜索命令的历史记录

3、记住编辑信息：:set viminfo='1000

每次退出 Vim 时它都会创建一个特殊的标记。最后的一个是'0。上次的'0 现在会变成'1，原来的'1 成了'2，如此类推。

'0 你就会准确地回到你上次退出时的位置，继续干吧。

四、在一个 shell 脚本中使用 Vim

1、Ex 模式的命令的脚本

```
vim change.vim
%s/-person-/Jones/g
quit
```

```
vim -e -s $file < change.vim
```

"-s"的意思是将"script"中的脚本作为 Normal 模式的命令执行。与"-e"连用时它意思是 silent(安静)，并不会把下一个参数作为要执行的脚本文件

3、Normal 模式的脚本

如果你真的需要在脚本中执行 Normal 模式的命令，可以这样用：

```
vim -s script file.txt ...
```

2、从标准输入读取

ls | vim -	将"ls"的输出作为编辑的内容
producer | vim -S change.vim -	还可以用"-S"参数来读取脚本

五、vim 加密解密

加密		解密	
vim t.c	//打开一文件	vim t.c	//打开文件
: X	//据提示操作//加密	:set key=	//解密
:wq	//保存退出	:wq	//保存退出

UNIT25 VIM Customize

1、vimrc 文件

也许你早已厌倦于手工键入那些常用的命令。要使你喜好的选项和映射一次性准备就绪，你可以把它们统统写进一个叫 vimrc 的文件。Vim 在启动时会读取该文件。
:version"命令也会列出 Vim 是在哪些目录寻找该文件的。
对 Unix 和 Macintosh 系统而言通常是文件--这也是推荐的文件 ~/.vimrc 对 MS-DOS 和 MS-Windows 来说可以从下面的文件中选用一个： $HOME/_vimrc $VIM/_vimrc

2、vimrc 示例

set nocompatible 让 Vim 工作于 not-compatible 增强模式下
set backspace=indent,eol,start 这条命令告诉 Vim 在 Insert 模式下退格键何时可以删除光标之前的字符。选项中以逗号分隔的三项内容分别指定了 Vim 可以删除位于行首的空格，断行，以及开始进入 Insert 模式之前的位置。
set autoindent 这个命令让 Vim 在开始一个新行时对该行施以上一行的缩进方式。这样，你在 Insert 模式下按回车或在 ormal 模式下按 o 来添加新行时该行将会与上一行有相同的缩进。
set showcmd 在 Vim 窗口的右下角显示一个完整的命令已经完成的部分。比如说你键入"2f", Vim 就会在你键入下一个要查找的字符之前显示已经键入的"2f"。一旦你接下来再键入一个字符比如"w"，那么一个完整的命令"2fw"就会被 Vim 执行，同时刚才显示的"2f"也将消失。
set history=50 设置冒号命令和搜索命令的命令历史列表的长度。 :set nowrapscan 不循环搜索 :set showmode 现在是什么工作模式? :set magic 关闭遇到错误时声音提示 :set showmatch 显示匹配的行号

:set showcmd 在状态栏显示命令

:set mouse=a 控制台启用鼠标

:set encoding=utf-8 设置 VIM 内部的编码

:set termencoding=utf-8 设置终端编码，就是 VIM 显示的编码

:set fileencoding=zh_CN.utf-8 设置文件编码

3、选项窗口

如果你要查找一个选项，可以使用这个命令：

:options

该命令会打开一个新窗口，在该窗口的最开头的注释下面是一个选项列表，每行一个，对每个选项有一个对应的简短说明。这些选项根据主题分组。把光标移动到你想了解的主题词上按下回车键就可以跳转到对该主题的详细解释。再按下回车键或 CTRL-O 就又会回到该选项列表。

万一你把一个选项值改到自己难以收拾残局，还可以在该选项的后面放一个&符号使它恢复其默认值，如：

:set iskeyword&

4、忽略大小写

默认情况下 Vim 的搜索是大小写敏感的。

现在打开 ignorecase 选项：

:set ignorecase

下面的命令又可以暂时关闭这一选项：

:set noignorecase

现在打开 smartcase 选项：

:set ignorecase smartcase

如果你要搜索的内容中至少包括一个大写字母，整个搜索就会是大小写敏感的。这样设计你就不必总是输入大写字符了，你想要进行大小写敏感的搜索时准确键入就行了。这看起来智能多了。

设了上面这两个选项，下面的所有 word 都可以搜索得到：

word word, Word, WORD, WoRd, etc.

Word Word

WORD WORD

WoRd WoRd

5、备份

:set backupext=.bak 生成.bak 的备份文件。

:set backupdir 它指定备份文件将被置于哪个目录下

:set backup　生成的备份文件名将是原文件名后面附加一个~。

备注：如果 backup 选项是关闭的但 writebackup 选项是打开的，Vim 还会生成一个备份文件。但是，一旦该文件被成功地保存它就会被自动删除。

:set patchmode=.orig 如果你第一次开始编辑 data.txt，改一些东西后存盘,Vim 会保留一份该文件的原始版在 data.txt.orig 中。

如果你把 patchmode 选项设置为空(默认情况正是如此)，文件的原始副本就不会被额外保存。

6、有色或无色（:syntax enable）

:syntax clear 暂时关闭语法高亮，新开窗口时又会应用彩色显示。
:syntax off 将彻底禁用语法高亮功能，并立即对各个缓冲区生效。
:set syntax=ON 当前缓冲区打开语法高亮功能。

7、告诉你当前位置

使用 CTRL-G 命令。会得到一些类似于下面的信息行(假设 ruler 选项已关闭) "pad.sed" line 233 of 650 --35%-- col 45-52
:set number　显示列号 :set nonumber 不显列号
:set ruler　显示列号 :set norouler 不显行号

8、常用选项

不要折行 Vim 通常会把超出当前显示窗口显示宽度的行折到下一行显示，这样你还是可以看到整行的内容。有时候让它不管多长都放到窗口最右边去会更好。这时你要看这些超出当前视野的部分就要左右滚动该行了。控制长行是否折到下一行显示的命令是： :set nowrap
Vim 会自动保证你把光标移动到某字符上时它会显示给你看，必要时它自动左右滚动。要查看左右 10 个字符的上下文，用命令：

:set sidescroll=10
跨行移动命令 Vim 中多数移动光标的命令会在遇到行首或行尾时停止不动。whichwrap 选项可以用来控制这些移动光标的命令此时的行为 规则。下面的设置是它的默认值 :set whichwrap=b,s
这样光标位于行首时按退格键会往回移动到上一行的行尾。同时在行尾按空格键也会移动到下一行的行首。 要让左右箭头键 3 在遇到行的边界时也能智能地上上下下，使用命令： :set whichwrap=b,s,<,>
这些都是只针对于 Normal 模式。要让左右箭头键在 Insert 模式下也能如此： :set whichwrap=b,s,<,>,[,]
查看制表符 文件中含有制表符时，你并不能看到它们。要让这些制表符成为可见的字符： :set list
现在每个制表符都会以^I 显示。同时每行行尾会有一个$字符，以便你能一眼看出那些位于一行尾部的多余空格。 这样做的缺点是文件中制表符很多时整个屏幕看起来就很抱歉了。如果你的终端支持彩色显示，或者使用的是 GUI, Vim 就可以把制表符和空白字符高亮起来显示。这要配合使用下面的 listchars 选项： :set listchars=tab:>-,trail:- 现在每个制表符会以"\>---"显示，同时行尾空格以"-"显示，看起来会好一点，你觉得呢？

UNIT26 SEd Basic Usage

Stream Editor 主要用来自动编辑一个或多个文件；简化对文件的反复操作；编写转换程序等。以下介绍的是 Gnu 版本的 Sed。

定址选定希望编辑的行的范围	
$ sed -n p file	$ sed -n '2,$p' file
$ sed '/music/=' 打印行号	$ sed -n '3,+1p' file
$ sed -n '/.*ing/!p' install.log	把不含‘样式’的数据行删除
$ sed -n /lang/!p file	$ sed -n '5,/^test/p' file
$ sed '/his/,/her/s/$/*******/' file	末尾用*******替换
$ sed 's/erors/errors/' file one errors, two erors.	$ sed 's/erors/errors/g' file one errors, two errors.

一、sed [option] [action]

1、[option]：-nerfi

-n ：使用安静(silent)模式。在一般 sed 的用法中，所有来自 STDIN 的数据一般都会被列出到屏幕上。但如果加上 -n 参数后，则只有经过 sed 特殊处理的那一行(或者动作)才会被列出来。
-e ：直接在指令列模式上进行 sed 的动作编辑；
-f ：直接将 sed 的动作写在一个档案内， -f filename 则可以执行 filename 内的 sed 动作；
-r ：sed 的动作支持的是延伸型正则表达式的语法。(预设是基础正则表达式语法)
-i ：直接修改读取的档案内容，而不是由屏幕输出。

2、[action]：[n1[,n2]] function

①n1, n2 ：不见得会存在，一般代表『选择进行动作的行数』，举例来说，如果我的动作是需要在 10 到 20 行之间进行的，则『 10,20[动作行为] 』

②function

a\ ：在当前行后面加入一行文本。
b lable ：分支到脚本中带有标记的地方，如果分支不存在则分支到脚本的末尾。
c\ ：用新的文本改变本行的文本。
d ：从模板块（Pattern space）位置删除行。
D：删除模板块的第一行。
i\ ：在当前行上面插入文本。
h ：拷贝模板块的内容到内存中的缓冲区。
H ：追加模板块的内容到内存中的缓冲区

g ： 获得内存缓冲区的内容，并替代当前模板块中的文本。
G ： 获得内存缓冲区的内容，并追加到当前模板块文本的后面。

l ： 列表不能打印字符的清单。
n ： 读取下一个输入行，用下一个命令处理新的行而不是用第一个命令。
N ： 追加下一个输入行到模板块后面并在二者间嵌入一个新行，改变当前行号码。
p ： 打印模板块的行。
P ： 打印模板块的第一行。
q ： 退出 Sed。
r file ： 从 file 中读行。
t label ： if 分支，从最后一行开始，条件一旦满足或者 T，t 命令，将导致分支到带有标号的命令处，或者到脚本的末尾。
T label ： 错误分支，从最后一行开始，一旦发生错误或者 T，t 命令，将导致分支到带有标号的命令处，或者到脚本的末尾。
w file ： 写并追加模板块到 file 末尾。
W file ： 写并追加模板块的第一行到 file 末尾。
! ： 表示后面的命令对所有没有被选定的行发生作用。
s/re/string ： 用 string 替换正则表达式 re。
= ： 打印当前行号码。
： 把注释扩展到下一个换行符以前。

{} 在定位行执行的命令组

以下的是替换标记
g：表示行内全面替换。
p：表示打印行。
w：表示把行写入一个文件。
x：表示互换模板块中的文本和缓冲区中的文本。
y：表示把一个字符翻译为另外的字符（但是不用于正则表达式）

三、经典范例

1、列出/etc/passwd 的内容，并且打印行号，同时，请将第 2~5 行删除！，在第二行后面加入两行字，例如『Drink tea or .....』『drink beer?』

```
# nl /etc/passwd | sed '2aDrink tea or ......\
> drink beer ?'
     1  root:x:0:0:root:/root:/bin/bash
     2  bin:x:1:1:bin:/bin:/sbin/nologin
Drink tea or ......
drink beer ?
```

```
     3  daemon:x:2:2:daemon:/sbin:/sbin/nologin
```

```
#每一行之间都必须要以反斜线 \ 来进行新行的增加
#在 a 后面加上的字符串就已将出现在第二行后面啰！那如果是要在第二行前呢？
# nl /etc/passwd | sed '2i drink tea' 就对啦！
```

2、我想将第 2-5 行的内容取代成为『No 2-5 number』呢？

```
# nl /etc/passwd | sed '2,5cNo 2-5 number'
     1  root:x:0:0:root:/root:/bin/bash
No 2-5 number
     6  sync:x:5:0:sync:/sbin:/bin/sync
```

3、仅列出第 5-7 行

```
# nl /etc/passwd | sed -n '5,7p'
```

```
# sed '5,7p' (没加-n5-7 行会重复输出)
```

4、我们可以使用 ifconfig 来列出 IP ，若仅要 eth0 的 IP 时？

```
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:51:FD:52:9A:CA
          inet addr:192.168.1.12  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::250:fcff:fe22:9acb/64 Scope:Link
.....(以下省略).....
```

```
# 其实，我们要的只是那个 inet addr:..那一行而已，所以利用 grep 与 sed 来捉
# ifconfig eth0 | grep 'inet ' | sed 's/^.*addr://g' |  sed 's/Bcast.*$//g'
192.168.0.189
```

```
# 您可以将每个管线 (|) 的过程都分开来执行，就会晓得原因啰！
```

5、将/etc/man.config 中，有 MAN 的设定取出来，但不要说明内容

```
# cat /etc/man.config | grep 'MAN'| sed 's/#.*$//g' | sed '/^$/d'
```

```
# 每一行当中，若有 # 表示该行为批注
```

6、利用 sed 直接在 ~/.bashrc 最后一行加入『# This is a test』

```
# sed -i '$a # This is a test'  ~/.bashrc
```

```
# -i 参数可以让你的 sed 直接去修改后面接的档案内容喔！而不是由屏幕输出。
# 至于那个 $a  则代表最后一行才新增的意思
```

7、sed 脚本文件(以#开头的行为注释行)

行尾不能有空白或文本，一行中有多个命令用分号分隔。

```
$ cat fixup.sed
#n                                  关闭自动打印
s/foo/bar/g
s/chicken/cow/g
$ sed -f fixup.sed myfile.xml > myfile2.xml
```

8、按顺序执行{}括号里的命令

```
Sed '/^root/{ s/^root/blues/;s/bash$/nologin/; }'        passwd
查找以 root 开头的行把 root 和 bash 替换成 blues 和 nologins
```

9、sed –r 使用扩展正则

```
sed -r 's/^([a-z]+)([^a-z].*[^a-z])([a-z]+)$/\3\2\1/' passwd
把 passwd 文件的首末单词互换，并输出到屏幕
sed -r 's/^([a-zA-Z0-9_]+):.*/\1/' passwd        只输出用户名
```

10、行号:=命令

sed -n '$='	计算行数 （模拟 "wc -l"）
# df -h|sed =|sed 'N;s/\n/ /' 1 Filesystem Size Used Avail Use% Mounted on 2 /dev/mapper/vol0-root 3 7.8G 2.6G 4.9G 35% /	
# df -h|sed = | sed 'N;s/\n/\t/' 1 Filesystem Size Used Avail Use% Mounted on 2 /dev/mapper/vol0-root 3 7.8G 2.6G 4.9G 35% /	

11、下一个:n 命令

sed 'n;d'	删除所有偶数行
sed -n '/regexp/{n;p;}'	显示匹配行的下一行
sed 'n;n;n;n;n;n;n;d;'	其他 sed, # 删除 8 的倍数行
sed -n '3,${p;n;n;n;n;n;n;}'	从第 3 行开始，每 7 行显示一次
sed '/test/{ n; s/aa/bb/; }' file	移到匹配行的下一行替换并替换
[address1 ,[address2]]N 最多两个地址 sed -e 'N' -e 's/\n/ /' input.dat The UNIX Operating System 原 input.dat 的内容如下: The UNIX Operating System	添加一笔资料在 pattern space 内 每两行连成一行,类似 paste
sed '/./=' file | sed '/./N; s/\n/ /'	对所有行编号，但只显示非空行的行号

12、退出:q 命令(最多配合一个地址参数)

sed 10q	显示文件中的前 10 行 （模拟“head”）
sed q	显示文件中的第一行 （模拟“head -1”）
sed -e '/zcs/q'	遇到"zcs"就中止 sed

13、标记：b:label 命令

与函数参数 b 可在 sedscript 内建立类似 BASIC 语言中 GOTO 指令的功能。其中，函数参数: 建立标记;函数参数 b 将下一个执行的指令 branch 到标记处执行。函数参数: 与 b ， 在 script file 内配合的情况如下 编辑指令 m1 :记号 编辑指令 m2

[address1,[address2]]b [记号]

其中，当 sed 执行至指令[address1,[address2]]b [记号] 时，如 pattern space 内的数据符合地址参数，则 sed 将下一个执行的位置 branch 至由:记号设定的标记处，也就是再由"编辑指令 m2" ... 执行。另外，如果指令中参数 b 后没有记号，则 sed 将下一个执行的指令 branch 到 script file 的最后，利用此可使 sedscript 内有类似 C 语言中的 case statement 结构。

题目：将 input.dat 文件内数据行的开头字母重复印 40 次。假设 input.dat 檔的内容如下:

A

B

C

说明：用指令 b p1 与:p1 构成执行增加字母的循环(loop)，同时在字母出现 40 个时，也用指令 b 来跳出循环。下面就以文件内第一行数据"A" 为例，描述它如何连续多添加 39 个"A" 在同一行:

用指令 s/A/AA/将"A" 替换成"AA"。

用指令 b p1 与:p1 构成循环(loop)，它目的使上述动作被反复的执行。每执行一次循环，则数据行上的"A" 就多出一个。例如，第一次循环数据行变成"AA"，第二次循环资料行变成"AAA" ...。

用指令[ABC]\{40\}/b 来作为停止循环的条件。当数据行有连续 40 个 A 出现时，参数 b 将执行的指令跳到最后，停止对此行的编辑。

同样，对其他数据行也如同上述的方式执行。

sed

命令列如下:

```
sed
-e '{
:p1
/A/s/A/AA/
/B/s/B/BB/
/C/s/C/CC/
/[ABC]\{40\}/b
b p1
}' input.dat
```

14、标记：t:label 命令

执行 t 的 branch 前，会先去测试其前的替换指令有没有执行替换成功。

在 script file 内的情况如下:

编辑指令 m1

:记号

编辑指令 m2

s/.../.../

[address1,[address2]]t [记号]

编辑指令 m3

其中，与参数 b 不同处在于，执行参数 t branch 时，会先检查其前一个替换指令成功与否。如成功，则执行 branch；不成功，则不 branch，而继续执行下一个编辑指令，

题目：将 input.dat 文件中资料 A1 替换成 C1、C1 替换成 B1、B1 替换成 A1。

input.dat 檔的内容如下:

代号

B1

A1

B1

C1

A1

C1

说明：input.dat 文件中全部数据行只需要执行一次替换动作，但为避免数据被替换多次，所以利用函数参数 t 在 sedscript 内形成一类似 C 语言中 case statement 结构，使每行数据替换一次后能立即用函数参数 t 跳离替换编辑。

sed

命令列:

```
sed
-e '{
s/A1/C1/
t
s/C1/B1/
t
s/B1/A1/
t
}' input.dat
```

UNIT27 SEd pattern space&hold space

一、sed 中的模式空间和保留空间的概念

1、sed 的一般工作模式（没有利用到 Hold space 时）

每次从 input 中取一行数据到 pattern space 中,经一些处理，将一行数据放入 output 中。记住，此时 pattern space 还储存着这一行数据，直到 input 再放入第二行数据取代第一行数据。

2、h/H;g/G;x 涉及到 hold space(x:换 p 空间和 h 空间)

理解了 sed 中的模式空间和保留空间（初始为一个空行）有助于从本质上理解实现逆转、只输出奇数或偶数行等等功能。

可以简单的将 hold space 理解为 sed 的一个缓冲区，只是这个缓冲区不会直接进行输出，并且只有 pattern space 可以对其进行操作，放入或者拿出数据。

3、多将 h/H;g/G;x 联合使用，以达到栈和队列的使用目的

h 拷贝 p-space 的内容到 h-space。
H 追加 p-space 的内容到 h-space
g 获得 h-space 的内容，并替代当前 p-space 中的文本。
G 获得内存-space 的内容，并追加到当前 p-space 文本的后面。

```
[root@station1 ~]# sed  -e '1,5h' -e '$G' file
111111
22222
3333
444
55
6
55

[root@station1 ~]# sed  -e '1,5H' -e '$G' file
111111
22222
3333
444
55
6

111111
22222
3333
444
55
```

4、文本间隔

sed G	在每一行后面增加一空行
sed 'G;G'	在每一行后面增加两行空行
sed '/^$/d;G'	所有空行删除并在每行后增加一空行
sed '/regex/G'	在匹配式样“regex”的行之后插入一空行
sed 's/regxp/&\n/g' filename sed 's/ regxp /\n&/g' filename	行后添加新行 行前添加新行
sed 'n;d'	即删除偶数行
sed 'n;n;n;n;G;' # 其他 sed	在每 5 行后增加一空白行
sed '/regex/{x;p;x;}'	在匹配行前插入一空行
sed '{x;p;x;}'	在每一行行前插入一空行
sed '/regex/G'	在匹配行后插入一空行
sed '/regex/{x;p;x;G;}'	在匹配行前和后各插入一空行
sed -e '1h' -e '3x' input.dat	第 3 行资料就被第 1 资料替代。
sed '{x;p;x;}'	在匹配式样行之前插入一空行
sed -e '/test/h' -e '/check/x'	把包含 test 与 check 的行互换

二、h/H;g/G;x 案例

1、下面有 4 个例子来解释下选项 n 的作用：

sed '' file，这句命令会将 output 中的数据进行输出：

```
[root@station1 ~]# sed '' file
111
22
3
```

sed -n 'p' file 这句命令是要求输出 pattern space 中的数据

```
[root@station1 ~]# sed -n 'p' file
111
22
3
```

sed 'p' file 不仅将每次 output 中的数据进行了输出，接着又将 pattern space 中的数据再要求输出一次

```
[root@station1 ~]# sed 'p' file
111
111
22
22
3
3
```

sed -n '' file 输出结果是空，因为选项-n 指明了要将 pattern space 中的数据进行输出，而缺少了 p 命令，所以不能将 pattern space 中的数据进行输出。

2、G：sed 'G' file

```
[root@station1 ~]# sed 'G' file
111

22

3
```

因为 hold space 的初始为一个空行，并且始终没有修改其中的数据，一直保持的是空行，所以在每次执行 G 命令时，会将空行追加到每行数据之后。

3、x：sed 'x;G' file

```
[root@station1 ~]# sed 'x;G' file

111
111
22
22
3
```

因为开始 x 命令将 hold space 的空行和 pattern space 中的 111 交换，然后 G 命令又将 111 加到 pattern apace 的空行之后，然后输出了 111 ，此时 hold space 中依旧为 111

接着 pattern space 中进入 22，x 命令将 hold space 的 111 和 pattern space 中的 22 交换，然后 G 命令又将 22 追加到 pattern apace 的 111 之后，然后输出了 111
22 依次类推。过程如表格所示：

sed 'x;G' file	hold space	pattern space	输出
x 前	空行	111111111	无输出
x 后	111	空行	
G 前	111	空行	111
G 后	111	空行 111	
x 前	111	22	无输出
x 后	22	111	
G 前	22	111	111 22
G 后	22	111 22	
x 前	22	3	无输出
x 后	3	22	

G 前	3	22	22 3
G 后	3	22 3	

4、h：sed 'h;G' file(可参照 x 命令的方法进行一步步的演示)

```
[root@station1 ~]# sed 'h;G' file
111
111
22
22
3
3
```

5、H：sed 'H;x' file

```
[root@station1 ~]# sed 'h;x' file
111
22
3
```

6、g：sed '1h;g;x' file

```
[root@station1 ~]# sed '1h;g;x' file
111
111
111
```

将 1h=111 复制到 h-space 中去，之后每次执行 g 命令时，都是从 h-space 中将 111 复制出来，覆盖掉了 p-space 中的数据，故结果显示为打印了 3 行 111。

“1“表示只有第一行执行 h 命令 ； $表示只有最后一行执行这个命令
在 1 或$和命令中添加“！ “，表示只有第一行或者最后一行不执行这个命令。

7、patten space 与 hold space

例. 将文件中含"omega" 字符串的数据输出。其命令为 sed -f gp.scr file

gp.scr 的内容如下:

```
/omega/b
N
h
s/.*\n//
/omega/b
g
D
```

在上述 sedscript，因借着参数 b 形成类似 C 语言中的 case statement 结构，使 sed 可分别处理当数据内含"omega"；当"omega"字符串被拆成两行；以及数据内没有"omega"字符串的情况。接下来就依上述三种情况 将 sedscript 分三部份讨论。

当数据内含"omega"，则执行编辑指令

/omega/b

它表示当资料内含"omega"时，sed 不用再对它执行后面的指令，而直接将它输出。

当数据内没有"omega"，则执行编辑指令如下

N

h

N

h

s/.*\n//

/omega/b

其中，参数 N，它表示将下一行资料读入使得 pattern space 内含前后两行数据。参数 h，它表示将 pattern space 内的前后两行资料存入 hold space 。参数 s/.*\n//，它表示将 patternspace 内的前后两行资料合并成一行。/omega/b，它表示如果合并后的数据内含"omega"，则不用再执行它之后的指令，而将此数据自动输出；

当合并后的数据依旧不含"omega"，则执行编辑指令如下

g

D

其中，参数 g，它表示将 hold space 内合并前的两行资料放回 pattern space。参数 D，它表示删除两行资料中的第一行资料，并让剩下的那行数据，重新执行 sedscript。如此，无论的资料行内或行间的字符串才可搜寻完全。

将文件中的前 100 数据，搬到文件中第 300 后输出。其命令列为:sed-f mov.scr

mov.scr 的内容为

1,100{

H

d

}

300G

UNIT28 SEd Command Summary

一、SEd 文本转换和替代(-n 和 p 一起用只打印变动行)

$ sed -n ‘2,4s/B/567/p’	
$ sed -n ‘/AA/s/237/567/’	
$ sed -n ‘/AA/,/DD/s/B/567/p’	
$ sed -n ‘s/La//p	
$ sed ’s/:.*//’ /etc/passwd	删除第一个冒号之后的所有东西
$ sed -n ‘s/^...//p’	删除行首的 3 个字符
$ sed -n ‘/...$//p’	删除行尾的 3 个字符
$ sed ’/baz/s/foo/bar/g’	只在行中出现 baz 的情况下替换
$ sed ’/baz/!s/foo/bar/g’	只在行中未出现 baz 的情况下替换
$ sed ’s/ruby/red/g;s/puce/red/g’	ruby|puce 一律换成 red
$ gsed ’ruby|puce/red/g’	ruby|puce 一律换成 red
$ sed ’s/^[\t]*//’	删除行前的“空白字符”，左对齐
$ sed ’s/[\t]*$//’	删除拖尾的“空白字符”
$ sed ’s/^[\t]*//;s/[\t]*$//’	删除前导和拖尾的空白字符
$ sed -n ’s/^/zcs/p’ file	在行首添加 zcs
$ sed ’s/^/ /’	行首插入 5 个空格（全文右移 5 字符）
$ sed ’s#10#100#g’	紧跟 s 命令的被认为是新的分隔符(代替/)
$ sed -e ″s/$var1/$var2/g″	使用变量替换(使用双引号)
$ sed -n ’s/^root/&user/p’	^root 改为^rootuser
$ sed -n -r ’s/(love) able/\1rs/p’	love 被标记为 1
$ sed ’s/foo/bar/g’	每行中的所有 foo 都换成 bar

二、SEd 选择性地显示特定行

sed -n ’/Iowa/,/Montana/p’	区分大小写方式
sed -n ’/regexp/,$p’	从包含正则的行到最后一行
sed -n ’8,12p’或 sed ’8,12!d’	显第 8 至第 12 行
sed ’$!d’ 或 sed -n ’$p’	模拟“tail -1”
sed -n ’/reg/p’或 sed ’/reg/!d’	模拟 grep
sed -n ’/reg/!p’或 sed ’/reg/d’	模拟 grep -v
sed -n ’/^.\{65\}/p’	显示包含 65 个或以上字符的行
sed -n ’/^.\{65\}/!p’	显示包含 65 个以下字符的行

sed '/^.\{65\}/d'	显示包含 65 个以下字符的行
sed -n '52p'或 sed '52!d'	显示第 52 行
sed '/AAA/!d; /BBB/!d; /CCC/!d'	显示含 AAA、BBB 或 CCC 的行，任意次序
sed '/AAA.*BBB.*CCC/!d'	显示含 AAA、BBB 或 CCC 的行，固定次序
sed '/AAA|BBB|CCC/!d'	显示含 AAA、BBB 或 CCC 的行，类 egrep

三、SEd 选择性地删除特定行

sed '2,$d' file sed '/^d/d' file sed '/[0-9]/\{3\}/d' file	
sed '/test/d' file	删除含 test 的行
sed '/test/'d file	删除含 test 的行
sed 's/^[ ]*//g' file	删除行首空格
sed 's/^ *//g' file	删除行首空格
sed 's/^[[:space:]]*//g' file	删除行首空格
sed '/In/,/Out/d'	除了两个正则表达式之间的内容都显示
sed '/^$/d' 或 sed '/./!d'或 grep '.'	显示非空行
sed '/./,$!d'	删除文件顶部的所有空行
grep -v ^# filename | sed /^[[:space:]]*$/d \| sed /^$/d	删空行和空格及注释行

四、sed 案例

```
$cat data
northwest NW Charles Main      3.0 .98 3 34
western   WE Sharon Gray       5.3 .97 5 23
southwest SW Levis 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
```

```
$ sed ' s/[0-9][0-9]$/&.5/'       data
northwest NW Charles Main      3.0 .98 3 34.5
western   WE Sharon Gray       5.3 .97 5 23.5
```

```
$ sed -n ' s/\(Mar\)got/\lianne/p'  data
north NO Marianne Weber 4.5  .89 5 9
$sed ' s#3#88#g'  data
```

```
northwest NW Charles Main      88.0 .98 88 884
western   WE Sharon Gray       5.88 .97 5  288
```

```
$ sed ' /west/,/east/s/$/**VACA**/'  data
northwest NW Charles Main      3.0 .98 3 34**VACA**
western   WE Sharon Gray       5.3 .97 5 23**VACA**
```

```
$ sed -e ' 1,3d'  -e ' s/Hemenway/Jones/'  data
```

```
$ cat notice
| *** SUAN HAS LEFT THE COMPANY *** |
$ sed ' /Suan/r notice'  data
southern  SO Suan Chin         5.1 .95 4 15
| *** SUAN HAS LEFT THE COMPANY *** |
```

```
$ sed -n ' /north/w newfile'  data
```

```
$ sed ' /^north /a -->THE NORTH SALES DISTRICT HAS MOVED<--'  data
north     NO Margot Weber      4.5 .89 5 9
-->THE NORTH SALES DISTRICT HAS MOVED<--
```

```
$ sed ' /eastern/{n;s/AM/Archie/;}'  data
eastern   EA TB Savage         4.4 .84 5 20
northeast NE Archie Main Jr.   5.1 .94 3 13
```

```
$ sed ' 1,3y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/'  data
```

```
$ sed -e ' /northeast/h'  -e ' $G'  data
northwest NW Charles Main 3.0 .98 3 34
northeast NE AM Main Jr.  5.1 .94 3 13
north     NO Margot Weber 4.5 .89 5 9
northeast NE AM Main Jr.  5.1 .94 3 13
```

UNIT29 awk Program

Awk 是文本工具中最难掌握的，它借鉴了 C 语言、python 和 bash 的一些精华部分。awk 自解释型编程语言，它支持用户自定义函数和动态正则表达式等先进功能。

awk 设计简单，速度表现很好,可结合 Shell(w|awk…),它在命令行中使用，但多是作为脚本来使用。很多基于 shell 的日志分析工具可用它完成。

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。awk 能够用很短的程序对文档里的资料做修改、比较、提取、打印等处理,如果使用 C 或 Pascal 等完成上述的任务会十分不方便而且很花费时间，所写的程序也会很大。

一、awk 模式扫描语言：ed→(sed&grep)→awk→perl

1、像 shell 一样 awk 也有几种(旧 awk、新 nawk、gawk 兼容 nawk)

awk:最初在 1977 年完成，支持幂 ^，getline，system 等;
nawk： （在 20 世纪 80 年代中期，new awk)许多系统中旧的 awk 解释器通常安装为 oawk (old awk) 命令，而 nawk 解释器则安装为 nawk 命令。Dr. Kernighan 仍然在对 nawk 进行维护，与 gawk 一样，它也是开放源代码的，并且可以免费获得;
gawk：是 GNU Project 的 awk 解释器的开放源代码实现。尽管早期的 GAWK 发行版是旧的 AWK 的替代程序，但不断地对其进行了更新，以包含 NAWK 的特性;
目前，大家都比较倾向于使用 awk 和 gawk。

2、虽然 awk 与 sed 指令的结构相同，但 awk 与 sed 有以下不同

awk 非常适合于结构化的文本文件（行、列数据）处理。相对于 sed,它可进行复杂的编程处理,并且可以产生复杂的报表输出。

awk 缺省情况下使用扩展的正则。
awk 中用 C 函数取代了一两个字符的行编辑器命令集，如 print 取代了 p。
awk 倾向于一行当中分成数个字段来处理。

3、为了避免碰到 awk 错误总结出以下规律

指令必须包括单引号，因为$在 shell 中有特殊意义的。
因为模式和动作可有可无，确保用{}括起动作语句以便与模式相区别。
{}内的多个指令可用分号分隔，或直接以‘Enter’按键来隔开

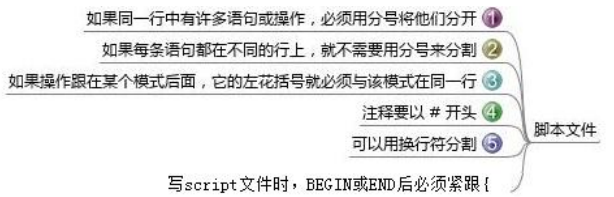
用圆括号括起条件语句。
printf 格式化输出时，务必加上\n,才能进行分行！
字符串一定要保证被双引号括起来(在模式中除外)。
逻辑运算的‘等于’请务必使用两个等号‘==’
在 awk 当中变量可直接使用，不需加上$符号

二、awk 的三种调用方式

1、awk [options] 'script' var=value file(s) ,有以下两个特例

# awk ‘pattern’ filename	# awk ‘{action}’ filename

2、awk [options] -f scriptfile var=value file(s)

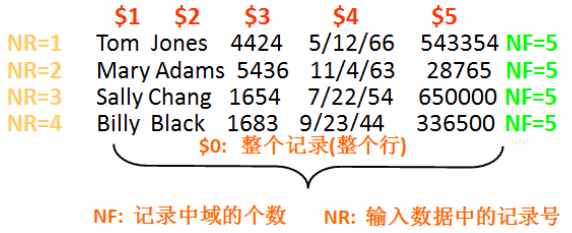



```
#cat  my_awk( 要特别注意操作顺序和流程 )
/Sally/ {print “**** found  Sally!  ****”}
{print  $1, $2, $3}
```

```
#awk  -f  my_awk   employees
```

3、#!/bin/nawk -f … … 执行：myscript.awk filename

三、记录和域字段



1、awk 程序数据处理流程

→读入第一行，并将第一行的资料填入$0,$1,$2. . .等变量当中；
→依据“条件类型”的限制，判断是否需要进行后面的“动作”；

→做完所有的动作与条件类型；
→若还有后续的‘行’的数据，则重复上面的 3 个步骤，直到所有的数据都读完为止。

2、NR 与 NF(缺省的 RS 为\n,缺省的 FS 为 Blank)

列出登陆历史用户中的每一行的账号，并且列出目前处理的行数，并且说明，该行有多少字段（注意，内容想要以 print 或 printf 打印时，非变量的文字部分，都需要使用双引号括住）：

```
[root@linux ~]# last | awk '{print $1 "\t lines: " NR "\t columes: " NF}'
dmtsai   lines: 1       columes: 10
```

四、获得外部变量方法

1、获得普通外部变量

格式：awk ‘{action}’ 变量名=变量值，这样传入变量，可以在 action 中获得值。 注意：变量名与值放到’{action}’后面。
test='awk code' # echo|awk '{print test}' test="$test" awk code
echo | awk 'BEGIN{print test}' test="$test" 在 BEGIN 的 action 不能获得

2.BEGIN 程序块中变量

格式：awk -v 变量名=变量值 [-v 变量 2=值 2 …] 'BEGIN{action}'
test='awk code' # echo | awk -v test="$test" 'BEGIN{print test}' awk code # echo \| awk -v test="$test" '{print test}' awk code

3.获得环境变量

#awk 'BEGIN{for (i in ENVIRON) {print i"="ENVIRON[i];}}'

LANG=en_US.UTF-8

.....

awk 内置变量 ENVIRON,就可以直接获得环境变量。它是一个字典数组。环境变量名 就是它的键值。

UNIT30 awk Operator

awk 作为文本处理优秀工具之一，它有拥有丰富的运算符。awk 运算符、表达式及功能与 c 语言基本相同。下面我们一起归纳总结一下所有运算符。

级别	运算符	说明	级别	运算符	说明
13	-,+	正，负	12	!,~	逻辑非、按位取反或补码
11	*,/,%	乘、除、取模	10	+,-	加，减
9	<<,>>	按位左移、按位右移	8	<=,>=,<,>	小于等于,大于等于，小于，大于
7	==,!=	等于，不等于	6	&	按位与
5	^	按位异或	4	|	按位或
3	&&	逻辑与	2	||	逻辑或
1	=,+=,-=,*=,/=,%=,&=,^=,|=,<<=,>>=				赋值、运算且赋值
较高运算级别的运算符先于低级别的运算符进行求值运算。					

一、字段引用

$ 字段引用 空格 字符串连接符
# gawk -F"[:]" '{ print $1 }' list.txt 同时使用两个分隔符

二、字段引用

x*y	x 乘 y	x+y	x 加 y
x%y	计算 x/y 的余数（求模）	x-y	x 减 y
x/y	x 除 y	x=y	将 y 的值赋给 x
x**y	x 的 y 次幂	x+=y	将 x+y 的值赋给 x
x^y	x 的 y 次幂	x-=y	将 x-y 的值赋给 x
++y	y 加 1 后使用 y(前置加)	x*=y	x*y 的值赋给 x

y++	使用 y 值后加 1（后缀加）	x^=y	将 x^y 的值赋给 x
--y	y 减 1 后使用 y(前置减）	x/=y	将 x/y 的值赋给 x
y--	使用后 y 减 1(后缀减）	x%=y	将 x%y 的值赋给 x
-y	负 y;也称一目减	x**=y	将 x**y 的值赋给 x

二、awk 赋值运算符

awk /^$/{print x = x + 1} example　统计出空行数

awk /^$/{print x += 1} example　使代码更简洁

awk /^$/{print x++} example　使代码更简洁

= += -= *= /= %= ^= **=
a+=5; 等价于：a=a+5; 其它同类
AWK 字段间做除法 #cat file 500,300 200,100 取第一个字段，然后第一行除以第二行，计算出百分比，怎么写? #awk -F"," '{print $1}' file | awk 'BEGIN{FS="\n";RS="";OFS"\t"};{print $1/$2*100"%"}' 250%
#awk '{tot+=$2}{print $0} END{print "total point: "tot}' example # ls -l | awk '/^[^d]/ {print $9"\t"$5} {tot+=$5} END {print "KB:" tot}
算述运算符举例：计算 pay.txt 中每个人的总额并格式化输出 **Name 1st 2nd 3th** **VBird 23000 24000 25000** **DMTsai 21000 20000 23000**

```
[root@linux ~]# cat pay.txt | \
> awk 'NR==1{printf "%10s %10s %10s %10s %10s\n",$1,$2,$3,$4,"Total" }
NR>=2{total = $2 + $3 + $4
printf "%10s %10d %10d %10d %10.2f\n", $1, $2, $3, $4, total}'
      Name        1st        2nd        3th      Total
     VBird      23000      24000      25000   72000.00
    DMTsai      21000      20000      23000   64000.00
```

awk 的动作内{} 也是支持 if(条件) 的喔！ 举例来说，上面的指令可以修订成为这样：

```
[root@linux ~]# cat pay.txt | \
> awk '{if(NR==1) printf "%10s %10s %10s %10s %10s\n",$1,$2,$3,$4,"Total"}
NR>=2{total = $2 + $3 + $4
printf "%10s %10d %10d %10d %10.2f\n", $1, $2, $3, $4, total}'
```

2、逻辑运算符

|| 逻辑或 && 逻辑与
$ awk 'BEGIN{a=1;b=2;print (a>5 && b<=2),(a>5 || b<=2);}' 0 1

3、awk 正则运算符

~ ~! 匹配正则表达式和不匹配正则表达式
$ awk 'BEGIN{a="100testa";if(a ~ /^100*/){print "ok";}}' ok
#awk '$3 ~ /^d/ {print "ok"}' example

4、awk 关系运算符

< <= > >= != == 关系运算符

如：> < 可以作为字符串比较，也可以用作数值比较，关键看操作数如果是字符串 就会转换为字符串比较。两个都为数字 才转为数值比较。字符串按 ascii 码顺序比较。
$ awk 'BEGIN{a="11";if(a >= 9){print "ok";}}'
$ awk 'BEGIN{a=11;if(a >= 9){print "ok";}}'
ok

#awk '$2 > 10 {print "ok"}' example
查/etc/passwd 中第 3 栏小于 10 的数据，且仅列账号与第 3 栏: # awk -F":" '$3<10 {print $1 "\t " $3}' /etc/passwd # awk 'BEGIN{FS=":"}$3<10 {print $1 "\t " $3}' /etc/passwd

5、算术运算符

+ - 加，减 * / & 乘，除与求余 + - ！一元加，减和逻辑非 ^ *** 求幂 ++ -- 增加或减少，作为前缀或后缀
说明，所有用作算术运算符 进行操作，操作数自动转为数值，所有非数值都变为 0。 $ awk 'BEGIN{a="b";print a++,++a;}' 0 2

6、?: C 条件表达式

?:
$ awk 'BEGIN{a="b";print a=="b"?"ok":"err";}' ok

7、in 数组中是否存在某键值

in 运算符， i 判断数组中是否存在该键值。

$ awk 'BEGIN{a="b";arr[0]="b";arr[1]="c";print (a in arr);}'
0
$ awk 'BEGIN{a="b";arr[0]="b";arr["b"]="c";print (a in arr);}'
1

UNIT31 awk Variables

一、AWK 的部分内置变量(自定义变量的方法类似 Shell)

$0 当前记录（作为单个变量）	$1~$NF 当前记录的第 n 个字段，字段间由 FS 分隔
NF 当前记录中的字段个数	NR 已读出的记录数，就是行号，从 1 开始
FS 输入字段分隔符,默认是空格	RS 输入的记录分隔符,默认为换行符
OFS 输出字段分隔符,默认也是空格	ORS 输出的记录分隔符，默认为换行符
FILENAME 当前输入文件的名字	FNR 当前文件的当前记录号，读入新的记录时 FNR 增加。
ARGC 命令行参数的个数	SUBSEP 下标分隔符，缺省为"\034"
ARGV 命令行参数数组	RSTART 由 match() 匹配的第一个字符的索引
ENVIRON 环境变量数组	OFMT 数的输出格式，缺省为"%.6g"
ARGCIND 当前命令行参数下标	RLENGTH 由 match() 匹配的串的长度
IGNORECASE 忽略正则表达式和串的大小写	

二、内置变量应用举例（在 BEGIN 定义对整个文件生效）

1、常用操作（/^root/ 为选择表达式，$0 代表当前行记录）

```
$ awk '/^root/{print NR " line ",$0}' /etc/passwd
1 line  root:x:0:0:root:/root:/bin/bash
```

2、设置字段分隔符号(FS 使用方法, NF 是字段总数, NR 得到当前记录所在行）

```
# awk 'BEGIN{FS=":"}/^root/{print NR,$1,$NF}' /etc/passwd|head -n 1
1 root /bin/bash
```

3、设置输出字段分隔符（OFS 设置默认字段分隔符）

```
# awk 'BEGIN{FS=":";OFS="^^"}/^root/{print FNR,$1,$NF}' /etc/passwd
1^^root^^/bin/bash
$ awk 'BEGIN {ORS=""} // { print } END {print "\n"}' example
```

4、设置输出行记录分隔符(ORS 使用方法)

awk 'BEGIN{FS=":";ORS="^^"}{print FNR,$1,$NF}' /etc/passwd

1 root /bin/bash^^2 bin /sbin/nologin^^3 daemon /sbin/nologin^^.....

5、获得传入的文件名(FILENAME 使用, BEGIN 中不能获得任何与文件记录操作的变量)

awk 'BEGIN{FS=":" }{print FILENAME}' /etc/passwd

/etc/passwd

awk -F: 'END{print FILENAME}NF > 2' list.txt 当前记录中字段的个数>2

awk -F: NF > 2; 'END{print FILENAME}' list.txt

6、获得 linux 环境变量（ENVIRON 使用）

awk 'BEGIN{print ENVIRON["PATH"];}' /etc/passwd

/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

ENVIRON 是子典型数组，可以通过对应键值获得它的值。

7、输出数据格式设置：（OFMT 默认输出格式是：%.6g 保留六位小数）

awk 'BEGIN{OFMT="%.3f";print 2/3,123.11111111;}' /etc/passwd

0.667 123.111 。

8、按宽度指定分隔符（FIELDWIDTHS="4 2 2"就表示$1 宽度是 4，$2 是 2，$3 是 2）

echo 20100117054932 | awk 'BEGIN{FIELDWIDTHS="4 2 2 2 2 3"}{print $1"-"$2"-"$3,$4":"$5":"$6}'

2010-01-17 05:49:32

9、RSTART 被匹配正则表达式首位置，RLENGTH 匹配字符长度，没有找到为-1

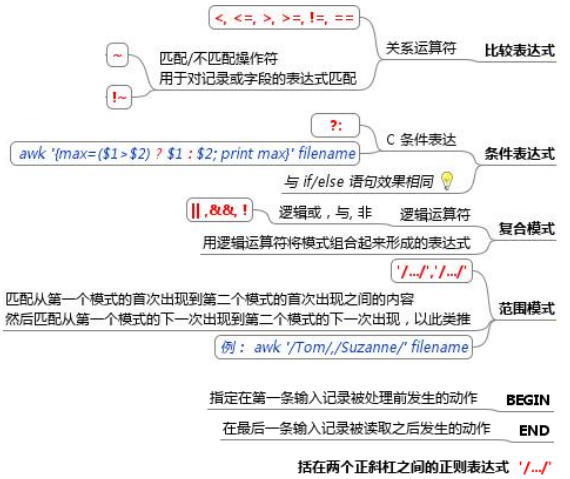
awk 'BEGIN{start=match("this is a test",/[a-z]+$/); print start, RSTART, RLENGTH }'

11 11 4

awk 'BEGIN{start=match("this is a test",/^[a-z]+$/); print start, RSTART, RLENGTH }'

0 0 -1

UNIT32 awk Patten



awk '{print ($1 > 5 ? "ok "$1: "error"$1)}' test　//$1 大于 5 打印 ok

0、‘模式’是一种表达式

1、/(RE)/:加圆括号以确保正确的值。

2、比较表达式中两操作数都是数进行数值比较，否则进行串比较。

3、awk 字符类是一种特殊的方括号表达式，此概念来自于 POSIX 标准，用于描述具有某种特定属性的字符集合。常用字符类有：

[:alnum:] 字母或数字字符	[:graph:] 既能看见又能打印的字符
[:alpha:] 字母字符	[:print:] 可打印字符（非控制字符）
[:blank:] 空格或制表符	[:punct:] 标点符号字符
[:cntrl:] 控制字符	[:space:] 空白字符，包括空格、制表、换页等
[:digit:] 数字字符	[:upper:] 大写字符
[:lower:] 小写字母字符	[:xdigit:] 十六进制数字字符

一、BEGIN|END（特殊模式，与内容无关，只执行 1 次）

1、BEGIN{action}处理第一条记录之前将 BEGIN 后面大括号之内的动作运行且只运行一次。BEGIN 通常用来设置变量。

```
#awk 'BEGIN{var="500"} {if ($3<var) print $0}' /etc/passwd
```

#awk 'BEGIN{RS=""} /root/' /etc/passwd　　//段落 grep
通过 awk+正则把括弧里的字段取出来,file.txt 内容如下： groups=001(group1), 002(group2), 003(group3) #awk 'BEGIN{FS="[()]"}{print $2}' file.txt group1 group2 group3

2、END{action}处理完最后一条记录之后将 END 后面大括号之内的动作运行且只运行一次。可在程序结束后输出一些消息。

#awk 'END {print NR}'　passwd	#awk '{nlines++}END {print nlines}' passwd
#awk '//;END {print "**";print "End"}' /etc/passwd	
#ls -l files | awk '{x += $5};END {print "total bytes: " x}'　显示出所有文件的总字节数	

二、/RE/与/re1/,/re2/(支持\ ^ $. [] | () * + ?)

1、匹配单个模式

# awk '/x.*x/' /etc/passwd	
# awk '/root/ { print }' /etc/passwd	
# awk '/^(no|so)/' test	打印所有以模式 no 或 so 开头的行
# awk '/test/{print $1 + 10}' test	记录包含 test，则$1 加 10 并打印
# awk '/^[ns]/{print $1}' test	打印以 n 或 s 开头的记录

2、匹配模式范围

与其他的搜索不同，范围模式在匹配文本时可跨越不同的记录。它输出包含匹配项部分内容的完整的记录。

#awk '/root/,/mail/' /etc/passwd

三、条件表达式

1、2 个匹配运算符：～　～!

echo 1|awk '$0 ~/^[0-9]*$/ {print $0 " is a integer"}' 1 is a integer
awk '$0 ~/^root/' /etc/passwd
ls|awk '$0 ~ /progam[0-9]+\.c/ { print $0 }'　　找 progam[0-9]+.c 的文件
awk '$3 ~/s/' /etc/passwd　　$3 包含 s 字符的记录
awk 'BEGIN {FS=":"} $7 !~ /bash/ {print $5}'　passwd　不是 bash 的用户全名 # awk 'BEGIN{FS=":";} {if ($3>499) print $1 "@zcs.cn"}' /etc/passwd
awk '$1 ~/[09][09]$/{print $1}' test　　$1 以两个数字结束就打印

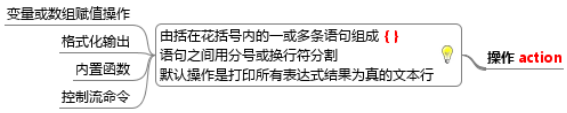
2、6 个关系运算符：< > <= >= == !=

#awk　'length($0)<=80 {print $0}'　　/etc/passwd
#awk '$3 > 500 {print "Normal user"}' /etc/passwd
awk -F: '$1=="360565687" && $2=="15026736523"' list.txt

四、使用布尔操作符的复合模式（&&与,||或,!非）

#awk　'($1 < 5) && ($2 > 8) {print "ok"}'　file1 #awk '/^d/　　||　　/x$/　　{print "ok"}'　　file1
awk 'FNR>=1 && FNR<=2' /etc/passwd
awk 'FNR==1,FNR==3' /etc/passwd
awk -F: '/root/&&/bash/{print $6}' /etc/passwd
awk -F: 'BEGIN{ORS=""}!/oprofile/&&(/home/||/bash/){print $1"\n"}' passwd

UNIT33 awk Action



一、格式化输出

1、print 为无格式输出语句：

print expr1,expr2,...,exprN print 语句显示每个表达式的串值，默认的 ORS 和 OFS 分别为\n 和 Blank。
ps -e | awk '/ tty5 / {print "tty05: " $4}' 看终端 5 的用户现在干什么 tty05: find

2、printf 为格式化输出语句（用法同 C 语言）

printf format,expr1,expr2,...,exprN //format 是规格说明，语法与 C 语言类似
使用 printf 时，不会自动输出 ORS，必须自己在 format 中使用\n 来显式地产生。
awk 'BEGIN{printf "%f\n",999}' 以符点数显示结果 999.000000
echo "65" | awk '{printf "%c\n",$0}' 以 ASCII 码显示 65 即 A awk 'BEGIN{printf "%c\n",65}' 以 ASCII 码显示 65 即 A
awk '{printf "uname: %-8s ID: %6d\n", $1, $3}' employees uname: Tom ID: 4424

3、awk 字符串连接与数字互转

awk 中数据类型，是不需要定义，自适应的。 有时候需要强制转换。我们可以通过下面操作完成。

awk 字符串转数字 $ awk 'BEGIN{a="100";b="10test10";print (a+b+0);}' 110 只需要将变量通过”+”连接运算。自动强制将字符串转为整型。非数字变成 0，发现第一个非数字字符，后面自动忽略。
awk 数字转为字符串

$ awk 'BEGIN{a=100;b=100;c=(a""b);print c}' 100100 只需要将变量与”“符号连接起来运算即可。

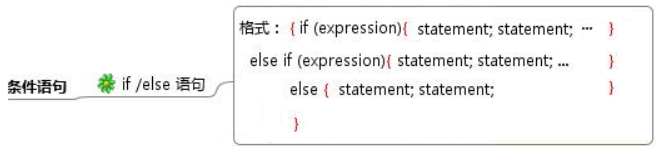
awk 字符串连接操作
$ awk 'BEGIN{a="a";b="b";c=(a""b);print c}'
ab
$ awk 'BEGIN{a="a";b="b";c=(a+b);print c}'
0
字符串连接操作通”二“，”+”号操作符。模式强制将左右 2 边的值转为 数字类型。然后进行操作。

二、重定向和管道

#awk 'BEGIN {FS = ":"}{print $1 | "sort"}' /etc/passwd

三、流控语句(C 语言相似的流控)

1、条件语句



# awk '{if (NF > max) max = NF}END {print max}'	所有输入行之中字段的最大个数
# awk -F: '{if($2~/15026736523/) print $0}' list.txt	匹配第二域,并显示该记录
# awk -F: '{if($2=="15026736523")print $0}' list.txt	精确匹配$2 的值
# awk -F: '{if($1 < $2) print $0}' list.txt	判断$1 小于$2 的记录并 print

# awk -F: '{if ($1~/360/ || $2 ~/150/)print $0}' list.txt	匹配任何一个,并集
# awk '{if (NR>0 && $1~/3605656/)print $0}' list.txt	匹配 if 条件
# awk 'BEGIN{var="200"} {if ($1<var) print $0}' list.txt	利用 begin 赋值
# awk '{if($1=="baoping"){$2=$2+1};print $2}' list.txt	算术运算
# awk '{if($1=="baoping") $2="1986";print $2}' list.txt	重新赋值并显示所有$2
# awk '{if($1=="baoping") {$2="1986";print $2}}' list.txt	重新赋值并显示单个$2
# awk '{if ($1<QQ) print $0}' QQ=360565687 list.txt	传递参数给 awk 使用

2、程序控制

3、循环(for 循环中间隔符为“;”而不是“,”)


```
awk '{for(i=1;i<100;i++){if($i ~/UN=/){print $2"\t"substr($5,2)" "$8" "$i}}}' kkk.txt
   //使用标准风格的 for 语句
awk 'a=0;{for(i=1;i<=NF;i++)a+=$i;b=a/NF;print b}' 1.txt //求 a.txt 每行的平均值
awk '{for(i=1;i<=NF;i++)a[i]+=$i}{if(NR==3){for(i=1;i<=NF;i++){print a[i]/NR}}}' 1.txt
   //计算一列的平均值
```

一段文字，每两个单词之间有空格，让所有重复的单词只剩一个？ 比如 ABC fff ddd ABC eee ABC 替换后只剩下第一个 ABC。

```
#! /bin/awk -f
{
      for (i = 1; i <= NF; i++)
        {
            ++word[$i]
```

```
            if (word[$i] == 1)
            printf(" %s  ", $i)
        }
 printf(" \n" )
}
# chmod u+x t.sh
# ./t.sh <your_file_name_here>
```

四、本讲引用文件 list.txt

54786542:13744232156	19862009
360565687:15026736523	19862007
360565687:13597572727	baoping2007
100000:13898754555	Malist
100000:13898754555	dj121M112d12nmm
1510121:155554215544:TTTa	tete
baoping207	

UNIT34 awk Built-in function

内部函数

1、内部算数函数(执行与 C 语言中名称相同的子例程相同的操作)

atan2(x,y)	y,x 范围内的余切	sin(x)	正弦
cos(x)	余弦函数	sqrt(x)	平方根
exp(x)	求幂	srand(x)	x 是 rand()函数的种子
int(x)	取整	int(x)	取整，去尾法
log(x)	自然对数	rand()	0<=随机数<1
rand()	随机数	srand(x)	用于设置 rand()的种子。

atan2(y, x) 返回 y/x 的反正切。
cos(x) 返回 x 的余弦；x 是弧度。
sin(x) 返回 x 的正弦；x 是弧度。
exp(x) 返回 x 幂函数。
log(x) 返回 x 的自然对数。
sqrt(x) 返回 x 平方根。
int(x) 返回 x 的截断至整数的值。
rand( ) 返回任意数字 n，其中 0 <= n < 1。
srand([Expr]) 将 rand 函数的种子值设置为 Expr 参数的值，或如果省略 Expr 参数则使用某天的时间。返回先前的种子值。

2、awk 提供的串函数有 (r 正则，s 和 t 串表达式，n 和 p 整数)

length(s) 返回 s 的长度

gsub(r,s) 将当前记录中的 r 替换为 s，全局，返回替换数

gsib(r,s,t) 在串 t 中全局用 s 替换 r,返回替换数

index(s,t) 返回 s 中串 t 的位置,不出现时为 0

length(s) 返回串 s 的长度

match(s,r) 返回 r 在 s 中出现的位置,不出现时为 0

split(s,a) 利用 FS 把 s 分裂成数组 a,返回字段数

split(s,A,r) 利用 r 把 s 分裂成数组 A,返回字段数

sprintf(fmt,expr_list) 根据格式串 fmt，返回经格式编排的 expr_list

sub(r,s) 在当前记录中把第一个 r 替换成 s 之后的部分，返回替换的个数

sub(r,s,t) 在 t 中把第一个 r 替换成 s 之后的部分

substr(s,p)　返回从位置 p 开始的 s 之后的部分

tolower(s)　将串 s 中的大写字母改为小写

toupper(s)　将串 s 中的小写字母改为大写

substr(String, M, [N]) 返回具有 N 参数指定的字符数量子串。子串从 String 参数指定的字符串取得，其字符以 M 参数指定的位置开始。M 参数指定为将 String 参数中的第一个字符作为编号 1。如果未指定 N 参数，则子串的长度将是 M 参数指定的位置到 String 参数的末尾 的长度。

blength [(String)] 返回 String 参数指定的字符串的长度（以字节为单位）。如果未给出 String 参数，则返回整个记录的长度（$0 记录变量）。

```
# awk -F ":" '{if($1=="test"){print$0,length($1)}}' passwd
test:x:502:502::/home/test:/bin/bash 4
# awk -F ":" '{if($1=="test") {print substr($1,1,3)}}' passwd
Tes
# echo test.txt | awk '{print substr($0,1,4)}'
test
# echo test.txt | awk '{print substr($0,6)}'
txt
# awk  -F  ":"  'gsub(/502/,213213213) {print$0}'  passwd
test:x:213213213:213213213::/home/test:/bin/bash
# awk -F ":" 'gsub(/502/,213213213,$3) {print$0}'  passwd
test x 213213213 502  /home/test /bin/bash
# awk 'length($0) > 80'                                    显示出超过 80 字符的行
# awk 'BEGIN {for (i = 1; i <= 7; i++)print int(101 * rand())}'   0 ~100 之间的 7 个随机数
# awk 'BEGIN {"date"|getline d; print d}' 通过管道把 date 的结果送给 getline，并赋给变量
```

将指定文件里最长一行的长度显示出来。expand 会将 TAB 改成 SPACE，所以是用实际的右边界来做长度的比较。

```
#expand file | awk '{if (x < length()) x = length()}END {print "maximum line length is " x}'
```

```
awk 'gsub("2007","2009",$0){print $0}' list.txt          将 2007 改为 2009 并显示
awk 'BEGIN {print index("150","5")}'                     显示 2,数所在的位数
awk '$1=="baoping" {print length($1)" "$1}' list.txt     显示结果 7 baoping,字符长度
awk '$1=="baoping" {print match($2,"7")}' list.txt       相同字符首位置,本例 4
```

awk 'BEGIN {print split("1#2#3#4",shuzu,"#")}'	结果为数组有 4 个元素 shuzu[]
echo az bg cde|awk '{sub(/[ab]/,"g",$2);print $2}' sub	用正则表达式替换,结果 gg
awk '$1=="1986" {print substr($2,1,3)}' list.txt	显示匹配的$2 从 1 后 3 个字符
awk '{print substr($1,3)}' list.txt	显示每行第 3 字符后的字符
awk 'BEGIN{STR="hello"}END{print substr(STR,4)}' list.txt	从第四位开始即 lo
echo "Stand-by" | awk '{print length($0)}'	统计变量字符个数
STR="mydoc.txt"; echo $STR\|awk '{print substr($STR,1,5)}'	结果 mydoc

3、自己定义函数

function function_name(参数 1，。。。) {

statements

return expression　　// return 可有可无

}

这节详细介绍 awk 内置函数，主要分以下 3 种类似：算数函数、字符串函数、其它一般函数、时间函数。

一、算术函数：

以下算术函数执行与 C 语言中名称相同的子例程相同的操作：

函数名 说明
atan2(y, x) 返回 y/x 的反正切。
cos(x) 返回 x 的余弦；x 是弧度。
sin(x) 返回 x 的正弦；x 是弧度。
exp(x) 返回 x 幂函数。
log(x) 返回 x 的自然对数。
sqrt(x) 返回 x 平方根。
int(x) 返回 x 的截断至整数的值。
rand( ) 返回任意数字 n，其中 0 <= n < 1。
srand([Expr]) 将 rand 函数的种子值设置为 Expr 参数的值，或如果省略
举例说明：

```
$ awk 'BEGIN{OFMT="%.3f";fs=sin(1);fe=exp(10);fl=log(10);fi=int(3.1415);print fs,fe,fl,fi;}'
0.841 22026.466 2.303 3
```

OFMT 设置输出数据格式是保留 3 位小数

获得随机数：

```
$ awk 'BEGIN{srand();fr=int(100*rand());print fr;}'
78
$ awk 'BEGIN{srand();fr=int(100*rand());print fr;}'
31
$ awk 'BEGIN{srand();fr=int(100*rand());print fr;}'
41
```

二、字符串函数是：

函数 说明

gsub(Ere, Repl, [In]) 除了正则表达式所有具体值被替代这点，它和 sub 函数完全一样地执行，。

sub(Ere, Repl, [In]) 用 Repl 参数指定的字符串替换 In 参数指定的字符串中的由 Ere 参数指定的扩展正则表达式的第一个具体值。sub 函数返回替换的数量。出现在 Repl 参数指定的字符串中的 & （和符号)由 In 参数指定的与 Ere 参数的指定的扩展正则表达式匹配的字符串替换。如果未指定 In 参数，缺省值是整个记录（$0 记录变量）。

index(String1, String2) 在由 String1 参数指定的字符串（其中有出现 String2 指定的参数）中，返回位置，从 1 开始编号。如果 String2 参数不在 String1 参数中出现，则返回 0（零）。

length [(String)] 返回 String 参数指定的字符串的长度（字符形式）。如果未给出 String 参数，则返回整个记录的长度（$0 记录变量）。

blength [(String)] 返回 String 参数指定的字符串的长度（以字节为单位）。如果未给出 String 参数，则返回整个记录的长度（$0 记录变量）。

substr(String, M, [N]) 返回具有 N 参数指定的字符数量子串。子串从 String 参数指定的字符串取得，其字符以 M 参数指定的位置开始。M 参数指定为将 String 参数中的第一个字符作为编号 1。如果未指定 N 参数，则子串的长度将是 M 参数指定的位置到 String 参数的末尾 的长度。

match(String, Ere) 在 String 参数指定的字符串（Ere 参数指定的扩展正则表达式出现在其中）中返回位置（字符形式），从 1 开始编号，或如果 Ere 参数不出现，则返回 0（零）。RSTART 特殊变量设置为返回值。RLENGTH 特殊变量设置为匹配的字符串的长度，或如果未找到任何匹配，则设置为 -1（负一）。

split(String, A, [Ere]) 将 String 参数指定的参数分割为数组元素 A[1], A[2], . . ., A[n]，并

返回 n 变量的值。此分隔可以通过 Ere 参数指定的扩展正则表达式进行，或用当前字段分隔符（FS 特殊变量）来进行（如果没有给出 Ere 参数）。除非上下文指明特定的元素还应具有一个数字值，否则 A 数组中的元素用字符串值来创建。

tolower(String) 返回 String 参数指定的字符串，字符串中每个大写字符将更改为小写。大写和小写的映射由当前语言环境的 LC_CTYPE 范畴定义。

toupper(String) 返回 String 参数指定的字符串，字符串中每个小写字符将更改为大写。大写和小写的映射由当前语言环境的 LC_CTYPE 范畴定义。

sprintf(Format, Expr, Expr, . . .) 根据 Format 参数指定的 printf 子例程格式字符串来格式化 Expr 参数指定的表达式并返回最后生成的字符串。

Ere 都可以是正则表达式

gsub,sub 使用

$ awk 'BEGIN{info="this is a test2010test!";gsub(/[0-9]+/,"!",info);print info}'

this is a test!test!

在 info 中查找满足正则表达式，/[0-9]+/ 用”“替换，并且替换后的值，赋值给 info 未给 info 值，默认是$0

查找字符串（index 使用）

$ awk 'BEGIN{info="this is a test2010test!";print index(info,"test")?"ok":"no found";}'

ok

未找到，返回 0

正则表达式匹配查找(match 使用）

$ awk 'BEGIN{info="this is a test2010test!";print match(info,/[0-9]+/)?"ok":"no found";}'

ok

截取字符串(substr 使用）

$ awk 'BEGIN{info="this is a test2010test!";print substr(info,4,10);}'

s is a tes

从第 4 个 字符开始，截取 10 个长度字符串

字符串分割（split 使用）

$ awk 'BEGIN{info="this is a test";split(info,tA," ");print length(tA);for(k in tA){print k,tA[k];}}'

4

4 test

1 this

2 is

3 a

分割 info,动态创建数组 tA,这里比较有意思，awk for …in 循环，是一个无序的循环。 并不是从数组下标 1…n ，因此使用时候需要注意。

格式化字符串输出（sprintf 使用）

格式化字符串格式：

其中格式化字符串包括两部分内容：一部分是正常字符，这些字符将按原样输出；另一部分是格式化规定字符，以"%"开始，后跟一个或几个规定字符,用来确定输出内容格式。

格式符	说明
%d	十进制有符号整数
%u	十进制无符号整数
%f	浮点数
%s	字符串
%c	单个字符
%p	指针的值
%e	指数形式的浮点数
%x %X	无符号以十六进制表示的整数
%o	无符号以八进制表示的整数
%g	自动选择合适的表示法

```
$ awk 'BEGIN{n1=124.113;n2=-1.224;n3=1.2345; printf
("%.2f,%.2u,%.2g,%X,%o\n",n1,n2,n3,n1,n1);}'
124.11,18446744073709551615,1.2,7C,174
```

三、一般函数是：

函数	说明
close(Expression)	用同一个带字符串值的 Expression 参数来关闭由 print 或 printf 语句打开的或调用 getline 函数打开的文件或管道。如果文件或管道成功关闭，则返回 0；其它情况下返回非零值。

如果打算写一个文件，并稍后在同一个程序中读取文件，则 close 语句是必需的。

system(Command) 执行 Command 参数指定的命令，并返回退出状态。等同于 system 子例程。

Expression | getline [Variable] 从来自 Expression 参数指定的命令的输出中通过管道传送的流中读取一个输入记录，并将该记录的值指定给 Variable 参数指定的变量。如果当前未打开将 Expression 参数的值作为其命令名称的流，则创建流。创建的流等同于调用 popen 子例程，此时 Command 参数取 Expression 参数的值且 Mode 参数设置为一个是 r 的值。只要流保留打开且 Expression 参数求得同一个字符串，则对 getline 函数的每次后续调用读取另一个记录。如果未指定 Variable 参数，则 $0 记录变量和 NF 特殊变量设置为从流读取的记录。

getline [Variable] < Expression 从 Expression 参数指定的文件读取输入的下一个记录，并将 Variable 参数指定的变量设置为该记录的值。只要流保留打开且 Expression 参数对同一个字符串求值，则对 getline 函数的每次后续调用读取另一个记录。如果未指定 Variable 参数，则 $0 记录变量和 NF 特殊变量设置为从流读取的记录。

getline [Variable] 将 Variable 参数指定的变量设置为从当前输入文件读取的下一个输入记录。如果未指定 Variable 参数，则 $0 记录变量设置为该记录的值，还将设置 NF、NR 和 FNR 特殊变量。

打开外部文件（close 用法）

```
$ awk 'BEGIN{while("cat /etc/passwd"|getline){print
$0;};close("/etc/passwd");}'
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
```

逐行读取外部文件(getline 使用方法）

```
$ awk 'BEGIN{while(getline < "/etc/passwd"){print
$0;};close("/etc/passwd");}'
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
$ awk 'BEGIN{print "Enter your name:";getline
name;print name;}'
Enter your name:
chengmo
chengmo
```

调用外部应用程序(system 使用方法）

```
$ awk 'BEGIN{b=system("ls -al");print b;}'
total 42092
drwxr-xr-x 14 chengmo chengmo     4096 09-30 17:47 .
drwxr-xr-x 95 root    root        4096 10-08 14:01 ..
```

b 返回值，是执行结果。

四、时间函数

函数名 说明
mktime(YYYY MM DD HH MM SS[DST]) 生成时间格式
strftime([format [, timestamp]]) 格式化时间输出，将时间戳转为时间字符串
具体格式，见下表.
systime() 得到时间戳,返回从 1970 年 1 月 1 日开始到当前时间(不计闰年)的整秒数创建指定时间(mktime 使用)

```
$ awk 'BEGIN{tstamp=mktime("2001 01 01 12 12 12");print strftime("%c",tstamp);}'
2001 年 01 月 01 日 星期一 12 时 12 分 12 秒
$ awk 'BEGIN{tstamp1=mktime("2001 01 01 12 12 12");tstamp2=mktime("2001 02 01 0 0 0");print tstamp2-tstamp1;}'
2634468
```

求 2 个时间段中间时间差,介绍了 strftime 使用方法

```
$ awk 'BEGIN{tstamp1=mktime("2001 01 01 12 12 12");tstamp2=systime();print tstamp2-tstamp1;}'
308201392
```

strftime 日期和时间格式说明符

格式 描述
%a 星期几的缩写(Sun)
%A 星期几的完整写法(Sunday)
%b 月名的缩写(Oct)
%B 月名的完整写法(October)
%c 本地日期和时间
%d 十进制日期
%D 日期 08/20/99

%e 日期，如果只有一位会补上一个空格

%H 用十进制表示 24 小时格式的小时

%I 用十进制表示 12 小时格式的小时

%j 从 1 月 1 日起一年中的第几天

%m 十进制表示的月份

%M 十进制表示的分钟

%p 12 小时表示法(AM/PM)

%S 十进制表示的秒

%U 十进制表示的一年中的第几个星期(星期天作为一个星期的开始)

%w 十进制表示的星期几(星期天是 0)

%W 十进制表示的一年中的第几个星期(星期一作为一个星期的开始)

%x 重新设置本地日期(08/20/99)

%X 重新设置本地时间(12：00：00)

%y 两位数字表示的年(99)

%Y 当前月份

%Z 时区(PDT)

%% 百分号(%)

UNIT35 awk Integrated case

filename 为以下文件，第二列用“-”隔开，以避免与 FS 冲突

```
ID   Name       Age City Country Tel              Salary  Children
1001 Steven     25  NY   U.S.A   +01-02-323222    $4900   2
1002 Huang-Yu   30  BJ   CHN     +86-10-36789966  $6000   1
1003 Fish-Mad   27  SG   SG      +65-67456632     $3000   3
1004 Vale-Kiss  46  LD   ENG     +44-20-87634321  $6280   3
```

一、无 pattern 的 action 实例

awk ‘{print NR $1 $NF}’ data.txt 打印行号，第一列和最后一列，中间无分隔符
awk ‘{print $1,$NF}’ data.txt 打印第一列和最后一列，并且中间有分隔符
awk ‘{print $0,$NF+10}’ data.txt 打印整行，并打印最后一行加上 10 的结果

二、有 pattern 的 action 实例

awk ‘/[0-9]/’ data.txt 打印记录中任意列包含数字 0－9 的行
awk ‘/01/||/02/’ data.txt 打印包含 01 或者 02 的行
awk ‘/01/,/02/’ data.txt 打印既包含 01 又包含 02 的行；等同 awk ‘/01/&&/02’
awk ‘$1= =1001{print $2}’ data.txt 打印符合第一列等于 1001 的第二列
awk ‘$2= =”Steven”{print}’data.txt 打印符合第二列等于 Steven 的那些行
awk ‘$3>20&&$3<30’ data.txt 打印第三列在 20 到 30 之间的那些行
nawk ‘$3*$NF<100’ data.txt 打印第三列和最后一列乘积小于 100 的那些行
awk ‘$6~/01/{print $2}’ data.txt 打印符合仅仅第六列里包含 01 那些行的第二列
awk ‘NR>3{print $1,$2}’ data.txt 从第四行才开始打印第一列和第二列

三、有 BEGIN、END 并包含 FS、OFS 的实例

awk -F”+” ‘{print $1}’ data.txt 按+为分隔符来分隔列，并打印第一列

gawk -F’[+\t$]’ ‘{print $6,$7,$8,$9}’ data.txt 按+或\t 或$都可作分隔符，并打印指定列
gawk ‘BEGIN{FS=”[\t+$]”}{print $6,$8,$9}’ data.txt 按+,\t,$（顺序无所谓）为分割符,并打印指定列
gawk ‘BEGIN{FS=”[\t+$]”;OFS=”%”}{print $6,$7,$8,$9}’ data.txt 按+,\t,$为分割符，用%作输出分隔符打印指定列
gawk -F‘[[ ]]’ ‘{print $1}’ data.txt 按 [或]为分隔符，并打印第一列
gawk -F‘[ ][ ]’ ‘{print $1}’ data.txt 按[或] 为分隔符，并打印第一列

关于-F 和 BEGIN 内的 FS 定义特别注意如下：

-F 参数后紧跟单个分隔符，则用双引号“”，例如 -F”+”
-F 参数后紧跟多个分隔符，则用单引号‘ ’并用[ ]，中间顺序无所谓，例如-F’[+$]’
BEGIN 中的 FS 无论是单个还是多个分隔，均用双引号“”，多个则需加[ ] 例如：nawk ‘BEGIN{FS=”[\t+$]”}{print $6,$8,$9}’ data.txt
BEGIN 中若同时有 FS 和 OFS，则建议中间用分号“；”来分隔开，否则提示出错 例如：nawk ‘BEGIN{FS=”[\t+$]”;OFS=”%”}{print $6,$7,$8,$9}’ data.txt
半方括号[ ]也可以作为分隔符，顺序无所谓。例如：-F‘[[]]’ 或 -F‘[][]’
在用多个分隔符的时候，为避免语法错误或想得到正确的结果，最好用 nawk

四、条件 if else 和 for while 循环实例

gawk ’{print($3>25?$2"old":$2"yong")}’ data.txt $3>25 显某某 old 否某某 young
awk ’{if($1>1002)print $2;else print $3}’ data.txt 若$2>1002 打印$2 否则打印$3
awk ’NR>1{if($3<30)print $2;else if($3<40)print $2"--m";else exit;}END{print 10}’ data.txt 从第二行开始判断，$3<30 打印$2，30<$3 <40 打印$2--m，都不符合就退出处理行，但是 END 后的操作仍执行
awk ’BEGIN{ORS=""}{for(i=1;i<NF-2;i++) print $i"\t"}{print "\n"}’ data.txt 打印出每行的前面几列，最后几列不打印。特别注意：默认情况下，print 每执行一次时，另起一行打印。因此为避免每打印一列就重新一行，设置 ORS 为空，但是在每打印完符合条件的所有列后，需要手工换行（循环外的 print 起此作用）。

awk 'BEGIN{ORS=""}{i=1;while(i<NF-2){print $i"\t";i++}}{print "\n"}' data.txt 功能同上，打印出每行的前面几列，最后几列不打印。

五、数学运算和字符串操作实例

gawk '{print 2^5+sin(2.1)+int(0.9)}' data.txt 在每一行都打印运算值（32.8632）
awk 'END{print length("How are you?")}' data.txt 打印出字符串的长度（12）
awk 'END{print index("How are you?","you")}' data.txt 返回 you 在字符串中的开始位置（9）
gawk '{gsub(/ \$/,"");print $0}' data.txt 把每行的$符号用空替掉，并打印行
awk 'END{print substr("How are you?",9,3)}' data.txt 从字符串的第九个位置开始截出后面 3 个长度的子字符串
gawk '{print toupper($2), tolower($2)}' data.txt 分别打印第二列大写和小写
gawk 'END{print match("How are you you?",/you/),RSTART,RLENGTH}' data.txt 打印 you 在字符串中第一个匹配的位置以及长度（9，9，3）
gawk 'END{str="How are you doing?";sub(/o/,"0",str);print str}' data.txt 将字符串变量指定字符中第一个符合含有 o 的用 0 替换调 特别注意：sub 函数的第三个参数不可直接用字符串，而必须用字符串变量。
awk 'END{print split("Jan,Feb,Mar,Apr,May",mymonths,","),mymonths[2]}' data.txt 将字符串用第三个参数分隔符号进行分隔，把分隔的各个元素放在第二个参数的数组中，函数返回分隔得到的元素数目。结果：（5，Feb）

六、数组与关联数组（a[1]，a[$1] a[$0]，a[b[i]]）

awk '{for(i=1;i<NF;i++) {a[i]=$i; print a[i]}}' data.txt 把每一列按行打印
awk '{a[$1]=$2}END{for(x in a) print a[x]}' data.txt 数组 a 关联到第一列，并将第二列值赋给 a，最后打印 a 中的所有内容。注意：但不是顺序打印。for 中的 x 是随意变量，awk 自动确定。

```
awk ‘{b[i]=$1;a[$1]=$2;i++}END{for(j=1;j<i;j++) print b[j],a[b[j]]}’ data.txt
```

实现从第二行开始打印每一行的第一列和第二列的功能。

注意：i 从 0 开始取值，而 j 从 1 开始。a 为关联数组，b 为顺序数组。

七、多输入文件和 NR、FNR 的实例

```
a. awk ‘BEGIN{OFS=FS=”:”} NR==FNR{a[$1]=$2} NR>FNR{$2=a[$1];print}’ /etc/shadow /etc/passwd
```

该语句中第一个模式匹配到第一个输入文件 shadow，第二个模式匹配到第二个输入文件 passwd，并使用关联数组，实现不同文件关联列的其他列值相互替换。

A. awk 对多输入文件的执行顺序是，先将代码作用于第一个文件（一行行读入），然后该重复的代码又作用于第二个文件，再作用于第三个文件。

B. awk 对多输入文件的执行顺序产生了行序号的问题。当第一个文件执行完，下次读入第二个文件，那么第二个文件的第一行怎么算呢？如果又计为 1 的话，那不就两个 1 了么？（因为第一个文件也有第一行）。这就是 NR 和 FNR 的问题。

NR ：全局行数（第二个文件的第一行接着第一个文件尾行数顺序计数）

FNR：当前文件自身的行数（不考虑前几个输入文件的自身行数及总数）

例如:data1.txt 中有 40 行,data2.txt 中有 50 行,那么 awk ‘{}’ data1.txt data2.txt

NR 的值依次为：1，2……40，41，42……90

FNR 的值依次为：1，2……40， 1， 2……50

八、重定向输出和特别函数 getline 实例

```
awk ‘{print FILENAME,$0}’ data1.txt data2.txt >data_all.txt
```

把第一个文件和第二个文件合并到 data_all.txt 中,新文件第一列为原始文件名,后面列为原始文件内容。

```
awk ‘$1!=fd{close(fd);fd=$1} {print substr($0,index($0,“ ”)+1)>$1}’ data_all.txt
```

把合并后的新文件 data.txt 重新拆开成原来的两个子文件，依照新文件的第一列来产生新文件名。生成一个完整子文件后，关闭之；再生成下一个子文件。

```
gawk ‘BEGIN{system(“echo \”input your name:\” ”);getline var; print “\nYour
```

name is”,d,”\n”}’

系统提示输入名字，然后将输入的名字打印出来。特别注意：该 awk 语句后没有输入文件名，而是利用键盘输入作为文件名。

A. getline 从整体上来说，应这么理解它的用法：

当其左右无重定向符 | 或 < 时，getline 作用于当前文件，读入当前文件的第一行给其后跟的变量 var 或$0（无变量）；应该注意到，由于 awk 在处理 getline 之前已经读入了一行，所以 getline 得到的返回结果是隔行的。

当其左右有重定向符 | 或 < 时，getline 则作用于定向输入文件，由于该文件是刚打开，并没有被 awk 读入一行，只是 getline 读入，那么 getline 返回的是该文件的第一行，而不是隔行。

B. getline 用法大致可分为三大类（每大类又分两小类），即总共有 6 种用法。代码如下：

gawk ‘BEGIN{“cat data.txt”|getline d; print d}’ data2.txt

gawk ‘BEGIN{“cat data.txt”|getline; print $0}’ data2.txt

gawk ‘BEGIN{getline d < “data.txt”; print d}’ data2.txt

gawk ‘BEGIN{getline < “data.txt”; print $0}’ data2.txt

以上四行代码均实现“只打印 data.txt 文件的第一行”（若打印全部行，用循环）

eg. nawk ‘BEGIN{FS=”:”;while(getline<”/etc/passwd”>0){print $1}}’ data.txt

gawk ‘{getline d; print d”#”$3}’ data.txt

awk 首先读入第一行，接着处理 getline 函数，然后把下一行指定给变量 d，再先打印 d，由于 d 后面有换行符，所以后面紧跟的＃会覆盖 d，后面的$3 同样也会覆盖 d。

gawk ‘{getline; print $0”#”$3}’ data.txt

awk 首先读入第一行接着处理 getline 函数，然后把下一行指定给$0，现在的$0 已经是下一行内容，后面的#和$3（从$0 中取）会覆盖$0 的内容。

UNIT36 awk 常见应用

一、awk 查看 ip 连接数

处理文本，是 awk 的强项了。 无论性能已经速度都是让人惊叹！

```
$ awk 'BEGIN{
    while("netstat -an"|getline){
        if( $5 ~ /[1-255]/)
        {
            split($5,t1,":");
            tarr[t1[1]]++;
        }
    }
    for(k in tarr)
        {
            print k,tarr[k] | "sort -r -n -k2";
        }
};'
```

$5 是 netstat -an 第 5 个字段。默认就是对方连接 ip 以及端口。

```
$ time awk 'BEGIN{while("netstat -an"|getline){if( $5 ~
/[1-255]/){split($5,t1,":");tarr[t1[1]]++;}}for(k in
tarr){print k,tarr[k] | "sort -r -n -k2";}};'
211.151.33.14 28
113.65.21.200 14
121.32.89.106 13
60.191.178.230 12
118.133.177.104 12
58.61.152.154 11
```

```
real    0m1.149s
user    0m0.032s
sys     0m1.055s
```

awk 常见应用系列，会一直更新！我会把这些年我在服务器管理方面一些代码总结归纳与朋友共同学习。

二、awk 实现实时监控网卡流量脚本

通过第 3 方工具获得网卡流量，这个大家一定很清楚。其实通过脚本一样可以实现效果。下面是我个人工作中整理的数据。以下是 shell 脚本统计网卡流量。

实现原理：

```
$ cat /proc/net/dev
Inter-|   Receive                                                |  Transmit
face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo
colls carrier compressed
    lo:1068205690 1288942839    0    0    0     0          0          0 1068205690 1288942839
0    0    0     0       0          0
  eth0:91581844 334143895    0    0    0     0          0 145541676 4205113078 3435231517
0    0    0     0       0          0
```

proc/net/dev 文件保存了网卡总流量信息，通过间隔一段间隔，将入网卡与出记录加起来。减去之前就得到实际速率。

程序代码：

```
awk 'BEGIN{
OFMT="%.3f";
devf="/proc/net/dev";
while(("cat "devf) | getline)
{
    if($0 ~ /:/ && ($10+0) > 0)
```

```
    {
        split($1,tarr,":");
        net[tarr[1]]=$10+tarr[2];
        print tarr[1],$10+tarr[2];
    }
}
close(devf);
while((system("sleep 1 ")) >=0)
{
    system("clear");
    while( getline < devf )
    {
        if($0 ~ /:/ && ($10+0) > 0)
            {
                split($1,tarr,":");
                if(tarr[1] in net)
                {
                    print
tarr[1],":",($10+tarr[2]-net[tarr[1]])*8/1024,"kb/s";
                    net[tarr[1]]=$10+tarr[2];
                }
            }
    }
    close(devf);
}
}'
```

说明：第一个 while 是获得总的初始值，$1 是网卡出流量，$10 是网卡进流量。第 2 个 while 会间隔 1 秒钟启动一次。计算总流量差得到平均每秒流量。

注意：通过 getline 逐行读取文件，需要 close 关闭 。否则在第 2 次 while 循环中不能获得数据。

三、awk 分析 web 日志（页面执行时间）

前一段时间，我写过一篇文章，shell 脚本分析 nginx 日志访问次数最多及最耗时的页面(慢查询），其中提到了分析耗时页面重要性。今天主要讲的，是通过 awk 分析日志，快捷得到执行时间。在性能以及效率方面比前一篇提到的有很大提高！

1、web 日志文件格式

222.83.181.42 - - [09/Oct/2010:04:04:03 +0800] GET /pages/international/tejia.php HTTP/1.1 "200" 15708 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; Sicent; WoShiHoney.B; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)" "-" 0.037

按照空格分隔的话，最后一个字段[0.037] 是页面执行时间，第 7 个字段 是页面访问地址。

2、执行代码

```
awk 'BEGIN{
print "Enter log file:";
getline logs;
#logs="/var/log/nginx/access.log-20101008";
OFMT="%.3f";


while(getline < logs)
{
    split($7,atmp,"?");
    aListNum[atmp[1]]+=1;
    aListTime[atmp[1]]+=$NF;
    ilen++;
}
close(logs);
print "\r\ntotal:",ilen,"\r\n==================================\r\n";
```

```
for(k in aListNum)
{
    print k,aListNum[k],aListTime[k]/aListNum[k] | "sort -r -n -k3";
}

}'
```

```
Enter log file:
/var/log/nginx/access.log-20101008

total: 422780
=====================================

/images/activity/seckill_new_bg.gif 1 64.755
/js/international/international_search.js 1 53.396
/images/mak_line.gif 1 43.007
/js/jquery.js 3 42.472
/images/international/seckill/seckill_07.jpg 1 40.568
```

```
real    0m5.153s
user    0m4.911s
sys     0m0.143s
```

422780 条日志，统计完成速度是：5 秒左右

四、awk 多行合并【next 使用介绍】

在 awk 进行文本处理时候，我们可能会遇到。将多行合并到一行显示问题。 有点象 sql 里面，经常遇到的行转列的问题。 这里需要用到 next 语句。

Awk next 语句使用：在循环逐行匹配，如果遇到 next,就会跳过当前行，直接忽略下面语句。而进行下一行匹配。

text.txt 内容是：

```
a
b
c
d
e
$ awk 'NR%2==1{next}{print NR,$0;}' text.txt
```

2 b

4 d

当记录行号除以 2 余 1，就跳过当前行。下面的 print NR,$0 也不会执行。 下一行开始，程序有开始判断 NR%2 值。这个时候记录行号是：2 ，就会执行下面语句块：'print NR,$0'

awk next 使用实例：

要求：

文件：text.txt 格式：

```
httpd                    ok
tomcat                   ok
sendmail                 ok
web02[192.168.2.101]
httpd                    ok
postfix                 ok
web03[192.168.2.102]
mysqld                  ok
httpd                   ok
```

需要通过 awk 将输出格式变成：

```
web01[192.168.2.100]:   httpd           ok
web01[192.168.2.100]:   tomcat          ok
web01[192.168.2.100]:   sendmail        ok
web02[192.168.2.101]:   httpd           ok
web02[192.168.2.101]:   postfix         ok
web03[192.168.2.102]:   mysqld          ok
web03[192.168.2.102]:   httpd           ok
```

分析：

分析发现需要将包含有“web”行进行跳过，然后需要将内容与下面行合并为一行。

```
$ awk '/^web/{T=$0;next;}{print T":\t"$0;}' test.txt
web01[192.168.2.100]:   httpd           ok
```

```
web01[192.168.2.100]:   tomcat          ok
web01[192.168.2.100]:   sendmail        ok
web02[192.168.2.101]:   httpd           ok
web02[192.168.2.101]:   postfix         ok
web03[192.168.2.102]:   mysqld           ok
web03[192.168.2.102]:   httpd           ok
```

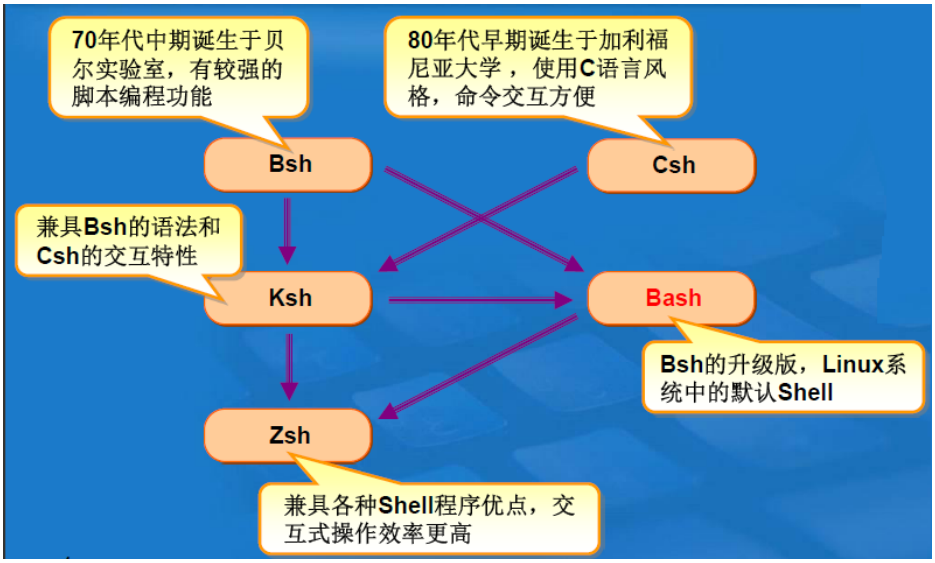
next 在多行合并，以及选择性输出方面，非常方便。大家在使用时候不妨试试。

UNIT37 Shell level test

一、OVERVIEW 篇

1. 有很多种 shell，你熟悉几种？ 各个 shell 的 home page 在那里？

2. 为什么说 zsh 是目前为止功能最为强大的 shell.



3. 为什么说 pdksh 功能较弱？

4. ksh88 与 ksh93 有何区别？

5. 为什么 shell 编程最好用 ksh?

6. 你的系统都有哪些 shell? 版本是多少？

7. 你知道 POSIX 吗？最新版本是多少？和你的 shell 有什么关系？

8. /sbin/sh 和 /bin/sh 有何区别?

9. 你分析过 1000 行以上的 shell 程序吗?

10. 各种 shell 的变量名长度有何限制?

11. 各种 shell 的 array size 有何限制?

12. FIFO 比 co-process 有什么优点?

13. (..) 产生的 subshell 与调用另一个脚本产生的 subshell 有何不同?

14. shell 中的函数可以递归吗? 设计时要注意什么?

15. 如何用 subshell 产生的多个值改变多个 shell 中的变量?

16. File pattern 与 regular expression 有什么不同?

17. shell 中含有大量文档. 不想在每行之前用 # 注释. 有那两种方法可以实现?

18. 用 shell 程序安装配置一个二进制可执行软件. 希望将被安装软件含在 shell 程序中形成一个安装文件, 可以吗?

19. 可以在一个 shell 程序里包含 ksh 及 perl 两种代码吗?

20. 如何保证 shell 程序只有单一 instance 运行? (有难度, 绝非一个 ps 命令可以搞定)

21. 当有同名的 alias, function, builtin command, external command 时，如何知道并控制其运行顺序？

22. 如何在自定义 function 中调用同名的 builtin command ？

23. 怎样运行各种 shell 时具有 login shell 的行为?

24. 为何 ksh, rksh, pfksh 都是同一个 hard link? 功能却不同？

25. 当 while read .. 这样的语句用于 pipe 右侧时，需注意什么？

26. 对于多个相连的 pipe，最后的返回码是 pipe 中那个命令的？ 如何取得所有命令的返回码？

27. cmd >a 2>a 和 cmd >a 2>&1 为什么不同？

28. Stdin, Stderr 可以关闭吗? 有什么结果?

29. GNU tools 与其他 Unix 上的相比有什么特点？ 为什么说不能过分依赖 GNU 扩展？

30. VI 及 VIM 对行长度及 file size 有什么限制？

31. 请写出对 field 1 (数字)， field 3 (数字倒序)， field 3 (ascii) 进行排序的完整语句.

32. 如何用 diff 及其他工具写一个版本控制系统？（要有 check in, check out, file lock .. 等功能）

二、 SED 篇

1. 你能看懂 sed one-lines 中的每一条语句吗?

2. /regex/! command 与 /regex/ !command 有什么区别?

3. 你能熟练使用 N,n, P,p,D,d,H,h,G,g,x,:,b,t 吗?

4. 什么是 sed 高级编程中的 lookup table 技术

5. sed debuger 的原理是什么?

6. 为什么 sed 的 guru 和 fans 比 awk 的多. (请看 yahoo groups)

下面几题根据使用的技术，难度不同. （难度 最小:1,最大:5）

7. 将数据文件中的每个词的第一个字母变成大写. (难度: 2 - 4)

8. 在 sed 中实现记数器. 可加 1 或减 1. (难度: 3 - 4)

9. 提取 html 文件中 table 中每个单员的内容(table 可嵌套). (难度: 4 - 5)

每一个 cell 做为一个单员输出：

Table #1, Row #1, Column #1

Contents .....

Table #1, Row #1, Column #2

10. 一般的 sed 的 regex 的匹配都是 greedy 的。如何用 sed 实现 lazy 匹配?

三、AWK 篇

1. awk, oawk, nawk, gawk, mawk 有什么区别?

2. 在一个 awk 文件中, 第一行可以如下吗?

```
#! /bin/awk -F: -f
```

3. awk -F"" 与 awk -F "" 有区别吗?

4. 可以这样设置 FS 吗?

```
FS = "[ \t]+\|[ \t]+"
```

5. gawk 有一个扩展表达式与其他 GNU 工具不同, 是哪个, 为什么?

6. 那两种方法可以实现大小写无关匹配?

7. 下列两句有何区别?

```
awk '$0 ~ "[ \t\n]"'
awk '$0 ~ /[ \t\n]/'
```

8. FS=" " 和 FS=" +" 有何区别?

9. 如何将每一个字符作为一个 field ？

10. 如何将整个文件作为一个 field ？

11. RS="" 与 RS="\n\n+" 有何区别?

12. NR 和 FNR 有何区别?

13. getline < "file" 改变那些 builtin 变量?

14. 如何不打印最后一个 field? (不用循环)

15. $1 ~ /aaa/ 与 /aaa/ ~ $1 有何区别?

16. a = /a/ 是什么意思?

17. awk 中的 array 可以排序吗?

18. 如何将 awk 中得到的值赋给 shell 变量.

19. 如何模拟二维数组？

20. 你的 awk 数组最多可能的单元是多少？有限制没有？

21. 你的 awk 中 field number, record length 及 file size 有何限制？

22. awk 中如何删除 array 及 关闭管道？

23. 如何完成 rev 功能？

24. 如何在 awk 中使用 coprocess？

25. 你写过 100 行以上的 awk 程序吗？

四、REGEX 篇

1. 传统 regex, POSIX regex, GNU regex, PCRE 都有什么特点及异同之处？

2. ERE 是 BRE 的 super set 吗？

3. ERE 看起来强大，但使用 ERE 有一点最不爽，是什么？

4. 依赖 GNU 扩展的危害是什么？

5. Back reference 可以嵌套吗？

6. DFA 和 NFA 有什么特点及区别？

7. 常用的 [e]grep, [ng]awk, [g]sed, perl 哪些是 DFA？ 哪些是 NFA？ 因此这些程序具有哪些相应的特点？

8. Greedy regex 和 lazy regex 有何区别？ 哪些工具是 lazy regex.

9. 猜一猜一个符合 RFC 标准的 email adderss 大概需要多长的 regex 来匹配？

10. 你是否有这样的经历： 某个使用过的 regex， 用在另个软件中却不行。

也就是说你无法确定某个 regex 在同一系统的不同软件中或在不同系统的相同软件中肯定能用?

五、附加题

1. 为什么说 CU 的整体较为业余，但 shell 版的水平相对较高?

2. 为什么 shell 版中的大部分高手都是玩 Linux 的，而低手往往是 Sco unix，HP-UX 等其他平台的?

3. 为什么当低手怯怯地抛出一个问题，高手快速地给出了漂亮的解答，却往往最后并没有解决低手的问题?

UNIT38 Common used script

一、用户管理

1、用 grep /etc/passwd 判断账号是否存在:显示$ZH is exist 或$ZH is not exist

```
#!/bin/bash
shopt -s -o nounset
echo 'Enter  username to check:'
read  ZH
echo ''
                echo '******************'
if grep -q $ZH /etc/passwd; then
                echo "$ZH is exist."
        else
                echo '$ZH is not exist.'
fi
                echo '******************'
~
```

2、用 id 判断: 账号是否存在:若无此账号输出 custom，相反则输出 engineer

```
#!/bin/bash
shopt -s -o nounset
#echo ""
#echo -n  "Enter username:"
#read ZH

TMP1="/tmp/tmp.$$"
TMP2="/tmp/taccount.$$"
dialog --inputbox "User:" 10 40 2> $TMP1
ZH=$(cat $TMP1)

id $ZH &> $TMP2  && grep $ZH -q $TMP2

if [ $? -eq 0 ] ; then
           dialog --msgbox "$ZH is engineer"  10 40
   else
           dialog --msgbox "$ZH is customer"  10 40
fi

rm -f  $TMP1 $TMP2
#sleep 5
clear
/bin/taccount2.sh
```

3、批量添加账号

```
#! /bin/bash
read -p "the number of user(1-99):"  Num
read -p "the prefix of username:"  Pre
read -p "the missing time:"  Etime
read -p "the password of users:"  Pw

I=1
while [ $I -le $Num ]

do

   if [ $I -lt 10 ];then
      Un="${Pre}0$I"
   else
      Un="${Pre}$I"
   fi

useradd -e $Etime $Un
echo $Pw|passwd --stdin $Un &> /dev/null
     let I++

done
```

4、批量删除账号

```
#!/bin/bash
if  [ $# -le 0 ];then
   echo "error:there is no prefix."
   echo "usage:$0 userprefix."
   exit 1
fi
tar zcvf /usr.config.tar.gz /etc/shadow /etc/passwd /etc/group &> /dev/null
delete=$(cut -d ":" /etc/passwd|grep $1|grep -v "root")
for Un in $delete
do
userdel -r $Un &> /dev/null
done

```

5、批量创建用户

#mkdir /scripts

#vim /scripts/useradd.sh

#!/bin/bash

i=0

u=1000

while ["$i" -lt 500] && ["$u" -ge 1000]

do

   mkdir /rhome

i=$[$i + 1];

   /usr/sbin/useradd pp$i

   /usr/sbin/usermod -s /sbin/nologin pp$i

   /usr/sbin/usermod -u $u pp$i

   /usr/sbin/usermod -d /rhome pp$i

   /usr/sbin/groupadd ppall

   /usr/sbin/usermod -aG ppall pp$i

```
    echo pp$i | passwd    --stdin    pp$i
done
#chmod u+x /scripts/useradd.sh
#/scripts/useradd.sh
```

6、批量删除用户

```
#mkdir /scripts
#vim /scripts/userdel.sh
#!/bin/bash
i=0
while [ "$i" -lt 500 ];
do
    i=$[ $i + 1 ];
    /usr/sbin/userdel -r pp$i
    rm -rf /home/pp$i
done
#chmod u+x /scripts/userdel.sh
#/scripts/userdel.sh
```

二、网络测试

1、getpid1.sh

```
1 #! /bin/bash
2 if [ $# -ne 1 ]; then
3         echo "Usage: $0 PROCESS-NAME"
4         exit 1
5 fi
6
7 pid=$(ps aux | grep $1 | grep -v grep)
8
9 if [ -n "$pid" ];then
10         echo $pid | awk '{ print $2 }'
11 else
12         echo "Can't find this PROCESS"
13 fi
```

2、getip.sh

```
1 #! /bin/bash
2
3 tmp=$(ifconfig eth0 | grep 'inet addr')
4 r=${tmp/inet addr:/}
5 ip=${r/ Bcast*/}
6
7 echo $ip
```

3、#!/bin/bash

```
#alive.sh
for n in {1..30};do
Host=192.168.0.$n
Ping  -c2 $host &>/dev/null

if [ $>=0 ];then
    echo  “$host is up”
else
    echo  “$host is down”
fi
done
```

4、网络状态监控脚本

①邹的

```
#!/bin/bash
HOST=192.168.0.1
while true
do
$(ping -c2 $HOST &> /dev/null)
if [ $? = 0 ]
then
        echo "remote host is UP"
        $(pgrep nfs &> /dev/null)

             if [ $? != 0 ]; then
              service nfs start
             fi
            echo "NFS service is start"
else
```

```
        echo "remote host is DOWN"
        $(pgrep nfs &> /dev/null)
            if [ $? == 0 ]; then
            service nfs stop > /dev/null 2>&1
            fi
            echo "NFS service is stop"
fi
sleep 1
done
```

②我的

```
#! /bin/bash
# chkconfig: 2345 19 99

# description: network monitoring
# chkconfig --add netjk;reboot
echo > /stat.txt

ping -c3 -t1 192.168.0.254 &> /dev/null
  if [ $? != 0 ];then
    echo 'bad' >> /stat.txt &> /dev/null
  fi


pddx=`wc -l /stat.txt | cut -c1`
  if [ $pddx = 3 ];then
    service nfs stop
  else
          $(pgrep nfs &> /dev/null)
           if [ $? == 0 ]; then
           service nfs stop > /dev/null 2>&1
           fi
  fi
```

```
nice -n 19 /etc/init.d/netjk &
```

5、for num in $(seq 1 10)

Do

USER=admin$NUM

useradd $USER

echo redhat|passwd -stdin $USER

done

6、#!/bin/bash

cat <<AAA

my useradd is:$ (id -un)

my id is: $(id -u)

AAAOF

7、#!/bin/bash

For USER in {1..10}

Do

Useradd user$USER

Echo password|passwd -stdin user$USER

Done

```
[root@localhost ~]# cat u.sh
for SZ in {1..10}
do
useradd kxy$SZ
echo $SZ|passwd --stdin kxy$SZ
done
```

8、for users in $(cat userlist)

Do

Useradd $users

Done

三、文件有关

1、backup /etc/sysconfig to datestamped subdir of ~/backups

```
#!/bin/bash
test -d ~/backup||mkdir ~/backups
Cp -a $1 ~/backups/back-$[$(date +%Y%m%d)-1]
#echo “backup of /etc/sysconfig completed at:
$(date)”|mail -s “backup” root
```

2、批量改*.txt 为*.doc

```
for FILE in $(ls /tmp/*.txt)
do
mv $FILE $(ls $FILE|cut -d. -f 1 $FILE).doc
done
```

4、批量创建文件

```
for num in $(seq 1 1000)
do                          //$(seq 1 1000)等价于{1…1000}
touch /tmp/file_num
done
```

5、批量复制文件

```
#! /bin/bash
for i in `seq 1 20`;do
cp ~/install.log a$i.log
done
```

6、统计文件与目录数量

```
#!/bin/sh
FNUM=0
DNUM=0
COUNT=0
ls -al
for FILENAME in `ls -a`
do
        if [ -d $FILENAME ]
          then
                DNUM=`expr $DNUM + 1`
          else
                FNUM=`expr $FNUM + 1`
```

fi
COUNT=`expr $COUNT + 1`
done
echo Directory:$DNUM
echo File:$FNUM
echo Total:$COUNT

7、比较文件是否相同

```
#! /bin/bash
DIFF=/usr/bin/diff
if [ $# -ne 2 ];then
    echo "Usage: $0 file1 file2"
   else
    $DIFF $1 $2 &> /dev/null
    if [ $? -eq 0  ] ; then
        echo "file is same"
       else
        echo "file not same"
    fi
fi
```

8、ERROR：参数行过长

```
#! /bin/bash
#listtoolong.sh to solve "Argument list tool long" error.
shopt -s -o nounset
destdir=/tmp
while read f
do
 echo $f
done < <(find $destdir -name '*.txt')
```

9. 完整备份

#mkdir /scripts
#vim /scripts/fbackup.sh
#!/bin/bash
$1=/
$2=备份名称
$3=设备名
$4=/full
if [-e $1]
then
mount $3 /media
mkdir /media/$4

```
    cp -rpf $1 /media/$4/-fbackup[$(date '+%Y%m%d')].bak
    else
        echo "error"
fi
#chmod u+x /scripts/fbackup.sh
#/scripts/fbackup.sh
#crontab -e
* * 1,16 * * /scripts/fbackup.sh
#service crond restart
#chkconfig --level 35 crond on
```

10、 增量备份

```
#mkdir /scripts
#vim /scripts/backup.sh
#!/bin/bash
    $1=/var
    $2=备份名称
if [ -d /backups ]
    then
    cp -rpf $1 /backups/$(basename $2)-s[$(date -d "day ago'+%Y%m%d')].bak
    else
        mkdir /backups
fi
#chmod u+x /scripts/backup.sh
#/scripts/backup.sh
#crontab -e
* 12-14 2-15,17-31 * * /scripts/backup.sh
#service crond restart
#chkconfig --level 35 crond on
```

四、进程、监控、服务

1、磁盘利用率

```
# df -h|awk -F"%" '{print $1}'|awk '/[0-9][0-9].*/{print $NF}'
```

2、trap '' INT　　//禁用 CTRL+C

```
[root@server1 ~]# kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
root@server1 ~]# trap '' INT
root@server1 ~]# trap -p
rap -- '' SIGINT

[root@server1 ~]# kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
[root@server1 ~]# trap 'echo hello' HUP
[root@server1 ~]# kill -HUP $$
hello
[root@server1 ~]# trap -p
trap -- 'echo hello' SIGHUP
trap -- '' SIGINT
```

3、编写脚本 htmon.sh，用于检测 httpd 服务的状态:

服务状态是失效时在/var/log/htmon.log 中有日志信息；
自动将失常的服务启动；
若重启失败，将尝试重新启动服务器主机；
结合 crontab 计划任务，每周一到周五每隔 15 分钟执行一次检测

```
#! /bin/bash
service httpd status &> /dev/null
if [ $? -ne 0 ]; then
echo "httpd is down. At time: `time` " >> /var/log/htmon.log
service httpd restart
service httpd status &> /dev/null

  if [ $? -ne 0 ];then
  chkconfig --level 35 httpd on
  shutdown -r now
  fi
fi
```

5、编写登陆欢迎脚本

在 root 用户每次登陆到 Shell 环境时运行(~/.bashrc)，报告当前登陆用户数，打开的进程数，剩余可用的内存，剩余交换空间等信息

```
#! /bin/bash
#
shopt -o -s nounset
UserNum=$(w|grep "pts"|wc -l )
PsNum=$(ps -elf|wc -l)
MemNum=$(free -m |grep Mem| awk '{print $4}')
SwapNum=$(free -m|grep Swap|awk '{print $4}')
echo "Welcome `logname` to login to this Server!"
echo "Number of login user:        $UserNum Peoples"
echo "Number of running processes: $PsNum Processes"
echo "Free memory size:            $MemNum MB"
echo "Free swap space size:        $SwapNum MB"
```

6、如何读取所有命令行参数进行显示？

```
declare -i i
for a
do
 i=i+1
 echo $i parameter is $a
done
```

7、 ps 进程

```
#mkdir /scripts
#vim /scripts/ps.sh
#!/bin/bash
   ps axo %cpu,%mem |sort -r |head -n 6 |mail -s "cpu" root
#chmod u+x /scripts/ps.sh
#/scripts/ps.sh
#crontab -e
*/10 * * * * /scripts/ps.sh
#service crond restart
#chkconfig --level 35 crond on
```

8、拒绝匿名访问 ftp 服务

```
#mkdir /scripts
#vim /scripts/vsftpd.sh
#!/bin/bash
i=4
while [ "$i" -le 5 ];
do
   i=$[ $i + 1 ]
   if [ -f /etc/vsftpd/vsftpd.conf ]
   then
        groupadd ftpd
        groupmod -g 1000 ftpd
        useradd user$i
        usermod -a -G ftpd user$i
        usermod -s /sbin/nologin user$i
        echo user$i |passwd --stdin user$i
        sed -i.bak '12s/YES/NO/g' /etc/vsftpd/vsftpd.conf
        echo userlist_deny=NO >> /etc/vsftpd/vsftpd.conf
```

```
        echo user$i >> /etc/vsftpd/user_list
        /bin/touch /home/user$i/aaa.txt
        /bin/touch /home/user$i/bbb.sh
        /usr/sbin/setenforce 1
        /usr/sbin/setsebool -P ftp_home_dir on
        /sbin/service vsftpd restart
        /sbin/chkconfig --level 35 vsftpd on
        /sbin/service iptables stop
        /sbin/iptables -F
        /sbin/service iptables save
        /sbin/chkconfig --level 35 iptables on
        echo "ok" | mail -s "server is ok" user5@desktop15.example.com
else
        /usr/bin/yum install vsftpd lftp -y
        echo "error" | mail -s "server is error" user5@desktop15.example.com
fi
done
#chmod u+x /scripts/vsftpd.sh
#/scripts/vsftpd.sh
```

9、ftp 下载流量限速

```
#mkdir /scripts
#vim /scripts/rate.sh
#!/bin/bash
i=200
while [ "$i" -le 201 ];
do
   i=$[ $i + 1 ]
if [ -f /etc/vsftpd/vsftpd.conf ]
   then
        groupadd ftpd
        groupmod -g 1000 ftpd
        useradd user$i
        usermod -a -G ftpd user$i
        usermod -s /sbin/nologin user$i
        echo user$i |passwd --stdin user$i
        sed -i.bak '12s/YES/NO/g' /etc/vsftpd/vsftpd.conf
        sed -i.bak '96,98s/^#//g' /etc/vsftpd/vsftpd.conf
```

```
        sed -i.bak '118s/YES/NO/g' /etc/vsftpd/vsftpd.conf
        touch /etc/vsftpd/chroot_list
        echo user$i >> /etc/vsftpd/chroot_list
        mkdir /ftprate
        /usr/bin/chcon -R -t public_content_t /ftprate
        echo local_root=/ftprate >> /etc/vsftpd/vsftpd.conf
        echo userlist_deny=NO >> /etc/vsftpd/vsftpd.conf
        echo user$i >> /etc/vsftpd/user_list
        echo user_config_dir=/var/users >> /etc/vsftpd/vsftpd.conf
        mkdir /var/users
        /bin/cp /etc/vsftpd/vsftpd.conf /var/users/user$i
        echo local_max_rate=100 >> /var/users/user201
        echo local_max_rate=5120 >> /var/users/user202
        /bin/dd if=/dev/zero of=/ftprate/sysccp.txt bs=1M count=200
        /bin/touch /home/user$i/aaa.txt
        /bin/touch /home/user$i/bbb.sh
        /sbin/service vsftpd restart
        /sbin/chkconfig --level 35 vsftpd on
        /sbin/service iptables stop
        /sbin/iptables -F
        /sbin/service iptables save
        /sbin/chkconfig --level 35 iptables on
        echo "ok" | mail -s "server is ok" user5@desktop15.example.com
else
        /usr/bin/yum install vsftpd lftp -y
        echo "error" | mail -s "server is error" user5@desktop15.example.com
fi
done
#chmod u+x /scripts/rate.sh
#/scripts/rate.sh
```

五、算术运算

1、read+expr+echo

```
#vim    jia.sh
echo “Please enter a value for a”
read a
```

```
echo “Please enter a value for b”
read b
echo “a+b=`expr $a + $b`”
rem c=$((a+b))
rem echo $c
```

```
[root@instructor ~]# bash a.sh
please enter a value for a
123
please enter a value for b
456
a+b= 579
```

2、九九乘法表

```
#!/bin/sh
for ((ROW=1;ROW<10;ROW++))
do
        for ((COL=1;COL<=$ROW;COL++))
        do
            echo -ne " $ROW x $COL= "`expr $ROW \* $COL`
        done
        echo
done
```

3、假的彩票：显示结果

```
#mkdir /scripts
#vim /scripts/script.sh
#!/bin/bash
if [ "$1" = all ];
   then
        echo "none"
elif [ "$1" = none ];
   then
        echo "all"
else
        echo "error all | none;"
fi
```

```
#chmod u+x /scripts/script.sh
#/scripts/script.sh
```

第 18 讲 据日志统计排名前几的 IP

维护的一个应用，统计 PV/UV 的时候，发现有天的访问超出其他天一大截，感觉是被攻击了，或者是被爬虫抓取。因此需要统计一些 ip，让 PE 分析一下这些攻击 ip 为何被封掉。

根据访问日志统计排名前几的访问 ip，一条 AWK 便能完成：

```
cat xxx-access_log |awk '{++S[$3]}  END{for(a in S) if(S[a] > 2000) print a,S[a]}' |sort -nr -k2
```

解释一下：ip 是每行的第三个字段（以空格分隔），根据第三个字段构造一个数组，最后将数组输出，输出时，根据第二个字段做个倒排序。这样便能得到访问量排名前几的 ip

第 19 讲 Netstat+awk 统计网络连接数

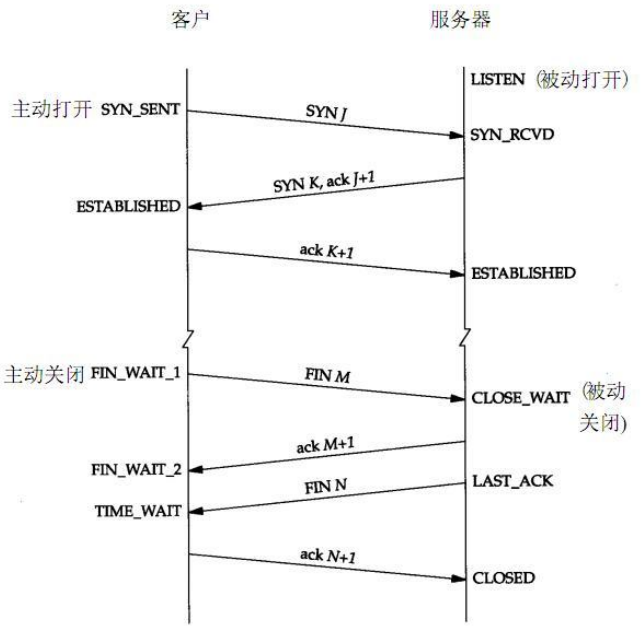
```
netstat -an  |  awk '{print $1,$6}' | sort -r -k1 | uniq -c
```

一、TCP 连接的建立与终止

建立一个连接需要三次握手，而终止一个连接要经 4 次握手。正常情况下的 TCP 状态迁移：

客户端	CLOSED->SYN_SENT->ESTABLISHED->FIN_WAIT_1->FIN_WAIT_2->TIME_WAIT->CLOSED
服务器	CLOSED->LISTEN->SYN_RECV->ESTABLISHED->CLOSE_WAIT->LAST_ACK->CLOSED

CLOSED 状态不是一个真正的状态，而是这个状态迁移的假想起点和终点。



CLOSED	无连接是活动的或正在进行
LISTEN	服务器在等待进入呼叫
SYN_RECV	一个连接请求已经到达，等待确认

SYN_SENT	应用已经开始，打开一个连接
ESTABLISHED	正常数据传输状态
FIN_WAIT1	应用说它已经完成
FIN_WAIT2	另一边已同意释放
ITMED_WAIT	等待所有分组死掉
CLOSING	两边同时尝试关闭
TIME_WAIT	另一边已初始化一个释放
LAST_ACK	等待所有分组死掉

也就是说，这条命令可把当前系统的网络连接状态分类汇总。

二、netstat+awk 会得到类似下面的结果：

```
netstat -n | awk '/^tcp/ {++state[$NF]} END {for(key in state) print key,"\t",state[key]}'
LAST_ACK 1
SYN_RECV 14
ESTABLISHED 79
FIN_WAIT1 28
FIN_WAIT2 3
CLOSING 5
TIME_WAIT 1669
```

三、下面解释一下为啥要这样写

1、netstat -n

```
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address Foreign Address State
tcp 0 0 123.123.123.123:80 234.234.234.234:12345 TIME_WAIT
```

你实际执行这条命令的时候，可能会得到成千上万条类似上面的记录，不过我们就拿其中的一条就足够了。

2、再来看看 awk

/^tcp/	滤出 tcp 开头的记录，屏蔽 udp, socket 等无关记录。
state[]	相当于定义了一个名叫 state 的数组
NF	表示记录的字段数，如上所示的记录，NF 等于 6
$NF	表示某个字段的值，如上所示的记录，$NF 也就是$6，表示第 6 个字段的值，也就是 TIME_WAIT
state[$NF]	表示数组元素的值，如上所示的记录，就是 state[TIME_WAIT]状态的连接数
++state[$NF]	表示把某个数加一，如上所示的记录，就是把 state[TIME_WAIT]状态的连接数加一
END	表示在最后阶段要执行的命令
for(key in state)	遍历数组
print key,"\t",state[key]	打印数组的键和值，中间用\t 制表符分割，美化一下。

如发现系统存在大量 TIME_WAIT 状态的连接，通过调整内核参数解决，vim /etc/sysctl.conf 编辑文件，加入以下内容：

1.net.ipv4.tcp_syncookies = 1

2.net.ipv4.tcp_tw_reuse = 1

3.net.ipv4.tcp_tw_recycle = 1

4.net.ipv4.tcp_fin_timeout = 30

然后执行 /sbin/sysctl -p 让参数生效。

net.ipv4.tcp_syncookies = 1 表示开启 SYN Cookies。当出现 SYN 等待队列溢出时，启用 cookies 来处理，可防范少量 SYN 攻击，默认为 0，表示关闭；

net.ipv4.tcp_tw_reuse = 1 表示开启重用。允许将 TIME-WAIT sockets 重新用于新的 TCP 连接，默认为 0，表示关闭；

net.ipv4.tcp_tw_recycle = 1 表示开启 TCP 连接中 TIME-WAIT sockets 的快速回收，默认为 0，表示关闭。

net.ipv4.tcp_fin_timeout 修改系統默认的 TIMEOUT 时间

下面附上 TIME_WAIT 状态的意义：

客户端与服务器端建立 TCP/IP 连接后关闭 SOCKET 后，服务器端连接的端口状态为 TIME_WAIT 是不是所有执行主动关闭的 socket 都会进入 TIME_WAIT 状态呢？
有没有什么情况使主动关闭的 socket 直接进入 CLOSED 状态呢？

主动关闭的一方在发送最后一个 ack 后就会进入 TIME_WAIT 状态 停留 2MSL（max segment lifetime）时间这个是 TCP/IP 必不可少的，也就是“解决”不了的。

也就是 TCP/IP 设计者本来是这么设计的主要有两个原因
1。防止上一次连接中的包，迷路后重新出现，影响新连接
（经过 2MSL，上一次连接中所有的重复包都会消失）
2。可靠的关闭 TCP 连接
在主动关闭方发送的最后一个 ack(fin) ，有可能丢失，这时被动方会重新发 fin, 如果这时主动方处于 CLOSED 状态 ，就会响应 rst 而不是 ack。所以主动方要处于 TIME_WAIT 状态，而不能是 CLOSED 。

TIME_WAIT 并不会占用很大资源的，除非受到攻击。

还有，如果一方 send 或 recv 超时，就会直接进入 CLOSED 状态 netstat -an | grep SYN | awk '{print $5}' | awk -F:'{print $1}' | sort | uniq -c | sort -nr | more

netstat -tna | cut -b 49- |grep TIME_WAIT | sort
取出目前所有 TIME_WAIT 的连接 IP （ 排序过 ）

net.ipv4.tcp_syncookies = 1 表示开启 SYN Cookies。当出现 SYN 等待队列溢出时，启用 cookies 来处理，可防范少量 SYN

攻击，默认为 0，表示关闭；

net.ipv4.tcp_tw_reuse = 1 表示开启重用。允许将 TIME-WAIT sockets 重新用于新的 TCP 连接，默认为 0，表示关闭；

net.ipv4.tcp_tw_recycle = 1 表示开启 TCP 连接中 TIME-WAIT sockets 的快速回收，默认为 0，表示关闭。

net.ipv4.tcp_fin_timeout = 30 表示如果套接字由本端要求关闭，这个参数决定了它保持在 FIN-WAIT-2 状态的时间。

net.ipv4.tcp_keepalive_time = 1200 表示当 keepalive 起用的时候，TCP 发送 keepalive 消息的频度。缺省是 2 小时，改为 20 分钟。

net.ipv4.ip_local_port_range = 1024 65000 表示用于向外连接的端口范围。缺省情况下很小：32768 到 61000，改为 1024 到 65000。

net.ipv4.tcp_max_syn_backlog = 8192 表示 SYN 队列的长度，默认为 1024，加大队列长度为 8192，可容纳更多等待连接的网络连接数。

net.ipv4.tcp_max_tw_buckets = 5000 表示系统同时保持 TIME_WAIT 套接字的最大数量，如果超过这个数字，TIME_WAIT 套接字将立刻被清除并打印警告信息。默认为 180000，改为 5000。对于 Apache、Nginx 等服务器，上几行的参数可很好地减少 TIME_WAIT 套接字数量，但是对于 Squid，效果却不大。此项参数可控制 TIME_WAIT 套接字的最大数量，避免 Squid 服务器被大量的 TIME_WAIT 套接字拖死。

awk 实用程序

一种强大的解释性的编程语言

可用作过滤或操纵文本

可用于处理格式化的文本文件

常与 sed 配合使用

语法: awk [-F char] [-f file | program] [files…]

awk 是 Aho, Weinberger 和 Kernighan 三位早期 UNIX 的作者

awk 通常硬连接到新的 awk, 即 nawk

awk 是高级系统管理员必须学习的!

$1====第 1 列,$0 表示整个记录

```
ifconfig | grep ^[a-z] | awk '{print $1}'
cat /etc/passwd | awk –F: '{print $1}'
awk –F: '{print $1}' /etc/passwd

awk 'BEGIN {print "I am counting"}
        {for (I=1;I<=NF;I++) words[$I] += 1}
        END {for (w in words) print w ":" words[w]}' datafile

grep awk /etc/rc0.d/*
```

6.3.4 条件语句

Shell 具有一般高级语言所具有的控制结构，如 if 语句、case 语句。

1. if 语句

if 语句可根据表达式的值是真或假来决定要执行的程序段落。

```
if expressionl                    //若 expressionl 为真
then
```

```
commands                          //则执行这些命令
elif expression2                 //否则若 expression2 为真
then
commands                          //则执行这些命令
else                              //若以上的表达式都不成立
commands                          //则执行这些命令
fi                                //结束 if 语句
fi                                //结束 if 语句
```

例 1：将显示目录内是否有 example_if 文件。

```
#!/bin/bash
if [ -f example_if ]                    //判断文件是否存在
then
echo "There is a example_if file in current directory. "
else
echo "no example_if file in current directory. "
fi
#end
```

例 2：在键盘上读取一个字符，然后根据字符的值来判断对错。

```
#!/bin/bash
echo  -n "Please input the answer: "   //-n 不换行
read I                                    //从键盘读入数
据
if [ $I = y ]
then
echo "The answer is right"
elif [ $I = n ]
then
echo "The answer is wrong"
else
echo "Bad Input."
```

```
fi
#end
```

例 3：从键盘一次读入多个变量。

用语句 read var1 var2 ……变量之间用空格隔开。

```
read name gender                    //读入两个变量
echo $name
echo $gender
```

假如在键盘输入 linux man，则上边的语句执行的结果为

```
linux                               //变量 name 的值
man                                 //变量 gender 的值
```

如果输入文本过长，Shell 将所有的超长的部分赋给最后一个变量。

2. case 语句

case 语句用来从很多的测试条件中选择符合的条件执行。

```
case string in                      //测试 string 字符串
str1)                               //若 str1 符合
commands;;                          //则执行这些命令
str2)
commands;;
*)                                  //若 str1 和 str2 都不符合
commands;;                          //则执行这些命令
esac                                //结束 case 语句
```

例：检查命令行的第一个参数是“-i”或“-e”，如果是“-i”，则计算由第二个参数指定的文件中以 i 开头的行数；如果是“-e”，则计算由第二个参数指定的文件中以 e 开头的行数。如果第一个参数既不是“-i”也不是“-e”，则在屏幕上显示一条错误的信息。其中“^”符号为匹配行首的符号。

假如 aaaa 文件为包含 8 行以 i 开头和 18 行以 e 开头的文本文件，则检索该文件程序如下：

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

```
case $1 in                          //$1 是命令行第一个参数
-i )
count='grep ^i $2|wc -l'      //查找并计算以 i 开头的行数,“`”是反引号
echo "The number of lines in $2 that start with an i is $count."
;;
-e )
count='grep ^e $2|wc -l'
echo "The number of lines in $2 that start with an e is $count."
;;
* )                                  //默认匹配
echo "That option is not recognized. "
;;
esac                                //和 case 成对出现
#end
```

将其存为“a1”，将其属性修改为可执行 chmod a+x a1。执行程序“a1”。

```
$ ./a1 -i aaaa
The number of lines in file1 that start with an i is 8.
$ ./a1 -e aaaa
The number of lines in file1 that start with an i is 18.
```

6.3.5 循环命令

Shell 中提供了几种执行循环的命令，比较常见的命令有 for、while、until、shift 命令。

1. for 语句

for 语句有两种格式。第一种格式：

```
for var in list
do
commands
```

done

第二种格式：

for var

do

statements

done

使用这种形式时，对变量 var 中的每一项，for 语句都执行一次。此时 Shell 程序假定 var 包含 Shell 程序在命令行的所有位置参数。所以此种方式也可以写成：

for var in "$@"

do

statements

done

例 1：使用通配符显示当前目录下所有文本文件（*.txt）的名称和内容。

#!/bin/bash

for file in *.txt　　//对目录下的每个 txt 文件

do

echo $file　　//显示文件名

cat $file　　//显示文件内容

done

#end

例 2：把表中的几个值显示出来。

#! /bin/bash

for p in 1 2 3 4 5

do

echo $p

done

#end

例 3：求命令行所有整数之和。

```
#! /bin/bash
sum=0
for p in $*
do
let sum=$sum+$p
done
echo “sum=  $sum”
#end
```

2. while 及 until 语句

while 语句与 until 语句的语法结构和用途相似。while 语句会在测试条件为真时循环执行，语法如下：

```
while expression
do
commands
done
```

而 until 语句会在其测试条件为假时循环执行，语法为

```
until expression
do
commands
done
```

例：编程计算 1+2+3+4+5 的值。

```
#! /bin/bash
let s=0; p=1
while test $p  -le 5
do
let s=$s+$p
let p=$p+1
done
echo ″s=$s″
```

#end

3. shift 命令

shift 命令用来将命令行参数左移。假设当前的命令行有 3 个参数，分别是：$1=-r $2=file1 $3=file2

则在执行 shift 命令之后，其值会变成：$1=file1 $2=file2

shift 命令也可以指定参数左移的位数，下面的命令将使命令行参数左移 2 个位置：

shift 2

shift 命令常与 while 语句或 until 语句一起使用。

```
列出所有参数及其位置号。
count=1
while [ -n "$*" ]            //*表示程序的所有参数
do
echo "This is parameter number $count $1"
shift                          //将命令行参数左移一位
let count=$count+1
done
```

```
count=0
while [ $# != 0 ]
do
 count=$(($count + $1))
 #count=`expr $count + $1`
 shift
done
echo count is $count
```

4. break 和 continue 语句

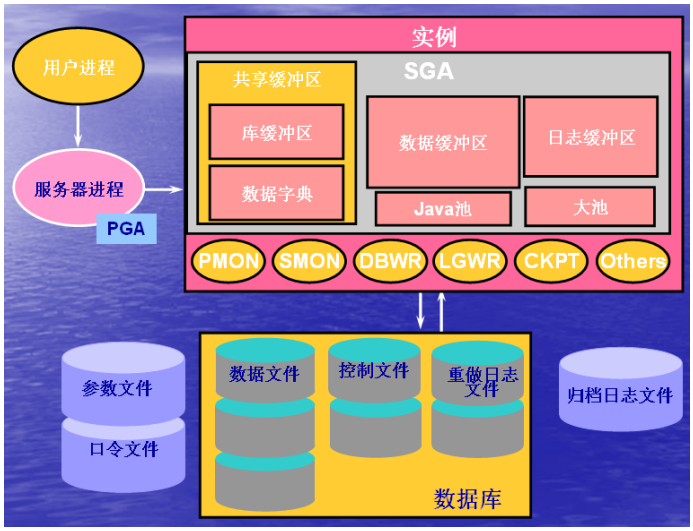
在 Shell 的 for、while、until 循环语句中,也可以使用如 C 语言的 break 和 continue 语句以跳离现有的循环。break 语句用于中断循环的执行，将程序流程移至循环语句结束之后的下一个

命令。而 continue 语句则忽略之后的命令，将程序流程转移至循环开始的地方。break 和 continue 语句都可以加上数字，以指示要跳出的循环数目。

例：观察程序执行结果。

```
#!/bin/bash
for x in 1 2 3 4
do
echo "hello"
continue
echo "shell"
done
#end
```

UNIT39 Oracle auto install



一、准备工作

#1.完全安装 rhel5.5;

#装安装包，插入光盘

#mount /dev/cdrom /mnt/cdrom

#cd /mnt/Server

#rpm -ivhp make* gcc* glibc* compat-db* compat-gcc* compat-gcc-c++* compat-libstdc++* compat-libstdc++-devel* openmotif21* setarch*

#rpm -Uvh compat-db-4*

#rpm -Uvh libaio-0*

#rpm -Uvh compat-libstdc++-33-3*

#rpm -Uvh compat-gcc-34-3*

#rpm -Uvh compat-gcc-34-c++-3*

#rpm -Uvh libXp-1*

#rpm -Uvh openmotif-2*

```
#rpm -Uvh gcc-4
#2.在 xp 启动 xmanager/xshell,接着用 root 连接到 linux(使用 putty 不方便粘贴)
#3.运行 gnome-terminal &
gnome-terminal &
#######################################
```

二、系统设置(粘到 shell)

```
#以下 xmanager/xshell 中的 gnome-termial 中操作粘贴
test -f /bin/yutian.sh && exit

hostname yutianedu
sed '/HOSTNAME/  s/^/#/' /etc/sysconfig/network >> /etc/sysconfig/network
echo 'HOSTNAME=yutianedu' >> /etc/sysconfig/network

#####解决 oracle GUI 安装画面乱码问######
echo 'LANG="en_US.UTF-8"'  > /etc/sysconfig/i18n

########################################
service network restart
IPDZ=`ifconfig eth0|grep "inet addr"|cut -d: -f2 |cut -d" " -f1`
echo "$IPDZ yutianedu  oracle"
sleep 2
echo "$IPDZ yutianedu  oracle" >> /etc/hosts

########################################

cat << oracleok  > /bin/yutian.sh
#!/bin/bash
```

```
########################################
groupadd oinstall
groupadd dba
useradd -g oinstall -G dba  oracle
echo oracle|passwd --stdin oracle

########################################
cat << oracle1  >> /etc/sysctl.conf

# for oracle powered by zcs
kernel.shmall = 2097152
kernel.shmmax = 2147483648
kernel.shmmni = 4096
kernel.sem = 250 32000 100 128
fs.file-max = 65536
net.ipv4.ip_local_port_range = 1024 65000
net.core.rmem_default = 1048576
net.core.rmem_max = 1048576
net.core.wmem_default = 262144
net.core.wmem_max = 262144
oracle1

sysctl -p

#####################################
cat << oracle2  >> /etc/security/limits.conf
# for oracle powered by zcs
oracle              soft    nproc    2047
oracle              hard    nproc    16384
```

```
oracle              soft    nofile  1024
oracle              hard    nofile  65536
oracle2

#####################################
cat << oracle3  >> /etc/pam.d/login

# for oracle powered by zcs
session    required     /lib/security/pam_limits.so
session    required     pam_limits.so
oracle3

#####################################
cat << oracle4  >> /etc/profile
# for oracle powered by zcs
if [ $USER = "oracle" ]; then
        if [ $SHELL = "/bin/ksh" ]; then
              ulimit -p 16384
              ulimit -n 65536
        else
              ulimit -u 16384 -n 65536
        fi
fi
oracle4
#####################################
mkdir -p  /u01/app/oracle
chown -R oracle:oinstall /u01
chmod -R  775  /u01

#####################################
```

```
cat << oracle5 >> /home/oracle/.bash_profile

# for oracle powered by zcs
export ORACLE_BASE=/u01/app/oracle
export ORACLE_HOME=/u01/app/oracle/product/10.2.0/db_1
export ORACLE_SID=orcl
export PATH=$ORACLE_HOME/bin:$PATH
oracle5

#######################################
oracleok

chmod +x /bin/yutian.sh
yutian.sh
#######################################
```

三、上传 oracle.zip 到 linux

#用 oracle 将 0201_database_linux32.zip 上传到/u01

四、改为 xp 的 IP 后粘到 shell 中

```
# xshell/gnome-terninal 将脚本粘到 [root@yutianedu] #下
su - oracle
export DISPLAY=192.168.2.1:0.0
xhost +
cd /u01
#wget http://192.168.0.254/10201_database_linux32.zip
#rm -rf database
unzip 10201_database_linux32.zip
cd database
./runInstaller
```

五、创建侦听+建库

```
###################创建侦听#################
#在 xshell 的 gnome-terminal 中表建终端
su - oracle
export DISPLAY=192.168.2.1:0.0
echo 'export PATH=$ORACLE_HOME/bin:$PATH' >>
~/.bash_profile
export PATH=$ORACLE_HOME/bin:$PATH
netca

###################建库#################
dbca
#####################################
sqlplus / as sysdba
```