LINUX 内核精髓

精通LINUX内核必会的75个绝技

LINUX KERNEL HACKS™

高桥 浩和 主编

池田 宗广、大岩 尚宏、岛本 裕志
竹部 晶雄、平松 雅巳 著

杨婷 译

刘波 审校

O'REILLY®
机械工业出版社
China Machine Press
华章科技

Linux 内核精髓

精通 Linux 内核必会的 75 个绝技

高桥 浩和 主编
池田 宗广、大岩 尚宏、岛本 裕志
竹部 晶雄、平松 雅巳 著
杨婷 译
刘波 审校

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc. 授权机械工业出版社出版

机 械 工 业 出 版 社
China Machine Press

图书在版编目（CIP）数据

Linux 内核精髓：精通 Linux 内核必会的 75 个绝技 /（日）高桥 浩和等著；杨婷译 . —北京：机械工业出版社，2013.1

（O’Reilly 精品图书系列）

书名原文：Linux Kernel Hacks

ISBN 978-7-111-41049-2

I. L… II. ① 高… ② 杨… III. Linux 操作系统 IV. TP316.89

中国版本图书馆 CIP 数据核字（2012）第 318908 号

北京市版权局著作权合同登记

图字：01-2011-7458 号

书　　名/　Linux内核精髓：精通Linux内核必会的75个绝技
书　　号/　ISBN 978-7-111-41049-2
责任编辑/　谢晓芳
出版发行/　机械工业出版社
地　　址/　北京市西城区百万庄大街22号（邮政编码　100037）
印　　刷/
开　　本/　178毫米×233毫米　16开本　26.5印张
版　　次/　2013年2月第1版　2013年2月第1次印刷
定　　价/　79.00元（册）

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：(010)88378991；88361066
购书热线：(010)68326294；88379649；68995259
投稿热线：(010)88379604
读者信箱：hzjsj@hzbook.com

O'Reilly Media, Inc. 介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

编者与作者介绍

主编简介

高桥 浩和（Hirokazu Takahashi）

毕业于北海道大学电子工学系。从 VAX 全盛时代开始致力于各种 UNIX 系列操作系统的功能强化和内核调整，以及大规模系统的实时操作系统的设计等。以 ISP 的服务器构建为契机，开始正式研究 Linux。

作者简介

池田 宗广（Munehiro IKEDA）

大学时代，亲眼看到 X68000 的 gcc 生成比主流编译器还要快好几倍的代码，因此开始确信免费软件 / 开源软件的可能性。此后，在历经咖啡店店员、生产技术人员、硬件工程师后，终于开始从事 Linux 内核开发。这个行业最吸引人的就是能够跨公司甚至跨国界与世界最优秀的技术人员进行交流。现居住在美国，爱好音乐演奏，当过鼓手，也当过主唱，最近几年一直在弹贝斯。不管是作为技术人员还是贝斯手都喜欢做幕后工作，只不过天生就不喜欢半途而废。

大岩 尚宏（Naohiro Ooiwa）

任职于 Miracle Linux 株式会社的软件工程师。大学时研究的是类似手机这样使用天线接收无线高频信号的模拟线路。从事 Linux 开发工作的时候开始深入研究软件。他是《Debug Hacks》的作者，本书是 O'REILLY JAPAN 的第二本 Hacks 系列图书。

岛本 裕志（Hiroshi Shimamoto）

软件工程师。负责问题分析和调试。主要工作就是在出现故障时，根据日志和核心转储找出问题所在。因此在工作中会经常用到二进制和 CPU 运行的知识。同时也在论坛中从事过一些关于 x86 架构和调度程序的活动。目前关注虚拟化方面的活动。

竹部 晶雄（Akio Takebe）

在 Xen、KVM 等与虚拟化相关的开源论坛参与开发活动。主要负责 IA64 架构、RAS 系列和 PCI pass through 的开发。在开源论坛认识了专门研究省电技术的工程师，从而开始对省

电方面产生兴趣。现在正使用 Ruby on Rails 开发云计算相关软件。

平松 雅巳（Masami Hiramatsu）

Linux 内核追踪的相关维护人员。主要工作是对 perf 和 ftrace 的动态事件进行维护。也参与了 SystemTap 的开发，最近热衷于将系统 SystemTap 的用途从专门用于追踪扩展到游戏编程等。主要使用的是 bash 和 vim，但是因为 bash 不能用 hjkl 移动光标，总的来说属于 vim 用户。喜欢使用 Ubuntu 和 Fedora。现在的研究方向是 ARM Linux、Btrfs 等。

撰稿人简介

畑山 大辅（HATAYAMA Daisuke）

crash gcore 扩展模块的维护人员。对调试和故障分析感兴趣。最喜欢做的事情就是从元数据对系统进行研究。正在努力练习马拉松长跑，争取在搞技术的同时锻炼出健康的体魄。近期目标是四小时内跑完马拉松。

藤田 朗（Akira Fujita）

任职于 NEC 软件东北株式会社。担任软件工程师。大学毕业之后开始转向软件行业。喜欢 Linux 文件系统（ext3/ext4）。喜欢 defrag，爱好五人足球。

技术审校者简介

刘波，资深 Linux 内核开发工程师、应用开发工程师和嵌入式开发工程师，现在重庆工商大学计算机科学与信息工程学院担任教师，从事 Linux 程序开发和 Oracle 管理方面的教学工作，在读博士。此外，他还专注于大规模机器学习、数值分析与计算、最优化理论（凸优化）的研究。

致谢

本书的编著工作受到了多方的大力支持。计划还未确定时，O’REILLY JAPAN 就对此表示了很大的兴趣，并给予我们参与写作的机会，对此我们要向 O’REILLY JAPAN 的各位表示衷心的感谢。特别是受到担任编辑的赤池凉子女士的很多照顾。写作进度滞后时，赤池女士依旧迅速地安排了后面的多次修改，非常感谢她。

对作为作者参与写作的池田宗广、平松雅巳先生，以及非常爽快地同意作为撰稿人紧急参与写作的藤田朗、畑山大辅先生也要表示衷心的感谢。在各位的辛勤努力下，本书才能够具有如此丰富和引人入胜的内容。

对百忙之中抽出时间负责主编的高桥浩和先生也表示衷心的感激。对写作过程中的技术

趋势以及所有章节都进行了详细的指导。也非常感谢三好和人、永野武先生为我们免费提供很多原稿。

此外，非常感谢《Debug Hacks》的作者安部东洋，为本书进行指导，使得本书质量大幅提高。

在从一开始就共同执笔的岛本裕志、竹部晶雄先生的努力下，本书才得以顺利出版。感谢你们。

最后，借此机会向本书写作过程中为我们提供协助的各方人士表示衷心的感谢。

大岩 尚宏

主编致辞

从 1991 年 Linux 内核诞生，到现在已经过去了 20 多年，现在 Linux 3.0 也即将发布。在这 20 多年间，Linux 内核已经进化成可以在便携式计算机到大型服务器的各种硬件上运行的操作系统。至今仍有众多开发人员在不断地对 Linux 内核进行开发。

虽然网络上有很多关于 Linux 的信息，但是关于熟练使用 Linux 内核或者参与 Linux 内核开发所需的信息并不多。因此我们决定从这些信息中筛选出 Linux 技术人员可能感兴趣的内容，汇编成一本书。关于省电和虚拟化的介绍是非常符合当前市场需求的。高级内核的概要分析功能也是内核开发人员的必需工具。

在有限的篇幅内能够介绍的内容并不是很多，我们只是希望能够以此为契机，激发更多人对 Linux 内核的兴趣，并实际参与内核开发。

高桥 浩和

前言

内核是操作系统的核心，操作系统的基本功能都是由内核提供的。文件生成和数据包传输等也是通过内核的功能实现的。但这些都不是简单的任务。平时可能意识不到，但这其中确实包含了很多先进技术。例如，在文件系统方面，配置文件时尽量减少磁盘扫描，在网络方面，由于路由表的入口数量庞大，因此设计时尽量保证对系统整体影响较小的设计。在内存管理、进程管理方面也作出了很多努力。解读这种先进技术也是内核构建的魅力之一。

然而，最近的 Linux 所提供的并不只有基本功能。随着功能的不断发展，现在已经出现了很多特定领域的便捷功能和独特功能。即使是内核黑客也很少有人能够完全掌握。

本书从 Linux 内核的众多先进功能中选取了一些必备并且有趣的内容进行介绍，同时也对内部的运行机制和结构进行了阐述。此外，本书还介绍了熟练使用这些功能所需的工具、设置方法以及调整方法等。

省电就是其中一项内容。除了使用方法以外，本书还介绍了省电的理念、与硬件的关系等。此外，还提到了当前广受关注的虚拟化、资源管理、标准文件系统中所采用的 ext4 等已有功能和新功能。对于已有功能，本书结合最新的源码，介绍它的更改内容和新增功能。其中也包括文档中没有记载，且必须对内核内部有一定理解才能得知的信息，因此，即使是比较了解这个功能的人也可能会有新的发现。另外，本书还介绍了内核的相关工具，其中 `gcore` 在重要的系统中就是非常可靠的工具。

最新的 Linux 内核中安装了强大的追踪、概要分析功能，具备很多方便实用的功能。这些功能不仅能够很方便地达到预期的目的，而且对于分析内核功能也非常有用。甚至对于内核构建的高手也有一定帮助。

全书列举了非常多的实例，让读者更快地学会如何使用。对于想要熟练使用内核的读者来说，本书也是非常好的参考书。

本书还为想要了解 Linux 内核的读者以及读过本书后开始对 Linux 内核开发产生兴趣的读者，介绍了获取内核源码的方法和内核开发方法等内核构建入门所需的信息。我们希望读者能够通过本书更加了解 Linux 的世界。

在电脑刚刚诞生的时候，有一段时期人们认为“如果想要提高编程水平就查看 UNIX 代码”。因为最快的方式就是参考天才所编写的最先进的代码并进行模仿。而在阅读 Linux

内核的代码时，相信大家也会深有同感。

Linux 内核是开源软件，无论是谁都可以参与开发。Linux 内核的代码花费了大量的时间和精力来编写。各领域都由具有专业知识的维护人员进行长期的管理，从而得到不断的改进。基于电子邮件的开发也在不断进行，因此可以看到各种讨论，并了解到当前代码的发展历程。每次看到 Linux 内核的代码，都会让人感叹其中凝聚的智慧和努力，也感受到当时的辛苦。希望读者能够从本书开始接触 Linux 这个不一般的世界，诞生更多的内核高手。

本书主要内容

本书介绍的是 Linux 内核所提供的功能。不仅有比较基础的功能，还有一些功能需要具有一定的知识才能使用。

此外，还介绍了使用功能时需要用到的信息和命令。除了内核以外，本书还将介绍相关的应用程序。基本上是基于 TUI 进行说明的，但也有一部分关于 GUI 的介绍。

涉及的主要版本为 Linux 内核 2.6.18 到写作时最新的 Linux 内核 3.0[注 1]。其中一部分还介绍了 Red Hat Enterprise Linux 4（RHEL4 ：基于 Linux 内核 2.6.9）的功能。示例代码已经在工作中经常使用的 RHEL 和任何用户都可以使用的 Fedora、CentOS 等中进行过严格测试。

本书不涉及 Linux 内核的实际安装和以算法等为主体的内容。

本书使用方法

本书可以按顺序依次阅读，另外由于每一节之间都是独立的，因此也可以从感兴趣的章节开始阅读。第 1 章介绍了内核的基础知识，如果是第一次接触内核，建议先学习第 1 章。本书在介绍已有功能时也加入了一些新的信息。相信即使是经验丰富的人也可以在本书中有新的发现，因此希望各位读者能够将本书从头到尾完整读一遍。本书还收录了一些作者珍藏的信息。详细内容请参见参考文献。

本书约定

等宽字体（sample）

表示文件名、文件的内容、控制台的输出、变量名称、命令、命令选项、数据包名称、模块名称、驱动程序名称、键、内核配置、样本代码、其他代码等。

等宽粗体（sample）

表示应替换为用户输入的命令或文本等。

注 1：写作本书时已经发布了 3.0-rc 版本。

斜体（sample）

表示根据环境决定的值等。

小贴士：表示提示、建议、补充事项等。

注意事项：表示注意、警告等。

每一节标题左侧的温度计图标表示该节的相对难易度。

意见与提问

关于本书的内容，我们尽最大的努力进行了验证和确认，但可能还是会存在错误或不正确的地方，或者是会引起误解或混淆的表述、输入错误等。如果在阅读本书的过程中发现了这些问题，请告知我们，以便进行改善。

株式会社 O'REILLY（奥莱利）JAPAN

邮编 160-0002　东京都新宿区坂町 26 番地 27 Intelligent 大厦 1 层

电话　　03-3356-5227

FAX　　03-3356-5261

电子邮件　　japan@oreilly.co.jp

关于本书的技术性问题和意见请发送到下列邮件地址。

japan@oreilly.co.jp

本书的网站上可以找到示例代码[注2]、勘误表和附加信息。

http://www.oreilly.co.jp/books/9784873115016/

关于 O'REILLY 的其他信息请参考下列网站。

http://www.oreilly.co.jp/（日语）

http://www.oreilly.com/（英语）

注 2：这些示例代码是笔者写作时使用的程序，并不保证在各种环境下都可以运行。另外，有时会不经过提示进行修改。示例代码不一定都能对应，敬请谅解。

目录

第 1 章

内核入门

一提起内核包，总会让人感觉似乎困难至极、如临深渊一般。但其基本的操作与其他开放源代码软件包并没有什么不一样，都是首先获取源代码，进行解读，然后修改或者添加新功能对应的代码，并编译、测试。本章将介绍这些内核包操作中最基础的知识，以及 Linux 内核特有的方法。

HACK #1 如何获取 Linux 内核

本节介绍获取 Linux 内核源代码的各种方法。

“获取内核”这个说法看似简单，其实 Linux 内核有很多种衍生版本。要找出自己想要的源代码到底是哪一个，必须首先理解各种衍生版本的意义。

接下来将简单介绍 Linux 内核的开发模式，并分析各种衍生版本在其中所处的地位，然后介绍获取这些衍生版本的源代码的方法。

内核的种类

想要获取正确的 Linux 内核源代码，首先必须了解 Linux 内核的开发模式。

Linux 内核是由多个开发者以分散型的模式进行开发的。这里出现的“分散型”，是指多个衍生源码树同时存在。下面将简单介绍一些具有代表性的源码树及其地位。

Linus 树

最具有代表性的源码树，应属 Linux 内核的最初创始人——Linus Torvalds 所管理的 Linus 树。新版本 Linux 内核的发布，就意味着 Linus 树的源代码被贴上了新发布版本的标签。到 2011 年为止，Linux 内核的版本号一直是用 2.6.x 这样的三个数字来表示的[注 1]。Linus 树一直被认为是 Linux 内核源代码的“根源”，因此一旦其发布了新版本，其他的开发树就会将自己独特的开发成果移植到这个版本上，在此基础上再次进行开发。Linus 树由于其“根源”的地位而称为主线（mainline）。

注 1：Linux 2.6.39 的下一版本将是 Linux 3.0。

一旦发布新版本 Linus 树，就会立刻打开一个“合并窗口”（merge window），接受下一版本需要作出的改变。合并窗口将开启约两周时间。合并窗口关闭后，就会发布下一版本的候选版，即所谓的“rc 内核”[注2]。从 rc 内核发布后到下一版本发布的期间为测试期，这一期间基本只接受关于 bugfix 的修改。rc 版内核每隔约一周时间会依次推出 rc1、rc2……当 Linus 判断其质量已经达到可以发布的水平时，就会作为新版本发布。按照最近的实际情况来看，基本上在 rc6 ～ rc9 左右就会发布新版本，也就是说 Linux 内核每隔 2 ～ 3 个月就会发布新版本。新版本发布后，又会打开下一版本的合并窗口，然后对 rc 版进行测试。Linux 内核就是按照这样的周期来开发的。

小贴士：Linus 树的内核由于完全没有任何华而不实的东西，因此称为“香草”（vanilla）内核或“库存”（stock）内核。

linux-next 树

这是一个为发布将来的版本而积累新代码并进行测试的源码树，主要由 Stephen Rothwell 等人进行管理和运营。原则上要添加新功能或者进行安装配置时，首先要在 linux-next 树中进行测试，在确认各自之间可以兼容之后再添加到 Linus 树内。

stable 树

这是一个主要只针对过去发布的内核版本进行 bug 修改，使其更加稳定的树，由 Greg Kroah-Hartman、Chris Wright 进行维护管理。这个树的版本号是在 Linus 树的版本号后面加一位数字，以 2.6.x.y 这样的 4 个数字来表示。针对某个 Linus 树版本的稳定（stable）版维护一般持续 6 个月左右，但也有持续更久的。

开发树

Linux 内核可以说是各种功能的集合体。例如内存管理、文件系统、网络、各种设备驱动程序、CPU 架构固有部分等。这些功能部分称为“子系统”，各子系统分别在不同的源码树中进行开发。在开发、修改过程中也有一些不属于特定子系统的内容，这些内容首先会被发送到 Andrew Morton 管理的 mm 树（准确地说是 mmotm：mm on the moment，补丁包的缩写）。这样的源码树统称为“开发树”。

在各开发树中开发出的源代码在经过 linux-next 中的测试后再植入 Linus 树。

开发树的数量多如繁星。如果哪天你因为想要开发某个功能而在手边的源代码上进行了修改，这也可以说是一个“开发树”。

Linus 树、开发树等作为所有树的根源，也称为“upstream”，即“上游”。但这是广义上的叫法，有时也仅指最上游的 Linus 树。

注 2：rc 是 release candidate（发布候选）的缩写。

发布版内核

最后要介绍的是发布版内核（distribution kernel）。应该有很多人使用的都是作为 Linux 发布版的一部分发布的内核。这些来源于发布版的内核几乎都是在 Linus 树或 stable 树内核的基础上进行发布版特有的扩展和 bug 修改而得到的。像这样添加了发布版特有的修改，并作为发布版的一部分发布的内核，就称为“发布版内核”。

如何获取上游内核

在了解 Linux 内核的各种衍生版本后，我们首先尝试一下获取上游内核（upstream kernel）。Linus 树、linux-next 树，以及绝大部分的开发树都可以从 http://www.kernel.org/ 获取（见图 1-1）。

Linux 内核的开发都是在最新版上游内核的基础上进行的。其中最重要的就是作为所有树的根源的 Linus 树。下面介绍获取 Linus 树的两种方法。

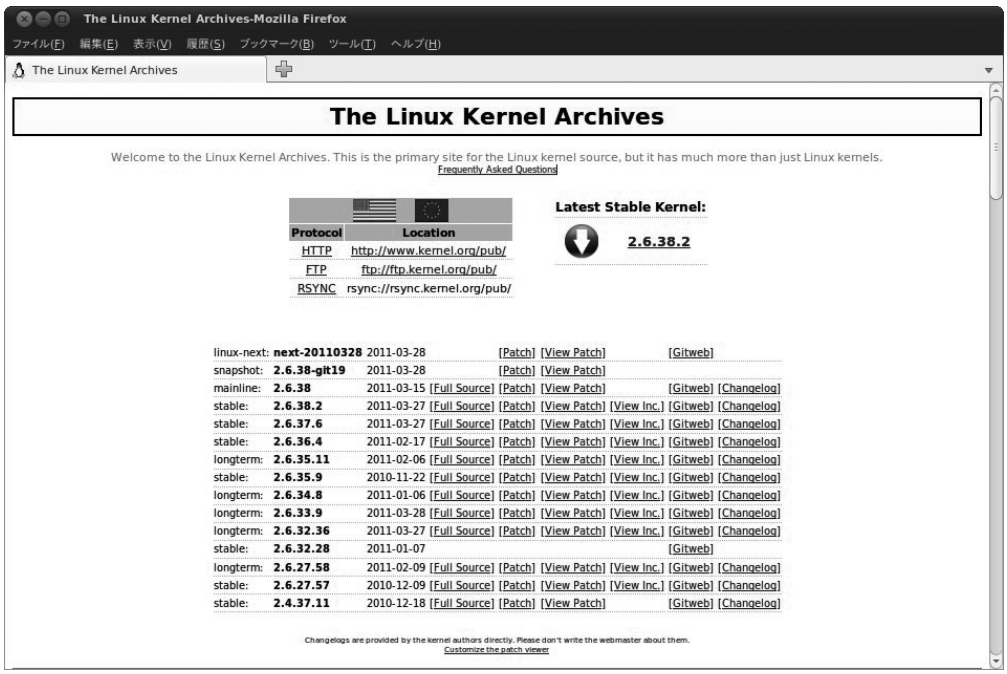

图 1-1　http://www.kernel.org

下载 tar 文件

获取 Linus 树最简单的方法就是从 kernel.org 下载 tar 文件。2.6 内核所有发布版本的 tar 文件都能够从 http://www.kernel.org/pub/linux/kernel/v2.6/ 获取。

这里有很多种类的文件。例如，表 1-1 所示的是与 2.6.38 对应的文件，可以从中随意选择一个下载。无论下载的是哪个，解压缩后或打补丁后的 tar 文件都是一样的。

表 1-1　Linux-2.6.38 的各种源文件

文 件 名	内　　容
linux-2.6.38.tar.bz2	完整的源码树。使用 tar+bzip2 压缩
linux-2.6.38.tar.gz	完整的源码树。使用 tar+gzip 压缩
patch-2.6.38.bz2	2.6.37 升级到 2.6.38 的补丁。使用 bzip2 压缩
patch-2.6.38.gz	2.6.37 升级到 2.6.38 的补丁。使用 gzip 压缩

除这些以外，还有一些文件名后缀为“.sign”的文件。这些文件都是用来确保各个文件兼容性的 GnuPG 签名。可以在验证下载是否正常时使用这些文件。

小贴士：使用 GnuPG 来检测兼容性时可以执行下列命令。

```
$ gpg --keyserver wwwkeys.pgp.net --recv-keys 0x517DoFoE
$ gpg -verify < 签名文件 > < 下载的文件 >
```

详细内容请参考 http://www.kernel.org/signature.html。

rc 版或者更新更为频繁的快照 tar 文件存放在子目录下。主要的子目录如表 1-2 所示。

表 1-2　http://www.kernel.org/pub/linux/kernel/v2.6/ 的子目录

目 录 名	内　　容
next	rc 版升级到 linux-next 的补丁
testing	rc 版源码树的 tar 文件与补丁。目录下的文件仅针对下一发布版本。针对以前版本的文件存放在下一层的子目录下
snapshots	每天更新 1 ～ 2 次的快照文件补丁

使用 Git

Linus 树和开发树通过修复各种补丁而不断更新。在最新的树中进行开发是最基本的原则，因此为了保持最新，必须每天多次下载 tar 文件修复补丁。这项工作是非常花费精力的，但是也不需要担心，因为可以用 Git 来解决。

Git 是 Linux 内核所采用的 SCM（Source Code Management system），具备分散开发所需的多个功能。Git 命令更为详细的使用方法将在 Hack #4 中介绍，这里就先了解一下怎样使用 Git 命令来获取最新的 Linus 树。要在适当的目录下执行下列命令，但是在此之前必须注意的是，因为这条命令会将包括修改记录在内的所有仓库数据复制到本地磁盘中，所以必须要有 1GB 以上的磁盘容量。在操作时请注意磁盘和网络的容量。

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6
```

命令执行完成后，应该就会生成一个标题为 linux-2.6 的目录。这就是包括修改记录在内的 Linus 树的最新、最完整的副本。

为了让手头的源码树时刻保持最新，需要在源码树的根目录（linux-2.6）下执行下列

命令。

```
$ git pull
```

如果没有对手头的源码树代码作出任何修改，该命令会使得手头的源码树与 Linus 树的最新状态保持一致。当使用 git clone 进行复制时，git 命令会记住复制源目录的 URL，因此执行 git pull 时不需要指定 URL。

使用 git 命令还可以获取除 Linus 树以外的开发树的最新版本。在 http://git.kernel.org/ 上能看到放置在 kernel.org 下的其他开发树一览表。如果希望开发或者追踪各领域最新开发情况，也可以从这里找到开发树的 URL。

如何获取发布版内核

在多数情况下，发布版内核的源代码都是按照各发布版所采用的方法进行打包的。因此要获取发布版内核的源代码，只需要下载源代码包，进行安装或解压缩就可以了。

下面选取具有代表性的发布版 Fedora 和 Ubuntu 为例，讲解如何获取这两种发布版内核源代码。这里选取的发布版的版本分别为 Fedora 14 和 Ubuntu 10.10。

Fedora

在 Fedora 中，内核源码是作为源码 RPM（SRPM）提供的。使用 `yum-units` 包里所带的 `yumdownloader` 下载 SRPM。此后要使用的 `yum-builddep` 也是 `yum-units` 包中所带的，所以如果事先没有安装，首先请安装这个工具包。

可以执行下列命令来下载内核的 SRPM。

```
$ yumdownloader --source kernel
```

在笔者的环境下，下载的是 `kernel-2.6.35.11-83.fc14.src.rpm`。

如前文所述，发布版内核都带有自身特有的补丁。SRPM 是将 vanilla 内核的源代码和补丁分开放置的，补丁在创建过程中被分配给 vanilla 内核的源代码。所以要获取发布版内核的源代码，就要完全执行 RPM 的创建过程，但并不完全执行。虽然每个 SRPM 在创建 RPM 时都需要用到不同的源码包，但只要执行下列命令，就能够安装创建 Linux 内核所需的所有源码包。这条命令请在 root 权限下执行。

```
# yum-builddep kernel-2.6.35.11-83.fc14.src.rpm
```

安装 SRPM 需要执行下列命令。安装 SRPM，就是指将所包含的文件解压缩。SRPM 包含的文件将解压缩到在主目录下生成的 `rpmbuild` 的几个子目录下。

小贴士：生成 `rpmbuild` 目录的位置，是通过 `%_topdir` 这个 `rpm` 的宏变量来设置的。在以前的发布版中是在 `/usr/src/redhat` 下的生成，近来的版本是在 `/usr/lib/rpm/macros` 下创建：

```
%_topdir          %{getenv:HOME}/rpmbuild
```

直接在用户目录下生成。

```
$ rpm -i kernel-2.6.35.11-83.fc14.src.rpm
```

RPM 包的创建是通过 rpmbuild 命令进行的。本次操作的目的是获取内核的源代码，所以为了让命令结束前能给源代码修复补丁，应在 rpmbuild 命令中加上 -bp 选项。命令中的参数会赋予一个用来创建源码包的设置文件，即 SPEC 文件。

```
$ cd ~/rpmbuild/SPEC
$ rpmbuild -bp kernel.spec
```

这时就会生成一个标题为 ~/rpmbuild/BUILD/kernel-2.6.35.fc14/linux-2.6.35.x86_64 的目录，在这个目录下会生成发布版内核的源代码。

使用源代码来创建内核二进制文件的方法，请参考 Hack #2。

Ubuntu

在 Ubuntu 或基于 Ubuntu 的 Debian 下，内核源代码是作为 deb 包提供的。首先，与其他的源码包一样用 apt-get 来执行安装。标题为 Linux-source 的源码包就是最新的内核源码包的元包。

```
# apt-get install linux-source
```

在笔者的环境下，到这一步就完成了 linux-source-2.6.35 的安装。

在安装内核源代码的 deb 包后，会在 /usr/src 下生成 tar 文件，只要将这个文件复制到适当的目录下并解压缩，就能够获取内核源代码。

```
$ cp /usr/src/linux-source-2.6.35.tar.bz2 ~
$ cd
$ tar xjf linux-source-2.6.35.tar.bz2
```

关于创建内核二进制码的方法，同样请参考 Hack #2。

小结

本节介绍了在上游内核与发布版内核这两种情形下获取内核源代码的方法。首先要获取源代码，然后才能够读取源代码、修改 bug 以及开发新功能。Linux 内核中有很多信息是必须读取源代码后才能理解的。通过读取源代码，能够从真正意义上理解一直以来“以为理解”的内容。因此一定要努力学习源代码。

参考文献

• 内核文档 Documentation/development-process/*

Linux 内核版本 3.0

2011 年 5 月，Linus Torvalds 宣布，Linux 内核版本由 2.6 升级到 3.0。据 Linus Torvalds 称，2.6 系列由于经过 39 次发布后，更新号（第三个数字）过大，因此本次版本升级最大的目的就是进行一次改头换面。由 2.6.39 到 3.0 的版本升级与 2.6 系列内的版本升级并没有什么不同，也是由 2.6 系列延续下来的。目前预计在版本 3 系列中，将第二个数字作为 Linus 树的更新号，第三个数字作为 stable 树的更新号使用。也就是说，Linus 树 3.0 的下一个发布版本将是 3.1，基于版本 3.0 的 stable 树内核将是 3.0.1、3.0.2。Linus 树内核的 tar 文件在版本 3 系列中放在以下位置：

`http://www.kernel.org/pub/linux/kernel/v3.0/`

Git 仓库也同样可以通过前面提到的 URL（`git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6`）来获取。

——Munehiro IKEDA

HACK #2　如何编译 Linux 内核

本节介绍编译 Linux 内核的方法。

当发现 bug 而修改源代码或者添加新功能时，就需要对内核进行重新编译，生成二进制映像文件。另外，如果想要使用发布版内核中无效的功能或者驱动程序时，或者相反地，想要删除不需要的功能从而使内核更精简、更快时，或者想使用最新版的上游内核时，也需要对内核进行编译。

下面主要介绍对上游内核进行设置、编译以及安装的方法。当使用发布版内核的源码包管理系统来管理内核映像文件时，需要将内核映像文件打包。接下来以两个具有代表性的发布版 Fedora 和 Ubuntu 为例来讲解具体的方法。最后将简单地介绍在源码树外对驱动程序等进行编译的方法，以及在不同平台的编译环境编译内核的方法，即所谓的交叉编译。

内核编译的过程

对内核进行编译的步骤如下：

1. 获取源代码，如有需要则进行修改。
2. 设置。
3. 编译。
4. 根据发布版生成相应的源码包。
5. 安装内核映像和模块。

使用上游内核安装内核映像时，若不使用发布版的源码包管理系统，则不需要进行步骤 4。想要使用源码包管理系统来安装时，可以使用各发布版的源码包创建系统。在这种情况下步骤 3 和步骤 4 的操作是合并进行的。

下面首先讲解不使用源码包管理系统来生成、安装内核映像文件的情形。然后介绍将内核映像文件打包和安装的方法。

需要的源码包

对内核进行设置和编译时可以使用各种工具。如果没有明确指示，就不会安装表 1-3 和表 1-4 所示的源码包，但是它们是必不可少的。这些需要事先安装好。

表 1-3　必要的源码包一览表（Fedora 14）

源 码 包 名	备　　注
ncurses-devel	基于控制台（文字界面）设置时需要
qt-devel	基于窗口（图形界面）设置时需要
qt3-devel	基于窗口（图形界面）设置时需要
gcc-c++	基于窗口（图形界面）设置时需要
rpm-build	生成 rpm 包时需要

表 1-4　必要的源码包一览表（Ubuntu 10.10）

源 码 包 名	备　　注
libncurses5-dev	基于控制台（文字界面）设置时需要
qt3-dev-tools	基于窗口（图形界面）设置时需要
g++	基于窗口（图形界面）设置时需要
kernel-package	生成 deb 包时需要
fakeroot	生成 deb 包时需要
dpkg-dev	生成 deb 包时需要

编译、安装上游内核

获取源代码

关于获取内核源代码的方法，请参考 Hack #1。

这里以源代码在 ~/linux-2.6 下的情况为例。

进行设置

Linux 内核自身的源代码树中就具备进行编译设置的结构，不仅可以设置编译或不编译某个功能，在进行编译时，还能非常细致地设置是将功能静态添加到 Linux 内核的二进制码中，还是作为模块进行编译。虽然能够进行细致的设置，但同时也造成设置项目数量过多。因此源代码树中还带有帮助进行设置的工具。这个工具包称为 kconfig，应先启动这个工具再进行设置。

小贴士：2.6.38 中的设置项目数量超过 12 000 个。

设置工具虽然准备了基于控制台（文字界面）和基于窗口（图形界面）的两种类型，但其实用户要执行的操作不管用哪一个界面都没有太大的区别。这里主要以基于控制台的工具为例展开介绍，基于窗口的工具仅在后面简单介绍。

要启动基于控制台的设置工具，需在源码树的根下执行下列命令。

```
$ make menuconfig
```

之后控制台就会显示如图 1-2 所示的项目。设置是按层次进行的，现在看到的是最上面一层。

在这里，先不执行任何操作，按一下【TAB】键，选择 Exit 后，再按下【Enter】键。就会出现询问“是否保存新设置”的选项，然后选择 Yes 按钮关闭设置工具。

图 1-2　make menuconfig 生成的设置画面

编译设置保存在源码树的根下标题为 .config 的文件里。因为这次是在 .config 文件不存在的状态下启动的设置工具，所以会生成一个默认设置内容的 .config 文件。

.config 的内容如下所示（细节部分会因为执行环境的不同而有所差异）。

```
#
# Automatically generated make config: don't edit
# Linux/x86_64 2.6.38 Kernel Configuration
```

```
# Sun Mar 27 03:17:57 2011
#
CONFIG_64BIT=y
#CONFIG_X86_32 is not set
CONFIG_X86_64=y
CONFIG_X86=y
...
```

以 # 开头的行是注释行。

CONFIG_* 是设置项目。这些设置项目与 Linux 内核的各功能相对应，编译时受这个值的控制。设置项目取表 1-5 所示三个值中的一个。

表 1-5　设置项目（CONFIG_*）所取的值

值	说　明
=y	该项目所对应的功能将静态添加到内核中
=m	该项目所对应的功能将编译为模块。当内核在执行时，模块会在需要时加载并添加到内核中。有一些功能无法作为模块进行编译。在这种情况下对应的设置项目不取该值
# CONFIG_* is not set	该项目所对应的功能不编译。这些项目的行全部被注释掉

注意事项： .config 文件不能手动编辑。有时某个功能会依赖于其他功能。在这种情况下，如果设置不能正确反映依赖关系，就会出现编译错误或者最终变成无法执行的内核。kconfig 可以掌控依赖关系，并保证设置的兼容性。

小贴士： 在没有 .config 文件的情况下启动并执行 make　menuconfig 命令后生成的 .config 文件，是根据 kconfig 的设置文件——Kconfig* 中的默认值生成的。

另外，在源码树中分别为每个架构准备了默认的 .config 文件。不按照 Kconfig 的默认设置，而是想根据各架构的默认设置生成 .config 文件时，需执行下列命令。

```
$ make defconfig
```

各架构的默认 .config 文件位于以下路径。

```
arch/<arch>/configs/*_defconfig
```

现在就可以尝试进行设置了。再次执行 make　menuconfig 命令打开设置菜单。设置菜单中经常使用到的按键如表 1-6 所示，以供参考。

表 1-6　设置菜单的按键一览

按　键	操　作
↑	将选择项目的光标向上移动
↓	将选择项目的光标向下移动
<TAB>、←、→	切换操作菜单（Select/Exit/Help）

（续）

按　键	操　作
<Enter>	按照所选择的操作菜单进行操作
Y	将项目设为 <*>（有效：静态添加）
N	将项目设为 <>（无效）
M	将项目设为 <M>（有效：作为模块进行编译）
<SPACE>	将项目在 <M>/<*>/<> 间切换
<ESC><ESC>	回到上一层（与操作菜单的 <Exit> 相同）
?	显示关于所选项目的帮助（与操作菜单的 <Help> 相同）
/	搜索设置项目。根据设置项目的符号名来搜索在菜单上的位置时十分方便

这里以软盘驱动器为例，尝试启用它。这个设置项目在 2.6.38 中定义为 CONFIG_BLK_DEV_FD，位于菜单层的如下位置。

```
Device Drivers --->
    Block devices --->
          Normal floppy disk support
```

图 1-3 所示就是在菜单上选择这个项目的界面。

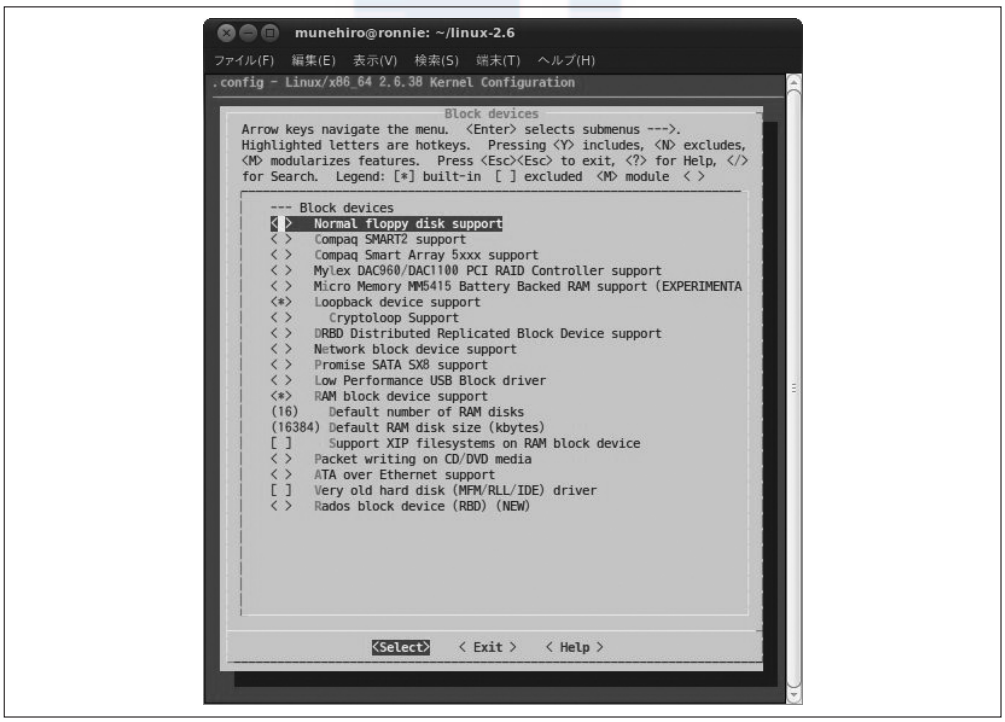

图 1-3　Normal floppy disk 支持的设置

这时按【y】键，左侧的选择显示就会变成 <*>。这就表示该项目已编译并静态添加到内核中。按【m】键，显示则变成 <M>。这时，该项目将作为模块编译。当使用该功能时，模块会根据需要动态添加到内核中。如果按【n】键，则会变成 <>，该功能不编译。

内核的编译设置就是这样指定一个个的项目来进行的。

此外，还有一些项目需要设置数值和字符串。例如，在当前打开的界面中，“Default number of RAM disks”等就是这种项目。选择这样的项目后按回车键，就会出现对话框，可以在其中输入数值或字符串。

大致完成自己想要的设置后，可以在菜单最上层的画面上选择操作菜单的 <Exit>，或者连按两次【Esc】键完成设置，在出现的“是否保存新设置？”对话框中选择 Yes 按钮，将设置保存到 .config 文件中。

到这里，内核的编译设置就完成了。

小贴士：经常有人因为手头有在旧版本中生成的 .config 文件，而想要在此基础上进行一些修改，以编译新的内核。在这种情况下，可以将原来的 .config 文件复制到源码树的根下，然后执行下列命令。

```
$ make oldconfig
```

执行该命令后，就会出现一个个的对话框，询问新增加的设定项目要如何设置。如果版本之间的差别较大，就需要回答较多的项目设置问题，所以可以先针对所有的询问都按回车键，以便使用默认设置，然后再用 make menuconfig 等进行设置。

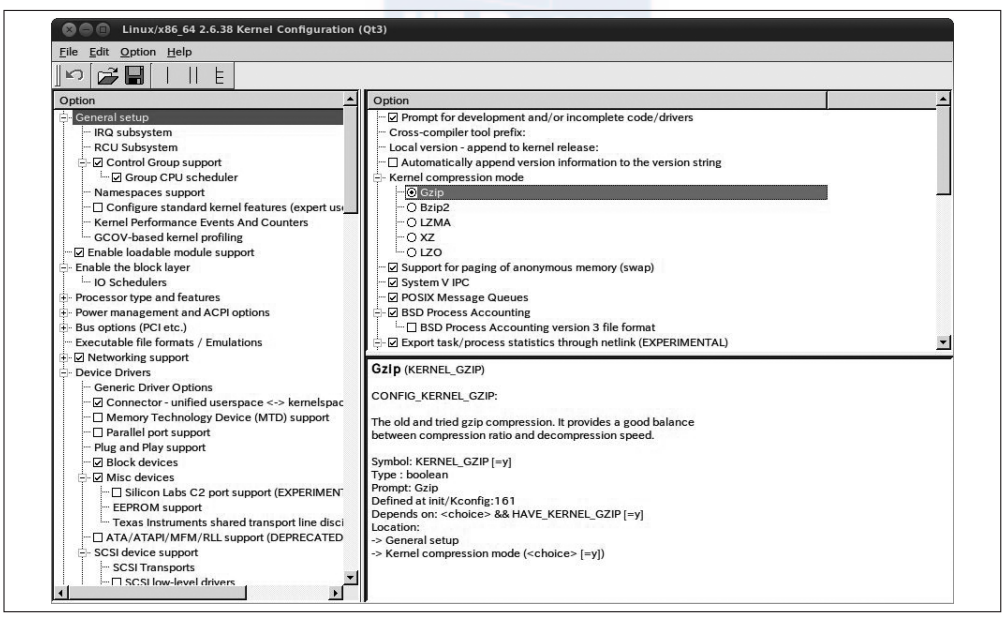

图 1-4　make xconfig 生成的设置界面

基于窗口（图形界面）的设置工具通过下列命令来启动。

```
$ make xconfig
```

启动后就会打开如图 1-4 所示的窗口，就在这里进行设置。

如果掌握了基于控制台的设置工具，那么基于窗口的工具也就很容易掌握了。可以根据个人喜好来选择使用哪一个。

进行编译

要对已完成设置的上游内核进行编译，需在源码树的根目录下执行下列命令。

```
$ make
```

编译需要花费的时间与机器的性能和设置有较大的关系，快的时候只需要几分钟，有时也可能需要花费几个小时。请耐心等待。

如何节约编译时间

下面列出了几个缩短编译时间的小提示。

在内核的设置上下工夫

编译的代码越少，编译就能越快完成。一般为了使发布版内核在多样的环境中能够顺利运行，都会带有多个驱动程序。将其中在自己的环境中不需要的驱动程序设置为无效，仅将必要的作为编译对象，就能大幅缩短编译时间。关于设置的技巧请参考 Hack #6。

使用 Make 的 -j 选项

-j 选项是用来指定 make 的并发性的选项。并发性是用数值来指定的，例如“-j 4”。当机器为多处理器时，可以按照处理器的数量来指定数值，这样就有可能实现快速编译。

购买高性能的机器

速度是永远的追求。

将内核安装到系统中

编译完成后，就可以将生成的内核安装到系统中。安装时必须有 root 权限。

安装分为两个阶段进行。第一阶段是模块的安装。在编译完成的源码树的根目录下执行下列命令。

```
# make modules_install
```

这时，编译后的模块就安装到 /lib/modules 下。

第二阶段是安装内核二进制映像文件，生成并安装 boot 初始化文件系统映像文件。同样

也是在源码树的根目录下执行下列命令。

```
# make install
```

这时，内核映像文件就安装到 /boot 下。如果使用的是 Fedora 系列的发布版，就会同时生成 boot 初始化文件系统映像，并同样安装到 /boot 下。而 Ubuntu 等 Debian 系列的发布版则需要执行下列命令，另行生成和安装 boot 初始化文件系统映像。在 < 内核版本 > 的部分请输入表示当前生成的内核版本的文字。

```
# update-initramfs -c -k < 内核版本 >
```

表 1-7 所示为通过安装生成的文件以及目录的一览表。在 < 内核版本 > 的部分中同样输入表示当前生成的内核版本的文字。

表 1-7　通过安装内核而生成的文件和目录

文件名或目录名	内　　容
/lib/modules/< 内核版本 >	安装模块的目录
/boot/vmlinuz-< 内核版本 >	内核映像文件
/boot/initramfs-< 内核版本 > 或 /boot/initrd.img-< 内核版本 >	boot 初始化文件系统映像
/boot/Systemmap-< 内核版本 >	地址信息文件

在一些系统设置下可能需要手动进行 GRUB 设置才能从当前安装的内核启动。在这种情况下，请适当地编辑 /boot/grub/menu.lst 或者执行 update-grub 命令。

到这里内核的安装就完成了。这时请重启机器，确认新的内核是否能够正常运行。

注意事项：在确认新内核能够正常运行之前，绝对不要删除现在所使用的内核以及相应的 GRUB 的记录。新内核经常会出现无法启动的情况。这时如果没有启动的内核，系统就无法进行操作。

小贴士：内核二进制映像的文件名和模块目录名是根据内核的版本来命名的。因此，当对已安装版本的内核进行再次编译并安装时，原来安装的内核以及模块会被覆盖。

为了防止这种情况的发生，可以在设置项目 CONFIG_LOCALVERSION 中设置文字。在这个设置项目中指定的文字会作为内核版本的一部分，因此可以防止被覆盖。

其他的 make 对象或变量

在进行内核的设置或编译时，有一些可用的 make 目标，如表 1-8 所示。

表 1-8　其他的 make 对象

对　　象	说　　明
clean	将源码树恢复到编译前的状态。obj 文件等被删除，.config 或编译过程中自动生成的部分文件不会被删除

（续）

对　象	说　明
mrproper	将源码树完全恢复到发布时的状态。发布时源码树中不存在的文件全部被删除，包括 .config 文件
help	显示可以使用的 make 对象
tags	生成标签文件。有了标签文件，就能使用 Emacs 等编辑器的 tag jump 功能跳到函数定义处，可以高效进行源码浏览
cscope	生成用于 cscope 的索引文件。cscope 是基于控制台（文字界面）的源码浏览器
allyesconfig	生成将所有设置项目设置为有效并静态添加到内核中的 .config 文件
allnoconfig	生成将允许范围内的设置项目设置为无效的 .config 文件
allmodconfig	生成将所有能设置为模块的项目设置为有效并设置为模块的 .config 文件
<dir>/<file>.o	仅进行生成指定的目标文件所必需的编译。当仅指定 <dir> 时，由 .config 文件生成该目录内所有目标文件
<dir>/<file>.ko	仅生成指定的模块

表 1-9 展示了为控制 make 的输出而设置的 make 变量。这些变量需在对象前指定，如：

```
$ make V=1 clean
```

表 1-9　控制 make 输出的变量

变　量	说　明
V=0|1|2	设置编译时控制台显示的详细程度。默认为仅显示概要，设置为 0。当设置为 1 时，显示更加详细的信息。当设置为 2 时，除了概要以外，还会显示要进行编译的原因（原因大多数都是“由于没有对象”）
O=<dir>	将编译最终生成的文件全部输出到 <dir>。在源码树禁止写入等情况下非常实用

卸载内核

按照前面介绍的安装方法安装的内核不在发布版源码包管理系统的管理范围内。因此不能用 rpm 或 dpkg 这样的命令来卸载。

但是也不需要担心，内核的文件配置如表 1-5 所示，相对来说比较简单。想要卸载时，只需要删除这些文件就可以了。

注意事项：卸载内核时请慎重考虑。另外，请一定要做好更改 GRUB 设置等工作。

生成内核包

Fedora

Fedora 的源码包管理系统是 RPM。要将内核纳入 RPM 的管理范围内，就需要生成 RPM 源码包。

其实 Linux 内核在创建时就具备生成 RPM 源码包的功能。编译时需要将 `rpm-pkg` 作为对象执行 `make` 命令。

```
$ make rpm-pkg
```

通过这条命令，编译内核后就会创建源码包（SRPM）和二进制码包（RPM），二进制码包存放在 `~/rpmbuild/rpms` 下，源码包存放在 `~/rpmbuild/SRPMS` 下。

如果拥有将 SRPM 解压缩后的发布版内核的源码，则使用 `rpmbuild` 创建源码包。如果内核的 SRPM 是解压缩到 `~/rpmbuild` 下的，则执行下列命令创建源码包。

```
$ rpmbuild -ba ~/rpmbuild/SPECS/kernel.spec
```

所创建的源码包存放的目录与上面相同。这些源码包和普通源码包一样，可以使用 `rpm` 命令来安装、卸载。

小贴士：在上游内核中创建源码包时也是用 `make` 来调出 `rpmbuild` 的。

Ubuntu

Ubuntu 的源码包管理系统是 dpkg。源码包为 deb 格式。

上游内核的创建与 RPM 同样，也能生成 deb 源码包。这一 `make` 操作的对象为 `deb-pkg`。通过执行下列 `make` 命令，就能够创建 deb 源码包。

```
$ make deb-pkg
```

所创建的源码包存放在源码树的根目录下。会生成数个源码包，其中包含内核映像和模块的是 linux-image-< 内核版本 >.deb 文件。这些源码包的操作和普通的 deb 源码包文件一样，可以用 dpkg 来进行。

此外，Ubuntu 还在 kernel-package 包里收录了用来协助内核包创建的工具——`make-kpkg` 命令。这个工具可以通过命令选项对创建操作进行设置，根据需要也可以使用这个工具。这里就不介绍详细的使用方法了。

在源码树外编译模块

有时可能想要对还未导入上游内核的驱动程序等与内核源码树分开提供的源代码进行编

译，并将其作为模块安装。

在这种情况下，只要驱动程序源代码的 Makefile 编写正确，就可以按照下列方法进行编译、安装。

```
$ make -C /lib/modules/$(uname -r)/build M=$PWD
# make -C /lib/modules/$(uname -r)/build M=$PWD modules_install
```

当为 make 指定 -C 选项时，make 首先会读取指定目录下的 Makefile。这里指定为 -C 选项的变量，是指向当前正在运行的内核源目录的符号链接。也就是说，这个驱动程序与当前运行的内核是在完全相同的环境下创建，具体来说，就是使用头文件或 .config 文件来创建的。

指定 M=$PWD 是为了告知 Linux 内核的创建操作正在源码树外执行创建操作。

交叉编译内核

交叉编译是指针对与正在执行编译的平台不同的其他平台生成二进制数据。例如，在 x86_64 环境下生成针对 ARM 的二进制数据的情形。这种编译器又称为“交叉编译器”。

只要拥有交叉编译器，对 Linux 内核进行交叉编译就变得非常简单。这时还需要为 make 赋予两个变量，如表 1-10 所示。

表 1-10　交叉编译所需的变量

变　量	说　明
ARCH	对象架构
CROSS_COMPILE	指定交叉编译器的前缀

举一个使用交叉编译器 armv5tel-linux-gcc 来交叉编译 ARM 内核的例子。在这种情况下，make 命令变成如下所示的内容。

ARM 内核的二进制映像较多使用的是 uImage 格式。第一行创建这个格式的二进制映像，第二行创建模块。

```
$ make ARCH=arm CROSS_COMPILE=armv5tel-linux- uImage
$ make ARCH=arm CROSS_COMPILE=armv5tel-linux- modules
```

创建的内核二进制映像作为源码树内的 arch/arm/boot/uImage 文件。

创建的内核和模块必须转移到对象机器上。如果在对象机器上可以使用源码包管理系统，则最简单的方法就是生成源码包并在对象机器上安装。然而，如果不能使用源码包管理系统，虽然内核映像转移起来很简单，但是模块就有一些问题。模块分散在源码树的各个目录下，想要手动查找这些模块并在 /lib/modules 下构建目录树，是不太现实的。

其实，通过 modules_install 安装模块的位置可以用变量 INSTALL_MOD_PATH 来指定。可以利用这一点，例如，当安装在主目录下时，可以用 tar 对每个目录进行整合，

再转移到对象机器上。这一操作可以用下列命令来实现。

```
$ make ARCH=arm CROSS_COMPILE=armv5tel-linux- INSTALL_MOD_PATH=~/armroot-2.6.38 modules_install
```

这样就会在主目录下生成一个标题为 ~/armroot-2.6.38/lib/modules 的目录，模块就安装在这个目录下。

模块的目录下有标题为 build 和 source 的符号链接，这些都是指向编译过内核的源码树。如果在对象机器上完全不进行编译，就不需要进行修改，如有必要可以在对象机器上适当修改。

小结

本节主要以上游内核为中心，讲解了对内核进行编译设置、编译、安装的方法。一方面，内核是系统的根源，如果设置或者安装错误，系统就有可能陷入无法启动的危险中。另一方面，由于基本不依赖于其他的软件，并且可以安装多个内核，因此笔者认为可以比较放心地尝试修改或改造。

为了保证系统的流畅性，也为了尽快使用最新的内核，不仅内核开发人员，而且 Linux 系统的普通用户也非常需要掌握内核的编译、安装方法。你也可以挑战一下。

参考文献

• Documentation/kbuild/*（内核源文档）

——Munehiro IKEDA

HACK #3　如何编写内核模块

本节将介绍向 Linux 内核中动态添加功能的结构——内核模块的编写方法。

内核模块

Linux 内核是单内核（monolithic kernel），也就是所有的内核功能都集成在一个内核空间内。但是内核具有模块功能，可以将磁盘驱动程序、文件系统等独立的内核功能制作成模块，并动态添加到内核空间或者删除。

内核模块是可以动态添加到 Linux 内核空间的二进制文件，文件扩展名为 ko。

内核模块的编写方法大致有两种。一种是将内核源码树带有的功能编写为模块的方法（参考 Hack #2），另一种是将内核源码树中所没有的特有功能编写为模块的方法。

通过内核配置编写模块

把内核源代码文件中 CONFIG_*=m 的项目所对应的驱动程序编写为模块。编写生成的模

块一般安装在 /lib/modules/ 内核版本 /kernel 下。

以 RHEL6 为例

```
# ls /lib/modules/2.6.32-71.29.1.el6.x86_64/kernel/
arch  crypto  drivers  fs  kernel  lib  mm  net  sound
```

编写特有的内核模块

下面将介绍如何编写内核源码树中所没有的特有内核模块。

以 mymod 模块为例说明，请将下面的代码以 mymod.c 为文件名保存。

```
#include <linux/module.h>
#include <linux/timer.h>
#include <linux/errno.h>

static int sec = 5;
module_param(sec, int, S_IRUGO|S_IWUSR);
MODULE_PARM_DESC(sec, "Set the interval.");

static void mymod_timer(unsigned long data);

static DEFINE_TIMER(timer, mymod_timer, 0, 0);

static void mymod_timer(unsigned long data)
{
        printk(KERN_INFO "mymod: timer\n");
        mod_timer(&timer, jiffies + sec * HZ);
}

static int mymod_init(void)
{
        printk(KERN_INFO "mymod: init\n");

        if (sec <= 0) {
                printk(KERN_INFO "Invalid interval sec=%d\n", sec);
                return -EINVAL;
        }

        mod_timer(&timer, jiffies + sec * HZ);

        return 0;
}

static void mymod_exit(void)
{
        del_timer(&timer);
        printk(KERN_INFO "mymod: exit\n");
}

module_init(mymod_init);
module_exit(mymod_exit);

MODULE_AUTHOR("Hiroshi Shimamoto");
```

```
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("My module");
```

在模块的源代码中包含（include）头文件 linux/module.h。

名为 module_int() 和 module_exit() 的宏，可以调用回调（callback）函数来进行初始化和终止模块的处理。在模块的源文件中进行如下描述，就可以在添加模块时调用初始化函数，在删除模块时调用终止函数。

```
module_init( 初始化函数名 );
module_exit( 终止函数名 );
```

在这个例子模块的情形下调用的分别是 mymod_init() 和 mymod_exit()。

初始化函数为了表示初始化已正常完成，需要返回 0。按照 Linux 内核中的写法，发生错误（error）时将返回一个值为负数的错误代码。在这个例子中，如果设定值出错，则处理为 -EINVAL（非法值）。

下面先用 3 个宏对模块进行定义，但在模块编写中并不是必需的。

MODULE_AUTHOR()	表示模块的作者
MODULE_LICENSE()	表示模块的许可证
MODULE_DESCRIPTION()	模块的说明

这个例子模块还用到了模块参数。模块参数可以使用 module_param() 宏来生成。

```
module_param( 参数名，参数类型，权限（permission）);
```

在例子模块中，sec 定义为 int 类型的模块参数。

另外，还可以使用 MODULE_PARM_DESC() 宏来对模块参数进行说明。

先简单介绍一下这个例子的运行过程。当添加模块时，会调用指定为初始化函数的 mymod_init()。在 mymod_init() 中首先通过 printk() 输出：

```
mymod: init
```

然后确认模块参数 sec 是否正常。在模块参数 sec 的值为 0 以下的异常情形时，会返回 EINVAL 错误代码并终止程序。在判断模块参数 sec 正常后，将内核计时器设置为 sec 秒后启动超时（timeout）函数 mymod_timer()。在每隔 sec 秒启动的 mymod_timer() 中，首先使用 printk() 输出：

```
mymod: timer
```

再次设置 sec 秒的内核计时器，然后终止。当删除模块时，会调用 mymod_exit() 函数，删除内核计时器，通过 printk() 输出：

```
mymod: exit
```

于是模块终止。

接下来需要准备编写模块所需的 Makefile。由于是使用内核的创建框架来生成，因此 Makefile 的内容非常简单。

```
obj-m :=mymod.o
```

最后执行下列 make 命令，通过当前目录（current directory）的源代码和 Makefile 生成模块 mymod.ko。

```
# make -C /lib/modules/'uname -r'/build M='pwd'
```

通过使用 modinfo 命令，可以看到所生成模块 mymod.ko 的信息。从这里可以看到使用 MODULE_* 宏所指定的内容。

```
# modinfo mymod.ko
filename:       mymod.ko
description:    My module
license:        GPL
author:         Hiroshi Shimamoto
srcversion:     61A3BB7CFC0C89B8344F5A5
depends:
vermagic:       2.6.32-71.29.1.el6.x86_64 SMP mod_unload modversions
parm:           sec:Set the interval. (int)
```

添加内核模块

添加内核模块需要用到 insmod 命令或 modprobe 命令。

通过执行 insmod 命令把生成的 mymod.ko 模块添加进来。

```
# insmod mymod.ko
```

使用 dmesg 命令，可以看到例子模块 mymod.ko 的输出内容。

```
# dmesg | tail
      :
mymod: init
```

作为模块初始化函数 mymod_init() 所调用的 printk() 的输出内容会在最后一行显示。

使用 lsmod 可以显示目前添加到内核中的模块列表。

```
# lsmod
Module                  Size   Used by
mymod                   1482   0
      :
```

可以看到，mymod 行存在，模块已添加。

要将已添加的模块从内核空间删除时，可以使用 rmmod 命令。

```
# rmmod mymod
```

执行 rmmod 命令后，模块将从内核空间内删除，使用 lsmod 命令就不会再输出 mymod 行。

此外，使用 dmesg 命令还可以看到终止模块的处理中 printk() 输出的信息 mymod: exit。

```
# dmesg | tail
```

```
       :
mymod: exit
```

下面针对模块参数作一些介绍。在添加模块后，就会在 /sys/module 下生成对应的目录和文件。

```
# ls /sys/module/mymod/
holders initstate notes parameters refcnt sections srcversion
```

可以确认在 parameters 下生成的模块 mymod 中所定义的参数 sec。

```
# ls -l /sys/module/mymod/parameters/sec
-rw-r--r--. 1 root root 4096 May 15 06:34 /sys/module/mymod/parameters/sec
```

其内容应当是初始值 5。

```
# cat /sys/module/mymod/parameters/sec
5
```

模块参数可以在使用 insmod 添加模块时对值进行指定。

```
# insmod mymod.ko sec=10
```

进行上述操作后，添加 mymod.ko 时模块参数 sec 就为 10，默认间隔 5 秒的超时变成间隔 10 秒。

小结

本节介绍了内核模块的编写方法。编写特有内核模块是 Kernel 构建的入门级操作，你也可以尝试一下。

参考文献

• Documentation/kbuild/modules.txt

——Hiroshi Shimamoto

HACK #4　如何使用 Git

本节介绍 Git 的使用方法。

Git 是 Linux 内核等众多 OSS（Open Source Software，开源软件）开发中所使用的 SCM（Source Code Management，源码管理）系统。在 2005 年以前，在 Linux 内核开发中一直使用一个叫做 BitKeeper 的 SCM。但是由于后来 BitKeeper 的许可证被更改，可能会对开发造成障碍，因此 Linux 不得不改用新的 SCM 进行开发。在这种情况下，Linux 内核的创始人 Linus Torvalds 就开发了 Git，将 Linus 树的仓库转移到了 Git 中。直到 2011 年的今天，Linus 树仍然使用 Git 进行管理，其他大部分的开发树使用的也是 Git。目前 Git 由维护人员滨野纯（Junio C Hamano）等人持续进行开发。

Git 的设计使其能够支持 Linux 或者 OSS 的开发模式。Git 具有这些特征 ：分布式仓库 ；

与互联网具有亲和性；版本更新记录管理不以单个文件为对象，而是将整个源码树作为一个对象；处理速度快等。

Git 具有非常多的功能，如果一一进行说明的话能写成一本书。这里主要针对第一次使用 Git 的读者，通过使用指南的形式介绍日常使用较多的基本功能，同时对相关基本概念进行解说。

笔者使用的是 1.7.1 版本的 Git。

分布式仓库型 SCM

仓库是指保存 SCM 中源代码等信息及历史记录的原始数据的地方。CVS 是以往使用较多的 SCM，而在 CVS 中一直是将源代码从仓库签出（checkout）到本地工作区，进行修改后将代码提交到仓库中。像这样，仓库和工作区明确分开，多个开发者针对单一仓库进行提交的 SCM 称为“单一式仓库型”。

而 Git 采用的是与之相反的“分布式仓库型”结构。在 Git 中，工作区本身就是仓库。也就是说，开发者拥有各自的仓库，它们之间不存在结构层面的上下关系，所有仓库都是并行存在的。在 Linux 内核中一般认为 linus 树的仓库是“中央”仓库，其实这只是大家一致认可的叫法，从 Git 的结构上看，完全没有任何设置是以 linus 树作为中央的。

如上所述，Git 直到完结时都是将本地磁盘的工作区作为仓库的。要能够熟练使用 Git，首先必须掌握如何在本地仓库进行操作。下面将首先讲述在本地仓库进行操作的流程，然后介绍与其他仓库进行协作的方法。

在本地仓库进行操作

创建新的仓库

使用 Git 管理源代码的第一步是创建新的仓库。创建仓库需要创建普通的目录，并将该目录作为 Git 的仓库使用。操作过程如下。

```
$ mkdir -p ~/hello
$ cd ~/hello
$ git init
```

这时，~/hello 就可以作为 Git 的仓库使用了。下面就使用这个仓库来了解一下 Git 的基本功能。

小贴士：Git 的命令以 git <command> 的形式启动。每条命令都有 man page（帮助页面），当想要阅读帮助页面时，请用连字符将 man git-<command> 和子命令连接起来。

例如，当想要阅读 init 子命令的帮助页面时，应写为：

```
$ man git-init
```

如果写成：

```
$ man git init
```

显示出来的就是 git(1) 和 init(8) 的页面。

Git 设置

在进行实际的文件操作前，首先要进行最低限度的必要设置。笔者一般进行的最低设置是提交者（committer）的姓名与邮件地址。在仓库目录下可以执行下列操作来进行这些设置。在这里设置的是笔者的信息。

```
$ git config --add user.email"m_ikeda@hogeraccho.com"
$ git config --add user.name"Munehiro \"Muuhh \" Ikeda"
```

把这个设置写入仓库目录的 .git/config 文件中。执行上面的 git config 命令后，这个文件中就应当写入了下列信息。

```
[user]
      email = m_ikeda@hogeraccho.com
      name = Munehiro \"Muuhh\" Ikeda
```

除此以外，还可以进行很多种设置，这里就先点到为止。

小贴士：写入 .git/config 的设置项目仅能适用于该仓库。

如果为 git config 指定 --global 选项，还可以参照要在用户已启动的所有仓库上共同使用的设置。与之对应的设置文件为 ~/.gitconfig。

当想要参照或添加整个系统的共同设置时，可以使用 --system 选项。与之对应的设置文件是 /etc/gitconfig。

将文件添加到仓库中

现在，尝试创建文件并将其添加到 Git 的仓库中。首先，创建一个 hello.c 文件，其内容如下。

```
/* hello.c */
#include <stdio.h>

int main(void)
{
       printf("Herro world!!\n");
       return 0;
}
```

在 Git 中向仓库增加或修改文件的过程称为“提交”。提交分为两阶段进行，首先指定要提交的对象文件，然后进行实际的提交。

```
$ git add hello.c
$ git commit
```

执行 git commit 后，会启动一个用来输入提交信息的编辑器。由于这是第一次提交，因此在第一行中输入 Initial commit，并保存文件。

小贴士：这里为 git add 明确指定了文件名 hello.c，如果有多个文件，可以使用：

```
$ git add
```

将当前目录下的所有文件作为提交对象。

小贴士：在编辑提交信息时，如果没有保存文件就关闭了编辑器，就不会提交文件。

这样，hello.c 就提交到了仓库中，以后就可以使用 Git 来管理和修改记录等。

修改并提交文件

仔细看一看刚才提交的文件，突然发现 Hello 居然写成了 Herro 了！下面就学习如何进行修改。

将 hello.c 的 printf("Herro world!!\n"); 行修改成 printf("Hello world!!\n"); 保存后提交。对文件进行修改时，也像新增加时一样需要对文件执行 git add 命令，将其作为提交对象。但是，当对多个文件进行修改时，一般会希望先把修改后的文件全部提交。在这种情况下，如果使用 git commit 的 -a 选项，就不需要执行 git add 命令。

```
$ git commit -a
```

这时会像新增加时一样启动编辑器，要求输入提交信息，在输入 Correct misspelling 后保存文件并关闭编辑器。

这样修改内容就提交到了仓库中。

小贴士：git commit -a 原本是用来提交 Git 所管理的所有文件的。一次也没有执行 git add 命令的文件不会提交，例如，新创建的文件等。

确认工作区的状态

如果进行了很多修改，就需要确认已经提交工作区的仓库处于什么状态。我们试着稍微改变源代码来进行确认。

首先将 hello.c 的 return 0; 改为 return 1;。然后创建新文件 goodbye.c。

```
/* goodbye.c */
#include <stdio.h>

int main(void)
{
        printf("Goodbye world!!\n");
        return 0;
}
```

用来确认状态的第一个命令是 git status，尝试执行以下命令。

```
$ git status
# On branch master
# Changed but not updated:
```

```
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   hello.c
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       goodbye.c
no changes added to commit (use "git add" and/or "git commit -a")
```

输出的内容显示 hello.c 的修改还没有提交，goodbye.c 不在 Git 的管理范围内。

要确认哪个文件在 Git 的管理范围内，可以使用下列命令。

```
$ git ls-files
hello.c
```

如果想要看到修改 hello.c 的历史记录，可以执行下列命令。差别就会分段显示出来。

```
$ git diff
diff --git a/hello.c b/hello.c
index aa28db5..7ef0a54 100644
--- a/hello.c
+++ b/hello.c
@@ -4,6 +4,6 @@
 int main(void)
 {
        printf("Hello world!!\n");
-       return 0;
+       return 1;
 }
```

显示出差别的只有 Git 管理范围内的文件。由于 goodbye.c 还不在 Git 的管理范围内，因此没有任何显示。

下面，对 goodbye.c 执行 git add 命令，确认前者是否在 Git 的管理范围内。

```
$ git add goodbye.c
$ git ls-files
goobye.c
hello.c
```

可以看到多出了 goodbye.c。

这时可以再次使用 git diff 来显示差别。但奇怪的是，刚才明明对 goodbye.c 使用了 git add 命令，为什么没有显示呢？这时，再执行下列命令。

```
$ git add hello.c
$ git diff
```

而这次竟然什么也不显示了。这是怎么回事？

事实上，git diff 显示的并不是最新提交与工作区之间的差别，而是“缓存区”（staging area，也称为分段存储区）与工作区之间的差别。缓存区是用来暂时存放下一次要提交到仓库的信息的区域。也就是说，git add 的作用是将当前工作区的内容存放到

缓存区。执行 git commit 后，最新提交和缓存区的内容是一致的。在工作区进行修改后再次执行 git add，最新提交与缓存区的内容就变得不同，而缓存区与工作区的内容一致。在上例中，在执行 git add hello.c 后工作区与缓存区的内容是完全一致的。因此 git diff 就不会再显示任何内容。

当想看到的不是缓存区与当前工作区的差别，而是最新提交与当前工作区的差别时，可以为 git diff 指定表示最新提交的 HEAD。

```
$ git diff HEAD
diff --git a/goodbye.c b/goodbye.c
new file mode 100644
index 0000000..13f79ea
--- /dev/null
+++ b/goodbye.c
@@ -0,0 +1,9 @@
+/* goodbye.c */
+#include <stdio.h>
+
+int main(void)
+{
+       printf("Goodbye world!!\n");
+       return 0;
+}
+
diff --git a/hello.c b/hello.c
index aa28db5..7ef0a54 100644
--- a/hello.c
+++ b/hello.c
@@ -4,6 +4,6 @@
 int main(void)
 {
        printf("Hello world!!\n");
-       return 0;
+       return 1;
 }
```

如果想要知道最新提交与缓存区的差别，可以使用 git diff --cached。由于当前缓存区与工作区是完全一致的，因此输入的内容与上述内容相同。

然后，使用 git commit -a 提交所作的修改，提交信息为 Add goodbye.c。

参照提交记录

当想要参照提交记录时，可以使用 git log 命令。如果在当前仓库执行这条命令，就会显示关于之前进行的 3 次提交的下列相关信息（日期等信息因环境不同而各异）。

```
$ git log
commit 9b670c34bd7bd772648a99738017802f2b24f859
Author: Munehiro "Muuhh" Ikeda <m_ikeda@hogeraccho.com>
Date:   Sat Apr 2 14:09:39 2011 -0700

    Add goodbye.c
```

```
commit c47feeef44a652bda15dbb580d48213dc1294664
Author: Munehiro "Muuhh" Ikeda <m_ikeda@hogeraccho.com>
Date:   Sat Apr 2 13:20:10 2011 -0700

    Correct misspelling

commit 83d9b3f95cdb43e76953c77f03d2700e978dde8d
Author: Munehiro "Muuhh" Ikeda <m_ikeda@hogeraccho.com>
Date:   Sat Apr 2 13:18:07 2011 -0700

    Initial commit
```

紧接着 commit 后面显示的，是仅指示该提交的散列（hash）值。Author 描述的是提交者的信息。事先使用 git config 设置提交者的信息，就是为了将这些作为提交信息记录下来。可以发现，每次提交的最后一行显示的都是提交时输入的提交信息。

git log 默认输出所有的提交记录，但也可以用如表 1-11 所示的命令行参数来指定提交的散列值，限制输出范围。

表 1-11　指定 git log 的提交范围（绝对散列值）

命　令	显示范围
git log <hash>	到提交散列值 <hash> 为止
git log <hash>..	从 <hash> 之后到最新的提交为止
git log <hash1>..<hash2>	从 <hash1> 之后到提交 <hash2> 为止

散列值不需要完整输入，只需输入一定长度使其仅指示这次提交（但最少要输入 4 个字）。

最新提交可以用 HEAD 这一别名进行参照。另外还可以将从 HEAD 开始的相对位置指定为 HEAD~2 等。使用这一方法还可以进行如表 1-12 所示的指定。

表 1-12　指定 git log 的提交范围（相对于 HEAD）

命　令	显示范围
git log、git log HEAD	到最新提交为止，即所有提交
git log HEAD~、git log HEAD~1	到倒数第二次提交为止
git log HEAD~~、git log HEAD~2	到倒数第三次提交为止
git log HEAD~2.. HEAD~1	从倒数第三次提交到倒数第二次提交为止，即仅倒数第二次提交

在 Git 的其他子命令下指定提交范围的方法也是相同的。使用 git diff 等也可以输出指定范围的差别。

如果为 git log 指定文件，则仅输出与该文件有关的提交。还可以使用 -p 选项将提交后的变更内容以段落形式显示出来。当前仓库显示的内容如下。

```
$ git log -p goodbye.c
commit 9b670c34bd7bd772648a99738017802f2b24f859
Author: Munehiro "Muuhh" Ikeda <m_ikeda@hogeraccho.com>
Date:   Sat Apr 2 14:09:39 2011 -0700

    Add goodbye.c

diff --git a/goodbye.c b/goodbye.c
new file mode 100644
index 0000000..13f79ea
--- /dev/null
+++ b/goodbye.c
@@ -0,0 +1,9 @@
+/* goodbye.c */
+#include <stdio.h>
+
+int main(void)
+{
+       printf("Goodbye world!!\n");
+       return 0;
+}
+
```

修改提交

在进行提交后，有时也会想要对已经提交的内容进行修改。修改方法大致可以分为两种。

第一种方法是进行新的提交来取消某个提交。在这种情况下，原先的提交和后来为了取消它而进行的提交都会保留记录。要取消当前仓库的最新提交时，进行如下操作。

```
$ git revert HEAD
```

提交信息可以根据个人喜好进行修改，这里就不作修改，直接以默认内容保存，并关闭编辑器。goodbye.c 从工作区被删除，hello.c 的内容也回到前一次提交的状态（返回值为 0）。使用 git log -p 来确认提交的内容，可以发现使用 git revert 进行的最新提交（提交信息 Revert "Add goodbye.c"）与前一次提交（提交信息 Add goodbye.c）的变更是完全相反的。

第二种方法是直接对提交进行修改。直接修改提交也有三种作法，得出的结果在细节上有一些不同。

直接修改提交的第一种作法，适用于对最新提交进行较小修改的情况。假设在刚才的 git revert 中，想保留 hello.c 的返回值 1，不作修改，并在提交信息中记录下来。在这种情况下需要进行如下操作。

```
$ vi hello.c
（将 return 0; 重新修改为 return 1;）
$ git add hello.c
$ git commit - -amend
```

将提交信息修改为 Revert "Add goodbye.c" except for return value，保存

并关闭编辑器。

再用 git log -p 来确认提交信息与变更内容，可以发现返回值的更改从最新提交中被删除，提交信息也发生了改变。

直接修改提交的第二种作法，就是取消提交。假设想要取消最新提交，也就是将记录恢复到最新提交前一次的提交（HEAD~1），但是希望工作区的源代码维持原状。这时应执行下列命令。

```
$ git reset - -soft HEAD~1
```

然后用 git log 查看记录，可以发现 Revert... 的提交已经消失了。但是由于并没有对工作区作出修改，因此原本通过当前最新提交的 Add goodbye.c 应当添加的文件 goodbye.c 不存在。

最后一种作法，是在恢复记录的同时，将工作区也恢复到相应的状态。可以执行下列命令：

```
$ git reset - -hard HEAD
```

由于工作区也已经回到了当前的 HEAD（即 Add goodbye.c），因此 goodbye.c 恢复。目前记录、工作区都已经完全恢复到提交 Add goodbye.c 后的状态。

小贴士：对提交记录进行修改时请慎重。特别需要注意的是，这个方法如果用在被其他仓库参照的仓库中，会出现相互之间记录不兼容的问题，因此不能在此情况下使用。

为提交加标签

可以为每次提交加上“标签”。为发布版本等关键的提交加上标签，以后就可以使用标签名称来参照本次提交，十分方便。

假设将 HEAD~1 的提交 Correct misspelling 发布为版本 1，然后尝试为 ver1 加上标签。

```
$ git tag ver1 HEAD~1
```

可以使用下列命令来显示仓库内的标签列表。

```
$ git tag -1
```

创建分支

想要在保留当前开发系统的同时进行其他系统的开发，就需要创建“分支”。下面以刚才加了标签 ver1 的提交为起点，创建名为 ver1x 的分支。ver1x 是针对下一版本的开发分支。

```
$ git branch ver1x ver1
```

仓库的分支列表可以用下列命令来显示。

```
$ git branch
```

```
* master
 ver1x
```

如图 1-5 所示，从输出的内容可以看到，新的分支 ver1x 已经创建。前面有 * 的是当前工作区需要的分支（当前分支）。然后，将当前分支更改为 ver1x。

```
$ git checkout ver1x
Switched to branch 'ver1x'
$ git branch
 master
* ver1x
```

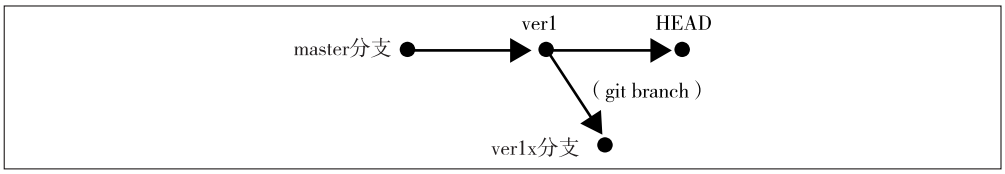

图 1-5　分支的创建

在 ver1x 分支中创建新的文件 thanks.c，并提交。

```
/* thanks.c */
#include <stdio.h>

int main(void)
{
        printf("Thank you guys!!\n");
        return 0;
}
```

```
$ git add thanks.c
$ git commit
```

将 ver1x:Add thanks.c 作为提交信息。

使用 git log 查看记录，可以发现，master 分支里的 Add goodbye.c 在分支 ver1x 内是无效的，提交 Add thanks.c 的祖先是分支 ver1x 的起点，即提交 Correct misspelling。像这样创建分支，就可以在同一仓库内独立地进行其他系统的开发。

rebase 命令

开发版必须一直在最新发行版的基础上进行开发。例如，当发行版安装新功能时，必须将其也安装到开发版中。在当前的仓库中，goodbye.c 就相当于新安装的功能。这时就需要将开发分支 ver1x 的起点移动到发行版的最新提交中。这种分支起点的移动称为复位基底（rebase），如图 1-6 所示。想要将当前分支复位基底到分支 master 的最新提交，需要执行下列命令。由于现在的当前分支为 ver1x，因此这条命令复位基底的是 ver1x。

```
$ git rebase master
```

小贴士：上例中是 rebase 到 master 的最新提交，但可以使用 --onto 选项来指定要 rebase 到的任意提交。默认为所指定分支（上例中为 master）的最新提交。

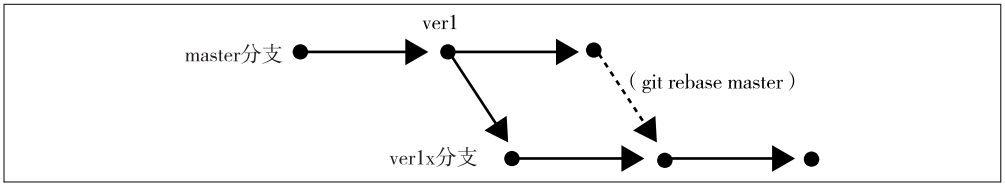

图 1-6　分支的 rebase

合并分支

为了合并发行版 master 分支与开发版分支 ver1x 中各自进行修改与开发的情况，就需要对文件进行修改。

首先在目前需要的 ver1x 分支中进行修改。通过下列命令来修改 goodbye.c 的注释。

```
$ git branch
master
* ver1x
$ vi goodbye.c
（将 /* goodbye.c */ 修改为 /* goodbye.c : needed? */）
$ git commit -a
```

将提交信息写为 ver1x: Modify comment in goodbye.c。然后移动到 master 分支，在这边也对 goodbye.c 进行修改。

```
$ git checkout master
$ vi goodbye.c
（将 /* goodbye.c */ 修改为 /* goodbye.c : yes, needed! */
将 return 0; 修改为 return 1;）
$ git commit -a
```

将提交信息写为 Modify comment and return value of goodbye.c。

到此为止，ver1x 分支下的开发就基本完成了，假设即将将其作为版本 2 进行发布。在这种情况下，需要将分支 ver1x 合并到分支 master 中，将 ver1x 下的开发成果整合到发行版中（见图 1-7）。将当前分支作为 master 执行下列命令。

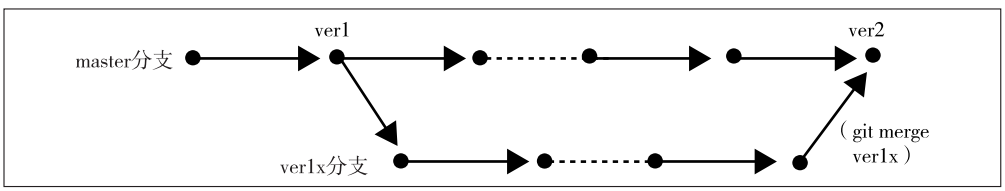

图 1-7　分支的合并

```
$ git merge ver1x
Auto-merging goodbye.c
CONFLICT (content): Merge conflict in goodbye.c
Automatic merge failed; fix conflicts and then commit the result.
```

这时提示由于 goodbye.c 中发生了冲突而无法合并。这是由于在两个分支下都

对goodbye.c进行了修改。发生冲突的文件可以使用git status命令，显示为unmerged。

```
$ git status
goodbye.c: needs merge
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   thanks.c
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       unmerged:   goodbye.c
#
```

再查看一下goodbye.c的内容。

```
<<<<<<< HEAD:goodbye.c
/* goodbye.c : yes, needed! */
=======
/* goodbye.c : needed? */
>>>>>>> ver1x:goodbye.c
#include <stdio.h>

int main(void)
{
        printf("Goodbye world!!\n");
        return 1;
}
```

发生冲突的部分是用冲突标记<<<<<<<和>>>>>>>显示的。必须人工决定选择其中的哪一个。这里选择采用master分支下的修改，即yes,needed!。

将goodbye.c修改为下列内容。

```
/* goodbye.c : yes, needed! */
#include <stdio.h>

int main(void)
{
        printf("Goodbye world!!\n");
        return 1;
}
```

使用git add通知Git修改已结束，并进行提交。

```
$ git add goodbye.c
$ git commit
```

到这一步，两个分支的合并就结束了。使用git log确认记录，可以发现进行合并后，在ver1x分支中进行的ver1x: Add thanks.c等修改在master分支中也体现出来。冲突的解决也作为一个提交记录下来。

然后加上标签，以便将这个状态作为版本 2 进行参照。

```
$ git tag ver2
```

小贴士：即使在两个分支下对相同文件进行了修改，如果是针对不同行进行的修改，Git 也会自动将这些修改合并。上例就在 `master` 分支下修改了 `goodbye.c` 的返回值，这个部分也由 Git 自动进行了合并。

参照图形记录

对分支进行合并后，提交之间的从属关系变得复杂，比较难把握。这时可以使用 `git log --graph` 命令，在文字界面上将从属关系以图形显示出来。

除此以外，也可以使用 `gitk` 数据包里所含的基于图形界面的工具 `gitk`。图 1-8 所示为 `gitk` 的界面。

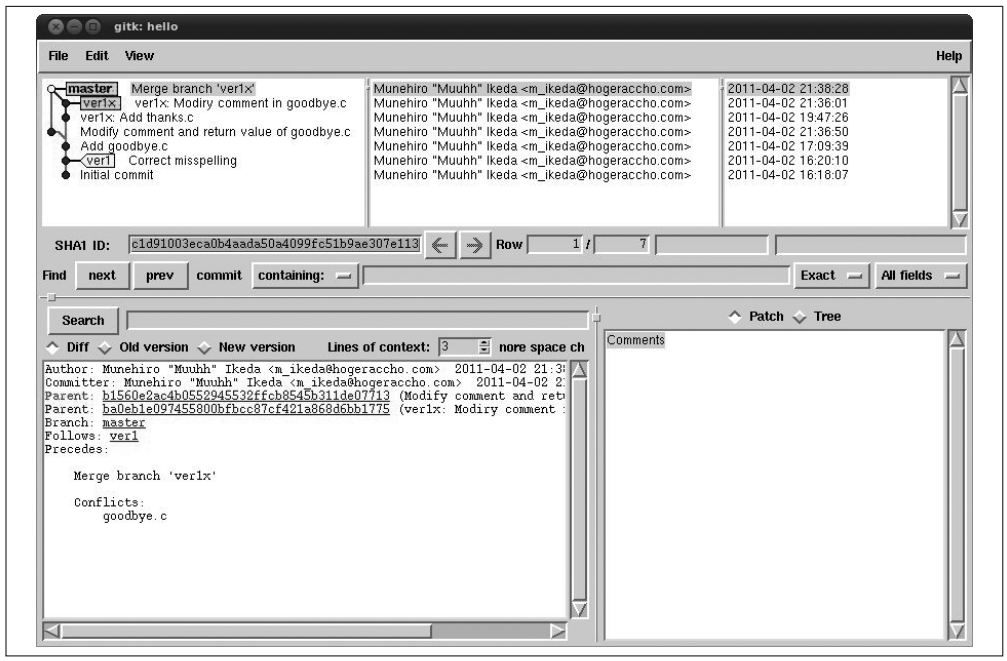

图 1-8　gitk

提取补丁

想要根据从版本 1 到版本 2 的各次提交的差别提取补丁文件，可以执行下列命令。

```
$ git format-patch ver1..ver2
```

当前目录下就会生成 `0001-Add-goodbye.c.patch` 等补丁文件。

在这里使用标签名称指定了补丁的起点和终点，除此以外，还可以指定提交的散列值与分支名。

提取源码树

执行下列命令，可以将版本 2 的源代码作为 tar 文件提取出来。

```
$ git archive -- format=tar - -prefix="hello-v2/" ver2 > ../hello-v2.tar
```

根目录下就会生成名为 hello-v2.tar 的 tar 文件。

小贴士：当各文件包含到 tar 文件中时，文件名前面会加上使用 --prefix 选项指定的文字。这只是单纯地添加了文字，因此，当指定 tar 文件内的根目录名时，要记得如上例中的“hello-v2/”这样在文字的最后加上“/”。

小贴士：.git 目录不会包含到 tar 文件中，因此即使将这里生成的 tar 文件解压缩，也不能发挥 Git 仓库的功能。

相反的，如果将 Git 仓库的目录连同 .git 目录在内全部复制，就能够发挥与原来的仓库完全相同的功能。从这一点也可以看出 Git 完全是分布式仓库。

与远程仓库进行共同作业

本地仓库的操作已经基本掌握，下面就将介绍与远程仓库进行共同作业的方法。

这里将按照一般开发者进行 Linux 内核上游开发时的流程来说明。大致流程如下。

- 将上游的仓库复制到本地。
- 不断追踪上游仓库的最新状态，同时在本地仓库进行开发。
- 以补丁的形式将开发成果提交维护人员及开发邮件列表。

复制仓库

当进行上游开发时，首先要复制各维护人员所管理的开发仓库（远程仓库），建立本地仓库。在 Git 中将这一步称为“复制”。 复制是通过 git clone 命令来进行的。例如，复制 Linus 树的仓库的命令为：

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6
```

复制 Linus 树时要下载 1GB 以上的数据，因此需要花费很长时间，仅在必要时再进行这个操作。这里将之前生成的 hello 仓库当做远程仓库，将其复制到其他位置。

```
$ cd
$ git clone hello local
```

生成 local 目录后，里面包含的文件就与 hello 目录完全相同。进入 local 目录，使用 git log 确认记录，就可以发现至今为止的记录已完全复制过来。

建立本地分支

请在本地仓库执行 git branch 命令。在这一阶段只有 master 分支。这个 master 分

支是在远程仓库（即 hello 仓库）的 master 分支关联的基础上，在本地仓库生成的分支。关联，就是指此后与远程仓库进行同步时，hello 仓库在 master 分支下所作的改动会合并到这个分支。因此，如果在 master 分支中进行开发，进行同步时两个仓库所作的更改就有可能发生冲突。为了避免发生这种情况，就要事先从 master 分支中分出一个用于在本地进行开发的分支 work。

```
$ git checkout -b work
```

git checkout -b 命令将在创建分支的同时进行检查（针对当前分支的最新修改）。

追踪分支

为了便于说明，上文的描述比较简单，可能会让人认为本地仓库的 master 分支是 hello 仓库 master 分支的副本，而其实并不是这样。hello 仓库的 master 分支，在本地仓库是以 origin/master 的标题出现的。这个分支才完全是 hello 仓库 master 分支的副本，这种分支称为“追踪分支”。在使用 git pull 对仓库进行同步时，首先同步的就是这个追踪分支。然后，把追踪分支的提交合并到追踪分支所关联的本地分支中。

通过执行 git branch -r 命令，可以显示追踪分支的列表。下方显示的就是在本地仓库中执行这一命令的输出结果（见图 1-9）。

```
$ git branch -r
  origin/HEAD -> origin/master
  origin/master
  origin/ver1x
```

远程仓库的相关信息可以使用 git remote show 命令来确认。虽然可以设置多个远程仓库，但仅设置一个时其默认名称为 origin，因此执行该命令时可以指定 origin。

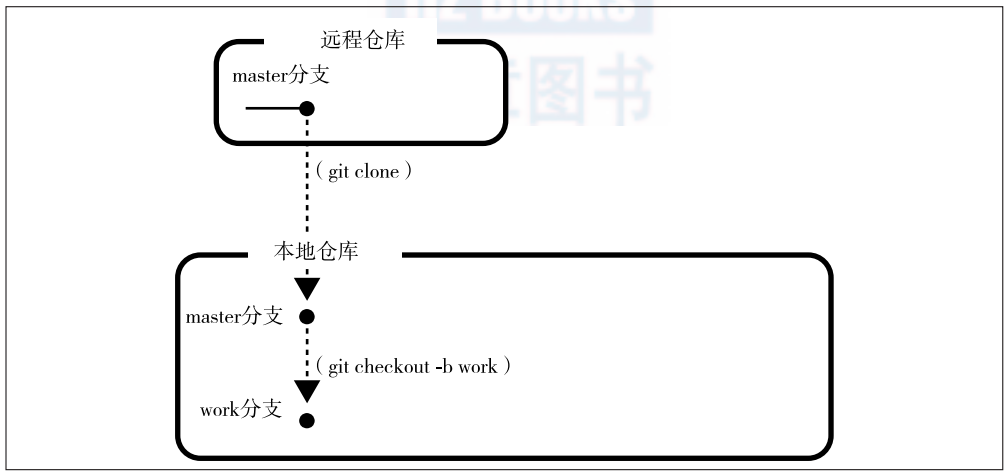

图 1-9　本地分支的建立

```
$ git remote show origin
* remote origin
  URL: /home/munehiro/hello
```

```
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
    master
    ver1x
```

这里的输出具有下列含义。

• 远程仓库 origin 的 URL：/home/munehiro/hello

• 在本地分支 master 上执行 git pull 时合并的远程分支：master

• 追踪分支：master,ver1x

这些信息是通过 .git/config 文件设置的。相关各部分的内容（section）如下所示。[remote "origin"] 部分规定了远程仓库的 URL、远程仓库上的分支、追踪分支之间的关系。[branch "master"] 部分规定了合并到本地分支 master 的远程分支为 origin 仓库（即 hello 仓库）的 master 分支。

```
$ cat .git/config
...
[remote "origin"]
        fetch = +refs/heads/*:refs/remotes/origin/*
        url = /home/munehiro/hello
[branch "master"]
        remote = origin
        merge = refs/heads/master
...
```

追踪分支是为了追踪远程仓库而存在的，因此不能在这个分支上进行本地修改（从技术上是可以的，但并不推荐）。

以追踪分支为起点建立本地分支后，本地分支就被追踪分支关联。例如，可以通过下列命令，建立与 master 分支的追踪分支相关联的本地分支 master2。

```
$ git branch master2 origin/master
```

本地分支 master 及 master2 虽然被关联，但二者完全是本地分支。因此也可以直接在上面进行本地开发。但是，由于需要通过 git pull 进行合并，因此如果发生了冲突，就必须在这时候解决。

与远程仓库同步

想要看到在远程仓库上不断进行的开发，可以在 hello 仓库的 master 分支下对 thanks.c 进行如下修改并提交（将负（-）的行改为正（+）的行）。

```
-     return 0;
+     return 2;
```

```
$ git commit -a
```

将 Modify return value of thanks.c into 2 作为提交信息。

使用 git pull 命令可以让本地仓库与远程仓库的最新状态保持同步。在本地仓库执行下列命令后，在 hello 仓库的 master 分支下进行的修改就会全部整合到本地仓库的 master 分支。

```
$ git checkout master
$ git pull
```

这时使用 git log 查看记录，可以发现 hello 仓库的提交 Modify return value of thanks.c into 2 已经整合并完成同步。

将开发分支 rebase 到最新状态

在本地仓库的 work 分支下不断进行本地开发。将当前分支设置为 work 后，对 thanks.c 进行如下的修改并提交。

```
$ git checkout work

-       printf("Thank you guys!!\n");
-       return 0;
+       printf("Thank you so much guys!!\n");
+       return 1;

$ git commit -a
```

在此以前，提交信息都只有 1 行，而这次需要输入多行，如下所示。第 2 行只需另起一空行。

```
Modify message thanks.c

I really appreciate your efforts.
```

另外，本地开发成果必须基于最新版的远程仓库（上游仓库）。当前的 work 分支是以版本 2 为起点的，而这已经不是最新版。处于最新状态的是刚才进行了同步的本地仓库的 master 分支。因此，如图 1-10 所示，只需要将 work 分支复位基底到 master 分支的 HEAD。

thanks.c 内发生了冲突，可以按照与上文所述“合并分支”同样的方法来消除冲突。为了保留上游的修改，并加入自己的开发成果，需要对 thanks.c 进行如下修改。

```
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Modify thanks.c
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging thanks.c
CONFLICT (content): Merge conflict in thanks.c
Failed to merge in the changes.
Patch failed at 0001 Modify thanks.c

When you have resolved this problem run "git rebase --continue".
If you would prefer to skip this patch, instead run "git rebase --skip".
To restore the original branch and stop rebasing run "git rebase --abort".
```

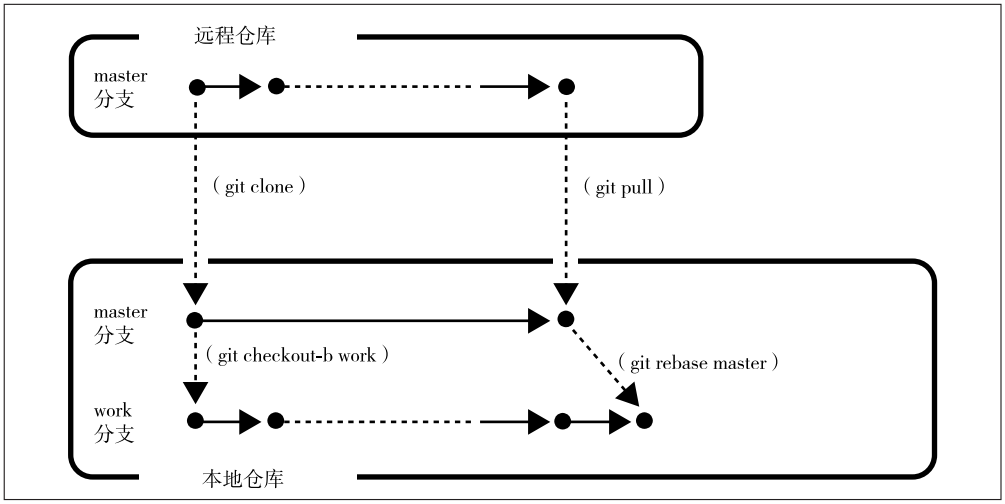

图 1-10　本地开发分支的复位基底

例 1-1　thanks.c 的冲突标记

```
<<<<<<< HEAD
        printf("Thank you guys!!\n");
        return 2;
=======
        printf("Thank you so much guys!!\n");
        return 1;
>>>>>>> Modify thanks.c
```

例 1-2　thanks.c 的修改结果

```
        printf("Thank you so much guys!!\n");
        return 2;
```

因为发生冲突而中断的复位基底，在消除冲突并对文件执行 git add 命令后，再执行 git rebase--continue 就会继续。

```
$ git add thanks.c
$ git rebase --continue
```

这样就成功地复位基底到最新版了。这时也可以使用 git log 确认记录。

用邮件将补丁发送给维护人员

现在，本地仓库的开发终于完成了。Linux 内核开发的流程是先以补丁的形式将开发成果发送到邮件列表，经过评估与讨论后再整合到上游的仓库内。可以使用 git format-patch 将补丁输出为文件，粘贴到平时使用的电子邮件的正文并发送，而 Git 也备有直接发送邮件的功能——git send-email 命令。这里使用这条命令来发送邮件。

另外，在笔者所使用的环境（Ubuntu 10.04、Fedora14）中，git send-mail 命令原来是放在与 Git 不同的 git-email 数据包里的，因此需要事先下载。

小贴士：通过 Git 直接发送邮件，就可以避免电子邮件软件出现的换行符等问题。

要将本地仓库的开发成果，即以 master 分支为起点的 work 分支的各次提交（目前只有一个），作为补丁以邮件发送，可以执行下列命令。另外，如果事先在配置文件中设置各种选项，就不需要每次都在命令行进行输入。表 1-13 所示为与选项对应的配置文件的段落名。

```
$ git send-email --to=hello_maintainer@hogeraccho.com --cc=hello-ml@
hogeraccho.com --smtp-server=smtp.googlemail.com --smtp-encryption=ssl --smtp-
server-port=465 --smtp-user=youruser@gmail.com --smtp-pass=yourpw master..work
```

表 1-13 git send-email **的选项**

选　项	说　明	设置文件段落
--to	邮件的 To 地址	sendmail.to
--cc	邮件的 Cc 地址	sendmail.cc
--smtp-server	SMTP 服务器名称	sendmail.smtp-server
--smtp-encription	连接的保护方式（ssl 或 tls）	sendmail.smtpencryption
--smtp-server-port	SMTP 的端口号	sendmail.smtp-server-port
--smtp-user	SMTP 的用户名	sendmail.smtpuser
--smtp-pass	SMTP 的密码	sendmail.smtp-pass

将 --to、--cc 等改成自己的地址，并尝试发送邮件，应当会收到如下列内容的邮件。

```
From: Munehiro Muuhh Ikeda <m_ikeda@hogeraccho.com>
To: you@your.domain.co.jp
Cc: Munehiro Muuhh Ikeda <m_ikeda@hogeraccho.com>
Subject: [PATCH] Modify message thanks.c

I really appreciate your efforts.
---
 thanks.c |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/thanks.c b/thanks.c
index c28de46..806371d 100644
--- a/thanks.c
+++ b/thanks.c
@@ -3,7 +3,7 @@

 int main(void)
 {
-       printf("Thank you guys!!\n");
+       printf("Thank you so much guys!!\n");
        return 2;
 }
```

```
--
1.7.1
```

提交信息的第 1 行是邮件标题，从空行之后 (即第 3 行开始) 是邮件的正文。

`git send-email` 有非常多的选项，可以进行各种设置。建议浏览帮助页面（man page），进行各种尝试，最后生成最佳的配置文件。

其他有用的命令

除上面介绍的以外，还有很多其他有用的命令。表 1-14 简单整理了其中的一部分，详细内容请参考 git 的帮助页面。

表 1-14　其他命令

命　令	操　作
`git push`	将本地修改直接传递到远程仓库。仅在拥有对远程仓库进行修改的权限时有效
`git fetch`	从远程仓库将修改抓取到追踪分支。与 `git pull` 不同的是不进行与本地分支的合并。`git pull` 是在内部调出 `git fetch` 和 `git merge` 操作的
`git stash`	要将工作区中还未提交的修改保存并移动到其他分支时使用。保存后的状态可以用 `git stash pop` 来调出
`git cherry-pick`	要将其他分支下的 1 个提交到使用当前分支时使用。由于提交会复制，因此会变成其他的散列值
`git gc`	删除未使用的对象或文件，优化仓库
`git am`	从 mbox 格式的文件导入补丁并提交
`git bisect`	用来指定导入了 bug 的提交

小结

当进行 Linux 内核的上游开发时，Git 可以说是必需的工具。Git 具有优秀的功能与速度，在多个开发人员参与的项目中可以成为非常有效的工具，并且不仅限于 Linux 内核的开发。一直以来使用单一仓库型 SCM 的人可以体验分布式仓库型 SCM 的高度可扩展性。当然，对于初次使用 SCM 的人来说也一定会是很好用的工具。

HACK #5　使用 checkpatch.pl 检查补丁的格式

本节介绍发布前检查补丁格式的方法。

Linux 内核是由多个开发者进行开发的。因此，为了保持补丁评估与源代码的可读性，按照统一的规则进行编写是非常重要的。编写规则写在 Linux 内核源代码的

Documentation/CodingStyle 中。所有开发者必须先阅读规则内容，遵照这些规则进行编写后再将补丁发布到论坛上。

话虽如此，但要从一开始就将这些规则完全记住也是不太可能的。因此 Linux 内核的源码树内准备了用来检查补丁格式的脚本 scripts/patchcheck.pl。下面将介绍使用这个脚本来检查补丁格式的方法。

检查格式的示例

首先，看一个对源代码进行一些简单修改并生成补丁的例子。在 fs/namei.c 内的符号链接系统调用（symbolic link system call）的入口函数中添加 printk()。这个补丁 wrong-patch-example.patch 的内容如下所示。

```
From 6064092a8a276fa6e09755872193cfe1e4a16f42 Mon Sep 17 00:00:00 2001
From: Munehiro "Muuhh" Ikeda <m_ikeda@hogeraccho.com>
Date: Sun, 22 May 2011 14:54:58 -0700
Subject: [PATCH] wrong patch example

Added printk() on sys_symlink().
---
 fs/namei.c |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)

diff --git a/fs/namei.c b/fs/namei.c
index e3c4f11..d40214a 100644
--- a/fs/namei.c
+++ b/fs/namei.c
@@ -2912,6 +2912,7 @@ out_putname:

 SYSCALL_DEFINE2(symlink, const char __user *, oldname, const char __user *,
 newname)
 {
+       printk( KERN_DEBUG "[TRIAL] trying symlink: %s --> %s ¥n",oldname,newname);
        return sys_symlinkat(oldname,AT_FDCWD,newname);
}
--
1.7.4
```

当检查补丁的格式时，需在内核源码树的根下，以补丁文件名为变量执行 scripts/checkpatch.pl。当不指定任何选项时，含有格式错误的行的内容也会输出。在这里如果指定 --terse 选项，就可以将各错误或警告的概要分别在 1 行中输出。

```
$ scripts/checkpatch.pl --terse wrong-patch-example.patch
wrong-patch-example.patch:19: WARNING: line over 80 characters
wrong-patch-example.patch:19: ERROR: space prohibited after that open
parenthesis '('
wrong-patch-example.patch:19: ERROR: space prohibited before that close
parenthesis ')'
wrong-patch-example.patch:25: ERROR: Missing Signed-off-by: line(s)
total: 3 errors, 1 warnings, 7 lines checked
```

短短 1 行的补丁，竟然输出了这么多的内容。这些内容依次分别是针对下列内容的错误或警告。

• [警告]1 行的字数超过 80 字。
• [错误] 前括号“(”后面有多余的空格。
• [错误] 后括号“)”前面有多余的空格。
• [错误] 没有 Signed-off-by（补丁发布人的署名）。

前 3 个是关于编写规则的错误或警告，最后 1 个是编写规则之外的补丁格式的错误。

对这些错误进行修改，将较长的行分成两行，删除不需要的空格并添加在 Signed-off-by 后，补丁的内容就如下所示。

```
From cb24866e8c989f55abebc3e6bf879cf3d17d3e87 Mon Sep 17 00:00:00 2001
From: Munehiro "Muuhh" Ikeda <m_ikeda@hogeraccho.com>
Date: Sun, 22 May 2011 14:54:58 -0700
Subject: [PATCH] correct patch example

Added printk() on sys_symlink().

Signed-off-by: Munehiro "Muuhh" Ikeda <m_ikeda@hogeraccho.com>
---
 fs/namei.c |    2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)

diff --git a/fs/namei.c b/fs/namei.c
index e3c4f11..1c47443c 100644
--- a/fs/namei.c
+++ b/fs/namei.c
@@ -2912,6 +2912,8 @@ out_putname:

 SYSCALL_DEFINE2(symlink, const char __user *, oldname, const char __user *,
newname)
 {
+       printk(KERN_DEBUG "[TRIAL] trying symlink: %s --> %s ¥n",
+            oldname,newname);
        return sys_symlinkat(oldname,AT_FDCWD,newname);
}
--
1.7.4
```

使用 checkpatch.pl 输出的主要错误或警告

scripts/checkpatch.pl 输出的错误或警告有很多种，其中有一些比较具有代表性的，如下所示。在编写的阶段就应当充分注意它们。

错误

• 换行符为 DOS 格式（CR+LF）。
• 行首、行尾有多余的空格。
• 不是用制表符，而是用空格缩进。
• switch 语句和 case 语句的缩进不一致。
• 函数定义块 (block) 以外的“{”写在独立的行中。

• 注释符使用的是“//”。
• 全局变量或静态变量是明确指定以 0 初始化的。
• 前括号“(”或“[”后面有多余的空格。
• 后括号“)”或“]”前面有多余的空格。
• 逗号“,”后面没有空格。
• if、for、while 的前括号“(”前面没有空格。
• else 未与 if 块结尾的“}”写在同一行。
• 使用了将来要废弃的头文件或函数。
• 补丁内没有 Signed-off-by 行。

警告

• 补丁内含有的路径起点不是内核源码树的根目录。
• 1 行的长度超过 80 字。
• 制表符前面有空格。
• `const` 关键词的使用方法有问题。
• `printk()` 没有指定输出级别（`KERN_*`）。
• `goto` 的分支终点的标签 `label` 缩进。
• 用“{}”括住了只有 1 行的代码块。
• 使用了 `volatile` 修饰符。
• `kmalloc()` 的返回值已经转换。

小结

使用 `scripts/checkpatch.pl` 可以在投稿前检查补丁的格式。将补丁列入邮件列表时，经常可以看到“未按照规则编写，请修改”的提示。一定要在发布前检查补丁的格式，才能集中对补丁内容进行讨论。

参考文献

• Documentation/CodingStyle（内核源文档）

——Munehiro IKEDA

HACK #6　使用 localmodconfig 缩短编译时间

本节介绍使用 `make localmodconfig` 生成精简的 `.config` 文件，缩短内核编译时间的方法。

为了能够应对各种各样的环境，发布版的内核包含很多内核模块。但是在某个特定机器，例如，大家自己平时使用的 PC 上实际用到的模块只是其中的极小一部分。重新构建内核时，对不使用的模块进行编译就会浪费时间。编译后的模块存放在磁盘里，因此

也会造成磁盘空间的浪费。

将 localmodconfig 作为 make 的目标，就可以生成仅以正在使用的内核模块为对象的 .config 文件，可以在 Linux 内核 2.6.32 以后的版本中使用它。使用这条命令，可以生成仅启用必要模块的 .config 文件，从而缩短内核的编译时间。

localmodconfig 的使用方法

将运行中的内核源代码解压缩，并在源码树的根下执行下列命令。

```
# make localmodconfig
```

如果是一般的发布版，只需进行这一操作就可以生成 .config 文件，将要编译的内核模块缩减到最少。此后只需执行下列命令，照常进行内核的编译、安装。

```
# make
# make modules_install
# make install
```

localmodconfig 的效果

使用两种 .config 文件对上游内核（Linux 2.6.34）进行了编译，并记录分别花费的时间。

第一次是使用 Fedora13 的默认内核 .config 文件对 Linux 2.6.34 进行了编译，将其记为“2.6.34-fc13”。第二次使用的是在 2.6.34-fc13 的 .config 文件的基础上使用 localmodconfig 生成的 .config 文件，编译后的文件记为“2.6.34-localmod”。两次花费的编译时间如下所示。

- 2.6.34-fc13：26 分 13 秒
- 2.6.34-localmod：8 分 20 秒

通过使用 localmodconfig 生成的 .config 文件，将编译时间缩短到了 1/3 以内。

另外，还对内核模块的数量以及它们所在目录的总容量进行了比较。

```
# du -sh /lib/modules/2.6.34-fc13/kernel/
  75M    /lib/modules/2.6.34-fc13/kernel
# find /lib/modules/2.6.34-fc13/kernel -name '*.ko' | wc -l
  2046

# du -sh /lib/modules/2.6.34-localmod/kernel/
  9.2M    /lib/modules/2.6.34-localmod/kernel/
# find /lib/modules/2.6.34-fc13-localmod1/kernel -name '*.ko' | wc -l
  138
```

localmodconfig 使内核模块的数量减少到约 1/15，占用的磁盘空间也减少到约 1/8。

localmodconfig 的结构

localmodconfig 是通过内核源码树的下列脚本执行的。

```
scripts/kconfig/streamline_config.pl
```

localmodconfig 首先会尝试提取一套配置选项作为模型。使用的模型为源码树的 .config 文件或者 /boot 下正在运行的内核的 .config 文件（/boot/config-< 内核版本 >）。当这些不存在时，将从正在运行的内核映像（/boot/vmlinuz-< 内核版本 >）、保存了设置信息的内核模块（configs.ko）等提取信息。

另外，要从内核映像或 configs.ko 提取出配置选项的信息，内核必须是在指定 CONFIG_IKCONFIG 选项的情况下编译的。当无法提取用做模型的配置选项时，即，找不到 .config 文件，且正在运行的内核是在未指定 CONFIG_IKCONFIG 选项的情况下编译的，就会导致 localmodconfig 失败。

然后，localmodconfig 通过 lsmod 获取当前安装到内核中的内核模块列表，找出对它们进行编译所需的配置选项并记录下来。

最后，localmodconfig 将模型的配置选项中指定要作为内核模块进行编译的部分（CONFIG_*=m）进行如下修改，并输出为 .config 文件。

- 当前安装到内核的内核模块所需要的选项：不修改
- 此外：改为禁用

模型中指定要静态安装到内核的选项（CONFIG_*=y）、设置为禁用的选项（# CONFIG_* is not set）不进行修改，直接输出。

小贴士：通过把作为模型的配置选项指定到模块中，却未安装到内核的内核模块中，导致其配置选项失效，无法编译。因此，在执行 localmodconfig 命令之前，可以将需要编译的内核模块手动安装到内核中。例如，可以使用下列命令，将用来虚拟化的模块 kvm.ko 安装到内核中。

```
# modprobe kvm
```

当然，localmodconfig 生成了 .config 文件后，也可以使用 make menuconfig 等手动对 .config 文件进行修改。

---

localyesconfig

localyesconfig 是与 localmodconfig 相似的 make 目标。使用这条命令，通过 localmodconfig 设置为内核模块的配置选项，将设置为在无提示的情况下安装到内核中。

在编写不使用 initramfs 启动的内核时 localyesconfig 非常方便。

小结

对于不是很清楚的配置选项，应当先启用，以避免出现内核无法启动的情况。相信凡是自己构建（make）过内核的人对这一点都深有体会。如果使用 localmodconfig 就不再需要担心这个问题。localmodconfig 既能够节约详细检查 config 选项的时间，又能缩短编译所花费的时间，为我们提供了强有力的支持。

——Munehiro IKEDA

第 2 章

资源管理

Linux 内核的主要工作是资源管理。资源管理这一叫法看似简单，其实管理的资源是非常多样的，包括对 CPU 时间进行分配的进程调度、物理内存的分配与虚拟空间的控制、磁盘 I/O 等。本章将介绍 Linux 内核中的几个资源管理功能。其中 Linux 内核特有的 Cgroup 资源控制，就是能够进行虚拟化资源控制的重要功能。

HACK #7　Cgroup、Namespace、Linux 容器

本节将介绍 Cgroup 与 Namespace 以及通过这两个功能实现的容器功能。

Cgroup

Cgroup（control group）是将任意进程进行分组化管理的 Linux 内核功能。Cgroup 本身是提供将进程进行分组化管理的功能和接口的基础结构，I/O 或内存的分配控制等具体的资源管理功能是通过这个功能来实现的。这些具体的资源管理功能称为 Cgroup 子系统或控制器。

Cgroup 子系统有控制内存的 Memory 控制器、控制进程调度的 CPU 控制器等。运行中的内核可以使用的 Cgroup 子系统由 /proc/cgroup 来确认。

Cgroup 提供了一个 cgroup 虚拟文件系统，作为进行分组管理和各子系统设置的用户接口。要使用 Cgroup，必须挂载 cgroup 文件系统。这时通过挂载选项指定使用哪个子系统。这里指定 debug 这个没有实质功能的调试用子系统来挂载。

```
# mount -t cgroup -o debug cgroup /cgroup
```

注意事项：这里所说的“虚拟文件系统”，是指 procfs 和 sysfs 这种不具有物理设备的文件系统。并不是用来接纳文件系统差异的内核内部层 layerVFS。

小贴士：关于 cgroup 文件系统的标准化挂载要点，是由开发论坛进行讨论的，但目前尚未得出结论。这里是挂载到 /cgroup。

挂载后，在挂载位置下应该可以看到下列几个文件。这些是 Cgroup 呈现出来的特殊文件。

```
# ls /cgroup
    cgroup.event_control    debug.current_css_set           debug.taskcount
    cgroup.procs            debug.current_css_set_cg_links  notify_on_release
    debug.cgroup_css_links  debug.current_css_set_refcount  release_agent
    debug.cgroup_refcount   debug.releasable                tasks
```

文件名前缀为 cgroup 的以及没有前缀的文件是由 Cgroup 的基础结构提供的特殊文件。而前缀为 debug 的文件是由 debug 子系统提供的特殊文件。Cgroup 的子系统提供的特殊文件都会像这样加上子系统的前缀。因此，根据挂载时指定的选项，即所使用的子系统不同，存在的特殊文件也不同。但是 Cgroup 的基础结构所提供的特殊文件则是无论指定哪种子系统都一直存在的。特殊文件分为只读文件和可读写文件。只读文件是为用户提供信息的文件。可读写的特殊文件通过写入值来更改 Cgroup 以及 Cgroup 子系统设置的文件。设置的值可以通过读入特殊文件来确认。在这些特殊文件中，最重要的是 tasks 特殊文件。其内容可以显示如下。

```
# cat /cgroup/tasks
   1
   2
   3
   4
   ...
```

虽然看上去只是一些数字的排列，但其实这些是属于这个分组的线程的线程 ID（TID）。在这时，系统上运行的所有线程的 TID 都包含在 /cgroup/tasks 中。这就表示全部线程都属于这个分组。那么这里出现的“分组”又是什么呢？分组，就是体现为 cgroup 文件系统目录的线程的集合。由于 /cgroup 也是目录，因此它也表示一个分组。像这样位于挂载点最上层的目录是自动生成的分组，称为根分组。在这个阶段，只有 /cgroup（即根分组）是系统上存在的唯一分组。

小贴士：英语中将通过 Cgroup 创建的分组称做 cgroup，容易与表示结构的“Cgroup”混淆，所以这里仅称为“分组”。

下面尝试创建一个分组，也就是在 /cgroup 下创建子目录。其内容如下所示。

```
# mkdir /cgroup/test
# ls /cgroup/test
    cgroup.event_control    debug.current_css_set           debug.taskcount
    cgroup.procs            debug.current_css_set_cg_links  notify_on_release
    debug.cgroup_css_links  debug.current_css_set_refcount  tasks
    debug.cgroup_refcount   debug.releasable
```

虽然是新生成的目录，但是已经有文件存在。cgroup 文件系统在目录生成的同时就会在其中配置特殊文件。

/cgroup/test 也和 /cgroup 一样有 tasks。其内容如下。

```
# cat /cgroup/test/tasks
```

tasks 的内容似乎是空的，这表示这个分组内一个线程也没有。可以将适当的线程添加

到这个分组中。要将线程添加到分组中，可以在 tasks 中写入该线程的 TID。这里以添加 shell 本身为例。

```
# echo $$
    2474
# echo $$ > /cgroup/test/tasks
# cat /cgroup/test/tasks
    2474
    3821
```

tasks 的内容中包含 shell 的 TID（也是 PID，即进程 ID）——2474，可以看出这个 shell 已经属于 test 分组。除此以外，这个分组内还有另一个 TID 为 3821 的线程，这是什么呢？我们再来看一下 tasks 的内容。

```
# cat /cgroup/test/tasks
   2474
   3822
```

结果居然发生了变化。事实上这个改变的部分，是显示了 tasks 内容的 cat 进程的 TID。最初的 cat（3821）和第二次的 cat（3822）是不同的进程，TID 也不同，所以结果发生了变化。但是似乎并没有将 cat 进程添加到 test 分组中。其实，属于分组的进程一旦生成子进程，其子进程就会自动属于母进程。由于 cat 是 shell 的子进程，因此前者自动属于 test 分组。大家应该还记得，在挂载 cgroup 文件系统后，系统上的所有线程是属于根分组的。也就是说，除了将明确指定为新生成分组内的进程为祖先进程以外，生成的进程都属于根分组。

这时，再显示 /cgroup/tasks 的内容的话，应该不会显示 shell 的 TID（2474）。这是因为 shell 不属于根分组，而是属于 test 分组。然后，再将这个 shell 返回到根分组。

```
# echo 2474 > /cgroup/tasks
```

这样，shell 的 TID（2474）就再次属于 /cgroup/tasks，而 /cgroup/test/tasks 就变空。如果分组中一个线程也没有，可以进行撤销。删除目录就可以撤销分组。

```
# rmdir /cgroup/test
```

表 2-1 是每个子系统中 Cgroup 都会提供的特殊文件列表。

表 2-1　Cgroup 提供的文件种类

文　件　名	R/W	用　　途
release_agent	RW	删除分组时执行的命令。这个文件只存在于根分组
notify_on_release	RW	设置是否执行 release_agent。为 1 时执行
tasks	RW	属于分组的线程 TID 列表
cgroup.procs	R	属于分组的进程 PID 列表。仅包括多线程进程的线程 leader 的 TID，这点与 tasks 不同
cgroup.event_control	RW	监视状态变化和分组删除事件的配置文件

这里仅介绍了最基本的 Cgroup 使用方法，也就是分组的创建、撤销和将线程添加到分组的方法。实际使用 Cgroup 时，应在将线程添加到分组后，在分组内的特殊文件中设置值，来控制系统的运行。

Namespace

使用 Namespace（命名空间），可以让每个进程组具有独立的 PID、IPC 和网络空间。

可以向 clone 系统调用的第 3 个参数 flags 设置划分命名空间的标志，通过执行 clone 系统调用可以划分命名空间。

例如，划分 PID 命名空间后，在新生成的 PID 命名空间内进程的 PID 是从 1 开始的。从新 PID 为 1 的进程 fork() 分叉得到的进程，被封闭到这个新的 PID 命名空间，与其他 PID 命名空间分隔开。在新创建的 PID 命名空间中生成的进程，其 PID 有可能与存在于原 PID 命名空间中的进程相同，但由于二者的 PID 命名空间划分开，就不存在相互影响。

同样，也可以用 PID、网络、文件系统的挂载空间、UTS（Universal Time sharing System）为对象进行资源划分。可以在 clone 系统调用的第 3 个参数中设置资源划分的种类，如表 2-2 所示。

表 2-2　资源划分

名　　称	说　　明
CLONE_NEWIPC	划分 IPC（进程间通信）命名空间。信号量（semaphore）、共享内存、消息队列等进程间通信用的资源
CLONE_NEWNET	划分网络命名空间。分配网络接口
CLONE_NEWNS	划分挂载的命名空间。与 chroot 同样分配新的根文件系统
CLONE_NEWPID	划分 PID 命名空间。分配新的进程 ID 空间
CLONE_NEWUTS	划分 UTS 命名空间。分配新的 UTS 空间

Linux 容器

使用 Cgroup 和 Namespace 就可以实现容器。容器这个技术也称为操作系统虚拟化，是将一个内核所管理的资源划分成多个分组。

在容器中，CPU 和内存资源是使用 Cgroup 来划分的。PID、IPC、网络等资源使用 Namespace 来划分。

LXC

Linux 中实际安装的容器有 LXC（Linux Container）。本节将以 Fedora 14 为例介绍 LXC 的使用方法。

```
# yum install lxc
```

要使用网络，还需要安装 bridge-utils。

```
# yum install bridge-utils
```

在使用 LXC 之前，必须启用 cgroup 文件系统。使用下列命令挂载 cgroup 文件系统。

```
# mount -t cgroup cgroup /cgroup
```

另外，向 /etc/fstab 添加下列语句，就可以在系统启动时自动挂载 cgroup 文件系统。

```
cgroup /cgroup cgroup defaults 0 0
```

首先，将 bash shell 进程放进容器。

这里要为容器中使用的文件系统准备一个 /lxc 目录。

```
# mkdir /lxc
# cd /lxc
```

然后准备作为容器内的根文件系统的目录。

```
# mkdir rootfs
# cd rootfs
```

还需要准备其他必要的目录（这些目录主要使用 bind mount）。

```
# mkdir bin dev etc lib lib64 proc sbin sys usr var
```

然后还要生成 LXC 的配置文件 lxc.conf 以及引用的 fstab。

```
# vi /lxc/lxc.conf
lxc.utsname = lxc
lxc.rootfs = /lxc/rootfs
lxc.mount = /lxc/fstab
```

```
# vi /lxc/fstab
/bin /lxc/rootfs/bin none ro,bind 0 0
/sbin /lxc/rootfs/sbin none ro,bind 0 0
/lib /lxc/rootfs/lib none ro,bind 0 0
/lib64 /lxc/rootfs/lib64 none ro,bind 0 0
/etc /lxc/rootfs/etc none ro,bind 0 0
/usr /lxc/rootfs/usr none ro,bind 0 0
/dev /lxc/rootfs/dev none rw,bind 0 0
/dev/pts /lxc/rootfs/dev/pts none rw,bind 0 0
/proc /lxc/rootfs/proc proc defaults 0 0
/sys /lxc/rootfs/sys sysfs defaults 0 0
```

准备工作完成后，使用 lxc-create 命令生成名为 lxc 的容器。

```
# lxc-create -n lxc -f /lxc/lxc.conf
```

使用 lxc-ls 命令可以确认容器列表。

```
# lxc-ls
lxc
```

使用 lxc-create 命令在 lxc 容器内执行 bash。

```
# lxc-execute -n lxc bash
```

小贴士：在笔者的环境下，出现了终端的按键输入不显示的情况。发生这种情况时可以执行reset命令，终端的操作就会恢复。

```
bash 4.1# reset（不显示在画面上）
```

执行ps命令，就可以发现PID是从1开始的，除lxc容器以外看不到其他进程。

```
bash-4.1# ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.1  0.1  14688   664 pts/0     S    21:34   0:00 /usr/lib64/
lxc/lxc-init -- bash
root          2  0.3  0.3 108440  1760 pts/0    S    21:34   0:00 bash
root          6  0.0  0.2 108120  1104 pts/0    R+   21:34   0:00 ps aux
```

另外，生成的容器可以使用lxc-destroy命令来撤销。

```
# lxc-destroy -n lxc
```

接下来尝试启用网络，并启动sshd进程。然后，创建用来连接分配到容器的网络接口的网桥（关于网桥请参考Hack #24）。

在这个例子中将IP地址设置为192.168.20.254。

```
# brctl addbr br0
# ifconfig br0 192.168.20.254
```

然后，修改lxc.conf，添加网络设置。

```
# vi /lxc/lxc.conf
lxc.utsname = lxc
lxc.rootfs = /lxc/rootfs
lxc.mount = /lxc/fstab
lxc.network.type = veth
lxc.network.flags = up
lxc.network.link = br0
lxc.network.name = eth0
lxc.network.ipv4 = 192.168.20.1/24
```

启动sshd时需要下列目录，要事先创建。当该目录不存在时，从lxc-execute启动sshd时就会失败。

```
# mkdir -p rootfs/var/empty/sshd
```

接下来生成容器。

```
# lxc-execute -n lxc /usr/sbin/sshd
```

这时打开其他终端，确认SSH服务器是否正在运行。

首先，使用ping命令确认网络是否已连接。

```
# ping 192.168.20.1
PING 192.168.20.1 (192.168.20.1) 56(84) bytes of data.
64 bytes from 192.168.20.1: icmp_req=1 ttl=64 time=0.899 ms
64 bytes from 192.168.20.1: icmp_req=2 ttl=64 time=0.174 ms
^C
```

使用 ssh 命令，连接分配到容器的 IP 地址 192.168.20.1。

```
# ssh 192.168.20.1
root@192.168.20.1's password:
```

SSH 连接已建立，输入密码后就成功登录。

通过 ps 命令所显示的进程来确认资源是否已被容器隔离。

```
-bash-4.1# ps auxw
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.1  14688   660 pts/0    S+   21:43   0:00 /usr/lib64/lxc/
lxc-init -- /usr/sbin/sshd
root          3  0.0  0.2  75104  1116 ?        Ss   21:43   0:00 /usr/sbin/sshd
root          4  1.4  0.8 108816  4068 ?        Ss   21:47   0:00 sshd: root@pts/3
root          6  1.6  0.3 108440  1904 pts/3    Ss   21:47   0:00 -bash
root         19  0.0  0.2 108124  1104 pts/3    R+   21:47   0:00 ps auxw

-bash-4.1# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 3A:A1:C2:A0:6F:1B
          inet addr:192.168.20.1  Bcast:192.168.20.0  Mask:255.255.255.0
          inet6 addr: fe80::38a1:c2ff:fea0:6f1b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:280 errors:0 dropped:0 overruns:0 frame:0
          TX packets:259 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:31389 (30.6 KiB)  TX bytes:31373 (30.6 KiB)

-bash-4.1# exit
logout
```

可以在其他终端执行 lxc-stop，来关闭装有 sshd 的容器。

```
# lxc-stop -n lxc
```

然后运行 debian。

使用 debootstrap 创建用来启动 debian 的根文件系统。deboot strap 使用 yum 来安装。

```
# yum install debootstrap
```

创建 debian 的根文件系统，需要准备 /debian。

```
# mkdir /debian
# cd /debian
```

现在执行 debootstrap，生成 debian lenny 的文件系统。

```
# debootstrap - -arch=amd64 lenny lenny
```

然后，准备 LXC 用的设置。

```
# vi /debian/lenny.conf
lxc.utsname = lenny
lxc.network.type = veth
```

```
lxc.network.flags = up
lxc.network.link = br0
lxc.network.name = eth0
lxc.network.ipv4 = 192.168.20.2/24
lxc.rootfs = /debian/lenny
lxc.mount = /debian/lenny.fstab
```

```
# vi /debian/lenny.fstab
devpts /debian/lenny/dev/pts devpts defaults 0 0
proc /debian/lenny/proc proc defaults 0 0
sysfs /debian/lenny/sys sysfs defaults 0 0
```

创建名称为 lenny 的容器。

```
# lxc-create -n lenny -f lenny.conf
```

所创建的容器可以通过 lxc-start 来启动。只到这一步的话，init 虽然启动，但不能进行任何操作。这是因为刚执行 debootstrap 后，安装的仅是最低限度需要的数据包。这里为了让外部能够连接到容器，需要安装 sshd。

```
# lxc-start -n lenny bash
```

debian 环境的 shell 就会在容器内启动。然后使用 apt-get 安装 openssh-server。

```
lenny:~# apt-get install openssh-server
```

另外，为了在 SSH 上进行登录，还需要设置 root 的密码。

```
lenny:~# passwd
```

在容器内启动 lenny。

```
# lxc-start -n lenny
INIT: version 2.86 booting
Setting the system clock.
Cannot access the Hardware Clock via any known method.
Use the --debug option to see the details of our search for an access method.
Unable to set System Clock to: Fri Dec 10 22:59:45 UTC 2010 (warning).
Activating swap...done.
Setting the system clock.
Cannot access the Hardware Clock via any known method.
Use the --debug option to see the details of our search for an access method.
Unable to set System Clock to: Fri Dec 10 22:59:46 UTC 2010 (warning).
Cleaning up ifupdown....
Loading kernel modules...FATAL: Could not load /lib/modules/2.6.35.9-64.fc14.
x86_64/modules.dep: No such file or directory
Checking file systems...fsck 1.41.3 (12-Oct-2008)
done.
Setting kernel variables (/etc/sysctl.conf)...done.
Mounting local filesystems...done.
Activating swapfile swap...done.
Setting up networking....
Configuring network interfaces...done.
INIT: Entering runlevel: 2
Starting enhanced syslogd: rsyslogd.
```

```
Starting OpenBSD Secure Shell server: sshd.
Starting periodic command scheduler: crond.
```

我们尝试从其他终端使用 SSH 来连接。

```
# ssh 192.168.20.2
root@192.168.20.2's password:
lenny:~#
```

在 debian 环境下实施关闭（shutdown）的话，容器结束。

```
lenny:~# shutdown -h now
```

按照前面所述方法使用 LXC 就可以简单地创建容器。

小结

本节介绍了 Linux 内核的资源划分功能：划分 CPU、内存空间、I/O 等的 Cgroup，以及划分 PID、IPC、网络、mount 命名空间的 Namespace。另外，还介绍了实际安装上述资源划分功能的容器 LXC。

参考文献

- man 2 clone
- man 2 unshare
- LXC
  http://lxc.sourceforge.net/
  http://www.ibm.com/developerworks/jp/linux/library/l-lxc-containers/
- debootstrap
  http://www.debian.org/releases/stable/i386/apds03.html.ja

——Munehiro IKEDA, Hiroshi Shimamoto

HACK #8　调度策略

本节介绍 Linux 的调度策略（scheduling policy）。

Linux 调度策略的类别大致可以分为 TSS（Time Sharing System，分时系统）和实时系统这两种。

一方面，一般的进程是通过分时运行的。也就是说，使用 CPU 的时间达到分配给进程的时间（时间片）时，就会切换到其他进程。这种分时运行的调度策略称为 TSS。

另一方面，在实时制约较严格且要求保证实时的处理中，就需要指定静态的执行优先级，并严格按照执行优先级进行调度。对这种对应答性有要求的进程，可以使用实时调度策略。另外，与 TSS 调度策略的进程相比，CPU 将优先分配给使用实时调度策略的进程。

在 Linux 中，进程的静态优先级为 0 ～ 99。TSS 调度策略的优先级为 0，实时调度策略的优先级可以指定的范围为 1 ～ 99。

Linux 2.6.23 以前一直采用的 O（1）调度程序，还会在 TSS 调度策略中添加动态优先级。长时间持续使用 CPU 的进程，其调度的动态优先级会渐渐降低。Linux 的进程调度程序是按照优先级来分配 CPU 的，因此长时间占用 CPU 的进程优先级与其他进程相比就相对较低。与之相对的是，频繁与用户进行交流的 shell 等对话进程由于 CPU 使用时间短，其调度的优先级变高，更容易分配到 CPU。因此，为等待用户输入而立刻腾出 CPU 的 shell 进程的优先级变得比其他进程高，用户的应答性就得到提高。

在 Linux 2.6.23 导入的 CFS（Completely Fair Scheduler）调度程序中，没有之前的 O（1）调度程序所进行的经验性优先级变化。CFS 通过称为公平的 CPU 时间分配的结构来运行。

调度策略

RHEL6（Linux 2.6.32）中定义了下列调度策略。

```
SCHED_OTHER
SCHED_FIFO
SCHED_RR
SCHED_BATCH
SCHED_IDLE
```

下面将分别对各调度策略进行介绍。

SCHED_OTHER

这是 Linux 的标准调度策略，也是所谓 TSS 调度策略。

在 RHEL5 等 Linux 2.6.23 之前的内核所使用的以优先级为基础的 O(1) 调度程序中，还加入了经验性的判断，优先为会话进程赋予执行权。TSS 的时间片由优先级决定。

在 RHEL6 等 Linux 2.6.23 之后的 CFS 中，会公平地为所有 TSS 策略的进程分配 CPU 时间。其时间片是动态决定的。

SCHED_FIFO

这是实时调度策略，即具有静态优先级的调度策略。Linux 内核中能够为实时调度策略的进程指定的优先级为 1 ～ 99。使用了 SCHED_FIFO 调度策略的进程，除了等待 I/O 完成时休眠、自发休眠或优先级更高的实时进程获得优先权以外，不会释放执行权。

使用 SCHED_FIFO 的实时调度策略时，需要注意的是，它的进程不会自动释放 CPU，也就是说执行权不会转移到其他进程。例如，实时调度策略的进程陷入无限循环时，其他所有优先级较低的进程永远不会被赋予执行权，此时系统就会死机。

小贴士：另外，要对进程使用实时调度策略，必须有 root 权限。

SCHED_RR

这也是实时调度策略。RR 是 round robin（轮询）的缩写，与 SCHED_FIFO 不同的是，它具有时间片。时间片使用完时，执行权将转移到其他进程。

在 2.6.23 以前导入的 O（1）调度程序中，时间片是由优先级决定的。

引入 CFS 时 SCHED_RR 的调度策略也进行了修改，时间片变为固定值（100 毫秒）。

SCHED_BATCH

指定这个调度策略的进程不是会话型，不会根据休眠时间更改优先级。

例如，备份处理等需要进行较大文件或大量文件存取的进程，是通过磁盘 I/O 来中止的。在 TSS 调度策略中，因为这个休眠，正在进行备份处理的进程优先级提高，需要应答性的 shell 等的优先级相对降低。这就会导致系统的应答性降低。

在 RHEL5 的 O（1）调度程序中，使用了这个调度策略的进程被识别为休眠时间为 0 的 CPU bound 进程。因此，优先级必然会变成比会话型 shell 进程低。

对非会话型的进程（即所谓的补丁处理）使用这个调度策略，就可以使会话型进程的优先级保持相对较高，并确保应答性。

在 Linux 2.6.23 导入的 CFS 中，对进行补丁处理的进程改变了处理的方法，优先级不会因休眠时间而发生变化。在导入 CFS 的 RHEL6 中，SCHED_BATCH 和 SCHED_OTHER 几乎没有区别，因此可以不使用。

SCHED_IDLE

这是由 CFS 导入的新等级。CPU 空闲时，即 SCHED_IDLE 等级以外处于可执行状态的进程消失时，将被赋予执行权。也就是它将成为优先级最低的进程。

特殊标志：SCHED_RESET_ON_FORK

为了限制实时调度策略的进程运行，而为调度策略添加了标志 flag。设置了标志 flag 的实时调度策略进程，在执行 `fork()` 时，新生成的子进程就成为 `SCHED_OTHER` 策略的进程。

如下例所示，通过向实时调度策略添加标志 flag 来设置。

```
sched_setscheduler(pid, SCHED_FIFO|SCHED_RESET_ON_FORK, &param);
```

关于调度策略的系统调用

关于调度策略的系统调用如下所示。

`sched_setscheduler()`

更改调度策略和进程优先级。

`sched_getscheduler()`

获取当前调度策略与进程优先级。

`sched_setparam()`

更改调度参数 (即进程优先级)。

`sched_getparam()`

获取当前调度参数。

```
sched_get_priority_max()
sched_get_priority_min()
```

获取调度策略的进程的静态优先级范围。

`sched_rr_get_interval()`

获取当前时间片。

chrt 命令

用户使用 `chrt` 命令可以很简单地更改调度策略。RHEL5 版本的 `chrt` 命令中不存在指定 `SCHED_IDLE` 的 `-i` 选项。

在 CentOS5（RHEL5）中 `chrt` 的使用方法如下所示。

```
$ chrt --help
chrt (util-linux 2.13-pre7)
usage: chrt [options] [prio] [pid | cmd [args...]]
manipulate real-time attributes of a process
  -b, --batch                   set policy to SCHED_BATCH
  -f, --fifo                    set policy to SCHED_FF
  -p, --pid                     operate on existing given pid
  -m, --max                     show min and max valid priorities
  -o, --other                   set policy to SCHED_OTHER
  -r, --rr                      set policy to SCHED_RR (default)
  -h, --help                    display this help
  -v, --verbose                 display status information
  -V, --version                 output version information
```

下面是 Fedora 12（RHEL6）中 `chrt` 的使用方法。

```
$ chrt --help

chrt - manipulate real-time attributes of a process.

Set policy:
  chrt [options] <policy> <priority> {<pid> | <command> [<arg> ...]}

Get policy:
```

```
  chrt [options] {<pid> | <command> [<arg> ...]}


Scheduling policies:
  -b | --batch          set policy to SCHED_BATCH
  -f | --fifo          set policy to SCHED_FIFO
  -i | --idle           set policy to SCHED_IDLE
  -o | --other          set policy to SCHED_OTHER
  -r | --rr             set policy to SCHED_RR (default)

Options:
  -h | --help           display this help
  -p | --pid            operate on existing given pid
  -m | --max            show min and max valid priorities
  -v | --verbose        display status information
  -V | --version        output version information
```

使用 chrt 命令，可以更改进程的调度策略和优先级。例如，使用 SCHED_IDLE 解压缩内核源代码存档时的命令行如下所示。

```
$ chrt -i o tar jxf linux-2.6.33.tar.bz2
```

正在运行的进程的调度策略也可以通过指定目的进程的 PID 来更改。

```
# chrt -p -r 99 <pid>
```

另外，使用实时调度策略，必须具有 root 权限。

小结

本节介绍了可以使用 Linux 进程调度程序指定的调度策略。可以尝试修改对备份处理和要求实时性的进程的调度策略。

参考文献

• CFS
  http://www.ibm.com/developerworks/jp/linux/library/l-cfs/?ca=dnj-0208
• man sched_setscheduler 他
  http://www.linux.or.jp/JM/html/LDP_man-pages/man2/sched_setscheduler.2.html

——Hiroshi Shimamoto

HACK #9　RT Group Scheduling 与 RT Throttling

本节介绍对实时进程所使用的 CPU 时间进行限制的功能 RT Group Scheduling 和 RT Throttling。

RT Group Scheduling 和 RT Throttling 功能是用来限制使用实时调度策略的进程的 CPU 时间。内核 2.6.25 以后的版本都可以使用这个功能。

本节将介绍如何使用 RT Scheduling 和 RT Throttling 来限制实时进程的 CPU 时间。

为了让 Linux 系统能够应用到需要实时性的领域，Linux 的进程调度程序采用的是实时调度策略（参考 Hack #8）。

实时调度策略具有静态优先级，调度的优先级比其他一般进程高，需要执行时一定会分配 CPU 时间。如果实时进程陷入无限循环，就会占用 CPU，其他处理完全无法运行。

该功能通过限制实时进程的 CPU 时间，使执行权即使在这种情况下也能切换到其他进程，可以避免产生系统死机的问题。

实时

实时功能的目的是实现满足实时限制的处理，即，在有限时间内得到处理结果。也就是将对特定事件进行处理的延迟控制在一定时间以内，在有意义的时间内一定作出应答的功能。因此，即使在内核运行过程中，也能迅速切换到要处理事件的进程。因此内核内部设置了优先权点（preemption point），可以根据事件立刻切换到实时进程。

要求实时性的处理，如图形处理。在实时图形处理中，画面更新的处理应当配合显示器上显示的刷新来进行。这个实时处理中重要的是要在一定时间（刷新率）内完成图形处理。

分配 CPU 时间提高吞吐量的目的和实时的目的是不同的，这点经常容易混淆。

RT Throttling

RT Throttling 是对分配给实时进程的 CPU 时间进行限制的功能。使用实时调度策略的进程由于 bug 等出现不可控错误时，完全不调度其他进程，系统就会无响应。通过限制分配给实时进程的每个单位时间的 CPU 时间，就可以防止使用实时调度策略的进程出现 bug。

还可以指定单位时间内分配多少 CPU 时间给实时进程。标准设置的单位时间是 1 秒，CPU 分配时间是 0.95 秒，非实时进程每 1 秒也可以使用 CPU 0.05 秒。

可是对分配给实时进程的 CPU 时间进行限制，会不会对实时处理造成影响呢？答案是不会。正如在关于实时性的介绍中提到的，对某个处理使用实时策略，是为了满足实时限制，即在一定时间内完成处理。如果对实时性有要求的进程占用 CPU 时间，就不能实现实时性。

为使用实时调度策略的进程的处理分配所必需的或实时限制量的 CPU 时间，就可以防止系统的实时进程出现不可控错误等意外情况。

系统的整体设置

整个系统的 CPU 时间设置可以使用 `sysctl` 来获取、设置。最近的内核都可以通过

sysctl 来限制实时进程能够使用的 CPU 时间。

下列为获取当前值的例子。这个例子中使用的是标准设置，单位时间为 1 秒，CPU 分配时间为 0.95 秒。

```
$ sysctl -n kernel.sched_rt_period_us
1000000
$ sysctl -n kernel.sched_rt_runtime_us
950000
```

设置示例

要将 CPU 分配时间改为 0.9 秒，可以执行下列操作。

```
# sysctl -w kernel.sched_rt_runtimes_us=900000
```

另外，将 CPU 分配时间指定为 –1，对实时进程的 CPU 时间限制就会消失。这与内核导入该功能之前的行为是一样的。

```
# sysctl -w kernel.sched_rt_runtime_us=-1
```

当然，也可以从 proc 文件系统存取。

```
/proc/sys/kernel/sched_rt_period_us
/proc/sys/kernel/sched_rt_runtime_us
```

当 CONFIG_RT_GROUP_SCHED 有效时，受到 Cgroup 设置值的限制，不能进行与 Cgroup 中的有效值相矛盾的设置。但是在这里，将 sched_rt_runtime_us 设置为 –1，是用来使 RT Throttling 失效的设置。

一般来说，sysctl 中的设置仅用于有效（启用）与无效（关闭）的切换，单个设置需要使用 Cgroup 来进行。

Cgroup 中的设置

RT Group Scheduling 是 Cgroup 的子系统。要使用 RT Group Scheduling，必须启用 CONFIG_RT_GROUP_SCHED。可以与其他 Cgroup 一样通过 cgroup 文件系统进行设置（参考 Hack #7）。

```
# mount -t cgroup cgroup /cgroup
```

与 RT Group Scheduling 相关的项目有下面两个。可以对每个分组分别设置 RT throttling 的单位时间与 CPU 分配时间。

```
cpu.rt_period_us
cpu.rt_runtime_us
```

小结

在需要实时性的领域，必须向进程赋予实时调度策略，将延迟控制在一定数量以下。但是，实时进程因 bug 等发生不可控错误时，就可能出现系统自身无法应答的情况。

使用 RT Group Scheduling 功能，可以仅分配实时进程真正需要的 CPU 时间，从而防止系统进程发生不可控错误等。

参考文献

• Documentation/scheduler/sched-rt-group.txt

——Hiroshi Shimamoto

HACK #10　Fair Group Scheduling

本节介绍 Cgroup 之一、管理 CPU 资源的 Fair Group Scheduling。

Fair Group Scheduling

Fair Group Scheduling 是 Cgroup 的资源管理之一，用来控制 Linux 内核的进程调度程序进行的 CPU 时间分配。与其他 Cgroup 进行的资源管理一样，可以对每个特定进程组进行资源（CPU 分配时间）管理。使用这个功能，就可以在分组间对 CPU 分配时间进行调整。

另外，Fair Group Scheduling 使用的是 Linux 2.6.23 以后引入的 CFS（Completely Fair Scheduler）的 CPU 分配时间控制功能，因此在没有安装 CFS 的 Linux 2.6.23 之前版本的内核中不能使用。因此，本节介绍怎样通过 CFS 对非实时调度策略进程的 CPU 分配进行控制。

Fair Group Scheduling 的使用方法

下面通过实例来讲解 Fair Group Scheduling 的使用方法。

使用前，需要挂载和安装 Cgroup 文件系统。由于使用了 Cgroup 进行资源控制，因此挂载时需要启用 CPU 资源控制。

```
# mount -t cgroup -o cpu cgroup /cgroup
```

在本示例中将创建两个控制 CPU 资源的分组，分别为 GroupA、GroupB。

```
# mkdir /cgroup/GroupA
# mkdir /cgroup/GroupB
```

为 GroupA、GroupB 这两个分组分配进程，从而在各分组间公平分享 CPU 时间。也就是说，GroupA 和 GroupB 的 CPU 使用率都是 50%。

然后，确认一下实际的 CPU 时间分配是否公平。打开一个新的终端，将 shell 进程分配给 GroupA。

```
# echo $$>  /cgroup/GroupA/tasks
```

然后，在这个 shell 上形成死循环，使 CPU 利用率达到 100%。

```
# while :; do true; done
```

接着，向 GroupB 分配新的 shell 进程，在 shell 上形成死循环，使 GroupB 中的 CPU 使用率也达到 100%。

```
# echo $$>  /cgroup/GroupB/tasks
# while :; do true; done
```

在这个状态下使用 top 命令确认 CPU 使用率，可以发现各分组中 shell（bash）的 CPU 使用率基本都是 50%，GroupA 和 GroupB 分别使用一半的 CPU 资源。

```
top - 03:53:13 up 1 day, 19:07,  4 users,  load average: 1.35, 0.42, 0.14
Tasks: 115 total,   3 running, 112 sleeping,   0 stopped,   0 zombie
Cpu(s):100.0%us,   0.0%sy,   0.0%ni,   0.0%id,   0.0%wa,   0.0%hi,   0.0%si,
0.0%st
Mem:   1021532k total,   497896k used,   523636k free,    80220k buffers
Swap:  2064376k total,        0k used,  2064376k free,   204004k cached

   PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
  8333 root      20   0  105m 1856 1444 R 50.2  0.2   0:37.83 bash
  8342 root      20   0  105m 1820 1424 R 49.9  0.2   0:32.43 bash
```

然后，向其中一个分组添加进程，并确认分组间（GroupA 和 GroupB）的 CPU 资源分配为各 50%。

打开一个新的终端，向 GroupB 添加 shell 进程。同样，在这个 shell 上形成死循环，使 CPU 使用率达到 100%。

```
# echo $$> /cgroup/GroupB/tasks
# while :; do true; done
```

这时使用 top 命令将显示下列结果。

```
top - 03:54:07 up 1 day, 19:08,  4 users,  load average: 1.89, 0.71, 0.25
Tasks: 115 total,   4 running, 111 sleeping,   0 stopped,   0 zombie
Cpu(s):100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1021532k total,   497896k used,   523636k free,    80228k buffers
Swap:  2064376k total,        0k used,  2064376k free,   204008k cached

   PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
  8333 root      20   0  105m 1856 1444 R 49.9  0.2   1:04.86 bash
  8342 root      20   0  105m 1820 1424 R 24.9  0.2   0:57.64 bash
  8354 root      20   0  105m 1852 1448 R 24.9  0.2   0:01.82 bash
```

可以看出 GroupA 和 GroupB 都为 50%，并且 GroupB 中的两个 shell 进程也各占 25%。

再确认一下，是否即使再向 GroupB 添加其他进程，也不会对分配给 GroupA 的 CPU 时间产生影响。

同样，添加终端，将 shell 分配给 GroupB，并形成死循环。

```
# echo $$> /cgroup/GroupB/tasks
# while :; do true; done
```

从 top 命令的结果可以看出，向 GroupA 分配了 50%，剩下的 50% 被 GroupB 的 3 个进程均分。

```
top - 03:57:11 up 1 day, 19:11,  5 users,  load average: 3.22, 1.88, 0.79
Tasks: 116 total,   5 running, 111 sleeping,   0 stopped,   0 zombie
Cpu(s): 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1021532k total,   498648k used,   522884k free,    80260k buffers
Swap:  2064376k total,        0k used,  2064376k free,   204008k cached

   PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
  8333 root      20   0  105m 1856 1444 R 49.9  0.2   2:38.69 bash
  8342 root      20   0  105m 1824 1424 R 16.9  0.2   1:36.27 bash
  8354 root      20   0  105m 1856 1448 R 16.6  0.2   0:39.11 bash
  8368 root      20   0  105m 1824 1424 R 16.6  0.2   0:14.23 bash
```

以上就是在分组间平分 CPU 资源的方法。

cpu.shares

下面介绍 cpu.shares 特殊文件。在启用 CPU 资源控制的 Cgroup 文件系统中，为 Fair Group Scheduling 准备了 cpu.shares 文件。

```
# ls /cgroup/GroupA
cgroup.procs      cpu.rt_runtime_us  notify_on_release
cpu.rt_period_us  cpu.shares         tasks
```

在 cpu.shares 文件中，可以对进程调度程序所处理的进程组设置 CPU 时间分配的比重。通过修改这个值，就可以在分组间调整 CPU 时间的比例。默认值为 1024。

```
# cat /cgroup/GroupA/cpu.shares
1024
```

这里将 GroupB 的 cpu.shares 设置为 GroupA 的一半，即 512。

```
# echo 512> /cgroup/GroupB/cpu.shares
```

在 CPU 资源分配中，GroupA 的比重变为 1024，GroupB 的比重变为 512。因此，分配给 GroupA 的时间就是 GroupB 的 2 倍。使用 top 命令，确认分配给刚才启动的 shell 进程的 CPU 资源。

```
top - 04:07:40 up 1 day, 19:22,  5 users,  load average: 4.00, 3.73, 2.34
Tasks: 116 total,   5 running, 111 sleeping,   0 stopped,   0 zombie
Cpu(s):100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1021532k total,   481024k used,   540508k free,    80348k buffers
Swap:  2064376k total,        0k used,  2064376k free,   204012k cached

   PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
  8333 root      20   0  105m 1856 1444 R 66.5  0.2   7:54.18 bash
  8342 root      20   0  105m 1824 1424 R 11.3  0.2   3:20.07 bash
  8354 root      20   0  105m 1856 1448 R 11.0  0.2   2:22.90 bash
  8368 root      20   0  105m 1824 1424 R 11.0  0.2   1:58.02 bash
```

可以发现，GroupA 变成 66%，GroupB 变成 33%，并且在 GroupB 中，3 个 shell 进程分

别占用 11%。

小结

本节介绍了 Fair Group Scheduling。它能以进程组为单位控制 CPU 资源分配，将 CPU 时间平分给多个用户，使特定处理不会对其他处理造成一定程度上的影响。

——Hiroshi Shimamoto

HACK #11　cpuset

本节介绍控制物理 CPU 分配的 cpuset。

cpuset 是 Linux 控制组（Cgroup）之一，其功能是指定特定进程或线程所使用的 CPU 组。另外，除 CPU 以外，同样还能指定内存节点的分配。

以前的内核具有 CPU affinity 功能，该功能将线程分配给特定 CPU。现在的内核中虽然也有 affinity（`taskset` 命令），但推荐使用 cpuset。

用法

使用 cpuset 前，必须通过内核 config 启用 cpuset 功能。

```
CONFIG_CPUSETS=y
```

最近的发布版在标准中就已启用。cpuset 就是作为 Cgroup 提供的一个功能。因此，使用 cpuset 时，就需要挂载 Cgroup 文件系统。使用下列方法启用 cpuset 选项，挂载 Cgroup 后，就可以使用 cpuset（参考 Hack #7）。

```
# mount -o cpuset -t cgroup cgroup /cgroup
```

在这里创建一个新的 CPU 分配组 GroupA。与其他 Cgroup 同样在挂载的 Cgroup 下创建新目录 GroupA，作为分组 GroupA。

```
# mkdir /cgroup/GroupA
```

编辑新创建分组 GroupA 的 cpuset，修改 CPU 分配情况。这里以仅将 CPU0 分配给分组 GroupA 的情况为例进行说明。在分组 GroupA 下的特殊文件 `cpuset.cpus` 内写入要分配的 CPU 编号，使用下列命令，来控制分组的 cpuset。

```
# echo 0 > /cgroup/GroupA/cpuset.cpus
```

到这一步，就完成了仅使用 CPU0 作为 GroupA 的 CPU 分配的设置。

接下来，在这个分组 GroupA 中添加进程。这里将当前 shell 添加到 GroupA 中。使用下列命令，将 PID（$$ 表示 shell 本身的 PID）写入 GroupA 下的 `task` 文件。

```
# echo $$ > /cgroup/GroupA/task
```

此后由当前 shell 启动的进程全部在这个 GroupA 下，使用的 CPU 仅限于 0 号 CPU。

现在确认所使用的 CPU 数量是否受限，以及产生的效果如何。本节显示的是以 Fedora 12 为例的情况。

本示例中使用的 Fedora 12 内核如下。

```
# uname -a
Linux fedora12 2.6.31.12-174.2.22.fc12.x86_64 #1 SMP Fri Feb 19 18:55:03 UTC 2010
x86_64 x86_64 x86_64 GNU/Linux
```

如果使用 cpuset 改变所使用的 CPU 数量会怎么样？比较内核的编译时间。

首先准备好要进行比较的编译。为了避免磁盘性能的影响，首先创建内存文件系统 tmpfs，并在其中配置源文件。创建目录 /tmp/build，挂载 tmpfs，命令如下所示。

```
# mkdir /tmp/build
# mount -t tmpfs none /tmp/build
```

本次测量的是内核源代码每次在创建的 tmpfs 下解压缩 tarball 时，使用默认 config 所花费的内核编译时间。使用的一系列命令行如下。

```
# cd /tmp/build/
# tar jxf /ext4data/kernel/linux-2.6.33.tar.bz2
# cd linux-2.6.33/
# make defconfig
# time make -j 2
```

首先测量 Linux 2.6.33 的编译时间。

将 Cgroup 挂载到 /cgroup，创建分组 GroupA。

```
# mount -o cpuset -t cgroup cgroup /cgroup
# mkdir /cgroup/GroupA
```

接下来看一下向分组 GroupA 分配两个 CPU 时的结果。

```
# echo"0-1" > /cgroup/GroupA/cpuset.cpus
# echo 0 > /cgroup/GroupA/cpuset.mems    mems 默认为空，因此需要填入值
# echo $$ > /cgroup/GroupA/tasks
```

编译时间如下。

```
# time make -j 2
real    4m55.568s
user    2m42.066s
sys     5m4.575s
```

然后将 CPU 缩减到只有 0 号 CPU。

```
# echo 0 > /cgroup/GroupA/cpuset.cpus
# echo 0 > /cgroup/GroupA/cpuset.mems
# echo $$ > /cgroup/GroupA/tasks

# mount -t tmpfs none /tmp/build
# cd /tmp/build/
# tar jxf /ext4data/kernel/linux-2.6.33.tar.bz2
```

```
# cd linux-2.6.33/
# make defconfig
# time make -j 2
real    7m44.491s
user    2m47.319s
sys     4m56.737s
```

可以看到，CPU 数量变为 1，实际花费的时间（real）增加。

下面以对虚拟化（KVM）进程所使用的 CPU 进行限制的情况为例，看一下将分配给 KVM 进程的 CPU 固定，并确保主机操作系统能够一直使用 CPU 后，是否能够减少虚拟化的影响。

这个示例同样使用 Fedora 12。

首先使用 KVM，启动两个客户端操作系统。然后，在客户端操作系统中循环进行内核编译，加大 CPU 负载。

在解压缩 Linux 2.6.33 源代码的目录下，无限循环执行 make clean 和 make 命令，增加客户端操作系统的 CPU 负载。

```
hshimamoto@ubuntu:~/kernel/linux-2.6.33$ while :; do make clean; time make; done
hshimamoto@opensuse:~/kernel/linux-2.6.33> while :; do make clean; time make; done
```

在这种情况下计算主机操作系统上的内核编译时间。同前例一样，需要创建 tmpfs，消除磁盘性能的影响后再进行测量。计算结果如下。

```
# time make -j 2
real    8m20.468s
user    2m45.890s
sys     4m51.091s
```

然后，将 KVM 的 qemu-kvm 进程可以使用的 CPU 设置为只有 0 号 CPU。

```
# mount -o cpuset -t cgroup cgroup /cgroup
# mkdir /cgroup/kvm

# echo 0 > /cgroup/kvm/cpuset.meme  将 kvm 分组的 cpuset 设为只有 0
# echo 0 > /cgroup/kvm/cpuset.cpus
```

将启动中的 qemu-kvm 移动到 kvm 分组。

```
# ps x | grep qemu
 2495 pts/2    Sl+  238:37 qemu-kvm
 2628 pts/3    Sl+  255:33 qemu-kvm

# for i in '\ls /proc/2495/task/'; do echo $i > /cgroup/kvm/tasks; done
# for i in '\ls /proc/2628/task/'; do echo $i > /cgroup/kvm/tasks; done
```

另外，ksmd（参考 Hack #36）也使用 CPU，因此这里也将其加入 kvm 分组。

```
# ps ux | grep ksmd
root     35  2.2  0.0     0    0?     SN   Mar23 119:42[ksmd]

# echo 35 > /cgroup/kvm/tasks
```

现在，主机操作系统的内核编译时间就变成如下所示。

```
# time make -j 2

real    7m55.081s
user    2m43.303s
sys     5m12.039s
```

可以发现，内核编译所花费的实际时间减少了接近 30 秒。

这是因为虚拟化的 KVM 进程只在 CPU0 上运行，在主机操作系统上就可以使用 100% 的 CPU。

小结

本节介绍了使用 Linux 中的 Cgroup 的 cpuset。通过使用这个功能，就可以限制特定进程所使用的 CPU。从另一个角度来看，通过固定使用的 CPU，还可以提高缓存的利用效率和性能。

——Hiroshi Shimamoto

HACK #12　使用 Memory Cgroup 限制内存使用量

Memory Cgroup 是 Cgroup 的资源限制功能之一，可以控制特定进程可以使用的内存量。

Memory Cgroup

Memory Cgroup 是 Cgroup（参考 Hack #7）之一，用来控制进程所使用的内存（LRU 管理的缓存）数量。

其用法有很多种，例如，可以用来避免因一时处理较大文件或大量文件，而导致无用的页面缓存增大，内存资源紧张的情况。另外，还可以在多用户环境中限制各用户可以使用的内存量。

用法

Memory Cgroup 是 Cgroup 的一种，因此使用前必须挂载 cgroup 文件系统。启用 Memory Cgroup 时，可以为挂载命令指定 memory 选项，也可以不指定选项以启用 Cgroup 的所有功能。

本节使用下列方式挂载到 /cgroup。

```
# mount -t cgroup -o memory memcg /cgroup
```

挂载 cgroup 后，通过在 /cgroup 下创建新目录来创建新的分组。Memory Cgroup 可以通过对该目录下的文件设置参数，来控制内存使用量。

另外，能否通过 Cgroup 文件系统控制以及特殊文件的种类多少，会根据内核版本和内核 config 的不同有所差异。

Cgroup 文件系统中关于 Memory Cgroup 配置的主要特殊文件如表 2-3 所示。

表 2-3　关于 Memory Cgroup 的主要文件

文 件 名	说　明
memory.usage_in_bytes	显示当前内存（进程内存 + 页面缓存）的使用量
memory.memsw.usage_in_bytes	显示当前内存（进程内存 + 页面缓存）+ 交换区使用量
memory.limit_in_bytes	设置、显示内存（进程内存 + 页面缓存）使用量的限制值
memory.memsw.limit_in_bytes	设置、显示内存（进程内存 + 页面缓存）+ 交换区使用量的限制值
memory.falicnt	显示内存（进程内存 + 页面缓存）达到限制值的次数
memory.memsw.failcnt	显示内存（进程内存 + 页面缓存）+ 交换区到达限制值的次数
memory.max_usage_in_bytes	显示记录的内存（进程内存 + 页面缓存）使用量的最大值
memory.memsw.max_usage_in_bytes	显示记录的内存（进程内存 + 页面缓存）+ 交换区使用量的最大值
memory.stat	输出统计信息，详细内容后面叙述
memory.force_empty	强制释放分配给分组的内存
memory.use_hierarchy	设置、显示层次结构的使用
memory.swappiness	设置、显示针对分组的 swappiness（相当于 sysctl 的 vm.swappiness）

限制内存使用量

内存使用量可以使用 memory.limit_in_bytes 进行限制。这里创建一个名称为 GroupA 的分组，尝试将内存使用量限制为 10MB。可以通过如下命令行实现。

```
# mkdir /cgroup/GroupA
# echo 10M > /cgroup/GroupA/memory.limit_in_bytes
# echo $$ > /cgroup/GroupA/tasks
```

下面看一下 Memory Cgroup 限制内存使用量的效果。

1. 获取较大的文件

首先，看一下不对内存使用量进行限制时的结果。

```
# free
             total       used       free     shared    buffers     cached
Mem:       1021532     415728     605804          0      24260     141764
-/+ buffers/cache:     249704     771828
Swap:      2064376          0    2064376

# wget http://ftp.yz.yamagata-u.ac.jp/pub/linux/centos/5.6/isos/x86_64/CentOS-
5.6-x86_64-LiveCD.iso
--2011-04-19 00:35:27--  http://ftp.yz.yamagata-u.ac.jp/pub/linux/centos/5.6/
isos/x86_64/CentOS-5.6-x86_64-LiveCD.iso
Resolving ftp.yz.yamagata-u.ac.jp... 133.24.255.153, 133.24.255.161
Connecting to ftp.yz.yamagata-u.ac.jp|133.24.255.153|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 733669376 (700M) [application/x-iso9660-image]
Saving to: "CentOS-5.6-x86_64-LiveCD.iso"

100%[======================================>] 733,669,376  496K/s   in 22m 44s

2011-04-19 00:58:11 (525 KB/s) - "CentOS-5.6-x86_64-LiveCD.iso" saved
[733669376/733669376]

# free
             total       used       free     shared    buffers     cached
Mem:       1021532     957288      64244          0      12944     676844
-/+ buffers/cache:     267500     754032
Swap:      2064376          0    2064376
```

从 free 来看，原本有约 600MB 的空闲内存减少到约 60MB。而 cached 的值大幅增加，可以看出对 wget 命令获取的约 700MB 的文件进行缓存时使用了空闲内存。

下面看一下将内存使用量限制为 10MB 时的结果。

```
# free
             total       used       free     shared    buffers     cached
Mem:       1021532     419988     601544          0      23280     154960
-/+ buffers/cache:     241748     779784
Swap:      2064376          0    2064376

# wget http://ftp.yz.yamagata-u.ac.jp/pub/linux/centos/5.6/isos/x86_64/CentOS-
5.6-x86_64-LiveCD.iso
--2011-04-19 23:04:47--  http://ftp.yz.yamagata-u.ac.jp/pub/linux/centos/5.6/
isos/x86_64/CentOS-5.6-x86_64-LiveCD.iso
Resolving ftp.yz.yamagata-u.ac.jp... 133.24.255.153, 133.24.255.161
Connecting to ftp.yz.yamagata-u.ac.jp|133.24.255.153|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 733669376 (700M) [application/x-iso9660-image]
Saving to: "CentOS-5.6-x86_64-LiveCD.iso"

100%[======================================>] 733,669,376  579K/s   in 24m 4s

2011-04-19 23:28:51 (496 KB/s) - "CentOS-5.6-x86_64-LiveCD.iso" saved
```

```
[733669376/733669376]

# free
             total       used       free     shared    buffers     cached
Mem:       1021532     432732     588800          0      25744     164656
-/+ buffers/cache:     242332     779200
Swap:      2064376          0    2064376
```

从 free 命令中 cached 的差别可以看出内存使用量限制在约 10MB。

2. 备份处理

在通过 tar 命令创建数据库时也可以使用。以 Linux 内核源代码数据库为例进行说明。

不限制内存使用量时：

```
# free; tar cf linux.tar linux; free
             total       used       free     shared    buffers     cached
Mem:       1021532     286136     735396          0      20144     110392
-/+ buffers/cache:     155600     865932
Swap:      2064376          0    2064376
             total       used       free     shared    buffers     cached
Mem:       1021532     949472      72060          0      20644     718776
-/+ buffers/cache:     210052     811480
Swap:      2064376          0    2064376
```

内存使用量限制为 10MB 时：

```
# free; tar cf linux.tar linux; free
             total       used       free     shared    buffers     cached
Mem:       1021532     288760     732772          0      20204     110372
-/+ buffers/cache:     158184     863348
Swap:      2064376          0    2064376
             total       used       free     shared    buffers     cached
Mem:       1021532     340476     681056          0      21732     118752
-/+ buffers/cache:     199992     821540
Swap:      2064376          0    2064376
```

可以发现内存同样限制为 10MB。

层次结构

通过 Memory Cgroup 控制的分组可以采用层次结构。可以在 memory.use_hierarchy 中写入 1，启用分组的层次结构。

```
# echo 1 > /cgroup/memory.use_hierarchy
```

例如，通过执行下列命令[审校者注 1]，可以创建如图 2-1 所示的分组结构。

```
# mkdir /cgroup/A
# echo 100M > /cgroup/A/memory.limit_in_bytes
# mkdir /cgroup/A/{B1,B2}
# echo 70M > /cgroup/A/B1/memory.limit_in_bytes
# echo 30M > /cgroup/A/B2/memory.limit_in_bytes
# mkdir /cgroup/A/B1/{C11,C12}
```

审校者注 1：下面的第 1、3、6、9 行命令原书有误，应该加上“-p”参数。

```
# echo 40M > /cgroup/A/B1/C11/memory.limit_in_bytes
# echo 30M > /cgroup/A/B1/C12/memory.limit_in_bytes
# mkdir /cgroup/A/B2/{C21,C22}
# echo 20M > /cgroup/A/B2/C21/memory.limit_in_bytes
# echo 10M > /cgroup/A/B2/C22/memory.limit_in_bytes
```

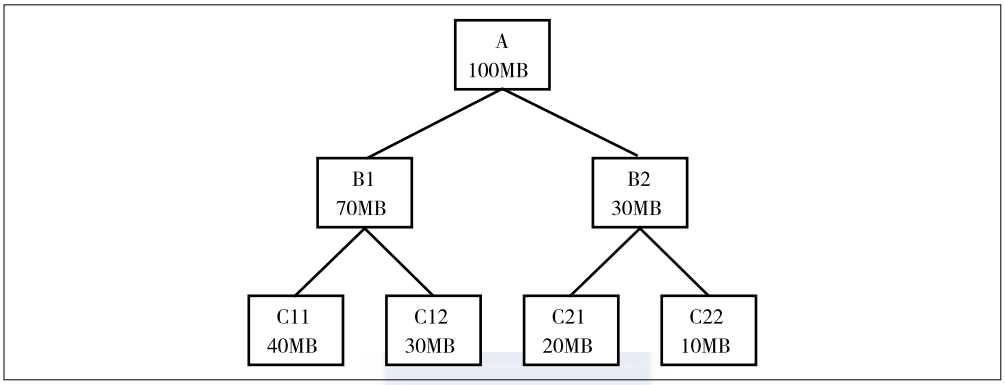

图 2-1　创建的分组结构

显示统计信息

关于各分组内存使用量的统计信息可以从 memory.stat 文件中读取（见表 2-4）。

表 2-4　内存使用量的统计信息

名　称	说　明
cache	页面缓存量（字节数）
rss	匿名页面与交换区缓存的内存量（字节数）
mapped_file	指向进程空间的文件映射所使用的内存量（字节数）
pgpgin	页面换入次数
pgpgout	页面换出次数
swap	交换区使用量（字节数）
inactive_anon	LRU 列表中无效的匿名页面（字节数）
active_anon	LRU 列表中有效的匿名页面（字节数）
inactive_file	LRU 列表中无效的文件缓存（字节数）
active_file	LRU 列表中有效的文件缓存（字节数）
unevictable	不能用 mlock 等回收的内存量（字节数）

下列内容在使用层次结构时有效，将显示层次结构中处于上层的分组所限制的值（见表 2-5）。

表 2-5　使用层次结构时的分组限制值

名　称	说　明
hierarchical_memory_limit	上层分组对内存（进程内存 + 页面缓存）的限制值
hierarchical_memsw_limit	上层分组对内存（进程内存 + 页面缓存）+ 交换区的限制值

表 2-6 所示为层次结构中分组的合计，在本分组下创建的所有分组的合计值。

表 2-6　层次结构中分组下的合计值

名　称	说　明
total_cache	本分组下所有页面缓存量（字节数）的合计值
total_rss	本分组下所有匿名页面与交换区缓存内存量（字节数）的合计值
total_mapped_files	本分组下所有指向进程空间的文件映射所使用的内存量（字节数）的合计值
total_pgpgin	本分组下所有页面换入次数的合计值
total_pgpgout	本分组下所有页面换出次数的合计值
total_swap	本分组下所有交换区使用量（字节数）的合计值
total_inactive_anon	本分组下所有 LRU 列表中无效的匿名页面（字节数）的合计值
total_active_anon	本分组下所有 LRU 列表中有效的匿名页面（字节数）的合计值
total_inactive_file	本分组下所有 LRU 列表中无效的文件缓存（字节数）的合计值
total_active_file	本分组下所有 LRU 列表中有效的文件缓存（字节数）的合计值
total_unevictable	本分组下所有不能用 molock 等回收的内存量（字节数）的合计值

小结

本节介绍了 Memory Cgroup。使用 Memory Cgroup 设置内存使用量的上限，就可以避免产生多余的页面缓存，减少对其他处理的影响。

参考文献

• Documentation/cgroup/memory.txt

——Hiroshi Shimamoto

HACK #13　使用 Block I/O 控制器设置 I/O 优先级

本节介绍使用 Block I/O 控制器的功能设置 I/O 优先级的方法。

Block I/O 控制器可以将任意进程分组，并对该分组设置 I/O 的优先级。这个功能是在 Linux 2.6.33 时添加到 Linux 内核中的。例如，在前台进行一般处理的同时，在后台磁盘备份处理的情况下，如果备份处理频繁地向磁盘进行 I/O 操作，前台的处理即使有 I/O 请求，也不能立刻进行 I/O 处理，结果导致前台处理的性能下降。

Block I/O 控制器在这种情况下就非常有效。创建 I/O 优先级较高的分组和较低的分组，并将前台处理的进程和后台处理的进程分别分配到这些分组中。这样可以使前台处理的 I/O 优先于后台处理，防止性能下降。

本节将介绍 Block I/O 控制器的使用方法。使用的 Linux 内核版本为 2.6.35。

使用 Block I/O 控制器的前提条件

Block I/O 控制器是 Cgroup 的子系统之一，是作为 I/O 调度程序之一的 CFQ 的一部分安装的。因此，使用 Block I/O 控制器时，必须使用启用了下列 config 选项编译的内核。

```
CONFIG_BLK_CGROUP
CONFIG_CFQ_GROUP_IOSCHED
```

确认 Cgroup 支持

首先确认运行中的内核是否支持 Cgroup 和 Cgroup 子系统 Block I/O 控制器。如果有 /proc/cgroups，运行中的内核就可以支持 Cgroup。/proc/cgroups 的内容如下。

```
$ cat /proc/cgroups
    subsys_name        hierarchy       num_cgroups     enabled
    blkio                              1               1               1
```

只要 subsys_name 列显示了 blkio，且 enabled 为 1 就没问题。如果看不到 blkio，就需要重新编译内核。如果 enabled 为 0，则启动时的内核命令行应当如下所示。

```
cgroup_disabled=blkio
```

这时需要修改 /boot/grub/grub.conf 等，将上述指定从内核命令行中删除，并重新启动。

确认 CFQ 支持

接下来确认是否能够将 CFQ 作为 I/O 调度程序使用。例如，想要查看块设备 sdb 中可以使用的 I/O 调度程序时，需要执行下列命令。

```
$ cat /sys/class/block/sdb/queue/scheduler
   noop deadline [cfq]
```

可以使用的 I/O 调度程序会显示出来，其中当前选择的调度程序已加上了方括号。如果

显示 cfq 则没问题；如果不显示就需要启用 CFQ，重新编译内核。

注意事项：设备种类不同，scheduler 文件的内容也不同。I/O 调度程序是对一般的块设备使用的，因此，例如在 loopback 设备 loop0 等中不会显示 cfq。请对 sda、sdb 等一般的块设备进行确认。

尝试使用 Block I/O 控制器

Block I/O 控制器的设置通过 cgroup 文件系统进行。关于 Cgroup 和 cgroup 文件系统的基本情况请参考 Hack #7。

首先挂载 cgroup 文件。将 blkio 作为挂载选项，指定将 Block I/O 控制器作为子系统使用。

```
# mount -t cgroup -o blkio cgroup/cgroup
```

然后，在 CFQ 中为想要控制 I/O 优先级的块设备设置 I/O 调度程序。这里使用的块设备是 sdb。

```
# echo cfq > /sys/class/block/sdb/queue/scheduler
$ cat /sys/class/block/sdb/queue/scheduler
    noop deadline [cfq]
```

这时，使用 Block I/O 控制器前的准备工作就完成了。为了方便讲解，在这里创建优先级较高的分组“high”和优先级较低的分组“low”，并分别向各分组分配 1 个进程来观察 I/O 的情况。首先创建分组。

```
# mkdir /cgroup/high
# mkdir /cgroup/low
```

对各分组设置 I/O 的优先级。设置优先级时，在 blkio.weight 中写入 100~1000 的“weight 值”。初始值为 500，值越大表示优先级越高。

```
# echo 1000 > /cgroup/high/blkio.weight
# echo 100 > /cgroup/low/blkio.weight
```

小贴士：仅根分组的 weight 初始值为 1000。

为了进行测试，制作一个简单的脚本。这个脚本将进程添加到所指定的两个分组 low 和 high，分别计算出读文件所需的时间。创建如下的“blkio_test.sh”脚本，并为其赋予执行权限。

```
#! /bin/sh

# blkio_test.sh
#     Test script for Block IO Controller
#     read /mnt/sdb/low.dat ($flow) as cgroup $cg_low
#     and read /mnt/sdb/high.dat ($fhigh) as cgroup $cg_high simultaneously.
#
#     $1: cgroup path for lower priority (--> $cg_low)
#     $2: cgroup path for higher priority (--> $cg_high)
```

```
function print_usage()
{
        echo "usage: $0 <cgroup_low> <cgroup_high>"
        exit 1
}


################################
# params and variables

flow=/mnt/sdb/low.dat
fhigh=/mnt/sdb/high.dat

# cgroups to which processes will be asigned

if [ $# != 2 ]; then
        print_usage
fi

cg_low=$1
cg_high=$2

for cg in $cg_low $cg_high; do
        if [ ! -d $cg ]; then
                echo "$cg does not exists"
                print_usage
        fi
done

# temporary files
out_low=$(mktemp)
out_high=$(mktemp)


################################
# sync, drop caches and read files

echo -n "sync and drop all caches..."
sync
echo 3 > /proc/sys/vm/drop_caches
echo "done"

echo -n "reading files..."
echo $$ > $cg_low/tasks
(time dd if=$flow of=/dev/null) > $out_low 2>&1 &

echo $$ > $cg_high/tasks
(time dd if=$fhigh of=/dev/null) > $out_high 2>&1 &

wait
echo "done"


################################
```

```
# print the results

echo "--------------------"
echo "dd in $cg_low:"
cat $out_low
echo "--------------------"
echo "dd in $cg_high:"
cat $out_high

rm -f $out_low $out_high
```

首先，在不分组的情况下进行测试。使用同属于根分组的两个进程，同时读入文件。由于脚本读入的是 /mnt/sdb/low.dat、/mnt/sdb/high.dat 文件，因此需要先创建两个大小适当的文件，再运行脚本。这里创建了约 400MB 的文件并执行相关代码。

```
# dd if=/dev/zero of=/mnt/sdb/low.dat bs=1M count=400
# dd if=/dev/zero of=/mnt/sdb/high.dat bs=1M count=400
# ./blkio_test.sh /cgroup /cgroup
--------------------
dd in /mnt/cgroups:
819200+0 records in
819200+0 records out
419430400 bytes (419 MB) copied, 14.0493 s, 29.9 MB/s

real    0m14.156s
user    0m0.261s
sys     0m1.183s
--------------------
dd in /mnt/cgroups:
819200+0 records in
819200+0 records out
419430400 bytes (419 MB) copied, 14.2007 s, 29.5 MB/s

real    0m14.292s
user    0m0.281s
sys     0m1.186s
```

两个进程同样使用约 14 秒的时间完成读入。接下来，在分组的情况下进行测试。将两个进程分别添加到 low、high 分组中，同时读入文件。

```
# ./blkio_test.sh /cgroup/low /cgroup/high
--------------------
dd in /cgroup/low:
819200+0 records in
819200+0 records out
419430400 bytes (419 MB) copied, 13.0829 s, 32.1 MB/s

real    0m13.388s
user    0m0.250s
sys     0m1.208s
--------------------
dd in /cgroup/high:
819200+0 records in
819200+0 records out
```

```
419430400 bytes (419 MB) copied, 7.43459 s, 56.4 MB/s

real    0m7.818s
user    0m0.256s
sys     0m1.209s
```

属于 /cgroup/low 的进程完成文件读入耗费约 13 秒，而属于 /cgroup/high 的进程完成读入耗费约 8 秒。与优先级较低，即 weight 设置值较小的分组相比，优先级较高，即属于 weight 设置值较大的分组（/cgroup/high）的进程可以优先执行 I/O 操作。

Block I/O 控制器提供的特殊文件

除了 blkio.weight 以外，Block I/O 控制器还提供了一些其他的特殊文件。文件列表如表 2-7 所示。只读属性的特殊文件是用来获取统计信息的文件，多数是根据各设备、I/O 的类型（read/write、sync/async）另起一行的。

表 2-7　Block I/O 控制器的设置用特殊文件

文　件　名	R/W	用　　途
blkio.weight	RW	设置分组 weight 值的文件。weight 值可以设置为 100~1000。weight 值越大，优先级越高
blkio.weight_device	RW	按照 < 设备主号码 >:< 设备副号码 ><weight 值 > 的格式写入，就可以为各设备设置 weight 值。将 <weight 值 > 设置为 0 时，该设备的设置被清除。这里没有进行设置的设备使用 blkio.weight 的 weight 值
blkio.io_merged	R	合并的 I/O 数
blkio.io_queued	R	当前保留的 I/O 数
blkio.io_service_bytes	R	I/O 请求总字节数
blkio.io_serviced	R	I/O 请求数
blkio.io_service_time	R	从 I/O 请求设备到完成所花费的总时间。单位为纳秒
blkio.io_wait_time	R	I/O 请求到达设备之前保留在等待队列的总时间。单位为纳秒
blkio.reset_stats	W	写入后，统计信息被清除
blkio.sectors	R	I/O 请求的总扇区数
blkio.time	R	目前为止分配给分组的时间片的长度。单位为纳秒

小贴士：I/O 的合并，是指将应用程序发出的多个 I/O 请求合并为 1 个。把相邻扇区的 I/O 整理到一起后，仅需一次 DMA 就可以完成 I/O，因此可以提高 I/O 处理的效率。

另外，在启用内核选项 CONFIG_DEBUG_BLK_CGROUP 进行编译的内核中，还有为用户提供调试用信息的文件，这里不作说明。

关于 Block I/O 控制器的 CFQ 设置用虚拟文件

/sys/block/<dev>/queue/iosched 中有 CFQ 的设置用虚拟文件，表 2-8 所示文件会对 Block I/O 控制器的运行产生影响。

表 2-8　关于 Block I/O 控制器的 sysfs 的 CFQ 设置文件

文　件　名	R/W	用　　途
group_isolation	RW	用来设置在 I/O 性能和分组间优先级控制二者中优先切换到哪一个。为 0 时，为了实现 I/O 性能最大化，会适度降低一些分组优先级的兼容性

限制事项

由于 Block I/O 控制器仍是比较新的功能，因此在使用上还有一些限制。接下来列出了到 Linux 内核版本 2.6.35 为止存在的限制事项。

不支持非同步 I/O

需要各分组进行优先级控制的，当前仅为同步 I/O，即初次读入和 Direct I/O 的读写。普通的写入是经过页面缓存的非同步 I/O，因此不属于优先级控制的对象，都被看做根分组发出的 I/O。

不支持分组层次化

把分组的层次限制为仅一层，因此无法创建从根分组开始有两层次以上的分组。用 cgroup 文件系统的目录层次可以显示如下。

```
/cgroup          # 根分组
/cgroup/gr1       # 第一层：可以创建
/cgroup/gr1/gr2    # 第二层：不可以创建
```

根分组与子分组作同等处理

根分组不作为其他子分组的上级分组，而是作为同等分组进行处理。当根分组内存在拥有实时 I/O 优先权的进程时，这个影响比较明显。根分组会被其他子分组抢占，因此即使是实时进程，也不能占用 I/O 带宽。

Block I/O 控制器的结构与注意事项

为了更准确地理解 Block I/O 控制器的运行，下面介绍其内部结构及其使用上的注意事项。

Block I/O 控制器对各分组的 I/O 请求分配时间片。仅许可各分组在这个时间片内执行 I/O 操作。分组的 weight 值越大，时间片的长度越长；weight 值越小，时间片的长度越短。通过这种方式来指定分组间的 I/O 操作的优先级。

这里需要注意的是，优先级不是通过 I/O 带宽（字节 / 秒，bps），而是通过时间片的长度来指定的。因此，两个分组在 I/O 完成所需时间差距极大的模式下执行 I/O 时，具体来说，就是一个依次读入，另一个随机读入的情况下，各分组的 I/O 带宽就会与 weight 值的设置迥然不同。

另外，还需要注意的是，Block I/O 控制器只有在针对设备的 I/O 发生竞争时，才会根据优先级对 I/O 进行控制。在 I/O 不发生竞争的情况下，即使是优先级较低的分组，Block I/O 控制器也不会禁止 I/O 的执行。也就是说，如果优先级较高的分组没有对某个设备发出 I/O 请求，那么即使是优先级较低的分组，也可以使用该设备的全部带宽。这里所说的设备是指实际的物理块设备。dm（device-mapper）等可以为应用程序提供逻辑性块设备，但是 Block I/O 控制器与逻辑块设备完全无关，只在向物理设备执行 I/O 时进行控制。当多个分组同时对同一逻辑设备发出 I/O 请求时，根据这些 I/O 请求所针对的物理设备不同，实际的 I/O 有可能竞争，有可能不竞争。这时，想要预测出哪个 I/O 会优先进行，就必须正确把握逻辑设备和物理设备的对应关系。

小结

Block I/O 控制器是正在开发的新功能。还存在前面所述的一些限制，因此更需要大家通过实际体验来发现存在问题或尚有不足的功能，反馈给开发者或者自己大胆地制作出来，才能够将其不断地完善。

参考文献

- 关于 Cgroup 请参考 Hack #7。
- Documentation/cgroups/blkio-controller.txt（内核源文档）

——Munehiro IKEDA

HACK #14　虚拟存储子系统的调整

本节介绍如何使用 /proc 进行虚拟存储子系统的调整。

虚拟空间存储方式

在 Linux 上向应用程序分配内存时，是通过以页面为单位的虚拟存储方式进行的。采用虚拟存储方式，在实际操作中具有不需要确保连续的物理内存（不用担心内存碎片）的优点。最近的处理器大部分都具备用于虚拟存储的处理器嵌入式 TLB（Translation

lookaside buffer，旁路转换缓冲区，或称为页表缓冲区）和处理不存在的页面访问的结构。除了部分嵌入式应用外，大多数 Linux 应用中都可以使用虚拟存储方式的内存管理。

使用虚拟存储方式的内存管理，具有下列特点。

- 程序使用的页面是在应用程序最初访问时由内核分配的。
- 如果分配的页面为程序文本、有初始值的数据（.data）区域或（被 mmap 的）数据文件区域，则在页面分配的同时从对应的文件读取数据，页面通过这些数据初始化。
- 如果对于不存在拥有初始值的数据的区域，则只进行页面分配处理。该页面作为匿名（anonymous）页面处理。

应用程序通过 `malloc()` 等分配可用的存储区时，不会立刻向该区域（空间）分配实际的页面。而是在必要时仅分配需要用到的页面。

多数应用程序一般都不会使用分配到的所有存储区。有时分配与最大数据量大小相等的缓冲区，有时也会需要为散列表（hash table）等分配没有实体的空间。这种情况下，使用虚拟存储方式延迟内存分配，就不需要向未使用的空间分配内存。

由于根据需要分配页面，因此应用程序启动时不能事先得知该应用程序最终使用的最大页面数，这是虚拟存储方式的缺点。进程在逻辑上的虚拟内存空间与实际分配给该空间（已使用）的实际内存空间之间不再有直接的关系，最大使用内存量会根据环境（处理的数据等）的变化而变化。

应用程序使用的内存量对系统稳定性有很大的影响，因此不能无限度地使用内存。

虚拟空间超额使用量的调整

只存在一个进程时，即使不知道该进程的最大内存使用量，也可以使用已有的进程单位的资源限制功能对内存使用量进行充分限制。实际安装的内存为 10GB 时，可以使用 `ulimiti` 命令将该进程的最大虚拟空间大小设置为 10GB。考虑到应用程序与虚拟空间大小相比要使用多少物理内存，也可以设置比 10GB 更大的虚拟空间大小（在 Linux 中未安装进程的物理内存驻留大小限制功能，因此无法使用。）

但是，考虑到多个进程互相争夺内存的情况，就需要限制整个系统的虚拟空间量。如上所述，即使无限度地向进程分配虚拟空间，只要不实际使用也就没问题。分配给进程的虚拟空间的大小与本质上实际安装的物理量无关。但是，从系统的稳定性来看，分配的虚拟空间大小应该保证达到物理内存的量。将没有物理内存保证的虚拟空间进行大量分配，并实际访问时，如果同时分配大量的物理页面，系统就会崩溃。

在 Linux 中有规定“允许超过物理内存量分配多少虚拟空间”的参数，通过下列两个 /proc 入口来进行控制。

- `/proc/sys/vm/overcommit_memory`
- `/proc/sys/vm/overcommit_ratio`

/proc/sys/vm/overcommit_memory 是控制虚拟空间分配的策略的参数，可以设置下列 3 种值。

- OVERCOMMIT_GUESS(0)
- OVERCOMMIT_ALWAYS(1)
- OVERCOMMIT_NEVER(2)

默认为 OVERCOMMIT_GUESS。

```
# cat/proc/sys/vm/overcommit_memory
0
```

设置为 OVERCOMMIT_NEVER 时，执行下列命令。

```
# echo 2 > /proc/sys/vm/overcommit_memory
```

可以在 /proc/sys/vm/overcommit_ratio 中指定允许过量使用的虚拟空间所占物理内存总量的百分比。默认为 50%。可以分配的最大虚拟空间为总物理内存量的 150%。

下面介绍 overcommit_memory 的不同取值对应的不同虚拟空间分配。

OVERCOMMIT_GUESS

overcommit_memory 的默认值为 OVERCOMMIT_GUESS。指定这个参数时，预测将空闲内存、页面缓存量、空闲交换区量、可回收 slab（长字节）量等回收的页面数，虚拟空间要求分配的量比这个数小时，分配成功。

（请注意，在多个进程同时要求大量的虚拟空间时是无法正确预测的。下面所述的 OVERCOMMIT_NEVER 中就没有这种问题。）

在 OVERCOMMIT_GUESS 的情况下，可分配的虚拟空间大小基本就是物理内存大小和交换区大小的合计值。物理内存为 2GB，交换区为 2GB，当前消耗 1GB 时，还可以分配约 3GB 的虚拟空间。

OVERCOMMIT_ALWAYS

在 OVERCOMMIT_ALWAYS 的情况下，虚拟空间分配总是成功。即使对于过大的虚拟空间要求，也会分配虚拟空间。可以在与实际安装的物理内存量完全无关的形态下使用虚拟空间，如前面所述的散列表等。

OVERCOMMIT_NEVER

在 OVERCOMMIT_NEVER 的情况下，对可分配虚拟空间量的管理更加严格。

首先，记录下整个系统内已分配的虚拟空间量。这个值严格由系统进行集中管理，在分配或释放虚拟空间时重新计算。这个值为 /proc/meminfo 的 Committed_AS。

对于虚拟空间大小的计算也比其他参数严格。例如，在 OVERCOMMIT_GUESS 的情况下，对 mmap 系统调用设置了 MAP_NORESERVE 的虚拟空间量不添加到 Committed_AS 中。但是，在 OVERCOMMIT_NEVER 的情况下会添加到 Committed_AS 中。指定了 MAP_

NORESERVE 的区域也作为可能分配物理内存的虚拟空间处理。

将“所有物理内存量 + 总交换区量”加上通过 /proc/sys/vm/overcommit_ratio 指定的比例得到的值，作为可分配的虚拟空间总量。这个值可以使用 /proc/meminfo 的 CommitLimit 查看。CommitLimit 的值仅在利用 OVERCOMMIT_NEVER 时有效。在 OVERCOMMIT_GUESS、OVERCOMMIT_ALWAYS 的情况下，这个项目没有意义。

要求分配虚拟空间时，如果“整个系统中已经分配（Committed_AS）”的虚拟空间量超过“可分配虚拟空间量（CommitLimit）”，则分配失败。

小结

本节介绍了虚拟空间的过量使用。根据系统的不同，需要对虚拟空间分配策略进行调整，管理要确保的内存量。

——Naohiro Ooiwa

关于进程的虚拟内存分配

在 Hack #13 中已经介绍过，用 malloc() 等预留内存后，就会分配虚拟内存。但使用 mmap() 时分配虚拟内存有一些限制条件。

例如，启动进程后，动态链接器就会进行动态库（library）的安装。由动态库的执行文件信息决定 mmap 的大小。接着，链接器会配置文本段和数据段，如果是 64 位操作系统，就会设置 alignment，当文本段和数据段较小时，它们之间有约 2MB 的空间。

以适当的访问权限对代码段和数据段执行 mmap 命令，使用 mprotect(PROT_NONE) 让此外未使用的段无法访问。

下面是在 CentOS 5.4 64 位操作系统下启动 apache 时 ps 命令的结果。

```
# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
...
root      4469  8.5  0.5 238244 11136 ?        Ss   10:27   0:00 /usr/sbin/httpd
apache    4471  0.0  0.3 238244  6516 ?        S    10:27   0:00 /usr/sbin/httpd
apache    4472  0.0  0.3 238244  6512 ?        S    10:27   0:00 /usr/sbin/httpd
apache    4473  0.0  0.3 238244  6512 ?        S    10:27   0:00 /usr/sbin/httpd
...
```

每一个进程确保了 200MB 以上的虚拟内存（VSZ）。使用 pmap 命令查看详细内容。pmap 命令显示的是进程的内存映射。

```
# pmap -d 4469
4469:   /usr/sbin/httpd
Address           Kbytes Mode  Offset           Device    Mapping
...
00002aff7a62e000      20 r-x-- 0000000000000000 008:00005 mod_cgi.so 文本段
```

```
00002aff7a633000    2048 ----- 0000000000005000 008:00005 mod_cgi.so 剩余区域
00002aff7a833000       8 rw--- 0000000000005000 008:00005 mod_cgi.so 数据段
00002aff7a835000       8 r-x-- 0000000000000000 008:00005 mod_version.so
00002aff7a837000    2044 ----- 0000000000002000 008:00005 mod_version.so
00002aff7aa36000       8 rw--- 0000000000001000 008:00005 mod_version.so
00002aff7aa38000     216 r-x-- 0000000000000000 008:00005 mod_perl.so
00002aff7aa6e000    2044 ----- 0000000000036000 008:00005 mod_perl.so
00002aff7ac6d000      16 rw--- 0000000000035000 008:00005 mod_perl.so
00002aff7ac71000    1200 r-x-- 0000000000000000 008:00005 libperl.so
00002aff7ad9d000    2044 ----- 000000000012c000 008:00005 libperl.so
00002aff7af9c000      36 rw--- 000000000012b000 008:00005 libperl.so
...
00002aff85b6e000    3844 rw--- 00002aff85b6e000 000:00000   [ anon ]
00007fff79a9c000      84 rw--- 00007ffffffea000 000:00000   [ stack ]
ffffffffff600000    8192 ----- 0000000000000000 000:00000   [ anon ]
mapped: 246436K    writeable/private: 6580K    shared: 692K
```

在 32 位的 CentOS 下的情况如下所示。

```
# ps aux
...
root      2715  6.6  7.6  23168  9488 ?        S    10:58   0:00 /usr/sbin/httpd
apache    2718  0.0  3.8  23168  4796 ?        S    10:58   0:00 /usr/sbin/httpd
apache    2719  0.0  3.8  23168  4796 ?        S    10:58   0:00 /usr/sbin/httpd
apache    2720  0.0  3.8  23168  4796 ?        S    10:58   0:00 /usr/sbin/httpd
...
```

每一个进程的虚拟内存（VSZ）为约 23MB。使用 pmap 命令查看内存映射，内容如下。

```
# pmap -d 2715
...
00508000      28 r-x-- 0000000000000000 008:00002 mod_proxy_ftp.so
0050f000       8 rwx-- 0000000000006000 008:00002 mod_proxy_ftp.so
00511000       4 r-x-- 0000000000000000 008:00002 mod_suexec.so
00512000       8 rwx-- 0000000000000000 008:00002 mod_suexec.so
00514000      16 r-x-- 0000000000000000 008:00002 mod_disk_cache.so
00518000       8 rwx-- 0000000000004000 008:00002 mod_disk_cache.so
0051a000       8 r-x-- 0000000000000000 008:00002 mod_file_cache.so
0051c000       8 rwx-- 0000000000001000 008:00002 mod_file_cache.so
0051e000      20 r-x-- 0000000000000000 008:00002 mod_cgi.so
00523000       8 rwx-- 0000000000004000 008:00002 mod_cgi.so
00525000       8 r-x-- 0000000000000000 008:00002 libutil-2.5.so
00527000       4 r-x-- 0000000000001000 008:00002 libutil-2.5.so
00528000       4 rwx-- 0000000000002000 008:00002 libutil-2.5.so
...
mapped: 23168K    writeable/private: 4916K    shared: 684K
```

在 32 位操作系统的情况下不存在访问权限为 -----（PROT_NONE）的区域。这是因为 32 位操作系统、64 位操作系统中链接器执行共享库映射的策略不同。将上述 pmap 命令的结果进行比较，看起来像是 64 位操作系统浪费了虚拟内存，但不用担心。

映射的区域中没有访问权限（PROT_NONE）的区域、写入权限（PROT_WRITE）或不是共享（MAP_SHARED）的区域不添加到 Committed_AS 中。因此在 64 位操作系统的情况下，即使消耗虚拟内存地址空间，也不会消耗实际的虚拟内存。

HACK #15　ramzswap

本节介绍将一部分内存作为交换设备使用的 ramzswap。

ramzswap 是将一部分内存空间作为交换设备使用的基于 RAM 的块设备。对要换出（swapout）的页面进行压缩后，不是写入磁盘，而是写入内存。可以使用的内存仅为完成压缩的部分。压缩处理使用的是 LZO[注1]。

ramzswap 是从 Linux 2.6.33 合并到 Staging 驱动程序的。Staging 驱动程序是指尚未达到某种程度的质量的试验性驱动程序。

通过使用 ramzswap，运转速度可以比换出到一般磁盘设备时更高。这是因为内存的 I/O 较快，且经过压缩后 I/O 变小。只有用于嵌入式系统的内存等的机器中，可以避免内存不足时由于内存回收处理导致性能极端下降，或抑制 OOM Killer 的运行。

ramzswap 的项目在如下环境中，即使减去压缩 / 解压缩的 CPU 系统开销，也可以提高性能。

• 上网本或瘦客户机（thin client）这种配备了内存容量小但 CPU 性能较高的的 PC。

• 在组装机器上，不想在外部闪存存储器（flash memory storage）中生成交换区时。

使用 ramzswap 时，可以使用已经整合到上游内核的，也可以从论坛中下载并使用。

整合到上游内核的 ramzswap 实际安装了论坛的部分成果。本节将针对上游内核和论坛版内核进行介绍。Linux 内核以 2.6.35 为例，论坛数据包以版本 0.6.2 为例。操作系统使用 Fedora 12。

使用论坛版 ramzswap

使用论坛版的数据包时，首先需要下载数据包进行编译。由于要对内核模块进行编译，因此必须事先安装 kernel-devel。

```
$ wget http://compcache.googlecode.com/files/compcache-0.6.2.tar.gz
$ tar zxvf compcache-0.6.2.tar.gz
$ cd compcache-0.6.2/
$ make
```

make 命令结束后，将生成内核模块 ramzswap.ko 和 ramzswap 设备的控制工具——rzscontrol 命令和 rzscontrol 命令的 manual 文件。

小贴士：ramswap 版本 0.6.2 的运行已经在 Linux 2.6.32 中确认。使用 RHEL6 编译时，需要在 ramzswap_drv.c 的最前面加上 #include <linux/slab.h>。

---

rzscontrol 命令创建的路径为 compcache-0.6.2/sub-projects/rzscontrol/

注 1：关于 LZO 请见参考文献。

rzscontrol，manual 文件创建的路径为 compcache-0.6.2/sub-projects/rzscontrol/man/rzscontrol.1。ramzswap 版本 0.6.2 不会通过 make 命令自动安装。可以直接使用这些文件。

ramzswap 的使用方法有两种。一种是在内存中创建虚拟交换区磁盘的 ramzswap disk，另一种是在使用内存的同时使用交换文件或交换块设备的 backing swap。一般系统与使用 ramzswap disk、backing swap 系统的内存和交换区的关系如图 2-2 所示。

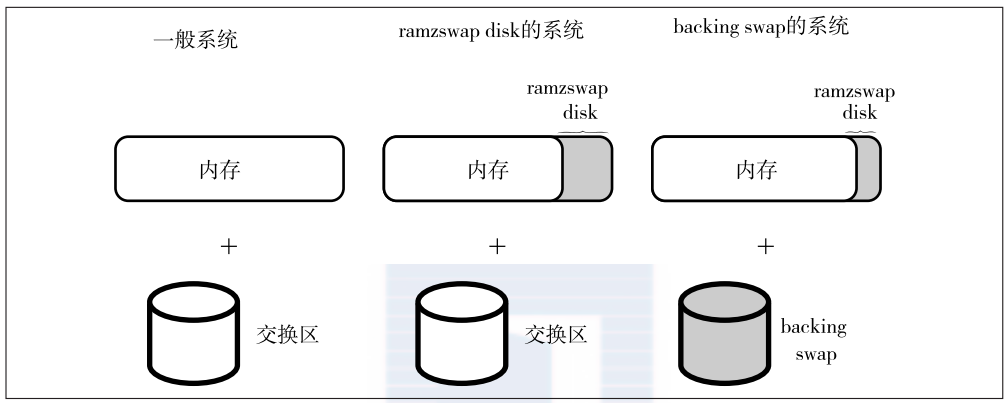

图 2-2　一般系统与使用 ramzswap disk、backing swap 的系统的内存和交换区的关系

首先介绍第一个 ramzswap disk。

ramzswap disk 的使用方法

使用 ramzswap disk，首先需要将用来压缩 / 解压缩数据的 LZO 模块安装到内核中。

```
# modprobe lzo_compress
#modprobe lzo_decompress
```

注意事项：在部分发布版（Fedora14 等）中，把 lzo_decompress 模块静态安装到内核中，有时会因为没有模块而导致 modprobe 失败。

```
# modprobe lzo_decompress
FATAL:Module lzo_decompress not found.
```

没有模块，也没有静态安装到内核时，对 ramzswap.ko 执行 insmod 命令，就会出现如下错误。

```
# insmod ramzswap.ko
insmod: error inserting 'ramzswap.ko': -1 Unknown symbol in module
```

自己构建内核时，请将 CONFIG_LZO_DECOMPRESS 设置为 y 或 m。把像 Fedora 14 这样 lzo_decompress 静态安装到内核的情况下，ramzswap.ko 的 insmod 会成功。

接下来安装 ramzswap 模块。这里将设置 num_devices=4，以生成 4 个设备文件。

```
# insmod ./ramzswap.ko num_devices=4
# ls /dev/ramzswap*
```

```
/dev/ramzswap0  /dev/ramzswap1  /dev/ramzswap2  /dev/ramzswap3
# lsmod
Module                  Size  Used by
ramzswap               21108  0
lzo_decompress          2768  1 ramzswap
lzo_compress            2480  1 ramzswap
......
```

将 ramzswap 模块安装到内核的同时，还可以设置各参数。下面是设置作为 ramzswap disk 使用的内存大小的例子。

```
# insmod ramzswap.ko num_devices=4 disksize_kb=20480
```

只有 /dev/ramzswap0 的初始化（后面介绍 --init 选项）和 disksize_kb 参数设置是自动进行的。/dev/ramzswap1~3 的初始化和 disksize_kb 参数需要另行设置。

后面也可以在 rzscontol 命令中设置相同参数。但是 num_devices 只能在安装模块时进行设置。表 2-9 所示为可设置的参数列表。其内容将在后面详细说明。

表 2-9　ramzswap 的模块参数与 rzscontrol 命令的选项

上游内核中含有的 ramzswap 模块参数	论坛版的 ramzswap 模块参数	ramzswap 命令选项
num_devices	num_devices	无
	disksize_kb	-d、--disksize_kb
	memlimit_kb	-m、--memlimit_kb
	backing_swap	-b、--backing_swap

安装 ramzswap 模块后，为了作为交换区使用对 ramzswap 设备进行初始化，并启用交换功能。

```
# sub-projects/rzscontrol/rzscontrol /dev/ramzswap0 - -init
# swapon /dev/ramzswap0
```

这时也可以使用 -p 选项指定优先级，与已有的交换设备同时使用。这个值较大表示优先级较高，因此应当指定比一般的交换设备更大的值。

```
# swapon -p 100 /dev/ramzswap0
# swapon -s
Filename                                Type            Size        Used    Priority
/dev/ramzswap0                          partition       511992      0       100
/dev/sda2                               partition       2047992     0       -1
```

这时，ramzswap 设备的交换功能就已启用。使用一定数量的内存后，就会发生换出。

可以使用 rzscontrol 命令的 --stats 选项来确认 ramzswap 的统计信息和状态。

```
# sub-projects/rzscontrol/rzscontrol /dev/ramzswap --stats
```

表 2-10 为此时的输出结果和各项目的说明。

表 2-10　执行 ramzswap disk 命令时 rzscontrol --stats 的输出结果

输　出	说　明
DiskSize: 255 852KB	ramzswap disk 的大小。默认设置为物理内存的 25%
NumReads: 91 576	从 ramzswap 设备读出的页面总数（解压缩页面的次数）
NumWrites: 224 893	写入 ramzswap 设备的页面总数（压缩页面的次数）
FailedReads: 0	因解压缩失败等导致 swap 区域的数据读取失败的页面数
FailedWrites: 0	因压缩失败等导致无法写入 swap 区域的页面数
InvalidIO: 0	因某些原因（内存不足等）导致无法处理的 I/O 请求数。这些 I/O 请求会撤销
NotifyFree: 220 016	释放的页面总数
ZeroPages: 163	在换出的页面中，数据全部为 0（零页面）的页面数。为零页面时仅记录其为零页面，压缩处理或实际的 I/O 会省略
GoodCompress: 96%	成功将大小压缩到页面大小 50% 以下的比例
NoCompress: 0%	未能将页面压缩到 75% 以下的比例。此时，压缩会放弃，换出压缩前的原始页面
PagesStored: 4871	正在换出的页面数。包括未能压缩（已计入 NoCompress）的页面。不包括零页面
PagesUsed: 1255	ramzswap 实际使用的页面数。这是 ramzswap 保留的内存量
OrigDataSize: 19 484KB	有换出要求的数据的合计大小。 PagesStored× 页面大小（一般为 4096 字节）。 该值对应压缩前的大小
ComprDataSize: 4801KB	换出后数据的合计大小。PagesUsed× 页面大小（一般为 4096 字节）。该值对应压缩后的大小
MemUsedTotal: 5020KB	ramzswap 实际使用的总内存量

由于 ramzswap 不会立刻释放保留的内存，因此 OrigDataSize 和 free 命令的数值不一定一致。

rzscontrol 命令也可以对各个 ramzswap 设备进行设置。使用 --disksize_kb 选项可以设置 ramzswap 设备的大小（单位为千字节）。执行下列命令就可以设置 ramzswap 设备的容量。

```
# sub-projects/rzscontrol/rzscontrol /dev/ramzswap0 --init --disksize_kb=10240
```

要将 ramzswap 设备排除在交换对象之外，可以使用 swapoff 命令。

```
# swapoff /dev/ramzswap0
```

要关闭 ramzswap，需要在 swapoff 之后使用 rzscontrol 命令的 --reset 选项释放残留在 ramzswap 磁盘中的内存。最后将模块从内核中移除。

```
# sub-projects/rzscontrol/rzscontrol /dev/ramzswap0 --reset
# rmmod ramzswap
```

backing swap 的使用方法

ramzswap 还有另一种使用方法，就是将部分内存作为 ramzswap disk 使用，再将交换文件或交换块设备作为 backing swap 使用。

ramswap 为内存和磁盘的两层。如果内存稍有不足，则仅使用内存的 ramzswap disk 进行处理，但如果缺少更多内存，则页面的内容存放到 backing swap 中。

下面介绍 backing swap 的使用方法。

首先与 ramzswap disk 同样进行设置。

```
# modprobe lzo_compress
# modprobe lzo_decompress
# insmod ./ramzswap.ko num_devices=4
```

然后使用 rzscontrol 命令指定 backing swap。

```
# sub-projects/rzscontrol/rzscontrol /dev/ramzswap0 --init --backing_swap=/
dev/sda2 --memlimit_kb=10240
```

使用 --backing_swap 选项指定交换文件或交换块设备。这里指定的是块设备 /dev/sda2。--memlimit_kb 选项指定的是作为 ramzswap disk 使用的内存大小。使用内存的方式基本与 ramzswap disk 相同。没有指定时设置为所有内存大小的 15%。这里设置为 10240KB。

最后启用已生成设备的交换功能。

```
# swapon /dev/ramzswap0
```

内存使用量一旦增加，首先压缩的页面会写入 ramzswap disk 的区域中。这时未压缩到 50% 以下的页面则写入 backing swap 中，而非 ramzswap disk 中。另外，超过 --memlimit_kb 选项指定的内存大小时也会写入 backing swap 中。

表 2-11 所示为执行 rzscontrol --stats 的结果。说明中仅记载与 ramzswap disk 的不同之处。

表 2-11　执行 ramzswap --stats 的结果

输　出	说　明
BackingSwap: /dev/sda2	使用 --backing_swap 指定的文件
DiskSize: 2 048 000KB	backing swap 的大小
MemLimit: 10 240KB	使用 --memlimit_kb 指定的大小

（续）

输　出	说　明
NumReads: 21 542	
NumWrites: 986 033	
FailedReads: 0	
FailedWrites: 0	
InvalidIO: 0	
NotifyFree: 0	
ZeroPages: 644	
GoodCompress: 99%	
NoCompress: 0%	未能将页面压缩到 50% 以下的情况
PagesStored: 16 961	
PagesUsed: 2592	
OrigDataSize: 67 844KB	
ComprDataSize:10 236KB	换出后的数据的合计大小。PagesUsed× 页面大小（通常为 4096 字节）。为压缩后的大小。在这个示例中如果进行更多的写入，就会超过 MemLimit，超过的部分会转移到 BackingSwap 中
MemUsedTotal: 10 368KB	
BDevNumReads: 110 861	从 BackingSwap 读入的次数
BDevNumWrites: 944 417	写入 BackingSwap 的次数

使用上游内核的 ramzswap

要使用安装在上游内核的 ramzswap，需要首先启用内核 config（CONFIG_RAMZSWAP=y），编译内核。使用 make menuconfig 命令启用下列项目。

```
Device Drivers
-> Staging drivers
 -> Compressed in-memory swap device (ramzswap)
```

启动编译后的内核。

使用方法

使用方法与论坛版相同，但需要另外编译用于上游内核的 rzscontrol 命令。由于上游内核驱动程序内没有安装 backing swap，因此必须修改 rzscontrol 命令的代码。

上游内核中还没有安装 backing swap 和 memlimit，因此将这部分代码从这个补丁中删除。下面使用这个补丁来编译 rzscontrol 命令。

```
# wget http://compcache.googlecode.com/files/compcache-0.6.2.tar.gz
# tar zxvf compcache-0.6.2.tar.gz
# cd compcache-0.6.2/sub-projects/rzscontrol
# patch -p1 < ramzswap-for-2.6.35.patch
```

按照下列方式指定上游内核的 include 文件进行编译。

```
# gcc -g -Wall -D_GNU_SOURCE rzscontrol.c -o rzscontrol -I /linux-2.6.35/drivers/
staging/ramzswap/ -I../include
```

使用方法与论坛版相同。

小结

本节介绍了 ramzswap。是否能够通过压缩页面数据受益，是与内存数据的内容相关的。但是即使多少有一些压缩 / 解压缩的系统开销，也比内存耗尽好得多。这在没有交换区的无磁盘（diskless）组装机器中尤其有效。

参考文献

- compcache Compressed Caching for Linux
  http://code.google.com/p/compcache/
- LZO1X Compressor from MiniLZO
  http://www.oberhumer.com/opensource/lzo/
- Compcache: in-memory compressed swapping
  http://lwn.net/Articles/334649/

——Naohiro Ooiwa

HACK #16　OOM Killer 的运行与结构

本节介绍 OOM Killer 的运行与结构。

Linux 中的 Out Of Memory（OOM）Killer 功能作为确保内存的最终手段，可以在耗尽系统内存或交换区后，向进程发送信号，强制终止该进程。

这个功能即使在无法释放内存的情况下，也能够重复进行确保内存的处理过程，防止系统停滞。还可以找出过度消耗内存的进程。本节将介绍 2.6 内核的 OOM Killer。

确认运行、日志

进行系统验证或负载试验时，有时会出现正在运行中的进程终止或者 SSH 连接突然断开、尝试重新登录也无法连接的情况。

这时需要查看日志。有时会输出如下内核信息。

```
Pid: 4629, comm: stress Not tainted 2.6.26 #3
```

```
Call Trace:
 [<ffffffff80265a2c>] oom_kill_process+0x57/0x1dc
 [<ffffffff80238855>] __capable+0x9/0x1c
 [<ffffffff80265d39>] badness+0x16a/0x1a9
 [<ffffffff80265f59>] out_of_memory+0x1e1/0x24b
 [<ffffffff80268967>] __alloc_pages_internal+0x320/0x3c2
 [<ffffffff802726cb>] handle_mm_fault+0x225/0x708
 [<ffffffff8047514b>] do_page_fault+0x3b4/0x76f
 [<ffffffff80473259>] error_exit+0x0/0x51

Node 0 DMA per-cpu:
CPU    0: hi:    0, btch:   1 usd:   0
CPU    1: hi:    0, btch:   1 usd:   0
...
Active:250206 inactive:251609 dirty:0 writeback:0 unstable:0
 free:3397 slab:2889 mapped:1 pagetables:2544 bounce:0
Node 0 DMA free:8024kB min:20kB low:24kB high:28kB active:8kB inactive:180kB
present:7448kB pa
ges_scanned:308 all_unreclaimable? yes
lowmem_reserve[]: 0 2003 2003 2003
...
Node 0 DMA: 6*4kB 4*8kB 2*16kB 2*32kB 5*64kB 1*128kB 3*256kB 1*512kB 2*1024kB
2*2048kB 0*4096k
B = 8024kB
Node 0 DMA32: 1*4kB 13*8kB 1*16kB 6*32kB 2*64kB 2*128kB 1*256kB 1*512kB
0*1024kB 0*2048kB 1*40
96kB = 5564kB
29 total pagecache pages
Swap cache: add 1630129, delete 1630129, find 2279/2761
Free swap  = 0kB
Total swap = 2048248kB
Out of memory: kill process 2875 (sshd) score 94830592 or a child
Killed process 3082 (sshd)
```

最后出现了 Out of memory（内存不足）。这就表示 OOM Killer 已经运行。无法重新连接的情况就是因为 sshd 被 OOM Killer 终止。如果不重新启动 sshd 就无法登录。

OOM Killer 通过终止进程来确保空闲内存，接下来将介绍如何选定这个进程。

进程的选定方法

OOM Killer 在内存耗尽时，会查看所有进程，并分别为每个进程计算分数。将信号发送给分数最高的进程。

计算分数的方法

在 OOM Killer 计算分数时要考虑很多方面。首先要针对每个进程确认下列 1 ～ 9 个事项再计算分数。

1. 首先，计算分数时是以进程的虚拟内存大小为基准的。虚拟内存大小可以使用 ps 命

令的 VSZ 或 /proc/<PID>/status 的 VmSize[注2] 来确认。对于正在消耗虚拟内存的进程，其最初的得分较高。单位是将 1KB 作为 1 个得分。消耗 1GB 内存的进程，得分约为 1 000 000。

2. 如果进程正在执行 swapoff 系统调用，则得分设置为最大值（unsigned long 的最大值）。这是因为禁用 swap 的行为与消除内存不足是相反的，会立刻将其作为 OOM Killer 的对象进程。

3. 如果是母进程，则将所有子进程内存大小的一半作为分数。

4. 根据进程的 CPU 使用时间和进程启动时间调整得分。这是因为在这里认为越是长时间运行或从事越多工作的进程越重要，需保持得分较低。

首先，用得分除以 CPU 使用时间（以 10 秒为单位）的平方根。如果 CPU 使用时间为 90 秒，由于以 10 秒为单位，因此就是用得分除以 9 的平方根“3”。另外，根据进程启动开始的时间也可以调整得分。用得分除以启动时间（以 1000 秒为单位）的平方根的平方根。如果是持续运行 16 000 秒的进程，则用得分除以 16 的平方根“4”的平方根“2”。越是长时间运行的进程就越重要。

小贴士：虽然源代码的备注中写有以 10 秒为单位、以 1000 秒为单位，但是实际上在位运算中是以 8 和 1024 为单位来计算。

---

5. 对于通过 nice 命令等将优先级设置得较低的进程，要将得分翻倍。nice-n 中设置为 1~19 的命令的得分翻倍。

6. 特权进程普遍较为重要，因此将其得分设置为 1/4。

7. 通过 capset(3) 等设置了功能（capability）CAP_SYS_RAWIO[注3] 的进程，其得分为 1/4。将直接对硬件进行操作的进程判断为重要进程。

8. 关于 Cgroup，如果进程只允许与促使 OOM Killer 运行的进程所允许的内存节点完全不同的内存节点，则其得分为 1/8。

9. 最后通过 proc 文件系统 oom_adj 的值调整得分。

依据以上规则，为所有进程打分，向得分最高的进程发送信号 SIGKILL（到 Linux 2.6.10 为止，在设置了功能 CAP_SYS_RAWIO 的情况下，发送 SIGTERM，在没有设置的情况下，发送 SIGKILL）。

各进程的得分可以使用 /proc/<PID>/oom_score 来确认。

但是 init（PID 为 1 的）进程不能成为 OOM Killer 的对象。当成为对象的进程包含子进程时，先向其子进程发送信号。

注 2：有时 /proc/<PID>status 的 VmSize 与计算出的分值多少有些差异。
注 3：默认为已设置。

向成为对象的进程发送信号后，对于引用系统的全线程，即使线程组（TGID）不同，如果存在与对象进程共享相同内存空间的进程，则也向这些进程发送信号。

关于 OOM Killer 的 proc 文件系统

下面开始介绍与 OOM Killer 相关的 proc 文件系统。

/proc/<PID>/oom_adj

为 /proc/<PID>/oom_adj 设置值就可以调整得分。调整值的范围为 –16~15。正的值容易被 OOM Killer 选定。负值可能性较低。例如，当指定 3 时，得分就变为 2^3 倍；当指定 –5 时，得分就变为 $1/2^5$。

“–17”是一个特殊的值。如果设置为 –17，就会禁止 OOM Killer 发出的信号（从 Linux 2.6.12 开始支持设置 –17）。

在 OOM Killer 运行的情况下，为了实现远程登录而想要将 sshd 排除在对象外时，可以执行下列命令。

```
# cat /proc/'cat /var/run/sshd.pid'/oom_score
15
# echo -17 >  /proc/'cat /var/run/sshd.pid'/oom_adj
# tail /proc/'cat /var/run/sshd.pid'/oom_*
==> /proc/2278/oom_adj <==
-17
==> /proc/2278/oom_score <==
0                                   /* 得分变成 0*/
```

从 Linux 2.6.18 开始可以使用 /proc/<PID>/oom_adj。内容记载在 Documentation/filesystems/proc.txt 中。

/proc/sys/vm/panic_on_oom

将 /proc/sys/vm/panic_on_oom 设置为 1 时，在 OOM Killer 运行时可以不发送进程信号，而是使内核产生重大故障。

```
# echo 1 > /proc/sys/vm/panic_on_oom
```

/proc/sys/vm/oom_kill_allocating_task

从 Linux 2.6.24 开始 proc 文件系统就有 oom_kill_allocating_task。如果对此设置除 0 以外的值，则促使 OOM Killer 运行的进程自身将接收信号。此处省略对所有进程的得分计算过程。

```
# echo 1 > /proc/sys/vm/oom_kill_allocating_task
```

这样就不需要参照所有进程，但是也不会考虑进程的优先级和 root 权限等，只发送信号。

/proc/sys/vm/oom_dump_tasks

从 Linux 2.6.25 开始，将 oom_dump_tasks 设置为除 0 以外的值时，在 OOM Killer 运行时的输出中会增加进程的列表信息。

下面为设置示例。

```
# echo 1 > /proc/sys/vm/oom_dump_tasks
```

列表信息显示如下，可以使用 dmesg 或 syslog 来确认。

```
[ pid ]   uid  tgid total_vm      rss cpu oom_adj name
[    1]     0     1     2580        1   0       0 init
[  500]     0   500     3231        0   1     -17 udevd
[ 2736]     0  2736     1470        1   0       0 syslogd
[ 2741]     0  2741      944        0   0       0 klogd
[ 2765]    81  2765     5307        0   0       0 dbus-daemon
[ 2861]     0  2861      944        0   0       0 acpid
...
[ 3320]     0  3320   525842   241215   1       0 stress
```

/proc/<PID>/oom_score_adj

从 Linux 2.6.36 开始都安装了 /proc/<PID>/oom_score_adj，此后将替换为 /proc/<PID>/oom_adj。详细内容请参考 Documentation/feature-removal-schedules.txt。即使当前是对 /proc/<PID>/oom_adj 进行的设置，在内核内部进行变换后的值也是针对 /proc/<PID>/oom_score_adj 设置的。

/proc/<PID>/oom_score_adj 可以设置 –1000~1000 之间的值。设置为 –1000 时，该进程就被排除在 OOM Killer 强制终止的对象外。

在内核 2.6.36 以后的版本中写入 oom_adj，只会输出一次如下的信息。

```
# dmesg
.....
udevd (60): /proc/60/oom_adj is deprecated, please use /proc/60/oom_score_adj
instead.
.....
```

RHEL5 的特征

在 RHEL5 中运行 OOM Killer 时要比在上游内核中更加慎重。OOM Killer 会计算调用的次数，仅在一定时间段内超出调用一定次数的情况下运行。

1.OOM Killer 从上次调出到下一次调出之间超过 5 秒时，调用次数重新开始计算。这是为了避免仅因为产生突发性的内存负载就终止进程。

2. 在计数变成 0 后的 1 秒以内调出时，不计入调用的次数。

3.OOM Killer 的调用次数不足 10 次时，实际不会运行。OOM Killer 调用 10 次时才开始认为内存不足。

4. 最后 OOM Killer 运行不到 5 秒的话，OOM Killer 不会再次运行。因此运行频率最高也有 5 秒一次。这是为了防止不必要地连续终止多个进程。也有等待接收到 OOM Killer 发出信号的进程终止（释放内存）的意思。

5. OOM Killer 一旦运行，调用的次数就重新回到 0。

也就是说，只有在 OOM Killer 在 5 秒以内调出的状态连续出现 10 次以上时才会运行。

这些限制原本是到 Linux 2.6.10 为止都有的。因此在基于 Linux 2.6.9 的 RHEL4 中也需要实施这些限制。当前的上游内核中已经取消了这些限制。

RHEL4 的运行

查看 OOM Killer 在 RHEL4（Linux 2.6.9）中的运行情况。在下例中，是内存、交换区都为 2GB 的环境下，使用负载测试工具 stress 刻意消耗内存。

stress 是给内存、CPU、磁盘 I/O 施加负载的工具。既可以为其中一项增加负载，也可以同时为这三项中的几项增加负载。stress 在运行中如果接收到信号，就会输出信息并终止。

```
# wget -t0 -c http://weather.ou.edu/~apw/projects/stress/stress-1.0.0.tar.gz
# tar zxvf stress-1.0.0.tar.gz
# cd stress-1.0.0
# ./configure ; make ; make install
# stress --vm 2 --vm-bytes 2G --vm-keep     /* 两个进程分别消耗 2GB 内存 */
stress: info: [17327] dispatching hogs: 0 cpu, 0 io, 2 vm, 0 hdd
stress: FAIL: [17327] (416) <-- worker 17328 got signal 15       /* 接收 SIGTERM 信号 */
stress: WARN: [17327] (418) now reaping child worker processes
stress: FAIL: [17327] (452) failed run completed in 70s
```

此时的控制台画面显示如下。

```
oom-killer: gfp_mask=0xd0
Mem-info:
...
Swap cache: add 524452, delete 524200, find 60/102, race 0+0
Free swap:            0kB                      /* 交换区剩余为 0 */
524224 pages of RAM                            /* 1 页 4KB，因此内存大小为 2GB */
10227 reserved pages                           /* 在内核内部预约的内存 */
19212 pages shared
253 pages swap cached
Out of Memory: Killed process 17328 (stress).   /* 根据信号终止的进程 */
```

在上游内核中无法禁用 OOM Killer，而在 RHEL4 中则通过 /proc/sys/vm/oom-kill 可以禁用 OOM Killer。

```
# echo 0 > /proc/sys/vm/oom-kill
或者
# /sbin/sysctl -w vm.oom-kill=0
```

禁用后 OOM Killer 就不会发送信号，但是会输出如上内存信息。

RHEL5 的运行

在 RHEL5（Linux 2.6.18）中对 OOM Killer 的运行进行确认的方法与 RHEL4 中相同。

```
# stress --vm 2 --vm-bytes 2G --vm-keep
stress: info: [11779] dispatching hogs: 0 cpu, 0 io, 2 vm, 0 hdd
stress: FAIL: [11779] (416) <-- worker 11780 got signal 9         /* SIGKILL */
stress: WARN: [11779] (418) now reaping child worker processes
stress: FAIL: [11779] (452) failed run completed in 46s
```

此时的控制台画面如下所示。添加了运行 OOM Killer 时的回溯输出，便于调试。

```
Call Trace:
 [<ffffffff800bf551>] out_of_memory+0x8e/0x321
 [<ffffffff8000f08c>] __alloc_pages+0x22b/0x2b4
...
 [<ffffffff800087fd>] __handle_mm_fault+0x208/0xe04
 [<ffffffff80065a6a>] do_page_fault+0x4b8/0x81d
 [<ffffffff800894ad>] default_wake_function+0x0/0xe
 [<ffffffff80039dda>] tty_ldisc_deref+0x68/0x7b
 [<ffffffff8005cde9>] error_exit+0x0/0x84

Mem-info:
...
Swap cache: add 512503, delete 512504, find 90/129, race 0+0
Free swap  = 0kB
Total swap = 2048276kB
Free swap:            0kB
524224 pages of RAM
42102 reserved pages
78 pages shared
0 pages swap cached
Out of memory: Killed process 11780 (stress).
```

RHEL6 的运行

RHEL6.0 中 OOM Killer 计算得分的方式基本和 RHEL5 中没有不同。RHEL6 系不会如“RHEL5 的特征”中所述慎重地运行。其运行基本与上游内核相同。

小结

本节介绍了 OOM Killer 的结构和各种设置。当系统运行异常时确认 syslog 等，如果有 OOM Killer 的输出，就可以得知曾出现内存不足。

参考文献

• stress

http://weather.ou.edu/~apw/projects/stress/

——Naohiro Ooiwa

第 3 章

文件系统

本章将介绍 RHEL6 等用做标准文件系统的 ext4 的使用和调整方法，以及从 ext2/ext3 转换的方法。另外，还将介绍进行 I/O 基准测试（benchmark）的 fio 以及用户空间的文件系统 FUSE。

HACK #17　如何使用 ext4

本节介绍 ext4 的编写和挂载方法、开发版 ext4 的使用方法。

ext4 是 ext3 的后续文件系统，从 Linux 2.6.19 开始使用。现在主要的发布版中多数都是采用 ext4 作为标准文件系统。

除了间接参照块管理以外，ext4 还以扩展形式支持块的管理，使其能够处理更大的文件、文件系统。另外，还增加了确保多块（multiblock）[注1]、确保延迟块、提高 fsck 速度、碎片整理等新的功能。在 ext3 中，时间戳（time stamp）的单位为毫秒，而 ext4 中变成了纳秒，可管理的时间日期上限也从 2038 年为止扩展到 2514 年为止。时间戳的种类也在以往的 mtime、atime、ctime 基础上增加了保存文件生成时间的 crtime[注2]。

表 3-1 所示为 ext3 和 ext4 主要功能的差异。

表 3-1　ext3 和 ext4 的性能、功能比较

性能、功能	ext3	ext4
最大文件大小	约 2TB	约 16TB
最大文件系统大小	约 16TB	约 1EB
块管理方式	间接参照块方式	区间方式（标准）、间接参照块方式
日志（journaling）功能	○	○
日志校验和（journal checksum）	—	○
延迟分配	—	○

注 1：关于确保多块的内容请参考 HACK#19。
注 2：ext4 的索引节点大小为 256 字节的情况。

（续）

性能、功能	ext3	ext4
多块分配	—	○
持久预分配	—	○（在区间形式的情况下）
条带（stripe）方式	—	○
时间戳单位	毫秒	纳秒
在线碎片整理	—	○

ext4 的生成与挂载

下面介绍 ext4 的生成方法。生成文件系统时可以使用 e2fsprogs 中的 mke2fs 命令。mke2fs 有很多种选项。表 3-2 介绍其中的一部分。

表 3-2　mke2fs 的选项

选　项	说　明
b	指定块的大小
F	强制执行 mke2fs
I	指定索引节点大小（字节单位）
O	文件系统功能的启用 / 禁用
T	指定文件系统的种类
v	显示详细内容

选项的详细情况请参考 mke2fs 命令的操作指南。

生成 ext4 时，需要将文件系统的种类指定为 ext4，执行 mke2fs 命令。在该例子中使用的是 Fedora 14 的 e2fsprogs-1.41.12-5，在 /dev/sdb1 上生成 ext4。

```
# mke2fs -t ext4 /dev/sdb1 或mkfs -t ext4 /dev/sdb1
```

挂载 ext4 时可以执行 mount 命令。需要向变量指定设备和挂载点。在该例子中是挂载到 /mnt。

```
# mount -t ext4 /dev/sdb1 /mnt
```

关于 mount 选项

ext4 中增加了很多功能。这些功能多数都可以在生成文件系统时或挂载时选择启用 / 禁用。这里介绍可以在挂载中设置的一部分选项（见表 3-3）。

mount 选项的详细内容请参考 mount 命令的操作指南或内核文档（Documentation/filesystems/ext4.txt）。

表 3-3　ext4 的挂载选项

选　项	说　明	默　认	ext4 特有
data=writeback	将日志模式设置为 writeback	—	—
data=ordered	将日志模式设置为 ordered	○	—
data=journal	将日志模式设置为 journal	—	—
journal_checksum	为要写入日志的事务添加校验和	—	○
journal_async_commit	非同步地将记录写入日志	—	○
barrier=1	启用写入屏障（barrier）	○	—
barrier=0	禁用写入屏障	—	—
discard	向下级块设备通知块已释放	—	—
nodiscard	不向下级块设备通知块已释放	○	—
delalloc	写入时使用延迟分配	○	○
nodelalloc	写入时不使用延迟分配。在出现写入请求的当时确保块	—	○
auto_da_alloc	通过 rename 进行文件替换、通过 truncate 后的写入进行文件替换时，不使用延迟分配功能，而是在当时立刻确保块	○	○
noauto_da_alloc	rename 和 truncate 处理时也使用延迟分配	—	○

开发版 ext4 的获取方法

现在论坛也在对 ext4 进行积极开发。开发版的 ext4 包含新的功能和 bug 的修改等。利用开发版时，需要从 ext4 的维护人员所管理的 Git 树中获取。这里将介绍获取开发版内核、命令的方法。获取时可以使用 git 命令。

ext4 patch queue 的获取

正在开发的 ext4 的补丁包括在 ext4 patch queue 中。可以使用下列方法来获取。

```
# git clone http://repo.or.cz/r/ext4-patch-queue.git
```

获取成功后，就会生成 ext4-patch-queue 目录。其中就有适用于 ext4 的补丁。对应的内核版本、适用的补丁的顺序记载在 series 文件中。

```
# cat ext4-patch-queue/series

# BASE v3.0-rc1
#
correct-comments-for-ext4_free_blocks
```

```
fix-max-file-size
use-FIEMAP_EXTENT_LAST-flag-for-last-extent
fixed-tracepoints-cleanup

# potential problems?
fix-oops-in-jbd2_journal_remove_journal_head

##########################################
# unstable patches
##########################################################
stable-boundary
...
```

将这些补丁适用于内核的源代码后，生成的内核就具有 ext4 的最新功能。

开发版的 e2fsprogs 的获取

e2fsprogs 的 Git 树可以执行下列命令来获取。

```
# git clone http://git.kernel.org/pub/scm/fs/ext2/e2fsprogs.git
```

这时获取的是最新的稳定版 e2fsprogs，因此需要切换到开发版的“next”分支。

```
# git checkout next
```

这样就切换到了开发版的 e2fsprogs。可以通过执行 configure，make 来使用各种命令。由于内核、命令都是开发版，因此在使用过程中可能会发现 bug。这时请向 linux-ext4@vger.kernel.org 报告。

小结

本节介绍了 ext4 的生成与挂载、获取开发版 ext4 的内核补丁和命令的方法。ext4 具有比 ext3 更多的功能，作为更加便捷的文件系统，能够吸引更多的用户。

参考文献

- Ext4(and Ext2/Ext3)Wiki
  https://ext4.wiki.kernel.org/index.php/Main_Page
- Mailing list ARChives
  http://marc.info/?l=linux-ext4&r=1&w=2

——Akira Fujita

HACK #18　向 ext4 转换

ext4 可以与 ext2/ext3 在后台进行互换。这里将介绍从 ext2/ext3 转换的方法以及转换时的注意事项。

转换

有两种方法可以将 ext2/ext3 的磁盘映像作为 ext4 来使用。

1. 直接作为 ext4 挂接

执行下列命令，就可以将 ext2/ext3 的磁盘映像 /dev/sdb1 作为 ext4 挂载到 /mnt。

```
# mount -t ext4 /dev/DEV MOUNTPOINT
```

通过上述方法，ext4 的多块分配、延迟分配等功能也可以使用，因此性能比 ext2/ext3 更高。但是，如果只是将 ext2/ext3 的磁盘映像直接作为 ext4 挂载，ext4 的很多功能仍然是无效的。例如，由于用间接块管理文件块，因此最大文件大小、最大文件系统大小都与作为 ext2/ext3 使用时相同。

2. 启用并挂载 ext4 的功能标志

要启用 / 禁用功能标志，可以使用 e2fsprogs 工具包的 tune2fs 命令或 debugfs 命令。在下例中使用 tune2fs 命令，启用 extent 标志。

```
# tune2fs -0 extent /dev/sdb1
```

当前设置的功能标志可以同样使用 e2fsprogs 工具包的 debugs 命令或 dumpe2fs 命令。在下例中使用的是 debugfs 命令，输出结果的“Filesystem features”中有“extent”，可以确认 extent 的功能标志已经设置。

```
# debugfs -R stats  /dev/sdb1
debugfs 1.41.12 (17-May-2010)
Filesystem volume name:   <none>
Last mounted on:          <not available>
Filesystem UUID:          03487be4-ed6e-4621-a3e6-326d443305f7
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode dir_index filetype
extent sparse_super large_file
...
```

tune2fs 命令中可以设置多个功能标志，但不能设置 flex_bg 等。flex_bg 是将块组虚拟整合，将元数据所在位置局部化，从而提高文件系统检测（fsck）速度。这个功能与块组的布局有关，块组的布局是在文件系统生成时由 mke2fs 命令决定的，因此不能通过文件系统生成后的 tune2fs 命令等来更改。

如果使用 tune2fs 命令取消已设置的功能标志，就可以从 ext4 磁盘映像返回 ext2/ext3 磁盘映像。但是，extent 标志一旦设置，就不能使用 tune2fs 命令来禁用，因此一旦设置这个标志，就无法返回 ext2/ext3。ext4 上以 extent 形式生成的文件在 ext2/ext3 文件系统中是无法读写的，因此必须事先知晓。

通过允许未初始化的块组，设置提高 fsck 检测速度的 uninit_bg 标志时，需要执行下列 e2fsck 命令。这样就可以对未使用的块组作标记，提高此后的 fsck 速度。

```
# e2fsck -pD /dev/sdb1
```

在本书写作时，ext 文件系统之间的互换性中仍然存在一些 bug。如果磁盘有空间，个人推荐不要将 ext2/ext3 的磁盘映像转换为 ext4，而是直接创建为 ext4。

关于功能标志

我们已经了解 ext4 的各种功能是由功能标志来管理的，那么文件系统生成时设置的功能标志应当如何决定呢？ /etc/mke2fs.conf 是管理 mke2fs 命令的标准设置的文件，除了功能标志之外，还可以设置块的大小或索引节点大小等的默认值。下列为 Fedora 14 的 /etc/mke2fs.conf 的内容。

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

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

在没有 /etc/mke2fs.conf 的情形或者执行 mke2fs 命令时指定的不是 -T 而是 -t 的情形下，mke2fs 命令将根据指定的文件系统种类和使用的磁盘大小来设置参数。

功能标志中有一些是 ext2/ext3/ext4 中共通的功能标志，也有一些是 ext4 特有的。表 3-4、表 3-5 所示为与 ext4 相关的主要功能标志列表。

表 3-4　ext2/ext3/ext4 共通的主要功能标志列表

标　志　名	说　　明
sparse_super	不在所有块组内生成备份用的超级块。空的块作为数据块使用
filetype	文件格式信息保存在目录中
resize_inode	为了使块组描述表可以扩展而预分配空间
dir_index	使用散列值的 B 树，提高目录检索速度
ext_attr	使文件的扩展属性可以使用
large_file	使其可以处理 2GB 以上的文件

表 3-5　ext4 标准功能标准列

标　志　名	说　　明	ext4 特有
has_journal	支持日志	
extent	使其能够处理 extent 形式的文件	○
huge_file	使其能够处理 2TB 以上的文件	
flex_bg	将数个块组的元数据分别集中在一起布局	○
uninit_bg	具有提高允许未初始化块组的文件系统检测（fsck）速度	○
dir_nlink	可以在 1 个目录下生成 65 000 个以上的目录	○
extra_isize	扩大索引节点的大小[注3]	-
auto_64-bit_support[注4]	支持 64 位的块编号	○

小结

本节介绍了从 ext2/ext3 转换到 ext4 的方法以及 ext4 的功能标志。ext4 在继承 ext2/ext3 的文件系统结构的同时，还新增了很多功能，使性能得到提高。用户在使用时也不会明显感觉到与一直使用的 ext3 有什么变化，这也是 ext4 的魅力之一。

参考文献

• Ext4 (and Ext2/Ext3) Wiki
https://ext4.wiki.kernel.org/index.php/Main_Page
• Mailing list ARChives
http://marc.info/?l=linux-ext4&r=1&w=2

——Akira Fujita

HACK #19　ext4 的调整

本节介绍可以从用户空间执行的 ext4 调整。

ext4 在 sysfs 中有一些关于调整的特殊文件（见表 3-6）。使用这些特殊文件，就不用进行内核编译、重启，直接从用户空间确认、更改内核空间的设置参数。

表 3-6　sysfs 中的 ext4 文件

文　件　名	说　　明	默认值
delayed_allocation_blocks	等待延迟分配的块数	

注 3：ext4 为了支持纳秒或更长的时间戳，使用的是扩展的 inode 区域。
注 4：块数超过 32 位可处理的情形下自动设置。

（续）

文 件 名	说 明	默认值
mb_max_to_scan	分配多块时为找出最佳 extent 而搜索的最大 extent 数	200
mb_min_to_scan	分配多块时为找出最佳 extent 而搜索的最小 extent 数	10
inode_goal	下一个要分配的 inode 编号（调试用）	0（禁用）
inode_readahead_blks	先行读入缓冲器缓存（buffer cache）的 inode 表块 (table block) 数的最大值	32
mb_order2_req	对于大于该值（2 的幂次方）的块，要求使用 Buddy 检索	2
lifetime_write_kbytes	文件系统生成后写入的数据量（KB）	
mb_stats	指定收集 (1) 或不收集 (0) 多块分配的相关统计信息。统计信息在卸载时显示	0（禁用）
max_writeback_mb_bump	进行下一次 inode 处理前尝试写入磁盘的数据量的最大值（MB）	128
mb_stream_req	块数小于该值的文件群被集中写入到磁盘上相近的区域	16
mb_group_prealloc	未指定挂载选项的 stripe 参数时，以该值的倍数为单位确保块的分配	512
session_write_kbytes	挂载后写入文件系统的数据量（KB）	

/sys/fs/ext4/< 设备名 > 下有与文件系统相关的各种文件。表 3-6 所示为这些文件的说明和默认值的列表。

这里将具体介绍上述文件中最为有效的部分文件。

1. lifetime_write_kbytes 与 session_write_kbytes

lifetime_write_kbytes 虽然不是可调整的入口，但是在 SSD 上生成 ext4 的情况下有时非常有用。SSD 对写入次数有限制。通过磨损平衡（wear leveling）将写入分散化，从而延长了寿命，但掌握目前为止写入文件系统的数据量对于预测 SSD 寿命也是非常有用的信息。

lifetime_write_kbytes 以千字节为单位记录写入文件系统的数据量，session_write_kbytes 以千字节为单位记录文件系统挂载后写入的数据量。

下例中展现的就是 lifetime_write_kbytes 和 session_write_kbytes 的值。

即使在刚生成 ext4 后，lifetime_write_kbytes 也会显示已有数据写入。这表示执

行 mkfs 时写入的元数据量。而 sesson_write_kbytes 中记录的是挂载文件系统的处理中产生的数据被写入的信息。

```
# cat /sys/fs/ext4/sda5/lifetime_write_kbytes
10637

# cat /sys/fs/ext4/sda5/session_write_kbytes
1
```

在这个 ext4 上生成 100MB 的文件。可以看到值分别增加了约 100MB。由于元数据的写出也会计数，因此是约 100MB。

```
# dd if=/dev/urandom of=/mnt/mp1/file bs=1M count=100
100+0 records in
100+0 records out
104857600 bytes (105 MB) copied, 18.322 s, 5.7 MB/s

# cat /sys/fs/ext4/sda5/lifetime_write_kbytes
113099

# cat /sys/fs/ext4/sda5/session_write_kbytes
102463
```

再次挂载文件系统后，lifetime_write_kbytes 的值会有一些增加。这是因为文件系统的挂载处理中有更新的数据（例如，内存上的超级块信息等）写入磁盘。而 session_write_kbyte 重置为 0。

```
# umount /mnt/mp1
# mount -t ext4 /dev/sda5 /mnt/mp1

# cat /sys/fs/ext4/sda5/lifetime_write_kbytes
113105
# cat /sys/fs/ext4/sda5/session_write_kbytes
1
```

2. mb_stream_req

ext4 中安装了多块分配处理功能，可以降低块分配处理中的 CPU 成本。可以将多个块分配到物理上连续的区域，提高 I/O 的效率。

另外，ext4 可以使用 fallocate 系统调用进行块的持久预分配，而多块分配处理中安装的则是内核内部使用的 group 预分配（下称 group PA）和 inode 预分配（下称 inode PA）。

group PA 由各 CPU 分别管理，用来将对象文件块（满足某条件的块）排列到物理上较近的区域，inode PA 用来将某一个文件的块排列到物理上连续的区域。

这些预分配算法的切换是如何进行的呢？ ext4.h 里的下列定义就是其阈值。

```
#define MB_DEFAULT_STREAM_THRESHOLD    16    /* 64k */
```

由于是以块数为单位，因此块的大小 4KB 时阈值为 64KB，块的大小为 1KB 时阈值为 16KB。

默认设置表示对于 16 块以下的块分配处理使用 group PA，对于 16 块以上则使用 inode PA。这个值可以使用 mb_stream_req 来更改。

默认值设置为 16 块，是因为设置文件等多数文件都在 16 块（块的大小 4KB 时阈值为 64KB）以内。因此，这些文件通过 group PA 实现物理上的局部化。在系统启动时等数据读入处理操作中，通过将文件数据局部化，可以缩短磁盘的寻道时间（seek time）。

实际操作 mb_stream_req，来验证一下 group PA 和 inode PA 的运行和效果。为了便于测定预分配的效果，这次操作中将延迟分配禁用。

```
# mount -t ext4 -o nodelalloc /dev/sdb5 /mnt/mp1

# cat /proc/mounts  | grep sdb5
/dev/sdb5 /mnt/mp1 ext4
rw,relatime,user_xattr,barrier=1,nodelalloc,data=ordered 0 0
```

为了使用 group PA，将 mb_stream_req 设置为 64，并向文件分配小于这个值的 32 块。

```
# echo 64 > /sys/fs/ext4/sdb5/mb_stream_req

# for i in 1 2 3; do dd if=/dev/urandom of=/mnt/mp1/file$i bs=4K count=32 >
/dev/null 2>&1; done

# ls -l /mnt/mp1/file*
-rw-r--r-- 1 root root 135168 2011-05-17 01:10 /mnt/mp1/file1
-rw-r--r-- 1 root root 135168 2011-05-17 01:10 /mnt/mp1/file2
-rw-r--r-- 1 root root 135168 2011-05-17 01:10 /mnt/mp1/file3
```

可以使用 e2fsprogsa 工具包的 fileflag 命令来确认分配给磁盘的数据块。根据 physical 和 length 显示的值，可以看出通过 group PA，把各个文件的数据块排列在物理上相近的位置。

```
# for i in 1 2 3; do filefrag -v /mnt/mp1/file$i; done

Filesystem type is: ef53
File size of /mnt/mp1/file1 is 131072 (32 blocks, blocksize 4096)
 ext logical physical expected length flags
   0       0    33280              32 eof
/mnt/mp1/file1: 1 extent found
Filesystem type is: ef53
File size of /mnt/mp1/file2 is 131072 (32 blocks, blocksize 4096)
 ext logical physical expected length flags
   0       0    33792              32 eof
/mnt/mp1/file2: 1 extent found
Filesystem type is: ef53
File size of /mnt/mp1/file3 is 131072 (32 blocks, blocksize 4096)
 ext logical physical expected length flags
   0       0    33312              32 eof
```

然后，为了确认 inode PA 的运行，再将 mb_stream_req 设置为 16，并向文件分配大于这个值的 32 块。

```
# rm -rf /mnt/mp1/file*
```

```
# umount /mnt/mp1

# mount -t ext4 -o nodelalloc /dev/sdb5 /mnt/mp1

# echo 16 > /sys/fs/ext4/sdb5/mb_stream_req

# for i in 1 2 3; do dd if=/dev/urandom of=/mnt/mp1/file$i bs=4K count=32 >
/dev/null 2>&1; done

# for i in 1 2 3; do filefrag -v /mnt/mp1/file$i; done

Filesystem type is: ef53
File size of /mnt/mp1/file1 is 131072 (32 blocks, blocksize 4096)
 ext logical physical expected length flags
   0       0    33280              16
   1      16    33072    33295     16 eof
/mnt/mp1/file1: 2 extents found
Filesystem type is: ef53
File size of /mnt/mp1/file2 is 131072 (32 blocks, blocksize 4096)
 ext logical physical expected length flags
   0       0    33296              16
   1      16    33088    33311     16 eof
/mnt/mp1/file2: 2 extents found
Filesystem type is: ef53
File size of /mnt/mp1/file3 is 131072 (32 blocks, blocksize 4096)
 ext logical physical expected length flags
   0       0    33792              16
   1      16    33104    33807     16 eof
/mnt/mp1/file3: 2 extents found
```

在多于 mb_stream_request 的块分配处理中，将使用 inode PA。inode PA 只在引用对象文件期间保留有效的预分配块。从而在按顺序向对象文件写入时，把块分配到物理上连续的区域，抑制碎片的产生。要注意的是，inode PA 在块分配中并不是每次都会生成，而是只在多块分配处理中获取的连续空闲块与要求的块数之间有差异时使用。例如，对于 16 块的分配要求，当多块分配处理检索到连续空闲块有 16 个时，就不会生成 inode PA。

在上述块分配中不生成 inode PA，file1 的第一个范围（extent）是从物理偏移量 33 280 分配 16 块，而从其后的物理偏移量 33 296 分配的是 file2 的第一个范围的块。这样就会产生碎片[注5]。

为了避免产生碎片，文件系统上就必须有足够的连续空闲块。使用 e2fsprogs 工具包中包含的 e2freefrag 命令来确认有多少连续的空闲块。在下例中，Extent Size Range 越大，在整体中所占的比例越大，可见这个文件系统上有充足的连续空闲块。

```
# e2freefrag /dev/sdb5

Device: /dev/sdb5
Blocksize: 4096 bytes
Total blocks: 1220703
```

注 5：本次为了便于测定而禁用了延迟分配，因此结果非常明显。ext4 默认是启用延迟分配的，因此在实际的块分配中能够尽量减少碎片的产生。

```
Free blocks: 1166377 (95.5%)

Min. free extent: 40 KB
Max. free extent: 917504 KB
Avg. free extent: 358884 KB
Num. free extent: 13

HISTOGRAM OF FREE EXTENT SIZES:
Extent Size Range :  Free extents   Free Blocks  Percent
   32K...    64K-  :             2            20    0.00%
   64M...   128M-  :             2         49102    4.21%
  128M...   256M-  :             5        326180   27.97%
  512M... 1024M-  :             4        791075   67.82%
```

产生的碎片可以使用 e4defrag 命令来消除。发布版（例如 Fedora 14）的 e2fsprogs 工具包中还未收入 e4defrag 命令。在使用时就需要获取由 Git 管理的 e2fsprogs 工具包。使用 git 命令获取 e2fsprogs 的方法请参考 HACK #17[注6]。

下面是执行 e4defrag 命令的示例。可以看出原来有 17 个的文件碎片已消除。

```
# e4defrag -v  /mnt/mp1/file1
ext4 defragmentation for /mnt/mp1/file1
[1/1]/mnt/mp3/file1:    100%  extents: 17 -> 1 [ OK ]
 Success:                      [1/1]
```

上文介绍了更改 mb_stream_request 的值时块的分配的差异。通过切换 group PA、inode PA，就可以在文件系统上对文件进行布局。有时还可以提高顺序读入中的 I/O 性能。

3. inode_readahead_blks

使用 inode_readahead_blks 可以控制进行预读的 inode 表的数量。inode 表是指磁盘上储存 inode 结构的区域，被分配给各块组。与下列 flex_bg 标志同时使用，就可以提高运行效率。

下例输出的是启用 / 禁用功能标志 flex_bg 时 inode 表的位置。

• 启用 flex_bg

```
# mke2fs -t ext4 /dev/sda4; dumpe2fs /dev/sda4 | egrep "Group|Inode table"
dumpe2fs 1.41.14 (22-Dec-2010)
Group 0: (Blocks 0-32767) [ITABLE_ZEROED]
  Primary superblock at 0, Group descriptors at 1-1
  Inode table at 332-833 (+332)
Group 1: (Blocks 32768-65535) [INODE_UNINIT, ITABLE_ZEROED]
  Backup superblock at 32768, Group descriptors at 32769-32769
  Inode table at 834-1335 (bg #0 + 834)
Group 2: (Blocks 65536-98303) [INODE_UNINIT, BLOCK_UNINIT, ITABLE_ZEROED]
  Inode table at 1336-1837 (bg #0 + 1336)
Group 3: (Blocks 98304-131071) [INODE_UNINIT, ITABLE_ZEROED]
  Backup superblock at 98304, Group descriptors at 98305-98305
  Inode table at 1838-2339 (bg #0 + 1838)
...
```

注 6：e2fsprogs 的 master 分支内包含 e4defrag 命令，因此不需要切换分支。

• 禁用 flex_bg

```
# mke2fs -t ext4 -O ^flex_bg /dev/sda4; dumpe2fs /dev/sda4 | egrep "Group|Inode table"
dumpe2fs 1.41.14 (22-Dec-2010)
Group 0: (Blocks 0-32767) [ITABLE_ZEROED]
  Primary superblock at 0, Group descriptors at 1-1
  Inode table at 302-803 (+302)
Group 1: (Blocks 32768-65535) [INODE_UNINIT, ITABLE_ZEROED]
  Backup superblock at 32768, Group descriptors at 32769-32769
  Inode table at 33070-33571 (+302)
Group 2: (Blocks 65536-98303) [INODE_UNINIT, BLOCK_UNINIT, ITABLE_ZEROED]
  Inode table at 65538-66039 (+2)
Group 3: (Blocks 98304-131071) [INODE_UNINIT, ITABLE_ZEROED]
  Backup superblock at 98304, Group descriptors at 98305-98305
  Inode table at 98606-99107 (+302)
...
```

启用 flex_bg 时，“Inode table at”显示的物理块编号连续，可以看出各块组的 inode 表排列在物理上连续的区域。

flex_bg 在标准设置中是将每 16 块组的元数据集中在一处。因此，各块组的 inode 表按照 0 ～ 15、16 ～ 31 的单位集中在一处。

因此不启用 flex_bg 时，有时即使为 inode_readahead_blks 设置较大的值，inode 表的区域也并不连续，因此没有什么意义。

在连续访问多个文件的处理中，可以将 inode_readahead_blks 设置较大的值，一次读入大量 inode 表，这样效率更高。

在下面这个极端的例子中，将 inode_readahead_blks 分别设置为 1 和 4096 时的内核源代码读入时间进行了比较。启用 ext4 的 flex_bg。

• 当 inode_readahead_blks 为 1 时

```
# time find /mnt/mp1/linux-2.6.39-rc1 -type f -exec cat > /dev/null {} \;
real    3m36.599s
user    0m5.092s
sys     0m43.510s
```

• 当 inode_readahead_blks 为 4096 时

```
# time find /mnt/mp1/linux-2.6.39-rc1 -type f -exec cat > /dev/null {} \;
real    3m23.613s
user    0m5.080s
sys     0m43.341s
```

执行时间的差异为约 6%。根据运用环境的不同也会有一些差异，在经常需要连续读取多个文件的环境下，就可以通过更改 inode_readahead_blks 来提高读入性能。

小结

本章介绍了使用 sysfs 的 ext4 调整。可以从用户空间对内核参数进行操作，非常方便。

如果在 ext4 的源代码中发现了其他可以从用户空间进行调整的项目，请尝试向邮件地址列表投稿。对于新的结构或功能会有积极的参考作用，这时如果有补丁的话一定会得到很多反馈。

参考文献

- Ext4 (and Ext2/Ext3)Wiki
  https://ext4.wiki.kernel.org/index.php/Main_Page
- Mailing list ARChives
  http://marc.info/?l=linux-ext4&r=1&w=2

——Akira Fujita

HACK #20　使用 fio 进行 I/O 的基准测试

本节介绍使用 fio 进行模拟各种情况的 I/O 基准测试的操作方法。

I/O 的基准测试中有无数需要考虑的因素。是 I/O 依次访问还是随机访问？是通过 `read/write` 的 I/O ？还是通过访问 `mmap` 的空间的 I/O ？是单一进程发出的 I/O ？还是多个进程同时发出的 I/O ？进程是受 I/O 限制还是受 CPU 限制？等等。

如果使用 fio，就不需要每次都根据不同情况来编写用于性能评估的程序，就可以模拟这些情况的 I/O。

安装 fio

Fedora、Ubuntu 等主流发布版中都备有 fio 的二进制文件包。请使用 `yum`、`apt` 等安装 fio 工具包。

这里按照 Fedora 13 中包含的 fio 版本 1.36 来进行说明。

想要使用最新版时，请先从下列网页下载 fio 的源代码，再进行安装。

程序页

http://freshmeat.net/projects/fio

Git 仓库

git://git.kernel.org/pub/scm/linux/kernel/git/axboe/fio.git

基本执行方法

指定记载了 I/O 操作模式的设置文件，然后启动 fio 命令，这就是 fio 最基本的执行方法。大多数的项目可以不使用配置文件而通过命令行选项来指定。首先举出两种分别使

用命令行选项的方法和配置文件的方法的简单例子，了解基本执行方法和命令行 / 设置文件的对应情况。

第一个示例执行的是下列操作。

- 进行 10MB 的顺序读入。
- 同时，其他进程随机进行 5MB 的写入。
- 使用 read(2) 和 write(2) 进行读写。
- 使用 Direct I/O。

在 fio 命令行进行这些设置时的操作如下。在当前目录下会生成读写用的文件，因此要在可以写入的目录下执行。

```
$ fio --name=global --direct=1 --name=read --rw=read --size=10m --name=write
--rw=randwrite --size=5m
```

与上述命令行选项具有相同意义的配置文件如下所示。

```
# cat fio_exam.fio

[global]
direct=1

[readjob]
size=10m
rw=read

[writejob]
size=5m
rw=randwrite
```

使用这个配置文件执行 fio 命令如下。

```
$ fio simple_read.fio
```

命令行选项与配置文件的各项之间的关系非常明显。这里简单介绍配置文件的各项的意义。

以 # 开头的是注释（comment）。内容直到行尾都被忽略。

方括号表示 [] 作业（job）定义选项的开始。方括号内是作业的名称。作业可以任意命名。但是 `global` 这个名称是内部使用的。作业 `global` 是用来定义所有作业共同的设置项。

fio 按照每个作业生成进程，生成的进程根据设置进行实际的 I/O 操作。因此，fio 的配置文件中除了 `global` 选项以外，必须要定义多个作业。

`global` 选项中设置了 `direct=1`。将 `direct` 设置为 1 时，使用的就 Direct I/O。未指定时，就等同于 `direct=0`（经由一般页面缓存进行 I/O）。

接下来定义名为 `readjob` 的作业。

readjob 作业中使用 size 设置 I/O 操作的大小（字节数）。后缀 m 表示兆字节。后缀可以使用 k、m、g、t、p。readjob 作业的进程在进行了 size 所设置的大小的 I/O 后就会结束。

rw 用来设置 I/O 模式。readjob 作业中指定的是表示顺序读出的 read。其他的将在下一节中介绍。

最后再定义一个名称为 writejob 的作业。

writejob 作业将进行 I/O 的大小设置为 5MB，I/O 模式设置为随机写入（randwrite）。

按照这个设置执行 fio 时，首先会在当前目录下生成用于 I/O 的文件，进行同步处理，取消页面缓存。然后，生成与 readjob 作业相对应的进程和与 writejob 作业相对应的进程，进行各自设置的 I/O 操作。各作业在完成 size 所设置的字节数的 I/O 后，就会结束。所有作业结束后，结果就会显示在控制台上。

输出结果包括很多信息。这些内容将在下一节中详细介绍。

模拟实验的例子和输出的意义

现在已经掌握了基本执行方法，就可以尝试生成并执行模拟某种情况的配置文件，然后对输出进行分析。

先生成符合下列情况的配置文件。

- 在文件数据的基础上执行查询（query）并返回发出请求的进程有两个正在执行。每隔 10 毫秒接受查询的请求并执行，从接受请求到返回结果的演算过程花费 5 毫秒。
- 1 个正在运行的进程，它用来将访问记录写入磁盘。记录每 1 秒通过 direct I/O 写入。通过 AIO 实现 I/O 复用（multiplexing）。
- 与此同时，执行文件系统备份处理。

模拟这种情况的配置文件以及其中使用的各种配置项的意义如表 3-7 所示。另外，模拟时间设置为执行 30 秒后结束。

```
# server_sim.fio

[global]
directory=/mnt/sdb
runtime=30
time_based

[query]
size=100m
rw=randread
ioengine=mmap
thinktime=10k
thinktime_spin=5k
filename=query.dat
numjobs=2
```

```
[log]
size=10m
rw=randrw
ioengine=libaio
iodepth=32
thinktime=1m
thinktime_blocks=2
blocksize=1k,4k
direct=1

[backup]
size=100m
rw=randrw
thinktime=1k
```

表 3-7　fio 的配置项

配　置　项	说　　明
blocksize	1 次的 I/O 大小。单位为字节。使用逗号隔开，可以分别指定 read、write。默认值为 4KB
direct	指定 1 时使用 Direct I/O。未指定时等同于指定为 0。使用经由页面缓存的 I/O
directory	存放 I/O 对象文件的目录。未指定时为当前目录
filename	I/O 对象文件名。默认由 fio 自动决定每个作业及进程的名称。在 filename 中可以指定设备文件。这时不经由文件系统，而是从设备直接读写数据。在上述设置中要让 query 作业的两个进程读取相同的文件，因此明确指定了 query.dat 这一文件名
ioengine	指定进程的 I/O 发出方法。可以指定下列内容。（具有代表性的） • sync　使用 read(2)、write(2) 的一般 I/O • libaio　Linux AIO • posixaio　使用 glibc 的 aio_read(3) 和 aio_write(3) 的 POSIX AIO • mmap　使用 mmap(2) 及 memcpy(3) 的读写 • cpuio　不进行 I/O。用于定义模拟执行 CPU 限制进程和 CPU 有负载的作业
iodepth	使用 AIO 时，同时发出 I/O 数的上限值。用于模拟用户进程为了防止大量并发 I/O 发生竞争而对 I/O 复用度设限的情况。默认值为 1
numjobs	通过这个作业定义执行的进程数。未指定时进程数为 1
runtime	执行时间。单位为秒。执行作业的进程在 I/O 的数据量达到 size 指定的值，或经过 runtime 指定的时间后就会结束。（参照 time_based）

（续）

配 置 项	说 明
rw	指定 I/O 模式。可以指定下列内容。 • read 顺序读 • write 顺序写 • randread 随机读 • randwrite 随机写 • rw 顺序读 / 写混合 • randrw 随机读 / 写混合
size	指定到作业结束为止进行 I/O 的数据量。单位为字节
thinktime	从发出 I/O 到下一次发出 I/O 的等待时间。单位为微秒。等待时进程通过 nanosleep(2) 休眠。（参照 thinktime_spin）
thinktime_blocks	设置在发出几次 I/O 后进入 thinktime 设置的等待。默认值为 1
thinktime_spin	设置 thinktime 指定的等待时间中不休眠而是实际消耗 CPU 等待的时间。单位为微秒
time_based	指定到达 runtime 所指定的时间为止反复作业。未指定 time_based 时，size 指定的数量的 I/O 完成后，作业的进程结束。指定 time_based 时，在 runtime 设置的时间内持续发出 I/O

然后实际执行，并详细查看输出结果。

```
$ fio server_sim.fio

query: (g=0): rw=randread, bs=4K-4K/4K-4K, ioengine=mmap, iodepth=1
query: (g=0): rw=randread, bs=4K-4K/4K-4K, ioengine=mmap, iodepth=1
log: (g=0): rw=randrw, bs=1K-1K/4K-4K, ioengine=libaio, iodepth=32
backup: (g=0): rw=randrw, bs=4K-4K/4K-4K, ioengine=sync, iodepth=1
Starting 4 processes
query: Laying out IO file(s) (1 file(s) / 100MB)
log: Laying out IO file(s) (1 file(s) / 10MB)
backup: Laying out IO file(s) (1 file(s) / 100MB)

query: (groupid=0, jobs=1): err= 0: pid=3158
  read : io=6,740KB, bw=225KB/s, iops=56, runt= 30006msec
    clat (usec): min=6, max=124K, avg=7553.49, stdev=5995.06
    bw (KB/s) : min=  162, max=  254, per=25.31%, avg=225.24, stdev=15.52
  cpu          : usr=28.83%, sys=0.24%, ctx=3544, majf=1639, minf=75
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued r/w: total=1685/0, short=0/0
     lat (usec): 10=2.67%, 20=0.06%, 500=0.18%, 750=0.65%, 1000=0.36%
```

```
     lat (msec): 2=1.54%, 4=15.73%, 10=58.16%, 20=17.69%, 50=2.79%
     lat (msec): 100=0.12%, 250=0.06%

query: (groupid=0, jobs=1): err= 0: pid=3159
  read : io=6,772KB, bw=226KB/s, iops=56, runt= 30008msec
    clat (usec): min=6, max=128K, avg=7470.76, stdev=6163.17
    bw (KB/s) : min=  181, max=  260, per=25.45%, avg=226.47, stdev=14.27
  cpu          : usr=28.99%, sys=0.20%, ctx=3622, majf=1630, minf=88
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued r/w: total=1693/0, short=0/0
     lat (usec): 10=3.66%, 20=0.06%, 500=0.35%, 750=0.77%, 1000=0.06%
     lat (msec): 2=1.24%, 4=16.42%, 10=55.64%, 20=19.31%, 50=2.30%
     lat (msec): 100=0.12%, 250=0.06%

log: (groupid=0, jobs=1): err= 0: pid=3160
  read : io=39,936B, bw=1,311B/s, iops=1, runt= 30449msec
    slat (usec): min=8, max=63, avg=27.92, stdev=19.38
    clat (msec): min=36, max=16,815, avg=10042.82, stdev=5584.03
    bw (KB/s) : min=    0, max=    1, per=0.01%, avg= 0.06, stdev= 0.24
  write: io=197KB, bw=6,625B/s, iops=1, runt= 30449msec
    slat (usec): min=10, max=89, avg=33.40, stdev=22.97
    clat (msec): min=33, max=16,778, avg=11045.30, stdev=5371.05
    bw (KB/s) : min=    0, max=    7, per=0.56%, avg= 2.62, stdev= 2.59
  cpu          : usr=0.00%, sys=0.01%, ctx=31, majf=0, minf=21
  IO depths    : 1=1.1%, 2=2.2%, 4=4.5%, 8=9.0%, 16=18.0%, 32=65.2%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=98.3%, 8=0.0%, 16=0.0%, 32=1.7%, 64=0.0%, >=64=0.0%
     issued r/w: total=39/50, short=0/0
     lat (msec): 50=2.25%, 2000=4.49%, >=2000=93.26%

backup: (groupid=0, jobs=1): err= 0: pid=3161
  read : io=13,556KB, bw=452KB/s, iops=112, runt= 30005msec
    clat (usec): min=419, max=104K, avg=6669.29, stdev=4584.93
    bw (KB/s) : min=  341, max=  525, per=50.82%, avg=452.29, stdev=40.41
  write: io=14,072KB, bw=469KB/s, iops=117, runt= 30005msec
    clat (usec): min=10, max=98,704, avg=79.48, stdev=2190.03
    bw (KB/s) : min=  321, max=  632, per=100.32%, avg=469.52, stdev=72.87
  cpu          : usr=0.38%, sys=0.74%, ctx=10302, majf=0, minf=28
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued r/w: total=3389/3518, short=0/0
     lat (usec): 20=38.63%, 50=12.25%, 500=0.51%, 750=0.61%, 1000=0.04%
     lat (msec): 2=1.61%, 4=9.64%, 10=28.99%, 20=7.37%, 50=0.25%
     lat (msec): 100=0.10%, 250=0.01%

Run status group 0 (all jobs):
   READ: io=27,107KB, aggrb=890KB/s, minb=1KB/s, maxb=462KB/s, mint=30005msec,
maxt=30449msec
  WRITE: io=14,269KB, aggrb=468KB/s, minb=6KB/s, maxb=480KB/s, mint=30005msec,
maxt=30449msec

Disk stats (read/write):
```

```
  sdb: ios=6698/2302, merge=0/26478, ticks=48982/1143531, in_queue=1207586,
util=93.27%
```

第一部分输出的是已执行进程的信息，分别输出了名称、I/O 模式、块的大小等设置。有两个 query 是因为指定为 `numjob=2`，所以要执行两个进程。

接下来第一行为 query 的两个段落是 query 作业两个进程的结果，其后显示的分别是 `log` 作业和 `backup` 作业的结果。下面对比各部分的内容，看一下各输出表示的意义。

从 `read:` 和 `write:` 开始的 3 行或 4 行表示的是关于读入和写入的数据量、平均带宽、延迟。平均带宽是将总数据量除以作业执行时间得出的值，因此在 `thinktime` 较长的作业中值较低。`clat`（completion latency）的行为延迟，是作业进程开始向内核发出 I/O 请要到完成为止的时间，即从调出 read(2) 等直到返回的时间。`log` 作业由于使用的是 AIO，因此多输出一个 `slat`（submitting latency）延迟。这表示 I/O 发出要求滞留在作业进程内部的 `queue` 时间。

从 `query` 作业的延迟来看，平均花费 7.5 毫秒左右，最多花费 130 毫秒左右。10 毫秒的查询间隔中大部分时间都耗费在等待 I/O 上。`log` 作业中出现了最长 16 秒的延迟。这时应当重新设计系统，分散 I/O 负载。

小贴士：`clat` 在 read(2) 或 Direct I/O 的 write(2) 中 I/O 实际花费的时间，而在常用的经由页面缓存的 write(2) 中只是将数据存放到页面缓存所需的时间。上例中 `backup` 作业的 `write` 就相当于这种情况，因此其值平均为 79 微秒，比其他情况短得多。

I/O depths 的行表示通过 AIO 实现复用的 I/O 数与时间之间的关系。例如，`log` 作业的 `IO depths` 中的“`8=9.0%，16=18.0%`”，就表示 9~16 个 I/O 同时执行的时间相当于所有执行时间的 18.0%。由于 I/O 复用只存在于 AIO 的情况下，因此在 `log` 以外的作业都是“`1=100%`”。

从 `log` 作业的 `IO depths` 可以看出，65.2% 的时间都达到复用上限值附近的 19~32。这也显示 I/O 负载仍然过高。

最后两个段落是关于所有进程总共的统计值。输出所有 I/O 的数据量的合计结果等。

小结

本节使用 fio 进行实际使用情况的 I/O 模拟实验，介绍了输出的结果。fio 有非常多的选项，可以进行更加精确和详细的设置，由于数量庞大，这里无法一一列举。源代码包含的文档和 man 中有详细的记载，但都是英文。参考这些内容并熟练使用 I/O 操作，I/O 基准测试就能成为强有力的工具。

——Munehiro IKEDA

HACK #21　FUSE

本节将介绍使用用户进程的文件系统框架——FUSE。

FUSE 概要

FUSE（Filesystem in Userspace，用户空间文件系统），是用来生成用户空间的一般进程的框架。使用 FUSE，就可以以一般应用程序进程的形式生成独特的文件系统，与已有的文件系统同样进行挂载。从 Linux 2.6.14 开始实际安装 FUSE。

例如，在最近的 Linux 发布版中，有一些标准配置用于挂载 Windows 的文件系统 NTFS 的 ntfs-3g（Ubuntu 等）。当连接到存在 NTFS 格式文件系统的分区表所在的磁盘时，经由 gvfs 挂载。此时，使用 FUSE 安装到用户空间的 `ntfs-3g` 开始将 NTFS 文件系统挂载到用户的目录树。

此外还有 ZFS on FUSE。ZFS 适用的是 CDDL 这个与 GPL 没有互换性的许可证。因此它不能整合到拥有 GPL 许可证的 Linux 内核中。在这种情况下，使用 FUSE 安装 ZFS，从而能够使用 ZFS 文件系统，就是 ZFS on FUSE。一旦启动 `zfs-fuse` 守护进程（demon），就可以使用 `zfs` 或 `zpool` 命令进行 ZFS 的操作。

安装 FUSE 文件系统

libfuse 就是用于简单安装使用 FUSE 的独特文件系统的标准库。例如，安装并读入专用的文件系统时，只需要生成下列 4 个功能。

`getattr`	获取文件属性
`readdir`	获取目录条目
`open`	打开文件
`read`	读入文件

卸载

使用 FUSE 挂载的文件系统，可以使用 `fusermount` 命令来卸载。

```
$ fusermount -u 挂载点
```

使用 FUSE 的文件系统

这里将介绍使用从普通发布版中可以获取的 FUSE 的文件系统。

`sshfs`

将可以把 SSH 远程登录的机器的目录挂载到本地机器上，在与远程服务器频繁进行文

件交换时非常便利。例如，将 remote 主机的 admin 用户的 foo 目录挂载到本地的 ~/remote-foo 时，命令行如下。类似一般的 SSH，同样要求输入密码。

```
$ sshfs hshimamoto@remote:foo ~/remote-foo
```

在此之后，对本地机器的 ~/remote 的 foo 目录进行的处理就是对远程机器 remote 的 foo 目录的处理。

fuseiso

挂载 ISO 映像文件。一般挂载 ISO 映像时是使用 loopback 设备进行挂载的，但有时普通用户权限是无法操作 loopback 设备的。另外 loopback 设备还存在可利用次数的限制。

使用 fuseiso，就可以很简单地将 ISO 映像挂载到自己的文件系统，并参照其内容。

```
# fuseiso foo.iso ~/iso
```

encfs

加密文件系统。用户将特定目录加密，然后挂载将该目录解密后的文件系统。

已加密的文件（目录）名和内容通过文件系统生成时指定的算法加密，但需要注意的是，目录中的文件数或文件大小、属性等不会因为加密而修改。

命令示例如下。创建目录 enc-foo，并挂载为 dec-foo。

```
$ mkdir ~/enc-foo
$ mkdir ~/dec-foo
$ encfs ~/enc-foo ~/dec-foo
```

首次挂载加密目录时，将进行加密的初始设置。将显示如下信息，输出用来进行加密设置的提示符。

生成新的加密卷标。

```
Please choose from one of the following options:
 enter "x" for expert configuration mode,
 enter "p" for pre-configured paranoia mode,
 anything else, or an empty line will select standard mode.
?>
```

针对详细的设置这里不作介绍。直接按【Enter】键，使用默认设置。

```
Standard configuration selected.
```

设置已完成。生成下列属性的文件系统：

```
文件系统加密算法："ssl/aes", 版本 3:0:2
Filename encoding: "nameio/block", version 3:0:1
键大小：192 位
Block Size: 1024 bytes
Each file contains 8 byte header with unique IV data.
Filenames encoded using IV chaining mode.
File holes passed through to ciphertext.
```

```
Now you will need to enter a password for your filesystem.
You will need to remember this password, as there is absolutely
no recovery mechanism.  However, the password can be changed
later using encfsctl.

新的 Encfs 密码:
确认 Encfs 密码:
```

再设置用来加密、解密的密码就完成了。尝试在 dec-foo 目录下创建文件。

```
$ vi ~/dec-foo/bar.txt
$ cat ~/dec-foo/bar.txt
Hello Encfs!!
```

卸载 dec-foo 后，就难以用普通的方法来参照这个加密目录。

```
$ fusermount -u ~/dec-foo
```

看一下 enc-foo 目录，可以发现由于已经加密，因此内容不是文本，而是二进制数据。

```
$ ls enc-foo/
QQJqAaAW-hx1QiWGH8fRHplZ
$ file enc-foo/QQJqAaAW-hx1QiWGH8fRHplZ
enc-foo/QQJqAaAW-hx1QiWGH8fRHplZ: data
```

再次使用 encfs 命令进行挂载时，这次只要求输入密码。

```
$ encfs ~/enc-foo ~/dec-foo
EncFS 密码:
$ ls dec-foo/
bar.txt
$ cat dec-foo/bar.txt
Hello Encfs!!
```

小结

本节介绍了 FUSE，还介绍了一些使用 FUSE 的便捷文件系统。此外可能还有其他使用 FUSE 的文件系统。

参考文献

- Documentation/filesystems/fuse.txt
- FUSE(http://fuse.sourceforge.net/)

——Hiroshi Shimamoto

LINUX KERNEL HACKS™

LINUX内核精髓 精通LINUX内核必会的75个绝技

经过近20年的发展，Linux操作系统已经成为当今最成功的开源软件之一，使用广泛，影响深远。随着Linux操作系统功能的不断丰富和完善，Linux内核的源代码也从最初的几万行增加到如今的数百万行，庞大无比，对于Linux内核的研究者和开发者而言，要系统研究Linux内核绝非易事。鉴于此，本书选取了Linux内核的资源管理（CPU、内存、进程等）、文件系统、网络、虚拟化、省电、调试、概要分析、追踪、内核调整等核心主题进行了深入剖析和讲解，总结出了75个能使读者深刻理解Linux内核精髓的技巧和最佳实践。

一线内核技术专家的经验和智慧结晶，深刻解读Linux内核的资源管理、文件系统、网络、虚拟化、省电技术、调试、性能调优、分析与追踪等核心主题。

客服热线：(010) 88378991，88361066
购书热线：(010) 68326294，88379649，68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com
华章网站：*http://www.hzbook.com*
网上购书：*www.china-pub.com*

O'Reilly Media, Inc.授权机械工业出版社出版

O'REILLY®
oreilly.com.cn

ISBN 978-7-111-41049-2
9 787111 410492 >

定价：79.00元