一步一步学Linux

极客学院出版

前言

Linux是当今一门赤手可热的技术，很多IT民工都向往成为一名Linux工程师。该专题是一个非常详细，非常全面的Linux教程，并且还在持续更新中，关注它会让你更方便更全面的学习Linux。

读者

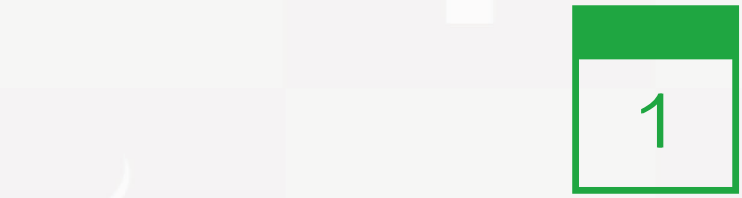
本教程是初级教程，旨在帮助需要使用 Linux 操作系统的程序开发者或多 Linnx 系统感兴趣的技术爱好者。

预备知识

学习本教程需要你了解目前主流的操作系统，对 shell 脚本的编写有一定的了解。

更新日期	更新内容
2015-07-31	一步一步学 Linux

目录



1 文件系统简介

对于 Linux 来说正规的文件系统为 EXT2，一个文件系统存储的数据通常包括文件权限和属性，以及文件数据，这两部分分别存储在不同的地方。

简单的说文件系统数据分为3部分 Superblock:记录次文件系统的整体信息，包括 inode/iblock 的总量，使用量，剩余量，已经文件系统的整体信息 **Inode**:记录文件的属性，一个文件占用一个 inode,同时记录此文件数据所在的 block 号码 **Iblock**:实际记录文件的内容，若文件过大时，会占用多个 block

我的理解 文件系统由包含多个文件，每个文件都会占用1个 inode 和若干 iblock,inode 用来存储文件权限和属性.以及文件数据存放的 iblock 的编号，iblock 则存放文件的实际数据（每个 inode 和 iblock都含有一个编号）

对于容量很大的文件系统，EXT2格式化时会分为多个 block group,每个组队都有一个独立的/inode/block/superblock

iblock (数据区块)

iblock 是用来放置文件内容数据地方，在 Ext2 文件系统中所支持的 block 大小有 1 K, 2 K 及 4 K 三种而已。每个 block 内最多只能够放置一个文件的数据； 承上，如果文件大于 block 的大小，则一个文件会占用多个 block 数量； 承上，若文件小于 block ，则该 block 的剩余容量就不能够再被使用了(磁盘空间会浪费)。

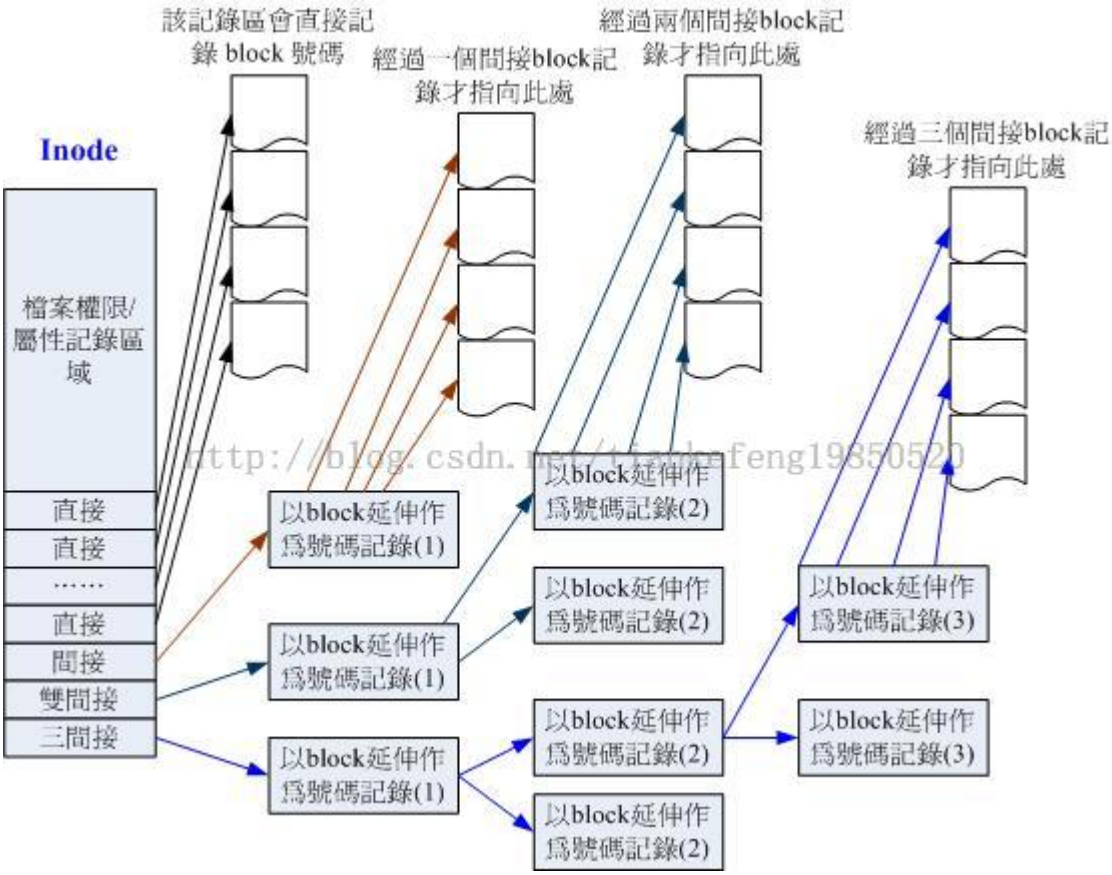
Inode

Inode 主要包含信息 1. 该文件的存取模式(read/write/excute)； 2. 该文件的拥有者与群组(owner/group)； 3. 该文件的容量； 4. 该文件创建或状态改变的时间(ctime)； 5. 最近一次的读取时间(atime)； 6. 最近修改的时间(mtime)； 7. 定义文件特性的旗标(flag)，如 SetUID...； 8. 该文件真正内容的指向(pointer)； 9. 其他信息

每个 inode 大小均固定为 128 bytes； 每个文件都仅会占用一个 inode 而已； 承上，因此文件系统能够创建的文件数量与 inode 的数量有关； **系统读取文件时需要先找到 inode，并分析 inode 所记录的权限与用户是否符合，若符合才能够开始实际读取 block 的内容。**

Inode 三级间接索引

inode 要记录的数据非常多，但偏偏又只有128 bytes 而已，为了记录更多的数据，系统很将 inode 记录 block 号码的区域定义为12个直接，一个间接, 一个双间接与一个三间接记录区。记录区。



假设 block 为1 K

12 个直接指向： 12*1 K=12 K 由于是直接指向，所以总共可记录 12 笔记录，因此总额大小为如上所示； 间接：256*1 K=256 K 每笔 block 号码的记录会花去 4 bytes，因此 1 K 的大小能够记录 256 笔记录，因此一个间接可以记录的文件大小如上； 双间接： 256*256*1 K=2562 K 第一层 block 会指定 256 个第二层，每个第二层可以指定 256 个号码，因此总额大小如上； 三间接： 256*256*256*1 K=2563 K 第一层 block 会指定 256 个第二层，每个第二层可以指定 256 个第三层，每个第三层可以指定 256 个号码，因此总额大小如上； 总额：将直接、间接、双间接、三间接加总，得到 12 + 256 + 256*256 + 256*256*256 (K) = 16 GB

Superblock (超级区块)

Superblock 是记录整个 filesystem 相关信息的地方， 没有 Superblock，就没有这个 filesystem 了。他记录的信息主要有： 1. block 与 inode 的总量； 2. 未使用与已使用的 inode / block 数量； block 与 inode 的大小 (block 为 1, 2, 4 K，inode 为 128 bytes)； filesystem 的挂载时间、最近一次写入数据的时间、最近一次检验磁盘 (fsck) 的时间等文件系统的相关信息； 一个 valid bit 数值，若此文件系统已被挂载，则 valid bit 为 0 ，若未被挂载，则 valid bit 为 1 。

Superblock 信息查看

通过 dumpe2fs 命令查看文件系统的 superblock

举例：观察文件系统的相关信息

```
[root@www ~]# df   <==这个命令可以叫出目前挂载的装置
Filesystem    1K-blocks      Used Available Use% Mounted on
/dev/hdc2      9920624   3822848   5585708  41% /
/dev/hdc3      4956316    141376   4559108   4% /home
/dev/hdc1       101086     11126     84741  12% /boot
tmpfs           371332         0    371332   0% /dev/shm

[root@www ~]# dumpe2fs /dev/hdc2
dumpe2fs 1.39 (29-May-2006)
Filesystem volume name:   /1            <==这个是文件系统的名称(Label)
Filesystem features:      has_journal ext_attr resize_inode dir_index
 filetype needs_recovery sparse_super large_file
Default mount options:    user_xattr acl <==默认挂载的参数
Filesystem state:         clean         <==这个文件系统是没问题的(clean)
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              2560864       <==inode 的总数
Block count:              2560359       <==block 的总数
Free blocks:              1524760       <==还有多少个 block 可用
Free inodes:              2411225       <==还有多少个 inode 可用
First block:              0
Block size:               4096          <==每个 block 的大小啦!
Filesystem created:       Fri Sep  5 01:49:20 2008
Last mount time:          Mon Sep 22 12:09:30 2008
Last write time:          Mon Sep 22 12:09:30 2008
```

```
Last checked:             Fri Sep  5 01:49:20 2008
First inode:              11
Inode size:               128           <==每个 inode 的大小
Journal inode:            8
Journal backup:           inode blocks
Journal size:             128M
Group 0: (Blocks 0-32767) <==第一个 data group 内容, 包含 block 的启始/结束号码
Primary superblock at 0, Group descriptors at 1-1  <==超级区块在 0 号 block
Reserved GDT blocks at 2-626
Block bitmap at 627 (+627), Inode bitmap at 628 (+628)
Inode table at 629-1641 (+629)                  <==inode table 所在的 block 0 free blocks, 32405 free inodes, 2 directories    <=
Free blocks:
Free inodes: 12-32416                           <==剩余未使用的 inode 号码
Group 1: (Blocks 32768-65535)
....(底下省略)....
```

目录与文件读取

在文件系统下创建一个文件都会分配一个 inode 和若干 iblock,目录和文件都是文件系统下的一个文件，

对于目录来说

inode ： 记录了目录的权限， block：记录了目录的名称，目录下包含的文件名名与文件名所占用的 inode 号码

举例：查看 Inode

[root@bogon ~]# ls -li 2366420 -rw------- 1 root root 1377 02-18 20:26 anaconda-ks.cfg 2366454 drwxr-xr-x 2 root root 4096 02-18 20:41 Desktop 2366370 -rw-r--r-- 1 root root 35014 02-1820:25 install.log 2366371 -rw-r--r-- 1 root root 6431 02-18 20:22 install.log.syslog

对于文件来说

文件名：在包含其的目录对应的 iblock 中记录 inode ： 记录了文件的权限， block：记录文件实际数据

举例来说，如果我想要读取/etc/passwd 这个文件时，系统是如何读取的呢?

[root@bogon ~]# ll -di / /etc /etc/passwd 2 drwxr-xr-x 24 root root 4096 02-22 19:34 / 1134561 drwx r-xr-x 114 root root 1228802-22 19:36 /etc 2528506 -rw-r--r-- 1 root root 2219 02-18 20:39 /etc/pa sswd

1. / 的 inode： 透过挂载点的信息找到 /dev/hdc2 的 inode 号码为 2 的根目录 inode，且 inode 规范的权限让我们可以读取该 block 的内容(有 r 与 x)；
2. / 的 block： 经过上个步骤取得 block 的号码，并找到该内容有 etc/ 目录的 inode 号码 (1912545)；
3. etc/ 的 inode： 读取 1912545 号 inode 得知 vbird 具有 r 与 x 的权限，因此可以读取 etc/ 的 block内容；
4. etc/ 的 block： 经过上个步骤取得 block 号码，并找到该内容有 passwd 文件的 inode 号码 (1914888)；
5. passwd 的 inode： 读取 1914888 号 inode 得知 vbird 具有 r 的权限，因此可以读取 passwd 的 block 内容；
6. passwd 的 block： 最后将该 block 内容的数据读出来

2

Linux 命令缩写

- ls：list(列出目录内容)
- cd：Change Directory（改变目录）
- su:switch user 切换用户
- rpm:redhat package manager 红帽子打包管理器
- pwd:print work directory 打印当前目录显示出当前工作目录的绝对路径
- ps: process status(进程状态，类似于 windows 的任务管理器)
- 常用参数：－auxf
  ps －auxf 显示进程状态
- df: disk free 其功能是显示磁盘可用空间数目信息及空间结点信息。换句话说，就是报告在任何安装的设备或目录中，还剩多少自由的空间。
- rpm：即 RedHat Package Management，是 RedHat 的发明之一
- rmdir：Remove Directory（删除目录）
- rm：Remove（删除目录或文件）
- cat: concatenate 连锁 cat file1 file2>>file3把文件1和文件2的内容联合起来放到 file3中
- insmod: install module,载入模块
- ln －s : link －soft 创建一个软链接，相当于创建一个快捷方式
- mkdir：Make Directory(创建目录
  touch
- man: Manual
- pwd：Print working directory
- su：Swith user
- cd：Change directory
- ls：List files
- ps：Process Status
- mkdir：Make directory
- rmdir：Remove directory
- mkfs: Make file system
- fsck：File system check

- cat: Concatenate
- uname: Unix name
- df: Disk free
- du: Disk usage
- lsmod: List modules
- mv: Move file
- rm: Remove file
- cp: Copy file
- ln: Link files
- fg: Foreground
- bg: Background
- chown: Change owner
- chgrp: Change group
- chmod: Change mode
- umount: Unmount
- dd: 本来应根据其功能描述“Convert an copy”命名为“cc”，但“cc”已经被用以代表“C Complier”，所以命名为“dd”
- tar：Tape archive
- ldd：List dynamic dependencies
- insmod：Install module
- rmmod：Remove module
- lsmod：List module
  文件结尾的"rc"（如.bashrc、.xinitrc 等）：Resource configuration
  Knnxxx / Snnxxx（位于 rcx.d 目录下）：K（Kill）；S(Service)；nn（执行顺序号）；xxx（服务标识）
- .a（扩展名 a）：Archive，static library
- .so（扩展名 so）：Shared object，dynamically linked library
- .o（扩展名 o）：Object file，complied result of C/C++ source file
- RPM：Red hat package manager
- dpkg：Debian package manager

- apt：Advanced package tool（Debian 或基于 Debian 的发行版中提供）
- bin = BINaries
- /dev = DEVices
- /etc = ETCetera
- /lib = LIBrary
- /proc = PROCesses
- /sbin = Superuser BINaries
- /tmp = TeMPorary
- /usr = Unix Shared Resources
- /var = VARiable ?
- FIFO = First In, First Out
- GRUB = GRand Unified Bootloader
- IFS = Internal Field Seperators
- LILO = LInux LOader
- MySQL = My 是最初作者女儿的名字，SQL = Structured Query Language
- PHP = Personal Home Page Tools = PHP Hypertext Preprocessor
- PS = Prompt String
- Perl = "Pratical Extraction and Report Language" = "Pathologically Eclectic Rubbish Lister" Python 得名于电视剧 Monty Python's Flying Circus
- Tcl = Tool Command Language
- Tk = ToolKit
- VT = Video Terminal
- YaST = Yet Another Setup Tool
- apache = "a patchy" server
- apt = Advanced Packaging Tool
- ar = archiver
- as = assembler
- awk = "Aho Weiberger and Kernighan" 三个作者的姓的第一个字母

- bash = Bourne Again SHell
- bc = Basic (Better) Calculator
- bg = BackGround
- biff = 作者 Heidi Stettner 在 U.C.Berkely 养的一条狗,喜欢对邮递员汪汪叫。
- cal = CALendar
- cat = CATenate
- cd = Change Directory
- chgrp = CHange GRouP
- chmod = CHange MODe
- chown = CHange OWNer
- chsh = CHange SHell
- cmp = compare
- cobra = Common Object Request Broker Architecture
- comm = common
- cp = CoPy
- cpio = CoPy In and Out
- cpp = C Pre Processor
- cron = Chronos 希腊文时间
- cups = Common Unix Printing System
- cvs = Current Version System
- daemon = Disk And Execution MONitor
- dc = Desk Calculator
- dd = Disk Dump
- df = Disk Free
- diff = DIFFerence
- dmesg = diagnostic message
- du = Disk Usage
- ed = editor

- egrep = Extended GREP
- elf = Extensible Linking Format
- elm = ELectronic Mail
- emacs = Editor MACroS
- eval = EVALuate
- ex = EXtended
- exec = EXECute
- fd = file descriptors
- fg = ForeGround
- fgrep = Fixed GREP
- fmt = format
- fsck = File System CheckK
- fstab = FileSystem TABle
- fvwm = F*** Virtual Window Manager
- gawk = GNU AWK
- gpg = GNU Privacy Guard
- groff = GNU troff
- hal = Hardware Abstraction Layer
- joe = Joe's Own Editor
- ksh = Korn SHell
- lame = Lame Ain't an MP3 Encoder
- lex = LEXical analyser
- lisp = LISt Processing = Lots of Irritating Superfluous Parentheses
- ln = LiNk
- lpr = Line PRint
- ls = list
- lsof = LiSt Open Files
- m4 = Macro processor Version 4

- man = MANual pages
- mawk = Mike Brennan's AWK
- mc = Midnight Commander
- mkfs = MaKe FileSystem
- mknod = MaKe NODe
- motd = Message of The Day
- mozilla = MOsaic GodZILLa
- mtab = Mount TABle
- mv = MoVe
- nano = Nano's ANOther editor
- nawk = New AWK
- nl = Number of Lines
- nm = names
- nohup = No HangUP
- nroff = New ROFF
- od = Octal Dump
- passwd = PASSWorD
- pg = pager
- pico = PIne's message COmposition editor
- pine = "Program for Internet News & Email" = "Pine is not Elm"
- ping = 拟声又 = Packet InterNet Grouper
- pirntcap = PRINTer CAPability
- popd = POP Directory
- pr = pre
- printf = PRINT Formatted
- ps = Processes Status
- pty = pseudo tty
- pushd = PUSH Directory

- pwd = Print Working Directory
- rc = runcom = run command, rc 还是 plan9的 shell
- rev = REVerse
- rm = ReMove
- rn = Read News
- roff = RunOFF
- rpm = RPM Package Manager = RedHat Package Manager
  rsh, rlogin, rvim 中的 r = Remote
- rxvt = ouR XVT
- seamoneky = 我
- sed = Stream EDitor
- seq = SEQuence
- shar = SHell ARchive
- slrn = S-Lang rn
- ssh = Secure SHell
- ssl = Secure Sockets Layer
- stty = Set TTY
- su = Substitute User
- svn = SubVersioN
- tar = Tape ARchive
- tcsh = TENEX C shell
- tee = T (T 形水管接口)
- telnet = TEminaL over Network
- termcap = terminal capability
- terminfo = terminal information
- tex = τ

3

命令与文件的查询

脚本与文件名查询：which

文件名查找：whereis ,locate find

数据库更新：updatedb

脚本文件名的查询(which)

语法：which [-a] command

选项和参数：

-a:将由 PATH 目录中能找到的指令都列出

说明：which 执行更具当前用户环境变量指定的位置去寻找 command,并返回第一个找到的结果(-a 则返回所有)

```
[root@localhost tmp]# echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
[root@localhost tmp]# which ifconfig
/sbin/ifconfig
[root@localhost tmp]#
```

文件名查找

whereis

语法：whereis [-bmsu]文件或目录

选项和参数：

-b: 只找二进制格式的文件

-m:只找在说明文件 manual 路径下的文件

-s:只找 source 源文件

-u:查找不在上述三个选项的其他文件

```
[root@localhost tmp]# whereis ifconfig
ifconfig: /sbin/ifconfig /usr/share/man/man8/ifconfig.8.gz
```

说明：whereis 并不是从 PATH 指定路径查找，而是利用数据库查询

locate

语法: locate [-ir] keyword

选项和参数:

-i:忽略大小写

-r:后可接正则表达式

举例:

```
[root@localhost tmp]# locate passwd
/etc/passwd
```

说明：linux 会将所有文件都记录在数据库中，locate 和 whereis 从这个数据库进行查询，并不是扫描硬盘，因此可以提高效率，但是也带来一个问题就是不能保准数据库的信息和硬盘式同步的。 为了避免上述问题，可以手都去更新数据库，updatedb

Find

语法：find [PATH] [OPTION] [ACTION]

PATH:要查找的路径

OPTION：

与时间相关的参数

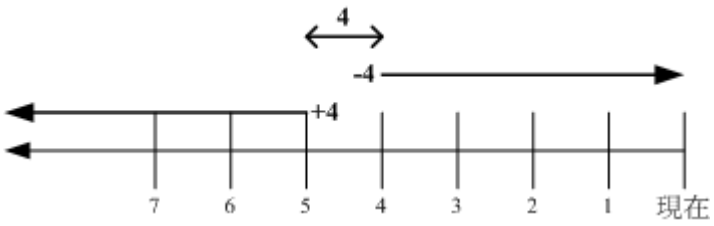
-atime,-ctime,-mtime，以-mtime 为例

-mtime n :表示在 n 天之前的“一天之内”被更改过的文件

-mtime +n： 列出在 n 天之前，不包含 n 天，被更改的文件

-mtime -n : 列出在 n 天之内，含 n 天本身被更改的文件

-newer file: file 为一个存在的文件。列出比 file 还新的文件



举例:

```
[root@bogon ~]# find / -mtime 0
[root@bogon ~]# find  /etc  -newer  /etc/passwd
```

与用户和用户组相关的参数

-uid n: 查询 UID(用户 ID)为 n 的文件

-gid n: 查询 GID(用户组 ID)为 n 的文件

-user name：查询所属用户名为 name 的文件

-group name: 查询所属用户组为 name 的文件

-nouser:查询不属于任何用户的文件

-nogroup查询不属于任何用户组的文件

举例:

```
[root@bogon ~]# find /home -user tkf
```

与文件权限及名称有关的参数

-name filename: 查找文件名为 filename 支持模糊查询

-size [+-]SIZE: 查询比 SIZE 大或小的的文件。大小单位 c 代表 byte ,k 代表 kb

-type TYPE: 查找文件类型为 TYPE 的文件

-perm mode： 搜寻文件权限刚好等于 ode 的文件，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： 搜寻文件权限必须要全部囊括 mode 的权限的文件，举例来说，我们要搜寻 -rwxr--r--，亦即 0744的文件，使用 -perm -0744，当一个文件的权限为 -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： 搜寻文件权限包含任一 mode 的权限的文件，举例来说，我们-rwxr-xr-x，亦即 -perm +755时，但一个文件属性为 -rw-------也会被列出来，因为他有 -rw....的属性存在!

举例：[root@bogon ~]# find / -name *http*

其他可进行的操作：

-exec command: command 为其他命令 - exec 后可接其他命令来处理查询到的结果

-print:将结果打印到屏幕上，默认操作

举例：

```
root@bogon ~]# find / -name *http*  -exec  ls -l {} \;
-rw-r--r-- 1 root root 97 2008-05-24 /etc/pam.d/system-config-httpd
-rw-r--r-- 1 root root 82 2008-05-24 /etc/security/console.apps/system-config-httpd
-rw------- 1 root root 464 2008-05-24 /etc/alchemist/switchboard/system-config-httpd.switchboard.adl
```

4

文件|目录的默认权限与隐藏权限

当我们创建一个文件或者目录时即使我们未对其非配权限，其也会存在默认权限

```
[root@localhost tmp]# mkdir newdir
[root@localhost tmp]# ls -dl newdir
drwxr-xr-x 2 root root 4096 02-21 11:10 newdir
[root@localhost tmp]# touch newfile
[root@localhost tmp]# ll newfile
-rw-r--r-- 1 root root 0 02-21 11:11 newfile
[root@localhost tmp]#
[root@localhost tmp]#
```

对于文件来说默认的权限是 rw-r--r—

对于目录来说默认的权限是 rwxr-xr-x

语法：

查看默认权限：umask [-S]

选项与参数：-S 以符号形式显示

设置默认权限：umask 权限数

说明：对于目录来说最大权限是777(rwxrwxrwx)

对于文件来说最大权限是666(rw-rw-rw-)

当权限数为022时代表：目录权限(777-022)=755(rwxr-xr-x)

当权限数为022时代表：目录权限(666-022)=644(rw-r--r—)

举例：查看默认权限

```
[ro[root@localhost tmp]# umask
0022
[root@localhost tmp]# umask -S
u=rwx,g=rx,o=rx
```

举例：设置默认权限

```
[root@localhost tmp]# umask 011
[root@localhost tmp]# mkdir newdir1
[root@localhost tmp]# ls -dl newdir1
drwxrw-rw- 2 root root 4096 02-21 13:09 newdir1
[root@localhost tmp]# touch newfile1
[root@localhost tmp]# ll newfile1
-rw-rw-rw- 1 root root 0 02-21 13:10 newfile1
```

文件隐藏属性(chattr|lsattr)

设置文件属性

语法：chattr [+-=][ASacdistu]文件或目录名称

选项与参数：

+：增加某一个特殊参数

-：删除某一个特殊参数

=：仅有后面接的参数

A:当配置了 A 这个属性时，若你有存取此文件(或目录)时，他的存取时间 atime 将不会被修改，可避免 I/O 较慢的机器过度的存取磁碟。这对速度较慢的计算机有帮助

S：一般文件是非同步写入磁碟的(原理请参考第五章 sync 的说明)，如果加上 S 这个属性时，当你进行任何文件的修改，该更动会[同步]写入磁碟中。

a：当配置 a 之后，这个文件将只能添加数据，而不能删除也不能修改数据，只有 root 才能配置这个属性。

c：这个属性配置之后，将会自动的将此文件『压缩』，在读取的时候将会自动解压缩， 但是在储存的时候，将会先进行压缩后再储存(看来对於大文件似乎蛮有用的！)

d：当 dump 程序被运行的时候，配置 d 属性将可使该文件(或目录)不会被 dump 备份

i：可以让一个文件[不能被删除、改名、配置连结也无法 写入或新增数据！]对与系统安全性有相当大的助益！只有 root 能配置此属性

s：当文件配置了 s 属性时，如果这个文件被删除，他将会被完全的移除出这个硬盘空间，所以如果误删了，完全无法救回来了喔!

u：与 s 相反的，当使用 u 来配置文件时，如果该文件被删除了，则数据内容其实还存在磁碟中，可以使用来救援该文件

举例：

```
[root@localhost tmp]# nano
[root@localhost tmp]# ll testa
-rw-r--r-- 1 root root 5 02-21 13:24 testa
[root@localhost tmp]# chattr +a testa
[root@localhost tmp]# nano testa  //此处修改不允许保存
[root@localhost tmp]# chattr =i testa
[root@localhost tmp]# rm testa
rm：是否删除有写保护的 一般文件 “testa”？y
rm: 无法删除 “testa”: 不允许的操作
```

查看文件属性

语法：lsattr [-adR] 文件或目录

选项与参数：

-a：将隐藏文件列出来

-d:如果接的是目录，仅列出目录本身属性而非目录内的文件名

-R：连同子目录的数据也一并列出来

查看文件类型

语法：file 文件

```
[root@localhost tmp]# file ~/.bashrc
/root/.bashrc: ASCII text
```

5

文件内容查阅

cat:由第一行开始显示文件内容

tac:由最后一行开始显示文件内容

nl:显示的时候，顺便输出行号

more:一页一页的显示文件内容

less:与 more 类似，但是它可以往前翻页

head:只看头几行

tail:只看结尾几行

touch:文件创建与文件时间修改

cat(concatenate)

语法：cat [-AbEnTv]

选项与参数：

-A：相当于-vET 的整合参数

-b:列出行号，仅针对非空白行做行号显示

-n:输出行号，空白与非空白都会列出

-E:将结尾的断行字符￥显示出来

-v:列出一些看不出的特殊字符

-T:将 Tab 按键以 ^I 显示出来

举例：

```
[root@localhost tmp]# cat /etc/issue
CentOS release 5.10 (Final)
Kernel \r on an \m

[root@localhost tmp]# cat -n /etc/issue
    1  CentOS release 5.10 (Final)
    2  Kernel \r on an \m
    3
[root@localhost tmp]# cat -A /etc/issue
CentOS release 5.10 (Final)$
Kernel \r on an \m$
$
```

添加行号与打印(nl)

语法： [root@www ~]# nl [-bnw] 文件

选项与参数:

-b：指定行号指定的方式，主要有两种：

-b a： 表示不论是否为空行，也同样列出行号(类似 cat -n)；

-b t： 如果有空行，空的那一行不要列出行号(默认值)；

-n：列出行号表示的方法，主要有三种：

-n ln： 行号在萤幕的最左方显示；

-n rn： 行号在自己栏位的最右方显示，且不加 0 ；

-n rz： 行号在自己栏位的最右方显示，且加 0 ；

-w：行号栏位的占用的位数。

举例

```
 [root@www ~]# nl /etc/issue
    1  CentOS release 5.3 (Final)
    2  Kernel \r on an \m

这个文件其实有三行，第三行为空白(没有任何字节)，
因为他是空白行，所以 nl 不会加上行号喔

[root@www ~]# nl -b a /etc/issue
    1  CentOS release 5.3 (Final)
    2  Kernel \r on an \m
    3
[root@www ~]# nl -b a -n rz /etc/issue
000001  CentOS release 5.3 (Final)
000002  Kernel \r on an \m
000003
自动在自己栏位的地方补上 0 了~默认栏位是六位数，如果想要改成 3 位数?

[root@www ~]# nl -b a -n rz -w 3 /etc/issue
001     CentOS release 5.3 (Final)
002     Kernel \r on an \m
003
```

语法：more|less文件

More:

空白键 (space)：代表向下翻一页；

Enter ： 代表向下翻『 一行 』；

/字串 ： 代表在这个显示的内容当中，向下搜寻『 字串 』这个关键字；

:f ： 立刻显示出档名以及目前显示的行数；

q ： 代表立刻离开 more ，不再显示该文件内容。

b 或 [ctrl]-b ： 代表往回翻页，不过这动作只对文件有用，对管线无用。

Less:

空白键 ： 向下翻动一页；

[pagedown]：向下翻动一页；

[pageup] ： 向上翻动一页；

/字串 ： 向下搜寻『 字串 』的功能；

?字串 ： 向上搜寻『 字串 』的功能；

n ： 重复前一个搜寻 (与 / 或 ? 有关！)

N ： 反向的重复前一个搜寻 (与 / 或 ? 有关！)

q ： 离开 less 这个程序；

举例:

```
[root@localhost tmp]# more /etc/man.config
 #
 # Generated automatically from man.conf.in by the
.....
 # and to determine the correspondence between extensions and decompressors.
 #
 # MANBIN              /usr/local/bin/man
 #
--More--(31%)
```

取出前面几行(head)

语法：head [-nnumber] 文件

选项与参数:

-n:后面接数字，代表行数

number 默认值是10 当 number 是负数，代表列出前面所有行数但是不包括后面 number 行

取出后面几行(tail)

语法：tail [-nnumber] 文件

选项与参数:

-n:后面接数字，代表行数

number 默认值是10 当 number 是正数（ + number ），代表该文件从 number 以后才会列出来

修改文件时间|创建新文件(touch)

时间属性

Mtime(modificationtime):当文件内容数据更改时就会更新这个时间，内容数据指的是文件的内容，不包括文件的权限和属性

Ctime(Statetime):当文件的状态（权限和属性）更改时会更新这个时间

Atime(accesstime):当文件内容被取用就会修改这个时间

举例:

```
[root@localhost ~]# ls -l --time-style=long-iso  /etc/man.config 默认是修改mtime
-rw-r--r-- 1 root root 4617 2012-05-30 20:34 /etc/man.config
[root@localhost ~]# ls -l --time=ctime --time-style=long-iso  /etc/man.config
-rw-r--r-- 1 root root 4617 2014-02-14 10:06 /etc/man.config
[root@localhost ~]# ls -l --time=atime --time-style=long-iso  /etc/man.config
-rw-r--r-- 1 root root 4617 2014-02-21 10:19 /etc/man.config
```

语法：touch[-acdmt] 文件

选项与参数:

-a:仅修改访问时间 atime

-c:仅修改文件的时间，若该文件不存在则不创建新文件

-d:后面可接欲修改的日期，也可以使用—date=”时间或日期”

-m:仅修改 mtime

-t:后面可以接欲修改的时间

主要功能:

创建一个空文件

修改文件日期(mtime,atime)

举例:

```
[root@localhost tmp]# cp -a /etc/man.config ./newman.config
[root@localhost tmp]# ls -l --time-style=long-iso  newman.config 指定时间格式
-rw-r--r-- 1 root root 4617 2012-05-30 20:34 newman.config
[root@localhost tmp]# touch -m -t 0709150203  newman.config //只修改mtime
[root@localhost tmp]# ls -l --time-style=long-iso  newman.config
-rw-r--r-- 1 root root 4617 2007-09-15 02:03 newman.config
 [root@localhost tmp]# ls -l --time=atime --time-style=long-iso  newman.config //只修改atime
-rw-r--r-- 1 root root 4617 2014-02-21 10:33 newman.config
```

```
[root@localhost tmp]# touch -a -t 0809150203  newman.config
 [root@localhost tmp]# ls -l --time=atime --time-style=long-iso  newman.config
-rw-r--r-- 1 root root 4617 2008-09-15 02:03 newman.config
[root@localhost tmp]#
[root@localhost tmp]# touch -d "2 days ago"  newman.config //默认修改atime 与 mtime
[root@localhost tmp]# ls -l --time=atime --time-style=long-iso  newman.config
-rw-r--r-- 1 root root 4617 2014-02-19 10:36 newman.config
[root@localhost tmp]# ls -l  --time-style=long-iso  newman.config
-rw-r--r-- 1 root root 4617 2014-02-19 10:36 newman.config
```

6

文件与目录管理

cd:切换目录

pwd:显示当前目录

mkdir:新建一个新的目录

rmdir:删除一个空的目录

ls:查看目录与文件

cp:复制

rm:删除

mv:移动|重命名文件与目录

切换目录(CD)

语法：cd [相对路径或绝对路径]

举例：

```
[root@localhost ~]# cd ~tkf  //~ 指定用户的主文件夹
[root@localhost tkf]# cd  //默认为当前用户的主文件夹
[root@localhost ~]# cd .. //发挥上一层
[root@localhost /]# cd /var/spool/mail   //绝对路径
[root@localhost mail]# cd ../mqueue/   //相对路径
[root@localhost mqueue]# pwd
/var/spool/mqueue
[root@localhost mqueue]# cd -        //前一个工作目录
/var/spool/mail
```

显示当前路径(PWD)

语法：pwd [-P]

选项与参数：

-P:显示当前路径，而非使用连接(link)路径

举例：

```
root@localhost ~]# pwd
/root
[root@localhost ~]# cd /var/mail/
[root@localhost mail]# pwd
/var/mail
[root@localhost mail]# pwd -P //  /var/mails是链接文件，实际路径是/var/spool/mail
/var/spool/mail
[root@localhost mail]#
```

新建目录(mkdir)

语法：mkdir [-mp] 目录名称

选项与参数：

-m:直接配置目录的权限

-p:帮助你直接将所需要的目录，递归创建起来

举例：

```
[root@localhost ~]# cd /tmp
[root@localhost tmp]# mkdir test
[root@localhost tmp]# ls -ald ./test
drwxr-xr-x 2 root root 4096 02-20 10:47 ./test
[root@localhost tmp]# mkdir test1/test2/test3
mkdir: 无法创建目录 “test1/test2/test3”: 没有那个文件或目录
[root@localhost tmp]# mkdir -p test1/test2/test3
[root@localhost tmp]# ls -ald ./test1/
drwxr-xr-x 3 root root 4096 02-20 10:48 ./test1/
[root@localhost tmp]# mkdir -m 711 test5
[root@localhost tmp]# ls -ald ./test5
drwx--x--x 2 root root 4096 02-20 10:48 ./test5
[root@localhost tmp]#
```

删除空目录(rmdir)

语法：rmdir [-p] 目录名称

选项与参数：

-p:连同上层“空的”目录一起删除

举例：

```
[root@localhost tmp]# rmdir test
[root@localhost tmp]# rmdir  test1
rmdir: test1: 目录非空
[root@localhost tmp]# rmdir -p  test1/test2/test3/
```

查看目录与文件(ls)

语法: [root@www ~]# ls [-aAdfFhilnrRSt] 目录名称

[root@www ~]# ls [--color={never,auto,always}] 目录名称

[root@www ~]# ls [--full-time] 目录名称

选项与参数:

-a：全部的文件，连同隐藏档(开头为 . 的文件) 一起列出来(常用)

-A：全部的文件，连同隐藏档，但不包括 . 与 .. 这两个目录

-d：仅列出目录本身，而不是列出目录内的文件数据(常用)

-f：直接列出结果，而不进行排序 (ls 默认会以档名排序！)

-F：根据文件、目录等资讯，给予附加数据结构，例如：

*:代表可运行档； /:代表目录； =:代表 socket 文件； |:代表 FIFO 文件；

-h：将文件容量以人类较易读的方式(例如 GB, KB 等等)列出来；

-i：列出 inode 号码，inode 的意义下一章将会介绍；

-l：长数据串列出，包含文件的属性与权限等等数据；(常用)

-n：列出 UID 与 GID 而非使用者与群组的名称 (UID 与 GID 会在帐号管理提到！)

-r：将排序结果反向输出，例如：原本档名由小到大，反向则为由大到小；

-R：连同子目录内容一起列出来，等于该目录下的所有文件都会显示出来；

-S：以文件容量大小排序，而不是用档名排序；

-t：依时间排序，而不是用档名。

--color=never： 不要依据文件特性给予颜色显示；

--color=always： 显示颜色

--color=auto： 让系统自行依据配置来判断是否给予颜色

--full-time： 以完整时间模式 (包含年、月、日、时、分) 输出

--time={atime,ctime}： 输出 access 时间或改变权限属性时间 (ctime)

而非内容变更时间(modification time)

复制(cp)

语法:

cp [-adfilprsu] 源文件目标文件

选项与参数:

-a:相当于-pdr

-d:若源文件为链接文件的属性，则复制连接文件属性而非文件本身

-f: 若目标文件已经存在且无法复制，则删除后在尝试一次

-i:若目标文件已经存在时，在覆盖是会先询问操作的进行

-l:进行硬链接

-p:连同文件的属性一起复制

-r:递归持续复制

-s:复制成符号链接文件

-u:若目标文件比源文件旧才更新

举例1：文件复制

```
[root@localhost tmp]# cp /var/log/wtmp wtmpTest
[root@localhost tmp]# cp -i /var/log/wtmp wtmpTest  //参数 i
cp：是否覆盖“wtmpTest”？y
[root@localhost tmp]# ll /var/log/wtmp wtmpTest
-rw-rw-r-- 1 root utmp 125952 02-20 10:25 /var/log/wtmp
-rw-r--r-- 1 root root 125952 02-20 13:18 wtmpTest    //属性变成了当前用户
[root@localhost tmp]# cp -a /var/log/wtmp wtmpTest_a //-a 连同属性一起复制
[root@localhost tmp]# ll /var/log/wtmp wtmpTest wtmpTest_a
-rw-rw-r-- 1 root utmp 125952 02-20 10:25 /var/log/wtmp
-rw-r--r-- 1 root root 125952 02-20 13:18 wtmpTest
-rw-rw-r-- 1 root utmp 125952 02-20 10:25 wtmpTest_a
[root@localhost tmp]# su tkf
[tkf@localhost tmp]$ cp -a /var/log/wtmp wtmpTest_t
[tkf@localhost tmp]$ ll  wtmpTest_t
-rw-rw-r-- 1 tkf tkf 125952 02-20 10:25 wtmpTest_t  //当用户权限不足时，即使-a 也无法更改属性
```

1. 源文件所在需要具有的权限 RX
2. 目的文件所在目录需要 WX 权限
3. 在权限不足的情况下即使-a 也无法更改文件属性

举例2：目录复制

```
[root@localhost tmp]# ll ./copydir/
-rw-r--r-- 1 root root 0 02-20 13:54 afile
-rw-r--r-- 1 root root 0 02-20 13:54 bfile
[root@localhost tmp]# cp /etc ./copydir/  // etc文件夹还有文件，直接复制失败
cp: 略过目录 “/etc”
[root@localhost tmp]# cp -r /etc ./copydir/  // 递归复制
[root@localhost tmp]# ll ./copydir/
-rw-r--r--   1 root root     0 02-20 13:54 afile
-rw-r--r--   1 root root     0 02-20 13:54 bfile
drwxr-xr-x 114 root root 12288 02-20 13:59 etc
```

举例3：软硬连接复制

```
[root@localhost tmp]# cp -s passwd passwd_slink
[root@localhost tmp]# cp -l passwd passwd_hlink
[root@localhost tmp]# ll passwd passwd_*
-rw-r--r-- 3 root root 2219 02-17 12:22 passwd
-rw-r--r-- 3 root root 2219 02-17 12:22 passwd_hlink
lrwxrwxrwx 1 root root    6 02-20 14:07 passwd_slink -> passwd  //连接文件
[root@localhost tmp]# cp passwd_slink passed_slink1
[root@localhost tmp]# cp -d passwd_slink passed_slink2

[root@localhost tmp]# ll pass*
-rw-r--r-- 1 root root 2219 02-20 14:20 passed_slink1 //源文件
lrwxrwxrwx 1 root root    6 02-20 14:21 passed_slink2 -> passwd // 连接文件
-rw-r--r-- 3 root root 2219 02-17 12:22 passwd
```

删除(rm)

语法:rm [-fir] 文件或目录

选项与参数:

-f:强制模式，不会进行询问

-i:互动模式

-r:递归删除

举例:

```
[root@localhost tmp]# ll copydir/
-rw-r--r--  1 root root     0 02-20 13:54 afile
-rw-r--r--  1 root root     0 02-20 13:54 bfile
drwxr-xr-x 114 root root 12288 02-20 13:59 etc
[root@localhost tmp]# rm -r ./copydir/
rm：是否进入目录 “./copydir/”？y
rm：是否删除 一般空文件 “./copydir//afile”？y
...
rm：是否删除 一般文件 “./copydir//etc/tux.mime.types”？y
可以rm -rf ./copydir/ 来避免提示
```

移动|重命名文件与目录(mv)

语法:rm [-fiu] 源文件，目标目录

选项与参数:

-f:强制模式，不会进行询问

-i:互动模式

-u:若目标文件已经存在，且源文件比较新才会更新

举例:

```
[root@localhost tmp]# mkdir movedir
[root@localhost tmp]# cp ~/.bashrc ./bashrc
[root@localhost tmp]# mv -i bashrc ./movedir/  //文件的移动
[root@localhost tmp]# ll ./movedir/
-rw-r--r-- 1 root root 176 02-20 14:55 bashrc
[root@localhost tmp]# mv ./movedir/bashrc ./movedir/b1 //文件的重命名
[root@localhost tmp]# ll ./movedir/
-rw-r--r-- 1 root root 176 02-20 14:55 b1
```

7

目录配置 FHS

随着 Linux 开发产品或 distributions 越来越多，如果每个人都按照自己的想法配置目录结构放置配置文件，那么就会造成很多管理的困扰，基于此后来 FHS(filesystem hierarchy standary)标准出现了，其主要目的是希望用户可以了解到已按照软件通常放置于那个目录下

	可分享	不可分享
不变的	/usr(软件放置处)	/etc(配置文件)
	/opt(第三方软件)	/boot(开机与内核相关)
可变的	/var/mail	/var/run(程序相关)
	/var/spool/news	/var/lock(程序相关)

其中不变的内容不因 Linux distributions 不同而改变其目录结构

可分享内容在网络上可分享给其他系统挂载使用

目录	应放置的文件内容
/	根目录 root(/)，一般建议在根目录下只有目录，不要直接有文件。根目录 是启动时系统第一个载入的分区，所以所有启动过程中会用到的文件都应该放在这个分区中。举例来说，/etc、/bin、/dev、/lib、/sbin 这 5个子目录都应该与根目录连在一起，不可独立成为某个分区。
/bin,/usr/bin,/us	放置用户可执行的二进制文件的目录。

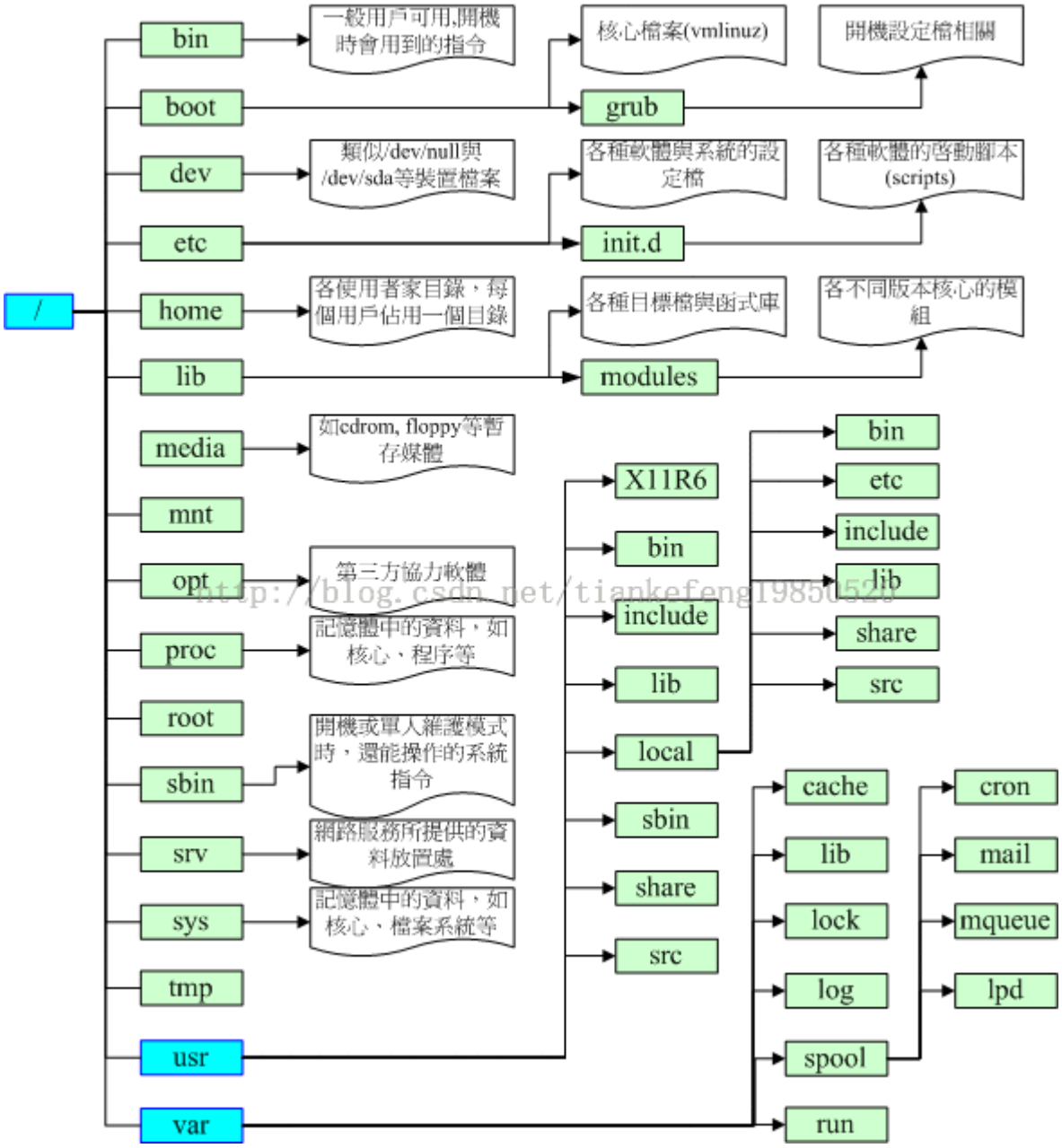
r/local/bin	
/boot	放置 Linux 系统启动时用到的文件。启动会用到 Linux 的核心文件。这个目录下面的文件 vmlinuz 就是 Linux 的核心。这一点非常重要，如果引导程序（loader）选择 grub，那么这个目录内还有/boot/grub 子目录。
/dev	在 Linux 系统上，任何设备都以文件类型存放在这个目录中，例如键盘、鼠 标、硬盘、光盘等。在此目录下的文件会多出两个属性，分别是主设备号(major device number)与辅设备号(minor device number)。系统核心就是通过这两个号码来判断设备的。重要的文件有/dev/null、/dev/tty[1-6]、/dev/ttyS*、 /dev/lp*、/dev/hd*、/dev/sd*等。
/etc	系统主要的设置文件几乎都放在这个目录内，例如人员的账号密码文件、各种服务的起始文件等。一般来说，这个目录下的各文件属性是可以让一般用户查看的，但只有 root 有权修改。在此目录下的文件几乎都是 ASII 的纯文本文件。不 过，FHS 建议不要在这个目录中放置可执行文件。比较重要的文件有：/etc/inittab、/etc/init.d、/etc /modprobe.conf、/etc/X11、/etc/fstab、/etc/sysconfig 等。
/home	这是系统默认的家目录(home directory)。
/lib,/usr/lib,/usr/local/lib	系统使用的函数库的目录。程序在运行过程中，可能会调用一些额外的参数，这需要函数库的协助。这些函数库就放在此处。比较重要的是/lib/modules 目录内有核心的相关模块。
/lost+f	系统出现异常，产生错误时，会将一些遗失的片段放于此目录下，通常这个目录会自动出现在某个分区顶层的目录下。

ound	
/mnt/media	这是软盘与光盘的默认载入点。通常软盘挂在/mnt/floppy 下，光盘挂在/mnt/cdrom 下。
/opt	这是给主机额外安装软件所放的目录。举例来说，FC4使用 Fedora 团队开发的软件，如果想要自行安装新的 KDE 桌面软件，可以将该软件安装在这个目录下。不过，以前的 Linux 系统中，我们还是习惯放在/usr/local 目录下。
/proc	这个目录本身是一个“虚拟文件系统”，它放置的数据都在内存中，例如系统核心、外部设备的状态及网络状态等。因为这个目录下的数据都在内存中，所以本身不占任何硬盘空间。比较重要的文件有/proc/cpuinfo、/proc /dma、/proc/interrupts、/proc/ioports、/proc/net/*等。
/root	系统管理员(root)的家目录。之所以放在这里，是因为系统第一个启动就载入的分区为/，而我们希望/root 能够与/放在同一块分区上。
/sbin,/usr/sbin,/usr/local/sbin	放一些系统管理员才会用到的可执行命令，例如：fdisk、mke2fs、fsck、mkswap、mount 等。与/bin 不太一样的地方是，这几个目录是给 root 系统管理用的。但目录下的执行文件可以让一般用户用来“查看”而不能设置。
/srv	一些服务启动之后，这些服务所需要访问的数据目录。举例来说，WWW 服务器需要的网页数据就可以放在/srv/www 中。

/tmp	这是让一般用户或者是正在执行的程序临时放置文件的地方。这个目录是任何人都能访问的，所以需要定期清理。当然，重要数据不可放在此目录中。
/usr	根据 FHS 规范的第二层内容，在/usr 目录下，包含系统的主要程序、图形界面 所需要的文件、额外的函数库、本机自行安装的软件，以及共享的目录与文件。它有点像 Windows 操作系统中的“Program files”与“Windows”这两个目录的结合。在此目录下的重要子目录有： • /usr/bin,/usr/sbin：一般身份用户与系统管理员可执行文件放置目录 • /usr/include：c/c++等程序语言的文件头（header）与包含文件（include）放置处，当以tarball 方式（*.tar.gz 的方式安装软件）安装某些数据时，会使用到里面的许多包含文件。 • /usr/lib：各种应用软件的函数库文件放置目录。 • /usr/local：本机自行安装的软件默认放置的目录。当前也适用于 /opt 目录。在安装完 Linux 之后，基本上所有的配置都有了，但软件总是可以升级的，例如要升级代理服务，则通常软件默认的安装地方就是/usr /local中。当安装完之后所得到的执行文件，为了与系统原执行文件区分，升级后的执行文件通常放在/usr/local/bin中。建议将后来才安装 的软件放在这里，便于管理。 • /usr/share：共享文件放置的目录，例如/usr/share/doc 目录放置一些系统帮助文件、/usr/share/man 放置 manpage 文件。 • /usr/src：Linux 系统相关的程序代码放置目录，例如/usr/src/linux 为核心源码。 • /usr/X11R6：系统内的 X Window System 所需的执行文件几乎都放在这里。
/var	这个目录也很重要，也是 FHS 规范的第二层目录内容。它主要放置系统执行过程中经常变化的文件，例如缓存（cache）或者是随时更改的日志文件（log file）。此外，某些软件执行过程中会写入的数据库文件，例如 MySQL 数据库，也都写入这个目录中。它下面的主要目录有： • /var/cache：程序文件在运行过程中的一些暂存盘。 • /var/lib：程序执行的过程中，使用的数据文件放置的目录。例如 locate 数据库与 MySQL 及 rpm 等数据库系统，都写在这个目录中。 • /var/log：登录文件放置的目录，很重要。例如/var/log/messages 就是总管所有登录文件的文件。

- /var/lock：某些设备具有一次性写入的特性，例如 tab（磁带机），此时，为了避免被其他人干扰正在运行的操作，会将该设备 lock（锁）起来，以确定该设备只能被单一程序所用。
- /var/run：某些程序或者是服务启动后，会将它们的 PID 放在这个目录下。
- /var/spool：是一些队列数据存放的地方。例如主机收到电子邮件后，就会放到/var/spool/mail 中，若信件暂时发不出去，就会放到/var/spool/mqueue 目录下，用户工作任务分配（cron）则是放在/var/spool/cron 中。

目录树以图标的方法来显示(图片摘自鸟哥的私房菜)



8

Linux 学习记录--文件权限

文件权限 Linux 针对文件权限分为三组，即用户，用户组，其他 可通过 ll(ls -l) 查看文件权限,此命令后续介绍

[root@localhost ~]# ll /etc/termcap

-rw-r--r--1 rootroot 807103 2007-01-07/etc/termcap

红色部分代表文件权限

黄色部分代表该文件所属用户

绿色部分代表该文件所属用户组

对于文件权限可分为3种（严格说并不是3种）

权限种类	值	描述
r	4	可读
w	2	可写
x	1	可执行

字符意义

文件权限相关共10个字符，其意义分别为

第1个字符：文件类型

[d]表示文件目录

[-]表示文件

[l]表示连接文件

[b]表示设备文件里的可供存储的接口设备

[c]表示设备文件里面的串行端口设备，如键盘

第2~4个字符：用户权限

第5~7个字符：用户组权限

第8~10个字符：其他用户权限

举例 `-rw-r----x` 1 A G1 807103 2007-01-07 /file

由上信息可看出：

该文件属于用户 A,属于用户组 G1

该文件对于用户权限为 rw- 即为可读写权限

该文件对于用户组权限为 r-- 即为可读权限

该文件对于其他权限为--x 即为可执行权限

假设 A,B,C 三个用户，A B 属于用户组 G1 ， C 属于用户组 G2

那么 A 就有拥有 rw-权限

由于 B 属于用户组 G1，因此 B 拥有 r--权限

由于 C 属于用户组 G2，因此 C 拥有--x 权限

权限与属性的更改

chgrp:更改文件所属用户组

chown:更改文件所有者

chmod：更改文件权限

chgrp

这个命令就是 change group 的简称，不过要被改变的组名要在/etc/group/文件内存在才行，否则会报错

语法：chgrp [-R] 用户组 dirname/filename

选项与参数：

-R：递归参数(recursive) 的持续更改，连同子目录下的所有文件，目录一起更改

举例：

```
[root@localhost ~]# ll install.log  //查看 install.log属性
-rw-r--r-- 1 root root 35014 02-14 10:29 install.log
[root@localhost ~]# chgrp users install.log //更改用户组为 users
[root@localhost ~]# ll install.log //查看 install.log 属性
-rw-r--r-- 1 root users 35014 02-14 10:29 install.log
[root@localhost ~]# chgrp nogronp install.log //输入一个无效的组
chgrp: 无效的组 “nogronp”
```

chown

这个命令就是 change owenr 的简称，不过要被改变的用户要在/etc/passwd/文件内存在才行，否则会报错

语法：chown[-R] 用户 文件/目录

chown[-R] 用户：组名文件/目录

选项与参数：

-R：递归参数(recursive) 的持续更改，连同子目录下的所有文件一起更改

举例：

```
[root@localhost ~]# ll install.log
-rw-r--r-- 1 root root 35014 02-14 10:29 install.log
[root@localhost ~]# chown bin install.log
[root@localhost ~]# ll install.log
-rw-r--r-- 1 bin root 35014 02-14 10:29 install.log
 have new mail in /var/spool/mail/root
```

```
[root@localhost ~]# chown tkf:users install.log
[root@localhost ~]# ll install.log
-rw-r--r-- 1 tkf users 35014 02-14 10:29 install.log
```

chmod

权限设置分为2种，分别可以使用数字和符号

语法：chmod [-R] 权限 文件/目录

chmod [-R] 符号表达式文件/目录

|:-----|:----------------|:-------|:--------| |chmod|u(user)g(group)o(other)a(all)|+(加入)-(除去)=(设置)|文件或目录|

选项与参数： -R：递归参数(recursive) 的持续更改，连同子目录下的所有文件一起更改

举例

```
[root@localhost ~]# ll baa
-rw-r--r-- 1 root root 176 2007-01-06 baa
[root@localhost ~]# chmod 754 baa
[root@localhost ~]# ll baa
-rwxr-xr-- 1 root root 176 2007-01-06 baa
[root@localhost ~]# chmod u=rw,g=x,o=x baa
[root@localhost ~]# ll baa
-rw---x--x 1 root root 176 2007-01-06 baa
[root@localhost ~]# chmod a+w baa
[root@localhost ~]# ll baa
-rw--wx-wx 1 root root 176 2007-01-06 baa
[root@localhost ~]#
```

目录与文件权限的意义

R(Read):可读取此文件的实际内容，如读取文本文件的文件内容 当你具一个目录读取 r 权限。表示你可以查看该目录下的文件名结构。 W(write): 可以编辑，新增或者是修改该文件的内容（但不含删除该文件） 当你具一个目录写入 w 权限。表示你可以更改该目录结构 1. 新建新的文件与目录

2. 删除已经存在的文件和目录（不论该文件的权限是什么）

3. 将以存在的文件或目录进行重命名

4. 转移该目录内的文件，目录位置

X(execute)：该文件具有可以被系统执行的权限（对于目录来说 X 就是进入文件夹的权限）

举例

```
[root@localhost ~]# mkdir -m 000 /tmp/testdir //root 用户权限为000的 文件夹 testdir
[root@localhost ~]# cd /tmp/testdir/   //超级用户无权限也可进入
[root@localhost testdir]# touch testfile  //创建文件
[root@localhost testdir]# chown tkf . // 将文件夹 testdir 用户变更为 tkf,以便一会切换用户操作
[root@localhost testdir]# ls -ald /tmp/testdir/ ./testfile
-rw-r--r-- 1 root root    0 02-19 12:59 ./testfile
d--------- 2 tkf  root 4096 02-19 12:59 /tmp/testdir/
[root@localhost testdir]# su tkf //切换用户
[tkf@localhost testdir]$ cd ..
[tkf@localhost tmp]$ ll  ./testdir/       //无 r 权限
ls: ./testdir/: 权限不够
[tkf@localhost tmp]$ chmod u+r ./testdir  //分配 r 权限
[tkf@localhost tmp]$ ll  ./testdir/
?--------- ? ? ? ?          ? testfile    //可以看到目录结构 但是因为无 X 权限，目录下文件属性看不到
[tkf@localhost tmp]$ cd testdir/          //无 x 权限
bash: cd: testdir/: 权限不够
[tkf@localhost tmp]$ chmod u=rx ./testdir  //分配 rx 权限
[tkf@localhost tmp]$ ll  ./testdir/
-rw-r--r-- 1 root root 0 02-19 12:59 testfile  //可以看到目录结构和文件属性
[tkf@localhost tmp]$ cd testdir/
[tkf@localhost testdir]$ rm -f testfile  //无 W 权限，不能删除目录下的文件
rm: 无法删除 “testfile”: 权限不够
[tkf@localhost tmp]$ chmod u=w ./testdir  //仅分配 W 权限
[tkf@localhost tmp]$ ls -ald ./testdir/
d-w------- 2 tkf root 4096 02-19 12:59 ./testdir/
[tkf@localhost tmp]$ rm ./testdir/testfile //缺少 X 权限 因此删不了
rm: 无法删除 “./testdir/testfile”: 权限不够
[tkf@localhost tmp]$ chmod u=rw ./testdir  //分配 RW 权限
```

```
[tkf@localhost tmp]$ rm ./testdir/testfile  //缺少 X 权限 因此删不了
rm: 无法删除 “./testdir/testfile”: 权限不够
[tkf@localhost tmp]$ chmod u=wx ./testdir //分配 XW 权限
[tkf@localhost tmp]$ rm ./testdir/testfile //可以删除文件，同时文件的用户属于 root ,在 tkf 用户仍可以删除
rm：是否删除有写保护的 一般空文件 “./testdir/testfile”？y
```

1. 如果目录只有 R 权限。可以查看目录下文件结构，但是看不到文件属性，并且进入不了目录（cd）
2. 如果目录有 RX 权限,可以查看目录下文件结构和属性，并且可以进入目录
3. 要删除目录下的文件。目录至少需要 WX 权限

9

关机相关指令

将数据同步写入硬盘指令 sync

关机指令 shutdown

重启，关机指令，reboot halt poweroff

只有 root 用户可以进行关机操作

数据同步写入磁盘 sync

由于所有的数据都要数据都要读入到内存才能被 CPU 所处理，但有时数据又需要由内存写回硬盘中，为了提高性能，已经加载到内存的中的数据不会理解被写回硬盘，当内存数据更改单位同步到硬盘中如果断电回引起数据，因此 sync 指令时强行将内存数据写入硬盘

reboot/shutdown/halt 执行前都会自动调用 sync

关机指令 shutdown

shutdown [-t 秒][arkhncfF] 时间 [警告信息]

-t sec: -t 后面加秒数，也即“过几秒后关机”的意思

-k:不是真的关机，知识发出警告信息

-r:将系统服务停掉后立即重启

-h:将系统服务停掉后立即关机

-n:不经过 init 程序,直接以 shutdown 功能来关机

-f:关机并开机之后，强制略过 fsck 的磁盘检查

-F:系统重启之后，强制进行 fsck 的磁盘检查

-c:取消已经在进行的 shutdown 命令内容

举例:

shutdown -h 10 “I will shutdown after 10 mins”

告诉大家10分钟后服务器重启

shutdown - h now

立刻关机

shutdown -h 20:15

20:15分自动关机

Shutdown -r now

立刻重启启动

Shutdown -r 30 “The System will Reboot”

30分钟后重新启动并通知在线用户

Shutdown -k now “The System will Reboot”

仅发出警告，并不会重启

关机指令 halt

halt 就是调用 shutdown –h now halt 执行时，杀死应用进程，执行 sync 系统调用，文件系统写操作完成后就会停止内核 halt [–dfinpw] –d :不要在 wtmp 中记录。 –f :不论目前的 runlevel 为何，不调用 shutdown 即强制关闭系统。 –i :在 halt 之前，关闭全部的网络界面。 –n :halt 前，不用先执行 sync。 –p :halt 之后，执行 poweroff。 –w :仅在 wtmp 中记录，而不实际结束系统。

halt 会先检测系统的 runlevel。若 runlevel 为0或6，则关闭系统，否则即调用 shutdown 来关闭系统。

切换执行等级 init

linux 操作系统自从开始启动至启动完毕需要经历几个不同的阶段，这几个阶段就叫做 runlevel，通常有8个 runlevel

Runlevel System State

0 Halt the system

1 Single user mode

2 Basic multi user mode

3 Multi user mode

5 Multi user mode with GUI

6 Reboot the system

S, s Single user mode

多数的桌面的 linux 系统缺省的 runlevel 是5，用户登陆时是图形界面，而多数的服务器版本的linux 系统缺省的 runlevel 是3，用户登陆时是字符界面，runlevel 1和2除了调试之外很少使用，runlevel s 和 S 并不是直接给用户使用，而是用来为 Single user mode 作准备。

10

工作管理与进程管理

始终不能明白进程的正确理解和定义。就说我自己的理解吧

进程是 CPU 调度的基本单位，对于 unix like 来说，当我们登录取得 bash 时，系统会根据用户的uid 和 gid 分配给我们一个进程，在当前 bash 下，这个进程就是所有进程的父进程，当我们执行一些命令时，每个命令都由一个新的子进程来完成。

工作管理

在单一终端下，可以同时进行多项工作，如：一边复制数据，一边查询文件。每一项工作都由独立的子进程来完成，他们的父进程就是当前终端对应 bash 的那个进程。

对于终端来说分为前台和后台

前台：你可以控制于执行命令的那个环境（串行工作）

后台：可以自行运行的工作，无法使用 ctrl+c 终端它（并行工作）

如果所有的工作都由前台来做，那么必须等一个工作处理完成才能进行下一个工作，这样做效率很低，因此我们可以把一些不需要人工交互的工作放到后台，使多个工作可以共同执行

说明：前台和后台是针对同一个终端来说，tty1环境是无法管理 tty2的

查看目前的后台工作状态(jobs)

语法：jobs[-lrs]

选项与参数：

-l:列出工作与命令串之外，同时列出 PID

-r:仅列出后台 run 的工作

-s: 仅列出后台 stop 的工作

举例

```
[root@localhost tmp]# jobs -l
[1]-  9154 停止                vim newfile.txt
[2]+  9368 停止                find / -print
[3]   9374 Running             tar -jcpP -f /tmp/etc1.tar.bz2 /etc &
```

以上输出的格式为：工作序号|顺序 PID 状态 命令

顺序：分为(+,-,空白),+号代表最后一个被放到后台的工作；-号代表倒数第二个被放到后台的工作，倒数第三个及以后用空白

直接将命令放到后台执行 (&)

[root@localhost tmp]# tar -jcpP -f/tmp/etc.tar.bz2 /etc &

[2] 8985 其中【 2 】表示工作号，8985表示处理这个工作的子进程号

如果我们将压缩信息显示出来

[root@localhost tmp]# tar - jcpP -v -f/tmp/etc.tar.bz2 /etc &

虽然这个工作在后台进行，但是输出信息还是会在前台输出的，因此可以应用数据流重定向将输出信息写到文件

将目前工作放到后台并暂停(ctrl+z)

[root@localhost tmp]# vim newfile.txt

=>按下 ctrl+z

[1]+ Stopped vimnewfile.txt

将后台工作拿到前台来处理(fg)

语法：fg %工作序号

fg -

fg +

将后台工作由停止变为运行(bg)

语法：fg %工作序号

举例：

```
[root@localhost tmp]# jobs -l;bg %2;jobs -l
[1]-  9154 停止                 vim newfile.txt
[2]+  9729 停止                 tar -jcpP -f /tmp/etc1.tar.bz2 /etc
[2]+ tar -jcpP -f /tmp/etc1.tar.bz2 /etc &
[1]+  9154 停止                 vim newfile.txt
[2]-  9729 Running              tar -jcpP -f /tmp/etc1.tar.bz2 /etc &
```

后台任务管理(kill)

语法：kill －signal %工作序号

Sighal:

1:重新读取一次参数的配置文件

2:代表有 ctrl+c 同样的操作

9:立刻强制删除一个工作

15.以正常方式终止工作

说明：unix like 中的信号64个，比较复杂以后在整体学习

举例:

```
[root@localhost tmp]# jobs -l;kill -9 %1; jobs -l
[1]+  已杀死               tar -jcpP -f /tmp/etc1.tar.bz2 /etc
```

进程管理

进程的查看

静态进程查看(ps)

语法:

ps aux|ps -lA 查看系统所有进程数据

ps -l 查看自己相关进程

ps -axjf 查看进程树

举例:

```
[root@localhost ~]# su -l tkf
[tkf@localhost tmp]$ tar -cjPp -f ./home.bz2 /home

[1]+  Stopped                 tar -cjPp -f ./home.bz2 /home

查看 ps -l
[tkf@localhost tmp]$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S   500 10894 10893  0  75   0 -  1224 wait   pts/1    00:00:00 bash
0 T   500 10951 10894  0  78   0 -  1224 finish pts/1    00:00:00 tar
0 T   500 10952 10951  0  78   0 -  2289 finish pts/1    00:00:01 bzip2

查看 ps -lA
[tkf@localhost tmp]$ ps -lA
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  75   0 -   544 stext  ?        00:00:00 init
0 R     0 10691     1  0  75   0 - 20262 stext  ?        00:00:01 gnome-terminal
4 S     0 10694 10691  0  77   0 -   649 stext  ?        00:00:00 gnome-pty-help
0 S     0 10695 10691  0  76   0 -  1253 wait   pts/1    00:00:00 bash
4 S     0 10893 10695  0  78   0 -  1319 wait   pts/1    00:00:00 su
4 S   500 10894 10893  0  75   0 -  1224 wait   pts/1    00:00:00 bash
0 T   500 10951 10894  0  78   0 -  1224 finish pts/1    00:00:00 tar
0 T   500 10952 10951  0  78   0 -  2289 finish pts/1    00:00:01 bzip2
```

处理 Init 程序的进程为所有进程的父进程 tar 命令内部会调用 bzip2命令因此产生10952进程，且bzip2进程的父进程就是 tar 进程

l F：代表这个程序旗标 (process flags)，说明这个程序的总结权限，常见号码有：

若为 4 表示此程序的权限为 root ；

若为 1 则表示此子程序仅进行复制(fork)而没有实际运行(exec)。

l S：代表这个程序的状态 (STAT)，主要的状态有：

R (Running)：该程序正在运行中；

S (Sleep)：该程序目前正在睡眠状态(idle)，但可以被唤醒(signal)。

D ： 不可被唤醒的睡眠状态，通常这支程序可能在等待 I/O 的情况(ex>列印)

T ： 停止状态(stop)，可能是在工作控制(背景暂停)或除错 (traced) 状态；

Z (Zombie)：僵尸状态，程序已经终止但却无法被移除至内存外。

l UID/PID/PPID：代表[此程序被该 UID 所拥有/程序的 PID 号码/此程序的父程序 PID 号码]

l C：代表 CPU 使用率，单位为百分比；

l PRI/NI：Priority/Nice 的缩写，代表此程序被 CPU 所运行的优先顺序，数值越小代表该程序越快被 CPU 运行。

l ADDR/SZ/WCHAN：都与内存有关，ADDR 是 kernel function，指出该程序在内存的哪个部分，如果是个 running 的程序，一般就会显示[-] / SZ 代表此程序用掉多少内存 / WCHAN 表示目前程序是否运行中，同样的，若为 - 表示正在运行中。

l TTY：登陆者的终端机位置，若为远程登陆则使用动态终端介面 (pts/n)；

l TIME：使用掉的 CPU 时间，注意，是此程序实际花费 CPU 运行的时间，而不是系统时间；

l CMD：简单描述命令

```
查看 ps aux
[tkf@localhost tmp]$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   2176   640 ?        Ss   08:50   0:00 init [5]
root     10691  0.1  0.8  81336 17104 ?        Sl   15:03   0:04 gnome-terminal
root     10694  0.0  0.0   2596   676 ?        S    15:03   0:00 gnome-pty-helper
root     10695  0.0  0.0   5012  1476 pts/1    Ss   15:03   0:00 bash
root     10893  0.0  0.0   5276  1312 pts/1    S    15:16   0:00 su -l tkf
tkf      10894  0.0  0.0   4896  1444 pts/1    S    15:16   0:00 -bash
tkf      10951  0.0  0.0   4896   992 pts/1    T    15:18   0:00 tar -cjPp -f ./home.bz2 /home
tkf      10952  0.0  0.3   9156  7792 pts/1    T    15:18   0:01 bzip2
```

USER：该 process 属於那个使用者帐号的?

PID ： 该 process 的程序识别码。

%CPU：该 process 使用掉的 CPU 资源百分比；

%MEM：该 process 所占用的实体内存百分比；

VSZ ： 该 process 使用掉的虚拟内存量 (Kbytes)

RSS ： 该 process 占用的固定的内存量 (Kbytes)

TTY： 该 process 是在那个终端机上面运行，若与终端机无关则显示 ?，另外， tty1-tty6 是本机上面的登陆者程序，若为 pts/0 等等的，则表示为由网络连接进主机的程序。

STAT：该程序目前的状态，状态显示与 ps -l 的 S 旗标相同 (R/S/T/Z)

START：该 process 被触发启动的时间；

TIME： 该 process 实际使用 CPU 运行的时间。

COMMAND：详细描述命令

```
查看 ps axjf
[tkf@localhost tmp]$ ps axjf
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
    1 10691  4628  4628 ?          -1 Sl       0   0:06 gnome-terminal
10691 10694  4628  4628 ?          -1 S        0   0:00  \_ gnome-pty-helper
10691 10695 10695 10695 pts/1    11743 Ss       0   0:00  \_ bash
10695 10893 10893 10695 pts/1    11743 S        0   0:00      \_ su -l tkf
10893 10894 10894 10695 pts/1    11743 S      500   0:00          \_ -bash
10894 10951 10951 10695 pts/1    11743 T      500   0:00            \_ tar -cjPp -f ./home.bz2 /home
10951 10952 10951 10695 pts/1    11743 T      500   0:01            |  \_ bzip2
```

动态进程查看(top)

语法：top[-d 数字] [-bn 数字] [-p pid]

选项与参数

-d:动态刷新时间间隔

-b:以批次方式执行，与-n 连用

-n:执行次数

-p:查看单个进程信息

在top 执行过程中可以使用的命令

? :显示帮助文档

P:以 CPU 使用资源排序

M: 以 mem 使用资源排序

N: 以 PID 排序

T: 以 TIME+排序

k:给某个 PID 一个信号

r:给某个 PID 设置新的 Ni 值

q:离开

Z|z:设置|取消颜色

B|b:设置|取消关键字加粗

W:将设置信息写入配置文件

n:显示进程数量

c:显示完整的 command

u:设置查看某个 USER 的进程

举例:

```
[tkf@localhost ~]$ top

top - 10:08:12 up 55 min,  2 users,  load average: 0.01, 0.01, 0.00
Tasks: 162 total,   1 running, 160 sleeping,   0 stopped,   1 zombie
Cpu(s):  1.5%us,  0.7%sy,  0.0%ni, 97.7%id,  0.0%wa,  0.2%hi,  0.0%si,  0.0
Mem:   2074908k total,   640268k used,  1434640k free,    41856k buffers
Swap:  1020088k total,        0k used,  1020088k free,   387260k cached

PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
4784 root     15   0 64580  17m  10m S  5.4  0.9   0:06.63 gnome-terminal
4502 root     16   0  148m  11m 5964 S  0.6  0.6   0:12.18 Xorg
```

特殊字段说明

us 用户空间占用 CPU 百分比

sy 内核空间占用 CPU 百分比

ni 用户进程空间内改变过优先级的进程占用 CPU 百分比

id 空闲 CPU 百分比

wa 等待输入输出的 CPU 时间百分比

hi 硬件中断

si 软件中断

st: 实时

USER：该 process 所属的使用者；

PR： Priority 的简写，程序的优先运行顺序，越小越早被运行；

NI： Nice 的简写，与 Priority 有关，也是越小越早被运行；

%CPU：CPU 的使用率；

%MEM：内存的使用率；

TIME+：CPU 使用时间的累加；

进程树查看(pstree)

语法：pstree[-A|-U] [-up]

选项与参数

-A： 各程序树之间的连接以 ASCII 字节来连接；

-U： 各程序树之间的连接以 unicode 的字节来连接。在某些终端介面下可能会有错误；

-p： 并同时列出每个 process 的 PID；

-u： 并同时列出每个 process 的所属帐号名称。

举例

```
[tkf@localhost ~]$ pstree -Apu
init(1)-+-/usr/bin/sealer(4774)
        |-acpid(3891)
        |-atd(4227)
        |-auditd(3599)-+-audispd(3601)---{audispd}(3602)
        |              `-{auditd}(3600)
        |-automount(3994)-+-{automount}(3995)
        |                 |-{automount}(3996)
        |                 |-{automount}(3999)
        |                 `-{automount}(4002)
```

进程的管理

进程的管理就是给进程发送一个信号，告诉进程做什么事情

语法：kill - signal PID

killall [-ile] - signal 服务名称

说明 使用 kill 需要知道 PID，killall 根据服务发送信号

举例1：给 bash 进程发送信号1

[root@localhost ~]# kill -1 $(ps aux |awk '{print $11 "" $2}'|grep '^bash'|awk '{print $2}')

举例2：给 bash 这个服务对应的进程发送信号1

[root@localhost ~]# killall -i -1 bash Kill bash(10202) ? (y/N)

设置进程优先级(nice)

进程的优先级相关的只要是 PRI 与 NI 这两个值, 数值越小优先级越高，PRI 是系统内核设置的，用户无法设置，用户只能是指 NI 值来改变进程的优先级

进程的优先级=PRI+NI

在执行命令时给予新的 NI 值:

语法：nice －n 数值 命令

数值的范围从−20~19

设置以及启动的进程

语法：renice 数值 PID

举例:

```
[root@localhost ~]# nice -n 4 vim &
[1] 11467
[root@localhost ~]# ps -l|grep 'vim'
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 T     0 11467 10202  0  81   4 -  2525 finish pts/1    00:00:00 vim

[root@localhost ~]# renice -6 $(ps -l|grep 'vim'|awk '{print $4}')
11467: old priority 4, new priority -6
[root@localhost ~]# ps -l|grep 'vim'
0 T     0 11467 10202  0  71  -6 -  2525 finish pts/1    00:00:00 vim
```

查询文件|目录与进程使用关系

我们经常会遇到这种情况，要删除某个文件或卸载某个文件系统，提示我们正在使用，无法被删除，此时我们就需要找到哪些进程正在使用它们，“杀死“这些进程后我们才能删除这些文件

查询文件被哪个进程使用(fuser)

语法：fuser[−umv] [−ki −signal] file/dir

选项与参数:

−u:列出 user

−m:会查询后面文件所在的文件系统

−v:列出详细命令

-k:找出使用该文件 PID 并试图发送 SIGKILL 这个信号给这个进程

-i:与-k 连用，发送信号之前询问用户

举例

```
[root@localhost tmp]# tar -jtv -f ./etc.tar.bz2 > ./newfile.txt
= >按下 ctrl+z 工作在后台暂停
[1]+  Stopped                 tar -jtv -f ./etc.tar.bz2 > ./newfile.txt
[root@localhost tmp]# fuser -uv ./etc.tar.bz2
                     USER        PID ACCESS COMMAND
./etc.tar.bz2:       root      11822 f.... (root)bzip2
= >这个文件正在被11822这个进程所使用
```

查询进程正在使用的文件(lsof)

语法:lsof [-aUu][+d]

选项与参数:

-a： 多项数据需要同时成立才显示出结果时

-U： 仅列出 Unixlike 系统的 socket 文件类型；

-u： 后面接 username，列出该使用者相关程序所开启的文件；

+d： 后面接目录,找出某个目录底下已经被开启的文件

举例:查看 tar 对应经常所打开的文件

```
[root@localhost tmp]# lsof |grep '^tar'
COMMAND    PID     USER   FD      TYPE     DEVICE SIZE/OFF       NODE NAME
tar      11821     root  cwd       DIR        8,2     4096     745569 /tmp
tar      11821     root  rtd       DIR        8,2     4096          2 /
tar      11821     root  txt       REG        8,2   229652    1199406 /bin/tar
……..
tar      11821     root    2u      CHR      136,1      0t0          3 /dev/pts/1
tar      11821     root    3r     FIFO        0,6      0t0      38124 pipe
```

查询正在执行进程的 PID

语法:pidof [-sx] 程序名

选项与参数:

-s:仅列出一个 PID

-x:列出程序所对应进程的父进程的 PID

举例:

[root@localhost tmp]# ps aux |grep $(pidoftar)|grep -v 'grep' root 11821 0.0 0.0 4880 944 pts/1 T< 1 3:09 0:00 tar -jtv -f./etc.tar.bz2

shell script

shell script 是利用 shell 的功能所写的一个程序，这个程序使用纯文本文件，将一些 shell 的语法和命令写在里面，搭配正则表达式，管道命令与数据流重定向等功能，达到我们想要的目的

shell script 执行

直接命令执行

shell script 文件必须具备 rx 的权限，假设 my.sh 在 /root 下

绝对路径

[root@bogon ~]# /root/my.sh

相对路径

[root@bogon ~]# . /my.sh

bash(sh)命令执行

shell script 文件必须具备 r 的权限

root@bogon ~]# sh my.sh

source 或.命令执行

source 或. 命令执行与上面执行不同，上面执行 shell 都会分配一个子进程来进行，source 或. 命令就在本进程执行，也就是说一些非环境变量也能读取到

root@bogon ~]# source my.sh

shell script 编写

前面提到 shell script 中可以使用 shell 命令，那么 shell script 格式是怎样的?

shellscript 文件 my.sh

```
#!/bin/bash

#This is my first shell script

echo "hello world"
```

第1行：必须要以#!/bin/bash 开头代表这个文件内的语法使用 bash 的语法

第2行：注释以#开头

第3行：shell 命令

shell script 重要功能

test测试命令

用来检测系统上面某些文件或者相关的属性

测试的标志	代表意义
1. 关于某个文件名的[文件类型]判断，如 test -e filename 表示存在否	
-e	该[文件名]是否存在? (常用)
-f	该[文件名]是否存在且为文件(file)? (常用)
-d	该[文件名]是否存在且为目录(directory)? (常用)
-b	该[文件名]是否存在且为一个 block device 装置?
-c	该[文件名]是否存在且为一个 character device 装置?
-S	该[文件名]是否存在且为一个 Socket 文件?
-p	该[文件名]是否存在且为一个 FIFO (pipe) 文件?
-L	该[文件名]是否存在且为一个连结档?

2. 关于文件的权限侦测，如 test -r filename 表示可读否 (但 root 权限常有例外)
-r |侦测该文件名是否存在且具有[可读]的权限?
-w |侦测该文件名是否存在且具有[可写]的权限?
-x |侦测该文件名是否存在且具有[可运行]的权限?
-u |侦测该文件名是否存在且具有[SUID]的属性?
-g |侦测该文件名是否存在且具有[SGID]的属性?
-k |侦测该文件名是否存在且具有[Sticky bit]的属性?
-s |侦测该文件名是否存在且为[非空白文件]?

3. 两个文件之间的比较，如： test file1 -nt file2
-nt |(newer than)判断 file1 是否比 file2 新
-ot |(older than)判断 file1 是否比 file2 旧
-ef |判断 file1 与 file2 是否为同一文件，可用在判断 hard link 的判定上。主要意义在判定，两个文件是否均指向同一个 inode 哩!

4. 关于两个整数之间的判定，例如 test n1 -eq n2
-eq |两数值相等 (equal)
-ne |两数值不等 (not equal)
-gt |n1 大于 n2 (greater than)

4. 关于两个整数之间的判定，例如 test n1 -eq n2
-lt |n1 小于 n2 (less than)
-ge |n1 大于等于 n2 (greater than or equal)
-le |n1 小于等于 n2 (less than or equal)

5. 判定字串的数据
test -z string |判定字串是否为 0？若 string 为空字串，则为 true
test -n string |判定字串是否非为 0？若 string 为空字串，则为 false。注： -n 亦可省略
test str1 = str2 |判定 str1 是否等于 str2 ，若相等，则回传 true
test str1 != str2 |判定 str1 是否不等于 str2 ，若相等，则回传 false

6. 多重条件判定，例如： test -r filename -a -x filename
-a |(and)两种情况同时成立！例如 test -r file -a -x file，则 file 同时具有 r 与 x 权限时，才回传 true。
-o| (or)两种情况任何一个成立！例如 test -r file -o -x file，则 file 具有 r 或 x 权限时，就可回传 true。
! |反相状态，如 test ! -x file ，当 file 不具有 x 时，回传 true

举例

shellscript 文件内容：

```
 #!bin/bash

read -p "please input file/dir name: " name
echo "your input is $name"
test ! -e $name && echo "file is not exist! "&&exit 1
test -d $name &&echo "file is directory"
test -f $name &&echo "file is regulate file"
test -r $name &&echo "you can read this file"
test -w $name &&echo "you can write this file"
test -x $name &&echo "you can exec this file"
exit 0
```

执行结果

```
[root@localhost scripts]# sh test.sh
please input file/dir name: n
your input is n
file is not exist!
[root@localhost scripts]# sh test.sh
please input file/dir name: sh01.sh
your input is sh01.sh
file is regulate file
you can read this file
you can write this file
you can exec this file
```

```
[root@localhost scripts]# ll sh01.sh
-rwxr--r-- 1 root root 48 03-07 10:15 sh01.sh
```

测试命令

[]测试命令和 test 用法一直，只是使用时要注意空格符使用

在中括号内的每个组件又要有空格符分割

在中括号内的变量常量都要使用“ ”括起来

举例

shellscript 文件内容：

```
 #!/bin/bash

read -p "please input y/n: " yn
[ -z "$yn" ] &&echo "your input is empty"&&exit 1
[ "$yn" == "y" -o "$yn" == "Y" ]&&echo "it is ok"&&exit 0
[ "$yn" == "n" -o "$yn" == "N" ]&&echo "it is not ok"&&exit 0
echo "you input is $yn .please use input (y/n)"
exit 2
```

执行结果

```
[root@localhost scripts]# sh [].sh
please input y/n:
your input is empty
[root@localhost scripts]# sh [].sh
please input y/n: Y
it is ok
[root@localhost scripts]# sh [].sh
please input y/n: A
you input is A .please use input (y/n)
```

执行参数

可以在执行 shell script 时添加参数

sh var.sh first second third

$1 $2 $3

其中 first 作为第一个参数存储在$1中，依次类推。

几个重要的参数：

$#：获取参数的数量

$@：获取参数的整体内容

举例

shellscript 文件内容：

```
#!/bin/bash

echo "total parameter number is : $#"
echo "whole parameter is : $@"
echo "1st  parameter is : $1"
echo "2st  parameter is : $2"
```

执行结果

```
[root@localhost scripts]# sh var.sh first second third
total parameter number is : 3
whole parameter is : first second third
1st  parameter is : first
2st  parameter is : second
```

if……then

语法：

If [条件判断式] ;then

待执行的 shell 命令

fi

举例

shellscript 文件内容：

```
#!/bin/bash

read -p "please input y/n: " yn
if [ -z "$yn" ];then
echo "your input is empty"
exit 1
fi
if [ "$yn" == "y" -o "$yn" == "Y" ];then
echo "it is ok"
exit 0
fi
if [ "$yn" == "n" -o "$yn" == "N" ];then
echo "it is not ok"
exit 0
fi
exit 2
```

执行结果

```
[root@localhost scripts]# sh ifthen.sh
please input y/n:
your input is empty
[root@localhost scripts]# sh ifthen.sh
please input y/n: Y
it is ok
[root@localhost scripts]# sh ifthen.sh
please input y/n: N
it is not ok
if …else
```

语法：

If [条件判断式] ;then

待执行的 shell 命令

else

待执行的 shell 命令

fi

if …elif..else

语法：

If [条件判断式] ;then

待执行的 shell 命令

elif[条件判断式] ;then

待执行的 shell 命令

else

待执行的 shell 命令

fi

case …esac

语法：

case$变量 in

“第一个变量内容”)

待执行的 shell 命令

;;

“第 n 个变量内容”)

待执行的 shell 命令

;;

*) è 最后一个变量内容. *代表所有

待执行的 shell 命令

;;

esac

举例

shellscript 文件内容：

```
 #!/bin/bash

case $1 in
"hello")
echo " hello how are you!"
;;
"")
echo "you must input parameter!"
;;
*)
echo "parameter is hello,yours is $1"
;;
esac
```

执行结果

```
[root@localhost scripts]# sh case.sh
you must input parameter!
[root@localhost scripts]# sh case.sh hello
hello how are you!
[root@localhost scripts]# sh case.sh he
parameter is hello,yours is he
```

函数功能

语法：

functionfname()

{

程序段

}

举例

shellscript 文件内容：

```
 #!/bin/bash

function print()
```

```
{
if [ -z "$1" ];then
  echo "你没有输入参数"
else
  echo -n "你输入的是 $1"
fi
}
function returnval()
{
  return 3
}
case $1 in
"one")
print 1
;;
"return")
returnval
echo "return value is $?"
;;
"nopara")
print
;;
esac
```

执行结果

```
[root@localhost scripts]# sh function.sh one
你输入的是 1
[root@localhost scripts]# sh function.sh nopara
你没有输入参数
[root@localhost scripts]# sh function.sh return
return value is 3
```

函数后面也可以添加参数$0记录函数名，$1记录参数1，依次类推

函数也可以具有返回值，返回值内容通过$?查看

while do done

语法：

while[条件表达式] =>只要条件满足就执行

do

程序段

done

举例

shellscript 文件内容：

```
declare -i sum=0;
declare -i num=0
while [ $num -le 100 ]
do
sum=sum+num
num=num+1
done
echo "总和是： $sum"
```

执行结果

```
[root@localhost scripts]# sh while.sh
总和是： 5050
```

until do done

语法：

while[条件表达式] =>只要条件不满足就执行

do

程序段

done

for do done

语法：

for varin con1 con2….

for varin conarr

for (初始值;限制值;执行步长)

举例

shellscript 文件内容：

```
 #!/bin/bash

userarr=$(cat /etc/passwd | cut -d ':' -f 1|head -n 5)
for user in $userarr
do
echo "user is : $user"
done
for char in A B C
do
echo "char is $char"
done
sum=0
for ((i=1; i<=$1; i++))
```

```
do
sum=$(($sum+$i))
done
echo "1+2..+$1=$sum"
```

执行结果

```
[root@localhost scripts]# sh for.sh 100
user is : root
user is : bin
user is : daemon
user is : adm
user is : lp
char is A
char is B
char is C
1+2..+100=5050
```

shell script 追踪与调试

语法：sh [-nvx] xxx.sh

选项与参数：

-n:不执行，仅检查语法错误

-v:在执行前，输出 script 内容

-x:将使用到 script 内容显示到屏幕，追踪主要靠这个

举例：

```
[root@localhost scripts]# sh -x function.sh nopara
+ case $1 in
+ print
+ '[' -z '' ']'
+ echo $'\344\275\240\346\262\241\346\234\211\350\276\223\345\205\245\345\217\202\346\225\260'
你没有输入参数
[root@localhost scripts]# sh -x function.sh one
+ case $1 in
+ print 1
+ '[' -z 1 ']'
+ echo '你输入的是 1'
你输入的是 1
```

12

Linux 学习记录--ACL 权限控制

第 12 章 ACL 权限控制

设置 ACL 权限：setfacl

查看 ACL 权限：getfacl

ACL 权限控制主要目的是提供传统的 owner,group,other 的 read,wirte,execute 权限之外的具体权限设置，可以针对单一用户或组来设置特定的权限

比如：某一目录权限为

drwx------ 2 root root 4096 03-10 13:51./acldir

用户 user 对此目录无任何权限因此无法进入此目录，ACL 可单独为用户 user 设置这个目录的权限，使其可以操作这个目录

ACL 启动

要使用 ACL 必须要有文件系统支持才行，目前绝大多数的文件系统都会支持，EXT3文件系统默认启动ACL 的

查看文件系统是否支持 ACL

```
[root@localhost tmp]# dumpe2fs -h /dev/sda2
dumpe2fs 1.39 (29-May-2006)
......
sparse_super large_file
Default mount options:    user_xattr acl
```

加载 ACL 功能

如果 UNIX LIKE 支持 ACL 但是文件系统并不是默认加载此功能，可自己进行添加

```
[root@localhost tmp]# mount -o remount,acl /
[root@localhost tmp]# mount
/dev/sda2 on / type ext3 (rw,acl)
```

同样也可以修改磁盘挂在配置文件设置默认开机加载

```
[root@localhost tmp]# vi /etc/fstab
LABEL=/                 /                       ext3    defaults,acl        1 1
```

查看 ACL 权限

语法：getfacl filename

设置 ACL 权限

语法：setfacl [-bkRd] [-m|-x acl 参数] 目标文件名

选项与参数：

-m:设置后续的 acl 参数，不可与-x 一起使用

-x: 删除后续的 acl 参数，不可与-m 一起使用

-b:删除所有的 acl 参数

-k:删除默认的 acl 参数

-R:递归设置 acl 参数

-d:设置默认 acl 参数，只对目录有效

针对特殊用户

设置格式：u:用户账号列表：权限

权限：rwx 的组合形式

如用户列表为空，代表设置当前文件所有者权限

举例：

```
[root@localhost tmp]# mkdir -m 700 ./acldir; ll -d ./acldir
drwx------ 2 root root 4096 03-10 13:51 ./acldir
[root@localhost tmp]# su tkf
[tkf@localhost tmp]$ cd ./acldir/
bash: cd: ./acldir/: 权限不够  =>用户无 X 权限
[tkf@localhost tmp]$ exit
exit
[root@localhost tmp]# setfacl -m u:tkf:x ./acldir/
=>针对用户 tkf 设置 acldir 目录的权限为 x
[root@localhost tmp]# ll -d ./acldir/
drwx--x---+ 2 root root 4096 03-10 13:51 ./acldir/
=>通过 ACL 添加权限在权限末尾会增加多个一个“+”同时文件原本权限也发生变化。
=>可通过 getfacl 查看原始目录权限
[root@localhost tmp]# getfacl ./acldir/
 # file: acldir

 # owner: root

 # group: root
```

```
user::rwx
user:tkf:--x  =>记录 tkf 用户针对此目录有 acl 权限
group::---
mask::--x
other::---
=>这里需要特殊说明，只是 tkf 这个用户具有 X 权限，其他用户还是无权限的
[root@localhost tmp]# su tkf
[tkf@localhost tmp]$ cd ./acldir/
[tkf@localhost acldir]$
=>用户 tkf 可以具有 x 权限可以进入目录
```

针对特定用户组

设置格式：g:用户组列表：权限 权限：rwx 的组合形式 如用户组列表为空，代表设置当前文件所属用户组权限

举例:

```
[root@localhost tmp]# setfa
setfacl   setfattr
[root@localhost tmp]# setfacl -m g:users:rx ./acldir/
[root@localhost tmp]# getfacl ./acldir/
 # file: acldir

 # owner: root

 # group: root

user::rwx
user:tkf:--x
group::--- => 其他用户组(非 acl 设置)的权限
group:users:r-x => 记录 users 用户组针对此目录有 acl 权限
mask::r-x
other::---
```

针对有效权限设置

有效权限（mask）就是 acl 权限设置的极限值，也就是你所设置的 acl 权限一定是 mask 的一个子集，如果超出 mask 范围会将超出的权限去掉 设置格式：m:权限 权限：rwx 的组合形式

举例:

```
[root@localhost tmp]# setfacl -m m:x ./acldir/
[root@localhost tmp]# getfacl ./acldir/
```

```
 # file: acldir

 # owner: root

 # group: root

user::rwx
user:tkf:--x
group::r-x                  #effective:--x
group:users:r-x             #effective:--x
mask::--x
other::---
```

针对默认权限设置

我们前面都是针对一个目录为一个用户（组）设置特定权限，但是如果这个目录下在新创建的文件是不具有这些针对这个用户的特定权限的。为了解决这个问题，就需要设置默认 acl 权限，使这个目录下新创建的文件有和目录相同的 ACL 特定权限

设置格式：d:[u|g]:用户(组)列表：权限

举例

```
[root@localhost tmp]# mkdir -m 711 ./defdir
[root@localhost tmp]# setfacl -m u:tkf:rxw ./defdir
[root@localhost tmp]# ll -d ./defdir/
drwxrwx--x+ 2 root root 4096 03-10 15:23 ./defdir/
=>目录权限具有 acl 特定权限（后面+）
[root@localhost tmp]# touch ./defdir/a.file;ll ./defdir/
-rw-r--r--  1 root root 0 03-10 15:25 a.file
=>新创建的文件不具有 acl 特定权限（后面无+）
[root@localhost tmp]# setfacl -m d:u:tkf:rxw ./defdir
=>设置默认权限
[root@localhost tmp]# getfacl ./defdir/
 # file: defdir

 # owner: root

 # group: root

user::rwx
user:tkf:rwx
group::--x
mask::rwx
```

```
other::--x
default:user::rwx
default:user:tkf:rwx
default:group::--x
default:mask::rwx
default:other::--x

[root@localhost tmp]# touch ./defdir/b.file;ll ./defdir/
-rw-r--r--  1 root root 0 03-10 15:25 a.file
-rw-rw----+ 1 root root 0 03-10 15:26 b.file
=>新创建文件默认带有 acl 特定权限
[root@localhost tmp]# getfacl ./defdir/b.file
 # file: defdir/b.file

 # owner: root

 # group: root

user::rw-
user:tkf:rwx                #effective:rw-
group::--x                  #effective:---
mask::rw-
other::---
=>这快我有个疑问，为什么 mask 值是 rw，我猜测和文件最大权限有关，
=>对于文件来时默认最大权限是666即 UMASK 为0000.那个对于可执行文件来说
=>没有 X,难道还需要使用 chmod 设置？ 疑问！！
```

13

文件特殊权限

文件除了读写(r),写(w),执行(x) 权限，还有些特殊权限(s,t)

SUID

功能:

SUID 权限仅对二进制程序有效

执行者对于程序需要有 X 可执行的权限

执行者将均有改程序所有者的权限

本权限只在执行程序过程中有效

举例:

普通用户也可以通过命令 passwd 修改自己的密码。修改的密码内容将会记录/etc/shadow 文件中，但是普通用户对这个文件无任何权限，那如何修改这个文件呢?

以上步骤可以理解为这样

普通用户执行 passwd 命令修改密码 è è passwd 命令程序修改/etc/shadow 文件将密码记录其中

[root@localhost /]# ll /etc/shadow/usr/bin/passwd

-r-------- 1 root root 1352 02-14 10:36 /etc/shadow

-rwsr-xr-x 1 root root 23420 2010-08-11/usr/bin/passwd

这就是 SUID 的功能，当普通用户在执行 passwd 程序命令时，由于 passwd 具有 SUID 权限，同时普通用户对于 passwd 命令具有 X 权限，那么在 passwd 执行过程中普通用户将程序所有者(root)的权限，因此/etc/shadow 就可以被修改

SGID

SGID 对于二进制程序来说，功能和 SUID 差不多

SGID 权限对二进制程序有效

执行者对于程序需要有 X 可执行的权限

执行者将均有改程序用户组的权限

本权限只在执行程序过程中有效

SGID 也可以针对目录设置，功能如下

用户在具有 SGID 权限的目录下创建的文件或目录其所属的用户组就是目录所有的有户组

说明，默认情况下用户创建的文件所属的用户组为用户的有效用户组

举例

```
[root@localhost /]# mkdir -m 777 /tmp/newdir;ll -d /tmp/newdir
drwxrwxrwx 2 root root 4096 03-10 12:35 /tmp/newdir
[root@localhost /]# cd /tmp/newdir/
[root@localhost newdir]# touch rootfile
[root@localhost newdir]# ll rootfile
-rw-r--r-- 1 root root 0 03-10 12:35 rootfile
=>所属用户和所属用户组都是 root
[root@localhost newdir]# su tkf
=>切换到普通用户
[tkf@localhost newdir]$ touch tkffile
[tkf@localhost newdir]$ ll
-rw-r--r-- 1 root root 0 03-10 12:35 rootfile
-rw-rw-r-- 1 tkf  tkf  0 03-10 12:36 tkffile
=>所属用户和所属用户组都是 tkf

[root@localhost ~]# chmod g+s /tmp/newdir/
=>给 newdir 添加 SGID 权限
[root@localhost ~]# ll -d /tmp/newdir/
drwxrwsrwx 2 root root 4096 03-10 12:36 /tmp/newdir/

[root@localhost ~]# su tkf
[tkf@localhost root]$ touch /tmp/newdir/sgidfile;ll /tmp/newdir/
-rw-r--r-- 1 root root 0 03-10 12:35 rootfile
-rw-rw-r-- 1 tkf  root 0 03-10 12:40 sgidfile
-rw-rw-r-- 1 tkf  tkf  0 03-10 12:36 tkffile
=> sgidfile 文件所属用户组发生变化，和目录(newdir)的用户组一样
```

SBIT

SBIT 只针对目录有效，其主要功能是 当用户拥有目录的 WX 权限时，用户可以删除（删除，重命名，移动）目录下的任意文件 当目录拥有 SBIT 权限时，即使用户拥有目录的 WX 权限，用户只能删除自己创建的文件(可以修改不是自己创建的文件)。root 用户都可以删除 **举例 **

```
[root@localhost ~]# mkdir -m 1777 /tmp/bitdir
[root@localhost ~]# su tkf
[tkf@localhost root]$ ll -d /tmp/bitdir/
drwxrwxrwt 2 root root 4096 03-10 12:59 /tmp/bitdir/
=>tkf 用户拥有目录的 rwx 权限
[tkf@localhost root]$ touch /tmp/bitdir/tkffile ;ll /tmp/bitdir/
-rw-rw-r-- 1 tkf tkf 0 03-10 12:59 tkffile
=>在目录下创建文件

[tkf@localhost root]$ su userA
=>切换另一个用户
[userA@localhost root]$ ll -d /tmp/bitdir/
drwxrwxrwt 2 root root 4096 03-10 12:59 /tmp/bitdir/
=> userA 用户拥有目录的 rwx 权限

[userA@localhost root]$ cd /tmp/bitdir/
[userA@localhost bitdir]$ touch userfile;ll
-rw-rw-r-- 1 tkf  tkf  0 03-10 12:59 tkffile
-rw-rw-r-- 1 userA userA 0 03-10 13:04 userfile
=>在目录下创建文件 userfile

[userA@localhost bitdir]$ rm tkffile
rm：是否删除有写保护的 一般空文件 “tkffile”？y
rm: 无法删除 “tkffile”: 不允许的操作
=>由于目录具有 SBIT 权限 虽然 userA 对目录具有 WX 权限，但是不能删除非他创建的文件

[root@localhost ~]# chmod o-t /tmp/bitdir/
=>将目录 SBIT 权限去掉
[root@localhost ~]# su userA
[userA@localhost root]$ cd /tmp/bitdir/
[userA@localhost bitdir]$ ll
-rw-rw-r-- 1 tkf  tkf  0 03-10 12:59 tkffile
-rw-rw-r-- 1 userA userA 0 03-10 13:04 userfile
[userA@localhost bitdir]$ rm tkffile
rm：是否删除有写保护的 一般空文件 “tkffile”？y
[userA@localhost bitdir]$ ll
```

```
-rw-rw-r-- 1 userA userA 0 03-10 13:04 userfile
=>可以删除不是自己创建的文件
```

权限设置方法

和设置和基本权限(rwx)方法基本，可以通过数字设置也可以通过符号设置

数字设置

SUID:4

SGID:2

SBIT:1

符号设置(+-=)

SUID：u+s

SGID：g+s

SBIT：o+t

举例:

```
[root@localhost ~]# touch test
[root@localhost ~]# chmod 4755 test; ll test
-rwsr-xr-x 1 root root 0 03-10 13:14 test
[root@localhost ~]# chmod 6755 test; ll test
-rwsr-sr-x 1 root root 0 03-10 13:14 test
[root@localhost ~]# chmod 1755 test; ll test
-rwxr-xr-t 1 root root 0 03-10 13:14 test
[root@localhost ~]# chmod 7666 test; ll test
-rwSrwSrwT 1 root root 0 03-10 13:14 test
=>这里 S,T 都是大写是因为文件本身不具有X权限，而 S,T 权限设置成功的
=>前提是文件具有 X 权限，因此大写的 ST 代表文件无这些特殊权限
```

14

正则表达式与其应用

数据处理工具：awk ,sed

正则表达式基本上是一种“表示法”，只要工具程序支持这种表示法，那么该工具程序就可以用来作为正则表达式的字符串处理只用。例如 vi,grep,awk,sed 等工具

正则表达式特殊符号

语系对应正在表达式也会存在影响。比如

LANG=C 时：0 1 2 3 4 … A B C D ..Z a b c d ..z

LANG=ZH_CN 时：0 1 2 3 4 …a A b B c C d D …….

因此[a-z]当 C 语系时代表的意义是获取小写字母。在 ZH_CN 语系时代表的意义就是获取字母(大写与小写)

为了避免数字和字母的选取错误，正则表达式采用特殊符号来代表

[:alnum:]：代表英文大小写字符及数字。A-Z a-z 0-9

[:alpha:]：代表英文大小写字符 A-Z a-z

[:blank:]：代表空格与 TAB 键

[:cntrl:]：代表键盘上的控制按键 CR,LF,TAB,DEL 等

[:digit:]：代表数据 0-9

[:graph:]：代表除了空格与 TAB 键的其他所有按键

[:upper:]：代表大写字符

[:print:]：代表任何可以被打印出来了的字符

[:punct:]：代表标点符号字符

[:space:]：代表会产生的空白的字符如 TAB 空格 CR

[:xdigit:]：代表十六进制的数字类型 0-9 A-F a-f

举例

```
[root@localhost ~]# cat xargsfile |grep -n '[[:upper:]]'
2:FRA
4:AWEE
```

基础正则表达式字符

|字符|意义与范例| |:--|:--------| |^word| 意义：查找以 word 为行首的数据 举例：查找以#开始的那一行 grep ‘^#’ file.txt| |Word$ |意义：查找以 word 为行尾的数据 举例：查找以#为结尾的那一行 grep‘#$’ file.txt| |. |意义：代表一定有一个任意字符 举例查找字符串 eae,ebe e e，ee 之间一定有一个字符，空格也算字符 G rep ‘e.e’ file.txt | |* |意义：重复0个到无穷个前一个字符 举例：查找含有 es ess esss 等的字符串 grep ‘ess*’ file.txt | |[] |意义：从字符集合中找出想要选取的字符 举例：查找含有 gl 或 gd 的那一行 grep ‘g[ld]’ file.txt| |[n1-n2] |意义：从字符集合里找出想要选取的字符范围 举例：查找含有任意数字的哪一行 grep ‘[0-9]’ file.txt| |[^] |意义：从字符集合中找处不要的字符或范围 举例：查找不含大写字母的那一行 grep ‘[^A-Z]’ file.txt| |{n,m}|意义：连续 n 个到 m 个的前一个字符，如{n}则是连续n个前一个字符，如{n,}则是连续 n 个以上前一个字符 举例1：查找 g 与 g 之间包含2个到3个 o 的字符串如：goog gooog grep ‘{2,\3}’ file.txt 举例2：查找 g 与 g 之间包含2个 o 的字符串如：goog grep ‘{2}’ file.txt 举例3：查找 g 与 g 之间包含3个及以上 o 的字符串如：gooog,gooood,goo….od grep ‘{3,}’ file.txt|

扩展正则表达式

|字符| 意义与范例| |:---|:----------| |+ |意义：重复一个或一个以上的字符 举例：查找god,good,good等字符串 egrep ‘go+d’ file.txt| |? |意义：0个过1个前一个字符 举例：查找gd god egrep ‘go?d’ file.txt| | | |意义：用或的方式找出数个字符串 举例：找出my ,own egrep ‘my|own’ file.txt| |() |意义：找出“组“的字符串 举例：找出good 或glad egrep ‘g(oo|la)d’ file.txt| |()+ |意义：多个重复组的判别 举例：找出Axy123 123123C egrep ‘Axy(123)+C’ file.txt|

说明：如 grep 需要使用扩展正则表达式，可使用 grep -e 或 egrep

sed

sed 本身是一个也是一个管道命令。可以分析输入数据流，还可以将数据进行替换，删除，选去等操作

sed 与 tr 的区别

tr 操作的单元是字符，它针对字符进行删除和替换

sed 操作单元的是行，它针对行进行删除和替换 **sed 与 vim 的区别**

sed 是管道命令它修改只是输入数据流，并不会修改文件本身。虽然 sed 也可以直接修改文件，但是不需要打开文件，对于大文件来说很有帮助

vim 是文本编辑器，它修改的是文件本身

语法：sed [-nefr] ‘动作’

选项与参数:

-n:silent 模式，只将 sed 处理过的内容显示 i 出来

-e:设置多个 sed 动作

-f filename: 文件内记录 sed 脚本 scipt

-r:sed 支持的扩展正则表达式语法（默认是基础正则表达式）

-i:直接修改读取文件内容，而不是屏幕输出

动作：n1,n2function

n1,n2不一定存在

function:

a：新增，a 后面接字符串，这些字符在当前的下一行显示

c：替换，c 后面接字符串，这些字符替换 n1-n2之间的行

d：删除，删除 n1-n2之间的行

i：添加，i 后面接字符串，这些字符在当前的上一行显示

p: 打印，打印 n1~n2行之间的数据

s: 替换以关键字形式替换，并不是替换整行. sed ‘s/旧字符串/新字符串/g’’

举例:

```
[root@localhost ~]# cat sedfile |sed -n 'p' =>查询所有内容
line 1
line 2
line 3
line 4
line 5
line 6
line 7
line 8
line 9
line 10
[root@localhost ~]# cat sedfile |sed -n '1,4p'=>查询1~4内容
line 1
line 2
line 3
line 4
[root@localhost ~]# cat sedfile |sed '2a new line'|sed -n '1,5p'
=>在第2行下添加新的一行
line 1
line 2
new line
line 3
line 4
[root@localhost ~]# cat sedfile |sed '3d'|sed -n '1,4p'
=>删除第3行
line 1
line 2
line 4
line 5
[root@localhost ~]# cat sedfile |sed '2i insert line'|sed -n '1,5p'
=>在第2行上添加新的一行
line 1
insert line
line 2
line 3
line 4
[root@localhost ~]# cat sedfile |sed '2,3c replace line'|sed -n '1,3p'
=>替换2~3行
line 1
replace line
line 4
[root@localhost ~]# cat sedfile |sed '1,3s/ne/NEL/g'|sed -n '1,6p'
```

```
=>用 NEL 替换2~3行的 ne
liNEL 1
liNEL 2
liNEL 3
line 4
line 5
line 6
```

举例2：使用 sed 直接修改文件

```
[root@localhost ~]# sed -i '$a this is line' sedfile ;cat sedfile|tail -n 2
line 10
this is line
```

awk

(awk 功能很强大，这里只是功能介绍性说明)

Awk 是一个数据处理工具，相比 sed 作用于一整行的处理，awk 则将一行分为数个“字段”处理

awk 的处理流程

1.读入第一行，并将第一行的数据填入$0,$1,$2……等变量中

2.依据条件类型的限制，判断是否需要后面的动作

3.做完所有的动作与条件类型

4.若还有后续的行的数据，则重复1-3步骤

语法：awk ‘条件类型1 {动作1}条件类型2 {动作2}……’ filename

说明：

1.awk 默认用空格或 tab 来分割一行数据，并将数据填充到$1,$2..中

如：root pts1 192这一行， $1=root $2=pts1.

2. awk 后方语句中非变量需使用双引号来定义，变量可以直接使用

awk 内置变量

运算符	描述
= += -= *= /= %= ^= **=	赋值
?:	C 条件表达式
||	逻辑或
&&	逻辑与
~ ~!	匹配正则表达式和不匹配正则表达式
< <= > >= != ==	关系运算符
空格	连接
+ -	加，减

运算符	描述
* / &	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
$	字段引用
in	数组成员

举例1：查看$1 NR NF

```
[root@bogon ~]# last -n 5 | awk '{print $1 "\t lines: " NR "\t cols: "NF }'
root     lines: 1       cols: 10
root     lines: 2       cols: 9
root     lines: 3       cols: 9
reboot   lines: 4       cols: 9
root     lines: 5       cols: 10
```

举例2：带有条件的，仅输出$1==root 的数据

```
[root@bogon ~]# last -n 5 | awk '$1=="root" {print $1 "\t lines: " NR "\t cols: "NF }'
root     lines: 1       cols: 10
root     lines: 2       cols: 9
root     lines: 3       cols: 9
root     lines: 5       cols: 10
```

awk 关键字

BEGIN

BEGIN 关键字作用是预设，在读如第一行前面就执行 BEGIN 后面的动作

比如每一行默认分割方式是空格或是 TAB，所以我们可以设置 FS 来改变分割符，但是此时第一行数据已经读取分析完毕，列信息已经存在$1，$2..中，改变只能从第2行开始。

举例:

```
[root@bogon ~]# cat /etc/passwd|head -n 5 |awk 'BEGIN {FS=":"} NR=="1" {print"UID\tGID"} NR>="1" {print $1 "\t" $3}'
UID     GID
root    0
bin     1
daemon  2
adm     3
lp      4
```

举例2 计算数据(num1+num2)

```
数据文件
month:num1:num2
1:100:150
2:200:250
3:300:350
4:400:450
5:500:550
6:600:650

[root@bogon ~]# cat cal.file |awk 'BEGIN {FS=":"} NR=="1" {print$1"\t"$2"\t"$3"\ttotal"} NR>"1" {print$1"\t"$2"\t"$3"\t"$2+$3}'
```

```
month   num1    num2    total
1       100     150     250
2       200     250     450
3       300     350     650
4       400     450     850
5       500     550     1050
6       600     650     1250
```

END

END 操作将在扫描完全部的输入之后执行

举例:

```
[root@bogon ~]# cat cal.file |awk 'BEGIN {FS=":"} NR=="1" {print$1"\t"$2"\t"$3"\ttotal"} NR>"1" {print$1"\t"$2"\t"$3"\t"$2+$3}
month   num1    num2    total
1       100     150     250
2       200     250     450
3       300     350     650
4       400     450     850
5       500     550     1050
6       600     650     1250
sum
```

15

管道命令

选取命令：cut，grep

排序命令:sort,wc,uniq

双重数据量：tee

字符转换命令：tr,expand,col

切割命令：split

参数代换：xargs

管道命令与连续命令不同，连续命令中的各个命令不存在相关性只是顺序执行。

对于管道命令来说 cmd1|cmd2.

cmd2需要 cmd1产生的输出流作为 cmd2的输入流,命令之间存在很强的依赖关系，并且管道命令只能处理正确的输出数据流

选取命令

cut

从某一行将一段信息切出来

语法：cut －d ‘分割字符’ −f field

cut －c 字符范围

选项与参数： −d:后接分割字符与−f 连用

−f:获取经−d 分割后的第几个字段

−c:以字符的单位取出固定字符区间，适用于排列正确的信息

选取范围 a−b 如果是从第 a 个字符到最后可写成 a−

说明：cut 可以进行单行与多行分割，对于多行每一行都看做单独的一行分割与获取 field

举例1：单行分割

```
[root@localhost ~]# echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
[root@localhost ~]# echo $PATH |cut −d ':' −f 1
/usr/kerberos/sbin
[root@localhost ~]# echo $PATH |cut −d '
/usr/kerberos/sbin:/usr/local/sbin
```

举例2：多行分割

```
[root@localhost ~]# last −5
root     pts/1        :0.0             Wed Mar  5 09:41   still logged in
root     :0                            Wed Mar  5 09:40   still logged in
root     :0                            Wed Mar  5 09:40 − 09:40  (00:00)
reboot   system boot  2.6.18−371.el5   Wed Mar  5 09:20         (05:08)
root     pts/1        :0.0             Tue Mar  4 15:27 − crash  (17:53)
[root@localhost ~]# last −5|cut −d ' ' −f 1
root
root
root
reboot
root
```

举例3：范围选取

```
[root@localhost ~]# export
declare −x COLORTERM="gnome−
```

```
declare -x DBUS_SESSION_BUS_
declare -x DESKTOP_SESSION="
declare -x DESKTOP_STARTUP_I
declare -x DISPLAY=":0.0"
[root@localhost ~]# export|cut -c 12-
COLORTERM="gnome-terminal"
DBUS_SESSION_BUS_ADDRESS="unix:abstract=/tmp/dbus-OeMZpvhP93,guid=30f2d841bcc5b92980611600531680a
DESKTOP_SESSION="default"
DESKTOP_STARTUP_ID=""
DISPLAY=":0.0"
```

grep

分析一行信息，若当中存在我们需要的信息，则将该行输出，grep 后还可接正则表达式或通配符进行查询。

语法：grep [-acinv] [-A] [-B] [--color=auto] ‘查找字符串’ filename

选项与参数：

-a:将 binary 文件以 text 文件方式查找数据

-c:计算‘查找字符串’次数

-i:忽略大小写

-n:输出行号

-v:反向选择

-A:后面可跟数字，代表除了本行外，后续的 n 行也都列出来

-B: 后面可跟数字，代表除了本行外，前面的 n 行也都列出来

--color=auto: 关键字部分添加颜色

举例：

```
[root@localhost ~]# last -3|grep 'root'
root     pts/1        :0.0          Wed Mar  5 09:41   still logged in
root     :0                         Wed Mar  5 09:40   still logged in
root     :0                         Wed Mar  5 09:40 - 09:40  (00:00)
[root@localhost ~]# last -5|grep -vn 'root'
4:reboot   system boot  2.6.18-371.el5   Wed Mar  5 09:20        (05:33)
6:
7:wtmp begins Fri Feb 14 10:32:51 2014
[root@localhost ~]# last |grep -c 'root'
84
[root@localhost ~]# last -5|grep -n 'roo*' = >通配符查找
1:root     pts/1        :0.0          Wed Mar  5 09:41   still logged in
2:root     :0                         Wed Mar  5 09:40   still logged in
3:root     :0                         Wed Mar  5 09:40 - 09:40  (00:00)
5:root     pts/1        :0.0          Tue Mar  4 15:27 - crash  (17:53)
```

排序命令

sort

sort 可以按照不同的数据类型来排序，例如按数字或文字排序，排序结果也受语系编码的影响，例如有的语系字符是这么排序的 AaBbCc…建议语系使用 LANG=C

语法：sort [-fbMnrtuk]文件或输入流

选项与参数：

-f:忽略大小写

-b:忽略最前面的空格

-M:以月份(英文)来排序

-r:反向排序

-t:分隔符与-k 连用

-u:就是 uniq

-k:以那个 field 的进行排序

举例1：

```
[root@localhost ~]# cat /etc/passwd |sort -
avahi:x:70:70:Avahi daemon:/:/sbin/nologin
bin:x:1:1:bin:/bin:/sbin/nologin
cimsrvr:x:100:500:tog-pegasus OpenPegasus WBEM/CIM services:/var/lib/Pegasus:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
```

举例2：用‘：’分割第三段进行排序

```
[root@localhost ~]# cat /etc/passwd |sort -t ':' -k 3
root:x:0:0:root:/root:/bin/bash
cimsrvr:x:100:500:tog-pegasus OpenPegasus WBEM/CIM services:/var/lib/Pegasus:/sbin/nologin
luci:x:101:101::/var/lib/luci:/sbin/nologin
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
```

uniq

将重复的数据仅列出一列

语法：uniq [-ic]

选项与参数：

-i:忽略大小写

-c:进行计数

举例:

```
[root@localhost ~]# last|cut -d ' ' -f 1 |sort|uniq -c|sort -n
      1 wtmp
      3 tkf
     26 reboot
     84 root
 => last|cut -d ' ' -f 1 |sort 截取登录名并排序
=> uniq -c 删除重复列,并计数
=>sort  - n 按照计数排序
```

WC

wc 可以帮助我们统计文件字符信息

语法：wc [lwm]

选项与参数:

-i:仅列出行

-w:仅列初子

-m:字符数

举例:

```
[root@localhost ~]# wc /etc/man.config
141  722 4617 /etc/man.config
[root@localhost ~]# cat /etc/man.config |wc
   141     722    4617
=>分别代表行数，字数，字符数
```

双重数据流(tee)

前面提到输出数据流介质可以是设备或文件，但数据流只能被一个介质全部接收，那么如果希望数据可以被2个介质接收就需要使用双重数据流，简单的说，如果你既想将输出数据流保存到文件也想同时控制台也会显示，那你就需要使用这个了

语法：tee [-a] file

选项与参数：-a:以累加的方式进行添加

举例

```
[root@localhost ~]# export|tee export.list|cut -c 12-
COLORTERM="gnome-terminal"
DBUS_SESSION_BUS_ADDRESS="unix:abstract=/tmp/dbus-OeMZpvhP93,guid=30f2d841bcc5b92980611600531680a
DESKTOP_SESSION="default"
[root@localhost ~]# vim export.list
declare -x COLORTERM="gnome-terminal"
declare -x DBUS_SESSION_BUS_ADDRESS="unix:abstract=/tmp/dbus-OeMZpvhP93,guid=30f2d841bcc5b9298061160
declare -x DESKTOP_SESSION="default"
```

字符转换命令

tr

tr 可以用来删除和替换一些文字信息

说明，tr 只是改变输出内容，并不会真正去修改文件的内容

语法：tr －d ‘字符’

tr －s ‘原字符’‘替换字符’

选项与参数：

−d:删除

−s:替换

举例：

```
trfile 文件内容
abcdefgh
abcdefgh
abcdefgh
[root@localhost ~]# cat trfile |tr −s 'b' 'B'  =>替换
aBcdefgh
aBcdefgh
aBcdefgh
[root@localhost ~]# cat trfile |tr −d 'b'  =>删除
acdefgh
acdefgh
acdefgh

=>操作结束后 trfile 文件内容不会有任何改变
```

col

col 主要将一些特殊字符进行转换

语法：col [−xb]

选项与参数：

−x:将 tab 键转成相应的空格

−b:在文字内有反斜杠，仅保留反斜杠后面接的那个字符

举例1:去掉反斜杠(^H)

```
[root@bogon ~]# man col > /root/col.man
[root@bogon ~]# cat -A /root/col.man|more
N^HNA^HAM^HME^HE$
    c^Hco^Hol^HI - filter reverse line feeds from input$

[root@bogon ~]# man col |col -b > /root/col.b.man
[root@bogon ~]# cat -A /root/col.b.man|more
    col - filter reverse line feeds from input$
```

举例2:将 TAB 转换为空格 (^I)

```
[root@bogon ~]# cat -A /etc/man.config|more # MANPATH^I/opt/*/man$
 # MANPATH^I/usr/lib/*/man$

 # MANPATH^I/usr/share/*/man$

 # MANPATH^I/usr/kerberos/man$


[root@bogon ~]# cat /etc/man.config |col -x > /root/man.tab.config
[root@bogon ~]# cat -A /root/man.tab.config|more
 # MANBIN              pathname$

 # MANPATH             manpath_element [corresponding_catdir]$

 # MANPATH_MAP         path_element    manpath_element$
```

expand

将[tab]按键转为空格键

语法：expand [- t] file

选项与参数：

-t:[tab] 按键替换多少个空格字符

举例

```
[root@localhost ~]# grep '^MANPATH' /etc/man.config |head -n 3|cat -A
MANPATH^I/usr/man$
MANPATH^I/usr/share/man$
MANPATH^I/usr/local/man$
[root@localhost ~]# grep '^MANPATH' /etc/man.config |head -n 3|expand -6|cat -A
MANPATH     /usr/man$
MANPATH     /usr/share/man$
MANPATH     /usr/local/man$
```

切割命令(split)

语法：split [-bl] file PREFIX

选项与参数：

-b:后面可接欲切割的文件大小

-1:以行数进行切割

PREFIX：切割后文件的前导符

举例1：切割文件

```
[root@localhost ~]# ll -h /etc/termcap
-rw-r--r-- 1 root root 789K 2007-01-07 /etc/termcap
[root@localhost ~]# split -b 300k /etc/termcap newter
[root@localhost ~]# ll -h newter*
-rw-r--r-- 1 root root 300K 03-06 09:56 newteraa
-rw-r--r-- 1 root root 300K 03-06 09:56 newterab
-rw-r--r-- 1 root root 189K 03-06 09:56 newterac
```

举例2： 合成文件

```
[root@localhost ~]# cat newter* >> termap-back
[root@localhost ~]# ll -h termap-back
-rw-r--r-- 1 root root 789K 03-06 10:02 termap-back
```

参数代换(xargs)

参数代换的作用:

1.作为某些指令的参数。比如 which, finger ,find ,whereis 等

2.作为某些不支持管道命令的输入数据流

语法：xargs [-epn] command

选项与参数：

-e:就是 EOF 的意思，后面可接一个字符串，当分析到这个字符串时，就会停止继续工作

-p:在执行每个参数时，都会询问用户

-n:后面接次数，执行 command 的次数

举例1：指令参数

```
[root@localhost ~]# cat ./xargsfile
cd
ll
grep
[root@localhost ~]# cut -d ' ' -f 1 ./xargsfile|xargs whereis
cd: /usr/share/man/man1p/cd.1p.gz /usr/share/man/man1/cd.1.gz
ll:
grep: /bin/grep /usr/share/man/man1p/grep.1p.gz /usr/share/man/man1/grep.1.gz
```

举例2：作为输入数据流

```
[root@localhost ~]# find /sbin/ -perm +7000|ls -l
总计 227644
-rw-------- 1 root root      1377 02-14 10:29 anaconda-ks.cfg
drwxr-xr-x 2 root root      4096 02-21 13:30 Desktop
.....
=>ls 不支持管道命令，一次查询的结果是~/下的内容
[root@localhost ~]# find /sbin/ -perm +7000|xargs ls -l
-rwsr-xr-x 1 root root 73108 10-02 05:10 /sbin/mount.nfs
-rwsr-xr-x 1 root root 73112 10-02 05:10 /sbin/mount.nfs4
....
=>将 find 查询到的内容作为输入数据流供 ls 使用
```

16

数据流重定向

数据流可以分为2种：

输入数据流：以写文件为例，从键盘输入的字符就输入数据流

输出数据流：以读文件为例，将文件内容显示到屏幕上，显示的内容就是输出字符流

数量流重定向就是指改变数据流输入的方式或输出的介质。比如，输入数据流可以是一个文件的内容，输出数据流介质可以是文件而不单单的屏幕

对于命令行来说输入数据流主要来自键盘，输出数据流只要介质是屏幕。

同时输出数据流又可分为：

正确输出

错误输出

语法：

输入数据流：使用<(覆盖)或<<(累加)

正确输出数据流：使用>(覆盖)或>>(累加)

错误输出数据流：使用2>(覆盖)或2>>(累加)

说明：如果某些信息不想显示到屏幕上也不保存到文件或设备上,可以讲输出数据流指向/dev/null

举例1：正确输出数据流(覆盖)

```
[root@localhost ~]# ll > ll.file
[root@localhost ~]# vim ll.file
总计 225968
-rw-------. 1 root root     1377 02-14 10:29 anaconda-ks.cfg
-rw-r--r--. 1 root root      207 03-05 11:00 bashrc-back
……..
```

举例2：正确输出数据流(累加)

```
[root@localhost ~]# ll /root >> ll.file
总计 225968
-rw-------. 1 root root     1377 02-14 10:29 anaconda-ks.cfg
-rw-r--r--. 1 root root      207 03-05 11:00 bashrc-back
……..
总计 225972
-rw-------. 1 root root     1377 02-14 10:29 anaconda-ks.cfg
-rw-r--r--. 1 root root      207 03-05 11:00 bashrc-back
……..
```

举例3：正确输出与错误输出数据流

```
[root@localhost ~]# ll /root /root/error
ls: /root/error: 没有那个文件或目录 =>错误信息
```

```
/root:          =>正确信息
总计 225972
-rw-------. 1 root root     1377 02-14 10:29 anaconda-ks.cfg
-rw-r--r-- 1 root root      207 03-05 11:00 bashrc-back
.....................
[root@localhost ~]# ll /root /root/error >right.list 2>error.list
[root@localhost ~]# cat right.list
/root:
总计 225984
-rw-------. 1 root root     1377 02-14 10:29 anaconda-ks.cfg
-rw-r--r-- 1 root root      207 03-05 11:00 bashrc-back
.....................
[root@localhost ~]# cat error.list
ls: /root/error: 没有那个文件或目录
```

举例4：正确与错误输出数据流写在一个文件中

```
[root@localhost ~]# ll /root /root/error >all.list 2>&1
[root@localhost ~]# cat all.list
ls: /root/error: 没有那个文件或目录
/root:
总计 225996
-rw-r--r-- 1 root root       45 03-05 13:02 all.list
-rw-------. 1 root root     1377 02-14 10:29 anaconda-ks.cfg
.....................
```

命令执行的判断依据(; && ||)

语法：

cmd;cmd:不考虑命令相关性连续额的命令执行

cmd1&& cmd2:若 cmd1执行完毕且正确，则执行 cmd2

若 cmd1执行错误则不执行 cmd2

cmd1|| cmd2：若 cmd1执行完毕且正确，则不执行 cmd2

若 cmd1执行完毕且为错误，则执行 cmd2

17

命名别名与历史命令

命名别名

语法：alias 别名=’命令’

unalias别名

alias 如后面什么也不跟。代表查询所有别名命名信息

举例1：查看所有别名

```
[root@localhost ~]# alias
alias cp='cp -i'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias mv='mv -i'
alias rm='rm -i'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
```

举例2：设置别名

```
[root@localhost ~]# alias dir='cd'
[root@localhost ~]# dir /tmp
[root@localhost tmp]# [root@localhost tmp]#
```

举例3：取消别名

```
[root@localhost tmp]# unalias dir
```

历史命令

语法：history n

history [-c] history [raw]histfiles

选项与参数：

n:数字，列出最近 n 条命令

-c:将目前 shell 中所有历史命令全部清除

-a:将目前新增的历史命令添加到 histfiles，若没有加 histfiles 则默认添加到 ~/.bash_history

-r:将 histfiles 读取到 这个 shell 的记忆

-w:将目前的 history 记忆写入 histfiles 中

说明：$HISTSIZE 记录了 shell 以及文件中最大存储历史记录数量

系统注销时会将 bashshell 历史记录记录到文件中~/.bash_history

举例：

```
[root@localhost tmp]# history 3 =>查看历史最近3条记录
  876  echo $HISTSIZE
  877  history
  878  history 3

[root@localhost tmp]# history -c =>清空 shell 中的历史记录
[root@localhost tmp]# history 4 =>以前的被清空 因此这里只能查询到这1条
1    history 4
[root@localhost tmp]# history -w =>shell 中的历史记录写入文件
[root@localhost tmp]# vim ~/.bash_history
history 5
vim ~/.bash_history
history -w
```

使用历史命令执行命令

语法：

!number:执行第几条命令

!command:由最近的命令向前搜索由 command 开头的命令

!!:执行上一个命令

说明：使用以上命令可以做好保密性，别人看到你的命令历史记录却不能知道你的操作

举例：

```
[root@localhost /]# history 5 =>先查询历史命令
   16  cd /
   17  history -a
   18  vim ~/.bash_history
   19  alias
   20  history 5
[root@localhost /]# !19 =>执行第19条命令
alias
alias cp='cp -i'
alias grep='grep --color=auto'
......
[root@localhost /]# !! =>执行上一个命令
alias
alias cp='cp -i'
alias grep='grep --color=auto'
[root@localhost /]# !al =>执行以 al 开头命令
alias
alias cp='cp -i'
```

shell变量

变量操作

变量显示(echo)

语法：echo $var

变量设置

语法：var=value

变量的设置规则

1. 变量两端不能直接接空格符
2. 变量名称只能是因为字母与数字，但开头不能使数字
3. 双引号内的特殊字符如$等，保持原本特性

[root@bogon ~]# var="lang is $LANG"

[root@bogon ~]# echo %var

root@bogon ~]# echo $var

lang is zh_CN.UTF-8

1.单引号内的特殊字符则仅为一般字符

[root@bogon ~]#

[root@bogon ~]# var='lang is $LANG';echo $var

lang is $LANG

2.在一串命令中，还需要通过其他命令提供的信息，可用单引号‘命令’或&（命令），举例：指令1在执行的过程中需要先知道指令2的的值，但是指令1,2在一串指令中

[root@bogon ~]# uname -r

2.6.18-371.el5

[root@bogon ~]# cd /lib/modules/$(uname -r)/kernel

[root@bogon kernel]#

3.变量的累加

[root@bogon kernel]# var=${var}yes

[root@bogon kernel]# echo $var

lang is $LANGyes

4.数组变量设置与读取

[root@bogon ~]# array[1]=a

[root@bogon ~]# array[2]=b

[root@bogon ~]# array[3]=c

[root@bogon ~]# echo ${array[1]}

a

[root@bogon ~]# echo ${array[2]}

```
b
[root@bogon ~]# echo ${array[3]}
c
```

取消变量(unset)

语法: unset var

变量查看(set)

语法：set

比较重要的几个自定义变量

HISTFILE:历史记录存储位置

MAILCHECK:多少秒扫描次邮箱，查看是否有新邮件

PS1:提示符设置

$:目前这个 shell 的 PID

?:刚才执行完命令的回传码。0为正确，非0为错误

举例：

```
[root@bogon kernel]# echo $$
11874  =>PID
[root@bogon kernel]# echo &?
[1] 22131
bash: ?: command not found
[root@bogon kernel]# echo $?
127 =>上一个命令执行错误，回传码非0
[root@bogon kernel]# echo $?
0 =>上一个命令执行正确，回传码为0
```

变量键盘读取(read)

语法： read [-pt] var

选项与参数：

-p:后可跟提示信息

-t:后跟等待输入的描述

举例：

```
[root@bogon ~]# read atest
this is a test
[root@bogon ~]# echo $atest
this is a test
[root@bogon ~]# read -p "please input.. " attest
please input.. hello world      =>提示信息
[root@bogon ~]# echo $atest
hello world
[root@bogon ~]# read -p "please input.. " -t 5  atest
please input..  =>5秒未输入回到命令行模式
[root@bogon ~]#
[root@bogon ~]# echo $atest
hello world
```

变量声明(declare)

语法：declare [-aixr] var

选项与参数

declare 后不接任何内容，代表查询所有变量，作用和 set 一致

-a： 将后面名为 variable 的变量定义成为数组 (array)类型

-i： 将后面名为 variable 的变量定义成为整数数字 (integer)类型

-x： 用法与 export 一样，就是将后面的 variable 变成环境变量;

+x： 将环境变量变为自定义变量

-r： 将变量配置成为 readonly 类型，该变量不可被更改内容，也不能 unset（需要注销后再登陆才能变回）

举例：

```
[root@bogon ~]# echo $sum
100+50+10 =>默认当做字符串处理
[root@bogon ~]# declare -i sum=100+50+10
[root@bogon ~]# echo $sum
160 =>声明为int 因此可以做加法
[root@bogon ~]# declare -x sum
[root@bogon ~]# export | grep sum
declare -ix sum="160" =>查询到是环境变量
[root@bogon ~]# declare +x sum
[root@bogon ~]# export | grep sum= >查询不到是环境变量
[root@bogon ~]# declare -r sum;sum=test
bash: sum: readonly variable =>只读允许修改
```

变量内容删除

语法

${var#/key}:从前往后删除符合 key 最短的那一个

${var##/key}:从前往后删除符合 key 最长的那一个

${var%/key}:从后往前删除符合 key 最短的那一个

${var%%/key}:从后往前删除符合 key 最短的那一个

举例：${var#/key}

```
[root@bogon ~]# path=${PATH};echo $path
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
[root@bogon ~]# echo ${path#/*:}        =>key为*.(*为通配符)
/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
```

举例：${var##/key}

```
[root@bogon ~]# path=${PATH};echo $path
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
[root@bogon ~]# echo ${path##/*:}
/root/bin
```

变量内容替换

语法：

${var/旧字符串/新字符串}:替换第一个满足条件的字符串

${var//旧字符串/新字符串}:替换所有满足条件的字符串

举例：${var/旧字符串/新字符串}

```
root@bogon ~]# path=${PATH};echo $path
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
[root@bogon ~]# echo ${path/sbin/SBIN}
/usr/kerberos/SBIN:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
```

举例：${var//旧字符串/新字符串}

```
[root@bogon ~]# path=${PATH};echo $path
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
[root@bogon ~]# echo ${path//sbin/SBIN}
/usr/kerberos/SBIN:/usr/kerberos/bin:/usr/local/SBIN:/usr/local/bin:/SBIN:/bin:/usr/SBIN:/usr/bin:/usr/X11R6/bin:/root/bin
```

命令内容的获取

将一些命令的输出内容存入变量

语法：$(command)

举例：

```
[root@localhost tmp]# wc /etc/passwd
  46   74 2219 /etc/passwd
[root@localhost tmp]# var=$(wc /etc/passwd)
[root@localhost tmp]# echo $var
46 74 2219 /etc/passwd
```

$与 echo 的理解

$负责是读取，echo 负责显示

以 Int a= 5对比 var 好比是 a

$var是读取 a 的值也就是5

echo &var 就是将5显示出啦

对于变量的连接：

[root@localhost tmp]# str1=hello

[root@localhost tmp]# str2=world

[root@localhost tmp]# var=$str1$str2

$str1将 str1的值也就是 hello 读取出来

$str2将 str2的值也就是 world 读取出来

var=$str1$str2 将连个值连接

环境变量

普通变量可以理解为局部变量，环境变量可以理解为全局变量，登陆成功获得的 bash shell 就是一个进程，在此情况下再去打开一个新 SHELL 就是他的子进程，子进程是无法获取父进程的自定义变量，但是可以获取父进程的环境变量

第 18 章 环境变量导出(export)

语法：export var

第 18 章 环境变量查看(env)

语法：env

比较重要的几个环境变量

HOME:代表用户的主文件夹

SHELL:代表目前使用的 shell 是哪个程序，我现在使用的是/bin/bash

HISTSIZE:历史记录最大存储条数

MAIL:mail命令系统收信时，系统会读取的信箱文件

PATH:执行文件查找路径

LANG:语系信息

RANDOM:随机数变量（0~32767）

提示符的设置(PS1)

变量 PS1='[\u@\h \W]\$ '记录了命令提示符的显示格式 [root@bogon ~]#

符号意义

\d： 可显示出[星期月日]的日期格式，如："Mon Feb 2"

\H： 完整的主机名。

\h： 仅取主机名在第一个小数点之前的名字

\t： 显示时间，为 24小时格式的[HH:MM:SS]

\T： 显示时间，为 12小时格式的[HH:MM:SS]

\A： 显示时间，为 24小时格式的[HH:MM]

\@： 显示时间，为 12小时格式的[am/pm]样式

\u： 目前使用者的账号名称，如[root];

\v： BASH的版本信息，如鸟哥的测试主板本为 3.2.25(1)，仅取[3.2]显示

\w： 完整的工作目录名称，由根目录写起的目录名称。但家目录会以 ~取代;

\W： 利用 basename 函数取得工作目录名称，所以仅会列出最后一个目录名。

#： 下达的第几个命令。

\$： 提示字符，如果是 root 时，提示字符为 #，否则就是 $

举例:

[root@bogon ~]# PS1='[\u@\h\A \W #]\$ '

[root@bogon23:45 ~ 82]#

shell

命令类型查询:type

读入配置文件:source

操作系统内核(kernel)负责管理整个计算机硬件，但是这个内核是需要保护的，用户不能直接操作内核，因此就需要一个可以帮助我们操作内核的工具。Shell 功能就在于此，他可以将我们输入的命令与内核通信，好让内核可以控制硬件来正确无误地工作

我们使用的是 linux 默认 shell 即 bash shell,其主要功能是:

命令记忆功能:

命令与文件不全功能

命名别名设置功能

作业控制，前台，后台控制

程序脚本

通配符

命令类型查询(type)

对于 shell 能够识别的变量分为：

内部命令：由 bash 内置的命令

外部命令：来字外部的命令，非 bash 内置

语法：type [-tpa] name

选项与参数：不加任何参数是，会显示出是内部命令还是外部命令

-t:会已以下关键字说明命令的意义

File:外部命令

Alias:通过别名设置的命令

Builtin:内置命令

-p:-如果后面接的是外部命令时才会显示完整文件名

-a:会有 PATH 变量定义的路径中，所有 name 命令列出来，包括 alias

举例：

```
[root@localhost ~]# type ls
ls is aliased to `ls --color=tty'
[root@localhost ~]# type -t ls
alias
[root@localhost ~]# type cd
cd is a shell builtin
[root@localhost ~]# type egrep
egrep is hashed (/bin/egrep)
[root@localhost ~]# type -t egrep
file
[root@localhost ~]# type -p egrep
/bin/egrep
```

路径与命令查找顺序

在我们系统中存在多个名字相同的名字，那么 bash shell 究竟使用的是哪个命令呢？其遵循的顺序如下：

1. 以相对/绝对路径来执行命令
2. 由 alias 找到命令来执行
3. 由 bash 内置命令来执行
4. 通过$PATH 的顺序找到的第一个命令来执行

举例

```
[root@localhost ~]# alias echo='echo -n'
[root@localhost ~]# type -a echo
echo is aliased to `echo -n'
echo is a shell builtin
echo is /bin/echo
=>可以看到先找alias在内置命令 最后PATH
```

Bash 的环境配置文件

当我们进如 bash 之后虽然我们什么也没有设置，但是系统的一些变量已经别名等信息就已经设置好了，这些信息就存在环境配置文件中，bash 启动时就会读取这些文件，对配置信息进行加载和设置

环境配置文件分类：

整体配置文件

个人配置文件

配置文件读取流程

用户输入完正确的用户名密码获得的 bash 首先会读取

1./etc/profile

这个文件设置的只要变量有 PATH,MAIL,USER,HOSTNAME.HISTSIZE,接下来调用其他文件加载数据 2./etc/inputrc

3./etc/profile.d/*.sh

这个目录下的文件规定了 bash 的操作借口颜色，语系，公共别名等信息

4./etc/sysconfig/i18n

这个文件由/etc/profile.d/lang.sh 调用复制设置语系

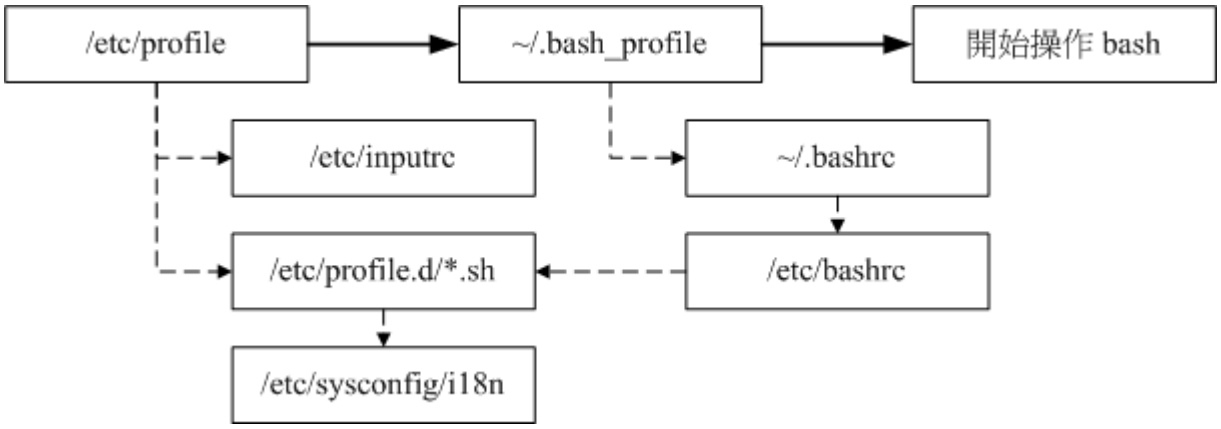
以上为整体配置文件，设置完成后开始设置个人配置文件

5.~/.bash_profile

~/.bash_profile,~/.bash_login ~/.profile 三个文件只读取一个，而且顺序按照前面的顺序

```
root@localhost ~]# cat ~/.bash_profile
 # .bash_profile


 # Get the aliases and functions

if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

 # User specific environment and startup programs


PATH=$PATH:$HOME/bin

export PATH
unset USERNAME
```

在这个文件中将用户主目录添加到 PATH 中，并将 PATH 变为环境变量。并且还回去读取~/.bashrc 文件（我通常将我个人的配置信息写到这里，如别名）



其他重要配置文件

/etc/man.config: 这个文件中记录了帮助信息在哪里

~/.bash_history:记录操作命令历史记录

~./bash_loyout:注销时系统做的事情记录在这里

第 19 章 读入环境配置文件(source)

环境配置文件只有 bash 在启动时读入，如果启动以后再修改就需要重新登录，才能让配置文件再一次被读入，source 命令就是避免重新登录，使修改后的配置文件重新读入后立即生效

语法：source 配置文件名

终端机环境设置

终端机中有的按键代表特殊的意义，如[backspace]代表删除，[ctrl+c]代表终止命令。在不同的Linux distributions 中终端机环境设置不尽相同。Stty 可以查看并设置这些命令

语法：stty －a

stty name 热键

选项与参数：

−a:查看所有环境中所有按键设置

name:按键设置名称

举例：查看所有按键信息

```
[root@localhost ~]# stty −a
speed 38400 baud; rows 41; columns 143; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M−^?; eol2 = M−^?; swtch = M−^?; start = ^Q; stop = ^S; susp = ^Z
werase = ^W; lnext = ^V; flush = ^O; min = 1; time = 0;
−parenb −parodd cs8 hupcl −cstopb cread −clocal −crtscts −cdtrdsr
−ignbrk brkint −ignpar −parmrk −inpck −istrip −inlcr −igncr icrnl ixon −ixoff −iuclc ixany imaxbel iutf8
opost −olcuc −ocrnl onlcr −onocr −onlret −ofill −ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok −echonl −noflsh −xcase −tostop −echoprt echoctl echoke
```

几个重要的内容：

Eof:代表输入结束

Erase:删除字符操作

Intr:终止目前命令

Kill:在提示符下，将正行命令删除

Quit:送出 quit 给正在运行的程勋

Start:暂停屏幕的输出

Stop:回复屏幕的输出

Susp: 暂停目前命令

举例2：修改按键信息

```
[root@localhost ~]# stty erase ^h
=>将删除字符按键变为[ctrl+h],此时[backspace]按键变成[ctrl+？ ]操作,
[root@localhost ~]# stty −a
speed 38400 baud; rows 41; columns 143; line = 0;
intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D; eol = M−^?; eol2 = M−^?; swtch = M−^?; start = ^Q; stop = ^S; susp = ^Z
```

通配符

Bash shell 特点之一就是通配符，可以方便我们查找。

说明：通配符和正则表达式不是一个东西

常用的通配符：

*：代表0~n 个任意字符

?: 代表一定有一个字符

[]:代表一定有一个括号内的字符

[-]代表一定是连续字符中的一个.例如：[0-9]代表一个是0-9中的一个字符。前提字符必须是连续的

[^]:^表示反向选择例如[^abc]代表一个非 abc 的字符

举例：

```
[root@ localhost ~]# LANG=C        <==由于与编码有关（字符连续），先配置语系
找出 /etc/ 底下以 cron 为开头的档名
[root@ localhost ~]# ll -d /etc/cron*    <==加上 -d 是为了仅显示目录而已
找出 /etc/ 底下文件名『刚好是五个字母』的文件名
[root@ localhost ~]# ll -d /etc/?????
找出 /etc/ 底下文件名含有数字的文件名
[root@ localhost ~]# ll -d /etc/*[0-9]*
找出 /etc/ 底下，档名开头非为小写字母的文件名：
[root@ localhost ~]# ll -d /etc/[^a-z]*
```

20

vim 与 vi 常用命令

语系编码转换：iconv

vi 是个文本编辑器，所有 UNIX Like 系统都会内置这个编辑器

vim 是 vi 的强加版，其具有程序编辑的能力，可以主动以字体颜色辨识语法的正确性。

常用命令

移动光标的方法

h 或向左箭头键(←)	光标向左移动一个字符
j 或向下箭头键(↓)	光标向下移动一个字符
k 或向上箭头键(↑)	光标向上移动一个字符
l 或向右箭头键(→)	光标向右移动一个字符
如果你将右手放在键盘上的话，你会发现 hjkl是排列在一起的，因此可以使用这四个按钮来移动光标。如果想要进行多次移动的话，例如向下移动 30行，可以使用 "30j" 或 "30↓"的组合按键，亦即加上想要进行的次数(数字)后，按下动作即可.	
[Ctrl] + [f]	屏幕[向下]移动一页，相当于 [Page Down]按键 (常用)
[Ctrl] + [b]	屏幕[向上]移动一页，相当于 [Page Up]按键 (常用)
[Ctrl] + [d]	屏幕[向下]移动半页
[Ctrl] + [u]	屏幕[向上]移动半页
+	光标移动到非空格符的下一列

-	光标移动到非空格符的上一列
n<space>	那个 n 表示[数字]，例如 20。按下数字后再按空格键，光标会向右移动这一行的 n 个字符。例如 20<space>则光标会向后面移动 20 个字符距离。
0 或功能键[Home]	这是数字[0]：移动到这一行的最前面字符处 (常用)
$ 或功能键[End]	移动到这一行的最后面字符处(常用)
H	光标移动到这个屏幕的最上方那一行的第一个字符
M	光标移动到这个屏幕的中央那一行的第一个字符
L	光标移动到这个屏幕的最下方那一行的第一个字符
G	移动到这个档案的最后一行(常用)
nG	n 为数字。移动到这个档案的第 n行。例如 20G 则会移动到这个档案的第 20行(可配合 :set nu)
gg	移动到这个档案的第一行，相当于 1G啊. (常用)
n<Enter>	n 为数字。光标向下移动 n行(常用)

搜寻与取代

/word	向光标之下寻找一个名称为 word的字符串。例如要在档案内搜寻 asde 这个字符串，就输入 /asde即可. (常用)
?word	向光标之上寻找一个字符串名称为 word的字符串。
n	这个 n 是英文按键。代表[重复前一个搜寻的动作]。举例来说，如果刚刚我们执行 /asde去向下搜寻 asde 这个字符串，则按下 n后，会向下继续搜寻下一个名称为 asde 的字符串。如果是执行 ?asde的话，那么按下 n 则会向上继续搜寻名称为 asde的字符串.
N	这个 N 是英文按键。与 n刚好相反，为[反向]进行前一个搜寻动作。例如 /asde后，按下 N 则表示[向上]搜寻 asde。
使用 /word 配合 n 及 N是非常有帮助的.可以让你重复的找到一些你搜寻的关键词.	
:n1,n2s/word1/word2/g	n1 与 n2为数字。在第 n1 与 n2行之间寻找 word1 这个字符串，并将该字符串取代为 word2 .举例来说，在 100到 200 行之间搜寻 asde并取代为 ASDE 则: [:100,200s/asde/ASDE/g]。(常用)
:1,$s/word1/word2/g	从第一行到最后一行寻找 word1字符串，并将该字符串取代为 word2 .(常用)
:1,$s/word1/word2/gc	从第一行到最后一行寻找 word1字符串，并将该字符串取代为 word2 .且在取代前显示提示字符给用户确认 (confirm)是否需要取代.(常用)

删除复制与粘贴

x, X	在一行字当中，x 为向后删除一个字符 (相当于 [del]按键)， X为向前删除一个字符(相当于[backspace]亦即是退格键) (常用)
nx	n 为数字，连续向后删除 n个字符。举例来说，我要连续删除 10 个字符， [10 x]。
dd	删除游标所在的那一整列(常用)
ndd	n 为数字。删除光标所在的向下 n列，例如 20dd 则是删除 20列 (常用)
d1 G	删除光标所在到第一行的所有数据
dG	删除光标所在到最后一行的所有数据
d$	删除游标所在处，到该行的最后一个字符
d0	那个是数字的 0 ，删除游标所在处，到该行的最前面一个字符
yy	复制游标所在的那一行(常用)
nyy	n 为数字。复制光标所在的向下 n列，例如 20yy 则是复制 20列(常用)
y1G	复制游标所在列到第一列的所有数据
yG	复制游标所在列到最后一列的所有数据

y0	复制光标所在的那个字符到该行行首的所有数据
y$	复制光标所在的那个字符到该行行尾的所有数据
p, P	p 为将已复制的数据在光标下一行贴上，P则为贴在游标上一行. 举例来说，目前光标在第 20行，且已经复制了 10 行数据。则按下 p后，那 10 行数据会贴在原本的 20行之后，亦即由 21 行开始贴。但如果是按下 P呢? 那么原本的第 20 行会被推到变成 30行。 (常用)
J	将光标所在列与下一列的数据结合成同一列
c	重复删除多个数据，例如向下删除 10行，[10cj]
u	复原前一个动作。(常用)
[Ctrl]+r	重做上一个动作。(常用)
.	意思是重复前一个动作的意思。如果你想要重复删除、重复贴上等等动作，按下小数点[.]就好了. (常用)

进入插入和替换

i, I	进入插入模式(Insert mode): i 为[从目前光标所在处插入]， I为[在目前所在行的第一个非空格符处开始插入]。 (常用)
a, A	进入插入模式(Insert mode): a 为[从目前光标所在的下一个字符处开始插入]， A为[从光标所在行的最后一个字符处开始插入]。(常用)
o, O	进入插入模式(Insert mode): 这是英文字母 o 的大小写。o为[在目前光标所在的下一行处插入新的一行]； O为在目前光标所在处的上一行插入新的一行.(常用)
r, R	进入取代模式(Replace mode): r 只会取代光标所在的那一个字符一次；R会一直取代光标所在的文字，直到按下 ESC为止；(常用)
上面这些按键中，在 vi 画面的左下角处会出现[--INSERT--]或[--REPLACE--]的字样。特别注意的是，我们上面也提过了，你想要在档案里面输入字符时，一定要在左下角处看到 INSERT或 REPLACE 才能输入	
[Esc]	退出编辑模式，回到一般模式中(常用)

存储离开与文件保存

:w	将编辑的数据写入硬盘档案中(常用)
:w!	若文件属性为[只读]时，强制写入该档案。不过，到底能不能写入，还是跟你对该档案的档案权限有关啊.
:q	离开 vi (常用)
:q!	若曾修改过档案，又不想储存，使用 !为强制离开不储存档案。
注意一下啊，那个惊叹号 (!)在 vi 当中，常常具有[强制]的意思~	
:wq	储存后离开，若为 :wq! 则为强制储存后离开 (常用)
ZZ	若档案没有更动，则不储存离开，若档案已经被更动过，则储存后离开.
:w [filename]	将编辑的数据储存成另一个档案（类似另存新档）
:r [filename]	在编辑的数据中，读入另一个档案的数据。亦即将 [filename]这个档案内容加到游标所在行后面
:n1,n2 w [filename]	将 n1 到 n2的内容储存成 filename 这个档案。
:! command	暂时离开 vi 到指令列模式下执行 command 的显示结果.例如 [:! ls /home]即可在 vi当中察看 /home 底下以 ls输出的档案信息.

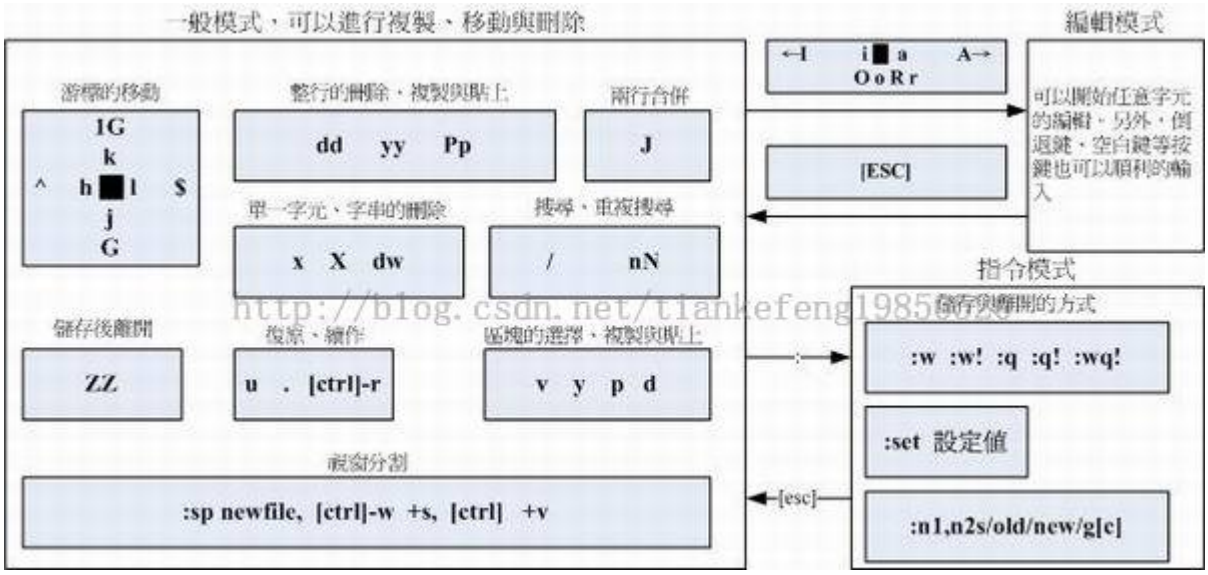
vim环境设定与参数

:set nu :set nonu	就是设定与取消行号啊.
:set hlsearch :set nohlsearch	hlsearch 就是 high light search(高亮度搜寻)。这个就是设定是否将搜寻的字符串反白的设定值。默认值是 hlsearch
:set autoindent :set noautoindent	是否自动缩排？autoindent就是自动缩排。
:set backup	是否自动储存备份档？一般是 nobackup的，如果设定 backup 的话，那么当你更动任何一个档案时，则源文件会被另存成一个档名为 filename~的档案。举例来说，我们编辑 hosts ，设定 :set backup，那么当更动 hosts 时，在同目录下，就会产生 hosts~文件名的档案，记录原始的 hosts 档案内容
:set ruler	还记得我们提到的右下角的一些状态栏说明吗？这个 ruler就是在显示或不显示该设定值的 .
:set show mode	这个则是，是否要显示 --INSERT--之类的字眼在左下角的状态栏。
:set backspac	一般来说，如果我们按下 i 进入编辑模式后，可以利用退格键 (backspace) 来删除任意字符的。但是，某些 distribution则不许如此。此时，我们就可以透过 backspace 来设定当 ba

e=(0 12)	ckspace为 2 时，就是可以删除任意值；0或 1 时，仅可删除刚刚输入的字符，而无法删除原本就已经存在的文字了.
:set a ll	显示目前所有的环境参数设定值。
:set	显示与系统默认值不同的设定参数，一般来说就是你有自行变动过的设定参数 .
:synt ax on :synt ax off	是否依据程序相关语法显示不同颜色举例来说，在编辑一个纯文本档时，如果开头是以 #开始，那么该行就会变成蓝色。如果你懂得写程序，那么这个 :syntax on 还会主动的帮你除错呢.但是，如果你仅是编写纯文本档案，要避免颜色对你的屏幕产生的干扰，则可以取消这个设定。
:set b g=da rk :set bg=li ght	可用以显示不同的颜色色调，预设是[light]。如果你常常发现批注的字体深蓝色实在很不容易看，那么这里可以设定为 dark.看看，会有不同的样式呢.

说明：如果不想每次都进行设置 VIM 环境，可以讲环境命令添加到~/.vimrc 中(此文件需自行创建) [root@local host ~]# vim ~/.vimrc set hlsearch set backspace=2 set autoindent set ruler set showmode set nu set bg=dark syntax on

以下图列出常用命令


一般模式，可以進行複製、移動與刪除
編輯模式
游標的移動
1G
k
^ h l $
j
G
整行的刪除，複製與貼上
dd yy Pp
兩行合併
J
單一字元、字串的刪除
x X dw
搜尋、重複搜尋
/ nN
儲存後離開
ZZ
複原、續作
u . [ctrl]-r
區塊的選擇、複製與貼上
v y p d
視窗分割
:sp newfile, [ctrl]-w +s, [ctrl] +v
←I i a A→
O o R r
[ESC]
可以開始任意字元的編輯，另外，倒退鍵，空白鍵等按鍵也可以順利的輸入
指令模式
:
[esc]
儲存與離開的方式
:w :w! :q :q! :wq!
:set 設定值
:n1,n2s/old/new/g[c]

语系编码转换(iconv)

经常通过文本编辑器查看文字时会出现乱码，出现乱码的主要原因是环境的语系编码与文件的编码不一致导致的，比如系统语系是繁体中文(big5),文件语系是简体中文(gb2312)。 可以通过2种方式解决问题 **1：设置系统语系编码** [root@localhost ~]# LANG GB2312 **2：语系转为和系统一致 语法**：iconv --list iconv -f 原编码 -t 新编码 filename [-o newfile] **选项与参数：** --list:列出支持的语系 -f:原编码 -t:新编码 -o file:保留源文件 file 为新文件名

21

文件备份|还原

dump 备份

restore 还原

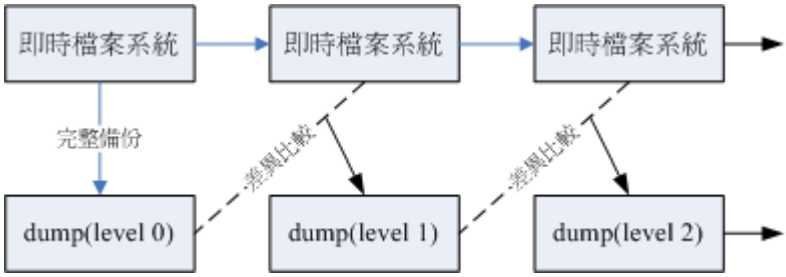
dd 数据备份

mkisofs 镜像文件制作

dump 备份

dump 主要用于备份真个文件系统备份，虽然也可以备份单一目录，但是对目录的支持不足，单一目录还是建议使用打包压缩的方式进行备份

dump 另一个只要功能就是制定等级，也就是可以进行增量备份。



dump 等级分为0~9 10个等级，0是完全备份，1是在0的基础上进行增量备份，依次类推

当待备份的数据为单一文件系统

可以利用了level 0~9进行备份，同时可以使用 dump 完整功能

当待备份的数据只是目录，并非单一文件系统

限制：

所有备份数据必须都在该目录下

仅能使用 level 0 进行数据备份

不支持-u 参数，即无法创建/etc/dumpdates 这个 level 备份的时间记录文件

语法：dump [-Suvj] [-level] [-f 备份文件]待备份数据

dump -W

选项与参数：

-S:仅列出后面的待备份数据需要多少磁盘空间才能够备份完毕

-u:将这次备份记录到/etc/dumpdates 文件中

-v:将 dump 文件过程显示出来

-j:加入 bzip2的支持，将数据进行压缩，默认压缩等级2

-level:备份等级0~9

-f:备份文件

-W:列出在/etc/fstab 里面的具有 dump 设置的分区是否有过备份

举例1：备份挂载到/boot 文件系统 level -0

```
[root@localhost ~]# dump -S /boot
16752640
[root@localhost ~]# dump -u -0 -f /root/boot.dump.0 /boot
```

```
  DUMP: Date of this level 0 dump: Fri Feb 28 15:05:56 2014
  DUMP: Dumping /dev/sda1 (/boot) to /root/boot.dump.0
  DUMP: Label: /boot
  DUMP: Writing 10 Kilobyte records
  DUMP: mapping (Pass I) [regular files]
  DUMP: mapping (Pass II) [directories]
  DUMP: estimated 16360 blocks.
  DUMP: Volume 1 started with block 1 at: Fri Feb 28 15:05:56 2014
  DUMP: dumping (Pass III) [directories]
  DUMP: dumping (Pass IV) [regular files]
  DUMP: Closing /root/boot.dump.0
  DUMP: Volume 1 completed at: Fri Feb 28 15:05:58 2014
  DUMP: Volume 1 16440 blocks (16.05MB)
  DUMP: Volume 1 took 0:00:02
  DUMP: Volume 1 transfer rate: 8220 kB/s
  DUMP: 16440 blocks (16.05MB) on 1 volume(s)
  DUMP: finished in 2 seconds, throughput 8220 kBytes/sec
  DUMP: Date of this level 0 dump: Fri Feb 28 15:05:56 2014
  DUMP: Date this dump completed:  Fri Feb 28 15:05:58 2014
  DUMP: Average transfer rate: 8220 kB/s
  DUMP: DUMP IS DONE
[root@localhost ~]# cat /etc/dumpdates
/dev/sda1 0 Fri Feb 28 15:05:56 2014 +0800
=>可以看出 etc/dumpdates 记录着这次备份信息
```

举例2：查看文件系统备份记录

```
[root@localhost ~]# dump -W
Last dump(s) done (Dump '>' file systems):
> /dev/sda2     (     /) Last dump: never
> /dev/sda3     ( /home) Last dump: never
  /dev/sda1     ( /boot) Last dump: Level 0, Dat
> /dev/sda6     (/mnt/sda6) Last dump: never
=>可以看出 sda1已经进行了 level0备份，其他还未备份
```

举例3:增量备份 level 1

```
[root@localhost ~]# dd if=/dev/zero of=/boot/bigfile.img bs=1M count=20
20+0 records in
20+0 records out
20971520 bytes (21 MB) copied, 0.320717 seconds, 65.4 MB/s
=>先创建一个20M 左右的文件
[root@localhost ~]# dump -u -1 -f /root/boot.dump.1 /boot
  DUMP: Date of this level 1 dump: Fri Feb 28 15:17:51 2014
  DUMP: Date of last level 0 dump: Fri Feb 28 15:05:56 2014
  DUMP: Dumping /dev/sda1 (/boot) to /root/boot.dump.1
```

```
  DUMP: Label: /boot
  DUMP: Writing 10 Kilobyte records
  DUMP: mapping (Pass I) [regular files]
  DUMP: mapping (Pass II) [directories]
  DUMP: estimated 20543 blocks.
  DUMP: Volume 1 started with block 1 at: Fri Feb 28 15:17:52 2014
  DUMP: dumping (Pass III) [directories]
  DUMP: dumping (Pass IV) [regular files]
  DUMP: Closing /root/boot.dump.1
  DUMP: Volume 1 completed at: Fri Feb 28 15:17:53 2014
  DUMP: Volume 1 20580 blocks (20.10MB)
  DUMP: Volume 1 took 0:00:01
  DUMP: Volume 1 transfer rate: 20580 kB/s
  DUMP: 20580 blocks (20.10MB) on 1 volume(s)
  DUMP: finished in 1 seconds, throughput 20580 kBytes/sec
  DUMP: Date of this level 1 dump: Fri Feb 28 15:17:51 2014
  DUMP: Date this dump completed:  Fri Feb 28 15:17:53 2014
  DUMP: Average transfer rate: 20580 kB/s
  DUMP: DUMP IS DONE
[root@localhost ~]# cat /etc/dumpdates
/dev/sda1 0 Fri Feb 28 15:05:56 2014 +0800
/dev/sda1 1 Fri Feb 28 15:17:51 2014 +0800
=>这次配备写入备份记录中
[root@localhost ~]# dump -W
Last dump(s) done (Dump '>' file systems):
> /dev/sda2     (     /) Last dump: never
> /dev/sda3     ( /home) Last dump: never
  /dev/sda1     ( /boot) Last dump: Level 1, Date Fri Feb 28 15:17:51 2014
> /dev/sda6     (/mnt/sda6) Last dump: never
[root@localhost ~]# ll /root/boot*
-rw-r--r-- 1 root root 16834560 02-28 15:05 /root/boot.dump.0
-rw-r--r-- 1 root root 21073920 02-28 15:17 /root/ boot.dump.1
=> boot.dump.1大小约为20M，可见是增量备份
```

举例4：单一目录进行备份

```
[root@localhost ~]# dump -0 -f /root/etc.dump /etc
  DUMP: Date of this level 0 dump: Fri Feb 28 15:23:39 2014
  DUMP: Dumping /dev/sda2 (/ (dir etc)) to /root/etc.dump
DUMP: Label: /
  DUMP: Writing 10 Kilobyte records
  DUMP: mapping (Pass I) [regular files]
  DUMP: mapping (Pass II) [directories]
  DUMP: estimated 177675 blocks.
  DUMP: Volume 1 started with block 1 at: Fri Feb 28 15:23:41 2014
  DUMP: dumping (Pass III) [directories]
```

```
  DUMP: dumping (Pass IV) [regular files]
  DUMP: Closing /root/etc.dump
  DUMP: Volume 1 completed at: Fri Feb 28 15:24:23 2014
  DUMP: Volume 1 188600 blocks (184.18MB)
  DUMP: Volume 1 took 0:00:42
  DUMP: Volume 1 transfer rate: 4490 kB/s
  DUMP: 188600 blocks (184.18MB) on 1 volume(s)
  DUMP: finished in 42 seconds, throughput 4490 kBytes/sec
  DUMP: Date of this level 0 dump: Fri Feb 28 15:23:39 2014
  DUMP: Date this dump completed:  Fri Feb 28 15:24:23 2014
  DUMP: Average transfer rate: 4490 kB/s
  DUMP: DUMP IS DONE
[root@localhost ~]# ll /root/etc.dump
-rw-r--r-- 1 root root 193126400 02-28 15:24 /root/etc.dump
```

restore 还原

dump 备份的文件由 restore 进行还原

语法：

查看 dump 文件：restore -t [-f dumpfile] [-h]

比较 dump 与实际文件：restore -C [-f dumpfile] -D 挂载点

进入互动模式(还原单个文件)：restore -i [-f dumpfile]

还原整个文件系统：restore -r [-f dumpfile]

选项与参数：

相关的各种模式，各种模式无法混用.例如不可以写 -tC

-t:此模式用在察看 dump 起来的备份档中含有什么重要数据！类似 tar -t 功能；

-C:此模式可以将 dump 内的数据拿出来跟实际的文件系统做比较，最终会列出[在 dump 文件内有记录 的，且目前文件系统不一样]的文件；

-i:进入互动模式，可以仅还原部分文件，用在 dump 目录时的还原

-r:将整个 filesystem 还原的一种模式，用在还原针对文件系统的 dump 备份；

其他较常用到的选项功能：

-h:察看完整备份数据中的 inode 与文件系统 label 等信息

-f:后面就接你要处理的那个 dump 文件

-D:与 -C 进行搭配，可以查出后面接的挂载点与 dump 内有不同的文件

举例1：查看 dump 备份文件

```
[root@localhost ~]# restore -t -f /root/boot.dump.0
Dump   date: Fri Feb 28 15:05:56 2014
Dumped from: the epoch
Level 0 dump of /boot on localhost.localdomain:/dev/sda1
Label: /boot
        2     .
       11     ./lost+found
    10041     ./grub
    10059     ./grub/grub.conf
.......
       14     ./System.map-2.6.18-371.el5
       15     ./config-2.6.18-371.el5
       16     ./symvers-2.6.18-371.el5.gz
       17     ./vmlinuz-2.6.18-371.el5
```

举例2：比较文件差异

```
[root@localhost ~]# mv /boot/message /boot/message-back
[root@localhost ~]# restore -C -f /root/boot.dump.0 -D /boot
Dump   date: Fri Feb 28 15:05:56 2014
Dumped from: the epoch
Level 0 dump of /boot on localhost.localdomain:/dev/sda1
Label: /boot
filesys = /boot
restore: unable to stat ./message: No such file or directory
Some files were modified!  1 compare errors
```

举例3：还原整个文件系统

```
[root@localhost ~]# dd if=/dev/zero of=/home/newfile bs=1M count=200
200+0 records in
200+0 records out
209715200 bytes (210 MB) copied, 3.83857 seconds, 54.6 MB/s
[root@localhost ~]# mkfs -t ext3 /home/newfile
mke2fs 1.39 (29-May-2006)
/home/newfile is not a block special device.
.....
180 days, whichever comes first.  Use tune2fs -c or -i to override.
[root@localhost ~]# mount -o loop /home/newfile /mnt
[root@localhost ~]# df -h
文件系统            容量  已用 可用 已用% 挂载点
/dev/sda2             9.5G  4.4G  4.7G  49% /
/dev/sda3             4.8G  339M  4.2G   8% /home
/dev/sda1              99M   42M   53M  45% /boot
tmpfs                1014M     0 1014M   0% /dev/shm
/home/newfile         194M  5.6M  179M   4% /mnt
=>创建一个文件挂载到 mnt 下
[root@localhost ~]# cd /mnt
[root@localhost mnt]# restore -r -f /root/boot.dump.0
restore: ./lost+found: File exists
[root@localhost mnt]# ll
总计 16149
-rw-r--r-- 1 root root    70400 10-01 21:10 config-2.6.18-371.el5
drwxr-xr-x 2 root root     1024 02-18 09:51 grub
-rw------- 1 root root  2748313 02-18 09:46 initrd-2.6.18-371.el5.img
drwx------ 2 root root    12288 02-14 18:00 lost+found
-rw-r--r-- 1 root root    80032 2009-03-13 message
-rw------- 1 root root    27676 02-28 15:54 restoresymtable
-rw-r--r-- 1 root root   117436 10-01 21:10 symvers-2.6.18-371.el5.gz
-rw-r--r-- 1 root root   996296 10-01 21:10 System.map-2.6.18-371.el5
-rw-r--r-- 1 root root 10485760 02-28 13:25 testing.img
-rw-r--r-- 1 root root  1912148 10-01 21:10 vmlinuz-2.6.18-371.el5
=>还原 level 0备份
```

```
[root@localhost mnt]# restore -r -f /root/boot.dump.1
[root@localhost mnt]# ll
总计 36711
-rw-r--r-- 1 root root 20971520 02-28 15:17 bigfile.img
-rw-r--r-- 1 root root    70400 10-01 21:10 config-2.6.18-371.el5
drwxr-xr-x 2 root root     1024 02-18 09:51 grub
-rw------- 1 root root  2748313 02-18 09:46 initrd-2.6.18-371.el5.img
drwx------ 2 root root    12288 02-14 18:00 lost+found
## -rw-r--r-- 1 root root    80032 2009-03-13 message
------- 1 root root    27724 02-28 15:55 restoresymtable
-rw-r--r-- 1 root root   117436 10-01 21:10 symvers-2.6.18-371.el5.gz
-rw-r--r-- 1 root root   996296 10-01 21:10 System.map-2.6.18-371.el5
-rw-r--r-- 1 root root 10485760 02-28 13:25 testing.img
-rw-r--r-- 1 root root  1912148 10-01 21:10 vmlinuz-2.6.18-371.el5
=>还原 level 1备份可以看到多了 bigfile.img 这个增量文件
```

dd

dd 功能不仅限于创建文件，更多功能在于“备份”，cp,dump 只是简单的文件数据拷贝，而 dd 可以读取设备的所有内容，比如 superblock ,boot sector,mete data 等

语法：dd if=”input file” of=”output file” bs=”block” count=”number”

选项与参数：

if:输入文件，也可以是设备

of:输出文件，也可以是设备

bs:每个 block 的大小，默认是512 K

count:block 数量

举例1.文件备份

```
[root@localhost ~]# dd if=~/.bashrc of=/tmp/bashrc
0+1 records in
0+1 records out
176 bytes (176 B) copied, 7.3142e-05 seconds, 2.4 MB/s
[root@localhost ~]# ll /tmp/bashrc
-rw-r--r-- 1 root root 176 02-28 16:17 /tmp/bashrc
```

举例2：文件系统备份

```
[root@localhost ~]# dd if=/dev/sda1 of=/tmp/boot.dd bs=1M
101+1 records in
101+1 records out
106896384 bytes (107 MB) copied, 9.60492 seconds, 11.1 MB/s
[root@localhost ~]# ll /tmp/boot.dd
-rw-r--r-- 1 root root 106896384 02-28 16:19 /tmp/boot.dd
```

举例3：文件系统还原

```
[root@localhost ~]# dd if=/tmp/boot.dd of=/dev/sda1 bs=1M
```

举例4.文件系统完全复制

Dump 备份时，我们需要先用 Dump 将文件系统备份，然后创建新的文件系统，格式化，再将备份文件还原到新的文件系统。

使用 dd 可以不用格式化，就可以完全复制一个文件系统，因为 dd 将 uperblock ,boot sector,mete data 等信息都进行复制，格式化要做的不也正是这些事吗

```
[root@bogon ~]# fdisk /dev/sda
…….
```

```
Command (m for help): n
......
Command (m for help): P
......
   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391   83  Linux
......
/dev/sda7            2116        2134      152586   83  Linux

Command (m for help): w
......
[root@bogon ~]# partprobe
=>创建完分区
[root@bogon ~]# dd if=/dev/sda1 of=/dev/sda7
208782+0 records in
208782+0 records out
106896384 bytes (107 MB) copied, 23.5363 seconds, 4.5 MB/s
[root@bogon ~]# mount /dev/sda7 /mnt
[root@bogon ~]# ll /mnt
总计 5838
-rw-r--r-- 1 root root   70400 10-01 21:10 config-2.6.18-371.el5
drwxr-xr-x 2 root root    1024 02-18 20:26 grub
-rw------- 1 root root 2748762 02-27 19:45 initrd-2.6.18-371.el5.img
drwx------ 2 root root   12288 02-19 03:59 lost+found
-rw-r--r-- 1 root root   80032 2009-03-13 message
-rw-r--r-- 1 root root  117436 10-01 21:10 symvers-2.6.18-371.el5.gz
-rw-r--r-- 1 root root  996296 10-01 21:10 System.map-2.6.18-371.el5
-rw-r--r-- 1 root root 1912148 10-01 21:10 vmlinuz-2.6.18-371.el5
=> /mnt和/boot 下的内容一样 并且没有进行格式化
```

mkisofs(镜像文件备份)

语法：mkisofs [-o 镜像文件] [-rv] [-m file]待备份的文件 [-V vol] - graft-point isodir=sysdir

选项与参数:

-o:镜像文件

-r:产生 UNIX/Linux 支持的文件数据

-v:显示构建 ISO 的过程

-m:排除的文件

-V:卷标名称

-graft-point:目录对照名称，如果不进行指定所以的信息都会保持在根目录

举例:

```
[root@bogon ~]# mkisofs -o /tmp/system.img -r -m /home/lost+found -V 'tkf_file' -graft-point /root=/root /home=/home /
[root@bogon ~]# mount -o loop /tmp/system.img /mnt
[root@bogon ~]# ll /mnt
dr-xr-xr-x 114 root root 34816 03-01 14:31 etc
dr-xr-xr-x   3 root root  2048 03-01 14:31 home
dr-xr-xr-x  18 root root  4096 03-01 14:31 root
```

22

文件压缩

机器语言与程序语言

对于机器来说只能识别0,1，我们如果让机器运行必须输入机器能够识别的语言，可是机器语言不利于人们使用可理解，因此科学家就开发出人类能看的懂的程序语言，然后再创造出“编译器”将程序语言转换为机器语言。

压缩的简单原理

我们都知道1 byte=8 bit. 比如，对于这1这个数字来说可以表示为0000 0001，前7个 bit 都是“空的”只有最后一个 bit，有实际意义。压缩的原理就是通过复杂的计算方式将这个“空的“内容尽可能的去掉以减少文件的存储空间

常见压缩|打包命令

Linux 常见的压缩命令式 gzip,bzip2,这些压缩命令都是针对于一个文件进行压缩，因此当要压缩很多文件时，就需要先进行打包（tar）然后再进行压缩。

*.Z :compress 程序压缩的文件
*.gz:gzip 程序压缩的文件
*.bz2:bzip2程序压缩的文件
*.tar:打包文件，并未进行压缩
*.tar.gz:打包文件并以 gzip 程序压缩打包文件
*tar.bz2: 打包文件并以 bzip2程序压缩打包文件

gzip

gzip 可以解开 compress,zip,gzip 等软件压缩的文件

语法：gzip[cdtv#] 文件名
选项与参数：
-c: 将压缩数据输出到屏幕上
-d:解压缩
-t:可以检验一个压缩文件的一致性，看文件有无错误
-v:显示源文件/压缩文件的压缩比等信息
-#:压缩等级，-1最快，-9最慢，默认值时-6

举例1：压缩文件

```
[root@bogon ~]# cp /etc/man.config /tmp/man.config
[root@bogon ~]# gzip -v /tmp/man.config
/tmp/man.config:        56.1% -- replaced with /tmp/man.config.gz
[root@bogon ~]# ll /etc/man.config /tmp/man.config.gz
-rw-r--r-- 1 root root 4617 2012-05-30 /etc/man.config
-rw-r--r-- 1 root root 2057 02-27 22:26 /tmp/man.config.gz
```

举例2：解压缩

```
[root@bogon ~]# gzip -d /tmp/man.config.gz
[root@bogon ~]# ll  /tmp/man.config
-rw-r--r-- 1 root root 4617 02-27 22:26 /tmp/man.config
```

举例3：数据流重定向(压缩后保留原来文件)

```
[root@bogon ~]# gzip -c /tmp/man.config > /tmp/man.config.gz
[root@bogon ~]# ll /tmp/man.config /tmp/man.config.gz
-rw-r--r-- 1 root root 4617 02-27 22:26 /tmp/man.config
-rw-r--r-- 1 root root 2057 02-27 22:31 /tmp/man.config.gz
```

可以 zcat 来读取由 gzip 压缩的文件

[root@bogon ~]# zcat /tmp/man.config.gz

bzip2

bzip2的压缩比比 gzip 还要好

语法：bzip2[-cdkzv#] 文件名

选项与参数：

-c:将压缩数据输出到屏幕上

-d:解压缩

-k:保留原始文件

-z:压缩

-v:显示源文件/压缩文件的压缩比等信息

-#:压缩等级，-1最快，-9最慢

可以 bzcat 来读取由 bzip2压缩的文件

tar

语法：

打包与压缩：tar [-j|-z] [-cv] [-f 新建的文件名] filename

查看文件名：tar [-j|-z] [-tv] [-f 新建的文件名]

解压缩：tar [-j|-z] [-xv] [-f 新建的文件名] [-C 目录]

选项与参数：

-c:新建打包文件

-t:查看打包文件内容

-x:加压缩打包文件

-j:使用 bzip2进行压缩/解压缩

-z:使用该 gzip 进行压缩/解压缩

-v:在压缩过程中，将正在处理的文件名显示出来

-f filename:需要被压缩成(解压缩)的文件名

-C:解压缩到的目录

-p:保留备份数据的原有权限和属性

-P:保留绝对路径

--exclude=File:在压缩中不将 FILE 打包

--newer-mtime=”时间”： 打包比指定时间新的文件

举例1:对文件打包压缩

```
[root@localhost ~]# tar -jcv -f /root/etc.tar.bz2 /etc
……压缩文件信息
[root@localhost ~]# tar -zcv -f /root/etc.tar.gz /etc
……压缩文件信息
[root@localhost ~]# ll --block-size=M /root/etc.tar.bz2 /root/etc.tar.gz ;du -sm /etc
-rw-r--r-- 1 root root 10M 02-28 10:42 /root/etc.tar.bz2
-rw-r--r-- 1 root root 16M 02-28 10:43 /root/etc.tar.gz
179     /etc
```

可以看到压缩后，文件小了很多

举例2：查看打包压缩文件内容

```
[root@localhost ~]# tar -ztv -f /root/etc.tar.gz |grep 'shadow*'
-r-------- root/root      1352 2014-02-14 10:36:09 etc/shadow
-r-------- root/root       657 2014-02-14 10:36:09 etc/gshadow
-r-------- root/root       648 2014-02-14 10:36:09 etc/gshadow-
-r-------- root/root      1352 2014-02-14 10:36:09 etc/shadow-
```

举例3：解压缩

```
[root@localhost ~]# tar -jxv -f /root/etc.tar.bz2 -C /tmp
……解压缩文件信息
[root@localhost ~]# ll -d /tmp/etc/
drwxr-xr-x 114 root root 12288 02-28 10:15 /tmp/etc/
```

当不使用绝对路径压缩时，解压后则解压到指定路径下，如压缩文件/etc,解压后直接放在了/tmp/etc 使用绝对路径压缩，则在解压缩后可以使用文件的绝对路径解压缩到文件的原来目录

举例4：打包目录，但排除一些文件

```
[root@localhost ~]# tar -jcv -f /root/system.tar.bz2 --exclude=/root/etc* --exclude=/root/system.tae.bz2 /root /etc
……压缩文件信息
```

```
[root@localhost ~]# ll /root/system.tar.bz2
-rw-r--r-- 1 root root 10531659 02-28 11:19 /root/system.tar.bz2
```

23

内存交换空间的构建

我们知道 CPU 计算与数据的存储都会使用到内存，使用内存可以大大减少从磁盘读取的时间，但是当物理内存不足时，就需要暂时将用不到的程序和数据挪到内存交换空间(swap)

作法：

1.创建分区（fdisk ,文件）

2.格式化为 swap

3.启动

4.查看

创建分区

举例

```
[root@bogon ~]# fdisk /dev/sda

The number of cylinders for this disk is set to 2610.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): p

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391   83  Linux
/dev/sda2              14        1288    10241437+  83  Linux
/dev/sda3            1289        1925     5116702+  83  Linux
/dev/sda4            1926        2610     5502262+   5  Extended
/dev/sda5            1926        2052     1020096   82  Linux swap / Solaris
/dev/sda6            2053        2115      506016   83  Linux

Command (m for help): t
Partition number (1-6): 6
Hex code (type L to list codes): 82
Changed system type of partition 6 to 82 (Linux swap / Solaris)

Command (m for help): p

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391   83  Linux
/dev/sda2              14        1288    10241437+  83  Linux
/dev/sda3            1289        1925     5116702+  83  Linux
/dev/sda4            1926        2610     5502262+   5  Extended
```

```
/dev/sda5          1926        2052     1020096   82  Linux swap / Solaris
/dev/sda6          2053        2115      506016   82  Linux swap / Solaris

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: Re-reading the partition table failed with error 16: 设备或资源忙.
The kernel still uses the old table.
The new table will be used at the next reboot.
Syncing disks.
part[root@bogon ~]# partprobe
这里需要在进行设置下 system ID
```

格式化

语法：mkswap 设备名称

举例

```
[root@bogon ~]# mkswap /dev/sda6

Setting up swapspace version 1, size = 518156 kB
```

启动|关闭

语法：swapon [-s]设备名称

swapoff 设备名称

选项与参数：

-s:查看所有 swap 文件系统

举例1：启动 swap

```
[root@bogon ~]# swapon /dev/sda6
```

举例2：查看所有 swap

```
[root@bogon ~]# swapon -s
Filename                        Type            Size    Used    Priority
/dev/sda5                       partition       1020088 0       -1
/dev/sda6                       partition       506008  0       -2
```

查看

语法：free

举例

```
             total       used       free     shared    buffers     cached
Mem:       2074972    1380996     693976          0     106740    1000288
-/+ buffers/cache:     273968    1801004
Swap:      1526096          0    1526096
```

可以看到 Swap 空间增加1526096

24

磁盘挂载与卸载

文件系统的格式化完毕后，需要将文件系统挂载到目录树上我们才可以使用，如果你要用来挂载的目录里面并不是空的，那么挂载了文件系统之后，原目录下的东西就会暂时的消失。举个例子来说，假设你的 /home 原本与根目录 (/) 在同一个文件系统中，底下原本就有 /home/test 与 /home/vbird 两个目录。然后你想要加入新的硬盘，并且直接挂载 /home 底下，那么当你挂载上新的分割槽时，则 /home 目录显示的是新分割槽内的数据，至于原先的 test 与 vbird 这两个目录就会暂时的被隐藏掉了！并不是被覆盖掉，而是暂时的隐藏了起来，等到新分割槽被卸除之后，则 /home 原本的内容就会再次的跑出来

磁盘挂载

语法：

[root@www ~]# mount -a

[root@www ~]# mount [-l]

[root@www ~]# mount [-t 文件系统] [-LLabel 名] [-o 额外选项] 装置文件名 挂载点

选项与参数：

-a ： 依照配置文件/etc/fstab 的数据将所有未挂载的磁盘都挂载上来

-l ： 单纯的输入 mount 会显示目前挂载的信息。加上-l 可增列 Label 名称!

-t ： 与 mkfs 的选项非常类似的，可以加上文件系统种类来指定欲挂载的类型。常见的 Linux 支持类型有：ext 2, ext3, vfat, reiserfs, iso9660(光盘格式),nfs, cifs,smbfs(此三种为网络文件系统类型)

-n ： 在默认的情况下，系统会将实际挂载的情况实时写入 /etc/mtab 中，以利其他程序的运行。但在某些情况下(例如单人维护模式)为了避免问题，会刻意不写入。此时就得要使用这个 -n 的选项了。

-L ： 系统除了利用装置文件名(例如 /dev/hdc6) 之外，还可以利用文件系统的标头名称 (Label)来进行挂载。最好为你的文件系统取一个独一无二的名称吧!

-o ： 后面可以接一些挂载时额外加上的参数！比方说账号、密码、读写权限等：

ro, rw: 挂载文件系统成为只读(ro) 或可擦写(rw)

async, sync: 此文件系统是否使用同步写入(sync) 或异步 (async) 的内存机制，请参考文件系统运行方式。默认为 async。

auto, noauto: 允许此 partition 被以 mount -a 自动挂载(auto)

dev, nodev: 是否允许此 partition 上，可创建装置文件？ dev 为可允许

suid, nosuid: 是否允许此 partition 含有 suid/sgid 的文件格式?

exec, noexec: 是否允许此 partition 上拥有可运行 binary 文件?

user, nouser: 是否允许此 partition 让任何使用者运行 mount ？一般来说 mount 仅有 root 可以进行，但下达 user 参数，则可让一般 user 也能够对此 partition 进行 mount 。

defaults: 默认值为：rw,suid, dev, exec, auto, nouser, and async

remount: 重新挂载，这在系统出错，或重新升级参数时，很有用

举例1：挂载 EXT2/EXT3文件系统

```
[root@localhost ~]# mkdir /mnt/sda7
[root@localhost ~]# mount /dev/sda7/mnt/sda7
[root@localhost ~]# df
文件系统            1K-块       已用     可用 已用% 挂载点
/dev/sda2            9920624   4329132  5079424  47% /
/dev/sda3            4956316    141272   4559212  4% /home
```

```
/dev/sda1             101086    11726    84141  13% /boot
tmpfs                1037452        0  1037452   0% /dev/shm
/dev/sda6             1976312    42072  1833836   3% /mnt/sda6
.host:/              80148252  59099424 21048828  74% /mnt/hgfs
/dev/sda7              194450     9016   175396   5% /mnt/sda7
```

举例2：挂载 cd/dvd 光盘

```
[root@localhost ~]# mount -t iso9660/dev/cdrom /media/cdrom/
mount: block device /dev/cdrom iswrite-protected, mounting read-only
[root@localhost ~]# df
文件系统              1K-块       已用     可用 已用% 挂载点
/dev/sda2             9920624   4329132  5079424  47% /
/dev/sda3             4956316    141272  4559212   4% /home
/dev/sda1              101086     11726    84141  13% /boot
tmpfs                1037452        0  1037452   0% /dev/shm
/dev/sda6             1976312    42072  1833836   3% /mnt/sda6
.host:/              80148252  59231380 20916872  74% /mnt/hgfs
/dev/sda7              194450     9016   175396   5% /mnt/sda7
/dev/hdc              1651852   1651852        0 100% /media/cdrom
```

举例3：挂载 U 盘

```
[root@localhost ~]# mkdir /media/flash
[root@localhost ~]# mount -t vfat -o iocharset=cp950 /dev/sdb1 /media/flash
// iocharset为指定中文字符
[root@localhost ~]# df
文件系统              1K-块       已用     可用 已用% 挂载点
/dev/sda2             9920624   4329164  5079392  47% /
/dev/sda3             4956316    141272  4559212   4% /home
/dev/sda1              101086     11726    84141  13% /boot
tmpfs                1037452        0  1037452   0% /dev/shm
/dev/sda6             1976312    42072  1833836   3% /mnt/sda6
.host:/              80148252  59231444 20916808  74% /mnt/hgfs
/dev/sda7              194450     9016   175396   5% /mnt/sda7
/dev/hdc              1651852   1651852        0 100% /media/cdrom
/dev/sdb1             3977678   1385740  2591938  35% /media/flash
```

举例4：挂载信息会写入/etc/mtab 文件中

```
[root@localhost ~]# cat /etc/mtab
/dev/sda2 / ext3 rw 0 0
proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0
devpts /dev/pts devpts rw,gid=5,mode=620 00
/dev/sda3 /home ext3 rw 0 0
```

```
/dev/sda1 /boot ext3 rw 0 0
tmpfs /dev/shm tmpfs rw 0 0
/dev/sda6 /mnt/sda6 ext3 rw 0 0
none /proc/sys/fs/binfmt_misc binfmt_miscrw 0 0
.host:/ /mnt/hgfs vmhgfs rw,ttl=1 0 0
none /proc/fs/vmblock/mountPoint vmblock rw0 0
sunrpc /var/lib/nfs/rpc_pipefs rpc_pipefsrw 0 0
/dev/sda7 /mnt/sda7 ext3 rw 0 0
/dev/hdc /media/cdrom iso9660 ro 0 0
/dev/sdb1 /media/flash vfatrw,iocharset=cp950 0 0
```

举例5：系统默认挂载信息会记录在/etc/fstab 中

```
[root@localhost~]# cat /etc/fstab
LABEL=/              /                    ext3    defaults       1 1
LABEL=/home          /home                ext3    defaults       1 2
LABEL=/boot          /boot                ext3    defaults       1 2
tmpfs                /dev/shm             tmpfs   defaults       0 0
devpts               /dev/pts             devpts  gid=5,mode=620  0 0
sysfs                /sys                 sysfs   defaults       0 0
proc                 /proc                proc    defaults       0 0
LABEL=SWAP-sda5      swap                 swap    defaults       0 0
/dev/sda6            /mnt/sda6            ext3    defaults   1 2
```

磁盘卸载

语法：umount[-fn] 设备文件名或者挂载点

选项和参数：

-f:强制卸载

-n:不更新/etc/mtab 文件

举例：

```
[root@localhost ~]# df
文件系统             1K-块       已用     可用 已用% 挂载点
/dev/sda2            9920624   4329164  5079392  47% /
/dev/sda3            4956316    141272  4559212   4% /home
/dev/sda1             101086     11726    84141  13% /boot
tmpfs                1037452         0  1037452   0% /dev/shm
/dev/sda6            1976312     42072  1833836   3% /mnt/sda6
.host:/             80148252  59231444 20916808  74% /mnt/hgfs
/dev/sda7             194450      9016   175396   5% /mnt/sda7
/dev/hdc             1651852   1651852        0 100% /media/cdrom
/dev/sdb1            3977678   1385740  2591938  35% /media/PENDRIVE
/dev/sdb1            3977678   1385740  2591938  35% /media/flash
[root@localhost ~]# umount /media/flash
[root@localhost ~]# umount /media/cdrom
[root@localhost ~]# umount /dev/sda7
[root@localhost ~]# df
文件系统             1K-块       已用     可用 已用% 挂载点
/dev/sda2            9920624   4329164  5079392  47% /
/dev/sda3            4956316    141272  4559212   4% /home
/dev/sda1             101086     11726    84141  13% /boot
tmpfs                1037452         0  1037452   0% /dev/shm
/dev/sda6            1976312     42072  1833836   3% /mnt/sda6
.host:/             80148252  59231444 20916808  74% /mnt/hgfs
/dev/sdb1            3977678   1385740  2591938  35% /media/PENDRIVE
```

磁盘参数修改

文件系统卷标 (Label) 修改

磁盘的挂载可以通过文件系统的卷标(Label)来进行,但是要保证这个值的唯一性 我们可以通过 mke2fs 进行磁盘格式化来指定这个值，也可以通过 elabel 或 tune2fs 来修改这个值 e2label

语法：e2label 设备名称 新的 Label 名称

举例：修改 sda7Label 名称

```
[root@localhost ~]# e2label /dev/sda7"tkflabel"
[root@localhost ~]# df /dev/sda7
文件系统             1K-块       已用     可用 已用% 挂载点
-                 1037452      156  1037296   1% /dev
[root@localhost ~]# dumpe2fs  /dev/sda7
dumpe2fs 1.39 (29-May-2006)
Filesystemvolume name:   tkflabel
```

举例2：使用新 Label 进行挂载

```
[root@localhost ~]# mount -L"tkflabel" /mnt/sda7
[root@localhost ~]# df
文件系统             1K-块       已用     可用 已用% 挂载点
/dev/sda2            9920624   4329164  5079392  47% /
/dev/sda3            4956316    141272  4559212   4% /home
/dev/sda1             101086     11726    84141  13% /boot
tmpfs                1037452         0  1037452   0% /dev/shm
/dev/sda6            1976312     42072  1833836   3% /mnt/sda6
.host:/             80148252  59231444 20916808  74% /mnt/hgfs
/dev/sdb1            3977678   1385740  2591938  35% /media/PENDRIVE
/dev/sda7             194450      9016   175396   5% /mnt/sda7
```

tune2fs

语法：tune2fs[-jlL] 设备名称

选项与语法：

-l:类似 dump2fs - h 将 superblock 信息读取出来

-j:将 EXT2文件系统转换为 ext3

-L:类似 e2labe 功能

举例：

```
[root@localhost ~]# tune2fs -L"newlabel" /dev/sda7
tune2fs 1.39 (29-May-2006)
[root@localhost ~]# tune2fs -l /dev/sda7
tune2fs 1.39 (29-May-2006)
Filesystemvolume name:   newlabel
```

开机挂载

前面说到过开机挂载主要是从/etc/fstab 文件中读取挂载信息进行挂载，话句话说主要进行更改这个文件，添加新的挂载信息就可以进行自动开机加载

```
[root@www ~]# cat /etc/fstab
 # Device        Mount point   filesystem parameters    dump fsck

LABEL=/1         /          ext3      defaults       1 1
LABEL=/home      /home      ext3      defaults       1 2
LABEL=/boot      /boot      ext3      defaults       1 2
tmpfs            /dev/shm    tmpfs     defaults       0 0
devpts           /dev/pts    devpts    gid=5,mode=620  0 0
sysfs            /sys        sysfs     defaults       0 0
proc             /proc       proc      defaults       0 0
LABEL=SWAP-hdc5   swap       swap      defaults       0 0
```

Device：设备卷标(Label)

Mountpoint： 挂载点

Filesystem:文件系统类型

Parameters:文件系统参数（ -o 后面的参数 ）

Dump:是否被 dump 备份

Fsck:是否以 FSCK 检验扇区

启动的过程中，系统默认会以 fsck 检验我们的 filesystem 是否完整 (clean)。 不过，某些 filesystem 是不需要检验的，例如内存置换空间 (swap)，或者是特殊文件系统例如 /proc 与 /sys 等等。所以，在这个字段中，我们可以配置是否要以 fsck 检验该 filesystem。 0 是不要检验， 1 表示最早检验(一般只有根目录会配置为 1)， 2 也是要检验，不过 1 会比较早被检验啦！ 一般来说，根目录配置为 1 ，其他的要检验的 filesystem 都配置为 2 就好了。

特殊设备 loop 挂载

假如我们分区不够合理，没有足够的空间在创建一个分区，那么我们可以在已有分区上创建一个大文件，并将这个大文件作为单独的文件系统进行挂载。这就用到了特殊文件挂载

作法：

1.创建大文件

2.格式化

3.挂载

举例1：创建大文件

```
[root@bogon ~]# df -h
文件系统            容量  已用 可用 已用% 挂载点
/dev/sda2             9.5G  4.1G  5.0G  45% /
/dev/sda3             4.8G  138M  4.4G   4% /home
/dev/sda1              99M   12M   83M  13% /boot
tmpfs                1014M     0 1014M   0% /dev/shm
.host:/               49G  6.5G   43G  14% /mnt/hgfs
[root@bogon ~]# dd if=/dev/zero of=/home/newdev bs=1M count=512
512+0 records in
512+0 records out
536870912 bytes (537 MB) copied, 6.97647 seconds, 77.0 MB/s
[root@bogon ~]# df -h
文件系统            容量  已用 可用 已用% 挂载点
/dev/sda2             9.5G  4.1G  5.0G  45% /
/dev/sda3             4.8G  651M  3.9G  15% /home
/dev/sda1              99M   12M   83M  13% /boot
tmpfs                1014M     0 1014M   0% /dev/shm
.host:/               49G  6.5G   43G  14% /mnt/hgfs
[root@bogon ~]# ll /home/newdev
-rw-r--r-- 1 root root 536870912 02-27 20:14 /home/newdev
```

以上发现 home 文件系统使用量增大了512 M

举例2：格式化

```
[root@bogon ~]# mkfs -t ext3 /home/newdev
mke2fs 1.39 (29-May-2006)
/home/newdev is not a block special device.
Proceed anyway? (y,n) y
Filesystem label=
```

```
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
......
```

举例3：挂载

```
[root@bogon ~]# mount -o loop /home/newdev /media/cdrom
[root@bogon ~]# df -h
文件系统            容量  已用 可用 已用% 挂载点
/dev/sda2             9.5G  4.1G  5.0G  45% /
/dev/sda3             4.8G  651M  3.9G  15% /home
/dev/sda1              99M   12M   83M  13% /boot
tmpfs                1014M     0 1014M   0% /dev/shm
.host:/               49G  6.6G   43G  14% /mnt/hgfs
/home/newdev         496M   19M  452M   4% /media/cdrom
```

25

磁盘分区，格式化与检验

磁盘分区：fdisk

磁盘格式化：mkfs,mke2fs

磁盘检测：fsck

大容量磁盘分区：parted

磁盘分区

语法：fdisk[-l] 设备名称

-l:输出系统内所有分区

举例：

```
[root@localhost ~]# fdisk -l

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280bytes

  Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391   83  Linux
/dev/sda2              14        1288    10241437+  83  Linux
/dev/sda3            1289        1925     5116702+  83  Linux
/dev/sda4            1926        2610     5502262+   5  Extended
/dev/sda5            1926        2052     1020096   82 Linux swap / Solaris
/dev/sda6            2053        2302     2008093+  83  Linux
```

1. 查看磁盘文件名

[root@localhost ~]# df /

文件系统 1K-块 已用 可用 已用% 挂载点

/dev/sda2 9920624 4329108 5079448 47% /

2. 查看磁盘分区功能

```
[root@localhost ~]# fdisk /dev/sda  //这里不带数字

The number of cylinders for this disk isset to 2610.
There is nothing wrong with that, but thisis larger than 1024,
and could in certain setups cause problemswith:
1) software that runs at boot time (e.g.,old versions of LILO)
2) booting and partitioning software fromother OSs
  (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): m
Command action
  a   toggle a bootable flag
  b   edit bsd disklabel
  c   toggle the dos compatibilityflag
```

```
 d   delete a partition //删除磁盘分区
 l   list known partition types
 m   print this menu  //查看磁盘分区功能
 n   add a new partition //增加一个磁盘分区
 o   create a new empty DOSpartition table
 p   print the partition table //查看磁盘分区
 q   quit without saving changes
 s   create a new empty Sundisklabel
 t   change a partition's system id
 u   change display/entry units
 v   verify the partition table
 w   write table to disk and exit
 x   extra functionality (expertsonly)
```

删除磁盘分区

```
[root@localhost ~]# fdisk /dev/sda

The number of cylinders for this disk isset to 2610.
There is nothing wrong with that, but thisis larger than 1024,
and could in certain setups cause problemswith:
1) software that runs at boot time (e.g.,old versions of LILO)
2) booting and partitioning software fromother OSs
  (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): p

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280bytes

  Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391  83  Linux
/dev/sda2              14        1288    10241437+  83  Linux
/dev/sda3            1289        1925     5116702+  83  Linux
/dev/sda4            1926        2610     5502262+   5  Extended
/dev/sda5            1926        2052     1020096   82  Linux swap / Solaris
/dev/sda6            2053        2302     2008093+  83  Linux
```

由上可知我的磁盘主要分为6个分区，1,2,3为主分区，4为扩展分区，5为 swap 分区，6是逻辑分区

```
Command (m for help): d
Partition number (1-6): 3
```

```
Command (m for help): p

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280bytes

  Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391   83  Linux
/dev/sda2              14        1288    10241437+  83  Linux
/dev/sda4            1926        2610     5502262+   5  Extended
/dev/sda5            1926        2052     1020096   82  Linux swap / Solaris
/dev/sda6            2053        2302     2008093+  83  Linux

删除主分区 sad3 后可以看到磁盘信息不在包含 sad3
Command (m for help): d
Partition number (1-6): 4

Command (m for help): p

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280bytes

  Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391   83  Linux
/dev/sda2              14        1288    10241437+  83  Linux
```

删除扩展分区 sad4 后可以看到扩展分区，逻辑分区都被删除（因为逻辑分区是由扩展分区衍生而来的）。

增加磁盘分区

磁盘分区最多只能有4个主分区+扩展分区组成，其中扩展分区最多只能有一个，剩下在创建的分区都是由扩展分区衍生出来的逻辑分区

举例1. 由于磁盘现分区分为3个主分区，1个扩展分区。因此在创建时将直接创建逻辑分区，而不在询问是否创建主分区或者扩展分区

```
[root@localhost ~]# fdisk /dev/sda

The number of cylinders for this disk isset to 2610.
There is nothing wrong with that, but thisis larger than 1024,
and could in certain setups cause problemswith:
1) software that runs at boot time (e.g.,old versions of LILO)
2) booting and partitioning software fromother OSs
```

```
  (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): p

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280bytes

  Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391  83  Linux
/dev/sda2              14        1288    10241437+  83  Linux
/dev/sda3            1289        1925     5116702+  83  Linux
/dev/sda4            1926        2610     5502262+   5  Extended
/dev/sda5            1926        2052     1020096   82 Linux swap / Solaris
/dev/sda6            2053        2302     2008093+  83  Linux

Command (m for help): n
First cylinder (2303-2610, default 2303):
```

举例2：创建主/扩展分区

```
[root@localhost ~]# fdisk /dev/sda

The number of cylinders for this disk isset to 2610.
There is nothing wrong with that, but thisis larger than 1024,
and could in certain setups cause problemswith:
1) software that runs at boot time (e.g.,old versions of LILO)
2) booting and partitioning software fromother OSs
  (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): d //先将主分区和逻辑分区删除（如果为4个则默认创建逻辑分区）
Partition number (1-6): 2

Command (m for help): d
Partition number (1-6): 4

Command (m for help): p

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280bytes

  Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391  83  Linux
/dev/sda3            1289        1925     5116702+  83  Linux
```

```
Command (m for help): n
Command action
   e  extended
   p  primary partition (1-4)
```

提示用户选择是是创建主分区还是扩展分区

举例3.创建逻辑分区与扩展分区

```
root@localhost ~]# fdisk /dev/sda

The number of cylinders for this disk isset to 2610.
There is nothing wrong with that, but thisis larger than 1024,
and could in certain setups cause problemswith:
1) software that runs at boot time (e.g.,old versions of LILO)
2) booting and partitioning software fromother OSs
  (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): p

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280bytes

  Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391   83  Linux
/dev/sda2              14        1288    10241437+  83  Linux
/dev/sda3            1289        1925     5116702+  83  Linux
/dev/sda4            1926        2610     5502262+   5  Extended
/dev/sda5            1926        2052     1020096   82  Linux swap / Solaris
/dev/sda6            2053        2302     2008093+  83  Linux

Command (m for help): d
Partition number (1-6): 4

Command (m for help): n
Command action
  e   extended
  p   primary partition (1-4)
e
Selected partition 4
First cylinder (1926-2610, default 1926):
Using default value 1926
Last cylinder or +size or +sizeM or +sizeK(1926-2610, default 2610):
Using default value 2610
```

```
Command (m for help): p

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280bytes

  Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391   83  Linux
/dev/sda2              14        1288    10241437+  83  Linux
/dev/sda3            1289        1925     5116702+  83  Linux
/dev/sda4            1926        2610     5502262+   5  Extended
```

sd4为新创建的扩展分区，大小为从柱面1926到2610

```
Command (m for help): n
Firstcylinder (1926-2610, default 1926):
Usingdefault value 1926
Lastcylinder or +size or +sizeM or +sizeK (1926-2610, default 2610): +500M
```

对于此处可以指定柱面号码，以可以通过+XXM 指定大小，让其自动分配柱面

Command (m for help): p

```
Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280bytes

  Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391   83  Linux
/dev/sda2              14        1288    10241437+  83  Linux
/dev/sda3            1289        1925     5116702+  83  Linux
/dev/sda4            1926        2610     5502262+   5  Extended
/dev/sda5            1926        1987      497983+ 83  Linux
```

sd5为新创建的逻辑分区，大小为500M

内核查找分区

当我们增加分区后，系统让我们 reboot 以加载分区。也可以不用重启，只需要通知内容重新查找分区即可

```
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: Re-reading the partition tablefailed with error 16: 设备或资源忙.
```

```
The kernel still uses the old table.
The new table will be used at the nextreboot.
Syncing disks.
[root@localhost~]# partprobe
```

磁盘格式化

分区完毕后要进行文件系统的格式化

mkfs

语法：mkfs[-t 文件系统格式] 设备文件名

选项与参数：

-t:文件系统格式，例如 ext3,ext2,vfat 等

举例

```
[root@localhost ~]# mkfs -t ext3 /dev/sda7
mke2fs 1.39 (29-May-2006)
Filesystemlabel=
OS type: Linux
Blocksize=1024 (log=0)
Fragment size=1024 (log=0)
50200 inodes, 200780 blocks
10039 blocks (5.00%) reserved for the superuser
First data block=1
Maximum filesystem blocks=67371008
25 block groups
8192 blocks per group, 8192 fragments pergroup
2008 inodes per group
Superblock backups stored on blocks:
        8193, 24577, 40961, 57345, 73729

Writing inode tables: done
Creating journal (4096 blocks): done
Writing superblocks and filesystemaccounting information: done

This filesystem will be automaticallychecked every 37 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

其中文件系统 Label 以及 iBLOCK 大小均采用默认大小。如果对于 EXT2/EXT3 我们对这些信息由特殊的需求,可以使用 mke2fs

mke2fs

语法：mke2fs[-b block大小] [-i inode 大小] [-L 卷标] [-cj] 设备

选项与参数：

-b:设置 block 大小，目前支持1024,2048,4096

-i:多少容量给予一个 inode

-c:检查磁盘错误

-L:卷标名称（Label）

-j:自动加入日志系统成为 EXT3文件系统，不加在默认为 EXT2

举例

```
[root@localhost ~]# mke2fs -b 2048 -i 4096-L "TKFDISK" -j /dev/sda7
mke2fs 1.39 (29-May-2006)
Filesystemlabel=TKFDISK
OS type: Linux
Blocksize=2048 (log=1)
Fragment size=2048 (log=1)
50288 inodes, 100390 blocks
5019 blocks (5.00%) reserved for the superuser
First data block=0
Maximum filesystem blocks=103809024
7 block groups
16384 blocks per group, 16384 fragments pergroup
7184 inodes per group
Superblock backups stored on blocks:
        16384, 49152, 81920

Writing inode tables: done
Creatingjournal (4096 blocks): done
Writing superblocks and filesystemaccounting information: done

This filesystem will be automaticallychecked every 31 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

磁盘检测(fsck)

语法: **fsck [-t 文件系统格式] [-ACay]

**选项与参数

-t： 文件系统格式。

-A： 依据/etc/fstab 的内容，将需要的装置扫瞄一次。

-a： 自动修复检查到的有问题的扇区.

-y： 与 -a 类似，但是某些 filesystem 仅支持 -y 这个参数

-C： 可以在检验的过程当中，使用一个直方图来显示目前的进度!

EXT2/EXT3 的额外选项功能：(e2fsck 这支命令所提供)

-f： 强制检查！一般来说，如果 fsck 没有发现任何 unclean 的旗标，不会主动进入细部检查的，如果您想要强制 fsck 进入细部检查，就得加上 -f

-D： 针对文件系统下的目录进行优化配置。

举例

```
[root@localhost ~]# fsck -Cf /dev/sda7
fsck 1.39 (29-May-2006)
e2fsck 1.39 (29-May-2006)
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
TKFDISK: 11/50288 files (9.1%non-contiguous), 7673/100390 blocks
```

说明：需要磁盘检查的分区不能挂载在系统上，需要先被卸载才能磁盘检测

大容量磁盘分区(parted)

由于 fdisk 无法支持到高于2 TB 以上的分区，此时就需要 parted 来处理了

语法：parted [设备] [命令 [参数]]

选项与参数：

新增分区：mkpart [primary|logical|extended] [ext3|vfat]开始结束

分区表：print

删除分区：rm [partition]

举例1：查看分区表

```
[root@bogon ~]# parted /dev/sda print

Model: VMware, VMware Virtual S (scsi)
Disk /dev/sda: 21.5GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos

Number  Start   End     Size    Type      File system  标志
 1      32.3kB  107MB   107MB   主分区    ext3         启动
 2      107MB   10.6GB  10.5GB  主分区    ext3
 3      10.6GB  15.8GB  5240MB  主分区    ext3
 4      15.8GB  21.5GB  5634MB  扩展分区
 5      15.8GB  16.9GB  1045MB  逻辑分区  linux-swap

信息: 如果必要，不要忘记更新 /etc/fstab。
```

通过以上信息可以看出，扩展分区到21.5 G，逻辑分区使用到16.9 G，那么16.9 G~21.5 G只部分空间还为被使用（未被分区）

举例2：新增分区

```
[root@bogon ~]# parted /dev/sda mkpart logical ext3 16.9G 18.9G
信息: 如果必要，不要忘记更新 /etc/fstab。

[root@bogon ~]# parted /dev/sda print

Model: VMware, VMware Virtual S (scsi)
Disk /dev/sda: 21.5GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
```

```
Number  Start   End     Size    Type     File system  标志
 1      32.3kB  107MB   107MB   主分区   ext3         启动
 2      107MB   10.6GB  10.5GB  主分区   ext3
 3      10.6GB  15.8GB  5240MB  主分区   ext3
 4      15.8GB  21.5GB  5634MB  扩展分区
 5      15.8GB  16.9GB  1045MB  逻辑分区  linux-swap
 6      16.9GB  18.9GB  2023MB  逻辑分区
```

举例3：删除分区

```
[root@bogon ~]# parted /dev/sda rm 6
信息: 如果必要，不要忘记更新 /etc/fstab。

[root@bogon ~]# parted /dev/sda print

Model: VMware, VMware Virtual S (scsi)
Disk /dev/sda: 21.5GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos

Number  Start   End     Size    Type     File system  标志
 1      32.3kB  107MB   107MB   主分区   ext3         启动
 2      107MB   10.6GB  10.5GB  主分区   ext3
 3      10.6GB  15.8GB  5240MB  主分区   ext3
 4      15.8GB  21.5GB  5634MB  扩展分区
 5      15.8GB  16.9GB  1045MB  逻辑分区  linux-swap

信息: 如果必要，不要忘记更新 /etc/fstab。
```

说明：parted 分区提交即执行，因此使用起来需小心

26

文件系统简单操作

磁盘的容量查看 df

目录的容量查看 du

连接文件 ln

磁盘的容量查看(df)

语法：df[-ahikhtm] 目录或文件名 **选项与参数：** -a:列出所有的文件系统，包括系统特有的 proc 等文件系统 -k:以 KB 为单位显示 -m:以 MB 为单位显示 -h:以 GB,MB,KB 等格式显示 -H:以 M=1000 K 代替 M=1024 K 显示 -T:连同该分区的文件系统名称一起列出 -i:以 inode 的数量来显示

举例：

```
[root@localhost ~]# df -hT
文件系统      类型    容量  已用 可用 已用% 挂载点
/dev/sda2     ext3    9.5G  4.2G  4.9G  47% /
/dev/sda3     ext3    4.8G  138M  4.4G   4% /home
/dev/sda1     ext3     99M   12M   83M  13% /boot
tmpfs         tmpfs   1014M     0 1014M   0% /dev/shm
/dev/sda6     ext3    1.9G   42M  1.8G   3% /mnt/sda6
.host:/     vmhgfs     77G   57G   21G  74% /mnt/hgfs
[root@localhost ~]# df -ihT
文件系统      类型     Inode (I)已用 (I)可用 (I)已用% 挂载点
/dev/sda2     ext3      2.5M    168K    2.3M    7% /
/dev/sda3     ext3      1.3M      22    1.3M    1% /home
/dev/sda1     ext3       26K      35     26K    1% /boot
tmpfs         tmpfs     219K       1    219K    1% /dev/shm
/dev/sda6     ext3      247K      11    247K    1% /mnt/sda6
.host:/     vmhgfs         0       0       0    -  /mnt/hgfs
```

目录的容量查看(du)

语法：du[-ahskm] 目录或文件名

选项与参数：

-a:列出所有文件与目录容量

-h:以 G/M 容量格式显示

-s:列出总量，不在列出目录下面文件量

-S:不包括子目录下的统计()

-k:以 KB 为单位显示

-m:以 MB 为单位显示

举例：

[root@localhost ~]# du 8 /bin 6 /boot 1 /dev … 216 /tmp 4077 /usr 99 /var [root@localhost ~]#

连接文件 ln

语法： ln [-sf]源文件 目标文件

选项与参数：

-s:如果不加任何参数默认是 hardlink ,加上-s 是 symboliclink

-f:如果目标文件存在，就主动将目标文件删除后创建

Hard link(硬连接)

Hard link 只是在某个目录下新建一个文件名连接到某个 inode 上

说明： 1. 创建文件 F1，文件系统为其分配一个 INODE(F1I)和若干 IBLOCK, 此时连接到 INODE(F1I)只有 F1 因此 INODE(F1I)连接数为1

```
[root@localhost ~]# touch f1
[root@localhost ~]# ll -i f1
846433 -rw-r--r-- 1 root root 0 02-24 09:33 f1
```

1. 创建 F1的 Hard Link FH1, Hard link 并不会分配新的 INODE 和 IBLOCK，只是将文件名连接都 F1的 INode 上

```
[root@localhost ~]# ln f1 fh1
[root@localhost ~]# ll -i f1 fh1
846433 -rw-r--r-- 2 root root 0 02-24 09:33 f1
846433 -rw-r--r-- 2 root root 0 02-24 09:33 fh1
```

可以看到 inode 有1变成了2，INODE 所指向的文件现在是 f1,fh1,指向的数据还是以前的那份iblock

硬连接的好处：

1.不会创建新的 INODE 和 iblock

2.硬连接文件或源文件删除不会影响其他(删除只是接触 inode 与文件的连接关系，猜想只要连接数不为0，就不会删除)

Symbolic link

symbolic link 创建的文件时一个独立的新文件会占用一个新的 INODE 和若干 iblock

说明：

1. 创建文件 F2，文件系统为其分配一个 INODE(F2I)和若干 IBLOCK, 此时连接到 INODE(F2I)只有 F1因此 INODE(F2I)连接数为1

```
[root@localhost ~]# touch f2
[root@localhost ~]# ll -i f2
846434 -rw-r--r-- 1 root root 0 02-24 09:49 f2
```

2. 创建 F2的符号文件 F2S, 文件系统会分配一个新的 INODE(F2SI)和若干 IBLOCK 给 F2S

```
[root@localhost ~]# ln -s f2 f2s
[root@localhost ~]# ll -i f2 f2s
846434 -rw-r--r-- 1 root root 0 02-24 09:49 f2
846435 lrwxrwxrwx 1 root root 2 02-24 09:51 f2s -> f2
```

可以看到 f2,f2S 的 INODE 不是同一个，并且连接数都是1.说明他们是不同的独立文件，但是 f2S 对 f2进行符号链接的呢？原因就是 f2s 的 iblock 其大小为2，记录就是 f2的文件名。因此可以这样理解，\

1. f2s 对应的 INODE(F2SI)记录了 iblock 编号
2. iblock 里记录了 F2的文件名
3. 通过 F2的文件名就可以找到 F2对应的 INODE 和 iblock

所以加入我们删除了 F2 那么 F2S 就无法再开启，印在 F2S 需要去讯在 F2这个文件，此时已经被删除了

```
[root@localhost ~]# rm -f f2
[root@localhost ~]# cat f2s
cat: f2s: 没有那个文件或目录
```

目录的连接数量

当我们创建一个目录是默认会在这个目录下创建两个隐藏文件“.与..” 其中.指的是本层目录..指的是上层目录

```
[root@localhost ~]# ll -id /tmp
745569 drwxrwxrwt 26 root root 409
[root@localhost ~]# cd /tmp
[root@localhost tmp]# mkdir newdir
[root@localhost tmp]# ll -id /tmp /tmp/newdir
 745569 drwxrwxrwt 27 root root 4096 02-24 10:05 /tmp
1008319 drwxr-xr-x  2 root root 4096 02-24 10:05 /tmp/newdir
```

由上面可以看出

1. 在未创建 newdir 是，tmp 文件夹对应 INODE 的连接数26
2. 当创建 newdir 后，系统默认创建“.与..”文件，一个指向自己，一个指向上一层（ /tmp ）

3. “.与..”文件都是以硬链接的方式连接，因此可以看到此时，tmp 文件夹对应 INODE 连接数27，newdir 的连接数为2（一个是 newdir 连接，一个是“.”连接）

27

服务

常驻在内存中的进程，且提供一些系统功能，就是服务。这个进程称为 daemon.换另外一种说法：服务包括一个提供系统功能的程序以及一个执行该程序的进程

每个服务对应设备的一个端口

服务主要分类

按照服务的启动方式可以分为2类:

自启动的服务：大部分为开机就会启动的服务。每一个服务都有一个进程进行控制

统一控制启动服务:由一个独立进程负责启动这些服务，至于何时启动由用户进行控制。这个独立的进程就是 xinetd

统一控制启动服务也是一个自启动服务，只是其控制的服务不一定开机就启动

几个重要的目录

/etc/init.d/*:所有服务启动脚本存放处(学习 shell script 语法好去处)*

/etc/sysconfig/(各服务的初始化环境配置文件)

/etc/xined.conf**统一控制启动服务总体配置文件**

/etc/xined.d/* **统一控制启动服务配置文件（每个服务的配置文件）**

/etc/*:自启动服务各自的配置文件*

*/var/lib/*自启动服务各自的配置文件

/var/run/*:各个服务的程序的 PID 记录处

以自启动服务 syslogd 为例

[root@localhost~]# ll /etc/sysconfig/syslog /etc/init.d/syslog /etc/syslog.conf

-rwxr-xr-x1 root root 2043 2010-04-03 /etc/init.d/syslog =>记录程序文件

-rw-r--r--1 root root610 2010-04-03/etc/sysconfig/syslog =>记录初始化信息

-rw-r--r--1 root root938 02-14 10:10/etc/syslog.conf =>记录配置信息

自启动服务的操作

自启动服务在系统启动的时候可能会启动(需要配置)，当然我们也可以控制它的启动和停止以及以下其他操作。

直接执行服务脚本

前面说到所有服务的启动脚本都存放在/etc/init.d/* ，我们就以 syslog 服务为例

syslog 服务对应的 shellscript

```
case "$1" in
 start)
      start
      ;;
 stop)
      stop
      ;;
 status)
      rhstatus
      ;;
 restart)
      restart
      ;;
 reload)
      reload
      ;;
 condrestart)
      [ -f /var/lock/subsys/syslog ] && restart || :
      ;;
 *)
      echo $"Usage: $0 {start|stop|status|restart|condrestart}"
      exit 2
esac
```

通过以上可以粗略的看到这里包含6个方法(start,stop,rhstatus….) 调用这些方法的条件是 执行 shell script 后面跟的参数（ start|stop|status|restart|condrestart ）

通过以上分析，如果我们要知道一个服务有哪些操作，可以之间查看这个服务的脚本文件

```
[root@localhost init.d]# ./syslog status
syslogd (pid  3637) 正在运行...
klogd (pid  3640) 正在运行...
```

```
[root@localhost init.d]# ./syslog restart
关闭内核日志记录器:                                [确定]
关闭系统日志记录器:                                [确定]
启动系统日志记录器:                                [确定]
启动内核日志记录器:                                [确定]
```

通过 service 指令执行服务

语法：service[服务名称] 执行操作

service --status-all

选项与参数：

执行操作:服务需要进行的工作（start|stop|status|restart….）

--status-all:将系统所有自启动服务显示

举例：

```
[root@localhost ~]# service syslog restart
关闭内核日志记录器:                                [确定]
关闭系统日志记录器:                                [确定]
启动系统日志记录器:                                [确定]
启动内核日志记录器:                                [确定]
[root@localhost ~]# service --status-all
acpid (pid 3901) 正在运行...
anacron 已停
atd (pid 4240) 正在运行...
auditd (pid  3609) 正在运行...
…….
```

统一控***务的操作

前面提到统一控***务是由一个特殊的进程(xinetd)来控制其他服务的行为

整体配置文件

如果针对个体服务配置文件未配置下面项目，那么服务的设置值将去下面内容作为默认值

```
[root@localhost etc]# vim /etc/xinetd.conf
defaults
{
 # 服务启动成功或失败，以及相关登陆行为的记录文件

       log_type       = SYSLOG daemon info
       log_on_failure  = HOST
       log_on_success  = PID HOST DURATION EXIT
 # 允许或限制联机的默认值

       cps        = 50 10
       instances   = 50
       per_source  = 10
 # 网络 (network) 相关的默认值

       v6only
 # 环境参数的配置

       groups
       umask
}
```

服务配置文件分析

举例：rsync 是统一控*务中的一个，下面是这个服务的以下配置**

```
[root@localhost etc]# vim /etc/xinetd.d/rsync
 # default: off

 # description: The rsync server is a good addition to an ftp server, as it \

 #       allows crc checksumming etc.
```

```
service rsync
{
        disable = yes
        socket_type     = stream
        wait            = no
        user            = root
        server          = /usr/bin/rsync
        server_args     = --daemon
        log_on_failure  += USERID
}
```

配置文件标识说明

=： 表示后面的配置参数就是这样

+=： 表示后面的配置为在原来的配置里头加入新的参数

-=： 表示后面的配置为在原来的参数舍弃这里输入的参数

以下图表来自鸟哥私房菜

attribute(功能)	说明与范例
一般配置项目：服务的识别、启动与程序	
disable(启动与否)	配置值：[yes|no]，默认 disable = yes ，此值可配置该服务是否要启动若要启动就得要配置为[disable = no]
id(服务识别)	配置值：[服务的名称] 虽然服务在配置文件开头[service 服务名称]已经指定了，不过有时后会有重复的配置值，此时可以用 id 来取代服务名称。

server (程序文件名)	配置值：[程序的绝对路径名] 这个就是指出这个服务的启动程序.例如 /usr/bin/rsync 为启动 rsync 服务的命令，所以这个配置值就会成为： [server = /usr/bin/rsync]
server_args (程序参数)	配置值：[程序相关的参数] 这里应该输入的就是你的 server 那里需要输入的一些参数.例如 rsync 需要加入 --daemon ， 所以这里就配置：[server_args = --daemon]。与上面 server 搭配，最终启动服务的方式[/usr/bin/rsync --daemon]
user (服务所属 UID)	配置值：[使用者账号] 如果 xinetd 是以 root 的身份启动来管理的，那么这个项目可以配置为其他用户。此时这个 daemon 将会以此配置值指定的身份来启动该服务的程序。举例来说，你启动 rsync 时会以这个配置值作为该程序的 UID。
group	跟 user 的意思相同.此项目填入组名即可。
一般配置项目：联机方式与联机封包协议	
socket_type (封包类型)	配置值：[stream|dgram|raw]，与封包有关 stream 为联机机制较为可靠的 TCP 封包，若为 UDP 封包则使用 dgram 机制。raw 代表 server 需要与 IP 直接对谈.举例来说 rsync 使用 TCP ，故配置为[socket_type = stream]
protocol (封包类型)	配置值：[tcp|udp]，通常使用 socket_type 取代此配置 使用的网络协议，需参考 /etc/protocols 内的通讯协议，一般使用 tcp 或 udp。由于与 socket_type 重复， 因此这个项目可以不指定。

wait (联机机制)	配置值：[yes(single)|no(multi)]，默认 wait = no 这就是我们刚刚提到的 Multi-threaded 与 single-threaded .一般来说，我们希望大家的要求都可以同时被激活，所以可以配置[wait = no] 此外，一般 udp 配置为 yes 而 tcp 配置为 no。
instances (最大联机数)	配置值：[数字或 UNLIMITED] 这个服务可接受的最大联机数量。如果你只想要开放 30 个人联机 rsync 时，可在配置文件内加入：[instances = 30]
per_source (单一用户来源)	配置值：[一个数字或 UNLIMITED] 如果想要控制每个来源 IP 仅能有一个最大的同时联机数，就指定这个项目吧.例如同一个 IP 最多只能连 10 条联机[per_source = 10]
cps (新联机限制)	配置值：[两个数字] 为了避免短时间内大量的联机要求导致系统出现忙碌的状态而有这个 cps 的配置值。第一个数字为一秒内能够接受的最多新联机要求， 第二个数字则为，若超过第一个数字那暂时关闭该服务的秒数。

一般配置项目：登录文件的记录

log_type (登录档类型)	配置值：[登录项目 等级] 当数据记录时，以什么登录项目记载？且需要记载的等级为何(默认为 info 等级)。
log_o	配置值：[PID,HOST,USERID,EXIT,DURATION]

n_success log_on_failure (登录状态)	在[成功登陆]或[失败登陆]之后，需要记录的项目：PID 为纪录该 server 启动时候的 process ID ， HOST 为远程主机的 IP、USERID 为登陆者的账号、EXIT 为离开的时候记录的项目、DURATION 为该用户使用此服务多久?

进阶配置项目：环境、网络端口口与联机机制等

env (额外变量配置)	配置值：[变量名称=变量内容] 这一个项目可以让你配置环境变量
port (非正规埠号)	配置值：[一组数字(小于 65534)] 这里可以配置不同的服务与对应的 port ，但是请记住你的 port 与服务名称必须与 /etc/services 内记载的相同才行.不过，若服务名称是你自定义的，那么这个 port 就可以随你指定
redirect (服务转址)	配置值：[IP port] 将 client 端对我们 server 的要求，转到另一部主机上去. 例如当有人要使用你的 ftp 时，你可以将他转到另一部机器上面去.那个 IP_Address 就代表另一部远程主机的 IP .
includedir (呼叫外部配置)	配置值：[目录名称] 表示将某个目录底下的所有文件都给他塞进来

安全控管项目：

bind (服务接口锁定)	配置值：[IP] 这个是配置[允许使用此一服务的适配卡]的意思.举个例子来说，你的 Linux 主机上面有两个 IP ，而你只想要让 IP1 可以使用此一服务，但 IP2 不能使用此服务，这里就可以将 IP1 写入即可.那么 IP2 就不可以使用此一 server
interface	配置值：[IP] 与 bind 相同
only_from (防火墙机制)	配置值：[0.0.0.0, 192.168.1.0/24, hostname, domainname] 这东西用在安全机制上面，也就是管制[只有这里面规定的 IP 或者是主机名可以登陆.]
no_access (防火墙机制)	配置值：[0.0.0.0, 192.168.1.0/24, hostname, domainname] 跟 only_from 差不多.就是用来管理可否进入你的 Linux 主机激活你的 server 服务的管理项目. no_access 表示[不可登陆]的 PC 啰.
access_times (时间控管)	配置值：[00:00-12:00, HH:MM-HH:MM] 这个项目在配置[该服务 server 启动的时间]，使用的是 24 小时的配置.例如你的 ftp 要在 8 点到 16 点开放的话，就是： 08:00-16:00。
umask	配置值：[000, 777, 022] 可以配置用户创建目录或者是文件时候的属性.系统建议值是 022 。

通过统一控***务启动一个服务

1. 将服务设置为启动

```
[root@localhost xinetd.d]# cat /etc/xinetd.d/rsync|sed 's/yes/no/g' >/etc/xinetd.d/tmp
[root@localhost xinetd.d]# vim tmp
[root@localhost xinetd.d]# cat ./tmp >./rsync
[root@localhost xinetd.d]# cat ./rsync # default: off

 # description: The rsync server is a good addition to an ftp server, as it \

 #       allows crc checksumming etc.

service rsync
{
        disable         = no
        socket_type     = stream
        wait            = no
        user            = root
        server          = /usr/bin/rsync
        server_args     = --daemon
        log_on_failure  += USERID
}
```

2. 重新启动统一控***务进程，以重启我们刚才更改的这个服务

```
[root@localhost etc]# service xinetd restart
停止 xinetd：                                  [确定]
启动 xinetd：                                  [确定]
```

3. 查看端口判断服务是否启用成功

```
[root@localhost xinetd.d]# cat /etc/services |grep 'rsync'
rsync           873/tcp                 # rsync
rsync           873/udp                  # rsync
root@localhost xinetd.d]# netstat -tnlp|grep 873
tcp        0      0 0.0.0.0:873               0.0.0.0:*                   LISTEN      10301/xinetd
```

说明：所有服务端口在/etc/services 可以查看到

设置服务开机启动

前面说到部分自启动服务会开机启动，通过 xinetd 进程控制的统一控***务也可以通过更改服务配置文件中disable=no,也可以控制其启动，

那么通过什么配置让我们可以选择哪些服务开机就启动，哪些服务开机时不启动

语法：chkconfig--list

chkconfig [--level [0123456]] 服务名称 [on|off]

参数与选项：

--list：查看所有服务开机启动情况

--level:启动级别（就是 init 后面那个数字，3为命令行模式，5为图形界面模式）

举例1：查看所有服务开机启动情况

```
[root@localhost xinetd.d]# chkconfig --list
NetworkManager  0:关闭  1:关闭  2:关闭  3:关闭  4:关闭  5:关闭  6:关闭
acpid           0:关闭  1:关闭  2:启用  3:启用  4:启用  5:启用  6:关闭
anacron         0:关闭  1:关闭  2:启用  3:启用  4:启用  5:启用  6:关闭

基于 xinetd 的服务：
        chargen-dgram:  关闭
        chargen-stream: 关闭
        daytime-dgram:  关闭
tftp:           启用
=>可以看出 xinetd 进程控制的服务通过更改 disable=no 也是可以设置开机启动的
```

举例2：设置服务开机启动

```
[root@localhost xinetd.d]# chkconfig --level 345 NetworkManager on
[root@localhost xinetd.d]# chkconfig --list
NetworkManager  0:关闭  1:关闭  2:关闭  3:启用  4:启用  5:启用  6:关闭
```

总结：1.对于自启动的服务来说，通过 chkconfig[--level [0123456]]可设置是否开机启动

2.对于统一控制的服务，可以通过更改服务配置文件 disable=no 来设置开启启动

如何制作自己的服务

语法：chkconfig[--add|--del] 服务名称

选项与参数：

--add:添加一个服务到服务管理器

--del:删除一个服务从服务管理器

步骤1：创建一个程序，提供某种功能

说明：此程序执行文件必须在/etc/init.d/目录下

```
#!/bin/bash

# chkconfig: 35 80 70

# description:hello


echo "这是我的 script. 参数是 $1"
=> chkconfig: 35 80 70其中，35指的是启动级别，80指的是启动顺序，70指的是结束顺序(因为服务启动与结束是有依赖关系的因此
=> description 添加服务描述信息

[root@localhost init.d]# ll myscript
-rwxrwxrwx 1 root root 97 03-20 16:08 myscript
```

步骤2：添加服务到服务管理器

```
[root@localhost init.d]# chkconfig --add myscript
[root@localhost init.d]# chkconfig --list myscript
myscript        0:关闭  1:关闭  2:关闭  3:启用  4:关闭  5:启用  6:关闭
```

开机显示效果

28

日志系统

日志系统对于一个系统来说是非常重要的，从日志文件我们可以获取到系统的运行状况，协助我们排查问题。

对于 CentOs 来说，日志系统主要包含2个服务与1个程序

syslogd:记录系统与网络服务的信息

klogd:记录内核产生的各项信息

logrotate:日志文件的轮替功能

说明：不同的 UNIX LIKE 对应的服务可能不一样

syslogd 服务

syslogd 服务配置文件分析

```
[root@localhost ~]# cat /etc/syslog.conf
 #kern.*                                 /dev/console

*.info;mail.none;news.none;authpriv.none;cron.none          /var/log/messages
authpriv.*                               /var/log/secure
mail.*                                  -/var/log/maillog
 # Log cron stuff

cron.*                                   /var/log/cron
*.emerg                                   *
uucp,news.crit                                /var/log/spooler
local7.*                                 /var/log/boot.log
news.=crit                              /var/log/news/news.crit
news.=err                               /var/log/news/news.err
news.notice                              /var/log/news/news.notice
```

配置文件格式如下

【服务类型】【信息等级设置】【信息存储方式】

以 cron.* /var/log/cron 为例

cron 是服务类型

. 是信息等级设置

/var/log/cron 信息存储方式

服务类别

服务类别	说明
auth (authpriv)	主要与认证有关的机制
cron	就是例行性工作 cron/at 等产生信息记录的地方；

daemon	与各个 daemon(服务进程) 有关的信息；
kern	内核核 (kernel) 产生信息的地方
lpr	与打印相关的信息
mail	与邮件相关的信息
news	与新闻组相关的信息
syslog	syslogd 程序本身产生的信息
user, uucp, local0 ~ local7	与 Unix like 机器本身有关的一些信息。

日志信息等级

等级	等级名称	说明
1	info	仅是一些基本的信息说明而已；
2	notice	需要注意的信息；
3	warning (warn)	警示的信息， info, notice, warn 这三个信息都是在告知一些基本信息而已，应该还不至于造成一些系统运行困扰；

4	err (error)	一些重大的错误信息
5	crit	比 error 还要严重的错误
6	alert	警告警告，已经很有问题的等级，比 crit 还要严重
7	emerg (panic)	疼痛等级，意指系统已经几乎要死机的状态

. ： 代表比后面还要高的等级 (含该等级) 都被记录下来的意思。如 mail .info

.=: 代表所需要的等级就是后面接的等级而已

.!: 代表不等于，除了该等级外的其他等级都记录。

.*:代表说所有等级的信息都记录

日志文件的轮替(logrotate)

日志文件随着时间会变得越来越多，此时就需要进行日志文件轮替。以控制日志文件规模.logrotate 就是做这个用的

日志轮替程序配置文件包括

/etc/logrotate.conf： 记录整体轮替配置信息
/etc/logrotate.d/* : 记录各个服务类型的轮替配置信息

其实可以在/etc/logrotate.conf 中定义配置各种服务类型的日志轮替配置信息，为了方便管理将每个服务类型的日志轮替配置信息形成独立文件存储在/etc/logrotate.d/*

```
[root@localhost logrotate.d]# ll
-rw-r--r-- 1 root root  144 2012-02-23 acpid
-rw-r--r-- 1 root root  288 2007-11-12 conman
.....
-rw-r--r-- 1 root root  100 10-02 06:17 wpa_supplicant
-rw-r--r-- 1 root root  100 2012-07-26 yum
```

logrotate 的配置文件

```
[root@localhost ~]# vim /etc/logrotate.conf

=>下面为默认值。如不单独配置将才有下面的默认值
weekly    <==默认每周进行一次 rotate 的工作
rotate 4  <==默认保留4个登录文件
create    <==以新创建文件继续存储日志文件
 #compress <==被更动的登录文件是否需要压缩? 如果登录文件太大则可考虑此参数启动


include /etc/logrotate.d

/var/log/wtmp {      <==仅针对 /var/log/wtmp 所配置的参数
   monthly          <==每个月一次，取代每周!
   minsize 1M       <==文件容量一定要超过 1M 后才进行
   create 0664 root utmp <==指定新建文件的权限与所属帐号/群组
   rotate 1         <==仅保留一个
}
```

日志轮替规则

当第一次执行轮替，原本日志文件 msg 被重命名为 msg1,同时创建新的 msg 以记录新的日志信息，当的当第二次执行轮替，日志文件 msg1 被重命名为 msg2, msg 重命名为 msg1,同时创建新的 msg 以记录新的日志信息,依次类推，日志文件最多保存的数量为 rotate 这个属性所指定的数值

日志轮替文件语法

正如前面看到的。日志轮替方式是写在配置文件中的，这里只做简单的说明

```
[root@localhost logrotate.d]# cat named

/var/log/named.log {
    missingok
create 0644 named named
sharedscripts
    postrotate
        /sbin/service named reload  2> /dev/null > /dev/null || true
    endscript
}
```

如上前面文件。Logrotate 轮替信息可以分为2部分

1. 内部参数

2. 引用外部执行

引用外部命令来进行额外的命令下达，这个配置需与 sharedscripts .... endscript 配置合用才行。至于可用的环境为：

prerotate：在启动 logrotate 之前进行的命令

postrotate：在做完 logrotate 之后启动的命令

**内部参数 **

compress 通过 gzip 压缩转储以后的日志

nocompress 不需要压缩时，用这个参数

copytruncate 用于还在打开中的日志文件，把当前日志备份并截断

nocopytruncate 备份日志文件但是不截断

create mode owner group 转储文件，使用指定的文件模式创建新的日志文件

nocreate 不建立新的日志文件

delaycompress 和 compress 一起使用时，转储的日志文件到下一次转储时才压缩

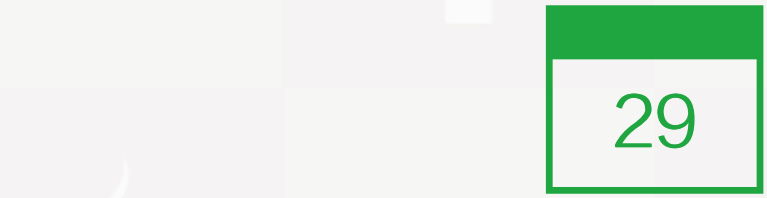
nodelaycompress 覆盖 delaycompress 选项，转储同时压缩。
errors address 专储时的错误信息发送到指定的 Email 地址
ifempty 即使是空文件也转储，这个是 logrotate 的缺省选项。
notifempty 如果是空文件的话，不转储
mail address 把转储的日志文件发送到指定的 E-mail 地址
nomail 转储时不发送日志文件
olddir directory 转储后的日志文件放入指定的目录，必须和当前日志文件在同一个文件系统
noolddir 转储后的日志文件和当前日志文件放在同一个目录下
prerotate/endscript 在转储以前需要执行的命令可以放入这个对，这两个关键字必须单独成行
postrotate/endscript 在转储以后需要执行的命令可以放入这个对，这两个关键字必须单独成行
daily 指定转储周期为每天
weekly 指定转储周期为每周
monthly 指定转储周期为每月
rotate count 指定日志文件删除之前转储的次数，0 指没有备份，5 指保留5 个备份
tabootext [+] list 让 logrotate 不转储指定扩展名的文件，缺省的扩展名是：.rpm-orig, .rpmsave, v, 和 ~
size size 当日志文件到达指定的大小时才转储，Size 可以指定 bytes (缺省)以及 K (sizek)或者M (sizem).

执行日志轮替

语法：logrotate [-vf] logfile
选项与参数：
-v:显示运行过程
-f:不论是否符合配置文件的数据，强制每个日志文件都进行轮替操作

举例

```
[root@localhost /]# logrotate -vf /etc/logrotate.conf
.....................
[root@localhost /]# ll /var/log/messages*
-rw------- 1 root root      50 03-28 14:00 /var/log/messages
-rw------- 1 root root      50 03-28 14:00 /var/log/messages.1
-rw------- 1 root root      50 03-28 13:59 /var/log/messages.2
-rw------- 1 root root  333879 03-28 10:44 /var/log/messages.3
-rw------- 1 root root 1289326 03-25 08:50 /var/log/messages.4
```



29

Boot Loader

Bootloader 的作用是加载内核到内存，使内核开始执行，Grub 是 linux 上面一个功能强大的 bootloader,当我们登陆系统就会看到如下界面，它就是 Grub 的 menu.lst，通过它我们可以选择不同的系统（多操作系统时）(这里介绍的是 grub，ubantu 使用的是 grub2，两者存在很多差异)

功能介绍

menu .list

menu.lst 是 Grub 的开机菜单，里面的配置决定了我们去哪里读取内核与 initrd

```
default=0
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title CentOS (2.6.18-371.el5)
        root (hd0,0)
        kernel /vmlinuz-2.6.18-371.el5 ro root=LABEL=/ rhgb quiet rgb=0x317
        initrd /initrd-2.6.18-371.el5.img
```

default: 默认启动项这个与 title 对照，menu 中配置了几个 title，启动菜单就有几个选择，0代表使用第一个 title 内容

timeout:启动是的倒数读秒操作，-1代表不进行倒数读秒

splashimage:menu.lst 的背景图片

hiddenmenu:隐藏菜单

root:代表内核文件放置那个分区，不是根目录的意思

kernel:后面接内核文件名，在后面指定根目录挂载到那个分区

initrd：后面接虚拟文件系统文件名（其实就是指定它的位置）

(hd num1,num2): hd 代表在 grub 中硬盘与分区的代号，num1 代表硬盘代号(0开启)。Num2 代表分区号（0开始）.比如：内核文件存储在第一块硬盘的 D 分区（第2个分区），可以表示为(hd 0,1)

举例：menu.lst 配置说明

```
[root@localhost ~]# find / -name vmlinuz-2.6.1*;df
/boot/vmlinuz-2.6.18-371.el5
文件系统             1K-块       已用     可用 已用% 挂载点
/dev/sda1              101086     11727     84140  13% /boot
```

通过上面我们可以看到内核文件存在/boot/vmlinuz-2.6.18-371.el5下，同时/boot 挂载到硬盘的第一分区，因此内核文件存储位置可以写成（hd 0,0）

```
root (hd0,0)
kernel  /vmlinuz-2.6.18-371.el5 ro root=LABEL=/ rhgb quiet rgb=0x317
initrd  /initrd-2.6.18-371.el5.img
```

由于前面指定了 root 了因此后面的 kernel，initrd 只需写接下来的路径就可以了如：/vmlinuz-2.6.18-371.el5，接下来为根据 LABEL 挂载根目录到分区 root=LABEL=/
同样上面配置也可以写成这样

```
kernel  (hd0,0)/vmlinuz-2.6.18-371.el5 ro root=LABEL=/ rhgb quiet rgb=0x317
initrd  (hd0,0)/initrd-2.6.18-371.el5.img
```

chain loader 控制权转移

我们知道 boot loader 装在 MBR 或者分区的第1扇区中，chain loader 功能就是将控制权交给指定分区的 bootloader 让其进行加载相应的内核文件

```
title /dev/sda1 boot sector
      root (hd0,0)
      chainloader +1
```

比如我们的 LINUX 系统的 bootloader 装在了第1个硬盘第1个分区，那 bootloader 的位置就是第1块硬盘的第一个分区的第一扇区，因此
root (hd0,0)指定分区与磁盘，这里是第一个磁盘的第一个分区
chainloader +1 指定为第一扇区

同样假如我们 LINUX 系统的 bootloader 再在整个硬盘的 MBR 中，那可以这么指定

```
title   MBR loader
      root (hd0)
      chainloader +1
```

由于 MBR 位置为硬盘的一个扇区，因此
root (hd0)指定第一个硬盘
chainloader +1指定为第一扇区

多系统并存环境

如果想让一台机器上存在多个操作系统可以通过控制权转移将控制权交给指定分区的 loader 进行加载相应的操作系统

假如，我的机器只有一个硬盘，我想在第1分区装 WINXP,第2个分区装 linux,那个就可以在 menu.list 中设置2个选项，第1个选项为 winxp,第2个选项为 linux,当选择第一个时控制权交给第1分区的 bootloader,当选择第2分区时将控制权交给第2个分区的 bootloader 即 linux 的 loader

但是这里需要先安装 WINXP 在安装 LINUX 因为 window 不具有控制权转移功能

Grub 安装

Grub 安装分为3个步骤

1. grup 配置文件安装
2. menu.list 文件编辑
3. grup 主程序安装到 MBR 或分区第一扇区

步骤1：grub 配置文件安装

语法：grub-install[--root-directory=DIR] 设备代号

选项与参数

--root-directory：当指定 DIR 是，grub 配置文件安装在 DIR/boot/grub

如不指定此属性，此默认安装在/boot/grub

```
[root@localhost ~]# grub-install /dev/sda
Installation finished. No error reported.
This is the contents of the device map /boot/grub/device.map.
Check if this is correct or not. If any of the lines is incorrect,
fix it and re-run the script `grub-install'.
 # this device map was generated by anaconda

(hd0)     /dev/sda
[root@localhost lib]# ll /boot/grub/
-rw-r--r-- 1 root root   7584 03-31 10:52 e2fs_stage1_5
-rw-r--r-- 1 root root   7456 03-31 10:52 fat_stage1_5
```

步骤2：编写 menu.list

```
[root@localhost lib]# vim /boot/grub/menu.lst
default=0
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title CentOS (2.6.18-371.el5)
        root (hd0,0)
        kernel /vmlinuz-2.6.18-371.el5 ro roo
        initrd /initrd-2.6.18-371.el5.img
```

步骤3：grup 主程序安装到 MBR 或分区第一扇区

Grubshell 的简单语法

root(hdx,x):选择含有 grub 目录的那个分区

find 文件路径，

find 路径/stage1 查找是否有安装信息

find 路径/vmlinuz…. 查找内核文件

setup(hdx,x) 安装 grub 到分区的第1扇区

setup(hd 0) 安装 grub 到 MBR 中

```
[root@localhost /]# grub =>进入grub shell
    GNU GRUB  version 0.97  (640K lower / 3072K upper memory)

 [ Minimal BASH-like line editing is supported.  For the first word, TAB
   lists possible command completions.  Anywhere else TAB lists the possible
   completions of a device/filename.]

grub> root (hd0,0)
 Filesystem type is ext2fs, partition type 0x83

grub> find /vmlinuz-2.6.18-371.el5
 (hd0,0)

grub> setup (hd0) => 安装到MBR中
 Checking if "/boot/grub/stage1" exists... no
 Checking if "/grub/stage1" exists... yes
 Checking if "/grub/stage2" exists... yes
 Checking if "/grub/e2fs_stage1_5" exists... yes
 Running "embed /grub/e2fs_stage1_5 (hd0)"...  15 sectors are embedded.
succeeded
 Running "install /grub/stage1 (hd0) (hd0)1+15 p (hd0,0)/grub/stage2 /grub/grub.conf"... succeeded
Done.

grub> setup (hd0,0) 安装到sector中
 Checking if "/boot/grub/stage1" exists... no
 Checking if "/grub/stage1" exists... yes
 Checking if "/grub/stage2" exists... yes
 Checking if "/grub/e2fs_stage1_5" exists... yes
 Running "embed /grub/e2fs_stage1_5 (hd0,0)"... failed (this is not fatal)
 Running "embed /grub/e2fs_stage1_5 (hd0,0)"... failed (this is not fatal)
 Running "install /grub/stage1 (hd0,0) /grub/stage2 p /grub/grub.conf "... succeeded
Done.

grub> quit
```

忘记 root 密码解决

开机后按下 e 进入 menu.lst 编辑模式

编辑 Kernel 信息。指定为单用户模式

按 Esc 返回刚才那个页面，按下 b, 此时系统会给你个 root 权限的 shell.使用 passwd 命令修改密码即可

30

Linux 学习记录--启动流程

第 30 章 启动流程

系统的启动过程大致可以分为如下几个步骤：

1. 加载 BIOS 的硬件信息与进行自我测试，并依据设置取得第一个可启动的设备
2. 读取并执行第一个启动设备内 MBR 的 bootloader
3. 依据 boot loader 的设置加载 kernel,kernel 会开始检测硬件与加载驱动程序
4. 在硬件驱动成功后，Kernel 会主动调用 init 进程，而 init 进程回去的 run-level 信息
5. Init 执行/etc/rc.d/rc.sysinit 文件来准备软件执行的操作系统（网络，时区等）
6. Init 执行 run-level 的各个服务的启动
7. Init 执行/etc/rc.d/rc.local 文件
8. Init 执行终端机模拟程序 mingetty 来启动 login 进程，最后就等待用户登录

名词解释

BIOS:开机的时候计算机系统会主动执行的程序，它会识别第一个可开机的设备
MBR:第一个可开机设备的第一个扇区内的主引导分区，内包含 bootloader
Boot loader:可进行内核与虚拟文件系统加载的软件
虚拟文件系统(initrd):内存中仿真的根目录，用于当 loader 不能进行根目录挂载时使用

根据以上启动流程：我的理解上这样的一个流程

设备通电开机 è BIOS 执行 è 认识第一个开机设备 è 将 MBR 内容载入内存
è bootloader 执行 è 载入内核与 initrd è 根目录挂载 è init è 登陆系统

Bootloader 能够识别操作系统文件格式，所以可以解压缩内核到内存中执行，内核在执行中进行测试与驱动各个设备，由于驱动(内核模块)是挂载到/lib/modules 下，如要读取必要先挂在根目录，可挂载根目录必须读取驱动。此处就产生的了矛盾，因此为了解决这个问题引入了 initrd, BootLoader 将initrd 解压缩到内存并在内存中形成一个仿真的根目录，加载启动时是必需的驱动，如磁盘的驱动，并完成根目录实际的挂载，以完成后续的操作

Init 处理流程

(此流程是针对 centos，不同的 UNIX LIKE 流程不尽相同，但思路是一致的)
在内核加载完驱动后，硬件就已经准备完毕了，此时内核会主动调用第一个进程，即，Init 进程，init 进程开始开始执行软件环境，如服务的开启，网络的设置等。Init 所以操作都会记录在其配置文件中/etc/inittab

其大致流程如下

Init è 读取 init 配置文件并执行里面的命令 è 执行 runlevel è 执行相应的服务和服务 è 登录图形页面(runlevel 5)

init 配置文件

```
d:5:initdefault: =>默认启动级别是5

 # System initialization.

si::sysinit:/etc/rc.d/rc.sysinit =>初始化操作

l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6

 # Trap CTRL-ALT-DELETE => 重新启动组合键

ca::ctrlaltdel:/sbin/shutdown -t3 -r now

 # When our UPS tells us power has failed, assume we have a few minutes

 # of power left.  Schedule a shutdown for 2 minutes from now.

 # This does, of course, assume you have powerd installed and your

 # UPS connected and working correctly.

pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"
```

```
 # If power was restored before the shutdown kicked in, cancel it.

 # If power was restored before the shutdown kicked in, cancel it.

pr:12345:powerokwait:/sbin/shutdown -c "Power Restored; Shutdown Cancelled"


 # Run gettys in standard runlevels

1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6

 # Run xdm in runlevel 5

x:5:respawn:/etc/X11/prefdm -nodaemon
```

init 处理工作

1. 取得 runlevel 默认级别。这里是5
2. 执行 script /etc/rc.d/rc.sysinit 进行初始化
3. 因为 run level 是5，所以执行5:5:wait:/etc/rc.d/rc 5
4. 设置重启组合键[ctrl]+[alt]+[del]
5. 设置不断电系统 pr 与 pf
6. 启动6个终端机
7. 启动图形界面

runlevel 介绍

Init 配置文件(/etc/inittab)最重要的就是这个运行时启动级别，对于 XWindow 来说将 run level 分为7个等级，每个等级只是所启动的服务不尽相同 0- halt:系统直接关机

1- single user mode:单用户模式

2- multi user without NFS

3- full multi user mode: 命令行模式

4- unused

5- X11:图形模式

6- rboot:重启模式

举例：

```
[root@bogon ~]# runlevel =>查看当前运行等级
N 5
[root@bogon ~]# init 3 =>执行runlevel 3
```

runlevel 程序与服务

/etc/rc.d/下包含不同等级的 run level 的启动内容，前面提到过不同的 run level 差别在于启动的服务不同

```
[root@bogon ~]# ll /etc/rc.d/rc5.d/
lrwxrwxrwx 1 root root 17 02-18 20:06 K01dnsmasq -> ../init.d/dnsmasq
lrwxrwxrwx 1 root root 19 02-18 20:14 K01rgmanager -> ../init.d/rgmanager
….
lrwxrwxrwx 1 root root 16 02-18 20:07 S56xinetd -> ../init.d/xinetd
lrwxrwxrwx 1 root root 18 02-18 20:06 S80sendmail -> ../init.d/sendmail
…….
lrwxrwxrwx 1 root root 11 02-18 20:05 S99local -> ../rc.local
lrwxrwxrwx 1 root root 21 02-18 20:12 S99modclusterd -> ../init.d/modclusterd
```

上面可以看到的 runlevel 5下面启动的服务

K【数字】：K 代表停止。后面的数字代表的停止的顺序

S【数字】：S 代表开启。后面的数字代表的开启的顺序

在服务里面提到过为解决服务的依赖性因此需要制定服务启动停止的顺序

用户自定义开机启动程序(rc.local)

从上面服务 runlevel5启动内容中有一项是

lrwxrwxrwx 1 root root 11 02-18 20:05S99local -> ../rc.local

这个 script 可以添加自定义的命令，从整个启动流程来看，它处于用户登录之前进行的

```
#!/bin/sh

#

# This script will be executed *after* all the other init scripts.

# You can put your own initialization stuff in here if you don't

# want to do the full Sys V style init stuff.
```

```
touch /var/lock/subsys/local
```

用户执行自定义命令位置点对比

~/.bash_profile 文件

文件加载时机：用户登录完毕 。 特点：只针对单一登录用户的个人设置

~./bash_loyout:

文件加载时机：用户注销时。

特点：只针对单一登录用户的个人设置

/etc/rc.local

文件加载时机：init 启动流程。在用户为登录之前

特点：不区分用户的命令

/etc/init.d/

文件加载时机：以服务的形式加载

特点：可以针对不同的 runlevel 设置是否开启服务

(此处无时机项目经验，只是简单对比，对于安全性方面无考虑)

虚拟文件系统(initrd)

前面提到了 initrd 的作用，现在查看下这个里面究竟包含什么

```
[root@localhost tmp]# mkdir /tmp/initrd
[root@localhost tmp]# cp /boot/initrd-2.6.18-371.el5.img /tmp/initrd/
[root@localhost tmp]# ll ./initrd/
-rw------- 1 root root 2748313 03-31 09:59 initrd-2.6.18-371.el5.img
[root@localhost tmp]# mv ./initrd/initrd-2.6.18-371.el5.img ./initrd/initrd-2.6.18-371.el5.gz
[root@localhost tmp]# ll ./initrd/
-rw------- 1 root root 2748313 03-31 09:59 initrd-2.6.18-371.el5.gz
[root@localhost tmp]# cd initrd/
[root@localhost initrd]# gzip -d ./initrd-2.6.18-371.el5.gz
[root@localhost initrd]# ll

-rw------- 1 root root 6332928 03-31 09:59 initrd-2.6.18-371.el5
[root@localhost initrd]# file initrd-2.6.18-371.el5
initrd-2.6.18-371.el5: ASCII cpio archive (SVR4 with no CRC)
[root@localhost initrd]# cpio -ivcdu <./initrd-2.6.18-371.el5
.....
[root@localhost initrd]# ll
```

```
drwx------ 2 root root    4096 03-31 10:03 bin
drwx------ 3 root root    4096 03-31 10:03 dev
drwx------ 2 root root    4096 03-31 10:03 etc
-rwx------ 1 root root    2708 03-31 10:03 init
drwx------ 3 root root    4096 03-31 10:03 lib
drwx------ 2 root root    4096 03-31 10:03 proc
lrwxrwxrwx 1 root root       3 03-31 10:03 sbin -> bin
drwx------ 2 root root    4096 03-31 10:03 sys
drwx------ 2 root root    4096 03-31 10:03 sysroot
=>和根目录结构很像

 [root@localhost initrd]# cd ./lib/;ll
-rw------- 1 root root  31664 03-31 10:03 ata_piix.ko
-rw------- 1 root root  18060 03-31 10:03 dm-log.ko
.........
=>这里包含里就是启动时必需的内核模块
```

创建 initrd

正常安装 unix like 版本是 initrd 是不需要额外创建的，但是有时候我们可能需要将一些其他的内核模块加到 initrd 中，此时就需要创建一个新的initrd

语法：mkinitrd [-v] [--with=模块名称] initrd 文件名内核版本

31

Linux 学习记录--开机挂载错误

第 31 章 开机挂载错误

我们可以在/etc/fstab 设置开启挂载，不过如果在/etc/fstab 定义了挂载信息可是实际并不存在此分区，开机进行分区挂载时就会产生错误，导致我们不能进入操作系统，如下图

根源在于 /etc/fstab中/dev/sda6分区并不存在，只需要将这行去掉即可

```
LABEL=/              /                  ext3    defaults       1 1
LABEL=/home          /home              ext3    defaults       1 2
LABEL=/boot          /boot              ext3    defaults       1 2
tmpfs                /dev/shm           tmpfs   defaults       0 0
devpts               /dev/pts           devpts  gid=5,mode=620  0 0
sysfs                /sys               sysfs   defaults       0 0
proc                 /proc              proc    defaults       0 0
LABEL=SWAP-sda5      swap               swap    defaults       0 0
/dev/sda6            /mnt/sda6          ext3    defaults    1 2
```

可是/etc/fstab 是只读文件，正确情况下不能修改，所以需要些特殊操作

方法1：单用户模式进行重挂载修改

步骤1.设置 menu.lst 开机信息。指定为单用户

步骤2：设置重新挂载，解决只读问题

步骤3:修改文件并重新启动即可

方法2：使用其他操作系统挂载分区进行修改

Ubantu 光盘可以不需要安装而进行使用，应用此特点，我们可以将修需要修改的文件所在分区挂载到这个系统下进行修改。（我是在虚拟机实验，PC 效果是一样的）

步骤1：设置 BIOS 进行开启光盘启动

添加镜像文件，并设置通电时进行连接（PC 下不需要，这是模拟光驱）

步骤2: 试用 Ubantu

步骤3: 挂载与修改问题文件

```
root@ubuntu:~# fdisk -l

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *          63      208844      104391   83  Linux
/dev/sda2          208845    20691719    10241437+  83  Linux
/dev/sda3        20691720    30925124     5116702+  83  Linux
/dev/sda4        30925125    41929649     5502262+   5  Extended
/dev/sda5        30925188    32965379     1020096   82  Linux swap / Solaris

root@ubuntu:~# mount /dev/sda2 /mnt
root@ubuntu:~# df
Filesystem     1K-blocks    Used Available Use% Mounted on
…….
/dev/sda2        9920624 4275888   5132668  46% /mnt
root@ubuntu:~# vim /mnt/etc/fstab
root@ubuntu:~# umount /dev/sda2
```

32

Linux 学习记录--程序编译与函数库

第 32 章 程序编译与函数库

前面提到过对于机器来说只能识别0,1，我们如果让机器运行必须输入机器能够识别的语言，可是机器语言不利于人们使用可理解，因此科学家就开发出人类能看的懂的程序语言，然后再创造出“编译器”将程序语言转换为机器语言。

C 语言就是我们能够看懂的机器语言，gcc 就是 Linux 下编译器。我们通常 C 语言写的程序通过 gcc 编译后，就能成为机器能够识别的语言

gcc 程序编译

如果 LINUX 系统中为安装 GCC 编译器，可以使用下面命令安装

[root@bogon ~]# yum install gcc

gcc 常用语法

语法： gcc －c file.c

仅将源代码编译成目标文件。并不会进行链接以及生成可执行文件

gcc －o 执行文件名 源代码文件

直接生成指定名称的执行文件。但不会生成目标文件

gcc －o 执行文件名 目标文件

通过目标文件生成可执行文件

gcc [其他编译操作] －L库文件路径

查找库文件的路径默认是/usr/lib 与/ib

gcc [其他编译操作] －I 包含文件路径

查找包含文件的路径默认是/usr/include

gcc [其他编译操作] －Wall

更加严谨的编译方式，会输出很多警告信息

gcc –O [其他编译操作]

编译时依据操作环境优化执行速度

gcc [其他编译操作] －l库文件名称

编译时引入其他的库文件，其中库文件 lib 与扩展名不需要写。如引入 libm.so 文件，可写成–lm

单一程序编译

1. 编写 C 语言程序 I

```
[root@bogon code]# vim hello.c
 #include <stdio.h>
```

```
int main(void)
{
   printf("hello world!");
}
```

1. 编译

```
[root@bogon code]# gcc hello.c
[root@bogon code]# ll hello.c a.out
-rwxr-xr-x 1 root root 4947 04-05 16:07 a.out
-rw-r--r-- 1 root root   66 04-05 16:07 hello.c
```

说明：默认 gcc 编译器编译出来的执行文件 a.out, 可以使用-o 来制定编译后生产的执行文件名称

```
[root@bogon code]# gcc -o hello hello.c
[root@bogon code]# ll hello
-rwxr-xr-x 1 root root 4947 04-05 16:11 hello
```

1. 执行

```
[root@bogon code]# ./a.out
hello world!

[root@bogon code]# ./hello
hello world!
```

多文件程序编译

假设我们有 A.c ,B.c 两个程序文件，并且他们之间存在函数调用，那么当其中有一个文件更改了。是不是需要将这两个文件都重新编译？当然不需要，这就需要引入目标文件

目标文件：编译器编译源代码后生成的文件，目标文件从结构上讲，它是已经编译后的可执行文件格式，只是没有经过链接的过程。 接着上面的说，当 B.c 文件更改时，我们执行重新编译 B 文件生产目标文件。再讲整体链接即可

1. 编写 C 语言程序

```
File:A.c
 #include <stdio.h>

 #include “B.c”

int main ()
{
printf("这是第一个文件\n");
```

```
method();
}

File:B.c
 #include <stdio.h>

void method(void)
{
 printf("这是第二个文件！\n";)
}
```

1. 编译

```
root@bogon code]# gcc -c A.c B.c -I./
[root@bogon code]# ll
-rw-r--r-- 1 root root   81 04-05 16:35 A.c
-rw-r--r-- 1 root root  912 04-05 16:35 A.o
-rw-r--r-- 1 root root   80 04-05 16:34 B.c
-rw-r--r-- 1 root root  860 04-05 16:35 B.o
```

1. 链接

```
[root@bogon code]# gcc -o result A.o B.o
```

1. 执行

```
[root@bogon code]# ./result
这是第一个文件
这是第二个文件!
```

1. 更改 B.c 文件

```
 #include <stdio.h>

void method(void)
{
 printf("这是更改后第二个文件！\n";)
}
```

1. 重新编译链接执行

```
[root@bogon code]# gcc -c B.c =>只编译了B这个文件
[root@bogon code]# gcc -o result A.o B.o
[root@bogon code]# ./result
这是第一个文件
这是更改后第二个文件!
```

调用外部函数库

1. 编写 C 语言程序

```
#include<stdio.h>

#include<math.h>


int main ()
{
  float val=sin(3.14);
  printf("val值是：%f\n",val);
}
```

1. 编译与执行

```
[root@bogon code]# gcc -o sinmath sinmath.c -lm
[root@bogon code]# ./sinmath
val值是：0.001593
```

函数库介绍

函数库依照是否编译到程序内部可分为

静态函数库：通常以.a 为扩展名，编译时会整合到程序文件中

动态函数库：通常以.so 为扩展名，编译时会不会整合到程序文件中，只是在程序文件中存在一个指向的位置

因此程序执行时是不需要静态函数库的，但是需要动态函数库，使用动态函数库的好处再在可以减少程序文件的大小

动态函数库加载内存

我们知道内存的访问速度是硬盘的好几倍，如果先将动态函数库加载到内存中，那么在使用动态函数库时就会，就会提高很多效率

语法：ldconfig[-f 需要缓存函数库信息所在文件] [-C 已缓存函数库信息所在文件]

ldconfig -p 列出已缓存函数库信息

需要缓存函数库信息所在文件:在这个文件中记录所有需要缓存的的函数库默认值是

/etc/ ld.so.conf

举例：查看下我的系统下缓存的函数库

```
[root@bogon etc]# vim ld.so.conf
include ld.so.conf.d/*.conf
```

```
[root@bogon etc]# cd ld.so.conf.d/
[root@bogon ld.so.conf.d]# ll
-rw-r--r-- 1 root root  15 2013-01-23 mysql-i386.conf
-rw-r--r-- 1 root root  17 2013-01-09 openais-athlon.conf
-rw-r--r-- 1 root root  20 2012-08-20 qt-i386.conf
-rw-r--r-- 1 root root 276 02-22 19:23 vmware-tools-libraries.conf
-rw-r--r-- 1 root root  19 2013-08-07 xulrunner-32.conf
[root@bogon ld.so.conf.d]# vim mysql-i386.conf
[root@bogon ld.so.conf.d]# ll /usr/lib/mysql
lrwxrwxrwx 1 root root      26 02-18 20:03 libmysqlclient_r.so.15 -> libmysqlclient_r.so.15.0.0
-rwxr-xr-x 1 root root 1460684 2013-01-23 libmysqlclient_r.so.15.0.0
lrwxrwxrwx 1 root root      24 02-18 20:03 libmysqlclient.so.15 -> libmysqlclient.so.15.0.0
-rwxr-xr-x 1 root root 1452764 2013-01-23 libmysqlclient.so.15.0.0
-rwxr-xr-x 1 root root   13220 2013-01-23 mysqlbug
-rwxr-xr-x 1 root root    6215 2013-01-23 mysql_config
```

已缓存函数库信息所在文件：这个文件中记录了已经缓存的函数库，默认文件为

/etc/ld.so.cache ,通过-p 查询到的信息就是从这个文件读取而来

举例：查看所有已缓存的函数库

```
[root@bogon ld.so.conf.d]# ldconfig -p|more
947 libs found in cache `/etc/ld.so.cache'
        libz.so.1 (libc6) => /lib/libz.so.1
        libz.so.1 (libc6) => /usr/lib/libz.so.1
        libx11globalcomm.so.1 (libc6) => /usr/lib/libx11globalcomm.so.1
  ……..
```

查看程序所包含的动态函数库

语法：ldd –v 文件名

-v:列出所有函数库信息

举例：

```
[root@bogon ld.so.conf.d]# ldd  /usr/bin/passwd
        linux-gate.so.1 =>  (0x00ddc000)
        libuser.so.1 => /usr/lib/libuser.so.1 (0x007c5000)
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x05c74000)
       ……
```

make 编译

如果一个程序中有很多文件，那么还像上面那样讲每个文件列出来在进行编译就会很麻烦。因此这种时候就需要使用 make 工具了

Make 编译好处
简化编译时所需的指令
若在编译完成后，修改了某个源文件，只会针对修改的文件编译

Make 使用方法
Make 是有个二进制文件，其会查找当前目录下的 Makefile 文件，根据其里面定义的内容执行操作。 Makefile 里面包含了若干目标与操作

其基本关于规则如下
目标:
操作

```
REST2HTML=html.py --compact-lists --date --generator

all: user_manual.html dev_manual.html

user_manual.html: user_manual.rst
	$(REST2HTML) user_manual.rst user_manual.html

dev_manual.html: dev_manual.rst
	$(REST2HTML) dev_manual.rst dev_manual.html

clean:
	rm *.html
```

以上内容分为三个目标 all，user_manual.html，clean，其下面分别对应的是其操作
我们可以通过 make 后面参数为目标进行执行。如：make clean

Tarball 的安装

由于 Unix like 具有很多种。因此一个软件安装包不可能适用所有所本，因此有时我们需要根据软件提供者提供的源码自行编译，以满足在自己的操作系统上运行

大部分软件开发包编译与安装的流程大致是这样的

1. 讲压缩文件解压缩
2. 解压缩后执行里面的 configure 文件，其作用就是建立 makefile 文件
3. Make clean:清理一些上次操作的残留
4. Make :默认操作进行编译的行为
5. Make install:安装

说明：安装前如果有安装文档最好先查阅

3-4步骤 不一定都存在。可查看 makefile 内容判断具体包括哪些目标

举例：

```
[root@bogon shared]# tar -zxvf ntp-4.2.4p7.tar.gz -C /tmp
…..
[root@bogon ntp-4.2.4p7]# ./configure --prefix=/usr/loacl/ntp
=> --prefix=/usr/loacl/ntp 为指定安装目录
[root@bogon ntp-4.2.4p7]# ll Makefile
-rw-r--r-- 1 root 6011 23950 04-05 21:40 Makefile
=>生成了Makefile文件
root@bogon ntp-4.2.4p7]# ll Makefile
root@bogon ntp-4.2.4p7]# make clean
root@bogon ntp-4.2.4p7]# make
root@bogon ntp-4.2.4p7]# make install
```

本文出自 “StarFlex” 博客，请务必保留此出处http://tiankefeng.blog.51cto.com/8687281/1372503

33

Linux 学习记录--软件安装 RPM|SRPM|YUM

第 33 章 软件安装RPM|SRPM|YUM

前面说到了软件安装可以直接下载源码压缩版编译安装。还有一种安装形式是使用厂商提供给用户的安装文件。厂商在他们的系统上编译好用户所需要的软件，然后将编译好的软件发布给用户使用。

目前厂商发布软件机制主要分为2大类
Dpkg:由 Debian Linux 社区开发，B2D,Ubantu 等 Linuxdistributions 使用就是这种机制
RPM:由 Red Hat 开发，CentOs,SuSe 使用就是这种机制
这两种机制安装软件会先检测前驱软件是否存在，如果不存在则不安装.如软件 A 安装。需系统内含有软件 B，那么不存在则不会安装软件
为了解决这种因依赖问题而导致软件不能安装厂商又提供了在线升级机制，简单的说就是先将前驱软件都安装以保证要安的软件能正确安装

Dpkg 机制对应的在线升级机制 APT
RPM 机制对应的在线升级机制 YUM

RPM 软件管理程序

Rpm 软件相关信息会写入/var/lib/rpm 目录下的数据库文件内，未来软件升级以及版本比较都源自这个数据库，查询已安装 RPM 软件也会查询这个数据库

RPM 安装与升级

前面提到过 RPM 软件的安装前会检查前驱依赖的软件是否已安装，如果为安装则此次安装不会进行，同时需要说明 PRM 机制软件是厂商根据特定系统所提供，因此不同的 Linux distributions，以及不同版本之间 PRM 机制软件是不能相互安装的

语法：rpm –[i|F|U]vh 软件名
选项与参数
-i: 安装
-U：后接的软件如果没有安装过则安装，如果安装过且版本较旧则更新
-F:后接的软件如未安装则不进行安装，如果安装过且版本较旧则更新
-v:查看详细安装信息
-h:显示安装进度

举例1:安装

```
[root@localhost ~]#
rpm -ivh /media/CentOS_5.10_Final/CentOS/pam-devel-0.99.6.2-12.el5.i386.rpm
Preparing...                ########################################### [100%]
   1:pam-devel              ########################################### [100%]
```

举例2：存在依赖的安装

```
[root@localhost ~]# rpm -i /media/CentOS_5.10_Final/CentOS/ant17-junit-1.7.1-1jpp.0.i386.rpm
error: Failed dependencies:
        ant17 = 0:1.7.1-1jpp.0 is needed by ant17-junit-1.7.1-1jpp.0.i386
        junit is needed by ant17-junit-1.7.1-1jpp.0.i386
=>提示其前驱软件为安装
```

RPM 查询

语法： rpm －qa

rpm －q[iR] 存在于系统的软件名

rpm －qf 存在于系统的某个文件名

rpm －qp[iR] 未安装的某个文件名

－q:进查询后面接的软件名是否安装

－qa：列出所有已安装的软件信息

－qi： 列出后面接软件的详细信息

－qR:列出与该软件有关的依赖软件所含的文件

－qf:由后面接的文件名称找出还文件属于哪一个以安装的软件

查询某个未安装软件包含文件的信息

－qp[iR]:iR 属于与上面说的一致

```
[root@localhost ~]# rpm -q pam-devel =>不需要列出版本号
pam-devel-0.99.6.2-12.el5

[root@localhost ~]# rpm -qi pam-devel
Name        : pam-devel                    Relocations: (not relocatable)
Version     : 0.99.6.2                     Vendor: CentOS
Release     : 12.el5                       Build Date: 2013年01月09日星期三 13时30分55秒
Install Date: 2014年04月08日星期二 14时19分02秒      Build Host: builder17.centos.org
Group       : Development/Libraries        Source RPM: pam-0.99.6.2-12.el5.src.rpm
Size        : 504034                       License: GPL or BSD
Signature   : DSA/SHA1, 2013年01月10日星期四 03时18分55秒, Key ID a8a447dce8562897
URL         : http://www.us.kernel.org/pub/linux/libs/pam/index.html
Summary     : Files needed for developing PAM-aware applications and modules for PAM
Description :
PAM(可插入验证模块)是一个系统安全工具。它允许
系统管理员无需重新编译处理验证的程序而设置验证
策略。该软件包包括用于建构留意 PAM 的程序和 PAM
使用的模块所需的头文件和静态库。

[root@localhost ~]# rpm -qR pam-devel
libpam.so.0
libpam_misc.so.0
libpamc.so.0
pam = 0.99.6.2-12.el5
rpmlib(CompressedFileNames) <= 3.0.4-1
```

```
rpmlib(PayloadFilesHavePrefix) <= 4.0-1

[root@localhost ~]# rpm -qf /usr/lib/libpam.so
pam-devel-0.99.6.2-12.el5

[root@localhost ~]# rpm -qa
tomcat5-servlet-2.4-api-5.5.23-0jpp.40.el5_9
xml-commons-resolver-1.1-1jpp.12
……
```

RPM 卸载与重建数据库

卸载：rpm -e 软件名

重建数据库：rpm -rebuilddb

YUM 在线升级

前面说到 RPM 软件安装是如果存在前驱软件且未安装此次安装就不会进行，为了解决这个问题，就引入了 YUM 在线升级机制，简答的说，YUM 在线升级机制就是将需要安装的软件的前驱软件(RPM)事先都进行安装以保障软件的顺利安装

YUM 安装与更新

语法：yum [option] [install|update]
Option:
-y:安装过程中询问用户操作，默认是 yes
-installroot=路径：软件安装路径
install：安装操作
update 更新操作

举例：

```
[root@bogon ~]# yum -y install pam-devel
Loaded plugins: fastestmirror, security
Loading mirror speeds from cached hostfile
 * base: mirrors.yun-idc.com
 * extras: mirrors.yun-idc.com
 * updates: mirrors.yun-idc.com
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package pam-devel.i386 0:0.99.6.2-12.el5 set to be updated
--> Finished Dependency Resolution
.....
  Installing     : pam-devel                                                 1/1
Installed:
  pam-devel.i386 0:0.99.6.2-12.el5
Complete!
```

YUM 查询

语法：yum[search|info |provides]软件名
yum list|list updates

search: 查询后接名称的相关的软件

info: 查询后接软件的相关信息

provides:查询提供后接文件的软件有哪些

list:列出服务器上所提供的软件

list updates：列出服务器上可供升级的软件

举例：

```
[root@bogon ~]# yum search pam
…….
pam.i386 : A security tool which provides authentication for applications
pam-devel.i386 : Files needed for developing PAM-aware applications and modules for PAM
…..
[root@bogon ~]# yum info pam-devel
……
Installed Packages
Name       : pam-devel
Arch       : i386
Version    : 0.99.6.2
Release    : 12.el5
Size       : 492 k
Repo       : installed
Summary    : Files needed for developing PAM-aware applications and modules for PAM
URL        : http://www.us.kernel.org/pub/linux/libs/pam/index.html
License    : GPL or BSD
Description: PAM(可插入验证模块)是一个系统安全工具。它允许
           : 系统管理员无需重新编译处理验证的程序而设置验证
           : 策略。该软件包包括用于建构留意 PAM 的程序和 PAM
           : 使用的模块所需的头文件和静态库。

[root@bogon ~]# yum provides /bin/sh
……..
bash-3.2-32.el5_9.1.i386 : The GNU Bourne Again shell (bash) version 3.2
Repo       : base
Matched from:
Filename   : /bin/sh
```

YUM 删除

语法：yum [- y]remove 软件名

举例：

```
[root@bogon ~]# yum remove pam-devel
Loaded plugins: fastestmirror, security
Setting up Remove Process
Resolving Dependencies
--> Running transaction check
---> Package pam-devel.i386 0:0.99.6.2-12.el5 set to be erased
--> Finished Dependency Resolution
.............
Transaction Test Succeeded
Running Transaction
  Erasing        : pam-devel                                                    1/1
Removed:
  pam-devel.i386 0:0.99.6.2-12.el5
```

YUM 软件组

前面说的 YUM 安装都是一个一个安装软件，当我们安装一个复杂的功能时可能需要安装很多个软件，比如安装 KDE 桌面系统，其包括很多软件，如果在一个个安装就会很费事。此时就可以使用软件组功能。（至于有哪些软件组。这就需要服务器提供了）

语法： yum grouplist

yum [groupinfo|groupinstall|groupremove] 软件组名

举例：

```
[root@bogon ~]# yum grouplist
Installed Groups:=>已经安装的软件组
   DNS 名称服务器
   FTP 服务器
  .......
Available Groups: =>可安装的软件组
   Beagle
   Eclipse
.......
Done
```

YUM 服务器配置

Yum 软件在线升级信息都要从远端的服务器端来获取数据，有时我们需要更改下服务器地址，选择些离我们较近，或者资源较好的服务器以提高传输效率，那么服务器配置信息在哪里设置的?

查看系统中配置服务器站点

语法：yum repolist all

```
root@bogon ~]# yum repolist all
repo id              repo name                    status
C5.0-base            CentOS-5.0 - Base            disabled
base                 CentOS-5 - Base              enabled
extras               CentOS-5 - Extras            enabled
updates              CentOS-5 - Updates           enabled
=>只有 states 是 enabled 才是被激活的，上面有三个站点是被激活的
```

当我们查询安装时会看到如下提示信息，就代表从上面三个激活站点查询到的数据

Loaded plugins: fastestmirror, security

Loading mirror speeds from cached hostfile

*base: mirrors.yun-idc.com

*extras: mirrors.yun-idc.com

*updates: mirrors.yun-idc.com

配置服务器站点

服务器配置信息都在/etc/yum.repos.d/目录下

```
[root@localhost /]# ll /etc/yum.repos.d/
-rw-r--r-- 1 root root 1926 04-02 09:21 CentOS-Base.repo
-rw-r--r-- 1 root root  636 10-08 04:57 CentOS-Debuginfo.repo
-rw-r--r-- 1 root root  626 10-08 04:57 CentOS-Media.repo
-rw-r--r-- 1 root root 7574 10-08 04:57 CentOS-Vault.repo
-rw-r--r-- 1 root root  342 04-02 09:26 mystation.repo
```

以上可见分为4个容器，每个容器内配置了一些列站点

```
[root@localhost /]# vim /etc/yum.repos.d/CentOS-Base.repo
=>查看包含了哪些站点
[base]
name=CentOS-$releasever - Base
mirrorlist=http://mirrorlist.centos.org/?release=$releasever&arch=$basearch&repo=os
 #baseurl=http://mirror.centos.org/centos/$releasever/os/$basearch/

gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5
enabled=1
```

名称说明：

[base]：代表站点的名字！中刮号一定要存在

name：只是说明一下这个容器的意义而已

mirrorlist=：列出这个容器可以使用的映射站台，如果不想使用，可以注解到这行；

baseurl=：因为后面接的就是容器的实际网址！

enable=1：就是让这个容器被启动。如果不想启动可以使用 enable=0 。

gpgcheck=1：这就是指定是否需要查阅 RPM 文件内的数码签章！

gpgkey=：就是数码签章的公钥档所在位置。使用默认值即可

举例：管理一个站点

```
[base]
name=CentOS-$releasever - Base
mirrorlist=http://mirrorlist.centos.org/?release=$releasever&arch=$basearch&repo=os
 #baseurl=http://mirror.centos.org/centos/$releasever/os/$basearch/

gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5
enabled=0

[root@bogon ~]# yum search gcc
Loaded plugins: fastestmirror, security
Loading mirror speeds from cached hostfile
 * extras: mirrors.btte.net
 * updates: mirrors.btte.net
extras                                                   | 2.1 kB     00:00
updates
=>查询站点只剩下2个了
```

对于自定义的服务器地址需要在/etc/yum.repos.d/目录下新建.repo 文件，在里面配置自己的服务器站点

本文出自 “StarFlex” 博客，请务必保留此出处http://tiankefeng.blog.51cto.com/8687281/1372503

34

内核|内核模块编译

（对于内核的知识觉得了解不够，等学习完LFS再来详细整理下这方面的知识）

内核：系统上面的一个文件，这个文件包含了驱动主机各项硬件的检测程序和驱动模块。
计算机真正工作的是硬件，内核是用来控制这些硬件工作的（主要通过硬件驱动），如果我们需要硬件来完成某项工作时需要内核的帮助才能完成
内核模块：编译成模块的驱动程序。既然内核中已经包括了驱动程序，那么为什么需要驱动模块呢？因为硬件发展很快，如果一有新的硬件出现，就需要重新编译内核就会很麻烦，因此将部分驱动编译成内核模块，在内核需要的时候加载，这样就既可以保证新硬件的使用，也可以保证内核不会一出现新硬件就重新编译

对于硬件驱动程序存在以下两种处理方式
1. 编译到内核
2. 编译成内核模块，内核需要时候进行加载
对于内核来说越简单越好，一些不常用的驱动可以独立成内核模块

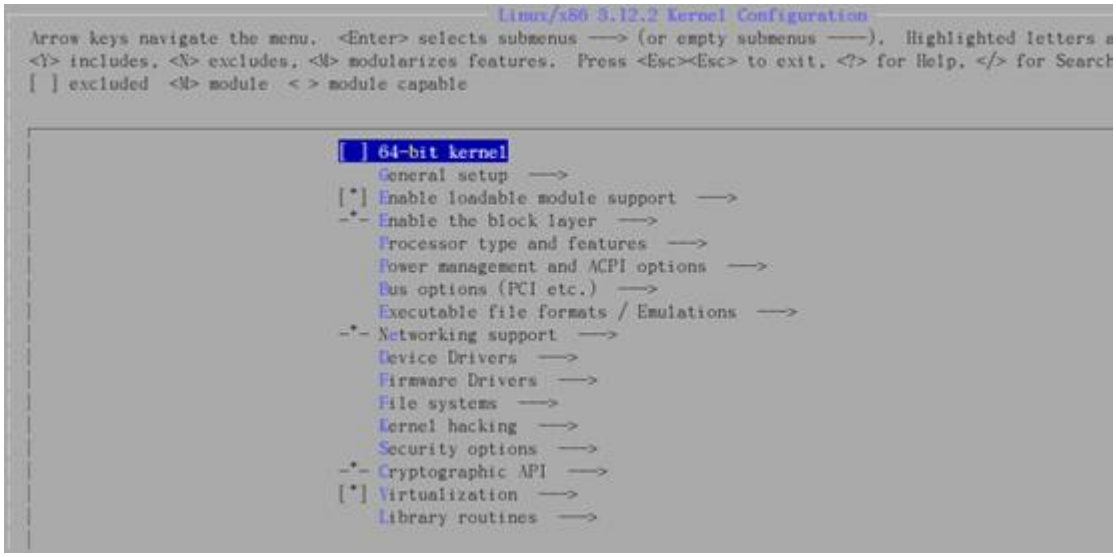
内核的编译

1. 解压缩内核文件

   [root@localhost shared]# tar -jxvf linux-3.12.2.tar.bz2-C /usr/src/kernels/

2. 打开内核配置菜单，开始配置内核

   [root@localhost linux-3.12.2]# make menuconfig

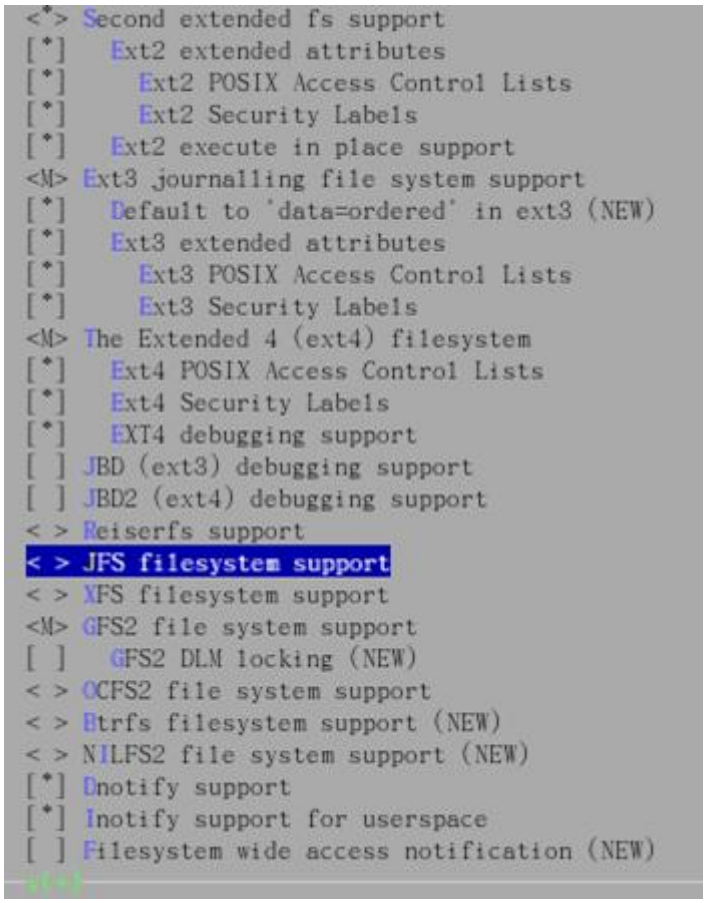


配置项主要分为3中选择类型

<>:不参加编译

<*>参加编译

以内核模块形式参加编译



1. 配置完毕后，会生成一个.config 文件保存刚才的配置信息

```
[root@localhost linux-3.12.2]# ll -a
-rw-r--r--  1 root root 107133 04-03 15:08 .config
```

1. 清理上一次缓冲后开始编译内核

```
[root@localhost linux-3.12.2]# make clean
[root@localhost linux-3.12.2]# make bzImage
Setup is 15704 bytes (padded to 15872 bytes).
System is 2890 kB
CRC 4dc9f3fc
Kernel: arch/x86/boot/bzImage is ready  (#1)
[root@localhost linux-3.12.2]# ll  ./arch/x86/boot/bzImage
-rw-r--r-- 1 root root 2974816 04-03 16:15 ./arch/x86/boot/bzImage
```

5.编译在第2步选择的内核模块

```
[root@localhost linux-3.12.2]# make modules
[root@localhost ~]# ll /lib/modules/
drwxr-xr-x 3 root root 4096 04-03 12:57 2.6.30.3
=>编译完成内核模块但未安装，所以在/lib/modules/下并未产生任何内核模块
```

6.安装内核模块，在/lib/modules/会产生对应版本的内核模块库

```
[root@localhost ~]# make modules_install
[root@localhost linux-3.12.2]# ll /lib/modules/
drwxr-xr-x 3 root root 4096 04-03 16:54 3.12.2
=>到处为止内核与内核模块已经编译安装完毕
```

7.利用 GRUB 做一个多重引导，新的引导加载刚编好的内核

```
[root@localhost ~]#
cp /usr/src/kernels/linux-3.12.2/arch/x86/boot/bzImage /boot/vmlinuz-3.12.2
```

8.创建虚拟文件系统

```
[root@localhost ~]# mkinitrd -v /boot/initrd-3.12.2.img 3.12.2
[root@localhost ~]# ll /boot/initrd-3.12.2.img /boot/vmlinuz-3.12.2
-rw------- 1 root root 2752740 04-04 08:53 /boot/initrd-3.12.2.img
-rw-r--r-- 1 root root 2974816 04-04 08:52 /boot/vmlinuz-3.12.2
```

1. 将内核配置信息保存以便下回使用

```
[root@localhost kernels]# cp /usr/src/kernels/linux-3.12.2/.config /boot/config-3.12.2
```

1. 编辑 menu.lst 添加新的指向新内核的引导

```
[root@localhost ~]# vim /boot/grub/menu.lst
title CentOS (3.12.2)
        root (hd0,0)
        kernel /vmlinuz-3.12.2 ro root=LABEL=/ rhgb quiet rgb=0x317
        initrd /initrd-3.12.2.img
```

11.重启之后查看内核版本

```
[root@localhost ~]# uname -r
3.12.2.3
=>已经是最新的内核了
```

总结起来编译内核主要分为这么几步

1. **配置内核**make menuconfig
2. **清除缓存**make clean
3. **编译内核**make bzImage
4. **编译内核模块**make modules
5. **安装内核模块**make modules_install

独立内核模块的编译

在内核编译的时候我们可以通过内核菜单选择内核模块进行编译。但如果内核已经编译完成，此时有新的硬件产生，但是内核菜单并没有此新硬件对应的驱动，那怎么编译其对应的内核模块，即使内核菜单存在此驱动，难道还需要重新编译内核？这就需要进行独立内核模块的编译来解决了。

简单的说，独立内核模块编译，就是单独编译一个内核模块，并将内核模块添加到内核管理器。这样内核就可以使用新增加内核模块而不需要重新编译了

```
[root@bogon shared]# tar -jxvf r8168-8.038.00.tar.bz2
=>解压缩内核模块
[root@bogon shared]# ll
drwxrwxrwx 1 root root       0 01-08 16:56 r8168-8.038.00
-rwxrwxrwx 1 root root   74460 04-08 21:07 r8168-8.038.00.tar.bz2

[root@localhost r8168-8.038.00]# make clean
[root@localhost r8168-8.038.00]# make modules
=>编译内核模块
[root@localhost r8168-8.038.00]# make install
=>安装内核模块
make -C src/ install
make[1]: Entering directory `/mnt/hgfs/shared/r8168-8.038.00/src'
make -C /lib/modules/2.6.18-371.el5/build SUBDIRS=/mnt/hgfs/shared/r8168-8.038.00/src INSTALL_MOD_DIR=kernel/
make[2]: Entering directory `/usr/src/kernels/2.6.18-371.el5-i686'
  INSTALL /mnt/hgfs/shared/r8168-8.038.00/src/r8168.ko
  DEPMOD  2.6.18-371.el5
make[2]: Leaving directory `/usr/src/kernels/2.6.18-371.el5-i686'
make[1]: Leaving directory `/mnt/hgfs/shared/r8168-8.038.00/src'

[root@localhost ~]# ll /lib/modules/2.6.18-371.el5/kernel/drivers/net/r8168.ko
-rwxr-xr-x 1 root root 1083997 04-09 14:44 /lib/modules/2.6.18-371.el5/kernel/drivers/net/r8168.ko
=>内核模块已经安装成功
[root@localhost ~]# depmod  -a
=>建立内核模块依赖关系
 [root@localhost ~]# ll /lib/modules/2.6.18-371.el5/modules.dep
-rw-r--r-- 1 root root 228487 04-09 14:54 /lib/modules/2.6.18-371.el5/modules.dep
=> modules.dep 已经被修改
```

总结起来编译独立内核模块主要分为这么几步

1. **清除缓存** make clean

2. **编译内核模块** make modules

3. 安装内核模块 make install

4. 建立依赖关系 depmod -a

说明：上面只是将内核模块编译与安装，还未将内核模块加载到内核，因此还需使用 insmod 进行内核模块的加载

内核模块管理

内核模块依赖性

内核模块存放在/lib/modules/$(uname －r)/kernel 当中，其中内核模块一直是存在依赖性的。这些依赖关系记录在/lib/modules/$(uname －r)/modules.dep 当我们增加会减少内核模块时就要修改内核模块之间的依赖关系。依赖关系的修改可以通过 depmod 命令来进行

语法：depmod
depmod [-Anea]
选项与参数：
不加参数：分析目前所有内核，模块并重写写入
-A：只分析比 modules.dep 记录还新的内核模块。才会更新
-n: 不写入 modules.dep 只输出到聘雇
-e: 显示目前已经加载的不可执行的模块名称
-a:分析所有可用模块

内核模块的查看

查看所有内核模块
lsmod
查看指定名称的内核模块
modinfo 模块名称

[root@bogon kernels]# lsmod

```
Module                  Size  Used by
ppdev                  12613  0
autofs4                28997  3
hidp                   22977  2
=> Used by 代表此内核模块被其他内核模块所使用

[root@localhost ~]# modinfo r8168
filename:       /lib/modules/2.6.18-371.el5/kernel/drivers/net/r8168.ko
version:        8.038.00-NAPI
license:        GPL
description:    RealTek RTL-8168 Gigabit Ethernet driver
```

```
…….
depends:
vermagic:       2.6.18-371.el5 SMP mod_unload 686 REGPARM 4KSTACKS gcc-4.1
```

内核模块的加载与删除

内核模块加载

Insmod 内核模块绝对路径

```
[root@localhost 2.6.18-371.el5-i686]#
 insmod /lib/modules/2.6.18-371.el5/kernel/drivers/net/r8168.ko
[root@localhost 2.6.18-371.el5-i686]# lsmod|grep r8168
r8168                 248428  0
```

内核模块删除

rmmod 内核模块绝对路径

```
[root@localhost ~]# rmmod /lib/modules/2.6.18-371.el5/kernel/drivers/net/r8168.ko
[root@localhost ~]# lsmod|grep r8168
```

modprobe 内核模块处理

使用上面的方法可以管理内核模块，但是需要指定内核模块的完整路径，同时如果要删除某个内核模块时如果存在依赖关系，则无法删除成功。使用modprobe 可以避免此问题

Modprobe 去 modules.dep 中寻找内核模块因此不需要指定绝对路径（前提我们已经使用的 depmod 将依赖关系写入该文件）

语法：modprobe[-lfr] 内核模块名称

选项与参数：

-l:列出所有内核模块

-f:强制加载内核模块

-r:删除内核模块

举例：

```
[root@localhost ~]# modprobe vfat =>加载内核模块
[root@localhost ~]# lsmod |grep vfat
vfat                  15937  0
[root@localhost ~]# modprobe -r vfat =>删除内核模块
```

35

系统调用：进程控制

fork 系统调用

函数作用：创建一个子进程

形式：pid_tfork(void);

pid_t vfork(void);

说明：使用 vfork 创子进程时，不会进程父进程的上下文

返回值：[返回值=-1]子进程创建失败

[返回值=0]子进程创建成功

[返回值>0]对父进程返回子进程 PID

```
#include <stdio.h>

#include <sys/stat.h>

#include <unistd.h>

int main() {
    pid_t id = fork();
    if (id < 0) {
        perror("子进程创建失败！");
    } else {
        if (id == 0) {
            printf("子进程工作:PID=%d,PPID=%d\n", getpid(), getppid());
        }else
        {
            printf("父进程工作:PID=%d,PPID=%d，子进程PID=%d\n", getpid(), getppid(),id);
            sleep(5)
        }
    }
}
```

控制台输出

父进程工作:PID=3173,PPID=2432，子进程 PID=3176

子进程工作:PID=3176,PPID=3173

exit 系统调用

函数作用：终止发出调用的进程

形式：voidexit(int status);

说明

1. exit 返回信息可由 wait 系统函数获得

2. 如果父进程先退出子进程的关系被转到 init 进程下

```
#include <stdio.h>

#include <sys/stat.h>

#include <unistd.h>

#include <stdlib.h>

int main() {
    pid_t id = fork();
    if (id < 0) {
        perror("子进程创建失败！");
    } else {
        if (id == 0) {
            printf("子进程工作:PID=%d,PPID=%d\n", getpid(), getppid());
            sleep(20);
            printf("此时子进程:PID=%d,PPID=%d\n", getpid(), getppid());
        }else
        {
            printf("父进程工作:PID=%d,PPID=%d，子进程PID=%d\n", getpid(), getppid(),id);
            sleep(5);
            exit(3);
        }
    }
    return 0;
}
```

控制台输出

父进程工作:PID=3068,PPID=2432，子进程 PID=3071

子进程工作:PID=3071,PPID=3068

此时子进程:PID=3071,PPID=1

wait 系统调用

函数作用：父进程与子进程同步，父进程调用后。进入睡眠状态，直到子进程结束或者父进程在被其他进程终止，

形式：pid_twait(int *status)

pid_t waitpid(pid_t pid ,int *status,int option)

参数：status è exit 是设置的代码

pid è 进程号

option: WNOHANG|WUNTRACED

WNOHANG:,即使没有子进程退出,它也会立即返回,不会像 wait 那样永远等下去.

WUNTRACED:子进程进入暂停则马上返回,但结束状态不予以理会.

返回值：如果成功等待子进程结束，则返回子进程 PID。后者为-1

用来检查子进程返回状态的宏

WIFEXITED 这个宏用来指出子进程是否为正常退出的,如果是,它会返回一个非零值.

WEXITSTATUS 当 WIFEXITED 返回非零值时,我们可以用这个宏来提取子进程的返回值

wait 函数使用

```
#include <sys/types.h>

#include <sys/uio.h>

#include <string.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

int main() {
   pid_t cid;
   cid = fork();
```

```
    if (cid < 0) {
        perror("子进程创建失败！");
    } else {
        if (cid == 0) {
            printf("子进程工作\n");
            printf("子进程PID=%d,PPID=%d\n", getpid(),getppid());
            //sleep(20); //1
        } else {
            //wait(NULL);//2
            //sleep(20);//3
            printf("父进程工作\n");
            printf("父进程PID=%d,PPID=%d\n", getpid(),getppid());
        }
    }
    return 0;
}
```

针对上述代码作以下分析：

1. 当子进程退出时，如果父进程没有 wait 进行回收资源，子进程就会一直变为僵尸进程（Z）直到父进程退出

作法：

打开3处注释后执行程序，查看进程状态，如下

[root@localhostDebug]# ps -C Process -o pid,ppid,stat,cmd

PID PPID STAT CMD

12233 11563S /root/workspace/Process/Debug/Process

12238 12233Z [Process]

=>可以看到子进程此时的状态时 Z（僵尸进程）

控制台输出如下

子进程工作

子进程 PID=12238,PPID=12233

（20S后……）

父进程工作

父进程 PID=12233,PPID=11563

2. 使用 wait 进行进程同步，父进程直到子进程退出，wait 才会结束等待

作法：

打开1，2处注释后执行程序，查看进程状态，如下

[root@ Debug8$] ps -C Process -o pid,ppid,stat,cmd

PID PPID STAT CMD

3425 2432 S /root/workspace/Process/Debug/Process

3430 3425 S /root/workspace/Process/Debug/Process

=>父进程与子进程都处于 sleep 状态

控制台输出如下

子进程工作

子进程 PID=3430,PPID=3425

（20S后…..）

父进程工作

父进程 PID=3425,PPID=2432

3. 使用 wait 进行进程同步，子进程退出后，父进程结束 wait 等待，同时清空子进程信息，此时子进程不再是僵尸进程

作法：

打开2，3处注释后执行程序，查看进程状态，如下

[root@localhostDebug]# ps -C Process -o pid,ppid,stat,cmd

PID PPID STAT CMD

1250611563 S /root/workspace/Process/Debug/****Process

=>可以看到此时只有父进程信息

控制台输出如下

子进程工作

子进程 PID=12511,PPID=12506

（20S后…..）

父进程工作

父进程 PID=12506,PPID=11563

WEXITSTATUS 与 WIFEXITED 宏的使用

```
#include <sys/types.h>

#include <sys/uio.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/wait.h>
```

```
#include <stdio.h>

#include <stdlib.h>

int main() {
    pid_t cid;
    int pr, status;
    cid = fork();
    if (cid < 0) {
        perror("子进程创建失败！");
    } else {
        if (cid == 0) {
            printf("子进程工作 PID=%d,父进程 PID=%d\n", getpid(),getppid());
            sleep(20);
            exit(3);
        } else {
            pr = wait(&status);
            if (WIFEXITED(status)) {
                printf("父进程工作 PID=%d\n", getpid());
                printf("WAIT 返回值=%d\n", pr);
                printf("子进程正常退出 PID=%d\n", getpid());
                printf("WIFEXITED(status)=%d\n", WIFEXITED(status));
                printf("WEXITSTATUS(status)=%d\n", WEXITSTATUS(status));
            } else {
                printf("子进程异常退出 PID=%d,信号=%d\n", getpid(), status);
                printf("WAIT 返回值=%d\n", pr);
            }
        }
    }
    return 0;
}
```

基于上面代码做出分析：

1. 子进程正常退出

控制台输出信息如下：

子进程工作 PID=12070,父进程 PID=12069

（20S后…..）

父进程工作 PID=12069

WAIT 返回值=12070

子进程正常退出 PID=12069

WIFEXITED(status)=1

WEXITSTATUS(status)=3

2. 子进程异常退出

作法：

运行程序，在子进程 SLEEP 期间，杀死子进程

[root@localhost Debug]# kill -9 11990

控制台台输出如下

子进程工作 PID=11990,父进程 PID=11985

（ kill -9 PID 杀死子进程 ）

子进程异常退出 PID=11985,信号=9

可以看出子进程正常退出时，status 返回值是 exit 的退出值，子进程异常退出时 status 返回值信号值

waitpid 函数使用

waitpid 的参数说明

参数 pid 的值有以下几种类型:

pid>0时,只等待进程 ID 等于 pid 的子进程,不管其它已经有多少子进程运行结束退出了,只要指定的子进程还没有结束,waitpid 就会一直等下去.

pid=-1时,等待任何一个子进程退出,没有任何限制,此时 waitpid 和 wait 的作用一模一样.

pid=0时,等待同一个进程组中的任何子进程,如果子进程已经加入了别的进程组，waitpid 不会对它做任何理睬.

pid<-1时,等待一个指定进程组中的任何子进程,这个进程组的 ID 等于 pid 的绝对值.

参数 options 的值有以下几种类型:

如果使用了 WNOHANG 参数,即使没有子进程退出,它也会立即返回,不会像 wait 那样永远等下去.

如果使用了 WUNTRACED 参数,则子进程进入暂停则马上返回,但结束状态不予以理会.

如果我们不想使用它们,也可以把 options 设为0,如:ret=waitpid(-1,NULL,0);

WNOHANG 使用

```
#include <sys/types.h>

#include <sys/uio.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/wait.h>
```

```
#include <stdio.h>

#include <stdlib.h>

int main() {
    pid_t cid;
    int pr, status;
    cid = fork();
    if (cid < 0) {
        perror("子进程创建失败！");
    } else {
        if (cid == 0) {
            printf("子进程工作 PID=%d\n", getpid());
            sleep(5);
            exit(3);
        } else {
            do{
                pr = waitpid(0,&status, WNOHANG);
                if(pr==0)
                {
                 printf("没有子进程退出,继续执行..\n");
                 sleep(1);
                }
            }while(pr==0);
            printf("子进程正常退出 PID=%d\n", pr);
        }
    }
    return 0;
}
```

控制台输出：

没有子进程退出,继续执行..

子进程工作 PID=3632

没有子进程退出,继续执行..

没有子进程退出,继续执行..

没有子进程退出,继续执行..

没有子进程退出,继续执行..

子进程正常退出 PID=3632

针对某一进程进行等待

```
#include <sys/types.h>
```

```
#include <unistd.h>

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

int main() {
    pid_t cid;
    int pr, status;
    cid = fork();
    if (cid < 0) {
        perror("子进程创建失败！");
    } else {
        if (cid == 0) {
            printf("子进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
            sleep(20);
            exit(3);
        } else {
            pr = waitpid(cid, &status, 0);
            printf("父进程正常退出 PID=%d\n", pr);
        }
    }
    return 0;
}
```

控制台输出

子进程工作 PID=4257,PPID=4252

父进程正常退出 PID=4257

WUNTRACED 使用

```
#include <sys/types.h>

#include <unistd.h>

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

int main() {
```

```
    pid_t cid;
    int pr, status;
    cid = fork();
    if (cid < 0) {
        perror("子进程创建失败！");
    } else {
        if (cid == 0) {
            printf("子进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
            sleep(30);
            exit(3);
        } else {
            pr = waitpid(cid, &status, WUNTRACED);
            printf("父进程正常退出 PID=%d,status=%d\n", pr,status);
        }
    }
    return 0;
}
```

作法：在子进程 SLEEP 时，通过 SHELL 命令停止子进程

[root@ ~ 6$] kill -STOP PID

控制台输出

子进程工作 PID=4110,PPID=4108

(SLEEP 期间，停止子进程)

父进程正常退出 PID=4110,status=4991

在查看进程状态，发现此时父进程子进程都已经退出

[root@ Debug 13$] ps -C Process -opid,ppid,stat,cmd

PID PPID STAT CMD

exec 系统调用

函数作用：以新进程代替原有进程，但 PID 保持不变

形式:

int execl(const char *path, const char*arg, ...);

int execlp(const char *file, const char*arg, ...);

int execle(const char *path, const char*arg, ..., char * const envp[]);

int execv(const char *path, char *constargv[]);

int execvp(const char *file, char *constargv[]);

int execve(const char *path, char *constargv[], char *const envp[]);

举例：

```
exec1.c
 #include <stdio.h>

 #include <unistd.h>

int main()
{
  printf("这是第一个进程 PID=%d\n",getpid());
  execv("e2",NULL);
  printf("asa");
  return 0;
}
exec2.c
 #include <stdio.h>

 #include <unistd.h>

int main()
{
  printf("这是第二个进程 PID=%d\n",getpid());
}
```

运行结果：

[root@ Process 9$] gcc -o e1 exec1.c

[root@ Process 10$] gcc -o e2 exec2.c

[root@ Process 11$] ./e1

这是第一个进程 PID=3051
这是第二个进程 PID=3051

36

匿名管道通讯

管道是 Linux 支持的最初 Unix IPC 形式之一，具有以下特点：

1.管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；

2.只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）；

什么是管道

管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在与内存中。

数据的读出和写入

一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

管道的创建

```
#include int pipe(int fd[2])
```

管道两端可分别用描述字 fd[0]以及 fd[1]来描述，需要注意的是，管道的两端是固定了任务的。即一端只能用于读，由描述字 fd[0]表示，称其为管道读端；另一端则只能用于写，由描述字 fd[1]来表示，

管道的规则

1. 当管道内容长度为0时，读端将处于阻塞状态，等待写端向管道写入内容
2. 当写端数据长度小于缓冲区长度时，数据将以原子性写入缓冲区。

对读进程来说：

3. 当写端被关闭时，所有数据被读出后，read 返回0。
4. 当写端未被关闭时，所有数据被读出后，读端阻塞。

对写进程来说：

5. 当读端关闭时，如写端数据长度大于管道最大长度时，写完管道长度时，产生信号 SIGPIPE 后退出程序。（以存入管道的数据读进程可以读取到）
6. 当读端未被关闭时，如写端数据长度大于管道最大长度时，写完管道长度时，写端将处于阻塞状态

规则分析1

```
#include<unistd.h>

#include<stdio.h>

#include<string.h>

#include<sys/types.h>

#include<stdlib.h>

#include<sys/wait.h>

int main() {
    int fd[2];
    pid_t cid;
    if (pipe(fd) == -1) {
        perror("管道创建失败！");
        exit(1);
    }
    cid = fork();
    switch (cid) {
    case -1:
        perror("子进程创建失败");
```

```
        exit(2);
        break;
    case 0:
        close(fd[1]);
        char message[1000];
        int num = read(fd[0], message, 1000);
        printf("子进程读入的数据是：%s,长度是=%d", message, num);
        close(fd[0]);
        break;
    default:
        close(fd[0]);
        char *writeMsg = "父进程写入的数据！";
        sleep(10);//1
        write(fd[1], writeMsg, strlen(writeMsg));
        close(fd[1]);
        break;
    }
    return 0;
}
```

[root@ Release 18$] ps -C processcomm -opid,ppid,stat,cmd

PID PPID STAT CMD

5973 2488 S /root/workspace/processcomm/Release/processcomm

5976 5973 S /root/workspace/processcomm/Release/processcomm

=>读端由于阻塞中，其所在进程（子进程）处于 sleep 状态

控制台输出

父进程工作 PID=5973,PPID=2488

子进程工作 PID=5976,PPID=5973

子进程读入的数据是：父进程写入的数据！,长度是=27

规则分析2

```
switch (cid) {
    case -1:
        perror("子进程创建失败");
        exit(2);
        break;
    case 0:
        printf("子进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
        break;
    default:
```

```
        printf("父进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
        close(fd[0]);
        const long int writesize=4000;
        char writeMsg[writesize];
        int i;
        for(i=0;i<writesize;i++)
        {
            writeMsg[i]='a';
        }
        int writenum=write(fd[1], writeMsg, strlen(writeMsg));
        printf("父进程写入的数据长度是=%d\n", writenum);
        close(fd[1]);
        wait(NULL);
        break;
    }
```

控制台输出

父进程工作 PID=7072,PPID=2488

父进程写入的数据长度是=4001

子进程工作 PID=7077,PPID=7072

规则分析3

```
switch (cid) {
    case -1:
        perror("子进程创建失败");
        exit(2);
        break;
    case 0:
        printf("子进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
        close(fd[1]);
        char message[40001];
        int num = read(fd[0], message, 4001);
        printf("子进程读入的数据长度是=%d\n", num);
        num = read(fd[0], message, 4000);
        printf("子进程再次读入的数据长度是=%d", num);
        close(fd[0]);
        break;
    default:
        printf("父进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
        close(fd[0]);
        const long int writesize = 4000;
        char writeMsg[writesize];
```

```
        int i;
        for (i = 0; i < writesize; i++) {
            writeMsg[i] = 'a';
        }
        int writenum = write(fd[1], writeMsg, strlen(writeMsg));
        printf("父进程写入的数据长度是=%d\n", writenum);
        close(fd[1]);
    //  wait(NULL);
        break;
    }
```

[root@ Release30$] ps -C processcomm -o pid,ppid,stat,cmd

PID PPID STAT CMD

=>读写进程都已退出

控制台输出

父进程工作 PID=8004,PPID=2488

父进程写入的数据长度是=4001

子进程工作 PID=8009,PPID=1

子进程读入的数据长度是=4001

子进程再次读入的数据长度是=0

规则分析4

```
switch (cid) {
    case -1:
        perror("子进程创建失败");
        exit(2);
        break;
    case 0:
        printf("子进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
        char message[40001];
        int num = read(fd[0], message, 4001);
        printf("子进程读入的数据长度是=%d", num);
        num = read(fd[0], message, 4000);
        printf("子进程再次读入的数据长度是=%d", num);
        break;
    default:
        printf("父进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
        close(fd[0]);
        const long int writesize = 4000;
        char writeMsg[writesize];
```

```
        int i;
        for (i = 0; i < writesize; i++) {
            writeMsg[i] = 'a';
        }
        int writenum = write(fd[1], writeMsg, strlen(writeMsg));
        printf("父进程写入的数据长度是=%d\n", writenum);
        close(fd[1]);
        break;
    }
```

[root@ Release29$] ps -C processcomm -o pid,ppid,stat,cmd

PID PPID STAT CMD

7916 1 S /root/workspace/processcomm/Release/processcomm

=>读进程阻塞

控制台输出：

父进程工作 PID=7914,PPID=2488

父进程写入的数据长度是=4001

子进程工作 PID=7916,PPID=1

规则分析5

```
switch (cid) {
    case -1:
        perror("子进程创建失败");
        exit(2);
        break;
    case 0:
        printf("子进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
            close(fd[1]);
        char message[65535];
        int num = read(fd[0], message, 65535);
        printf("子进程读入的数据长度是=%d", num);
    close(fd[0]);
        break;
    default:
        printf("父进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
        close(fd[0]);
        const long int writesize = 80000;
        char writeMsg[writesize];
        int i;
        for (i = 0; i < writesize; i++) {
```

```
        writeMsg[i] = 'a';
    }
    int writenum = write(fd[1], writeMsg, strlen(writeMsg));
    printf("父进程写入的数据长度是=%d\n", writenum);
    close(fd[1]);
    wait(NULL);
    break;
}
```

[root@ Release25$] ps -C processcomm -o pid,ppid,stat,cmd

PID PPID STAT CMD

=>所有进程都以退出

控制台输出

父进程工作 PID=7776,PPID=2488

子进程工作 PID=7778,PPID=7776

子进程读入的数据长度是=65535

规则分析6

switch (cid) { case -1: perror("子进程创建失败"); exit(2); break; case 0: printf("子进程工作 PID=%d,PPID=%d\n", getpid(), getppid()); break; default: printf("父进程工作 PID=%d,PPID=%d\n", getpid(), getppid()); const long int writesize=80000; char writeMsg[writesize]; int i; for(i=0;i<writesize;i++) { writeMsg[i]='a'; } int writenum=write(fd[1], writeMsg, strlen(writeMsg)); printf("父进程写入的数据长度是=%d\n", writenum); wait(NULL); break; } 父进程工作 PID=7309,PPID=2488

子进程工作 PID=7314,PPID=7309

[root@ Release24$] ps -C processcomm -o pid,ppid,stat,cmd

PID PPID STAT CMD

7309 2488 S /root/workspace/processcomm/Release/processcomm

7314 7309 Z [processcomm]

管道代码举例

1. 当发送信息小于管道最大长度

```
#include<unistd.h>

#include<stdio.h>

#include<string.h>

#include<sys/types.h>

#include<stdlib.h>

#include<sys/wait.h>

int main() {
    int fd[2];
    pid_t cid;
    if (pipe(fd) == -1) {
        perror("管道创建失败！");
        exit(1);
    }
    cid = fork();
    switch (cid) {
    case -1:
        perror("子进程创建失败");
        exit(2);
        break;
    case 0:
        printf("子进程工作PID=%d,PPID=%d\n", getpid(), getppid());
        close(fd[1]);
        char message[1000];
        int num;
        do {
            num = read(fd[0], message, 1000);
            printf("子进程读入的数据长度是=%d\n", num);
        } while (num != 0);
        close(fd[0]);
        break;
    default:
        printf("父进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
```

```
        close(fd[0]);
        const long int writesize = 37;
        char writeMsg[writesize];
        int i;
        for (i = 0; i < writesize-1; i++) {
            writeMsg[i] = 'a';
        }
        writeMsg[writesize-1]='\0';
        int writenum = write(fd[1], writeMsg, strlen(writeMsg)+1);
        printf("父进程写入的数据长度是=%d\n", writenum);
        close(fd[1]);
        break;
    }
    return 0;
}
```

2. 当发送信息大于管道最大长度

此例子主要应该规则6，当发送信息大于管道长度时且写进程在未全部将新数据写入管道中，写进程处于阻塞状态，直到所有数据写入管道

```
int main() {
    int fd[2];
    pid_t cid;
    if (pipe(fd) == -1) {
        perror("管道创建失败！");
        exit(1);
    }
    cid = fork();
    switch (cid) {
    case -1:
        perror("子进程创建失败");
        exit(2);
        break;
    case 0:
        printf("子进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
        close(fd[1]);
        char message[1000];
        int num;
        do {
            num = read(fd[0], message, 1000);
            printf("子进程读入的数据长度是=%d\n", num);
        }while(num!=0);
        close(fd[0]);
```

```
        break;
    default:
        printf("父进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
        const long int writesize = 80000;
        char writeMsg[writesize];
        int i;
        for (i = 0; i < writesize-1; i++) {
            writeMsg[i] = 'a';
        }
        writeMsg[writesize-1]='\0';
        int writenum = write(fd[1], writeMsg, strlen(writeMsg)+1);
        printf("父进程写入的数据长度是=%d\n", writenum);
        close(fd[0]);
        close(fd[1]);
        break;
    }
    return 0;
}
```

3. 写进程多次写入

此例应用规则6，防止多次写入，写入数据长度管道最大长度

```
int main() {
    int fd[2];
    pid_t cid;
    if (pipe(fd) == -1) {
        perror("管道创建失败！");
        exit(1);
    }
    cid = fork();
    switch (cid) {
    case -1:
        perror("子进程创建失败");
        exit(2);
        break;
    case 0:
        printf("子进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
        close(fd[1]);
        char message[1000];
        int num;
        do {
            num = read(fd[0], message, 1000);
            if (num > 0) {
```

```
            printf("子进程读入的数据长度是=%d %s\n", num, message);
        }
    } while (num != 0);
    close(fd[0]);
    break;
default:
    printf("父进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
    const long int writesize = 10;
    char writeMsg[writesize];
    int i;
    for (i = 0; i < writesize - 1; i++) {
        writeMsg[i] = 'a';
    }
    writeMsg[writesize - 1] = '\0';
    int writenum = write(fd[1], writeMsg, strlen(writeMsg));
    printf("父进程写入的数据长度是=%d\n", writenum);
    char *newmsg = "helloworld";
    writenum = write(fd[1], newmsg, strlen(newmsg) + 1);
    printf("父进程再次写入的数据长度是=%d\n", writenum);
    close(fd[0]);
    close(fd[1]);
    break;
}
return 0;
}
```

4. 兄弟间的管道通讯

```
int main() {
    int fd[2];
    pid_t cid, did;
    if (pipe(fd) == -1) {
        perror("管道创建失败！");
        exit(1);
    }
    cid = fork();
    switch (cid) {
    case -1:
        perror("兄进程创建失败");
        exit(2);
        break;
    case 0:
        printf("兄进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
        close(fd[1]);
```

```
        char message[1000];
        int num;
        do {
            num = read(fd[0], message, 1000);
            if (num > 0) {
                printf("兄进程读入的数据长度是=%d,%s\n", num, message);
            }
        } while (num != 0);
        close(fd[0]);
        break;
    default:
        did = fork();
        if (did == 0) {
            printf("弟进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
            const long int writesize = 10;
            char writeMsgs[writesize];
            int i;
            for (i = 0; i < writesize - 1; i++) {
                writeMsgs[i] = 'a';
            }
            writeMsgs[writesize - 1] = '\0';
            int writenum = write(fd[1], writeMsgs, strlen(writeMsgs) + 1);
            printf("弟进程写入的数据长度是=%d\n", writenum);
            close(fd[0]);
            close(fd[1]);
        } else if (did == -1) {
            perror("弟进程创建失败！");
            exit(3);
        }
        break;
    }
    return 0;
}
```

5. 父子双通道管道通讯

```
int main() {
    int fd[2], backfd[2];
    pid_t cid;
    if (pipe(fd) == -1) {
        perror("管道创建失败！");
        exit(1);
    }
    if (pipe(backfd) == -1) {
```

```
    perror("管道创建失败！");
    exit(2);
}
cid = fork();
switch (cid) {
case -1:
    perror("子进程创建失败");
    exit(2);
    break;
case 0:
    printf("子进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
    close(fd[1]);
    char message[10000];
    int num;
    do {
        num = read(fd[0], message, 10000);
        printf("子进程读入的数据长度是=%d\n", num);
    } while (num != 0);
    close(fd[0]);
    close(backfd[0]);
    char *msg1 = "消息返回成功啊！";
    write(backfd[1], msg1, strlen(msg1) + 1);
    close(backfd[1]);
    break;
default:
    printf("父进程工作 PID=%d,PPID=%d\n", getpid(), getppid());
    const long int writesize = 80000;
    char writeMsg[writesize];
    int i;
    for (i = 0; i < writesize - 1; i++) {
        writeMsg[i] = 'a';
    }
    writeMsg[writesize - 1] = '\0';
    int writenum = write(fd[1], writeMsg, strlen(writeMsg) + 1);
    printf("父进程写入的数据长度是=%d\n", writenum);
    close(fd[0]);
    close(fd[1]);
    close(backfd[1]);
    char msg2[1000];
    int num1 = read(backfd[0], msg2, 1000);
    printf("返回消息是：%s", msg2);
    close(backfd[0]);
    break;
}
```

```
    return 0;
}
```

37

有名管道通讯

什么是有名管道

匿名管道应用的一个重大限制是它没有名字，因此，只能用于具有亲缘关系的进程间通信，在有名管道（named pipe 或 FIFO）提出后，该限制得到了克服。FIFO 不同于管道之处在于它提供一个路径名与之关联，以 FIFO 的文件形式存在于文件系统中。这样，即使与 FIFO 的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过 FIFO 相互通信

有名管道创建

int mkfifo(const char * pathname, mode_t mode)

和普通文件创建一样 pathname 为文件名称，mode为权限

有名管道通信规则

第 37 章 管道关闭规则

int close (int __fd);

1. 当最后一个读进程管道关闭时，写进程无论是阻塞还是非阻塞，都会将管道写满（如果能写满）并退出

2. 当最后一个写进程管道关闭时，向管道写入一个结束标识，当读进程从管道读到这个结束标识时，如果是阻塞读进程将结束阻塞返回读入数据个数为0.（对于未阻塞读进程如果管道内没有数据则返回-1，如果读到结束标识则返回读入数据个数为0）

第 37 章 规则分析1

写进程

```
#include<unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

#include<fcntl.h>

#include<limits.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<errno.h>

#define FIFO_NAME "/tmp/my_fifo"

#define BUF_SIZE 80000

intmain(int argc, char *argv[]) {
    unlink(FIFO_NAME);
    int pipe_fd;
    int res;
    char buf[BUF_SIZE];
    memset(buf, 3, BUF_SIZE);
    if (access(FIFO_NAME, F_OK) == -1) {
        res = mkfifo(FIFO_NAME, 0766);
        if (res != 0) {
            fprintf(stderr, "不能创建管道文件 %s\n", FIFO_NAME);
            exit(1);
        }
    }
    printf("进程 PID %d 打开管道 O_WRONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, O_WRONLY);
    if (pipe_fd != -1) {
        res = write(pipe_fd, buf, sizeof(buf));
        printf("写入数据的大小是%d \n", res);
        close(pipe_fd);
        sleep(1000);
    } else
```

```
        exit(1);
    exit(1);
}
```

读进程

```
#include<unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

#include<fcntl.h>

#include<limits.h>

#include<sys/types.h>

#include<sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"

#define BUF_SIZE 20

intmain(int argc, char *argv[]) {
    char buf[BUF_SIZE];
    memset(buf, 0, BUF_SIZE);
    int pipe_fd;
    int res;
    int bytes_read = 0;
    printf("进程 PID %d 打开管道 O_RDONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, O_RDONLY);
    if (pipe_fd != -1) {
        bytes_read = read(pipe_fd, buf, sizeof(buf));
        printf("读入数据的大小是%d \n", bytes_read);
        sleep(10);
        close(pipe_fd);
    } else
        exit(1);
    exit(1);
}
```

控制台信息

读进程：

进程 PID 10930打开管道 O_RDONLY

读入数据的大小是20

写进程：

进程 PID 10918打开管道 O_WRONLY

（10S后输出…..）

写入数据的大小是65536

分析：当读进程执行到 close(pipe_fd);时，写进程一次性将数据写满缓冲区（65536）并退出。

第 37 章 规则分析2

写进程

```
#define FIFO_NAME "/tmp/my_fifo"

#define BUF_SIZE 80000

int main(int argc, char *argv[]) {
    unlink(FIFO_NAME);
    int pipe_fd;
    int res;
    char buf[BUF_SIZE];
    memset(buf, 3, BUF_SIZE);
    if (access(FIFO_NAME, F_OK) == -1) {
        res = mkfifo(FIFO_NAME, 0766);
        if (res != 0) {
            fprintf(stderr, "不能创建管道文件 %s\n", FIFO_NAME);
            exit(1);
        }
    }
    printf("进程 PID %d 打开管道 O_WRONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, O_WRONLY);
    if (pipe_fd != -1) {
        res = write(pipe_fd, buf, sizeof(buf));
        printf("写入数据的大小是%d \n", res);
        sleep(10);
        close(pipe_fd);
    } else
        exit(1);
    exit(1);
}
```

读进程

```
#define FIFO_NAME "/tmp/my_fifo"

#define BUF_SIZE 4000

int main(int argc, char *argv[]) {
    char buf[BUF_SIZE];
    memset(buf, 0, BUF_SIZE);
    int pipe_fd;
    int bytes_read = 0;
    printf("进程 PID %d 打开管道 O_RDONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, O_RDONLY);
    if (pipe_fd != -1) {
        do {
```

```
            bytes_read = read(pipe_fd, buf, sizeof(buf));
            printf("读入数据的大小是%d \n", bytes_read);
        } while (bytes_read != 0);
        close(pipe_fd);
    } else
        exit(1);
    exit(1);
}
```

控制台输出：

读进程

进程 PID 12240打开管道 O_RDONLY

读入数据的大小是4000.

……

(10S后)

读入数据的大小是0

写进程

进程 PID 12227打开管道 O_WRONLY

写入数据的大小是80000

分析：

如果读进程为阻塞的，当写进程关闭管道时，读进程收到写进程发来的结束符，读进程结束阻塞（此时bytes_read =0）

如果读进程为非阻塞的，首先将所有数据读取出来，然后在读进程未收到写进程发来的结束符时，由于管道没有数据读进程不会阻塞且返回-1，因为此例 WHILE 退出条件是 bytes_read =0，因此在未读到结束符之前返回值一直是-1，直到读取到结束符才返回0

第 37 章 管道写端规则

对于设置了阻塞标志的写操作：

1.当要写入的数据量不大于 PIPE_BUF 时，linux 将保证写入的原子性。如果此时管道空闲缓冲区不足以容纳要写入的字节数，则进入睡眠，直到当缓冲区中能够容纳要写入的字节数时，才开始进行一次性写操作。
2.当要写入的数据量大于 PIPE_BUF 时，linux 将不再保证写入的原子性。FIFO 缓冲区一有空闲区域，写进程就会试图向管道写入数据，写操作在写完所有请求写的数据后返回。

对于没有设置阻塞标志的写操作：

3.当要写入的数据量大于 PIPE_BUF 时，linux 将不再保证写入的原子性。在写满所有 FIFO 空闲缓冲区后，写操作返回。

4.当要写入的数据量不大于 PIPE_BUF 时，linux 将保证写入的原子性。如果当前 FIFO 空闲缓冲区能够容纳请求写入的字节数，写完后成功返回；如果当前 FIFO 空闲缓冲区不能够容纳请求写入的字节数，则返回 EAGAIN 错误，提醒以后再写

第 37 章 管道读端规则

对于设置了阻塞标志的写操作：

1.如果有进程写打开 FIFO，且当前 FIFO 内没有数据，将一直阻塞。

对于没有设置阻塞标志的写操作：

2.如果有进程写打开 FIFO，且当前 FIFO 内没有数据。则返回-1，当前 errno 值为 EAGAIN，提醒以后再试。

第 37 章 管道读写规则代码举例

写进程

```
#include<unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

#include<fcntl.h>

#include<limits.h>

#include<sys/types.h>

#include<sys/stat.h>

#include <errno.h>

#define FIFO_NAME "/tmp/my_fifo"

#define BUF_SIZE 88888

int main(int argc,char *argv[])
{
   int pipe_fd;
   int res;
   char buf[BUF_SIZE];
   memset(buf,3,BUF_SIZE);
   if(access(FIFO_NAME,F_OK)==-1)
   {
      res=mkfifo(FIFO_NAME,0766);
      if(res!=0)
      {
         fprintf(stderr,"不能创建管道文件 %s\n",FIFO_NAME);
         exit(1);
      }
   }
   printf("进程 PID %d 打开管道 O_WRONLY\n",getpid());
   pipe_fd=open(FIFO_NAME,O_WRONLY|O_TRUNC);//1
  // pipe_fd=open(FIFO_NAME,O_WRONLY|O_TRUNC|O_NONBLOCK);//2
   if(pipe_fd!=-1)
   {
      res=write(pipe_fd,buf,sizeof(buf));
```

```
        printf("写入数据的长度是%d \n",res);
        close(pipe_fd);
    }
    else
        exit(1);
    exit(1);
}
```

读进程

```
#include<unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

#include<fcntl.h>

#include<limits.h>

#include<sys/types.h>

#include<sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"

#define BUF_SIZE 4000

int main(int argc, char *argv[]) {
    int res;
    if(access(FIFO_NAME,F_OK)==-1)
    {
        res=mkfifo(FIFO_NAME,0766);
        if(res!=0)
        {
            fprintf(stderr,"不能创建管道文件 %s\n",FIFO_NAME);
            exit(1);
        }
    }
    char buf[BUF_SIZE];
    memset(buf, 0, BUF_SIZE);
    int pipe_fd;
 int num=0;
    int bytes_read = 0;
```

```
    printf("进程 PID %d 打开管道 O_RDONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, O_RDONLY);//3
    //pipe_fd = open(FIFO_NAME, O_RDONLY|O_NONBLOCK);//4
    if (pipe_fd != -1) {
        do {
            num++;
            bytes_read = read(pipe_fd, buf, sizeof(buf));
            printf("第%d次读入数据，数据的长度是%d \n",num, bytes_read);
        } while (bytes_read != 0);
        close(pipe_fd);
    } else
        exit(1);
    exit(1);
}
```

上面两段代码分别是管道的读端进程与写端进程。其中有4个注释行。分别代表

1. pipe_fd=open(FIFO_NAME,O_WRONLY|O_TRUNC);//1阻塞写端
2. pipe_fd= open (FIFO_NAME,O_WRONLY|O_TRUNC|O_NONBLOCK);//2非阻塞写端
3. pipe_fd =open(FIFO_NAME, O_RDONLY);//3阻塞读端
4. pipe_fd =open(FIFO_NAME, O_RDONLY|O_NONBLOCK);//4非阻塞读端

可以分以下3种情况分析：

说明：写端与读端不能同时都不阻塞

第 37 章 写端阻塞，读端不阻塞

控制台输出如下：

读端进程：

进程 PID 5919打开管道 O_RDONLY

第1次读入数据，数据的长度是-1

……..

第5次读入数据，数据的长度是4000

……..

第26次读入数据，数据的长度是4000

第27次读入数据，数据的长度是888

第28次读入数据，数据的长度是0

写端进程：

进程 PID 5906打开管道 O_WRONLY

写入数据的长度是88888

分析：读端满足读端规则2，前面由于写进程还未开始写入数据到管道因此返回-1

写端满足写端规则2

第 37 章 写端不阻塞，读端阻塞

执行流程：先执行读端程序，在执行写端

控制台输出如下：

读端进程：

进程 PID 6046打开管道 O_RDONLY

第1次读入数据，数据的长度是4000

……

第16次读入数据，数据的长度是4000

第17次读入数据，数据的长度是1536

第18次读入数据，数据的长度是0

写端进程：

进程 PID 6056打开管道 O_WRONLY

写入数据的长度是65536

分析：读端满足读端规则1，读进程在为读取到管道数据时一直处于等待阻塞状态

写端满足写端规则3，写端写满管道后推出，因此写入数据长度是65535，而不是88888

第 37 章 写端阻塞，读端阻塞

控制台输出如下：

读端进程：

进程 PID 8386打开管道 O_RDONLY

第1次读入数据，数据的长度是4000

……

第22次读入数据，数据的长度是4000

第23次读入数据，数据的长度是888

第24次读入数据，数据的长度是0

写端进程：

进程 PID 8373打开管道 O_WRONLY

写入数据的长度是88888

分析：读端满足读端规则1，读进程在为读取到管道数据时一直处于等待阻塞状态

写端满足写端规则2

38

文件管理相关系统编程

重要文件标识

打开文件标识

O_RDONLY：只读方式打开
O_WRONLY：只写方式打开
O_RDWR：可读写方式打开

打开文件操作副标识

O_CREAT：若路径中文件不存在则创建,使用 Open 函数时需同时指定文件权限
O_EXCL：若与 O_CREAT 连用，检查文件是否已经存在，若不存在则建立文件存在则返回错误，这使创建和测试成为一个原子操作
O_APPEND：读写文件从文件尾部开始移动，所有写入数据都加入文件尾部
O_TRUNC：若文件存在并且可以写入，此标识会将源文件内容清空
O_NONBLOCK:如果打开或创建文件是管道文件，或一个块特殊文件，一个字符特殊文件，该表示代表后续操作非阻塞

文件权限标志

S_IRUSR：用户读权限
S_IWUSR:用户写权限
S_IXUSR：用户执行权限
S_IRWX：用户读写执行权限
S_IRGRP：用户组读权限
S_IWGRP：用户组写权限
S_IXGRP：用户组执行权限
S_IRWXG：用户组读写执行权限
S_IROTH:其他用户读权限
S_IWOTH:其他用户写权限
S_IXOTH:其他用户执行权限
S_IRWXO:其他用户读写执行权限

S_ISUID：SUID 权限
S_ISGID：SGID 权限

文件同步输入标识

O_SYNC:每次 write 都等到物理 I/O 完成才返回，包括文件属性更新 I/O 操作完成
O_DSYNC: 每次 write 都等到物理 I/O 完成才返回，不包括文件属性更新 I/O 操作完成
O_RSYNC:使每一个以文件描述符作为参数的 read 的参数等待，直到任何对文件同一部分进行的写操作都完成

重要函数

文件操作

open

用于打开或创建文件

Int open(文件路径，标识，权限标识)

文件路径：绝对路径与相对路径均可

标识：文件标识与操作副标识以及文件同步标识的结合结合

权限标识：是使用权限标识，也可用数字法标识

返回值：成功返回文件标识符.出错返回-1

creat

用于创建文件

int creat(文件路径, 权限标识)

返回值：成功返回文件标识符.出错返回-1

说明：以只写方式打开文件

close

用于关闭文件,当一个进程终止时，内核会自动关闭它打开的所有文件

int close(int fd)

返回值：成功返回0.出错返回-1

sleek

用于设置文件偏移量

off_tsleek(int fd,off_t offset,int whence)

若 whence= SEEK_SET,设置当前偏移量为距离文件开始处 offset 字节

若 whence= SEEK_CUR,设置当前偏移量为距离文件当前偏移处 offset 字节（ offset 可为正负 ）

若 whence= SEEK_END,设置当前偏移量为文件长度加 offset（ offset 可为正负 ）

返回值：成功返回新的文件偏移量，失败返回-1（ 对于管道文件不能设置偏移文件，因此返回-1 ）

dup|dup2

复制一个现存的文件描述符

Int dup(intfd)

Int dup(intfd1,int fd2)

Fd1为复制的文件描述符源

Fd2为复制的文件描述符目的地

如果 fd2文件为关闭应先关闭

返回值：成功返回新的文件描述符，失败返回-1

举例：

```
#include <unistd.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <stdio.h>

int main() {
    int f = open("output", O_CREAT | O_TRUNC | O_RDWR, 0644);
    if (f == -1) {
        perror("文件创建失败！");
        return 0;
    }
    int newf = dup(f);
    write(f, "往文件里写输入！\n", 25);
    write(newf, "使用新的文件描述符！\n", 31);
    int oldInput = dup(STDOUT_FILENO);
    puts("使用标准输出到控制台");
    dup2(f, STDOUT_FILENO);
    puts("使用标准输出到文件");
    dup2(oldInput, STDOUT_FILENO);
    puts("还原标准输出到控制台");
    return 0;
}
```

控制台输出：

使用标准输出到控制台

还原标准输出到控制台

output 文件内容：

往文件里写输入！

使用新的文件描述符!

使用标准输出到文件

fcnt1

改变已打开文件的文件性质

int fcntl (int fd, int cmd, ...);

主要功能:

1: 复制一个现有描述符：Cmd=F_DUPFD

2：获取/设置文件描述符标注 cmd= F_GETFD 或 F_SETFD

3：获取/设置文件状态标注 cmd=F_GETFL 或 F_SETFL

4：获取/设置异步 I/O 所有权 cmd= F_GETOWN 或 F_SETOWN

5：获取/设置记录锁 cmd= F_GETLK 或 F_SETLK

Fcntl 的文件状态标识

O_RDONLY

O_WRONLY

O_RDWR

O_APPENT

O_NONBLOCK

O_SYNC

O_DSYNC

O_FSYNC

说明:

1：由于 O_RDONLY，O_WRONLY，O_RDWR 只能同时存在一个，因此需要用 O_ACCMODE 取得访问屏蔽位

2：F_SETFL 只能设置 O_APPENT，O_NONBLOCK，O_SYNC，O_DSYNC，O_FSYNC

举例:

```
#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

int main(int argc, char *argv[]) {
   int fd = open("fcnt", O_RDWR | O_APPEND|O_SYNC );
```

```
    printf("文件描述符=%d\n", fd);
    int flag = fcntl(fd, F_GETFL, 0);
    switch (flag & O_ACCMODE) {
    case O_RDWR:
        printf("O_RDWR\n");
        break;
    case O_RDONLY:
        printf("O_RDONLY\n");
        break;
    case O_WRONLY:
        printf("O_WRONLY\n");
        break;
    default:
        printf("default\n");
        break;
    }
    if (flag & O_APPEND) {
        printf("O_APPEND\n");
    }
 #if defined (O_SYNC)

    if (flag & O_SYNC) {
        printf("O_SYNC\n");
    }
 #endif

    if (flag & O_NONBLOCK) {
        printf("O_NONBLOCK\n");
    }
    close(1);
    fcntl(fd,F_DUPFD,1);
    puts("通过标准输出写到文件\n");
    return 0;
}
```

控制台输出

文件描述符=3

O_RDWR

O_APPEND

O_SYNC

sync|fsync|fdatasync

当数据写入文件时，内核通常先将数据复制到一个缓冲区中，如果该缓冲区尚未写满，则不将其排入输出队里，直到其写满或者内核需要使用这块缓冲区做其他使用，这种方式叫做延迟写
好处是，可以减少 IO 操作，但是带来的风险就是系统发生故障时，会造成数据的丢失

总结起来，数据写入文件分为以下3步：
1.写入缓冲区
2.缓冲区数据排入输出队里
3.将缓冲区数据写入磁盘

void **sync** (void)
将缓冲区排入输出队里后返回

int **fsync** (int __fd);
等待数据写入磁盘并且文件属性更新后返回

int **fdatasync** (int __fildes);
等待数据写入磁盘返回

access
测试实际用户是否有相应权限
int **access** (constchar *name, int mode)

mode:
R_OK:测试读权限
W_OK：测试写权限
X_OK：测试执行权限
F_OK：测试文件是否存在

文件链接

文件链接分为2种情况
1.硬链接：不会产生新的 INODE,IBLOCK,只是在原有数据连接数上+1
不能跨文件系统使用
硬链接目录需要 ROOT 权限

2.软链接：产生的 INODE,IBLOCK,新的 IBLOCK 记录链接的内容
可以跨文件系统使用

对于软链接来说，有些函数时直接作用链接文件本身，有些函数则跟随源文件链接到源文件

??	???????	??????
access		√
chdir		√
chmod		√
chown	√	√
creat		√
exec		√
lchown	√	
link		√
lstat	√	
open		√
opendir		√
pathconf		√
readlink	√	
remove	√	
rename	√	

stat		√
truncate		√
unlink	√	

link|unlink

创建文件的硬链接

int **link** (constchar *__from, constchar *__to)

删除一个文件

int **unlink** (constchar *__name)

说明：对于硬链接来说 unlink 只是删除文件链接符，文件实际数据的连接数-1，如果为文件实际数据0则在所有进程关闭对此文件连接时删除

举例

```
#include <unistd.h>

#include <stdio.h>

#include <fcntl.h>

#define BUFFER_SIZE 409600

int printfilestat(int fd, struct stat *buf) {
    int results = fstat(fd, buf);
    if (results == -1) {
        perror("文件属性获取失败");
        return -1;
    }
    printf("文件连接数是=%d\n", buf->st_nlink);
    return 0;
}
int main() {
    int fd = open("linkfile",  O_RDWR, 0777);
    if (fd == -1) {
        perror("文件打开失败");
        return -1;
    }
    struct stat buf;
```

```
    printfilestat(fd, &buf);
    if (link("linkfile", "newlinkfile") == 0) {
        puts("链接文件创建成功");
    }
    printfilestat(fd, &buf);
    if (unlink("newlinkfile") == 0) {
        puts("newlinkfile文件删除成功");
    }
    printfilestat(fd, &buf);
    if (unlink("linkfile") == 0) {
        puts("linkfile文件删除成功");
    }
    printfilestat(fd, &buf);
    sleep(30);
    close(fd);
    puts("关闭文件！");
    sleep(-1);
    return 0;
}
```

控制台输出

文件连接数是=1

链接文件创建成功

文件连接数是=2

newlinkfile 文件删除成功

文件连接数是=1

linkfile 文件删除成功

文件连接数是=0

虽然文件连接数为0 但是文件数据没有被删除，需要等待所有进程都 close 该文件才会被从磁盘删除

tkf@tkf:~/workspace/FileOperator$ ll linkfile ;df ./

-rwxrwxr-x 1 tkf tkf 4096005月9 16:34linkfile*

文件系统1K-blocks已用可用已用% 挂载点

/dev/sda128768380 17972780931121266% /

执行程序

tkf@tkf:~/workspace/FileOperator$ df ./

文件系统1 K-blocks 已用可用已用% 挂载点

/dev/sda128768380 17972784931120866% /

linkfile, newlinkfile 文件符号连接删除，因此可用资源变多了

sleep(30)
tkf@tkf:~/workspace/FileOperator$df ./
文件系统1 K-blocks 已用可用已用% 挂载点
/dev/sda128768380 17972344931164866% /
执行了 close,因此在无进程来接到数据，所以文件数据被释放，可用资源再一次变多了

symlink|readlink

创建一个软链接
int **symlink** (constchar *__from, constchar *__to)
返回值：成功返回0，失败返回-1

打开软链接文件
ssize_t **readlink** (constchar *__restrict __path,
char *__restrict __buf, size_t __len)
返回值：成功返回0，失败返回-1

文件状态和属性

获取文件状态
Int fstat(文件标识符，struct stat *buf)
Int lstat(文件路径，struct stat *buf)
Int stat(文件路径，struct stat *buf)

文件路径：绝对路径与相对路径均可
文件标识符：文件创建或打开时返回的文件标示符
struct stat *buf：文件属性结构体

返回值：成功返回0，失败返回-1

说明：stat 和 lstat 的区别：当文件是一个符号链接时，lstat 返回的是该符号链接本身的信息；而 stat 返回的是该链接指向的文件的信息

stat 结构体成员意义

```
struct stat {
dev_t st_dev; 文件所在设备的 ID
ino_t st_ino; 与该文件关联的 inode
mode_t st_mode;
nlink_t st_nlink; /* 链向此文件的连接数(硬连接)*/
uid_t st_uid; 文件属主的 UID 号
gid_t st_gid; 文件属主的 GID 号
dev_t st_rdev; 设备号，针对设备文件
off_t st_size; 文件大小
blksize_t st_blksize; 系统块的大小(IO 缓冲区适合大小)
blkcnt_t st_blocks; 文件所占块数
time_t st_atime;
time_t st_mtime;
time_t st_ctime;
}
```

st_mode 标志

文件类型标志：

S_IFBLK：文件是一个特殊的块设备

S_IFDIR：文件是一个目录

S_IFCHR：文件是一个特殊的字符设备

S_IFIFO：文件是一个 FIFO 设备

S_IFREG：文件是一个普通文件（REG 即使 regular 啦）

S_IFLNK：文件是一个符号链接

其他模式标志：

S_ISUID: 文件设置了 SUID 位

S_ISGID：文件设置了 SGID 位

S_ISVTX：文件设置了 SBIT 位

用于解释 st_mode 标志的掩码：

S_IFMT：文件类型

S_IRWXU：属主的读/写/执行权限，可以分成 S_IXUSR,S_IRUSR, S_IWUSR

S_IRWXG：属组的读/写/执行权限，可以分成 S_IXGRP,S_IRGRP, S_IWGRP

S_IRWXO：其他用户的读/写/执行权限，可以分为 S_IXOTH,S_IROTH, S_IWOTH

确定文件类型

S_ISBLK：测试是否是特殊的块设备文件

S_ISCHR：测试是否是特殊的字符设备文件

S_ISDIR：测试是否是目录（我估计 find .-type d 的源代码实现中就用到了这个宏）

S_ISFIFO：测试是否是 FIFO 设备

S_ISREG：测试是否是普通文件

S_ISLNK：测试是否是符号链接

S_ISSOCK：测试是否是 socket

文件权限

umask

设置文件权限屏蔽字

mode_t **umask** (mode_t __mask)

chmod|lchmod|fchmod

设置文件权限

int **chmod** (constchar * file, __mode_tmode)

int **lchmod** (constchar * file, __mode_tmode)

int **fchmod** (int fd, mode_tmode)

chown|fchown|lchown

设置文件所属用户及用户组

int **chown** (constchar *__file, __uid_t __owner, __gid_t__group)

int **fchown** (int __fd, __uid_t __owner, __gid_t __group) _

int **lchown** (constchar *__file, __uid_t __owner, __gid_t __group)

目录操作

创建目录

Int mkdir(路径，权限)

路径:绝对路径相对路径均可

权限:以数字形式表示的权限

返回值：成功返回0，失败返回-1

进入|获取工作目录

进入工作目录

Int chdir(路径)

路径:绝对路径相对路径均可

返回值：成功返回0，失败返回-1

Int fchdir(intfiledes)

返回值：成功返回0，失败返回-1

获取工作目录

char *getcwd (char *__buf, size_t __size)

返回值：当前工作目录

子目录流操作

打开目录，获得子目录流指针

DIR*opendir(char *name)

读取子目录

structdirent* readdir((DIR *dirp)

返回子目录流里的当前位置

longint telldir(DIR* drip)

设置子目录流的当前数据项指针

voidseekdir(DIR* drip,long int loc)

关闭子目录流

DIR*opendir(DIR* drip)

删除目录或文件

删除目录：int rmdir(路径)

删除文件：int unlink(路径)

返回值：成功返回1，失败返回−1

第 38 章 综合例子

```
#include<fcntl.h>

#include<sys/stat.h>

#include<unistd.h>

#include<stdio.h>

#include<malloc.h>

#include<string.h>

#include<dirent.h>

#include<stdlib.h>

typedefenum {
    false = 0, true = 1
} bool;
voidprintFileInfo(struct stat* buf) {
    bool userall = false;
    printf("文件权限是:%o. 详细信息如下：\n", (buf->st_mode& 0x0fff));
    if (buf->st_mode& S_IRWXU) {
        userall = true;
        printf("所有者拥有读写执行权限\n");
    }
    if (buf->st_mode& S_IRWXG) {
        printf("用户组拥有读写执行权限\n");
    }
    if (buf->st_mode& S_IRWXO) {
        printf("其他人拥有读写执行权限\n");
    }
    if (userall) {
        if (buf->st_mode& S_IRUSR) {
            printf("所有者拥有读权限\n");
        }
        if (buf->st_mode& S_IWUSR) {
            printf("所有者拥有写权限\n");
        }
    }
    if (buf->st_mode& S_IFREG) {
        printf("文件是一个普通文件\n");
    }
    if (buf->st_mode& S_ISUID) {
        printf("文件设置了 SUID 权限\n");
```

```
    }
    if (buf->st_mode& S_ISGID) {
        printf("文件设置了 GUID 权限\n");
    }
    printf("UID=%d\n", buf->st_uid);
    printf("GID=%d\n", buf->st_gid);
    printf("占用block=%ld\n", buf->st_blocks);
    printf("block大小=%ld\n", buf->st_blksize);
    printf("最后访问时间=%ld\n", buf->st_atim.tv_sec);
    printf("最后状态更新时间=%ld\n", buf->st_ctim.tv_sec);
    printf("最后修改时间=%ld\n", buf->st_mtim.tv_sec);
}
intOpenFile(constchar *fpath) {
    unlink(fpath);
    int f = open(fpath, O_RDWR);
    if (f == -1) {
        f = creat(fpath, S_IWUSR | S_IRUSR);
        if (f != -1) {
            printf("创建一个文件\n");
        } else {
            printf("无法创建文件\n");
            return -1;
        }
    } else {
        printf("文件打开成功\n");
    }
    return f;
}
voidscan_dir(constchar* dir, int depth) {
    DIR *dp;
    struct dirent* entry;
    if ((dp = opendir(dir)) == NULL) {
        printf("无法打开目录:%s\n", dir);
        return;
    }
    struct stat statbuf;
    chdir(dir);
    while ((entry = readdir(dp)) != NULL) {
        constchar* name = entry->d_name;
        lstat(name, &statbuf);
        if (S_IFDIR & statbuf.st_mode) {
            if (strcmp(".", entry->d_name) == 0
                    || strcmp("..", entry->d_name) == 0) {
                continue;
            }
```

```
            printf("%*s%s:%o\n", depth, "", entry->d_name,
                    (statbuf.st_mode& 0x0fff));
            scan_dir(entry->d_name, depth + 4);
        } else {
            printf("%*s%s:%o\n", depth, "", entry->d_name,
                    (statbuf.st_mode& 0x0fff));
        }
    }
    chdir("..");
    closedir(dp);
}
intmain() {
    constchar *fpath = "test";
    int f = OpenFile(fpath);
    struct stat *buf = malloc(sizeof(struct stat));
    fstat(f, buf);
    printf("==================================================\n");
    printFileInfo(buf);
    printf("==================================================\n");
    close(f);
    sleep(1);
    chmod("test", 7777);
    printf("更改文件权限为7777\n");
    stat("test", buf);
    printf("==================================================\n");
    printFileInfo(buf);
    printf("==================================================\n");
    free(buf);
    printf("==================扫描文件夹==========================\n");
    scan_dir("/home", 0);
    umask(0011);
    mkdir("/tmp/mydir", 0777);
    creat("/tmp/mydir/myfile", 0777);
    printf("==================扫描文件夹==========================\n");
    scan_dir("/tmp/mydir", 0);
    chdir("/tmp");
    unlink("mydir/myfile");
    rmdir("mydir");
    return 0;
}
```

执行结果

> 创建一个文件

文件权限是:600. 详细信息如下:
所有者拥有读写执行权限
所有者拥有读权限
所有者拥有写权限
文件是一个普通文件
UID=0
GID=0
占用 block=8
block 大小=4096
最后访问时间=1397539372
最后状态更新时间=1397539372
最后修改时间=1397539372

> 更改文件权限为7777

文件权限是:7141. 详细信息如下:
所有者拥有读写执行权限
用户组拥有读写执行权限
其他人拥有读写执行权限
文件是一个普通文件
文件设置了 SUID 权限
文件设置了 GUID 权限
UID=0
GID=0
占用 block=8
block 大小=4096
最后访问时间=1397539372
最后状态更新时间=1397539373
最后修改时间=1397539372

> 扫描文件夹

.bashrc:644
.bash_logout:644
.mozilla:755
extensions:755
plugins:755

.nautilus:755
metafiles:700
目录创建成功
文件创建成功

> 扫描文件夹

myfile:766
文件删除成功
目录删除成功