Linux Notes

www.theqiong.com

2015 年 1 月 16 日

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

UNIX

1.1 Overview

最初在开发计算机的时候是希望可以辅助与简化人们进行大量的运算工作，后来才发展成为一些特殊用途，但是计算机基本的功能都是接受用户输入命令，经由 CPU 的算术与逻辑单元运算处理后，产生或存储成有用的信息。

为了实现这些功能，计算机就必须要有：

1. 输入单元：例如鼠标、键盘、读卡器等；
2. 中央处理器（CPU）：含有算术逻辑、控制、存储等单元；
3. 输出单元：例如显示器、打印机等

上面这些其实就是组成计算机的主要元件，而为了连接各个元件才有了主板，所以主机里面就包含了主板以及 CPU，还有各种需要的适配卡，再加上屏幕、键盘、鼠标等则通过与主机的连接，构成了一台可以运行的计算机。

整台主机的重点在于中央处理器（Central Processing Unit，CPU），CPU 是一个具有特定功能的芯片（Chip），里面还有微指令集。CPU 的工作主要在于管理和运算，因此 CPU 内又分为两个主要的单元——逻辑运算单元和运算器。

另外，由于计算机仅认识 0 和 1，因此计算机主要是以二进制的方式来计算的，因此通常计算机的运算/存储单位都是以 Byte 或 bits 为基本单位，换算关系如下：

```
1 Bytes = 8 bits
1 KB = 1024 Bytes
1 MB = 1024 KB
1 GB = 1024 MB
```

计算机也因为它的复杂度细分为不同等级。

- 超级计算机（Supercomputer）
  超级计算机是运行速度最快的电脑，但是维护、操作费用也最高，主要是用于需要有高速计算的环境中，例如国防军事、气象预测、太空科技等需要模拟的领域。
- 大型机（Mainframe Computer）
  大型计算机通常也具有数个高速的 CPU，功能上虽不及超级计算机，但也可用来处理大量数据与复杂的运算，例如大型企业的主机、全国性的证券交易所等每天需要处理数百万笔信息的企业机构等。
- 小型机（Minicomputer）
  小型机仍具有大型电脑同时支持多用户的特性，但是主机可以放在一般场所，通常用来作为科学研究、工程分析与工厂的流程管理等。
- 微机（Microcomputer）
  微机又可以称为个人计算机，具有体积最小、价格最低的特点，但功能还是都具备的，大致又可分为桌面型、笔记本型等。

虽然在目前个人电脑的使用广泛，但是在 1990 年以前，个人电脑是不被重视的，因为其运算速度在当时实在很慢，而且当时比较有名的操作系统也没有对个人电脑提供支持，所以不太流行。

UNIX 操作系统具有多任务、多用户的特征，于 1969 年由美国 AT&T 公司的贝尔实验室实现。

UNIX 的前身为 Multics，由 Bell Lab 的 Ken Thompson 和 Dennis Ritchie 等人通过简化 Multics 而实现的 UNIX 具有安全可靠，高效强大的特点，因而在服务器领域得到了广泛的应用，并成为了科学计算、大型机、超级计算机等主流操作系统，现在其仍然被应用于对稳定性要求极高的数据中心等环境中。

UNIX 最早由 Ken Thompson、Dennis Ritchie、Douglas McIlroy 和 Joe Ossanna 于 1969 年在 AT&T 贝尔实验室开发，并于 1971 年首次发布，最初是完全用汇编语言编写。后来，在 1973 年 UNIX 被 Dennis Ritchies 使用 C 语言重新实现（内核和 I/O 例外），从而使其具有高可用性的同时，更容易地移植到不同的计算机平台。

此后的 10 年，UNIX 在学术机构和大型企业中得到了广泛的应用，当时的 UNIX 拥有者 AT&T 公司以合理的许可将 UNIX 源码授权给学术机构做研究或教学用途，因此在源码基础上通过扩充和改进而产生了很多的“UNIX 变种”，同时这些“UNIX 变种”反过来又促进了 UNIX 的发展，其中最著名的变种之一是由加州大学伯克利分校开发的 BSD UNIX。

BSD 使用主版本加次版本的方法标识（如 4.2BSD、4.3BSD），在原始版本的基础上

还有派生版本，这些版本通常有自己的名字（如 4.3BSD-Net/1 和 4.3BSD-Net/2 等）。

后来，Richard Stallman 创建 GNU 项目来构建一个能够自由发布的类 UNIX 系统，GNU 项目不断发展壮大并包含了越来越多的内容，现在 GNU 项目开发的产品（比如 Emacs、GCC 等）已经成为其他自由发布的类 UNIX 产品中的核心应用。

在 AT&T 停止将 UNIX 源码授权给学术机构，并对之前的 UNIX 及其变种声明了版权权利后，BSD 被很多商业厂家采用并成为了很多商用 UNIX 的基础，后来接手 Bell Lab 的 Novell 采取了一种比较开明的做法，允许伯克利分校自由发布自己的 UNIX 变种，但是前提是必须将来自于 AT&T 的代码完全删除，于是诞生了 4.4 BSD Lite 版。4.4 BSD Lite 版本不存在法律问题，因此它成为了现代 BSD 的基础版本。

尽管后来非商业版的 UNIX 系统又经过了很多演变，但其中有不少最终都是创建在 BSD 版本上（Linux、Minix 等系统除外），因此 4.4 BSD 又是所有自由版本 UNIX 的基础，很多公司在取得了 UNIX 的授权之后开发了自己的 UNIX 产品，比如 IBM AIX、HP-UX、Sun Solaris 和 IRIX 等。

目前 UNIX 的商标权由国际开放标准组织（The Open Group，缩写为 TOG）所拥有，只有符合单一 UNIX 规范的 UNIX 系统才能使用 UNIX 这个名称，否则只能称为类 UNIX（UNIX-like），现在通过 UNIX 认证的操作系统包括 AIX、HP/UX、OS X、Reliant UNIX、SCO、Solaris、Tru64 UNIX、z/OS 等。

国际开放标准组织（又译为国际标准化组织）是以制定电脑架构的共通标准为目的而成立的国际性非营利组织，在英国登记注册。在 1996 年，由 X/Open 与开源软件基金会（Open Software Foundation）合组而成，拥有 UNIX 的商标权，并且制定和发布了单一 UNIX 规范（Single UNIX Specification）。

这里，单一 UNIX 规范（Single UNIX Specification，缩写为 SUS）是一套 UNIX 系统的统一规格书。它扩充了 POSIX 标准，定义了标准 UNIX 操作系统。最初 SUS 由 IEEE 与 The Open Group 所提出，目前由 Austin Group 负责维护。

- 1980 年代
  在 1980 年代中，开始提出计划想要统一不同 UNIX 操作系统的接口。
- 1988 年：POSIX
  1988 年，这些标准被汇整为 IEEE 1003（ISO/IEC 9945），也就是 POSIX。
- 1990 年代：Spec 1170
  通用应用程式接口规格（Common API Specification），又称为 Spec 1170。
- 1997 年：单一 UNIX 规范第二版
  单一 UNIX 规范第二版（Single UNIX Specification version 2）。
- 2001 年：POSIX:2001，单一 UNIX 规范第三版

- 2004 年：POSIX:2004
- 2008 年：POSIX:2008

1990 年，Linus Torvalds 决定编写一个自己的 Minix 内核，初名为 Linus' Minix（意为 Linus 的 Minix 内核），后来改名为 Linux。

Linux 内核于 1991 年正式发布，并逐渐引起人们的注意。当 GNU 软件与 Linux 内核结合后，GNU 软件构成了兼容 POSIX 操作系统 GNU/Linux 的基础，今天 GNU/Linux 已经成为发展最为活跃的自由/开放源码的操作系统。

1994 年，BSD UNIX 走上了复兴[1]的道路，其开发也走向了几个不同的方向，并最终产生了 FreeBSD、OpenBSD、NetBSD 和 DragonFlyBSD 的出现。

在 UNIX 操作系统的发展过程中，UNIX 也影响了用户的思考方式和看待世界的角度，例如 UNIX 提出了重要的操作系统设计原则等。

- 简洁至上（KISS 原则）
- 提供机制而非策略

从 1980 年代开始，由 IEEE 制定的开放的操作系统标准（POSIX）（ISO/IEC 9945）成为了 UNIX 系统的基础部分。

1.2 Histroy

早期的计算机都是用于军事或者是高科技用途以及学术研究，而且早期的计算机的输入设备只有卡片阅读机，输出设备只有打印机，用户也无法与操作系统交互，早期计算机的操作系统称为多道批处理操作系统。

开发人员在编写程序时，必须要将程序相关的信息在读卡纸上打孔，然后再将读卡纸插入读卡机来将信息输入主机中运算。后来可以使用键盘来进行信息的输入/输出，不过毕竟主机数量太少，只能是大家轮流等待使用。

在 1960 年代初期 MIT 开发了“兼容分时系统”（Compatible Time-Sharing System, CTSS），它可以让大型主机通过提供数个终端（terminal）以连线进入主机来使用主机的资源。

CTSS 可以说是近代操作系统的鼻祖，它可以让多个用户在某一段时间内分别使用 CPU 的资源，感觉上用户会觉得大家是同时使用该主机的资源，但是事实上是 CPU 在

[1] Although not released until 1992 due to legal complications, development of 386BSD, from which NetBSD, OpenBSD and FreeBSD descended, predated that of Linux. Linus Torvalds has said that if 386BSD had been available at the time, he probably would not have created Linux.
386BSD 因为法律问题直到 1992 年还没有发布，NetBSD 和 FreeBSD 是 386BSD 的后裔，早于 Linux。Linus Torvalds 曾说，当时如果有可用的 386BSD，他就可能不会编写 Linux。

每个用户的工作之间进行切换，但是在当时 CTSS 是划时代的技术。

无论主机在哪里，只要在终端进行操作就可利用主机提供的功能了，不过终端只具有输入/输出的功能，本身完全不具备任何运算或者软件安装的能力，而且比较先进的主机大概也只能提供不到 30 个终端机。

为了更加强化大型主机的系统，让主机的资源可以提供更多用户来使用，在 1965 年前后由 Bell Lab、MIT 及 GE 共同发起了 Multics[2]的计划，Multics 目的是想要让大型主机可以提供 300 个以上的终端连线使用。不过到了 1969 年前后，计划进度落后，资金也短缺，所以该计划就宣告失败[3]。

在认为 Multics 计划不可能成功之后，Bell Lab 就退出了该计划，不过原本参与 Multics 计划的人员中 Ken Thompson 因为自己的需要，希望开发一个小的操作系统以满足自己的需求，并在 DEC PDP-7 上以汇编语言开发了操作系统核心程序，同时包括一些核心工具程序以及一个文件系统——UNIX 的原型。

Thompson 将 Multics 庞大的复杂系统进行了简化，于是将这个操作系统为 Unics，Thompson 的文件系统提出了两个重要的概念，分别是：

1. 所有的程序或系统设备都是文件；
2. 不管构建编辑器还是附属文件，所写的程序只有一个目的，且要有效的完成目标。

这些概念在后来对于 Linux 的开发有相当重要的影响，并且 Thompson 编写的操作系统及其改版不断被传播和修改，不过其中比较重要的改版发生在 1973 年。

UNIX 本来是以汇编语言编写的，后来因为系统移植与效率的需求使用 B 语言进行了改写，不过效率依旧不是很好。后来 Dennis Ritchie 将 B 语言重新改写成 C 语言，C 语言可以在不同的机器上面运行，而且 Ritchie 等人再以 C 语言重新改写与编译了 Unics，最后开发出 UNIX 的正式版本。

在当时，以较高阶的 C 语言来实现的 UNIX 相对于汇编语言，与硬件平台的相关性很小，这个改变也使得 UNIX 很容易被移植到不同的机器上。

UNIX 的高度可移植性与效率使得很多商业公司也加入了 UNIX 操作系统的开发，例如 AT&T 自家的 System V、IBM 的 AIX 以及 HP 与 DEC 等公司，并且都推出了自家的主机搭配的 UNIX 操作系统。

操作系统的核心（kernel）必须要跟硬件配合以控制硬件的资源进行良好的工作，而在早期每一家生产计算机硬件的公司还没有所谓的“标准协议”的概念，所以每一个计算机公司生产的硬件自然就不相同，因此他们必须要为自己的计算机硬件开发合适的 UNIX 系统，例如 Sun、Cray 与 HP 就是这一种情况。

[2]注：Multics 有复杂、多数的意思

[3]最终 Multics 还是成功地开发出了他们的系统，参考http://www.multicians.org/

早期开发出来的 UNIX 操作系统以及内含的相关软件并没有办法在其他的硬件架构下工作的，而且由于 UNIX 强调的是多用户、多任务的环境，但是早期的 286 CPU 是没有能力进行多任务处理，因此没有厂商针对个人计算机设计 UNIX 系统，于是在早期并没有出现支持个人计算机的 UNIX 操作系统。

1979 年时 AT&T 推出的 System V 第七版 UNIX 可以支持 x86 架构的个人计算机系统，也就是说 System V 可以在个人计算机上安装和运行。

1.3 Standards

Beginning in the late 1980s, an open operating system standardization effort now known as POSIX provided a common baseline for all operating systems; IEEE based POSIX around the common structure of the major competing variants of the UNIX system, publishing the first POSIX standard in 1988. In the early 1990s, a separate but very similar effort was started by an industry consortium, the Common Open Software Environment (COSE) initiative, which eventually became the Single UNIX Specification administered by The Open Group. Starting in 1998, the Open Group and IEEE started the Austin Group, to provide a common definition of POSIX and the Single UNIX Specification.

In 1999, in an effort towards compatibility, several UNIX system vendors agreed on SVR4's Executable and Linkable Format (ELF) as the standard for binary and object code files. The common format allows substantial binary compatibility among UNIX systems operating on the same CPU architecture.

The Filesystem Hierarchy Standard was created to provide a reference directory layout for UNIX-like operating systems, and has mainly been used in Linux.

1.4 Components

本列表中的 UNIX 实用程序由 IEEE Std 1003.1-2008 定义，是单一 UNIX 规范（SUS）的一部分，这些实用程序可以在 UNIX 操作系统和绝大多数类 UNIX 操作系统中找到。

Table 1.1: IEEE 标准 1003.1-2008 实用程序

名称	分类	描述
admin	源代码控制系统	创建和管理源代码控制系统文件
alias	其他	定义或者显示别名
ar	其他	生成并维护函数库
asa	文字处理	Interpret carriage-control characters

名称	分类	描述
at	进程管理	在设定时间执行命令
awk	文字处理	模式扫描和处理语言
basename	文件系统	输入文件完整路径，只返回其文件名
batch	进程管理	按队列执行 at 命令
bc	其他	计算器编程语言
bg	进程管理	后台运行作业
c99	C 语言编程	标准 C 语言编译器
cal	其他	输出日历
cat	文件系统	连接和输出文件
cd	文件系统	改变工作目录
cflow	C 语言编程	生成 C 语言流程图
chgrp	文件系统	改变文件组拥有者
chmod	文件系统	改变文件权限
chown	文件系统	改变文件所有者
cksum	文件系统	计算文件校验和和大小
clear	文件系统	清除屏幕
cmp	文件系统	比较 2 个文件
comm	文字处理	按行比较两个已排序文件
command	Shell 编程	执行简单命令
compress	文件系统	压缩数据
cp	文件系统	复制文件
crontab	其他	设制定期运行的后台程序
csplit	文字处理	基于内容分割文件
ctags	C 语言编程	创建 C 语言的标记（tag）文件
cut	Shell 编程	选择文本中每行的特定区域
cxref	C 语言编程	生成 C 语言程序交叉引用表
date	其他	输出日期和时间
dd	文件系统	转换或复制文件
delta	源代码控制系统	为源代码控制系统生成差异文件
df	文件系统	报告磁盘剩余空间
diff	文字处理	比较 2 个文件
dirname	文件系统	返回路径的目录
du	文件系统	计算磁盘占用空间
echo	Shell 编程	输出命令参数到标准输出
ed	文字处理	标准文本编辑器
env	其他	为命令设置环境变量
ex	文字处理	文字编辑器
expand	文字处理	转换跳格为空格
expr	Shell 编程	计算表达式的值
false	Shell 编程	返回假值
fc	其他	处理命令行历史
fg	进程管理	在前台运行命令
file	文件系统	判断文件类型
find	文件系统	查找文件
fold	文字处理	回折每行文本到特定宽度
fort77	FORTRAN77 编程	FORTRAN 编译器

名称	分类	描述
fuser	进程管理	列出所有打开文件的进程的进程号
gencat	其他	生成一个格式化的消息目录
get	源代码控制系统	取得源代码控制系统文件某个版本
getconf	其他	查询系统配置变量
getopts	Shell 编程	解析命令行选项参数
grep	其他	根据模式搜索文字
hash	其他	提示或者报告程序位置
head	文字处理	显示文件开头几行
iconv	文字处理	转换字符集
id	其他	返回用户标示符
ipcrm	其他	删除消息队列，信号集或者共享内存段标识
ipcs	其他	显示进程间通信的状态
jobs	进程管理	显示当前会话中任务状态
join	文字处理	关系型数据库操作
kill	进程管理	结束进程或向进程发信号
lex	C 语言编程	为词法分析程器审查功能程序
link	文件系统	创建文件硬链接
ln	文件系统	创建文件链接
locale	其他	获得本地信息
localedef	其他	定义本地环境变量
logger	Shell 编程	记录消息日志
logname	其他	返回当前登陆用户名
lp	文字处理	发送文件到打印机
ls	文件系统	列出目录内容
m4	其他	宏处理器
mailx	其他	发送电子邮件
make	编程	维护一整套代码库，组织编译
man	其他	显示系统文档
mesg	其他	允许或者拒绝消息
mkdir	文件系统	创建目录
mkfifo	文件系统	生成 FIFO 类型文件
more	文字处理	逐页显示文件
mv	文件系统	移动文件
newgrp	其他	登陆到其他用户组
nice	进程管理	用新的 nice 值运行程序
nl	文字处理	加行号显示文本
nm	C 语言编程	显示目标文件的符号表
nohup	进程管理	运行一个忽略 SIGHUP 信号的程序
od	其他	将文件以八进制或其他进制输出
paste	文字处理	合并文件
patch	文字处理	将改变写入文件
pathchk	文件系统	检验路径名
pax	其他	Portable archive interchange
pr	文字处理	打印文件
printf	Shell 编程	格式化输出
prs	源代码控制系统	打印源代码控制系统文件

名称	分类	描述
ps	进程管理	报告进程状态
pwd	文件系统	输出当前目录
qalter	批处理实用程序	Alter 批处理任务
qdel	批处理实用程序	删除批处理任务
qhold	批处理实用程序	暂停批处理任务
qmove	批处理实用程序	移动批处理任务
qmsg	批处理实用程序	向批处理任务发送消息
qrerun	批处理实用程序	返回批处理任务
qrls	批处理实用程序	释放批处理任务
qselect	批处理实用程序	选择批处理任务
qsig	批处理实用程序	发信号给批处理任务
qstat	批处理实用程序	显示批处理任务状态
qsub	批处理实用程序	提交脚本
read	Shell 编程	从标准输入读取一行
renice	进程管理	设置进程的 nice 值
rm	文件系统	删除整个目录
rmdel	源代码控制系统	从 SCCS 文件中删除差异
rmdir	文件系统	删除空目录
sact	源代码控制系统	显示 SCCS 文件正在进行的编辑
sccs	源代码控制系统	源代码控制系统前端
sed	文字处理	流编辑器
sh[4]	Shell 编程	Shell, 标准命令语言解析器
sleep	Shell 编程	延时
sort	文字处理	文本排序
split	其他	分割文件
strings	C 语言编程	查找文件中可打印字符串
strip	C 语言编程	从可执行文件中移除无用信息
stty	其他	设置终端选项
tabs	其他	定义终端跳格
tail	文字处理	显示文件结尾
talk	其他	与另外用户对话
tee	Shell 编程	从标准输入读入，写到标准输出
test	Shell 编程	计算表达式
time	进程管理	计算一个命令的执行时间
touch	文件系统	改变文件访问和修改时间
tput	其他	改变终端字符
tr	文字处理	翻译字符
true	Shell 编程	返回真值
tsort	文字处理	拓扑排序
tty	其他	返回用户终端名
type	其他	显示命令类型
ulimit	其他	设置或显示文件限制
umask	其他	设置或显示文件生成掩码
unalias	其他	移除别名定义

[4]早期版本 sh 可能是 Thompson shell 或者 PWB shell。

名称	分类	描述
uname	其他	返回系统名
uncompress	其他	解压缩数据
unexpand	文字处理	转换空格为制表符
unget	源代码控制系统	回退之前从源代码控制系统获得的文件
uniq	文字处理	报告或者删除文件中重复行
unlink	文件系统	调用未链接函数
uucp	网络	系统间拷贝
uudecode	网络	解码二进制文件
uuencode	网络	编码二进制文件
uustat	网络	uucp 状态查询和作业控制
uux	进程管理	远程命令调用
val	源代码控制系统	验证 SCCS 文件
vi	文字处理	面向屏幕的可视化编辑器
wait	进程管理	等待进程结束
wc	文字处理	字、行字节或者字符计数
what	源代码控制系统	鉴别源代码控制系统文件
who	系统管理	显示登录用户
write	其他	输出到另一个用户终端
xargs	Shell 编程	从输入列表中执行命令
yacc	C 语言编程	用来生成编译器的编译器
zcat	文字处理	显示或连接 zip 压缩的文件

1.5 Daemons

This is a list of UNIX daemons that are found on various UNIX-like operating systems. UNIX daemons typically have a name ending with a d.

Table 1.2: IEEE Std 1003.1-2008 utilities

init	The UNIX program which spawns all other processes.
biod	Works in cooperation with the remote nfsd to handle client NFS requests.
crond	Time-based job scheduler, runs jobs in the background.
dhcpd	Dynamically configure TCP/IP information for clients.
fingerd	Provides a network interface for the finger protocol, as used by the finger command.
ftpd	Services FTP requests from a remote system.
httpd	Web server daemon.
inetd	Listens for network connection requests. If a request is accepted, it can launch a background daemon to handle the request, was known as the super server for this reason. Some systems use the replacement command xinetd.
lpd	The line printer daemon that manages printer spooling.
nfsd	Processes NFS operation requests from client systems. Historically each nfsd daemon handled one request at a time, so it was normal to start multiple copies.
ntpd	Network Time Protocol daemon that manages clock synchronization across the network. xntpd implements the version 3 standard of NTP.

Process	Description
portmap/rpcbind	Provides information to allow ONC RPC clients to contact ONC RPC servers
sshd	Listens for secure shell requests from clients.
sendmail	SMTP daemon.
swapper	Copies process regions to swap space in order to reclaim physical pages of memory for the kernel. Also called sched.
syslogd	System logger process that collects various system messages.
syncd	Periodically keeps the file systems synchronized with system memory.
xfsd	Serve X11 fonts to remote clients.
vhand	Releases pages of memory for use by other processes. Also known as the “page stealing daemon”
ypbind	Find the server for an NIS domain and store the information in a file.

Chapter 2

GNU

2.1 FSF

1983 年，Richard Mathew Stallman 创立了 GNU 计划，目标是为了开发一个完全自由的类 UNIX 操作系统（Free UNIX）。但是，在当时的 GNU 是仅有自己一个人单打独斗的史托曼，但又不能不做这个计划，于是史托曼反其道而行——“既然操作系统太复杂，我就先写可以在 UNIX 上面运行的程序”，于是史托曼便开始了程序的编写。

计算机仅认识 0/1 的信息，但是人类只能识别纯文本的信息（就是 ASCII 文件格式），但是计算机又不认识 ASCII 格式的文本，为此就需要用所谓的“编译器”来代替人工编译程序。

计算机软件都得要编译成二进制文件（binary file）后才能够执行，因此 Richard Mathew Stallman 便开始编写 C 语言编译器，那就是后来的 GNU C（gcc）。另外，Richard Mathew Stallman 还编写了更多可以被调用的 C 函数库（GNU C library）以及可以被用来操作操作系统的基本界面 BASH Shell，这些都在 1990 年左右完成。

这样，后来很多的软件开发者可以通过这些基础的工具来进行程序开发，进一步壮大了自由软件团体。不过对于 GNU 的最初构想“建立一个自由的 UNIX 操作系统”来说，有这些优秀的程序是仍无法满足，因为当下并没有“自由的 UNIX 核心”存在，所以这些软件仍只能在那些有专利的 UNIX 平台上工作，一直到 Linux 的出现。

在 1985 年，Richard Mathew Stallman 发起自由软件基金会（FSF）并且在 1989 年撰写了 GPL 协议。1990 年代早期，GNU 开始大量的产生或收集各种系统所必备的组件，例如库、编译器、调试工具、文本编辑器、网页服务器以及 UNIX shell，但是一些底层环境（如硬件驱动、守护进程运行内核）仍然不完整和陷于停顿。

GNU 计划中是在 Mach microkernel 的架构之上开发系统内核（也就是所谓的 GNU Hurd），但是基于 Mach 的设计异常复杂，发展进度则相对缓慢，最近发布的 GNU 系统版本是 2011 年 4 月 1 日发布的 GNU 0.401，采用 GNU Hurd 作为操作系统内核。

其他的内核，例如最著名的是 Linux kernel 等被应用到了 GNU 系统中。

2.2 GPL

Linux 操作系统是基于 GPL 许可证授权下的，GPL 许可证的目的就是防止二进制包成为唯一的软件发行源[2]，其核心观念是“版权制度是促进社会进步的手段，版权本身不是自然权力。”

“Free software” is a matter of liberty, not price. To understand the concept, you should think of “free speech”, not “free beer”. “Free software” refers to the users freedom to run, copy, distribute, study, change, and improve the software.

大意是说，Free Software（自由软件）是一种自由的权力，并非是“价格”。举例来说，你可以拥有自由呼吸的权力、你拥有自由发表言论的权力，但是这并不代表您可以到处喝“免费的啤酒（free beer）”，也就是说自由软件的重点并不是指“免费”的，而是指具有“自由度，freedom”的软件。

1. 取得软件与源代码：您可以根据自己的需求来执行这个自由软件；
2. 复制：您可以自由的复制该软件；
3. 修改：您可以将取得的源代码进进程序修改工作，使之适合您的工作；
4. 再发行：您可以将您修改过的程序，再度的自由发行，而不会与原先的编写者冲突；
5. 回馈：您应该将您修改过的程序代码回馈于社群。

但是，您所修改的任何一个自由软件都不应该也不能这样：

1. 修改授权：您不能将一个 GPL 授权的自由软件，在您修改后而将它取消 GPL 授权；
2. 单纯销售：您不能单纯的销售自由软件。

也就是说，既然 GPL 是站在互助互利的角度上去开发的，您自然不应该将大家的成果占为己有而取消 GPL 授权，因此当然不可以将一个 GPL 软件的授权取消，即使您已经对该软件进行大幅度的修改。

自由软件是可以销售的，不过不可以仅销售该软件，应同时搭配售后服务与相关手册，因此 Linux 开发商大多都是销售“售后服务”，例如咨询服务、售后服务、软件升级与其他协助工作等的附加价值。

2.3 XFree86

1988 年 — 图形界面 XFree86 计划

图形用户界面的需求日益增加，在 1984 年由 MIT 与其他厂商首次发布了 X Window System，并在 1988 年成立了非营利性质的 XFree86 组织。

XFree86 其实是 X Window System + Free + x86 的整合名称，而这个 XFree86 的 GUI 界面更是在 Linux 的核心 1.0 版于 1994 年发布时整合于 Linux 操作系统中。

具体来说，X Window System 只是 Linux 上的一套软件（而不是核心），因此即使 X Window 出现问题，对 Linux 也不会造成直接的影响。

当使用 XFree86 来管理图形界面输出时，它管理着几乎所有与显示相关的控制（例如显卡、屏幕、键盘、鼠标等）。或者，可以称 XFree86 为 X Window System 的服务器，简称为 X Server。

使用 X Server 提供的相关显示硬件的功能来完成图形化显示的窗口管理器（Window Manager，WM）是挂载在 X Server 上运行的一套显示图形用户界面的软件（包括 GNOME 和 KDE 等）。

Chapter 3

BSD

3.1 Overview

BSD（Berkeley Software Distribution）最早是 1970 年代由伯克利加州大学（University of California, Berkeley）的学生 Bill Joy 开创的一个 UNIX 的衍生系统（UNIX 变种），后来 BSD 也被用来代表其衍生出的各种套件，BSD 系统更为类似于 UNIX。

跟 AT&T UNIX 一样，BSD 也采用单内核，这意味着内核中的设备驱动作为操作系统的核心部分在核心态下运行。事实上，BSD 就是传统 UNIX 的直接衍生品，而 Linux 则是一个松散的基于 UNIX 衍生品（Minix）而新创建的一个操作系统内核及其工具。

虽然现在 BSD 和 Linux 系统上都运行着许多相同的软件，但是 BSD 项目维护的是整个操作系统，而 Linux 则只是主要集中在单一的内核上面，这点确实是需要注意的。

BSD 常被当作工作站级别的 UNIX 系统，这得归功于 BSD 使用授权非常地宽松，许多 1980 年代成立的计算机公司（例如如 DEC 的 Ultrix，以及 Sun 公司的 SunOS）都从 BSD 中获益。

1990 年代以后，BSD 很大程度上被 System V 4.x 版以及 OSF/1 系统所取代，但其开源版本被采用，促进了因特网的开发。例如，Bill Joy 创办的 Sun 公司就是以 BSD 开发的核心来构建商业 UNIX 版本——SPARC，后来可以安装在 x86 硬件架构上面 FreeBSD 也是由 BSD 改版而来。

BSD 开创了现代计算机的潮流，其中伯克利的 BSD 率先包含了库来支持互联网协议栈（TCP/IP Stack）、伯克利套接字（sockets）。通过将套接字与 UNIX 操作系统的文件描述符相整合，用户可以通过计算机网络读写数据，跟直接在磁盘上操作一样容易。

AT&T 实验室最后也发布了类似的 STREAMS 库，在软件栈中引入了相似的功能。虽然结构层有所改进，但是套接字库已经使用广泛，另外 STREAMS 库缺乏对开放套接字的轮询功能（类似于伯克利库中的 select 调用），增加了将软件移植到新的 API 的困难。

与 GNU General Public License（GPL）相比，BSD 许可协议要更加灵活。

通过一个二进制兼容层（compatibility layer），在 BSD 操作系统上可以运行相同构架下其他操作系统上的原程序，因此比模拟器要快得多。通过这个方法，针对 Linux 的应用程序也可以在 BSD 上运行，所以 BSD 不仅适合作为服务器，也可作为工作站来使用。

另外，管理员也可以将一些原本只用于商业 UNIX 变种的专属软件转移到 BSD，这样操作系统在保持原有功能的同时更趋于现代化。

3.2 History

最初的 UNIX 套件源自 1970 年代的贝尔实验室，操作系统中包含源码，这样研究人员以及大学都可以参与修改扩充。

1974 年，第一个伯克利的 UNIX 系统被安装在 PDP-11 机器上，计算机科学系而后将其用作扩展研究，后来伯克利的研究生 Bill Joy 在 1977 年将程序整理到磁带上并作为 first Berkeley Software Distribution（1BSD）发行。

1BSD 被作为第六版 UNIX 系列，而不是单独的操作系统，其主要程序包括 Pascal 编译器以及 Joy 的 ex 行编辑器。

Second Berkeley Software Distribution（2BSD）于 1978 年发布，除了对 1BSD 中的软件进行升级，还包括了 Joy 写的两个新程序：vi 文本编辑器（ex 的可视版本）以及 C Shell，它们在 UNIX 系统中至今仍被使用。

2BSD 以后的版本逐渐从 PDP-11 结构向 VAX 计算机移植，2.11BSD 于 1992 年发布，并一直更新维护持续到 2003 年。

1978 年，伯克利安装了第一台 VAX 计算机，但是将 UNIX 移植到 VAX 构架的 UNIX/32V 并没有利用 VAX 虚拟内存的能力，因此伯克利的学生重写了 32V 的大部分内核来实现虚拟内存的支持。

1979 年，3BSD 诞生了，这个新系统完整包括了一个新内核、从 2BSD 移植到 VAX 的工具，以及 32V 原来的工具。

3BSD 的成功使得 Defense Advanced Research Projects Agency（DARPA）决定资助伯克利的 Computer Systems Research Group（CSRG，计算机系统研究组）来开发一个 UNIX 标准平台以供 DARPA 未来的研究。

1980 年 10 月，CSRG 发布了 4BSD，该版本对 3BSD 有许多改进。

相较于 VAX 机器的主流系统 VMS，用户对 BSD 时有批评，1981 年 6 月终于发布了 4.1BSD。Bill Joy 大幅度提高了 4.1BSD 内核的性能，可以跟 VMS 在多个平台上媲美。为了避免与 AT&T 的 UNIX System V（UNIX 第五版）混淆，这个版本没有取名为 5BSD。

4.2BSD 在历经两年于 1983 年 8 月正式发布，并实现了多项重大改进，不过之前有三个中间版本相继推出：

- 4.1a 引入了修改版的 BBN 预试中 TCP/IP；
- 4.1b 引入了由 Marshall Kirk McKusick 实现的新型 Berkeley Fast File System（FFS）；

- 4.1c 是 4.2BSD 开发最后几个月的过渡版。

4.2BSD 这是 1982 年 Bill Joy 离开前去创建 Sun 公司后的第一个版本，此后 Mike Karels 和 Marshall Kirk McKusick 一直负责领导该项目。值得一提的是，这次 BSD 小恶魔正式出场，最初是 Marshall Kirk McKusick 的画作，出现在打印好的文档封面上，由 USENIX 发行。

1986 年 6 月，4.3BSD 发布。该版本主要是将 4.2BSD 的许多新贡献作性能上的提高，原来的 4.1BSD 没有很好地协调。在该版本之前，BSD 的 TCP/IP 实现已经跟 BBN 的官方实现有较大差异。经过数月测试后，DARPA 认为 4.2BSD 更合适，所以在 4.3BSD 中作了保留。

在 4.3BSD 开始逐渐抛开老式的 VAX 平台，1988 年 6 月移植的 4.3BSD-Tahoe 表现不俗，并且 BSD 开始将依赖于机器跟不依赖于机器的代码分离，为未来系统的可移植性打下了良好的基础。

到此为止，所有的 BSD 版本混合了专属的 AT&T UNIX 代码，这样继续使用就需要从 AT&T 获得许可证，因此在 1989 年 6 月，没有 AT&T 授权也能使用的 Networking Release 1（Net/1）诞生了，并可以遵照 BSD 许可证进行自由再发布。

1990 年初，推出了 4.3BSD-Reno，这是 4.4BSD 早期开发的过渡版，使用该版本被戏称为是一种赌博，因为 Reno 就是内华达州的赌城雷诺。

Net/1 以后，Keith Bostic 提议 BSD 系统中应该有更多的非 AT&T 部分以 Net/1 的协议发布。随后，他开始着手重新实现一些 UNIX 标准工具，其中不使用原来的 AT&T 代码，例如 Vi 被重写为 nvi（new vi）。18 个月后，所有 AT&T 的工具被替换，剩下的只是存留在内核的一些 AT&T 文件。

1991 年 6 月，Net/2 诞生了，这是一个全新的操作系统，并且可以自由发布，因此 Net/2 成为 Intel 80386 构架上两种移植的主要版本，包括由 William Jolitz 负责的自由的 386BSD 以及由 Berkeley Software Design（BSDi）负责的专属的 BSD/OS。

386BSD 本身持续时间并不长，并且在不久后成为 NetBSD 和 FreeBSD 原始代码的基础，与此同时，Linux 跟 386BSD 的开发几乎同时起步。

1992 年，拥有 System V 版权以及 UNIX 商标的 AT&T 的 UNIX Systems Laboratories（USL）附属公司发起的针对 BSDi 的诉讼终止了 Net/2 的发布，直到其源码能够被鉴定为符合 USL 的版权，并且最终导致 BSD 后裔的开发（特别是自由软件）延迟了两年，反而使没有法律问题的 Linux 内核获得了极大的支持。

伯克利套件的 18,000 个文件中，只有 3 个文件要求删除，另有 70 个文件要求修改，并显示 USL 的版权说明，并且 USL 将不得对后续的 4.4BSD 提起诉讼，不管是用户还是伯克利代码的分发者。

1994 年 6 月，4.4BSD 以两种形式发布：可自由再发布的 4.4BSD-Lite（不包含 AT&T 源码），以及遵照 AT&T 的许可证的 4.4BSD-Encumbered，而且伯克利的最终版本是 1995 年的 4.4BSD-Lite Release 2，而后 CSRG 解散。

在伯克利的 BSD 开发告一段落后，基于 4.4BSD 的其他套件（比如 FreeBSD、OpenBSD 和 NetBSD）继续被维护。

另外，BSD 许可证的宽容促使许多其他的操作系统都采用了 BSD 的代码。相比 BSD 许可证，GPL 许可证让您有权拥有任何你想要使用该软件的方法，但你必须确保提供源代码给下一个使用它的人（包括你对它的改变部分），而 BSD 许可证并不是要求你必须那么做，因此 Microsoft Windows 在 TCP/IP 的实现上引入了 BSD 代码，而且在 Windows 操作系统中还采用了许多 BSD 命令行下的网络工具。

BSD 许可证的限制则要少得多，可以允许二进制包成为唯一的发行源，这就是 GPL 与 BSD 的核心差异。当前的 BSD 操作系统变种支持各种通用标准，包括 IEEE、ANSI、ISO 以及 POSIX，同时保持了传统 BSD 的良好风范。

BSD 是一个包括众多工具的基本系统，包括 libc 在内都是基本系统的一部分，这些组件都被作为一个基本系统被一起开发和打包。

在 BSD 操作系统中，可以使用 `freebsd-update fetch update` 命令，或者下载整个源代码树来编译升级，而在 Linux 中可以通过内置的包管理系统来升级系统，因此前者（BSD）仅更新基本系统，而后者（Linux）则会升级整个系统。不过，BSD 升级到最新的基本系统并不意味着所有的附加软件包也将会被更新，而 Linux 升级的时候所有的软件包都会被升级。

除了最基本的系统软件，在 386BSD 基础上开发的 FreeBSD 还提供了一个拥有成千上万广受欢迎的程序组成的软件的 Ports Collection（包括服务器软件、游戏、程序设计语言、编辑器等），完整的 Ports Collection 大约需要 500 MB 的存储空间，所有的只提供对原始代码的“修正”，从而使用户能够容易地更新软件，并且减少了老旧的 1.0 Ports Collection 对硬盘空间的浪费。

要编译一个 port，只要切换到想要安装的程序的目录，输入 `make install`，然后让系统去做剩下的事情。

每一个程序完整的原始代码可以从 CD-ROM 或本地 FTP 获得，所以只需要编译想要的软件具备足够的磁盘空间，而且大多数的软件都提供了事先编译好的“package”以方便安装，不希望从源代码编译安装 ports 的用户只要使用一个简单的命令 (`pkg_add`) 就可以安装。

Chapter 4

Solaris

Solaris 原先是 Sun Microsystem 公司研制的类 UNIX 操作系统，在 Sun 公司被 Oracle 并购后称作 Oracle Solaris。

Sun 公司的创始人之一——Bill Joy 来自柏克莱加州大学（U.C.Berkeley），因此早期的 Solaris 是由 BSDUNIX 发展而来。

随着时间的推移，Solaris 现在在接口上正在逐渐向 System V 靠拢，并且 Sun 公司在 2005 年 6 月 14 日将正在开发中的 Solaris 11 的源代码以 CDDL 许可开放成为 OpenSolaris。

2010 年 8 月 23 日，OpenSolaris 项目被 Oracle 中止，并于 2011 年 11 月 9 日发布 Solaris 11。

Sun 的操作系统最初叫做 SunOS[1]，SUN 的操作系统开发从 SunOS 5.0 开始转向 System V4，并且有了新的名字叫做 Solaris 2.0。

Solaris 2.6 以后，SUN 删除了版本号中的“2”，因此 SunOS 5.10 就叫做 Solaris 10，而且 Solaris 的早期版本后来又被重命名为 Solaris 1.x，因此“SunOS”被用做专指 Solaris 操作系统的内核，因此 Solaris 被认为是由 SunOS 和图形化的桌面计算环境以及网络增强部分组成。

早期的 Solaris 主要用于 Sun 工作站上，后来 Solaris 可以运行在 Intel x86 及 SPARC/UltraSPARC 平台上，后者是太阳工作站使用的处理器，而且 Solaris 在 SPARC 上拥有强大的处理能力和硬件支持。

对于这两个平台，Solaris 屏蔽了底层平台差异来为用户提供了尽可能一致的使用体验，并且最新发布的 Solaris10 包含若干创新技术（包括 ZFS、DTrace、Solaris Zones（Container）、预测性自愈等），其中一些以往只可能在专业服务器等具有相关硬件的大型机器上才可能得到支持，但是 Solaris10 使得任何一台普通 PC 都可以具有这些能力。

[1]现在，SunOS 仍旧用来称呼 Solaris 的核心，其中 SunOS 的版本号是以 5.Solaris 版本号来表示，例如 Solaris 10 在 SunOS 5.10 上运行。
Solaris 的 man 手册是以 SunOS 为标记的，启动的时候也显示它，但是 “SunOS” 这个词不再用于 Sun 的市场文档中。

Solaris 支持 SPARC、x86 及 x64（即 AMD64 及 EM64T 处理器）等系统架构。与 Linux 相比，Solaris 可以更有效地支持对称多处理器（即 SMP 架构）。同时，Sun 宣布将在 Solaris 10 的后续版本中提供 Linux 运行环境，允许 Linux 二进制程序直接在 Solaris x86 和 x64 系统上运行，目前这一技术已通过 Solaris Zone 的一个特殊实现（BrandZ）得到支持。

Solaris 传统上与基于 Sun SPARC 处理器的硬件体系结构结合紧密，在设计上和市场上经常捆绑在一起，整个软硬件系统的可靠性和性能也因此大大增强。不过，Solaris 10 已经能很好地支持 x64(AMD64/EM64T) 架构，而且 Sun 公司已推出自行设计的基于 AMD64 的工作站和服务器并支持 Solaris10。

Solaris 的第一个桌面环境是 OpenWindows，紧接着是 Solaris 2.5 的 CDE。

- Solaris 10 中推出了基于 GNOME 的 Java Desktop System，另外也支持 KDE、XFCE、WindowMaker 等。
- Solaris 11 采用 GNOME。

Solaris 的大多数源代码已经在 CDDL 的许可下在 OpenSolaris 开源项目中发布。不过，虽然 Solaris 已开放其部分源代码，但是 Solaris“不是”自由软件，而 OpenSolaris 才是，因此 Solaris 和 OpenSolaris 是两个“不同”的操作系统。

Solaris 的二进制和源代码目前都可以被下载和许可而无需任何费用，Sun 的 Common Development and Distribution License（CDDL）被选择用做 OpenSolaris 的许可，并通过了 Open Source Initiative 评审和批准，但其授权条款与时下流行的 GPL 互不兼容。

OpenSolaris 于 2005 年 6 月 14 日正式启动，源代码来自当时的 Solaris 开发版本，不过已被收购 Sun 公司的 Oracle 中止，由社区发起的 Illumos 计划继承。

Chapter 5

UNIX-like

其实，UNIX-like 可以说是目前服务器类型的操作系统的统称，FreeBSD、BSD、Sun UNIX、HP UNIX、Red Hat Linux、Mandrake Linux 等都是由同一个祖先“UNIX”产生的，它们都被统称为 UNIX-like 的操作系统。

5.1 MINIX

MINIX 原来是荷兰阿姆斯特丹自由大学计算机科学系的塔能鲍姆教授（Andrew S. Tanenbaum）所开发的一个轻量的微内核小型类 UNIX 操作系统。

MINIX 取自 Mini UNIX 的缩写，并且其第三版在 2000 年 4 月重新以 BSD 许可证发布为自由软件，而且被“严重的”重新设计。

与 Xinu、Idris、Coherent 和 Uniflex 等类 UNIX 操作系统类似，MINIX 派生自 Version 7 UNIX，但并没有使用任何 AT&T 的代码。

MINIX 的第一版于 1987 年发布并提供完整的源代码给大学与学生作为授课及学习用途，现在已经是开放源代码软件。

1979 年的版权声明中说明在 UNIX version7 推出之后将 UNIX 源代码私有化，不过 UNIX ersion 7 可以在 Intel 的 x86 架构上进行移植，因此 Andrew S. Tanenbaum 将 UNIX 改写为 Minix 并移植到 x86 上，并且 Minix 与 UNIX 完全兼容。

1986 年发布 Minix，最初可以在 1980 年代到 1990 年代的 IBM PC 和 IBM PC/AT 兼容计算机上运行，次年出版 Minix 相关书籍同时与新闻组相结合。

Minix 以 C 语言开发，并且与 Version 7 UNIX 兼容，支持 16-bits 的 Intel 8080 平台，其全部的代码共约 12,000 行，并作为 Operating Systems: Design and Implementation 的示例。

Minix 1.5 版也有移植到已 Motorola 68000 系列 CPU 为基础的计算机上（如 Atari ST、Amiga 和早期的 Apple Macintosh）和以 SPARC 为基础的机器（如 Sun 公司的工作站）。Minix 2 于 1997 年发布并可以在 Intel 80386 等 x86 平台上运行。

2004 年，Andrew S. Tanenbaum 重新架构与设计了整个系统，更进一步的将程序模块化，最终推出 Minix 3。

Minix 3 除了启动的部份以汇编语言编写以外，其他大部份（内核、内存管理及文件系统等）都是纯粹用 C 语言编写。

在设计之初，为了使 Minix 简化而将程序模块化，例如文件系统与存储器管理都不是在操作系统内核中运行，而是在用户空间运行，Minix 3 的 I/O 设备等也都被移到用户空间运行。

5.2 FreeBSD

FreeBSD 是一种由经过 BSD、386BSD 和 4.4BSD 发展而来的自由的类 UNIX 操作系统的重要分支[1]，而且 FreeBSD 被认为是自由操作系统中的不知名的巨人。

FreeBSD 不是 UNIX，但是可以和 UNIX 一样运行，并且兼容 POSIX。

FreeBSD 是以一个完善的操作系统的定位来做开发，其核心、驱动程序以及所有的用户层（Userland）应用程序（比方说是 Shell）均由同一源代码版本控制系统保存。相较于 Linux，其核心为一组开发人员设计，而用户应用程序则交由他人开发（例如 GNU 计划），最后再由其他团体集成并包装成 Linux 包。

FreeBSD 发展采用 Core Team 的方式，Core Team 的成员决定整个 FreeBSD 计划的大方向，对于开发者间的问题有最后的决定权，其他的开发者也可以提交建议或是他们修改过的代码。但是，Core Team 保留最终的决定权，决定是否将这功能放进 FreeBSD，因此 FCore Team 的方式与 Linux 发展大相径庭。

- Contributor（也可以说是 Submitter）无 FreeBSD 的 CVS 的访问权限，但是可以通过其它的方式，例如提交 Problem Reports 或是在 Mailing list 上面参与讨论来对 FreeBSD 做出贡献。
- Committer 有对 FreeBSD 的 CVS 及 Subversion 访问的权限，可以将其代码或是文件送到版本库里面。一个 committer 必须要在过去的 12 个月中有 commit 的动作，而一个活跃的 committer 指在每个月至少都有一次以上的 commit 动作。
  虽然说没有必要限制一个有 commit 权限的人只能在代码树中可以访问的地方，但是如果一个 committer 要在他/她没有做出贡献或是不熟悉的地方，他/她必须要读那个地方的历史记录以及 MAINTAINER 文件以确认这个部份的维护者对于更改这边的代码有没有什么特殊的要求。
- 作为整个 FreeBSD 计划的领导人的 Core Team 成员由 committer 互相推选出来，他们提升活跃的 contributor 成为 committer，还有可以指派“Hat”（指在计划中负责一些特定工作或领域的人），也是对于决定整个计划的大方向的最后仲裁者。

[1] FreeBSD 是一个支持 Intel (x86 和 Itanium®)，AMD64，Sun UltraSPARC® 计算机的基于 4.4BSD-Lite 的操作系统。

源于 386BSD 的 FreeBSD 的发展始于 1993 年，不过 FreeBSD 在 1995 年 1 月发布的 2.0-RELEASE 中以柏克莱加州大学的 4.4BSD-Lite Release 全面改写。

FreeBSD 2.0 最值得注意的部份也许是对卡内基美隆大学的 Mach Virtual Memory 系统翻修以及 FreeBSD Ports system 的发明。前者对于高负荷的系统优化，后者则是创建了一套简单且强大的机制维护第三方软件。

FreeBSD 3.0 则引入了 ELF binary 格式，并开始支持多处理器系统（SMP）以及 64 位 Alpha 平台，后续的 3.x 对于系统做了非常多的改革，这些措施在当时并没有带来好处，但却是 4.X 成功的基石。

4.0-RELEASE 于 2000 年 3 月发布，最后一个版本 4.11-RELEASE 于 2005 年 1 月发布，并支持到 2007 年 1 月。

FreeBSD 4 也是 FreeBSD 最长寿的主版本，在 FreeBSD 4 所发展出来的 kqueue 也被移植到各种不同 BSD 平台。

FreeBSD 5 的最后一个版本 FreeBSD 5.5 于 2006 年五月发布。在 FreeBSD 4 的 SMP 架构下，在同一时间内只允许一个 CPU 进入核心（即 Giant Lock），FreeBSD 5 最大的改变在于改善底层核心 Locking 机制，审视并改写核心代码，使得不同的 CPU 可以同时进入系统核心来增加效率。

自 FreeBSD 5.3 开始支持 m:n 线程的 KSE（Kernel Scheduled Entities），表示 m 个用户线程共用 n 个核心线程的模式，其中的许多贡献是由商业化版本的 BSD OS 团队的支持。

FreeBSD 6.3 在 2008 年 1 月 18 日发布，主要是针对软件的更新，并加入 lagg（可以对多张网卡操作）的支持，以及引入重新改写的 unionfs，其最终版本 FreeBSD 6.4 于 2008 年 11 月 28 日发行。

FreeBSD 7 于 2007 年 6 月 19 日进入发布程序，2008 年 2 月 27 日 7.0-RELEASE 正式发布，后来在 2010 年 03 月 23 日正式发布 FreeBSD 7.3-RELEASE 版本，其中新增的功能包括了：

- SCTP
- 日志式 UFS 文件系统：gjournal
- 移植 DTrace
- 移植 ZFS 文件系统
- 使用 GCC4
- 对 ARM 与 MIPS 平台的支持
- 重写过的 USB stack
- Scalable concurrent malloc 实现
- ULE 调度表 2.0（SCHED_ULE），并修改加强为 SCHED_SMP，在交付至 CVS 时的正式名称为 ULE 3.0[2]。
- Linux 2.6 模拟层

[2]ULE 3.0 在 8 核心的机器上以 sysbench MySQL 测试的结果，速度上比 Linux 2.6 快大约 10%（无论是使用 Google 的 tcmalloc 或是 glibc+cfs）

- Camellia Block Cipher
- ZFS 的运行

FreeBSD 8 于 2009 年 11 月 27 日发布，2010 年 07 月 24 日发布 FreeBSD 8.1-RELEASE，并增加了如下新特性：

- 虚拟化方面：Xen DOM-U、VirtualBox guest 及 host 支持、层次式 jail。
- NFS：对 NFSv3 GSSAPI 的支持，以及试验性的 NFSv4 客户端和服务器。
- 802.11s D3.03 wireless mesh 网络，以及虚拟 Access Point 支持。
- ZFS 不再是试验性的了。
- 基于 Juniper Networks 提供 MIPS 处理器的实验性支持。
- SMP 扩展性的增强，显著改善在 16 核心处理器系统中的性能。
- VFS 加锁的重新实现，显著改善文件系统的可扩展性。
- 显著缓解缓冲区溢出和内核空指针问题。
- 可扩展的内核安全框架（MAC Framework）现已正式可用。
- 完全更新的 USB 堆栈改善了性能和设备兼容性，增加了 USB target 模式。

FreeBSD 9.0 于 2012 年 1 月发布，该版本是第一个 9.x 的 FreeBSD 稳定分支，具有以下特性：

- 采用了新的安装程序 bsdinstall
- ZFS 和 NFS 文件系统得到改进
- 升级了 ATA/SATA 驱动并支持 AHCI
- 采用 LLVM/Clang 代替 GCC
- 高效的 SSH（HPN-SSH）
- PowerPC 版支持索尼的 PS3

目前，FreeBSD 具有的一些非凡的特性可以列表如下。

- 抢占式多任务与动态优先级调整确保在应用程序和用户之间平滑公正的分享计算机资源，即使工作在最大的负载之下。
- 多用户设备使得许多用户能够同时使用同一 FreeBSD 系统做各种事情。比如，打印机和磁带驱动器等系统外设可以完全地在系统或者网络上的所有用户之间共享，可以对用户或者用户组进行个别的资源限制，以保护临界系统资源不被滥用。
- 符合业界标准的强大 TCP/IP 网络支持，例如 SCTP、DHCP、NFS、NIS、PPP，SLIP，IPsec 以及 IPv6。这意味着 FreeBSD 主机可以很容易地和其他系统互联，也可以作为企业的服务器，比如 NFS(远程文件访问) 以及 email 服务，或接入 Internet 并提供 WWW，FTP，路由和防火墙 (安全) 服务。
- 内存保护确保应用程序 (或者用户) 不会相互干扰。一个应用程序崩溃不会以任何方式影响其他程序。
- FreeBSD 是一个 32 位操作系统 (在 Itanium®，AMD64，和 UltraSPARC® 上是 64 位)，并且从开始就是如此设计的。

- 业界标准的 X Window 系统[3] (X11R7) 为便宜的常见 VGA 显示卡和监视器提供了一个图形化的用户界面 (GUI)，并且完全开放代码。
- 和许多 Linux、SCO、SVR4、BSDI 和 NetBSD 程序的二进制代码兼容性
- 数以千计的 ready-to-run 应用程序可以从 FreeBSD ports 和 packages 套件中找到。
- 可以在 Internet 上找到成千上万其它 easy-to-port 的应用程序。FreeBSD 和大多数流行的商业 UNIX® 代码级兼容，因此大多数应用程序不需要或者只要很少的改动就可以编译。
- 页式请求虚拟内存和“集成的 VM/buffer 缓存”设计有效地满足了应用程序巨大的内存需求并依然保持其他用户的交互式响应。
- SMP 提供对多处理器的支持。
- 内建了完整的 C、C++、Fortran 开发工具。许多附加的用于高级研究和开发的程序语言，也可以在通过 ports 和 packages 套件获得。
- 完整的系统源代码意味着用户对生产环境的最大程度的控制。

FreeBSD 在 BSD 许可证下发布，允许任何人在保留版权和许可协议信息的前提下随意使用和发行。BSD 许可协议并不限制将 FreeBSD 的源代码在另一个协议下发行，因此任何团体都可以自由地将 FreeBSD 代码融入它们的产品之中去。

FreeBSD 包含了 GNU 通用公共许可证、GNU 宽通用公共许可证、ISC、CDDL 和 Beerware 许可证的代码，也有使用三条款和四条款的 BSD 许可证的代码。另外有些驱动程序也包涵了 binary blob，像是 Atheros 公司的硬件抽象层。这使得所有人都可以自由地使用还有再散布 FreeBSD。

不过，FreeBSD 的核心和新开发的代码大多都使用两条款的 BSD 许可证发布，许多使用 GPL 的代码都必须经过净室工程，以其他授权方式重写，这主要是避免整个核心受到 GPL 影响。

在过去的几年中 FreeBSD 的中央源代码树是由 CVS (并行版本控制系统) 来维护的，自 2008 年六月起开始转为使用 SVN。

现在 FreeBSD 的主源码库使用 SVN，客户端的工具（例如 CVSup 和 csup 这些依赖于旧的 CVS 基础结构）依然可以使用，对于 SVN 源码库的修改会被导回进 CVS，只有中央源代码树是由 SVN 控制的。

FreeBSD 的文档、万维网和 Ports 库还仍旧使用着 CVS，主 CVS 代码库放置在美国加利福尼亚州圣克拉拉的机器上，并被复制到全世界的大量镜像站上，其中包含`--CURRENT`和`--STABLE`的 SVN 树也同样能非常容易的复制到用户的机器上。

committer 是那些对 CVS 树有写权限的人，他们被授权修改 FreeBSD 的源代码（术语“committer”来自于 cvs 的 commit 命令），提交修正的最好方法是使用 `send-pr` 命令。

[3] FreeBSD 是廉价 X 终端的一种绝佳解决方案，可以选择使用免费的 X11 服务器。与 X 终端不同，如果需要的话 FreeBSD 能够在本地直接运行程序，因而减少了中央服务器的负担。

5.3 OpenBSD

OpenBSD 是一个在 1995 年由项目发起人西奥 · 德 · 若特从 NetBSD 分支而出的类 UNIX 操作系统。

OpenBSD 以对开放源代码的坚持、高质量的文件、坚定的软件授权条款和专注于系统安全及代码质量而闻名，并且 OpenBSD 以河豚作为项目吉祥物。

OpenBSD 包含了一些在其他操作系统缺少或是列为选择性的安全特色，而且 OpenBSD 非常重视代码审阅，至今开发者仍然保有这样的传统，此外 OpenBSD 对软件授权条款相当坚持，所有对核心的修改都必须符合 BSD 许可证的条款。

当创立 OpenBSD 的时候，西奥 · 德 · 若特就决定任何人都可以在任何时间取得源代码，在 Chuck Cranor 的协助下他创建了一个公开且匿名的 CVS 服务器，因此 OpenBSD 也是第一个以开放式 CVS 作为开发方式的软件。在当时，CVS 的应用上大多只让少数的开发者有访问权，外部的开发者没有办法知道目前的工作进度，贡献的修正档也常常是已经完成过的修正，正是这种开发方式让 OpenBSD 成为开放源代码的代表软件。

1994 年 12 月，NetBSD 的共同发起人西奥 · 德 · 若特被要求辞去 NetBSD 的开发工作，而他访问 NetBSD 代码的权利也被取消。

1995 年 10 月，西奥 · 德 · 若特从 NetBSD 1.0 派生出了 OpenBSD 计划，在 1996 年 7 月发布了最初的发布版 OpenBSD 1.2，同年 10 月发布了 OpenBSD 2.0。之后每隔 6 个月 OpenBSD 便会发布一个新版本，每个发布版本会维护 1 年。

2007 年 7 月 25 日，OpenBSD 决定成立一个非营利性质的 OpenBSD 基金会以提供 OpenBSD 用户或是组织对 OpenBSD 法律上的支持服务，组织的地点设在加拿大。

5.4 NetBSD

NetBSD 是一个具有高度可定制性的安全的免费的类 UNIX 操作系统，用户可以通过完整的源代码获得支持，而且许多程序都可以很容易地通过 NetBSD Packages Collection 获得。

NetBSD 和 FreeBSD 都是从加州柏克莱大学的 4.3BSD-Net/2 及 386BSD 为基础发展而来，NetBSD 的四位发起人（Chris Demetriou、西奥 · 德 · 若特、Adam Glass 以及 Charles Hannum）的目的在于发展一种跨平台、高质量、以柏克莱软件包为基础的操作系统。

NetBSD 原始代码的版本库创建于 1993 年 3 月 21 日，并于 1993 年四月发布了第一个版本——NetBSD 0.8。同年 9 月，NetBSD 发布 0.9 版，包含了许多修正与功能的加强。

1994 年 10 月，NetBSD 发布 1.0 版，这个版本是 NetBSD 一个提供多平台的版本，目前 NetBSD 的最新版本为 2013 年 5 月 18 日所发布的 6.1 版。

NetBSD 已移植到了大量的 32 -和 64 位体系结构平台上，而且 NetBSD 的发行版比任何单一的 GNU / Linux 发行版支持更多的平台，这些平台的内核和用户空间都是由中央统一管理的 CVS 源代码树。

NetBSD 内核在任何给定的目标架构需要 MMU 的存在。

5.5 Darwin

Darwin 是由 Apple 公司于 2000 年所发布的一个开放源代码操作系统。

Darwin 是 Mac OS X 和 iOS 操作环境的操作系统部份，Apple 公司于 2000 年把 Darwin 发布给开放源代码社区。

Darwin 是一种包含开放源代码的 XNU 核心的类 UNIX 操作系统，而且 Darwin 使用一种以微核心为基础的核心架构来实现 Mach kernel。

Darwin 操作系统的服务和 userland 工具是以 4.4BSD（特别是 FreeBSD 和 NetBSD）为基础，而且类似其他 UNIX-like 操作系统，Darwin 也支持对称多处理器（SMP），以及高性能的网络设施和多种集成的文件系统。

Darwin 操作系统集成 Mach 到 XNU 核心的好处在不同形式的系统使用软件的能力。举例来说，一个集成了 Mach 微核心的操作系统核心能够提供多种不同 CPU 架构的二进制格式到一个单一的文件（例如 x86 和 PowerPC），这是因为它使用了 Mach-O 的二进制格式。

不过，Mach 也增加了操作系统核心 - 核心 - 的复杂度，因此这种复杂度在过去的微核心实现上会导致很难分离核心性能的问题，不过潜在的好处是广泛的可携性。以 Darwin 可携性的具体例子来说，在 2005 年 6 月苹果计算机宣布它会于 2006 年在 Mac 计算机上开始采用 Intel 处理器。

在 2002 年 4 月，Apple 公司在 ISC（Internet Software Consortium）上成立 OpenDarwin.org 来协助合作 Darwin 发展，OpenDarwin 创建它自己发布的 Darwin 操作系统。值得注意的是，OpenDarwin 子计划中包含了 DarwinPorts，目标是组合下一世代的 port 集合给 Darwin 使用 (对于长期而言，也给其他的 BSD 所派生的操作系统)。

2003 年 7 月，苹果在 APSL 的 2.0 版本下发布了 Darwin，是由自由软件基金会（FSF）批准为自由软件的许可证。

OS X 是苹果公司为麦金塔计算机开发的专属操作系统的最新版本，Mac OS X 于 1998 年首次推出，并从 2002 年起随麦金塔计算机发售。

作为以 UNIX 基础的操作系统，Mac OS X 包含两个主要的部份，分别是以 FreeBSD 源代码和 Mach 微核心为基础的 Darwin 核心，以及由 Apple 公司开发的图形用户界面 Aqua。

OS X Server 亦同时于 2001 年发售，架构上来说与工作站（客户端）版本相同，只有在包含的工作组管理和管理软件工具上有所差异，提供对于关键网络服务的简化访问（例如邮件传输服务器、Samba 软件、LDAP 目录服务器以及 DNS 服务器。

简单来说，OS X 是 Mac OS“版本 10”的分支，然而它与早期发布的 Mac OS 相比，在 Mac OS 的历史上是倾向独立发展的，并以 Mach 内核为基础，加入 UNIX 的 BSD 实现，再集成到 NeXTSTEP[4]中。

[4]NeXTSTEP 为当时史蒂夫 · 乔布斯（Steve Jobs）于 1985 年被迫离开苹果后，到 NeXT 公司所发展的。

NeXT 的操作系统（在当时称作 OPENSTEP）被选为苹果下个操作系统的基础形式，后来转换为 Apple 计算机主要销售的家用市场，以及受到专业人士欢迎的 Rhapsody 系统上，Mac OS 9 转换到新系统之后也将 Rhapsody 演化为 Mac OS X。

现在，Mac OS X 与先前麦金塔操作系统彻底地分离开来，其底层代码完全地与先前版本不同，这个新的核心 Darwin 是一个开放源代码、符合 POSIX 标准的操作系统，伴随着标准的 UNIX 命令行与其强大的应用工具。

Mac OS X 包含了独有的软件开发程序，其重大的特色是名为 Xcode 的集成开发环境，可以编译出 Mac OS X 所运行的硬件平台上的可执行文件，也可以指定编译成 PowerPC 平台专用和 x86 平台专用，或是跨越两种平台的通用二进制。

Mac OS X 通过提供一种称为 Classic 的模拟环境，保留了与较旧的 Mac OS 应用程序的兼容性，允许用户在 Mac OS X 中把 Mac OS 9 当作一个程序进程来运行，使大部分旧的应用程序就像在旧的操作系统下运行一样。

另外，用户可以给 Mac OS 9 和 Mac OS X 的 Carbon API 创造出允许在两种系统运行的代码，OPEBSTEO（现在称为 Cocoa 技术）的 API 也依然可以使用，不过大部分的类型名称都是以 NeXTSTEP 的缩写”NS” 开头。

Mac OS X 可以运行很多 BSD 或 Linux 软件包，而且编译过的代码通常是以 Mac OS X 封装的方式来发布，但有些可能需要命令行的组态设置或是编译。

从 Mac OS X 10.3 版开始，Mac OS X 已经包含以 XFree86 4.3 和 X11R6.6 为基础实现的 Apple X11，并搭配一个模仿 Mac OS X 外观的窗口管理器，因此早期的 Mac OS X 版本可使用 XDarwin 来运行 X11 应用程序。

Chapter 6

Linux

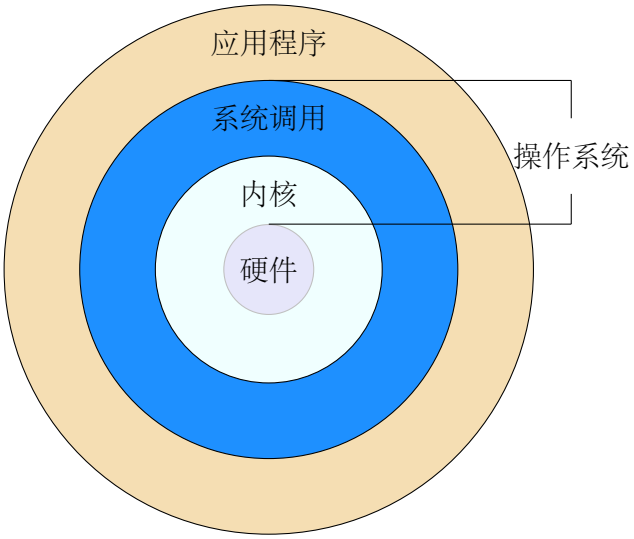
6.1 Overview

Linux 是一种自由和开放源代码的类 UNIX 操作系统，其内核原型由 Linus Torvalds 在 1991 年 10 月 5 日首次发布。

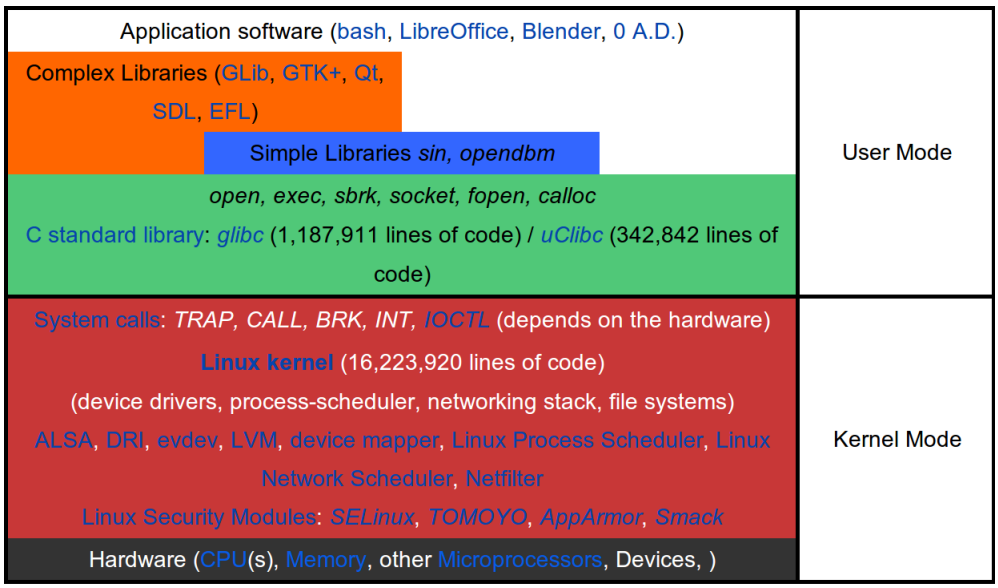
通常情况下，Linux 被打包成供个人计算机和服务器使用的 Linux 发行版（例如 Debian 及其派生版本 Ubuntu、Fedora 及其相关版本 Red Hat Enterprise Linux 和 openSUSE 等）。

计算机由硬件和运行在计算机中的软件组成，操作系统可以使用户更有效地使用硬件资源，因此操作系统属于软件范畴。

现代的操作系统除了可以有效地控制计算机硬件资源的分配，还可以提供计算需要的其他功能（例如网络功能）以及供开发人员调用的系统接口等。



作为自由软件，任何人都可以创建一个符合自己需求的 Linux 发行版。严格来讲，术语“Linux”只表示操作系统内核本身，通常称为 Linux 内核，现在常用 Linux 来指代基于 Linux 内核的完整操作系统（包括 GUI 组件和其他实用工具）。



Various layers within Linux, also showing separation between the userland and kernel space.

只要遵循 GNU 通用公共许可证，任何个人和团体都可以自由地使用 Linux 的所有底层源代码，也可以自由地修改和再发布，而且支持用户空间的系统工具和库主要由 GNU 计划提供，因此自由软件基金会提议将该组合系统命名为“GNU/Linux”。

最初，Linux 是作为支持英特尔 x86 架构的个人计算机的一个自由操作系统，后来被移植到更多的硬件平台，远远超出其他任何操作系统，因此 Linux 可以运行在服务器和其他大型平台（如大型主机和超级计算机）之上，90% 以上的超级计算机运行 Linux 发行版或变种。

Linux 系统使用单内核，并由 Linux 内核负责处理进程控制、网络，以及外围设备和文件系统的访问，而且设备驱动程序可以与内核直接集成，或者以加载模块形式添加。现在，Linux 也广泛应用在嵌入式系统（如手机、平板电脑、路由器、电视和电子游戏机等）上，而且移动设备上广泛使用的 Android 操作系统就是创建在 Linux 内核上。

Linux 提供了一个沿袭了 UNIX 的完整的操作系统中最底层的硬件控制和资源管理的完整架构，因此 Linux 发行版包含 Linux 内核和支撑内核的实用程序和库，以及可以满足各类需求的应用程序。例如，个人计算机使用的 Linux 发行版通常包含 X Window 和相应的桌面环境（如 GNOME 或 KDE）。

现在，大多数 Linux 系统提供了 X Window 等 GUI 程序，大多数人都可以直接使用 Linux 发布版，而不是自己选择每一样组件或自行设置。

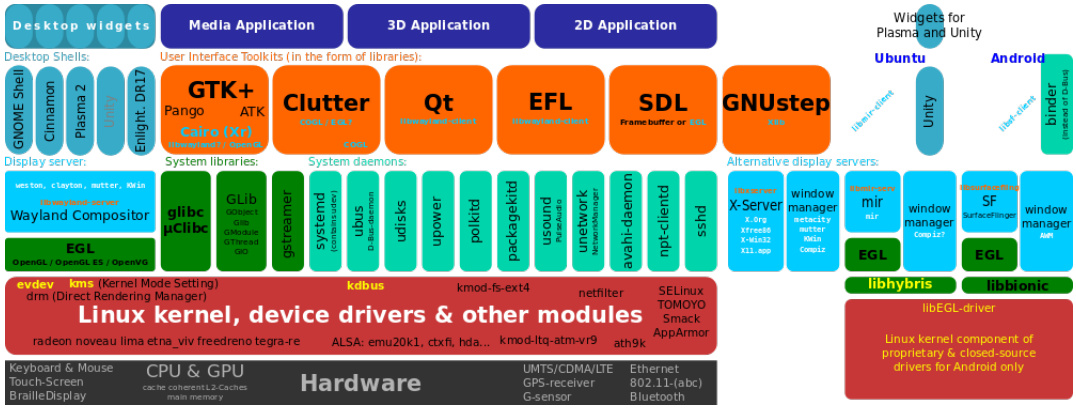


Figure 6.1: Visible software components of the Linux desktop stack

6.2 Linux 0.02

最初，Linus Torvalds 在 MINIX 上开发 Linux 内核，因此为 MINIX 写的软件也可以在 Linux 内核上使用，直到后来 Linux 成熟之后就可以在自己上面开发自己了。

使用 GNU 软件代替 MINIX 的软件，这对新操作系统是有益的，而且使用 GNU GPL 协议的源代码可以被其他项目所使用，只要这些项目使用同样的协议发布。

单个 CPU 在一个时间内理论上仅能进行一项工作，如果在计算机中同时开启两个以上的办公软件（例如电子制表与文字处理软件），那么同时开启的动作意味着多个工作同时要交给 CPU 来处理。

在某一个确定的时间点，CPU 仅能处理一个工作，但是支持多任务的 CPU 就会自动在不同的工作间切换，也就是先运行 10% 的电子制表，再转到文字处理运行 10%，再回去电子制表……，一直到将两个工作结束为止（不一定同时结束，如果某个工作先结束了，CPU 就会全速去运行剩下的那个工作了）。

以 1GHz 的 CPU 为例，该 CPU 每一秒可以进行 10^9 次工作。假设 CPU 对每个程序都只进行 1000 次运行周期，然后就得切换到下个程序，那么 CPU 在 1 秒钟内就能够切换 10^6 次，这样运行的程序在高速的 CPU 下会使用户感觉几乎是同步在运行的。

为什么有的时候同时打开两个文件（假设为 A，B 文件）所花的时间，要比打开完 A 再去打开 B 文件的时间还要多？

因为如果同时开启的话，CPU 就必须要在两个工作之间不停的切换，而切换的动作还是会耗去一些 CPU 时间的，所以同时运行两个以上的工作在一个 CPU 上，要比一个一个的执行还要耗时一点。这也是为何现在 CPU 开发商要整合两个 CPU 于一个芯片中的原因，也是为何在运行情况比较复杂的服务器上，需要比较多的 CPU 的原因。

事实上，Linus Torvalds 对于个人计算机的 CPU 其实并不满意，因为他之前接触的计算机都是工作站型的计算机，这类计算机的 CPU 特色就是可以进行“多任务处理”的能力。

早期 Intel 的 x86 架构计算机不是很受重视的原因，就是因为早期 x86 的芯片不支持多任务处理，CPU 在不同的进程之间切换不是很顺畅。

为了彻底发挥 386 的效率，于是 Linus Torvalds 花了不少时间在测试 386 机器上，而且他写了三个小程序，一个程序会持续输出 A，另一个会持续输出 B，最后一个会将两个程序进行切换，当他将三个程序同时执行，结果他看到显示器上很顺利的一直出现 ABABABAB$\cdots$，他知道，他成功了。

探索完了 386 的硬件相关信息，并且也安装了类似 UNIX 的 Minix 操作系统，同时还取得 Minix 的源代码，接下来 Linus Torvalds 干嘛去了？因为 Minix 的开发控制在谭宁邦教授手上，他希望 Minix 能以教育的立场去开发，所以对于 Minix 的开发并不是十分的热衷，但是用户对于 Minix 的功能需求又很强烈，例如一些接口与周边的驱动程序与新的协议等。在无法快速的得到解决后，Linus Torvalds 就想，那我自己写一个更适合自己用的 Minix，于是他就开始进行核心程序的编写了。

对于 Linus Torvalds 来说 GNU 真的是一个不可多得的好帮手，因为他用来编写属于自己小核心的工具就是 GNU 的 bash 工作环境与 gcc 编译器等自由软件。他以 GNU 的软件针对 386 并参考 Minix 的设计理念（注意，仅是程序设计理念，并没有使用 Minix 的源代码）来写这个小核心，而这个小核心竟然可以在 386 上面顺利的运行起来，还可以读取 Minix 的文件系统。

不过还不够，他希望这个小核心可以获得大家的一些修改建议，于是他将这个核心放置在网络上提供大家下载，同时在 BBS 上面贴了一则消息：

```
Hello everybody out there using minix-
I'm doing a (free) operation system (just a hobby,
won't be big and professional like gnu) for 386(486) AT clones.
I've currently ported bash (1.08) and gcc (1.40),
and things seem to work. This implies that i'll get
something practical within a few months, and I'd like to know
what features most people want. Any suggestions are welcome,
but I won't promise I'll implement them :-)
```

他说，他完成了一个好玩的小核心操作系统，这个核心是运行在 386 机器上的，同时他真的仅是好玩，并不是想要做一个跟 GNU 一样大的计划。这则新闻引起很多人的注意，他们也去 Linus Torvalds 提供的网站上下载了这个核心来安装。有趣的是，因为 Linus Torvalds 放置核心的那个 FTP 网站的目录为 Linux，从此大家便称这个核心为 Linux 了[1]。

为了让 Linux 能够兼容于 UNIX 系统，Linus Torvalds 开始参考标准的 POSIX[2]规范来“修

[1]注意，此时的 Linux 就是那个 kernel，而且 Linus Torvalds 所放到该目录下的第一个核心版本为 0.02。

[2]POSIX 是可携式操作系统接口（Portable Operating System Interface）的缩写，重点在于规范内核与应用程序之间的接口，这是由美国电气与电子工程师学会（IEEE）发布的标准。

改 Linux”。

POSIX 标准主要是针对 UNIX 与一些软件运行时候的标准规范，只要依据这些标准规范来设计的核心与软件，理论上就可以搭配在一起执行。

Linux 依据 POSIX 标准进行开发，因此 UNIX 中的软件也可以与 Linux 相兼容，同时 Linux 直接放置在网络上进行传播，所以 Linux 的使用率大增。

Linux 的第一个版本在 1991 年 9 月被大学 FTP server 管理员 Ari Lemmke 发布在 Internet 上。

最初，Torvalds 称这个内核的名称为“Freax”（意思是自由（“free”）和奇异（“freak”）的结合字），并且附上了“X”这个常用的字母来配合类 UNIX 的系统，但是 FTP 服务器管理员嫌原来的命名“Freax”的名称不好听，把内核的称呼改成“Linux”。

当时，Linux 仅有 10000 行程序码，仍必须运行于 Minix 操作系统之上，并且必须使用硬盘开机，随后在 10 月份第二个版本（0.02 版）就发布了，同时这位芬兰赫尔辛基的大学生在 comp.os.minix 上发布一则信息

Hello everybody out there using minix- I’m doing a (free) operation system (just a hobby, won’t be big and professional like gnu) for 386(486) AT clones.

为了让 Linux 可以在商业上使用，Linus Torvalds 决定改变原来的协议并使用 GNU GPL 协议来代替，开发者可以致力于融合 GNU 元素到 Linux 中，从而开发出一个有完整功能的、自由的操作系统。

1994 年 3 月，Linux1.0 版正式发布，Marc Ewing 成立了 Red Hat 软件公司，成为最著名的 Linux 经销商之一。

早期 Linux 的 boot loader 使用的 LILO 存在着一些难以容忍的缺陷（例如无法识别 1024 柱面以后的硬盘空间），后来的 GRUB[3]（GRand Unified Bootloader）克服了这些缺点，并具有‘动态搜索内核文件’的功能，可以让用户在开机时自行编辑开机设置文件来从 ext2 或 ext3 文件系统中加载 Linux Kernel。

Linux 具有设备独立性，而且 Linux 内核具有高度适应能力，从而给系统提供了更高级的功能。另外，GNU 用户接口组件提供常用的 C 函数库、shell 以及 UNIX 实用工具来完成许多基本的操作系统任务。

6.3 GNU/Linux

最初的 Linux 虽然是 Linus Torvalds 发明的，而且内容还绝不会涉及专利软件的版权问题。不过，如果单靠 Linus Torvalds 自己一个人的话，那么 Linux 要继续发展壮大实在是很困难。

[3]GRUB 通过不同的文件系统驱动可以识别几乎所有 Linux 支持的文件系统，因此可以使用很多文件系统来格式化内核文件所在的扇区，并不局限于 ext 文件系统。

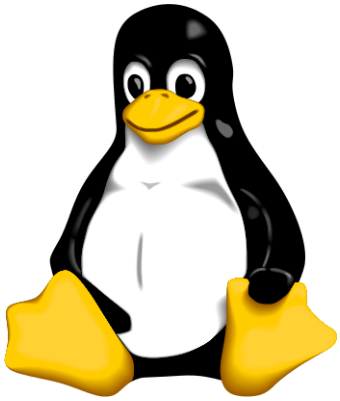


Figure 6.2: Tux - the penguin, mascot of Linux

一个人的力量是很有限的，Linus Torvalds 选择 Linux 的开发方式相当的务实，他首先将发布的 Linux 核心放置在 FTP 上面，并请告知大家新的版本信息，等到用户下载了这个核心并且安装之后，如果发生问题或者是由于特殊需求亟需某些硬件的驱动程序，那么这些用户就会主动汇报给 Linus Torvalds。在 Linus Torvalds 能够解决的问题范围内，他都能很快速的进行 Linux 核心的更新与除错。

不过，Linus Torvalds 总是有些硬件无法取得，那么他当然无法帮助进行驱动程序的编写与相关软件的改进，这个时候就会有些志愿者出来说：“这个硬件我有，我来帮忙写相关的驱动程序。”因为 Linux 的核心是 Open Source 的，志愿者们很容易就能够跟随 Linux 的设计架构，并且写出相容的驱动程序或者软件。

对于志愿者们开发的驱动程序与软件，Linus Torvalds 首先它们加入核心中并且加以测试，只要测试可以运行，并且没有什么主要的大问题，那么他就会很乐意的将志愿者们写的程序代码加入核心中，总之 Linus Torvalds 是个很务实的人，对于 Linux 核心所欠缺的项目，他总是“先求有且能运行，再求进一步改进”的心态，这让 Linux 用户与志愿者得到相当大的鼓励。

另外，Linux 逐渐开发成具有模块（module）的功能来顺应程序代码的加入，也就是将某些功能独立于核心外，在需要的时候才载入到内核空间中。如果有新的硬件驱动程序或者其他协议的程序代码进来时就可以模块化，而且模块化之后原先的核心不需要变动，大大的增加了 Linux 核心的可维护性。

Linux 核心在加入了太多的功能后，光靠 Linus Torvalds 一个人进行核心的实际测试并加入核心原始程序实在太费力，结果就有很多的朋友（例如 Alan Cox 与 Stephen Tweedie 等）来帮忙处理这个前置操作，这些重要的副手会先将来自志愿者们的修补程序或者新功能的程序代码进行测试，并且将结果上传给 Linus Torvalds 作出最后核心加入的源代码的选择与整合，这个分层负责的结果让 Linux 的开发更加的容易。

1994 年发布了 Linux 1.0 正式版，同时还加入了 X Window System 的支持，接着于 1996

年完成了 2.0 版，同时顺应商业版本的需求开始将核心版本以测试版及稳定版同时开发，次版本偶数为稳定版，奇数为开发中的测试版。

可以使用如下的命令查看 Linux 的内核版本编号：

```
[root@linux ~]$ uname -r
3.17.3-200
```

主版本号.次版本号.发布版本-修改版本

例如，2.6 与 2.5 版为相同的版本，不过 2.6 为稳定版，2.5 则为测试版，其中测试版含有较多的功能，不过稳定性还不确定，并且 Linus Torvalds 指明了企鹅为 Linux 的吉祥物。

Linus Torvalds 将内核的开发版本分为两个，并根据这两个内核的开发分别给予不同的内核编号。

- 开发中版本（development）：主、次版本号为奇数
  主要用在测试与开发新功能，通常这种版本仅有内核开发工程师会使用。新增的内核程序代码会加到这种版本中，等到测试通过后才加入下一版的稳定内核中。
- 稳定版本（stable）：主、次版本号为偶数
  内核功能开发成熟后会加到这类的版本中，主要用在家用计算机与企业版本中，重点在于提供给用户一个相对稳定的 Linux 操作系统环境平台。
- 发布版本（release）
  在主、次版本架构不变的情况下，新增的功能累积到一定的程度后所发布的内核版本。

Linux 内核版本与 Linux 发行版并不相同，Linux 版本指的是内核版本，不同的 Linux 发行版使用的都是 Linux 内核，但是版本号并不相同，而且不同的发行版所选用的软件以及它们自己开发的工具并不相同，多少还是有一定差异。

基于 Linux 的系统是一个模块化的类 UNIX 操作系统，而且 Linux 的大部分设计思想来源于 UNIX 操作系统所创建的基本设计思想。现在，大多数商业 Linux 发行版依然将操作系统称为 Linux，或者将操作系统的内核叫作 Linux，Linux 发行版是在 Linux 内核的基础上加入各种 GNU 工具打包而成的。

大多数 Linux 系统使用的 GUI 创建在 X 窗口系统基础上，并由 X 窗口系统通过软件工具及架构协议来创建操作系统所用的图形用户界面。

默认情况下，Linux 操作系统包含的组件包括启动器、init 程序、软件库以及 GUI 程序。

- 启动程序——例如 GRUB 或 LILO。该程序在计算机开机启动的时候运行，并将 Linux 内核加载到内存中。
- init 程序。init 是由 Linux 内核创建的第一个进程，称为根进程，所有的系统进程都是它的子进程，即所有的进程都是通过 init 启动。init 启动的进程如系统服务和登录提示（图形或终端模式的选择）。
- 软件库包含代码，可以通过运行的进程。在 Linux 系统上使用 ELF 格式来执行文件，负责管理库使用的动态链接器是“ld-linux.so”。Linux 系统上最常用的软件库是 GNU

C 库。

- 图像用户界面程序，例如命令行程序 bash shell 或窗口环境。

6.4 Features

早在 1994 年 Linux 1.0 版发布时，就已经含有 XFree86 的 X Window 架构了。

不过，X Window 毕竟是 Linux 上的一个软件，并不是 Linux 最核心的部分，有没有它对 Linux 的服务器运行都没有影响。现在，KDE[4]及 GNOME[5]等优秀的视窗管理程序广泛用于 Linux Desktop 计算机中。

6.4.1 Standards

Linux 是基于 UNIX 概念而开发出来的操作系统，因此 Linux 具有与 UNIX 系统相似的的程序接口跟操作方式，也继承了 UNIX 稳定并且有效率的特点。

Linux 支持多种标准，例如开发遵循 POSIX 规范。

另外，为了让软件开发商、与硬件开发者有一个依循的方向，因此发布了 Linux Standard Base（LSB）及 File system Hierarchy Standard（FHS）等。

Linux 发行版都要遵循 LSB 规范，软硬件开发者也都会依循 LSB，所以不同的 Linux 发行版只是在提供的工具与创意上面有所不同，但是基本上它们的架构都是很类似的。

- FHS: http://www.pathname.com/fhs/
- LSB: http://www.linuxbase.org/

另外，现在 Linux 发行版已经可以独力完成几乎所有的工作站或服务器的服务（例如 Web、Mail、Proxy、FTP 等），因此 Linux 发行版一般被用来作为服务器的操作系统，并且已经在该领域中占据重要地位。

Linux 发行版是构成 LAMP[6]（Linux 操作系统，Apache，MySQL，Perl/PHP/Python）的重要部分。

另外，FreeBSD 也具备完整的 NIS 客户端和服务器端支持，事务式 TCP 协议支持，按需拨号的 PPP，集成的 DHCP 支持，改进的 SCSI 子系统，ISDN 的支持，ATM，FDDI，快速 Gigabit 以太网 (1000 Mbit) 支持，并且提升了最新的 Adaptec 控制器的支持等。

6.4.2 Multi-user & Multi-tasking

要实现多任务（multitasking）的环境，除了硬件（主要是 CPU）需要能够具有多任务的特性外，操作系统也需要支持这个功能。

[4] http://www.kde.org/

[5] http://www.gnome.org/

[6] LAMP 是一个常见的网站托管平台，在开发者中已经得到普及。

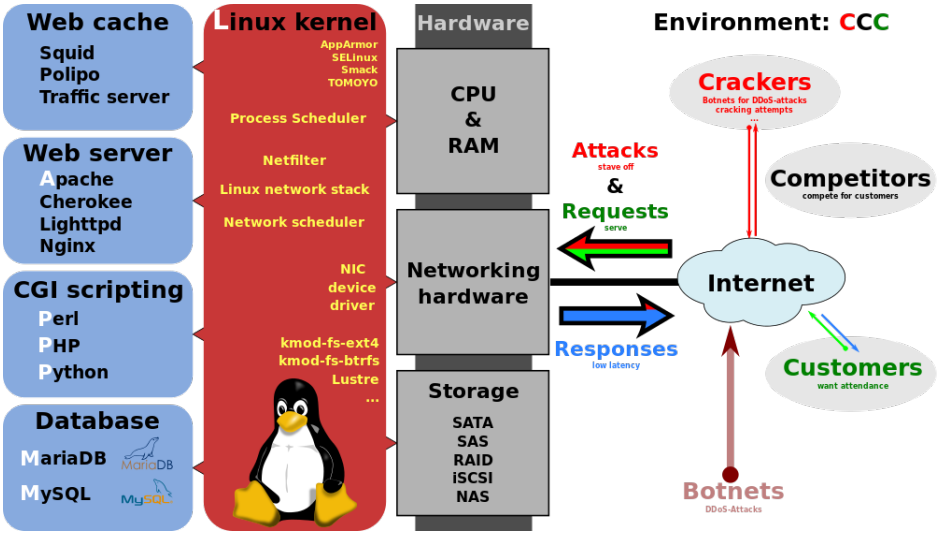


Figure 6.3: Broad overview of the LAMP software bundle

一些不具有多任务特性的操作系统，想要同时执行两个程序是不可能的。除非先被执行的程序执行完毕，否则后面的程序不可能被主动执行，而在多任务的操作系统中，每个程序被执行时，都会有一个最大 CPU 使用时间，若某一个任务运行的时间超过这个 CPU 使用时间，该任务就会先被移出 CPU 的工作序列，而再次的进入核心工作调度中，等待下一次被 CPU 取来继续执行。

以记者发布会为例，主持人（CPU）会问“谁要发问”？一群记者（工作程序）就会举手（看谁的工作重要），先举手的自然就被允许发问，问完之后，主持人又会问一次谁要发问，当然所有人（包括刚刚那个记者）都可以举手，如此一次一次的将工作完成，多任务的环境对于复杂的工作情况帮助很大。

Linux 主机上可以同时允许多用户同时在线工作，并且资源的分配较为公平，而且 Linux 支持不同等级的用户，而且每个用户登录系统时的工作环境都可以不相同。此外，还可以允许不同的用户在同一个时间登入主机以同时使用主机的资源。

Linux 系统的文件属性可以分为“可读、可写、可执行”等参数来定义一个文件的适用性，而且这些属性还可以分为三个种类——分别是“文件拥有者、文件所属群组、其他非拥有者与用户组者”，从而对项目计划或者开发者提供良好的安全性。

6.4.3 Embedded Devices

Linux 内核可以进行裁剪并完整地驱动整个计算机的硬件并成为一个完整的操作系统，因此也适合作为电子设备的操作系统。

在工业界，开发人员通过裁剪 Linux kernel 来使其运行在省电以及较低硬件资源的环境

下，比如手机、数码相机、PDA、家电用品等嵌入式系统。

Linux 的低成本、强大的定制功能以及良好的移植性能，使得 Linux 在嵌入式系统方面也得到广泛应用。例如，流行的 TiVo 数字视频录像机采用了定制的 Linux，思科在网络防火墙和路由器也使用了定制的 Linux。

在手机、平板电脑等移动设备方面，Linux 也得到重要发展，基于 Linux 内核的操作系统也成为最广泛的操作系统。例如，基于 Linux 内核的 Android 操作系统已经成为当今全球最流行的智能手机操作系统。

- Palm 推出了一个新的基于 Linux 的 webOS 操作系统，并应用在 Palm Pre 智能手机上。
- MeeGo 是诺基亚和英特尔于 2010 年 2 月联合推出的基于 Linux 的操作系统。
- MeeGo 与 LiMo 合并成为新的系统 Tizen。
- Jolla Mobile 公司推出了由 MeeGo 发展而来的 Sailfish 操作系统。

另外，FreeBSD 通过整合虚拟内存/文件系统中的高速缓存改进了虚拟内存系统，不仅提升了性能，而且减少了 FreeBSD 对内存的需要，使得 5 MB 内存成为可接受的最小配置。

Chapter 7

Debian

7.1 Overview

Debian 是由 GPL 和其他自由软件许可协议授权的自由软件组成的操作系统，由 Debian 计划（Debian Project）组织维护。

Debian 以其坚守 UNIX 和自由软件的精神，以及其给予用户的众多选择而闻名。

作为一个大的系统组织框架，在 Debian 框架下有多种不同操作系统核心的分支计划，主要为采用 Linux 核心的 Debian GNU/Linux 系统，采用 GNU Hurd 核心的 Debian GNU/Hurd 系统以及采用 FreeBSD 核心的 Debian GNU/kFreeBSD 系统，还有采用 NetBSD 核心的 Debian GNU/NetBSD 系统等。

在这些不同的 Debian 系统中，以采用 Linux 核心的 Debian GNU/Linux 最为著名，例如众多的 Linux 发布版（包括 Ubuntu、Knoppix 和 Linspire 及 Xandros 等）都建构于 Debian GNU/Linux 的基础上。

1993 年 8 月 16 日由美国普渡大学学生 Ian Murdock 首次发表 Debian，最初称为“Debian Linux Release”，并且在定义文件 Debian Manifesto 中宣布将以开源的方式，本着 Linux 及 GNU 的精神发布 GNU/Linux 发布版。

- 在 1994 年和 1995 年分别发布了 0.9x 版本。
- 1.x 版本则在 1996 年发布。

1996 年，布鲁斯 · 佩伦斯接替了 Ian Murdock 成为了 Debian 计划的领导者。同年，开发者 Ean Schuessler 提议 Debian 应在其计划与用户之间创建一份社区契约，后来经过讨论由布鲁斯 · 佩伦斯发表了 Debian 社区契约及 Debian 自由软件指导方针，定义了开发 Debian 的基本承诺。

把 Debian 移植至其他内核的工作正在进行，其中 Hurd 是一组在微内核（例如 Mach）上运行的服务器，而且 Debian 操作系统中的大部分基本工具来自于 GNU 计划，因此也命名为 GNU/Linux 和 GNU/Hurd，这些工具同样都是自由的。

7.2 Packages

1998 年在建基于 GNU C 运行期库的 Debian 2.0 发布之前，布鲁斯 · 佩伦斯离开了 Debian 的开发工作。

第一个建基于 Debian 的 Linux 发布版 Corel Linux 和 Stormix 的 Storm Linux 在 1999 年开始开发，它们两个发布版成为了建基于 Debian 的 Linux 发布版的先驱。

在 2000 年后半年，Debian 对数据库和发布的管理作出了重大的改变，并重组了收集软件的过程以及创造了“测试”（testing）版本作为较稳定的对下一个发布的演示。同年，Debian 的开发者开始举办名为 Debconf 的年会，为其开发者和技术专家提供讲座和工作坊。

正在开发中的软件会被上载到名为“不稳定”（unstable，代号 sid）和“实验性”（experimental）的计划分支上。

- 上载至“不稳定”分支上的软件通常是由软件的原开发者发布的稳定版本，但包含了一些未经测试的 Debian 内部的修改（例如软件的打包）。
- 未达到“不稳定”分支要求的软件会被置于“实验性”分支。

处于“不稳定”分支的软件经过修改和测试后将会自动转移至“测试”分支，但是如果软件有严重错误被报告，或其所依存的软件不符合“测试”分支的要求，该软件则不会被移至“测试”分支。

在“测试”分支中的软件三年没有回报一个 bug 后，“测试”分支会成为下一个稳定版本，因此 Debian 官方发布的正式版本并不包含新的特色，因此桌面用户可以选择安装“测试”甚至“不稳定”分支，只是这两个分支所进行的测试比稳定版本少些，可能较不稳定，也没有定时的安全更新。

Debian 的软件管理系统为 APT，亦有图形界面的 synaptic 和 aptitude 可供使用，这些软件包都已经被编译包装为一种方便的格式 deb 包。

7.3 Branches

Debian 以稳定性闻名，所以很多服务器都使用 Debian 作为其操作系统，Debian 主要分三个版本：稳定版本（stable）、测试版本（testing）、不稳定版本（unstable）。

- 目前的稳定版本为 Debian Wheezy。
- 目前的测试版本为 Debian Jessie。
- 不稳定版本永远为 Debian sid[1]。

很多 Linux 的 LiveCD 亦以 Debian 为基础改写，最为著名的例子为 Knoppix，而且 Debian 在桌面领域的修改版 Ubuntu Linux 也获得了很多 Linux 用户的支持。

[1]Debian sid 也称为 Debian unstable，即不稳定版本，凡是 Debian 要收录的软件都必须首先放在这个版本里面进行测试，等到足够稳定以后会放到 testing 版本里面。

相比 Ubuntu、Fedora 等 Linux 发布版，较少桌面用户会选择使用 Debian，主要原因是其基于较新功能的考量。

Debian Project 独立运作，不带有任何商业性质，不依附任何商业公司或者机构，使得它能够有效地坚守其信奉的自由理念和风格，因此 Debian 对 GNU 和 UNIX 精神的坚持，也获得开源社区和自由软件或开源软件信奉者的支持。

不过，Debian 的发布周期较长，稳定版本的包可能已经过时，而且 Debian 很大程度上是为“不动的”平台（例如服务器和用于开发的机器）设计，而这些平台只需要安全性的更新。

Chapter 8

Ubuntu

8.1 Overview

Ubuntu[5] 是一个以桌面应用为主的基于 Debian 发布版和 GNOME 桌面环境的 GNU/Linux 操作系统，其名称来自非洲南部祖鲁语或科萨语的“ubuntu”一词，意思是“人性”、“我的存在是因为大家的存在”，代表了非洲传统的一种价值观。

Ubuntu 由马克·舍特尔沃斯创立，其首个版本（4.10）发布于 2004 年 10 月 20 日，后续的每个新版本均会包含当时最新的 GNOME 桌面环境，通常在 GNOME 发布新版本后一个月内发布，后来在在 11.04 版以后开始采用 Unity 桌面。

Ubuntu 与 Debian 的不同在于它每 6 个月会发布一个新版本，每 2 年发布一个 LTS 长期支持版本[1]。

Ubuntu 建构于 Debian 的不稳定分支（不论其软件格式（deb）还是软件管理与安装系统（apt-get），而且 Ubuntu 的开发者会把对软件的修改实时反馈给 Debian 社区。事实上，很多 Ubuntu 的开发者同时也是 Debian 主要软件的维护者。不过 Debian 与 Ubuntu 的软件并不一定完全兼容，也就是说，将 Debian 的包安装在 Ubuntu 上可能会出现兼容性问题，反之亦然。

普通的 Ubuntu 桌面版可以获得发布后 18 个月内的支持，LTS（长期支持）的桌面版可以获得更长时间的支持。例如，Ubuntu 8.04 LTS（代号 Hardy Heron），其桌面应用系列可以获得为期 3 年的技术支持，服务器版可以获得为期 5 年的技术支持。

自 Ubuntu 12.04 LTS 开始，桌面版和服务器版均可获得为期 5 年的技术支持[2]。

Ubuntu 12.04 桌面版与服务器版都有 5 年支持周期，而之前的长期支持版本为桌面版 3

[1] 与 Debian 稳健的升级策略不同，Ubuntu 每六个月便会发布一个新版，以便人们实时地获取和使用新软件。

[2] Ubuntu 计划在 4 月 25 日 Ubuntu 13.04 发布后，将非 LTS 版本的支持时间自 18 个月缩短至 9 个月，并采用滚动发布模式，允许开发者在不升级整个发布版的情况下升级单个核心包。

年，服务器版 5 年。

2013 年，Ubuntu 推出了新产品 Ubuntu Phone OS 和 Ubuntu Tablet，意图统一桌面设备和移动设备的屏幕。

8.2 Features

Ubuntu 所有系统相关的任务均需使用 sudo[3]指令是其一大特色，这种方式比传统的以系统管理员账号进行管理工作的方式更为安全。

Ubuntu 的开发者与 Debian 和 GNOME 开源社区合作密切，其各个正式版本的桌面环境均采用 GNOME 的最新版本，通常会紧随 GNOME 项目的进展而及时更新，同时也提供基于 KDE、XFCE 等桌面环境的派生版本。

Ubuntu 与 Debian 使用相同的 deb 软件包格式，可以安装绝大多数为 Debian 编译的软件包，虽不能保证完全兼容，但大多数情况是通用的。

Ubuntu 计划强调易用性和国际化来让尽可能多的人可以使用，例如 Ubuntu 使用 Unicode 作为系统默认编码来处理不同的语言文字。与其它大型 Linux 厂商不同，Ubuntu 不对所谓“企业版”收取升级订购费（意即没有所谓的企业版本），用户所使用的版本皆一样，用户只有在购买官方技术支持服务时才要付费。

8.2.1 Installation

Ubuntu 支持主流的 i386、AMD64 与 PowerPC 平台，并且 Ubuntu 从 2006 年 6 月开始新增了对 UltraSPARC 与 UltraSPARC T1 平台的支持。

由 Ubuntu 母公司 Canonical 有限公司所架设的 Launchpad 网站提供了在线翻译服务，任何人都可以通过这个网站协助翻译 Ubuntu，只是经由此方式对非 Ubuntu 独有组件的翻译成果将不会自动反馈到上游。

8.2.2 Package

Ubuntu 能够使用的软件大多存放在被称为“软件源”的服务器中，用户只要运行相应的 apt-get 指令（或使用 Synaptic 工具进行相关操作），系统就会自动查找、下载和安装软件。

Ubuntu 的包管理系统与 Debian 的类似，所有软件分为 main、restricted、universe 和 multiverse 等 4 类，每一类为一个“组件（component）”并代表着不同的使用许可和可用的支持级别。

[3]sudo 为 substitute user do 的简写，即超级用户的工作，在 Ubuntu 的默认环境里，root（即管理员）账号是停用的，所有与系统相关的工作指令均需在进行时在终端机接口输入 sudo 在指令并输入密码确认，这样做是为了防止因一时失误对系统造成破坏。sudo 工具的默认密码是目前账户的密码。

- main 即“基本”组件，其中只包含符合 Ubuntu 的许可证要求并可以从 Ubuntu 团队中获得支持的软件，致力于满足日常使用，位于这个组件中的软件可以确保得到技术支持和及时的安全更新。

  “main”组件内的软件是必须符合 Ubuntu 版权要求（Ubuntu license requirements）的自由软件，而 Ubuntu 版权要求大致上与 Debian 自由软件指导纲要（Debian Free Software Guidelines）相同。
- restricted 即“受限”组件，其中包含了非常重要的，但并不具有合适的自由许可证的软件，例如只能以二进制形式获得的显卡驱动程序，不过 restricted 组件能够获得的支持与 main 组件相比是非常有限的。
- universe 即“社区维护”组件，其中包含的软件种类繁多，均为自由软件，但都不为 Ubuntu 团队所支持。
- multiverse 即“非自由”组件，其中包括了不符合自由软件要求而且不被 Ubuntu 团队支持的软件包，通常为商业公司编写的软件。

一般来说，官方支持的 main 组件主要用来满足大多数个人计算机用户的基本要求，restricted（“版权限制”）组件主要用来提高系统的可用性，因此通常需要安装这两类组件中的软件。

	自由软件	非自由软件
官方支持	Main	Restricted
非官方支持	Universe	Multiverse

Ubuntu 接纳的部分可以自由散发的私有软件位于 multiverse 组件中，而且 Ubuntu 还为第三方软件设立了认证程序。

Ubuntu Grumpy Groundhog 的分支直接从 Ubuntu 的软件版本控制系统里获取软件的源代码，主要用于测试和开发，不对公众开放。

Ubuntu 的新版发布后就会冻结该版本的包库，此后只对该包库提供安全性更新，为此 Ubuntu 官方推出了后续支持计划 Ubuntu Backports，让用户可以在不更新包库的情况下，获得和使用各类新版的应用软件。Launchpad 网站提供了在线提交软件程序错误的机制，任何人都可以把自己所发现的软件程序错误、功能缺陷和安全漏洞发送给开发小组。

Ubuntu 系统中默认带有 ufw 防火墙软件，但不提供相应的图形设置界面，用户可自行安装 firestarter，以便通过图形界面设置防火墙。

在 Ubuntu 中安装软件时可以通过运行 apt-get 命令，或使用图形接口的 Synaptic 工具以及“软件中心”来完成。

Chapter 9

Licensing

Linux 其实就是一个操作系统最底层的核心及其提供的核心工具，任何人均可取得源代码并能够执行 Linux 核心程序以及进行修改。

此外，Linux 遵循 POSIX 设计规范并兼容于 UNIX 操作系统，故亦可称之为 UNIX-like 的一种。[1]

现在，GNU 软件大多以 Linux 为主要操作系统来进行开发，此外很多其他的自由软件团队（例如 sendmail、wu-ftp、apache 等）也都有以 Linux 为开发测试平台的计划出现。

由 Torvalds 负责开发的 Linux 提供了 kernel 与 kernel 相关的工具，而且应用软件也已经可以在 Linux 上面运行，因此 Linux + 软件就可以组成一个相当完整的操作系统。

为了让用户能够接触到 Linux，很多的商业公司或非营利团体就将 Linux kernel（含 tools）与可运行的软件整合起来，加上自己具有创意的工具程序就可以称为 Linux distribution（Linux 发行版），因此 Linux 发布版指的就是通常所说的“Linux 操作系统”，它可能是由一个组织、公司或者个人发布的。

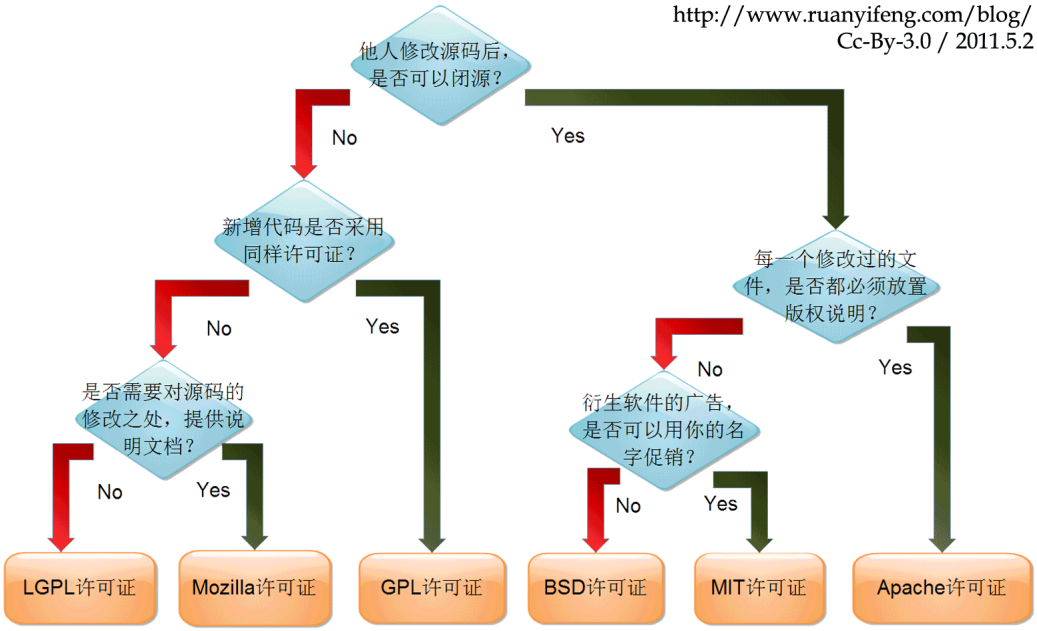
不同的 Linux 发行版差异性并不大，只是套件管理的方式主要分为 Debian 的 `dpkg` 及 Red Hat 系统的 `RPM` 方式。

现今存在的开源协议很多，而经过 Open Source Initiative 组织通过批准的开源协议目前有 58 种（http://www.opensource.org/licenses/alphabetical），其中常见的开源协议（如 BSD、GPL、LGPL 和 MIT 等）都是 OSI 批准的协议。

开源软件或开源组件都会基于某种协议[7] 来提供源码和授权，其代表授权为 GNU 的 GPL 及 BSD 等，而且这些开源协议确实具有相应的约束。

如果要开源自己的代码，最好也是选择这些被批准的开源协议，下面是乌克兰程序员Paul Bagwell画的开源协议分析图，说明应该怎么选择开源协议，后来由阮一峰[8] 将其翻译成中文如下：

[1]在 Linux 发布之前，GNU 一直以来就是缺乏核心程序，导致 GNU 自由软件只能在其他的 UNIX 上面运行。



9.1 OpenSource

9.1.1 GPL

在自由软件所使用的各种许可证之中，最为人们注意的也许是通用性公开许可证(General Public License，简称 GPL)[9]。目前有 version 2 和 version 3 两种版本，主要定义在“自由软件”上面，任何遵循 GPL 授权的软件，需要公布其源代码（Open Source），Linux 使用的是 version 2 这一版。

GPL 同其它的自由软件许可证一样，许可社会公众享有运行、复制软件的自由，发行传播软件的自由，获得软件源码的自由，改进软件并将自己作出的改进版本向社会发行传播的自由。

GPL 还规定只要这种修改文本在整体上或者其某个部分来源于遵循 GPL 的程序，该修改文本的整体就必须按照 GPL 发布，不仅该修改文本的源码必须向社会公开，而且对于这种修改文本的流通不准许附加修改者自己作出的限制。

遵循 GPL 发布的程序不能同非自由的软件合并，因此 GPL 所表达的这种流通规则称为 copyleft，表示与 copyright(版权) 的概念“相左”。具体说来，GPL 有如下几个主要的大方向：

1. 任何个人或公司均可发布自由软件（free software）；
2. 任何发布自由软件的个人或公司，均可由自己的服务来收取适当的费用；
3. 该软件的源代码（Source Code）需要随软件附上，并且是可公开发表的；
4. 任何人均可通过任何正常管道取得此自由软件，且均可取得此一授权模式。

GPL 协议最主要的几个原则可以概括为：

- 确保软件自始至终都以开放源代码形式发布，保护开发成果不被窃取用作商业发售。
- 只要其中使用了受 GPL 协议保护的第三方软件的源程序，那么向非开发人员发布时，软件本身也就自动成为受 GPL 保护并且约束的实体。
- GPL 精髓就是只要使软件在完整开源的情况下，尽可能使使用者得到自由发挥的空间，使软件得到更快更好的发展。
- 无论软件以何种形式发布，都必须同时附上源代码。例如在 Web 上提供下载，就必须在二进制版本（如果有的话）下载的同一个页面，清楚地提供源代码下载的链接。如果以光盘形式发布，就必须同时附上源文件的光盘。
- 开发或维护遵循 GPL 协议开发的软件的公司或个人，可以对使用者收取一定的服务费用，但是必须无偿提供软件的完整源代码，不得将源代码与服务做捆绑或任何变相捆绑销售。

9.1.2 BSD

BSD 授权模式授权模式其实与 GPL 很类似，而其精神也与 Open Source 相呼应。

BSD 开源协议[10] 是一个给于使用者很大自由的协议，例如可以自由的使用，修改源代码，也可以将修改后的代码作为开源或者专有软件再发布。

当发布使用了 BSD 协议的代码，或者以 BSD 协议代码为基础做二次开发自己的产品时，需要满足三个条件：

1. 如果再发布的产品中包含源代码，则在源代码中必须带有原来代码中的 BSD 协议。
2. 如果再发布的只是二进制类库/软件，则需要在类库/软件的文档和版权声明中包含原来代码中的 BSD 协议。
3. 不可以用开源代码的作者/机构名字和原来产品的名字做市场推广。

BSD 代码鼓励代码共享，但需要尊重代码作者的著作权。

BSD 允许使用者修改和重新发布代码，也允许使用或在 BSD 代码上开发商业软件发布和销售，因此是对商业集成很友好的协议。很多的公司企业在选用开源产品的时候都首选 BSD 协议，因为可以完全控制这些第三方的代码，在必要的时候可以修改或者二次开发。

9.1.3 Apache

Apache Licence[11] 是著名的非盈利开源组织 Apache 采用的协议。该协议和 BSD 类似，同样鼓励代码共享和尊重原作者的著作权，同样允许代码修改，再发布（作为开源或商业软件）。

在 Apache 的授权中规定，如果想要重新发布此软件时（如果修改过该软件），软件的名称依旧需要命名为 Apache 才可以，因此使用 Apache 许可证时需要满足的条件也和 BSD 类似

- 需要给代码的用户一份 Apache Licence

- 如果你修改了代码，需要在被修改的文件中说明。
- 在延伸的代码中（修改和有源代码衍生的代码中）需要带有原来代码中的协议，商标，专利声明和其他原来作者规定需要包含的说明。
- 如果再发布的产品中包含一个 Notice 文件，则在 Notice 文件中需要带有 Apache Licence。
- 可以在 Notice 中增加自己的许可，但不可以表现为对 Apache Licence 构成更改。

Apache Licence 也是对商业应用友好的许可，使用者也可以在需要的时候修改代码来满足需要并作为开源或商业产品发布/销售。

9.1.4 MIT

MIT 许可证[12] 源自麻省理工学院（Massachusetts Institute of Technology），又称为「X 条款」（X License）或「X11 条款」（X11 License）

MIT 是和 BSD 一样宽范的许可协议，作者只想保留版权，而无任何其他了限制。也就是说，你必须在你的发行版里包含原许可协议的声明，无论你是以二进制发布的还是以源代码发布的。

其中，MIT 内容与三条款 BSD 许可证（3-clause BSD license）内容颇为近似，但是赋予软体被授权人更大的权利与更少的限制。

- 被授权人有权利使用、复制、修改、合并、出版发行、散布、再授权及贩售软体及软体的副本。
- 被授权人可根据程式的需要修改授权条款为适当的内容。
- 在软件和软件的所有副本中都必须包含版权声明和许可声明。

MIT 授权条款并非属 copyleft 的自由软体授权条款，允许在自由/开放源码软体或非自由软体（proprietary software）所使用，这也是 MIT 与 BSD（The BSD license, 3-clause BSD license）本质不同。

MIT 条款可与其他授权条款并存。另外，MIT 条款也是自由软体基金会（FSF）所认可的自由软体授权条款，与 GPL 相容。

9.1.5 MPL

MPL[13]（The Mozilla Public License）是 1998 年初 Netscape 的 Mozilla 小组为其开源软件项目设计的软件许可证。

MPL 许可证出现的最重要原因是 Netscape 公司认为 GPL 许可证没有很好地平衡开发者对源代码的需求和他们利用源代码获得的利益。和著名的 GPL 许可证和 BSD 许可证相比，MPL 在许多权利与义务的约定方面与它们相同（因为都是符合 OSIA 认定的开源软件许可证）。

不过，相比而言 MPL 还有以下几个显著的不同之处。

MPL 虽然要求对于经 MPL 许可证发布的源代码的修改也要以 MPL 许可证的方式再许可出来，以保证其他人可以在 MPL 的条款下共享源代码。但是，在 MPL 许可证中对“发布”的定义是“以源代码方式发布的文件”，这就意味着 MPL 允许一个企业在自己已有的源代码库上加一个接口，除了接口程序的源代码以 MPL 许可证的形式对外许可外，源代码库中的源代码就可以不用 MPL 许可证的方式强制对外许可。这些，就为借鉴别人的源代码用做自己商业软件开发的行为留了一个豁口。

MPL 许可证第三条第 7 款中允许被许可人将经过 MPL 许可证获得的源代码同自己其他类型的代码混合得到自己的软件程序。

MPL 许可证不像 GPL 许可证那样明确表示反对软件专利，但是却明确要求源代码的提供者不能提供已经受专利保护的源代码（除非他本人是专利权人，并书面向公众免费许可这些源代码），也不能在将这些源代码以开放源代码许可证形式许可后再去申请与这些源代码有关的专利。

在 MPL（1.1 版本）许可证中，对源代码的定义是“源代码指的是对作品进行修改最优先择取的形式，它包括所有模块的所有源程序，加上有关的接口的定义，加上控制可执行作品的安装和编译的‘原本’（原文为‘Script’），或者不是与初始源代码显著不同的源代码就是被源代码贡献者选择的从公共领域可以得到的程序代码。”

MPL 许可证第 3 条有专门的一款是关于对源代码修改进行描述的规定，就是要求所有再发布者都得有一个专门的文件就对源代码程序修改的时间和修改的方式有描述。

9.1.6 LGPL

LGPL[14] 是 GPL 的一个为主要为类库使用设计的开源协议。和 GPL 要求任何使用/修改/衍生之 GPL 类库的的软件必须采用 GPL 协议不同。LGPL 允许商业软件通过类库引用 (link) 方式使用 LGPL 类库而不需要开源商业软件的代码。这使得采用 LGPL 协议的开源代码可以被商业软件作为类库引用并发布和销售。

如果修改 LGPL 协议的代码或者衍生，则所有修改的代码，涉及修改部分的额外代码和衍生的代码都必须采用 LGPL 协议。因此 LGPL 协议的开源代码很适合作为第三方类库被商业软件引用，但不适合希望以 LGPL 协议代码为基础，通过修改和衍生的方式做二次开发的商业软件采用。

GPL/LGPL 都保障原作者的知识产权，避免有人利用开源代码复制并开发类似的产品。

9.2 Close Source

相对于 Open Soure，Close Source 仅推出可执行的二进制程序，程序的核心是封闭的，优点是有专人维护，不需要去更改它。缺点则是灵活度大打折扣，用户无法变更该程序成为自己想要的样式，此外若有木马程序或者安全漏洞，将会花上相当长的一段时间来除错，

这也是所谓专利软件（copyright）常见的软件销售方式。

闭源软件的代表授权模式包括 Freeware 和 Shareware。

不同于 Free software，Freeware 为“免费软件”而非“自由软件”。虽然它是免费的软件，但是不见得要公布其源代码，要看发布者的意见。

共享软件（Shareware）与免费软件有点类似的是，Shareware 在使用初期也是免费的，但是到了所谓的“试用期限”之后，就必须要选择“付费后继续使用”或者“将它移除”。通常，这些共享软件都会自行编写失效程序，在试用期限之后就无法使用该软件。

Bibliography

[1] Wikipedia. List of unix utilities (2008).
URL http://en.wikipedia.org/wiki/List_of_Unix_utilities.

[2] riku. Linux 与 bsd 的区别到底在哪里？(2010).
URL http://os.51cto.com/art/201008/217538.htm.

[3] Wikipedia. Linux (1991).
URL http://en.wikipedia.org/wiki/Linux.

[4] Wikipedia. Ubuntu (operating system) (2004).
URL http://en.wikipedia.org/wiki/Ubuntu_(operating_system).

[5] Wikipedia. Ubuntu (2004).
URL http://zh.wikipedia.org/zh/Ubuntu.

[6] Wikipedia. Tanenbaum–torvalds debate.
URL http://en.wikipedia.org/wiki/Tanenbaum%E2%80%93Torvalds_debate.

[7] TUNA. 开源 ≠ 免费，开源协议 license 详解 (2014).
URL http://blog.tektea.com/archives/3108.html.

[8] 阮一峰. 如何选择开源许可证？(2011).
URL http://www.ruanyifeng.com/blog/2011/05/how_to_choose_free_software_licenses.html.

[9] 红薯. 详细介绍 gpl 协议 (2010).
URL http://www.oschina.net/question/12_2826.

[10] 红薯. 详细介绍 bsd 开源协议 (2010).
URL http://www.oschina.net/question/12_2825.

[11] 红薯. 详细介绍 apache licene 2.0 协议 (2010).
URL http://www.oschina.net/question/12_2828.

[12] 红薯. 详细介绍 mit 协议 (2010).
URL http://www.oschina.net/question/12_2829.

[13] 红薯. 详细介绍 mpl (mozilla public license) 协议 (2010).
URL http://www.oschina.net/question/12_2830.

[14] 红薯. 五种开源协议的比较 (bsd,apache,gpl,lgpl,mit) (2010).
URL http://www.oschina.net/question/12_2664.

Part II

Foundation

Chapter 10

Overview

从打开电源到开始操作，计算机的启动是一个非常复杂的过程。[1]

启动计算机又称为 boot[1]，工程师们用 boot 来比喻计算机启动是一个很矛盾的过程——必须先运行程序，然后计算机才能启动，但是计算机不启动就无法运行程序。

早期启动计算机时必须想尽各种办法，把一小段程序装进内存，然后计算机才能正常运行。

CMOS 是嵌入到主板中的记录各种硬件参数的存储器，而 BIOS 则是写入其中的固件程序（firmware），而且 BIOS 就是计算机启动时系统主动执行的第一个程序。

10.1 BIOS

上个世纪 70 年代初，“只读内存”（read-only memory，缩写为 ROM）发明，开机程序被刷入 ROM 芯片。

最早的开机程序叫做“基本输入/输出系统”（Basic Input/Output System），简称为 BIOS。

BIOS 是一种用于 IBM PC 兼容机上的固件接口，是操作系统输入输出管理系统的一部分，现在可以将 BIOS 视为是一个永久地记录在 ROM 中的一个软件，也是个人电脑启动时加载的第一个软件。

计算机通电或被重置（reset）时，第一件事就是读取 BIOS，处理器第一条指令的地址会被定位到 BIOS 的存储器中，让初始化程序开始运行。

[1] boot 原来的意思是靴子，这里的 boot 是 bootstrap（鞋带）的缩写，它来自一句谚语：“pull oneself up by one's bootstraps”，其字面意思是” 拽着鞋带把自己拉起来”，这当然是不可能的事情。

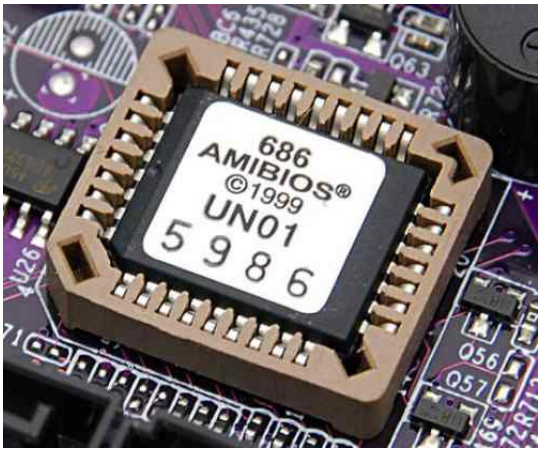


Figure 10.1: BIOS

早期的 BIOS 芯片确实是”只读”的，里面的内容是用一种烧录器写入的，一旦写入就不能更改，除非更换芯片。

现在的主机板都使用 Flash EPROM 芯片来存储系统 BIOS，里面的内容可通过使用主板厂商提供的擦写程序擦除后重新写入，这样就给用户升级 BIOS 提供了极大的方便。

在计算机启动的过程中，BIOS 担负着初始化硬件，执行系统各部分的的自检以及启动引导程序或装载在内存的操作系统的责任。

此外，BIOS 还向操作系统提供一些系统参数，例如系统硬件的变化就可以由 BIOS 隐藏，因而程序可以使用 BIOS 服务而不是直接访问硬件，现代操作系统会忽略 BIOS 提供的抽象层（HAL）并直接访问硬件。

当计算机的电源打开，BIOS 就会由主板上的闪存运行，并将芯片组和存储器子系统初始化。具体来说，BIOS 会把自己从闪存中解压缩到系统的主存，然后从主存开始运行。

BIOS 代码也包含诊断功能来保证某些重要硬件组件（比如 CPU、内存、硬盘设备、键盘、输出输入端口等）可以正常运作且正确地初始化。

几乎所有的 BIOS 都可以选择性地运行 CMOS 存储器的设置程序，也就是保存 BIOS 会访问的用户自定义设置数据（时间、日期、硬盘细节等）。

现代的 BIOS 可以配置 PnP（Plug and Play，即插即用）设备并定义出可启动的设备顺序，从而可以让用户选择由哪个设备（例如光盘驱动器、硬盘、软盘和 USB 闪存盘等）来启动计算机。

另外，有些 BIOS 系统允许用户可以选择要加载哪个操作系统（例如从其他硬盘加载操作系统），虽然这项功能通常是由第二阶段的开机管理程序（Bootloader）来处理。

10.1.1 POST

BIOS 的功能由两部分组成，分别是 POST 码和 Runtime 服务。

- POST 阶段完成后它将从存储器中被清除。
- Runtime 服务会被一直保留，用于目标操作系统的启动。

BIOS 程序执行系统完整性检查时，首先检查计算机硬件能否满足运行的基本条件，这叫做“开机自我检测”（Power-On Self-Test，缩写为 POST），主要负责检测系统外围关键设备（如 CPU、内存、显卡、I/O、键盘鼠标等）是否正常。例如，最常见的是内存松动的情况，BIOS 自检阶段会报错，系统就无法启动起来。

1. 开机系统重置 REST 启动 CPU。
2. CPU 指向 BIOS 自我测试的地址 FFFFOH 并打开 CPU 运行第一个指令。
3. CPU 内部暂存器的测试。
4. CMOS 146818 SRAM 检查。
5. ROM BIOS 检查码测试。
6. 8254 计时/计数器测试。
7. 8237 DMA 控制器测试。
8. 74612 页暂存器测试。
9. REFRESH 刷新电路测试。
10. 8042 键盘控制器测试。
11. DRAM 64KB 基本存储器测试。
12. CPU 保护模式的测试。
13. 8259 中断控制器的测试。
14. CMOS 146818 电力及检查码检查。
15. DRAM IMB 以上存储器检查。
16. 显卡测试。
17. NMI 强制中断测试。
18. 8254 计时/计数器声音电路测试。
19. 8254 计时/计数器计时测试。
20. CPU 保护模式 SHUT DOWN 测试。
21. CPU 回至实模式 (REAL MODE)。
22. 键盘鼠标测试。
23. 8042 键盘控制器测试。
24. 8259 中断控制器 IRQ0 至 IRQ18 创建。
25. 硬盘驱动器及界面测试。
26. 设置并行打印机及串行 RS232 的界面。
27. 检查 CMOS IC 时间、日期。

28. 检查完成

开机自检结果会显示在固件可以控制的输出接口，像显示器[2]、LED、打印机等等设备上。如果硬件出现问题，主板会发出不同含义的蜂鸣，启动中止。

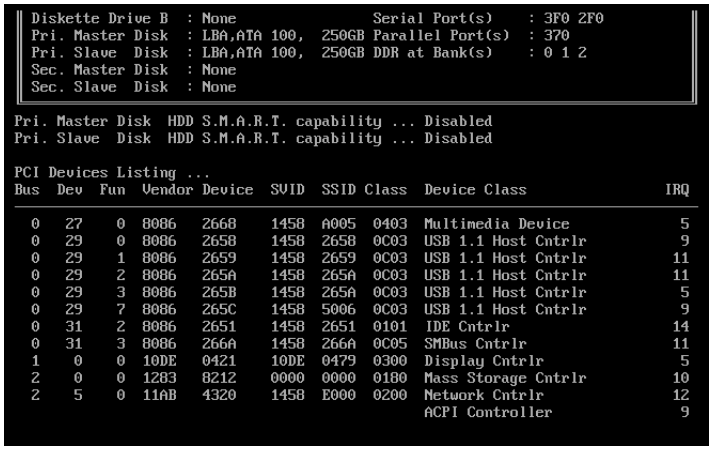


Figure 10.2: 硬件自检

计算机系统中可以包含多个 BIOS 固件芯片，其中开机 BIOS 主要是包含访问基本硬件组件（例如键盘或软盘驱动器）的代码。额外的适配器（例如 SCSI/SATA 硬盘适配器、网络适配器、显卡等）也会包含他们自己的 BIOS，补充或取代系统 BIOS 代码中有关这些硬件的部份。

为了在开机时找到这些存储器映射的扩充只读存储器，PC BIOS 会扫描物理内存，从 0xC0000 到 0xF0000 的 2KB 边界中查找 0x55 0xaa 记号，接在其后的是一个比特，表示有多少个扩充只读存储器的 512 位区块占据真实存储器空间。接着 BIOS 马上跳跃到指向由扩充只读存储器所接管的地址，以及利用 BIOS 服务来提供用户设置接口，注册中断矢量服务供开机后的应用程序使用，或者显示诊断的信息。

确切地说扩展卡上的 ROM 不能称之为 BIOS。它只是一个程序片段，用来初始化自身所在的扩展卡。

10.1.2 Boot Sequence

硬件自检完成后，BIOS 便会执行一段小程序用来枚举本地设备并对其初始化，这一步主要是根据我们在 BIOS 中设置的系统启动顺序来搜索用于启动系统的驱动器，例如硬盘、光盘、U 盘、软盘和网络等。

BIOS 中有一个外部储存设备的排序，排在前面的设备就是优先转交控制权的设备，这种排序叫做 “启动顺序”（Boot Sequence），可以在 BIOS 中 “设定启动顺序”。

[2]如果没有显示器，我们可以通过 POST CARD 来完成上面的测试工作。

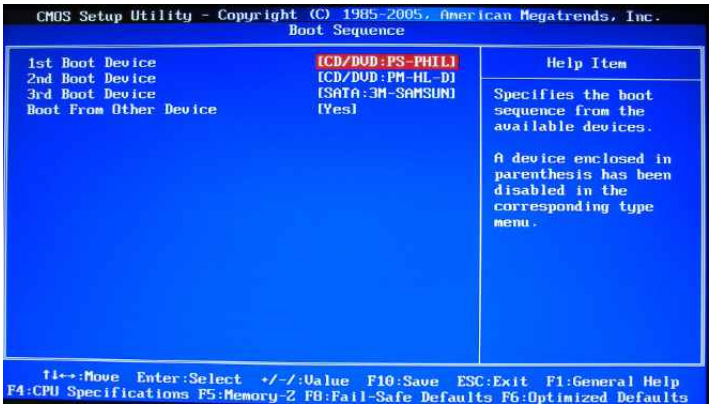


Figure 10.3: 设定启动顺序

这时，BIOS 需要知道，“下一阶段的启动程序” 具体存放在哪一个设备，然后 BIOS 就把控制权转交给下一阶段的启动程序（Bootloader）。

接下来，计算机读取启动设备的第一个扇区，也就是读取最前面的 512 个字节。如果这 512 个字节的最后两个字节是 0x55 和 0xAA，表明这个设备可以用于启动；如果不是，表明设备不能用于启动，控制权于是被转交给 “启动顺序” 中的下一个设备。

以硬盘启动为例，BIOS 此时去读取硬盘驱动器的第一个扇区，然后执行里面的代码。硬盘的 0 柱面、0 磁头、1 扇区称为主引导扇区，“主引导记录”（Master Boot Record，缩写为 MBR）就位于其第一块扇区内。实际上，这里 BIOS 并不关心启动设备第一个扇区中是什么内容，它只是负责读取该扇区内容并执行。

至此，BIOS 的任务就完成了，此后将系统启动的控制权移交到 MBR 部分的代码。

10.2 UEFI

统一可扩展固件接口（Unified Extensible Firmware Interface，UEFI[2]）是一种替代 BIOS 的个人计算机系统规格，用来定义操作系统与系统固件之间的软件界面。

UEFI 和 BIOS 这两个技术都在计算机启动的时候发出第一个命令指示，并使得操作系统能够被顺利加载。UEFI 负责加电自检（POST）、连系操作系统以及提供连接操作系统与硬件的接口。

UEFI 原名本为可扩展固件接口（Extensible Firmware Interface），最初是由英特尔开发并于 2002 年 12 月发布其订定的版本-1.1 版。英特尔于 2005 年将此规范格式交由 UEFI 论坛来推广与发展，UEFI 论坛于 2007 年 1 月 7 日发布 2.1 版本的规格，其中较 1.1 版本增加与改进了加密编码（cryptography）、网络认证（network authentication）与用户接口架构（User Interface Architecture）。

2009 年 5 月 9 日，UEFI 论坛发布 2.3 版本，最新的公开的版本是 2.3.1。

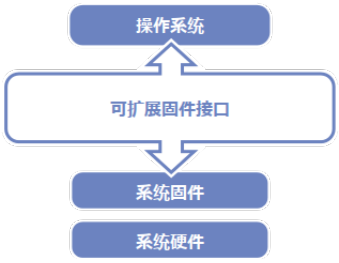


Figure 10.4: 可扩展固件接口在软件层次中的位置

10.2.1 EFI

EFI 和 BIOS 二者显著的区别就是 EFI 是用模块化，C 语言风格的参数堆栈传递方式，动态链接的形式构建的系统，较 BIOS 而言更易于实现，容错和纠错特性更强，缩短了系统研发的时间。EFI 运行于 32 位或 64 位模式，乃至未来增强的处理器模式下，突破传统 16 位代码的寻址能力，达到处理器的最大寻址。

EFI 利用加载 EFI 驱动的形式来识别和操作硬件，不同于 BIOS 利用挂载真实模式中断的方式增加硬件功能。后者必须将一段类似于驱动的 16 位代码，放置在固定的 0x000C0000 至 0x000DFFFF 之间存储区中，运行这段代码的初始化部分，它将挂载实模式下约定的中断矢量向其他程序提供服务。例如，VGA 图形及文本输出中断（INT 10h），硬盘访问中断服务（INT 13h）等等。由于这段存储空间有限（128KB），BIOS 对于所需放置的驱动代码大小超过空间大小的情况无能为力。

另外，BIOS 的硬件服务程序都以 16 位代码的形式存在，这就给运行于增强模式的操作系统访问其服务造成了困难。因此 BIOS 提供的服务在现实中只能提供给操作系统引导程序或 MS-DOS 类操作系统使用。而 EFI 系统下的驱动并不是由可以直接运行在 CPU 上的代码组成的，而是用 EFI Byte Code 编写而成的。这是一组专用于 EFI 驱动的虚拟机器语言，必须在 EFI 驱动运行环境（Driver Execution Environment，或 DXE）下被解释运行。这就保证了充分的向下兼容性，打个比方说，一个带有 EFI 驱动的扩展设备，既可以将其安装在安腾处理器的系统中，也可以安装于支持 EFI 的新 PC 系统中，而它的 EFI 驱动不需要重新编写。这样就无需对系统升级带来的兼容性因素作任何考虑。

由于 EFI 驱动开发简单，所有的 PC 部件提供商都可以参与，情形非常类似于现代操作系统的开发模式。基于 EFI 的驱动模型可以使 EFI 系统接触到所有的硬件功能，在操作系统运行以前浏览万维网站不再是天方夜谭，甚至实现起来也非常简单。这对基于传统 BIOS 的系统来说是件不可能的任务，在 BIOS 中添加几个简单的 USB 设备支持都曾使很多 BIOS 设计师痛苦万分，更何况除了添加对无数网络硬件的支持外，还得凭空构建一个 16 位模式下的 TCP/IP 协议栈。

一般认为，EFI 由以下几个部分组成：

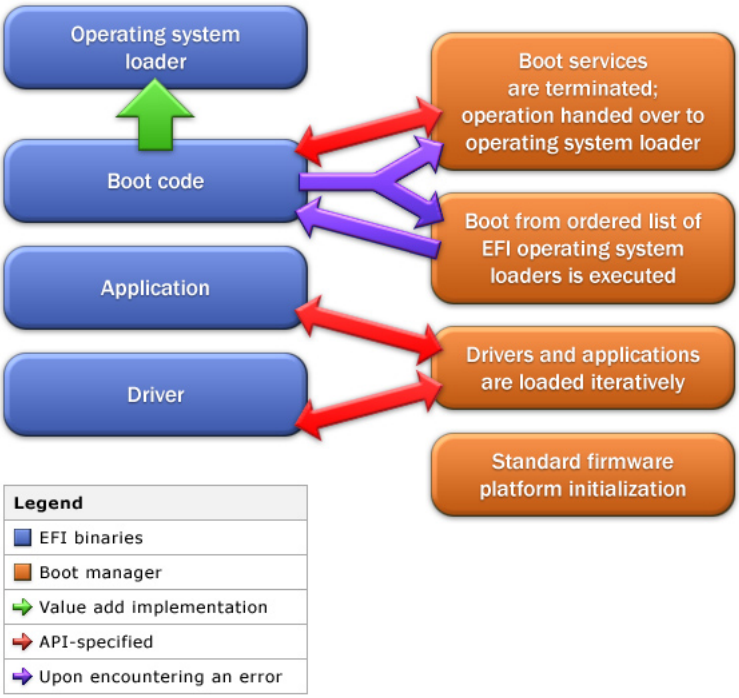


Figure 10.5: EFI boot manager and EFI drivers

1. Pre-EFI 初始化模块
2. EFI 驱动执行环境
3. EFI 驱动程序
4. 兼容性支持模块 (CSM)
5. EFI 高层应用
6. GUID 硬盘分区表 (GPT)

EFI 在概念上非常类似于一个低阶的操作系统，并且具有操控所有硬件资源的能力，因此 EFI 将有可能代替现代的操作系统。事实上，EFI 的缔造者们在第一版规范出台时就将 EFI 的能力限制于不足以威胁操作系统的统治地位。

- 首先，它只是硬件和预启动软件间的接口规范；
- 其次，EFI 环境下不提供中断的机制，也就是说每个 EFI 驱动程序必须用轮询 (polling) 的方式来检查硬件状态，并且需要以解释的方式运行，较操作系统下的机械码驱动效率更低；
- 再则，EFI 系统不提供复杂的缓存器保护功能，它只具备简单的缓存器管理机制，具体来说就是指运行在 x86 处理器的段保护模式下，以最大寻址能力为限把缓存器分为一个平坦的段 (Segment)，所有的程序都有权限访问任何一段位置，并不提供真实的保护服务。

当 EFI 所有组件加载完毕时，系统可以开启一个类似于操作系统 Shell 的命令解释环境，用户可以在其中执行任何 EFI 应用程序，这些程序可以是硬件检测、引导管理或设置软件等。

理论上来说，对于 EFI 应用程序的功能并没有任何限制，任何人都可以编写这类软件，并且效果较以前 MS-DOS 下的软件更华丽，功能更强大。一旦引导软件将控制权交给操作系统，所有用于引导的服务代码将全部停止工作。

在实现中，EFI 初始化模块和驱动执行环境通常被集成在一个只读存储器中。Pre-EFI 初始化程序在系统开机的时候最先得到执行，它负责最初的 CPU，主桥及存储器的初始化工作，紧接着载入 EFI 驱动执行环境 (DXE)。当 DXE 被载入运行时，系统便具有了枚举并加载其他 EFI 驱动的能力。在基于 PCI 架构的系统中，各 PCI 桥及 PCI 适配器的 EFI 驱动会被相继加载及初始化；这时，系统进而枚举并加载各桥接器及适配器后面的各种总线及设备驱动程序，周而复始，直到最后一个设备的驱动程序被成功加载。正因如此，EFI 驱动程序可以放置于系统的任何位置，只要能保证它可以按顺序被正确枚举。例如一个具 PCI 总线接口的 ATAPI 大容量存储适配器，其 EFI 驱动程序一般会放置在这个设备的符合 PCI 规范的扩展只读存储器 (PCI Expansion ROM) 中，当 PCI 总线驱动被加载完毕，并开始枚举其子设备时，这个存储适配器旋即被正确识别并加载它的驱动程序。部分 EFI 驱动程序还可以放置在某个硬盘的 EFI 专用分区中，只要这些驱动不是用于加载这个硬盘的驱动的必要部件。

在 EFI 规范中，一种突破传统 MBR 硬盘分区结构限制的 GUID 硬盘分区系统 (GPT) 被引入，硬盘的分区数不再受限制 (在 MBR 结构下，只能存在 4 个主分区)。

另外，EFI/UEFI+GUID 结合还可以支持 2.1 TB 以上硬盘[3]，并且分区类型将由 GUID 来表示。

在众多的分区类型中，EFI 系统分区可以被 EFI 系统访问，用于存放部分驱动和应用程序，不过 EFI 系统比传统的 BIOS 更易于受到计算机病毒的攻击，当一部分 EFI 驱动程序被破坏时，系统有可能面临无法引导的情况。

实际上，系统引导所依赖的 EFI 驱动部分通常都不会存放在 EFI 的 GUID 分区中，因此即使分区中的驱动程序遭到破坏，也可以用简单的方法得到恢复，这与操作系统下的驱动程序的存储习惯是一致的。

CSM 是在 x86 平台 EFI 系统中的一个特殊的模块，它将为不具备 EFI 引导能力的操作系统提供类似于传统 BIOS 的系统服务。

10.3 MBR

BIOS 搜索并加载执行 Boot loader 程序后，接下来 BIOS 把控制权转交给排在第一位的存储设备。

[3]有测试显示，3TB 硬盘使用 MBR，并且安装 Windows 6.x 64 位系统，只能识别到 2.1TB。

在 ROM BIOS 检查结束时，接下来 BIOS 会按照 “启动顺序” 来搜索并执行系统中第一个启动设备的第一个物理扇区（sector）中的启动程序（Bootloader）[4]。

简单的 Bootloader 的虚拟汇编码，如其后的八个指令:

0. 将 P 暂存器的值设为 8；
1. 检查纸带 (paper tape) 读取器，是否已经可以进行读取；
2. 如果还不能进行读取，跳至 1；
3. 从纸带读取器，读取一 byte 至累加器；
4. 如为带子结尾，跳至 8；
5. 将暂存器的值，存储至 P 暂存器中的数值所指定的地址；
6. 增加 P 暂存器的值；
7. 跳至 1。

Bootloader 程序使用分区表（partition table）来确定哪个分区是可引导的（通常是第一个主分区）并尝试从该分区引导。其中，硬盘的第一个物理扇区称为主引导记录（Master Boot Record，MBR[3]），又叫做主引导扇区或主引导块，它是计算机开机后访问硬盘时所必须要读取的首个扇区。

主引导扇区的读取流程如下:

1. BIOS 加电自检 (POST)。BIOS 执行内存地址为 FFFF:0000H 处的跳转指令，跳转到固化在 ROM 中的自检程序处，对系统硬件 (包括内存) 进行检查。
2. 读取主引导记录 (MBR)。当 BIOS 检查到硬件正常并与 CMOS 中的设置相符后，按照 CMOS 中对启动设备的设置顺序检测可用的启动设备。BIOS 将相应启动设备的第一个扇区 (也就是 MBR 扇区) 读入内存地址为 0000:7C00H 处并跳转执行。
3. 检查 0000:7CFEH-0000:7CFFH(MBR 的结束标志位) 是否等于 55AAH，若不等于则转去尝试其他启动设备，如果没有启动设备满足要求则显示 “NO ROM BASIC” 然后死机。
4. 当检测到有启动设备满足要求后，BIOS 将控制权交给相应启动设备。启动设备的 MBR 将自己复制到 0000:0600H 处, 然后继续执行。
5. 根据 MBR 中的引导代码启动引导程序。

事实上，BIOS 不仅检查 0000:7CFEH-0000:7CFFH(MBR 的结束标志位) 是否等于 55AAH，往往还对硬盘是否有写保护、主引导扇区中是否存在活动分区等进行检查。如果发现硬盘有写保护，则显示硬盘写保护出错信息；如果发现硬盘中不存在活动分区，则显示类似如下的信息“Remove disk or other media Press any key to restart”。

对于硬盘而言，MBR 在硬盘上的三维地址为:

$$（柱面，磁头，扇区）= （0，0，1）$$

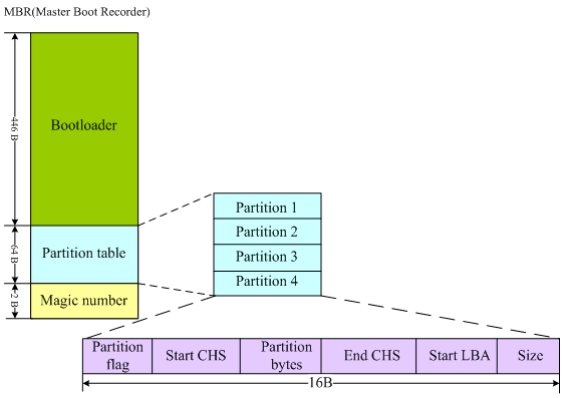
[4]BIOS 可以通过硬件的 INT 13 中断功能来读取 MBR。也就是说，只要 BIOS 可以检测到硬盘，就可以通过 INT 13 硬件中断来读取该硬盘的第一个扇区内的 MBR，这样 Bootloader 就能够被执行。

而一个扇区可能的字节数为 128×2^n (n=0,1,2,3)，大多数情况下取 n=2，因此一个扇区（sector）的大小为 512 字节。

MBR 记录着硬盘本身的相关信息以及硬盘各个分区的大小及位置信息。具体而言，MBR 由三个部分组成，分别是主引导程序 (Bootloader)、硬盘分区表 DPT（Disk Partition Table）和硬盘有效标志（55AA）。

MBR 一共占用 512bytes，其中：

1. Bootloader（第 1 ～ 446 字节）：调用操作系统的机器码。
2. DPT（第 447 ～ 510 字节）：硬盘分区表（Disk Partition table）。
3. MBR 验证信息（第 511 ～ 512 字节）：主引导记录签名（0x55 和 0xAA）[5]。



在硬盘的主引导记录最开头是第一阶段引导代码，其中的硬盘主引导程序的主要作用是检查分区表是否正确并且在系统硬件完成自检以后将控制权交给硬盘上的引导程序（如 GNU GRUB[6]），该程序使用分区信息来确定哪个分区是可引导的并尝试从该分区引导。

在深入讨论主引导扇区内部结构的时候，有时也将其开头的 446 字节内容特指为“主引导记录”（MBR），其后是 4 个 16 字节的“硬盘分区表”（DPT）以及 2 字节的结束标志（55AA），因此在使用“主引导记录”（MBR）这个术语的时候，需要根据具体情况判断其到底是指整个主引导扇区，还是主引导扇区的前 446 字节。

MBR 的主要作用是告诉计算机到硬盘的哪一个位置去找操作系统，分区信息（或分区表）存储在该扇区的末尾处。如果 MBR 受到破坏，硬盘上的基本数据结构信息将会丢失，需要用繁琐的方式试探性的重建数据结构信息后才可能重新访问原先的数据。主引导扇区内的信息可以通过任何一种基于某种操作系统的分区工具软件[7]写入，但和某种操作系统没有特定的关系，即只要创建了有效的主引导记录就可以引导任意一种操作系统（操作系统是创建在高级格式化的硬盘分区之上，是和一定的文件系统相联系的）。

[5]结束标志字 55 和 AA（偏移 1FEH －偏移 1FFH）是主引导扇区的最后两个字节，是检验主引导记录是否有效的标志。

[6]通常情况下，诸如 LILO、GUN GRUB 这些常见的引导程序都直接安装在 MBR 中。

[7]例如，MS-DOS Fdisk 实用工具通常仅当不存在主引导记录时才会更新主引导记录 (MBR)。

标准 MBR 结构

地址			描述		长度 (字节)
Hex	Oct	Dec			
0000	0000	0	代码区		**440** (最大 446)
01B8	0670	440	选用磁盘标志		4
01BC	0674	444	一般为空值; 0x0000		2
01BE	0676	446	**标准 MBR 分区表规划** (四个16 byte的主分区表入口)		**64**
01FE	0776	510	55h	MBR 有效标志: 0x55AA	**2**
01FF	0777	511	AAh		
MBR, 总大小: 446 + 64 + 2 =					**512**

Figure 10.6: 标准 MBR 结构

10.3.1 DPT

硬盘分区表（Disk Partition Table，简写为 DPT）的作用是将硬盘分成若干个区。硬盘分区有很多好处，比如考虑到每个区可以安装不同的操作系统，此时 MBR 就必须知道该将控制权转交给哪个分区。

DPT 的长度只有 64 个字节（偏移 01BEH ～偏移 01FDH），里面又分成四项记录，每项 16 个字节，因此一个硬盘最多只能分四个一级分区（又叫做“主分区”（Primary））。

每个主分区记录的 16 个字节，由 6 个部分组成，记录了开始和结束柱面号码等。

1. 第 1 个字节：如果为 0x80，就表示该主分区是激活分区，控制权要转交给这个分区。四个主分区里面只能有一个是激活的。
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。

根据 16 字节分区表的结构：当前分区的扇区数用 4 个字节表示，前面各分区扇区数的总和也是 4 个字节，而 $2^{32} \times 512$⌧2 199 023 255 552Byte。

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。

Table 10.1: 硬盘分区结构信息

偏移	长度 (字节)	意义	备注
00H	1	分区状态	00–> 非活动分区;
			80–> 活动分区[8];
			其它数值没有意义
01H	1		分区起始磁头号 (HEAD)，用到全部 8 位
02H	2		分区起始扇区号 (SECTOR)，占据 02H 的位 0 － 5;
			该分区的起始磁柱号 (CYLINDER)，占据 02H 的位 6 － 7 和 03H 的全部 8 位
04H	1		文件系统标志位
05H	1		分区结束磁头号 (HEAD)，用到全部 8 位
06H	2		分区结束扇区号 (SECTOR)，占据 06H 的位 0 － 5;
			该分区的结束磁柱号 (CYLINDER)，占据 06H 的位 6 － 7 和 07H 的全部 8 位
08H	4		分区起始相对扇区号
0CH	4		分区总的扇区数

下面是一个例子，如果某一分区在硬盘分区表的信息如下:

80 01 01 00 0B FE BF FC 3F 00 00 00 7E 86 BB 00

则我们可以看到，

- 最前面的 “80” 是一个分区的激活标志，表示系统可引导;
- “01 01 00” 表示分区开始的磁头号为 1，开始的扇区号为 1，开始的柱面号为 0;
- “0B” 表示分区的系统类型是 FAT32，其他比较常用的有 04（FAT16)、07（NTFS);
- “FE BF FC” 表示分区结束的磁头号为 254，分区结束的扇区号为 63、分区结束的柱面号为 764;
- “3F 00 00 00” 表示首扇区的相对扇区号为 63;
- “7E 86 BB 00” 表示总扇区数为 12289662。

对于现代大于 8.4G 的硬盘，CHS 已经无法表示, BIOS 使用 LBA 模式，对于超出的部分，CHS 值通常设为 FEFFFF 并加以忽略，直接使用 08-0f 的 4 字节相对值，再进行内部转换。

硬盘的分区个数还要受到分区大小的限制，具体来说，硬盘分区是按照柱面来进行分区的。

[8]对于一个操作系统而言，系统分区设为活动分区并不是必须的，这主要视引导程序而定，如果使用的引导程序是 Grub4Dos，MBR 中的引导代码仅仅按照分区的顺序依次探测第二阶段引导器 grldr 的位置，并运行第一个探测到的 grldr 文件。

一个分区至少要占一个柱面，现代硬盘的寻址结构不再是 CHS 寻址，柱面大小不同于相关软件显示的柱面大小，因此对于物理结构上有 n 个面的硬盘，其分区空间的最小值为：

$$n \times 扇区/磁道 \times 512字节$$

主引导记录仅仅包含一个 64 个字节的硬盘分区表，每个分区信息需要 16 个字节，因此采用 MBR 型分区结构的硬盘最多只能识别 4 个主要分区（Primary partition）。

对于一个采用 MBR 分区结构的硬盘来说，想要得到 4 个以上的主要分区是不可能的，这时就需要引入扩展分区。扩展分区也是主要分区的一种，但它与主分区的不同在于，扩展分区类似于独立的磁盘空间，理论上可以在扩展分区中划分出无数个逻辑分区。

扩展分区中逻辑驱动器的引导记录是链式的，每一个逻辑分区都有一个和 MBR 结构类似的扩展引导记录 (EBR)，其分区表的第一项指向该逻辑分区本身的引导扇区，第二项指向下一个逻辑驱动器的 EBR，分区表第三、第四项没有用到。

如果分区超过 4 个，就一定要有扩展分区，而且必须将所有剩下的空间都分配给扩展分区，然后再以逻辑分区来规划扩展分区的空间。早期的 Windows 系统一般都是只划分一个主分区给系统，剩余的部分全部划入扩展分区。

考虑到磁盘的连续性，一般建议将扩展分区的柱面号码分配在最后面的柱面内。

- 在 MBR 分区表中最多 4 个主分区或者 3 个主分区 + 1 个扩展分区，也就是说扩展分区只能有一个，然后可以再细分为多个逻辑分区。
- 在 Linux 系统中，硬盘分区命名为 sda1 − sda4 或者 hda1 − hda4（其中 a 表示硬盘编号可能是 a、b、c 等等）。在 MBR 硬盘中，分区号 1 − 4 是主分区（或者扩展分区），逻辑分区号只能从 5 开始。
- 在 MBR 分区表中，一个分区最大的容量为 2T，且每个分区的起始柱面必须在这个 disk 的前 2T 内。

如果使用 MBR 分区表，对于 3T 以上的硬盘，根据要求至少要把它划分为 2 个分区，且最后一个分区的起始扇区要位于硬盘的前 2T 空间内，否则必须改用 GPT。

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。再考虑到扇区的逻辑地址也是 32 位，所以单个硬盘可利用的空间最大也不超过 2TB。如果想使用更大的硬盘，只有 2 个方法：一是提高每个扇区的字节数，二是增加扇区总数。

与支持最大卷为 2 TB 并且每个硬盘最多有 4 个主分区（或 3 个主分区，1 个扩展分区和无限制的逻辑驱动器）的 MBR 硬盘分区的样式相比，GPT 硬盘分区样式支持最大卷为 128 EB（Exabytes）并且每硬盘的分区数没有上限，只受到操作系统限制（由于分区表本身需要占用一定空间，最初规划硬盘分区时，留给分区表的空间决定了最多可以有多少个分区，IA-64 版 Windows 限制最多有 128 个分区，这也是 EFI 标准规定的分区表的最小尺寸）。

与 MBR 分区的硬盘不同，GPT 分区表中至关重要的平台操作数据位于分区，而不是位于非分区或隐藏扇区。

另外，GPT 分区硬盘有备份分区表来提高分区数据结构的完整性。

10.4 GPT

全局唯一标识分区表（GUID Partition Table，GPT[4]）是一个实体硬盘的分区表的结构布局的标准。

GPT 是可扩展固件接口（EFI）标准的一部分，被用于替代 BIOS 系统中的 32bits 来存储逻辑块地址和大小信息的主引导记录（MBR）分区表。对于那些扇区为 512 字节的硬盘，MBR 分区表不支持容量大于 2.2TB（2.2×10^{12} 字节）的分区。然而，一些硬盘制造商（诸如希捷和西部数据）注意到了这个局限性，并且将他们的容量较大的硬盘升级到了 4KB 的扇区，这意味着 MBR 的有效容量上限提升到了 16 TiB。这个看似“正确的”解决方案，在临时地降低了人们对改进硬盘分配表的需求的同时，也给市场带来了关于在有较大的块 (block) 的设备上从 BIOS 启动时，如何最佳的划分硬盘分区的困惑。

GPT 分配 64bits 给逻辑块地址，因而使得最大分区大小在 $2^{64}-1$ 个扇区成为了可能。对于每个扇区大小为 512 字节的硬盘，那意味着可以有 9.4ZB(9.4 x 10^{21} 字节) 或 8 ZiB 个 512 字节（9,444,732,965,739,290,426,880 字节或 18,446,744,073,709,551,615（$2^{64}-1$）个扇区 x 512（29）字节每扇区）。

在 MBR 硬盘中，分区信息直接存储于主引导记录（MBR）中（主引导记录中还存储着系统的引导程序）。但在 GPT 硬盘中，分区表的位置信息储存在 GPT 头中。但出于兼容性考虑，硬盘的第一个扇区仍然用作 MBR，之后才是 GPT 头。

跟现代的 MBR 一样，GPT 也使用逻辑区块地址（LBA）取代了早期的 CHS 寻址方式。传统 MBR 信息存储于 LBA 0，GPT 头存储于 LBA 1，接下来才是分区表本身。64 位 Windows 操作系统使用 16,384 字节（或 32 扇区）作为 GPT 分区表，接下来的 LBA 34 是硬盘上第一个分区的开始。

苹果公司曾经警告说：“不要假定所有设备的块大小都是 512 字节。”一些现代的存储设备如固态硬盘可能使用 1024 字节的块，一些磁光盘（MO）可能使用 2048 字节的扇区（但是磁光盘通常是不进行分区的）。一些硬盘生产商在计划生产 4096 字节一个扇区的硬盘，但截至 2010 年初，这种新硬盘使用固件对操作系统伪装成 512 字节一个扇区。

为了减少分区表损坏的风险，GPT 在硬盘最后保存了一份分区表的副本。

对于不标准的 MBR/GPT 混合硬盘，不同的系统中的实现有些不一致。除非另加说明，操作系统在处理混合硬盘时优先读取 GPT 分区表

10.4.1 Legacy MBR

在 GPT 分区表的最开头，处于兼容性考虑仍然存储了一份传统的 MBR，用来防止不支持 GPT 的硬盘管理工具错误识别并破坏硬盘中的数据，这个 MBR 也叫做保护 MBR。在支持从 GPT 启动的操作系统中，这里也用于存储第一阶段的启动代码。

在这个 MBR 中，只有一个标识为 0xEE 的分区，以此来表示这块硬盘使用 GPT 分区表。不能识别 GPT 硬盘的操作系统通常会识别出一个未知类型的分区，并且拒绝对硬盘进行操

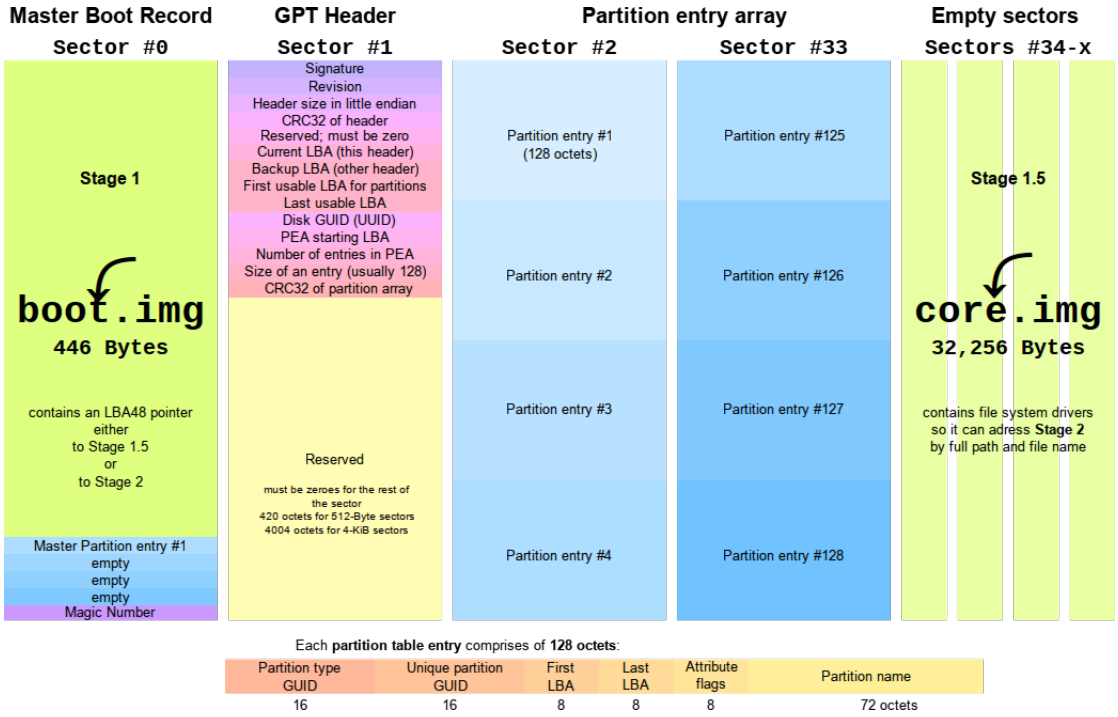


Figure 10.7: GRUB 位于 GPT 分区的示意图

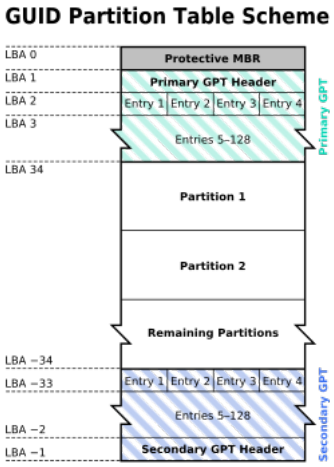


Figure 10.8: GPT 分区表的结构。此例中，每个逻辑块（LBA）为 512 字节，每个分区的记录为 128 字节。负数的 LBA 地址表示从最后的块开始倒数，–1 表示最后一个块。

作，除非用户特别要求删除这个分区。这就避免了意外删除分区的危险。另外，能够识别 GPT 分区表的操作系统会检查保护 MBR 中的分区表，如果分区类型不是 0xEE 或者 MBR 分区表中有多个项，也会拒绝对硬盘进行操作。

在使用 MBR/GPT 混合分区表的硬盘中，这部分存储了 GPT 分区表的一部分分区（通常是前四个分区），可以使不支持从 GPT 启动的操作系统从这个 MBR 启动，启动后只能操作 MBR 分区表中的分区。如 Boot Camp 就是使用这种方式启动 Windows。

10.4.2 Partition Table Header

分区表头定义了硬盘的可用空间以及组成分区表的项的大小和数量。在使用 64 位 Windows Server 2003 的机器上，最多可以创建 128 个分区，即分区表中保留了 128 个项，其中每个都是 128 字节。（EFI 标准要求分区表最小要有 16,384 字节，即 128 个分区项的大小）

分区表头还记录了这块硬盘的 GUID，记录了分区表头本身的位置和大小（位置总是在 LBA 1）以及备份分区表头和分区表的位置和大小（在硬盘的最后）。它还储存着它本身和分区表的 CRC32 校验。固件、引导程序和操作系统在启动时可以根据这个校验值来判断分区表是否出错，如果出错了，可以使用软件从硬盘最后的备份 GPT 中恢复整个分区表，如果备份 GPT 也校验错误，硬盘将不可使用。所以 GPT 硬盘的分区表不可以直接使用 16 进制编辑器修改。

Table 10.2: 分区表头的格式

起始字节	长度	内容
0	8 字节	签名（“EFI PART”, 45 46 49 20 50 41 52 54）
8	4 字节	修订（在 1.0 版中，值是 00 00 01 00）
12	4 字节	分区表头的大小（单位是字节，通常是 92 字节，即 5C 00 00 00）
16	4 字节	分区表头（第 0 − 91 字节）的 CRC32 校验，在计算时，把这个字段作为 0 处理，需要计算出分区串行的 CRC32 校验后再计算本字段
20	4 字节	保留，必须是 0
24	8 字节	当前 LBA（这个分区表头的位置）
32	8 字节	备份 LBA（另一个分区表头的位置）
40	8 字节	第一个可用于分区的 LBA（主分区表的最后一个 LBA + 1）
48	8 字节	最后一个可用于分区的 LBA（备份分区表的第一个 LBA – 1）
56	16 字节	硬盘 GUID（在类 UNIX 系统中也叫 UUID）
72	8 字节	分区表项的起始 LBA（在主分区表中是 2）
80	4 字节	分区表项的数量

起始字节	长度	内容
84	4 字节	一个分区表项的大小（通常是 128）
88	4 字节	分区串行的 CRC32 校验
92	*	保留，剩余的字节必须是 0（对于 512 字节 LBA 的硬盘即是 420 个字节）

主分区表和备份分区表的头分别位于硬盘的第二个扇区（LBA 1）以及硬盘的最后一个扇区。备份分区表头中的信息是关于备份分区表的。

10.4.3 Partition Entries

GPT 分区表使用简单而直接的方式表示分区。一个分区表项的前 16 字节是分区类型 GUID。例如，EFI 系统分区的 GUID 类型是 {C12A7328-F81F-11D2-BA4B-00A0C93EC93B}。接下来的 16 字节是该分区唯一的 GUID（这个 GUID 指的是该分区本身，而之前的 GUID 指的是该分区的类型)。再接下来是分区起始和末尾的 64 位 LBA 编号，以及分区的名字和属性。

Table 10.3: GPT 分区表项的格式

起始字节	长度	内容
0	16 字节	分区类型 GUID
16	16 字节	分区 GUID
32	8 字节	起始 LBA（小端序)
40	8 字节	末尾 LBA
48	8 字节	属性标签（例如 60 表示“只读”）
56	72 字节	分区名（可以包括 36 个 UTF-16 (小端序）字符）

10.4.4 Partition Type GUIDs

操作系统	分区类型	GUID
(None)	未使用	00000000-0000-0000-0000-000000000000
	MBR 分区表	024DEE41-33E7-11D3-9D69-0008C781F39F
	EFI 系统分区	C12A7328-F81F-11D2-BA4B-00A0C93EC93B
	BIOS 引导分区	21686148-6449-6E6F-744E-656564454649
Windows	微软保留分区	E3C9E316-0B5C-4DB8-817D-F92DF00215AE

操作系统	分区类型	GUID
	基本数据分区[9]	EBD0A0A2-B9E5-4433-87C0-68B6B72699C7
	逻辑软盘管理工具元数据分区	5808C8AA-7E8F-42E0-85D2-E1E90434CFB3
	逻辑软盘管理工具数据分区	AF9B60A0-1431-4F62-BC68-3311714A69AD
	Windows 恢复环境	DE94BBA4-06D1-4D40-A16A-BFD50179D6AC
	IBM 通用并行文件系统 (GPFS) 分区	37AFFC90-EF7D-4e96-91C3-2D7AE055B174
HP- UX	数据分区	75894C1E-3AEB-11D3-B7C1-7B03A0000000
	服务分区	E2A1E728-32E3-11D6-A682-7B03A0000000
Linux	数据分区	EBD0A0A2-B9E5-4433-87C0-68B6B72699C7
	RAID 分区	A19D880F-05FC-4D3B-A006-743F0F84911E
	交换分区	0657FD6D-A4AB-43C4-84E5-0933C84B4F4F
	逻辑卷管理器 (LVM) 分区	E6D6D379-F507-44C2-A23C-238F2A3DF928
	保留	8DA63339-0007-60C0-C436-083AC8230908
FreeBSD	启动分区	83BD6B9D-7F41-11DC-BE0B-001560B84F0F
	数据分区	516E7CB4-6ECF-11D6-8FF8-00022D09712B
	交换分区	516E7CB5-6ECF-11D6-8FF8-00022D09712B
	UFS 分区	516E7CB6-6ECF-11D6-8FF8-00022D09712B
	Vinum volume manager 分区	516E7CB8-6ECF-11D6-8FF8-00022D09712B
	ZFS 分区	516E7CBA-6ECF-11D6-8FF8-00022D09712B
Mac OS X	HFS(HFS+) 分区	48465300-0000-11AA-AA11-00306543ECAC
	苹果公司 UFS	55465300-0000-11AA-AA11-00306543ECAC
	ZFS	6A898CC3-1DD2-11B2-99A6-080020736631
	苹果 RAID 分区	52414944-0000-11AA-AA11-00306543ECAC
	苹果 RAID 分区，下线	52414944-5F4F-11AA-AA11-00306543ECAC
	苹果启动分区	426F6F74-0000-11AA-AA11-00306543ECAC
	Apple Label	4C616265-6C00-11AA-AA11-00306543ECAC
	Apple TV 恢复分区	5265636F-7665-11AA-AA11-00306543ECAC
Solaris	启动分区	6A82CB45-1DD2-11B2-99A6-080020736631
	根分区	6A85CF4D-1DD2-11B2-99A6-080020736631
	交换分区	6A87C46F-1DD2-11B2-99A6-080020736631
	备份分区	6A8B642B-1DD2-11B2-99A6-080020736631
	/usr 分区[10]	6A898CC3-1DD2-11B2-99A6-080020736631
	/var 分区	6A8EF2E9-1DD2-11B2-99A6-080020736631
	/home 分区	6A90BA39-1DD2-11B2-99A6-080020736631
	备用扇区	6A9283A5-1DD2-11B2-99A6-080020736631
		6A945A3B-1DD2-11B2-99A6-080020736631
	保留分区	6A9630D1-1DD2-11B2-99A6-080020736631
		6A980767-1DD2-11B2-99A6-080020736631
		6A96237F-1DD2-11B2-99A6-080020736631
		6A8D2AC7-1DD2-11B2-99A6-080020736631
NetBSD[11]	交换分区	49F48D32-B10E-11DC-B99B-0019D1879648
	FFS 分区	49F48D5A-B10E-11DC-B99B-0019D1879648
	LFS 分区	49F48D82-B10E-11DC-B99B-0019D1879648

[9] Linux 和 Windows 的数据分区使用相同的 GUID。

[10] Solaris 系统中/usr 分区的 GUID 在 Mac OS X 上被用作普通的 ZFS 分区。

[11] NetBSD 的 GUID 在单独定义之前曾经使用过 FreeBSD 的 GUID。

操作系统	分区类型 RAID 分区	GUID 49F48DAA-B10E-11DC-B99B-0019D1879648
	concatenated 分区	2DB519C4-B10F-11DC-B99B-0019D1879648
	加密分区	2DB519EC-B10F-11DC-B99B-0019D1879648

本表中的 GUID 使用小端序表示。例如，EFI 系统分区的 GUID 在这里写成C12A7328-F81F-11D2-BA4B-00A0C93EC93B，但是实际上它对应的 16 字节的串行是28 73 2A C1 1F F8 D2-11 BA 4B 00 A0 C9 3E C9 3B，即只有前 3 部分的字节序被交换了。

10.5 Bootloader

这时，计算机的控制权就要转交给硬盘的某个分区了，这里又分成三种情况。

1. 情况 A：卷引导记录
   前面提到，四个主分区里面，只有一个是激活的。计算机会读取激活分区的第一个扇区，叫做 “卷引导记录”（Volume boot record，缩写为 VBR）。VBR 的主要作用是告诉计算机，操作系统在这个分区里的位置。然后，计算机就会加载操作系统了。
2. 情况 B：扩展分区和逻辑分区
   随着硬盘越来越大，四个主分区已经不够了，需要更多的分区。但分区表只有四项，因此规定有且仅有一个区可以被定义成 “扩展分区”（Extended Partition）。所谓 “扩展分区”，就是指这个区里面又分成多个区，扩展分区里面的分区称为 “逻辑分区”（Logical Partition）。
   - 计算机先读取扩展分区的第一个扇区，叫做 “扩展引导记录”（Extended boot record，缩写为 EBR）。它里面也包含一张 64 字节的分区表，但是最多只有两项（也就是两个逻辑分区）。
   - 计算机接着读取第二个逻辑分区的第一个扇区，再从里面的分区表中找到第三个逻辑分区的位置，以此类推，直到某个逻辑分区的分区表只包含它自身为止（即只有一个分区项）。

   因此，扩展分区可以包含无数个逻辑分区。但是，似乎很少通过这种方式启动操作系统。如果操作系统确实安装在扩展分区，一般采用下一种方式启动。
3. 情况 C：启动管理器
   在这种情况下，计算机读取 “主引导记录” 前面 446 字节的机器码之后，不再把控制权转交给某一个分区，而是运行事先安装的 “启动加载器”（bootloader），由用户选择启动哪一个操作系统。

启动程序（Bootloader）也称启动加载器和引导程序，是指位于计算机或其他计算机应用上用于引导操作系统启动的程序。

Bootloader 启动方式及程序视应用机型种类而不同，例如在个人电脑上的引导程序通常分为两部分：第一阶段引导程序位于主引导记录（MBR），用以引导位于某个分区上的第二

阶段引导程序，如 NTLDR、GNU GRUB 等。

BIOS 开机完成后，Bootloader 就接手初始化硬件设备、创建存储器空间的映射，以便为操作系统内核准备好正确的软硬件环境，从而可以使操作系统准备好软件执行的环境来加载系统运行所需要的软件信息。

Bootloader 不依赖任何操作系统并且可以改变启动代码，从而可以指定使用哪个内核文件来引导操作系统启动，并实际加载内核到内存中解压缩与执行，此时内核就能够在内存中运行。接下来将会利用内核的功能再次检测所有硬件信息（相对于 BIOS 而言）和加载适当的驱动程序来使周边设备开始运行。也就是说，此时操作系统内核开始接管 BIOS 后面的工作了，这也就是 Bootloader 实现多系统引导的原理。

不过，随着计算机操作系统越来越复杂，不同操作系统的文件格式也不一致，每种操作系统都有自己相应的 Bootloader，因此位于主引导记录的空间已经放不下引导操作系统的代码，于是就有了第二阶段的引导程序，而 MBR 中代码的功能也从直接引导操作系统变为了引导第二阶段的引导程序。

其实，每个文件系统（filesystem 或 partition）都会保留一个引导扇区（boot sector）以提供操作系统安装 Bootloader。通常，操作系统默认都会将 Bootloader 安装到其根目录所在的文件系统的 boot sector 上，因此当安装了 Windows、Linux 多系统后，boot sector、bootloader 与 MBR 的相关性示意图如下：

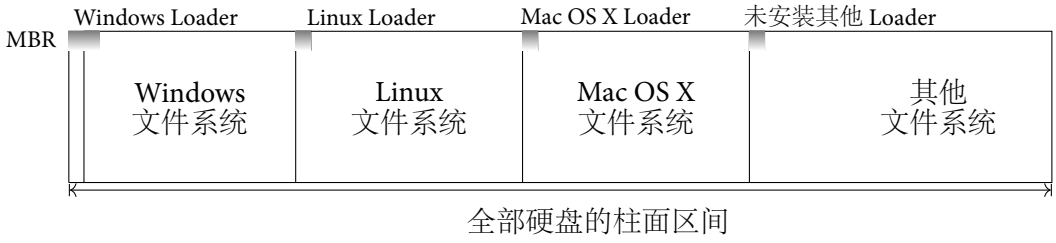


Figure 10.9: MBR、boot sector 与操作系统的关系

10.5.1 GRUB

目前 Linux 环境中最流行的启动引导程序是 GRUB（Grand Unified Bootloader），主要用于选择操作系统分区上的不同内核，也可用于向这些内核传递启动参数。

GRUB 非常轻便，而且支持多种可执行格式。除了可适用于支持多启动的操作系统外，还可以通过链式启动功能支持诸如 Windows 和 OS/2 之类的不支持多启动的操作系统。

GRUB 支持所有的 Unix 文件系统，也支持 Windows 适用的 FAT 和 NTFS 文件系统，还支持 LBA 模式。

GRUB 允许用户查看它支持的文件系统里文件的内容，这样就可以在启动时通过 GRUB 加载配置信息并进行修改（如选择不同的内核和 initrd）。为此目的，GRUB 提供了一个简单

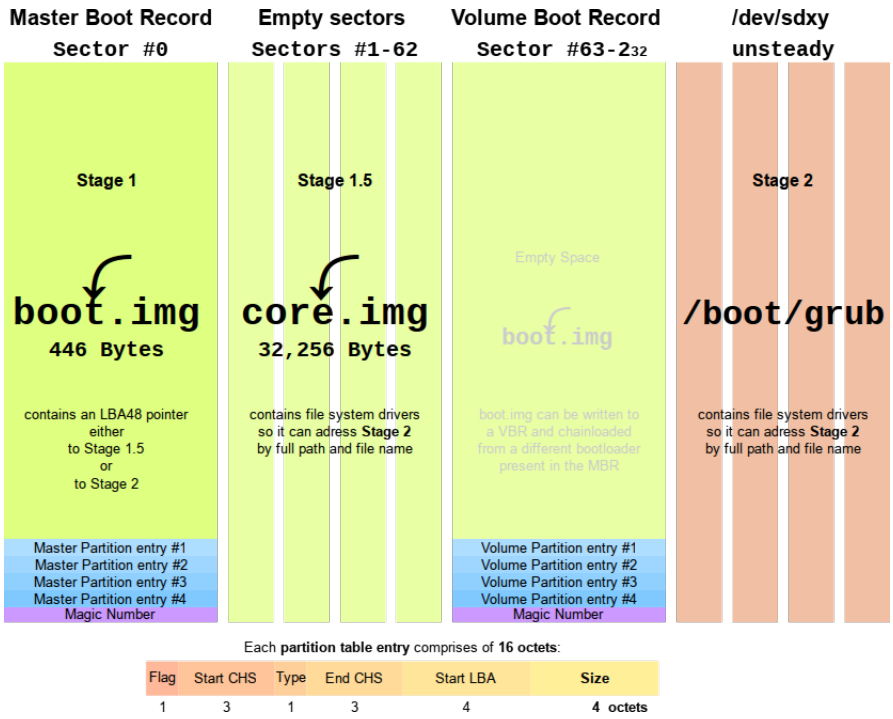


Figure 10.10: GRUB 位于 MBR 分区的示意图

的类似 Bash 的命令行界面，它允许用户编写新的启动顺序。

与其它启动器不同，GRUB 可以通过 GRUB 提示符直接与用户进行交互。载入操作系统前，在 GRUB 文本模式屏幕下键入 c 键可以进入 GRUB 命令行。在没有作业系统或者有作业系统而没有”menu.lst” 文件的系统上，同样可以进入 GRUB 提示符。

通过类似 bash 的命令，GRUB 提示符允许用户手工启动任何操作系统。把合适的命令记录在“menu.lst”文件里，可以自动启动一个操作系统。

GRUB 拥有丰富的终端命令，在命令行下使用这些命令，用户可以查看硬盘分区的细节，修改分区设置，临时重新映射硬盘顺序，从任何用户定义的配置文件启动，以及查看 GRUB 所支持的文件系统上的其它启动器的配置，因此即便不知道计算机上安装的操作系统，也可以从外部设备启动它。

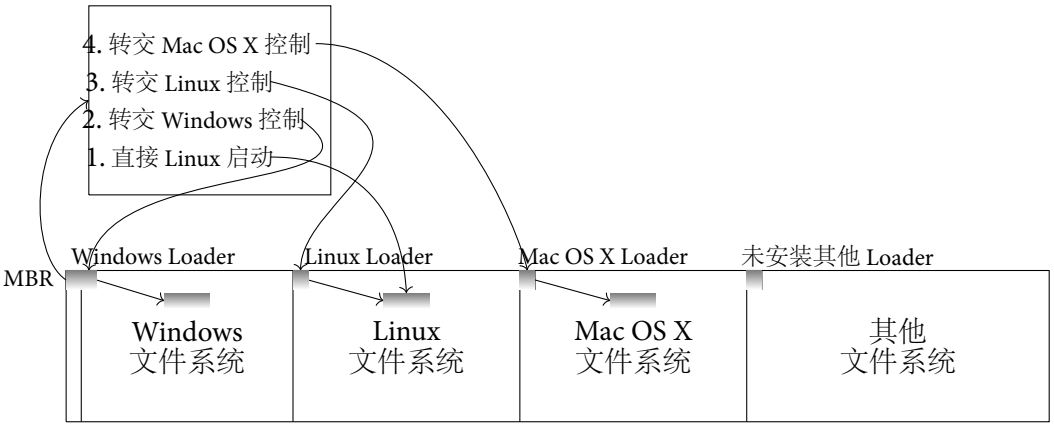
GRUB 具有多种用户界面，可以利用 GRUB 对图形界面的支持来提供定制的带有背景图案的启动菜单并支持鼠标。另外，通过对 GRUB 的文字界面的设定，可以通过串口实现远程终端启动。

- 提供选择菜单用以提示用户选择不同的启动选项，从而实现多重引导。
- 直接指向可启动的程序区段并执行以加载内核程序来引导操作系统。
- 将引导装载功能转交于其他 boot sector 内的 Boot Loader 来负责。

在 Linux 系统安装时，可以选择是否把 Bootloader 安装到 MBR，这样理论上 MBR 和

boot sector 都会保存一份 Bootloader 程序。

在 Windows 系统在安装时则默认主动将 MBR 和 boot sector 中都安装一份 Bootloader，因此在安装多重操作系统时，MBR 经常会被不同的操作系统的 boot sector 所覆盖。不过，Windows 的 Loader 默认不具有控制权转交的功能，因此也就不能使用 Windows 的 Loader 来加载 Linux 的 Loader。



引导装载程序除了可以安装在 MBR 之外，还可以在安装在每个分区的引导扇区（boot sector），从而可以实现“多重启动”功能。

通过链式启动，一个启动器可以启动另一个启动器，因此通过 GRUB 可以从 DOS、Windows、Linux、BSD 和 Solaris 等系统启动。

如果有多个 Kernel Images 安装在当前系统中时，就可以通过 GRUB 选择哪一个被执行，从而进入不同的内核所控制的系统。如果用户没有任何输入的话，将会加载 GRUB 配置文件中指定的默认 Kernel Image 。

选择了启动选项之后，GRUB 把选择的内核载入内存并把控制交给内核。

Windows 操作系统不支持多启动标准的操作系统，不过 GRUB 也可以通过链式启动把

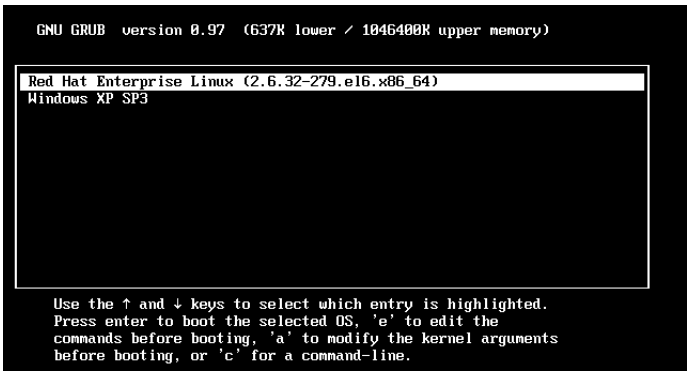


Figure 10.11: Grub 启动管理器

控制传给其它启动器，其它操作系统的启动程序被 GRUB 保存，其它操作系统如同直接自 MBR 启动。

类似 Windows 的启动菜单实际上是另一个启动管理器，它允许在多个不支持多启动的操作系统中做进一步的选择。例如，在已有 Windows 的系统或者包含多个 Windows 版本的系统上安装 Linux 而不修改原操作系统，即属于这类情况。

10.5.2 GRUB legacy

GRUB 的步骤 1 包含在 MBR 中。由于受 MBR 的大小限制，步骤一所做的几乎只是装载 GRUB 的下一步骤（存放在硬盘的其它位置）。步骤 1 既可以直接装载步骤 2，也可以装载步骤 1.5：GRUB 的步骤 1.5 包含在 MBR 后面的 30 千字节中。步骤 1.5 载入步骤 2。

当步骤 2 启动后，它将呈现一个界面来让用户选择启动的操作系统。这步通常采用的是图形菜单的形式，如果图形方式不可用或者用户需要更高级的控制，可以使用 GRUB 的命令行提示，通过它，用户可以手工指定启动参数。GRUB 还可以设置超时后自动从某一个内核启动。

10.5.3 GRUB 2

与 GRUB 第一版相似的是,boot.img 像步骤 1 一样在 MBR 或在启动分区中，但是，它可以从任何 LBA48 地址的一个扇区中读取，boot.img 将读取 core.img(产生于 diskboot.img）的第一个扇区以用来后面读取 core.img 的剩余部分。core.img 正常情况下跟步骤 1.5 储存在同一地方并且有着同样的问题，可是，当它被移动到一个文件系统或一个纯粹的分区时会比在步骤 1.5 移动或删除引起更少的麻烦。一旦完成读取，core.img 会读取默认的配置文件和其他需要的模块。

GRUB 的一个重要的特性是安装它不需依附一个操作系统；但是，这种安装需要一个 Linux 副本。由于单独工作，GRUB 实质上是一个微型系统，通过链式启动的方式，它可以启动所有安装的主流操作系统。

与 LILO 不同，修改 GRUB 的配置文件后，不必把 GRUB 重新安装到 MBR 或者某个分区中。

在 Linux 中，“grub-install”命令是用来把 GRUB 的步骤 1 安装到 MBR 或者分区中的。GRUB 的配置文件、步骤 2 以及其它文件必须安装到某个可用的分区中。如果这些文件或者分区不可用，步骤 1 将把用户留在命令行界面。

GRUB 配置文件的文件名和位置随系统的不同而不同，例如在 Debian(GRUB Legacy) 和 OpenSUSE 中，这个文件为/boot/grub/menu.lst，而在 Fedora 和 Gentoo 中为/boot/grub/grub.conf。Fedora、Gentoo Linux 和 Debian(GRUB 2) 使用/boot/grub/grub.conf[12]。

[12]Fedora 为了兼容文件系统层次结构标准提供了一个从/etc/grub.conf 到/boot/grub/grub.conf 的符号链接

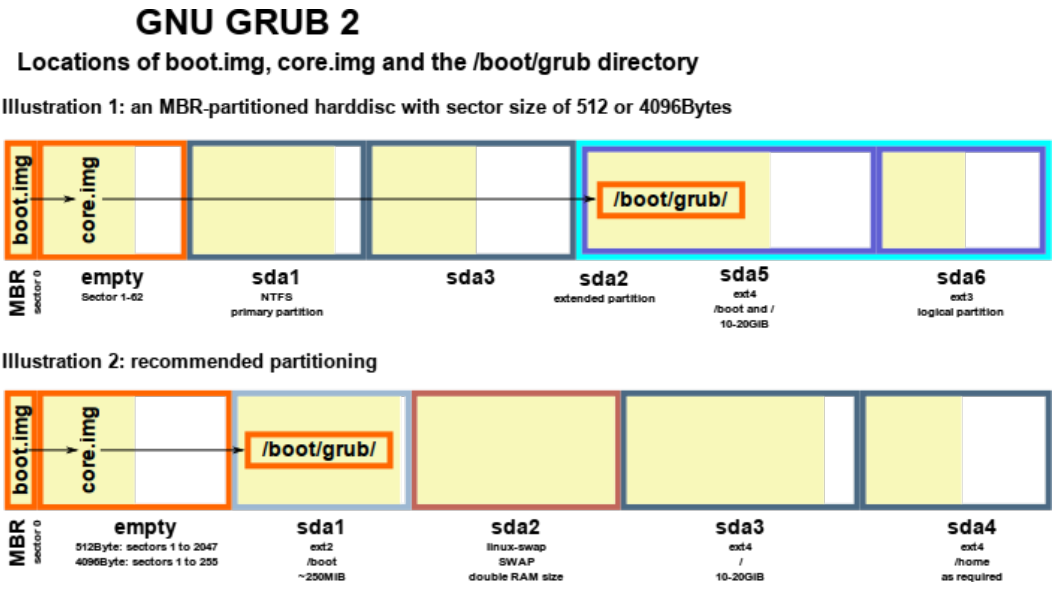


Figure 10.12: boot.img has the exact size of 446 Bytes and is written to the MBR (sector 0). core.img is written to the empty sectors between the MBR and the first partition, if available (for legacy reasons the first partition starts at sector 63 instead of sector 1, but this is not mandatory). The /boot/grub-directory can be located on an distinct partition, or on the /-partition.

除了硬盘外，GRUB 也可安装到光盘、软盘和闪存盘等移动介质中，这样就可以带起一台无法从硬盘启动的系统。

当 Grub 被载入内存执行时，它首先会去解析配置文件/boot/grub/grub.conf (/etc/grub.conf is a link to this)[13]，然后加载内核映像到内存中，并将控制权转交给内核。

```
#boot=/dev/sda
default=0
timeout=15
#splashimage=(hd0,0)/grub/splash.xpm.gz hiddenmenu
serial --unit=0 --speed=115200 --word=8 --parity=no --stop=1
terminal --timeout=10 serial console

title Red Hat Enterprise Linux Server (2.6.17-1.2519.4.21.el5xen)
   root (hd0,0)
   kernel /xen.gz-2.6.17-1.2519.4.21.el5 com1=115200,8n1
   module /vmlinuz-2.6.17-1.2519.4.21.el5xen ro
    root=/dev/VolGroup00/LogVol00
```

[13]也有的系统中，GRUB 的配置文件为/boot/grub2/grub.cfg(Fedora)。

```
module /initrd-2.6.17-1.2519.4.21.el5xen.img
```

也就是说挂载 grub.conf 中指定“root=”的根目录，并把控制权转交给操作系统[14]，而操作系统内核会立即初始化系统中各设备并做相关的配置工作，其中包括 CPU、I/O、存储设备等。

这里，关于 Linux 的设备驱动程序的加载，有一部分驱动程序是直接被编译进内核镜像中，另一部分驱动程序则是以模块的形式放在 initrd(ramdisk) 中。

Linux 内核需要适应多种不同的硬件架构，但是将所有的硬件驱动编入内核又是不实际的，因此实际上 Linux 的内核镜像仅是包含了基本的硬件驱动，在系统安装过程中会检测系统硬件信息，根据安装信息和系统硬件信息将一部分设备驱动写入 initrd 。这样在以后启动系统时，一部分设备驱动就放在 initrd 中来加载。

10.6 Init

操作系统的启动实际上是非常复杂的，内核首先要检测硬件并加载适当的驱动程序，还要调用程序来准备好系统运行的环境来让用户能够顺利的使用计算机。

Bootloader 的最终运行结果都是加载操作系统内核文件，因此当将控制权转交给操作系统后，操作系统的内核首先被载入内存。

以 Linux 系统为例，先载入/boot 目录下面的 kernel。内核加载成功后，第一个运行的程序是/sbin/init。它根据配置文件（Debian 系统是/etc/initab）产生 init 进程。这是 Linux 启动后的第一个进程，pid 进程编号为 1[15]，其他进程都是它的后代。

initrd 的英文含义是 bootloader initialized RAM disk，就是由 bootloader 初始化的内存盘。在 linu2.6 内核启动前，Bootloader 会将存储介质中的 initrd 文件加载到内存，内核启动时会在访问真正的根文件系统前先访问该内存中的 initrd 文件系统。grub 将 initrd 加载到内存里，让后将其中的内容释放到内容中，内核便去执行 initrd 中的 init 脚本，这时内核将控制权交给了 init 文件处理。

在 Bootloader 配置了 initrd 的情况下，内核启动被分成了两个阶段，第一阶段先执行 initrd 文件系统中的 init 来完成加载驱动模块等任务，第二阶段才会执行真正的根文件系统中的 /sbin/init 进程。

在内核完全启动起来，root 文件系统被挂载之前，initrd 被 kernel 当做临时 root 文件系统。当然 initrd 还包含了一些编译好的驱动，这些驱动用来在启动的时候访问硬件。

initramfs 是在 kernel 2.5 中引入的技术，实际上它的含义就是：在内核镜像中附加一个 cpio 包，这个 cpio 包中包含了一个小型的文件系统，当内核启动时，内核将这个 cpio 包解开，并且将其中包含的文件系统释放到 rootfs 中，内核中的一部分初始化代码会放到这个文

[14]在个人电脑中，Linux 的启动是从 0xFFFF0 地址开始的。

[15]因为/sbin/init 是 LINUX kernel 执行的第一个程序，所以/sbin/init 的 PID 为 1。

件系统中，作为用户层进程来执行。这样带来的明显的好处是精简了内核的初始化代码，而且使得内核的初始化过程更容易定制。

init 脚本的内容主要是加载各种存储介质相关的设备驱动程序。当所需的驱动程序加载完后，会创建一个根设备，然后将根文件系统 rootfs 以只读的方式挂载。这一步结束后，释放未使用的内存，转换到真正的根文件系统上面去，同时运行/sbin/init 程序，执行系统的 1 号进程。此后系统的控制权就全权交给/sbin/init 进程了。

/sbin/init 进程是系统其他所有进程的父进程，当它接管了系统的控制权之后，在加载各项系统服务之前，它首先会去读取/etc/inittab 配置文件来执行相应的脚本进行系统初始化，例如设置键盘、字体，装载模块和设置网络等。

具体而言，/sbin/init 进程会从/etc/inittab 文件中得到默认启动级别，加载系统的各个模块并执行相应级别的程序（比如窗口程序和网络程序），直至执行/bin/login 程序，跳出登录界面，等待用户输入用户名和密码。

当 Linux 系统启动完成后，许多的服务进程也启动了，这些服务程序都放在相应 Linux 系统启动级别的目录中。根据 Linux 的默认启动级别，系统将会执行以下其中一个目录中的服务程序:

- Run level 0 – /etc/rc.d/rc0.d/
- Run level 1 – /etc/rc.d/rc1.d/
- Run level 2 – /etc/rc.d/rc2.d/
- Run level 3 – /etc/rc.d/rc3.d/
- Run level 4 – /etc/rc.d/rc4.d/
- Run level 5 – /etc/rc.d/rc5.d/
- Run level 6 – /etc/rc.d/rc6.d/

注意，在/etc 下面的那些文件有些是连接文件，例如/etc/rc0.d 连接到/etc/rc.d/rc0.d[5]。

Chapter 11

Boot

在 BIOS 阶段，计算机的行为基本上被写死了，程序员可以做的事情并不多。

但是，一旦进入操作系统，程序员几乎可以定制所有方面，在下图中显示了典型 Linux 计算机系统启动的 6 个主要阶段[6]。

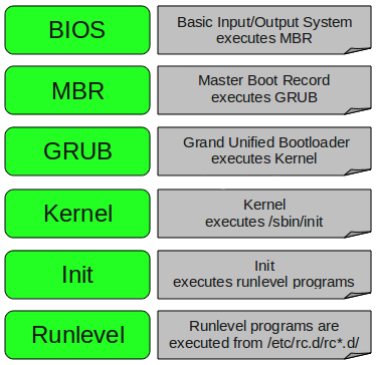


Figure 11.1: 通电->BIOS->MBR->GRUB->Kernel->/sbin/init->runlevel

这里，针对 Linux 来探讨操作系统接管硬件以后发生的事情，也就是 Linux 操作系统的启动流程[7]。

11.1 vmlinuz

内核文件一般放置于/boot 下并被命名为/boot/vmlinuz，因此操作系统接管硬件以后，会首先读入/boot 目录下的内核文件。

一般来说，当创建一个可启动的核心时，此核心会先经过 zlib 算法压缩，而在核心内会包含一个相当小的解压缩程序 stub，当 stub 解压缩核心程序的时候会对 console 视窗印出“点”来表示解压缩进度，而解压缩所花费的时间在开机时间中所占程度来说其实是相当小

的，早期的 bzImage 的发展中对于核心的大小会有所限制（特别是 i386 架构），在此情况下压缩则是必须的。

vmlinuz[8] 是 vmlinux 经过 gzip 和 objcopy 制作出来的压缩文件。vmlinuz 作为一种统称，有两种具体的表现形式 zImage 和 bzImage[1]。bzimage 和 zImage 的区别在于本身的大小，以及加载到内存时的地址不同，其中 zImage 在 0 ～ 640KB，而 bzImage 则在 1M 以上的位置。

/boot 目录下面大概是这样一些文件：

```
$ ls /boot
%对应的内核被编译时选择的功能与模块配置文件
config-3.2.0-3-amd64
config-3.2.0-4-amd64
efi
%引导装载程序GRUB相关的文件目录
grub2
%虚拟文件系统文件
initrd.img-3.2.0-3-amd64
initrd.img-3.2.0-4-amd64
%内核功能放置到内存地址的对应表
System.map-3.2.0-3-amd64
System.map-3.2.0-4-amd64
%对应的内核文件
vmlinuz-3.2.0-3-amd64
vmlinuz-3.2.0-4-amd64
```

Linux 内核可以通过动态加载内核模块，这些内核模块存储在/lib/modules 目录中。

由于内核模块位于根目录下，因此/lib 不能与/放置在不同的分区中，而且在启动的过程中内核必须要挂载根目录才能读取内核模块。为了避免影响到系统文件，在启动过程中根

[1]随着 Linux kernel 的成长，核心的内容日益增加超越了原本的限制大小。bzImage (big zImage) 格式则为了克服此缺点开始发展，利用将核心切割成不连续的存储器区块来克服大小限制。
bzImage 格式仍然是以 zlib 算法来做压缩，虽然有一些广泛的误解就是因为以 bz- 为开头，而让人误以为是使用 bzip2 压缩方式（bzip2 包所带的工具程序通常是以 bz- 为开头的，例如 bzless, bzcat ...）。
bzImage 文件是一个特殊的格式，包含了 bootsect.o + setup.o + misc.o + piggy.o 串接，其中 piggy.o 包含了一个 gzip 格式的 vmlinux 文件（可以参看 arch/i386/boot/下的 compressed/Makefile piggy.o）。

目录/是以只读的方式挂载的。

一般来说，非必要的且可以编译为模块的内核功能，Linux 都会将它们编译成模块，因此 USB、SATA、SCSI 等硬盘设备的驱动程序通常都被编译成模块的形式。

initrd（Boot Loader Initialized RAM Disk）就是由 Bootloader 初始化的内存盘。在 linux 2.6 内核启动前，Bootloader 会将存储介质中的/boot/initrd 文件加载到内存，内核启动时会在访问真正的根文件系统前先访问该内存中的 initrd 文件系统。GRUB 将 initrd 加载到内存里，让后将其中的内容释放到内容中并模拟成一个根目录，内核便去执行 initrd 中的 init 脚本，这时内核将控制权交给了 init 进程。

initrd 实质上是通过 cpio 命令生成的文件，其组成结构类似于：

```
lrwxrwxrwx  1 root root        7 Apr 12 13:08 bin -> usr/bin
drwxr-xr-x  2 root root     4096 Apr 12 13:08 dev
drwxr-xr-x 13 root root     4096 Apr 12 13:08 etc
lrwxrwxrwx  1 root root       23 Apr 12 13:08 init ->
    usr/lib/systemd/systemd
lrwxrwxrwx  1 root root        7 Apr 12 13:08 lib -> usr/lib
lrwxrwxrwx  1 root root        9 Apr 12 13:08 lib64 -> usr/lib64
drwxr-xr-x  2 root root     4096 Apr 12 13:08 proc
drwxr-xr-x  2 root root     4096 Apr 12 13:08 root
drwxr-xr-x  2 root root     4096 Apr 12 13:08 run
lrwxrwxrwx  1 root root        8 Apr 12 13:08 sbin -> usr/sbin
-rwxr-xr-x  1 root root     3041 Apr 12 13:08 shutdown
drwxr-xr-x  2 root root     4096 Apr 12 13:08 sys
drwxr-xr-x  2 root root     4096 Apr 12 13:08 sysroot
drwxr-xr-x  2 root root     4096 Apr 12 13:08 tmp
drwxr-xr-x  7 root root     4096 Apr 12 13:08 usr
drwxr-xr-x  3 root root     4096 Apr 12 13:08 var
```

通过上述执行文件的内容可以知道 initrd 加载了相关模块并尝试挂载了虚拟文件系统，因此在内核完全启动起来，root 文件系统被挂载之前，initrd 都被 kernel 当做临时 root 文件系统。当然 initrd 还包含了一些编译好的驱动，这些驱动用来在启动的时候访问硬件。

在 Bootloader 配置了 initrd 的情况下，内核启动被分成了两个阶段，第一阶段先执行 initrd 文件系统中的 init 来完成加载启动过程中需要的内核模块等任务。在第一阶段结束时将完成加载 USB、RAID、LVM 和 SCSI 等文件系统与硬盘接口的驱动程序。接下来，在第二阶段在真正的根文件系统中内核将会调用/sbin/init 程序来开始后续的启动流程。

Bootloader 可以加载 kernel 与 initrd，然后在内存中将 initrd 解压缩并模拟成根目录。该模拟根目录在内存中的文件系统能够提供一个可执行的程序来加载启动过程中所需要的内核模块，接下来 Kernel 就能够借此加载适当的驱动程序（通常是 USB、RAID、LVM 和 SCSI

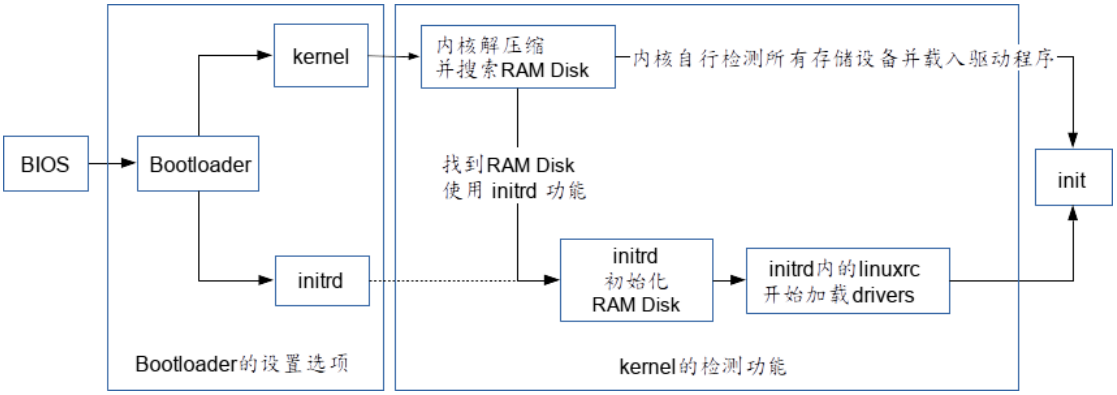


Figure 11.2: BIOS 与 Bootloader 以及内核加载流程示意图

等文件系统与硬盘接口的驱动程序)，最终释放虚拟文件系统并挂载实际的根目录文件系统，从而能够帮助内核重新调用/sbin/init 来开始后续的正常启动流程。

需要注意的是，initrd 只是应用在无法挂载根目录的情况下，如果 Linux 安装在 IDE 接口的硬盘上，并且使用默认的 Ext2/Ext3 文件系统，那么不需要 initrd 也能够顺利的启动 Linux 操作系统的。

init ramfs 是在 kernel 2.5 中引入的技术，实际上它的含义就是：在内核镜像中附加一个 cpio 包，这个 cpio 包中包含了一个小型的文件系统，当内核启动时，内核将这个 cpio 包解开，并且将其中包含的文件系统释放到 rootfs 中，内核中的一部分初始化代码会放到这个文件系统中，作为用户层进程来执行。这样带来的明显的好处是精简了内核的初始化代码，而且使得内核的初始化过程更容易定制。

11.2 init

内核文件完整加载以后，就可以开始运行第一个程序：/sbin/init。

/sbin/init 是 Unix 和类 Unix 系统中用来产生其它所有进程的程序。作为第一个运行的程序，其进程编号（pid）就是 1，其他所有进程都从它衍生，都是它的子进程。

/sbin/init 以守护进程的方式存在，其作用是系统初始化，具体来说就是指从 init 进程成功启动，一直到系统启动并弹出登录提示之间的过程，所有这些操作都会通过 init 的配置文件——/etc/inittab 来规划，具体包括系统的主机名、网络设置、语系设置、文件系统格式及其他服务的启动等。另外，initab 还确定了操作系统默认的 runlevel（启动执行等级）。

UNIX（例如 System III 和 System V）系列中 init 的作用和 UNI-like 和 BSD 衍生版本相比，发生了一些变化。

具体来说，BSD init 运行存放于 '/etc/rc' 的初始化 shell 脚本，然后启动基于文本模式的

终端 (getty) 或者基于图形界面的终端 (窗口系统)。这里没有运行模式的问题，因为文件 'rc' 决定了 init 如何执行。这种做法的优点是简单且易于手动编辑，但如果第三方软件需要在启动过程执行它自身的初始化脚本，它必须修改已经存在的启动脚本，一旦这种过程中有一个小错误，都将导致系统无法正常启动。

现代的 BSD 衍生系统一直支持使用 'rc.local' 文件的方式，它将在正常启动过程接近最后的时间以子脚本的方式来执行。这样做减少了整个系统无法启动的风险。然后，第三方软件包可以将它们独立的 start/stop 脚本安装到一个本地的 'rc.d' 目录中 (通常这是由 ports collection/pkgsrc 完成的)。与 SysV 类似，FreeBSD 和 NetBSD 现在默认使用 rc.d，该目录中所有的用户启动脚本，都被分成更小的子脚本。rcorder 通常根据在 rc.d 目录中脚本之间的依赖关系来决定脚本的执行顺序。

- Apple Mac OS X 使用 SystemStarter, 用来替代 launchd;
- Ubuntu 使用 Upstart 来完全代替 init;
- Fedora 使用 systemd 来完全替代 init，并可并行启动服务来减少在 shell 上的系统开销。

作为 Linux 下的一种 init 软件，systemd 的开发目标是提供更优秀的框架以表示系统服务间的依赖关系，并依此实现系统初始化时服务的并行启动，同时达到降低 Shell 的系统开销的效果，最终代替现在常用的 System V 与 BSD 风格 init 程序。

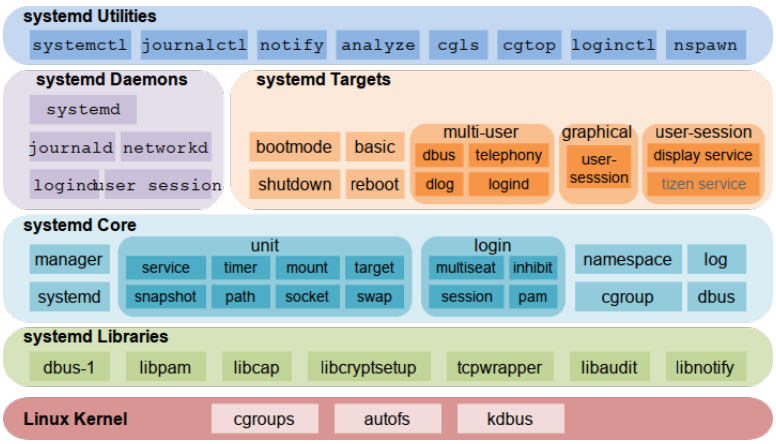


Figure 11.3: systemd componentos

与多数发行版使用的 System V 风格 init 相比，systemd 采用了以下新技术:

- 采用 Socket 激活式与总线激活式服务，以提高相互依赖的各服务的并行运行性能;
- 用 cgroups 代替 PID 来追踪进程，以此即使是两次 fork 之后生成的守护进程也不会脱

离 systemd 的控制。

通常在 /etc/inittab 文件中定义了各种运行模式的工作范围，例如 System V init 会检查 '/etc/inittab' 文件中是否含有 'initdefault' 项，然后通知 init 系统是否有一个默认运行模式。如果没有默认的运行模式，那么用户将进入系统控制台来手动决定进入何种运行模式。System V 中运行模式描述了系统各种可能的状态，通常会有 8 种运行模式，即运行模式 0 到 6 和 S 或者 s，其中运行模式 3 为 "保留的" 运行模式。

- 0. 关机
- 1. 单用户模式
- 6. 重启

除了模式 0、1 和 6 之外，每种 Unix 和 Unix-like 系统对运行模式的定义不太一样，例如通常模式 5 是多用户图形环境 (X Window System)，通常还包括 X 显示管理器，然而在 Solaris 操作系统中，模式 5 被保留用来执行关机和自动切断电源。

大多数 Linux 发行版是和 System V[2]相兼容的，但是一些发行版如 Arch 和 Slackware 采用的是 BSD 风格，其它的如 Gentoo 是自己定制的。Ubuntu 和其它一些发行版现在开始采用 Upstart 来代替传统的 init 进程。

当 init 进程成功启动后，它会根据配置文件/etc/inittab[9] 中的设置初始化系统，这个过程主要完成的工作包括重新挂载文件系统、运行系统需要的进程和服务等。

```
[root@localhost ~]#cat /etc/inittab
#
#inittab
#
#This file describes how the INIT process should set up the system in a
    certain run-level.
#
#Author: Miquel van Smoorenburg, <miquels@drinkel.nl.mugnet.org>
#Modified for RHS Linux by Marc Ewing and Donnie Barnes
#

#Default runlevel. The runlevels used by RHS 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
```

[2]从 System V 开始，pidof 或者 killall5 被用在很多发行版中。

```
#6 - reboot (Do NOT set initdefault to this)
#
#设置系统默认的运行级别
id:5:initdefault:

#初始化系统脚本
#System initialization.
si::sysinit:/etc/rc.d/rc.sysinit

#启动系统服务以及需要启动的服务的 script 的放置路径
l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6

#定义 Ctrl+Alt+Delete 键的作用
#Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now

#设置电源选项
#电源失效时的设置
#When our UPS tells us power has failed, assume we have a few minutes
#of power left. Schedule a shutdown for 2 minutes from now.
#This does, of course, assume you have powerd installed and your
#UPS connected and working correctly.
pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting
    Down"

#电源恢复时的设置
#If power was restored before the shutdown kicked in, cancel it.
pr:12345:powerokwait:/sbin/shutdown -c "Power Restored; Shutdown Cancelled"


#启动终端
#Run gettys in standard runlevels
```

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

#Run xdm in runlevel 5
x:5:respawn:/etc/X11/prefdm -nodaemon
```

事实上，/etc/inittab 的设置类似于 shell script，上面的有效行都是按顺序从上往下处理的，而且它们都有一个共同的格式，就是使用冒号“:”将设置字段分隔为 4 个不同的字段。

/etc/inittab 中每一行的格式如下:

```
#[设置选项]:[runlevel]:[init 的操作行为]:[命令选项]
id:runlevels:action:process
```

- id：配置行在配置文件中的标识符，最长可以由 4 个字符组成，代表 init 的主要工作选项。
  对于大多数文本行来说，这个字段没有太大的意义，但要求每行的 id 值在该文件中是唯一的。
- runlevels：配置行起作用的运行级别列表。如果作用于多个运行级别，可以将其写在一起，例如 35 则代表 runlevel 3/5 都会执行。
- action：配置 init 应该执行的动作。通常有 initdefault、sysinit、wait、ctrlaltdel、powerfail、powerokwait 和 respawn 这几个动作。
- process：表示配置行要执行的脚本、命令及参数等内容。

inittab 设置值	意义
initdefault	代表默认的 runlevel 设置值;
sysinit	代表系统初始化的操作选项;
ctrlaltdel	代表 [ctrl]+[alt]+[del] 三个按键是否可以重新启动的设置;
wait	代表后面字段设置的命令项目必须要执行完毕才能继续后面其他的操作;
respawn	代表后面字段的命令可以无限制的再生 (重新启动)。例如，tty1 的 mingetty 产生的可登录界面，在用户注销结束后，系统可以再生成一个一个新的可登录界面等待下一个登录。

由上述可知，根据当前/etc/inittab 的设置，init 的处理流程如下:

1. 取得 runlevel 即默认执行等级；
2. 使用/etc/rc.d/rc.sysinit 进行系统初始化；
3. 执行`5:5:wait:/etc/rc.d/rc5` 中的脚本，其他略过；
4. 设置 [ctrl]+[alt]+[del] 三个按键是否可以重新启动；
5. 设置电源选项中 pf 和 pr 机制；
6. 启动 mingetty 的 6 个终端机（tty1 ～ tty6）；
7. 以`/etc/X11/perfdm -nodaemon` 启动图形界面。

11.3 initdefault

许多程序需要开机启动。它们在 Windows 叫做“服务”（service），在 Linux 中叫做“守护进程”（daemon）。Linux 通过设置 runlevel 来规定系统使用不同的服务来启动，从而进入不同的使用环境，基本上根据有无网络与有无 X Window 而将 runlevel 分为 7 个等级。

具体而言，Linux 通过/etc/inittab 配置文件来决定 Linux 的运行等级（runlevel），包括：

- 0 – halt（关机）
- 1 – Single user mode （单用户维护模式）
- 2 – Multiuser, without NFS（多用户模式，无 NFS 服务）
- 3 – Full multiuser mode（包含完整网络功能的纯文本多用户模式）
- 4 – unused（用户自定义）
- 5 – X11（多用户模式，有网络，有图形界面）
- 6 – reboot （重新启动）

init 进程的一大任务，就是去运行这些开机启动的程序。但是，不同的场合需要启动不同的程序，比如用作服务器时，需要启动 Apache，用作桌面就不需要。Linux 允许为不同的场合，分配不同的开机启动程序，这就叫做”运行级别”（runlevel）。也就是说，启动时根据“运行级别”，确定要运行哪些程序。

Linux 预置七种运行级别（0-6）。一般来说，0 是关机，1 是单用户模式（也就是维护模式），6 是重启。运行级别 2-5，各个发行版不太一样，对于 Debian 来说，都是同样的多用户模式（也就是正常模式）。

init 进程首先读取文件 /etc/inittab，它是运行级别的设置文件。

/etc/inittab 文件中设置的默认的运行模式在:initdefault: 项中，initdefault 用于设置系统启动时的默认运行级别，即系统启动后将自动进入到运行级别。这里，initdefault 的值是 5，表明系统启动时的运行级别为 5。

```
id:5:initdefault:
```

如果需要指定其他级别，可以手动修改 initdefault 值，但是同一时间内只能使用一个运行级别。在 root 权限下，运行 telinit 或者 init 命令可以改变当前的运行模式。

大多数操作系统的用户可以用下面的命令来判断当前的运行模式是什么：

```
$ runlevel
N    5
$ who -r
      run-level 5  2014-04-12 12:16
```

修改系统默认的运行级别时，注意不要将字段设置为 0 和 6，否则系统将无法正常开机。如果黑客将此字段修改为 0 和 6，通常我们将其视为拒绝服务式攻击（Denial of Service，简称 DoS）。

那么，运行级别 5 有些什么程序呢，系统怎么知道每个级别应该加载哪些程序呢？..... 回答是每个运行级别在/etc 目录下面，都有一个对应的子目录，指定要加载的程序。

```
/etc/rc0.d
/etc/rc1.d
/etc/rc2.d
/etc/rc3.d
/etc/rc4.d
/etc/rc5.d
/etc/rc6.d
```

上面目录名中的 “rc”，表示 run command（运行程序），最后的 d 表示 directory（目录）[3]。下面让我们看看 /etc/rc5.d 目录中到底指定了哪些程序。

```
$ ls  /etc/rc5.d

README
K50netconsole
K90network
S00livesys
S95jexec
S99livesys-late
...
```

[3].d 结尾是代表与 SysV init 相关的配置文件[10]，init 是有很多种实现的，上一代的是 SysV init，现在是两种——即 Ubuntu 所使用的 Upstart 实现以及主流的 systemd 实现，两种新的实现都兼容老的 SysV init 配置文件，但实际上几乎没有什么主流 distro 继续在用 SysV init 了，都是通过兼容实现的，所以有必要看一下 systemd 和 upstart 的手册和文档（runlevel 这个概念其实也是老的 SysV init 里面的）

可以看到，除了第一个文件 README 以外，其他文件名都是” 字母 S+ 两位数字 + 程序名” 的形式。字母 S 表示 Start，也就是启动的意思（启动脚本的运行参数为 start），如果这个位置是字母 K，就代表 Kill（关闭），即如果从其他运行级别切换过来，需要关闭的程序 (启动脚本的运行参数为 stop)。后面的两位数字表示处理顺序，数字越小越早执行。数字相同时，则按照程序名的字母顺序启动。

这个目录里的所有文件（除了 README），就是启动时要加载的程序。如果想增加或删除某些程序，不建议手动修改 /etc/rcN.d 目录，最好是用一些专门命令进行管理。

Linux 系统中，现代的 bootloader（包括 LILO 或 GRUB）允许用户在初始化过程中以最后启动的进程来取代默认的 /sbin/init。通常是在 bootloader 环境中通过执行 init=/foo/bar 命令。例如，如果执行 init=/bin/bash，启动单用户 root 的 shell 环境，无需用户密码。

BSD 的大多数变种可以设置 bootstrap 程序被中断后执行 `boot -s` 命令进入单用户模式。

当系统发现启动过程中报错时，比如不正常关机造成的文件系统的不一致的情况时，操作系统会主动进入单用户维护模式。

单用户模式并没有跳过 init，它仍然可以执行 /sbin/init，但是它将使 init 询问 exec() 将要执行的命令 (默认为 /bin/sh) 的路径，而不是采用正常的多用户启动顺序。如果内核启动时在 /etc/ttys 文件中被标注为 “不安全” (在某些系统中，当前的 “安全模式” 可能会有些变化)，在允许这种情况 (或者回退到单用户模式，如果用户执行 CTRL+D)，init 将首先询问 root 用户的密码。如果该程序退出，内核将在多用户模式下重新执行 init。如果系统从多用户模式切换到单用户模式，还将碰到上述的情况。

如果内核加载后，init 不能被正常启动将导致 panic 错误，此时系统将不可使用。想要通过 init 自身来改变 init 的路径，不同的版本情况不太一样。

- NetBSD 中可执行 `boot -a`;
- FreeBSD 中利用 init_path 命令装载变量。

11.4 rc.sysinit

/sbin/init 进程执行系统初始化脚本 (/etc/rc.d/rc.sysinit) 来对系统进行基本的配置（例如网络、时区等），并以读写方式挂载根文件系统（/）及其它文件系统。也就是说，/etc/rc.d/rc.sysinit 的作用就是在加载各项系统服务之前，为操作系统设置好系统环境。

```
#初始化系统脚本
#System initialization.
si::sysinit:/etc/rc.d/rc.sysinit
```

上面这行中的 runlevels 字段为空，表示 init 将会忽略这一字段，并在每个运行级别都执行系统初始化脚本文件/etc/rc.d/rc.sysinit。

系统初始化脚本文件/etc/rc.d/rc.sysinit[4]是一个可执行文件，它所做的事情如下：

1. 获取网络环境与主机类型。
   首先会读取网络环境设置文件”/etc/sysconfig/network”，获取主机名称与默认网关（gateway）等网络环境。
2. 测试与载入内存设备/proc 及 USB 设备/sys。
   除了挂载/proc 外，系统会主动检测是否有 USB 设备，并主动加载 USB 驱动，尝试载入 USB 文件系统。
3. 决定是否启动 SELinux。
   SELinux 检测是否需要启动，并检测是否需要为所有文件重新编写标准的 SELinux 类型（autorelabel）。
4. 接口设备的检测与即插即用（PnP）参数的测试。
   根据内核在启动时检测的结果（/proc/sys/kernel/modprobe）开始进行 IDE/SCSI/Network/Audio 等接口设备的检测，以及利用已加载的内核模块进行 PnP 设备的参数测试。
5. 用户自定义模块的加载。用户可以在”/etc/sysconfig/modules/*.modules”加入自定义的模块，此时会加载到系统中。
6. 加载内核的相关设置。系统会主动去读取”/etc/sysctl.conf”这个文件的设置值并配置内核相关功能。
7. 设置主机名与初始化电源管理模块（ACHI）
8. 设置系统时间（clock）与时区设置，并载入/etc/sysconfig/clock 设置。
9. 设置终端的控制台的字体。
10. 设置显示于启动过程中的欢迎画面。
11. 初始化硬盘阵列 RAID 及 LVM 等硬盘功能，主要是通过/etc/mdadm.conf 来设置。
12. 以 fsck 方式查看并检验硬盘文件系统。
13. 进行硬盘配额 Quota 的转换（非必要）。
14. 重新以可读写模式挂载系统硬盘。
15. 启动 Quota 功能。
16. 启动系统伪随机数生成器（pseudo-random）[5]。
    随机数生成器可以产生随机数，并用于系统进行密码加密演算，在此需要启动两次随机数生成器。

[4]根据不同的 Linux 发行版，/etc/rc.d/rc.sysinit 可能有些差异。例如，在 SUSE Server 9 中使用/etc/init.d/boot 与/etc/init.d/rc 来进行系统初始化的。

[5]Theodore Y. Ts'o（曹子德）于 1994 年在 Linux 内核中实现了/dev/random 以及对应的核心驱动程序，让 Linux 成为所有操作系统中第一个实现了以系统背景噪音产生的真正随机数生成器。/dev/random 可以不用依靠硬件随机乱数产生器来独立运作，提升效能的同时也节省了成本。其他的守护行程（例如 rngd）可以从硬件取得随机数，提供给/dev/random，应用程序可以经由/dev/random 取得随机数。在/dev/random 与/dev/urandom 实现出来之后，很快就成为在 Unix、Linux、BSD 与 Mac OS 的共通标准接口。

17. 清除启动过程中的临时文件。
18. 将启动相关信息加载到”/var/log/dmesg”文件中。

    通过查看/var/log/dmesg 可以了解启动过程中的各个步骤，执行：

```
dmesg
```

可以列出所有与启动过程有关的信息。

当/etc/rc.d/rc.sysinit 执行结束后，系统的基本设置就完成了，此时系统就可以顺利工作了。

基本上，在/sbin/init 执行过程中的很多工作的默认设置文件其实都在/etc/sysconfig/目录中。另外，用户还可以在系统初始化过程中加载自定义模块或驱动程序。

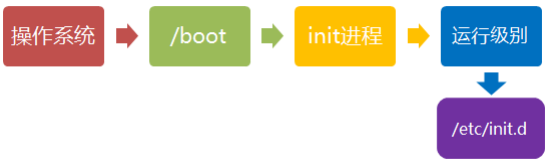
如果想要加载自定义内核模块，可以将整个模块写入到/etc/sysconfig/modules/*.modules 中，此时就需要系统自定义的设备与模块的对应文件（/etc/modprobe.conf)。

一般来说，/etc/modprobe.conf 大多用于指定系统内的硬件所需要的模块，通常它可以由系统来自行配置。只有当系统检测到错误的驱动程序或需要使用更新的驱动程序时来需要用户手动配置。

下一步需要启动系统相关的服务与网络服务等，这样主机才可以提供相关的网络和主机功能，因此便会执行接下来的脚本。

系统启动过程中必须要从/etc/sysconfig 目录中读取的相关配置文件包括：

- authconfig
  设置用户的身份认证的机制，包括是否使用本机的/etc/passwd、/etc/shadow 等，/etc/shadow 密码记录使用的加密算法，以及是否使用外部密码服务器（NIS、LDAP 等）提供的认证功能等。
  系统默认使用 MD5 加密算法，并且不使用外部的身份认证机制。
- clock
  设置主机的时区，可以使用格林威治时间（GMT)，也可以使用本地时间（local)。基本上，在 clock 文件内的设置选项“ZONE”所参考的时区位于/usr/share/zoneinfo 目录下的相对路径中。如果要修改时区，需要把/etc/localtime 链接到/usr/share/zoneinfo 目录下相应的时区文件。
- i18n
  设置主机的语系。例如，为了在命令行下正确的显示日期，需要修改该文件中的 LC_TIME。
- keyboard
  设置主机的键盘。
- mouse
  设置主机的鼠标。
- network



设置主机的网络选项，包括是否要启动网络、设置主机名和网关等。

- network-scripts
  设置主机的网卡选项。

11.5 rc.d

Linux 预设的“运行级别”各自有一个目录，存放需要开机启动的程序。Linux 将/etc/rcN.d 目录里列出的程序都设为链接文件，并统一指向另外一个目录 /etc/init.d ，/etc/init.d 这个目录名最后一个字母 d，是 directory 的意思，表示这是一个目录，用来与程序 /etc/init 区分。

真正的启动脚本都统一放在/etc/init.d 目录中，init 进程逐一加载开机启动程序，其实就是运行这个目录里的启动脚本。

/etc/rc.d/rc.sysinit 通过分析/etc/inittab 文件来确定系统的启动级别，然后执行相应的/etc/rc.d/rc*.d 下的脚本来启动相应的服务。

```
#设置系统默认的运行级别
id:5:initdefault:

#初始化系统脚本
#System initialization.
si::sysinit:/etc/rc.d/rc.sysinit

#启动系统服务
l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6
```

在确定了系统的运行级别后，接下来通过对应的/etc/rc.d/rc 脚本的执行如下：

- 通过外部第 1 号参数（$1）来取得要执行的脚本目录，这里得到的是/etc/rc.d/rc5.d 目

录来准备处理相关的脚本程序;

- 定义相关服务启动的顺序是先 K 后 S，而具体的每个运行级别的服务状态是放在/etc/rc.d/rc*.d（*=0 ～ 6）目录下;
- 所有的文件均是指向/etc/init.d 下相应文件的符号链接。

```
/etc/init.d—> /etc/rc.d/init.d
/etc/rc —>/etc/rc.d/rc
/etc/rc*.d —>/etc/rc.d/rc*.d
/etc/rc.local—> /etc/rc.d/rc.local
/etc/rc.sysinit—> /etc/rc.d/rc.sysinit
```

/etc 目录下的 init.d、rc、rc*.d、rc.local 和 rc.sysinit 均是指向/etc/rc.d 目录下相应文件和文件夹的符号链接。这样做的另一个好处，就是如果要手动关闭或重启某个进程，直接到目录 /etc/init.d 中寻找启动脚本即可。比如要重启 Apache 服务器，就运行下面的命令:

```
$ sudo /etc/init.d/apache2 restart
```

这里以启动级别 5 为例来简要说明。/etc/rc.d/rc5.d 目录下的内容全部都是以 S 或 K 开头的链接文件，都链接到”/etc/rc.d/init.d”目录下的各种 shell 脚本，其中:

- S 表示的是启动时需要 start 的服务内容。
  对于/etc/rc5.d/S??.* 开头的文件，执行`/etc/rc5.d/K??.* stop` 的操作。
- K 表示关机时需要关闭的服务内容。
  对于/etc/rc5.d/K??.* 开头的文件，执行`/etc/rc5.d/S??.* start` 的操作。

/etc/rc.d/rc*.d 中的系统服务会在系统后台启动或关闭，实质上系统服务主要就是以/etc/int.d/服务文件名 {start, stop} 来启动或关闭的。如果要对某个运行级别中的服务进行更具体的定制，可以通过 chkconfig 命令来完成，或者通过 setup、ntsys、system-config-services 来完成。

如果用户需要自己增加启动的内容，可以在 init.d 目录中增加相关的 shell 脚本，然后在 rc*.d 目录中建立链接文件指向该 shell 脚本。

各个不同的服务其实是有依赖关系的，这里的 shell 脚本的启动或结束顺序是由 S 或 K 字母后面的数字决定，数字越小的脚本越先执行。例如，/etc/rc.d/rc5.d /S01sysstat 就比/etc/rc.d/rc5.d /S99local 先执行。这里/etc/rc.d/rc5.d/S99local 就是最后一个执行的选项，也就是/etc/rc.d/rc.local。

11.6 rc.local

在完成了默认的 runlevel 指定的各项服务的启动后，init 执行用户自定义引导程序/etc/rc.d/rc.local。

其实当执行/etc/rc.d/rc5.d/S99local 时，它就是在执行/etc/rc.d/rc.local。S99local 是指向 rc.local 的符号链接。就是一般来说，自定义的程序不需要执行上面所说的繁琐的建立 shell 增加链接文件的步骤，只需要将命令放在 rc.local 里面就可以了，这个 shell 脚本就是保留给用户自定义启动内容的。

11.7 mingetty

在完成了系统所有服务的启动后，init 启动终端模拟程序 mingetty 来启动 login 进程或 X Window 来等待用户登录。

```
#启动终端
#Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6

#启动 X Window
#Run xdm in runlevel 5
x:5:respawn:/etc/X11/prefdm -nodaemon
```

mingetty 就是启动终端的命令，上述设置表示在 run level 2/3/4/5 时，都会执行/sbin/mingetty。这里执行了 6 个，所以会有 6 个纯文本终端。

除了这 6 个文本模式终端之外，还会执行”/etc/X11/prefdm-nodaemon”来启动 X-Window。

respawn 表示当后面的命令被终止时，init 会主动重新启动该选项，因此当退出终端后会重新生成新的终端来等待输入。

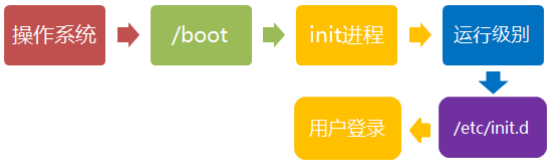
11.8 login

开机启动程序加载完毕以后，就要让用户登录了。

一般来说，用户的登录方式有三种：

1. 命令行登录
2. ssh 登录
3. 图形界面登录

这三种情况，都有自己的方式对用户进行认证。



1. 命令行登录：init 进程调用 getty 程序（意为 get teletype），让用户输入用户名和密码。输入完成后，再调用 login 程序，核对密码（Debian 还会再多运行一个身份核对程序/etc/pam.d/login）。如果密码正确，就从文件 /etc/passwd 读取该用户指定的 shell，然后启动这个 shell。
2. ssh 登录：这时系统调用 sshd 程序（Debian 还会再运行/etc/pam.d/ssh ），取代 getty 和 login，然后启动 shell。
3. 图形界面登录：init 进程调用显示管理器，Gnome 图形界面对应的显示管理器为 gdm（GNOME Display Manager），然后用户输入用户名和密码。如果密码正确，就读取/etc/gdm3/Xsession，启动用户的会话。

Chapter 12

Shell

12.1 Login Shell

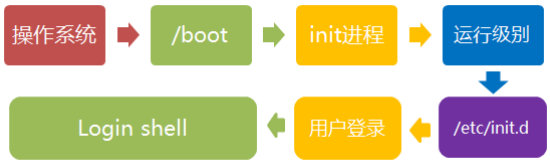
实际上，用户是通过程序来使用操作系统的，而 Shell 就是让用户可以直接与操作系统交互的命令行交互界面，其中用户登录时打开的 shell 就叫做 login shell。

Debian/Fedora 等操作系统中的默认 shell 是 Bash，它会读入一系列的配置文件。上一步的三种情况，在这一步的处理，也存在差异。

1. /etc/profile 是对所有用户都有效的配置，命令行登录时会首先读入，然后/etc/profile 会依次调用其他的针对当前用户的配置文件。

   ```
   ~/.bash_profile
   ~/.bash_login
   ~/.profile
   ```

   上述关于当前用户的文件中只要有一个存在，就不再读入后面的文件了。比如，要是～/.bash_profile 存在，就不会再读入后面两个文件了。
2. ssh 登录：与第一种情况完全相同。
3. 图形界面登录：只加载 /etc/profile 和～/.profile。也就是说，～/.bash_profile 不管有没有，都不会运行。



bash 配置文件的读入可以通过 source 命令来读取，例如～/.bash_profile 会调用～/.bashrc 的设置内容。例如，在下面的示意图中，实线方向代表主线流程，虚线方向代表被调用的配置文件。

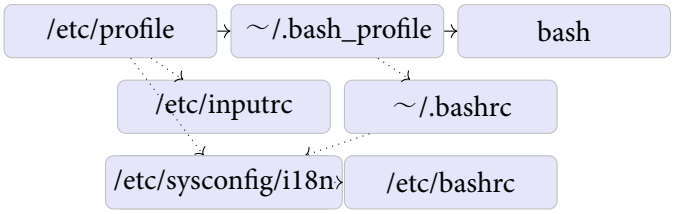


Figure 12.1: login shell 的配置文件读取流程

12.1.1 source

如果用户将自己的偏好设置写入家目录下的.bashrc 中，需要使用 source 命令读入配置后才能生效。

在实际应用中，source 命令和.（小数点）命令是相同的。例如，下面的示例都是将家目录下的.bashrc 设置读入当前环境中。

```
# source ~/.bashrc
# . ~/.bashrc
```

12.1.2 issue

在 bash 的登录界面（例如 tty1 ～ tty6）中的说明是在/etc/issue 中指定的。

```
# vim /etc/issue
Fedora release 20 (Heisenbug)
Kernel \r on an \m (\l)
```

在/etc/issue 中使用反斜杠作为变量调用，其中：

- \d 表示本地端的日期
- \l 显示终端接口
- \m 显示硬件的等级
- \n 显示主机的网络名称
- \o 显示 domain name
- \r 表示操作系统的版本（相当于 `uname -r`）
- \t 显示本地端的时间

- \s 显示操作系统的名称
- \v 显示操作系统的版本

12.1.3 motd

/etc/issue.net 用于向通过 telnet 登录的用户进行说明，而且可以将公共信息加入到/etc/motd 中并呈现给所有登录的用户。例如，如果需要向登录用户通知固定的维护时间，可以在/etc/motd 中加入如下的信息：

```
# vim /etc/motd
Hello everyone,
We will update system at 0:00 1st every month.
Please don't login at that time
```

12.2 Non-login Shell

上一步完成以后，Linux 的启动过程就算结束了，用户已经可以看到命令行提示符或者图形界面了。

用户进入操作系统以后，常常会再手动开启一个 shell，也就是 non-login shell，意思是它不同于登录时出现的那个 shell，不读取/etc/profile 和.profile 等配置文件。

login shell 在取得 shell 时需要完整的登录流程，例如从 tty1 ～ tty6 登录系统时都需要输入用户名和口令。但是，non-login shell 在获得 bash 时不需要执行登录的操作。

在提供图形用户界面的操作系统中，当以 X Window 登录后再以 X 来启动终端时就不再需要输入用户名和口令，这样获得 bash 环境就是 non-login shell，而且在进入 bash 后继续执行 bash 命令来获得的 bash（子进程）也是 non-login shell。

login shell 和 non-login shell 读取的配置文件并不相同。其中，login shell 需要读取/etc/profile 以及～/.bash_profile、～/.bash_login 或～/.profile。

- /etc/profile 是系统整体的设置，只有 login shell 可以读取；

- ～目录下的设置文件属于用户个人设置。

/etc/profile 可以利用用户的标识符（UID）来确定相关的变量，而且也是每个用户登录后取得 bash 时必须读取的配置文件，因此可以用于设置整体环境。

- PATH：根据 UID 决定 PATH 变量是否需要包含/usr/sbin 的系统命令目录；
- MAIL：根据用户帐号设置邮件目录（/var/spool/mail/username）；
- USER：根据用户的帐号设置 USER 变量值；
- HOSTNAME：依据主机的 hostname 来 HOSTNAME 值；
- HISTSIZE：设置历史命令数据，默认为 1000。

另外，/etc/profile 还可以调用外部的设置数据。例如，在默认情况下需要调用如下的数据：

- /etc/inputrc：设置 bash 的热键、Tab 键等；
- /etc/profile.d/*.sh：用于设置 bash 操作接口的颜色、语言、alias 等。
- /etc/sysconfig/i18n（可选）：通过/etc/profile.d/lang.sh 中 LANG 变量来设置 bash 默认的 locale 设置。

注意，Redhat 系列的 Linux 发行版（REHL、Fedora、CentOS）使用/etc/bashrc 文件来为 bash 定义如下的数据：

- 依据不同的 UID 规定 umask 值；
- 依据不同的 UID 规定提示符（即 PS1 变量）；
- 调用/etc/profile.d/*.sh 的设置。

non-login shell 的重要性，不仅在于它是用户最常接触的那个 shell，还在于它仅会读入用户自己的 bash 配置文件～/.bashrc。

大多数情况下，用户对于 bash 的定制都是写在～/.bashrc 文件中。

事实上，Debian 在文件～/.profile 中先判断变量 $BASH_VERSION 是否有值，然后判断主目录下是否存在.bashrc 文件，如果存在就运行该文件。

第三行开头的那个小数点就是 source 命令的简写形式，表示运行某个文件，写成“source ～/.bashrc”也是可以的。

```
if [ -n "$BASH_VERSION" ]; then
  if [ -f "$HOME/.bashrc" ]; then
    . "$HOME/.bashrc"
  fi
fi
```

只要运行～/.profile 文件，～/.bashrc 文件就会连带运行。

如果存在～/.bash_profile 文件，那么有可能不会运行～/.profile 文件。为了让～/.profile 的配置也被读入当前环境，可以把下面代码写入.bash_profile。

```
if [ -f ~/.profile ]; then
```

```
    . ~/.profile
fi
```

早期的时候，计算机运行速度很慢，载入配置文件需要很长时间，Bash 的开发者只能把配置文件分成了几个部分并分阶段载入。

- 系统的通用设置放在 /etc/profile；
- 用户个人的、需要被所有子进程继承的设置放在.profile；
- 不需要被继承的设置放在.bashrc。

除了 Linux 以外，Mac OS X 使用的 shell 也是 Bash，不过它只加载.bash_profile，然后在.bash_profile 里面调用.bashrc。另外，不管是 ssh 登录，还是在图形界面里启动 shell 窗口，都是如此。

Chapter 13

Run Level

运行级别（runlevel[11]）指的是 Unix[1]或者 Linux 等类 Unix 操作系统下不同的运行模式。运行级别通常分为 7 等，分别是从 0 到 6，但如果必要的话也可以更多。例如在大多数 linux 操作系统下一共有如下 6 个典型的运行级别：

- 0 停机
- 1 单用户，Does not configure network interfaces, start daemons, or allow non-root logins
- 2 多用户，无网络连接 Does not configure network interfaces or start daemons
- 3 多用户，启动网络连接 Starts the system normally.
- 4 用户自定义
- 5 多用户带图形界面
- 6 重启

在 Debian Linux 中，2 ～ 5 这四个运行级别都集中在级别 2 上，这个级别也是系统预设的正常运行级别。在 Debian Linux 中，下列路径对应不同的运行级别。当系统启动时，通过其中的脚本文件来启动相应的服务。

- /etc/rc0.d Run level 0
- /etc/rc1.d Run level 1
- /etc/rc2.d Run level 2
- /etc/rc3.d Run level 3
- /etc/rc4.d Run level 4
- /etc/rc5.d Run level 5
- /etc/rc6.d Run level 6

可以通过 chkconfig 来添加服务到不同的运行级别或者取消服务的自动启动。例如，使用 chkconfig 命令来配置服务的示例如下：

[1] The BSD[12] variants don't use the concept of run levels, although on some versions init(8) provides an emulation of some of the common run levels.

```
#将服务添加到服务列表中，可以使用 service camsd start 来启动服务
chkconfig ——add camsd
#将服务删除出服务列表
chkconfig ——del camsd
```

设置服务自动运行的示例如下：

使camsd服务在运行级别3和运行级别5自动运行。

```
chkconfig ——level 35 camsd on
```

使camsd服务在运行级别3和运行级别5不再自动运行。

```
chkconfig ——level 35 camsd off
```

查看服务的自启动状态的示例如下：

```
chkconfig ——list camsd
```

事实上，与 runlevel 有关的启动其实是在/etc/rc.d/rc.sysinit 执行结束之后才开始的。也就是说，其实 runlevel 的不同仅是/etc/rc[0-6].d 目录中启动的服务不同而已，依据启动是否自动进入不同的 runlevel 的设置，可以说：

- 若要每次启动都执行某个默认的 runlevel，可以修改/etc/inittab 内的设置选项。
- 如果仅是暂时更改系统的 runlevel，可以使用`init [0~5]` 来进行更改。
- 每次系统重新启动时都以/etc/inittab 中的设置选项为准。

不同的 runlevel 只是加载的服务不同而已，因此当从 runlevel 5 切换到 runlevel 5 时，系统执行的配置如下：

- 首先比较/etc/rc.d/rc3.d 与/etc/rc.d/rc5.d 中以 K 与 S 开头的文件；
- 在新的 runlevel 中以 K 开头的文件，予以关闭；
- 在新的 runlevel 中以 S 开头的文件，予以启动。

查看当前 runlevel 的示例如下：

```
[root@theqiong ~]#runlevel
N  5
#左边代表前一个 runlevel，右边代表当前的 runlevel。
```

另外，可以分别利用`init 0` 和`init 6` 来关机和重启。

Chapter 14

Usage

接下来需要思考一下，计算机系统是如何工作的。

举例来说，计算机屏幕上面显示的信息，是如何显示出来的？是通过显卡与屏幕显示的。那么如果要看视频呢？这时需要：

- 有视频文件；
- 可以转换视频文件输出的中央处理器；
- 可以显示图像的显示芯片（显卡）；
- 可以传输声音的音效芯片（声卡）；
- 可以输出图像的显示器；
- 可以发出声音的播放器。

也就是说，所有在“工作”的设备都是“硬件”，而且这些就是硬件的工作之一。

现在问题在于，计算机所进行的工作都是硬件来完成的，但是为什么这些硬件知道如何播放视频文件呢？原因就是操作系统在正确的控制硬件的工作。

操作系统可以管理计算机的硬件，包括控制 CPU 进行正确的运算，识别硬盘里的文件并进行读取，还能够识别所有的适配卡，这样才能让所有的硬件全部正确的操作。所以如果没有操作系统，计算机是没有用处的。

虽然操作系统可以完整的控制所有的硬件资源，但是对于用户来说还是不够的。因为操作系统虽然可以控制所有的硬件，但如果用户无法与操作系统交互，那么这个操作系统也就没有什么用处了。简单的来说，以上面的视频文件为例，虽然操作系统可以控制硬件播放视频，但是如果用户没有办法控制何时要播放，那么到底用户要怎么看视频呢？所以说，一个比较“完整的操作系统”应该要包含两部分，分别是：

- 核心与其提供的接口工具；
- 利用核心提供的接口工具所开发出来的软件。

计算机系统由硬件组成，操作系统的出现是为了更有效地控制计算机的硬件资源。操作系统在提供计算机运行所需要的功能（如网络功能）之外，还提供软件开发环境，也就是

在定义上，只要能够让电脑硬件正确无误的运行，那就算是操作系统了。所以说操作系统其实就是核心与其提供的接口工具，不过就如同上面所说，因为核心缺乏与用户沟通的界面，所以目前一般我们提到的“操作系统”都会包含核心与相关的应用软件。

提供了一整套系统调用接口来满足软件开发需求。

现在用 Windows 电脑来做一个简单的说明。打开 Windows 电脑里面的“我的电脑”时，就会显示出硬盘中的文件。这个“显示硬盘里面的文件”，就是核心帮我们做的，但是如果我们要核心去显示硬盘中的某一个目录下的文件，则是由资源管理器这个工具来完成。

那么核心有没有做不到的事？当然有的，举例来说，如果曾经自行安装了比较新的显卡在计算机上，那么常常会发现 Windows 系统会提示：“找不到合适的驱动程序”，也就是说即使有最新的显卡安装在计算机上，而且也有播放视频文件的程序，但是因为“核心”无法使用这个最新的显卡，这时就无法正常的播放视频文件了。虽然所有硬件是由核心来管理的，但如果核心不能识别硬件，那么就无法使用该硬件设备。

核心是操作系统的最底层的东西，由它来管理所有硬件资源的工作状态。每个操作系统都有自己的核心，当有新的硬件加入到系统中的时候，若核心无法识别该硬件，那么这个新的硬件就肯定是无法工作的，因为控制它的操作系统并不能识别它。

一般来说，操作系统核心为了给出用户所需要的正确运算结果，必须要管理的事项有：

1. 系统调用接口（System Call Interface）
   为了方便开发者与操作系统核心的沟通来进一步的利用硬件的资源，需要操作系统核心提供接口来方便程序开发者。
2. 进程管理（Process Control）
   可能同一时间有很多的工作进入到 CPU 等待计算处理，操作系统核心必须要能够控制并调度这些工作进程，让 CPU 的资源得到有效的分配。
3. 内存管理（Memory Management）
   控制整个系统的内存管理，若内存不足，操作系统核心最好还能够提供虚拟内存的功能。
4. 文件系统管理（File System Management）
   文件系统的管理包括数据的输入输出（I/O）和不同文件格式的支持等，如果核心不能识别某个文件系统，那么将无法使用该文件格式的文件。
5. 设备驱动程序（Device Drivers）
   管理硬件是操作系统核心的主要工作之一，当然设备的驱动就是核心需要做的事情。在目前都有所谓的“可加载模块”功能，可以将驱动程序编译成模块，从而就不需要重新编译核心。

所有硬件的资源都是由操作系统核心来管理的，而用户要完成一些工作时，除了通过核心本身提供的功能之外，还可以借助其他的应用软件来完成。举个例子来说，要看视频文件时，除了系统提供的默认媒体播放程序之外，也可以自行安装视频播放软件来播放，这个

由上面的说明中，我们知道硬件是由操作系统“核心”来控制的，而每种操作系统都有其自己的核心，这就产生了一个很大的问题。因为早期硬件的开发者所开发的硬件架构或多或少都不相同，举例来说，2006 年以前的麦金塔是使用 IBM 公司开发的硬件系统（即所谓的 PowerPC），苹果公司则在该硬件架构上开发麦金塔操作系统，而 Windows 则是运行在 x86 架构上的操作系统。由于不同的硬件的功能函数并不相同，IBM 的 PowerPC CPU 与 Intel 的 x86 架构是不一样的，因此 Windows 就不能在苹果计算机上面运行。或者说，如果想要让 x86 上面运行的操作系统也能够在 PowerPC CPU 上运行，就要对操作系统进行修改才可以。不过在 2006 年以后，苹果公司转而使用 Intel x86 硬件架构，因此在 2006 年以后的苹果计算机，其硬件则可能“可以”支持安装 Windows 操作系统了。

播放软件就是应用软件，而这个应用软件可以帮助用户去控制核心来工作（就是播放视频）。可以这样说，核心是控制整个硬件系统的，也是一个操作系统的最底层。然而要让整个操作系统更完备的话，那还需要包含相当丰富的由核心提供的工具以及与核心相关的应用软件。

其实 Linux 就是一个操作系统，它含有最主要的操作系统核心[1]以及操作系统核心提供的工具。或者从本质上讲 Linux 就是一组软件，具体从计算机系统的分层架构来说，就是操作系统核心与系统调用接口。

现在，Linux 提供了一个完整的操作系统中最底层的硬件控制与资源管理的完整架构，这个架构沿袭了 UNIX 良好的传统。此外由于 Linux 的架构可以在 x86 架构计算机上面运行，所以很多的软件开发者将软件也移植到这个架构上面，于是也就有了很多的应用软件。虽然 Linux 仅是其核心与核心提供的工具，不过由于核心、核心工具与这些软件开发者提供的软件的整合，使得 Linux 成为一个更完整的、功能强大的操作系统。

约略了解 Linux 是什么之后，接下来我们要谈一谈，为什么说 Linux 是很稳定的操作系统以及它是如何来的。

如果能够参考硬件的功能函数来修改已有的操作系统或软件代码，那经过修改后的操作系统或软件就能够在其他的硬件平台上运行，这种操作就是我们通常所说的“软件移植”。

因为 Windows 操作系统本来就是针对个人计算机 x86 架构的硬件设计的，所以它当然只能在 x86 的个人电脑上面运行。也就是说，每种操作系统都是在对应的机器上面运行的，这一点需要先了解。不过由于 Linux 是 Open Source（开放源代码）的操作系统，所以其程序代码可以被修改成适合在各种架构的设备上面运行，也就是说 Linux 是具有“可移植性”的。

现代企业对于数字化的目标在于提供消费者或员工一些产品方面的信息以及整合整个企业内部的数据同一性（例如统一的账号管理/文件管理系统等）。另外，金融业等则强调在数据库、安全性等重大关键应用，而学术单位则需要强大的运算能力来进行模拟等，因此 Linux 的应用至少有下面这些：

[1] Torvalds 在开发出 Linux 0.02 内核的时候，其实该内核仅能“驱动 386 所有的硬件”而已，即所谓的“让 386 计算机开始运行，并且等待用户命令输入”，事实上当时能够在 Linux 上面运行的软件还很少。

1. 网络服务器

   承袭了 UNIX 高稳定性的良好传统，Linux 具有稳定而强大的网络功能，因此作为网络服务器来提供诸如 WWW、Mail Server、File Server、FTP Server 的功能等都成为了 Linux 的强项。

   金融业与大型企业也逐渐开始采用 x86 架构主机，学术机构的研究经常需要自己开发应用软件，而 Linux 自身的性能和软件开发及编译环境正好符合这些需求。在实际应用场景中，为了加强整体系统的性能，计算机集群系统（Cluster）的并行计算能力需求也很突出。并行计算指的是将原来的工作分成多份然后交给多台主机去运算，最后再把结果收集起来。这些主机一般是通过高速网络连接起来的，使用计算机集群就可以缩短运算时间，例如气象预报、数值模拟等。

2. 工作站计算机

   工作站计算机与服务器不一样的地方大概就是在于网络服务。工作站计算机本身是不应该提供 Internet 的服务的（LAN 内的服务则可接受)。此外，工作站计算机与桌上型计算机不太一样的地方在于工作站通常得要应付比较重要的计算任务，例如流体力学的数值模式运算、娱乐事业的特效处理、软件开发者的工作平台等。

   Linux 上面有强大的运算能力以及支持度相当广泛的 GCC 编译软件，因此在工作站当中也是相当良好的一个操作系统选择。

3. 桌上型计算机

   桌上型计算机其实就是在办公室使用的计算机，一般称之为 Desktop。

4. 嵌入式系统

   Linux 的核心拥有可定制性以及高效率等特性，从而在嵌入式设备的市场当中具有很大的竞争优势。

   Linux 核心核心里面包含了很多嵌入式设备可能用不到的模块，所以可以将所有不需要的功能移除仅保留需要的程序，这对于嵌入式设备锱铢必较的存储空间来说是很关键的。由于 Linux 核心非常小巧精致，因此可以在很多省电以及较低硬件资源的环境下运行。

目前 Linux 有两种主要的操作模式，分别是图形界面与文字界面。用户所看到的图形界面是使用 X Server 提供的显示相关硬件的功能来达到图形显示的“Window Mananger”所产生的，也就是说，WM 是在 X Server 上面来运行的一套显示图形界面的软件，例如常见的 KDE、GNOME 等都是 WM。如果没有图形界面的辅助，那么将对 Desktop 用户造成很大的困扰。

Linux 早期都是由工程师所发展的，对于图形界面并不是很需要。如果仅想要了解 Linux，并且利用 Linux 来作为桌上型计算机，那么只需要了解 Linux 桌面设定，例如中文输入法、打印机设定、网络设定等概念即可，这时可以选择专为桌上型计算机发行的 Linux 发行版。

在 1986 年图形界面就已经在 UNIX 上面出现了，那时图形界面被简称为 X 系统，而

后来到了 1994 年的时候正式被整合在 Linux 中，微软的 Windows 则是在 1995 年才出现。X Window System 就是以 XFree86 这个计划开发的 X11 视窗软件为管理显示核心的一套图形界面的软件，简称为图形用户界面 (Graphical User Interface，GUI)。

发生错误时可以先自行以显示器输出的信息来进行处理，如果是网络服务的问题时，可以到/var/log 目录里去查看日志文件，这样可以几乎解决大部分的问题了。

一般而言，从 Linux 在发送命令的过程当中或者是 log file 里就可以自己查得错误信息了。

Chapter 15

Configuration

为了配置合理的 Linux 主机系统，除了考虑后续的维护之外首先要根据使用的发行版的特性、软件、升级需求、硬件扩展性以及主机预期的“工作任务”来选择最合适的硬件。

举例来说，需要桌面环境的用户应该会用到 X Window，那么就需要高性能的显卡与内存。如果要做文件服务器，那么就需要配置大容量的硬盘或者是其他的储存设备。

Linux 对于计算机中各个部件/设备的识别与 Windows 系统不同，各个部件或设备在 Linux 系统下都是“一个文件”。Linux 系统中的硬件设备都位于/dev 目录中，例如 IDE 接口的硬盘的文件为/dev/hd[a-d]，而 SATA 接口的硬盘对应的文件为/dev/sd[a-d]。

15.1 Hardware

Linux 早期是与 x86 架构的个人计算机系统紧密结合的，而且硬件与操作系统的关系也很大。

不同的 CPU 之间不可以单纯用时钟频率来判断运算的效率，时钟频率目前仅能用来比较同样的 CPU 的速度。另外比较特别的是 CPU 的“倍频”与“外频”，其中外频是 CPU 与周边设备进行数据传输/运算的速度，倍频则是 CPU 本身运算时候加上去的一个运算速度，两者相乘才是 CPU 的时钟频率。

与 CPU 外频有关的是内存与主板芯片组。一般来说，越快的时钟频率代表越快的 CPU 运算速度，以 Intel 的 P III 时钟频率 933MHz 为例说明如下：

- CPU 外频与倍频：133（外频）× 7（倍频）MHz；
- RAM 频率：通常与 CPU 的外频相同，为 133MHz；
- PCI 接口（包含网卡、声卡等的接口）：133/4=33MHz；
- AGP 接口：133/2=66MHz（这是 AGP 正常的频率）

外频是可以超频的，原本的 CPU 外部频率假设是 133，如果通过某些工具或者主板自身提供的工具就可以将 133 提升到比较高的频率，这就是所谓的超频。

超频可以在比较便宜的 CPU 上面让频率升到比较高。不过超频本身的风险很高，如果是超外频的话，例如到 166MHz 时，AGP 将达（166/2=83）而 PCI 也将达（166/4=41.5），高出正常值很多。

通常，越快的外频会让所有的设备运算频率都会提升，所以可以让效率提高不少，但也可能会造成系统不稳定，例如常常死机或者是缩短某些部件的寿命等。

CPU 是分等级的，很多的程序都针对 CPU 作了优化，所以就会有所谓的 i386、i586、i686 等。基本上，在 P MMX 以及 K6-III 都称为 586 的 CPU，而 Intel 的赛扬以上等级与 AMD 的 K7 以上等级，就被称为 686 的 CPU。

计算机真正运算的核心是 CPU，但是真正传送给 CPU 运算数据的是内存。操作系统的核心、软硬件的驱动程序、所有要读取的文件等都需要先读入内存之后才会传输到 CPU 来进行数据的运算。

此外，操作系统也会将常用的文件或程序等数据常驻在内存内而不直接移除，这样下次取用这个数据时就不需要在去周边存取设备再重新读取一次。

对于一个系统来说，通常越大的内存代表越快速的系统，这是因为系统不用常常释放一些内存内部的数据。以服务器为例，内存容量的重要性有时比 CPU 的速度还要高，而且如果内存不够大，系统就会使用一部分硬盘空间作为虚拟内存，因此内存太小可能会影响到整体系统的性能。

显卡对于图形接口有相当大的影响，要将视频数据显示到显示器时就需要使用到显卡（VGA Card）的相关硬件功能。目前 3D 的画面在计算机游戏接口与工作接口被大量的使用，但是如果这些 3D 画面没有先经过处理而直接进入 CPU，将会影响到整体运算的速度。

显卡开发商在显卡上面安装可以处理这些很耗 CPU 运算时间的硬件来处理这些画面数据，这样不但图形画面处理的速度加快，CPU 的资源也会被用来执行其他的工作。

另外显存的大小可以影响显示器输出的解析度与像素，显存是直接嵌入显卡的，与系统内存没有关系。服务器没有 X Window，显卡并不重要，个人计算机是需要使用到图形接口的，那么这个显存的容量就比较重要。

在个人计算机上面，主流的硬盘存取接口应该是 SATA 与 IDE，SATA 硬盘存取效率要比传统的 IDE 接口高。此外 SATA 的特色就是它与主板连接的排线可以比较长（可长达 1m），并且排线比较细，可以改善主机机壳内部的通风散热。

在 Linux 中 SATA 与 IDE 接口的命名方法稍有不同。一个 IDE 插槽可以接两个 IDE 接口的设备，通过 IDE 设备的跳针（Jumper）来设定主盘（Master）和从盘（Slave），可以在一个 IDE 接口接的两个设备上面以排线接一个 Master 以及一个 Slave 的设备。

IDE 设备的文件名如下表所示：

IDE/Jumper	Master	Slave
IDE1(Primary)	/dev/hda	/dev/hdb
IDE2(Secondary)	/dev/hdc	/dev/hdd

Master 与 Slave 可以在任何一个 IDE 设备上面找到，也就是说如果有两个硬盘，那么可

以将任何一个设成 Master，但是另外一个则必须为 Slave，否则 IDE 接口会无法识别。

再以 SATA 接口来说，由于 SATA/USB/SCSI 等硬盘接口都是使用 SCSI 模块来驱动的，因此这些接口的硬盘设备文件名都是/dev/sd[a-p] 的格式，但是与 IDE 接口不同的是，SATA/USB 接口的硬盘根本就没有一定的顺序，因此只能根据 Linux 内核检测到的硬盘的顺序来命名设备代号，而与实际插槽代号无关。

另外，USB 硬盘设备代号要开机完成后才能被系统识别。

通常在 586 之后生产的主板上面都有两条接排线的接口，而我们称这种接口为 IDE 接口，这也是之前的主流硬盘接口（目前已被 SATA 取代），为了区分硬盘读取的先后顺序，主板上面的这两个接口分别被称为 Primary（主要的）与 Secondary（次要的）IDE 接口，或者被称为 IDE1（Primary）与 IDE2（Secondary）。每一条排线上面还有两个插孔，也就是说一条排线可以接两个 IDE 接口的设备（硬盘或光驱），而现在有两条排线，因此主板在默认的情况下，应该都可以连接四个 IDE 接口的设备。

另外硬盘的 master/slave 判断方法中，除了利用 jumper 主动调整之外，还可以通过 cable 自动选择。如果光驱已经占用了一个硬盘位置，那么最多就只能再安装三个 IDE 接口的硬盘到主机上。

硬盘与 Linux 的硬盘代号有关，由 IDE 1（Primary IDE）的 Master 硬盘开始计算，然后是 IDE 2 的 Slave 硬盘，所以各个硬盘的代号是：

```
IDE\Jumper Master Slave
IDE1(Primary) /dev/hda /dev/hdb
IDE2(Secondary) /dev/hdc /dev/hdd
```

如果只有一个硬盘，而且该硬盘接在 IDE 2 的 Master 上面，那么它在 Linux 里面的代号就是/dev/hdc。如果这个硬盘被分区成两个硬盘分区（partition），那么每一分区在 Linux 里面的代号又是什么，另外如何知道每个 partition 的代号呢？

目前的主机系统硬件要求中，硬盘的转速至少为 7200 转/分，缓冲内存最好大一些。如果主机作为备份服务器或者是文件服务器，那就可能要考虑使用硬盘阵列（RAID），硬盘阵列是利用硬件或软件技术将数个硬盘整合成为一个大硬盘的方法，操作系统只会看到最后被整合起来的大硬盘。由于硬盘阵列是由多个硬盘组成，所以可以在速度、性能、备份等方面有明显优势。

PCI 接口的设备也包括主板内置的 PCI 设备等，其中网卡可以用来连接 Internet，网卡目前都已经可以支持 10/100/1000Mbps 的传输速度，但是网卡的好坏却差别很大。同样是支持 10/100/1000Mbps 的网卡，Intel 与 3Com 的网卡要比一般的杂牌网卡稳定性好，并且占用 CPU 资源低，还会具备其他特殊功能等。

SCSI 接口卡可以用来连接 SCSI 接口的设备。以硬盘为例，目前的硬盘除了个人计算机主流的 IDE/SATA 接口之外，还有 SCSI 接口。由于 SCSI 接口的设备比较稳定，而且运转速度较快，因而速度也会快的多，而且占用 CPU 的资源也比较低。

主板负责沟通所有接口的工作，而沟通所有上面提到硬件的就是主板的芯片组。由于主板上面的芯片组将负责与 CPU、RAM 及其他相关的输出、输入设备的数据通信，所以芯片组设计的好坏也相差很多。

需要再次强调的是，CPU 的外频就是指 CPU 与其他周边设备沟通的速度，而主板芯片组就负责管理各种不同设备的时钟频率操作，因此芯片组的好坏对于系统的影响也是相当大的。另外，目前很多技术可以提升各个设备与芯片组之间沟通的时钟频率。

15.1.1 I/O Address

主板负责各个计算机系统部件之间的沟通，但是计算机的东西又太多了，既有输出输入，又有不同的存储设备，主板芯片组主要通过设备 I/O 地址与 IRQ 来完成这些功能。

I/O 地址有点类似门牌号码，每个设备都有它自己的地址，一般来说不能有两个设备使用同一个 I/O 地址，否则系统就会不知道该如何运算，例如如果你家门牌与隔壁家的相同，那么邮差就没法正确送信到你家。

不过，万一还是造成不同的设备使用了同一个 I/O 地址和而造成 I/O 冲突时，就需要手动的设定一下各个设备的 I/O 地址。

15.1.2 IRQ

除了 I/O 地址之外，还有 IRQ 中断。如果 I/O 可以想象成是门牌号码的话，那么 IRQ 就可以想象成是各个门牌连接到邮件中心（CPU）的专门路径，IRQ 可以用来沟通 CPU 与各个设备。

目前 IRQ 只有 15 个，周边接口太多时可能就会不够用，此时可以选择将一些没有用到的周边接口关掉以空出一些 IRQ 来给真正需要使用的接口，当然也有所谓的 sharing IRQ 的技术。

BIOS 是 Basic Input/Output System 的缩写，输出与输入以及 I/O 地址，IRQ 等都是通过 BIOS 或操作系统来进行设定的。

15.2 Device

以 Windows 系统的观点来看，一块硬盘可以被分区为 C:、D:、E: 等分区，但是在 Linux 系统中，每个设备都被当成一个文件来处理，举例来说，硬盘的文件名称可能就是/dev/hd[a-d]，其中括号内的字母为 a-d 当中的任何一个，也就是/dev/hda，/dev/hdb，/dev/hdc 及/dev/hdd 这四个文件，这四个文件分别代表一个硬盘，另外光驱与软驱分别是/dev/cdrom 和/dev/fd0。

在 Linux 系统当中，几乎所有的硬件设备代号文件都在/dev 这个目录中，所以会看到/dev/hda、/dev/cdrom 等。

下面列出几个常见的设备与其在 Linux 当中的代号：

设备	设备在 Linux 内的代号
IDE 硬盘驱动器	/dev/hd[a-d]
SCSI/SATA/USB 硬盘驱动器	/dev/sd[a-p]
USB 存储器	/dev/sd[a-p]（与 SCSI 硬盘）
当前 CD ROM/DVD ROM	/dev/cdrom
软驱	/dev/fd[0-1]
打印机	25 针：/dev/lp[0-2] USB：/dev/usb/lp[0-15]
当前鼠标	/dev/mouse
磁带机	IDE：/dev/ht0 SCSI：/dev/st0

需要特别留意的是硬盘驱动器代号（不论是 IDE/SCSI/USB 都一样），每个硬盘驱动器的分区（partition）不同时，其硬盘代号还会改变。

/dev 是放置设备文件的目录，需要特别注意的是磁带机的代号，在某些不同的 distribution 当中可能会发现不一样的代号。

15.2.1 uname

Linux 内核有三个不同的命名方案，其中第一个版本的内核是 0.01，其次是 0.02、0.03、0.10、0.11、0.12（第一 GPL 版本）、0.95[1]、0.96、0.97、0.98、0.99 及 1.0。

接下来的旧计划（1.0 和 2.6 版之间) 的版本格式为 A.B.C，其中 A、B、C 分别代表：

- A 大幅度转变的内核。这是很少发生变化，只有当发生重大变化的代码和核心发生才会发生。
- B 是指一些重大修改的内核。内核使用了传统的奇数次要版本号码的软件号码系统 (用偶数的次要版本号码来表示稳定版本)。
- C 是指轻微修订的内核。这个数字当有安全补丁、bug 修复、新的功能或驱动程序，内核便会有变化。

自 2.6.0（2003 年 12 月）发布后，人们认识到更短的发布周期将是有益的，也就自那时起将版本的格式改为 A.B.C.D。

- A 和 B 是无关紧要的
- C 是内核的版本
- D 是安全补丁

自 3.0（2011 年 7 月）发布后，版本的格式为 3.A.B，其中 A、B 分别代表：

- A 是内核的版本；
- B 是安全补丁。

[1]从 0.95 版有许多的补丁发布于主要版本版本之间。

查看当前的内核版本的代码示例如下：

```
$ uname -a
Linux theqiong 3.12.10-300.fc20.x86\_64 #1 SMP Thu Feb 6 22:11:48 UTC 2014
    x86_64 x86_64 x86_64 GNU/Linux
$ uname -r
3.12.10-300.fc20.x86_64
$ cat /proc/version
Linux version 3.12.10-300.fc20.x86\_64 (mockbuild@bkernel02) (gcc version
    4.8.2 20131212 (Red Hat 4.8.2-7) (GCC) ) #1 SMP Thu Feb 6 22:11:48 UTC
    2014
```

15.2.2 lsb_release

每个版本的 Linux 都是使用http://www.kernel.org所发布的核心，它们都遵循 LSB 与 FHS 等的架构，所以差异性其实不大。不过每个 Linux 发行版在发布的时候都有相应的用户群，因此在默认配置中每个版本都有比较适合的用户群体。

如果要查看当前使用的 Linux distribution 使用的 Linux 标准（Linux Standard Base）以及 Linux 内核版本，可以使用如下的命令：[13]

```
$ lsb_release -a
LSB Version:
:core-4.1-amd64
:core-4.1-noarch
:cxx-4.1-amd64
:cxx-4.1-noarch
:desktop-4.1-amd64
:desktop-4.1-noarch
:languages-4.1-amd64
:languages-4.1-noarch
:printing-4.1-amd64
:printing-4.1-noarch
Distributor ID:   Fedora
Description:   Fedora release 20 (Heisenbug)
Release: 20
Codename:   Heisenbug
```

事实上，每个 linux 发行版差异性不大，可以随意选择一个发行版来加以改造以符合自己的应用需求。

要选择的发行版的标准之一就是“选择比较新的 distribution”，这是因为比较新的版本在持续维护套件的安全性上可以让系统比较稳定，而且比较新的发行版在新硬件的支持上面当然也会比较好。

Linux 开发商在发布新版本之前都会针对该版本所默认可以支持的硬件做说明，因此除了可以在 Linux 的 How-To 文件中查询硬件的支持信息之外，也可以到各个相关的 Linux 发行版网站去查询。

Chapter 16

Disk Structure

在系统管理中要管理好自己的分区和文件系统，每个分区不可太大也不能太小，太大会造成硬盘容量的浪费，太小则会产生文件无法存储的困扰。

在了解硬盘的组成及文件系统的概念之后才能理解 Linux 文件系统如何记录文件以及读取文件，而文件的权限与属性就要引入文件系统的 inode 和 block 的概念。

文件系统是创建在硬盘[14] 上面的，硬盘又是由一些圆形硬盘片组成，根据硬盘片能够容纳的数据量而有所谓的单片（一块硬盘里面只有一个硬盘片）或者是多片（一块硬盘里面含有多个硬盘片）的硬盘。

- 圆形的硬盘片是主要记录数据的部分。
- 硬盘里面有所谓的磁头（Head）可以读写硬盘片上的数据，而磁头是固定在机械臂上面的，机械臂上有多个磁头可以进行读取的动作。
- 主轴马达可以转动硬盘片，当磁头固定不动（假设机械臂不动），硬盘片转一圈所画出来的圆就是所谓的磁道（Track）。
- 一块硬盘里面可能具有多个硬盘片，所有硬盘片上面相同半径的那一个磁道就组成了所谓的柱面（Cylinder）。
- 硬盘片上面的同一个磁道就是一个柱面，柱面是硬盘分区（Partition）时的最小单位。
- 由圆心向外划直线，则可将磁道再细分为一个一个的扇区（Sector），扇区就是硬盘片上面的最小存储物理量。
- 通常一个扇区的大小约为 512 Bytes。

在计算整个硬盘的存储量时，简单的计算公式就是：

Cylinder × Head × Sector × 512 Bytes

另外，硬盘在读取时主要是“硬盘片会转动，利用机械臂将磁头移动到正确的数据位置（单方向的前后移动），然后将数据依序读出”。在这个操作的过程当中，由于机械臂上的磁头与硬盘片的接触是很细微的空间，如果有震动或者是脏污在磁头与硬盘片之间时，就会造成数据的损坏或者是实体硬盘整个损坏。

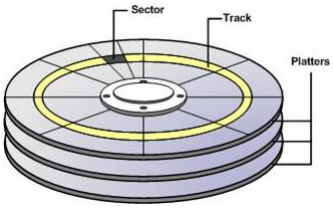


Figure 16.1: Hard Disk Schematic

正确使用计算机的方式应该是在计算机通电之后就绝对不要移动主机并避免震动到硬盘而导致整个硬盘数据发生问题。另外也不要随便将插头拔掉就以为是顺利关机，因为机械臂必须要归回原位，所以使用操作系统的正常关机方式才能够比较好的保养硬盘，因为这会让硬盘的机械臂归回原位。

所谓的硬盘分区就是告知操作系统这块硬盘可以存取的区域是由 A 柱面到 B 柱面的块，这样操作系统才能够控制硬盘磁头去 A−B 范围内的柱面存取数据。如果没有告诉操作系统这个信息，操作系统就无法使用硬盘来进行数据的存取，因为操作系统将无法知道它要去哪里读取数据。

硬盘分区的重点就是指定每一个分区（Partition）的起始与结束柱面。事实上，MBR 就是在一块硬盘的第 1 个扇区上面，也是计算机开机之后要使用该硬盘时必须要读取的第一个区域，在这个区域内记录的就是硬盘里面的所有分区信息以及开机所用的引导加载程序写入的地方。

当一个硬盘的 MBR 坏掉时，由于分区的数据不见了，那么这个硬盘也就几乎可以说是报废了，因为操作系统不知道该去哪个柱面上读取数据。

MBR 最大的限制来自于它的大小不够大，无法存储所有分区与引导加载程序的信息，其中分区表仅有 64 Bytes，因此 MBR 仅提供最多 4 个 Partition 的存储，这就是所谓的 Primary（P）与 Extended（E）的分区最多只能有 4 个的原因，所以说如果预计分区超过 4 个，那么势必需要使用 3P + 1E，并且将所有的剩余空间都分给扩展分区才行，而且扩展分区最多只能有一个，否则只要 3P + E 之后还有剩下的空间将不能使用。

以上叙述的结论就是如果要进行硬盘分区并且已经预计规划使用完 MBR 所提供的 4 个分区（3P + E 或 4P），那么硬盘的全部容量就需要使用完，否则剩下的容量也不能再被使用。不过如果仅是分区出 1P + 1E 的话，那么剩下的空间就还能再分区成两个主分区（Primary Partition）。

- 主分区与扩展分区最多可以有 4 个（硬盘的限制）；
- 扩展分区最多只能有 1 个（操作系统的限制）；
- 逻辑分区是由扩展分区持续分出来的分区；
- 能够被格式化后作为数据访问的分区为主分区（Primary）与逻辑分区（Logical），扩展分区（Extended）无法格式化；

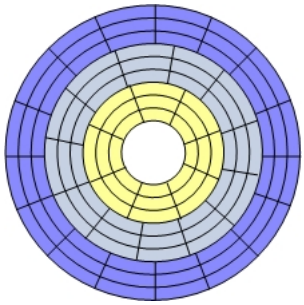


Figure 16.2: Zoned Bit Recording(ZBR)

- 逻辑分区的数量根据操作系统而不同，Linux 操作系统中的 IDE 硬盘最多有 59 个逻辑分区（5 号到 63 号），SATA 硬盘则有 11 个逻辑分区（5 号到 15 号）。

4K 对齐是一种高级硬盘使用技术，用特殊方法将文件系统格式与硬盘物理层上进行契合，为提高硬盘寿命与高效率使用硬盘空间提供解决方案。因该技术将物理扇区与文件系统的每簇 4096 字节对齐而得名。

传统机械硬盘的每个扇区一般大小为 512 字节。当使用某一文件系统将硬盘格式化时，文件系统会将硬盘扇区、磁道与柱面统计整理并定义一个簇为多少扇区以方便快速存储。

例如：现时 Windows 中常见使用的 NTFS 文件系统，默认定义为 4096 字节大小为一个簇，但 NTFS 文件格式因为主引导记录占用了一个磁道共 63 个扇区，真正储存用户文件的扇区一般都在 63 号扇区之后，那么依照计算得出前 63 个扇区大小为：

$$512\text{B} \times 63 = 32256\text{B}$$

并按照默认簇大小得出 63 扇区为多少个簇：

$$32256\text{B} \div 4096\text{B} = 7.875$$

即为每 7 个簇后的第 8 个簇都会是跨越 2 个物理扇区并造成 3584 字节的空间浪费，该结果累计起来将会是个庞大数目，并长时间如此使用将会产生硬盘扇区新旧程度不同导致寿命下降。现时一般使用一些硬盘分区软件在主引导记录的 63 个扇区后作牺牲地空出数个扇区以对齐文件系统的 4096B 每簇，用以节约空间、均匀使用硬盘的各个部分以延长寿命并能避免过多的磁头读写操作，也一定程度上能提升读写速度。

在硬盘发展早期，每扇区为 512 字节比较适合当时硬盘的储存结构。但随着单盘容量的增加，储存密度的上升会明显降低磁头读取硬盘的信噪比，虽然可以用 ECC 校验保证数据可靠性，但消耗的空间会抵消储存密度上升带来的多余空间。所以提出了以 4KB 为一个扇区为主的改变。现时硬盘厂商新推出的硬盘，都将遵循先进格式化（4KB 扇区）的设计以对应新的储存结构和文件系统问题。

相对于机械硬盘来说 4K 对齐对于固态硬盘用意更大，现时的固态硬盘多为使用 NAND Flash 闪存作存储核心，该闪存是有删除写入次数限制的，当次数用完后该固态硬盘便会性能下降甚至报废。很多厂商设计固态硬盘存储方式为不在短时间内删除写入同一个位置，尝试全面地均匀地使用每一个扇区以达到期望寿命，然而在没有 4K 对齐的电脑上这将会使固态硬盘寿命快速下降。

16.1 Disk

Linux 是多用户多任务的环境，因此很可能主机上面已经有很多人的数据在其中了，所以硬盘的分区规划是相当重要的。

系统对于硬盘的需求跟主机开放的服务、数据的分类以及安全性的考量有关，其中“数据安全”并不是指数据被网络入侵所破坏，而是指当主机系统的硬件出现问题时能安全保存用户文件。

在规划硬盘分区时，需要对于 Linux 文件结构有相当程度的认知之后才能够做比较完善的规划。例如，Linux 操作系统中至少要有两个分区，一个是“/”，另一个则是交换分区“swap”。

在默认情况下，Linux 操作系统的大部分组件位于/usr/中，所以可以将/usr/设定的大一点。另外，用户的信息都是在/home 下，因此这个也可以大一些，而/var 下是记录所有预设服务器的日志记录，而且 Mail 与 WWW 预设的路径也在/var，因此这个空间可以加大一些。

下面的目录结构是比较符合容量大且（或）读写频繁的目录结构：

```
/
/usr
/home
/var
swap
```

在硬盘的分区方面建议先暂时以/及 swap 两个分区即可，而且还要预留一个未分区的空间。

在理解 Linux 文件结构之后可以再对硬盘进行规划，通过分析主机的未来用途来确定需要较大容量的目录，以及读写较为频繁的目录，将这些重要的目录分别独立出来而不与根目录放在一起，这样当读写频繁的分区有问题时，至少不会影响到根目录的系统数据，而且书局恢复也比较容易。

BIOS 本身无法支持大硬盘的问题可能导致 Linux 的开机程序找不到 BIOS 提供的硬盘信息，不过 BIOS 可以读写硬盘中最前面的扇区，因此可以在硬盘的最前面分出一个 BIOS 能够识别的小分区，并将系统启动文件放入其中，一般是直接将开机分区安装在 1024 柱面以前，这样就可以避免 Linux 变成“可以安装，但是无法开机”的情况。

在进行安装的时候，规划出三个分区，分别是：

```
/boot
/
swap
```

/boot 只要给 100M Bytes 以内即可，而且/boot 要放在整块硬盘的最前面。

16.2 Partition

硬盘分区是使用分区编辑器（partition editor）在硬盘上划分出若干逻辑部分，硬盘一旦划分成数个分区（Partition），不同种类的目录与文件可以存储进不同的分区。越多分区，也就有更多不同的地方，可以将文件的性质区分得更细，按照更为细分的性质，存储在不同的地方以管理文件。合理的硬盘分区可以提高硬盘使用效率，因而必须要规划出大小适当的硬盘分区。

通过硬盘分区可以允许在一个硬盘上有多个文件系统，但是由于不同的操作系统所使用的文件系统（file system）并不相同，例如 Windows 所使用的是 FAT、FAT32、NTFS 格式，而 Linux 所使用的是 ext2/3/4 文件格式，这两种格式完全不相同。Linux 可以支持 Windows 的 FAT 文件格式，但是 Windows 无法读取 Linux 的文件格式。

- 系统一般单独放在一个分区，其他分区不会受到系统分区出现硬盘碎片的性能影响。
- 碍于技术限制（例如旧版的微软 FAT 文件系统不能访问超过一定的硬盘空间；旧的 PC BIOS 不允许从超过硬盘 1024 个柱面的位置启动操作系统）
- 如果一个分区出现逻辑损坏，不会影响到整块硬盘。
- 在一些操作系统（如 Linux）交换文件通常自己就是一个分区。在这种情况下，双重启动配置的系统就可以让几个操作系统使用同一个交换分区以节省硬盘空间。
- 避免过大的日志或者其他文件占满导致整个计算机故障，将它们放在独立的分区，这样可能只有那一个分区出现空间耗尽。
- 两个操作系统经常不能存在同一个分区上或者使用不同的“本地”硬盘格式。为了安装不同的操作系统，可以将硬盘分成不同的逻辑硬盘。
- 许多文件系统使用固定大小的簇将文件写到硬盘上，这些簇的大小与所在分区文件系统大小直接成比例。如果一个文件大小不是簇大小的整数倍，文件簇组中的最后一个将会有不能被其它文件使用的空闲空间。这样，使用簇的文件系统使得文件在硬盘上所占空间超出它们在内存中所占空间，并且越大的分区意味着越大的簇大小和越大的浪费空间。所以，使用几个较小的分区而不是大分区可以节省空间。
- 每个分区可以根据不同的需求定制。例如，如果一个分区很少往里写数据，就可以将它加载为只读。如果想要许多小文件，就需要使用有许多节点的文件系统分区。

`All problems in computer science can be solved by another level of indirection.——Dav`
计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决。

- 在运行 Unix 的多用户系统上，有可能需要防止用户的硬连接攻击。为了达到这个目的，/home 和/tmp 路径必须与如/var 和/etc 下的系统文件分开。

硬盘分区可做看作是逻辑卷管理（Logical volume management，LVM）前身的一项简单技术。LVM 为计算机中的大量存储设备（Mass storage devices）提供更有弹性的硬盘分区方式。

LVM 作为一种抽象化存储技术，实现的方式会根据操作系统而有所不同。基本上，它是在驱动程序与操作系统之间增加一个逻辑层，以方便系统管理硬盘分区系统。

16.2.1 Windows

Microsoft Windows 的标准分区机制是创建一个分区 C:，其中操作系统、数据和程序都在这个分区上。另外，它推荐创建不同的分区或者使用不同的硬盘，其中一个分区上存储操作系统；而其它分区或者驱动器，则供应用程序或者数据使用。

如果可能的话，在不包含操作系统的硬盘上，为交换文件创建一个单独的分区，尽管这并不意味着两个硬盘都不会断电。在进行预分区工作之后，很容易就可实现操作系统不存储在 C 分区上甚至是 C 分区根本就不存在。这样做有一些益处，一些设计拙劣的病毒或者特洛伊木马将不能覆盖关键的系统文件或者控制系统。“我的文档”文件夹、“特殊文件夹”主目录可以加载到一个独立分区上以利用所有空闲空间。

16.2.2 UNIX

对于基于 UNIX 或者如 Linux 这样类似于 Unix 的操作系统来说，分区系统创建了 /、/boot、/home、/tmp、/usr、/var、/opt 和交换分区。这就保证了如果其中一个文件系统损坏，其它的数据 (其它的文件系统）不受影响，这样就减少了数据丢失。这样做的一个缺点是将整个驱动器划分成固定大小的小分区，例如，一个用户可能会填满 /home 分区并且用完可用硬盘空间，即使其它分区上还有充足的空闲空间。

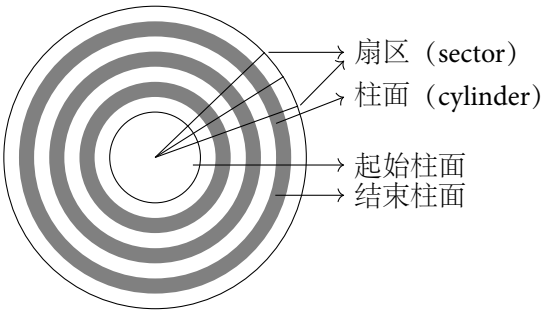
良好的实现方法要求用户预测每个分区可能需要的空间，比如典型的桌面系统使用另外一种约定。

- “/”（根目录）分区包含整个文件系统和独立的交换分区。
- /home 分区对于桌面应用来说是一个有用的分区，因为它允许在不破坏数据的前提下干净地重新安装（或者另外一个 Linux 发行版的更新安装)。

硬盘主要由盘片、机械臂、磁头和主轴马达等组成，实际上硬盘是以 sectors（扇区)、cylinder (柱面)、partitions (分区) 来作为存储的单位，而最底层的实体硬盘单位就是 sectors，

通常一个 sector 大约是 512 bytes 左右，不过在硬盘进行格式化时可以将数个 sector 格式化成为一个逻辑扇区（logical block），通称为 block，blocks 为一个文件系统存取的最小量。

假如硬盘只有一个盘片，那么盘片组成可以如下所示：



基本上，硬盘是由最小的物理组成单位——扇区（sector）所组成，数个扇区组成一个同心圆时就称为柱面（cylinder），最后构成整个硬盘。

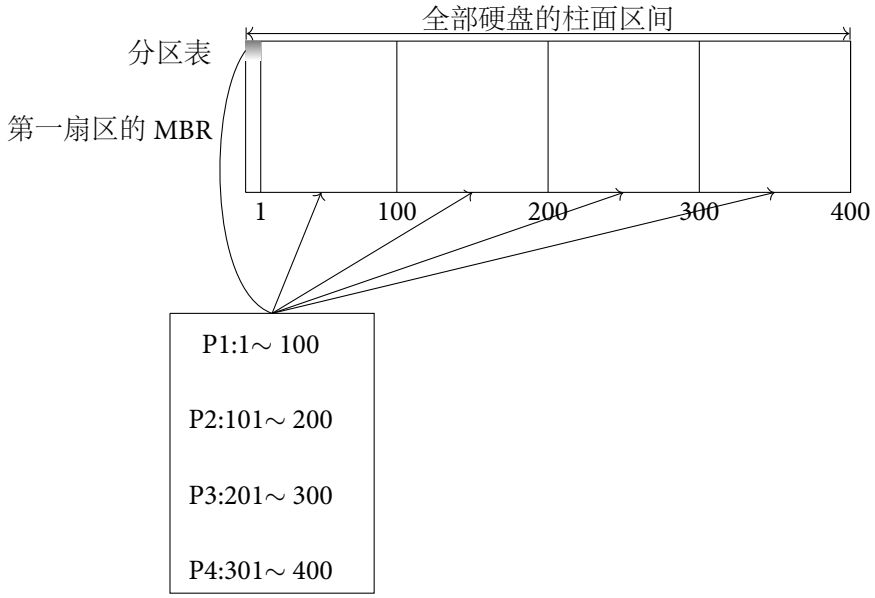
整块硬盘的第一个扇区特别重要，它记录了整块硬盘的重要信息，主要包括两个重要的信息，分别是：

1、主引导分区（Master Boot Record，MBR）是可以安装引导加载程序的地方，有 446bytes。

2、分区表（partition table）记录整块硬盘分区的状态，有 64bytes。

MBR 是很重要的，系统在开机的时候会主动去读取这个区块的内容，这样系统才会知道系统的安装位置以及应该如何开机，尤其是需要通过多重引导系统来选择相应的操作系统的时候。

若将硬盘以长条形来看，然后将柱面以柱形图来看，那么这 64bytes 的分区表记录区段可以用下图表示：



至于 partition table，简单的说“硬盘分区”就是在修改这个 partition table 而已，操作系统访问硬盘是利用参考柱面号码的方式来进行的，基本上 partition table 就定义了“第 n 个硬盘区块是由第 x 柱面到第 y 个柱面”，所以每次当系统要去读取第 n 硬盘区块时，就只会去读取第 x 到 y 个扇区之间的数据。

不过，由于这个分区表的容量有限，在分区表所在的 64bytes 容量中，当初设计的时候就只设计成 4 组记录区，每组记录区记录了该区段的起始与结束的柱面号码，这些分区记录又被称为 Primary（主分区）及 extended（扩展分区），也就是说一块硬盘最多可以有 4 个（Primary + extended）的扇区，其中 extended 只能有一个，因此如果要分区成四个硬盘分区的话，那么最多就是可以按照：

P + P + P + P
P + P + P + E

的情况来分区了。

其中需要特别注意的是，如果上面的情况中，3P + E 只有三个“可用”的硬盘分区，如果要四个都“可用”，就得分区成 4P。extended 不能直接被使用，还需要分区成 Logical 才可以。

关于主分区、扩展分区与逻辑分区

1、主分区与扩展分区最多可以有 4 个（硬盘自身限制）；

2、扩展分区最多只能有一个（操作系统自身限制）；

3、逻辑分区是由扩展分区持续细分出来的分区；

4、主分区和逻辑分区能够在被格式化后作为数据访问的分区，扩展分区无法格式化；

5、逻辑分区的树林根据操作系统有所不同。

在 Linux 系统中，IDE 硬盘最多有 59 个逻辑分区（5 号到 63 号），SATA 硬盘则有 11 个逻辑分区（5 号到 15 号）。

假设上面的硬盘设备文件名为/dev/hda 时，那么这四个分区在 Linux 系统中的设备文件名如下所示，这里重点在于文件名后名会再接一个数字，这个数字与该分区所在的位置有关。

```
P1：/dev/hda1
P2：/dev/hda2
P3：/dev/hda3
P4：/dev/hda4
```

那么为什么要有 extended？这是因为如果我们要将硬盘分成 5 个分区的话，就需要 extended。由于 MBR 仅能保存四个 partition 的数据记录，超过 4 个以上时系统允许在额外的硬盘空间放置另一份硬盘分区信息，那就是 extended。

假设将硬盘分区成为 3P + E，那么那个 E 其实是告诉系统，硬盘分区表在另外的那份 partition table 中，也就是说，那个 extended 其实就是“指向（point to）”正确的那个额外的 partition table。

本身 extended 是不能在任何系统上面被使用的，还需要再额外的将 extended 分区成 Logical（逻辑）分区才能被使用，所以通过 extended 就可以分区超过 5 个可以利用的 partition。

不过，在实际的分区时还是容易出现问题的，下面我们来思考看看：

思考一：如果要将硬盘“暂时”分区成四个 partition，同时还有其他的空间可以在以后进行规划，那么该如何分区？

说明：

Primary + extended 最多只能有四个 partition，而如果要超过 5 个 partition，那就需要 extended，因此在这个例子中，千万不能分区成四个 Primary ，假如 20 GB 的硬盘，而 4 个 Primary 共用去了 15 GB，那么剩下的 5 GB 将完全不能使用，这是因为已经没有多余的 partition table 记录区可以记录了，因此也就无法进行额外的分区，当然空间也就被浪费了。

因此，如果要分区超过 4 个分区以上时，记得一定要有 extended 分区，而且必须将所有剩下的空间都分配给 extended，然后再以 Logical 的分区来规划 extended 的空间。另外考虑到硬盘的连续性，一般建议将 extended 的分区放在最后面的柱面内。

思考二：可不可以仅分区 1 个 Primary 与 1 个 extended 呢？

可以！

说明：

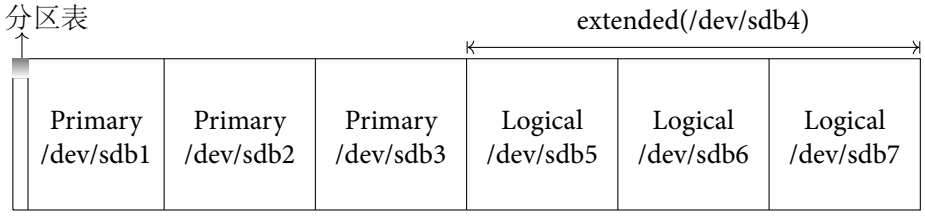
基本上，Logical 的号码可达 63，因此可以仅分一个主分区并且将所有其他的分区都给 extended，利用 Logical 分区来进行其他的 partition 规划，这也是早期 Windows 操作系统的操作方式。

此外，逻辑分区的号码在 IDE 可达 63 号，SATA 可达 15 号，因此仅分区为 1 个 Primary 和 1 个 extended 时，可以通过 extended 分区继续分出 Logical 分区。

思考三：如果要将 SATA 硬盘分区成 6 个可以使用的硬盘分区，那么每个硬盘在 Linux 下面的代号是什么？

说明：

由于硬盘在 Primary + extended 最多可以有 4 个，因此在 Linux 下已经将 partition table 1～4 先留下来了，如果只用了 2 个 P + E 的话，那么将会空出两个 partition number。具体来说，假设将四个 P + E 都用完了，那么硬盘的实际分区会如下图所示：



针对安装的硬盘的类型，实际可以使用的是：

SATA：

/dev/sda1，/dev/sda2，/dev/sda3，/dev/sda5，/dev/sda6，/dev/sda7

IDE：

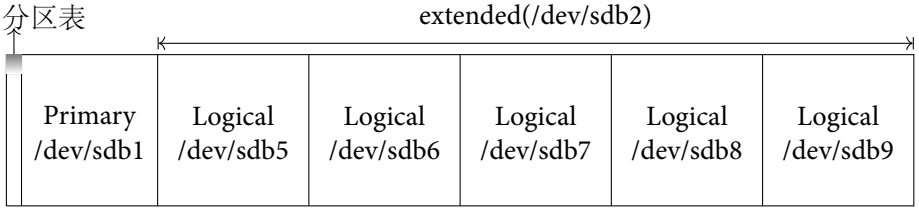
/dev/hda1，/dev/hda2，/dev/hda3，/dev/hda5，/dev/hda6，/dev/hda7

这六个 partition。

至于/dev/sda4（/dev/hda4）这个 extended 分区本身仅是用来规划出来让 Logical 可以利用的硬盘空间。

其实在每个 partition 的最前面扇区都会有一个特殊的区块，称为 super block，extended 指向的就是/dev/hda4 的 super block 处，该处就是额外记录的那个 partition table。

如果只想要分区 1 个 Primary 与 1 个 extended，这个时候的硬盘分区会变成如下所示：



1 ～ 4 号已经被预留下来了，/dev/sda3，/dev/sda4 这两个代号则是空的，所以第一个逻辑分区的代号由 5 号开始，后面的就以累加的方式增长。

举例来说，假设硬盘总共有 1024 个 cylinder（利用 blocks 结合而成的硬盘计算单位），那么在硬盘的起始柱面（就是硬盘分区表，可以想象成要读取一块硬盘时最先读取的地方），如果写入 partitions 共有两块，一块是 Primary，一块是 extended，而且 extended 也只规划成一个 Logical，那么硬盘就是只有两个分区（对于系统来说，真正能使用的有 Primary 与 Logical 的扇区，extended 并无法直接使用的，需要再加以规划成为 Logical 才行），而且在 partition table 中也会记录 Primary 是由“第 n_1 个 cylinder 到第 n_2 个 cylinder”，所以当系统要去读取 Primary 时，就只会在 $n_1 \sim n_2$ 之间的实体硬盘区域活动。

基本上，硬盘分区时仅支持一个 Primary 与一个 extended，其中 extended 可以再细分成多个 Logical 的硬盘分区。

硬盘中的主分区和扩展分区最多可以有 4 个（硬盘分区表的限制），扩展分区最多只能有一个（操作系统的限制），逻辑分区是由扩展分区持续划分出来的分区。

Linux 基本上最多可以有 4 个 Primary 的硬盘，可以支持到 3 个 Primary 与一个 extended，其中 extended 若再细分成 Logical 的话，则全部 Primary + extended + Logical 应该可以支持到 64 个（针对 IDE 硬盘而言）和 16 个（针对 SATA 硬盘而言）。

硬盘分区主要可分为下面几个步骤：

1、将旧有的分区表删除；

2、建立新的主分区（Primary）及扩展分区（extended）；

3、保存分区表；

4、以工具格式化已分区的硬盘。

删除分区之后硬盘中就没有分区表的存在了，所以这个硬盘的系统种类就变成了未分区。

分区表被删除后，重新分区时需要确定分区类型是主分区还是扩展分区。主分区在 Microsoft Windows 系统下就是 C 盘，其他的是扩展分区并非逻辑分区。注意，所谓的“逻辑分区”是包含在扩展分区中的，可以使用逻辑分区将扩展分区分为几个分区，这些新分区就是“逻辑分区”。

分区时通常在第一步是输入“启始柱面”，然后会要求输入“结束柱面”，结束柱面的输入方法有两种模式，一种是输入柱面区，一种是输入所需要的 MB 数，通常是输入 MB 数。

其实所谓的“分区”只是针对主引导扇区中的 64bytes 的分区表进行设置，而且硬盘默认的分区表仅能写入四组分区信息，这四组分区信息被称为主（Primary）或扩展（extended）分区。

硬盘分区的最小单位为柱面（cylinder），当系统要写入硬盘时，需要参考分区表才能针对某个分区进行数据处理。

16.2.3 Linux

安装 Linux 时可选的分区方式[15] 包括：

1. 删除硬盘上的所有分区，并建立自动 Linux 默认分区表。
2. 删除所选硬盘上已有的 Linux 分区，并自动建立 Linux 默认分区表。
3. 使用硬盘上所剩余的自由空间自动建立 Linux 分区表。
4. 自定义分区。

当硬盘上不存在任何操作系统或是想要删除现有操作系统，并且只想安装 Linux 系统的话，可以选择第一种方式进行分区，它会删除现有的所有分区，并自动建立一套 Linux 分区。

如果当前硬盘上已经安装了一份 Linux 系统，并且想要覆盖该系统的话，可以选择第二种方式，安装程序同样会删除现有的 Linux 分区并自动建立一套 Linux 分区。

如果硬盘上面还有未分配的硬盘空间的话，你可以选择第三种方式，安装程序不会修改现有分区，而会在未分配的自由分区自动建立一套 Linux 分区。

如果对 Linux 分区非常熟悉，或者需要自定义分区大小，可以选择第四种方式。在选择了自定义分区后，那么下一步就需要手动分配硬盘分区了，Linux 可以分为这样几个分区：

- /（根分区）；
- /boot（启动分区）；
- /home（家目录分区）；
- /tmp（临时文件分区）；
- /usr（程序分区）；
- /var（日志分区）；
- swap（虚拟内存，又称交换分区）。

其中必须要有的是/（根分区）和/boot（启动分区），此外的/home、/tmp、/usr、/var 可以单独建立分区，也可以不建立分区。如果/home、/tmp、/usr、/var 不建立分区，它们会以文件夹的形式自动挂载到/（根分区）下。

- /分区是每个 Linux 系统必须要有的分区，根分区/是 Linux 文件系统的起点。
- /boot 分区是存放启动文件的，也是必须要有的分区。
- /home 分区是存放用户个人文件的地方，可以视用户文件的多少、大小而确定该分区的大小。
- 在 Linux 中绝大多数程序默认是安装在/usr 下面的，/usr 分区又称为程序分区。
- /var 是用来存放系统日志的地方。
- /tmp 是用来存放系统临时文件的地方。
- swap 分区是虚拟内存分区。

Chapter 17

Formatting

格式化是指对硬盘或硬盘中的分区（partition）进行初始化的一种操作，这种操作通常会导致现有的硬盘或分区中所有的文件被清除。

格式化通常分为低级格式化和高级格式化。如果没有特别指明，对硬盘的格式化通常是指高级格式化，而对软盘的格式化则通常同时包括这两者。

- 低级格式化处理碟片表面格式化赋予磁片扇区数的特质；
- 低级格式化完成后，硬件碟片控制器（disk controller）即可看到并使用低级格式化的成果；
- 高级格式化处理“伴随着操作系统所写的特定信息”。

17.1 Low-Level Formatting

低级格式化（Low-Level Formatting）又称低层格式化或物理格式化（Physical Format），对于部分硬盘制造厂商，它也被称为初始化（initialization）。

最早，伴随着应用 CHS 编址方法、频率调制（FM）、改进频率调制（MFM）等编码方案的硬盘的出现，低级格式化被用于指代对硬盘进行划分柱面、磁道、扇区的操作。现今，随着软盘的逐渐退出日常应用，应用新的编址方法和接口的硬盘的出现，这个词已经失去了原本的含义，大多数的硬盘制造商将低级格式化（Low-Level Formatting）定义为创建硬盘扇区（sector）使硬盘具备存储能力的操作。

现在，人们对低级格式化存在一定的误解。多数情况下，低级格式化往往是指硬盘的填零操作，而且“非必要”的情况下尽量不对硬盘进行低级格式化。

对于一张标准的 1.44 MB 软盘，其低级格式化将在软盘上创建 160 个磁道（track）（每面 80 个），每磁道 18 个扇区（sector），每扇区 512 位位组（byte）；共计 1,474,560 位组。需要注意的是：软盘的低级格式化通常是系统所内置支持的。通常情况下，对软盘的格式化操作即包含了低级格式化操作和高级格式化操作两个部分。

17.2 High-Level Formatting

高级格式化又称逻辑格式化，它是指根据用户选定的文件系统（如 FAT12、FAT16、FAT32、NTFS、EXT2、EXT3 等），在硬盘的特定区域写入特定数据，以达到初始化硬盘或硬盘分区、清除原硬盘或硬盘分区中所有文件的一个操作。高级格式化包括对主引导记录中分区表相应区域的重写、根据用户选定的文件系统，在分区中划出一片用于存放文件分配表、目录表等用于文件管理的硬盘空间，以便用户使用该分区管理文件。

在 DOS 环境下，有多种软件可以执行格式化的操作，系统通常也以外部命令的形式提供一个命令行界面的格式化软件“Format”。

Format 命令的参数包括将被执行格式化的硬盘，以及一些其他次要参数，如簇的大小、文件系统的格式等。

Format 命令通常的格式是：

```
Format X:
```

X 为所希望被执行格式化操作的盘符，如希望格式化 C 盘，则将 X 替换为 C，如此类推。加入 “Q” 参数可以执行快速格式化。

在 Windows 环境下，格式化的操作相对简单。除了可以使用图形化的操作界面执行格式化操作之外，也可以使用命令行的方式进行操作，具体的方法与 DOS 环境下类似。不过对硬盘执行格式化操作时，用户需要拥有系统管理员权限。

在 Unix/Linux 环境下，通常使用 mkfs 命令执行格式化操作，mkfs 命令需要的参数有设备路径和文件系统格式等。对硬盘执行格式化操作时，用户同样需要拥有 root 权限。

17.3 Advanced Format

先进格式化（Advanced Format）是 IDEMA（International Disk Drive Equipment and Materials Association）于 2009 年 12 月 [1] 制定的硬盘格式化标准。IDMEA 在 2005 年与 Hitachi、Seagate、WD、LSI、Intel、Microsoft、Dell、HP、Lenovo 等硬软件厂商制定出 1024 字节、2048 位组和 4096 字节三种容量扇区配置，先进格式化是规范中的 4096 字节（4KB）配置。

从 2011 年 1 月 1 日起，硬盘厂商新推出的硬盘时，都将遵循先进格式化（4KB）的设计。

在硬盘发展早期，每扇区为 512byte 比较适合当时硬盘的储存结构。但随着单盘容量的增加，储存密度的上升会明显降低磁头读取硬盘的信噪比，虽然可以用 ECC 校验保证数据可靠性，但消耗的空间会抵消储存密度上升带来的多余空间。所以提出了以 4kbyte 为一个扇区为主的改变。

最主要的好处减少 ECC 的占用和提升 ECC 校验效率。因为 512byte 扇区需要另外 40byte 作为 ECC 校验空间，而 4kbyte 扇区只需要 100byte，所以，同样提供 4kbyte 扇区空间，使用先进格式化能节约出 220byte 的储存空间，而且能令 ECC 校验完成更多空间的检验纠错，提高 ECC 校验的效率。

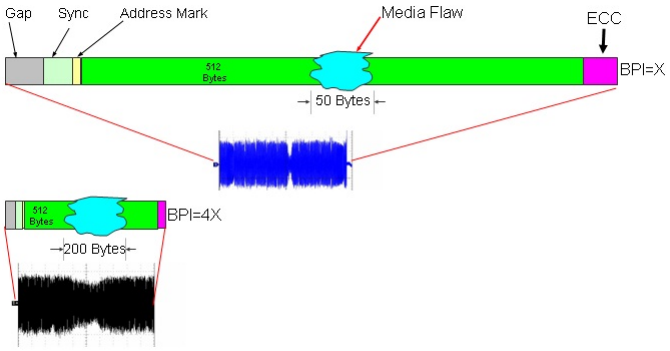


Figure 17.1: 512byte 和 4Kbyte 扇区受到物理污染时所产生的电磁信号影响示意

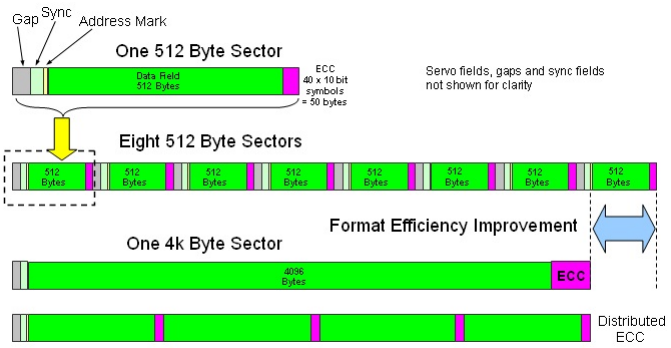


Figure 17.2: 512byte 和 4Kbyte 的物理空间占用比较示意

另外，通常在 x86 架构下的内存分页容量为 4KB，而且很多硬盘文件系统（如 NTFS、ext3、HFS+ 等）的簇容量也为 4KB，而如果使用 4KB 为一个扇区，硬盘对一个扇区的读写数据量刚好装满一个内存页或对应文件系统分区的一个簇，能避免过多的磁头读写操作，一定程度上能提升读写速度。

现在推行主要问题为 Windows 5.x 核心系统（Windows 2000、Windows XP、Windows Server 2003）读取分区无法对准扇区而读取出错和文件系统，簇横跨多个扇区造成转换延迟影响随机写入性能。除外一些较旧版本的硬盘管理工具在不支持 4Kbyte 扇区的情况下也会发生类似的情况。WD 提供了固件模拟和工具校正的方法（WD Align 程序）临时解决，但最根本的解决为升级原生支持 4Kbyte 扇区的 Windows 6.x 核心操作系统，如 Windows Vista、Windows 7。

较新的 Linux、Mac OS X 由于较早开始对 4kbyte 扇区的支持，所以基本能不做调整就能直接使用先进格式化后的硬盘。

Chapter 18

Installaton

在规划硬盘分区安装 Linux 时，如果硬盘大于 60GB 时可能会出现找不到启动分区的问题，那就必须要独立出/boot 这个分区。

18.1 Partition

空间管理、访问许可与目录搜索的方式，依属于安装在分区上的文件系统。当改变大小的能力依属于安装在分区上的文件系统时，需要谨慎地考虑分区的大小。

实际上在 Linux 安装的时候，已经提供了相当多的默认模式让用户选择分区的方式，不过分区的行为可能都不是很符合用户主机的需求，毕竟每个人的“想法”都不太一样。强烈建议使用“自定义安装，Custom”这个安装模式。在某些 Linux distribution 中会将这个模式称为“Expert，专家模式”。

18.1.1 Directory tree

Linux 内的所有数据都是文件，因此整个 Linux 系统最重要的地方就是目录树结构。

目录树结构就是以根目录为主，然后向下呈现分枝状目录形式的一种文件结构，而且整个目录树结构最重要的就是根目录（root directory），其表示方法为一条斜线“/”，所有的文件都与目录树有关。

所有的文件都是由根目录（/）衍生来的，而次目录下还能够有其他的数据存在。

Linux 系统使用的是目录树结构，而实际上文件数据是存放在硬盘分区中的，而如果要理解目录树结构与硬盘内的数据的关系，就要引入“挂载（mount）”的概念。

所谓的“挂载”就是利用一个目录当成进入点，将硬盘分区的数据放置在该目录下，进入该目录就可以读取该分区，这个操作就称为“挂载”，那个进入点的目录称为“挂载点”。

Linux 系统最重要的就是根目录，因此根目录一定要挂载到某个分区，其他的目录则可以根据用户自己的需求挂载到其他不同的分区。

另外，默认情况下 Linux 是将光驱的数据存放到/media/cdrom 里，但光驱也可以被用户挂载到其他目录。

18.1.2 Mount-point

在安装 Linux 系统时就得要规划好硬盘分区与目录树的挂载，初次接触 Linux 时可能只需要最简单的分区方式，即只要分区“/”及“swap”即可，其中直接以最大的分区“/”来安装系统。

另外，/usr/是 Linux 的可执行程序及相关的文件目录。建议分区时预留一个备份的分区。

Linux 默认的目录是固定的，所以通常我们会将/var 及/home 这两个目录稍微加大一些。另外建议可以预留一个分区来备份系统核心与脚本（scripts），当 Linux 重新安装的时候，一些文件马上就可以直接在硬盘当中找到。

另外，Linux 安装之前会自动的把硬盘格式化，而且硬盘至少需要 2GB 以上才可以选择“Server”安装模式。

目前几乎所有的 Linux 发行版都是支持光盘启动的，只是必须要确定系统的第一个启动设备为光驱，可以在 BIOS 里面设定引导设备的次序。

如果使用光盘启动时发生错误，很可能是由于硬件不支持，建议再仔细地确认一下硬件是否有超频或者其他不正常的现象，另外安装光盘来源也需要确认。

一般 Linux 都会支持至少两种安装以上的安装模式，分别是文字（text）模式与图形（graphic）界面。

若想要以文字界面来安装，可以在启动时输入“linux text”来让安装程序以命令行模式安装。不过要注意的是，如果在 10 秒左右没有在 boot: 后输入任何提示的话，安装程序就会以默认的图形界面来安装。

硬件检测完毕之后会出现一个是否校验光盘的画面，如果要检查光盘的话会花去很多时间，所以如果确定光盘来源没有问题，可以直接跳过该选项即可。

略过光盘校验工作后，由于使用的是图形界面的安装模式，安装程序就会去检测显示器、键盘、鼠标等相关的硬件。

在完成了一些硬件方面的检测之后就可以进入图形界面的安装。基本上，图形安装界面分为左右两个区域，左边主要是作为“说明”之用，右边才是真正的操作区域。

安装程序可以使用不同的语言，可以选择相应的语言来进行安装。因为每个地区的键盘上面的字母设置都不一样，如果使用英文的键盘配置，可以选择“English（United States）”。

如果主机用来进行软件开发，就需要安装“系统开发工具”，这样就可以把 gcc、kernel-headers、kernel-source 等安装到主机上。

基本上，Linux distribution 为用户规划好一些主机使用模式了，举例来说，如果想要使

用 Desktop 型计算机的功能，那么可以选择“个人计算机”项目，它会主动的进行相应的硬盘分区以及相关的软件选择。不过可能硬盘 partition 就交给系统主动去判断处理，而且系统的默认分区与套件的选择也不见得就会跟用户需求一致，因此建议务必选择“自定义安装”。

可以使用 Linux distribution 附带的 Disk Druid 工具来进行硬盘分区。注意，如果是新硬盘，可能会发生错误告知用户安装程序找不到 partition table。接下来的画面则是在操作硬盘分区的主要画面，这个画面主要分为三大区块，最上方为硬盘的分区示意图，目前因为硬盘并未分区，所以呈现的就是一整块而且为 Free 的字样。中间则是命令区，下方则是每个分区（partitions）的设备文件名、挂载点与目录、文件系统类型、是否需要格式化、分区所占容量大小以及开始与结束的柱面号码等。

至于命令区，总共有六大区块，其中 RAID 与 LVM 是硬盘特殊的应用，命令的作用如下：

（1）“新增”是指增加新分区，也就是建立新的硬盘分区；

（2）“编辑”是指编辑已经存在的硬盘分区，可以在实际状态显示区选择想要修改的分区，然后再选择“编辑”即可进行该分区的编辑动作。

（3）“删除”是指删除一个硬盘分区。同样地，要在实际状态显示区点击选择想要删除的分区。

（4）“重设”则是恢复最原始的硬盘分区状态。

18.2 Directory

18.2.1 /

创建根目录“/”分区时，默认使用 Linux 提供的文件系统格式，比如 ext2/3/4。挂载点的选择可以手动输入或者使用系统默认，此外这里要说明 Linux 支持的文件系统类型包括：ext2/3/4、physical volume（LVM）、software RAID、swap、vfat 等。

- ext3：ext3 比 ext2 文件系统多了日志的记录，在系统恢复时比较快。
- physical volume（LVM）：这是用来弹性调整文件系统大小的一种机制，可以让文件系统大小变大或变小而不改变原有的文件数据的内容。
- software RAID：利用 Linux 系统的特性，通过软件模拟仿真动态硬盘阵列的功能。
- swap：内存交换空间，由于 swap 并不会使用到目录树的挂载，所以用 swap 就不需要指定挂载点。
- vfat：同时被 Linux 和 Windows 所支持的文件系统类型，如果有数据交换的需要，可以构建一个 vfat 的文件系统。

18.2.2 /boot

如果将/boot 独立分区，务必要把该分区放在整块硬盘的最前面，因此针对/boot 就要选择“强制为主分区”选项。

18.2.3 swap

简单的说，内存交换分区（swap）可以被看作是“虚拟内存”，它不需要挂载点，因此分区时就就没有“/”挂载点前缀。如果物理内存不足，当系统负荷突然之间加大时，此时可以使用硬盘的 swap 分区来模拟内存的数据存取。不过，虚拟内存的速度会比较慢。当有数据被存放在物理内存里，但是这些数据又不是常被 CPU 所取用时，那么这些不常被使用的程序将会被放到虚拟内存当中，而将速度较快的物理内存空间释放出来给真正需要的程序使用。

在传统的 Linux 说明文件中，通常 swap 建议的值大约是“内存的 1.5 到 2 倍之间”，但是这个数值也需要根据实际情况而定，比如说如果系统内存达到 4G 以上时，swap 也可以不必额外设置。

18.2.4 /home

接下来就是创建/home 目录分区。

18.3 Notebook

笔记本电脑与一般计算机略有不同，它加入了非常多的电源管理机制或者是其他硬件的管理机制，因此在安装 Linux 操作系统过程加载内核时不要加载一些功能，于是就要输入下面这些“内核参数”：

```
boot:linux nofb apm=off acpi=off pci=noacpi
```

其中 apm（Advanced Power Management）是早期的电源管理模块，acpi（Advanced Configuration and Power Interface）则是近期的电源管理模块。这两者都是硬件本身就提供支持的，但是笔记本电脑本身可能不需要这些功能，因此安装系统时如果启动这些机制就会造成一些错误，导致无法顺利安装。

如果笔记本电脑的显卡是集成的，就需要使用 nofb 来取消显卡的缓冲存储器检测，这样 Linux 安装程序就不会去检测集成显卡模块。

18.4 Bootloader

完成硬盘分区后接下来就来选择引导加载程序，在 Linux 里面主要有 LILO 与 GRUB 两种引导加载程序，不过目前 LILO 已经较少使用，取而代之的就是 GRUB。值得注意的是，引导加载程序可以被安装在 MBR 也可以安装在每个 partition 最前面的 super block 处。

如果安装在/dev/hda 内，那就是 MBR 的安装点，如果是类似/dev/hda1 这个就是 super block 的安装点。在 GRUB 中可以通过“新增”、“编辑”与“删除”来管理启动菜单要显示的项目。

举例来说，如果已经把 Microsoft Windows 安装在当前系统中，就可以通过“新增”项目操作将 Windows 启动分区加到当前启动菜单中，从而实现多系统引导启动。

多系统启动是有很多风险存在的，而且也不能随时更改多重操作系统的启动分区。

如前所述，GRUB 接下来就会去读取内核文件来进行硬件检测，并加载适当的硬件驱动程序，随后便开始启动操作系统的各项服务等。

GRUB 引导加载程序还提供密码管理机制，虽然这会影响到远程管理主机。另外，如果有特殊需求，可以把 GRUB 安装到每个分区的启动扇区（boot sector）。

多系统引导时要特别注意：

（1）Windows 的环境中最好将 Linux 的根目录与 swap 取消挂载，否则在 Windows 环境中可能会提示用户格式化这些分区。

（2）Linux 不能随意删除，开机时 GRUB 会去读取 Linux 根目录下的/boot 目录内容，一旦删除 Linux 会导致同时 Windows 也就无法启动了，因为此时整个开机启动菜单都会丢失。

多个硬盘就会有多个 MBR，这时就需要在 BIOS 中调整引导开机的默认设备顺序，只有第一个可引导开机设备内的 MBR 会被主动读取。

理论上是不能把 Windows 的引导加载程序安装到/dev/hda（或/dev/sda）而将 Linux 安装到/dev/hdb（或/dev/sdb）上，而是必须把引导加载程序安装到指定的第一个引导开机硬盘上。

由于 SATA 类型的设备文件名是利用检测到顺序来决定的，与 SATA 插槽无关，因此建议此时 BIOS 使用确定好的开机顺序，然后由 GRUB 来控制全部的开机菜单，否则对于 Linux 的运行会产生影响。

18.5 Network

如果网卡可以被系统检测到就可以设定网络参数，目前各大版本几乎都会默认网卡 IP 的获取方式为“自动取得 IP”，也就是所谓的“DHCP”网络协议。不过，由于这个协议需要有 DHCP 主机的辅助，搜索的过程中可能会等待一段时间，因此也可以改成手动设定。不过无论如何，都要与网络环境相同才是。

18.6 SELinux

操作系统的防火墙是一套可以设置允许或拒绝从其他机器上访问主机系统的服务，同时可以防止来自外界的，未经验证的系统对主机系统的访问。

另外 SELinux 的设定值得特别留意，SELinux 是 Security Enhanced Linux 的简写，由美国国家安全局 NSA（National Security Agency，http://www.nsa.gov/selinux/）开发。

现在 SELinux 被集成到 Linux 内核当中，它并不是防火墙，其主要功能是管理整个 Linux 系统的访问权限控制（access control），可以避免一些可能造成 Linux 操作系统安全问题（Security）的软件的破坏。

最早之前，SELinux 的开发是有鉴于系统经常被一般用户误操作而造成系统数据的安全性问题，因此加上这个模块来防止系统被终端用户不小心滥用系统资源。

SELinux 具有较好的系统防护能力，不过如果不熟悉，那么启用 SELinux 后系统服务可能会因为这个较为严密的安全机制而导致无法提供联网服务，或者无法进行数据存取。

18.7 kdump

当内核出现错误时，使用 kdump 可以将当时的内存内的信息写到文件中，从而帮助内核开发者改进内核。

18.8 Timezone

全世界被细分为 24 个时区，所以要告知系统具体时区，可以根据地区选择，也可以直接用鼠标在地图上选择。

要特别注意的是“UTC”，UTC 与所谓的“日光节约时间”有关。不过一般不需要选择这个，不然的话还可能造成时区被影响，导致系统显示的时间会与本地时间不同。

事实上，UTC 与 GMT 时间是一样的，GMT 就是格林威治时间——也是标准的地球时间，以格林威治（英国）所在地为 GMT 0 点，将地球细分为 24 个时区。

18.9 root

Linux 的系统管理员的默认名称为 root，root 的密码最好设定的严格一点，至少 8 个字符以上，并且含有特殊符号。

在 UNIX-like 操作系统中，root 的权限最大，除非必要否则不要使用 root 账户，一般都是创建一个一般身份账户来处理日常操作，只有当需要额外的 root 权限时才使用身份切换命令成为 root 来继续操作。

另外，所有安装的信息都会被记录在/root/install.log 及/root/anaconda-ks.cfg 这两个文件中。

可能在第一次安装完 Linux 后却发现无法启动的问题，也就是说确实安装好了 Linux distribution，但就是无法顺利启动，只要重新启动程序就会出现类似下面的画面：

```
GRUB> _
```

然后等待输入一些数据，如果发生了这样的问题，那么可能的原因有：

（1）主板 BIOS 太旧，导致识别不到大硬盘；

（2）主板不支持大硬盘。

这时首先需要更新 BIOS，将硬盘的 cylinders、heads、sectors 在 BIOS 内手动设定。更简单的解决方法就是重新安装 Linux 并且在分区时建立一个 100MB 左右的分区，然后将其挂载到/boot。要注意/boot 必须要在整个硬盘的最前面，例如，必须是/dev/hda1 才行。

至于会产生这个问题的原因确实是与 BIOS 支持的硬盘容量有关，处理方法虽然比较麻烦，不过也只能这样做了。

Chapter 19

Booting

在计算机中，CMOS 与 BIOS 是特别要注意的两个概念，其中 CMOS 是记录各项硬件参数且嵌入在主板上的只读存储器（ROM），BIOS 则是一个写入到主板中的固件（firmware，就是写入到硬件中的一个软件程序），BIOS 也就是在开机的时候计算机系统会主动执行的第一个程序。

接下来 BIOS 会去分析计算机里面有哪些存储设备，比如 BIOS 会依据用户的设置去搜索硬盘，并且到硬盘里面去读取第一个扇区的 MBR 位置。

MBR 这个仅有 446bytes 的硬盘容量里存放有最基本的引导加载程序，接下来的工作就由 MBR 内的引导加载程序来接着工作。引导加载程序的作用是加载（load）操作系统内核文件，而且由于引导加载程序是操作系统在安装时提供的，所以它会识别硬盘内的文件系统格式，于是就能够读取操作系统内核文件，然后接下来就是操作系统内核文件的工作了。

简单的说，整个开机流程到操作系统之前的过程应该是如下这样的：

1. BIOS：开机主动执行的固件，BIOS 会识别第一个可引导启动的设备。
2. MBR：第一个可引导开机设备的第一个扇区内的主引导分区，包含引导加载程序。
3. 引导加载程序（Boot Loader）：可读取内核文件来执行载入操作系统内核的软件。
4. 操作系统内核文件：逐步开始操作系统的各项功能。

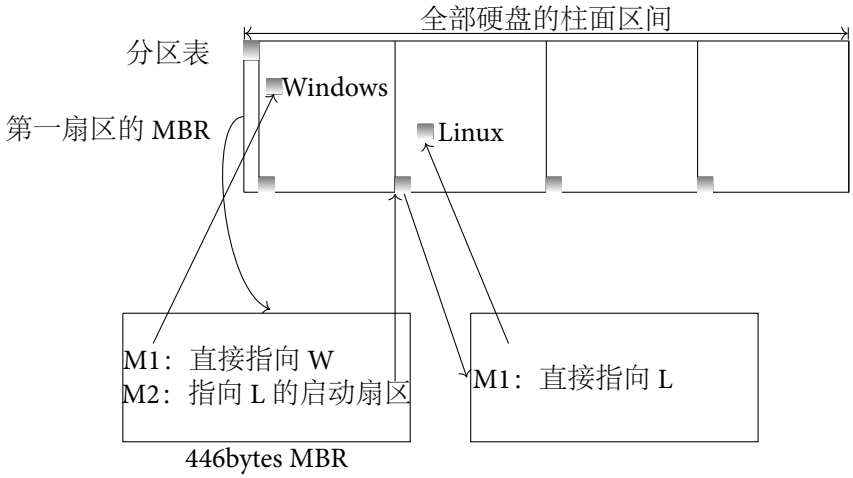
其中，BIOS 和 MBR 都是硬件本身会支持的功能，其中 Boot Loader 则是操作系统安装在 MBR 的软件，其主要任务包括如下：

- 选择开机菜单：用户可以根据不同的开机选项来实现多重引导。
- 加载操作系统内核文件：直接指向可引导启动的程序区段来载入操作系统。
- 转交控制权给其他 Boot Loader：将引导加载功能转交给其他 Booter Loader。

最后一项说明计算机系统中可以安装两个以上的引导加载程序，而且引导加载程序除了可以安装在 MBR 之外，还可以安装在每个分区的引导扇区（Boot Sector），这个特色才提供了“多重引导”的功能。

举例来说，假设计算机上安装了 Windows 及 Linux 操作系统，而且 MBR 内安装的是可

同时识别 Windows 和 Linux 操作系统的引导加载程序，那么整个流程可用下图表示。



在上图中，MBR 的引导加载程序提供两个菜单，菜单一（M1）可以直接加载 Windows 系统的内核文件来开机；菜单二（M2）则是将引导加载工作交给第二个分区的启动扇区（Boot Sector）。当用户在开机的时候选择菜单二时，整个引导加载工作就会交给第二分区的引导加载程序了。当第二个引导加载程序启动后，该引导加载程序内仅有一个开机菜单，因此就能够使用 Linux 的内核文件来开机，这就是多重引导的工作情况。

- 每个分区都拥有自己的启动扇区（Boot Sector）；
- 实际可开机的内核文件是可以放置到各分区内的；
- Boot Loader 只会认识自己的系统分区内的系统内核文件，以及其他 Boot Loader 而已；
- Boot Loader 可以直接指向或者间接将管理权转交给另一个启动加载程序。

Linux 系统在安装的时候可以选择将引导加载程序安装在 MBR 或个别分区的启动扇区，而且 Linux 的 Boot Loader 可以手动设置系统启动菜单，因此可以在 Linux 的 Boot Loader 中加入 Window 开机选项。而 Windows 在安装的时候，其安装程序会主动覆盖掉 MBR 以及自己所在分区的启动扇区，因此就没有选择的机会，这时可以使用 Linux 的系统恢复光盘来修复 MBR。

要注意区分引导加载程序与启动扇区的概念，现代计算机操作系统开机时需要有引导加载程序，而且引导加载程序可以安装在 MBR 及启动扇区等不同地方。

在 Linux 里面默认使用两种引导加载程序，分别是 LILO 与 GRUB，其中 LILO 是比较早期的引导加载程序，LILO 中的硬盘代号设置与 Linux 的硬盘代号相同。较新的 GRUB 最大的功能也最具魅力的地方是具有“动态搜索核心文件”的功能，GRUB 可以让用户在搜索的同时自行编辑启动设置文件。

硬盘分区与配置的好坏会影响到未来主机的使用情况，此外好一点的分区方式会让用户数据具有一定的安全性。除此之外硬盘分区的好坏还可以影响到系统存取数据的效率。

正常情况下的 Linux 主机通常会依照目录与主机的特性来进行硬盘分区。不过，Linux

的硬盘分区比较弹性，而且 Linux 的硬盘分区程序 fdisk 功能也比较强大，但是要说明的是，如果要分区合理，必须要理解基础的硬盘结构。

Chapter 20

Initialization

Linux 在运行的过程中会有很多的程序常驻在内存中，而且 Linux 使用非同步的硬盘/内存数据传输的模式，因此硬盘使用效率比较高，同时 Linux 提供的是多用户多任务的操作环境，由此对于 Linux 系统来说，不正常开关机有可能造成硬盘数据的损坏。

在 Linux 使用过程中，正确的开关机是很重要的，不正常的关机可能会导致整个系统的分区错乱并造成数据的损坏，这也是为什么通常 Linux 主机都会附加一个不断电系统的原因。

事实上，GRUB 的功能很多，其中就包含可以在系统发生错误的时候以额外的参数来强制开机并进行系统的修复等的功能。

此外，也可以以另一个开机管理程序 LILO 来设定 MBR 的开机菜单。不过在默认的情况下，Linux 并不会主动的安装 LILO。

一般来说，在操作 Linux 系统时，除非必要，否则不要使用 root 的权限，这是因为管理员（root）的权限太大了，所以建立一个一般身份用户来操作才是好习惯。举例来说，一般身份用户的帐号用来操作 Linux ，而当主机需要额外的 root 权限时，才使用身份切换命令来切换身份成为 root 来管理和维护。

> Tips:
> 为了让 X Window 的显示效果更好，很多团体开始开发桌面应用的环境，KDE/GNOME 就是其中的代表，这些团体的目标就是开发出类似 Windows 桌面的一整套可以工作的桌面环境，KDE 是构建在 X Window 上面的，可以进行视窗的定位、放大、缩小，同时还提供很多的桌面应用软件，详情可以参考http://www.kde.org/，GNOME 则是另外一个计划。

另外，Linux 是多用户多任务的操作系统，那么每个用户自然应该都会有自己的“工作目录”，这个目录是用户可以完全控制的，所以就称为“用户个人目录”，一般来说，主文件夹都在/home 下。

资源管理器在 GNOME 中其实称为“鹦鹉螺（Nautilus）”，而在 KDE 中则称为“征服家

(Konqueror)”。在使用资源管理器查看文件时，文件名开头为小数点“.”的是隐藏文件。

一般来说，用户是可以手动来直接修改 X Window 的设定，不过修改完成之后 X Window 并不会立即载入，必须要重新启动 X 才行。特别注意，不是重新开机，而是重新启动 X。

重新启动 X 最简单的方法就是在 X 的画面中直接按下 [Alt]+ [Ctrl] + [Backspace]，这样就可以直接重新启动 X，也就可以直接读入设定。

另外，如果 X Window 因为不明原因导致有点问题时也可以利用这个方法来重新启动 X。

20.1 startx

默认情况下，Linux 会提供 6 个 terminal 来让用户登录，可以使用 [Ctrl] + [Alt] + [F1] ～ [F6] 的组合键来进行切换，而且系统会将 [F1] ～ [F6] 定义为 tty1 ～ tty6 的操作界面环境，也就是说当按下 [crtl] + [Alt] + [F1] 这三个组合键时，就会进入到 tty1 的 terminal 界面中。同样的 [F2] 就是 tty2 ，如果要回到刚刚的 X Window，按下 [Ctrl] + [Alt] + [F7] 就可以。

某些 Linux distribution 会使用到 F8 这个终端界面做为桌面终端。

- Ctrl + Alt + F1～F6：命令行界面登录 tty1～ tty6 终端；
- Ctrl + Alt + F7 ： 图形界面桌面。

也就是说，如果是以命令行界面登录，那么可以有 tty1～ tty6 这 6 个命令行界面的终端可以使用，但是图形界面则没有任何东西。如果以图形界面登录，就可以使用图形界面跟命令行界面。

如果是以命令行界面启动 Linux，tty7 默认是空白的，可以直接通过 startx 命令“理论上”可以启动图形界面。

```
[root@linux ~]# startx
```

在 Linux 开机之后，可以通过“Run Level”进入 X Window 或者是纯命令行界面。

Linux 默认提供了 7 个 Run Level 给用户使用，如果需要 Linux 下次以纯文本环境（run level 3）启动，可以将默认启动的 X Window（run level 5）改为不启动（run level 3），这只要修改/etc/inittab 这个文件的内容就可以。

20.2 tty

如果使用命令行界面（其实是 run level 3）启动 Linux 主机，那么默认就是登录到 tty1 这个环境中。

```
linux login: root
Password:
[root@linux ~]# _
```

root 就是“系统管理员”，也就是“超级用户, Super User”。在 Linux 主机内，root 帐号代表的是“无穷的权力”，任何事都可以进行，因此使用这个帐号要小心。

注意，在输入密码的时候显示器上面不会显示任何字符，正确登录之后最左边的 root 显示的是“目前用户的帐号”，而之后接的 linux 则是“主机名称”，最右边的 ~ 则指的是“目前所在的目录”，那么那个 # 则是用户常常讲的“提示字符”。

~ 符号代表的是“用户的家目录”，它是个“变量”，例如 root 的家目录在/root，~ 就代表/root。

Linux 中默认 root 的提示字符为 # ，而一般身份用户的提示字符为 $，登录成功后显示的内容部分其实是来自于/etc/issue 文件。

在一般的 Linux 使用情况中，为了“系统与网络安全”的考虑，通常用户都希望不要以 root 身份来登录主机，这是因为系统管理员帐号 root 具有无穷大的权力，例如 root 可以删除任何一个文件或目录，因此一个称职的网络/系统管理人员通常都会具有两个帐号，平时以一般帐号来使用 Linux 主机的资源，有需要用到系统功能修改时才会转换身份成为 root。

退出 Linux 系统可以直接这样做：

```
[root@linux ~]# exit
```

注意，注销、退出系统和结束当前会话并不是关机，只是让当前账号离开系统而已。基本上 Linux 本身已经有相当多的工作在进行，而登录也仅是其中的一个“工作”而已，所以当某一个用户离开时，那么该工作就停止了，但此时 Linux 其他的工作是还是进行的。

其实用户都是通过“程序”来跟系统通信的，比如窗口或命令行都是一组或一个程序在负责用户所想要完成的命令。“命令行模式”就是指在登录 Linux 的时候得到的一个 Shell，Shell 提供用户一些工具，可以通过 Shell 来改变 kernel 的行为。其实整个命令下达的方式很简单，只要记得几个重要的概念就可以了。举例来说，可以这样下达命令：

```
[root@linux ~]# command [-options] parameter1 parameter2 ...
                  命令      选项       参数(1)    参数(2)
```

说明：

0、一行命令中第一个输入的绝对是“命令（command)”或“可执行文件”。

1、command 为命令的名称，例如变换路径的命令为 cd 等；

2、中刮号 [] 并不存在于实际的命令中，而加入参数设定时，通常为-号，例如-h；

有时候完整参数名称会输入--符号，例如--help；

3、parameter1 parameter2... 为依附在 option 后面的参数，或者是 command 的参数；

4、command, -options, parameter1.. 中间以空格来区分，不论空几格 Shell 都视为一个空格；

5、按下 [Enter] 按键后，该命令就立即执行。[Enter] 按键为 <CR> 字符（Command Run），代表著一行命令的开始启动。

6、命令太长的时候，可以使用\符号来跳出 [Enter] 符号，使命令连续到下一行。注意\后要立即接特殊字符才能转义。

其他：

a、Linux 是区分大小写的，比如 cd 与 CD 在 Linux 命令中意义并不同。

注意到上面的说明当中，“第一个被输入的数据绝对是命令或者是可执行的文件”。

例：以 ls 这个“命令”列出“/root”这个目录下的“所有隐藏档与相关的文件属性”，文件的属性的 option 为-al ，所以有：

```
[root@linux ~]# ls -al /root
[root@linux ~]# ls -al /root
```

上面这两个命令的下达方式是一模一样的执行结果。

Linux 是区分大小写的，在下达命令的时候千万要注意到命令是大写还是小写。

```
[root@linux ~]# date
[root@linux ~]# Date
[root@linux ~]# DATE
```

Linux 是支持多国语言的，若可能的话，显示器的信息是会以该支持语系来输出的，但是终端界面在默认的情况下无法支持以中文编码输出数据，此时需要将支持语系改为英文才能够显示出正确的信息。

```
[root@linux ~]#echo $LANG
en_US.UTF-8
[root@linux ~]#LANG=en
[root@linux ~]#LANGUAGE=en
[root@linux ~]#LC_ALL=en
[root@linux ~]#LC_CTYPE=en
[root@linux ~]#LC_TIME=en
```

注意一下，上面每一行命令都是用等号“=”连接并且等号两边没有空格，是连续输入的。这样一来就能够在“本次的登录”查看英文信息，但如果退出 Linux 后，刚刚下达的命令就没有用了。

20.2.1 date

如果想要在命令行界面显示目前的时间，可以直接在命令行模式输入 date：

```
[root@linux ~]# date
```

另外，date 还有其他相关功能，例如：

```
[root@linux ~]# date +%Y/%m/%d
[root@linux ~]# date +%H:%M
```

+%Y/%m/%d 就是 date 的一些参数，这同时也说明在某些特殊情况下，参数前面也会带有正号“+”等。

20.2.2 cal

如果想要列出目前这个月份的月历，可以直接使用 cal 命令：

```
[root@linux ~]# cal
```

基本上，cal（calendar）命令可以做的事情还有很多，可以显示整年的月历情况：

```
[root@linux ~]# cal 2012
```

也就是说，基本上 cal 的语法为：

```
[root@linux ~]# cal [[month] year]
```

所以如果想要知道 2012 年 7 月的月历，可以直接使用如下命令：

```
[root@linux ~]# cal 7 2005

[root@linux ~]# cal 13 2012
```

某些命令有特殊的参数存在，若输入错误的参数则该命令会有错误信息的提示，通过这个提示可以知道命令执行错误之处。

20.2.3 bc

如果想要使用计算器，可以使用 bc 命令。输入 bc 之后，显示出版本信息之后就进入到等待命令的阶段，如下所示：

```
[root@linux ~]# bc
```

事实上，用户是“进入到 bc 这个命令的工作环境”中了，下面用户输入的数据都是在 bc 程序当中在进行运算的动作，所以输入的数据当然就得要符合 bc 的要求。在基本的 bc 计算操作之前，先要理解 bc 可以执行哪些运算：

- + 加法
- - 减法
- * 乘法
- / 除法
- ^ 指数
- % 余数

这里怎么 10/100 会变成 0 呢？原来是 bc 默认仅输出整数，如果要输出小数点下位数，那么就必须要执行 scale=number，那个 number 就是小数点位数，例如：

```
[root@linux ~]# bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
scale=3
1/3
.333
340/2349
.144
quit
```

Tips：

就像执行 bc 会进入 bc 的软件功能一样，那怎么知道目前等待输入的地方是某个软件的功能还是 Shell 的可输入命令的环境下呢？

其实，在进入 Linux 的时候就会出现提示字符了，如果发现在等待输入的地方并非提示字符，那通常就是已经进入到某个软件的功能当中了。

命令行模式里有很多的功能键，这些按键可以辅助用户进行命令的编写与程序的中断。

20.2.4 dc

20.3 [tab]

[tab] 按键具有“命令补全”与“文件补齐”的功能，可以避免用户输入错误的命令或文件名称，而且 [tab] 按键在不同的地方输入有不一样的结果。

```
[root@linux ~]# ca[tab][tab] <==[tab]按键是紧接在 a 字母后面
```

上面的 [tab] 指的是“按下那个 tab 键”，不是要输入 ca[... 的意思。

```
cabextract           cache_restore        canberra-gtk-play    caribou
cacertdir_rehash    cairo-sphinx       cancel            caribou-preferences
cache_check          cal                  cancel.cups          case
cache_dump           caller               capinfos             cat
cache_metadata_size  callgrind_annotate   capsh                catchsegv
cachepic             callgrind_control    captoinfo            catman
cache_repair         canberra-boot        card_eventmgr
```

所有以 ca 为开头的命令都被显示出来了，那如你输入 ls -al /.bash，然后两个 [tab] 会出现什么呢？

```
[root@linux ~]# ls -al ~/.bash[tab][tab]
.bash_history  .bash_logout   .bash_profile  .bashrc
```

在该目录下所有以.bash 开头的文件名称都会被显示出来了。

用户按 [tab] 按键的地方如果是在 command（第一个输入的数据）后面时，它就代表“命令补全”，如果是接在第二个字以后就会变成“文件补齐”的功能。

- [*tab*] 接在命令的第一个字的后面，则为“命令补全”；
- [*tab*] 接在命令的第二个字以后时，则为“文件补齐”。

20.4 [ctrl]-c

在 Linux 下如果输入了错误的命令或参数，导致程序无法停止或循环执行，可以使用 [ctrl] 与 c 组合键，可以中断当前程序。

不过应该要注意的是，这个组合键是可以将正在运行中的命令中断，如果正在运行比较重要的命令，就不能使用这个组合按键了。

20.5 [ctrl]-d

[ctrl] 与 d 的组合键通常代表“键盘输入结束（End Of File，EOF 或 End Of Input）的意思。

另外，也可以用来取代 exit 的输入，例如可以直接按下 [ctrl]-d 直接离开命令行界面（相当于输入 exit)。

当输入了错误的命令或得到了错误的结果时，可以通过显示器上显示的错误信息来了解问题出在哪里，那就很容易知道如何处理这个错误信息。

例如，在执行 date 却打错成为 DATE 时，这个错误的信息是这样显示的：

```
[root@linux ~]# DATE
-bash: DATE: command not found
```

上面那个 bash: 表示的是用户的 Shell 名称，在构成计算机的“用户、用户界面、核心、硬件”的架构中，Shell 就是用户界面，在 Linux 中默认的用户界面是 bash Shell。

那么上面的例子就说明了，bash 有错误，具体什么错误呢？在这里 bash 告诉用户：

```
DATE: command not found
```

字面上的意思是说“找不到命令”，哪个命令呢？就是 DATE 这个命令，这里的错误信息意思是说系统上面可能没有 DATE 这个命令。

```
[root@linux ~]# cal 13 2012
cal: illegal month value: use 1-12
```

cal 警告用户 illegal month value: use 1-12，意思是说“不合法的月份值，应该使用 1-12 之间的数字”。

20.6 Documentation

一般情况下，与软件相关的文档保存在/usr/share/doc/中。

20.6.1 help

help command provides online information about available commands and the shell environment.

On UNIX, help is part of the Source Code Control System. and prints help information for the SCCS commands.

```
help [command]
```

In Bash, the builtin command help lists all Bash builtin commands if used without arguments. Otherwise, it prints a brief summary of a command. Its syntax is:

```
help [-dms] [pattern]
```

```
$ python
Python 2.7.5 (default, Nov  3 2014, 14:26:24)
[GCC 4.8.3 20140911 (Red Hat 4.8.3-7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> help()

Welcome to Python 2.7!  This is the online help utility.

If this is your first time using Python, you should definitely check out the tutorial on the Internet

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Pyth

To get a list of available modules, keywords, or topics, type "modules", "keywords", or "topics".  Ea

help> string
help> math
```

20.6.2 man

man 是 manual（操作说明）的简写，比如查询 date 命令的使用方法，可以使用 man date 来获得相应的说明。

```
[root@linux ~]# LANG="en"
[root@linux ~]# man date
DATE(1)             User Commands             DATE(1)
NAME
       date - print or set the system date and time
SYNOPSIS
       date [OPTION]... [+FORMAT]
       date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]
DESCRIPTION
       Display the current time in the given FORMAT,
       or set the system date.
       -d, --date=STRING
              display time described by STRING, not 'now'
```

```
       -f, --file=DATEFILE
              like --date once for each line of DATEFILE
       -ITIMESPEC, --iso-8601[=TIMESPEC]
              output  date/time  in ISO 8601 format.
              TIMESPEC='date' for date only,'hours','minutes',
              'or' seconds' for date and time to the indicated
              precision. --iso-8601  without TIMESPEC defaults
              to 'date'.
.................(略)....
AUTHOR
       Written by David MacKenzie.
REPORTING BUGS
       Report bugs to .
COPYRIGHT
       Copyright (c)2004 Free Software Foundation, Inc.
       This is free software; see the source for copying conditions.
       There is NO warranty; not even for MERCHANtabILITY or FITNESS
       FOR A PARTICULAR PURPOSE.
SEE ALSO
       The full documentation for date is maintained as a  Texinfo
       manual.If the info and date programs are properly installed
       at your site, the command
              info coreutils date
       should give you access to the complete manual.
date (coreutils) 5.2.1     May 2005         DATE(1)
```

上述的页称为 man page，可以在里面查询命令的用法与相关的参数说明。

在 man page 中的第一行可以看到的是：“DATE(1)”，其中 (1) 代表的是“一般用户可使用的命令”，可以帮助用户了解或者是直接查询相关的数据。

代号	代表内容
1	在 Shell 环境中可以执行的命令或可执行文件
2	系统内核可调用的函数与工具等
3	一些常用的函数（function）与函数库（library），大部分为 C 语言函数库（libc）
4	设备文件的说明，通常是在/dev 下的文件
5	配置文件或者是某些文件的格式
6	游戏（games）
7	惯例与协议等，例如 Linux 标准文件系统、网络协议、ASCII 码等的说明内容
8	系统管理员可用的管理命令
9	跟内核有关的文件

使用 man page 在查看某些数据时，就会知道该命令/文件所代表的基本意义是什么了。举例来说，如果下达了 man null 时，会出现的第一行是：“NULL(4)”，对照一下上面的数字意义，原来 null 是一个“配置文件”。

man page 的内容也分成好几个部分来说明，也就是上面 man date 那个表格内以 NAME 作为开始介绍，最后还有 SEE ALSO 来作为结束。

基本上 man page 大致分成下面这几个部分：

Table 20.1: MAN Page 组成

代号	备注
NAME	简短的命令、数据名称说明
SYNOPSIS	简短的命令执行语法 (syntax) 简介
DESCRIPTION	较为完整的说明
OPTIONS	针对 SYNOPSIS 部分列举的所有可用的参数说明
COMMANDS	当这个程序（软件）在执行时，可以在此程序（软件）中执行的命令
FILES	这个程序或数据所使用或参考或连接到的某些文件
SEE ALSO	可以参考的，跟这个命令或数据有相关的其他说明
EXAMPLE	一些可以参考的范例
BUGS	是否有相关的 bug

有时候除了这些外，还可能会看到 AUTHORS 与 COPYRIGHT 等，不过也有很多时候仅有 NAME 与 DESCRIPTION 等部分。通常在查询某个数据时，一定要查看 NAME，简单看一下这个数据的意思，再详看一下 DESCRIPTION，这个 DESCRIPTION 会提到很多相关

的数据与使用时机，从这个地方可以学到很多小细节。接下来主要就是查询关于 OPTIONS 的部分，从而可以知道每个参数的意思，这样就可以执行比较细部的命令内容。最后会再看一下跟这个数据有关的还有哪些信息可以使用的。举例来说，上面的 SEE ALSO 就告知用户还可以利用“info coreutils date”来进一步查阅信息，某些说明内容还会列举有关的文件（FILES 部分）来供用户参考，这些都是很有帮助的。

在 man page 中，如果要向下翻页的话，可以按下键盘的空格键，也可以使用 [Page Up] 与 [Page Down] 来翻页。同时，如果知道某些关键字，那么可以在任何时候输入“/word”来主动搜寻关键字。

当用户按下“/”之后，光标应该就会移动到显示器的最下面一行，并等待输入搜寻的字符串。此时输入 date 后，man page 就会开始搜寻跟 date 有关的字符串，并且移动到该区域。最后如果要离开 man page，直接按下“q”就可以离开 man page。

按键	备注
空格键	向下翻一页
[Page Down]	向下翻一页
[Page Up]	向上翻一页
[Home]	去到第一页
[End]	去到最后一页
/string	向“下”搜寻 string 这个字串
?string	向“上”搜寻 string 这个字串
n, N	利用/或？来查询字串时，可以用 n 来继续下一个查询（不论是/还是?），可以利用 N 来进行“反向”查询，反过来 N 和 n 可以互换。
q	结束这次的 man page

注意上面的按键是在 man page 的画面当中才能使用的，有趣的是搜索功能，用户可以往下或者是往上搜寻某个字串，例如要在 man page 内搜寻 vbird 这个字串，可以输入/vbird 或者是?vbird，只不过一个是往下而一个是往上来搜寻的。而要“重复搜索”某个字符串时，可以使用 n 或者是 N 即可。

至于这些 man page 的数据的存放位置，不同的 distribution 通常可能有点差异，不过通常是放在/usr/share/man 里，然而用户可以通过修改 man page 搜索路径，可以通过修改/etc/man.config（或 man.conf、manpath.conf）来完成，更多的关于 man 的信息可以使用“man man”来查询。

man 还可以查询特定命令/文件的 man page 说明文件，具体来说，当要使用某些特定的命令或者是修改某些特定的配置文件时，可以使用 man 命令在找到所需要的 man page。

下面的例子可以用来找到系统中与“man”有关的说明文件或者更多跟 man 较相关的信息。

```
[root@linux ~]# man -f man
man             (1)  - format and display the on-line manual pages
man             (7)  - macros to format man pages
man.conf [man]  (5)  - configuration data for man
```

使用-f参数可以取得更多的man的相关信息，而上面这个表格中也有提示了（数字）的内容，举例来说，第二行的“man (7)”表示有个man (7)的说明文件存在，也有个man (1)存在。对于上表当中的两个man，可以使用这样的命令将对应的文件找出来：

```
[root@linux ~]# man 1 man  <==这里是用 man(1) 的文件数据
[root@linux ~]# man 7 man  <==这里是用 man(7) 的文件数据
```

将上面两个命令执行之后就可以看到，两个命令输出的结果是不同的，这里1和7就是分别取出在man page里面关于1与7相关数据的文件。而至于搜索的文件类型是1还是7，就和查询的顺序有关。

查询的顺序是记录在/etc/man.conf这个配置文件中的，先查询到的说明文件会先被显示出来。一般来说，因为排序的关系通常会先找到数字较小的那个文件，所以man man会跟man 1 man结果相同。

除此之外，用户还可以利用“关键字”找到更多的说明文件，当我们使用“man -f命令”时，man只会找数据中的命令（或文件）的完整名称，有一点不同都不行，而使用“man -f关键字”命令，则只要含有关键字就会列出来，例如：

```
[root@linux ~]# man -k man
. [builtins]         (1)  - bash built-in commands, see bash(1)
alias [builtins]     (1)  - bash built-in commands, see bash(1)
........(中间省略)....
xsm                  (1x) - X Session Manager
zshall               (1)  - the Z Shell meta-man page
zshbuiltins          (1)  - zsh built-in commands
zshzle               (1)  - zsh command line editor
```

在系统的说明文件中，只要有man这个关键字就会将该manual列出来，这里就是利用关键字将说明文件里面只要含有man那个字符串（不见得是完整字串）的文件查找出来。

/etc/man.config文件规定了man page的查找路径等信息。例如，如果使用tarball方式来安装软件时，man page可能会被保存到/usr/local/softpackage/man/中，需要手动将相关的路径添加到/etc/man.config中。

/etc/man.config文件中通过指定MANPATH来说明man命令查找的路径。

在不同的Linux发行版中，/etc/man.config有不同的别名。

- CentOS：/etc/man.config
- SuSE：/etc/manpath.config
- Fedora：/etc/man_db.conf

20.6.3 info

在所有的 UNIX-like 系统中都可以利用 man 来查询命令或者是相关文件的用法，Linux 又额外提供了 info 命令。

基本上，info 与 man 都是用来查询命令的用法或者是文件的格式，但是与 man 一次就输出所有信息不同，info page 将文件数据拆成一个一个的段落，每个段落用自己的页面来撰写，并且在各个页面中还有类似网页的“超链接”导航来跳转到各个不同的页面中，因此 info 文件数据必须要以 info 写成的才会比较完整。

info page 的每个独立的页面也被称为一个节点（node），因此可以将 info page 看作是命令行模式的网页。

支持 info 命令的文件放置在/usr/share/info/目录中，例如 info 的说明文件是 info 格式，所以使用 info info 可以得到 info 的 info page。

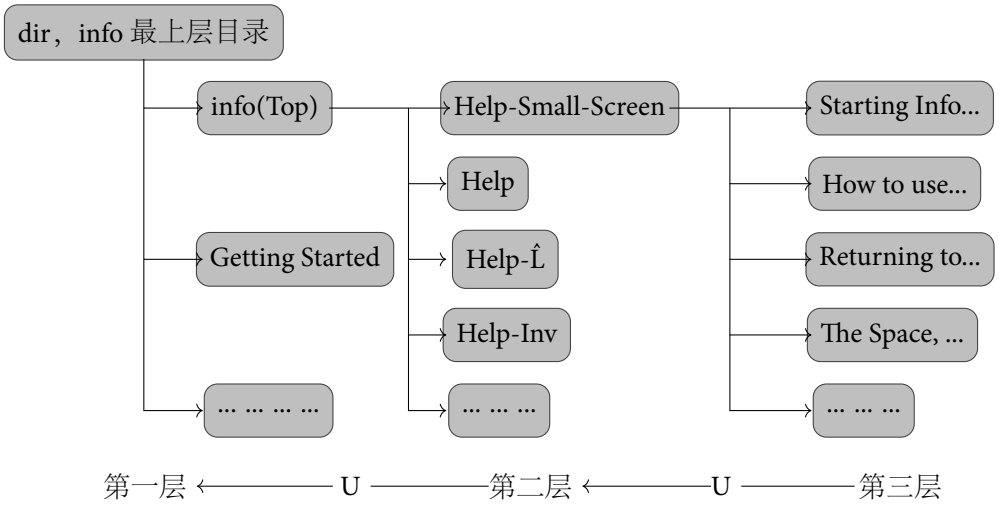
```
$ info info
File: info.info,  Node: Top,  Next: Getting Started,  Up: (dir)

Info: An Introduction
**********************

The GNU Project distributes most of its on-line manuals in the "Info
format", which you read using an "Info reader".  You are probably using
an Info reader to read this now.

   There are two primary Info readers: 'info', a stand-alone program
designed just to read Info files (*note What is Info?: (info-stnd)Top.),
and the 'info' package in GNU Emacs, a general-purpose editor.  At
present, only the Emacs reader supports using a mouse.

   If you are new to the Info reader and want to learn how to use it,
type the command 'h' now.  It brings you to a programmed instruction
sequence.

   To read about advanced Info commands, type 'n' twice.  This brings
```

```
you to 'Advanced Info Commands', skipping over the 'Getting Started'
chapter.

   This file describes how to use Info, the on-line, menu-driven GNU
documentation system.

   Copyright (C) 1989, 1992, 1996-2012 Free Software Foundation, Inc.

     Permission is granted to copy, distribute and/or modify this
     document under the terms of the GNU Free Documentation License,
     Version 1.3 or any later version published by the Free Software
     Foundation; with no Invariant Sections, with the Front-Cover texts
     being "A GNU Manual," and with the Back-Cover Texts as in (a)
     below.  A copy of the license is included in the section entitled
     "GNU Free Documentation License" in the Emacs manual.

     (a) The FSF's Back-Cover Text is: "You have the freedom to copy and
     modify this GNU manual.  Buying copies from the FSF supports it in
     developing GNU and promoting software freedom."

     This document is part of a collection distributed under the GNU
     Free Documentation License.  If you want to distribute this
     document separately from the collection, you can do so by adding a
     copy of the license to the document, as described in section 6 of
     the license.

* Menu:

* Getting Started::             Getting started using an Info reader.
* Advanced::                    Advanced Info commands.
* Expert Info::                 Info commands for experts.
* Index::                      An index of topics, commands, and variables.
```

info page 中的最后一行显示目前的 info 程序的版本信息，按下 m 按键就可以有更多的命令说明。

第一行则显示 info page 的文件名以显示数据来源，其中的 Node 显示当前这个画面是

“在第几层”（所属节点位置），因为 info page 将所有有关的数据都进行了连接，因此它可以利用分层的架构来说明每个文件数据，而且还有下一层数据，因此会看到第一行还有 Next 字符串，这表示只要输入“n”（Next）键后就可以跑到下一层，也就是 Getting Started 那个章节，输入“u”（Up）回到上一层，输入“p”（Pre）回到上一个节点。



用户可以将光标移动到该命令行或者 * 上面，按下 Enter 就可以前往该小节，利用 [tab] 键就可以快速的将光标在上表中的 node 间移动，这里 node 就是各个入口点。

info page 将内容分成多个节点，并且每个节点都有定位与链接。在各链接之间还可以具有类似“超链接”的快速按钮，可以通过 [tab] 键在各个超链接之间移动，也可以使用 u、p、n 字母键在各个分层与相关链接中显示。

按键	进行工作
空白键	向下翻一页
[Page Down]	向下翻一页
[Page Up]	向上翻一页
[tab]	在 node 之间移动，有 node 的地方，通常会以 * 显示。
[Enter]	当游标在 node 上面时，按下 Enter 可以进入该 node。
b	移动游标到该 info 画面当中的第一个 node 处
e	移动游标到该 info 画面当中的最后一个 node 处
n	前往下一个 info page 处
p	前往上一个 info page 处
u	向上移动一层
s(/)	在 info page 当中进行搜寻
h	显示求助菜单
?	命令一览表
q	结束这次的 info page

一般而言，命令或者开发者都会将命令或者软件的说明发布成“在线帮助文件”，但是还有相当多的说明需要额外的文件，即所谓的 How-To Documents。

另外，某些软件还会提供一些相关的原理说明，这些说明文件一般是存放在/usr/share/doc 中。例如，如果想要知道与当前版本的 Fedora 相关的各项信息，可以直接到/usr/share/doc/fedora-release-x 目录查看。如果想要知道 bash 是什么，则可以到/usr/share/doc/bash-x 目录中查阅。

/usr/share/doc 目录下的数据主要是以软件包（packages）为主的，例如 GCC 的相关信息在/usr/share/doc/gcc-xxx（xxx 表示版本的意思）。

20.6.4 whatis

事实上，还有两个命令与 man page 有关，而这两个命令是 man 的简略写法，分别是：

```
[root@linux ~]# whatis  [命令或者是数据]   <==相当于 man -f [命令或者是数据]
[root@linux ~]# apropos [命令或者是数据]   <==相当于 man -k [命令或者是数据]
```

20.6.5 makewhatis

为了能够使用 whatis 和 apropos 命令，必须要以 root 身份执行 makewhatis 命令来创建 whatis 数据库。

```
[root@linux ~]# makewhatis
```

20.6.6 apropos

apropos is a command to search the man page files in Unix and Unix-like operating systems.

Often a wrapper for the ”man -k” command, the apropos command is used to search all manual pages for the string specified. This is often useful if one knows the action that is desired, but does not remember the exact command

The following example demonstrates the output of the apropos command:

```
# apropos php
phar (1)             - PHAR (PHP archive) command line tool
phar.phar (1)        - PHAR (PHP archive) command line tool
php (1)              - PHP Command Line Interface 'CLI'
php-cgi (1)          - PHP Common Gateway Interface 'CGI' command
php-config (1)     - get information about PHP configuration and compile options
phpize (1)           - prepare a PHP extension for compiling
zts-php (1)          - PHP Command Line Interface 'CLI'
zts-php-config (1)  - get information about PHP configuration and compile options
```

```
zts-phpize (1)       - prepare a PHP extension for compiling
# apropos apropos
apropos (1)          - search the manual page names and descriptions
```

20.6.7 nano

nano 是一种单模式编辑器，用户可以直接输入文字。

如果需要禁用自动换行，可以使用-w 参数。

```
$ nano -w test.txt
```

- ^-G：取得在线帮助。
- ^-X：退出。
- ^-O：保存文件。
- ^-R：从其他文件读入数据。
- ^-W：查询字符串。
- ^-C：说明光标当前位置的行数和列数等信息。
- ^-_：直接输入行号并快速跳转到对应行。
- Alt-Y：打开或关闭语法检查。
- Alt-M：支持使用鼠标来移动光标。

20.7 shutdown

在 Linux 下，每个程序（或者说是服务）都是在后台执行的，因此在用户看不到的显示器背后其实可能有相当多人同时在主机上面工作，如果直接按下电源开关来关机，其他人的数据可能就此中断。

Linux 使用异步的磁盘/内存数据传输模式，若不正常关机则可能因为来不及将数据回写到文件中而造成文件系统的损坏，因此正常情况下要关机时需要注意下面几件事：

1、观察系统的使用状态：如果要看目前有谁在线上，可以执行 `who` 命令，而如果要看网络的连接状态，可以执行 `netstat -a` 命令，而要看后台执行的程序可以执行 `ps -aux` 命令。使用这些命令可以让用户稍微了解主机目前的使用状态，当然也就可以判断是否可以关机了。

2、通知在线用户关机的时间：要关机前总得给在线的用户一些时间来结束他们的工作，所以这个时候可以使用 `shutdown` 的命令来完成这个功能。

3、正确的关机命令使用 `shutdown` 与 `reboot` 命令。

Linux 系统的关机/重启是重大的系统操作，因此只有 root 账号才能够执行 shutdown/reboot 等命令，不过某些发行版（例如 CentOS）允许在主机的 tty7 使用图形界面登录时使用一般账号来关机或重启，其他一些发行版则在要关机时要求输入 root 密码。

20.7.1 who

who 命令用于显示登录系统的所有用户。

20.7.2 sync

在 Linux 系统中，为了加快数据的读取速度，在默认的情况下某些数据将不会直接被写入硬盘而是先暂存在内存当中，这样如果一个数据被用户重复的读写，那么由于它尚未被写入硬盘中，因此可以直接由内存当中读取出来，在速度上要快一些。

不过这也造成一些困扰，那就是当用户关机/重启或者是不正常的断电的情况下，数据尚未被写入硬盘中，所以就会造成数据的更新不正常。

sync 命令可以用来强制进行数据的写入动作，在内存中尚未被更新的数据就会被写入硬盘中，在系统关机或重启之前最好执行 sync。

sync 命令也只有 root 才可以执行，虽然目前的 shutdown/reboot/halt 等命令均已经在关机前调用了 sync 命令，不过仍然可以通过手动执行该命令来保护数据。

事实上，sync 也可以被一般账号使用，只不过一般账号用户所更新的硬盘数据仅是其自己的数据，而 root 账号使用 sync 可以更新整个系统中的数据。

```
# sync
```

shutdown 命令会结束系统内的各个进程（processes），并且将关闭 run level 内的服务。具体来说，shutdown 可以完成如下功能：

1、可以自由选择关机模式：是要关机、重新开机或进入单人操作模式均可；

2、可以设定关机时间：可以设定成现在立刻关机, 也可以设定某一个特定的时间才关机；

3、可以自定义关机信息：在关机之前，可以将自己设定的信息传送给在线用户；

4、可以仅发出警告信息：有时有可能要进行一些测试而不想让其他的用户干扰，或者是明白的告诉用户某段时间要注意一下，这个时候可以使用 shutdown 来通知用户但却不是真的要关机；

4、可以选择是否要使用 fsck 命令检查文件系统。

```
# /sbin/shutdown [-t 秒][-arkhncfF][时间][警告信息]
```

```
# /sbin/shutdown -h 10 'I will shutdown after 10 mins'
告诉大家，主机会在十分钟后关机，并且会显示在目前登录用户的显示器上。
```

```
-t sec    -t 后面加秒数，也就是“过几秒后关机”的意思
-k        不要真的关机，只是发送警告信息出去。
```

-r 在将系统的服务停掉之后就重新开机
-h 将系统的服务停掉后，立即关机。
-n 不经过init程序，直接以shutdown的功能来关机
-f 关机并开机之后，强制略过fsck的硬盘检查
-F 系统重新开机之后，强制进行fsck的硬盘检查
-c 取消已经在进行的shutdown命令内容。
时间 ： 这是一定要加入的参数，指定系统关机的时间。

此外需要注意的是，时间参数请务必加入，否则会自动跳到 run-level 1（就是单人维护的登录情况)。

```
[root@linux ~]# shutdown -h now
立即关机，其中 now 相当于时间为 0 的状态
[root@linux ~]# shutdown -h 20:25
系统在今天的 20:25 分会关机。若在21:25才执行此命令，则隔天才关机。
[root@linux ~]# shutdown -h +10
系统再过十分钟后自动关机
[root@linux ~]# shutdown -r now
系统立刻重启
[root@linux ~]# shutdown -r +30 'The system will reboot'
再过三十分钟系统会重新开机，并显示后面的信息给所有在线的用户。
[root@linux ~]# shutdown -k now 'This system will reboot'
仅发出警告信件的参数，系统并不会关机。
```

20.7.3 reboot

这三个命令差不多，它们调用的函数库都差不多，仅仅是用途上有些不同而已。

reboot 其实与 shutdown -r now 几乎相同。不过，建议在关机之前还是将数据同步的命令执行一次再执行关机命令。

```
[root@linux ~]# sync; sync; sync; reboot
```

20.7.4 halt

此外，halt 与 poweroff 也具有相同的功能。

20.7.5 poweroff

通常在重启的时候，都会执行如下的命令：

```
[root@linux ~]# sync; sync; sync; reboot
```

基本上，在默认的情况下，reboot、helt、poweroff、shutdown 这几个命令都可以完成一样的工作（因为 halt 会先调用 shutdown，而 shutdown 最后会调用 halt)。

shutdown 可以依据目前已启动的服务来逐次关闭各服务后才关机，halt 却能够在不理会目前系统状况下进行硬件关机的特殊功能。

```
[root@linux ~]# shutdown -h now
[root@linux ~]# poweroff -f
```

20.7.6 init

系统运作的模式分为命令行界面（run level 3）和图形模式界面（run level 5)。Linux 共有 7 种执行等级，其中：

- run level 0：关机；
- run level 3：纯命令行界面；
- run level 5：含有图形界面模式；
- run level 6：重启。

使用 init 命令切换执行等级，也就是说，如果要关机，除了执行上述的 shutdown -h now 以及 poweroff 之外，也可以使用如下的命令来关机：

```
[root@linux ~]# init 0
```

20.7.7 fsck

在开机的过程中最容易遇到的问题就是硬盘可能有坏道或分区错乱（数据损坏）的情况，这种情况虽然不容易发生在稳定的 Linux 系统下，不过由于不当的开关机还是可能会发生的，原因可能有：

最可能发生的原因是因为断电或不正常关机所导致的硬盘扇区错乱，硬盘使用率过高或主机所在环境不良也是一个可能的原因，例如用户开放了 FTP 服务，但是使用的又不是稳定的 SCSI 接口硬盘，仅使用 IDE/SATA 等接口的硬盘，虽然机率真的不高，但还是有可能造成扇区错乱的。另外，如果主机所在环境散热不良，或者是湿度相对较高，也很容易造成硬盘损坏。

解决的方法其实很简单，也可能很困难，这要取决于出错扇区所挂载的目录位置。如果根目录“/”并没有损坏，那就很容易解决，如果根目录已经损坏了，那就比较麻烦。

1、如果根目录没有损坏：假设发生错误的硬盘区块是在/dev/hda7，那么在开机的时候，系统自检信息应该会告诉用户：press root password or ctrl+D:

这时候输入 root 的密码登录系统，进行单用户的维护工作。

输入 fsck /dev/hda7（fsck 为命令，/dev/hda7 为错误的硬盘区块），然后依据实际情况设置相应的执行参数，这时会显示开始整理硬盘的信息，如果有发现任何的错误时，会显示：clear [Y/N] 的询问信息，直接输入 Y 即可。整理完成之后，以 reboot 重新开机。

2、如果根目录损坏了：此时可以将硬盘连接接到另一台 Linux 系统的计算机上，并且不要挂载（mount）该硬盘，然后以 root 的身份执行 fsck /dev/hdb1（/dev/hdb1 指的是硬盘设备名称，要根据实际状况来设定）。

另外，也可以使用 Live CD 启动主机进入 Linux 操作系统，再加载（mount）原本的/，以 fsck /dev/hda1 来修复根目录。

3、如果硬盘整个损毁：如果硬盘整体损毁，就需要尽可能先抢救硬盘内的数据并尽快更换硬盘。如果不愿意更换硬盘，那就只能重新安装 Linux，并且在重新安装的过程中，在格式化硬盘阶段选择“error check”，只是如此一来，format 会很慢，并且何时会再坏掉也不确定，所以最好还是更换一块新硬盘。

硬盘需要妥善保养，比如主机通电之后不要搬动，避免移动或震动硬盘，尽量降低硬盘的温度，可以加装风扇来冷却硬盘或者可以换装 SCSI/SSD 硬盘。

使用硬盘时要划分不同的分区，Linux 每个目录被读写的频率是不同的，因而 Linux 不同的安装模式就在于硬盘划分的不同，通常会建议用户划分成下列的硬盘分区：

```
/
/boot
/usr
/home
/var
```

如果/var 是系统默认的一些数据暂存或者是 cache 数据的存储目录，当这部分的硬盘损坏时，由于其他的地方是没问题的，从而数据得以保存，而且在解决起来也比较容易。

如果设定好了 Linux 之后忘记 root 密码，此时只要以单用户维护模式登录即可更改 root 密码。不过目前的多重引导程序主要有 LILO 与 GRUB 两种，这两种模式并不相同，有必要来说明一下。

1、LILO

只要在出现 LILO 选择菜单的时候输入：

```
boot: linux -s
```

注意，如果是 Red Hat 7.0 以后的版本，会出现图形界面的 LILO，这个时候要按下 [Ctrl] + x 即可进入纯命令行界面的 LILO。

用户进入单人单机维护模式（即为 run-level 1）后再输入 passwd 命令就可以直接更改 root 的密码了。同时，如果图形界面无法登录的时候，也可以使用该方法来进入单人单机的维护工作，然后再去修改/etc/inittab 更改一下登录的默认模式，这样就可以在下次开机的时候以命令行模式登录。同时要注意，如果在设定启动的名称的时候更改了启动的名称，就必须在 boot: 下面输入类似于下面的命令：

```
boot: Red-Hat-2.4.7linux -s
boot: Red-Hat-2.4.7linux single
```

2、GRUB

GRUB 作为多重引导程序，要进入单人维护模式就比较麻烦一些。在开机的过程当中会有读秒的时刻，此时按下任意按键就会进入选择菜单界面，然后只要选择相应的核心文件并且按下“e”就可以进入编辑界面了。此时看到的画面有点像：

```
root   (hd0,0)
kernel /boot/vmlinuz-2.4.19 ro root=LABEL=/ rhgb quiet
```

此时，需要将游标移动到 kernel 那一行，再按一次“e”进入 kernel 行的编辑界面中，然后在出现的画面当中输入 single：

```
root   (hd0,0)
kernel /boot/vmlinuz-2.4.19 ro root=LABEL=/ rhgb quiet single
```

再按下回车确定之后，按下“b”就可以开机进入单用户维护模式了，在这个模式下可以在 tty1 的地方不需要输入密码即可取得终端的控制权（而且是使用 root 的身份），这时就可以使用下面的命令修改 root 的密码。

```
[root@linux ~]# passwd
```

接下来系统会要求输入两次新的密码，然后再重启计算机。

Bibliography

[1] 阮一峰. 计算机是如何启动的？(2013).
URL http://www.ruanyifeng.com/blog/2013/02/booting.html.

[2] Wikipedia. 统一可扩展固件接口 (2005).
URL http://zh.wikipedia.org/zh-cn/%E7%B5%B1%E4%B8%80%E5%8F%AF%E5%BB%B6%E4%BC%B8%E9%9F%8C%E9%AB%94%E4%BB%8B%E9%9D%A2.

[3] Wikipedia. 主引导记录.
URL http://zh.wikipedia.org/wiki/%E4%B8%BB%E5%BC%95%E5%AF%BC%E8%AE%B0%E5%BD%95.

[4] Wikipedia. 全局唯一标识分区表.
URL http://zh.wikipedia.org/zh-cn/GUID%E7%A3%81%E7%A2%9F%E5%88%86%E5%89%B2%E8%A1%A8.

[5] 阿冬米博客. linux 启动过程中都发生了什么？(2012).
URL http://www.adonmi.com/linux/8.html.

[6] NATARAJAN, R. 6 stages of linux boot process (startup sequence) (2011).
URL http://www.thegeekstuff.com/2011/02/linux-boot-process/.

[7] 阮一峰. Linux 的启动流程 (2013).
URL http://www.ruanyifeng.com/blog/2013/08/linux_boot_process.html.

[8] Wikipedia. vmlinux.
URL http://zh.wikipedia.org/zh-cn/Vmlinux.

[9] ITtecman. 系统初始化过程 (2013).
URL http://www.cnblogs.com/nufangrensheng/archive/2013/12/17/3477783.html.

[10] Sheep. Linux 的启动流程留言 (2013).
URL http://www.ruanyifeng.com/blog/user/sheep.html.

[11] Wikipedia. 运行级别.
URL http://zh.wikipedia.org/zh/%E8%BF%90%E8%A1%8C%E7%BA%A7%E5%88%AB.

[12] Wikipedia. Runlevel.
URL http://en.wikipedia.org/wiki/Runlevel.

[13] 紫月冰河. Linux 查看系统和内核版本 (2012).
URL http://yansublog.sinaapp.com/2012/12/17/linux-%E6%9F%A5%E7%9C%8B%E7%B3%BB%E7%BB%9F%E5%92%8C%E5%86%85%E6%A0%B8%E7%89%88%E6%9C%AC-2/.

[14] Silva, R. Disk geometry (2005).
URL http://www.msexchange.org/articles-tutorials/exchange-server-2003/planning-architecture/Disk-Geometry.html.

[15] 丁明一. 首次安装 linux 之磁盘分区 (2012).
URL http://manual.blog.51cto.com/3300438/788705.

Part III

Filesystem

Chapter 21

Introduction

21.1 Overview

在计算机科学中，文件系统（file system）[2] 是一种在永久储存装置上管理数据的方式，绝大多数的磁盘文件系统都是为了针对如何让存储系统运作最佳化而设计的。

文件系统创建于存储设备之上，通常使用硬盘和光盘[1]等存储设备，并维护文件在设备中的物理位置。

通过向用户提供底层数据访问的文件系统机制，可以将设备中的空间划分为特定大小的块（扇区），一般每块 512 字节。

数据存储在块（扇区）中，并且文件系统会将大小被修正为占用整数个块。作为通用操作系统重要的组成部分，文件系统软件负责将这些块组织为文件和目录，并记录哪些块被分配给了哪个文件，以及哪些块没有被使用。

实际上，文件系统也可能仅仅是一种访问数据的接口而已，实际的数据是通过网络协议（如 NFS[2]、SMB、9P 等）提供的或者在内存上，或者可能根本没有对应的文件（例如 proc 文件系统）。

传统意义的操作系统在内核层面上对文件系统提供支持，但是通常内核态的代码难以调试，因此 Linux 从 2.6.14 版本开始通过 FUSE 模块支持在用户空间实现文件系统。

FUSE（Filesystem in Userspace）是操作系统中的概念，特指完全在用户态实现的文件系统。

目前，Linux、FreeBSD、NetBSD、OpenSolaris 和 Mac OSX 都支持用户空间态文件系统，其中 Linux 是通过内核模块对此进行支持，因此 FUSE 有时也特指 Linux 下的用户空间文件

[1] ISO 9660 和 UDF 文件系统被用于 CD、DVD 与蓝光光盘。

[2] 网络文件系统（NFS，Network File System）是一种将远程主机上的分区（目录）经网络挂载到本地系统的一种机制。

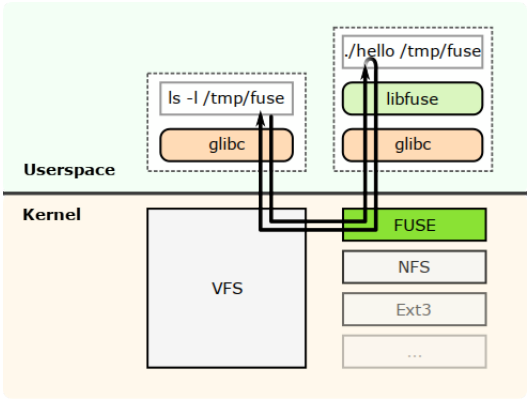


Figure 21.1: Linux 中 FUSE 的运行机制

系统。另外，诸如 ZFS[3]、GlusterFS[4]和 Lustre[5]也使用 FUSE 实现。

在分布式计算中，Hadoop 提供的分布式文件系统 HDFS 可以通过一系列命令访问，并不一定经过 Linux FUSE。

在用户空间实现文件系统能够大幅提高生产率，简化了为操作系统提供新的文件系统的工作量，特别适用于各种虚拟文件系统和网络文件系统，例如上述的 ZFS 和 glusterfs 都属于网络文件系统。但是，在用户态实现文件系统必然会引入额外的内核态/用户态切换带来的开销，对性能会产生一定影响。

文件系统是数据的组织者和提供者，因此文件系统并不一定只在特定存储设备上出现，其底层可以是磁盘，也可以是其它动态生成数据的设备（比如网络设备）。例如，文件管理方面的一个新概念是一种基于数据库的文件系统的概念，不再（或者不仅仅）使用分层结构管理，文件按照它们的特征（如文件类型、专题、作者或者亚数据）进行区分，因此文件检索就可以按照 SQL 风格或者自然语言风格进行。

21.1.1 Hardware

磁盘的物理组成包括盘片、机械臂和磁头以及主轴马达。

- 盘片主要用于记录数据；
- 机械臂和磁头用于读写盘片上的数据；
- 主轴马达转动盘片来让机械臂的磁头可以在盘片上读写数据。

磁盘数据存储与读写的关键在于盘片，盘片的物理组成包括扇区、柱面等。

- 扇区（sector）是最小的物理存储单位，每个扇区为 512bytes。
- 柱面（cylinder）是由扇区组成的同心圆，并且柱面是分区（partition）的最小单位。

[3]ZFS 可以认为是 Lustre 的 Linux 版本。
[4]GlusterFS 是用于集群的分布式文件系统，可以扩展到 PB 级。
[5]Sun 的 Lustre 是和 GlusterFS 类似但更早的一个集群文件系统。

硬盘的第一个扇区是最重要的，硬盘主引导记录（MBR）和分区表都存储在第一个扇区中。

- MBR 占用 446bytes；
- 分区表占用 54bytes。

21.1.2 Partition

不同接口的磁盘在操作系统中有不同的文件名。

- /dev/sd[a-p][1-15] 等文件名代表 SCSI、SATA、USB、Flash 等接口的磁盘；
- /dev/hd[a-d][1-63] 等文件名代表 IDE 接口的磁盘。

磁盘分区用于通知操作系统在访问磁盘时的可访问区域，起始点和结束点都是柱面号码。

在第一个扇区的分区表中存储了分区的范围，而且分区表只能记录 4 个分区的记录，因此分区表中的记录被称为主（primary）分区或扩展（extended）分区，从扩展分区中还可以再分出逻辑（logical）分区。

所有的分区中，可以被格式化为指定的文件系统的只有主分区和逻辑分区。

- 磁盘限制主分区和扩展分区最多可以有 4 个；
- 操作系统限制扩展分区最多只能有一个；
- 扩展分区无法格式化。

逻辑分区的数量受到操作系统的限制。例如，在 Linux 操作系统中的 IDE 硬盘最多有 59 个逻辑分区（5 号到 63 号），SATA 硬盘则有 11 个逻辑分区（5 号到 15 号）。

21.1.3 Format

操作系统在对磁盘分区进行格式化后才能对分区进行数据读写。

不同的操作系统对文件属性/权限的设置并不相同，磁盘分区格式化就是设置文件所需的数据，以便操作系统能够利用文件系统格式。

一般情况下，一个分区只能被格式化为一个文件系统，不过 LVM 与 RAID 等技术可以一个分区格式化为多个文件系统（例如 LVM），或者将多个分区合成一个文件系统（例如 LVM 和 RAID），因此现在可以认为是将可挂载的数据格式化为文件系统，不再是指分区。

文件系统中的数据除了文件实际内容之外，通常还包含其他属性，例如权限、用户、用户组、时间参数等。

文件系统通常将文件实际内容和其他属性分别存放在不同的块中，而且不同的块都有编号，其中：

- 权限与属性数据放置到 inode 中，一个文件占用 inode，并同时记录文件的数据所在的 block 号码；
- 实际数据则放置到数据块（data block）中，如果文件太大，则会占用多个 block；

- 超级块（super block）则记录整个文件系统的整体信息，例如 inode 和 block 的总量、使用量和剩余量等，以及文件系统的格式与相关信息等。

每个 inode 和 block 都有编号，每个文件都会占用一个 inode，在每个 inode 中保存着文件数据所在的 block 号码，因此通过文件的 inode 就可以从对应的 block 中读取文件的实际内容。

在格式化文件系统时，先格式化出 inode 和 block 的块，操作系统通过 inode 来对 block 进行读写，因此称为索引式文件系统（indexed allocation）。

21.1.4 Operation

在 von Neumann 系统结构的计算机中，所有的数据都要加载到内存后才能被 CPU 进行处理，Linux 使用异步处理（asynchronously）方式来提高文件系统的效率。

当系统加载某个文件到内存后，如果该文件没有被修改或使用，那么在内存区段的文件数据就会被设置为 clean。但是，如果内存中的文件数据被更改了，那么内存中的文件就会被设置为 dirty。

操作系统会不定时地将内存中被设置 dirty 的数据写回磁盘来保持磁盘与内存数据的一致性，用户也可以使用 sync 命令在手动强制将 dirty 数据写回磁盘。

- 操作系统将文件数据保存在主存储器的缓冲区来提高文件系统的读/写速度。
- 关机命令会主动调用 sync 命令来将内存数据写回磁盘。
- 不正常关机再重启后，操作系统会进行磁盘检验来确保文件系统的一致性。

21.1.5 Mount

文件系统使用 inode、block、super block 等信息确保文件系统能够链接到目录树中进行使用，将文件系统与目录树结合的操作称为挂载（mount）。

挂载点必须是目录，而且挂载点就是进入文件系统的入口，只有可以被挂载到目录树的某个目录的文件系统才能被使用。

默认情况下，Linux 操作系统中的挂载点包括/、/boot 和/home。

```
# ls -lid / /boot /home
2 drwxr-xr-x. 18 root root 4096 Dec 21 16:28 /
2 dr-xr-xr-x.  6 root root 3072 Dec 15 14:51 /boot
2 drwxr-xr-x.  4 root root 4096 Oct 29 20:10 /home
```

文件系统最顶层的目录的 inode 号码一般为 2，同一个文件系统的某个 inode 只能对应到一个文件，因此一个文件占用一个 inode。

```
# ls -lid . .. /
```

```
2 drwxr-xr-x. 18 root root 4096 Dec 21 16:28 /
2 drwxr-xr-x. 18 root root 4096 Dec 21 16:28 .
2 drwxr-xr-x. 18 root root 4096 Dec 21 16:28 ..
# ls -lid ../ ./ /
2 drwxr-xr-x. 18 root root 4096 Dec 21 16:28 /
2 drwxr-xr-x. 18 root root 4096 Dec 21 16:28 ./
2 drwxr-xr-x. 18 root root 4096 Dec 21 16:28 ../
```

由 1 号 inode 读取的连接文件的内容仅有文件名，并且通过 symbolic link 创建的文件也是独立的新的文件，同样会占用 inode 和 block。

21.2 Principle

作为一种存储和组织计算机数据的方法，文件系统的引入使得对数据的访问和查找变得容易，文件系统使用文件和目录树的抽象逻辑概念代替了硬盘和光盘等物理设备使用数据块的概念。

现在，用户使用文件系统来保存数据时不必关心数据实际保存在硬盘（或者光盘）的地址为多少的数据块上，只需要记住这个文件的所属目录和文件名。在写入新数据之前，用户不必关心硬盘上的哪个块地址没有被使用，硬盘上的存储空间管理（分配和释放）功能由文件系统自动完成，用户只需要记住数据被写入到了哪个文件中。

严格地说，文件系统是一套实现了数据的存储、分级组织、访问和获取等操作的抽象数据类型（Abstract data type）。

21.2.1 Filename

在文件系统中，文件名（filename）用于定位存储位置。

大多数的文件系统对文件名的长度有限制，而且文件名的大小写也有一定的限制。

大多现代文件系统允许文件名包含 Unicode 字符，不过仍然会限制某些特殊字符出现在文件名中，一般不建议在文件名中包含特殊字符。

文件系统可能会用这些特殊字符来表示一个设备、设备类型、目录前缀、或文件类型，不过会允许将这些特殊的字符存在于用双引号内的文件名。

21.2.2 Metadata

文件的元信息（metadata）常常伴随着文件自身保存在文件系统中。

- 文件长度可能是分配给这个文件的区块数，也可能是这个文件实际的字节数。
- 文件最后修改时间也可以记录在文件的时间戳中。

有的文件系统还保存文件的创建时间、最后访问时间及属性修改时间。

不过，大多数早期的文件系统不记录文件的时间信息，其它信息还可以包括文件设备类型（包括区块数、字符集、套接口和子目录等），文件所有者的 ID、组 ID 以及还有访问权限（例如只读、可执行等）。

21.2.3 Seurity

针对基本文件系统操作的安全访问可以通过访问控制列表（ACL）或功能（capabilities）实现，不过访问控制列表难以保证安全，这也是后来的文件系统倾向于使用 capabilities 的原因。

目前，多数商业性的文件系统仍然使用访问控制列表，可以直接或者间接地连接到计算机上的文件系统包括 FAT、exFAT、NTFS、HFS、HFS+、ext2、ext3、ext4、ODS-5 和 btrfs 等。另外，有些文件系统是进程文件系统（也称为日志文件系统）或者追踪文件系统。

闪存文件系统（Flash File System）是一种为了在闪存设备上存储数据而设计的文件系统，而且闪存设备跟磁盘存储设备在硬件上有不同的特性。

- 抹除区块（Erasing blocks）
  闪存的区块（block）在写入之前，要先做抹除（erase）的动作。抹除区块的时间可能会很长，因此最好利用系统闲置的时间来进行抹除。
- 耗损平均技术（Wear leveling）
  闪存的区块有抹写次数的限制，重复抹除、写入同一个单一区块将会造成读取速度变慢，直到损坏而无法使用，因此闪存设备的驱动程序需要将抹写的区块分散来延长闪存寿命。用于闪存的文件系统，也需要设计出平均写入各区块的功能。
- 随机存取（Random access）
  一般的硬盘在读写数据时，需要旋转磁盘来找到存放的扇区，因此一般会对用于磁盘的文件系统进行优化，以避免搜索磁盘的作用。但是闪存可以随机存取，没有查找延迟时间，因此不需要这个优化。

尽管磁盘文件系统（例如 FAT、NTFS、EXT4 和 btrfs）也能在闪存上使用，但闪存文件系统是闪存设备的首选。

- 擦除区块：闪存的区块在重新写入前必须先进行擦除，而且擦除区块会占用相当可观的时间，因此在设备空闲的时候擦除未使用的区块有助于提高速度。
- 随机访问：在磁盘上寻址有很大的延迟，因此磁盘文件系统有针对寻址的优化以尽量避免寻址，但闪存没有寻址延迟。
- 写入平衡（Wear levelling）：闪存中经常写入的区块往往容易损坏，因此闪存文件系统的设计需要使数据均匀地写到整个设备。

计算机上通行的大部份文件系统都是针对磁盘存储设备设计的，应用到闪存上并不适合。一般的文件系统可以通过闪存转换层（Flash Translation Layer，FTL）写入闪存，但是它

的缺点是写入的效率较差，因此设计闪存文件系统仍然是有必要的。

设计闪存文件系统的基本概念是，当存储数据需要更新时，文件系统将会把新的复本写入一个新的闪存区块，并将文件指针重新指向，并在闲置时期将原有的区块抹除，例如 JFFS2 和 YAFFS 等日志文件系统具有闪存文件系统需要的特性，而 exFAT 则是为了避免日志频繁写入而导致闪存寿命衰减的非日志文件系统。

在 1990 年代由微软研发的闪存文件系统 FFS2（Flash File System 2）被应用在 MS-DOS 上，后来 PCMCIA 组织通过了闪存转换层（Flash Translation Layer，FTL）的规格，允许 Linear Flash 设备能够看起来像是 FAT 磁盘设备，但是仍然保有耗损平均技术的能力。

在 Linux 上实现的闪存转换层被称为 MTD。MTD 是一个硬件的抽象层，能够让闪存设备看起来像是一种区块设备，因此能够将既有的文件系统（如 FAT、Ext、XFS 等）直接应用在闪存上。

Linux 支持多种文件系统，不过最通用的文件系统是 ext* 系列文件系统（ext2、ext3 和 ext4）、XFS、JFS、ReiserFS 和 btrfs 等。

对于没有 FTL（Flash Translation Layer）的闪存或内存设备（MTD），Linux 支持 UBIFS、JFFS2 和 YAFFS 等文件系统格式。

另外，SquashFS 是一种通用的压缩只读文件系统，可以用于磁带机等线性存储设备中。

为了了解当前 Linux 支持的文件系统，可以查看如下目录：

```
$ ls -l /lib/modules/$(uname -r)/kernel/fs
total 172
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 9p
drwxr-xr-x. 2 root root  4096 Nov 22 07:39 affs
drwxr-xr-x. 2 root root  4096 Nov 22 07:39 befs
-rw-r--r--. 1 root root 17831 Nov 22 08:00 binfmt_misc.ko
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 btrfs
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 cachefiles
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 ceph
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 cifs
drwxr-xr-x. 2 root root  4096 Nov 22 07:39 coda
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 cramfs
drwxr-xr-x. 2 root root  4096 Nov 22 07:39 dlm
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 ecryptfs
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 exofs
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 fat
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 fscache
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 fuse
drwxr-xr-x. 2 root root  4096 Nov 22 07:39 gfs2
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 hfs
```

```
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 hfsplus
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 isofs
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 jfs
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 lockd
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 minix
drwxr-xr-x. 2 root root  4096 Nov 22 07:39 ncpfs
drwxr-xr-x. 5 root root  4096 Dec  6 22:28 nfs
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 nfs_common
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 nfsd
drwxr-xr-x. 2 root root  4096 Nov 22 07:39 nilfs2
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 nls
drwxr-xr-x. 5 root root  4096 Dec  6 22:28 ocfs2
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 pstore
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 reiserfs
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 romfs
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 squashfs
drwxr-xr-x. 2 root root  4096 Nov 22 07:39 sysv
drwxr-xr-x. 2 root root  4096 Nov 22 07:39 ubifs
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 udf
drwxr-xr-x. 2 root root  4096 Nov 22 07:39 ufs
drwxr-xr-x. 2 root root  4096 Dec  6 22:28 xfs
```

如果需要了解当前已加载到内存中的文件系统，可以使用 cat 命令。

```
$ cat /proc/filesystems
nodev sysfs
nodev rootfs
nodev ramfs
nodev bdev
nodev proc
nodev cgroup
nodev cpuset
nodev tmpfs
nodev devtmpfs
nodev debugfs
nodev securityfs
nodev sockfs
nodev pipefs
nodev configfs
```

```
nodev devpts
   ext3
   ext2
   ext4
nodev hugetlbfs
nodev autofs
nodev pstore
nodev mqueue
nodev selinuxfs
nodev binfmt_misc
   fuseblk
nodev fuse
nodev fusectl
```

Chapter 22

Hierarchy

Linux 系统并不是特指某一个操作系统，而是指使用了由 Linus Torvalds 发明并领导开发的 Linux 内核的所有操作系统。

由不同的团队开发出来的基于 Linux 系统自然有很多地方是无法统一的，而且并且计算机配置与使用方法完全不统一，因此 Linux 基金会发布了 Linux 标准规范 LSB（Linux Standard Base）用以规范 Linux 的开发，其中就规定了 Linux 的文件系统层次结构标准（Filesystem Hierarchy Standard，缩写为 FHS）。

FHS[3] 定义了 Linux 操作系统中的主要目录及目录内容，例如在/（根目录）下的各个主要目录应该存放的主要文件内容，此外还专门定义了/usr 和/var 两个目录及其子目录的结构。在大多数情况下，FHS 是一个传统 BSD 文件系统层次结构的形式化与扩充。

多数 Linux 发行版遵从 FHS 标准并且声明其自身政策以维护 FHS 的要求，例如 Linux 系统采用的是树状存储结构，在 Linux 中所有文件与目录都是由/（根）开始的，不过绝大多数发行版并没有完全执行建议的标准。例如，GoboLinux 和 Syllable Server 使用了和 FHS 完全不同的文件系统层次组织方法。

在 Linux 发行版中包含一个`/sys`目录作为虚拟文件系统 (sysfs，类似于`/proc`，一个 procfs)，它存储且允许修改连接到系统的设备，然而许多传统 UNIX 和类 Unix 操作系统使用`/sys`作为内核代码树的符号链接。

当 FHS 建立之时，其他的 UNIX 和类 Unix 操作系统已经有了自己的标准，尤其是 hier(7) 文件系统布局描述自从第七版 Unix(于 1979 年) 发布以来已经存在，或是 SunOS filesystem(7)，和之后的 Solaris filesystem(5)。例如，Mac OS X 使用/Library、/Applications 和/Users 等长名与传统 UNIX 目录层次保持一致。

在 FHS 中，所有的文件和目录都出现在根目录 "/" 下，即使它们存储在不同的物理设备中。但是这些目录中的一些可能或可能不会在 Unix 系统上出现，这取决于系统是否含有某些子系统，例如 X Window 系统的安装与否。

这些目录中的绝大多数都在所有的 UNIX 操作系统中存在，并且一般都以大致类似的

方法使用。然而，这里的描述是针对于 FHS 的，并未考虑除了 Linux 平台以外的权威性。

Table 22.1: Linux Filesystem Hierarchy Standard

目录	描述
/	第一层次结构的根，整个文件系统层次结构的根目录，所有文件、文件夹的入口。
/bin/	需要在单用户模式可用的必要命令（面向所有用户的可执行文件），例如 cat、ls、cp。
/boot/	存放引导程序文件与内核，例如 kernel、initrd，通常是一个单独的分区
/dev/	设备目录，例如/dev/null、/dev/sda1[1]
/etc/[2]	特定主机，系统范围内的配置文件。
/etc/opt/	/opt/的配置文件
/etc/X11/	X Window 系统 (版本 11) 的配置文件
/etc/sgml/	SGML 的配置文件
/etc/xml/	XML 的配置文件
/home/	用户的家目录，包含保存的文件、个人设置等，一般为单独的分区。
/lib/	/bin/和/sbin/中二进制文件必要的库文件。
/media/	以前是挂接外部存储器的，现在是可移除媒体 (如 CD-ROM) 的挂载点。
/mnt/	临时挂载的文件系统或外接设备目录，例如移动硬盘、U 盘的内容在其子目录下存放
/opt/	可选应用软件包。
/proc/	虚拟文件系统，将内核与进程状态归档为文本文件。例如 uptime、network。在 Linux 中，对应 procfs 格式挂载。
/root/	超级用户的家目录
/sbin/	必要的系统二进制文件，同时也是管理员使用的命令，例如 init、ip、mount

[1]在 Linux 所有设备也都是以文件的形式出现的，打开/dev/sda1 就是打开了硬盘的第一个分区。

[2]关于这个名称目前有争议。在贝尔实验室关于 UNIX 实现文档的早期版本中，/etc 被称为 etcetera，这是由于过去此目录中存放所有不属于别处的所有东西（然而 FHS 限制/etc 存放静态配置文件，不能包含二进制文件）。自从早期文档出版以来，目录名称已被以各种方式重新称呼。最近的解释包括反向缩略语如："可编辑的文本配置"（Editable Text Configuration）或"扩展工具箱"（Extended Tool Chest)。

目录	描述
/srv/	站点的具体数据，由系统提供
/tmp/	临时文件 (参见/var/tmp)，在系统重启时目录中文件不会被保留
/usr/	用于存储只读用户数据的第二层次，包含绝大多数的 (多) 用户工具和应用程序
/usr/bin/	非必要可执行文件 (在单用户模式中不需要)，面向所有用户
/usr/include/	标准包含文件
/usr/lib/	/usr/bin/和/usr/sbin/中二进制文件的库
/usr/sbin/	非必要的系统二进制文件，例如大量网络服务的守护进程
/usr/share/	体系结构无关（共享）的数据
/usr/src/	源代码，例如内核源代码及其头文件
/usr/X11R6/	X Window R11, Release 6
/usr/local/	本地数据的第三层次，具体到特定主机。通常而言有进一步的子目录，例如 bin/、lib/、share/
/var/	变量文件——在正常运行的系统中其内容不断变化的文件，如日志、脱机文件和临时电子邮件文件，有时是一个单独的分区。
/var/cache/	应用程序缓存数据。这些数据是在本地生成的一个耗时的 I/O 或计算结果，应用程序必须能够再生或恢复数据，缓存的文件可以被删除而不导致数据丢失。
/var/lib/	状态信息。由程序在运行时维护的持久性数据。例如数据库、包装的系统元数据等
/var/lock/	锁文件，一类跟踪当前使用中资源的文件
/var/log/	日志文件，包含大量日志文件
/var/mail/	用户的电子邮箱
/var/run/	自最后一次启动以来运行中的系统的信息，例如当前登录的用户和运行中的守护进程。现已经被/run 代替
/var/spool/	等待处理的任务的脱机文件，例如打印队列和未读的邮件
/var/spool/mail/	用户的邮箱 (不鼓励的存储位置)
/var/tmp/	在系统重启过程中可以保留的临时文件
/run	代替/var/run 目录

开发一套文件系统层次结构标准的进程始于 1993 年 8 月，标准努力重整 Linux 的文件和目录结构。FSSTND (Filesystem Standard)，一个针对 Linux 操作系统的文件系统层次结构标准在 1994 年 2 月 14 日发布。后续的修正版本分别在 1994 年 10 月 9 日和 1995 年 3 月 28

日发布。

在 1996 年初，开发一个更加全面的、不仅解决 Linux，而且解决其他类 Unix 系统目录层次结构问题的 FSSTND 的计划在 BSD 开发社区成员的协助下正式被采纳。因此，计划重点解决在类 Unix 系统上普遍存在的问题。为了适应标准范围的扩充，标准的名称修改为文件系统层次结构标准。

FHS 现在由 Linux 基金会维护，这是一个由主要软件或硬件供应商组成的非营利组织，例如 HP、Red Hat、IBM、和 Dell。当前的 FHS 版本是 2.3，在 2004 年 1 月 29 日公布。

22.1 Overview

目前的操作系统大多数是将数据由硬盘读出来，那么每个操作系统使用的硬盘在 x86 架构上都是一样，但是每种操作系统都有其独特的读取文件的方法，也就是说，每种操作系统对硬盘读取的方法不同，所以就出现了不同的文件系统。

举例来说，Windows 98 默认的文件系统是 FAT（或 FAT16）文件系统，Windows 2000 有 NTFS 文件系统，Linux 的正统文件系统则为 ext2（Linux second extended file system，ext2fs）。

系统能不能读取某个文件系统，与“核心功能”有关，Linux 核心必须要能够认识某种文件系统才能读取该文件系统的数据内容，也就是说，必须要将所想要支持的文件系统编译到核心中才能被支持，因此可以发现 Windows 与 Linux 安装在同一个硬盘的不同 partition 时，Windows 将不能使用 Linux 的硬盘数据，就因为 Windows 的核心不认识 Linux 的文件系统。

目前 Fedora 默认的文件系统为 ext3（Third extended File System），它是 ext2 的升级版，主要是增加了日志（journaling）的功能，但是 ext3 还是向下支持 ext2 的。

另外，如果需要将原有的 Windows 系统也挂载在 Linux 下，Linux 同时也支持 MS-DOS、VFATFAT、BSD 等的文件系统。Window NT 的 NTFS 文件系统则不见得每一个 Linux distribution 都有支持。

Linux 能够支持的文件系统与核心是否有编译进去有关，所以可以到 Linux 系统的：

```
/lib/modules/‘uname -r‘/kernel/fs
```

查看，如果有想要的文件系统，那么这个核心就有支持。很多 Linux 所需要的功能都可以在 ext2 上面完成，不过 ext2 缺乏日志管理系统，发生问题时修复过程会比较慢一些，所以最新的 Linux distribution 大多已经默认采用 ext3 或 reiserfs 这种具有日志式管理的文件系统了。

ext3 其实只是多做了一个日志式数据的记录。当要在将数据写入硬盘时，ext2 是直接将数据写入，但是 ext3 则会将这个“要开始写入”的信息写入日志式记录区，然后才开始进行数据的写入。在数据写入完毕后，又将“完成写入动作”的信息写入日志记录区，这有什么好处呢？最大的好处就是数据的完整性与“可恢复性”。

早期的 ext2 文件系统如果发生类似断电后时，文件系统就得要检查文件一致性。这个检查的过程要将整个 partition 内的文件做一个完整的比较，比较耗时间。

如果是 ext3，那么只要通过检查“日志记录区”就可以知道断电时，是否有哪些文件正在进行写入的动作运行，只要检查这些地方即可，这样就能够节省很多文件检查的时间。

还可以引用 Red Hat 公司首席核心开发者 Michael K. Johnson 的话：

“为什么你想要从 ext2 转换到 ext3 呢？有四个主要的理由：可利用性、数据完整性、速度及易于转换”。

可利用性指出，这意味著从系统中止到快速重新复原而不是持续的让 e2fsck 执行长时间的修复。ext3 的日志式条件可以避免数据毁损的可能，它也指出，“除了写入若干数据超过一次时，ext3 往往会较快于 ext2，因为 ext3 的日志使硬盘读取头的移动能更有效的进行”，然而或许决定的因素还是在 Johnson 先生的第四个理由中。

“它是可以轻易的从 ext2 更改到 ext3 来获得一个强而有力的日志式文件系统而不需要重新做格式化”。“那是正确的，为了体验一下 ext3 的好处是不需要去做一种长时间的，冗长乏味的且易于产生错误的备份工运行及重新格式化的动作运行”。

上列数据可在 Whitepaper: Red Hat's New Journaling File System: ext3（http://www.redhat.com/support/wpapers/redhat/ext3/）查看得到，所以使用 ext3 或者是其它的日志式文件系统是有好处的，最大的好处当然是错误问题的排除效率比较高。

22.2 Virtual Filesystem

最初的 UNIX 系统一般都只支持一种单一类型的文件系统，这种情况下的文件系统的结构会深入到整个系统内核中。

Sun 在 1985 年开发的 SunOS 2.0 实现了第一个虚拟文件系统，之后被加入到 UNIX System V 第四版中，它让 UNIX 的系统调用可以适用于本地端的 UFS 以及远程的 NFS。

现在的操作系统大多都在系统内核和文件系统之间提供一个标准的接口，这样不同文件结构之间的数据可以十分方便地交换，因此 Linux 也在系统内核和文件系统之间提供了一种叫做 VFS（Virtual Filesystem Switch）的标准接口。

Linux 操作系统通过 VFS（Virtual Filesystem Switch）的核心功能去读取文件系统，也就是说，整个 Linux 认识的文件系统其实都是 VFS 在进行管理，用户并不需要知道每个分区上的文件系统是什么，VFS 会主动的帮用户做好读取的工作。

只要设定好主要文件系统对应的内核模块后，核心的 VFS 就会主动接管文件系统的存取工作，用户可以在不知道每个文件系统是什么的情况下就能自由的使用系统中的各种文件系统。

虚拟文件系统又称为虚拟文件切换系统（virtual filesystem switch），是操作系统的文件系统虚拟层，在其下是实体的文件系统。虚拟文件系统的主要用途在于让上层的软件能够用单一的方式来跟底层不同的文件系统沟通。

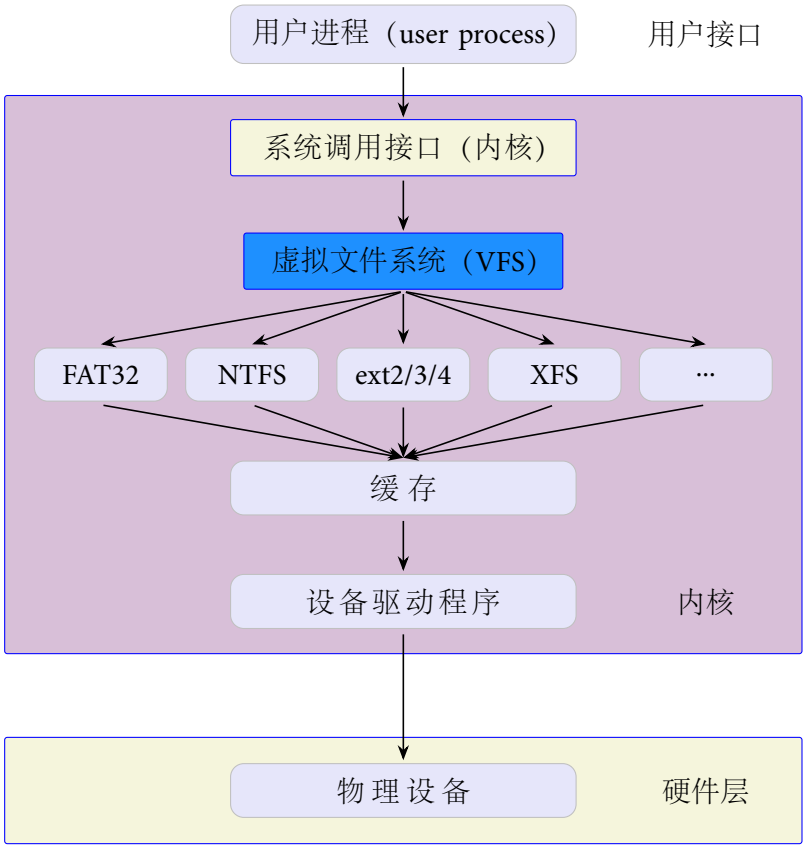


Figure 22.1: VFS 文件系统示意图

在操作系统与文件系统之间，虚拟文件系统提供了标准的操作接口，让操作系统能够很快的支持新的文件系统，因此文件系统的代码可以被划分为两部分。

- 上层用于处理系统内核的各种表格和数据结构；
- 下层用来实现文件系统本身的函数，并通过 VFS 来调用。

这些函数主要包括：

- 管理缓冲区 (buffer. c)；
- 响应系统调用 fcntl() 和 ioctl()(fcntl.c and ioctl.c)；
- 将管道和文件输入/输出映射到索引节点和缓冲区 (fifo.c, pipe.c)；
- 锁定和不锁定文件和记录 (locks.c)；
- 映射名字到索引节点 (namei.c, open.c)；
- 实现 select( ) 函数 (select . c)；
- 提供各种信息 (stat.c)；
- 挂接和卸载文件系统 (super.c)；
- 调用可执行代码和转存核心 (exec.c)；

- 装载各种二进制格式 (bin_fmt*.c)；

VFS 接口则由一系列相对高级的操作组成，这些操作由和文件系统无关的代码调用，并且由不同的文件系统执行。其中最主要的结构有 inode_operations 和 file_operations。file_system_type 是系统内核中指向真正文件系统的结构。每挂接一次文件系统，都将使用 file_system_type 组成的数组。file_system_type 组成的数组嵌入到了 fs/filesystems.c 中。相关文件系统的 read_super 函数负责填充 super_block 结构。

22.3 Filesystem Hierarchy

Linux 的所有文件与目录都是由根目录/开始的，/是所有目录与文件的源头，然后再一个一个的分支下来。

Linux 的目录配置方式也称目录树（directory tree），其主要的特性有：

- 目录树的启始点为根目录（/，root）；
- 每一个目录不止能使用本地端的文件系统，也可以使用网络上的文件系统。举例来说，可以利用 Network File System（NFS）服务器挂载某特定目录等。
- 每一个文件在此目录树中的文件名（包含完整路径）都是独一无二的。

目录配置标准（FHS，Filesystem Hierarchy Standard）的目的在于希望用户可以了解到已安装软件通常会放置于哪个目录下，并希望软件开发商、操作系统发行者以及想要维护系统的用户都能够遵循。

FHS 的重点在于规范每个特定的目录下应该放置什么样的数据，这样 Linux 操作系统就能够在既有的面貌（目录结构不变）下发展出开发者想要的风格。

事实上，FHS 仅是规范出在根目录（/）下各个主要的目录应该是要放置的文件，并且 FHS 根据文件系统使用的频繁与否以及是否允许用户随意改动而将目录定义成为四种互相作用的形态。

	可分享的（shareable）	不可分享的（unshareable）
不变的（static）	/usr（软件） /opt（第三方软件）	/etc（配置文件） /boot（开机与内核文件）
可变动的（variable）	/var/mail（用户邮件信箱） /var/spool/news（新闻组）	/var/run（程序相关） /var/lock（程序相关）

上表中的目录就是一些代表性的目录。

1、可分享的（shareable）：

可以分享给其他系统挂载使用的目录，包括执行文件与用户的邮件等数据，是能够分享给网络上其他主机挂载用的目录。

2、不可分享的（unshareable）：

本地机器上面运行的设备文件或者是与程序有关的 socket 文件等，因为这些文件仅与本机有关，因此不适合分享。

3、不变的（static）:

有些数据是不会经常变动的，不同的 distribution 都一样，例如函数库、文件说明文件、系统管理员所管理的主机服务配置文件等。

4、可变动的（variable）:

经常改变的数据，例如登录文件、新闻组等。

FHS 针对目录树架构仅定义出三层目录规范出来，第一层是/下的各个目录应该要放置什么样内容的文件数据，例如/etc 应该要放置设置文件，/bin 与/sbin 则应该要放置可执行文件等。然后则是针对/usr 及/var 这两个目录的子目录来进行定义，例如/var/log 放置系统登录文件，/usr/share 放置共享数据等。

- /（root，根目录）：与系统启动有关；
- /usr（UNIX software resource）：与软件安装/执行有关；
- /var（variable）：与系统运行过程有关。

root 在 Linux 里面的意义有很多种。

- 如果以“账号”的角度来看，所谓的 root 指的是“系统管理员”的身份，
- 如果以“目录”的角度来看，所谓的 root 意即指的是根目录，就是/。

在其它各子目录层级就可以随开发者自行来配置，比如 Fedora 的网络设置数据放在/etc/sysconfig/network-script/目录下，但是 SuSE Server 9 则是放置在/etc/sysconfig/network/目录下，这二者目录名称可是不同的。

特别注意下面这两个特殊的目录:

- . 代表当前的目录，也可以使用./ 来表示；
- .. 代表上一层目录，也可以使用../来代表。

这个. 与.. 目录概念是很重要的，常常会看到 cd .. 或./command 之类的命令执行方式，就是代表上一层与目前所在目录的运行状态。

此外，针对“文件名”与“完整文件名（由/开始写起的文件名）”的字符限制大小为:

- 单一文件或目录的最大允许文件名为 255 个字符；
- 包含完整路径名称及目录（/）的完整文件名为 4096 个字符。

/var/log/下有个文件名为 message，这个 message 文件的最大的文件名可达 255 个字符，var 与 log 这两个上层目录最长也分别可达 255 个字符，但总的来说，由/var/log/messages 这样完整文件名最长则可达 4096 个字符。

FHS 定义出两层目录内的规范，那么如果来到根目录查看目录数据，会看到:

```
#ls -l /
```

```
total 64
lrwxrwxrwx.   1 root root     7 Dec 12  2013 bin -> usr/bin
dr-xr-xr-x.   6 root root  3072 Dec 15 14:51 boot
drwxr-xr-x.  20 root root  3400 Dec 18 09:41 dev
drwxr-xr-x. 148 root root 12288 Dec 18 14:13 etc
drwxr-xr-x.   4 root root  4096 Oct 29 20:10 home
lrwxrwxrwx.   1 root root     7 Dec 12  2013 lib -> usr/lib
lrwxrwxrwx.   1 root root     9 Dec 12  2013 lib64 -> usr/lib64
drwx------.   2 root root 16384 Dec 12  2013 lost+found
drwxr-xr-x.   2 root root  4096 Aug  7  2013 media
drwxr-xr-x.   2 root root  4096 Aug  7  2013 mnt
drwxr-xr-x.   6 root root  4096 Nov 24 13:44 opt
dr-xr-xr-x. 234 root root     0 Dec 18  2014 proc
dr-xr-x---.  15 root root  4096 Dec 18 15:05 root
drwxr-xr-x.  38 root root  1020 Dec 18 15:04 run
lrwxrwxrwx.   1 root root     8 Dec 12  2013 sbin -> usr/sbin
drwxr-xr-x.   2 root root  4096 Aug  7  2013 srv
dr-xr-xr-x.  13 root root     0 Dec 18 09:41 sys
drwxrwxrwt.  19 root root   520 Dec 18 15:12 tmp
drwxr-xr-x.  13 root root  4096 Sep 27 09:58 usr
drwxr-xr-x.  22 root root  4096 Dec 18  2014 var
```

从属性的角度来看，上面的文件名每个都是“目录名称”，较为特殊的是 root，由于 root 这个目录是管理员 root 的主文件夹，这个主文件夹很重要，所以一定要设定成较为严密的 700（`rwx------`）这个属性。

Linux 中每个目录都是依附在/这个根目录下的，所以在安装的时候一定要有一个/对应的 partition 才能安装的原因即在于此，这也就是树状目录。

22.3.1 /

根据 FHS 定义出来的根目录（/）内应该要有下面这些子目录，如下表所示：

目录	应放置的文件内容
/bin	系统有很多放置可执行文件的目录，但/bin 放置的是在单用户维护模式下还能被运行的可执行文件，它们可被 root 与一般账号使用，包括 cat、chmod、chown、date、mv、mkdir、cp、bash 等
/boot	放置开机会用到的文件，包括内核文件、开机菜单与开机所需配置文件等。如果使用的是 grub 来引导加载，则还会有/boot/grub 这个目录。
/dev	Linux 系统上的任何设备与接口设备都会以文件的形式存在于/dev，访问该目录下的某个文件就等于访问某个设备。在此目录下的文件会多出两个属性，分别是 major device number 与 minor device number，系统核心就是通过这两个 number 来判断设备的。比较重要的文件有/dev/null、/dev/zero、/dev/tty*、/dev/ttyS*、/dev/lp*、/dev/hd*、/dev/sd* 等。
/etc	系统主要的配置文件几乎都放置在/etc，例如用户的账号密码文件、各种服务的起始文件等。/etc 下的文件属性是可以让一般用户查看的，但只有 root 有权力修改，并且在此目录下的文件几乎都是文本文件。比较重要的文件有/etc/inittab，/etc/init.d/，/etc/modprobe.d，/etc/X11，/etc/fstab，/etc/sysconfig/等。
/home	系统默认的用户主文件夹（home directory）。在新增一般用户账号时默认的用户主文件夹都会在这里创建。
/lib	放置的仅是在开机时会用到的函数库，以及在/bin 或/sbin 中的命令会调用的函数库，比较重要的是/lib/modules 这个目录内会存放 kernel 相关的模块（驱动程序）。
/media	放置的是可以删除的设备，常见的文件有/media/floppy、/media/cdrom。
/mnt	用于挂载其他额外的设备，在早期 Linux 中/mnt 与/media 的用途相同。
/opt	用于放置第三方软件，在早期 Linux 中则是放置在/usr/local 中。
/root	系统管理员（root）的主文件夹。进入单用户维护模式时而仅挂载根目录时，/root 与根目录（/）放置在同一个分区。
/sbin	放置的是开机过程中所需要的文件，包括开机、修复、还原系统所需要的命令，例如 fdisk、fsck、ifconfig、init、mke2fs、mkswap 等。与/bin 不太一样的地方是这几个目录是给 root 等系统管理用的，但是本目录下的可执行文件还是可以让一般用户用来“查看”但不能修改。至于某些服务器软件程序，一般放置在/usr/sbin 中，本机安装的软件所产生的系统执行文件（system binary）则放置在/usr/local/sbin 中。
/srv	放置的是一些服务启动之后所需要取用的数据。举例来说，WWW 服务器需要的网页数据就可以放置在/srv/www。
/tmp	这是让一般用户或者是正在执行的程序暂时放置文件的地方。/tmp 是任何人都能够存取的，所以需要定期进行清理。当然重要数据不可存放在该目录，FHS 建议开机时应将/tmp 下的数据都删除。

事实上，FHS 针对根目录所定义的标准就仅有上面列举的数据，不过现在实际上在 Linux 中还有几个重要的目录，如下表所示：

目录	应放置的文件内容
/lost+found	该目录是使用标准的 ext2/ext3 文件系统格式时才会产生的一个目录，目的在于当文件系统发生错误时将一些丢失的片段放置于此目录下，通常这个目录会自动出现在某个 partition 最顶层的目录下，例如加装一个硬盘于/disk 中，那在这个目录下就会自动产生一个这样的目录/disk/lost+found。
/proc	该目录本身是一个“虚拟文件系统（virtual filesystem）”，它放置的数据都是在内存当中，例如系统核心（kernel）、进程信息（processes）、周边设备的状态及网络状态等。因为这个目录下的数据都是在内存当中，所以本身不占任何硬盘空间，包括一些比较重要的文件，例如/proc/cpuinfo、/proc/dma、/proc/interrupts、/proc/ioports 等。
/sys	该目录其实与/proc 非常类似，也是一个虚拟文件系统，主要也是记录与内河相关的信息，包括目前已加载的内核模块与内核检测到的硬件设备信息等，/sys 同样不占硬盘容量。

根目录（/）是整个 Linux 系统最重要的一个目录，所有的目录都是由根目录衍生出来的，而且根目录也与开机、还原、系统修复等操作有关。由于系统开机时需要特定的启动加载程序、内核文件、开机所需程序、函数库等文件数据，若系统出现错误时，根目录也必须要包含有能够修复文件系统的程序。

FHS 希望根目录不要放在非常大的分区内，因为越大的分区会放入越多的数据，这会导致根目录所在分区可能有较多发生错误的机会。

FHS 建议根目录（/）所在分区应该越小越好，并且应用程序所安装的软件最好不要与根目录放在同一个分区内，保持根目录越小越好，这样不但保证性能，而且根目录的文件系统也较不容易发生问题。

除了/bin 之外，/usr/local/bin、/usr/bin 也是放置“用户可执行的 binary file 的目录”。举例来说，ls、mv、rm、mkdir、rmdir、gzip、tar、cat、cp、mount 等指令都放在这个目录中。

一般建议在根目录下只接目录，不要直接有文件在/下。根目录与开机有关，开机过程中仅有根目录会被加载，其他分区则是在开机完成之后才会陆续进行挂载的，因此根目录下与开机过程有关的目录不能放到与根目录不同的分区中，这些目录分别是：

- /etc：设置文件；
- /bin：重要执行文件；
- /dev：开机所需要的设备文件；
- /lib：开机所需要的可执行文件相关的函数库及内核所需的模块；
- /sbin：重要的系统执行文件。

/etc、/bin、/dev、/lib、/sbin 这五个目录都应该要与根目录在一起，不可独立成为某个 partition。

另外，/etc 下重要的目录有：

（1）/etc/init.d/：所有服务的默认启动脚本都是放在这里的，例如要启动或者关闭 iptables，可以运行下面的命令：

- /etc/init.d/iptables start
- /etc/init.d/iptables stop

（2）/etc/xinetd.d/：这就是 super daemon 管理的各项服务的配置文件目录。

（3）/etc/X11：与 X Window 有关的配置文件目录。

22.3.2 /boot

22.3.3 swap

22.3.4 /usr

根据 FHS 的基本定义，/usr 里面放置的数据属于可分享的（shareable）与不可变动的（static），而且/usr 还可以挂载网络文件系统。

/usr 事实上是“UNIX Software Resource”的缩写，在/usr 目录下包含系统的主要程序、图形介面所需要的文件、额外的函数库、本地所自行安装的软件以及共享的目录与文件等，都可以在这个目录当中发现。事实上，它有点像是 Windows 操作系统当中的“Program files”与“Windows”这两个目录的结合。

FHS 建议所有软件开发者应该将它们的数据合理地分别放置到这个目录下的子目录，而不要自行新建软件自己独立的目录。一般来说，/usr 下的重要子目录建议有如下这些：

/usr/bin	绝大部分的用户可使用的命令都放在这里。/usr/bin 与/bin 的不同之处在于后者与开机过程有关。
/usr/sbin	非系统正常运行所需要的系统命令，最常见的就是某些网络服务器软件的服务命令（daemon）。
/usr/include	C/C++ 等程序语言的头文件（header）与包含文件（include）放置处，当以 tarball 方式安装某些数据时会使用到里头的许多包含文件。
/usr/lib	应用软件的函数库、目标文件（object file）以及执行文件或脚本的放置目录，某些软件会提供一些特殊的命令来进行服务器的设置，这些命令也不会经常被系统管理员使用，那就会被放置到该目录下。如果使用的是 x86_64 系统，就可能会生成/usr/lib64 目录。
/usr/local	本地自行安装的软件默认放置的目录，目前也适用于/opt 目录。
/usr/share	共享文件放置的目录，例如下面/usr/share/doc 与/usr/share/man 这两个目录，其中前者放置一些系统说明文件的地方，例如安装了 grub 后可以在这里查到 grub 的说明文件。/usr/share/man 是 manpage 的文件文件目录，也就是使用 man 的时候会去查询的路径。
/usr/src	存放 Linux 系统相关代码的目录，例如/usr/src/linux 为核心源代码。
/usr/X11R6	系统内的 X Window System 重要数据所放置的目录。

在安装完 Linux 之后，基本上所有的配备都有了，但是软件总是可以升级的，例如要升级 proxy 服务，通常软件默认的安装地方就是在/usr/local（local 是“本地”的意思)。为了与系统原先的执行文件有区别，因此升级后的执行文件通常放在/usr/local/bin，为便于管理通常都会将后来才安装上去的软件放置在/usr/local。

另外，之所以命名为 X11R6，是因为最后的 X 版本为第 11 版，且该版的第 6 次发布的意思。

22.3.5 /var

/var 这个目录也很重要，也是 FHS 规范的第二层目录内容，如果/usr 是安装时会占用较大硬盘空间的目录，那么/var 就是在系统运行后才会逐渐占用硬盘的目录，/var 目录主要针对常态性变动的文件，包括缓存（cache)、登录文件（log file）以及某些软件运行所产生的文件，包括程序文件（lock file、run file）等。此外，某些软件执行过程中会写入的数据库文件，例如 MySQL 数据库也都写入在这个目录中。

/var 下的重要目录如下表所示：

/var/cache	程序文件在运行过程当中的一些暂存文件。
/var/lib	程序本身执行的过程中需要使用到的数据文件放置的目录，在此目录下各自的软件应该要有各自的目录，比如 MySQL 的数据库文件放置到/var/lib/mysql，而 rpm 的数据库则放到/var/lib/rpm 目录下。
/var/log	登录文件放置的目录，其中很重要的包括/var/log/messages、/var/log/wtmp（记录登录者的信息）等。
/var/lock	某些设备或文件一次只能被一个应用程序所使用，如果同时有两个程序使用该设备，就可能报错，因此就要将该设备上锁（lock）以确保该设备只会给单一软件使用。举例来说当刻录机正在刻录光盘时就会把刻录机上锁，只有当刻录机刻录完毕后其他用户就得要等到刻录机设备被解除锁定（也就是说前面的用户使用结束）才能继续使用。
/var/run	某些程序或者是服务启动后，会将它们的 PID 放置在这个目录下。
/var/spool	通常放置队列数据，所谓的“队列”就是排队等待其他程序使用的数据，这些数据被使用后通常会被删除。举例来说，主机收到新电子邮件后，就会放到/var/spool/mail 当中，若信件暂时发不出去就会放置到/var/spool/mqueue 目录下，如果是用户工作排程数据（crontab）则放置在/var/spool/cron 目录中。
/var/mail	放置个人电子邮件的目录，不过这个目录也被放置到/var/spool/mail 中，通常这两个目录是互为连接文件。

22.4 Partition/Directory

一般情况下，不会将所有的数据放置在一个目录中（就是只有一个“/”根目录），将数据分别放置在不同的目录中会比较安全。

- 系统数据通常放置在/usr 中；
- 个人数据则可能放置在/home 中；
- 配置信息则放置在/etc 中。

在升级系统时，/home 中保存的个人用户数据，在系统升级或者更改时不受影响，因此可以将系统进行如下的划分：

- /
- /boot
- /usr
- /home
- /var

根据 FHS 的定义，最好能够将/var 独立来提高系统数据的安全性。

对于 Linux 系统，只要根目录没有问题，那么就可以进入系统恢复模式进行相关的数据恢复工作。

22.5 Directory Tree

在文件系统中新建一个目录时就会分配一个 inode 与至少一个 block 给该目录，每个 block 的大小可以是 1024bytes、2048bytes 或 4096bytes。

- inode 记录目录的相关权限与属性，以及 block 的号码；
- block 记录目录下的文件名与文件名所使用的 inode 号码数据。

如果要查看目录内的文件所占用的 inode 号码，可以使用 `ls -li`。

```
$ ls -li
$ ll -i
```

如果要查看目录的 block 使用的 block 大小，可以使用 `ls -ld`，并且可以发现目录并不只会占用一个 block。如果目录下的文件数量太多时可以使用多个 block 来存储文件名和 inode 对照表。

```
$ ls -ld
$ ll -d
```

在文件系统中新建一个文件时会分配一个 inode 和对应该文件大小的 block 数量，其中 inode 仅有 12 个直接指向，因此在创建大文件时会使用多个 block 来记录 block 号码。

inode 本身并不记录文件名，文件名记录在目录的 block 中，因此新增/删除/重命名文件名等操作都与目录的 w 权限有关。

读写文件时都会读取目录的 inode 和 block，然后才能找到要读取的文件的 inode 号码，并最终得到正确的文件 block 号码来进行读写。

目录树从根目录开始，操作系统可以通过挂载的信息找到挂载点的 inode 号码（通常一个文件系统的最顶层 inode 号码从 2 开始），并依据根目录的 inode 内容读取根目录的 block 内的文件名数据，并依次逐层向下读取目录树。

```
$ ll -di /
2 drwxr-xr-x. 18 root root 4096 Dec 21 11:19 /
```

下面以读取/etc/passwd 为例来说明目录树和文件数据的读取。

1. /的 inode

   ```
   $ ll -di /
   2 drwxr-xr-x. 18 root root 4096 Dec 21 11:19 /
   ```

inode 具有的权限 `drwxr-xr-x` 允许普通用户读取该 block 的内容。

2. /的 block
   读取/的 block 并找到/etc 目录的 inode 号码。
3. /etc 的 inode
   ```
   $ ll -di /etc
   1835009 drwxr-xr-x. 150 root root 12288 Dec 21 11:18 /etc
   ```
4. /etc 的 block
   读取/etc 的 block 并获得 passwd 文件的 inode 号码。
5. passwd 的 inode
   ```
   $ ll -i /etc/passwd
   1842888 -rw-r--r--. 1 root root 2069 Dec 20 14:01 /etc/passwd
   ```
6. passwd 的 block
   从 passwd 所在的 inode 获取 block 所在的号码并进行读取。

下面是新建一个文件或目录时，目录树和文件数据的操作。

- 首先确定用户拥有对应目录的 w 与 x 的权限，否则不能执行。
- 根据 inode bitmap 找到未被使用的 inode 号码，并将新文件的权限/属性写入。
- 根据 block bitmap 找到未被使用的 block 号码，并将实际的数据写入 block 中，以及更新 inode 的 block 来执行数据。
- 将 inode 和 block 数据同步更新 inode bitmap 和 block 比特 map，并更新 super block 中的数据。

一般情况下，inode table 和 data block 称为数据存放区域，而 super block、block bitmap 与 inode bitmap 等则被称为 metadata（元数据）。

super block、inode bitmap 和 block bitmap 的数据是经常变动的，每次执行添加、删除和编辑时都可能影响到这三部分的数据。例如，新建的目录的连接数默认为 2，并且其上层的连接数会增加 1。

```
$ mkdir /tmp/testing
$ ls -lid /tmp/testing
1016204 drwxrwxr-x. 2 theqiong theqiong 40 Dec 21 22:09 testing/
$ ls -lid /tmp/testing/.
1016204 drwxrwxr-x. 2 theqiong theqiong 40 Dec 21 22:09 testing/.
$ ls -lid /tmp/testing/..
11576 drwxrwxrwt. 14 root root 320 Dec 21 22:10 testing/..
```

在新增数据时发生系统中断，那么写入的数据仅有 inode table 和 data block 等，同步更新 metadata 的步骤无法完成将导致 meta data 和实际数据存放区域不一致的问题，称为 inconsistent。

早期的 ext2 文件系统可以通过对 super block 中记录的 valid bit（是否有挂载）和文件系统的 state（是否 clean）等状态来判断是否需要使用 e2fsck 来强制进行数据一致性检查。

为了避免文件系统的不一致情况发生，在日志文件系统中规划出专门的块来记录写入或修订文件时的步骤，从而实现简化一致性检查的目的。

- 在写入文件时，首先在日志记录块中记录某个文件准备要写入的信息。
- 在写入阶段执行文件权限和数据的写入，并更新 meta data 的数据。
- 在数据写入结束并更新了 meta data 后，在日志记录块中写入文件的写入记录。

如果数据的记录过程中发生问题，可以通过检查日志记录块来找出发生问题的文件，并针对该问题进行一致性检查，从而可以不必对整个文件系统进行检查，同时也实现了快速修复文件系统的功能。

例如，在下面的 super block 中说明 8 号 inode 记录 joural 块的 block 指向，并且使用 8M 的日志来记录日志。

```
# dumpe2fs /dev/sda
Journal inode:            8
Default directory hash:   half_md4
Directory Hash Seed:      e2ba7e3d-02f9-4a8f-b7a0-3084e3fa0fb6
Journal backup:           inode blocks
Journal features:         journal_incompat_revoke
Journal size:             8M
Journal length:           8192
Journal sequence:         0x00000122
Journal start:            1
```

Chapter 23

Extended Filesystem

通过磁盘分区指定了硬盘分区所在的起始与结束柱面之后，接下来就是需要将分区格式化为“操作系统能识别的文件系统”，不同操作系统所设置的文件属性/权限并不相同，为了存放这些文件所需的数据就需要将分区进行格式化，之后操作系统才能使用这些分区。

每个操作系统可以识别的文件系统并不相同，例如 Windows 操作系统在默认状态下就无法识别 Linux 的文件系统（这里指 Linux 的标准文件系统 ext2），所以要针对操作系统来格式化分区。

可以说，每一个分区就是一个文件系统。传统的磁盘与文件系统的应用中，理论上一个分区是不可以具有两个文件系统的，每个文件系统都有其独特的支持方式，例如 Linux 的 ext3 就无法被 Windows 系统所读取。不过由于新技术的发展，比如 LVM 与 Software RAID，这些技术可以将一个分区格式化为多个文件系统（例如 LVM），也能够将多个分区合成一个文件系统（LVM，RAID），所以目前我们在格式化时已经不再说成针对分区来格式化了，通常是称呼一个可被挂载的数据为一个文件系统而不是一个分区。

不论是哪一种文件系统，数据总是需要存储的，既然硬盘是用来存储数据的，数据就必须写入硬盘，硬盘的最小存储单位是 Sector，不过数据所存储的最小单位并不是 Sector，因为用 Sector 来存储效率太低。一个 Sector 只有 512 Bytes，而磁头是一个一个 Sector 的读取，也就是说，假设文件有 10M Bytes，那么为了读这个文件，磁头必须要进行读取（I/O）20480 次。

为了克服这个效率上的不足，所以就有逻辑区块（Block）的产生了。逻辑区块是在分区后进行文件系统的格式化时所指定的“最小存储单位”，这个最小存储单位当然是建立在 Sector 的大小上（因为 Sector 为硬盘的最小物理存储单位），所以 Block 的大小为 Sector 的 2 的次方倍数。

完成文件系统格式化后，磁头一次就可以读取一个 block，假设格式化时指定 Block 为 4K Bytes（也就是由连续的 8 个 Sector 所构成一个 block），那么同样一个 10M Bytes 的文件，磁头要读取的次数则大幅降为 2560 次，这样就大大的增加文件的读取效率。

不过，Block 单位的规划并不是越大越好，因为一个 Block 最多仅能容纳一个文件（这里指 Linux 的 ext2 文件系统）。举例来说，假如 Block 规划为 4K Bytes，而现在有一个文件大小为 0.1K Bytes，这个小文件将占用掉一个 Block 的空间，也就是说该 Block 虽然可以容纳 4K Bytes 的容量，然而由于文件只占用了 0.1K Bytes，所以实际上剩下的 3.9K Bytes 是不能再被使用了，所以在考虑 Block 的规划时，需要同时考虑到：

- 文件读取的效率；
- 文件大小可能造成的硬盘空间浪费。

在规划硬盘时根据主机的用途来进行规划较好，比如 BBS 主机由于文章较短，也就是说文件较小，那么 Block 小一点的好；如果主机主要用在存储大容量的文件，考虑到效率当然 Block 理论上规划的大一点会比较妥当。

文件系统的运行与操作系统的文件数据有关。较新的操作系统的文件数据除了文件实际内容外，通常还含有非常多的属性，例如 Linux 操作系统的文件权限（r、w、x）与文件属性（所有者、用户组、时间参数等）。文件系统通常会将者两部分的数据分别存放在不同的块，权限与属性存放到 inode 中，实际数据则存放到 data block 块中。另外，还有一个超级块（super block）会记录整个文件系统的整体信息，包括 inode 与 block 的总量、使用量、剩余量等。

每个 inode 与 block 都有编号，inode、block、super block 的意义可以简略说明如下：

- super block：记录此文件系统的整体信息，包括 inode/block 的总量、使用量、剩余量以及文件系统的格式与相关信息等；
- inode：记录文件的属性，1 个文件占用 1 个 inode，同时记录此文件的数据所在的 block 号码；
- block：实际记录文件的内容，若文件太大，则会占用多个 block。

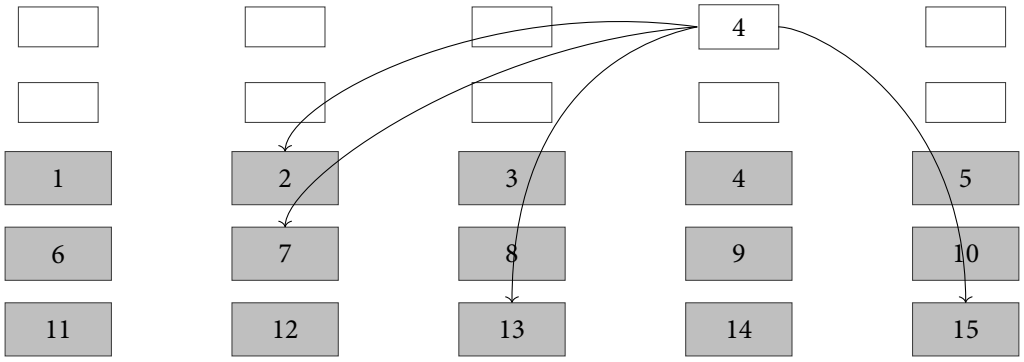
由于每个 inode 与 block 都有编号，而每个文件都会占用一个 inode，而 inode 内存有文件数据放置的 block 号码。找到文件的 inode 自然就会知道这个文件所放置数据的 block 号码，当然也就能够读出该文件的实际数据了，这样读写磁盘时通过 inode 和 block 就能在短时间内读取出全部的数据，提高了读写性能。

23.1 Super Block

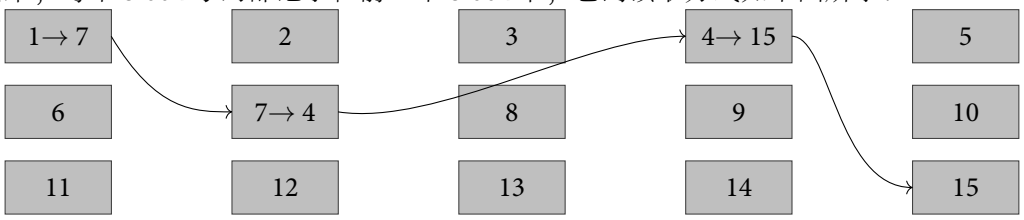
在进行硬盘分区时，每个硬盘分区就是一个文件系统，而每个文件系统开始位置的那个 block 就称为 super block。super block 用于存储文件系统的大小、空的和填满的区块，以及它们各自的总数和其他的信息等，也就是说，当要使用这一个硬盘分区（或者说是文件系统）来进行数据存取的时候，第一个要经过的就是 super block 这个区块，如果 super block 损坏，这个硬盘分区大概也就回天乏术。

23.2 Inode Block

下面使用示意图来说明 inode 与 block，文件系统先格式化出 inode 与 block 的块，假设某一个文件的属性与权限数据是放置到 inode 4 号，而这个 inode 记录了文件数据的实际存放点为 2、7、13、15 这 4 个 block 号码，于是操作系统就能够根据这些来排列磁盘的阅读顺序，从而可以很快将 4 个 block 内容读出来，于是数据的读取就如同下图中的箭头所指定的。



这种数据访问的方法称为索引式文件系统（indexed allocation）。而 U 盘等所使用的 FAT 文件格式并没有 inode 存在，因此在 FAT 文件系统中无法将文件的所有 block 在起始时就读取出来，每个 block 号码都记录在前一个 block 中，它的读取方式如下图所示：



在 FAT 文件系统中需要一个接一个地将 block 读出后才会知道下一个 block 的位置。如果同一个文件数据写入的 block 离散度很大，磁盘磁头将无法在磁盘转一圈就读取到所有的数据，此时磁盘就需要多转几圈才能完整地读取到这个文件的内容。

操作系统需要“碎片整理”的原因就是文件写入的 block 太过于分散了，此时文件读取的性能将会变得很差，通过碎片整理就可以把同一个文件所属的 block 汇合在一起，这样数据的读取会比较容易。

Linux 系统所使用的 ext2/3/4 文件系统是索引式文件系统，基本上不太需要经常进行碎片整理，但是如果文件系统使用太久，或经常删除/编辑/新增文件时就有可能会有文件数据过于离散的问题，就需要进行碎片整理。

23.3 EXT2

Linux 操作系统是一个多用户多任务的环境，为了要保护每个使用者所拥有数据的隐密性，所以具有多样化的文件属性是难免的。

23.3.1 Introduction

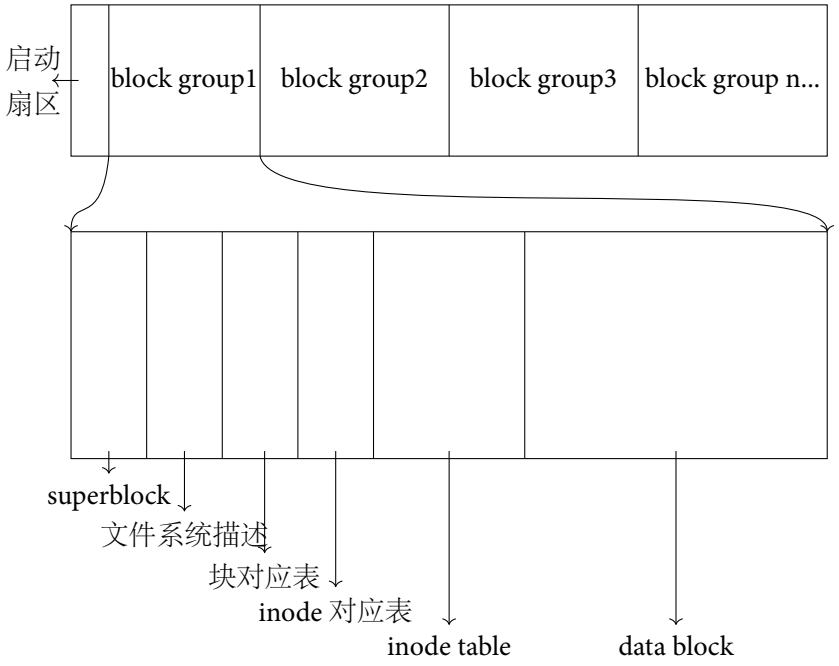
ext2 文件系统从 2.4.15 版本的内核开始引入，Stephen Tweedie 在 1999 年 2 月的内核邮件列表中最早显示了他使用扩展的 ext2。

Linux ext2 文件系统中每个文件除了文件的内容数据外，还包括文件的各种权限与属性(例如所属用户组、所属用户、能否执行、文件建立时间、文件特殊属性等)。

标准的 ext2 文件系统就是使用 inode 为基础的文件系统，其中会将每个文件的内容分为两个部分来存储，一个是文件的属性，另一个则是文件的内容。

为了满足这两个不同的需求，ext2 规划出 inode 与 block 来分别存储文件的属性（放在 inode 当中）与文件的内容（放置在 block area 当中），而且文件系统一开始就将 inode 与 block 规划好了，除非重新格式化（或者利用 resize2fs 等命令更改文件系统大小），否则 inode 与 block 固定后就不再变动。

当要将一个分区格式化为 ext2 时，就必须要指定 inode 与 block 的大小，也就是说，当分区被格式化为 ext2 文件系统时，一定会有 inode table 与 block area 这两个区域，因此 ext2 文件系统在格式化时基本上是区分为多个块组（block group）的，每个块组都有独立的 inode/block/superblock 系统。



在整体的规划中，文件系统最前面有一个启动扇区（boot sector），这个启动扇区可以安装引导加载程序，这样用户就能够将不同的引导加载程序安装到其他的文件系统的最前端而不用覆盖整块硬盘唯一的 MBR，这样才能够构造出多重引导的环境。

23.3.2 Data Block

data block 是用来放置文件内容的地方，ext2 文件系统所支持的 block 大小有 1 KB，2 KB 及 4 KB 三种。

在格式化文件系统时就已经确定了 data block 的大小，并且每个 block 都有编号以方便 inode 的记录。不过要注意的是，由于 block 的大小的区别会导致该文件系统能够支持的最大磁盘容量与最大单一文件容量不同，由 block 大小而产生的 ext2 文件系统的限制如下表所示：

block 大小	1 KB	2 KB	4 KB
最大单一文件限制	16GB	256GB	2TB
最大文件系统总容量	2TB	8TB	16TB

需要注意的是，虽然 ext2 已经能够支持大于 2GB 以上的单一文件容量，不过某些应用程序还是使用旧的限制，也就是说，这些程序只能够支持 2GB 的文件，这就跟文件系统无关了。

除此之外，ext2 文件系统的 block 的基本限制如下：

- 原则上，block 的大小与数量在格式化完成后就不能够再修改（除非重新格式化）；
- 每个 block 内最多只能放置一个文件的数据；
- 如果文件大于 block 的大小，则一个文件会占用多个 block；
- 如果文件小于 block 的大小，则该 block 的剩余空间无法再被利用。

每个 block 仅能容纳一个文件的数据，因此如果要存储的文件都非常小，而此时 block 在格式化时选用的是最大的 4 KB 时可能就会产生一些空间的浪费。

但是，如果指定的 block 较小，那么大型文件将会占用数量更多的 block，而 inode 也要记录更多的 block 号码，此时将可能导致文件系统读写性能下降，因此在进行文件系统的格式化之前，就要先想好该文件系统预计使用的情况。

23.3.3 Inode Table

block 是数据存储的最小单位，或者说 block 是记录“文件内容数据”的区域，inode 则是记录“该文件的相关属性以及文件内容放置在哪一个 block 之内”的信息。

简单的说，inode 除了记录文件的属性外，同时还必须要具有指向（pointer）的功能，也就是指向文件内容放置的区块之中，这样操作系统才可以正确的去取得文件的内容。

下面是 inode 记录的文件数据信息包括但不限于：

- 该文件的拥有者与群组（owner/group）；
- 该文件的存取模式（read/write/excute）；
- 该文件的类型（type）；

- 该文件创建或状态改变的时间（ctime）、最近一次的读取时间（atime）、最近修改的时间（mtime）；
- 该文件的大小；
- 定义文件特性的标志（flag），如 SetUID 等；
- 该文件真正内容的指向（pointer）；

利用 ls 查询文件所记录的时间，就是 atime/ctime/mtime 三种时间，这三种时间就是记录在 inode 里面的。利用 ls 的相关参数就可以取得想要知道的文件相关的三种时间，默认的显示时间是 mtime。

```
[root@linux ~]#ls -la -time=atime PATH
```

至于一个 inode 的大小为 128 bytes 这么大，可以使用 dumpe2fs 来查看 inode 的大小。

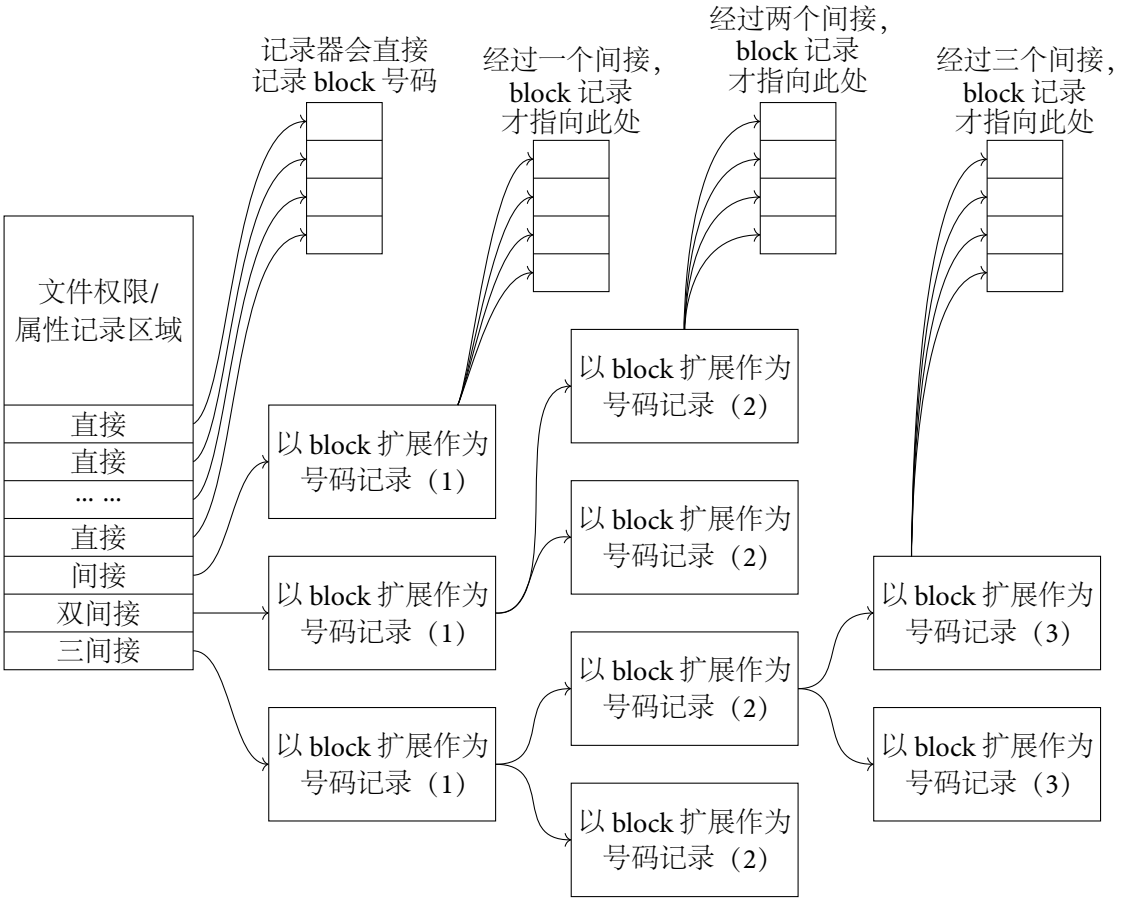
inode 的数量与大小也是在格式化时就已经固定了，除此之外，inode 还有下面的特性：

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

下面简要分析 inode/block 与文件大小的关系：

inode 要记录的数据非常多，但其本身只有 128Bytes，而 inode 要记录一个 block 号码要花掉 4Bytes，此时假设某文件有 40 MB 且每个 block 为 4 KB 时，那么就至少要 10 万条 block 号码的记录，这很不符合现实。

现代操作系统采用的是将 inode 记录 block 号码的区域定义为 12 个直接、1 个间接、1 个双间接与 1 个三间接记录区，如下图所示：



图中最左边是 inode 本身（128bytes），里面有 12 个直接指向 block 号码的对照，这 12 个记录就能够直接取得 block 号码，而间接就是再拿一个 block 来作为记录 block 号码的记录区，如果文件太大时就会使用间接的 block 来记录编号。如果文件持续增大，那么就会使用“双间接”，第一个 block 仅指出下一个记录编号的 block 的位置，实际记录的在第二个 block 中。以此类推，三间接就是利用第三层的 block 来记录编号。

下面以较小的 1KB 的 block 来说明 inode 能够指定的 block 的数目。

（1）12 个直接指向：12×1K=12K；

由于是直接指向，所以总共可以记录 12 条记录。

（2）间接：256×1K=256K；

每条 block 号码的记录会用去 4bytes，因此 1K 的大小能够记录 256 条记录。

（3）双间接：$256\times256\times1K=256^2K$；

第一层 block 会指定 256 个第二层，每个第二层可以指定 256 个号码。

（4）三间接：$256\times256\times256\times1K=256^3K$；

第一层 block 会指定 256 个第二层，每个第二层可以指定 256 个第三层，每个第三层可以指定 256 个号码。

（5）总额：将直接、间接、双间接加总，得到 12+256+256×256+256×256×256（K）=16GB

此时可知，当文件系统将 block 格式化为 1K 大小时，能够容纳的最大文件为 16GB，比较上面的文件系统限制表的结果可看出前后一致。但这个方法不能用在 2K 及 4K 的 block 大小的计算中，因为大于 2K 的 block 将会受到 ext2 文件系统本身的限制，所以计算结果会不太符合。

23.3.4 Superblock

superblock 是记录整个文件系统相关信息的地方，没有 superblock 也就没有这个文件系统了，superblock 记录的信息主要有：

- block 与 inode 的总量；
- 未使用与已使用的 inode/block 数量；
- 一个 block 与一个 inode 的大小；
- filesystem 的挂载时间、最近一次写入数据的时间、最近一次检验磁盘（fsck）的时间等文件系统的相关信息；
- 一个 valid bit 数值，若此文件系统已被挂载，则 valid bit 为 0 ，若未被挂载，则 valid bit 为 1。

如果 superblock 损坏，那么文件系统可能就需要花费很多时间来恢复。一般来说，superblock 的大小为 1024bytes，相关的 superblock 信息可以使用 dumpe2fs 命令来查看。

此外，每个 block group 都可能含有 superblock，但是实际上每个文件系统应该仅有 1 个 superblock。事实上除了第一个 block group 内会含有 superblock 之外，后续的 block group 不一定会含有 superblock，若含有 superblock 则该 superblock 主要是作为第一个 block group 内的 superblock 的备份，从而可以用于 superblock 的恢复。

23.3.5 File System Description

file system description（文件系统描述）区段可以描述每个 block group 的开始与结束的 block 号码，以及说明每个区段（superblock，bitmap，inodemap，data block）分别位于哪一个 block 号码之间，这些信息也可以使用 dumpe2fs 来查看。

23.3.6 Block Bitmap

在添加文件时就需要用到 block，通过 block bitmap（块对照表）的辅助就可以找到空的 block 来放置文件。同样的，当删除文件时原本被占用的 block 就会被释放出来，此时在 block bitmap 中对应到该 block 号码的标志就得要修改为“未使用”，上述这就是 block bitmap 的功能。

23.3.7 Inode Bitmap

inode bitmap（inode 对照表）与 block bitmap 功能类似，只是 block bitmap 记录的是使用的与未使用的 block 号码，而 inode bitmap 则是用来记录使用的与未使用的 inode 号码。

文件系统的各个块组的数据都与 block 号码有关，每个区段与 superblock 的信息都可以使用 dumpe2fs 命令来查询，查询的方法与结果如下：

```
[root@linux ~]#dumpe2fs [-bh] 设备文件名
参数:
-b: 列出保留为坏道的部分;
-h: 仅列出superblock的数据，不会列出其他的区段内容。

[root@linux ~]#dumpe2fs /dev/hda1
File system volume name:      /
File system state:      clean
Errors behavior:     Continue
File system OS type:    Linux
Inode count:      1537088
Block count:      1536207
Free blocks:      735609
Free inodes:      1393089
First block:      0
Block size:       4096
File system created:    Sat Jun 25 16:21:13 2005
Last mount time:     Sat Jul 16 23:45:04 2005
Last write time:     Sat Jul 16 23:45:04 2005
Last checked:     Sat Jun 25 16:21:13 2005
First inode:      11
Inode size:          128
Journal inode:       8

Group 0:  (Blocks 0-32767)
  Primary superblock at 0, Group descriptors at 1-1
  Reserved GDT blocks at 2-376
  Block bitmap at 377 (+377), Inode bitmap at 378 (+378)
  Inode table at 379-1400 (+379)
  0 free blocks, 32424 free inodes, 11 directories
  Free blocks:
  Free inodes: 281-32704
```

```
Group 1: (Blocks 32768-65535)
  Backup superblock at 32768, Group descriptors at 32769-32769
  Reserved GDT blocks at 32770-33144
  Block bitmap at 33145 (+377), Inode bitmap at 33146 (+378)
  Inode table at 33147-34168 (+379)
  18 free blocks, 24394 free inodes, 349 directories
  Free blocks: 37882-37886, 38263-38275
  Free inodes: 38084-38147, 39283-39343, 41135, 41141-65408
#因为数据很多，这里略去了一些信息，上面是比较精简的显示内容。
#在 Group 0 之前的都是 Superblock 的内容，记录了 inode/block 的总数，
#还有其他相关的信息。至于由 Group 0 之后，则是说明各个 bitmap 及 inode table
#与 block area 等。
```

通过 dumpe2fs 查询到的信息依内容可以区分为两部分，上半部分是 superblock 的内容，而下半部分则是每个 blockgroup 的信息。

通过这些记录可以知道哪些 inode 没有被使用，哪些 block 还可以记录，这样在新增、建立文件与目录时，系统就会根据这些记录来将数据分别写入尚未被使用的 inode 与 block area。

不过，要注意的是，当新增一个文件（目录）时：

- 根据 inode bitmap/block bitmap 的信息，找到尚未被使用的 inode 与 block，进而将文件的属性与数据分别记录进 inode 与 block;
- 将刚刚被利用的 inode 与 block 的号码（number）告知 superblock、inode bitmap、block bitmap 等，让这些 metadata 更新信息。

一般来说，将 inode table 与 block area 称为数据存储区域，而其他的例如 superblock、block bitmap 与 inode bitmap 等记录就被称为 metadata。经由上面两个动作就可以知道一次数据写入硬盘时会有这两个动作。

那么 Linux 系统到底是如何读取一个文件的内容呢？下面分别针对目录与文件来说明：

1、目录：

在 Linux 下的 ext2 文件系统建立一个目录时，ext2 会分配一个 inode 与至少一块 Block 给该目录。其中 inode 记录该目录的相关属性，并指向分配到的那块 Block，而 Block 则是记录在这个目录下的相关连的文件（或目录）的关连性。

2、文件：

当在 Linux 下的 ext2 建立一个一般文件时，ext2 会分配至少一个 inode 与相对于该文件大小的 Block 数量给该文件。例如假设一个 Block 为 4 Kbytes，而要建立一个 100 KBytes 的文件，那么 Linux 将分配一个 inode 与 25 个 Block 来存储该文件。

要注意的是，inode 本身并不记录文件名，而是记录文件的相关属性，至于文件名则是记录在目录所属的 block 区域，那么文件与目录的关系又是如何呢？

就如同上面的目录提到的，文件的相关连接会记录在目录的 block 数据区域，所以当要读取一个文件的内容时，Linux 会先由根目录/取得该文件的上层目录所在 inode，再由该目录所记录的文件关连性（在该目录所属的 block 区域）取得该文件的 inode，最后在经由 inode 内提供的 block 指向取得最终的文件内容。

以 /etc/crontab 这个文件的读取为例，内容数据是这样取得的：

一块 Partition 在 ext2 底下会被格式化为 inode table 与 block area 两个区域，所以在图三里面，我们将 Partition 以长条的方式来示意，会比较容易理解的。

而读取 /etc/crontab 的流程为

操作系统根据根目录 (/) 的相关数据可取得 /etc 这个目录所在的 inode ，并前往读取 /etc 这个目录的所有相关属性；

根据 /etc 的 inode 的数据，可以取得 /etc 这个目录底下所有文件的关连数据是放置在哪一个 Block 当中，并前往该 block 读取文件的关连性内容；

由上个步骤的 Block 当中，可以知道 crontab 这个文件的 inode 所在地，并前往该 inode ；

由上个步骤的 inode 当中，可以取得 crontab 这个文件的所有属性，并且可前往由 inode 所指向的 Block 区域，顺利的取得 crontab 的文件内容。

整个读取的流程大致上就是这样。

如果想要实作一下以了解整个流程的话，可以这样试做看看

1. 查看一下根目录所记录的所有文件关连性数据

```
[root@linux ~]#ls -lia /
      2 drwxr-xr-x  24 root root  4096 Jul 16 23:45 .
      2 drwxr-xr-x  24 root root  4096 Jul 16 23:45 ..
 719489 drwxr-xr-x  83 root root 12288 Jul 21 04:02 etc
 523265 drwxr-xr-x  24 root root  4096 Jun 25 20:16 var
#注意看一下，在上面的. 与.. 都是连结到 inode 号码为 2 的那个 inode ，
#也就是说，/ 与其上层目录.. 都是指向同一个 inode number，两者是相同的。
#而在根目录所记录的文件关连性（在 block 内）得到/etc 的 inode number
#为 719489 那个 inode number。
```

2. 查看一下 /etc/ 内的文件关连性的数据

```
[root@linux ~]#ls -liad /etc/crontab /etc/.
719489 drwxr-xr-x  83 root root 12288 Jul 21 04:02 /etc/.
723496 -rw-r--r--   1 root root   663 Jul  4 12:03 /etc/crontab
#此时就能够将/etc/crontab 找到关连性。
```

目录的最大功能就是在提供文件的关连性，在关连性里面，当然最主要的就是“文件名与 inode 的对应数据”。

另外，关于 ext2 文件系统，这里有几点要提醒一下：

1、ext2 与 ext3 文件在建立时（format）就已经设置好固定的 inode 数与 block 数目；

2、格式化 Linux 的 ext2 文件系统，可以使用 mke2fs 这个程序来执行；

3、ext2 允许的 block size 为 1024、2048 及 4096 bytes；

4、一个 Partition（File system）所能容许的最大文件数，与 inode 的数量有关，因为一个文件至少要占用一个 inode。

5、在目录底下的文件数如果太多而导致一个 Block 无法容纳的下所有的关连性数据时，Linux 会给予该目录多一个 Block 来继续记录关连数据；

6、通常 inode 数量的多少设置为（Partition 的容量）除以（一个 inode 预计想要控制的容量)。举例来说，若 block 规划为 4K bytes，假设一个 inode 会控制两个 block，也就是是假设一个文件大致的容量在 8Kbytes 左右时，假设这个 Partition 容量为 1GBytes，则 inode 数量共有：(1G * 1024M/G * 1024K/M) / (8K) = 131072 个。

而一个 inode 占用 128 bytes 的空间，因此格式化时就会有 131072 个 * 128bytes/个) = 16777216 byes = 16384 Kbytes 的 inode table 。也就是说，这一个 1GB 的 Partition 在还没有存储任何数据前，就已经少了 16MBytes 的容量。

因为一个 inode 只能记录一个文件的属性，所以 inode 数量比 block 多是没有意义的！举上面的例子来说，Block 规划为 4 Kbytes，所以 1GB 大概就有 262144 个 4Kbytes 的 block ，如果一个 block 对应一个 inode 的话，那么当 inode 数量大于 262144 时，多的 inode 将没有任何用处，只是浪费硬盘的空间而已。

另外一层想法，如果文件容量都很大，那么一个文件占用一个 inode 以及数个 block，当然 inode 数量就可以规划的少很多。

当 block 大小越小，而 inode 数量越多，则可利用的空间越多，但是大文件写入的效率较差；这种情况适合文件数量多，但是文件容量小的系统，例如 BBS 或者是新闻组（News Group）这方面服务的系统。

当 Block 大小越大，而 inode 数量越少时，大文件写入的效率较佳，但是可能浪费的硬盘空间较多；这种状况则比较适合文件容量较大的系统。

简单的归纳一下，ext2 有几个特色：

1、Blocks 与 inodes 在一开始格式化时（format）就已经固定了；

2、一个 Partition 能够容纳的文件数与 inode 有关；

3、一般来说，每 4Kbytes 的硬盘空间分配一个 inode；

4、一个 inode 的大小为 128 bytes；

5、Block 为固定大小，目前支持 1024/2048/4096 bytes 等；

6、Block 越大，则消耗的硬盘空间也越多。

关于单一文件：

若 block size=1024，最大容量为 16GB，若 block size=4096，容量最大为 2TB；

关于整个 Partition：

若 block size=1024，则容量达 2TB，若 block size=4096，则容量达 32TB。文件名最长达 255 字符，完整文件名长达 4096 字符。

另外，Partition 的使用效率上，当一个 Partition 规划的很大时，例如 100GB，由于硬盘上面的数据总是来来去去的，所以整个 Partition 上的文件通常无法连续写在一起，而是填入式的将数据填入没有被使用的 block 当中。如果文件写入的 block 真的分的很分散，此时就会有所谓的文件离散的问题发生了。虽然 ext2 在 inode 处已经将该文件所记录的 block number 都记上了，所以数据可以一次性读取，但是如果文件真的太过离散，确实还是会发生读取效率下降的问题，可以将整个 Partition 内的数据全部复制出来，将该 Partition 重新格式化，再将数据复制回去即可解决。

此外，如果 Partition 真的太大了，那么当一个文件分别记录在这个 Partition 的最前面与最后面的 block，此时会造成硬盘的机械臂移动幅度过大，也会造成数据读取效率的下降。因此 Partition 的规划并不是越大越好，而是真的要针对主机用途来进行规划才行。

Linux 支持多用户多任务环境，为了让各个用户具有较安全的文件管理机制，因此 Linux 文件权限管理将文件可存取访问的身份分为三个类别，分别是 owner/group/other，且各自具有 read/write/execute 等权限。

23.4 EXT3

23.4.1 Introduction

第三代扩展文件系统（Third extended filesystem，缩写为 ext3）是一个日志文件系统，

ext3 有一个相对较小的对于单个文件和整个文件系统的最大尺寸，这些限制依赖于文件系统的块大小（其中 8KiB 块只能用于允许 8KiB 页面的架构（例如 alpha））。

块尺寸	最大文件尺寸	最大文件系统尺寸
1KiB	16GiB	2TiB
2KiB	256GiB	8TiB
4KiB	2TiB	16TiB
8KiB	16TiB	32TiB

ext3 文件系统的性能（速度）低于 JFS2、ReiserFS 和 XFS，但是它支持从 ext2 文件系统升级时无需备份和恢复数据，而且它还具有比 ReiserFS 和 XFS 更低的的 CPU 使用率。

具体来说，ext3 文件系统增加的超越其前代 ext2 的包括：

- 日志
- 位目录跨越多个块提供基于树的目录索引
- 在线系统增长

ext2 和 ext3 文件系统共享相同的工具集（例如带有 fsck 工具的 e2fsprogs），因此在它们之间进行转换（包括升级到 ext3 和降级为 ext2）非常容易。

Linux 实现的 ext3 文件系统，包括 3 个级别的日志，分别是日志、顺序和回写。

- 日志（慢，但风险小）
  元数据和文件内容都在提交到主文件系统前写入，这样将提高稳定性但性能上有所损失，因为所有的数据都要写入 2 次。如果没有在/etc/fstab 中加上这个选项，修改中的文件在发生 kernel panic 或突然断电的时候就可能发生损毁的情况。
- 顺序（中速，中等风险）
  顺序和写回类似，但在对应的元数据标记为提交前，强制写入文件内容，这也是很多 Linux 发行版默认的方式。
- 回写（快，但风险最大，在某种感觉上和 ext2 相当）
  回写时写入日志的只有 metadata，文件的内容并不会跟着写入日志中，这样的作法提升了整体效率变，不过也同样造成了文件写入时不按顺序的结果。举例来说，文件在附加变大的同时发生了 crash 的情况，就可能造成下次挂载时文件后面附加垃圾数据的情况。

跟树状结构的文件系统相比，在 ext3 上面 metadata 存放在固定的位置，而且在写入的同时会重复写入的一些信息让 ext2/3 在面临数据损毁的情况下还有挽救的机会。

ext3 的设计目标是提供对 ext2 的高度兼容，很多磁盘上的结构和都和 ext2 很相似，也导致 ext3 缺乏很多最新设计中的功能（例如动态分配 inode 和可变块大小（frags 或 tails））。

ext3 文件系统在被挂载为写入时是不能进行 fsck 的，ext3 文件系统在处于挂载中时执行 dump 操作可能会造成数据的损坏。

ext3 不支持在其他文件系统上已经支持（例如 JFS2 和 ext4）的扩展。

ext3 在写入日志时并不做校验和。如果 barrier=1 没有作为加载参数（在文件/etc/fstab），并且如果硬件在无次序的写入缓存是发生崩溃就会严重损坏文件系统。

23.4.2 Defragmentation

在多任务环境下，磁盘碎片的出现根本是不可避免的，而且碎片化的速度非常之快。如果固态硬盘普及乃至取代传统硬盘，那么磁盘碎片的概念就会成为历史。

操作系统的任务不是不负责任地给用户提供一个整理工具，而是应该在系统设计的时候消除碎片化对性能的伤害。比如 Linux 的块设备操作都要经过一个 I/O 调度层，通过在调度层中使用带有电梯算法的调度策略来消除碎片对性能的影响。

在文件系统级别上，没有在线的 ext3 磁盘碎片整理工具。

离线的 ext2 磁盘碎片整理工具 e2defrag 可以用于 ext3 文件系统，前提是在使用前要将文件系统转换回 ext2，但是依赖于功能位在文件系统中打开，而且 e2defrag 可能会损毁数据。

23.4.3 Undelete

和 ext2 不同，ext3 会在删除文件时把文件的节点（inode）中的块指标清除。这样做可以在 unclean 载入文件系统后，重放日志时，可以减少对文件系统的访问。但也同样也增加了文件在反删除上面的困难。用户唯一的补救是在硬盘中捞取数据，并且要知道文件的起始到结束的块指标。尽管提供了比 ext2 在删除文件上稍微高一些的安全性，却也无可避免的带来了不便之处。

23.4.4 Compression

Ext3 不支持透明压缩（Ext2 以非官方补丁支持)。

23.5 EXT4

23.5.1 Introduction

2006 年 10 月 10 日发布了一个使用 ext4 作为名称的增强版本的文件系统，该文件系统包含很多新的功能。

作为 ext3 文件系统的后继版本，ext4（Fourth extended filesystem）可以支持最高 1 Exbibyte 的分区与最大 16 Tebibyte 的文件。

23.5.2 Extents

ext4 引进了 Extent 文件存储方式来取代 ext2/3 使用的 block mapping 方式。

Extent 指的是一连串的连续实体 block，可以增加大型文件的效率并减少分裂文件，ext4 支持的单一 Extent，在单一 block 为 4KB 的系统中最高可达 128MB。

- 单一 inode 中可存储 4 笔 Extent；
- 超过四笔的 Extent 会以 Htree 方式被索引。

23.5.3 Compatibility

ext4 向下兼容于 ext3 与 ext2，因此可以将 ext3 和 ext2 的文件系统挂载为 ext4 分区。

某些 ext4 的新功能可以直接运用在 ext3 和 ext2 上，直接挂载即可提升少许性能。

ext3 文件系统可以部分向上兼容于 ext4（也就是说 ext4 文件系统可以被挂载为 ext3 分区)，但是使用 Extent 技术的 ext4 将无法被挂载为 ext3。

23.5.4 Pre-allocation

目前大多数文件系统实现预留空间的方式是直接产生一个填满 0 的文件，ext4 允许对文件预先保留磁盘空间，而且 ext4 和 XFS 可以使用 Linux 核心中的“fallocate()”系统调用来获取足够的预留空间。

23.5.5 Delay

ext4 使用 allocate-on-flush 方式在数据将被写入磁盘（sync）前才开始获取空间，大多数文件系统会在之前便获取需要的空间，可以增加性能并减少文件分散程度。

23.5.6 Sundirectory

ext3 的一个目录下最多只能有 32000 个子目录，ext4 的子目录最高可达 64000，且使用“dir_nlink”功能后可以达到更高（虽然父目录的 link count 会停止增加）。

为了避免性能受到大量目录的影响，ext4 默认打开 Htree（一种特殊的 B 树）索引功能。

23.5.7 Journal

ext4 使用校验和特性来提高文件系统可靠性，而且可以通过安全地避免日志处理时磁盘 I/O 的等待来提高性能。

23.5.8 Defragmentation

即使 ext4 包含有许多避免磁盘碎片的技术，但是磁盘碎片还是难免会在一个长时间使用过的文件系统中存在。

23.5.9 Check

ext4 将未使用的区块标记在 inode 当中，这样可以使 e2fsck 等工具在磁盘检查时将这些区块完全跳过来节约大量的文件系统检查的时间，并且已经在 2.6.24 版本的 Linux 内核中支持快速文件系统检查。

Chapter 24

File Attribute

在以 root 身份登录 Linux 后，执行 `ls -al` 可以看到文件的相关权限与属性。

```
[root@linux ~]#ls -al
total 248
drwxr-x---   9 root  root   4096   Jul 11 14:58   .
drwxr-xr-x  24 root  root   4096   Jul  9 17:25   ..
-rw--------   1 root  root   1491   Jun 25 08:53   anaconda-ks.cfg
-rw--------   1 root  root   13823  Jul 10 23:12   .bash_history
-rw-r--r--   1 root  root   24       Dec  4  2004   .bash_logout
-rw-r--r--   1 root  root   191    Dec  4  2004   .bash_profile
-rw-r--r--   1 root  root   395    Jul  4 11:45   .bashrc
-rw-r--r--   1 root  root   100    Dec  4  2004   .cshrc
drwx------   3 root  root   4096   Jun 25 08:35   .ssh
-rw-r--r--   1 root  root   68495  Jun 25 08:53   install.log
-rw-r--r--   1 root  root   5976   Jun 25 08:53   install.log.syslog
[ 1 ]    [ 2 ][ 3 ] [ 4 ]   [ 5 ]  [ 6 ]          [ 7 ]
```

[属性]　[连接] [所有者] [用户组] [文件容量] [修改日期] [文件名]

ls（“list”）与早期的 DOS 指令 dir 功能类似，参数“`-al`”则表示列出所有的文件（包含隐藏文件）的详细的权限（Permission）与属性（Attribute）。

第一栏代表这个文件的属性，共有 10 个字符，分别代表 10 个属性，其中第一个属性代表这个文件是“目录、文件或连接文件等”。

- 若是 [d] 则是目录；
- 若是 [-] 则是文件；

- 若是 [l] 则表示为连接文件（link file）；
- 若是 [b] 则表示为设备文件里面的可供存储的接口设备；
- 若是 [c] 则表示为设备文件里面的串行端口设备，例如键盘、鼠标等一次性读取设备。

文件的权限属性被分为 3 个为一组且均为“rwx”的 3 个参数的组合，其中 [r] 代表可读（read）、[w] 代表可写（write）、[x] 代表可执行（execute）。

要注意的是，这 3 个权限的位置不会改变，如果没有权限，就会出现减号（-）。

- 第一组为“文件所有者 (owner) 的权限”；
- 第二组为“同用户组 (usergroup) 的权限”；
- 第三组为“其它非本用户组 (others) 的权限”。

若有一个文件的属性为“`-rwxr-xr--`”，可简单说明如下：

```
[-][rwx][r-x][r--]
 1  234  567  890
```

1 为：代表这个文件名为目录或文件（上面为文件）；
234 为：文件所有者的权限（上面为可读、可写、可执行）；
567 为：同用户组用户权限（上面为可读可执行）；
890 为：其它用户权限（上面为仅可读）。

上面的属性情况代表一个文件、这个文件的文件所有者可读可写可执行但同用户组的人仅可读与执行，非同用户组的用户仅可读。

除此之外，需要特别留意的是“x”这个标志，若文件名为一个目录，例如上表中的.ssh 这个目录：

```
drwx------  3  root  root  4096  Jun 25 08:35  .ssh
```

可以看到这是一个目录，而且只有 root 可以读写与执行。但是若为下面的样式时，root 的其它人是否可以进入该目录呢？

```
drwxr--r--  3  root  root  4096  Jun 25 08:35  .ssh
```

此时表示非 root 这个账号的其它用户均不可进入.ssh 这个目录，为什么呢？

因为 x 与目录的关系相当的重要，如果在该目录下不能执行任何指令的话，那么自然也就无法进入了，因此特别留意的是如果想要开放某个目录让一些人进来的话，就要该目录的 x 属性开放，因此目录与文件的权限意义并不相同，这是因为目录与文件所记录的数据内容不相同所致。

第二栏表示为连接占用的节点（i-node），每个文件都会将它的权限与属性记录到文件系统的 i-node 中。

Linux 的目录树是使用文件名来记录，因此每个文件名就会连接到一个 i-node，这个跟连接文件（link file）比较有关系。如果是目录的话，那么就与该目录下还有多少目录有关，因此 i-node 属性记录了有多少个不同的文件名连接到相同的一个 i-node 号码。

在 Linux 中，ID（如 root 或 test 等账号均是所谓的 ID）就是用户的身份，而且这些账号还可以附属在一个或多个用户组中。

特别注意，如果是以中文来安装 Linux 时，那么默认的语系可能会被改为中文。如果中文无法显示在文字形式的终端上，就需要把/etc/sysconfig/i18n 中的“LC_TIME”修改为 LC_TIME=en_US 保存后注销，再登录后就可以得到英文字符显示的日期。

如果文件名之前多一个“.”，则代表这个文件为“隐藏文件”。

与 Windows 系统不一样的是，在 Linux 系统（或者说 UNIX like 系统）中的每一个文件都附加了很多的属性，尤其是用户组的概念。

例如，关于系统服务的文件通常只有 root 才能读写或者是执行，/etc/shadow 记录了系统中的所有账号的数据，不能让任何人读取。

```
[theqiong@localhost ~]$ ll /etc/shadow
----------. 1 root root 1075 Nov 24 11:44 /etc/shadow
```

在修改 Linux 文件与目录的属性之前，一定要明白什么数据是可变的，什么数据是不能改变的。

文件的权限大致上包括用户组、所有者和各种身份的权限等，其中用于修改文件权限的命令主要是 chgrp、chown 和 chmod。

- chgrp：改变文件所属用户组
- chown：改变文件所属人
- chmod：改变文件的属性、SUID 等的特性

24.1 chgrp

改变一个文件的用户组可以直接以 chgrp（change group）来修改。不过，用户组名称必须是在/etc/group 文件内已存在，否则就会显示错误。

```
#chgrp [-R] dirname/filename ...
```

参数

-R：进行递归的持续更改，也就是连同子目录下的所有文件、目录都更改成为这个用户组。常常用在更改某一目录内所有文件的情况。

```
#chgrp users install.log
#ls -l
-rw-r--r--  1 root users 68495 Jun 25 08:53 install.log
#chgrp testing install.log
chgrp: invalid group name 'testing'
```

24.2 chown

chown（change owner）用来改变文件的所有者，必须是已经存在于/etc/passwd 文件中的用户才可以更改。

chown 还可以直接修改用户组的名称，而且可以通过-R 的参数来更改目录下的所有子目录或文件的文件所有者属性。

```
#chown [-R] 账号名称文件或目录
#chown [-R] 账号名称：用户组名称文件或目录
```

参数

-R：进行递归的持续更改，也就是连同子目录下的所有文件、目录都更新成为这个用户。常常用在更改某一目录的情况。

范例

```
#chown bin install.log
#ls -l
-rw-r--r--  1 bin  users 68495 Jun 25 08:53 install.log
#chown root:root install.log
#ls -l
-rw-r--r--  1 root root 68495 Jun 25 08:53 install.log
```

事实上，chown 也可以有“chown user.group file”的命令，即所有者与用户组间加上小数点“.”，只是现在设置账号时也会在账号中加入小数点，可能就会造成系统的误判，所以一般建议使用冒号“:”来隔开所有者与用户组。

另外，chown 也能单纯地修改所属用户组，例如“**chown .sshd install.log**”就是修改用户组，这就是点号的用途。

在需要更改文件的所有者的场景中，最常见的就是在复制文件给其它人时进行文件属性修改。

```
#cp 来源文件目的文件
```

假设要将.bashrc 文件拷贝成.bashrc_test 且要给 bin 这个用户，可以这样做：

```
#cp .bashrc .bashrc_test
#ls -al .bashrc*
```

```
-rw-r--r--  1 root root 395 Jul  4 11:45  .bashrc
-rw-r--r--  1 root root 395 Jul 13 11:31  .bashrc\_test
```

如果没有执行 chown 指令，那么.bashrc_test 还是属于 root 所有，这时即使将文件发给 bin 这个用户了，该文件仍然无法修改的，所以必须要将这个文件的所有者与用户组进行修改。

24.3 chmod

文件权限的改变使用的是 chmod 指令，可以使用数字或者是符号来进行权限的更改。

24.3.1 Numeric Permission

Linux 文件的基本权限就有 9 个，分别是 owner/group/others 三种身份各自的 read-/write/excute 权限。

例如，`-rwxrwxrwx` 这 9 个权限是三个一组的，其中可以使用数字来代表各个权限。

```
r:4
w:2
x:1
```

同一组（owner/group/others）的三个权限（r/w/x）是需要累加的，例如：

```
owner   = rwx   = 4+2+1 = 7
group   = rwx   = 4+2+1 = 7
others  = ---   = 0+0+0 = 0
```

设定权限的更改时，该权限的数字就是 770，那么更改权限的指令 chmod 的语法如下：

```
#chmod [-R] xyz 文件或目录
```

参数：
xyz：就是刚刚提到的数字类型的权限属性，为rwx属性数值的相加。
-R：进行递归的持续更改，也就是连同子目录下的所有文件、目录
都更新成为这个用户组之意，常常用在更改某一目录的情况。

举例来说，如果要将.bashrc 这个文件所有的属性都打开，那么就执行：

```
#ls -al .bashrc
-rw-r--r--  1 root root 395 Jul  4 11:45 .bashrc
#chmod 777 .bashrc
#ls -al .bashrc
-rwxrwxrwx  1 root root 395 Jul  4 11:45 .bashrc
```

一个文件有三组属性，所以可以发现上面 777 为三组，将所有的属性都打开后的数字都相加得到 `r+w+x = 4+2+1 = 7`。

另外，实际应用中最常发生的一个问题就是：以 vi 编辑一个 shell 的文字文件后，它的属性通常是`-rw-rw-rw-`，也就是 666 的属性，如果要将它变成可执行文件，并且不要让其它人修改该文件的话，那就需要`-rwxr-xr-x` 这一个 755 的属性，所以需要这样做：

```
chmod 755 test.sh
```

另外，有些文件不希望被其它人看到，例如`-rwxr-----`，那么就执行：

```
chmod 740 filename
```

24.3.2 Symbolic Permission

Linxu 中基本上就是（1）user（2）group（3）others 这三组的 9 个属性，于是就可以通过 u、g、o 来代表这三个组的属性。

此外，a 则代表 all，也就是全部的三组，那么读写的属性就可以写成 r、w、x，也就是可以使用下面的方式：

chmod	u g o	+（加入） -（除去） =（设置）	r w x	文件或目录
chmod	a	+（加入） -（除去） =（设置）	r w x	文件或目录

假如要设定一个文件的属性为“`-rwxr-xr-x`”，基本上就是：

（1）user（u）：具有可读、可写、可执行的权限；

（2）group 与 others（g/o）：具有可读与执行的权限。

所以就是：

```
[root@linux ~]#chmod u=rwx,go=rx .bashrc
#注意，u=rwx,go=rx 是用逗号连在一起的，中间并没有任何空白字符。
[root@linux ~]#ls -al .bashrc
-rwxr-xr-x  1 root root 395 Jul  4 11:45 .bashrc
```

注意，u=rwx,og=rx这一段文字之间并没有空白字符隔开。

如果是“`-rwxr-xr--`”，可以使用

```
chmod u=rwx,g=rx,o=r filename
```

来设定。

此外，如果不知道原先的文件属性，只想要增加.bashrc 这个文件的每个人均可写入的权限，那么就可以使用：

```
[root@linux ~]#ls -al .bashrc
-rwxr-xr-x  1 root root 395 Jul  4 11:45 .bashrc
[root@linux ~]#chmod a+w .bashrc
[root@linux ~]#ls -al .bashrc
-rwxrwxrwx  1 root root 395 Jul  4 11:45 .bashrc
```

而如果是要将属性去掉而不更动其它的属性，例如要去掉所有人的 x 的属性，则：

```
[root@linux ~]#chmod a-x .bashrc
[root@linux ~]#ls -al .bashrc
-rw-rw-rw-  1 root root 395 Jul  4 11:45 .bashrc
```

在 + 与-的状态下，只要是没有指定到的项目，则该属性“不会被变动”，例如上面的例子中，由于仅以-去掉 x 则其它两个保持当时的值不变，另外利用

```
chmod a+x filename
```

就可以让该程序拥有执行的权限了。

24.4 File Permission

文件是实际含有数据的地方，包括一般文本文件、数据库内容文件、二进制可执行文件等。

权限对于文件来说，它的意义如下：

（1）r（read）：可读取此文件的实际内容，如读取文本文件的文字内容等。

（2）w（write）：可以编辑、新增或者是修改该文件的内容（但不含删除该文件）；

（3）x（execute）：该文件具有可以被系统执行的权限。

对一个文件具有 w 权限时，可以具有写入、编辑、新增、修改文件的内容的权限，但此时并不具有删除该文件本身的权限。

另外，也必须要更加的注意的是，在 Windows 下一个文件是否具有执行的能力是通过“扩展名”（例如.exe、.bat、.com 等）来判断，但是在 Linux 下的文件是否能执行则是通过是否具有 x 这个属性来决定的，所以跟文件名是没有绝对的关系。

对于文件的 r、w、x 来说，主要都是针对“文件的内容”而言，与文件名的存在与否没有关系，在 Linux 中文件记录的是真实的数据。

24.5 Directory Permission

文件是存放实际数据的所在，目录主要的内容是记录文件名列表，当 r、w、x 针对目录时，简单的说：

（1）r（read contents in directory）：

表示具有读取目录结构清单的权限，所以当具有读取（r）一个目录的权限时就可以查询该目录下的文件名数据，于是就可以利用 ls 这个指令将该目录的内容列表显示出来。

（2）w（modify contents of directory）：这个可写入的权限表示用户将具有更改该目录结构清单的权限，也就是下面这些权限：

- 建立新的文件与目录；
- 删除已经存在的文件与目录（不论该文件的权限是什么）；
- 将已存在的文件或目录进行重命名；
- 移动该目录内的文件、目录位置。

w 则具有相当重要的权限，它可以让用户删除、更新、新建文件或目录，所以说，如果一般身份用户 xx 在/home/xx 这个主文件夹内，无论是谁（包括 root）建立的文件，无论该文件属于谁，无论该文件的属性是什么，xx 这个用户都“有权力将该文件删除”。

（3）x（access directory）：这里的 x 与能否进入该目录有关。

```
[root@linux ~]#cd /tmp
[root@linux tmp]#mkdir testing
[root@linux tmp]#chmod 744 testing
[root@linux tmp]#touch testing/testing
[root@linux tmp]#chmod 600 testing/testing
#mkdir 是在建立目录用的指令，make directory 的缩写。
#用 root 的身份在/tmp 下建立一个名为 testing 的目录，
#并且将该目录的权限变为 744，该目录的所有者为 root。
#另外，touch 可以用来建立一个没有内容的文件，因此 touch testing/testing
#可以建立一个空的/tmp/testing/testing 文件。
[root@linux tmp]#ls -al
drwxr--r--   2 root root 4096 Jul 14 01:05 testing
#目录的权限是 744 ，且所属用户组与用户均是 root，
#接下来，将 root 的身份切换成为一般身份用户。
#切换身份成为 dmtsai
[root@linux tmp]#su dmtsai
#su 的指令是用来“切换身份”的一个指令。
#也就是说，现在当前用户变成 dmtsai，那么 dmtsai 这个人对于
#/tmp/testing 是属于 others 的权限。
[dmtsai@linux tmp] ls -l testing   <== 此时身份为dmtsai
total 0
#可以查看里面的信息，因为 dmtsai 具有 r 的权限，不过毕竟权限不够，
```

```
#很多数据竟然是问号 (?) 来的
[dmtsai@linux tmp] cd testing  <== 此时身份为dmtsai
bash: cd: testing/: Permission denied
#即使具有 r 的权限，但是没有 x，所以 dmtsai 无法进入/tmp/testing
[dmtsai@linux tmp] exit
[root@linux tmp]#chown dmtsai testing
#使用 exit 就可以离开 su 的功能。将 testing 目录的所有者设定为
#dmtsai，此时 dmtsai 就成为 owner。
[root@linux tmp]#su dmtsai
[dmtsai@linux tmp] cd testing  <== 此时身份为dmtsai
[dmtsai@linux testing] ls -l  <== 此时身份为dmtsai
-rw-------  1 root root 0 Jul 14 01:13 testing
#再切换身份成为 dmtsai ，此时就能够进入 testing。查看一下内容。
#发现 testing 这个文件存在，权限是只有 root 才能够存取。
#那我们测试一下能否删除呢？
[dmtsai@linux testing] rm testing  <== 此时身份为dmtsai
rm: remove write-protected regular empty file ‘testing’? y
#可以删除。
```

能不能进入某一个目录，只与该目录的 x 权限有关，目录只是记录文件名而已，也就是说目录是不能被执行的，目录的“x”代表的是用户能否进入该目录并成为工作目录的用途。

“工作目录（work directory）”就是当前所在的目录，Linux 中变换目录的命令是“cd（change directory）”。

此外，工作目录对于命令的执行是非常重要的，如果在某目录下不具有 x 的权限，那么就无法切换到该目录下，也就无法执行该目录下的任何命令，即使具有对该目录的 r 权限。

网站服务器中如果只开放目录的 r 权限，将导致任何服务器软件都无法到该目录下读取文件（最多只能看到文件名），访客也总是无法正确查看目录内的文件的内容（显示权限不足），因此要开放目录给任何人浏览时，应该至少也要给予 r 及 x 的权限，但不能赋予 w 权限。

权限对于用户账号来说是非常重要的，它可以限制用户能不能读取/新建/删除/修改文件或目录，下面分别说明命令在什么样的权限下才能运行：

1、让用户能进入某目录成为“可工作目录”的基本权限。

- 可使用的命令：例如 cd 等切换工作目录的命令；
- 目录所需权限：用户对这个目录至少需要具有 x 的权限；
- 额外需求：如果用户想要在这个目录内利用 ls 查阅文件名，则用户对此目录还需要 r 的权限。

2、让用户在某个目录内读取一个文件的基本权限。

- 可使用的命令：例如 cat、more、less 等；
- 目录所需权限：用户对这个目录至少需要具有 x 权限；
- 文件所需权限：用户对文件至少需要具有 r 的权限。

3、让用户可以修改一个文件的基本权限。

- 可使用的命令：例如 vi 等；
- 目录所需权限：用户在该文件所在的目录至少要有 x 权限；
- 文件所需权限：用户对该文件至少要有 r、w 权限。

4、让用户可以创建一个文件的基本权限。

- 目录所需权限：用户在该目录要具有 w、x 的权限，重点在 w。

5、用户进入某目录并执行该目录下的某个命令的基本权限。

- 目录所需权限：用户在该目录至少要有 x 的权限。
- 文件所需权限：用户在该文件至少需要有 x 的权限。

24.6 File Type

任何设备在 Linux 下都是文件，例如数据通信的接口也有专门的文件负责。

24.6.1 Regular File

在由 `ls -al` 所显示出来的属性中，第一个字符为 `[-]`，例如 `[-rwxrwxrwx]`。另外，依照文件的内容，又大略可以分为：

- 纯文本文件（ASCII）：纯文本文件中的内容是人类可以直接读取的数据（例如数字、字母等）。几乎只要我们可以用来做为设置的文件都属于这一种文件类型。举例来说，可以执行 `cat /.bashrc` 就可以看到该文件的内容，cat 是将一个文件内容读出来的命令。

- 二进制文件（binary）：操作系统其实仅认识且可以执行二进制文件，Linux 中的可执行文件（不包括 scripts 和批处理文件）就是这种格式。举例来说，cat 就是一个二进制文件。

- 数据格式文件（data）：有些程序在运行的过程当中会读取某些特定格式的文件，那些特定格式的文件可以被称为数据文件（data file）。举例来说，Linux 在用户登录时都会将登录的数据记录在/var/log/wtmp 那个文件内，该文件就是一个 data file，它能够通过 last 这个指令读出来，但是使用 cat 读取时会读出乱码，因为它是属于一种特殊格式的文件。

24.6.2 Directory

第一个属性为 [d] 的文件就是目录，例如 [drwxrwxrwx]。

24.6.3 Link

类似 Windows 系统下的快捷方式，第一个属性为 [l]，例如 [lrwxrwxrwx]。

24.6.4 Device

与系统外设及存储等相关的一些文件，通常都集中在/dev 这个目录下，通常又分为两种：

（1）块（block）设备文件就是一些存储数据以提供系统随机访问的接口设备，比如硬盘，现代操作系统可以随机地在硬盘的不同区块读写，例如 1 号硬盘的代码是/dev/hda1 等文件，第一个属性为 [b]；

（2）字符（character）设备文件也就是一些串行端口的接口设备，例如键盘、鼠标等，这些设备的特征就是“一次性读取”，不能够截断输出，这些文件的第一个属性为 [c]。

24.6.5 Socket

socket 类型的文件通常被用于网络上的数据连接，当启动一个程序来监听用户端的请求时，客户端就可以通过 socket 来进行数据的通信。

socket 文件的第一个属性为 [s]，通常会在/var/run 目录中看到 socket 文件。

24.6.6 Pipe

FIFO（first-in-first-out）的主要用途是解决多个程序同时存取一个文件所造成的错误问题。

FIFO 文件的第一个属性为 [p]。

除了设备文件是系统中很重要的文件，最好不要随意修改之外（通常它也不会让用户修改的），另一个比较有趣的文件就是连接文件，可以将 linux 下的连接文件简单的视为一个文件或目录的快捷方式。

socket 与 FIFO 文件与进程（process）有关。

24.7 File Extension

基本上 Linux 的文件是没有所谓的“扩展名”的，Linux 文件能不能被执行与其属性有关，与文件名无关。

Windows 中的可执行文件的扩展名通常是.com、.exe、.bat 等，在 Linux 文件只要权限当中有 x 就代表文件可以被执行。

不过，可以被执行跟可以执行成功是不一样的，x 代表文件具有可执行的能力，但是能不能执行成功还要看文件的内容。

扩展名一般用来了解文件格式等信息，所以通常还是会以适当的扩展名来表示文件。

- *.sh：脚本或批处理文件（scripts），批处理文件是使用 shell 编写的。
- *Z, *.tar, *.tar.gz, *.zip, *.tgz：经过打包的压缩文件，表示压缩软件分别为 gunzip、tar 等，使用不同的压缩软件可以设置相关的扩展名。
- *.html、*.php：网页相关文件，分别代表 HTML 语法与 PHP 语法的网页文件等。
- *.pl：使用 perl 语言编写的文件。

基本上，Linux 中的文件名只是让用户了解文件的可能用途，真正的执行与否仍然需要权限的规范。例如，虽然文件本身为可执行文件（例如代理服务器软件 squid），但是如果这个文件的属性被修改成无法执行时，那么它就变成不能执行，这种问题最常发生在文件传送的过程中，例如在网络上下载一个可执行文件，但是偏偏在 Linux 系统中就是无法执行，原因可能就是文件的属性被改变了。

24.8 File Limit

当 Linux 系统使用默认的 ext2/ext3 文件系统时，针对文件的文件名长度限制为：

1、单一文件或目录的最大容许文件名为 255 个字符；

2、包含完整路径名称及目录（/）的完整文件名为 4096 个字符。

一般来说，在设置 Linux 下的文件名称时最好可以避免一些特殊字符，例如 `* ? > < ; & ! [ ] | \ ' " ` ( ) { }` ，这些符号在命令行模式下是有特殊意义的。

在 Linux 下单一文件或目录的文件名，加上完整路径时，最长可达 4096 个字符，可以通过 [tab] 按键来确认文件的文件名。

另外，文件名称的开头为小数点“.”时代表“隐藏文件”。另外，指令执行时常常会使用到`-option` 等参数，所以最好要避免将文件文件名的开头以-或 + 来命名。

Chapter 25

File Management

25.1 Path

“绝对路径”（absolute path）是从根目录/开始的，例如/usr/bin/。

相对路径（relative path）从当前目录开始，可以向前或向下继续。

一般来说，在 Shell Scripts 中务必使用绝对路径，如果使用相对路径则可能无法在不同的工作环境保持一致，在进程调度（at、cron）与例行性命令中尤其重要。

25.1.1 cd

切换目录的指令是 cd（change directory），下面这些就是 Linux 中比较特殊的目录：

```
.          代表此层目录，也可以写成./;
..         代表上一层目录，也可有写成../;
-          代表前一个工作目录;
~          代表“目前用户身份”所在的主文件夹;
~account   代表account这个使用者的主文件夹;
```

在所有目录下有两个目录是一定会存在的，那就是.与..，它们分别代表当前层与上层目录。

在 Linux 文件属性与目录配置中提到根目录（/）是所有目录的最顶层，而且/也有..。

```
# ls -al ../
total 316
drwxr-xr-x.  18 root root   4096 Dec 19 10:10 .
drwxr-xr-x.  18 root root   4096 Dec 19 10:10 ..
lrwxrwxrwx.   1 root root      7 Dec 12  2013 bin -> usr/bin
```

```
dr-xr-xr-x.   6 root root   3072 Dec 15 14:51 boot
drwxr-xr-x.  20 root root   3400 Dec 19 10:09 dev
drwxr-xr-x. 149 root root  12288 Dec 19 12:57 etc
drwxr-xr-x.   4 root root   4096 Oct 29 20:10 home
lrwxrwxrwx.   1 root root      7 Dec 12  2013 lib -> usr/lib
lrwxrwxrwx.   1 root root      9 Dec 12  2013 lib64 -> usr/lib64
drwx------.   2 root root  16384 Dec 12  2013 lost+found
drwxr-xr-x.   2 root root   4096 Aug  7  2013 media
drwxr-xr-x.   2 root root   4096 Aug  7  2013 mnt
drwxr-xr-x.   6 root root   4096 Nov 24 13:44 opt
dr-xr-xr-x. 230 root root      0 Dec 19 18:09 proc
-rw-r--r--.   1 root root 247372 Dec 19 10:10 .readahead
dr-xr-x---.  15 root root   4096 Dec 19 21:33 root
drwxr-xr-x.  38 root root   1040 Dec 19 19:22 run
lrwxrwxrwx.   1 root root      8 Dec 12  2013 sbin -> usr/sbin
drwxr-xr-x.   2 root root   4096 Aug  7  2013 srv
dr-xr-xr-x.  13 root root      0 Dec 19 10:09 sys
drwxrwxrwt.  13 root root    320 Dec 19 21:33 tmp
drwxr-xr-x.  13 root root   4096 Sep 27 09:58 usr
drwxr-xr-x.  22 root root   4096 Dec 19 18:09 var
```

根目录的上层目录时会看到根目录本身（.）与其上一层（..）属性完全一样，根目录的顶层（..）与它自己（.）是同一个目录。

除了要注意 FHS 目录配置外，在文件名部分也要特别注意，根据文件名写法的不同也可将所谓的路径（path）定义为绝对路径（absolute）与相对路径（relative）。

- 绝对路径：由根目录（/）开始写起的文件名或目录名称，例如/home/theiqong/.bashrc；
- 相对路径：相对于当前工作目录的文件名写法，例如./home/theqiong 或../../home/theqiong/等。

开头不是/就属于相对路径的写法，而且必须要明白相对路径是以“当前所在路径的相对位置”来表示的。例如，从/home 进入/var/log，就可以使用如下的命令：

```
cd /var/log (absolute)
cd ../var/log (relative)
```

用户 theqiong 的主是/home/theqiong，而 root 的主目录则是/root，假设以 root 身份登录 Linux 系统，那么下面就是这几个特殊的目录的意义：

```
[root@linux ~]#cd [相对路径或绝对路径]
#最重要的就是目录的绝对路径与相对路径，还有一些特殊目录的符号。
[root@linux ~]#cd  dmtsai
```

```
#代表去到 dmtsai 这个用户的主文件夹，也就是/home/dmtsai
[root@linux dmtsai]#cd
#表示回到自己的主文件夹，也就是/root 这个目录
[root@linux ~]#cd
#没有加上任何路径，也还是代表回到自己主文件夹的意思。
[root@linux ~]#cd ..
#表示去到目前的上层目录，也就是/root 的上层目录的意思;
[root@linux /]#cd -
#表示回到刚刚的那个目录，也就是/root。
[root@linux ~]#cd /var/spool/mail
#这个就是绝对路径的写法，直接指定要去的完整路径名称。
[root@linux mail]#cd ../mqueue
#这个是相对路径的写法，由/var/spool/mail 去到/var/spool/mqueue 就这样写。
```

以 root 身份登录 Linux 系统后会在 root 的主文件夹也就是/root 下。使用相对路径时，必须要确认目前的路径才能正确的转到想要去的目录。

其实提示字符 `[root@linux ~]#` 中就已经指出目前目录了，刚登录时会到自己的主文件夹，而主文件夹还有一个代码，那就是~ 符号，从上面的例子可以发现，使用 `cd ~` 可以回到普通用户的主文件夹中。

另外，针对 cd 的使用方法，如果仅输入 cd 代表的就是 `cd ~` 的意思，也就是会回到自己的主文件夹，而 `cd -`则代表回到前一个工作目录。

```
$ cd ~
$ cd -
```

25.1.2 pwd

```
[root@linux ~]#pwd [-P]
参数:
-P  : 显示出确实的路径，而非使用连接 (link) 路径。
范例:
[root@linux ~]#pwd
/root   <== 显示出目录
[root@linux ~]#cd /var/mail
[root@linux mail]#pwd
/var/mail
[root@linux mail]#pwd -P
/var/spool/mail
```

```
[root@linux mail]#ls -l /var/mail
lrwxrwxrwx  1 root root 10 Jun 25 08:25 /var/mail -> spool/mail
#看到这里应该知道，/var/mail 是连接文件，连接到/var/spool/mail。
#所以，加上 pwd -P 的参数后会不以连接文件的数据显示，而是显示正确的完整路径。
```

pwd（Print Working Directory）可以显示目前所在目录。

-P 的参数可以让用户取得正确的目录名称而不是以连接文件的路径来显示。例如在 Fedora 中，/var/mail 是/var/spool/mail 的连接文件，所以通过到/var/mail 执行 pwd -P 就能知道这个参数的意义。

25.1.3 mkdir

```
[root@linux ~]#mkdir [-mp] 目录名称
参数:
-m 设定文件的权限，使用-m参数可以直接设定，不需要看默认权限（umask）。
-p 帮助用户直接将所需要的目录递归地建立起来。
范例:
[root@linux ~]#cd /tmp
[root@linux tmp]#mkdir test <== 建立一个名为 test 的新目录
[root@linux tmp]#mkdir test1/test2/test3/test4
mkdir: cannot create directory ‘test1/test2/test3/test4’:
No such file or directory  <== 无法直接建立此目录。
[root@linux tmp]#mkdir -p test1/test2/test3/test4
#加了这个-p 参数，可以自行帮用户建立多层目录。
[root@linux tmp]#mkdir -m 711 test2
[root@linux tmp]#ls -l
drwxr-xr-x  3 root  root 4096 Jul 18 12:50 test
drwxr-xr-x  3 root  root 4096 Jul 18 12:53 test1
drwx--x--x  2 root  root 4096 Jul 18 12:54 test2
#仔细看上面的权限部分，如果没有加上-m 来强制设定属性，系统会使用默认属性。
#而默认属性则由 umask 决定的。
```

mkdir（make directory）命令可以用于建立新的目录，在默认的情况下用户所需要的目录需要逐层建立。

如果需要建立/home/theqiong/testing/test1，那么首先必须要有/home 然后是/home/theqiong，再然后是/home/theqiong/testing 都必须要存在，才可以建立/home/theqiong/testing/test1 目录。如果没有/home/bird/testing 就无法建立 test1 目录。

在使用 mkdir 命令时指定-p 参数可以递归地建立目录。例如，在下面的示例中，系统会自动的将/home，/home/theqiong，/home/theqiong/testing 以此建立，并且如果该目录本来

就已经存在时，系统也不会显示错误信息。

```
mkdir –p /home/theqiong/testing/test1
```

关于目录的“默认权限”，可以利用-m 来强制给予一个新的目录相关的属性，例如可以利用-m 711 来强制给予新的目录 drwx--x--x 的属性。

如果没有指定-m 属性时，那么默认的新建目录属性就和 umask 有关。

25.1.4 rmdir

```
[root@linux ~]#rmdir [-p] 目录名称
参数:
–p 连同上层“空的”目录也一起删除
范例:
[root@linux tmp]#ls -l
drwxr–xr–x  3 root  root 4096 Jul 18 12:50 test
drwxr–xr–x  3 root  root 4096 Jul 18 12:53 test1
drwx––x––x  2 root  root 4096 Jul 18 12:54 test2
[root@linux tmp]#rmdir test
[root@linux tmp]#rmdir test1
rmdir: ‘test1’: Directory not empty
[root@linux tmp]#rmdir -p test1/test2/test3/test4
[root@linux tmp]#ls -l
drwx––x––x  2 root  root 4096 Jul 18 12:54 test2
#利用-p 参数就可以将 test1/test2/test3/test4 直接删除。
#不过要注意的是，rmdir 仅能“删除空的目录”。
```

如果想要删除旧有的目录时，可以使用 rmdir 命令，不过要注意目录需要逐层地进行删除，而且被删除的目录里面不能还有其它的目录或文件，这也是所谓的空的目录（empty directory）的原因。

如果要将所有目录下的内容都删除，就必须使用 rm -rf test。

25.1.5 $PATH

在执行命令时，系统会依照 PATH 变量的设定去 PATH 定义的每个路径下搜索执行文件，先搜索到的指令先被执行。

通过执行 echo $PATH 可以查看 PATH 变量的内容，echo 有“显示、打显示出”的意思，而 PATH 前面加的 $ 表示后面接的是变量，所以就会显示出当前用户的 PATH。

```
$ echo $PATH
```

```
/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:
/home/theqiong/.local/bin:/home/theqiong/bin
```

PATH 主要用来规范指令搜索的目录，而且每个目录是有顺序的，这些目录中间以冒号“:”来分隔。

如果需要将/root 路径加入 PATH 中，可以执行下面的命令：

```
# PATH=$PATH:/root
```

或者，可以使用完整文件名执行指令，也就是直接使用相对或绝对路径来执行，例如：

```
#/root/ls
#./ls
```

为了安全起见，不建议将“.”加入 PATH 的查询目录中。

- 不同身份用户默认的 PATH 不同，默认能够随意执行的命令也不同；
- PATH 是可以修改的，一般用户还是可以通过修改 PATH 来执行某些位于/sbin 或/usr/sbin 下的命令来查询；
- 使用绝对路径或相对路径直接指定某个命令的文件名来执行，会比查询 PATH 要正确；
- 命令应该要放置到正确的目录下，执行才会比较方便；
- 当前目录（.）最好不要放到 PATH 当中。

25.2 文件与目录管理

文件与目录的管理包括“显示属性”、“拷贝”、“删除文件”及“移动文件或目录”等，在执行程序时系统默认有一个搜索的路径顺序，如果有两个以上相同文件名的执行文件分别在不同的路径时，就需要特别留意。

25.2.1 ls

在 Linux 系统中，通过 ls 指令可以输出文件或者是目录的相关信息。

```
[root@linux ~]#ls [-aAdfFhilRS] 目录名称
[root@linux ~]#ls [-color=never,auto,always] 目录名称
[root@linux ~]#ls [-full-time] 目录名称
```

参数：

-a ：全部的文件，连同隐藏文件（开头为.的文件）一起列出来。
-A ：全部的文件，连同隐藏文件，但不包括.与..这两个目录，一起列出来。
-d ：仅列出目录本身，而不是列出目录内的文件数据
-f ：直接列出结果，而不进行排序（ls默认会以文件名排序）。

-F ：根据文件、目录等信息，给予附加数据结构，例如
*：代表可执行文件； /：代表目录； =：代表 socket 文件；|：代表FIFO文件；
-h ：将文件容量以较易读的方式（例如GB、KB等）列出来；
-i ：列出inode位置，而非列出文件属性；
-l ：长数据串列出，包含文件的属性等等数据；
-n ：列出UID与GID而非使用者与用户组的名称。
-r ：将排序结果反向输出，例如原本文件名由小到大，反向则为由大到小；
-R ：连同子目录内容一起列出来；
-S ：以文件容量大小排序!
-t ：依时间排序
--color=never ：不要依据文件特性给予颜色显示；
--color=always ：显示颜色
--color=auto ：让系统自行依据设定来判断是否给予颜色
--full-time ：以完整时间模式（包含年、月、日、时、分）输出
--time={atime,ctime}
：输出access时间或改变权限属性时间（ctime）而非内容变更时间（modification time）

默认情况下，执行 ls 时显示的只有非隐藏文件的文件名、以文件名进行排序及文件名代表的颜色显示等。

如果还想要加入其它的显示信息时可以加入参数，例如`-al` 显示数据内容以及隐藏文件。

Linux 文件系统记录了关于文件的很多有用的信息，其中文件系统中与权限、属性有关的数据放在 i-node 中。

在默认的情况中，可以将 ll 设定成为 `ls -l`，使用 bash shell 的 alias 功能可以实现输入 `ll` 就等于是输入 `ls -l`。

25.2.2 cp

要复制文件或目录，可以使用 cp（copy）指令。

除了用于单纯的复制，cp 指令还可以建立连接文件，对比文件的新旧而更新以及复制整个目录等。

```
[root@linux ~]#cp [-adfilprsu] 来源文件（source） 目的文件（destination）
[root@linux ~]#cp [options] source1 source2 source3 .... directory
```

参数:

-a ：相当于-pdr的意思；
-d ：若来源文件为连接文件的属性（link file），则复制连接文件属性而非文件本身；

−f
: 为强制（force）的意思，若有重复或其它疑问时，不会询问使用者，而强制复制；
−i ：若目的文件（destination）已经存在时，在覆盖时会先询问是否真的执行。
−l ：进行硬式连结（hard link）的连接文件建立，而非复制文件本身；
−p ：连同文件的属性一起复制过去，而非使用默认属性；
−r ：递归持续复制，用于目录的复制行为；
−s ：复制成为符号连接文件（symbolic link），也就是“快捷方式”文件；
−u ：若destination比source旧才更新destination。
最后需要注意的，如果来源文件有两个以上，则最后一个目的文件一定要是“目录”才行！
范例：
范例一：将主文件夹下的.bashrc复制到/tmp下，并更名为bashrc

```
[root@linux ~]#cd /tmp
[root@linux tmp]#cp  /.bashrc bashrc
[root@linux tmp]#cp -i  /.bashrc bashrc
cp: overwrite ‘basrhc’? n
#重复作两次动作，由于/tmp 下已经存在 bashrc 了，加上-i 参数，
#则在覆盖前会询问使用者是否确定，可以按下 n 或者 y。
#但是，反过来说，如果不想要询问时，则加上-f 参数来强制直接覆盖。
```

范例二：将/var/log/wtmp复制到/tmp下

```
[root@linux tmp]#cp /var/log/wtmp . <== 想要复制到目前的目录，最后的. 不要忘
[root@linux tmp]#ls -l /var/log/wtmp wtmp
−rw−rw−r−−  1 root utmp 71808 Jul 18 12:46 /var/log/wtmp
−rw−r−−r−−  1 root root 71808 Jul 18 21:58 wtmp
#在不加任何参数的情况下，文件的所属者会改变，权限也跟着改变，连文件建立的时间也
    不一样了。
#如果想要将文件的所有特性都一起复制过来，可以加上-a。
[root@linux tmp]#cp -a /var/log/wtmp wtmp_2
[root@linux tmp]#ls -l /var/log/wtmp wtmp_2
−rw−rw−r−−  1 root utmp 71808 Jul 18 12:46 /var/log/wtmp
−rw−rw−r−−  1 root utmp 71808 Jul 18 12:46 wtmp\_2
#整个数据特性完全一模一样，这就是-a 的特性。
```

范例三：复制/etc/这个目录下的所有内容到/tmp下

```
[root@linux tmp]#cp /etc/ /tmp
cp: omitting directory ‘/etc’   <== 如果是目录，不能直接复制，要加上−r的参数
[root@linux tmp]#cp -r /etc/ /tmp
#再次强调，-r 是可以复制目录，但是文件与目录的权限会被改变。
```

```
#所以，也可以利用 cp -a /etc /tmp 来执行指令。
```

范例四：将范例一复制的bashrc建立一个连接文件（symbolic link）

```
[root@linux tmp]#ls -l bashrc
-rw-r--r--  1 root root 395 Jul 18 22:08 bashrc
[root@linux tmp]#cp -s bashrc bashrc_slink
[root@linux tmp]#cp -l bashrc bashrc_hlink
[root@linux tmp]#ls -l bashrc*
-rw-r--r--  2 root root 395 Jul 18 22:08 bashrc
-rw-r--r--  2 root root 395 Jul 18 22:08 bashrc\_hlink
lrwxrwxrwx  1 root root   6 Jul 18 22:31 bashrc_slink -> bashrc
#bashrc_slink 是由-s 的参数造成的，建立的是一个“快捷方式”，
#所以会看到在文件的最右边显示这个文件是“连接”到哪里去的。
#建立了 bash_hlink 之后，bashrc 与 bashrc_hlink 所有的参数都一样，
#只是，第二栏的 link 数改变成为 2，而不是原本的 1。
```

范例五：若~/.bashrc比/tmp/bashrc新才复制过来

```
[root@linux tmp]#cp -u  /.bashrc /tmp/bashrc
#-u 的特性是在目标文件与来源文件有差异时才会复制，所以常被用于“备份”的工作中。
```

范例六：将范例四产生的bashrc_slink复制成为bashrc_slink_2

```
[root@linux tmp]#cp bashrc_slink bashrc_slink_2
[root@linux tmp]#ls -l bashrc_slink*
lrwxrwxrwx  1 root root   6 Jul 18 22:31 bashrc_slink -> bashrc
-rw-r--r--  1 root root 395 Jul 18 22:48 bashrc_slink_2
#原本复制的是连接文件，但是却将连接文件的实际文件复制过来了
#也就是说，如果没有加上任何参数时，复制的是原始文件，而非连接文件的属性。
#若要复制连接文件的属性，就得要使用-d 或者-a 参数。
```

范例七：将主文件夹的.bashrc及.bash_history复制到/tmp下

```
[root@linux tmp]#cp  /.bashrc  /.bash_history /tmp
#可以将多个数据一次复制到同一个目录去。
```

不同身份的用户执行 cp 命令会有不同的结果产生，尤其是使用-a、-p 参数时。

一般来说，如果去复制别人的数据（当然必须要有 read 权限才行）时，总是希望复制到的数据最后是我们自己的，所以在默认的条件中，cp 的来源文件与目的文件的权限是不同的，目的文件的所有者通常会是指令操作者本身。

在进行备份的时候，某些需要特别注意的特殊权限文件（例如密码文件（/etc/shadow）

以及一些配置文件）就不能直接以 cp 来复制，而必须要加上-a 或者是-p 等可以完整复制文件权限的参数。

另外，如果想要复制文件给其它的使用者，也必须要注意到文件的权限（包含读、写、执行以及文件所有者等），否则其它人还是无法针对给予的文件进行修改的动作。

如果使用-l 及-s 都会建立所谓的连接文件（link file），但是这两种连接文件确有不一样的表现形式，其中-l 就是所谓的 hard link，-s 则是 symbolic link，

在使用 cp 命令进行复制时，必须要清楚的注意以下事项：

- 是否需要完整的保留来源文件的信息；
- 来源文件是否为连接文件（symbolic link file）；
- 来源文件是否为特殊的文件，例如 FIFO，socket；
- 来源文件是否为目录。

25.2.3 rm

移除文件或目录的指令 rm（remove）相当于 dos 下的 del 指令。

```
[root@linux ~]#rm [-fir] 文件或目录
参数：
-f  ：就是force的意思，强制移除；
-i  ：互动模式，在删除前会询问使用者是否执行；
-r  ：递归删除，最常用在目录的删除。
范例：
范例一：建立一文件后予以删除
[root@linux ~]#cd /tmp
[root@linux tmp]#cp  /.bashrc bashrc
[root@linux tmp]#rm -i bashrc
rm: remove regular file ‘bashrc’? y
#如果加上-i 的参数就会主动询问，如果不要询问，就加-f 参数。

范例二：删除一个不为空的目录
[root@linux tmp]#mkdir test
[root@linux tmp]#cp  /.bashrc test/ <== 将文件复制到此目录去就不是空的目录了
[root@linux tmp]#rmdir test
rmdir: ‘test’: Directory not empty <== 删不掉，因为这不是空的目录。
[root@linux tmp]#rm -rf test

范例三：删除一个带有 – 开头的文件
[root@linux tmp]#ls *aa*
```

```
-rw-r--r--  1 root  root      0 Aug 22 10:52 -aaa-
[root@linux tmp]#rm -aaa-
rm: invalid option -- a
Try 'rm --help' for more information.  <== 因为 "-" 是参数
[root@linux tmp]#rm ./-aaa-
```

通常，在 Linux 系统下在使用 rm 命令时默认有-i 参数，-i 是指每个文件被删掉之前都会让使用者确认一次，以预防误删文件。

如果要连目录下的东西都一起删除，例如子目录里面还有子目录时，那就要使用-rf 参数。不过使用 rm -rf 指令之前，该目录或文件“肯定”会被 root 删除，系统不会再次询问是否要删除，如果确定要删除干净，那可以使用 rm -rf 来循环删除。

另外，文件名最好不要使用“-”开头，因为“-”后面接的是参数，因此单纯的使用 rm -aaa-系统的指令就会误判，如果使用后面会谈到的正规表示法也还是会出问题的，所以只能用避过首位字符是“-”的方法。就是加上当前目录“./”即可，如果 man rm 的话，其实还有一种方法，那就是 rm -- -aaa-也可以。

25.2.4 mv

移动目录与文件可以使用 mv（move），mv 指令也可以直接用来进行重命名（rename）等操作。

```
[root@linux ~]#mv [-fiu] source destination
[root@linux ~]#mv [options] source1 source2 source3 .... directory
参数:
-f  : force，强制的意思，强制直接移动而不询问;
-i  : 若目标文件（destination）已经存在时，就会询问是否覆盖。
-u  : 若目标文件已经存在且source比较新，才会更新（update）。
范例:
范例一：复制一文件，建立一目录，将文件移动到目录中
[root@linux ~]#cd /tmp
[root@linux tmp]#cp  /.bashrc bashrc
[root@linux tmp]#mkdir mvtest
[root@linux tmp]#mv bashrc mvtest
#将某个文件移动到某个目录去，就是这样做。

范例二：将刚刚的目录名称更名为 mvtest2
[root@linux tmp]#mv mvtest mvtest2
#其实在 Linux 下还有 rename 命令用于重命名。
```

范例三：再建立两个文件，再全部移动到 /tmp/mvtest2 当中

```
[root@linux tmp]#cp  /.bashrc bashrc1
[root@linux tmp]#cp  /.bashrc bashrc2
[root@linux tmp]#mv bashrc1 bashrc2 mvtest2
#注意到这边，如果有多个来源文件或目录，则最后一个目标档一定是“目录”。
#意思是说，将所有的数据移动到该目录的意思。
```

mv（move）命令中也可以使用-u（update）来测试新旧文件，看看是否需要移动。

另外一个用途就是“重命名文件”，可以使用 mv 来变更一个文件的文件名。

另外，Linux 也提供了 rename 命令来更改大量文件的文件名。

25.2.5 rename

25.3 File Link

25.3.1 ln

ln 是一个用于创建链接文件（link file）的标准 Unix 命令，不同的文件名可以通过链接文件指向同一个文件。

ln 可以创建两种类型的链接文件，分别是符号连接（symbolic link）和硬连接（hard link）。

- 符号连接也称软连接，类似 Windows 的快捷方式，符号连接指向另一个不同路径文件（或目录）的一个符号路径。
- 硬连接（hard link）又称实际连接，这是一个存储了连接建立时它所指向文件的实际数据的文件副本。

hard link 的限制较多（例如无法链接目录），使用 ln 如果不加任何参数则产生的就是硬连接。

25.3.2 Hard Link

硬连接是通过文件系统的 inode 连接来产生新文件名，而不是产生新文件，因此实际上常用的是符号连接。

每个文件都会占用一个 inode，文件内容由 inode 的记录来指向。

读取文件时必须要经过目录记录的文件名来指向正确的 inode 号码才能读取。也就是说，其实文件名只与目录有关，但是文件内容则与 inode 有关。

硬连接（hard link）就是多个文件名对应到同一个 inode 号码，因此硬连接只是在某个目录下新建一个文件名连接到某个 inode 号码的关联记录。

在下面的示例中，test.jpg 和 TEST.jpg 都连接到号码为 5006751 的 inode，而且将任何一个硬连接文件名删除，其实 inode 与 block 都还是存在的。

```
[root@theqiong ~]#ll -i
total 64
5007425 lrwxrwxrwx. 1 root  root      8 Dec  9 13:59 test -> test.jpg
5006751 -rw-r------. 2 guoyu guoyu 26301 Dec  8 21:46 test.jpg
5006751 -rw-r------. 2 guoyu guoyu 26301 Dec  8 21:46 TEST.jpg
```

用户可以通过硬连接“文件名”来读取到正确的文件数据，而且最终结果都会写入到相同的 inode 和 block 中。

使用硬连接设置连接文件时，磁盘的空间与 inode 的数目都不会改变，实际上硬连接只是在某个目录下的 block 中多写入一个关联数据，因此既不会增加 inode 也不会占用 block 数量。

事实上，硬连接应该仅可以在单一文件系统中进行的，无法超越文件系统的限制。

- 硬连接不能跨文件系统；
- 硬连接不能连接到目录。

25.3.3 Symbolic Link

相对于 hard link，符号连接基本上就是在创建一个独立的文件，该文件会让数据的读取指向它连接的文件的文件名。

```
[root@theqiong ~]#ln -s test.jpg test
total 64
5007425 lrwxrwxrwx. 1 root  root      8 Dec  9 13:59 test -> test.jpg
5006751 -rw-r------. 2 guoyu guoyu 26301 Dec  8 21:46 test.jpg
5006751 -rw-r------. 2 guoyu guoyu 26301 Dec  8 21:46 TEST.jpg
```

从以下命令示例可看出两种链接文件的区别，当原始文件被删除后，符号链接将失效，因此访问软链接时会提示找不到文件，但硬链接文件还在，而且还保存有原始文件的内容。

```
$ echo '文件内容' > oringinal.file
$ ln oringinal.file hardlink.file
$ ln -s oringinal.file softlink.file
$ cat softlink.file
文件内容
$ rm oringinal.file
$ cat softlink.file
cat: softlink.file: 没有那个文件或目录
$ cat hardlink.file
```

文件内容

单一 Unix 规范（SUS）规定了创建一个原始文件（或目录）的链接（不管是符号链接还是硬链接）文件时的行为。

ln 可以用两种方式使用，其中：

- 第一个参数指定原始文件，第二个参数指定链接文件；
- 指定多于两个选项，应该先是多个原始文件（或目录），最后指定一个目录，所有原始文件（或目录）的链接将会被创建于最后指定的目录里。

在后一种方式中，ln 命令的行为和具体的程序实现有关。

ln 和标准的 unlink() 和 link() 函数执行完全一致的操作。

25.4 File Name

完整文件名（包含目录名称与文件名称）最长可以到达 4096 个字符，其实 basename 与 dirname 可以用来取得文件名或者是目录名称。

25.4.1 basename

```
$ basename /etc/sysconfig/network
network
```

25.4.2 dirname

```
$ dirname /etc/sysconfig/network
/etc/sysconfig
```

25.5 File Content

如果要查看一个文件的内容，最常使用的显示文件内容的指令是 cat、more 及 less。此外，如果要查看一个大型的文件，但是我们只需要后端的几行字而已，可以使用 tail，此外使用 tac 这个指令也可以完成。

- cat：由第一行开始显示文件内容；
- tac：从最后一行开始显示；
- nl：显示的时候输出行号；
- more：一页一页的显示文件内容；
- less：less 与 more 类似，但是比 more 更好的是，less 可以往前翻页；

- head：只看头几行；
- tail：只看尾巴几行；
- od：以二进位的方式读取文件内容。

直接查看一个文件的内容可以使用 cat/tac/nl 这几个命令。

25.5.1 cat

cat 是 Concatenate（连续）的简写，主要的功能是将一个文件的内容连续的显示出在显示器上。

如果加上-n，则每一行前面还会加上行号。当文件内容的行数超过 40 行以上，仅使用 cat 根本来不及看，所以配合 more 或者 less 来执行会比较好。

此外，如果是一般的 DOS 文件时，就需要特别留意一些奇怪的符号了，例如断行与 [tab] 等，要显示出来就得加入-A 之类的参数。

25.5.2 tac

tac 刚好是将 cat 反写过来，所以 tac 的功能也跟 cat 相反，cat 是由“第一行到最后一行连续显示在显示器上”，而 tac 则是“由最后一行到第一行反向在显示器上显示出来”。

```
[root@linux ~]#tac /etc/issue
Kernel \r on an \m
Fedora Core release 4 (Stentz)
#与刚刚上面的范例一比较，是由最后一行先显示。
```

25.5.3 nl

nl 可以在输出文件内容时添加行号。

```
[root@linux ~]#nl [-bnw] 文件
参数:
-b : 指定行号指定的方式，主要有两种
     -b a : 表示不论是否为空行，也同样列出行号;
     -b t : 如果有空行，空的那一行不要列出行号;
-n : 列出行号表示的方法，主要有三种
     -n ln : 行号在显示器的最左方显示;
     -n rn : 行号在自己栏位的最右方显示，且不加0;
     -n rz : 行号在自己栏位的最右方显示，且加0;
-w : 行号栏位的占用的位数。
范例:
```

范例一：列出/etc/issue的内容

```
[root@linux ~]#nl /etc/issue
     1  Fedora Core release 4  (Stentz)
     2  Kernel \r on an \m

#注意看，这个文件其实有三行，第三行为空白（没有任何字符），
#因为它是空白行，所以 nl 不会加上行号，如果确定要加上行号，可以这样做：
[root@linux ~]#nl -b a /etc/issue
     1  Fedora Core release 4  (Stentz)
     2  Kernel \r on an \m
     3
#如果要让行号前面自动补上 0，可以这样
[root@linux ~]#nl -b a -n rz /etc/issue
000001  Fedora Core release 4  (Stentz)
000002  Kernel \r on an \m
000003
#自动在自己栏位的地方补上 0，默认栏位是六位数，如果想要改成 3 位数，可以这样
[root@linux ~]#nl -b a -n rz -w 3 /etc/issue
001     Fedora Core release 4  (Stentz)
002     Kernel \r on an \m
003
#变成仅有 3 位数
```

nl 可以将输出的文件内容自动加上行号，其结果与 cat -n 不太一样，nl 可以将行号做比较多的显示设计，包括位数与是否自动补齐 0 等。

25.5.4 more

nl、cat、tac 等都是一次性的将数据显示到显示器上面，如果要一页一页翻页查看，可以使用 more 与 less 命令。

more 命令用于按页来显示文件内容。

```
[root@linux ~]#more /etc/man.config
#Generated automatically from man.conf.in by the
#configure script.
#man.conf from man-1.5p
#
.......中间省略.......
--More--（28%）  <== 重点在这一行。
```

仔细的看上面的范例，如果 more 后面接的文件长度大于显示器输出的行数时，就会出现类似上面的图示。

重点在最后一行，最后一行会显示出目前显示的百分比，而且还可以在最后一行输入一些有用的指令。在 more 这个程序的运作过程中，有几个快捷键如下：

- 空格键（space）：代表向下翻一页；
- Enter：代表向下翻“一行”；
- /字符串：代表在这个显示的内容中向下搜索“字符串”；
- :f：立刻显示出文件名以及目前显示的行数；
- q：代表立刻离开 more，不再显示该文件内容。

比较有用的是搜索字符串的功能，举例来说，使用 `more /etc/man.config` 来测试该文件，若想要在该文件内搜索 MANPATH 这个字符串，可以这样做：

```
[root@linux ~]#more /etc/man.config
#
#Generated automatically from man.conf.in by the
#configure script.
#man.conf from man-1.5p
#
.......中间省略.......
/MANPATH    <== 输入/之后，光标就会自动跑到最底下一行等待输入。
```

如同上面的说明，输入/之后光标就会跑到最下面一行并等待输入，输入了字符串之后，more 就会开始向下搜索该字符串，而重复搜索同一个字符串，可以直接按下 n。最后不想要看了，就按下 q 即可离开 more。

25.5.5 less

less 的用法比起 more 又更加弹性，在使用 more 命令时用户并没有办法向前面翻，只能往后面看。

使用 less 命令时，可以使用 [pageup]、[pagedown] 等按键的功能来往前往后翻看文件。

```
[root@linux ~]#less /etc/man.config
#Generated automatically from man.conf.in by the
#configure script.
#
#man.conf from man-1.5p
......中间省略........
:    <== 这里可以等待输入指令。
```

除此之外，在 less 里可以拥有更多的“搜索”功能，不仅可以向下搜索，也可以向上搜索，基本上可以输入的指令有：

- 空格键：向下翻动一页；

pagedown ：向下翻动一页；

pageup ：向上翻动一页；

- /字符串：向下搜索“字符串”的功能；
- ? 字符串：向上搜索“字符串”的功能；
- n：重复前一个搜索（与/或? 有关）；
- N：反向的重复前一个搜索（与/或? 有关）；
- q：离开 less。

使用 less 查看文件内容还可以进行搜索，其实 less 还有很多的功能，而且 less 的使用界面和环境与 man page 很类似，并且实际上 man 这个命令就是调用 less 来显示说明文件的内容的。

25.6 Data Selection

25.6.1 head

可以将输出的数据作一个最简单的选取，那就是取出前面（head）与取出后面（tail）的文字，不过要注意 head 与 tail 都是以“行”为单位来进行数据选取的。

head（取出前面几行）

```
[root@linux ~]#head [-n number] 文件
```

参数：

—n ：后面接数字，代表显示几行的意思

范例：

```
[root@linux ~]#head /etc/man.config
#默认的情况中，显示前面十行。若要显示前 20 行，就要这样：
[root@linux ~]#head -n 20 /etc/man.config
```

head 的英文意思就是“头”，那么 head 的用法自然就是显示出一个文件的前几行，若没有加上-n 这个参数时，默认只显示 10 行，若只要一行，那就使用 `head -n 1 filename` 即可。

25.6.2 tail

tail（取出后面几行）

```
[root@linux ~]#tail [-n number] 文件
```

参数:
-n : 后面接数字，代表显示几行的意思
范例:

```
[root@linux ~]#tail /etc/man.config
#默认的情况中，显示最后的 10 行，若要显示最后的 20 行，就得要这样:
[root@linux ~]#tail -n 20 /etc/man.config
```

tail 的用法跟 head 的用法类似，只是显示的是后面几行，tail 默认也是显示 10 行，若要显示非 10 行，就加`-n number`参数。

假如想要显示`~/.bashrc`的第 11 到第 20 行，那么可以先取前 20 行，再取后 10 行，所以结果就是

```
head -n 20 ~/.bashrc | tail -n 10
```

这样就可以得到第 11 到第 20 行之间的内容，这里涉及到管道命令。

25.6.3 od

如果想要查看非文本文件，举例来说，例如/usr/bin/passwd 这个执行文件的内容时，由于可执行文件通常是 binary file，使用上面提到的指令来读取其内容时确实会产生类似乱码的数据，这时可以利用 od 这个指令来读取。

```
[root@linux ~]#od [-t TYPE] 文件
```

参数:
-t : 后面可以接各种“类型 (TYPE) ”的输出，例如:
    a : 利用默认的字符来输出;
    c : 使用ASCII字符来输出
    d[size] : 利用十进位 (decimal) 来输出数据，每个整数占用size bytes;
    f[size] : 利用浮点数值 (floating) 来输出数据，每个数占用size bytes;
    o[size] : 利用八进位 (octal) 输出数据，每个整数占用size bytes;
    x[size] : 利用十六进位 (hexadecimal) 输出数据，每个整数占用size bytes;
范例:

```
[root@linux ~]#od -t c /usr/bin/passwd
0000000 177   E   L   F 001 001 001  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000020 002  \0 003  \0 001  \0  \0  \0 260 225 004  \b   4  \0  \0  \0
0000040 020   E  \0  \0  \0  \0  \0  \0   4  \0      \0  \a  \0    (  \0
0000060 035  \0 034  \0 006  \0  \0  \0   4  \0  \0  \0   4 200 004  \b
0000100   4 200 004  \b 340  \0  \0  \0 340  \0  \0  \0 005  \0  \0  \0
.....中间省略.....
```

利用 od 指令，可以将 data file 或者是 binary file 的内容数据读出来，虽然读出的数值默认是使用非文本形式，也就是是 16 进位的数值来显示的，不过还是可以通过`-t c` 的参数来将数据内的字符以 ASCII 类型的字符来显示，虽然对于一般使用者来说这个指令的用处可能不大，但是对于工程师来说这个指令可以将 binary file 的内容作一个大致的输出，它们可以看出其中的含义。

25.6.4 touch

每个文件在 linux 下都会记录三个主要的时间参数，分别是：

- modification time（mtime）：当该文件的“内容数据”变更时，就会更新这个时间，内容数据指的是文件的内容，而不是文件的属性。
- status time（ctime）：当该文件的“状态（status)”改变时，就会更新这个时间，比如权限与属性被更改了，都会更新这个时间啊。
- access time（atime）：当“该文件的内容被取用”时，就会更新这个读取时间（access)，比如用 cat 去读取 ~/.bashrc，就会更新 atime。

先来看一看您自己的/etc/man.config 的时间参数：

```
[root@linux ~]#ls -l /etc/man.config
-rw-r--r--  1 root root 4506 Apr  8 19:11 /etc/man.config
[root@linux ~]#ls -l -time=atime /etc/man.config
-rw-r--r--  1 root root 4506 Jul 19 17:53 /etc/man.config
[root@linux ~]#ls -l -time=ctime /etc/man.config
-rw-r--r--  1 root root 4506 Jun 25 08:28 /etc/man.config
```

在默认的情况下，ls 显示出来的是该文件的 mtime，也就是这个文件的内容上次被更改的时间。

touch 可以用来更改文件访问和修改时间，也被用于创建新文件。

单一 UNIX 规范包含下列程序选项：

`-a`，只更改访问时间
`-c`，如果文件不存在，不创建且不声明
`-m`，只更改修改时间
`-r file`，使用`file`的访问、修改时间而非当前时间
`-t time`，使用`time`（格式见下）更改访问、修改时间

time 的格式为 [[cc]yy]MMDDhhmm[.ss]，其中 cc 代表世纪，yy 代表年份的后二位数字，MM 代表月份，DD 代表天数，hh 代表小时，mm 代表分钟，ss 代表秒数。其他 Unix 系统或类 Unix 系统可能添加额外的选项。

touch 不修改文件的内容，仅修改文件的访问和修改时间，如果文件不存在，它会被创建。例如，以当前时间更改访问、修改时间：

```
$ touch myfile.txt
```

用指定时间更改访问、修改时间：

```
$ touch -t 200701310846.26 index.html
$ touch -d '2007-01-31 8:46:26' index.html
$ touch -d 'Jan 31 2007 8:46:26' index.html
```

```
[root@linux ~]#touch [-acdmt] 文件
参数:
-a  : 仅修改access time;
-c  : 仅修改时间，而不建立文件;
-d  : 后面可以接日期，也可以使用--date="日期或时间"
-m  : 仅修改mtime;
-t  : 后面可以接时间，格式为[YYMMDDhhmm]

范例一: 新建一个空的文件
[root@linux ~]#cd /tmp
[root@linux tmp]#touch testtouch
[root@linux tmp]#ls -l testtouch
-rw-r--r--  1 root root    0 Jul 19 20:49 testtouch
#注意这个文件的大小是 0。默认状态下如果 touch 后面有接文件，
#则该文件的三个时间 (atime/ctime/mtime) 都会更新为目前的时间。
#若该文件不存在，则会主动的建立一个新的空的文件。

范例二: 将~/.bashrc复制成为bashrc，假设复制完全的属性，检查其日期
[root@linux tmp]#cp  /.bashrc bashrc
[root@linux tmp]#ll bashrc; ll -time=atime bashrc; ll -time=ctime bashrc
-rwxr-xr-x  1 root root 395 Jul  4 11:45 bashrc  <==这是 mtime
-rwxr-xr-x  1 root root 395 Jul 19 20:44 bashrc  <==这是 atime
-rwxr-xr-x  1 root root 395 Jul 19 20:53 bashrc  <==这是 ctime
#在这个案例当中，使用了；这个指令分隔符号。
#此外，ll 是 ls -l 的命令别名，这个也会在 Bash shell 中再次提及，
#目前可以简单的想成，ll 就是 ls -l 的简写即可，至于；则是同时执行两个指令，
#且让两个指令“按顺序”执行的意思。上面的结果当中可以看到该文件变更的日期
```

```
#Jul 4 11:45, 但是 atime 与 ctime 不一样
```

范例三：修改案例二的bashrc文件，将日期调整为两天前

```
[root@linux tmp]#touch -d "2 days ago" bashrc
[root@linux tmp]#ll bashrc; ll -time=atime bashrc; ll -time=ctime bashrc
-rwxr-xr-x  1 root root 395 Jul 17 21:02 bashrc
-rwxr-xr-x  1 root root 395 Jul 17 21:02 bashrc
-rwxr-xr-x  1 root root 395 Jul 19 21:02 bashrc
#跟上个范例比较看看，本来是 19 日的变成了 17 日了 (atime/mtime)。
#不过 ctime 并没有跟着改变。
```

范例四：将上个范例的bashrc日期改为 2005/07/15 2:02

```
[root@linux tmp]#touch -t 0507150202 bashrc
[root@linux tmp]#ll bashrc; ll -time=atime bashrc; ll -time=ctime bashrc
-rwxr-xr-x  1 root root 395 Jul 15 02:02 bashrc
-rwxr-xr-x  1 root root 395 Jul 15 02:02 bashrc
-rwxr-xr-x  1 root root 395 Jul 19 21:05 bashrc
#注意看看，日期在 atime 与 mtime 都改变了，但是 ctime 则是记录目前的时间。
```

通过 touch 命令可以轻易的修订文件的日期与时间，并且也可以建立一个空的文件。不过要注意的是，复制一个文件时即使复制所有的属性，也没有办法复制 ctime 这个属性。ctime 可以记录这个文件最近的状态（status）被改变的时间。无论如何还是要说明的是，平时看的文件属性中比较重要的还是属于那个 mtime。

无论如何，touch 命令最常被使用的情况是：

- 建立一个空的文件；
- 将某个文件日期修订为目前（mtime 与 atime）。

其他操作系统（例如 Windows 等）也存在执行相似功能的软件，例如 File Date Touch。

25.7 Default Permission

在 Linux 中每个文件都有若干个属性，包括（r，w，x）等基本属性以及是否为目录（d）与文件（-）或者是连接文件（l）等的属性。

除了基本 r，w，x 权限外，在 Linux 的 ext2/3 文件系统下还可以设定其它的系统隐藏权限，使用 chattr 来设定并以 lsattr 来查看，其中最重要的属性就是可以设定其不可修改的特性，连文件的所有者（owner）都不能进行修改，这个属性可是相当重要的，尤其是在安全机制上面（security）。

25.7.1 umask

当建立一个新的文件或目录时，其默认属性与 umask 有关。

基本上，umask 就是指定“目前用户在建立文件或目录时的权限默认值”，那么如何得知或设定 umask 呢？umask 的指定条件可以以下面的的方式来指定：

```
[root@linux ~]#umask
0022
[root@linux ~]#umask -S
u=rwx,g=rx,o=rx
```

查看的方式有两种，一种可以直接执行 umask，就可以看到数字型态的权限设定分数，一种则是加入-S（Symbolic）参数，就会以符号类型的方式来显示出权限。

奇怪的是，怎么 umask 会有四组数字？第一组是特殊权限用的，所以先看后面三组即可。

在默认权限的属性上，目录与文件是不一样的。由于用户不希望文件具有可执行的权力，默认情况下文件是没有可执行（x）权限的。

1、若使用者建立为“文件”则默认“没有可执行（x）权限”，也就是只有 rw 这两个项目，也就是最大为 666，默认属性如下：

```
-rw-rw-rw-
```

2、若使用者建立为“目录”，则由于 x 与是否可以进入此目录有关，因此默认为所有权限均开放，也就是为 777，默认属性如下：

```
drwxrwxrwx
```

umask 指定的是“该默认值需要减掉的权限”，因为 r、w、x 分别是 4、2、1，所以当要去掉可写入的权限，就是输入 2，而如果要去掉读取的权限，也就是 4，那么要去掉读与写的权限，也就是 6，而要去掉执行与写入的权限，也就是 3。

如果以上面的例子来说明，因为 umask 为 022，所以 user 并没有被去掉属性，不过 group 与 others 的属性被去掉了 2（也就是 w 属性），那么由于当使用者：

- 建立文件时：（-rw-rw-rw-）-（—-w-w-）==> -rw-r-r-
- 建立目录时：（drwxrwxrwx）-（d—-w-w-）==> drwxr-xr-x

```
[root@linux ~]#umask
0022
[root@linux ~]#touch test1
[root@linux ~]#mkdir test2
[root@linux ~]#ll
-rw-r--r--  1 root root    0 Jul 20 00:36 test1
drwxr-xr-x  2 root root 4096 Jul 20 00:36 test2
```

假如想要让与使用者同群组的人也可以存取文件，也就是说，假如 dmtsai 是 users 这个群组的人，而 dmtsai 文件希望让 users 同群组的人也可以存取，这也是常常被用在团队开发计划时会考虑到的权限问题。

在这样的情况下，umask 自然不能取消 group 的 w 权限，也就是说希望文件应该是-rw-rw-r--，所以 umask 应该是要 002 才行（仅去掉 others 的 w 权限），设定 umask 时直接在 umask 后面输入 002 就可以。

```
[root@linux ~]#umask 002
[root@linux ~]#touch test3
[root@linux ~]#mkdir test4
[root@linux ~]#ll
-rw-rw-r--  1 root root    0 Jul 20 00:41 test3
drwxrwxr-x  2 root root 4096 Jul 20 00:41 test4
```

这个 umask 对于文件与目录的默认权限是很有关系的，这个概念可以用在任何服务器上面，尤其是未来在架设文件服务器（file server）时，比如 SAMBA Server 或 FTP Server 时，umask 都是很重要的观念，这牵涉到用户是否能够将文件进一步利用的问题。

在默认的情况中，root 的 umask 会去掉比较多的属性，root 的 umask 默认是 022，这是基于安全的考虑。

一般身份用户的 umask 通常为 002，也就是保留同群组的写入权力。其实关于默认 umask 的设定可以参考/etc/bashrc 这个文件的内容，不过不建议修改该文件，可以参考 bash shell 提到的环境参数配置文件（~/.bashrc）的说明。

25.8 Hidden Permission

25.8.1 chattr

文件的隐藏属性确实对于系统有很大的帮助的，尤其是在系统安全（Security）上。

1、chattr（设定文件隐藏属性）

```
[root@linux ~]#chattr [+-=][ASacdistu] 文件或目录名称
```

参数:

+ ：增加某一个特殊参数，其它原本存在参数则不动。

− ：移除某一个特殊参数，其它原本存在参数则不动。

= ：设定一定，且仅有后面接的参数

A ：当设定了A这个属性时，这个文件（或目录）的存取时间atime（access）将不可被修改，可避免例如笔记本电脑容易有磁盘I/O错误的情况发生。

S ：这个功能有点类似sync的功能，就是会将数据同步写入磁盘当中，可以有效的避免数据丢失。

a ：当设定a之后，这个文件将只能增加数据而不能删除，只有root才能设定这个属性。

c ：这个属性设定之后，将会自动的将此文件“压缩”，在读取的时候将会自动

解压缩，但是在存储的时候，将会先进行压缩后再存储（对于大文件似乎有用）。

d ：当dump（备份）程序被执行的时候，设定d属性将可使该文件（或目录）不具有dump功能。

i ：i可以让一个文件“不能被删除、改名、设定连结也无法写入或新增数据”，对于系统安全性有相当大的助益。

j ：当使用ext3这个文件系统格式时，设定j属性将会使文件在写入时先记录在journal中，但是当filesystem设定参数为data=journalled时，由于已经设定了日志了，所以这个属性无效。

s ：当文件设定了s参数时，它将会被完全的移除出这个硬盘空间。

u ：与s相反的，当使用u来设定文件时，则数据内容其实还存在磁盘中，可以使用来undeletion。

这个属性设定上比较常见的是a与i的设定值，很多设定值必须要root才能够设定。

范例：

```
[root@linux ~]#cd /tmp
[root@linux tmp]#touch attrtest
[root@linux tmp]#chattr +i attrtest
[root@linux tmp]#rm attrtest
rm: remove write-protected regular empty file ‘attrtest’? y
rm: cannot remove ‘attrtest’: Operation not permitted
#连 root 也没有办法将这个文件删除。
[root@linux tmp]#chattr -i attrtest
```

这个指令是重要的，尤其是在系统的安全性上。这些属性是隐藏的，需要以 lsattr 才能看到，其中最重要的当属 +i 这个属性了，因为它可以让一个文件无法被更改，对于需要强烈的系统安全的人来说是相当的重要的，还有相当多的属性是需要 root 才能设定。

此外，如果是 log file 这种登录文件就更需要 +a 这个可以增加但是不能修改旧有的数据与删除的参数。

25.8.2 lsattr

2、lsattr （显示文件隐藏属性）

```
[root@linux ~]#lsattr [-aR] 文件或目录
```

参数：

-a ：将隐藏文件的属性也秀出来；

-R ：连同子目录的数据也一并列出来。

范例：

```
[root@linux tmp]#chattr +aij attrtest
```

```
[root@linux tmp]#lsattr
----ia---j--- ./attrtest
```

使用 chattr 设定后，可以利用 lsattr 来查看隐藏的属性。不过这两个指令在使用上必须要特别小心，否则会造成很大的困扰。

25.9 SUID/SGID/Sticky Bit

查看/tmp 和/usr/bin/passwd 的权限时，会看到如下的结果：

```
[root@linux ~]#ls -ld /tmp ; ls -l /usr/bin/passwd
drwxrwxrwt  5 root root 4096  Jul 20 10:00 /tmp
-r-s--x--x  1 root root 18840 Mar  7 18:06 /usr/bin/passwd
```

25.9.1 Set UID

Linux 中的 s 与 t 的权限与系统的账号及系统的进程较为相关，它们可以让一般用户在执行某些程序时能暂时的具有该程序所有者的权限。

w 举例来说，所有帐号的密码都记录在/etc/shadow 中，而/etc/shadow 这个文件的权限是“----------”，而且它的所有者是 root，在这个权限中仅有 root 可以“强制”存储，其它人是连看都没有办法看的。

当 s 这个标志出现在文件所有者的 x 权限上时，例如/usr/bin/passwd 这个文件的权限状态“-rwsr-xr-x”，此时就被称为 Set UID，简称为 SUID 的特殊权限。由上面的定义中知道，当 vbird 这个用户执行/usr/bin/passwd 时，它就会“暂时”的得到文件所有者 root 的权限，这里 UID 代表的是 User 的 ID，而 User 代表的则是这个程序（/usr/bin/passwd）的所有者（root）。

- SUID 仅可用在“二进制文件（binary file）”上，SUID 因为是程序在执行的过程中拥有文件所有者的权限，因此它仅可用于 binary file，不能够用在脚本（shell script），这是因为 shell script 只是将很多的 binary 执行文件调用进来执行，所以 SUID 的权限部分还是得要看 shell script 调用进来的程序的设定而不是 shell script 本身。当然 SUID 对于目录也是无效的。
- 执行者对于该程序需要具有 x 的可执行权限。
- s 权限仅在执行该程序的过程（run-time）中有效。
- 执行者将具有该程序所有者（owner）的权限。

在 Linux 系统中，所有账号的密码都记录在/etc/shadow 这个文件里，这个文件的权限为“---------- 1 root root”，那么 vbird 这个一般用户账号是无法访问这个文件的，但是每个用户都是可以自己修改自己的密码的，而这就是 s 权限的用途，也就是说 vbird 这个一般身份使用者可以修改/etc/shadow 这个文件内的密码。

通过上述说明，可以知道：

- vbird 对于/usr/bin/passwd 这个程序来说是具有 x 权限的，表示 vbird 能执行 passwd；
- passwd 的拥有者是 root 这个账号；
- vbird 执行 passwd 的过程中会“暂时”获得 root 的权限；
- /etc/shadow 可以被 vbird 所执行的 passwd 所修改。

而如果 vbird 执行“`cat /etc/shadow`”时，仍然是不能读取/etc/shadow 的，可以用下面的示意图来说明如下：

25.9.2 Set GID

进一步来说，如果 s 的权限是在 group 时就是 Set GID，简称为 SGID。与 SUID 不同的是，SGID 可以针对文件或目录来设置。

1、若针对文件，SGID 有如下的功能：

- SGID 对二进制程序有用；
- 程序执行者对于该程序来说，需具备 x 的权限；
- 如果 SGID 是设定在 binary file 上，则不论使用者是谁，在执行该程序的时候，它的有效用户组（effective group）将会获得该程序的用户组（group id）的支持。

举例来说，先查看具有 SGID 权限的文件：

```
[root@linux ~]#ls -l /usr/bin/locate
-rwx--s--x  1  root  slocate    23856  Mar 15 2007   /usr/bin/locate
```

而/usr/bin/locate 这个程序可以去查询/var/lib/mlocate/mlocate.db 这个文件的内容，mlocate.db 的权限如下：

```
[root@linux ~]#ll /var/lib/mlocate/mlocate.db
-rw-r-----  1  root  slocate  3175776  Sep 28 04:02  /var/lib/mlocate.db
```

与 SGID 非常类似，若使用 vbird 这个账号去执行 locate 时，vbird 将会取得 slocate 用户组的支持，于是就能够读取 mlocate.db。

2、若针对目录，SGID 有如下的功能：

- 如果 SGID 是设定在 A 目录上，则在该 A 目录内所建立的文件或目录的 group 将会是这个 A 目录的 group。
- 用户若对于此目录具有 r 与 x 的权限时，该用户能够进入此目录；
- 用户在此目录下的有效用户组（effective group）将会变成该目录的用户组；
- 若用户在此目录下具有 w 的权限（可以新建文件），则用户所创建的新文件的用户组与此目录的用户组相同。

一般来说，SGID 应该是比较多用在特定的多人团队的项目开发上，因为会涉及到用户组权限的问题。

25.9.3 Sticky Bit

Sticky Bit（SBIT）目前只针对目录有效，对于文件已经没有效果了。SBIT 对于目录的作用是：

- 当用户对于此目录具有 w、x 权限，就具有写入的权限。
- 在具有 SBit 的目录下，使用者若在该目录下具有 w 及 x 权限，则当使用者在该目录下建立文件或目录时，只有文件所有者与 root 才有权力删除。
- 换句话说，当甲这个用户在 A 目录下是拥有 group 或者是 other 的项目，并且拥有 w 的权限，这表示“甲使用者对该目录内任何人建立的目录或文件均可进行删除/更名/搬移等动作。”不过，如果将 A 目录加上了 Sticky Bit 的权限项目时，则甲只能够针对自己建立的文件或目录进行删除/更名/移动等动作，而无法删除别人的文件。

举例来说，/tmp 本身的权限是 `drwxrwxrwt`，在这样的权限下任何人都可以在/tmp 内新增、修改文件，但仅有该文件/目录建立者与 root 能够删除自己的目录或文件，这个特性也是挺重要的，可以这样做个简单的测试：

（1）以 root 登录系统，并且进入/tmp 中；

（2）执行 `touch test`，并且更改 test 权限成为 777；

（3）以一般使用者登录，并进入/tmp；

（4）尝试删除 test 这个文件。

25.9.4 SUID/SGID/SBIT Permission

数字型态个更改权限方式为“三个数字”的组合，那么如果在这三个数字之前再加上一个数字，那最前的面数字就代表这几个属性。（注：通常我们使用 `chmod xyz filename` 的方式来设定 filename 的属性时，则是假设没有 SUID、SGID 及 Sticky Bit）

- 4 为 SUID；
- 2 为 SGID；
- 1 为 Sticky Bit。

假设要将一个文件属性改为`-rwsr-xr-x` 时，由于 s 在用户权限中，所以是 SUID，因此在原先的 755 之前还要加上 4 ，也就是使用：

```
chmod 4755 filename
```

来设定。

此外还有大 S 与大 T 的产生，参考底下的范例，注意底下的范例只是练习，所以使用同一个文件来设定，首先必须了解 SUID 不是用在目录上，而 SBIT 不是用在文件上。

```
[root@linux ~]#cd /tmp
[root@linux tmp]#touch test
[root@linux tmp]#chmod 4755 test; ls -l test
```

```
-rwsr-xr-x  1 root root 0 Jul 20 11:27 test
[root@linux tmp]#chmod 6755 test; ls -l test
-rwsr-sr-x  1 root root 0 Jul 20 11:27 test
[root@linux tmp]#chmod 1755 test; ls -l test
-rwxr-xr-t  1 root root 0 Jul 20 11:27 test
[root@linux tmp]#chmod 7666 test; ls -l test
-rwSrwSrwT  1 root root 0 Jul 20 11:27 test
#这个例子就要特别小心，怎么会出现大写的 S 与 T? 不都是小写的吗?
#因为 s 与 t 都是取代 x 这个参数的，但是有没有发现我们执行的是 7666，
#也就是说，user、group 以及 others 都没有 x 这个可执行的标志（因为 666），
#所以这个 S、T 代表的就是“空的”。
#SUID 是表示“该文件在执行的时候具有文件所有者的权限”，但是文件
#所有者都无法执行了，哪里来的权限给其它人使用，当然就是空的。
```

25.10 File Type

25.10.1 file

如果想要知道某个文件的基本数据，例如是属于 ASCII 还是 data 文件，或者是 binary，且其中有没有使用到动态函数库（share library）等的信息，可以利用 file 指令来查看，举例来说：

```
[root@linux ~]#file  /.bashrc
/root/.bashrc: ASCII text
[root@linux ~]#file /usr/bin/passwd
/usr/bin/passwd: setuid ELF 32-bit LSB executable, Intel 80386, version 1
 (SYSV) , for GNU/Linux 2.2.5, dynamically linked  (uses shared libs) ,
    stripped
[root@linux ~]#file /var/lib/slocate/slocate.db
/var/lib/slocate/slocate.db: data
```

通过 file 指令可以简单的判断文件的格式。

25.10.2 cksum

25.11 Printer

25.11.1 lp

25.11.2 lpr

25.12 Scanner

25.12.1 simple-scan

25.13 File Comparison

25.13.1 cmp

cmp is a command line utility for computer systems that use Unix or a Unix-like operating system. It compares two files of any type and writes the results to the standard output. By default, cmp is silent if the files are the same; if they differ, the byte and line number at which the first difference occurred is reported.

- 0 —files are identical
- 1 —files differ
- 2 —inaccessible or missing argument

cmp may be qualified by the use of command-line switches. The switches supported by the GNU version of cmp are:

- -b, –print-bytes
  Print the differing bytes. Display control bytes as a ‘ˆ’ followed by a letter of the alphabet and precede bytes that have the high bit set with ’M-’ (which stands for ”meta”).
- -i SKIP, –ignore-initial=SKIP
  Skip the first SKIP bytes of input.
- -i SKIP1:SKIP2, –ignore-initial=SKIP1:SKIP2
  Skip the first SKIP1 bytes of FILE1 and the first SKIP2 bytes of FILE2.
- -l, –verbose
  Output the (decimal) byte numbers and (octal) values of all differing bytes, instead of the default standard output. Also, output the EOF message if one file is shorter than the other.
- -n LIMIT, –bytes=LIMIT
  Compare at most LIMIT bytes.
- -s, –quiet, –silent

Output nothing; yield exit status only.

- -v, –version
  Output version info.
- –help
  Outputs a help file.

Operands that are byte counts are normally decimal, but may be preceded by '0' for octal and '0x' for hexadecimal.

A byte count can be followed by a suffix to specify a multiple of that count; in this case an omitted integer is understood to be 1. A bare size letter, or one followed by 'iB', specifies a multiple using powers of 1024. A size letter followed by 'B' specifies powers of 1000 instead. For example, '-n 4M' and '-n 4MiB' are equivalent to '-n 4194304', whereas '-n 4MB' is equivalent to '-n 4000000'. This notation is upward compatible with the SI prefixes[1] for decimal multiples and with the IEC 60027-2 prefixes for binary multiples.

25.14 File Search

我们经常需要知道哪个文件放在哪里才能对该文件进行一些修改或维护等操作，虽然文件名通常不变，但是不同的 Linux distribution 放置的目录可能不同，就需要使用 Linux 的搜索系统来将文件的完整文件名找出来，通常 find 不很常用，除了速度慢之外也很消耗硬盘。通常我们都是先使用 whereis 或者是 locate 来检查，如果真的找不到了才以 find 来搜索，whereis 与 locate 是利用数据库来搜索数据，所以相当的快速，而且并没有实际的搜索硬盘，比较省时间。

25.14.1 which

```
[root@linux ~]#which [-a] command
参数:
-a : 将所有可以找到的指令均列出，而不止第一个被找到的指令名称
范例:
[root@linux ~]#which passwd
/usr/bin/passwd
[root@linux ~]#which traceroute -a
/usr/sbin/traceroute
/bin/traceroute
```

这个指令是根据“PATH”这个环境变量所规范的路径去搜索“执行文件”的文件名，所以重点是找出“执行文件”且 which 后面接的是“完整文件名”。若加上-a 参数，则可以

列出所有的可以找到的同名执行文件而非仅显示第一个而已。

25.14.2 whereis

```
[root@linux ~]#whereis [-bmsu] 文件或目录名
参数:
-b   : 只找binary的文件;
-m   : 只找在说明文件manual路径下的文件;
-s   : 只找source来源文件;
-u   : 没有说明文件的文件。
范例:
[root@linux ~]#whereis passwd
passwd: /usr/bin/passwd /etc/passwd /etc/passwd.OLD
/usr/share/man/man1/passwd.1.gz /usr/share/man/man5/passwd.5.gz
#任何与 passwd 有关的文件名都会被列出来。
[root@linux ~]#whereis -b passwd
passwd: /usr/bin/passwd /etc/passwd /etc/passwd.OLD
[root@linux ~]#whereis -m passwd
passwd: /usr/share/man/man1/passwd.1.gz /usr/share/man/man5/passwd.5.gz
```

find 是很强大的搜索指令，但 find 是直接搜索硬盘，所以时间消耗很大。

whereis 可以加入参数来寻找相关的数据，例如查找可执行文件（binary）时可以加上-b。

25.14.3 locate

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

locate 的使用更简单，直接在后面输入“文件的部分名称”后就能够得到结果。继续上面的例子，输入 `locate passwd`，那么在完整文件名（包含路径名称）中只要有 passwd 就会被显示出来，如果忘记某个文件的完整文件名时，这也是个很方便好用的指令。

但是 locate 还是有使用上的限制，之所以使用 locate 查找数据的时候特别快是因为 locate 查找的数据是由“已建立的数据库/var/lib/slocate/”里面的数据所搜索到的，所以不用直接在去硬盘当中存取数据。

locate 也有一定的限制，因为 locate 是通过数据库来搜索，而数据库的建立默认是在每天执行一次（每个 distribution 都不同），所以对于新建立起来的文件而在数据库更新之前搜索该文件的话，locate 会“找不到该文件”。

使用 locate 时，可以自己选择需要建立文件数据库的目录，可以在/etc/updatedb.conf 文件内设定，建议使用默认值即可。不过在/etc/updatedb.conf 里面可以把 `DAILY_UPDATE=no` 改成 `DAILY_UPDATE=yes`，当然也可以自行手动执行 updatedb 即可。

25.14.4 updatedb

Linux 系统会将系统内的所有文件都记录在一个数据库文件里面，当使用 whereis 或者是 locate 时，都会以此数据库文件的内容为准，因此有的时候还会发现使用这两个执行文件时会找到已经被删除的文件，而且也找不到最新的刚刚建立的文件，这就是因为这两个指令是由数据库当中的结果去搜索文件的所在位置。

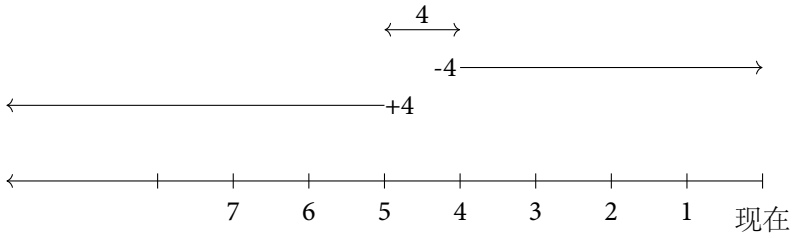
基本上 Linux 每天会针对主机上所有文件的位置进行搜索数据库的更新，执行 updatedb 命令会去读取/etc/updatedb.conf 的设置，然后再去硬盘里面进行查找文件名的操作并更新/var/lib/mlocate 内的数据库文件，可以在/etc/cron.daily/slocate.cron 这个文件找到相关的机制，也可以直接使用/usr/bin/updatedb 来更新数据库文件。

25.14.5 find

```
[root@linux ~]#find [PATH] [option] [action]
1、与时间有关的参数:
   -atime n : n为数字，意义为在n天之前的“一天之内”被access过的文件;
   -ctime n : n为数字，意义为在n天之前的“一天之内”被change过状态的文件;
   -mtime n : n为数字，意义为在n天之前的“一天之内”被modification过的文件;
   -newer file
    : file为一个存在的文件，意思是说只要文件比file还要新就会被列出来。
范例一：将过去系统上面24小时内有更动过内容（mtime）的文件列出
[root@linux ~]#find / -mtime 0
#那个 0 是重点，0 代表目前的时间，所以从现在开始到 24 小时前，
#有变动过内容的文件都会被列出来。那如果是三天前的 24 小时内?
#find / -mtime 3，意思是说今天之前的 3*24   4*24 小时之间
#有变动过的文件都被列出的意思，同时-atime 与-ctime 的用法相同。
范例二：查找/etc下的文件，如果文件日期比/etc/passwd新就列出
[root@linux ~]#find /etc -newer /etc/passwd
#-newer 用在分辨两个文件之间的新旧关系是很有用。
```

如果想找出一天内被改动过的文件名，可以使用上述的做法，即执行 `find / -mtime 0`。

但如果想要找出“4 天内被改动过的文件名”，就要使用 `fine / -mtime -4`，如果想要找出“4 天前的那一天”，就需要用 `find / -mtime 4`，这里有没有加上“+、−”差别很大，可以用如下的图示来说明：



图中最右边为当前的时间，越往左边则代表越早之前的时间，由图中可以看出：

- +4 代表大于等于 5 天前的文件名：ex>find / -mtime +4；
- −4 代表小于等于 4 天内的文件名：ex>find / -mtime -4；
- 4 代表 4~5 这一天的文件名：ex>find / -mtime 4。

2、与使用者或群组名称有关的参数：

```
-uid n : n为数字，这个数字是使用者的帐号ID，也就是UID，UID是记录在
         /etc/passwd里面与帐号名称对应的数字。
-gid n : n为数字，这个数字是群组名称的ID，也就是GID，GID记录在/etc/group。
-user name : name为使用者帐号名称，例如dmtsai；
-group name: name为群组名称，例如users；
-nouser    : 查找文件的所有者不存在于/etc/passwd的人。
-nogroup   : 查找文件的拥有群组不存在于/etc/group的文件，当用户自行安装
             软件时，很可能该软件的属性当中并没有文件所有者，在这个时候
             就可以使用-nouser与-nogroup搜索。
```

范例三：搜索/home下属于dmtsai的文件

```
[root@linux ~]#find /home -user dmtsai
#当要找出任何一个使用者在系统当中的所有文件时，
#就可以利用这个指令将属于某个使用者的所有文件都找出来。
```

范例四：搜索系统中不属于任何人的文件

```
[root@linux ~]#find / -nouser
#通过这个指令，可以轻易的就找出那些不太正常的文件。如果有找到不属于系统任何人的
    文件时，不要太紧张，
#那有时候是正常的，尤其是曾经以源代码自行编译软件时。
```

如果系统管理员需要知道某个用户在系统下创建了什么，可以使用上述的参数。对于`-nouser` 或`-nogroup` 的参数，除了用户自行从网络上下载文件时会发生外，如果系统中某个账号删除了，但是该账号在系统内已经创建过多个文件时就有可能发生找不到文件的

用户的情况，此时就可以使用**-nouser** 来找出这些文件。

3、与文件权限及名称有关的参数：
−name filename：搜索文件名称为filename的文件；
−size [+−]SIZE：搜索比SIZE还要大（+）或小（−）的文件。这个SIZE的规格有：

c：代表byte，k：代表1024bytes。所以要找比50KB还要大的文件，
就是“−size +50k”
−type TYPE：搜索文件的类型为TYPE的，类型主要有：
一般正规文件（f）；
装置文件（b，c）；
目录（d）；
连接文件（l）；
socket（s）及FIFO（p）等属性。
−perm mode：搜索文件属性“刚好等于”mode的文件，这个mode为类似chmod
的属性值，举例来说，−rwsr−xr−x的属性为4755。
−perm −mode ：搜索文件属性“必须要全部囊括mode的属性”的文件，举例来说，
要搜索−rwxr−−r−−，也就是0744的文件，使用−perm −0744，
当一个文件的属性为−rwsr−xr−x，也就是4755时，也会被列出来，
因为−rwsr−xr−x的属性已经囊括了−rwxr−−r−−的属性了。
−perm +mode ：搜索文件属性“包含任一mode的属性”的文件，举例来说，我们搜索
−rwxr−xr−x ，也就是 −perm +755 时，但一个文件属性为
−rw−−−−−−−
也会被列出来，因为它有 −rw.... 的属性存在！
4、额外可进行的动作：
−exec
command：**command**为其它指令，**−exec**后面可再接额外的指令来处理搜索到的结果。
−print ：将结果打印到显示器上，这个动作是默认动作。

范例五：找出文件名为passwd这个文件

```
[root@linux ~]#find / -name passwd
#利用这个-name 可以搜索文件名
```

范例六：搜索文件属性为f（一般文件）的文件

```
[root@linux ~]#find /home -type f
#-type 属性也很有帮助，尤其是要找出那些怪异的文件，
#例如 socket 与 FIFO 文件，可以用 find /var -type p 或-type s。
```

范例七：搜索文件当中含有SGID/SUID/SBIT的属性

```
[root@linux ~]#find / -perm +7000
#所谓的 7000 就是 -s-s-t，那么只要含有 s 或 t 的就列出，
#所以当然要使用 +7000，使用-7000 表示要含有-s-s-t 的所有三个权限，
```

范例八：将上个范例找到的文件使用ls –l列出来

```
[root@linux ~]#find / -perm +7000 -exec ls -l
#注意到，-exec 后面的 ls -l 就是额外的指令，
#而代表的是“由 find 找到的内容”的意思，所以-exec ls -l
#就是将前面找到的那些文件以 ls -l 列出长的数据。
# 则是表示-exec 的指令到此为止的意思，也就是说整个指令其实只有在
#-exec（里面就是指令执行）
#也就是说，-exec 最后一定要以 结束才行。
```

范例九：找出系统中大于1MB的文件

```
[root@linux ~]#find / -size +1000k
```

上述范例中的-perm 参数的重点在于找出特殊权限的文件，因为 SUID 与 SGID 都可以设置在二进制程序上，假设想要将/bin、/sbin 这两个目录下主要具有 SUID 或 SGID 的文件就列出来，就可以执行下面的命令：

```
[root@linux ~]#find /bin /sbin -perm +6000
```

因为 SUID 是 4，SGID 是 2，因此可用 6000 来处理这个权限。

注意，find 后面可以接多个目录来进行查找，另外 find 本来就会查找子目录，这个也要特别注意。而 find 的特殊功能是可以进行额外的动作（action），比如：

```
[root@linux ~]#find / -perm +7000 -exec ls -l
```

该范例中特殊的地方有“”与“\;”以及“-exec”关键字，分别介绍如下：

- 代表的是“由 find 找到的内容”，find 的结果会放到位置中；
- -exec 一直到“\;”是关键字，代表 find 额外命令的开始（-exec）到结束（\;），在这中间的就是 find 命令内的额外命令。在本例中就是“ls -l ”。
- ；在 bash 环境中是有特殊意义的，因此利用反斜杠来转义。

因此如果要查找的文件是具有特殊属性的，例如 SUID、SGID、文件所有者、文件大小等，这些条件 locate 是无法达到，那么使用 find 是一个不错的主意，它可以根据不同的参数来给予文件的搜索功能，例如要查找一个文件名为 httpd.conf 的文件，知道它应该是在/etc 下，那么就可以使用 find /etc -name httpd.conf。

另外，find 还可以利用通配符来查找文件，如果记得有一个文件文件名包含了 httpd，但是不知道全名，就可以用通配符 *，如上以：find /etc -name '*httpd*' 就可将文件名含有 httpd 的文件都列出来，不过由于 find 在查找数据的时后相当的消耗硬盘，所以一般情

况下不使用 find，而是用 whereis 与 locate 来代替。

find 可以指定查找的目录（连同子目录），而且可以利用额外的参数，因此在查找寻特殊的文件属性以及特殊的文件权限（SUID/SGID 等）时 find 是相当有用的工具程序之一。

25.14.6 fuser

The Unix command fuser is used to show which processes are using a specified file, file system, or unix socket. The equivalent command on BSD operating systems is fstat(1).

For example, to check process IDs and users accessing a USB drive:

```
$ fuser -m -u /mnt/usb1
 /mnt/usb1:   1347c(root)  1348c(guido)  1349c(guido)
```

The command displays the process identifiers of processes using the specified files or file systems. In the default display mode, each file name is followed by a letter denoting the type of access:

- c current directory.
- e executable being run.
- f open file.
- F open file for writing.
- r root directory.
- m mmap'ed file or shared library

The command can also be used to check what processes are using a network port:

```
# fuser -v -n tcp 80
                     USER        PID ACCESS COMMAND
80/tcp:              root       3788 F.... httpd
                     apache     3789 F.... httpd
                     apache     3790 F.... httpd
                     apache     3792 F.... httpd
                     apache     3795 F.... httpd
                     apache     3796 F.... httpd
                     apache     3799 F.... httpd
                     apache     3800 F.... httpd
                     apache     3801 F.... httpd
                     apache     4301 F.... httpd
# fuser -v -n tcp 3306
                     USER        PID ACCESS COMMAND
3306/tcp:            mysql      1007 F.... mysqld
```

The command returns a non-zero code if none of the files are accessed or in case of a fatal error. If at least one access has succeeded, fuser returns zero. The output of "fuser" may be useful in diagnosing "resource busy" messages arising when attempting to unmount filesystems.

- -k kills all process accessing a file.
  For example fuser -k /path/to/your/filename kills all processes accessing this directory without confirmation. Use -i for confirmation
- -i interactive mode. Prompt before killing process
- -v verbose.
- -u append username
- -a display all files
- -m name specifies a file on a mounted file system or a block device that is mounted.
  All processes accessing files on that file system are listed. If a directory file is specified, it is automatically changed to name/. to use any file system that might be mounted on that directory.

Also note that -k sends a SIGKILL to all process. Use the -signal to send a different signal. For a list of signals supported by the fuser run '`fuser -l`'.

The list of all open files and the processes that have them open can be obtained through the lsof command.

Chapter 26

文件系统操作

磁盘的整体信息保存在 superblock 块中，每个单独的文件的容量保存在 inode 中。

26.1 df

df 命令用于列出文件系统的整体磁盘使用量。

```
$ df
```

或者，可以使用-h 参数来输出更可读的信息。

```
$ df -h
Filesystem                          Size  Used Avail Use% Mounted on
/dev/mapper/fedora_theqiong-root     50G   20G   27G  43% /
devtmpfs                            2.0G     0  2.0G   0% /dev
tmpfs                               2.0G   18M  2.0G   1% /dev/shm
tmpfs                               2.0G  960K  2.0G   1% /run
tmpfs                               2.0G     0  2.0G   0% /sys/fs/cgroup
tmpfs                               2.0G   28K  2.0G   1% /tmp
/dev/sda1                           477M  147M  301M  33% /boot
/dev/mapper/fedora_theqiong-home     95G   87G  3.2G  97% /home
```

26.2 du

du 命令用于评估文件系统的磁盘使用量，也可以用于评估目录的容量。

```
$ du
```

为了查询某个目录所占用的所有空间，可以使用 du 命令，而且如果-s 参数可以依据不同的单位来找出文件的所占用的空间。

```
$ du -sm /home
88940 /home
```

如果需要列出某个目录下的子目录的容量，可以使用如下的命令：

```
$ du -hs /home
87G   /home
```

26.3 cat

26.4 modify

26.5 Disk Partition

在系统中新增硬盘时，需要对硬盘进行分区来新建可用的分区，并且对分区进行格式化来创建系统可用的文件系统。

磁盘分区可做看作是逻辑卷管理（LVM）前身的一项简单技术。

磁盘分区是使用分区编辑器（partition editor）在磁盘上划分出若干个逻辑部分，一旦划分成数个分区（Partition），不同类的目录与文件可以存储进不同的分区。分区越多，也就有更多不同的地方来将文件的性质区分得更细，并按照更为细分的性质存储在不同的地方以管理文件。

空间管理、访问许可与目录搜索的方式都依属于安装在分区上的文件系统。其中，当改变大小的能力依属于安装在分区上的文件系统时，需要谨慎地考虑分区的大小。

硬盘分区允许在一个硬盘上有多个文件系统，这样当其中一个分区出现逻辑损坏时，仅损坏的分区而不是整个硬盘受影响。

在运行 Unix 的多用户系统上，为了防止用户的硬连接攻击，可以将/home 和/tmp 路径与/var 和/etc 等系统文件分开。

基于 UNIX 或者 Linux 操作系统中创建了/、/boot、/home、/tmp、/usr、/var、/opt 和交换分区，从而保证了如果其中一个文件系统损坏，其它的数据（其它的文件系统）不受影响，从而就减少了数据丢失。

将整个硬盘驱动器划分成固定大小的小分区的缺点是可能会填满 /home 分区并且用完可用硬盘空间，即使其它分区上还有充足的空闲空间。

典型的操作系统使用另外一种约定：“/”（根目录）分区包含整个文件系统和独立的交换分区。

/home 分区允许在不破坏数据的前提下干净地重新安装（或者其他 Linux 发行版的更新安装）。

26.5.1 fdisk

fdisk（磁盘分区表操作工具）是用来磁盘分区的程序，它采用传统的问答式界面来将硬盘划分成若干个分区，同时也能为每个分区指定分区的文件系统。

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

Welcome to fdisk (util-linux 2.24.2).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.


Command (m for help): m

Help:

  DOS (MBR)
   a   toggle a bootable flag
   b   edit nested BSD disklabel
   c   toggle the dos compatibility flag

  Generic
   d   delete a partition
   l   list known partition types
   n   add a new partition
   p   print the partition table
   t   change a partition type
   v   verify the partition table

  Misc
   m   print this menu
   u   change display/entry units
   x   extra functionality (experts only)

  Save & Exit
   w   write table to disk and exit
```

```
  q   quit without saving changes

Create a new label
  g   create a new empty GPT partition table
  G   create a new empty SGI (IRIX) partition table
  o   create a new empty DOS partition table
  s   create a new empty Sun partition table
```

fdisk 可以列出整体硬盘的额状态信息，以及每个分区的信息。

- Device（设备文件名）
- Boot 表示是否为开机引导模块
- Start,End 表示分区的开始和结束柱面的号码
- Blocks 表示以 1KB 为单位的容量
- ID,System 表示分区 Id 和文件系统格式

fdisk 无法处理大于 2TB 以上的硬盘分区，因此必须使用 parted 命令来处理大于 2TB 以上的硬盘。

26.5.2 parted

Andrew ClausenIt 与 Lennert Buytenhek 开发的 GNU Parted（由“PARTition”与“EDitor”结合）是一个自由分区工具，可以用于创建、删除、移动分区，调整分区大小，检查、复制分区等操作。

GNU Parted 包含了一个库（libparted）以及一个命令行接口的前端（parted）来调整分区以安装新操作系统、备份特定分区到其他硬盘等。

GParted 以及 KDE Partition Manager 都是使用 GNU Parted 库的图形前端，它们分别作为 GNOME 及 KDE 桌面环境的硬盘分区工具。

GParted 使用 libparted 来识别、调整分区表，并有各个文件系统工具来处理分区上的文件系统。这些文件系统工具并不是必须的，但要处理一种文件系统就必须先安装相应的工具。

nparted 是一个基于 newt 的 GNU Parted 前端，fatresize 提供了使用 GNU Parted 库并且可以对 FAT16/FAT32 进行非破坏性的调整分区大小的命令行接口。

Pyparted（也被称作 python-parted）是一个以 Python 写成的图形前端。

26.5.3 partprobe

在系统运行过程中，如果磁盘无法卸载，那么内核就无法重新读取分区表信息。

使用分区工具对分区表的修改后，需要重新启动系统以更新内核的分区表信息，或者使用 partproble 命令来强制内核在不重启的情况下读取新的分区表。

26.6 Filesystem Format

26.6.1 mkfs

mkfs 命令可以调用正确的文件系统格式化工具来执行格式化。例如，当 mkfs 通过-t 指定文件系统格式时，实际上是调用对应的命令（例如 mkfs.ext4）来进行格式化操作。

```
[root@theqiong ~]#mkfs -t ext4 /dev/sdb
```

mkfs 支持的文件系统格式化工具包括 mkfs.ext2、mkfs.ext4dev、mkfs.hfsplus、mkfs.ntfs、mkfs.xfs、mkfs.btrfs、mkfs.ext3、mkfs.fat、mkfs.minix、mkfs.reiserfs、mkfs.cramfs、mkfs.ext4 mkfs.gfs2、mkfs.msdos 和 mkfs.vfat 等。

如果执行 mkfs 命令没有指定文件系统的具体选项，那么就会使用默认值来执行格式化，例如文件系统的卷标（label）、block 的大小以及 inode 的数量。

26.6.2 mke2fs

mke2fs 命令用于创建 ext2/ext3/ext4 文件系统，其效果与 `mkfs -t extX` 相同。

26.6.3 fsck

fsck（file system consistency check）可以指定文件系统来检查和修复硬盘错误，不过 Linux 可以自动通过 super block 来识别文件系统。

fsck 执行时可能会造成部分文件系统的损坏，因此应该先卸载需要检查的分区再执行 fsck 命令。

ext2/ext3 文件系统中的 lost+found 目录用于包含在执行 fsck 检查文件系统时发生问题的数据，理论上该目录中没有任何数据。

实际执行的 fsck 命令其实调用的是 e2fsck 命令。

26.6.4 badblocks

fsck 用来检验文件系统是否出错，badblocks 则是用来检查硬盘扇区有无坏块的。

badblocks 其实可以通过 `mke2fs -c` 设备文件名在进行格式化时处理硬盘表面的读取测试。

26.7 Partition Mount

用户在 UNIX 的机器上打开一个文件以前，包含该文件的文件系统必须先进行挂载的动作，此时用户要对该文件系统下 mount 的指令以进行挂载。通常是使用在 USB 或其他可

移除存储设备上，而根目录则保持挂载的状态。

Unix 文件系统可以对应一个文件而不一定要是硬件设备。

- 单一文件系统不应该被重复挂载在不同的挂载点（目录）中。
- 单一目录不应该重复挂载多个文件系统。
- 作为挂载点的目录理论上应该是空目录。

26.7.1 mount

mount 指令是告诉操作系统对应的文件系统已经准备好，可以使用了，而该文件系统会对应到一个特定的点（称为挂载点)。

挂载好的文件、目录、设备以及特殊文件即可提供用户使用。

除了操作系统调用的 mount 指令外，mount_root() 会优先挂载（或称根目录）。在这个情况下，操作系统会在调用 setup 前，先调用 mount。

每个在指定机器上被挂载的文件系统都会在 super_blocks[] 表格中以 super_block 的形式表现出来（最大数量由 NR_SUPER 决定)。

在虚拟文件系统中，superblock 是由 read_super( ) 进行初始化的动作。

下面的示例代码说明了挂载硬盘的第二个分区的指令。

```
$ mount /dev/hda2 /new/subdir
```

mount 命令可以将某个目录挂载另外一个目录上，尤其是在无法使用符号链接的程序运行时可以使用 mount 命令来额外挂载某个目录。

```
# mount --bind /home/test /mnt
```

使用 mount 的`--bind` 参数可以把不同的目录指向同一个 inode，从而就不必挂载整个文件系统。

下面的示例说明了如何列出所有已挂载的文件系统的指令。

```
$ mount
```

Linux 可以通过分析文件系统的 super block 并搭配自己的驱动程序去测试挂载，如果挂载成功就立刻自动使用合适的文件系统类型进行挂载。

- /etc/filesystems 包含系统指定的测试挂载文件系统类型；
- /proc/filesystems 包含 Linux 系统已经挂载的文件系统类型。

Linux 支持的文件系统的驱动程序放置在目录/lib/modules/$(uname -r)/kernel/fs/中，例如 vfat 文件系统的驱动就放置在/bin/modules/$(uname -r)/fs/kernel/fs/fat 目录中。

```
-rw-r--r--. 1 root root 97167 Nov 22 08:00 fat.ko
-rw-r--r--. 1 root root 16999 Nov 22 08:00 msdos.ko
-rw-r--r--. 1 root root 20791 Nov 22 08:00 vfat.ko
```

除了使用磁盘的设备文件名来挂载，还可以使用文件系统的卷标（label）来挂载。

26.7.2 umount

umount 告诉操作系统断开与该文件系统的连接，使其脱离挂载点。

mount 与 umount 指令必须以超级用户的权限运行，而且执行 umount 时必须退出文件系统的挂载点。

文件系统也可在/etc/fstab 文件中指定特定用户才能挂载，同样也只能由超级用户进行修改。

下面的代码示例说明了如何卸载同一个分区的指令。

```
$ umount /dev/hda2
```

或

```
$ umount /new/subdir
```

26.7.3 remount

如果需要以特定选项重新挂载分区，可以指定 remount 选项。

```
$ mount —o remount,rw /dev/hda2
```

目录树中最重要的是根目录，而且根目录是不能卸载的。不过，如果需要修改根目录的挂载参数，或者在根目录出现“只读”状态时，可以使用 remount 选项来重新挂载，尤其是进入单用户维护模式时很可能会出现根目录被系统挂载为只读的情况。

```
# mount -o remount,rw, auto /
```

26.7.4 pmount

pmount 是从标准的 mount 指令延伸出来的版本，其可以使普通的用户挂载可移除设备而忽略/etc/fstab 中的设置。

pmount 软件包也包含了另一个派生软件 pmount-hal，其可从 HAL（软件）读取设备信息以及使用 pmount 挂载。

gnome-mount 软件包包含了挂载、卸载以及退出存储设备的程序，其目标是代替原本的 mount 指令供其他的 GNOME 程序使用，不过 gnome-mount 并不会直接让用户运行。

所有的 gnome-mount 程序都使用了 HAL 的模式运行，所以不需要提高权限即可使用，因此 gnome-mount 可放置于 GConf 以方便集中管理。

26.8 Disk Parameter

用户可以修改磁盘参数（例如 label name 或 journal）以及磁盘运行时的相关参数（例如 DMA）。

26.8.1 mknod

UNIX 及 UNIX-like 操作系统中所有的设备都以文件来表示，并通过文件的 major 和 minor 数值来表示设备的界限。

major 和 minor 数值具有特殊的意义，其中 major 表示主设备代码，minor 表示次设备代码，Linux 内核就是根据 major 和 minor 来识别设备的。

下面列出了常见的硬盘文件名和设备代码。

硬盘文件名	Major	Minor
/dev/hda	3	0 ～ 63
/dev/hdb	3	64 ～ 127
/dev/sda	8	0 ～ 15
/dev/sdb	8	16 ～ 31

Linux 从 2.6 内核开始可以自动实时产生硬件文件名，用户可以不必手动创建设备文件。

在某些特殊情况下，如果需要使用 mknod 来手动处理设备文件，例如在某些服务被放到特定目录下（chroot 时。

```
#mknod 设备文件名 [bcp] [Major] [Minor]
```

26.8.2 e2label

在使用 mkfs 时可以设置文件系统卷标（lable），并且在格式化后也可以使用 e2label 来修改卷标。

操作系统可以通过卷标来识别磁盘，并且可以使用卷标来挂载磁盘。

```
#e2label device [ new-label ]
```

26.8.3 tune2fs

tune2fs 可以将 ext2 的文件系统转换为 ext3 文件系统，以及读取 superblock 中的数据和修改文件系统的卷标。

26.8.4 hdparm

hdparm 可以用来显示/设置硬盘的高级参数，或者测试硬盘性能。例如，hdparm 可以用来设置磁盘缓存、休眠模式、电源管理、噪音管理和 DMA 设置等。

通过调整硬盘的高级参数可以提高效率，例如开启 DMA 模式可以双倍或三倍地提高效率，而且 hdparm 可以用来测试 SATA 硬盘的缓存的访问性能和硬盘的实际访问性能。

```
#hdparm -Tt /dev/sda
```

在 GParted 和 Parted Magic 中都包含 hdparm。

```
// display information of hard drive
# hdparm -I /dev/sda
// turn on DMA mode
# hdparm -d1 /dev/sda
// test device read performance
# hdparm -t /dev/sda
// enable energy saving spindown after inactivity(24*5=120 seconds)
# hdparm -S 24 /dev/sda
// retain hdparm settings after a software reset
# hdparm -K 1 /dev/sda
// enable read-ahead
# hdparm -A 1 /dev/sda
// change acoustic management
# hdparm -M 128 /dev/sda
// switch drive to the lowest degree of power management (255: turn power management off)
# hdparm -B 254 /dev/sda
```

26.9 e2fsprogs

e2fsprogs（又称为 e2fs programs）是用以维护 ext2，ext3 和 ext4 文件系统的工具程序集。

ext2/3/4 是绝大多数 Linux 发行版默认的文件系统，因此 2fsprogs 工具集也包含在很多 Linux 发行版中。

具体来说，e2fsprogs 包含以下独立的程序：

- e2fsck, ext2/3/4 文件系统的 fsck 程序，用于检查文件系统的完整性。
- mke2fs 用于创建 ext2/3/4 文件系统。
- resize2fs 调整已创建的 ext2/3/4 文件系统的大小。
- tune2fs 修改 ext2/3/4 文件系统的相关参数。

- dumpe2fs 显示 ext2/3/4 文件系统的相关信息。
- debugfs 用于调试 ext2/3/4 文件系统，可以查看与更改文件系统的状态。

26.9.1 e2fsck

e2fsck 针对整个文件系统的 meta data 区域与实际数据存放区域进行比对。

26.9.2 mke2fs

26.9.3 resize2fs

26.9.4 tune2fs

26.9.5 dumpe2fs

```
# dumpe2fs /dev/sda
Journal inode:            8
Default directory hash:   half_md4
Directory Hash Seed:      e2ba7e3d-02f9-4a8f-b7a0-3084e3fa0fb6
Journal backup:           inode blocks
Journal features:         journal_incompat_revoke
Journal size:             8M
Journal length:           8192
Journal sequence:         0x00000122
Journal start:            1
```

26.9.6 debugfs

26.10 Boot Mount

为了在开机时自动挂载文件系统，可以对/etc/fstab 进行设置。

```
#/etc/fstab
#Created by anaconda on Wed Oct 29 20:09:22 2014
#
#Accessible filesystems, by reference, are maintained under '/dev/disk'
#See man pages fstab(5), findfs(8), mount(8) and/or blkid(8) for more info
#
/dev/mapper/fedora_theqiong-root /                       ext4    defaults
          1 1
```

```
UUID=e2b3c43b-091f-4446-b24b-fff8e889b5ed /boot                   ext4
    defaults        1 2
/dev/mapper/fedora_theqiong-home /home                    ext4    defaults
         1 2
/dev/mapper/fedora_theqiong-swap swap                     swap    defaults
         0 0
```

根目录/是必须挂载的，而且必须先于其他挂载点被挂载。

一般的挂载点必须是已存在的目录，可以任意指定，但是一定要遵守必需的系统目录层次结构（FHS)。

所有的挂载点在同一时间内只能挂载一次，而且所有分区在同一时间内只能挂载一次。

卸载挂载点时必须先将工作目录转移到挂载点（及其子目录）外部。

实际上，/etc/fstab（file system table）保存的就是 mount 指令需要的参数。另外，/etc/fstab 还增加了对 dump 命令的支持，以及开机时是否进行文件系统检查 fsck 等操作。

在手动挂载时可以让系统自动测试挂载，不过在/etc/fstab 中必须手动设置文件系统（例如 ext2/3/4、swap 等)。

/etc/fstab 中的内容共有 6 个字段，分别是磁盘设备文件名或设备卷标、挂载点、分区的文件系统、文件系统参数、是否支持 dump 以及是否执行 fsck 检查。

Table 26.1: 文件系统参数

参数	备注
async/sync	设置磁盘是否以异步方式运行，默认为 async
auto/noauto	设置文件系统是否会被自动测试挂载，默认为 auto
rw/ro	设置分区以可读写或只读的状态被挂载，默认为 rw
exec/noexec	限制是否可以执行，用于存储文件时可以设置为 noexec，默认为 exec
user/nouser	设置是否允许用户使用 mount 来挂载。默认不允许一般身份的用户执行 mount
suid/nosuid	设置文件系统是否允许设置 SUID，如果不用于存放可执行文件，可以设置为 nosuid
usrquota	设置在启动文件系统时是否支持磁盘配额模式
grpquota	启动文件系统对用户组磁盘配额模式的支持
defaults	默认值为 rw,suid,dev,exec,auto,nouser,async 等

用户可以通过/etc/fstab 来指定需要进行 dump 备份的文件系统，默认值为 0。

- 0 代表不需要 dump 备份;
- 1 代表每天进行 dump 备份;
- 2 代表其他不定日期的 dump 备份。

在系统开机时，默认会以 fsck 对文件系统完整性进行校验。不过，某些文件系统是不需要校验的，例如 swap 或者特殊特殊文件系统（/proc 与/sys 等)。

- 0 代表不进行校验;
- 1 代表最早校验（一般只有根目录会设置为 1);
- 其他需要校验的文件系统可以为 2。

/etc/fstab 是开机时的配置文件，不过实际文件系统的挂载是记录到文件/etc/mtab 与/proc/mounts 中。

在改动文件系统的挂载时，也会同时更改上述两个文件。

如果无法开机而进入单用户维护模式时，无法修改/etc/fstab 和/etc/mtab 的，这种情况下可以使用下面的命令来挂载/根目录文件系统。

```
#mount -n -o remount,rw /
```

Chapter 27

Loop Device

使用 loop 设备来挂载镜像文件时，同时也可以修改镜像文件内部的内容，因此很多镜像文件要经过 MD5 校验码来校验。

UNIX 中的硬件设备驱动程序和特殊设备文件（例如/dev/zero 和/dev/random）都以普通文件的形式出现，使用 dd 命令可以在这些文件中进行读写，因此 dd 命令可以用来备份硬盘驱动器的启动扇区以及从随机数发生器中获取数据等。

默认情况下，dd 命令从标准输入（stdin）设备（例如键盘）读入，并写入到标准输出（stdout）设备，并且在读取到 EOF 标志时退出。

用户可以通过 if（输入文件）和 of（输出文件）来定制 dd 命令，也可以创建大型文件并且将其格式化后进行挂载，这样就可以用来解决分区不合理的情况。

一个块（block）是每次读取/写入/转换的字节数目的最小单元，因此可以通过 ibs 来指定 input/reading 的块大小，以及 obs 来指定 output/writing 的块大小。

一般使用 bs 来代替 ibs 和 obs，而且缺省的读/写块大小为 512bytes，并且使用 count 参数来指定读入的块的数目。

dd 命令每次操作的数据量就是 bs，因此 bs 参数的大小可以影响 dd 操作的效率，尤其是当用于网络数据传输时需要依据使用的网络协议的包大小来指定 bs（block size）。

下面的示例使用 dd 命令创建一个 512MB 的大文件，其中/dev/zero 是不断输出 0 的设备。

```
#dd if=/dev/zero of=/home/theqiong/loopdev bs=1M count=512
512+0 records in
512+0 records out
...
#ll -h /home/theqiong/loopdev
-rw-r--r-- root root 512M Dec 11 14:12 /home/theqiong/loopdev
```

27.1 mkfs

```
#mkfs -t ext4 /home/theqiong/loopdev
```

27.2 mount

```
#mount -o loop /home/theqiong/loopdev /media/cdrom/
```

在不改动原本的分区的情况下可以创建大容量的文件，可以使用使用 VMware、Xen 等虚拟机软件配合 Loop device 的文件类型来进行根目录的挂载。

Chapter 28

Swap

在 Linux 操作系统中，swap 可以用于处理物理内存不足的情况，而且一般不会被系统使用。

在 von Neumann 系统结构的计算机中，CPU 读取的数据都来自于内存，内存中的数据在空闲时可以放置到 swap 中，然后就可以把内存提供给需要执行的程序使用。

在服务器上，为了处理大量的网络请求，可以预留 swap 来作为内存缓冲。

新建 swap 分区时，首先使用 fdisk 新建物理分区，然后使用 mkswap 命令来格式化分区，最后使用 swapon 命令启动 swap 设备。

如果物理分区无法支持 swap，可以使用 loop device 来创建 swap 分区。

28.1 fdisk

在使用 fdisk 命令来创建新分区时，需要设置 system ID，并且需要使用 partprobe 来让内核更新分区表。

或者，使用 dd 命令新建一个大型文件来作为 swap 分区。

```
#dd if=/dev/zero of=/tmp/swap bs=1M count=512
```

28.2 mkswap

mkswap 命令可以将物理分区或 loop 设备格式化为 swap 文件格式。

```
#mkswap /tmp/swap
```

28.3 swapon

28.4 swapoff

swapoff 命令可以用来关闭 swap。

swap 对现代计算机系统结果来说已经意义不大，只是对于服务器或者工作站系统来说还是必要的。

swap 的主要功能是用于在物理内存不足时将内存中的闲置程序暂时转移到 swap 中，或者在系统进入休眠模式时将运行中的程序状态记录到 swap 中，以作为“唤醒”主机的状态根据。

某些程序在运行时也可能会利用 swap 的特性来存放某些数据段。

在 2.4.10 内核以后，单一 swap 已经没有了 2GB 的限制，但是最多还是仅能创建 32 个 swap。

x86_64 系统的最大内存寻址到 64GB，因此 swap 总量最大也仅能达到 64GB。

Chapter 29

Boot Sector

一般情况下，Linux 系统的引导装载程序可以安装到 super block 中，不过可安装开机信息的 boot sector（启动扇区）可以独立出来，不必放置在 super block 中。

- super block 的大小为 1024types；
- super block 前面需要保留 1024bytes 来安装引导装载程序。

当文件系统的 block 为 1024bytes 时，boot sector 和 super block 各自会占用一个 block，因此 boot sector 是独立于 super block 的。

```
#dumpe2fs /dev/sda1
Group 0: (Blocks 1-8192) [ITABLE_ZEROED]
  Checksum 0xc27d, unused inodes 1992
  Primary superblock at 1, Group descriptors at 2-3
  Reserved GDT blocks at 4-259
  Block bitmap at 260 (+259), Inode bitmap at 276 (+275)
  Inode table at 292-545 (+291)
  3810 free blocks, 1993 free inodes, 5 directories, 1992 unused inodes
  Free blocks: 4379-4380, 4385-8192
  Free inodes: 39, 41-2032
Group 1: (Blocks 8193-16384) [INODE_UNINIT, ITABLE_ZEROED]
  Checksum 0x7331, unused inodes 2032
  Backup superblock at 8193, Group descriptors at 8194-8195
  Reserved GDT blocks at 8196-8451
  Block bitmap at 261 (bg \#0 + 260), Inode bitmap at 277 (bg #0 + 276)
  Inode table at 546-799 (bg \#0 + 545)
  2449 free blocks, 2032 free inodes, 0 directories, 2032 unused inodes
```

```
Free blocks: 8831—9332, 9905—10046, 10048—10077, 10222—10240,
  12289—12298, 12442—13087, 13231—13318, 13324—13824, 13826—14336
Free inodes: 2033—4064
```

0 号 block 是保留的，预留给 boot sector，整个分区的文件系统分区情况如下：

- Block 0：Boot sector（1024bytes）
- Block 1：Super block（1024bytes）
- 其他文件系统数据

如果文件系统的 block 大于 1024bytes，那么 super block 还是在 0 号 block 中，而且引导装载程序还是被安装到 super block 所在的 block 中（即 Block 0）。

不管文件系统的 block 大小如何设置，引导装载程序都是安装到文件系统最前面的 1024bytes 内的区域中，即启动扇区。

Chapter 30

Sector Ratio

在文件系统中，一个 block 只能放置一个文件，其实整个文件系统中包括 super block、inode table 和其他数据中都会存在磁盘空间浪费的问题。

如果使用 `ll -s` 查看某个目录，第一行中显示的 total 指的是该目录下的所有数据占用的实际 block 数量 ×block 大小的值。

```
#ll -s /
total 62
 0 lrwxrwxrwx.   1 root root     7 Dec 12  2013 bin -> usr/bin
 2 dr-xr-xr-x.   6 root root  1024 Dec  6 22:31 boot
 0 drwxr-xr-x.  20 root root  3400 Dec 13 13:30 dev
12 drwxr-xr-x. 148 root root 12288 Dec 13 10:43 etc
 4 drwxr-xr-x.   4 root root  4096 Oct 29 20:10 home
 0 lrwxrwxrwx.   1 root root     7 Dec 12  2013 lib -> usr/lib
 0 lrwxrwxrwx.   1 root root     9 Dec 12  2013 lib64 -> usr/lib64
16 drwx------.   2 root root 16384 Dec 12  2013 lost+found
 4 drwxr-xr-x.   2 root root  4096 Aug  7  2013 media
 4 drwxr-xr-x.   2 root root  4096 Aug  7  2013 mnt
 4 drwxr-xr-x.   6 root root  4096 Nov 24 13:44 opt
 0 dr-xr-xr-x. 226 root root     0 Dec 13  2014 proc
 4 dr-xr-x---.  14 root root  4096 Dec 13 15:00 root
 0 drwxr-xr-x.  38 root root  1020 Dec 13 13:55 run
 0 lrwxrwxrwx.   1 root root     8 Dec 12  2013 sbin -> usr/sbin
 4 drwxr-xr-x.   2 root root  4096 Aug  7  2013 srv
 0 dr-xr-xr-x.  13 root root     0 Dec 13 10:43 sys
 0 drwxrwxrwt.  14 root root   360 Dec 13 15:31 tmp
 4 drwxr-xr-x.  13 root root  4096 Sep 27 09:58 usr
```

```
 4 drwxr-xr-x.  22 root root  4096 Dec 13  2014 var
```

在每个文件或目录前面的数字说明所用掉的 block 的数目。如果是空目录，则 `ll -s` 的结果如下：

```
#ll -s Desktop/
total 0
```

Chapter 31

File Archive

31.1 Overview

目前的计算机系统中都是使用字节（bytes）为单位来计量的，不过事实上，计算机最小的计量单位应该是 bits（1 byte = 8 bits）。

一个 byte 中会有 8 个 bit，每个 bit 可以是 0 或 1，二进制 1 会在最右边占据 1 个 bit，而其他的 7 个 bits 将会自动的被填上 0。

为了要满足现代操作系统对数据的存取，可以将该数据使用字节（byte）来记录。如果利用某些计算方式将没有使用到的空间释放出来，就可以让文件占用的空间变小，这就是压缩的技术。

使用压缩命令可以大大缩小文件的大小，从而方便用户从网络上下载大型的文件，不过针对不同的压缩指令所产生的压缩文件，还是会有一些特殊的命名方式。

通过文件压缩技术可以将磁盘使用量降低并达到减小文件容量的目的。另外，某些压缩程序可以进行容量限制，将一个大型文件可以分割成为数个小文件。

31.1.1 Ratio

压缩前与压缩后的文件所占用的硬盘空间大小，就可以被称为是“压缩比”（ratio）。

为了理解压缩计数，可以认为其实文件里面有相当多的“空间”存在，并不是完全填满的，而“压缩”的技术就是将这些“空间”填满以让整个文件占用的容量下降。不过，这些“压缩过的文件”无法直接被操作系统使用，因此若要使用这些被压缩过的文件数据，则必须将其“还原”回来未压缩前的模样，那就是所谓的“解压缩”。

在网络数据的传输中，通过压缩计数可以降低数据量，从而让带宽用来作更多的工作，例如 HTTP 协议可以利用文件压缩的技术来进行数据的传输，并且加速网站的访问速度。

- gzip

- compress
- deflate

在浏览网页的过程中，主要在“数据的传输”方面消耗时间（而不是 CPU 的运算），因此使用压缩过的数据可以提高传送的速度。

现在，普遍使用的 gzip 压缩技术可以让网站的数据在网络传输时，使用的是“压缩过的数据”，这些压缩过的数据到达客户端主机时，再进行解压缩。

31.1.2 Extension

使用不同的压缩技术产生的文件，通常其扩展名都是 *.tar, *.tar.gz, *.tgz, *.gz, *.Z, *.bz2 等，而且 Linux 提供的压缩命令的计算方法都不是完全相同的。

不过，适当的文件名称扩展名还是必要的，下面仅列出常见的压缩文件扩展名。

*.Z	compress 程序压缩的文件；
*.bz2	bzip2 程序压缩的文件；
*.gz	gzip 程序压缩的文件；
*.tar	tar 程序打包的数据，并没有压缩过；
*.tar.gz	tar 程序打包的文件，其中并且经过 gzip 的压缩

最早期的压缩程序是 compress，后来 GNU 计划开发出新一代的压缩指令 gzip（GNU zip）用来取代 compress，以及 bzip2 可以用来提供更好的压缩比。

不过，这些指令通常仅能针对一个文件来压缩与解压缩，tar 程序可以将很多文件（或目录）“打包”成为一个文件，于是将整个 tar 与压缩的功能结合在一起，这样用户就可以更方便地进行压缩与打包。

File Roller 可以处理归档文件的创建修改、查看，以及解压缩等，但是它仅是一个前端，需要同时安装相应的后端程序才能正常运行。

31.2 compress

compress 是用来压缩与解压缩扩展名为 *.Z 的文件的命令。

compress 是最简单的压缩指令，基于LZW压缩算法。

```
[root@linux ~]#compress [-dcr] 文件或目录
```

参数

-d：用来解压缩的参数；

-r：可以连同目录下的文件也同时给予压缩；

-c：将压缩数据输出成为standard output（输出到显示器）。

范例

范例一 将/etc/man.config复制到/tmp，并加以压缩

```
[root@linux ~]#cd /tmp
[root@linux tmp]#cp /etc/man.config .
[root@linux tmp]#compress man.config
[root@linux tmp]#ls -l
-rw-r--r--  1 root root 2605 Jul 27 11:43 man.config.Z
```

范例二 将压缩文件解开

```
[root@linux tmp]#compress -d man.config.Z
```

范例三 将man.config压缩成另外一个文件来备份

```
[root@linux tmp]#compress -c man.config > man.config.back.Z
[root@linux tmp]#ll man.config*
-rw-r--r--  1 root root 4506 Jul 27 11:43 man.config
-rw-r--r--  1 root root 2605 Jul 27 11:46 man.config.back.Z
#使用-c 参数会将压缩过程的数据输出到显示器上，而不是写入成为 file.Z 文件。
所以可以通过数据流重导向的方法将数据输出成为另一个文件名。
```

不过，使用的时候需要特别留意的是，当以 compress 压缩之后，如果没有其他的参数，那么原本的文件就会被后来的 *.Z 所取代。例如，原本压缩的文件为 man.config，那么当压缩完成之后将只会剩下 man.config.Z 文件，解压缩则是将 man.config.Z 解压缩成 man.config。

31.3 uncompress

除了可以使用 `compress -d` 参数来解压缩之外，也可以直接使用意义相同的 uncompress。

如果不想让原本的文件被更名成为 *.Z 而是产生另外的一个文件名，就可以利用数据流重定向（也就是 >）将原本应该在显示器上面出现的数据把它存储到其他文件去。当然，这要加上-c 的参数才行。

```
$ compress -c man.config > man
```

compress 无法解压缩 *.gz 文件，但是 gzip 可以解压缩 *.Z 文件，现在 compress 已经集成到 ucompress 软件中。

31.4 gzip

gzip（GUN zip）是用来压缩与解压缩扩展名为 *.gz 的命令。相比 compress，gzip 可以提供更高的压缩比。

gzip 的基础是 DEFLATE[1]，DEFLATE 是 LZ77 与哈夫曼编码的一个组合体。

zlib 是 DEFLATE 算法的实现库，它的 API 同时支持 gzip 文件格式以及一个简化的数据流格式。zlib 数据流格式、DEFLATE 以及 gzip 文件格式均已被标准化，分别是 RFC 1950、RFC 1951 以及 RFC 1952。

```
[root@linux ~]#gzip [-cdt#] 文件名
[root@linux ~]#zcat 文件名.gz
```

参数

```
-c   将压缩的数据输出到显示器上，可通过数据流重导向来处理;
-d   解压缩的参数;
-t   可以用来检验一个压缩文件的一致性，看看文件有无错误;
-#  压缩等级，-1 最快，但是压缩比最差，-9 最慢，但是压缩比最好，预设是 -6。
```

范例

范例一 将/etc/man.config复制到 /tmp，并且以gzip压缩

```
[root@linux ~]#cd /tmp
[root@linux tmp]#cp /etc/man.config .
[root@linux tmp]#gzip man.config
#此时 man.config 会变成 man.config.gz。
```

范例二 将范例一的文件内容读出来。

```
[root@linux tmp]#zcat man.config.gz
#此时显示器上会显示 man.config.gz 解压缩之后的文件内容。
```

范例三 将范例一的文件解压缩

```
[root@linux tmp]#gzip -d man.config.gz
```

范例四 将范例三解开的man.config用最佳的压缩比压缩，并保留原本的文件

```
[root@linux tmp]#gzip -9 -c man.config > man.config.gz
```

gzip 可以解压缩 compress、zip 和 gzip 等软件产生的压缩文件。具体来说，gzip 是若干种文件压缩程序的简称，通常指 GNU 计划的实现。其中，OpenBSD 中所包含的 gzip 版本实际上是 compress，其对 gzip 文件的支持在 OpenBSD 3.4 中被添加。

gzip 文件格式如下：

- 10 字节的头，包含幻数、版本号以及时间戳
- 可选的扩展头，如原文件名

[1]最初，DEFLATE 是作为 LZW 以及其它受专利保护的数据压缩算法的替代版本而开发的，当时那些专利限制了 compress 以及其它压缩工具的应用。

- 文件体，包括 DEFLATE 压缩的数据
- 8 字节的尾注，包括 CRC-32 校验和以及未压缩的原始数据长度

gzip 文件格式允许多个这样的数据拼接在一起，在解压时也能识别出它们是拼接在一起的数据，但通常 gzip 仅用来压缩单个文件。

多个文件的压缩归档通常是首先将这些文件合并成一个 tar 文件，然后再使用 gzip 进行压缩，最后生成的.tar.gz 或者.tgz 文件就是所谓的“tar 压缩包”或者“tarball”。

相比 gzip，ZIP 也使用 DEFLATE 算法，而且可移植性更好，不需要一个外部的归档工具就可以包容多个文件。但是，由于 ZIP 对每个文件进行单独压缩而没有利用文件间的冗余信息（即固实压缩），所以 ZIP 的压缩率会稍逊于 tar 压缩包。

7-Zip 在类 UNIX 系统上的接口也被称为 p7zip，其内部也有一个 DEFLATE 实现，可以产生 gzip 兼容的压缩文件，并可以产生的比 gzip 更高的压缩率，不过需要耗费更多的处理器时间成本。

- `-c`，`--stdout`：将解压缩的内容输出到标准输出，原文件保持不变
- `-d`，`--decompress`：解压缩
- `-f`，`--force`：强制覆盖旧文件
- `-l`，`--list`：列出压缩包内储存的原始文件的信息（如，解压后的名字、压缩率等）
- `-n`，`--no-name`：压缩时不保存原始文件的文件名和时间戳，解压缩时不恢复原始文件的文件名和时间戳（此时，解出来的文件，其文件名为压缩包的文件名）
- `-N`，`--name`：压缩时保存原始文件的文件名和时间戳，解压缩时恢复原始文件的文件名和时间戳
- `-q`，`--quiet`：抑制所有警告信息
- `-r`，`--recursive`：递归
- `-t`，`--test`：测试压缩文件完整性
- `-v`，`--verbose`：冗余模式（即显示每一步的执行内容）
- `-1`、`-2`、`...`、`-9`：压缩率依次增大，速度依次减慢，默认为-6

另外，gzip 也提供压缩比的服务，1 是最差的压缩比，但是压缩速度最快，9 虽然可以达到较佳的压缩比（经过压缩之后，文件比较小一些），但是却会损失一些速度，默认的压缩比是 6。

HTTP/1.1 协议允许客户端选择要求从服务器下载压缩内容，这个标准本身定义了三种压缩方法：gzip、compress 和 deflate。

- “gzip”（内容用 gzip 数据流进行封装）；
- “compress”（内容用 compress 数据流进行封装）；
- “deflate”（内容是原始格式、没有数据头的 DEFLATE 数据流）。

大部分 HTTP 客户端库、服务器平台和绝大多数现代浏览器都支持 gzip 和 deflate 两种格式。

31.4.1 zcat

zcat 是用来读取压缩文件数据内容的指令，使用 cat 读取文字文件，使用 zcat 读取压缩文件。

gzip 命令主要是用来取代 compress 的，所以 compress 的压缩文件也可以使用 gzip 来解压缩。同时，zcat 命令可以同时读取 compress 与 gzip 的压缩文件。

31.4.2 gunzip

gzip 压缩文件对应的解压程序是 gunzip。

31.5 bzip2

自 1990 年代末期以来，基于数据块排序算法的文件压缩工具 bzip2 作为 gzip 的替代逐渐得到流行，它可以生成相当小的压缩文件，尤其是对于源代码和其他结构化文本。

使用 bzip2 的代价是更大的内存与处理器时间消耗，bzip2 压缩的 tar 包一般为.tar.bz2 或.tbz。

在有些情况下，按照绝对压缩效率来讲，bzip2 不如 7z 和 RAR 格式，不过 bzip2 比传统的 gzip 或者 ZIP 的压缩效率更高，但是它的压缩速度较慢。

bzip2 使用 Burrows-Wheeler transform 将重复出现的字符序列转换成同样字母的字符串，然后用 move-to-front transform 进行处理，最后使用哈夫曼编码进行压缩。

在 bzip2 中所有的数据块都是大小一样的纯文本数据块，它们可以用命令行变量进行选择，然后用从 π 的十进制表示得到的一个任意位序列标识成压缩文本。

在 UNIX-like 系统下，bzip2 可以独立使用也可以与 tar 一起使用。bzip2 file 压缩文件，bzip2 -d file.bz2 解压文件，解压也可以使用另外一个名字 bunzip2。

```
[root@linux ~]#bzip2 [-cdz] 文件名
[root@linux ~]#bzcat 文件名.bz2
参数
-c   将压缩的过程产生的数据输出到显示器上
-d   解压缩的参数
-z   压缩的参数
-# 与 gzip 同样的，都是在计算压缩比的参数，-9 最佳，-1 最快。

范例
范例一 将/tmp/man.config以bzip2压缩
[root@linux tmp]#bzip2 -z man.config
#此时 man.config 会变成 man.config.bz2。
```

范例二 将范例一的文件内容读出来。

```
[root@linux tmp]#bzcat man.config.bz2
#此时显示器上会显示 man.config.bz2 解压缩之后的文件内容。
```

范例三 将范例一的文件解压缩

```
[root@linux tmp]#bzip2 -d man.config.bz2
```

范例四 将范例三解开的man.config用最佳的压缩比压缩并保留原本的文件

```
[root@linux tmp]#bzip2 -9 -c man.config > man.config.bz2
```

bzip2 的命令行标志大部分与 gzip 相同，所以，从 tar 文件解压 bzip2 压缩的文件可以用：

```
bzcat ''archivefile''.tar.bz2 | tar -xvf -
```

生成 bzip2 压缩的 tar 文件可以使用：

```
tar -cvf - ''filenames'' | bzip2 > ''archivefile''.tar.bz2
```

GNU tar 支持 -j 标志，这就可以不经过管道直接生成 tar.bz2File:

```
tar -cvjf ''archivefile''.tar.bz2 ''file-list''
```

解压 GNU tar 文件可以使用：

```
tar -xvjf ''archivefile''.tar.bz2
```

使用 compress 扩展名自动建立为.Z，使用 gzip 扩展名自动建立为.gz，这里的 bzip2 则是自动的将扩展名约定为.bz2。例如，当使用具有压缩功能的 bzip2 -z 时，那么 man.config 就会自动的变成 man.config.bz2 这个文件名。

与 RAR 或者 ZIP 等其它不同的是，bzip2 只是一个数据压缩工具，而不是归档工具，在这一点上它与 gzip 类似。bzip2 本身不包含用于多个文件、加密或者文档切分的工具，相反按照 UNIX 的传统需要使用如 tar 或者 GnuPG 等外部工具。

31.5.1 bzcat

bzcat 命令可以用来读取压缩文件内容，例如可以使用 bzcat man.config.bz2 来读取数据而不需要解压缩。

```
$ bzcat man.config.bz2
```

当要解压缩某个压缩文件时，文件名后缀可以为.bz、.bz2、.tbz 和.tbz2 等，那么就可以尝试使用 bzip2 来解压缩。

31.5.2 bunzip2

bunzip2 命令可以用来取代 `bzip2 -d` 执行解压缩。

31.6 ZIP

ZIP（原名 Deflate）文件格式的发明者 Phil Katz 于 1989 年 1 月公布了该格式的资料。

从性能上比较，RAR 及 7z 格式较 ZIP 格式压缩率较高，

31.6.1 7-Zip

由 Igor Pavlov 于 1999 年开始开发的 7-Zip 提供命令行接口的程序或图形用户界面的程序，而且可以与资源管理器结合。

- 7-Zip 的主体在 GNU LGPL 下发布；
- 加密部份使用了高级加密标准（AES）的代码，使用 BSD 许可证发布；
- 解压 RAR 部分使用 RAR 特定的许可协议发布。

LZMA 算法比起其他常见的传统压缩算法（例如 Zip、RAR）来说相对较新，压缩率也比较高。具体来说，LZMA（Lempel-Ziv-Markov chain-Algorithm）使用类似于 LZ77 的字典编码机制，并且数据流、重复序列大小以及重续序列位置单独进行了压缩。

LZMA 支持散列链变体、二叉树以及基数树作为它的字典查找算法基础，用于压缩的字典文件大小可达 4GB。

Igor Pavlov 使用 C++ 开发的 LZMA 库使用了区间编码支持的 LZ77 改进压缩算法以及特殊的用于二进制的预处理程序。

7-Zip 预设的格式是其自行开发的 7z 格式，扩展名为.7z，并且在 7z 格式中包含了多种算法，例如 bzip2 以及 Igor Pavlov 开发的 LZMA。

7z 格式支持 256 位键钥 AES 算法加密，键钥则由用户提供的暗码进行 SHA-256 散列算法得到，并且使用大量迭代以使得对暗码的暴力解码更加困难。

7z 格式的开发结构允许添加标准以外的压缩算法。

- 改良和优化算法后的 LZMA 最新版本，使用马尔可夫链/熵信息编码和 Patricia trie。
- 经过改良后的 LZMA2 算法。
- 基于 Dmitry Shkarin 的算法 2002 PPMdH (PPMII/cPPMII) 并加以优化的 PPMD 算法。
- 32 位 x86 可执行文件转换程序 BCJ，对短程 jump 操作和调用操作的目标地址进行压缩。
- 32 位 x86 可执行文件转换程序 BCJ2 对 jump 操作，并且调用操作和有条件 jump 操作的目标地址进行单独压缩。
- 标准 BWT 算法 bzip2 使用（更快的）哈夫曼编码和（更强的）熵信息编码。
- 标准 LZ77-based 算法 DEFLATE。

BCJ/BCJ2 压缩工具所附带的 LZMA SDK 包括在 X86、ARM、PowerPC、IA-64 以及 ARM Thumb 处理器上在压缩之前跳转目标进行归一化处理。对于 x86 平台来说，这是一个近跳转、近调用以及近条件跳转需要从“向后跳 1665 字节”这样的机器语言归一化到“跳转到 5554”这样的格式，但是短跳转及短条件跳转不需要进行这样的处理。

BCJ 与 BCJ2 之间的区别在于前者只将近跳转及近调用目标地址转换到归一化的形式，而 BCJ2 只将 x86 平台下的近跳转、近调用及条件近跳转目标分别进行压缩。

另外，7z 格式默认使用 Unicode 来存储文件名称，可以避免不同系统间压缩解压乱码的问题。

31.6.2 p7zip

7-Zip 的 CLI 版本 7zip 是移植到 POSIX/Linux 的 7-Zip 软件，可以压缩解压 7z 格式的文件，并且在 7z 和 7za 命令中可以使用参数调整压缩/解压设置。

31.6.3 xz

xz 是一个使用 LZMA 压缩算法的无损数据压缩文件格式，支持多文件压缩，并且在 GNU coreutils 中被使用，也受到 tar 的透明支持。

通常情况下，xz 作为一种归档文件自身的压缩格式，例如使用 tar 或 cpio 程序创建的归档。

31.7 JAR

在软件领域，JAR 文件（Java ARchive）是一种以 ZIP 格式构建的软件包文件格式，通常用于聚合大量的 Java 类文件、相关的元数据和资源（文本、图片等）文件到一个文件，以便分发 Java 平台应用软件或库。

用户可以使用 JDK 自带的 jar 命令或其他 zip 压缩工具来创建或提取 JAR 文件，不过压缩时 zip 文件头里的条目顺序很重要，因为 Manifest 资源配置文件常需放在首位。

JAR 文件内的文件名是 Unicode 文本，在包 java.util.zip 中提供了读写 JAR 文件的类。

- WAR （Web application archive）可以存储 XML 文件、Java 类、JSP 和 Web 应用程序中的其他文件。
- RAR（Resource adapter archive）可以存储 XML 文件、Java 类和 Java EE 连接器架构（JCA）应用中的其他文件。
- EAR（Enterprise archive）包含了 XML 文件、Java 类和针对 Java EE 应用的其他 Java 归档文件，例如 JAR、WAR 和 RAR。
- SAR（Service archive）与 EAR 类似，并提供了 service.xml 文件和相应的 JAR 文件。

- APK (Android application package）是 Java 归档格式的一个变种，用于 Android 应用程序。

例如，WAR 文件（Web application ARchive）是一种 JAR 文件，其中包含用来分发的 JSP、Java Servlet、Java 类、XML 文件、标签库、静态网页（HTML 和相关文件），以及构成 Web 应用程序的其他资源。

一般情况下，一个 WAR 文件可能会以与 JAR 文件相同的方式进行数字签名，以便他人确定哪些源代码来自哪一个 JAR 文件。

另外，WAR 文件也有其特殊的文件和目录。例如，如果 Web 应用程序使用了 servlet，则 Servlet 容器会使用 web.xml 文件来确定某个 URL 请求将被路由到哪个 Servlet 上。web.xml 还用于定义 Servlet 中可以引用的上下文变量，以及部署器所需配置的环境依赖关系。例如，一个依赖于邮件会话、用于发送电子邮件的程序，而 Servlet 容器负责提供这项服务，这就需要在 web.xml 进行一些配置。

下面的示例 web.xml 文件，演示了一个 Servlet 是怎样被声明和被关联的。

```
<?xml version="1.0" encoding="UTF-8"?>
 <!DOCTYPE web-app
     PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
     "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

 <web-app>
     <servlet>
         <servlet-name>HelloServlet</servlet-name>
         <servlet-class>mypackage.HelloServlet</servlet-class>
     </servlet>

     <servlet-mapping>
         <servlet-name>HelloServlet</servlet-name>
         <url-pattern>/HelloServlet</url-pattern>
     </servlet-mapping>

     <resource-ref>
         <description>

    资源引用到一个可被用于发送电子邮件的javax.mail.Session的实例工厂。
         </description>
         <res-ref-name>mail/Session</res-ref-name>
         <res-type>javax.mail.Session</res-type>
         <res-auth>Container</res-auth>
```

```
    </resource-ref>
</web-app>
```

Ant 构建工具可以使用其自己的包（org.apache.tools.zip）来读写 Zip 和 JAR 归档文件，并包括对 Unix 文件系统的支持。

31.7.1 JAR File

一个 JAR 文件允许 Java 运行时高效地部署一组类和它们相关的资源，而且 JAR 文件中的元素可以被压缩，这样就可以在单个请求中下载整个应用程序。

JAR 文件在路径 META-INF/MANIFEST.MF 下有一个可选的 Manifest 文件，Manifest 文件中的条目定义这个 JAR 文件如何被使用。例如，类路径条目由其他 JAR 文件的绝对或相对路径的列表组成，用于指定在加载本 JAR 文件时同时加载的其他 JAR 文件。

虽然 Manifest 文件的目的在于简化 JAR 的使用，但是在实践中证明 Manifest 文件是非常脆弱的。其中，入口点 JAR 在创建时依赖于所有相关的 JAR 的确切位置，如果需要更改版本或库的位置，必须重建 Manifest 文件。

开发者可以对 JAR 文件进行数字签名，签名信息成为嵌入的 Manifest 文件的一部分。JAR 本身并没有被签名，归档内的每一个文件的校验和连同其名字作为签名后被记录在 Manifest 文件中。

多个实体可能签署一个 JAR 文件，并在每次签名时改变这个 JAR 文件，虽然签署文件本身仍然有效。当 Java 运行时加载签名的 JAR 文件时，它可以验证签名并拒绝加载签名不匹配的类。

Java 运行时也支持“密封”的包（Sealed Packages），类加载器成功装载密数据包中的某个类后，其后加载的类必须是由同一实体签名，才被允许加载到同一个包，从而可以防止恶意代码被插入到现有的软件包，或者接触到包范围内的类和数据。

开发者还可以对 JAR 文件进行混淆，这样 JAR 文件的用户无法得到关于 JAR 文件所包含的代码的太多信息，并且能够压缩文件大小，因而在空间受限的嵌入式系统开发中很有用。

31.7.2 Executable JAR

可执行 Java 程序以及其使用的库文件也可以打包在一个 JAR 文件中。

可执行的 JAR 文件中的 Manifest 文件用代码 Main-Class: myPrograms.MyClass 指定了入口点类，注意要指明该类的路径（-cp 参数将被忽略）。

有些操作系统可以在点击后直接运行可执行 JAR 文件，更典型的调用则是通过命令行执行“`java -jar foo.jar`”，而且在多数平台上可以使用封装器来封装可执行 JAR 文件，并将单个 JAR 文件转换为可执行文件。

31.7.3 Manifest File

在 Java 平台中, Manifest 资源配置文件是 JAR 归档中所包含的特殊文件，Manifest 文件被用来定义扩展或档案打包相关数据。

Manifest 文件本身是一个元数据文件，包含了不同部分中的键-值对数据。如果一个 JAR 文件被当作可执行文件，则其中的 Manifest 文件需要指出该程序的主类文件。

通常 Manifest 文件的文件名为 MANIFEST.MF，而且 Manifest 文件都与 Java 档案相关，其他的情况比较少见。

所有的 Java EE 容器都支持.WAR 文件，已部署的应用程序，其版本很容易辨别。不过，使用 WAR 文件进行 Web 部署的缺陷在于即便是细微的修改，也不能在程序运行时进行，任何修改都需要重新生成和部署整个 WAR 文件。

31.8 tar

compress 与 gzip 可以用于单个文件的压缩，但是如果是要将一个目录压缩成一个文件，可以使用 tar。

tar 是 Unix 和类 Unix 系统上的压缩打包工具，最初的设计目的是将文件备份到磁带上（tape archive），现在已经发展成为一个多用途的压缩指令，可以用于将多个文件合并为一个文件，打包后的文件名亦为“tar”。

现在，tar 文件格式已经成为 POSIX 标准，最初是 POSIX.1-1988，目前是 POSIX.1-2001。

tar 可以将整个目录或者是指定的文件都整合成一个文件，例如 tar 可以将/etc 下的文件全部整合成一个文件。同时，tar 可以配合 gzip（gzip 的功能已经已经附加上 tar 中了）进行整合和压缩。

tar 代表未被压缩的 tar 文件，已被压缩的 tar 文件则追加压缩文件的扩展名，例如经过 gzip 压缩后的 tar 文件，扩展名为“.tar.gz”。

常用的 tar 是自由软件基金会开发的 GNU 版，同时也有多个压缩率不同的版本（如 tar.xz 和 tar.gz），其中 tar.xz 的压缩率更高，不过可能有兼容性问题。

- .tgz 等价于.tar.gz
- .tbz 与 tb2 等价于.tar.bz2
- .taz 等价于.tar.Z
- .tlz 等价于.tar.lzma
- .txz 等价于.tar.xz

GNU tar 的命令格式为：`tar 功能选项文件`，可以将代表功能和选项的单个字母合并。如果使用单个字母，可以不用在字母前面加“-”。

```
#tar [-cxtzjvfpPN] 文件与目录...
```

参数

-c，--create：创建新的tar文件；
-x，--extract，--get：解开tar文件；
-t，--list：列出tar文件中包含的文件的信息；

特别注意，在参数中c/x/t仅能存在一个，不可同时存在，不可能同时压缩与解压缩。
-r，--append：附加新的文件到tar文件中；
-u，--update：用已打包的文件的较新版本更新tar文件；
-A，--catenate，--concatenate：将tar文件作为一个整体追加到另一个tar文件中；
-d，--diff，--compare：将文件系统里的文件和tar文件里的文件进行比较；
--delete：删除tar文件里的文件，该功能不能用于已保存在磁带上的tar文件；
-z，--gzip，--gunzip，--ungzip：调用gzip执行压缩或解压缩。
-Z，--compress，--uncompress：调用compress执行压缩或解压缩。
-j，
--bzip2：调用bzip2执行压缩或解压缩。部分旧版本的tar使用-I实现本功能，因此编写脚本时，最好使用-
-J，--xz，--lzma：调用XZ Utils执行压缩或解压缩，依赖XZ Utils。
-k，--keep-old-files：不覆盖文件系统上已有的文件
-v，--verbose：列出每一步处理涉及的文件的信息，不建议用在背景执行过程。只用一个“v”时仅列出文件名，
-f，--file
[主机名:]文件名：指定要处理的文件名，可以用“-”代表标准输出或标准输入。
注意，在f之后要立即跟文件名，不要再加参数。例如，使用tar -zcvfP tfile
sfile就是错误的写法，要写成tar -zcvPf tfile sfile。
-p 使用原文件的原来属性（属性不会依据使用者而变）。
-P，--absolute-names：使用绝对路径进行压缩。
-N 比后面接的日期(yyyy/mm/dd)还要新的才会被打包进新建的文件中
--exclude FILE 在压缩的过程中，不要将FILE打包。

范例一 将整个/etc目录下的文件全部打包成为/tmp/etc.tar。

```
[root@linux ~]#tar -cvf /tmp/etc.tar /etc <== 仅打包，不压缩；
[root@linux ~]#tar -zcvf /tmp/etc.tar.gz /etc <== 打包后，以 gzip 压缩；
[root@linux ~]#tar -jcvf /tmp/etc.tar.bz2 /etc <== 打包后，以 bzip2 压缩。
#特别注意，在参数 f 之后的文件文件名是自己取的，我们习惯上都用.tar 来作为辨识。
#如果加 z 参数，则以.tar.gz 或.tgz 来代表 gzip 压缩过的 tar file。
#如果加 j 参数，则以.tar.bz2 来作为扩展名。
#上述指令在执行的时候会显示一个警告信息
#tar: Removing leading ‘/’ from member names，那是关于绝对路径的特殊设定。
```

范例二 查阅上述/tmp/etc.tar.gz文件内有哪些文件？

```
[root@linux ~]#tar -ztvf /tmp/etc.tar.gz
#由于我们使用 gzip 压缩，所以要查阅该 tar file 内的文件时，
#就得要加上 z 这个参数了，这很重要的
```

范例三 将 /tmp/etc.tar.gz 文件解压缩在 /usr/local/src下

```
[root@linux ~]#cd /usr/local/src
[root@linux src]#tar -zxvf /tmp/etc.tar.gz
#在预设的情况下，我们可以将压缩文件在任何地方解开，以这个范例来说，
#我将工作目录变换到/usr/local/src 下，并且解开 /tmp/etc.tar.gz ，
#则解开的目录会在/usr/local/src/etc，另外如果进入 /usr/local/src/etc
#则会发现，该目录下的文件属性与/etc/可能会有所不同。
```

范例四 在/tmp下，只想要将/tmp/etc.tar.gz内的etc/passwd解开而已

```
[root@linux ~]#cd /tmp
[root@linux tmp]#tar -zxvf /tmp/etc.tar.gz etc/passwd
#可以通过 tar -ztvf 来查阅 tarfile 内的文件名称，如果单只要一个文件，
#就可以通过这个方式来下达，注意 etc.tar.gz 内的根目录/是被拿掉。
```

范例五 将/etc/内的所有文件备份下来，并且保存其权限。

```
[root@linux ~]#tar -zcvpf /tmp/etc.tar.gz /etc
#这个-p 的属性是很重要的，尤其是当要保留原本文件的属性时。
```

范例六 在/home当中，比2005/06/01新的文件才备份

```
[root@linux ~]#tar -N '2005/06/01' -zcvf home.tar.gz /home
```

范例七 要备份/home，/etc，但不要/home/dmtsai

```
[root@linux ~]#tar -exclude /home/dmtsai -zcvf myfile.tar.gz /home/* /etc
```

范例八 将/etc/打包后直接解开在/tmp，而不产生文件！

```
[root@linux ~]#cd /tmp
[root@linux tmp]#tar -cvf - /etc | tar -xvf -
#这个动作有点像是 cp -r /etc /tmp 啦～依旧是有其有用途的！
#要注意的地方在于输出档变成 - 而输入档也变成 - ，又有一个 | 存在～
#这分别代表 standard output，standard input 与管道命令
```

tar 用来作备份是很重要的指令，tar 整合过后的文件我们通常会取名为 *.tar，而如果还含有 gzip 的压缩属性，那么就取名为 *.tar.gz，这样命名只是为了方便记忆文件属性，并没有实际的意义。

```
tar -cvf home_backup.tar /home
```

可以将/home 目录下的所有文件打包入 home_backup.tar 文件中。理解 tar 命令时，注意“home_backup.tar”实际上是-f 选项的参数。

tar 默认记录相对路径，即使给出的是绝对路径，也会自动将代表根目录的“/”去掉，所以使用“/home”和“home”是相同的。

如果需要使用绝对路径，则要加上“P”选项，但是一般不推荐使用绝对路径，原因之一是可能导致 tar 炸弹攻击。

攻击者利用绝对路径或者“`tar -cf bomb.tar *`”的方式创建的 tar 文件，然后诱骗受害者在根目录下解压，或者使用绝对路径解压。可能使受害系统上已有的文件被覆盖掉，或者导致当前工作目录凌乱不堪，这就是所谓的“tar 炸弹”。因此，要养成良好的解压习惯：

- 解压前用“t”查看 tar 的文件内容。
- 拒绝使用绝对路径。
- 新建一个临时子目录，然后在这个子目录里解压。

如果在 tar 输出的警告信息中有“tar: Removing leading ‘/’ from member names”，则说明 tar 去掉了/etc 目录中的/去掉，这是因为担心未来在解压缩的时候会产生一些困扰。例如，在 tar 里面的文件如果是具有“绝对路径”的话，那么解压缩后的文件将会“一定”在该路径下也就是/etc，而不是相对路径。

在预设的情况中，如果是以“绝对路径”来建立打包文件，那么 tar 将会自动的将/去掉，因此这是为“安全”前提所做的预设值。

如果要以绝对路径来建立打包的文件，那么就需要加入-P 这个参数（注意是大写字符）。

-p（小写字符）是 permission 的意思，因此使用-p 打包的文件将不会依据使用者的身份来改变权限。

```
cd /home
 tar -cvf home_backup.tar *
```

这也是一种制作备份的方法，但是不推荐这样做。因为 tar 在默认解压时，会将文件直接输出到当前目录下，而不会新建并输出到一个名为 home 的子目录，令到当前目录显得很凌乱。这也是一种形式的 tar 炸弹攻击。

```
tar -tf home_backup.tar
```

列出 home_backup.tar 文件里已被打包的文件。此时仅仅显示文件名。如果加上“v”，则能列出权限、所有者、大小、时间、文件名等信息。为防止 tar 炸弹攻击，应该养成解压前查看 tar 文件内容的好习惯。

```
tar -xvf home_backup.tar
```

在当前目录下解压 home_back.tar。解压后的文件，其访问权限得到保留；其所有者是执行 tar 命令的用户，如果 tar 的执行者是 root，则所有者是文件原来的所有者。解压前，最好先查看 tar 文件的内容，以决定是否需要新建一个临时子目录安放。

```
tar -xvf home_backup.tar home/test.c
```

指定解压出 test.c 这个文件。解压过程中会自动创建 home 这个子目录。

这里还有一个值得注意的参数，那就是在备份的情况中很常使用的-N 的这个参数，在备份时都希望只要备份较新的文件。如果旧的文件已经有备份，如果再备份一次浪费时间也浪费系统资源。

如果需要把 tar 和管道配合，文件默认是标准输入/输出，不需再额外指定。

```
tar -c "${源目录}" | tar -xvC "${目标目录}"
```

可以将源目录下的文件及子目录复制到目标目录中，尤其适用于复制含有特殊文件 (如软链接、设备文件) 的目录。

如果直接以管道命令“pipe”来进行压缩、解压缩，可以使用-字符。

```
tar cvf - /etc | tar -xvf -
```

例如，如果需要将“/etc 下的数据直接 copy 到目前所在的路径”，那么就可以直接以 tar 的方式来打包，其中指令里面的 - 就表示被打包的文件。

tar 是经由“打包”之后再处理的一个过程，因此 tarball 文件就是通过 tar 打包再压缩的文件，仅是打包而没有压缩的文件称为 tarfile。

此外，tar 也可以用在备份的存储介质上（例如磁带机)。例如，假设磁带机代号为/dev/st0，那么要将/home 下的数据都备份到磁带机上时，可以执行如下的命令：

```
tar /dev/st0 /home
```

如果 tar 没有整合 gzip，那么在需要解压缩时可以执行如下的命令

```
gzip -d testing.tar.gz
tar -xvf testing.tar
```

其中，首先将文件包解开，然后将数据解压缩出来。

另外，与其他压缩程序不太一样的是，bzip2、gzip 与 compress 在没有加入特殊参数时，原先的文件会被覆盖，但是使用 tar 时将不会进行覆盖，因此原来的与后来的文件都会存在。

随着备份策略的进步，逐渐采用 dump、restore 等工具替代 tar，因此现在 tar 一般与 gzip 联合使用来弥补后者无法将多个文件打包的不足，新的 tar 版本已能自动调用多种压缩工具执行压缩。

已压缩的 tar 文件也叫“tarball”，大部分自由软件的源代码采用 tarball 的形式发布。

Option	Mode	Description
(none)	”list”	shows contents of archive, does not modify or extract anything.
-r	”read”	reads and extracts contents of an archive
-w	”write”	creates archives or appends files to an archive
-rw	”copy”	reads and copies files and directory tree to a specified directory

31.9 pax

pax is an archiving utility created by POSIX and defined by the POSIX.1-2001 standard. Rather than sort out the incompatible options that have crept up between tar and cpio, along with their implementations across various versions of UNIX, the IEEE designed a new archive utility.

The name ”pax” is an acronym for portable archive exchange. Furthermore, ”pax” means ”peace” in Latin, so its name implies that it shall create peace between the tar and cpio format supporters. The command invocation and command structure is somewhat a unification of both tar and cpio.

pax has four general modes which are invoked by a combination of the -r (”read”) option and -w (”write”) option.

List contents of an archive:

```
pax < archive.tar
```

Extract contents of an archive into the current directory:

```
pax -r < archive.tar
```

Create an archive of the current directory, when used in the cpio style, the find command can be used to get a list of files to be archived:

```
find . -depth -print | pax -wd > archive.tar
```

Copy current directory tree to another location, the target directory must exist beforehand.

```
find . -depth -print | pax -rwd target_dir
```

pax can be either used in a similar manner as cpio or tar. The cpio syntax takes a list of files from standard input (stdin) when archiving or an already existing archive, when in listing contents or extracting files:

```
find . -depth -print | pax -wd > archive.tar
pax -r < archive.tar
```

It is possible to invoke these commands in a tar-like syntax as well:

```
pax -wf archive.tar .
pax -rf archive.tar
```

Listing files from an archive:

```
pax -f archive.tar
```

and ”copy” mode:

```
pax -rw . archive_dir
```

The -f option specifies which archive to use, instead of writing to stdio or reading from stdin. Also note the -d option when using pax together with find, this keeps pax from traversing directory trees.

31.10 dd

dd 命令由单一 UNIX 规范的一部分，IEEE 标准 1003.1-2008 所规定。

dd 只做好一件事（并被认为做得“好”）。与复杂的和高度抽象的实用程序不同，除了为不同的选项做底层决定，dd 没有其它的算法。一般在每一次运行时，会改变 dd 的选项以分步处理一个计算机问题。

dd 命令可以用于数据传输、MBR 备份和恢复、数据修改、数据擦除、数据恢复、性能测试和数据转换等，其命令行语句与其他的 Unix 程序不同，因为它的命令行选项格式为选项 = 值，而不是更标准的--选项值或-选项 = 值。

dd 命令可以用于在文件、设备、分区和卷之间进行数据复制。例如，使用 cp 来进行硬盘对拷时可能会忽略最后一个块，但是使用 dd 命令可以进行完整拷贝。

```
// create an iso disk image from a cd-rom
dd if=/dev/sr0 of=CD.iso bs=2048 conv=noerror,sync
// clone one partition to another
dd if=/dev/sda2 of=/dev/sdb2 bs=4096 conv=noerror
// clone one hard disk "ad0" to "ad1''
dd if=/dev/ad0 of=/dev/ad1 bs=1M conv=noerror
```

dd 可以读取设备的内容，然后将整个设备备份成一个文件，因此 dd 指令最大的用途应该是“备份”。

默认情况下，dd 从标准输入中读取，并写入到标准输出中，但可以用选项 if（input file，输入文件）和 of（output file，输出文件）改变。

```
[root@linux ~]#dd if="input_file" of="output_file" bs="block_size"
    count="number"
```

参数

if : 就是 input file 棉～也可以是设备喔!

of: 就是 output file 喔～也可以是设备;

bs: 规划的一个block的大小，如果没有设定时，预设是512 bytes

count 多少个 bs 的意思。

范例：

范例一：将/etc/passwd 备份到/tmp/passwd.back 当中

```
[root@linux ~]#dd if=/etc/passwd of=/tmp/passwd.back
3+1 records in
3+1 records out
[root@linux ~]#ll /etc/passwd /tmp/passwd.back
-rw-r--r--  1 root root 1746 Aug 25 14:16 /etc/passwd
-rw-r--r--  1 root root 1746 Aug 29 16:57 /tmp/passwd.back
#/etc/passwd 文件大小为 1746 bytes，因为没有设定 bs，
#所以预设是 512 bytes 为一个单位，因此上面那个 3+1 表示有 3 个完整的
#512 bytes，以及未满 512 bytes 的另一个 block 的意思。
#事实上，感觉好像是 cp 这个指令。
```

范例二：备份/dev/hda 的 MBR

```
[root@linux ~]#dd if=/dev/hda of=/tmp/mbr.back bs=512 count=1
1+0 records in
1+0 records out
#我们知道整颗硬盘的 MBR 为 512 bytes，
#就是放在硬盘的第一个 sector，因此可以利用这个方式来将
#MBR 内的所有数据都记录下来，真的很厉害吧!
```

范例三：将整个 /dev/hda1 partition 备份下来。

```
[root@linux ~]#dd if=/dev/hda1 of=/some/path/filenaem
#这个指令很厉害啊! 将整个 partition 的内容全部备份下来～
#后面接的 of 必须要不是在 /dev/hda1 的目录内啊～否则，怎么读也读不完～
#这个动作是很有效用的，如果改天你必须要完整的将整个 partition 的内容填回去，
#则可以利用 dd if=/some/file of=/dev/hda1 来将数据写入到硬盘当中。
#如果想要整个硬盘备份的话，就类似 Norton 的 ghost 软件一般，
#由 disk 到 disk ，利用 dd 就可以。
```

实际上，dd 的主要功能在于转换和复制文件，tar 可以用来备份关键数据，而 dd 则可以用来备份整块 partition 或整个 disk。

UNIX 上的硬件设备驱动（如硬盘）和特殊设备文件（如/dev/zero 和/dev/random）可以和普通文件一样出现在文件系统中，因此只要在各自的驱动程序中实现了对应的功能，dd 也可以读取自和/或写入到这些文件。

dd 也可以用在备份硬件的引导扇区、获取一定数量的随机数据等任务中，而且还可以在复制时处理数据，例如转换字节序、或在 ASCII 与 EBCDIC 编码间互换。

不过，如果要将数据填回到 filesystem 当中，可能需要考虑到原本的 filesystem 才能成功。

另外，dd 的一些特定功能取决于计算机系统的能力（例如直接访问内存）。如果向运行中的 dd 进程发送 SIGINFO 信号（Linux 上为 USR1）可以使它将 I/O 统计信息打印到标准错误一次，然后继续复制（注意在 OS X 上，信号可能导致进程终止）。

dd 可以从键盘中读取标准输入，在到达文件结尾时，dd 将会退出。

信号和 EOF 是由软件决定。例如，移植到 Windows 的 Unix 工具使用不同的 EOF：

- Cygwin 使用 <ctrl-d>（通常的 Unix EOF）；
- MKS 工具箱使用 <ctrl-z>（通常的 Windows EOF）。

GNU coreutils 提供的变种没有描述运行结束时，dd 输出到标准输出消息的格式，不过 BSD 等其他的实现对其进行了描述。

“记录读入”和“记录写出”行显示了已完整传输的块数 + 不完整的块数，例如物理介质以不完整的块结尾，或是一个物理错误使得一个完整的块无法被读取。

为了显示 dd 操作的进度，可以使用 pkill、killall、kill 向 dd 命令发送 SIGUSR1 信息，dd 命令进程接收到信号之后就打印出自己当前的进度。

```
watch -n 5 pkill -USR1 ^dd$
watch -n 5 killall -USR1 dd
while killall -USR1 dd; do sleep 5; done
while (ps auxww |grep " dd " |grep -v grep |awk '{print $2}' |while read pid; do kill -USR1 $pi
```

31.10.1 Block Size

块是衡量一次读取、写入和转换字节的单位。

命令行选项可以为输入/读取（ibs）和输出/写入（obs）指定一个不同的块大小，尽管块大小（bs）选项会覆盖 ibs 和 obs 选项。

输入和输出的默认块大小为 512 字节（传统的磁盘块及 POSIX 规定的“块”大小）复制的 count 选项、读取的 skip 选项和写入的 seek 选项都是以块为单位。转换操作也受“转换块大小”（cbs）影响。

块大小可能会影响 dd 的某些应用的表现。例如，当转换硬盘中数据时，较小的块大小通常会导致更多的字节被转换。发出许多小块的读取是一种开销的浪费，且可能会对执行性能有负面影响。较大的块大小可能会提高复制速度。但是，由于要复制的字节量是由 bs×count 给出的，因此不可能在一次 dd 命令中复制素数个字节，除非使用两个糟糕选项之一：bs=N count=1（消耗内存）或 bs=1 count=N（大量读请求开销）。

dd 的替代程序允许指定字节而不是块，否则在用作网络传输时，根据使用的网络协议，块大小可能会与包大小冲突。

提供给块大小的值会被解释成十进制整数，也可以加入后缀指定倍数。后缀 w 表示 2 倍，b 表示 512 倍，k 表示 1024 倍，M 表示 1024×1024 倍，G 表示 $1024 \times 1024 \times 1024$ 倍，等等。

另外，在块大小和计数参数中，一些实现也可以使用 x 表示乘运算。例如，块大小 bs=2x80x18b 表示 $2 \times 80 \times 18 \times 512 = 1474560$ 字节，也就是一张 1440 KiB 软盘的确切大小。

31.10.2 Data Transform

dd 可以在文件、设备、分区和卷之间复制数据，而且数据可以从其中任何地方输入或输出，但是输出到分区时有重要差异。

- noerror 选项意味着如果发生错误，程序也将继续运行。
- sync 选项表示填充每个块到指定字节。

另外，在传输过程中，数据可以用 conv 选项修改以适应介质。

如果最后一个块有意外长度，试图使用 cp 复制整个磁盘可能会忽略掉它，不过 dd 却可能成功，只是源和目标磁盘应该具有相同的大小。

从CD-ROM中创建ISO磁盘镜像

```
dd if=/dev/sr0 of=myCD.iso bs=2048 conv=noerror,sync
```

克隆一个分区到另一个

```
dd if=/dev/sda2 of=/dev/sdb2 bs=4096 conv=noerror
```

克隆硬盘“ad0”到“ad1”

```
dd if=/dev/ad0 of=/dev/ad1 bs=1M conv=noerror
```

31.10.3 MBR Backup/Recovery

dd 命令可以用来修复 MBR（Master Boot Record），这样就可以将主引导记录转移到文件，或从中转移出来。

```
// duplicate the first two sectors of a floppy drive
dd if=/dev/fd0 of=MBRBoot.img bs=512 count=2
// create mbr img(include a MS-DOS partition table and MBR magic bytes
```

```
dd if=/dev/sda of=MBR.img bs=512 count=1
// create an image of only the boot code of the mbr(without partition
    table and magic bytes)
dd if=/dev/sda of=MBR_boot.img bs=446 count=1
```

要复制软盘的前两个扇区

```
dd if=/dev/fd0 of=MBRboot.img bs=512 count=2
```

要创建整个x86主引导记录的镜像（包括MS-DOS分区表和MBR魔法字节）

```
dd if=/dev/sda of=MBR.img bs=512 count=1
```

要创建仅含主引导记录引导代码的镜像（不包括分区表和开机所需的魔法字节）

```
dd if=/dev/sda of=MBR_boot.img bs=446 count=1
```

31.10.4 Data Modification

dd 命令可以用来进行数据修改、数据覆盖以及 dd 可以原地修改数据等。

用空字节覆盖文件的前512个字节：

```
dd if=/dev/zero of=path/to/file bs=512 count=1 conv=notrunc
```

这里，转换选项 notrunc 意味着不缩减输出文件，也就是说，如果输出文件已经存在，只改变指定的字节，然后退出，并保留输出文件的剩余部分。没有这个选项，dd 将创建一个 512 字节长的文件。

```
// overwrite the first 512 bytes of a file with null bytes
dd if=/dev/zero of=path/to/file bs=512 count=1 conv=notrunc
```

dd 命令可以用来将磁盘分区备份为镜像文件，从而可以在其他地方进行挂载。

```
dd if=/dev/sdb2 of=partition.img bs=4096 conv=noerror
```

在不同的分区中复制磁盘分区到磁盘镜像文件中：

```
dd if=/dev/sdb2 of=partition.image bs=4096 conv=noerror
```

31.10.5 Disk Wipe

出于安全方面的考虑，有时需要擦除丢弃的磁盘。

检查驱动器上是否有数据，并将其输出到标准输出：

```
dd if=/dev/sda
```

用零擦除磁盘：

```
dd if=/dev/zero of=/dev/sda bs=4k
```

为了安全的进行数据擦除，使用 dd 命令可以将原来的数据覆盖为/dev/zero 或随机数据来实现。

```
dd if=/dev/zero of=/dev/sda bs=4k
dd if=/dev/urandom of=/dev/sda bs=4k
```

相较于上面数据修改的例子，不需要使用转换选项 notrunc，因为当 dd 的输出文件为块设备时，它没有效果。

bs=4k 选项使 dd 一次读取或写入 4 千字节。在现代系统中，由于传输容量（如 RAID 系统），一个更大的块大小可能更有利。注意用随机数据填充磁盘总是比用零慢的多，因为随机数据必须先由 CPU 和/或 HWRNG 生成，且不同的设计有不同的性能特点。（后面 PRNG 的/dev/urandom 可能比 libc 中的要慢。）在大多数较现代的磁盘中，用零擦除会使其中的数据永久丢失。

用零擦除磁盘会使它的数据无法被软件恢复，然而数据仍可能用特殊的实验室技术恢复。

wipe 命令可以执行数据擦除任务，效果和 dd 命令相同。另外，shred 程序提供了完成相同任务的替代方法。

31.10.6 Data Recovery

1984 年，GNU dd 开启了开源软件（OSS）恢复数据、文件、驱动器和分区的历史，dd 进程一次处理一个块，它的算法只是在用户界面显示运行状态。

1999 年 10 月，dd_rescue 的算法可以一次能处理两个块，但是改进 dd_rescue 的数据恢复算法的 dd_rhelpshell 脚本的作者现在推荐 GNU ddrescue。

2004 年发布的 ddrescue 有最先进的块大小变换算法，尽管 ddrescue 和 dd_rescue 名字相近，但是不同的程序。相比之下，GNU ddrescue 既稳定又安全。

savehd7 使用更复杂的算法，但它需要安装自己的语言解释器。

31.10.7 Driver Benchmark

对驱动器进行基准测试（通常是单线程），使用 1024 字节块分析连续系统读取和写入的性能：

```
dd if=/dev/zero bs=1024 count=1000000 of=file_1GB
dd if=file_1GB of=/dev/null bs=1024
```

使用内核随机数驱动，用 100 个随机字节生成文件：

```
dd if=/dev/urandom of=myrandom bs=100 count=1
```

将文件转换为大写：

```
dd if=filename of=filename1 conv=ucase
```

创建 1GiB 的稀疏文件，或增加现有文件的大小：

```
dd if=/dev/zero of=mytestfile.out bs=1 count=0 seek=1G
```

更先进的工具是 GNU coreutils 中的 fallocate 或 truncate。

在对硬盘进行底层操作时，类似颠倒输入和输出文件的一个小错误都可能造成部分或全部硬盘数据的丢失。

希捷的文档警告说，“一些依赖底层硬盘访问的硬盘工具（如 DD）可能不支持 48 位逻辑区块地址（LBA），除非进行升级”。使用超过 128 GiB 的 ATA 硬盘时需要 48 位 LBA。

在 Linux 中，dd 使用内核读取或写入原始设备文件，并且在 2.4.23 版本内核中已经实现了对 48 位 LBA 的支持。

dcfldd 是 dd 的一个分支，与 dd 相比，dcfldd 允许一个以上的输出文件，同时支持多种校验计算方法，还提供了验证模式以匹配文件，并能显示操作进度百分比。

31.11 shred

shred 用于安全删除文件与设备数据使之难以被恢复，而且 shred 也是 GNU 核心工具组的组成部分之一。

为保证效率，删除某一文件一般只需删除其在文件系统中的对应项，实际的文件内容则仍完好无损，这一特性也多为常见的数据恢复软件所利用以恢复文件。

假使该文件内容被其他内容所覆盖，专业的数据恢复设备也能利用本有存储内容的剩余磁场来恢复原始内容。针对这些特点，shred 以用能最大程度破坏原始数据的模式多次覆写文件的方式来达到安全删除的目的。

shred 可以对普通文件或设备（在 Unix 与类 Unix 系统中设备也是一类文件）调用。若使用默认设定，shred 就会以多种模式覆写文件 3 遍，但覆写的次数也可以由用户以使用-n 的方式来定义。在使用-z 的情况下，shred 还可以在最后一次覆写时将文件全部以 0 覆写，这也可借以掩盖文件曾用 shred 覆写过的痕迹。

默认情况下，shred 也会覆写文件系统里为文件分配的空间中的空闲空间。例如，在一簇占用 4KB 的文件系统中存储一个 5KB 的文件，那该文件实际分配的空间就是 8KB（2 簇），而对该文件执行 shred 时就会将这 8KB 空间全部覆写。但是，若额外使用-x 选项就可以避免覆写空闲空间。

除此以外，shred 还可搭配-u 选项以在覆写完毕后自动删除文件。

shred 的局限性之一是当对某一普通文件使用时，shred 只会覆写该文件而不会覆写文件的其他副本，而如手动/自动备份、文件系统快照、文件系统的 COW 机制、闪存介质的

耗损平均机制、网络文件系统的缓存机制和日志文件系统的日志机制都可能导致文件副本的产生。

文件系统本身的机制所产生的局限性都可以用一个办法解决，即是直接对包含目标数据的所有设备使用 shred，但对于应用了耗损平均技术的设备来说，数据分配所得的逻辑块与实际的物理位置并不能保证是固定关联的，因而对设备进行 shred 也无法保证有足够的安全性。

在这种情况下，SATA 设备的安全擦除命令（如 hdparm）就是更好的选择，对磁性存储设备来说使用 SATA 安全擦除工具也更快且更可靠。除此之外，对于无法使用的硬盘等设备就可以考虑用物理破坏的方式使内容不可还原了。

31.12 cpio

cpio 是 UNIX 操作系统的一个文件备份程序及文件格式，最初在 PWB/UNIX 中用于备份磁带，后来被引进到 UNIX System III 及 System V。

cpio 指令是通过数据流重导向的方法来将文件进行输出/输入的一个方式。

```
[root@linux ~]#cpio -covB > [file|device] <== 备份
[root@linux ~]#cpio -icduv < [file|device] <== 还原
```

参数

−o 将数据 copy 输出到文件或设备上
−i 将数据自文件或设备 copy 出来系统当中
−t 查看 cpio 建立的文件或设备的内容
−c 一种较新的 portable format 方式存储
−v 让存储的过程中文件名称可以在显示器上显示
−B 让预设的Blocks可以增加至5120 bytes，预设是512 bytes，这样的好处是可以让大文件的存储速度加快。
−d 自动建立目录，由于cpio的内容可能不是在同一个目录内，

如此的话在反备份的过程会有问题，这个时候加上−d就可以自动的将需要的目录建立起来了。
−u 自动的将较新的文件覆盖较旧的文件。

范例 范例一 将所有系统上的数据通通写入磁带机内。

```
[root@linux ~]#find / -print | cpio -covB > /dev/st0
#一般来说，使用 SCSI 介面的磁带机，代号是 /dev/st0。
```

范例二 检查磁带机上面有什么文件？

```
[root@linux ~]#cpio -icdvt < /dev/st0
[root@linux ~]#cpio -icdvt < /dev/st0 > /tmp/content
```

```
#第一个动作当中，会将磁带机内的文件名列出到显示器上面，而我们可以通过第二个动作，

#将所有的文件名通通记录到 /tmp/content 文件去!
```

范例三：将磁带上的数据还原回来～

```
[root@linux ~]#cpio -icduv < /dev/st0
#一般来说，使用 SCSI 介面的磁带机，代号是 /dev/st0 喔!
```

范例四：将 /etc 底下的所有『文件』都备份到 /root/etc.cpio 中!

```
[root@linux ~]#find /etc -type f | cpio -o > /root/etc.cpio
#这样就能够备份棉～您也可以将数据以 cpio -i < /root/etc.cpio
#来将数据捉出来。
```

cpio 是最适用于备份的时候使用的一个指令，它并不像 cp 一样，可以直接的将文件 copy 过去，例如 cp * /tmp 就可以将所在目录的所有文件 copy 到/tmp 下。

在 cpio 这个指令的用法中，由于 cpio 无法直接读取文件，而是需要“每一个文件或目录的路径连同文件名一起”才可以被记录下来，因此 cpio 最常跟 find 这个指令一起使用。

cpio 是备份的时候的一项利器，它可以备份任何的文件，包括/dev 下的任何设备文件，而由于 cpio 必需要配合其他的程序，例如 find 来建立文件名，所以 cpio 与管道命令及数据流重导向的相关性就相当的重要了。

cpio 可以从 cpio 或 tar 格式的归档包中存入和读取文件, 归档包是一种包含其他文件和有关信息的文件。有关信息包括：文件名, 属主, 时标 (timestamp), 和访问权限。归档包可以是磁盘上的其他文件, 也可以是磁带或管道.

使用以下命令可以用当前目录下的所有文件和文件夹来创建新的 cpio 归档文件：

```
find * -depth -print | cpio -H newc -o > /somepath/archive.cpio
```

31.13 dump

dump 命令可以针对整个文件系统进行备份，也可以仅针对目录进行备份。

在使用 dump 来备份整个文件系统时，可以进行完整备份、差异备份和增量备份，即在后续的备份中可以指定是否只备份有差异的文件。

- level 0：完整备份
- level 1：差异备份
- level 2：增量备份

如果待备份的数据为单一文件系统，那么该文件系统可以使用完整的 dump 功能，并且可以指定 0 ～ 9 的备份等级。同时，dump 可以使用挂载点或者设备文件名来执行备份。

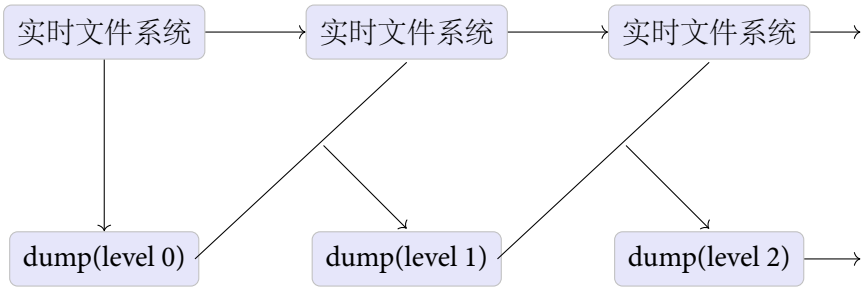


Figure 31.1: dump 的运行等级

```
dump [-level#] [-ackMnqSuv] [-A file] [-B records] [-b blocksize]
[-d density] [-D file] [-e inode numbers] [-E file] [-f file] [-F script]
[-h level] [-I nr errors] [-jcompression level] [-L label] [-Q file]
[-s feet] [-T date] [-y] [-zcompression level] files-to-dump

dump [-W | -w]
```

如果待备份的数据只是目录（并非单一文件系统），那么所有的备份数据都必须在目录中，而且仅能使用 level 0 进行备份。

在使用 dump 对目录进行备份时，不支持-u 参数，即无法创建/etc/dumpdates 时间记录文件。

一般来说，dump 不会使用包含压缩的功能，-j 参数可以加入 bzip2 支持，默认 bzip2 的压缩等级为 2。

31.14 restore

与 dump 相对的是 restore，用于从备份文件中恢复数据。

如果需要查看备份数据的内容，可以使用 restore 命令的-t 参数。

如果使用 dump 备份的是独立的文件系统，在恢复时可以一个全新的文件系统来执行还原操作，不过要更改目录到相应的挂载点。

为了仅还原备份文件中的部分文件，可以使用 restore 的交互模式。

```
# restore -i -f /home/backup.dump
restore >
```

Chapter 32

File Split

32.1 split

split 可以用于将大文件根据容量或行数分割成小文件。

```
split [OPTION] [INPUT [PREFIX]]
```

- -b 指定分割文件的大小，并支持 b、k 和 m 等尺寸单位；
- -l 指定以行数进行分割。

默认情况下，split 将大文件分割成固定大小的小文件（1000 行），并且将分割产生的文件名追加 aa、ab、ac 等标识符。例如，在不指定输出文件名的情况下，一个名字为 x 的文件被分割后产生的文件分别为 xaa、xab 和 xac 等。

如果使用-来代替输入文件，那么 split 从标准输入中获得数据。

下面的示例说明将文件分割成 50M 大小的小文件，并且指定分割后的小文件名以 part 开头。

```
split -b50m filename part
```

为了把分割后的文件合并，可以使用 cat 命令配合重定向来实现。

```
cat xaa xab xac > filename
cat xa[a-c] > filename
cat xa? > filename
```

另外，split 可以指定分割文件时的最大字符数、最大行长度、分割后的文件名长度，以及是否使用字母或数字等。

32.2 csplit

csplit is used to split file by content rather than by size

Chapter 33

File Image

如果需要将文件保存到光盘中，可以按照如下的方式进行。

- 使用 mkisofs 命令将需要备份的数据构建成镜像文件；
- 使用 cdrecord 命令将镜像文件刻录到光盘中。

33.1 mkisofs

mkisofs 命令用于将数据打包成镜像文件。

```
$ genisoimage [options] [-o filename] pathspec [pathspec ...]
```

如果需要保留比较完整的文件信息（例如 UID/GID 和权限等），可以使用`-r` 参数。

在默认情况下，所有要被加到镜像文件中的文件都会被放置到镜像文件中的根目录，可以使用`-graft-point` 来保持原有的文件路径。

33.2 cdrecord

cdrecord 命令可以用来进行镜像刻录。

早期的刻录机使用的是 SCSI 接口，不过在使用 cdrecord 命令来查询刻录机时需要识别 SCSI/IDE/SATA 等接口的设备。

```
$ cdrecord -scanbus dev=ATA
```

在使用 cdrecord 刻录 DVD 时需要额外的 `driveropts=burnfree` 或`-dao` 等参数。

Chapter 34

File Size

34.1 size

size is a command line utility originally written for use with the Unix-like operating systems. It processes one or more ELF files and its output are the dimensions (in bytes) of the text, data and uninitialized sections, and their total.

```
$ size <option> <filename> ...
```

Here follows some examples on Solaris (/usr/ccs/bin/size); options and syntax may vary on different Operating Systems:

```
$ size  /usr/ccs/bin/size
9066 + 888 + 356 = 10310
```

With -f option name and size of each section are printed out, plus their total:

```
$ size -f /usr/ccs/bin/size
 17(.interp) + 636(.hash) + 1440(.dynsym) + 743(.dynstr) + 64(.SUNW_version) + 48(.rela.ex_shared)
 80(.init) + 80(.fini) + 4(.exception_ranges) + 28(.rodata) + 590(.rodata1) + 12(.got) + 388(.plt) +
 140(.data1) + 352(.bss) = 10086
```

With -F option size and permission flag of each sections are printed out, plus their total:

```
$ size -F /usr/ccs/bin/size
 9066(r-x) + 1244(rwx) = 10470
```

Chapter 35

LVM

35.1 Introduction

作为一种抽象化存储技术，逻辑卷管理（Logical volume management，LVM）为计算机的大容量存储设备（Mass storage devices）提供更有弹性的硬盘分区方式。

基本上，LVM 实现的方式根据操作系统而有所不同，一般都是在驱动程序与操作系统之间增加一个逻辑层来方便系统管理硬盘分区系统。

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

Welcome to fdisk (util-linux 2.24.2).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Command (m for help): m

Help:

  DOS (MBR)
   a   toggle a bootable flag
   b   edit nested BSD disklabel
   c   toggle the dos compatibility flag

  Generic
   d   delete a partition
   l   list known partition types
```

```
   n   add a new partition
   p   print the partition table
   t   change a partition type
   v   verify the partition table

  Misc
   m   print this menu
   u   change display/entry units
   x   extra functionality (experts only)

  Save & Exit
   w   write table to disk and exit
   q   quit without saving changes

  Create a new label
   g   create a new empty GPT partition table
   G   create a new empty SGI (IRIX) partition table
   o   create a new empty DOS partition table
   s   create a new empty Sun partition table


Command (m for help): p
Disk /dev/sda: 238.5 GiB, 256060514304 bytes, 500118192 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
Disklabel type: dos
Disk identifier: 0x00003459

Device    Boot     Start       End    Blocks  Id System
/dev/sda1           2048   1026047    512000  83 Linux
/dev/sda2        1026048 316420095 157697024   5 Extended
/dev/sda5        1028096 316420095 157696000  8e Linux LVM

Command (m for help):
```

35.2 Windows LVM

在一般的 Windows 操作系统中，基本存储设备（例如硬盘）可以分区成为主分区（Primary Partition）或扩展分区（Extended Partition）。

理论上，扩展分区可以有无限个，但是主要分区最多只能有四个。

从 Windows 2000 开始引入的逻辑磁盘管理工具（Logical Disk Manager）是 Microsoft 及 Veritas Software 共同开发的硬盘分区系统，在 Windows XP、Windows Server 2003、Windows Vista 及 Windows 7 中亦有提供。

在 Windows 操作系统中，把一般的未处理硬盘称为基本硬盘，基本硬盘可以“升级”成为动态硬盘。

动态磁盘的功能使操作系统可以通过软件控制来模拟 RAID 控制卡，把多个硬盘集成成为一个或多个不同架构的扇区。不过，只有 Windows 2000 或以上的系统支持动态磁盘，Linux 2.4.8 或更新的版本才支持。

从 Windows 2000 起，为了提高动态硬盘的可移植性，动态硬盘的分区数据（约占 8MB）存储在硬盘上，因此动态硬盘的可用空间比实际应有的少了大约 8MB。

理论上，动态硬盘上最多可以有 2000 个动态扇区，但是 Microsoft 建议最多只创建 32 个。

在以 MBR 分区表模式分区的磁盘上，LDM 并不存储在分区中，而是在磁盘末尾 1MiB 的、未指派到任何分区的区域。

Windows XP 的分区工具不会使用上述的 1MiB 区域，但其它操作系统上的工具可能会使用。

35.3 Linux LVM

逻辑卷管理器（Logical Volume Manager）是 Linux 核心所提供的逻辑卷管理（Logical volume management）功能。

LVM 最早由 IBM 开发，并在 AIX 系统上实现，Heinz Mauelshagen 在 1998 年根据在 HP-UX 上的逻辑卷管理器实现了 Linux LVM。

LVM 在硬盘的硬盘分区的上层创建了一个逻辑层来方便操作系统管理硬盘分区系统。

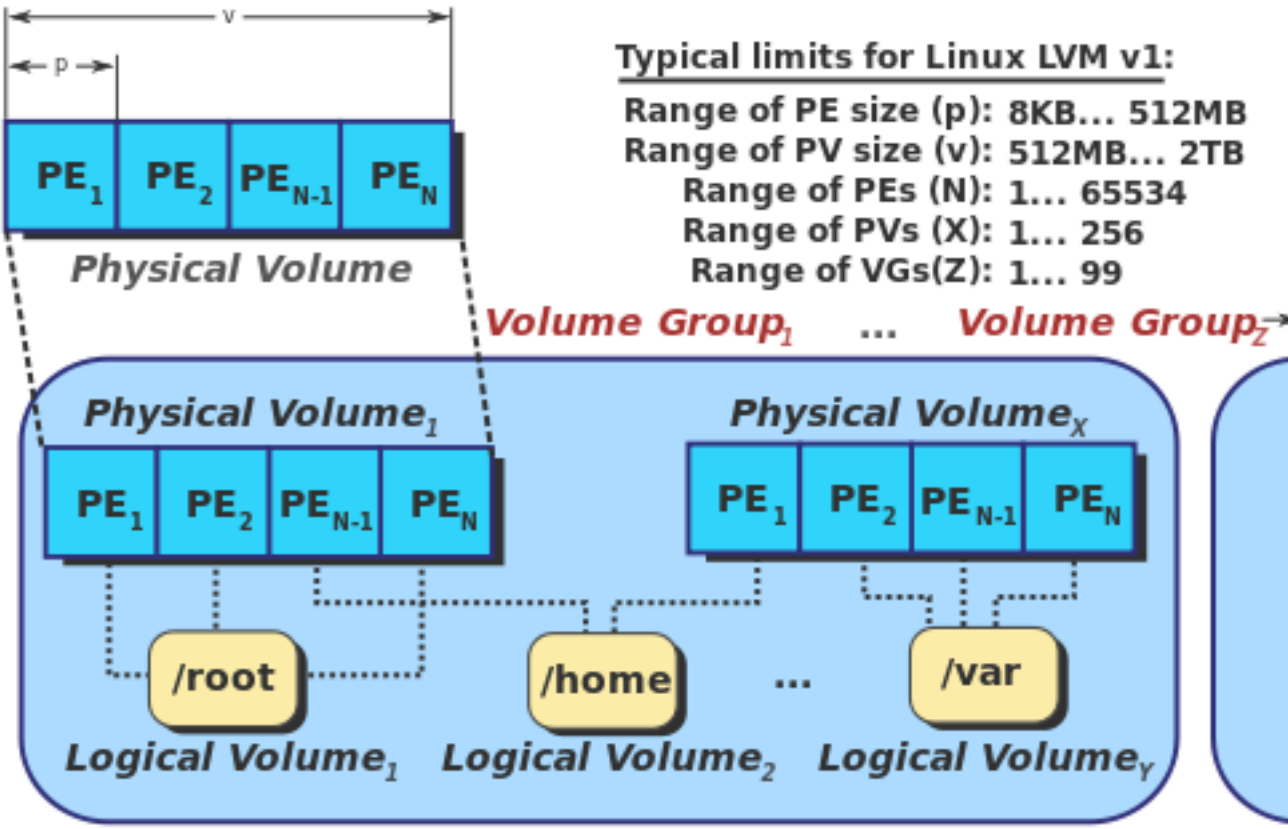


Figure 35.1: Linux Logical Volume Manager (LVM) v1

Bibliography

[1] Wikipedia. Filesystem hierarchy standard (fhs), . URL http://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard.

[2] Wikipedia. 文件系统, . URL http://zh.wikipedia.org/zh-cn/%E6%96%87%E4%BB%B6%E7%B3%BB%E7%BB%9F.

[3] Wikipedia. 文件系统层次结构标准, . URL http://zh.wikipedia.org/wiki/%E6%96%87%E4%BB%B6%E7%B3%BB%E7%BB%9F%E5%B1%82%E6%AC%A1%E7%BB%93%E6%9E%84%E6%A0%87%E5%87%86.

[4] Wikipedia. Unix filesystem, . URL http://en.wikipedia.org/wiki/Unix_filesystem.

[5] Wikipedia. Extended file system, 03 1992. URL http://en.wikipedia.org/wiki/Extended_file_system.

[6] Wikipedia. Xfs, 1993. URL http://zh.wikipedia.org/wiki/XFS.

Part IV

User

Chapter 36

Overview

Chapter 37

User

Linux 中的每个文件都有相当多的属性与权限，其中最重要的可能就是文件的所有者的概念。

37.1 owner

Linux 是多用户多任务的系统，在多人同时使用主机来进行运算时需要考虑到每个人的隐私以及每个人喜好的运行环境，因此“文件所有者”的角色是相当重要的。

通过文件的属性可以设定适当的权限，这样就只有具有适当权限的用户才能查看与修改文件。

37.2 group

在 Linux 下可以通过文件权限设定来限制非同一用户组的用户浏览文件的权限，而且还可以只能让同一用户组中的用户可以修改文件。

在 Linux 系统中，每个用户账号都可以有多个用户组的支持，通过设定用户组共享可以让用户之间共享文件。

37.3 other

Linux 中的任何一个文件都具有“User、Group 及 Others”三个权限，其中“用户身份”与该用户所支持的“用户组”概念很重要，它们可以使多任务 Linux 环境变的更容易管理。

37.4 /etc/passwd

默认的情况下，Linux 系统中的所有用户的相关信息都记录在/etc/passwd 文件内。

```
[theqiong@localhost ~]$ ll /etc/passwd
-rw-r--r--. 1 root root 1899 Nov 24 11:44 /etc/passwd
```

37.5 /etc/shadow

用户的密码则是记录在/etc/shadow 文件中。

```
[theqiong@localhost ~]$ ll /etc/shadow
----------. 1 root root 1075 Nov 24 11:44 /etc/shadow
```

37.6 /etc/group

Linux 操作系统中的所有用户组名称都记录在/etc/group 内，因此这三个文件集中了 Linux 系统中账号、密码、用户组信息。

```
[theqiong@localhost ~]$ ll /etc/group
-rw-r--r--. 1 root root 830 Nov 24 11:44 /etc/group
```

Part V

Administration

Chapter 38

Introduction

38.1 Superuser

In computing, the superuser[?] is a special user account used for system administration. Depending on the operating system, the actual name of this account might be: root, administrator, admin or supervisor. In some cases the actual name is not significant, rather an authorization flag in the user's profile determines if administrative functions can be performed.

In operating systems which have the concept of a superuser, it is generally recommended that most application work be done using an ordinary account which does not have the ability to make system-wide changes.

Most configuration, testing and maintenance of networked systems has the potential to adversely affect multiple systems. In an effort to prevent inexperienced and disruptive individuals from causing problems, operating systems network utilities often require superuser authority. For example stress testing, if done at inappropriate times or without clear understanding of the effects, can deny users access to portions or all of the services of multiple computers. This is more of a problem than the original implementers of some utilities envisioned since it is now common for novices to build systems they own and have the ability to use the superuser account.

38.1.1 UNIX and UNIX-like

In Unix-like computer operating systems, root is the conventional name of the user who has all rights or permissions (to all files and programs) in all modes (single- or multi-user). Alternative names include baron in BeOS and avatar on some Unix variants.[1] BSD often provides a toor ("root" backwards) account in addition to a root account.[2] Regardless of the name, the superuser always has user ID 0. The root user can do many things an ordinary user cannot, such as changing the

ownership of files and binding to network ports numbered below 1024. The name ”root” may have originated because root is the only user account with permission to modify the root directory of a Unix system and this directory was originally considered to be root's home directory.

The first process bootstrapped in a Unix-like system, usually called init, runs with root privileges. It spawns all other processes directly or indirectly, which inherit their parents' privileges. Only a process running as root is allowed to change its user ID to that of another user; once it's done so, there is no way back. Doing so is sometimes called dropping root privileges and is often done as a security measure to limit the damage from possible contamination of the process. Another case is login and other programs that ask users for credentials and in case of successful authentication allow them to run programs with privileges of their accounts.

It is never good practice for anyone (including system administrators) to use root as their normal user account, since simple typographical errors in entering commands can cause major damage to the system. It is advisable to create a normal user account instead and then use the su command to switch when necessary. The sudo utility is preferred as it only executes a single command as root, then returns to the normal user.

Some operating systems, such as OS X and some Linux distributions, allow administrator accounts which provide greater access while shielding the user from most of the pitfalls of full root access. In some cases the root account is disabled by default, and must be specifically enabled. In mobile platform-oriented Unixes such as Apple iOS and Android, the device's security systems must be cracked in order to obtain superuser access. In a few systems, such as Plan 9, there is no superuser at all.

38.1.2 Windows NT

In Windows NT and later systems derived from it (such as Windows 2000, Windows XP, Windows Server 2003, and Windows Vista/7/8), there must be at least one administrator account (Windows XP and earlier) or one able to elevate privileges to superuser (Windows Vista/7/8 via User Account Control).[4] In Windows XP and earlier systems, there is a built-in administrator account that remains hidden when a user administrator-equivalent account exists.[5] This built-in administrator account is created with a blank password.[5] This poses security risks, so the built-in administrator account is disabled by default in Windows Vista and later systems due to the introduction of User Account Control (UAC).

A Windows administrator account is not an exact analogue of the Unix root account - some privileges are assigned to the ”Local System account”. The purpose of the administrator account is to allow making system-wide changes to the computer (with the exception of privileges limited to Local System).

The built-in administrator account and a user administrator account have the same level of privileges. The default user account created in Windows systems is an administrator account. Unlike OS X, Linux, and Windows Vista/7/8 administrator accounts, administrator accounts in Windows systems without UAC do not insulate the system from most of the pitfalls of full root access. One of these pitfalls includes decreased resilience to malware infections. In Microsoft Windows 2000, Windows XP Professional, and Windows Server 2003, administrator accounts can be insulated from more of the these pitfalls by changing the account from the administrator group to the power user group in the user account properties[6] but this solution is not as effective as using newer Windows systems with UAC.

In Windows Vista/7/8 administrator accounts, a prompt will appear to authenticate running a process with elevated privileges. No user credentials are required to authenticate the UAC prompt in administrator accounts but authenticating the UAC prompt requires entering the username and password of an administrator in standard user accounts. In Windows XP (and earlier systems) administrator accounts, authentication is not required to run a process with elevated privileges and this poses another security risk that lead to the development of UAC. Users can set a process to run with elevated privileges from standard accounts by setting the process to ”run as administrator” or using the ”runas” command and authenticating the prompt with credentials (username and password) of an administrator account. Much of the benefit of authenticating from a standard account is negated if the administrator account's credentials being used has a blank password (as in the built-in administrator account in Windows XP and earlier systems).

In Windows Vista/7/8, the root user is TrustedInstaller. In Windows NT, 2000, and XP, the root user is System.

38.1.3 Other

In Novell NetWare, the superuser was called “upervisor”, later “admin”. In OpenVMS, “SYSTEM” is the superuser account for the operating system.

Many older operating systems on computers intended for personal and home use, including MS-DOS and Windows 95, do not have the concept of multiple accounts and thus have no separate administrative account; anyone using the system has full privileges. The lack of this separation in these operating systems has been cited as one major source of their insecurity.

超级用户（Superuser）在计算机操作系统领域中指一种用于进行系统管理的特殊用户，其在系统中的实际名称也因系统而异，如 root、administrator 与 supervisor。为了使病毒、恶意软件与普通的用户错误不对整个系统产生不利的影响，在系统里日常任务都是由无法进行全系统变更的普通用户账户所完成。

在 Unix 与类 Unix 系统中，root 是在所有模式（单/多用户）下对拥有对所有文件与程序拥有一切权限的用户（也即超级用户）的约定俗成的通名，但也有例外，如在 BeOS 中超级用户的实名是 baron，在其他一些 Unix 派生版里则以 avatar 作为超级用户的实名 [1]，而 BSD 中一般也提供“toor”账户（也即“root”的反写）作为 root 账户的副本 [2]，但无论实名为何，超级用户的用户 ID（UID）一般都为 0。root 用户可以进行许多普通用户无法做的操作，如更改文件所有者或绑定编号于 1024 之下的网络端口。之所以将“root”设定为超级用户之名，可能是因为 root 是唯一拥有修改 UNIX 系统根目录（root directory）的权限的用户，而根目录最初就被认为是 root 的家目录一般的存在。

在类 Unix 系统引导过程中引导的第一个程序（常被称为 init）就是以 root 权限运行的，其他所有进程都由其直接或间接派生而出，并且这些进程都继承了各自的父进程的权限。只有以 root 权限运行的进程才能将自己的 UID 修改为其他用户对应的 UID，且对应 UID 在修改完成之后无法改回，这种行为有时也被称为丢弃 root 权限，其目的主要是为了安全考虑-(在进程出错等情况下）降低进程污染所造成的损失。另一种情况是，用户登录后，有些程序会向用户请求认证提升权限，当认证成功后用户就能以其账户所对应的权限来执行程序。

对任何人（也包括系统管理员自己）来说，将 root 当作一般用户账户使用都绝不是一个好习惯，因为即使是输入命令时的微小错误都可能对系统造成严重破坏，因而相较之下较为明智的做法是创建一个普通用户账户用作日常使用，需要 root 权限时再用 su 切换到 root 用户。sudo 工具也是个暂时性获取 root 权限的替代方法。

在 Mac OS X 与一些 Linux 发行版中则允许管理员账户（这与 root 这样的全权账户有别）拥有更多的权限，同时也屏蔽掉大部分容易（因误操作）造成损害的 root 权限。某些情况下，root 账户是被默认禁用的，需要时必须另外启用。另外在极少数系统（如 Plan 9）中，系统中根本没有超级用户。

某些软件缺陷能使用户获得 root 权限（即提升权限来以 root 权限执行用户提供的代码），这会造成严重的计算机安全问题，相对应的修复这些软件则是系统安全维护的重要组成部分。让某个正以超级用户权限运行的程序的缓冲区溢出（某些情况下亦称缓冲区攻击）是一种常见的（非法）获得 root 权限的方式，在现代的操作系统中则一般采取将关键程序(如网络服务器程序）运行于一个特别的限权用户之下的方式来预防这种情况的发生。

在 Windows NT 及其派生的后继系统（如 Windows 2000,Windows XP，Windows Server 2003 及 Windows Vista/7），系统里要么至少有一个管理员账户（Windows XP 及更早的系统），要么可以使用用户账户控制（User Account Control，简称 UAC）机制提升到超级用户权限（Windows Vista/7）。在 Windows XP 及更早的系统中有内建一个初始密码为空的管理账户(实名为“Administrator”），此账户在存在其他有系统管理权限的用户账户时是默认隐藏的

[6]；显而易见的，这种做法会带来安全问题，所以在 Windows Vista 与其后继系统中默认禁止了内建的 Administrator 账户，并引入 UAC 机制取而代之。

Windows 的 Administrator 与 Unix 的 root 账户不尽相同-Windows 将部分权限分配给了“本地系统账户”，Administrator 账户的目的只是为了允许在计算机上进行全系统范围（包括本地系统账户权限所不能及之范围）的更改。

Windows 内建的 Administrator 账户与一般的管理员用户享有同等的权限，Windows 系统中默认创建的用户账户也拥有管理权限，而与 Mac OS X，Linux 与 Windows Vista/7 的管理员账户不同的是，Windows 中无 UAC 限制的 Administrator 账户无法将系统与超级用户权限容易造成的损害（如降低对恶意软件的抗力）相隔离。在 Windows 2000,Windows XP 专业版与 Windows Server 2003 中，管理员可用在账户属性中将账户所属组由管理员组更改为权力用户（Power User）组的方式来解决这一问题 [9]，但这一解决方法不如使用带有 UAC 机制的新版 Windows 系统有效。

对于 Windows XP（及早期的系统）的管理员账户来说，提升权限执行程序并不需认证；这种做法有相当的安全隐患，解决这一问题也是开发 UAC 的目的之一。与之相对的，在 Windows Vista/7 中，管理员账户提升权限运行进程的时候会有确认提示，但不需要进行用户资格认证，而普通用户账户则需在提示框内输入任一管理员账户的用户名和密码才能通过认证；具体来说，用户可以以将进程设置为“以管理员权限运行”或使用“runas”命令并用任一管理员账户的用户名和密码进行资格认证的方式在普通用户下提升权限运行进程，但如若提权运行认证时所使用的管理员账号对应的密码为空（就像 XP 及早期系统内建的 Administrator 账户一样）时，认证机制的意义就大减了。

在 Novell NetWare 中，超级用户的实名是“Supervisor”，后来又改为“admin”。另外，许多早期的针对个人/家庭应用而设计的操作系统（如 MS-DOS 与 Windows 9x）上并没有多用户的概念，因而也没有单独的管理账户，所有使用系统的人都有所有管理特权，这种没有分隔权限的情况也被认为是这些系统的重要安全隐患之一。

38.2 su

The su command, also referred to as substitute user, super user, or switch user, allows a computer operator to change the current user account associated with the running virtual console.

By default, and without any other command line argument, this will elevate the current user to the superuser of the local system.

When run from the command line, su asks for the target user's password, and if authenticated, grants the operator access to that account and the files and directories that account is permitted to access.

```
theqiong@localhost:~$ su
Password:
root@localhost:/home/john# exit
logout
john@localhost:~$
```

Additionally, one can switch to another user who is not the superuser; e.g. su jane.

```
theqiong@localhost:~$ su jane
Password:
theqiong@localhost:/home/john$ exit
logout
john@localhost:~$
```

It should generally be used with a hyphen by administrators (su -, which is identical to su - root), which can be used to start a login shell. This way users can assume the user environment of the target user:

```
john@localhost:~$ su - jane
Password:
jane@localhost:~$
```

A related command called sudo executes a command as another user but observes a set of constraints about which users can execute which commands as which other users (generally in a configuration file named /etc/sudoers, best editable by the command visudo). Unlike su, sudo authenticates users against their own password rather than that of the target user (to allow the delegation of specific commands to specific users on specific hosts without sharing passwords among them and while mitigating the risk of any unattended terminals).

Some Unix-like systems have a wheel group of users, and only allow these users to su to root. This may or may not mitigate these security concerns, since an intruder might first simply break into one of those accounts. GNU su, however, does not support a wheel group for philosophical reasons. Richard Stallman argues that because a wheel group would prevent users from utilizing root passwords leaked to them, the group would allow existing admins to ride roughshod over ordinary users.

su 命令也被称为“替代用户”、“超级用户”或“切换用户”，是可以让计算机操作者在虚拟控制台切换当前用户帐户的命令。没有其他命令行参数时，默认将会将当前用户提权至本地超级用户。

在命令行中运行时，su 会要求目标用户的密码。如果验证通过，操作者将会授予该帐户的权限，并且允许访问该帐户可以访问的文件和目录。

```
john@localhost:~$ su
密码:
root@localhost:/home/john# exit
登出
john@localhost:~$
\begin{Verbatim}
```

此外，还可以切换到另一个不是超级用户的帐户，例如 su jane。

```
\begin{Verbatim}
john@localhost:~$ su jane
密码:
jane@localhost:/home/john$ exit
登出
john@localhost:~$
\begin{Verbatim}
```

一般来说，管理员应该使用一个连字号（su -，等同于 su - root），来启动登录 shell。这样，用户可以

```
\begin{Verbatim}
john@localhost:~$ su - jane
密码:
jane@localhost:~$
\begin{Verbatim}
```

相关的命令 sudo 也可以允许以另一个用户的身份执行命令，但遵守一组的限制，以决定哪些用户可以以

一些类 Unix 系统有 wheel 组，且只允许组内用户 su 到 root。很难说这是否会降低安全风险，因为入

```
\section{sudo}
```

```
sudo\cite{sudo} is a program for Unix-like computer operating systems that allows users to run

Sudo (su "do") allows a system administrator to delegate authority to give certain users (or gr

\begin{compactitem}
\item The ability to restrict what commands a user may run on a per-host basis.
\item Sudo does copious logging of each command, providing a clear audit trail of who did what.
\item Sudo uses timestamp files to implement a ``ticketing" system. When a user invokes sudo an
\item Sudo's configuration file, the sudoers file, is setup in such a way that the same sudoers
\end{compactitem}



The program was originally written by Robert Coggeshall and Cliff Spencer ``around 1980" at the

Unlike the \texttt{su} command, users typically supply their own password to sudo rather than t

The /etc/sudoers file allows listed users access to execute a huge amount of configurability wh

\begin{compactitem}
\item Enabling root commands only from the invoking terminal;
\item Not requiring a password for certain commands;
\item Requiring a password per user or group;
\item Requiring re-entry of a password every time or never requiring a password at all for a pa
\end{compactitem}

It can also be configured to permit passing arguments or multiple commands, and even supports c

By default the user's password can be retained through a grace period (15 minutes per pseudo te

sudo is able to log each command run. Where a user attempts to invoke sudo without being listed

In some cases sudo has completely supplanted the superuser login for administrative tasks, most
```

visudo is a command-line utility that allows editing of the /etc/sudoers file in a safe f

The runas command provides similar functionality in Microsoft Windows, but it cannot pass

There exist several frontends to sudo for use in a GUI environment, notably kdesudo, and

\clearpage

Sudo（substitute user [或 superuser] do）是一种程序，用于类 Unix 操作系统如 BSD、Mac OS X

在 sudo 于 1980 年前后被写出之前，一般用户管理系统的方式是利用 su 切换为超级用户，只是使用 s

sudo 使一般用户不需要知道超级用户的密码即可获得权限。首先超级用户将普通用户的名字、可以执行的

在一般用户需要取得特殊权限时，其可在命令前加上“sudo”，此时 sudo 将会询问该用户自己的密码（以

由于不需要超级用户的密码，部分 Unix 系统甚至利用 sudo 使一般用户取代超级用户作为管理账号，例

\begin{Verbatim}
语法：
sudo [-bhHpV][-s][-u < 用户>][指令]
或
sudo [-klv]
参数
-b 在后台执行指令。
-h 显示帮助。
-H 将 HOME 环境变量设为新身份的 HOME 环境变量。
-k 结束密码的有效期限，也就是下次再执行 sudo 时便需要输入密码。
-l 列出目前用户可执行与无法执行的指令。
-p 改变询问密码的提示符号。
-s 执行指定的 shell。
-S 从标准输入流替代终端来获取密码
-u < 用户> 以指定的用户作为新的身份。若不加上此参数，则预设以 root 作为新的身份。
-v 延长密码有效期限 5 分钟。
-V 显示版本信息。

38.3 root

Android rooting is the process of allowing users of smartphones, tablets, and other devices running the Android mobile operating system to attain privileged control (known as "root access") within Android's sub-system.

Rooting lets all user-installed applications run privileged commands typically unavailable to the devices in the stock configuration. Rooting is required for more advanced and potentially dangerous operations including modifying or deleting system files, removing carrier- or manufacturer-installed applications, and low-level access to the hardware itself (rebooting, controlling status lights, or recalibrating touch inputs.)

The process of rooting varies widely by device, but usually includes exploiting one or more security bugs in the firmware of (i.e., in the version of the Android OS installed on) the device. Once an exploit is discovered, a custom recovery image can be flashed which will skip the digital signature check of firmware updates. Then a modified firmware update can be installed which typically includes the utilities needed to run apps as root.

A typical rooting installation also installs the Superuser application, which supervises applications that are granted root or superuser rights. For example, the `su` binary can be copied to a location in the current process' PATH (e.g., `/system/xbin/`) and granted executable permissions with the `chmod` command. A supervisor application, like SuperUser or SuperSU, can then regulate and log elevated permission requests from other applications.

The process of rooting a device may be simple or complex, and it even may depend upon serendipity. For example, shortly after the release of the HTC Dream (HTC G1), it was discovered that anything typed using the keyboard was being interpreted as a command in a privileged (root) shell. Although Google quickly released a patch to fix this, a signed image of the old firmware leaked, which gave users the ability to downgrade and use the original exploit to gain root access. By contrast, the Google-branded Android phones, the Nexus One, Nexus S, Galaxy Nexus, Nexus 4 and Nexus 5, as well as their tablet counterparts, the Nexus 7 and Nexus 10, can be boot-loader unlocked by simply connecting the device to a computer while in boot-loader mode and running the Fastboot program with the command fastboot oem unlock. After accepting a warning, the boot-loader is unlocked, so a new system image can be written directly to flash without the need for an exploit.

A secondary operation, unlocking the device's bootloader verification, is required to remove or replace the installed operating system. In contrast to iOS jailbreaking, rooting is not needed to run applications distributed outside of the Google Play Store, sometimes called sideloading. The Android OS supports this feature natively in two ways: through the "Unknown sources" option in the Settings menu and through the Android Debug Bridge. However some carriers, like AT&T, prevent the installation of applications not on the Store in firmware, although several devices (including the

Samsung Infuse 4G) are not subject to this rule, and AT&T has since lifted the restriction on several older devices.

As of 2012 the Amazon Kindle Fire defaults to the Amazon Appstore instead of Google Play, though like most other Android devices, Kindle Fire allows sideloading of applications from unknown sources, and the ”easy installer” application on the Amazon Appstore makes this easy. Other vendors of Android devices may look to other sources in the future. Access to alternate apps may require rooting but rooting is not always necessary. Rooting an Android phone lets the owner modify or delete the system files, which in turn lets them perform various tweaks and use apps that require root access.

Rooting is often performed with the goal of overcoming limitations that carriers and hardware manufacturers put on some devices, resulting in the ability to alter or replace system applications and settings, run specialized apps that require administrator-level permissions, or perform other operations that are otherwise inaccessible to a normal Android user. On Android, rooting can also facilitate the complete removal and replacement of the device's operating system, usually with a more recent release of its current operating system.

As Android derives from the Linux kernel, rooting an Android device gives similar access administrative permissions as on Linux or any other Unix-like operating system such as FreeBSD or OS X.

Root access is sometimes compared to jailbreaking devices running the Apple iOS operating system. However, these are different concepts. Jailbreaking describes the bypass of several types of Apple prohibitions for the end user: modifying the operating system (enforced by a “locked bootloader”), installing non-officially approved apps via sideloading, and granting the user elevated administration-level privileges. Only a minority of Android devices lock their bootloaders—and many vendors such as HTC, Sony, Asus and Google explicitly provide the ability to unlock devices, and even replace the operating system entirely. Similarly, the ability to sideload apps is typically permissible on Android devices without root permissions. Thus, it is primarily the third aspect of iOS jailbreaking relating to giving users superuser administrative privileges that most directly correlates to Android rooting.

In the past, many manufacturers have tried to make “unrootable” phones with harsher protections (like the Droid X), but they're usually still rootable in some way, shape, or form. There may be no root exploit available for new or recently updated phones, but one is usually available within a few months.

In 2011, Motorola, LG Electronics and HTC added security features to their devices at the hardware level in an attempt to prevent users from rooting retail Android devices.[citation needed] For instance, the Motorola Droid X has a security boot-loader that puts the phone in ”recovery mode” if a user loads unsigned firmware onto the device, and the Samsung Galaxy S II displays a yellow triangle indicator if the device firmware has been modified.

root 通常是针对 Android 系统的手机而言，它使得用户可以获取 Android 操作系统的超级用户权限。root 通常用于帮助用户越过手机制造商的限制，使得用户可以卸载手机制造商预装在手机中某些应用，以及运行一些需要超级用户权限的应用程序。

原始出厂的手机并未开放 root 权限，获取 root 的方法都是不受官方支持的，目前获取 root 的方法都是利用系统漏洞实现的。

不同手机厂商可能存在的漏洞不同，也就导致了不同手机 root 的原理可能不同。为了让用户可以控制 root 权限的使用，防止其被未经授权的应用所调用，通常还有一个 Android 应用程序来管理 su 程序的行为。不过，不管采用什么原理实现 root，最终都需要将 su 可执行文件和对应的 Android 管理应用复制到 Android 系统的`/system` 分区下 (例如`/system/xbin/su`) 并用 chmod 命令为其设置可执行权限和 setuid 权限。

目前最广泛利用的系统漏洞是 zergRush，该漏洞适用于 Android 2.2-2.3.6 的系统，其它的漏洞还有 Gingerbreak, psneuter 等。其中，zergRush 漏洞必须在 adb shell 下运行，而 adb shell 只能将手机用 USB 数据线与 PC 连接之后才能在 PC 上打开，因此目前常用的 root 工具都是 PC 客户端程序，亦有部分工具能直接在 Android 设备上运行。

在 Android 设备上直接运行的 root 工具中，部分以 App 的形式在各类应用商店（非 Google 官方）上发布，供用户下载使用。这些工具通常通常采用傻瓜化操作，即用户只需按下一个按钮就可以等应用来获取 root 权限。利用这些应用获取 root 权限之后，应用本身就会成为 root 权限的授权者，其他应用使用 root 权限时都需要通过此授权者的允许。有些应用（如百度一键 root）会在通知栏推送广告，还有一些应用在获取 root 权限之后将自己变成系统应用，就像厂商预装的那样。

38.4 sa

A system administrator[?], or sysadmin, is a person who is responsible for the upkeep, configuration, and reliable operation of computer systems; especially multi-user computers, such as servers.

The system administrator seeks to ensure that the uptime, performance, resources, and security of the computers he or she manages meet the needs of the users, without exceeding the budget.

To meet these needs, a system administrator may acquire, install, or upgrade computer components and software; automate routine tasks; write computer programs; troubleshoot; train and/or supervise staff; and provide technical support.

The duties of a system administrator are wide-ranging, and vary widely from one organization to another. Sysadmins are usually charged with installing, supporting, and maintaining servers or other computer systems, and planning for and responding to service outages and other problems. Other duties may include scripting or light programming, project management for systems-related projects.

Many organizations staff other jobs related to system administration. In a larger company, these may all be separate positions within a computer support or Information Services (IS) department. In a smaller group they may be shared by a few sysadmins, or even a single person.

- A database administrator (DBA) maintains a database system, and is responsible for the integrity of the data and the efficiency and performance of the system.
- A network administrator maintains network infrastructure such as switches and routers, and diagnoses problems with these or with the behavior of network-attached computers.
- A security administrator is a specialist in computer and network security, including the administration of security devices such as firewalls, as well as consulting on general security measures.
- A web administrator maintains web server services (such as Apache or IIS) that allow for internal or external access to web sites. Tasks include managing multiple sites, administering security, and configuring necessary components and software. Responsibilities may also include software change management.
- A computer operator performs routine maintenance and upkeep, such as changing backup tapes or replacing failed drives in a RAID. Such tasks usually require physical presence in the room with the computer; and while less skilled than sysadmin tasks require a similar level of trust, since the operator has access to possibly sensitive data.
- A postmaster administers a mail server.
- A Storage (SAN) Administrator. Create, Provision, Add or Remove Storage to/from Computer systems. Storage can be attached local to the system or from a Storage Area Network (SAN) or Network Attached Storage (NAS). Create File Systems from newly added storage.

In some organizations, a person may begin as a member of technical support staff or a computer

operator, then gain experience on the job to be promoted to a sysadmin position.

Most important skills to a system administrator is problem solving. This can some times lead into all sorts of constraints and stress. When a workstation or server goes down, the sysadmin is called to solve the problem. They should be able to quickly and correctly diagnose the problem. They must figure out what is wrong and how best it can be fixed in a short time.

The subject matter of system administration includes computer systems and the ways people use them in an organization. This entails a knowledge of operating systems and applications, as well as hardware and software troubleshooting, but also knowledge of the purposes for which people in the organization use the computers.

Perhaps the most important skill for a system administrator is problem solving—frequently under various sorts of constraints and stress. The sysadmin is on call when a computer system goes down or malfunctions, and must be able to quickly and correctly diagnose what is wrong and how best to fix it. They may also need to have team work and communication skills; as well as being able to install and configure hardware and software.

System administrators are not software engineers or developers. It is not usually within their duties to design or write new application software. However, sysadmins must understand the behavior of software in order to deploy it and to troubleshoot problems, and generally know several programming languages used for scripting or automation of routine tasks.

Particularly when dealing with Internet-facing or business-critical systems, a sysadmin must have a strong grasp of computer security. This includes not merely deploying software patches, but also preventing break-ins and other security problems with preventive measures. In some organizations, computer security administration is a separate role responsible for overall security and the upkeep of firewalls and intrusion detection systems, but all sysadmins are generally responsible for the security of computer systems.

A system administrator's responsibilities might include:

- Analyzing system logs and identifying potential issues with computer systems.
- Introducing and integrating new technologies into existing data center environments.
- Performing routine audits of systems and software.
- Performing backups.
- Applying operating system updates, patches, and configuration changes.
- Installing and configuring new hardware and software.
- Adding, removing, or updating user account information, resetting passwords,etc.
- Answering technical queries and assisting users.
- Responsibility for security.
- Responsibility for documenting the configuration of the system.
- Troubleshooting any reported problems.

- System performance tuning.
- Ensuring that the network infrastructure is up and running.
- Configure, Add, Delete File Systems. Knowledge of Volume management tools like Veritas (now Symantec), Solaris ZFS, LVM.
- The system administrator is responsible for following things:
- User administration (setup and maintaining account)
- Maintaining system
- Verify that peripherals are working properly
- Quickly arrange repair for hardware in occasion of hardware failure
- Monitor system performance
- Create file systems
- Install software
- Create a 'backup'and recover policy
- Monitor network communication
- Update system as soon as new version of OS and application software comes out
- Implement the policies for the use of the computer system and network
- Setup security policies for users. A sysadmin must have a strong grasp of computer security (e.g.
- firewalls and intrusion detection systems)
- Documentation in form of internal wiki
- Password and identity management

In larger organizations, some of the tasks above may be divided among different system administrators or members of different organizational groups. For example, a dedicated individual(s) may apply all system upgrades, a Quality Assurance (QA) team may perform testing and validation, and one or more technical writers may be responsible for all technical documentation written for a company. System administrators, in larger organizations, tend not to be systems architects, system engineers, or system designers.

In smaller organizations, the system administrator might also act as technical support, Database Administrator, Network Administrator, Storage (SAN) Administrator or application analyst.

Unlike many other professions, there is no single path to becoming a system administrator. Many system administrators have a degree in a related field: computer science, information technology, computer engineering, information systems, or even a trade school program. On top of this, nowadays some companies require an IT certification. Other schools have offshoots of their Computer Science program specifically for system administration.

Some schools have started offering undergraduate degrees in System Administration. The first, Rochester Institute of Technology started in 1992. Others such as Rensselaer Polytechnic Institute,

University of New Hampshire,[2] Marist College, and Drexel University have more recently offered degrees in Information Technology. Symbiosis Institute of Computer Studies and Research (SICSR) in Pune, India offers Masters degree in Computers Applications with a specialization in System Administration. The University of South Carolina[1] offers an Integrated Information Technology B.S. degree specializing in Microsoft product support.

As of 2011, only five U.S. universities, Rochester Institute of Technology,[3] Tufts,[4] Michigan Tech, and Florida State University [5] have graduate programs in system administration.[citation needed] In Norway, there is a special English-taught MSc program organized by Oslo University College [6] in cooperation with Oslo University, named "Masters programme in Network and System Administration." There is also a "BSc in Network and System Administration" [7] offered by Gjøvik University College. University of Amsterdam (UvA) offers a similar program in cooperation with Hogeschool van Amsterdam (HvA) named "Master System and Network Engineering". In Israel, the IDF's ntmm course in considered a prominent way to train System administrators.[8] However, many other schools offer related graduate degrees in fields such as network systems and computer security. One of the primary difficulties with teaching system administration as a formal university discipline, is that the industry and technology changes much faster than the typical textbook and coursework certification process. By the time a new textbook has spent years working through approvals and committees, the specific technology for which it is written may have changed significantly or become obsolete.

In addition, because of the practical nature of system administration and the easy availability of open-source server software, many system administrators enter the field self-taught. Some learning institutions are reluctant to, what is in effect, teach hacking to undergraduate level students.

Generally, a prospective will be required to have some experience with the computer system he or she is expected to manage. In some cases, candidates are expected to possess industry certifications such as the Microsoft MCSA, MCSE, MCITP, Red Hat RHCE, Novell CNA, CNE, Cisco CCNA or CompTIA's A+ or Network+, Sun Certified SCNA, Linux Professional Institute among others.

Sometimes, almost exclusively in smaller sites, the role of system administrator may be given to a skilled user in addition to or in replacement of his or her duties. For instance, it is not unusual for a mathematics or computing teacher to serve as the system administrator of a secondary school.

系统管理员（System Administrator，简称 SA）是负责管理计算机系统的人。其具体的含义因环境而异。

拥有庞大复杂的计算机系统的组织，通常按照专长分派计算机员工，其中系统管理员负责维护现有的计算机系统，因此其工作不同于：

- 系统设计师（Systems design，SD）
- 系统分析师（Systems analyst，SA）
- 数据库管理员（Database administrator，DBA）

- 存储管理员 (Storage Administrator)
- 程序员（Programmer，PG）
- 技术支持（Technical support），或信息技术（Information technology，IT）
- 网络管理员（Network administrator）
- Security administrator
- 网站管理员（Web administrator，Webmaster）

Chapter 39

User

Chapter 40

User Group

Chapter 41

login

Chapter 42

root

Chapter 43

Permission

Chapter 44

SELinux

Part VI

Shell

Chapter 45

Introduction

45.1 Overview

计算机操作系统的内核负责硬件和任务调度等，而且内核是需要保护的，因此一般用户只能通过 Shell 来与内核通信。

Shell[1] 起源于 Multics 计划，具体是指“提供用户使用接口”的软件，通常指的是命令行界面的解析器。

一般来说，Shell 是指操作系统中提供访问内核所提供的服务的程序（或接口），也用于泛指所有为用户提供操作界面的程序，也就是程序和用户交互的层面。例如，IPython 就是基于 Python 的交互式解释器，并且提供了比原生的 Python Shell 更为强大的编辑和交互功能。

只要能够操作应用程序的接口都可以被称为 shell，因此狭义的 shell 可以指命令行程序(包括 bash)，广义的 shell 则可以包含提供图形用户界面的程序等，而且图形用户界面的程序也可以调用内核功能。

Shell 需要调用其他程序来响应用户的操作，而且与 Shell 相对的就是内核（Core），内核不提供和用户的交互功能，全部通过 Shell 来实现。

通常情况下，Shell 分为命令行（CLI）与图形界面（GUI）两类，其中命令行 Shell 提供一个命令行界面（CLI），而图形 Shell 层提供一个图形用户界面（GUI）。

当用户在终端（tty）登录后，Linux 就会根据/etc/passwd 文件的设置给用户分配一个 shell（默认为 bash），用户可以使用 startx 命令来启动 X 环境进入图形用户环境。

Shell 也可以用来指代应用软件或任何在特定组件外围的软件，例如浏览器或电子邮件软件是 HTML 排版引擎的 Shell。

普通意义上的 Shell 就是可以接受用户输入命令的程序，Shell 隐藏了操作系统低层的细节，因此图形用户界面 GNOME 和 KDE 被叫做“虚拟 shell”或“图形 shell”。

图形用户界面（GUI shell）通常会构建在视窗系统上，例如 X Window System 有独立的

X 窗口管理器以及依靠窗口管理器的完整桌面环境。

45.2 CLI

命令行界面（command-line interface shell）包括 sh、bash、ksh、zsh、csh 和 PowerShell 等。其中，UNIX 上的第一个 Unix shell 是 Ken Thompson 以 Multics 上的 shell 为模板来为 UNIX 所开发的 sh。

用户通过键盘输入指令，计算机接收到指令后予以执行，并输出结果（Read-Eval-Print Loop）。

Microsoft Windows 操作系统也提供了类似的功能，分别是 Windows 95/98 下的 command.com 和 Windows NT 系统下的 cmd.exe。

Windows PowerShell 是 Windows 环境的壳程序（shell）及脚本语言技术，并且通过内置的脚本语言以及辅助脚本程序的工具提供了丰富的控制与自动化的系统管理能力。

Windows PowerShell 以.NET Framework 技术为基础，并且与现有的 Windows Shell 保持向后兼容，因此它的脚本程序不仅能访问.NET CLR，也能使用现有的 COM 技术。另外，Windows PowerShell 还包含了多种系统管理工具、简易且一致的语法来提升工作效率（例如登录数据库和 WMI 等）。

Unix 操作系统下的 shell 既是用户交互的界面，也是控制系统的脚本语言。具体而言，shell 仍然是控制操作系统启动、X Window 启动和很多其他实用工具的脚本解释程序。

Shell Builtin Command 是指包含在 Shell 层代码中一起编译而属于 Shell 程序本身的功能或命令，所有的指令调用功能都直接在 Shell 程序中运行，而非由 Shell 程序去调用外部程序。

通常情况下，Shell Builtin Command 的运行速度比外部程序快，这些指令与 Shell 程序本身同属一个程序，无需额外的程序加载。

不过，Shell Builtin Command 功能的代码与 Shell 在同一个文件中（或是在 Shell 的源代码中被包含），所以当需要对这些功能进行修改或更新时，也必须修改到 Shell，因此 Shell Builtin Command 通常为简易或是不重要的功能（例如文字输出等）。

Shell 可以实现在前台或后台进行任务控制或调度，从而可以在单一登录的环境中实现多任务操作。

基于某些操作系统的本质特性，通常会实现一些必要的 Shell 内置命令，其中最常见的内置命令‘cd’的作用就是在壳层中移动到指定的工作目录。

每个程序在运行时都是一个进程，则每个工作目录都会被各个进程引用，将 cd 以外部程序的方式调用并加载就不会改变 Shell 当前的工作目录。即使壳层的目录改变了，Shell 所运行的其他程序引用的工作目录仍没有改变。

另外，logout、exit 都是最常见的 Shell Builtin Command，可以注销或中断终端机连接。

使用 help 指令可以显示出所有 bash 壳层内置指令（例如 cd、echo 和 exit 等），而且 help 本身也是一个内置指令。

```
$ help
GNU bash, version 4.2.53(1)-release (x86_64-redhat-linux-gnu)
These shell commands are defined internally.  Type `help' to see this list.
Type `help name' to find out more about the function `name'.
Use `info bash' to find out more about the shell in general.
Use `man -k' or `info' to find out more about commands not in this list.

A star (*) next to a name means that the command is disabled.
```

45.3 Script

使用 shell script 可以以交互的方式来进行主机的维护工作，或者通过 shell 提供的环境变量及变量进行程序设计。

45.4 Commands

45.4.1 Move Commmands

`<C-f>` 向前一个字符
`<C-b>` 向后一个字符
`<alt-f>` 向前一个单词
`<alt-b>` 向后一个单词
`<C-a>` 跳转到命令行首
`<C-e>` 跳转到命令行尾

45.4.2 Copy/Paste Commands

`<C-u>` 剪切光标之前的内容
`<C-k>` 剪切光标之后的内容
`<C-w>` 剪切光标之前的一个单词
`<C-y>` 粘贴终端下最后剪切的字符串，到当前光标位置

如果需要粘贴系统剪切板中的字符串，可以使用 Ctrl + Shift + V，同时适用于 VIM。

45.4.3 Histroy Commands

`<C-p>` 上一个历史命令
`<C-n>` 下一个历史命令
`<C-r>` 搜索历史命令

45.4.4 Virtual Terminal

`Ctrl + Alt + Fn` 切换虚拟终端 Fn (n 为 1~6，代表第几个终端)

系统启动过程中，在完成了系统所有服务的启动后，init 启动终端模拟程序就产生了第一个终端 (当前终端或图形界面) Ctrl + Alt + F1（有的系统使用的是 Ctrl + Alt + F7，例如 openSUSE)。

另外，还有一些终端快捷键如下:

`<C-c>` 终止命令
`<C-d>` 退出 shell 或 注销
`<C-l>` 清除屏幕，同 clear
`<C-z>` 转入后台运行，当前用户退出后会终止，如果不想终止 & 或 screen 虚拟终端可供选择

Chapter 46

vi/vim

46.1 Introduction

不同的 Linux 发行版都有其不同的附加软件，例如 Red Hat Enterprise Linux 与 Fedora 的 ntsysv 与 setup 等，而 SuSE 则有 YAST 管理工具等。

在所有的 Linux 发行版中都会有的一种文本编辑器就是 vi[1]，而且很多软件默认也是使用 vi 作为编辑器的。vi[2]是一种计算机文本编辑器，由美国计算机科学家比尔 · 乔伊（Bill Joy）于 1976 年以 BSD 授权发布[2]。

vi 是一种模式编辑器，使用不同的按钮和键击可以更改不同的“模式”，比如说：

- 在“插入模式”下，输入的文本会直接被插入到文档；
- 当按下“退出键”，“插入模式”就会更改为“命令模式”，并且光标的移动和功能的编辑都由字母来响应，例如：
  - “j”用来移动光标到下一行；
  - “k”用来移动光标到上一行；
  - “x”可以删除当前光标处的字符；
  - “i”可以返回到“插入模式”（也可以使用方向键）。

在“命令模式”下，敲入的键（字母）并不会插入到文档，而且多重文本编辑操作是由一组键（字母）来执行，而不是同时按下 <Alt>、<Ctrl> 和其他特殊键来完成。更多更复杂的编辑操作可以使用多重功能基元的组合，比如说：

- “dw”用来删除一个单词；
- “c2fa”可以更改当前的光标处中“a”之前的文本。

因此，对于熟练的 vi 用户可以有更快的操作，因为双手就可以不必离开键盘。

早期的版本中，vi 并没有指示出当前的模式，用户必须按下“退出键”来确认编辑器返

[1] vim 也是 Linux 下第二强大的编辑器，emacs 是公认的世界第一。

[2] vi 是“Visual”的不正规的缩写，来源于另外一个文本编辑器 ex 的命令 visual。

回“命令模式”(会有声音提示)，后来的 vi 版本可以在“状态条”中（或用图形显示)，而在最新的版本中，用户可以在“终端”中设置并使用除主键盘以外的其他键，例如：PgUp，PgDn，Home，End 和 Del 键。另外，图形化界面的 vi 可以很好的支持鼠标和菜单。

直到 Emacs 的出现（1984 年以后)，vi 几乎是所有“黑客”所使用的标准 UNIX 编辑器。从 2006 年开始，vi 成为了“单一 UNIX 规范”（Single UNIX Specification）的一部分，因此 vi 或 vi 的一种变形版本（vim）一定会在 UNIX 中找到。当救急软盘作为恢复硬盘崩溃的媒介以来，vi 通常被用户选择，因为一张软盘正好存储下 vi，并且几乎所有人都可以很轻松的使用 vi。

vim（Vi IMproved)[4] 是一种升级版，类似 nvi。作为从 vi 发展出来的一种编辑器[3]，vim 的第一个版本由布莱姆 · 米勒在 1991 年发布。最初，vim 的简称是 Vi IMitation，随着功能的不断增加，正式名称改成了 Vi IMproved。

vim 具有代码补全、编译和错误调转等功能，还能够进行 shell script 等编程功能，因此可以将 vim 视为一种程序编辑器。另外 vim 还支持正则表达式的搜索模式、多文件编辑、区块复制等。

vim 会依据文件的扩展名或者是文件内的开头信息，判断该文件的内容而自动的调用该程序的语法判断式，再以颜色来显示程序代码与一般信息。

- 1994 年的 vim 3.0 加入了多视窗编辑模式（分区视窗)。
- 1996 年发布的 vim 4.0 是第一个利用 GUI（图形用户界面）的版本。
- 1998 年 5.0 版本的 vim 加入了 highlight（语法高亮）功能。
- 2001 年的 vim 6.0 版本加入了代码折叠、插件、多国语言支持、垂直分区视窗等功能。
- 2006 年 5 月发布的 vim 7.0 版更加入了拼字检查、上下文相关补全，标签页编辑等新功能。
- 2008 年 8 月发布的 vim 7.2，合并了 Vim 7.1 以来的所有修正补丁，并且加入了脚本的浮点数支持。
- 2010 年 8 月发布的 vim 7.3，加入了“永久撤销”、“Blowfish 算法加密”、“文本隐藏”和“Lua 以及 Python3 的接口”等新功能。

和集成开发环境一样，vim 具有可以配置成在编辑代码源文件之后直接进行编译的功能。编译出错的情况下，可以在另一个窗口中显示出错误。根据错误信息可以直接跳转到正在编辑的源文件出错位置。

在文件比较方面，vim 可以逐行的对文本文件进行比较。例如，vim 可以并排显示两个版本的文件，同时以不同的颜色来表示有差别部分。修改过的、新增的或者是被删除的行会以颜色高亮来强调，没有改变过的部分则会被自动折叠表示。

- 对于已经在 vim 中打开的两个缓冲区，可以使用 “:diffthis” 对这两个缓冲区的内容进行比较，被比较的缓冲区可以是一个尚未存盘的内存中的缓冲区。
- 在比较两个文件的不同之处时，可以用 “:diffget” 和 “:diffput” 命令对每一处不同进行双向的同步，也可以在比较不同时对内容进行其它编辑，然后用 “:diffupdate” 对最新内

容重新进行比较。

- 在浏览两个文件的不同之处时，可以用 “[c” 和 “]c” 两个普通（Normal）模式的命令直接跳转到上一个和下一个不同之处。
- 可以通过 “:diffopt” 等选项更精细地控制哪些区别被认为是真正的不同之处，比如可以设置比较时忽略空白字符数量的不同。

UNIX 下可以用 vimdiff 命令来使用这个功能。

vim 有其脚本语言 vimscript，使用 vimscript 写成的宏可以实现自动执行复杂的操作。使用 “-s” 选项启动 vim，或者直接切换到宏所在目录使用 “:source” 命令都可以执行 Vim 脚本。其中，vim 的配置文件就可以作为 vim 脚本的一个范例，在 UNIX 和 Linux 下 vim 的配置文件名是.vimrc，在 Windows 下 vim 的配置文件一般叫做 _vimrc。这个文件在启动 vim 的时候被自动执行。

- vimscript 可以使用 vim 命令行模式的所有命令，使用 “:normal” 命令则可以使用通常模式中的所有命令。
- vimscript 具有数字和字符串两种数据类型，其中用数字代表布尔类型，0 代表假，之外的数全代表真。vim 7 还提供了列表、关联数组等高级数据结构。
- vimscript 也拥有各种比较运算符和算术运算符。
- vimscript 的控制结构实现了 if 分支和 for/while 循环。
- 用户可以自己定义函数，并且可以使用超过 100 种的预定义函数。
- vimscript 编写成的脚本文件可以在调试模式中进行调试。

不过，vimscript 处理速度并不快，导致 vim 读取大文件的速度很慢（可以通过 LargeFile 脚本优化），而且在处理非常长的行时，vim 速度也会变慢。

46.2 Documents

通常可以在 Unix 系统命令行下输入 “vimtutor” 进入《VIM 教程》，在 Vim 用户手册中更加详细的描述了 Vim 的基础和进阶功能。另外，可以在 Vim 中输入 “:help user-manual” 进入用户手册。

vim 提供了文本形式的大量文档，并且提供了各种各样的功能用于快速找到问题的解决方案。根据 vim 自己的帮助文件语法，关键字会被各种各样醒目的颜色表示出来，可以用快捷键像在浏览器中那样浏览帮助文件。

在 GUI 版的 vim（gvim）中还可以使用鼠标在帮助文件中移动。方便用户寻找问题解决方案的功能还不止这些，其中最主要的是 “:helpgrep” 命令。使用这条命令，用户可以在所有帮助文件中搜索想要察看的内容，用 “:cwindows” 可以在另一个窗口中表示搜索的结果，根据搜索的结果自动在帮助文件内跳转。使用 vim 的帮助功能，还可以在搜索的结果中继续进行搜索。

46.3 Basic Modes

从 vi 演生出来的 vim 具有多种模式，而且 vim 和 vi 都是仅仅通过键盘来在这些模式之中切换。这就使得 vim 可以不用进行菜单或者鼠标操作，并且最小化组合键的操作。具体来说，vim 包括 6 种基本模式和 5 种派生模式。

几乎所有的编辑器都会有插入和执行命令两种模式，并且大多数的编辑器使用了与 Vim 截然不同的方式：命令目录（鼠标或者键盘驱动），组合键（通常通过 control 键和 alt 键组成）或者鼠标输入。vim 和 vi 一样，仅仅通过键盘来在这些模式之中切换，这使得 vim 可以不用进行菜单或者鼠标操作，并且最小化组合键的操作。

46.3.1 Normal

在普通模式中，用户可以执行一般的编辑器命令（比如移动光标，删除文本等），这也是 Vim 启动后的默认模式。这正好和许多新用户期待的操作方式相反（大多数编辑器默认模式为插入模式）。

Vim 强大的编辑能力中很大部分是来自于其普通模式命令。普通模式命令往往需要一个操作符结尾，例如：

- 普通模式命令 “dd” 删除当前行，但是第一个 “d” 的后面可以跟另外的移动命令来代替第二个”d”；
- 用移动到下一行的 “j” 键就可以删除当前行和下一行；
- 可以指定命令重复次数，“2dd”（重复 “dd” 两次）和 “dj” 的效果是一样的。

用户熟悉了文本间移动/跳转的命令和其他的普通模式的编辑命令，并能够灵活组合使用的话，可以比那些没有模式的编辑器更加高效的进行文本编辑。

在普通模式中，有很多方法可以进入插入模式。比较普通的方式是 “a”（append/追加）键或者 “i”（insert/插入）键。

46.3.2 Insert

在 Insert 模式中，大多数按键都会向文本缓冲中插入文本。大多数新用户希望文本编辑器编辑过程中一直保持这个模式。

在插入模式中，可以按 ESC 键回到普通模式。

46.3.3 Visual

可视（Visual）模式与普通模式比较相似，只是移动命令会扩大高亮的文本区域，高亮区域可以是字符、行或者是一块文本。

当执行一个非移动命令时，命令会被执行到这块高亮的区域上，而且 vim 的“文本对象”也能和移动命令一样用在可视模式中。

46.3.4 Select

选择模式和无模式编辑器的行为比较相似（Windows 标准文本控件的方式）。在选择模式中，可以用鼠标或者光标键高亮选择文本，不过输入任何字符的话，vim 会用这个字符替换选择的高亮文本块，并且自动进入插入模式。

46.3.5 Commandline

在命令行模式中可以输入会被解释成并执行的文本，例如执行命令（":" 键）、搜索（"/" 和 "?" 键）或者过滤命令（"!" 键）。在命令执行之后，vim 返回到命令行模式之前的模式，通常是普通模式。

46.3.6 Ex Mode

Ex 模式和命令行模式比较相似，在使用 ":visual" 命令离开 Ex 模式前，可以一次执行多条命令。

46.4 Derivative Modes

46.4.1 Operator-pending

操作符等待模式是指在普通模式中，执行一个操作命令后 vim 等待一个“动作”来完成这个命令。vim 也支持在操作符等待模式中使用“文本对象”作为动作，包括 "aw" 一个单词（a word）、"as" 一个句子（a sentence）、"ap" 一个段落（a paragraph）等。

比如，在普通模式下 "d2as" 删除当前和下一个句子。在可视模式下 "apU" 把当前段落所有字母大写。

46.4.2 Insert Normal

插入普通模式是在插入模式下按下 ctrl-o 键的时候进入。这个时候暂时进入普通模式，执行完一个命令之后，vim 返回插入模式

46.4.3 Insert Visual

插入可视模式是在插入模式下按下 ctrl-o 键并且开始一个可视选择的时候开始。在可视区域选择取消的时候，vim 返回插入模式。

46.4.4 Insert Select

通常，插入选择模式由插入模式下鼠标拖拽或者 shift 方向键来进入。当选择区域取消的时候，vim 返回插入模式。

46.4.5 Replace

替换模式是一个特殊的插入模式，在这个模式中可以做和插入模式一样的操作，但是每个输入的字符都会覆盖文本缓冲中已经存在的字符。在普通模式下按 “R” 键进入。

另外，Evim（Easy Vim）是一个特殊的 GUI 模式用来尽量表现的和 “无模式” 编辑器一样。编辑器自动进入并且停留在插入模式，用户只能通过菜单、鼠标和键盘控制键来对文本进行操作。可以在命令行下输入 “evim” 或者 “vim -y” 进入。在 Windows 下，通常也可以点击桌面上 Evim（Easy Vim）的图标。

Table 46.1: VIM 使用说明

操作	说明
h 或向左箭头键 (←)	光标向左移动一个字符
j 或向下箭头键 (⌧)	光标向下移动一个字符[3]
k 或向上箭头键 (⌧)	光标向上移动一个字符
l 或向右箭头键 (→)	光标向右移动一个字符
[Ctrl] + [f]	屏幕『向下』移动一页，相当于 [Page Down] 按键 (常用)
[Ctrl] + [b]	屏幕『向上』移动一页，相当于 [Page Up] 按键 (常用)
[Ctrl] + [d]	屏幕『向下』移动半页
[Ctrl] + [u]	屏幕『向上』移动半页
+	光标移动到非空格符的下一列
-	光标移动到非空格符的上一列
n<space>	那个 n 表示『数字』，例如 20 。按下数字后再按空格键，光标会向右移动这一行的 n 个字符。例如 20<space> 则光标会向后面移动 20 个字符距离。
0 或功能键 [Home]	这是数字『0 』：移动到这一行的最前面字符处 (常用)
$ 或功能键 [End]	移动到这一行的最后面字符处 (常用)
H	光标移动到这个屏幕的最上方那一行的第一个字符

[3]键盘上的 hjkl 是排列在一起的，因此可以使用这四个按钮来移动光标。如果想要进行多次移动的话，例如向下移动 30 行，可以使用”30j” 或”30⌧” 的组合按键，亦即加上想要进行的次数 (数字) 后，按下动作即可，其他按键同理。

操作	说明
M	光标移动到这个屏幕的中央那一行的第一个字符
L	光标移动到这个屏幕的最下方那一行的第一个字符
G	移动到这个文件的最后一行 (常用)
nG	n 为数字。移动到这个文件的第 n 行。例如 20G 则会移动到这个文件的第 20 行 (可配合:set nu)
gg	移动到这个文件的第一行，相当于 1G 啊！(常用)
n<Enter>	n 为数字。光标向下移动 n 行 (常用)
/word	向光标之下寻找一个名称为 word 的字符串。例如要在文件内搜寻 vbird 这个字符串，就输入 /vbird 即可！(常用)
?word	向光标之上寻找一个字符串名称为 word 的字符串。
n	这个 n 是英文按键。代表『重复前一个搜寻的动作』。举例来说，如果刚刚我们执行 /vbird 去向下搜寻 vbird 这个字符串，则按下 n 后，会向下继续搜寻下一个名称为 vbird 的字符串。如果是执行?vbird 的话，那么按下 n 则会向上继续搜寻名称为 vbird 的字符串！
N	这个 N 是英文按键。与 n 刚好相反，为『反向』进行前一个搜寻动作。例如 /vbird 后，按下 N 则表示『向上』搜寻 vbird。使用 /word 配合 n 及 N 是非常有帮助的！可以让你重复的找到一些你搜寻的关键词
:n1,n2s/word1/word2/g	n1 与 n2 为数字。在第 n1 与 n2 行之间寻找 word1 这个字符串，并将该字符串取代为 word2！举例来说，在 100 到 200 行之间搜寻 vbird 并取代为 VBIRD 则：『:100,200s/vbird/VBIRD/g』。(常用)
:1,$s/word1/word2/g	从第一行到最后一行寻找 word1 字符串，并将该字符串取代为 word2！(常用)
:1,$s/word1/word2/gc	从第一行到最后一行寻找 word1 字符串，并将该字符串取代为 word2！且在取代前显示提示字符给用户确认 (confirm) 是否需要取代！(常用)
x, X	在一行字当中，x 为向后删除一个字符 (相当于 [del] 按键)，X 为向前删除一个字符 (相当于 [backspace] 亦即是退格键) (常用)

操作	说明
nx	n 为数字，连续向后删除 n 个字符。举例来说，我要连续删除 10 个字符，『10x』。
dd	删除游标所在的那一整列 (常用)
ndd	n 为数字。删除光标所在的向下 n 列，例如 20dd 则是删除 20 列 (常用)
d1G	删除光标所在到第一行的所有数据
dG	删除光标所在到最后一行的所有数据

46.5 Visual Block

46.6 Visual Windows

46.7 Configuration

46.8 Hotkey

除了上面简易范例的 i, [Esc], :wq 之外，其实 vim 还有非常多的按键可以使用喔！在介绍之前还是要再次强调，vim 的三种模式只有一般模式可以与编辑、指令列模式切换，编辑模式与指令列模式之间并不能切换的！这点在图 2.1 里面有介绍到，注意去看看喔！底下就来谈谈 vim 软件中会用到的按键功能。

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

[Ctrl] + [d]	屏幕『向下』移动半页
[Ctrl] + [u]	屏幕『向上』移动半页
+	光标移动到非空格符的下一列
-	光标移动到非空格符的上一列
n<space>	那个 n 表示『数字』，例如 20 。按下数字后再按空格键，光标会向右移动这一行的 n 个字符。例如 20<space> 则光标会向后面移动 20 个字符距离。
0 或功能键 [Home]	这是数字『0 』: 移动到这一行的最前面字符处 (常用)
$ 或功能键 [End]	移动到这一行的最后面字符处 (常用)
H	光标移动到这个屏幕的最上方那一行的第一个字符
M	光标移动到这个屏幕的中央那一行的第一个字符
L	光标移动到这个屏幕的最下方那一行的第一个字符
G	移动到这个档案的最后一行 (常用)
nG	n 为数字。移动到这个档案的第 n 行。例如 20G 则会移动到这个档案的第 20 行 (可配合:set nu)
gg	移动到这个档案的第一行，相当于 1G 啊！(常用)
n<Enter>	n 为数字。光标向下移动 n 行 (常用)
搜寻与取代	
/word	向光标之下寻找一个名称为 word 的字符串。例如要在档案内搜寻 vbird 这个字符串，就输入 /vbird 即可！(常用)
?word	向光标之上寻找一个字符串名称为 word 的字符串。
n	这个 n 是英文按键。代表『重复前一个搜寻的动作』。举例来说，如果刚刚我们执行 /vbird 去向下搜寻 vbird 这个字符串，则按下 n 后，会向下继续搜寻下一个名称为 vbird 的字符串。如果是执行?vbird 的话，那么按下 n 则会向上继续搜寻名称为 vbird 的字符串！
N	这个 N 是英文按键。与 n 刚好相反，为『反向』进行前一个搜寻动作。例如 /vbird 后，按下 N 则表示『向上』搜寻 vbird 。
使用 /word 配合 n 及 N 是非常有帮助的！可以让你重复的找到一些你搜寻的关键词！	

:n1,n2s/word1/word2/g	n1 与 n2 为数字。在第 n1 与 n2 行之间寻找 word1 这个字符串，并将该字符串取代为 word2！举例来说，在 100 到 200 行之间搜寻 vbird 并取代为 VBIRD 则：『:100,200s/vbird/VBIRD/g』。(常用)
:1,$s/word1/word2/g	从第一行到最后一行寻找 word1 字符串，并将该字符串取代为 word2！(常用)
:1,$s/word1/word2/gc	从第一行到最后一行寻找 word1 字符串，并将该字符串取代为 word2！且在取代前显示提示字符给用户确认 (confirm) 是否需要取代！(常用)
x, X	在一行字当中，x 为向后删除一个字符 (相当于 [del] 按键)，X 为向前删除一个字符 (相当于 [backspace] 亦即是退格键) (常用)
nx	n 为数字，连续向后删除 n 个字符。举例来说，我要连续删除 10 个字符，『10x』。
dd	删除游标所在的那一整列 (常用)
ndd	n 为数字。删除光标所在的向下 n 列，例如 20dd 则是删除 20 列 (常用)
d1G	删除光标所在到第一行的所有数据
dG	删除光标所在到最后一行的所有数据
d$	删除游标所在处，到该行的最后一个字符
d0	那个是数字的 0，删除游标所在处，到该行的最前面一个字符
yy	复制游标所在的那一行 (常用)
nyy	n 为数字。复制光标所在的向下 n 列，例如 20yy 则是复制 20 列 (常用)
y1G	复制游标所在列到第一列的所有数据
yG	复制游标所在列到最后一列的所有数据
y0	复制光标所在的那个字符到该行行首的所有数据
y$	复制光标所在的那个字符到该行行尾的所有数据

p, P	p 为将已复制的数据在光标下一行贴上，P 则为贴在游标上一行！举例来说，我目前光标在第 20 行，且已经复制了 10 行数据。则按下 p 后，那 10 行数据会贴在原本的 20 行之后，亦即由 21 行开始贴。但如果是按下 P 呢？那么原本的第 20 行会被推到变成 30 行。(常用)
J	将光标所在列与下一列的数据结合成同一列
c	重复删除多个数据，例如向下删除 10 行，[10cj]
u	复原前一个动作。(常用)
[Ctrl]+r	重做上一个动作。(常用)
这个 u 与 [Ctrl]+r 是很常用的指令！一个是复原，另一个则是重做一次	
.	不要怀疑！这就是小数点！意思是重复前一个动作的意思。如果你想要重复删除、重复贴上等等动作，按下小数点. 即可

Chapter 47

Bash

47.1 Overview

bash（GNU Bourne-Again SHell）是 1987 年由 Brian J. Fox 为了 GNU 计划而编写的一种 UNIX Shell，其命令语法是 Bourne shell 命令语法的超集。

大多数 Bourne shell 脚本不经修改即可以在 bash 中执行，只有那些引用了 Bourne 特殊变量或使用了 Bourne 的内置命令的脚本才需要修改。

bash 的命令语法很多来自 Korn shell（ksh）和 C shell（csh），例如命令行编辑、命令历史、目录栈、$RANDOM 和 $PPID 变量，以及 POSIX 的命令置换语法（$(...)）。

在 bash 终端下，可以使用 tab 键自动补全已部分输入的程序名、文件名、变量名等来加快输入速度。

默认情况下，Bash Shell 中命令的查找和执行顺序如下：

- 以绝对/相对路径执行命令，例如/usr/sbin/apachectl
- 执行使用 alias 设置的命令；
- 执行 bash 内置的命令；
- 执行 $PATH 中按顺序找到的第一个命令。

47.1.1 Symbol

符号	备注
#	注释符号
\	转义符号
|	管道符号，用于分隔两个管道命令
;	分隔符，用于分隔顺序执行的命令

符号	备注
~	用户的家目录
$	变量前导符
&	任务调度（job control），将命令转到后台执行
!	逻辑非
/	目录符号，用于分隔路径
>,>>	数据流重定向，输出导向，分别是覆盖与追加
<,<<	数据流重定向，输入导向
' '	单引号，不具有变量替换功能
" "	双引号，具有变量替换功能
``	反单引号，可以预先执行命令，或者使用 $()
()	中间为子 shell 的起始与结束
{}	中间为命令块的组合

在不考虑命令相关性时，连续执行的命令之间可以使用分号（;）进行分隔。例如，关机之前执行同步写入操作后再执行 shutdown 命令。

```
# sync; shutdown -h now
```

如果命令之间存在相关性，例如前面的命令执行成功是后面的命令执行的前提时，可以使用 && 和 ||。

$ 本身也是一个变量，用于表示当前 Shell 的进程代号 PID（Process ID）。

```
echo $$
```

? 也是一个变量，表示上一个执行的命令回传的值。一般来说，如果命令执行成功，则会回传 0，否则回传以非 0 值表示的“错误代码”。

```
echo $?
```

对命令之间的相关性进行判断的依据是命令返回值，若前一个命令执行成功则返回 `$?=0`。

- && 用于前面的命令执行成功（`$?=0`）才执行后面的命令，例如 `cmd1&&cmd2`
- || 用于前面的命令执行失败（`$?≠0`）才执行后面的命令，例如 `cmd1||cmd2`

例如，在执行创建文件命令时需要首先判断目录是否存在，若存在才进行创建。

```
$ ls test/ && touch test/backup.txt
```

如果要求在目录不存在时自动创建目录后才创建文件，可以对上述命令进行改写。

```
$ ls test/ || mkdir test/ && touch test/backup.txt
```

47.1.2 Integer

与 Bourne shell 不同, bash 不用另外生成进程即能进行整数运算, 并且可以使用 `((...))` 命令和 `$[...]` 变量语法来实现。

```
VAR=55              # 将整数55赋值给变量VAR
((VAR = VAR + 1))   # 变量VAR加1。注意这里没有'$'
((++VAR))           # 另一种方法给VAR加1。使用C语言风格的前缀自增
((VAR++))           # 另一种方法给VAR加1。使用C语言风格的后缀自增
echo $((VAR * 22)) # VAR乘以22并将结果送入命令
echo $[VAR * 22]    # 同上，过时的用法
```

`((...))` 命令可以用于条件语句, 因为它的退出状态是 0 或者非 0 (大多数情况下是 1), 因此可以用于是与非的条件判断。

```
if((VAR == Y * 3 + X * 2))
then
    echo Yes
fi


((Z > 23)) && echo Yes
```

`((...))` 命令支持下列比较操作符: '==', '!=', '>', '<', '>=' 和 '<='。

bash 不能在自身进程内进行浮点数运算, 只有 Korn shell 和 Z shell 支持。

47.1.3 Redirect

bash 拥有传统 Bourne shell 缺乏的 I/O 重定向语法, 而且可以同时重定向标准输出和标准错误。

```
command &> file
```

等价的重定向标准输出与标准错误的 Bourne shell 语法如下:

```
command > file 2>&1
```

另外, bash 可以用下列语法重定向标准输入至字符串 (称为 here string), 如果字符串包括空格就需要用引号包裹字符串。

```
command <<< "string to be read as standard input"
```

下面的示例说明了使用重定向标准输出至文件，写数据，关闭文件，重置标准输出。

```
# 生成标准输出（文件描述符1）的拷贝文件描述符6
exec 6>&1
# 打开文件"test.data"以供写入
exec 1>test.data
# 产生一些内容
echo "data:data:data"
# 关闭文件"test.data"
exec 1>&-
# 使标准输出指向FD 6（重置标准输出）
exec 1>&6
# 关闭FD6
exec 6>&-
```

下面的示例说明了使用重定向打开及关闭文件。

```
# 打开文件test.data以供读取
exec 6<test.data
# 读文件直到文件尾
while read -u 6 dta
do
   echo "$dta"
done
# 关闭文件test.data
exec 6<&-
```

下面的示例说明了使用重定向来抓取外部命令的输出。

```
# 运行'find'并且将结果存于VAR
# 搜索以"h"结尾的文件名
VAR=$(find . -name "*h")
```

47.1.4 Regular Expression

bash 支持进程内的正则表达式，并且正则表达式匹配字符串时上述命令的退出状态为 0，不匹配为 1。

```
[[ string =~ regex ]]
```

正则表达式中用圆括号括起的子表达式可以访问 shell 变量 BASH_REMATCH，而且不需要使用引号包裹空格或者 shell 关键字（例如 '*' 或者 '?'）。

```
if [[ abcfoobarbletch =~ 'foo(bar)bl(.*)' ]]
then
   echo The regex matches!
   echo $BASH_REMATCH      -- outputs: foobarbletch
   echo ${BASH_REMATCH[1]} -- outputs: bar
   echo ${BASH_REMATCH[2]} -- outputs: etch
fi
```

正则表达式匹配在 bash 进程内完成，因此使用进程内的正则表达式的性能要比生成一个新的进程来运行 grep 命令优越。

47.1.5 Escape Character

`$'string'` 形式的字符串会被特殊处理，字符串会被展开成 string，并将反斜杠及紧跟的字符进行替换，扩展后的结果将被单引号包裹。

转义字符	结果
\a	响铃符
\b	退格符
\e	ANSI 转义符，等价于\033
\f	翻页符
\n	换行符
\r	回车符
\t	水平制表符
\v	垂直制表符
\\	反斜杠
\'	单引号
\nnn	十进制值为 nnn 的 8-bit 字符（1 − 3 位）
\xHH	十六进制值为 HH 的 8-bit 字符（1 或 2 位）
\cx control-X 字符	

双引号包裹的字符串前若有一个美元符号（`$"..."`）将会使得字符串被翻译成符合当前 locale 的语言。

- 如果当前 locale 是 C 或者 POSIX，美元符号会被忽略。
- 如果字符串被翻译并替换，替换后的字符串仍被双引号包裹。

47.1.6 Wildcard

除了完整的字符串之外，bash 还支持使用通配符来进行查询或执行命令。例如，可以使用的示例代码来搜索/usr/bin/下以 a 开头的文件。

```
$ ls -l /usr/bin/a*
```

通配符	备注
*	0 个或多个任意字符
?	1 个任意字符
[]	在 [] 中的任何一个字符
[-]	代表在编码顺序中的所有字符，例如 [a-z] 代表 a 到 z 之间的所有字符
[^]	[] 中的第一个字符为 ^ 时表示原向选择，例如 [^abc] 表示非 a，b，c 的其他任意一个字符

通配符可以用来配合查找文件或目录等。

```
# ll -d a*
# ll -d a?
# ll -d a??
# ll -d /etc/httpd/conf.modules.d/*[0-9]*
# ll -d /etc/[^a-z]*
```

47.1.7 Script

bash 启动的时候会运行各种不同的脚本。

当 bash 作为一个登录的交互 shell 被调用，或者作为非交互 shell 但带有`--login`参数被调用时，它首先读入并执行文件/etc/profile，然后它会依次寻找～`/.bash_profile`、～`/.bash_login`和～`/.profile`，读入并执行第一个存在且可读的文件。使用`--noprofile`参数可以阻止 bash 启动时的这种行为。

当一个登录 shell 退出时，bash 读取并执行～`/.bash_logout`文件（如果此文件存在）。具体来说，～/.bash_logout 可以记录用户注销 bash 后系统继续执行的清理动作。

默认情况下，～/.bash_logout 只负责注销巴士时的清理屏幕动作，用户可以将备份或其他操作（例如清空暂存盘）写入该文件来在注销时执行。

当一个交互的非登录 shell 启动后，bash 读取并执行～/.bashrc 文件，而且这个行为可以用--norc 参数阻止。--rcfile file 参数强制 bash 读取并执行指定的 file 而不是默认的～/.bashrc。

如果用 sh 来调用 bash 时，bash 在启动后进入 posix 模式，它会尽可能模仿 sh 历史版本的启动行为以便遵守 POSIX 标准。用 sh 名字调用的非交互 shell 不会去读取其他启动脚本，--rcfile 参数无效。

当 bash 以 POSIX 模式启动时（例如带有--posix 参数）它使用 POSIX 标准来读取启动文件。在此模式下，交互 shells 扩展变量 ENV，从以此为文件名的文件中读取命令并执行。

bash 会探测自己是不是被远程 shell 守护程序运行（通常是 rshd），并读取并执行～/.bashrc 中的命令, 但是 rshd 一般不会用 rc 相关参数调用 shell，也不会允许指定这些参数。

47.2 Environment Variables

环境变量可以用于切换主文件夹、修改提示符以及修改文件查找的路径等，因此 Shell 脚本和批处理文件使用环境变量来存储临时值，用于以后在脚本中引用，也用于传递数据和参数给子进程。

一般情况下，环境变量使用大写字母表示，用户自定义的变量可以使用小写字母表示。

- 在 Unix 和类 Unix 系统中，一个在脚本或程序中更改的环境变量值只会影响该进程，亦可能影响其子进程。其父进程和无关进程将不受影响。
- 在 DOS 中，更改或删除一个批处理文件中的环境变量值将改变变量的期限命令的存在。

在多用户多任务的环境中，每个用户登录系统后都会取得一个 shell（默认为 bash），每个用户都有自己的环境变量，而且所有 Unix 和类 Unix 系统中的每个进程都有其各自的环境变量。

在 Unix 和类 Unix 系统通过初始化脚本启动时，通常会在此时初始化环境变量，这样环境变量才可以被系统中的其它进程所继承，而且用户可以添加环境变量到各自的 shell 脚本中。例如，在 Linux 操作系统中的所有的命令执行时都需要一个执行码，系统在进入 shell 之前使用环境变量来完成初始化操作等。

在 Windows 系统中，环境变量的缺省值存储在 Windows 注册表中，或者在 autoexec.bat 自动执行的批处理文件中设置。

在 Bash 中，当一个变量未被设置时，默认的值为“空”，而且在设置变量时应该遵循一定的规则。

- 使用 = 来连接变量和值;
- = 两侧不能直接接空格;
- 变量名只能是英文字母或数字，而且不能以数字开头;

- 变量值有空格必须使用双引号或单引号括起来;
- 双引号中的特殊字符将维持原有变量意义;
- 单引号中的特殊字符不进行转义;
- 转义字符“\”可以对特殊符号（例如 [Enter]、$、\、空格符和! 等）转义为普通字符。
- 在命令中可以使用反单引号“`command`”或“$(command)”来插入其他命令的返回值。

```
$ echo "version=`uname -r`"
$ echo "version=$(uname -r)"

$ cd /lib/modules/$(uname -r)/kernel/
$ cd /lib/modules/`uname -r`/kernel/
```

- 可以使用冒号（:）对变量内容进行追加。

```
$ PATH="$PATH:/usr/local/texlive/2014/bin/x86_64-linux"
$ PATH=$PATH:/usr/local/texlive/2014/bin/x86_64-linux
$ PATH="${PATH}:/usr/local/texlive/2014/bin/x86_64-linux"
```

在变量的设置中，单引号和双引号的最大不同在于双引号仍然可以保持变量的内容，但是单引号内只能是一般字符，不会有特殊符号。

在 Linux 中执行命令时，反单引号（`）包含的命令将会被最先执行，而其执行结果将作为外部的输入信息。例如，为了查看每个 crontab 相关文件名的权限，可以执行:

```
# ls -l `locate crontab`
-rw-r--r--. 1 root root   541 Sep 18 16:01 /etc/anacrontab
-rw-r--r--. 1 root root   451 Dec 20  2013 /etc/crontab
-rwsr-xr-x. 1 root root 57536 Sep 18 16:01 /usr/bin/crontab
-rw-r--r--. 1 root root  435 Nov 30 13:59 /usr/lib/python2.7/site-packages/cheat/cheatsheet
-rw-r--r--. 1 root root  1184 Aug  3  2013 /usr/share/bash-completion/completions/crontab
-rw-r--r--. 1 root root  2618 Sep 18 16:01 /usr/share/man/man1/crontab.1.gz
-rw-r--r--. 1 root root  4229 Sep  8 15:11 /usr/share/man/man1p/crontab.1p.gz
-rw-r--r--. 1 root root  1121 Dec 20  2013 /usr/share/man/man4/crontabs.4.gz
-rw-r--r--. 1 root root  1658 Sep 18 16:01 /usr/share/man/man5/anacrontab.5.gz
-rw-r--r--. 1 root root  4981 Sep 18 16:01 /usr/share/man/man5/crontab.5.gz
-rw-r--r--. 1 root root  2566 Oct 13 16:56 /usr/share/vim/vim74/syntax/crontab.vim
```

47.2.1 env

缺省情况下, 当一个进程被创建时，除了创建过程中的明确更改外，它继承了其父进程的绝大部分环境设置。

如果需要修改环境变量，可以在 API 层级上使用 fork 和 exec 函数进行变量设置，或者利用 Shell 文件的特殊命令调用来改变环境变量。例如，通过 env 间接替代或者使用 `ENVIRONMENT_VARIABLE=VALUE <command>` 标识。

所有的 Unix 操作系统以及 Windows 操作系统都使用环境变量，但是它们使用不同的环境变量名称，可以通过运行程序来访问环境变量的值。

- PATH 列出 Shell 搜索用户输入的执行命令所在的目录，PATH 变量中的路径顺序可能会影响命令的执行。

  ```
  # echo $PATH
  ```

- HISTSIZE 与历史命令有关，表示可以记录的最大命令数量。

  ```
  echo $HISTSIZE
  ```

- HOME（类 Unix 系统）和 userprofile（Windows）表示用户的主目录在文件系统中的位置。

  ```
  echo $HOME
  ```

- PWD 列出当前工作目录，其值和 pwd 命令的输出相同。

  ```
  echo $PWD
  ```

- DISPLAY 指出 X11 程序使用的默认显示器。

  ```
  echo $DISPLAY
  ```

- RANDOM 变量可以用来获得随机数，例如 bash 中的 $RANDOM 的范围为 0 ～ 32767。大多数操作系统都有自己的随机数发生器（例如/dev/random），并且可以通过 $RANDOM 变量来获得随机数。如果需要手动指定随机数的范围，可以使用 declare 命令来进行声明。

  ```
  # declare -i num=$RANDOM*10/32768; echo $num
  ```

- SHELL 说明当前环境使用的 shell 程序，例如 Linux 默认使用/bin/bash。

  ```
  echo $SHELL
  ```

- TERM（类 Unix 系统）指定使用 Terminal 或 Terminal Emulator 的类型（例如 vt100 或 dumb）。

  ```
  $ echo $TERM
  xterm-256color
  ```

- LANG 用于输出默认的 Locale 设置，可以使用 $LC_ 开头的变量来输出详细的设置（例如 $LC_CTYPE、$LC_COLLATE、$LC_DATE 等）。

  ```
  echo $LANG
  echo $LC_CTYPE
  echo $LC_COLLATE
  echo $LC_DATE
  ```

- CVS_RSH（类 Unix 系统）可用于 ext 方式中指明 cvs 客户端寻找远端 shell 的路径，来作为连接 cvs 服务器和以更高的优先权覆盖 $CVS_RSH 环境变量中指定的路径。

- MAIL（类 Unix 系统）指定当前用户的邮件存放目录。

```
echo $MAIL
/var/spool/mail/theqiong
echo ${MAIL}
/var/spool/mail/theqiong
```

用户可以逐一设置每个与 LANG 有关的变量，但是事实上如果未设置其他的语系变量，那么设置 LANG 或 LC_ALL 后，其他的语系变量都会被它们替代。

一般情况下，在 Linux 中仅设置 LANG 即可满足字符显示需要，整体系统默认的语系定义位于/etc/sysconfig/i18n。

```
cat /etc/sysconfig/i18n
```

env 用于查看环境变量与常见环境变量说明，也可以用于在不修改当前的环境设置的情况下执行其他工具。

env 命令可以添加或修改环境变量，其使用方式如下：

```
-i, --ignore-environment 不带环境变量启动
-u, --unset=NAME 从环境变量中删除一个变量
--help 显示帮助并退出
--version 输出版本信息并退出
```

在启动一个新的 Shell 时，如果需要清除当前的环境配置，可以使用如下的命令：

```
env -i /bin/sh
```

例如，如果需要在不同的环境中执行 X 应用（这里是 xcalc），可以执行：

```
env DISPLAY=foo.bar:1.0 xcalc
```

下面的示例说明如何在 Shell 中执行 Python 脚本，其中/usr/bin/env 是 env 命令的完整路径。

```
#!/usr/bin/env python2
print "Hello World."
```

env 和 export 都可以输出当前环境中的所有的环境变量，但是语法不同。

```
# env
# export
```

47.2.2 echo

echo 可以用来输出变量的内容，每个变量前要加上 $ 符号。

```
$ echo $MAIL
/var/spool/mail/theqiong
```

echo 命令可以执行简单的算术计算，例如 +、-、*、/、%，例如：

```
$ echo $((89%5))
4
```

另外，declare 命令可以定义变量的类型，这样只有当变量被定义为数值类型之后才能执行数值计算。

默认情况下，bash shell 仅支持整数的数值计算，因此当用户从命令输入数据并执行数值计算时，需要使用 declare 命令，例如：

```
$ cat > test.sh
#! /bin/bash
read -p "first number: " firstnum
read -p "second number: " secondnum
declare -i total=$firstnum*$secondnum
echo -e "$total"
```

47.2.3 set

除了使用 stty 之外，还可以使用 set 命令来设置 Shell 的其他变量（包括环境变量和自定义变量等）。

实际上，set 可以设置整个命令输入/输出环境，包括历史命令、显示错误内容等。

```
# set [-uvCHhmBx]
-u: 默认不启用，启用后在使用未设置变量时会输出错误信息；
-v: 默认不启用，启用后当输出信息时会首先输出信息的原始内容；
-x: 默认不启用，启用后在命令执行前会显示命令内容（前面有++符号）；
-h: 默认启用，与历史命令相关；
-H: 默认启用，与历史命令相关；
-m: 默认启用，与任务管理相关；
-B: 默认启用，与[]的作用相关；
-C: 默认不启用，使用>时文件存在则不会被覆盖。
```

```
# 显示当前set的所有设置，bash默认为himBH。
$ echo $-
himBH
$ echo $OSTYPE
linux-gnu
$ echo $HOSTTYPE
x86_64
$ echo $MACHTYPE
x86_64-redhat-linux-gnu
```

Windows 操作系统的 set 命令可以用于显示、设置和删除环境变量（例如如时间，提示符等）。从 Windows 2000 起，通过添加/P 参数，set 命令可以用来接收命令行的输入，例如：

```
Set /P Choice = Type your text.
echo You typed: "%choice%"
```

在当前环境的变量中，比较重要的包括 PS1、$、? 和 OSTYPE、HOSTTYPE、MACHTYPE 等。

PS1 表示命令提示符，每次执行完当前命令并返回命令行时都会去读取变量 PS1 的值，而且 PS1 中的特殊符号可以显示不同的命令。

- \d 表示日期格式；
- \H 表示完整的主机名；
- \h 仅取主机名在第一个小数点之前的命令，后面的省略；
- \t 表示 24 小时的时间格式（HH:MM:SS）；
- \T 表示 12 小时的时间格式（HH:MM:SS）；
- \A 表示 24 小时的时间格式（HH:MM）；
- \@ 表示 12 小时的时间格式（am/pm）；
- \u 表示当前用户的帐号；
- \v 表示 bash 的版本信息；
- \w 表示完整的工作目录名称（由根目录开始），用户家目录以～表示；
- \W 表示通过 bashname 函数取得的工作目录名称，仅会列出最后一个目录名；
- \# 表示指定的命令号码；
- \$ 用来说明 root 用户（#）和普通用户（$），默认情况下的 PS1 值如下：

  ```
  set | grep -i 'ps1'
  PS1='[\u@\h \W]\$ '
  ```

  $ 本身也是一个变量，用于表示当前 Shell 的进程代号 PID（Process ID）。

```
echo $$
```

? 也是一个变量，表示上一个执行的命令回传的值。一般来说，如果命令执行成功，则会回传 0，否则回传以非 0 值表示的“错误代码”。

```
echo $?
```

/etc/inputrc 也可以用来设置其他的按键。

47.2.4 unset

unset 命令可以用于取消变量。

```
$ unset MAIL
```

47.3 Variable Setting

除了可以通过直接设置来修改变量值之外，还可以对变量值进行删除、替代和替换。

```
# echo $PATH
# echo ${PATH}
# echo ${PATH#/*home/theqiong/bin:}
```

从前面对变量值进行删除的一般语法为`${variable#/*path:}`

- `${}` 是删除模式中必须包含的；
- `variable` 说明原来的变量名；
- `${variable#}` 内部的 # 表示从变量值的最前面开始向右删除，且仅删除最短的变量值；
- `${variable#/*path:}` 内部的 *path 代表要被删除的部分，其中 # 代表从前面开始删除，而且可以通过通配符 * 表示 0 到无穷多个任意字符。

要删除的目录就是介于斜杠（/）和冒号（:）之间的数据，其中一个 # 表示仅删除指定的最短的路径，两个 # 则表示删除最长的路径。

- `${PATH#/*path:}` 表示删除与 path 匹配的最短的路径；
- `${PATH##/*path:}` 表示删除与 path 匹配的最长的路径。

如果需要从后面向前删除变量值，需要使用百分号（%），其一般形式为`${PATH%:*bin}`同理，一个百分号表示由最后面开始删除最短的路径，两个百分号则表示最长的路径。

如果需要对变量值进行替换，可以使用如下的语法：

```
# echo ${PATH/old/new}
```

如果使用两条斜杠则表示对所有符合的内容都进行替代，例如：

```
# echo ${PATH//old/new}
```

设置方式	备注
${变量 # 关键字}	若变量内容从头开始的数据符合“关键字”，则将符合的最短数据删除
${变量 ## 关键字}	若变量内容从头开始的数据符合“关键字”，则将符合的最长数据删除
${变量% 关键字}	若变量内容从尾开始的数据符合“关键字”，则将符合的最短数据删除
${变量%% 关键字}	若变量内容从尾开始的数据符合“关键字”，则将符合的最长数据删除
${变量/old/new}	若变量内容符合“old”，则第一个“old”将会被“new”替换
${变量//old/new}	若变量内容符合“old”，则所有的“old”将会被“new”替换

在某些情况下可能需要“判断”变量是否存在，若变量存在则使用既有的设置，否则将给予一个常用的设置。

```
# echo $username
# username=${username-theqiong}
# echo $username
theqiong
# username=${username:-root}
# echo $username
theqiong
```

- 如果在减号（-）后面跟上关键字，则可以在变量不存在时为其指定一个值；
- 如果在减号前有冒号（:），则可以在变量内容为空或未设置时为其指定一个新值。

变量设置方式	str 没有设置	str 为空字符串	str 已设置为非空字符串
var=${str-expr}	var=expr	var=	var=$str
var=${str:-expr}	var=expr	var=expr	var=$str
var=${str+expr}	var=	var=expr	var=expr
var=${str:+expr}	var=	var=	var=expr
var=${str=expr}	str=expr var=expr	str 不变 var=	str 不变 var=$str
var=${str:=expr}	str=expr var=expr	str=expr var=expr	str 不变 var=$str
var=${str?expr}	expr 输出到 stderr	var=	var=str
var=${str:?expr}	expr 输出到 stderr	expr 输出到 stderr	var=str

47.4 Shell builtins

47.4.1 alias

```
$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
```

```
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
```

47.4.2 expr

47.4.3 sleep

47.4.4 test

47.4.5 true

47.4.6 false

47.4.7 yes

47.4.8 ulimit

bash 可以限制用户使用的系统资源，例如可以打开的文件数量，可以使用的 CPU 时间和可以使用的内存总量等。

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 15732
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 1024
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

- -H：hard limit，严格限制不能超过的数值；
- -S：soft limit，如果超过则有警告信息，而且通常 soft 会比 hard 小。
- -a：列出所有的限制额度；
- -c：限制每个内核文件的最大容量；
- -f：限制可以创建的最大文件容量（单位为 KB），一般可能设置为 2GB；
- -d：进程可以使用的最大段内存容量（segment）；
- -l：可以用于锁定（lock）的内存量；
- -t：可以使用的最大 CPU 时间（单位为秒）；
- -u：单一用户可以使用的最大进程数量；

单一文件系统能够支持的单一文件大小与 block 的大小有关。例如，block size 为 1024byte 时，单一文件可以达到 16GB 的容量，不过可以使用 `ulimit -f` 来设置用户可以创建的文件大小（单位为 KB）。

另外，一般用户只能通过 ulimit 来减小文件容量，不能增大文件容量，而且可以使用 pam 来设置用户的 ulimit 限值。

47.4.9 stty

用户在登录终端时可以自动获得输入环境的设置，使用 stty 命令可以设置终端的输入按键意义等。

```
# stty -a
speed 38400 baud; rows 65; columns 104; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D;
eol = M-^?; eol2 = M-^?; swtch = M-^?; start = ^Q;
stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr
icrnl ixon -ixoff -iuclc ixany imaxbel iutf8 opost -olcuc -ocrnl
onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh
-xcase -tostop -echoprt echoctl echoke
```

其中：

- eof：代表输入结束（End Of File）；
- erase：向后删除字符；
- intr：发送中断信号给当前运行的进程；

快捷键	备注
ctrl+c	终止当前命令
ctrl+d	输入结束（EOF），或者从前向后删除字符
ctrl+m	Enter
ctrl+s	暂停屏幕输出
ctrl+q	恢复屏幕输出
ctrl+u	向前删除整行命令到当前位置
ctrl+z	暂停当前命令
ctrl+r	历史命令
ctrl+l	清空当前屏幕
ctrl+d	退出虚拟终端
ctrl+a	跳转到命令开始
ctrl+e	跳转到命令结尾
ctrl+p	切换历史命令
ctrl+t	将单个字符向后移动
alt+t	调换命令和参数
alt+b	向前移动单个单词
alt+.	插入历史命令

- kill：删除命令行上的所有输入；
- quit：发送退出信号给当前运行的进程；
- start：在某个进程停止后重新启动；
- stop：停止当前屏幕输出；
- susp：发送停止信号给当前运行的进程。

47.5 User Environment

47.5.1 clear

clear 命令用于清屏，其功能与 MS-DOS 命令 cls 相似。

clear 使用 terminfo 或 termcap 数据库，以及查询环境变量以获取终端类型，并以此决定清空屏幕的方式。

在实际使用中，clear 命令不需任何参数，如果附加参数执行则 clear 命令会自动忽略所有参数。

47.5.2 exit

47.5.3 finger

47.5.4 history

历史命令记录在用户主目录下的.bash_history 文件中，而且.bash_history 中记录的是当前登录之前执行过的命令，在当前登录中所执行的命令都暂存在内存中，在注销登录后才会被追加到.bash_history 文件中。

默认情况下，bash 可以将 1000 个历史命令保存到～/.bash_history 中，并且在在登录 bash 后可以将～/.bash_history 读入内存，因此可以在命令行中使用上下键切换。

一般情况下，历史命令的读取与记录的过程如下:

- 当用户以 bash 登录主机后，系统会主动由主目录下的～/.bash_history 读取历史命令，并且通过 bash 的 HISTSIZE 变量来设置。
- 当用户注销登录后，系统会将新执行的历史命令更新到～/.bash_history 中。
- 用户也可以使用 `history -w` 强制将历史命令立刻写入～/.bash_history 中。

～/.bash_history 中记录的历史命令总是等于 HISTSIZE，旧的命令会被主动清理，并保留最新的。

- ! 可以用来执行指定号码的历史命令;
- !! 可以用来执行上一条命令。
- !x 可以用来执行最近以 x 开头的命令。

同一帐号以多个 bash 登录时，只有最后注销的 bash 的历史命令会被保存，其他的将被覆盖，因此可以使用单一 bash 登录后使用任务调度（job control）来切换不同的工作，这样可以记录所有的历史命令。

在网络入侵中，可以从用户的历史命令中获取机密信息（例如 MySQL 密码），而且在退出时会清除命令操作记录，从而增加追查的难度。

47.5.5 id

47.5.6 logname

47.5.7 mesg

47.5.8 export

在一般情况下，父进程的自定义变量是无法在子进程内使用的，不过可以通过 export 命令来把普通变量导出为环境变量来提供其他子进程使用。

```
$ PATH=$PATH:/usr/local/texlive/2014/bin/x86_64-linux
$ export PATH
```

```
$ export $PATH=PATH:/usr/local/texlive/2014/bin/x86_64-linux
```

另外，环境变量也可以称为全局变量（global variable），用户自定义变量可以称为局部变量（local variable）。

环境变量和用户自定义变量之间的差异在于环境变量的数据可以被子进程引用，或者说子进程仅可以继承父进程的环境变量，不会继承父进程的自定义变量。

- 当登录系统并启动 Shell（例如 bash）时，操作系统会分配内存给 Shell，该内存区域内的变量可以被子进程引用；
- 如果返回主进程，必须结束子进程（执行 exit 或 logout）；
- 如果在父进程中使用 export 导出变量，将把自定义变量的值写入 Shell 的内存区域中。
- 在加载另外的 Shell（即启动子进程而离开父进程）时，子 Shell 可以将父 shell 的环境变量所在的内存区的内容导入自己的环境变量块中。

需要注意的是，“环境变量”和“bash 环境”意义不同，例如 PS1 并不是环境变量，但是 PS1 可以影响 bash 的接口（提示符）。

47.5.9 read

read 命令可以读取来自键盘输入的变量，通常用于在 shell script 与用户交互。

```
$ read [-pt] variable
```

- 不加任何参数时，直接加上变量名称时，read 命令将读取键盘输入并存入变量；
- -p 参数后面可以跟上提示符；
- -t 参数可以设置输入等待的时间。

47.5.10 array

47.5.11 declare

declare 和 typeset 都用于声明变量的类型，其中如果直接使用 declare 则会输出所有的变量及其值，效果同 set。

- -a：将后面的变量定义为数组类型；
- -i：将后面的变量定义为整数类型；
- -x：将后面的变量导出为环境变量（同 export）；
- -r：将后面的变量设置为 readonly 类型，该变量无法更改内容，也不能重设；
- -p：输出变量的类型。

```
$ sum=1+2
$ echo $sum
1+2
$ declare -i sum=1+2
$ echo $sum
3
```

Shell 中的变量默认为“字符串”，而且 bash 环境中的数值计算，默认最多仅能到达整数范围，因此 1/3 的结果是 0。

另外，在 shell script 中可以使用 declare 来声明数组的属性。如果将变量设置为“只读”，通常需要注销后重新登录才能恢复变量的类型。

在 bash 中，数组的设置方式如下：

```
var[index]=content
```

```
$ var[1]="one''
$ var[2]="two''
$ var[3]="three''
$ echo "${var[1]}, ${var[2]}, ${var[3]}"
one, two, three
```

47.5.12 passwd

47.5.13 su

47.5.14 sudo

47.5.15 uptime

47.5.16 talk

47.5.17 tput

47.5.18 uname

47.5.19 w

47.5.20 wall

47.5.21 who

47.5.22 whoami

47.5.23 write

47.5.24 locale

如果要查询当前系统支持的所有语系信息，可以使用 locale 命令。

```
// show current locale info
$ locale
LANG=en_US.utf8
LC_CTYPE="en_US.utf8"
LC_NUMERIC="en_US.utf8"
LC_TIME="en_US.utf8"
LC_COLLATE="en_US.utf8"
LC_MONETARY="en_US.utf8"
LC_MESSAGES="en_US.utf8"
LC_PAPER="en_US.utf8"
LC_NAME="en_US.utf8"
LC_ADDRESS="en_US.utf8"
LC_TELEPHONE="en_US.utf8"
LC_MEASUREMENT="en_US.utf8"
```

```
LC_IDENTIFICATION="en_US.utf8"
LC_ALL=
// show all locale info
$ locale -a
```

47.5.25 type

type 命令可以查询可执行文件的类型，并说明是否为内置命令或者外部程序提供的命令。如果查询的命令为外部命令，则 type 可以输出命令的路径，因此 type 的作用类似 which。

```
$ type dd
dd is /usr/bin/dd
$ which dd
/usr/bin/dd
```

type 命令支持的命令类型包括 shell built-in、function、alias、hashed command 和 keyword 等，在查询失败时会返回非零值。

type [-aftpP] name [name ...]
不加参数时，type会说明name是外部命令还是内置命令。
-t，type将name以file、alias或builtin来说明其意义。
-p，type输出外部命令的完整文件名。
-a，type输出PATH变量定义的路径中所有包含name的命令（包含alias）。

```
$ type test
test is a shell builtin
$ type cp
cp is /bin/cp
$ type unknown
-bash: type: unknown: not found
$ type type
type is a shell builtin
$ type -a gzip
gzip is /opt/local/bin/gzip
gzip is /usr/bin/gzip
```

47.6 Redirect

数据流重定向是将某个命令执行后本来应该输出到屏幕的数据保存到其他的位置（例如文件或设备)。

- stdout 可以认为是命令执行成功后回传的信息;
- stderr 可以认为是命令执行失败后回传的信息;

数据流重定向就是将 stdout 或 stderr 传送到其他的文件或设备，其中 1>> 和 2>> 之间是没有空格的。

- stdin（标准输入）的代码为 0，使用 < 或 <<;
- stdout(标准输出）的代码为 1，使用 > 或 >>;
- stderr（标准错误）的代码为 2，使用 2> 或 2>>。

管道中使用前一个命令的 stdout 作为输入数据（stdin）时，某些命令（例如 tar）需要使用文件名来进行处理，那么 stdin 和 stdout 可以使用减号（-）来代替。

例如，下面的示例使用 tar 将/home 中的文件打包，而且打包的数据不是记录到文件，而是发送到标准输出来供管道中的 tar 使用，那么后面的 tar 命令中的-就是使用前一个命令的标准输出，从而省去使用文件名的麻烦。

```
# tar -cvf - /home | tar -xvf -
```

47.6.1 stdout

使用重定向将正确的和错误的输出数据分别保存到不同的文件中。

```
find /home -name theqiong > list_right 2> list_error
```

- 1> 以覆盖的方式将正确的数据输出到指定的文件或设备上;
- 1>> 以追加的方式将正确的数据输出到指定的文件或设备上;
- 2> 以覆盖的方式将错误的数据输出到指定的文件或设备上;
- 2>> 以追加的方式将错误的数据输出到指定的文件或设备上。

47.6.2 stderr

在实际应用中，为了忽略错误信息，可以将错误信息重定向到/dev/null 中。

```
find /home -name theqiong > list_right 2> /dev/null
```

另外，如果需要将所有的信息（正确的和错误的）都写入同一个文件，可以在将正确信息写入后再写入错误信息。

```
find /home -name theqiong > list 2>&1
```

或者，可以对上述格式进行简化。

```
find /home -name theqiong &> list
```

如果使用数据同时写入的语法则会导致数据交叉地写入文件而产生次序错误。

```
find /home -name theqiong > list 2> list
```

47.6.3 stdin

使用重定向来读入标准输入可以将原本需要由键盘输入的数据改由文件内容来替代。

```
$ cat > testfile
123
abc

$ cat testfile
123
abc
$ cat > testfile < .bashrc
$ cat testfile
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
. /etc/bashrc
fi

# Uncomment the following line if you don't like systemctl's auto-paging feature:
# export SYSTEMD_PAGER=

# User specific aliases and functions

export PATH=$PATH:/usr/local/texlive/2014/bin/x86_64-linux

# sogou input
# export GTK_IM_MODULE=fcitx
# export QT_IM_MODULE=fcitx
```

```
# export XMODIFIERS="@im=fcitx"

export EDITOR=/usr/bin/vim
export CHEATCOLORS=true

export JAVA_HOME="/usr/java/latest"
```

在使用标准输入时，<< 可以用于表示结束输入。例如，使用 cat 命令输出内容到其他文件时，可以使用 eof 来表示结束输入，从而可以省略 Ctrl+D 结束输入操作。

```
$ cat > testfile << "eof"
123
abc
eof
$ cat testfile
123
abc
```

47.7 Pipe

管道（Pipeline）是一个由标准输入输出链接起来的进程集合。

- 每一个进程的输出（stdout）被直接作为下一个进程的输入（stdin）；
- 每一个链接都由未命名管道实现。

在每个管道后面接的第一个数据必须是命令，而且命令必须要能够接收 stdout 数据。

- less，more，head 和 tail 等都是管道命令；
- ls，cp，mv 等都不是管道命令。

管道命令仅能处理由前面的一个命令产生的正确信息（stdout），不能处理错误信息（stderr）。另外，netcat 和 socat 等工具可以将管道连接到 TCP/IP 套接字。

下面的管线示例是一种由 URL 标示的万维网资源的拼写检查器。

```
curl "http://en.wikipedia.org/wiki/Pipeline_(Unix)" | \
sed 's/[^a-zA-Z ]/ /g' | \
tr 'A-Z ' 'a-z\n' | \
grep '[a-z]' | \
sort -u | \
comm -23 - /usr/share/dict/words | \
less
```

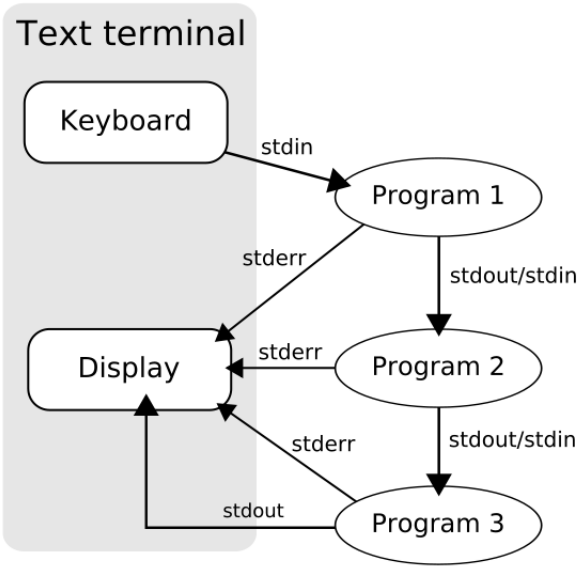


Figure 47.1: 管道示例

1. curl 获取该网页的 HTML 内容（在有些系统上可以使用 wget)。
2. sed 移除非空格的字符和网页内容的字母，并以空格取代之。
3. tr 把大写字母改成小写字母，并把行列里的空格换成新行（每个词现在各占有独立的一行)。
4. grep 过滤得到那些至少有一个小写字母的行（删除空行)。
5. sort 将“单词”（也就是每一个行）按照字母顺序排序，并且通过命令行的-u 参数来删除重复的行。
6. comm 查找两个文件中的共同行，-23 过滤掉只有第二个文件拥有的行、两个文件共有的行，仅仅留下只在第一个文件中有的行。在文件名的位置上的-参数表示要求 comm 使用标准输入（在这个例子里，comm 的标准输入来自于管道上游的标准输出）作为输入，而不是以普通文件作为输入。最终得到一串没有出现在/usr/share/dict/words 之中的“单词”(也就是一行)。
7. less 允许用户翻页浏览结果。

所有广泛应用于 UNIX 和 Windows 中的 shell 程序都有特殊的语法构建管线，典型语法是使用 ASCII 中的垂直线“|”。

当出现管道符（|）时 shell 会启动各个进程，并调整各个进程的标准流之间的连接（还包括安排一些缓存)。

通常，管线中的进程的标准错误流（“stderr”）会通过管道传输，它们被合并输出到控制台。不过，很多 Shell 提供一些扩充的语法去改变这一行为。

- 在 csh Shell 和 bash 中，使用“|&”代替“|”来表示错误流也需要被合并进入标准输

出，并传递给下一个进程。

- Bourne shell 也可以合并错误流，通过 2>&1 也可以将错误流重定向到一个不同的文件。

在一些常用的简单管线中，shell 仅仅只是用管道来连接每个子进程，然后在子进程中执行外部命令，因此 shell 本身没有通过管线来处理数据。

实际上，shell 也有可能直接处理管线数据，称为 pipemill。

```
command | while read var1 var2 ...; do
   # process each line, using variables as parsed into $var1, $var2, etc
   # (note that this is a subshell: var1, var2 etc will not be available
   # after the while loop terminates)
   done
```

使用 C 语言在 UNIX 中使用 pipe(2) 系统调用时，这个函数会让系统构建一个匿名管道，这样在进程中就打开了两个新的，打开的文件描述符：一个只读端和一个只写端。

匿名管道的两端是两个普通的匿名文件描述符，这就让其他进程无法连接该管道。为了避免死锁并利用进程的并行运行的好处，有一个或多个管道的 UNIX 进程通常会调用 fork(2) 产生新进程。

每个子进程在开始读或写管道之前都会关掉不会用到的管道端，或者进程会产生一个子线程并使用管道来让线程进行数据交换。

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int
main(int argc, char *argv[])
{
  int pipefd[2];
  pid_t cpid;
  char buf;

  if (argc != 2) {
    fprintf(stderr, "Usage: %s <string>\n", argv[0]);
    exit(EXIT_FAILURE);
  }
```

```
  if (pipe(pipefd) == -1) {
    perror("pipe");
    exit(EXIT_FAILURE);
  }

  cpid = fork();
  if (cpid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
  }

  if (cpid == 0) {    /* Child reads from pipe */
    close(pipefd[1]);          /* Close unused write end */
    while (read(pipefd[0], &buf, 1) > 0)
      write(STDOUT_FILENO, &buf, 1);

    write(STDOUT_FILENO, "\n", 1);
    close(pipefd[0]);
    _exit(EXIT_SUCCESS);

  } else {            /* Parent writes argv[1] to pipe */
    close(pipefd[0]);          /* Close unused read end */
    write(pipefd[1], argv[1], strlen(argv[1]));
    close(pipefd[1]);          /* Reader will see EOF */
    wait(NULL);                /* Wait for child */
    exit(EXIT_SUCCESS);
  }
}
```

具名管道可以通过调用 mkfifo(2) 或 mknod(2) 来构建，当被调用时表现为输入或输出的文件，这样可以允许创建多个管道，并且将其同标准错误重定向或 tee 结合起来使用更为有效。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  char filename[] = "test_fifo";
  if (!mkfifo(filename,S_IRUSR | S_IWUSR| S_IRGRP|S_IWGRP)){
    pid_t pid = fork();
    if (pid == 0){ //child
      int fd = open(filename, O_WRONLY);
      if (fd < 0)
perror("child open()");
      else{
if (strlen(argv[1]) != write(fd, argv[1], strlen(argv[1])))
  perror("child write error");
else
  close(fd);
      }
    }
    else if (pid > 0){ //father
      int fd = open(filename, O_RDONLY);
      if (fd < 0)
perror("father open()");
      else{
char buffer[200];
int readed = read(fd, buffer, 199);
close(fd);
buffer[readed] = '\0';
printf("%s\n",buffer);
      }
    }
    else
      perror("fork()");
  }
  else
```

```
    perror("mkfifo() error:");
}
```

以上代码在编译后运行时给出一个参数，子进程会将该参数内容写入管道（该管道在当前目录下，文件名为“test_fifo”），父进程从管道中读取内容并显示出来。

在大多数类 UNIX 操作系统中，管线上的所有进程同时启动，输入输出流也已经被正确地连接，并且这些进程被调度程序所管理。

所有的 UNIX 管道和其他管道实现不一样的地方就是缓存的概念：输出进程可能会以每秒 5000 byte 的速度输出，但是接收进程也许每秒只能接收 100 byte，但不会有数据丢失。

管道上游的进程的所有输出都会被放入一个队列中。当下游进程开始接收数据时，操作系统就会将数据从队列传至接收进程，并将传完的数据从队列中移除。当缓存队列空间不足时，上游进程会被终止，直到接收进程读取数据为上游进程腾出空间。

在 Linux 中，缓存队列的大小是 65536 byte。

47.7.1 cut

一般来说，选取信息通常都是针对“行”为单位进行分析，例如 cut 命令就是截取每行输入（通常是文件的片段）。

截取行片段可以通过比特（-b）、字符（-c）、或者以分隔字符（-d，默认为跳位字符）分隔的字段（-f）达成。

```
cut -d 'DELIM' -f fields
cut -c characters
```

- -d 后面接分隔字符，例如冒号（:）；
- -f 用于指定信息段的序号:
- -c 指定获取固定字符区间的单位。

```
$ echo $PATH | cut -d ':' -f 2
/usr/local/bin
$ echo $PATH | cut -d ':' -f 3,5
/usr/bin:/usr/local/sbin
$ echo $PATH | cut -d ':' -f 1-3
/usr/lib64/qt-3.3/bin:/usr/local/bin:/usr/bin
```

例如，可以使用 cut 配合 last 命令来输出最近登录的用户。

```
$ last | grep -d ' ' -f 1
```

使用 cut 执行截取时都必须给定范围，包括下面四种之一：N, N-M, N-（N 到行尾），或者-M（行头到 M）。

cut 的主要应用范围是对同一行中的数据进行分解，并且使用某些字符来作为切割的参数分解数据获得用户需要的数据，尤其在分析 log 文件等应用。

47.7.2 grep

grep 处理一行信息并找出匹配所给正则式的行或片段。

具体来说，在给出文件列表或标准输入后，grep 会对匹配一个或多个正则表达式的文本进行搜索，并只输出匹配（或者不匹配）的行或文本。

```
grep [-acinv] [--color=auto] 'pattern' file
```

- -a 表示在二进制文件以文本文件的方式查找数据;
- -c 计算找到的符合条件的次数;
- -i 忽略大小写进行查找;
- -n 输出行号;
- -v 表示反向选择，即不符合条件的行。

```
last | grep 'root' | cut -d ' ' -f 1
```

grep 原先是 ed 下的一个应用程序，其名称来自于 g/re/p（globally search a regular expression and print，以正规表示法进行全域查找以及打印）。

grep 的名称来自 Unix 文本编辑器 ed 类似操作的命令，在 ed 下输入 g/re/p 这个命令后可以将所有符合先定义样式的字符串，并以行为单位进行打印。

```
g/re/p
```

grep 命令搜索整个文件中匹配给定正则表达式的文本行并显示出来，而且可以修改 grep 的默认行为。

- 显示出不匹配的文本行;
- 查找或排除搜索的文件;
- 用不同的方式在输出中进行注释。

默认情况下，grep 是大小写敏感的，而且可以参数 -e 来使用多个匹配样式来进行搜索。

grep 可以配合正则表达式使用，并且可以支持某些高级操作。

- -A 参数后面（after）可以加数字，可以列出本行之后的指定行数;
- -B 参数后面（before）可以加数字，可以列出本行之前的指定行数。

```
$ dmesg | grep -n -A3 -B2 --color=auto 'eth'
$ grep -n 'the' test.txt
$ grep -vn 'the' test.txt
$ grep -in 'the' test.txt
$ grep -n 't[es]sing' test.txt
$ grep -n '[^g]oo' test.txt
$ grep -n '[^a-z]oo' test.txt
$ grep -n '^[a-z]' test.txt
$ grep -n '^[[:lower:]]' test.txt
$ grep -n '[0-9]' test.txt
$ grep -n '[^[:lower:]]' test.txt
$ grep -n '[:digit:]' test.txt
$ grep -n '^the' test.txt
$ grep -n '^[^a-zA-Z]' test.txt
$ grep -n '\.$' test.txt
$ grep -n '^$' test.txt
$ grep -n 'g..d' test.txt
$ grep -n 'oo*' test.txt
```

在 Perl 中，grep 是内置的功能，当提供正则表达式（或通用代码块）和一个列表时可以返回列表中匹配表达式的元素。

Windows 操作系统提供了 findstr 工具来执行 grep 的大多数功能。

许多文档处理器现在也有了使用正则表达式搜索的功能，这些功能常被称为“grep 工具”或“grep 模式”并可以创建“grep 样式”。

47.7.3 agrep

agrep 表示“近似的 grep”，用于模糊字符串搜索。

47.7.4 egrep

egrep 用于搜索更复杂的正则表达式语法。

egrep 使用了 Ken Thompson 的正则表达式实现后添加到 UNIX 的扩展正则表达式语法。

47.7.5 fgrep

fgrep 用于固定样式搜索。

fgrep 简单地读取一系列固定字符串来对文件进行搜索。

egrep、fgrep 和 grep 基本上是相同的，只是调用了不同的参数，而且 grep 可以实现大多数功能。

egrep 和 fgrep 等早期的修改版都被加入到多数现代的 grep 实现中，只需要使用简单的命令行参数就可以调用，例如在 GNU 中只要分别简单地加上-E 和-F 就可以调用 egrep 和 fgrep。

47.7.6 pgrep

pgrep 可以用来显示名称匹配正则表达式的进程。

默认情况下，pgrep 命令可以搜索出所有名字与所给正则表达式相匹配的进程，并返回相应进程标识符。

- -l 参数可以返回进程名；
- -g 参数指定搜索的进程组范围；
- -u 参数指定进程所属用户、
- -n 参数指定是否最近启动进程；
- -v 参数指定反转搜索。

47.7.7 sort

sort 命令可以依据不同的数据类型以及编码等条件进行排序。

```
sort [-fbMnrtuk] [file|stdin]
```

- -f 忽略大小写的差异；
- -b 忽略最前面的空格符部分；
- -M 以月份的名字进行排序；
- -n 使用“纯数字”进行排序（默认以字母顺序进行排序）；
- -r 反向排序；
- -u 表示在相同的数据中仅出现一行代表；
- -t 表示分隔符，默认为 Tab 键进行分隔；
- -k 指定进行排序的区间。

47.7.8 wc

wc 可以用于统计文件的整体信息，例如行数、单词数和字节数等，而且较新版本的 wc 可以区别比特和字符的统计。

对于 Unicode 字符集等包含了多字节的字符，可以通过选择 -c 或是 -m 参数来选择所需的行为。

- -l 输出行数统计（line）;
- -L 输出文件中最长一行的长度;
- -w 仅输出单词数统计（word）;
- -m 输出字符数统计（char）;
- -c 输出字节数统计（byte）。

具体来说，wc 从标准输入流或文件列表读取文件，并生成一个或多个下列统计信息：文件包含的字节数、单词数以及文件的行数（也就是换行符的个数）。

如果用户提供的是一个文件列表，则每个文件的单独统计和总体统计结果都会给出。

```
$ wc foo bar
      40     149     947 foo
    2294   16638   97724 bar
    2334   16787   98671 total
```

第一列表示文件中的行数，以上实例表示文本文件 foo 有 40 行，并且 bar 文件包含 2294 行，总计 2334 行。

第二列表示文件中的单词个数：foo 文件包含 149 个单词，且 bar 文件中有 16638 个单词，总计 16787 个单词。

第三列表示文件中包含的字符个数：foo 文件总共有 947 个字符，且 bar 文件中有 97724 个字符，总计有 98761 个字符。

例如，下面的命令用于输出当前月中登录系统的总人数。

```
last | grep [a-zA-Z] | grep -v 'wtmp' | wc -l
```

类似地，可以输出当前系统中的帐号数目。

```
cat /etc/passwd | wc -l
```

47.7.9 uniq

uniq 命令用于在排序行数据时只显示一个重复的行。

```
last | cut -d ' ' -f 1 | sort | uniq -c
```

47.7.10 tee

tee 命令能够将某个指令的标准输出进行重定向或保存到某个文件中，Unix shell、Windows PowerShell 都支持 tee 功能。

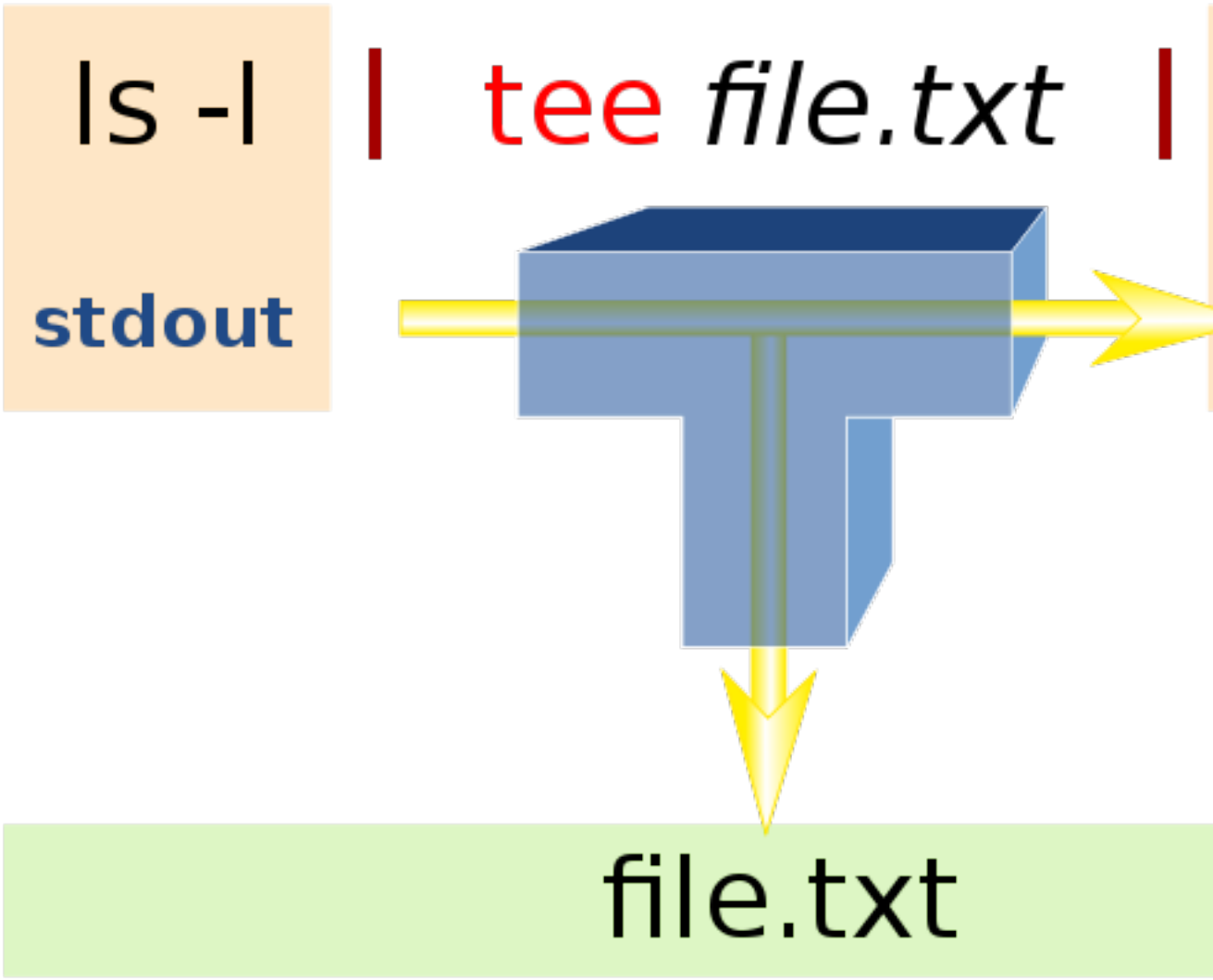


Figure 47.2: 使用 tee 的示意图

在下面使用 tee 的示意图中，`ls -l` 的输出被导向 tee，并且复制到文件 file.txt 以及下一个命令 less。

tee 的功能通常是使管道不仅可以在屏幕输出，而且也能够将其保存在文件中。

当一个数据在被另一个指令或程序改变之前的中间输出，也能够用 tee 来捕捉它。

tee 命令能够读取标准输入，之后将它的内容写入到标准输出，同时将它的副本写入特定的档案或变量中。

```
tee [ -a ] [ -i ] [文件 ... ]
```

- -a 追加到目标文件而不是覆盖
- -i 忽略中断。

tee 可以同时将数据流发送到文件和屏幕。

```
last | tee last.backup | cut -d ' ' -f 1
```

47.7.11 tr

对于 DOS 断行字符和 UNIX 断行字符，可以使用 dos2UNIX 和 UNIX2dos 进行转换。

对于更普遍的操作（例如大小写转换以及键转换），可以使用 tr 来删除指定的字符，或者进行文本信息的替换。

```
tr [-ds] SET1 ...
```

- -d 删除信息中的字符串 SET1；
- -s 替换重复的字符。

例如，将 last 命令输出的信息中的所有小写字符替换为大写字符。

```
last | tr -s '[a-z]' '[A-Z]'
```

或者，将/etc/passwd 输出的信息中的冒号（:）删除。

```
cat /etc/passwd | tr -d ':'
```

使用 tr 命令执行的操作也可以写入正则表达式，而且 tr 本身也是使用正则表达式的方式来执行替换的。

47.7.12 col

col 命令可以用来将 tab 键转换为对等的空格符（-x），或者在文本中有反斜杠（/）时仅保留反斜杠最后接的字符（-b）。

例如，调用 cat 命令的-A 参数可以显示所有特殊按键，然后使用 col 将其转换为空白，因此可以使用 col 命令将 man page 转换为纯文本文件。

```
cat -A ~/test.txt | col -x | cat -A | more
```

47.7.13 join

join 可以处理相关的两个文件之间的数据，并且将其中相同的数据的那一行拼接到一起。

```
join [-ti12] file1 file2
```

- -t 表示 join 默认以空格符分隔数据，并且对比“第一个字段”的数据。如果两个文件相同，则将两条数据连成一行，且第一个字段放在开始；
- -i 表示忽略大小写；
- -1 指定第一个文件要用哪个字段来分析；
- -2 指定第二个文件要用哪个字段来分析。

```
join -t ':' - 1 /etc/passwd -2 3 /etc/group
```

在使用 join 之前，需要处理的文件应该事先进行排序处理。

47.7.14 paste

在使用 join 命令时需要比较两个文件的数据相关性，但是 paste 命令直接将两行拼接到一起，并且在中间以 tab 键分隔。

```
paste [-d] file1 file2
```

- -d 后面可以接分隔字符，默认以 tab 来分隔；
- 指定 file 部分为-时，表示数据来自 stdin。

47.7.15 expand

expand 可以用于在 tab 键和空格键之间转换。

```
expand [-t] file
```

一般说来，一个 tab 键可以使用 8 个空格键替换，-t 后面可以指定数字来定义 tab 键和空格键之间的对应关系。

例如，下面的示例可以列出 man_db.conf 中 MANPATH 开头的前 3 行，并且手动指定 tab 键对应的空格键。

```
grep '^MANPATH' /etc/man_db.conf | head -n 3 | expand -t 6 - | cat -A
```

47.7.16 unexpand

unexpand 命令可以用来将空白转成 tab 键。

47.8 Xargs

xargs 可以读入 stdin 的数据，并且以空格符或断行字符进行确认，并将 stdin 的数据分隔成其他命令的参数。

xargs 的作用一般等同于大多数 Unix shell 中的反引号，但是更加灵活易用，并可以正确处理输入中有空格等特殊字符的情况，因此对于经常产生大量输出的命令如 find、locate 和 grep 来说非常有用。

```
xargs [-0epn] command
```

- 如果输入的 stdin 中包含特殊字符（例如‘，\或空格符等），-0 参数可以将它们转换为一般字符。
- -e 参数代表（end of file），其后可以接一个字符串，xargs 分析到 eof 时就会停止；
- -p 参数可以指定执行每个命令的参数时询问用户；
- -n 参数指定后面的命令执行的次数

xargs 可以将参数列表转换成小块分段传递给其他命令，以避免参数列表过长的问题。如果 xargs 后面没有任何命令，默认是以 echo 来进行输出。

xargs 可以弥补不支持管道的命令，而且 xargs 可以供这些命令引用 stdin。

47.8.1 find

例如，下面的命令可能因为 path 目录下文件过多而“参数列表过长”导致无法执行。

```
rm `find /path -type f`
```

这里，xargs 命令可以将 find 产生的长串文件列表拆散成多个子串，然后对每个子串调用 rm 命令。

```
find /path -type f -print0 | xargs -0 rm
```

- -print0 表示输出以 null 分隔（-print 使用换行）;
- -0 表示输入以 null 分隔。

使用 xargs 命令可以提高 find 命令的效率，下面的改进前的命令会对每个文件调用 rm 命令。

```
find /path -type f -exec rm '{}' \;
```

使用较新版的 find 命令也可以得到和 xargs 命令同样的效果:

```
find /path -type f -exec rm '{}' +
```

47.8.2 finger

下面的示例使用 finger 命令将/etc/passwd 中的前两个帐号内容显示到屏幕。

```
# cut -d ':' -f 1 /etc/passwd | head -n 2 | xargs finger
Login: root             Name: root
Directory: /root                    Shell: /bin/bash
Last login Tue Jan  6 16:59 (CST) on pts/0
No mail.
No Plan.

Login: bin              Name: bin
Directory: /bin                     Shell: /sbin/nologin
Never logged in.
No mail.
No Plan.
```

如果需要指定上述分析操作到达 lp 就结束，可以使用下面的命令:

```
# cut -d ':' -f 1 /etc/passwd | xargs -e'lp' finger
```

Chapter 48

Text Process

48.1 awk

48.2 banner

48.3 basename

当向 basename 传递一个路径名时，它会删除任何前缀，直到最后一个斜线（'/'）字符，然后返回结果，因此 basename 主要用于 shell 脚本中。

单一 UNIX 规范中的 basename 格式如下：

```
basename string [suffix]
```

- string - 路径名
- suffix - 若指定 suffix，则 basename 也将删除此后缀。

下面的示例说明了 basename 的用法。

```
$ basename /home/theqiong/base.wiki
base.wiki
$ basename /home/theqiong/base.wiki .wiki
base
```

basename 只接受一个操作数，因此在 Shell 脚本的内层循环使用它可能会影响性能。

```
while read file; do
  basename "$file" ;
done < some-input
```

在上述的示例中，每一个输入行都会调用一个单独的进程，因此通常用于壳层替代。

```
echo "${file##*/}";
```

48.4 comm

48.5 csplit

48.6 cut

48.7 diff

diff 程序最早是由贝尔实验室研发的，由 Douglas McIlroy 在 1974 年开发的第五版成为第一个正式版本。

diff 可以在行级上比较两个文件之间的不同，通常被用来比较同一个文件在不同版本间的差异。

diff 可以产生一个扩展名为.diff 或.patch 的文件，这个文件可以被另一个工具 patch 使用。

```
$ diff [-bBi] from-file to-file
$ diff [-bBi] from-file -
$ diff[-bBi] - to-file
```

- -b 表示忽略一行当中仅有多个空白的区别（例如“Hello world”和“Hello world”将被视作是相同的）;
- -B 表示忽略空白行的区别;
- -i 表示忽略大小写的不同。

diff 也可以用于比较整个目录，以及不同目录下的同名文件等。

使用 diff 程序来产生 patch 文件的语法如下:

```
$ diff -Naur oldfile newfile > diff.patch
```

一般来说，使用 diff 产生的补丁文件的扩展名为 patch。

```
$ patch -pN < patch_file 更新
$ patch -R -pN < patch_file 还原
```

- -p 参数后面的 N 表示取消的目录层数;
- -R 表示将新的文件还原为旧版本。

如果补丁文件和要更新的文件位于同一目录下，可以使用-p0 参数。

48.8 cmp

cmp 对文件进行“字节”层级的比较。

```
$ cmp [-s] file1 file2
```

cmp 默认仅会输出第一个发现的不同点，使用-s 参数可以输出所有的不同点的字节位置。

48.9 patch

1985 年 5 月，Larry Wall 在 mod.sources（为 comp.sources.unix 的前身）上发布了 patch 程序的第一个稳定版本

patch 程序可以使用由 diff（或由 CVS、Subversion 和 Git 等）产生的“patch file”的文本文件来进行文件更新。

要使用一个 patch 文件, 可以在 shell 直接执行以下命令：

```
$ patch < patch.diff
```

48.10 dirname

当给予 dirname 一个路径名时，它会删除最后一个斜线（‘/’）后的任何后缀，并返回结果，因此 dirname 命令主要用于 Shell 脚本中。

单一 UNIX 规范中的 dirname 定义如下：

```
dirname NAME
```

```
$ dirname /usr/home/theqiong/dirname.wiki
/usr/home/theqiong
```

dirname 只接受一个操作数，在 Shell 脚本内循环中使用可能会降低性能。

```
while read file; do
    dirname "$file"
done < some-input
```

上面的例子会导致每行输入调用一个单独的进程，因此通常会用 Shell 替换来代替。

```
echo "${file%/*}";
```

48.11 ed

48.12 ex

48.13 fmt

48.14 fold

48.15 head

48.16 iconv

48.17 join

48.18 less

48.19 more

48.20 nl

48.21 paste

48.22 sort

48.23 spell

48.24 strings

48.25 tail

48.26 tr

48.27 uniq

48.28 wc

48.29 xargs

Bibliography

[1] Wikipedia. Shell, . URL http://zh.wikipedia.org/zh-cn/%E6%AE%BC%E5%B1%A4.

[2] Wikipedia. vi, . URL http://zh.wikipedia.org/zh-cn/Vi.

[3] Wikipedia. vim, . URL http://zh.wikipedia.org/zh-cn/Vim.

[4] 吉庆. Vim 使用笔记, 06 2012. URL http://www.cnblogs.com/jiqingwu/archive/2012/06/14/vim_notes.html.

Part VII

Regex

Chapter 49

Introduction

49.1 Overview

正则表达式使用单个字符串来描述、匹配一系列符合某个句法规则的字符串，从而可以被用来以行为单位检索、替换以及删除符合指定模式的文本。

正则表达式的概念最初是由 Unix 中的工具软件（例如 sed 和 grep）引入的，基本上可以认为正则表达式就是一种“表示法”，现在许多程序设计语言都支持利用正则表达式进行字符串操作。例如，Perl 就内置了一个功能强大的正则表达式引擎（PCRE），从而可以用来过滤广告邮件等。

广告邮件几乎都有一定的标题或内容，因此可以使用正则表达式来匹配邮件的标题和内容，现在的 sendmail 和 postfix 等邮件服务器程序以及支持邮件服务器的相关分析程序都支持正则表达式。

另外，很多 Web 服务器也支持使用正则表达式来制定 rewrite 规则等。

49.2 History

最初的正则表达式出现于理论计算机科学的自动控制理论和形式化语言理论中。在这些领域中有对计算（自动控制）的模型和对形式化语言描述与分类的研究。

1940 年，Warren McCulloch 与 Walter Pitts 将神经系统中的神经元描述成小而简单的自动控制元。

1950 年代，数学家斯蒂芬·科尔·克莱尼利用“正则集合”的数学符号来描述此模型。

肯·汤普逊将“正则集合”的符号系统引入编辑器 QED，然后是 Unix 上的编辑器 ed，并最终引入 grep，从此正则表达式被广泛地使用于各种 Unix 或者类似 Unix 的工具（例如 Perl）。

- vi、vim、grep、awk、sed 等都支持正则表达式；

- cp、ls 等不支持正则表达式，只能使用 bash 本身的通配符。

正则表达式和通配符是完全不同的概念，通配符是 bash 接口的一个功能，而正则表达式则是一种字符串处理的表示方式。

- 通配符中的 * 表示零到无限个字符；
- 正则表达式中的 * 表示重复 0 到无限个匹配字符。

例如，为了列出当前目录下任何以 a 开头的文件或目录，可以使用通配符和正则表达式分别实现如下：

```
$ ls -l a*
$ ls | grep -n '^a.*'
```

如果需要找出/etc 目录下的连接文件，可以使用 ls 命令配合正则表达式实现如下：

```
$ ls -l /etc/ | grep '^l'
```

操作系统的字符编码也会影响正则表达式的使用，例如在英文环境中使用 zh_CN 或 C 时产生不同的输出结果.

- LANG=C
  0 1 2 3 4···A B C D···Z a b c d···z
- LANG=zh_CN
  0 1 2 3 4···a A b B c C d D···z Z

Table 49.1: 正则表达式中的特殊符号

特殊符号	备注
[:alnum:]	英文大小写字母及数字，即 0 ～ 9，A ～ Z，a ～ z
[:alpha:]	任何英文大小写字母，即 A ～ Z，a ～ z
[:blank:]	空格键和 tab 键
[:cntrl:]	控制键，包括 CR、LF、Tab、Del 等
[:digit:]	数字，即 0 ～ 9
[:graph:]	除空格符（空格键和 tab 键）外的所有其他按键
[:lower:]	小写字母，即 a ～ z
[:print:]	任何可被打印的字符
[:punct:]	标点符号（punctuation symbol），即" ' ？ ！ ； ： # $
[:upper:]	大写字母，即 A ～ Z
[:space:]	任何可以产生空白的字符，包括空格键、Tab、CR 等
[:xdigit:]	十六进制的数字，其中包括 0 ～ 9、A ～ F，a ～ f 的数字和字符

在实际应用中，正则表达式的字符串表示方式按照不同的严谨度又可以分为基础正则表达式和扩展正则表达式。

扩展正则表达式除了字符串处理之外，还支持使用特殊的“(”与“|”等协助处理成组的字符串。

Perl 的正则表达式源自于 Henry Spencer 写的 regex，并且已经演化成了 pcre 库（Perl 兼容正则表达式，Perl Compatible Regular Expressions），并且在很多现代工具中可以使用。

49.3 Theory

正则表达式可以用形式化语言理论的方式来表达，并且由常量和算子组成，分别指示字符串的集合和在这些集合上的运算。

给定有限字母表 Σ 定义了下列常量：

- （“空集”）$\varnothing$ 指示集合 $\varnothing$
- （“空串”）ε 指示集合 $\{\varepsilon\}$
- （“文字字符”）在 Σ 中的 a 指示集合 {a}

并且定义了下列运算：

- （“串接”）RS 指示集合 $\alpha\beta|\alpha \in R, \beta \in S$。例如：”$ab$”, ”$c$””$d$”, ”$ef$” = ”$abd$”, ”$abef$”, ”$cd$”, ”$cef$”。
- （“选择”）R|S 指示 R 和 S 的并集。例如：”ab”, ”c”|”ab”, ”d”, ”ef” = ”ab”, ”c”, ”d”, ”ef”
- （“Kleene 星号”）R* 指示包含 ε 并且闭合在字符串串接下的 R 的最小超集。这是可以通过 R 中的零或多个字符串的串接得到所有字符串的集合。例如，”ab”, ”c”* = ε, ”ab”, ”c”, ”$abab$”, ”abc”, ”cab”, ”cc”, ”$ababab$”, ...。

上述常量和算子形成了克莱尼代数，也可以选择使用符号 $\cup$, + 或 $\vee$ 替代竖杠。

为了避免括号，假定 Kleene 星号有最高优先级，接着是串接，接着是并集。如果没有歧义则可以省略括号，例如，$(ab)c$ 可以写为 abc 而 $a|(b(c*))$ 可以写为 $a|bc*$。

- $a|b*$ 指示 $\varepsilon, a, b, bb, bbb, ...$。
- $(a|b)*$ 指示由包括空串、任意数目个 a 或 b 字符组成的所有字符串的集合。
- $ab*(c|\varepsilon)$ 指示开始于一个 a 接着零或多个 b 和最终可选的一个 c 的字符串的集合。

正则表达式的定义非常精简，避免多余的量词? 和 +，它们可以被表达为：$a+ = aa*$ 和 $a? = (a|\varepsilon)$。有时增加补算子 ， R 指示在 Σ^* 上的不在 R 中的所有字符串的集合。

补算子是多余的，因为它使用其他算子来表达（尽管计算这种表示的过程是复杂的，而结果可能以指数增大）。

这种意义上的正则表达式可以表达正则语言，精确的是可被有限状态自动机接受的语言类。但是在简洁性上有重要区别。某类正则语言只能用大小指数增长的自动机来描述，而要求的正则表达式的长度只线性的增长。

正则表达式对应于乔姆斯基层级的类型-3 文法，另外在正则表达式和不导致这种大小上的爆炸的非确定有限状态自动机（NFA）之间有简单的映射，为此 NFA 经常被用作正则

表达式的替代表示。

我们还要在这种形式化中研究表达力。如下面例子所展示的，不同的正则表达式可以表达同样的语言，这种形式化中存在着冗余。

有可能对两个给定正则表达式写一个算法来判定它们所描述的语言是否本质上相等，简约每个表达式到极小确定有限自动机，确定它们是否同构（等价）。

这种冗余可以消减到什么程度? 我们可以找到仍有完全表达力的正则表达式的有趣的子集吗? Kleene 星号和并集明显是需要的，但是我们或许可以限制它们的使用。这提出了一个令人惊奇的困难问题。

因为正则表达式如此简单，没有办法在语法上把它重写成某种规范形式。过去公理化的缺乏导致了星号高度问题，最近 Dexter Kozen 用克莱尼代数公理化了正则表达式。

很多现实世界的“正则表达式”引擎实现了不能用正则表达式代数表达的特征。

49.4 Syntax

一个正则表达式通常被称为一个模式（pattern），也就是用来描述或者匹配一系列符合某个句法规则的字符串。例如：Handel、Händel 和 Haendel 这三个字符串，都可以由“H(a|ä|ae)ndel”这个模式来描述。

大部分正则表达式的形式都包括选择、数量限定及匹配等结构。

- 选择 | 竖直分隔符代表选择。例如“gray|grey”可以匹配 grey 或 gray。
- 数量限定
  某个字符后的数量限定符用来限定前面这个字符允许出现的个数。最常见的数量限定符包括“+”、“?”和“*”(不加数量限定则代表出现一次且仅出现一次)：
  + 加号代表前面的字符必须至少出现一次。(1 次、或多次)。例如，“goo+gle”可以匹配 google、gooogle、goooogle 等;
  ? 问号代表前面的字符最多只可以出现一次。(0 次、或 1 次)。例如，“colou?r”可以匹配 color 或者 colour。
  * 星号代表前面的字符可以不出现，也可以出现一次或者多次。(0 次、或 1 次、或多次)。例如，“0*42”可以匹配 42、042、0042、00042 等。
- 匹配
  圆括号可以用来定义操作符的范围和优先度。例如，“gr(a|e)y”等价于“gray|grey”，“(grand)?father”匹配 father 和 grandfather。

上述这些构造子都可以自由组合，因此“H(ae?|ä)ndel”和“H(a|ae|ä)ndel”是相同的，不过精确的语法可能因不同的工具或程序而异。

49.5 Style

正则表达式有多种不同的风格。

下表是在 PCRE 中元字符及其在正则表达式上下文中的行为的一个完整列表，适用于 Perl 或者 Python 编程语言，而且 grep 或者 egrep 的正则表达式文法是 PCRE 的子集。

Table 49.2: PCRE

字符	备注
\	将下一个字符标记为一个特殊字符、或一个原义字符、或一个向后引用、或一个八进制转义符。 例如，“n”匹配字符“n”，“\n”匹配一个换行符，序列“\\”匹配“\”而“\(”则匹配“(”。 例如，`grep -n \' file`
^	匹配输入字符串的开始位置。 如果设置了 RegExp 对象的 Multiline 属性，^ 也匹配“\n”或“\r”之后的位置。 例如：`grep -n '^#' file`
$	匹配输入字符串的结束位置。 如果设置了 RegExp 对象的 Multiline 属性，$ 也匹配“\n”或“\r”之前的位置。 例如：`grep -n '!$' file`
*	匹配前面的子表达式零次或多次。 例如，zo* 能匹配“z”、“zo”以及“zoo”，* 等价于 {0,}，而任意字符则为“.*”。 例如：`grep -n 'zo*' file`

+	匹配前面的子表达式一次或多次，+ 等价于 {1,}。
	例如，“zo+”能匹配“zo”以及“zoo”，但不能匹配“z”。
	`grep -E -n 'go+d' test.txt`
	`egrep -n 'go+d' test.txt`

?	匹配前面的子表达式零次或一次，? 等价于 {0,1}。
	例如，“do(es)?”可以匹配“do”或“does”中的“do”。
	`grep -E -n 'go?d' test.txt`
	`egrep -n 'go?d' test.txt`

{n}	n 是一个非负整数。匹配确定的 n 次。例如，“o{2}”不能匹配“Bob”中的“o”，但是能匹配“food”中的两个 o。
{n,}	n 是一个非负整数。至少匹配 n 次。例如，“o{2,}”不能匹配“Bob”中的“o”，但能匹配“fooooood”中的所有 o。“o{1,}”等价于“o+”。“o{0,}”则等价于“o*”。
{n,m}	m 和 n 均为非负整数（其中 n<=m），最少匹配 n 次且最多匹配 m 次。 例如，“o{1,3}”将匹配“fooooood”中的前三个 o。“o{0,1}”等价于“o?”，注意在逗号和两个数之间不能有空格。
\{n,m\}	m 和 n 均为非负整数（其中 n<=m），连续 n 到 m 个匹配字符。 例如：\{n\} 代表连续 n 个匹配字符，\{n,\} 代表连续 n 个以上的匹配字符。 例如：`grep -n 'go\{2,3\}g' file`
?	当该字符紧跟在任何一个其他限制符（*，+，?，{n}，{n,}，{n,m}）后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串“oooo”，“o+?”将匹配单个“o”，而“o+”将匹配所有“o”。
.	匹配除“\n”之外的任何单个字符（包括空格）。 要匹配包括“\n”在内的任何字符，可以使用像“(.|\n)”的模式。 例如：`grep -n 'e.e' file`

字符	备注
(pattern)	匹配 pattern 并获取匹配组的子字符串，而且该子字符串用于向后引用，所获取的匹配可以从产生的 Matches 集合得到，在 VBScript 中使用 SubMatches 集合，在 JScript 中则使用 $0…$9 属性。要匹配圆括号字符，使用“\(”或“\)”。 例如，为了匹配 glad 和 good，可以使用 `egrep -n 'g(al|oo)d' test.txt`
(pattern)+	使用 + 配合组匹配可以进行多个重复的组的判别。例如： `echo 'AxyzxyzxyzC' | egrep 'A(xyz)+C'`
(?:pattern)	匹配 pattern 但不获取匹配的子字符串，也就是说这是一个非获取匹配，不存储匹配的子字符串用于向后引用。这在使用或字符“(|)”来组合一个模式的各个部分是很有用。例如“industr(?:y|ies)”就是一个比“industry|industries”更简略的表达式。
(?=pattern)	正向肯定预查，在任何匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，“Windows(?=95|98|NT|2000)”能匹配“Windows2000”中的“Windows”，但不能匹配“Windows3.1”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。
(?!pattern)	正向否定预查，在任何不匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如“Windows(?!95|98|NT|2000)”能匹配“Windows3.1”中的“Windows”，但不能匹配“Windows2000”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始 (?<=pattern)& 反向肯定预查，与正向肯定预查类似，只是方向相反。例如，“(?<=95|98|NT|2000)Windows”能匹配“2000Windows”中的“Windows”，但不能匹配“3.1Windows”中的“Windows”。
(?<!pattern)	反向否定预查，与正向否定预查类似，只是方向相反。例如“(?<!95|98|NT|2000)Windows”能匹配“3.1Windows”中的“Windows”，但不能匹配“2000Windows”中的“Windows”。
x|y	匹配 x 或 y，因此 | 表示使用或（or）的方式找出匹配值。 例如，“z|food”能匹配“z”或“food”。“(z|f)ood”则匹配“zood”或“food”。 `egrep -n 'gd|good|dog' test.txt`
[xyz]	字符集合，匹配所包含的任意一个字符。例如“[abc]”可以匹配“ask”中的“a”。 特殊字符仅有反斜线\保持特殊含义（用于转义字符），其它特殊字符如星号、加号、各种括号等均作为普通字符。 如果脱字符 ^ 出现在首位则表示负值字符集合，如果出现在字符串中间就仅作为普通字符。 如果连字符（-）出现在字符串中间表示字符范围描述，如果出现在首位则仅作为普通字符。 例如：`grep -n 'g[ld]' file`
[^xyz]	排除型（negate）字符集合，匹配未列出的任意字符，因此又称为“反向选择”。 例如，“[^abc]”可以匹配“plain”中的“plin”。 例如：`grep -n 'oo[^t]' file`
[a-z]	字符范围，匹配指定范围内的任意字符，减号（-）代表两个字符之间的任意 ASCII 字符。 例如，“[a-z]”可以匹配“a”到“z”范围内的任意小写字母字符。 例如：`grep -n '[0-9a-zA-Z]' file`
[^a-z]	排除型的字符范围。匹配任何不在指定范围内的任意字符。例如，“[^a-z]”可以匹配任何不在“a”到“z”范围内的任意字符。
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如，“er\b”可以匹配“never”中的“er”，但不能匹配“verb”中的“er”。
\B	匹配非单词边界。“er\B”能匹配“verb”中的“er”，但不能匹配“never”中的“er”。
\cx	匹配由 x 指明的控制字符。例如，\cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的“c”字符。
\d	匹配一个数字字符。等价于 [0-9]。

字符	备注
\D	匹配一个非数字字符。等价于 [^0-9]。
\f	匹配一个换页符。等价于\x0c 和\cL。
\n	匹配一个换行符。等价于\x0a 和\cJ。
\r	匹配一个回车符。等价于\x0d 和\cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于\x09 和\cI。
\v	匹配一个垂直制表符。等价于\x0b 和\cK。
\w	匹配包括下划线的任何单词字符。等价于“[A-Za-z0-9_]”。
\W	匹配任何非单词字符。等价于“[^A-Za-z0-9_]”。
\xn	匹配 n，其中 n 为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，“\x41”匹配“A”。“\x041”则等价于“\x04&1”。正则表达式中可以使用 ASCII 编码。.
\num	向后引用（back-reference）一个子字符串（substring），该子字符串与正则表达式的第 num 个用括号围起来的子表达式（subexpression）匹配。其中 num 是从 1 开始的正整数，其上限可能是 99。例如：“(.)\1”匹配两个连续的相同字符。
\n	标识一个八进制转义值或一个向后引用。如果\n 之前至少 n 个获取的子表达式，则 n 为向后引用。否则，如果 n 为八进制数字（0-7），则 n 为一个八进制转义值。
\nm	标识一个八进制转义值或一个向后引用。如果\nm 之前至少有 nm 个获得子表达式，则 nm 为向后引用。如果\nm 之前至少有 n 个获取，则 n 为一个后跟文字 m 的向后引用。如果前面的条件都不满足，若 n 和 m 均为八进制数字（0-7），则\nm 将匹配八进制转义值 nm。
\nml	如果 n 为八进制数字（0-3），且 m 和 l 均为八进制数字（0-7），则匹配八进制转义值 nml。
\un	匹配 n，其中 n 是一个用四个十六进制数字表示的 Unicode 字符。例如，\u00A9 匹配版权符号（©）。

在基础正则表达式之外，为了简化命令操作，可以使用范围更广的扩展型正则表达式。例如，为了去除空白行与行首为 # 的行，可以使用命令如下：

```
grep -v '^$' test.txt | grep -v '^#'
```

如果使用扩展性正则表达式，可以通过组功能“|”将上述命令简化为：

```
egrep -v '^$|^#' test.txt
```

默认情况下，grep 仅支持基础正则表达式，通过-E 参数或者直接调用 egrep 可以使用扩展型正则表达式。

49.6 Priority

下面的示例中以 PHP 的语法来验证字符串是否只含数字与英文，字符串长度并在 4 16 个字符之间。

```
<?php
```

优先权	符号
最高	\
高	( )、(?:)、(?=)、[ ]
中	*、+、?、{n}、{n,}、{m,n}
低	^、$、中介字符
最低	|

```
$str = 'a1234';
if (preg_match("/^[a-zA-Z0-9]{4,16}$/", $str)) {
    echo "验证成功";
} else {
    echo "验证失败";
}
?>
```

对于同样的需求，可以使用 Perl 实现如下：

```
print $str = "a1234" =~ m:^[a-zA-Z0-9]{4,16}$: ? "COMFIRM" : "FAILED";
```

下面的示例中以 PHP 的语法来验证台湾身份证号。

```
<?php
$str = 'a1234';
if (preg_match("/^[A-Za-z]\d{9}$/", $str)) {
    echo "验证成功";
} else {
    echo "验证失败";
}
?>
```

类似的，可以使用 Perl 来验证台湾身份证号。

```
print $str = "a1234" =~ m"^\w[12]\d{8}$" ? "COMFIRM" : "INVALID";
```

正则表达式中的! 和 ^ 是不同的，其中! 只是作为普通字符使用。例如，[^a-z] 表示反向选择，而 [!a-z] 表示普通字符。

Chapter 50

sed

sed（stream editor）用来把文档或字符串里面的文字经过一系列编辑命令转换为另一种格式输出，通常用来匹配一个或多个正则表达式的文本进行处理。

sed 也可以是一个用于分析标准输入的管道命令，而且可以对数据执行替换、删除、新增和选择特定行等操作。

```
sed [-nefr] 'op'
```

sed 执行的操作使用单引号括起来，例如删除文件中的单行、多行以及删除到末尾可以执行:

```
$ cat > test.txt
1
2
3
4
5
$ cat test.txt | nl | sed '2d'
1 1
3 3
4 4
5 5
$ cat test.txt | nl | sed '2,5d'
1 1
$ cat test.txt | nl | sed '3,$d'
1 1
```

```
2 2
```

- -n 静默模式，只将经过 sed 处理的行（或操作）输出到屏幕；
- -e 表示在 e 后面的文字是正则表达式；
- -f 按照指定的 sed 脚本里面的命令来进行转换；
- -r 使 sed 支持扩展正则表达式；
- -i 表示将转换结果直接插入文件中，从而直接修改读取的文件内容，而不是向屏幕输出。

  若不用-i，一般 sed 命令不会改变原文档里的内容，而只会输出到命令行。当然命令行输出的内容也可以用“>”转存到另外一个文件里。

sed 执行的操作可以使用 [n1[,n2]]function 来说明，其中 n1、n2 一般代表选择进行操作的次数。例如，如果需要在 10 到 20 行之间进行操作，可以使用“`10,20[op]`”。

sed 可以执行的操作使用如下参数说明：

- a 代表新增，后接的字符串将会在当前行的下一行出现；
- c 代表替换，后接的字符串可以替换 `n1,n2` 之间的行，例如：

```
$ cat test.txt | nl | sed '2,3c newline'
     1 1
newline
     4 4
     5 5
```

- d 代表删除，后面通常不带任何参数。

  例如，/pattern/d 用于删除所有匹配模式的行。
- i 代表插入，后接的字符串将会在当前行的上一行出现；
- p 代表打印，用于输出某个选中的数据，通常和参数 `sed -n` 一起使用。

  例如，/pattern/p 用于输出所有匹配模式的行。
- s 代表替换，可以直接进行替换，或者搭配正则表达式，其用法类似于 VIM。

  例如，`sed 's/pattern/string/'` 用于将匹配模式的行转换成指定的 string。

  s 命令预设只替换每行匹配的第一串文字，也就是说，若每行里有多个匹配该模式的字符串，后面的将不会被 s 转换，可以用 g 命令来替换所有匹配的文字。

  `sed 's/pattern/string/g'` 用于将所有匹配模式的字符串替换为指定的 string。例如，可以使用 sed 命令来替换网络接口中的关键字。

```
$ ifconfig | grep 'inet' | nl | sed 's/^.*inet//g'
 127.0.0.1  netmask 255.0.0.0
 192.168.1.16  netmask 255.255.255.0  broadcast 192.168.1.255
$ ifconfig | grep 'inet' | nl | sed ''s/^.*inet//g | sed 's/net.*$//g'
 127.0.0.1
```

```
192.168.1.16
```

为了查找当前系统中的 MAN 目录，可以使用配合如下的 sed 命令。

```
cat /etc/man_db.conf | grep 'MAN' | sed 's/#.*$//g' | sed '/^$/d'
MANDATORY_MANPATH /usr/man
MANDATORY_MANPATH /usr/share/man
MANDATORY_MANPATH /usr/local/share/man
MANPATH_MAP /bin /usr/share/man
MANPATH_MAP /usr/bin /usr/share/man
MANPATH_MAP /sbin /usr/share/man
MANPATH_MAP /usr/sbin /usr/share/man
MANPATH_MAP /usr/local/bin /usr/local/man
MANPATH_MAP /usr/local/bin /usr/local/share/man
MANPATH_MAP /usr/local/sbin /usr/local/man
MANPATH_MAP /usr/local/sbin /usr/local/share/man
MANPATH_MAP /usr/X11R6/bin /usr/X11R6/man
MANPATH_MAP /usr/bin/X11 /usr/X11R6/man
MANPATH_MAP /usr/games /usr/share/man
MANPATH_MAP /opt/bin /opt/man
MANPATH_MAP /opt/sbin /opt/man
MANDB_MAP /usr/man /var/cache/man/fsstnd
MANDB_MAP /usr/share/man /var/cache/man
MANDB_MAP /usr/local/man /var/cache/man/oldlocal
MANDB_MAP /usr/local/share/man /var/cache/man/local
MANDB_MAP /usr/X11R6/man /var/cache/man/X11R6
MANDB_MAP /opt/man /var/cache/man/opt
```

-a 和-i 都可以用于插入新行，不过位置不同，前者追加在当前行的下一行，后者则是当前行的上一行。

```
$ cat test.txt | nl | sed '2a hello'
     1 1
     2 2
hello
     3 3
     4 4
     5 5
```

```
$ cat test.txt | nl | sed '2i hello'
     1 1
hello
     2 2
     3 3
     4 4
     5 5
```

为了读取指定范围的行，除了可以使用 head 和 tail 命令来实现，也可以使用 sed 命令。

```
$ cat test.txt | nl | head -n 5 | tail -n 3
     3 3
     4 4
     5 5
$ cat test.txt | nl | sed -n '3,5p'
     3 3
     4 4
     5 5
```

- `sed G` 在每一行后面增加一空行
- `sed 'G;G'` 在每一行后面增加两行空行

使用 sed 的-i 参数可以直接修改文件的内容，不必使用管道命令或数据流重定向。

```
$ sed -i 's/\.$/test/g' test.txt
```

sed 直接修改文件的功能可以配合 VIM 来使用，从而可以实现对大文件进行直接修改和替换等操作。

Chapter 51

awk

51.1 Overview

最初在实现 awk 时，其目的就是用于文本处理，并且这种语言的基础是只要在输入数据中有模式匹配，就执行一系列指令。

具体来说，awk 的设计思想来源于 SNOBOL4、sed、有效性语言、yacc、lex 以及 C 语言中的一些优秀的思想，因此 awk 在在很多方面都类似于 shell 编程语言，尽管 awk 具有完全属于其本身的语法。

awk 可以用于正则表达式的匹配、样式装入、流控制、数学运算符、进程控制语句以及内置的变量和函数，因此可以说 awk 具备了一个完整的语言所应具有的所有特性。

awk 允许用户创建程序来读取输入文件、为数据排序、处理数据、对输入执行计算以及生成报表等，gawk 是 awk 的 GNU 版本。

下面是 awk 的 hello world 程序，这里无需写出 exit 语句，因为唯一的模式是 BEGIN。

```
BEGIN{print "Hello, world!"}
```

awk 扫描文件中的每一行，查找与命令行中所给定内容相匹配的模式。如果发现匹配内容，则进行下一个编程步骤。如果找不到匹配内容，则继续处理下一行。

相比只能按行处理数据的 sed，awk 工具及编程语言可以将一行分割为多个字段来进行处理，而且 awk 支持使用 if 和 for 等进行条件判断和循环处理。

51.2 Structure

作为一种处理文本文件的语言和工具，awk 将文件作为记录序列处理。

在一般情况下，文件内容的每行都是一个记录。每行内容都会被分割成一系列的域，因此可以认为一行的第一个词为第一个域，第二个词为第二个，以此类推。

awk 程序是由一些处理特定模式的语句块构成的，awk 一次可以读取一个输入行。

对每个输入行，awk 解释器会判断它是否符合程序中出现的各个模式，并执行符合的模式所对应的动作。

awk 程序由一系列模式–动作对组成，而且 awk 可以处理后面的文件，也可以读取来自前面的命令的输出。

```
pattern {action}
```

- pattern 表示 awk 在数据中查找的内容；
- action 是在找到匹配内容时所执行的一系列命令。

awk 主要处理每一行的字段内的数据，默认的字段的分隔符为空格符或 Tab 键，下面说明了 awk 命令的一般形式：

```
awk 'condition1{action1} condition2{action2} ... ' filename
```

其中，输入行被分成了一些记录，记录默认由换行符分割，因此输入会按照行进行分割。例如，下面的示例说明如何使用 awk 来获取最近登录的帐号和 IP，并且帐号和 IP 之间以 Tab 空格隔开。

```
# last -n 5 | awk '{print $1 "\t" $3}'
```

awk 程序使用给定的条件一个个的测试每条记录，并执行测试通过的条件所对应的 action，而且 pattern 和 action 都可以省略。

- 无 pattern 默认匹配全部的记录；
- 无 action 则是打印原始记录。

具体来说，awk 命令的处理过程可以表述如下：

1. 读入第一行并将第一行的数据填入 $0、$1、$2、$3 等变量中；
2. 根据条件类型的限制来判断是否需要执行后面的操作；
3. 执行所有的动作与条件类型；
4. 如果还有后续的“行”的数据，则重复上述 1 ～ 3 的步骤直到所有的数据都读取完毕。

除了简单的 awk 表达式之外，pattern 可以是 BEGIN 或 END，这两种条件对应的 action 分别是读取所有的记录之前和之后。

pattern1 和 pattern2 等条件表示符合条件 pattern1 和 pattern2 等的记录及其之间的部分。

除了一般的 C 语言风格的算术和逻辑运算符外，awk 允许使用运算符～测试正则表达式是否可以与一字符串匹配。

作为语法糖，没有～运算符的正则表达式会被用来对当前记录进行测试，相当于`/regexp/ ~ $0`。

51.3 Command

awk 命令就是以 action 指代的语句，可以包括函数调用、变量赋值、计算及/或各项的组合。

标准 awk 提供了许多内建函数，而且其部分实现可能还提供了更多的内建函数，同时 awk 的部分实现支持动态连接库来支持更多的函数。

awk 后续的所有动作都是以单引号（'）括起来的，而且内部的非变量文字部分可以使用双引号（"）进行定义。

51.3.1 print

print 命令用于输出文本，其输出的文本总是以" 输出记录分隔符"（Output record separator, ORS）分割的，其默认值为换行符。

- `print` 会输出当前记录的内容。例如，awk 中的记录会被分割成“域”，它们可以被分别显示或使用；
- `print $1` 显示当前记录的第 1 个域；
- `print $1, $3` 显示当前记录的第 1 和第 3 个域，并以预定义的输出域分隔符（Output field separator, OFS）分隔，其默认值为一个空格符。

  域的符号（$X）类似于某些语言中的变量（例如 PHP 和 perl），但是在 awk 中的于指代的是当前记录的域。

  在使用 printf 命令进行格式化输出时，务必加上\n 才能进行分行。

  另外，`$0` 指代整个记录。事实上，命令 `print` 和 `print $0` 的效果是相同的。

  print 命令也可以用于显示变量、计算、函数调用的结果等。

```
print 3+2
print foobar(3)
print foobar(variable)
print sin(3-2)
```

print 的输出可以重定向到文件或管道。

```
print "expression" > "file name"
print "expression" | "command"
```

例如，为了对输入中的单词进行计数，然后输出行数、单词数和字符数（类似 wc），可以使用如下的语法：

```
{
    w += NF
```

```
    c += length + 1
}
END { print NR, w, c }
```

这里没有提供模式，输入的全部行都可以匹配该模式，因此对每行都会执行预定操作。

注意，在上面的示例中，`w+=NF` 的含义等同于 `w = w + NF`。

如果需要计算最后一个单词的和，可以使用如下的语法：

```
{ s += $NF }
END { print s + 0 }
```

s 是数值 $NF 的累加，而 $NF 则是每条记录中的最后一个域。NF 是当前行中域的数量，例如，$4 是第 4 个域的值，在这种情况下 $NF 就等于 $4。

事实上，$ 是一个具有最高优先级的一元运算符。

若一行没有域，则有 NF 为 0，而 $0 是整行，那么在这种情况下，要么是空串，要么只有空白符，因此其数值为 0。

文件结束时，END 模式得到了匹配，因此可以输出 s。

不过，由于可能没有输入行，此时 s 会没有值，从而导致没有输出，因此对其加 0 可以使 awk 在这种情况下对其赋值，从而得到一个数值。

上述这种方法是将字符串强制转化为数值的惯用法，反之，与空串连接则是将数值强制转换为字符串的方法（例如 `s ""`），因此如果程序输入为空文件，可以得到“0”作为输出（而不是一个空行）。

51.3.2 buildin

awk 的内建变量包括域变量（例如 $1, $2, $3 以及 $0 等），域变量给出了记录中域的内容。

另外，内建变量也包括一些其他变量，例如：

- NR：已输入记录的条数，也就是当前 awk 处理的行。
- NF：当前记录（$0）中域的个数，记录中最后一个域可以以 $NF 的方式引用。
- FILENAME：当前输入文件的文件名。
- FS：用于将输入记录分割成域的“域分隔符”（默认值为“空白字符”（即空格和制表符）），而且 FS 可以替换为其它字符来改变域分隔符。
- RS：当前的“记录分隔符”。默认状态下，输入的每行都被作为一个记录，因此默认记录分隔符是换行符。
- OFS：“输出域分隔符”，即分隔 print 命令的参数的符号，其默认值为空格。
- ORS：“输出记录分隔符”，即每个 print 命令之间的符号，其默认值为换行符。
- OFMT：“输出数字格式”（Format for numeric output），其默认值为”`%.6g`”。

51.3.3 variable

变量名可以是语言关键字外的只包含大小写拉丁字母、数字和下划线（“_”）的任意字。

与 bash、shell 的变量不同，awk 中的变量可以直接使用，不需要加上 $ 符号。

操作符“+ - * /”则分别代表加、减、乘和除，而且 awk 没有显式的字符串连接操作符。

- 若只是简单的将两个变量（或字符串常量）放在一起，则会将二者串接为一个字符串；
- 若二者间至少有一个是常量，则中间可以不加空格；
- 若二者均为变量，中间必须包括空格。

字符串常量是以双引号进行分隔，语句无需以分号结尾。

另外，注释是以“#”开头的。

在 awk 编写程序时，模式的默认行为是输出当前行。例如，下面的示例说明如何输出长度大于 80 字符的行。

```
length($0) > 80
```

例如，为了匹配输入行中的范围，可以使用下面的示例来说明。

```
$ yes Wikipedia | awk 'NR % 4 == 1, NR % 4 == 3 { printf "%6d  %s\n", NR, $0 }' | sed 7q
     1  Wikipedia
     2  Wikipedia
     3  Wikipedia
     5  Wikipedia
     6  Wikipedia
     7  Wikipedia
     9  Wikipedia
$
```

yes 命令重复输入其参数 (默认则是输出“y”)。在这里，我们让该命令输出“Wikipedia”。动作块则输出带行号的内容。

printf 函数可以模拟标准 C 中的 [[printf]] 函数，其效果与前述的 print 函数类似，其中符合模式的行是这样产生的：

NR 是记录的编号，也就是 AWK 正在处理行的行号（从 1 开始)。“%”是取余数操作符，因此 `NR % 4 == 1` 对第 1、5、9 等行为真。类似的，`NR % 4 == 3` 对 3、7、11 等行为真。

范围模式在其第一部分匹配（例如对第 1 行）之前为假，并在第二部分匹配（例如第 3 行）之前为真。

接下来再在第二次匹配上其第一部分（例如第 5 行）前为假。sed 命令则是用于截取其前 7 行输出，防止 yes 命令一直运行下去。

若 head 命令可用，这行命令的效果和 `head -n7` 相同。

- 若范围模式的第一部分永远为真（例如设定为“1”），可以用来使该范围从输入的最开始开始。
- 若范围模式的第二部分总是为假（例如“0”），则该范围的结束即为输入的结束。

下面的命令可以输出从符合正则表达式“^--cut here--$”开始的输入行，也即从只包含“--cut here--”的行开始，直到输入的结束。

```
/^--cut here--$/, 0
```

51.3.4 operator

awk 中的逻辑运算符包括 >、<、>=、<=、== 和!= 等。

- == 表示相等判断;
- = 用于变量赋值。

下面的示例说明如何使用 awk 来检索/etc/passwd 中的第三列小于 10 的数据，并且仅列出帐号与第三列。

```
# cat /etc/passwd | awk '{FS=":"} $3 < 10 {print $1 "\t " $3}'
root:x:0:0:root:/root:/bin/bash
bin  1
daemon  2
adm  3
lp  4
sync  5
shutdown 6
halt  7
mail  8
```

上述程序可以进一步改进，例如:

```
# cat /etc/passwd | awk 'BEGIN {FS=":"} $3 < 10 {print $1 "\t " $3}'
root  0
bin  1
daemon 2
adm  3
lp  4
sync  5
shutdown 6
halt  7
mail  8
```

51.3.5 function

函数是以与 C 语言类似的方式定义的，以关键字 function 开头，后面跟函数名称、参数列表和函数体。

```
# 示例函数
function add_three (number) {
  return number + 3
}
```

如果需要调用上面的函数，可以使用下面的调用格式：

```
print add_three(36)     # 输出39
```

函数可以拥有其私有变量，而且私有变量可以写在参数列表之后，因为这些值会在调用函数时被忽略。

通常可以在参数列表中参数和私有变量之间加入一些空格，用以区别“真正的”参数和私有变量。

函数声明中的函数名和括号间可以有任意空格，但在调用时二者必须紧邻。

下面的函数使用关联数组来计算词频，其中 BEGIN 块设定域分隔符为任意非字母字符。

```
BEGIN {
    FS="[^a-zA-Z]+"
}
{
     for(i=1; i<=NF; i)
          words[tolower($i)]++
}
END {
    for(i in words)
         print i, words[i]
}
```

值得注意的是，分隔符不仅可以是字符串，也可以是正则表达式，程序对每个输入行都执行相同的操作。

这里，我们对每个域都累加其小写形式出现的次数，最后在 END 块中，我们输出单词及其出现的次数。

下面的示例代码可以用于建立一个遍历关联数组中元素的循环，其中 i 会被设为对应的键。

```
for(i in words)
```

awk 的 for 语法和多数语言不同，和 Objective-C 中的 `for...in` 语法相似，都允许以简单的方式遍历数组，从而输出这些单词。

awk 可以从命令匹配模式，而且可以以多种不同形式出现。

```
$ cat grepinawk
pattern=$1
shift
awk '/'$pattern'/ { print FILENAME ":" $0 }' $*
$
```

awk 命令中的 $pattern 并没有为引号所保护，这里的模式可以检查输入行（$0）是否与之匹配，FILENAME 变量则包含了当前的文件名。

与 bash 相似，只需简单的将字符串并列即可使用 $0 来输出原始的输入行。

用户也可以以另外的方法来完成同样的任务，例如下面的脚本直接在 awk 中访问环境变量。

```
$ cat grepinawk
pattern=$1
shift
awk '$0 ~ ENVIRON["pattern"] { print FILENAME ":" $0 }' $*
$
```

这个脚本用到了数组 ENVIRON，其作用类似与 POSIX 标准中的 getenv (3) 函数。

在这个脚本中，首先建立了一个环境变量 pattern，其值为脚本的第一个参数，然后让 awk 在其余的参数所代表的文件内寻找该模式。

~ 是用于检查其两个操作数是否匹配的运算符，其逆则为`!~`。

注意，正则表达式也属于普通的字符串，可以储存于变量中。

下面的方法采用了在命令行对变量赋值的方法 (即在 awk 的参数中写入一个变量的值)。

```
$ cat grepinawk
pattern=$1
shift
awk '$0 ~ pattern { print FILENAME ":" $0 }' "pattern=$pattern" $*
$
```

如果使用纯 awk 来实现，无需 shell 的帮助，也无需知道太多关于 awk 脚本实现的细节 (而在命令行对变量赋值的方法可能与 awk 的实现相关)。

```
BEGIN {
    pattern = ARGV[1]
    for (i = 1; i < ARGC; i++) # 去除第一个参数
        ARGV[i] = ARGV[i + 1]
    ARGC—if (ARGC == 1) { # 模式是唯一参数，因此强制从标准输入读取
        ARGC = 2
        ARGV[1] = "-"
    }
}
$0 ~ pattern { print FILENAME ":" $0 }
```

BEGIN 块的作用不仅仅是提取出第一个参数，也防止第一个参数在 BEGIN 块结束后直接被解释为输入文件。

- ARGC 输入参数的数量永远是不小于 1 的，因为 ARGV[0] 是执行脚本的命令名，通常是”awk”。
- ARGV[ARGC] 永远是空串，其中的 if 块表明若没有指定输入文件，awk 会直接读取标准输入流（stdin）。

现在，程序中已经将 ARGC 置为了 2，因此 `awk 'prog'` 也可以工作。

如果该值为 1，则 awk 会认为没有文件需要读取而直接退出。同时，若需从标准输入读取数据，需要将文件名显式的指定为-。

与许多其他的程序语言相似，可以利用“shebang”语法构建自包含的 awk 脚本。例如，下面的程序 hello.awk 可以输出“Hello, world!”。

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!" }
```

`-f` 参数告诉 awk 将该文件作为 awk 的程序文件，然后即可运行该程序。

Part VIII

Script

Chapter 52

Overview

52.1 Shell Script

shell script 可以使用 shell 语法和命令（以及外部命令）来搭配正则表达式、管道命令和数据重定向等完成复杂的操作，而且 shell script 可以提供数组、循环、条件与逻辑判断等功能。

在实际应用中，shell script 可以用来查询登录文件、追踪流量、监控主机状态、查询硬件信息以及更新软件等。

一般情况下，Linux 系统的服务启动的接口位于/etc/init.d/目录下，而且 Linux 操作统的启动过程中也需要使用 shell script 来查找系统的相关设置等。另外，防火墙功能、启动加载项和数据处理等都可以使用 shell script 来实现。

用户可以使用 shell script 配合 bash shell 提供的工具来调用外部的函数库，并执行一定的数据处理操作。

52.2 Executation

在编写 shell script 时，需要了解如下事项，例如：

- 命令按照从上到下，从左到右的方向执行；
- 命令、参数间的多个空白都会被忽略；
- 空白行和 Tab 键空格都被看作是空格；
- 在读入 Enter 符号（CR）时就会尝试执行当前行中的命令；
- 可以使用“\[Enter]”来进行断行；
- 使用 # 作为注释符号。

52.3 Shebang

在计算机科学中，Shebang（也称为 Hashbang）是一个由井号和叹号构成的字符序列(`#!`)，其出现在文本文件的第一行的前两个字符。

在文件中存在 Shebang 的情况下，类 Unix 操作系统的程序载入器会分析 Shebang 后的内容，将这些内容作为解释器指令，并调用该指令，而且将载有 Shebang 的文件路径作为该解释器的参数。

例如，以指令 `#!/bin/sh` 开头的文件在执行时会实际调用/bin/sh 程序（通常是 Bourne shell 或兼容的 shell）来执行，而且这行内容也是 shell 脚本的标准起始行。

`#` 符号在许多脚本语言中都是注释标识符，Shebang 的内容会被这些脚本解释器自动忽略。在 `#` 字符不是注释标识符的语言（例如 Scheme）中，解释器也可能忽略以 `#!` 开头的首行内容，以提供与 Shebang 的兼容性。

“Shebang” 或者说 “Hashbang” 的名字有时也被当做 Ajax 应用程序中的分段标识符，用于浏览器的状态保存。例如，Google 蜘蛛将索引以叹号开头的分段标识符(即`...url#!state...`)。

具体来说，在 `#!` 字符之后可以有一个或数个空白字符，后接解释器的绝对路径，用于调用解释器。

在直接调用脚本时，调用者会利用 Shebang 提供的信息调用相应的解释器，从而使得脚本文件的调用方式与普通的可执行文件类似。

下面列出了一些典型的 shebang 解释器指令：

- `#!/bin/sh`—使用 sh（即 Bourne shell 或其它兼容 shell）执行脚本；
- `#!/bin/csh`—使用 csh（即 C shell）执行；
- `#!/usr/bin/perl -w`—使用带警告的 Perl 执行；
- `#!/usr/bin/python -O`—使用具有代码优化的 Python 执行；
- `#!/usr/bin/php`—使用 PHP 的命令行解释器执行。

在许多系统上，/bin/sh 是链接到 Bash 的，而/bin/csh 则是链接到 tcsh 的，因此设定前面的解释器实际上是运行的与之兼容的或改进的版本。

Shebang 行也可以包含需要传递到解释器的特定选项，不过，选项传递的方式随实现的不同而不同。

解释器指令允许脚本和数据文件充当系统命令，无需在调用时由用户指定解释器，从而对用户和其它程序隐藏其实现细节。

假设/usr/local/bin/foo 中有一以下行开头的 Bourne shell 脚本：

```
#!/bin/sh -x
```

而它被如此调用（“$” 是命令提示符）

```
$ foo bar
```

该命令的输出等同于

```
$ /bin/sh -x /usr/local/bin/foo bar
```

- argv[0] 被设定为脚本的文件名，而非解释器的文件名外。
- sh 从其命令行指定的文件中读取命令，上面的命令就会执行/usr/local/bin/foo 中的命令，同时将 bar 作为 foo 命令的参数 $1。

shebang 开头的井号也是 Bourne shell 和许多其它解释性语言的注释符，因此在这些语言中，解释器指令本身会被解释器认为是单纯的注释而跳过。

不过，并不是每一种解释器都会自动忽略 shebang 行，例如对于下面的脚本,cat 会把文件中的两行都输出到标准输出中。

```
#!/bin/cat
Hello world!
```

使用 `#!/usr/bin/env` 脚本解释器名称是一种常见的在不同平台上都能正确找到解释器的办法。

Linux 操作系统的文件一般是 UTF-8 编码，如果脚本文件是以 UTF-8 的 BOM（0xEF 0xBB 0xBF）开头的，那么 exec 函数将不会启动 shebang 指定的解释器来执行该脚本，因此 Linux 的脚本文件不应在文件开头包含 UTF-8 的 BOM。

另外，在执行 shell script 之前需要设置好重要的环境变量，例如 PATH 和 LANG 等。

在 shell script 的最后，可以使用 exit 命令中断程序并回传一个值给操作系统，例如 `exit 0` 代表将离开 script 并回传一个 0 给操作系统，用户通过 `$?` 可以获取程序退出代码。

用户通过 `exit n` 可以进一步设置自定义错误信息。

下面是一个 shell script 的示例，其中需要说明 script 的功能、版本信息、作者与联系方式、版权声明和历史信息等。

```
#! /bin/bash
# Program:
# Version:
# Author:
# Email:
# License:
# History:
```

为了提高 script 的可移植性，可以在其中设置好 PATH 等的绝对路径等信息。

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

```
# Program:
# Version:
# Author:
# Email:
# License:
# History:
PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin:~/bin
export PATH
```

Part IX

SSH

Chapter 53

Introduction

53.1 Overview

SSH（Secure SHell）是由 IETF 的网络工作小组（Network Working Group）所制定的创建在应用层和传输层基础上的一项安全协议，为计算机上的 Shell（壳层）提供安全的传输和使用环境。

最初的 SSH 协议是由芬兰研究员 Tatu Ylönen 于 1995 年设计开发的，现在人们已经转而使用没有授权和算法限制的 OpenSSH。

传统的网络服务程序（如 rsh、FTP、POP 和 Telnet）在网络上用明文传送数据、用户帐号和用户口令，因此其本质上都是不安全的，很容易受到中间人[1]（man-in-the-middle）攻击方式的攻击。

作为专门为远程登录会话和其他网络服务提供安全性的协议，在进行数据传输时，SSH 可以对数据包进行加密，因此可以有效防止远程联机过程中的信息泄露问题。

SSH 可以对所有传输的数据进行加密，也能够防止 DNS 欺骗和 IP 欺骗。另外，SSH 可以对其传输的数据进行压缩，从而可以加快传输的速度。

现在，SSH 既可以取代 Telnet、finger、rcp、rlogin、rsh 和 talk 等，又可以为 FTP、POP 或 PPP 提供一个安全的“通道”。

53.2 Architecture

SSH 协议框架中最主要的部分是三个协议，同时还有为许多高层的网络安全应用协议提供扩展的支持。

[1]所谓中间人攻击，就是存在另一个人或者一台机器冒充真正的服务器接收用户传给服务器的数据，然后再冒充用户把数据传给真正的服务器。

- 传输层协议（The Transport Layer Protocol）：传输层协议提供服务器认证，数据机密性，信息完整性等的支持。
- 用户认证协议（The User Authentication Protocol）：用户认证协议为服务器提供客户端的身份鉴别。
- 连接协议（The Connection Protocol）：连接协议将加密的信息隧道复用成若干个逻辑通道，提供给更高层的应用协议使用。

现在，各种高层应用协议可以相对地独立于 SSH 基本体系之外，并依靠这个基本框架，通过连接协议使用 SSH 的安全机制。

SSH 协议框架中设计了大量可扩展的冗余能力，比如用户自定义算法、客户自定义密钥规则、高层扩展功能性应用协议。这些扩展大多遵循 IANA 的有关规定，特别是在重要的部分（例如命名规则和消息编码）。

目前，SSH 协议中主要使用 RSA/DSA/Diffie-Hellman 等加密算法。

53.2.1 SSH 1

使用 SSH protocol version 1 时，SSH 服务器可以使用 RSA 加密算法来产生一个 1024bit 的 RSA key。

当 SSH daemon（sshd）启动时，就会产生一个 768bit 的公钥（或称为 Server key）并保存在 SSH 服务器中。

客户端的 SSH 联机请求传入 SSH 服务器后，SSH 服务器就会将公钥和加密数据传回客户端，并且客户端会根据/etc/ssh/ssh_known_hosts 或～/.ssh/known_hosts 来检查公钥的正确性。

客户端自己也会随机产生一个 256bit 的私钥（或称为 Private key），并且在接收到 768bit 的 Server key 后组合成一对完整的 Key pair，然后将这对 Key pair 传送回 SSH 服务器，并在之后的远程联机中使用这一对 1024bit 的 Key pair 进行数据的传递。

在使用 SSH 1 进行远程联机时，Server key 保存在 SSH 服务器中，客户端接收 Server key 后计算出 Private key，然后把二者组合成一把独一无二的 Key pair。

客户端每次的 256bit 的 Private key 都是随机产生的，因此每次联机时的 Private key 可能是不同的。

53.2.2 SSH 2

在使用 SSH protocol version 2 时，SSH 服务器接收到客户端的 Private key 后就不再针对 Key pair 进行校验，从而产生了潜在的危险。

SSH protocol version 2 增加了一个确认联机正确性的 Diffie-Hellman 机制，这样在每次数据传输中都会校验数据的来源是否合法，从而避免了在联机过程中被插入恶意代码的可能。

53.3 SSH Server

默认状态下，SSH 协议本身提供了 SSH 服务器和 SFTP 服务器功能，并且二者都架设在端口 22 上。

- 类似 telnet 远程登录 shell 的 SSH 服务器，并提供了相应的 Shell；
- 类似 ftp 服务器的 sftp-server，并提供更安全的 FTP 服务。

为了启动 SSH 服务器（sshd daemon），可以使用/etc/init.d/sshd 脚本来启动。

```
// start sshd daemon automatically at the boot time
# systemctl enable sshd.service
// start sshd daemon
// # /etc/init.d/sshd start
# systemctl start sshd.service
# netstat -tlp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address        Foreign Address      State     PID/Program name
tcp      0     0 0.0.0.0:ssh          0.0.0.0:*           LISTEN     19018/sshd
tcp6     0     0 [::]:http            [::]:*              LISTEN     7331/httpd
tcp6     0     0 [::]:ssh             [::]:*              LISTEN     19018/sshd
```

如果需要查看是否开机启动 sshd，可以使用 chkconfig 来检查 sshd 的设置。

```
# chkconfig sshd
```

与 SSH 有关的设置保存在/etc/ssh/sshd_config 中，如果对其进行了修改，那么必须重新启动 sshd 才能生效。

```
// restart sshd daemon
# /etc/init.d/sshd restart
# systemctl restart sshd.service
```

53.4 SSH Client

SSH 客户端是使用 Secure Shell（SSH）协议连接到远程计算机的软件程序，在客户端的 /.ssh/known_hosts 中会记录 SSH 服务器的 Server key 来确认远程联机是否正确。

SSH 客户端可以作为本地 SOCKS 代理来执行动态端口转发，不过 SSH 客户端本身是否可以通过代理来连接与提供 SOCKS 代理或端口转发不同。

用户可以使用 TUN/TAP 来从 SSH 客户端创建 VPN 链接。

另外，SSH 使用公钥/私钥来验证数据来提供对用户数据的加密，这样就可以利用密钥对的功能来省略密码。

将客户端产生的私钥（Private key）复制到 SSH 服务器中，当从客户端登录 SSH 服务器时，二者在 SSH 联机的信号传递中执行密钥的比对，这样用户就可以立即登入数据传输接口，不再需要密码。

- 在客户端建立 Public/Private key。

```
$ ssh-keygen
// $ ssh-keygen -t rsa
```

- 将 Private key 放置到客户端的 $HOME/.ssh 中，并修改权限为仅所有者可读。

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/theqiong/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/theqiong/.ssh/id_rsa.
Your public key has been saved in /home/theqiong/.ssh/id_rsa.pub.
The key fingerprint is:
b1:05:64:c5:ed:8e:2f:a4:26:a3:87:72:59:d7:e9:6c theqiong@localhost
The key's randomart image is:
+--[ RSA 2048]----+
|       .+o..     |
|       . .. .    |
|        . ..     |
|         +  .    |
|        S. +     |
|      . . = .    |
|     + . = .     |
|  . + + o E .    |
|   o.o + . .     |
+-----------------+
$ ls -a id_rsa
-rw-------. 1 theqiong theqiong 1675 Dec 19 10:18 id_rsa
```

- 将 Public key 放置到远程 SSH 服务器的指定用户的默认目录下的.ssh 目录中。

默认情况下，Private key 必须放置在默认目录下的.ssh 目录中。如果使用 SSH 2 的 RSA 算法，则需要放置在 $HOME/.ssh/id_rsa 中。

sshd_config 文件指定了 SSH 服务器中 Public key 放置的位置和默认的文件名。

```
$ cat /etc/ssh/sshd_config
AuthorizedKeysFile      .ssh/authorized_keys
$ cat .ssh/id_rsa.pub >> authorized_keys
$ ls -a authorized_keys
-rw-r--r--. 1 theqiong theqiong 396 Dec 19 10:35 authorized_keys
```

实际上，SSH 服务器可以开放给多个客户端，因此 authorized_keys 中可能会保存多个 Public key 内容，这里使用 >> 来将新的 Public key 内容追加到 authorized_keys 文件中。

为了登录其他的 SSH 服务器，只要将 Public key 复制到指定主机的 $HOME/.ssh/authorized_keys 文件中。

53.5 Security Authentication

在密码学中，加密是将明文信息隐匿起来，使之在缺少特殊信息时不可读。

加密算法（encryption）就是加密的方法，因此加密实际上就是将普通信息（明文，plaintext）转换成难以理解的信息（密文，ciphertext）的过程。

加密算法可以分为两类：对称加密和非对称加密。

- 对称加密就是将信息使用一个密钥进行加密，解密时使用同样的密钥，同样的算法进行解密。
- 非对称加密（又称公开密钥加密）是加密和解密使用不同密钥的算法，广泛用于信息传输中。

网络数据包的加密/解密技术通常是通过一对公钥和私钥组成的密钥对（public/private rsa key pair）进行加密和解密的过程。例如，当 SSH 客户端接收到 SSH 服务器的 Public key 时，客户端会通过～/.ssh/known_hosts 来比对 Public key 的正确性。

使用 SSH 协议时，远程主机要发送回客户端的数据会先通过公钥加密后再经过网络传输，客户端接收到的加密数据会使用私钥来解密。

在客户端来看，SSH 提供两种级别的安全验证（Security Authentication）。

- 第一种级别（基于密码的安全验证），知道帐号和密码就可以登录到远程主机，并且所有传输的数据都会被加密，但是可能会有别的服务器在冒充真正的服务器，无法避免被“中间人”攻击。
- 第二种级别（基于密钥的安全验证），需要依靠密钥，也就是你必须为自己创建一对密钥，并把公有密钥放在需要访问的服务器上。客户端软件会向服务器发出请求，请求用你的密钥进行安全验证。服务器收到请求之后，先在你在该服务器的用户根目录下寻找你的公有密钥，然后把它和你发送过来的公有密钥进行比较。如果两个密钥一致，服务器就用公有密钥加密“质询”（challenge）并把它发送给客户端软件，从而避免被“中间人”攻击。

在服务器端，SSH 也提供安全验证。

在第一种方案中，主机将自己的公用密钥分发给相关的客户端，客户端在访问主机时则使用该主机的公开密钥来加密数据，主机则使用自己的私有密钥来解密数据，从而实现主机密钥认证来确保数据的保密性。

在第二种方案中，存在一个密钥认证中心，所有提供服务的主机都将自己的公开密钥提交给认证中心，而任何作为客户端的主机则只要保存一份认证中心的公开密钥就可以了。在这种模式下，客户端必须访问认证中心然后才能访问服务器主机。

53.6 Security Setting

SSH 协议可以对传输的数据进行加密来确保安全性，但是 SSH 程序本身并不安全，因此 SSH 服务器对 Internet 开放登录的权限应该限制在指定的 IP 或主机范围内。

53.6.1 /etc/sshd_config

任何时候都应该禁止以 root 登录远程主机，可以通过下面的设置取消 root 的登录权限。

```
# vim /etc/ssh/sshd_config
#PermitRootLogin yes
PermitRootLogin no
```

如果需要禁止用户以 SSH 联机远程主机，可以在 SSH 服务器的设置中将用户放到特殊的用户组中（例如 nossh），并且在 sshd_config 中加入如下的设置：

```
# vim /etc/ssh/sshd_config
DenyGroups nossh
```

如果要禁止特定的用户以 SSH 联机，那么进行加入相应的设置如下：

```
# vim /etc/ssh/sshd_config
DenyUsers nosshuser
```

53.6.2 /etc/hosts.allow

在/etc/hosts.allow 中可以设置允许远程登录的主机 IP。

```
# vim /etc/hosts.allow
sshd: 10.0.0.2,10.0.0.2: allow
```

53.6.3 /etc/hosts.deny

在/etc/hosts.deny 中可以设置禁止远程登录的主机，并给出相应的提示。

```
# vim /etc/hosts.deny
sshd: ALL: spawn(/bin/echo Security notice from host `/bin/hostname`;\
/bin/echo; /usr/sbin/safe_finger @%h) | \
/bin/mail -s "%d -%h security" root@localhost & \
: twist(/bin/echo -e "WARNING connection not allowed.")
```

53.6.4 iptables

使用 iptables 可以设置不要开放 SSH 的登录权限给所有的 Internet 客户端。

53.7 SFTP

在计算机领域，SSH 文件传输协议 (SSH File Transfer Protocol，也称 Secret File Transfer Protocol，Secure FTP 或 SFTP) 是一种加密的针对远程文件的安全存取、传输和管理的网络传输协议。

跟早期的 SCP 协定只允许文件传输相比，SFTP 允许对远程文件执行更广泛的操作，而且 SFTP 客户端提供了额外的断点续传，目录列表和远程文件移动等功能。

最初，SFTP 由 IETF 设计来通过 SSH 2.0 的扩充提供安全文件传输能力，不过也能够被其它协议使用，例如在传输层安全（TLS）和 VPN 中进行文件传输。

现在，作为 SSH 的一部份，SFTP 是一种文件传输的安全方式，而且在 SSH 软件包中已经包含了 SFTP 子系统。

SFTP 本身没有单独的守护进程，它必须使用 sshd 守护进程（端口号默认是 22）来完成相应的连接操作，所以从某种意义上来说，SFTP 并不像一个服务器程序，而更像是一个客户端程序。

SFTP 同样是使用加密传输认证信息来确保传输的数据安全，所以使用 SFTP 是非常安全的。但是，由于 SFTP 传输方式使用了加密/解密技术，传输效率比普通的 FTP 要低得多，可以在对网络安全性要求更高时使用 SFTP 代替 FTP。

Windows 中可以使用 FTP 客户端软件来连接 SFTP 进行上传/下载文件，以及建立/删除目录等操作。

Linux 下可以直接在终端中输入：`sftp username@remote ip(or remote host name)` 连接远程 FTP 服务器，并且出现验证时输入正确的密码来实现远程登录。

在 sftp 的环境下的操作和一般 ftp 的操作类似，其中 ls、rm、mkdir、dir 和 pwd 等指令都是对远端进行操作。

如果使用 sftp 执行本地操作，只需在上述的指令上加上‘l’后变为 lls、lcd 和 lpwd 等。

- 上传：put /path/filename(本地主机) /path/filename(远端主机)
- 下载：get /path/filename(远端主机) /path/filename(本地主机)

如果需要使用 sftp 在非正规端口进行登录，可以使用 `sftp -o port=1000 username@remote ip`

Chapter 54

OpenSSH

OpenSSH（OpenBSD Secure Shell）是使用 SSH 通过计算机网络加密通信的实现，用于取代由 SSH Communications Security 所提供的商用版本。

OpenSSH 为 OpenSSL 有不同的目的和不同的发展团队，名称相近只是因为两者有同样的软件发展目标——提供开放源代码的加密通信软件。

目前，OpenSSH 是隶属于 OpenBSD 的开放源代码子项目。

54.1 History

1999 年 10 月，OpenSSH 第一次在 OpenBSD 2.6 里出现，最初的目的就是取代由 SSH Communications Security 所提供的 SSH 软件。

54.2 Architecture

54.2.1 ssh

ssh 是 rlogin 与 Telnet 的替代方案。

ssh 命令可以指定连接的 SSH 协议版本（version1 或 version2），还可以手动指定端口。

```
$ ssh account@hostname
```

如果直接以 `ssh hostname` 连接远程主机，则进入远程 SSH 服务器的“帐号名称”是目前工作目录的用户帐号，因此通常使用 E-mail 方式或`-l name` 形式来登录远程 SSH 服务器。

54.2.2 scp

scp 是 rcp 的替代方案，将文件复制到其他主机上。

```
// upload
$ scp test.txt user@hostname:/home/theqiong/doc/
// download
$ scp user@hostname:/home/theqiong/doc/test.txt
```

当使用 scp 命令在远程主机上进行文件或目录的复制操作时，使用“hostname:PATH”的形式来表示远程路径，并且以“-r”参数进行递归操作。

54.2.3 sftp

sftp 是 ftp 的替代方案，而且 sftp 与 ftp 有着几乎一样的语法和功能。

sftp 与 ssh 的登录方式相同，都是使用 `sftp -l username hostname` 或者直接以 `sftp user@hostname` 来登录。

```
$ sftp -l username hostname
$ sftp user@hostname
```

针对远程主机和本机的行为的区别在于，针对本机的操作要在前面加上 l。

- lcd：改变目录到本机的路径；
- lls：列出本机所在目录的文件；
- lmkdir：在本机新建目录；
- lpwd：显示当前所在的本机工作目录。

在不考虑 SFTP 的图形接口的情况下，可以以 sftp-server 来提供 FTP 服务，并使用 put/get 命令来执行上传/下载操作。

54.2.4 sshd

sshd 是 SSH 服务器。

54.2.5 ssh-keygen

ssh-keygen 产生用来认证使用的 RSA 或 DSA 密钥。

54.2.6 ssh-agent

OpenSSH 需要额外补丁以识别智能卡的引脚，不过 ssh-agent 可以直接识别智能卡。

- PuTTY 本身不支持智能卡，但是 PuTTY-CAC 可以支持智能卡。
- PuTTY 不支持 URL 超链接，只有 PuTTY 的分支 PuTTY Tray 支持。

- PuTTY 不能直接支持会话标签，在安装 PuTTY 连接管理器或 SuperPuTTY 的情况下才可以使用会话标签。
- PuTTY 不支持 AES-NI，PuTTY 的分支 PuTTY-AES-NI 可以。

54.2.7 ssh-add

帮助用户不需要每次都要输入密钥密码的工具。

54.2.8 ssh-keyscan

扫描多台机器，并记录其公钥。

Chapter 55

lsh

lsh 是一套由 GNU 项目推行的实施 SSH-2 协议的自由软件，并包含服务器及客户端程序，并依照 secsh-srp 实施远程安全密码协议。

此外，lsh 还包括公共密钥身份验证，Kerberos 也一定程度上得到支持，目前只支持密码验证，不支持 SSO 方式。

默认情况下，lsh 官方只支持一个 BSD 平台：FreeBSD。

Debian 提供的 lsh 的官方包包括 lsh-server、lsh-utils、lsh-doc 和 lsh-client。

Part X

Management

Chapter 56

log

Chapter 57

proc

Chapter 58

dmesage

Chapter 59

tail

Chapter 60

more/less

Part XI

Process

Chapter 61

Introduction

61.1 Overview

61.2 Parent Process

在计算机领域，父进程（Parent Process）指已创建一个或多个子进程的进程。

在 POSIX.1-2001 标准规定中，父进程可将 SIGCHLD 的处理函数设为 SIG_IGN（亦为默认设定），或为 SIGCHLD 设定 SA_NOCLDWAIT 标记，以使内核可以自动回收已终止的子进程的资源。

自 Linux 2.6 与 FreeBSD 5.0 起，两种内核皆支持了上述这两种方式。

不过，在忽略 SIGCHLD 信号的问题上，由于 System V 与 BSD 由来已久的差异，若要回收派生出的子进程的资源，调用 wait 仍是最便捷的方式。

61.2.1 UNIX

在 UNIX 里，除了进程 0（即 PID=0 的交换进程，Swapper Process）以外的所有进程都是由其他进程使用系统调用 fork 创建的，这里调用 fork 创建新进程的进程即为父进程，而相对应的为其创建出的进程则为子进程，因而除了进程 0 以外的进程都只有一个父进程，但一个进程可以有多个子进程。

操作系统内核以进程标识符（Process Identifier，即 PID）来识别进程。

- 进程 0 是系统引导时创建的一个特殊进程，在其调用 fork 创建出一个子进程（即 PID=1 的进程 1，又称 init）后，进程 0 就转为交换进程（有时也被称为空闲进程）。
- 进程 1（init 进程）就是系统里其他所有进程的祖先。

当一个子进程结束运行（一般是调用 exit、运行时发生致命错误或收到终止信号所导致）时，子进程的退出状态（返回值）会回报给操作系统，系统则以 SIGCHLD 信号将子进

程被结束的事件告知父进程，此时子进程的进程控制块（PCB）仍驻留在内存中。

一般来说，收到 SIGCHLD 后，父进程会使用 wait 系统调用以获取子进程的退出状态，然后内核就可以从内存中释放已结束的子进程的 PCB。但是，如果父进程没有这么做的话，子进程的 PCB 就会一直驻留在内存中，也即成为僵尸进程。

为避免产生僵尸进程，实际应用中一般采取的方式如下：

- 将父进程中对 SIGCHLD 信号的处理函数设为 SIG_IGN（忽略信号）；
- fork 两次并杀死一级子进程，令二级子进程成为孤儿进程而被 init 所“收养”或清理。

孤儿进程则是指父进程结束后仍在运行的子进程。

在类 UNIX 系统中，孤儿进程一般会被 init 进程所“收养”，从而成为 init 的子进程。

61.2.2 Linux

在 Linux 内核中，进程和 POSIX 线程有着相当微小的区别，父进程的定义也与 UNIX 不尽相同。

Linux 有两种父进程，分别称为（形式）父进程与实际父进程，对于一个子进程来说，其父进程是在子进程结束时收取 SIGCHLD 信号的进程，而实际父进程则是在多线程环境里实际创建该子进程的进程。

对于普通进程来说，父进程与实际父进程是同一个进程，但对于一个以进程形式存在的 POSIX 线程，父进程和实际父进程可能是不一样的。

61.3 Child Process

在计算机领域中，子进程为由另外一个进程（对应称之为父进程）所创建的进程。子进程继承了父进程的大部分属性（例如文件描述符）。

在 Unix 中，子进程通常为系统调用 fork 的产物，因此子进程开始时是父进程的副本。

根据具体需要，子进程还可以借助 exec 调用来链式加载另外的程序，并且可以以 `pstree PID` 的方式查询 PID 对应进程的子进程。

```
$ pstree 2594
chrome-sandbox─────chrome─┬─chrome─┬─3*[chrome─────10*[{chrome}]]
                          │        ├─13*[chrome─────9*[{chrome}]]
                          │        ├─10*[chrome─────11*[{chrome}]]
                          │        ├─2*[chrome─────8*[{chrome}]]
                          │        ├─chrome─────12*[{chrome}]
                          │        ├─3*[chrome─────18*[{chrome}]]
                          │        └─chrome─────76*[{chrome}]
                          └─chrome-sandbox─────nacl_helper
```

一个进程可能下属多个子进程，但是最多只能有 1 个父进程。如果某一进程没有父进程，则进程很可能由内核直接生成的。

61.4 Init Process

在 Unix 与类 Unix 系统中，进程 ID 为 1 的进程（即 init 进程）是在系统引导阶段由内核直接创建的，且不会在系统运行过程中终止执行。

对于其他无父进程的进程，则可能是为在用户空间完成各种后台任务而执行的。

当某一子进程结束、中断或恢复执行时，内核会发送 SIGCHLD 信号予其父进程。在默认情况下，父进程会以 SIG_IGN 函数进行忽略。

61.5 Orphan Process

在操作系统领域中，孤儿进程指的是在其父进程执行完成或被终止后仍继续运行的一类进程。

一般情况下，在对应的父进程结束执行后，进程就会变成孤儿进程，但是之后会立即由 init 进程“收养”为其子进程。

61.5.1 Process Re-parenting

在相容于 POSIX 标准的操作系统中，进程组（Process group）是指一个或多个进程的集合。

进程组被使用于控制信号的分配，而且一个进程组发出的的信号会被个别传递到该进程组下的每个进程成员中。

在类 UNIX 操作系统中，为避免孤儿进程退出时无法释放所占用的资源而僵死，任何孤儿进程产生时都会立即为系统进程 init 自动接收为子进程，这一过程也被称为“收养”（re-parenting）

不过，虽然事实上该进程已有 init 作为其父进程，但由于创建该进程的进程已不存在，所以仍应称之为“孤儿进程”。

61.5.2 Process Group

父进程终止或崩溃都会导致对应子进程成为孤儿进程，因此也无法预料一个子进程执行期间是否会被“遗弃”。

多数类 UNIX 系统都引入了进程组以防止产生孤儿进程：在父进程终止后，用户的 Shell 会将父进程所在进程组标为“孤儿进程组”，并向终止的进程下属所有子进程发出 SIGHUP 信号以试图结束其运行，从而避免子进程继续以“孤儿进程”的身份运行。

RPC 过程中也会产生孤儿进程。例如，若客户端进程在发起请求后突然崩溃，且对应的服务器端进程仍在运行，则该服务器端进程就会成为孤儿进程。这样的孤儿进程会浪费服务器的资源，甚至有耗尽资源的潜在危险，但也有对应的解决办法：

- 终止机制：强制杀死孤儿进程（最常用的手段）；
- 再生机制：服务器在指定时间内查找调用的客户端，若找不到则直接杀死孤儿进程；
- 超时机制：给每个进程指定一个确定的运行时间，若超时仍未完成则强制终止之。若有需要，亦可让进程在指定时间耗尽之前申请延时。

除此之外，用户也可能会刻意使进程成为孤儿进程，以使之与用户会话脱钩，并转至后台运行。

实际上，上述这一做法常应用于启动需要长时间运行的进程（也即守护进程）。

另外，UNIX 命令 nohup 也可以完成这一操作。

61.5.3 Session Group

进程组本身也可以被集合成一个群组来管理，称为会话群组（sessions）。

归属于某个特定会话群组下的进程组不能移动到别的会话组下，在某个进程组下的特定行程在创造出新的进程时，这个进程也只能属于这个父进程所归属的相同会话群组。

61.5.4 Control groups

cgroups（control groups）是 Linux 内核的一个功能，用来限制、控制与分离一个进程组群的资源（如 CPU、内存、磁盘输入输出等）。

cgroups 项目由 Google 工程师 Paul Menage 和 Rohit Seth 在 2006 年发起，最早的名称为进程容器（process containers）。

cgroups 的一个设计目标是为不同的应用情况提供统一的接口，从控制单一进程（例如 nice）到系统级虚拟化（例如 opeNVZ、Linux-VServer、LXC），而且可以提供如下的特性：

- 资源限制：组可以被设置不超过设定的内存限制（其中也包括虚拟内存）。
- 优先化：一些组可能会得到大量的 CPU 或磁盘输入输出通量。
- 报告：用来衡量系统确实把多少资源用到适合的目的上。
- 分离：为组分离命名空间，这样一个组不会看到另一个组的进程、网络连接和文件。
- 控制：冻结组或检查点和重启动。

61.6 Zombie Process

61.6.1 Overview

某一子进程终止执行后，若其父进程未提前调用 wait，则内核会持续保留子进程的退出状态等信息，以使父进程可以 wait 获取它。

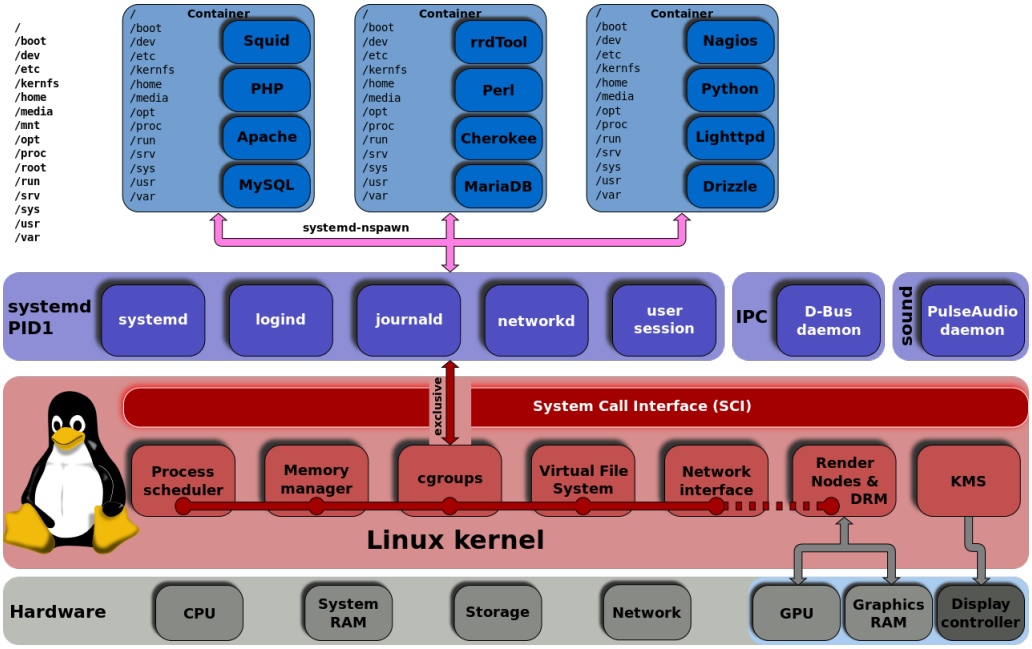


Figure 61.1: Unified hierarchy cgroups

在这种情况下，子进程虽已终止，但是仍然在消耗系统资源，所以其亦称僵尸进程。

wait 常于 SIGCHLD 信号的处理函数中调用。例如，子进程死后，系统会发送 SIGCHLD 信号给父进程，父进程对其默认处理是忽略。

如果想响应这个消息，父进程通常在 SIGCHLD 信号事件处理程序中，使用 wait 系统调用来响应子进程的终止。

在类 UNIX 系统中，僵尸进程是指完成执行（通过 exit 系统调用，或运行时发生致命错误或收到终止信号所致）但在操作系统的进程表中仍然有一个表项（进程控制块 PCB），处于” 终止状态” 的进程。

僵尸进程发生于子进程需要保留表项以允许其父进程读取子进程的 exit status 时，一旦退出态通过 wait 系统调用读取，僵尸进程条目就从进程表中删除，称之为” 回收（reaped）”。

正常情况下，进程直接被其父进程 wait 并由系统回收，如果进程长时间保持僵尸状态一般是错误的并导致资源泄漏。

UNIX 命令 ps 列出的进程的状态（”STAT”）栏标示为”Z” 则为僵尸进程。

```
$ ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.0  50924  3996 ?        Ss   10:39   0:01 /usr/lib/system
root          2  0.0  0.0      0     0 ?        S    10:39   0:00 [kthreadd]
```

与正常进程不同，kill 命令对僵尸进程无效，因此收割僵尸进程的正确方法是通过 kill 命令手工向其父进程发送 SIGCHLD 信号。如果其父进程仍然拒绝收割僵尸进程，则终止父进程，使得 init 进程收养僵尸进程。init 进程周期执行 wait 系统调用收割其收养的所有僵尸进程。

孤儿进程不同于僵尸进程，其父进程已经死掉，但是孤儿进程仍能正常执行，而且并不会变为僵尸进程，因为被 init（进程 ID 号为 1）收养并 wait 其退出。

僵尸进程被收割后，其进程号 (PID) 与在进程表中的表项都可以被系统重用。但如果父进程没有调用 wait，僵尸进程将保留进程表中的表项，导致了资源泄漏。

不过，某些情况下这反倒是期望的：父进程创建了另外一个子进程，并希望具有不同的进程号。如果父进程通过设置事件处理函数为 SIG_IGN 显式忽略 SIGCHLD 信号，而不是隐式默认忽略该信号，或者具有 SA_NOCLDWAIT 标志，所有子进程的退出状态信息将被抛弃并且直接被系统回收。

为避免产生僵尸进程，实际应用中一般采取的方式如下：

- 将父进程中对 SIGCHLD 信号的处理函数设为 SIG_IGN（忽略信号）；
- fork 两次并杀死一级子进程，从而使二级子进程成为孤儿进程而被 init 所“收养”和清理。

```
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
   pid_t pids[10];
   int i;

   for (i = 9; i >= 0; --i) {
      pids[i] = fork();
      if (pids[i] == 0) {
         sleep(i+1);
         _exit(0);
      }
   }

   for (i = 9; i >= 0; --i)
      waitpid(pids[i], NULL, 0);

   return 0;
```

```
}
```

61.7 at

61.8 bg

61.9 chroot

61.10 cron

61.11 fg

61.12 kill

61.13 killall

61.14 wait

多进程系统内的进程（或任务）有时需要等待其他进程以完成自己的执行过程，而在包含父-子进程机制的类 UNIX 系统中，父进程能创建可独立运行的子进程，并在需要时调用 wait（函数声明为 `pid_t wait(int *stat_loc)`）以使自己在子进程执行过程中保持休眠状态。

当任一子进程结束后，该子进程会向操作系统返回一个退出状态，而后系统即向休眠中的父进程发送一个 SIGCHLD 信号以提醒之，至此父进程“复苏”并从内核获取子进程的退出状态，而后内核释放原有子进程所占用的资源，父进程也继续执行。

对于带有线程机制的类 UNIX 系统来说，对于线程调度也有对应 wait 的实现。例如，pthread_join 会让当前进程强制休眠，等待指定线程执行完毕后再继续执行。

类 UNIX 系统还提供多种 wait 的派生调用（如 waitpid 和 waitid）以扩展适用范围。借助于这些变种，父进程可以休眠至任一子进程结束，也可以等待满足指定条件（如匹配给定的进程标识符）的子进程结束后再继续执行。

另外，若利用额外选项做参数，waitpid 和 waitid 在指定进程继续运行或暂停执行时也会返回。

即使没有提前调用 wait，在任一进程终止后，系统内核都会向其父进程发送 SIGCHLD，这时父进程可以选择使用 SIG_IGN 作为处理函数，令内核知晓自己不需获得状态，并直接交由 init 进程处理。或者，也可调用 wait，则立即返回子进程退出状态。

若两者皆不做，则子进程在进程表中占用的资源就无法得到释放，进而成为僵尸进程，持续浪费资源。

为解决这一问题，系统常以特殊进程 reaper（“收割者”）定位僵尸进程，并获取其状态以使系统可以解除资源分配，从而实现进程“收割”。

61.15 nice

61.16 pgrep

61.17 pidof

61.18 pkill

61.19 ps

61.20 pstree

61.21 time

61.22 top

Part XII

Daemon

Chapter 62

Introduction

62.1 Overview

守护进程（daemon）是指在 UNIX 或其他多任务操作系统中在后台执行的程序，而且不会接受用户的直接操控。

守护进程通常会被以进程的形式初始化，守护进程程序的名称通常以字母“d”结尾。例如，syslogd 就是指管理系统日志的守护进程。

通常情况下，守护进程没有任何存在的父进程（即 PPID=1），并且在 UNIX 系统进程层级中直接位于 init 之下。

守护进程程序通常通过如下方法使自己成为守护进程：对一个子进程调用 fork，然后使其父进程立即终止，使得这个子进程能在 init 下运行，因此这种方法通常被称为“脱壳”。

操作系统通常在启动时一同启动守护进程。

- 在 DOS 环境中的守护进程被称为驻留程序（TSR）。
- 在 Windows 系统中使用 Windows 服务来履行守护进程的职责。
- Mac OS 系统中将守护进程称为“extensions”。
- Mac OS X 有守护进程，而且其“服务”不同于 Windows 中的服务。

具体来说，守护进程为对网络请求、硬件活动等进行响应，或其他通过某些任务对其他应用程序的请求进行回应提供支持。

另外，守护进程也能够对硬件进行配置（例如 Linux 系统中的 devfsd）、运行计划任务（例如 cron），以及运行其他任务。

62.2 Background

后台进程（Background Process）是一种在不需用户干预的情况下运行于操作系统后台的计算机进程，通常用于执行如日志记录、系统监测、作业调度以及用户提醒等任务。

在 UNIX 与类 UNIX 系统中，后台进程的进程组 ID（即 PGID，可用 ps 命令获得）与控制终端进程组 ID（即 TPGID）不同，因而也可以此辨识后台进程。

后台进程无法接收从键盘传送的信号（如 Ctrl-C），因此从更专业的定义来说，程序是否能收到用户的中断信号并非后台进程的判别标准。

后台进程通常用于仅需少量资源的应用，但任何进程无论占用资源多少都可以运行在后台，且即使程序在后台运行，其行为与前台进程也并无差异。

在类 UNIX 系统的命令行模式下，用户可使用“&”操作符以启动进程并使之运行于后台，但是标准输出（stdout）和标准错误输出（stderr）若未重定向则仍于前台（即当前父终端）输出。

Chapter 63

Signal

Chapter 64

Thread

64.1 Multithread

Part XIII

Software Management

Chapter 65

Applications

65.1 Compile

通常当使用手动编译源码方式来安装软件时，表示没有找到 RPM 或 DEB 安装资源，或者是需要以自定义的方式安装软件。采用手动编译源码方式安装需要自己编译源文件后再安装，所有通常需要系统有 gcc、make 之类的编译软件。

1. 下载源码，通常是 tar 文件。
2. 解压 tar 包，并在 Linux 终端运行如下命令：

   ```
   tar -xzvf <文件名>
   ```

   或：

   ```
   tar -xjvf <文件名>
   ```

   参数说明：
   - -x，表示解压；
   - -z 解压 gzip 格式文件；
   - -j 解压 bzip2 格式的文件
   - -v 显示详细信息；
   - -f 解压到文件。
3. 编译，（通常在解压好的文件夹下有个 configure 文件，运行该文件即可，如果需要自定义安装，就需要查看帮助文档，查看编译参数，在 Linux 终端中运行./configure 命令。）
4. make。
5. make install。

在 configure 的过程中会提示错误，通常是提示你缺少某个组件，只需按照提示安装组件即可完成编译。

65.1.1 configure

65.1.2 make

65.1.3 make install

65.2 Package

软件包是对于一种软件所进行打包的方式，一般包括二进制文件和元数据（例如描述、版本和依赖）等。

如果正在使用某一款音乐播放器，那么它并不需要直接去操作声卡之类的硬件设备，而只需要去调用系统内核间接地控制声卡即可。更复杂一点的是如果要设计一个视频播放器，但现在并不知道如何去使用内核去操作显卡等硬件。如果知道有某个别人已经设计好的组件可以实现这样的功能，那么就可以只设计播放器的界面效果，然后直接使用别人的组件去调用内核来间接地控制硬件设备。

在 Linux 中的软件一般都是比较小巧、零散的，所以也就出现了安装某一个软件时提示依赖关系错误。即使是安装一个非常小的软件，但该软件可能需要依托于其他几十个组件的帮助才可以实现该软件应有的功能等情况。

在不同的操作系统中，软件包的类型有很大的区别。例如，Linux、BSD 系统中的软件包主要以两种形式出现：二进制包以及源代码包，其中源代码包主要适用于自由软件的安装，需要用户自己进行编译。

在 Windows 操作系统中，软件包大多数以安装程序的方式出现，可以将软件安装在制定的目录中，也有直接使用压缩工具打包的，解压缩之后便可运行。

不同的 Linux 发行版对于安装软件[?]提供了多种解决方案。

- rpm：传统的 Red Hat Linux 二进制包。
- deb：Debian 系列的二进制包。

65.2.1 RPM

RPM、deb 安装方式一般针对特定发行版本，RPM 是针对红帽系统的安装包，deb 是针对 Ubuntu 系统的安装包，这种包会把相关软件及组件打包在一起，可以直接从网上下载 RPM 格式或 deb 格式的文件直接安装到相对应的系统里，但这种方式还是不能彻底解决依赖关系的问题。因为每个个人用户在安装系统时选择安装的组件不同，所以 RPM 包也不可能把所有相关的软件及组件都包括在里面。

安装 RPM 包的方法也很简单，直接在终端运行中运行：

```
rpm -ivh <文件名>
```

参数说明：

- -i，表示安装（install）。
- -v，显示附加信息。
- -h，显示 hash 符号（#）。

现在，RPM 可能是指.rpm 的文件格式的软件包，也可能是指其本身的软件包管理器（RPM Package Manager）。

RPM 仅适用于安装用 RPM 来打包的软件，具体可以分为二进制包（Binary）、源代码包（Source）和 Delta 包三种。二进制包可以直接安装在计算机中，而源代码包将会由 RPM 自动编译、安装，源代码包经常以 src.rpm 作为后缀名。

65.2.2 deb

deb 是 Debian 软件包格式，文件扩展名为.deb。处理 deb 包的经典程序是 dpkg，经常是通过前端工具 apt 来执行。

Debian 包是 Unixar 的标准归档，将包文件信息以及包内容，经过 gzip 和 tar 打包而成。

具体来说，deb 文件是使用 ar 打包，包含了三个文件：

- debian-binary - deb 格式版本号码
- control.tar.gz - 包含包的元数据[1]，如包名称、版本、维护者、依赖、冲突等。
- data.tar.* - 实际安装的内容

其中，“*” 所指代的内容随压缩算法不同而不同，常见的可能值为 xz、gz、bz2 或 lzma 等。

另外，通过 Alien 工具可以将 deb 包转换成其他形式的软件包。

65.2.3 APK

Android 应用程序包文件（Android application package，APK）是一种 Android 操作系统上的应用程序安装文件格式。

每一个 Android 应用程序的代码都必须先进行编译，然后被打包成为一个被 Android 系统所能识别的文件才可以被运行，而这种能被 Android 系统识别并运行的文件格式便是“APK”。

一个 APK 文件内包含被编译的代码文件 (.dex 文件)、文件资源（resources）、assets、证书（certificates）和清单文件（manifest file）。

[1]在 dpkg 1.17.6 之后添加了对 xz 压缩和不压缩的 control 元数据的支持。

Chapter 66

Environment Variables

Chapter 67

Printer

67.1 printf

67.1.1 printf

使用 printf 命令并配合 awk 可以格式化输出信息。

- \a：警告声音
- \b：退格键（backspace）
- \f：清空屏幕（form feed）
- \n：新的一行
- \r：Enter 键
- \t：水平的 Tab 键
- \v：垂直的 Tab 键
- \xNN：NN 为两位数的数字，可以转换数字为字符

在 C 语言中，常见的变量格式如下：

- %ns 表示字符个数；
- %ni 表示整数个数；
- %N.nf 中的 n 和 N 都是数字，其中 f 表示 float（浮点）。

例如，为了使用 Tab 空格来分隔数据，可以使用如下的命令；

```
printf '%s\t %s\t %s\t %s\t %s\t \n' $(cat test.txt)
```

printf 本身不是管道命令，因此需要将文件内容提取出来并作为 printf 命令的后续数据。

如果需要以固定的字段宽度来输出格式化内容，可以对上述命令进行进一步限制。

```
printf '%10s %5i %5i %5i %8.2f \n' $(cat test.txt | grep -v Name)
```

除了可以格式化处理数据之外，printf 命令还可以依据 ASCII 的数字与图形对应来显示结果。

```
printf '\x45\n'
```

67.2 pr

pr 命令用于将文本文件转换为适合打印的格式并发送到打印机济进行打印。

pr 命令对文本文件的处理包括设置打印标题、号码、宽度等。

Chapter 68

Scanner

68.1 SANE

Chapter 69

USB

69.1 lsusb

Chapter 70

Kernel

70.1 Setup

70.2 X Window

70.2.1 GNOME

70.2.2 KDE

Chapter 71

Development Envirment

71.1 Introduction

就软件开发项目而言，开发环境这一术语是指软件在开发和部署系统时所需的全部工件，其中包括工具、指南、流程、模板和基础设施。

其中，软件开发环境（Software Development Environment）指的是在基本硬件和软件的基础上，为支持系统软件和应用软件的工程化开发和维护而使用的一组软件，简称 SDE。具体来说，SDE 由软件工具和环境集成机制组成，前者用以支持软件开发的相关过程、活动和任务，后者为工具集成和软件的开发、维护及管理提供统一的支持。

软件开发环境在欧洲又叫集成式项目支援环境（Integrated Project Support Environment，IPSE），其主要组成部分是软件工具。

具有统一的交互方式的人机界面是软件开发环境的重要质量标志，而存储各种软件工具加工所产生的软件产品或半成品（如源代码、测试数据和各种文档资料等）的软件环境数据库是软件开发环境的核心。工具间的联系和相互理解都是通过存储在信息库中的共享数据得以实现的。

软件开发环境数据库是面向软件工作者的知识型信息数据库，其数据对象是多元化、带有智能性质的，从而可以用来支撑各种软件工具，尤其是自动设计工具、编译程序等的主动或被动的工作。

- 较初级的 SDE 数据库一般包含通用子程序库、可重组的程序加工信息库、模块描述与接口信息库、软件测试与纠错依据信息库等；
- 较完整的 SDE 数据库还应包括可行性与需求信息档案、阶段设计详细档案、测试驱动数据库、软件维护档案等。

更进一步的要求是面向软件规划到实现、维护全过程的自动进行，这要求 SDE 数据库系统是具有智能的，其中比较基本的智能结果是软件编码的自动实现和优化、软件工程项目的多方面不同角度的自我分析与总结。

对软件开发环境的配置还应主动地被重新改造、学习，以丰富 SDE 数据库的知识、信息和软件积累，因此软件开发环境在软件工程人员的恰当的外部控制或帮助下会逐步向高度智能与自动化迈进。

根据开发阶段的不同，开发环境可以划分为前端开发环境（支持系统规划、分析、设计等阶段的活动)、后端开发环境（支持编程、测试等阶段的活动)、软件维护环境和逆向工程环境等，这些开发环境往往可通过对功能较全的环境进行剪裁而得到。

软件开发环境由工具集和集成机制两部分构成，工具集和集成机制间的关系犹如“插件”和“插槽”间的关系。软件开发环境由工具集和集成机制两部分构成，工具集和集成机制间的关系犹如“插件”和“插槽”间的关系。

软件开发环境中的工具可包括支持特定过程模型和开发方法的工具，例如支持瀑布模型及数据流方法的分析工具、设计工具、编码工具、测试工具、维护工具，支持面向对象方法的 OOA 工具、OOD 工具和 OOP 工具等。

独立于模型和方法的工具（例如界面辅助生成工具和文档出版工具）亦可包括管理类工具和针对特定领域的应用类工具。

集成机制对工具的集成及用户软件的开发、维护及管理提供统一的支持。按功能可划分为环境信息库、过程控制及消息服务器、环境用户界面三个部分。

- 环境信息库是软件开发环境的核心，用以储存与系统开发有关的信息并支持信息的交流与共享。库中储存两类信息，一类是开发过程中产生的有关被开发系统的信息（例如分析文档、设计文档、测试报告等)，另一类是环境提供的支持信息（例如文档模板、系统配置、过程模型、可复用构件等)。
- 过程控制和消息服务器是实现过程集成及控制集成的基础。过程集成是按照具体软件开发过程的要求进行工具的选择与组合，控制集成并行工具之间的通信和协同工作。
- 环境用户界面包括环境总界面和由它实行统一控制的各环境部件及工具的界面。统一的、具有一致视感（Look & Feel）的用户界面是软件开发环境的重要特征。

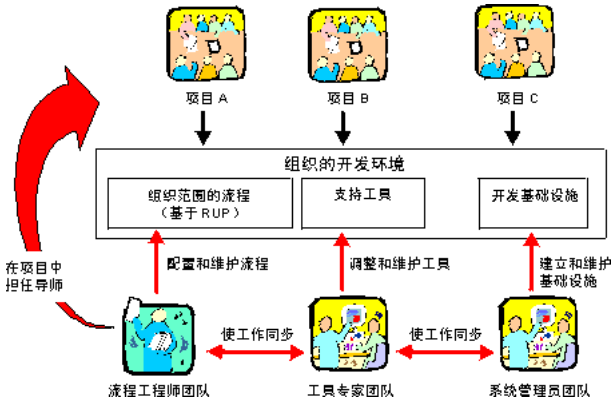
在某些情况下，有必要对软件开发环境的某些部分进行讨论，例如以下的两个示例：

- 测试环境，包括测试工件的模板（例如测试计划和测试评估概要的模板)、测试指南、测试工具和必要的开发基础设施。
- 实施环境，包括实施工件的模板（例如集成构建计划的模板)、编程指南、实施工具（例如编译器和调试器）以及必要的开发基础设施。

开发组织内的不同项目之间通常会存在许多相似之处，各项目集本上都以相似的方法使用同样的工具。对于不同的项目，流程大致相似，某些指南也可能一样。如果开发组织让一个团队来开发和维护组织的开发环境（即由组织范围的流程、工具使用和基础设施组成的开发环境)，就可以提高开发效率。

流程工程师将负责开发和维护组织范围的流程。有了组织范围的流程，就可减少单独的软件开发项目中的流程定制工作，大部分定制工作已经在开发组织范围的流程时完成。

工具专家可以负责设置和维护支持工具，并协助各个软件开发项目进行工具设置。另



外，系统管理员也可以成为此环境团队中的成员。

71.2 IDE

具体来说，集成开发环境（Integrated Development Environment，简称 IDE，也称为 Integration Design Environment、Integration Debugging Environment）是一种辅助程序开发人员开发软件的应用软件。

IDE 通常包括编辑器、自动构建工具、调试器、编译器/解释器，或者调用第三方编译器来实现代码的编译工作。另外，IDE 还可以包含 UML 建模工具、版本控制工具和图形用户界面设计工具，以及类浏览器、组件检查器等。

虽然也可以把 UNIX 当成是一个 IDE，但是多数的开发人员会把 IDE 当成是一个可以完成各种开发工作的一个程序，UNIX 提供了创建、修改、编译、发布、分析和调试等功能。IDE 试图把各种命令行的开发工具结合起来形成一个抽象化的工具，从而将各种开发工作进行更密切的整合（例如在编码的时候就直接编译，发现有语法错误就立即修改）。

在集成化开发环境之后出现的可视化开发环境使开发者可以将控件根据自己的意图进行组装，这样就解决了很多例行的、标准化的代码，比非可视化的开发环境更加直观，开发速度快，效率高。

以 Delphi 为例，Delphi 包含了程序代码文件（.PAS）和控件布局文件（.dfm），当在画布（FORM）上拖放一个按钮（BUTTON）时，Delphi 开发环境会自动创建一个 DFM 文件标明 BUTTON 位置，并且自动在 PAS 文件中自动输出最基本的完整代码，这样开发人员就只需要在需要修改的地方修改或者增加就可以完成很多功能。

71.3 SDK

软件开发工具包（Software Development Kit，SDK）一般是一些被开发人员用于为特定的软件包、软件框架、硬件平台、操作系统等创建应用软件的开发工具的集合。

SDK 可以只是简单的为某个程序设计语言提供应用程序接口的一些文件，但也可能包括能与某种嵌入式系统通讯的复杂的硬件，而且 SDK 一般都包括用于调试和其他用途的实用工具。

另外，SDK 还经常包括示例代码、支持性的技术文档或者其他的为基本参考资料澄清疑点的支持文档等。

为了鼓励开发者使用其系统或者语言，系统开发人员提供的 SDK 很多都是免费提供的。注意，SDK 可能附带了使其不能在不兼容的许可证下开发软件的许可证。例如，一个专有的 SDK 可能与自由软件开发抵触，而 GPL 能使 SDK 与专有软件开发近乎不兼容，只有 LGPL 下的 SDK 没有这个问题。

71.3.1 DirectX SDK

DirectX 是由 Microsoft 创建的一系列专为多媒体以及游戏开发的应用程序接口，包含了 Direct3D、Direct2D、DirectCompute 等多个不同用途的子部分，它们主要基于 C++ 编程语言实现，并遵循 COM 架构。

DirectX 被广泛用于 Microsoft Windows、Microsoft Xbox 电子游戏开发，并且只能支持这些平台。除了游戏开发之外，DirectX 亦被用于开发许多虚拟三维图形相关软件，例如 Direct3D 是 DirectX 中最广为应用的子模块。

- Direct3D 主要用于绘制 3D 图形。
- Direct2D 为 DirectDraw 的替代者，主要提供 2D 动画的硬件加速。
- DirectWrite 主要字体显示 API，可以控制 GPU 来使字体显示更为平滑，类似 ClearType。
- XInput 主要用于 Xbox360 的控制器。
- XAudio2 主要用于低延迟游戏音频播放。
- DirectCompute 为 GPU 通用计算 API。
- DirectXMath 为针对游戏优化的高速数学运算 API，使用 SSE2 指令集，特别支持单精度浮点运算及矩阵运算。
- DirectSetup 用于 DirectX 组件的安装，以及检查 DirectX 的版本。
- DirectX Media 包含 DirectAnimation、DirectShow 等，其中 DirectAnimation 可用于 2D 的网页动画（web animation），DirectShow 可支持多媒体回放、流媒体以及 DirectX 在网页上的转换。DirectShow 亦包含有 DirectX plugins 用于 audio signal processing 以及 DirectX Video Acceleration 加速影音音效。
- DirectX Media Objects 用于支持数据流对象（streaming objects），包括编码（encoders）、解码（decoder）以及效果（effects）等。

DirectX SDK 版本编号由微软的 DxDiag 工具获得（4.09.0000.0900 以及更高版本，在开始菜单 | 运行中输入 DxDiag 即可），编号统一使用 x.xx.xxxx.xxxx 格式，而微软网站上给出的编号使用 x.xx.xx.xxxx 格式，因此如果网站上编号为 4.09.00.0904，那么在电脑上安装后就

会变为 4.09.0000.0904。

2010 年 6 月 7 日发布的 DirectX 11 SDK 是最后独立发布的 SDK 版本，之后的 DirectX SDK 被集成进新版的 Microsoft Windows SDK 里。例如，DirectX 11.2 SDK 被集成到 Windows Software Development Kit (SDK) for Windows 8.1 中。

71.3.2 Java SDK

Java Development Kit（JDK）是 Sun 针对 Java 开发人员发布的免费软件开发工具包。

自从 Java 推出以来，JDK 已经成为使用最广泛的 Java SDK。由于 JDK 的一部分特性采用商业许可证，因此 2006 年 Sun 宣布将发布基于 GPL 协议的开源 JDK，使 JDK 成为自由软件。在去掉了少量闭源特性之后，Sun 最终促成了 GPL 协议的 OpenJDK 的发布。

作为 Java 语言的 SDK，普通用户并不需要安装 JDK 来运行 Java 程序，而只需要安装 JRE（Java Runtime Environment），但是程序开发者必须安装 JDK 来编译、调试程序。

JDK 包含了一批用于 Java 开发的组件，其中包括：

Table 71.1: JDK 组件

JDK 组件	备注
appletviewer	运行和调试 applet 程序的工具，不需要使用浏览器
apt	注释处理工具
extcheck	检测 jar 包冲突的工具
idlj	IDL-to-Java 编译器，可以将 IDL 语言转化为 java 文件
jar	打包工具，将相关的类文件打包成一个文件
jarsigner	
java	运行器，执行.class 的字节码
javac	编译器，将后缀名为.java 的源代码编译成后缀名为.class 的字节码
javadoc	文档工具，从源码注释中提取文档，注释需符合规范
javafxpackager	
javah	从 Java 类生成 C 头文件和 C 源文件。这些文件提供了连接胶合，使 Java 和 C 代码可进行交互。
javap	反编译程序
javapackager	
java-rmi.cgi	
javaws	运行 JNLP 程序
jcmd	
jconsole	
jcontrol	

JDK 组件	备注
jdb	java 调试器
jdeps	
jhat	堆分析工具
jinfo	获取正在运行或崩溃的 java 程序配置信息
jjs	
jmap	获取 java 进程内存映射信息
jmc	
jmc.ini	
jps	显示当前 java 程序运行的进程状态
jrunscript	命令行脚本运行工具
jsadebugd	
jstack	栈跟踪程序
jstat	JVM 检测统计工具
jstatd	jstat 守护进程
jvisualvm	
keytool	
native2ascii	
orbd	
pack200	
policytool	策略文件创建和管理工具
rmic	
rmid	
rmiregistry	
schemagen	
serialver	
servertool	
tnameserv	
unpack200	
wsgen	
wsimport	
xjc	

JDK 中还包括完整的 JRE（Java Runtime Environment），用于生产环境的各种库类（例如基础类库 rt.jar），以及给开发人员使用的补充库（例如国际化与本地化的类库、IDL 库等）。

另外，JDK 中还包括各种样例程序，用以展示 Java API 中的各部分。

无论 Linux、Windows 或者 Mac OS 系统，JDK 均有 X86 与 X64 的发行版本，并且均为多语言发行，可以根据系统语言的不同自动显示不同语言的信息。

自 JDK 5.0 起，Java 以两种方式发布更新：

- Limited Update 包含新功能和非安全修正，版本号是 20 的倍数。
- Critical Patch Updates（CPUs）只包含安全修正，版本号将是上一个 Limited Update 版本号加上五的倍数后的奇数。

71.3.3 OpenJDK

OpenJDK 是甲骨文公司公司为 Java 平台构建的 Java 开发环境的开源版本，完全自由，开放源码。

71.3.4 Android SDK

Android SDK 提供了 Android 开发过程中需要的 API 库和必要的工具用于构建、测试和调试 Android 应用程序。

- ADT Plugins
- Android SDK Tools
- Android Platform-tools
- Android Platform
- Android system image

71.3.5 Android NDK

71.3.6 iOS SDK

iOS 软件开发工具包（iOS SDK，亦称 iPhone SDK）是由苹果公司开发的为 iOS 设计的应用程序开发工具包，允许开发者开发 iPad、iPhone、iPod touch 应用程序。

iOS SDK 的首个版本于 2008 年 2 月发布，苹果通常会发布两个 iOS 软件开发工具包，包括主要的 iOS X.0 和次要的 iOS X.X。iOS SDK 只能在 Mac OS X Leopard 及以上系统并拥有英特尔处理器上运行，其他的操作系统（包括微软的 Windows 操作系统和旧版本的 Mac OS X 操作系统）都不被支持。

iOS 软件开发工具包本身是可以免费下载的，但开发人员如果希望向 App Store 发布应用，就必需加入 iOS 开发者计划，需要付款以获得苹果的批准。加入 iOS 开发者计划后，开发人员们将会得到一个牌照，可以用这个牌照将编写的软件发布到苹果的 App Store。

开发人员可以通过应用商店发布任意设价的应用程序，付费应用将让开发人员获得 70% 的费用配额，免费的应用程序没有任何费用配额。

iOS 开发者计划的出现使人们不能根据 GPLv3 的授权代码发布软件。任何根据 GPLv3 任何代码的开发者也必须得到 GPLv3 的授权。同时，开发商在散发布已经由 GPLv3 授权的应用软件的同时必须提供由苹果公司提供的密匙以允许该软件修改版本的上传。

iOS 是从于 Mac OS X 核心演变而来，因此从 Xcode 3.1 发布以后，Xcode 就成为了 iOS 软件开发工具包的开发环境。和 Mac OS X 的应用程序一样，iOS 应用程序使用 Objective-C 语言，也可以写成 C 或 C++ 语言。

- 触控 (Cocoa Touch)：多点触控事件和控制 (Multi-touch events and controls)、加速支持 (Accelerometer support)、查看等级 (View hierarchy)、本地化（i18n）(Localization (i18n))、相机支持。
- 媒体：OpenAL、混音及录音 (Audio mixing and recording)、视频播放、图像文件格式 (Image file formats)、Quartz、Core Animation、OpenGL ES。
- 核心服务：网络、SQLite 嵌入式数据库、地理位置 (GeoLocation)、线程 (Threads)。
- OS X 核心：TCP/IP 协议、套接字 (Sockets)、电源管理、文件系统 (File system)、安全。

iOS 软件开发工具包中包含和 Xcode 工具一样的 iOS 模拟器，让开发人员在计算机上拥有仿真的外观和感觉，但是 iOS 模拟器并不是一个用于运行 x86 目标代码的工具。

从 iOS 2.1 固件开始，iPhone 版 Safari 开始支持 SVG 1.1 的编码特征和大部分静态功能。不过 Safari 的图形界面还不支持 SMIL 动画，这需要等 SMIL 引擎足够成熟之后才能被支持。另外，Safari 还支持 HTML Canvas。

目前，Apple 仍未开放在浏览器内执行 Flash 内容，因此只能在 iOS 越狱之后安装第三方 Flash 软件。

Chapter 72

Library

72.1 Introduction

在计算机科学中，库（library）是用于开发软件的子程序集合，还可能包括配置、文档、帮助、模板、类或函数、值或类型规格等。

现代软件开发往往利用模块化开发的方式，每一个软件模块（Module）都是一套一致而互相有紧密关连的软件组织，分别包含了程序和数据结构两部份。

模块是可能分开地被编写的单位，从而使得它们可重用和允许不同的开发人员同时协作、编写及研究不同的模块。

模块的接口表达了由该模块提供的功能和调用它时所需的元素，因此库与可执行文件不同，它不是独立程序，其他程序必须通过库的接口来引用库的功能。

静态链接和动态链接都可以把一个或多个库包括到程序中。相应的，前者链接的库叫做静态库，后者的叫做动态库。

标准库是程序设计语言的每种实现中都统一提供的库。在某些情况下，编程语言规格说明中会直接提及该库，或者由编程社区中的非正式惯例决定。

根据宿主语言构成要素的不同，标准库可包含如下要素：

- 子程序
- 宏定义
- 全局变量
- 类定义
- 模板

大多数标准库都至少含有如下常用组件的定义：

- 算法（例如排序算法）
- 数据结构（例如表、树、哈希表）
- 与宿主平台的交互，包括输入输出和操作系统调用

72.2 Static Library

静态库是相对于动态库而言的。对静态库代码的调用中，在链接阶段就会把库代码包含入可执行文件。

在计算机科学里，静态库是一个外部函数与变量的集合体，静态函数库通常包含变量与函数，在编译期间由编译器与链接器将它们整合至应用程序内，并生成目标文件及可以独立运行的可执行文件。

链接器是一个独立程序，将一个或多个库或目标文件（先前由编译器或汇编器生成）链接到一块生成可执行程序。静态链接是由链接器（Linker）在链接时将库的内容加入到可执行程序中的做法，因此以过去的观点来说，函数库只能算是静态（static）类型。

静态函数库可以用 C/C++ 语言来建立，它们可以提供关键字来指定函数与变量是否为外部（external）或是内部（internal）链接。如果需要将函数或是变量导出（export），则一定要用外部链接（external linkage ）的语法来指定它们。

```
//static_lib.h

# ifndef _STATIC_LIB_H_
# define _STATIC_LIB_H_

# include <iostream>
# include <string>
# include <Windows.h>

using namespace std;

namespace STAIC_LIB
{
   BOOL PRINT(__in string& STRING);
}

# endif
```

Listing 72.1: static_lib.h

```
//static_lib.cpp

# include "static_lib.h"

BOOL STAIC_LIB::PRINT(__in string& STRING)
```

```
{
    if ( STRING.empty() )
    {
        return FALSE;
    }
    //显示一个字符串
    cout<<STRING<<endl;
    return TRUE;
}
```

Listing 72.2: static_lib.cpp

GCC 编译器中的静态库文件名为 lib*.a，为了使用其中的函数，可以使用-l* 参数要求链接器连入。例如，在许多系统上，当使用了 math.h 中的函数后，需要使用-lm 参数连接 libm.a 文件。

在 Visual Studio 中，静态库文件名为 *.lib。为了使用其中的函数，可以使用 `#pragma comment(lib,"*")` 预编译指令要求连接器连入。

与动态链接相比，静态链接的最大缺点是生成的可执行文件太大，需要更多的系统资源，在装入内存时也会消耗更多的时间。但是，使用静态库时，只需保证在开发者的计算机有正确的库文件，在以二进制发布时不需考虑在用户的计算机上库文件是否存在及版本问题，可避免 DLL 地狱等问题。

72.3 Shared Library

动态链接是指在可执行文件装载时或运行时由操作系统的装载程序加载库。可以动态链接的库，在 Windows 上是 dynamic link library (DLL)，在 UNIX 或 Linux 上是 Shared Library。

库文件是预先编译链接好的可执行文件，存储在计算机的硬盘上。大多数情况下，同一时间多个应用可以使用一个库的同一份拷贝，操作系统不需要加载这个库的多个实例。

大多数操作系统将解析外部引用（比如库）作为加载过程的一部分，这些系统中的可执行文件包含一个叫做 import directory 的表，该表的每一项包含一个库的名字。根据表中记录的名字，装载程序在硬盘上搜索需要的库，然后将其加载到内存中预先不确定的位置，之后根据加载库后确定的库的地址更新可执行程序。

装载程序在加载应用软件时要完成的最复杂的工作之一就是加载时链接。可执行程序根据更新后的库信息调用库中的函数或引用库中的数据，这种类型的动态加载称为装载（load-time）时加载，已被 Windows 和 Linux 等大多数系统采用。

其他操作系统可能在运行时解析引用，这些系统中的可执行程序调用操作系统 API 后，将库的名字、函数在库中的编号和函数参数一同传递。操作系统负责立即解析然后代表应

用调用合适的函数，这种动态链接叫做运行时链接。因为每个调用都会有系统开销，运行时链接要慢得多，对应用的性能有负面影响，现代操作系统已经很少使用运行时链接。

动态链接的最大缺点是可执行程序依赖分别存储的库文件才能正确执行。如果库文件被删除了、移动了、重命名了或者被替换为不兼容的版本了，那么可执行程序就可能工作不正常，这就是常说的 DLL-hell。

72.4 Runtime Library

在计算机程序设计领域中，运行时库（runtime library）是指一种被编译器用来实现编程语言内置函数以提供该语言程序运行时（执行）支持的一种特殊的计算机程序库。

运行时库一般包括基本的输入输出或是内存管理等支持，与操作系统合作提供诸如数学运算、输入输出等功能，让开发者可以不需要“重新发明轮子”，并高效使用操作系统提供的功能。

运行时库由编译器决定，以面向编程语言来提供其最基本的执行时需要。比如，Visual Basic 需要复杂的运行时库支持而 C 的运行时库则相对简单。

运行时库中的函数可能对程序员透明，也可能不透明，这取决于编译器对语言执行环境的需求。

早期的运行时库（例如 Fortran）提供了数学运算的能力。其他语言增加了诸如垃圾回收的先进功能，通常用于支持对象数据结构。

许多近代语言设计了更大的运行环境并添加更多功能，很多面向对象语言也包含了分派器与类读取器。例如，Java 虚拟机（JVM）便是此类的典型运行环境，它也在运行期直译或编译具可移植性的二进制 Java 程序。另外，.NET 架构也是另外一个运行时库的实例。

一个以 Java 语言开发的软件，可通过 Java 软件运行可预测的指令接收 Java 运行环境的服务功能。根据 Java 运行时库提供的这些服务，Java 运行环境可视为此程序的运行时环境，程序与 Java 环境都向操作系统提出请求并获取服务。

操作系统核心为它自己、所有进程与在它控制之下的软件提供服务，因此操作系统可视为自己提供自己的运行时环境。

异常处理（Exception handling）是专门处理运行期错误的语言机制，使程序员可以完全捕捉非预期错误，或没有适当处理的错误结果。

动态链接库或静态链接库与运行时库的分类角度不同，不得相提并论。

72.5 Class Libary

72.5.1 Framework Class Library

框架类库（Framework Class Library）通常被定义为可重用类、接口和值类型等，其中基本类库（Base Class Library）是框架类库的核心，提供了基本类和名称空间等，从而实现了框架的基本功能。例如，.NET Framework Class Library 就包括了 System, System.CodeDom, System.Collections, System.Diagnostics, System.Globalization, System.IO, System.Resources and System.Text 等名称空间。

Microsft 的.NET Framework 就是一个以通用语言运行库（Common Language Runtime）为基础，致力于敏捷软件开发（Agile software development）、快速应用开发（Rapid application development）、平台无关性和网络透明化的类库集合。

.NET Framework 也为应用程序接口（API）提供了新功能和开发工具，以及新的反射性的且面向对象程序设计编程接口。

.NET Framework 的初级组成是 CLI 和 CLR，其中：

- CLI（Common Language Infrastructure，通用语言架构）是一套运作环境帮助，包括一般系统、基础类库和与机器无关的中间代码。
- CLR（Common Language Runtime，通用语言运行平台）是对通用语言架构的实现。在通用中间语言（CIL）运行前，CLR 必须将指令及时编译转换成原始机器码。

所有 CIL 都可经由.NET 自我表述，CLR 检查元数据以确保正确的方法被调用。元数据通常是由语言编译器生成，但开发人员也可以通过使用客户属性创建自己的元数据。

如果一种语言实现生成了 CLI，它也可以通过使用 CLR 被调用，这样它就可以与任何其他.NET 语言生成的数据相交互，也就是说 CLR 被设计为具有操作系统无关性。

当一个汇编体被加载时，CLR 运行各种各样的测试（例如确认与核查）。在确认的时候，CLR 检查汇编体是否包含有效的元数据和 CIL，并且检查内部表的正确性。核查则不那么精确，核查机制主要检查代码是否会运行一些“不安全”的操作。

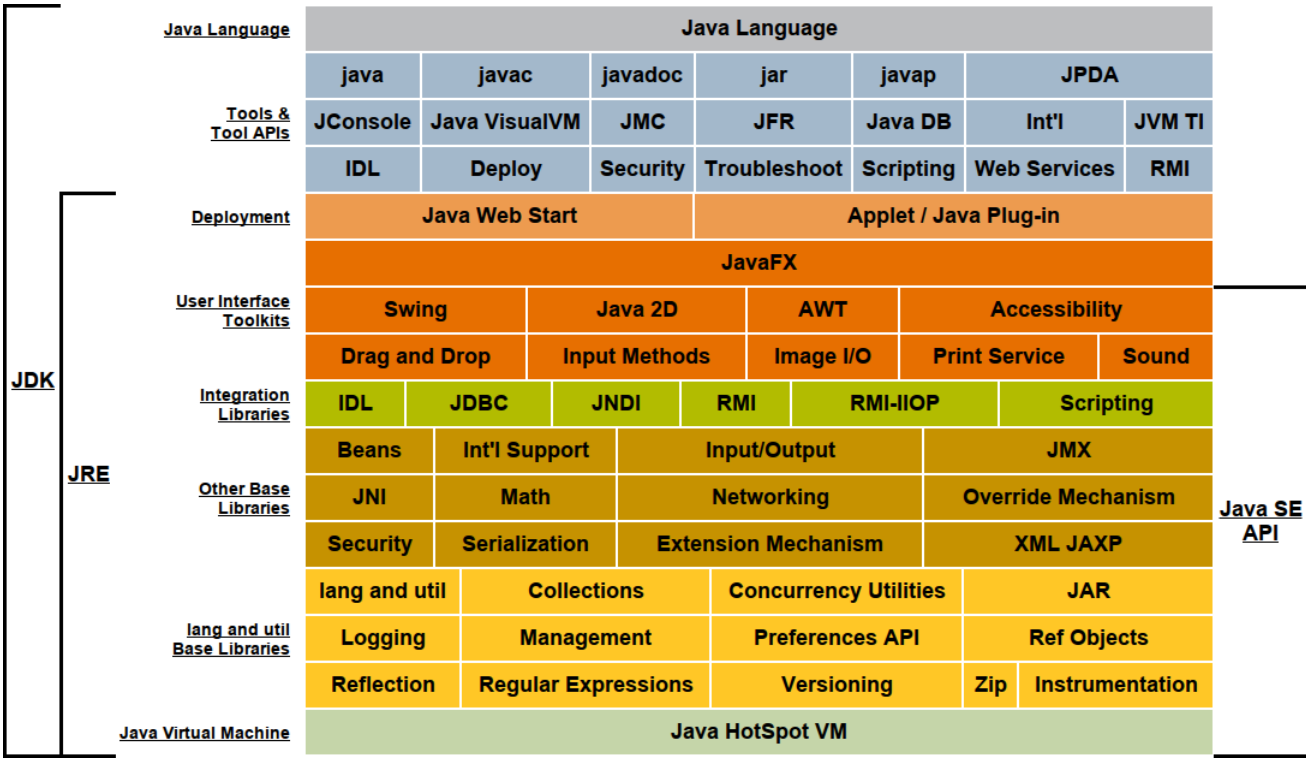
公共语言基础（CLI）、通用中间语言（CIL）以及 C# 与 Sun 的 Java 虚拟机和 Java 之间有不少的相似之处，二者都使用它们各自的中间码，因此它们无疑是强烈的竞争者。

CLI 被设计成支持任何面向对象的编程语言，CIL 被设计来实时编译（JIT），而 Java 的字节码在最初的时候则是设计成用于解释运行，而非实时编译。

ASP.NET 和 ADO.NET 也都是内含于.NET 框架中的可用于开发 Web 应用程序以及数据访问的类库。

72.5.2 Java Class Library

Java 被设计成具有平台无关性，因此 Java 类库（JCL）封装了一组在 Java 应用运行时可以动态载入的标准类库，包含了在不同操作系统上都通用的功能。



基本上来说，Java 类库都是由 Java 语言实现的，并且几乎所有的 Java 类库都包含在 rt.jar 中，它们都可以通过 Java 本地接口（Java Native Interface，JNI）来访问操作系统 API。

Java 集合框架（Java Collections Framework，JCF）是一组实现了通用数据结构的集合类和接口。

- Collection 接口是一组允许重复的对象。
- Set 接口继承 Collection，但不允许重复，使用自己内部的一个排列机制。
- List 接口继承 Collection，允许重复，以元素安插的次序来放置元素，不会重新排列。
- Map 接口是一组成对的键－值对象，即所持有的是 key-value pairs。Map 中不能有重复的 key，拥有自己的内部排列机制。

72.5.3 Java Package

Java Package 是一种将 Java 类或接口等组织到名字空间中的机制，它可以被保存为压缩的 jar 文件中，从而以组的形式下载 Java 类。

Java API 本身就是包的集合，例如 javax.xml 包就包含了处理 XML 的类。另外，每一个包都按照它所包含的类型提供了唯一的名字空间，而且在同一个包内的类可以访问其中的成员。

在开发 Java 应用时，可以使用 package 关键字来引入相关的类，例如：

```
package java.awt.event;
\end{lstlsiting}
```

为了使用某个Java包中的所有类，可以使用**import**关键字来导入某个包中的所有的类。

```
\begin{lstlisting}[language=Java,xleftmargin=.25in]
import java.awt.event.*;
\end{lstlsiting}
```

或者，使用improt关键字也可以导入某个确定的类。例如，下面的示例说明了如何从java.awt.event包中导入Acti

```
\begin{lstlisting}[language=Java,xleftmargin=.25in]
import java.awt.event.ActionEvent;
\end{lstlsiting}
```

使用上述方式导入Java包之后，就可以在代码中使用相关的类。

```
\begin{lstlisting}[language=Java,xleftmargin=.25in]
ActionEvent testEvent = new ActionEvent();
\end{lstlsiting}
```

如果不使用Java包来导入类，也可以使用绝对位置来引用类，例如:

```
\begin{lstlisting}[language=Java,xleftmargin=.25in]
java.awt.event.ActionEvent testEvent = new java.awt.event.ActionEvent();
\end{lstlsiting}
```

为了更好地组织代码结构，可以使用jar命令来将.**class**文件打包到jar文件中，例如:

```
\begin{lstlisting}[language=bash,xleftmargin=.25in]
jar cf testPackage.jar *.class
\end{lstlsiting}
```

在Java应用开发中，经常使用的核心包如下:

```
\begin{compactitem}
\item java.lang——basic language functionality and fundamental types
```

```
\item java.util--collection data structure classes
\item java.io--file operations
\item java.math--multiprecision arithmetics
\item java.nio --the New I/O framework for Java
\item java.net--networking operations, sockets, DNS lookups, ...
\item java.security--key generation, encryption and decryption
\item java.sql--Java Database Connectivity (JDBC) to access databases
\item java.awt--basic hierarchy of packages for native GUI components
\item javax.swing--hierarchy of packages for platform-independent rich
    GUI components
\item java.applet--classes for creating an applet
\end{compactitem}
```

```
\chapter{Package Manager}
```

```
\section{Introduction}
```

软件一般是按模块被设计出来的，用户可以直接面向UI界面操作，而计算机在设计之初就是分层次分模块被

软件包是对于一种软件所进行打包的方式。在不同的操作系统中，软件包的类型有很大的区别。

在Linux发行版中，几乎每一个发行版都重度依赖于某种软件包管理系统（例如RPM、dpkg等），可以将其理

采用RPM、dkpg等解决方案的原因是使用简单，基本无依赖关系问题。具体来说，软件包管理系统的原理是搭建一台

\begin{compactitem}
\item RPM (RPM Package Manager) 是由Red Hat推出的软件包管理系统。
\item dpkg (Debian
    Package) 由Debian发行版开发以用于安装、卸载以及提供和deb软件包相关的信息。
\end{compactitem}

其他软件包管理系统有ArchLinux中使用的Pacman，Gentoo使用的基于源代码的Portage和Mac系统下的Homebrew等。

\begin{compactitem}
\item
    ArchLinux基于KISS原则，针对i686和x86−64的CPU做了优化，以.pkg.tar.xz格式打包并由包管理器进行跟踪
\item
    Gentoo采用Portage包管理系统来自行编译及调整源码依赖等选项，以获得至高的自定义性及优化的软件。
\item Mac OS X系统使用Homebrew简化软件安装和2管理。
\end{compactitem}

在典型的软件包管理系统中，使用dpkg以及它的前端apt（使用于Debian、Ubuntu）来管理deb软件包，而rpm以及它

\subsection{YUM}

使用软件包管理系统将大大简化在Linux发行版中安装软件的过程，例如YUM源配置文件在/etc/yum.repos.d/目录下

.repo文件内容格式如下:

\begin{compactitem}
\item 可选项[]中的内容为项目名称。
\item name为服务器名称。
\item baseurl为服务器地址，该地址一定是一个真实、可用的YUM服务器地址。
\item enable=0表示不启动Yum服务，如果想使用该服务，需要修改为1。
\item gpgcheck=1表示是否对软件进行签名检验，0为不校验。

\item gpgkey表示校验签名文件位置。
\end{compactitem}

如果有多个网络YUM服务器，可以在下面继续添加YUM项目并且格式相同，修改完YUM配置文件后需要运行\te
clean all}来初始化新的配置文件。

\subsection{APT}

apt-get原理与YUM一样，只不过RedHat公司用的是yum命令，而Ubuntu公司用的是apt-get命令。

\section{Python}

\subsection{pip}

pip是一个以Python语言开发的软件包管理系统，可以用于安装和管理软件包。类似地，EasyInstall也是一

easy_install是一个附带设置工具的模块和一个第三方函数库，类似用于Ruby语言的RubyGems，可以加快

通过pip提供的命令行接口，可以方便地安装软件包。

\begin{lstlisting}[language=bash,xleftmargin=.25in]
pip install package-name

Python Package Index（简称 PyPI）是一个由 Python 基金会维护的第三方 Python 软件仓库，因此 pip 安装软件包时默认都是从 PyPI 进行查找的。

此外，也可以方便地使用下面的命令来移除软件包。

pip uninstall package-name

pip 也可以通过“需求”文件来管理软件包和其相应版本数目的完整列表，从而可以对一个完整软件包组合在另一个环境（如另一台计算机）或虚拟化环境中进行有效率的重新创造。

```
pip install -r requirements.txt
```

另一方面，pip 也可以通过“Heroku”等软件支持 Python 在云端网页缓存中的使用。

72.5.4 easy_install

72.6 JavaScript

72.6.1 npm

Node.js 是一个基于 Google 的 V8 引擎的事件驱动 I/O 服务端 JavaScript 环境，可以用来开发可扩展的网络程序（例如 Web 服务）。

与一般 JavaScript 不同的地方在于，Node.js 并不是在浏览器上运行的，而是在服务器上运行服务端 JavaScript。

Node 包管理器（Node Package Manager）运行在命令行下，可以用来管理 Node.js 应用的依赖。例如，可以使用下面的命令安装 LESS。

```
$ npm install less
```

从 Node.js0.6 版本开始，npm 被自动附带在安装包中。

Node.js 实现了部份 CommonJS 规范，并包含了一个交互测试 REPL 环境。例如，下面是一个用 Node.js 撰写的 HTTP Server 版 hello world 示例。

```
var http = require('http');

http.createServer(function (request, response) {
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.end('Hello World\n');
}).listen(8000);

console.log('Server running at http://127.0.0.1:8000/');
```

下面是另一个简单的 TCP 服务器示例，监听（Listening）端口 7000 并输出 (echo) 之前输入的消息。

```
var net = require('net');

net.createServer(function (stream) {
```

```
    stream.write('hello\r\n');

    stream.on('end', function () {
        stream.end('goodbye\r\n');
    });

    stream.pipe(stream);
}).listen(7000);
```

72.6.2 nvm

目前，Node.js 的各种特性都没有稳定下来，经常由于老项目或尝新的原因，需要切换各种版本，因此开发了 nvm（Node Version Manager）。

首先，通过下面的命令来安装 nvm。

```
$ curl https://raw.githubusercontent.com/creationix/nvm/v0.17.2/install.sh
    | bash
```

安装完成后，在 shell 里面调用 nvm 命令，如果有如下输出，则说明 nvm 安装成功。

```
Node Version Manager

Usage:
    nvm help                    Show this message
    nvm --version               Print out the latest released version of
    nvm
    nvm install [-s] <version>  Download and install a <version>, [-s]
    from source. Uses .nvmrc if available
    nvm uninstall <version>     Uninstall a version
    nvm use <version>           Modify PATH to use <version>. Uses .nvmrc
    if available
    nvm run <version> [<args>]  Run <version> with <args> as arguments.
    Uses .nvmrc if available for <version>
    nvm current                 Display currently activated version
    nvm ls                      List installed versions
    nvm ls <version>            List versions matching a given description
    nvm ls-remote               List remote versions available for install
    nvm deactivate              Undo effects of NVM on current shell
    nvm alias [<pattern>]       Show all aliases beginning with <pattern>
```

```
    nvm alias <name> <version>  Set an alias named <name> pointing to
    <version>
    nvm unalias <name>          Deletes the alias named <name>
    nvm copy-packages <version> Install global NPM packages contained in
    <version> to current version
    nvm unload                  Unload NVM from shell

Example:
    nvm install v0.10.24        Install a specific version number
    nvm use 0.10                Use the latest available 0.10.x release
    nvm run 0.10.24 myApp.js    Run myApp.js using node v0.10.24
    nvm alias default 0.10.24   Set default node version on a shell

Note:
    to remove, delete, or uninstall nvm - just remove ~/.nvm, ~/.npm, and
    ~/.bower folders
```

72.6.3 Bower

Bower is a package management system for client-side programming on the World Wide Web. It depends on Node.js and npm. It works with git and GitHub repositories.

72.7 Perl

72.7.1 ppm

Perl Package Manager（PPM）可以用于简化查找、安装、升级和移除 Perl 软件包的工作。

ppm 可以检查已安装软件的版本，并且还可以从本地或远程主机来安装或升级软件。

ppm 可以从“PPM 仓库”中查找预编译的 Perl 模块，例如 ppm 可以重新构建从 CPAN 下载的 PPM 模块。

72.8 PHP

72.8.1 PECL

72.8.2 PEAR

72.8.3 Composer

72.9 Ruby

72.9.1 gem

RubyGems 是 Ruby 的一个包管理器，提供了分发 Ruby 程序和函数库的标准格式“gem”，旨在方便地管理 gem 安装的工具，以及用于分发 gem 的服务器，类似于 Python 的 pip。

RubyGems 创建于约 2003 年 11 月，从 Ruby 1.9 版起成为 Ruby 标准库的一部分。

Gem 是类似于 Ebuilds 的包，其中包含包信息，以及用于安装的文件。

Gem 通常是依照“.gemspec”文件构建的，其为包含了有关 Gem 信息的 YAML 文件。不过，Ruby 代码也可以利用 Rake 来直接建立 Gem。

具体来说，gem 命令用于构建、上传、下载以及安装 Gem 包，在功能上与 apt-get、portage、yum 和 npm 非常相似。

安装:

```
gem install mygem
```

卸载:

```
gem uninstall mygem
```

列出已安装的gem:

```
gem list --local
```

列出可用的gem，例如:

```
gem list --remote
```

为所有的gems创建RDoc文档:

```
gem rdoc --all
```

下载一个gem，但不安装:

```
gem fetch mygem
```

从可用的gem中搜索，例如:

```
gem search STRING --remote
```

gem 命令也被用来构建和维护.gemspec 和.gem 文件。例如，可以通过下面的命令来利用.gemspec 文件构建.gem。

```
gem build mygem.gemspec
```

Chapter 73

Update

Chapter 74

内核

在计算机科学中，内核（Kernel）又称核心[?]，是操作系统最基本的部分。

内核主要负责管理系统资源，作为众多应用程序提供对计算机硬件的安全访问的一部分软件，对内核的访问是有限的，并由内核决定一个程序在什么时候对某部分硬件操作多长时间。由于直接对硬件操作是非常复杂的，因此通常是通过内核来提供一种硬件抽象的方法以完成这些操作。

通过进程间通信机制及系统调用，应用进程可间接控制所需的硬件资源（特别是处理器及 IO 设备）。

严格地说，内核并不是计算机系统中必要的组成部分，程序也可以直接地被调入计算机中执行，不过这种情况常见于早期计算机系统的设计中。当时使用这样的设计，说明设计者不希望提供任何硬件抽象和操作系统的支持。最终，一些辅助性程序（例如程序加载器和调试器）被设计到机器核心当中，或者写入在只读记忆体里。这些变化发生时，操作系统内核的概念就渐渐明晰起来了。

内核可分为四大类：

- 单内核：单内核为潜在的硬件提供了大量完善的硬件抽象操作。
- 微内核：微内核只提供了很小一部分的硬件抽象，大部分功能由一种特殊的用户态程序——服务来完成。
- 混合内核：混合内核很像微内核结构，只不过它的组件更多的在核心态中运行，用以获得更快的执行速度。
- 外内核：外内核不提供任何硬件抽象操作，但是允许为内核增加额外的运行库，通过这些运行库应用程序可以直接地或者接近直接地对硬件进行操作。

74.1 Monolithic kernels

宏内核结构在硬件之上，定义了一个高阶的抽象界面，应用一组原语 (或者叫系统调用 (System call)) 来实现操作系统的功能，例如进程管理，文件系统，和存储管理等等，这些功能由多个运行在核心态的模块来完成。

尽管每一个模块都是单独地服务这些操作，内核代码是高度集成的，而且难以编写正确。因为所有的模块都在同一个内核空间上运行，一个很小的 bug 都会使整个系统崩溃。然而，如果开发顺利，单内核结构就可以从运行效率上得到好处。

很多现代的宏内核结构内核，如 Linux 和 FreeBSD 内核，能够在运行时将模块调入执行，这就可以使扩充内核的功能变得更简单，也可以使内核的核心部分变得更简洁。

宏内核结构的例子:

- 传统的 UNIX 内核,BSD 伯克利大学发行的版本
- Linux 内核
- MS-DOS, Windows 9x (Windows 95, 98, Me)

74.2 Micro kernels

微内核结构由一个非常简单的硬件抽象层和一组比较关键的原语或系统调用组成；这些原语，仅仅包括了创建一个系统必需的几个部分；如线程管理，地址空间和进程间通讯等。

微核的目标是将系统服务的实现和系统的基本操作规则分离开来。例如，进程的输入/输出锁定服务可以由运行在微核之外的一个服务组件来提供。这些非常模块化的用户态服务器用于完成操作系统中比较高级的操作，这样的设计使内核中最核心的部分的设计更简单。一个服务组件的失效并不会导致整个系统的崩溃，内核需要做的，仅仅是重新启动这个组件，而不必影响其它的部分。

微内核将许多 OS 服务放入分离的进程, 如文件系统, 设备驱动程序, 而进程通过消息传递调用 OS 服务. 微内核结构必然是多线程的, 第一代微内核, 在核心提供了较多的服务, 因此被称为' 胖微内核', 它的典型代表是 MACH, 它既是 GNU HURD 也是 APPLE SERVER OS 的核心, 可以说, 蒸蒸日上. 第二代微内核只提供最基本的 OS 服务, 典型的 OS 是 QNX,QNX 在理论界很有名, 被认为是一种先进的 OS。

微内核结构的例子:

- AIX
- BeOS
- L4 微内核系列
- Mach, 用于 GNU Hurd
- Minix

- MorphOS
- QNX
- RadiOS
- VSTa

单内核结构是非常有吸引力的一种设计，由于在同一个地址空间上实现所有复杂的低阶操作系统控制代码的效率会比在不同地址空间上实现更高些。

20 世纪 90 年代初，单内核结构被认为是过时的。把 Linux 设计成为单内核结构而不是微内核，引起了无数的争议 (参见塔能鲍姆-林纳斯辩论)。

现在，单核结构正倾向于设计不容易出错，所以它的发展会比微内核结构更迅速些。两个阵营中都有成功的案例。微核经常被用于机器人和医疗器械的嵌入式设计中，因为它的系统的关键部分都处在相互分开的，被保护的存储空间中。这对于单核设计来说是不可能的，就算它采用了运行时加载模块的方式。

尽管 Mach 是众所周知的多用途的微内核，人们还是开发了除此之外的几个微内核。L3 是一个演示性的内核，只是为了证明微内核设计并不总是低运行速度。它的后续版本 L4，甚至可以将 Linux 内核作为它的一个进程，运行在单独的地址空间。

QNX 是一个从 20 世纪 80 年代，就开始设计的微内核系统。它比 Mach 更接近微内核的理念。它被用于一些特殊的领域；在这些情况下，由于软件错误，导致系统失效是不允许的。例如航天飞机上的机械手，还有研磨望远镜镜片的机器，一点点失误就会导致上千美元的损失。

很多人相信，由于 Mach 不能够解决一些提出微内核理论时针对的问题，所以微内核技术毫无用处。Mach 的爱好者表明这是非常狭隘的观点，遗憾的是似乎所有人都开始接受这种观点。

74.3 Hybrid kernel

混合内核实质上是微内核，只不过它让一些微核结构运行在用户空间的代码运行在内核空间，这样让内核的运行效率更高些。这是一种妥协做法，设计者参考了微内核结构的系统运行速度不佳的理论。然而后来的实验证明，纯微内核的系统实际上也可以是高效率的。大多数现代操作系统遵循这种设计范畴，微软视窗就是一个典型的例子。另外还有 XNU，运行在苹果 Mac OS X 上的内核，也是一个混合内核。

混合内核的例子:

- Windows NT/2000/XP/Server 2003 以及 Windows Vista/7 等基于 NT 技术的微软视窗操作系统
- Mac OS X
- BeOS 内核
- DragonFly BSD

- ReactOS 内核
- XNU

一些人认为可以在运行时加载模块的单核系统和混合内核系统没有区别。这是不正确的，混合意味着它从单核和微核系统中都吸取了一定的设计模式，例如一些非关键的代码在用户空间运行，另一些在内核空间运行，单纯是为了效率的原因。

林纳斯 · 托瓦兹认为混合核心这种分类只是一种市场营销手法，因为它的架构实现与运作方式与宏内核并没有什么实质不同。

74.4 Exokernel

外内核系统，也被称为纵向结构操作系统，是一种比较极端的设计方法。

它的设计理念是让用户程序的设计者来决定硬件接口的设计。外内核本身非常的小，它通常只负责系统保护和系统资源复用相关的服务。

传统的内核设计 (包括单核和微核) 都对硬件作了抽象，把硬件资源或设备驱动程序都隐藏在硬件抽象层下。比方说，在这些系统中，如果分配一段物理存储，应用程序并不知道它的实际位置。

而外核的目标就是让应用程序直接请求一块特定的物理空间，一块特定的磁盘块等等。系统本身只保证被请求的资源当前是空闲的，应用程序就允许直接访问它。既然外核系统只提供了比较低级的硬件操作，而没有像其他系统一样提供高级的硬件抽象，那么就需要增加额外的运行库支持。这些运行库运行在外核之上，给用户程序提供了完整的功能。

理论上，这种设计可以让各种操作系统运行在一个外核之上，如 Windows 和 Unix。并且设计人员可以根据运行效率调整系统的各部分功能。

现在，外核设计还停留在研究阶段，没有任何一个商业系统采用了这种设计。

Chapter 75

Linux 内核

Linux 内核（Linux Kernel)[?] 是 Linux 操作系统的内核，符合 POSIX 标准并以 C 语言开发。Linux 最早是由芬兰黑客林纳斯 · 托瓦兹为尝试在英特尔 x86 架构上提供自由免费的类 Unix 系统而开发的。该计划开始于 1991 年，林纳斯 · 托瓦兹当时在 Usenet 新闻组 comp.os.minix 发表帖子，这份著名的帖子标示着 Linux 计划的正式开始。

从技术上说 Linux 是一个内核，“内核”指的是一个提供硬件抽象层、磁盘及文件系统控制、多任务等功能的系统软件。一个内核不是一套完整的操作系统，一套基于 Linux 内核的完整操作系统叫作 Linux 操作系统（或是 GNU/Linux)。

Linux 实质上是一个宏内核，设备驱动程序可以完全访问硬件，而且 Linux 内的设备驱动程序可以方便地以模块化（modularize）的形式设置，并在系统运行期间可直接装载或卸载。

Linux 是用 C 语言中的 GCC 版开发的，还有几个用汇编语言 (用的是 GCC 的 “AT&T 风格”) 写的目标构架短段。

因为要支持扩展的 C 语言，GCC 在很长的时间里是唯一一个能正确编译 Linux 的编译器。在 2004 年，Intel 主张通过修改内核来使它的编译器能正确编译内核。在内核构建过程中 (这里指从源代码创建可启动镜像) 可以使用许多其他的语言，包括 Perl、Python 和多种脚本语言。有一些驱动可能是用 C++、Fortran 或其他语言写的，虽然这样是强烈不建议的，Linux 的官方构建系统仅仅支持 GCC 作为其内核和驱动的编译器。

在 Linux 系统中，vmlinux（vmlinuz）是一个包含 Linux kernel 的静态链接的可执行文件[1]。

一般来说，核心的位置会在文件系统的/root 目录下，然而 Bootloader 必须使用 BIOS 的硬盘驱动程序，在一些 i386 的机器上必须要放在前 1024 个磁柱内。为了克服这个限制，Linux 发行版鼓励用户创建一个扇区用来存放 bootloader 与核心相关的开机文件（例

[1]vmlinux 若要用于除错时则必须要在开机前增加 symbol table。

有时候一些驱动程序原先并非为 Linux 设计，而是为其他操作系统而作（意即并非为 Linux 作的派生创作），这是个灰色地带……这“的确”是个灰色地带，而我个人相信一些模块可视为非 Linux 派生创作，是针对 Linux 设计，也因此不会遵守 Linux 订下的行为准则。

如 GRUB、LILO 与 SYSLINUX 等）。这个扇区通常会挂载到系统的/boot 上，这也是 FHS（Filesystem Hierarchy Standard0 标准内定义的。

Linux 内核是在 GNU 通用公共许可证第 2 版之下发布的（加上一些固件与各种非自由许可证）。原先托瓦兹将 Linux 置于一个禁止任何商业行为的条例之下，但之后改用 GNU 通用公共许可证第二版。该协议允许任何人对软件进行修改或发行，包括商业行为，只要其遵守该协议，所有基于 Linux 的软件也必须以该协议的形式发表，并提供源代码。

另外，关于 GPL v2 许可证争议的一个重点是 Linux 使用固件二进制包以支持某些硬件设备。理察 · 马修 · 斯托曼认为这些东西让 Linux 某部份成为非自由软件，甚至以此散布 Linux 更会破坏 GPL，因为 GPL 需要完全可获取的源代码。林纳斯 · 托瓦兹及 Linux 社区中的领导者，支持较宽松的许可证，不支持理察 · 马修 · 斯托曼的立场。社区中的 Linux-libre 提供完整的自由软件固件。

另一个争论点，就是加载式核心模块是否算是知识产权下的派生创作，意即 LKM 是否也受 GPL 约束？托瓦兹本人相信 LKM 仅用一部分“公开”的核心接口，因此不算派生创作，因此允许一些仅有二进制包裹的驱动程序或不以 GPL 声明的驱动程序用于核心。但也不是每个人都如此同意，且托瓦兹也同意很多 LKM 的确是纯粹的派生创作，也写下“基本上，核心模块是派生创作”这样的句子。另一方面托瓦兹也说过：

75.1 Loadable Kernel Module

可加载核心模块（Loadable kernel module，缩写为 LKM）[?] 又译为加载式核心模组、可装载模块、可加载内核模块，是一种目标文件（object file），在其中包含了能在操作系统内核空间运行的代码。

LKM 运行在核心基底（base kernel），通常是用来支持新的硬件、新的文件系统或是新增的系统调用（system calls）。当不需要时，它们也能从内存中被卸载，清出可用的内存空间。

Microsoft Windows 及类 UNIX 系统都支持 LKM 功能，只是在不同的操作系统中有不同的名称，比如 FreeBSD 称为核心加载模组（kernel loadable module，缩写为 kld），Mac OS X 称为核心扩充（kernel extension，缩写为 kext）。也有人称它为核心可加载模组（Kernel Loadable Modules，缩写为 KLM），或核心模组（Kernel Modules，KMOD）。

没有可加载模块时，操作系统需要将所有可能需要的功能，一次全加入核心之中。其中许多功能虽然占据着内存空间，但是从来没被使用过。这不但浪费内存空间，而且每次

在增加新功能时，使用者需要重新编译整个内核，之后重新开机。可加载模组避免了以上的缺点，让操作系统可以在需要新功能时可以动态加载，减少开发及使用上的困难。

75.2 Dynamic Kernel Module Support

动态内核模块支持（Dynamic Kernel Module Support，缩写为 DKMS)[?] 是用来生成 Linux 的内核模块的一个框架，其源代码一般不在 Linux 内核源代码树。当新的内核安装时，DKMS 支持的内核设备驱动程序到时会自动重建。DKMS 可以用在两个方向：如果一个新的内核版本安装，自动编译所有的模块，或安装新的模块（驱动程序）在现有的系统版本上，而不需要任何的手动编译或预编译软件包需要。例如，这使得新的显卡可以使用在旧的 Linux 系统上。

DKMS 是由戴尔的 Linux 工程团队在 2003 年开发的，并已经被许多 Linux 发行版所包含。

DKMS 是以 GNU 通用公共许可证（GPL）v2 或以后的条款发布下的免费软件，DKMS 原生支持 RPM 和 DEB 软件包格式。

DKMS 旨在创建一个内核相关模块源可驻留的框架，以便在升级内核时可以很容易地重建模块。这将允许 Linux 供应商提供较低版本的驱动程序，而无需等待新内核版本发行，同时还可以省去尝试重新编译新内核模块的客户预期要完成的工作。

Oikawa 等人在 1996 年提出一种与 LKM 类似的动态核心模块（DKMs）技术。与 LKM 一样，DKMs 以文件的形式存储并能在系统运行过程中动态地加载和卸载。DKMs 由一个用户层的 DKM 服务器来管理，并非由内核来管理。当核心需要某模块时，由 DKM 服务器负责把相应的 DKM 加载；当核心的内存资源紧缺时，由 DKM 服务器负责卸载一个没有被使用的 DKM。缺点是所有的 DKM 是存储在本地系统上的，占用了大量宝贵的存储空间。

75.3 Preemptive Scheduling

抢占式多任务处理（Preemption)[?] 是计算机操作系统中的一种实现多任务处理（multi task）的方式，相对于协作式多任务处理而言。协作式环境下，下一个进程被调度的前提是当前进程主动放弃时间片；抢占式环境下，操作系统完全决定进程调度方案，操作系统可以剥夺耗时长的进程的时间片，转而提供给其它进程。

- 每个任务赋予唯一的一个优先级（有些操作系统可以动态地改变任务的优先级）；
- 假如有几个任务同时处于就绪状态，优先级最高的那个将被运行；
- 只要有一个优先级更高的任务就绪，它就可以中断当前优先级较低的任务的执行；

75.4 Kernel Panic

在 Linux 中，内核错误 (Kernel panic) 是指操作系统在监测到内核系统内部无法恢复的错误，相对于在用户空间代码类似的错误。操作系统试图读写无效或不允许的内存地址是导致内核错误的一个常见原因。内核错误也有可能在遇到硬件错误或操作系统 BUG 时发生。在许多情况中，操作系统可以在内存访问违例发生时继续运行。然而，系统处于不稳定状态时，操作系统通常会停止工作以避免造成破坏安全和数据损坏的风险，并提供错误的诊断信息。

75.5 Kernel oops

在 Linux 上，oops 即 Linux 内核的行为不正确，并产生了一份相关的错误日志。许多类型的 oops 会导致内核错误，即令系统立即停止工作，但部分 oops 也允许继续操作，作为与稳定性的妥协。这个概念只代表一个简单的错误。

当内核检测到问题时，它会打印一个 oops 信息然后杀死全部相关进程。oops 信息可以帮助 Linux 内核工程师调试，检测 oops 出现的条件，并修复导致 oops 的程序错误。

Linux 官方内核文档中提到的 oops 信息被放在内核源代码 Documentation/oops-tracing.txt 中。通常 klogd 是用来将 oops 信息从内核缓存中提取出来的，然而，在某些系统上，例如最近的 Debian 发行版中，rsyslogd 代替了 klogd，因此，缺少 klogd 进程并不能说明 log 文件中缺少 oops 信息的原因。

若系统遇到了 oops，一些内部资源可能不再可用。即使系统看起来工作正常，非预期的副作用可能导致活动进程被终止。内核 oops 常常导致内核错误，若系统试图使用被禁用的资源。

Kernelloops 提到了一种用于收集和提交 oops 到 http://www.kerneloops.org/的软件，Kerneloops.org 同时也提供 oops 的统计信息。

计算机安全是一个非常公众化的主题，因为大量在内核中的错误可能成为潜在的安全漏洞，是否允许提升权限漏洞或拒绝服务攻击源漏洞。在过去一直有许多这样的缺陷被发现，并在 Linux 内核中被修补好。例如在 2012 年五月，SYSRET 指令被发现在 AMD 和英特尔处理器间在实现方面有差异，这个差异在 Windows、FreeBSD、XenServer 和 Solaris 这些主流操作系统会导致漏洞。2012 年六月，Linux 核心中该问题已被修复。

Chapter 76

内核模块

在操作系统启动的过程中，内核用来驱动主机的硬件及配置。一般内核都是压缩文件，因此在启动过程中会把内核解压缩后才能加载到内存中。

现代操作系统内核都具有可读取模块化驱动程序的功能，所谓的模块化可以理解为“插件”，模块可能由硬件厂商提供，也可能由内核提供。一般情况下，Linux 的内核文件位于/boot/vmlinuz 或/boot/vmlinuz-version，内核解压缩所需的 RAM Disk 为/boot/initrd 或/boot/initrd-version。

Linux 内核整体结构包含了很多的模块，有两种方法将需要的功能包含进内核当中，分别是：

- 将所有的功能都编译进 Linux 内核，这样不会有版本不兼容的问题，不需要进行严格的版本检查，但同时也会导致生成的内核会很大，而且要在现有的内核中添加新的功能时，就需要编译整个内核。
- 将需要的功能编译成模块，在需要的时候动态地添加，这样模块本身不编译进内核，从而控制了内核的大小，而且模块一旦被加载后，将和其它的部分完全一样。缺点是可能会有内核与模块版本不兼容的问题，导致内核崩溃，而且还会造成内存的利用率比较低。

目前，Linux 内核采用的是模块化技术，这样的设计使得系统内核可以保持最小化，同时确保了内核的可扩展性与可维护性，模块化设计允许我们在需要时才将模块加载至内核，实现动态内核调整[?]。

内核模块一般位于/lib/modules/version/kernel 或/lib/modules/$(uname -r)/kernel 中，内核源码则位于/usr/src/linux 或/usr/src/kernels。内核被加载到系统中后会产生相关的记录如下：

- 内核版本：/proc/version
- 系统内核功能：/proc/sys/kernel

如果要增加新的内核模块，首先需要将其编译为模块，然后通过 modprobe 等命令来加载。如果希望系统开机自动挂载内核模块则需要将 modprobe 命令写入/etc/rc.sysinit 文件中。

Linux 内核模块文件的命名方式通常为 < 模块名称.ko>，CentOS 6.3 系统的内核模块被集中存放在/lib/modules/`uname -r`/目录下（这里，uname -r 获得的信息为当前内核的版本号）。

与内核模块加载相关的配置文件是/etc/modules.conf 或 /etc/modprobe.conf[?]。在配置文件中，一般是写入模块的加载命令或模块的别名的定义等，比如在 modules.conf 中可能会发行类似的一行：

```
alias eth0 8139too
```

76.1 lsmod

lsmod 命令用来显示当前系统中加载的 Linux 内核模块状态，不是使用任何参数会显示当前已经加载的所有内核模块。输出的三列信息分别为模块名称、占用内存大小、是否在被使用，如果第三列为 0 则该模块可以随时卸载，非 0 则无法执行 modprobe 删除模块。

```
[root@localhost ~]#lsmod
Module                  Size  Used by
loop                   27870  2
vfat                   17411  0
fat                    60923  1 vfat
nls_utf8               12557  2
isofs                  39794  2
tcp_lp                 12663  0
fuse                   86889  3
...
```

lsmod 命令可以和 grep 命令结合使用，例如：

```
[root@localhost ~]#lsmod | grep mei
mei_me                 18581  0
mei                    76861  1 mei_me
```

另外还可以通过查看 /proc/modules 来知道系统已经加载的模块。

76.2 depmod

通过 depmod 命令可以了解内核提供的模块之间的依赖关系。所谓的依赖关系，指的是加载指定的模块之前，必须要首先加载其所依赖的其他模块，该模块加载后才能正常工作。

depmod 会依据相关目录的定义将所有模块进行分析，并将分析的结果写入 modules.dep 文件，而且这个文件还将影响到 modprobe 命令的使用。

内核模块目录的结构如下：

```
arch #与硬件平台有关的选项，例如 CPU 的等级等；
crypto #内核所支持的加密的技术，例如 MD5 或 DES 等；
drivers #一些硬件的驱动程序，例如显卡、网卡、PCI 相关硬件等；
fs #内核所支持的文件系统，例如 vfat、reiserfs 等；
lib #函数库；
net #与网络有关的协议数据以及防火墙模块 (net/ipv4/netfilter/*) 等；
sound #与音效有关的模块。
```

Linux 使用/lib/modules/$(uname -r)/modules.dep 文件来记录模块之间的依赖关系，可以通过 depmod 命令来创建该文件，因此 Linux 内核是自动解决依赖关系的。

```
#为所有列在/etc/modprobe.conf 或/etc/modules.conf 中的所有模块创建依赖关系，并
    且写入到 modules.dep 文件；
[root@localhost beinan]#depmod -a
#列出已挂载但不可用的模块；
[root@localhost beinan]#depmod -e
#列出所有模块的依赖关系
[root@localhost beinan]#depmod -n
```

76.3 modprobe

modprobe 命令可以动态加载与卸载指定的内核模块，或是载入一组相依赖的模块。

modprobe 会根据 depmod 所产生的依赖关系，决定要载入哪些模块。若在载入过程中发生错误，在 modprobe 会卸载整组的模块。依赖关系是通过读取 /lib/modules/2.6.xx/modules.dep 得到的，而该文件是通过 depmod 建立的。

在/etc/modprobe.conf 文件中存在的内容形式如下：

```
alias scsi_hostadapter mptbase
alias scsi_hostadapter1 mptspi
...
```

其中，最后一列是模块名字，中间的是模块的别名。如果要通过模块的名字来查询它的别名，可以用下面的命令：

```
#modprobe -c
```

当然 modprobe 也有列出内核所有模块和移除模块的功能，其中 `modprobe -l` 是列出内核中所有的模块，包括已挂载和未挂载的。模块名是不能带有后缀的，实际上模块都是带有.ko 或.o 后缀。

通过 modprobe -l 可以查看到所需要的模块，然后根据需要来挂载，实际上 modprobe -l 读取的模块列表就位于/lib/modules/`uname -r`目录中，这里 uname -r 是内核的版本。

```
# modprobe ip_vs     #动态加载ip_vs模块
# lsmod |grep ip_vs  #查看模块是否加载成功
# modprobe -r ip_vs  #动态卸载ip_vs模块
```

通过上述 modprobe 方式加载的内核模块仅在当前有效，计算机重启后并不会再次加载该模块，如果希望系统开机自动挂载内核模块则需要将 modprobe 命令写入/etc/rc.sysinit 文件中：

```
# echo "modprobe ip_vs" >> /etc/rc.sysinit
```

当内核模块不再需要时可以通过将/etc/rc.sysinit 文件中的对应 modprobe 命令删除，但需要重启计算机才生效。此时，可以通过 modprobe -r 命令来立刻删除内核模块：

```
# modprobe -r ip_vs
```

76.4 insmod

将模块插入内核中。

与 modprobe 相比，insmod 完全是由用户自行加载一个完整路径和文件后缀名的模块，并不会主动去分析模块依赖性，而 modprobe 则会去主动分析 modules.dep 后才加载模块，这样既可以克服模块的依赖性问题，而且还不需要了解模块的具体路径。

```
[root@localhost ~]#insmod xxx.ko
```

76.5 rmmod

将模块从内核中删除。

```
[root@localhost ~]#rmmod xxx.ko
```

76.6 modinfo

modinfo 命令可以查看内核模块信息，通过查看模块信息来判定这个模块的用途。

```
[root@localhost ~]# modinfo ip_vs
filename:     /lib/modules/3.13.8-200.fc20.x86_64/kernel/net/netfilter/ipvs/ip_vs.ko
license:        GPL
depends:        nf_conntrack,libcrc32c
intree:         Y
vermagic:       3.13.8-200.fc20.x86_64 SMP mod_unload
signer:         Fedora kernel signing key
sig_key:        68:89:8C:AD:02:C5:A1:BA:89:74:28:C5:5E:53:C8:F9:0C:BC:BB:8C
sig_hashalgo:   sha256
parm:           conn_tab_bits:Set connections' hash size (int)
```

事实上，modinfo 除了可以查阅内核内的模块之外，还可以检查其他模块文件。

76.7 tree

查看当前目录的整个树结构。

```
[root@localhost ~]#tree -a
```

76.8 临时调整内核参数

Linux 内核参数随着系统的启动会被写入内存中，我们可以直接修改/proc 目录下的大量文件来调整内核参数，并且这种调整是立刻生效的。

```
# 开启内核路由转发功能（通过0或1设置开关）
[root@localhost ~]# echo "1" > /proc/sys/net/ipv4/ip_forward
[root@localhost ~]# echo "1" > echo "1" >t /proc/sys/net/ipv4/icmp_echo_ignore_all
```

```
#调整所有进程总共可以打开的文件数量（当大量的用户访问网站资源时可能会因该数字过
    小而导致错误）
[root@localhost ~]#echo "108248" >/proc/sys/fs/file-max
```

76.9 永久调整内核参数

可以通过 man proc 可以获得大量关于内核参数的描述信息。但以上通过直接修改/proc 相关文件的方式在系统重启后将不再有效，如果希望设置参数并永久生效可以修改/etc/sysctl.conf 文件，文件格式为选项 = 值，我们通过修改该文件将前面 3 个案例参数设置为永久有效：

```
[root@localhost ~]#vim /etc/sysctl.conf
net.ipv4.ip_forward = 1
net.ipv4.icmp_echo_ignore_all = 1
fs.file-max = 108248
```

注意，通过 sysctl.conf 文件修改的内核参数不会立刻生效，修改完成后使用 `sysctl -p` 命令可以使这些设置立刻生效。

76.10 内核模块程序结构

内核模块程序结构[?] 主要由加载、卸载函数功能函数等一系列声明组成。它可以被传入参数，也可以导出符号，供其它的模块使用。

76.10.1 模块加载函数

在用 insmod 或 modprobe 命令加载模块时，一般需要执行模块加载函数以完成模块的初始化工作。

Linux 内核的模块加载函数一般用 __init 标识声明，模块加载函数必须以 module_init(函数名) 的形式被指定。该函数返回整型值，如果执行成功，则返回 0，初始化失败时则返回错误编码，Linux 内核当中的错误编码是负值，在 <linux/errno.h> 中定义。

在 Linux 中，标识 __init 的函数在连接时放在.init.text 这个区段，而且在.initcall.init 中保留一份函数指针，初始化的时候内核会根据这些指针调用初始化函数，初始化结束后释放这些 init 区段（包括前两者）。

代码清单：

```
static int __init XXX_init(void)
{
   return 0;
}


moudle_init(XXX_init);
```

76.10.2 模块卸载函数

在用 rmmod 或 modprobe 命令卸载模块时，一般需要执行模块卸载函数来完成与加载相反的工作。

模块的卸载函数和模块加载函数实现相反的功能，主要包括：

- 若模块加载函数注册了 XXX，则模块卸载函数注销 XXX；

- 若模块加载函数动态分配了内存，则模块卸载函数释放这些内存；
- 若模块加载函数申请了硬件资源，则模块卸载函数释放这些硬件资源；
- 若模块加载函数开启了硬件资源，则模块卸载函数一定要关闭这些资源。

```
static void __exit XXX_exit(void)
{

}

moudle_exit(XXX_exit);
```

76.10.3 模块许可证声明

模块许可证必须声明，如果不声明，则在模块加载时会收到内核被污染的警告，一般应遵循 GPL 协议。

```
MODULE_LICENSE("GPL");
```

76.10.4 模块参数

模块参数是模块在被加载时传递给模块的值，本身应该是模块内部的全局变量，是可选的。

```
/*
* book.c
*/
#include <linux/init.h>
#include <linux/module.h>

static char *bookName = "Good Book.";
static int bookNumber = 100;

static int __init book_init(void)
{
   printk(KERN_INFO "Book name is %s\n", bookName);
   printk(KERN_INFO "Book number is %d\n", bookNumber);

    return 0;
}
```

```
static void __exit book_exit(void)
{
   printk(KERN_INFO "Book module exit.\n");
}

module_init(book_init);
module_exit(book_exit);
module_param(bookName, charp, S_IRUGO);
module_param(bookNumber, int, S_IRUGO);

MODULE_LICENSE("GPL");
```

在向内核插入模块的时候可以用以下方式，并且可以在内核日志中看到模块加载以后变量已经有了值。

```
[root@localhost ~]#rmod book.ko
[root@localhost ~]#insmod book.ko bookName="The World is Plat"
    bookNumber="100"
[root@localhost ~]#tail -n /var/log/kern.log
Book name is The World is Plat
Book number is 100
```

76.10.5 模块导出符号

模块导出符号是可选的。

使用模块导出符号，方便其它模块依赖于该模块，并使用模块中的变量和函数等。

在 Linux2.6 的内核中，/proc/kallsyms 文件对应着符号表，它记录了符号和符号对应的内存地址。对于模块而言，使用下面的宏可以导出符号。

```
EXPORT_SYMBOL(符号名);
```

或

```
EXPORT_GPL_SYMBOL(符号名);
```

76.10.6 模块信息

模块信息是可选的，用于存储模块的作者信息等。

76.11 模块使用计数

Linux 内核提供了 MOD_INC_USE_COUNT 和 MOD_DEC_USE_COUNT 宏来管理模块使用计数。但是对于内核模块而言，一般不会自己管理使用计数。

76.12 模块的编译

将下面的 Makefile 文件放在 book.c 同级的目录下，然后使用 `make` 命令或 `make all` 命令编译即可生成 book.ko 模块文件。

对应的 Makefile：

```
ifneq ($(KERNELRELEASE),)
mymodule_objs := book.o
obj-m := book.o
else
PWD := $(shell pwd)
KVER ?= $(shell uname -r)
KDIR := /usr/src/linux-headers-2.6.38-8-generic

all:
	$(MAKE) -C $(KDIR) M=$(PWD)

clean:
	rm -rf *.mod.c *.mod.o *.ko *.o *.tmp_versions *.order *symvers
endif
```

如果功能不编译成模块，则无法绕开 GPL，编译成模块后公司发布产品则只需要发布模块，而不需要发布源码。为了 Linux 系统能够支持模块，需要做以下的工作：

- 内核编译时选择“可以加载模块”，嵌入式产品一般都不需要卸载模块，则可以不选择“可卸载模块”
- 将我们的 ko 文件放在文件系统中
- Linux 系统实现了 insmod、rmmod 等工具
- 使用时可以用 insmod 手动加载模块，也可以修改/etc/init.d/rcS 文件，从而在系统启动的时候就加载模块。

Part XIV

Performance

Part XV

Network

Chapter 77

Introduction

77.1 LAN

在同一个网段内的所有联机的主机及网络设备组成了局域网（LAN，Local Area Network）。

另外，同一个物理网段内可以借助网络地址来划分更多不同的 IP 网段，因此局域网就是在同一个物理网段环境中，使用私有 IP 或者局域网适用的通信协议组成的网络环境。

一般来说，内部局域网都希望直接使用私有 IP 来设置网络环境，。

- 简单的局域网可以使用 Hub 或 Switch 来获得 Public IP 联网。
- 复杂的局域网可以使用 NAT（Network Address Translation）服务器来联网。

77.2 WAN

Chapter 78

Protocol

网络通信协议就是一些用于确保数据通过网络设备进行正确无误地传送和接收的标准和规则。

TCP/IP 协议规定了 IP 基础和路由协议等信息，例如 SAMBA 协议也是通过 Net BIOS over TCP/IP 协议来实现数据传输的。

在连接 Internet 时，需要设置的网络参数包括 IP、Netmask、Network、Broadcast、Gateway 以及 DNS 等。

78.1 TCP

在实际应用中，为了在主机和客户端之间传输信息，按照实现的方式的不同，可以采取的方式包括：

- 利用浏览器或 URL 类的格式将数据从服务器获取到本地；
- 利用网络服务程序使用 Socket 的形式获取数据；
- 利用 RMI 远程调用的方法的来获取数据并交互。

78.2 IP

Chapter 79

Networking

79.1 Configuration

Linux 系统中 IP、主机名等信息都可以通过修改相应的配置文件来完成。

- /etc/sysconfig/network 用于设置主机名[1]及启动网络接口。
- /etc/sysconfig/network-scripts/ifcfg-ethX[2]用于设置网卡参数，包括 Network、IP、Netmask、Broadcast、Gateway 以及获取 IP 的方式（DHCP、Static）和是否开机自动启动。
- /etc/modprobe.conf 用于设置内核模块的加载，而且网卡驱动程序一般都是被编译为模块并在/etc/modproble.conf 中设置为在开机时自动进行加载。
- /etc/resolv.conf 用于设置 DNS，其中保存了当前提供网络服务的 IP。

```
# Generated by NetworkManager
nameserver 219.233.241.166
nameserver 211.167.97.67
nameserver 192.168.1.1
# NOTE: the libc resolver may not support more than 3 nameservers.
# The nameservers listed below may not be recognized.
nameserver 8.8.8.8
```

- /etc/hosts 记录本机 IP 对应主机名或主机别名，在局域网中进行反查时就需要使用主机名对应 IP 的数据。
  在网络联机时，局域网内部的主机之间会互相询问对方的主机名称来确认对方身份。DNS 服务器在解析主机名和 IP 的对应时无法处理局域网内部使用的私有 IP，因此需要在/etc/hosts 中直接写入主机名和 IP 的对应关系，这样当局域网内部的访问/etc/hosts 中指定的 IP 时就会直接联机，不需要再进行 DNS 解析。

[1] SuSE 中的主机名记录在/etc/HOSTNAME 中。
[2] SuSE 中网卡配置文件为/etc/sysconfig/network/ifcfg-ethX。

/etc/hosts 包含主机 IP、主机名和主机别名。

```
# cat /etc/hosts
127.0.0.1 localhost.localdomain localhost
::1 localhost6.localdomain6 localhost6
```

一般情况下，当需要解析主机名和 IP 的对应关系时，系统首先检查/etc/hosts 来查找对应的设置值，然后再使用/etc/resolv.conf 中的设置到 DNS 服务器中查找。

- /etc/services 记录在 TCP/IP 协议上的所有协议（包括 HTTP、FTP、SSH、Telnet 等）所使用的端口号。
- /etc/protocols 可以定义 IP 数据包协议的相关数据，包括 ICMP、TCP、UDP 等协议的数据包的定义等。

为了启动网络，可以使用/etc/init.d/network 主动去读取所有的网络设置文件，同时可以恢复对系统设置的异常更改。

```
# /etc/init.d/network restart
```

ifup/ifdown 等可以用来启动或者停止某个网络接口，它们主动读取/etc/sysconfig/network-scripts/目录下的配置文件（例如 ifcfg-eth0）。

```
# ifup eth0
# ifdown eth0
```

在设置网络接口的参数时，要注意 DEVICE、BOOTPROTO、GATEWAY、GATEWAYDEV 和 HWADDR 等参数。

- DEVICE 表示网卡设备的设备代号；
- BOOTPROTO 表示启动网络接口时使用的协议，可选项是 static、none、dhcp 等。
- GATEWAY 表示整个主机系统的 Default Gateway，而且在设置需要保证没有重复。
- GATEWAYDEV 表示用作网关的设备（例如路由器等）。
- HWADDR 表示网卡的物理地址。

79.2 Gateway

默认网关通常仅有一个，用来作为同一网段的其他主机传递非本网段的数据包网关，不过在每个网络设置文件（/etc/sysconfig/network-scripts/ifcfg-ethX）内部都可以指定 GATEWAY 参数。

在非拨号连接情况下，不要在 ifcfg-ethX 中指定 GATEWAY 或 GATEWAYDEV 等参数。

79.3 PPPoE

rp-pppoe 软件可以用于 ADSL 拨号，然后再通过 ADSL 调制解调器的电话线路连接到 ISP 服务器来连接外网。

实际上，使用拨号连接时就是在主机上额外增加一个实体的 ppp0 接口，通过设置 ifcfg-xxx 文件可以配置网络参数，而且网卡可以绑定多个 IP。

rp-pppoe 使用 Point-to-Point（ppp）的点对点协议产生的网络接口，并且在连通后产生实体网络接口 ppp0。

ppp0 架构在以太网上，因此 eth0 也要启动，并且拨号成功后产生了如下三个完全独立的接口：

- 内部循环测试用的 lo 接口；
- 网卡 eth0 接口；
- 拨号连接产生的与 ISP 对接的 ppp0 接口。

79.3.1 adsl-setup

在 adsl-setup 命令中设置登录外网的用户名以及连接 ADSL 调制解调器的网卡代号等，并将相关的设置写入/etc/sysconfig/network-scripts/ifcfg-ppp0 中。

- /etc/sysconfig/network-scripts/ifcfg-ppp0
- /etc/resolv.conf
- /etc/ppp/chap-secrets
- /etc/ppp/pap-secrets

79.3.2 adsl-start

79.3.3 adsl-stop

79.4 Wireless

在 RJ-45 的以太网环境中，由 switch/hub、网卡和网线等组成了以 switch/hub 串接所有网络设备的架构。

无线网络的中线是无线访问点（wireless access point），其本身是一个 IP 转发设备，不仅可以与外网通信，而且提供内部网络访问外网的网关。

- WEP
- AES
- WPA-PSK
- WPA2-PSK

79.4.1 iwconfig

iwconfig 与 ifconfig 类似，可以用来检测网卡的相关信息。

```
# iwconfig
enp0s25   no wireless extensions.

wlp16s0   IEEE 802.11abgn  ESSID:"theqiong"
          Mode:Managed  Frequency:2.437 GHz  Access Point: 08:10:77:98:C2:B9
          Bit Rate=144.4 Mb/s   Tx-Power=14 dBm
          Retry short limit:7   RTS thr:off   Fragment thr:off
          Encryption key:off
          Power Management:off
          Link Quality=57/70  Signal level=-53 dBm
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:60906  Invalid misc:352   Missed beacon:0

lo        no wireless extensions.
```

79.4.2 iwlist

iwlist 命令可以用于搜索无线接入点。

```
# iwlist
Usage: iwlist [interface] scanning [essid NNN] [last]
              [interface] frequency
              [interface] channel
              [interface] bitrate
              [interface] rate
              [interface] encryption
              [interface] keys
              [interface] power
              [interface] txpower
              [interface] retry
              [interface] ap
              [interface] accesspoints
              [interface] peers
              [interface] event
```

```
[interface] auth
[interface] wpakeys
[interface] genie
[interface] modulation
```

79.5 dig

79.6 host

79.7 ifconfig

Linux 系统中默认的网卡文件名为 eth0，并以此类推，用户可以通过 ifconfig 获得网卡的相关信息。

```
# ifconfig {interface} {up|down}
```

ifconfig 命令可以用来查询、设置和调整网卡和 IP 网段等相关参数。

```
# ifconfig
enp0s25: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether 00:1e:ec:1e:25:ae  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 22  memory 0xe8020000-e8040000

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 0  (Local Loopback)
        RX packets 15756  bytes 27715616 (26.4 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 15756  bytes 27715616 (26.4 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

wlp16s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
```

```
inet 192.168.1.16  netmask 255.255.255.0  broadcast 192.168.1.255
inet6 fe80::21f:3bff:fe58:192f  prefixlen 64  scopeid 0x20<link>
ether 00:1f:3b:58:19:2f  txqueuelen 1000  (Ethernet)
RX packets 479583  bytes 638092457 (608.5 MiB)
RX errors 0  dropped 0  overruns 0  frame 0
TX packets 487725  bytes 58399671 (55.6 MiB)
TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

- HWaddr：网卡的硬件地址（即 MAC）；
- inet：IPv4 地址；
- inet6：IPv6 地址；
- RX：由启动到目前为止接收的数据包；
- TX：由启动到目前为止发送的数据包；

其中，packets 代表数据包数目。

- errors 代表数据包发生错误的数目；
- dropped 代表丢弃的数据包数量，
- collisions 代表发生数据碰撞的数目；
- txqueuelen 代表用于数据传输的缓冲区的长度；
- overruns
- carrier
- frame

ifconfig 命令可以直接手动指定网络接口的参数，包括 IP 参数、MTU、子网掩码（netmask）和广播（broadcast）等。

```
# ifconfig interface {options}
```

```
# ifconfig wlp16s0 192.168.1.22
# ifconfig wlp16s0 192.168.1.22 netmask 255.255.255.0 mtu 8000
```

ifconfig 可以为网卡绑定多个 IP 地址，即在物理网络接口基础上继续仿真一个网络接口。

```
# ifconfig wlp16s0:0 192.168.1.23
```

ifconfig 可以用来手动设置或修改网络接口参数，也可以设置虚拟网络接口来绑定多个 IP 地址，并且可以通过重启网络接口来注销设置。

网卡需要内核支持才能工作，因此用户需要通过编译内核或编译网卡的内核模块来驱动网卡。

用户可以使用 dmesg 命令来检查内核是否支持网卡，或者使用 lspci 获得网卡的相关模块。

```
# dmesg | grep -in eth
446:[   1.484496] e1000e 0000:00:19.0 eth0: (PCI Express:2.5GT/s:Width x1) 00:1e:ec:1e:25:ae
447:[   1.484499] e1000e 0000:00:19.0 eth0: Intel(R) PRO/1000 Network Connection
448:[   1.484536] e1000e 0000:00:19.0 eth0: MAC: 6, PHY: 6, PBA No: 1032FF-0FF
449:[    1.503441] e1000e 0000:00:19.0 enp0s25: renamed from eth0
450:[   1.514160] systemd-udevd[238]: renamed network interface eth0 to enp0s25
742:[    7.660128] Bluetooth: BNEP (Ethernet Emulation) ver 1.3
```

与网卡驱动相关的内核模块存放在/lib/modules/`uname -r`/kernel/drivers/net/中。

79.7.1 lsmod

lsmod 可以输出相关的模块名称

79.7.2 modinfo

modinfo 命令可以用于查看相关模块的信息。

79.7.3 depmod

depmod 命令可以用于对模块进行分析来建立关联文件。

79.7.4 modprobe

CentOS 等操作系统使用 modprobe.conf 来指定硬件代号与模块的对应。

79.8 ifup

ifup 脚本可以用来启动网络接口，仅针对/etc/sysconfig/network-scripts 内的 ifcfg-ethx（x 为数字）执行启动操作，不能直接修改网络参数，除非手动修改 ifcfg-ethx 文件。

```
# ifup {interface}
```

实际上，ifup 通过在/etc/sysconfig/network-scripts/目录下搜索对应的配置文件（例如 ifcfg-eth0）来启动网络接口。

79.9 ifdown

ifdown 脚本可以用来关闭网络接口，仅针对/etc/sysconfig/network-scripts 内的 ifcfg-ethx（x 为数字）执行关闭操作，不能直接修改网络参数，除非手动修改 ifcfg-ethx 文件。

```
# ifdown {interface}
```

实际上，ifdown 通过在/etc/sysconfig/network-scripts/目录下搜索对应的配置文件（例如 ifcfg-eth0）来关闭网络接口。

79.10 route

一般情况下，网络接口都可以产生一个路由，route 命令可以用来查询和设置路由表（route table）。

```
# route [-nee]
```

- -n 表示不使用通信协议或主机名，直接使用 IP 或 Port number；
- -ee 输出路由的详细信息

79.11 ip

ip 命令是 ifconfig、ifup、ifdown 和 route 等命令的复合形式，可以直接执行与网络相关的操作。

79.12 inetd

79.13 netcat

79.14 netstat

79.15 nslookup

如果设置了防止转发 DNS 的防火墙规则，可以使用 nslookup 命令进行检查。

```
# nslookup theqiong.com
Server: 211.167.97.67
Address: 211.167.97.67#53
```

```
Non-authoritative answer:
Name: theqiong.com
Address: 106.187.48.133
```

另外，CentOS 会主动在/etc/dhClient-eth0.conf 中创建说明文件。

79.16 ping

79.17 rdate

79.18 rlogin

79.19 route

79.20 ssh

79.21 traceroute

Chapter 80

NFS

Chapter 81

NIS

Chapter 82

DHCP

Chapter 83

HTTP

Chapter 84

FTP

Chapter 85

Samba

Chapter 86

NTP

Chapter 87

Gateway

Chapter 88

Wireless Network

Chapter 89

Bluetooth

Chapter 90

Bridging

Chapter 91

NAS

Chapter 92

ATM

Part XVI

Virtualization

Part XVII

Security

Chapter 93

Introduction

Chapter 94

Password

为了实施密码安全策略，可以修改/etc/login.defs 中的密码规则来要求用户定期更改密码，以及使用一定的密码规则（例如密码长度和混合模式等)。

另外，还可以通过 PAM 模块来额外地进行密码的验证工作。

Chapter 95

Permission

为了给系统中的不同用户指定不同的权限，可以在/etc/security/limits.conf 中对每个用户的权限进行规范。

Chapter 96

Firewall

xinetd 和 tcp wrappers 都可以用来管理服务的权限，从而可以直接使用/etc/hosts.allow 和/etc/hosts.deny 来管理服务的权限，并对 IP、网段和网域等进行权限设置来实现有限的信任网段。

除了/etc/hosts.allow 和/etc/hosts.deny 之外，还可以使用防火墙机制 iptables 来配置主机的防火墙。

- Linux Kernel 2.4 及以上版本可以使用 iptables；
- Linux Kernel 2.2 及以下版本可以使用 ipchains。

Chapter 97

SELinux

Chapter 98

MAC