Linux 基础千锤百炼 v3

作者博客：骏马金龙(https://www.junmajinlong.com/)

说明：

1.文章完全原创，为本人复习总结而来。

2.命令用法基本整理自 man 文档或 info 文档翻译。

3.文章内含很多原理、机制和"新大陆"，内容也比较细，不适合没有耐心的纯初学者，更适合回炉深造。

4.为了代码排版，页面较大，请谅解。

5.本人博客区分享了非常多的系列文章，都很完整，欢迎各位光临。目前包括的系列有：

(1).**Linux 基础 & Shell 系列文章总目录：** http://www.cnblogs.com/f-ck-need-u/p/7048359.html

(2).**网站架构从 0 开始系列文章总目录：** http://www.cnblogs.com/f-ck-need-u/p/7576137.html

(3).**数据库系列文章总目录：** http://www.cnblogs.com/f-ck-need-u/p/7586194.html

(4).**Python 系列文章总目录：** https://www.cnblogs.com/f-ck-need-u/p/9832640.html

(5).**Golang 系列文章总目录：** https://www.cnblogs.com/f-ck-need-u/p/9832538.html

(6).**Perl 和 Perl 一行式系列文章总目录：** https://www.cnblogs.com/f-ck-need-u/p/9512185.html

6.本人也已录制关于 Shell 的进阶课程，课程链接 Shell 精品进阶教程：理解 Shell 的方方面面。

7.如发现错误、或有疑惑之处 ，欢迎到本人博客区留言或者联系邮箱 mlongshuai@gmail.com。

更新历史：

v1：2018-03-14

v2：2019-02-13

v3：2019-10-28

第1章 文件类基础命令

1.1 关于路径和通配符

Linux 中分绝对路径和相对路径，绝对路径一定是从/开始写的，相对路径不从根开始写，且还可能使用路径符号。

路径符号：

```
.  : (一个点)表示当前目录
.. : (两个点)表示上一层目录
-  : (一个短横线)表示上一次使用的目录，例如从/tmp 直接切换到/etc 下，"-"就表示/tmp
~  : (波浪符号)表示用户的家目录，例如"~account"表示 account 用户的家目录
/dir/和/dir：一般都表示 dir 目录和 dir 目录中的文件。但在有些地方会严格区分是否加尾
             随斜线，此时对于加了尾随斜线的表示此目录中的文件，不加尾随斜线的表示
             该目录本身和此目录中的文件
```

切换路径用 cd 命令；

显示当前所在目录用 pwd 命令。若当前所在目录为链接目录，使用 pwd 显示的将是链接自身，使用-P 选项将定位到链接的原始目录。

```
[root@xuexi ~]# ll ; cd tmp; pwd; pwd -P
total 0
lrwxrwxrwx 1 root root 4 May 30 19:17 tmp -> /tmp
/root/tmp
/tmp
```

获取文件名使用 basename 命令，获取文件所在目录使用 dirname 命令。注意，这两个命令其实不太完善，它不会检查文件或目录是否存在，只要写出来了就会去获取。

```
[root@xuexi tmp]# basename /etc/shadow
shadow
[root@xuexi tmp]# basename /etc/
etc
[root@xuexi tmp]# dirname /etc/shadow
/etc
[root@xuexi tmp]# dirname /etc/    # 对目录使用 dirname 获取的是上级目录
/
[root@xuexi ~]# dirname /kalsldk/kdkskks/djfjdjdjsj   # 获取不存在的目录
/kalsldk/kdkskks
```

bash shell 通配符：

可以使用"*"、"?"、"[]"等的通配符来扩展路径或文件名。例如，"ls *.log"将列出当前路径下所有以".log"字符结尾的文件名(但不包括"."开头的隐藏文件)。

默认情况下，bash 提供的通配符规则比较弱，例如"*"无法匹配文件名开头的".", 无法匹配路径分隔符号(即斜线"/")，但可以通过 set 或 shopt 命令开启额外的通配功能，实现更完善的通配符规则。

例如，默认情况下，想要匹配目录/path 下所有隐藏文件和非隐藏文件，如下：

```
ls  .*  *
```

开启 dotglob 功能，"*"就可以匹配以"."开头的文件：

```
shopt -s dotglob
ls *
```

有时想要递归到目录内部，又想要匹配文件名，例如想要递归找出多层目录/path 下所有的".css"文件，这时可以开启 globstar 功能，使用"两星连珠"(**)就可以匹配匹配路径斜线。

```
shopt -s globstar      # 开启星号匹配模式
ls /path/**/*.css      # 开启后，使用两个星号**就会匹配斜线
```

必须要说明的是，对于非 bash 内置命令，有些可能也提供了自己的通配符匹配方式，它们的通配模式和 shell 提供的可能并不一样。例如 find 的"-name"选项就可以采用自己的通配符，它的星号"*"可以匹配以点开头的隐藏文件，如" find /var/log -name "*.log" "。

1.2 查看目录内容(ls 和 tree)

ls 命令列出目录中的内容，和 dir 命令完全等价。tree 命令按树状结构递归列出目录和子目录中的内容，而 ls 使用-R 选项时才会递归列出。

注意：ls 的结果中是以制表符分隔多个文件的。

1. **ls 命令**

ls 的各个选项说明如下：

```
-l：(long)长格式显示，即显示属性等信息(包括 mtime)。注意：显示的目录大小是节点所占大小。像 win 一样计算目录大小时包括文件大小要用 du -sh
-c：列出 ctime
-u：列出 atime
-d：(direcorty)查看目录本身属性信息，不查看目录里面的东西。不加-d 会查看里面文件的信息
-a：会显示所有文件，包括两个相对路径的文件"."和".."以及以点开头的隐藏文件
-A：会列出绝大多数文件，即忽略两个相对路径的文件"."和".."
-h：(human)人类可读的格式，将字节换成 k,将 K 换成 M，将 M 换成 G
-i：(inode)权限属性的前面加上一堆数字
-p：对目录加上/标识符以作区分
-F：对不同类型的文件加上不同标识符以作区分，对目录加的文件也是/
-t：按修改时间排序内容。不加任何改变顺序的选项时，ls 默认按照字母顺序排序
-r：反转排序
-R：递归显示
-S：按文件大小排序，默认降序排序
--color：显示颜色
-m：使用逗号分隔各文件，当然，只适用于未使用长格式(ls -l)的情况
-1：(数值一)，以换行符分隔文件，当然，和-m 或-l(小写字母)是冲突的
-I pattern：忽略被 pattern 匹配到的文件
```

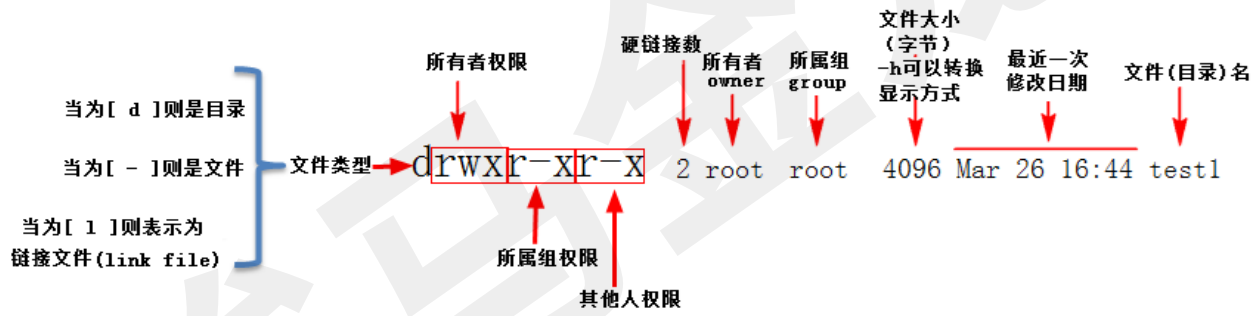
注意，ls 以-h 显示文件大小时，一般显示的都是不带 B 的单位，如 K/M/G，它们的转换比例是 1024，如果显示的都是带了 B 的，如 KB/MB/GB，则它们的转换比例为 1000 而非 1024，一般很少显示带 B 的大小。

不得不说，ls 本身不能显示出文件的全路径名是一大缺陷，但好在 find 可以很简单的就获取到。

以下是使用 `ls -l` 显示文件长格式的属性。

```
[root@xuexi ~]# ll /tmp
drwxr-xr-x  2 root root 4096 Mar 26 16:44 test1
```

可以查出 7 列属性。



2. tree 命令

有可能 tree 命令不存在，需要安装 tree 包才有(安装：`yum -y install tree`)。

tree 命令的选项说明如下：

➢ **匹配选项：**

```
-L：用于指定递归显示的深度，指定的深度必须是大于 0 的整数。
-P：用于显示通配符匹配模式的目录和文件，但是不管是否匹配，目录一定显示。
-I：用于显示除被通配符匹配外的所有目录和文件。
```

➢ **显示选项：**

```
-a：用于显示隐藏文件，默认不显示。
-d：指定只显示目录。
-f：指定显示全路径。
-i：不缩进显示。和-f 一起使用很有用。
-p：用于显示权限位信息。
-h：用于显示大小。
-u：显示 username 或 UID(当没有 username 时只能显示 UID 了)。
-g：显示 groupname 或 GID。
-D：显示文件的最后一次 Mtime。
--inodes：显示 inode 号。
--device：显示文件或目录所属的设备号。
-C：显示颜色。
```

➢ **输出选项：**

```
-o filename：指定将 tree 的结果输出到 filename 文件中。
```

下图是一个较全的输出结果。

```
[root@xuexi tmp]# tree /tmp -L 2  -f  --device --inodes -ugDhp -C
/tmp                [--device
├──[ 921793 2051 drwxr-xr-x root     root     4.0K Oct  3  9:45]  /tmp/1 space
├──[ 913930 2051 drwxr-xr-x root     root     4.0K Sep 27 14:50]  /tmp/a
├──[ 913931 2051 drwxr-xr-x root     root     4.0K Sep 27 14:50]  /tmp/b
├──[ 913997 2051 drwxr-xr-x root     root     4.0K Sep 27 14:50]  /tmp/c
├──[ 921778 2051 -rw-r--r-- root     root     4.6K Oct  2 10:41]  /tmp/csplit.man
├──[ 914012 2051 drwxr-xr-x root     root     4.0K Sep 27 14:50]  /tmp/d
├──[ 130562 2051 lrwxrwxrwx root     root        4 Oct  3 15:55]  /tmp/etc.d -> /etc
├──[ 921789 2051 -rwxr-xr-x root     root      139 Oct  2 21:39]  /tmp/expect.sh
├──[ 921784 2051 -rwxr-xr-x root     root      119 Oct  2 21:02]  /tmp/interactive.sh
├──[ 914053 2051 drwxr-xr-x root     root     8.8M Oct  2 21:54]  /tmp/logdir
│   ├──[ 921786 2051 -rw-r--r-- root     root     200M Oct  2 21:52]  /tmp/logdir/a
│   ├──[ 921790 2051 -rw-r--r-- root     root      50M Oct  2 21:52]  /tmp/logdir/b
│   ├──[ 921791 2051 -rw-r--r-- root     root     150M Oct  2 21:53]  /tmp/logdir/c
│   ├──[ 921792 2051 -rw-r--r-- root     root     200M Oct  2 21:54]  /tmp/logdir/d
│   ├──[ 921773 2051 -rw-r--r-- root     root       47 Oct  1 16:27]  /tmp/logdir/md5file.2
│   ├──[ 921775 2051 -rw-r--r-- root     root      167 Oct  1 16:42]  /tmp/logdir/x.tar.gz
│   └──[ 921634 2051 -rw-r--r-- root     root       45 Oct  1 16:40]  /tmp/logdir/x.txt
├──[ 921772 2051 -rw-r--r-- root     root       40 Oct  1 16:56]  /tmp/md5file.1
└──[ 921776 2051 -rw-r--r-- root     root        0 Oct  1 16:59]  /tmp/md5file.2
```

1.3 文件的时间戳(atime/ctime/mtime/relatime)

文件的时间属性有三种：atime/ctime/mtime。atime 是 access time，即上一次的访问时间；mtime 是 modify time，是文件的修改时间； ctime 是 change time，也是文件的修改时间，只不过这个修改时间计算的 inode 修改时间，也就是元数据修改时间，它由操作系统自身维护，一般来说通过上层工具无法手动修改 ctime。文件还有一个创建时间(create time)，大多数 unix 系统上都认为这是个无用的属性，一般工具无法获取这个时间，但是对于 ext 家族文件系统，通过它的底层调试工具 debugfs 可以获取 create time。

但 mtime 只有修改文件内容才会改变，更准确的说是修改了它的 data block 部分；而 ctime 是修改文件属性时改变的，确切的说是修改了它的元数据部分，例如重命名文件，修改文件所有者，移动文件(移动文件没有改变 datablock，只是改变了其 inode 指针，或文件名)等，当然，修改文件内容也一定会改变 ctime(修改文件内容至少已经修改了 inode 记录上的 mtime，这也是元数据)，也就是说 mtime 的改变一定会引起 ctime 的改变。

对目录而言，考虑目录文件的 data block，可知在目录中创建、删除文件以及其他任意目录内的操作都会改变 mtime，因为目录里的任何东西都是目录的内容；而目录的 ctime，除了目录的 mtime 引起 ctime 改变之外，对目录本身的元数据修改也会改变 ctime。

总结下：

(1)atime 只在文件被打开访问时才改变，若不是打开文件编辑内容，则 ctime 和 mtime 的改变不会引起 atime 的改变;

(2)mtime 的改变一定引起 ctime 的改变，而访问文件时(例如 cat)，atime 不一定会改变，所以 atime"改变"(这个改变是假象，见下文分析)不一定会影响 ctime。(见下面的 relatime 说明)

1.3.1 关于 relatime

atime/ctime/mtime 是 Posix 标准要求操作系统维护的时间戳信息。但是每次将 atime、ctime 和 mtime 写入到硬盘中(这些不会写入缓存，只要修改就是写入磁盘，即使从缓存读取文件内容也如此)效率很低。有多低？下图是写 ctime 消耗的时间，几乎总要花费零点几秒。

```
[root@xuexi perlapp]# ./21.plx
atime:1970-01-01 08:00:00.000000000 +0800
mtime:2018-09-05 17:14:41.000000000 +0800
ctime:2018-09-05 18:14:41.260380405 +0800
[root@xuexi perlapp]# ./21.plx
atime:2018-09-05 18:14:55.000000000 +0800
mtime:2018-09-05 17:14:55.000000000 +0800
ctime:2018-09-05 18:14:55.356698219 +0800
[root@xuexi perlapp]# ./21.plx
atime:1970-01-01 07:59:59.000000000 +0800
mtime:2018-09-05 17:15:02.000000000 +0800
ctime:2018-09-05 18:15:02.265107285 +0800
[root@xuexi perlapp]# ./21.plx
atime:1970-01-01 08:00:01.000000000 +0800
mtime:2018-09-05 17:15:15.000000000 +0800
ctime:2018-09-05 18:15:15.554342659 +0800
[root@xuexi perlapp]# ./21.plx
atime:1970-01-01 08:00:00.000000000 +0800
mtime:2018-09-05 17:15:30.000000000 +0800
ctime:2018-09-05 18:15:30.584492783 +0800
[root@xuexi perlapp]# man touch
```

mtime 要被修改，必然是修改了文件内容，这时候将 mtime 写入到硬盘中是应该的。但是 atime 和 ctime 呢？很多情况下根本用不到 atime 和 ctime，在频繁访问文件的时候，都要修改 atime 和 ctime，这样效率会降低很多很多，所以 mount 有个 noatime 选项来避免这种负面影响。

CentOS6 引入了一个新的 atime 维护机制 relatime：除非两次修改 atime 的时间超过 1 天(默认设置 86400 秒)，或者修改了 mtime，否则访问文件的 inode 不会引起 atime 的改变。换句话说，当 cat 一个文件的时候，它的 atime 可能会改变，但是你稍后再 cat，它不会再改变。

由于 cat 文件的时候 atime 可能不会改变，所以可能也就不会引起 ctime 的改变。

relatime 维护的 atime 是可以控制的，详见 man mount 的 relatime 和 redhat 官方手册：https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/power_management_guide/relatime?tdsourcetag=s_pctim_aiomsg

1.4 文件/目录的创建和删除

1.4.1 创建目录 mkdir

```
mkdir [-mp] 目录名
-m：表示直接设置权限
-p：表示递归创建多层目录，即上层目录不存在时也会直接将其创建出来(parent)
```

```
[root@xuexi ~]# mkdir /tmp/test1                # 在 tmp 目录中创建一个 test1 目录
[root@xuexi ~]# mkdir -m 711 /tmp/test2         # 直接创建 test2 时就赋予权限 711
[root@xuexi ~]# mkdir -p /tmp/test3/test4/test5  # 创建 test5，此时会将不存在的 test3 和 test4 目录也创建好
```

1.4.2 创建文件 touch

```
touch  file_name
```

```
[root@xuexi ~]# touch /tmp/test1/test1.txt
[root@xuexi ~]# touch {1..10}      # 创建文件名为 1-10 的文件
```

多个{}还可以交换扩展。类似(a+b)(c+d)=ac+ad+bc+bd。

```
[root@xuexi ~]# touch {a,b}_{c,d}   # 创建 a_c、a_d、b_c、b_d 四个文件
```

touch 主要是修改文件的时间信息，当 touch 的文件不存在时就自动创建该文件。可以使用 touch -c 来取消创建动作。

touch 可以更改最近一次访问时间(atime)，最近一次修改时间(mtime)，文件属性修改时间(ctime)，这些时间可以通过命令 stat file 来查看。其中 ctime 是文件属性上的更改，即元数据的更改，比如修改权限。

touch -a 修改 atime，-m 修改 mtime，没有修改 ctime 的选项。

-t 选项表示使用"[[CC]YY]MMDDhhmm[.ss]"格式的时间替代当前时间。

```
touch -a -t 201212211212 file      # 将 file 文件的 atime 修改为 2012 年 12 月 21 号 12 点 12 分
```

-d 选项表示使用指定的字符串描述的时间格式替代当前时间，如"3 days ago","next Sunday"等很多种格式。

所以，touch 命令选项说明如下：

```
-c：强制不创建文件
-a：修改文件 access time(atime)
-m：修改文件 modification time(mtime)
-t：使用"[[CC]YY]MMDDhhmm[.ss]"格式的时间替代当前时间
-d：使用字符串描述的时间格式替代当前时间
```

1.4.3 删除文件/目录

```
rm [-rfi] file_name
-r：表示递归删除，删除目录时需要加此参数
-i：询问是否删除(yes/no)
-f：强制删除，不进行询问
```

```
[root@xuexi ~]# rm  -rf /tmp/test2
```

删除空目录时还可以使用 rmdir。

在删除文件之前，一定一定要确定是否真的删除。最好使用 rm -i(默认已经在~/.bashrc 中定义了该别名)，除非在脚本中，否则不要使用-f 选项。已经有非常非常多的人不小心 rm -rf *和 rm -rf /NNNNN 了。例如想删除"rm -rf abc*"，结果习惯性的多敲了一个空格"rm -rf abc *"，完了。

1.5 查看文件类型 file 命令

这是一个简单查看文件类型的命令，查看文件是属于二进制文件还是数据文件还是 ASCII 文件。

```
[root@xuexi tmp]# file /etc/aliases.db
/etc/aliases.db: Berkeley DB (Hash, version 9, native byte-order)  # 数据文件
[root@xuexi tmp]# file ~/.bashrc
/root/.bashrc: ASCII text     # ASCII 文件
[root@xuexi tmp]# file /bin/ls
/bin/ls: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.18, stripped
```

除了基本的查看文件类型的功能外，file 还有一个"-s"选项，是一个超强力的选项，可以查看设备的文件系统类型。像有些分区工具如 parted 在分区时是可以指定文件系统的(虽然不建议这么做)，但在分区后格式化前，一般是比较难查看该分区的文件系统类型的，但使用 file 可以查看到。

```
[root@server1 ~]# file -s /dev/sda1
/dev/sda1: Linux rev 1.0 ext4 filesystem data (needs journal recovery) (extents) (huge files)
[root@server1 ~]# file -s /dev/sda
```

```
/dev/sda: x86 boot sector; GRand Unified Bootloader, stage1 version 0x3, boot drive 0x80, 1st sector stage2 0x7f86, GRUB version 0.94;
partition 1: ID=0x83, active, starthead 32, startsector 2048, 512000 sectors; partition 2: ID=0x83, starthead 254, startsector 514048,
37332992 sectors; partition 3: ID=0x82, starthead 254, startsector 37847040, 4096000 sectors, code offset 0x48
```

1.6 文件/目录复制和移动

1.6.1 cp 命令

```
cp [-apdriulfs] src  dest   # 复制单文件或单目录
cp [-apdriuslf] src1  src2  src3......dest_dir  # 复制多文件、目录到一个目录下
选项说明：
-p：  文件的属性(权限、属组、时间戳)也复制过去。如果不指定 p 选项，谁执行复制动作，文件所有者和组就是谁。
-r 或-R：递归复制，常用于复制非空目录。
-d：复制的源文件如果是链接文件，则复制链接文件而不是指向的文件本身。即保持链接属性，复制快捷方式本身。如果不指定-d，则复制的是链接所指向的文件。
-a：a=pdr 三个选项。归档拷贝，常用于备份。
-i：复制时如果目标文件已经存在，询问是否替换。
-u：(update)若目标文件和源文件同名，但属性不一样(如修改时间，大小等)，则覆盖目标文件。
-f：强制复制，如果目标存在，不会进行-i 选项的询问和-u 选项的考虑，直接覆盖。
-l：在目标位置建立硬链接，而不是复制文件本身。
-s：在目标位置建立软链接，而不是复制文件本身(软链接或符号链接相当于 windows 的快捷方式)。
```

一般使用 cp -a 即可，对于目录加上-r 选项即可。

注意，bash 内置命令在进行通配符匹配文件的时候，"*"、"?"、"[]"是无法匹配到以"."开头的文件的，所以"*"不会匹配隐藏文件。要通配隐藏文件，使用"."代替上述几种通配元字符即可，它能匹配除了"."和".."这两个特殊目录外的所有文件。它并非通配符，而是表示当前目录，显然直接复制目录，是可以将隐藏文件复制走的。

例如，复制/etc/skel 目录下所有文件包括隐藏文件到/tmp 目录下。

```
cp -a /etc/skel/. /tmp
```

如果有重复文件，则即使加上-f 选项，也一样会交互式询问。解决方法可以是使用"yes"这个工具，它会不断的生成 y 字母直到进程被杀掉，当然也可以自行指定要生成的字符串。

```
yes | cp -a /etc/skel/. /tmp
```

1.6.2 scp 命令和执行过程分析

scp 是基于 ssh 的安全拷贝命令(security copy)，它是从古老的远程复制命令 rcp 改变而来，实现的是在 host 与 host 之间的拷贝，可以是本地到远程的、本地到本地的，甚至可以远程到远程复制。注意，scp 可能会询问密码。

如果 scp 拷贝的源文件在目标位置上已经存在时(文件同名)，scp 会替换已存在目标文件中的内容，但保持其 inode 号。

如果 scp 拷贝的源文件在目标位置上不存在，则会在目标位置上创建一个空文件，然后将源文件中的内容填充进去。

之所以解释上面的两句，是为了理解 scp 的机制，scp 拷贝本质是只是填充内容的过程，它不会去修改目标文件属性(时间戳属性除外)，对于从远程复制到另一远程时，其机制见后文。

```
scp [-12BCpqrv] [-l limit] [-o ssh_option] [-P port] [[user@]host1:]file1 ... [[user@]host2:]file2
选项说明：
-1：使用 ssh v1 版本，这是默认使用协议版本
-2：使用 ssh v2 版本
-C：拷贝时先压缩，节省带宽
-l limit：限制拷贝速度，Kbit/s.
-o ssh_option：指定 ssh 连接时的特殊选项，一般用不上。偶尔在连接过程中等待提示输入密码较慢时，可以设置 GSSAPIAuthentication 为 no
-P port：指定目标主机上 ssh 端口，大写的字母 P，默认是 22 端口
-p：拷贝时保持源文件的 mtime,atime,owner,group,privileges
-r：递归拷贝，用于拷贝目录。注意，scp 拷贝遇到链接文件时，会拷贝链接的源文件内容填充到目标文件中(scp 的本质就是填充而非拷贝)
-v：输出详细信息，可以用来调试或查看 scp 的详细过程，分析 scp 的机制
```

示例：

1.把本地文件/home/a.tar.tz 拷贝到远程服务器 192.168.0.2 上的/home/tmp，连接时使用远程的 root 用户：

```
scp /home/a.tar.tz  root@192.168.0.2:/home/tmp/
```

2.目标主机不写路径时，表示拷贝到对方的家目录下：

```
scp /home/a.tar.tz  root@192.168.0.2
```

3.把远程文件/home/a.tar.gz 拷贝到本机：

```
scp root@192.168.0.2:/home/a.tar.tz        # 不接本地目录表示拷贝到当前目录
scp root@192.168.0.2:/home/a.tar.tz  /tmp  # 拷贝到本地/tmp 目录下
```

4.拷贝远程机器的/home/目录到本地/tmp 目录下。

```
scp -r root@192.168.0.2:/home/  /tmp
```

5.从远程主机 192.168.100.60 拷贝文件到另一台远程主机 192.168.100.62 上。

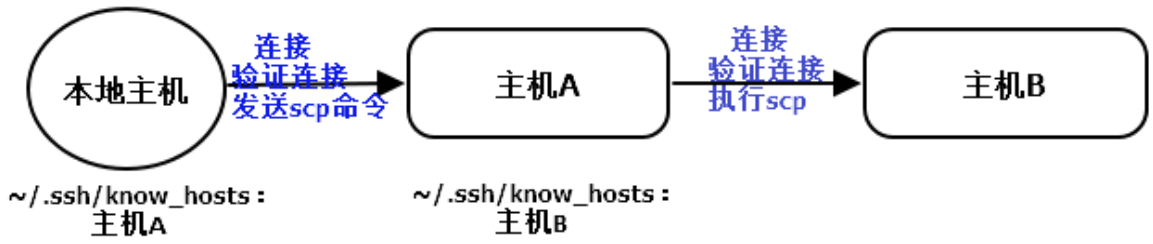
```
scp root@192.168.100.60:/tmp/copy.txt root@192.168.100.62:/tmp
```

在远程复制到远程的过程中，例如在本地执行 scp 命令将 A 主机(192.168.100.60)上的/tmp/copy.txt 复制到 B 主机(192.168.100.62)上的/tmp 目录下，如果使用-v 选项查看调试信息的话，会发现它的步骤类似是这样的。

```
# 以下是从结果中提取的过程
# 首先输出本地要执行的命令
Executing: /usr/bin/ssh -v -x -oClearAllForwardings yes -t -l root 192.168.100.60 scp -v /tmp/copy.txt root@192.168.100.62:/tmp

# 从本地连接到 A 主机
debug1: Connecting to 192.168.100.60 [192.168.100.60] port 22.
debug1: Connection established.

# 要求验证本地和 A 主机之间的连接
debug1: Next authentication method: password
root@192.168.100.60's password:

# 将 scp 命令行修改后发送到 A 主机上
debug1: Sending command: scp -v /tmp/copy.txt root@192.168.100.62:/tmp

# 在 A 主机上执行 scp 命令
Executing: program /usr/bin/ssh host 192.168.100.62, user root, command scp -v -t /tmp

# 验证 A 主机和 B 主机之间的连接
debug1: Next authentication method: password
root@192.168.100.62's password:

# 从 A 主机上拷贝源文件到最终的 B 主机上
debug1: Sending command: scp -v -t /tmp
Sending file modes: C0770 24 copy.txt
Sink: C0770 24 copy.txt
copy.txt                                                                                      100%   24     0.0KB/s

# 关闭本地主机和 A 主机的连接
Connection to 192.168.100.60 closed.
```

也就是说，远程主机 A 到远程主机 B 的复制，实际上是将 scp 命令行从本地传输到主机 A 上，由 A 自己去执行 scp 命令。也就是说，本地主机不会和主机 B 有任何交互行为，本地主机就像是一个代理执行者一样，只是帮助传送 scp 命令行以及帮助显示信息。

其实从本地主机和主机 A 上的~/.ssh/know_hosts 文件中可以看出，本地主机只是添加了主机 A 的信息，并没有添加主机 B 的信息，而在主机 A 上则添加了主机 B 的信息。



1.6.3 mv 命令

mv 命令移动文件和目录，还可以用于重命名文件或目录。

```
mv [-iuf] src dest                    # 移动单个文件或目录
mv [-iuf] src1  src2  src3  dest_dir  # 移动多个文件或目录
选项说明：
--backup[=CONTROL]：如果目标文件已存在，则对该文件做一个备份，默认备份文件是在文件名后加上波浪线，如/b.txt~
-b：类似于--backup，但不接受参数，默认备份文件是在文件名后加上波浪线，如/b.txt~
-f：如果目标文件已存在，则强制覆盖文件
-i：如果目标文件已存在，则提示是否要覆盖，这是 alias mv 的默认选项
-n：如果目标文件已存在，则不覆盖已存在的文件
    如果同时指定了-f/-i/-n，则后指定的生效
```

```
-u：(update)如果源文件和目标文件不同，则移动，否则不移动
```

mv 默认已经是递归移动,不需要-r 参数。

1.6.4 mv 的一个经典问题(mv 的本质)

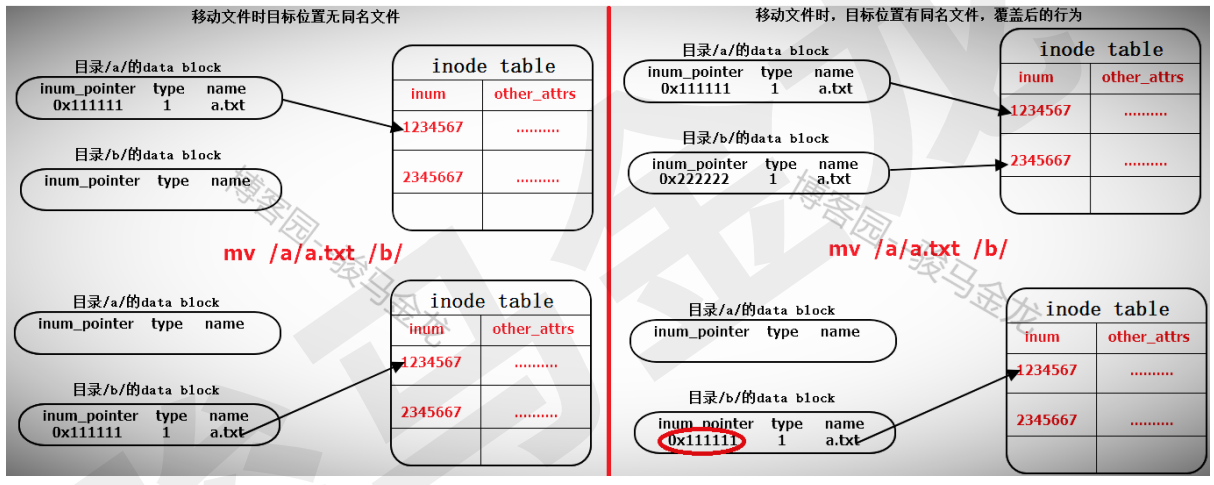
该问题涉及文件系统操作文件的机制，若不理解，请先深入学习文件系统。

mv 不能实现里层同名目录覆盖外层同名目录。如/tmp 下有 a 目录，a 目录里还有 a 目录。

```
[root@toystory tmp]# tree -L 3 a -fC
a
└── a/a
    ├── a/a/a

2 directories, 1 file
[root@toystory tmp]# mv  a/* .
mv: overwrite `./a'? y
mv: cannot move `a/a' to `./a': Directory not empty
[root@toystory tmp]# mv -f /tmp/a/* /tmp
mv: cannot move `/tmp/a/a' to `/tmp/a': Directory not empty
```

要解释为何会如此，先说明移动和覆盖动作的本质。

同文件系统下移动文件实际上是修改目标文件所在目录的 data block，向其中添加一行指向 inode table 中待移动文件的 inode 的指针，如果目标路径下有同名文件，则会提示是否覆盖，实际上是覆盖指向该同名文件的 inode 指针，由于同名文件的 inode 记录指针被覆盖，就无法再找到该文件的 data block，所以该文件被标记位删除。



跨文件系统移动文件的本质：如果目标路径下没有同名文件，则先为此文件分配一个 inode 号，并在目标目录的 data block 中添加一条指向该 inode 号的新记录(是全新的)，然后将文件复制到目标位置，复制成功则删除源文件，复制失败则保留源文件；如果目标路径下有同名文件，则提示是否要覆盖，如果选择覆盖，则将该同名文件的 inode 指针指向新分配的 inode 号，然后将文件复制到目标位置，复制成功则删除源文件，复制失败则保留源文件。

也就是说，同文件系统下移动文件时，inode 记录不变(如 inode 号)，当然，时间戳是一定会改变的，因为移动过程中修改了 inode 指向 data block 的指针。而跨文件系统下移动文件时，inode 记录完全改变，它是新添加的记录。

再考虑上面的问题，同文件系统下移动文件时先在目标位置/tmp 的 data block 中添加一条记录，如果同名则提示覆盖，覆盖时会先删除/tmp 的 data block 中的 a 对应的记录，再添加将要移动文件的记录。从上面的结果也可以看出是先提示覆盖再提示目录非空的错误。

设想下，如果/tmp/a/a 移动到/tmp 下并重命名为 b，则其动作是直接向/tmp 的 data block 中添加 b 的记录，如果此时正好/tmp 下已有 b 目录，则先删除/tmp 的 data block 中 b 目录对应的记录，再添加移动后的 b 记录。

但是现在不是重命名为 b，而是覆盖/tmp/a，此时的动作按原理应该是先提示是否覆盖，如果是，则删除/tmp 的 data block 中 a 对应的记录，但由于此时/tmp/a 目录中还有文件，该记录无法删除(因为如果要删除了该记录，代表删除了/tmp/a 整个目录，而删除整个/tmp/a 目录需要删除里面所有的文件，在删除它们之前的一个动作是把/tmp/a 中的所有目录和文件的 inode 号标记为未使用，但此刻要移动的源目录/tmp/a/a 是在使用当中的)，所以提示目录非空而无法删除，这里所指的非空目录指的是/tmp/a，而非是/tmp/a/a 非空。

但是在 Windows 操作系统下，里层目录是可以直接覆盖外层同名目录的，这和文件系统的行为有关。

其实在这个问题中，可以看出 mv 的很多原理。

1.7 查看文件内容

1.7.1 cat 命令

输出一个或多个文件的内容。

```
cat [OPTION]... [FILE]...
```

```
选项说明
-n：显示所有行的行号
-b：显示非空行的行号
-E：在每行行尾加上$符号
-T：将 TAB 符号输出为"^I"
-s：压缩连续空行为单个空行
```

cat 还有一个重要功能，允许将分行键入的内容输入到一个文件中去。

首先测试<<eof，这表示将键入的内容追加到标准输入 stdin 中(不是从标准输入中读取)， eof 可以随便使用其他符号代替。

```
[root@xuexi tmp]# cat <<eof
> abc.com
> eof
abc.com
```

再测试<eof，发现没有输入的机会，并且此时只能使用 eof 作为符号，EOF 或其他任何都不可以。因为<eof 是读取标准输入，会将 eof 当成输入文件处理。所以一定要使用<<eof，这表示 here document，而两个 eof 正是 document 的起始和结束标志。

```
[root@xuexi tmp]# cat <eof
[root@xuexi tmp]# cat <eox
-bash: eox: No such file or directory
[root@xuexi tmp]# cat <EOF
-bash: EOF: No such file or directory
```

再进一步测试<<eof 的功能，将键入的内容重定向到文件而非标准输入中。这时有两种书写方案：

第一种方案：>>filename<<eof 或>filename<<eof

```
[root@xuexi ~]# cat >>/tmp/test.txt<<EOF    # 输入到这里按回车键继续输入下一行
> xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx   # 按回车输入下一行
> yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy                          # 按回车输入下一行
> zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz      # 按回车输入下一行
> EOF                                                    # 顶格写 EOF 结束输入
```

第二种方案：<<eof>filename 或<<eof>>filename

```
[root@xuexi tmp]# cat <<eof>log.txt
> abc.com
> eof
```

两种方案结果是一样的，且总是使用<<eof，只不过所写的位置不同而已，不管写在哪个位置，它都表示将键入的内容追加到标准输入。然后再使用>filename 或>>filename 控制重定向的方式，将标准输入中的内容重定向到 filename 文件中。

1.7.2 tac

tac 和 cat 字母正好是相反的，其作用也是和 cat 相反的，它会反向输出行，将最后一行放在第一行的位置输出，依此类推。但是，tac 没有显示行号的参数。

```
echo -e '1\n2\n3\n4\n5' | tac
5
4
3
2
1
```

1.7.3 head

head 打印前面的几行。

```
head  [-n num] | [-num] [-v] filename
-n：显示前 num 行；如果 num 是负数，则显示除了最后|num|(绝对值)行的其余所有行，即显示前"总行数-|num|"
-v：会显示出文件名
```

-n num 是显示文件的前 num 行，num 可以是+/-或不加正负号的整数，如果是正整数或不写+号，则显示前 num 行。如果是负整数，则从后向前数 num 行，并打印除了这些行的前面所有的行，即打印除了最后 num 行的所有行，也即总行数减 num 的前正数行。不写-n 时默认是前 10 行。正整数时"-n num"可以直接简写"-num"。

不管怎么样，它取的都是前几行，哪怕是负整数也是前几行。

示例：

```
[root@xuexi ~]# echo -e '1\n2\n3\n4\n5' | head          # 取出默认前 10 行，但总共才有 5 行。
1
2
```

```
3
4
5
[root@xuexi ~]# echo -e '1\n2\n3\n4\n5' | head -2          # 取出前 2 行
1
2
```

或者

```
[root@xuexi ~]# echo -e '1\n2\n3\n4\n5' | head -n 2         # 取出前 2 行
1
2
[root@xuexi ~]# echo -e '1\n2\n3\n4\n5' | head -n -1        # 取出前 5-1=4 行
1
2
3
4
```

1.7.4 tail

tail 和 head 相反，是显示后面的行，默认是后 10 行。

```
tail [OPTION]... [FILE]...
选项说明：
-n：输出最后 num 行，如果使用-n +num 则表示输出从第 num 行开始的所有行
-f：监控文件变化
--pid=PID：和-f 一起使用，在给定 PID 的进程死亡后，终止文件监控
-v：显示文件名
```

"-n -num"或"-num"或"-n num"(num 为正整数)表示输出最后的 num 行。使用"-n +num"(num 为正整数)则表示输出从第 num 行开始的所有行。

```
[root@xuexi ~]# echo -e '1\n2\n3\n4\n5' | tail -3     # 等价于 tail -n 3 和 tail -n -3
3
4
5
[root@xuexi tmp]# seq 6 | tail -n +3  # 打印除了前 3-1=2 行的所有行
3
4
5
6
```

tail 还有一个重要的参数-f，监控文件的内容变化。当一个用户不断修改某个文件的尾部，另一个用户就可以通过这个命令来刷新并显示这些修改后的内容。

1.7.5 nl

以行号的方式查看内容。

常用"-b a"，表示不论是否空行都显示行号，等价于 cat -n；不写选项时，默认"-b t"，表示空行不显示行号，等价于 cat -b。

```
[root@xuexi ~]# nl /etc/issue               # 默认空行不显示行号
     1  CentOS release 6.6 (Final)
     2  Kernel \r on an \m

[root@xuexi ~]# nl -b a /etc/issue
     1  CentOS release 6.6 (Final)
     2  Kernel \r on an \m
     3
```

1.7.6 more 和 less

按页显示文件内容。使用 more 时，使用/搜索字符串，按下 n 或 N 键表示向下或向上继续搜索。使用 less 时，还多了一个搜索功能，使用?搜索字符串，同样，使用 n 或 N 键可以向上或向下继续搜索。

1.7.7 比较文件内容

```
diff file1 file2
vimdiff file1 file2
```

1.8 文件查找类命令

1.8.1 which

显示命令或脚本的全路径，默认也会将命令的别名显示出来。

```
which mv
```

```
alias mv='mv -i'
        /bin/mv
```

1.8.2 whereis

找出二进制文件、源文件和 man 文档文件。

```
whereis cd
```

```
cd: /usr/bin/cd /usr/share/man/man1/cd.1.gz /usr/share/man/man1p/cd.1p.gz /usr/share/man/mann/cd.n.gz
```

1.8.3 whatis

列出给定命令(并非一定是命令)的 man 文档信息。

```
whatis passwd
```

```
sslpasswd (1ssl)      - compute password hashes
passwd (1)            - update user's authentication tokens
passwd (5)            - password file
```

根据上面的结果，执行：

```
man 1 passwd    # 获取 passwd 命令的 man 文档
man 5 passwd    # 获取 password 文件的 man 文档，文件类的 man 文档说明的是该文件中各配置项意义
man sslpasswd   # 获取 sslpasswd 命令的 man 文档，实际上是 openssl passwd 的 man 文档
```

1.8.4 locate

没什么好说的。

1.8.5 find

内容太多，所以下面单独说明。

1.9 find

搜索文件或目录，功能非常强大。该工具是 findutils 包提供的，该包中还包括一个老版本的 oldfind 工具，以及另一个非常强大的 xargs 命令，在搜索文件时，如果还要对搜索的文件进行后续的处理，一般都会结合 xargs 来实现。但本文并不过多涉及 xargs，如果要了解 xargs 用法，见我的另一篇关于 xargs 的总结，目前在网上暂时还没找到比这篇文档更详细的 xargs 说明。

find 搜索是从磁盘开始搜索，而不是从数据库搜索。

```
find [path...] [expression_list]
```

1.9.1 find 基本用法示例

在此处只给出 find 的基本用法示例，如果有不理解的部分，则看后面的 find 理论说明，也建议在看完这些基本示例后阅读一遍下面的理论说明，它是本人翻译自 find 的 man 文档并加上了个人的理解。另外，在理论说明结束后，还有 find 深入用法示例。

(1). 最基础的打印操作

find 命令默认接的命令是-print，它默认以\n 将找到的文件分隔。可以使用-print0 来使用\0 分隔，这样就不会分行了。但是一定要注意，-print0 针对的是\n 转\0，如果查找的文件名本身就含有空格，则 find 后-print0 仍然会显示空格文件。所以-print0 实现的是\n 转\0 的标记，可以使用其他工具将\0 标记替换掉，如 xargs，tr 等。

```
[root@xuexi tmp]# mkdir /tmp/a
[root@xuexi tmp]# touch /tmp/a/{1..5}.log
[root@xuexi tmp]# find /tmp/a   # 等价于 find /tmp/a  -print，表示搜索/tmp/a 目录
```

```
/tmp/a
/tmp/a/4.log
/tmp/a/2.log
/tmp/a/5.log
/tmp/a/1.log
/tmp/a/3.log
```

```
[root@xuexi tmp]# find /tmp/a -print0
```

```
/tmp/a/tmp/a/4.log/tmp/a/2.log/tmp/a/5.log/tmp/a/1.log/tmp/a/3.log
```

(2). 文件名搜索：-name 或-path

-name 可以对文件的 basename 进行匹配，-path 可以对文件的 dirname+basename。查找的文件名最好使用引号包围，可以配合通配符进行查找。

```
[root@xuexi tmp]# find /tmp -name "*.log"
/tmp/screen.log
/tmp/x.log
/tmp/timing.log
/tmp/a/4.log
/tmp/a/2.log
/tmp/a/5.log
/tmp/a/1.log
/tmp/a/3.log
/tmp/b.log
```

但不能在-name 的模式中使用"/"，除非文件名中包含了字符"/"，否则将匹配不到任何东西，因为-name 只对 basename 进行匹配。例如，想要匹配/tmp 目录下某包含字符 a 的目录下的 log 文件。

```
find /tmp -name '*a*/*.log'
find: warning: Unix filenames usually don't contain slashes (though pathnames do).  That means that '-name '*a*/*.log'' will probably evaluate to false all the time on this system.  You might find the '-wholename' test more useful, or perhaps '-samefile'.  Alternatively, if you are using GNU grep, you could use 'find ... -print0 | grep -FzZ '*a*/*.log''.
```

所以想要在指定目录下搜索某目录中的某文件，应该使用-path 而不是-name。

```
[root@server2 tmp]# find /tmp -path '*a*/*.log'
/tmp/abc/axyz.log
```

注意，配合通配符[]时应该注意是基于字符顺序的，大小写字母的顺序是 a-z --> A-Z，指定[a-z]表示小写字母 a-z，同理[A-Z]，而[a-zA-Z]和[a-Z]都表示所有大小写字母。当然还可以指定[a-A]表示 a-z 外加一个 A。

字母的处理顺序较容易理解，关于数字的处理方法，见下面的示例。

```
[root@xuexi test]# ls
11.sh  1.sh  22.sh  2.sh  3.sh
[root@xuexi test]# find -name "[1-2].sh"
./2.sh
./1.sh
[root@xuexi test]# find -name "[1-23].sh"
./2.sh
./3.sh
./1.sh
[root@xuexi test]# touch 0.sh
[root@xuexi test]# find -name "[1-20].sh"
./2.sh
./0.sh
./1.sh
[root@xuexi test]# find -name "[1-22-3].sh"
./2.sh
./3.sh
./1.sh
```

从上面结果可以看出，其实[]只能匹配单个字符，[0-9]表示 0-9 的数字，[1-20]表示[1-2]外加一个 0，[1-23]表示[1-2]外加一个 3，[1-22-3]表示[1-2]或[2-3]，迷惑点就是看上去是大于 10 的整数，其实是两个或者更多的单个数字组合体。也可以用这种方法表示多种匹配：[1-2,2-3]。

(3). 根据文件类型搜索：-type

一般需要搜索的文件类型就只有普通文件(f)，目录(d)，链接文件(l)。

例如，搜索普通文件类的文件，且名称为 a 开头的 sh 文件。

```
[root@xuexi test]# find /tmp -type f -name "a*.sh"
```

搜索目录类文件，且目录名以 a 开头。

```
[root@xuexi test]# find /tmp -type d -name "a*"
```

(4). 根据文件的时间戳搜索：-atime/-mtime/-ctime

例如搜索/tmp 下 3 天内修改过内容的 sh 文件，因为是文件内容，所以不考虑搜索目录。

```
find /tmp -type f -mtime -3 -name "*.sh"
```

至于为什么是"-3"，见后面 find 理论部分内容。

(5). 根据文件大小搜索：-size

例如搜索/tmp 下大于 100K 的 sh 文件。

```
find /tmp -type f -size +100k -name '*.sh'
```

(6). 根据权限搜索：-perm

例如搜索/tmp 下所有者具有可读可写可执行权限的 sh 文件。

```
find /tmp -type f -perm -0700 -name '*.sh'
```

(7). 搜索空文件

空文件可以是没有任何内容的普通文件，也可以是没有任何内容的目录。

例如搜索目录中没有文件的空目录。

```
find /tmp -type d -empty
```

(8). 搜索到文件后并删除

例如搜索到/tmp 下的".tmp"文件然后删除。

```
find /tmp -type f -name "*.tmp" -exec rm -rf  '{}'  \;
```

(9). 搜索指定日期范围的文件，例如搜索/test 下 2017-06-03 到 2017-06-06 之间修改过的文件

```
find /test -type f -newermt 2017-06-03 -a ! -newermt 2017-06-06
```

或者，创建两个临时文件，并用 touch 修改这两个文件的修改时间，然后`find -newer`去参照这两个文件

```
touch -m -d 2017-06-03 tmp1.txt
touch -m -d 2017-06-06 tmp2.txt
find /test -type f -newer tmp1.txt -a ! -newer tmp2.txt
```

不过这样会把 tmp2.txt 也搜索出来，因为 newer 搜索的是比 xxx 文件更新，取反则表示更旧或时间相同。

(10). 并行加速搜索

有时候，想要搜索的内容并不知道在哪里，这时我们会从根"/"开始搜索，这样的搜索速度可能会稍微长那么一点点。为了加速搜索，使用 xargs 的并行功能。例如，搜索"/"下的所有"Find.pm"结尾的文件：

```
ls --hide proc / | xargs -i -P 0 find /{} -type f -name "*Find.pm"
```

可以使用 time 命令看看 cpu 利用率：

```
/usr/bin/time bash -c 'ls --hide proc / | xargs -i -P 0 find /{} -type f -name "*Find.pm" | sort'
/perlapp/perl-5.26.2/cpan/Pod-Parser/lib/Pod/Find.pm
/perlapp/perl-5.26.2/ext/File-Find/lib/File/Find.pm
/usr/share/perl5/vendor_perl/Pod/Find.pm
/usr/share/perl5/File/Find.pm
0.04user 0.25system 0:00.19elapsed 149%CPU (0avgtext+0avgdata 5492maxresident)k
0inputs+0outputs (0major+12685minor)pagefaults 0swaps
```

(11). 获取搜索到文件的绝对路径

当 find 结合管道，而管道后的命令很可能想要获取到搜索到的文件的绝对路径，或者说是全路径。而问题是，当 find 的搜索路径是相对路径时，搜索出来的显示结果也是以相对路径显示的。

```
mkdir /tmp/test
touch /tmp/test/{a,b,c}.png
find .
```

```
.
./a.png
./b.png
./c.png
```

想要获取全路径，方式有很多种：

```
# 搜索前先 pwd
find $(pwd)
```

```
/tmp/test
/tmp/test/a.png
/tmp/test/b.png
/tmp/test/c.png
```

```
# 或使用$PWD 环境变量
find $PWD
```

```
/tmp/test
/tmp/test/a.png
/tmp/test/b.png
/tmp/test/c.png
```

```
# 执行 readlink，它不仅解析软链接，也可以使用-f 选项解析普通文件
find . -exec readlink -f {} \;
```

```
/tmp/test
/tmp/test/a.png
/tmp/test/b.png
/tmp/test/c.png
```

```
# 使用 bash 的波浪号扩展 ~+
find ~+
```

```
/tmp/test
/tmp/test/a.png
/tmp/test/b.png
/tmp/test/c.png
```

(12). 获取搜索到文件的文件名部分(basename)

find 的-printf 选项有很多修饰符功能，对于处理路径方面的修饰符有%f、%p、%P，其中%f 是获取 basename（去除所有路径前缀），%p 是获取路径自身，一般用不上，%P 是获取除了 find 搜索路径的剩余部分。

首先，想要获取 basename，建议使用%f。

```
mkdir /tmp/test/test1
touch /tmp/test/test1/{x,y,z}.png
find /tmp/test -printf "%f\n"
```

```
test
a.png
b.png
c.png
test1
x.png
y.png
z.png
```

再看使用%P 的效果。结果仅仅是去掉了 find 搜索路径/tmp/test 部分。当搜索路径只有一层(即没有子目录)时，它也可以用来获取 basename。

```
find /tmp/test -printf "%P\n"
```

```

a.png
b.png
c.png
test1
test1/x.png
test1/y.png
test1/z.png
```

(13). 从结果中排除目录自身

find 搜索目录时，总是会将搜索路径自身也包含到搜索结果中。想办法排除它是必须的。

排除的方法是，加上一个-path 选项并取反，-path 的参数和 find 的搜索路径参数必须一致。

```
find /tmp/test ! -path /tmp/test
```

```
/tmp/test/a.png
/tmp/test/b.png
/tmp/test/c.png
/tmp/test/test1
/tmp/test/test1/x.png
/tmp/test/test1/y.png
/tmp/test/test1/z.png
```

```
find . ! -path .
```

```
./a.png
./b.png
./c.png
./test1
./test1/x.png
./test1/y.png
./test1/z.png
```

1.9.2 find 理论部分

```
find [path...] [expression_list]
```

expression 分为三种：options、test、action。对于多个表达式，find 是**从左向右处理的**，所以表达式的前后顺序不同会造成不同的搜索性能差距。

find 首先对整个命令行进行语法解析，并应用给定的 options，然后定位到搜索路径 path 下开始对路径下的文件或子目录进行表达式评估或测试，评估或测试的过程是按照表达式的顺序从左向右进行的(此处不考虑操作符的影响)，如果最终表达式的表达式评估为 true，则输出(默认)该文件的全路径名。

一定要明白的是，find 的搜索机制是根据表达式返回的 true/false 决定的，每搜索一次都判断一次是否能确定最终评估结果为 true，只有评估的最终结果为 true 才算是找到，并切入到下一个搜索点。

1.9.2.1 expression-operators

操作符控制表达式运算方式。确切的说，是控制 expression 中的 options/tests/actions 的运算方式，无论是 options、tests 还是 actions，它们都可以给定多个，例如 find /tmp -type f -name "*.log" -exec ls '{}' \; -print，该 find 中给定了两个 test，两个 action，它们之间从前向后按顺序进行评估，所以如果想要改变运算逻辑，需要使用操作符来控制。

注意，理解 and 和 or 的评估方式非常重要，很可能写在 and 或 or 后面的表达式不起作用，而导致跟想象中的结果不一样。

下面的操作符优先级从高到低。

- (expr)：优先级最高。为防止括号被 shell 解释(进入子 shell)，所以需要转义，即\(...\)
- ! expr：对 expr 的 true 和 false 结果取反。同样需要使用引号包围
- -not expr：等价于"! expr"
- expr1 expr2：等同于 and 操作符。
- expr1 -a expr2：等同于 and 操作符
- expr1 -and expr2：首先要求 expr1 为 true，然后 expr2 以 expr1 搜索的结果为基础继续检测，然后再返回检测值为 true 的文件。因为 expr2 是以 expr1 结果为基础的，所以如果 expr1 返回 false，则 expr2 直接被忽略而不会进行任何操作
- expr1 -o expr2：等同于 or 操作符
- expr1 -or expr2：只有 expr1 为假时才评估 expr2。
- expr1 , expr2：逗号操作符表示列表的意思，expr1 和 expr2 都会被评估，但 expr1 的 true 或 false 是被无视的，只有 expr2 的结果才是最终状态值。

关于 and 和 or 操作符，一定要明确的是 and 后的表达式操作的对象是前面表达式的结果，而 or 操作符的对象则是前面评估后为假时文件。

例如：

```
find /tmp -type f -name "*.log"
```

它是一个 and 操作符，-name 表达式是在-type 筛选的结果基础上再匹配文件名的。但如果是：

```
find /tmp -type f -o -name "*.log"
```

则-type f 返回真的文件直接执行它后面默认的-print，而不满足-type f 的才会去被-name 评估，所以返回结果中即有任意普通文件，也有任意 log 文件，但两者同名的文件只返回一次。

总之，and 和 or 对于 find 的影响比较大，后面会专门用一节彻底搞懂 operator 和 action 来详细解释。另外，在后文 find 深入用法示例中还有一个关于"忽略目录"的例子，它很好的解释了操作符的行为。

1.9.2.2 expression-options

options 总是返回 true。除了"-daystart"，options 会影响所有指定的 test 表达式部分，哪怕是 test 部分写在 options 的前面。这是因为 options 是在命令行被解析完后立即处理的，而 test 是在检测到文件后才处理的。对于"-daystart"这个选项，它们仅仅影响写在它们后面的 test 部分，因此，建议将任何 options 部分写在 expression 的最前面，若不如此，会给出一个警告信息。

- -daystart：指定以每天的开始(凌晨零点)计算关于天的时间，用于改变时间类(-amin,-atime,-cmin,-ctime,-mmin 和-mtime)的计算方式。默认情况下，天的计算是从 24 小时前计算的。例如，当前时间为 5 月 3 日 17:00，要求搜索出 2 天内修改过的文件，默认搜索文件的起点是 5 月 1 日 17:00，如果使用-daystart，则搜索文件的起点是是 5 月 1 日 00:00。注意，该选项只会影响写在它后面的 test 表达式。
- -depth：搜索到目录时，先处理目录中的文件(子目录)，再处理目录本身。对于"-delete"这个 action，它隐含了"-depth"选项。
- -maxdepth levels：指定 tests 和 actions 作用的最大目录深度，只能为非负整数。可以简单理解为目录搜索深度，但并非如此。当前 path 目录的层次为 1，所以若指定-maxdepth 0 则得不到任何结果。
- -mindepth levels：tests 和 actions 不会应用于小于指定深度的目录，"-mindepth 1"表示应用于所有的文件。
- -ignore_readdir_race：有时候 find 无法用 stat 检测文件的信息时(如无权限)会给出如下图所示的错误信息，如果想要忽略该信息，可以使用该选项。

```
find: `/proc/11142/task/11142/fd/5': No such file or directory
find: `/proc/11142/task/11142/fdinfo/5': No such file or directory
find: `/proc/11142/fd/5': No such file or directory
find: `/proc/11142/fdinfo/5': No such file or directory
```

- -warn：忽略警告信息。

1.9.2.3 expression-tests

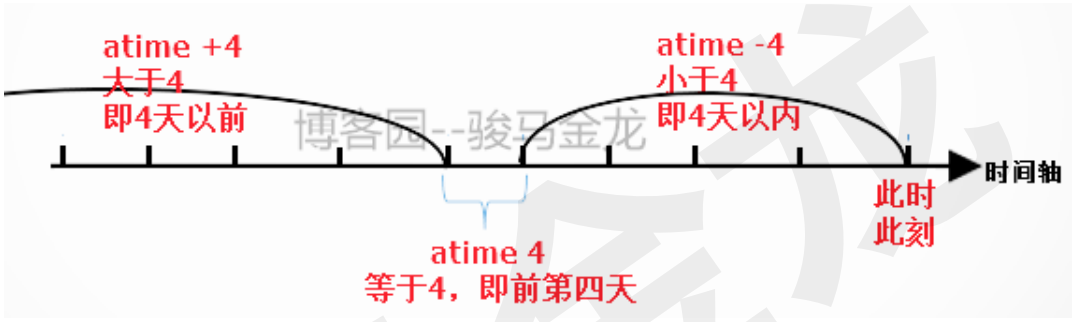
find 解析完命令行语法之后，开始搜索文件，在搜索过程中，每次检测到的文件都会被 test expression 进行测试，符合条件的将被保留。

数值部分可以设置为以下 3 种模式：n 可以是小数。

- → +n：大于 n
- → -n：小于 n
- → n：精确的等于 n

对于文件大小而言，文件的大小是精确的，指定 100KB，则比对的值必定是 100KB。此时(+ -)n 和字面意思是一样的。

但对于时间而言，时间是有时间段的，例如指定前第四天，第四天也整整占用了一天，所以(+ -)n 和文件大小的计算方法是不一样的。



find 在计算以天数为单位的时间时，默认会转换为 24 小时制，除非同时指定了"-daystart"这个选项，这在前面已经解释过了。例如当前时间为 5 月 3 号 17:00，那么计算"atime +1"的时候，真正计算的是 24*1=24 小时之前，也就是 5 月 2 号 17:00 以前被访问过，若指定了"-daystart"，则计算的是 5 月 2 号 00:00 之前被访问过。

具体的选项如下：

- -type X：根据文件类型来搜索
  - → b：块设备文件
  - → c：字符设备文件
  - → d：目录
  - → p：命名管道文件(FIFO 文件)
  - → f：普通文件
  - → l：符号链接文件，即软链接文件
  - → s：套接字文件(socket)

【文件大小或内容类测试条件】

- -size n[cwbkMG]：根据文件大小来搜索，可以是(+-)n，单位可以是：
  - → b：512 字节的(默认单位)
  - → c：1 字节的
  - → w：2 字节
  - → k：1024 字节
  - → M：1024k
  - → G：1024M
- empty：空文件，对于目录来说，则是空目录

【文件名或路径名匹配类测试条件】

- -name pattern：文件的 basename(不包括其前导目录的纯文件名)能被通配符模式的 pattern 匹配到。由于前导目录被移除，所以 find 对包含"/"的 pattern 是绝对不可能匹配到内容的，例如" -name a/b"的结果一定是空且会给出警告信息，若要匹配这样的文件，可以考虑使用"-path"或者"-name b"。需要注意的是，在 find 中的通配元字符"*"、"?"和"[]"是能够匹配以点开头的文件的，之所以要在此说明这一点，是因为在 bash 中，这些通配元字符默认是无法匹配"."开头的文件的，例如"cp ~/* /tmp"是不会把隐藏文件也拷贝走的。若要忽略一个目录及其内的文件，可以配合"-prune"，它会跳过整个目录而不对此目录做任何检查。注意 pattern 要用引号包围防止被 shell 解释
- -iname pattern：不区分大小的"-name"
- -path pattern：文件名能被通配符模式的 pattern 匹配到。此模式下，通配元字符"*"、"?"和"[]"不会认为字符"/"或"."是特殊字符，也就是说这两个字符也在通配的范围内，所以能匹配到这两个字符。例如 find . -path "./sr*sc"可以匹配到名为"./src/misc"的目录。find 会将"-path"的 pattern 与文件的 dirname 和 basename 的结合体进行比较，由于 dirname 和 basename 的结合体是不包含尾随"/"的，所以如果 pattern 中指定了尾随"/"是不可能匹配到任何东西的，例如 find /tmp -path "/tmp/ab*/"，实际上它会给出警告信息提示 pattern 是以"/"结尾的。使用"-path"的时候，一定要注意"-path"后指定的路径起点是属于"find path expression"的 path 内的，例如"find /bar -path /foo/bar/myfile -print"是不可能匹配到任何东西的。若要忽略目录及其内的文件，可以配合"-prune"，它会跳过整个目录而不对此目录做任何检查。例如"find . -path ./src/emacs -prune -o -print"将跳过对目录"./src/emacs"的检查。注意 pattern 要用引号包围防止被 shell 解释
- -ipath pattern：不区分大小写的"-path"
- -regex pattern：文件名能被正则表达式 pattern 匹配到的文件。正则匹配会匹配整个路径，例如要匹配文件名为"./fubar3"的文件，可以使用".*bar."或".*b.*3"，但是不能是"f.*r3"，默认 find 使用的正则表达式类型是 Emacs 正则，但是可以使用-regextype 来改变正则类型
- -iregex pattern：不区分大小写的"-regex"

【权限类测试条件】

- -perm mode：精确匹配给定权限的文件。"-perm g=w"将只匹配权限为 0020 的文件。当然，也可以写成三位数字的权限模式
- -perm -mode：匹配完全包含给定权限的文件，这是最可能用上的权限匹配方式。例如给定的权限为"-0766"，则只能匹配"N767"、"N777"和"N776"这几种权限的文件，如果使用字符模式的权限，则必须指定 u/g/o/a，例如"-perm -u+x,a+r"表示至少所有人都有读权限，且所有者有执行权限的文件
- -perm /mode：匹配任意给定权限位的权限，例如"-perm /640"可以匹配出 600，040,700,740 等等，只要文件权限的任意位能包含给定权限的任意一位就满足
- -perm +mode：由于某些原因，此匹配模式被替换为"-perm /mode"，所以此模式已经废弃
- -executable：具有可执行权限的文件。它会考虑 acl 等的特殊权限，只要是可执行就满足。它会忽略掉-perm 的测试
- -readable：具有可读权限的文件。它会考虑 acl 等的特殊权限，只要是可读就满足。它会忽略掉-perm 的测试
- -writable：具有可写权限的文件。它会考虑 acl 等的特殊权限，只要是可写就满足。它会忽略掉-perm 的测试

【所有者所属组类测试条件】

- -gid n：gid 为 n 的文件
- -group gname：组名为 gname 的文件
- -uid n：文件的所有者的 uid 为 n
- -user uname：文件的所有者为 uname，也可以指定 uid
- -nogroup：匹配那些所属组为数字格式的 gid，且此 gid 没有对应组名的文件
- -nouser：匹配那些所有者为数字格式的 uid，且此 uid 没有对应用户名的文件

【时间戳类测试条件】

- -anewer file：atime 比 mtime 更接近现在的文件。也就是说，文件修改过之后被访问过
- -cnewer file：ctime 比 mtime 更接近现在的文件
- -newer file：比给定文件的 mtime 更接近现在的文件。
- -newer[acm]t TIME：atime/ctime/mtime 比时间戳 TIME 更新的文件
- -amin n：文件的 atime 在范围 n 分钟内改变过。注意，n 可以是(+-)n，例如-amin +3 表示在 3 分钟以前
- -cmin n：文件的 ctime 在范围 n 分钟内改变过
- -mmin n：文件的 mtime 在范围 n 分钟内改变过
- -atime n：文件的 atime 在范围 24*n 小时内改变过
- -ctime n：文件的 ctime 在范围 24*n 小时内改变过
- -mtime n：文件的 mtime 在范围 24*n 小时内改变过
- -used n：最近一次 ctime 改变 n 天范围内，atime 改变过的文件，即 atime 比 ctime 晚 n 天的文件，可以是(+-)n

【软硬链接类测试条件】

- -samefile name：找出指定文件同 indoe 的文件，即其硬链接文件
- -inum n：inode 号为 n 的文件，可用来找出硬链接文件。但使用"-samefile"比此方式更方便

- -links n：有 n 个软链接的文件

【杂项测试】

- -false：总是返回 false
- -true：总是返回 true

1.9.2.4 expression-actions

actions 部分一般都是执行某些命令，或实现某些功能。这部分是 find 的 command line 部分。

- -delete：删除文件，如果删除成功则返回 true，如果删除失败，将给出错误信息。"-delete"动作隐含了"-depth"这个 option。
- -exec command ;：注意有个分号";"结尾，该 action 是用于执行给定的命令。如果命令的返回状态码为 0 则该 action 返回 true。command 后面的所有内容都被当作 command 的参数，直到分号";"为止，其中参数部分使用字符串"{}"时，它表示的是 find 找到的文件名，即在执行命令时，"{}"会被逐一替换为 find 到的文件名，"{}"可以出现在参数中的任何位置，只要出现，它都会被文件名替换。注意，分号";"需要转义，即"\;"，如有需要，可以将"{}"用引号包围起来
- -ls：总是返回 true。将找到的文件以"ls -dils"的格式打印出来，其中文件的 size 部分是以 KB 为单位
- -ok command ;：类似于-exec，但在执行命令前会交互式进行询问，如果不同意，则不执行命令并返回 false，如果同意，则执行命令，但执行的命令是从/dev/null 读取输入的
- -print：总是返回 true。这是默认的 action，输出搜索到文件的全路径名，并尾随换行符"\n"。由于在使用"-print"时所有的结果都有换行符，如果直接将结果通过管道传递给管道右边的程序，应该要考虑到这一点：文件名中有空白字符(换行符、制表符、空格)将会被右边的程序误分解，如文件名为"ab c.txt"的文件将被认为是 ab 和 c.txt 两个文件，如果不想被此分解影响，可以考虑使用"-print0"替代"-print"将所有的换行符替换为"\0"
- -printf：输出格式太多，所以具体用法见 man 文档
- -print0：总是返回 true。输出搜索到文件的全路径名，并尾随空字符"\0"。由于尾随的是空字符，所以管道传递给右边的程序，然后只需对这个空字符进行识别分隔就能保证文件名不会因为其中的空白字符被误分解
- -prune： 不进入目录，所以可用于忽略目录，但不会忽略普通文件。没有给定-depth 时，总是返回 true，如果给定了-depth，则直接返回 false，所以-delete(隐含了-depth)是不能和-prune 一起使用的。

一定要注意，**action 是可以写在 tests 表达式前面的，它并不一定是在 test 表达式之后执行**。

例如：

```
find /tmp -print -type f -name "*.txt"
/tmp
/tmp/userfile
/tmp/passwdfile
/tmp/testdir
/tmp/testdir/a.log
/tmp/a.txt
/tmp/time.sh
/tmp/b.txt
/tmp/abc
/tmp/abc/axyz.log
/tmp/about.html
```

它将输出/tmp 下所有文件，而不是输出满足-type f -name "*.txt"的文件，因为-print 总是返回 true，它是第一个表达式，所以直接输出整个目录，后面的表达式-type f -name "*.txt"虽然在-print 之后也被评估了，但是却没有对应的 action，所以它们的匹配行为并没显示出来。如果在表达式-type f -name "*.txt"之后加上另外一个 action，那么这个 action 将只针对匹配到的文件进行操作。

```
[root@server2 tmp]# find /tmp -print -type f -name "*.txt" -ls
/tmp
/tmp/userfile
/tmp/passwdfile
/tmp/testdir
/tmp/testdir/a.log
/tmp/a.txt
69617151    4 -rw-r--r--   1 root     root         1758 Jun  8 03:50 /tmp/a.txt
/tmp/time.sh
/tmp/b.txt
101409594    4 -rw-r--r--   1 root     root           68 Jun  8 08:02 /tmp/b.txt
/tmp/abc
/tmp/abc/axyz.log
/tmp/about.html
```

可以看到结果中，只有满足条件的两个文件才执行了 ls 命令。

还可以写多个 action，写在哪里要注意它们的执行顺序。其实，不推荐使用除了"-print"、"-print0"和"-prune"外其他所有的 action，如有需要，应该配合 xargs 命令来实现目的。

1.9.3 彻底搞懂 operators 和 actions

find 的逻辑是很严格的，但是如果没有深究过，可能会出现不容易理解的偏差。

下面，我对 and 和 or 操作符，并以-print 这个 action 为例来详细说明它们的关系。在测试的时候，find 的"-D debug"选项非常好用，它能让我们知道 find 是按照什么逻辑处理各个选项的。debug 有几种类型，这里选择"-D rates"这种调试类型。

首先要明确一个结论，**and 的优先级高于 or**。

```
"expr1 -a expr2"：-a 的逻辑是只有当 expr1 为真时才评估 expr2。
"expr1 -o expr2"：-o 的逻辑是只有当 expr1 为假时才评估 expr2。
expr1 -o expr2 -a expr3：and 的优先级高于 or，所以等价于 expr1 -o (expr2 -a expr3)
```

例如，'-name "*.log" -o -name "*.txt"'，搜索到 1.txt 文件时，*.log 评估结果为假，于是评估*.txt，评估为真，于是整个-o 逻辑返回真。

第二个结论：**如果 find 评估完所有表达式后发现没有 action(-prune 这个 action 除外)，则在最末尾加上-print 作为默认的 action**。注意，这个默认的 action 是在评估完所有表达式后加上的。且还需注意，如果只有-prune 这个 action，它还是会补上-print。

以下是准备数据：

```
rm -rf /tmp/*
touch {1,2,3,4}.txt {a,b}.log
```

1.9.3.1 测试"and"操作符

现在先测试"expr1 -a expr2"中的"-a"。

```
[root@xuexi tmp]# find /tmp -type f -a -name "*.log"
/tmp/a.log
/tmp/b.log
```

用"-D rates"来看调试结果：

```
[root@xuexi tmp]# find -D rates /tmp -type f -a -name "*.log"
/tmp/a.log
/tmp/b.log
Predicate success rates after completion:
 ( -name *.log [0.8] [2/12=0.166667] -a [0.95] [2/12=0.166667] [need type] -type f [0.95] [2/2=1]  ) -a [0.76] [2/12=0.166667] -print [1] [2/2=1]
```

将最后一行进行简化，删掉所有中括号中的内容，得到的结果是：

```
( -name *.log -a -type f ) -a -print
```

所以，上面的语句执行逻辑是：

```
find /tmp ( -name *.log -a -type f ) -a -print
```

可见，find 自动补上了-print 这个默认的 action，且逻辑是评估-name，在-name 为真的基础上再评估-type，最后在-type 为真的基础上评估-print，也就是将 log 后缀且是普通文件的文件打印出来。

这里的-name 为什么被修改到了-type 的前面去？这是因为 find 自带优化功能，它会评估各个表达式的开销，将开销小的表达式放在前面，以便优化搜索。但要注意的是，优化后的表达式和我们给出的 find 命令行的结果不会有任何出入，它们在逻辑上是相同的，只是更换了一些表达式的前后位置。

然后，按照前面的结论，猜测下面 3 个 find 语句的逻辑：

```
find /tmp -type f -a -name "*.log" -print
find /tmp -type f -print -a -name "*.log"
find /tmp -type f -print -a -name "*.log" -print
```

上面 3 个 find 都给出了 action，所以 find 不会再补上默认的 action：-print。

所以，上面 3 个语句分别等价于下面 3 个语句：

```
find /tmp ( -name *.log -a -type f ) -a -print
find /tmp ( -type f -a -print ) -a -name *.log
find /tmp (( -type f -a -print ) -a -name *.log ) -a -print
```

唯一需要注意的是，有时候某些 expr 后面没有 action，这部分的 expr 将不会做任何处理，正如上面第二个语句，-name *.log 后并不会补上-pinrt，所以对于 find 来说，-a -name *.log 这个条件完全是多余的表达式。

以下是上述 3 个 find 命令的输出结果以及分析。

```
[root@xuexi tmp]# find /tmp -type f -a -name "*.log" -print
```

```
/tmp/a.log
/tmp/b.log
```

-type f 评估后得到所有普通文件，-name 再在普通文件的基础上筛选文件名后缀为".log"的文件，最后评估-print 输出".log"文件。

```
[root@xuexi tmp]# find /tmp -type f -print -a -name "*.log"
/tmp/a.log
/tmp/1.txt
/tmp/2.txt
/tmp/3.txt
/tmp/4.txt
/tmp/b.log
```

-type f 评估后得到所有普通文件，然后在普通文件的基础上评估-print，它直接将所有普通文件都输出，再在输出的文件的基础上，评估-name，得到 log 文件，但之后没有 action，所以评估得到 log 文件的结果作废。

```
[root@xuexi tmp]# find /tmp -type f -print -a -name "*.log" -print
/tmp/a.log
/tmp/a.log
/tmp/1.txt
/tmp/2.txt
/tmp/3.txt
/tmp/4.txt
/tmp/b.log
/tmp/b.log
```

-type f 评估后得到所有普通文件，然后在普通文件的基础上评估-print，它直接将所有普通文件都输出，再在输出的文件的基础上，评估-name，得到 log 文件，再对得到的 log 文件评估-print，它将再次输出 log 文件，所以结果中将输出两次 log 文件。

1.9.3.2 测试"or"操作符

有了上面的基础，再理解 or 操作符就简单多了。

直接猜下面 4 个命令的等价语句：

```
find /tmp -type f -o -name "*.log"
    --> find /tmp ( -type f -o -name *.log ) -a -print

find /tmp -type f -o -name "*.log" -print
    --> find /tmp -type f -o ( -name *.log -a -print )

find /tmp -type f -print -o -name "*.log"
    --> find /tmp ( -type f -a -print ) -o -name *.log

find /tmp -type f -print -o -name "*.log" -print
    --> find /tmp ( -type f -a -print ) -o ( -name *.log -a -print )
```

显然，上面第三个语句中的-o -name *.log 是多余的。

以下是这 4 个命令的输出结果以及分析。

```
[root@xuexi tmp]# find /tmp -type f -o -name "*.log"
/tmp/a.log
/tmp/1.txt
/tmp/2.txt
/tmp/3.txt
/tmp/4.txt
/tmp/b.log
```

-type f 评估后得到所有普通文件，但因为只有-o 前面的表达式为假时才会评估-o 后面的表达式，所以将会对非普通文件评估-name *.log，由于本示例中/tmp 下没有非普通文件，所以-name 将得不到结果，于是评估得到普通文件，最后在普通文件的基础上评估-print，即输出普通文件。

```
[root@xuexi tmp]# find /tmp -type f -o -name "*.log" -print
```

-type f 评估后得到所有普通文件，然后对非普通文件评估(-name *.log -a -print)，由于没有非普通文件，所以这个括号评估结果为空。由于-type f 后面没有给 action，所以评估得到普通文件的结果也作废。因此，这个 find 命令什么都不会输出。

如果在/tmp 下创建一个以".log"为后缀的目录，则该 find 命令会输出这个目录。

```
[root@xuexi tmp]# find /tmp -type f -print -o -name "*.log"
/tmp/a.log
/tmp/1.txt
/tmp/2.txt
/tmp/3.txt
/tmp/4.txt
/tmp/b.log
```

-type f 评估后得到所有普通文件，然后直接输出，然后再对未输出的文件评估-name，也就是找出".log"结尾的非普通文件，但即使如此，后面也没有 action，所以这里的-name 是多余的。所以该命令等价于 `find /tmp -type f -print`。

注意，-name 评估的对象是未被-print 输出的文件，而不是未被-type 评估的文件。虽然在本示例中是等价的，但如果-print 前还有-a 条件，如 `find /tmp -type f -a "*.txt" -print -o -name "*.log"`，这里的-name 评估的对象是未被输出的文件，也就是(-type f -a -name .txt)的取反，而不是"-type f"的取反，所以 log 结尾的普通文件也会被-name 评估，但因为没有后续的 action，评估的结果会作废。

```
[root@xuexi tmp]# find /tmp -type f -print -o -name "*.log" -print
/tmp/a.log
/tmp/1.txt
/tmp/2.txt
/tmp/3.txt
/tmp/4.txt
/tmp/b.log
```

-type f 评估后得到所有普通文件，然后直接输出，然后对非普通文件评估(-name *.log -a -print)，由于本处示例中没有以".log"结尾的非普通文件，所以这个括号的评估结果为空。但如果在/tmp 下创建一个以".log"为后缀的目录，则该 find 命令会也输出这个目录。

1.9.4 find 深入用法示例

(1). 指定目录的搜索深度

例如搜素/etc 目录下的".conf"文件，但不搜索任何子目录。

```
find /etc -maxdepth 1 -type f -name "*.conf"
```

注意，"-depth"、"-maxdepth"和"-mindepth"是 find 的 option 表达式，所以它们写在任意位置都会对 test 和 action 产生影响，所以它的位置可随意书写。

(2). 指定目录的处理顺序：-depth

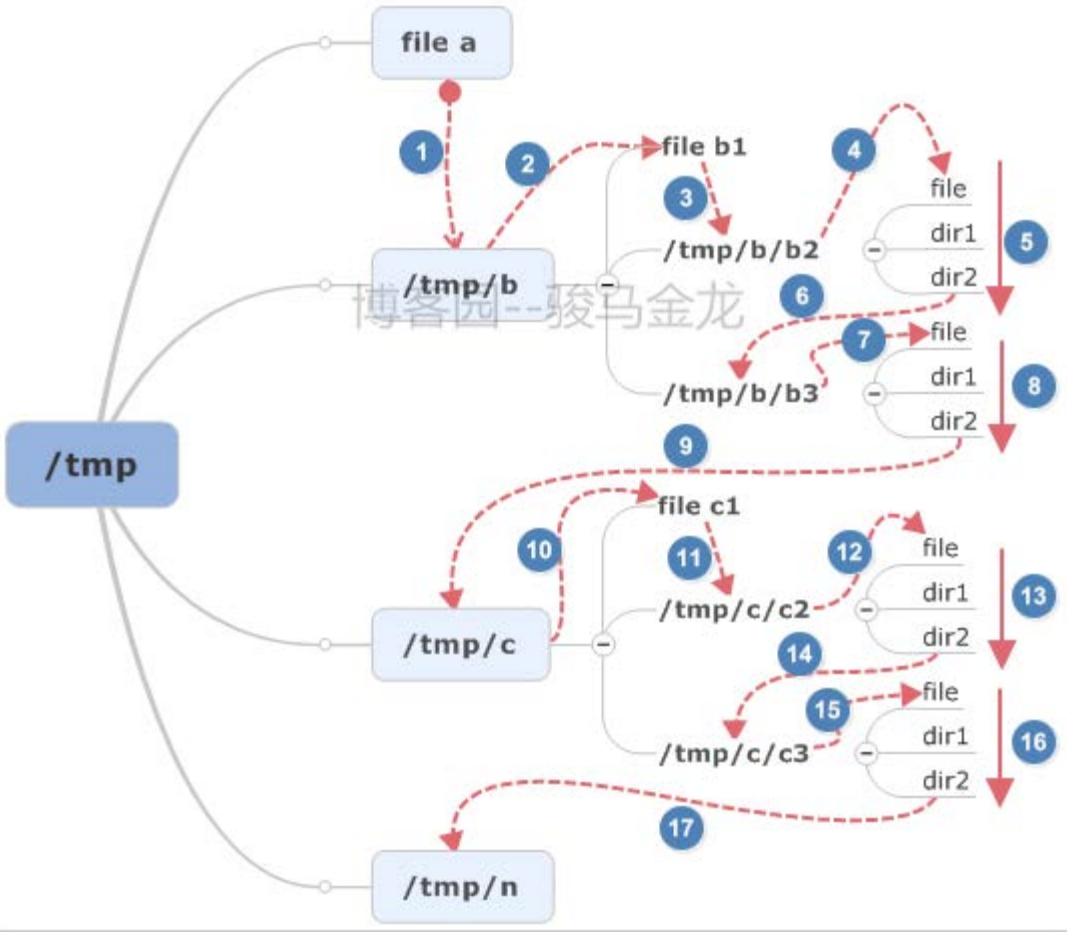
"-depth"选项用于改变 find 使得它先搜索目录中的文件，然后才处理目录本身。默认情况下，find 查找文件和目录的顺序是：假如/tmp 下有 a 文件、b 和 c 两个目录，b 和 c 目录中都有一些文件和目录。

- **先查找 /tmp 本身**
- **再查找 /tmp 下的各个文件，如 a**
- **再按序查找 /tmp 下的第一个目录，如 b**
- **再查找 b 中的文件。**
- **查找 /tmp 下的下一个目录，如 c**
- **查找 c 中的文件。**

如果 b 和 c 目录中还有其他目录，则依照上面的处理规则进行处理。

即/tmp --> /tmp 中的文件 --> /tmp 中第一个目录 --> 第一个目录中的文件 --> 第一个文件中的目录 --> ... --> /tmp 中第二个目录 --> 第二个目录中的文件 --> ... --> /tmp 中的第三个目录 --> .../tmp 中的最后一个目录。

如下面的图所示。



如果指定了-depth，则先处理目录中的文件，再处理目录本身。在 Linux 一切皆文件，子目录也是文件。

例如，下面在/tmp 目录下新建一个目录/tmp/tmp，并在子/tmp 下创建文件 a，目录 b 和 c，其中分别有一些文件。使用-depth 参数运行 find。

```
[root@xuexi tmp]# touch a
[root@xuexi tmp]# mkdir b c
[root@xuexi tmp]# touch ./b/{1..3}.log ./c/{1..3}.sh
[root@xuexi tmp]# find /tmp/tmp -depth
/tmp/tmp/b/2.log
/tmp/tmp/b/1.log
/tmp/tmp/b/3.log
/tmp/tmp/b
/tmp/tmp/c/2.sh
/tmp/tmp/c/3.sh
/tmp/tmp/c/1.sh
/tmp/tmp/c
/tmp/tmp/a
/tmp/tmp
```

上面的 find 工作过程是：先找到/tmp/tmp，准备处理/tmp/tmp 下的文件 a、b、c(目录也是文件)，在这里的处理顺序是 b、c、a(为什么这样的顺序我也不知道)，处理 b 的时候发现它是一个目录，转去处理 b 目录里的文件，处理完 b 中的文件后处理 b 目录，再处理 c，发现是目录，转去处理 c 中的文件，处理完 c 中的文件后处理 a 文件。

需要注意的是，不一定总是先处理目录和目录中的内容。在同一个目录下的文件是有处理顺序的。例如下面在/tmp/tmp 下添加一个.x 隐藏文件，再测试"-depth"。

```
[root@xuexi tmp]# touch .x
[root@xuexi tmp]# find /tmp/tmp -depth
/tmp/tmp/b/2.log
/tmp/tmp/b/1.log
/tmp/tmp/b/3.log
/tmp/tmp/b
/tmp/tmp/.x   #   # 提前于 c 目录被查找到
/tmp/tmp/c/2.sh
/tmp/tmp/c/3.sh
/tmp/tmp/c/1.sh
/tmp/tmp/c
/tmp/tmp/a
/tmp/tmp
```

可以看到.x 隐藏文件是在中途被处理的。

最后看看不加"-depth"的搜索顺序。

```
[root@xuexi tmp]# find /tmp/tmp
/tmp/tmp
/tmp/tmp/b
```

```
/tmp/tmp/b/2.log
/tmp/tmp/b/1.log
/tmp/tmp/b/3.log
/tmp/tmp/.x
/tmp/tmp/c
/tmp/tmp/c/2.sh
/tmp/tmp/c/3.sh
/tmp/tmp/c/1.sh
/tmp/tmp/a
```

(3). 忽略搜索的结果：-prune

一般只考虑忽略目录，不考虑忽略文件，一般忽略文件时会给出警告。再者，忽略某种文件可以使用其他条件来忽略。忽略目录后，Linux 将直接不处理忽略的位置。

-prune 需要放在定义忽略表达式的后面，之所以会如此，请看示例并考虑 true 和 false 进行理解。因为要忽略的是目录，所以一般都只和"-path"配合，而不跟"-name"配合，实际上和-name 配合的时候会给出警告信息。

例如，查找/tmp 中的".log"文件，但排除/tmp 子 abc 目录内的".log"文件。

```
find /tmp -path "/tmp/abc" -prune -o -name "*.log"
/tmp/testdir/a.log
/tmp/abc          # 注意这个被忽略的目录也在结果内
/tmp/a.log
/tmp/axyz.log
/tmp/xyz/axyz.log
```

为何这里使用"-o"操作符而不是"-a"操作符呢？现将上述 find 表达式进行分解。第一个表达式-path "/tmp/abc"可以搜索出/tmp/abc 文件，它可能是目录，也可能是文件，但无论它是什么文件，该表达式返回的总是 true；第二个表达式是 action 类的"-prune"，它和第一个表达式是"expr1 expr2"这样的操作符格式，它等价于 and 逻辑关系，所以-path "/tmp/abc" -prune 表示不进入匹配到的/tmp/abc 目录(若 abc 为文件，则给出警告信息)，由于下一个表达式是使用"-o"连接的，所以到此为止就确定了该目录/tmp/abc，且是不进入的，由于返回 true，-prune 会将结果输出出来。

```
find /tmp -path "/tmp/abc" -prune
/tmp/abc
```

继续，目的是搜索出非 abc 目录下的 log 文件，它的表达式应该是-name "*.log"。但由于前面返回的是 true，如果使用"-a"选项连接前后，则表示后面的表达式的操作对象是/tmp/abc，但前面的逻辑是不进入/tmp/abc，所以将得不到任何结果。

```
find /tmp -path "/tmp/abc" -prune -a -name "*.log"  | wc -l
0
```

而使用"-o"连接，则表示-name "*.log"的操作对象是 path 部分(即/tmp 目录)除了/tmp/abc 的其余文件，它将找出/tmp 下所有的 log 文件，但由于前面明确表示了不进入/tmp/abc 目录，所以就实现了忽略/tmp/abc 目录的作用。

应该注意到了，上面的/tmp/abc 目录也出现在结果中了，但是它并非 log 文件。这是因为-prune 的副作用，find 认为-prune 不是一个完整的 action，它会补上-print。如果想要完完全全的忽略该目录，则可以使用下面的方式，在-prune 之后加上-false 强制使得该段表达式返回 false，这样该段结果就不会被输出。

```
find /tmp -path "/tmp/abc" -prune -false -o -name "*.log"
/tmp/testdir/a.log
/tmp/a.log
/tmp/axyz.log
/tmp/xyz/axyz.log
```

-prune 的一个弱点是不适合通过通配符来忽略目录，因为通配符出来的很可能导致非预期结果。

```
find /tmp -path "/tmp/a*" -prune -o -name "*.log"
/tmp/testdir/a.log
/tmp/a.txt
/tmp/abc
/tmp/about.html
/tmp/a.log
/tmp/axyz.log
/tmp/xyz/axyz.log
```

所以想要忽略多个目录，最好的方法是多次使用-path，但要注意它们的逻辑顺序。例如，搜索/tmp 下所有 log 文件，但忽略/tmp/abc 和/tmp/xyz 两个目录中的 log 文件。

```
find /tmp \( -path '/tmp/abc' -o -path '/tmp/xyz' \)
/tmp/abc
/tmp/xyz
```

所以完整的写法是：

```
find /tmp \( -path /tmp/abc -o -path /tmp/xzy \) -prune -o -name "*.log"
/tmp/testdir/a.log
/tmp/abc
/tmp/a.log
/tmp/axyz.log
/tmp/xyz
```

(4). 不显示待搜索目录本身

在 find 显示结果的时候，如果没有表达式过滤掉目录本身，那么目录本身也会被显示出来，但是很多时候这时不必要的。一般用于只显示一级目录下所有的文件，但不包括目录本身。

```
[root@xuexi ~]# find ~ -maxdepth 1
/root      # 搜索目录本身也被显示出来
/root/.bash_history
/root/.lesshst
/root/install.log
/root/.viminfo
/root/install.log.syslog
/root/.tcshrc
/root/.lftp
/root/.cshrc
/root/raid.sh
/root/.bashrc
/root/anaconda-ks.cfg
/root/.bash_profile
/root/bin
/root/.bash_logout
```

要过滤掉目录本身，方式也很简单，多使用一个匹配表达式即可。

```
[root@xuexi ~]# find ~ -maxdepth 1 ! -name root
/root/.bash_history
/root/.lesshst
/root/install.log
/root/.viminfo
/root/install.log.syslog
/root/.tcshrc
/root/.lftp
/root/.cshrc
/root/raid.sh
/root/.bashrc
/root/anaconda-ks.cfg
/root/.bash_profile
/root/bin
/root/.bash_logout
```

(5). 搜索指定目录下非空文件

"-empty"测试条件用于测试文件是否非空，对于目录而言则是目录为空目录。一般可用于在搜索时，排除空文件或空目录，所以一般和"!"或"-not"一起使用。

例如，搜索/tmp 下非空文件和非空目录。

```
[root@xuexi tmp]# find /tmp/tmp ! -empty
```

第2章 系统用户/组管理

2.1 用户和组的基本概念

用户和组是操作系统中一种身份认证资源。

每个用户都有用户名、用户的唯一编号 uid(user id)、所属组及其默认的 shell，可能还有密码、家目录、附属组、注释信息等。

每个组也有自己的名称、组唯一编号 gid(group id)。一般来说，gid 和 uid 是可以不相同的，但绝大多数都会让它们保持一致，大致属于约定俗成类的概念吧。

组分为主组(primary group)和辅助组(secondary group)两种，用户一定会属于某个主组，也可以同时加入多个辅助组。

在 Linux 中，用户分为 3 类：

(1). 超级管理员

超级管理员是最高权限者，它的 uid=0，默认超级管理员用户名为 root。因为 uid 默认具有唯一性，所以超级管理员默认只能有一个(如何添加额外的超级管理员，见 useradd 命令)，但这一个超级管理员的名称并非一定要是 root。但是没人会去改 root 的名称，在后续非常非常多的程序中，都认为超级管理员名称为 root，这里要是一改，牵一发而动全身。

(2). 系统用户

有时候需要一类具有某些特权但又不需要登录操作系统的用户，这类用户称为系统用户。它们的 uid 范围从 201 到 999(不包括 1000)，有些老版本范围是 1 到 499(centos 6)，出于安全考虑，它们一般不用来登录，所以它们的 shell 一般是/sbin/nologin，而且大多数时候它们是没有家目录的。

(3). 普通用户

普通用户是权限受到限制的用户，默认只能执行/bin、/usr/bin、/usr/local/bin 和自身家目录下的命令。它们的 uid 从 500 开始。尽管普通用户权限收到限制，但是它对自身家目录下的文件是有所有权限的。

超级管理员和其他类型的用户，它们的命令提示符是不一样的。uid=0 的超级管理员，命令提示符是"#"，其他的为"$"。

```
[root@server2 ~]# su longshuai
[longshuai@server2 root]$ echo $PATH
```

默认 root 用户的家目录为/root，其他用户的家目录一般在/home 下以用户名命名的目录中，如 longshuai 这个用户的家目录为/home/longshuai。当然，家目录是可以自定义位置和名称的。

2.2 用户和组管理相关的文件

2.2.1 用户文件/etc/passwd

/etc/passwd 文件里记录的是**操作系统中用户**的信息，这里面记录了几行就表示系统中有几个系统用户。它的格式大致如下：

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
nginx:x:498:499:Nginx web server:/var/lib/nginx:/sbin/nologin
longshuai:x:1000:1000::/home/longshuai:/bin/bash
```

每一行表示一个用户，每一行的格式都是 6 个冒号共 7 列属性，其中有很多用户的某些列属性是留空的。

```
用户名:x:uid:gid:用户注释信息:家目录:使用的 shell 类型
第一列：用户名。注意两个个特殊的用户名，root、nobody
第二列：x。在以前老版本的系统上，第二列是存放用户密码的，但是密码和用户信息放在一起不便于管理(密钥要保证其特殊属性)，所以后来将密码单独放在另一个文件/etc/shadow 中，这里就都写成 x 了。
第三列：uid。
第四列：gid。
第五列：用户注释信息。
第六列：用户家目录。注意 root 用户的家目录为/root。
第七列：用户的默认 shell，虽然叫 shell，但其实可以是任意一个可执行程序或脚本。例如上面的/bin/bash、/sbin/nologin、/sbin/shutdown。
```

用户的默认 shell 表示的是用户登录(如果允许登录)时的环境或执行的命令。例如 shell 为/bin/bash 时，表示登录时就执行/bin/bash 命令进入 bash 环境；shell 为/sbin/nologin 表示该用户不能登录，之所以不能登录不是因为指定了这个特殊的程序，而是由/sbin/nologin 这个程序的功能实现的，假如修改 Linux 的源代码，将/sbin/nologin 这个程序变成可登录，那么 shell 为/sbin/nologin 时也是可以登录的。

2.2.2 密码文件/etc/shadow

/etc/shadow 文件中存放的是用户的密码信息。该文件具有特殊属性，除了超级管理员，任何人都不能直接读取和修改该文件，而用户自身之所以能修改密码，则是因为 passwd 程序的 suid 属性，使得修改密码时临时提升为 root 权限。

该文件的格式大致如下：

```
root:$6$hS4yqJu7WQfGlkOM$Xj/SCS5z4BWSZKNOraNncu6VMuWdUVbDScMYxOgB7mXUj./dXJNOzADAXQUMgOCuWVRyZUu6npPLWoyv8eXPA.::0:99999:7:::
ftp:*:16659:0:99999:7:::
nobody:*:16659:0:99999:7:::
longshuai:$6$8LGe6Eh6$vox9.OF3J9nDOKtOYj2hE9DjfU3iRN.v3up4PbKKGWLOy3k1Up5Obbo7Xii/Uti05hlqhktAf/dZFy2RrGp5W/:17323:0:99999:7:::
```

每一行表示一个用户密码的属性，有 8 个冒号共 9 列属性。该文件更详细的信息看 wiki：https://en.wikipedia.org/wiki/Passwd#Shadow_file。

第一列：用户名。
第二列：加密后的密码。但是这一列是有玄机的，有些特殊的字符表示特殊的意义。
①.该列留空，即"::"，表示该用户没有密码。
②.该列为"!"，即":!:"，表示该用户被锁，被锁将无法登陆，但是可能其他的登录方式是不受限制的，如 ssh key 的方式，su 的方式。
③.该列为"*"，即":*:"，也表示该用户被锁，和"!"效果是一样的。
④.该列以"!"或"!!"开头，则也表示该用户被锁。
⑤.该列为"!!"，即":!!:"，表示该用户从来没设置过密码。
⑥.如果格式为"idsalt$hashed"，则表示该用户密码正常。其中$id$的 id 表示密码的加密算法，$1$表示使用 MD5 算法，$2a$表示使用 Blowfish 算法，"$2y$"是另一算法长度的 Blowfish,"$5$"表示 SHA-256 算法，而"$6$"表示 SHA-512 算法，可见上面的结果中都是使用 sha-512 算法的。5和6这两种算法的破解难度远高于 MD5。$salt$是加密时使用的 salt，$hashed 才是真正的密码部分。
第三列：从 1970 年 1 月 1 日到上次密码修改经过的时间(天数)。通过计算现在离 1970 年 1 月 1 日的天数减去这个值，结果就是上次修改密码到现在已经经过了多少天，即现在的密码已经使用了多少天。
第四列：密码最少使用期限(天数)。省略或者 0 表示不设置期限。例如，刚修改完密码又想修改，可以限制多久才能再次修改
第五列：密码最大使用期限(天数)。超过了它不一定密码就失效，可能下一个字段设置了过期后的宽限天数。设置为空时将永不过期，后面设置的提醒和警告将失效。root 等一些用户的已经默认设置为了 99999，表示永不过期。如果值设置小于最短使用期限，用户将不能修改密码。
第六列：密码过期前多少天就开始提醒用户密码将要过期。空或 0 将不提醒。
第七列：密码过期后宽限的天数，在宽限时间内用户无法使用原密码登录，必须改密码或者联系管理员。设置为空表示没有强制的宽限时间，可以过期后的任意时间内修改密码。
第八列：帐号过期时间。从 1970 年 1 月 1 日开始计算天数。设置为空帐号将永不过期，不能设置为 0。不同于密码过期，密码过期后账户还有效，改密码后还能登录；帐号过期后帐号失效，修改密码重设密码都无法使用该帐号。
第九列：保留字段。

2.2.3 组文件/etc/group 和/etc/gshadow

大致知道有这么两个文件即可，至于文件中的内容无需关注。

/etc/group 包含了组信息。每行一个组，每一行 3 个冒号共 4 列属性。

```
root:x:0:
longshuai:x:500:
xiaofang:x:501:zhangsan,lisi
```

第一列：组名。
第二列：占位符。
第三列：gid。
第四列：该组下的 user 列表，这些 user 成员以该组做为辅助组，多个成员使用逗号隔开。

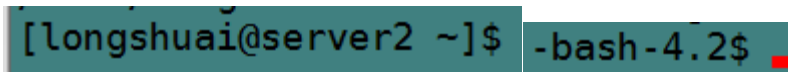
/etc/gshadow 包含了组密码信息

2.2.4 骨架目录/etc/skel

骨架目录中的文件是每次新建用户时，都会复制到新用户家目录里的文件。默认只有 3 个环境配置文件，可以修改这里面的内容，或者添加几个文件在骨架目录中，以后新建用户时就会自动获取到这些环境和文件。

```
ll -A /etc/skel
total 12
-rw-r--r--. 1 root root  18 Oct 16  2014 .bash_logout
-rw-r--r--. 1 root root 176 Oct 16  2014 .bash_profile
-rw-r--r--. 1 root root 124 Oct 16  2014 .bashrc
```

删除家目录下这些文件，会导致某些设置出现问题。例如删除".bashrc"这个文件，会导致提示符变异的问题，如下右图。



要解决这个问题，只需拷贝一个正常的.bashrc 文件到其家目录中即可。一般还会修改该文件的所有者和权限。

2.2.5 /etc/login.defs

设置用户帐号限制的文件。该文件里的配置对 root 用户无效。

如果/etc/shadow 文件里有相同的选项，则以/etc/shadow 里的设置为准，也就是说/etc/shadow 的配置优先级高于/etc/login.defs。

该文件有很多配置项，文件的默认内容只给出了一小部分，若想知道全部的配置项以及配个配置项的详细说明，可以"man 5 login.defs"查看。

```
[root@xuexi ~]# less /etc/login.defs
#QMAIL_DIR      Maildir              # QMAIL_DIR 是 Qmail 邮件的目录，所以可以不设置它
MAIL_DIR        /var/spool/mail      # 默认邮件根目录，即信箱
#MAIL_FILE      .mail                # mail 文件的格式是.mail

# Password aging controls:
PASS_MAX_DAYS   99999         # 密码最大有效期(天)
PASS_MIN_DAYS   0             # 两次密码修改之间最小时间间隔
PASS_MIN_LEN    5             # 密码最短长度
PASS_WARN_AGE   7             # 密码过期前给警告信息的时间

# 控制 useradd 创建用户时自动选择的 uid 范围
# Min/max values for automatic uid selection in useradd
UID_MIN                  1000
UID_MAX                 60000
# System accounts
SYS_UID_MIN               201
SYS_UID_MAX               999

# 控制 groupadd 创建组时自动选择的 gid 范围
# Min/max values for automatic gid selection in groupadd
GID_MIN                  1000
GID_MAX                 60000
# System accounts
SYS_GID_MIN               201
SYS_GID_MAX               999

# 设置此项后，在删除用户时，将自动删除用户拥有的 at/cron/print 等 job
#USERDEL_CMD    /usr/sbin/userdel_local

# 控制 useradd 添加用户时是否默认创建家目录，useradd -m 选项会覆盖此处设置
CREATE_HOME     yes

# 设置创建家目录时的 umask 值，若不指定则默认为 022
UMASK           077

# 设置此项表示当组中没有成员时自动删除该组
# 且 useradd 是否同时创建同用户名的主组。
# (该文件中并没有此项说明，来自于 man useradd 中-g 选项的说明)
USERGROUPS_ENAB yes

# 设置用户和组密码的加密算法
ENCRYPT_METHOD SHA512
```

注意，/etc/login.defs 中的设置控制的是 shadow-utils 包中的组件，也就是说，该组件中的工具执行操作时会读取该文件中的配置。该组件中包含下面的程序：

```
/usr/bin/gpasswd      : administer /etc/group and /etc/gshadow
/usr/bin/newgrp       : log in to a new group，可用来修改 gid，即使是正在登陆的会话也可修改
/usr/bin/sg           : execute command as different group ID
/usr/sbin/groupadd    : 添加组
/usr/sbin/groupdel    : 删除组
/usr/sbin/groupmems   : 管理当前用户的主组中的成员，root 用户则可以指定要管理的组
/usr/sbin/groupmod    : modify a group definition on the system
/usr/sbin/grpck       : verify integrity of group files
/usr/sbin/grpconv     : 无视它
/usr/sbin/grpunconv   : 无视它
/usr/sbin/pwconv      : 无视它
/usr/sbin/pwunconv    : 无视它
/usr/sbin/adduser     : 是 useradd 的一个软链接，添加用户
/usr/sbin/chpasswd    : update passwords in batch mode
```

```
/usr/sbin/newusers    : update and create new users in batch
/usr/sbin/pwck        : verify integrity of passsword files
/usr/sbin/useradd     : 添加用户
/usr/sbin/userdel     : 删除用户
/usr/sbin/usermod     : 重定义用户信息
/usr/sbin/vigr        : edit the group and shadow-group file
/usr/sbin/vipw        : edit the password and shadow-password file
/usr/bin/lastlog      : 输出所有用户或给定用户最近登录信息
```

2.2.6 /etc/default/useradd

创建用户时的默认配置。useradd -D 修改的就是此文件。

```
[root@xuexi ~]# cat /etc/default/useradd
# useradd defaults file
GROUP=100        # 在useradd使用-N或/etc/login.defs中USERGROUPS_ENAB=no时表示创建
                 # 用户时不创建同用户名的主组(primary group)，此时新建的用户将默认以
                 # 此组为主组，网上关于该设置的很多说明都是错的，具体可看man useradd
                 # 的-g选项或useradd -D的-g选项
HOME=/home       # 把用户的家目录建在/home中
INACTIVE=-1      # 是否启用帐号过期设置(是帐号过期不是密码过期)，-1表示不启用
EXPIRE=          # 帐号过期时间，不设置表示不启用
SHELL=/bin/bash # 新建用户默认的shell类型
SKEL=/etc/skel  # 指定骨架目录，前文的/etc/skel就在这里
CREATE_MAIL_SPOOL=yes  # 是否创建用户mail缓冲
```

man useradd 的 useradd -D 选项介绍部分说明了这些项的意义。

2.3 用户和组管理命令

2.3.1 useradd 和 adduser

adduser 是 useradd 的一个软链接。

```
useradd [options] login_name
选项说明：
-b：指定家目录的basedir，默认为/home目录
-d：指定用户家目录，不写时默认为/home/user_name
-m：要创建家目录时，若家目录不存在则自动创建，若不指定该项且/etc/login.defs中的CREATE_HOME未启用时将不会创建家目录
-M：显式指明不要创建家目录，会覆盖/etc/login.defs中的CREATE_HOME设置

-g：指定用户主组，要求组已存在
-G：指定用户的辅助组，多个组以逗号分隔
-N：明确指明不要创建和用户名同名的组名
-U：明确指明要创建一个和用户名同名的组，并将用户加入到此组中

-o：允许创建一个重复UID的用户，只有和-u选项同时使用时才生效
-r：创建一个系统用户。useradd命令不会为此选项的系统用户创建家目录，除非明确使用-m选项
-s：指定用户登录的shell，默认留空。此时将选择/etc/default/useradd中的SHELL变量设置
-u：指定用户uid，默认uid必须唯一，除非使用了-o选项
-c：用户的注释信息

-k：指定骨架目录(skeleton)
-K：修改/etc/login.defs文件中有关于用户的配置项，不能修改组相关的配置。设置方式为KEY=VALUE，如-K UID_MIN=100
-D：修改useradd创建用户时的默认选项，就修改/etc/default/useradd文件
-e：帐户过期时间，格式为"YYYY-MM-DD"
-f：密码过期后，该账号还能存活多久才被禁用，设置为0表示密码过期立即禁用帐户，设置为-1表示禁用此功能
-l：不要将用户的信息写入到lastlog和faillog文件中。默认情况下，用户信息会写入到这两个文件中

useradd -D [options]
修改/etc/default/useradd文件
选项说明：不加任何选项时会列出默认属性
-b, --base-dir BASE_DIR
-e, --expiredate EXPIRE_DATE
-f, --inactive INACTIVE
-g, --gid GROUP
-s, --shell SHELL
```

示例：

```
[root@xuexi ~]# useradd -D -e "2016-08-20"    # 设置用户 2016-08-20 过期
[root@xuexi ~]# useradd -D
GROUP=100
HOME=/home
INACTIVE=-1
EXPIRE=2016-08-20
SHELL=/bin/bash
SKEL=/etc/skel
CREATE_MAIL_SPOOL=yes
[root@xuexi ~]# cat /etc/default/useradd
# useradd defaults file
GROUP=100
HOME=/home
INACTIVE=-1
EXPIRE=2016-08-20
SHELL=/bin/bash
SKEL=/etc/skel
CREATE_MAIL_SPOOL=yes
```

useradd 创建用户时，默认会自动创建一个和用户名相同的用户组，这是/etc/login.defs 中的 USERGROUP_ENAB 变量控制的。

useradd 创建普通用户时，不加任何和家目录相关的选项时，是否创建家目录是由/etc/login.defs 中的 CREATE_HOME 变量控制的。

2.3.2 批量创建用户 newusers

newusers 用于批量创建或修改已有用户信息。在创建用户时，它会读取/etc/login.defs 文件中的配置项。

```
newusers [options] [file]
```

newusers 命令从 file 中或标准输入中读取要创建或修改用户的信息，文件中每行格式都一样，一行代表一个用户。格式如下：

```
pw_name:pw_passwd:pw_uid:pw_gid:pw_gecos:pw_dir:pw_shell
```

各列的意义如下：

```
pw_name：用户名，若不存在则新创建，否则修改已存在用户的信息
pw_passwd：用户密码，该项使用明文密码，在修改或创建用户时会按照指定的算法自动对其进行加密转换
pw_uid：指定 uid，留空则自动选择 uid。如果该项为已存在的用户名，则使用该用户的 uid，但不建议这么做，uid 应尽量保证唯一性
pw_gid：用户主组的 gid 或组名。若给定组不存在，则自动创建组。若留空，则创建同用户名的组，gid 将自动选择
pw_gecos：用户注释信息
pw_dir：指定用户家目录，若不存在则自动创建。留空则不创建。注意，newusers 命令不会递归创建父目录，父目录不存在时将会给出信息，但 newusers 命令仍会继续执行以完成创建剩下的用户，所以这些错误的用户家目录需要手动去创建。
pw_shell：指定用户的默认 shell
```

```
newusers [options] [file]
选项说明：
-c：指定加密方法，可选 DES,MD5,NONE,SHA256 和 SHA512
-r：创建一个系统用户
```

newusers 首先尝试创建或修改所有指定的用户，然后将信息写入到 user 和 group 的文件中。如果尝试创建或修改用户过程中发生错误，则所有动作都将回滚，但如果在写入过程中发生错误，则写入成功的不会回滚，这将可能导致文件的不一致性。要检查用户、组文件的一致性，可以使用 showdow-utils 包提供的 grpck 和 pwck 命令。

示例：

```
cat /tmp/userfile
zhangsan:123456:2000:2000::/home/zhangsan:/bin/bash
lisi:123456:::::/bin/bash
newusers -c SHA512 /tmp/userfile
tail -2 /etc/passwd
zhangsan:x:2000:2000::/home/zhangsan:/bin/bash
lisi:x:2001:2001:::/bin/bash
tail -2 /etc/shadow
zhangsan:$6$aI1Mk/krF$xNOTFOIRibrb/mYngJ/sV3M7g4zOxqOh8CWyD1IOuwmr5qNTzsmwauRFvCpfLtvtiJYZ/5bil.XfJMNB.sqDY1:17323:0:99999:7:::
lisi:$6$bngXo/V6wWW$.TlQCJtEm9krBXOOiep/iahS59a/BwVYcSc8F91AnMGF55K6W5YoUZ2nK6WkMta3p7sihkxHm/AuNrrJ6hqNn1:17323:0:99999:7:::
```

2.3.3 groupadd

创建一个新组。

```
groupadd [options] group
```

```
选项说明：
-f：如果要创建的组已经存在，默认会错误退出，使用该选项则强制创建且以正确状态退出，只不过 gid 可能会不受控制。
-g：指定 gid，默认 gid 必须唯一，除非使用了-o 选项。
-K：修改/etc/login.defs 中关于组相关的配置项。配置方式为 KEY=VALUE，例如-K GID_MIN=100 -K GID_MAX=499
-o：允许创建一个非唯一 gid 的组
-r：创建系统组
```

2.3.4 修改密码 passwd

修改密码的工具。默认 passwd 命令不允许为用户创建空密码。

passwd 修改密码前会通过 pam 认证用户，pam 配置文件中与此相关的设置项如下：

```
        passwd password requisite pam_cracklib.so retry=3
        passwd password required pam_unix.so use_authtok
```

命令的用法如下：

```
passwd options [username]
选项说明：
-l：锁定指定用户的密码，在/etc/shadow 的密码列加上前缀"!"或"!!"。这种锁定不是完全锁定，使用 ssh 公钥还是能登录。要完全锁定，使用 chage -E 0 来设置帐户过期。
-u：解锁-l 锁定的密码，解锁的方式是将/etc/shadow 的密码列的前缀"!"或"!!"移除掉。但不能移除只有"!"或"!!"的项。
--stdin：从标准输入中读取密码
-d：删除用户密码，将/etc/shadow 的密码列设置为空
-f：指定强制操作
-e：强制密码过期，下次登录将强制要求修改密码
-n：密码最小使用天数
-x：最大密码使用天数
-w：过期前几天开始提示用户密码将要过期
-i：设置密码过期后多少天，用户才过期。用户过期将被禁用，修改密码也无法登陆。
```

2.3.5 批量修改密码 chpasswd

以批处理模式从标准输入中获取提供的用户和密码来修改用户密码，可以一次修改多个用户密码。也就是说不用交互。适用于一次性创建了多个用户时为他们提供密码。

```
chpasswd [-e -c] "user:passwd"
-c：指定加密算法，可选的算法有 DES,MD5,NONE,SHA256 和 SHA512
user:passwd 为用户密码对，其中默认 passwd 是明文密码，可以指定多对，每行一个用户密码对。前提是用户是已存在的。
-e：passwd 默认使用的是明文密码，如果要使用密文，则使用-e 选项。参见 man chpasswd
```

chpasswd 会读取/etc/login.defs 中的相关配置，修改成功后会将密码信息写入到密码文件中。

该命令的修改密码的处理方式是先在内存中修改，如果所有用户的密码都能设置成功，然后才写入到磁盘密码文件中。在内存中修改过程中出错，则所有修改都回滚，但若在写入密码文件过程中出错，则成功的不会回滚。

示例：

修改单个用户密码。

```
echo "user1:123456" | chpasswd -c SHA512
```

修改多个用户密码，则提供的每个用户对都要分行。

```
echo  -e 'usertest:123456\nusertest2:123456' | chpasswd
```

更方便的是写入到文件中，每行一个用户密码对。

```
cat /tmp/passwdfile
zhangsan:123456
lisi:123456
chapasswd -c SHA512 </tmp/passwdfile
```

2.3.6 chage

chage 命令主要修改或查看和密码时间相关的内容。具体的看 man 文档，可能用到的两个选项如下：

```
-l：列出指定用户密码相关信息
-E：指定帐户(不是密码)过期时间，所以是强锁定，如果指定为 0，则立即过期，即直接锁定该用户
chage -l zhangsan
Last password change                                    : Jun 06, 2017
Password expires                                        : never
Password inactive                                       : never
Account expires                                         : never
```

```
Minimum number of days between password change      : 0
Maximum number of days between password change      : 99999
Number of days of warning before password expires   : 7
[root@server2 ~]# chage -E 0 zhangsan
[root@server2 ~]# chage -l zhangsan
Last password change                                : Jun 06, 2017
Password expires                                    : never
Password inactive                                   : never
Account expires                                     : Jan 01, 1970
Minimum number of days between password change      : 0
Maximum number of days between password change      : 99999
Number of days of warning before password expires   : 7
```

2.3.7 删除用户和组

userdel 命令用于删除用户。

```
userdel [options] login_name
-r：递归删除家目录，默认不删除家目录。
-f：强制删除用户，即使这个用户正处于登录状态。同时也会强制删除家目录。
```

工作中一般不直接删除家目录，即不用-r，可以 vi /etc/passwd，将不需要的用户直接注释掉。

groupdel 命令删除组。如果要删除的组是某用户的主组，需要先删除主组中的用户。

2.3.8 usermod

修改帐户属性信息。必须要确保在执行该命令的时候，待修改的用户没有在执行进程。

```
usermod [options] login
```

选项说明：

```
-l：修改用户名，仅仅只是改用户名，其他的一切都不会改动(uid、家目录等)
-u：新的 uid，新的 uid 必须唯一，除非同时使用了-o 选项
-g：修改用户主组，可以是以 gid 或组名。对于那些以旧组为所属组的文件(除原家目录)，需要重新手动修改其所属组
-m：移动家目录内容到新的位置，该选项只在和-d 选项一起使用时才生效
-d：修改用户的家目录位置，若不存在则自动创建。默认旧的家目录不会删除
    如果同时指定了-m 选项，则旧的家目录中的内容会移到新家目录
    如果当前用户家目录不存在或没有家目录，则也不会创建新的家目录
-o：允许用户使用非唯一的 UID
-s：修改用的 shell，留空则选择默认 shell
-c：修改用户注释信息

-a：将用户以追加的方式加入到辅助组中，只能和-G 选项一起使用
-G：将用户加入指定的辅助组中，若此处未列出某组，而此前该用户又是该组成员，则会删除该组中此成员

-L：锁定用户的密码，将在/etc/shadow 的密码列加上前缀"!"或"!!"
-U：解锁用户的密码，解锁的方式是移除 shadow 文件密码列的前缀"!"或"!!"
-e：帐户过期时间，时间格式为"YYYY-MM-DD"，如果给一个空的参数，则立即禁用该帐户
-f：密码过期后多少天，帐户才过期被禁用，0 表示密码过期帐户立即禁用，-1 表示禁用该功能
```

同样，还有 groupmod 修改组信息，用法非常简单，几乎也用不上，不多说了。

2.3.9 vipw 和 vigr

vipw 和 vigr 是编辑用户和组文件的工具，vipw 可以修改/etc/passwd 和/etc/shadow，vigr 可以修改/etc/group 和/etc/gshadow，用这两个工具比较安全，在修改的时候会检查文件的一致性。

删除用户出错时，提示用户正在被进程占用。可以使用 vi 编辑/etc/paswd 和/etc/shadow 文件将该用户对应的行删除掉。也可以使用 vipw 和 vipw -s 来分别编辑/etc/paswd 和/etc/shadow 文件。它们的作用是一样的。

2.3.10 手动创建用户

手动创建用户的全过程：需要管理员权限。

- 在/etc/group 中添加用户所属组的相关信息。如果用户还有辅助组则在对应组中加入该用户作为成员。
- 在/etc/passwd 和/etc/shadow 中添加用户相关信息。此时指定的家目录还不存在，密码不存在，所以/etc/shadow 的密码位使用"!!"代替。
- 创建家目录，并复制骨架目录中的文件到家目录中。

```
mkdir /home/user_name
```

```
cp -r /etc/skel /home/user_name。
```

- 修改家目录及子目录的所有者和属组。

```
chown -R user_name:user_name /home/user_name
```

- 修改家目录及子目录的权限。例如设置组和其他用户无任何权限但所有者有

```
chmod -R 700 /home/user_name
```

到此为止，用户已经创建完成了，只是没有密码，所以只能 su，不能登录。

- 生成密码。

  → 使用 openssl passwd 生成密码。但 openssl passwd 生成的密码只能是 MD5 算法的，很容易被破解

```
# 生成使用 md5 算法的密码，然后将其复制到/etc/shadow 对应的密码位
# 其中-1 是指 md5，-salt '12345678'是使用 8 位字符创建密码的杂项
shell> openssl passwd -1 -salt '12345678' '123456'
```

其中-1 是指 md5，-salt '12345678'是使用 8 位的字符创建密码的杂项，8 位字符任意指定。

  → 直接使用 passwd 命令创建密码

- 测试手动创建的用户是否可以正确登录。

以下是全过程。

```
mkdir /tmp/12;cp /etc/group /etc/passwd /etc/shadow /tmp/12/    # 备份这些文件
echo "userX:x:666" >> /etc/group
echo "userX:x:666:666::/home/userX:/bin/bash" >> /etc/passwd
echo 'userX:!!:17121:0:99999::::' >> /etc/shadow
cp -r /etc/skel /home/userX
chown -R userX:userX /home/userX
chmod -R go= /home/userX
passwd --stdin userX <<< '123456'
```

测试使用 userX 是否可以登录。

如果是使用 openssl passwd 创建的密码。那么使用下面的方法将这部分密码替换到/etc/shadow 中。

```
field=$(tail -1 /etc/shadow | cut -d":" -f2)
password=$(openssl passwd -1 -salt 'abcdefg' 123456)
sed -i '$s%'$field'%'$password'%' /etc/shadow
```

2.4 其他用户相关命令

2.4.1 finger 查看用户信息

从 CentOS 6 版本开始就没有该命令了，要先安装。

```
yum -y install finger
useradd zhangsan
finger zhangsan
Login: zhangsan                         Name:
Directory: /home/zhangsan               Shell: /bin/bash
Never logged in.
No mail.
No Plan.
```

2.4.2 id

```
id username
-u：得到 uid
-n：得到用户名而不是 uid
-z：无任何空白字符输出模式，不能在默认的格式下使用。
```

示例：

```
id root
uid=0(root) gid=0(root) groups=0(root)
id wangwu
uid=500(wangwu) gid=500(wangwu) groups=500(wangwu)
id -u wangwu
500
id -u -z wangwu
```

```
2002[root@server2 ~]#
```

2.4.3 users

查看当前正在登陆的用户名。

2.4.4 last

查看最近登录的用户列表，其实 last 查看的是/var/log/wtmp 文件。

```
-n 显示行数：列出最近几次登录的用户
[root@xuexi ~]# last -4
root     pts/0        192.168.100.1    Wed Mar 30 15:16   still logged in
root     pts/1        192.168.100.1    Wed Mar 30 14:21 - 14:21  (00:00)
root     pts/1        192.168.100.1    Wed Mar 30 14:04 - 14:10  (00:06)
root     pts/0        192.168.100.1    Wed Mar 30 13:12 - 15:16  (02:04)

wtmp begins Thu Feb 18 20:59:39 2016
```

2.4.5 lastb

查看谁尝试登陆过但没有登录成功的。即能够审核和查看谁曾经不断的登录，可能那就是黑客。

```
-n:只列出最近的 n 个尝试对象。
```

2.4.6 who 和 w

都是查看谁登录过，并干了什么事

w 查看的信息比 who 多。

```
who
root     tty1         2017-06-07 00:49
root     pts/0        2017-06-07 02:06 (192.168.100.1)
w
 08:26:38 up 18:48, 13:48, 2 users,  load average: 0.00, 0.01, 0.05
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
root     tty1                      00:49    7:36m  0.24s  0.24s -bash
root     pts/0    192.168.100.1    02:06    6.00s  0.97s  0.02s w
```

其中 w 的第一行，分别表示当前时间，已开机时长，当前在线用户，过去 1、5、15 分钟的平均负载率。这一行和 uptime 命令获取的信息是完全一致的。

2.4.7 lastlog

可以查看登录的来源 IP

```
-u 指定查看用户
lastlog|head -n 10
Username         Port     From             Latest
root             pts/0    192.168.100.1    Wed Mar 30 15:16:25 +0800 2016
bin                                        **Never logged in**
daemon                                     **Never logged in**
adm                                        **Never logged in**
lp                                         **Never logged in**
sync                                       **Never logged in**
shutdown                                   **Never logged in**
halt                                       **Never logged in**
mail                                       **Never logged in**
```

2.5 su

切换用户或以指定用户运行命令。

使用 su 可以指定运行命令的身份(user/group/uid/gid)。

为了向后兼容，su 默认不会改变当前目录，且仅设置 HOME 和 SHELL 这两个环境变量(若目标用户非 root，则还设置 USER 和 LOGNAME 环境变量)。推荐使用--login 选项(即"-"选项)避免环境变量混乱。

```
su [options...] [-] [user [args...]]
选项说明：
-c command：使用-c 选项传递要指定的命令到 shell 上执行。使用-c 执行命令会为每个 su 都分配新的会话环境
-, -l, --login：启动 shell 作为登录的 shell，模拟真正的登录环境。它会做下面几件事：
```

```
                    1.清除除了 TERM 外的所有环境变量
                    2.初始化 HOME,SHELL,USER,LOGNAME,PATH 环境变量
                    3.进入目标用户的家目录
                    4.设置 argv[0]为"-"以便设置 shell 作为登录的 shell
              使用--login 的 su 是交互式登录。不使用--login 的 su 是非交互式登录(除不带任何参数的 su 外)
-m, -p, --preserve-environment：保留整个环境变量(不会重新设置 HOME,SHELL,USER 和 LOGNAME),
                                保留环境的方法是新用户 shell 上执行原用户的各配置文件，如~/.bashrc。
                                当设置了--login 时，将忽略该选项
-s SHELL：运行指定的 shell 而非默认 shell，选择 shell 的顺序优先级如下：
                  1.--shell 指定的 shell
                  2.如果使用了--preserve-environment，选择 SHELL 环境变量的 shell
                  3.选项目标用户在 passwd 文件中指定的 shell
                  4./bin/sh
```

注意：

(1). 若 su 没有给定任何参数，将默认以 root 身份运行交互式的 shell(交互式，所以需要输入密码)，即切换到 root 用户，但只改变 HOME 和 SHELL 环境变量。

(2). su - username 是交互式登录，要求密码，会重置整个环境变量，它实际上是在模拟真实的登录环境。

(3). su username 是非交互登录，不会重置除 HOME/SHELL 外的环境变量。

例如：用户 wangwu 家目录为/home/wangwu，其 shell 为/bin/csh。

```
head -1 /etc/passwd ; tail -1 /etc/passwd
root:x:0:0:root:/root:/bin/bash
wangwu:x:2002:2002::/home/wangwu:/bin/csh
```

首先 su 到 wangwu 上，再执行一个完全不带参数的 su。

```
su - wangwu       # 使用 su - username 后，以登录 shell 的方式模拟登录，会重新设置各环境变量。su - username 是交互式登录
env | egrep -i '^home|^shell|^path|^logname|^user'
HOME=/home/wangwu
SHELL=/bin/csh
USER=wangwu
LOGNAME=wangwu
PATH=/usr/lib64/qt-3.3/bin:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin
PWD=/home/wangwu
su       # 不带任何参数的 su，是交互式登录切换回 root，但只会改变 HOME 和 SHELL 环境变量
env | egrep -i '^home|^shell|^path|^logname|^user|^pwd'
SHELL=/bin/bash
USER=wangwu
PATH=/usr/lib64/qt-3.3/bin:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin
PWD=/home/wangwu
HOME=/root
LOGNAME=wangwu
su  -            #  su - 的方式切换回 root
Password:
env | egrep -i '^home|^shell|^path|^logname|^user|^pwd'
SHELL=/bin/bash
USER=root
PATH=/usr/lib64/qt-3.3/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin
PWD=/root
HOME=/root
LOGNAME=root
su wangwu       # 再直接 su username，它只会重置 SHELL 和 HOME 两个环境变量，其他的环境变量是保持不变的
env | egrep -i '^home|^shell|^path|^logname|^user|^pwd'
SHELL=/bin/csh
USER=wangwu
PATH=/usr/lib64/qt-3.3/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin
PWD=/root
HOME=/home/wangwu
LOGNAME=wangwu
```

在某些环境下或脚本中，可能需要临时切换身份执行命令，注意这时候的环境变量是否会改变，否则很可能报错提示命令找不到。

2.6 sudo

sudo 可以让一个用户以某个身份(如 root 或其他用户)执行某些命令，它隐含的执行方式是切换到指定用户再执行命令，因为涉及到了用户的切换，所以环境变量是否重置是需要设置的。

sudo 支持插件实现安全策略。默认的安全策略插件是 sudoers，它是通过/etc/sudoers 或 LDAP 来配置的。

安全策略是控制用户使用 sudo 命令时具有什么权限，但要注意，安全策略可能需要用户进行身份认证，如密码认证的机制或其他认证机制，如果开启了认证要求，则在指定时间内未完成认证时 sudo 会退出，默认超时时间为 5 分钟。

安全策略支持对认证进行缓存，使得在一定时间内该用户无需再次认证就可以执行 sudo 命令，默认缓存时间为 5 分钟，sudo -v 可以更新认证缓存。

sudo 支持日志审核，可以记录下成功或失败的 sudo。

2.6.1 /etc/sudoers 文件

该文件里主要配置 sudo 命令时指定的用户和对应的权限。

```
$ visudo     # 选取的是部分行
## hostname or IP addresses instead.   # 主机别名 Host_Alias
# Host_Alias     FILESERVERS = fs1, fs2
# Host_Alias     MAILSERVERS = smtp, smtp2

## User Aliases           # 用户别名 User_Alias
# User_Alias ADMINS = jsmith, mikem

## Command Aliases        # 命令别名
# Cmnd_Alias SERVICES = /sbin/service, /sbin/chkconfig
# Cmnd_Alias LOCATE = /usr/bin/updatedb

root    ALL=(ALL)       ALL   # sudo 权限的配置
# %sys ALL = NETWORKING, SOFTWARE, SERVICES, STORAGE, DELEGATING, PROCESSES, LOCATE, DRIVERS
## Allows people in group wheel to run all commands
# %wheel        ALL=(ALL)       ALL
## Same thing without a password
# %wheel        ALL=(ALL)       NOPASSWD: ALL
```

在这个文件里，主要有别名(用户别名，主机别名，命令别名)的配置和 sudo 权限的配置。

安全策略配置格式为：

```
用户名  主机名=(可切换到的用户身份)  权限和命令
 ①       ②          ③                ④
```

①用户名：可以用组，只需在组名前加个百分号%表示。

②主机名：表示该用户可以在哪些主机上运行 sudo，可以用 hostname 也可以用 ip 指定。

③可切换的用户身份，即指定执行命令的用户，也可以用组。

④权限和命令：允许执行和不允许执行的命令(多个命令间用逗号分隔)和特殊权限，命令可以带其选项及参数。命令要写绝对路径。不允许执行的命令需要在命令前加上"!"来表示。可以使用标签，如 NOPASSWD 标签表示切换或以指定用户执行该标签后的命令时不需要输入密码。一行写不下时可使用"\"续行。

标签使用方法：

```
NOPASSWD:/usr/sbin/useradd,PASSWD:/usr/sbin/userdel
```

它表示 useradd 命令不需要输入密码，而 userdel 需要输入密码。

对于别名，相当于用户对于用户组。权限配置处都可以使用别名，即①②③④处都能使用别名来配置。

例如，主机别名里设置多个主机，以后在②位置处直接使用主机别名。

```
FILESERVERS = fs1, fs2
```

以下是某设置示例：

```
DEFAULT=/bin/*,/sbin/ldconfig,/sbin/ifconfig,/usr/sbin/useradd,/usr/sbin/userdel,/bin/rpm,/usr/bin/yum,/sbin/service,/sbin/chkconfig,sudoedit /etc/rc.local,sudoedit /etc/hosts,sudoedit /etc/ld.so.conf,/bin/mount,sudoedit /etc/exports,/usr/bin/passwd [!-]*,!/usr/bin/passwd root,/bin/su - [!-]*,!/bin/su - root,!/bin/su root, /bin/bash, /usr/sbin/dmidecode, /usr/sbin/lsof, /usr/bin/du, /usr/bin/python, /usr/sbin/xm,sudoedit /etc/profile,sudoedit /etc/bashrc,/usr/bin/make,sudoedit /etc/security/limits.conf,/etc/init.d/*,/usr/bin/ruby

ABC ALL=(ALL)NOPASSWD:DEFAULT
```

其中上面的"/usr/bin/passwd [!-]*"表示允许修改加参数的密码。"/bin/su - [!-]*"表示允许"su -"到某用户下，但必须给参数。

2.6.2 sudo 和 sudoedit 命令

当 sudo 执行指定的 command 时，它会调用 fork 函数，并设置命令的执行环境(如某些环境变量)，然后在子进程中执行 command，sudo 的主进程等待命令执行完毕，然后传递命令的退出状态码给安全策略并退出。

sudoedit 等价于 sudo -e，它是以 sudo 的方式执行文件编辑动作。

```
sudo [options] [command]
选项说明：
-b              ：(background)该选项告诉 sudo 在后台执行指定的命令。
                  注意，如果使用该选项，将无法使用任务计划(job)来控制维护这些后台进程，
                  需要交互的命令应该考虑是否真的要后台，因为可能会失败
-l[l] [command]：当单独使用-l 选项时，将列出(list)用户可执行和被禁止的命令。
                  当配合 command 时，且该 command 是被允许执行的命令，将列出命令的全路径及该命令参数。
                  如果 command 是不被允许执行的，则 sudo 直接以状态码-1 退出。
                  可以指定多个字母"l"来显示更详细的格式
-n              ：使得 sudo 变成非交互模式，但如果安全策略是要求输入密码的，则 sudo 将报错
-S              ：(stdin)该选项使得 sudo 从标准输入而非终端设备上读取密码，给定的密码必须在尾部加上换行符
-s [command]    ：(shell)指定要切换到的 shell，如果给定 command，则在此 shell 上执行该命令
-U user         ：(other user)配合-l 选项来指定要列出哪个用户的权限信息
-u user         ：(user)该选项明确指定要以此处指定的用户而非 root 来运行 command。
                  若使用 uid 的方式指定用户，则需要使用"#uid"，但很多时候可能需要对"#"使用"\"转义，即使用"\#uid"
-E              ：(environment)该选项告诉 sudo 在执行命令时保留自己的环境变量，保留环境变量的方式是执行环境配置文件。
                  但因为跨了用户，所以很可能某些家目录下的环境配置文件会因为无权限而执行失败，此时 sudo 将报错
-k [command]    ：当单独使用-k 选项时，sudo 将使得用户的认证缓存失效。下次执行 sudo 命令需要输入密码。
                  当配合 command 时，-k 选项将忽略用户的缓存，所以 sudo 将要求用户输入密码，但这次输入密码不会更新认证缓存
                  但执行-k 选项本身，不需要密码
-K              ：(sure kill)类似于-k 选项，但它会完全移除用户的认证缓存，且不会配合 command，执行-K 本身不需要密码
-v              ：(validate)该选项使得 sudo 更新用户认证缓存
--              ：暗示 sudo 命令行参数到此结束

在 sudo 上可以直接设置环境变量，它会传递为 command 的环境。设置的方式为 var=value，如 LD_LIBRARY_PATH=/usr/local/pkg/lib
```

由于 sudo 默认的安全策略插件是 sudoers，所以当用户执行 sudo 时，系统会自动去寻找/etc/sudoers 文件(该文件里被 root 配置了用户对应的权限，也即安全策略)，查看 sudo 要使用的用户是否有对应的权限，如果有则执行，如果没有权限就失败退出 sudo。

第3章 文件的权限

3.1 文件/目录的权限

3.1.1 文件的权限

每个文件都有其所有者(u:user)、所属组(g:group)和其他人(o:other)对它的操作权限，a:all 则同时代表这 3 者。权限包括读(r:read)、写(w:write)、执行(x:execute)。在不同类型的文件上读、写、执行权限的体现有所不同，所以目录权限和普通文件权限要区分开来。

在普通文件上：

r：可读，可以使用类似 cat 等命令查看文件内容；读是文件的最基本权限，没有读权限，普通文件的一切操作行为都被限制。

w：可写，可以编辑此文件；

x：可执行，表示文件可由特定的解释器解释并运行。可以理解为 windows 中的可执行程序或批处理脚本，双击就能运行起来的文件。

在目录上：

r：可以对目录执行 ls 以列出目录内的所有文件；读是文件的最基本权限，没有读权限，目录的一切操作行为都被限制。

w：可以在此目录创建或删除文件/子目录；

x：可进入此目录，可使用 ls -l 查看文件的详细信息。可以理解为 windows 中双击就进入目录的动作。

如果目录没有 x 权限，其他人将无法查看目录内文件属性(只能查看到文件类型和文件名，至于为什么，见后文)，所以一般目录都要有 x 权限。而如果只有执行却没有读权限，则权限拒绝。

一般来说，普通文件的默认权限是 644(没有执行权限)，目录的默认权限是 755(必须有执行权限，否则进不去)，链接文件的权限是 777。当然，默认文件的权限设置方法是可以通过 umask 值来改变的。

3.1.2 权限的表示方式

权限的模式有两种体现：数字体现方式和字符体现方式。

权限的数字表示："-"代表没有权限,用 0 表示。

```
r-----4
w-----2
x-----1
```

例如：rwx rw- r--对应的数字权限是 764，732 代表的权限数值表示为 rwx -wx -w-。

3.1.3 chmod 修改权限

能够修改权限的人只有文件所有者和超级管理员。

```
chmod [OPTION]... MODE[,MODE]... FILE...
chmod [OPTION]... num_mode FILE...
chmod [OPTION]... --reference=RFILE FILE...
选项说明：
--reference=RFILE：引用某文件的权限作为权限值
-R：递归修改，只对当前已存在的文件有效
```

(1). 使用数值方式修改权限

```
chmod 755 /tmp/a.txt
```

(2). 使用字符方式修改权限

由于权限属性附在文件所有者、所属组和其它上，它们三者都有独立的权限位，所有者使用字母"u"表示，所属组使用"g"来表示，其他使用"o"来表示，而字母"a"同时表示它们三者。所以使用字符方式修改权限时，需要指定操作谁的权限。

```
chmod [ugoa][+ - =] [权限字符] 文件/目录名
"+"是加上权限，"-"是减去权限，"="是直接设置权限
```

```
[root@xuexi tmp]# chmod u-x,g-x,o-x test        # 将 ugo 都去掉 x 权限，等价于 chmod -x test
[root@xuexi tmp]# chmod a+x test                # 为 ugo 都加上 x 权限，等价于 chmod +x test
```

3.1.4 chgrp

更改文件和目录的所属组，要求组已经存在。

注意，对于链接文件而言，修改组的作用对象是链接的源文件，而非链接文件本身。

```
chgrp [OPTION]... GROUP FILE...
chgrp [OPTION]... --reference=RFILE FILE..
-R：递归修改
```

--reference=dest_file file_list：引用某文件的 group 作为文件列表的组,即将 file 文件列表的组改为 dest_file 的组

3.1.5 chown

chown 可以修改文件所有者和所属组。

注意，对于链接文件而言，默认不会穿过链接修改源文件，而是直接修改链接文件本身，这和 chgrp 的默认是不一样的。

```
chown [OPTION]... [OWNER][:[GROUP]] FILE...
chown [OPTION]... [OWNER][.[GROUP]] FILE...
chown [OPTION]... --reference=RFILE FILE...
选项说明：
--from=CURRENT_OWNER:CURRENT_GROUP：只修改当前所有者或所属组为此处指定的值的文件
--reference=RFILE：引用某文件的所有者和所属组的值作为新的所有者和所属组
-R：递归修改。注意，当指定-R 时，且同时指定下面某一个选项时对链接文件有不同的行为
      -H：如果 chown 的文件参数是一个链接到目录的链接文件，则穿过此链接文件修改其源目录的所有者和所属组
      -L：目录中遇到的所有链接文件都穿越过去，修改它们的源文件的所有者和所属组
      -P：不进行任何穿越，只修改链接文件本身的所有者和所属组。(这是默认值)
      这 3 项若同时指定多项时，则最后一项生效
```

chown 指定所有者和所属组的方式有两种，使用冒号和点。

```
chown root.root test
chown root:root test
chown root test     # 只修改所有者
chown :root test
chown .root test
```

3.2 实现权限的本质

涉及文件系统的知识点，若不理解，可以先看看文件系统的内容。此处是以 ext4 文件系统为例的，在其他文件系统上结果可能会有些不一样(centos 7 上使用 xfs 文件系统时结果可能就不一样)，但本质是一样的。

不同的权限表示对文件具有不同能力，如读写执行(rwx)权限，但是它是怎么实现的呢？描述文件权限的数据放在哪里呢？

首先，权限的元数据放在 inode 中，严格地说是放在 inode table 中，因为每个块组的所有 inode 组成一个 inode table。在 inode table 中使用一列来存放数字型的权限，比如某文件的权限为 644。每次用户要对文件进行操作时系统都会先查看权限，确认该用户是否有对应的权限来执行操作。当然，inode table 一般都已经加载到内存中，所以每次查询权限的资源消耗是非常小的。

Inode Number	File Type	Permissiontion	Link count	UID	GID	size	...	pointer
1	-	644	1	500	500			
2	d	755	1	0	0			
...	...	...	...	...	...	...	...	...
n	-	644	2	501	501			

无论是读、写还是执行权限，所体现出来的能力究其本质都是因为它作用在对应文件的 data block 上。

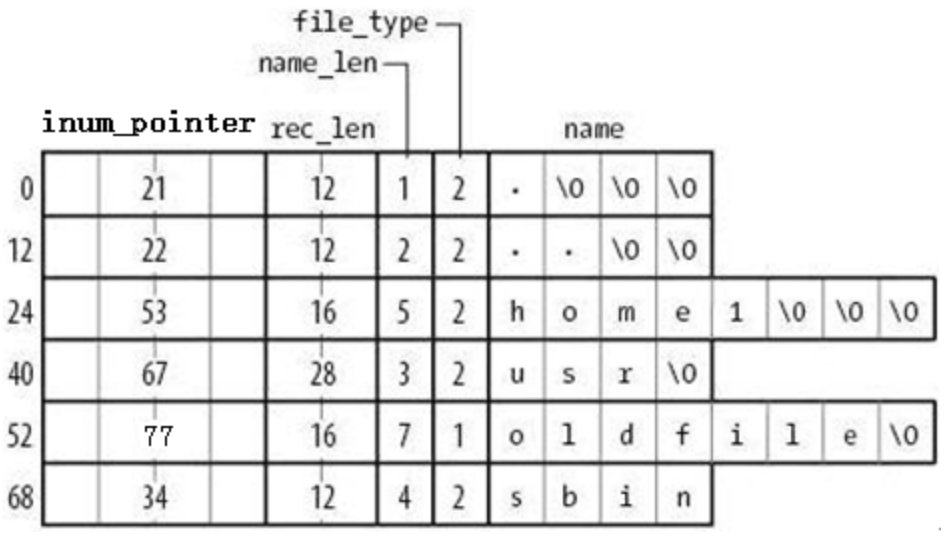
3.2.1 读权限(r)

对普通文件具有读权限表示的是具有读取该文件内容的能力，对目录具有读权限表示具有浏览该目录中文件或子目录的能力。其本质都是具有读取其 data block 的能力。

对于普通文件而言，能够读取文件的 data block，而普通文件的 data block 存储的直接就是数据本身， 所以对普通文件具有读权限表示能够读取文件内容。

对于目录文件而言，能够读取目录的 data block，而目录文件的 data block 存储的内容包括但不限于：目录中文件的 inode 号(并非直接存储，而是存储指向 inode table 中该 inode 号的指针)以及这些文件的文件类型、文件名。所以能够读取目录的 data block 表示仅能获取到这些信息。

目录的 data block 内容示例如下：



例如：

```
mkdir -p /mydata/data/testdir/subdir     # 创建 testdir 测试目录和其子目录 subdir
touch /mydata/data/testdir/a.log        # 再在 testdir 下创建一个普通文件
chmod 754 /mydata/data/testdir         # 将 testdir 设置为对其他人只有读权限
```

然后切换到普通用户查看 testdir 目录的内容。

```
su - wangwu
ll -ai /mydata/data/testdir/
ls: cannot access /mydata/data/testdir/..: Permission denied
ls: cannot access /mydata/data/testdir/a.log: Permission denied
ls: cannot access /mydata/data/testdir/subdir: Permission denied
ls: cannot access /mydata/data/testdir/.: Permission denied
total 0
? d????????? ? ? ? ?            ? .
? d????????? ? ? ? ?            ? ..
? -????????? ? ? ? ?            ? a.log
? d????????? ? ? ? ?            ? subdir
```

从结果中看出，testdir 下的文件名和文件类型是能够读取的，但是其他属性都不能读取到。而且也读取不到 inode 号，因为它并没有直接存储 inode 号，而是存储了指向 Inode 号的指针，要定位到指针的指向需要执行权限。

3.2.2 执行权限(x)

执行权限表示的是能够执行。如何执行？执行这个词不是很好解释，可以简单的类比 Windows 中的双击行为。例如对目录双击就能进入到目录，对批处理文件双击就能运行(有专门的解释器解释)，对可执行程序双击就能运行等。

当然，读权限是文件的最基本权限，执行权限能正常运行必须得配有读权限。

对目录有执行权限，表示可以通过目录的 data block 中指向文件 inode 号的指针定位到 inode table 中该文件的 inode 信息，所以可以显示出这些文件的全部属性信息。

3.2.3 写权限(w)

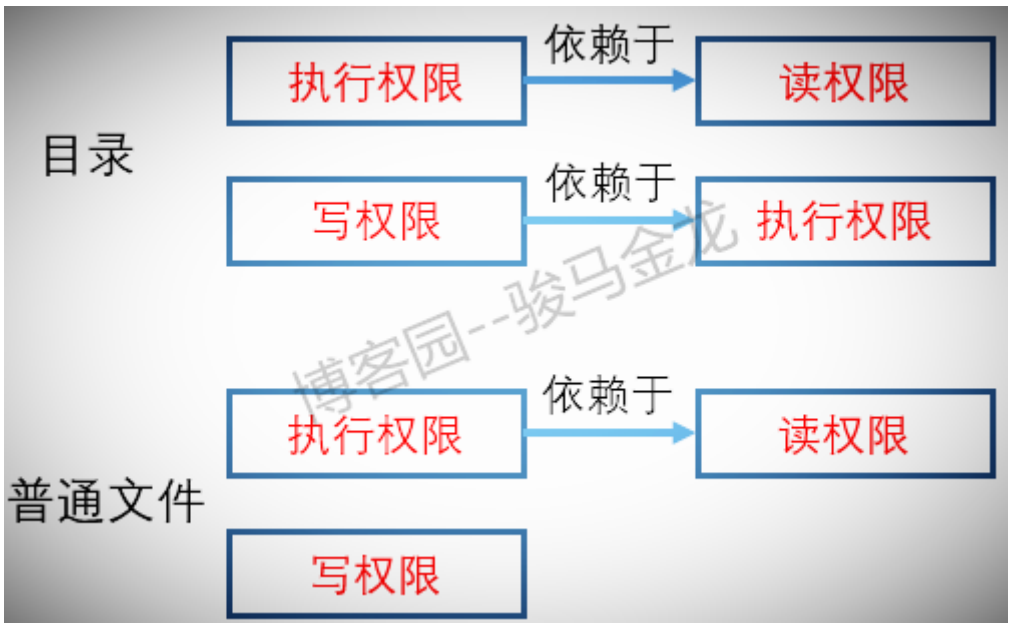
写权限很简单，就是能够将数据写入分配到的 data block。

对目录文件具有写权限，表示能够创建和删除文件。目录的写操作实质是能够在目录的 data block 中创建或删除关于待操作文件的记录。它要求对目录具有执行权限，因为无论是创建还是删除其内文件，都需要将其 data block 中 inode 号和 inode table 中的 inode 信息关联或删除。

对普通文件具有写权限，实质是能够改写该文件的 data block。

还是要说明的是，对文件有写权限不代表能够删除该文件，因为删除文件是要在目录的 data block 中删除该文件的记录，也就是说删除权限是在目录中定义的。

所以，对目录文件和普通文件而言，读、写、执行权限它们的依赖关系如下图所示。



3.3 umask 说明

umask 值用于设置用户在创建文件时的默认权限。对于 root 用户(实际上是 UID 小于 200 的 user)，系统默认的 umask 值是 022；对于普通用户和系统用户，系统默认的 umask 值是 002。

默认它们的设置是写在/etc/profile 和/etc/bashrc 两个环境配置文件中。

```
grep -C 5 -R 'umask 002'  /etc | grep 'umask 022'
/etc/bashrc-       umask 022
/etc/csh.cshrc-    umask 022
/etc/profile-    umask 022
```

相关设置项如下：

```
if [ $UID -gt 199 ] && [ "`id -gn`" = "`id -un`" ]; then
   umask 002
else
   umask 022
fi
```

执行 umask 命令可以查看当前用户的 umask 值。

```
[root@xuexi tmp]# umask
0022
[longshuai@xuexi tmp]$ umask
0002
```

执行 umask num 可以临时修改 umask 值为 num，但这是临时的，要永久有效，需要写入到环境配置文件中，至于写入到/etc/profile、/etc/bashrc、~/.bashrc 还是~/.bash_profile 中，看你自己的需求了。不过一般来说，不会去永久修改 umask 值，只会在特殊条件下临时修改下 umask 值。

umask 是如何决定创建文件的默认权限的呢？

如果创建的是目录，则使用 777-umask 值，如 root 的 umask=022，则 root 创建目录时该目录的默认权限为 777-022=755，而普通用户创建目录时，权限为 777-002=775.

如果创建的是普通文件，在 Linux 中，深入贯彻了一点：文件默认不应该有执行权限，否则是危险的。所以在计算时，可能会和想象中的结果不一样。

如果 umask 的三位都为偶数，则直接使用 666 去减掉 umask 值，因为 6 减去一个偶数还是偶数，任何位都不可能会有执行权限。如 root 创建普通文件时默认权限为 666-022=644，而普通用户创建普通文件时默认权限为 666-002=664。

如果 umask 值某一位为奇数，则 666 减去 umask 值后再在奇数位上加 1。如 umask=021 时，创建文件时默认权限为 666-021=645，在奇数位上加 1，则为 646。

```
[longshuai@xuexi tmp]$ umask 021
[longshuai@xuexi tmp]$ touch b.txt
[longshuai@xuexi tmp]$ ls -l b.txt
-rw-r--rw- 1 longshuai longshuai 0 Jun  7 12:02 b.txt
```

总之计算出后默认都是没有执行权限的。

3.4 文件的扩展 ACL 权限

在计算机界，所有的 ACL(access control list)都表示访问控制列表。

文件的 owner/group/others 的权限就是一种 ACL，它们是基本的 ACL。很多时候，只通过这 3 个权限位是无法完全合理设置权限问题的，例如如何仅设置某单个用户具有什么权限。这时候需要使用扩展 ACL。

扩展 ACL 是一种特殊权限，它是文件系统上功能，用于解决所有者、所属组和其他这三个权限位无法合理设置单个用户权限的问题。所以，扩展 ACL 可以针对单一使用者，单一档案或目录里的默认权限进行 r,w,x 的权限规范。

需要明确的是，扩展 ACL 是文件系统上的功能，且工作在内核，默认在 ext4/xfs 上都已开启。

在下文中，都直接以 ACL 来表示代替扩展 ACL 的称呼。

3.4.1 查看文件系统是否开启 ACL 功能

对于 ext 家族的文件系统来说，要查看是否开启 acl 功能，使用 dumpe2fs 导出文件系统属性即可。

```
dumpe2fs -h /dev/sda2 | grep -i acl
dumpe2fs 1.41.12 (17-May-2010)
Default mount options:    user_xattr acl
```

对于 xfs 文件系统，则没有直接的命令可以输出它的相关信息，需要使用 dmesg 来查看。其实无需关注它，因为默认 xfs 会开启 acl 功能。

```
dmesg | grep -i acl
[    1.465903] systemd[1]: systemd 219 running in system mode. (+PAM +AUDIT +SELINUX +IMA -APPARMOR +SMACK +SYSVINIT +UTMP +LIBCRYPTSETUP +GCRYPT +GNUTLS +ACL +XZ -LZ4 -SECCOMP +BLKID +ELFUTILS +KMOD +IDN)
[    2.517705] SGI XFS with ACLs, security attributes, no debug enabled
```

开启 ACL 功能后，不代表就使用 ACL 功能。是否使用该功能，不同文件系统控制方法不一样，对于 ext 家族来说，通过 mount 挂载选项来控制，而对于 xfs 文件系统，mount 命令根本不支持 acl 参数(xfs 文件系统如何关闭或启用的方法本人也不知道)。

3.4.2 设置和查看 ACL

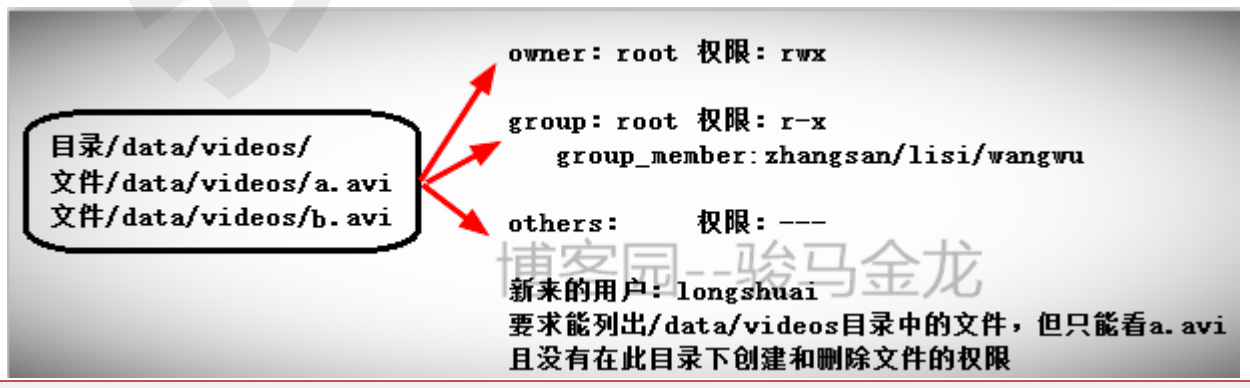
设置使用 setfacl 命令。

```
setfacl [options] u:[用户列表]:[rwx] 目录/文件名    # 对用户设置使用 u
setfacl [options] g:[组列表]:[rwx]   目录/文件名    # 对组设置使用 g
选项说明：
-m：设定 ACL 权限(modify)
-x：删除指定的 ACL 权限，可以指定用户、组和文件来删除(remove)
-M：写了 ACL 条目的文件，将从此文件中读取 ACL 条目，需要配合-m，所以-M 指定的是 modify file
-X：写了 ACL 条目的文件，将从此文件中读取 ACL 条目，需要配合-x，所以-X 指定的是 remove file
-n：不重置 mask
-b：删除所有的 ACL 权限
-d：设定默认 ACL 权限，只对目录有效，设置后子目录(文件)继承默认 ACL，只对未来文件有效
-k：删除默认 ACL 权限
-R：递归设定 ACL 权限，只对目录有效，只对已有文件有效
```

查看使用 getfacl 命令

```
getfacl filename
```

案例：假设现有目录/data/videos 专门存放视频，其中有一个 a.avi 的介绍性视频。该目录的权限是 750。现在有一个新用户加入，但要求该用户对该目录只有查看的权限，且只能看其中一部视频 a.avi，另外还要求该用户在此目录下没有创建和删除文件的权限。




```
mkdir -p /data/videos
chmod 750 /data/videos
touch /data/videos/{a,b}.avi
echo "xxx" >/data/videos/a.avi
echo "xxx" >/data/videos/b.avi
chown -R root.root /data/videos
```

1.首先设置用户 longshuai 对/data/videos 目录有读和执行权限。

```
setfacl -m u:longshuai:rx /data/videos
```

2.现在 longshuai 对/data/videos 目录下的所有文件都有读权限，因为默认文件的权限为 644。要设置 longshuai 只对 a.avi 有读权限，先设置所有文件的权限都为不可读。

```
chmod 640 /data/videos/*
```

3.然后再单独设置 a.avi 的读权限。

```
setfacl -m u:longshuai:r /data/videos/a.avi
```

到此就设置完成了。查看/data/videos/和/data/videos/a.avi 上的 ACL 信息。

```
getfacl /data/videos/
getfacl: Removing leading '/' from absolute path names
# file: data/videos/
# owner: root
# group: root
user::rwx
user:longshuai:r-x          # 用户 longshuai 在此文件上的权限是 r-x
group::r-x
mask::r-x
other::---
getfacl /data/videos/a.avi
getfacl: Removing leading '/' from absolute path names
# file: data/videos/a.avi
# owner: root
# group: root
user::rw-
user:longshuai:r--          # 用户 longshuai 在此文件上的权限是 r--
group::r--
mask::r--
other::---
```

3.4.3 ACL:mask

设置 mask 后会将 mask 权限与已有的 acl 权限进行与计算，计算后的结果会成为新的 ACL 权限。

设定 mask 的方式为：

```
setfacl -m m:[rwx] 目录/文件名
```

A	B	and
r	r	r
r	-	-
-	r	-
-	-	-

注意：默认每次设置文件的 acl 都会重置 mask 为此次给定的用户的值。既然如此，要如何控制文件上的 acl 呢？如果一个文件上要设置多个用户的 acl，重置 mask 后就会对已有用户的 acl 重新计算，而使得 acl 权限得不到有效的控制。使用 setfacl 的"-n"选项，它表示此次设置不会重置 mask 值。

例如：

当前的 acl 权限：

```
getfacl /data/videos
getfacl: Removing leading '/' from absolute path names
# file: data/videos
# owner: root
# group: root
user::rwx
user:longshuai:rwx
group::r-x
mask::rwx
other::---
```

设置 mask 值为 rx。

```
setfacl -m m:rx /data/videos
getfacl /data/videos
getfacl: Removing leading '/' from absolute path names
# file: data/videos
# owner: root
# group: root
user::rwx
user:longshuai:rwx              #effective:r-x
```

```
group::r-x
mask::r-x
other::---
```

设置 mask 后，它提示有效权限是 r-x。这是 rwx 和 r-x 做与运算之后的结果。

再设置 longshuai 的 acl 为 rwx，然后查看 mask，会发现 mask 也被重置为 rwx。

```
setfacl -m u:longshuai:rwx /data/videos
getfacl  /data/videos
getfacl: Removing leading '/' from absolute path names
# file: data/videos
# owner: root
# group: root
user::rwx
user:longshuai:rwx
group::r-x
mask::rwx
other::---
```

所以，在设置文件的 acl 时，要使用-n 选项来禁止重置 mask。

```
setfacl -m m:rx /data/videos
setfacl -n -m u:longshuai:rwx /data/videos
getfacl  /data/videos
getfacl: Removing leading '/' from absolute path names
# file: data/videos
# owner: root
# group: root
user::rwx
user:longshuai:rwx              #effective:r-x
group::r-x
mask::r-x
other::---
```

3.4.4 设置递归和默认 ACL 权限

递归 ACL 权限只对目录里已有文件有效，默认权限只对未来目录里的文件有效。

设置递归 ACL 权限：

```
setfacl -m u:username:[rwx] -R 目录名
```

设置默认 ACL 权限：

```
setfacl -m d:u:username:[rwx] 目录名
```

3.4.5 删除 ACL 权限

```
setfacl -x u:用户名 文件名        # 删除指定用户 ACL
setfacl -x g:组名 文件名          # 删除指定组名 ACL
setfacl -b 文件名                 # 指定文件删除 ACL，会删除所有 ACL
```

3.5 文件隐藏属性

chattr:change file attributes

lsattr:list file attributes

```
chattr [+ - =] [ai] 文件或目录名
```

常用的参数是 a(append，追加)和 i(immutable，不可更改)，其他参数略。

设置了 a 参数时，文件中将只能增加内容，不能删除数据，且不能打开文件进行任何编辑，哪怕是追加内容也不可以，所以像 sed 等需要打开文件的再写入数据的工具也无法操作成功。文件也不能被删除。只有 root 才能设置。

设置了 i 参数时，文件将被锁定，不能向其中增删改内容，也不能删除修改文件等各种动作。只有 root 才能设置。可以将其理解为设置了 i 后，文件将是永恒不变的了，谁都不能动它。

例如，对/etc/shadow 文件设置 i 属性，任何用户包括 root 将不能修改密码，而且也不能创建用户。

```
chattr +i /etc/shadow
```

此时如果新建一个用户。

```
useradd newlongsuai
useradd: cannot open /etc/shadow        # 提示文件不能打开，被锁定了
```

lsattr 查看文件设置的隐藏属性。

```
lsattr /etc/shadow
----i--------e- /etc/shadow        # i 属性说明被锁定了，e 是另一种文件属性，忽略它
```

删除隐藏属性：

```
chattr -i /etc/shadow
lsattr /etc/shadow
-------------e- /etc/shadow
```

再来一例：

```
chattr +a test1.txt       # 对 test1.txt 设置 a 隐藏属性
echo 1234>>test1.txt     # 追加内容是允许的行为
cat /dev/null >test1.txt    # 但是清空文件内容是不允许的
-bash: test1.txt: Operation not permitted
```

3.6 suid/sgid/sbit

3.6.1 suid

suid 只针对可执行文件，即二进制文件。它的作用是对某个命令(可执行文件)授予所有者的权限，命令执行完成权限就消失。一般是提权为 root 权限。

例如/etc/shadow 文件所有人都没有权限(root 除外)，其他用户连看都不允许。

```
ll /etc/shadow
----------. 1 root root 752 Apr  8 12:42 /etc/shadow
```

但是他们却能修改自己的密码，说明他们一定有一定的权限。这个权限就是 suid 控制的。

```
ls -l /usr/bin/passwd
-rwsr-xr-x. 1 root root 30768 Feb 22  2012 /usr/bin/passwd
```

其中的"s"权限就是 suid，它出现在所有者位置上(是 root)，其他用户执行 passwd 命令时，会暂时拥有所有者位的 rwx 权限，也就是 root 的权限，所以能向/etc/shadow 写入数据。

suid 必须和 x 配合，如果没有 x 配合，则该 suid 是空 suid，仍然没有执行命令的权限，所有者都没有了 x 权限，suid 依赖于它所以更不可能有 x 权限。空的 suid 权限使用大写的"S"表示。

数字 4 代表 suid，如 4755。

3.6.2 sgid

针对二进制文件和目录。

- 针对二进制文件时，权限升级为命令的所属组权限。
- 针对目录时，目录中所建立的文件或子目录的组将继承默认父目录组。必须和 rx 配合，普通用户才能进入目录，如果普通用户有 w 权限，新建的文件和目录则以父目录组为默认组。

以 2 代表 sgid，如 2755，和 suid 组合如 6755。

3.6.3 sbit

只对目录有效。对目录设置 sbit，将使得目录里的文件只有所有者能删除，即使其他用户在此目录上有 rwx 权限，即使是 root 用户。

以 1 代表 sbit。

补充：suid/sgid/sbit 的标志位都作用在 x 位，当原来的 x 位有 x 权限时，这些权限位则为 s/s/t，如果没有 x 权限，则变为 S/S/T。例如，/tmp 目录的权限有个 t 位，使得该目录里的文件只有其所有者本身能删除。

第4章 ext 文件系统机制

将磁盘进行分区，分区是将磁盘按柱面进行物理上的划分。划分好分区后还要进行格式化，然后再挂载才能使用(不考虑其他方法)。格式化分区的过程其实就是创建文件系统。

文件系统的类型有很多种，如 CentOS 5 和 CentOS 6 上默认使用的 ext2/ext3/ext4，CentOS 7 上默认使用的 xfs，windows 上的 NTFS，光盘类的文件系统 ISO9660，MAC 上的混合文件系统 HFS，网络文件系统 NFS，Oracle 研发的 btrfs，还有老式的 FAT/FAT32 等。

本文将非常全面且详细地介绍 ext 家族的文件系统，中间还非常详细地介绍了 inode、软链接、硬链接、数据存储方式以及操作文件的理论，基本上看完本文，对文件系统的宏观理解将再无疑惑。ext 家族的文件系统有 ext2/ext3/ext4，ext3 是有日志的 ext2 改进版，ext4 对相比 ext3 做了非常多的改进。虽然 xfs/btrfs 等文件系统有所不同，但它们只是在实现方式上不太同，再加上属于自己的特性而已。

4.1 文件系统的组成部分

4.1.1 block 的出现

硬盘最底层的读写 IO 一次是一个扇区 512 字节，如果要读写大量文件，以扇区为单位肯定很慢很消耗性能，所以硬盘使用了一个称作逻辑块的概念。逻辑块是逻辑的，由磁盘驱动器负责维护和操作，它并非是像扇区一样物理划分的。一个逻辑块的大小可能包含一个或多个扇区，每个逻辑块都有唯一的地址，称为 LBA。有了逻辑块之后，磁盘控制器对数据的操作就以逻辑块为单位，一次读写一个逻辑块，磁盘控制器知道如何将逻辑块翻译成对应的扇区并读写数据。

到了 Linux 操作系统层次，通过文件系统提供了一个也称为块的读写单元，文件系统数据块的大小一般为 1024bytes(1K)或 2048bytes(2K)或 4096bytes(4K)。文件系统数据块也是逻辑概念，是文件系统层次维护的，而磁盘上的逻辑数据块是由磁盘控制器维护的，文件系统的 IO 管理器知道如何将它的数据块翻译成磁盘维护的数据块地址 LBA。对于使用文件系统的 IO 操作来说，比如读写文件，这些 IO 的基本单元是文件系统上的数据块，一次读写一个文件系统数据块。比如需要读一个或多个块时，文件系统的 IO 管理器首先计算这些文件系统块对应在哪些磁盘数据块，也就是计算出 LBA，然后通知磁盘控制器要读取哪些块的数据，硬盘控制器将这些块翻译成扇区地址，然后从扇区中读取数据，再通过硬盘控制器将这些扇区数据重组写入到内存中去。

本文既然是讨论文件系统的，那么重点自然是在文件系统上而不是在磁盘上，所以后文出现的 block 均表示的是文件系统的数据块而不是磁盘维护的逻辑块。

文件系统 block 的出现使得在文件系统层面上读写性能大大提高，也大量减少了碎片。但是它的副作用是可能造成空间浪费。由于文件系统以 block 为读写单元，即使存储的文件只有 1K 大小也将占用一个 block，剩余的空间完全是浪费的。在某些业务需求下可能大量存储小文件，这会浪费大量的空间。

尽管有缺点，但是其优点足够明显，在当下硬盘容量廉价且追求性能的时代，使用 block 是一定的。

4.1.2 inode 的出现

如果存储的 1 个文件占用了大量的 block 读取时会如何？假如 block 大小为 1KB，仅仅存储一个 10M 的文件就需要 10240 个 block，而且这些 blocks 很可能在位置上是不连续在一起的(不相邻)，读取该文件时难道要从前向后扫描整个文件系统的块，然后找出属于该文件的块吗？显然是不应该这么做的，因为太慢太傻瓜式了。再考虑一下，读取一个只占用 1 个 block 的文件，难道只读取一个 block 就结束了吗？并不是，仍然是扫描整个文件系统的所有 block，因为它不知道什么时候扫描到，扫描到了它也不知道这个文件是不是已经完整而不需要再扫描其他的 block。

另外，每个文件都有属性(如权限、大小、时间戳等)，这些属性类的元数据存储在哪里呢？难道也和文件的数据部分存储在块中吗？如果一个文件占用多个 block 那是不是每个属于该文件的 block 都要存储一份文件元数据？但是如果不在每个 block 中存储元数据文件系统又怎么知道某一个 block 是不是属于该文件呢？但是显然，每个数据 block 中都存储一份元数据太浪费空间。

文件系统设计者当然知道这样的存储方式很不理想，所以需要优化存储方式。如何优化？对于这种类似的问题的解决方法是使用索引，通过扫描索引找到对应的数据，而且索引可以存储部分数据。

在文件系统上索引技术具体化为索引节点(index node)，在索引节点上存储的部分数据即为文件的属性元数据及其他少量信息。一般来说索引占用的空间相比其索引的文件数据而言占用的空间就小得多，扫描它比扫描整个数据要快得多，否则索引就没有存在的意义。这样一来就解决了前面所有的问题。

在文件系统上的术语中，索引节点称为 inode。在 inode 中存储了 inode 号（注，inode 中并未存储 inode num，但为了方便理解，这里暂时认为它存储了 inode 号）、文件类型、权限、文件所有者、大小、时间戳等元数据信息，最重要的是还存储了指向属于该文件 block 的指针，这样读取 inode 就可以找到属于该文件的 block，进而读取这些 block 并获得该文件的数据。由于后面还会介绍一种指针，为了方便称呼和区分，暂且将这个 inode 记录中指向文件 data block 的指针称之为 block 指针。以下是 ext2 文件系统中 inode 包含的信息示例：

```
Inode: 12   Type: regular    Mode:  0644   Flags: 0x0
Generation: 1454951771    Version: 0x00000000:00000001
User:     0   Group:     0   Size: 5
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 8
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x5b628db2:15e0aff4 -- Thu Aug  2 12:50:58 2018
 atime: 0x5b628db2:15e0aff4 -- Thu Aug  2 12:50:58 2018
 mtime: 0x5b628db2:15e0aff4 -- Thu Aug  2 12:50:58 2018
crtime: 0x5b628db2:15e0aff4 -- Thu Aug  2 12:50:58 2018
Size of extra inode fields: 28
BLOCKS:
```

```
(0):1024
TOTAL: 1
```

一般 inode 大小为 128 字节或 256 字节，相比那些 MB 或 GB 计算的文件数据而言小得多的多，但也要知道可能一个文件大小小于 inode 大小，例如只占用 1 个字节的文件。

4.1.3 bmap 出现

在向硬盘存储数据时，文件系统需要知道哪些块是空闲的，哪些块是已经占用了的。最笨的方法当然是从前向后扫描，遇到空闲块就存储一部分，继续扫描直到存储完所有数据。

优化的方法当然也可以考虑使用索引，但是仅仅 1G 的文件系统就有 1KB 的 block 共 1024*1024=1048576 个，这仅仅只是 1G，如果是 100G、500G 甚至更大呢，仅仅使用索引，索引的数量和空间占用也将极大，这时就出现更高一级的优化方法：使用块位图(bitmap 简称 bmap)。

位图只使用 0 和 1 标识对应 block 是空闲还是被占用，0 和 1 在位图中的位置和 block 的位置一一对应，第一位标识第一个块，第二个位标识第二个块，依次下去直到标记完所有的 block。

考虑下为什么块位图更优化。在位图中 1 个字节 8 个位，可以标识 8 个 block。对于一个 block 大小为 1KB、容量为 1G 的文件系统而言，block 数量有 1024*1024 个，所以在位图中使用 1024*1024 个位共 1024*1024/8=131072 字节=128K，即 1G 的文件只需要 128 个 block 做位图就能完成一一对应。通过扫描这 100 多个 block 就能知道哪些 block 是空闲的，速度提高了非常多。

但是要注意，bmap 的优化针对的是写优化，因为只有写才需要找到空闲 block 并分配空闲 block。对于读而言，只要通过 inode 找到了 block 的位置，cpu 就能迅速计算出 block 在物理磁盘上的地址，cpu 的计算速度是极快的，计算 block 地址的时间几乎可以忽略，那么读速度基本认为是受硬盘本身性能的影响而与文件系统无关了。

虽然 bmap 已经极大的优化了扫描，但是仍有其瓶颈：如果文件系统是 100G 呢？100G 的文件系统要使用 128*100=12800 个 1KB 大小的 block，这就占用了 12.5M 的空间了。试想完全扫描 12800 个很可能不连续的 block 这也是需要占用一些时间的，虽然快但是扛不住每次存储文件都要扫描带来的巨大开销。

所以需要再次优化，如何优化？简而言之就是将文件系统划分开形成块组，至于块组的介绍放在后文。

4.1.4 inode 表的出现

回顾下 inode 相关信息：inode 存储了 inode 号（注，同前文，inode 中并未存储 inode num）、文件属性元数据、指向文件占用的 block 的指针；每一个 inode 占用 128 字节或 256 字节。

现在又出现问题了，一个文件系统中可以说有无数多个文件，每一个文件都对应一个 inode，难道每一个仅 128 字节的 inode 都要单独占用一个 block 进行存储吗？这太浪费空间了。

所以更优的方法是将多个 inode 合并存储在 block 中，对于 128 字节的 inode，一个 block 存储 8 个 inode，对于 256 字节的 inode，一个 block 存储 4 个 inode。这就使得每个存储 inode 的块都不浪费。

在 ext 文件系统上，将这些物理上存储 inode 的 block 组合起来，在逻辑上形成一张 inode 表(inode table)来记录所有的 inode。

举个例子，每一个家庭都要向派出所登记户口信息，通过户口本可以知道家庭住址，而每个镇或街道的派出所将本镇或本街道的所有户口整合在一起，要查找某一户地址时，在派出所就能快速查找到。inode table 就是这里的派出所。它的内容如下图所示。

Inode Number	File Type	Permission	Link count	UID	GID	size	...	pointer
1	-	644	1	500	500			
2	d	755	1	0	0			
...	...	...	...	...	...	...	...	...
n	-	644	2	501	501			

再细细一思考，就能发现一个大的文件系统仍将占用大量的块来存储 inode，想要找到其中的一个 inode 记录也需要不小的开销，尽管它们已经形成了一张逻辑上的表，但扛不住表太大记录太多。那么如何快速找到 inode，这同样是需要优化的，优化的方法是将文件系统的 block 进行分组划分，每个组中都存有本组 inode table 范围、bmap 等。

4.1.5 imap 的出现

前面说 bmap 是块位图，用于标识文件系统中哪些 block 是空闲哪些 block 是占用的。

对于 inode 也一样，在存储文件(Linux 中一切皆文件)时需要为其分配一个 inode 号。但是在格式化创建文件系统后所有的 inode 号都已被事先计算好（创建文件系统时会为每个块组计算好该块组拥有哪些 inode 号），因此产生了问题：要为文件分配哪一个 inode 号呢？又如何知道某一个 inode 号是否已经被分配了呢？

既然是"是否被占用"的问题，使用位图是最佳方案，像 bmap 记录 block 的占用情况一样。标识 inode 号是否被分配的位图称为 inode map 简称为 imap。这时要为一个文件分配 inode 号只需扫描 imap 即可知道哪一个 inode 号是空闲的。

imap 存在着和 bmap 和 inode table 一样需要解决的问题：如果文件系统比较大，imap 本身就会很大，每次存储文件都要进行扫描，会导致效率不够高。同样，优化的方式是将文件系统占用的 block 划分成块组，每个块组有自己的 imap 范围。

4.1.6 块组的出现

前面一直提到的优化方法是将文件系统占用的 block 划分成块组(block group)，解决 bmap、inode table 和 imap 太大的问题。

在物理层面上的划分是将磁盘按柱面划分为多个分区，即多个文件系统；在逻辑层面上的划分是将文件系统划分成块组。每个文件系统包含多个块组，每个块组包含多个元数据区和数据区：元数据区就是存储 bmap、inode table、imap 等的数据；数据区就是存储文件数据的区域。注意块组是逻辑层面的概念，所以并不会真的在磁盘上按柱、按扇区、按磁道等概念进行划分。

4.1.7 块组的划分

块组在文件系统创建完成后就已经划分完成了，也就是说元数据区 bmap、inode table 和 imap 等信息占用的 block 以及数据区占用的 block 都已经划分好了。那么文件系统如何知道一个块组元数据区包含多少个 block，数据区又包含多少 block 呢？

它只需确定一个数据——每个 block 的大小，再根据 bmap 至多只能占用一个完整的 block 的标准就能计算出块组如何划分。如果文件系统非常小，所有的 bmap 总共都不能占用完一个 block，那么也只能空闲 bmap 的 block 了。

每个 block 的大小在创建文件系统时可以人为指定，不指定也有默认值。

假如现在 block 的大小是 1KB，一个 bmap 完整占用一个 block 能标识 1024*8= 8192 个 block(当然这 8192 个 block 是数据区和元数据区共 8192 个，因为元数据区分配的 block 也需要通过 bmap 来标识)。每个 block 是 1K，每个块组是 8192K 即 8M，创建 1G 的文件系统需要划分 1024/8=128 个块组，如果是 1.1G 的文件系统呢？128+12.8=128+13=141 个块组。

每个组的 block 数目是划分好了，但是每个组设定多少个 inode 号呢？inode table 占用多少 block 呢？这需要由系统决定了，因为描述"每多少个数据区的 block 就为其分配一个 inode 号"的指标默认是我们不知道的，当然创建文件系统时也可以人为指定这个指标或者百分比例。见后文"inode 深入"。

使用 dumpe2fs 可以将 ext 类的文件系统信息全部显示出来，当然 bmap 是每个块组固定一个 block 的不用显示，imap 比 bmap 更小所以也只占用 1 个 block 不用显示。

下图是一个文件系统的部分信息，在这些信息的后面还有每个块组的信息。

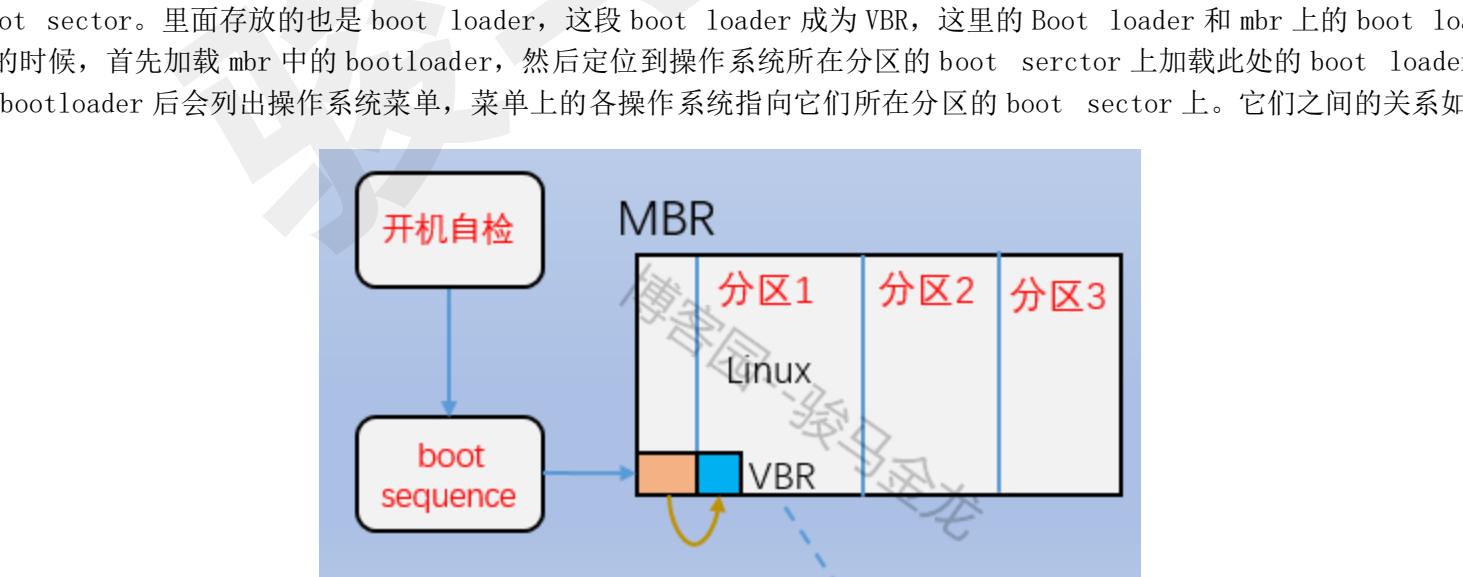
从这张表中能计算出文件系统的大小，该文件系统共 4667136 个 blocks，每个 block 大小为 4K，所以文件系统大小为 4667136*4/1024/1024=17.8GB。

也能计算出分了多少个块组，因为每一个块组的 block 数量为 32768，所以块组的数量为 4667136/32768=142.4 即 143 个块组。由于块组从 0 开始编号，所以最后一个块组编号为 Group 142。如下图所示是最后一个块组的信息。

```
Group 142: (Blocks 4653056-4667135) [ITABLE_ZEROED]
  Checksum 0x9401, unused inodes 0
  Block bitmap at 4194318 (+4294508558), Inode bitmap
  Inode table at 4201476-4201985 (+4294515716)
  14080 free blocks, 8160 free inodes, 0 directories
  Free blocks: 4653056-4667135
  Free inodes: 1158721-1166880
```

4.2 文件系统的完整结构

将上文描述的 bmap、inode table、imap、数据区的 blocks 和块组的概念组合起来就形成了一个文件系统，当然这还不是完整的文件系统。完整的文件系统如下图。



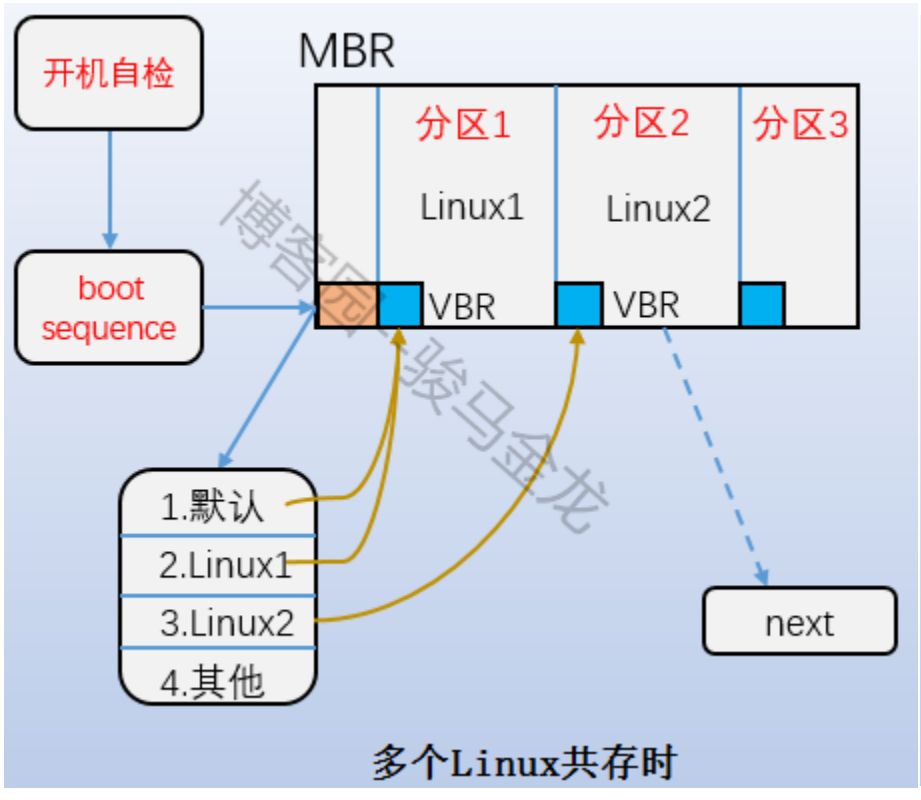
首先，该图中多了 Boot Block、Super Block、GDT、Reserver GDT 这几个概念。下面会分别介绍它们。

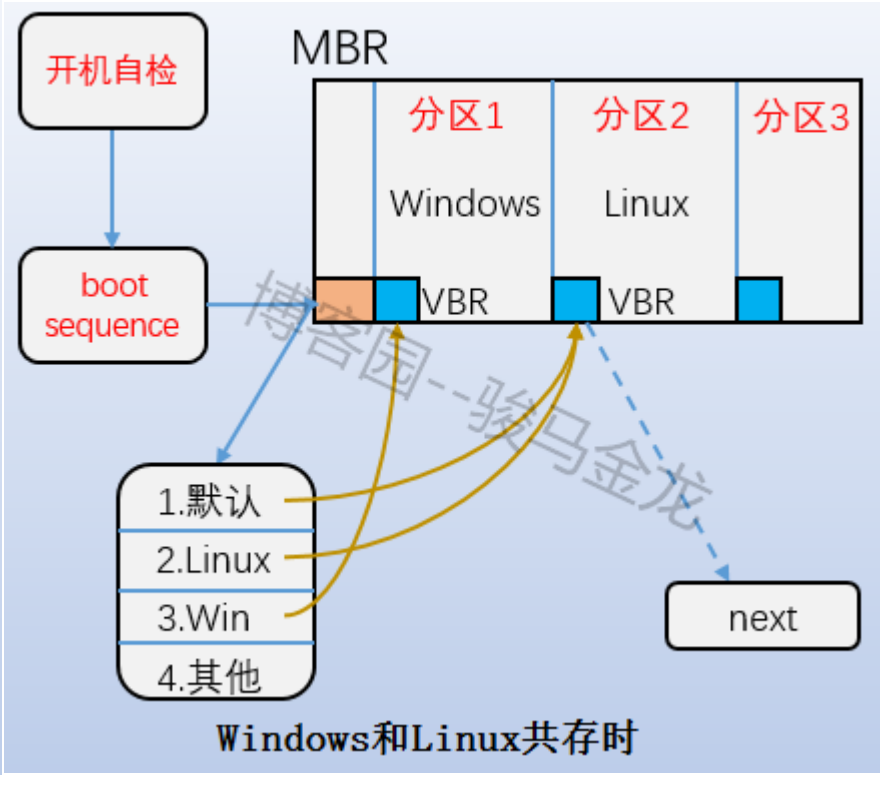
然后，图中指明了块组中每个部分占用的 block 数量，除了 superblock、bmap、imap 能确定占用 1 个 block，其他的部分都不能确定占用几个 block。

最后，图中指明了 Superblock、GDT 和 Reserved GDT 是同时出现且不一定存在于每一个块组中的，也指明了 bmap、imap、inode table 和 data blocks 是每个块组都有的。

4.2.1 引导块

即上图中的 Boot Block 部分，也称为 boot sector。它位于分区上的第一个块，占用 1024 字节，并非所有分区都有这个 boot sector，但装有操作系统的分区必定有 boot sector。里面存放的也是 boot loader，这段 boot loader 成为 VBR，这里的 Boot loader 和 mbr 上的 boot loader 是存在交错关系的。开机启动的时候，首先加载 mbr 中的 bootloader，然后定位到操作系统所在分区的 boot serctor 上加载此处的 boot loader。如果是多系统，加载 mbr 中的 bootloader 后会列出操作系统菜单，菜单上的各操作系统指向它们所在分区的 boot sector 上。它们之间的关系如下图所示。





4.2.2 超级块(superblock)

既然一个文件系统会分多个块组，那么文件系统怎么知道分了多少个块组呢？每个块组又有多少 block 多少 inode 号等等信息呢？还有，文件系统本身的属性信息如各种时间戳、block 总数量和空闲数量、inode 总数量和空闲数量、当前文件系统是否正常、什么时候需要自检等等，它们又存储在哪里呢？

毫无疑问，这些信息必须要存储在 block 中。存储这些信息占用 1024 字节，所以也要一个 block，这个 block 称为超级块(superblock)，它的 block 号可能为 0 也可能为 1。如果 block 大小为 1K，则引导块正好占用一个 block，这个 block 号为 0，所以 superblock 的号为 1；如果 block 大小大于 1K，则引导块和超级块同置在一个 block 中，这个 block 号为 0。总之 superblock 的起止位置是第二个 1024(1024-2047)字节。

使用 df 命令读取的就是每个文件系统的 superblock，所以它的统计速度非常快。相反，用 du 命令查看一个较大目录的已用空间就非常慢，因为不可避免地要遍历整个目录的所有文件。

```
[root@xuexi ~]# df -hT
Filesystem     Type   Size  Used Avail Use% Mounted on
/dev/sda3      ext4    18G  1.7G   15G  11% /
tmpfs          tmpfs  491M     0  491M   0% /dev/shm
/dev/sda1      ext4   190M   32M  149M  18% /boot
```

superblock 对于文件系统而言是至关重要的，超级块丢失或损坏必将导致文件系统的损坏。所以旧式的文件系统将超级块备份到每一个块组中，但是这又有所空间浪费，所以 ext2 文件系统只在块组 0、1 和 3、5、7 幂次方的块组中保存超级块的信息，如 Group9、Group25 等。尽管保存了这么多的 superblock，但是文件系统只使用第一个块组即 Group0 中超级块信息来获取文件系统属性，只有当 Group0 上的 superblock 损坏或丢失才会找下一个备份超级块复制到 Group0 中来恢复文件系统。

下图是一个 ext4 文件系统的 superblock 的信息，使用 dumpe2fs -h 可以获取。

```
Filesystem volume name:   <none>
Last mounted on:          /
Filesystem UUID:          e1b90643-6b1f-4e0b-ae41-049765784435
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode dir_index filetype
needs_recovery extent flex_bg sparse_super large_file huge_file uninit_bg dir_nlink
extra_isize
Filesystem flags:         signed_directory_hash
Default mount options:    user_xattr acl
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              1166880
Block count:              4667136
Reserved block count:     233356
Free blocks:              3963754
Free inodes:              1106146
First block:              0
Block size:               4096
Fragment size:            4096
Reserved GDT blocks:      1022
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         8160
Inode blocks per group:   510
Flex block group size:    16
Filesystem created:       Thu Feb 18 20:27:46 2016
Last mount time:          Fri Oct  7 21:26:58 2016
Last write time:          Thu Feb 18 20:59:06 2016
Mount count:              20
Maximum mount count:      -1
Last checked:             Thu Feb 18 20:27:46 2016
Check interval:           0 (<none>)
Lifetime writes:          4112 MB
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
First inode:              11
Inode size:               256
Required extra isize:     28
Desired extra isize:      28
Journal inode:            8
Default directory hash:   half_md4
Directory Hash Seed:      c9b3ad52-775e-491f-82ff-c808c5373893
Journal backup:           inode blocks
Journal features:         journal_incompat_revoke
Journal size:             128M
Journal length:           32768
Journal sequence:         0x00004e39
Journal start:            13134
```

4.2.3 块组描述符表(GDT)

既然文件系统划分了块组，那么每个块组的信息和属性元数据又保存在哪里呢？

ext 文件系统每一个块组信息使用 32 字节描述，这 32 个字节称为块组描述符，所有块组的块组描述符组成块组描述符表 GDT(group descriptor table)。

虽然每个块组都需要块组描述符来记录块组的信息和属性元数据，但是不是每个块组中都存放了块组描述符。ext 文件系统的存储方式是：将它们组成一个 GDT，并将该 GDT 存放于某些块组中，存放 GDT 的块组和存放 superblock 和备份 superblock 的块相同，也就是说它们是同时出现在某一个块组中的。

假如 block 大小为 4KB 的文件系统划分了 143 个块组，每个块组描述符 32 字节，那么 GDT 就需要 143*32=4576 字节即两个 block 来存放。这两个 GDT block 中记录了所有块组的块组信息，且存放 GDT 的块组中的 GDT 都是完全相同的。

下图是一个块组描述符的信息(通过 dumpe2fs 获取)。

```
Group 142: (Blocks 4653056-4667135) [ITABLE_ZEROED]
  Checksum 0x9401, unused inodes 0
  Block bitmap at 4194318 (+4294508558), Inode bitmap
  Inode table at 4201476-4201985 (+4294515716)
  14080 free blocks, 8160 free inodes, 0 directories
  Free blocks: 4653056-4667135
  Free inodes: 1158721-1166880
```

4.2.4 保留 GDT(Reserved GDT)

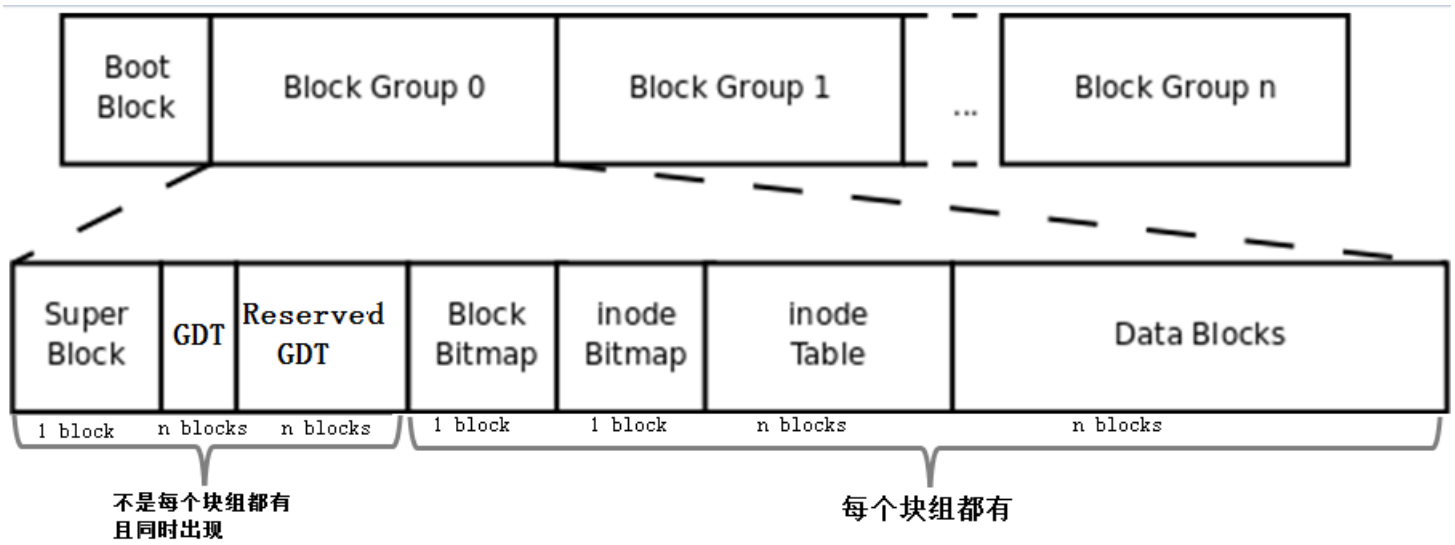
保留 GDT 用于以后扩容文件系统使用，防止扩容后块组太多，使得块组描述符超出当前存储 GDT 的 blocks。保留 GDT 和 GDT 总是同时出现，当然也就和 superblock 同时出现了。

例如前面 143 个块组使用了 2 个 block 来存放 GDT，但是此时第二个 block 还空余很多空间，当扩容到一定程度时 2 个 block 已经无法再记录块组描述符了，这时就需要分配一个或多个 Reserved GDT 的 block 来存放超出的块组描述符。

由于新增加了 GDT block，所以应该让每一个保存 GDT 的块组都同时增加这一个 GDT block，所以将保留 GDT 和 GDT 存放在同一个块组中可以直接将保留 GDT 变换为 GDT 而无需使用低效的复制手段备份到每个存放 GDT 的块组。

同理，新增加了 GDT 需要修改每个块组中 superblock 中的文件系统属性，所以将 superblock 和 Reserved GDT/GDT 放在一起又能提升效率。

4.3 Data Block



如上图，除了 Data Blocks 其他的部分都解释过了。data block 是直接存储数据的 block，但事实上并非如此简单。

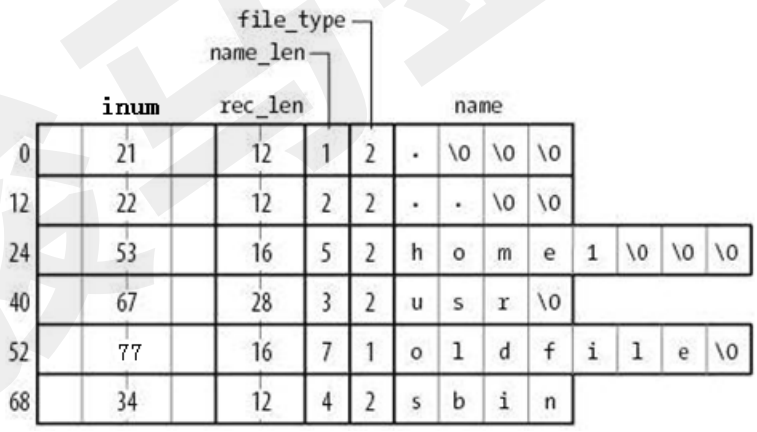
数据所占用的 block 由文件对应 inode 记录中的 block 指针找到，不同的文件类型，数据 block 中存储的内容是不一样的。以下是 Linux 中不同类型文件的存储方式。

- 对于常规文件，文件的数据正常存储在数据块中。
- 对于目录，该目录下的所有文件和一级子目录的目录名存储在数据块中。
  - → 文件名和 inode 号不是存储在其自身的 inode 中，而是存储在其所在目录的 data block 中。
- 对于符号链接，如果目标路径名较短则直接保存在 inode 中以便更快地查找，如果目标路径名较长则分配一个数据块来保存。
- 设备文件、FIFO 和 socket 等特殊文件没有数据块，设备文件的主设备号和次设备号保存在 inode 中。

常规文件的存储就不解释了，下面分别解释特殊文件的存储方式。

4.3.1 目录文件的 data block

目录的 data block 的内容如下图所示。



由图可知，在目录文件的数据块中存储了其下的文件名、目录名、目录本身的相对名称"."和上级目录的相对名称"..."，还存储了这些文件名对应的 inode 号、目录项长度 rec_len、文件名长度 name_len 和文件类型 file_type。注意到除了文件本身的 inode 记录了文件类型，其所在的目录的数据块也记录了文件类型。由于 rec_len 只能是 4 的倍数，所以需要使用"\0"来填充 name_len 不够凑满 4 倍数的部分。至于 rec_len 具体是什么，只需知道它是一种偏移即可。

需要注意的是，inode table 中的 inode 自身并没有存储每个 inode 的 inode 号，它是存储在目录的 data block 中的，通过 inode 号可以计算并索引到 inode table 中该 inode 号对应的 inode 记录，可以认为这个 inode 号是一个“inode 指针”（当然，并非真的是指针，但有助于理解通过 inode 号索引找到对应 inode 的这个过程，后文将在需要的时候使用 inode 指针这个词来表示 inode 号。至此，已经知道了两种指针：一种是 inode table 中每个 inode 记录指向其对应 data block 的 block 指针，一个此处的“inode 指针”）。

除了 inode 号，目录的 data block 中还使用数字格式记录了文件类型，数字格式和文件类型的对应关系如下图。

编码	文件类型
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

注意到目录的 data block 中前两行存储的是目录本身的相对名称"."和上级目录的相对名称"..", 它们实际上是目录本身的硬链接和上级目录的硬链接。什么是硬链接后面说明。

4.3.2 如何根据 inode 号找到 inode

前面提到过，inode 结构自身并没有保存 inode 号（同样，也没有保存文件名），那么 inode 号保存在哪里呢？目录的 data block 中保存了该目录中每个文件的 inode 号。

另一个问题，既然 inode 中没有 inode 号，那么如何根据目录 data block 中的 inode 号找到 inode table 中对应的 inode 呢？

实际上，只要有了 inode 号，就可以计算出 inode 表中对应该 inode 号的 inode 结构。在创建文件系统的时候，每个块组中的起始 inode 号以及 inode table 的起始地址都已经确定了，所以只要知道 inode 号，就能知道这个 inode 号和该块组起始 inode 号的偏移数量，再根据每个 inode 结构的大小(256 字节或其它大小)，就能计算出来对应的 inode 结构。

所以，目录的 data block 中的 inode number 和 inode table 中的 inode 是通过计算的方式一一映射起来的。从另一个角度上看，目录 data block 中的 inode number 是找到 inode table 中对应 inode 记录的唯一方式。

考虑一种比较特殊的情况：目录 data block 的记录已经删除，但是该记录对应的 inode 结构仍然存在于 inode table 中。这种 inode 称为孤儿 inode（orphan inode）：存在于 inode table 中，但却无法再索引到它。因为目录中已经没有该 inode 对应的文件记录了，所以其它进程将无法找到该 inode，也就无法根据该 inode 找到该文件之前所占用的 data block，这正是创建便删除所实现的真正临时文件，该临时文件只有当前进程和子进程才能访问。

4.3.3 符号链接存储方式

符号链接即为软链接，类似于 Windows 操作系统中的快捷方式，它的作用是指向原文件或目录。

软链接之所以也被称为特殊文件的原因是：它一般情况下不占用 data block，仅仅通过它对应的 inode 记录就能将其信息描述完成；符号链接的大小是其指向目标路径占用的字符个数，例如某个符号链接的指向方式为"rmt --> ../sbin/rmt"，则其文件大小为 11 字节；只有当符号链接指向的目标的路径名较长(60 个字节)时文件系统才会划分一个 data block 给它；它的权限如何也不重要，因它只是一个指向原文件的"工具"，最终决定是否能读写执行的权限由原文件决定，所以很可能 ls -l 查看到的符号链接权限为 777。

注意，软链接的 block 指针存储的是目标文件名。也就是说，链接文件的一切都依赖于其目标文件名。这就解释了为什么/mnt 的软链接/tmp/mnt 在/mnt 挂载文件系统后，通过软链接就能进入/mnt 所挂载的文件系统。究其原因，还是因为其目标文件名"/mnt"并没有改变。

例如以下筛选出了/etc/下的符号链接，注意观察它们的权限和它们的大小。

```
[root@xuexi ~]# ll /etc/ | grep '^l'
lrwxrwxrwx.  1 root root     56 Feb 18  2016 favicon.png -> /usr/share/icons/hicolor/16x16/apps/system-logo-icon.png
lrwxrwxrwx.  1 root root     22 Feb 18  2016 grub.conf -> ../boot/grub/grub.conf
lrwxrwxrwx.  1 root root     11 Feb 18  2016 init.d -> rc.d/init.d
lrwxrwxrwx.  1 root root      7 Feb 18  2016 rc -> rc.d/rc
lrwxrwxrwx.  1 root root     10 Feb 18  2016 rc0.d -> rc.d/rc0.d
lrwxrwxrwx.  1 root root     10 Feb 18  2016 rc1.d -> rc.d/rc1.d
lrwxrwxrwx.  1 root root     10 Feb 18  2016 rc2.d -> rc.d/rc2.d
lrwxrwxrwx.  1 root root     10 Feb 18  2016 rc3.d -> rc.d/rc3.d
lrwxrwxrwx.  1 root root     10 Feb 18  2016 rc4.d -> rc.d/rc4.d
lrwxrwxrwx.  1 root root     10 Feb 18  2016 rc5.d -> rc.d/rc5.d
lrwxrwxrwx.  1 root root     10 Feb 18  2016 rc6.d -> rc.d/rc6.d
lrwxrwxrwx.  1 root root     13 Feb 18  2016 rc.local -> rc.d/rc.local
lrwxrwxrwx.  1 root root     15 Feb 18  2016 rc.sysinit -> rc.d/rc.sysinit
lrwxrwxrwx.  1 root root     14 Feb 18  2016 redhat-release -> centos-release
lrwxrwxrwx.  1 root root     11 Apr 10  2016 rmt -> ../sbin/rmt
lrwxrwxrwx.  1 root root     14 Feb 18  2016 system-release -> centos-release
```

4.3.4 设备文件、FIFO、套接字文件

关于这 3 种文件类型的文件只需要通过 inode 就能完全保存它们的信息，它们不占用任何数据块，所以它们是特殊文件。

设备文件的主设备号和次设备号也保存在 inode 中。以下是/dev/下的部分设备信息。注意到它们的第 5 列和第 6 列信息，它们分别是主设备号和次设备号，主设备号标识每一种设备的类型，次设备号标识同种设备类型的不同编号；也注意到这些信息中没有大小的信息，因为设备文件不占用数据块所以没有大小的概念。

```
[root@xuexi ~]# ll /dev | tail
crw-rw---- 1 vcsa tty      7, 129 Oct  7 21:26 vcsa1
crw-rw---- 1 vcsa tty      7, 130 Oct  7 21:27 vcsa2
crw-rw---- 1 vcsa tty      7, 131 Oct  7 21:27 vcsa3
crw-rw---- 1 vcsa tty      7, 132 Oct  7 21:27 vcsa4
crw-rw---- 1 vcsa tty      7, 133 Oct  7 21:27 vcsa5
crw-rw---- 1 vcsa tty      7, 134 Oct  7 21:27 vcsa6
crw-rw---- 1 root root    10,  63 Oct  7 21:26 vga_arbiter
crw------- 1 root root    10,  57 Oct  7 21:26 vmci
crw-rw-rw- 1 root root    10,  56 Oct  7 21:27 vsock
crw-rw-rw- 1 root root     1,   5 Oct  7 21:26 zero
```

4.4 inode 基础知识

每个文件都有一个 inode，在将 inode 关联到文件后系统将通过 inode 号来识别文件，而不是文件名。并且访问文件时将先找到 inode，通过 inode 中记录的 block 位置找到该文件。

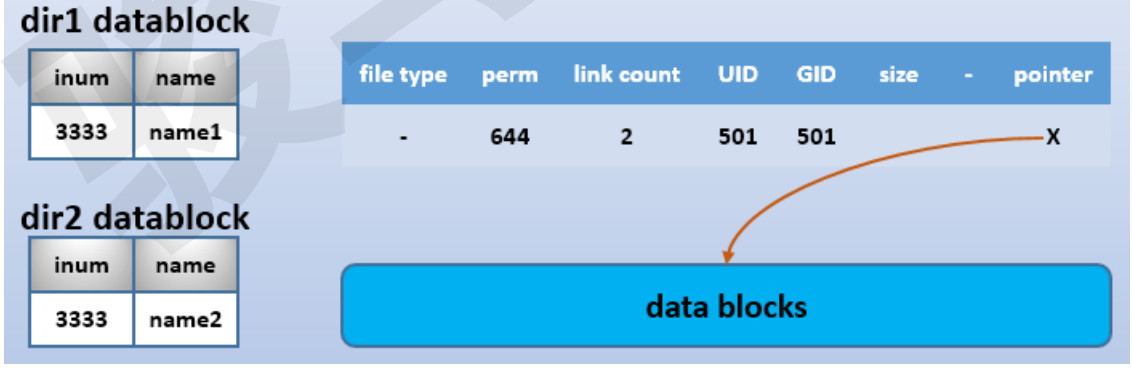
4.4.1 硬链接

虽然每个文件都有一个 inode，但是存在一种可能：多个文件的 inode 相同，也就即 inode 号、元数据、block 位置都相同，这是一种什么样的情况呢？能够想象这些 inode 相同的文件使用的都是同一条 inode 记录，所以代表的都是同一个文件，这些文件所在目录的 data block 中的 inode 号都是一样的，只不过各 inode 号对应的文件名互不相同而已。这种 inode 相同的文件在 Linux 中被称为"硬链接"。

硬链接文件的 inode 都相同，每个文件都有一个"硬链接数"的属性，使用 ls -l 的第二列就是被硬链接数，它表示的就是该文件有几个硬链接。

```
[root@xuexi ~]# ls -l
total 48
drwxr-xr-x  5 root root  4096 Oct 15 18:07 700
-rw-------. 1 root root  1082 Feb 18  2016 anaconda-ks.cfg
-rw-r--r--  1 root root   399 Apr 29  2016 Identity.pub
-rw-r--r--. 1 root root 21783 Feb 18  2016 install.log
-rw-r--r--. 1 root root  6240 Feb 18  2016 install.log.syslog
```

例如下图描述的是 dir1 目录中的文件 name1 及其硬链接 dir2/name2，右边分别是它们的 inode 和 data block。这里也看出了硬链接文件之间唯一不同的就是其所在目录中的记录不同。注意下图中有一列 Link Count 就是标记硬链接数的属性。



每创建一个文件的硬链接，实质上是多一个指向该 inode 记录的 inode 指针，并且硬链接数加 1。

删除文件的实质是删除该文件所在目录 data block 中的对应的 inode 行，所以也是减少硬链接次数，由于 block 指针是存储在 inode 中的，所以不是真的删除数据，如果仍有其他 inode 号链接到该 inode，那么该文件的 block 指针仍然是可用的。当硬链接次数为 1 时再删除文件就是真的删除文件了，此时 inode 记录中 block 指针也将被删除。

不能跨分区创建硬链接，因为不同文件系统的 inode 号可能会相同，如果允许创建硬链接，复制到另一个分区时 inode 可能会和此分区已使用的 inode 号冲突。

硬链接只能对文件创建，无法对目录创建硬链接。之所以无法对目录创建硬链接，是因为文件系统已经把每个目录的硬链接创建好了，它们就是相对路径中的"."和".."，分别标识当前目录的硬链接和上级目录的硬链接。每一个目录中都会包含这两个硬链接，它包含了两个信息：(1)一个没有子目录的目录文件的硬链接数是 2，其一是目录本身，其二是"."；(2)一个包含子目录的目录文件，其硬链接数是 2+子目录数，因为每个子目录都关联一个父目录的硬链接".."。很多人在计算目录的硬链接数时认为由于包含了"."和"..."，所以空目录的硬链接数是 2，这是错误的，因为".."不是本目录的硬链接。另外，还有一个特殊的目录应该纳入考虑，即"/"目录，它自身是一个文件系统的入口，是自引用(下文中会解释自引用)的，所以"/"目录下的"."和".."的 inode 号相同，它自身不占用硬链接，因为其 datablock 中只记录 inode 号相同的"."和"..."，不再像其他目录一样还记录一个名为"/"的目录，所以"/"的硬链接数也是 2+子目录数，但这个 2 是"."和".."的结果。

```
[root@xuexi ~]# ln /tmp /mydata
ln: `/tmp': hard link not allowed for directory
```

为什么文件系统自己创建好了目录的硬链接就不允许人为创建呢？从"."和".."的用法上考虑，如果当前目录为/usr，我们可以使用"./local"来表示/usr/local，但是如果我们人为创建了/usr 目录的硬链接/tmp/husr，难道我们也要使用"/tmp/husr/local"来表示/usr/local 吗？这其实已经是软链接的作用了。若要将其认为是硬链接的功能，这必将导致硬链接维护的混乱。

不过，通过 mount 工具的"--bind"选项，可以将一个目录挂载到另一个目录下，实现伪"硬链接"，它们的内容和 inode 号是完全相同的。

硬链接的创建方法：ln file_target link_name。

4.4.2 软链接

软链接就是字符链接，链接文件默认指的就是字符文件，使用"l"表示其类型。

硬链接不能跨文件系统创建，否则 inode 号可能会冲突。于是实现了软链接以便跨文件系统建立链接。既然是跨文件系统，那么软链接必须得有自己的 inode 号。

软链接在功能上等价与 Windows 系统中的快捷方式，它指向原文件，原文件损坏或消失，软链接文件就损坏。可以认为软链接 inode 记录中的指针内容就是目标路径的字符串。

创建方式：ln -s source_file softlink_name

查看软链接的值：readlink softlink_name

在设置软链接的时候，source_file 虽然不要求是绝对路径，但建议给绝对路径。是否还记得软链接文件的大小？它是根据软链接所指向路径的字符数计算的，例如某个符号链接的指向方式为"rmt --> ../sbin/rmt"，它的文件大小为 11 字节，也就是说只要建立了软链接后，软链接的指向路径是不会改变的，仍然是"../sbin/rmt"。如果此时移动软链接文件本身，它的指向是不会改变的，仍然是 11 个字符的"../sbin/rmt"，但此时该软链接父目录下可能根本就不存在/sbin/rmt，也就是说此时该软链接是一个被破坏的软链接。

4.5 inode 深入

4.5.1 inode 大小和划分

inode 大小为 128 字节的倍数，最小为 128 字节。它有默认值大小，它的默认值由/etc/mke2fs.conf 文件中指定。不同的文件系统默认值可能不同。

```
[root@xuexi ~]# cat /etc/mke2fs.conf
[defaults]
        base_features = sparse_super,filetype,resize_inode,dir_index,ext_attr
        enable_periodic_fsck = 1
        blocksize = 4096
        inode_size = 256
        inode_ratio = 16384

[fs_types]
        ext3 = {
                features = has_journal
        }
        ext4 = {
                features = has_journal,extent,huge_file,flex_bg,uninit_bg,dir_nlink,extra_isize
                inode_size = 256
        }
```

同样观察到这个文件中还记录了 blocksize 的默认值和 inode 分配比率 inode_ratio。inode_ratio=16384 表示每 16384 个字节即 16KB 就分配一个 inode 号，由于默认 blocksize=4KB，所以每 4 个 block 就分配一个 inode 号。当然分配的这些 inode 号只是预分配，并不真的代表会全部使用，毕竟每个文件才会分配一个 inode 号。但是分配的 inode 自身会占用 block，而且其自身大小 256 字节还不算小，所以 inode 号的浪费代表着空间的浪费。

既然知道了 inode 分配比率，就能计算出每个块组分配多少个 inode 号，也就能计算出 inode table 占用多少个 block。

如果文件系统中大量存储电影等大文件，inode 号就浪费很多，inode 占用的空间也浪费很多。但是没办法，文件系统又不知道你这个文件系统是用来存什么样的数据，多大的数据，多少数据。

当然 inode size、inode 分配比例、block size 都可以在创建文件系统的时候人为指定。

4.5.2 ext 文件系统预留的 inode 号

Ext 预留了一些 inode 做特殊特性使用，如下：某些可能并非总是准确，具体的 inode 号对应什么文件可以使用"find / -inum NUM"查看。

```
Ext4 的特殊 inode
Inode 号      用途
0         不存在 0 号 inode，可用于标识目录 data block 中已删除的文件
1         虚拟文件系统，如/proc 和/sys
2         根目录
3         ACL 索引
4         ACL 数据
5         Boot loader
```

```
6       未删除的目录
7       预留的块组描述符 inode
8       日志 inode
11      第一个非预留的 inode，通常是 lost+found 目录
```

所以在 ext4 文件系统的 dumpe2fs 信息中，能观察到 fisrt inode 号可能为 11 也可能为 12。

并且注意到"/"的 inode 号为 2，这个特性在文件访问时会用上。

需要注意的是，每个文件系统都会分配自己的 inode 号，不同文件系统之间是可能会出现使用相同 inode 号文件的。例如：

```
[root@xuexi ~]# find / -ignore_readdir_race -inum 2 -ls
     2    4 dr-xr-xr-x  22 root     root         4096 Jun  9 09:56 /
     2    2 dr-xr-xr-x   5 root     root         1024 Feb 25 11:53 /boot
     2    0 c---------   1 root     root              Jun  7 02:13 /dev/pts/ptmx
     2    0 -rw-r--r--   1 root     root            0 Jun  6 18:13 /proc/sys/fs/binfmt_misc/status
     2    0 drwxr-xr-x   3 root     root            0 Jun  6 18:13 /sys/fs
```

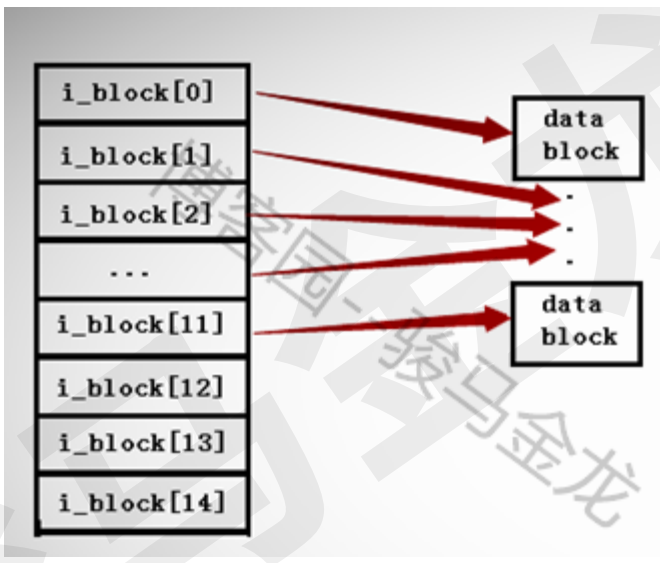
从结果中可见，除了根的 Inode 号为 2，还有几个文件的 inode 号也是 2，它们都属于独立的文件系统，有些是虚拟文件系统，如/proc 和/sys。

4.5.3 ext2/3 的 inode 直接、间接寻址

前文说过，inode 中保存了 blocks 指针，但是一条 inode 记录中能保存的指针数量是有限的，否则就会超出 inode 大小(128 字节或 256 字节)。

在 ext2 和 ext3 文件系统中，一个 inode 中最多只能有 15 个指针，每个指针使用 i_block[n]表示。

前 12 个指针 i_block[0]到 i_block[11]是直接寻址指针，每个指针指向一个数据区的 block。如下图所示。



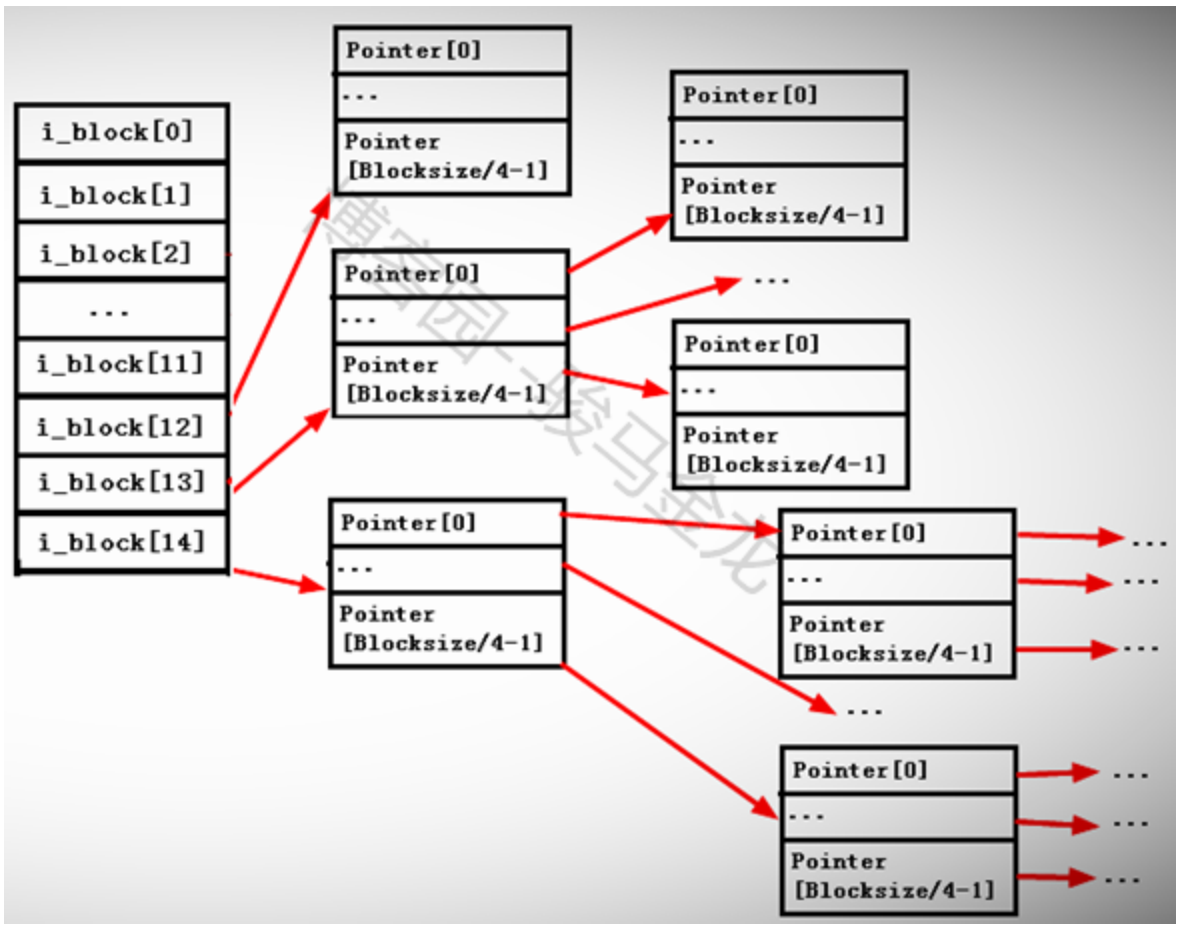
第 13 个指针 i_block[12]是一级间接寻址指针，它指向一个仍然存储了指针的 block 即 i_block[12] --> Pointerblock --> datablock。

第 14 个指针 i_block[13]是二级间接寻址指针，它指向一个仍然存储了指针的 block，但是这个 block 中的指针还继续指向其他存储指针的 block，即 i_block[13] --> Pointerblock1 --> PointerBlock2 --> datablock。

第 15 个指针 i_block[14]是三级间接寻址指针，它指向一个任然存储了指针的 block，这个指针 block 下还有两次指针指向。即 i_block[13] --> Pointerblock1 --> PointerBlock2 --> PointerBlock3 --> datablock。

其中由于每个指针大小为 4 字节，所以每个指针 block 能存放的指针数量为 BlockSize/4byte。例如 blocksize 为 4KB，那么一个 Block 可以存放 4096/4=1024 个指针。

如下图。



为什么要分间接和直接指针呢？如果一个 inode 中 15 个指针全是直接指针，假如每个 block 的大小为 1KB，那么 15 个指针只能指向 15 个 block 即 15KB 的大小，由于每个文件对应一个 inode 号，所以就限制了每个文件最大为 15*1=15KB，这显然是不合理的。

如果存储大于 15KB 的文件而又不太大的时候，就占用一级间接指针 i_block[12]，这时可以存放指针数量为 1024/4+12=268，所以能存放 268KB 的文件。

如果存储大于 268K 的文件而又不太大的时候，就继续占用二级指针 i_block[13]，这时可以存放指针数量为[1024/4]^2+1024/4+12=65804，所以能存放 65804KB=64M 左右的文件。

如果存放的文件大于 64M，那么就继续使用三级间接指针 i_block[14]，存放的指针数量为[1024/4]^3+[1024/4]^2+[1024/4]+12=16843020 个指针，所以能存放 16843020KB=16GB 左右的文件。

如果 blocksize=4KB 呢？那么最大能存放的文件大小为([4096/4]^3+[4096/4]^2+[4096/4]+12)*4/1024/1024/1024=4T 左右。

当然这样计算出来的不一定就是最大能存放的文件大小，它还受到另一个条件的限制。这里的计算只是表明一个大文件是如何寻址和分配的。

其实看到这里的计算数值，就知道 ext2 和 ext3 对超大文件的存取效率是低下的，它要核对太多的指针，特别是 4KB 大小的 blocksize 时。而 ext4 针对这一点就进行了优化，ext4 使用 extent 的管理方式取代 ext2 和 ext3 的块映射，大大提高了效率也降低了碎片。

4.6 单文件系统中文件操作的原理

在 Linux 上执行删除、复制、重命名、移动等操作时，它们是怎么进行的呢？还有访问文件时是如何找到它的呢？其实只要理解了前文中介绍的几个术语以及它们的作用就很容易知道文件操作的原理了。

注：在这一小节所解释的都是在单个文件系统下的行为，在多个文件系统中如何请看下一个小节：多文件系统关联。

4.6.1 读取文件

当执行"cat /var/log/messages"命令在系统内部进行了什么样的步骤呢？该命令能被成功执行涉及了 cat 命令的寻找、权限判断以及 messages 文件的寻找和权限判断等等复杂的过程。这里只解释和本节内容相关的如何寻找到被 cat 的/var/log/messages 文件。

- **找到根文件系统的块组描述符表所在的 blocks，读取 GDT(已在内存中)找到 inode table 的 block 号。**

因为 GDT 总是和 superblock 在同一个块组，而 superblock 总是在分区的第 1024-2047 个字节，所以很容易就知道第一个 GDT 所在的块组以及 GDT 在这个块组中占用了哪些 block。

其实 GDT 早已经在内存中了，在系统开机的时候会挂载根文件系统，挂载的时候就已经将所有的 GDT 放进内存中。

- **在 inode table 的 block 中定位到根"/"的 inode，找出"/"指向的 data block。**

前文说过，ext 文件系统预留了一些 inode 号，其中"/"的 inode 号为 2，所以可以根据 inode 号直接定位根目录文件的 data block。

- **在"/"的 datablock 中记录了 var 目录名和 var 的 inode 号，找到该 inode 记录，inode 记录中存储了指向 var 的 block 指针，所以也就找到了 var 目录文件的 data block。**

通过 var 目录的 inode 号，可以寻找到 var 目录的 inode 记录，但是在寻找的过程中，还需要知道该 inode 记录所在的块组以及所在的 inode table，所以需要读取 GDT，同样，GDT 已经缓存到了内存中。

- **在 var 的 data block 中记录了 log 目录名和其 inode 号，通过该 inode 号定位到该 inode 所在的块组及所在的 inode table，并根据该 inode 记录找到 log 的 data block。**

- **在 log 目录文件的 data block 中记录了 messages 文件名和对应的 inode 号，通过该 inode 号定位到该 inode 所在的块组及所在的 inode table，并根据该 inode 记录找到 messages 的 data block。**

- **最后读取 messages 对应的 datablock。**

将上述步骤中 GDT 部分的步骤简化后比较容易理解。如下:找到 GDT-->找到/的 inode-->找到/的数据块读取 var 的 inode-->找到 var 的数据块读取 log 的 inode-->找到 log 的数据块读取 messages 的 inode-->找到 messages 的数据块并读取它们。

当然，在每次定位到 inode 记录后，都会先将 inode 记录加载到内存中，然后查看权限，如果权限允许，将根据 block 指针找到对应的 data block。

4.6.2 删除、重命名和移动文件

注意这里是不跨越文件系统的操作行为。

- **删除文件分为普通文件和目录文件，知道了这两种类型的文件的删除原理，就知道了其他类型特殊文件的删除方法。**

对于删除普通文件：(1)找到文件的 inode 和 data block(根据前一个小节中的方法寻找)；(2)将 inode table 中该 inode 记录中的 data block 指针删除；(3)在 imap 中将该文件的 inode 号标记为未使用；(4)在其所在目录的 data block 中将该文件名所在的记录行删除，删除了记录就丢失了指向 inode 的指针（实际上不是真的删除，直接删除的话会在目录 data block 的数据结构中产生空洞，所以实际的操作是将待删除文件的 inode 号设置为特殊的值 0，这样下次新建文件时就可以重用该行记录)；(5)将 bmap 中 data block 对应的 block 号标记为未使用。

对于删除目录文件：找到目录和目录下所有文件、子目录、子文件的 inode 和 data block；在 imap 中将这些 inode 号标记为未使用；将 bmap 中将这些文件占用的 block 号标记为未使用；在该目录的父目录的 data block 中将该目录名所在的记录行删除。需要注意的是，删除父目录 data block 中的记录是最后一步，如果该步骤提前，将报目录非空的错误，因为在该目录中还有文件占用。

关于上面的(2)-(5)：当(2)中删除 data block 指针后，将无法再找到这个文件的数据；当(3)标记 inode 号未使用，表示该 inode 号可以被后续的文件重用；当(4)删除目录 data block 中关于该文件的记录，真正的删除文件，外界再也定位也无法看到这个文件了；当(5)标记 data block 为未使用后，表示开始释放空间，这些 data block 可以被其他文件重用。

注意，在第(5)步之前，由于 data block 还未被标记为未使用，在 superblock 中仍然认为这些 data block 是正在使用中的。这表示尽管文件已经被删除了，但空间却还没有释放，df 也会将其统计到已用空间中(df 是读取 superblock 中的数据块数量，并计算转换为空间大小)。

什么时候会发生这种情况呢？当一个进程正在引用文件时将该文件删除，就会出现文件已删除但空间未释放的情况。这时步骤已经进行到(4)，外界无法再找到该文件，但由于进程在加载该文件时已经获取到了该文件所有的 data block 指针，该进程可以获取到该文件的所有数据，但却暂时不会释放该文件空间。直到该进程结束，文件系统才将未执行的步骤(5)继续完成。这也是为什么有时候 du 的统计结果比 df 小的原因，关于 du 和 df 统计结果的差别，详细内容见：详细分析 du 和 df 的统计结果为什么不一样。

- **重命名文件分为同目录内重命名和非同目录内重命名。非同目录内重命名实际上是移动文件的过程，见下文。**

同目录内重命名文件的动作仅仅只是修改所在目录 data block 中该文件记录的文件名部分，不是删除再重建的过程。

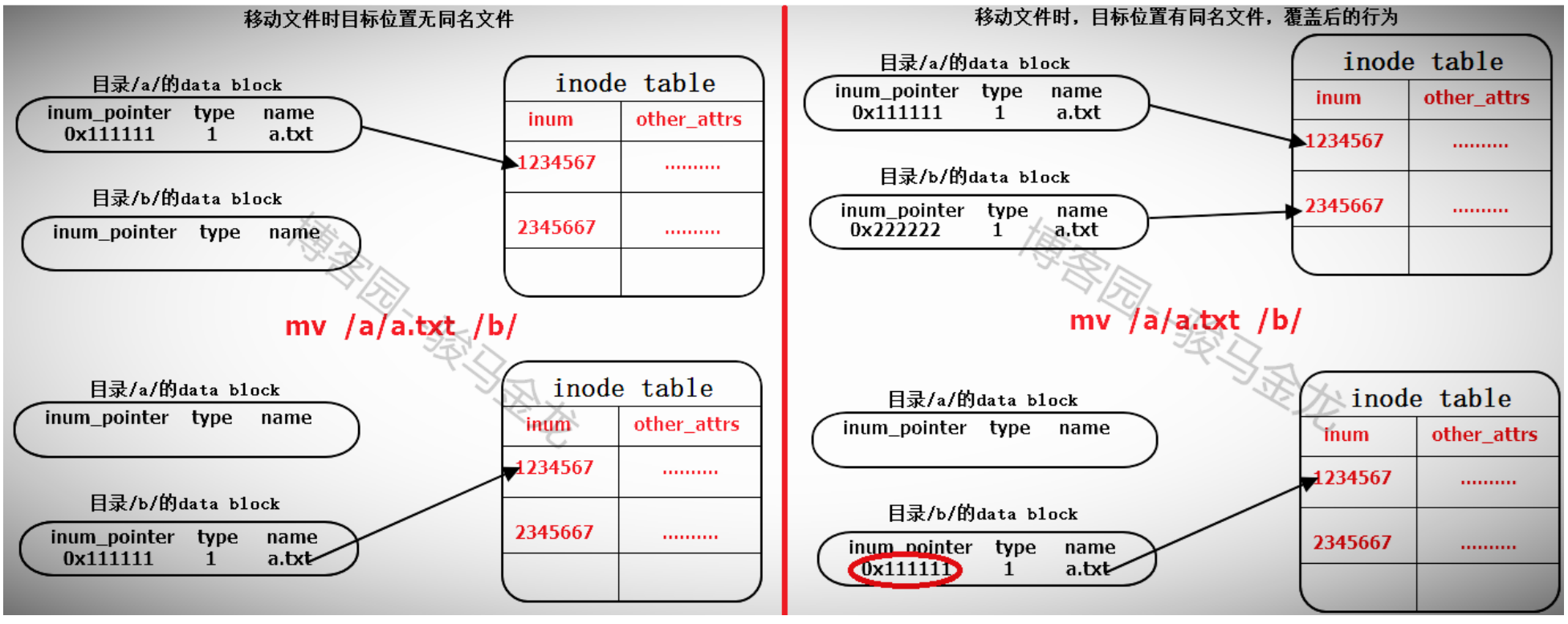
如果重命名时有文件名冲突(该目录内已经存在该文件名)，则提示是否覆盖。覆盖的过程是覆盖目录 data block 中冲突文件的记录。例如/tmp/下有 a.txt 和 a.log，若将 a.txt 重命名为 a.log，则提示覆盖，若选择覆盖，则/tmp 的 data block 中关于 a.log 的记录被覆盖。

- **移动文件**

同文件系统下移动文件实际上是修改目标文件所在目录的 data block，向其中添加一行指向 inode table 中待移动文件的 inode 记录，如果目标路径下有同名文件，则会提示是否覆盖，实际上是覆盖目录 data block 中冲突文件的记录，由于同名文件的 inode 记录指针被覆盖，所以无法再找到该文件的 data block，也就是说该文件被标记为删除(如果多个硬链接数，则另当别论)。

所以在同文件系统内移动文件相当快，仅仅在所在目录 data block 中添加或覆盖了一条记录而已。也因此，移动文件时，文件的 inode 号是不会改变的。

对于不同文件系统内的移动，相当于先复制再删除的动作。见后文。



4.6.3 存储和复制文件

- **对于文件存储**

(1).读取 GDT，找到各个(或部分)块组 imap 中未使用的 inode 号，并为待存储文件分配 inode 号；

(2).在 inode table 中完善该 inode 号所在行的记录；

(3).在目录的 data block 中添加一条该文件的相关记录；

(4).将数据填充到 data block 中。

注意，填充到 data block 中的时候会调用 block 分配器：一次分配 4KB 大小的 block 数量，当填充完 4KB 的 data block 后会继续调用 block 分配器分配 4KB 的 block，然后循环直到填充完所有数据。也就是说，如果存储一个 100M 的文件需要调用 block 分配器 100*1024/4=25600 次。

另一方面，在 block 分配器分配 block 时，block 分配器并不知道真正有多少 block 要分配，只是每次需要分配时就分配，在每存储一个 data block 前，就去 bmap 中标记一次该 block 已使用，它无法实现一次标记多个 bmap 位。这一点在 ext4 中进行了优化。

(5)填充完之后，去 inode table 中更新该文件 inode 记录中指向 data block 的寻址指针。

- **对于复制，完全就是另一种方式的存储文件。步骤和存储文件的步骤一样。**

4.7 多文件系统关联

在单个文件系统中的文件操作和多文件系统中的操作有所不同。本文将对此做出非常详细的说明。

4.7.1 根文件系统的特殊性

这里要明确的是，任何一个文件系统要在 Linux 上能正常使用，必须挂载在某个已经挂载好的文件系统中的某个目录下，例如/dev/cdrom 挂载在/mnt 上，/mnt 目录本身是在"/"文件系统下的。而且任意文件系统的一级挂载点必须是在根文件系统的某个目录下，因为只有"/"是自引用的。这里要说明挂载点的级别和自引用的概念。

假如/dev/sdb1 挂载在/mydata 上，/dev/cdrom 挂载在/mydata/cdrom 上，那么/mydata 就是一级挂载点，此时/mydata 已经是文件系统/dev/sdb1 的入口了，而/dev/cdrom 所挂载的目录/mydata/cdrom 是文件系统/dev/sdb1 中的某个目录，那么/mydata/cdrom 就是二级挂载点。一级挂载点必须在根文件系统下，所以可简述为：文件系统 2 挂载在文件系统 1 中的某个目录下，而文件系统 1 又挂载在根文件系统中的某个目录下。

再解释自引用。首先要说的是，自引用的只能是文件系统，而文件系统表现形式是一个目录，所以自引用是指该目录的 data block 中，"."和".."的记录中的 inode 号都对应 inode table 中同一个 inode 记录，所以它们 inode 号是相同的，即互为硬链接。而根文件系统是唯一可以自引用的文件系统。

```
[root@xuexi /]# ll -ai /
total 102
     2 dr-xr-xr-x.  22 root root  4096 Jun  6 18:13 .
     2 dr-xr-xr-x.  22 root root  4096 Jun  6 18:13 ..
```

由此也能解释 cd /.和 cd /..的结果都还是在根下，这是自引用最直接的表现形式。

```
[root@xuexi tmp]# cd /.
[root@xuexi /]#
[root@xuexi tmp]# cd /..
[root@xuexi /]#
```

注意，根目录下的"."和".."都是"/"目录的硬链接，且其 datablock 中不记录名为"/"的条目，因此除去根目录下子目录数后的硬链接数为 2。

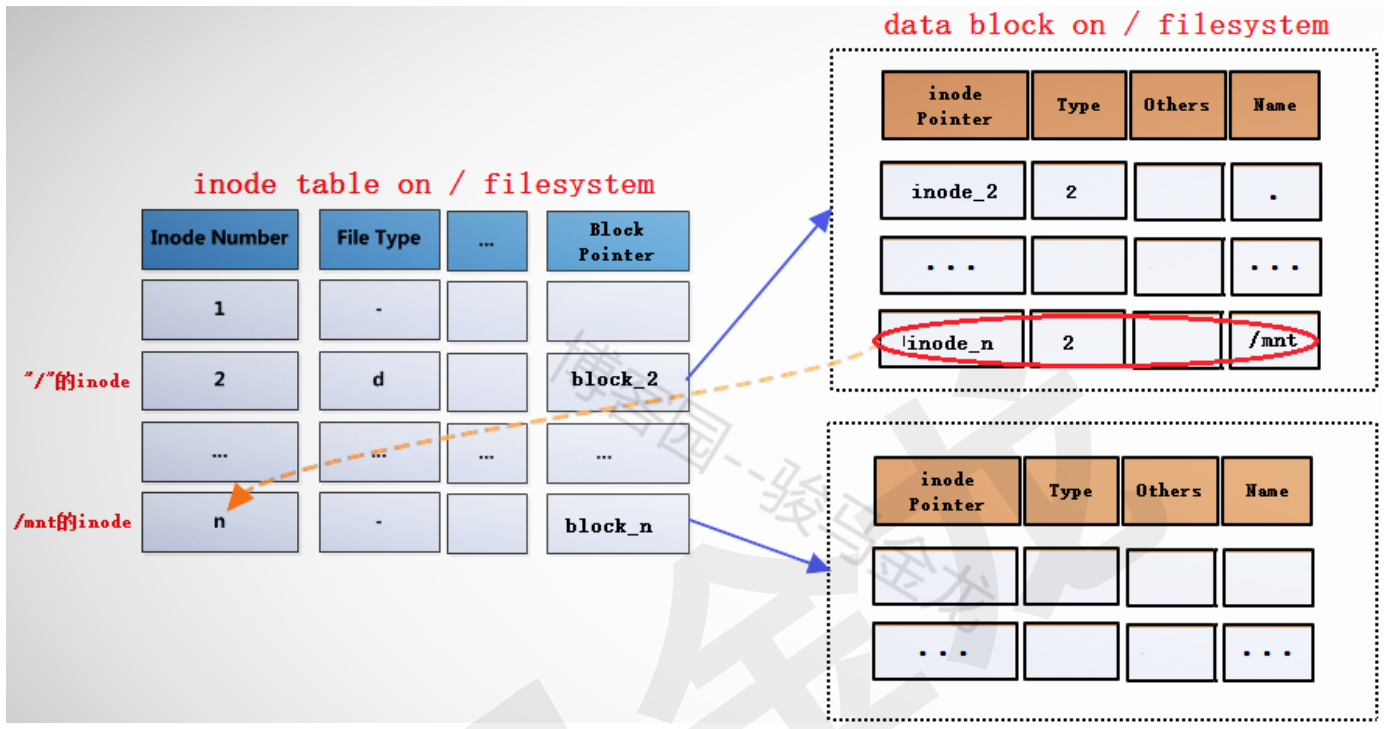
```
[root@server2 tmp]# a=$(ls -ld / | awk '{print $2}')
[root@server2 tmp]# b=$(ls -l / | grep "^d" |wc -l)
[root@server2 tmp]# echo $((a - b))
2
```

4.7.2 挂载文件系统的细节

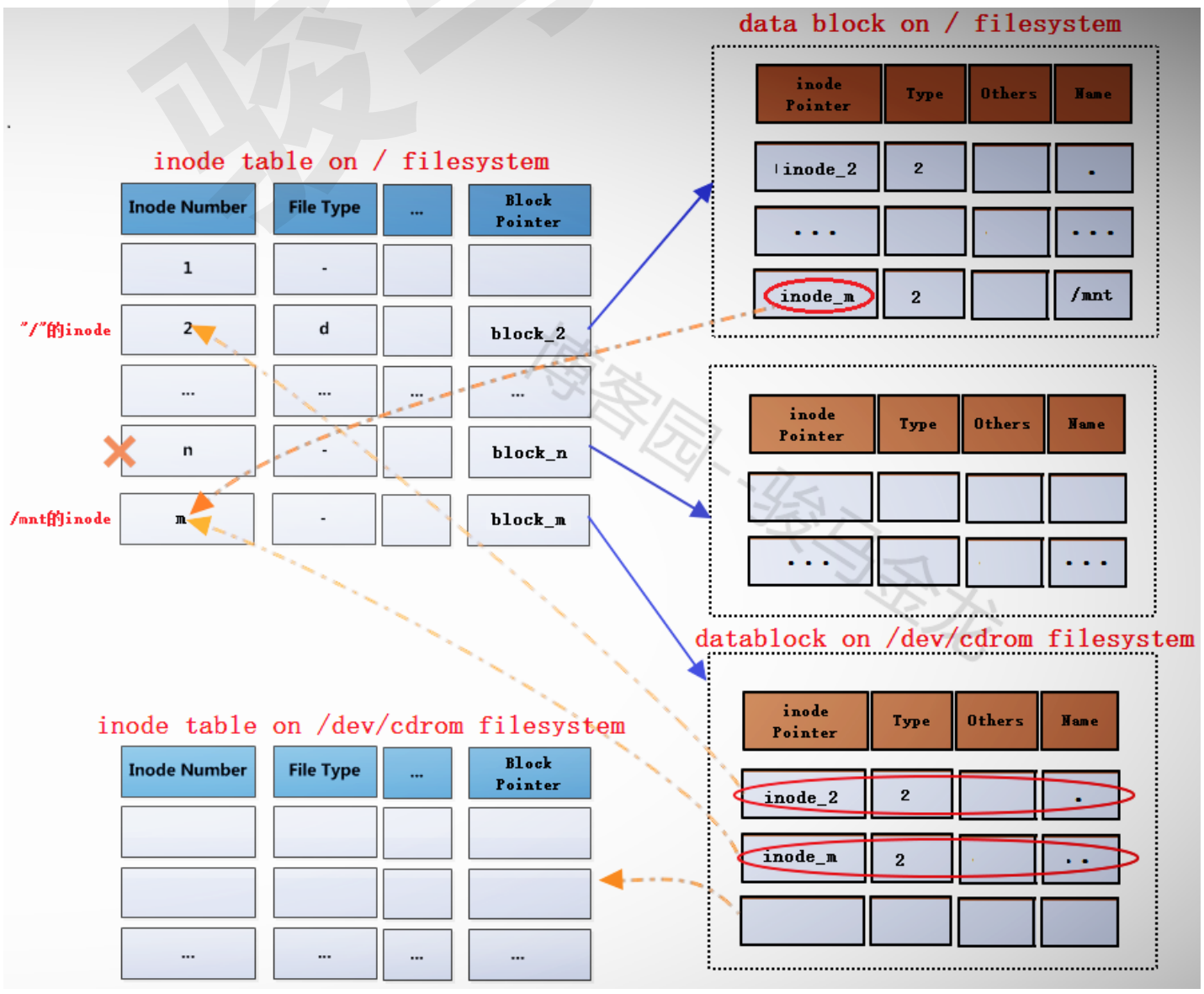
挂载文件系统到某个目录下，例如"mount /dev/cdrom /mnt"，挂载成功后/mnt 目录中的文件全都暂时不可见了，且挂载后权限和所有者(如果指定允许普通用户挂载)等的都改变了，知道为什么吗？

下面就以通过"mount /dev/cdrom /mnt"为例，详细说明挂载过程中涉及的细节。

在将文件系统/dev/cdrom(此处暂且认为它是文件系统)挂载到挂载点/mnt 之前，挂载点/mnt 是根文件系统中的一个目录，"/"的 data block 中记录了/mnt 的一些信息，其中包括 inode 号 inode_n，而在 inode table 中，/mnt 对应的 inode 记录中又存储了 block 指针 block_n，此时这两个指针还是普通的指针。



当文件系统/dev/cdrom 挂载到/mnt 上后，/mnt 此时就已经成为另一个文件系统的入口了，因此它需要连接两边文件系统的 inode 和 data block。但是如何连接呢？如下图。



在根文件系统的 inode table 中，为/mnt 重新分配一个 inode 记录 m，该记录的 block 指针 block_m 指向文件系统/dev/cdrom 中的 data block。既然为/mnt 分配了新的 inode 记录 m，那么在"/"目录的 data block 中，也需要修改其 inode 指针为 inode_m 以指向 m 记录。同时，原来 inode table 中的 inode 记录 n 就被标记为暂时不可用。

block_m 指向的是文件系统/dev/cdrom 的 data block，所以严格说起来，除了/mnt 的元数据信息即 inode 记录 m 还在根文件系统上，/mnt 的 data block 已经是在/dev/cdrom 中的了。这就是挂载新文件系统后实现的跨文件系统，它将挂载点的元数据信息和数据信息分别存储在不同的文件系统上。

挂载完成后，将在/proc/self/{mounts,mountstats,mountinfo}这三个文件中写入挂载记录和相关的挂载信息，并会将/proc/self/mounts 中的信息同步到/etc/mtab 文件中，当然，如果挂载时加了-n 参数，将不会同步到/etc/mtab。

而卸载文件系统，其实质是移除临时新建的 inode 记录(当然，在移除前会检查是否正在使用)及其指针，并将指针指回原来的 inode 记录，这样 inode 记录中的 block 指针也就同时生效而找回对应的 data block 了。由于卸载只是移除 inode 记录，所以使用挂载点和文件系统都可以实现卸载，因为它们是联系在一起的。

下面是分析或结论。

(1).挂载点挂载时的 inode 记录是新分配的。

```
# 挂载前挂载点/mnt 的 inode 号
[root@server2 tmp]# ll -id /mnt
100663447 drwxr-xr-x. 2 root root 6 Aug 12  2015 /mnt
[root@server2 tmp]# mount /dev/cdrom /mnt
# 挂载后挂载点的 inode 号
[root@server2 tmp]# ll -id /mnt
1856 dr-xr-xr-x    8 root root  2048 Dec 10  2015 mnt
```

由此可以验证，inode 号确实是重新分配的。

(2).挂载后，挂载点的内容将暂时不可见、不可用，卸载后文件又再次可见、可用。

```
# 在挂载前，向挂载点中创建几个文件
[root@server2 tmp]# touch /mnt/a.txt
[root@server2 tmp]# mkdir /mnt/abcdir
# 挂载
[root@server2 tmp]# mount /dev/cdrom /mnt
# 挂载后，挂载点中将找不到刚创建的文件
[root@server2 tmp]# ll /mnt
total 636
-r--r--r-- 1 root root     14 Dec 10  2015 CentOS_BuildTag
dr-xr-xr-x 3 root root   2048 Dec 10  2015 EFI
-r--r--r-- 1 root root    215 Dec 10  2015 EULA
-r--r--r-- 1 root root  18009 Dec 10  2015 GPL
dr-xr-xr-x 3 root root   2048 Dec 10  2015 images
dr-xr-xr-x 2 root root   2048 Dec 10  2015 isolinux
dr-xr-xr-x 2 root root   2048 Dec 10  2015 LiveOS
dr-xr-xr-x 2 root root 612352 Dec 10  2015 Packages
dr-xr-xr-x 2 root root   4096 Dec 10  2015 repodata
-r--r--r-- 1 root root   1690 Dec 10  2015 RPM-GPG-KEY-CentOS-7
-r--r--r-- 1 root root   1690 Dec 10  2015 RPM-GPG-KEY-CentOS-Testing-7
-r--r--r-- 1 root root   2883 Dec 10  2015 TRANS.TBL
# 卸载后，挂载点/mnt 中的文件将再次可见
[root@server2 tmp]# umount /mnt
[root@server2 tmp]# ll /mnt
total 0
drwxr-xr-x 2 root root 6 Jun  9 08:18 abcdir
-rw-r--r-- 1 root root 0 Jun  9 08:18 a.txt
```

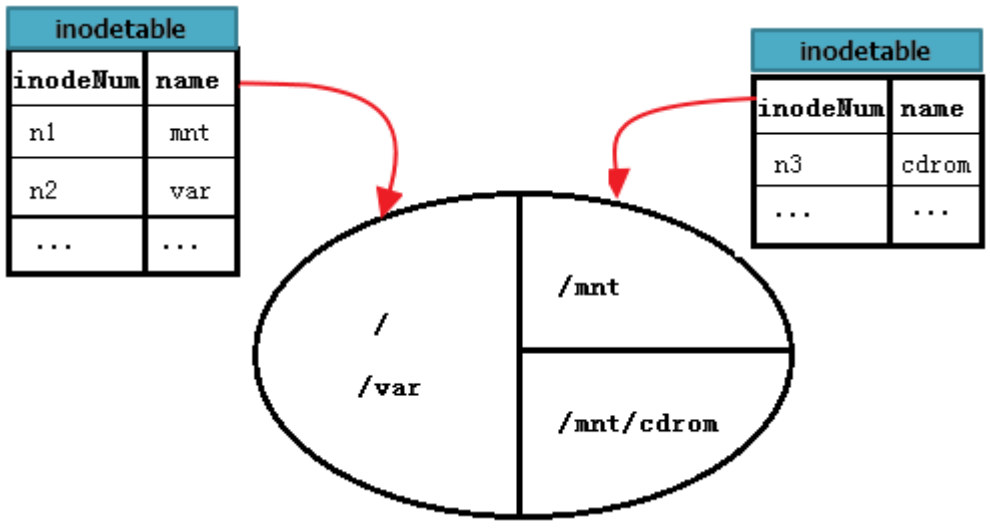
之所以会这样，是因为挂载文件系统后，挂载点原来的 inode 记录暂时被标记为不可用，关键是没有指向该 inode 记录的 inode 指针了。在卸载文件系统后，又重新启用挂载点原来的 inode 记录，"/"目录下的 mnt 的 inode 指针又重新指向该 inode 记录。

(3).挂载后，挂载点的元数据和 data block 是分别存放在不同文件系统上的。

(4).挂载点即使在挂载后，也还是属于源文件系统的文件。

4.7.3 多文件系统操作关联

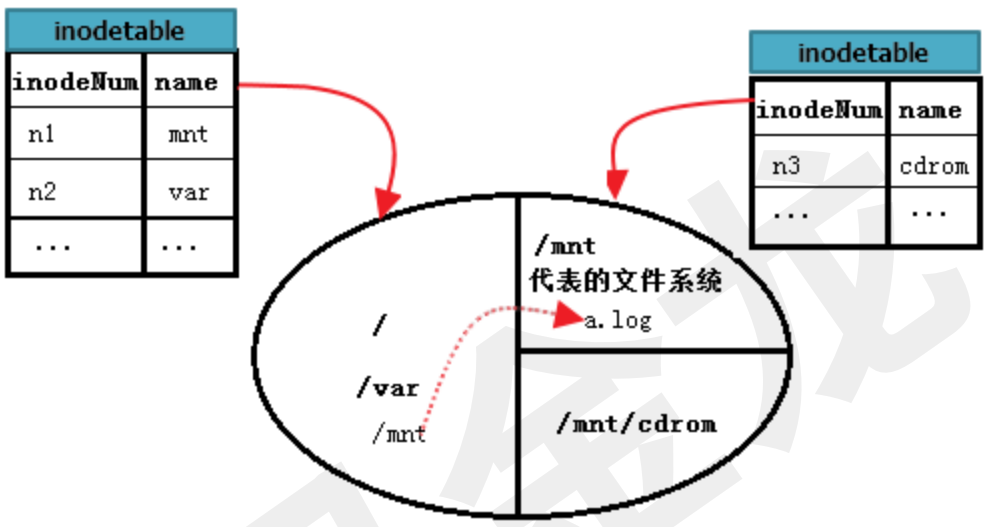
假如下图中的圆代表一块硬盘，其中划分了 3 个区即 3 个文件系统。其中根是根文件系统，/mnt 是另一个文件系统 A 的入口，A 文件系统挂载在/mnt 上，/mnt/cdrom 也是一个文件系统 B 的入口，B 文件系统挂载在/mnt/cdrom 上。每个文件系统都维护了一些 inode table，这里假设图中的 inode table 是每个文件系统所有块组中的 inode table 的集合表。



如何读取/var/log/messages 呢？这是和"/"在同一个文件系统的文件读取，在前面单文件系统中已经详细说明了。

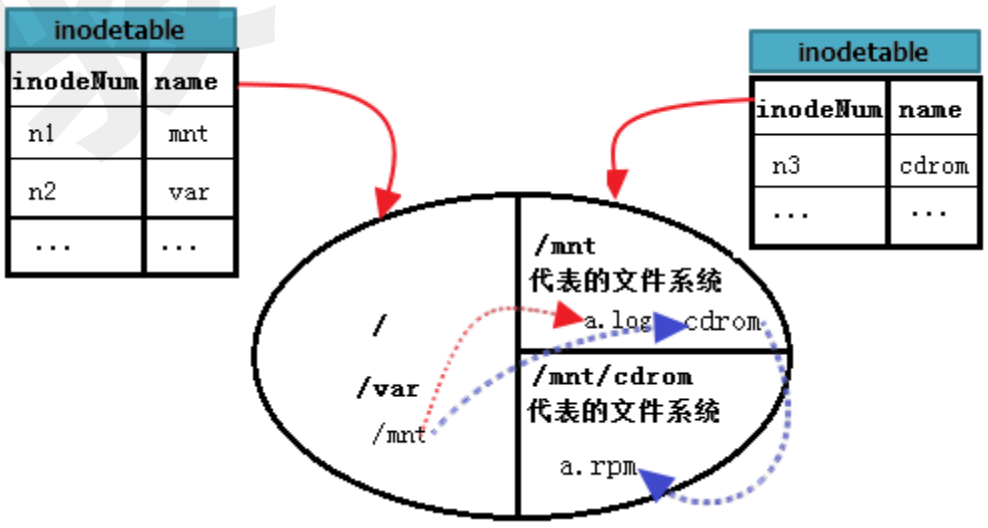
但如何读取 A 文件系统中的/mnt/a.log 呢？首先，从根文件系统找到/mnt 的 inode 记录，这是单文件系统内的查找；然后根据此 inode 记录的 block 指针，定位到/mnt 的 data block 中，这些 block 是 A 文件系统的 data block；然后从/mnt 的 data block 中读取 a.log 记录，并根据 a.log 的 inode 指针定位到 A 文件系统的 inode table 中对应 a.log 的 inode 记录；最后从此 inode 记录的 block 指针找到 a.log 的 data block。至此，就能读取到/mnt/a.log 文件的内容。

下图能更完整的描述上述过程。



那么又如何读取/mnt/cdrom 中的/mnt/cdrom/a.rpm 呢？这里 cdrom 代表的文件系统 B 挂载点位于/mnt 下，所以又多了一个步骤。先找到"/"，再找到根中的 mnt，进入到 mnt 文件系统中，找到 cdrom 的 data block，再进入到 cdrom 找到 a.rpm。也就是说，mnt 目录文件存放位置是根，cdrom 目录文件存放位置是 mnt，最后 a.rpm 存放的位置才是 cdrom。

继续完善上图。如下。



4.8 ext3 文件系统的日志功能

相比 ext2 文件系统，ext3 多了一个日志功能。

在 ext2 文件系统中，只有两个区：数据区和元数据区。如果正在向 data block 中填充数据时突然断电，那么下一次启动时就会检查文件系统中数据和状态的一致性，这段检查和修复可能会消耗大量时间，甚至检查后无法修复。之所以会这样是因为文件系统在突然断电后，它不知道上次正在存储的文件的 block 从哪里开始、哪里结束，所以它会扫描整个文件系统进行排除(也许是这样检查的吧)。

而在创建 ext3 文件系统时会划分三个区：数据区、日志区和元数据区。每次存储数据时，先在日志区中进行 ext2 中元数据区的活动，直到文件存储完成后标记上 commit 才将日志区中的数据转存到元数据区。当存储文件时突然断电，下一次检查修复文件系统时，只需要检查日志区的记录，将 bmap 对应的 data block 标记为未使用，并把 inode 号标记未使用，这样就不需要扫描整个文件系统而耗费大量时间。

虽说 ext3 相比 ext2 多了一个日志区转写元数据区的动作而导致 ext3 相比 ext2 性能要差一点，特别是写众多小文件时。但是由于 ext3 其他方面的优化使得 ext3 和 ext2 性能几乎没有差距。

4.9 ext4 文件系统

回顾前面关于 ext2 和 ext3 文件系统的存储格式，它使用 block 为存储单元，每个 block 使用 bmap 中的位来标记是否空闲，尽管使用划分块组的方法优化提高了效率，但是一个块组内部仍然使用 bmap 来标记该块组内的 block。对于一个巨大的文件，扫描整个 bmap 都将是一件浩大的工程。另外在 inode 寻址方面，ext2/3 使用直接和间接的寻址方式，对于三级间接指针，可能要遍历的指针数量是非常非常巨大的。

ext4 文件系统的最大特点是在 ext3 的基础上使用区(extent，或称为段)的概念来管理。一个 extent 尽可能的包含物理上连续的一堆 block。inode 寻址方面也一样使用区段树的方式进行了改进。

默认情况下，EXT4 不再使用 EXT3 的 block mapping 分配方式 ，而改为 Extent 方式分配。

以下是 ext4 文件系统中一个文件的 inode 属性示例，注意最后两行的 EXTENTS。

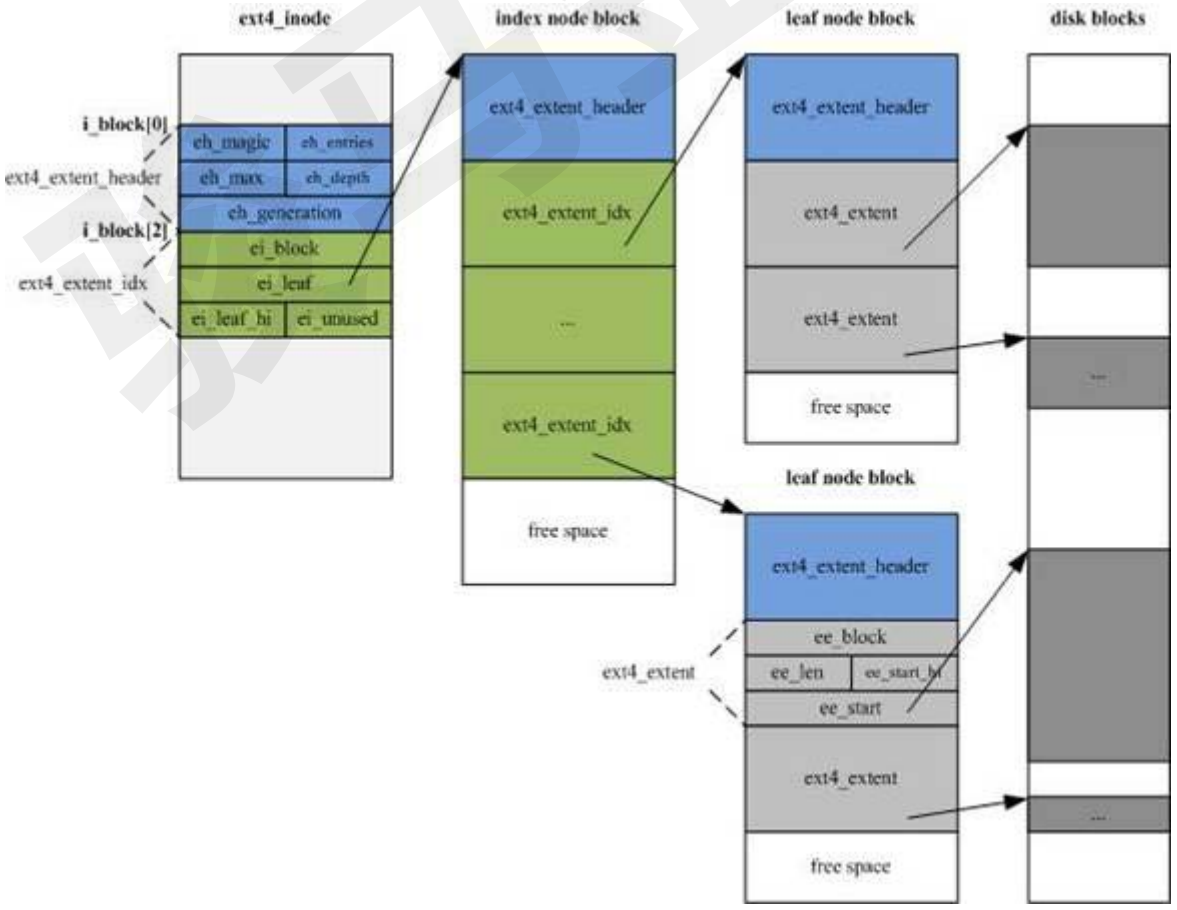
```
Inode: 12   Type: regular    Mode:  0644   Flags: 0x80000
Generation: 476513974    Version: 0x00000000:00000001
User:     0   Group:     0   Size: 11
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 8
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x5b628ca0:491d6224 -- Thu Aug  2 12:46:24 2018
 atime: 0x5b628ca0:491d6224 -- Thu Aug  2 12:46:24 2018
 mtime: 0x5b628ca0:491d6224 -- Thu Aug  2 12:46:24 2018
crtime: 0x5b628ca0:491d6224 -- Thu Aug  2 12:46:24 2018
Size of extra inode fields: 28
EXTENTS:
(0):33409
```

(1). 关于 EXT4 的结构特征

EXT4 在总体结构上与 EXT3 相似，大的分配方向都是基于相同大小的块组，每个块组内分配固定数量的 inode、可能的 superblock(或备份)及 GDT。

EXT4 的 inode 结构做了重大改变，为增加新的信息，大小由 EXT3 的 128 字节增加到默认的 256 字节，同时 inode 寻址索引不再使用 EXT3 的"12 个直接寻址块+1 个一级间接寻址块+1 个二级间接寻址块+1 个三级间接寻址块"的索引模式，而改为 4 个 Extent 片断流，每个片断流设定片断的起始 block 号及连续的 block 数量(有可能直接指向数据区，也有可能指向索引块区)。

片段流即下图中索引节点(index node block)部分的绿色区域，每个 15 字节，共 60 字节。



(2). EXT4 删除数据的结构更改。

EXT4 删除数据后，会依次释放文件系统 bitmap 空间位、更新目录结构、释放 inode 空间位。

(3). ext4 使用多 block 分配方式。

在存储数据时，ext3 中的 block 分配器一次只能分配 4KB 大小的 Block 数量，而且每存储一个 block 前就标记一次 bmap。假如存储 1G 的文件，blocksize 是 4KB，那么每存储完一个 Block 就将调用一次 block 分配器，即调用的次数为 1024*1024/4KB=262144 次，标记 bmap 的次数也为 1024*1024/4=262144 次。

而在 ext4 中根据区段来分配，可以实现调用一次 block 分配器就分配一堆连续的 block，并在存储这一堆 block 前一次性标记对应的 bmap。这对于大文件来说极大的提升了存储效率。

4.10 ext 类的文件系统的缺点

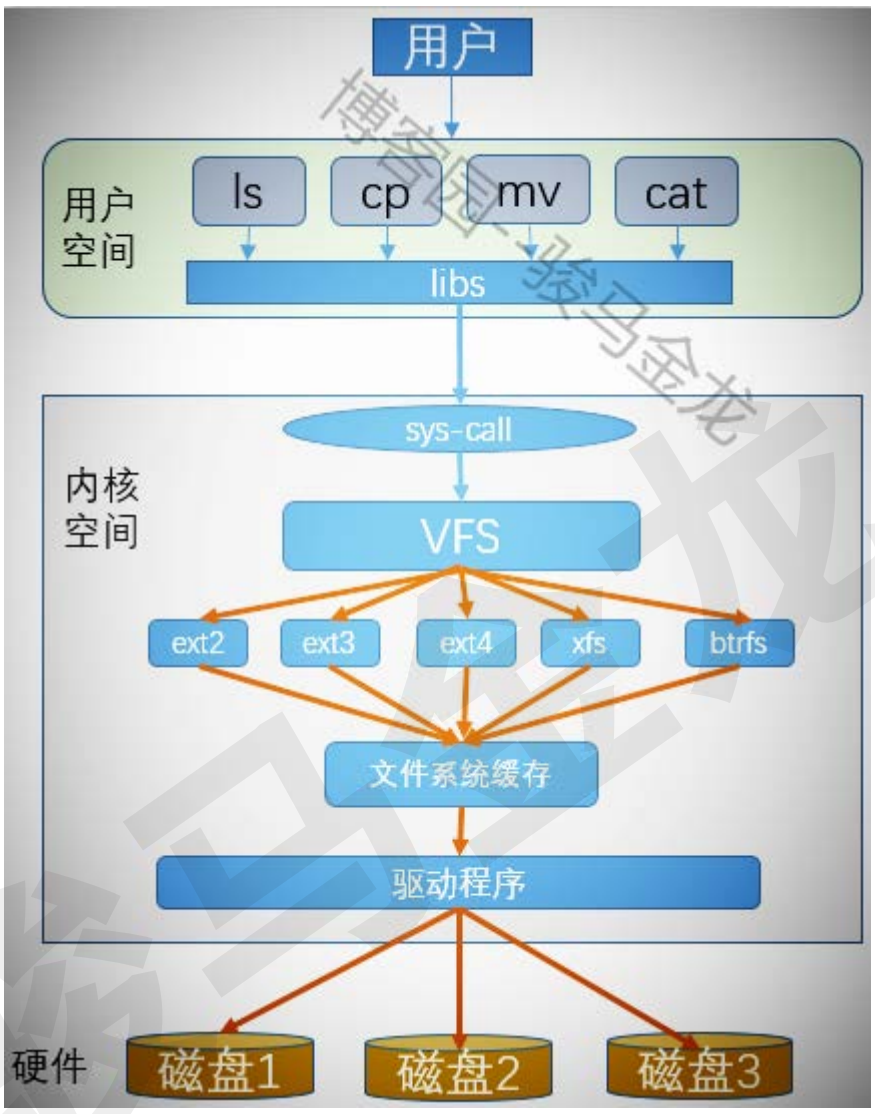
最大的缺点是它在创建文件系统的时候就划分好一切需要划分的东西，以后用到的时候可以直接进行分配，也就是说它不支持动态划分和动态分配。对于较小的分区来说速度还好，但是对于一个超大的磁盘，速度是极慢极慢的。例如将一个几十 T 的磁盘阵列格式化为 ext4 文件系统，可能你会因此而失去一切耐心。

除了格式化速度超慢以外，ext4 文件系统还是非常可取的。当然，不同公司开发的文件系统都各有特色，可以根据需求选择合适的文件系统类型。

4.11 虚拟文件系统 VFS

每一个分区格式化后都可以建立一个文件系统，Linux 上可以识别很多种文件系统，那么它是如何识别的呢？另外，在我们操作分区中的文件时，并没有指定过它是哪个文件系统的，各种不同的文件系统如何被我们用户以无差别的方式操作呢？这就是虚拟文件系统的作用。

虚拟文件系统为用户操作各种文件系统提供了通用接口，使得用户执行程序时不需要考虑文件是在哪种类型的文件系统上，应该使用什么样的系统调用什么样的系统函数来操作该文件。有了虚拟文件系统，只要将所有需要执行的程序调用 VFS 的系统调用就可以了，剩下的动作由 VFS 来帮忙完成。

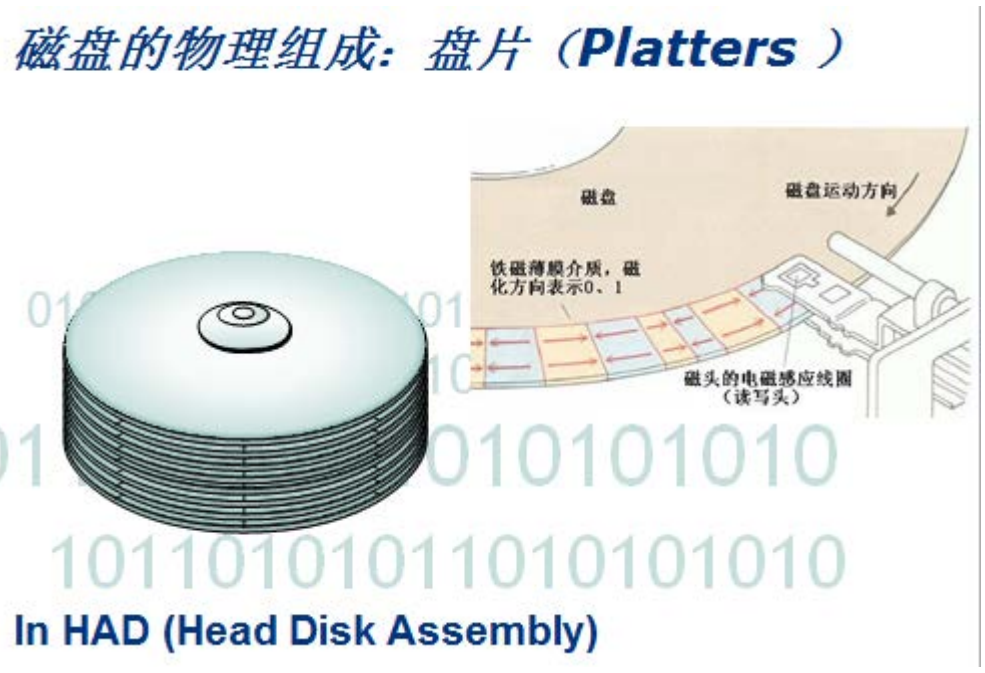


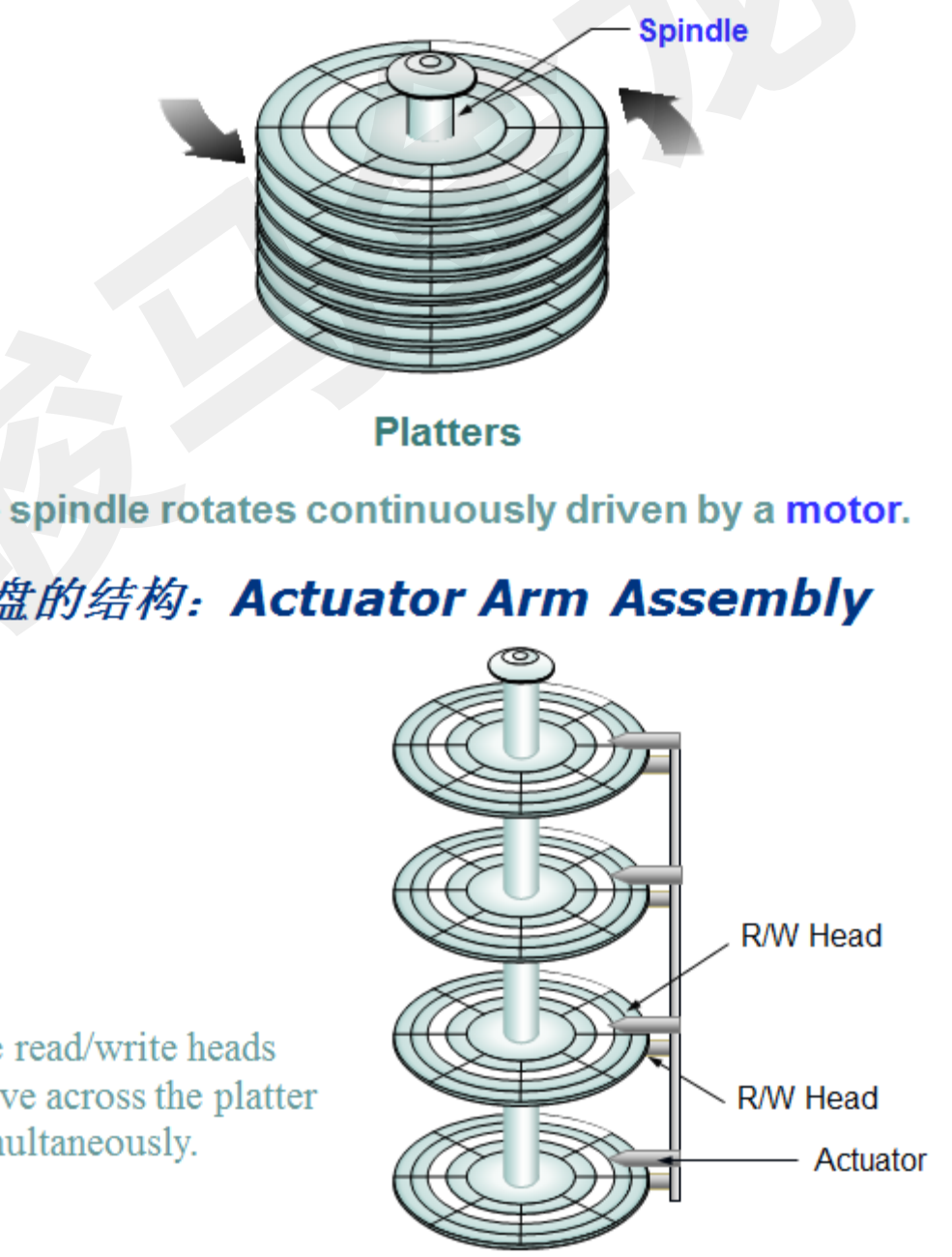
第5章 管理文件系统

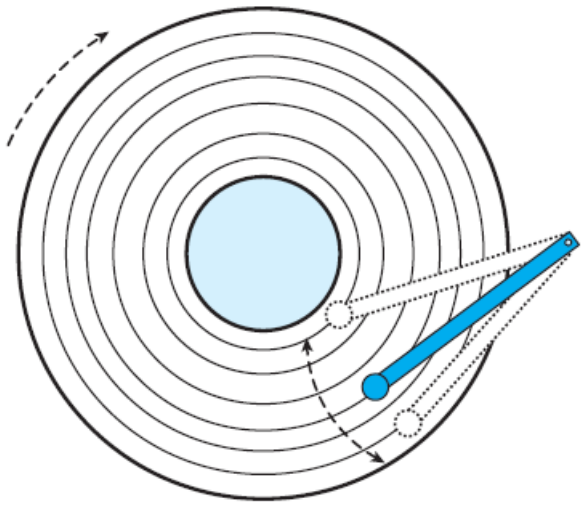
5.1 机械硬盘

机械硬盘由多块盘片组成，它们都绕着主轴旋转。每块盘片上下方都有读写磁头悬浮在盘片上下方，它们与盘片的距离极小。在每次读写数据时盘片旋转，读写磁头被磁臂控制着不断的移动来读取其中的数据。

所有的盘片都是同时同步转动，所有的磁头也是同步移动。





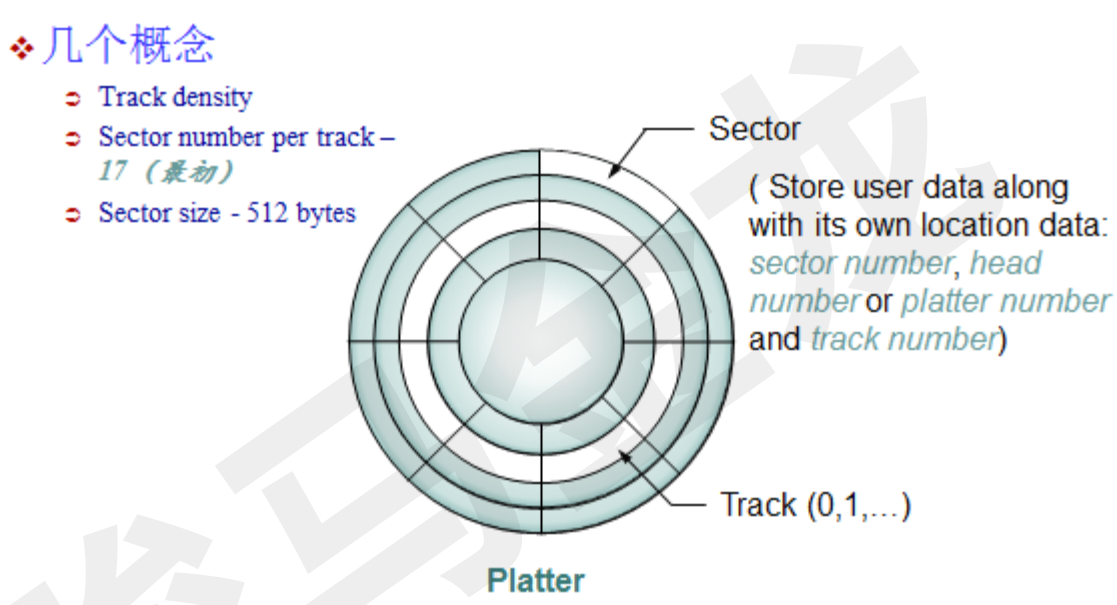


磁盘在物理上划分了扇区、磁道和柱面。如果划分了分区，则分区是逻辑上柱面的分隔边界。

读写磁头在停止状态下，在盘片旋转时磁头扫过的一圈轨迹称为磁道，所有的磁道都是同心圆。从盘片外圈开始向内数，磁道号从 0 开始逐数增加。

每个磁道以 512 字节等分为多个弧段，每个弧段就是一个扇区。但是需要说明的是，扇区的大小并非一定是 512 字节。所以外圈磁道的扇区数较多，内圈磁道的扇区数较少，有些硬盘参数上写的磁道扇区数通常用一个范围来标识，如 373-768 表示最外圈磁道有 768 个扇区，最内圈有 373 个扇区，这就可以计算出每个磁道的字节数。

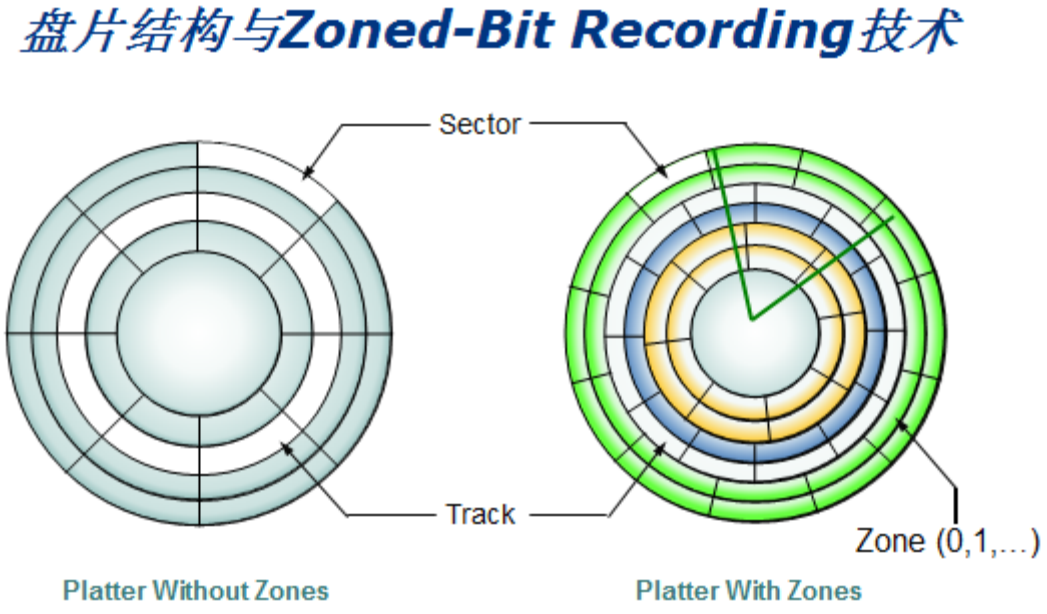
扇区上记录了物理数据、扇区号、磁头号(或者盘片号)及磁道号。



旧式磁盘的结构不分区(without zones)，每个磁道扇区数相同，但是每个扇区仍然是 512 字节，也就是说磁性材料记录的 0 和 1 的数量是相同的。这种结构的缺点是外圈磁道的面积大，存储的数据分布宽松，内圈磁道面积小，存储的数据分布密集，这样就导致盘片外圈面积浪费。

新式磁盘结构进行了分区，将每个磁道等面积划分 512 字节的空间作为一个扇区，所以不同磁道扇区数不同。

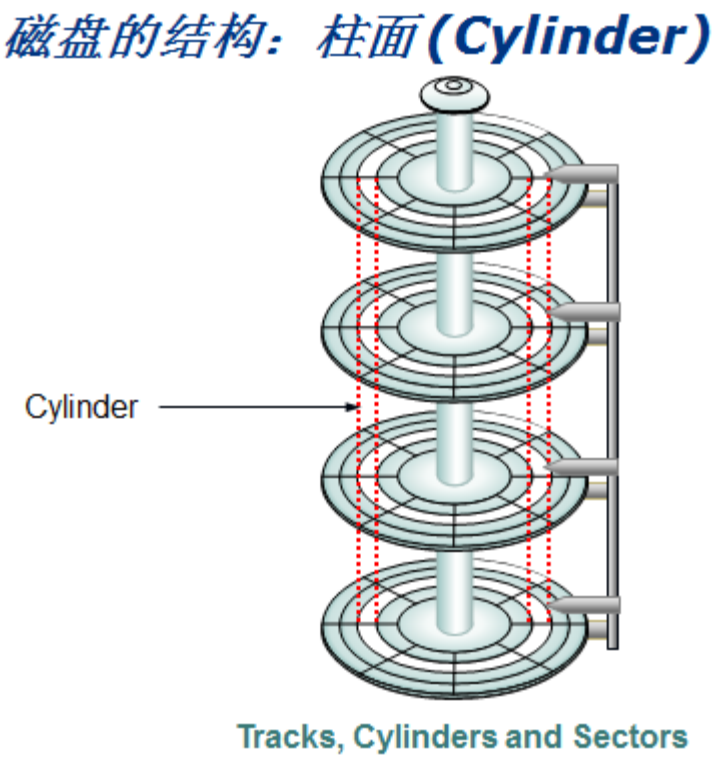
现在的磁盘都是新式扇区划分结构。



将所有盘片相同磁道数的磁道划分为柱面。和磁道号的标记方式一样，从外向内从 0 开始逐数增加。

之所以划分柱面，是因为所有磁盘同步旋转，所有磁头同步移动，所有的磁头在任意一个时刻总是会出在同一个磁道同一个扇区上。读写数据时，任意一段数据总是按柱面来读写的。所以盘片数越多，读写所扫的扇区数就越少，所需的时间相对就越少，性能就越好。

向磁盘写数据是从外圈柱面向内圈柱面写的，只有写完一个柱面才写下一个柱面。



5.2 磁盘或分区容量计算

虽然现在的磁盘都是新式结构(每磁道扇区数不同)，但是在磁盘信息上还是根据旧式结构来计算的，也就是说每个磁道扇区数相同，"扇区/磁道"的值说明每个磁道上有多少个扇区，也可以将其认为是新式结构下的平均值。

磁盘相关英文：

disk	磁盘
heads	磁头。Linux 系统中查看到的 heads 一般包括很多虚拟磁头，实际的物理磁盘的一块盘片上下两面一面一磁头，即 2 个磁头。
sectors	扇区。一磁道上划分多个扇形区域，一般默认一扇区 512 字节。
track	磁道。盘片上一圈算一磁道。
cylinders	柱面。所有盘片的同一半径的磁道组成一柱面。柱面数=盘片数*盘片上的磁道数。
units	单元块。大小等于一个柱面大小。

磁盘或分区大小计算方法：

```
磁盘大小=units×柱面数(cylinders)
磁盘大小=磁头数(heads)×每磁道上的扇区数(sectors)×512×柱面数(cylinders)
```

例如：查看/dev/sda3。

```
[root@xuexi tmp]# fdisk -l /dev/sda3
Disk /dev/sda3: 19.1 GB, 19116589056 bytes          # 总大小 19G
255 heads, 63 sectors/track, 2324 cylinders         # 磁头 255 柱面 2324  每磁道扇区数 63(这是平均数)
Units = cylinders of 16065 * 512 = 8225280 bytes    # 单元块大小
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000
```

Units=255×63×512=16065×512=8225280

磁盘大小=255*63*2324*512=19115550720= 19.11555072GB，不用 1024 算，用 1000 算。

5.3 分区

分区是为了在逻辑上将某些柱面隔开形成边界。它是以柱面为单位来划分的，但是从 CentOS 7 开始，是按照扇区进行划分的。

在磁盘数据量非常大的情况下，划分分区的好处是扫描块位图等更快速：不用再扫描整块磁盘的块位图，只需扫描对应分区的块位图。

5.3.1 分区方法(MBR 和 GPT)

MBR 格式的磁盘中，会维护磁盘第一个扇区——MBR 扇区，在该扇区中第 446 字节之后的 64 字节是分区表，每个分区占用 16 字节，所以限制了一块磁盘最多只能有 4 个主分区(Primary,P)，如果多于 4 个区，只能将主分区少于 4 个，通过建立扩展分区(Extend,E)，然后在扩展分区建立逻辑分区(Logical,L)的方式来突破 4 个分区的限制，逻辑分区的数量不受限制。

在 Linux 中，MBR 格式的磁盘主分区号从 1-4，扩展分区号从 2-4，逻辑分区号从 5 开始。

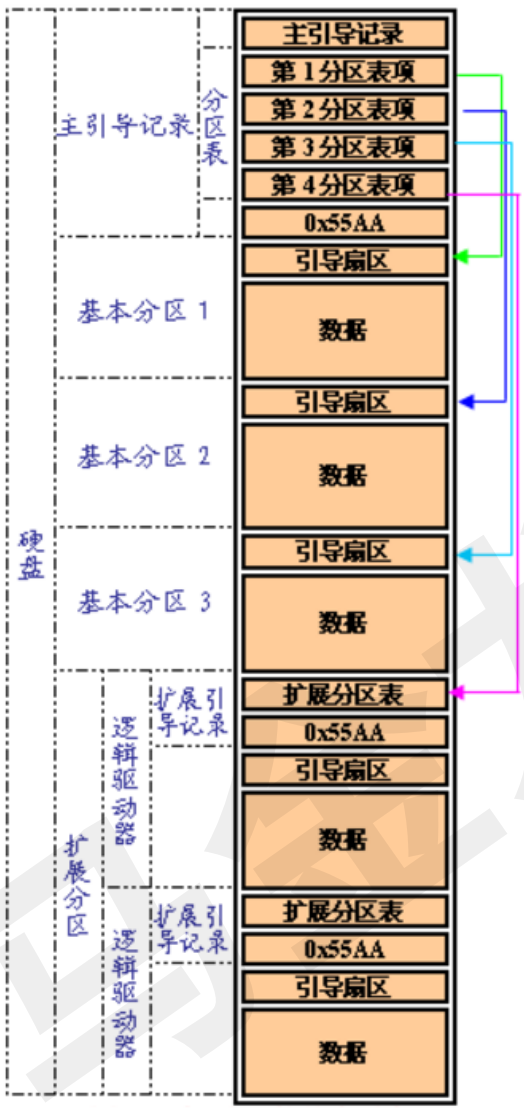
例如，一块盘想分成 6 个分区，可以：

```
1P+5L:sda1+sda5+sda6+sda7+sda8+sda9
2P+4L:sda1+sda2+sda5+sda6+sda7+sda8
3P+3L:sda1+sda2+sda3+sda5+sda6+sda7
```

而 GPT 格式突破了 MBR 的限制，它不再限制只能存储 4 个分区表条目，而是使用了类似 MBR 扩展分区表条目的格式，它允许有 128 个主分区，这也使得它可以对超过 2TB 的磁盘进行分区。

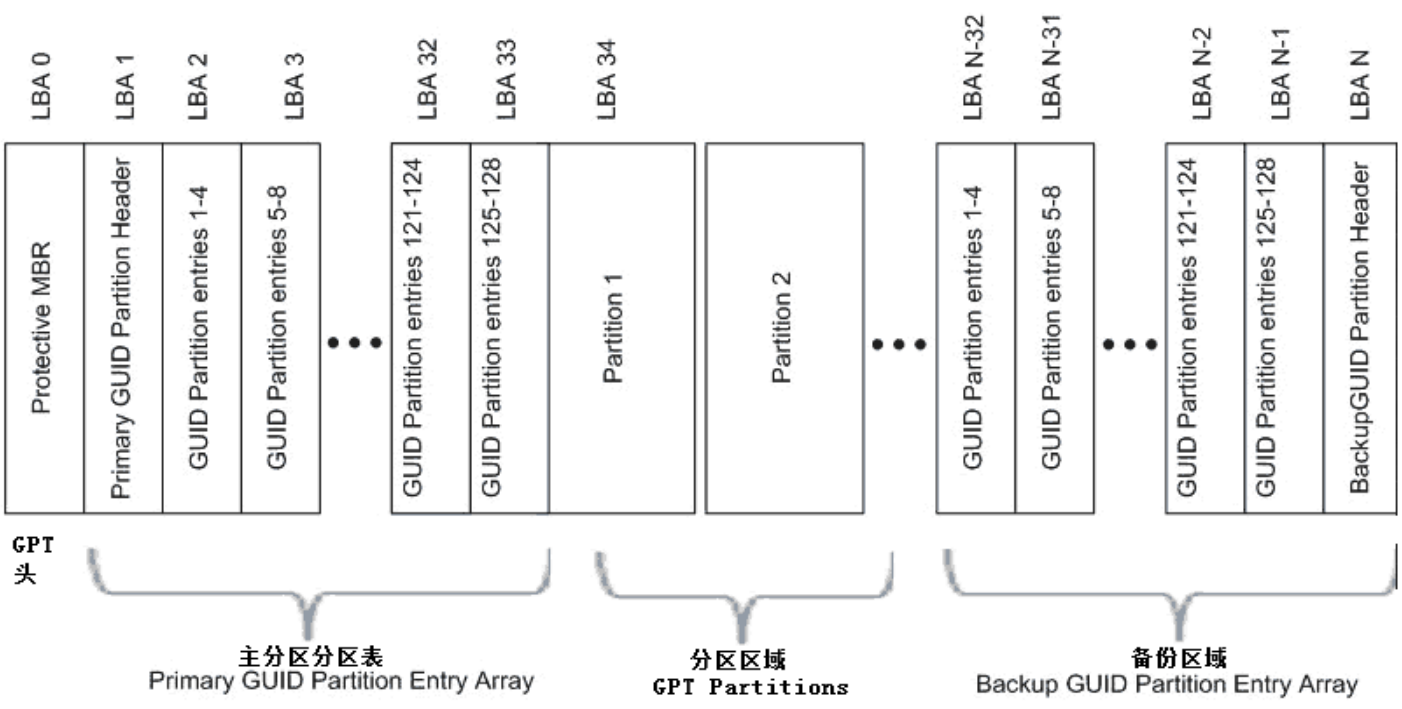
5.3.2 MBR 和 GPT 分区表信息

在 MBR 格式分区表中，前 446 个字节是主引导记录，即 boot loader。中间 64 字节记录着分区表信息，每个主分区信息占用 16 字节，因此最多只能有 4 个主分区。如果使用扩展分区，则扩展分区对应的 16 字节记录的是指向扩展分区中扩展分区表的指针。



在 MBR 磁盘上，分区和启动信息是保存在一起的，如果这部分数据被覆盖或破坏，只能重建 MBR。而 GPT 在整个磁盘上保存多个这部分信息的副本，因此它更为健壮，并可以恢复被破坏的这部分信息。GPT 还为这些信息保存了循环冗余校验码(CRC)以保证其完整和正确，如果数据被破坏，GPT 会发现这些破坏，并从磁盘上的其他地方进行恢复。

下面是 GPT 格式的分区表信息。



EFI 部分可以分为 4 个区域：EFI 信息区(GPT 头)、分区表、GPT 分区区域和备份区域。

- EFI 信息区(GPT 头)：起始于磁盘的 LBA1，通常也只占用这个单一扇区。其作用是定义分区表的位置和大小。GPT 头还包含头和分区表的校验和，这样就可以及时发现错误。

- 分区表：分区表区域包含分区表项。这个区域由 GPT 头定义，一般占用磁盘 LBA2～LBA33 扇区，每扇区可存储 4 个主分区的分区信息，所以共能分 128 个主分区。分区表中的每个分区项由起始地址、结束地址、类型值、名字、属性标志、GUID 值组成。分区表建立后，128 位的 GUID 对系统来说是唯一的。
- GPT 分区：最大的区域，由分配给分区的扇区组成。这个区域的起始和结束地址由 GPT 头定义。
- 备份区：备份区域位于磁盘的尾部，包含 GPT 头和分区表的备份。它占用 GPT 结束扇区和 EFI 结束扇区之间的 33 个扇区。其中最后一个扇区用来备份 1 号扇区的 EFI 信息，其余的 32 个扇区用来备份 LBA2～LBA33 扇区的分区表。

5.3.3 添加磁盘

正常情况下，添加磁盘后需要重启系统才能被内核识别，在/dev/下才有对应的设备号，使用 fdisk -l 才会显示出来。但是有时候不方便重启。此时可以使用下面的方法。

```
[root@node1 ~]# ls /sys/class/scsi_host/  # 查看主机 scsi 总线号
host0  host1  host2
```

重新扫描 scsi 总线来添加新设备，之所以扫描 scsi 总线是因为虚拟机中添加硬盘插入的是 scsi 硬盘。

```
[root@node1 ~]# echo "- - -" > /sys/class/scsi_host/host0/scan
[root@node1 ~]# echo "- - -" > /sys/class/scsi_host/host1/scan
[root@node1 ~]# echo "- - -" > /sys/class/scsi_host/host2/scan
[root@node1 ~]# fdisk -l      # 再查看就有了
```

如果 scsi_host 目录系很多 hostN 目录，则使用循环来完成。

```
[root@xuexi scsi_host]# ls /sys/class/scsi_host/
host0   host11  host14  host17  host2   host22  host25  host28  host30  host4  host7
host1   host12  host15  host18  host20  host23  host26  host29  host31  host5  host8
host10  host13  host16  host19  host21  host24  host27  host3   host32  host6  host9
[root@xuexi scsi_host]# for i in /sys/class/scsi_host/host*/scan;do echo "- - -" >$i;done
```

5.3.4 使用 fdisk 分区工具

fdisk 工具用来分 MBR 磁盘上的区。要分 GPT 磁盘上的区，可以使用 gdisk。parted 工具对这两种格式的磁盘分区都支持。

如果一个存储设备已经分过区，那么它可能是 mbr 格式的，也可能是 gpt 格式的，如果已经是 mbr 格式的，则只能继续使用 fdisk 进行分区，如果已经是 gpt 格式的，则只能使用 gdisk 进行分区。当然，无论什么格式的都可以使用 parted 进行分区，只不过也只能划分和已存在分区格式一样的分区，因为无论何种格式的分区，它的分区表和分区标识是已经固定的。

使用 fdisk 分区，它只能实现 MBR 格式的分区。

```
[root@xuexi ~]# fdisk /dev/sdb   # sdb 后没加数字
Command (m for help): m          # 输入 m 查看可用命令帮助
Command action
   a   toggle a bootable flag
   b   edit bsd disklabel
   c   toggle the dos compatibility flag
   d   delete a partition        # 删除分区，如果删除扩展分区同时会删除里面的逻辑分区
   l   list known partition types # 列出分区类型
   m   print this menu           # 显示帮助信息
   n   add a new partition       # 创建新分区
   o   create a new empty DOS partition table
   p   print the partition table      # 输出分区信息
   q   quit without saving changes    # 不保存退出
   s   create a new empty Sun disklabel
   t   change a partition's system id  # 修改分区类型
   u   change display/entry units
   v   verify the partition table
   w   write table to disk and exit    # 保存分区信息并退出
   x   extra functionality (experts only)
```

新建第一个主分区：

```
Command (m for help): n                    # 添加分区
Command action
   e   extended                            # 添加扩展分区
   p   primary partition (1-4)             # 添加主分区
p                                          # 输入 p 来创建第一个主分区
Partition number (1-4): 1                  # 输入分区号，从 1 开始
First cylinder (1-1305, default 1):     # 输入柱面号，不输人默认是 1
Using default value 1
Last cylinder, +cylinders or +size{K,M,G} (1-1305, default 1305): +2G  # 给第一个主分区/dev/sdb1 分 2G，也可以使用柱面号来指定大小。
```

```
Command (m for help): p     # 第一个分区结束，p 查看下已分区信息

Disk /dev/sdb: 10.7 GB, 10737418240 bytes
255 heads, 63 sectors/track, 1305 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x2d8d64eb

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1               1         262     2104483+  83  Linux
```

新建扩展分区：

```
Command (m for help): n      # 再建一个分区
Command action
   e   extended
   p   primary partition (1-4)
e      # 创建扩展分区
Partition number (1-4): 2     # 扩展分区号为 2
First cylinder (263-1305, default 263):
Using default value 263
Last cylinder, +cylinders or +size{K,M,G} (263-1305, default 1305): # 剩余空间全部给扩展分区
Using default value 1305
Command (m for help): p
Disk /dev/sdb: 10.7 GB, 10737418240 bytes
255 heads, 63 sectors/track, 1305 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x2d8d64eb

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1               1         262     2104483+  83  Linux
/dev/sdb2             263        1305     8377897+   5  Extended
```

新建扩展分区：

```
Command (m for help): n         # 新建逻辑分区
Command action
   l   logical (5 or over)      # 这里不再是扩展分区标识 e，只有 l。
            # 如果已有 3 个主分区，这里连 l 都没有
   p   primary partition (1-4)
l       # 新建逻辑分区
First cylinder (263-1305, default 263):      # 这里也不能选逻辑分区号了
Using default value 263
Last cylinder, +cylinders or +size{K,M,G} (263-1305, default 1305): +3G
Command (m for help): p
Disk /dev/sdb: 10.7 GB, 10737418240 bytes
255 heads, 63 sectors/track, 1305 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x2d8d64eb

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1               1         262     2104483+  83  Linux
/dev/sdb2             263        1305     8377897+   5  Extended
/dev/sdb5             263         655     3156741   83  Linux
```

分区结束，保存。如果不保存，则按 q。

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

Calling ioctl() to re-read partition table.
Syncing disks.
```

分区的过程，实质上是划分柱面以及修改分区表。

上面的 fdisk 操作全部是在内存中执行的，必须保存生效。保存后，内核还未识别该分区，可以查看/proc/partition 目录下存在的文件，这些文件是能被内核识别的分区。运行 partprobe 或 partx 命令重新读取分区表让内核识别新的分区，内核识别后才可以格式化。而且分区结束时按 w 保存分区表有时候会失败，提示重启，这时候运行 partprobe 命令可以代替重启就生效。

```
[root@xuexi ~]# partprobe        # 执行 partprobe，下面一堆信息，不理它
Warning: WARNING: the kernel failed to re-read the partition table on /dev/sda (Device or resource busy).  As a result, it may not reflect all of your changes until after reboot.
Warning: Unable to open /dev/sr0 read-write (Read-only file system).  /dev/sr0 has been opened read-only.
Warning: Unable to open /dev/sr0 read-write (Read-only file system).  /dev/sr0 has been opened read-only.
Error: Invalid partition table - recursive partition on /dev/sr0.
```

也可指定在/dev/sdb 上重加载分区表，省的无法读取正忙的/dev/sda 磁盘，给出上面一堆信息。

```
[root@xuexi ~]# partprobe /dev/sdb
```

分区之后再使用 fdisk -l 查看新的分区状态。

将上面分区所需要使用的命令总结如下，以后便于使用脚本分区。

- fdisk /dev/sdb # 选择要分区的设备
- n # 创建分区
- p/e/l # 选择分区类型

如果主分区数有 3 个，且已经划分了扩展分区，再继续分区时将只能划分逻辑分区，这种情况下 l 选项会直接跳过进入下一个阶段。

- N # 指定分区号
- \n # 指定起始柱面号，使用默认值就直接回车即换行
- +N # 指定分区大小为 N
- w # 分区结束保存退出
- partprobe /dev/sdb &>/dev/null # 重读分区表
- fdisk -l | grep "^/dev/sdb" &>/dev/null # 检查分区状态

5.3.5 使用 gdisk 分区工具

gdisk 用来划分 gpt 分区，需要单独安装这个工具包。

```
yum -y install gdisk
```

分区的时候直接带上设备即可。以下是对新硬盘划分 gpt 分区的过程。

```
[root@xuexi ~]# gdisk /dev/sdb
GPT fdisk (gdisk) version 0.8.10

Partition table scan:
  MBR: not present
  BSD: not present
  APM: not present
  GPT: not present

Creating new GPT entries.

Command (? for help): ?
b       back up GPT data to a file
c       change a partition's name
d       delete a partition                              # 删除分区
i       show detailed information on a partition        # 列出分区详细信息
l       list known partition types                      # 列出所以已知的分区类型
n       add a new partition                             # 添加新分区
o       create a new empty GUID partition table (GPT)   # 创建一个新的空的 guid 分区表
p       print the partition table                       # 输出分区表信息
q       quit without saving changes                     # 退出 gdisk 工具
r       recovery and transformation options (experts only)
s       sort partitions
t       change a partition's type code                  # 修改分区类型
v       verify disk
w       write table to disk and exit                    # 将分区信息写入到磁盘
x       extra functionality (experts only)
?       print this menu
```

添加一个新分区。

```
Command (? for help): n
Partition number (1-128, default 1):
First sector (34-41943006, default = 2048) or {+-}size{KMGTP}:
Last sector (2048-41943006, default = 41943006) or {+-}size{KMGTP}: +10G
Current type is 'Linux filesystem'
Hex code or GUID (L to show codes, Enter = 8300):
Changed type of partition to 'Linux filesystem'

Command (? for help): p
Disk /dev/sdb: 41943040 sectors, 20.0 GiB
Logical sector size: 512 bytes
Disk identifier (GUID): F8AE925F-515F-4807-92ED-4109D0827191
Partition table holds up to 128 entries
First usable sector is 34, last usable sector is 41943006
Partitions will be aligned on 2048-sector boundaries
Total free space is 20971453 sectors (10.0 GiB)

Number  Start (sector)    End (sector)  Size       Code  Name
   1            2048        20973567   10.0 GiB    8300  Linux filesystem


Command (? for help): i
Using 1
Partition GUID code: 0FC63DAF-8483-4772-8E79-3D69D8477DE4 (Linux filesystem)
Partition unique GUID: B2452103-4F32-4B60-AEF7-4BA42B7BF089
First sector: 2048 (at 1024.0 KiB)
Last sector: 20973567 (at 10.0 GiB)
Partition size: 20971520 sectors (10.0 GiB)
Attribute flags: 0000000000000000
Partition name: 'Linux filesystem'
```

保存分区表到磁盘。

```
Command (? for help): w

Final checks complete. About to write GPT data. THIS WILL OVERWRITE EXISTING
PARTITIONS!!

Do you want to proceed? (Y/N): Y
OK; writing new GUID partition table (GPT) to /dev/sdb.
The operation has completed successfully.
```

执行 partprobe 重新读取分区表信息。

```
[root@server2 ~]# partprobe  /dev/sdb
```

gdisk 还有几个 expert only 的命令，其实没什么专家不专家可用的，咱们需要知道的是命令何时能用，它们的作用是什么？

在 gdisk 交互过程命令行下，按下 x 表示进入扩展功能模式，该模式下的功能大部分都和 gpt 分区表相关，在不是非常了解 gpt 分区表结构的时候不建议做修改动作，但是查看信息类是没问题的。以下是扩展功能模式下的命令。

```
Command (? for help): x

Expert command (? for help): ?
a       set attributes
c       change partition GUID
d       display the sector alignment value
e       relocate backup data structures to the end of the disk
g       change disk GUID
h       recompute CHS values in protective/hybrid MBR
i       show detailed information on a partition
l       set the sector alignment value
m       return to main menu
n       create a new protective MBR
o       print protective MBR data
p       print the partition table
q       quit without saving changes
r       recovery and transformation options (experts only)
s       resize partition table    # 修改分区表大小，注意不是分区大小
```

```
t       transpose two partition table entries
u       Replicate partition table on new device  # 将分区表导出
v       verify disk
w       write table to disk and exit
z       zap (destroy) GPT data structures and exit     # 损毁 gpt 上的数据
?       print this menu
```

5.3.6 使用 parted 分区工具

parted 支持 mbr 格式和 gpt 格式的磁盘分区。它的强大在于可以一步到位而不需要不断的交互式输入(也可以交互式)。

parted 分区工具是实时的，所以每一步操作都是直接写入磁盘而不是写进内存，它不像 fdisk/gdisk 还需要 w 命令将内存中的结果保存到磁盘中。

```
[root@xuexi ~]# parted /dev/sdc
GNU Parted 2.1
Using /dev/sdc
Welcome to GNU Parted! Type 'help' to view a list of commands.

(parted) help
  align-check TYPE N                       check partition N for TYPE(min|opt) alignment
  check NUMBER (centos 7 上已删除该功能) do a simple check on the file system
  cp [FROM-DEVICE] FROM-NUMBER TO-NUMBER (centos 7 上已删除该功能)   copy file system to another partition
  help [COMMAND]                          print general help, or help on COMMAND
  mklabel,mktable LABEL-TYPE               create a new disklabel (partition table)
  mkfs NUMBER FS-TYPE   (centos 7 上已删除该功能)  make a FS-TYPE file system on partition NUMBER
  mkpart PART-TYPE [FS-TYPE] START END     make a partition
  mkpartfs PART-TYPE FS-TYPE START END  (centos 7 上已删除该功能)   make a partition with a file system
  move NUMBER START END     (centos 7 上已删除该功能)  move partition NUMBER
  name NUMBER NAME                        name partition NUMBER as NAME
  print [devices|free|list,all|NUMBER]   display the partition table,available devices,free space, all found partitions,
                                          or a particular partition
  quit                                      exit program
  rescue START END                        rescue a lost partition near START and END
  resize NUMBER START END (修改分区大小(centos 7 上已删除该功能))  resize partition NUMBER and its file system
  rm NUMBER      (删除分区)                delete partition NUMBER
  select DEVICE (重选磁盘进入 parted 状态)  choose the device to edit
  set NUMBER FLAG STATE  (设置分区状态，如将其 off 或 on)   change the FLAG on partition NUMBER
  toggle [NUMBER [FLAG]] (修改文件系统类型，如 swap、lvm)   toggle the state of FLAG on partition NUMBER
  unit UNIT  (修改默认单位，kB/MB/GB 等)   set the default unit to UNIT
  version                       display the version number and copyright information of GNU Parted
```

常用的命令是 mklabel/rm/print/mkpart/help/quit，至于 parted 中一些看上去很好的功能如 mkfs/mkpartfs/resize 等可能会损毁当前数据而不够安全，所以只要使用它的 5 个常用命令即可。

parted 分区的前提是磁盘已经有分区表(partition table)或磁盘标签(disk label)，否则将显示"unrecognised disk label"，这是和 fdisk/gdisk 不同的地方，所以需要先使用 mklabel 创建标签或分区表，最常见的标签(分区表)为"msdos"和"gpt"，其中 msdos 分区就是 MBR 格式的分区表，也就是会有主分区、扩展分区和逻辑分区的概念和限制。

下面使用 parted 对/dev/sdc 创建 msdos 的新分区。

```
[root@xuexi ~]# parted /dev/sdc
GNU Parted 2.1
Using /dev/sdb
Welcome to GNU Parted! Type 'help' to view a list of commands.
(parted) mklabel              # 创建磁盘分区标签(分区表类型)
New disk label type? msdos    # 选择 msdos 即 MBR 类型
                              # 上面的两步也可以直接一步进行：(parted) mklabel msdos
(parted) mkpart               # 开始进行分区
Partition type?  primary/extended? p      # 创建主分区
File system type?  [ext2]? ext4           # 创建 ext4 文件系统
                                          # (注意，这里虽然指明了文件系统，但没有任何意义，后面还是需要手动格式化并选择文件系统类型)
Start? 1                                  # 分区开始位置，默认是 M 为单位，表示从 1M 开始，也可直接指定 1G 这种方式
End? 1024                                 # 分区结束位置，1024-1=1023M

(parted) p                    # print，查看分区信息
Model: VMware, VMware Virtual S (scsi)
Disk /dev/sdc: 21.5GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
```

```
Number  Start   End     Size    Type     File system  Flags
 1      1049kB  1024MB  1023MB  primary

# 可以一步完成一个命令中的多个动作
(parted) mkpart p ext4 1026M 4096M      # 可以一步完成，也可以一步完成到任何位置，然后继续交互下一步
   # 可能会提示分区未对齐"Warning: The resulting partition is not properly aligned for best performance."，忽略它

(parted) mkpart e 4098 -1    # 创建扩展分区，注意创建扩展分区时不指定文件系统类型；-1 表示剩余的全部分配给该分区
(parted) p
Model: VMware, VMware Virtual S (scsi)
Disk /dev/sdc: 21.5GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos

Number  Start   End     Size    Type      File system  Flags
 1      1049kB  1024MB  1023MB  primary
 2      1026MB  4096MB  3070MB  primary
 3      4098MB  21.5GB  17.4GB  extended              lba

(parted) mkpart l ext4 4099 8194     # 创建逻辑分区，指定 ext4
(parted) mkpart l ext4 8195 -1       # 继续创建逻辑分区
(parted) p
Model: VMware, VMware Virtual S (scsi)
Disk /dev/sdc: 21.5GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos

Number  Start   End     Size    Type     File system  Flags
 1      1049kB  1024MB  1023MB  primary
 2      1026MB  4096MB  3070MB  primary
 3      4098MB  21.5GB  17.4GB  extended             lba
 5      4099MB  8194MB  4095MB  logical
 6      8195MB  21.5GB  13.3GB  logical

(parted) rm 5    # 删除 5 号分区
(parted) p
Model: VMware, VMware Virtual S (scsi)
Disk /dev/sdc: 21.5GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos

Number  Start   End     Size    Type     File system  Flags
 1      1049kB  1024MB  1023MB  primary
 2      1026MB  4096MB  3070MB  primary
 3      4098MB  21.5GB  17.4GB  extended             lba
 5      8195MB  21.5GB  13.3GB  logical

(parted) quit                                    # 退出 parted 工具
Information: You may need to update /etc/fstab.  # 提示你要更新/etc/fstab 中的配置，说明该工具是可以在线分区的
```

mkfs 和 mkpartfs 等命令不完善，下面的警告信息已经给出了提示。

```
(parted) mkfs 1
WARNING: you are attempting to use parted to operate on (mkfs) a file system.
parted's file system manipulation code is not as robust as what you'll find in
dedicated, file-system-specific packages like e2fsprogs.  We recommend
you use parted only to manipulate partition tables, whenever possible.
Support for performing most operations on most types of file systems
will be removed in an upcoming release.
Warning: The existing file system will be destroyed and all data on the partition will be lost. Do you want to
continue?
parted: invalid token: 1
Yes/No? n
```

使用 parted 工具进行的分区无需运行 partprobe 重新读取分区表，内核会即时识别已经分区的分区信息。如下所示。

```
[root@xuexi tmp]# cat /proc/partitions  | grep "sdc"
   8       32   20971520 sdc
   8       33     999424 sdc1
```

```
   8       34    2998272 sdc2
   8       35          1 sdc3
   8       37   12967936 sdc5
```

一定要注意，虽然 parted 工具中指定了文件系统，但是并没有意义，它仍需要手动进行格式化并指定分区类型。实际上，在 parted 中文件系统是可以不用指定的，即使是非交互模式下也可以省略。

5.3.7 fdisk/gdisk 以及 parted 非交互式操作分区

使用非交互分区时，最重要的是待分的区的起始点不能是已使用的。可以使用 lsblk 或 fdisk -l 或 parted DEV print 来判断将要从哪个地方开始分区。其实 parted 在非交互分区是最佳的工具，不仅是因为其书写方式简洁，而且待分区的起点如不合理它会自动提示是否要自动调整。

5.3.7.1 parted 实现非交互

parted 命令只能一次非交互一个命令中的所有动作。如下所示：

```
parted /dev/sdb mklabel msdos                 # 设置硬盘 flag
parted /dev/sdb mkpart primary ext4 1 1000    # Mbr 格式分区，分别是 partition type/fstype/start/end
parted /dev/sdb mkpart 1 ext4 1M 10240M     # gpt 格式分区，分别是 name/fstype/start/end
parted /dev/sdb mkpart 1 10G 15G              # 省略 fstype 的交互式分区
parted /dev/sdb rm 1                            # 删除分区
parted /dev/sdb p                               # 输出信息
```

如果不确定分区的起点大小，可以加上-s 选项使用 script 模式，该模式下 parted 将回答一切默认值，如 yes、no。

```
parted -s /dev/sdb mkpart 3 14G 16G
Warning: You requested a partition from 14.0GB to 16.0GB.
The closest location we can manage is 15.0GB to 16.0GB.
Is this still acceptable to you?
Information: You may need to update /etc/fstab.
```

5.3.7.2 fdisk 实现非交互

fdisk 实现非交互的原理是从标准输入中读取，每读取一行传递一次操作。

所以可以有两种方式：使用 echo 和管道传递；将操作写入到文件中，从文件中读取。

例如：下面的命令创建了两个分区。使用默认值时传递空行即可。

```
echo -e "n\np\n1\n\n+5G\nn\np\n2\n\n+1G\nw\n"  | fdisk /dev/sdb
```

如果要传递的操作很多，则可以将它们写入到一个文件中，从文件中读取。

```
echo -e "n\np\n1\n\n+5G\nn\np\n2\n\n+1G\nw\n" >/tmp/a.txt
fdisk /dev/sdb </tmp/a.txt
```

5.3.7.3 gdisk 实现非交互

原理同 fdisk。例如：

```
echo -e "n\n1\n\n+3G\n\nw\nY\n" | gdisk /dev/sdb
```

上面传递的各参数意义为：新建分区，分区 number 为 1，使用默认开始扇区位置，分区大小+3G，使用默认分区类型，保存，确认。

5.4 格式化分区

分区结束后就需要格式化创建文件系统了，格式化分区的过程就是创建文件系统的过程。可以使用 mkfs(make filesystem)工具进行格式化，也可以使用该工具家族的其他工具如 mkfs.ext4/mkfs.xfs 等专门针对文件系统的工具。

要查看支持的文件系统类型，只需简单的输入 mkfs 然后按两下 tab 键，就可以列出各文件系统对应的格式化命令，这些就是支持的文件系统类型。

CentOS 6 上支持的：

```
[root@xuexi ~]# mkfs
mkfs      mkfs.cramfs   mkfs.ext2     mkfs.ext3     mkfs.ext4     mkfs.ext4dev  mkfs.msdos    mkfs.vfat
```

CentOS 7 上支持的：

```
[root@server2 ~]# mkfs
mkfs      mkfs.btrfs    mkfs.cramfs   mkfs.ext2     mkfs.ext3     mkfs.ext4     mkfs.fat      mkfs.minix    mkfs.msdos    mkfs.vfat     mkfs.xfs
```

5.4.1 mkfs 工具

```
mkfs [-t fstype] 分区
```

该工具非常简单，它只需指定一个可选的"-t"选项指定要创建的文件系统类型，如果省略则默认创建 ext2 文件系统。该工具指定的"-t"选项其实是在调用对应文件系统专属的格式化工具。

5.4.2 mke2fs 工具

mkfs.ext2/mkfs.ext3/mkfs.ext4 或 mkfs -t extX 其实都是在调用 mke2fs 工具。

该工具创建文件系统时，会从/etc/mke2fs.conf 配置中读取默认的配置项。

```
mke2fs [ -c ] [ -b block-size ] [ -f fragment-size ] [ -g blocks-per-group ] [ -G number-of-groups ] [ -i bytes-per-inode ] [ -I inode-size ] [ -j ] [ -N number-of-inodes ] [ -m reserved-blocks-percentage ] [ -q ] [ -r fs-revision-level ] [ -v ] [ -L volume-label ] [ -S ] [ -t fs-type ] device [ blocks-count ]

选项说明：
-t fs-type          ：指定要创建的文件系统类型(ext2,ext3 ext4)，若不指定，则从/etc/mke2fs.conf 中获取默认的文件系统类型。
-b block-size       ：指定每个 block 的大小，有效值有 1024、2048 和 4096，单位是字节。
-I inode-size       ：指定 inode 大小，单位为字节。必须为 2 的幂次方，且大于等于 128 字节。值越大，说明 inode 的集合体 inode table 占用越多的空
                      间，这不仅会挤占文件系统中的可用空间，还会降低性能，因为要扫描 inode table 需要消耗更多时间，但是在 linux kernel 2.6.10
                      之后，由于使用 inode 存储了很多扩展的额外属性，所以 128 字节已经不够用了，因此 ext4 默认的 inode size 已经变为 256，尽管
                      inode 大小增大了，但因为使用 inode 存储扩展属性带来的性能提升远高于 inode size 变大导致的负面影响，所以仍建议使用 256 字
                      节的 inode。
-i bytes-per-inode  ：指定每多少个字节就为其分配一个 inode 号。值越大，说明一个文件系统中分配的 inode 号越少，更适用于存储大量大文件，值越
                      小，inode 号越多，更适用于存储大量小文件。该值不能小于一个 block 的大小，因为这样会造成 inode 多余。
                      注意，创建文件系统后该值就不能再改变了。
-c                  ：创建文件系统前先检查设备是否有 bad blocks。
-f fragment-size    ：指定 fragments 的大小，单位字节。
-g blocks-per-group：指定每个块组中的 block 数量。不建议修改此项。
-G number-of-groups：该选项用于 ext4 文件系统(严格地说是启用了 flex_bg 特性)，指定虚拟块组(即一个 extent)中包含的块组个数，必须为 2 的幂次方。
                      对于 ext4 文件系统来说，使用 extent 的功能能极大提升其性能。
-j                  ：创建带有日志功能的文件系统，即 ext3。如果要指定关于日志方面的设置，在-j 的基础上再使用-J 指定，不过一般默认即可，具体可
                      指定的选项看 man 文档。
-L new-volume-label：指定卷标名称，名称不得超出 16 字节。
-m reserved-blocks-percentage：指定文件系统保留 block 数量的比例，保留一部分 block，可以降低物理碎片。默认比例为 5%。
-N number-of-inodes ：强制指定该文件系统应该分配多少个 inode 号，它会覆盖通过计算得出 inode 数量的结果(根据 block 大小、数量和每多少字节分配
                      一个 inode 得出 Inode 数量)，但是不建议这么做。
-q                  ：安静模式，可用于脚本中
-S                  ：重建 superblock 和 group descriptions。在所有的 superblock 和备份的 superblock 都损坏时有用。它会重新初始化 superblock 和
                      group descriptions，但不会改变 inode table、bmap 和 imap(若真的改变，该分区数据就全丢了，还不如重新格式化)。在重建
                      superblock 后，应该执行 e2fsck 来保证文件系统的一致性。但要注意，应该完全正确地指定 block 的大小，其改选项并不能完全保
                      证数据不丢失。
-v                  ：输出详细执行过程
```

所以，有可能用到的选项也就"-t"指定文件系统类型，"-b"指定 block 大小，"-I"指定 inode 大小，"-i"指定分配 inode 的比例。

例如：

```
mke2fs -t ext4 -I 256 /dev/sdb2 -b 4096
mke2fs 1.41.12 (17-May-2010)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
655360 inodes, 2621440 blocks
131072 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=2684354560
80 block groups
32768 blocks per group, 32768 fragments per group
8192 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632

Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 39 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

提示使用 tune2fs 修改自动检测文件系统的频率。见下文。

5.4.3 tune2fs 修改 ext 文件系统属性

该工具其实没什么太大作用，文件系统创建好后很多属性是固定不能修改的，能修改的属性很有限，且都是无关紧要的。

但有些时候还是可以用到它做些事情，例如刚创建完 ext 文件系统后会提示修改自检时间。

```
tune2fs [  -c  max-mount-counts ] [ -i interval-between-checks ] [ -j ] device
-j：将 ext2 文件系统升级为 ext3；
-c：修改文件系统最多挂载多少次后进行自检，设置为 0 或-1 将永不自检；
-i：修改过了多少时间进行自检。时间单位可以指定为天(默认)/月/星期[d|m|w]，设置为 0 将永不自检。
```

```
tune2fs -i 0 /dev/sdb1
```

5.5 查看文件系统状态信息

5.5.1 lsblk

lsblk(list block devices)用于列出设备及其状态，主要列出非空的存储设备。其实它只会列出/sys/dev/block 中的主次设备号文件，且默认只列出非空设备。

```
[root@server2 ~]# lsblk
```

```
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda      8:0    0   20G  0 disk
├─sda1   8:1    0  250M  0 part /boot
├─sda2   8:2    0 17.8G  0 part /
└─sda3   8:3    0    2G  0 part [SWAP]
sdb      8:16   0   20G  0 disk
└─sdb1   8:17   0  9.5G  0 part /mydata/data
sr0     11:0    1    4G  0 rom  /mnt
```

其中上面的几列意义如下：

```
NAME：设备名称；
MAJ:MIN：主设备号和此设备号；
RM：是否为可卸载设备，1 表示可卸载设备。可卸载设备如光盘、USB 等。并非能够 umount 的就是可卸载的；
SIZE：设备总空间大小；
RO：是否为只读；
TYPE：是磁盘 disk，还是分区 part，亦或是 rom，还有 loop 设备；
mountpoint：挂载点。
```

```
[root@server2 ~]# lsblk /dev/sdb
NAME     MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sdb        8:16   0   20G  0 disk
├─sdb1     8:17   0  9.5G  0 part /mydata/data
└─sdb2     8:18   0    3G  0 part
[root@server2 ~]# lsblk /dev/sdb1
NAME MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sdb1   8:17   0  9.5G  0 part /mydata/data
```

另外常用的一个选项是"-f"，它可以查看到文件系统类型，和文件系统的 uuid 和挂载点。

```
[root@xuexi ~]# lsblk -f
NAME   FSTYPE LABEL UUID                                 MOUNTPOINT
sda
├─sda1 ext4         77b5f0da-b0f9-4054-9902-c6cdacf29f5e /boot
├─sda2 ext4         f199fcb4-fb06-4bf5-a1b7-a15af0f7cb47 /
└─sda3 swap         6ae3975c-1a2a-46e3-87f3-d5bd3f1eff48 [SWAP]
sr0
sdb
├─sdb1 ext4         95e5b9d5-be78-43ed-a06a-97fd1de9a3fe
├─sdb2 ext2         45da2d94-190a-4548-85bb-b3c46ae6d9a7
└─sdb3
```

每个已经格式化的文件系统都有其类型和 uuid，而没有格式化的设备(如/dev/sdb3)，将只显示一个 Name 结果，表示该设备还未进行格式化。

5.5.2 blkid

虽然它有不少比较强大的功能，但一般只用它一个功能，就是查看器文件系统类型和 uuid。

```
[root@xuexi ~]# blkid
```

```
/dev/sda1: UUID="77b5f0da-b0f9-4054-9902-c6cdacf29f5e" TYPE="ext4"
/dev/sda2: UUID="f199fcb4-fb06-4bf5-a1b7-a15af0f7cb47" TYPE="ext4"
/dev/sda3: UUID="6ae3975c-1a2a-46e3-87f3-d5bd3f1eff48" TYPE="swap"
/dev/sdb1: UUID="95e5b9d5-be78-43ed-a06a-97fd1de9a3fe" TYPE="ext4"
/dev/sdb2: UUID="45da2d94-190a-4548-85bb-b3c46ae6d9a7" TYPE="ext2"
[root@xuexi ~]# blkid /dev/sdb1
/dev/sdb1: UUID="95e5b9d5-be78-43ed-a06a-97fd1de9a3fe" TYPE="ext4"
```

5.5.3 parted /dev/sda print 和 fdisk -l

```
parted /dev/sdb p
Model: VMware, VMware Virtual S (scsi)
Disk /dev/sdb: 21.5GB
Sector size (logical/physical): 512B/512B
Partition Table: gpt
Disk Flags:

Number  Start   End     Size    File system  Name              Flags
 1      1049kB  10.2GB  10.2GB  ext4
 2      10.2GB  13.5GB  3221MB  ext2         Linux filesystem
fdisk -l /dev/sda
Disk /dev/sda: 21.5 GB, 21474836480 bytes, 41943040 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: dos
Disk identifier: 0x000cb657

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *        2048      514047      256000   83  Linux
/dev/sda2          514048    37847039    18666496   83  Linux
/dev/sda3        37847040    41943039     2048000   82  Linux swap / Solaris
```

虽然 fdisk 和 gdisk 分别是 mbr 和 gpt 格式的专用工具，但是仅用于查看信息还是可以的。parted 能兼容两者，所以也可以。

5.5.4 file -s

```
[root@xuexi ~]# file -s /dev/sdb2
/dev/sdb2: Linux rev 1.0 ext2 filesystem data (large files)
```

5.5.5 du

du 命令用于评估文件的空间占用情况，它会统计每个文件的大小，统计时会递归统计目录中的文件，也就是说，它会遍历整个待统计目录，所以统计速度上可能并不理想。

```
du [OPTION]... [FILE]...
-a, --all：列出目录中所有文件的统计信息，默认只会列出目录中子目录的统计信息，而不列出文件的统计信息
-h, --human-readable：人性化显示大小
-0, --null：以空字符结尾，即"\0"而非换行的"\n"
-S, --separate-dirs：不包含子目录的大小
-s, --summarize：对目录做总的统计，不列出目录内文件的大小信息
-c,--total：对给出的文件或目录做总计。在统计非同一个目录文件大小时非常有用。见下文例子。
-d,--max-depth=N：只列出给定层次的目录统计，如果 N=0，则等价于"-s"
-x, --one-file-system：忽略不同文件系统上的文件，不对它们进行统计
-X, --exclude-from=FILE：从文件中读取要排除的文件
--exclude=PATTERN：指定要忽略不统计的文件
```

注意:

(1).上面的选项中，有些是不列出某些项，有些是不统计某些项，它们是不一样的。

(2).如果要统计的目录下挂载了一个文件系统，那么这个文件系统的大小也会被计入该目录的大小中。

```
[root@xuexi ~]# du -sh /etc
29M     /etc
[root@xuexi ~]# du -ah /tmp
4.0K    /tmp/b.txt
4.0K    /tmp/a
4.0K    /tmp/.ICE-unix
4.0K    /tmp/testdir/subdir
0       /tmp/testdir/a.log
```

```
8.0K    /tmp/testdir
24K     /tmp
[root@xuexi ~]# du -h --max-depth=1 /usr
15M     /usr/include
383M    /usr/lib64
132K    /usr/local
391M    /usr/share
4.0K    /usr/etc
118M    /usr/lib
44M     /usr/libexec
49M     /usr/src
32M     /usr/sbin
4.0K    /usr/games
75M     /usr/bin
1.1G    /usr
[root@xuexi ~]# du -h --max-depth=1 --exclude=/usr/lib64 /usr
15M     /usr/include
132K    /usr/local
391M    /usr/share
4.0K    /usr/etc
118M    /usr/lib
44M     /usr/libexec
49M     /usr/src
32M     /usr/sbin
4.0K    /usr/games
75M     /usr/bin
721M    /usr
```

搜索符合条件的文件，然后统计它们的总大小。结合 find 使用，效果极佳。

```
[root@xuexi ~]# find /boot/ -type f -name "*.img" -print0 | xargs -0 du -ch
28K     /boot/grub2/i386-pc/core.img
4.0K    /boot/grub2/i386-pc/boot.img
592K    /boot/initrd-plymouth.img
44M     /boot/initramfs-0-rescue-d13bce5e247540a5b5886f2bf8aabb35.img
17M     /boot/initramfs-3.10.0-327.el7.x86_64.img
16M     /boot/initramfs-3.10.0-327.el7.x86_64kdump.img
76M     total
```

请注意"-c"和"-s"统计的区别。

```
[root@xuexi ~]# find /boot/ -type f -name "*.img" -print0 | xargs -0 du -sh
28K     /boot/grub2/i386-pc/core.img
4.0K    /boot/grub2/i386-pc/boot.img
592K    /boot/initrd-plymouth.img
44M     /boot/initramfs-0-rescue-d13bce5e247540a5b5886f2bf8aabb35.img
17M     /boot/initramfs-3.10.0-327.el7.x86_64.img
16M     /boot/initramfs-3.10.0-327.el7.x86_64kdump.img
```

5.5.6 df

df 用于报告磁盘空间使用率，默认显示的大小是 1K 大小 block 数量，也就是以 k 为单位。

和 du 不同的是，df 是读取每个文件系统的 superblock 信息，所以评估速度非常快。由于是读取 superblock，所以如果目录下挂载了另一个文件系统，是不会将此挂载的文件系统计入目录大小的。注意，du 和 df 统计的结果是不一样的，如果对它们的结果不同有兴趣，可参考我的另一篇文章：详细分析 du 和 df 的统计结果为什么不一样。

如果用 df 统计某个文件的空间使用情况，将会转而统计该文件所在文件系统的空间使用情况。

```
df [OPTION]... [FILE]...
选项说明：
-h：人性化转换大小的显示单位
-i：统计 inode 使用情况而非空间使用情况
-l, --local：只列出本地文件系统的使用情况，不列出网络文件系统信息
-T, --print-type：同时输出文件系统类型
-t, --type=TYPE：只列出给定文件系统的统计信息
-x, --exclude-type=TYPE：指定不显示的文件系统类型的统计信息
[root@server2 ~]# df -hT
Filesystem     Type      Size  Used Avail Use% Mounted on
/dev/sda2      xfs        18G  2.3G   16G  13% /
```

```
devtmpfs       devtmpfs  904M     0  904M   0% /dev
tmpfs          tmpfs     913M     0  913M   0% /dev/shm
tmpfs          tmpfs     913M  8.6M  904M   1% /run
tmpfs          tmpfs     913M     0  913M   0% /sys/fs/cgroup
/dev/sda1      xfs       247M  110M  137M  45% /boot
tmpfs          tmpfs     183M     0  183M   0% /run/user/0
/dev/sdb1      ext4      9.3G   37M  8.8G   1% /mydata/data
[root@server2 ~]# df -i
Filesystem      Inodes  IUsed    IFree IUse% Mounted on
/dev/sda2     18666496 106474 18560022    1% /
devtmpfs        231218    388   230830    1% /dev
tmpfs           233586      1   233585    1% /dev/shm
tmpfs           233586    479   233107    1% /run
tmpfs           233586     13   233573    1% /sys/fs/cgroup
/dev/sda1       256000    330   255670    1% /boot
tmpfs           233586      1   233585    1% /run/user/0
/dev/sdb1       625856     14   625842    1% /mydata/data
```

5.5.7 dumpe2fs

用于查看 ext 类文件系统的 superblock 及块组信息。使用-h 选项将只显示 superblock 信息。

以下是 ext4 文件系统 superblock 的信息。

```
[root@xuexi ~]# dumpe2fs -h /dev/sda2
dumpe2fs 1.41.12 (17-May-2010)
Filesystem volume name:   <none>
Last mounted on:          /
Filesystem UUID:          f199fcb4-fb06-4bf5-a1b7-a15af0f7cb47
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode dir_index filetype needs_recovery extent flex_bg sparse_super large_file
huge_file uninit_bg dir_nlink extra_isize
Filesystem flags:         signed_directory_hash
Default mount options:    user_xattr acl
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              1166880
Block count:              4666624
Reserved block count:     233331
Free blocks:              4196335
Free inodes:              1111754
First block:              0
Block size:               4096
Fragment size:            4096
Reserved GDT blocks:      1022
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         8160
Inode blocks per group:   510
Flex block group size:    16
Filesystem created:       Sat Feb 25 11:48:47 2017
Last mount time:          Tue Jun  6 18:13:10 2017
Last write time:          Sat Feb 25 11:53:49 2017
Mount count:              6
Maximum mount count:      -1
Last checked:             Sat Feb 25 11:48:47 2017
Check interval:           0 (<none>)
Lifetime writes:          2657 MB
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
First inode:              11
Inode size:               256
Required extra isize:     28
Desired extra isize:      28
Journal inode:            8
Default directory hash:   half_md4
```

```
Directory Hash Seed:      d4e6493a-09ef-41a1-9d66-4020922f1aa9
Journal backup:           inode blocks
Journal features:         journal_incompat_revoke
Journal size:             128M
Journal length:           32768
Journal sequence:         0x00001bd9
Journal start:            23358
```

其中一个块组信息。

```
[root@xuexi ~]# dumpe2fs /dev/sda2 | tail -7
dumpe2fs 1.41.12 (17-May-2010)
Group 142: (Blocks 4653056-4666623) [INODE_UNINIT, ITABLE_ZEROED]
  Checksum 0x64ce, unused inodes 8160
  Block bitmap at 4194318 (+4294508558), Inode bitmap at 4194334 (+4294508574)
  Inode table at 4201476-4201985 (+4294515716)
  13568 free blocks, 8160 free inodes, 0 directories, 8160 unused inodes
  Free blocks: 4653056-4666623
  Free inodes: 1158721-1166880
```

5.6 挂载和卸载文件系统

在此，只简单介绍 mount 和 umount 的用法，至于实现挂载和卸载的机制和原理细节，参看挂载文件系统的细节。

5.6.1 mount

mount 用来显示挂载信息或者进行文件系统挂载，它的功能及其的强大(强大到离谱)，它不仅支持挂载非常多种文件系统，如 ext/xfs/nfs/smbfs/cifs(win 上的共享目录)等，还支持共享挂载点、继承挂载点(父子关系)、绑定挂载点、移动挂载点等等功能。在本文只介绍其最简单的挂载功能。

不同的文件系统挂载选项是有所差别的，在挂载过程中如果出错，应该 man mount 并查看对应文件系统的挂载选项。

mount 并非只能挂载文件系统，也可以将目录挂载到另一个目录下，其实它实现的是目录"硬链接"，默认情况下，是无法对目录建立硬链接的，但是通过 mount 可以完成绑定，绑定后两个目录的 inode 号是完全相同的，但尽管建立的是目录的"硬链接"，但其实也仅是拿来当软链接用。

以下是 ext 类文件系统的选项，可能有些选项是不支持其他文件系统的。

```
mount # 将显示当前已挂载信息
mount [-t 欲挂载文件系统类型 ] [-o 特殊选项] 设备名 挂载目录
-a  将/etc/fstab 文件里指定的挂载选项重新挂载一遍。
-t  支持 ext2/ext3/ext4/vfat/fat/iso9660(光盘默认格式)。不用-t 时默认会调用 blkid 来获取文件系统类型。
-n  不把挂载记录写在/etc/mtab 文件中，一般挂载会在/proc/mounts 中记录下挂载信息，然后同步到/etc/mtab，指定-n 表示不同步该挂载信息。
-o  指定挂载特殊选项。下面是两个比较常用的：
    loop  挂载镜像文件，如 iso 文件
    ro  只读挂载
    rw  读写挂载
    auto  相当于 mount -a
    dev 如果挂载的文件系统中有设备访问入口则启用它，使其可以作为设备访问入口
    default rw,suid,dev,exec,auto,nouser,async,and relatime
    async   异步挂载，只写到内存
    sync    同步挂载，通过挂载位置写入对方硬盘
    atime   修改访问时间，每次访问都修改 atime 会导致性能降低，所以默认是 noatime
    noatime 不修改访问时间，高并发时使用这个选项可以减少磁盘 IO
    nodiratime  不修改文件夹访问时间，高并发时使用这个选项可以减少磁盘 IO
    exec/noexec  挂载后的文件系统里的可执行程序是否可执行，默认是可以执行 exec，优先级高于权限的限定
    remount  重新挂载，此时可以不用指定挂载点。
    suid/nosuid 对挂载的文件系统启用或禁用 suid，对于外来设备最好禁用 suid
    _netdev 需要网络挂载时默认将停留在挂载界面直到加载网络了。使用_netdev 可以忽略网络正常挂载。如 NFS 开机挂载。
    user  允许普通用户进行挂载该目录，但只允许挂载者进行卸载该目录
    users  允许所有用户挂载和卸载该目录
    nouser  禁止普通用户挂载和卸载该目录，这是默认的，默认情况下一个目录不指定 user/users 时，将只有 root 能挂载
```

一般 user/users/nouser 都用在/etc/fstab 中，直接在命令行下使用这几个选项意义不是很大。

例如：

(1).挂载 CentOS 的安装镜像到/mnt。

```
mount /dev/cdrom /mnt
```

其实/dev/cdrom 是/dev/sr0 的一个软链接，/dev/sr0 是光驱设备，所以也可以用/dev/sr0 进行挂载。

```
mount /dev/sr0 /mnt
```

(2).重新挂载。

```
[root@xuexi ~]# mount -t ext4 -o remount /dev/sdb1 /data1
```

(3).重新挂载文件系统为可读写。

```
mount -t ext4 -o rw remount /dev/sdb1 /data1
```

(4).挂载 windows 的共享目录。

win 上共享文件的文件系统是 cifs 类型，要在 Linux 上挂载，必须得有 mount.cifs 命令，如果没有则安装 cifs-utils 包。

假设 win 上共享目录的 unc 路径为\\192.168.100.8\test，共享给的用户名和密码分别为 long3:123，要挂在 linux 上的/mydata 目录上。

```
shell> mount.cifs -o username="long3",password="123" //192.168.100.8/test /mydata
```

注意，如果是比较新版本的 win10(2017 年之后更新的版本)或较新版本的 win server，直接 mount.cifs 会报错：

```
[root@xuexi ~]# mount.cifs -o username="long3",password="123" //192.168.100.8/test /mnt
mount error(112): Host is down
Refer to the mount.cifs(8) manual page (e.g. man mount.cifs)
```

这是因为 2017 年微软的一个补丁禁用了 SMBv1 协议，通过 smbclient 的报告可知：

```
[root@xuexi ~]# yum -y install samba-client
[root@xuexi ~]# smbclient -L //192.168.100.8
Enter root's password:
protocol negotiation failed: NT_STATUS_CONNECTION_RESET
```

因此，在 mount 的时候指定 cifs(SMB)的版本号为 2.0 即可。

```
[root@xuexi ~]# mount.cifs -o username="long3",password="123",vers=2.0 //192.168.100.8/test /mnt
```

但是需要注意，在 CentOS 4,5,6 下的模块 cifs.ko 版本(较低)只能使用 SMBv1 协议，因此即使指定版本号也一样无效。只有在 CentOS 7 上才能使用 SMBv2 或 SMBv3。

(5).基于 ssh 挂载远程目录。

如何基于 ssh 像 NFS 一样挂载远程主机上的目录？可以通过 sshfs 工具，该工具在 fuse-sshfs 包中，这个包在 epel 源中提供。

```
yum -y install fuse-sshfs
```

例如，挂载 192.168.100.8 上的根目录到本地的/mnt 上。

```
sshfs 192.168.100.8:/ /mnt
```

卸载时直接 umount 即可。

关于 sshfs，详细内容见：https://www.cnblogs.com/f-ck-need-u/p/9104950.html

(6).挂载目录到另一个目录下。挂载目录时，挂载目录和挂载点的 inode 是相同的，它们两者的内容也是完全相同的。

```
mount --bind /mydata /mnt
```

(7).查看某个目录是否是挂载点，使用 mountpoint 命令。

```
[root@xuexi ~]# mountpoint /mydata/
/mydata/ is a mountpoint
[root@xuexi ~]# echo $?
0
[root@xuexi ~]# mountpoint /mnt
/mnt is not a mountpoint
[root@xuexi ~]# echo $?
1
```

挂载的参数信息存放在/proc/mounts(是/proc/self/mounts 的软链接)中，在/proc/self/mountstats 和/proc/mountinfo 里则记录了更详细的挂载信息。

```
[root@xuexi ~]# cat /proc/mounts
rootfs / rootfs rw 0 0
proc /proc proc rw,relatime 0 0
sysfs /sys sysfs rw,relatime 0 0
devtmpfs /dev devtmpfs rw,relatime,size=491000k,nr_inodes=122750,mode=755 0 0
devpts /dev/pts devpts rw,relatime,gid=5,mode=620,ptmxmode=000 0 0
tmpfs /dev/shm tmpfs rw,relatime 0 0
/dev/sda2 / ext4 rw,relatime,barrier=1,data=ordered 0 0
/proc/bus/usb /proc/bus/usb usbfs rw,relatime 0 0
/dev/sda1 /boot ext4 rw,relatime,barrier=1,data=ordered 0 0
```

```
none /proc/sys/fs/binfmt_misc binfmt_misc rw,relatime 0 0
```

文件系统是需要驱动支持的，没有驱动的文件系统也无法挂载，Linux 中支持的文件系统驱动在/lib/modules/$(uname -r)/kernel/fs 下。

```
[root@xuexi ~]# ls /lib/modules/$(uname -r)/kernel/fs/
autofs4  cachefiles  configfs  dlm       exportfs  ext3  fat      fuse  jbd   jffs2  mbcache.ko  nfs_common  nls       ubifs  xfs
btrfs    cifs        cramfs    ecryptfs  ext2      ext4  fscache  gfs2  jbd2  lockd  nfs         nfsd        squashfs  udf
```

5.6.2 直接挂载镜像文件

有时候需要挂载 CentOS 的镜像文件，在虚拟机中经常是将镜像放入虚拟机的 CD/DVD 虚拟光驱中，然后在 Linux 上对/dev/cdrom 进行挂载。其实/dev/cdrom 是/dev/sr0 的一个软链接，/dev/sr0 是 Linux 中的光驱，所以上面的过程相当于是将镜像文件通过虚拟软件的虚拟光驱和 linux 的光驱连接起来，这样只需要挂载 Linux 中的光驱就可以了。但是，在非虚拟环境中没有虚拟光驱，而且在 Linux 中的一个镜像文件难道一定要拷贝到主机上通过虚拟光驱进行连接吗？

mount 是一个极其强大的挂载工具，它支持挂载很多种文件类型，其中就支持挂载镜像文件，其实它连挂载目录都支持。

```
[root@xuexi ~]# mount -o loop CentOS-6.6-x86_64-bin-DVD2.iso /mnt
[root@xuexi ~]# lsblk
NAME     MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
loop0      7:0    0   1.2G  0 loop /mnt
sda        8:0    0    20G  0 disk
├─sda1   8:1    0   250M  0 part /boot
├─sda2   8:2    0  17.8G  0 part /
└─sda3   8:3    0     2G  0 part [SWAP]
sr0       11:0    1  1024M  0 rom
```

5.6.3 umount

```
umount 设备名或挂载目录
umount -lf 强制卸载
```

卸载时，既可以使用设备名也可以使用挂载点卸载。有时候挂载网络系统(如 NFS)时，设备名很长，这时候可以使用挂载点来卸载就方便多了。

如果用户正在访问某个目录或文件，使得卸载一直显示 Busy，使用 fuser -v DIR 可以知道谁正在访问该目录或文件。

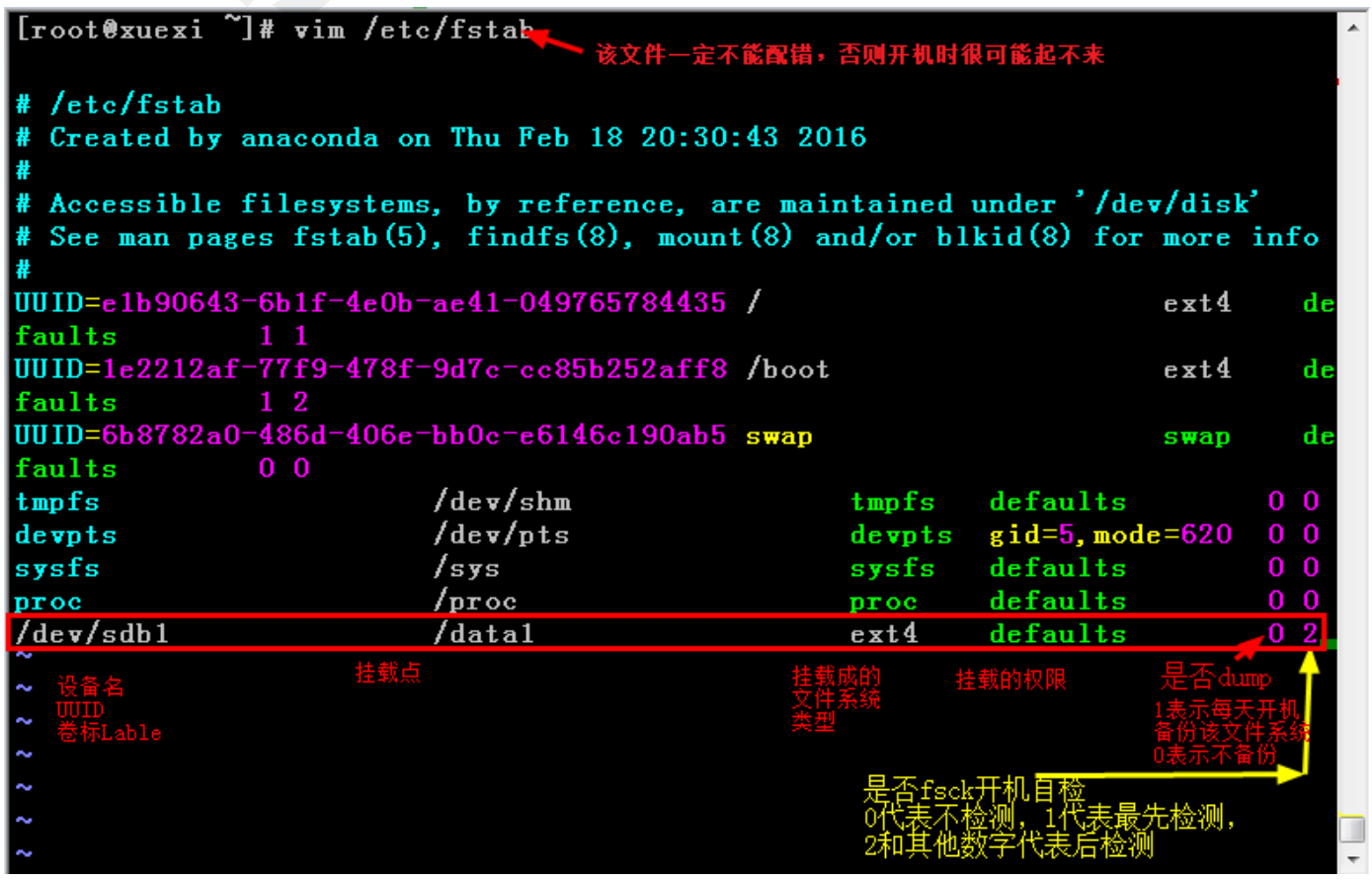
```
[root@xuexi ~]# fuser -v /root
                     USER        PID ACCESS COMMAND
/root:               root      37453 ..c.. bash
```

使用-k 选项 kill 掉正在使用目录或文件的进程，使用-km 选项 kill 掉文件系统上的所有进程，然后再 umount。

```
[root@xuexi ~]# fuser -km /mnt/cdrom;umount /mnt/cdrom
```

5.6.4 开机自动挂载/etc/fstab

通过将挂载选项写入到/etc/fstab 中，系统会自动挂载该文件中的配置项。但要注意，该文件在开机的前几个过程中就被读取，所以配置错误很可能会导致开机失败。



其中最后两列，它们分别表示备份文件系统和开机自检，一般都可以设置为 0。

由于能用的备份工具众多，没人会在这里设置备份，所以备份列设置为 0。

最后一列是开机自检设置列，开机自检调用的是 fsck 程序，所有有些 ext 类文件系统作为"/"时，可能会设置为 1，但是 fsck 是不支持 xfs 文件系统的，所以对于 xfs 文件系统而言，该项必须设置为 0。

其实无需考虑那么多，直接将这两列设置为 0 就可以了。

5.6.5 修复错误的/etc/fstab

万一/etc/fstab 配置错误，导致开机无法加载。这时提示输入 root 密码进入单人维护模式，只不过担任模式下根文件系统是只读的，哪怕是 root 也无法直接修改/etc/fstab，所以应该将"/"文件系统进行重新挂载。

执行下面的命令，重挂载根分区，并给读写权限，再去修改错误的 fstab 文件记录，再重启。

```
[root@xuexi ~]# mount -n -o remount,rw /
```

5.6.6 按需自动挂载(autofs)

使用 autofs 实现需要挂载时就挂载，不需要挂载时 5 分钟后自动卸载。但是在实际环境中基本不会使用按需挂载。

autofs 是一个服务程序，需要让其运行在后台，可以用来挂 NFS，也可挂本地的文件系统。

默认不装 autofs，需要自己装。

```
[root@xuexi ~]# yum install -y autofs
```

autofs 实现按需挂载的方式是指定监控目录，可在其配置文件/etc/auto.master 中指定。

/etc/auto.master 里面只有两列：第一列是监控目录；第二列是记录挂载选项的文件，该文件可以随便取名。

```
[root@xuexi ~]# cat /etc/auto.master
/share  /etc/auto.mount    # 监控/share 目录，使用/etc/auto.nfs 记录挂载选项
```

上述监控的/share 目录，其实这是监控的父目录，在此目录下的目录如/share/data 目录可以作为挂载点，当访问到/share/data 时就被监控到，然后会按照挂载选项将挂载设备挂载到/share/data 上。

上述配置中配置的挂载选项文件是/etc/auto.mount，所以建立此文件，写入挂载选项。

```
[root@xuexi ~]# cat /etc/auto.mount
#cd              -fstype=iso9660,ro,nosuid,nodev :/dev/cdrom

# the following entries are samples to pique your imagination
#linux          -ro,soft,intr           ftp.example.org:/pub/linux
#boot           -fstype=ext2            :/dev/hda1
#floppy         -fstype=auto            :/dev/fd0
#floppy         -fstype=ext2            :/dev/fd0
#e2floppy       -fstype=ext2            :/dev/fd0
#jaz            -fstype=ext2            :/dev/sdc1
#removable      -fstype=ext2            :/dev/hdd
```

该文件有 3 列：

第一列指定的是在/etc/auto.master 指定的/share 下的目录/share/data，它是真正的被监控路径，也是挂载点。可使用相对路径 data 表示/share/data。

第二列是 mount 的选项，前面使用一个"-"表示，该列可有可无。

第三列是待挂载设备，可以是 NFS 服务端的共享目录，也可以本地设备。

```
[root@xuexi ~]# vim /etc/auto.mount
data    -rw,bg,soft,rsize=32768,wsize=32768      192.168.100.61:/data
```

上面的配置表示当访问到/share/data 时，自动使用参数(rw,bg,soft,rsize=32768,wsize=32768)挂载远端 192.168.100.61 的/data 目录到/share/data 上。

剩下的步骤就是启动 autofs 服务。

```
[root@xuexi data]# /etc/init.d/autofs restart
```

5.7 swap 分区

虽说个人电脑上基本已经无需设置 swap 分区了，但是在服务器上还是应该准备 swap 分区，以做到有备无患和防止众多"玄学"问题。

5.7.1 查看 swap 使用情况

```
[root@xuexi ~]# free
             total       used       free     shared    buffers     cached
Mem:       1906488     349376    1557112        200      16920     200200
```

```
-/+ buffers/cache:     132256    1774232
Swap:       2097148          0    2097148
[root@xuexi ~]# free -m        # 以 MB 显示
             total       used       free     shared    buffers     cached
Mem:          1861        341       1520          0         16        195
-/+ buffers/cache:        129       1732     # 这个是真正的可用内存空间
Swap:         2047          0       2047     # 这个是 swap 空间，发现一点都没被用
```

使用 mount/lsblk 等可以查看出哪个分区在充当 swap 分区。使用 swapon -s 也可以直接查看出。

```
[root@server2 ~]# swapon -s
Filename                                Type            Size    Used    Priority
/dev/sda3                               partition       2047996 37064   -1
```

5.7.2 添加 swap 分区

(1).可以新分一个区，在分区时指定其分区的 ID 号为 SWAP 类型。

mbr 和 gpt 格式的磁盘上这个 ID 可能不太一样，不过一般 gpt 中的格式是在 mbr 格式的 ID 后加上两位数的数值，如 mbr 中 swap 的类型 ID 为 82，在 gpt 中则是 8200，在 mbr 中 linux filesystem 类型的 ID 为 83，在 gpt 中则为 8300，在 mbr 中 lvm 的 ID 为 8e，在 gpt 中为 8e00。

(2).格式化为 swap 分区：mkswap

```
[root@xuexi ~]# mkswap /dev/sdb5
Setting up swapspace version 1, size = 1951096 KiB
no label, UUID=02e5af44-2a16-479d-b689-4e100af6adf5
```

(3).加入 swap 分区空间(swapon)：

```
[root@xuexi ~]# swapon /dev/sdb5
[root@xuexi ~]# free -m
             total       used       free     shared    buffers     cached
Mem:          1861        343       1517          0         16        196
-/+ buffers/cache:        131       1730
Swap:         3953          0       3953
```

(4).取消 swap 分区空间(swapoff)：

```
[root@xuexi ~]# swapoff /dev/sdb5
[root@xuexi ~]# free -m
             total       used       free     shared    buffers     cached
Mem:          1861        343       1518          0         16        196
-/+ buffers/cache:        130       1731
Swap:            0          0          0
```

(5).开机自动加载 swap 分区：

修改/etc/fstab，加上一行。

```
/dev/sda3    swap    swap    defaults    0    0
```

第6章 LVM

6.1 lvm 相关的概念和机制

LVM(Logical Volume Manager)可以让分区变得弹性，可以随时随地的扩大和缩小分区大小，前提是该分区是 LVM 格式的。

lvm 需要使用的软件包为 lvm2，一般在 CentOS 发行版中都已经预安装了。

- **PV(Physical Volume)即物理卷**

硬盘分区后(还未格式化为文件系统)使用 pvcreate 命令可以将分区创建为 pv，要求分区的 system ID 为 8e，即为 LVM 格式的系统标识符。

- **VG(Volume Group)即卷组**

将多个 PV 组合起来，使用 vgcreate 命令创建成卷组，这样卷组包含了多个 PV 就比较大了，相当于重新整合了多个分区后得到的磁盘。虽然 VG 是整合多个 PV 的，但是创建 VG 时会将 VG 所有的空间根据指定的 PE 大小划分为多个 PE，在 LVM 模式下的存储都以 PE 为单元，类似于文件系统的 Block。

- **PE(Physical Extend)**

PE 是 VG 中的存储单元。实际存储的数据都是存储在这里面的。

- **LV(Logical Volume)**

VG 相当于整合过的硬盘，那么 LV 就相当于分区，只不过该分区是通过 VG 来划分的。VG 中有很多 PE 单元，可以指定将多少个 PE 划分给一个 LV，也可以直接指定大小(如多少兆)来划分。划分为 LV 之后就相当于划分了分区，只需再对 LV 进行格式化即可变成普通的文件系统。

通俗地讲，非 LVM 管理的分区步骤是将硬盘分区，然后将分区格式化为文件系统。而使用 LVM，则是在硬盘分区为特定的 LVM 标识符的分区后将其转变为 LVM 可管理的 PV，其实 PV 仍然类似于分区，然后将几个 PV 整合为类似于磁盘的 VG，最后划分 VG 为 LV，此时 LV 就成了 LVM 可管理的分区，只需再对其格式化即可成为文件系统。

- **LE(logical extent)**

PE 是物理存储单元，而 LE 则是逻辑存储单元，也即为 lv 中的逻辑存储单元，和 pe 的大小是一样的。从 vg 中划分 lv，实际上是从 vg 中划分 vg 中的 pe，只不过划分 lv 后它不再称为 pe，而是成为 le。

能够通过 LVM 伸缩容量，其实现的方法就是将 LV 里空闲的 PE 移出，或向 LV 中添加空闲的 PE。

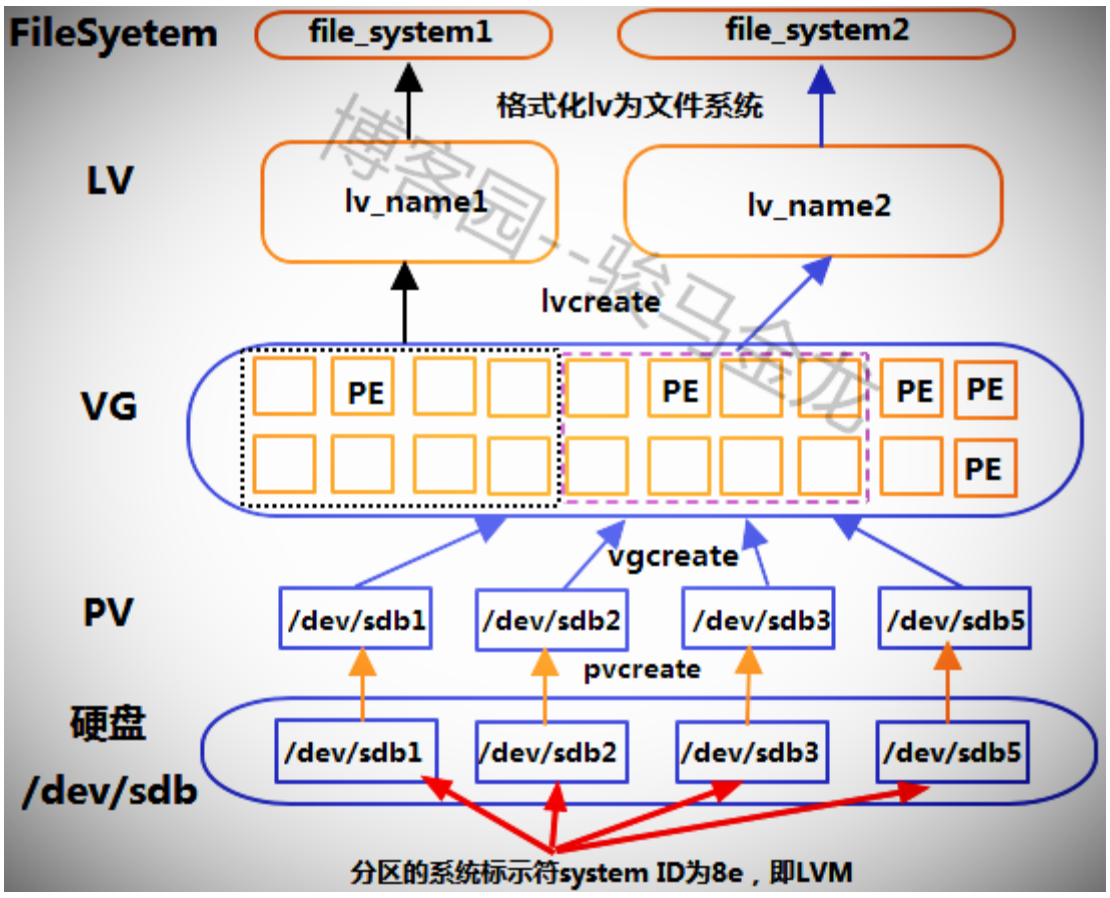
6.2 LVM 的写入机制

LV 是从 VG 中划分出来的，LV 中的 PE 很可能来自于多个 PV。在向 LV 存储数据时，有多种存储机制，其中两种是：

- 线性模式(linear)：先写完来自于同一个 PV 的 PE，再写来自于下一个 PV 的 PE。
- 条带模式(striped)：一份数据拆分成多份，分别写入该 LV 对应的每个 PV 中，所以读写性能较好，类似于 RAID 0。

尽管 striped 读写性能较好也不建议使用该模式，因为 lvm 的着重点在于弹性容量扩展而非性能，要实现性能应该使用 RAID 来实现，而且使用 striped 模式时要进行容量的扩展和收缩将比较麻烦。默认的是使用线性模式。

6.3 LVM 实现图解



6.4 LVM 的实现

以上图为例。

首先需要将/dev/sdb 下的分区/dev/sdb{1,2,3,5}修改为 LVM 格式的标识符(/dev/sdb4 在后面扩容实验中使用)。mbr 格式下标识符是 8e，gpt 格式下是 8300。

以下是 gpt 分区表格式的部分分区信息。

```
[root@server2 ~]# gdisk -l /dev/sdb
GPT fdisk (gdisk) version 0.8.6

Number  Start (sector)    End (sector)  Size       Code
   1            2048        20000767   9.5 GiB     8E00
   2        20000768        26292223   3.0 GiB     8E00
   3        26292224        29296639   1.4 GiB     8E00
   4        29296640        33202175   1.9 GiB     8300
   5        33202176        37109759   1.9 GiB     8E00
```

6.4.1 管理 PV

管理 PV 有几个命令：pvscan、pvdisplay、pvcreate、pvremove 和 pvmove。

命令很简单，基本都不需要任何选项。

功能	命令
创建 PV	pvcreate
扫描并列出所有的 pv	pvscan
列出 pv 属性信息	pvdisplay
移除 pv	pvremove
移动 pv 中的数据	pvmove

其中 pvscan 搜索目前有哪些 pv，扫描之后将结果放在缓存中；pvdisplay 会显示每个 pv 的详细信息，如 PV name 和 pv size 以及所属的 VG 等。

直接将上述/dev/sdb{1,2,3,5}创建为 pv。

```
[root@server2 ~]# pvcreate -y /dev/sdb1 /dev/sdb2 /dev/sdb3 /dev/sdb5     #  -y 选项用于自动回答 yes
  Wiping ext4 signature on /dev/sdb1.
  Physical volume "/dev/sdb1" successfully created
  Wiping ext2 signature on /dev/sdb2.
  Physical volume "/dev/sdb2" successfully created
  Physical volume "/dev/sdb3" successfully created
  Physical volume "/dev/sdb5" successfully created
```

使用 pvscan 来查看哪些 pv 和基本属性。

```
[root@server2 ~]# pvscan
  PV /dev/sdb1         lvm2 [9.54 GiB]
  PV /dev/sdb2         lvm2 [3.00 GiB]
  PV /dev/sdb5         lvm2 [1.86 GiB]
  PV /dev/sdb3         lvm2 [1.43 GiB]
  Total: 4 [15.83 GiB] / in use: 0 [0   ] / in no VG: 4 [15.83 GiB]
```

注意最后一行显示的是"pv 的总容量/已使用的 pv 容量/空闲的 pv 容量"

使用 pvdisplay 查看其中一个 pv 的属性信息。

```
[root@server2 ~]# pvdisplay /dev/sdb1
  "/dev/sdb1" is a new physical volume of "9.54 GiB"
  --- NEW Physical volume ---
  PV Name               /dev/sdb1
  VG Name
  PV Size               9.54 GiB
  Allocatable           NO
  PE Size               0
  Total PE              0
```

```
  Free PE               0
  Allocated PE          0
  PV UUID               fRChUf-CL8d-2UwC-d94R-xa8a-MRYa-yvgFJ9
```

pvdisplay 还有一个很重要的选项"-m"，可以查看该设备中 PE 的使用分布图。以下是某次显示结果。

```
[root@server2 ~]# pvdisplay -m /dev/sdb2
  --- Physical volume ---
  PV Name               /dev/sdb2
  VG Name               firstvg
  PV Size               3.00 GiB / not usable 16.00 MiB
  Allocatable           yes
  PE Size               16.00 MiB
  Total PE              191
  Free PE               100
  Allocated PE          91
  PV UUID               uVgv3q-ANyy-02M1-wmGf-zmFR-Y16y-qLgNMV

  --- Physical Segments ---
  Physical extent 0 to 0:                 # 说明第 0 个 PE 正被使用。PV 中 PE 的序号是从 0 开始编号的
    Logical volume      /dev/firstvg/first_lv
    Logical extents     450 to 450    # 该 PE 在 LV 中的第 450 个 LE 位置上
  Physical extent 1 to 100:               # 说明/dev/sdb2 中，1-100 序号的 PE 是空闲未被使用的
    FREE
  Physical extent 101 to 190:             # 说明 101-190 序号的 PE 是正使用中的，其在 LV 中的位置是 551-640
    Logical volume      /dev/firstvg/first_lv
    Logical extents     551 to 640
```

知道了 PE 的分布，就可以轻松地使用 pvmove 命令在设备之间进行 PE 数据的移动。具体关于 pvmove 的用法，见"收缩 lvm 磁盘"部分。

再测试 pvremove，移除/dev/sdb5，然后将其添加回 pv。

```
[root@server2 ~]# pvremove /dev/sdb5
  Labels on physical volume "/dev/sdb5" successfully wiped
[root@server2 ~]# pvcreate /dev/sdb5
  Physical volume "/dev/sdb5" successfully created
```

6.4.2 管理 VG

管理 VG 也有几个命令。

功能	命令
创建 VG	vgcreate
扫描并列出所有的 vg	vgscan
列出 vg 属性信息	vgdisplay
移除 vg，即删除 vg	vgremove
从 vg 中移除 pv	vgreduce
将 pv 添加到 vg 中	vgextend
修改 vg 属性	vgchange

同样，vgscan 搜寻有几个 vg 并显示 vg 的基本属性，vgcreate 是创建 vg，vgdisplay 是列出 vg 的详细信息，vgremove 是删除整个 vg，vgextend 用于扩展 vg 即将 pv 添加到 vg 中，vgreduce 是将 pv 移除出 vg。除此之外还有一个命令 vgchange，用于改变 vg 的属性，如修改 vg 的状态为激活状态或未激活状态。

创建一个 vg，并将上述 4 个 pv /dev/sdb{1,2,3,5}都添加到该 vg 中。注意 vg 是需要命名的，vg 可以等同于磁盘的层次，而磁盘是有名称的，如/dev/sdb，/dev/sdc 等。同时创建 vg 时可以使用-s 选项指定 pe 的大小，如果不指定默认为 4M。

```
[root@server2 ~]# vgcreate -s 16M firstvg /dev/sdb{1,2,3,5}
  Volume group "firstvg" successfully created
```

此处创建的 vg 名称为 firstvg，指定 pe 大小为 16M。创建 vg 后，是很难再修改 pe 大小的，只有空数据的 vg 可以修改，但这样还不如重新创建 vg。

注意，lvm1 中每个 vg 中只能有 65534 个 pe，所以指定 pe 的大小能改变每个 vg 的最大容量。但在 lvm2 中已经没有该限制了，而现在说的 lvm 一般都指 lvm2，这也是默认的版本。

创建了 vg 实际上是在/dev 目录下管理了一个 vg 目录/dev/firstvg，不过只有在创建了 lv 该目录才会被创建，而该 vg 中创建 lv，将会在该目录下生成链接文件指向/dev/dm 设备。

再看看 vgscan 和 vgdisplay 吧。

```
[root@server2 ~]# vgscan
  Reading all physical volumes.  This may take a while...
  Found volume group "firstvg" using metadata type lvm2
[root@server2 ~]# vgdisplay firstvg
  --- Volume group ---
  VG Name               firstvg
  System ID
  Format                lvm2
  Metadata Areas        4
  Metadata Sequence No  1
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                0
  Open LV               0
  Max PV                0
  Cur PV                4
  Act PV                4
  VG Size               15.80 GiB
  PE Size               16.00 MiB
  Total PE              1011
  Alloc PE / Size       0 / 0
  Free  PE / Size       1011 / 15.80 GiB
  VG UUID               GLwZTC-zUj9-mKas-CJ5m-Xf91-5Vqu-oEiJGj
```

从 vg 中移除一个 pv，如/dev/sdb5，再 vgdisplay，发现 pv 少了一个，pe 相应的也减少了。

```
[root@server2 ~]# vgreduce firstvg /dev/sdb5
  Removed "/dev/sdb5" from volume group "firstvg"
[root@server2 ~]# vgdisplay firstvg
  --- Volume group ---
  VG Name               firstvg
  System ID
  Format                lvm2
  Metadata Areas        3
  Metadata Sequence No  2
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                0
  Open LV               0
  Max PV                0
  Cur PV                3
  Act PV                3
  VG Size               13.94 GiB
  PE Size               16.00 MiB
  Total PE              892
  Alloc PE / Size       0 / 0
  Free  PE / Size       892 / 13.94 GiB
  VG UUID               GLwZTC-zUj9-mKas-CJ5m-Xf91-5Vqu-oEiJGj
```

再将/dev/sdb5 加入 vg。

```
[root@server2 ~]# vgextend firstvg /dev/sdb5
  Volume group "firstvg" successfully extended
```

vgchange 用于设置卷组的活动状态，卷组的激活状态主要影响的是 lv。使用-a 选项来设置。

将 firstvg 设置为活动状态(active yes)。

```
vgchange -a y firstvg
```

将 firstvg 设置为非激活状态(active no)。

```
vgchange -a n firstvg
```

6.4.3 管理 LV

有了 vg 之后就可以根据 vg 进行分区，即创建 LV。管理 lv 也有类似的一些命令。

功能	命令
创建 LV	lvcreate
扫描并列出所有的 lv	lvscan
列出 lv 属性信息	lvdisplay
移除 lv，即删除 lv	lvremove
缩小 lv 容量	lvreduce(lvresize)
增大 lv 容量	lvextend(lvresize)
改变 lv 容量	lvresize

对于 lvcreate 命令有几个选项：

```
lvcreate {-L size(M/G) | -l PEnum} -n lv_name vg_name
-L：根据大小来创建 lv，即分配多大空间给此 lv
-l：根据 PE 的数量来创建 lv，即分配多少个 pe 给此 lv
-n：指定 lv 的名称
```

前面创建的 vg 有 1011 个 PE，总容量为 15.8G。

```
[root@server2 ~]# vgdisplay | grep PE
  PE Size               16.00 MiB
  Total PE              1011
  Alloc PE / Size       0 / 0
  Free  PE / Size       1011 / 15.80 GiB
```

使用-L 和-l 分别创建名称为 first_lv 和 sec_lv 的 lv。

```
[root@server2 ~]# lvcreate -L 5G -n first_lv firstvg
  Logical volume "first_lv" created.
[root@server2 ~]# lvcreate -l 160 -n sec_lv firstvg
  Logical volume "sec_lv" created.
```

创建 lv 后，将在/dev/firstvg 目录中创建对应 lv 名称的软链接文件，同时也在/dev/mapper 目录下创建链接文件，它们都指向/dev/dm 设备。

```
[root@server2 ~]# ls -l /dev/firstvg/
total 0
lrwxrwxrwx 1 root root 7 Jun  9 23:41 first_lv -> ../dm-0
lrwxrwxrwx 1 root root 7 Jun  9 23:42 sec_lv -> ../dm-1
[root@server2 ~]# ll /dev/mapper/
total 0
crw------- 1 root root 10, 236 Jun  6 02:44 control
lrwxrwxrwx 1 root root       7 Jun  9 23:41 firstvg-first_lv -> ../dm-0
lrwxrwxrwx 1 root root       7 Jun  9 23:42 firstvg-sec_lv -> ../dm-1
```

使用 lvscan 和 lvdisplay 查看 lv 信息。需要注意的是，如果 lvdisplay 要显示某一个指定的 lv，需要指定其全路径，而不能简单的指定 lv 名，当然如果不指定任何参数将显示所有 lv 的信息。

```
[root@server2 ~]# lvscan
  ACTIVE            '/dev/firstvg/first_lv' [5.00 GiB] inherit
  ACTIVE            '/dev/firstvg/sec_lv' [2.50 GiB] inherit
[root@server2 ~]# lvdisplay /dev/firstvg/first_lv
  --- Logical volume ---
  LV Path                /dev/firstvg/first_lv
  LV Name                first_lv
  VG Name                firstvg
  LV UUID                f3cRXJ-vucN-aAw3-HRbX-Fhnq-mW6c-kmL7WA
  LV Write Access        read/write
  LV Creation host, time server2.longshuai.com, 2017-06-09 23:41:42 +0800
  LV Status              available
```

```
# open                 0
LV Size                5.00 GiB
Current LE             320
Segments               1
Allocation             inherit
Read ahead sectors     auto
- currently set to     8192
Block device           253:0
```

关于 lv 其他的命令留在后文 lvm 扩容和缩减的部分进行演示。

6.4.4 格式化 lv 为文件系统

再对 lv 进行格式化，即可形成文件系统，然后进行挂载使用。

```
[root@server2 ~]# mke2fs -t ext4 /dev/firstvg/first_lv
```

对于 lv 格式化的文件系统类型如何查看？

可以先挂载再查看。

```
[root@server2 ~]# mount /dev/firstvg/first_lv /mnt
[root@server2 ~]# mount | grep /mnt
/dev/mapper/firstvg-first_lv on /mnt type ext4 (rw,relatime,data=ordered)
```

也可以使用 file -s 查看，但是由于/dev/firstvg 和/dev/mapper 下的 lv 都是链接到/dev/下块设备的链接文件，所以只能对块设备进行查看，否则查看的结果也仅仅只是个链接文件类型。

```
[root@server2 ~]# file -s /dev/dm-0
/dev/dm-0: Linux rev 1.0 ext4 filesystem data, UUID=f2a3b608-f4e9-431b-8c34-9c75eaf7d3b5 (needs journal recovery) (extents) (64bit) (large files) (huge files)
```

再去看看/dev/sdb 的情况。

```
[root@server2 ~]# lsblk -f /dev/sdb
NAME                  FSTYPE      LABEL UUID                                   MOUNTPOINT
sdb
├─sdb1                LVM2_member       fRChUf-CL8d-2UwC-d94R-xa8a-MRYa-yvgFJ9
│ ├─firstvg-first_lv ext4                f2a3b608-f4e9-431b-8c34-9c75eaf7d3b5   /mnt
│ └─firstvg-sec_lv
├─sdb2                LVM2_member       uVgv3q-ANyy-O2M1-wmGf-zmFR-Y16y-qLgNMV
├─sdb3                LVM2_member       L1byov-fbjK-M48t-Uabz-Ljn8-Q74C-ncdv8h
├─sdb4
└─sdb5                LVM2_member       Lae2vc-VfyS-QoNS-rz2h-IXUv-xKQc-Q6YCxQ
```

6.5 对 lvm 磁盘扩容

lvm 最大的优势就是其可伸缩性，而其伸缩性又更偏重于扩容，这是使用 lvm 的最大原因。

扩容的实质是将 vg 中空闲的 pe 添加到 lv 中，所以只要 vg 中有空闲的 pe，就可以进行扩容，即使没有空闲的 pe，也可以添加 pv，将 pv 加入到 vg 中增加空闲 pe。

扩容的两个关键步骤如下：

- (1).使用 lvextend 或者 lvresize 添加更多的 pe 或容量到 lv 中
- (2).使用 resize2fs 命令(xfs 则使用 xfs_growfs)将 lv 增加后的容量增加到对应的文件系统中(此过程是修改文件系统而非 LVM 内容)

例如，将一直没用到的/dev/sdb4 作为 first_lv 的扩容来源。首先将/dev/sdb4 创建成 pv，加入到 firstvg 中。

```
[root@server2 ~]# parted /dev/sdb toggle 4 lvm
[root@server2 ~]# pvcreate /dev/sdb4
[root@server2 ~]# vgextend firstvg /dev/sdb4
```

查看 firstvg 中 空闲的 pe 数量。

```
[root@server2 ~]# vgdisplay firstvg  | grep -i pe
  Open LV               1
  PE Size               16.00 MiB
  Total PE              1130
  Alloc PE / Size       480 / 7.50 GiB
  Free  PE / Size       650 / 10.16 GiB
```

现在 vg 中有 650 个 PE 共 10.16G 容量可用。将其全部添加到 first_lv 中，有两种方式添加：按容量大小添加和按 PE 数量添加。

```
[root@server2 ~]# umount /dev/firstvg/first_lv
[root@server2 ~]# lvextend -L +5G /dev/firstvg/first_lv   # 按容量大小添加
[root@server2 ~]# vgdisplay firstvg  | grep -i pe
  Open LV               1
  PE Size               16.00 MiB
  Total PE              1130
  Alloc PE / Size       800 / 12.50 GiB
  Free  PE / Size       330 / 5.16 GiB
[root@server2 ~]# lvextend -l +330 /dev/firstvg/first_lv  # 按 PE 数量添加
[root@server2 ~]# lvscan
  ACTIVE            '/dev/firstvg/first_lv' [15.16 GiB] inherit
  ACTIVE            '/dev/firstvg/sec_lv' [2.50 GiB] inherit
```

也可以使用 lvresize 来增加 lv 的容量方法和 lvextend 一样。如：

```
lvresize -L +5G /dev/firstvg/first_lv
lvresize -l +330 /dev/firstvg/first_lv
```

将 first_lv 挂载，查看该 lv 对应文件系统的容量。

```
[root@server2 ~]# mount /dev/mapper/firstvg-first_lv /mnt
[root@server2 ~]# df -hT /mnt
Filesystem                    Type  Size  Used Avail Use% Mounted on
/dev/mapper/firstvg-first_lv ext4  4.8G   20M  4.6G   1% /mnt
```

发现容量并没有增加，为什么呢？因为只是 lv 的容量增加了，而文件系统的容量却没有增加。所以使用 resize2fs 工具来改变 ext 文件系统的大小，如果是 xfs 文件系统，则使用 xfs_growfs。

首先简单看下 resize2fs 工具的使用说明。

```
NAME
       resize2fs - ext2/ext3/ext4 file system resizer

SYNOPSIS
       resize2fs [ -fFpPM ] [ -d debug-flags ] [ -S RAID-stride ] device [ size ]

DESCRIPTION
       The resize2fs program will resize ext2, ext3, or ext4 file systems.  It can
       be used to enlarge or shrink an unmounted file system  located  on  device.
       If  the  filesystem  is  mounted,  it can be used to expand the size of the
       mounted filesystem, assuming the kernel supports on-line resizing.  (As  of
       this  writing, the Linux 2.6 kernel supports on-line resize for filesystems
       mounted using ext3 and ext4.).
```

可见，该工具可用于增大和缩减已卸载的设备对应的文件系统大小，对于 linux 2.6 内核之后的版本，还支持在线 resize 而无需卸载，但在实验过程中好像不支持在线缩减，只能先卸载。

一般无需使用选项，直接使用 resize2fs device 的方式即可，如果失败则尝试使用-f 选项强制改变大小。

```
[root@server2 ~]# resize2fs  /dev/firstvg/first_lv
resize2fs 1.42.9 (28-Dec-2013)
Filesystem at /dev/firstvg/first_lv is mounted on /mnt; on-line resizing required
old_desc_blocks = 1, new_desc_blocks = 2
The filesystem on /dev/firstvg/first_lv is now 3973120 blocks long.
[root@server2 ~]# df -hT | grep -i /mnt
/dev/mapper/firstvg-first_lv ext4       15G   25M   15G   1% /mnt
```

再查看，size 已经变为 15G 了。

6.6 收缩 lvm 磁盘

不用考虑收缩的功能，且 xfs 文件系统也不支持收缩。不过，看看如何收缩，可以加深 lvm 的理解。

目前 first_lv 的容量为 15.16G。

```
[root@server2 ~]# lvscan
  ACTIVE            '/dev/firstvg/first_lv' [15.16 GiB] inherit
  ACTIVE            '/dev/firstvg/sec_lv' [2.50 GiB] inherit
```

而 pv 的使用情况则如下：

```
[root@server2 ~]# pvscan
  PV /dev/sdb1   VG firstvg   lvm2 [9.53 GiB / 0    free]
  PV /dev/sdb2   VG firstvg   lvm2 [2.98 GiB / 0    free]
  PV /dev/sdb3   VG firstvg   lvm2 [1.42 GiB / 0    free]
  PV /dev/sdb5   VG firstvg   lvm2 [1.86 GiB / 0    free]
  PV /dev/sdb4   VG firstvg   lvm2 [1.86 GiB / 0    free]
  Total: 5 [17.66 GiB] / in use: 5 [17.66 GiB] / in no VG: 0 [0   ]
```

如果想回收/dev/sdb2 的 2.98G 呢？收缩的步骤和扩容的步骤相反。

(1). 首先卸载设备并使用 resize2fs 收缩文件系统的容量为目标大小

这里要收缩 2.98G，原有 15.16G，所以文件系统的目标容量为 12.18G 约算做 12470M(因为 resize2fs 不能接受小数点的 size 参数，所以换算成整数，也可以直接算为 12G)，计算大小时应尽量多给出一点点的容量，所以此处算作收缩 3.18G，目标 12G。

```
[root@server2 ~]# umount /mnt
[root@server2 ~]# resize2fs /dev/firstvg/first_lv 12G
resize2fs 1.42.9 (28-Dec-2013)
Please run 'e2fsck -f /dev/firstvg/first_lv' first.
```

提示需要先运行 e2fsck -f /dev/Myvg/first_lv，主要是为了检查是否修改后的大小会影响数据。

```
[root@server2 ~]# e2fsck -f /dev/firstvg/first_lv
e2fsck 1.42.9 (28-Dec-2013)
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
/dev/firstvg/first_lv: 11/999424 files (0.0% non-contiguous), 101892/3973120 blocks
[root@server2 ~]# resize2fs /dev/firstvg/first_lv 12G
resize2fs 1.42.9 (28-Dec-2013)
Resizing the filesystem on /dev/firstvg/first_lv to 3145728 (4k) blocks.
The filesystem on /dev/firstvg/first_lv is now 3145728 blocks long.
```

(2). 再收缩 lv。可以直接使用"-L"指定收缩容量，也可以使用"-l"指定收缩的 PE 数量。

例如此处使用-L 来收缩。

```
[root@server2 ~]# lvreduce -L -3G /dev/firstvg/first_lv
  Rounding size to boundary between physical extents: 2.99 GiB
  WARNING: Reducing active logical volume to 12.16 GiB
  THIS MAY DESTROY YOUR DATA (filesystem etc.)
Do you really want to reduce first_lv? [y/n]: y
  Size of logical volume firstvg/first_lv changed from 15.16 GiB (970 extents) to 12.16 GiB (779 extents).
  Logical volume first_lv successfully resized.
```

发现有警告：可能会损毁你的数据。如果在该 lv 下存储的实际数据大于收缩后的容量，那么肯定会损毁一部分数据，但是如果存储的数据小于收缩后的容量，那么就不会损毁任何数据，这是 lvm 无损修改分区大小的优点。此处由于在 lv 下完全没有存储数据，所以无需担心会损毁，直接 y 确定 reduce。

(3). pvmove 移动 PE

上面的过程已经释放了 3G 大小的 PE，但是这部分 PE 来源于何处？是否可以判断此时能否移除/dev/sdb2？

首先查看哪些 PV 上有空闲的 PE。

```
[root@server2 ~]# pvdisplay | grep 'PV Name\|Free'
  PV Name               /dev/sdb1
  Free PE               0
  PV Name               /dev/sdb2
  Free PE               0
  PV Name               /dev/sdb3
  Free PE               91
  PV Name               /dev/sdb5
  Free PE               0
  PV Name               /dev/sdb4
  Free PE               100
```

可见，此时空闲的 PE 分布在/dev/sdb4 和/dev/sdb3 上，/dev/sdb2 并不能卸载。

那么现在需要做的事情是将/dev/sdb2 上的 PE 移到其他设备上，使/dev/sdb2 空闲出来。使用 pvmove 命令，该命令的执行内部会涉及到不少步骤，所以可能会消耗点时间。

因为/dev/sdb4 上空闲了 100 个 PE，所以从/dev/sdb2 上移动 100 个 PE 到/dev/sdb4 上。

```
[root@server2 ~]# pvmove /dev/sdb2:0-99 /dev/sdb4
```

这表示将/dev/sdb2 上 0-99 编号的 PE 共 100 个移动到/dev/sdb4 上。如果不加上[-99]这部分，则表示只移动 0 编号这一个 PE。当然，在目标位置/dev/sdb4 上也可以以同样的方式指定移动到的目标 PE 位置上。

再移动/dev/sdb2 上剩余的 PE 到/dev/sdb3 上，但此时应该先看看/dev/sdb2 上仍被占用的 PE 编号是哪些。

```
[root@server2 ~]# pvdisplay -m /dev/sdb2
  --- Physical volume ---
  PV Name               /dev/sdb2
  VG Name               firstvg
  PV Size               3.00 GiB / not usable 16.00 MiB
  Allocatable           yes
  PE Size               16.00 MiB
  Total PE              191
  Free PE               100
  Allocated PE          91
  PV UUID               uVgv3q-ANyy-02M1-wmGf-zmFR-Y16y-qLgNMV

  --- Physical Segments ---
  Physical extent 0 to 99:
    FREE
  Physical extent 100 to 100:
    Logical volume      /dev/firstvg/first_lv
    Logical extents     778 to 778
  Physical extent 101 to 190:
    Logical volume      /dev/firstvg/first_lv
    Logical extents     551 to 640
```

说明从 100-190 都是被占用的，要移动到/dev/sdb3 上的就是这些 PE。

```
[root@server2 ~]# pvmove /dev/sdb2:100-190 /dev/sdb3
```

现在/dev/sdb2 已经完全空闲了，也就是可以从 VG 中移除，然后卸载出来。

```
[root@server2 ~]# pvdisplay  /dev/sdb2
  --- Physical volume ---
  PV Name               /dev/sdb2
  VG Name               firstvg
  PV Size               3.00 GiB / not usable 16.00 MiB
  Allocatable           yes
  PE Size               16.00 MiB
  Total PE              191
  Free PE               191
  Allocated PE          0
  PV UUID               uVgv3q-ANyy-02M1-wmGf-zmFR-Y16y-qLgNMV
```

(4). 从 vg 中移除 pv

```
[root@server2 ~]# vgreduce firstvg /dev/sdb2
```

(5). 移除该 pv

```
[root@server2 ~]# pvremove /dev/sdb2
```

现在/dev/sdb2 就完全被移除了。

6.7 逻辑卷的快照功能

LVM 逻辑卷管理器还具备有"快照卷"的功能，这项功能很类其他软件的还原时间点功能。例如可以对某一个 LV 逻辑卷设备做一次快照，如果今后发现数据被改错了，可以将之前做好的快照卷进行覆盖还原。

LVM 逻辑卷管理器的快照功能有两项特点：一是快照卷的大小应该尽量等同于 LV 逻辑卷的容量；二是快照功能仅一次有效，一旦被还原后则会被自动立即删除。

首先应当查看下卷组的信息：

```
vgdisplay
```

```
[root@server2 ~]# vgdisplay
  --- Volume group ---
  VG Name               firstvg
  System ID
```

```
  Format                lvm2
  Metadata Areas        4
  Metadata Sequence No  31
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                2
  Open LV               0
  Max PV                0
  Cur PV                4
  Act PV                4
  VG Size               14.67 GiB
  PE Size               16.00 MiB
  Total PE              939
  Alloc PE / Size       939 / 14.67 GiB
  Free  PE / Size       0 / 0
  VG UUID               GLwZTC-zUj9-mKas-CJ5m-Xf91-5Vqu-oEiJGj
```

通过卷组的输出信息可以很清晰的看到卷组中已用 120M，空闲资源有 39.88G，接下来在逻辑卷设备所挂载的目录中用重定向写入一个文件：

```
echo "Welcome to Linux " > /linux/readme.txt
```

(1). 第 1 步：使用-s 参数来生成一个快照卷，使用-L 参数来指定切割的大小，另外要记得在后面写上这个快照是针对哪个逻辑卷做的。

```
lvcreate -L 120M -s -n SNAP /dev/storage/vo
```

(2). 第 2 步：在 LV 设备卷所挂载的目录中创建一个 100M 的垃圾文件，这样再来看快照卷的状态就会发现使用率上升了：

```
dd if=/dev/zero of=/linuxprobe/files count=1 bs=100M
lvdisplay
```

(3). 第 3 步：为了校验 SNAP 快照卷的效果，需要对逻辑卷进行快照合并还原操作，在这之前记得先卸载掉逻辑卷设备与目录的挂载

```
umount /linux
lvconvert --merge /dev/storage/SNAP
```

(4). 第 4 步：快照卷会被自动删除掉，并且刚刚在逻辑卷设备被快照后再创建出来的 100M 垃圾文件也被清除了：

```
mount xxxx
ls /linux
```

第7章 RAID

7.1 RAID 概念

RAID 独立磁盘冗余阵列(Redundant Array of Independent Disks)，RAID 技术是将许多块硬盘设备组合成一个容量更大、更安全的硬盘组，可以将数据切割成多个区段后分别存放在各个不同物理硬盘设备上，然后利用分散读写需求来提升硬盘组整体的性能，同时将重要数据同步保存多份到不同的物理硬盘设备上，可以有非常好的数据备份效果。

由于对成本和技术两方面的考虑，因此需要针对不同的需求在数据可靠性及读写性能上做权衡，制定出各自不同的合适方案，目前已有的 RAID 硬盘组的方案至少有十几种，RAID0、RAID1、RAID5、RAID10 和 RAID01 是五种最常见的方案。

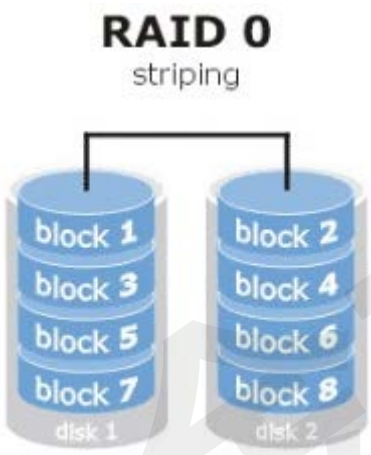
不得不说的是，raid 了解越深入，越能体会到选择和平衡的思想。

关于详细的 raid 技术和原理实现方面，查看 man md，该文档中给出了非常详细的实现方式，包括数据是如何组织的。

7.1.1 RAID 0

首先是 RAID0 硬盘组，这项技术是将多块物理硬盘设备通过硬件或软件的方式串联在一起，成为一个大的卷组，有时它称为条带卷(striping)。它将数据依次分别写入到各个物理硬盘中，这样最理想的状态会使得读写性能提升数倍，但若任意一块硬盘故障则会让整个系统的数据都受到破坏。

通俗来说 RAID0 硬盘组技术至少需要两块物理硬盘设备，能够有效的提高硬盘的性能和吞吐量，但没有数据的冗余和错误修复能力。如图中所示，数据被分别写入到不同的硬盘设备中。

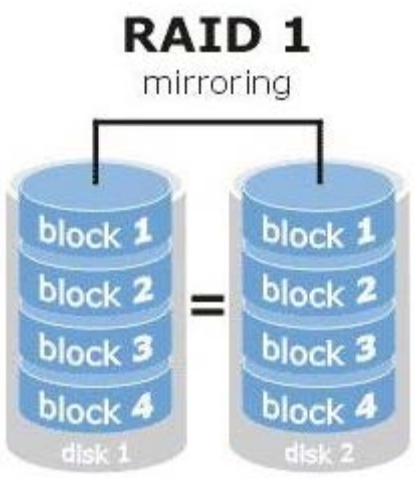


RAID0硬盘组技术示例图

7.1.2 RAID 1

RAID0 技术虽然提高了存储设备的 IO 读写速度，但 RAID0 中数据是被分开存放的，也就是说其中任何一块硬盘出现问题都会破坏数据完整性。因此追求数据安全性的时候就不应该使用 RAID0，而是使用 RAID1。

如图所示，RAID1 硬盘组技术是将两块以上的存储设备进行绑定，目的是让数据被多块硬盘同时写入，类似于把数据再制作出多份镜像，当有某一块硬盘损坏后,一般可以立即通过热交换方式来恢复数据的正常使用。



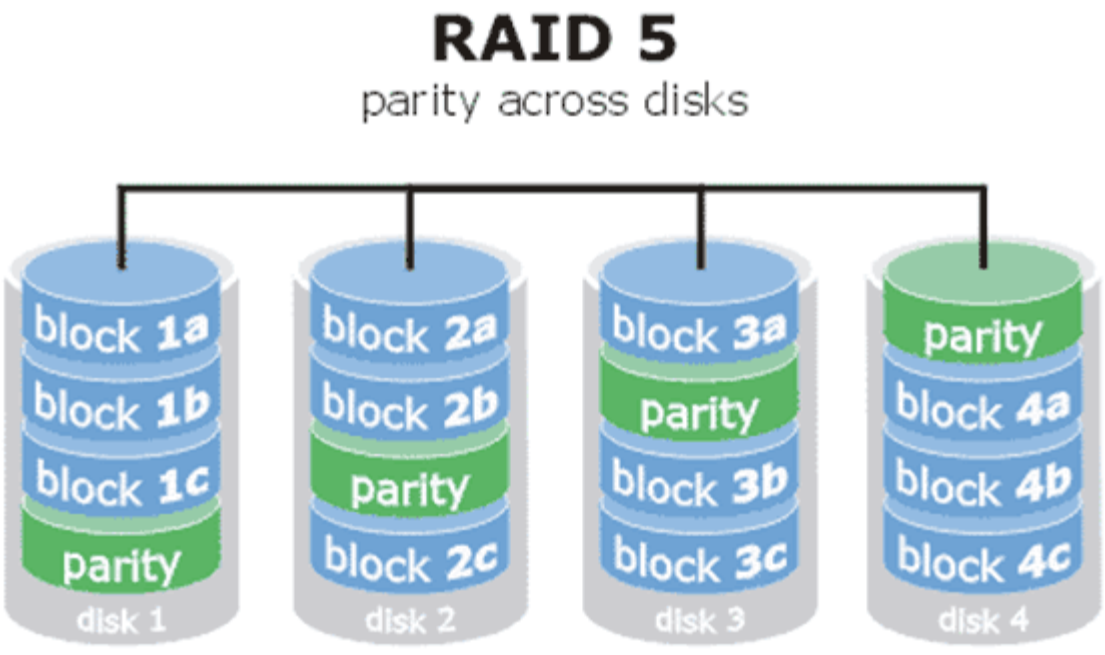
RAID1硬盘组技术示例图

RAID1 注重了数据的安全性，但因为是将多块硬盘中写入相同的数据，也就是说硬盘空间的真实可用率在理论上只有 50%(利用率是 1/n，n 是阵列中的磁盘数量，不分奇偶)，因此会明显的提高硬盘组成本。另外，因为需要将数据同时写入到两块以上的硬盘设备中，这无疑也会增加一定系统负载。

但要注意，raid1 因为同一份数据保存了多份，所以读性能和 RAID0 是一样的(粗略的说是一样。更细致的要分随机读、顺序读，这时不一定和 raid0 一样)。

7.1.3 RAID 5

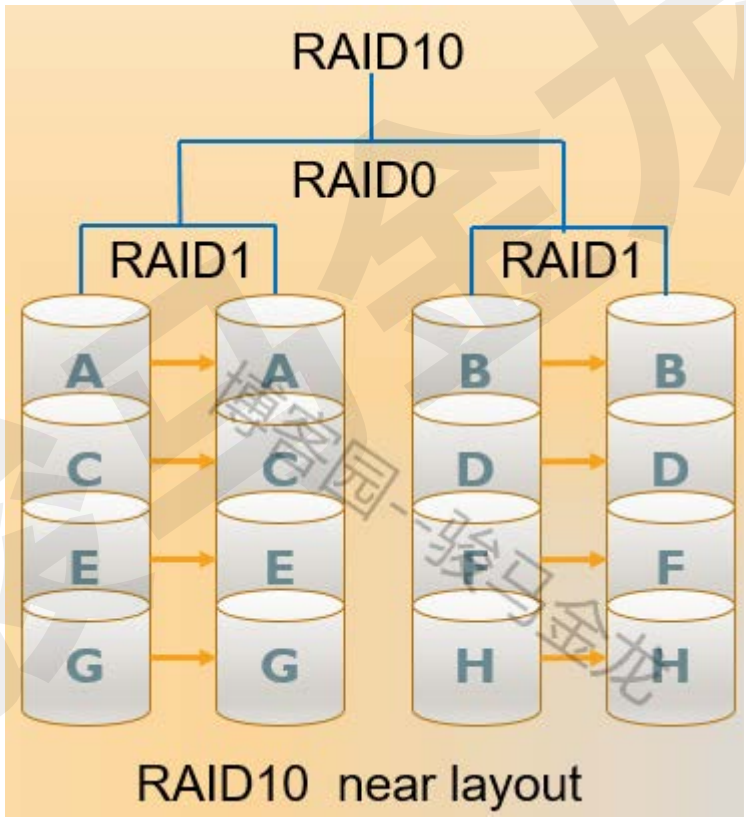
实际上单从数据安全和成本问题上来讲，就不可能在保持存储可用率的同时还不增加新设备的情况下大幅提升数据的安全性，RAID5 硬盘组技术虽然理论上是兼顾三者的，但实际上更像是一种对各个方面的"互相妥协"。



如图所示，RAID5 是将其它存储设备中的数据奇偶校验信息互相保存到硬盘设备中。RAID5 的特点是：第一，数据的奇偶校验信息并不单独保存到某一块硬盘设备中，而是分别存储到每一块硬盘上，这样的好处就是当其中任何一设备损坏后不至于出现致命缺陷；第二，图中 parity 部分表示的就是奇偶校验信息。换句话说就是 RAID5 并不是简单备份硬盘实际数据，而是当设备出现问题后通过奇偶校验信息来计算并尝试重建损坏的数据。这样的技术特性"妥协"的兼顾了存储设备性能、数据安全性与存储成本问题。

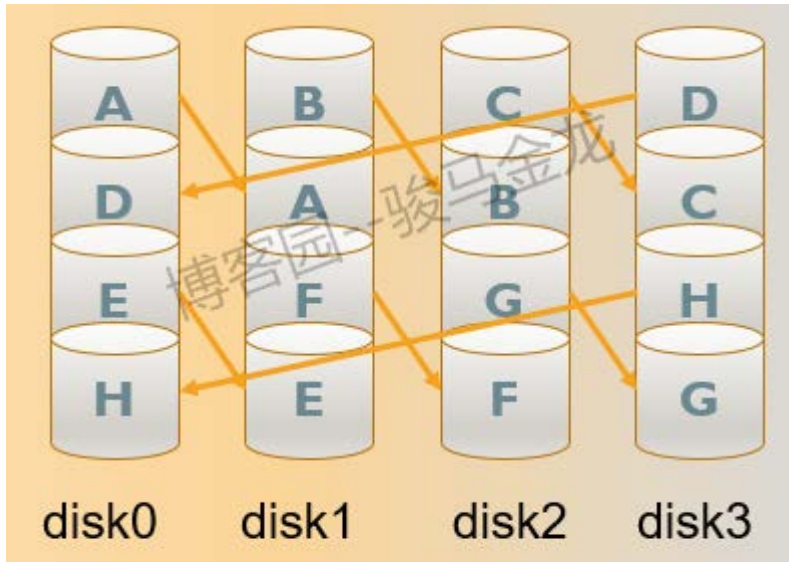
7.1.4 RAID 10

RAID5 在成本问题和读写速度以及安全性能上进行了妥协，但绝大部分情况下，相比硬盘的价格，数据的价值才是更重要的.因此更多的是使用 RAID10，就是对 RAID1+RAID0 的一个"组合体"。

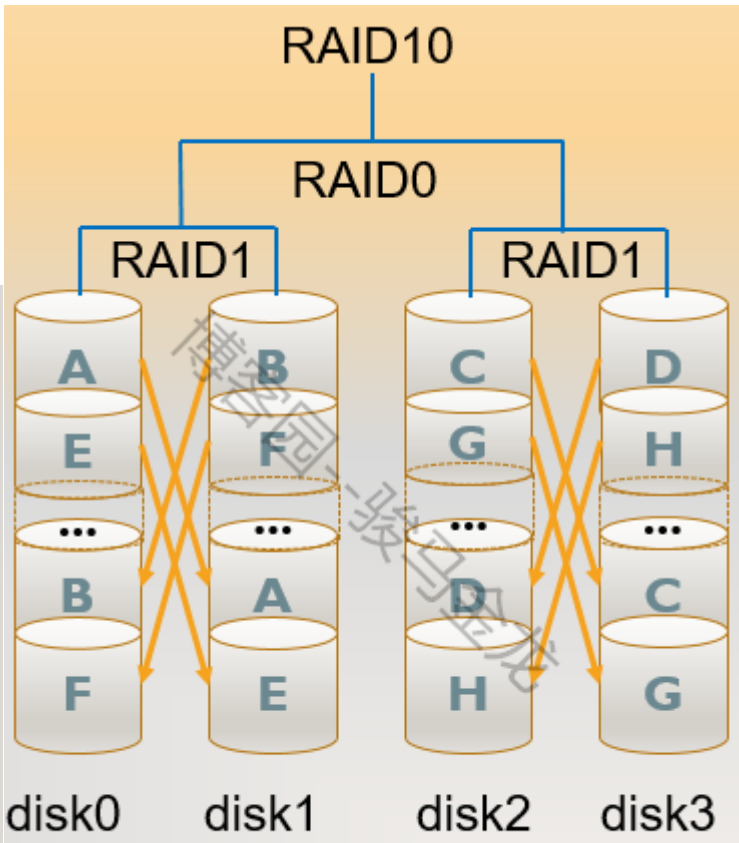


如图所示，RAID10 需要至少 4 块硬盘，先分别两两制作成 RAID1，保证数据的安全性，然后再对两个 RAID1 实施 RAID0 技术，进一步的提高存储设备的读写速度，这样理论上只要坏的不是同一组中的所有硬盘，那么最多可以损坏 50%的硬盘设备而不丢失数据，因此 RAID10 硬盘组技术继承了 RAID0 更高的读写速度和 RAID1 更安全的数据保障，在不考虑成本的情况下 RAID10 在读写速度和数据保障性方面都超过了 RAID5，是较为广泛使用的存储技术。

注意，上图中一个数据块是相邻存储在相同偏移的，即 A 和 A 在相邻设备的同一高度，这只是 RAID10 的一种复制方法，称为 near 复制方法，也是默认复制方法。此外，还有 far、offset 两种复制方法。



RAID10 offset layout

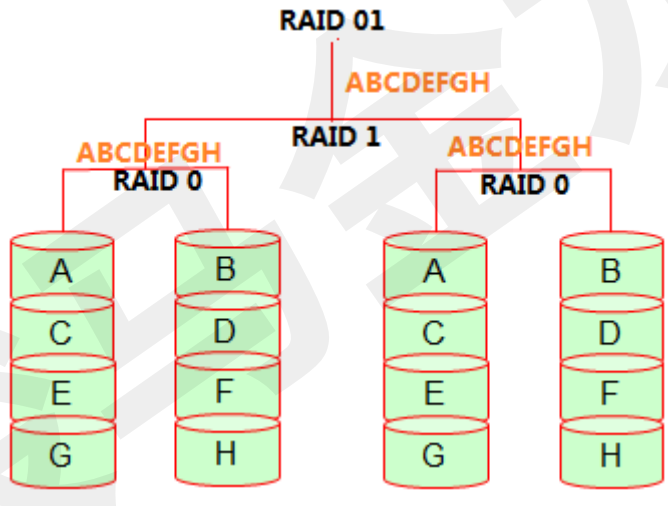


RAID10 far layout

7.1.5 RAID 01

RAID 10 是 RAID1+RAID0，而 RAID01 则是 RAID0+RAID1。看起来它们是相同的，但实际上它们的差别在安全性上差很大。

如下图。从图中可以看出，在 RAID01 正常工作的时候，它的性能和 RAID10 是一样的，都是条带一份，镜像一份。但是区别就在于安全性上。



如果 RAID0 组中的一块磁盘坏了，对于这一个 RAID0 组来说，它已经失效了，并不是还可以从该组中的另一块磁盘中读取一半数据。

也就是说，RAID01 只要坏了一块盘后，该 RAID0 组就失效，IO 的压力就只在另一个 RAID0 组上，这很容易导致这一个 raid0 组也损坏磁盘，只要这时再坏一块盘，所有数据就丢失了。

所以 RAID01 基本无人使用，太不安全。

7.1.6 RAID 的冗余和性能计算

以下图片截自 wiki：https://en.wikipedia.org/wiki/Standard_RAID_levels

Level	Minimum number of drives[b]	Space efficiency	Fault tolerance	Read performance	Write performance
				as factor of single disk	
RAID 0	2	1	None	n	n
RAID 1	2	$\frac{1}{n}$	$n-1$ drive failures	n[a][15]	1[c][15]
RAID 2	3	$1-\frac{1}{n}\log_2(n-1)$	One drive failure[d]	Depends	Depends
RAID 3	3	$1-\frac{1}{n}$	One drive failure	$(n-1)$	$(n-1)$[e]
RAID 4	3	$1-\frac{1}{n}$	One drive failure	$1-(1-r)^n-nr(1-r)^{n-1}$[*citation needed*]	$(n-1)$[e][*citation needed*]
RAID 5	3	$1-\frac{1}{n}$	One drive failure	n[e]	single sector: $\frac{1}{4}$ full stripe: $(n-1)$[e][*citation needed*]
RAID 6	4	$1-\frac{2}{n}$	Two drive failures	n[e]	single sector: $\frac{1}{6}$ full stripe: $(n-2)$[e][*citation needed*]

7.1.7 关于 raid 的理想值和实际应用值

上面介绍的，已经很多书上、老师讲解的都是 raid 的理论情况，比如 raid10 的持续写速度是"单盘速度*盘数量除 2"，raid5 能实现盘数量-1 的最大理论速度。但是这些都是"去他妈"的理论值，RAID 的实际使用情况和理论之间有巨大的差别。所以，很有必要了解一下实际应用中那些理论之外的内容。

以下是我找到的一篇很不错的介绍 raid 的理论外文章：https://zhuanlan.zhihu.com/p/31944934

7.2 Linux 上软 raid 管理

7.2.1 实现 RAID10

首先需要为虚拟机中添加 4 块硬盘设备(若以分区加入，则分区标识符为 raid)来制作一个 RAID10。

mdadm 工具用于在 Linux 系统中创建和管理软 RAID，命令的格式为：

```
mdadm [模式] <RAID 设备名称> [选项] [成员设备名称]
选项说明：
【创建模式】
-C：创建(create a new array)
-l：指定 raid 级别(Set RAID level,0,1,5,10)
-c：指定 chunk 大小(Specify chunk size of kibibytes，default 512KB)
-a：检测设备名称(—auto=yes)，yes 表示自动创建设备文件/dev/mdN
-n：指定设备数量(—raid-devices:Specify the number of active devices in the array)
-x：指定备用设备数量(—spare-devices:Specify the number of spare (eXtra) devices in the initial array)
-v：显示过程
-f：强制行为
-r       ：移除设备(remove listed devices)
-S       ：停止阵列(--stop:deactivate array，releasing all resources)
-A       ：装配阵列，将停止状态的阵列重新启动起来

【监控模式】
-Q：查看摘要信息(query)
-D：查看详细信息(Print details of one or more md devices)
    mdadm -D --scan >/etc/mdadm.conf，以后可以直接 mdadm -A 进行装配这些阵列

【管理模式】
mdadm --manage /dev/md[0-9] [--add 设备名] [--remove 设备名] [--fail 设备名]
--manage ：mdadm 使用 manage 模式，此模式下可以做--add/--remove/—fail/--replace 动作
-add     ：将后面列出的设备加入到这个 md
--remove ：将后面列出的设备从 md 中移除，相当于硬件 raid 的拔出动作
--fail   ：将后面列出的设备设定为错误状态，即人为损坏，损坏后该设备放在 raid 中已经是无意义状态的
```

- **第 1 步：准备磁盘或分区，以分区为例。**

/dev/sd{b,c,d,e}1 这四个分区都是 200M 大小。

- **第 2 步:使用 mdadm 命令创建 RAID10,名称为"/dev/md0"。**

用-C 参数代表创建一个 RAID 阵列卡，-v 参数来显示出创建的过程，同时在后面追加一个设备名称，-a yes 参数代表自动创建设备文件，-n 4 参数代表使用 4 块硬盘(分区)来制作这个 RAID 组，而-l 10 参数则代表 RAID10，最后再加上 4 块设备的名称就可以了。

```
[root@xuexi ~]# mdadm -C /dev/md0 -n 4 -l 10 -a yes /dev/sd{b,c,d,e}1
mdadm: /dev/sdb1 appears to contain an ext2fs file system
       size=10485760K  mtime=Thu Jan  1 08:00:00 1970
Continue creating array? y
mdadm: Defaulting to version 1.2 metadata
mdadm: array /dev/md0 started.
```

查看 RAID 设备信息。

```
[root@xuexi ~]# mdadm -D /dev/md0
/dev/md0:
        Version : 1.2
  Creation Time : Fri Jun  9 21:44:59 2017
     Raid Level : raid10
     Array Size : 387072 (378.06 MiB 396.36 MB)  # 总共能使用的空间，因为是 raid10，所以总可用空间为 400M 左右，除去元数据，大于 370M 左右
  Used Dev Size : 193536 (189.03 MiB 198.18 MB)  # 每颗 raid 组或设备上的可用空间，也即每个 RAID1 组可用大小为 190M 左右
   Raid Devices : 4     # raid 中设备的个数
  Total Devices : 4     # 总设备个数，包括 raid 中设备个数，备用设备个数等
    Persistence : Superblock is persistent

    Update Time : Fri Jun  9 21:45:02 2017
          State : clean    # 当前 raid 状态，有 clean/degraded(降级)/recovering/resyncing
 Active Devices : 4
Working Devices : 4
 Failed Devices : 0
  Spare Devices : 0

         Layout : near=2  # RAID10 数据分布方式，有 near/far/offset，默认为 near，即数据的副本存储在相邻设备的相同偏移上
     Chunk Size : 512K

           Name : xuexi.longshuai.com:0  (local to host xuexi.longshuai.com)
           UUID : ff2b7d7c:381a4c47:c31e7edd:7cdef01e
         Events : 17

    Number   Major   Minor   RaidDevice State
       0       8       17        0      active sync set-A   /dev/sdb1      # /dev/sdb1 是第一个 raid1 组 A 成员
       1       8       33        1      active sync set-B   /dev/sdc1      # /dev/sdc1 是第一个 raid1 组 B 成员
       2       8       49        2      active sync set-A   /dev/sdd1      # /dev/sdd1 是第二个 raid1 组 A 成员
       3       8       65        3      active sync set-B   /dev/sde1      # /dev/sde1 是第二个 raid1 组 B 成员
```

raid 创建好后，它的运行状态信息放在/proc/mdstat 中。

```
[root@xuexi ~]# cat /proc/mdstat
Personalities : [raid10]
md0 : active raid10 sde1[3] sdd1[2] sdc1[1] sdb1[0]
      387072 blocks super 1.2 512K chunks 2 near-copies [4/4] [UUUU]

unused devices: <none>
```

其中"md0 : active raid10 sde1[3] sdd1[2] sdc1[1] sdb1[0]"表示 md0 是 raid10 级别的 raid，且是激活状态，sdX[N]表示该设备在 raid 组中的位置是 N，如果有备用设备，则表示方式为 sdX[N][S]，S 就表示 spare 的意思。

其中"387072 blocks super 1.2 512K chunks 2 near-copies [4/4] [UUUU]"表示该 raid 可用总空间为 387072 个 block，每个 block 为 1K，所以为 378M，chunks 的大小 512K，[m/n]表示此 raid10 阵列需要 m 个设备，且 n 个设备正在正常运行，[UUUU]表示分别表示 m 个的每一个运行状态，这里表示这 4 个设备都是正常工作的，如果是不正常的，则以"_"显示。

再看看 lsblk 的结果。

```
[root@xuexi ~]# lsblk -f
NAME      FSTYPE           LABEL                UUID                                   MOUNTPOINT
loop0     iso9660          CentOS_6.6_Final                                            /mnt
sda
├─sda1    ext4                                  77b5f0da-b0f9-4054-9902-c6cdacf29f5e   /boot
├─sda2    ext4                                  f199fcb4-fb06-4bf5-a1b7-a15af0f7cb47   /
└─sda3    swap                                  6ae3975c-1a2a-46e3-87f3-d5bd3f1eff48   [SWAP]
```

```
sr0
sdb
 └─sdb1  linux_raid_member xuexi.longshuai.com:0 ff2b7d7c-381a-4c47-c31e-7edd7cdef01e
   └─md0
sdc
 └─sdc1  linux_raid_member xuexi.longshuai.com:0 ff2b7d7c-381a-4c47-c31e-7edd7cdef01e
   └─md0
sdd
 └─sdd1  linux_raid_member xuexi.longshuai.com:0 ff2b7d7c-381a-4c47-c31e-7edd7cdef01e
   └─md0
sde
 └─sde1  linux_raid_member xuexi.longshuai.com:0 ff2b7d7c-381a-4c47-c31e-7edd7cdef01e
   └─md0
```

- **第 3 步：将制作好的 RAID 组格式化创建文件系统**

以创建 ext4 文件系统为例。

```
[root@xuexi ~]# mke2fs -t ext4 /dev/md0
```

- **第 4 步：挂载 raid 设备，挂载成功后可看到可用空间为 359M，因为 RAID 在创建文件系统时也消耗了一部分空间存储文件系统的元数据**

```
[root@xuexi ~]# mount /dev/md0 /mydata
[root@xuexi ~]# df -hT
Filesystem     Type   Size  Used Avail Use% Mounted on
/dev/sda2      ext4    18G  2.7G   14G  16% /
tmpfs          tmpfs  491M     0  491M   0% /dev/shm
/dev/sda1      ext4   239M   28M  199M  13% /boot
/dev/md0       ext4   359M  2.1M  338M   1% /mydata
```

7.2.2 损坏磁盘阵列及修复

通过 manage 模式可以模拟阵列中的设备损坏。

```
mdadm --manage /dev/md[0-9] [--add 设备名] [--remove 设备名] [--fail 设备名]
选项说明：
manage   ：mdadm 使用 manage 模式，此模式下可以做--add/--remove/—fail/--replace 动作
--add    ：将后面列出的设备加入到这个 md
--remove ：将后面列出的设备从 md 中移除
--fail   ：将后面列出的设备设定为错误状态，即人为损坏
```

模拟/dev/sdc1 损坏。

```
[root@xuexi mydata]# mdadm --manage /dev/md0 --fail /dev/sdc1
mdadm: set /dev/sdc1 faulty in /dev/md0
```

再查看 raid 状态。

```
[root@xuexi mydata]# mdadm -D /dev/md0
/dev/md0:
        Version : 1.2
  Creation Time : Fri Jun  9 21:44:59 2017
     Raid Level : raid10
     Array Size : 387072 (378.06 MiB 396.36 MB)
  Used Dev Size : 193536 (189.03 MiB 198.18 MB)
   Raid Devices : 4
  Total Devices : 4
    Persistence : Superblock is persistent

    Update Time : Fri Jun  9 22:22:29 2017
          State : clean, degraded
 Active Devices : 3
Working Devices : 3
 Failed Devices : 1
  Spare Devices : 0

         Layout : near=2
     Chunk Size : 512K

           Name : xuexi.longshuai.com:0  (local to host xuexi.longshuai.com)
```

```
          UUID : ff2b7d7c:381a4c47:c31e7edd:7cdef01e
        Events : 19

    Number   Major   Minor   RaidDevice State
       0       8       17        0      active sync set-A   /dev/sdb1
       2       0        0        2      removed
       2       8       49        2      active sync set-A   /dev/sdd1
       3       8       65        3      active sync set-B   /dev/sde1

       1       8       33        -      faulty   /dev/sdc1
```

由于 4 块磁盘组成的 raid10 允许损坏一块盘，且还允许坏第二块非对称盘。所以这里损坏了一块盘后 raid10 是可以正常工作的。

现在可以将损坏的磁盘拔出，然后向 raid 中加入新的磁盘即可。

```
mdadm --manage /dev/md0 --remove /dev/sdc1
```

再修复时，可以将新磁盘加入到 raid 中。

```
mdadm --manage /dev/mn0 --add /dev/sdc1
```

7.2.3 raid 备份盘

使用 mdadm 的"-x"选项可以指定备份盘的数量，备份盘的作用是自动顶替 raid 组中坏掉的盘。

7.2.4 停止和装配 raid

```
umount /dev/md0
mdadm --stop /dev/md0
```

关闭 raid 阵列后，该 raid 组/dev/md0 就停止工作了。

如果下次想继续启动它，直接使用-A 来装配/dev/md0 是不可以的，需要再次指定该 raid 中的设备成员，且和关闭前的成员一样，不能有任何不同。

```
mdadm -A /dev/md0 /dev/sd{b,c,d,e}1
```

这样做不太保险，其实可以在停止 raid 前，扫描 raid，将扫描的结果保存到配置文件中，下次启动的时候直接读取配置文件即可。

```
mdadm -D --scan >> /etc/mdadm.conf   # 这是默认配置文件
```

下次直接使用-A 就可以装置配置文件中的 raid 组。

```
mdadm -A /dev/md0
```

如果不放在默认配置文件中，则装配的时候使用"-c"或"--config"选项指定配置文件即可。

```
mdadm -D --scan >> /tmp/mdadm.conf
mdadm -A /dev/md0 -c /tmp/mdadm.conf
```

7.2.5 彻底移除 raid 设备

当已经确定一个磁盘不需要再做为 raid 的一部分，可以将它移除掉。彻底移除一个 raid 设备并非那么简单，因为 raid 控制器已经标记了一个设备，即使将它"mdadm --remove"也无法彻底消除 raid 的信息。

以移除/dev/md127 中的/dev/sdb1 为例。

首先，卸载、停止、移除：

```
umount /dev/sdb1
mdadm --stop /dev/md127
mdadm --manage /dev/md127 --remove /dev/sdb1
```

虽然从 raid 中移除了，但是江湖上还有它的传说：删除分区、创建分区、格式化，格式化的时候将被保护

```
$ parted /dev/sdb rm 1
$ parted /dev/sdb mkpart p 1 20G
$ mke2fs -t ext4 /dev/sdb1
/dev/sdb1 is apparently in use by the system; will not make a filesystem here!
```

然后再去扫描 raid 设备，发现它又出现在 raid 组中：

```
$ mdadm -D -s
ARRAY /dev/md/xuexi.longshuai.com:0 metadata=1.2 ............
$ lsblk -f
NAME      FSTYPE    LABEL    UUID                                 MOUNTPOINT
sda
 ├─sda1    xfs             367d6a77-033b-4037-bbcb-416705ead095 /boot
```

```
├─sda2    xfs             b2a70faf-aea4-4d8e-8be8-c7109ac9c8b8 /
└─sda3    swap            d505113c-daa6-4c17-8b03-b3551ced2305 [SWAP]
sdb
└─sdb1    linux_raid_member xue........
  └─md127 ext4            2fed1dcc-b9a2-477f-8c8f-7131bbd4e919
```

换句话说，只要这个设备曾经是 raid 的一份子，你就没法再直接使用它。就算你分区了，也不让你格式化。

所以，要彻底移除一个 raid 设备，需要清空控制器可以读取的 raid 签名，只需将这个 raid 设备(可能是一个分区)的 raid superblock 用 0 去覆盖掉就行了：

```
$ umount /dev/sdb1
$ mdadm --stop /dev/md127
$ mdadm --manage /dev/md127 --remove /dev/sdb1
$ mdadm --zero-superblock --force /dev/sdb1    # 这条命令是关键
```

然后，这个设备就和 raid 控制器完全 say goodbye 了：

```
$ lsblk -f
NAME   FSTYPE LABEL UUID     MOUNTPOINT
sda
...........
sdb
└─sdb1
```

现在格式化也可以正常进行了：

```
mke2fs -t ext4 /dev/sdb1
```

第8章 包管理

8.1 Linux 上构建 C 程序的过程

在说明包相关的内容之前，我觉得有必要说一下在 Linux 上构建一个 C 程序的过程。我个人并没有学习过 C，内容总结自网上，所以可能显得很小白，而且也并非一定正确，只希望对和我一样菜鸟的同学有所帮助。

(1).拿到源程序。C 的源程序包中包括一堆的 c 文件和 h 文件。

(2).编译。使用编译器(如 gcc)将源程序文件(c 和 h 文件)编译成为目标文件 o 文件。

在编译过程中，使用 makefile 文件中的配置选项进行编译。makefile 文件可以使用 configure 工具生成，有了 makefile 文件，就可以使用 make 命令根据 makefile 文件进行编译，再使用 make install 安装。

但是有些源程序中并没有提供 configure 或 config 这样的文件，这时很可能应该直接使用 make(或其他工具)带上相关编译参数进行编译，参数如何给定以及给定参数的规则，见下文。

(3).链接。在 Linux 上使用 ld 工具，将 o 文件和所需的库文件链接起来组成一个可正常工作的可执行程序。链接了库文件之后就可以发起一些系统调用。

在 make 编译时，可能会需要提供头文件和库文件的路径，如果不提供，则搜索默认路径。当搜索的路径下都找不到所需文件时，将报错提示库文件(lib.xxxx)或头文件(*.h)不存在。一般来说，缺少某头文件，只需安装其对应的 devel 包即可，缺少库文件则可能需要安装主包，也可能需要 devel 包，还可能是需要单独的 libs 包。

其中 gcc 编译器的头文件默认搜索路径为：(使用 cpp -v 命令可以查看)

```
/usr/local/include
/usr/lib/gcc/x86_64-redhat-linux/4.4.7/include  # 此项在不同系统上，具体路径不一样
/usr/include
```

而库文件的默认搜索路径及顺序为：

(1).编译目标代码时指定的动态库搜索路径；

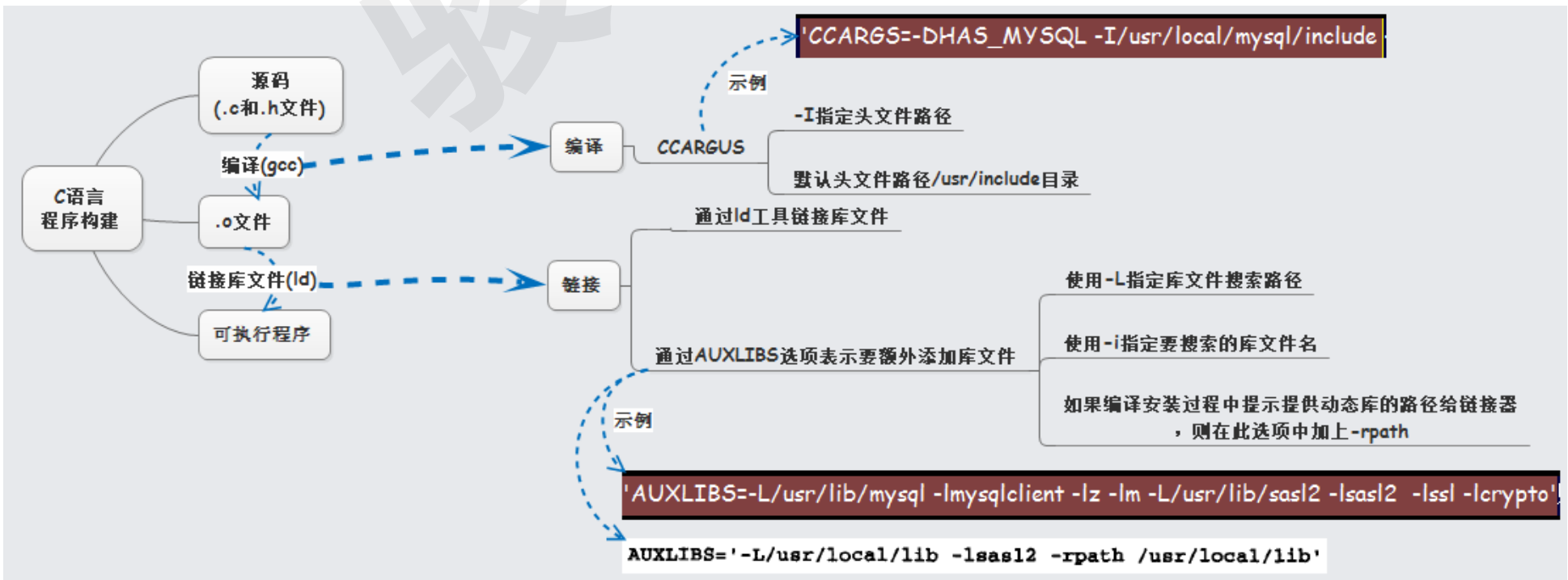
(2).环境变量 LD_LIBRARY_PATH 指定的动态库搜索路径；

(3).配置文件/etc/ld.so.conf 中指定的动态库搜索路径；

(4).默认的动态库搜索路径/lib；

(5).默认的动态库搜索路径/usr/lib

在 make 编译时，如果要指定参数，参数的给定方式一般遵循一些规则。如下图。



(1).使用 CCARGS 编译选项表示提供头文件路径，在该选项的参数中，-I 表示搜索路径，-DHAS_MYSQL 表示使用定义的宏 HAS_MYSQL。

(2).使用 AUXLIBS 链接选项表示提供库文件搜索路径和库文件名，其中-L 指定搜索路径，-l 指定库文件名。有时候提示提供动态链接库路径给链接器，则可能需要使用-rpath 指定一个搜索路径给链接器。

关于更多可能用到的编译选项，可以参考每个源程序包中的 README 或 INSTALL 文件。

在指定库名的时候，可能和想象中的指定方法不一样，它不是指定库的全名，而是指定库名。关于库文件的全名，它遵循一定规律：

(1).前面必须是 lib，中间是库名，后面是后缀，例如 libutil.so，其库名为 util。

(2).库文件的后缀有几种：".a"为静态库，".so"或者".sl"为动态库文件。

(3).很多库的".so"后缀后面还会带上数字，如 libutil.so.1。这些数字和库版本有关，带有数字版本的库文件是不带数字库文件的链接，例如 libutil.so.1 它会指向 libutil.so。这样就可以实现多版本共存，如果有多个库版本，只要找到 libutil.so 就可以找到最新版本的库文件。当然，如果想要使用特定版本的库，只需修改下它的软链接指向即可。

例如上面的图中，要链接/usr/local/mysql 中的 libmysqlclient.a，则指定方式为：AUXLIBS=-L/usr/lib/mysql -lmysqlclient。

而'AUXLIBS=-L/usr/lib/mysql -lmysqlclient -lz -lm -L/usr/lib/sasl2 -lsasl2 -lssl -lcrypto'则表示查找/usr/lib/mysql 目录下的 libmysqlclient.xxxx 以及 libz.xxxx 以及 libm.xxxx 的库文件还查找/usr/lib/sasl2 目录下的 libsasl2.xxxx 以及 SSL 的两个库文件 libssl.xxxx 以及 libcrypto.xxxx。

以下是编译安装 postfix 时的一个示例。

```
# make makefiles 'CCARGS=-DHAS_MYSQL -I/usr/include/mysql -DUSE_SASL_AUTH -DUSE_CYRUS_SASL -I/usr/include/sasl -DUSE_TLS ' 'AUXLIBS=-L/usr/lib/mysql -lmysqlclient -lz -lm -L/usr/lib/sasl2 -lsasl2 -lssl -lcrypto'
```

8.2 包基础知识

8.2.1 包名称

在 rhel/centos/fedora 上，包的名称以 rpm 结尾，分为二进制包和源码包。源码包以".src.rpm"结尾，它是未编译过的包，可以自行进行编译或者用其制作自己的二进制 rpm 包，非".src.rpm"结尾的包都是二进制包，都是已经编译完成的，安装 rpm 包的过程实际上就是将包中的文件复制到 Linux 上，有可能还会在复制文件的前后执行一些命令，如创建一个必要的用户，删除非必要文件等。关于 rpm 包的制作，可能是件很烦人的事，但学习它还是非常有必要的，至少也得能制作简单的包吧？官网：

(1)https://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/Packagers_Guide/index.html

(2)https://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/RPM_Guide/index.html

注意区分源码包和源码的概念，源码一般是打包压缩后的文件(如.tar.gz 结尾的文件)。源码包中包含了源码，还包含了一些有助于制作二进制 rpm 的文件。最有力的说明就是源码编译安装的程序都没有服务启动脚本(/etc/init.d/下对应的启动脚本)，而二进制 rpm 包安装的就有，因为二进制 rpm 包都是通过源码包".src.rpm"定制而来，在源码包中提供了必要的文件(如服务启动脚本)，并在安装 rpm 的时候复制到指定路径下。

回归正题，一个 rpm 包的名称分为包全名和包名，包全名如 httpd-2.2.15-39.el6.centos.x86_64.rpm，包全名中各部分的意义如下：

httpd	包名
2.2.15	版本号，版本号格式[主版本号.[次版本号.[修正号]]]
39	软件发布次数
el6.centos	适合的操作系统平台以及适合的操作系统版本
x86_64	适合的硬件平台，硬件平台根据 cpu 来决定，有 i386、i586、i686、x86_64、noarch 或者省略，noarch 或省略表示不区分硬件平台
rpm	软件包后缀扩展名

使用 rpm 工具管理包时，如果要操作未安装的包，则使用包全名，如安装包，查看未安装包的信息等；如果要操作已安装的 rpm 包，则只需要给定其包名即可，如查询已装包生成了哪些文件，查看已装包的信息等。

而对于 yum 工具来说，只需给定其包名即可，若有需要，再指定版本号，如明确指明要安装 1.6.10 版本的 tree 工具，yum install tree-1.6.10。

8.2.2 主包和子包

对于一个程序而言，在制作 rpm 包时，很多时候都将其按功能分割成多个子包，如客户端程序包、服务端程序包等。

以 mysql 这个程序来说，它分有以下几个包。

```
mysql-server.x86_64
mysql.x86_64
mysql-bench.x86_64
mysql-libs.x86_64
mysql-devel.x86_64
```

其中 mysql-server.x86_64 是提供服务的主包，mysql.x86_64 是客户端主包，mysql-bench 是用于对 MySQL 进行压力测试的包，mysql-libs 和 mysql-devel 分别是库文件包和头文件包。后两者是提供给其他需要联合 mysql 的程序使用的，仅就实现 mysql 服务而言，只需安装 mysql-server 即可。

而源码编译安装的包会包含所有功能包，也就是说编译安装一个程序后，它的客户端工具、服务提供程序、库文件、头文件等等都已经安装了。

8.3 rpm 管理包

rpm 包被安装后，会在/var/lib/rpm 下会建立已装 rpm 数据库，以后有任何 rpm 的升级、查询、版本比较等包的操作都是从这个目录下获取信息并完成相应操作的。

```
[root@xuexi ~]# ls /var/lib/rpm/
Basenames     __db.003     Group         Packages        Requirename     Triggername
Conflictname  __db.004     Installtid    Providename     Requireversion
__db.001      Dirnames     Name          Provideversion  Sha1header
```

```
__db.002      Filedigests  Obsoletename  Pubkeys         Sigmd5
```

8.3.1 安装包后的文件分布

rpm 安装完成后，相关的文件会复制到多个目录下(具体复制的路径是在制作 rpm 包时指定的)。一般来说，分布形式差不多如下表。

/etc	放置配置文件的目录
/bin、/sbin、/usr/bin 或/usr/sbin	一些可执行文件
/lib、/lib64、/usr/lib(/usr/lib64)	一些库文件
/usr/include	一些头文件
/usr/share/doc	一些基本的软件使用手册与帮助文件
/usr/share/man	一些 man page 档案

8.3.2 rpm 安装、升级、卸载(install、update、uninstall)

rpm 工具安装、升级和卸载的功能都很少使用。对于安装来说，它需要人为解决包的依赖关系，这是极其令人恶心的事对于升级来说，基本上都会使用 yum 工具进行安装和升级，而卸载行为在 Linux 上很少出现，大不了直接覆盖重装。

```
rpm -ivhUe --nodeps --test --force --prefix
选项说明：
-i 表示安装，install 的意思
-v 显示安装信息，还可以"-vv"、"-vvv"，v 提供的越多显示信息越多
-h 显示安装进度，以#显示安装的进度
-U 升级或升级包
-F 只升级已安装的包
-e 卸载包，卸载也有依赖性,"--erase"
--nodeps 忽略依赖性强制安装或卸载(no dependencies)
--test 测试是否能够成功安装指定的 rpm 包
--prefix 新路径 自行指定安装路径而不是使用默认路径，基本上都不支持该功能，功能极其简单的软件估计才支持重定位安装路径
--force 强制动作
--replacepkgs 替换安装，即重新覆盖安装。
```

有时误删文件可以不用卸载再装，直接使用--replacepkgs 选项再次安装即可。

rpm 包另一个缺陷是只能安装本地或给定 url 路径的 rpm 包。

注意：不要对内核进行升级；多版本的内核可以并存，因此可以执行安装操作。

8.3.3 rpm 查询功能

rpm 工具的安装功能很少使用，毕竟解决依赖关系不是件容易的事。但是 rpm 的查询功能则非常实用。

```
-q[p] -q 查询已安装的包，-qp 查询未安装的包。它们都可接下面的参数
    -a 查询所有已安装的包，也可以指定通配符名称进行查询
    -i 查询指定包的信息（版本、开发商、安装时间等）。从这里面可以查看到软件包属于哪个包组。
    -l 查询包的文件列表和目录（包在生产的时候就指定了文件路径，因此可查未装包）
    -R 查询包的依赖性（Required）
    -c 查询安装后包生成的配置文件
    -d 查询安装后包生成的帮助文档
-f 查询系统文件属于哪个已安装的包（接的是文件而不是包）
--scripts 查询包相关的脚本文档。脚本文档分四类：安装前运行、安装后运行、卸载前运行、卸载后运行
```

例如：

(1).查询文件/etc/yum.conf 是通过哪个包安装的。

```
[root@xuexi cdrom]# rpm -qf /etc/yum.conf
yum-3.2.29-60.el6.centos.noarch
```

(2).查询安装 httpd 时生成了哪些目录和文件，还可以过滤出提供了哪些命令行工具。

```
rpm -ql httpd
rpm -ql httpd | grep 'bin/'
```

(3).查询某个未安装包的依赖性如 zip-3.0-1.el6.x86_64.rpm 的依赖性。

```
[root@xuexi cdrom]# rpm -qRp zip-3.0-1.el6.x86_64.rpm
libc.so.6()(64bit)
libc.so.6(GLIBC_2.2.5)(64bit)
libc.so.6(GLIBC_2.3)(64bit)
libc.so.6(GLIBC_2.3.4)(64bit)
libc.so.6(GLIBC_2.4)(64bit)
libc.so.6(GLIBC_2.7)(64bit)
rpmlib(CompressedFileNames) <= 3.0.4-1
```

```
rpmlib(FileDigests) <= 4.6.0-1
rpmlib(PayloadFilesHavePrefix) <= 4.0-1
rtld(GNU_HASH)
rpmlib(PayloadIsXz) <= 5.2-1
```

实际上，查看包的依赖性时，使用 yum-utils 包中的 repoquery 工具更好，"repoquery -R pkg_name"会更简洁。yum-utils 中包含了好几个非常实用的包管理工具，可以大致都了解下。

8.3.4 提取 rpm 包中文件

安装 rpm 包会安装 rpm 中所有文件，如果将某个文件删除了，除了重装 rpm，还可以通过从 rpm 包提取缺失文件的方式来修复。在 win 上安装个万能压缩工具"好压"，可以直接打开 rpm 包，然后从中解压需要的文件出来。但是在 Linux 上，过程还是有点小复杂的，其中涉及了 cpio 这个古来的归档工具。

方法：使用 rpm2cpio 命令组合 cpio -idv 命令的方式来提取。cpio 具体用法参见 http://www.cnblogs.com/f-ck-need-u/p/7008380.html。

rpm2cpio 是将 rpm 转换为 cpio 格式归档文件的命令，有了 cpio 文件，就可以使用 cpio 命令对其进行相关的操作。

cpio 命令是从归档文件中提取文件或向归档文件中写入文件的工具，一般都从标准输入或输出操作归档文件，所以都使用管道或重定向符号。

```
-i:  运行在 copy-in 模式，即从归档文件中将文件 copy 出来，即提取文件（提取）
-o:  运行在 copy-out 模式，将文件 copy 到归档文件中，即将文件拷贝到归档文件中（写入）
-d:  需要目录时自动建立目录
-v:  显示信息
```

提取 rpm 包文件的一般格式为以下格式：

```
rpm2cpio package_full_name|cpio -idv dir_name
```

例如，删除/bin/ls 文件，将导致 ls 命令不可用，使用文件提取的方式去修复。

```
[root@xuexi cdrom]# which ls     # 查找需要删除的 ls 文件位置
alias ls='ls --color=auto'
        /bin/ls
[root@xuexi cdrom]# rpm -qf /bin/ls   # 查找 ls 命令属于哪个包
coreutils-8.4-37.el6.x86_64
[root@xuexi cdrom]# rm -f /bin/ls      # 删除 ls 命令
[root@xuexi cdrom]# ls                   # ls 命令已不可用
-bash: /bin/ls: No such file or directory
[root@xuexi ~]# yumdownloader    coreutils      # 下载所需的 rpm 包
[root@xuexi ~]# rpm2cpio coreutils-8.4-37.el6.x86_64.rpm | cpio -id ./bin/ls   # 提取 bin/ls 到当前目录下
[root@xuexi ~]# dir ~/bin     # 使用 dir 命令查看已经提取成功，dir 命令功能等价于 ls
ls
[root@xuexi tmp]# cp bin/ls /bin/      # 复制 ls 命令到/bin 下
[root@xuexi tmp]# ls                  # 测试，ls 已经可用
```

8.4 yum 管理包

yum 工具通过仓库的方式简化 rpm 包的管理。它从仓库中搜索相关的软件包，并自动下载和解决软件包的依赖性，非常方便。

8.4.1 /etc/yum.conf

/etc/yum.conf 是 yum 的默认文件，里面配置的也是全局默认项。

```
[root@server2 ~]# cat /etc/yum.conf
[main]
cachedir=/var/cache/yum/$basearch/$releasever   # 缓存目录
keepcache=0          # 是否保留缓存，设置为 1 时，安装包时所下载的包将不会被删除
debuglevel=2         # 调试信息的级别
logfile=/var/log/yum.log   # 日志文件位置
exactarch=1         # 设置为 1 将只会安装和系统架构完全匹配的包
obsoletes=1         # 是否允许更新旧的包
gpgcheck=1          # 是否要进行 gpg check
plugins=1           # 是否允许使用 yum 插件
installonly_limit=5
bugtracker_url=http://bugs.centos.org/set_project.php?project_id=23&ref=http://bugs.centos.org/bug_report_page.php?category=yum
distroverpkg=centos-release    # 指定基准包，yum 会根据这个包判断发行版本
```

8.4.2 配置 yum 仓库

首先配置 yum 仓库，配置文件为/etc/yum.conf 和/etc/yum.repos.d/中的".repo"文件，其中/etc/yum.conf 配置的是仓库的默认项，一般配置 yum 源都是在/etc/yum.repos.d/*.repo 中配置。注意，该目录中任意 repo 文件都会被读取。

默认/etc/yum.repos.d/下会有以下几个仓库文件，除了 CentOS-Base.repo，其他的都可以删掉，基本没用。

```
[root@xuexi yum.repos.d]# ls /etc/yum.repos.d/
CentOS-Base.repo       CentOS-fasttrack.repo  CentOS-Vault.repo
CentOS-Debuginfo.repo  CentOS-Media.repo
```

repo 文件的配置格式如下：

```
[root@xuexi yum.repos.d]# vim CentOS-Base.repo
[base]      # 仓库 ID，ID 必须保证唯一性
name        # 仓库名称，可随意命名
mirrorlist  # 该地址下包含了仓库地址列表，包含一个或多个镜像站点，和 baseurl 使用一个就可以了
#baseurl    # 仓库地址。网络上的地址则写网络地址，本地地址则写本地地址，格式为"file://"后接路径，如 file:///mnt/cdrom
gpgcheck=1  # 指定是否需要 gpg 签名，1 表示需要，0 表示不需要
gpgkey  =   # 签名文件的路径
enable      # 该仓库是否生效，enable=1 表示生效，enable=0 表示不生效
cost=       # 开销越高，优先级越低
```

repo 配置文件中可用的宏：

```
$releasever：程序的版本(release version)，对 Yum 而言指的是 redhat-relrase 版本。只替换为主版本号，如 Redhat6.5 则替换为 6
$arch：系统架构
$basharch：系统基本架构，如 i686，i586 等的基本架构为 i386
$YUM0-9：在系统定义的环境变量，可以在 yum 中使用
```

8.4.3 repo 配置示例：配置 epel 仓库

系统发行商在系统中放置的 rpm 包一般版本都较老，可能有些包有较大的延后性。而 epel 是由 fedora 社区维护的高质量高可靠性的安装源，有很多包是比系统包更新的，且多出很多系统没有的包。总之，用到 epel 的机会很多很多，所以就拿来当配置示例了。

有两种方式可以使用 epel 源。

方法一：安装 epel-release-noarch.rpm

```
shell> rpm -ivh epel-release-latest-6.noarch.rpm
```

安装后会在/etc/yum.repo.d/目录下生成两个 epel 相关的 repo 文件，其中一个是 epel.repo。此文件中 epel 的源设置在了 fedora 的镜像站点上，这对国内网来说可能会较慢，可以修改它为下面的内容。

```
[epel]
name=Extra Packages for Enterprise Linux 6 - $basearch
baseurl=http://mirrors.sohu.com/fedora-epel/6Server/$basearch/
#mirrorlist=https://mirrors.fedoraproject.org/metalink?repo=epel-6&arch=$basearch
failovermethod=priority
enabled=1
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6
```

方法二：直接增加 epel 仓库

在/etc/yum.repos.d/下任意一个 repo 文件中添加上 epel 的仓库即可。

```
[epel]
name=epel
baseurl=http://mirrors.sohu.com/fedora-epel/6Server/$basearch/
enabled=1
gpgcheck=0
```

然后清除缓存再建立缓存即可。

```
yum clean all ; yum makecache
```

8.4.4 yum 命令

不同的版本，yum 命令可能功能上有所不同，例如在 CentOS 7 上的 yum 有--downloadonly 的功能，而在 CentOS 6.6 上就没有(更新 yum 包后就有了)。此处介绍几个 yum 命令。

```
Usage: yum [options] COMMAND

List of Commands:
help            命令的帮助信息，用法：yum help command，如 yum help install 则查看 install 命令的用法说明
clean           清除缓存数据，如 yum clean all
makecache       生成元数据缓存数据，yum makecache
deplist         列出包的依赖关系
erase           卸载包
```

```
fs              为当前文件系统创建快照，或者列出或删除当前已有快照。快照是非常有用的，升级或打补丁前拍个快照，就能放心地升级或打补丁了
fssnapshot      同 fs 一样
groups          操作包组
history         查看 yum 事务信息，yum 是独占模式的进程，所以有时候查看事务信息还是有用的
info            输出包或包组的信息，例如该包是谁制作的，大概是干什么用的，来源于哪个包组等信息
install         包安装命令
list            列出包名，一般会结合 grep 来搜索包，如 yum list all | grep -i zabbix
provides        搜索给定的内容是谁提供的，可用来搜索来源于哪个包，如 CentOS 7 上 mysql 被 mariadb 替代，搜索 Mysql 提供者时就能找出包 mariadb
reinstall       重新安装包
repolist        列出可用的仓库列表
search          给定字符串搜索相关包，并给出相关包较为详细的信息
update          更新包

Options:
  -R [minutes], --randomwait=[minutes]：最多等待时间
  -q, --quiet           安静模式
  -v, --verbose         详细模式
  -y, --assumeyes       对所有问题回答 yes
  --assumeno            对所有问题回答 no
  --enablerepo=[repo]   启用一个或多个仓库，可用通配符通配仓库 ID
  --disablerepo=[repo]  禁用一个或多个仓库，可用通配符通配仓库 ID
  -x [package], --exclude=[package]  通配要排除的包
  --nogpgcheck          禁用 gpgcheck
  --color=COLOR         带颜色
  --downloadonly        仅下载包，不安装或升级。默认下载在 yum 的缓存目录中，默认为/var/cache/yum/$basearch/$releasever
  --downloaddir=DLDIR   指定下载目录
```

很多时候，yum 操作失败的原因是 repo 的配置错误，或者缓存未更新。

8.4.5 yum 操作包组

```
[root@server2 ~]# yum help groups
Loaded plugins: fastestmirror, langpacks
groups [list|info|summary|install|upgrade|remove|mark] [GROUP]

Display, or use, the groups information

aliases: group, grouplist, groupinfo, groupinstall, groupupdate, groupremove, grouperase
```

所以，可以使用别名 group， grouplist，groupinfo，groupinstall， groupupdate， groupremove， grouperase 替代相应的命令。

如安装包组 yum groups install pkg_groupname，也可以 yum groupinstall pkg_groupname。

8.5 补丁工具 diff 和 patch

补丁文件都是通过 diff 命令生成的，patch 工具则是应用补丁，即将补丁打到旧文件中。绝对不能跨版本打补丁，只能一个版本一个版本按顺序慢慢打。

生成补丁：

```
diff -[r]Nu old_file new_file > patch_file
```

该命令将对 old_fiel 和 new_file 做比较，然后生成补丁，将补丁内容保存到 patch_file 中。其中"-r"选项是递归比较，也就是说可以对整个目录生成补丁。

应用补丁：

```
patch -p 数字 [file_name] < patch_file
```

去除补丁：

```
patch -R -p 数字 [file_name] < patch_file
```

对于-p 选项后的数字，它表示从前向后要忽略的目录层数，其实没必要想的那么复杂。首先，进入到待打补丁所在目录，如果是对改目录下单个文件打补丁，则使用-p0，对目录打补丁，则使用-p1。

```
# 对文件而言
diff  -uN from-file to-file > to-file.patch
patch  -p0 < to-file.patch
patch  -R  -p0 < to-file.patch
# 对目录而言
diff  -uNr from-dir to-dir > to-dir.patch
cd from-dir
```

```
patch -p1 < to-dir.patch
patch -R -p1 < to-dir.patch
```

8.6 源码编译安装程序

8.6.1 源码编译的几个阶段

拿到源码的压缩包后，首先就是解压，这就不需说了。解压后，进入解压目录，这是必须动作，之后就是源码编译的一般步骤。并非适用所有程序的编译，但知道过程之后也可以举一反三了。

1. 阅读解压目录中的 INSTALL/README 文件。如果不是对着官方手册或文档，那么在安装前务必读一读 INSTALL 文件或 README 文件，只需读其中如何安装的部分即可。

2.解压后的目录里一般还有 configure 文件（也可能是 config 文件）。执行"./configure"或带有编译选项的"./configure"，检查系统环境是否符合满足安装要求，并将定义好的安装配置写入和系统环境信息写入 Makefile 文件中。里面包含了如何编译、启用哪些功能、安装路径等信息。

3.执行 make 命令进行编译。make 命令会根据 Makefile 文件进行编译。编译工作主要是调用编译器(如 gcc)将源码编译为可执行文件，通常需要一些函数库才能产生一个完整的可执行文件。

4.make install 将上一步所编译的数据复制到指定的目录下。

这就已经完成编译程序的过程了。

8.6.2 configure 脚本的通用功能

configure 一般都会接受以下几个编译选项：

```
--prefix=          : 指定安装的路径
--sysconfdir=      : 指定配置文件目录
--enable-feature   : 启用某个特性
--disable-fecture  : 禁用特性
--with-function    : 启用某功能
--without-function : 禁用某功能
```

不同的程序，其 configure 选项不尽相同，应使用"./configure --help"获取具体的信息。

8.6.3 源码编译安装须知

1.上面的每一个步骤都不能出错，否则后一步都不能正常进行。

2.上面的步骤每一步如果出现警告或错误，如果步骤未停止而是继续，则属于可忽略错误或警告，不影响安装。但是进行的步骤停止了出现警告或错误，则根据步骤考虑对策。可以使用"$?"命令查看上一个命令是否正确执行，如果是返回 0 则是正确，其他的则是错误。

3.卸载时，只需删除安装目录即可。

因此，若要便于删除，最好将源码程序安装在/usr/local/对应的目录下。例如 apache2 安装在/usr/local/apache2 下。

4.通过源码编译的软件，需要做一些后续操作，虽非必须，但是都是个性化定制，方便以后的操作。个性化定制大致包括以下几项：

(1).将安装路径下的命令路径加入到环境变量。

```
echo "export PATH=/usr/local/apache/bin:$PATH" > /etc/profile.d/apache.sh
chmod +x /etc/profile.d/apache.sh
source /etc/profile.d/apache.sh
```

(2).按需求定制服务启动脚本，并考虑是否加入开机启动项。

(3).输出头文件和库文件。

头文件库文件很多时候只是为其他程序提供的，所以可能不输出它们的路径也不会影响该程序的运行。

```
# 输出头文件
ln -s /usr/local/apache/include /usr/include/apache
# 输出库文件
echo "/usr/local/apache/lib" >/etc/ld.so.conf.d/apache.conf
ldconfig
```

(4).导出 man 路径

```
echo  "MANPATH /usr/local/apache/man" >> /etc/man.conf
```

第9章 进程和信号

9.1 进程简单说明

进程是一个非常复杂的概念，涉及的内容也非常非常多。在这一小节所列出内容，已经是我极度简化后的内容了，应该尽可能都理解下来，我觉得这些理论比如何使用命令来查看状态更重要，而且不明白这些理论，后面查看状态信息时基本上不知道状态对应的是什么意思。

但对于非编程人员来说，更多的进程细节也没有必要去深究，当然，多多益善是肯定的。

9.1.1 进程和程序的区别

程序是二进制文件，是静态存放在磁盘上的，不会占用系统运行资源(cpu/内存)。

进程是用户执行程序或者触发程序的结果，可以认为进程是程序的一个运行实例。进程是动态的，会申请和使用系统资源，并与操作系统内核进行交互。在后文中，不少状态统计工具的结果中显示的是 system 类的状态，其实 system 状态的同义词就是内核状态。

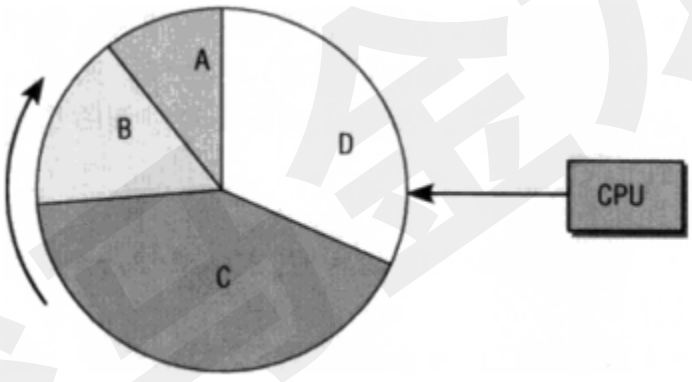
9.1.2 多任务和 cpu 时间片

现在所有的操作系统都能"同时"运行多个进程，也就是多任务或者说是并行执行。但实际上这是人类的错觉，一颗物理 cpu 在同一时刻只能运行一个进程，只有多颗物理 cpu 才能真正意义上实现多任务。

人类会产生错觉，以为操作系统能并行做几件事情，这是通过在极短时间内进行进程间切换实现的，因为时间极短，前一刻执行的是进程 A，下一刻切换到进程 B，不断的在多个进程间进行切换，使得人类以为在同时处理多件事情。

不过，cpu 如何选择下一个要执行的进程，这是一件非常复杂的事情。在 Linux 上，决定下一个要运行的进程是通过"调度类"(调度程序)来实现的。程序何时运行，由进程的优先级决定，但要注意，优先级值越低，优先级就越高，就越快被调度类选中。除此之外，优先级还影响分配给进程的时间片长短。在 Linux 中，改变进程的 nice 值，可以影响某类进程的优先级值。

有些进程比较重要，要让其尽快完成，有些进程则比较次要，早点或晚点完成不会有太大影响，所以操作系统要能够知道哪些进程比较重要，哪些进程比较次要。比较重要的进程，应该多给它分配一些 cpu 的执行时间，让其尽快完成。下图是 cpu 时间片的概念。



通过时间片分配CPU时间

由此可以知道，所有的进程都有机会运行，但重要的进程总是会获得更多的 cpu 时间，这种方式是"抢占式多任务处理"：内核可以强制在时间片耗尽的情况下收回 cpu 使用权，并将 cpu 交给调度类选中的进程，此外，在某些情况下也可以直接抢占当前运行的进程。随着时间的流逝，分配给进程的时间也会被逐渐消耗，当分配时间消耗完毕时，内核收回此进程的控制权，并让下一个进程运行。但因为前面的进程还没有完成，在未来某个时候调度类还是会选中它，所以内核应该将每个进程临时停止时的运行时环境(寄存器中的内容和页表)保存下来(保存位置为内核占用的内存)，这称为保护现场，在下次进程恢复运行时，将原来的运行时环境加载到 cpu 上，这称为恢复现场，这样 cpu 可以在当初的运行时环境下继续执行。

看书上说，Linux 的调度器不是通过 cpu 的时间片流逝来选择下一个要运行的进程的，而是考虑进程的等待时间，即在就绪队列等待了多久，那些对时间需求最严格的进程应该尽早安排其执行。另外，重要的进程分配的 cpu 运行时间自然会较多。

调度类选中了下一个要执行的进程后，要进行底层的任务切换，也就是上下文切换，这一过程需要和 cpu 进程紧密的交互。进程切换不应太频繁，也不应太慢。切换太频繁将导致 cpu 闲置在保护和恢复现场的时间过长，保护和恢复现场对人类或者进程来说是没有产生生产力的(因为它没有在执行程序)。切换太慢将导致进程调度切换慢，很可能下一个进程要等待很久才能轮到它执行，直白的说，如果你发出一个 ls 命令，你可能要等半天，这显然是不允许的。

至此，也就知道了 cpu 的衡量单位是时间，就像内存的衡量单位是空间大小一样。进程占用的 cpu 时间长，说明 cpu 运行在它身上的时间就长。注意，cpu 的百分比值不是其工作强度或频率高低，而是"进程占用 cpu 时间/cpu 总时间"，这个衡量概念一定不要搞错。

9.1.3 父子进程及创建进程的方式

根据执行程序的用户 UID 以及其他标准，会为每一个进程分配一个唯一的 PID。

父子进程的概念，简单来说，在某进程(父进程)的环境下执行或调用程序，这个程序触发的进程就是子进程，而进程的 PPID 表示的是该进程的父进程的 PID。由此也知道了，子进程总是由父进程创建。

在 Linux，父子进程以树型结构的方式存在，父进程创建出来的多个子进程之间称为兄弟进程。CentOS 6 上，init 进程是所有进程的父进程，CentOS 7 上则为 systemd。

Linux 上创建子进程的方式有三种：一种是 fork 出来的进程，一种是 exec 出来的进程，一种是 clone 出来的进程。

(1).fork 是复制进程，它会复制当前进程的副本(不考虑写时复制的模式)，以适当的方式将这些资源交给子进程。所以子进程掌握的资源和父进程是一样的，包括内存中的内容，**所以也包括环境变量和变量**。但父子进程是完全独立的，它们是一个程序的两个实例。

(2).exec 是加载另一个应用程序，替代当前运行的进程，也就是说在不创建新进程的情况下加载一个新程序。exec 还有一个动作，在进程执行完毕后，退出 exec 所在的环境(实际上是进程直接跳转到 exec 上，执行完 exec 就直接退出。而非 exec 加载程序的方式是：父进程睡眠，然后执行子进程，执行完后回到父进程，所以不会立即退出当前环境)。所以为了保证进程安全，若要形成新的且独立的子进程，都会先 fork 一份当前进程，然后在 fork 出来的子进程上调用 exec 来加载新程序替代该子进程。例如在 bash 下执行 cp 命令，会先 fork 出一个 bash，然后再 exec 加载 cp 程序覆盖子 bash 进程变成 cp 进程。但要注意，fork 进程时会复制所有内存页，但使用 exec 加载新程序时会初始化地址空间，意味着复制动作完全是多余的操作，当然，有了写时复制技术不用过多考虑这个问题。

(3).clone 用于实现线程。clone 的工作原理和 fork 相同，但 clone 出来的新进程不独立于父进程，它只会和父进程共享某些资源，在 clone 进程的时候，可以指定要共享的是哪些资源。

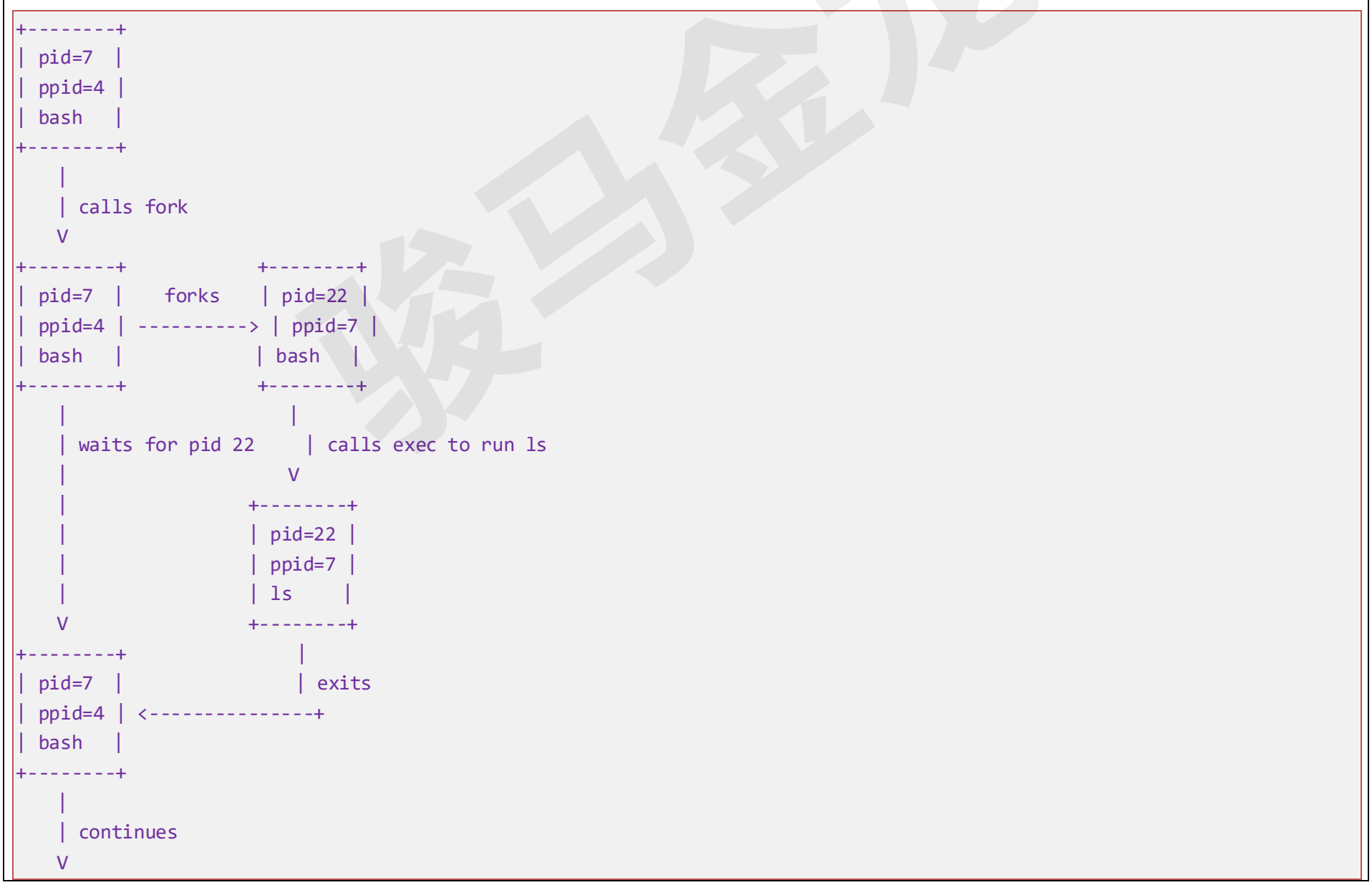
题外知识：如何创建一个子进程？

每次 fork 一个进程的时候，虽然调用一次 fork()，会分别为两个进程返回两个值：对子进程的返回值为 0，对父进程的返回值是子进程的 pid。所以，可以使用下面的 shell 伪代码来描述运行一个 ls 命令时的过程：

```
fpid=`fork()`
if [ $fpid = 0 ]{
    exec(ls) || echo "Can't exec ls"
    exit
}
wait($fpid)
```

假设上面是在 shell 脚本中执行 ls 命令，那么 fork 的是 shell 脚本进程。fork 后，父进程将继续执行，且 if 语句判断失败，于是执行 wait；而子进程执行时将检测到 fpid=0，于是执行 exec(ls)，当 ls 执行结束，子进程因为 exec 的原因将退出。于是父进程的 wait 等待完成，继续执行后面的代码。

如果在这个 shell 脚本中某个位置，执行 exec 命令(exec 命令调用的其实就是 exec 家族函数)，shell 脚本进程直接切换到 exec 命令上，执行完 exec 命令，就表示进程终止，于是 exec 命令后面的所有命令都不会再执行。

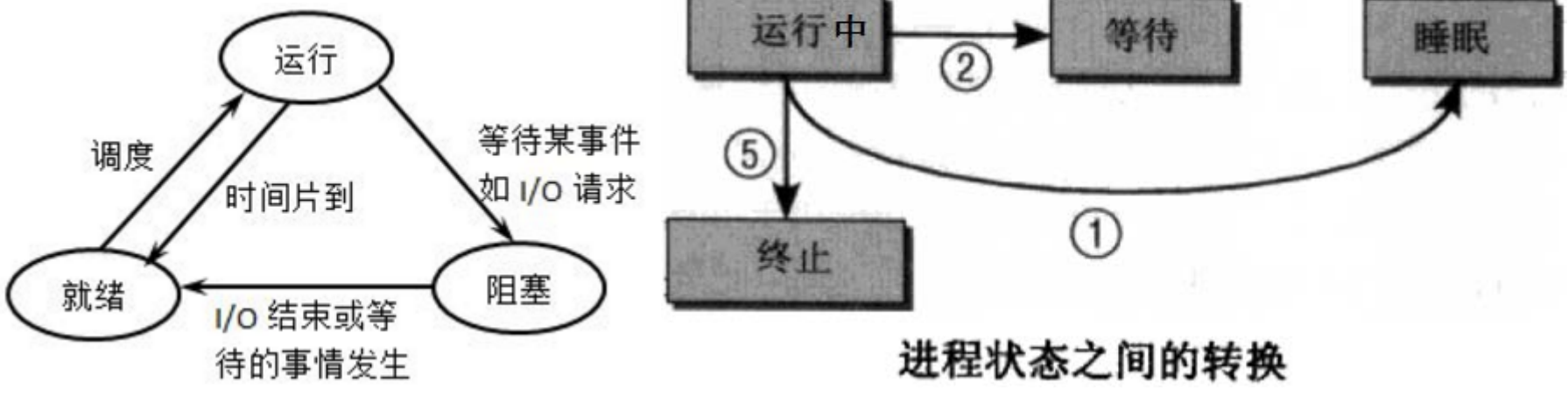


一般情况下，兄弟进程之间是相互独立、互不可见的，但有时候通过特殊手段，它们会实现进程间通信。例如管道协调了两边的进程，两边的进程属于同一个进程组，它们的 PPID 是一样的，管道使得它们可以以"管道"的方式传递数据。

进程是有所有者的，也就是它的发起者，某个用户如果它非进程发起者、非父进程发起者、非 root 用户，那么它无法杀死进程。且杀死父进程(非终端进程)，会导致子进程变成孤儿进程，孤儿进程的父进程总是 init/systemd。

9.1.4 进程的状态

进程并非总是处于运行中，至少 cpu 没运行在它身上时它就是非运行的。进程有几种状态，不同的状态之间可以实现状态切换。下图是非常经典的进程状态描述图，个人感觉右图更加易于理解。



进程状态之间的转换

运行态：进程正在运行，也即是 cpu 正在它身上。

就绪(等待)态：进程可以运行，已经处于等待队列中，也就是说调度类下次可能会选中它

睡眠(阻塞)态：进程睡眠了，不可运行。

各状态之间的转换方式为：(也许可能不太好理解，可以结合稍后的例子)

(1)新状态－>就绪态：当等待队列允许接纳新进程时，内核便把新进程移入等待队列。

(2)就绪态－>运行态：调度类选中等待队列中的某个进程，该进程进入运行态。

(3)运行态－>睡眠态：正在运行的进程因需要等待某事件(如 IO 等待、信号等待等)的出现而无法执行，进入睡眠态。

(4)睡眠态－>就绪态：进程所等待的事件发生了，进程就从睡眠态排入等待队列，等待下次被选中执行。

(5)运行态－>就绪态：正在执行的进程因时间片用完而被暂停执行；或者在抢占式调度方式中，高优先级进程强制抢占了正在执行的低优先级进程。

(6)运行态－>终止态：一个进程已完成或发生某种特殊事件，进程将变为终止状态。对于命令来说，一般都会返回退出状态码。

注意上面的图中，没有"就绪-->睡眠"和"睡眠-->运行"的状态切换。这很容易理解。对于"就绪-->睡眠",等待中的进程本就已经进入了等待队列，表示可运行，而进入睡眠态表示暂时不可运行，这本身就是冲突的；对于"睡眠-->运行"这也是行不通的，因为调度类只会从等待队列中挑出下一次要运行的进程。

再说说运行态-->睡眠态。从运行态到睡眠态一般是等待某事件的出现，例如等待信号通知，等待 IO 完成。信号通知很容易理解，而对于 IO 等待，程序要运行起来，cpu 就要执行该程序的指令，同时还需要输入数据，可能是变量数据、键盘输入数据或磁盘文件中的数据，后两种数据相对 cpu 来说，都是极慢极慢的。但不管怎样，如果 cpu 在需要数据的那一刻却得不到数据，cpu 就只能闲置下来，这肯定是不应该的，因为 cpu 是极其珍贵的资源，所以内核应该让正在运行且需要数据的进程暂时进入睡眠，等它的数据都准备好了再回到等待队列等待被调度类选中。这就是 IO 等待。

其实上面的图中少了一种进程的特殊状态——僵尸态。僵尸态进程表示的是进程已经转为终止态，它已经完成了它的使命并消逝了，但是内核还没有来得及将它在进程列表中的项删除，也就是说内核没给它料理后事，这就造成了一个进程是死的也是活着的假象，说它死了是因为它不再消耗额外资源(但可能会占用未释放的资源)，调度类也不可能选中它并让它运行，说它活着是因为在进程列表中还存在对应的表项，可以被捕捉到。僵尸态进程并不一定会占用多少资源(除非 fork 出来的大量进程都占用了未释放的资源且成了僵尸进程)，正常情况下的大多数僵尸进程仅在进程列表中占用一点点的内存，大多数僵尸进程的出现都是因为进程正常终止(包括 kill -9)，但父进程没有确认该进程已经终止，内核也不知道该进程已经终止了。僵尸进程更具体说明见后文。

另外，睡眠态是一个非常宽泛的概念，分为可中断睡眠和不可中断睡眠。可中断睡眠是允许接收外界信号和内核信号而被唤醒的睡眠，绝大多数睡眠都是可中断睡眠，能 ps 或 top 捕捉到的睡眠也几乎总是可中断睡眠；不可中断睡眠只能由内核发起信号来唤醒，外界无法通过信号来唤醒，主要表现在和硬件交互的时候。例如 cat 一个文件时，从硬盘上加载数据到内存中，在和硬件交互的那一小段时间一定是不可中断的，否则在加载数据的时候突然被人为发送的信号手动唤醒，而被唤醒时和硬件交互的过程又还没完成，所以即使唤醒了也没法将 cpu 交给它运行，所以 cat 一个文件的时候不可能只显示一部分内容。而且，不可中断睡眠若能被人为唤醒，更严重的后果是硬件崩溃。由此可知，不可中断睡眠是为了保护某些重要进程，也是为了让 cpu 不被浪费。

其实只要发现进程存在，且非僵尸态进程，还不占用 cpu 资源，那么它就是睡眠的。包括后文中出现的暂停态、追踪态，它们也都是睡眠态。

9.1.5 举例分析进程状态转换过程

进程间状态的转换情况可能很复杂，这里举一个例子，尽可能详细地描述它们。

以在 bash 下执行 cp 命令为例。在当前 bash 环境下，处于可运行状态(即就绪态)时，当执行 cp 命令时，首先 fork 出一个 bash 子进程，然后在子 bash 上 exec 加载 cp 程序，cp 子进程进入等待队列，由于在命令行下敲的命令，所以优先级较高，调度类很快选中。在 cp 这个子进程执行过程中，父进程 bash 会进入睡眠状态(不仅是因为 cpu 只有一颗的情况下一次只能执行一个进程，还因为进程等待)，并等待被唤醒，此刻 bash 无法和人类交互。当 cp 命令执行完毕，它将自己的退出状态码告知父进程，此次复制是成功还是失败，然后 cp 进程自己消逝掉，父进程 bash 被唤醒再次进入等待队列，并且此时 bash 已经获得了 cp 退出状态码。根据状态码这个信号，父进程 bash 知道了子进程已经终止，所以通告给内核，内核收到通知后将进程列表中的 cp 进程项删除。至此，整个 cp 进程正常完成。

假如 cp 这个子进程复制的是一个大文件，一个 cpu 时间片无法完成复制，那么在一个 cpu 时间片消耗尽的时候它将进入等待队列。

假如 cp 这个子进程复制文件时，目标位置已经有了同名文件，那么默认会询问是否覆盖，发出询问时它等待 yes 或 no 的信号，所以它进入了睡眠状态(可中断睡眠)，当在键盘上敲入 yes 或 no 信号给 cp 的时候，cp 收到信号，从睡眠态转入就绪态，等待调度类选中它完成 cp 进程。

在 cp 复制时，它需要和磁盘交互，在和硬件交互的短暂过程中，cp 将处于不可中断睡眠。

假如 cp 进程结束了，但是结束的过程出现了某种意外，使得 bash 这个父进程不知道它已经结束了(在这个例子中肯定是不可能出现这种情况的)，那么 bash 就不会通知内核回收进程列表中的 cp 表项，cp 此时就成了僵尸进程。

9.1.6 进程结构和子 shell

前台进程：一般命令(如 cp 命令)在执行时都会 fork 子进程来执行，**在子进程执行过程中，父进程会进入睡眠，这类是前台进程。**前台进程执行时，其父进程睡眠，因为 cpu 只有一颗，即使是多颗 cpu，也会因为执行流(进程等待)的原因而只能执行一个进程，要想实现真正的多任务，应该使用进程内多线程实现多个执行流。

后台进程：若在执行命令时，在命令的结尾加上符号"&"，它会进入后台。将命令放入后台，会立即返回父进程，并返回该后台进程的的 jobid 和 pid，所以后台进程的父进程不会进入睡眠。当后台进程出错，或者执行完成，总之后台进程终止时，父进程会收到信号。所以，通过在命令后加上"&"，再在"&"后给定另一个要执行的命令，可以实现"伪并行"执行的方式，例如"cp /etc/fstab /tmp & cat /etc/fstab"。

bash 内置命令：bash 内置命令是非常特殊的，父进程不会创建子进程来执行这些命令，而是直接在当前 bash 进程中执行。但如果将内置命令放在管道后，则此内置命令将和管道左边的进程同属于一个进程组，所以仍然会创建子进程。

说到这了，应该解释下子 shell，这个特殊的子进程。

一般 fork 出来的子进程，内容和父进程是一样的，包括变量，例如执行 cp 命令时也能获取到父进程的变量。但是 cp 命令是在哪里执行的呢？在子 shell 中。执行 cp 命令敲入回车后，当前的 bash 进程 fork 出一个子 bash，然后子 bash 通过 exec 加载 cp 程序替代子 bash。请不要在此纠结子 bash 和子 shell，如果搞不清它们的关系，就当它是同一种东西好了。

那是否可以理解为所有命令、脚本其运行环境都是在子 shell 中呢？显然，上面所说的 bash 内置命令不是在子 shell 中运行的。其他的所有方式，都是在子 shell 中完成，只不过方式不尽相同。

分为几种情况(只列出几种比较能说明问题的例子，还有其它很多种会进入子 shell 的情况)：

①.执行 bash 内置命令：bash 内置命令是非常特殊的，父进程不会创建子进程来执行这些命令，而是直接在当前 bash 进程中执行。但如果将内置命令放在管道后，则此内置命令将和管道左边的进程同属于一个进程组，所以仍然会创建子进程，但却不一定是子 shell。请先阅读完下面的几种情况再来考虑此项。

②.显然它会进入子 shell 环境，它的绝大多数环境都是新配置的，因为会加载一些环境配置文件。事实上 fork 出来的 bash 子进程内容完全继承父 shell，但因**重新加载了环境配置项，所以子 shell 没有继承普通变量，更准确的说是覆盖了从父 shell 中继承的变量。**不妨试试在/etc/bashrc 文件中定义一个变量，再在父 shell 中 export 名称相同值却不同的环境变量，然后到子 shell 中看看该变量的值为何？

- 其实执行 bash 命令，既可以认为进入了子 shell，也可以认为没有进入子 shell。在执行 bash 命令后从变量$BASH_SUBSHELL 的值为 0 可以认为它没有进入子 shell。但从执行 bash 命令后进入了新的 shell 环境来看，它有其父 bash 进程，且$BASHPID 值和父 shell 不同，所以它算是进入了子 shell。
- 执行 bash 命令更应该被认为是进入了一个完全独立的、全新的 shell 环境，而不应该认为是进入了片面的子 shell 环境。

③.执行 shell 脚本：因为脚本中第一行总是"#!/bin/bash"或者直接"bash xyz.sh"，所以这和上面的执行 bash 进入子 shell 其实是一回事，都是使用 bash 命令进入子 shell。只不过此时的 bash 命令和情况②中直接执行 bash 命令所隐含的选项不一样，所以继承和加载的 shell 环境也不一样。事实也确实如此，**shell 脚本只会继承父 shell 的一项属性：父进程所存储的各命令的路径。**

- 另外，执行 shell 脚本有一个动作：命令执行完毕后自动退出子 shell。

④.执行非 bash 内置命令：例如执行 cp 命令、grep 命令等，它们直接 fork 一份 bash 进程，然后使用 exec 加载程序替代该子 bash。此类子进程会继承所有父 bash 的环境。但严格地说，这已经不是子 shell，因为 exec 加载的程序已经把子 bash 进程替换掉了，这意为着丢失了很多 bash 环境。

⑤.非内置命令的命令替换：当命令行中包含了命令替换部分时，将开启一个子 shell 先执行这部分内容，再将执行结果返回给当前命令。因为这次的子 shell 不是通过 bash 命令进入的子 shell，所以它会继承父 shell 的所有变量内容。这也就解释了"$(echo $$)"中"$$"的结果是当前 bash 的 pid 号，而不是子 shell 的 pid 号，因为它不是使用 bash 命令进入的子 shell。

⑥.使用括号()组合一系列命令：例如(ls;date;echo haha)，独立的括号将会开启一个子 shell 来执行括号内的命令。这种情况等同于情况⑤。

最后需要说明的是，子 shell 的环境设置不会粘滞到父 shell 环境，也就是说子 shell 的变量等不会影响父 shell。

还有两种特殊的脚本调用方式：exec 和 source。

exec：exec 是加载程序替换当前进程，所以它不开启子 shell，而是直接在当前 shell 中执行命令或脚本，执行完 exec 后直接退出 exec 所在的 shell。这就解释了为何 bash 下执行 cp 命令时，cp 执行完毕后会自动退出 cp 所在的子 shell。

source：source 一般用来加载环境配置类脚本。它也不会开启子 shell，直接在当前 shell 中执行调用脚本且执行脚本后不退出当前 shell，所以脚本会继承当前已有的变量，且脚本执行完毕后加载的环境变量会粘滞给当前 shell，在当前 shell 生效。

9.2 job 任务

大部分进程都能将其放入后台，这时它就是一个后台任务，所以常称为 job，每个开启的 shell 会维护一个 job table，后台中的每个 job 都在 job table 中对应一个 Job 项。

手动将命令或脚本放入后台运行的方式是在命令行后加上"&"符号。例如：

```
[root@server2 ~]# cp /etc/fstab  /tmp/ &
[1] 8701
```

将进程放入后台后，会立即返回其父进程，一般对于手动放入后台的进程都是在 bash 下进行的，所以立即返回 bash 环境。在返回父进程的同时，还会返回给父进程其 jobid 和 pid。未来要引用 jobid，都应该在 jobid 前加上百分号"%"，其中"%%"表示当前 job，例如"kill -9 %1"表示杀掉 jobid 为 1 的后台进程，如果不加百分号，完了，把 Init 进程给杀了（但该进程特殊，不会受影响）。

通过 jobs 命令可以查看后台 job 信息。

```
jobs [-lrs] [jobid]
-l：jobs 默认不会列出后台工作的 PID，加上-l 会列出进程的 PID
-r：显示后台工作处于 run 状态的 jobs
-s：显示后台工作处于 stopped 状态的 jobs
```

通过"&"放入后台的任务，在后台中仍会处于运行中。当然，对于那种交互式如 vim 类的命令，将转入暂停运行状态。

```
[root@server2 ~]# sleep 10 &
[1] 8710
[root@server2 ~]# jobs
[1]+  Running                 sleep 10 &
```

一定要注意，此处看到的是 running 和 ps 或 top 显示的 R 状态，它们并不总是表示正在运行，处于等待队列的进程也属于 running。它们都属于 task_running。

另一种手动加入后台的方式是按下 CTRL+Z 键，这可以将正在运行中的进程加入到后台，但这样加入后台的进程会在后台暂停运行。

```
[root@server2 ~]# sleep 10
^Z
[1]+  Stopped                 sleep 10
[root@server2 ~]# jobs
[1]+  Stopped                 sleep 10
```

从 jobs 信息也看到了在每个 jobid 的后面有个"+"号，还有"-"，或者不带符号。

```
[root@server2 ~]# sleep 30&vim /etc/my.cnf&sleep 50&
[1] 8915
[2] 8916
[3] 8917
[root@server2 ~]# jobs
[1]   Running                 sleep 30 &
[2]+  Stopped                 vim /etc/my.cnf
[3]-  Running                 sleep 50 &
```

发现 vim 的进程后是加号，"+"表示最近进入后台的作业，也称为当前作业，"-"表示倒数第二个进入后台的作业。如果后进入后台的作业先执行完成了，则"+"和"-"所代表的作业顺位前移。如果只有一个后台作业，则"+"和"-"都代表这个作业。所以，使用"%+"和"%%"可以代表最后一个作业的 jobid，使用"%-"可以代表倒数第二个 job 的 jobid。

回归正题。既然能手动将进程放入后台，那肯定能调回到前台，调到前台查看了下执行进度，又想调入后台，这肯定也得有方法，总不能使用 CTRL+Z 以暂停方式加到后台吧。

fg 和 bg 命令分别是 foreground 和 background 的缩写，也就是放入前台和放入后台，严格的说，是以运行状态放入前台和后台，即使原来任务是 stopped 状态的。

操作方式也很简单，直接在命令后加上 jobid 即可(即[fg|bg] [%jobid])，不给定 jobid 时操作的将是当前任务，即带有"+"的任务项。

```
[root@server2 ~]# sleep 20
^Z              # 按下 CTRL+Z 进入暂停并放入后台
[3]+  Stopped                 sleep 20
[root@server2 ~]# jobs
[2]-  Stopped                 vim /etc/my.cnf
[3]+  Stopped                 sleep 20        # 此时为 stopped 状态
[root@server2 ~]# bg %3                   # 使用 bg 或 fg 可以让暂停状态的进程变会运行态
[3]+ sleep 20 &
[root@server2 ~]# jobs
[2]+  Stopped                 vim /etc/my.cnf
[3]-  Running                 sleep 20 &      # 已经变成运行态
```

disown 命令可以从 job table 中直接移除一个 job，仅仅只是移出 job table，并非是结束任务。而且移出 job table 后，作业将脱离 shell 管理，不再依赖于终端，当终端断开会立即挂在 init/systemd 进程之下。所以，disown 命令提供了让进程脱离终端的另一种方式。

```
disown [-ar] [-h] [%jobid ...]
选项说明：
-h：给定该选项，将不从 job table 中移除 job，而是将其设置为不接受 shell 发送的 sighup 信号。具体说明见"信号"小节。
-a：如果没有给定 jobid，该选项表示针对 Job table 中的所有 job 进行操作。
-r：如果没有给定 jobid，该选项严格限定为只对 running 状态的 job 进行操作
```

如果不给定任何选项，该 shell 中所有的 job 都会被移除，移除是 disown 的默认操作，如果也没给定 jobid，而且也没给定-a 或-r，则表示只针对当前任务即带有"+"号的任务项。例如：

```
[root@server2 ~]# sleep 30 & sleep 40 &
[1] 5199
[2] 5200
[root@server2 ~]# jobs
[1]-  Running                 sleep 30 &
[2]+  Running                 sleep 40 &
[root@server2 ~]# disown %2
[root@server2 ~]# jobs        # 已经移除一个
[1]+  Running                 sleep 30 &
```

9.3 终端和进程的关系

使用 pstree 命令查看下当前的进程，不难发现在某个终端执行的进程其父进程或上几个级别的父进程总是会是终端的连接程序。

例如下面筛选出了两个终端下的父子进程关系，第一个行是 tty 终端(即直接在虚拟机中)中执行的进程情况，第二行和第三行是 ssh 连接到 Linux 上执行的进程。

```
[root@server2 ~]# pstree -c | grep bash
        |-login---bash---bash---vim
        |-sshd-+-sshd---bash
        |      `-sshd---bash-+-grep
```

正常情况下杀死父进程会导致子进程变为孤儿进程，即其 PPID 改变，但是杀掉终端这种特殊的进程，会导致该终端上的所有进程都被杀掉。这在很多执行长时间任务的时候是很不方便的。比如要下班了，但是你连接的终端上还在执行数据库备份脚本，这可能会花掉很长时间，如果直接退出终端，备份就终止了。所以应该保证一种安全的退出方法。

一般的方法也是最简单的方法是使用 nohup 命令带上要执行的命令或脚本放入后台，这样任务就脱离了终端的关联。当终端退出时，该任务将自动挂到 init(或 systemd)进程下执行。

另一种方法是使用 screen 这个工具，该工具可以模拟多个物理终端，虽然模拟后 screen 进程仍然挂在其所在的终端上的，但同 nohup 一样，当其所在终端退出后将自动挂到 init/systemd 进程下继续存在，只要 screen 进程仍存在，其所模拟的物理终端就会一直存在，这样就保证了模拟终端中的进程继续执行。它的实现方式其实和 nohup 差不多，只不过它花样更多，管理方式也更多。一般对于简单的后台持续运行进程，使用 nohup 足以。

另外，在**子 shell 中的后台进程在终端被关闭时也会脱离终端，因此不会受 shell 和终端的控制。例如 shell 脚本中的后台进程，再如"(sleep 10 &)"。**

可能你已经发现了，很多进程是和终端无关的，也就是不依赖于终端，这类进程一般是内核类进程/线程以及 daemon 类进程，若它们也依赖于终端，则终端一被终止，这类进程也立即被终止，这是绝对不允许的。

9.4 信号

信号在操作系统中控制着进程的绝大多数动作，信号可以让进程知道某个事件发生了，也指示着进程下一步要做出什么动作。信号的来源可以是硬件信号(如按下键盘或其他硬件故障)，也可以是软件信号(如 kill 信号，还有内核发送的信号)。不过，很多可以感受到的信号都是从进程所在的控制终端发送出去的。

9.4.1 需知道的信号

Linux 中支持非常多种信号，它们都以 SIG 字符串开头，SIG 字符串后的才是真正的信号名称，信号还有对应的数值，其实数值才是操作系统真正认识的信号。但由于不少信号在不同架构的计算机上数值不同(例如 CTRL+Z 发送的 SIGSTP 信号就有三种值 18,20,24)，所以在不确定信号数值是否唯一的时候，最好指定其字符名称。

以下是需要了解的信号。

```
Signal     Value     Comment
──────────────────────────────────────────────────────
SIGHUP       1       终止进程，特别是终端退出时，此终端内的进程都将被终止
SIGINT       2       中断进程，可被捕捉和忽略，几乎等同于 sigterm，所以也会尽可能的释放执行 clean-up，释放资源，保存状态等(CTRL+C)
SIGQUIT      3       从键盘发出杀死(终止)进程的信号。

SIGKILL      9       强制杀死进程，该信号不可被捕捉和忽略，进程收到该信号后不会执行任何 clean-up 行为，所以资源不会释放，状态不会保存
SIGTERM     15       杀死(终止)进程，可被捕捉和忽略，几乎等同于 sigint 信号，会尽可能的释放执行 clean-up，释放资源，保存状态等
```

SIGCHLD	17	当子进程中断或退出时，发送该信号告知父进程自己已完成，父进程收到信号将告知内核清理进程列表。所以该信号可以解除僵尸进程，也可以让非正常退出的进程工作得以正常的 clean-up，释放资源，保存状态等。
SIGSTOP	19	该信号是不可被捕捉和忽略的进程停止信息，收到信号后会进入 stopped 状态
SIGTSTP	20	该信号是可被忽略的进程停止信号(CTRL+Z)
SIGCONT	18	发送此信号使得 stopped 进程进入 running，该信号主要用于 jobs，例如 bg & fg 都会发送该信号。 可以直接发送此信号给 stopped 进程使其运行起来
SIGUSR1	10	用户自定义信号 1
SIGUSR2	12	用户自定义信号 2

除了这些信号外，还需要知道一个特殊信号：代码为 0 的信号。此信号为 EXIT 信号，表示直接退出。如果 kill 发送的信号是 0(即 kill -0)则表示不做任何处理直接退出，但执行错误检查：当检查发现给定的 pid 进程存在，则返回 0，否则返回 1。也就是说，0 信号可以用来检测进程是否存在，可以代替"ps aux | grep proc_name"。(man kill 中的原文为：If sig is 0, then no signal is sent, but error checking is still performed。而 man bash 的 trap 小节中有如下描述：If a sigspec is EXIT (0)，这说明 0 信号就是 EXIT 信号)

以上所列的信号中，只有 SIGKILL 和 SIGSTOP 这两个信号是不可被捕捉且不可被忽略的信号，其他所有信号都可以通过 trap 或其他编程手段捕捉到或忽略掉。

此外，经常看到有些服务程序(如 httpd/nginx)的启动脚本中使用 WINCH 和 USR1 这两个信号，发送这两个信号时它们分别表示 graceful stop 和 graceful restart。所谓的 graceful，译为优雅，不过使用这两个字去描述这种环境实在有点不伦不类。它对于后台服务程序而言，传达了几个意思：(1)当前已经运行的进程不再接受新请求(2)给当前正在运行的进程足够多的时间去完成正在处理的事情(3)允许启动新进程接受新请求(4)可能还有日志文件是否应该滚动、pid 文件是否修改的可能，这要看服务程序对信号的具体实现。

再来说说，为什么后台服务程序可以使用这两个信号。以 httpd 的为例，在其头文件 mpm_common.h 中有如下几行代码：

```
/* Signal used to gracefully restart */
#define AP_SIG_GRACEFUL SIGUSR1

/* Signal used to gracefully stop */
#define AP_SIG_GRACEFUL_STOP SIGWINCH
```

这说明注册了对应信号的处理函数，它们分别表示将接收到信号时，执行对应的 GRACEFUL 函数。

注意，SIGWINCH 是窗口程序的尺寸改变时发送改信号，如 vim 的窗口改变了就会发送该信号。但是对于后台服务程序，它们根本就没有窗口，所以 WINCH 信号对它们来说是没有任何作用的。因此，大概是约定俗成的，大家都喜欢用它来作为后台服务程序的 GRACEFUL 信号。但注意，WINCH 信号对前台程序可能是有影响的，不要乱发这种信号。同理，USR1 和 USR2 也是一样的，如果源代码中明确为这两个信号注册了对应函数，那么发送这两个信号就可以实现对应的功能，反之，如果没有注册，则这两个信号对进程来说是错误信号。

更多更详细的信号理解或说明，可以参考 wiki 的两篇文章：

jobs 控制机制：https://en.wikipedia.org/wiki/Job_control_(Unix)

信号说明：https://en.wikipedia.org/wiki/Unix_signal

9.4.2 SIGHUP

(1).当控制终端退出时，会向该终端中的进程发送 sighup 信号，因此该终端上行的 shell 进程、其他普通进程以及任务都会收到 sighup 而导致进程终止。

多种方式可以改变因终端中断发送 sighup 而导致子进程也被结束的行为，这里仅介绍比较常见的三种：一是使用 nohup 命令启动进程，它会忽略所有的 sighup 信号，使得该进程不会随着终端退出而结束；二是将待执行命令放入子 shell 中并放入后台运行，例如"(sleep 10 &)"；三是使用 disown，将任务列表中的任务移除出 job table 或者直接使用 disown -h 的功能设置其不接收终端发送的 sighup 信号。但不管是何种实现方式，终端退出后未被终止的进程将只能挂靠在 init/systemd 下。

(2).对于 daemon 类的程序(即服务性进程)，这类程序不依赖于终端(它们的父进程都是 init 或 systemd)，它们收到 sighup 信号时会重读配置文件并重新打开日志文件，使得服务程序可以不用重启就可以加载配置文件。

9.4.3 僵尸进程和 SIGCHLD

一个编程完善的程序，在子进程终止、退出的时候，内核会发送 SIGCHLD 信号给父进程，父进程收到信号就会对该子进程进行善后(接收子进程的退出状态、释放未关闭的资源)，同时内核也会进行一些善后操作(比如清理进程表项、关闭打开的文件等)。

在子进程死亡的那一刹那，子进程的状态就是僵尸进程，但因为发出了 SIGCHLD 信号给父进程，父进程只要收到该信号，子进程就会被清理也就不再是僵尸进程。所以正常情况下，所有终止的进程都会有一小段时间处于僵尸态，只不过这种僵尸进程存在时间极短(倒霉的僵尸)，几乎是不可被 ps 或 top 这类的程序捕捉到的。

如果在特殊情况下，子进程终止了，但父进程没收到 SIGCHLD 信号，没收到这信号的原因可能是多种的，不管如何，此时子进程已经成了永存的僵尸，能轻易的被 ps 或 top 捕捉到。僵尸不倒霉，人类就要倒霉，但是僵尸爸爸并不知道它儿子已经变成了僵尸，因为有僵尸爸爸的掩护，僵尸道长即内核见不到小僵尸，所以也没法收尸。悲催的是，人类能力不足，直接发送信号(如 kill)给僵尸进程是无效的，因为僵尸进程本就是终结了的进程，它收不到信号，只有内核从进程列表中将僵尸进程表项移除才算完成收尸。

要解决掉永存的僵尸有几种方法：

(1).**杀死僵尸进程的父进程**。没有了僵尸爸爸的掩护，小僵尸就暴露给了僵尸道长的直系弟子 init/systemd，init/systemd 会定期清理它下面的各种僵尸进程。所以这种方法有点不讲道理，僵尸爸爸是正常的啊，不过如果僵尸爸爸下面有很多僵尸儿子，这僵尸爸爸肯定是有问题的，比如编程不完善，杀掉是应该的。

(2).**手动发送 SIGCHLD 信号给僵尸进程的父进程**。僵尸道长找不到僵尸，但被僵尸祸害的人类能发现僵尸，所以人类主动通知僵尸爸爸，让僵尸爸爸知道自己的儿子死而不僵，然后通知内核来收尸。

当然，第二种手动发送 SIGCHLD 信号的方法要求父进程能收到信号，而 SIGCHLD 信号默认是被忽略的，所以应该显式地在程序中加上获取信号的代码。也就是人类主动通知僵尸爸爸的时候，默认僵尸爸爸是不搭理人类的，所以要强制让僵尸爸爸收到通知。不过一般 daemon 类的程序在编程上都是很完善的，发送 SIGCHLD 总是会收到，不用担心。

9.4.4 手动发送信号(kill 命令)

使用 kill 命令可以手动发送信号给指定的进程。

```
kill [-s signal] pid...
kill [-signal] pid...
kill -l
```

使用 kill -l 可以列出 Linux 中支持的信号，有 64 种之多，但绝大多数非编程人员都用不上。

使用-s 或-signal 都可以发送信号，不给定发送的信号时，默认为 TREM 信号，即 kill -15。

```
shell> kill -9 pid1 pid2...
shell> kill -TREM pid1 pid2...
shell> kill -s TREM pid1 pid2...
```

9.4.5 pkill 和 killall

这两个命令都可以直接指定进程名来发送信号，不指定信号时，默认信号都是 TERM。

(1).pkill

pkill 和 pgrep 命令是同族命令，都是先通过给定的匹配模式搜索到指定的进程，然后发送信号(pkill)或列出匹配的进程(pgrep)，pgrep 就不介绍了。

pkill 能够指定模式匹配，所以可以使用进程名来删除，想要删除指定 pid 的进程，反而还要使用"-s"选项来指定。默认发送的信号是 SIGTERM 即数值为 15 的信号。

```
pkill [-signal] [-v] [-P ppid,...] [-s pid,...][-U uid,...] [-t term,...] [pattern]
选项说明：
-P ppid,... ：匹配 PPID 为指定值的进程
-s pid,...  ：匹配 PID 为指定值的进程
-U uid,...  ：匹配 UID 为指定值的进程，可以使用数值 UID，也可以使用用户名称
-t term,... ：匹配给定终端，终端名称不能带上"/dev/"前缀，其实"w"命令获得终端名就满足此处条件了，所以 pkill 可以直接杀掉整个终端
-v          ：反向匹配
-signal     ：指定发送的信号，可以是数值也可以是字符代表的信号
-f          ：默认情况下，pgrep/pkill 只会匹配进程名。使用-f 将匹配命令行

在 CentOS 7 上，还有两个好用的新功能
-F, --pidfile file：匹配进程时，读取进程的 pid 文件从中获取进程的 pid 值。这样就不用去写获取进程 pid 命令的匹配模式
-L, --logpidfile  ：如果"-F"选项读取的 pid 文件未加锁，则 pkill 或 pgrep 将匹配失败。
```

例如：

```
[root@xuexi ~]# ps x | grep ssh[d]
  1291 ?        Ss     0:00 /usr/sbin/sshd
 13193 ?        Ss     0:02 sshd: root@pts/1,pts/3,pts/0
```

现在想匹配/usr/sbin/sshd。

```
[root@xuexi ~]# pgrep bin/sshd
[root@xuexi ~]# pgrep -f bin/sshd
1291
```

可以看到第一个什么也不返回。因为不加-f 选项时，pgrep 只能匹配进程名，而进程名指的是 sshd，而非/usr/sbin/sshd，所以匹配失败。加上-f 后，就能匹配成功。**所以，当 pgrep 或 pkill 匹配不到进程时，考虑加上-f 选项。**

再例如踢出终端：

```
shell> pkill -t pts/0
```

(2).killall

killall 主要用于杀死一批进程，例如杀死整个进程组。其强大之处还体现在**可以通过指定文件来搜索哪个进程打开了该文件，然后对该进程发送信号，在这一点上，fuser 和 lsof 命令也一样能实现。**

```
killall [-r,--regexp] [-s,--signal signal] [-u,--user user] [-v,--verbose] [-w,--wait] [-I,--ignore-case] [--] name ...
选项说明：
-I          ：匹配时不区分大小写
-r          ：使用扩展正则表达式进行模式匹配
-s, --signal：发送信号的方式可以是-HUP 或-SIGHUP，或数值的"-1"，或使用"-s"选项指定信号
-u, --user  ：匹配该用户的进程
-v,         ：给出详细信息
-w, --wait  ：等待直到该杀的进程完全死透了才返回。默认 killall 每秒检查一次该杀的进程是否还存在，只有不存在了才会给出退出状态码。
              如果一个进程忽略了发送的信号、信号未产生效果、或者是僵尸进程将永久等待下去
```

9.5 fuser 和 lsof

fuser 可以查看文件或目录所属进程的 pid，即由此知道该文件或目录被哪个进程使用。例如，umount 的时候提示 the device busy 可以判断出来哪个进程在使用。而 lsof 则反过来，它是通过进程来查看进程打开了哪些文件，但要注意的是，一切皆文件，包括普通文件、目录、链接文件、块设备、字符设备、套接字文件、管道文件，所以 lsof 出来的结果可能会非常多。

9.5.1 fuser

```
fuser [-ki] [-signal] file/dir
-k：找出文件或目录的 pid，并试图 kill 掉该 pid。发送的信号是 SIGKILL
-i：一般和-k 一起使用，指的是在 kill 掉 pid 之前询问。
-signal：发送信号，如-1 -15，如果不写，默认-9，即 kill -9
不加选项:直接显示出文件或目录的 pid
在不加选项时，显示结果中文件或目录的 pid 后会带上一个修饰符：
    c:在当前目录下
    e:可被执行的
    f:是一个被开启的文件或目录
    F:被打开且正在写入的文件或目录
    r:代表 root directory
```

例如：

```
[root@xuexi ~]# fuser /usr/sbin/crond
/usr/sbin/crond:      1425e
```

表示/usr/sbin/crond 被 1425 这个进程打开了，后面的修饰符 e 表示该文件是一个可执行文件。

```
[root@xuexi ~]# ps aux | grep 142[5]
root       1425  0.0  0.1 117332  1276 ?        Ss   Jun10   0:00 crond
```

9.5.2 lsof

例如：

```
[root@linux ~]# lsof -u root | grep bash
bash    26199 root  cwd    DIR    3,2    4096    159875 /root
bash    26199 root  rtd    DIR    3,1    4096         2 /
bash    26199 root  txt    REG    3,1  686520    294425 /bin/bash
bash    26199 root  mem    REG    3,1   83160     32932 /usr/lib/gconv/BIG5.so
bash    26199 root  mem    REG    3,1   46552    915764 /lib/libnss_files-2.3.5.so
```

输出信息中各列意义：

COMMAND：进程的名称

PID：进程标识符

USER：进程所有者

FD：文件描述符，应用程序通过文件描述符识别该文件。如 cwd、txt 等

TYPE：文件类型，如 DIR、REG 等

DEVICE：指定磁盘的名称

SIZE/OFF：文件的大小或文件的偏移量(单位 kb)(size and offset)

NODE：索引节点（文件在磁盘上的标识）

NAME：打开文件的确切名称

```
lsof  /path/to/somefile：显示打开指定文件的所有进程之列表；建议配合 grep 使用
lsof -c string：显示其 COMMAND 列中包含指定字符(string)的进程所有打开的文件；可多次使用该选项
lsof -p PID：查看该进程打开了哪些文件
lsof -U：列出套接字类型的文件。一般和其他条件一起使用。如 lsof -u root -a -U
```

```
lsof -u uid/name：显示指定用户的进程打开的文件；可使用脱字符"^"取反，如"lsof -u ^root"将显示非root用户打开的所有文件
lsof +d /DIR/：显示指定目录下被进程打开的文件
lsof +D /DIR/：基本功能同上，但lsof会对指定目录进行递归查找，注意这个参数要比grep版本慢
lsof -a：按"与"组合多个条件，如lsof -a -c apache -u apache
lsof -N：列出所有NFS（网络文件系统）文件
lsof -n：不反解IP至HOSTNAME
lsof -i：用以显示符合条件的进程情况
lsof -i[46] [protocol][@host][:service|port]
    46：IPv4或IPv6
    protocol：TCP or UDP
    host：host name或ip地址，表示搜索哪台主机上的进程信息
    service：服务名称(可以不只一个)
    port：端口号 (可以不只一个)
```

大概"-i"是使用最多的了，而-i中使用最多的又是服务名或端口了。

```
[root@www ~]# lsof -i :22
COMMAND   PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
sshd     1390 root    3u  IPv4  13050      0t0  TCP *:ssh (LISTEN)
sshd     1390 root    4u  IPv6  13056      0t0  TCP *:ssh (LISTEN)
sshd    36454 root    3r  IPv4  94352      0t0  TCP xuexi:ssh->172.16.0.1:50018 (ESTABLISHED)
```

第10章 系统状态统计和查看

10.1 /proc 的意义及说明

在 Linux 中查看各种状态，其实质是查看内核中相关进程的数据结构中的项，通过工具将其格式化后输出出来。但是内核的数据是绝对不能随意查看或更改的，至少不能直接去修改。所以，在 linux 上出现了伪文件系统/proc，它是内核中各属性或状态向外提供访问和修改的接口。

在/proc 下，记录了内核自己的数据信息，各进程独立的数据信息，统计信息等。绝大多数文件都是只读不可改的，即使对 root 也一样，但/proc/sys 除外，为何如此稍后解释。

```
[root@xuexi ~]# ls /proc
1     7690  866          driver        key-users   mtd          swaps
10    78    9            execdomains   kmsg        mtrr         sys
1045  79    acpi         fb            kpagecount  net          sysrq-trigger
1084  8     buddyinfo    filesystems   kpageflags  pagetypeinfo sysvipc
11    80    bus          fs            loadavg     partitions   timer_list
1134  81    cgroups      interrupts    locks       sched_debug  timer_stats
1163  82    cmdline      iomem         mdstat      schedstat    tty
117   824   cpuinfo      ioports       meminfo     scsi         uptime
118   825   crypto       irq           misc        self         version
1195  84    devices      kallsyms      modules     slabinfo     vmallocinfo
12    85    diskstats    kcore         mounts      softirqs     vmstat
1205  86    dma          keys          mpt         stat         zoneinfo
```

其中数字命名的目录对应的是各进程的 pid 号，其内的文件记录的都是该进程当前的数据信息，且都是只读的，例如记录命令信息的 cmdline 文件，进程使用哪颗 cpu 信息 cpuset，进程占用内存的信息 mem 文件，进程 IO 信息 io 文件等其他各种信息文件。

```
[root@xuexi ~]# ls /proc/6982
attr       clear_refs       cpuset   fd      loginuid   mounts      numa_maps   pagemap      schedstat  stat     task
autogroup  cmdline          cwd      fdinfo  maps       mountstats  oom_adj     personality  sessionid  statm    wchan
auxv       comm             environ  io      mem        net         oom_score   root         smaps      status
cgroup     coredump_filter  exe      limits  mountinfo  ns          oom_score_adj  sched     stack      syscall
```

非数字命名的目录各有用途，例如 bus 表示总线信息，driver 表示驱动信息，fs 表示文件系统特殊信息，net 表示网络信息，tty 表示跟物理终端有关的信息，最特殊的两个是/proc/self 和/proc/sys。

先说/proc/self 目录，它表示的是当前正在访问/proc 目录的进程，因为/proc 目录是内核数据向外记录的接口，所以当前访问/proc 目录的进程表示的就是当前 cpu 正在执行的进程。如果执行 cat /proc/self/cmdline，会发现其结果总是该命令本身，因为 cat 是手动敲入的命令，它是重要性进程，cpu 会立即执行该命令。

再说/proc/sys 这个目录，该目录是为管理员提供用来修改内核运行参数的，所以该目录中的文件对 root 都是可写的，例如管理数据包转发功能的/proc/sys/net/ipv4/ip_forward 文件。使用 sysctl 命令修改内核运行参数，其本质也是修改/proc/sys 目录中的文件。

10.2 查看进程信息

10.2.1 pstree 命令

pstree 命令将以树的形式显示进程信息，默认树的分支是收拢的，也不显示 pid，要显示这些信息需要指定对应的选项。

```
pstree [-a] [-c] [-h] [-l] [-p] [pid]
选项说明：
-a：显示进程的命令行
-c：展开分支
-h：高亮当前正在运行的进程及其父进程
-p：显示进程 pid，此选项也将展开分支
-l：允许显示长格式进程。默认在显示结果中超过 132 个字符时将截断后面的字符。
```

例如：

```
[root@xuexi ~]# pstree -h
init─┬─abrtd
     ├─acpid
     ├─atd
     ├─auditd───{auditd}
     ├─console-kit-dae───63*[{console-kit-da}]
     ├─crond
     ├─dbus-daemon
     ├─dhclient
     ├─hald─┬─hald-runner─┬─hald-addon-acpi
     │      │             ├─hald-addon-inpu
     │      │             └─hald-addon-rfki
     │      └─{hald}
     ├─irqbalance
     ├─login───bash
     ├─master─┬─pickup
     │        └─qmgr
     ├─mcelog
     ├─5*[mingetty]
     ├─rsyslogd───3*[{rsyslogd}]
     ├─sshd─┬─2*[sshd───bash]
     │      └─sshd───bash───pstree
     ├─udevd───2*[udevd]
     └─xinetd
```

10.2.2 ps 命令

ps 命令查看当前这一刻的进程信息，注意查看的是静态进程信息，要查看随时刷新的动态进程信息(如 windows 的进程管理器那样，每秒刷新一次)，使用 top 或 htop 命令。

这个命令的 man 文档及其复杂，它同时支持 3 种类型的选项：GUN/BSD/UNIX，不同类型的选项其展示的信息格式不一样。有些加了"-"的是 SysV 风格的选项，不加"-"的是 BSD 选项，加不加"-"它们的意义是不一样的，例如 ps aux 和 ps -aux 是不同的。

其实只需掌握少数几个选项即可，关键的是要了解 ps 显示出的进程信息中每一列代表什么属性。

对于 BSD 风格的选项，只需知道一个用法 ps aux 足以，选项"a"表示列出依赖于终端的进程，选项"x"表示列出不依赖于终端的进程，所以两者结合就表示列出所有进程，选项"u"表示展现的进程信息是以用户为导向的，不用管它什么是以用户为导向，用 ps aux 就没错。

```
[root@server2 ~]# ps aux | tail
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       1340  0.0  0.0  27176   588 ?        Ss   20:30   0:00 /usr/sbin/xinetd -stayalive -pidfile /var/run/xinetd.pid
root       2266  0.0  0.1  93212  2140 ?        Ss   20:30   0:00 /usr/libexec/postfix/master -w
postfix    2268  0.0  0.2  93384  3992 ?        S    20:30   0:00 qmgr -l -t unix -u
postfix    2306  0.0  0.2  93316  3972 ?        S    20:31   0:00 pickup -l -t unix -u
root       2307  0.0  0.2 145552  5528 ?        Ss   20:31   0:00 sshd: root@pts/0
root       2309  0.0  0.0      0     0 ?        S<   20:31   0:00 [kworker/3:1H]
root       2310  0.0  0.1 116568  3184 pts/0    Ss   20:31   0:00 -bash
root       2352  0.0  0.0      0     0 ?        S<   20:31   0:00 [kworker/1:2H]
root       2355  0.0  0.0 139492  1632 pts/0    R+   20:34   0:00 ps aux
root       2356  0.0  0.0 107928   676 pts/0    R+   20:34   0:00 tail
```

各列的意义：

```
%CPU：表示 CPU 占用百分比，注意，CPU 的衡量方式是占用时间，所以百分比的计算方式是"进程占用 cpu 时间/cpu 总时间"，而不是 cpu 工作强度的状态。
%MEM：表示各进程所占物理内存百分比。
VSZ：表示各进程占用的虚拟内存，也就是其在线性地址空间中实际占用的内存。单位为 kb。
RSS：表示各进程占用的实际物理内存。单位为 Kb。
TTY：表示属于哪个终端的进程，"?"表示不依赖于终端的进程。
STAT：进程所处的状态。
    D：不可中断睡眠
    R：运行中或等待队列中的进程(running/runnable)
    S：可中断睡眠
    T：进程处于 stopped 状态
    Z：僵尸进程
    对于 BSD 风格的 ps 选项，进程的状态还会显示下面几个组合信息。
    <：高优先级进程
    N：低优先级进程
    L：该进程在内存中有被锁定的页
```

```
    s：表示该进程是 session leader，即进程组的首进程。例如管道左边的进程，shell 脚本中的 shell 进程
    l：表示该进程是一个线程
    +：表示是前端进程。前端进程一般来说都是依赖于终端的
START：表示进程是何时被创建的
TIME：表示各进程占用的 CPU 时间
COMMAND：表示进程的命令行。如果是内核线程，则使用方括号"[]"包围
```

注意到了没，ps aux 没有显示出 ppid，

另外常用的 ps 选项是 ps -elf。其中"-e"表示输出全部进程信息，"-f"和"-l"分别表示全格式输出和长格式输出。全格式会输出 cmd 的全部参数。

```
[root@server2 ~]# ps -lf
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S postfix     2306    2266  0  80   0 - 23329 ep_pol 20:31 ?        00:00:00 pickup -l -t unix -u
4 S root        2307    1141  0  80   0 - 36388 poll_s 20:31 ?        00:00:00 sshd: root@pts/0
1 S root        2309       2  0  60 -20 -     0 worker 20:31 ?        00:00:00 [kworker/3:1H]
4 S root        2310    2307  0  80   0 - 29142 wait   20:31 pts/0    00:00:00 -bash
1 S root        2433       2  0  60 -20 -     0 worker 21:21 ?        00:00:00 [kworker/1:1H]
1 S root        2479       2  0  80   0 -     0 worker 21:25 ?        00:00:00 [kworker/1:0]
1 S root        2503       2  0  60 -20 -     0 worker 21:28 ?        00:00:00 [kworker/1:2H]
1 S root        2532       2  0  80   0 -     0 worker 21:30 ?        00:00:00 [kworker/1:1]
0 R root        2539    2310  0  80   0 - 34873 -      21:33 pts/0    00:00:00 ps -elf
0 S root        2540    2310  0  80   0 - 26982 pipe_w 21:33 pts/0    00:00:00 tail
```

各列的意义：

```
F：程序的标志位。0 表示该程序只有普通权限，4 表示具有 root 超级管理员权限，1 表示该进程被创建的时候只进行了 fork，没有进行 exec
S：进程的状态位，注意 ps 选项加了"-"的是非 BSD 风格选项，不会有"s""<""N""+"等的状态标识位
C：CPU 的百分比，注意衡量方式是时间
PRI：进程的优先级，值越小，优先级越高，越早被调度类选中运行
NI：进程的 NICE 值，值为-20 到 19，影响优先级的方式是 PRI(new)=PRI(old)+NI，所以 NI 为负数的时候，越小将导致进程优先级越高。
  ：但要注意，NICE 值只能影响非实时进程。
ADDR：进程在物理内存中哪个地方。
SZ：进程占用的实际物理内存
WCHAN：若进程处于睡眠状态，将显示其对应内核线程的名称，若进程为 R 状态，则显示"-"
```

10.2.3 ps 后 grep 问题

在 ps 后加上 grep 筛选目标进程时，总会发现 grep 自身进程也被显示出来。先解释下为何会如此。

```
[root@xuexi ~]# ps aux | grep "crond"
root       1425  0.0  0.1 117332  1276 ?        Ss   Jun10   0:00 crond
root       8275  0.0  0.0 103256   856 pts/2    S+   17:07   0:00 grep crond
```

先解释下为何会如此。管道是 bash 创建的，bash 创建管道后 fork 两个子进程，然后两子进程各自 exec 加载 ps 程序和 grep 程序，exec 之后这两个子进程就称为 ps 进程和 grep 进程，所以 ps 和 grep 进程几乎可以认为是同时出现的，尽管 ps 进程作为管道的首进程（进程组首进程）它是先出现的，但是在 ps 出现之前确实两个进程都已经 fork 完成了。也就是说，管道左右两端的进程是同时被创建的(不考虑父进程创建进程消耗的那点时间)，但数据传输是有先后顺序的，左边先传，右边后收。

要将 grep 自身进程排除在结果之外，方法有二：

```
[root@xuexi ~]# ps aux | grep "crond" | grep -v "grep"   # 使用-v 将 grep 自己筛选掉
root       1425  0.0  0.1 117332  1276 ?        Ss   Jun10   0:00 crond
[root@xuexi ~]# ps aux | grep "cron[d]"
root       1425  0.0  0.1 117332  1276 ?        Ss   Jun10   0:00 crond
```

第二种方法能成功是因为 grep 进程被 ps 捕获时的结果是"grep cron[d]"，而使用 cron[d]匹配时，它将只能匹配 crond，所以"grep cron[d]"被筛选掉了。其实加上其他字符将更容易理解。

```
[root@xuexi ~]# ps aux | grep "cron[dabc]"
root       1425  0.0  0.1 117332  1276 ?        Ss   Jun10   0:00 crond
```

10.2.4 uptime 命令

```
[root@xuexi ~]# uptime
 08:38:11 up 22:35,  2 users,  load average: 0.00, 0.01, 0.05
```

显示当前时间，已开机运行多少时间，当前有多少用户已登录系统，以及 3 个平均负载值。

所谓负载率(load)，即特定时间长度内，cpu 运行队列中的平均进程数(包括线程)，一般平均每分钟每核的进程数小于 3 都认为正常，大于 5 时负载已经非常高。在 UNIX 系统中，运行队列包括 cpu 正在执行的进程和等待 cpu 的进程(即所谓的可运行 runable)。在 Linux 系统中，还包括不可中断睡眠态(IO 等待)的进程。运行队列中每出现一个进程，load 就加 1，进程每退出运行队列，Load 就减 1。如果是多核 cpu，则还要除以核数。

详细信息见 man uptime 和 https://en.wikipedia.org/wiki/Load_(computing)

例如，单核 cpu 上的负载值为"1.73 0.60 7.98"时，表示：

最近 1 分钟：1.73 表示平均可运行的进程数，这一分钟要一直不断地执行这 1.73 个进程。0.73 个进程等待该核 cpu。

最近 5 分钟：平均进程数还不足 1，表示该核 cpu 在过去 5 分钟空闲了 40%的时间。

最近 15 分钟：7.98 表示平均可运行的进程数，这 15 分钟要一直不断地执行这 7.98 个进程。

结合前 5 分钟的结果，说明前 15-前 10 分钟时间间隔内，该核 cpu 的负载非常高。

如果是多核 cpu，则还要将结果除以核数。例如 4 核时，某个最近一分钟的负载值为 3.73，则意味着有 3.73 个进程在运行队列中，这些进程可被调度至 4 核中的任何一个核上运行。最近 1 分钟的负载值为 1.6，表示这一分钟内每核 cpu 都空闲(1-1.6/4)=60%的时间。

所以，load 的理想值是正好等于 CPU 的核数，小于核数的时候表示 cpu 有空闲，超出核数的时候表示有进程在等待 cpu，即系统资源不足。

10.2.5 top、htop 以及 iftop 命令

top 命令查看动态进程状态，默认每 5 秒刷新一次。

```
top 选项说明：
-d：指定 top 刷新的时间间隔，默认是 5 秒
-b：批处理模式，每次刷新分批显示
-n：指定 top 刷新几次就退出，可以配合-b 使用
-p：指定监控的 pid，指定方式为-pN1 -pN2 ...或-pN1, N2 [,...]
-u：指定要监控的用户的进程，可以是 uid 也可以是 user_name
```

在 top 动态模式下，按下各种键可以进行不同操作。使用"h"或"?"可以查看相关键的说明。

```
1     ：(数字一)表示是否要在 top 的头部显示出多个 cpu 信息
H     ：表示是否要显示线程，默认不显示
c,S   ：  c 表示是否要展开进程的命令行，S 表示显示的 cpu 时间是否是累积模式，cpu 累积模式下已死去的子进程 cpu 时间会累积到父进程中
x,y   ：x 高亮排序的列，y 表示高亮 running 进程
u     ：仅显示指定用户的进程
n or #：设置要显示最大的进程数量
k     ：杀进程
q     ：退出 top
P ：以 CPU 的使用资源排序显示
M ：以 Memory 的使用资源排序显示
N ：以 PID 来排序
```

以下是 top 的一次结果。

```
[root@xuexi ~]# top
top - 17:43:44 up 1 day, 14:16,  2 users,  load average: 0.10, 0.06, 0.01
Tasks: 156 total,   1 running, 155 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  0.0%us,  0.0%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.3%si,  0.0%st
Cpu2  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  :  0.3%us,  0.0%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1004348k total,   417928k used,   586420k free,    52340k buffers
Swap:  2047996k total,        0k used,  2047996k free,   243800k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    1 root      20   0 19364 1444 1132 S  0.0  0.1   0:00.96 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:01.28 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:00.59 ksoftirqd/0
    5 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 stopper/0
```

```
第 1 行：和 w 命令的第一行一样，也和 uptime 命令的结果一样。
第 2 行：分别表示总进程数、running 状态的进程数、睡眠状态的进程数、停止状态进程数、僵尸进程数。
第 3-6 行：每颗 cpu 的状况。
    us = user mode
    sy = system mode
    ni = low priority user mode (nice)(用户空间中低优先级进程的 cpu 占用百分比)
    id = idle task
    wa = I/O waiting
    hi = servicing IRQs(不可中断睡眠，hard interruptible)
    si = servicing soft IRQs(可中断睡眠，soft interruptible)
    st = steal (time given to other DomU instances)(被偷走的 cpu 时间，一般被虚拟化软件偷走)
第 7-8 行：从字面意思理解即可。

VIRT：虚拟内存总量
```

RES：实际内存总量
SHR：共享内存量
TIME：进程占用的 cpu 时间(若开启了时间累积模式，则此处显示的是累积时间)

top 命令虽然非常强大，但是太老了。所以有了新生代的 top 命令 htop。htop 默认没有安装，需要手动安装。

```
[root@xuexi ~]# yum -y install htop
```

htop 可以使用鼠标完成点击选中。其他使用方法和 top 类似，使用 h 查看各按键意义即可。

```
CPU usage bar: [low-priority/normal/kernel/virtualiz                    used%]
Memory bar:    [used/buffers/cache                                  used/total]
Swap bar:      [used                                                used/total]
Type and layout of header meters are configurable in the setup screen.

 Status: R: running; S: sleeping; T: traced/stopped; Z: zombie; D: disk sleep
 Arrows: scroll process list                Space: tag process
 Digits: incremental PID search                 c: tag process and its children
   F3 /: incremental name search                U: untag all processes
   F4 \: incremental name filtering          F9 k: kill process/tagged processes
   F5 t: tree view                           F7 ]: higher priority (root only)
      u: show processes of a single user     F8 [: lower priority (+ nice)
      H: hide/show user threads                 a: set CPU affinity
      K: hide/show kernel threads               i: set IO prority
      F: cursor follows process                 l: list open files with lsof
F6 + -: expand/collapse tree                    s: trace syscalls with strace
 P M T: sort by CPU%, MEM% or TIME
      I: invert sort order                  F2 S: setup
   F6 >: select sort column                 F1 h: show this help screen
                                           F10 q: quit
```

iftop 用于动态显示网络接口的数据流量。用法也很简单，按下 h 键即可获取帮助。

10.2.6 分析系统负载(system load average)

根据前文 uptime 中对系统负载(system load)的描述，分析一下这个 top 的结果。

```
top - 16:13:01 up 163 days, 17:59,  1 user,  load average: 383.19, 210.66, 87.46
Threads: 1664 total,   1 running, 1663 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.0 us,  0.2 sy,  0.0 ni, 98.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 32780452 total,   290280 free, 11104252 used, 21385920 buff/cache
KiB Swap:  4194300 total,  3891196 free,   303104 used. 21054300 avail Mem
```

上图中，系统负载非常之高，最近一分钟的负载量高达 383.19，这表示这一分钟有 383.19 个进程正在运行或等待调度，如果是单核 CPU，表示这一分钟要毫不停留地执行这么多进程，如果是 8 核 CPU，表示这一分钟内平均每核心 CPU 要执行大概 50 个进程。

从 load average 上看，确实是非常繁忙的场景。但是看 CPU 的 idle 值为 98.8，说明 CPU 非常闲。为什么系统负载如此高，CPU 却如此闲？

前面解释 system load average 的时候，已经说明过可运行的(就绪态，即就绪队列的长度)、正在运行的(运行态)和不可中断睡眠(如 IO 等待)的进程任务都会计算到负载中。现在**负载高、CPU 空闲，说明当前正在执行的任务基本不消耗 CPU 资源，大量的负载进程都在 IO 等待中**。

可以从 ps 的进程状态中获取哪些进程是正在运行或运行队列中的(状态为 R)，哪些进程是在不可中断睡眠中的(状态为 D)。

```
[root@xuexi src]# ps -eo stat,pid,ppid,comm --no-header |grep -E "^(D|R)"
R+     11864    9624 ps
```

10.3 vmstat 命令

注意 vmstat 的第一次统计是自开机起的平均值信息，从第二次开始的统计才是指定刷新时间间隔内的资源利用信息，若不指定刷新时间间隔，则默认只显示一次统计信息。

```
vmstat [-d] [delay [ count]]
vmstat [-f]
选项说明：
-f：统计自开机起 fork 的次数。包括 fork、clone、vfork 的次数。但不包括 exec 次数。
-d：显示磁盘统计信息。
delay：刷新时间间隔，若不指定，则只统计一次信息就退出 vmstat。
count：总共要统计的次数。
```

例如，只统计一次信息。

```
[root@xuexi ~]# vmstat
```

```
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 583692  52684 244200    0    0     5     3    4    5  0  0 100  0  0
```

其中各列的意义如下：

```
Procs
   r: 等待队列中的进程数
   b: 不可中断睡眠的进程数
Memory
   swpd: 虚拟内存使用总量
   free: 空闲内存量
   buff: buffer 占用的内存量(buffer 用于缓冲)
   cache: cache 占用的内存量(cache 用于缓存)
Swap
   si:从磁盘加载到 swap 分区的数据流量，单位为"kb/s"
   so: 从 swap 分区写到磁盘的数据流量，单位为"kb/s"
IO
   bi: 从块设备接受到数据的速率，单位为blocks/s
   bo: 发送数据到块设备的速率，单位为 blocks/s
System
   in: 每秒中断数，包括时钟中断数量
   cs: 每秒上下文切换次数
CPU：统计的是 cpu 时间百分比，具体信息和 top 的 cpu 统计列一样
   us: Time spent running non-kernel code. (user time, including nice time)
   sy: Time spent running kernel code. (system time)
   id: Time spent idle. Prior to Linux 2.5.41, this includes IO-wait time.
   wa: Time spent waiting for IO. Prior to Linux 2.5.41, included in idle.
   st: Time stolen from a virtual machine. Prior to Linux 2.6.11, unknown.
```

还可以统计磁盘的 IO 信息。统计信息的结果很容易看懂，所以略过。

10.4 iostat 命令

iostat 主要统计磁盘或分区的整体使用情况。也可以输出 cpu 信息，甚至是 NFS 网络文件系统的信息。同 vmstat/sar 一样，第一次统计的都是自系统开机起的平均统计信息。

```
iostat [ -c ] [ -d ] [ -n -h ][ -k | -m ] [ -p [device][,...]] [ interval [ count ] ]
选项说明：
-c：统计 cpu 信息
-d：统计磁盘信息
-n：统计 NFS 文件系统信息
-h：使 NFS 统计信息更人类可读化
-k：指定以 kb/s 为单位显示
-m：指定以 mb/s 为单位显示
-p：指定要统计的设备名称
-y：指定不显示第一次统计信息，即不显示自开机起的统计信息。
interval：刷新时间间隔
count：总统计次数
```

例如：

```
[root@xuexi ~]# iostat
Linux 2.6.32-504.el6.x86_64 (xuexi.longshuai.com)       06/11/2017      _x86_64_        (4 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.01    0.00    0.03    0.01    0.00   99.96

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda               0.58        39.44        23.14    5557194    3259968
sdb               0.00         0.03         0.00       4256          0
```

各列的意义都很清晰，从字面即可理解。

```
tps：每秒 transfer 速率(transfers per second),一次对物理设备的 IO 请求为一个 transfer，但多个逻辑请求可能只组成一个 transfer
Blk_read/s：每秒读取的 block 数量
Blk_wrtn/s：每秒写入的 block 总数
Blk_read：读取的总 block 数量
Blk_wrtn：写入的总 block 数量
```

10.5 sar 命令

sar 是一个非常强大的性能分析工具，它可以获取系统的 cpu/等待队列/磁盘 IO/内存/网络等性能指标。

功能多的必然结果是选项多，应用复杂，但只要知道一些常用的选项足以。

```
sar [options] [-o filename] [delay [count] ]
选项说明：
-A：显示系统所有资源运行状况
-b：显示磁盘 IO 和 tranfer 速率信息，和 iostat 的信息一样，是总体 IO 统计信息
-d：显示磁盘在刷新时间间隔内的活跃情况，可以指定一个或多个设备，和-b 不同的是，它显示的是单设备的 IO、transfer 信息。
  ：建议配合-p 使用显示友好的设备名，否则默认显示带主次设备号的设备名
-P：显示指定的某颗或某几颗 cpu 的使用情况。指定方式为，-P 0,1,2,3 或 ALL。
-u：显示每颗 cpu 整体平均使用情况。-u 和-P 的区别通过下面的示例很容易区分。
-r：显示内存在刷新时间间隔内的使用情况
-n：显示网络运行状态。后可接 DEV/NFS/NFSD/ALL 等多种参数。
  ：DEV 表示显示网路接口信息，NFS 和 NFSD 分别表示显示 NFS 客户端服务端的流量信息，ALL 表示显示所有信息。
-q：显示等待队列大小
-o filename：将结果存入到文件中
delay：状态刷新时间间隔
count：总共刷新几次
```

10.5.1 统计 cpu 使用情况

```
[root@server2 ~]# sar -P ALL 1 2
Linux 3.10.0-327.el7.x86_64 (server2.longshuai.com)     06/20/2017      _x86_64_        (4 CPU)

01:18:49 AM     CPU     %user     %nice   %system   %iowait    %steal     %idle
01:18:50 AM     all      0.00      0.00      0.25      0.00      0.00     99.75
01:18:50 AM       0      0.00      0.00      0.00      0.00      0.00    100.00
01:18:50 AM       1      0.00      0.00      0.00      0.00      0.00    100.00
01:18:50 AM       2      0.00      0.00      0.00      0.00      0.00    100.00
01:18:50 AM       3      0.00      0.00      0.00      0.00      0.00    100.00

01:18:50 AM     CPU     %user     %nice   %system   %iowait    %steal     %idle
01:18:51 AM     all      0.00      0.00      0.00      0.00      0.00    100.00
01:18:51 AM       0      0.00      0.00      0.00      0.00      0.00    100.00
01:18:51 AM       1      0.00      0.00      0.99      0.00      0.00     99.01
01:18:51 AM       2      0.00      0.00      0.00      0.00      0.00    100.00
01:18:51 AM       3      0.00      0.00      0.00      0.00      0.00    100.00

Average:        CPU     %user     %nice   %system   %iowait    %steal     %idle
Average:        all      0.00      0.00      0.12      0.00      0.00     99.88
Average:          0      0.00      0.00      0.00      0.00      0.00    100.00
Average:          1      0.00      0.00      0.50      0.00      0.00     99.50
Average:          2      0.00      0.00      0.00      0.00      0.00    100.00
Average:          3      0.00      0.00      0.00      0.00      0.00    100.00
```

各列的意义就不再赘述了，在前面几个信息查看命令已经解释过多次了。

在上面的例子中，统计了所有 cpu(0,1,2,3 共 4 颗)每秒的状态信息，每秒还进行了一次汇总，即 all，最后还对每颗 cpu 和汇总 all 计算了平均值。而我们真正需要关注的是最后的 average 部分的 idle 值，idle 越小，说明 cpu 处于空闲时间越少，该颗或整体 cpu 使用率就越高。

或者直接对整体进行统计。如下：

```
[root@server2 ~]# sar -u 1 2
Linux 3.10.0-327.el7.x86_64 (server2.longshuai.com)     06/20/2017      _x86_64_        (4 CPU)

01:18:37 AM     CPU     %user     %nice   %system   %iowait    %steal     %idle
01:18:39 AM     all      0.00      0.00      0.00      0.00      0.00    100.00
01:18:40 AM     all      0.00      0.00      0.23      0.00      0.00     99.77
Average:        all      0.00      0.00      0.12      0.00      0.00     99.88
```

10.5.2 统计内存使用情况

```
[root@server2 ~]# sar -r 1 2
Linux 3.10.0-327.el7.x86_64 (server2.longshuai.com)     06/20/2017      _x86_64_        (4 CPU)

01:49:04 AM kbmemfree kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit  kbactive   kbinact   kbdirty
01:49:05 AM   1315968    552720     29.58       932    319888    225164      5.75    282760     85740         0
01:49:06 AM   1315984    552704     29.58       932    319888    225164      5.75    282760     85740         0
```

```
Average:      1315976    552712     29.58       932    319888    225164      5.75    282760     85740          0
```

其中 kbdirty 表示内存中脏页的大小，即内存中还有多少应该刷新到磁盘的数据。

10.5.3 统计网络流量

第一种方法是查看/proc/net/dev 文件。

```
[root@server2 ~]# cat /proc/net/dev
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo colls carrier compressed
  eth0:  209644    1834    0    0    0     0          0         0   981664    1679    0    0    0     0       0          0
    lo:     340       4    0    0    0     0          0         0      340       4    0    0    0     0       0          0
```

关注列：receive 和 transmit 分别表示收包和发包，关注每个网卡的 bytes 即可获得网卡的情况。写一个脚本计算每秒的差值即为网络流量。

或者使用 sar -n 命令统计网卡接口的数据。

```
[root@server2 ~]# sar -n DEV 1 2
Linux 3.10.0-327.el7.x86_64 (server2.longshuai.com)     06/20/2017      _x86_64_        (4 CPU)

01:51:11 AM     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s
01:51:12 AM      eth0      0.00      0.00      0.00      0.00      0.00      0.00      0.00
01:51:12 AM        lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00

01:51:12 AM     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s
01:51:13 AM      eth0      0.99      0.99      0.06      0.41      0.00      0.00      0.00
01:51:13 AM        lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00

Average:        IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s
Average:         eth0      0.50      0.50      0.03      0.21      0.00      0.00      0.00
Average:           lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00
```

各列的意义如下：

rxpck/s：每秒收到的包数量

txpck/s：每秒发送的包数量

rxkB/s：每秒收到的数据，单位为 kb

txkB/s：每秒发送的数据，单位为 kb

rxcmp/s：每秒收到的压缩后的包数量

txcmp/s：每秒发送的压缩后的包数量

rxmcst/s：每秒收到的多播包数量

10.5.4 查看队列情况

```
[root@server2 ~]# sar -q
Linux 3.10.0-327.el7.x86_64 (server2.longshuai.com)     06/20/2017      _x86_64_        (4 CPU)

12:00:01 AM   runq-sz  plist-sz   ldavg-1   ldavg-5  ldavg-15   blocked
12:10:01 AM         0       446      0.01      0.02      0.05         0
12:20:01 AM         0       445      0.02      0.03      0.05         0
12:30:01 AM         0       446      0.00      0.01      0.05         0
Average:            0       446      0.01      0.02      0.05         0
```

每列意义解释：

runq-sz：等待队列的长度，不包括正在运行的进程

plist-sz：任务列表中的进程数量，即总任务数

ldavg-N：过去 1 分钟、5 分钟、15 分钟内系统的平均哎

blocked：当前因为 IO 等待被阻塞的任务数量

10.5.5 统计磁盘 IO 情况

```
[root@server2 ~]# sar -d -p 1 2
Linux 3.10.0-327.el7.x86_64 (server2.longshuai.com)     06/20/2017      _x86_64_        (4 CPU)

12:53:06 AM       DEV       tps  rd_sec/s  wr_sec/s  avgrq-sz  avgqu-sz     await     svctm     %util
```

```
12:53:07 AM       sda      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00

12:53:07 AM       DEV       tps  rd_sec/s  wr_sec/s  avgrq-sz  avgqu-sz     await     svctm     %util
12:53:08 AM       sda      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00

Average:          DEV       tps  rd_sec/s  wr_sec/s  avgrq-sz  avgqu-sz     await     svctm     %util
Average:          sda      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
```

分别统计的是 12:53:06 到 12:53:07 和 12:53:07 到 12:53:08 这两秒的 IO 使用情况。

各列的意义如下：

tps：transfer per second，每秒的 transfer 速率，一次物理 IO 请求算一次 transfer，但多次逻辑 IO 请求可能组合起来才算一次 transfer。

rd_sec/s：每秒读取的扇区数，扇区大小为 512 字节。

wr_sec/s：每秒写入的扇区数。

avgrq-sz：请求写入设备的平均大小，单位为扇区。(The average size (in sectors) of the requests that were issued to the device)

avgqu-sz：请求写入设备的平均队列长度。(The average queue length of the requests that were issued to the device.)

await：写入设备的 IO 请求的平均(消耗)时间，单位微秒(The average time for I/O requests issued to the device to be served.)

svctm：不可信的列，该列未来将被移除，所以不用管

%util：最重要的一列，显示的是设备的带宽情况。该列若接近 100%，说明磁盘速率饱和了。

10.6 free

free 用于查看内存使用情况。CentOS 6 和 CentOS 7 上显示格式不太一样。

```
free [options]
选项说明：
-h：人类可读方式显式单位
-m：以 MB 为显示单位
-w：将 buffers 和 cache 分开单独显示。只对 CentOS 7 上有效
-s：动态查看内存信息时的刷新时间间隔
-c：一共要刷新多少次退出 free
```

以下以 CentOS 7 上的 free 结果说明各列的意义。

```
[root@server2 ~]# free -m
              total        used        free      shared  buff/cache   available
Mem:           1824         131        1286           8         407        1511
Swap:          1999           0        1999
```

Mem 和 Swap 分别表示物理内存和交换分区的使用情况。

```
total：总内存空间
used：已使用的内存空间。该值是 total-free-buffers-cache 的结果
free：未使用的内存空间
shared：/tmpfs 总用的内存空间。对内核版本有要求，若版本不够，则显示为 0。
buff/cache：buffers 和 cache 的总占用空间
available：可用的内存空间。即程序启动时，将认为可用空间有这么多。
```

所以 available 才是真正需要关注的可使用内存空间量。

使用-w 可以将 buffers/cache 分开显示。

```
[root@server2 ~]# free -w -m
              total        used        free      shared     buffers       cache   available
Mem:           1824         131        1286           8           0         406        1511
Swap:          1999           0        1999
```

还可以动态统计内存信息，例如每秒统计一次，统计 2 次。

```
[root@server2 ~]# free -w -m -s 1 -c 2
              total        used        free      shared     buffers       cache   available
Mem:           1824         130        1287           8           0         406        1512
Swap:          1999           0        1999

              total        used        free      shared     buffers       cache   available
Mem:           1824         130        1287           8           0         406        1512
Swap:          1999           0        1999
```

以下是 CentOS 6 上的 free 结果。

```
[root@xuexi ~]# free -m
             total       used       free     shared    buffers     cached
Mem:           980        415        565          0         53        239
-/+ buffers/cache:        121        859
Swap:         1999          0       1999
```

在此结果中，"-/+ buffers/cache"的 free 列才是真正可用的内存空间了，即 CentOS 7 上的 available 列。

一般来说，内存可用量的范围低于 20%应该要引起注意了。

第11章 服务管理

CentOS 7 和 CentOS 6 管理服务的方式完全不同。本文先说明 CentOS 6 上的管理方式，在最后列出 CentOS 7 上服务管理方式。

11.1 服务

服务是向外提供服务的进程，一般来说都会放在后台，既然要持续不断的提供外界随时发来的服务请求，服务进程就需要常驻在内存中，且不应该和终端有关，否则终端退出服务程序就退出了。另外，要能够接待外界的请求为外界提供服务，那么就需要有个专属于这个服务的"服务窗口"，这个服务窗口就是端口号，通过端口号就能找到服务的提供者。

提供服务的一端叫做服务端，向服务端请求服务的叫做客户端。首先，服务端启动服务进程，此时将开放对应的端口号；然后客户端指定服务端 IP 地址和端口号向该服务端发起请求，服务端所在主机的内核接收到请求数据包，然后分析数据包发现请求的是某某端口号，内核知道该端口号是哪个应用程序监听的端口，所以将请求报文发送给对应的应用程序，应用程序收到报文后，将和客户端建立连接，并进行数据传输。

另外需要注意的是，并非所有服务都总是提供端口号的，例如 xinetd 这个服务，只有在需要的时候才接管相应的端口，如 rsync 监听端口为 222 时，那么请求 rsync 时，xinetd 在监听过程中的端口号就是 222。在不被请求的时候，xinetd 是没有端口号的。

在 Linux 中，服务分为独立守护进程和超级守护进程。独立守护进程是自行监听在后台的，基本上所有的服务都是独立守护进程类的服务。超级守护进程专指 xinetd 这个服务，这个服务代为管理着一些特殊的服务，这类服务在被请求的时候才会由 xinetd 通知它启动服务，服务提供完毕后就关闭服务，这类服务称为瞬时守护进程，即只存在于瞬时。

但要明白，超级守护进程 xinetd 本身是一个常驻内存的独立守护进程，因为它要监听来自外界对其管理的瞬时守护进程的请求。只不过一般不工作的时候，xinetd 不占用端口号，在工作的时候它占用被请求的瞬时守护进程的端口号，并处于监听状态。

11.2 管理独立守护进程

在 CentOS 6 上，所有的服务脚本都在/etc/rc.d/init.d/目录下，/etc/init.d/是它的软链接。在此目录下的脚本都是 LSB 风格的脚本，它们基本上都能接受 start/stop/restart/reload/status 等参数。

```
[root@xuexi tmp]# ls /etc/init.d
abrt-ccpp         cpuspeed   irqbalance    messagebus  psacct       saslauthd
abrtd             crond      kdump         netconsole  quota_nld    single
abrt-oops         functions  killall       netfs       rdisc        smartd
acpid             haldaemon  lvm2-lvmetad  network     restorecond  sshd
atd               halt       lvm2-monitor  ntpd        rngd         svnserve
auditd            ip6tables  mcelogd       ntpdate     rsyslog      sysstat
blk-availability  iptables   mdmonitor     postfix     sandbox      udev-post
```

要管理独立守护进程类的服务

```
/etc/init.d/service_name   restart|start|stop|status    # 方法一
service  service_name    restart|start|stop|status     # 方法二
```

要让服务能够被 service 命令管理，将其服务脚本放在/etc/init.d 目录下即可。

11.3 管理服务的开机自启动

chkconfig 命令能管理/etc/init.d/目录下存在且脚本的内容满足一定条件的服务。

要能让 chkconfig 管理服务的开机是否自启动行为，只需将脚本放在/etc/init.d 目录下，然后在脚本的前部加上 chkconfig 行和 description 行。如：

```
#!/bin/bash

# chkconfig: - 85 15
# description: The Apache HTTP Server is an efficient and extensible
```

这两行必须在所有非注释行的前面，且这两行必须得被"注释"。其中 chkconfig 行"-"表示适用于运行级别 123456 上，85 表示开机启动时，它的启动顺序为 85，15 表示关机停止服务时，它的停止顺序为 15。description 行随便给一点描述信息就可以，但是必须得给"description:"关键字。

然后，就可以有 chkconfig 来管理服务的开机自启动了。

```
chkconfig [--add | --del] <name>  # 将/etc/init.d 中可以被 chkconfig 管理的服务添加到 chkconfig 的管理列表中，或者从列表中删除
chkconfig [--list] [name]         # 列出指定名称的服务的开启自启动信息。name 可以使用 all 来表示列出所有 chkconfig 管理列表中的服务
chkconfig [--level <levels>] <name> <on|off|reset>  # 将指定名称的服务在指定级别上打开开机自启动或关闭开机自启动功能。
                                                    # reset 则表示重置为脚本中指定的级别
```

当然，除了 chkconfig 可以管理开机自启动，将启动命令放在/etc/rc.d/rc.local 文件中也是可以的。

11.4 管理 xinetd 及相关瞬时守护进程

11.4.1 管理瞬时守护进程

该类服务不能直接使用 service 命令来启动。只能去/etc/xinetd.d/目录下的对应文件中进行设置(当然，也可以在/etc/xinetd.conf 中配置)，然后由 xinetd 进行管理。

首先安装 xinetd 程序。

```
[root@xuexi tmp]# yum -y install xinetd
[root@xuexi tmp]# chkconfig --list
```

```
......省略
xinetd          0:off   1:off   2:off   3:on    4:on    5:on    6:off

xinetd based services:
        chargen-dgram:  off
        chargen-stream: off
        daytime-dgram:  off
        daytime-stream: off
        discard-dgram:  off
        discard-stream: off
        echo-dgram:     off
        echo-stream:    off
        rsync:          off
        tcpmux-server:  off
        time-dgram:     off
        time-stream:    off
```

首先得保证 xinetd 是已经工作在后台的。

```
service xinetd start
```

然后管理瞬时守护进程，该类服务比较特别，其自启动状态和服务运行状态是同步的，也就是说 chkconfig 设置了其自启动则表示启动该服务，否则为停止该服务。另外，对其指定级别是无效的，它们的启动级别继承与 xinetd 的启动级别，并且 xinetd 会接管其触发的瞬时守护进程的端口号。

例如启动 rsync 这个瞬时守护进程。

```
chkconfig rsync on
```

11.4.2 瞬时守护进程的配置

瞬时守护进程受两个配置文件控制，一个是 xinetd 的配置文件/etc/xinetd.conf 提供默认配置，一个是/etc/xinetd.d/下的配置文件针对对应的服务提供配置。

例如配置 rsync，以下是/etc/xinetd.d/rsync 的默认配置。

```
[root@xuexi tmp]# vi /etc/xinetd.d/rsync
```

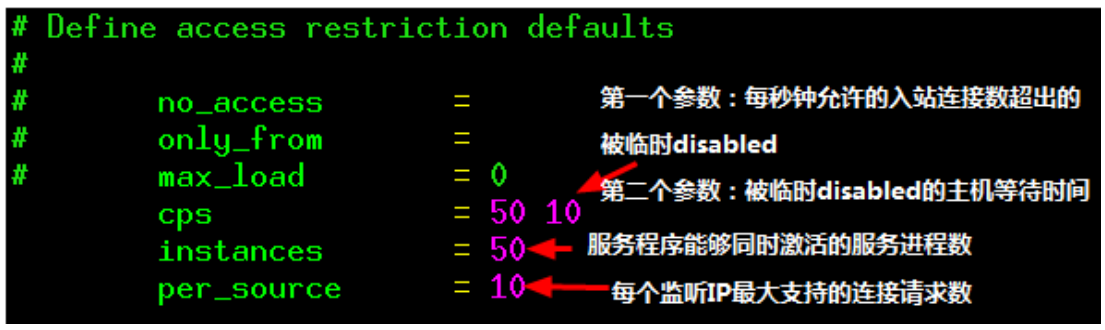
```
# default: off
# description: The rsync server is a good addition to an ftp server, as it \
#       allows crc checksumming etc.
service rsync       # 定义 rsync 服务，名称要和/etc/xinetd.d/下的文件同名
{
        disable         = yes      # yes 表示不启动，no 表示启动，等价于 chkconfig rsync {on|off}，所以这里设置后将直接在 chkconfig 中生效
        flags           = IPv6     # 不用管
        socket_type     = stream   # 这代表的是 tcp 类型的套接字
        wait            = no       # 该服务是单线程还是多线程的，表现形式是超出的请求是否进行等待，no 表示多线程
        user            = root     # 以什么身份运行 rsync
        server          = /usr/bin/rsync # 服务程序
        server_args     = --daemon # 服务程序启动时传递的参数
        log_on_failure  += USERID  # 连接失败的日志记录，+表示在全局对应的条目上新增此处指定的 USERID
}
```

除此之外，还有几个选项：

```
【访问控制选项以下两个控制列表中最好不要出现冲突的地址。
only_from:定义允许连接的访问控制列表，支持单 IP，CIDR 格式和长 mask 格式的网段，主机名 hostname，域 DOMAIN（.abc.com）
no_access:定义不允许访问的列表，语法格式同 only_from
【监听地址】
bind      = ip_addr
interface  = ip_addr  # 等价于 bind
【资源控制】
cps=args1 args2
```

```
instances=N
per_source=N
```

这 3 个选项的意义如下图。



11.5 CentOS 7 上管理服务

```
service name start   ==> systemctl start name.service
service name stop    ==> systemctl stop name.service
service name restart ==> systemctl restart name.service
service name status  ==> systemctl status name.service

查看服务是否激活(在运行中)：systemctl is-active name.service
查看所有已经激活          ：systemctl list-units --type service
查看所有服务              ：systemctl list-units --type service --all

设置开机自启动：chkconfig name on ==> systemctl enable name.service
禁止开机自启动：chkconfig name off ==> systemctl disable name.service
查看服务是否开机自启动：chkconfig --list name ==> is-enabled name.service
查看所有服务的开机自启动状态：chkconfig --list ==> systemctl list-unit-files --type service
```

第12章 定时任务

12.1 配置定时任务

一个不错的定时任务在线测试网站：http://www.atool9.com/crontab.php

关于定时任务，首先需弄清的概念：

(1).crond 是一个 daemon 类程序，路径为/usr/sbin/crond。默认会以后台方式启动，service 或 systemd 方式启动 crond 默认也是后台方式的。

(2).crondtab 是管理 crontab file 的工具，而 crontab file 是定义定时任务条目的文件。

(3).crontab file 存在于多处，包括系统定时任务文件/etc/crontab 和/etc/cron.d/*，还有独属于各用户的任务文件/var/spool/cron/USERNAME。

再就是 crontab 命令：

```
-l：列出定时任务条目
-r：删除当前任务列表终端所有任务条目
-i：删除条目时提示是否真的要删除
-e：编辑定时任务文件，实际上编辑的是/var/spool/cron/*文件
-u：操作指定用户的定时任务
```

执行 crontab -e 命令编辑当前用户的 crontab file，例如当前为 root 用户，则编辑的是/var/spool/cron/root 文件。例如写入下面这一行。

```
* * * * * /bin/echo "the first cron entry"  >>/tmp/crond.txt
```

这将会每分钟执行一次 echo 命令，将内容追加到/tmp/crond.txt 文件中。

任务计划中的任务条目如何定义，可以查看/etc/crontab 文件。

```
[root@server2 ~]# cat /etc/crontab
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root

# For details see man 4 crontabs

# Example of job definition:
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,wed,thu,fri,sat
# |  |  |  |  |
# *  *  *  *  * user-name  command to be executed
```

在此文件中定义了 3 个变量，其中一个是 PATH，该变量极其重要。在最后还给出了任务条目的定义方式：

(1).每个任务条目分为 6 段，每段以空格分隔，之所以此处多了 user-name 段是因为/etc/crontab 为系统定时任务文件，而一般定时任务是没有该段的。

(2).前五段为时间的设定段，分别表示"分时日月周"，它们的定义不能超出合理值范围，第六段为所要执行的命令或脚本任务段。

(3).在时间定义段中，使用"*"表示每单位，即每分钟，每小时，每天，每月，每周几(仍然是每天)。实际上，按 man 文档中解释，"*"表示的是从每个时间段的起始到结尾，也就是全部时间单位的意思。例如在小时上设置*，表示 0,1,2,3...22,23 的意思。

(4).每个时间段中，都可以使用逗号","来表示枚举，例如定义"0,30,50 * * * *"表示每个时辰的整点、第 30 分钟和第 50 分钟都执行该任务。

(5).每个时间段中，都可以使用"-"定义范围，可以结合逗号使用。如分钟段定义了"00,20-30,50"表示每个时辰的整点、第 20 到 30 分钟的每分钟、第 50 分钟都执行该任务。

(6).每个时间段中，使用"/"表示忽略时间，如在小时段定义了"0-13/2"表示在"0/2/4/6/8/10/12"点才满足时间定义。常使用"*/N"表示每隔多久的意思。例如"00 */2 * * *"表示在每天每隔两小时的整点执行该任务(严格地说是 0-23/2，也就是 0,2,4,...,22，所以凌晨 1 点不会执行任务)。

(7).如果定义的日和周冲突了，则会多次执行(不包括因为*号导致的冲突)。例如每月的 15 号执行该任务，同时又定义了周三执行该任务，正常无冲突情况下，将在周三和每月 15 号执行，但如果某月的 15 号同时是周三，则该任务在此日执行两次。因此，应该尽力避免同时定义周和日的任务。

(8).命令段(即第 6 段)中，不能随意出现百分号"%"，因为它表示换行的特殊意义，且第一个%后的所有字符串将当作命令的标准输入。

例如下面的定义：

```
* * * * * /bin/cat >>/tmp/crond.txt %"the first %%cron entry%"
```

该任务输出的结果将是：

```
"the first
```

```
cron entry
"
```

所以，在定时任务条目中若以时间定义文件名时，应当将%使用反斜杠转义。如：

```
* * * * * cp /etc/fstab /tmp/`date +\%Y-\%m-\%d`.txt
```

另外一个需要注意的时间段设置是，使用*号问题。例如"* */2 * * *"，它表示每隔两小时后的每一分钟都执行任务，也就是凌晨 0 点的每分钟执行任务，凌晨 1 点不执行任务，凌晨 2 点的每分钟执行任务，凌晨 4 点的每分钟执行任务，依此类推。同理，"*/5 */2 * * *"表示每隔 2 小时后的每 5 分钟执行一次任务。

12.2 crondtab file

crondtab file 为任务定义文件。

(1).在此文件中，空行会被忽略，首个非空白字符且以#开头的行为注释行，但#不能出现在行中。

(2).可以在 crontab file 中设置环境变量，方式为"name=value"，等号两边的空格可随意，即"name = value"也是允许的。但 value 中出现的空格必须使用引号包围。

(3). 默认 crond 命令启动的时候会初始化所有变量，除了某几个变量会被 crond daemon 自动设置好，其他所有变量都被设置为空值。自动设置的变量包括 SHELL=/bin/sh，以及 HOME 和 LOGNAME(在 CentOS 上则称为 USER)，后两者将被默认设置为/etc/passwd 中指定的值。其中 SHELL 和 HOME 可以被 crontab file 中自定义的变量覆盖，但 LOGNAME 不允许覆盖。当然，自行定义的变量也会被加载到内存。

(4).除了 LOGNAME/HOME/SHELL 变量之外，如果设置了发送邮件，则 crond 还会寻找 MAILTO 变量。如果设置了 MAILTO，则邮件将发送给此变量指定的地址，如果 MAILTO 定义的值为空(MAILTO="")，将不发送邮件，其他所有情况邮件都会发送给 crontab file 的所有者。

(5).在系统定时任务文件/etc/crontab 中，默认已定义 PATH 环境变量和 SHELL 环境变量，其中 PATH=/sbin:/bin:/usr/sbin:/usr/bin。

(6).crond daemon 每分钟检测一次 crontab file 看是否有任务计划条目需要执行。

12.3 crond 命令的调试

很多时候写了定时任务却发现没有执行，或者执行失败，但因为 crond 是后台运行的，有没有任何提示，很难进行排错。但是可以让 crond 运行在前端并进行调试的。

先说明下任务计划程序 crond 的默认执行方式。

使用下面三条命令启动的 crond 都是在后台运行的，且都不依赖于终端。

```
[root@xuexi ~]# systemctl start crond.service
[root@xuexi ~]# service crond start
[root@xuexi ~]# crond
```

但 crond 是允许接受选项的。

```
crond [-n] [-P] [-x flags]
-n：让 crond 以前端方式运行，即不依赖于终端。
-P：不重设环境变量 PATH，而是从父进程中继承。
-x：设置调试项，flags 是调试方式，比较有用的方式是 test 和 sch，即"-x test"和"-x sch"。
  ：其中 test 调试将不会真正的执行，sch 调试显示调度信息，可以看到等待时间。具体的见下面的示例。
```

先看看启动脚本启动 crond 的方式。

```
[root@server2 ~]# cat /lib/systemd/system/crond.service
[Unit]
Description=Command Scheduler
After=auditd.service systemd-user-sessions.service time-sync.target

[Service]
EnvironmentFile=/etc/sysconfig/crond
ExecStart=/usr/sbin/crond -n $CRONDARGS
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process

[Install]
WantedBy=multi-user.target
```

它的环境配置文件为/etc/sysconfig/crond，该文件中什么也没设置。

```
[root@server2 ~]# cat /etc/sysconfig/crond
# Settings for the CRON daemon.
# CRONDARGS= :  any extra command-line startup arguments for crond
CRONDARGS=
```

所有它的启动命令为：/usr/sbin/crond -n。但尽管此处加了"-n"选项，crond 也不会前端运行，且不会依赖于终端，这是 systemctl 决定的。

在解释下如何进行调试。以下面的任务条目为例。

```
[root@server2 ~]# crontab -e
* * * * * echo "hello world" >>/tmp/hello.txt
```

执行 crond 并带上调试选项 test。

```
[root@server2 ~]# crond -x test
debug flags enabled: test
[4903] cron started
log_it: (CRON 4903) INFO (RANDOM_DELAY will be scaled with factor 8% if used.)
log_it: (CRON 4903) INFO (running with inotify support)
log_it: (CRON 4903) INFO (@reboot jobs will be run at computer's startup.)
log_it: (root 4905) CMD (echo "hello world" >>/tmp/hello.txt )
```

执行 crond 并带上调试选项 sch。

```
[root@server2 ~]# crond -x sch
debug flags enabled: sch
[4829] cron started
log_it: (CRON 4829) INFO (RANDOM_DELAY will be scaled with factor 73% if used.)
log_it: (CRON 4829) INFO (running with inotify support)
[4829] GMToff=28800
log_it: (CRON 4829) INFO (@reboot jobs will be run at computer's startup.)
[4829] Target time=1497950880, sec-to-wait=38      # 等待 crond daemon 下一次的检测，所以表示 38 秒后 crond 将检测 crontab file
user [root:0:0:...] cmd="echo "hello world" >>/tmp/hello.txt "
[4829] Target time=1497950940, sec-to-wait=60
Minute-ly job. Recording time 1497922081
log_it: (root 4831) CMD (echo "hello world" >>/tmp/hello.txt )
user [root:0:0:...] cmd="echo "hello world" >>/tmp/hello.txt "
[4829] Target time=1497951000, sec-to-wait=60
Minute-ly job. Recording time 1497922141
log_it: (root 4833) CMD (echo "hello world" >>/tmp/hello.txt )
```

但要注意，在 sch 调试结果中的等待时间是 crond 这个 daemon 的检测时间，所以它表示等待下一次检测的时间，因此除了第一次，之后每次都是 60 秒，因为默认 crond 是每分钟检测一次 crontab file 的。例如，下面是某次的等待结果，在这几次等待检测过程中没有执行任何任务。

```
[4937] Target time=1497951720, sec-to-wait=18
[4937] Target time=1497951780, sec-to-wait=60
[4937] Target time=1497951840, sec-to-wait=60
```

还可以同时带多个调试方式，如：

```
[root@server2 ~]# crond -x test,sch
debug flags enabled: sch test
[4914] cron started
log_it: (CRON 4914) INFO (RANDOM_DELAY will be scaled with factor 21% if used.)
log_it: (CRON 4914) INFO (running with inotify support)
[4914] GMToff=28800
log_it: (CRON 4914) INFO (@reboot jobs will be run at computer's startup.)
[4914] Target time=1497951540, sec-to-wait=9
user [root:0:0:...] cmd="echo "hello world" >>/tmp/hello.txt "
[4914] Target time=1497951600, sec-to-wait=60
Minute-ly job. Recording time 1497922741
log_it: (root 4916) CMD (echo "hello world" >>/tmp/hello.txt )
```

这样在调试定时任务时间时，也不会真正执行命令。

12.4 精确到秒的任务计划

默认情况下，crond 执行的任务只能精确到分钟，无法精确到秒。但通过技巧，也是能实现秒级任务的。

(1).方法一：不太精确的方法

写一个脚本，在脚本中 sleep3 秒钟的时间，这样能实现每 3 秒执行一次命令。

```
[root@xuexi ~]# cat /tmp/a.sh
#!/bin/bash
#
PATH="$PATH:/usr/local/bin:/usr/local/sbin"
for ((i=1;i<=20;i++));do
ls /tmp
```

```
sleep 3
done
[root@xuexi ~]# cat /var/spool/cron/lisi
* * * * * /bin/bash /tmp/a.sh
```

但是这样的方法不是最佳方法，因为执行命令也需要时间，且 crond 默认会有一个随机延时，随机延时由变量 RANDOM_DELAY 定义。

(2).方法二：在 cron 配置文件中写入多条 sleep 命令和其他命令。

```
[root@xuexi ~]# cat /var/spool/cron/lisi
* * * * * ls /tmp
* * * * * sleep 3 && ls /tmp
* * * * * sleep 6 && ls /tmp
* * * * * sleep 9 && ls /tmp
* * * * * sleep 12 && ls /tmp
* * * * * sleep 15 && ls /tmp
* * * * * sleep 18 && ls /tmp
* * * * * sleep 21 && ls /tmp
* * * * * sleep 24 && ls /tmp
* * * * * sleep 27 && ls /tmp
* * * * * sleep 30 && ls /tmp
...
* * * * * sleep 57 && ls /tmp
```

这种方式很繁琐，但是更精确。如果定义到每秒级别就得写 60 行 cron 记录。

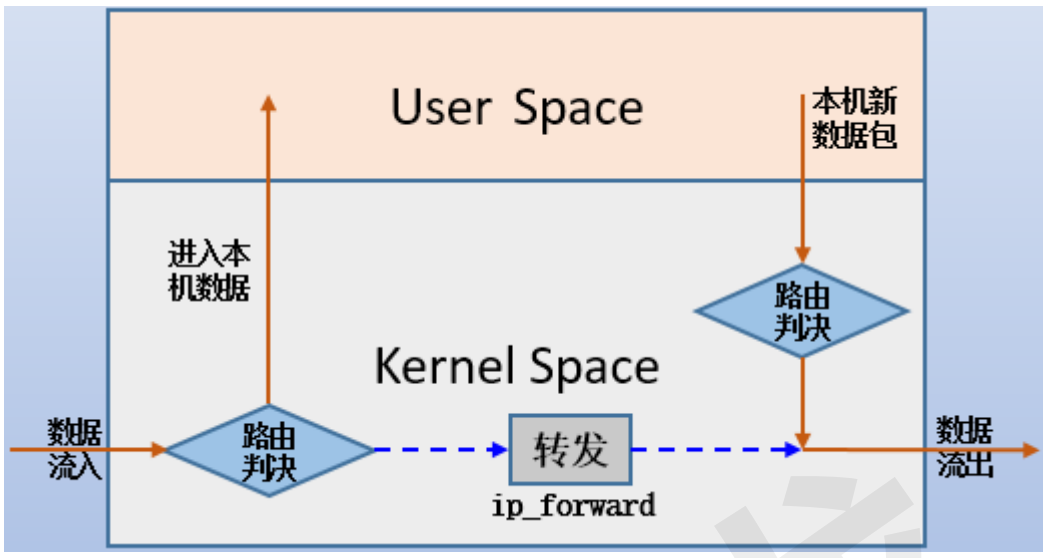
由此能看出，秒级的任务本就不是 crond 所擅长的。实际上能用到秒级的任务也比较少。

第13章 Linux 的网络管理

在解释 Linux 网络管理相关的内容时，我觉得有必要先解释下数据包转发功能以及因此涉及到的路由决策。然后再介绍网络配置命令和相关文件。

13.1 Linux 处理数据包过程

当向外界主机发送数据时，在它从网卡流入后需要对它做路由决策，根据其目标决定是流入本机数据还是转发给其他主机，如果是流入本机的数据，则数据会从内核空间进入用户空间(被应用程序接收、处理)。当用户空间响应(应用程序生成新的数据包)时，响应数据包是本机产生的新数据，在响应包流出之前，需要做路由决策，根据目标决定从哪个网卡流出。如果不是流入本机的，而是要转发给其他主机的，则必然涉及到另一个流出网卡，此时数据包必须从流入网卡完整地转发给流出网卡，这要求 Linux 主机能够完成这样的转发。但 Linux 主机默认未开启 ip_forward 功能，这使得数据包无法转发而被丢弃。Linux 主机和路由器不同，路由器本身就是为了转发数据包，所以路由器内部默认就能在不同网卡间转发数据包，而 Linux 主机默认则不能转发。如下图：



另外，IP 地址是属于内核的(不仅如此，整个 tcp/ip 协议栈都属于内核，包括端口号)，只要能和其中一个地址通信，就能和另一个地址通信(这么说是不准确的，即使地址属于内核，但还存在一个检查数据包是否丢弃的问题，不过这不是本文内容)，而不管是否开启了数据包转发功能。例如某 Linux 主机有两网卡 eth0:172.16.10.5 和 eth1:192.168.100.20，某 192.168.100.22 主机网关指向 192.168.100.20，若它 ping 172.16.10.5，结果将是通的，因为地址属于内核，从 eth1 进来的数据包被内核分析时，发现目标地址为本机地址，直接就回应 192.168.100.22，回应数据包继续从 eth1 出去。

如果 Linux 主机有多块网卡，如果不开启数据包转发功能，则这些网卡之间是无法互通的。例如 eth0 是 172.16.10.0/24 网段，而 eth1 是 192.168.100.0/24 网段，到达该 Linux 主机的数据包无法从 eth0 交给 eth1 或者从 eth1 交给 eth0，除非 Linux 主机开启了数据包转发功能。

在 Linux 上开启转发功能有多种方法：

```
shell> echo 1 > /proc/sys/net/ipv4/ip_forward
shell> sysctl -w net.ipv4.ip_forward=1
```

以上两种方法是临时生效的，若要永久生效，则应该写入配置文件。在 CentOS 6 中，将/etc/sysctl.conf 文件中的"net.ipv4.ip_forward"值改为 1 即可，但在 CentOS 7 中，systemd 管理了太多的功能，sysctl 的配置文件分化为多个，包括/etc/sysctl.conf、/etc/sysctl.d/*.conf 和 /usr/lib/sysctl.d/*.conf，并且这些文件中都不再包括 net.ipv4.ip_forward 项。当然，直接将此项写入到这些配置文件中也都是可以的，建议写在 /etc/sysctl.d/*.conf 中，这是 systemd 提供自定义内核修改项的目录。例如：

```
shell> echo "net.ipv4.ip_forward=1" > /etc/sysctl.d/ip_forward.conf
```

可以使用以下几种方式查看是否开启了转发功能。

```
[root@xuexi ~]# sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 0
[root@xuexi ~]# cat /proc/sys/net/ipv4/ip_forward
0
[root@xuexi ~]# sysctl -a | grep ip_forward
net.ipv4.ip_forward = 0
net.ipv4.ip_forward_use_pmtu = 0
```

13.2 和网络相关的几个文件说明

13.2.1 网卡配置文件 ifcfg-*

在/etc/sysconfig/network-scripts/目录下有不少文件，绝大部分都是脚本类的文件，但有一类 ifcfg 开头的文件为网卡配置文件(interface config)，所有 ifcfg 开头的文件在启动网络服务的时候都会被加载读取，但具体的文件名 ifcfg-XX 的 XX 可以随意命名。

以下是一个(CentOS 7 上)ifcfg-XX 文件的内容示例。

```
[root@xuexi ~]# cat /etc/sysconfig/network-scripts/ifcfg-eth0
DEVICE="eth0"      # 显示的名称，必须/sys/class/net/目录下的某个网卡名相同
IPV6INIT="no"
```

```
BOOTPROTO="dhcp"
ONBOOT=yes
TYPE="Ethernet"
DEFROUTE="yes"
PEERDNS="yes"      # 设置为 yes 时，此文件设置的 DNS 将覆盖/etc/resolv.conf，
                   # 若开启了 DHCP，则默认为 yes，所以 dhcp 的 dns 也会覆盖/etc/resolv.conf
PEERROUTES="yes"
IPV4_FAILURE_FATAL="no"
NAME="System eth0"
DNS1=114.114.114.114
DNS2=8.8.8.8
DNS3=114.114.115.115
```

13.2.2 DNS 配置文件/etc/resolv.conf

该文件用于设置 DNS 指向，以及解析顺序。该文件格式如下：

```
domain  domain_name         # 声明本地域名，即解析时自动隐式补齐的域名
search  domain_name_list    # 指定域名搜索顺序(最多 6 个)，和 domain 不能共存，若共存了，则写在后面的行生效
nameserver  IP1             # 设置 DNS 指向，最多 3 个
nameserver  IP2
nameserver  IP3
options timeout:n attempts:n  # 指定解析超时时间(默认 5 秒)和解析次数(默认 2 次)
```

例如将/etc/resolv.conf 设置为下所示，为了测试，暂且不设置 nameserver。

```
domain malong.com
```

当解析不带点"."的主机名时，如"www"，认为不是 fqdn，将自动加上".malong.com"变成解析"www.malong.com"。

```
[root@xuexi ~]# host -a www
Trying "www.malong.com"
;; connection timed out; trying next origin
Trying "www"
;; connection timed out; no servers could be reached
```

当解析的名称末尾不带点但中间带了点的，如"www.host"，认为是 fqdn，将直接解析"www.host"，解析完这个后再解析加上"malong.com"的名称，即再解析"www.host.malong.com"。

```
[root@xuexi ~]# host -a www.host
Trying "www.host"
;; connection timed out; trying next origin
Trying "www.host.malong.com"
;; connection timed out; no servers could be reached
```

当解析末尾带点的名称时，如"www.host."认为是完整的 fqdn，将直接解析"www.host",解析完后直接结束解析，不会再补齐本地域名再解析。

```
[root@xuexi ~]# host -a www.host.
Trying "www.host"
;; connection timed out; trying next origin
Trying "www.host"  # 默认解析两次
;; connection timed out; no servers could be reached
```

search 关键字的作用和 domain 是一样的，只不过 search 同时还暗含域名搜索的顺序。例如设置 search 为如下内容：

```
search  malongshuai.com longshuai.com mashuai.com
```

此时若解析"www.host"，将依次解析"www.host","www.host.malongshuai.com"，"www.host.longshuai.com"，"www.host.mashuai.com"。

```
[root@xuexi ~]# host -a www.host
Trying "www.host"
;; connection timed out; trying next origin
Trying "www.host.malongshuai.com"
;; connection timed out; trying next origin
Trying "www.host.longshuai.com"
;; connection timed out; trying next origin
Trying "www.host.mashuai.com"
;; connection timed out; no servers could be reached
[root@xuexi ~]# host -a www
Trying "www.malongshuai.com"
;; connection timed out; trying next origin
Trying "www.longshuai.com"
;; connection timed out; trying next origin
```

```
Trying "www.mashuai.com"
;; connection timed out; trying next origin
Trying "www"
;; connection timed out; no servers could be reached
```

domain 部分和 search 部分不能共存，如果共存了，则后出现的行有效。

13.2.3 /etc/udev/rules.d/70-persistent-net.rules

当插入新的网络设备时，内核首先识别到，随后在 sysfs 文件系统(一般挂载在/sys 下)中生成该设备对应的信息文件。然后内核通知 udev 的后台守护进程 udevd(若不知道它是什么东西，请认为它是 Windows 系统中的设备管理器，管理和监视硬件设备)，udevd 将读取 sysfs 中对应设备的相关信息，并比对或生成 udev 的规则集，能匹配上的则做对应的操作。对于网卡来说，它的的规则集文件默认为/etc/udev/rules.d/70-persistent-net.rules，匹配该规则集成功后，最后还在/sys/class/net 目录中生成对应的设备子目录。

以下为两个网卡的规则集的内容：

```
[root@xuexi ~]# cat /etc/udev/rules.d/70-persistent-net.rules

# PCI device 0x8086:0x100f (e1000)
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="00:0c:29:7f:cf:a4", ATTR{type}=="1", KERNEL=="eth*", NAME="eth0"

# PCI device 0x8086:0x100f (e1000)
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="00:0c:29:7f:cf:ae", ATTR{type}=="1", KERNEL=="eth*", NAME="eth1"
```

具体的 udev 规则集语法并非本文内容，所以此处仅简单解释下上面的两个规则集。规则集文件的写法都是 key/value 格式，但分为匹配 key/vaule 和行为 key/value。上述例子中，从 SUBSYSTEM 直到 KERNEL 都是使用"=="号，表示匹配 key/value，最后一个 NAME 使用单"="号，表示赋值 key/value。所以上述文件的意思是：当/sys 中的某设备各信息都能匹配上述某条规则，则赋值该设备名称为 eth0 或 eth1，/sys/class/net 目录下也由此名称命名设备信息的目录，**ifcfg-*配置文件中的 DEVICE 的值必须和它们相同。**

注意，网络规则集文件会由内核检测到设备时自动生成或写入，因此清空它不会有任何影响。

克隆虚拟机时，总是会出现 MAC 地址冲突，这是因为规则集文件和 ifcfg 配置文件都被克隆了，而新克隆出来的机器中 MAC 地址又是新的，所以会生成新的规则集，但克隆过来的 ifcfg 配置文件中的 DEVICE 值和该规则对应不上，导致克隆主机的网络将启动不了。解决办法是清空该文件，然后重启克隆主机，这样内核将新生成对应新 MAC 地址的规则集文件。当然，在克隆前清空模板主机的规则集文件，然后再克隆也是可以的。

值得一提的是，在 CentOS 7 中，systemd 已经将 udevd 的功能整合在了一起，所以 udev 的规则集文件已经少见了，但却更方便了，克隆 CentOS 7 主机时，根本就不会出现 MAC 地址冲突的可能，因为一切规则集都由内核生成在内存中。不过，显式书写的规则集文件仍然是生效的。

13.2.4 /etc/services

该文件中记录的是端口和服务的对应关系。

```
[root@xuexi ~]# grep '^ftp\|^ssh' /etc/services
ftp-data        20/tcp
ftp-data        20/udp
ftp             21/tcp
ftp             21/udp          fsp fspd
ssh             22/tcp                          # The Secure Shell (SSH) Protocol
ssh             22/udp                          # The Secure Shell (SSH) Protocol
ftp-data        20/sctp                 # FTP
ftp             21/sctp                 # FTP
ssh             22/sctp                 # SSH
ftp-agent       574/tcp                 # FTP Software Agent System
ftp-agent       574/udp                 # FTP Software Agent System
sshell          614/tcp                 # SSLshell
sshell          614/udp                 #       SSLshell
ftps-data       989/tcp                 # ftp protocol, data, over TLS/SSL
ftps-data       989/udp                 # ftp protocol, data, over TLS/SSL
ftps            990/tcp                 # ftp protocol, control, over TLS/SSL
ftps            990/udp                 # ftp protocol, control, over TLS/SSL
ssh-mgmt        17235/tcp               # SSH Tectia Manager
ssh-mgmt        17235/udp               # SSH Tectia Manager
```

13.3 网络接口配置和主机名

13.3.1 ifconfig

该命令虽然在 man 文档中被说明已废弃，但大众显然无法忘记它。ifconfig 命令是一个接口配置命令，但更多的被用来显示已激活的网络接口信息。

```
ifconfig [ interface | -a ]
ifconfig interface options
选项说明：
```

```
interface：指定被操作的网络接口名，如 eth0
up        ：激活指定的网络接口，如果在命令行中为网络接口分配了 IP 地址，则默认会 up
down      ：将指定的接口设置为 down 状态
[-]arp    ：启用或禁用该接口上使用 ARP 协议，如"ifconfig eth0 -arp"
mtu N     ：设置指定接口的最大传输单元(MTU)
netmask   ：设置该接口的 IP netmask，默认会采用 A/B/C 类地址的掩码位数
address   ：要分配给该接口的 IP 地址
```

ifconfig 示例：

```
[root@xuexi ~]# ifconfig eth0:1 192.168.100.20 netmask 255.255.255.0 up  # 添加 IP 地址
[root@xuexi ~]# ifconfig eth0:1 192.168.100.20/24 up                      # 使用 CIDR 格式的掩码也可以
[root@xuexi ~]# ifconfig eth1 up       # 激活该网络接口
[root@xuexi ~]# ifconfig eth1 down    # 临时 down 掉 eth1 接口
[root@xuexi ~]# ifconfig eth1 -arp     # 抑制 eth1 上的 arp
[root@xuexi ~]# ifconfig eth1 arp      # 启用 eth1 上的 arp
```

需要注意的是，ifconfig 所有的配置都是应用于内核的，所以只会临时生效，重启网络服务后会立即失效。

对于 slave 地址，即别名地址，若要永久生效，应该建立对应的别名接口配置文件，如/ets/sysconfig/network-scripts/ifcfg-eth0:0，然后在该文件中的 DEVICE 关键字上给定 eth0:0 名称，该 DEVICE 项必须配置正确。

13.3.2 ifcfg

用法很简单。

```
ifcfg DEV [[add|del [ADDR[/LEN]] | stop]
       add - add new address
       del - delete address
       stop - completely disable IP
```

例如：

```
[root@xuexi ~]# ifcfg eth1:0 add 192.168.100.20/24   # 添加一个地址
[root@xuexi ~]# ifcfg eth1:0 del 192.168.100.20      # 删除一个地址
[root@xuexi ~]# ifcfg eth1 stop      # 临时禁用 eth1
```

13.3.3 hostname 命令

用于设置主机名，但也有几个其它好用的功能。

```
hostname [-I] [-f] [-d] [-s] [hostname]
选项说明：
-I         ：获取该主机上所有非环回 IP 地址，该选项不依赖于主机名解析
-f,--fqdn  ：获取 fqdn
-d,--domain：获取 fqdn 的域名部分，等价于命令 dnsdomainname
-s,--short ：获取 fqdn 的主机名部分，严格地说是获取第一个"."前的部分，例如"www.baidu.com"将获取为"www"
```

使用-I 选项可以直接获取该主机上的所有 IP 地址，这在某些时候太方便了。

```
[root@xuexi ~]# hostname -I
192.168.100.54 172.16.10.10
```

hostname 修改的主机名为临时生效，它修改的其实是/proc/sys/kernel/hostname 文件。

```
[root@xuexi ~]# cat /proc/sys/kernel/hostname
xuexi.longshuai.com
```

虽然在 man 文档中说有个永久有效的选项(-b)，但测试时却毫无效果。要想永久生效，需要修改配置文件/etc/hostname(CentOS 7)或/etc/sysconfig/network(CentOS 6)。例如在 CentOS 7 上：

```
[root@xuexi ~]# echo "ma.longshuai.com" >/etc/hostname
```

13.4 网关/路由

分为 3 种路由：

- 主机路由：直接指明到某台具体的主机怎么走；
- 网络路由：指明某类网络怎么走；
- 默认路由：不走主机路由的和网络路由的就走默认路由。默认路由一般也称为默认网关。

若 Linux 上到某主机有多条路由可以选择，这时候会挑选优先级高的路由。在 Linux 中，路由条目的优先级确定方式是先匹配掩码位长度，再比较管理距离(比如 metric)。也就是说，掩码位长的路由条目优先级一定比掩码位短的优先级高，所以主机路由的优先级最高，然后是直连网络(即同网段)的路由(也算是网络路由)次之，再是网络路由，最后才是默认路由。若路由条目的掩码长度相同，则比较节点之间的管理距离，管理距离短的生效。

例如下面的路由表中，若 ping 192.168.5.20，则先比对 192.168.100.78 发现无法匹配，然后比对 192.168.100.0，发现也无法匹配，接着再匹配 192.168.0.0 这条网络路由条目，发现能匹配，所以选择该路由条目。

```
[root@xuexi ~]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.100.2   0.0.0.0         UG    100    0        0 eth0
172.16.10.0     0.0.0.0         255.255.255.0   U     100    0        0 eth1
192.168.0.0     192.168.100.70  255.255.0.0     UG    0      0        0 eth0
192.168.100.0   0.0.0.0         255.255.255.0   U     100    0        0 eth0
192.168.100.78  0.0.0.0         255.255.255.255 UH    0      0        0 eth0
```

再比如下面的路由表。由于两块网卡 eth0 和 eth1 都是 192.168.100.0/24 网段地址，所以它们的路由条目在掩码长度上是相同的，但是和 eth0 直连的网段主机通信时，肯定会选择 eth0 这条路由条目，因为 eth1 和该网段主机隔了一个 eth0，距离增加了 1。

```
[root@xuexi ~]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.100.2   0.0.0.0         UG    100    0        0 eth0
192.168.100.0   0.0.0.0         255.255.255.0   U     100    0        0 eth0
192.168.100.0   0.0.0.0         255.255.255.0   U     101    0        0 eth1
```

13.4.1 route 命令

route 命令用于显示和管理路由表。当使用了 add 或 del 选项时，route 命令将设置路由条目，否则 route 命令将显示路由表。

要显示路由表信息，只需简单的 route -n 即可，其中-n 选项表示不解析主机名。

例如：

```
[root@xuexi ~]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.100.2   0.0.0.0         UG    100    0        0 eth0
172.16.10.0     0.0.0.0         255.255.255.0   U     100    0        0 eth1
192.168.0.0     192.168.100.70  255.255.0.0     UG    0      0        0 eth0
192.168.100.0   0.0.0.0         255.255.255.0   U     100    0        0 eth0
192.168.100.78  0.0.0.0         255.255.255.255 UH    0      0        0 eth0
```

对于 CentOS 6 以上的系统，请忽略 Metric 和 Ref 两列，它们已经不被内核使用，只是有些路由软件可能会用上。

对于 Flags 列，如果没有安装路由软件，则只可能出现下面的 3 种值：

- U (route is up)
- H (target is a host)
- G (use gateway，也即是设置了下一跳的路由条目)

若要管理路由表，则使用 add 或 del 选项。

```
route [add/del] [-host/-net/default] [address[/mask]] [netmask] [gw] [dev]
选项说明：
add/del：增加或删除路由条目
-net：增加或删除的是一条网络路由
-host：增加或删除的是一条主机路由
default：增加或删除的是一条默认路由
netmask：明确使用 netmask 关键字指定掩码，要可以不使用该选项直接在地址上使用 cidr 格式的掩码，即 IP/MASK。
gw：指定下一跳的地址。要求下一跳地址必须是能到达的，且一般是和本网段直连的接口。
dev：强制将路由条目关联到指定的接口上。一般内核会自动判断路由条目应该关联到哪个网络接口。
```

例如：

(1).添加和删除默认路由

```
shell> route add default gw 192.168.100.10
shell> route del default
shell> route del default gw 192.168.100.10   # 若有多条默认路由，则再加上 gw 即可唯一删除指定路由条目
```

因为默认路由的目的地是 0.0.0.0，所以操作默认路由也可以使用 0.0.0.0 替代 default 关键字，但这样就麻烦的多了。

(2).添加和删除网络路由

```
shell> route add -net 172.16.10.0/24 gw 192.168.100.70
shell> route add -net 172.16.10.0 netmask 255.255.255.0 gw 192.168.100.70
```

若实在不知道下一跳给谁，那么指定本机接口也是可以的。

```
shell> route add -net 172.16.10.0/24 dev eth0
```

删除路由可以直接在增加路由的语句上将 add 改为 del 关键字。如

```
shell> route del -net 172.16.10.0/24 gw 192.168.100.70
shell> route del -net 172.16.10.0 netmask 255.255.255.0 gw 192.168.100.70
shell> route del -net 172.16.10.0/24 dev eth0
```

但大多数时候，可以偷懒，只要能唯一确定删除的是哪条路由即可。如：

```
shell> route del -net 172.16.10.0/24
```

(3)添加和删除主机路由

```
shell> route add -host 172.16.10.55 gw 192.168.10.20
shell> route del -host 172.16.100.55
```

13.4.2 配置永久路由

根据接口创建路由配置文件/etc/syconfig/network-scripts/route-ethX，要从哪个接口出去 X 就是几。

路由配置文件的配置格式非常简单，每一行一个路由条目，先是要到达的目标，然后是 via 关键字，最后是下一跳地址。要求下一跳必须能到达，且一般都和 ethX 同网段。

```
DEST    via     nexthop
```

例如 eth0 网卡的 IP 地址是 192.168.10.123，要通过网卡 eth0 出去到达 10.0.0.10，那么下一跳的地址要和 eth0 的地址在同网段，如 192.168.10.222。

```
10.0.0.0 via 192.168.10.222
```

添加主机路由、默认路由、网段路由示例如下，其中 dev 是可以省略的，因为没有任何用处，配置在哪个 eth 文件中就会从哪个接口出去。

```
#默认路由
default      via 192.168.100.1
0.0.0.0/0    via 192.168.100.1

#网段路由
192.168.10.0/24    via 192.168.100.1

#主机路由
192.168.100.52/32 via 192.168.100.33 dev eth1
```

配置完后，重启 network 服务即可立即生效。

route-ethX 文件的还有另外一种永久路由的配置写法，但上面的方法更简单快捷，所以此处就不多说了。

配置永久路由时，需要注意几点：

(1).**route-ethX 的对应网卡配置文件 ifcfg-ethX 必须存在，否则路由无效。**(对于虚拟机，通常新添加的网卡都没有对应的 ifcfg-ethX 文件，但 ifconfig 却能找到该网卡)

(2).**如果在文件中配置永久默认路由，则必须保证所有使用了 DHCP 服务的网卡配置文件 ifcfg-ethX 中的 DEFROUTE 指令设置为"no"，表示 DHCP 不设置默认路由。**

(3).**如果在 route-ethX 文件中配置永久路由，且该网卡使用了 DHCP 服务分配地址，则必须保证该网卡的 ifcfg-ethX 文件中的 PEERROUTES 指令设置为"no"，表示 DHCP 设置的路由允许被覆盖。**

13.5 arp 和 arping 命令

维护或查看系统 arp 缓存，该命令已废弃，使用 ip neigh 代替。

arp 为地址解析协议，将给定的 ipv4 地址在网络中查找其对应的 MAC 地址。

一般会使用 arp 协议获取局域网内的主机 MAC，所以局域网主机之间也互称为网络邻居。

13.5.1 arp

arp 命令语法：

```
arp -n -v -i    # 查看 arp 缓存
arp -i -d hostname      # 删除 arp 缓存条目
选项说明：
-n：不解析 ip 地址为名称
-v：详细信息
-i：指定操作的接口
```

-d：删除一个 arp 条目
hostname：操作该主机的 arp 条目，除了删除还有其他动作，如手动添加主机的 arp 条目，此处就不解释该用法了

例如：

```
[root@xuexi ~]# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.100.1            ether   00:50:56:c0:00:08   C                     eth1
192.168.100.254          ether   00:50:56:e7:e1:d4   C                     eth0
192.168.100.70           ether   00:0c:29:71:81:64   C                     eth0
192.168.100.1            ether   00:50:56:c0:00:08   C                     eth0
192.168.100.2            ether   00:50:56:e2:16:04   C                     eth1
192.168.100.254          ether   00:50:56:e7:e1:d4   C                     eth1
192.168.100.2            ether   00:50:56:e2:16:04   C                     eth0
```

其实查看的信息是/proc/net/arp 文件中的内容。

```
[root@xuexi ~]# cat /proc/net/arp
IP address       HW type     Flags       HW address            Mask     Device
192.168.100.1    0x1         0x2         00:50:56:c0:00:08     *        eth1
192.168.100.254  0x1         0x2         00:50:56:e7:e1:d4     *        eth0
192.168.100.70   0x1         0x2         00:0c:29:71:81:64     *        eth0
192.168.100.1    0x1         0x2         00:50:56:c0:00:08     *        eth0
192.168.100.2    0x1         0x2         00:50:56:e2:16:04     *        eth1
192.168.100.254  0x1         0x2         00:50:56:e7:e1:d4     *        eth1
192.168.100.2    0x1         0x2         00:50:56:e2:16:04     *        eth0
[root@xuexi ~]# arp -d 192.168.100.70 -i eth0   # 删除 arp 缓存条目
```

arp 命令一次只能删除一条 arp 条目，要批量删除或清空整个 arp 条目，使用 ip neigh flush 命令。如：

```
[root@xuexi ~]# ip neigh flush all            # 清空所有
[root@xuexi ~]# ip neigh flush dev eth0     # 删除 eth0 上缓存的 arp 条目
```

13.5.2 arping

arping 用于发送 arp 请求报文，解析并获取目标地址的 MAC。默认将先发送广播报文，收到回复后再发送单播报文，局域网内所有主机都能收到广播报文，但只有目标主机才会回复自己的 MAC 地址。

注意：发送 arp 请求报文实际上是另类的 ping，所以可以探测目标是否存活，也需要和目标通信，通信时目标主机上也会缓存本主机(即源地址)的 arp 条目。

语法：

```
arping [-fqbDU] [-c count] [-w timeout] [-I device] [-s source] destination
  -f : 收到第一个 reply 就立即退出
  -q : 安静模式，什么都不输出
  -b : 只发送广播，不发送单播
  -D : 地址冲突检测
  -U : 主动更新邻居的 arp 缓存(Unsolicited ARP mode)
  -c count : 发送多少个 arp 请求包后退出
  -w timeout : 等待 reply 的超时时间
  -I device : 使用哪个接口发送请求包。发送 arp 请求包接口的 MAC 地址将缓存在目标主机上
  -s source : 指定 arp 请求报文中源地址，若发送的接口和源地址不同，则目标主机将缓存该地址和接口的 MAC 地址，而非该源地址所在接口的 MAC 地址
  destination : 向谁发送 arp 请求报文，即要获取该 IP 或主机名的 MAC 地址
```

例如：

(1).请求解析 192.168.100.70 主机的 MAC 地址

```
[root@xuexi ~]# arping -f 192.168.100.70
```

这将会发送广播报文，直到收到 192.168.100.70 的回复才退出。

同时，192.168.100.70 也会缓存本机的 IP 和 MAC 对应条目，由于此处没有指定请求报文的发送接口和源地址，所以发送报文时是根据路由表来选择接口和对应该接口地址的。

(2).指定发送一个请求报文给 192.168.100.70 就退出，发送报文的接口为 eth1，并指定请求报文中的源地址为本机 eth0 接口上的地址 192.168.100.54

```
[root@xuexi ~]# arping -c 1 -I eth1 -s 192.168.100.54 192.168.100.70
```

发送这样的 arp 请求包，将会使得目标主机 192.168.100.70 缓存本机的 arp 条目为"192.168.100.54 MAC_eth1"，但实际上，192.168.100.54 所在接口的 MAC 地址为 MAC_eth0。

arping 命令仅能实现这种简单的 arp 欺骗，更多的 arp 欺骗方法可以使用专门的工具。

(3).探测对方主机是否存活

例如发送 4 个探测报文，有回复就说明对方存活

```
[root@xuexi ~]# arping -c 4 -I eth0 192.168.100.2
ARPING 192.168.100.2 from 192.168.100.54 eth0
Unicast reply from 192.168.100.2 [00:50:56:E2:16:04]  0.593ms
Unicast reply from 192.168.100.2 [00:50:56:E2:16:04]  0.930ms
Unicast reply from 192.168.100.2 [00:50:56:E2:16:04]  0.868ms
Unicast reply from 192.168.100.2 [00:50:56:E2:16:04]  0.844ms
Sent 4 probes (1 broadcast(s))
Received 4 response(s)
```

可见发送了 4 个探测报文，其中第一个报文是广播报文，并且收到了 4 个回复。

13.6 ip 命令

这是一个极其强大的命令，前面所有的网络信息显示和管理的命令，都可以由 ip 命令来替代完成。它是一个严格模式化的命令。

13.6.1 获取 ip 命令的帮助

先简单说明下 ip 命令的基础和获取帮助的方法。

```
[root@xuexi ~]# ip -h
Usage: ip [ OPTIONS ] OBJECT { COMMAND | help }
       ip [ -force ] -batch filename
where  OBJECT := { link | addr | addrlabel | route | rule | neigh | ntable |
                   tunnel | tuntap | maddr | mroute | mrule | monitor | xfrm |
                   netns | l2tp | tcp_metrics | token }
       OPTIONS := { -V[ersion] | -s[tatistics] | -d[etails] | -r[esolve] |
                    -h[uman-readable] | -iec |
                    -f[amily] { inet | inet6 | ipx | dnet | bridge | link } |
                    -4 | -6 | -I | -D | -B | -0 |
                    -l[oops] { maximum-addr-flush-attempts } |
                    -o[neline] | -t[imestamp] | -b[atch] [filename] |
                    -rc[vbuf] [size] | -n[etns] name | -a[ll] }
```

可见命令非常复杂，有很多 options，还有很多 object，每个 Object 又对应不同的命令。但其实能用到的就几个 object：addr/route/neigh/link。

使用 ip object help 可以获取到该 object 的语法帮助。例如：

```
[root@xuexi ~]# ip addr help
```

在 ip 命令行下，任何 object 都可以写其全名，也可以写其缩写名，例如 address 这个 object，可以简写为 addr，也可以简写为一个字母 a。

```
[root@xuexi ~]# ip a help      # 等价于 ip address help 和 ip addr help
```

尽管还有一个 a 开头的 object 为 addrlabel。这时因为 ip 会从上述语法给出的 object 顺序从前向后匹配，例如"ip m"将匹配到"ip maddr"，如果想匹配别的，如 addrlabel，则写长一点即可"ip addrl"。

对于 CentOS 6，man ip 时会输出整个 ip 的帮助文档，包括每个 object 的命令和说明。在 CentOS 7 中，则要对每个 object 独立进行 man，例如 addr 这个 object。

```
[root@xuexi ~]# man ip-address
```

以下是所有 Object 的 man 列表。

```
[root@xuexi ~]# rpm -ql iproute | grep "man8/ip-"
/usr/share/man/man8/ip-address.8.gz
/usr/share/man/man8/ip-addrlabel.8.gz
/usr/share/man/man8/ip-l2tp.8.gz
/usr/share/man/man8/ip-link.8.gz
/usr/share/man/man8/ip-maddress.8.gz
/usr/share/man/man8/ip-monitor.8.gz
/usr/share/man/man8/ip-mroute.8.gz
/usr/share/man/man8/ip-neighbour.8.gz
/usr/share/man/man8/ip-netconf.8.gz
/usr/share/man/man8/ip-netns.8.gz
/usr/share/man/man8/ip-ntable.8.gz
/usr/share/man/man8/ip-route.8.gz
/usr/share/man/man8/ip-rule.8.gz
/usr/share/man/man8/ip-tcp_metrics.8.gz
/usr/share/man/man8/ip-token.8.gz
```

```
/usr/share/man/man8/ip-tunnel.8.gz
/usr/share/man/man8/ip-xfrm.8.gz
```

13.6.2 ip addr

ip addr 用于管理网络设备上的 ip 地址，也可以查看 ip 地址的属性信息。在老版本的 Linux 中，一块网卡上设置多个 IP，这些 IP 称为别名 IP，但是从 CentOS 6 开始，这些 IP 称为 secondary IP 或 slave IP，因为这些 IP 自身也可以附带属性。

- **ip addr add/del**

```
ip address { add | del } IFADDR dev STRING
IFADDR := PREFIX [ broadcast ADDR ] [ anycast ADDR ] [ label STRING ]
以 add 为例：
dev NAME：指定要设置 IP 地址的网卡
local ADDRESS (default)：接口的 IP 地址。IP 地址的格式依赖于是 ipv4 还是 ipv6。对于 ipv4 而言，给定地址，可能还需要给定 cidr 的掩码位长度
broadcast ADDRESS：接口的广播地址
label NAME：为该接口的 IP 地址设置 label 名，label 名称必须以网络接口名开头后接冒号，如 eth0:X
```

del 和 add 的参数相同，且 dev 是必须要给定的，其余的参数可选，因为 del 的时候是通配 del，如果删除时有多个满足条件的条目，则删除第一个条目。

```
[root@xuexi ~]# ip addr add 192.168.100.45 dev eth0
[root@xuexi ~]# ip addr add 192.168.100.35/24 dev eth1
```

此方式添加的地址不会在 ifconfg 命令中显示，ifconfg 能捕捉到的是别名，所以可以为地址加上 label，以让 secondary 被 ifconfig 查看到。例如：

```
[root@xuexi ~]# ip addr add 192.168.100.45 dev eth0 label eth0:0
```

要删除 ip，则简单的多，但必须指定 dev，且最好也指定 cidr 的掩码长度。

```
[root@xuexi ~]# ip addr del 192.168.100.45 dev eth0
[root@xuexi ~]# ip addr del 192.168.100.35/24 dev eth1
```

- **ip addr show**

虽然也有几个选项，但是感觉没什么用，直接 ip addr show 就够了。因为 ip 命令可以缩写，所以可以写为

```
[root@xuexi ~]# ip a show
[root@xuexi ~]# ip a s
[root@xuexi ~]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:71:81:64 brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.70/24 brd 192.168.100.255 scope global eth0
    inet6 fe80::20c:29ff:fe71:8164/64 scope link
       valid_lft forever preferred_lft forever
```

- **ip addr flush**

用于批量删除地址，该命令其实非常危险，一个不小心就会误伤无辜，所幸的是 flush 的时候不给定任何参数或者没有任何条目可以匹配上的时候将不执行 flush 动作，总之该命令要小心使用。同样也必须给定 dev 参数。

例如删除 eth1 上所有地址。

```
[root@xuexi ~]# ip a flush dev eth1
```

删除 eth1 上所有的 secondary 地址。

```
[root@xuexi ~]# ip a f secondary dev eth1
```

13.6.3 ip route

该命令维护和查看内核中的路由表。

- **ip route add/del/change/append/replace**

语法格式为：

```
ip route { add | del | change | append | replace } dest[/cidr_mask] [ via ADDRESS ] [ dev STRING ]
```

其中 dest 为目标地址，可以是主机地址、网段地址，一般在地址后都会带上 cidr 格式的掩码长度，不带时默认为 32 位长度。如果 dest 为"0/0"或者写为"default"，则表示默认路由。

例如添加/修改/替换普通路由：

```
[root@xuexi ~]# ip route add/change/replace 172.16.10.0/24 via 192.168.10.20
```

添加/修改/替换默认路由：

```
[root@xuexi ~]# ip route add/change/replace default via 192.168.10.20
[root@xuexi ~]# ip route add/change/replace 0/0 via 192.168.100.2
```

删除某路由：

```
[root@xuexi ~]# ip route del 172.16.10.0/24
[root@xuexi ~]# ip route del default   # 删除默认路由
```

- **ip route show**

列出路由表。

语法格式为：

```
ip route show [to [ root | match | exact ] ADDR_pattern ] [ via ADDR ]
```

其中 to 关键字是默认关键字，用来匹配路由的目标地址。其后可以跟上修饰符 root/match/exact，exact 为默认修饰符，表示精确匹配掩码位长度，root 修饰符表示匹配的掩码位长度大于或等于 ADDR_pattern 给定的掩码位长度，match 修饰符匹配短于或等于 ADDR_pattern 掩码位长度。例如"to match 16.0/16"将能匹配到"16.0/16"、"16/8"和"0/0"，但却无法匹配"16.1/16"和"16.0/24"以及"16.0.1/24"，而"to root 16.0/16"将能匹配"16.0/24"和"16.0.1/24"。

via 是根据下一跳的方式来列出路由条目。

例如：

```
[root@xuexi ~]# ip route show | column -t
default          via  192.168.100.2  dev    eth0    proto  static  metric  100
172.16.10.0/24   dev  eth1           proto  kernel  scope  link    src     172.16.10.20   metric  100
172.168.10.0/24  dev  eth0           proto  kernel  scope  link    src     172.168.10.20
172.168.10.0/24  dev  eth0           proto  kernel  scope  link    src     172.168.10.20  metric  100
192.168.10.0/24  dev  eth0           proto  kernel  scope  link    src     192.168.10.20
192.168.10.0/24  dev  eth0           proto  kernel  scope  link    src     192.168.10.20  metric  100
192.168.100.0/24 dev  eth0           proto  kernel  scope  link    src     192.168.100.54 metric  100
192.168.100.0/24 dev  eth1           proto  kernel  scope  link    src     192.168.100.74 metric  101
[root@xuexi ~]# ip route show to match 192.168.10/24 | column -t
default          via  192.168.100.2  dev    eth0    proto  static  metric  100
192.168.10.0/24  dev  eth0           proto  kernel  scope  link    src     192.168.10.20
192.168.10.0/24  dev  eth0           proto  kernel  scope  link    src     192.168.10.20  metric  100
[root@xuexi ~]# ip route show to match 192.168/24 | column -t
default  via  192.168.100.2  dev  eth0  proto  static  metric  100
[root@xuexi ~]# ip route show to root 192.168/16 | column -t
192.168.10.0/24   dev  eth0  proto  kernel  scope  link  src  192.168.10.20
192.168.10.0/24   dev  eth0  proto  kernel  scope  link  src  192.168.10.20  metric  100
192.168.100.0/24  dev  eth0  proto  kernel  scope  link  src  192.168.100.54 metric  100
192.168.100.0/24  dev  eth1  proto  kernel  scope  link  src  192.168.100.74 metric  101
```

其实无需那么花哨，简简单单的"ip r"多方便。

- **ip route flush**

批量删除路由表条目。参数和 ip route show 的参数一样。

例如删除由 eth1 出去的路由条目。

```
[root@xuexi ~]# ip route flush eth1
```

删除下一跳为 192.168.100.70 的路由条目。

```
[root@xuexi ~]# ip r flush via 192.168.100.70
```

删除目标为 192.168.0.0/16 网段的路由

```
[root@xuexi ~]# ip route flush 192.168/16
```

- **ip route save / restore**

用于保存当前的路由表以及恢复路由表。保存路由表时，路由表将以二进制裸数据的格式输出，也就是看不懂的二进制文件。恢复路由表时，要求设备的设置和保存路由表时是一样的，恢复时已存在于路由表中的路由条目将被忽略。

例如当前路由表信息如下：

```
[root@xuexi ~]# ip r
default via 192.168.100.2 dev eth0  proto static  metric 100
```

```
192.168.10.0/24 dev eth0  proto kernel  scope link  src 192.168.10.20
192.168.10.0/24 dev eth0  proto kernel  scope link  src 192.168.10.20  metric 100
192.168.100.0/24 dev eth0  proto kernel  scope link  src 192.168.100.54  metric 100
192.168.100.0/24 dev eth1  proto kernel  scope link  src 192.168.100.74  metric 101
```

保存当前路由表。

```
[root@xuexi ~]# ip route save > /tmp/route.txt
```

删除几条路由表。

```
[root@xuexi ~]# ip route flush dev eth0
[root@xuexi ~]# ip r
192.168.100.0/24 dev eth1  proto kernel  scope link  src 192.168.100.74  metric 101
```

恢复路由表。

```
[root@xuexi ~]# ip route restore < /tmp/route.txt
```

13.6.4 ip link

link 表示 link layer 的意思，即链路层。该命令用于管理和查看网络接口，甚至可以添加虚拟网络接口，将网络接口分组进行管理。

- **ip link set**

```
ip link set DEVICE  { up | down | arp { on | off } | name NEWNAME | address LLADDR }
选项说明：
dev DEVICE：指定要操作的设备名
up and down：启动或停用该设备
arp on or arp off：启用或禁用该设备的 arp 协议
name NAME：修改指定设备的名称，建议不要在该接口处于运行状态或已分配 IP 地址时重命名
address LLADDRESS：设置指定接口的 MAC 地址
```

例如，禁用 eth1 网卡。

```
[root@xuexi ~]# ip link set eth1 down
```

其实等价于:

```
[root@xuexi ~]# ifconfig eth1 down
```

修改网卡 eth1 的 MAC 地址。

```
[root@xuexi ~]# ip link set eth1 address 00:0c:29:f3:33:77
```

- **ip link show**

语法格式：

```
ip [ -s | -h ] link show [dev DEV]
选项说明：
-s：将显示各网络接口上的流量统计信息
-h：以人类可读的方式显式，即单位转换。注："-h"在 CentOS 7 上才支持。
```

例如：

```
[root@xuexi ~]# ip -s -h link show  dev eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
    link/ether 00:0c:29:f7:43:77 brd ff:ff:ff:ff:ff:ff
    RX: bytes  packets  errors  dropped overrun mcast
    13.7M      20.0k    0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    1.59M      9.97k    0       0       0       0
```

第14章 开机流程

计算机启动分为内核加载前、加载时和加载后 3 个大阶段，这 3 个大阶段又可以分为很多小阶段，本文将非常细化分析每一个重要的小阶段。

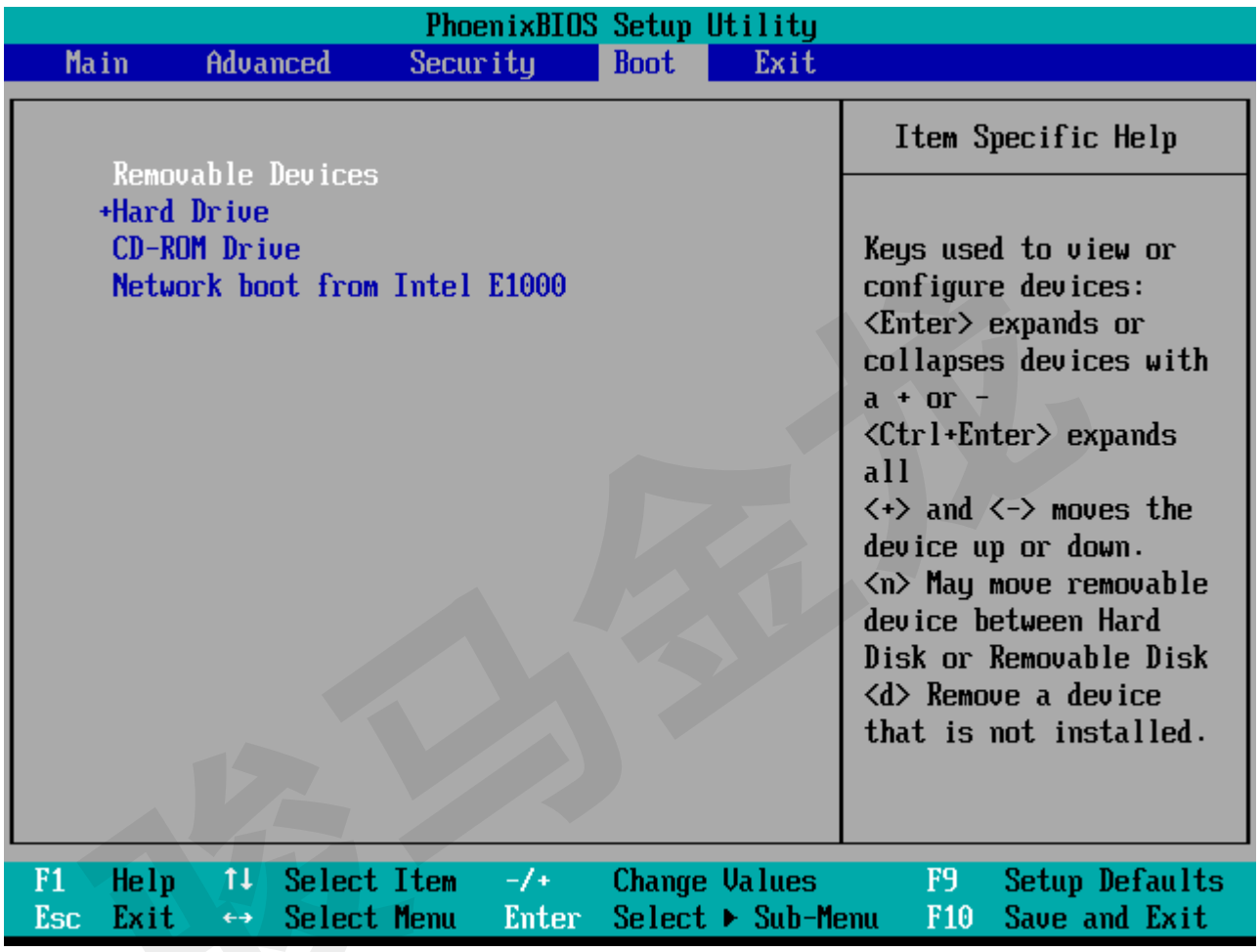
内核加载前的阶段和操作系统无关，Linux 或 Windows 在这部分的顺序是一样的。由于使用 anaconda 安装 Linux 时，默认的图形界面是不支持 GPT 分区的，即使是目前最新的 CentOS 7.3 也仍然不支持，所以在本文中主要介绍传统 BIOS 平台(MBR 方式)的启动方式(其实是本人愚笨，看不懂 uefi 启动方式)。

在内核加载时和加载后阶段，由于 CentOS 7 采用的是 systemd，和 CentOS 5 或 CentOS 6 的 sysV 风格的 init 大不相同，所以本文也只介绍 sysV 风格的 init。

14.1 按下电源和 bios 阶段

按下电源，计算机开始通电，最重要的是要接通 cpu 的电路，然后通过 cpu 的针脚让 cpu 运行起来，只有 cpu 运行起来才能执行相关代码跳到 bios。

bios 是按下开机键后第一个运行的程序，它会读取 CMOS 中的信息，以了解**部分硬件**的信息，比如硬件自检(post)、硬件上的时间、硬盘大小和型号等。其实，手动进入 bios 界面看到的信息，都是在这一阶段获取到的，如下图。对本文来说，最重要的还是获取到了启动设备以及它们的启动顺序(顺序从上到下)信息。



当硬件检测和信息获取完毕，开始初始化硬件，最后从排在第一位的启动设备中读取 MBR，如果第一个启动设备中没有找到合理的 MBR，则继续从第二个启动设备中查找，直到找到正确的 MBR。

14.2 MBR 和各种 bootloader 阶段

这小节将介绍各种 BR(boot record)和各种 boot loader，但只是简单介绍其基本作用。

MBR 是主引导记录，位于磁盘的第一个扇区，和分区无关，和操作系统无关，bios 一定会读取 MBR 中的记录。

在 MBR 中存储了 bootloader/分区表/BRID。bootloader 占用 446 个字节，用于引导加载；分区表占用 64 个字节，每个主分区或扩展分区占用 16 个字节，如果 16 个字节中的第一个字节为 0x80，**则表示该分区为激活的分区(活动分区)，且只允许有一个激活的分区**；最后 2 个字节是 BRID(boot record ID)，它固定为 0x55AA，用于标识该存储设备的 MBR 是否是合理有效的 MBR，如果 bios 读取 MBR 发现最后两个字节不是 0x55AA，就会读取下一个启动设备。

14.2.1 boot loader

MBR 中的 bootloader 只占用 446 字节，所以可存储的代码有限，能加载引导的东西也有限，所以在磁盘的不同位置上设计了多种 boot loader。下面将说明各种情况。

在创建文件系统时，是否还记得有些分区的第一个 block 是 boot sector？这个启动扇区中也放了 boot loader，大小也很有限。**如果是主分区上的 boot sector，则该段 boot loader 所在扇区称为 VBR(volumn boot record)，如果是逻辑分区上的 boot sector，则该段 boot loader 所在扇区称为 EBR(Extended boot sector)**。但很不幸，这两种方式的 boot loader 都很少被使用上了，因为它们很不方便，加上后面出现了启动管理器(LILO 和 GRUB)，它们就被遗忘了。但即使如此，在分区中还是存在 boot sector。

14.2.2 分区表

硬盘分区的好处之一就是可以在不同的分区中安装不同的操作系统，但 boot loader 必须要知道每个操作系统具体是在哪个分区。

分区表的长度只有 64 个字节，里面又分成四项，每项 16 个字节。所以，一个硬盘最多只能分四个主分区。

每个主分区表项的 16 个字节，都由 6 个部分组成：

(1).第 1 个字节：只能为 0 或者 0x80。0x80 表示该主分区是激活分区，0 表示非激活分区。单磁盘只能有一个主分区是激活的。
(2).第 2-4 个字节：主分区第一个扇区的物理位置（柱面、磁头、扇区号等等）。
(3).第 5 个字节：主分区类型。
(4).第 6-8 个字节：主分区最后一个扇区的物理位置。
(5).第 9-12 字节：该主分区第一个扇区的逻辑地址。
(6).第 13-16 字节：主分区的扇区总数。

最后的四个字节"主分区的扇区总数"，决定了这个主分区的长度。也就是说，一个主分区的扇区总数最多不超过 2 的 32 次方。如果每个扇区为 512 个字节，就意味着单个分区最大不超过 2TB。

14.2.3 采用 VBR/EBR 方式引导操作系统

暂且先不讨论 grub 如何管理启动操作系统的，以 VBR 和 EBR 引导操作系统为例。

当 bios 读取到 MBR 中的 boot loader 后，**会继续读取分区表**。分两种情况：

(1)如果查找分区表时发现某个主分区表的第一个字节是 0x80，也就是激活的分区，那么说明操作系统装在了该主分区，然后执行已载入的 MBR 中的 boot loader 代码，加载该激活主分区的 VBR 中的 boot loader，至此，控制权就交给了 VBR 的 boot loader 了；

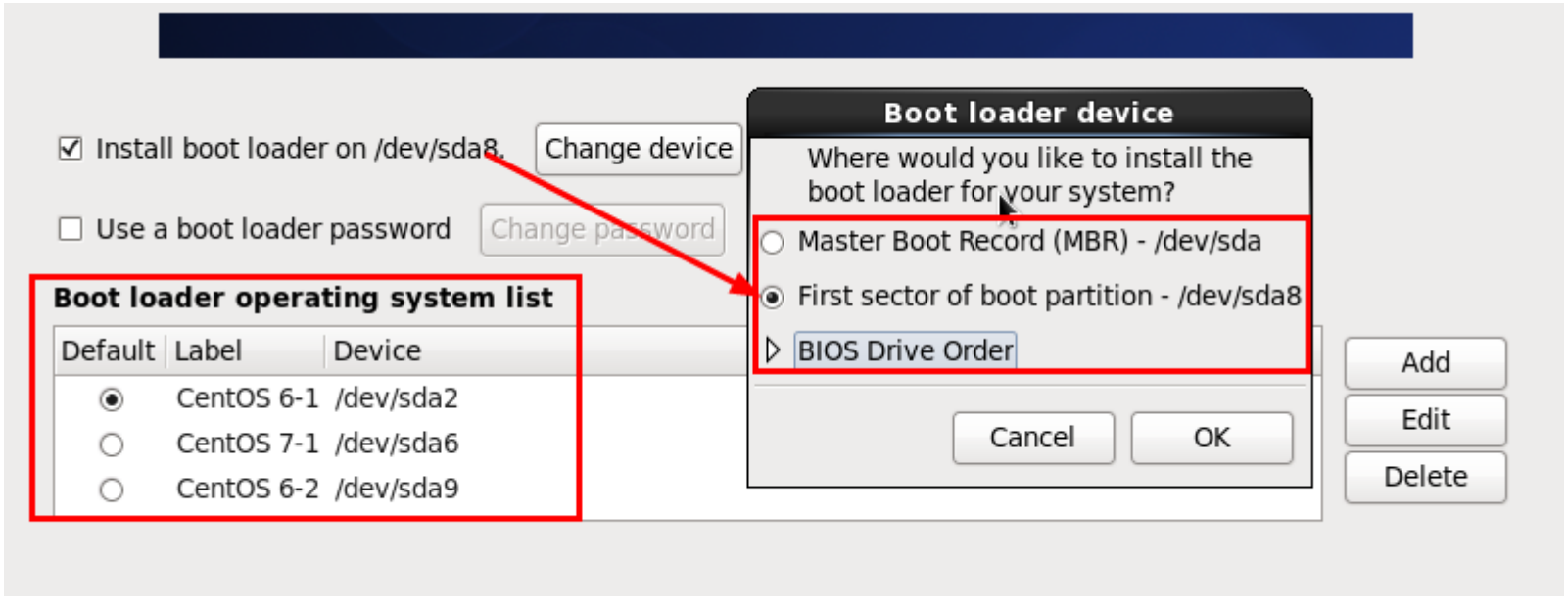
(2)如果操作系统不是装在主分区，那么肯定是装在逻辑分区中，所以查找完主分区表后会继续查找扩展分区表，直到找到 EBR 所在的分区，然后 MBR 中的 boot loader 将控制权交给该 EBR 的 boot loader。

也就是说，如果一块硬盘上装了多个操作系统，那么 boot loader 会分布在多个地方，可能是 VBR，也可能是 EBR，但 MBR 是一定有的，这是被 bios 给"绑定"了的。在装 LINUX 操作系统时，其中有一个步骤就是询问你 MBR 装在哪里的，但这个 MBR 并非一定真的是 MBR，可能是 MBR，也可能是 VBR，还可能是 EBR，并且想要单磁盘多系统共存，则 MBR 一定不能被覆盖(此处不考虑 grub)。

如下图，是我测试单磁盘装 3 个操作系统时的分区结构。其中/dev/sda{1,2,3}是第一个 CentOS 6 系统，/dev/sda{5,6,7}是第二个 CentOS 7 系统，/dev/sda{8,9,10}是第三个 CentOS 6 系统，每一个操作系统的分区序号从前向后都是/boot 分区、根分区、swap 分区。

```
[anaconda root@localhost /]# parted /dev/sda p
Model: VMware, VMware Virtual S (scsi)
Disk /dev/sda: 64.4GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos

Number  Start   End     Size    Type      File system     Flags
 1      1049kB  211MB   210MB   primary   ext4            boot
 2      211MB   21.2GB  21.0GB  primary   ext4
 3      21.2GB  22.4GB  1258MB  primary   linux-swap(v1)
 4      22.4GB  64.4GB  42.0GB  extended
 5      22.4GB  23.0GB  524MB   logical   ext4
 6      23.0GB  40.8GB  17.8GB  logical   ext4
 7      40.8GB  41.9GB  1095MB  logical   linux-swap(v1)
 8      41.9GB  42.1GB  210MB   logical   ext4
 9      42.1GB  56.7GB  14.7GB  logical   ext4
10      56.7GB  64.1GB  7340MB  logical   linux-swap(v1)
```

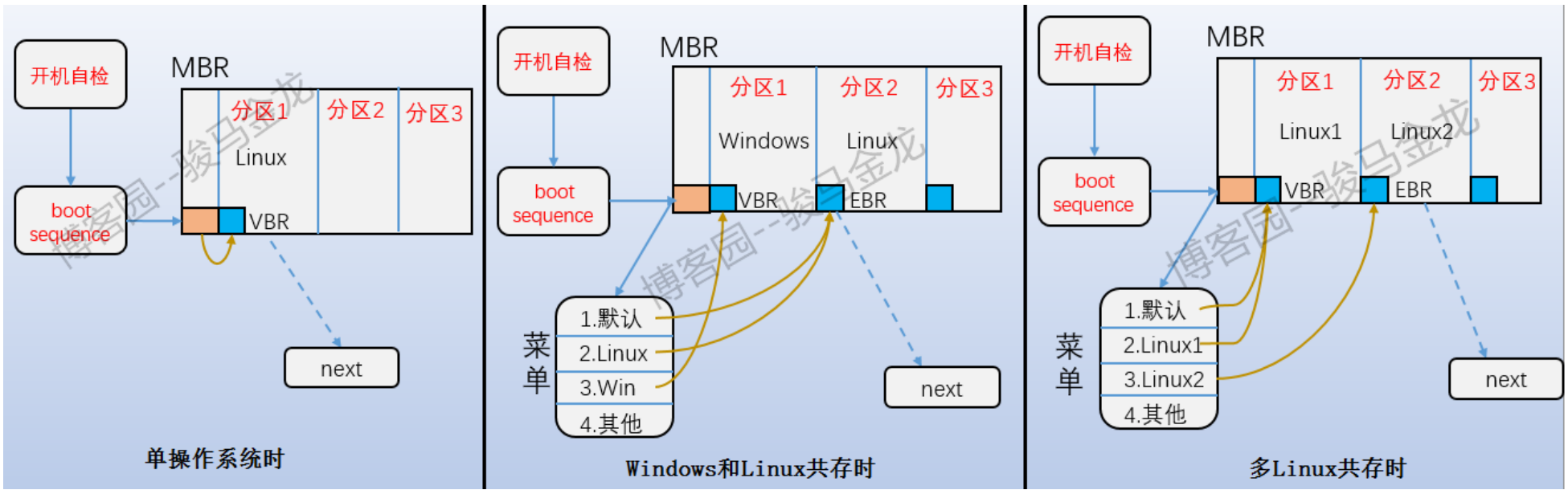
再看下图，是装第三个操作系统时的询问 boot loader 安装位置的步骤。



装第一个操作系统时，boot loader 可以装在/dev/sda 上，也可以选择装在/dev/sda1 上，这时装的是 MBR 和 VBR，任选一个都会将另一个也装上，从第二个操作系统开始，装的是 EBR 而非 MBR，且应该指定 boot loader 位置(如/dev/sda5 和/dev/sda8)，否则默认选项是装在/dev/sda 上，但这会覆盖原有的 MBR。

另外，在指定 boot loader 安装路径的下方，还有一个方框是操作系统列表，这就是操作系统菜单，其中可以指定默认的操作系统，这里的默认指的是 MBR 默认跳转到哪个 VBR 或 EBR 上。

所以，MBR/VBR 和 EBR 之间的跳转关系如下图。



使用这种方式的菜单管理操作系统启动，无需什么 stage1，stage1.5 和 stage2 的概念，只要跳转到了分区上的 VBR 或 EBR，那么直接就可以加载引导该分区上的操作系统。

但是，这种管理操作系统启动的菜单已经没有意义了，现在都是使用 grub 来管理，所以装第二个操作系统或第 n 个操作系统时不手动指定 boot loader 安装位置，覆盖掉 MBR 也无所谓，想要实现单磁盘多系统共存所需要做的，仅仅只是修改 grub 的配置文件而已。

使用 grub 管理引导菜单时，VBR/EBR 就毫无用处了，具体的见下文。

14.3 grub 阶段

使用 grub 管理启动，则 MBR 中的 boot loader 是由 grub 程序安装的，此外还会安装其他的 boot loader。CentOS 6 使用的是传统的 grub，而 CentOS 7 使用的是 grub2。

如果使用的是传统的 grub，则安装的 boot loader 为 stage1、stage1_5 和 stage2，如果使用的是 grub2，则安装的是 boot.img 和 core.img。传统 grub 和 grub2 的区别还是挺大的，所以下面分开解释，如果对于 grub 有不理解之处，见 grub2 详解。

14.3.1 使用 grub2 时的启动过程

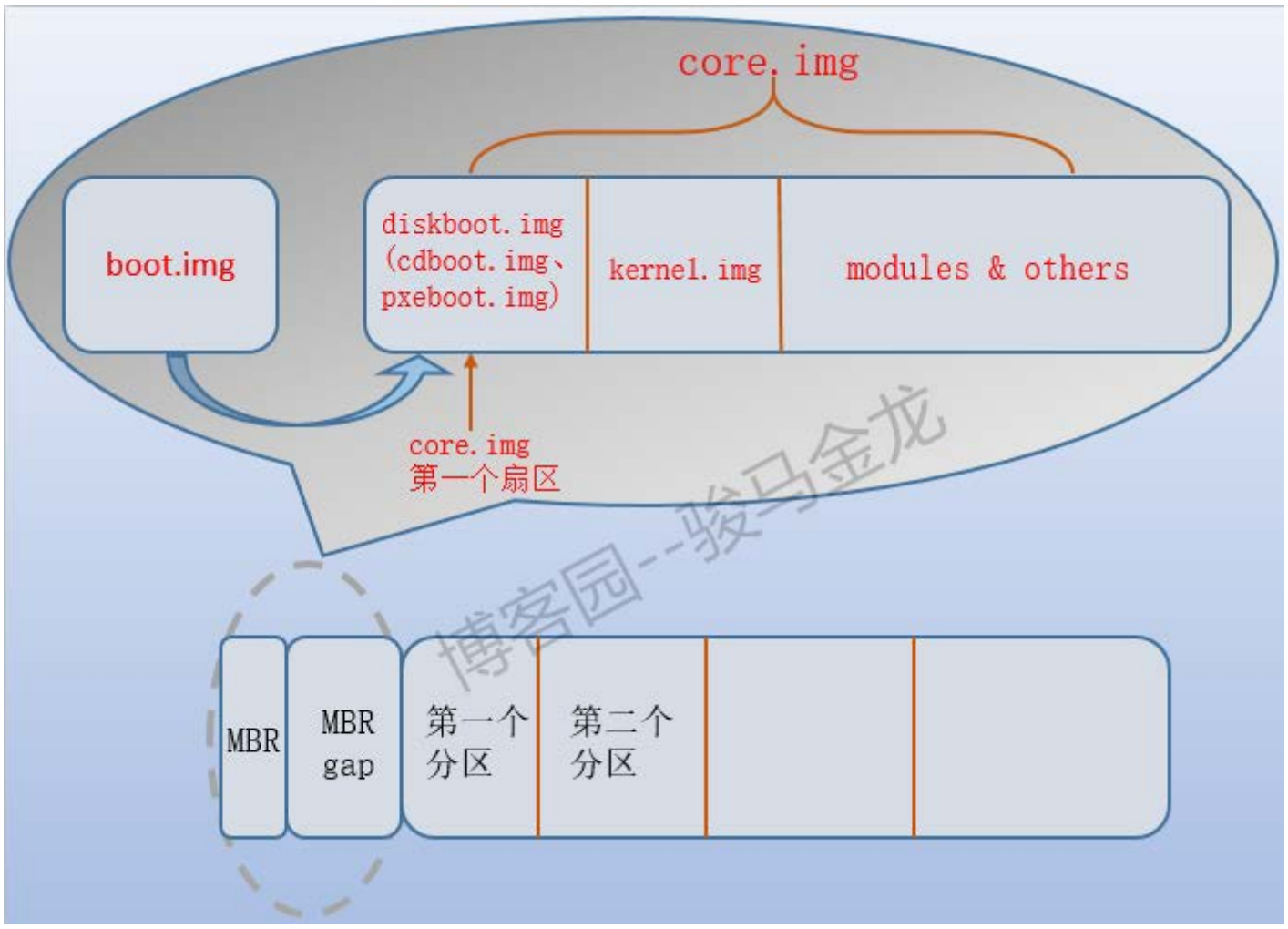
grub2 程序安装 grub 后，会在/boot/grub2/i386-pc/目录下生成 boot.img 和 core.img 文件，另外还有一些模块文件，其中包括文件系统类的模块。

```
[root@xuexi ~]# find /boot/grub2/i386-pc/ -name '*.img' -o -name "*fs.mod" -o -name "*ext[0-9].mod"
/boot/grub2/i386-pc/affs.mod
/boot/grub2/i386-pc/afs.mod
/boot/grub2/i386-pc/bfs.mod
/boot/grub2/i386-pc/btrfs.mod
/boot/grub2/i386-pc/cbfs.mod
/boot/grub2/i386-pc/ext2.mod   # ext2、ext3 和 ext4 都使用该模块
/boot/grub2/i386-pc/hfs.mod
/boot/grub2/i386-pc/jfs.mod
/boot/grub2/i386-pc/ntfs.mod
/boot/grub2/i386-pc/procfs.mod
/boot/grub2/i386-pc/reiserfs.mod
/boot/grub2/i386-pc/romfs.mod
/boot/grub2/i386-pc/sfs.mod
/boot/grub2/i386-pc/xfs.mod
/boot/grub2/i386-pc/zfs.mod
/boot/grub2/i386-pc/core.img
/boot/grub2/i386-pc/boot.img
```

其中 boot.img 就是安装在 MBR 中的 boot loader。当然，它们的内容是不一样的，安装 boot loader 时 grub2-install 会将 boot.img 转换为合适的汇编代码写入 MBR 中的 boot loader 部分。

core.img 是第二段 Boot loader 段，grub2-install 会将 core.img 转换为合适的汇编代码写入到紧跟在 MBR 后面的空间，这段空间是 MBR 之后、第一个分区之前的空闲空间，被称为 MBR gap，这段空间最小 31KB，但一般都会是 1MB 左右。

实际上，core.img 是多个 img 文件的结合体。它们的关系如下图：

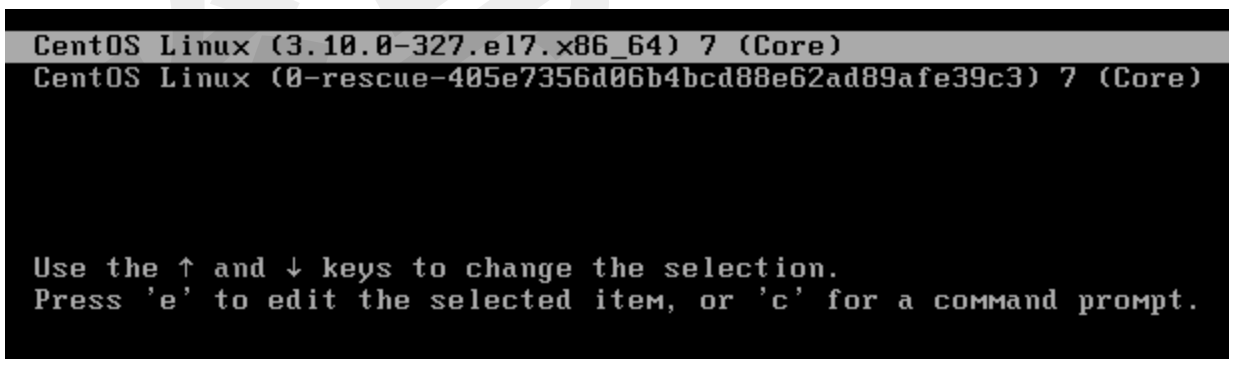


这张图解释了开机过程中 grub2 阶段的所有过程，boot.img 段的 boot loader 只有一个作用，就是跳转到 core.img 对应的 boot loader 的第一个扇区，对于从硬盘启动的系统来说，该扇区是 diskboot.img 的内容，diskboot.img 的作用是加载 core.img 中剩余的内容。

由于 diskboot.img 所在的位置是以硬编码的方式写入到 boot.img 中的，所以 boot.img 总能找到 core.img 中 diskboot.img 的位置并跳转到它身上，随后控制权交给 diskboot.img。随后 diskboot.img 加载压缩后的 kernel.img(注意，是 grub 的 kernel 不是操作系统的 kernel)以初始化 grub 运行时的各种环境，控制权交给 kernel.img。

但直到目前为止，core.img 都还不识别/boot 所在分区的文件系统，所以 kernel.img 初始化 grub 环境的过程就包括了加载模块，严格地说不是加载，因为在安装 grub 时，文件系统类的模块已经嵌入到了 core.img 中，例如 ext 类的文件系统模块 ext2.mod。

加载了模块后，kernel.img 就能识别/boot 分区的文件系统，也就能找到 grub 的配置文件/boot/grub2/grub.cfg，有了 grub.cfg 就能显示启动菜单，我们就能自由的选择要启动的操作系统。



当选择某个菜单项后，kernel.img 会根据 grub.cfg 中的配置加载对应的操作系统内核(/boot 目录下 vmlinuz 开头的文件)，并向操作系统内核传递启动时参数，包括根文件系统所在的分区，init ramdisk(即 initrd 或 initramfs)的路径。例如下面是某个菜单项的配置：

```
menuentry 'CentOS 6' --unrestricted {
        search --no-floppy --fs-uuid --set=root f5d8939c-4a04-4f47-a1bc-1b8cbabc4d32
        linux16 /vmlinuz-2.6.32-504.el6.x86_64 root=UUID=edb1bf15-9590-4195-aa11-6dac45c7f6f3 ro quiet
        initrd16 /initramfs-2.6.32-504.el6.x86_64.img
}
```

加载完操作系统内核后 grub2 就将控制权交给操作系统内核。

总结下，从 MBR 开始后的过程是这样的：

1.执行 MBR 中的 boot loader(即 boot.img)跳转到 diskboot.img。

2.执行 diskboot.img，加载 core.img 剩余的部分，并跳转到 kernel.img。

3.kernel.img 读取/boot/grub2/grub2.cfg，并显示启动管理菜单。

4.选中某菜单后，kernel.img 加载该菜单项配置的操作系统内核/boot/vmlinux-XXX，并传递内核启动参数，包括根文件系统所在分区和 init ramdisk 的路径。

5.控制权交给操作系统内核。

14.3.2 使用传统 grub 时的启动过程

传统 grub 对应的 boot loader 是 stage1 和 stage2，从 stage1 跳转到 stage2 大多数情况下还会用到 stage1_5 对应的 boot loader。

与 grub2 相比，stage1 和 boot.img 的作用是类似的，都在 MBR 中。当该段 boot loader 执行后，它的目的是跳转到 stage1_5 的第一个扇区上，然后由该扇区的代码加载剩余的内容，并跳转到 stage2 的第一个扇区上。

stage1_5 存在的理由是因为 stage2 功能较多，导致其文件体积较大(一般至少都有 100 多 K)，所以并没有像 core.img 一样嵌入到磁盘上，而是简单地将其放在了 boot 分区上，但 stage1 并不识别 boot 分区的文件系统类型，所以借助中间的辅助 boot loader 即 stage1_5 来跳转。

stage1_5 的目的之一是识别文件系统，但文件系统的类型有很多，所以对应的 stage1_5 也有很多种。

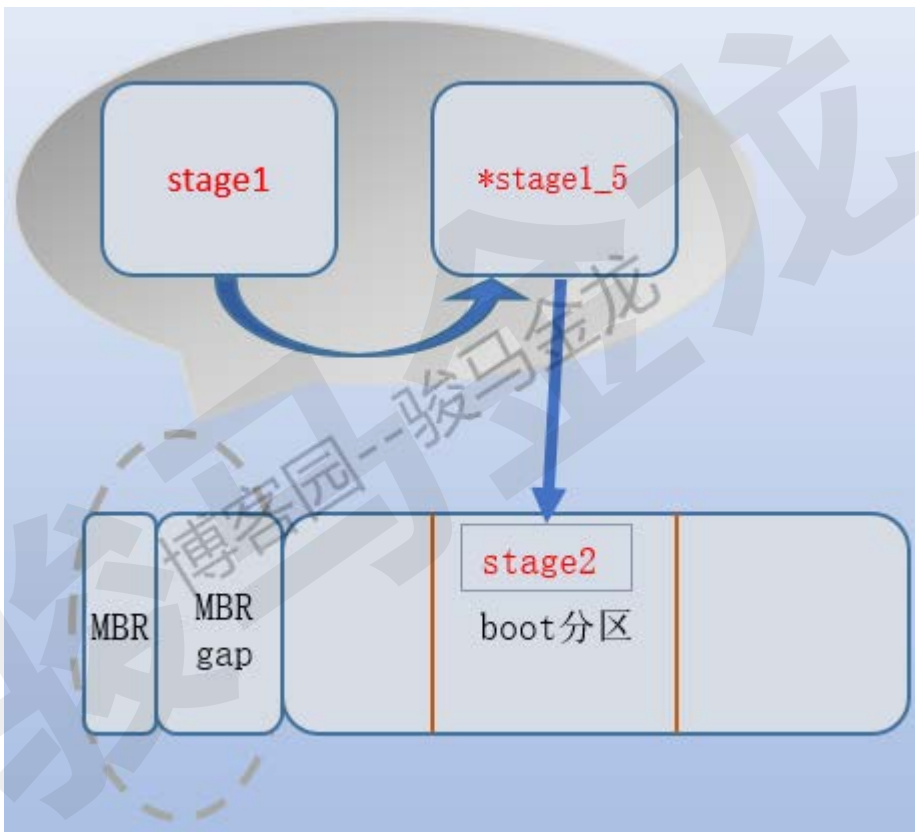
```
[root@xuexi ~]# ls -C /boot/grub/*stage1_5*
/boot/grub/e2fs_stage1_5      /boot/grub/jfs_stage1_5       /boot/grub/vstafs_stage1_5
/boot/grub/fat_stage1_5       /boot/grub/minix_stage1_5     /boot/grub/xfs_stage1_5
/boot/grub/ffs_stage1_5       /boot/grub/reiserfs_stage1_5
/boot/grub/iso9660_stage1_5   /boot/grub/ufs2_stage1_5
```

虽然有很多种 stage1_5，但每个 boot 分区也只能对应一种 stage1_5。这个 stage1_5 对应的 boot loader 一般会被嵌入到 MBR 后、第一个分区前的中间那段空间(即 MBR gap)。

当执行了 stage1_5 对应的 boot loader 后，stage1_5 就能识别出 boot 所在的分区，并找到 stage2 文件的第一个扇区，然后跳转过去。

当控制权交给了 stage2，stage2 就能加载 grub 的配置文件/boot/grub/grub.conf 并显示菜单并初始化 grub 的运行时环境，当选中操作系统后，stage2 将和 kernel.img 一样加载操作系统内核，传递内核启动参数，并将控制权交给操作系统内核。

所以，stage1、stage1_5 和 stage2 之间的关系如下图：



虽然绝大多数都提供了 stage1_5，但它不是必须的，它的作用仅仅只是识别 boot 分区的文件系统类型，对于一个会编程的人来说，可以将固定 boot 分区的文件系统识别代码嵌入到 stage1 中，这样 stage1 自身就能识别 boot 分区，就不需要 stage1_5 了。

看看安装 grub 时，grub 到底做了些什么工作。

```
grub> setup (hd0)
 Checking if "/boot/grub/stage1" exists... yes
 Checking if "/boot/grub/stage2" exists... yes
 Checking if "/boot/grub/e2fs_stage1_5" exists... yes
 Running "embed /boot/grub/e2fs_stage1_5 (hd0)"...  15 sectors are embedded.
succeeded
 Running "install /boot/grub/stage1 (hd0) (hd0)1+15 p (hd0,0)/boot/grub/stage2 /boot/grub/menu.lst"... succeeded
Done.
```

首先检测各 stage 文件是否存在于/boot/grub 目录下，随后嵌入 stage1_5 到磁盘上，该文件系统类型的 stage1_5 占用了 15 个扇区，最后安装 stage1，并告知 stage1 stage1_5 的位置是第 1 到第 15 个扇区，之所以先嵌入 stage1_5 再嵌入 stage1 就是为了让 stage1 知道 stage1_5 的位置，最后还告知了 stage1 stage2 和配置文件 menu.lst(它是 grub.conf 的软链接)的路径。

14.4 内核加载阶段

提前说明，下文所述均为 sysV init 系统启动风格，systemd 的启动管理方式大不相同，所以不要将 systemd 管理的启动方式与此做比较。

到目前为止，内核已经被加载到内存掌握了控制权，且收到了 boot loader 最后传递的内核启动参数以及 init ramdisk 的路径。

所有的内核都是以 bzImage 方式压缩过的，压缩后 CentOS 6 的内核大小大约为 4M，CentOS 7 的内核大小大约为 5M。内核要能正常运作下去，它需要进行解压释放。

解压释放之后，将创建 pid 为 0 的 idle 进程，该进程非常重要，后续内核所有的进程都是通过 fork 它创建的，且很多 cpu 降温工具就是强制执行 idle 进程来实现的。

然后创建 pid=1 和 pid=2 的内核进程。pid=1 的进程也就是 init 进程，pid=2 的进程是 kthread 内核线程，它的作用是在真正调用 init 程序之前完成内核环境初始化和设置工作，例如根据 grub 传递的内核启动参数找到 init ramdisk 并加载。

所谓的**救援模式**就是刚加载完内核，init 进程接收到控制权的那一阶段，因为没有进行任何操作系统初始化过程，所以可以修复和操作系统相关的很多问题。另外，安装镜像中也有内核，可以通过安装镜像进入救援模式，这种进入救援模式的方式几乎可修复任何操作系统启动相关的问题，即使是/boot 目录下内核镜像缺失都可以重装。(还有一种单用户模式，它是运行级别为 1 的环境，所以已经初始化完运行级别，见后文)

14.4.1 加载 init ramdisk

在前面，已经创建了 pid=1 的 init 进程和 pid=2 的 kthread 进程，但注意，它们都是内核线程，全称应该是 kernel_init 和 kernel_kthread，而真正能被 ps 捕获到的 pid=1 的 init 进程是由 kernel_init 调用 init 程序后形成的。

要加载/sbin/init 程序，首先要找到根分区，根分区是有文件系统的，所以内核需要先识别文件系统并加载文件系统的驱动，但文件系统的驱动又是放在根分区的，这就出现了先有鸡还是先有蛋的矛盾。

解决的方法之一是像 grub2 识别 boot 分区的文件系统一样，将根文件系统驱动模块嵌入到内核中，但文件系统的种类太多，而且会升级，这样就导致内核不断的嵌入新的文件系统驱动模块，内核不断增大，这显然是不合适的。

解决方法之二则像传统 grub 借助中间过渡引导段 stage1_5 一样，将根文件系统的驱动模块放入一个中间过渡文件，在加载根文件系统之前先加载这个过渡文件，再由过渡文件跳转到根文件系统。

方法二正是现在采用的，其采用的中间过渡文件称为 init ramdisk，它是在安装完操作系统时生成的，这样它会收集到当前操作系统的根文件系统是什么类型的文件系统，也就能只嵌入一个对应的文件系统驱动模块使其变得足够小。

在 CentOS 5 上采用的 init ramdisk 称为 initrd，而 CentOS 6 和 CentOS 7 采用的则是 initramfs，它们的目的是一样的，但在实现上却大有不同。但它们都存放在/boot 目录下。

```
[root@xuexi ~]# ll -h /boot/init*
-rw-------. 1 root root 19M Feb 25 11:53 /boot/initramfs-2.6.32-504.el6.x86_64.img
```

可以看到，它们的大小有十多兆，由此也可知道 init ramdisk 的作用肯定不仅仅只是找到根文件系统，它还会做其他工作。具体还做什么工作，请继续阅读下文。

14.4.2 initrd

initrd 其实是一个镜像文件系统，是在内存中划分一片区域模拟磁盘分区，在该文件中包含了找到根文件系统的脚本和驱动。

既然是文件系统，那么内核也必须要带有对应文件系统的驱动，另外文件系统要使用就必须有根"/"，这个根是内存中的"虚根"。由于内核加载到这里已经初始化一些运行环境了，所以**内核的运行状态等参数也要保存下来，保存的位置就是内存中虚根下的/proc 和/sys，此外还有收集到的硬件设备信息以及设备的运行环境也要保存下来，保存的位置是/dev**。到此为止，pid=2 的内核线程 kernel_kthread 就完成了基本工作，开始转到 kernel_init 进程上了。

再之后就是 kernel_init 挂载真正的根文件系统并从虚根切换到实根，最后 kernel_init 将调用 init 程序，也就是真正的 pid=1 的 init 进程，然后将控制权交给 init，所以从现在开始，将切换到用户空间，后续剩余的事情都将由用户空间的程序完成。

以下是 CentOS 5.8 中 initrd 文件的解压过程和解包后的目录结构。

```
[root@localhost ~]# cp /boot/initrd-2.6.18-308.el5.img /tmp/initrd.gz
[root@localhost tmp]# gunzip initrd.gz
[root@localhost tmp]# cpio -id < initrd
[root@localhost tmp]# ls
bin  dev  etc  init  initrd  lib  proc  sbin  sys  sysroot
```

14.4.3 initramfs

initramfs 比 initrd 又先进了一些，initrd 必须是一个文件系统，是在内存中模拟出磁盘分区的，所以内核必须要带有它的文件系统驱动，而 initramfs 则仅仅只是一个镜像压缩文件而非文件系统，所以它不需要带文件系统驱动，在加载时，内核会将其解压的内容装入到一个 tmpfs 中。

initramfs 和 initrd 最大的区别在于 init 进程的区别对待。**initramfs 为了尽早进入用户空间，它将 init 程序集成到了 initramfs 镜像文件中，这样就可以在 initramfs 装入 tmpfs 时直接运行 init 进程，而不用去找根文件系统下的/sbin/init，**由此挂载根文件系统的工作将由 init 来完成，而不再是内核线程 kernel_init 完成。最后从虚根切换到实根。

那根分区下的/sbin/init 是干嘛的呢？可以认为是 init ramdisk 中 init 的一个备份，如果 ramdisk 中找不到 init 就会去找/sbin/init。另外，在正常运行的操作系统环境下，/sbin/init 还经常用来完成其他工作，如发送信号。

其实 initramfs 完成了很多工作，解开它的镜像文件就能发现它的目录结构和真实环境下的目录结构类似。以下是 CentOS 7 上 initramfs-3.10.0-327.el7.x86_64 解包过程和解包后的目录结构。

```
[root@xuexi ~]# cp /boot/initramfs-3.10.0-327.el7.x86_64.img /tmp/initramfs.gz
[root@xuexi ~]# cd /tmp; gunzip /tmp/initramfs.gz
[root@xuexi tmp]# cpio -id < initramfs
[root@xuexi tmp]# ls -l
total 8
lrwxrwxrwx  1 root root    7 Jun 29 23:28 bin -> usr/bin
drwxr-xr-x  2 root root   42 Jun 29 23:28 dev
drwxr-xr-x 11 root root 4096 Jun 29 23:28 etc
lrwxrwxrwx  1 root root   23 Jun 29 23:28 init -> usr/lib/systemd/systemd
lrwxrwxrwx  1 root root    7 Jun 29 23:28 lib -> usr/lib
lrwxrwxrwx  1 root root    9 Jun 29 23:28 lib64 -> usr/lib64
drwxr-xr-x  2 root root    6 Jun 29 23:28 proc
drwxr-xr-x  2 root root    6 Jun 29 23:28 root
drwxr-xr-x  2 root root    6 Jun 29 23:28 run
lrwxrwxrwx  1 root root    8 Jun 29 23:28 sbin -> usr/sbin
-rwxr-xr-x  1 root root 3041 Jun 29 23:28 shutdown
drwxr-xr-x  2 root root    6 Jun 29 23:28 sys
drwxr-xr-x  2 root root    6 Jun 29 23:28 sysroot
drwxr-xr-x  2 root root    6 Jun 29 23:28 tmp
drwxr-xr-x  7 root root   61 Jun 29 23:28 usr
drwxr-xr-x  2 root root   27 Jun 29 23:28 var
```

另外，还可以在其 sbin 目录下发现 init 程序。

```
[root@xuexi tmp]# ll sbin/init
lrwxrwxrwx 1 root root 22 Jun 29 23:28 sbin/init -> ../lib/systemd/systemd
```

14.5 操作系统初始化

下文解释的是 sysV 风格的系统环境，与 systemd 初始化大不相同。

当 init 进程掌握控制权后，意味着已经进入了用户空间，后续的事情也将以用户空间为主导来完成。

init 的名称是 initialize 的缩写，是初始化的意思，所以它的作用也就是初始化的作用。在内核加载阶段，也有初始化动作，初始化的环境是内核的环境，是由 kernel_init、kernel_thread 等内核线程完成的。而 init 掌握控制权后，已经可以和用户空间交互，意味着真正的开始进入操作系统，所以它初始化的是操作系统的环境。

操作系统初始化涉及了不少过程，大致如下：读取运行级别；初始化系统类的环境；根据运行级别初始化用户类的环境；执行 rc.local 文件完成用户自定义开机要执行的命令；加载终端；

14.5.1 运行级别

在 sysV 风格的系统下，使用了运行级别的概念，不同运行级别初始化不同的系统类环境，你可以认为 windows 的安全模式也是使用运行级别的一种产物。

在 Linux 系统中定义了 7 个运行级别，使用 0-6 的数字表示。

```
0：halt，即关机
1：单用户模式
2：不带 NFS 的多用户模式
```

```
3：完整多用户模式
4：保留未使用的级别
5：X11，即图形界面模式
6：reboot，即重启
```

实际上，执行关机或重启命令的本质就是向 init 进程传递 0 或 6 这两个运行级别。

sysV 的 init 程序读取/etc/inittab 文件来获取默认的运行级别，并根据此文件所指定的配置执行默认运行级别对应的操作。注意，systemd 管理的系统是没有/etc/inittab 文件的，即使有也仅仅只是出于提醒的目的，因为 systemd 没有了运行级别的概念，说实话，systemd 管的真的太多了。

CentOS 6.6 上该文件内容如下：

```
[root@xuexi ~]# cat /etc/inittab
# inittab is only used by upstart for the default runlevel.
#
# ADDING OTHER CONFIGURATION HERE WILL HAVE NO EFFECT ON YOUR SYSTEM.
#
# System initialization is started by /etc/init/rcS.conf
#
# Individual runlevels are started by /etc/init/rc.conf
#
# Ctrl-Alt-Delete is handled by /etc/init/control-alt-delete.conf
#
# Terminal gettys are handled by /etc/init/tty.conf and /etc/init/serial.conf,
# with configuration in /etc/sysconfig/init.
#
# For information on how to write upstart event handlers, or how
# upstart works, see init(5), init(8), and initctl(8).
#
# Default runlevel. The runlevels used are:
#   0 - halt (Do NOT set initdefault to this)
#   1 - Single user mode
#   2 - Multiuser, without NFS (The same as 3, if you do not have networking)
#   3 - Full multiuser mode
#   4 - unused
#   5 - X11
#   6 - reboot (Do NOT set initdefault to this)
#
id:3:initdefault:
```

该文件告诉我们，**系统初始化过程由/etc/init/rcS.conf 完成，运行级别类的初始化过程由/etc/init.conf 来完成，按下 CTRL+ALT+DEL 键要执行的过程由/etc/init/control-alt-delete.conf 来完成，终端加载的过程由/etc/init/tty.conf 和/etc/init/serial.conf 读取配置文件/etc/sysconfig/init 来完成。**再文件最后，还有一行"id:3:initdefault"，表示默认的运行级别为 3，即完整的多用户模式。

确认了要进入的运行级别后，init 将先读取/etc/init/rcS.conf 来完成系统环境类初始化动作，再读取/etc/init/rc.conf 来完成运行级别类动作。

14.5.2 系统环境初始化

先看看/etc/init/rcS.conf 文件的内容。

```
[root@xuexi ~]# cat /etc/init/rcS.conf
# rcS - runlevel compatibility
#
# This task runs the old sysv-rc startup scripts.
#
# Do not edit this file directly. If you want to change the behaviour,
# please create a file rcS.override and put your changes there.

start on startup

stop on runlevel

task

# Note: there can be no previous runlevel here, if we have one it's bad
# information (we enter rc1 not rcS for maintenance).  Run /etc/rc.d/rc
# without information so that it defaults to previous=N runlevel=S.
console output
pre-start script
        for t in $(cat /proc/cmdline); do
```

```
                case $t in
                        emergency)
                                start rcS-emergency
                                break
                        ;;
                esac
        done
end script
exec /etc/rc.d/rc.sysinit
post-stop script
        if [ "$UPSTART_EVENTS" = "startup" ]; then
                [ -f /etc/inittab ] && runlevel=$(/bin/awk -F ':' '$3 == "initdefault" && $1 !~ "^#" { print $2 }' /etc/inittab)
                [ -z "$runlevel" ] && runlevel="3"
                for t in $(cat /proc/cmdline); do
                        case $t in
                                -s|single|S|s) runlevel="S" ;;
                                [1-9])       runlevel="$t" ;;
                        esac
                done
                exec telinit $runlevel
        fi
end script
```

其中"exec /etc/rc.d/rc.sysinit"这一行就表示要执行/etc/rc.d/rc.sysinit 文件，该文件定义了系统初始化(system initialization)的内容，包括：

(1).确认主机名。

(2).挂载/proc 和/sys 等特殊文件系统，使得内核参数和状态可与人进行交互。是否还记得在内核加载阶段时的/proc 和/sys?

(3).启动 udev，也就是启动类似 windows 中的设备管理器。

(4)初始化硬件参数，如加载某些驱动，设置时钟等。

(5).设置主机名。

(6).执行 fsck 检测磁盘是否健康。

(7).挂载/etc/fstab 中除/proc 和 NFS 的文件系统。

(8).激活 swap。

(9).将所有执行的操作写入到/var/log/dmesg 文件中。

14.5.3 运行级别环境初始化

执行完系统初始化后，接下来就是执行运行级别的初始化。先看看/etc/init/rc.conf 的内容。

```
[root@xuexi ~]# cat /etc/init/rc.conf
# rc - System V runlevel compatibility
#
# This task runs the old sysv-rc runlevel scripts.  It
# is usually started by the telinit compatibility wrapper.
#
# Do not edit this file directly. If you want to change the behaviour,
# please create a file rc.override and put your changes there.

start on runlevel [0123456]

stop on runlevel [!$RUNLEVEL]

task

export RUNLEVEL
console output
exec /etc/rc.d/rc $RUNLEVEL
```

最后一行"exec /etc/rc.d/rc $RUNLEVEL"说明调用/etc/rc.d/rc 这个脚本来初始化指定运行级别的环境。Linux 采用了将各运行级别初始化内容分开管理的方式，将 0-6 这 7 个运行级别要执行的初始化脚本分别放入 rc[0-6].d 这 7 个目录中。

```
[root@xuexi ~]# ls -l /etc/rc.d/
total 60
drwxr-xr-x. 2 root root  4096 Jun 11 02:42 init.d
```

```
-rwxr-xr-x. 1 root root  2617 Oct 16  2014 rc
drwxr-xr-x. 2 root root  4096 Jun 11 02:42 rc0.d
drwxr-xr-x. 2 root root  4096 Jun 11 02:42 rc1.d
drwxr-xr-x. 2 root root  4096 Jun 11 02:42 rc2.d
drwxr-xr-x. 2 root root  4096 Jun 11 02:42 rc3.d
drwxr-xr-x. 2 root root  4096 Jun 11 02:42 rc4.d
drwxr-xr-x. 2 root root  4096 Jun 11 02:42 rc5.d
drwxr-xr-x. 2 root root  4096 Jun 11 02:42 rc6.d
-rwxr-xr-x. 1 root root   220 Oct 16  2014 rc.local
-rwxr-xr-x. 1 root root 19914 Oct 16  2014 rc.sysinit
```

实际上/etc/init.d/下的脚本才是真正的脚本，放入 rcN.d 目录中的文件只不过是/etc/init.d/目录下脚本的软链接。注意，/etc/init.d 是 Linux 耍的一个小把戏，它是/etc/rc.d/init.d 的一个符号链接，在有些类 unix 系统中是没有/etc/init.d 的，都是直接使用/etc/rc.d/init.d。

以/etc/rc.d/rc3.d 为例。

```
[root@xuexi ~]# ll /etc/rc.d/rc3.d/ | head
total 0
lrwxrwxrwx. 1 root root 16 Feb 25 11:52 K01smartd -> ../init.d/smartd
lrwxrwxrwx. 1 root root 16 Feb 25 11:52 K10psacct -> ../init.d/psacct
lrwxrwxrwx. 1 root root 19 Feb 25 11:51 K10saslauthd -> ../init.d/saslauthd
lrwxrwxrwx  1 root root 22 Jun 10 08:59 K15htcacheclean -> ../init.d/htcacheclean
lrwxrwxrwx  1 root root 15 Jun 10 08:59 K15httpd -> ../init.d/httpd
lrwxrwxrwx  1 root root 15 Jun 11 02:42 K15nginx -> ../init.d/nginx
lrwxrwxrwx. 1 root root 18 Feb 25 11:52 K15svnserve -> ../init.d/svnserve
lrwxrwxrwx. 1 root root 20 Feb 25 11:51 K50netconsole -> ../init.d/netconsole
lrwxrwxrwx  1 root root 17 Jun 10 00:50 K73winbind -> ../init.d/winbind
```

可见，rcN.d 中的文件都以 K 或 S 加一个数字开头，其后才是脚本名称，且它们都是/etc/rc.d/init.d 中文件的链接。S 开头表示进入该运行级别时要运行的程序，S 字母后的数值表示启动顺序，数字越大，启动的越晚；K 开头的表示退出该运行级别时要杀掉的程序，数值表示关闭的顺序。

所有这些文件都是由/etc/rc.d/rc 这个程序调用的，K 开头的则传给 rc 一个 stop 参数，S 开头的则传给 rc 一个 start 参数。

打开 rc0.d 和 rc6.d 这两个目录，你会发现在这两个目录中除了"S00killall"和"S01reboot"，其余都是 K 开头的文件。

而**在 rc[2-5].d 这几个目录中，都有一个 S99local 文件，且它们都是指向/etc/rc.d/rc.local 的软链接**。S99 表示最后启动的一个程序，所以 rc.local 中的程序是 2345 这 4 个运行级别初始化过程中最后运行的一个脚本。这是 Linux 提供给我们定义自己想要在开机时(严格地说是进入运行级别)就执行的命令的文件。

当初始化完运行级别环境后，将要准备登录系统了。

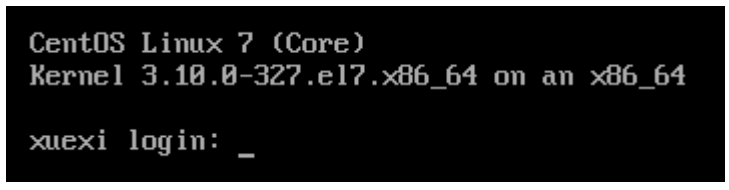
所谓的**单用户模式**(runlevel=1)，就是初始化完运行级别 1 对应的环境。因为已经初始化了操作系统和运行级别，所以单用户模式所处的层次要比救援模式高的多，能修复的问题也就只有它后面还未初始化的过程：终端初始化和用户登录问题。

14.6 终端初始化和登录系统

Linux 是多任务多用户的操作系统，它允许多人同时在线工作。但每个人都必须要输入用户名和密码才能验证身份并最终登录。但登陆时是以图形界面的方式给用户使用，还是以纯命令行模式给用户使用呢？这是终端决定的，也就是说在登录前需要先加载终端。至于什么是终端，见：http://www.cnblogs.com/f-ck-need-u/p/7099578.html。

14.6.1 终端初始化

在 Linux 上，每次开机都必然会开启所有支持的虚拟终端，如下图。



这些虚拟终端是由 getty 命令(get tty)来完成的，getty 命令有很多变种，有 mingetty、agetty、rungettty 等，在 CentOS 5 和 CentOS 6 都使用 mingetty，在 CentOS 7 上使用 agetty。getty 命令的作用之一是调用登录程序/bin/login。

例如，在 CentOS 6 下，捕获 tty 终端情况。

```
[root@xuexi ~]# ps -elf | grep tt[y]
4 S root       1412      1  0  80   0 -  1016 n_tty_ Jun21 tty2     00:00:00 /sbin/mingetty /dev/tty2
4 S root       1414      1  0  80   0 -  1016 n_tty_ Jun21 tty3     00:00:00 /sbin/mingetty /dev/tty3
4 S root       1417      1  0  80   0 -  1016 n_tty_ Jun21 tty4     00:00:00 /sbin/mingetty /dev/tty4
4 S root       1419      1  0  80   0 -  1016 n_tty_ Jun21 tty5     00:00:00 /sbin/mingetty /dev/tty5
4 S root       1421      1  0  80   0 -  1016 n_tty_ Jun21 tty6     00:00:00 /sbin/mingetty /dev/tty6
4 S root       1492   1410  0  80   0 - 27118 n_tty_ Jun21 tty1     00:00:00 -bash
```

在 CentOS 7 下，捕获 tty 终端情况。

```
[root@xuexi tmp]# ps -elf | grep tt[y]
4 S root       8258      1  0  80   0 - 27507 n_tty_ 04:17 tty2     00:00:00 /sbin/agetty --noclear tty2 linux
4 S root       8259      1  0  80   0 - 27507 n_tty_ 04:17 tty3     00:00:00 /sbin/agetty --noclear tty3 linux
4 S root       8260      1  0  80   0 - 27507 n_tty_ 04:17 tty4     00:00:00 /sbin/agetty --noclear tty4 linux
4 S root       8262    915  0  80   0 - 29109 n_tty_ 04:17 tty1     00:00:00 -bash
4 S root       8307   8305  0  80   0 - 29109 n_tty_ 04:17 tty5     00:00:00 -bash
4 S root       8348   8346  0  80   0 - 29136 n_tty_ 04:17 tty6     00:00:00 -bash
```

细心一点会发现，有的 tty 终端仍然以/sbin/mingetty 进程或/sbin/agetty 进程显示，有些却以 bash 进程显示。这是因为 getty 进程在调用 /bin/login 后，如果输入用户名和密码成功登录了某个虚拟终端，那么 gettty 程序会融合到 bash(假设 bash 是默认的 shell)进程，这样 getty 进程就不会再显示了。

虽然 getty 不显示了，但并不代表它消失了，它仍以特殊的方式存在着。是否还记得/etc/inittab 文件？此文件中提示了终端加载的过程由 /etc/init/tty.conf 读取配置文件/etc/sysconfig/init 来完成。

```
[root@xuexi ~]# grep tty -A 1 /etc/inittab
# Terminal gettys are handled by /etc/init/tty.conf and /etc/init/serial.conf,
# with configuration in /etc/sysconfig/init.
```

那么就看看/etc/init/tty.conf 文件。

```
[root@xuexi ~]# cat /etc/init/tty.conf
# tty - getty
#
# This service maintains a getty on the specified device.
#
# Do not edit this file directly. If you want to change the behaviour,
# please create a file tty.override and put your changes there.

stop on runlevel [S016]

respawn
instance $TTY
exec /sbin/mingetty $TTY
usage 'tty TTY=/dev/ttyX  - where X is console id'
```

此文件中的 respawn 表示进程由 init 进程监视，即使被杀掉了也会由 init 来重启它。所以，只要 getty 进程一结束，init 会立即监视到而重启该进程。因此，用户登录成功后 getty 只是融合到了 bash 进程中，并非退出，否则 init 会立即重启它，而它会调用 login 程序让你再次输入用户和密码。

再看看/etc/sysconfig/init 文件。

```
[root@xuexi ~]# cat /etc/sysconfig/init
# color => new RH6.0 bootup
# verbose => old-style bootup
# anything else => new style bootup without ANSI colors or positioning
BOOTUP=color
# column to start "[  OK  ]" label in
RES_COL=60
# terminal sequence to move to that column. You could change this
# to something like "tput hpa ${RES_COL}" if your terminal supports it
MOVE_TO_COL="echo -en \\033[${RES_COL}G"
# terminal sequence to set color to a 'success' color (currently: green)
SETCOLOR_SUCCESS="echo -en \\033[0;32m"
# terminal sequence to set color to a 'failure' color (currently: red)
SETCOLOR_FAILURE="echo -en \\033[0;31m"
# terminal sequence to set color to a 'warning' color (currently: yellow)
SETCOLOR_WARNING="echo -en \\033[0;33m"
# terminal sequence to reset to the default color.
SETCOLOR_NORMAL="echo -en \\033[0;39m"
# Set to anything other than 'no' to allow hotkey interactive startup...
PROMPT=yes
# Set to 'yes' to allow probing for devices with swap signatures
AUTOSWAP=no
# What ttys should gettys be started on?
ACTIVE_CONSOLES=/dev/tty[1-6]
# Set to '/sbin/sulogin' to prompt for password on single-user mode
# Set to '/sbin/sushell' otherwise
SINGLE=/sbin/sushell
```

其中 ACTIVE_CONSOLES 指令决定了要开启哪些虚拟终端。SINGLE 决定了在单用户模式下要调用哪个 login 程序和哪个 shell。

14.6.2 登录过程

如果不在虚拟终端登录，而是通过为 ssh 分配的伪终端登录，那么到创建完 getty 进程那一步其实开机流程已经完成了。但不管在哪种终端下登录，登录过程也可以算作开机流程的一部分，所以也简单说明下。

getty 进程启用虚拟终端后将调用 login 进程提示用户输入用户名或密码(或伪终端的连接程序如 ssh 提示输入用户名和密码)，当用户输入完成后，将验证输入的用户名是否合法，密码是否正确，用户名是否是明确被禁止登陆的，PAM 模块对此用户的限制是如何的等等，还要将登录过程记录到各个日志文件中。如果登录成功，将加载该用户的 bash，加载 bash 过程需要读取各种配置文件，初始化各种环境等等。但不管怎么说，只要登录成功就表示开机流程全部完成了。