

# 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、島本 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



---

# Linux 内核精髓

## 精通 Linux 内核必会的 75 个绝技

---

# 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

© 2012 by O'Reilly Japan, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Japan, Inc. and China Machine Press, 2013.  
Authorized translation of the Japanese edition, 2012 O'Reilly Japan, Inc., the owner of all rights to publish  
and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

日文原版由 O'Reilly Japan, Inc. 出版 2012。

简体中文版由机械工业出版社出版 2013。日文原版的翻译得到 O'Reilly Japan, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Japan, Inc. 的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

封底无防伪标均为盗版

本法律法律顾问

北京市展达律师事务所

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/ 董城市京瑞印刷有限公司印刷

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

### 畠山 大輔 (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、畠山大輔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內容，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內容并不是很多，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，由于路由表的人口数量庞大，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<sup>注1</sup>。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<sup>注2</sup>、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

## 编者与作者介绍

## 主编致辞

## 前言

## 第 1 章 内核入门 ..... 1

HACK #1	如何获取 Linux 内核	1
HACK #2	如何编译 Linux 内核	7
HACK #3	如何编写内核模块	18
HACK #4	如何使用 Git	22
HACK #5	使用 checkpatch.pl 检查补丁的格式	41
HACK #6	使用 localmodconfig 缩短编译时间	44

## 第 2 章 资源管理 ..... 47

HACK #7	Cgroup、Namespace、Linux 容器	47
HACK #8	调度策略	55
HACK #9	RT Group Scheduling 与 RT Throttling	59
HACK #10	Fair Group Scheduling	62
HACK #11	cpuset	65
HACK #12	使用 Memory Cgroup 限制内存使用量	68

HACK #13 使用 Block I/O 控制器设置 I/O 优先级 .....	74
HACK #14 虚拟存储子系统的调整 .....	80
HACK #15 ramzswap .....	85
HACK #16 OOM Killer 的运行与结构 .....	91
<b>第 3 章 文件系统 .....</b>	<b>98</b>
HACK #17 如何使用 ext4 .....	98
HACK #18 向 ext4 转换 .....	101
HACK #19 ext4 的调整 .....	104
HACK #20 使用 fio 进行 I/O 的基准测试 .....	111
HACK #21 FUSE .....	118
<b>第 4 章 网络 .....</b>	<b>121</b>
HACK #22 如何控制网络的带宽 .....	121
HACK #23 TUN/TAP 设备 .....	126
HACK #24 网桥设备 .....	129
HACK #25 VLAN .....	133
HACK #26 bonding 驱动程序 .....	136
HACK #27 Network Drop Monitor .....	141
<b>第 5 章 虚拟化 .....</b>	<b>147</b>
HACK #28 如何使用 Xen .....	147
HACK #29 如何使用 KVM .....	153
HACK #30 如何不使用 DVD 安装操作系统 .....	159
HACK #31 更改虚拟 CPU 分配方法，提高性能 .....	161
HACK #32 如何使用 EPT 提高客户端操作系统的性能 .....	166
HACK #33 使用 IOMMU 提高客户端操作系统运行速度 .....	173
HACK #34 使用 IOMMU+SR-IOV 提高客户端操作系统速度 .....	183
HACK #35 SR-IOV 带宽控制 .....	187
HACK #36 使用 KSM 节约内存 .....	189
HACK #37 如何挂载客户端操作系统的磁盘 .....	194

HACK #38 从客户端操作系统识别虚拟机环境 .....	200
HACK #39 如何调试客户端操作系统 .....	205
<b>第 6 章 省电 .....</b>	<b>213</b>
HACK #40 ACPI .....	213
HACK #41 使用 ACPI 的 S 状态 .....	224
HACK #42 使用 CPU 省电 (C、P 状态) .....	226
HACK #43 PCI 设备的热插拔 .....	236
HACK #44 虚拟环境下的省电 .....	240
HACK #45 远程管理机器的电源 .....	246
HACK #46 USB 的电力管理 .....	251
HACK #47 显示器的省电 .....	254
HACK #48 通过网络设备节省电能 .....	260
HACK #49 关闭键盘的 LED 来省电 .....	263
HACK #50 PowerTOP .....	269
HACK #51 硬盘的省电 .....	276
<b>第 7 章 调试 .....</b>	<b>282</b>
HACK #52 SysRq 键 .....	282
HACK #53 使用 diskdump 提取内核崩溃转储 .....	288
HACK #54 使用 Kdump 提取内核崩溃转储 .....	293
HACK #55 崩溃测试 .....	297
HACK #56 IPMI 看门狗计时器 .....	299
HACK #57 NMI 看门狗计时器 .....	305
HACK #58 soft lockup .....	307
HACK #59 crash 命令 .....	312
HACK #60 核心转储过滤器 .....	326
HACK #61 生成用户模式进程的进程核心转储 .....	329
HACK #62 使用 lockdep 查找系统的死锁 .....	335
HACK #63 检测内核的内存泄漏 .....	341

<b>第 8 章 概要分析与追踪</b>	<b>346</b>
HACK #64 使用 perf tools 的概要分析 (1)	346
HACK #65 使用 perf tools 的概要分析 (2)	349
HACK #66 进行内核或进程的各种概要分析	353
HACK #67 追踪内核的函数调用	360
HACK #68 ftrace 的插件追踪器	366
HACK #69 记录内核的运行事件	371
HACK #70 使用 trace-cmd 的内核追踪	378
HACK #71 将动态追踪事件添加到内核中	382
HACK #72 使用 SystemTap 进行内核追踪	388
HACK #73 使用 SystemTap 编写对话型程序	394
HACK #74 SystemTap 脚本的重复利用	399
HACK #75 运用 SystemTap	402

# 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<sup>注1</sup>。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 的下一个版本将是 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”<sup>注2</sup>。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。

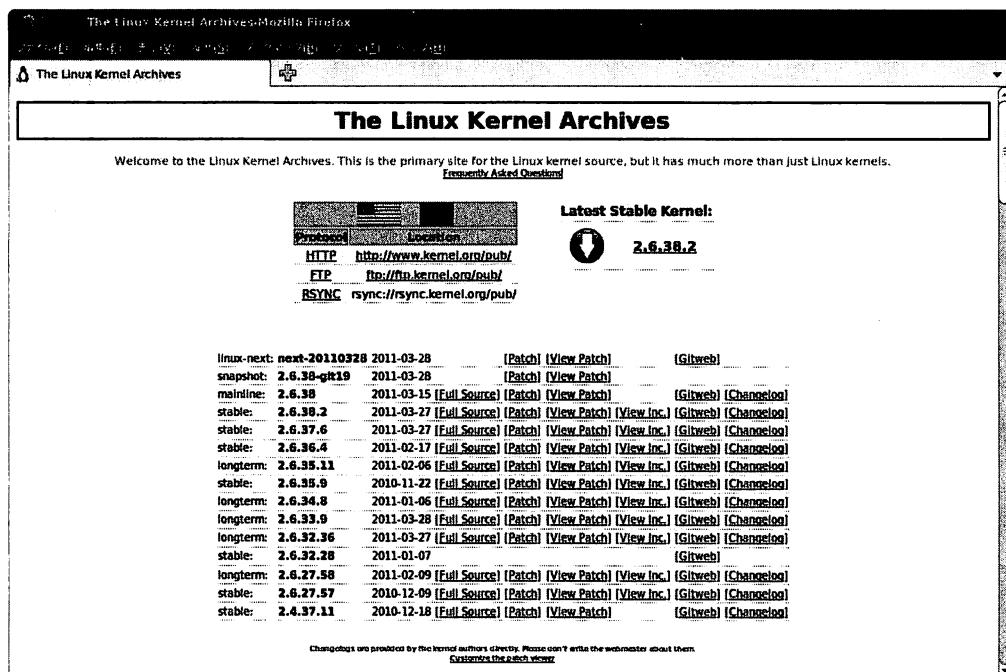


图 1-1 <http://www.kernel.org>

### 下载 tar 文件

获取 Linus 树最简单的方法就是从 [kernel.org](http://www.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对外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。

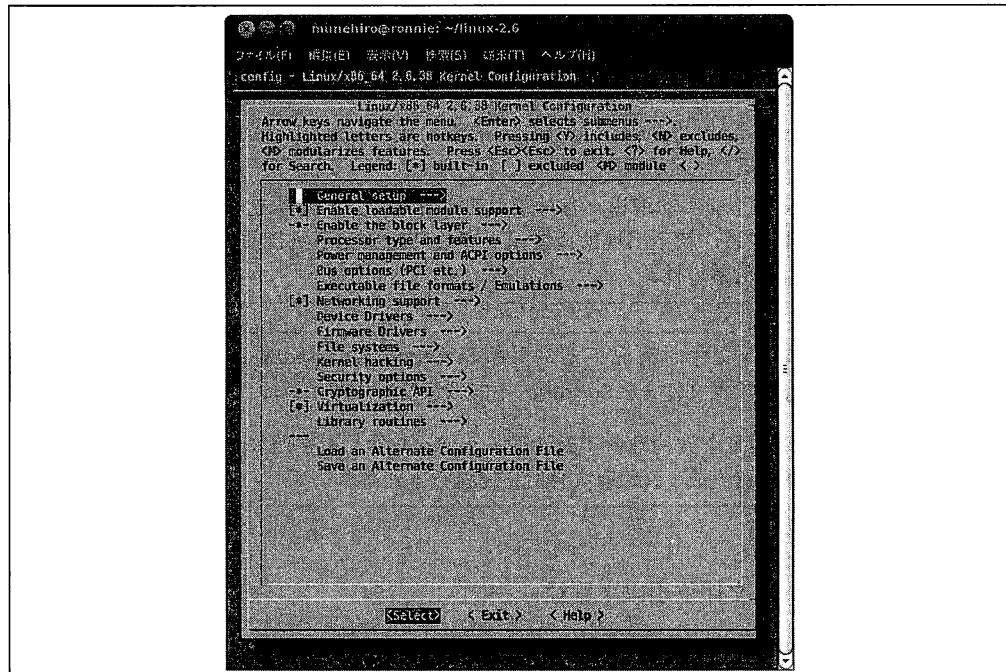


图 1-2 make menuconfig 生成的设置画面

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)

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。

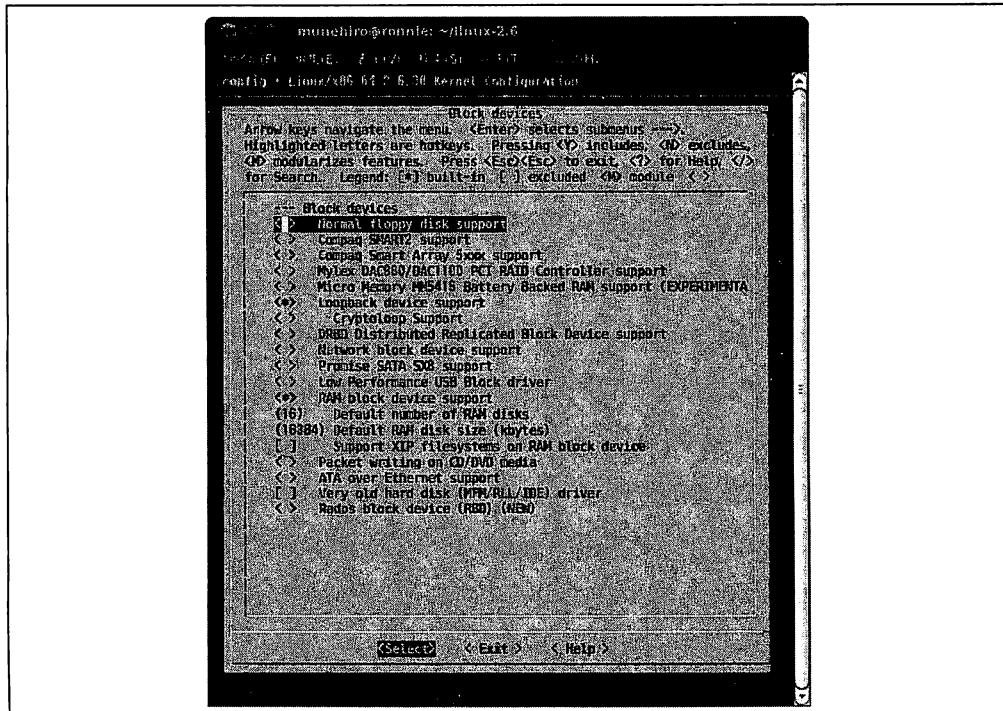


图 1-3 Normal floppy disk 支持的设置

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中选择操作菜单的<Exit>，或者连接两次【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。

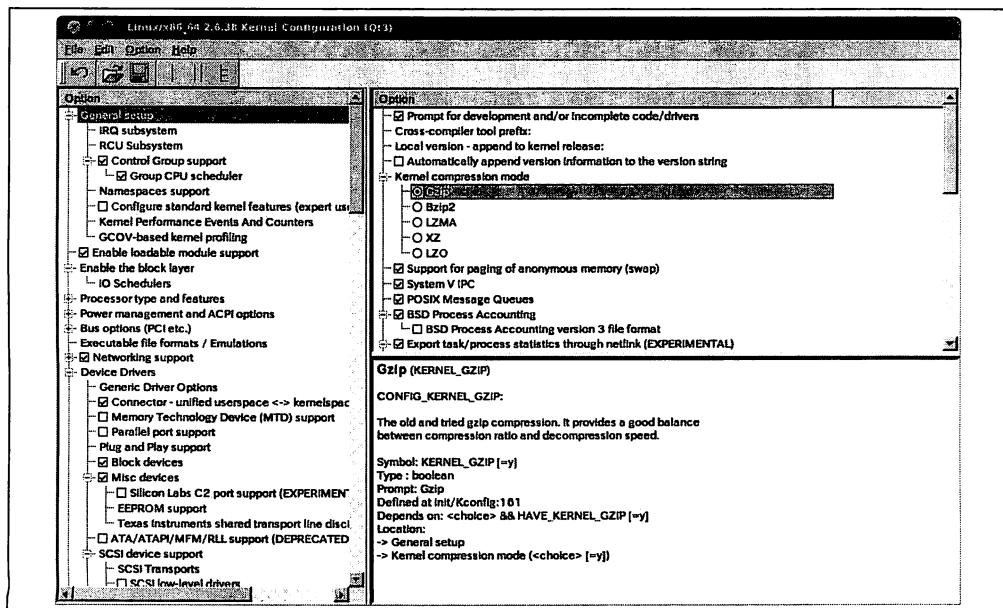


图 1-4 make xconfig 生成的设置界面

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，就在那里进行设置。

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 初始化文件系统映像
/boot/initrd.img-< 内核版本 >	
/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

对 象	说 明
<code>mrproper</code>	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 <code>.config</code> 文件
<code>help</code>	显示可以使用的 <code>make</code> 对象
<code>tags</code>	生成标签文件。有了标签文件, 就能使用 <code>Emacs</code> 等编辑器的 <code>tag jump</code> 功能跳到函数定义处, 可以高效进行源码浏览
<code>cscope</code>	生成用于 <code>cscope</code> 的索引文件。 <code>cscope</code> 是基于控制台(文字界面)的源码浏览器
<code>allyesconfig</code>	生成将所有设置项目设置为有效并静态添加到内核中的 <code>.config</code> 文件
<code>allnoconfig</code>	生成将允许范围内的设置项目设置为无效的 <code>.config</code> 文件
<code>allmodconfig</code>	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 <code>.config</code> 文件
<code>&lt;dir&gt;/&lt;file&gt;.o</code>	仅进行生成指定的目标文件所必需的编译。当仅指定 <code>&lt;dir&gt;</code> 时, 由 <code>.config</code> 文件生成该目录内所有目标文件
<code>&lt;dir&gt;/&lt;file&gt;.ko</code>	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
<code>V=0 1 2</code>	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”)
<code>O=&lt;dir&gt;</code>	将编译最终生成的文件全部输出到 <code>&lt;dir&gt;</code> 。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_init()` 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。

---

<code>MODULE_AUTHOR()</code>	表示模块的作者
<code>MODULE_LICENSE()</code>	表示模块的许可证
<code>MODULE_DESCRIPTION()</code>	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 `printf()` 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 `printf()` 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 `printf()` 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("Hello 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
goodbye.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 000000..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 c47feeeef44a652bda15dbb580d48213dc1294664
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 -l
```

## 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
```

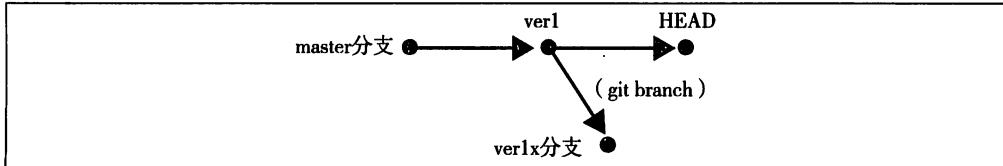


图 1-5 分支的创建

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。

---

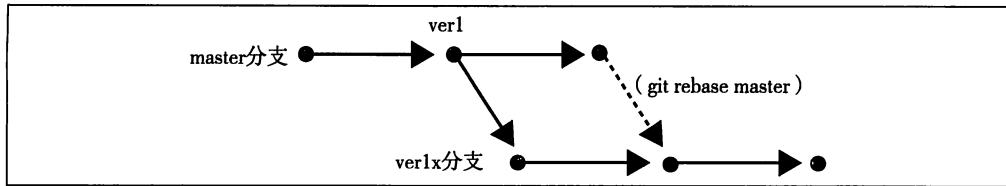


图 1-6 分支的 rebase

## 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。

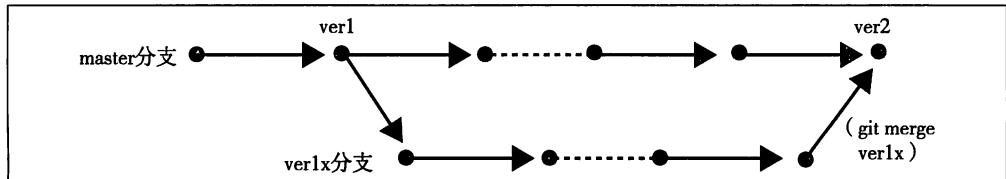


图 1-7 分支的合并

```
$ 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。

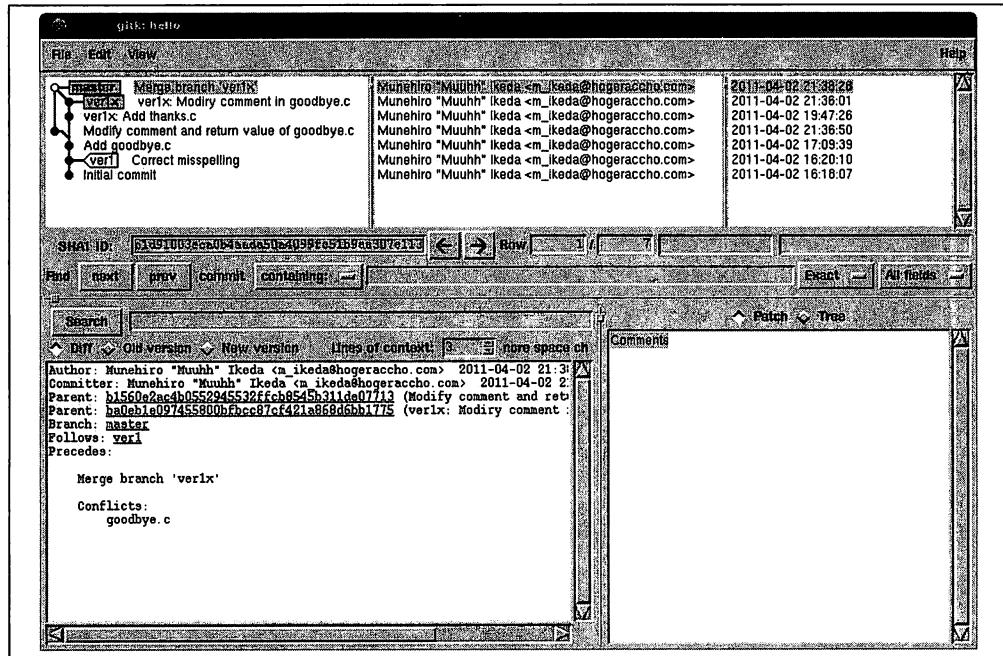


图 1-8 gitk

## 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`。

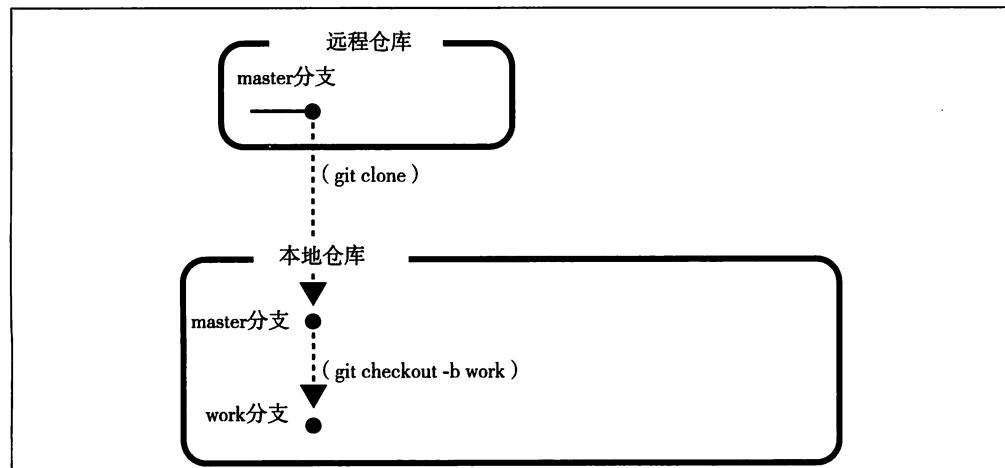


图 1-9 本地分支的建立

```
$ 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".
```

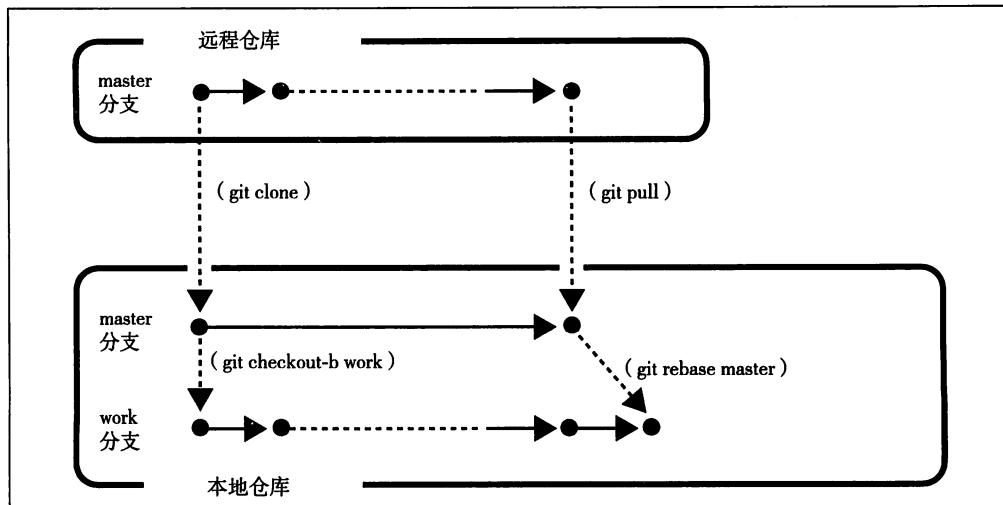


图 1-10 本地开发分支的复位基底

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-encryption	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;
    }
```

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
<code>git push</code>	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
<code>git fetch</code>	从远程仓库将修改抓取到追踪分支。与 <code>git pull</code> 不同的是不进行与本地分支的合并。 <code>git pull</code> 是在内部调出 <code>git fetch</code> 和 <code>git merge</code> 操作的
<code>git stash</code>	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 <code>git stash pop</code> 来调出
<code>git cherry-pick</code>	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
<code>git gc</code>	删除未使用的对象或文件，优化仓库
<code>git am</code>	从 mbox 格式的文件导入补丁并提交
<code>git bisect</code>	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-localmod/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。相信凡是你自己构建（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

# 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 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           tasks
```

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
<code>release_agent</code>	RW	删除分组时执行的命令。这个文件只存在于根分组
<code>notify_on_release</code>	RW	设置是否执行 <code>release_agent</code> 。为 1 时执行
<code>tasks</code>	RW	属于分组的线程 TID 列表
<code>cgroup.procs</code>	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 <code>tasks</code> 不同
<code>cgroup.event_control</code>	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。debootstrap 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_FIFO
  -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](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_CGROUPS=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.mems 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    S1+  238:37 qemu-kvm  
2628 pts/3    S1+  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.failcnt	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<sup>审校者注1</sup>，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}
```

---

<sup>审校者注1</sup>：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
```

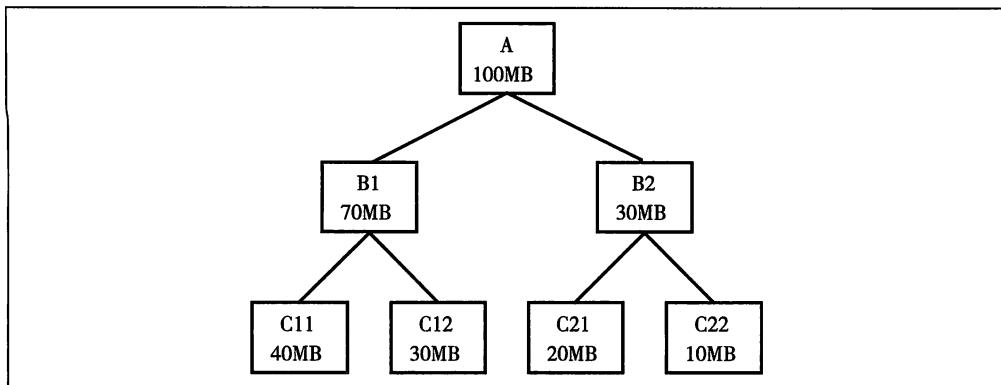


图 2-1 创建的分组结构

## 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 <code>mlock</code> 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_pgpfout	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 mlock 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_CFS_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 assigned

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时约 13 秒，而属于 /cgroup/high 的进程完成读入耗时约 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 `ulimit -v` 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` 人口来进行控制。

- `/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。根据系统的特点，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 ?
apache    4471  0.0  0.3 238244  6516 ?
apache    4472  0.0  0.3 238244  6512 ?
apache    4473  0.0  0.3 238244  6512 ?
...
...
```

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 ----- 00000000000036000 008:00005 mod_perl.so
00002aff7ac6d000 16 rw--- 00000000000035000 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--- 00007fffffeaa000 000:00000 [ stack ]
ffffffffffff600000 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<sup>注1</sup>。

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。

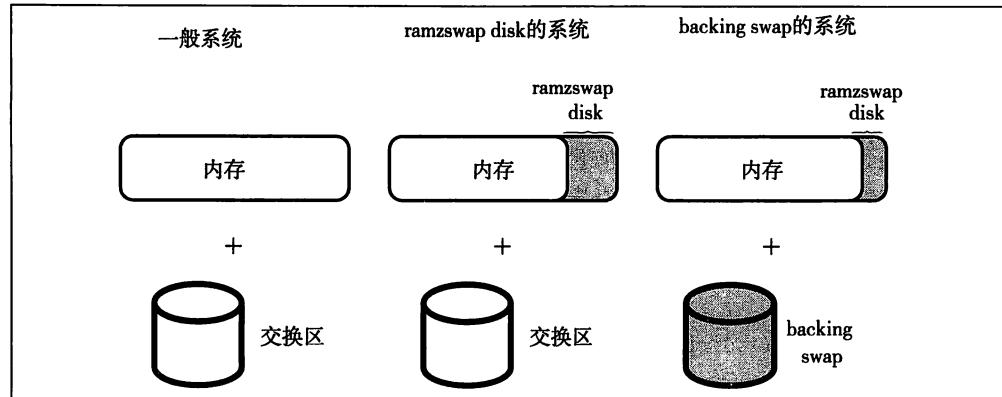


图 2-2 一般系统与使用 ramzswap disk、backing swap 的系统的内存和交换区的关系

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 rzscontrol 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
<code>DiskSize: 255 852KB</code>	<code>ramzswap disk</code> 的大小。默认设置为物理内存的 25%
<code>NumReads: 91 576</code>	从 <code>ramzswap</code> 设备读出的页面总数（解压缩页面的次数）
<code>NumWrites: 224 893</code>	写入 <code>ramzswap</code> 设备的页面总数（压缩页面的次数）
<code>FailedReads: 0</code>	因解压缩失败等导致 <code>swap</code> 区域的数据读取失败的页面数
<code>FailedWrites: 0</code>	因压缩失败等导致无法写入 <code>swap</code> 区域的页面数
<code>InvalidIO: 0</code>	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 请求数会撤销
<code>NotifyFree: 220 016</code>	释放的页面总数
<code>ZeroPages: 163</code>	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
<code>GoodCompress: 96%</code>	成功将大小压缩到页面大小 50% 以下的比例
<code>NoCompress: 0%</code>	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
<code>PagesStored: 4871</code>	正在换出的页面数。包括未能压缩（已计入 <code>NoCompress</code> ）的页面。不包括零页面
<code>PagesUsed: 1255</code>	<code>ramzswap</code> 实际使用的页面数。这是 <code>ramzswap</code> 保留的内存存量
<code>OrigDataSize: 19 484KB</code>	有换出要求的数据的合计大小。 $\text{PagesStored} \times \text{页面大小}$ （一般为 4096 字节）。 该值对应压缩前的大小
<code>ComprDataSize: 4801KB</code>	换出后数据的合计大小。 $\text{PagesUsed} \times \text{页面大小}$ （一般为 4096 字节）。该值对应压缩后的大小
<code>MemUsedTotal: 5020KB</code>	<code>ramzswap</code> 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

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*4096kB
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。

- 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`<sup>注2</sup> 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`<sup>注3</sup> 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（TID）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的连续终止多个进程。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）<sup>注1</sup>、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 ctime<sup>注2</sup>。

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 check sum）	—	○
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。

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
<code>b</code>	指定块的大小
<code>F</code>	强制执行 <code>mke2fs</code>
<code>I</code>	指定索引节点大小 (字节单位)
<code>O</code>	文件系统功能的启用 / 禁用
<code>T</code>	指定文件系统的种类
<code>v</code>	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	○	○
nodealloc	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](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 -O extent /dev/sdb1
```

当前设置的功能标志可以同样使用 e2fsprogs 工具包的 debugfs 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 <sup>注3</sup>	-
auto_64-bit_support <sup>注4</sup>	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](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_blk	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 `session_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_kbytes` 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 `e2fsprogs` 工具包的 `filefrag` 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 33280 分配 16 块，而从其后的物理偏移量 33296 分配的是 `file2` 的第一个范围的块。这样就会产生碎片<sup>注5</sup>。

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<sup>注6</sup>。

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_blk`s 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_blk`s 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_blk`s 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_blk`s 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_blk`s 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_blk`s 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](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
<code>blocksize</code>	1 次的 I/O 大小。单位为字节。使用逗号隔开，可以分别指定 <code>read</code> 、 <code>write</code> 。默认值为 4KB
<code>direct</code>	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
<code>directory</code>	存放 I/O 对象文件的目录。未指定时为当前目录
<code>filename</code>	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 <code>filename</code> 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 <code>query</code> 作业的两个进程读取相同的文件，因此明确指定了 <code>query.dat</code> 这一文件名
<code>ioengine</code>	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） <ul style="list-style-type: none"> <li><code>sync</code> 使用 <code>read(2)</code>、<code>write(2)</code> 的一般 I/O</li> <li><code>libaio</code> Linux AIO</li> <li><code>posix aio</code> 使用 glibc 的 <code>aio_read(3)</code> 和 <code>aio_write(3)</code> 的 POSIX AIO</li> <li><code>mmap</code> 使用 <code>mmap(2)</code> 及 <code>memcpy(3)</code> 的读写</li> <li><code>cpuio</code> 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</li> </ul>
<code>iodepth</code>	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
<code>numjobs</code>	通过这个作业定义执行的进程数。未指定时进程数为 1
<code>runtime</code>	执行时间。单位为秒。执行作业的进程在 I/O 的数据量达到 <code>size</code> 指定的值，或经过 <code>runtime</code> 指定的时间后就会结束。（参照 <code>time_based</code> ）

配置项	说明
<code>rw</code>	指定 I/O 模式。可以指定下列内容。
	<ul style="list-style-type: none"> <li>• <code>read</code> 顺序读</li> <li>• <code>write</code> 顺序写</li> <li>• <code>randread</code> 随机读</li> <li>• <code>randwrite</code> 随机写</li> <li>• <code>rw</code> 顺序读 / 写混合</li> <li>• <code>randrw</code> 随机读 / 写混合</li> </ul>
<code>size</code>	指定到作业结束为止进行 I/O 的数据量。单位为字节
<code>thinktime</code>	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 <code>nanosleep(2)</code> 休眠。(参照 <code>thinktime_spin</code> )
<code>thinktime_blocks</code>	设置在发出几次 I/O 后进入 <code>thinktime</code> 设置的等待。默认值为 1
<code>thinktime_spin</code>	设置 <code>thinktime</code> 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
<code>time_based</code>	指定到达 <code>runtime</code> 所指定的时间为止反复作业。未指定 <code>time_based</code> 时, <code>size</code> 指定的数量的 I/O 完成后, 作业的进程结束。指定 <code>time_based</code> 时, 在 <code>runtime</code> 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 `I/O 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 `I/O 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

# 网 絡

本章将介绍与网络相关的内核功能。内容包括 TUN/TAP 设备、网桥设备、VLAN 设备、bonding 设备和网络调度、dropwatch。

## HACK #22 如何控制网络的带宽

本节介绍控制网络带宽的功能。

Linux 内核中安装有称为网络调度或分组调度的带宽控制。这个功能可以控制每个网络设备传输的吞吐量。通过控制带宽，就可以优先进行重要的通信，或者优先其他服务的性能。

本节将以 CentOS 5.4 所提供的 CBQ (Class Based Queueing) 的设置为例进行说明。需要安装 iproute 数据包。

### 设置带宽控制

CBQ 是指以具有优先级的类为单位分配传输带宽。为每个类准备配置文件并介绍控制带宽的方法。

iproute 工具包中具有样本 /etc/sysconfig/cbq/cbq-0000.example。

```
# cat /etc/sysconfig/cbq/cbq-0000.example
DEVICE=eth0,10Mbit,1Mbit
RATE=128Kbit
WEIGHT=10Kbit
PRIO=5
RULE=192.168.1.0/24
```

下面以这个样本为基础，对各个项进行说明。

DEVICE

DEVICE 选项的是作为带宽控制对象的网络设备。

格式如下。

```
DEVICE=<iframe>,<bandwidth>[,<weight>]
```

`ifname` 网络接口。在上例中为 `eth0`。`bandwidth` 指定的是网络设备的物理带宽。如果是千兆以太网 (Gigabit Ethernet) 则为 1000Mbit。`weight` 是根类的比重。值越大，表示根类一次处理的数据比例越大。根类是针对每个设备分别生成的。推荐将 `weight` 指定为 `bandwidth` 的 1/10。

#### RATE

`RATE` 指定的是分配到该类的带宽。单位为 bit 或 bps。

`bit` 对应 bits/ 秒，`bps` 对应 bytes/ 秒。设置为 100Mbytes/ 秒时，需指定 `RATE=100Mbps`。

#### WEIGHT

`WEIGHT` 用来设置该类的比重。值越大，表示该类一次处理的数据比例越大。推荐将 `WEIGHT` 指定为 `RATE` 的 1/10。

#### PRIORITIES

`PRIORITIES` 为该类的优先级。可设定为 1 ~ 8。默认值为 5。数字越大，表示优先级越低。`PRIORITIES` 为 8 时 `WEIGHT` 自动变为 1。

#### RULE

`RULE` 指定的是带宽控制对象。可以使用通信对象的 IP 地址或端口号来限定，因此也可以只控制 HTTP 或 FTP 的带宽。格式如下。

`RULE=[[saddr[/prefix]][:port[/mask]],][daddr[/prefix]][:port[/mask]]]`

仅对 192.168.0.100 进行带宽控制时设置如下。

`RULE=192.168.0.100`

仅对 HTTP (端口号 80) 进行控制时设置如下。

`RULE=192.168.0.100:80,`

其他参数或配置的详细内容将在下一节中介绍。

#### TIME

`TIME` 参数在上例中并没有出现。格式可以按照下列方式通过时间或星期来修改带宽 `RATE` 和 `WEIGHT` 的配置。

`TIME=[<dow>,<dow>,...,<dow>/]<from>-<to>,<rate>/<weight>[/<peak>]`

描述如下。

`TIME=18:00-06:00;256Kbit/25Kbit`

## 启动脚本

CBQ 的设置使用 `tc` (traffic control) 命令来进行。但是 `tc` 命令的选项很多，非常复杂。

iproute 工具包中含有读入配置文件并自动执行 tc 命令的启动脚本，这里就使用这个脚本。

这个启动脚本默认读入 /etc/sysconfig/cbq/ 中的 cbq-\* 文件。由于这里已经存在示例文件，因此可以尝试运行脚本。

```
# chmod 777 /usr/share/doc/iproute-2.6.18/examples/cbq.init-v0.7.3
# /usr/share/doc/iproute-2.6.18/examples/cbq.init-v0.7.3 start

**CBQ: failed to compile CBQ configuration!
```

---

小贴士：RHEL6 中有 /etc/sysconfig/cbq/，但是没有 cbq.init 文件。RHEL6 中的 /sbin/cbq 就相当于 cbq.init。

---

在这个示例中进行 CBQ 的设置，就会出错。脚本已运行的命令记录在 /var/cache/cbq.init 中。

```
# cat /var/cache/cbq.init
/sbin/tc qdisc del dev eth0 root
/sbin/tc qdisc add dev eth0 root handle 1 cbq bandwidth 10Mbit avpkt 3000 cell 8
/sbin/tc class change dev eth0 root cbq weight 1Mbit allot 1514

**CBQ: class ID of cbq-0000.example must be in range <0002-FFFF>!
```

最后显示的是错误的详细信息。cbq 的配置文件名为 /etc/sysconfig/cbq/cbq-<类 ID>，类 ID 必须设置 2 以上的数值。

在上例中，对于配置文件由于类 ID 为 0，因此启动失败。根据环境将文件名改为 cbq-2.eth3 再次运行。在 eth3 中将发往 IP 地址 192.168.0.100 的分组通信设置为 100Mbit/sec。

```
# cat /etc/sysconfig/cbq/cbq-2.eth3
DEVICE=eth3,1000Mbit,100Mbit
RATE=100Mbit
WEIGHT=10Mbit
RULE=192.168.0.100
# mv /etc/sysconfig/cbq/cbq-0000.example /etc/sysconfig/cbq/_cbq-0000.example
# /usr/share/doc/iproute-2.6.22/examples/cbq.init-v0.7.3 start
```

设置内容可以使用 tc 命令来确认。

```
# tc -s -d class show dev eth3
```

或者也可以通过 /var/cache/cbq.init 的内容来确认。

```
# cat /var/cache/cbq.init
/sbin/tc qdisc del dev eth3 root
/sbin/tc qdisc add dev eth3 root handle 1 cbq bandwidth 1000Mbit avpkt 3000
cell 8
/sbin/tc class change dev eth3 root cbq weight 100Mbit allot 1514

/sbin/tc class add dev eth3 parent 1: classid 1:2 cbq bandwidth 1000Mbit rate
```

```
100Mbit weight 10Mbit
prio 5 allot 1514
cell 8 maxburst 20 avpkt 3000 bounded
/sbin/tc qdisc add dev eth3 parent 1:2 handle 2 tbf rate 100Mbit buffer 10Kb/8
limit 15Kb mtu 1500
/sbin/tc filter add dev eth3 parent 1:0 protocol ip prio 100 u32 match ip dst
192.168.0.100 classid 1:2
```

使用下列命令也可以确认。

```
# /usr/share/doc/iproute-2.6.22/examples/cbq.init-v0.7.3 list
```

## 确认带宽控制

本节使用 nuttcp 来确认吞吐量。

```
# wget -t0 -c http://www.lcp.nrl.navy.mil/nuttcp/nuttcp-5.5.5.tar.bz2
# tar jxvf nuttcp-5.5.5.tar.bz2
# cd nuttcp-5.5.5
# gcc -O2 -o nuttcp nuttcp-5.5.5.c
```

首先，在 IP 地址 192.168.0.100 (receiver) 一方作为服务器启动 nuttcp。

```
[receiver]# ./nuttcp -s
```

发送 (sender) 方按照下列方式执行 nuttcp。默认为发送数据 10 秒钟，计算吞吐量。

```
[sender]# ./nuttcp 192.168.0.100
0.1685 MB / 10.00 sec = 0.1413 Mbps 0 %TX 0 %RX
```

结果为 141kbps，非常缓慢。停止 CBQ，再次计算，就会得到如下所示的接近以太网物理带宽上限的速度。

```
[sender]# /usr/share/doc/iproute-2.6.22/examples/cbq.init-v0.7.3 stop
[sender]# ./nuttcp 192.168.0.100
1125.2779 MB / 10.03 sec = 941.4133 Mbps 5 %TX 13 %RX
```

使用数据包捕获进行确认时，每隔约 200 毫秒传输一次数据。这是因为 TSO (TCP Segmentation Offload) 与 CBQ 的组合使其成为了无意图的动作。

因此这时需要禁用 TSO。使用 ethtool 命令来确认 TSO 的设置。

```
[sender]# ethtool -k eth3
Offload parameters for eth3:
Cannot get device udp large send offload settings: Operation not supported
rx-checksumming: on
tx-checksumming: on
scatter-gather: on
tcp segmentation offload: on
udp fragmentation offload: off
generic segmentation offload: off
```

TSO 启用

本次使用的 NIC 具有 TSO 功能，默认是启用的。因此需要使用 ethtool 工具将 TSO 功能禁用，再次使用 nuttcp 命令来计算吞吐量。

```
[sender]# ethtool -K eth3 tso off
```

```
[sender]# ethtool -k eth3
Offload parameters for eth3:
Cannot get device udp large send offload settings: Operation not supported
rx-checksumming: on
tx-checksumming: on
scatter-gather: on
tcp segmentation offload: off
udp fragmentation offload: off
generic segmentation offload: off
[sender]# ./nuttcp 192.168.0.100
 92.3214 MB / 10.00 sec = 77.4083 Mbps 0 %TX 3 %RX
```

结果为 77Mbps，与刚才的数值相比，虽然性能得到了改善，但 77Mbps 与设置值 100Mbps 相比，吞吐量明显降低。

这是因为对于千兆以太网的 NIC 来说，TBF 的缓冲区不够大。TBF (Token Bucket Filter) 是 Qdisc (queueing discipline) 之一，用来把通信数据包放入队列，这里不作具体介绍。在 10Mbps 的情况下推荐将缓冲区大小设置为 10kb/8。可以在 CBQ 配置文件的 BUFFER 项目中进行设置。本次为 Gbit，因此设置为 1000kb/8。

```
[sender]# cat /etc/sysconfig/cbq/cbq-2.eth3
DEVICE=eth3,1000Mbit,100Mbit
RATE=100Mbit
WEIGHT=10Mbit
RULE=192.168.0.100
BUFFER=1000kb/8
[sender]# /usr/share/doc/iproute-2.6.22/examples/cbq.init-v0.7.3 restart
[sender]# ./nuttcp 192.168.0.100
 116.0705 MB / 10.01 sec = 97.3100 Mbps 0 %TX 3 %RX
```

这样就可以使用 CBQ 进行正确的带宽控制。

## 小结

本节介绍了使用 CBQ 进行网络带宽控制的方法。通过限制优先级较低的通信的带宽，可以有效地优先执行重要的通信。在 Web 浏览器上设置 CBQ 的 WebCBQ 工具现在已经公开。将文件解压缩，对 Apache 的设置进行一些编辑，就可以从远程设置网络带宽。关于使用 SR-IOV 的带宽控制，请参考 HACK #35。

## 参考文献

- CBQ.init traffic management script  
<http://sourceforge.net/projects/cbqinit/files/cbqinit/>
- WebCBQ Bandwidth Manager  
<http://sourceforge.net/projects/webcbq/>
- Linux Advanced Routing & Traffic Control HOWTO[ 第 9 章队列规则与带宽管理 ]  
<http://linuxjf.sourceforge.jp/JFdocs/Adv-Routing-HOWTO/lartc.qdisc.classful.html>

——Naohiro Ooiwa

# HACK #23 TUN/TAP 设备

本节介绍在 Linux 中实现网络通道的 TUN/TAP 设备。

## TUN/TAP 设备

TUN/TAP 设备是 TUN (Tunnel) 设备和 TAP 设备的总称。Linux 的内核源代码中加入了 Universal TUN/TAP 设备。因此可以很方便地生成 TUN/TAP 设备，实现网络隧道。

这里将简单介绍 TUN 设备和 TAP 设备。用一句话来说，TUN 设备用来实现三层隧道，TAP 设备用来实现二层隧道。

### TUN 设备

这是虚拟的点对点设备，在一般的 TCP/IP 网络中处理的是三层 IP 数据包。在 Linux 中一般体现为用 tunX 表示的点对点的网络设备。

对 TUN 设备 tun0 进行 ifconfig 的输出如下所示。

```
# ifconfig tun0
tun0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          inet addr:192.168.1.1  P-t-P:192.168.1.2  Mask:255.255.255.255
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

### TAP 设备

这是虚拟的以太网设备，在 TCP/IP 网络中处理二层的以太网帧。在 Linux 中一般体现为用 tapX 表示的以太网络设备。

对 TAP 设备 tap0 进行 ifconfig 的输出如下所示。

```
# ifconfig tap0
tap0      Link encap:Ethernet  HWaddr D2:76:E4:DD:E0:F8
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

以太网帧从用户空间的应用程序发送到用于网络隧道的设备——TAP 设备，对操作系统来说就是以太网帧从外部到达这个 TAP 设备。从外部接收的以太网帧经由操作系统的网络栈发送到 TAP 设备，然后由用户空间的应用程序接收。

# 应用程序示例

## VPN (Virtual Private Network)

在使用 TUN/TAP 设备的应用程序中，VPN 是最常见的。例如，实际安装的 VPN 之一 Vtun，就是本章介绍的 Universal TUN/TAP 驱动程序的母程序。此外还有 OpenVPN 等。TUN/TAP 设备对于 VPN 可以说是必需的。

## 虚拟机的网络连接

在 Linux 的虚拟结构 KVM 中所使用的 qemu，在生成虚拟网络时使用 TAP。如果没有 TAP 设备，将虚拟机的 NIC 桥接到物理 NIC 的结构是无法实现的。

## 使用 TUN/TAP 设备的程序设计示例

这里简单编写使用 TUN/TAP 设备的样本程序，并确认其运行结果。

生成点对点连接的 TUN 设备，对 ICMP 的 ECHO 进行处理，来确认 TUN 设备运行的情况。示例程序为 tunpong.c，是在只有 ICMP 的 ECHO REQUEST 进入的前提下生成的。这个程序只是为了查看 TUN 设备的使用方法，不包括错误处理等。

### 例 4-1 tunpong.c

```
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int tun_open(void)
{
    struct ifreq ifr;
    int fd;
    char dev[IFNAMSIZ];
    char buf[512];
    /* 打开用来操作 TUN/TAP 设备的文件 */
    fd = open("/dev/net/tun", O_RDWR);

    /* 生成 TUN 设备 (tun0) */
    memset(&ifr, 0, sizeof(ifr));
    ifr.ifr_flags = IFF_TUN | IFF_NO_PI;
    strncpy(ifr.ifr_name, "tun%d", IFNAMSIZ);
    ioctl(fd, TUNSETIFF, &ifr);

    strncpy(dev, ifr.ifr_name, IFNAMSIZ);
    /* 使用 ifconfig 命令添加 IP 地址 */
    sprintf(buf, "ifconfig %s 192.168.1.1 pointopoint 192.168.1.2", dev);
    system(buf);
```

```

        return fd;
    }

void dump_pkt(unsigned char *pkt, int len)
{
    int i;

    for (i = 0; i < len; i++)
        printf("%02x ", pkt[i]);
    printf(" \n");
}

void pingpong(int fd)
{
    fd_set fds;
    int len;
    unsigned char pkt[512];

    FD_ZERO(&fds);
    FD_SET(fd, &fds);

    select(fd+1, &fds, NULL, NULL, NULL);
    if(FD_ISSET(fd, &fds)) {
        len = read(fd, pkt, 512);
        dump_pkt(pkt, len);
        /* ICMP 头部分为 0x08 (echo request) */
        if (pkt[20] != 0x08)
            return;

        /* 将 IP 头的 src 设为 192.168.1.2, dst 设为 192.168.1.1 */
        pkt[15] = 0x02;
        pkt[19] = 0x01;
        pkt[20] = 0x00; /* echo reply */
        write(fd, pkt, len);
    }
}

int main(int argc, char **argv)
{
    int fd;

    fd = tun_open();
    for (;;) {
        pingpong(fd);
    }
    return 0;
}

```

使用下列命令行来进行示例程序的编译。

```
# gcc -g -O2 -o tunpong tunpong.c
```

对 TUN 设备进行操作需要有 root 权限。请首先成为 root 用户再尝试运行示例程序 tunpong。

```
# ./tunpong
```

在运行后打开另一个终端，使用 ifconfig 命令来确认。通过示例程序内部运行的 ifconfig，可以得知 IP 地址 192.168.1.1 和点对点的 192.168.1.2 被分配到 TUN 设备 (tun0)。

```
# ifconfig tun0
tun0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
```

```
inet addr:192.168.1.1 P-t-P:192.168.1.2 Mask:255.255.255.255
UP POINTOPOINT RUNNING NOARP MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:500
RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)
```

接下来，执行 ping 命令，尝试将 ICMP ECHO REQUEST 发送到 192.168.1.2。可以确认，ping 命令返回如下 REPLY。

```
$ ping 192.168.1.2
PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.
64 bytes from 192.168.1.2: icmp_req=1 ttl=64 time=0.100 ms
64 bytes from 192.168.1.2: icmp_req=2 ttl=64 time=0.248 ms
64 bytes from 192.168.1.2: icmp_req=3 ttl=64 time=0.207 ms
```

另外，还可以看到在 tunpong 的标准输出中输出了发送到 192.168.1.2 的数据包 (ICMP ECHO REQUEST) 的转储结果，程序 tunpong 通过 TUN 设备接收数据包。

```
# ./tunpong
45 00 00 54 00 00 40 00 40 01 b7 55 c0 a8 01 01 c0 a8 01 02 08 00 e6 31 1a 48
00 01 82 8f 06 4e 00 00 00 a2 d4 0d 00 00 00 00 00 10 11 12 13 14 15 16 17
18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31
32 33 34 35 36 37
45 00 00 54 00 00 40 00 40 01 b7 55 c0 a8 01 01 c0 a8 01 02 08 00 9a 30 1a 48
00 02 83 8f 06 4e 00 00 00 00 ed d4 0d 00 00 00 00 00 10 11 12 13 14 15 16 17
18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31
32 33 34 35 36 37
45 00 00 54 00 00 40 00 40 01 b7 55 c0 a8 01 01 c0 a8 01 02 08 00 68 30 1a 48
00 03 84 8f 06 4e 00 00 00 00 1e d4 0d 00 00 00 00 00 10 11 12 13 14 15 16 17
18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31
32 33 34 35 36 37
```

最后使用 ifconfig 命令确认统计信息。RX 和 TX 的 packets 和 bytes 同时增加，可以看出成功进行了发送和接收。

```
# ifconfig tun0
tun0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          inet addr:192.168.1.1 P-t-P:192.168.1.2 Mask:255.255.255.255
          UP POINTOPOINT RUNNING NOARP MULTICAST MTU:1500 Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:252 (252.0 b)  TX bytes:252 (252.0 b)
```

## 小结

本节介绍了 TUN/TAP 设备。实现 Linux 网络隧道的 TUN/TAP 设备，除了用于需要 VPN 等网络隧道的应用程序以外，最近也多用于实现虚拟环境中的虚拟网络，是非常重要的设备。

——Hiroshi Shimamoto

## HACK #24 网桥设备

介绍 Linux 内核中网桥设备的生成和网桥连接。

Linux 内核中安装有网桥功能。使用网桥功能，就可以将多个网络接口连接到一个网段上。如图 4-1 所示将多个网络接口连接到 Linux 机器的网桥接口上，就可以将各网段整理为一个网段。例如，图 4-1 中是将 4 个网络接口连接到一个网桥上。这个 Linux 机器拥有 4 个网络接口 (eth0~eth3)，eth0 连接到路由器，eth1~eth3 分别连接到不同的客户端。

物理上分为 4 个网段，而通过使用 Linux 内核的网桥功能，就可以整合为一个网段。也就是说，启用了网桥功能的 Linux 机器等同于交换式集线器 (switching hub)。

Linux 内核将 IP 地址分配到网桥接口 br0。不会向连接到网桥的网络接口 (图 4-1 中的 eth0~eth4) 分配 IP 地址。

最近开始越来越多地将网桥功能运用在虚拟环境中。在虚拟环境中是将 TAP 设备和物理 NIC 进行网桥连接。

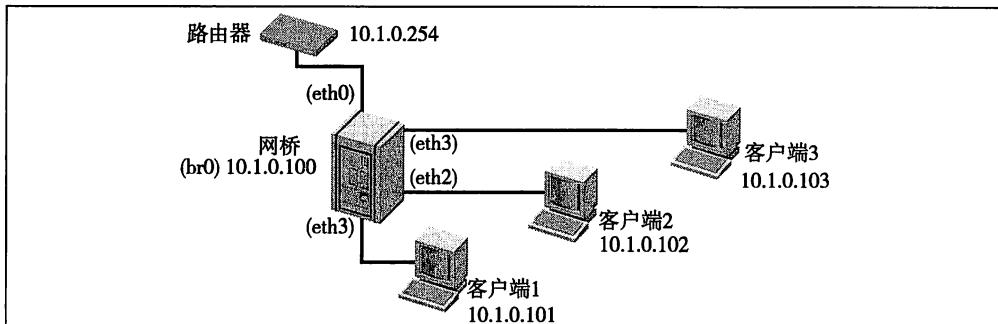


图 4-1 网桥连接

## brctl 命令

bridge-utils 工具包中包含对网桥进行操作的命令 brctl。可以按下列方式安装。

RedHat 系列

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

Debian 系列

```
# apt-get install bridge-utils
```

在已经安装虚拟功能的情况下，对网桥进行操作的命令也应该同时安装了。

常用的命令如表 4-1 所示。

表 4-1 网桥中经常使用的命令

名 称	功 能
brctl show [ 网桥名 ]	显示网桥列表与状态
brctl addbr 网桥名	生成网桥

名 称	功 能
brctl addif 网桥名 接口名	将接口连接到网桥网络
brctl delif 网桥名 接口名	拆除连接到网桥的网络接口
brctl delbr 网桥名	删除网桥

## 使用网桥功能的示例

进行图 4-1 的网桥连接时的操作如下。如果在经由作为设置对象的网络接口之一连接 (ssh 等) 的状态下实施这个作业, 需要注意有时会由于网络接口 IP 地址更换的时机等而无法进行网络断开的操作。

1. 生成网桥设备 br0。

```
# brctl addbr br0
```

2. 向网桥设备分配 IP 地址。

```
# ifconfig br0 10.1.0.100 netmask 255.255.255.0 broadcast 10.1.0.255 up
```

3. 将 eth0 连接到网桥 br0。

```
# brctl addif br0 eth0
```

4. 将 eth0 激活。

```
# ifconfig eth0 up
```

5. 同样对 eth1~eth3 进行操作 3、4。

```
# brctl addif br0 eth1
# ifconfig eth1 up
# brctl addif br0 eth2
# ifconfig eth2 up
# brctl addif br0 eth3
# ifconfig eth3 up
```

通过以上操作, 网络接口的网桥连接就完成了。brctl show 显示的网桥状态如下所示。

```
# brctl show
bridge name      bridge id      STP enabled      interfaces
br0              8000.545200006400    no            eth0
                                         eth1
                                         eth2
                                         eth3
```

## 网桥的设置

可以设置为操作系统启动时自动生成网桥。每个发布版中的设置方法都有所不同, 这里将介绍 Debian 系列和 RedHat 系列。

Debian 系列的情况在 `/etc/network/interfaces` 中描述。设置 `static address 0.0.0.0`, 使得不会对连接到网桥的接口本身设置 IP 地址。

#### 例 4-2 `/etc/network/interfaces`

```
auto eth0
iface eth0 inet static
    address 0.0.0.0

auto eth1
iface eth1 inet static
    address 0.0.0.0

auto eth2
iface eth2 inet static
    address 0.0.0.0

auto eth3
iface eth3 inet static
    address 0.0.0.0

auto br0
iface br0 inet dhcp
    bridge_ports eth0 eth1 eth2 eth3
    bridge_stp off
```

## RedHat 系列的情况

在目录 `/etc/sysconfig/network-scripts` 下配置 `ifcfg-` (网桥名)。

#### 例 4-3 `ifcfg-br0`

```
DEVICE=br0
TYPE=Bridge
BOOTPROTO=dhcp
ONBOOT=yes
STP=off
```

对与连接到网桥的网络接口相对应的 `ifcfg-` (接口名) 进行编辑, 设置为使用网桥。另外设置为 `NM_CONTROLLED=no`, 就可以将其排除在 NetworkManager 的管理范围之外。

#### 例 4-4 `ifcfg-eth0`

```
DEVICE=eth0
HWADDR=**:**:**:**:**:***
NM_CONTROLLED=no
ONBOOT=yes
BRIDGE=br0
```

## 虚拟机的网桥连接

在启动 `qemu` 的情况下, 使用 `-net` 选项指定 `tap` 启动, `TAP` 设备生成和删除时就会执行脚本。执行的脚本默认为 `qemu-ifup.sh`, 但可以使用选项来修改。通过这个脚本来向

TAP 设备的网桥进行连接。具体来说，就是在生成 TAP 设备的同时，会以这个 TAP 设备名为变量调出脚本。

```
# qemu-ifup.sh tap 设备名
```

qemu-ifup.sh 的内容如下。

```
#!/bin/sh
```

```
TAP=$1
```

```
brctl addbr br0 $TAP
ifconfig $TAP up
```

使用虚拟管理库的 libvirt，就可以在虚拟机启动时自动进行网桥连接。

## 小结

Linux 内核中安装有网桥功能，通过使用网桥功能，可以将多个网段整合为一个网段。Linux 机器通过这个功能，就能够像交换式集线器一样运行。此外，网桥功能还可以在 KVM 等虚拟环境中，将虚拟机的网络连接到物理网络。

——Hiroshi Shimamoto

## HACK #25 VLAN

本节介绍 VLAN (Virtual LAN) 的设置方法。

在 Linux 中安装了 802.1Q 标签 VLAN 功能。VLAN 是虚拟分配以太网的功能。使用 VLAN ID 从物理上将一个以太网分割开。在 VLAN 环境下，具有相同 VLAN ID 就可以相互通信，但是即使将 LAN 线连接到相同集线器或交换机上，VLAN ID 不同也不能相互通信。

本节将介绍 VLAN 的设置方法，包括准备配置文件的方法和使用命令行设置的方法。

### 使用命令进行设置

使用 vconfig 命令和 ip 命令可以进行 VLAN 环境的设置。

使用 vconfig 命令生成 VLAN 接口时的操作如下。把 VLAN ID 设置为 100。

```
# vconfig add eth0 100
Added VLAN with VID == 100 to IF -:eth0:-
```

使用 ip 命令也可以进行同样的设置。

```
# ip link add link eth0 name eth0.100 type vlan id 100
```

NIC eth0 中将生成 VLAN ID100 的 VLAN 接口。与一般的网络接口同样可以设置 IP 地址等。

```

# ifconfig eth0.100 192.168.1.100
# ifconfig eth0.100
eth0.100  Link encap:Ethernet  HWaddr 00:1B:21:0F:91:8F
          inet addr:192.168.1.100  Bcast:192.168.1.255  Mask:255.255.255.0
                         UP BROADCAST RUNNING SLAVE MULTICAST  MTU:1500  Metric:1
.....

```

生成的 VLAN 接口的状态可以使用 /proc/net/vlan 来确认。

```

# ls /proc/net/vlan/
config  eth0.100
# head /proc/net/vlan/*
==> /proc/net/vlan/config <==
VLAN Dev name  | VLAN ID
Name-Type: VLAN_NAME_TYPE_RAW_PLUS_VID_NO_PAD
eth0.100       | 100  | eth0

==> /proc/net/vlan/eth0.100 <==
eth0.100  VID: 100      REORDER_HDR: 1  dev->priv_flags: 1
          total frames received          0
          total bytes received          0
          Broadcast/Multicast Rcvd      0

          total frames transmitted      0
          total bytes transmitted      0
          total headroom inc          0
          total encap on xmit          0
Device: eth0

```

要删除 VLAN 接口时，可以执行下列命令。

```
# vconfig rem eth0.100
```

或者

```
# ip link delete eth0.100
```

在这个示例中，生成的 VLAN 接口的名称是物理网络接口 + 点 +VLAN ID（即 eth0.<VLAN ID>），但 VLAN 的接口名有 4 种，如表 4-2 所示。

表 4-2 VLAN 接口名的种类

种类	VLAN 接口名的示例	每个种类的命名规则
VLAN_PLUS_VID	vlan0005	字符串 vlan+0 padding（多个 0 叠加）+VLAN ID
VLAN_PLUS_VID_NO_PAD	vlan5	字符串 vlan+VLAN ID
DEV_PLUS_VID	eth0.0005	物理网络接口名 + 点 +0 padding+ VLAN ID
DEV_PLUS_VID_NO_PAD	eth0.5	物理网络接口名 + 点 +VLAN ID

使用 vconfig add 增加 VLAN 接口时，需要事先使用 vconfig set\_name\_type<种类> 来设置种类。设置了种类后，内核就会按照种类从 VLAN 接口名来识别 ID。

但是在已有配置文件中使用 ifup 命令的情况，是通过 ifup 脚本来识别 VLAN ID 的，因此无须在意 VLAN 设备名的种类。

## 使用设置文件进行设置

在 `etc/sysconfig/network-scripts/` 下准备好 `eth0.100` 用的配置文件，系统启动时就可以自动生成 VLAN 接口。请设置 `DEVICE= VLAN` 接口名，并设置为 `VLAN=yes`。

```
# cat ifcfg-eth0.100
DEVICE=eth0.100
TYPE=Ethernet
BOOTPROTO=static
IPADDR=192.168.1.100
ONBOOT=yes
VLAN=yes
```

`VLAN_PLUS_VID`、`VLAN_PLUS_VID_NO_PAD` 的 VLAN 接口需要设置为 `VLAN=yes`，并在 `PHYSDEV=` 后面写入物理网络接口名。

```
# cat ifcfg-vlan002
DEVICE=vlan002
TYPE=Ethernet
BOOTPROTO=static
IPADDR=192.168.1.102
ONBOOT=yes
VLAN=yes
PHYSDEV=eth0
```

准备 `eth0.001` 用和 `vlan3` 用的配置文件，然后执行 ifup 命令，内核中就能够正常识别这些 VLAN 接口。

```
# cat ifcfg-eth0.001
DEVICE=eth0.001
...
VLAN=yes
# cat ifcfg-vlan3
DEVICE=vlan3
...
VLAN=yes
PHYSDEV=eth0
# ifup eth0.001
# ifup vlan002
# ifup vlan3
# cat /proc/net/vlan/config
VLAN Dev name | VLAN ID
Name-Type: VLAN_NAME_TYPE_RAW_PLUS_VID_NO_PAD
eth0.001 | 1 | eth0
vlan002 | 2 | eth0
vlan3 | 3 | eth0
# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:1B:21:0F:91:8F
```

```

        inet addr:192.168.3.100 Bcast:192.168.3.255 Mask:255.255.255.0
        UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
...
eth0.001 Link encap:Ethernet HWaddr 00:1B:21:0F:91:8F
        inet addr:192.168.1.101 Bcast:192.168.1.255 Mask:255.255.255.0
        UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
...
vlan002 Link encap:Ethernet HWaddr 00:1B:21:0F:91:8F
        inet addr:192.168.1.102 Bcast:192.168.1.255 Mask:255.255.255.0
        UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
...
vlan3 Link encap:Ethernet HWaddr 00:1B:21:0F:91:8F
        inet addr:192.168.1.103 Bcast:192.168.1.255 Mask:255.255.255.0
        UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
...

```

## MAC-VLAN

从 2.6.23 版本开始就具有了 MAC-VLAN。在 2.6.38 版本中还是试验性 (EXPERIMENTAL) 的驱动程序。使用时需要启用内核 config 文件 CONFIG\_MACVLAN。在 RHEL6 中默认是编译为模块的。要生成 MAC-VLAN 接口，可以使用下列 ip 命令 (对每个 MAC 地址指定分配的 VLAN 接口)。MAC-VLAN 驱动程序 macvlan 自动安装到内核中，因此可以省略 modprobe 命令。

```

# modprobe macvlan
# lsmod | grep macvlan
macvlan           9727  0
# ip link add link eth0 name eth0.100 address 00:aa:bb:cc:dd:ee type macvlan
# ifconfig eth0.100
eth0.100 Link encap:Ethernet HWaddr 00:AA:BB:CC:DD:EE
          BROADCAST MULTICAST MTU:1500 Metric:1
.....

```

## 参考文献

- initscripts 数据包文档  
[/usr/share/doc/initscripts-\\*/sysconfig.txt](/usr/share/doc/initscripts-*/sysconfig.txt)
- 802.1Q VLAN implementation for Linux  
<http://www.candelatech.com/~greear/vlan.html>

——Naohiro Ooiwa

## HACK #26 bonding 驱动程序

本节介绍使用 bonding 驱动程序将多个物理网络设备绑定方法。

bonding 是将多个物理网络设备绑定的驱动程序。绑定 bonding 说法来自于黏合剂 bond。绑定的运行根据模式不同而有所差异。Linux 的 bonding 有 7 种模式，本节将介绍的是

普遍较常使用的激活备份模式。

本节中所举的示例是将物理网络设备 eth0 和 eth1 绑定，构成 bonding 设备 bond0 的环境（见图 4-2）。

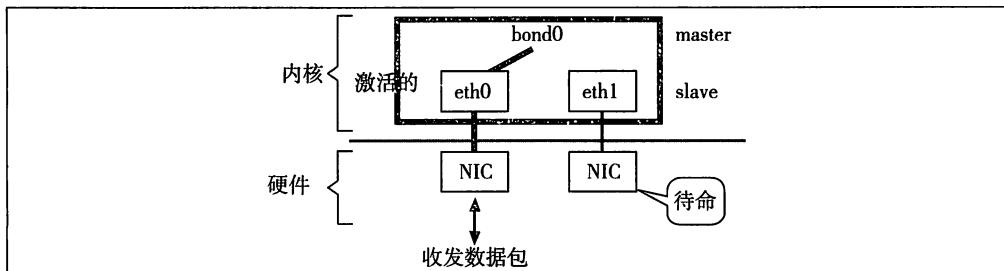


图 4-2 本节的 bonding 构成（激活备份模式）

构成 bond0 的 eth0/eth1 称为从设备（slave）。bond0 称为主设备（master）。激活备份模式时，激活的 slave eth0 进行通信。eth1 为备份，处于待命状态。

## 使用方法

Linux 的 bonding 有 7 种模式。下面先简单地介绍各种模式。模式的名称就是直接指定到模块参数的字符串。也可以使用 () 中的数字来指定模式。

`balance-rr(0)`

通过轮询（round robin）来分散负载。发送数据时，第一个数据从 eth0 发送，第二个数据从 eth1 发送。依次由 eth0/eth1 发送。

`active-backup(1)`

多个从设备中的一个运行。这个从设备称为激活的从设备。如果激活从设备出现故障，断开链接（link down）时，则切换为其他从设备。这个模式称为激活备份模式。

`balance-xor(2)`

通过 MAC 地址的异或结果（XOR）来决定从设备。从设备由散列队列（hashlist）进行管理，根据使用某计算公式求得的散列值来决定从设备。计算出散列值的方法称为散列策略（hash policy）。balance-xor 模式的计算公式为 ((发送方 MAC 地址 xor 接收方 MAC 地址) / 从设备数)。将得到的余数作为散列值。由于使用 Ether 头的 MAC 地址，因此将这个策略称为 layer2。可以在 `xmit_hash_policy` 模块选项中更改散列策略。除 layer2 以外，还有 layer2+3 和 layer3+4。layer2+3 是通过 MAC 地址和 IP 地址的组合来计算出散列值。layer3+4 是通过 MAC 地址、IP 地址和 TCP、UDP 的端口号的组合来算出散列值。在这个模式下，只要上述条件不改变，就会使用相同从设备。

```
broadcast(4)
```

已连接的所有从设备发送相同数据。即使某个从设备发送失败也不会中断，而是从其他从设备发送数据。

此外还有 802.3ad(4)、balance-tlb(5)、balance-alb(6)。详细情况请看参考文献。

默认为轮询。使用激活备份模式时需要按下列方法将模块安装到内核中。

```
# modprobe bonding mode=active-backup
```

或者执行下列命令。

```
# modprobe bonding mode=1
```

一般在准备其他模块参数时同时准备接口的配置文件。这样在启动时 bonding 接口就会启用。RHEL6 中不是在 modprobe.conf 中而是在 /etc/modprobe.d/\*.conf 中设置 bonding 模块的别名 (alias)。这里的文件名设置为 bond.conf。

```
# cat /etc/modprobe.d/bond.conf
alias bond0 bonding
alias bond1 bonding
```

接口的配置文件如下。

```
# cat /etc/sysconfig/network-scripts/ifcfg-eth0
DEVICE="eth0"
NM_CONTROLLED="no"
ONBOOT=no
HWADDR=AA:AA:AA:AA:AA:AA
TYPE=Ethernet
BOOTPROTO=none

MASTER=bond0
SLAVE=yes

# cat /etc/sysconfig/network-scripts/ifcfg-eth1
DEVICE="eth1"
NM_CONTROLLED="no"
ONBOOT=no
HWADDR=BB:BB:BB:BB:BB:BB
TYPE=Ethernet
BOOTPROTO=none

MASTER=bond0
SLAVE=yes

# cat /etc/sysconfig/network-scripts/ifcfg-bond0
DEVICE=bond0
ONBOOT=yes

BOOTPROTO=static
IPADDR=192.168.0.10
NETMASK=255.255.255.0
NETWORK=192.168.0.0
```

```
BROADCAST=192.168.0.255
```

```
BONDING_OPTS="mode=1 miimon=100 primary=eth0"
```

上例是在 `ifcfg-bond0` 中记载的 `BONDING_OPTS`。这是用来设置模块选项的。通过写入 `ifcfg-bond0` 文件，对每个 `bonding` 接口设置选项。因此在 RHEL6 并不推荐在 `/etc/modprobe.d/bond.conf` 下使用 `option` 来记载 `bonding` 模块的选项。在配置文件里记载后，就可以使用 `ifup bond0` 来启动 `bonding` 设备。成为从设备的接口可以使用 `ifconfig` 命令确认。

```
# ifconfig eth0
eth0      Link encap:Ethernet HWaddr 00:1B:21:0F:91:8F
          UP BROADCAST SLAVE MULTICAST MTU:1500 Metric:1      有 SLAVE
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
          Interrupt:16 Memory:efbc0000-efbe0000
```

`bonding` 接口的设置内容可以在下列 `proc` 文件系统中确认。

```
# cat /proc/net/bonding/bond0
Ethernet Channel Bonding Driver: v3.7.0 (June 2, 2010)

Bonding Mode: fault-tolerance (active-backup)
Primary Slave: eth2 (primary_reselect always)
Currently Active Slave: eth0          激活 slave
MII Status: down
MII Polling Interval (ms): 100
Up Delay (ms): 0
Down Delay (ms): 0

Slave Interface: eth0
MII Status: down
Speed: 100 Mbps
Duplex: full
Link Failure Count: 0
Permanent HW addr: aa:aa:aa:aa:aa:aa  原始的 MAC 地址
Slave queue ID: 0

Slave Interface: eth1
MII Status: down
Speed: 100 Mbps
Duplex: full
Link Failure Count: 0
Permanent HW addr: bb:bb:bb:bb:bb:bb
Slave queue ID: 0
```

## 关于激活备份模式

激活备份模式在激活从设备发生连接障碍时，就会切换到其他从设备。切换到的从设备就会作为激活从设备运行。障碍是指线缆或 NIC 发生问题，连接中断。

拔掉激活从设备的 LAN 线缆，或执行 `ifdown` 命令，就可以确认切换的运行。

当前激活的从设备可以在 `/proc/net/bonding/bond0` 中确认。`eth0` 第一次成为激活从设备时，`bond0` 的 MAC 地址设置为 `eth0` 的 MAC 地址。即使切换为 `eth1`，`bond0` 的 MAC 地址也不会变化。由于这样的构造，使得从外部看起来 `bond0` 是一个端口。

---

**小贴士：**从 Linux2.6.24 开始安装了 `fail_over_mac` 选项，可以更改决定 `bonding` 接口的 MAC 地址的策略。

---

使用 `ifconfig` 命令可以确认当前的 `bonding` 接口和从设备的 MAC 地址。接口的原始 MAC 地址可以在 `/proc/net/bonding/bond0` 中确认。

下面将介绍在激活备份模式下使用的主要模块选项。

### **miimon**

激活备份模式的连接障碍可以通过定期监控检查出来。`miimon` 设置的是通过 MII(Media Independent Interface) 监控时的周期。单位为毫秒，初始设置值为 100 毫秒。使用 MII 监视时 NIC 必须和 MII 相对应。

NIC 是否对应可以使用 `mii-tool` 命令来确认。如果与 MII 相对应，则会输出如下结果。

```
# mii-tool
eth0: negotiated 100baseTx-FD flow-control, link ok
eth1: negotiated 100baseTx-FD flow-control, link ok
```

在不与 MII 对应的 NIC 下使用 `bonding` 的激活备份模式时，使用 ARP 监控。在模块选项中设置 `arp_interval`、`arp_ip_target` 等。这种情况下必须设置使用 `arp_ip_target` 监控的 IP 地址，监控过程中一直传输的是 ARP 数据包。

### **updelay**

`updelay` 选项在检测出从设备的连接后，不会立刻连接，而是会等待 `updelay` 所指定的时间。这个选项以毫秒为单位来设置。

这个选项在 MII 连接监控中有效。推荐设置为 `miimon` 数值的倍数。

`updelay=100` (单位为毫秒)

从 Linux 2.6.31 开始，即使指定 `updelay`，最先输入的从设备也无视 `updelay`。

操作系统启动后，`bond0` 被 `ifup`，并按照 `eth0`、`eth1` 的顺序注册为 `slave`。`eth0` 是最先注册的从设备，因此无视 `updelay`，立刻被 `ifup`。由于 `slave` (`eth0`) 已经注册，接下来的 `eth1` 就需要等待 `updelay` 的时间后再 `ifup`。

### **primary**

这个选项用来设置要作为激活从设备的接口。如果设 `primary=eth0`，`eth0` 就设置为

激活从设备。例如，假设为了产生连接问题而执行 `ifdown eth0`。连接会断开，因此激活从设备会切换到 `eth1`。接下来执行 `ifup eth0` 命令，使 `eth0` 恢复。未设置 `primary` 模块选项时，激活从设备仍然是 `eth1`。设置 `primary=eth0` 时，在 `ifup` 的时候激活 `slave` 切换为 `eth0`。

## 参考文献

- Linux Ethernet Bonding Driver HOWTO  
[Documentation/networking/bonding.txt](#)

——Naohiro Ooiwa

## HACK #27 Network Drop Monitor

本节介绍 `dropwatch` 的使用方法，并以 UDP 为例介绍网络的调整方法。

从 Linux 2.6.30 开始具有 Network Drop Monitor（`dropwatch`）功能。

`dropwatch` 是监控内核的网络栈丢弃的数据包。

接收数据包后，数据从网络驱动程序传输到内核的网络层。在这里进行校验和或 IP 地址的确认等数据包验证处理。在这个过程中，如果是非法数据包，则依据 RFC 丢弃，即使是合法的数据包，在超过接收缓冲区大小等情况也会有意识地丢弃。

由于上述这些理由，接收数据包在内核内丢弃。这些不会在日志等中输出，因此一般难以发现。

如果是非法数据包，那么丢弃也没问题，但如果内存有空闲却因接收缓冲区较小而在网络负载较高时丢弃大量数据包，就需要通过调整来进行改善。

本节以 UDP 的网络负载为例，使用 `dropwatch` 来确认丢弃数据包的情况。使用 `proc` 文件系统调整接收缓冲区的大小，并使用 `dropwatch` 来确认调整结果。使用的操作系统为 RHEL6。

### dropwatch 的使用方法

使用上游内核时，需要启用 `Network packet drop alerting service (NET_DROP_MONITOR=y)` 并编译内核。

`dropwatch` 是使用 Kernel Tracepoint API 安装的。具体来说，就是各协议释放 socket 缓冲区时（`kfree_skb` 函数）收集信息。可以应对 IP、ARP、ICMP、IGMP、UDP 协议的数据包等。

因此，即使不使用 `proc` 文件系统或工具针对各个协议收集信息，也可以通过 `dropwatch` 将数据集中到一起进行确认。

dropwatch 是作为内核功能使用，因此不需要为了调查潜在的网络性能而改变已有的应用程序。便于使用也是其优点之一。

下面开始实际使用 dropwatch。无须进行特别的设置。执行 dropwatch 命令，显示出 dropwatch 和提示符后，输入 start。

```
# dropwatch
Initializing null lookup method
dropwatch> start                                <---- 输入 start
Enabling monitoring...
Kernel monitoring activated.
Issue Ctrl-C to stop monitoring
1 drops at location 0xffffffff81440849
16 drops at location 0xffffffff81440849
4 drops at location 0xffffffff81440849
4 drops at location 0xffffffff81440849
4 drops at location 0xffffffff81440849
1 drops at location 0xffffffff8149a8ed
1 drops at location 0xffffffff8149a8ed
....
```

使用 Ctrl+C 返回提示符。输入 stop 或 exit 就可以结束。最左边的数值就是废弃的数据包数。location 后面的数值是内核内的地址。在 -l 选项中使用 /proc/kallsyms，将地址转换成函数名输出。按照下列方法执行 -l 命令。kas 为 kernel all symbols 的缩写。

```
# dropwatch -l kas
Initializing kallsyms db
dropwatch> start
Enabling monitoring...
Kernel monitoring activated.
Issue Ctrl-C to stop monitoring
12 drops at ip_rcv_finish+199      --- ①
2 drops at ip_forward+288
1 drops at ip_forward+288
5 drops at netlink_broadcast+180
4 drops at ip_rcv_finish+199
1 drops at igmp_rcv+cb
8 drops at ip_rcv_finish+199
1 drops at __brk_limit+1e8a3284
....
```

①表示在 ip\_rcv\_finish 函数的偏移量  $0 \times 199$  中检测出废弃了 12 个数据包。这个信息为每隔 1 秒输出一次，或者在废弃的数据包数达到 64 时输出。在当前已安装的版本中，这些最小值是不能改变的。

ip\_rcv\_finish 函数的偏移量  $0 \times 194$  是下列代码中的 ①，实际检测出的内容为上一行的 ip\_rcv\_finish+0×194 <kfree\_skb>。

例 4-5 <内核 2.6.32-71.el6.X 86\_64>

```
0xffffffff8144082d <ip_rcv_finish+0x17d>:      mov      0xe0(%rax),%rax
```

```

0xfffffff81440834 <ip_rcv_finish+0x184>: add
-0x7e767c40(%rdx,8),%rax
0xffffffff8144083c <ip_rcv_finish+0x18c>: addq   $0x1,0x40(%rax)
0xffffffff81440841 <ip_rcv_finish+0x191>: mov    %rbx,%rdi
0xffffffff81440844 <ip_rcv_finish+0x194>: callq  0xffffffff814405560
<kfree_skb>
0xffffffff81440849 <ip_rcv_finish+0x199>: mov    $0x1,%eax ---①
0xffffffff8144084e <ip_rcv_finish+0x19e>: jmp    0xffffffff814407dd <ip_
rcv_finish+0x12d>
0xffffffff81440850 <ip_rcv_finish+0x1a0>: mov    0xcc(%rbx),%ecx

```

## 具体调整的例

在大量发送 UDP 数据包的测试程序 `udp-stress.c` 中进行具体的验证。

### 例 4-6 `<udp-stress.c>`

```

#include <stdio.h>
#include <string.h>
#include <arpa/inet.h>
#include <unistd.h>

#define DADDR "192.168.0.112"
#define PORT 5000

int
main(int argc, char **argv)
{
    int sock=0;
    struct sockaddr_in dest;
    char tmpbuf[] = {"stress"};

    sock = socket(PF_INET, SOCK_DGRAM, 0);
    if (!sock) {
        perror("socket");
        return 1;
    }

    dest.sin_family = AF_INET;
    dest.sin_port = htons(PORT);
    dest.sin_addr.s_addr = inet_addr(DADDR);

    if (connect(sock, (struct sockaddr *)&dest, sizeof(dest)) == -1) {
        perror("connect");
        return 1;
    }

    while (1)
        sendto(sock, tmpbuf, strlen(tmpbuf) + 1, 0,
               (struct sockaddr *)&dest, sizeof(dest));

    close(sock);
    return 0;
}

```

编译 `udp-stress.c`。

```
# gcc -Wall udp-stress.c -o udp-stress
```

在接收 UDP 数据包的机器 (RHEL6) 上执行 `nc` 命令。`nc` 命令只是接收 UDP 的数据包并显示数据的程序。

```
# nc -u -l 192.168.0.112 5000
```

对执行 `nc` 命令的服务器机器执行 `udp-stress`，并发送 UDP 数据包。在服务器机器上执行 `dropwatch`，在笔者的环境下首先会输出如下内容，并未能进行 UDP 通信。

```
64 drops at nf_hook_slow+e0
14 drops at ip_rcv_finish+176
64 drops at nf_hook_slow+e0
2 drops at ip_rcv_finish+176
64 drops at nf_hook_slow+e0
64 drops at nf_hook_slow+e0
8 drops at ip_rcv_finish+176
64 drops at nf_hook_slow+e0
64 drops at nf_hook_slow+e0
1 drops at unix_stream_recvmsg+2f3
6 drops at ip_rcv_finish+176
64 drops at nf_hook_slow+e0
```

`nf_hook_slow` 是 `netfilter` 的函数。这表示根据 DROP 的规则已废弃。

下面暂时删除所有 `iptables` 的规则。

```
# iptables -F
```

这样就不会输出 `nf_hook_slow+e0`。也就是说，由于删除了 `netfilter` 的规则，因此 `nf_hook_slow+e0` 不存在数据包丢失。

这样就可以进行 UDP 通信，因此再次执行 `dropwatch`。

```
2 drops at ip_forward+288
6 drops at ip_rcv_finish+199
2 drops at ip_rcv_finish+199
1 drops at ip_forward+288
1 drops at unix_stream_recvmsg+30d
1 drops at unix_stream_recvmsg+30d
4 drops at ip_rcv_finish+199
6 drops at ip_rcv_finish+199
2 drops at unix_stream_recvmsg+30d
14 drops at ip_rcv_finish+199
64 drops at __udp_queue_rcv_skb+79      <--- 执行 udp-stress 的时刻
1 drops at unix_stream_recvmsg+30d
64 drops at __udp_queue_rcv_skb+79
4 drops at ip_rcv_finish+199
64 drops at __udp_queue_rcv_skb+79
64 drops at __udp_queue_rcv_skb+79
64 drops at __udp_queue_rcv_skb+79
64 drops at __udp_queue_rcv_skb+79
```

```
64 drops at __udp_queue_rcv_skb+79
64 drops at __udp_queue_rcv_skb+79
1 drops at igmp_rcv+cb
64 drops at __udp_queue_rcv_skb+79
```

从执行 `udp-stressd` 的时刻开始, 输出变成 `64 drops at __udp_queue_rcv_skb+79`。这就是下列代码中的 ②。

```
0xffffffff81467691 <__udp_queue_rcv_skb+0x61>: movslq %edx,%rdx
0xffffffff81467694 <__udp_queue_rcv_skb+0x64>: mov %r12,%rdi
0xffffffff81467697 <__udp_queue_rcv_skb+0x67>: add -0x7e767c40(%rdx,8),%rax
0xffffffff8146769f <__udp_queue_rcv_skb+0x6f>: addq $0x1,0x18(%rax)
0xffffffff814676a4 <__udp_queue_rcv_skb+0x74>: callq 0xffffffff81405560
<kfree_skb>
0xffffffff814676a9 <__udp_queue_rcv_skb+0x79>: mov $0xffffffff,%eax ---②
0xffffffff814676ae <__udp_queue_rcv_skb+0x7e>: jmp 0xffffffff81467662 <__
udp_queue_rcv_skb+0x32>
```

通过 `__udp_queue_rcv_skb()` 调出 `kfree_skb()` 的只有一处 (②)。要让②通过编译, ③处的 `sock_queue_rcv_skb()` 必须是个错误。

#### 例 4-7 <net/ipv4/udp.c>

```
static int __udp_queue_rcv_skb(struct sock *sk, struct sk_buff *skb)
{
    int is_udplite = IS_UDPLITE(sk);
    int rc;

    if ((rc = sock_queue_rcv_skb(sk, skb)) < 0) { ---③
        /* Note that an ENOMEM error is charged twice */
        if (rc == -ENOMEM) {
            UDP_INC_STATS_BH(sock_net(sk), UDP_MIB_RCVBUFEERRORS,
                              is_udplite);
            atomic_inc(&sk->sk_drops);
        }
        goto drop;
    }

    return 0;

drop:
    UDP_INC_STATS_BH(sock_net(sk), UDP_MIB_INERRORS, is_udplite);
    kfree_skb(skb); ---②
    return -1;
}
```

使用 SystemTap 确认 `sock_queue_rcv_skb()` 返回的地方, 结果为 ③。

```
<net/core/sock.c>

281 int sock_queue_rcv_skb(struct sock *sk, struct sk_buff *skb)
282 {
283     int err = 0;
284     int skb_len;
285     unsigned long flags;
```

```
286     struct sk_buff_head *list = &sk->sk_receive_queue;
287
288     /* Cast sk->rcvbuf to unsigned... It's pointless, but reduces
289      number of warnings when compiling with -W --ANK
290      */
291     if (atomic_read(&sk->sk_rmem_alloc) + skb->truesize >=
292         (unsigned)sk->sk_rcvbuf) {
293         err = -ENOMEM;
294         goto out;           ---❸
295     }
296
297     ....
```

sk\_rcvbuf 就是 /proc/sys/net/core/rmem\_default，在这里不作详细说明。初始值为 124 928。

```
# cat /proc/sys/net/core/rmem_default
124928
```

如果增大这个值，废弃的数据包就会减少，因此可以尝试进行下列设置。

```
# echo 4194304 > /proc/sys/net/core/rmem_default
```

这时如果执行 `udp-stress`，就不会输出 `__udp_queue_rcv_skb+79`。

## 小结

本节介绍了 `dropwatch`。可以确认系统中是否产生了过多的数据包废弃，以及在哪里发生废弃。在这个过程中可能找出需要进行调整的地方。

## 参考文献

- Welcome to `dropwatch`  
<https://fedorahosted.org/dropwatch/>
- Kernel Tracepoint API.  
<Documentation/trace/tracepoints.txt>

——Naohiro Ooiwa

# 虚 拟 化

本章将介绍使用 KVM 或 Xen 的虚拟化技术。在计算机的历史上，对各种设备进行虚拟化。使用调度程序的 CPU 资源虚拟化、使用内存管理的虚拟内存等都属于虚拟化。这里介绍的虚拟机技术是将包括 CPU、内存、I/O 设备在内的整个系统虚拟化的技术。除了以省电等为目的整合系统资源以外，还可以用来使用遗留（legacy）操作系统和生成动态资源。这里介绍的就是最大限度使用虚拟机的技术。

## HACK #28 如何使用 Xen

本节介绍管理程序（hypervisor）Xen 的使用方法。

Xen 是众多 Linux 发布版中所采用的管理程序。管理程序（又称虚拟机监视器）是为虚拟机提供运行环境的基础软件。Xen 是在 CPU 中还没有称为 Intel VT 或 AMD-V 的虚拟化支持功能的时代开发出的管理程序，可以通过半虚拟化（paravirtualization）方式运行虚拟机。在 Intel VT 或 AMD-V 出现后，也可以支持使用这些功能的全虚拟化（full virtualization）方式。半虚拟化方式与全虚拟化方式的框架如图 5-1 所示。

### 半虚拟化方式

客户端操作系统在已虚拟化的情况下，修改内核或设备驱动程序的方法。半虚拟化方式的客户端操作系统称为半虚拟化客户端（PV 客户端）。半虚拟化方式的驱动程序称为半虚拟化驱动程序（PV 驱动程序）。

### 全虚拟化方式

使用支持虚拟化功能的 Intel VT 或 AMD-V 等 CPU 的方法。将 PC 硬件上运行的操作系统直接在虚拟机环境上运行的方法。全虚拟化方式的客户端操作系统称为全虚拟化客户端（HVM 客户端）。

在 RHEL5 等使用 Xen 时，一般是使用 `virt-manager` 等来安装客户端操作系统。如果使用 `virt-manager`，就算没有虚拟化知识也可以很简单地生成客户端操作系统。Xen 作为 Linux 的虚拟机功能已经为大家所熟悉，因此这里仅介绍 Xen 的概要和不使用 `virt-manager` 时 Xen 的使用方法。

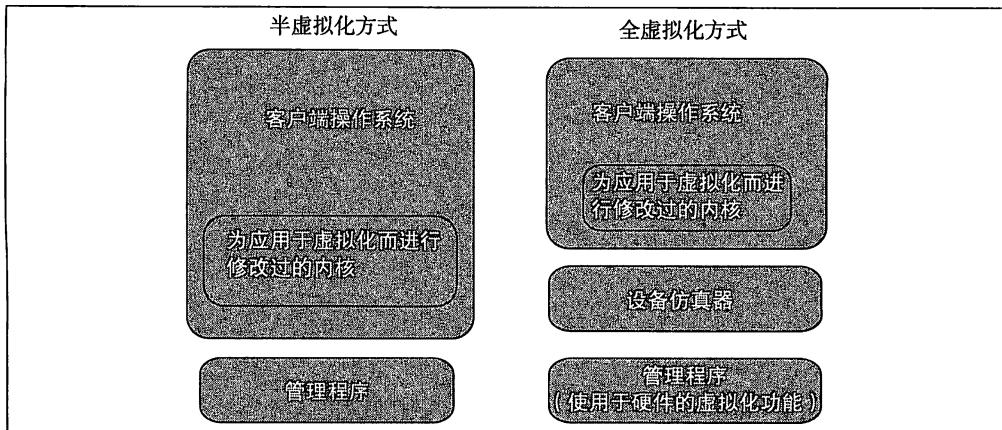


图 5-1 半虚拟化与全虚拟化

## Xen 的概要

构成 Xen 的主要要素有管理程序、主机操作系统 (Dom0)、客户端操作系统 (DomU)，如图 5-2 所示。

管理程序用来生成、删除、管理虚拟机的 CPU 或内存等。例如，进行虚拟 CPU 的调度或者向虚拟机分配内存等。从管理程序来看，主机操作系统看起来与客户端操作系统是一样的。主机操作系统与一般的客户端操作系统不同，具有向管理程序请求虚拟机环境控制处理的权限。主机操作系统中存在叫做 xend 的守护进程或者设备仿真器 qemu-dm 进程。xend 是接受生成客户端操作系统请求的守护进程，从 xm 命令接受处理要求，经由主机操作系统向管理程序请求生成虚拟机等。qemu-dm 虽然也称为设备模型，实际上 是仿真虚拟机设备的进程，用来仿真 VGA 或 IDE 等设备。

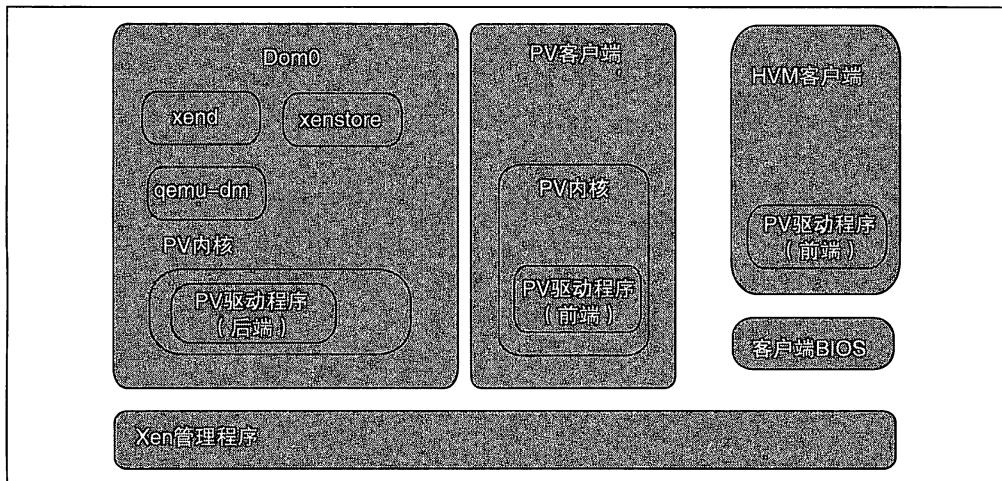


图 5-2 Xen 的概要

# Xen 的半虚拟化客户端的使用方法

为了能够看到生成的 Xen，这里不使用 `virt-manager` 生成半虚拟化客户端操作系统。下面以安装了 RHEL5-Xen 等 Xen 环境为前提。

1. 生成客户端操作系统的磁盘映像。

```
# dd if=/dev/zero of=/root/rhel5.img bs=1M seek=4096 count=0
# mke2fs -F -j /root/rhel5.img
```

2. 将当前运行中的 Linux 文件系统上的文件复制到步骤 1 中准备的磁盘映像中。

```
# mount -o loop /root/rhel5.img /mnt/
# cp -ax /{dev,etc,usr,bin,sbin,lib,lib64,var} /mnt/
# mkdir -p /mnt/{root,proc,sys,home,tmp}
```

3. 将磁盘映像上的 `fstab` 重新设置为用于客户端操作系统。

将 `/dev/xvda1` 设置为块设备。`/dev/xvda1` 是为客户端操作系统而准备的块设备名称。设置方法将在后面介绍。

```
# vim /mnt/etc/fstab
/dev/xvda1      /
tmpfs          /dev/shm    ext3      defaults      1 1
devpts         /dev/pts    devpts    defaults      0 0
sysfs          /sys       sysfs    defaults      0 0
proc           /proc      proc     defaults      0 0
```

4. 在磁盘映像的 `modprobe.conf` 中指定 PV 驱动程序模块。

在这里指定的是 PV 驱动程序的前端驱动程序——`xennet`、`xenblk`。

```
# vim /mnt/etc/modprobe.conf
alias eth0 xennet
alias scsi_hostadapter xenblk
```

5. 将 `runlevel` 设置为 3。

将 `/mnt/etc/inittab` 的 `initdefault` 行进行如下更改。

```
/mnt/etc/inittab initdefault
id:3:initdefault:
```

6. 向 `securetty` 添加 `tty0`。

为了可以从控制台登录，需要向 `securetty` 添加 `tty0`。

```
# echo tty0 >> /etc/securetty
```

7. 生成 `initrd`（在 Dom0 上的 `/boot` 下生成）。

指定客户端操作系统的 `fstab`，生成 `initrd`。指定时要使 PV 驱动程序模块 `xennet`、`xenblk` 也加入 `initrd` 中。

```
# mkinitrd -f /boot/initrd-2.6.18-164.el5xenU.img 2.6.18-164.el5xen --fstab /
mnt/etc/fstab --with xennet --with xenblk
```

```
# umount /mnt/
```

8. 启动客户端操作系统的启动配置文件。

客户端内核和 initrd 放在 Dom0 上。

```
# vim /etc/xen/rhel5
kernel = "/boot/vmlinuz-2.6.18-164.el5xen"
ramdisk = "/boot/initrd-2.6.18-164.el5xenU.img"
memory = 1024
name = "rhel5"
disk = [ 'file:/root/rhel5.img,xvda1,w' ]
root = "/dev/xvda1 ro"
extra = "3"
```

9. 启动客户端操作系统。

使用 xm create 命令启动客户端操作系统。通过 -c 选项来连接到启动的客户端操作系统的控制台。

```
# xm create -c rhel5
```

## Xen 的全虚拟化客户端的使用方法

1. 生成客户端操作系统的磁盘映像。

```
# dd if=/dev/zero of=/root/rhel5hvm.img bs=1M seek=4096 count=0
```

2. 生成客户端操作系统的配置文件。

指定步骤 1 中生成的磁盘映像和安装媒体的 ISO。启动顺序依次为 CD-ROM 驱动器 (d) 和 HDD (c)。

```
# vim /etc/xen/rhel5hvm
memory = 1024
builder = "hvm"
kernel = "/usr/lib/xen/boot/hvmloader"
pae = 1
acpi = 1
apic = 1
device_model = '/usr/lib64/xen/bin/qemu-dm' 注1
sdl = 0
vnc = 1
vncdisplay=7
vncpasswd=''
vnclisten="0.0.0.0"
keymap = "ja"
boot = "dc"
disk = [ "file:/root/rhel5hvm.img,hda,w", "file:/root/rhel-server-5.6-
x86_64-dvd.iso,hdc:cdrom,r" ]
vif = [ '' ]
serial = "pty"
on_poweroff = "destroy"
on_reboot = "restart"
```

---

注 1: 主机操作系统为 x86\_64 的情况。在 x86\_32 的情况下指定 device='/usr/lib/sen/bin/qemu-dm'。

```
on_crash = "restart"
```

### 3. 使用 `xm create` 启动客户端操作系统，安装操作系统。

```
# xm create rhel5hvm
```

为了获取客户端操作系统的 VGA 控制台，在主机操作系统上按照下列方式启动 `vncviewer`。

```
# vncviewer 127.0.0.1:7
```

安装完成后，如果重新启动客户端操作系统，安装程序就会再次启动，因此安装完成后应使用 `xm destroy` 停止客户端操作系统。

```
# xm destroy rhel5hvm
```

### 4. 编辑客户端操作系统的设置文件，卸载安装媒体的 ISO。

```
# vim /etc/xen/rhel5hvm
name = "rhel5hvm"
memory = 1024
builder = "hvm"
kernel = "/usr/lib/xen/boot/hvmloader"
pae = 1
acpi = 1
apic = 1
device_model = '/usr/lib64/xen/bin/qemu-dm'注2
sdl = 0
vnc = 1
vncdisplay=7
vncpasswd=''
vnclisten="0.0.0.0"
keymap = "ja"
boot = "dc"
disk = [ "file:/root/rhel5hvm.img,hda,w", " hdc:cdrom,r" ]
vif = [ '' ]
serial = "pty"
on_poweroff = "destroy"
on_reboot = "restart"
on_crash = "restart"
```

### 5. 启动客户端操作系统。

```
# xm create rhel5hvm
```

## 小结

这里介绍了使用低层次 API 的客户端操作系统生成方法。`virt-manager` 在内部的功能也应该有所了解了。Xen 的客户端操作系统中还有其他一些可以设置的项目（见表 5-1），大家可以尝试一下。

---

<sup>注2</sup>：主机操作系统为 `x86_64` 的情况。在 `x86_32` 的情况下指定 `device_model = '/usr/lib/xen/bin/qemu-dm'`。

表 5-1 客户端操作系统的设置

项 目	说 明
kernel	主要指定半虚拟化客户端时启动的 kernel
ramdisk	主要指定半虚拟化客户端时启动的 initrd
builder	指定全虚拟化客户端 (hvm)、半虚拟化客户端 (linux)
memory	指定内存大小。单位为 MB
uuid	用于客户端操作系统识别的 UUID。也指定到 SMBI 操作系统 type1 System information
name	客户端操作系统名称
cpus	指定客户端操作系统可以使用的物理 CPU
vcpus	虚拟 CPU 数
vif	设置虚拟 NIC (MAC 地址、以太网模型、连接的网桥、设备类型)
disk	设置虚拟磁盘 (磁盘类型、路径、客户端操作系统上的设备名称、读写模式)
vtpm	设置 TPM
dhcp	启用传递到内核启动参数的 dhcp
netmask	指定传递到内核启动参数的子网掩码
gateway	指定传递到内核启动参数的网关
Hostname	指定传递到内核启动参数的主机名称
root	指定传递到内核启动参数的 root 设备
nfs_server	指定传递到内核启动参数的 NFS 服务器
nfs_root	指定传递到内核启动参数的 NFS 的 root 设备
extra	指定其他传递到内核启动参数的选项
on_poweroff	电源切断时的操作 (destroy、restart、preserve、rename-restart)
on_reboot	重启时的操作 (destroy、restart、preserve、rename-restart)
on_crash	内核 panic 时的操作 (destroy、restart、preserve、rename-restart)
device_model:	设备仿真器 (qemu-dm) 的路径
boot	启动顺序 (c:HDD、d:CD-ROM 驱动器、a:FDD)
snapshot	磁盘映像的写时复制 (copy on write)
sdl	在 VGA 中使用 SDL 库
vnc	在 VGA 中使用 VNC 显示器

项 目	说 明
vnclisten	VNC 的连接许可地址
vncdisplay	VNC 的显示器编号
vncunused	自动分配 VNC 中未使用的显示器编号
vncpasswd	VNC 连接时使用的密码
stdvga	将 VGA 中使用的设备模型设置为 stdvga
serial	将串行控制台重定向到 pty 设备
soundhw	声卡的模型
localtime	启用 localtime
full-screen	全屏启动
usb	启用 USB
usbdevice	启用 USB 鼠标或 USB 图形输入板
keymap	指定键盘的布局

## HACK #29 如何使用 KVM

本节介绍 KVM 的使用方法。

KVM 是 Linux 2.6.20 中采用的 Linux 内核的管理程序功能。管理程序是指将多个操作系统在 1 台物理机器上运行的软件，KVM 在 RHEL6 等中采用。KVM 采用的是全虚拟化方式，必须在系统能够使用 CPU 的 Inter VT 或 AMD-V 功能的状态下才能使用。使用 KVM 时，请在 BIOS 的设置界面上确认虚拟化功能是否可用。

### KVM 的概要

如图 5-3 所示，KVM 是由管理程序——kvm 启动程序和设备仿真器——qemu-kvm 构成的。qemu-kvm 是在 PC 仿真器 qemu 上进行了一些针对 KVM 的修改后得到的。KVM 的特征是 kvm 驱动程序作为 Linux 内核的一部分运行，因此可以使用 Linux 内核的大部分功能。例如，进程调度程序功能或省电功能就可以直接使用 Linux 内核的功能。

### KVM 的使用方法

KVM 一般通过 virt-manager 来控制，但为了看到 KVM 是如何运行的，这里尝试使用 qemu-kvm 来控制 KVM。使用 qemu-kvm 就可以使用 qemu 的大部分选项。qemu-kvm 的选项非常多，这里仅介绍经常用到的一部分选项（见表 5-2）。

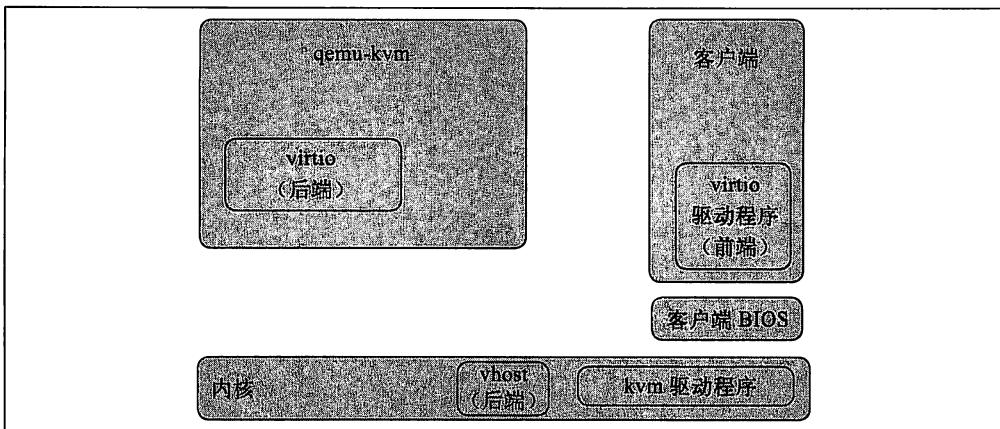


图 5-3 KVM 的概要

表 5-2 qemu 的主要选项

选 项	说 明
<code>-M</code>	指定仿真机器的种类（通过 <code>-M?</code> 可以显示 qemu 支持的机器种类的列表）
<code>-cpu</code>	指定 CPU 的模型（通过 <code>-cpu?</code> 可以显示可指定的模型的列表）
<code>-smp</code>	指定 CPU 数（未指定时设置为 1）。 <code>smp</code> 选项的子选项如下（用逗号分开指定）： <code>maxcpus=cpus</code> 包括可以动态添加的 CPU 在内的最大 CPU 数 <code>cores=cores</code> 平均 1 个套接字的核数 <code>threads=threads</code> 平均 1 个核的线程数 <code>sockets=sockets</code> 系统的套接字数
<code>-fda/-fdb</code>	分别将文件指定为第一台计算机的软盘、第二台计算机的软盘的映像
<code>-hda/-hdb/-hdc/-hdd</code>	分别将文件指定为第一台计算机、第二台计算机、第三台计算机、第四台计算机的 IDE 硬盘的映像
<code>-cdrom</code>	将文件指定为 IDE CD-ROM 映像（CD-ROM 连接到第二台计算机的 IDE 的主设备）
<code>-drive</code>	详细指定磁盘驱动程序 <code>drive</code> 选项的子选项如下（用逗号分开指定）： <code>file=file</code> 将 <code>file</code> 指定为磁盘映像 <code>if=interface</code> 指定连接驱动器的接口种类（ <code>ide</code> 、 <code>scsi</code> 、 <code>sd</code> 、 <code>mtd</code> 、 <code>floppy</code> 、 <code>pflash</code> 、 <code>virtio</code> ）

选 项	说 明
<b>-drive</b>	<b>bus=bus,unit=unit</b> 指定连接驱动器的总线编号和单元 id <b>index=index</b> 指定将驱动器连接到第几个接头 <b>media=media</b> 指定驱动器的种类 (disk、cdrom) <b>cache=cache</b> 指定访问磁盘上的数据时怎样使用 主机操作系统上的磁盘缓存 (none、 write back、unsafe、writethrough) <b>aio=aio</b> 指定 threads 或 native。threads 表示 基于 pthread 的非同步 I/O, native 表示 Linux AIO <b>format=format</b> 指定磁盘格式。这时不会自动检测磁 盘格式, 因此, 如果指定 format=raw 就可以避免错误运行 <b>serial=serial</b> 赋予设备的串行号 <b>addr=addr PCI</b> 指定 PCI 地址 (仅 if=virtio 时)
<b>-boot</b>	指定启动顺序。启动顺序可以指定软盘 (a)、硬盘 (c)、 CD-ROM (d)、网络 (n)。
<b>-m</b>	指定内存量 (单位为 MB)
<b>-k</b>	指定键盘布局
<b>-usb</b>	启用 usb
<b>-usbdevice</b>	指定 usb 设备 (mouse、tablet、disk、host、serial、braille、 net)
<b>-device</b>	指定添加的设备。可用的驱动程序列表可以通过 -device? 来显示。各驱动程序的可设置属性可以通过 -device driver,? 来显示
<b>-name</b>	设置客户端名称
<b>-uuid</b>	指定机器的 UUID
<b>-nographic</b>	禁用图形模式
<b>-spice</b>	启用 spice 远程桌面协议
<b>-vga</b>	指定显卡的种类 (std、cirrus、vmware、xenfb、qxl、 none)
<b>-vnc</b>	开启 VNC 服务器
<b>-enable-kvm</b>	启用 KVM

在 RHEL6 中使用 `qemu-kvm` 启动 KVM 的客户端操作系统时，操作步骤如下。

```
# /usr/libexec/qemu-kvm -M rhel6.0.0 -enable-kvm -m 1024 -smp 2 -name rhel6-4  
-boot c -hda /dev/sdc -vnc 127.0.0.1:2
```

如果是第一次安装等情况，则需要指定 `cdrom` 选项，并向 `boot` 选项指定 CD-ROM(d) 来启动。

```
# /usr/libexec/qemu-kvm -M rhel6.0.0 -enable-kvm -m 1024 -smp 2 -name rhel6-4  
-boot d -hda /dev/sdc -cdrom /root/rhel6.iso -vnc 127.0.0.1:2
```

在这里启动的是 1GB 内存、2CPU 的客户端操作系统。将主机操作系统的 `/dev/sdc` 显示为客户端操作系统的第一台计算机的 IDE 磁盘，设备驱动程序就是 `virtio`。客户端操作系统的 VGA 可以通过 VNC 连接到 2 号显示器。这里省略了网络的选项，可以使用默认设置的 `-net nic -net user` 选项。

## KVM 的网络选项

用于网络连接的选项也有一些（见表 5-3），但一般使用网桥或者 NAT 连接到外部网络。这里介绍经由网桥连接的方法。需要事先在主机操作系统内安装如下工具包。

```
bridge-utils  
tunctl  
iproute
```

将外部连接用的 NIC 设置为 `eth0` 时，为了将 `eth0` 连接到网桥，需要编辑 `/etc/sysconfig/network-scripts/ifcfg-eth0` 的如下内容。

```
DEVICE="eth0"  
NM_CONTROLLED="no"  
ONBOOT=yes  
HWADDR=00:1D:7D:53:C1:EC  
TYPE=Ethernet  
BRIDGE=br0  
NAME="System eth0"  
UUID=5fb06bd0-0bb0-7ffb-45f1-d6edd65f3e03
```

此外，将 `/etc/sysconfig/network-scripts/ifcfg-br0` 按下列内容进行编辑，以用于网桥。

```
DEVICE="br0"  
NM_CONTROLLED="no"  
ONBOOT=yes  
TYPE=Bridge  
BOOTPROTO=none  
IPADDR=192.168.1.2  
PREFIX=24  
GATEWAY=192.168.1.1  
DNS1=192.168.1.3  
DEFROUTE=yes  
IPV4_FAILURE_FATAL=yes  
IPV6INIT=no
```

要让编辑后的网络脚本立刻生效，需要重新启动网络服务。

```
# service network restart
```

在 /etc/qemu-ifup 中准备 qemu-kvm 用的网络脚本。

```
# !/bin/sh
set -x

switch=br0

if [ -n "$1" ];then
    /usr/bin/sudo /usr/sbin/tunctl -u 'whoami' -t $1
    /usr/bin/sudo /sbin/ip link set $1 up
    sleep 0.5s
    /usr/bin/sudo /usr/sbin/brctl addif $switch $1
    exit 0
else
    echo "Error: no interface specified"
    exit 1
fi
```

指定下列网络选项，启动 qemu。

```
# /usr/libexec/qemu-kvm -M rhel6.0.0 -enable-kvm -m 1024 -smp 2 -name rhel6-4 -boot c
-hda /dev/sdc -vnc 127.0.0.1:2 -net nic -net tap
```

表 5-3 qemu 的主要网络选项

选 项	说 明
-net nic	创建网络接口 -net nic 选项的子选项如下（用逗号分开指定）： vlan=n          指定 vlan id macaddr=mac    MAC 地址 model=type     网卡的模型类型（virtio、i82551、i82557b、i82559er、ne2k_                       pci、ne2k_isa、pcnet、rtl8139、e1000、smc91c111、lance、                       mcf_fe） name=name      网卡名称 addr=addr      PCI 的设备号 vectors=v      MSI-X 的矢量号
-net user	使用以用户模式运行的网络栈 -net user 选项的子选项如下（用逗号分开指定）： vlan=n          指定 vlan id name            网卡名称 net=addr[/mask]  指定 IP 网络地址和子网掩码（默认为 10.0.2.0/24） host=addr       指定从客户端操作系统看到的主机操作系统的 IP 地址 restrict=y yes n no  如果指定 yes，则客户端操作系统被隔离

选 项	说 明
-net user hostname=name	指定的主机通过 DHCP 服务器被通知到客户端操作系统
dhcpstart=addr	指定安装的 DHCP 服务器可以分配的 IP 地址。使用指定的 IP 地址中的 16 个地址
dns=addr	指定从客户端操作系统看到的虚拟 DNS 服务器的 IP 地址
-net tap	利用 TAP 的网络接口
-net tap 选项的子选项如下 (用逗号分开指定) :	
vlan=n	指定 vlan id
name=name	网卡名称
fd=h	已打开的 TAP 设备号
ifname=name	TAP 设备的名称
script=file	指定 Up 网络接口时使用的网络脚本。未指定时为 /etc/qemu-ifup
downscript=dfile	指定 Down 网络接口时使用的网络脚本。未指定时为 /etc/qemu-ifdown
sndbuf=nbytes	send buffer size 的上限大小。未指定时为 1 048 576
vnet_hdr=off on	off 表示禁用 IEF_VNET_HDR 标记 on 表示启用 IEF_VNET_HDR 标记
vhost=on	启用内核的 vhost 功能
vhostfd=h	连接到已经打开的 vhost net 设备
-net none	不使用网络设备。(qemu 的默认设置为 -net nic -net user)

## 小结

这里介绍了通过 qemu-kvm 使用 KVM 的方法。从 virt-manager 等使用时也是在内部执行 qemu-kvm。使用 ps 命令等进行确认，就可以根据这里介绍的内容，了解 KVM 的客户端操作系统是如何启动的。qemu-kvm 的选项除了这里介绍的以外还有很多，推荐参考 qemu 的用户手册。

## 参考文献

- KVM Howto's  
<http://www.linux-kvm.org/page/HOWTO>
- QEMU Emulator User Documentation  
<http://qemu.weilnetz.de/qemu-doc.html>

——Akio Takebe

# HACK #30 如何不使用 DVD 安装操作系统

本节介绍不使用 KVM，而是利用 KVM 安装在物理机器环境下启动的 Linux。

一般安装操作系统时需要将安装用的 ISO 映像刻录到 DVD 然后再进行，但是刻录 DVD 需要花费时间和精力，更重要的是要用到 DVD 盘，不利于环境保护。

本节介绍的是不需要刻录 DVD，使用 KVM 安装虚拟环境下的客户端操作系统并用在物理环境中的方法。

使用 KVM 安装操作系统时是通过 `virt-manager` 来进行的。这里使用的主机操作系统是 Fedora 14，客户端操作系统是 CentOS 5.6。另外，客户端操作系统安装的地方是 USB HDD。启动 `virt-manager`，根据提示安装，就可以顺利地将客户端操作系统安装到 USB HDD。

## 需要的准备

### 安装用媒体的映像

选择 CentOS 5.6 的 ISO 映像文件。本次使用的是 `CentOS 5.6-i386-bin-DVD.iso`。

### 安装用磁盘

这里将 USB HDD 分配为安装客户端操作系统的磁盘区。在笔者的环境下 `/dev/sdb` 为 USB HDD，因此按照图 5-4 所示进行选择。请向客户端操作系统分配整个磁盘，而不是磁盘分区表。

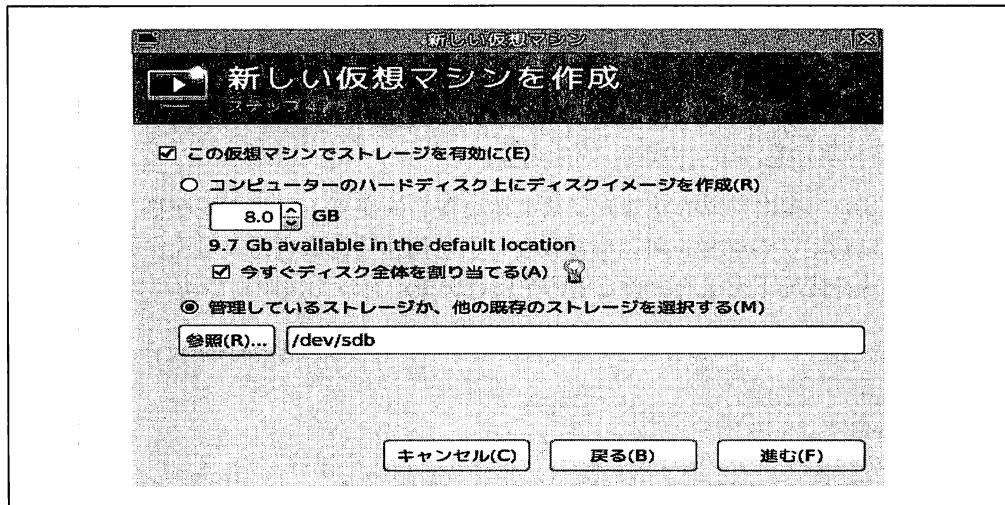


图 5-4 安装界面

成功地将 CentOS 5.6 安装为客户端操作系统后，登录到客户端操作系统，使用 `mkinitrd` 命令重新生成 `initrd`。在笔者的环境下安装的磁盘是 USB HDD，因此按照下列方法将 `usb-storage` 驱动程序模块安装到 `initrd`。

```
# mkinitrd -f /boot/initrd-2.6.18-238.el5.2.img 2.6.18-238.el5 --with usb-storage
```

接下来，在客户端操作系统上修改 `/boot/grub/grub.conf`，改为通过 UUID 指定 `root` 设备。`initrd` 指的是刚才生成的 `initrd`。

```
default=0
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title CentOS (2.6.18-238.el5)
    root (hd0,0)
#    kernel /vmlinuz-2.6.18-238.el5 ro root=LABEL=/ rhgb quiet
    kernel /vmlinuz-2.6.18-238.el5 ro root=UUID=b04d14e9-1a6c-417f-a981-
c9f2e9a4ce26 rhgb quiet
#    initrd /initrd-2.6.18-238.el5.img
    initrd /initrd-2.6.18-238.el5.2.img
```

UUID 可以通过 `blkid` 命令来确认。

```
# blkid /dev/vda2
/dev/vda2: LABEL="/" UUID="b04d14e9-1a6c-417f-a981-c9f2e9a4ce26"
SEC_TYPE="ext2" TYPE="ext3"
```

到这一步准备工作就完成了。在物理环境下启动时，请从 BIOS 的 `boot` 菜单中选择 USB HDD 作为启动磁盘。选择 USB HDD 作为启动磁盘后，BIOS 就会读入 USB HDD 最前面的 MBR，执行 `bootloader`。在这里是从 USB HDD 启动 GRUB。内核会安装 `initrd` 中的内核模块，但在虚拟环境下启动时是使用 `virtio-blk` 驱动程序模块读入磁盘，在物理环境下是使用 `usb-storage` 驱动程序模块读入磁盘，因此生成的磁盘不论在哪个环境下都可以启动。

需要注意的是，不能在 `grub.conf` 或 `/etc/fstab` 里直接写入 `/dev/sdb` 等设备文件名；因为在虚拟环境下，如果使用了 IDE 的仿真器就会分配 `hda` 设备文件名；如果使用了 `virtio` 等 PV 驱动程序就会分配 `vda` 的名称。

如果使用 UUID，操作系统的功能会屏蔽上述设备名的变更，因此不论在哪个环境下操作系统都可以启动。卷标号也可以屏蔽设备，但是使用其他磁盘时就可能会存在相同的卷标号。在这种情况下无法判断指定了哪个磁盘设备，有时无法顺利启动。

---

**小贴士：**如果实在不行，还可以按照下列方法解压缩 `initrd`，直接编辑其中的 `init` 文件。

1. 解压缩 `initrd`。

```
# mkdir work
# cd work
# gzip -cd /boot/initrd-2.6.18-238.el5.img | cpio -id
```

2. 编辑 `init` 文件。

```
# vim init
3. 重新生成 initrd。
# find . -print | cpio --quiet -c -o | gzip -9 > /boot/initrd-2.6.18-
238.el5.new.img
```

---

## 小结

本节介绍了使用 CentOS 5.6 作为客户端操作系统时的例子，而在 RHEL6 和 Fedora 14 等最近的发布版中采用的是 initramfs。initramfs 可以创建比 initrd 更加通用的 preboot 环境。RHEL6 等的 initramfs 一般包含驱动程序，默认使用的是通过 UUID 指定设备。因此多数情况下不用进行上述的特殊操作，就可以直接在物理、虚拟两种环境中使用。

本次测试中使用的环境如下，仅供参考。

- 系统：FMV-P8230
- 客户端操作系统：CentOS-5.6-i386
- USB-HDD：I-O Data HDPC-U320

——Akio Takebe

## HACK #31 更改虚拟 CPU 分配方法，提高性能

本节介绍虚拟环境下的虚拟 CPU 分配方法的技巧。

虚拟化环境下客户端操作系统的各虚拟 CPU 可以各自独立地分配物理 CPU。例如，可以将客户端操作系统 A 的虚拟 CPU1 分配（pin 指定）给主机操作系统的物理 CPU3。当各虚拟 CPU 这样占用物理 CPU 时，人们容易认为物理 CPU 的资源相互完全分离。但是，在现代的多核 CPU 中，各物理 CPU 资源多数未完全分离，如果不能正确选择分配给虚拟 CPU 的物理 CPU，就有可能导致性能大幅降低。本节将介绍应当使用怎样的分配方法。

首先，有的多核 CPU 是在核之间共享 L2 缓存或 L3 缓存的。例如，Intel ® Core™ 2 Quad CPU Q6700 中有 4 个核，其中两个核共享 1 个 L2 缓存（见图 5-5）。

在将某个虚拟机的两个虚拟 CPU 分别分配给不共享 L2 缓存的核时，一旦加大内存负载就有可能造成 CPU 性能降低（见图 5-6）。

### 使用 virt-manager 的物理 CPU 分配方法

virt-manager 可以如图 5-7 所示，通过 CPU pinning 菜单将虚拟机的虚拟 CPU 固定分配给物理 CPU。这里显示的 CPU 编号为 Linux 内核所识别的 CPU 编号，而不是用来标识核的位置。要找出共享缓存的 CPU，可以参照 /sys/devices/system/cpu/cpuN/cache/

index $M$ /shared\_cpu\_list<sup>注3</sup>。

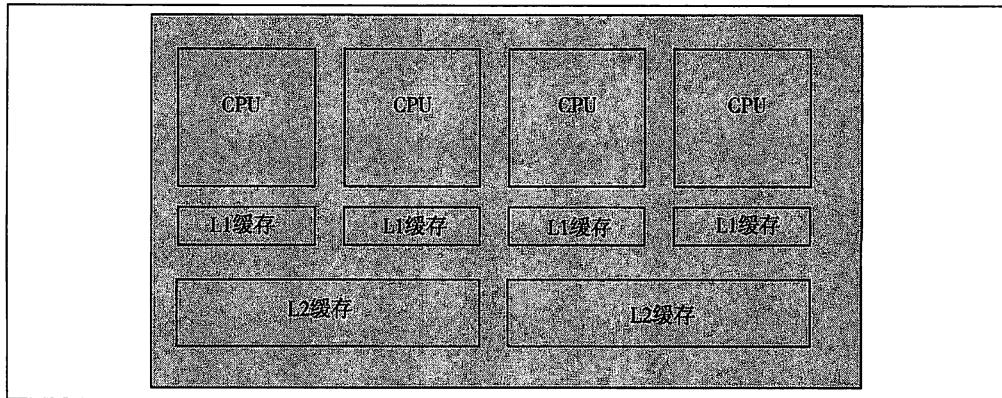


图 5-5 Q6700

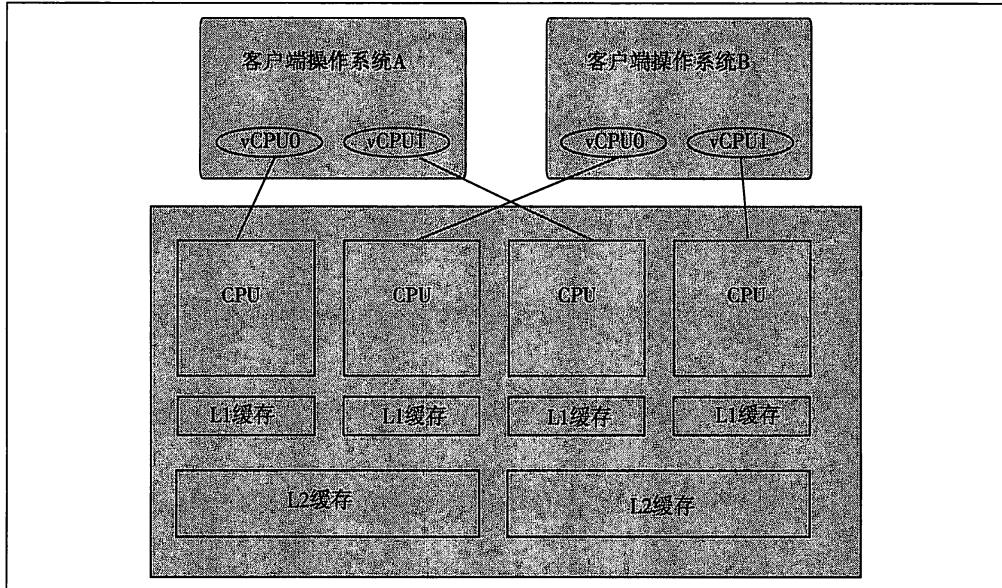


图 5-6 导致性能降低的分配方法

在这个环境下, 可以得知 CPU0 和 CPU2, CPU1 和 CPU3 共享缓存。

```
# cat /sys/devices/system/cpu/cpu0/cache/index2/shared_cpu_list
0,2
# cat /sys/devices/system/cpu/cpu1/cache/index2/shared_cpu_list
1,3
```

注 3:  $N$ 、 $M$  代表数字。

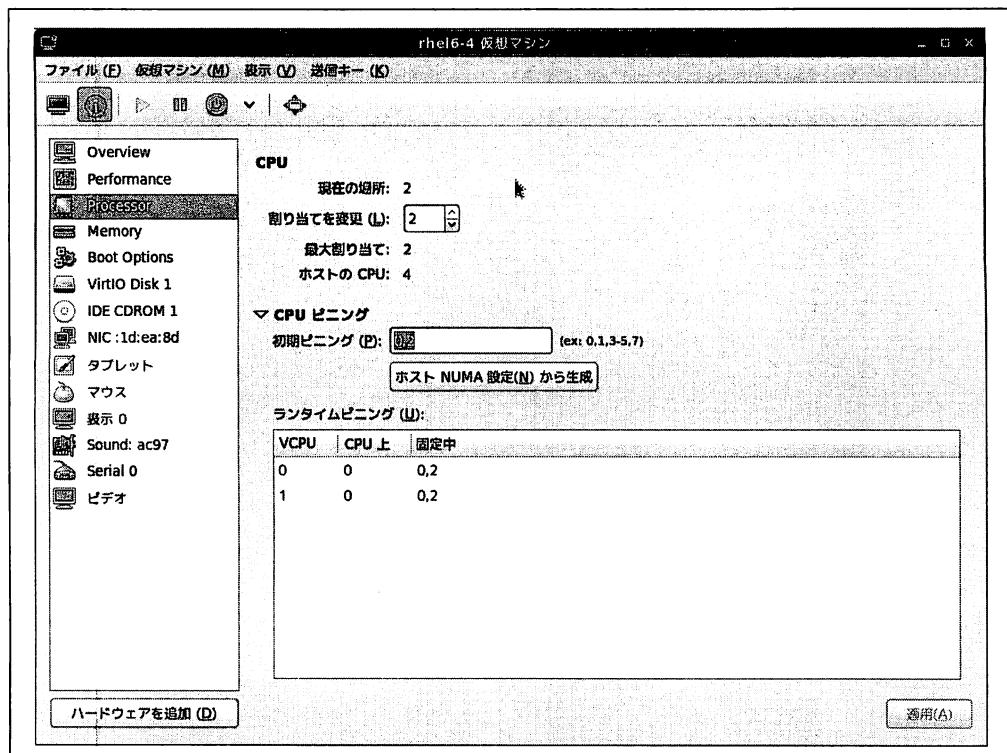


图 5-7 在 CPU pinning 菜单上设置物理 CPU 的固定分配

## 概要分析

下面查看在共享 L2 缓存的情况和不共享的情况下性能会有多大的变化（如图 5-8 和图 5-9 所示）。这里准备了两个客户端操作系统，在客户端操作系统 A 中执行 UnixBench，在客户端操作系统 B 中加大 CPU 或内存负载。

CPU	Intel® Core™ 2 Quad CPU Q6700 @ 2.66GHz
内存	4GB
主机操作系统	RHEL6
客户端操作系统	RHEL6
客户端操作系统 CPU	虚拟 CPU
客户端操作系统内存	1GB

CPU 和内存的负载工具是 stress<sup>注4</sup>。

注 4: <http://weather.ou.edu/~apw/projects/stress/stress-1.0.4.tar.gz>

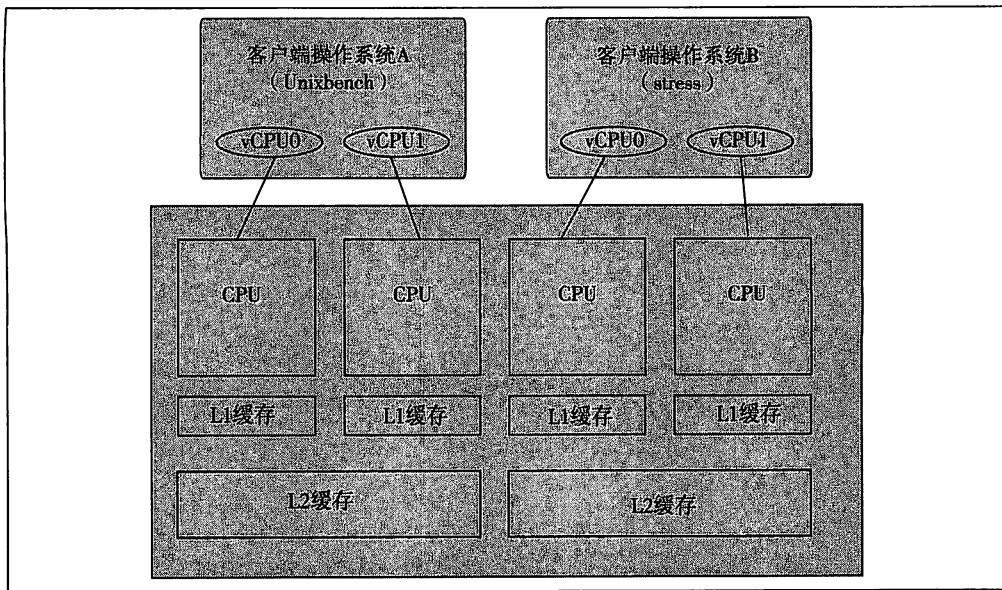


图 5-8 各客户端操作系统分别共享 L2 缓存的情况

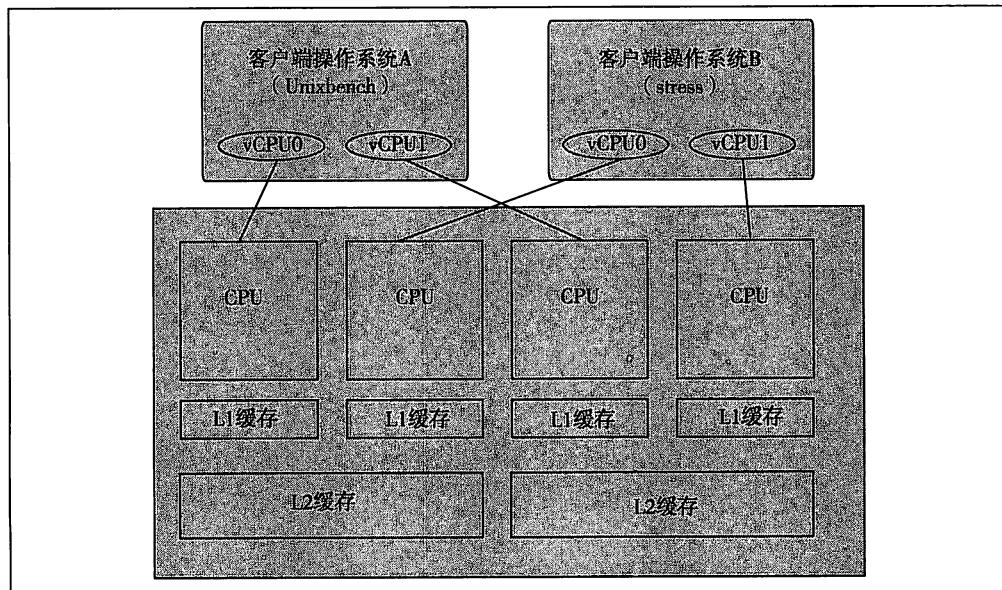


图 5-9 两个客户端操作系统共享 L2 缓存的情况

图 5-10 所示为 UnixBench 的结果。纵轴为与客户端操作系统 B 空间状态的比例。横轴为 UnixBench 的各基准测试的种类。从结果来看，在客户端操作系统之间共享 L2 缓存的情况下，对客户端操作系统 B 加大 CPU 负载时，客户端操作系统 A 的性能降低并不明显。但是对客户端操作系统 B 加大内存负载时，客户端操作系统 A 的性能降低了 70% 左右。但是，对客户端操作系统 B 加大内存负载时，客户端操作系统 A 的 CPU 相关基

准测试中并未出现太大的性能降低。

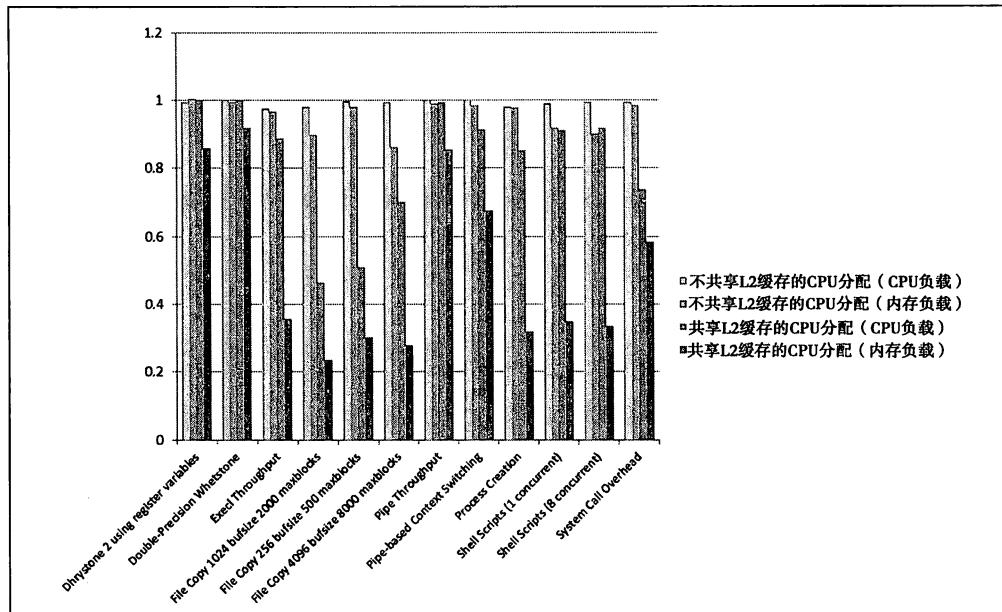


图 5-10 UnixBench 的结果

那么，在实际的 DB 服务器等上会产生多大的影响呢？图 5-11 所示为进行 sysbench 而非 UnixBench 时的示例。纵轴为执行 sysbench 的合计时间。在客户端操作系统共享

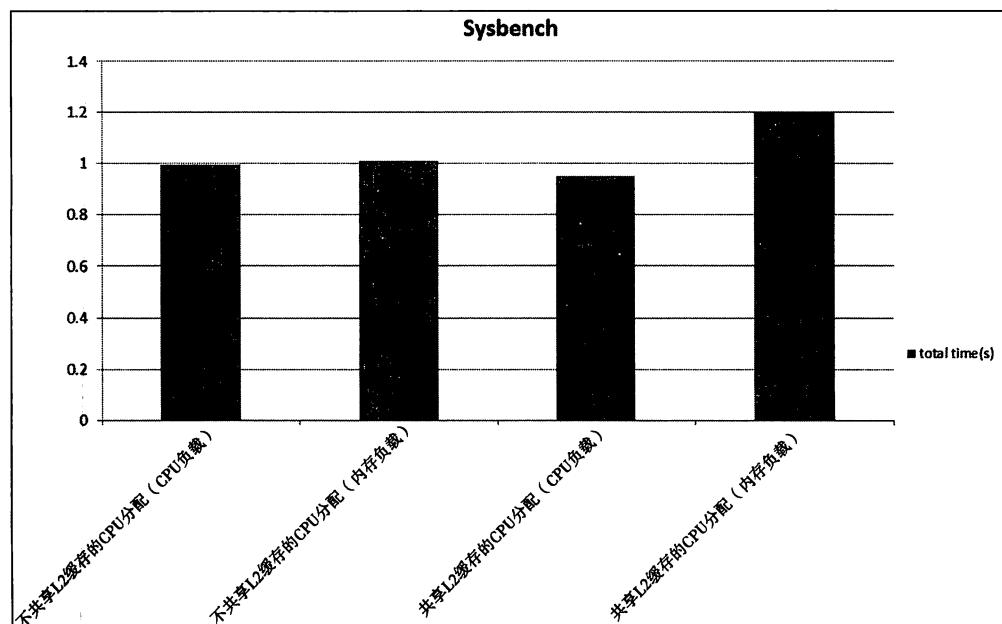


图 5-11 sysbench 的结果

L2 缓存的情况下，如果加大对客户端操作系统 B 的内存负载，从客户端操作系统 A 的 sysbench 结果也能够看出性能降低了 20% 左右。

根据以上结果，可以得知：

1. 内存负载较高的客户端操作系统在分配 CPU 时必须让该客户端操作系统的虚拟 CPU 共享缓存。
2. 由于 CPU 负载较高的客户端操作系统在共享 CPU 缓存时会产生影响较小，因此可用它来与内存负载较高的客户端操作系统共享 CPU 缓存。

## 小结

这里介绍了通过 CPU 分配来改善客户端操作系统性能的技巧。一般在 Java 等应用程序服务器或 DB 服务器中内存负载有增加的倾向。

除 CPU 以外，对各客户端操作系统共享的资源或组件进行一些改进，也可以改善性能。例如，在使用 virtio 等半虚拟化驱动程序时，就会共享主机操作系统的 I/O 调度程序。已有论文提出在这种情况下如果将主机操作系统的 I/O 调度程序换成更适合系统的 noop 或 deadline 等，就可以改善性能<sup>注5</sup>。

在云等大规模环境下，如果能够较好地配置客户端操作系统，就可以实现不会产生性能降低的系统，大家可以尝试一下。

## 参考文献

- XenSummit 2007 November Padmashree K Apparao: Characterization and Analysis of a Server Consolidation Benchmark
- UnixBench  
<http://code.google.com/p/byte-unixbench/>
- sysbench  
<http://sysbench.sourceforge.net/>

——Akio Takebe

## HACK#32 如何使用 EPT 提高客户端操作系统的性能

这里将介绍 Memory Management Unit (MMU) 的虚拟化功能——Extended Page Table (EPT) 功能。

---

注 5：SOSP HotStorage '09 Dave Boulter and Abhishek Chandra (University of Minnesota): Does Virtualization Make Disk Scheduling Passe?  
<http://www.sigops.org/sosp/sosp09/hotstorage.html>

MMU

Memory Management Unit (MMU) 是具有内存管理功能的硬件。MMU 有一个主要功能是将虚拟地址转换为物理地址。一般在 x86 系列的 CPU 上运行的操作系统是使用分页 (paging) 功能将虚拟地址转换为物理地址的。分页功能涉及进行地址转换的转换表。这个表称为页表 (page table)。MMU 使用从 CPU 的 CR3 寄存器找到的页表，从虚拟地址转化为物理地址。IA-32 的分页结构如图 5-12 所示。

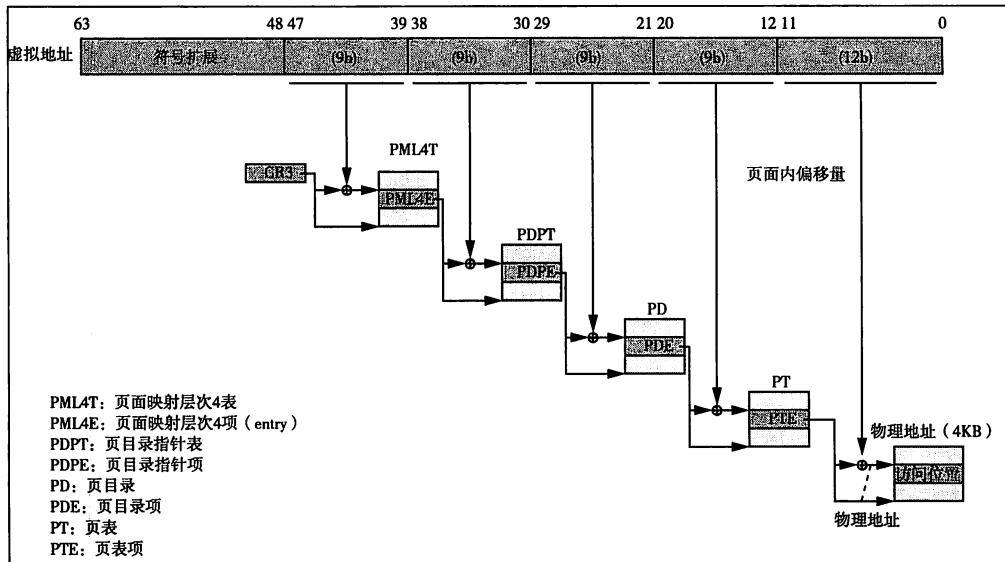


图 5-12 IA-32 的分页结构 (IA-32e 模式、4KB 页面)

## 影子页表

在虚拟环境下的内存地址有以下种类（见图 5-13）。

- **客户端虚拟地址**——从客户端操作系统上看到的虚拟地址。
  - **客户端物理地址**——从客户端操作系统上看到的物理地址。管理程序仿真的模拟物理地址。
  - **主机虚拟地址**——从主机操作系统上看到的虚拟地址。
  - **主机物理地址**——从主机操作系统上看到的物理地址。与实际的物理地址相同。在虚拟化环境下有时也称为机器地址（绝对地址）。

在虚拟化环境下，不能由客户端操作系统来管理物理地址，因此客户端物理地址就是管理程序仿真的模拟地址。但使用这个客户端物理地址是无法访问实际的内存的，因此需要转换成主机物理地址。在 Xen 或 KVM 中，管理程序通过影子分页（shadow page table）的方法进行 MMU 的仿真。如图 5-14 所示，在影子分页中，管理程序先检测出客户端操作系统的页表配置处理方式，再仿真客户端操作系统的页表配置处理方式。管理程序会设置将客户端虚拟地址转换为主机物理地址的表。这个表称为影子表。管理程序

将影子表设置为 CPU 的 CR3 寄存器。从而 MMU 就可以将客户端操作系统访问的客户端虚拟地址转换为主机物理地址。

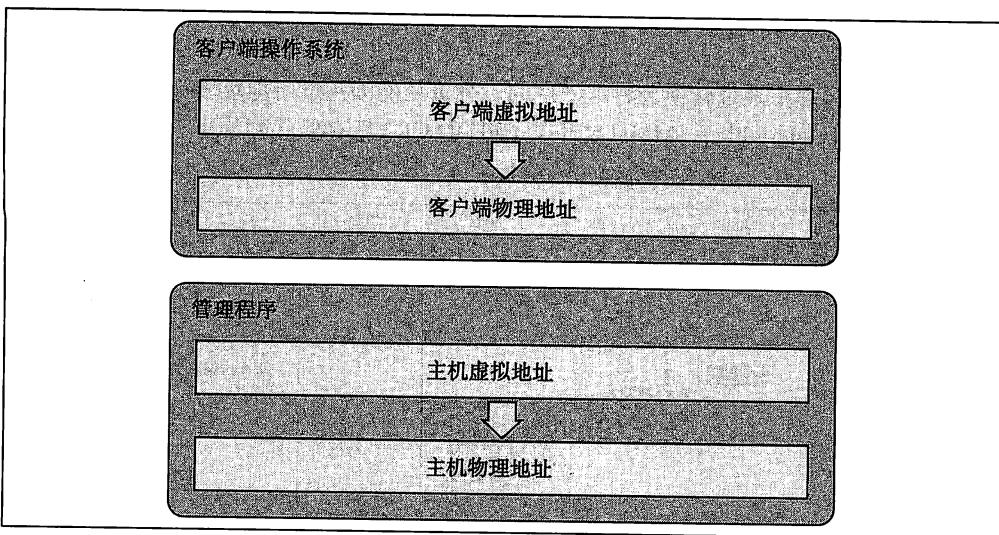


图 5-13 4 种地址

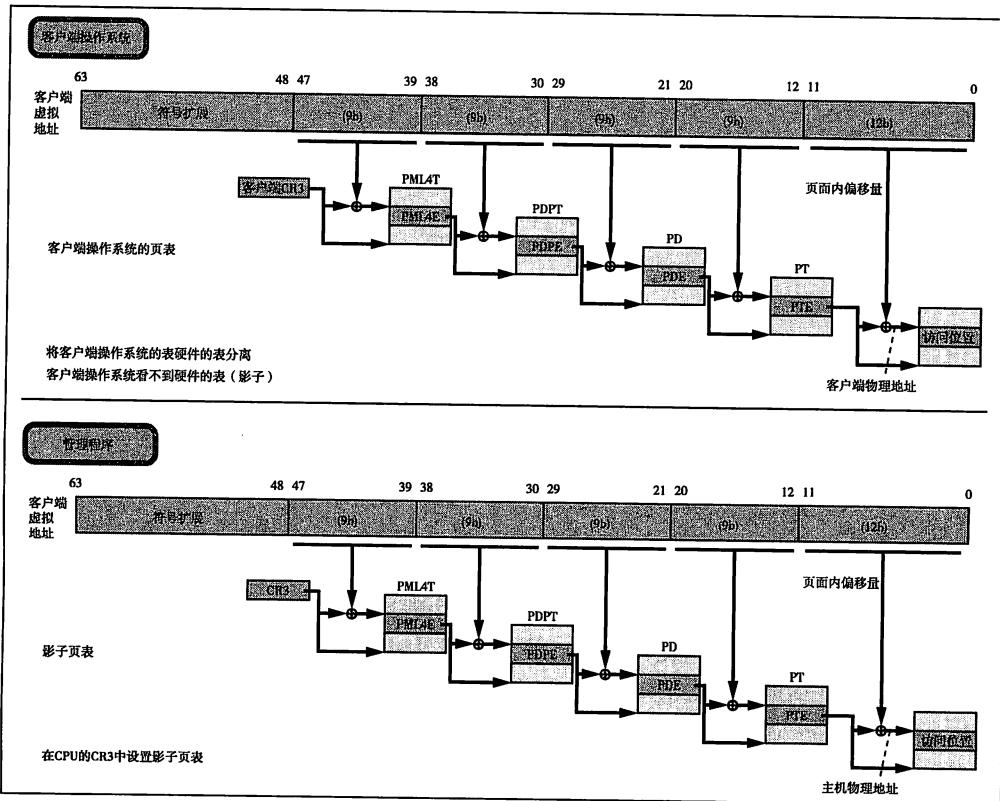


图 5-14 影子分页

## EPT

EPT (Extended Page Table) 是 Intel 公司的 CPU 所拥有的硬件虚拟化支持功能, 如图 5-15 所示。通过在硬件中将 MMU 虚拟化, 就可以减少管理程序的系统开销, 提高客户端操作系统运行速度。AMD 公司的 CPU 中也有同样的 NPT(Nested Page Table) 功能。这些功能由于使用两种页表而称为 2 维 (2D) 页表。EPT 中准备了管理程序将客户端物理地址转换为主机物理地址的表。当启用 EPT 时, 在客户端操作系统运行时 CPU 使用这个表自动将客户端虚拟地址转换为主机物理地址。

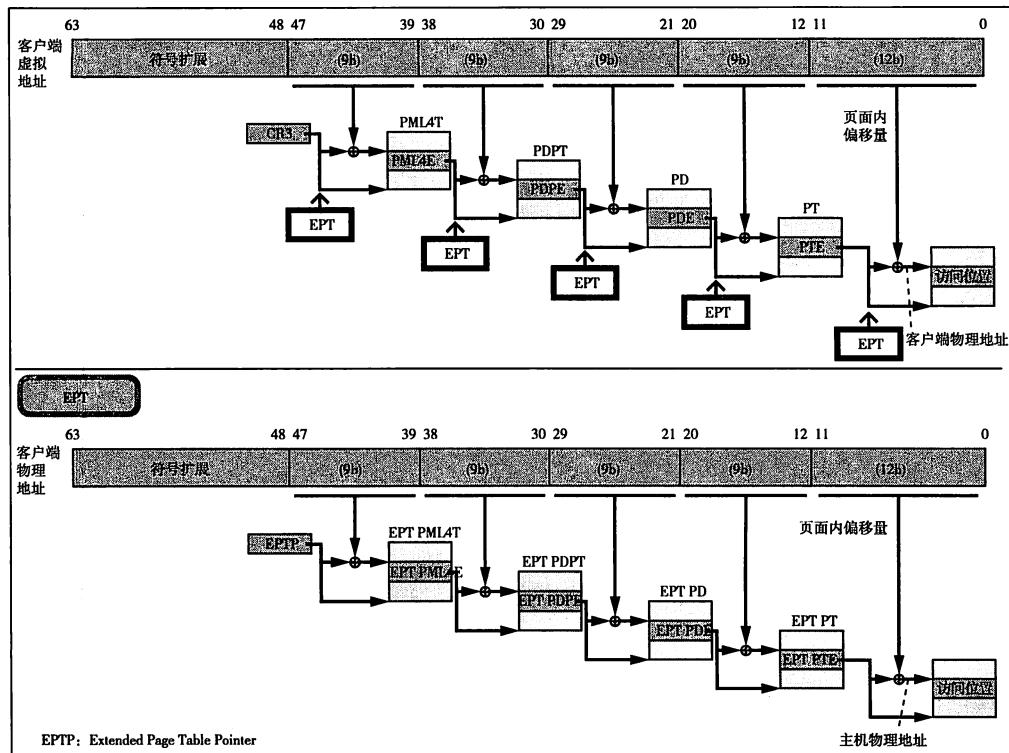


图 5-15 EPT

## 如何使用 EPT

当 CPU 拥有 EPT 时, /proc/cpuinfo 的 flags 会显示 ept。

```
processor      : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 26
model name    : Intel(R) Xeon(R) CPU          E5520  @ 2.27GHz
stepping      : 5
cpu MHz       : 2266.651
cache size    : 8192 KB
physical id   : 0
```

```

siblings      : 8
core id       : 0
cpu cores     : 4
apicid        : 0
initial apicid : 0
fpu           : yes
fpu_exception  : yes
cpuid level   : 11
wp             : yes
flags          : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_
tsc aperf mperf pni dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtrpr pdcm dca
sse4_1 sse4_2 popcnt lahf_lm ida dts tpr_shadow vnmi flexpriority ept vpid
bogomips      : 4533.30
clflush size   : 64
cache_alignment : 64
address sizes  : 40 bits physical, 48 bits virtual
power management:

```

在 RHEL6 等 Linux 的最新发布版中基本上都启用了 EPT。因此不需要特别的操作就可以使用 EPT。在使用 KVM 的情况下，如果 `/sys/module/kvm_intel/parameters/ept` 为 Y，就已启用 EPT。禁用 EPT 时显示为 N。

```
# cat /sys/module/kvm_intel/parameters/ept
Y
```

要禁用 EPT，可以把 `kvm_intel` 的模块参数指定为 `ept=0`，使用 `modprobe` 命令将其安装到内核中。

```
# modprobe kvm_intel ept=0
```

下面是禁用 EPT 时和启用 EPT 时 UnixBench 的比较。如图 5-16 所示，从 UnixBench 的结果也可以看出 EPT 的速度非常快。尤其是 *Process Creation*（进程生成）等性能得到大幅提高。

客户端操作系统的构成	
CPU 数量	2 (Intel® Xeon® CPU E5520 @2.27GHz)
内存	2GB
操作系统	RHEL6

图 5-17 中的图表是 lmbench 的结果。`*Local* Communication bandwidths in MB/s` 的值越大表示性能越高，其他的基准测试都是较小的值，表示性能较高。从 lmbench 的结果也能看出 EPT 是能够有效提高性能的。

在 lmbench 的结果中，还有 *Context switching* 等在禁用 EPT 时性能较高的项目。这表示 EPT 容易引起 TLB 错误。发生 TLB 错误时，在影子分页的情况下最多发生 5 次内存访问，而在启用 EPT 时最多发生 24 次内存访问。EPT 消耗的 TLB 很多，因此在容易发生 TLB 错误的基准测试中，影子分页的性能更高。

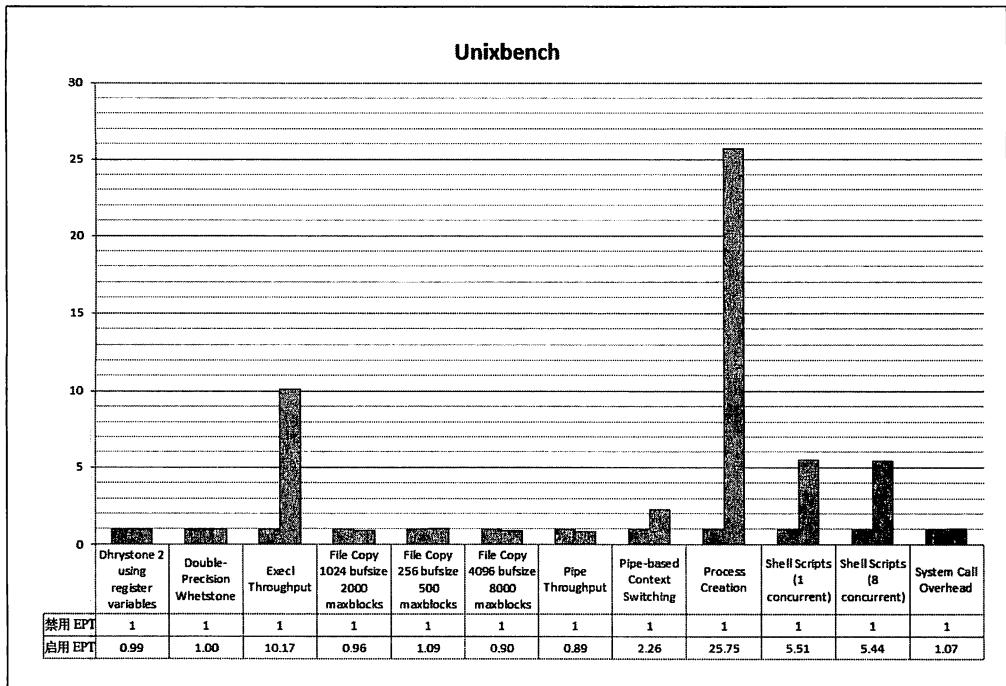


图 5-16 UnixBench 的结果

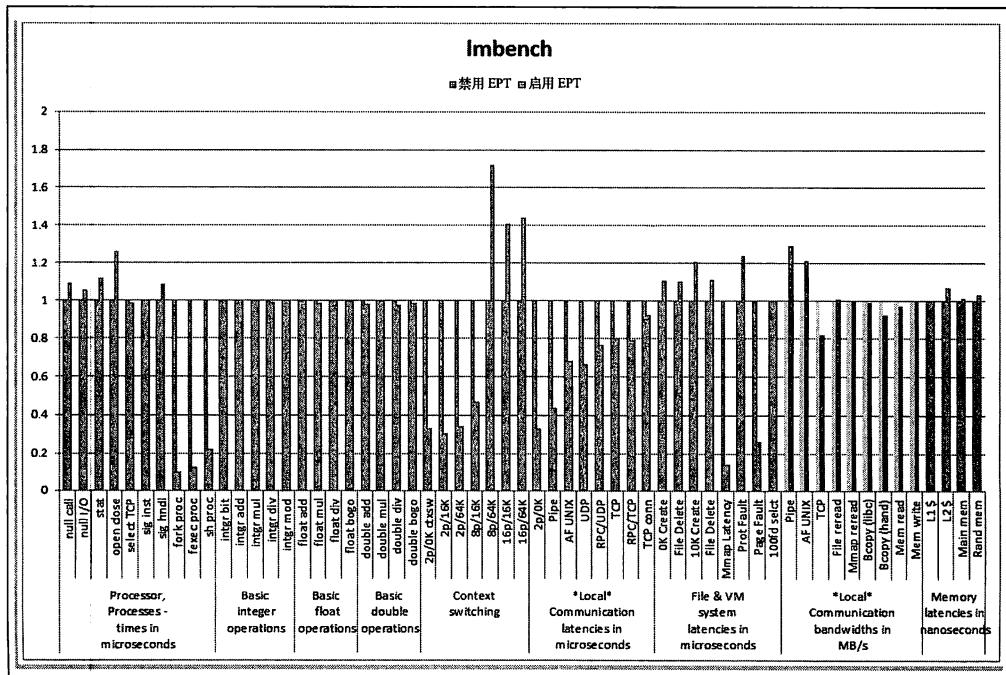


图 5-17 lmbench 的结果

## EPT+hugepage

可以使用 hugepage 来减轻 EPT 的 TLB 错误。下面介绍在 KVM 中使用 hugepage 的方法。hugepage 在使用影子分页的情况下也能提高性能。

1. 生成 hugepage。

```
# mkdir /hugepages
# mount -t hugetlbfs hugetlbfs /hugepages/
# echo 4096 > /proc/sys/vm/nr_hugepages
```

2. 重新启动 libvirtd。

```
# service libvirtd restart
```

3. 使用 virsh edit, 将客户端的结构文件改写为使用 hugepage 的设置。

在 memory 标签下添加 memoryBacking、hugepage 的标签。

```
# virsh edit <客户端操作系统名>
<domain type='kvm' id='1'>
  <name>rhel6-4</name>
  <uuid>983f2e4e-baa8-e5d7-a97c-b574a72484b8</uuid>
  <memory>1048576</memory>
  <currentMemory>1048576</currentMemory>
  <memoryBacking>
    <hugepages/>
  </memoryBacking>
  <vcpu cpuset='0,2'>2</vcpu>
  <os>
    <type arch='x86_64' machine='rhel6.0.0'>hvm</type>
    <boot dev='hd' />
  </os>
  ... snip ...
```

4. 启动客户端操作系统。

```
# virsh start <客户端操作系统名>
```

## 小结

本节介绍了 EPT 的性能特性。一般认为使用 EPT 时的性能较高。但是正如本节介绍的，在极少的环境下 EPT 性能也较低。在这种情况下需要理解 EPT 的特性，灵活使用 EPT 和影子分页。

## 参考文献

Intel®64 and IA-32 Architectures software Developer's Manual

——kio Takebe

# HACK #33 使用 IOMMU 提高客户端操作系统运行速度

本节介绍 IOMMU 功能，并说明如何提高客户端操作系统运行速度。

## 虚拟环境下客户端操作系统的 I/O 方式

本节将介绍在虚拟化环境下客户端 I/O 是通过什么样的方式进行的。Xen 或 KVM 等管理程序中主要使用的有下列三种方式。

1. 仿真方式（见图 5-18）
2. 半虚拟化方式（见图 5-19）
3. 直接 I/O 方式（见图 5-20）

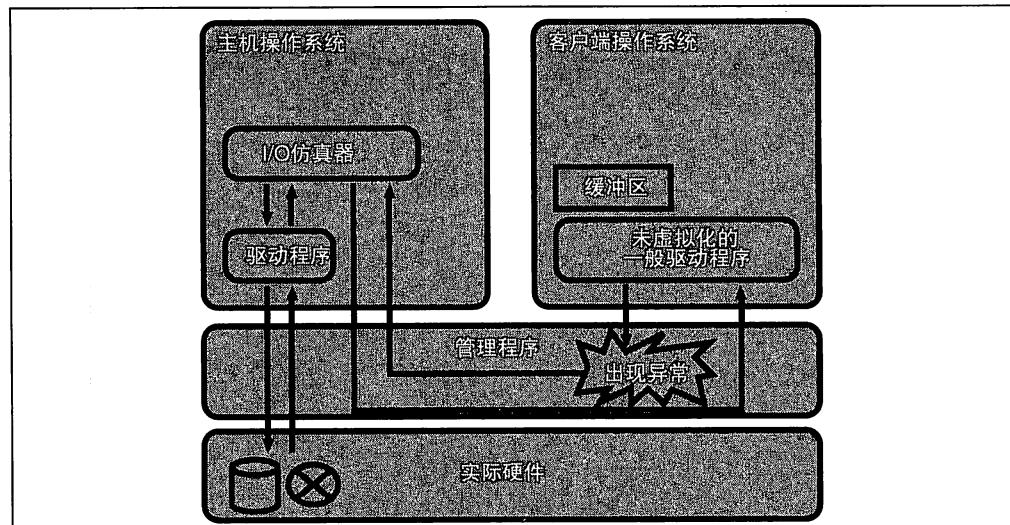


图 5-18 仿真方式

仿真方式是指使用仿真的 PCI 设备的 I/O 方式。从客户端操作系统看起来仿真设备与一般设备是同样的。Xen 或 KVM 在 I/O 仿真中采用的是 qemu，因此可以通过 qemu 仿真的设备都可以使用。使用这个方式时，客户端操作系统完全不需要考虑虚拟化，因此已有的设备驱动程序也可以直接使用，是这种方式的优点之一。但是进行 I/O 时必须进行下列仿真。仿真在 I/O 寄存器层进行，因此针对客户端操作系统的一次 I/O 请求需要进行多次的仿真，就会造成系统额外开销过大，I/O 速度缓慢。

1. 客户端操作系统发布 I/O 命令。
2. 管理程序进行捕捉（trap）。
3. 管理程序分析客户端操作系统发布的命令。
4. 如果是 I/O 命令，则通知主机操作系统上的 I/O 仿真器。

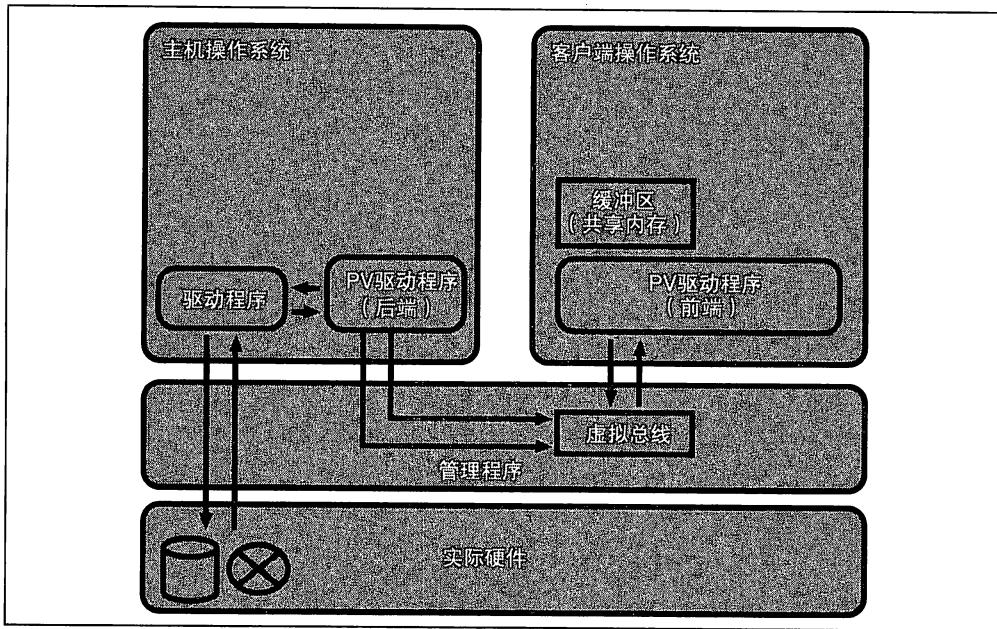


图 5-19 半虚拟化方式

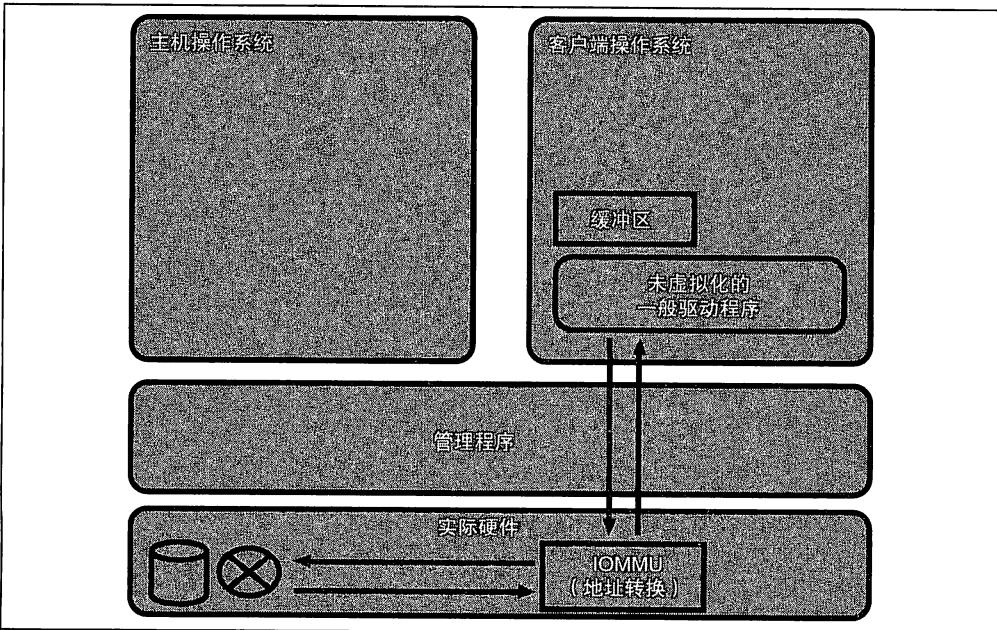


图 5-20 直接 I/O 方式

5. I/O 仿真器仿真 I/O 命令。
6. 如果需要实际的 I/O 发布，则 I/O 仿真器作为主机操作系统上的应用程序发布系统调用，实施 I/O。

7. I/O 仿真器一旦完成 I/O，就会进行设置中断位 (interrupt bit)、存取数据等的仿真。
8. 通知管理程序 I/O 完成。
9. 管理程序将中断提交给客户端操作系统。

半虚拟化方式是指使用称为 Para Virtual 驱动程序 (PV 驱动程序) 或 Split 驱动程序的方式。PV 驱动程序是指利用客户端操作系统使用的前端驱动程序和主机操作系统使用的后端驱动程序进行客户端 I/O 的方式。这个方式需要根据磁盘或网络、控制台等的 I/O 种类来生成驱动程序，但是可以非常快速地进行 I/O 处理。使用这种方式，可以减少客户端操作系统发出每一次 I/O 请求时必须进行的处理数量，从而提高 I/O 速度。但是，实际发出 I/O 请求的是主机操作系统或管理程序，因此这部分系统开销是仍然存在的。

在这种情况下，就出现了能够将这部分系统开销也消除的硬件。这就是 Intel 公司的 Virtualization Technology for Directed I/O (VT-d) 和 AMD 公司的 AMD IOMMU。Xen 或 KVM 中安装了称为 PCI 设备的直接 I/O 方式，就是运用这些硬件的。后面将介绍什么是直接 I/O 方式，以及如何使用。

## 关于 DMA

在介绍直接 I/O 方式前，先温习一下 DMA (Direct Memory Access) 传输。DMA 传输是指可以在内存和 I/O 之间直接进行数据交换的结构 (如图 5-21 所示)。

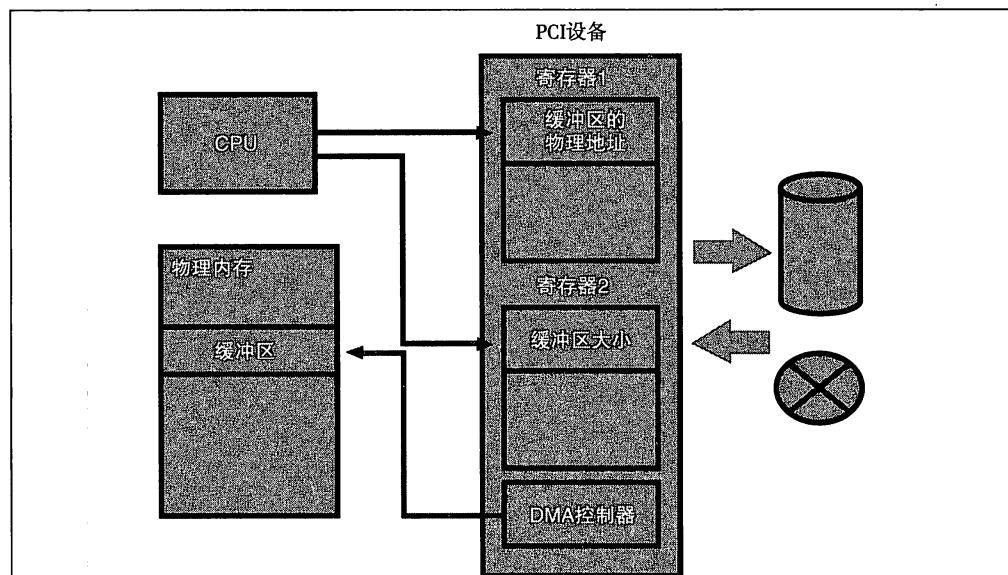


图 5-21 Native 操作系统的 DMA 传输

现在的 PCI 设备大多数都具有称为总线控制 DMA 方式的传输结构。DMA 传输一般按照如下流程进行。

1. 确保 DMA 传输用的内存。
2. 在设备的寄存器中确保第 1 步设置中的 DMA 传输用内存的地址。
3. 在设备的寄存器中确保第 1 步设置中的 DMA 传输用内存的大小。
4. 向 DMA 控制器请求开始在设备和内存之间进行数据传输。
5. DMA 传输完成后，设置 DMA 完成标志，将中断提交给 CPU。

第 2 步中设置的设备的寄存器地址就是物理内存的地址。第 4 步处理完成后，设备就会对所设置的物理内存进行数据的访问。这时物理内存和设备不在由 CPU 直接进行数据交换。

## IOMMU

IOMMU 是指 I/O 设备的 MMU (Memory Management Unit)，如图 5-22 所示。在虚拟化环境下使用 IOMMU，就可以直接从客户端操作系统发出 I/O 命令。

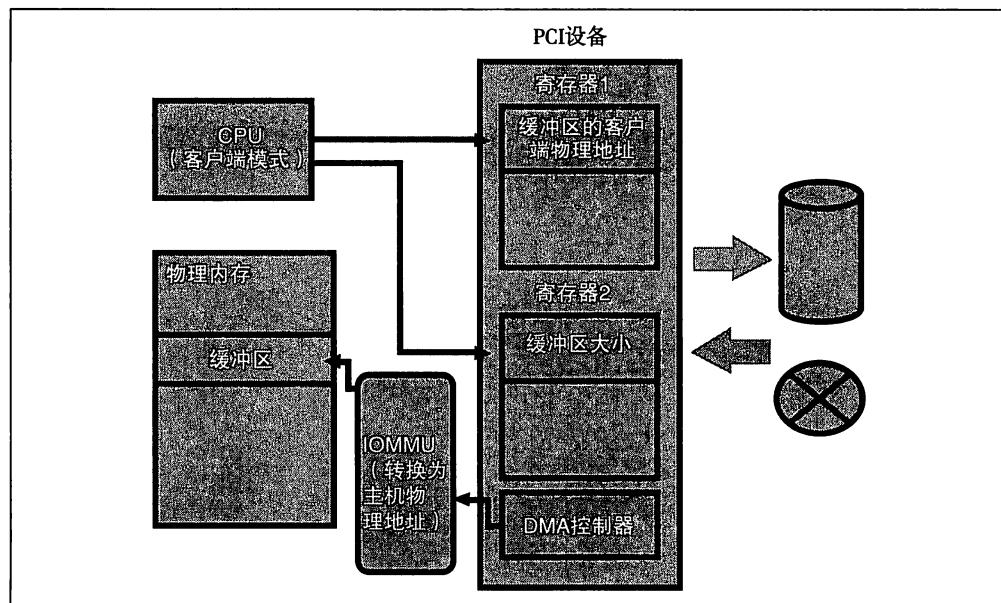


图 5-22 IOMMU 的运行

在 Xen/KVM 等虚拟化环境下，客户端操作系统无法识别主机物理内存地址。客户端操作系统识别通过管理程序虚拟化的客户端物理地址来运行。因此 PCI 设备的 DMA 传输就需要将客户端物理内存地址转换为主机物理内存地址的结构。

IOMMU 就是用来进行这个转换的硬件结构。客户端操作系统在 PCI 设备的寄存器中设置客户端物理内存地址，并向设备请求进行 DMA 传输。而设备则向所设置的物理内存地址写入数据，但将这个客户端物理内存地址转换为主机物理内存地址的任务是由 IOMMU 来承担的。

从客户端物理内存地址转换为主机物理内存地址时，管理程序必须事先在 IOMMU 中设置每个 PCI 设备的页表。在这个页表中，将使 PCI 设备的客户端操作系统的客户端物理内存地址和主机物理内存地址进行对应。

通过这样的方式使用 IOMMU，就可以不经由主机操作系统或管理程序，在客户端操作系统和 I/O 设备之间进行 DMA 数据传输，因此将使用 IOMMU 的方式称为直接 I/O 方式。另外，使用 IOMMU 时，所分配的设备只能从该客户端操作系统进行操作。PCI 设备的控制直接被传递到客户端操作系统，因此也称为 PCI 传递 (pass through) 方式。

## KVM 的 IOMMU 的使用方法

下面介绍 KVM 的 IOMMU 方法。本节以 Intel VT-d 的使用方法为中心进行说明。PCI 传递通过下列流程来设置 (如图 5-23 所示)。

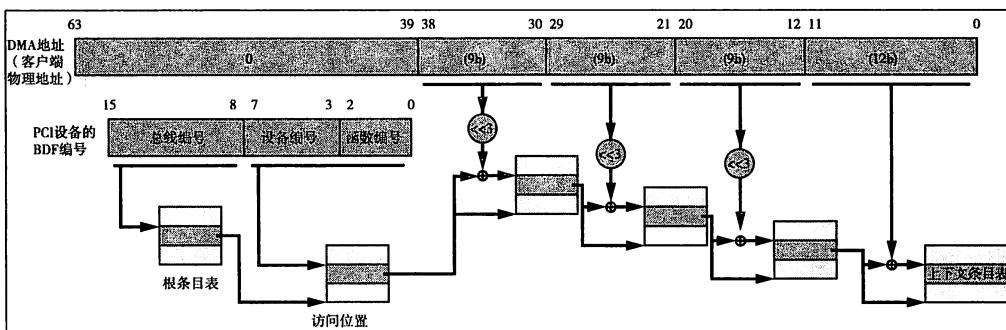


图 5-23 在 IOMMU 中设置的页表

1. 在 BIOS 中启用 VT-d。
2. 在内核启动选项中启用 VT-d。
3. 决定 PCI 传递的设备。
4. 从主机操作系统拆除设备。
5. 向客户端操作系统分配设备。

首先，为了检查系统是否可以使用 IOMMU，需要检查 ACPI 的表。在 Intel VT-d 中，使用 `dmesg` 按照下列方式来检查 ACPI 表中是否存在签名为 DMAR 的表。如果没有 DMAR，请回到 BIOS，启用 VT-d。

```
# dmesg | grep DMAR
ACPI: DMAR 00000000bf7cadb4 000E8 (v01 Intel OEMDMAR 00060000 LOHR 00000001) ★
DMAR: Host address width 39
DMAR: DRHD base: 0x0000000fed14000 flags: 0x1
DMAR: RMRR base: 0x0000000bf7df000 end: 0x0000000bf7effff
DMAR: ATSR flags: 0x0
```

使用 VT-d 时，需要将 `intel_iommu` 内核启动选项设置如下。也像 Fedora 13 等这样默

认为 on 的发布版。

```
default=0
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title Red Hat Enterprise Linux (2.6.32-71.el6.x86_64)
    root (hd0,0)
    kernel /vmlinuz-2.6.32-71.el6.x86_64 ro root=UUID=234a4e64-a785-48b7-
a8a9-
fd88428d62fe
rd_NO_LUKS rd_NO_LVM rd_NO_MD rd_NO_DM LANG=ja_JP.UTF-8 KEYBOARDTYPE=pc
KEYTABLE=jp106 rhgb quiet intel_iommu=on
    initrd /initramfs-2.6.32-71.el6.x86_64.img
```

然后决定将哪个设备 PCI 传递到哪个客户端操作系统。决定 PCI 传递的设备后，使用 `lscpi` 命令检查 PCI 的 BDP 编号 (Bus:Device.Function 编号)。

```
00:00.0 Host bridge: Intel Corporation 5520 I/O Hub to ESI Port (rev 13)
00:01.0 PCI bridge: Intel Corporation 5520/5500/X58 I/O Hub PCI Express Root
Port 1 (rev 13)
00:03.0 PCI bridge: Intel Corporation 5520/5500/X58 I/O Hub PCI Express Root
Port 3 (rev 13)
... snip ...
00:1e.0 PCI bridge: Intel Corporation 82801 PCI Bridge (rev 90)
00:1f.0 ISA bridge: Intel Corporation 82801JIB (ICH10) LPC Interface
Controller
00:1f.2 IDE interface: Intel Corporation 82801JI (ICH10 Family) 4 port SATA
IDE Controller #1
00:1f.3 SMBus: Intel Corporation 82801JI (ICH10 Family) SMBus Controller
00:1f.5 IDE interface: Intel Corporation 82801JI (ICH10 Family) 2 port SATA
IDE Controller #2
01:00.0 RAID bus controller: LSI Logic / Symbios Logic MegaRAID SAS 1078 (rev
04)
05:00.0 Fibre Channel: Emulex Corporation Zephyr LightPulse Fibre Channel Host
Adapter (rev 02)
06:00.0 SCSI storage controller: LSI Logic / Symbios Logic SAS1068E PCI-
Express Fusion-MPT SAS (rev 08)
07:00.0 Fibre Channel: Emulex Corporation Zephyr LightPulse Fibre Channel Host
Adapter (rev 02)
08:00.0 Ethernet controller: Intel Corporation 82575EB Gigabit Network
Connection (rev 02)
08:00.1 Ethernet controller: Intel Corporation 82575EB Gigabit Network
Connection (rev 02)
09:00.0 VGA compatible controller: Matrox Graphics, Inc. MGA G200e [Pilot]
ServerEngines (SEP1) (rev 02)
... snip ...
```

例如，假设将 08:00.0 的以太网控制器进行 PCI 传递。

```
08:00.0 Ethernet controller: Intel Corporation 82575EB Gigabit Network
Connection (rev 02)
```

还需要得知这个设备的 PCI 设备的厂商 ID、设备 ID。

```
# lspci -n -s 0000:08:00.0
08:00.0 0200: 8086:10a7 (rev 02)
```

在找出 BDF 编号和厂商 ID、设备 ID 后，将这个设备从主机操作系统上拆除。拆除的方法如下所示。不具有 FLR (Function-Level Reset)<sup>注6</sup> 功能的多功能设备在向客户端操作系统分配时，会重启所有的功能设备，因此需要将这些功能设备全部拆除。

```
# echo "8086 10a7" > /sys/bus/pci/drivers/pci-stub/new_id
# echo 0000:08:00.0 > /sys/bus/pci/devices/0000:08:00.0/driver/unbind
# echo 0000:08:00.0 > /sys/bus/pci/drivers/pci-stub/bind
# echo "8086 10a7" > /sys/bus/pci/drivers/pci-stub/new_id
# echo 0000:08:00.1 > /sys/bus/pci/devices/0000:08:00.1/driver/unbind
# echo 0000:08:00.1 > /sys/bus/pci/drivers/pci-stub/bind
```

最后指定 PCI 传递的 PCI 设备，使用 KVM 启动客户端操作系统。

```
# qemu-kvm -smp 2 -m2048 -drive file=/var/lib/libvirt/images/f13-kvm.
img,if=virtio,boot=on -pcidevice host=08:00.0
```

从客户端操作系统可以看到传递到 00:05.0 的 PCI 设备。

```
# lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 03)
00:02.0 VGA compatible controller: Cirrus Logic GD 5446
00:03.0 Ethernet controller: Realtek Semiconductor Co., Ltd. RTL-
8139/8139C/8139C+ (rev 20)
00:04.0 SCSI storage controller: Qumranet, Inc. Virtio block device
00:05.0 Ethernet controller: Intel Corporation 82575EB Gigabit Network
Connection (rev 02)
```

可以使用 qemu 的控制台时，在打开客户端操作系统的 VGA 控制台界面后，使用 Ctrl+Alt+2 快捷键切换到 qemu 控制台。在 qemu 控制台上，还可以使用下列命令将 PCI 设备动态添加到客户端操作系统中。

```
(qemu) pci_add pci_addr=auto host host=08:00.0
```

将 PCI 设备动态删除需进行如下操作。向 pci\_addr 指定的编号为在客户端操作系统上看到的 PCI 设备的 Bus 编号。

```
(qemu) pci_del pci_addr=5
```

在 RHEL6 或 Fedora13 等发布版中，可以使用 virt-manager 通过 PCI 传递 (pass-through) 将设备添加到客户端操作系统或者从客户端操作系统中删除（见图 5-24）。

---

注 6：以 PCI 设备的 Function 为单位重启的功能。



图 5-24 通过 RHEL6 的 PCI 传递添加设备

## Xen 的 IOMMU 的使用方法

在 Fedora 13 中, Xen 并未作为正式的工具包发布, 因此需要从以下网页下载 RPM 包。

Dom0 内核的 RPM

<http://repos.fedorapeople.org/repos/myoung/dom0-kernel/>

Xen 的其他 RPM

<http://ftp.funet.fi/pub/mirrors/fedora.redhat.com/pub/fedora/linux/updates/testing/14/>

要启用 VT-d, 需要在 Xen 中将选项 `iommu=1` 指定为管理程序的启动选项。下面是 `grub.conf` 的示例。

```
default=0
timeout=10
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
  root (hd0,0)
  kernel /xen.gz iommu=1 dom0_mem=2G
  module /vmlinuz-2.6.32.23-170.xendom0.fc12.x86_64 ro root=UUID=293a5a8e-9ddde-40a4-9356-5c8492d70eaa rd_NO_LUKS rd_NO_LVM rd_NO_MD rd_NO_DM LANG=ja_JP.UTF-8 KEYTABLE=jp106 rhgb
  module /initramfs-2.6.32.23-170.xendom0.fc12.x86_64.img
```

将分配给主机操作系统的设备从主机操作系统拆除的方法与 KVM 相同。这种情况下拆除方法如下。

```
# echo "8086 10a7" > /sys/bus/pci/drivers/pci-stub/new_id
# echo 0000:08:00.0 > /sys/bus/pci/devices/0000:08:00.0/driver/unbind
# echo 0000:08:00.0 > /sys/bus/pci/drivers/pci-stub/bind
# echo "8086 10a7" > /sys/bus/pci/drivers/pci-stub/new_id
# echo 0000:08:00.1 > /sys/bus/pci/devices/0000:08:00.1/driver/unbind
# echo 0000:08:00.1 > /sys/bus/pci/drivers/pci-stub/bind
```

在 Xen 中可以使用下列命令检查可分配的设备。

```
# xm pci-list-assignable-devices
0000:08:00.0
0000:08:00.1
```

在 Xen 中，客户端操作系统的配置文件可以通过设置下列选项，将 PCI 设备分配给客户端操作系统。

```
pci = [ '08:00.0' , '08:00.1' ]
```

分配给客户端操作系统的 PCI 设备默认依次分配到空的虚拟插槽，但如果使用 @<slot号> 选项，就可以指定要分配的插槽。

```
pci = [ '08:00.0' , '08:00.1@7' ]
```

在本次使用的 PCI 这种 Multi-Function 设备中，有一些必须从操作系统角度看也是 Multi-Function 设备，才能顺利运行。USB 控制器或显卡类设备尤其需要注意。

```
pci = [ '08:00.*' ]
```

下面是将 Multi-Function 设备直接作为 Multi-Function 设备分配给客户端操作系统的设置示例。

```
import os, re
arch_libdir = 'lib'
arch = os.uname()[4]
if os.uname()[0] == 'Linux' and re.search('64', arch):
    arch_libdir = 'lib64'
kernel = "/usr/lib/xen/boot/hvmloader"
builder='hvm'
memory = 2048
name = "f13-hvm"
vcpus=4
pae=1
acpi=1
apic=1
vif = [ '' ]
disk = [ 'file:/var/lib/xen/images/f13-hvm.img,hda,w', 'file:/root/Fedora-13-
x86_64-DVD.iso,hdc:cdrom,r' ]
device_model = '/usr/' + arch_libdir + '/xen/bin/qemu-dm'
boot="cda"
sdl=0
opengl=1
vnc=1
```

```
vnclisten="0.0.0.0"
vncpasswd=''
stdvga=0
serial='pty'
keymap='ja'
xen_platform_pci=1
pci=[ '08:00.*@6' ]
pci_msitranslate=1
```

使用上述配置文件，启动客户端操作系统。

```
# xm create /etc/xen/f13-hvm
```

```
# xm pci-list f13-hvm
VSlt VFn domain bus slot func
0x06 0x0 0x0000 0x08 0x00 0x0
0x06 0x1 0x0000 0x08 0x00 0x1
```

在客户端操作系统上确认 PCI 设备，可以看到如下结果。

```
# lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 01)
00:02.0 VGA compatible controller: Cirrus Logic GD 5446
00:03.0 Class ff80: XenSource, Inc. Xen Platform Device (rev 01)
00:04.0 Ethernet controller: Realtek Semiconductor Co., Ltd. RTL-8139/8139C/8139C+ (rev 20)
00:06.0 Ethernet controller: Intel Corporation 82575EB Gigabit Network Connection (rev 02)
00:06.1 Ethernet controller: Intel Corporation 82575EB Gigabit Network Connection (rev 02)
```

在笔者手头的客户端操作系统上设置 NIC，就可以进行网络通信了。

进行动态删除时，需在 Dom0 上执行下列命令。

```
# xm pci-detach f13-hvm "08:00.*"
```

进行动态添加时，需在 Dom0 上执行下列命令。

```
# xm pci-attach f13-hvm "08:00.*"
```

## 小结

VT-d 的基本功能是 DMA Remapping，同时也具有 IRQ Remapping 等功能。通过使用 IOMMU，就可以提高客户端之间的隔离性，同时安全性也得到加强。但是在没有 IOMMU 的 PCI 设备中，有一些是用于识别固件物理内存地址的，运行可能会异常。Xen 的论坛就收集了 SCSI 的 PCI 设备或图形的 PCI 设备中出现问题的报告。NIC 中这种设备较少，DMA Remapping 的效果也较好，因此如果想要尝试使用 VT-d，推荐首先在 NIC 中尝试。

另外，VT-d 的优点是可以发挥出与物理环境相同的性能，但同时也存在客户端操作系统

无法动态迁移（live migration）的缺点。在使用 PCI 传递时，请充分考虑这些优缺点。

## 参考文献

- Intel®Virtualization Technology for directed I/O  
[http://download.intel.com/technology/computing/vptech/intel\(r\)\\_VT\\_for\\_Direct\\_IO.pdf](http://download.intel.com/technology/computing/vptech/intel(r)_VT_for_Direct_IO.pdf)
- [http://www.linux-kvm.org/page/Hotadd\\_pci\\_devices](http://www.linux-kvm.org/page/Hotadd_pci_devices)
- <http://wiki.xensource.com/xenwiki/Fedora13Xen4Tutorial>
- <http://wiki.xensource.com/xenwiki/VTdHowTo>

——Akio Takebe

# HACK #34 使用 IOMMU+SR-IOV 提高客户端操作系统速度

本节介绍 SR-IOV 功能以及在虚拟环境下使用 SR-IOV 的方法。

## SR-IOV

以往的 PCI 设备必须经由管理程序或设备仿真器，才能在多个客户端操作系统上共享一个设备。x86 中由于出现了 IOMMU，就可以增加客户端操作系统使用 PCI 设备的机会。为了应对这种虚拟化环境，就出现了 SR-IOV (Single Root I/O Virtualization)。SR-IOV 是一个 PCI Express 设备向操作系统提供多个虚拟设备的功能。通过这个功能，就可以在管理程序中得到很多虚拟设备，因此就更容易使客户端操作系统使用 PCI 设备。

## SR-IOV 的功能

SR-IOV 是 PCI Express 的功能。在各 PCI Express 设备中，都安装了这个功能，作为 PCI Express 扩展功能。SR-IOV 是让操作系统将一个设备识别为多个虚拟设备的功能。在 SR-IOV 中，将以往以物理方式存在的设备称为 Physical Function (PF)。在 PF 中可以进行 SR-IOV 的设置。通过 SR-IOV 功能虚拟添加的设备称为 Virtual Function (VF)。

### Intel 82576 Gigabit Ethernet Controller

SR-IOV 对应的 PCI Express 卡有 Intel 82576 千兆以太网控制器等。在 Linux 中，可以使使用 `lspci` 命令来显示 SR-IOV 的信息。下面是 Intel 82576 的 `lspci -vvvv` 的结果。

```
02:00.0 Ethernet controller: Intel Corporation 82576 Gigabit Network
Connection (rev 01)
... snip ...
    Capabilities: [160] Single Root I/O Virtualization (SR-IOV)
        IOVCap: Migration-, Interrupt Message Number: 000
        IOVCtl: Enable- Migration- Interrupt- MSE- ARIHierarchy+
```

```
IOVSta: Migration-
        Initial VFs: 8, Total VFs: 8, Number of VFs: 8, Function
Dependency Link: 00
... snip ...
```

Capabilities: [160] Single Root I/O Virtualization (SR-IOV) 的部分就是 SR-IOV 的信息。

Initial VFs: VF 数的初始值。在 SR-IOV 中是与 Total VFs 相同的值。

Total VFs: 可以启用的最大 VF 数。

Number of VFs: 当前启用的 VF 数。

## 尝试启用 Intel 82576 的 VF

如图 5-25 所示, 要启用 Intel 82576 的 VF, 可以在使用 modprobe 将驱动程序安装到内核时指定 max\_vfs 选项。向 max\_vfs 选项指定希望启用的 VF 数量。VF 从操作系统方面来看是普通的 PCI 设备, 因此可以使用 lspci 来确认已启用的 VF。另外, 要使用 SR-IOV, BIOS 必须也支持该功能。如果 BIOS 不支持, 就会导致 VF 无法启用或只能对一部分 PF 启用 VF。

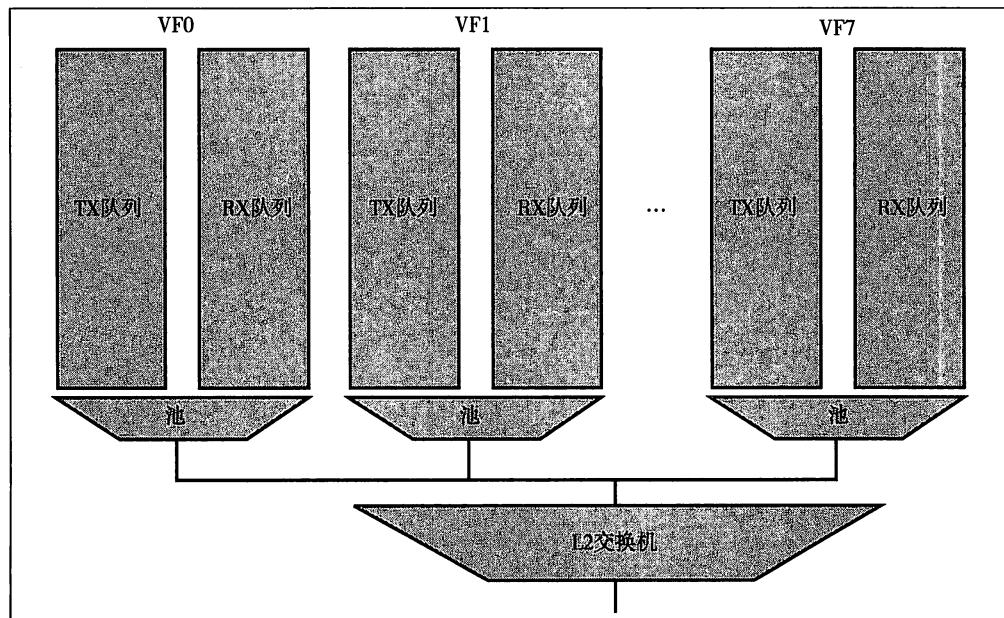


图 5-25 Intel 82576 内的 L2 交换机

```
# lspci | grep 82576
02:00.0 Ethernet controller: Intel Corporation 82576 Gigabit Network
Connection (rev 01)
# rmmod igb ; modprobe igb max_vfs=1
# lspci | grep 82576
02:00.0 Ethernet controller: Intel Corporation 82576 Gigabit Network
Connection (rev 01)
02:10.0 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
Intel 82576 最大能够提供 8 个 VF。使用了 SR-IOV 时, 把 PF 的通信带宽分配给各 VF。
```

Intel 82576 只有在每个 PF 上有物理端口，因此所有 VF 共享一个 PF 端口。PF 内部有 L2 交换机，VF 经由 L2 交换机进行通信。发送 (TX) 队列和接收 (RX) 队列统称为池 (pool)。把池分配给各 VF，在各个池中可以设置 MAC 过滤器和 VLAN 过滤器。

## 在 KVM 中使用 SR-IOV 的方法

在 RHEL6 中，通过 `virt-manager` 将 VF 指定为 PCI 传递设备，就可以在客户端操作系统使用 VF。在这里传递的是 02:11.4 的 VF。

```
# modprobe -r igb; modprobe igb max_vfs=7
# lspci
02:00.0 Ethernet controller: Intel Corporation 82576 Gigabit Network
Connection (rev 01)
02:00.1 Ethernet controller: Intel Corporation 82576 Gigabit Network
Connection (rev 01)
02:10.0 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
02:10.2 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
02:10.4 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
02:10.6 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
02:11.0 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
02:11.2 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
02:11.4 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
```

如图 5-26 所示，启动 `virt-manager`，选择“编辑”→“虚拟机详细”。在弹出的窗口上，选择“显示”→“详细”，就会显示虚拟机的硬件详细界面。这里在“添加硬件”中将硬件类型选择为物理主机设备。然后选择 02:11.4 的 VF，单击“完成”按钮（如图 5-27 所示）。

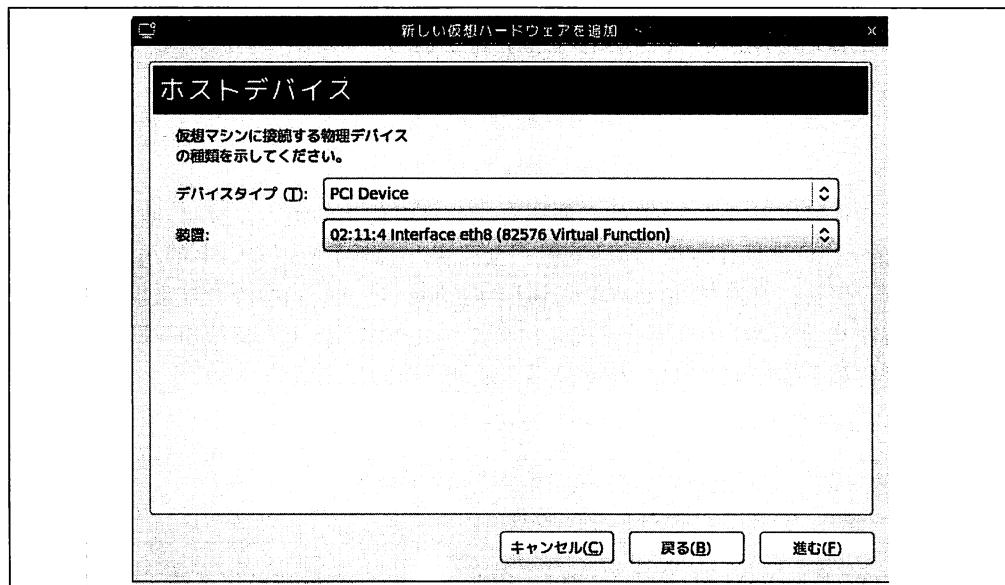


图 5-26 virt-manager 的设置界面

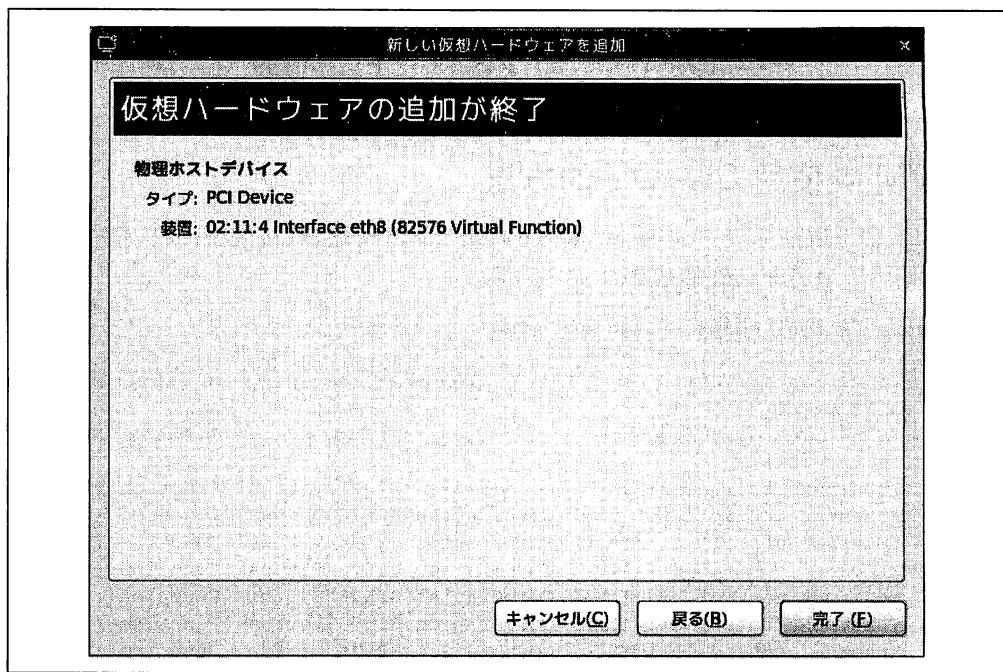


图 5-27 virt-manager 的完成界面

启动客户端操作系统后，在客户端操作系统上对 `igbvf` 模块执行 `modprobe` 命令，安装到内核。

```
# modprobe igbvf
```

这样就可以在客户端操作系统上使用 VF 了。由于使用了 IOMMU，因此使用 VF 的客户端操作系统基本不借助管理程序或主机操作系统就可以进行网络通信。

## 小结

这里介绍了 VF 的使用方法。VF 的功能还有一些难以使用的地方。MAC 地址就是其中之一。VF 的 MAC 地址在每次启用 VF 时都会改变。这对于使用 DHCP 的用户来说是非常麻烦的事。但是，在使用 libvirt 等工具时会向 `qemu` 的 `mac` option 分配虚拟 MAC，就不会出现每次启动 MAC 都改变的情况。如果在主机操作系统上直接使用 VF，在主机操作系统上进行如下操作就可以将 VF 的 MAC 地址更改为任意值。

```
ip link set dev <ethN> vf 0 mac <MAC 地址 >
```

---

**小贴士：**通常在更改 MAC 地址时，可以使用如下所示的 `ip` 命令。

```
ip link set dev <ethN> address <MAC 地址 >
```

---

另外，SR-IOV 是与 IOMMU 配合使用的，因此就会产生客户端操作系统无法迁移的限制。单独在迁移时使用 `vhost-net`，就可以解决这个问题。从将来的发展趋势来看，将能

够实现使用 Multi Root I/O Virtualization (MR-IOV) 硬件功能来进行迁移。

——Akio Takebe

## HACK #35 SR-IOV 带宽控制

本节介绍 Intel 82576 的带宽控制功能的使用方法。

### Intel 82576 的带宽控制

Intel 82576 是搭载了 SR-IOV 功能的 NIC。Intel 82576 可以在利用 SR-IOV 时使用带宽限制。Linux 2.6.39 以后的版本都可以使用这个带宽控制功能。带宽的设置是通过使用 iproute2 的 ip 命令来进行的。

### Intel 82576 的带宽控制的使用方法

ip 命令可以对 NIC 拥有的带宽进行设置，将其分配给各 VF。通过下列方法可以向 VF 分配任意带宽。这里分别为两个 VF 设置 200Mbps、800Mbps 的带宽。

1. 创建 VF。下面创建两个 VF。

```
# modprobe igb max_vfs=2
```

2. 确认 VF 已创建。

```
# lspci
... snip ...
01:00.0 Ethernet controller: Intel Corporation 82576 Gigabit Network
Connection (rev 01)    eth0
01:00.1 Ethernet controller: Intel Corporation 82576 Gigabit Network
Connection (rev 01)    eth1
01:10.0 Ethernet controller: Intel Corporation 82576 Virtual Function (rev
01)                  eth0 的 VF0
01:10.2 Ethernet controller: Intel Corporation 82576 Virtual Function (rev
01)                  eth0 的 VF1
... snip ...
```

3. 在主机 OS 上使用 ip 命令设置 VF 的带宽。将想要设置带宽的 VF 编号指定为 ip 参数中 vf 选项的变量。向 rate 选项指定带宽。这里为 VF0 设置的带宽是 200Mbps，为 VF1 设置的带宽是 800Mbps。

```
# ip link set eth0 vf 0 rate 200
# ip link set eth0 vf 1 rate 800
# ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen
1000
    link/ether 00:24:21:f1:e1:ec brd ff:ff:ff:ff:ff:ff
    vf 0 MAC de:c4:b4:78:00:1b, tx rate 200 (Mbps)  ☆
```

```

vf 1 MAC 8a:46:ef:e6:a8:e5, tx rate 800 (Mbps)      ☆
3: eth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000
   link/ether 00:24:21:f1:e1:ed brd ff:ff:ff:ff:ff:ff
6: virbr0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
   link/ether 52:54:00:93:33:79 brd ff:ff:ff:ff:ff:ff
7: virbr0-nic: <BROADCAST,MULTICAST> mtu 1500 qdisc noop master virbr0 state
DOWN qlen 500
   link/ether 52:54:00:93:33:79 brd ff:ff:ff:ff:ff:ff
9: vnet0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master
virbr0 state UNKNOWN qlen 500
   link/ether fe:54:00:74:e4:98 brd ff:ff:ff:ff:ff:ff
10: vnet1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master
virbr0 state UNKNOWN qlen 500
   link/ether fe:54:00:a7:05:b9 brd ff:ff:ff:ff:ff:ff

```

## 尝试测量带宽

然后实际测量一下确认带宽是否已得到了控制。这里创建了两个客户端 OS，分别向客户端 OS 分配刚才创建的 VF。测量带宽时使用的是 netperf。

### 测量环境

主机	
OS	RHEL6.1 64bit
内核	2.6.39-rc7(intel_iommu=on)
iproute2	iproute2-ss091226
客户端 1	
OS	RHEL6.1 64bit
内存	1G
VF	eth0 VF06
客户端 2	
OS	RHEL6.1 64bit
内存	1G
VF	eth0 VF1

netperf 服务器机器 192.168.0.200

测量结果如下所示。可以看出基本上达到了指定的性能。

#### 例 5-1 客户端 1 与 netperf 服务器之间的通信

```
# netperf -t UDP_STREAM -l 50 -H 192.168.0.200
UDP UNIDIRECTIONAL SEND TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
```

```

192.168.0.200 (192.168.0.200) port 0 AF_INET
Socket  Message  Elapsed  Messages
Size    Size    Time      Okay Errors  Throughput
bytes   bytes   secs      #      #  10^6bits/sec

124928   65507   50.00     18591      0      194.85  约 200Mbps
262144          50.00     18591          194.85

```

### 例 5-2 客户端 2 与 netperf 服务器机器之间的通信

```

# netperf -t UDP_STREAM -l 50 -H 192.168.0.200
UDP UNIDIRECTIONAL SEND TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
192.168.0.200 (192.168.0.200) port 0 AF_INET
Socket  Message  Elapsed  Messages
Size    Size    Time      Okay Errors  Throughput
bytes   bytes   secs      #      #  10^6bits/sec

124928   65507   50.00     73154      0      766.73  约 800Mbps
262144          50.00     73154          766.73

```

## 小结

本节介绍了使用 Intel 82576 的硬件结构控制带宽的方法。控制网络带宽的方法有不少，如果使用 SR-IOV 也可以选择设备所集成的功能进行尝试。

## 参考文献

Intel®82576EB Gigabit Ethernet Controller Datasheet

[http://download.intel.com/design/network/datashts/82576\\_Datasheet.pdf](http://download.intel.com/design/network/datashts/82576_Datasheet.pdf)

——Akio Takebe

## HACK #36 使用 KSM 节约内存

本节介绍共享相同内容的内存以节约内存的 KSM。

KSM (Kernel Samepage Merging)，是通过共享相同内容的存储页面，将其整合为一体，从而有效使用内存的功能。原始版本是在 KVM 上开发的 Kernel Shared Memory，从 Linux 2.6.32 开始合并到上游内核。在虚拟环境下启动多个相同 OS 映像的客户端 OS 时，就会出现相同内容的存储页面。在这样的情况下 KSM 就非常有效。

## 使用方法

编译内核时，需要设置为 CONFIG\_KSM=y。

KSM 会通过内核线程 ksmd 定期对用户内存进行扫描。如果为同一内容，则在 COW

(Copy On Write) 模式下合并该区域 (存储页面)。这个存储区更新时, 内核自动重新生成存储页面的副本。

要扫描的区域是通过 `madvise` 的 `MADV_MERGEABLE` 指定的内存, 并且匿名页面成为合并对象。

## sysfs

KSM 可以使用 `/sys/kernel/mm/ksm/` 下的特殊文件来进行设置 (见表 5-4)。

表 5-4 KSM 的设置

项 目	说 明
<code>pages_to_scan</code>	单次扫描的页数。体现在 KSM 通过 <code>MADV_MERGEABLE</code> 指定的页面扫描和确认是否可以合并时的扫描这二者当中
<code>sleep_millisecs</code>	<code>ksmd</code> 在下一次扫描前休眠的时间 (毫秒)
<code>run</code>	设置为 0~2 的值。0 为停止 <code>ksmd</code> 。到目前为止合并的页仍保持原状。1 为启动 <code>ksmd</code> 。2 为停止 <code>ksmd</code> 。到目前为止合并的页全部生成副本, 取消页面共享
<code>pages_shared</code>	共享后的页面数。例如, 如果 1000 页均为同一内容, 可以合并到 1 页, 则 <code>pages_shared</code> 的值为 1
<code>pages_sharing</code>	可共享的页面数。如果 1000 页可以合并为 1 页, 则 <code>pages_sharing</code> 的值为 1000
<code>pages_unshared</code>	当前未合并的页数。通过 <code>MADV_MERGEABLE</code> 指定的页且没有同一内容的页, 即, 不能合并的页数
<code>pages_volatile</code>	输入到 KSM 所管理的树中的页数。KSM 将 <code>MADV_MERGEABLE</code> 指定的页输入专用的管理树中。这个页数是扫描对象的页数, 还未确认是否可以合并。经过扫描, 确认是否合并后, 这个值就会减小
<code>full_scans</code>	从头至尾扫描合并区域的次数

可以认为, `pages_sharing` 与 `pages_shared` 的比值越高, KSM 的效果越明显。相反, 如果这个比值较低, 则表示即使进行扫描也无法进行页面合并。接下来, 在例子程序 (`mergeable.c`) 中确认运行情况。

```
# cat mergeable.c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h> // for madvise
```

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[])
{
    int fd = 0;
    char *file = NULL;
    char filename[64] = "";
    struct stat stat;

    if (argc < 2) {
        printf("Usage: %s [file]\n", argv[0]);
        return 1;
    }
    strcpy(filename, argv[1]);

    if ((fd = open(filename, O_RDWR | O_CREAT, 0664)) < 0) {
        printf("Could not open file \"%s\" with O_RDWR", filename);
        return 1;
    }
    if (fstat(fd, &stat) < 0) {
        printf("Could not stat file \"%s\"", filename);
        goto close;
    }
    if ((file = (char *)mmap(NULL, stat.st_size, PROT_WRITE, MAP_PRIVATE,
fd, 0)) == MAP_FAILED) {
        printf("Could not mmap file \"%s\"", filename);
        goto close;
    }

    if (madvise((void *)file, stat.st_size, MADV_MERGEABLE) != 0) {
        printf("Could not madvise file \"%s\"", filename);
        goto unmap;
    }

    memset(file, '1', stat.st_size);
    memset(file, '2', stat.st_size/2);
    memset(file, '3', stat.st_size/4);
    memset(file, '4', stat.st_size/8);

    /* write random pages */
    {
        long long i, j;
        int dpage_size = 4096*20; /* 20 pages size */

        srand((unsigned int)time((time_t *)0));
        for (i = 0; i < dpage_size; i++)
        {
            j = rand ();
            file[i] = j;
        }
    }
}

```

```

    }

    printf("sleeping forever ... \n");
    while (1)
        sleep (100);

    madvise((void *)file, stat.st_size, MADV_UNMERGEABLE);
unmap:
    munmap(file, stat.st_size);
close:
    close(fd);

    return 1;
}

```

在上例中向 mmap 指定了 PROT\_WRITE 和 MAP\_PRIVATE。

在 mmap 的区域中写入 20 页的随机值 (见图 5-28)。这 20 页应该是不能合并的。

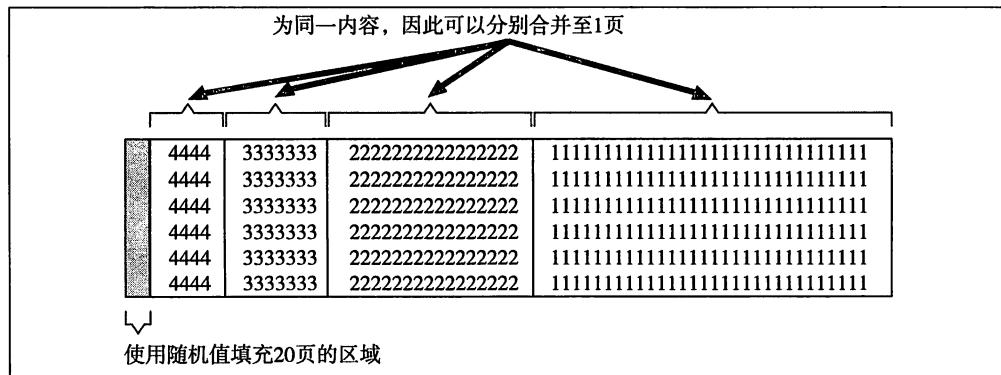


图 5-28 mergeable.c 的 mmap 区域映像

首先, 使用 dd 命令生成 100MB 的文件。

```
# dd if=/dev/zero of=ksm.dat bs=1M count=100
```

对 mergeable.c 进行编译。

```
# gcc mergeable.c -o mergeable
```

指定 dd 命令生成的文件, 执行进程。

```
# ./mergeable ksm.dat
```

```

# head /sys/kernel/mm/ksm/*
==> /sys/kernel/mm/ksm/full_scans <==
247
==> /sys/kernel/mm/ksm/pages_shared <==
0
==> /sys/kernel/mm/ksm/pages_sharing <==
0
==> /sys/kernel/mm/ksm/pages_to_scan <==

```

```
2046
==> /sys/kernel/mm/ksm/pages_unshared <==
0
==> /sys/kernel/mm/ksm/pages_volatile <==
18414      设置了 madvise 的 MADV_MERGEABLE 后, 首先这个值增加
==> /sys/kernel/mm/ksm/run <==
1          ksmd 正在运行
==> /sys/kernel/mm/ksm/sleep_millisecs <==
188
```

经过一段时间后, 再次执行下列命令。

```
# head /sys/kernel/mm/ksm/*
==> /sys/kernel/mm/ksm/full_scans <==
252      全部扫描的次数
==> /sys/kernel/mm/ksm/pages_shared <==
4          pages_sharing 的页数最终合并到 4 页
==> /sys/kernel/mm/ksm/pages_sharing <==
25576      合并的页数
==> /sys/kernel/mm/ksm/pages_to_scan <==
2046
==> /sys/kernel/mm/ksm/pages_unshared <==
20      写入随机值的页面
==> /sys/kernel/mm/ksm/pages_volatile <==
0          如果检查是否可以通过 KSM 合并, 则这个值减小
==> /sys/kernel/mm/ksm/run <==
1
==> /sys/kernel/mm/ksm/sleep_millisecs <==
188
```

在 RHEL6 中, 由 ksmtuned (Kernel Samepage Merging(KSM) Tuning Daemon) 定期地将 pages\_to\_scan 等改变为适当的值。

## 小结

本节介绍了 KSM。可以先确认内存的合并比例, 作为检测内存容量的大致标准。由于 KSM 原本就是为虚拟环境而设计的, 因此在有多个客户端操作系统的系统中, 尤其能够发挥较大效果。但是, KSM 在扫描内存时会产生一些不算大的负载, 因此在用不到 KSM 的系统中也可以禁用 ksm、ksmtuned。

## 参考文献

- Increasing memory density by using KSM  
<http://www.kernel.org/doc/ols/2009/ols2009-pages-19-28.pdf>
- KSM tries again  
<http://lwn.net/Articles/330589/>

——Naohiro Ooiwa

# HACK #37 如何挂载客户端操作系统的磁盘

本节介绍挂载客户端操作系统的磁盘的方法。

如果使用 KVM 或 Xen 等，有时就需要将客户端操作系统的磁盘挂载到主机操作系统上。例如，想在客户端操作系统启动前更改 IP 地址或主机名称，想把某个文件放到客户端操作系统的磁盘中等。

对于笔者来说，由于会对客户端操作系统的磁盘进行备份，但是很容易忘记是哪个磁盘，因此就需要确认磁盘的内容。

但是客户端操作系统的磁盘有时实际上是主机操作系统上的文件，大部分情况下难以使用一般的 `mount` 命令将客户端操作系统挂载到主机操作系统上。

这里就将介绍使用一些工具将客户端操作系统挂载到主机操作系统上的方法。

## guestfish

`guestfish` 是可以读写客户端操作系统的文件系统的工具。RHEL6 也包含这个工具，可以通过 `yum install guestfish` 来安装。

使用 `virt-manager` 生成客户端操作系统时，将在 `/var/lib/libvirt/images/` 下生成客户端操作系统的磁盘映像。

尝试使用 `guestfish` 来挂载这个客户端操作系统的磁盘。

```
# guestfish ... ①

Welcome to guestfish, the libguestfs filesystem interactive shell for
editing virtual machine filesystems.

Type: 'help' for help with commands
      'quit' to quit the shell

><fs> add /var/lib/libvirt/images/test.img ... ②
><fs> launch ... ③
><fs> list-devices ... ④
/dev/vda
><fs> sfdisk-l /dev/vda ... ⑤

Disk /dev/vda: 16644 cylinders, 16 heads, 63 sectors/track
Units = cylinders of 516096 bytes, blocks of 1024 bytes, counting from 0

  Device Boot Start      End  #cyls  #blocks  Id  System
/dev/vda1    *      2+    1017-    1016-    512000  83  Linux
/dev/vda2        1017+  16644-  15627-  7875584  8e  Linux LVM
/dev/vda3            0      -      0        0    0  Empty
/dev/vda4            0      -      0        0    0  Empty

><fs> lvs ... ⑥
/dev/VolGroup/lv_root
```

```

/dev/VolGroup/lv_swap
><fs> mount /dev/VolGroup/lv_root / ... ⑦
><fs> mounts ... ⑧
/dev/mapper/VolGroup-lv_root
><fs> ls / ... ⑨
bin
boot
cgroup
dev
etc
home
lib
lib64
lost+found
media
mnt
opt
proc
root
sbin
selinux
srv
sys
tmp
usr
var
><fs>
><fs> touch /root/hoge.txt ... ⑩
><fs> edit /root/hoge.txt ... ⑪
aaaa
bbbb
cccc

><fs> cat /root/hoge.txt ... ⑫
aaaa
bbbb
cccc

><fs> upload /root/foo /root/foo2 ... ⑬
><fs> cat /root/foo2
foofoo

><fs> download /root/hoge.txt /tmp/hoge2 ... ⑭
><fs> umount /dev/VolGroup/lv_root ... ⑮
><fs> mounts
><fs> quit ... ⑯

```

首先①启动 guestfish。guestfish 的 shell 就会启动，因此②使用 add 命令将客户端操作系统的磁盘映像添加到 guestfish。③启动 qemu 的子进程。在主机操作系统上执行 ps 命令就可以看到，guestfish 在后台启动 qemu 的子进程，并让在这个 qemu 上运行的用于作业的客户端操作系统来处理磁盘映像。

```

3145 pts/1      S+      0:00      |          \_ guestfish
3368 pts/1      S1+      0:04      |          \_ /usr/libexec/qemu-kvm -drive file=/

```

```

var/lib/libvirt/images/test.img,cache=off,if=virtio
-enable-kvm -nodefaults -nographic -serial stdio -m 500 -no-reboot -net
user,vlan=0,net=169.254.0.0/16 -net nic,model=virtio,vlan=0
-kernel /tmp/libguestfs-p5ipPA/kernel -initrd /tmp/libguestfs-p5ipPA/initrd
-append panic=1 console=ttyS0 udevtimeout=300 noapic
acpi=off printk.time=1 cgroup_disable=memory selinux=0 guestfs_vmchannel=tcp
:169.254.2.2:33653 TERM=xterm
3369 pts/1 S+ 0:00 |          \_ guestfish

```

④使用 `list-devices` 命令找出要挂载到的设备。⑤然后使用 `fdisk -l` 命令获取分区表信息。这里使用的是 LVM，⑥因此使用 `lvs` 命令获取逻辑卷。⑦然后使用 `mount` 命令将 `/dev/VolGroup/lv_root` 挂载到 `/` 下。⑧使用 `mounts` 命令确认是否已完成挂载。到这一步，挂载操作就完成了。⑨使用 `ls` 命令可以看到磁盘的内容。⑩使用 `touch` 命令在挂载的磁盘映像内生成文件。⑪使用 `edit` 命令后，`vi` 启动，就可以编辑文件。⑫还可以使用 `cat` 命令来参照文件的内容。⑬如果使用 `upload` 命令，就可以将主机操作系统上的文件复制到客户端操作系统的磁盘映像内。写法为 `upload <主机操作系统上的文件名> <客户端操作系统上的文件名>`。⑭使用 `download` 命令，就可以将客户端操作系统上的文件复制到主机操作系统上。⑮最后使用 `umount` 命令卸载。⑯再使用 `quit` 命令结束 `guestfish`。

除了这个 `guestfish` 以外，`libguest-tools` 工具包中还有能够更简单地将 `guestfish` 作为命令使用的命令（见表 5-5）。

表 5-5 其他命令

项 目	说 明
<code>virt-cat</code>	显示磁盘映像中的文件
<code>virt-df</code>	显示磁盘映像的磁盘使用量
<code>virt-edit</code>	编辑磁盘映像中的文件
<code>virt-inspector</code>	显示磁盘映像中的操作系统版本、内核、驱动程序、挂载点等
<code>virt-list-filesystems</code>	显示磁盘映像中的文件系统列表
<code>virt-list-partitions</code>	显示磁盘映像中的分区表列表
<code>virt-ls</code>	显示磁盘映像中的目录里的文件列表
<code>virt-rescue</code>	启动救援 <code>rescue shell</code>
<code>virt-resize</code>	扩大磁盘映像的大小
<code>virt-tar</code>	从磁盘映像中将文件以 <code>tar</code> 格式复制，或将 <code>tar</code> 格式文件解压缩并复制到磁盘映像中
<code>virt-win-reg</code>	显示、编辑磁盘映像中的 Windows 注册表项

### 例 5-3 `virt-ls`

```
# virt-ls /var/lib/libvirt/images/test.img /root
```

```
.bash_history
.bash_logout
.bash_profile
.bashrc
.cshrc
.tcshrc
anaconda-ks.cfg
foo
foo2
hoge.txt
install.log
install.log.syslog
```

#### 例 5-4 virt-cat

```
# virt-cat /var/lib/libvirt/images/test.img /root/hoge.txt
aaaa
bbbb
cccc
```

## lomount

在 RHEL5 等环境下，Xen 的 RPM 包中含有 lomount 命令。lomount 命令可以像普通的 mount 命令一样使用。与 guestfish 相比，由于不需要像 guestfish 那样启动操作用的客户端操作系统，因此速度较快，使用方便。

lomount 的使用方法如下。

```
lomount -t <文件系统的种类> -diskimage <挂载对象的磁盘映像名称>
-partition <分区表编号> <挂载位置>
```

下面所示为使用 lomount 命令的示例。

```
# lomount -diskimage ./x8664_domU_centos54.img ... ①
Please specify a partition number. Table is:
Num      Start -      End   OS Bootable
1:      32256 - 106928128: 83 80
2: 106928640 - 10733989888: 83 0
# lomount -diskimage ./x8664_domU_centos54.img -partition 2 /mnt ... ②
# ls /mnt
bin  boot  dev  etc  home  lib  lib64  lost+found  media  misc  mnt  net  opt
proc  root  sbin  selinux  srv  sys  tmp  usr  var
# umount /mnt ... ③
```

①使用 lomount 命令获取挂接磁盘映像的分区表信息。②使用 lomount 命令挂接磁盘映像的 2 号分区表。到这一步挂接就完成了。文件编辑等操作结束后，③使用 mount 命令卸载。

## kpartx

lomount 命令虽然非常方便，但是如果在挂载对象的磁盘映像中使用 LVM，就无法使

用该命令。要挂载 LVM 的磁盘映像，就需要使用 `kpartx` 命令。`kpartx` 用于设备映射的分区表管理工具。

首先，不使用 LVM 的磁盘映像的挂接方式如下。

```
# kpartx -av ./x8664_domU_centos54.img ... ①
add map loop3p1 : 0 208782 linear /dev/loop3 63
add map loop3p2 : 0 20755980 linear /dev/loop3 208845
# mount /dev/mapper/loop3p2 /mnt ... ②

# ls /mnt/
bin boot dev etc home lib lib64 lost+found media misc mnt net opt
proc root sbin selinux srv sys tmp usr var
# umount /mnt/ ... ③
# kpartx -dv ./x8664_domU_centos54.img ... ④
del devmap : loop3p1
del devmap : loop3p2
loop deleted : /dev/loop3
```

①使用 `kpartx` 命令的 `a` 选项将磁盘映像映射到设备。②磁盘映像内的 2 号分区表可以作为 `/dev/mapper/loop3p2` 使用，因此可以使用 `mount` 命令挂载。③卸载时使用 `umount` 命令来进行。最后④使用 `kpartx` 的 `d` 选项删除设备映射。

下列介绍的是挂接使用 LVM 的磁盘映像的情况。

```
# kpartx -av /var/lib/libvirt/images/test.img ... ⑤
add map loop0p1 (253:0): 0 1024000 linear /dev/loop0 2048
add map loop0p2 (253:1): 0 15751168 linear /dev/loop0 1026048
# vgscan ... ⑥
  Reading all physical volumes. This may take a while...
  Found volume group "VolGroup" using metadata type lvm2
# vgchange -ay VolGroup ... ⑦
  2 logical volume(s) in volume group "VolGroup" now active
# lvs ... ⑧
  LV      VG      Attr  LSize Origin Snap%  Move Log Copy%  Convert
  lv_root VolGroup -wi-- 5.54g
  lv_swap VolGroup -wi-- 1.97g
# mount /dev/VolGroup/lv_root /mnt ... ⑨
# ls /mnt
bin boot cgroup dev etc home lib lib64 lost+found media mnt opt
proc root sbin selinux srv sys tmp usr var
# umount /mnt/ ... ⑩
# vgchange -an VolGroup ... ⑪
  0 logical volume(s) in volume group "VolGroup" now active
# kpartx -dv /var/lib/libvirt/images/test.img ... ⑫
del devmap : loop0p2
del devmap : loop0p1
loop deleted : /dev/loop0
```

首先与前例相同，⑤使用 `kpartx` 的 `a` 选项将磁盘映像映射到设备。⑥使用 `vgscan` 命令检索 LVM 的卷组 (volume group)。⑦使用 `vgchange` 命令启用卷组。⑧使用 `lvs` 命令确认逻辑卷，⑨使用 `mount` 命令将要挂接的逻辑卷挂接。这里指定的是在 `/dev` 下生成的逻辑卷的设备文件。⑩使用 `umount` 命令卸载后，⑪使用 `vgchange` 命令禁用卷

组，⑫使用 kpartx 命令的 d 选项删除设备映射。

执行 vgscan 后，LVM 的卷组名有时会检测出相同的名称。卷组在系统中必须是唯一的，因此如果存在相同的卷组名，需要使用 vgrename 命令更改卷组名。下面是使用 vgrename 命令更改卷组名的例子。通过 vgdisplay 检查 VG UUID，使用 vgrename 来更改卷组名。操作结束后，再次使用 vgdisplay 命令将卷组名改回。

```
# vgdisplay
--- Volume group ---
VG Name          VolGroup
System ID
Format          lvm2
Metadata Areas  1
Metadata Sequence No 3
VG Access       read/write
VG Status        resizable
MAX LV          0
Cur LV          2
Open LV          0
Max PV          0
Cur PV          1
Act PV          1
VG Size         2.50 GiB
PE Size         32.00 MiB
Total PE        80
Alloc PE / Size 80 / 2.50 GiB
Free PE / Size  0 / 0
VG UUID         GvnrGk-xA3M-9M20-qI1X-zA94-IoQv-4JYKDM

--- Volume group ---
VG Name          VolGroup
System ID
Format          lvm2
Metadata Areas  1
Metadata Sequence No 3
VG Access       read/write
VG Status        resizable
MAX LV          0
Cur LV          2
Open LV          1
Max PV          0
Cur PV          1
Act PV          1
VG Size         2.50 GiB
PE Size         32.00 MiB
Total PE        80
Alloc PE / Size 80 / 2.50 GiB
Free PE / Size  0 / 0
VG UUID         Z48bLY-U9wv-Xrwy-0ZV3-23b7-Zv8N-SUNswN
# vgrename GvnrGk-xA3M-9M20-qI1X-zA94-IoQv-4JYKDM VolGroup_1
Volume group "VolGroup" successfully renamed to "VolGroup_1"
# vgscan
Reading all physical volumes. This may take a while...
Found volume group "VolGroup_1" using metadata type lvm2
```

```
Found volume group "VolGroup" using metadata type lvm2
# vgs
  VG      #PV #LV #SN Attr   VSize VFree
  VolGroup   1   2   0 wz--n- 2.50g    0
  VolGroup_1  1   2   0 wz--n- 2.50g    0
# vgchange -ay VolGroup_1
  2 logical volume(s) in volume group "VolGroup_1" now active
```

## 小结

在客户端操作系统无法与主机操作系统进行通信或客户端操作系统无法启动等情况下，有时会想要直接修改客户端操作系统的磁盘映像的内容。这时就可以用到本节介绍的内容。

## 参考文献

- <http://libguestfs.org/guestfish.1.html>

——Akio Takebe

# HACK #38 从客户端操作系统识别虚拟机环境

本节介绍识别客户端操作系统在哪个管理程序上运行的方法。

在使用虚拟环境的过程中，有时会需要确认已启动的操作系统在虚拟环境下是如何运行的。有时也会想要识别是在哪个管理程序上运行的。但是，客户端通过操作系统设备仿真实现仿真，基本上没有什么可以判别的信息。本节将介绍用来判别的关键信息和工具。

## CPUID 命令

CPUID 命令是根据输入到 EAX 寄存器的值，返回 CPU 识别信息或 CPU 功能信息的命令。例如，CPU 的型号或主频就可以使用 CPUID 命令来得知。也有一些管理程序使用在 Intel 和 AMD CPU 中不使用的值（0x40000000 等）来获取管理程序信息。

在客户端操作系统上发布如下 CPUID 命令，可以得到表 5-6 所示的返回值。

```
#include <stdio.h>
#include <stdint.h>

static void cpuid(uint32_t idx,
                  uint32_t *a,
                  uint32_t *b,
                  uint32_t *c,
                  uint32_t *d)
{
    asm volatile ("movl %1, %%eax; cpuid" : "=a"(*a), "=b"(*b), "=c"(*c),
    "=d"(*d) : "1"(idx) );
```

```

static int check_with_cpuid(void)
{
    uint32_t eax, ebx, ecx, edx;
    char signature[13];
    uint32_t base;

    for ( base = 0x40000000; base < 0x4000000f; base += 0x1 )
    {
        eax = 0;
        ebx = 0;
        ecx = 0;
        edx = 0;
        cpuid(base, &eax, &ebx, &ecx, &edx);

        *(uint32_t *) (signature + 0) = ebx;
        *(uint32_t *) (signature + 4) = ecx;
        *(uint32_t *) (signature + 8) = edx;
        signature[12] = '\0';
        if (ebx != 0 || ecx != 0 || edx != 0)
            printf("%x: signature=%12s eax=%x ebx=%x ecx=%x edx=%x\n",
                   base, signature, eax, ebx, ecx, edx);
    }
    return 0;
}

int main(void)
{
    return check_with_cpuid();
}

```

表 5-6 CPUID (0x40000000) 的返回值的示例

管理程序	返 回 值
KVM	KVMKVMKVM
Xen	XenVMMXenVMM
Hyper-V	Microsoft Hv

在 RHEL5 Xen 中还有 `xen-detect` 命令同样使用 CPUID 命令识别 Xen 的客户端操作系统。但需要注意的是，有一些版本（如旧的 VMware 等管理程序的版本）不支持对这个 CPUID 识别方法，因此不能使用这一功能。

## 固有文件

在 KVM 的情况下，将 Linux 作为 KVM 的主机操作系统使用时，存在名为 `/dev/kvm` 的特殊文件。在 Xen 的情况下，则会生成名为 `/proc/xen/capabilities` 的特殊文件。在 Dom0 上 `/proc/xen/capabilities` 内写有“control\_d”，而在半虚拟化客户端操作系统上 `/proc/xen/capabilities` 什么也没有。

## ACPI DSDT/FADT 的 OEM ID

有些情况可以使用 ACPI DSDT 或 FADT 的 OEM\_ID/OEM\_TABLEID 中的下列信息。这是因为各管理程序中准备了客户端操作系统用的 BIOS，FADT 的 OEM\_ID/OEM\_TABLEID 中多数情况下也写有与管理程序相应的值。

例如，在 RHEL5 Xen 的情况下，可以使用下列 shell 脚本识别。

```
#!/bin/bash
FADT=/proc/acpi/fadt
CAPABILITY=/proc/xen/capabilities
DD=/bin/dd
TR=/usr/bin/tr

if [ -r $FADT ]; then
    OEMID=$(($DD if=$FADT bs=1 skip=10 count=6 2>/dev/null | $TR -d ' '))
    OEM_TABLEID=$(($DD if=$FADT bs=1 skip=16 count=8 2>/dev/null | $TR -d ' '))
    case $OEMID/$OEM_TABLEID in
        Xen/HVM | INTEL/int-xen)
            echo hvm
            exit 0 ;;
        esac
fi
if [ -r $CAPABILITY ]; then
    [ "$(<$CAPABILITY)" = control_d ] && echo dom0 && exit 0
    echo pv ; exit 0
fi
echo native
```

## System Management BIOS (SMBIOS)

从 SMBIOS 可以获取 BIOS 提供的系统属性。例如，BIOS 的版本、CPU、内存等信息。有时通过 SMBIOS 的 System Information (Manufacturer/Product Name) 可以识别管理程序。相应在操作系统上确认 SMBIOS 的内容时，可以使用 dmidecode 命令。

### 例 5-5 KVM 的示例

```
Handle 0x0100, DMI type 1, 27 bytes
System Information
    Manufacturer: Red Hat
    Product Name: KVM
    Version: RHEL 6.0.0 PC
    Serial Number: Not Specified
    UUID: 983F2E4E-BAA8-E5D7-A97C-B574A72484B8
    Wake-up Type: Power Switch
    SKU Number: Not Specified
    Family: Red Hat Enterprise Linux
```

### 例 5-6 Xen 的示例

```
Handle 0x0100, DMI type 1, 27 bytes
System Information
    Manufacturer: Xen
```

```
Product Name: HVM domU
Version: 3.1.2-164.28.1.el5
Serial Number: 46fdb192-71ba-c2a1-fb69-4810f0e00c53
UUID: 46FDB192-71BA-C2A1-FB69-4810F0E00C53
Wake-up Type: Power Switch
SKU Number: Not Specified
Family: Not Specified
```

### 例 5-7 Hyper-V 的示例

```
Handle 0x0001, DMI type 1, 25 bytes.
System Information
    Manufacturer: Microsoft Corporation
    Product Name: Virtual Machine
    Version: 5.0
    Serial Number: 3061-9271-3107-7137-6198-0205-37
    UUID: 18570B26-2DF1-514F-ADFO-C6B8720F9636
    Wake-up Type: Power Switch
```

### 例 5-8 VMware 环境下的示例

```
Handle 0x0001, DMI type 1, 27 bytes
System Information
    Manufacturer: VMware, Inc.
    Product Name: VMware Virtual Platform
    Version: None
    Serial Number: VMware-42 2a 4f e9 90 e7 9b 39-c5 60 90 3d 57 6d 3d ad
    UUID: 422A4FE9-90E7-9B39-C560-903D576D3DAD
    Wake-up Type: Power Switch
    SKU Number: Not Specified
    Family: Not Specified
```

## virt-what

**virt-what** 是将前面所述的程序整合起来的工具。

<http://people.redhat.com/~rjones/virt-what/>

在客户端操作系统上进行如下操作，就可以识别各种管理程序。下面是 KVM 对应的示例。

```
# wget http://people.redhat.com/~rjones/virt-what/files/virt-what-1.9.tar.gz
# tar xzf virt-what-1.9.tar.gz
# cd virt-what-1.9
# ./configure
# make; make install
# virt-what
kvm
```

表 5-7 为 **virt-what** 能够识别的客户端操作系统环境。可以看出能够识别的管理程序种类非常多。

表 5-7 **virt-what** 能够识别的客户端操作系统的运行环境

<b>virt-what 的输出</b>	<b>客户端操作系统的运行环境</b>
hyperv	Microsoft Hyper-V 环境

<b>virt-what 的输出</b>	<b>客户端操作系统的运行环境</b>
<b>ibm_systemz</b>	IBM SystemZ (或 S/390) 的硬件划分系统环境 以下为附加信息: ibm_systemz-direct Linux 直接运行的环境 ibm_systemz-lpar LPAR 上 Linux 直接运行的环境 ibm_systemz-zvm LPAR 中 z/VM 客户端的环境
<b>linux_vserver</b>	Linux VServer 容器环境
<b>kvm</b>	KVM 环境
<b>openvz</b>	OpenVZ 或 Virtuozzo 环境
<b>parallels</b>	Parallels Virtual Platform (Parallels Desktop、Parallels Server) 环境
<b>powervm_lx86</b>	IBM PowerVM Lx86 环境
<b>qemu</b>	qemu 环境
<b>uml</b>	User-Mode Linux (UML) 环境
<b>virtage</b>	Virtage 环境
<b>virtualbox</b>	VirtualBox 环境
<b>virtualpc</b>	Microsoft VirtualPC 环境
<b>vmware</b>	VMware 环境
<b>xen</b>	Xen 环境 以下为追加信息: xen-dom0 Xen 的 Dom0 环境 xen-domU Xen 的半虚拟化客户端环境 xen-hvm 全虚拟化客户端环境

## 小结

当存在只想安装到客户端操作系统的内核的驱动程序时，或根据管理程序的种类想要更换驱动程序时等，这个 Hack 就能起到很大的作用。

## 参考文献

- CPUID usage for interaction between Hypervisors and Linux.  
<http://lwn.net/Articles/301888/>
- System Management BIOS Reference Specification
- virt-what  
<http://people.redhat.com/~rjones/virt-what/>
- [Xen-devel] [RFC] Is this process running on which machine?

## HACK #39 如何调试客户端操作系统

本节介绍客户端操作系统的调试方法。

客户端操作系统的调试关系到很多组件，因此经常会困惑应该从哪里下手？例如，当客户端操作系统的驱动程序出现异常时，很多情况下都难以分辨是设备驱动程序有问题？还是设备仿真器有问题？还是主机操作系统的驱动程序有问题？当客户端操作系统意外停机（hang up）时，有时也不知道应当从哪里检查。

这里就介绍可以用于客户端操作系统调试的一些技巧。

### Xen 的情况

首先，进行调试时需要确认日志（见表 5-8）和客户端操作系统的状态。确认客户端操作系统的状态时可以使用 `xm list` 或 `virsh list` 命令。

```
# xm list
Name                           ID   Mem(MiB)  VCPUs  State   Time(s)
Domain-0                        0    2048      1 r----  1507.8
testPV                          5     511      1 -b----  208.6
testvm54b                       12    1031      1 -b----  48.3
# virsh list
  Id  名称          状态
  --
  0  Domain-0      运行中
  5  testPV        idle
  12 testvm54b    idle
```

表 5-8 Xen 的日志文件

日志文件	说 明
<code>/var/log/xen/xend.log</code>	xend 输出的日志
<code>/var/log/xen/xend-debug.log</code>	xend 输出的日志
<code>/var/log/xen/xen-hotplug.log</code>	Xen 的 hotplug 事件的日志
<code>/var/log/xen/domain-builder- ng.log</code>	客户端操作系统启动中 Xen 的库输出的日志
<code>/var/log/xen/qemu-dm-&lt;客户端 操作系统名或进程 ID&gt;.log</code>	各个客户端操作系统的 qemu-dm 输出的日志
<code>/var/log/xen/console/ hypervisor.log</code>	Xen 的管理程序输出的日志

## 目志文件

## 说 明

/var/log/xen/console/guest-< 半虚拟化客户端操作系统输出的日志  
 客户端操作系统名>.log  
 /var/log/messages 主机操作系统的日志

使用 xenctx 命令，就可以简单方便地确认客户端操作系统是否意外停机。xenctx 命令是显示指定客户端操作系统的虚拟 CPU 上下文信息的命令。从主机操作系统观察 rip 寄存器的值或栈的内容是否发生变化，就可以确认客户端操作系统是否意外停机。

xenctx [ 选项 ] <domid> <虚拟 CPU 编号 >

### 例 5-9 半虚拟化客户端操作系统的情况

```
# /usr/lib64/xen/bin/xenctx 5 0
rip: ffffffff802063aa
rsp: ffffffff8063bf58
rax: 00000000    rbx: 00000000    rcx: ffffffff802063aa    rdx: 00000001
rsi: 00000000    rdi: 00000001    rbp: 00000000
r8: 00000081    r9: 1001d5381    r10: ffff880001fd3a3e0    r11: 00000246
r12: 00000000    r13: 00000000    r14: 00000000    r15: 00000000
cs: 0000e033    ds: 00000000    fs: 00000000    gs: 00000000

Stack:
0000000231002060 0000000000000000 ffffffff8026f4d5 0000000000000000
0000000000000000 0000000007020800 ffffffff8026ca50 0000000000000000
ffffffffff8024afa1 0000000007020800 ffffffff80644b05 0000000000000000
0000000000000000 ffffffff80683d20 ffffffff806441e5 ffff800000000000

Code:
cc 51 41 53 b8 1d 00 00 00 of 05 <41> 5b 59 c3
cc cc cc cc cc cc cc

Call Trace:
[<ffffffffff802063aa>] <-- 
[<ffffffffff8026f4d5>]
[<ffffffffff8026ca50>]
[<ffffffffff8024afa1>]
[<ffffffffff80644b05>]
[<ffffffffff80683d20>]
[<ffffffffff806441e5>]
```

### 例 5-10 全虚拟化客户端操作系统的情况

```
# /usr/lib64/xen/bin/xenctx 12 0
rip: ffffffff8006b319
rsp: ffffffff803f3f90
rax: 00000000    rbx: ffffffff8006b2f0    rcx: 00000000    rdx: 00000000
rsi: 00000001    rdi: ffffffff80303698    rbp: 00090000
r8: ffffffff803f2000    r9: 0000003f    r10: ffff81003ffa0008    r11: 00000286
r12: 00000000    r13: 00000000    r14: 00000000    r15: 00000000
cs: 00000010    ds: 00000000    fs: 00000000    gs: 00000000
```

```

Stack:
ffffffffff8004959b 0000000007000800 ffffffff803fd7fd 0000000000090000
0000000000000000 ffffffff80448740 ffffffff803fd22f 80008e000010019c
00000000ffffffff 0000000000000000 0000000000000000 00000000000200000
0000000000000000 0000000000000000

Code:
page entry not present in PT
page entry not present in PT
... snip ...
page entry not present in PT
page entry not present in PT
53 ff 00 f0 53 <ff> 00 f0 53
ff 00 f0 53 ff 00 f0

Call Trace:
[<ffffffffff8006b319>] <-- 
[<ffffffffff8004959b>]
[<ffffffffff803fd7fd>]
[<ffffffffff80448740>]
[<ffffffffff803fd22f>]

```

不停止客户端操作系统，提取客户端操作系统的内存转储时，使用 `xm dump-core` 命令（见表 5-9）。

```
xm dump-core [-L|- -live] [-C|- -crash] <Domain> [Filename]
```

表 5-9 `xm dump-core` 命令的选项

选 项	功 能
<code>-L - -live</code>	不停止客户端操作系统，提取转储
<code>-C - -crash</code>	提取转储后停止客户端操作系统
<code>Domain</code>	客户端操作系统名
<code>Filename</code>	指定转储的输出位置

```
# xm dump-core rhel5hvm /tmp/hoge
Dumping core of domain: rhel5hvm...
```

不指定文件名时，将在 `/var/lib/xen/dump` 下生成转储文件。也有一些发布版是在 `/var/xen/dump` 下生成转储文件。

转储文件通过使用 `crash` 命令来参照。然后，就可以与普通的 `crash` 命令一样进行调试。

```
# crash /usr/lib/debug/lib/modules/2.6.18-164.el5xen/vmlinu /tmp/hoge
crash 4.0-8.9.1.el5
Copyright (C) 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009 Red Hat, Inc.
Copyright (C) 2004, 2005, 2006 IBM Corporation
Copyright (C) 1999-2006 Hewlett-Packard Co
Copyright (C) 2005, 2006 Fujitsu Limited
Copyright (C) 2006, 2007 VA Linux Systems Japan K.K.
Copyright (C) 2005 NEC Corporation
There is absolutely no warranty for GDB. Type "show warranty" for details.
```

```
This GDB was configured as "x86_64-unknown-linux-gnu"...
```

```
KERNEL: /usr/lib/debug/lib/modules/2.6.18-164.el5xen/vmlinuz
DUMPFILE: /tmp/hoge
CPUS: 1
DATE: Fri May 20 01:10:48 2011
UPTIME: 00:00:10
LOAD AVERAGE: 0.00, 0.00, 0.00
TASKS: 24
NODENAME: (none)
RELEASE: 2.6.18-164.el5xen
VERSION: #1 SMP Mon Sep 27 12:59:01 EDT 2010
MACHINE: x86_64 (1861 MHz)
MEMORY: 1 GB
PANIC: ""
PID: 0
COMMAND: "swapper"
TASK: ffffffff804eeb00 [THREAD_INFO: ffffffff8063c000]
CPU: 0
STATE: TASK_RUNNING (ACTIVE)
WARNING: panic task not found
```

```
crash> ps
  PID  PPID  CPU      TASK      ST  %MEM    VSZ    RSS  COMM
> >    0      0    0  ffffffff804eeb00  RU  0.0      0      0  [swapper]
      1      0    0  ffff880000f2d7a0  IN  0.1  2684    700  init
      2      1    0  ffff880000f2d040  IN  0.0      0      0  [migration/0]
      3      1    0  ffff88000003f7e0  IN  0.0      0      0  [ksoftirqd/0]
      4      1    0  ffff88000003f080  IN  0.0      0      0  [watchdog/0]
      5      1    0  ffff88003ff28820  IN  0.0      0      0  [events/0]
      6      1    0  ffff88003ff280c0  IN  0.0      0      0  [khelper]
      7      1    0  ffff88003fff4860  IN  0.0      0      0  [kthread]
      9      7    0  ffff88003f80e7a0  IN  0.0      0      0  [xenwatch]
     10      7    0  ffff88003f80e040  IN  0.0      0      0  [xenbus]
     14      7    0  ffff88003f8120c0  IN  0.0      0      0  [kblockd/0]
     15      7    0  ffff88003f85f860  IN  0.0      0      0  [cqueue/0]
     19      7    0  ffff88003f8677e0  IN  0.0      0      0  [khubd]
     21      7    0  ffff88003f868820  IN  0.0      0      0  [kseriod]
     80      7    0  ffff88003faa70c0  IN  0.0      0      0  [pdflush]
     81      7    0  ffff88003faab860  IN  0.0      0      0  [pdflush]
     82      7    0  ffff88003faab100  IN  0.0      0      0  [kswapd0]
     83      7    0  ffff88003fad27a0  IN  0.0      0      0  [aio/0]
    213      7    0  ffff88003f10d0c0  IN  0.0      0      0  [kpsmoused]
    214      1    0  ffff88003f10d820  IN  0.0  2684    396  nash-hotplug
    240      7    0  ffff88003f118860  IN  0.0      0      0  [ata/0]
    241      7    0  ffff88003f118100  IN  0.0      0      0  [ata_aux]
    248      7    0  ffff88003f10e860  IN  0.0      0      0  [fc_sc_wq]
   255      7    0  ffff88003f0eb860  IN  0.0      0      0  [kstriped]
crash> q
```

## KVM 的情况

KVM 的日志中有如下内容。

```
/var/log/libvirt/qemu/<客户端操作系统名>.log: libvirt 的日志  
/var/log/messages: 主机操作系统的日志
```

在 KVM 中也和 Xen 一样可以使用 `virsh list` 命令确认客户端操作系统的状态。

```
# virsh list  
Id 名称      状态  
13rhel6-4    运行中  
14rhel6-1    运行中
```

在 KVM 中调试客户端操作系统时使用 `gdb` 就会非常简单。KVM 中是经由 `qemu` 将 `gdb` 连接到客户端操作系统。在 `qemu` 启动选项中加上 `-s`，就可以调试客户端操作系统。

在使用 `libvirt` 的情况下，有时也可以通过 `virsh edit` 来改写客户端操作系统的配置文件，以使用 `-s` 选项。但需要注意，一部分旧的 `libvirt` 是不支持这个 XML 的。另外，在系统中只能使用一个 `gdb` 来进行客户端操作系统调试。

```
# virsh edit <domain名>
```

修改如下<sup>注7</sup>：

```
- <domain type='kvm'>  
+ <domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>  
+   <qemu:commandline>  
+     <qemu:arg value='-s' />  
+   </qemu:commandline>  
  
# LC_ALL=C PATH=/sbin:/usr/sbin:/bin:/usr/bin QEMU_AUDIO_DRV=none /  
usr/libexec/qemu-kvm -M rhel6.0.0 -enable-kvm -m 1024 -smp  
2,sockets=2,cores=1,threads=1 -name test -uuid 031cda76-6500-7f53-6250-  
f44bec96a5a8 -nodefconfig -nodefaults -chardev socket,id=monitor,path=/var/lib/  
libvirt/qemu/test.monitor,server,nowait -mon chardev=monitor,mode=control -rtc  
base=utc -boot c -drive file=/var/lib/libvirt/images/test.img,if=none,id=drive-  
virtio-disk0,boot=on,format=raw,cache=none -device virtio-blk-pci,  
bus=pci.0,addr=0x5,drive=drive-virtio-disk0,id=virtio-disk0 -drive  
if=none,media=cdrom,id=drive-ide0-1-0,readonly=on,format=raw -device ide-dri-  
ve,bus=ide.1,unit=0,drive=drive-ide0-1-0,id=ide0-1-0 -chardev pty,id=serial0  
-device isa-serial,chardev=serial0 -usb -device usb-tablet,id=input0 -vnc  
127.0.0.1:3 -vga cirrus -device AC97,id=sound0,bus=pci.0,addr=0x4 -device  
virtio-balloon-pci,id=balloon0,bus=pci.0,addr=0x6 -s
```

从其他终端启动 `gdb`。使用 `target remote` 命令连接到 127.0.0.1 的 1234 号端口。连接后客户端操作系统就会进入停止状态。使用 `gdb` 的 `info register` 命令等可以确认寄存器的内容或内存状态等。

```
# gdb --quiet /usr/lib/debug/lib/modules/2.6.32-71.el6.x86_64/vmlinu-  
Reading symbols from /usr/lib/debug/lib/modules/2.6.32-71.el6.x86_64/  
vmlinu...done.  
(gdb) target remote 127.0.0.1:1234  
Remote debugging using 127.0.0.1:1234  
native_safe_halt () at /usr/src/debug/kernel-2.6.32-71.el6/linux-2.6.32-71.
```

---

注 7: ‘-’ 表示删除，‘+’ 表示添加。

```

e16.x86_64/arch/x86/include/asm/irqflags.h:50
50      }
(gdb) info registers
rax          0x0      0
rbx          0xffffffff8170dfd8      -2123309096
rcx          0x0      0
rdx          0x0      0
rsi          0x1      1
rdi          0xffffffff81a101e8      -2120154648
rbp          0xffffffff8170dec8      0xffffffff8170dec8
rsp          0xffffffff8170dec8      0xffffffff8170dec8
r8           0x0      0
r9           0x0      0
r10          0x0      0
r11          0x0      0
r12          0xffffffff818a1b60      -2121655456
r13          0x0      0
r14          0xfffffffffffffff      -1
r15          0x93780  604032
rip          0xffffffff8103be8b      0xffffffff8103be8b <native_safe_
halt+11>
eflags        0x246      [ PF ZF IF ]
cs            0x10      16
ss            0x18      24
ds            0x18      24
es            0x18      24
fs            0x0      0
gs            0x0      0
(gdb) x/2s log_buf
0xffffffff81a59b60:      "<6>Initializing cgroup subsys cpuset\
n<6>Initializing cgroup subsys cpu\n<5>Linux version 2.6.32-71.e16.x86_64
(mockbuild@x86-007.build.bos.redhat.com) (gcc version 4.4.4 20100726 (Red Hat
4.4.4-13) (GC)...
0xffffffff81a59c28:      "C) #1 SMP Wed Sep 1 01:33:01 EDT 2010\n<6>Command
line: ro root=/dev/mapper/VolGroup-lv_root rd_LVM_LV=VolGroup/lv_root rd_LVM_
LV=VolGroup/lv_swap rd_NO_LUKS rd_NO_MD rd_NO_DM LANG=ja_JP.UTF-8 KEYBOA"...
(gdb) detach
Ending remote debugging.
(gdb) q

```

使用这个 gdb 功能，还可以使操作系统像应用程序一样单步运行。另外，还可以设置断点（break point），在任意地方停止客户端操作系统运行等，这是非常强大的调试功能。

```

# gdb --quiet /usr/lib/debug/lib/modules/2.6.32-71.e16.x86_64/vmlinux
Reading symbols from /usr/lib/debug/lib/modules/2.6.32-71.e16.x86_64/
vmlinux...done.
(gdb) target remote 127.0.0.1:1234
Remote debugging using 127.0.0.1:1234
0xffffffff8103cc88 in pvclock_clocksource_read (src=0xffff880001e16900) at
arch/x86/kernel/pvclock.c:124
124      {
(gdb) info registers
rax          0x16900  92416
rbx          0x3dcc7011      1036808209
rcx          0x0      0
rdx          0x0      0

```

```

rsi          0x2      2
rdi          0xfffff880001e16900      -131941363783424
rbp          0xfffff880001e03ee8      0xfffff880001e03ee8
rsp          0xfffff880001e03ed8      0xfffff880001e03ed8
r8           0x0      0
r9           0x1      1
r10          0x0      0
r11          0xffffcb9 16776377
r12          0x61     97
r13          0x883a   34874
r14          0xfffffffff81736440      -2123144128
r15          0x93780  604032
rip          0xfffffffff8103cc88      0xfffffffff8103cc88 <pvclock_
clocksource_read+8>
eflags        0x86      [ PF SF ]
cs            0x10     16
ss            0x18     24
ds            0x18     24
es            0x18     24
fs            0x0      0
gs            0x0      0
(gdb) info b
No breakpoints or watchpoints.
(gdb) x/8i $rip
=> 0xfffffffff8103cc88 <pvclock_clocksource_read+8>:      push  %r13
    0xfffffffff8103cc8a <pvclock_clocksource_read+10>:      push  %r12
    0xfffffffff8103cc8c <pvclock_clocksource_read+12>:      push  %rbx
    0xfffffffff8103cc8d <pvclock_clocksource_read+13>:      mov   %rdi,%rbx
    0xfffffffff8103cc90 <pvclock_clocksource_read+16>:      sub   $0x18,%rsp
    0xfffffffff8103cc94 <pvclock_clocksource_read+20>:      mov   (%rdi),%r12d
    0xfffffffff8103cc97 <pvclock_clocksource_read+23>:
jmp   0xfffffffff8103cca3 <pvclock_clocksource_read+35>
    0xfffffffff8103cc99 <pvclock_clocksource_read+25>:      nopl  0x0(%rax)
(gdb) stepi
0xfffffffff8103cc8a    124      {
(gdb) b *0xfffffffff8103cc94
Breakpoint 1 at 0xfffffffff8103cc94: file arch/x86/kernel/pvclock.c, line 124.
(gdb) info b
Num      Type          Disp Enb Address          What
1       breakpoint    keep y  0xfffffffff8103cc94 in pvclock_clocksource_read
at arch/x86/kernel/pvclock.c:124
(gdb) cont
Continuing.

Breakpoint 1, pvclock_clocksource_read (src=0xfffff880001e16900) at arch/x86/
kernel/pvclock.c:124
124      {
(gdb) info registers
rax          0x16900  92416
rbx          0xfffff880001e16900      -131941363783424
rcx          0x0      0
rdx          0x0      0
rsi          0x2      2
rdi          0xfffff880001e16900      -131941363783424
rbp          0xfffff880001e03ee8      0xfffff880001e03ee8
rsp          0xfffff880001e03ea8      0xfffff880001e03ea8
r8           0x0      0
r9           0x1      1

```

```
r10          0x0      0
r11          0xffffcb9 16776377
r12          0x61      97
r13          0x883a  34874
r14          0xffffffff81736440      -2123144128
r15          0x93780 604032
rip          0xffffffff8103cc94      0xffffffff8103cc94 <pvclock_
clocksource_read+20>
eflags        0x92      [ AF SF ]
cs            0x10      16
ss            0x18      24
ds            0x18      24
es            0x18      24
fs            0x0       0
gs            0x0       0
(gdb)
```

在 KVM 上调试 BIOS 等情况必须在 8086 模式下，因此一定不能忘记使用下列命令更改模式。

```
(gdb) set arch i8086
```

也可以与 Xen 一样提取客户端操作系统的内存转储。在 KVM 的情况下使用 virsh dump。由于 virsh dump 是最近才添加的功能，因此也有一些发布版尚不能使用。

```
# virsh dump rhel6-1 /tmp/test-dump
```

域 rhel6-1 转储到 /tmp/test-dump。

转储文件的内容可以使用 crash 命令来查看。

```
# crash /usr/lib/debug/lib/modules/2.6.32-71.el6.x86_64/vmlinu /tmp/test-dump
```

---

注意事项：crash-4.0-8.12 中添加了能够分析使用 virsh dump 提取的内存转储文件的功能。然后进行了各种修改和改善。旧版的 crash 命令有时不能查看使用 virsh dump 提取的转储文件，建议尽量使用最新的 crash 命令。

---

## 小结

本章介绍了调试客户端操作系统的技巧。除了调试以外，在学习 BIOS 或内核的运行等时也可以使用。

## 参考文献

- QEMU Emulator User Documentation  
<http://qemu.weilnetz.de/qemu-doc.html>
- What monitor commands does libvirt support? What QEMU/KVM command line flags does libvirt support?  
<http://wiki.libvirt.org/page/QEMUSwitchToLibvirt>

——Akio Takebe

# 省 电

近年来，为了应对全球气候变暖、电力不足以及降低成本等，省电技术受到人们的广泛关注。从电脑方面来说，不打开电源最为省电，但这显然是不现实的。电脑省电的技巧和方法非常多，但是人们总是不知不觉中就浪费了电能。例如，关掉充满动画广告的 Web 网页就可以省电，减少使用进行无用的文件检索的软件也可以省电。在最近的电脑中都配备了各种各样的省电功能，但是也经常遇到不知道该怎么使用这些功能的情况。为了丰富大家的省电知识，本章将介绍 Linux 中的省电技术。

## HACK #40 ACPI

本节介绍在 ACPI (Advanced Configuration and Power Interface) 系统的功能。

ACPI 是进行各设备及整个系统的电源管理接口。ACPI 不仅能向操作系统提供各设备的资源和 CPU 的结构信息等，还具有操作系统直接对设备电源进行操作的接口。从省电的角度来看，ACPI 的作用非常大，在当前 PC 的省电中起核心作用。了解安装方法，就可以找到更加正确的省电方法或 Hack。但是，除了操作系统以外，ACPI 还需要硬件和 BIOS 等固件的支持，用户会觉得它的结构很复杂。因此在介绍 ACPI 相关的省电 Hack 前，先介绍一下 ACPI 的概要。

### ACPI 的用语

首先介绍一些不常见的 ACPI 用语。ACPI 中定义了 G、D、S、C、P 这 5 个大的电力状态。图 6-1 介绍了各电力状态的定义。

### G 状态与 S 状态

G 状态 (Global System State) 表示的是用户看到的整个系统的电力状态。G 状态的电力状态定义见表 6-1。

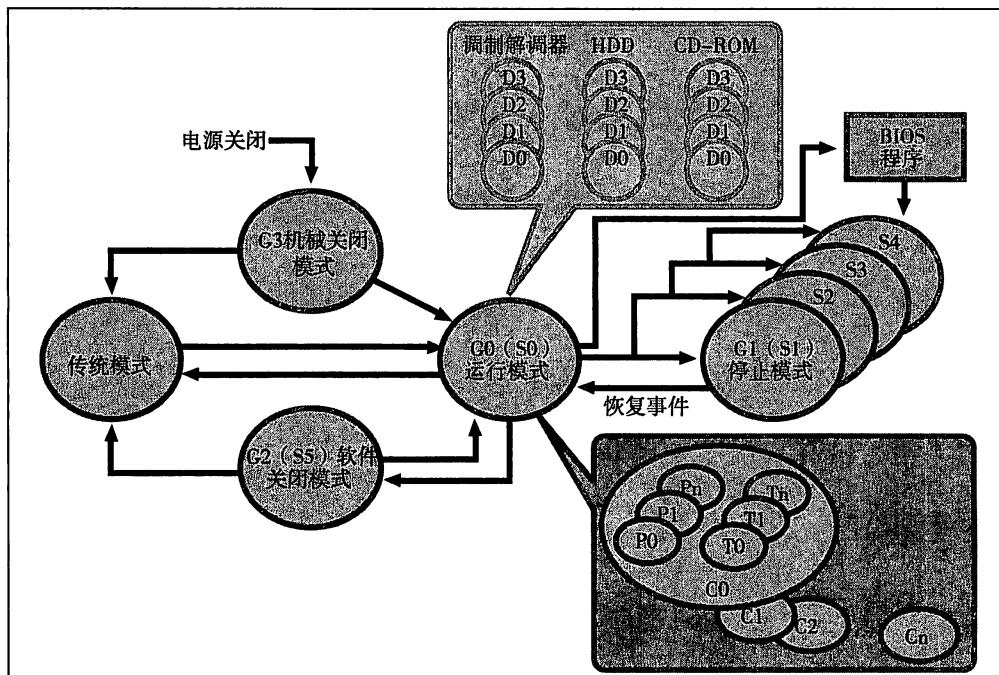


图 6-1 ACPI 的各状态之间的转换图

表 6-1 G 状态的说明

状态	说明
G0	运行模式。向硬件提供电源，软件可以运行的状态
G1	停止模式。所谓的待机或休眠状态
G2	软件为关闭状态，硬件消耗若干电力的状态
G3	系统完全关闭，电源关闭的状态

而 S 状态 (Sleeping state) 表示 G 状态的停止模式的种类。S1 ~ S4 表示 G1 的详细状态，描述的是停止的“深度”。各 S 状态的意义如表 6-2 所示。

表 6-2 S 状态的说明

状态	说明
S0	运行模式。与 G0 含义相同
S1	到恢复为止的延迟时间较少的停止模式。CPU 的上下文等都不会丢失
S2	丢失 CPU 和系统缓存上下文。这些上下文需要在操作系统唤醒时进行恢复。在 Linux 中与 S3 相同
S3	丢失除软件以外的系统上下文。这些上下文需要在操作系统唤醒时进行恢复

(续)

状态	说 明
S4	最省电, 到恢复为止花费时间最多的停止模式。停止向所有设备提供电源。启动时 BIOS 会通知是从 S4 恢复的
S5	除了不保存上下文以外, 其他与 S4 相同。S5 在恢复时进行的处理与普通的操作系统相同。与 G2 含义相同

停止模式根据停止的方法一般有多种叫法, 参见表 6-3。

表 6-3 停止模式的种类

S 状态	名 称
S0	运行中
S1	睡眠 (stand by)
S2	待机 suspend (CPU 的待机)
S3	待机 suspend (CPU 的待机)
S4	休眠 (磁盘的待机)
S5	软件关闭

## D 状态

D 状态 (Device Power State) 定义的是各个设备的电力状态。设备的状态有如下内容。

- 耗电量
- 保存设备内寄存器上下文的状态
- 直到设备驱动程序可使用为止必须进行的操作量
- 直到设备可使用为止所需时间

D 状态的定义如表 6-4 所示, 但几乎没有设备可以支持所有状态。一般设备主要只使用 D0 和 D3 状态。将设备的电源状态设置为 D3 再设置为 D0, 就可以将设备重启后再进行设备的初始化。

表 6-4 设备的电力状态

状 态	说 明
D0	设备可以完全运行的状态。所有上下文全部有效, 最耗电
D1	D1 对于每个设备类型的意义不同。一般来说, 耗电量比 D0 少, 丢失的上下文比 D2 更少
D2	D2 对于每个设备类型的意义也不同。一般来说, 耗电量比 D1 更少, 丢失的上下文比 D1 更多

(续)

状态	说明
D3hot	D3hot 对于每个设备类型的意义也不同。D3hot 状态的设备主电源开启，可以从软件访问设备。但上下文是否能保留取决于实际安装的设备
D3	设备电源完全断开的状态。设备的上下文全部丢失，到恢复为止花费的时间最长。在 PCI 用语中称为 D3cold。在 PCI 用语中将 D3hot 和 D3cold 统称为 D3

## C 状态

C 状态 (Processor Power State) 是 G0 中 CPU 空闲时进行的省电模式。各 C 状态的意义如表 6-5 所示。

表 6-5 C 状态的说明

状态	说明
C0	运行中状态。通常的运行模式
C1	CPU 停止状态。使用 hlt 命令停止 CPU 的时钟。到恢复为止几乎没有延迟时间，软件不需进行特殊操作
C2	总线的时钟也停止。恢复所花费的最长延迟时间传递给 ACPI 的固件，操作系统基于这个延迟时间判断使用 C1 还是 C2
C3	将花费时间最长的延迟传递给 ACPI 的固件，操作系统使用这个延迟时间判断使用 C2 还是 C3。操作系统需要考虑缓存的同步

ACPI 说明书中没有记载的 C 状态也已经在各 CPU 厂商的 CPU 数据表中出现。例如，Intel 公司的 CPU 中的定义就如表 6-6 所示。

表 6-6 Intel 公司 CPU 的 C 状态的定义

状态	定义
C1	Autohalt State
C1E	Enhanced Autohalt State
C3	Deep Sleep State
C3E	Enhanced Deep Sleep State
C4	Deeper Sleep State
C4E	Enhanced Deeper Sleep State
C5	Enhanced Deeper Sleep State
C6	Deep Power Down State
CC	核心层 C 状态

处理器的耗电量指标中有 TDP (Thermal Design Power)，通过尽量延长 CPU 空闲时间，即使运行中的 TDP 较高的 CPU 中也可以降低单位时间的平均耗电。

在 Intel ® Core 2 Duo 等 CPU 中将每个核的 C 状态定义为 CC 状态，而在 Intel ® Atom Processor Z5xx 系列等 CPU 中是将每个线程的 C 状态 (TC) 和每个核的 C 状态分别安装的。例如，当核内所有线程的 C 状态都变成 C2 时，核的 C 状态也会变成 C2。参见表 6-7。

表 6-7 Core 2 Duo 的 TDP (根据 Core 2 Duo 处理器数据表)

状态	TDP
C0	35 瓦
C1	13.5 瓦
C2	12.9 瓦
C3	7.7 瓦
C4	1.2 瓦

使用 C 状态时要注意，C 状态的程度越深，恢复到 C0 状态所需的时间越长。在对应答性要求非常高的系统中，需要避免使用 C 状态，或在使用时十分注意。

## P 状态

P 状态 (Device and Processor Performance State) 的目的是以控制电量消耗来代替降低设备或 CPU 的性能，对 D0 状态的设备、C0 状态的 CPU 进行了更细致的状态划分。参见表 6-8。

表 6-8 P 状态的说明

状态	说明
P0	通常的模式。以最高性能、最大耗电量运行
P1	运行在低于最高性能、最大耗电量的模式
Pn	n 的值越大，性能和耗电量越低。可以定义各设备或 CPU 中性能和耗电量的状态

每种 CPU 所支持的 P 状态级别不同。例如，Intel 的 CPU 中采用的是 Enhanced Intel SpeedStep Technology 技术，可以灵活地进行 P 状态控制。例如，Core 2 Duo 的 Enhanced Intel SpeedStep 可以对每个核进行设置，使用 CPU 的 MSR (model specific register) 进行控制。

使用 P 状态时，必须要在省电和性能之间取得平衡。例如，在必须确保单位时间性能的系统中，需要避免使用 P 状态，或在使用时充分注意。

此外还有为了温度控制而控制 CPU 时钟的 T 状态 throttling。在 ACPI T 状态下使用

throttling（切换 C0 状态和 C1 状态）就可以控制 CPU 温度和耗电。

## ACPI 的结构

构成 ACPI 的主要组件定义如下。

- ACPI 系统描述表
- ACPI 寄存器
- ACPI BIOS

从操作系统来看，ACPI 系统描述表在 ACPI 的接口中是核心组件，提供 ACPI 寄存器等信息。另外 ACPI BIOS 可以提供 ACPI 系统描述表以及启动、停止、唤醒等功能。

下面将介绍这些组件，以及获取 ACPI 信息的方法。

## 两个编程模型

ACPI 的硬件模型有下面两种：

- 固定硬件编程模型
- 通用硬件编程模型

固定硬件编程模型使用 ACPI 中定义的寄存器来访问 ACPI 的功能。使用这个编程模型仅限于操作系统几乎不能运行，或者从性能来看操作系统不应运行。例如，C2/C3 电源控制或电源管理计时器等重视性能的功能。硬件的事件通过 System Control Interrupt (SCI) 种类的中断获知，启动操作系统的驱动程序。通用硬件编程模型是让各厂商能够灵活安装硬件的模型。各厂商可以使用 ACPI Machine Language (AML) 将硬件固有的处理安装到 BIOS 中。操作系统可以通过分析 BIOS 提供的 AML 代码，来理解寄存器的地址和访问方法等。AML 是二进制码，通过编译 ACPI Source Language (ASL) 来生成。操作系统分析 AML，将 AML 中所写的内容按照解释器 (interpreter) 来执行。硬件的事件与固定硬件编程模型同样是通过 SCI 获知，而通用硬件编程模型则执行 AML 中的控制方法。通用硬件编程模型用于设备电源管理和设备热插拔等。

## ACPI 寄存器

ACPI 寄存器中有固定硬件寄存器和通用硬件寄存器。固定硬件寄存器是固定硬件编程模型中所使用的寄存器，是 ACPI 中定义的接口。通用硬件寄存器是在安装通用硬件编程模型中使用的硬件时所需的寄存器。ACPI 中还定义了寄存器块和寄存器组。寄存器块将多个寄存器集中到一个地址区域，例如，PM1a\_STS 寄存器和 PM1a\_EN 寄存器组成地址为 PM1a\_EVT\_BLK、大小为 PM1\_EVT\_LEN 的寄存器块，前半部分为 PM1a\_STS 寄存器，后半部分为 PM1a\_EN 寄存器。寄存器组是在想要将一个寄存器块放到不同地址时使用的。例如，名为 PM1 EVT Grouping 的寄存器组就是由 PM1a\_EVT\_BLK 和

PM1b\_EVT\_BLK 构成的。寄存器组的值通过取各寄存器块的逻辑和来决定。

ACPI 可以通过读写 ACPI 寄存器来控制硬件或是获取硬件所支持功能的信息。例如，转变为 S 状态时使用固定硬件寄存器 PM1 Control 寄存器。

## ACPI 系统描述表

ACPI 中定义了描述控制系统信息、功能、系统的 ACPI 方法的表。所有表都具有 ACPI 中规定的头文件。表的头文件中有签名，签名是从内存中检索表时识别表的关键。参见表 6-9。

表 6-9 ACPI 描述的头文件

字 段	字 节 长 度	字 节 偏 移 量	说 明
Signature	4	0	识别表的 ASCII 字符串
Length	4	4	包括头文件在内的整个表的长度(字节)
Revision	1	8	与该表相关的结构的修订
Checksum	1	9	该表整体的校验和
OEMID	6	10	用来识别 OEM 的字符串
OEM Table ID	8	16	OEM 为了识别特殊数据包而使用的字符串
OEM Revision	4	24	OEM 的修订
Creator ID	4	28	创建表的实用工具的厂商 ID
Creator Revision	4	32	创建表的实用工具的修订

这些表的位置是从系统内存中 Root System Description Pointer(RSDP) 结构的表开始的。RSDP 通过从 BIOS 内存中查找“RSD PTR”字符串来定位。“RSD PTR”是 RSDP 的签名。RSDP 指向的是 RSDT 或 XSDT。RSDT 或 XSDT 是管理各表位置的表，具有各表的位置信息。

XSDT 指向的表一定从 FADT (Fixed ACPI Description Table) 开始。FADT 中有 Firmware ACPI Control Structure (FACS)、Differentiated System Description Table (DSDT)、寄存器块。

寄存器块表示固定硬件寄存器的位置。FACS 是 BIOS 为使用 ACPI 而预留的内存空间，其中有记录从 S 状态唤醒时要执行的代码的物理地址的 Firmware Waking Vector 等。

DSDT 表示定义块的表。定义块是称为 ACPI 命名空间的 AML 中记载的树状数据结构，包括系统硬件安装的详细信息、通用硬件编程模型中使用的寄存器或方法等。

访问定义块的数据对象称为“评价”，是指通过 AML 解释器解码定义块的 AML。AML 解释器的动态数据对象具有可以通过 I/O 或访问系统内存进行程序评价的功能。

ACPI 中定义的表有如图 6-2 所示。

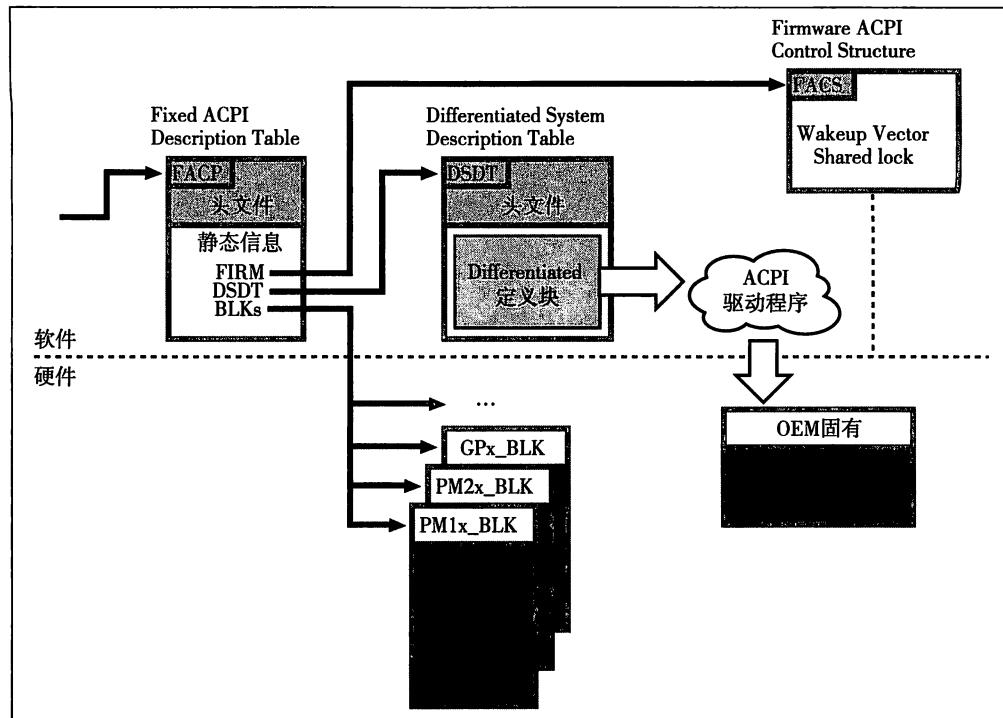


图 6-2 描述表结构

- Root System Description Pointer (RSDP)
- System Description Table Header
- Root System Description Table (RSDT)
- Fixed ACPI Description Table (FADT)
- Firmware ACPI Control Structure (FACS)
- Differentiated System Description Table (DSDT)
- Secondary System Description Table (SSDT)
- Multiple APIC Description Table (MADT)
- Smart Battery Table (SBST)
- Extended System Description Table (XSDT)
- Embedded Controller Boot Resources Table (ECDT)
- System Locality Distance Information Table (SLIT)
- System Resource Affinity Table (SRAT)

ACPI 系统描述表除了 ACPI 说明书中规定的表以外，还有 PCI 等说明书中规定的表。ACPI

表是通过签名来识别表的，因此其他说明书定义的表的签名作为 ACPI 的预留表保留。

- Simple Boot Flag Table (BOOT)
- DMA Remapping Table (DMAR)
- IA-PC High Precision Event Timer Table (HPET)
- iSCSI Boot Firmware Table (IBFT)
- I/O Virtualization Reporting Structure (IVRS)
- PCI Express memory mapped configuration space base address Description Table (MCFG)

## ACPI 命名空间和 AML (ASL)

ACPI 命名空间是定义块的层次性命名空间，所有定义块都被读入相同命名空间中。因此可以在命名空间内从其他位置参照对象或数据，但必须注意名称不要重复。下面介绍 ACPI 命名空间的命名规则。

### ACPI 命名空间的命名规则

- 所有名称的长度为固定长度 32 位
- 第一个字节为 ‘A’ ~ ‘Z’、‘\_’ (0x41 ~ 0x5A、0x5F)
- 其他 3 个字节为 ‘A’ ~ ‘Z’、‘0’ ~ ‘9’、‘\_’ (0x41 ~ 0x5A、0x30 ~ 0x39、0x5F)
- ASL 编译器为了将 4 个字以下的名称改为 4 个字，而添加 ‘\_’。
- 以 ‘\_’ 开头的名称在 ACPI 的说明书中预留。
- 以 ‘\’ 开头的名称就是参照命名空间的 root 的名称（‘\’ 不包括在 32 位固定长度的名称中）。
- 以 ‘^’ 开头的名称就是参照当前命名空间的上一层的名称（‘^’ 不包括在 32 位固定长度的名称中）。

另外，写 BIOS 的人使用 ASL 语言来描述这个定义块。将使用 ASL 写出的源代码编译，就可以生成 AML 的二进制码。操作系统通过执行这个 AML 的代码，来读取、写入定义块的系统结构。关于 ASL 的解释非常复杂，本书只作简单的介绍。

### ASL

ASL 是用来定义 ACPI 对象的语言。在 ASL 语言中，ACPI 对象由 ObjectType、FixedList、VariableList 这三者来定义。FixedList 和 VariableList 存在 null 的情况。

Object := ObjectType FixedList VariableList

Object        ACPI 对象

ObjectType    ACPI 对象的类型

FixedList 通过固定长度的列表来表示 ObjectType 的实例。表示为 (a,b,c,⋯)

VariableList 通过非固定长度的列表来表示子对象。表示为 {x,y,z,aa,bb,cc}

例如，查看 Bochs emulator<sup>注1</sup> 的 DSDT。以下为 \_S3 对象的部分。

```
Name (\_S3, Package (0x04)
{
    0x01, /* PM1a_CNT.SLP_TYP */
    0x01, /* PM1b_CNT.SLP_TYP */
    Zero, /* reserved */
    Zero /* reserved */
})
```

Name() 生成名为 \_S3 的对象。Package() 的使用方法如下：

```
Package (NumElements) {PackageList}
```

在 Package() 中，ObjectType 为 Package，NumElements 为 FixedList，PackageList 为 VariableList。NumElements(0x4) 表示 PackageList 的数量，用 PackageList 的值初始化 \_S3。\_Sx 对象定义见表 6-10。

表 6-10 \_Sx 对象的定义

字节长度	字节偏移量	说 明
1	0	系统进入 Sx 状态时 PM1a_CNT.SLP_TYP 寄存器的值
1	1	系统进入 Sx 状态时 PM1b_CNT.SLP_TYP 寄存器的值
2	2	预留

从上述示例中可以看出，要进入 S3 状态，写入 PM1 Control Grouping 的 PM1 Control 寄存器的 PM1a\_CNT.SLP\_TYP、PM1b\_CNT.SLP\_TYP 的值为 1。

## 查看 ACPI 的表

本节介绍如何在 Linux 上查看 ACPI 的表。首先安装用于将 ACPI 的信息输出到文件的名为 pmtools 工具包，该包是 RPM 格式，用于反汇编 (disassemble) AML 的工具包 iasl，该包为 RPM 格式。在 Fedora 13 的情况下可以使用 yum 命令来安装。

```
# yum install pmtools
# yum install iasl
```

如果是不包含 pmtools 和 iasl 的发布版，pmtools 可以从 <http://www.lesswatts.org/projects/acpi/utilities.php> 下载。iasl 可以从 <http://www.acpica.org/> 下载。

首先，使用 pmtools 中的 acpidump 命令输出 ACPI 的表。想要输出 DSDT 表时的操作如下。

```
# acpidump -b -t DSDT -o dsdt.dat
```

---

注 1: <http://bochs.sourceforge.net/>

然后使用 `iasl` 将这些数据反汇编。

这样就生成了名为 `dsdt.ds1` 的文件。从其中可以看到 DSDT 的 ACPI 表头文件和使用 ASL 写的定义块。查看 DSDT 等 ACPI 的表，就可以看到系统支持的功能。

```
/*
* Intel ACPI Component Architecture
* AML Disassembler version 20090123
*
* Disassembly of dsdt.dat, Fri Sep 24 07:45:31 2010
*
*
* Original Table Header:
*   Signature      "DSDT"
*   Length         0x00001E22 (7714)
*   Revision       0x01 **** ACPI 1.0, no 64-bit math support
*   Checksum        0x71
*   OEM ID         "BXPC"
*   OEM Table ID   "BXDSDT"
*   OEM Revision    0x00000001 (1)
*   Compiler ID     "INTL"
*   Compiler Version 0x20090123 (537461027)
*/
DefinitionBlock ("dsdt.aml", "DSDT", 1, "BXPC", "BXDSDT", 0x00000001
{
    Scope (\)
    {
        OperationRegion (DBG, SystemIO, 0xB044, 0x04)
        Field (DBG, DWordAcc, NoLock, Preserve)
        {
            DBGL,    32
        }
    }

    Scope (_SB)
    {
        Device (PCI0)
    }
}
```

```
Name (_HID, EisaId ("PNP0A03"))
Name (_ADR, Zero)
Name (_UID, One)
Name (_PRT, Package (0x80)
{
    ...
}
```

## 小结

本节介绍了 ACPI 的表和定义块的参照方法。下面的 Hack 将参照 ACPI 的数据，介绍关于系统省电的 Hack。

## 参考文献

- <http://download.intel.com/technology/itj/2008/v12i3/Paper1.pdf>
- Advanced Configuration and Power Interface Specification  
<http://www.acpi.info/DOWNLOADS/ACPIspec40a.pdf>

——Akio Takebe

## HACK #41 使用 ACPI 的 S 状态

本节介绍使用系统整体省电（S 状态）的方法。

不使用系统时，关闭系统是最省电的。但是，一旦关闭后，系统启动的时间就会变长，因此，如果只是暂时不用就不想关闭系统。这时，使用 S 状态就可以缩短系统恢复时间，省电效果也十分理想。

## S3 状态的使用方法

使用 S 状态时，硬件、BIOS、操作系统必须能够支持。在 Linux 下可以通过下列命令来确认可否利用 S 状态：

```
# cat /sys/power/state
standby disk
```

standby 表示 S1、disk 为 S4、mem 表示 S3，参见表 6-11。

表 6-11 可利用的 S 状态

standby	S1
mem	S3
disk	S4

在 Linux 中使用各 S 状态时，进行如下操作：

```
# echo "各状态" > /sys/power/state
```

要使用 S3(mem) 时，可以进行下列操作：

```
# echo "mem" > /sys/power/state
```

按下电源按钮，就可以重新恢复。

## S3 状态的结构

使用 S3 状态时，不同系统的反应也不同，但内核进行的操作大致如下。

1. 停止进程。
2. 停止设备运行。
3. 将唤醒时的开始地址作为 wakeup vector 登录到 BIOS。
4. 停止 BSP (Boot Strap Processor) 以外的 CPU 运行。
5. 停止系统设备运行。
6. 保存内核和 CPU 的状态。
7. 将评价 ACPI 的 \_S3 对象得到的值写入 FADT 的 PM1 寄存器，进入待机模式。

系统恢复时，从登录到 wakeup vector 的地址启动，按下列方式恢复到待机前的状态。

1. 启用 ACPI。
2. 恢复系统设备。
3. 启用 CPU。
4. 消除 wakeup vector。
5. 恢复停止的设备。
6. 恢复进程。

## S4 状态的使用方法

ACPI 中定义了两种进入 S4 的方法。一个是 BIOS 主导的方法 (S4BIOS)，另一个是操作系统主导的方法。如果使用 S4BIOS，则 BIOS 恢复内存，与从 S3 恢复的情况相同。S4BIOS 状态要求系统必须支持 S4BIOS。但是，Linux 2.6.30 中使用的不是 S4BIOS，而是操作系统主导的方法。Linux 中 S4 也称为 swap 待机，Linux 2.6.30 的 S4 处理通过将内存上的所有数据保存在交换区磁盘来停止电源，恢复时由引导加载程序 (bootloader) 启动内核，在内核初始化时，把之前保存到交换区磁盘的数据读入内存来快速恢复到原来的状态。

S4 状态的使用方法如下，与 S3 状态同，都是按下电源按钮来重新恢复。

```
# echo "platform" > /sys/power/disk
# echo "disk" > /sys/power/state
```

休眠要使用交换区磁盘，因此需要有内存量 +  $\alpha$  的磁盘容量用于交换。一般认为需要准备内存的 1.5 ~ 2 倍的磁盘容量。Linux 2.6.30 中默认保存在交换区磁盘的内存数据大小为 500MB。如果需要保存 500MB 以上的内存，可以进行如下操作。

```
# echo"0" > /sys/power/images_size
```

还需要注意恢复系统前的硬件结构。在 ACPI 中，恢复系统时只要有一个引导设备发生变更，恢复就会失败。由于依存于 BIOS 的实际安装，因此建议在待机时和系统恢复时不要改变硬件结构。与硬件一样，恢复时如果更改内核启动参数也有可能会出现问题。

另外，有的 PC 上有可能出现 BIOS 无法顺利运行，休眠失败的情况。这时可以尝试下列方法。

1. 编辑 `grub.conf`，在内核启动参数中添加用于恢复系统的交换设备“`resume=<交换设备名称>`”。

e.g.

```
title Fedora (2.6.30.10-105.2.16.fc11.i586)
root (hd0,4)
kernel /boot/vmlinuz-2.6.30.10-105.2.16.fc11.i586 ro root=UUID=eb600401-
ba10-44da-aa90-802740929780 nomodeset rhgb resume=/dev/sdb1
initrd /boot/initrd-2.6.30.10-105.2.16.fc11.i586.2.img
```

2. 将休眠设置为 `shutdown` 模式。

```
# echo shutdown > /sys/power/disk
```

3. 进行休眠。

```
# echo disk > /sys/power/state
```

如果恢复系统后系统运行状态有所不同，有时会通过将交换区的数据强制读入内存来改善性能。命令如下。

```
# swapoff -a
# swapon -a
```

## 小结

Fedora13 等中还有 `pm-utils` 的 `pm-suspend`、`pm-hibernate` 等便利的命令，可以轻松尝试待机或休眠。如果不能成功，可以查看 `/var/log/messages`，就可能找出原因。当交换区不足时，会出现 `kernel: PM: Not enough free swap` 消息，可以增大交换设备或停止一些应用程序。

# HACK #42 使用 CPU 省电 (C、P 状态)

本节介绍使用 C 状态和 P 状态的 CPU 省电方法。

## C 状态的使用方法

C 状态是 CPU 空闲时的电力状态。通过设置为更深的 C 状态就可以减少电能消耗。C 状态的层次越深入，C 状态将停止更多 CPU 功能，因此 C 状态的层次越深，从空闲状态

恢复的时间越长。ACPI 中从 C 状态的恢复基本上是以中断为契机进行的。因此，降低中断频率可以让 C 状态持续更长时间，从而抑制电能消耗。

ACPI 具有两个用来控制 C 状态的接口。

1. Processor 寄存器块 (P\_BLK) 的 P\_LVL2 和 P\_LVL3 寄存器。
2. 定义块中定义的处理器的对象列表的 \_CST 对象。

使用 \_CST 的 C 状态控制对使用 P\_BLK 的控制进行了扩展。它可以使使用称为 Functional 固定硬件 (FFH) 接口的厂商固有接口。可以根据 CPU 架构使用 C 状态，如 Intel 公司的 CPU 使用的就是 MWAIT 命令等。从而可以使用 ACPI 中未定义的 C 状态。\_CST 对象针对各个状态分别定义了下列内容。

Register	处理器进入 Type 所示的 C 状态时必须读出的位置
Type	表示用于 C1、C2、C3 等中的哪个 C 状态的字段
Latency	进入 C 状态时最差情况下的延迟时间 (毫秒)
Power	处于 Type 所示 C 状态时处理器的平均耗电量

Linux 中有用来控制包括 x86 在内的各种 CPU 空闲状态的 cpuidle 子系统。通过 cpuidle 子系统可以设置 CPU 空闲时的轮询 (polling) 方法和策略 (governor)。从而可以将每个 CPU 的空闲状态的特性抽象化，使用 CPU 固有功能或 ACPI 的功能。

控制 Linux 的 cpuidle 的用户接口位于 /sys/devices/system/cpu/cpuidle/ 下。各接口的说明如下所示。

available_governors	可利用的策略
current_governor_ro	显示当前的策略
current_governor	通过读出，可以显示当前的策略；通过写入，可以切换
current_driver	当前 cpuidle 子系统的驱动程序

Linux 中现有策略列表如下。

menu	Linux 中使用的标准策略
ladder	阶段性改变状态的策略。使用以往的定期中断计时器时可以顺利运行，而最近的 Linux 内核的计时器并不使用定期计时器中断。

menu 策略从中断等情况预测可以休眠的时间，选择与该休眠时间相符的最深 C 状态。使用 CPU 负载、中断、I/O 负载等进行预测。

基本上可以使用默认的 cpuidle 的策略来控制电能消耗。策略虽然有在运行中可以变更的接口，但切换的接口是面向开发者的，启动中不应进行切换。

每个 CPU 的 C 状态信息可以从 /sys/devices/system/cpu/cpuN/cpuidle/stateM/ 下获取 (N、M 为整数。N 为 CPU 编号，M 为 C 状态的状态编号)。

desc	状态 M 下 cpuidle 的说明
latency	到恢复为止的延迟时间
name	状态 M 的名称 (C1、C2、C3 等)
power	状态 M 运行时的电量
time	使用状态 M 的时间
usage	使用状态 M 的次数

除 sysfs 以外，还有可以通过 proc 文件系统参照的信息。其示例如下所示。/proc/acpi/processor/cpuN/power (N 为 CPU 编号)。

```
# cat /proc/acpi/processor/CPU0/power
active state:          C0
max_cstate:           C8
maximum allowed latency: 2000000000 usec
states:
  C1:                  type[C1] promotion[--] demotion[--] latency[001]
usage[000008972] duration[00000000000000000000]
  C2:                  type[C2] promotion[--] demotion[--] latency[001]
usage[00339308] duration[00000000000238036741]
  C3:                  type[C3] promotion[--] demotion[--] latency[017]
usage[05986201] duration[0000000089960255002]
# cat /proc/acpi/processor/CPU0/info
processor id:          0
acpi id:              0
bus mastering control: yes
power management:      yes
throttling control:    no
limit interface:       no
```

### 关于 sched\_mc\_power\_savings、sched\_smt\_power\_savings

/sys/devices/system/cpu 下存在 sched\_mc\_power\_savings、sched\_smt\_power\_savings 这些对进程调度程序进行调整的参数。它们可以在多核、同时多线程 (Intel Hyper-threading 等) 的 CPU 中发挥作用。

#### sched\_mc\_power\_savings

设置为 0 或 1。设置为 1 时，进程调度程序尽量仅针对某个特定 CPU 套接字调度进程数。只有在运行中的所有内核都繁忙时，才能对位于其他 CPU 套接字的核进行进程调度。这在每个核具有各自的 C 状态，所有的核超过某个特定 C 状态才能进入更深 C 状态的 CPU 中有效。

#### sched\_smt\_power\_savings

设置为 0 或 1。设置为 1 时，进程调度程序尽量仅对位于某个特定内核的线程进行调度。只有在运行中的所有线程繁忙时，才对其他内核的线程进行调度。这与上述 sched\_mc\_power\_savings 同样，对于每个线程具有各自的 C 状态的 CPU 有效。

Linux 下 x86 CPU 中的 C 状态运行情况如下所示。

1. Linux 下一旦变成空闲状态，就会调度 `idle` 进程，调用 `pm_idle` 函数。
2. 在使用 `cpuidle` 子系统的系统下，`pm_idle` 调用 `cpuidle_idle_call()`。
3. `cpuidle_idle_call()` 中通过 `cpuidle governor` 的 `select` 方法选择下一个状态。
4. 每个状态各自调用登录的 `enter` 方法，迁移到所选择的状态。

想要确保 CPU 的 bug 和应答性时，有时需要限制 C 状态。这时通过使用 `max_cstate` 这一内核启动参数，就可以限制 C 状态的最大值。向 `max_cstate` 输入希望使用的最深层的 C 状态的值。将 C1 设置为最大值时，在 `/etc/grub.conf` 中设置 `max_cstate=1`。

```
title Fedora (2.6.35.12-88.fc14.i686)
root (hd0,0)
kernel /boot/vmlinuz-2.6.35.12-88.fc14.i686 ro root=UUID=82f8f09d-b0d7-49ee-
a57b-13e6484bd840 rd_NO_LUKS rd_NO_LVM rd_NO_MD rd_NO_DM LANG=ja_JP.UTF-8
KEYTABLE=jp106 rhgb quiet processor.max_cstate=1
initrd /boot/initramfs-2.6.35.12-88.fc14.i686.img
title Fedora (2.6.35.11-83.fc14.i686)
```

使用 `max_cstate` 后，可以在 `/proc/acpi/processor/CPUN/power` (`N` 为 CPU 编号) 确认 `max_cstate` 已变更。

```
# cat /proc/acpi/processor/CPU0/power
active state:          C0
max_cstate:            C1
maximum allowed latency: 2000000000 usec
states:
  C1:                  type[C1] promotion[--] demotion[--] latency[001]
usage[00000000] duration[00000000000000000000]
```

## P 状态的使用方法

P 状态是 CPU 正在运行时的电力状态。通过设置为更深的 P 状态就可以控制电能消耗。P 状态下是通过降低频率来降低电能消耗的，因此性能和电能之间是不可兼得的关系。Linux 下可以根据 CPU 的使用情况来设置 P 状态。另外，与 C 状态同样，根据用户设置的策略 P 状态有不同的性能。

与 P 状态相关的 ACPI 接口如下。在启用 ACPI 的系统中，在 Linux 下使用这些接口来控制 `cpufreq` 驱动程序。

`_PCT (Performance Control)`

使处理器进入 P 状态的接口。向 `PERF_CTRL` 写入控制寄存器 (control register) 值来进入 P 状态。根据使用 `_PPC` 方法得到的值来选择 P 状态的值。写入时使用的控制寄存器值是通过 `_PPC` 的 `Control` 字段获得的值。

写入成功后，`PERF_STATUS` 的值与相关的 `_PSS` 的 `Status` 字段相同。在 x86 的情形下，`PERF_CTRL`、`PERF_STATUS` 安装在 PIO 或 MSR 中 (作为 Functional 固定硬件 (FFH) 安装时)。

Performance Control Register (PERF_CTRL)	通过写入来进入 P 状态
Performance Control Register (PERF_STATUS)	确认向 PERF_CTRL 写入后已进入 P 状态。如果与 _PSS 的 Status 是相同的值，则表示已进入 P 状态

#### \_PSS (Performance Supported Status)

该接口用来显示处理器支持的 P 状态。各 P 状态中分别保存了 CPU 频率、电能消耗、控制寄存器值、status 寄存器值。

Core Frequency	Px 状态下 CPU 内核的运行频率 (MHz)
Power	Px 状态下的最大耗电量 (mW)
Latency	迁移到 Px 状态时 CPU 变得无效的最大延迟时间 (ms)
Bus Master Latency	迁移到 Px 状态时阻碍总线主控器访问内存的最大延迟时间 (ms)
Control	写入 _PCT 的 PERF_CTRL 的控制寄存器值
Status	用来与从 _PCT 的 PERF_STATUS 读取的值作比较的值

#### \_PPC (Performance Present Capabilities)

\_PPC 用于动态显示某个时间点平台所支持的 P 状态值。操作系统等从 \_PSS 内的入口中，选择电能低于 \_PPC 所得的值的 P 状态。例如，当 \_PSS 有 P0 ~ P3 的入口，而 \_PCC 显示为 1 时，必须从 P1 状态到 P3 状态中选择 P 状态。\_PPC 的值是在连接、切断如 DCMI<sup>注2</sup> 等这样的外部接口或电源适配器 (AC adapter) 时，通过 BIOS 动态变更。\_PPC 的值一旦变更，就会发生通知事件，操作系统再次执行 \_PPC。

#### \_PSD (P-State Dependency)

P 状态的控制中用来获取 CPU 之间依存关系的信息的对象如下所示。

NumEntries	ACPI4.0 中为 5。包括 NumEntries 在内的 _PSD 入口数
Revision	ACPI4.0 中为 0。_PSD 的修订编号
Domain	该 P 状态入口所属的域名的编号
CoordType	设置为下列类型之一。 SW_ALL (要改变 P 状态时软件必须改变所有 CPU) SW_ANY (要改变 P 状态时软件必须改变域名中的 CPU) HW_ALL (要改变 P 状态时硬件执行域名的迁移)
Num Processor	有依存关系的 CPU 数量。只有在有依存关系的所有 CPU 都在该 P 状态以下时才能迁移

注 2: Data Center Manageability Interface。

在 Linux 的 cpufreq 子系统中，存在用来运行 cpufreq 的驱动程序和决定驱动程序运行策略的驱动程序。运行 cpufreq 的驱动程序在每个 CPU 或系统中各不相同，而在支持 ACPI 的系统中基本上使用的是 acpi-cpufreq 驱动程序。由于一些系统的 acpi-cpufreq 无法正常运行，因此有的系统使用 powernow-k8、p4-clockmod、speedstep-centrino 等驱动程序。

Linux 中可使用的省电 governor 如下。

performance	以可设置的最高频率运行
powersave	以可设置的最低频率运行
ondemand	根据系统负载更改频率
conservative	根据系统负载缓慢更改频率
userspace	可以从用户应用程序设置 P 状态

performance、powersave 作为 governor 比较容易理解，但有时 ondemand、conservative、userspace 会不清楚应使用哪一个。一般多使用 ondemand 或 userspace。ondemand 在内核内测定负载，根据负载改变频率。因此，就可以在不察觉性能降低的情况下降低电能消耗。但是由于需要立刻改变频率的功能，因此并不是所有 CPU 中都可以使用。在 userspace 中 cpuspeed 等守护进程考虑 /etc/sysconfig/cpuspeed 等的设置内容来改变频率。在 userspace 中检测负载、更改频率是有延迟的，因此在负载变化剧烈的系统中，系统的应答时间可能会很紧迫。conservative 和 ondemand 的区别就是 conservative 并非无论在哪种负载下都立即设置为最大频率，而是慢慢地改变频率。这个功能在笔记本电脑之类使用电池的系统中可以有效地发挥抑制电能消耗的效果。

需要注意的是，一般来说，performance 和 powersave 并不是以抑制电能消耗为目的的 governor。performance 无论何时都以最高频率运行，因此无法抑制电能消耗。powersave 由于降低运行频率，因此可以抑制电能消耗，但只有在进程几乎停止时才有效果。频率降低一般会导致运行时间增加，因此就会造成空闲时间（可以进入 C 状态的时间）减少，消耗的电能增加。

Linux 的 cpufreq 子系统在 /sys/devices/system/cpu/cpuN/cpufreq 下提供了下列（见表 6-12）与 CPU 性能相关的接口（N 为 CPU 编号）。

表 6-12 cpufreq 子系统的接口

接 口	功 能
affected_cpus	因 P 状态的改变而受影响的 CPU。这里的 CPU 编号的 CPU 同时进入 P 状态
cpuinfo_cur_freq	当前的 CPU 频率 (kHz)
cpuinfo_max_freq	CPU 最高频率 (kHz)

接 口	功 能
<code>cpuinfo_min_freq</code>	CPU 最低频率 (kHz)
<code>cpuinfo_transition_latency</code>	改变频率所需的延迟时间 (nm)
<code>related_cpus</code>	与 <code>affected_cpus</code> 的意思相同, 但考虑 <code>PSD</code> 的 <code>CoordType=HW_ALL</code> 时的情况
<code>scaling_available_frequencies</code>	可利用的频率列表
<code>scaling_available_governors</code>	可利用的 governor 名列表
<code>scaling_cur_freq</code>	最后设置的当前 CPU 频率 (kHz)。由于不使用驱动程序重新获取值, 因此有可能与 <code>cpuinfo_cur_freq</code> 的值不同 (BIOS 等有时会更改 CPU 频率)
<code>scaling_driver</code>	用来处理 <code>acpi-cpufreq</code> 这样的 cpufreq 的驱动程序名称
<code>scaling_governor</code>	governor 的驱动程序名称。通过写入这个文件就可以更改 governor
<code>scaling_max_freq</code>	governor 可利用的最高频率。从 <code>scaling_available_frequencies</code> 中选择频率, 通过写入这个文件就可以更改最高频率
<code>scaling_min_freq</code>	governor 可利用的最低频率。从 <code>scaling_available_frequencies</code> 中选择频率, 通过写入这个文件就可以更改最高频率
<code>scaling_setspeed</code>	使用 userspace governor 时, 通过在这里写入频率的值, 就可以更改 CPU 频率

使用 `cpufrequtils` 等就可以使上述接口更容易使用。下面是 `cpufrequtils` 中包含的 `cpufreq-info` 的例子。

```
# cpufreq-info
cpufrequtils 005: cpufreq-info (C) Dominik Brodowski 2004-2006
Report errors and bugs to cpufreq@vger.kernel.org, please.
analyzing CPU 0:
  driver: p4-clockmod
  CPUs which need to switch frequency at the same time: 0
  hardware limits: 317 MHz - 2.53 GHz
  available frequency steps: 317 MHz, 633 MHz, 950 MHz, 1.27 GHz, 1.58 GHz,
1.90 GHz, 2.22 GHz, 2.53 GHz
  available cpufreq governors: ondemand, userspace, performance
  current policy: frequency should be within 317 MHz and 2.53 GHz.
                  The governor "userspace" may decide which speed to use
                  within this range.
  current CPU frequency is 2.22 GHz (asserted by call to hardware).
  cpufreq stats: 317 MHz:0.00%, 633 MHz:0.00%, 950 MHz:0.00%, 1.27
GHz:0.00%, 1.58 GHz:0.00%, 1.90 GHz:0.00%, 2.22 GHz:0.00%, 2.53 GHz:0.00% (5)
```

在安装了 `cpufreq-stats` 内核模块的情况下, `/sys/devices/system/cpu/cpuN/cpufreq/stats` 下还准备了下列接口, 用于 `powertop` 等应用程序 ( $N$  为 CPU 编号)。

<code>time_in_state</code>	各 P 状态下使用的时间
<code>total_trans</code>	改变 P 状态的次数
<code>trans_table</code>	各 P 状态进入不同 P 状态的次数列表

`ondemand` 中提供了下列接口 (见表 6-13), 以便进行更加详细的调整。

表 6-13 `ondemand` 的接口

接 口	功 能
<code>ignore_nice_load</code>	设置为 0 或 1。设置为 1 时, 具有 nice 值的进程所使用的时间在 <code>cpufreq</code> 中不参加计算。不用于判断 P 状态的更改
<code>powersave_bias</code>	设置为 0~1000 的值。表示 0.1%~100%, 减少按照指定比例改变的频率。当 CPU 频率降低对用户几乎没有影响时, 用来减少电能消耗
<code>sampling_rate</code>	对 CPU 使用率进行采样的间隔 (微秒)。一般为 10000 左右的值, 默认值为 <code>transition_latency*1000</code>
<code>sampling_rate_max</code>	<code>sampling_rate</code> 可以设置的最大值
<code>sampling_rate_min</code>	<code>sampling_rate</code> 可以设置的最小值
<code>up_threshold</code>	判断是否要更改 P 状态的 CPU 使用率的阈值

使用 `ondemand governor` 时 `acpi-cpufreq` 的运行情况如下。

1. `cpufreq_ondemand` 驱动程序将初始化时用来计算负载的函数输入到计时器。
2. `cpufreq_ondemand` 驱动程序定期根据空闲时间测量系统负载, 决定要改变的 P 状态。
  - CPU 使用率一旦超过 `up_threshold` 中指定的值, 就将频率提高到 `scalling_max_freq`。
  - CPU 使用率如果保持 `up_threshold-10` 以下的状态, 就将频率逐渐降低到 `scalling_min_freq` 为止。
3. `cpufreq` 驱动程序 (`acpi-cpufreq` 等) 让 CPU 进入期望实现的 P 状态。

`conservative` 中提供的接口如表 6-14 所示。

表 6-14 `conservative` 中提供的接口

接 口 名 称	功 能
<code>down_threshold</code>	判断是否要降低 P 状态的 CPU 使用率的阈值。CPU 使用率一旦低于 <code>down_threshold-10</code> , 就判断为降低 P 状态

接 口 名 称	功 能
<code>freq_step</code>	提高 / 降低 CPU 频率的比例。为 0 则不改变频率。为 100 则与 <code>ondemand</code> 同样运行
<code>ignore_nice_load</code>	设置为 0 或 1。设置为 1 时，具有 nice 值的进程所使用的时间在 <code>cpufreq</code> 中不参加计算。不用于判断 P 状态的更改
<code>sampling_down_factor</code>	赋予 <code>sampling_rate</code> 的乘数因子。用于降低采样频率
<code>sampling_rate</code>	对 CPU 使用率进行采样的间隔 (μs)。 <code>sampling_rate</code> × <code>sampling_down_factor</code> 就是采样间隔
<code>sampling_rate_max</code>	<code>sampling_rate</code> 可以设置的最大值
<code>sampling_rate_min</code>	<code>sampling_rate</code> 可以设置的最小值
<code>up_threshold</code>	判断是否要提高 P 状态的 CPU 使用率的阈值

在使用服务器等系统中，有时想要抑制电力峰值但是不停止服务器。这种情况下可以使用 P 状态限制电力峰值。下面介绍能够方便地限制 P 状态的设置方法。`cpuspeed` 服务中可以在 `/etc/sysconfig/cpuspeed` 中进行限制 P 状态的设置。下列是将 P 状态固定在 800MHz 的例子。

```
MAX_SPEED=800000
MIN_SPEED=800000
```

对 `/etc/sysconfig/cpuspeed` 进行编辑后，重新启动 `cpuspeed` 服务，设置就会发挥作用。在 `scalling_max_freq`、`scalling_min_freq` 中设置 `MAX_SPEED`、`MIN_SPEED`。需要注意的是，如果删除 `/etc/sysconfig/cpuspeed` 中设置的值，则需要手动重新设置 `scalling_max_freq`、`scalling_min_freq` 等，或者重新启动操作系统。

另外，如图 6-3 和图 6-4 所示，在可以使用 GUI 的环境中，`gnome-applets` 数据包中包含的 CPU 频率测量监控器 (CPU Frequency Scalling Monitor)，这个小应用程序的使用也十分方便。

## Intel® 智能加速技术

`powertop` (参考 Hack #50) 是确认 P 状态的工具之一。但是通过 `powertop` 命令仅能显示 P 状态的额定频率。

下面是在 Intel® Xeon® CPU E3-1225 的机器 (Sandy Bridge) 上执行 `powertop` 时的输出内容。可以确认以 1.60 ~ 3.11GHz 运行时的情况。

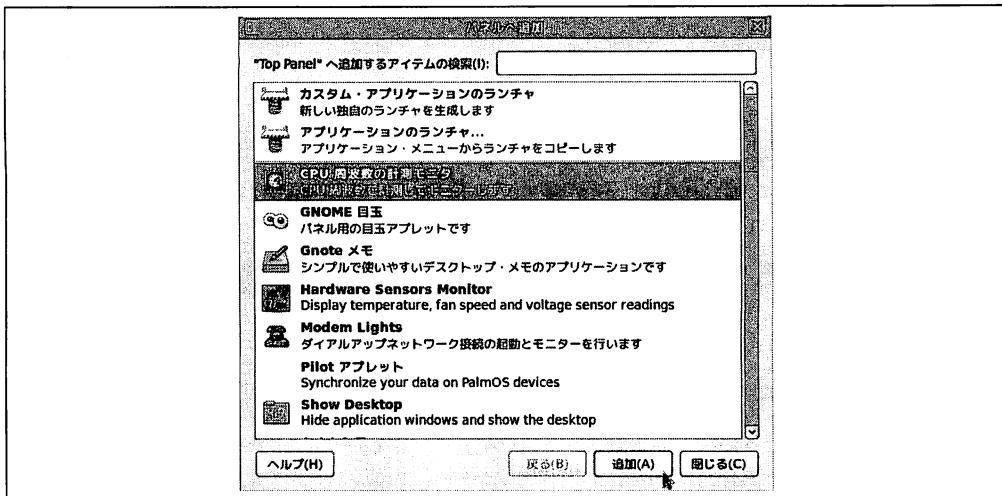


图 6-3 添加 CPU 频率测量监控器小程序 (monitor applet)

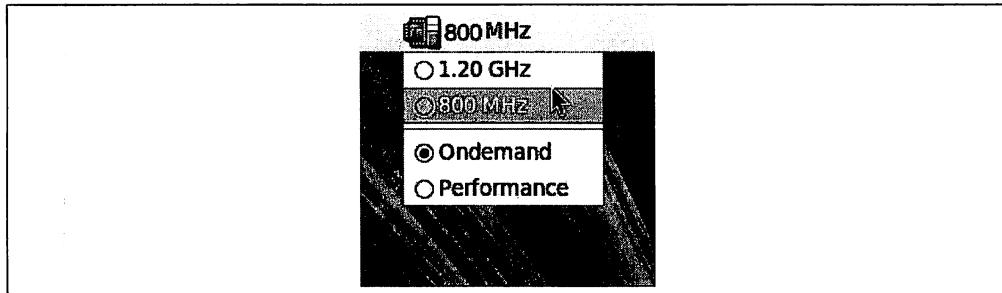


图 6-4 使用 CPU 频率测量监控器的例子

# powertop

```
PowerTOP version 1.11          (C) 2007 Intel Corporation

C 状态      平均滞留时间      P 状态 (频率)
C0 (CPU 运行状态)      (13.7%)      3.11 GHz      0.0%
轮询      C1 halt      0.0m      3.10 GHz      0.0%
C1 halt      0.3ms ( 0.0%)      2.90 GHz      0.0%
C3 mwait      39.5ms (86.2%)      2.71 GHz      0.0%
                                         1.60 GHz      100.0%
```

平均 1 秒的 CPU 分配次数: 23.4 时间间隔: 0.6 秒

具有智能加速技术 (Intel Turbo Boost Technology) 的 CPU 自动以高于固定频率的频率运行。从内核 2.6.38 开始, 内核源代码内都包含 turbostat 命令, 可以确认智能加速的运行频率。

下载内核 2.6.38, 编译 turbostat。

```
# wget -t0 -c http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.38.tar.bz2
# tar jxvf linux-2.6.38.tar.bz2
```

```
# cd linux-2.6.38/tools/power/x86/turbostat  
# make ; make install
```

在没有运行任何代码的状态下执行 turbostat。

```
# turbostat -v  
GenuineIntel 13 CPUID levels; family:model:stepping 0x6:2a:7 (6:42:7)  
16 * 100 = 1600 MHz max efficiency 以最低电压运行的频率  
31 * 100 = 3100 MHz TSC frequency 额定频率  
32 * 100 = 3200 MHz max turbo 4 active cores 4个内核运行时的频率  
33 * 100 = 3300 MHz max turbo 3 active cores 3个内核运行时的频率  
33 * 100 = 3300 MHz max turbo 2 active cores 2个内核运行时的频率  
34 * 100 = 3400 MHz max turbo 1 active cores 1个内核运行时的频率  
core CPU %c0 GHz TSC %c1 %c3 %c6 %c7 %pc2 %pc3 %pc6 %pc7  
0 0 0.01 1.59 3.09 0.02 0.00 99.96 0.00 6.48 0.00 93.40 0.00  
1 1 0.01 1.60 3.09 0.02 0.00 99.97 0.00 6.48 0.00 93.40 0.00  
2 2 0.01 1.60 3.09 0.02 0.00 99.98 0.00 6.48 0.00 93.40 0.00  
3 3 0.01 1.60 3.09 0.02 0.00 99.96 0.00 6.48 0.00 93.40 0.00
```

所有的 CPU 以约 1.60GHz 的频率运行，然后启动一个通过无限循环利用 CPU 的进程。

```
# while [ 0 ] ; do : ; done &  
# turbostat  
core CPU %c0 GHz TSC %c1 %c3 %c6 %c7 %pc2 %pc3 %pc6 %pc7  
0 0 24.87 3.39 3.09 1.15 0.00 73.98 0.00 0.00 0.00 0.00 0.00  
1 1 0.02 3.29 3.09 0.02 0.00 99.95 0.00 0.00 0.00 0.00 0.00  
2 2 98.59 3.39 3.09 1.41 0.00 0.00 0.00 0.00 0.00 0.00 0.00  
3 3 0.85 3.29 3.09 3.18 0.00 95.97 0.00 0.00 0.00 0.00 0.00  
3 3 0.00 3.29 3.09 0.01 0.00 99.99 0.00 0.00 0.00 0.00 0.00
```

一直提高到 3.39GHz，可以看出智能加速正在运行。turbostat 的详细情况请参考 man 8 turbostat。

## 小结

性能和省电之间是平衡的关系。建议根据系统分析需要什么样的性能，再启用省电功能。

## 参考文献

- 英特尔®智能加速技术

<http://www.intel.co.jp/jp/technology/turboboost/>

——Akio Takebe、Naohiro Ooiwa

## HACK #43 PCI 设备的热插拔

本节介绍 PCI 设备的热插拔。

热插拔功能，是指在不停止系统的情况下向系统添加、删除设备的功能。PCI 中也考虑了热插拔的结构，在部分系统中可以使用。这与 USB 的热插拔是不同的，需要首先了解热插拔的流程才能更好地使用 PCI 设备的热插拔，因此本节对其概述并介绍 Linux 下

的使用方法。

PCI 相关设备的热插拔规格有使用 PCI Standard Hot-Plug Controller (SHPC) 的热插拔、PCI Express 的热插拔、ACPI 热插拔、厂商固有功能的热插拔等。其中，作者认为使用最广的热插拔规格是 SHPC。理解 SHPC 的规格就可以理解其他 PCI 热插拔的流程。PCI Express 的热插拔基本上也是与 SHPC 规格相同的。

SHPC 是安装在 PCI-to-PCI 桥或 PCI 主桥上的热插拔用控制器。SHPC 中定义了 Standard Usage Model 这个热插拔的标准用户操作，其中构成 Standard Usage Model 的要素参见表 6-15。

表 6-15 构成 SHPC Standard Usage Model 的要素

要素	目的
Indicators	显示插槽 (slot) 的 power 和 attention 状态
Manual-operated Retention Latch (MRL)	固定 PCI 卡的物理锁扣 (latch)
MRL Sensor	SHPC 和操作系统用来检测 MRL 已打开
Electromechanical interlock	防止插槽通电时拔出 PCI 卡
Attention Button	使用户能够进行热插拔操作的硬件结构
Software User Interface	使用户能够进行热插拔操作的软件接口
Slot Numbering	物理插槽的识别符

Indicator 有 Power Indicator (见表 6-16) 和 Attention Indicator (见表 6-17)。各个 Indicator 分别有 on (亮灯)、off (灭灯)、blinking (闪烁) 这三个状态，Indicator 由软件完全控制，可以单独显示各个状态。

表 6-16 Attention Indicator (黄色或橙色)

状态	说明
off (灭灯)	灭灯。通常状态
on (亮灯)	亮灯。操作上发生问题的状态
blinking (闪烁)	闪烁。出现在操作系统进行了让用户识别指定插槽位置的操作时

表 6-17 Power Indicator (绿色)

状态	说明
off (灭灯)	灭灯。插槽未通电的状态。此时可以插拔 PCI 卡
on (亮灯)	亮灯。插槽通电的状态。此时不允许插拔 PCI 卡
blinking (闪烁)	闪烁。插槽渐渐通电或渐渐断电的状态。此时不允许插拔 PCI 卡

## Hot-add 的流程

SHPC 中定义了使用软件接口的 Hot-add 和使用 Attention 按钮的 Hot-add，这里介绍使

用软件接口的 Hot-add 的流程。

1. 打开 MRL (物理固定 PCI 卡的锁扣)。
2. 插入 PCI 卡, 关闭 MRL, 安装连接线。
3. 用户从软件发布让插槽变为可使用状态的要求。此时 Power Indicator 仍然为 off (灭灯) 状态。
4. 用户从软件接口通知可以将插槽变为可使用状态。此时 Power Indicator 变为 blinking (闪烁) 状态。
5. 用户等待插槽完全变为可使用状态。完全变为可使用状态后, Power Indicator 变为 on (亮灯) 状态。

## Hot-remove 的流程

Hot-remove 与 Hot-add 同样定义了使用软件接口的 Hot-remove 和使用 Attention 按钮的 Hot-remove。这里介绍使用软件接口的 Hot-remove 的流程。

1. 用户选择 MRL 关闭, 插槽为可使用状态的 PCI 卡, 使用软件发布要求让插槽变为不可使用状态的要求。此时 Power Indicator 仍为 on 状态。
2. 用户从软件接口通知可以将插槽变为不可使用状态。此时 Power Indicator 变为 blinking 状态。
3. 用户等待插槽变为不可使用状态。插槽完全变为不可使用状态后, Power Indicator 变为 off。
4. 用户拆除连接线, 打开 MRL, 拆除 PCI 卡。

## 确认热插拔功能

SHPC 可以作为 PCI 桥的功能来确认。使用 `lspci-vvvv` 命令确认时, 如果 Capability 列表中有 SHPC Capability ID (0x0c), 就可以确认具有 SHPC 功能 (①)。实际上, 也有 MRL 或 BIOS 等系统不支持的很多情况, 因此使用的系统如果不是厂商支持的系统, 则很多情况下无法使用。

### 例 6-1 `lspci-vvvv` 的例子

```
ad:00.0 PCI bridge: Intel Corporation 6700PXH PCI Express-to-PCI Bridge A (rev 09) (prog-if 00 [Normal decode])
    Control: I/O- Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr+
    Stepping- SERR+ FastB2B-
    Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort-
    <TAbort- <MAbort- >SERR- <PERR-
    Latency: 0, Cache Line Size: 128 bytes
    Region 0: Memory at d78fe000 (64-bit, non-prefetchable) [size=4K]
    Bus: primary=ad, secondary=ae, subordinate=af, sec-latency=64
    I/O behind bridge: 0000f0000-00000fff
    Memory behind bridge: d6a00000-d77fffff
    Prefetchable memory behind bridge: 000000f580000000-000000f5bff00000
    Secondary status: 66MHz+ FastB2B+ ParErr- DEVSEL=medium >TAbort-
```

```

<TAbsort- <MAbsort+ <SERR- <PERR-
BridgeCtl: Parity+ SERR+ NoISA- VGA- MAbsort- >Reset- FastB2B-
Capabilities: [44] Express PCI/PCI-X Bridge IRQ 0
Device: Supported: MaxPayload 256 bytes, PhantFunc 0, ExtTag-
...
Link: Speed 2.5Gb/s, Width x4
Capabilities: [5c] Message Signalled Interrupts: 64bit+ Queue=0/0
Enable-
Address: 0000000000000000 Data: 0000
Capabilities: [6c] Power Management version 2
Flags: PMEClk- DSI- D1- D2- AuxCurrent=0mA PME(D0+,D1-,D2-
,D3hot+,D3cold+)
Status: D0 PME-Enable- DSel=0 DScale=0 PME-
Capabilities: [78] #0c [0004] ...
Capabilities: [d8] PCI-X bridge device
Secondary Status: 64bit+ 133MHz+ SCD- USC- SCO- SRD- Freq=conv
Status: Dev=ad:00.0 64bit- 133MHz- SCD- USC- SCO- SRD-
Upstream: Capacity=65535 CommitmentLimit=65535
Downstream: Capacity=65535 CommitmentLimit=65535
Capabilities: [100] Advanced Error Reporting
Capabilities: [300] Power Budgeting

```

## Linux 的热插拔子系统

Linux 中有用于 PCI 热插拔的子系统, SHPC、PCI Express、ACPI 热插拔基本上是相同的。SHPC、PCI Express、ACPI 热插拔所使用的驱动程序如下表所示。

规 格	驱动程序名称
SHPC	shpchp
PCIExpress	pciehp
ACPI	acpihp

首先, 需要安装热插拔驱动程序 shachp。在 SHPC 的情况下, 使用下列命令把 shachp 安装到内核。

```
# modprobe shpchp
```

将驱动程序安装到内核后, 会在下列路径中显示可以热插拔的插槽编号。

```
/sys/bus/pci/slots
```

需要注意的是, 这个插槽编号与 PCI 中的一般 Segment:Bus:Device:Function 编号并不相同。基本上就是系统厂商提供的使用手册中记载的插槽编号。

将 0 写入对象插槽编号的目录下的 power 文件, 就会执行 Hot-remove。Power Indicator 变为 off (灭灯) 表示 Hot-remove 完成, 可以拆除设备。

```
# echo 0 > /sys/bus/pci/slots/ <插槽编号> /power
```

注意事项: 将设备进行 Hot-remove 时, 需要先停止正在使用的服务。

对于 Hot-add 的情况, 请在安装设备后对 power 写入 1。

```
# echo 1 > /sys/bus/pci/slots/ <插槽编号> / power
```

## 小结

KVM/Xen 虚拟环境等支持 ACPI 的 PCI 热插拔, 接触 PCI 热插拔的机会也增多。将不使用的 PCI 设备暂时 Hot-remove 也可以省电, 大家可以尝试一下。

——Akio Takebe

# HACK #44 虚拟环境下的省电

本节介绍在使用 KVM/Xen 时虚拟环境下的省电方法。

怎样才能在虚拟环境下省电? 掌握了基本思想就可以理解虚拟环境下的省电方法。

## 虚拟环境下的省电思想

例如, 某个客户端进入 S3 状态, 但其他客户端可能仍在运行, 因此不能将主机操作系统更改为 S 状态。因此需要将客户端操作系统内的电力管理和主机操作系统内的电力管理分开考虑。

### 客户端操作系统内的电力管理

客户端操作系统不对其他客户端、主机操作系统或管理程序造成影响, 不进行实际的电力管理, 但有时会成为主机操作系统或管理程序的电力管理的触发器 (trigger) (例如, 也可以当所有客户端操作系统都进入 S3 状态时对主机操作系统进入 S3 等的配置)。

- 关于客户端操作系统内的 S 状态

从客户端操作系统内可以进行待机等, 因此客户端操作系统管理者不使用客户端操作系统时可以通过使用 S 状态来省电。

- 关于客户端操作系统内的 C 状态

虚拟 CPU 变为空闲时就会发出 hlt 命令, 被排除在管理程序的调度对象之外。因此, 即使在客户端操作系统的各 C 状态下进行不同的运行也没有意义, 于是多数管理程序中没有配置客户端操作系统的 C 状态。

- 关于客户端操作系统内的 P 状态

与 C 状态基本相同, 即使配置也没有意义, 因此多数管理程序中没有配置客户端 P 状态。

### 主机操作系统内的电力管理

主机操作系统或管理程序内的电力管理是通过控制实际的硬件来进行电力控制的。

- 关于主机操作系统的 S 状态

如图 6-5 所示, 从管理程序来看, 所有客户端操作系统的虚拟 CPU 都被操作系统看做进程同样处理。因此, 保存客户端的虚拟 CPU 上下文等的方法, 与操作系统变为 S 状态时保存进程状态的方法相同。像 Xen 管理程序一样不进行磁盘 I/O 的管理程序, 不将内存内容写入磁盘, 因此有时不支持 S4 状态。

- 主机操作系统上的 C 状态

与非虚拟环境的普通操作系统一样, 根据主机操作系统或管理程序中设置的策略选择 C 状态。虚拟环境下需要注意的是客户端操作系统的中断频率。C 状态是通过中断来恢复的, 因此需要尽量减小中断的频率才能产生省电的效果。虚拟环境下有多个操作系统是中断的发生源, 因此如果某一个客户端操作系统内频繁中断, 则该客户端操作系统使用的 CPU 就无法进入深层 C 状态。也就是说, 想要通过 C 状态来省电, 就必须努力减小所有客户端操作系统上的中断次数。

- 主机操作系统上的 P 状态

主机操作系统上的 P 状态导致的性能变化直接反映客户端操作系统的性能。有时即使认为客户端操作系统的 CPU 以 3GHz 的频率运行, 实际却是设置为 600MHz 的。可能有人会担心这一点, 但客户端操作系统并不是原本就占用了 CPU 所有时间的。即使物理 CPU 原来具有 3GHz 的性能, 实际也可能只使用其一半的时间, 因此客户端操作系统降低运行中的 CPU 频率本身是没问题的。在主机操作系统中改变 P 状态时, 根据 CPU 使用率决定进入哪个 P 状态, 但管理程序的种类或设置不同, 运行情况有时也有所不同, 这将在后面介绍。

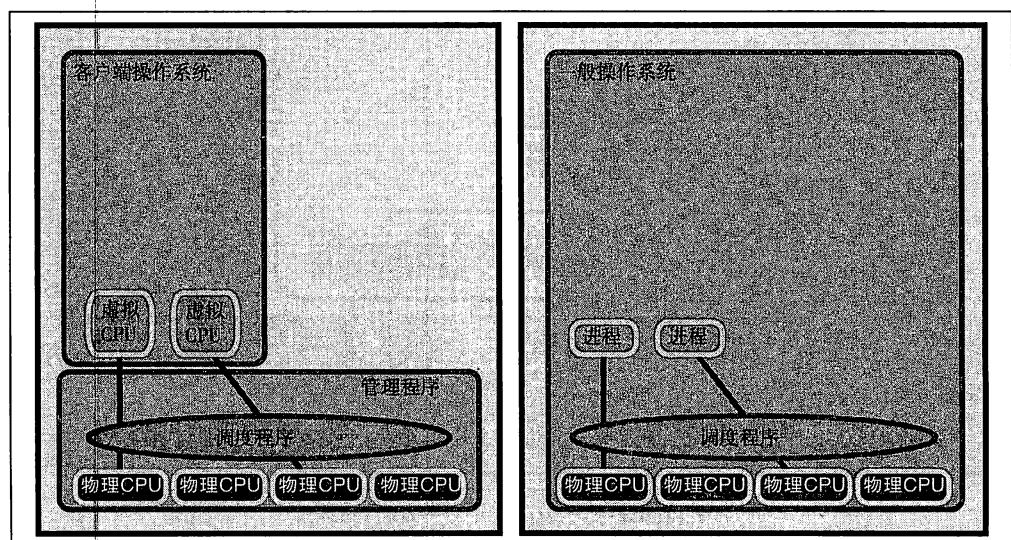


图 6-5 管理程序的虚拟 CPU 和操作系统的进程

## Xen 的 P 状态

Xen 是将管理程序和特权客户端 Dom0 作为主机操作系统的管理程序。Xen 准备了管理程序控制 P 状态的方法和 Dom0 控制 P 状态的方法。RHEL5 等 Linux 发布版采用的是 Dom0 控制 P 状态的方式。使用 Dom0 控制 P 状态的方式时，通过下列命令添加 Xen 的启动选项 `cpufreq=dom0-kernel`。在 RHEL5 等中，这个选项是默认的。这个方式的优点是可以在 Dom0 上使用 `cpuspeed` 等工具，因此可以设置与不使用 Xen 时的 Linux 相同的策略。

```
title Xen
root (hd0,0)
kernel /boot/xen.gz cpufreq=dom0-kernel
module /boot/vmlinuz-2.6.18-8-xen ro root=/dev/sda1
module /boot/initrd-2.6-xen.img
```

但是 `cpufreq=dom0-kernel` 的情形必须由 dom0 的虚拟 CPU 直接控制物理 CPU 的 P 状态，因此所有的虚拟 CPU 必须与所有物理 CPU 一对一固定，如图 6-6 所示。指定 `cpufreq=dom0-kernel` 时，会自动执行将虚拟 CPU 与物理 CPU 固定的处理，但如果通过 `dom0_max_vcpus` 启动选项将虚拟 CPU 的数量指定为少于物理 CPU，或者无法将虚拟 CPU 固定分配给物理 CPU 时，`cpufreq=dom0-kernel` 选项可以忽略。

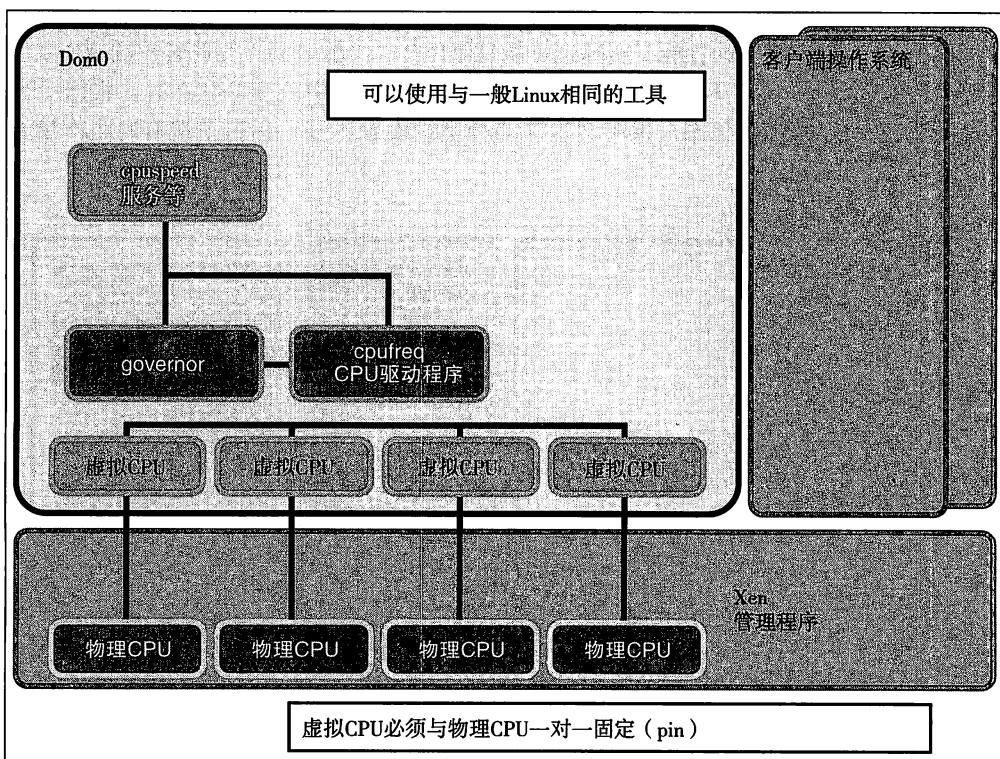


图 6-6 `cpufreq=dom0-kernel` 时虚拟 CPU 和物理 CPU 的关系图

Xen3.4 以后可以通过在启动选项中指定 `cpufreq=xen` 来使用管理程序控制 P 状态的方式（也有 RHEL5 等无法使用的发布版）。策略等的设置必须由管理程序来进行，因此不使用 `cpuspeed` 服务，而是使用自身特有的工具 `xenpm`。这种方式的优点是能够立即改变 P 状态。在 `cpufreq=dom0-kernel` 情形下，到 Dom0 的虚拟 CPU 被调度为止都不能改变 P 状态，但对于 `cpufreq=xen` 情形（如图 6-7 所示），是通过管理程序改变 P 状态的，因此必要时能够立即改变 P 状态。

```
title Xen
root (hd0,0)
kernel /boot/xen.gz cpufreq=xen
module /boot/vmlinuz-2.6.18.8-xen ro root=/dev/sda1
module /boot/initrd-2.6-xen.img
```

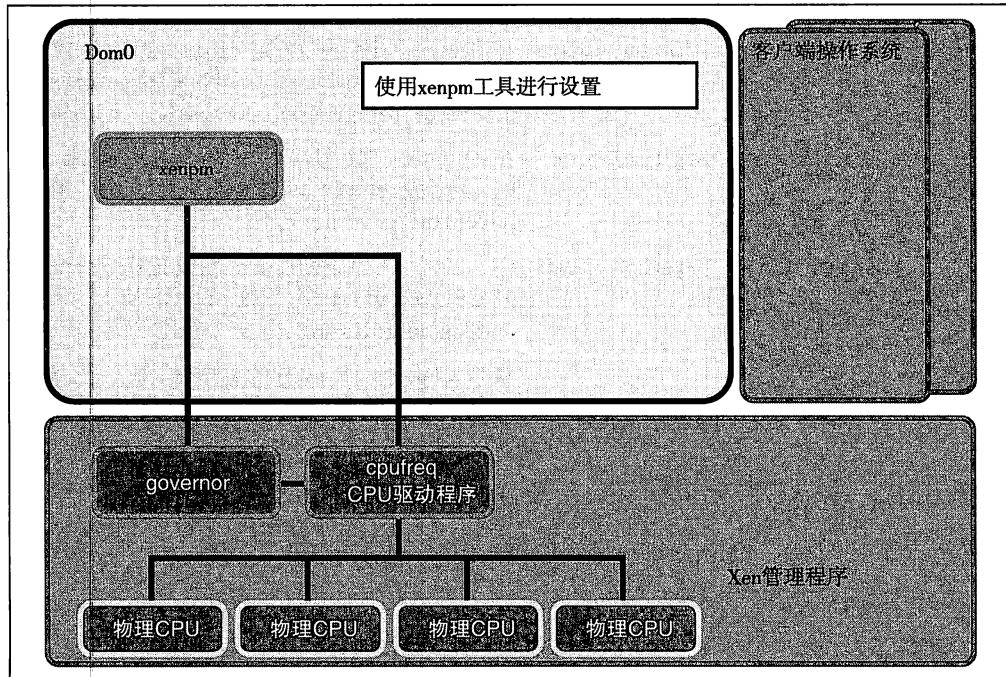


图 6-7 `cpufreq=xen` 时的虚拟 CPU 和物理 CPU 的关系图

表 6-18 是 `xenpm` 的使用方法。设置为 `xenpm start`，就可以像 `powertop` 实用程序（utility）一样观测 CPU 的 C/P 状态使用情况。

表 6-18 `xenpm` 的使用方法

子命令	变 量	说 明
<code>get-cpuidle-states</code>	<code>[cpuid]</code>	显示指定 cpuid 的 CPU 或者所有的 cpuidle 信息（C 状态信息）
<code>get-cpufreq-states</code>	<code>[cpuid]</code>	显示指定 cpuid 的 CPU 或者所有的 cpufreq 信息（P 状态信息）

子命令	变 量	说 明
get-cpufreq-para	[cpuid]	显示指定 cpuid 的 CPU 或者所有的 cpufreq 设置信息
set-scaling-maxfreq	[cpuid] <Hz>	在 <Hz> 中设置指定 cpuid 的 CPU 或者所有 CPU 的最大频率
set-scaling-minfreq	[cpuid] <Hz>	在 <Hz> 中设置指定 cpuid 的 CPU 或者所有 CPU 的最小频率
set-scaling-speed	[cpuid] <num>	在 <num> 中设置指定 cpuid 的 CPU 或者所有 CPU 的 scaling speed。用于 user space governor 的情形
set-scaling-governor	[cpuid] <gov>	在 <gov> 中设置指定 cpuid 的 CPU 或者所有 CPU 的 governor。可设置的 governor 有 userspace、performance、powersave、ondemand
set-sampling-rate	[cpuid] <num>	在 <num> (微秒) 中设置指定 cpuid 的 CPU 或所有 CPU 的采样间隔。用于 ondemand governor
set-up-threshold	[cpuid] <num>	在 <num> (CPU 使用率) 中设置指定 cpuid 的 CPU 或者所有 CPU 的阈值。用于 ondemand governor
start		获取 C 状态、P 状态的统计信息，在按下 CTRL+C 或收到 SIGINT 信号后输出结果

## Xen 的 C 状态

管理程序中安装有 Xen 的 cpuidle 子系统。要启用 Xen 的 cpuidle 子系统，需要按下列方法在管理程序的启动选项中添加 cpuidle。

```
title Xen
root (hd0,0)
kernel /boot/xen.gz cpuidle
module /boot/vmlinuz-2.6.18.8-xen ro root=/dev/sda1
module /boot/initrd-2.6-xen.img
```

在多个 CPU 的系统中，将 `sched_smt_power_savings` 用做管理程序的启动选项就可以更加省电。

启用 cpuidle 时，如果系统运行出现异常，可以尝试在管理程序的启动选项中添加 `max_cstate=2` 和 `lapic_timer_c2_ok`，限制使用的 C 状态。有时，在某些系统中也可以通过使用 `consistent_tscs` 启动选项来解决。

```
title Xen
root (hd0,0)
kernel /boot/xen.gz cpuidle max_cstate=2 lapic_timer_c2_ok
```

```
module /boot/vmlinuz-2.6.18.8-xen ro root=/dev/sda1
module /boot/initrd-2.6-xen.img
```

## KVM 的 C/P 状态

KVM 上的客户端操作系统从主机操作系统来看，只能看到 qemu 这个进程。在 KVM 中 Linux 内核自身就是管理程序。因此，KVM 中无须特别安装就可以直接使用 Linux 的省电功能。这是 KVM 的优点之一，使 Linux 内核中安装的大多数功能不需改造就可以作为管理程序的功能使用。也就是说，要在 KVM 上启用省电功能，只需像一般的 Linux 一样启用 cpuspeed 服务等（见图 6-8）。

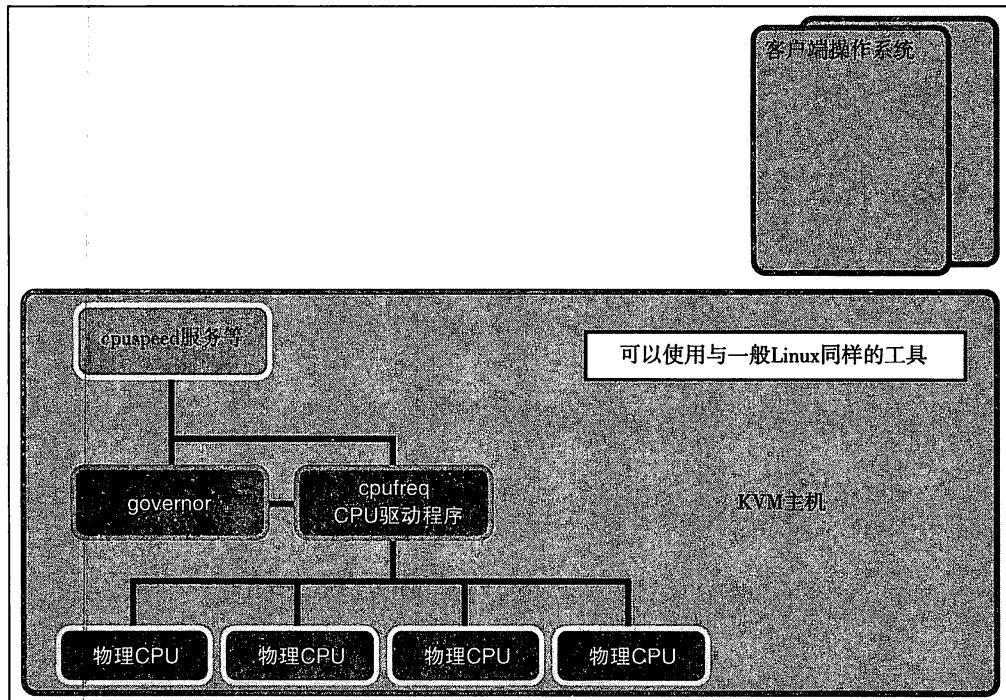


图 6-8 KVM 的省电关系图

## 小结

本节介绍了虚拟环境下的省电功能。在数据中心等地方，存在多个管理程序的虚拟环境下，有时使用动态迁移（live migration）改变客户端操作系统运行的系统配置，就可以关闭系统本身。例如，在云（cloud）等环境下就考虑了注重省电的客户端操作系统配备逻辑。具体来说，有白天和黑夜将负载峰值不同的客户端操作系统在一个系统中运行的方法，计算管理程序资源的空闲容量将客户端操作系统动态移动到其他系统上的方法等。

在虚拟环境下，对各个客户端操作系统的应答性能要求等是不同的，因此需要设计出满足各客户端操作系统的性能条件等的省电方法。

## 参考文献

- xenpm (<http://wiki.xensource.com/xenwiki/xenpm>)
- OLS2007 How virtualization makes power management different (Kevin Tian, Ke Yu, Jun Nakajima, and Winston Wang)

——Akio Takebe

## HACK #45 远程管理机器的电源

本节介绍使用 WOL 和 IPMI 远程管理机器电源的方法。

从远程接通或断开电源的方法有使用 NIC (Network Interface Card) 功能的 WOL 和 IPMI (BMC) 的方法。本节将介绍这些方法。

### Wake On LAN

Wake On LAN (WOL) 是指经由网络发送称为魔术包 (magic packet) 的特殊数据包，从网络设备接通系统电源的功能。

#### 设置 WOL

在使用 WOL 接入电源的机器上需要事先进行设置。首先，要在 BIOS 的设置中启用 WOL (也称为 Remote Power) 的功能。

本节中使用的 DELL 机器是在 BIOS 设置界面上选择 “Power Management” → “Remote Wake Up”，设置为 On。

然后，在操作系统启动后，使用 ethtool 命令进行网络设备的设置。下面是英特尔 82571EB 千兆位以太网控制器的例子。

```
# ethtool eth4
Settings for eth4:
.....
Transceiver: internal
Auto-negotiation: on
Supports Wake-on: pumbag          ...❶
Wake-on: d                         ...❷
Current message level: 0x00000001
Link detected: yes
#
```

❶的 Supports Wake-On 显示的是这个网络设备所支持的魔术包的种类。每个拉丁字母为显示种类。详细内容可以使用 man 命令来确认。

```
p  Wake on phy activity
u  Wake on unicast messages
m  Wake on multicast messages
b  Wake on broadcast messages
```

```
a Wake on ARP
g Wake on MagicPacket(tm)
s Enable SecureOn(tm) password for MagicPacket(tm)
d Disable (wake on nothing).
```

②的 Wake-on：显示的是 WOL 的设置情况。在这个例子中为 d，因此是关闭的。如果要启用 WOL，可以使用 ethtool 的 -s 选项来设置。下面是启用 MagicPacket 的例子。

```
# ethtool -s eth4 wol g
# ethtool eth4 | grep Wake-on
    Supports Wake-on: pumbag
    Wake-on: g
```

启用多个时需要执行下列命令。

```
# ethtool -s eth4 wol pumbag
# ethtool eth4 | grep Wake-on
    Supports Wake-on: pumbag
    Wake-on: pumbag
```

这样就进行了 WOL 的设置，即使切断一次电源也可以从远程重新接通电源。使用 ethtool 命令启用了 WOL 的 NIC，在切断电源后，如果连接上 LAN 电缆，LED 也会闪光。

## 发送魔术包

在远程机器上发送称为魔术包的数据包作为接通电源的命令。

本 Hack 中使用的是 net-tools 工具包中的 ether-wake 命令。按照下列方式指定 MAC 地址，执行 ether-wake 命令，就可以发送魔术包。

```
# /sbin/ether-wake 00:10:60:60:60:01
```

## 设置密码

为保证安全，也可以设置密码。使用 ethtool 启用 SecureOn。

```
# ethtool -s eth4 wol s
```

接下来设置密码。密码与 MAC 地址相似，设置以“:”分隔的 4~6 字节的字符串。下面是设置 6 字节密码的例子。

```
# ethtool -s eth4 sopass xx:yy:zz:aa:bb:cc
```

在远程方按照下列方式指定密码，执行 ether-wake 命令。

```
# ether-wake -p xx:yy:zz:aa:bb:cc 00:10:60:60:60:01
```

## IPMI

IPMI 中有 LAN 接口。可以使用这个接口通过网络从远程进行电源管理。

由于 WOL 在接收地址中指定 MAC 地址，因此在有些网络结构下魔术包无法到达。而 IPMI 可以将 IP 地址指定为接收地址进行电源管理，因此 IPMI 比 WOL 更为通用。

## 设置 IPMI

事先安装关于 IPMI 的数据包。

```
# yum install OpenIPMI.x86_64
```

由于使用 ipmitool 命令，因此需要安装 OpenIPMI-tools。

```
# yum install OpenIPMI-tools.x86_64
```

Fedora 12 等中的工具包名称有所不同，称为 ipmitool。安装 ipmitool。

```
# yum install ipmitool.x86_64
```

由于要将 IPMI 模块安装到内核中，因此需要启动 ipmi。

```
# /etc/init.d/ipmi start
```

---

小贴士：如果 ipmi 启动失败，可能是该机器中未配置 IPMI。

---

## 设置 IPMI LAN 接口

IPMI 中有多个接口。这些接口称为信道 (channel)。不同机器的信道编号也不相同，因此首先需要确认哪个信道编号相当于 LAN 接口。按照下列方式一个一个地指定信道编号，执行 ipmitool。

```
# ipmitool channel info 0
Channel 0x0 info:
  Channel Medium Type    : IPMB (I2C)
  Channel Protocol Type : IPMB-1.0
  Session Support       : session-less
  Active Session Count  : 0
  Protocol Vendor ID   : 7154
# ipmitool channel info 1
Channel 0x1 info:
  Channel Medium Type    : 802.3 LAN  ...①
  Channel Protocol Type : IPMB-1.0
  Session Support       : multi-session
  Active Session Count  : 0
  Protocol Vendor ID   : 7154
  Volatile(active) Settings
    Alerting           : disabled
    Per-message Auth   : disabled
    User Level Auth    : enabled
    Access Mode        : disabled  ...②
  Non-Volatile Settings
    Alerting           : disabled
    Per-message Auth   : disabled
    User Level Auth    : enabled
    Access Mode        : disabled
```

这个机器的信道编号 1 为 LAN 接口。通过①就可以确认其为 LAN 接口。

当②为 disabled 时，使用下列命令启用。

```

# ipmitool lan set 1 access on
# ipmitool channel info 1
Channel 0x1 info:
  Channel Medium Type    : 802.3 LAN
.....
  Volatile(active) Settings
.....
  Access Mode          : always available
  Non-Volatile Settings
.....
  Access Mode          : always available

```

这时就可以确认 LAN 接口的设置内容。

```

# ipmitool lan print
Set in Progress      : Set Complete
Auth Type Support   : NONE MD2 MD5 PASSWORD
Auth Type Enable    : Callback : MD2 MD5
                      : User      : MD2 MD5
                      : Operator   : MD2 MD5
                      : Admin      : MD2 MD5
                      : OEM        : MD2 MD5
IP Address Source   : Static Address
IP Address          : 0.0.0.0
Subnet Mask         : 0.0.0.0
MAC Address         : 00:19:19:19:19:1a
SNMP Community String : public
IP Header           : TTL=0x40 Flags=0x40 Precedence=0x00 TOS=0x10
Default Gateway IP  : 0.0.0.0
Default Gateway MAC : 00:00:00:00:00:00
Backup Gateway IP   : 0.0.0.0
Backup Gateway MAC  : 00:00:00:00:00:00
802.1q VLAN ID    : Disabled
802.1q VLAN Priority : 0
Cipher Suite Priv Max : Not Available

```

由于没有设置 IP 地址，因此使用 ipmitool 设置。

有的服务器在一般的 LAN 设备之外准备了 IPMI 专用的管理端口，有的服务器是共享 LAN 设备。在共享的情况下，IPMI 的 LAN 接口在内部与一般 LAN 设备相连接。在这种情况下设置 IPMI 的 LAN 接口的 IP 地址时，要使段（segment）与一般 LAN 设备相同。

```

# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:19:19:19:19:18
          inet addr:192.168.0.220  Bcast:192.168.0.255  Mask:255.255.255.0
          .....

# ipmitool lan set 1 ipaddr 192.168.0.221
Setting LAN IP Address to 192.168.0.221
# ipmitool lan set 1 netmask 255.255.255.0
Setting LAN Subnet Mask to 255.255.255.0
# ipmitool lan print
.....
IP Address Source      : Static Address

```

```
IP Address : 192.168.0.221
Subnet Mask : 255.255.255.0
MAC Address : 00:19:b9:f7:8f:1a
....
```

还可以通过 DHCP 获取 IP 地址。

```
# ipmitool lan set 1 ipsrc dhcp
# ipmitool lan print
.....
IP Address Source : DHCP Address
IP Address : 10.2.0.134
Subnet Mask : 255.255.255.0
MAC Address : 00:17:a4:3f:18:82
.....
```

从远程机器执行下列命令，就可以确认电源状态。

```
# ipmitool -I lan -H 192.168.0.221 -P " " chassis power status
Chassis Power is on
```

在 -H 选项中指定 IP 地址或主机名称。-P 为密码。默认为 NULL。当然也可以设置为任意短语 (phase)。命令的最后指定了确认电源状态的 status，除此以外，还有其他命令，如表 6-19 所示。

表 6-19 chassis power 命令列表

命令选项	说 明
status	显示电源的状态
on	接通电源
off	断开电源
cycle	重新启动。至少等待 1 秒钟直到断开电源为止。电源断开后，重新接入电源
reset	执行硬启动 (hard reset)。瞬时重启
diag	使 CPU 发生 NMI
soft	通过 ACPI 执行操作系统的软关闭 (soft shutdown)。等同于按下电源按钮的情况，因此由操作系统执行一般的关闭处理 (clean shutdown)

像下面这样指定为 on，就可以从远程接通电源。

```
# ipmitool -I lan -H 192.168.0.221 -P " " chassis power on
```

## 小结

本节介绍了从远程对机器的电源进行操作的功能。WOL 将 MAC 地址作为接收地址接通电源。只要 NIC 能够对应，无论笔记本电脑还是台式机都可以使用。但是由于位于网络中的以太网交换机的种类或设置，魔术包有时无法到达。

如果使用 IPMI，就可以将 IP 地址作为接收地址接入电源。还可以确认电源的状态或切断电源。但是必须安装有 IPMI (BMC)，因此仅限定于服务器等的机器。

通过这些功能就可以控制机器的电源，抑制电能消耗。

使用 IPMI 还可以控制 NMI 的产生和硬重启，因此也可以用于开发 Linux 内核时的调试等。

## 参考文献

- Intelligent Platform Management Interface Specification  
<http://www.intel.com/design/servers/ipmi/>

——Naohiro Ooiwa

## HACK #46 USB 的电力管理

Linux 中可以对经由 sysfs 的 USB 设备进行电力管理，本节介绍这种管理电力的方法。

### 概要

从 Linux 内核 2.6.21 开始安装了 USB 设备的自动待机 (auto suspend) 功能。自动待机是指 USB 设备在一定时间内一直保持空闲状态时，内核自动使 USB 设备待机的功能。要使用 USB 设备时自动重启 (auto resume)。

也可以手动进行待机。不需要将 USB 线拔出后再插上就可以重新开始 (resume) 供电。自己编译内核时，需要将内核的 config 设置为 CONFIG\_PM\_RUNTIME=y、CONFIG\_USB\_SUSPEND=y。make menuconfig 有下列内容。

```
Power management and ACPI options  --->
  Run-time PM core functionality

Device Drivers  --->
  USB support  --->
    USB runtime power management (suspend/resume and wakeup)
```

### 设置方法

在 sysfs 的文件 /sys/bus/usb/devices/<USB 设备 ID>/power/ 中进行设置。下面介绍其中的一些设置选项。

#### /sys/bus/usb/devices/<USB 设备 ID>/power/autosuspend

这个文件设置的是各个 USB 设备切换到自动待机为止的时间 (秒)。标准设置值为 2 秒。USB 设备处于空闲状态达到 2 秒以上时，内核就会自动使该设备待机。设置为 0 时，USB 设备一旦空闲就尽快待机。设置为 -1 时，关闭自动待机。

值的更改通过下列命令来进行。

```
# echo 5 > /sys/bus/usb/devices/<USB 设备 ID>/power/autosuspend
```

---

小贴士：标准设置的 2 秒这个值是在 /sys/module/usbcore/parameters/autosuspend 中设置的。如果将 /sys/module/usbcore/parameters/autosuspend 改为 5，则新连接的 USB 设备的 /sys/bus/usb/devices/<USB 设备 ID>/power/autosuspend 的标准设置值也为 5 秒。/sys/module/usbcore/parameters/autosuspend 还可以通过内核启动参数 usbcore.autosuspend=<值> 来更改。

---

USB 设备 ID 是指内核对 USB 设备添加的编号。下面是台式机的例子。编号为 1 ~ 7。

```
$ ls /sys/bus/usb/devices/
1-0:1.0 3-0:1.0 5-0:1.0 6-2      7-0:1.0 7-1:1.0  usb2  usb4  usb6
2-0:1.0 4-0:1.0 6-0:1.0 6-2:1.0 7-1      usb1      usb3  usb5  usb7
```

连接 USB 存储器后，就会输出如下所示的内核信息。

```
# dmesg
...
usb 1-3: new high speed USB device using ehci_hcd and address 21 检测到新的连接
usb 1-3: configuration #1 chosen from 1 choice
scsi25 : SCSI emulation for USB Mass Storage devices
usb-storage: device found at 21
usb-storage: waiting for device to settle before scanning
usb-storage: device scan complete
scsi 25:0:0:0: Direct-Access      BUFFALO  USB Flash Disk  3.10 PQ: 0
ANSI: 0 CCS
sd 25:0:0:0: [sdb] 2015232 512-byte hardware sectors (1032 MB)
sd 25:0:0:0: [sdb] Write Protect is off
sd 25:0:0:0: [sdb] Mode Sense: 23 00 00 00
sd 25:0:0:0: [sdb] Assuming drive cache: write through
sd 25:0:0:0: [sdb] 2015232 512-byte hardware sectors (1032 MB)
sd 25:0:0:0: [sdb] Write Protect is off
sd 25:0:0:0: [sdb] Mode Sense: 23 00 00 00
sd 25:0:0:0: [sdb] Assuming drive cache: write through
  sdb: unknown partition table
sd 25:0:0:0: [sdb] Attached SCSI removable disk
sd 25:0:0:0: Attached scsi generic sg2 type 0
```

这时的 USB 设备 ID 为 1 ~ 3。下面是连接了 CD-ROM 驱动器时输出的内核信息的例子。USB 设备 ID 为 1 ~ 4。

```
# dmesg
...
usb 1-4: new high speed USB device using ehci_hcd and address 22
usb 1-4: configuration #1 chosen from 1 choice
scsi26 : SCSI emulation for USB Mass Storage devices
usb-storage: device found at 22
usb-storage: waiting for device to settle before scanning
usb-storage: device scan complete
scsi 26:0:0:0: CD-ROM          GENERIC  DVD-ROM B763B  1.00 PQ: 0
ANSI: 0
sr1: scsi3-mmc drive: 24x/24x cd/rw xa/form2 cdda tray
sr 26:0:0:0: Attached scsi CD-ROM sr1
sr 26:0:0:0: Attached scsi generic sg3 type 5
```

这样连接 USB 设备后，输出内核信息，目录下就会变成下列内容。

```
# ls
1-0:1.0 1-4      3-0:1.0 6-0:1.0 7-0:1.0  usb1  usb4  usb7
1-3      1-4:1.0  4-0:1.0 6-2      7-2      usb2  usb5
1-3:1.0  2-0:1.0 5-0:1.0 6-2:1.0 7-2:1.0  usb3  usb6
```

此外，使用 `/sys/bus/usb/devices/<USB 设备 ID>/product`、`lsusb` 命令等也可以确认所连接的 USB 设备。

```
# cat /sys/bus/usb/devices/5-2/product
USB Optical Mouse

$ lsusb
...
Bus 005 Device 002: ID 0461:4d15 Primax Electronics, Ltd Dell Optical Mouse
...
```

在 `/sys/bus/usb/devices/<USB 设备 ID>/power/level` 中设置 `auto` 就可以启用自动待机功能。默认设置为 `on`，因此需要使用下列命令改为 `auto`。

```
# echo auto > /sys/bus/usb/devices/<USB 设备 ID>/power/level
```

这样就启用了自动待机功能。但是这个功能比较难以确认。即使 USB 设备通过自动待机进入待机状态，从外表上也是看不出变化的。

其中 USB 激光鼠标是比较容易看出效果的一种设备。鼠标经过一定秒数后，背面 LED 的红光就会消失。这就是已待机的状态。这时鼠标也不会自动移动。单击鼠标的按钮才会自动恢复。LED 也发出红光，鼠标就进入可使用状态。

键盘在经过一定秒数后也会自动待机，但从键盘的外观就看不出变化。按键后才会自动恢复。而在按下 Numlock 键、Shift+CapsLock 键、ScreenLock 键之一的状态下不会进入自动待机模式。

在 USB 存储器、CD-ROM 设备、软盘驱动器等其他设备上也进行了尝试，但有一些设备内部事件发生频繁，不会进入自动待机模式。

想要确认是否进入自动待机模式，可以启用内核编译选项 `Device Driver → USB support → USB verbose debug message`，重新编译内核，就可以通过内核信息确认自动待机和自动恢复。

### `/sys/bus/usb/devices/<USB 设备 ID>/power/level`

在这个文件中设置 `on`、`auto`。默认为 `on`。到内核 2.6.32 为止还可以设置 `suspend`。

如前所述，如果设置为 `auto`，就可以启用 USB 设备的自动待机和自动恢复。

按下列方式设置，则对 USB 设备待机。

```
# echo suspend > /sys/bus/usb/devices/<USB 设备 ID>/power/level
```

设备的 LED 熄灭，在操作系统上也识别不到设备。

指定为 on 时，就会向待机状态下的 USB 设备供电。与重新接上 USB 线时的运行情况相同。

```
# echo on > /sys/bus/usb/devices/<USB 设备 ID>/power/level  
/sys/bus/usb/devices/<USB 设备 ID>/power/connected_duration  
/sys/bus/usb/devices/<USB 设备 ID>/power/active_duration
```

从内核 2.6.25 开始安装了 connected\_duration 和 active\_duration。connected\_duration 为 USB 设备处于连接状态的总时间。单位为毫秒。active\_duration 为 USB 设备运行的总时间，也就是 USB 设备未待机的时间。单位也为毫秒。可以通过这些值来确认自动待机的效果或待机的时间。

下面是刻意待机约 10 秒的例子。active\_duration 的值比 connect\_duration 小。

```
# cd /sys/bus/usb/devices/2-1/power/  
# head *duration  
==> active_duration <==  
23100994  
  
==> connected_duration <==  
23100994  
# echo suspend > level ; sleep 10 ; echo on > level  
# head *duration  
==> active_duration <==  
23124290          比 connected_duration 还要少约 10 000 毫秒  
  
==> connected_duration <==  
23134371
```

## 小结

本节介绍了如何管理 USB 设备电力的自动待机功能。如果 USB 设备持续处于空闲状态，Linux 内核就会自动将使 USB 设备待机。一旦使用到 USB 设备，就会自动恢复。

## 参考文献

- Document/usb/power-management.txt

——Naohiro Ooiwa

# HACK #47 显示器的省电

本节介绍关于显示器的省电方法。

## 显示器的电源控制

关于显示器的硬件规格有 Video Electronics Standards Association (VESA) 标准中的 DPMS (Display Power Management Signaling)。DPMS 有 on、standby、suspend、off 模

式, 对于视频控制器和显示器也有规定。ACPI 的规格中配合 DPMS 规格定义了 D 状态。表 6-20 和表 6-21 分别是 LCD 和视频控制器的规格, 以供参考。

表 6-20 LCD 设备的规格

状 态	DPMS 状态	说 明
D0	on	显示器完全 on 的状态。亮度完全 on 的状态。画面为显示状态
D1	standby	显示器完全 off。画面为无显示状态。到恢复 D0 状态的延迟时间为 500 毫秒 <sup>注3</sup> 以下。这个状态有可能未安装。与 D3 状态不同的是, 一旦接收到 resume 请求, 驱动程序就会将显示器恢复到原来的状态。显示器在内部保存状态, 有可能抑制耗电量
D2	suspend	显示器完全 off。画面为无显示状态。到恢复 D0 状态为止的延迟时间为 500 毫秒 <sup>注4</sup> 以下。这个状态有可能未安装。与 D3 状态不同的是, 一旦接收到 resume 请求, 驱动程序就会将显示器恢复到原来的状态。显示器保存状态, 可以抑制耗电量
D3	off	显示器不工作的状态。显示器为完全 off 状态。画面为无显示状态。通知已无电源。到恢复 D0 状态的延迟时间为 500 毫秒 <sup>注5</sup> 以下

表 6-21 视频控制器的规格

状 态	DPMS 状态	说 明
D0	on	连接到控制器的设备为 on 状态。视频控制器保存内容。保存视频存储器的内容
D1	standby	连接到控制器的设备 (除显示器 /LCD 控制信号以外) 为 off 状态。视频控制器保存内容。保存视频存储器的内容。到恢复 D0 状态为止的延迟时间为 100 毫秒以下
D2	suspend	连接到控制器的设备 (除显示器 /LCD 控制信号以外) 为 off 状态。视频控制器不保存内容。视频存储器的内容不保存。到恢复 D0 状态为止的延迟时间为 200 毫秒以下
D3	off	连接到控制器的设备为 off 状态。视频控制器不保存内容 (也停止供电)。视频存储器的内容不保存 (也停止供电)。到恢复 D0 状态为止的延迟时间为 200 毫秒以下

在命令行进行这些 DPMS 设置时可以使用 xset 命令等, 表 6-22 是使用 xset 命令的例子。

注 3: 新设备中使用的是 100 毫秒, DPMS 中规定必须在 5 秒以下。

注 4: 新设备中使用的是 100 毫秒, DPMS 中规定为 10 秒以下。

注 5: 新设备中使用的是 100 毫秒, DPMS 中未规定到恢复为止的延迟时间。

表 6-22 使用 xset 命令的例子

命 令	说 明
xset -dpms	将 DPMS 设置为 off
xset +dpms	将 DPMS 设置为 on
xset dpms force on	强制设置 DPMS 的 on 模式
xset dpms force standby	强制设置 DPMS 的 standby 模式
xset dpms force suspend	强制设置 DPMS 的 suspend 模式
xset dpms force off	强制设置 DPMS 的 off 模式

## 显示器的亮度控制

ACPI 中还有控制亮度的规格，如果 DSDT 中有控制亮度的定义块，就可以进行亮度控制。亮度控制所需的主要是在 ACPI 的定义块中定义的 \_BCL、\_BCM、\_BQC 方法。要确认 ACPI 的定义块请参考 Hack #40。

还准备了从键盘的功能键等通知更改显示器亮度的结构。

### \_BCL (Query List of Brightness Control Levels Supported)

通过这个方法，操作系统可获取内置显示器设备所支持的亮度水平的列表。另外还显示 LCD 是否支持更改亮度水平。将亮度水平分别用 0~100 的数字来表示百分比。

参数	无
返回值	包括所支持的亮度水平的整数列表的 Package
<pre>Method (_BCL, 0) {     // List of supported brightness levels     Return (Package(7) {         80, // level when machine has full power         50, // level when machine is on batteries         // other supported levels:         20, 40, 60, 80, 100     }) }</pre>	

### \_BCM (Set the Brightness Level)

通过这个方法，操作系统设置内置显示器设备的亮度。

操作系统只能设置通过 BCL 获取的值。

参数	要设置的亮度水平的整数
返回值	无

## \_BQC (Brightness Query Current level)

这个方法返回当前的亮度水平。

参数	无
返回值	当前的亮度水平的整数

除 ACPI 以外，也有厂商固有的视频控制，但 Linux 中有控制背光灯的子系统，可以经由 /sys 来进行。

表 6-23 /sys/devices/virtual/backlight/acpi\_videoN/ 的接口

接口名称	功能
actual_brightness	当前的亮度水平
brightness	读入则显示 user 最后设置的亮度水平。写入则改变亮度水平。把亮度水平的更改通知到背光灯
max_brightness	最大亮度

现在已经出现了一些用于显示器省电控制的便捷的应用程序。Fedora 13 等中可以使用名为 gnome-preference 的应用程序来设置显示器的亮度等（见图 6-9）。

使用 xbacklight 命令还可以调节显示器的亮度。

根据需要可以安装 xbacklight 工具包。

```
# yum install xbacklight
```

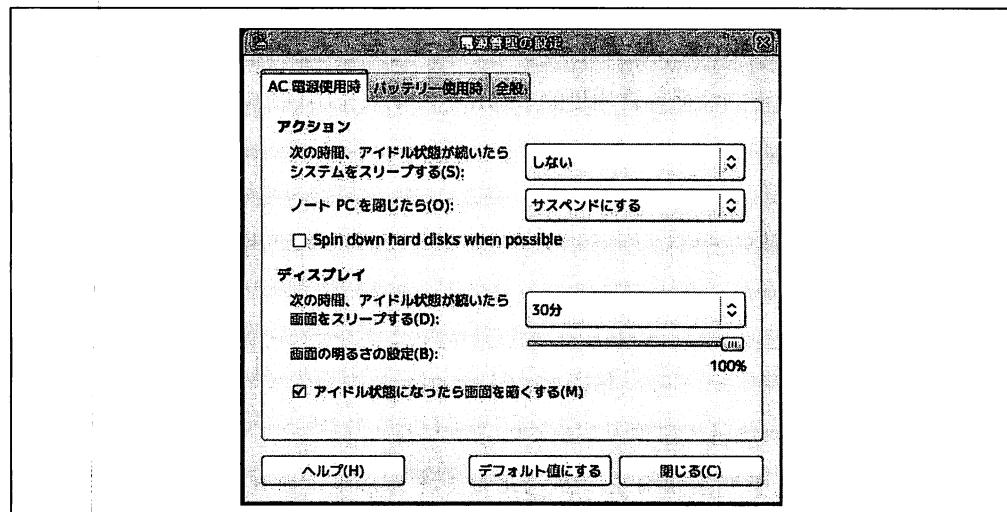


图 6-9 | gnome-power-preference 的设置界面

表 6-24 xbacklight 命令的使用方法

命 令	说 明
xbacklight -get	以百分比形式显示当前亮度
xbacklight -set <百分比>	设置为指定的亮度
xbacklight -inc <百分比>	在当前亮度上增加指定的百分比
xbacklight -dec <百分比>	在当前亮度上减少指定的百分比
xbacklight -time <毫秒>	到变为新亮度为止的时间
xbacklight -steps <数值>	变为新亮度时的单步 (step) 次数

下面是使用 xbacklight 命令的例子。

```
# xbacklight -steps 10 -time 1000 -dec 50
```

## 小结

本节介绍了显示器的控制方法。除此以外，还有如下所示的显示器省电方法。

- 为了抑制图形设备的负载，将桌面的壁纸换成单一颜色。
- 在 LCD 的显示器上使用有抑制背光效果的颜色。

**小贴士：** 液晶显示器有 normally white (白色背光) 型和 normally black (黑色背光) 型，一般 normally white 使用白色比较省电，normally black 使用黑色比较省电。视角较狭窄或 20 型以下的液晶显示器多为 normally white 型。

- 干脆不使用显示器（用于服务器等）

方法是多种多样的，可以根据各自的目的来选择适合的省电方法。

## 参考文献

- Advanced Configuration and Power Interface Specification Revision 4.0

——Akio Takebe

### Black Google

有些网站为了省电，想用黑色作为显示器的显示颜色。

在 CRT (阴极射线管) 显示器上，黑色比白色更为省电，因此就出现了将白底的 Google 检索画面改成黑色的 Black Google。简单检索一下，就找到了下列网页。

<http://black-google.blogspot.com/>  
<http://www.google.com/intl/en/>

<http://www.black1.com/black-google.html>  
<http://www.blackgoogle.eu/>  
<http://black-google.net/>  
<http://www.blackle.com/>

这些网页都把 Google 检索画面改成了黑底（均为写作过程中存在的 URL。现在有些已经无法连接）。

在 CRT 上黑色是不耗电的，因此能产生效果。但是最近 LCD（液晶显示器）已经普及。相反，LCD 是白色不耗电，但也有一些新的 LCD 抑制黑色显示效果的耗电量。

在 SHARP 的 AQUOS（42 寸，关闭省电功能）上进行了测定。测定的情况如图 6-10 所示。

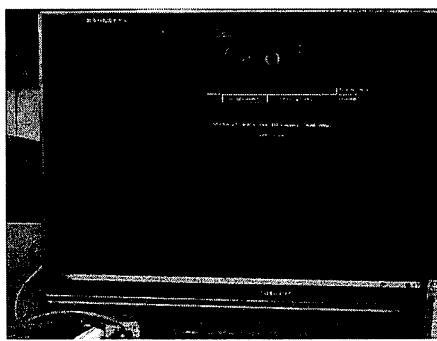


图 6-10 测定的情形

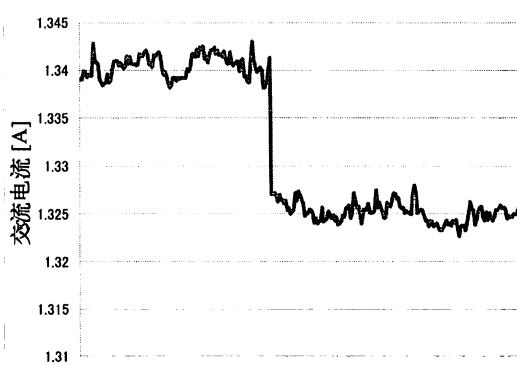


图 6-11 gnome-power-preference 的设置界面

将笔记本电脑连接到 AQUOS，测定 Web 浏览器显示一般的 Google 画面和设置为 Black Google 时的电流值。图 6-11 是反映电流变化的图表。中间电流急剧下降的地方正好从白色 Google 画面切换到 Black Google 的时刻。表 6-25 所示为大致的平均值。通过本次测定确认了 Black Google 更为省电。

表 6-25 Google 界面与大致的交流电流值

界面	交流电流值
一般的白色 Google	1.34A
Black Google	1.325A

## HACK #48 通过网络设备节省电能

本节介绍通过控制网络设备节省电能的方法。

本节将介绍 Intel Less Watts 项目提出的省电方法和笔者实际向论坛发布的 Hack 方案。

### 禁用 WOL

如果启用 Hack #45 中介绍的 WOL (Wake On LAN)，那么机器的电源关闭后也会消耗极少的电能。

如果不使用 WOL，可以禁用它。

```
# ethtool -s eth0 wol d
```

### 降低速度

最近，千兆位以太网成为网络设备的主流。但是有的时间段内，较低的通信速度可能就足够了。如果某种程度上可以预测到这个时间段，就可以通过降低通信速度来抑制电能消耗。

网络设备的速度可以使用 ethtool 命令来更改。使用下列命令就可以设置为 100Mbps。其自动协商 (auto negotiation) 功能也禁用。

```
# ethtool -s eth0 autoneg off speed 100
```

要回到 1000Mbps，可以执行下列命令。

```
# ethtool -s eth0 autoneg on speed 1000
```

使用 ethtool 控制通信速度就可以节省电能。

在降低性能节省电能这个想法上与 CPU 的 C 状态、P 状态是相同的。

### 进行改造

前面介绍的是 LessWatts 项目。下面介绍从另一个角度改善网络设备的例子。使用 ifdown 命令明确禁用网络设备就可以省电。

下面是安装在 e1000 设备驱动程序中的补丁。

Signed-off-by: Naohiro Ooiwa <nooiwa@miraclelinux.com>

---

drivers/net/e1000/e1000\_main.c | 32 ++++++-----

1 files changed, 32 insertions(+), 0 deletions(-)

```
diff --git a/drivers/net/e1000/e1000_main.c b/drivers/net/e1000/e1000_main.c
index bcd192c..12e1a42 100644
--- a/drivers/net/e1000/e1000_main.c
+++ b/drivers/net/e1000/e1000_main.c
@@ -26,6 +26,11 @@

*****
+/*
+ * define this if you want pci save power while ifdown.
+ */
+#define E1000_PCI_POWER_SAVE
+
#include "e1000.h"
#include <net/ip6_checksum.h>

@@ -1248,6 +1253,23 @@ static int e1000_open(struct net_device *netdev)
        struct e1000_hw *hw = &adapter->hw;
        int err;

+#ifdef E1000_PCI_POWER_SAVE
+        struct pci_dev *pdev = adapter->pdev;
+
+        pci_set_power_state(pdev, PCI_D0);
+        pci_restore_state(pdev);
+
+        if (adapter->need_ioport)
+            err = pci_enable_device(pdev);
+        else
+            err = pci_enable_device_mem(pdev);
+        if (err) {
+            printk(KERN_ERR "e1000: Cannot enable PCI device from power-
save\n");
+            return err;
+        }
+        pci_set_master(pdev);
#endif
+
/* disallow open during test */
if (test_bit(__E1000_TESTING, &adapter->flags))
    return -EBUSY;
@@ -1265,6 +1287,7 @@ static int e1000_open(struct net_device *netdev)
        goto err_setup_rx;

        e1000_power_up_phy(adapter);
+
        e1000_reset(adapter);

        adapter->mng_vlan_id = E1000_MNG_VLAN_NONE;
        if ((hw->mng_cookie.status &
@@ -1341,6 +1364,15 @@ static int e1000_close(struct net_device *netdev)
        e1000_vlan_rx_kill_vid(netdev, adapter->mng_vlan_id);
    }
```

```

+">#ifdef E1000_PCI_POWER_SAVE
+,#ifdef CONFIG_PM
+    pci_save_state(adapter->pdev);
+,#endif
+    pci_disable_device(adapter->pdev);
+    pci_wake_from_d3(adapter->pdev, true);
+    pci_set_power_state(adapter->pdev, PCI_D3hot);
+,#endif
+
+    return 0;
}

-- 1.5.4.1

```

在 ifdown 命令处理 (e1000\_close 函数) 的最后, 调用 pci\_wake\_from\_d3(), 启用 PCI 的 Power Management 寄存器的 PME pin。使用 pci\_set\_power\_state() 将设备改为 D3 状态。执行 ifup 命令时回到 D0 状态。

最近像电视这样的数码家电产品中也安装了网络设备。但是电视实际上大部分时间都只是用来收看节目。在这种情况下, 如果可以使用应用程序 ifdown, 仅在连接网络的时候执行 ifup 命令, 就可以减少耗电量。

---

**注意事项:** 这个补丁正在开发中。还未能考虑支持 ethtool 命令。因此如果安装了这个补丁, 执行 ethtool 命令的一些选项, 就会死机。论坛称当 ethtool 也能够顺利运行时, 就会合并这些选项。

---

## 小结

本节探讨了网络设备省电的可能性。基本原则是“不使用时就彻底断电”。CPU 等在 Linux 中也已进行了设备省电的改善。但是设备驱动程序数量非常多, 还有一些不支持省电功能的。可以尝试对已有的设备驱动程序进行新的修改, 相信会很有意思的。

## 参考文献

- LessWatts.org Tips & Tricks Ethernet  
<http://www.lesswatts.org/tips/ethernet.php>
- IEEE 802.3 Energy Efficient Ethernet Study Group  
[http://www.ieee802.org/3/eee\\_study/index.html](http://www.ieee802.org/3/eee_study/index.html)
- Linux Foundation Collaboration Summit 2009 Green Linux Saving Large Amounts of Energy With Network Connectivity Proxying  
[http://events.linuxfoundation.org/archive/lfcs09\\_nordman.pdf](http://events.linuxfoundation.org/archive/lfcs09_nordman.pdf)
- Linux Foundation Collaboration Summit 2009 Green Linux Survey of Green IT in Linux and Tooling to Support Measuring Electrical Power Consumption  
[http://events.linuxfoundation.org/archive/lfcs09\\_ooiwa.pdf](http://events.linuxfoundation.org/archive/lfcs09_ooiwa.pdf)
- カーネルドキメント PCI Power Management

## HACK #49 关闭键盘的 LED 来省电

为了尽量减少电能消耗，关闭键盘 LED 灯等，对键盘进行控制。

小时候，父母经常教育我“房间里没人的时候要关灯”。突然看到键盘上还有 LED 灯闪着美丽的光芒却无人欣赏。要省电就必须关掉这些没有必要的 LED。使用无线键盘等时，为了抑制电能消耗，可能会想要关闭显示键盘 NumLock 是否启用的 LED 等。这种情况下禁用 NumLock 也可以，这里介绍的是启用 NumLock 的同时关闭 LED 的方法。

首先介绍怎样控制键盘，即 PS/2 键盘的规格。

### PS/2 键盘

我第一次接触电脑的时候，说起键盘，都是 PS/2 连接的。我家的电脑至今还有 PS/2 连接的键盘。AT 键盘和 PS/2 键盘是 IBM 发布的主流键盘。PS/2 键盘是作为能够与 AT 键盘互换的键盘发布的。PS/2 是 AT 的扩展，添加了一些命令。笔者家用的键盘是 PS/2 兼容的键盘，但是操作系统启动时显示的是 AT 互换键盘。也有不少键盘不支持 PS/2 的所有命令。IBM 原本只是将 Intel i8042 作为键盘输入的编码用控制器使用。

因此，PS/2 兼容键盘一般是以 PS/2 兼容模式连接到 Intel i8042 互换芯片组的。PS/2 连接的键盘，是通过 PC 主板上的 i8042 互换控制器（板载的微控制器）与键盘中的微控制器（键盘微控制器）进行通信来控制键盘。可以使用 in 命令、out 命令经由 0x60、0x64 的 I/O 端口从 CPU 访问控制器。

通过读出（in 命令）I/O 端口 0x64，就可以检查板载微控制器的状态。板载微控制器的 Status 寄存器各位的意义如表 6-26 所示。

表 6-26 板载微处理器的状态

位	说 明
0	OBF (Output Buffer Full) (1= 有数据, 0= 空)
1	IBF (Input Buffer Full) (1= 有数据, 0= 空)
2	系统标志 system flag (1= 启动完成, 0= 正在启动)
3	命令 / 数据 (0= 由 IO 端口 0x60 处理数据, 1= 由 IO 端口 0x64 处理命令)
4	Inhibit Switch (1= 启用键盘, 0= 禁用键盘)
5	Auxiliary Device Output Buffer Full (1= 传输错误, 0= 正常)
6	General Time-out (1= 键盘超时, 0= 正常)
7	Parity error (1= 奇偶错误, 0= 正常)

针对 I/O 端口 0x64 执行 Write (out 命令) 就可以向板载控制器发送命令。针对 I/O 端口 0x60 的 Write 命令被发送到键盘微控制器。

从键盘微控制器发送的数据, 从 I/O 端口 0x60 读出。用于读取一般键盘输入等的数据。

Status 的第 0 位和第 1 位用于信号交换 (握手)。在向 I/O 端口 0x60、0x64 进行写入 (发送命令) 前需要确认第 0 位是否为 0 (Output Buffer empty)。当第 1 位为 1 (Input Buffer full) 时, 可以从 I/O 端口 0x60 读取数据。细节参见表 6-27。

表 6-27 板载微控制器 (端口 0x64)

值	说 明
0x20	将板载微控制器的命令字节的结果传递给 I/O 端口 0x60
0x60	将写入 I/O 端口 0x60 的下一个内容发送到板载微控制器的命令字节
0xA4	测试是否设置了密码 (仅 PS/2)。把结果发送到 I/O 端口 0x60。0xFA 表示设置了密码, 0xF1 表示没有密码
0xA5	传递密码 (仅 PS/2)。依次导入写入 I/O 端口 0x60 的内容, 直到写入 NULL 文字为止。将所写的内容作为新密码
0xA6	启用密码
0xA7	禁用鼠标设备 (仅 PS/2)。与把命令字节的第 5 位设置为 1 表示相同的意义
0xA8	启用鼠标设备 (仅 PS/2)。与把命令字节的第 5 位清 0 表示相同的意义
0xAD	禁用键盘。将命令字节的第 4 位设置为 1
0xAE	启用键盘。将命令字节的第 4 位设置为 0
0xC0	将键盘的输入端口读取到 I/O 端口 0x60
0xC1	将输入端口 (上述) 第 0 ~ 3 位复制到 Status 的第 4 ~ 7 位 (仅 PS/2)
0xC2	将输入端口 (上述) 第 4 ~ 7 位复制到 Status 的第 4 ~ 7 位 (仅 PS/2)
0xD0	将板载微控制器的输出端口的值复制到 I/O 端口 0x60 (参考以下定义)
0xD1	将写入 I/O 端口 0x60 的下一个内容发送到板载微控制器的输出端口
0xD2	向键盘缓冲区写入。将写入 I/O 端口 0x60 的下一个值作为生成的键盘输入的值返回板载微控制器 (仅 PS/2)
0xD3	向鼠标缓冲区写入。将写入 I/O 端口 0x60 的下一个值作为生成的鼠标操作的值返回板载微控制器 (仅 PS/2)
0xD4	将写入下一个 I/O 端口 0x60 的数据输出到鼠标 (辅助) 设备 (仅 PS/2)
0xE0	读取测试输入, 向 I/O 端口 0x60 返回测试结果 第 0 位 =Keyboard clock input, 第 1 位 =Keyboard data input
0xFx	向输出端口输出脉冲

命令 0x20 和 0x60 对板载微控制器的命令字节进行读写。这个命令字节的各位含义如表 6-28 所示。

表 6-28 读写命令字节的各位含义

位	说    明
0	键盘中断 (1= 启用, 0= 禁用)
1	鼠标设备中断 (1= 启用, 0= 禁用)
2	写入 Status 的系统标志的值
3	预约
4	键盘 (1= 禁用键盘)
5	鼠标设备 (1= 禁用鼠标)
6	键盘转换 (1= 启用, 0= 禁用)
7	预约

向 I/O 端口 0x60 写入表示直接向键盘微控制器发送命令。向 I/O 端口 0x60 写入必须在确认 Status 的第 0 位为 0 后进行。向键盘微控制器发送的命令如表 6-29 所示。

表 6-29 键盘微控制器的命令 (端口 0x60)

值	说    明
0xED	操作 LED。按照写入 I/O 端口 0x60 的下一个内容更新键盘的 LED。写入 I/O 端口 0x60 的下一个参数的意义如下 第 3 ~ 7 位: 必须为 0 第 2 位: Capslock LED (1=on, 0=off) 第 1 位: Numlock LED (1=on, 0=off) 第 0 位: Scroll lock LED (1=on, 0=off)
0xEE	Echo 命令。向输入端口 0x60 返回 0xEE
0xF0	选择 scan code set (仅 PS/2)。写入 I/O 端口 0x60 的内容从下列选项中选择。 00: 读取当前的 scan code set (下一个值从 I/O 端口 0x60 读取) 01: 选择 scan code set #1 (standard PC/AT scan code set) 02: 选择 scan code set #2 03: 选择 scan code set #3
0xF2	要求键盘的 ID 信息
0xF3	设置自动重复的延迟和重复率。由写入 I/O 端口 0x60 的下一个内容决定重复率 第 7 位: 必须为 0 第 5、6 位: 延迟 (00=1/4sec, 01=1/2sec, 10=3/4sec, 11=1sec) 第 0 ~ 4 位: 重复率 (0= 约 30chars/sec -> 0x1F= 约 2chars/sec)
0xF4	启用键盘
0xF5	复位到 PowerON 的状态, 等待命令启用

值	说 明
0xF6	复位到 PowerON 的状态, 开始键盘的扫描
0xF7	将所有键设置为自动重复 (仅 PS/2)
0xF8	进行设置使所有键生成 up code 和 down code (仅 PS/2)
0xF9	进行设置使所有键只生成 up code (仅 PS/2)
0xFA	进行设置使所有键自动重复, 生成 up code 和 down code (仅 PS/2)
0xFB	将写入 I/O 端口 0x60 的下一个键设置为自动重复 (仅 PS/2)
0xFC	进行设置使写入 I/O 端口 0x60 的下一个键生成 up code 和 down code (仅 PS/2)
0xFD	进行设置使写入 I/O 端口 0x60 的下一个键只生成 down code (仅 PS/2)
0xFE	重新发送最后的结果。当存在错误的接收数据时, 使用这个命令
0xFF	将键盘复位到 PowerON 的状态, 开始自我诊断

通过上面的介绍, 你应该已经了解 LED 的控制方法了。下面介绍如何控制 LED 的代码作为参考。

```

1  #include<stdio.h>
2  #include<sys/io.h>
3  #define KBD_CMD_PORT 0x60
4  #define OBD_STS_PORT 0x64
5
6  #define OBS_FULL      0x1
7  #define SEND_LED      0xED
8
9  #define SCROLL_LOCK   (1<<0)
10 #define NUM_LOCK      (1<<1)
11 #define CAPS_LOCK     (1<<2)
12
13 void send_cmd(cmd, port)
14 {
15     char sts;
16     do{
17         sts = inb(OBD_STS_PORT);
18     } while(sts & OBS_FULL);
19     outb(cmd, port);
20     usleep(100000);
21 }
22
23 int main(void)
24 {
25     ioperm(KBD_CMD_PORT, 1, 1);
26     ioperm(OBD_STS_PORT, 1, 1);
27
28     /* Turn on a LED of CAPS_LOCK */
29     send_cmd(SEND_LED, KBD_CMD_PORT);
30     send_cmd(CAPS_LOCK, KBD_CMD_PORT);

```

```

31             /* Turn on a LED of NUM_LOCK */
32             send_cmd(SEND_LED, KBD_CMD_PORT);
33             send_cmd(NUM_LOCK, KBD_CMD_PORT);
34
35             /* Turn on a LED of SCROLL_LOCK */
36             send_cmd(SEND_LED, KBD_CMD_PORT);
37             send_cmd(SCROLL_LOCK, KBD_CMD_PORT);
38
39             /* Turn off LEDs */
40             send_cmd(SEND_LED, KBD_CMD_PORT);
41             send_cmd(0, KBD_CMD_PORT);
42
43             ioperm(KBD_CMD_PORT, 1, 0);
44             ioperm(OBD_STS_PORT, 1, 0);
45             return 0;
46         }

```

首先在第 25、26 行允许 I/O 端口 0x60、0x64 的访问。然后使用 `send_cmd()` 函数，在写入指定的 I/O 端口 (port) 前获取板载控制器的 Status，确认 Output Buffer 为 empty，确认后向 I/O 端口写入值 (cmd)。在第 29 行为了点亮 CAPS\_LOCK 的 LED 灯，发送命令 (0xED)。然后写入用来亮灯的 CAPS\_LOCK 的 LED 值 (4)。按同样方法依次点亮 NUM\_LOCK、SCROLL\_LOCK 后，将所有的 LED 熄灭。

## 各式各样的键盘

现在市场上也出现了很多种 USB 连接的键盘等。虽然各自的规格并不相同，但是也无须担心。Linux 的 input 子系统中准备了十分方便的接口。通过访问 `/dev/input/eventN` 文件 \*，就可以从用户空间控制键盘 (\*N 为数字)。查找已分配给键盘的事件文件时，可以使用下列命令从 `/proc/bus/input/devices` 寻找键盘的设备。

```

# cat /proc/bus/input/devices
...
I: Bus=0011 Vendor=0001 Product=0001 Version=ab41
N: Name="AT Translated Set 2 keyboard"
P: Phys=isa0060/serio0/input0
S: Sysfs=/devices/platform/i8042/serio0/input/input3
U: Uniq=
H: Handlers=kbd event3
B: EV=120013
B: KEY=4 2000000 3803078 f800d001 feffffdf ffefefff ffffffff fffffffe
B: MSC=10
B: LED=7
...

```

从 `Handlers` 入口可以看出 `event3` 为键盘用的事件文件。EV 入口是表示可操作事件的位图 (bitmap)。当 EV 入口中有 LED 的位时，显示 LED 入口。如果有 LED 入口，表示可以使用事件文件对 LED 进行操作。LED 入口为位图，各个位表示所支持的 LED。LED 的位图意义如下。

```
#define LED_NUML 0x00
#define LED_CAPSL 0x01
#define LED_SCROLLL 0x02
#define LED_COMPOSE 0x03
#define LED_KANA 0x04
#define LED_SLEEP 0x05
#define LED_SUSPEND 0x06
#define LED_MUTE 0x07
#define LED_MISC 0x08
#define LED_MAIL 0x09
#define LED_CHARGING 0x0a
```

具体来说，可以用以下这些代码来控制。在第 34 行打开参数文件。将 /dev/input/eventN 作为 write 函数的参数。通过向事件文件写入，可以点亮或熄灭 LED。

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <fcntl.h>
6 #include <unistd.h>
7 #include <sys/ioctl.h>
8 #include <stdint.h>
9 #include <errno.h>
10 #include <linux/input.h>
11
12 #define OFF 0
13 #define ON 1
14
15 void usage(char **argv)
16 {
17         printf("Usage: %s <event-device>\n", argv[0]);
18         printf("      e.g. %s /dev/input/evdev0\n", argv[0]);
19         return;
20 }
21
22 int main (int argc, char **argv)
23 {
24
25         int fd;
26         int ret = 0;
27         struct input_event ev;
28
29         if (argc != 2) {
30                 usage(argv);
31                 exit(1);
32         }
33
34         if ((fd = open(argv[1], O_WRONLY)) < 0) {
35                 perror("evdev file open");
36                 usage(argv);
37                 exit(1);
38         }
```

```

39
40     /* Turn on a LED of CAPS_LOCK */
41     ev.type = EV_LED;
42     ev.value = ON;
43     ev.code = LED_CAPSL;
44     ret = write(fd, &ev, sizeof(struct input_event));
45     usleep(100000);
46     /* Turn off a LED of CAPS_LOCK */
47     ev.value = OFF;
48     ret = write(fd, &ev, sizeof(struct input_event));
49     usleep(100000);
50
51     /* Turn on a LED of NUM_LOCK */
52     ev.code = LED_NUML;
53     ev.value = ON;
54     ret = write(fd, &ev, sizeof(struct input_event));
55     usleep(100000);
56     /* Turn off a LED of NUM_LOCK */
57     ev.value = OFF;
58     ret = write(fd, &ev, sizeof(struct input_event));
59     usleep(100000);
60
61     /* Turn on a LED of SCROLL_LOCK */
62     ev.code = LED_SCROLLL;
63     ev.value = ON;
64     ret = write(fd, &ev, sizeof(struct input_event));
65     usleep(100000);
66     /* Turn off a LED of SCROLL_LOCK */
67     ev.value = OFF;
68     ret = write(fd, &ev, sizeof(struct input_event));
69     usleep(100000);
70
71     close(fd);
72
73     return 0;
74 }

```

## 参考文献

- IBM Personal System/2 Hardware Interface Technical Reference
- Intel®UPI-41AH/42AH UNIVERSAL PERIPHERAL INTERFACE 8-BIT  
SLAVE MICROCONTROLLER

——Akio Takebe

## HACK #50 PowerTOP

本节介绍显示应用程序电能消耗指标的 PowerTOP。通过这个工具可以看到影响 CPU 省电状态的应用程序的运行情况。

## 概要

PowerTOP 是由 Intel 公司运营的 LessWatts 项目开发出的工具，显示表示电能消耗的指标 WPS (Wakeups per second)。WPS 表示 CPU 从休眠状态迁移到运行状态的事件每秒发生次数。因此，如果 WPS 较多，则 CPU 相应地难以进入省电状态。C0 状态和 C5 状态之间的状态迁移需要花费不少时间，因此 WPS 越少效率越高。

图 6-12 所示为笔记本电脑，型号为 DELL Inspiron Mini 12 (两个 CPU)、操作系统为 Ubuntu 8.04 时的例子。

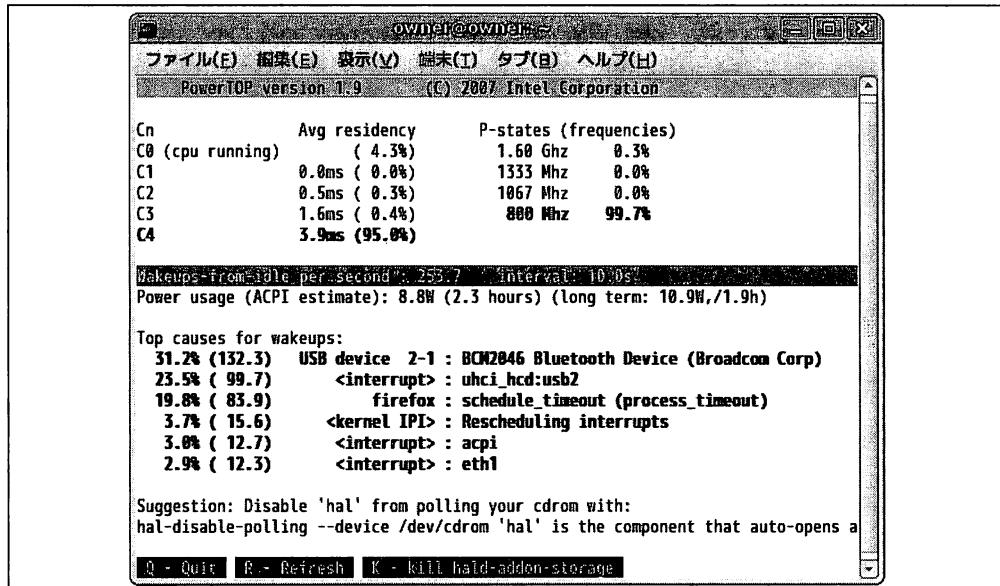


图 6-12 powertop 命令的输出

启动 firefox，在连接 LAN 电缆、USB 鼠标的状态下运行 PowerTop。图 6-12 显示了进程和 WPS 的比例。WPS 最多的是 Bluetooth 设备。

然后关闭 firefox，拔掉 LAN 电缆、USB 鼠标，禁用 Bluetooth。图 6-13 为此时 powertop 命令的输出。

整体的 WPS (wakeups-form-idle per second:) 减少到 62。

## PowerTOP 的详细情况和结构

图 6-12 左上方出现了 Cn。这表示的是 CPU 状态的层次和处于该状态的时间 (比例)。从这里可以看出 PowerTOP 测量期间内运行的 CPU 状态的比例。

图 6-14 显示的是运行了无限循环脚本 loop.sh 时的情况。

由于测量期间内一直无限循环，因此 C0 状态超过了 50%。测量的电脑有两个 CPU，其

中一个由 loop.sh 占用，因此表示整体上有 50% 为 C0 状态。

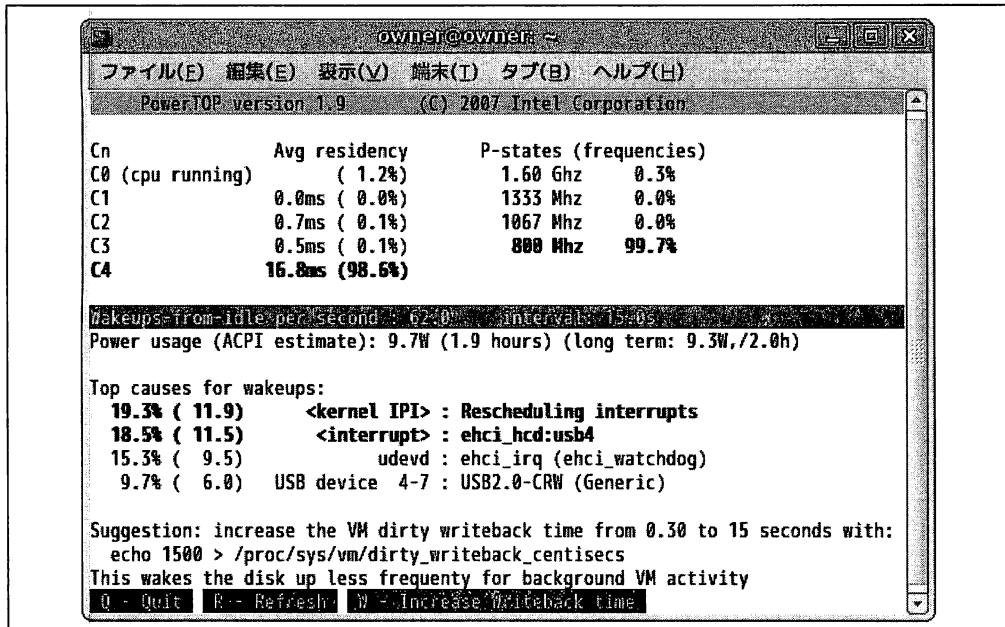


图 6-13 结束应用程序后的 powertop 命令

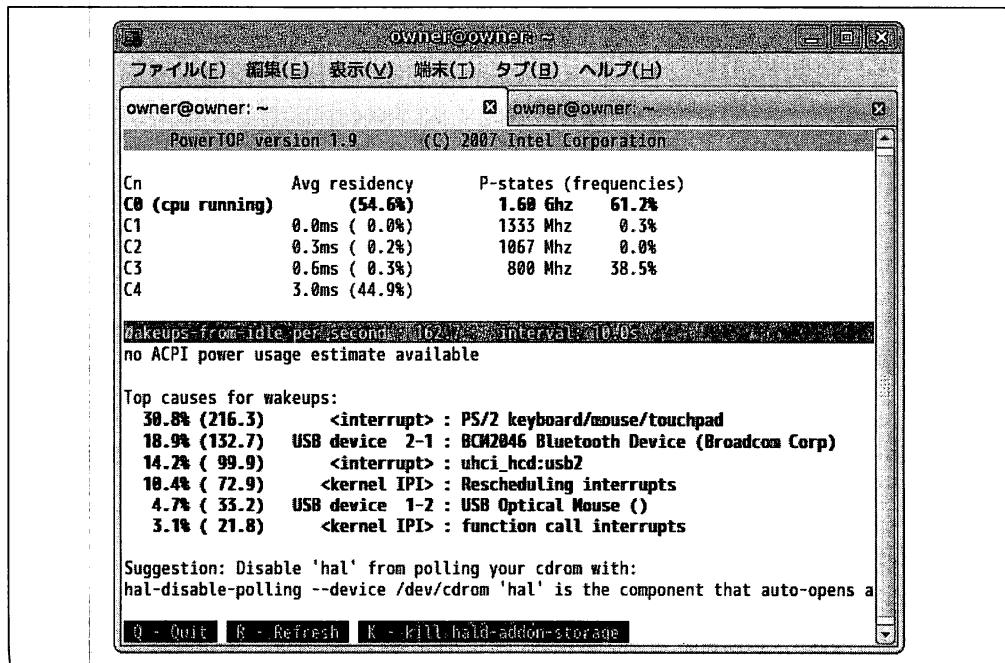
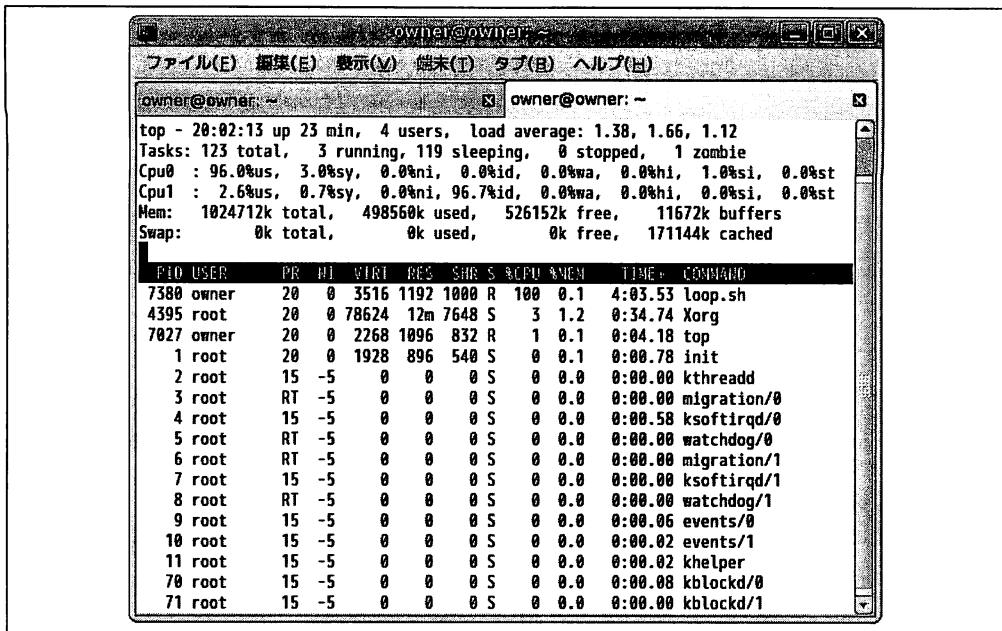


图 6-14 执行 loop.sh 时的 powertop 命令

图 6-15 显示的是此时的 top 命令。

执行 top 命令，按下键盘的【1】键，上面就会显示 Cpu0、Cpu1，显示各 CPU 的使用率。loop.sh 几乎使用了 100% 的 Cpu0。



```
owner@owner: ~
top - 20:02:13 up 23 min, 4 users, load average: 1.38, 1.66, 1.12
Tasks: 123 total, 3 running, 119 sleeping, 0 stopped, 1 zombie
Cpu0 : 96.0%us, 3.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 1.0%si, 0.0%st
Cpu1 : 2.6%us, 0.7%sy, 0.0%ni, 96.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 1024712k total, 498560k used, 526152k free, 11672k buffers
Swap: 0k total, 0k used, 0k free, 171144k cached

PID USER PR HI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
7380 owner 20 0 3516 1192 1000 R 100 0.1 4:03.53 loop.sh
4395 root 20 0 78624 12m 7648 S 3 1.2 0:34.74 Xorg
7027 owner 20 0 2268 1096 832 R 1 0.1 0:04.18 top
  1 root 20 0 1928 896 540 S 0 0.1 0:00.78 init
  2 root 15 -5 0 0 0 S 0 0.0 0:00.00 kthreadd
  3 root RT -5 0 0 0 S 0 0.0 0:00.00 migration/0
  4 root 15 -5 0 0 0 S 0 0.0 0:00.58 ksftirqd/0
  5 root RT -5 0 0 0 S 0 0.0 0:00.00 watchdog/0
  6 root RT -5 0 0 0 S 0 0.0 0:00.00 migration/1
  7 root 15 -5 0 0 0 S 0 0.0 0:00.00 ksftirqd/1
  8 root RT -5 0 0 0 S 0 0.0 0:00.00 watchdog/1
  9 root 15 -5 0 0 0 S 0 0.0 0:00.06 events/0
 10 root 15 -5 0 0 0 S 0 0.0 0:00.02 events/1
 11 root 15 -5 0 0 0 S 0 0.0 0:00.02 khelper
 70 root 15 -5 0 0 0 S 0 0.0 0:00.08 kblockd/0
 71 root 15 -5 0 0 0 S 0 0.0 0:00.00 kblockd/1
```

图 6-15 执行 loop.sh 时的 top 命令

## 关于颜色变化

图 6-14 中 WPS 的行是蓝色的。这是因为 WPS 的数值较少。

Wakeups-from-idle per second WPS 的数值为 0~10 时为绿色，11~25 时为黄色，26 以上为红色。但是，最后查看 C0 状态的时间如果在 25% 以上，就显示蓝色。

另外，图 6-14 中 WPS 的下一行有 no ACPI power usage estimate available。这是在连接 AC 电源的状态下执行 powertop 命令时输出的。拔掉 AC 电源线就会如图 6-12 一样输出瓦数和剩余时间等电池信息。

Top causes for wakeups：显示的是使 CPU 从休眠状态迁移到运行状态的每个事件种类的发生次数。

## 关于各事件的 WPS

PowerTOP 参照 /proc/interrupts 和 /proc/timer\_stats，重新排序显示使其看起来更清晰。另外，PowerTOP 在 Tickless 内核时能够正确测量，因此必须为 Linux 内核 2.6.21 以后的版本，并且启用 Tickless (CONFIG\_NO\_HZ=y) 进行编译。

各事件显示的 WPS 数字，是从硬件中断（/proc/interrupts）和 timer 启动次数（/proc/timer\_status）计算出的。

## PowerTOP 的建议

PowerTOP 界面下方在每次更新显示时就会显示省电的建议。建议仅显示与系统相符的内容。在 PowerTOP 界面上根据建议按键，就会执行这个建议。表 6-30 整理了部分建议，可以作为省电设置的参考。

表 6-30 PowerTOP 的建议列表

建议内容	按键动作详细内容
建议：使用下列命令可以禁用未使用的 Bluetooth 接口。 <code>hciconfig hci0 down; rmmod hci_usb</code> Bluetooth 消耗较多的电力，使 USB 进入 busy 状态	按【B】键，就会执行建议的命令
建议：使用下列命令可以禁用 NMI 监视。 <code>echo 0 &gt; /proc/sys/kernel/nmi_watchdog</code> NMI 监视是用来检出死锁（deadlock）的内核调试结构	按【N】键，就会执行建议的命令。NMI 看门狗（watchdog）请参考 Hack#57
建议：使用下列命令可以启用文件系统的 noatime 设置。 或按【T】键 不通过 noatime 记录访问时间，抑制磁盘 I/O 的发生	按【T】键，就会执行下列命令。 <code>/bin/mount -o remount, noatime, nodiratime</code>
建议：使用下列命令可以启用 CPU 调度程序的省电模式。 或按【C】键	按【C】键，就会执行建议的命令。CPU 调度程序的省电模式请参考 Hack #42
建议：使用下列命令可以将页面缓存的延迟写回间隔从 5 秒延长到 15 秒。 <code>echo 1500 &gt; /proc/sys/vm/dirty_writeback_centisecs</code> 这个设置可以降低 VM 背景运行访问磁盘的频率	按【W】键，就会执行建议的命令
建议：使用下列命令启用所有 CPU 的 ondemand 频率控制。 <code>echo ondemand &gt; /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor</code>	按【O】键，就会执行建议命令。CPU 频率控制请参考 Hack #42

建议内容	按键动作详细内容
建议：使用下列命令可以启用 SATA 连接 (link) 电源管理。 echo min_power > /sys/class/scsi_host/host0/link_power_management_policy 或按【S】键	按【S】键，就会执行建议的命令。硬盘的省电请参考 Hack #51
建议：使用下列命令可以禁用 TV 输出。 xrandr --output TV --off 或按【V】键	按【V】键，就会执行下列命令。 xrandr --auto;xrandr --output TV --off
建议：请使用下列命令禁用以太网的 Wake-on-LAN。 ethtool -s eth0 wol d 在 Wake-on-LAN 设置中 PHY 总是运行，因此消耗电能	按【W】键，就会执行建议的命令 (仅 eth0)。 WakeOnLAN 请参考 Hack #45
建议：按【U】键可以启用除输入设备以外的 USB 设备的自动待机模式	按【U】键，就会执行下列命令。 echo auto > /sys/bus/usb/devices/%s/power/control USB 的省电请参考 Hack #46
建议：按【P】键可以启用设备的电源管理	按【T】键，就会执行下列命令 (建议中的【P】键，在有的版本中是【T】键)。 echo auto > /sys/bus/pci/devices/%s/power/control echo auto > /sys/bus/spi/devices/%s/power/control echo auto > /sys/bus/i2c/devices/%s/power/control

表 6-31 所示为 PowerTOP 关于无线网络的建议。

表 6-31 PowerTOP 关于无线网络的建议列表

建议内容	按键动作详细内容
建议：使用下列命令可以启用无线的省电模式。 iwpriv < 无线设备名称 > set_power 5 这个设置会降低一些网络性能	按【W】键使无线设备进入省电模式。执行建议的命令
建议：使用下列命令可以启用无线的省电模式。 echo 5 > /sys/bus/pci/devices/< 无线设备名称 >/power_level 这个设置会降低一些网络性能	按【W】键启用无线省电模式。执行建议的命令。实际为 bug, echo 1 > /sys/bus/pci/devices/< 无线设备名称 >/power_level

(续)

建议内容	按键动作详细内容
建议：使用下列命令可以禁用未使用的 Wi-Fi 通信。 echo 1 > /sys/bus/pci/devices/< 无线设备名称 >/rfkill/rfkill0/state	按【I】键禁用 Wi-Fi 通信。执行建议的命令
建议：使用下列命令可以禁用未使用的 Wi-Fi 接口。 ifconfig < 网络设备 > down	按【D】键禁用无线 LAN。执行建议的命令
建议 “使用下列命令可以启用无线的省电模式。 iwconfig < 无线设备名称 > power timeout 500ms 这个设置会降低一些网络性能	按【W】键启用无线设备的省电模式。执行建议中的命令

PowerTOP 对进程结束的建议如表 6-32 所示。

表 6-32 PowerTOP 对进程结束的建议列表

建议内容	按键动作详细内容
建议：请禁用或卸载 gnome-power-manager。旧版本的 gnome-power-manager 过于频繁地启动检查电力，反而消耗电能	按【K】键就会通过 kill 系统调用向 gnome-power-manager 发送 SIGTERM，使其结束
建议：使用下列命令可以禁用 hal 的 cdrom 查询功能。 hal-disable-polling --device /dev/cdrom hal 提供插入 CD 时的自动执行功能，但禁用 SATA 的省电模式	按【K】键就会通过 kill 系统调用向 hal-addon-storage 发送 SIGTERM，使其结束
建议：请卸载 setroubleshoot-server，禁用 SE-Alert。对 SELinux 的违反策略发出警告的 SE-Alert 存在每 1 秒中断 10 次的 bug	按【K】键就会通过 kill 系统调用向 /usr/bin/sealert 发送 SIGTERM，使其结束

## 改善应用程序的示例

PowerTOP 的项目中进行了实际减少应用程序的 WPS 的修改。这些成果在下列 URL 上公开。对 kernel、firefox、xterm 等众多应用程序作出了贡献。

<http://www.Linuxpowertop.org/known.php>

## 小结

PowerTOP 是非常优秀的工具。应用程序的稳定性和性能虽然重要，但在使用电池的系统中省电也是非常重要的，PowerTOP 的指标对于改善应用程序是非常有效的。

## 参考文献

- PowerTOP  
<http://www.lesswatts.org/projects/powertop/>
- Documentation/hrtimer/timer\_stats.txt

——Naohiro Ooiwa

## HACK #51 硬盘的省电

本节介绍 SATA 的 LPM 和 hdparm 命令的电力控制。

从内核 2.6.24 开始支持 SATA 的 LPM (Link Power Management) 和 AHCI SATA 控制器 ALPM (Aggressive Link Power management)。

本节将介绍 LPM、ALPM 和使用 hdparm 命令进行省电设置。

### LPM

SATA 的 LPM (Link Power Management, 链路电源管理) 有 DIPM (Device Initiated Link Power Management) 和 HIPM (Host Initiated Power Management) 两种控制方法。

DIPM 是由设备控制链路状态。从操作系统设置一次 LPM，然后设备就会自动迁移到省电状态。

HIPM 通过 AHCI SATA 主控制器的 ALPM (Aggressive Link Power Management) 控制链路状态。因此 HIPM 必须 SATA 能够支持 AHCI，并在 BIOS 中设置为 AHCI 模式。

LPM 的链路状态有 ACTIVE、PARTIAL、SLUMBER 这三个状态。ACTIVE 为运行状态，不进行电能控制。PARTIAL 和 SLUMBER 为省电状态，SLUMBER 比 PARTIAL 更省电。

在 AHCI 规格中，从 PARTIAL 恢复到 ACTIVE 需要在 10 微秒以内完成，从 SLUMBER 恢复到 ACTIVE 需要在 10 毫秒以内完成。

与 CPU 等不同的是，省电状态不会进行 ACTIVE → PARTIAL → SLUMBER 的迁移。只会如图 6-16 所示的从 ACTIVE 进行迁移。

LPM 的设置是经由 sysfs 进行的。通过下面的设置使其能够迁移到 SLUMBER 状态。设置的字符串如表 6-33 所示。

```
# echo min_power > /sys/class/scsi_host/host0/link_power_management_policy
```

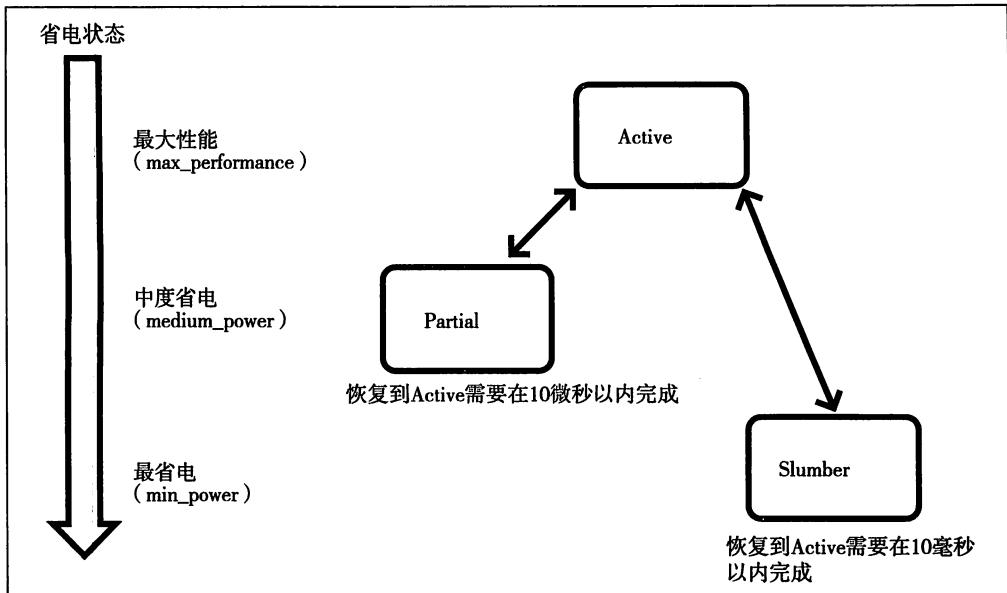


表 6-33 LPM 设置的种类

种    类	功    能
min_power	尽可能以最低电力状态运行。启用 DIPM。当处于 AHCI 模式时，迁移到 PARTIAL 状态和 SLUMBER 状态
max_performance	不进行电能控制。禁用 DIPM
medium_power	迁移到较低电能状态，但不会进入最低状态。禁用 DIPM，AHCI 模式的情况下仅迁移到 PARTIAL 状态

除 LPM 以外，还可以使用 hdparm 命令进行关于硬盘（SATA/PATA/SAS/IDE）的设置。设置中有关于省电的和关于 I/O 性能的内容。虽然省电与性能是矛盾的关系，但是对于服务器或台式机等有一些有效的设置。后半部分将介绍关于性能的设置。

## 显示正在使用的硬盘信息

首先使用 hdparm 命令查看硬盘的信息。可以确认能够支持的功能和当前的配置情况。

### 显示信息

-i 是旧的选项。在这里使用的是显示内容比 -i 更为详细的 -I 选项。下面是 -I 选项的输出结果。关于设置选项的地方用阴影表示。

```
# hdparm -I /dev/sda
/dev/sda:
```

ATA device, with non-removable media

Model Number: WDC WD1600AAJS-19M0A0  
 Serial Number: WD-WCAV32353380  
 Firmware Revision: 01.03E01  
 Transport: Serial, SATA 1.0a, SATA II Extensions, SATA Rev 2.5

Standards:

Supported: 8 7 6 5  
 Likely used: 8

Configuration:

...

Logical/Physical Sector size: 512 bytes  
 device size with M = 1024\*1024: 152627 MBytes  
 device size with M = 1000\*1000: 160041 MBytes (160 GB)  
 cache/buffer size = 8192 KBytes

Capabilities:

LBA, IORDY(can be disabled)  
 Queue depth: 32 -Q 选项 (硬件中支持的最大值)  
 Standby timer values: spec'd by Standard, with device specific minimum  
 R/W multiple sector transfer: Max = 16 Current = 16 -m 选项  
 Recommended acoustic management value: 128, current value: 128 -M 选项  
 DMA: mdma0 mdma1 mdma2 udma0 udma1 udma2 udma3 udma4 udma5 \*udma6 -X 选项  
 Cycle time: min=120ns recommended=120ns  
 PIO: pio0 pio1 pio2 pio3 pio4  
 Cycle time: no flow control=120ns IORDY flow control=120ns

Commands/features:

Enabled Supported:

- \* SMART feature set
- \* Security Mode feature set
- \* Power Management feature set
- \* Write cache -W 选项
- \* Look-ahead -A 选项
- \* Host Protected Area feature set
- \* WRITE\_BUFFER command
- \* READ\_BUFFER command
- \* NOP cmd
- \* DOWNLOAD\_MICROCODE
- Power-Up In Standby feature set
- \* SET\_FEATURES required to spinup after power up  
 SET\_MAX security extension
- \* Automatic Acoustic Management feature set -M 选项
- \* 48-bit Address feature set
- \* Device Configuration Overlay feature set
- \* Mandatory FLUSH\_CACHE
- \* FLUSH\_CACHE\_EXT
- \* SMART error logging
- \* SMART self-test
- \* General Purpose Logging feature set

...

- \* Native Command Queueing (NCQ) -Q 选项
- \* Host-initiated interface power management HIPM
- \* Phy event counters
- DMA Setup Auto-Activate optimization

```
...
Security:
  Master password revision code = 65534
...
```

这样就可以通过 -I 选项了解磁盘的详细情况。Commands/features: 显示的是磁盘功能列表。Enabled 列中有 \* 标记的，是这个硬盘支持的。

下面介绍 hdparm 命令的选项。

## 关于省电的设置

### APM

-B 选项进行的是 APM (Advanced Power Management) 的设置。值较小时比较省电，值较大时优先考虑性能。255 表示禁用 APM。

### 待机 standby、休眠模式

-y 选项可以将硬盘更改为待机模式。-C 选项显示的是当前电源模式的状态。

```
# hdparm -C /dev/sda
/dev/sda:
  drive state is:  active/idle
# hdparm -y /dev/sda
/dev/sda:
  issuing standby command
```

再次执行 -C 选项，就会显示 standby。

```
# hdparm -C /dev/sda
/dev/sda:
  drive state is:  standby
```

使用 -Y 选项可以设置为休眠模式。

```
# hdparm -Y /dev/sdb
/dev/sdb:
  issuing sleep command
```

### 待机超时

使用 -S 选项设置到待机 (spindown) 为止的超时时间。不访问磁盘的状态超过设置时间就会自动进入待机状态，节省电能。

关于设置值在 man 页面中作出了详细说明。下面是从 man 页面摘录的重点。241~255 的设置值请参考 man hdparm。

- 值为 0 时表示“禁用超时”，不自动进入待机模式。
- 值为 1~240 时表示 5 秒 × 设置值就是超时时间。超时为 5 秒至 20 分。

要在最短时间内进入待机模式，需要设置为 1 (5 秒)。

## 关于 I/O 性能的设置

### 预读功能

- 使用 -a 选项获取并设置文件系统的预读扇区。
- -A 选项指定是否有 IDE 驱动器的预读 (read-ahead)。
  - 使用 -A0 设置为无预读，使用 -A1 设置为有预读。

```
# hdparm -A1 /dev/sda
/dev/sda:
  setting drive read-lookahead to 1 (on)
  look-ahead      =  1 (on)
```

- 如果仅指定 -A 选项，就可以确认当前的预读设置。从 hdparm 的版本 7.0 开始可以使用。
- 使用 hdparm 命令的 -aA 选项可以确认当前的设置。

```
# hdparm -aA /dev/sda
/dev/sda:
  readahead      = 256 (on)    预读扇区数为 256
  look-ahead      =  1 (on)    启用预读
```

### 32 位 I/O 支持

使用 -c 选项进行 32 位 I/O 的设置。设置值为 0、1、3。32 位数据传输为 0 时表示禁用，为 1 时表示启用，为 3 时表示使用 sync 序列 sequence 的 32 位数据传输。

### DMA 设置

使用 -d 选项进行 IDE 的 DMA 设置。使用 -X 选项可以更改 Singleword DMA、Multiword DMA、Ultra DMA 的模式。

上例中的硬盘支持 mdma0 ~ 2、udma0 ~ 6，Ultra DMA 的模式 6 (udma6) 中显示了 \*。这表示当前启用的设置。

如果设备驱动程序、磁盘支持，只要按 hdparm -X mdma0 这样执行就可以改变。

### 同步

使用 -f 选项进行缓存的更新。在 hdparm 命令的内部依次执行 sync、fsync、fdatasync、sync，最后执行 ioctl (BLKFLSBUF)。这个 ioctl 需要文件系统支持该命令。可以在 grub-install 这样想要完全写出到磁盘时执行。

使用 -F 选项可以更新硬盘内的灯光缓存 (light cache)。

### 多扇区 I/O

使用 -m 选项进行多扇区 I/O 的设置。通过 1 次 I/O 操作向多个扇区进行数据传输。使

用 -I 选项, 如果显示 (例如 R/W multiple sector transfer:Max=16), 就可以指定到 16 扇区为止。下面是执行的例子。Current 显示的是当前的设置值。

```
# hdparm -I /dev/sda | grep multiple
      R/W multiple sector transfer: Max = 16          Current = 8
# hdparm -m 16 --yes-i-know-what-i-am-doing /dev/sda
# hdparm -I /dev/sda | grep multiple
      R/W multiple sector transfer: Max = 16          Current = 16
```

## 静音性能控制

使用 -M 选项可以对自动音响管理 (Automatic Acoustic Management, AAM) 进行设置。设置值为 128~254 或为 0。0 表示禁用。128 表示以静音性能为最优先 (低速), 254 表示设置为性能最优先。

```
# hdparm -M 254 /dev/sdb
```

设置为 254 就能听到硬盘运行的声音。在台式机上可能会比较明显。

## 命令队列

NCQ (Native Command Queuing) 是 Serial ATA II 所支持的功能。NCQ 将对硬盘驱动器的多条命令进行排队, 按照查找时间由长到短的顺序更换并执行命令的功能。

hdparm 命令的 -Q 选项用来设置这个 NCQ 队列的深度。必须内核和设备驱动程序都支持。可以通过 sysfs 下的文件是否具有写入权限来判断是否能够支持。

```
# ls -l /sys/block/sda/device/queue_depth
-rw-r--r-- 1 root root 4096 2010-03-21 20:30 /sys/block/sda/device/queue_depth
^^^^ 有写入权限
# cat /sys/block/sda/device/queue_depth
31
```

## 灯光缓存

使用 -w 选项可以设置灯光缓存。这是硬盘内部的缓存。0 表示禁用, 1 表示启用。

笔者在身边从旧到新一共 5 台机器上进行了确认, 发现灯光缓存默认都是启用的。停电或系统发生障碍等时, 灯光缓存的数据不会反映到磁盘上, 数据有可能会破坏。因此旧的 hdparm 中 -w 的设置为 (DANGEROUS), 但 RHEL6 安装的 hdparm 中 (DANGEROUS) 被串删除了。

## 参考文献

- Documentation/scsi/link\_power\_management\_policy.txt
- AHCI Specification  
<http://www.intel.com/technology/serialata/ahci.htm>

——Naohiro Ooiwa

## 第7章

---

# 调试

IT 系统越来越多样化，也变得越来越复杂。同时，出现的故障也变得非常复杂。要通过事前评估来网罗所有的实验在实际操作上也是很困难的。

因此，一旦发生故障，就需要迅速应对解决。为此就必须研究对策，作好万全的准备。

本章将介绍内核的调试功能。主要介绍检测死机的看门狗（watchdog）和内核崩溃转储（crash dump）功能。

## HACK #52 SysRq 键

本节介绍 SysRq 键的功能和使用方法。

SysRq 键一般在键盘的右上方。Magic System Request Key (Magic SysRq) 就是指通过这个 SysRq 键获取内核信息的功能。一般可以通过 proc 文件系统或命令来获取信息。但是如果系统死机，就无法输入命令。SysRq 键可以直接从内核输出信息。只要不是禁止中断状态，即使死机时也可以获取信息。SysRq 键在确认内核运行、调查内核死机原因等各种情况下都非常有效。

### 使用方法

要使用 SysRq 键，需要启用内核配置 CONFIG\_MAGIC\_SYSRQ，编译内核。

```
# make menuconfig
Kernel hacking    --->
 [*] Magic SysRq key
```

RedHat 系列的发布版的内核中并未安装 SysRq 功能。

启动后可以使用 sysctl 来切换启用、禁用。有的发布版在启动时是禁用的。可以使用下列命令来启用。

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

或

```
# echo 1 > /proc/sys/kernel/sysrq
```

如果在 `/proc/sys/kernel/sysrq` 中设置为 1，则所有 SysRq 键都可以使用。这个特殊文件的值是位掩码，还可以通过添加数字来限制可使用的 SysRq 键的命令。各值如表 7-1 所示。括号内为命令键（关于命令键将在后面介绍）。

表 7-1 在 `/proc/sys/kernel/sysrq` 中设置的位掩码

值	允许的命令
1	允许所有
2	允许控制控制台日志级别（0 ~ 9）
4	允许控制键盘（kr）
8	允许显示进程等信息（lptwmcz）
16	允许 Sync 命令（s）
32	允许只读下的重新挂载（u）
64	允许发送信号（ei）
128	允许重启（b）
256	允许控制即时进程（q）

要允许 Sync(s) 和重新挂载 (u)，而不能进行其他操作，可以进行如下设置。

```
# echo 48 > /proc/sys/kernel/sysrq
```

这个控制是从键盘和串行 (serial) 接口限制控制台的输入。后面将介绍的 `/proc/sysrq-trigger` 的操作没有限制。

另外，在内核启动参数中，无论 `/proc/sys/kernel/sysrq` 的设置如何，都可以启用 SysRq 键。

```
boot> linux sysrq_always_enabled
```

这个内核启动参数在 Linux 2.6.20 以后的版本中可以使用。

## SysRq 键的输入方法

SysRq 键有很多命令。可以使用这些命令控制机器或输出信息。命令键是指用来指定这些动作的键输入。

从键盘输入 SysRq 键时，需要同时输入 Alt 键、SysRq 键和命令键。

从串口控制台输入时，是在发送 break 信号后 5 秒内输入命令键。

此外向 proc 文件系统 `/proc/sysrq-trigger` 写入命令键可以进行与 SysRq 相同的动作。

```
# echo [命令键] > /proc/sysrq-trigger
```

## SysRq 命令键

首先将上游内核的版本中支持的命令汇总到表 7-2 中。

表 7-2 各内核版本的命令键支持情况

命 令 键	命 令 名 称	2.6.9 ~	2.6.16 ~	2.6.29	2.6.30 ~
		2.6.11	2.6.19	2.6.35	
0 ~ 9	loglevel0 ~ 8	○	○	○	○
b	reBoot	○	○	○	○
c	Crashdump <sup>注1</sup>		○	○	○
d	show-all-locks(D) <sup>注2</sup>		○	○	○
e	tErm	○	○	○	○
f	Full		○	○	○
i	kIlli	○	○	○	○
j	thaw-filesystem(J) <sup>注3</sup>				○
k	saK	○	○	○	○
l	aLlcpus <sup>注4</sup>	○	○	○	○
m	showMem		○	○	○
n	Nice	○	○	○	○
p	showPc			○	○
q	show-all-timers(Q) <sup>注5</sup>	○	○	○	○
r	unRaw	○	○	○	○
s	Sync	○	○	○	○
t	showTasks	○	○	○	○
u	Unmount			○	○
w	shoW-blocked-tasks			○	○
z	dump-ftrace-buffer(Z) <sup>注6</sup>		○		○

○表示内核能够支持。命令名称是表示命令键内容的名称。命令名称中的大写字母就是键。例如，命令“reBoot”中的大写字母 B 就是命令键。命令名称最后有 ( ) 的，( ) 内的文字为键。

如果输入表 7-2 以外的键，就会输出如下的帮助信息，可以确认命令键。

#### 例 7-1 Linux2.6.32-44.2.el6 的情形

注 1：需要启用 CONFIG\_KEXEC。

注 2：到 2.6.17 为止的版本需要启用 CONFIG\_DEBUG\_MUTEXES，从 2.6.18 开始需要启用 CONFIG\_LOCKDEP。

注 3：需要启用 CONFIG\_BLOCK。

注 4：需要启用 CONFIG\_SMP。

注 5：需要启用 CONFIG\_GENERIC\_CLOCKEVENTS。

注 6：需要启用 CONFIG\_TRACING。

```
SysRq : HELP : loglevel(0-9) reboot Crash terminate-all-tasks(E) memory-
full-oom-kill(F) kill-all-tasks(I) thaw/filesystems(J) saK show-backtrace-
all-active-cpus(L) show-memory-usage(M) nice-all-RT-tasks(N) powerOff show-
registers(P) show-all-timers(Q) unRaw Sync show-task-states(T) Unmount force-
fb(V) show-blocked-tasks(W) dump-ftrace-buffer(Z)
```

主要命令键的内容如表 7-3 所示。

表 7-3 SysRq 命令键的详细内容

命 令 键	说 明
0 ~ 9	设置控制台层次。与设置 /proc/sys/kernel/printk 是相同的
b	重新启动 Linux 内核
c	获取崩溃转储。用于故意造成系统故障时
d	输出获取的所有 lock 信息。但是 TASK_RUNNING 状态的进程获取的 lock 不显示。这是因为这个进程立刻就会释放
e	向 init 进程 (PID 为 1) 以外的所有进程发送 SIGTERM 信号
f	运行 OOM Killer (详细内容参考 HACK #16)
i	向 init 进程 (PID 为 1) 以外的所有进程发送 SIGKILL 信号
l	输出系统中的所有 CPU 的栈。还会显示进程栈的回溯 (backtrace)
n	将所有实时进程强制改变为普通进程。与在 sched_setscheduler(2) 中指定 SCHED_NORMAL 为调度策略相同
m	输出内存、交换区的状态
p	输出 CPU 的寄存器和进程的信息。有多个 CPU 时，仅输出处理键中断的 CPU 的信息
q	输出所有计时器的信息
s	在所有文件系统中尝试 sync (把内存缓冲区的内容写入磁盘)。内部是通过强制运行 pdflush 来实现的。在命令键 b 之前执行，就可以减少文件系统或文件破坏的风险，重启内核
t	输出运行中的所有进程的栈、回溯
u	对所有文件系统尝试只读的重新挂接
w	仅输出处于 UNINTERRUPTABLE 状态 (无视信号的进程状态) 的进程信息
z	输出 ftrace 的缓冲区

RHEL5 的命令键 w 与上游内核中的不同。命令键 w (showcpus) 输出系统中的所有 CPU 的栈。这与上游内核的命令键 l (aLlcpus) 进行的操作相同。RHEL6 的 w 与上游内核相同。RHEL4/5/6 的支持情况如表 7-4 所示。

表 7-4 RHEL4、RHEL5、RHEL6 的 SysRq 键支持情况

命令键	命令名称	RHEL4	RHEL5	RHEL6
0 ~ 9	loglevel0 ~ 8	○	○	○
b	reBoot	○	○	○
c	Crash	○	○	○
d	show-all-locks(D)		○	○
e	terminate-all-tasks(E)	○	○	○
f	memory-full-oom-kill(F)		○	○
i	kill-all-tasks(I)	○	○	○
j	thaw-filesystems(J)			○
k	saK	○	○	○
l	show-backtrace-all-active-spus(L)			○
m	show-memory-usage(M)	○	○	○
n	nice-all-TR-tasks(N)		○	○
p	show-registers(P)	○	○	○
q	show-all-timers(Q)			○
r	unRaw	○	○	○
s	Sync	○	○	○
t	show-task-states(T)	○	○	○
u	Unmount	○	○	○
w	shoWcpus		○	
w	show-blocked-tasks(W)			○
z	dump-ftrace-buffer(Z)			

## 上游内核的 SysRq 键显示的例子

下面是命令键 m (showMem) 的输出示例。输出的是内存和交换区的使用情况。

```
SysRq : Show Memory
Mem-info:
Node 0 DMA per-cpu:
...
Active:128436 inactive:87353 dirty:93 writeback:0 unstable:0
free:6411 slab:30787 mapped:1294 pagetables:435 bounce:0

Swap cache: add 0, delete 0, find 0/0
Free swap  = 1020116kB
Total swap = 1020116kB
261920 pages of RAM
5716 reserved pages
9262 pages shared
```

```
0 pages swap cached
```

下面是命令键 **t** (showTasks) 的输出示例。

```
SysRq : Show State
task          PC stack  pid father
...
sshd          S ffffffff8048d5e0      0 3121 2908
fffff81003ec31a28 0000000000000082 0000000000000000 ffff810032c455b8
...
Call Trace:
[<fffffff80471eb6>] schedule_timeout+0x1e/0xad
[<fffffff8036bc83>] tty_poll+0x5f/0x6d
...
[<fffffff802982f4>] sys_select+0xc1/0x183
[<fffffff8028c00a>] sys_write+0x45/0x6e
...
```

使用命令键 **w** (show-blocked-tasks) 也会输出同样的信息。

下面是命令键 **l** (aLlcpus) 的输出示例。输出的有安装的模块、SysRq 键的处理程序运行的 CPU 寄存器值、栈、回溯，这里省略部分内容。

```
SysRq : Show backtrace of all active CPUs
CPU 0:
Modules linked in: ipmi_watchdog ipmi_devintf ipmi_si ipmi_msghandler
...
Pid: 0; comm: swapper Not tainted 2.6.26 #2
RIP: 0010: [<fffffff80211d85>] [<fffffff80211d85>] mwait_idle+0x41/0x44
RSP: 0018:fffffff8087df60  EFLAGS: 00000246
RAX: 0000000000000000 RBX: 0000000000000000 RCX: 0000000000000000
...
Call Trace:
[<fffffff8020ab7b>] ? cpu_idle+0x6d/0x8b

CPU1:
...
Call Trace:
<IRQ>  [<fffffff8037c71d>] showacpu+0x0/0x52
[<fffffff8037c75d>] showacpu+0x40/0x52
[<fffffff8021a118>] smp_call_function_interrupt+0x3b/0x62
[<fffffff8020c8d6>] call_function_interrupt+0x66/0x70
<EOI>  [<fffffff8049994>] :ext3:ext3_bmap+0x0/0x78
...
```

使用命令键 **p** (showPc) 也会输出同样的信息。使用命令键 **q** (show-all-timers) 输出与 `/proc/timer_list` 相同的信息。如果是启用 `CONFIG_SCHEDSTATS` 编译的内核，在命令键 **t** 和 **w** 输出的信息中添加与 `/proc/sched_debug` 相同的信息。

## 各种情况下的使用方法

### 某进程看似停止的情况

使用命令键 **t** 确认进程运行的地方。通过确认回溯，就有可能找出等待获取资源、等待

获取 lock 等的进程之所以停止的原因。如果在等待获取 lock 时停止，有可能通过并用命令键 **d** 根据 lock 的依存关系找出原因。如果正在用户空间运行，就有可能是编程错误。

### 机器看似死机的情况

为了保全数据，使用命令键 **s** 将内存中还未完成写入的数据写出到磁盘。然后，使用命令键 **u** 将文件系统以只读形式重新挂接，可以防止发生数据不匹配。需要重新开始工作时，使用命令键 **b** 重启内核。如果不需重新开始工作，可以使用命令键 **c** 提取崩溃转储，找出死机的原因。

此外，还可以在死机时通过命令键 **i** 向所有进程发送 SIGKILL 信号尝试从死机状态恢复。内核在允许中断状态下死机时，在死机的状态下执行几次命令键 **w** 或 **p**，就可以知道死机的位置。进行再现时，事先禁用看门狗计时器就可以分析出死机的原因。

### 由于某些故障输出 Oops 信息等的情况

未进行内核崩溃转储的设置时，有时控制台的最后会输出 `Kernel panic - not syncing:~`，然后就只能重新启动。这时，可以先使用命令键 **s** 尝试把文件系统的数据写入磁盘，再重新启动，这更为安全。

## 小结

本节介绍了 SysRq 键。每次内核版本更新时都会添加方便的新功能。由于某些故障导致内核没有反应时，可以使用 SysRq 键来确认内存使用量和进程的状态。当发生故障机器停止时，建议在按电源键之前首先使用 SysRq 键收集信息。

另外，命令键 **c** 经常用于确认内核崩溃转储的情况。

## 参考文献

- Documentation/sysrq.txt

——Naohiro Ooiwa

# HACK #53 使用 diskdump 提取内核崩溃转储

本节介绍 RHEL4 等中采用的 diskdump 的使用方法。

diskdump 是 RHEL4 等 RedHat 系列的部分发布版中所采用的内核崩溃转储功能。需要注意的是，这里介绍的功能中有的在某些发布版中不能使用。本节将介绍在 RHEL 4.7 中已经确认的步骤。使用的架构是 x86\_64。

## 内核崩溃转储

内核崩溃转储是将系统内存的内容输出到文件的功能。如果应用程序发生段错误

segmentation fault，就会输出 core 文件，而内核崩溃转储就类似于内核的 core 文件。发生故障时输出当时的系统内存内容。通过对其进行分析，就可以了解此时内核进行了怎样的处理。因此内核崩溃转储是不可或缺的调试功能。

## diskdump 的限制事项

为了使 diskdump 内核中即使发生所有故障也能获取内核崩溃转储，在设置为禁止中断的状态下获取转储。磁盘驱动程序必须能够支持不使用中断执行 I/O 的结构——轮询 I/O。在 RHEL 4.7 中下列驱动程序能够支持轮询 I/O。

```
aic7xxx      sym53c8xx      IDE      ibmvscsi
aic79xx      sata_promise  qla2xxx   sata_nv
ipr          ata_piix      lpfc      aacraid
megaraid     CCISS         stex
mptfusion    megaraid_sas  ips
```

diskdump 不经由文件系统或设备映射器功能，直接对设备驱动程序进行操作。因此不能向文件或 LVM 上生成的磁盘分区写入。

## 启用 diskdump

diskdump 中必须指定转储用的分区。可以准备用于转储的分区，也可以将 swap 分区指定为转储位置。但是不使用 compress 选项时，分区必须大于系统中安装的内存。这次不使用交换分区，而是准备了专用于转储的分区 /dev/sda3。配置文件 /etc/sysconfig/diskdump 中有下列内容。

```
DEVICE=/dev/sda3
```

然后将 /dev/sda3 格式化，作为转储用分区。

```
# service diskdump initialformat
```

启用 diskdump 服务。

```
# chkconfig diskdump on
# service diskdump start
```

可以通过 service 命令或 /proc/diskdump 来确认 diskdump 是否已启用。/proc/diskdump 中显示的内容如下。

```
# cat /proc/diskdump
# sample_rate: 8
# block_order: 2
# fallback_on_err: 1
# allow_risky_dumps: 1
# dump_level: 0
# compress: 0
# total_blocks: 98197
#
sda3 14329980 2441880
```

另外，需要设置 `sysctl` 变量 `kernel.panic`，使 Linux 内核在转储提取完成后自动重新启动。这是在 `/etc/sysctl.conf` 中进行设置的。

```
kernel.panic=10
```

通过这样的设置，转储完成后 10 秒就会重新启动。指定设置后使用 `sysctl` 命令启用设置。

```
# sysctl -p
```

到这一步设置就完成了，可以尝试提取崩溃转储。

```
# echo c > /proc/sysrq-trigger
```

转储文件在内核崩溃后重新启动时另存为 `/var/crash/127.0.0.1-<日期>/vmcore`。可以使用 `crash` 命令确认转储文件的内容。关于 `crash` 命令的内容请参考 HACK #59。

## 使用压缩和部分转储功能缩小转储文件的大小

`diskdump` 可以缩小转储文件。关于 `Kdump` 的内容请参考 HACK #54。

压缩功能和部分转储功能可以使用 `diskdump` 内核模块的内核选项来指定。要启用压缩功能可以将 `compress` 选项指定为 1。部分转储功能则是在 `dump_level` 选项中指定转储级别。表 7-5 所示为每个转储级别跳过 `skip` 的页面种类。

但是，启用这个功能时需要注意，有些种类的页面在跳过时需要检索内核内部内存管理用的列表。

有时故障是由于该列表的破坏而引起的，这是因为 `diskdump` 在检索列表的过程中有时也会引起双重重大故障或系统死机。

表 7-5 转储级别与跳过的页面

转储级别	缓存页面 (cache_page)	私有缓存 (cache_private)	零页面 (zero_page)	可用页面 (free_page)	用户页面 (user_page)
0					
1	×		×		
2				×	
3	×		×	×	
4					×
5	×		×		×
6				×	×
7	×		×	×	×
8					×
9	×		×		×

(续)

转储级别	缓存页面 (cache page)	私有缓存 (cache private)	零页面 (zero page)	可用页面 (free page)	用户页面 (user page)
10			×		×
11	×	×	×		×
12				×	×
13	×	×		×	×
14			×	×	×
15	×	×	×	×	×
17	×				
19	×		×		
21	×			×	
23	×		×	×	
25	×				×
27	×		×		×
29	×			×	×
31	×		×	×	×

如果使用这个功能，建议选择转储级别 19。这是因为选择 19 时，就不需要检索内存管理用的列表。按下列方式在 `/etc/modprobe.conf` 中记载传递到 `diskdump` 内核模块的选项。

```
options diskdump dump_level=19 compress=1
```

为了启用这个设置，重新启动 `diskdump` 服务。

```
# service diskdump restart
```

通过 `/proc/diskdump` 可以确认指定的选项是否正确。

```
# cat /proc/diskdump
# sample_rate: 8
# block_order: 2
# fallback_on_err: 1
# allow_risky_dumps: 1
# dump_level: 19
# compress: 1
# total_blocks: 98197
#
sda3 14329980 2441880
```

## 发生故障时通过邮件通知

`diskdump` 具有将转储文件保存到 `/var/crash/` 之后启动用户定义的脚本的功能。使用

这个功能，还可以在提取转储时（即发生系统故障时）通过邮件通知。`/usr/share/doc/diskdumputils-<version>/example_scripts/` 下有示例，可以尝试一下。本次使用的是 `diskdump-success` 脚本。将其复制到 `/var/crash/scripts/` 下，进行如下编辑。

```
# cat /var/crash/scripts/diskdump-success
#!/bin/sh

ADDRESS=*****@oreilly.com

mail -s "[diskdump] 'hostname' crashed" $ADDRESS <<_EOF
The machine 'hostname' crashed.
Writing crash dump to $1
_EOF

# savecore always returns 0 whatever the result of this script because this is
# called after a dump file is created.

exit 0
```

## 将转储输出到的设备冗长化

在本节前面的“diskdump 的限制事项”中提到，diskdump 不经由文件系统，直接访问磁盘驱动程序。因此，如果转储用分区所使用的磁盘驱动程序发生故障，提取转储就有可能失败。因此 diskdump 可以将多个分区指定为转储用分区。在 `/etc/sysconfig/diskdump` 下进行具体设置的方法如下。

```
DEVICE=/dev/sda3:/dev/had
```

`/dev/sda3` 是至今为止使用的转储专用分区。在笔者的环境下，mptfusion 驱动程序正在运行。另一方面，`/dev/hda` 是由 IDE 驱动程序运行的其他磁盘设备。如果 mptfusion 驱动程序发生故障，向 `/dev/sda3` 写入转储就有可能失败。在这种情况下，diskdump 向下一个输入的 `/dev/hda` 进行转储。这是 IDE 驱动程序，因此可以顺利进行转储的写入。

## 小结

diskdump 在 RHEL4 系列等中都已采用。必须要有转储用的分区。另外，磁盘驱动程序也必须能够支持。而在下一代转储功能——Hack #54 中提到的 Kdump 中则不需要这些条件。

## 参考文献

关于配置的详细内容在 diskdump README 的 README 文件 (`/usr/share/doc/diskdumputils-<version>/README`) 中可以看到。

——Naohiro Ooiwa

# HACK #54 使用 Kdump 提取内核崩溃转储

本节介绍最近的发布版中采用的 Kdump 的使用方法。

Kdump 是从 Linux 2.6.13 开始安装到主线 (mainline) 中的内核崩溃转储功能。在 2.6.13 以后的内核的 Linux 发布版中都可以使用。本节将介绍在 RHEL6 中进行确认的 Kdump 使用步骤。所使用的架构为 x86\_64。

## 启用崩溃转储

首先向内核启动参数添加 `crashkernel=128M`。分配 128MB 的内存用于 Kdump。笔者的环境在 RHEL6 的 `/etc/grub.conf` 中显示下列内容。

```
title RHEL6 (2.6.32-71.el6.x86_64)
root (hd0,0)
kernel /boot/vmlinuz-2.6.32-71.el6.x86_64 root=UUID=987aa4dd-a712-4ff3-8ad9-28edb4ddcaae ro rd_NO_LUKS rd_NO_LVM rd_NO_MD rd_NO_DM LANG=ja_JP.UTF-8 KEYBOARDTYPE=pc KEYTABLE=jp106 crashkernel=128M rhgb quiet
initrd /boot/initramfs-2.6.32-71.el6.x86_64.img
```

从 RHEL6 开始扩展 `crashkernel` 内核启动参数，也可以按下列形式设置。

```
crashkernel=<range1>:<size1>[,<range2>:<size2>,...] [@offset]
```

可以不根据机器安装的物理内存大小分配内存，或者改变分配的内存量。例如，可以指定如下内存分配情况。

```
crashkernel=512M-2G:64M,2G-:128M
```

它表示的意义是：

- 如果物理内存不足 512MB，则不分配内存作为 Kdump 用。
- 如果物理内存大于 512MB 小于 2GB，则分配 64MB 作为 Kdump 用。
- 如果物理内存大于 2GB，则分配 128MB。

进行这样的指定后，即使从机器上拔下内存，也无须改变参数。

内存分配成功后，内核启动时就会输出下列信息。

```
# dmesg
...
Reserving 128MB of memory at 32MB for crashkernel (System RAM: 1022MB)
```

然后，启用 `kdump` 服务。使用 `chkconfig` 命令、`service` 命令。

```
# chkconfig kdump on
# service kdump start
```

可以通过 `service` 命令或 `/sys/kernel/kexec_crash_loaded` 确认是否已设置成功。

如果使用 `service` 命令显示出如下信息，就表示设置已启用。

```
# service kdump status
```

```
Kdump is operational
```

kexec\_crash\_loaded 的内容为 1 时表示设置已启用。

```
# cat /sys/kernel/kexec_crash_loaded
1
```

准备工作做好后，就可以尝试提取崩溃转储。

```
# echo c > /proc/sysrq-trigger
```

如果转储成功，内核重新启动后就会在 /var/crash/ 下生成目录，其中存放 vmcore 文件。

## 使用 makedumpfile 缩小转储的文件大小

使用前面介绍的设置，将生成与实际安装内存量大小相同的崩溃转储文件。也就是说，如果安装了 8GB 的内存，转储文件也是 8GB。但是 Kdump 也和 diskdump 一样可以压缩转储映像，使其变小。kexec-tools 中的 makedumpfile 命令就是负责这个处理过程的实用程序。使用这条命令前需要在 /etc/kdump.conf 中添加 core\_collector 的设置。

```
ext3 /dev/sda5
core_collector makedumpfile -c
```

一开始的 ext3/dev/sda5 指定了具有 root 文件系统的设备。把转储文件输出到这个分区的 ./var/crash 下。要将转储输出的位置设置为其他分区，例如，设置为挂接到 /dump 目录下的 /dev/sda6，需进行下列操作。

```
ext3 /dev/sda6
path .
```

这样，在 /dump 目录下就会生成各个日期的目录，把转储输出到这里。

-c 是压缩选项。设置转储级别的 -d 选项也非常方便。转储级别选项用来指定崩溃转储中不包括的页面（内存）种类。表 7-6 所示为各转储级别跳过的页面种类。

表 7-6 跳过的页面种类

转储级别	零页面 (zero page)	缓存页面 (cache page)	私有缓存 (cache private)	用户数据	可用页面 (free page)
0					
1		×			
2			×		
3	×		×		
4			×	×	
5	×	×		×	

(续)

转储级别	零页面 (zero page)	缓存页面 (cache page)	私有缓存 (cache private)	用户数据	可用页面 (free page)
6		×	×		
7	×	×	×		
8					×
9	×				×
10		×	×		×
11	×	×			×
12		×	×	×	×
13	×	×	×		×
14		×	×		×
15	×	×	×		×
16					×
17	×				×
18		×			×
19	×	×			×
20		×	×		×
21	×	×	×		×
22		×	×		×
23	×	×	×		×
24				×	×
25	×			×	×
26		×		×	×
27	×	×		×	×
28		×	×	×	×
29	×	×	×	×	×
30		×	×	×	×
31	×	×	×	×	×

转储级别可以通过表 7-6 中的数值来指定。例如，当不想包括零页面和可用页面时，可以将转储级别指定为 17。还可以像下面这样用“,” 将转储级别分开，指定两个转储级别。

```
core_collector makedumpfile -c -d 11,31
```

通过这个配置，`makedumpfile` 命令如果在使用转储级别 11 时失败，就会使用转储级别

31 重试。这个配置在磁盘容量不足且 `makedumpfile` 命令失败时十分有效。这是因为转储级别 31 跳过的页面比转储级别 11 多，转储文件更小。

要使设置生效，需要重启 `kdump` 服务。

```
# service kdump restart
```

使用前面介绍的方法确认转储。根据 `vmcore` 文件的大小可以判断出提取的转储是否已压缩。已压缩的转储文件就不再是 ELF 格式，因此不能使用 `gdb` 进行调试，必要时请使用 `crash` 命令。

使用 `makedumpfile` 提取自己重新构建的内核的转储时，需要将带调试信息的内核放到下列位置。

```
/usr/lib/debug/lib/modules/'uname -r'/vmlinux
```

在最近的 RHEL6 的发布版中，是不需要带调试信息的内核包的。这是因为内核中添加了 `vmcoreinfo` 功能。`kexec-tools` 数据包也相应地进行了修改。如果所使用的系统的内核和 `kexec-tools` 同时都支持 `vmcoreinfo`，则不需要专门安装带调试信息的内核。

## 向远程服务器传输崩溃转储

`Kdump` 具有使用 NFS 或 SSH 将获取的转储文件向远程传输的功能。

在 `/etc/kdump.conf` 添加 `net` 的设置。

```
/* 挂接 NFS 传输时 */
net <服务器名称或 IP 地址>:<导出的目录>
/* 经由 SSH 传输时 */
net <用户名>@<服务器名称或 IP 地址>
```

使用 NFS 时，需要在导出的目录下创建 `./var/crash` 目录。使用 SSH 时，将转储传输到远程服务器的 `/var/crash` 目录下。`/etc/kdump.conf` 中设置的登录用户必须具有向 `/var/crash` 写入的权限。如果觉得不够安全，可以另外准备转储用的目录，并将转储到的目录进行如下更改。

```
path /dump
```

转储用目录必须事先创建。使用 SSH 时指定为 `path` 的目录不是相对路径而是绝对路径。此外，还需要先输入公开密码，使人们不用输入密码就能登录。使用 `Kdump` 的 `init` 脚本的 `propagate` 选项，脚本就会进行这个操作。

```
# service kdump propagate
```

`link_delay` 指定的是从连接到 NIC 到开始传输为止的等待时间，单位为秒。这是因为有时 NIC 初始化需要花费一定时间。笔者为了以防万一，添加了 `link_delay` 的设置。

```
link_delay 10
```

注意事项：在配合使用 `makedumpfile` 和 SSH 时，就需要对转储文件进行转换。从输出到远程服务器的转储文件，可以看到文件名为 `vmcore.flat`。这样从 `crash` 命令是无法直接读入的，因此需要按下列方式转换文件格式。

```
# makedumpfile -R vmcore < vmcore.flat
```

## 小结

本节介绍了使用 Kdump 提取内核崩溃转储的方法。

## 参考文献

RHEL5 系列的使用方法请参考《Debug Hacks》中的 HACK # 20 “使用 Kdump 提取内核崩溃转储”。

——Naohiro Ooiwa

## HACK #55 崩溃测试

本节使用 `lkdtm` 进行内核的崩溃测试。

介绍为了测试 HACK #54 中介绍的内核崩溃转储功能是否正常运行，使用 `lkdtm` 功能在内核的各种位置使用崩溃测试的方法。

崩溃转储功能虽然在开发过程中非常完善，但也有可能因为 PC 的各种设备或设备驱动程序的操作导致运行失败。事先找出这些问题并进行修正，才能保证系统的品质。

`lkdtm` 是 Linux Kernel Dump Test Module 的缩写，能通过各种处理使内核崩溃。这个模块可以测试 Kdump 等崩溃转储功能是否能够正常运行。`lkdtm` 是用来进行崩溃测试这种特殊测试的，因此一般发布版内核中并未安装它。使用 `lkdtm` 时，需要启用 `CONFIG_LKDTM` 编译内核。`CONFIG_LKDTM` 可以通过选择下列内核选项来启用。

```
Kernel hacking - - - >
<M> Linux Kernel Dump Test Tool Module
```

`lkdtm` 的崩溃测试运行通过内核模块的变量或 `debugfs` 文件系统的接口来指定。需要指定发生崩溃的位置和崩溃原因这两个值作为崩溃测试运行的参数。将这些值作为内核模块参数时，分别作为 `cpoint_name` 和 `cpoint_type` 变量。`cpoint_name` 的种类、`cpoint_type` 的种类分别如表 7-7、表 7-8 所示。

表 7-7 崩溃位置 (`cpoint_name`) 的种类

<code>cpoint_name</code>	说 明
<code>INT_HARDWARE_ENTRY</code>	硬件中断处理程序共同入口 ( <code>do_IRQ</code> )
<code>INT_HW_IRQ_EN</code>	中断的处理程序处理 ( <code>handle_IRQ_event</code> )
<code>INT_TASKLET_ENTRY</code>	软中断 ( <code>tasklet_action</code> )

(续)

cpoint_name	说 明
FS_DEVRW	从文件系统向块设备发出的 I/O 请求处理 (ll_rw_block)
MEM_SWAPOUT	内存的换出处理 (shrink_inactive_list)
TIMERADD	添加计时器 (hrtimer_start)
SCSI_DISPATCH_CMD	发布 SCSI 命令 (scsi_dispatch_cmd)
IDE_CORE_CP	针对 IDE 设备的 ioctl (generic_ide_ioctl)
DIRECT	内核模块的安装处理或延长向 debugfs 的写入

表 7-8 崩溃原因 (cpoint\_type) 的种类

cpoint_type	说 明
PANIC	调用 panic() 函数
BUG	调用 BUG 宏
EXCEPTION	向 NULL 指针写入值, 使产生例外
LOOP	执行无限循环
OVERFLOW	进行递归调用
CORRUPT_STACK	产生栈溢出
UNALIGNED_LOAD_STORE_WRITE	无视内存的对齐写入
OVERWRITE_ALLOCATION	向分配的内存区域外写入
WRITE_AFTER_FREE	释放保留的内存后, (先等到进程调度程序运行) 写入数据
SOFTLOCKUP	在可中断状态下执行无限循环
HARDLOCKUP	在禁止中断状态下执行无限循环
HUNG_TASK	使当前进程进入不可中断状态, 调用进程调度程序

除这些选项以外, 还有下列追加选项 (见表 7-9)。

表 7-9 其他选项

选 项	说 明
recur_count	发生栈溢出时的递归调用次数 (默认为 10 次)
cpoint_count	经过崩溃位置多少次会发生崩溃 (默认为 10 次)

通过组合指定上述选项, 就可以造成需要的崩溃。例如, 在下例中, 如果从文件系统进行 20 次 I/O, 就会产生例外 (访问 NULL 指针)。

```
# modprobe lkdtm cpoint_name=FS_DEVRW cpoint_type=EXCEPTION cpoint_count=20
```

稍微过一段时间内核就会出现重大故障。也可以通过 debugfs 来指定崩溃位置和崩溃原

因。如果在不指定 `cpoint_name` 和 `cpoint_type` 的情况下将 `lkdtm` 安装到内核中，就会出现 `/sys/kernel/debug/provoke-crash` 目录。这个目录下存在表示崩溃位置的特殊文件，如果表 7-8 中表示崩溃原因的字符串写入这些文件中的某一个，就会发生相应的内核崩溃。下例通过 `debugfs` 进行与前面同样的操作。

```
# modprobe lkdtm cpoint_count=20
# cd /sys/kernel/debug/provoke-crash
# ls
DIRECT      IDE_CORE_CP      INT_HW_IRQ_EN      MEM_SWAPOUT      TIMERADD
FS_DEVRW    INT_HARDWARE_ENTRY  INT_TASKLET_ENTRY  SCSI_DISPATCH_CMD
# echo EXCEPTION > FS_DEVRW
```

## 小结

本节介绍了使用 LKDTM 功能在内核中造成崩溃的方法。一般情况下，必须避免内核崩溃，但在想要检查系统品质和故障时，LKDTM 却是非常有效的功能。尤其是在 Kdump 和集群系统（cluster system）的测试中更能发挥作用。

——Masami Hiramatsu

# HACK #56 IPMI 看门狗计时器

本节使用 IPMI 看门狗计时器可以检查出操作系统死机。

## IPMI 看门狗计时器

IPMI 看门狗计时器是 Intelligent Platform Management Interface (IPMI) 标准中使用硬件的看门狗计时器。系统死机时可以执行机器自身的重启 (reset) 等，从而可以提高系统的可用性。

IPMI 是数家电脑相关厂商制定的标准，为了获取电脑各部位的温度、电压、风扇等状态以及控制电源等而规定的接口。其中就包括本节要介绍的看门狗计时器的标准。

它与 NMI 看门狗计时器（参考 Hack #57）的不同之处在于可以通过硬件 (IPMI) 执行硬启动 (hard reset)。IPMI 与 CPU 是相互独立的，因此即使硬件出现问题也有可能恢复 (硬启动)。必须在服务器上安装有 IPMI 才能使用它。

---

**小贴士：**笔者经常遇到的硬件故障是执行 `shutdown` 或 `reboot` 命令，界面上输出 `Power down.` 或 `Restarting system.` 等最后的信息，机器却依然处于停止状态。CPU 中应当执行了 `shutdown` 或 `reboot` 命令，但由于某些原因导致未执行重启。

---

这种情况在非正式硬件（产品版）尤其是评估版的试验机上比较多见。这种情况下可以设置后面要介绍的 `nowayout` 参数，这样即使 CPU 未执行重启命令，IPMI 也会进行硬启动。

---

## IPMI 看门狗计时器的使用方法

要使用 IPMI 看门狗计时器，需要先启动将内核配置为 CONFIG\_IPMI\_WATCHDOG=y 的内核。在 RHEL6 中，接下来需要对配置文件 /etc/sysconfig/ipmi 进行如下编辑。

```
# cat /etc/sysconfig/ipmi
.....
IPMI_WATCHDOG=yes
.....
IPMI_WATCHDOG_OPTIONS="timeout=60 action=reset pretimeout=30 preaction=pre_int
preop=preop_panic"
.....
```

表 7-10 ~ 表 7-13 分别表示各参数的意义。图 7-1 所示为 timeout、pretimeout、action、preaction 的关系。

表 7-10 指定 timeout 的各参数时的处理

参    数	说    明
timeout	到超时为止的时间 (单位: 秒)
pretimeout	到执行超时运行为止的时间与到执行预超时 (pretimeout) 运行为止的时间差 (单位: 秒)
action	超时的处理 (reset、none、power_cycle、power_off)
preaction	预超时的处理 (pre_none、pre_smi、pre_nmi、pre_int)
preop	预超时驱动程序的运行 (preop_none、preop_panic)
start_now	设置为 1 时, ipmi_watchdog 模块安装到内核后立刻启动计时器。内核启动后立刻就可以通过看门狗计时器监视系统
nowayout	设置为 1 时, 一旦打开 IPMI 看门狗计时器, 就不会停止

表 7-11 指定 action 的各参数时的处理

参    数	说    明
none	不作任何处理
reset	重启系统 (默认)
power_cycle	切断电源后重新接入
power_off	切断电源

表 7-12 指定 preaction 的各参数时的处理

参    数	说    明
pre_none	不作任何处理 (默认)
pre_smi	通过 SMI (System Management Interface) 通知
pre_int	使用中断向 IPMI 驱动程序通知信息
pre_nmi	使用 NMI 中断通知信息 (RHEL5 系列中不能使用)

表 7-13 指定 preop 的各参数时的处理

参数	说 明
preop_none	不作任何处理（默认）
preop_panic	使发生内核重大故障

时间

看门狗计时器的最后更新

执行preaction

执行action

pretimeout

timeout

图 7-1 timeout、pretimeout、action、preaction 的关系

设置 nowayout 参数后，就可以根据计时器不停止的情况，在 reboot/shutdown 命令执行后期检测出 watchdog 守护进程结束后的死机。

指定 pre\_nmi 参数后，会通过 NMI 通知 IPMI 看门狗计时器超时，因此在禁止中断状态下 preaction、preop 也会运行。pre\_nmi 在 RHEL5 系列中不能使用。可以在上游内核、RHEL6 中使用。

在 /etc/sysconfig/ipmi 中进行设置后，启动 ipmi 服务，安装 IPMI 模块。

启动 ipmi 服务，安装 IPMI 模块。

```
# service ipmi start
# lsmod
Module           Size  Used by
ipmi_watchdog    53344  0
ipmi_devintf     44944  0
ipmi_si          77644  1
ipmi_msghandler  71768  3 ipmi_watchdog,ipmi_devintf,ipmi_si
....
```

在 RHEL6 中启动的是 watchdog 守护进程。

```
# service watchdog start
```

watchdog 守护进程的设置使用 /etc/watchdog.conf 来进行。设置如下。

```
# vi /etc/watchdog.conf
.....
watchdog-device = /dev/watchdog
....
```

启动 watchdog 守护进程，就会启动 IPMI 看门狗计时器。watchdog 守护进程定期对 IPMI 看门狗计时器进行重启（实时处理）。IPMI 看门狗计时器的结构如图 7-2 所示。

系统正常运行时，调度 watchdog 守护进程，因此实时处理继续进行，但系统如果死机，watchdog 守护进程作出的实时处理就会中断。这样计时器就不重启，而 IPMI 看门狗计

时器超时。此时的运行情况如图 7-3 所示。

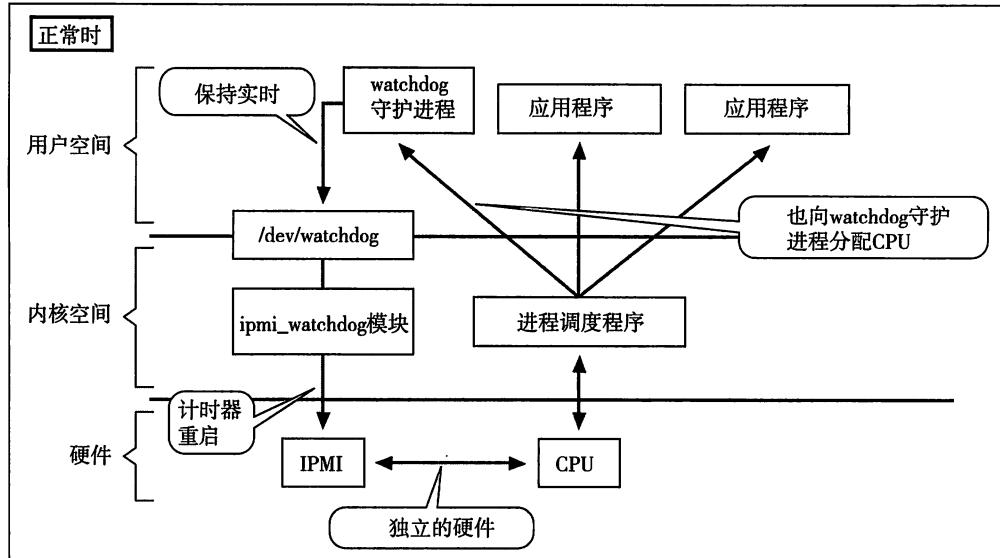


图 7-2 正常时的 IPMI 看门狗计时器

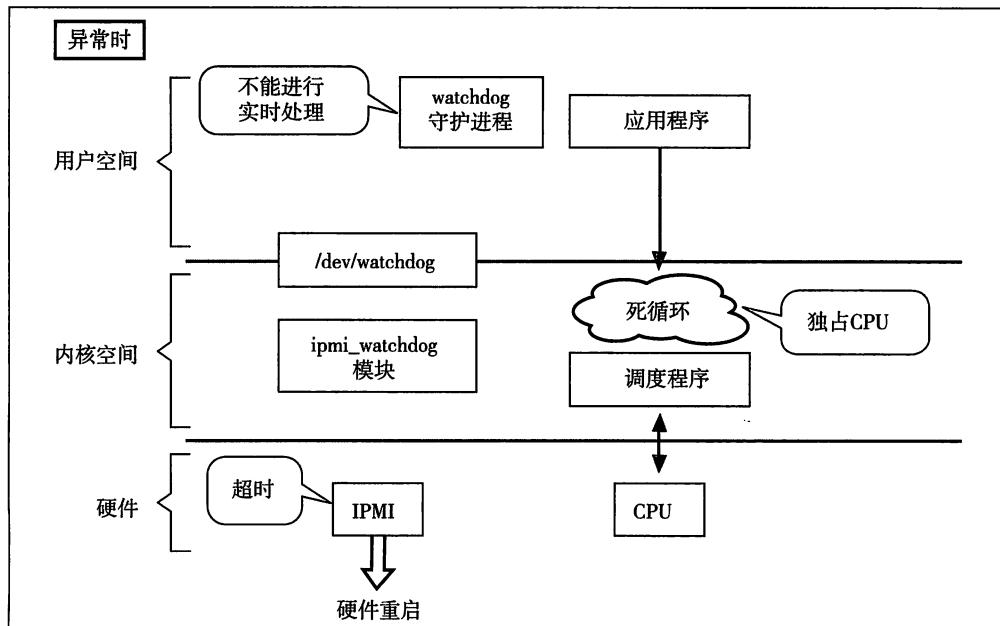


图 7-3 异常时的 IPMI 看门狗计时器

设置为 `start_now=1` 时，把 `ipmi_watchdog` 模块安装到内核时 IPMI 看门狗计时器就会开始运行，因此需要选择使 `watchdog` 守护进程启动前不会超时的 `pretimeout/timeout` 时间。

## 设置示例

举个一般的例子，向 pretimeout 设置小于 timeout 的值，向 preaction 设置 pre\_nmi 或 pre\_int。将 timeout 设置为 90、pretimeout 设置为 30 时，IPMI 看门狗计时器最后一次更新后的 60 秒内没有再次进行更新（系统已死机）时，执行 preaction 指定的操作。

另外，向 preaction 指定 pre\_int，向 preop 指定 preop\_panic，也可以获取转储。这种情况下，超过预超时时间就会通知 ipmi\_watchdog 驱动程序，发生重大故障。

操作系统无法进行重大故障处理的情况下，接下来就会发生超时。这将通过 IPMI 执行 action。action 在不通知软件的情况下由硬件执行。

## 确认运行

向 /dev/watchdog 写入字符串，就可以确认 IPMI 看门狗计时器的运行。写入 V 则禁用 IPMI 看门狗计时器。

使用 echo 命令执行时，写入时需要按照下面这样不加入换行符。

```
# echo -n V > /dev/watchdog
```

按照下列方式将其他字符串（在该示例中为 \0）写入 /dev/watchdog，文件就会关闭，通过实时处理进行的计时器重启过程就会停止。

```
# echo -n "\0" > /dev/watchdog
# dmesg
....
```

IPMI Watchdog: Unexpected close, not stopping watchdog!

计时器不会重启，因此假设设置为 timeout=60 action=reset 时，执行这条命令后 60 秒系统就会重启。

由于是从硬件执行的，因此重启时不会输出信息等，但 IPMI 会在 System Event Log (SEL) 中记录简单的日志。重新启动后可以使用下列命令确认通过 IPMI 看门狗计时器进行的系统重启。

```
# ipmitool sel list
 1 | 08/23/2007 | 00:21:59 | Event Logging Disabled #0x51 | Log area reset/
cleared | Asserted
.....
 1b | 03/10/2011 | 12:03:12 | Watchdog 2 #0x50 | Timer expired | Asserted ... ①
 1c | 03/10/2011 | 12:11:59 | Watchdog 2 #0x50 | Hard reset | Asserted ... ②
 1d | 03/17/2011 | 15:05:28 | Watchdog 2 #0x50 | Power down | Asserted ... ③
```

例如，①是在执行 echo -n V > /dev/watchdog 时记录的。②是在 action=reset 时执行重启时记录的日志。③是执行 action 的 power\_off 时的记录。

## NMI 看门狗计时器和 pre\_nmi

设置了 pre\_nmi 时，如果死机发生的时间超过 pretimeout 所指定的时间，就捕获主板上的 NMI 信号，发生 NMI 中断。内核上的 IPMI 看门狗计时器使 NMI 处理程序启动，如果设置了 preop\_panic，则发生重大故障。如果进行了 Kdump 等的设置，就可以获取转储。即使 IPMI 看门狗计时器使用 pre\_nmi，通过 NMI 通知的内存错误、I/O 错误也会正常运行。

在使用 NMI 看门狗计时器的情况下，即使向 preaction 设置 pre\_nmi，也会输出下列信息，无法使用。这时 pretimeout 不运行，只有 timeout 的 action 运行。

```
# dmesg
.....
IPMI Watchdog: IPMI NMI didn't seem to occur. The NMI pretimeout will likely
not work
IPMI Watchdog: driver initialized
```

## 其他看门狗计时器

在 RHEL6 等中也可以使用各厂商的主板上安装的看门狗计时器。Intel TCO (Total Cost of Ownership) 看门狗计时器就是其中之一。这是 ICH (I/O Controller Hub, I/O 控制器集线器) 的一个功能。这个看门狗计时器不使用 IPMI，因此机器中不需要安装 IPMI。

Intel TCO 看门狗驱动程序的内核模块名称为 iTCO\_wdt。机器能够支持该内核模块时它会自动安装到内核中。使用 dmesg 命令时出现下列信息就表示该内核模块已经安装到内核中。

```
# dmesg
.....
iTCO_vendor_support: vendor-support=0
iTCO_wdt: Intel TCO WatchDog Timer Driver v1.05
iTCO_wdt: Found a ICH7 or ICH7R TCO device (Version=2, TCOBASE=0x0860)
iTCO_wdt: initialized. heartbeat=30 sec (nowayout=0)
.....
# lsmod | grep iTCO
iTCO_wdt            49232  0
iTCO_vendor_support 37124  1 iTCO_wdt
```

Intel TCO 看门狗计时器的功能没有 IPMI 看门狗计时器那么多。只是在超时（默认为 30 秒）时内核重新启动。iTCO\_wdt 也是通过 watchdog 守护进程启动计时器。

iTCO\_wdt 驱动程序也是使用 /dev/watchdog。向 /dev/watchdog 写入 V 或 \0 时的情况与 IPMI 看门狗计时器相同。

/dev/watchdog 不能被多个内核模块同时使用。但是如果机器上安装了 IPMI，则对于 IPMI 看门狗计时器的 ipmi\_watchdog 模块和 iTCO\_wdt 模块，先安装到内核的驱动程序将使用 /dev/watchdog，后安装到内核的看门狗计时器不能使用。

要使用 IPMI 看门狗计时器，但是先安装到内核的是 iTCO\_wdt，需要通过如下方式禁

止 iTCO\_wdt 的安装。在 /etc/modprobe.d/blacklist.conf 中添加下列内容。

```
blacklist iTCO_wdt
```

或者在 /etc/modprobe.d/dist.conf 中添加下列内容。

```
install iTCO_wdt /bin/true
```

## 参考文献

- The Linux IPMI Driver  
Documentation/IPMI.txt
- 《Debug Hacks》“使用 IPMI 看门狗在死机时获取崩溃转储” [Hack #22]

——Naohiro Ooiwa

# HACK #57 NMI 看门狗计时器

本节使用 x86\_64 或 i386 架构的 NMI 看门狗计时器功能，可以检测出系统死机。

## NMI 看门狗计时器

顾名思义，NMI 看门狗计时器是使用 NMI 的看门狗计时器。可以检测出 PC 在禁止中断状态下的死机。

NMI 是 Non Maskable Interrupt 的缩写，表示无法禁止的中断。因此内核在禁止中断状态下也可以接收到 NMI 的通知。

中断是从硬件发出的信号，I/O 的完成或数据包的接收通过中断传递到内核。内核接收到这个中断时会进行各种处理，但如果进行并行处理就会出现数据不匹配，因此在接收到 1 个中断时，就会禁止此后的中断直到这个中断的处理完成。

NMI 并不是用来处理 I/O 或数据包等一般数据，而是用来通知紧急情况的，因此进行了无法禁止中断的特别处理。

内存的奇偶校验错误等系统致命错误就是需要通知的紧急情况。最近出现了定期发生 NMI 的功能。NMI 看门狗计时器就是使用这个功能的看门狗计时器。通过 NMI 看门狗计时器检测出系统死机。另外在检测出死机时还可以获取内核崩溃转储。

内核使用 NMI 看门狗计时器检测死机的结构如下（如图 7-4 所示）。一般来说，计时器中断在 1 秒钟内发生的次数为内核配置中设置的次数（100 ~ 1000 次）。

但是，例如，在禁止中断的状态下，内核进入无限循环或死锁时，就不会执行计时器中断处理。另一方面，NMI 即使在这种情况下也会发生，CPU 执行 NMI 处理程序。NMI 处理程序监视是否执行了计时器中断，如果执行了计时器中断，就会超时，认为系统已死机，如图 7-5 所示。

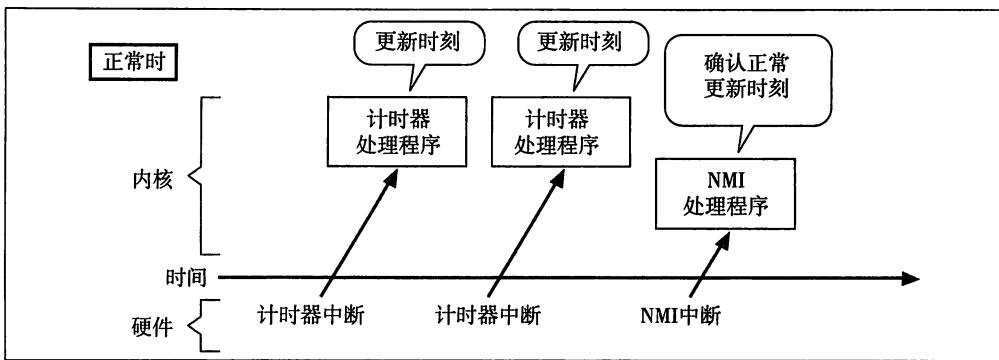


图 7-4 内核检测死机的结构

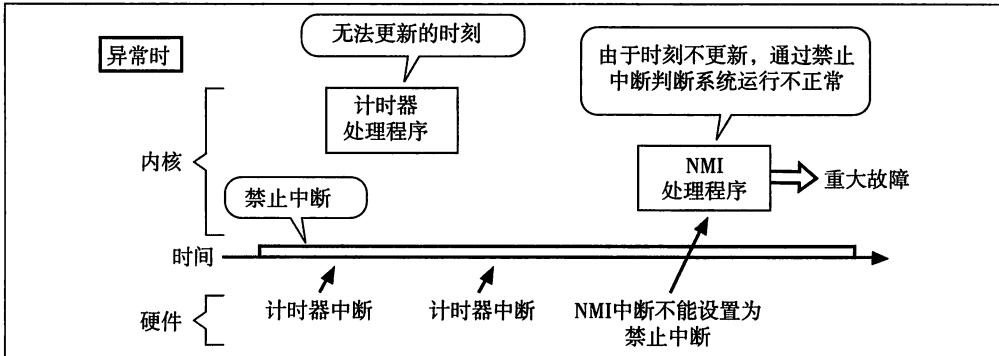


图 7-5 检测出死机时

RHEL6 中的超时时间默认为 30 秒。上游内核中为 5 秒。

## NMI 看门狗计时器的使用方法

使用 NMI 看门狗计时器前需要将内核启动参数进行如下设置。

```
nmi_watchdog=N
```

N 可以写入 0~2 的值。0 用来禁用 NMI 看门狗。如果是具有 IO-APIC 的机器则设置为 1 (I/O-APIC 模式)；如果不具有 IO-APIC 则设置为 2 (本地 APIC 模式)。

安装了 Local APIC NMI 的机器上将输出如①所示的信息。此时如果设置为 nmi\_watchdog=2，就会输出②，NMI 看门狗计时器运行。

```
# dmesg
...
ACPI: LAPIC_NMI (acpi_id[0xff] high level lint[0x1]) ... ①
...
Testing NMI watchdog ... OK. ... ②
```

如果在 Local APIC 的机器上设置为 nmi\_watchdog=1，则在 RHEL6 的情况下就会登录虚拟的 APIC 计时器，执行 NMI 看门狗计时器。

```
# dmesg
...
APIC timer registered as dummy, due to nmi_watchdog=1!
...
Testing NMI watchdog ... OK.
```

NMI 看门狗计时器超时，就是指系统由于某些故障死机。想要找出原因时，可以在 NMI 看门狗计时器超时时提取内核崩溃转储。设置如下。

`nmi_watchdog=panic,N` (`N` 为 1 或 2)

必须事先进行 Kdump (Hack #54) 或 diskdump (Hack #53) 的设置。在内核运行过程中，禁用 NMI 看门狗计时器时，可以将 `/proc/sys/kernel/nmi_watchdog` 设置为 0。

## 关于 NMI 的其他参数

`sysctl` 中还有其他关于 NMI 的参数，下面就将进行介绍。

`/proc/sys/kernel/unknown_nmi_panic`

如果将这个参数设置为 0 以外的值，就会在发生不明情况的 NMI 时造成重大故障<sup>译注1</sup>。当服务器上有 NMI 键时，需要事先将 `unknown_nmi_panic` 设置为 0 以外的值。当系统死机时，有时按 NMI 键就可以获取转储。

`/proc/sys/kernel/panic_on_unrecoverable_nmi`

如果将这个参数设置为 0 以外的值，就会在发生内存奇偶校验错误或不明情况的 NMI 时造成重大故障。即使 `unknown_nmi_panic` 为 0，如果将 `panic_on_unrecoverable_nmi` 设置为 1 等，也会在发生不明情况的 NMI 时造成重大故障。不明情况的错误有内存故障或 PCI 总线的错误等。设置了 EDAC (Error Detection And Correction，错误检测和校正) 时不会造成重大故障，而是由 EDAC 运行。

`/proc/sys/kernel/panic_on_io_nmi`

如果将这个参数设置为 0 以外的值，就会在因 I/O 错误而发生 NMI 时造成重大故障。

——Naohiro Ooiwa

## HACK #58 soft lockup

本节介绍 Linux 内核中安装的锁定检测功能 `soft lockup`。

Linux 内核中安装了称为 `soft lockup` 的锁定检测功能。使用 `soft lockup`，可以检测出由于内核或设备驱动程序等软件的故障导致内核功能无法运行的状态。

在关键任务 (critical mission) 等领域，如果一直处于服务停止的状态，就会造成很大的问题。因此，就需要使用这样的锁定检测功能，尽早检测出服务停止的情况。在检测锁定的同时使内核出现重大故障，就可以更简单地自动重启内核以恢复服务，或从外部检测出故障，尽快恢复服务。

---

译注 1：重大故障对应的英文为 `panic`。

## soft lockup 的结构

soft lockup 的锁定检测是使用名称为 watchdog 的内核线程进行的。watchdog 内核线程采用优先级最高的实时等级的 FIFO 调度策略。如果这个 watchdog 线程没有达到一定的运行时间，就可以检测出进程调度程序未正常运行，即发生锁定。

使用 ps 命令，可以确认 watchdog 内核线程是存在的，详情如下。

```
# ps aux|grep watchdog
root      5  0.0  0.0      0      0 ?        S    19:14   0:00 [watchdog/0]
root      8  0.0  0.0      0      0 ?        S    19:14   0:00 [watchdog/1]
root     11  0.0  0.0      0      0 ?        S    19:14   0:00 [watchdog/2]
root     14  0.0  0.0      0      0 ?        S    19:14   0:00 [watchdog/3]
```

这些内核线程定期（检测时间的 20%）由计时器唤醒。唤醒的 watchdog 内核线程会记录自己唤醒的时刻。如果从这个 watchdog 记录的时刻开始，不更新的时间超过指定的时间，则判断发生了锁定。

## soft lockup 的设置

soft lockup 可以使用 sysctl 进行设置。

**softlockup\_thresh**

指定到检测出锁定为止的时间。单位为秒，初始值为 60 秒。

**softlockup\_panic**

选择检测出锁定时是否使内核出现重大故障。如果将值设置为 on (1)，则在检测出锁定时立刻发生内核重大故障。初始值为 off (0)。

例如，要将检测时间设置为 10 秒，检测出锁定时发生重大故障，需将 sysctl 进行如下设置。

```
# sysctl -w kernel.softlockup_thresh=10
# sysctl -w kernel.softlockup_panic=1
```

## soft lockup 的确认

接下来实际确认锁定的检测情况。本次使用的是 Fedora 14。

这里在初始化代码中生成执行死循环的内核模块。安装这个模块后，内核就会进行无限循环处理，进入锁定状态。

源文件和 Makefile 如下所示。

```
# cat Makefile
obj-m := lockup.o

# cat lockup.c
#include <linux/module.h>
```

```

static int lockup_init(void)
{
    for (;;)
        ;

    return 0;
}

static void lockup_exit(void)
{
}

```

module\_init(lockup\_init);  
module\_exit(lockup\_exit);

执行 make, 创建内核模块。

```
# make -C /lib/modules/2.6.35.11-83.fc14.x86_64/build M='pwd' modules
```

当前目录下应当生成了 lockup.ko。下面使用 insmod 命令将问题模块安装到内核中。

当执行 insmod 命令时, 内核就会因无限循环而进入锁定状态。约 60 秒后锁定检测出来, 就会输出如下所示的 BUG: soft lockup 之后的内容。

这个模块初始化无限循环不会结束, 因此需要在确认锁定之后重启。

```

# insmod lockup.ko
[ 1001.569249] lockup: module license 'unspecified' taints kernel.
[ 1001.571751] Disabling lock debugging due to kernel taint
[ 1066.226006] BUG: soft lockup - CPU#0 stuck for 61s! [insmod:1374]
[ 1066.226006] Modules linked in: lockup(P+) ip6t_REJECT nf_conntrack_ipv6
ip6table_filter ip6_tables ipv6 i2c_piix4 ppdev parport_pc parport virtio_net
i2c_core virtio_blk virtio_pci virtio_ring virtio [last unloaded: scsi_wait_
scan]
[ 1066.226006] CPU 0
[ 1066.226006] Modules linked in: lockup(P+) ip6t_REJECT nf_conntrack_ipv6
ip6table_filter ip6_tables ipv6 i2c_piix4 ppdev parport_pc parport virtio_net
i2c_core virtio_blk virtio_pci virtio_ring virtio [last unloaded: scsi_wait_
scan]
[ 1066.226006]
[ 1066.226006] Pid: 1374, comm: insmod Tainted: P 2.6.35.11-83.
fc14.x86_64 #1 /Bochs
[ 1066.226006] RIP: 0010:[<fffffffffa00fa009>]  [<fffffffffa00fa009>] lockup_
init+0x9/0xb [lockup]
[ 1066.226006] RSP: 0018:ffff880037b59f08  EFLAGS: 00000246
[ 1066.226006] RAX:  ffff880037b59fd8  RBX:  ffff880037b59f08  RCX:
0000000000000000
[ 1066.226006] RDX:  0000000000000001  RSI:  0000000000000000  RDI:
fffffffffa00fa000
[ 1066.226006] RBP:  ffffffff8100a68e  R08:  0000000000000000  R09:
ffff88003d51a128
[ 1066.226006] R10:  fffffffffa00fa280  R11:  0000000000000000  R12:
fffffffffffa00fa0050

```

```

[ 1066.226006] R13: ffffffa00fa280 R14: ffffff810c3200 R15:
fffff880037b59ea8
[ 1066.226006] FS: 00007fb3b696b720(0000) GS:ffff880002000000(0000)
knlGS:0000000000000000
[ 1066.226006] CS: 0010 DS: 0000 ES: 0000 CR0: 00000008005003b
[ 1066.226006] CR2: 000000000e58050 CR3: 00000003bbe9000 CR4:
0000000000006f0
[ 1066.226006] DR0: 0000000000000000 DR1: 0000000000000000 DR2:
0000000000000000
[ 1066.226006] DR3: 0000000000000000 DR6: 0000000fffff0ff0 DR7:
0000000000000400
[ 1066.226006] Process insmod (pid: 1374, threadinfo ffff880037b58000, task
ffff88003b961740)
[ 1066.226006] Stack:
[ 1066.226006] ffff880037b59f38 ffffff810021a1 0000000000164e3
fffffa00fa050
[ 1066.226006] <0> 0000000000000000 0000000001d0e030 ffff880037b59f78
ffff8107cce1
[ 1066.226006] <0> 0000000001d0e010 00000000000164e3 00000000000164e3
00007fffedd1fe69
[ 1066.226006] Call Trace:
[ 1066.226006] [<ffff810021a1>] ? do_one_initcall+0x5e/0x155
[ 1066.226006] [<ffff8107cce1>] ? sys_init_module+0xa6/0x1e4
[ 1066.226006] [<ffff81009cf2>] ? system_call_fastpath+0x16/0x1b
[ 1066.226006] Code: <eb> fe 55 48 89 e5 0f 1f 44 00 00 c9 c3 90 90 04 00 00
00 14 00 00
[ 1066.226006] Call Trace:
[ 1066.226006] [<ffff810021a1>] ? do_one_initcall+0x5e/0x155
[ 1066.226006] [<ffff8107cce1>] ? sys_init_module+0xa6/0x1e4
[ 1066.226006] [<ffff81009cf2>] ? system_call_fastpath+0x16/0x1b

```

此外，也可以应对像下面这样适用实时等级的 FIFO 调度策略的进程持续占用 CPU 的情况。而 Fedora 14 在标准设置中对于实时等级的 CPU 时间不加限制，因此无法直接检测出锁定。这里设置为 `sched_rt_runtime_us=-1`，来解除对实时级进程的 CPU 时间的限制。

```

# cat lockup.c
#include <sched.h>

int main(int argc, char **argv)
{
    struct sched_param p = { .sched_priority = 99 };

    sched_setscheduler(0, SCHED_FIFO, &p);

    for (;;)
        ;

    return 0;
}
-----
# gcc -g -O2 -o lockup lockup.c

```

```

# sysctl -w kernel.sched_rt_runtime_us=-1
kernel.sched_rt_runtime_us = -1
# ./lockup
[ 638.877007] BUG: soft lockup - CPU#0 stuck for 61s! [lockup:1204]
[ 638.877007] Modules linked in: ip6t_REJECT nf_conntrack_ipv6 ip6table_filter
ip6_tables ipv6 ppdev parport_pc parport i2c_piix4 virtio_net i2c_core virtio_
blk virtio_pci virtio_ring virtio [last unloaded: scsi_wait_scan]
[ 638.877007] CPU 0
[ 638.877007] Modules linked in: ip6t_REJECT nf_conntrack_ipv6 ip6table_filter
ip6_tables ipv6 ppdev parport_pc parport i2c_piix4 virtio_net i2c_core virtio_
blk virtio_pci virtio_ring virtio [last unloaded: scsi_wait_scan]
[ 638.877007]
[ 638.877007] Pid: 1204, comm: lockup Not tainted 2.6.35.11-83.fc14.x86_64 #1
/Bochs
[ 638.877007] RIP: 0033:[<00000000004004fa>]  [<00000000004004fa>] 0x4004fa
[ 638.877007] RSP: 002b:00007fff4e495fb0  EFLAGS: 00000217
[ 638.877007] RAX: 0000000000000000 RBX: 0000000000000000 RCX: 00007eff7c9395a7
[ 638.877007] RDX: 00007fff4e495fb0 RSI: 0000000000000001 RDI: 0000000000000000
[ 638.877007] RBP: ffffff8100a68e R08: 00007eff7cc075e0 R09: 00007eff7cc1b230
[ 638.877007] R10: 00007fff4e495d20 R11: 0000000000000202 R12: 00007fff4e4960a0
[ 638.877007] R13: 00000000004003f0 R14: 0000000000000000 R15: 00000000004003f0
[ 638.877007] FS:  00007eff7ce22720(0000) GS:ffff880002000000(0000)
knlGS:0000000000000000
[ 638.877007] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 638.877007] CR2: 00007eff7c9395a0 CR3: 000000003750c000 CR4: 00000000000006f0
[ 638.877007] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[ 638.877007] DR3: 0000000000000000 DR6: 00000000ffff0ff0 DR7: 0000000000000400
[ 638.877007] Process lockup (pid: 1204, threadinfo ffff88003dc38000, task
ffff88003b961740)
[ 638.877007]
[ 638.877007] Call Trace:
```

## 锁定检测的限制

前面对内核 soft lockup 功能的锁定检测进行了确认，但是该功能的结构不能检测出禁止中断状态下的锁定。这是因为 soft lockup 的检测是通过计时器中断的延长来进行的。禁止中断状态下的锁定检测可以使用 NMI 看门狗计时器 Hack #57 或 IPMI 看门狗计时器 Hack #56。

为了验证 soft lockup 功能是无法检测的，这里使用测试用内核模块 (hardlockup.c) 在禁止中断状态下进行无限循环。

```

# cat hardlockup.c
#include <linux/module.h>
#include <linux/irqflags.h>

static int lockup_init(void)
{
    local_irq_disable();

    for (;;)
```

```
        ;  
  
    return 0;  
}  
  
static void lockup_exit(void)  
{  
}  
  
module_init(lockup_init);  
module_exit(lockup_exit);
```

## 避免 soft lockup 的错误检测

在自己生成的 Linux 内核的驱动程序或模块中，soft lockup 功能有时会出问题。当要实现的功能需要长时间的 CPU 运行时，soft lockup 功能有可能错误地识别锁定（或死档）。在这种情况下，可以从相应的功能中调用 `_touch_watchdog()` 函数，在不禁用 soft lockup 功能的情况下避免错误检测。

此外，也可以向内核启动参数传递 `nosoftlockup` 来禁用 soft lockup 功能。

## 小结

本节介绍了 Linux 内核中的锁定检测功能之一——soft lockup。使用 soft lockup，可以检测出软件 bug 引起的问题，并通过造成内核出现重大故障等自动进行恢复。在 Linux 应用中，为了不使服务由于内核的问题而一直处于停止状态，需要进行一些设置。

——Hiroshi Shimamoto

## HACK #59 crash 命令

本节介绍使用方便的 `crash` 命令和使用方法。

`crash` 命令是可以分析内核映像的调试工具。它在 `gdb` 命令的基础上设计，可以运行用来引用内核内部各种信息的众多命令，以会话形式进行操作。通过 `Kdump` 等转储机构提取的内核崩溃转储和运行中的实时内核就是作为分析对象的内核映像。本节将介绍便捷的 `crash` 命令和使用方法。

## 支持范围

本书写作时 `crash` 命令的最新版为 5.1.5。主要的 Linux 发布版都提供了工具包包。RHEL4 和 RHEL5 中的版本号为 `crash` 命令的 4 系列，RHEL6 中为 5 系列。

支持的架构如下。

- X86
- X86\_64
- LA64
- PPC64
- s390
- s390x
- ARM

可以处理通过下列方法提取的内核映像。

- 通过 `diskdump/netdump` 提取的内核崩溃转储 (RHEL4)。
- 通过 `Kdump` 提取的内核崩溃转储 (RHEL5、RHEL6)。
- 通过 `makedumpfile` 转换的内核崩溃转储 (RHEL5、RHEL6)。
- 通过 `virsh dump` 命令提取的 KVM 客户端的内核崩溃转储 (RHEL6)。
- 实时内核

---

**小贴士：**crash 的设计可以应对各种 Linux 内核的版本或发布版，以约 1 个月的较短周期进行更新。如果使用发布版的 crash 无法正常运行，可以先尝试论坛上的最新版。如果还是不能运行请向论坛报告。

---

## 安装与启动的方法

crash 命令的最新版可以从论坛的网页 (<http://people.redhat.com/anderson/>) 获得。有 tar 文件格式和 source rpm 数据包格式。tar 文件格式的创建方法如下。

```
$ wget -c "http://people.redhat.com/anderson/crash-5.1.5.tar.gz"
$ tar xf crash-5.1.5.tar.gz
$ cd ./crash-5.1.5
$ make
```

---

**小贴士：**从 5.0.7 版开始可以在 X86 上创建用于 ARM 的执行文件。在 X86 上创建的方法如下。

```
$ make target=ARM
```

同样，从 5.0.8 版开始可以在 X86\_64 上创建用于 X86 的执行文件。在 X86\_64 上创建的方法如下。

```
$ make target=X86
```

---

要使用 crash 命令，必须要有与内核映像对应的调试信息文件。调试信息文件就是内核创建时生成的名称为 `vmlinux` 的文件。在 RHEL 的工具包命名为 `kernel-debuginfo`。

启动 crash 的方法如下所示。`vmlinux` 为调试信息文件，`vmcore` 为内核崩溃转储文件。

```
$ crash vmlinux vmcore
```

分析实时内核时只需要第 1 个参数。

```
$ crash vmlinux
```

---

**小贴士：**在实时内核的情况下，crash 为了参照物理内存，使用 /dev/mem、/proc/kcore 以及 /dev/crash 中的某一个的内核接口。也可以将这些直接指定为第 2 个参数。但是有些内核无法使用 /dev/mem 和 /proc/kcore，这时使用 /dev/crash。/dev/crash 是为其他两个不能使用的情况而准备的 crash 特有驱动程序。省略第 2 个参数后，crash 就会在各个接口依次进行激活处理，选择可使用的接口。

---

**小贴士：**在 crash 5.1.3 以后的版本中，可以处理以 gzip 格式和 bzip2 格式压缩的 vmlinu。可以节约磁盘空间。

---

启动 crash 后，就会开始对话形式的命令输入。然后使用各种命令分析内核映像。

## 实用工具命令（utility command）

首先介绍使用 crash 时的实用工具命令。

### set 命令

set 命令是运用非常广泛的命令。指定进程后，就会将进程的上下文切换到指定的进程，显示这个上下文。没有指定时显示当前进程的上下文。进程上下文的初始值为内核发生重大故障时运行的进程的初始值。

```
crash> set
      PID: 1474
COMMAND: "sh"
  TASK: fffff88001c5c9740  [THREAD_INFO: fffff88001b76a000]
    CPU: 0
  STATE: TASK_RUNNING (PANIC)
```

使用 -c 选项可以指定 CPU，并显示之前在这个 CPU 上运行的进程。使用 -p 选项可以显示发生重大故障时运行的进程。

使用 set 命令可以确认当前的编辑器设置。

```
crash> set -v | grep edit
      edit: vi
```

要指定命令行编辑模式时，需要在 crash 启动时指定。

```
# crash -e [vi | emacs]...
```

设置为 emacs，就可以进行与 bash 相同的键操作。

crash 命令有时会输出大量的信息，可以使用下列命令禁用界面的 scroll 功能。

```
crash> set scroll off
```

此外使用下列命令也可以禁用界面的 scroll 功能。

```
crash> sf
```

sf 是通过别名定义设置的，是 set scroll off 的别名。

默认的别名定义可以使用 `alias` 来确认。

```
crash> alias
ORIGIN ALIAS COMMAND
builtin man help
builtin ? help
builtin quit q
builtin sf set scroll off
builtin sn set scroll on
builtin hex set radix 16
builtin dec set radix 10
builtin g gdb
builtin px p -x
builtin pd p -d
builtin for foreach
builtin size *
builtin dmesg log
builtin lsmod mod
builtin last ps -l
```

## eval 命令

`eval` 命令将输入的表达式的计算结果以十六进制、十进制、八进制以及二进制显示。指定 `-b` 选项就会显示在二进制中第几位是 1。表达式可以使用四则运算 (`-`、`+`、`*`、`/`、`%`)、位运算 (`&`、`|`、`^`)、shift 运算 (`<<`、`>>`)。

## ascii 命令

ascii 命令将十六进制转换为字符串。

```
crash> rd modprobe_path 2
ffffffff813213a0: 6f6d2f6e6962732f 000065626f727064  /sbin/modprobe..
crash> ascii 6f6d2f6e6962732f
6f6d2f6e6962732f: /sbin/mo
```

## history 命令

history 命令显示已输入命令的记录。也可以使用别名 h。

```
crash> history
[1] set
[2] set vi
[3] set scroll off
crash> h
[1] set
[2] set vi
[3] set scroll off
```

## 参照内核信息的命令

这里介绍用来获取内核内部信息的命令。

### bt 命令

bt 命令输出进程的内核栈的遍历。它是使用频率较高的命令。要显示所有进程的遍历，可以使用下列 foreach 命令。

```
crash> foreach bt -tf
```

当栈上存在具有内核文本符号 (text symbol) 的地址时，-t 选项显示这些符号。在一般的回溯处理失败时可以使用该选项。

```
crash> bt 839
PID: 839      TASK: fffff88001ca82e80  CPU: 2  COMMAND: "rsyslogd"
#0 [fffff88001d8fdd58] schedule at ffffffff8146888f
#1 [fffff88001d8fde20] do_syslog at ffffffff8104e167
#2 [fffff88001d8fdea0] kmsg_read at ffffffff81169c80
#3 [fffff88001d8fdec0] proc_reg_read at ffffffff81160cac
#4 [fffff88001d8fdf00] vfs_read at ffffffff8111750d
#5 [fffff88001d8fdf40] sys_read at ffffffff811175ab
...
.

crash> bt -t 839
PID: 839      TASK: fffff88001ca82e80  CPU: 2  COMMAND: "rsyslogd"
      START: schedule at ffffffff8146888f
[fffff88001d8fde20] do_syslog at ffffffff8104e167
[fffff88001d8fde48] autoremove_wake_function at ffffffff81066633
[fffff88001d8fde60] pvclock_clocksource_read at ffffffff8102c275
[fffff88001d8fdea0] kmsg_read at ffffffff81169c80
[fffff88001d8fdec0] proc_reg_read at ffffffff81160cac
[fffff88001d8fdf00] vfs_read at ffffffff8111750d
[fffff88001d8fdf40] sys_read at ffffffff811175ab
...
.
```

-f 选项显示框架 (frame) 内的所有栈数据。这个选项可以在确认函数的参数时使用。

-l 选项显示文件名和行数。

```
crash> bt -l 839
```

```
PID: 839      TASK: ffff88001ca82e80  CPU: 2      COMMAND: "rsyslogd"
#0 [ffff88001d8fdd58] schedule at ffffffff8146888f
    /usr/src/debug/kernel-2.6.35.fc14/linux-2.6.35.x86_64/kernel/sched.c: 2820
#1 [ffff88001d8fde20] do_syslog at ffffffff8104e167
    /usr/src/debug/kernel-2.6.35.fc14/linux-2.6.35.x86_64/kernel/printk.c: 289
#2 [ffff88001d8fdea0] kmsg_read at ffffffff81169c80
    /usr/src/debug/kernel-2.6.35.fc14/linux-2.6.35.x86_64/fs/proc/kmsg.c: 39
#3 [ffff88001d8fdec0] proc_reg_read at ffffffff81160cac
    /usr/src/debug/kernel-2.6.35.fc14/linux-2.6.35.x86_64/fs/proc/inode.c: 165
#4 [ffff88001d8fdf00] vfs_read at ffffffff8111750d
    /usr/src/debug/kernel-2.6.35.fc14/linux-2.6.35.x86_64/fs/read_write.c: 310
#5 [ffff88001d8fdf40] sys_read at ffffffff811175ab
    /usr/src/debug/kernel-2.6.35.fc14/linux-2.6.35.x86_64/fs/read_write.c: 388
...
...
```

-a 选项仅显示各 CPU 上原本正在运行的进程。但是针对运行中的系统（实时内核）启动 crash 命令时，不支持这个选项。

```
crash> bt -a
bt: -a option not supported on a live system
```

---

小贴士：从 5.1.1 版开始，当指定 -t 时，即使是活动系统也会运行。

---

使用 task 命令获取栈指针，通过 rd -s 引用栈区域，就可以得到接近 bt -t 的信息。

```
crash> task 839 | grep sp
    sp0 = 0xffff88001d8fe000,
    sp = 0xffff88001d8fdd58,
...
crash> rd 0xffff88001d8fdd58 -e 0xffff88001d8fe000 -s
...
ffff88001d8fde18: ffff88001d8fde98 do_syslog+0x10f
ffff88001d8fde28: ffff880000000004 ffffffff00000000
ffff88001d8fde38: 0000000000000000 ffff88001ca82e80
ffff88001d8fde48: autoremove_wake_function log_wait+0x8
ffff88001d8fde58: log_wait+0x8      pvclock_clocksource_read+0x48
ffff88001d8fde68: 00007f96fffffff 00007f969248b360
ffff88001d8fde78: 0000000000000fff ffff88001d8fdf58
ffff88001d8fde88: 0000000000000000 0000000000000003
ffff88001d8fde98: ffff88001d8fdeb8 kmsg_read+0x4d
ffff88001d8fdea8: ffff88001b455d80 ffff88001b6cc000
ffff88001d8fdeb8: ffff88001d8fdef8 proc_reg_read+0x73
ffff88001d8fdec8: ffff88001d8fdef8 ffff88001d8fdf58
ffff88001d8fded8: 0000000000000fff 00007f969248b360
ffff88001d8fdee8: ffff88001b6cc000 00007f969248b360
ffff88001d8fdef8: ffff88001d8fdf38 vfs_read+0xa9
ffff88001d8fdf08: ffff88001d8fdf18 ffff88001db55000
ffff88001d8fdf18: ffff88001d8fdf38 ffff88001b6cc000
ffff88001d8fdf28: 00007f969248b360 00007f968bfff9c0
ffff88001d8fdf38: ffff88001d8fdf78 sys_read+0x4a
...
...
```

## dev 命令

dev 命令显示字符设备（character device）的列表。使用 -p 选项显示 PCI 设备。它的输

出与 `lspci` 命令相同。使用 `-i` 选项显示 I/O 端口和 I/O 内存。内容基本与下列命令相同。

```
# cat /proc/ioports
# cat /proc/iomem
```

## dis 命令

`dis` 命令是进行反汇编的命令。如果指定 `-l` 选项，还会显示源代码名称和行编号。

```
crash> dis -l schedule
/usr/src/debug/kernel-2.6.35.fc14/linux-2.6.35.x86_64/kernel/sched.c: 3813
0xffffffff81468385 <schedule>: push    %rbp
0xffffffff81468386 <schedule+0x1>:    mov     %rsp,%rbp
0xffffffff81468389 <schedule+0x4>:    push    %r15
0xffffffff8146838b <schedule+0x6>:    push    %r14
0xffffffff8146838d <schedule+0x8>:    push    %r13
0xffffffff8146838f <schedule+0xa>:    push    %r12
0xffffffff81468391 <schedule+0xc>:    push    %rbx
...
```

## files 命令

`files` 命令显示进程打开的文件的信息。

```
crash> files 1474
PID: 1474    TASK: ffff88001c5c9740  CPU: 0    COMMAND: "sh"
ROOT: /    CWD: /home/hat
FD      FILE          DENTRY          INODE          TYPE  PATH
0  ffff88001bccd900  ffff88001eb23cc0  ffff88001edfbc30  CHR  /dev/tty1
1  ffff88001b655b40  ffff88001d215f00  ffff88001909d1b0  REG  /proc/sysrq-trigger
2  ffff88001bccd900  ffff88001eb23cc0  ffff88001edfbc30  CHR  /dev/tty1
10  ffff88001bccd900  ffff88001eb23cc0  ffff88001edfbc30  CHR  /dev/tty1
```

## irq 命令

`irq` 命令显示关于内核内部管理的中断的信息。

## kmem 命令

显示关于内核的内存使用情况的信息。`-s` 选项显示 slab 缓存 (slab cache) 的信息。信息与 `/proc/slabinfo` 相同。

```
crash> kmem -s
CACHE          NAME          OBJSIZE  ALLOCATED    TOTAL  SLABS  SSIZE
ffff88001f511840  rpc_inode_cache    816      19        19      1    16k
ffff88001f8ab540  ndisc_cache      320      17        24      2     4k
ffff880019ca66c0  UDPLITEv6       1000      0         0      0    16k
ffff880019ca6840  UDPv6          1000      48        48      3    16k
...
```

`-i` 选项显示内存的整体使用情况。等同于 `free` 命令。

```
crash> kmem -i
          PAGES      TOTAL      PERCENTAGE
```

TOTAL MEM	93411	364.9 MB	----
FREE	46292	180.8 MB	49% of TOTAL MEM
USED	47119	184.1 MB	50% of TOTAL MEM
SHARED	10469	40.9 MB	11% of TOTAL MEM
BUFFERS	5290	20.7 MB	5% of TOTAL MEM
CACHED	20584	80.4 MB	22% of TOTAL MEM
SLAB	9231	36.1 MB	9% of TOTAL MEM
TOTAL SWAP	507647	1.9 GB	----
SWAP USED	0	0	0% of TOTAL SWAP
SWAP FREE	507647	1.9 GB	100% of TOTAL SWAP

使用 -p 选项显示内存映射。也可以指定地址。[] 中的页面表示尚未释放内存。

```
crash> kmem ffff88001faf1cf0
CACHE          NAME          OBJSIZE  ALLOCATED  TOTAL  SLABS  SSIZE
ffff88001ec04780  vm_area_struct  184        3614    3630    165    4k
  SLAB          MEMORY          NODE      TOTAL  ALLOCATED  FREE
fffffea00006ee4b8  ffff88001faf1000  0        22      21      1
  FREE / [ALLOCATED]
  [ffff88001faf1cf0]

  PAGE      PHYSICAL      MAPPING      INDEX CNT FLAGS
fffffea00006ee4b8  1faf1000          0  ffff88001faf17e8  1 20000000000080
```

当引用某个内存发生重大故障时，如果使用 -p 选项，就可以确认这个内存是已释放还是仍在使用。

## list 命令

list 命令查找在内核中共同使用的 list\_head 结构的链接列表，依次显示列表上的对象 (object)。

```
crash> list modules
ffffffff81a59120
fffffffffa01e7598
...
ffffffffffa0007e08
ffffffffffa0000828
crash> list modules | wc -l
28
crash>
```

可以看出 modules 列表中连接了 28 个对象。下面是对象中间存在 list\_head 结构的成员的例子。这种情况下使用 -o 选项指定对象中的 list\_head 结构的位置，同样会查找列表。

```
crash> struct packet_type.list
struct packet_type {
  [0x40] struct list_head list;
}
```

可以得知 list\_head 结构的 offset 为 0x40，这可以在 -o 选项中指定。

```
crash> list -o 0x40 ip_packet_type
```

```
fffffffff81b881a0
fffffffff81b85ea0
fffffffff81b85ee0
fffffffff81b85f20
fffffffff81b88720
```

可以看出 `ip_packet_type` 的 `list` 成员连接了 5 个对象。

加上 `-s` 选项，这个对象的成员也会同时显示。下面是查找列表并显示各对象的 `func` 成员的例子。

```
crash> list -o 0x40 ip_packet_type -s packet_type.func
fffffffff81b881a0
    func = 0xfffffffff813f109b <ip_rcv>,
fffffffff81b85ea0
    func = 0xfffffffff81b85eb0,
fffffffff81b85ee0
    func = 0xfffffffff81b85ef0,
fffffffff81b85f20
    func = 0xfffffffff81b85f30,
fffffffff81b88720
    func = 0,
```

## mod 命令

`mod` 命令显示内核模块的信息，读入符号信息或调试信息。不指定选项时只显示内核模块信息。使用 `-s` 选项指定要读入的内核模块。使用 `-S` 选项指定目录，就会尝试从所指定的目录自动检索并读入模块。

```
crash> mod
      MODULE      NAME      SIZE  OBJECT FILE
fffffffffa0000d80  virtio_blk      5009  (not loaded)  [CONFIG_KALLSYMS]
fffffffffa003f900  ipv6          282108 (not loaded)  [CONFIG_KALLSYMS]
fffffffffa00514c0  ip6_tables    16850  (not loaded)  [CONFIG_KALLSYMS]
...
crash> mod -s virtio_blk /lib/modules/2.6.38.6-27.fc15.x86_64/kernel/drivers/
block/virtio_blk.ko
      MODULE      NAME      SIZE  OBJECT FILE
fffffffffa0000d80  virtio_blk      5009  /lib/modules/2.6.38.6-27.fc15.
x86_64/kernel/drivers/block/virtio_blk.ko
crash> mod -S
      MODULE      NAME      SIZE  OBJECT FILE
fffffffffa0000d80  virtio_blk      5009  /lib/modules/2.6.38.6-27.fc15.
x86_64/kernel/drivers/block/virtio_blk.ko
fffffffffa003f900  ipv6          282108 /lib/modules/2.6.38.6-27.fc15.
x86_64/kernel/net/ipv6/ipv6.ko
fffffffffa00514c0  ip6_tables    16850  /lib/modules/2.6.38.6-27.fc15.
x86_64/kernel/net/ipv6/netfilter/ip6_tables.ko
...
```

## net 命令

`net` 命令显示网络设备的信息列表。指定显示的 `net_device` 结构的地址，使用 `struct` 命令进行转储，就可以获取更加详细的信息。

```
crash> net
  NET_DEVICE      NAME      IP ADDRESS (ES)
ffffffffff8030f680  lo      127.0.0.1
ffff81007ea24000  eth2      192.168.0.155
ffff81007f7b8000  eth3      192.168.0.156
ffff81007ebbd000  eth0      172.16.0.153
ffff81007e1ff000  eth1
ffff81007d50a000  sit0
crash> struct net_device ffff81007ea24000
struct net_device {
  name = "eth2 $00045090668 $000 $000",
  name_hist={next=0x0,
  pprev=0xffffffff804bae90
},
...

```

## p 命令

向 gdb 的 print 命令传递输入时, p 命令可以显示结果。

```
crash> p init_mm
init_mm = $14 = {
  mmap = 0x0,
  mm_rb = {
    rb_node = 0x0
  },
  mmap_cache = 0x0,
  get_unmapped_area = 0,
...

```

输入为 CPU 个别符号 (percpu) 时, 显示各 CPU 个别区域的地址。

```
crash> p x86_bios_cpu_apicid
PER-CPU DATA TYPE:
  u16 x86_bios_cpu_apicid;
PER-CPU ADDRESSES:
[0]: ffff88003fc0dc10
[1]: ffff88003fd0dc10
```

## ps 命令

ps 命令显示进程信息。-a 选项显示赋予该进程的命令行变量和环境变量。

```
crash> ps -a rsyslogd
PID: 624      TASK: ffff880037b9dc80  CPU: 0      COMMAND: "rsyslogd"
ARG: /sbin/rsyslogd -n -c 5
ENV: PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
      LANG=ja_JP.UTF-8
      LISTEN_PID=624
      LISTEN_FDS=1
      SYSLOGD_OPTIONS=-c 5
```

-t 选项显示进程的运行时间、开始时刻、用户空间及内核空间上的执行时间。

```
crash> ps -t rsyslogd
```

```
PID: 624      TASK: fffff880037b9dc80  CPU: 0      COMMAND: "rsyslogd"  
  RUN TIME: 07:43:23  
  START TIME: 10  
    UTIME: 2  
    STIME: 13
```

## rd 命令

rd 命令是读取内存内容的命令。本节在 ascii 命令、bt 命令、wr 命令的解说中都使用过该命令。

## rung 命令

rung 命令显示连接到进程调度程序的运行队列（run queue）的进程（运行中的进程）。

## search 命令

search 命令检索内存空间中指定的值。

```
crash> search deadbeef  
fffff880011a43dd0: deadbeef  
...  
fffff88001620add0: deadbeef
```

使用 -c 选项也可以检索字符串。

```
crash> search -c "Linux version"  
fffff880001600050: Linux version 2.6.35.11-83.fc14.x86_64 (mockbuild@x86-01  
...  
fffffffff81cebcc7: Linux version 2.6.35.11-83.fc14.x86_64 (mockbuild@x86-01
```

## sig 命令

sig 命令显示进程的信号处理程序。另外，还会显示挂起（pending）的信号信息。-l 选项等同于 kill -l，显示已定义的信号编号。

## struct 命令

struct 命令根据结构的定义和实际的地址，配合结构显示数据。

```
crash> struct timespec xtime  
struct timespec {  
    tv_sec = 0x492ae39c,  
    tv_nsec = 0x25218473  
}
```

## swap 命令

swap 命令输出交换设备的信息。输出内容与 swapon -s 基本相同。

## sym 命令

sym 命令是解析符号 (symbol) 的命令。sym -l 与 cat System.map 相同。

## sys 命令

sys 命令显示系统的信息。显示时刻或 CPU 的平均负载 (load average)、描述出现重大故障原因的信息等。与 crash 命令启动时显示的信息相同。内核配置 CONFIG\_IKCONFIG 启用时，执行 sys config 就可以显示内核配置的列表。内容与 /proc/config.gz 解压缩后的文件相同。

```
crash> sys config
#
# Automatically generated make config: don't edit
# Linux/x86_64 2.6.39 Kernel Configuration
# Sun May 29 05:46:04 2011
#
# ...
CONFIG_64BIT=y
# CONFIG_X86_32 is not set
CONFIG_X86_64=y
CONFIG_X86=y
CONFIG_INSTRUCTION_DECODER=y
CONFIG_OUTPUT_FORMAT="elf64-x86-64"
CONFIG_ARCH_DEFCONFIG="arch/x86/configs/x86_64_defconfig"
...
```

对运行中的 (live) 系统运行 crash 命令时，可以使用 sys -panic 故意引起内核出现重大故障。运行结果与 echo c > /proc/sysrq-trigger 相同。

## task 命令

task 命令是显示 task\_struct 结构 (进程管理所用的数据结构) 的命令。

## timer 命令

timer 命令显示加入到计时器列表中的计时器。

## whatis 命令

whatis 命令显示结构等的类型或变量的定义。下面是显示全局变量 modules 的例子。可以看出 modules 是 list\_head 结构类型的变量。

```
crash> whatis modules
struct list_head modules;
```

## wr 命令

wr 命令是改写内存内容的命令。下例使用 crash 命令，在运行中的系统上改写表示时间的 jiffies 变量。改写后，表示从启动开始经过的时间的 UPTIME 增加了 3 天以上。

```
crash> sys
KERNEL: .../linux-2.6/vmlinux
```

```
DUMPFILE: /dev/mem
CPUS: 2
DATE: Sun May 29 05:26:13 2011
UPTIME: 00:01:57
...
crash> rd jiffies_64
ffffffff81b43880: 00000000fffd7f69
crash> wr jiffies_64 000000010ffd7f69
crash> sys
KERNEL: ./linux-2.6/vmlinux
DUMPFILE: /dev/mem
CPUS: 2
DATE: Sun May 29 05:26:57 2011
UPTIME: 3 days, 02:36:20
...
i.....
```

**小贴士：**/proc/kcore 和 /dev/crash 是只读的，因此要使用 wr 命令必须是 /dev/mem。2.6.27 版以后的内核在安全对策中引进了 CONFIG\_STRUCT\_DEVMEM，想要使用 wr 命令时，必须将这个选项设置为 off。

## 扩展命令

可以使用 crash 的库生成扩展模块，添加自己特有的命令。这里介绍 3 个方便的扩展命令。创建扩展模块时，可以在解压缩 crash 的目录下方，输入下列内容来进行。

```
$ make extensions
```

使用 extend 命令进行扩展命令的安装（load）和卸载（unload）。

```
crash> extend snap.so
./extensions/snap.so: shared object loaded
crash> extend -u snap.so
./extensions/snap.so: shared object unloaded
```

### snap 命令

使用 snap 命令，可以生成实时内核的内核崩溃转储。生成的内核崩溃转储文件也可以使用 crash 命令来分析的。

```
crash> snap vmcore
vmcore: [100%]
-rw-r--r-- 1 root root 536855288 2011-05-28 18:07 vmcore
```

### trace 命令

使用 trace 命令，可以得到内核映像中包含的 ftrace 的管理信息或者将其写出到文件。写出的文件可以使用 trace-cmd 命令来处理。除内核崩溃转储以外，在运行中的系统上也可以使用该命令。

trace 命令是由表 7-14 所示的子命令构成的。

表 7-14 trace 命令的子命令

名 称	功 能
trace	显示当前的追踪器 (tracer)。与 current_tracer 相同
trace show	显示追踪 (trace) 信息。与从 trace 读出的信息相同 trace-cmd 命令在内部调用
trace report	与 trace show 相同
trace dump < 目录名 >	根据 debugfs 的 tracing 目录下的结构, 将环形缓冲区 (ring buffer) 或管理信息写到文件
trace dump < 文件名 >	将环形缓冲区或管理信息生成 trace-cmd report 命令可以处理的数据格式的文件

详细内容请参考 HACK #67、HACK #69、HACK #70。

### gcore 命令

使用 gcore 命令可以生成内核崩溃转储中包括的用户模式进程的核心转储。详细内容请参考 HACK #61。

## crash 选项

下面介绍 crash 命令启动时的一些选项。

### -i 选项

向 -i 指定包含 crash 命令的文件, crash 就可以在启动时自动执行命令。使用这个选项可以使用 crash 进行自动处理。下面是执行 help 命令, 使用 exit 结束 crash 命令的示例。

```
$ cat crash_cmd.txt
help
exit
$ cat -s -i crash_cmd.txt
*      files      mod      runq      union
alias  foreach   mount   search   vm
ascii  fuser     net     set      vtop
bt     gdb       p       sig      waitq
bttop help     ps      struct   whatis
dev    irq      pte     swap     wr
dis    kmem    ptob    sym      q
eval   list     ptov    sys
exit   log      rd      task
extend mach    repeat  timer

crash version: 5.1.5      gdb version: 7.0
For help on any command above, enter "help <command>".
```

```
For help on input options, enter "help input".
For help on output options, enter "help output".
```

```
$ /* 显示 help 后返回 bash */
```

### -s 选项

像上例这样添加 -s 选项，就不再显示 crash 命令启动信息。

## 参考文献

- White Paper: Red Hat Crash Utility

[http://people.redhat.com/anderson/crash\\_whitepaper/](http://people.redhat.com/anderson/crash_whitepaper/)

——HATAYAMA Daisuke

## HACK #60 核心转储过滤器

本节介绍如何控制应用程序的 core 文件中包含的内存区域。

从内核 2.6.23 版本开始安装了核心转储过滤器 (core dump filter) 功能。

应用程序中一旦发生分段错误 (segment fault) 等，就会生成该进程的核心转储文件 (core 文件)。core 文件包括发生崩溃并强制结束的进程 (崩溃进程) 的内存区域。core 文件可以使用 gdb 命令等进行分析。

如果进程占用了大量内存，core 文件的大小也会相应地变大。另外，生成 core 文件也需要花费时间。可以使用 ulimit 来限制生成的 core 文件的大小，但如果进程内存的数据大小超过这个值，就无法全部获取数据。

---

**注意事项：**在 RHEL6 中，当 abrtd 运行时，根据 abrtd 的报告大小设置 (MaxCrashReportsSize)，核心文件有时不是在当前目录中生成，而是在 /var/spool/abrt 下生成。

---

使用核心转储过滤器限制 core 文件中包含的内存区域。例如，可以将共享内存的区域设置为不包括在 core 文件中。这样就可以缩小 core 文件的大小，并缩短获取 core 文件的时间。

## 使用方法

核心转储过滤器通过 /proc/<pid>/coredump\_filter 来设置。由于 coredump\_filter 是 /proc/<pid>/ 的文件，因此可以对每个进程分别设置。

coredump\_filter 为位掩码。表 7-15 所示为各值与指定的内存区域。

表 7-15 coredump\_filter 的位值与内存区域

位 值	内存区域
第 0 位	进程内存
第 1 位	共享内存
第 2 位	映射到文件的进程固有内存
第 3 位	映射到文件的进程共享内存
第 4 位	在文件的进程固有内存中, 仅 ELF 头文件部分的页面。在未指定第 2 位 (第 2 位为 0) 时有效
第 5 位	hugetlb 进程内存
第 6 位	hugetlb 共享内存

内存区域的设置是位掩码为 1 时包括, 为 0 时则不包括。

RHEL6 的默认为 0x33。0x33 的设置为将第 0、1、4、5 位的内存区域包括在 core 文件中。

不想将共享内存包括在 PID 1234 的 core 文件中时, 仅将第 0 位、第 2 位、第 5 位设置为 1。执行下列命令。

```
$ echo 0x21 > /proc/1234/coredump_filter
```

可以使用内核启动参数 coredump\_filter 设置默认的值。如果在 /etc/grub.conf 等中设置 coredump\_filter=0x21, 则所有进程的默认值都变成 0x21。

## sysctl

/proc/sys/kernel/ 下有关于 core 文件的 sysctl 文件。下面针对这些进行介绍。

### /proc/sys/kernel/core\_pattern

使用 core\_pattern 可以设置核心转储文件的命名规则。在 Fedora 12 中默认为 core。在 RHEL6 中默认的是下列内容, 实际上这也是 core。

```
# cat /proc/sys/kernel/core_pattern
|/usr/libexec/abrt-hook-ccpp /var/spool/abrt %p %s %u %c
```

当第一个字符为 | 时, 将执行在其后指定的进程 (这个进程成为 core\_pattern 进程)。/usr/libexec/abrt-hook-ccpp 是生成自动 bug 报告工具 (ABRT) 文件的命令。

默认的文件名最后有 .<PID> (参考后面介绍的 core\_uses\_pid)。在 core 的情况下为 core.<PID>。

更改 core\_pattern, 可以设置为 core 以外的文件名。如表 7-16 所示, 除了字符串以外, core\_pattern 还可以指定以 % 开始的模式 (pattern)。

表 7-16 core\_pattern 的字符串模式

模 式	内 容
%p	进程 ID
%u	用户 ID
%g	组 ID
%s	信号编号
%t	core 文件生成的时刻
%h	主机名称
%e	执行文件名称 (命令名称)
%c	core 文件大小的限制值 (ulimit -c 的大小) (2.6.24 以后)

指定 %p 时, 文件名的最后不加 .<PID>。下面是更改 core\_pattern 时的例子。

```
# echo "coredump-%p-%s-%h-%e" > /proc/sys/kernel/core_pattern
# ulimit -c unlimited
# sleep 1000 & kill -11 $!
[1] 2968
[1]+  段错误                  (核心转储) sleep 1000
# ls
coredump-2968-11-localhost.localdomain-bash
```

#### /proc/sys/kernel/core\_pipe\_limit

core\_pipe\_limit 是限制收集核心转储的进程同时执行的 sysctl 参数次数。在 RHEL6 中由 abrt 工具包将默认值设置为 4。

当 /proc/sys/kernel/core\_pattern 的第一个字符为 | 时, core\_pattern 进程会获取核心文件, 但此时内核会停止崩溃进程<sup>译注2</sup>。这是为了获取 /proc/<崩溃进程的 PID>/ 的数据。

内核等待 core\_pattern 进程结束后, 删除崩溃进程。但是, 当 core\_pattern 进程存在 bug 时, 核心转储处理可能不会结束。不断崩溃的进程都会由于这个有 bug 的 core\_pattern 进程而被迫停止。

要让数量超过 core\_pipe\_limit 的值的进程同时生成 core 文件时, 会放弃生成 core 文件, 输出下列信息。

```
Pid 9322(stress) over core_pipe_limit
Jan 12 19:30:46 localhost kernel: Skipping core dump
```

#### /proc/sys/kernel/core\_uses\_pid

将 core\_uses\_pid 设置为 1, core 文件的最后就会自动添加 .<PID>。设置为 0 时则不添加。RHEL6 中默认为 1。

译注 2: 崩溃进程对应的英文是 crash process。

## 小结

获取运行中进程的核心转储的 `gdb` 的 `gcore` 命令不支持核心转储过滤器。对于 `crash` 的扩展模块——`gcore` 命令，如果使用 `-f` 选项，同样可以限制获取的内存区域。详细内容请参考 Hack #61。

## 参考文献

- Documentation for `/proc/sys/kernel/*`  
`Documentation/sysctl/kernel.txt`
- 3.4 `/proc/<pid>/coredump_filter` - Core dump filtering settings  
`Documentation/filesystems/proc.txt`

——Naohiro Ooiwa

## HACK #61 生成用户模式进程的进程核心转储

本节介绍使用 `crash` 的 `gcore` 命令的方法，它生成内核崩溃转储中包含的用户模式进程的核心转储。

使用 `crash` 的 `gcore` 命令，可以生成内核崩溃转储中包含的用户模式进程的进程核心转储。`gcore` 命令并非 `crash` 的标准功能，而是作为扩展模块开发的。

## 使用案例

由于应用程序（用户模式进程）而造成崩溃时需要使用 `gcore` 命令。这时，应用程序的数据包含在内核崩溃转储中，因此某种程度上也可以使用 `crash` 进行故障分析。但是，对于应用程序开发人员来说，与要求具有内核相关知识的 `crash` 相比，`gdb` 作为应用程序的调试工具，使用起来更加方便。使用 `gcore` 命令将内核崩溃转储内的应用程序数据作为进程核心转储提取出来，就可以使用 `gdb` 进行故障分析。

另外，使用 `gdb` 还可以对 `crash` 命令不提供的下列用户模式进程进行操作：

- 用户空间的栈上的回溯。
- 显示代码级别上的执行点。

与 `gdb` 一样，也可以从 `crash` 获取用户模式进程的符号信息，这将在后面介绍。

---

**小贴士：**应用程序造成内核崩溃的情况很多，例如，在以确保较高可用性为目的的群集环境下运行应用程序的情况。在这种群集环境下，为了缩短检测出故障时的节点切换时间，有时会通过刻意造成崩溃来停止活动节点。与一般的关闭（`shutdown`）处理方法相比，崩溃时不用结束运行中的应用程序，速度相应得到提高。应用程序的数据包含在崩溃时生成的内核崩溃转储中，因此此后也可以进行故障分析。

---

# 安装

gcore 扩展模块在 crash 论坛网站 (<http://people.redhat.com/anderson/>) 的扩展模块发布页 (<http://people.redhat.com/anderson/extensions.html>) 上可以找到。

crash 的详细安装方法在 HACK #59 中已经介绍。创建 gcore 扩展模块时需要将源代码解压缩到扩展模块用的 extensions 目录下。

```
$ cd extensions
$ tar xf ~/src/gcore.tar.bz2
$ cd ../
$ make extensions
```

创建处理程序如果正常结束，在 extensions 目录下就会生成 gcore.so 文件。

```
$ ls -lhd ./extensions/gcore.so
-rwxr-xr-x 1 root root 235K 2011-05-28 18:02 ./extensions/gcore.so
```

## 基本使用方法

启动 crash 命令后，将 gcore 扩展模块安装到 crash 中，提取用户模式进程的进程核心转储，使用 gdb 显示提取的进程核心转储的回溯<sup>译注3</sup>。crash 启动后安装 gcore.so，安装时使用 extend 命令。

```
$ pwd
./crash-5.1.5/
$ ./crash vmlinux
...
crash> extend gcore.so
./extensions/gcore.so: shared object loaded
```

如果安装成功，就会添加 gcore 命令。可以使用 help 命令进行确认。

```
crash> help | grep gcore
bt          gcore        net          set          vtop
```

gcore 命令的语法如下所示。

```
gcore [-v <冗长级别>] [-f <过滤器级别>] [<PID> | <任务地址>] *
```

向 gcore 命令指定想要获取进程核心转储的用户模式进程的 PID 或任务地址。不指定时，就是作为 crash 命令对象的进程上下文。

下面尝试使用 gcore 生成进程核心转储。使用 ps 命令找到获取进程核心转储的用户进程的 PID。本次将 udevd 进程作为对象。

```
crash> ps
  PID  PPID  CPU  TASK  ST  %MEM  VSZ  RSS  COMM
...
  369    1  0  ffff88003c701730  IN  0.2  17524  1692  udevd
```

最左边的项目 PID 中显示 PID，从左边开始第 4 个项目 TASK 显示任务地址。crash 进程的 PID 为 369，任务地址为 ffff88003c701730。

---

译注 3：回溯对应的英文是 back trace。

下面尝试使用 `gcore` 命令。这里指定的是 `udevd` 进程的 PID 369。

```
crash> gcore 369
WARNING: page fault at 7f4a5fc98000
...<snip>...
WARNING: page fault at 7ffffc3d6c000
Saved core.369.udevd
crash> ls -hl ./core.369.udevd
-rw----- 1 hat hat 18M Jul  4 20:23 ./core.369.udevd
```

`gcore` 命令的处理顺利结束，生成了进程核心转储文件 `core.369.udevd`。`gcore` 生成的进程核心转储的文件名为如下格式。

`core.<PID>.<执行文件名>`

在 `gcore` 命令执行过程中，显示说明页面错误的下列信息。

```
gcore: WARNING: page fault at <地址>
```

用户内存空间中的下列页面区域不能从 `crash` 命令得到。

- 一次也没有分配内存的区域
- 分配过内存，但位于当前 `pageout` 的区域

`gcore` 命令对于这样的内存区域，都写入 0 来生成核心转储。为了让用户能够意识到这一点，将按照标准输出下列信息。

除页面错误以外，还可以显示 `gcore` 命令处理的进展信息。如表 7-17 所示，使用 `-v` 选项向 `gcore` 命令指定要显示的信息。例如，如果认为不需要上述页面错误的通知信息时，可以在 `-v` 选项中指定 0，来抑制信息的显示。

```
crash> gcore -v 0 369
Saved core.369.udevd
crash>
```

`gcore` 命令中也有与核心转储过滤器对应的功能，在这里没有用到。在 `-f` 选项中指定位掩码的值。能够过滤的内存种类与核心转储过滤器相同。过滤的内存种类和 `-f` 选项的值的对应关系如表 7-18 所示。

表 7-17 冗长信息的种类与 `-v` 选项的值的对应关系

值	crash 进展信息	crash 库错误	页面错误
0			
1	×		
2		×	
4 (默认值)			×
7	×	×	×

表 7-18 过滤的内存区域种类与 -f 选项的值的对应关系

	AP	AS	FP	FS	ELF	HP	HS
0							
1	×						
2		×					
4			×				
8				×			
16				×	×		
32						×	
64							×
127	×	×	×	×	×	×	×

下面使用 gdb 来显示该进程核心转储的回溯。

```
$ gdb /usr/lib/debug/sbin/udevd.debug ./core.369.udevd
(gdb) bt
#0 0x00007f4a60398258 in __poll (fds=0x7f4a610da020, nfds=5, timeout=-1) at
./sysdeps/unix/sysv/linux/poll.c:83
#1 0x00007f4a60ebb483 in main (argc=<optimized out>, argv=<optimized out>) at
udevd/udevd.c:1406
```

从回溯的信息可以看出，提取内核崩溃转储时，udevd 进程执行了函数 `__poll` 内的地址 `0x00007f4a60398258`。从函数名称可以看出，udevd 进程当时正在执行轮询 (polling)。如果使用 gdb 命令的 `x` 选项对这个地址进行反汇编，可以发现插入了执行系统调用的命令 `syscall` 的下列命令。

```
(gdb) x/2i 0x00007f4a60398256
0x7f4a60398256 <__poll+22>: syscall
=> 0x7f4a60398258 <__poll+24>: cmp    $0xfffffffffffffff000,%rax
```

使用 `syscall` 命令执行系统调用时，使用 RAX 寄存器指定系统调用编号。在把模式转移到内核时的寄存器值可以使用 `gcore` 命令的 `info register` 选项来找出。

```
(gdb) info register
rax          0x7          7
rbx          0x7f4a61a61ec0  139957442584256
rcx          0xfffffffffffffff      -1
rdx          0xfffffffffffffff      -1
rsi          0x5          5
rdi          0x7f4a610da020  139957432590368
rbp          0x0          0x0
rsp          0x7fffc3d739e0  0x7fffc3d739e0
r8           0x7f4a60655230  139957421560368
r9           0x0          0
r10          0x0          0
r11          0x246        582
r12          0x7f4a61a03010  139957442195472
```

```
r13          0x7f4a610da130  139957432590640
```

```
...
```

如上所示，RAX 的值就是 x86\_64 架构中 poll 系统调用的系统调用编号。可以看出 udevd 进程正在等待轮询的事件。

## 使用 crash 参照用户进程的符号信息的方法

crash 命令不使用 gcore 命令也可以获得添加了符号信息的进程用户内存。使用 add-symbol-file 命令添加用户进程的符号信息。add-symbol-file 命令是 gdb 命令之一。crash 沿用 gdb 的代码，可以从 crash 使用一些 gdb 命令。crash 启动时读入的符号信息与 gdb vmlinux 的情况相同。

add-symbol-file 命令的第 1 参数提取包括应用程序符号信息的 obj 文件，第 2 参数提取与符号信息相对应的文本区域的开始地址。按照如下方式找出文本区域的开始地址。

```
crash> vm | head
PID: 1771  TASK: ffff8800370ae080  CPU: 1  COMMAND: "crash"
      MM          PGD          RSS  TOTAL_VM
fffff880037910780  ffff88003a4c1000  153044k  187904k
      VMA          START          END  FLAGS FILE
fffff8800377e1e90      400000      a64000  8001875 /home/hat/build/crash-5.1.5/
crash
fffff8800377aba250      c64000      c87000  8101873 /home/hat/build/crash-5.1.5/
crash
fffff8800377e1850      c87000      e15000  100073
fffff8800376ffb70      2423000      50a6000  100073
```

首先使用 vm 命令显示 crash 进程所管理的部分内存区域。可以看出，从第 5 行以后的 FILE 项目开始，把 crash 的执行文件映射到最前面的 2 个内存区域。

```
crash> vm -f 8001875
8001875: (READ|EXEC|MAYREAD|MAYWRITE|MAYEXEC|DENYWRITE|EXECUTABLE)
```

然后使用 vm -f 命令，在 FLAGS 项目显示设置的标志。由于设置了 EXECUTABLE 标志，因此可以看出是映射到可执行文件的文本区域。这个文本区域的开始地址是 START 项目中显示的 400000。请注意是以十六进制显示的。

按照下列方式，根据文本区域添加符号信息。

```
crash> add-symbol-file ./crash 0x400000
```

这样就可以从 crash 命令使用 crash 的符号信息。尝试显示 crash 中使用的全局变量 program\_context。为了根据变量名称能够联想到这个变量，这个变量中保存了 crash 的程序上下文的相关数据。对应的地址为用户空间的地址，因此就需要在 p 命令中使用 -u 选项。

```
crash> p -u program_context | head
$7 = {
  program_name = 0x7fff3a2f7955 "crash",
  program_path = 0x7fff3a2f7953 "./crash",
```

```
program_version = 0x89f92d "5.1.5",
gdb_version = 0x9129a0 "7.0",
prompt = 0x2435ce0 "crash> ",
flags = 2306124484192570375,
namelist = 0x7fff3a2f795b "/usr/lib/debug/lib/modules/2.6.32-71.el6.x86_64/
vmlinux",
dumpfile = 0x0,
live_memsrc = 0x7fff3a2f7993 "/dev/crash",
```

## 支持范围

现在, `gcore` 命令可以支持下列架构和内核版本 (见表 7-19)。在较近的版本中也有可能运行。

表 7-19 可以使用 `gcore` 命令的环境

架 构	X86
	X86_64 (包括 32 位模式进程)
内核版本 (已确认运行)	RHEL4.8 (基于 2.6.9)
	RHEL5.6 (基于 2.6.18)
	RHEL6.0 (基于 2.6.32)
	Linux2.6.36

## 注意事项

**注意事项：**如前所述, `crash` 不能处理一次也没有分配内存的区域或交换出 `swap out` 的内存区域。`gcore` 将这些区域全部设置为 0。

**注意事项：**`crash` 命令不能获得通过 `makedumpfile` 进行的内核部分转储中排除在转储对象以外的内存区域。想要获得排除在转储区域外的内存区域时, 会显示下列信息。

```
crash> rd -u 0x400000
rd: page excluded: user virtual address: 400000 type: "64-bit UVADDR"
```

也可以将排除在转储对象外的内存区域改写为 0 进行引用。使用 `set` 命令将 `crash` 参数 `zero_excluded` 改为 `on`。其初始设置为 `off`。

```
crash> set -v
...
zero_excluded: off
...
crash> set zero_excluded on
zero_excluded: on
crash> rd -u 0x400000
400000: 0000000000000000 .....
```

在 `gcore` 命令处理中获得排除在转储对象外的内存区域时也是一样。这时 `gcore` 命令将中断处理, `crash` 返回命令行输入。对内核的部分转储使用 `gcore` 命令时, 需要将 `crash` 参数 `zero_excluded` 设置为 `on`。

---

**小贴士：** `gcore` 命令生成的进程核心转储所包含的执行信息中包含寄存器信息。这是崩溃时用户进程具有的用户空间中寄存器的值。崩溃时，在用户进程的不同状态下寄存器的退出位置也不同，因此需要根据各状态从适当的位置回收寄存器信息。但是，有时难以指定适当的位置，也有寄存器值自身没有退出的情况，并不一定能够回收全部的寄存器信息。但大多数情况下都能充分回收 `gdb` 进行回溯所需的寄存器信息。

---

**小贴士：** 使用 `gcore` 命令生成的进程核心转储中并没有提取引起核心转储的信号。因此使用 `gdb` 对 `gcore` 命令生成的核心转储进行分析时不会显示下列信息。

```
Program terminated with signal 6, Aborted.
```

---

## 参考文献

- White Paper: Red Hat Crash Utility  
<http://people.redhat.com/anderson/>

——HATAYAMA Daisuke

## HACK #62 使用 lockdep 查找系统的死锁

本节使用 `lockdep`，检查系统是否可能发生死锁。

内核中存在自旋锁（spinlock）、Mutex、信号量（semaphore）等各种锁。这些锁用于内核中的资源的排他控制，但是如果使用方法错误，就会造成称为死锁的状态，等待锁的进程就永远无法恢复，最严重的情况会导致内核死机。

典型的情况包括自己递归地获取锁的情况，以及多个处理顺序不当引起死锁的情况。例如，下面就是递归死锁的例子。

```
int func1(struct object *a)
{
    mutex_lock(&a->lock);
    ...
    ret = func1(a);
    ...
    mutex_unlock(&a->lock);
}
```

下面是 AB-BA 死锁的例子。

```
proc1() {
    mutex_lock(&lockA);
    mutex_lock(&lockB);
    ...
}

proc2() {
    mutex_lock(&lockB);
```

```
    mutex_lock(&lockA);  
    ...  
}
```

lockdep 的正式名称为“Runtime locking correctness validator”，其负责检查内核运行时锁之间的依存性和使用方法，检测系统发生死锁的可能性。

## lockdep 的结构

lockdep 记录内核启动后各个锁的获取状态（中断）、锁之间的获取顺序，将其作为锁的规则，对于这个规则以外的锁显示警告。

### 中断处理和单个锁

一般来说，递归死锁很容易就能通过程序的运行发现，在开发过程中也并不是非常难以发现。但是在编写操作系统时，中断处理是非常复杂的。也就是说，执行原来的程序时，处理过程中有时会收到硬件发出的中断（hardirq）或者软件中断（softirq）。在这些中断处理程序的内部获取与原来的程序相同的锁，就有可能发生意想不到的死锁。特别是像 Linux 这样，多个开发人员对一个程序进行增量式（incremental）开发时，就有可能由于相互之间未能沟通好而造成这样的死锁。

为了避免产生这样的死锁，对于中断处理程序中使用的锁，一般会在原来的程序上禁止 hardirq 和 softirq 以避免中断处理的运行。lockdep 记录在获取各锁时的禁止中断状态，在 softirq 或 hardirq 未禁止的地方获取的锁一旦用在禁止 softirq 或 hardirq 的地方，就会发出警告。像这样在不同情况下使用锁，很有可能是因为某一个锁的使用方法发生错误。另外，也因为禁止 hardirq 或 softirq 后使用的锁，一般在它们的中断处理程序内部使用的可能性较高。

在 lockdep 中，根据获取锁时的情况，将这些锁进行了如下的分类（见表 7-20）。

表 7-20 根据中断状态对锁进行分类

	允许 hardirq	禁止 hardirq
允许 softirq	Hardirq-unsafe	softirq-unsafe
禁止 softirq	softirq-safe	hardirq-safe

实际上禁止 hardirq 时 softirq 也会自动禁止，因此 softirq-unsafe 的意义与 hardirq-unsafe 相同，不存在仅允许 softirq 的情况。

### 多个锁之间的依存关系

另一方面，多个锁之间也有可能发生死锁。例如，称为 AB-BA 锁的死锁就非常有名，这是在某个处理路径与另一个处理路径上两个锁的获取顺序不同的情况下发生的。这样的处理看起来不太可能发生，但其实像 Linux 这样必须在别人制作的子系统上加入自己的代码时，就有可能弄错 API 的调用顺序，造成这样的死锁。

lockdep 会动态记录所有锁的依存关系（获取顺序），每次获得锁时都会与在此之前的获取模式进行比较，就可以确认会不会实际发生这样的问题。

在多个锁之间，还需要与单个锁时一样检查是否可以中断。

- 获取了 hardirq-safe 的锁后，获取 hardirq-unsafe 的锁时
- 获取了 softirq-safe 的锁后，获取 softirq-unsafe 的锁时

在这些情况下，第一个锁有可能在中断处理程序内部使用，但依存于后面仅在中断处理程序外部使用的锁。也就是说，在这个状态下进行处理并发生中断时，中断处理程序内有可能不获取 irq-safe 的锁就发生死锁。

## 锁的嵌套

在 Linux 内核内部，有时需要按照某个顺序获取相同种类的类（数据结构）的不同实例（instance）的锁。例如，从块设备（sda）机器内部的分区（sda1）来看，应当先获取块设备的锁，再获取分区的锁，才是正确的顺序。

lockdep 为了把握这个顺序，准备了专用的锁 API。对于 mutex，要向 mutex\_lock\_nested(mutex, subclass) 的 subclass 传递 1 以上的值。subclass 的值越大，嵌套的层次就一定越深（顺序必须是后获取的）。

在 bdev 结构的示例中，将传递给 subclass 的变量按如下方式定义。

```
enum bdev_bd_mutex_lock_class
{
    BD_MUTEX_NORMAL = 0,
    BD_MUTEX_WHOLE = 1,
    BD_MUTEX_PARTITION = 2
};

mutex_lock_nested(&bdev->bd_contains->bd_mutex, BD_MUTEX_PARTITION);
```

如果在使用 BD\_MUTEX\_PARTITION 锁定以后，再使用 BD\_MUTEX\_WHOLE 锁定，就是未按照应有的顺序锁定。

## lockdep 的系统开销

lockdep 必须记录所有的锁定操作，并与到目前为止的执行结果进行比较，因此这是非常重要的处理。但是 lockdep 将锁的使用方法设置为对于锁的各个使用模式仅处理一次，从而减轻负担。因此，lockdep 针对锁的每个使用模式，生成 64 位的散列值并记录下来，散列值相同的使用模式则从第二次以后不进行处理。

## 创建启用了 lockdep 的内核

要启用 lockdep，需要将内核配置的 DEBUG\_LOCKDEP 项目设置为 y。但是，在最近的内核中，必须要先启用 CONFIG\_LOCK\_STAT 或 CONFIG\_PROVE\_LOCKING 之一才会出现

这个项目。

```
% cd linux-2.6.xx
% make config
...
Lock debugging: prove locking correctness (PROVE_LOCKING) [N/y/?] y
...
Lock dependency engine debugging (DEBUG_LOCKDEP) [N/y/?] (NEW) y
...
% make bzImage
```

这样创建的内核中就包含 lockdep 功能，能够使用死锁检测功能。

## 尝试使用 lockdep 功能

为了实际运行 lockdep 功能，向内核中添加“带有 bug”的锁定处理的代码，查看 bug 是怎样检测出来的。

请看下面的代码，其中刻意地设置了  $AB \leftrightarrow BA$  死锁。

```
/* Lockdep test code */
static DEFINE_SPINLOCK(hack_spinA);
static DEFINE_SPINLOCK(hack_spinB);

void hack_spinAB(void)
{
    printk(KERN_ERR "hack_lockdep: A->B\n");
    spin_lock(&hack_spinA);
    spin_lock(&hack_spinB);
    spin_unlock(&hack_spinB);
    spin_unlock(&hack_spinA);
}

void hack_spinBA(void)
{
    printk(KERN_ERR "hack_lockdep: B->A\n");
    spin_lock(&hack_spinB);
    spin_lock(&hack_spinA);
    spin_unlock(&hack_spinB);
    spin_unlock(&hack_spinA);
}

void hack_lockdep_test(void)
{
    hack_spinAB();
    hack_spinBA();
}
```

这段代码包括按照  $A \rightarrow B$ 、 $B \rightarrow A$  的相反顺序获取锁的处理。因此，就存在潜在死锁的危险 (`hack_spinAB()` 和 `hack_spinBA()` 如果同时从不同的上下文中调用，则两个处理过程将互相等待而停止，造成死锁)。

这些反序的调用是由 `hack_lockdep_test()` 函数来进行的。实际运行这个处理过程时需要将向 `hack_lockdep_test()` 函数的调用添加到内核中。添加的位置可以设置为 `init/main.c` 的 `kernel_init()` 函数（必须在 `lockdep_init()` 的调用之后）。

```

extern void hack_lockdep_test(void);

static int __init kernel_init(void * unused)
{
    lock_kernel();
    /*
     * init can run on any cpu.
     */
    :
    :
    if (sys_access((const char __user *) ramdisk_execute_command, 0) != 0)
    {
        ramdisk_execute_command = NULL;
        prepare_namespace();
    }
    hack_lockdep_test();

    /*
     * Ok, we have completed the initial bootup, and
     * we're essentially up and running. Get rid of the
     * initmem segments and start the user-mode stuff..
     */
    init_post();
    return 0;
}

```

在进行了上述修改的内核中启动系统，在启动中就会检测并指出存在发生死锁的危险，如下所示。

```

[ 1.715230] hack_lockdep: A->B
[ 1.717816] hack_lockdep: B->A
[ 1.720204]
[ 1.720206] =====
[ 1.721178] [ INFO: possible circular locking dependency detected ]
[ 1.721178] 2.6.38 #1
[ 1.721178] -----
[ 1.721178] swapper/1 is trying to acquire lock:
[ 1.721178]  (hack_spinA){+.+...}, at: [<c040135b>] hack_spinBA+0x26/0x3d
[ 1.721178]
[ 1.721178] but task is already holding lock:
[ 1.721178]  (hack_spinB){+.+...}, at: [<c0401351>] hack_spinBA+0x1c/0x3d
[ 1.721178]
[ 1.721178] which lock already depends on the new lock.
[ 1.721178]
[ 1.721178]
[ 1.721178] the existing dependency chain (in reverse order) is:
[ 1.721178]
[ 1.721178] -> #1 (hack_spinB){+.+...}:
[ 1.721178]           [<c04626a3>] lock_acquire+0x98/0xba
[ 1.721178]           [<c07a738b>] _raw_spin_lock+0x20/0x2f
[ 1.721178]           [<c040131e>] hack_spinAB+0x26/0x3d
[ 1.721178]           [<c040137f>] hack_lockdep_test+0xd/0x16
[ 1.721178]           [<cc0a53994>] kernel_init+0x1a2/0x1b1
[ 1.721178]           [<c0403882>] kernel_thread_helper+0x6/0x10
[ 1.721178]

```

```

[ 1.721178] -> #0 (hack_spinA){+.+...}:
[ 1.721178]     [<c0461fcb>] __lock_acquire+0x9af/0xc3f
[ 1.721178]     [<c04626a3>] lock_acquire+0x98/0xba
[ 1.721178]     [<c07a738b>] _raw_spin_lock+0x20/0x2f
[ 1.721178]     [<c040135b>] hack_spinBA+0x26/0x3d
[ 1.721178]     [<c0401384>] hack_lockdep_test+0x12/0x16
[ 1.721178]     [<c0a53994>] kernel_init+0x1a2/0x1b1
[ 1.721178]     [<c0403882>] kernel_thread_helper+0x6/0x10
[ 1.721178]
[ 1.721178] other info that might help us debug this:
[ 1.721178]
[ 1.721178] 1 lock held by swapper/1:
[ 1.721178] #0: (hack_spinB){+.+...}, at: [<c0401351>] hack_spinBA+0x1c/0x3d
[ 1.721178]
[ 1.721178] stack backtrace:
[ 1.721178] Pid: 1, comm: swapper Not tainted 2.6.38 #1
[ 1.721178] Call Trace:
[ 1.721178]  [<c045fcf0>] ? print_circular_bug+0x8a/0x96

[ 1.721178]  [<c0461fcb>] ? __lock_acquire+0x9af/0xc3f
[ 1.721178]  [<c04626a3>] ? lock_acquire+0x98/0xba
[ 1.721178]  [<c040135b>] ? hack_spinBA+0x26/0x3d
[ 1.721178]  [<c07a738b>] ? _raw_spin_lock+0x20/0x2f
[ 1.721178]  [<c040135b>] ? hack_spinBA+0x26/0x3d
[ 1.721178]  [<c040135b>] ? hack_spinBA+0x26/0x3d
[ 1.721178]  [<c040135b>] ? hack_spinBA+0x26/0x3d
[ 1.721178]  [<c0401384>] ? hack_lockdep_test+0x12/0x16
[ 1.721178]  [<c0a53994>] ? kernel_init+0x1a2/0x1b1
[ 1.721178]  [<c0a537f2>] ? kernel_init+0x0/0x1b1
[ 1.721178]  [<c0403882>] ? kernel_thread_helper+0x6/0x10

```

在测试代码中只是将这些函数依次调用，因此并不会发生死锁。但是，内核注意到两个锁按照反序调用，并指出这段代码违反了避免死锁的规则。

## 小结

lockdep 在对内核的锁进行修改时可以起到很大的帮助作用。即使在没有使用锁定的情况下，也可以事先检测出因未按照开发人员设想的顺序使用内核 API 而有可能导致的锁竞争。在发布关于内核的补丁之前，可以使用 lockdep 进行检查，确认代码是否有发生死锁的可能性。这样就可以提高发布的补丁的质量，减少通过评价接受批评的可能性，缩短到合并补丁所需的时间。

## 参考文献

- Documentation/lockdep-design.txt

——Masami Hiramatsu

# HACK #63 检测内核的内存泄漏

本节介绍检测内核内存是否有泄漏的工具 kmemleak (Kernel memory leak detector)。

kmemleak 是检测内核空间的内存泄漏的调试功能。监测对象是通过内核的函数 `kmalloc()`、`vmalloc()`、`kmem_cache_alloc()` 分配的内存区域。从 Linux 2.6.31 开始安装了这个功能。启用 kmemleak，对内核、驱动程序、模块进行测试，就可以很方便地得出内存泄漏的可能性。

## 编译内核

要使用 kmemleak，需要启用内核配置 `CONFIG_DEBUG_KMEMLEAK`，重新构建内核。如果使用 `make menuconfig`，则配置项目如下。

```
Kernel hacking --->
  [*] Kernel memory leak detector
    (400) Maximum kmemleak early log entries
```

### 关于 Maximum kmemleak early log entries

`make menuconfig` 中除了 `Kernel memory leak detector` 以外还有 `Maximum kmemleak early log entries`。`kmemleak` 准备了仅在内核启动时使用的锁的缓冲区。这个项目就是用来设置这个缓冲区大小的最大值。

一旦内核启动就会进行 `kmemleak` 的初始化。初始化完成后，就可以随时输出日志，并定期对内存进行检查。从 `kmemleak` 的结构也可以检测出初始化结束之前的内存泄漏，但是不能输出日志。这时就将初始化结束之前的内存泄漏信息暂时存放在这个缓冲区。

启动时如果检测出大量的内存泄漏，输出了 `Early log buffer exceeded, please increase DEBUG_KMEMLEAK_EARLY_LOG_SIZE` 的信息，就需要增大 `Maximum kmemleak early log entries` 的值。单位是为输出日志而记录的内存个数。可以设置为  $200 \sim 40000$ 。默认为 400。

## 使用方法

启动使用 `CONFIG_DEBUG_KMEMLEAK=y` 重新构建的内核，`kmemleak` 线程就会每隔 10 分钟扫描一次内存。

当检测出可能存在内存泄漏时，就会输出信息。不需要进行设置等。

本节通过在样本程序中刻意加入内存泄漏，确认 `kmemleak` 检测出内存泄漏的过程。使用的内核版本为 2.6.35。

然后准备用来造成内存泄漏的内核模块 `kmleak.c`，进行编译。在所分配的内存区域写入了字符串 `memory leak test`。

```

# cat kmleak.c
#include <linux/module.h>
#include <linux/slab.h>

static int leak_func(void)
{
    char *p;
    char mark[] = "memory leak test";

    p = kmalloc(sizeof(mark), GFP_KERNEL);
    if (!p) {
        printk("kmleak can't allocate memory\n");
        return -ENOMEM;
    }
    printk("pointer to the allocated memory: %p\n", p);    信息
    memcpy(p, &mark, sizeof(mark));

    if (!p)           这里就是错误
        kfree(p);    不释放

    return 0;
}

static int __init kmleak_init(void)
{
    leak_func();
    return 0;
}

static void __exit kmleak_exit(void)
{
    return;
}

module_init(kmleak_init);
module_exit(kmleak_exit);

```

准备内核模块用的 Makefile，编译模块。向 make 的 -C 选项指定内核 2.6.35 的路径。

```

# cat Makefile
obj-m := kmleak.o
# ls
kmleak.c Makefile linux-2.6.35
# make -C ./linux-2.6.35 M=`pwd` modules

```

编译完成后，就生成了 kmleak.ko。然后将这个模块安装到内核中，接着从内核中卸载。

```

# insmod ./kmleak.ko
# lsmod
Module                  Size  Used by
kmleak                   914   0
.....
# rmmod kmleak

```

在释放内存的函数 kfree() 不调用的情况下，该内存的持有者消失。kmemleak 如果执

行内存扫描，就会输出表示内存泄漏的信息。kmemleak 进行内存扫描的间隔默认为 10 分钟。

虽然最多只需等待 10 分钟就会输出信息，但也可以在任意时刻执行内存扫描。首先挂载 debugfs。

```
# mount -t debugfs nodev /sys/kernel/debug/
```

向 /sys/kernel/debug/kmemleak 文件写入字符串 scan，kmemleak 线程就会在这时进行内存扫描。

```
# echo scan > /sys/kernel/debug/kmemleak
```

使用 dmesg 命令确认检测出内存泄漏的信息是否输出。

```
# dmesg
.....
pointer to the allocated memory: fffff88003c5b5960
这是 kmleak 模块输出的信息
kmemleak: 1 new suspected memory leaks (see /sys/kernel/debug/kmemleak)
kmemleak 输出的信息
```

这个信息会在检测出内存泄漏的可能性时显示。详细信息可以在 /sys/kernel/debug/kmemleak 中确认。

```
# cat /sys/kernel/debug/kmemleak
.....
unreferenced object 0xfffff88003c5b5960 (size 32):          地址
    comm "insmod", pid 6105, jiffies 4602131386 (age 2520.563s)    PID、jiffies
    hex dump (first 32 bytes):                                     内存的内容
        6d 65 6d 6f 72 79 20 6c 65 61 6b 20 74 65 73 74    memory leak test
        00 00 00 00 00 00 00 80 00 00 00 00 00 00 00 00    .....
    backtrace:                                                 回溯
        [<ffffffffff813c33a3>] kmemleak_alloc+0x21/0x3e
        [<ffffffffff810c9aa8>] kmem_cache_alloc+0xc7/0xd6
        [<fffffffffffa040901a>] 0xfffffff810c9aa8
        [<ffffffffff8100205f>] do_one_initcall+0x59/0x154
        [<ffffffffff8106ca16>] sys_init_module+0x9c/0x1da
        [<ffffffffff810089c2>] system_call_fastpath+0x16/0x1b
        [<ffffffffff810089c2>] 0xfffffff810089c2
```

显示的内容从上到下依次输出的是内存的地址、分配这个内存的进程（PID）、内存的分配时间（jiffies）与分配后经过的时间、内存的内容（最大 32 位）、回溯。

### /sys/kernel/debug/kmemleak 参数

/sys/kernel/debug/kmemleak 中还有除 scan 以外的参数，如表 7-21 所示。

表 7-21 /sys/kernel/debug/kmemleak 参数列表

参数	内容
off	禁用 kmemleak。不再追踪内存和分配和释放。一旦在这个参数中禁用，就不能再次启用

参    数	内    容
<code>stack=on</code>	启用任务栈区域的扫描。默认为 <code>on</code>
<code>stack=off</code>	禁用任务栈区域的扫描
<code>scan=on</code>	开始 <code>kmemleak</code> 线程的自动扫描。默认为 <code>on</code>
<code>scan=off</code>	停止 <code>kmemleak</code> 线程的自动扫描
<code>scan=&lt;secs&gt;</code>	设置 <code>kmemleak</code> 线程执行扫描的间隔。单位为秒。默认为 600 秒 (10 分钟)。设置为 0 则停止自动扫描
<code>scan</code>	立刻执行内存扫描
	清除检测出的数据。清除处理以前判断为存在内存泄漏可能的内存信息，不在 <code>/sys/kernel/debug/kmemleak</code> 中显示
<code>clear</code>	只是不显示，但在 <code>kmemleak</code> 内部是作为数据保存的，因此清除后也可以使用 <code>dump</code> 参数进行确认。在使用 <code>kmemleak</code> 前删除没有关系的信息时使用
<code>dump=&lt;addr&gt;</code>	显示内存信息

下面是使用了 `dump` 参数的例子。

```
# echo dump=0xfffff88003c5b5960 > /sys/kernel/debug/kmemleak
# dmesg
.....
pointer to the allocated memory: ffff88003c5b5960
kmemleak: 1 new suspected memory leaks (see /sys/kernel/debug/kmemleak)
kmemleak: Object 0xfffff88003c5b5960 (size 32):
kmemleak:   comm "insmod", pid 6105, jiffies 4602131386
kmemleak:   min_count = 1
kmemleak:   count = 0
kmemleak:   flags = 0x3
kmemleak:   checksum = 1470842492
kmemleak:   backtrace:
[<ffffffff813c33a3>] kmemleak_alloc+0x21/0x3e
[<ffffffff810c9aa8>] kmem_cache_alloc+0xc7/0xd6
[<fffffffffa040901a>] 0xfffffffffa040901a
[<ffffffff8100205f>] do_one_initcall+0x59/0x154
[<ffffffff8106ca16>] sys_init_module+0x9c/0x1da
[<ffffffff810089c2>] system_call_fastpath+0x16/0x1b
[<ffffffffffffffff>] 0xffffffffffffffff
```

## 内核启动参数

在安装了 `kmemleak` 的内核中，要禁用 `kmemleak` 功能，需要在内核启动参数中设置 `kmemleak=off`。

## 小结

使用 kmemleak 可以很方便地检测出内核的内存泄漏。可以用于设备驱动程序或内核模块的评估。也可以用于分辨内存泄漏的原因是应用程序还是 Linux 内核。

## 参考文献

- Documentation/kmemleak.txt

——Naohiro Ooiwa

## 第 8 章

---

# 概要分析与追踪

通过改善软件、系统的吞吐量或延迟，可以提高计算机的使用效率。特别是在 OSS 中可以自己获取源代码来改善这些性能。

Linux 内核的开发人员为了提高品质也会进行性能分析。因此，内核中就添加了各种性能分析和运行分析的功能。本章将介绍使用 perf tools 的性能分析、使用 ftrace 的运行分析、使用 SystemTap 的可编程追踪（programmable tracing）等。当然，这些功能对于内核的性能分析、用户程序的分析、系统的故障排除（trouble shooting）等也非常有用。

## HACK #64 使用 perf tools 的概要分析（1）

发现 Linux 内核功能变差时，或想要改善性能但不知道当前哪里有问题时等，可以通过进行程序概要分析来定位有问题的位置。

### perf tools

perf tools 是由内核维护人员 Ingo Molnar 等人开发的 Linux 内核的综合性能概要分析工具。可以使用 CPU 中内置的性能计数器（performance counter）功能、内核的追踪点等，进行内核或应用程序的概要分析。同样的工具中 Oprofile 比较有名。perf tools 工具就是由于 Oprofile 与 Linux 内核的维护频率不同导致 Oprofile 不能识别内核上支持的 CPU 而开始开发的。因此把 perf tools 的源代码合并到 Linux 内核的源码树中之后，tools/perf 下面有最新版的代码（反过来说，如果不使用与所用内核相对应的 perf tools 代码，就有可能无法正常运行）。

顾名思义，perf tools 是由多个子命令（工具）构成的。tools/perf/Documents/ 下的文件中整理了各个子命令的使用方法等。这里首先介绍创建一个能够完整使用 perf tools 的环境的方法。

### 确认 perf tools 的运行

perf tools 是至今仍在不断进行开发的内核的附属工具，因此也在不断地进行 bug 修正和功能添加。这里以 Linux 2.6.35 为基础进行介绍，同时也介绍新版本中所需的一些信息。

可以根据下列配置选项是否启用来确认内核是否支持 perf tools。

CONFIG\_PERF\_EVENTS

在运行中的系统中，可以按照下列方式直接运行 perf 命令来确认运行情况。

```
# perf list  
# perf top
```

## 安装 perf tools

使用比发布版更新的内核时，建议重新创建这个内核附带的 perf tools 再使用。在版本不同的内核中，perf tools 可能无法正常运行。另外，创建之前必须添加一些头文件或库，但在不同发布版中创建 perf tools 所需的数据包不同。Debian 系列的发布版中需要下列 deb 数据包。

```
glibc-dev, libelf-dev, libdw-dev, libnewt-dev, binutils-dev, zlib-static
```

Red Hat 系列的发布版中需要下列 RPM 数据包。

```
glibc-devel, elfutils-devel, elfutils-libelf-devel, newt-devel, binutils-devel
```

更新版本的 perf tools 的创建有时需要 Perl 或 Python 的开发版数据包。虽然没有这些也可以创建，但会失去 Perl 或 Python 脚本扩展的支持。perf tools 按照下列方式创建后就可以安装。

```
# cd tools/perf  
# make  
# make install
```

## 执行 perf tools

perf tools 命令是像 top 命令那样及时显示系统运行状态。二者根本上的不同在于 top 命令显示的是关于进程运行的统计信息，而 perf tools 命令显示的是通过性能计数器获取的用户进程和内核自身的概要分析信息。使用这个功能，就可以得知运行中内核的哪个函数影响了性能。执行 perf tools 需要执行下列命令

```
# perf top
```

将显示下列输出内容。默认为每隔 2 秒更新一次。

```
-----  
PerfTop: 411 irqs/sec  kernel:62.0%  exact: 0.0% [1000Hz cycles],  
(all, 2 CPUs)  
-----
```

samples	pcnt	function	DSO
345.00	18.5%	read_hpet	[kernel.kallsyms]
160.00	8.6%	ioread32	[kernel.kallsyms]
110.00	5.9%	acpi_idle_do_entry	[kernel.kallsyms]
61.00	3.3%	_spin_lock_irqsave	[kernel.kallsyms]

59.00	3.2% <code>mls_compute_sid</code>	[kernel.kallsyms]
42.00	2.2% <code>__GI strstr</code>	<code>libc-2.11.2.so</code>
41.00	2.2% <code>hpet_next_event</code>	[kernel.kallsyms]
34.00	1.8% <code>native_read_tsc</code>	[kernel.kallsyms]
30.00	1.6% <code>__pthread_mutex_lock_internal</code>	<code>libpthread-2.11.2.so</code>
27.00	1.4% <code>_spin_lock</code>	[kernel.kallsyms]
26.00	1.4% <code>unix_poll</code>	[kernel.kallsyms]
24.00	1.3% <code>_spin_unlock_irqrestore</code>	[kernel.kallsyms]
24.00	1.3% <code>schedule</code>	[kernel.kallsyms]
20.00	1.1% <code>fget_light</code>	[kernel.kallsyms]
20.00	1.1% <code>__pthread_mutex_unlock</code>	<code>libpthread-2.11.2.so</code>

`perf top` 命令在运行过程中，可以使用【d】键更改显示的更新频率。按下【q】键结束命令。`perf top` 命令通过对 CPU 上发生的事件进行采样来进行概要分析。`perf top` 默认通过统计所消耗的 CPU 周期的事件进行概要分析，此外还可以从缓存未命中（cache miss）事件、执行命令条数事件、分支预测错误事件等多个视图进行内核的概要分析。使用 `perf list` 命令可以获取这些事件的列表。

```
# perf list
```

由于种类非常多，这里仅介绍具有代表性的概要事件（profile event），见表 8-1。

表 8-1 `perf` 中可使用的概要事件列表

事件名称	说    明
<code>cpu-cycles</code> 或 <code>cycles</code>	CPU 的消耗周期（时间）的概要事件
<code>instructions</code>	执行的命令条数的概要事件
<code>cache-misses</code>	缓存未命中的概要事件
<code>branch-misses</code>	分支预测错误的概要事件
<code>page-faults</code> 或 <code>faults</code>	页面错误的概要事件
<code>minor-faults</code>	次要页面错误（不需 I/O 的页面错误）的概要事件
<code>major-faults</code>	严重页面错误（需要 I/O 的页面错误）的概要事件
<code>context-switches</code> 或 <code>cs</code>	上下文切换的概要事件

将这些事件名称作为 `-e` 选项的参数，就可以进行基于不同指标的概要分析。例如，使用下列命令就可以获取缓存未命中的概要。

```
# perf top -e cache-misses
```

## 小结

本节介绍了 `perf tools` 及其安装方法、使用 `perf top` 命令及时进行概要分析的方法。`perf top` 的概要分析可以用于查找系统整体性能降低的原因。

## 参考文献

- tools/perf/Documentation/perf.txt
- tools/perf/Documentation/perf-top.txt

——Masami Hiramatsu

## HACK #65 使用 perf tools 的概要分析（2）

本节介绍使用 perf tools 进行详细概要分析的方法。

Hack #64 介绍了使用 perf top 进行及时系统概要分析的方法。这里将进一步介绍实际的概要分析技术，使用 perf tools 的其他功能，获取更详细的概要，特别是在 x86 命令代码或 C 语言源代码层面追踪热点（hot spot）的方法。

### 使用 perf 进行概要分析的步骤

perf top 通过一条命令就可以进行概要分析，而 perf tools 的其他子命令则基本上是按照下列步骤进行概要分析。

1. 选定概要事件。
2. 记录概要分析和数据。
3. 分析记录的数据。

因此，首先从使用 perf list 显示的事件列表中，决定使用哪个事件，然后使用 perf record 记录事件，最后使用 perf report 或 perf annotate 进行详细的事件信息分析。关于进行概要分析的事件请参考 Hack #64 中的表 8-1。除了表 8-1 介绍的事件以外，有些处理器还可以在 L1 缓存信息、TLB 缓存信息等更为详细的信息的基础上进行概要分析。这里将介绍对发生了缓存未命中的位置进行详细概要分析的方法。perf record 命令需要使用下列方法之一指定进行概要分析的对象来执行。

- 通过进程 ID 指定（--pid）。
- 通过线程 ID 指定（--tid）。
- 指定特定 CPU 上的所有进程（--profile\_cpu）。
- 指定所有 CPU 上的进程（--all）。
- 从命令行指定要执行的命令。

然后与 perf top 一样，指定属于所记录种类的事件并将启动。例如，要对所有 CPU 上发生的缓存未命中进行概要分析时，进行下列操作。

```
# perf record -e cache-misses --all
```

可以在事件的“:”后面传递选项。

```
# perf record -e cache-misses:k --all
```

表 8-2 所示为 x86 上可以传递给事件的选项。

表 8-2 perf tools 事件的选项

选 项	说 明
k	仅获取内核 (ring0) 的事件
u	仅获取用户空间 (ring3) 的事件
p	获取事件更为准确的发生地址。如果可以使用 PEBS 和 LBR 则启用 (LBR 为可选项)。最少也可以通过启用 PEBS, 获取发生事件的命令之后下一个命令的地址
pp	如果可以配合使用 PEBS 和 LBR 则都两者启用。其中一个不能使用时出错。可以获取事件准确的发生地址

### 使用 PEBS 和 LBR 的正确采样

PEBS 为 Precise Event Based Sampling 的缩写, 是从 Intel Core 微架构 (与 NetBurst 中的一部分) 开始导入的更为准确的性能计数器结构。虽然可以适用 PEBS 的事件有限, 但是对于找出事件准确的发生地址是非常有用的。在此之前性能计数器都是在事件发生后造成外部中断, 因此发生中断的时间比事件发生时命令的地址稍靠后。因此从中断处理程序可以看到的 IP 寄存器显示的地址并不是正确的地址。而 PEBS 则可以在事件发生后立刻造成异常, 从而能获取事件发生时命令运行的地址。但是, 事件发生的命令本身已经运行, 因此 IP 寄存器显示的值就是事件发生的命令的下一条命令的地址。

x86 架构采用了 CISC 命令, 各命令的长度各不相同, 当事件发生的命令为分支命令时, 难以从分支的节点找出从哪里分出的, 仅使用 PEBS 很难正确地找到事件发生的命令。

在 perf tools 中将这个功能和 LBR (Last Branch Record) 配合使用, 通过逆运算求出事件实际发生的命令的地址。LBR 是记录最后执行分支命令的地址的功能。PEBS 如果在事件发生后立即调用处理程序, 则首先参照 LBR。LBR 中记录了最后分支的起点和终点的地址, 当 IP 寄存器显示的地址与 LBR 显示的分支终点地址一致时, 分支起点地址就是真正的事件发生地址。

当与 LBR 的信息不一致时, PEBS 处理程序对从分支终点的地址到 IP 寄存器显示的地址为止的各命令依次进行解码, 找出到达 IP 寄存器显示地址前的一个命令的地址。这样就可以得知事件真正发生的地址。

但是, 在使用有些处理器时是无法启用这些功能的。这时就会返回 -ENOTSUPP 错误。PEBS 的功能对于某些类型的事件也不能使用。在使用时需要注意。

## 进行缓存未命中的概要分析

`perf record`基本上是将概要事件和概要对象的命令指定为参数。一般来说，如果指定测量对象的基准测试软件或 `stress` 命令，则基准测试软件结束时 `perf tools` 也自动结束。例如，在下列命令中运行称为 `stress` 的命令 10 秒钟，并记录其概要信息。

```
# perf record -e cache-misses stress -c 8 -i 2 -m 2 --timeout 10s
stress: info: [12035] dispatching hogs: 8 cpu, 2 io, 2 vm, 0 hdd
stress: info: [12035] successful run completed in 10s
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.450 MB perf.data (~19640 samples) ]
```

记录的概要结果可以使用 `perf report` 命令来查看。

# perf report

启动 `perf report` 命令就会如图 8-1 所示在 TUI (文本用户界面) 的界面上显示概要结果。

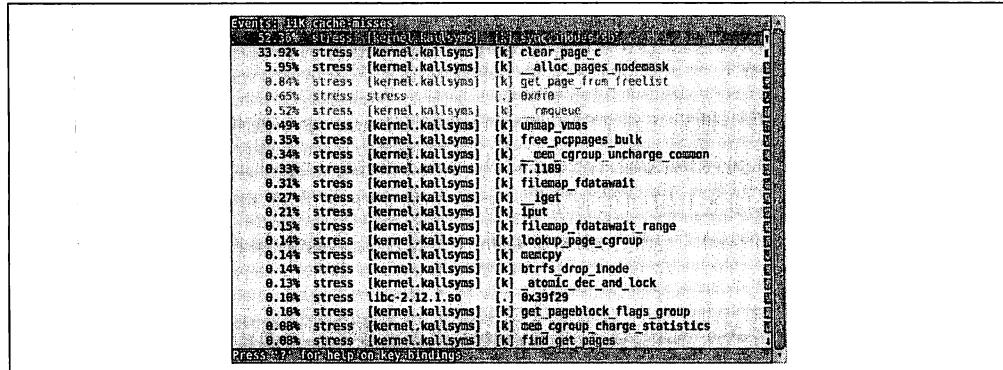


图 8-1 perf report 命令启动后

概要结果中显示了在哪个执行文件、库或者内核的哪个函数发生了缓存未命中。根据这个结果，与汇编程序（assembler）源代码进行对比时，将光标（cursor）移动到想要查看详细内容的行，按下【Enter】键就可以打开子窗口（见图 8-2）。

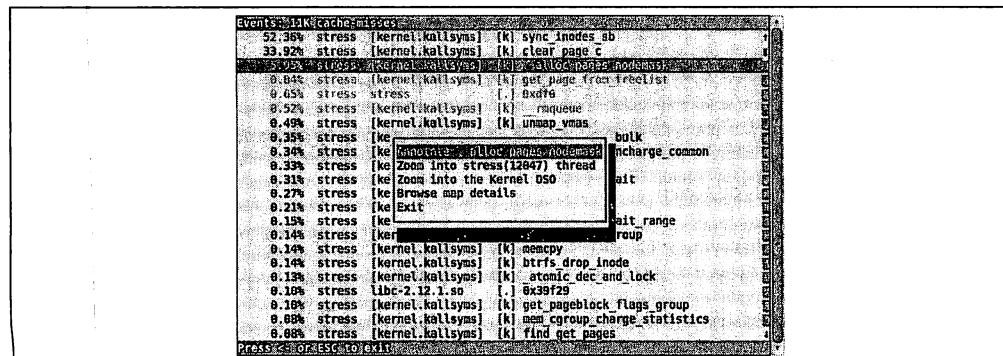
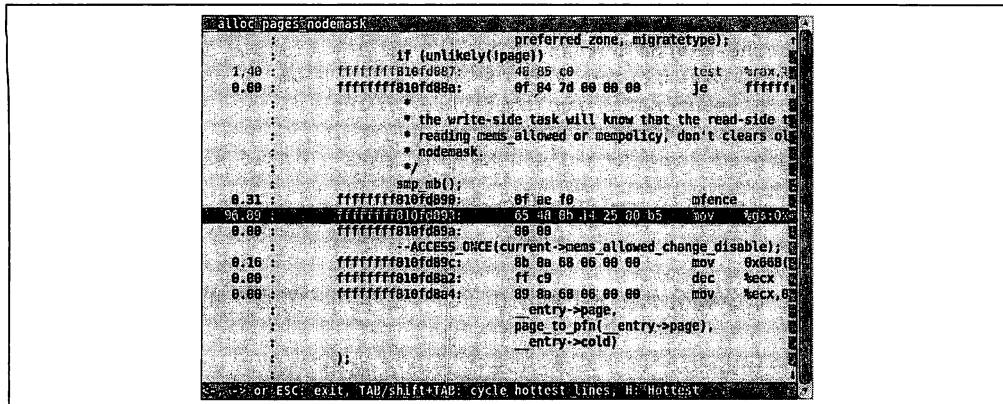


图 8-2 perf report 的子窗口

在这里选择“Annotate \_ \_alloc\_pages\_nodemask”的项目，就会显示包括事件发生位置周边的反汇编程序在内的详细报告（见图 8-3）。当分析对象（这里为内核）具有调试信息时，还会显示 C 的源代码和事件发生的行。本节是以分析内核中的缓存未命中为例的，此外还可以分析用户空间的应用程序。

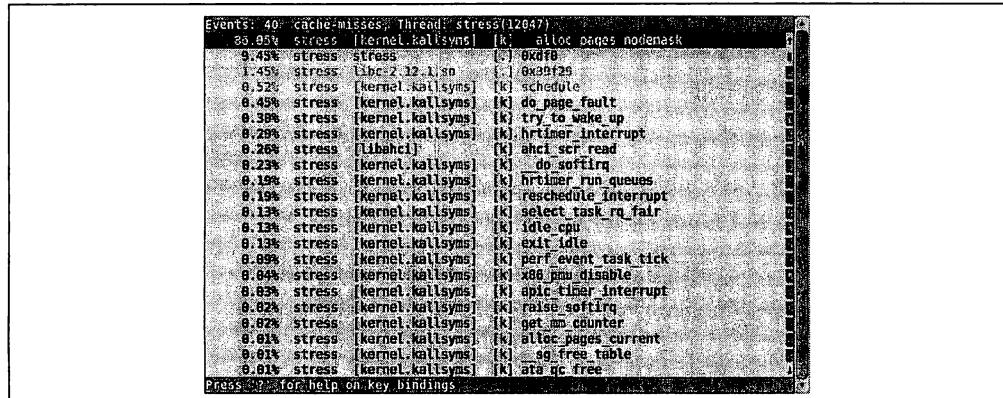


```
alloc_pages_nodemask
    preferred_zone, migratetype);
    if (unlikely(!page))
        ffffff810fd887: 48 85 c9      test    %rax,%r
0.68 :      ffffff810fd88a: 0f 94 7d 69 88 88    je     ffffff
    *
    * the write-side task will know that the read-side is
    * reading mems allowed or mempolicy, don't clears old
    * nodemask.
    *
    * smp_mb();
0.31 :      ffffff810fd899: 0f 8c 10      mfence
96.89 :      ffffff810fd899: 65 48 68 14 25 80 b5 309  %3:0x
0.89 :      ffffff810fd89a: 68 88
    -ACCESS_ONCE(current->mems_allowed.change_disable);
0.16 :      ffffff810fd89c: 9b 8a 68 06 88 88    mov    %0x688(%r
0.89 :      ffffff810fd892: ff c9      dec    %ecx
0.89 :      ffffff810fd894: 89 8a 68 06 88 88    mov    %0xcx,%r
    entry->page;
    page_to_phys(entry->page),
    entry->old)
    );

```

图 8-3 显示详细报告

在这个示例中，可以看出地址 `fffffff810fd893` 的 `mov` 命令发生了缓存未命中。在这个界面按【Tab】键就会依次显示相同文件内的不同缓存未命中的位置。另外，按左箭头键可以返回与图 8-1 相同的画面。如果选择图 8-2 的项目中的“Zoom into stress(12047) thread”，就会如图 8-4 所示仅显示与选择的事件在相同线程内的缓存未命中。



Events: 40 cache misses, Thread: stress(12047)	0.05% stress [kernel.kallsyms] [k] alloc_pages_nodemask
9.45% stress stress [k] 0x0000	
1.45% stress libc-2.12.so [k] 0x39f29	
0.52% stress [kernel.kallsyms] [k] schedule	
0.45% stress [kernel.kallsyms] [k] do_page_fault	
0.38% stress [kernel.kallsyms] [k] try_to_wake_up	
0.29% stress [kernel.kallsyms] [k] timer_interrupt	
0.26% stress [libhci] [k] ahci_sci_read	
0.23% stress [kernel.kallsyms] [k] do_softirq	
0.19% stress [kernel.kallsyms] [k] timer_run_queues	
0.19% stress [kernel.kallsyms] [k] reschedule_interrupt	
0.13% stress [kernel.kallsyms] [k] select_task_rq_fair	
0.13% stress [kernel.kallsyms] [k] idle_cpu	
0.13% stress [kernel.kallsyms] [k] exit_idle	
0.09% stress [kernel.kallsyms] [k] perf_event_task_tick	
0.04% stress [kernel.kallsyms] [k] x86_pmu_disable	
0.03% stress [kernel.kallsyms] [k] apic_timer_interrupt	
0.02% stress [kernel.kallsyms] [k] raise_softirq	
0.02% stress [kernel.kallsyms] [k] get_mm_counter	
0.01% stress [kernel.kallsyms] [k] alter_pages_current	
0.01% stress [kernel.kallsyms] [k] sg_free_table	
0.01% stress [kernel.kallsyms] [k] ata_qc_free	

图 8-4 仅显示与线程相关的事件

要从这里返回图 8-1 的界面，同样需要按左箭头键。如果选择图 8-2 的项目中的“Zoom into the kernel DSO”，则仅显示内核中发生的缓存未命中。关于键盘操作，可以按【?】键，就会出现如图 8-5 所示的帮助界面，可以作为参考。

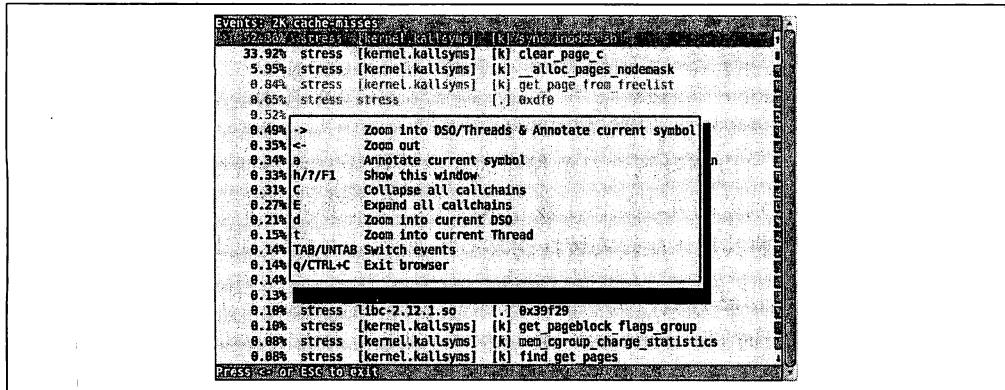


图 8-5 键盘帮助

## 小结

本节介绍了使用 perf tools 功能中的 perf record 和 perf report 进行详细的概要分析的方法。perf record 仅可以记录任意命令的运行。perf report 可以详细分析概要结果，添加反汇编程序显示出来。因此，在想要对使用 perf top 时觉得异常的部分进行集中分析的情况下，perf record 和 perf report 能够起到很大的作用。

——Masami Hiramatsu

# HACK #66 进行内核或进程的各种概要分析

本节使用 perf tools 从各种角度对内核或进程进行概要分析。

perf tools 的子命令中具有特定用途的性能分析功能。本节将举例说明其中一部分功能的使用方法。

## 使用 perf stat 获取综合统计信息

perf top 或 perf record/report 可以用来分析特定的性能，但是不适用于汇总综合性能。perf stat 就是针对指定的命令显示汇总的性能测量结果的功能。下面是显示执行“ls /”时汇总的所有命令的方法。

```
[~]# perf stat ls /
bin  dev  home  lib32      media  opt    root  selinux  sys  tracing  var
boot  etc  lib      lib64      mnt    proc  sbin  srv      tmp  usr
Performance counter stats for 'ls /':
    1.879831 task-clock-msecs      #      0.791 CPUs
          0 context-switches      #      0.000 M/sec
          1 CPU-migrations      #      0.001 M/sec
        269 page-faults      #      0.143 M/sec
1,562,779 cycles      #      831.340 M/sec
1,562,779 instructions      #      1.000 IPC
```

```
313,491 branches          #      166.766 M/sec
11,971 branch-misses     #      3.819 %
24,521 cache-references  #      13.044 M/sec
7,571 cache-misses       #      4.027 M/sec
0.002377647 seconds time elapsed
```

通过使用这个功能，可以测量测定对象程序的缓存未命中、执行命令条数、周期数等，还可以推测出性能特别差的处理是由于缓存未命中、分支预测错误，还是其他原因。一部分开发人员还经常将其用于比较安装性能改善补丁前后的结果，定量地显示效果。

下面查看测定结果的信息。第一个 `task-clock-msecs` 表示执行所花费的时间。还会显示消耗了多少 CPU 时间。`context-switches` 为进程或线程的切换次数。本次仅使用 `ls` 进行简单的处理，因此没有发生切换。`CPU-migrations` 为正在运行的 CPU 切换的次数。`page-faults` 表示按需分页或交换（swapping）等中使用的页面错误的发生次数和频率。`cycles` 是用时钟数量来表示进程执行所花费的 CPU 时间。`instructions` 表示执行的命令条数。`instructions` 中的 IPC 是 Instructions Per Cycle 的缩写。如果这个值较小的话，则每条命令的时钟数量越大，因此认为效率较低（例如，内存访问或 I/O 的频率较高）。`branches` 表示分支命令的数量和频率，`branch-misses` 表示分支预测错误的比例。另外，`cache-references` 表示缓存的参照，`cache-misses` 表示缓存未命中的频率等。一般来说，缓存未命中的次数越少，IPC 越大，就认为程序的执行效率越高。

## 使用 perf script 进行追踪

`perf tools` 记录内核中的采样事件，对其进行分析并得出性能信息。`perf tools` 还可以记录内核中的追踪事件。也就是说，可以与 Hack #69 中介绍的 `trace` 进行同样的操作。`perf script` 是显示已追踪的事件，或启动处理事件的脚本的功能。通过改写脚本，就可以根据需要对事件进行分析。可以记录的事件的种类除了可以如前所示从 `trace` 的 `debugfs` 接口获取，还可以使用 `perf list` 命令来获取。

```
# perf list
...
kvmmmu:kvm_mmu_pagetable_walk          [Tracepoint event]
kvmmmu:kvm_mmu_paging_element          [Tracepoint event]
kvmmmu:kvm_mmu_set_accessed_bit        [Tracepoint event]
kvmmmu:kvm_mmu_set_dirty_bit           [Tracepoint event]
kvmmmu:kvm_mmu_walker_error            [Tracepoint event]
kvmmmu:kvm_mmu_get_page                [Tracepoint event]
kvmmmu:kvm_mmu_sync_page               [Tracepoint event]
kvmmmu:kvm_mmu_unsync_page             [Tracepoint event]
kvmmmu:kvm_mmu_prepare_zap_page       [Tracepoint event]
```

在 `perf list` 的输出中，`Tracepoint event` 所示的事件就是追踪用的事件。

在下例中，将使用 `perf record` 获取的进程调度程序相关事件并通过 `perf script` 输出。

```
# perf record -e sched: '*' -a ls /
```

```

bin  dev  home  lib32 media  opt  root  selinux  sys  tracing  var
boot
etc  lib  lib64  mnt  proc  sbin  srv  tmp  usr
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.627 MB perf.data (~27391 samples) ]
# perf script
perf 5179 [000] 26985.658132: sched_stat_runtime: comm=perf
pid=5179 runtime=5751845 [ns] vruntime=392262038 [ns]
Xorg 1115 [001] 26985.658132: sched_stat_runtime: comm=Xorg
pid=1115 runtime=4757708 [ns] vruntime=106804101116 [ns]
Xorg 1115 [001] 26985.658142: sched_stat_sleep: comm=kworker/1:0
pid=4552 delay=9975987 [ns]
perf 5179 [000] 26985.658143: sched_stat_sleep: comm=kworker/0:2
pid=4027 delay=9972425 [ns]
swapper 0 [003] 26985.658144: sched_stat_sleep: comm=kworker/3:0
pid=4983 delay=9920987 [ns]
Xorg 1115 [001] 26985.658144: sched_wakeup: comm=kworker/1:0
pid=4552 prio=120 success=1 target_cpu=001
perf 5179 [000] 26985.658144: sched_wakeup: comm=kworker/0:2
pid=4027 prio=120 success=1 target_cpu=000
swapper 0 [003] 26985.658145: sched_wakeup: comm=kworker/3:0
pid=4983 prio=120 success=1 target_cpu=003
Xorg 1115 [001] 26985.658145: sched_stat_runtime: comm=Xorg
pid=1115 runtime=13009 [ns] vruntime=106804114125 [ns]
perf 5179 [000] 26985.658145: sched_stat_runtime: comm=perf
pid=5179 runtime=12798 [ns] vruntime=392274836 [ns]
...

```

在不向 `perf script` 赋予脚本的情况下运行时，只显示记录的追踪事件。也可以将这个追踪事件作为脚本的输入进行处理并返回结果。在 `perf script` 中添加 `--list` 选项并执行，就可以获取可使用的脚本列表。

```

# perf script --list
List of available trace scripts:
workqueue-stats                                workqueue stats (ins/exe/create/destroy)
rw-by-pid                                      system-wide r/w activity
failed-syscalls [comm]                          system-wide failed syscalls
rwtop [interval]                                system-wide r/w top
rw-by-file <comm>                             r/w activity for a program, by file
wakeup-latency                                 system-wide min/max/avg wakeup latency
sched-migration                                sched migration overview
syscall-counts-by-pid [comm]                   system-wide syscall counts, by pid
futex-contention                                futex contention measurement
netdev-times [tx] [rx] [dev=] [debug]          display a process of packet and
processing time
sctop [comm] [interval]                         syscall top
syscall-counts [comm]                          system-wide syscall counts
failed-syscalls-by-pid [comm]                  system-wide failed syscalls, by pid

```

脚本可以分为可在线处理的和不可在线处理的。但是一般 `sctop`、`rwtop` 等命令处理的事件频率非常高，因此如果在线处理，即进行追踪的同时运行脚本，则不可避免地会增加负载。另一方面，进行离线处理时是先追踪后处理的，因此可以抑制处理方式对追踪对象的影响。

要进行在线处理，需要在 `perf script` 后面指定脚本名称，如下所示。这里使用的是 `sctop`（测量系统调用次数）脚本。

```
# perf script sctop
syscall events:

event                                count
-----
read                                936451
futex                               10839
ioctl                               4359
write                               1661
close                               1133
swapoff                             694
getdents                            417
poll                                225
semop                               210
munmap                               198
setitimer                            148
writev                             146
sched_setparam                      137
wait4                               106
fcntl                               99
epoll_wait                           84
select                               76
mmap                                30
nanosleep                            24
open                                 19
fstat                               16
inotify_add_watch                   14
madvise                               9
recvmsg                             8
rt_sigprocmask                      8
sendmsg                             4
socket                               4
rt_sigaction                          4
stat                                 4
recvfrom                            1
...
```

要进行离线处理，需要在 `perf script` 的 `record` 和 `report` 后面指定脚本名称，如下所示。这里使用 `scscall-counts` 脚本，与前一次一样测量系统调用的次数。

```
# perf script record syscall-counts ls /
bin  dev  home  lib32  media  opt  root  selinux  sys  tracing  var
boot  etc  lib  lib64  mnt  proc  sbin  srv  tmp  usr
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.028 MB perf.data (~1226 samples) ]
# perf script report syscall-counts
Press control+C to stop and show the summary

syscall events:

event                                count
-----
```

mmap	26
mprotect	16
close	13
fstat	11
open	11
access	9
read	9
brk	3
munmap	3
futex	2
getdents	2
ioctl	2
rt_sigaction	2
write	2
set_robust_list	1
exit_group	1
set_tid_address	1
arch_prctl	1
statfs	1
getrlimit	1
fcntl	1
rt_sigprocmask	1
stat	1

## 使用自己的脚本处理数据

有时我们会打算自己写脚本来实现事件的统计处理，而不是执行现存的脚本。`perf script` 考虑这样的需求，准备了根据记录的事件生成脚本模型的功能。

下面查看一个用来调查系统调用和内核中分配内存的 `kmalloc()` 关系的脚本实例。首先生成使用 `perf record` 记录必要事件的数据文件。

```
# perf record -e kmem:kmalloc -e raw_syscalls:sys_enter ls /
bin  dev  home  lib32  media  opt  root  selinux  sys  tracing  var
boot  etc  lib  lib64  mnt  proc  sbin  srv  tmp  usr
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.037 MB perf.data (~1633 samples) ]
```

然后，向 `perf script` 传递 `-g` 选项，根据数据文件生成脚本的模型。`-g` 选项将生成的脚本语言（当前为 Perl 或 Python）作为参数。在下例中，生成的是 `perf` 脚本。

```
# perf script -g perl
generated Perl script: perf-script.pl
```

生成的脚本直接就具有显示事件内容的功能。

```
# perf script event handlers, generated by perf script -g perl
# Licensed under the terms of the GNU GPL License version 2

# The common_* event handler fields are the most useful fields common to
# all events. They don't necessarily correspond to the 'common_*' fields
# in the format files. Those fields not available as handler params can
# be retrieved using Perl functions of the form common_*(\$context).
```

```

# See Context.pm for the list of available functions.

use lib "$ENV{'PERF_EXEC_PATH'}/scripts/perl/Perf-Trace-Util/lib";
use lib "./Perf-Trace-Util/lib";
use Perf::Trace::Core;
use Perf::Trace::Context;
use Perf::Trace::Util;

sub trace_begin
{
    # optional
}

sub trace_end
{
    # optional
}

sub raw_syscalls::sys_enter
{
    my ($event_name, $context, $common_cpu, $common_secs, $common_nsecs,
        $common_pid, $common_comm,
        $id, $args) = @_;

    print_header($event_name, $common_cpu, $common_secs, $common_nsecs,
        $common_pid, $common_comm);

    printf("id=%d, args=%s\n",
        $id, $args);
}

sub kmem::kmalloc
{
    my ($event_name, $context, $common_cpu, $common_secs, $common_nsecs,
        $common_pid, $common_comm,
        $call_site, $ptr, $bytes_req, $bytes_alloc,
        $gfp_flags) = @_;

    print_header($event_name, $common_cpu, $common_secs, $common_nsecs,
        $common_pid, $common_comm);

    printf("call_site=%u, ptr=%u, bytes_req=%u, bytes_alloc=%u, ".
        "gfp_flags=%s\n",
        $call_site, $ptr, $bytes_req, $bytes_alloc,
        $gfp_flags);

    flag_str("kmem::kmalloc", "gfp_flags", $gfp_flags));
}

sub traceUnhandled
{
    my ($event_name, $context, $common_cpu, $common_secs, $common_nsecs,
        $common_pid, $common_comm) = @_;

    print_header($event_name, $common_cpu, $common_secs, $common_nsecs,
        $common_pid, $common_comm);
}

```

```

}

sub print_header
{
    my ($event_name, $cpu, $secs, $nsecs, $pid, $comm) = @_;

    printf("%-20s %5u %05u.%09u %8u %-20s ",
           $event_name, $cpu, $secs, $nsecs, $pid, $comm);
}

```

从生成的脚本的内容，可以看出是事件触发类型的程序，perf script 依次处理事件，然后将信息传递给与事件名称相对应的函数。通过更改处理这个事件的函数的内容，可以执行任意的处理。例如，要统计哪个系统调用调用了几次 kmalloc()，可以将生成的脚本修改为如下内容。

```

use lib "$ENV{'PERF_EXEC_PATH'}/scripts/perl/Perf-Trace-Util/lib";
use lib "./Perf-Trace-Util/lib";
use Perf::Trace::Core;
use Perf::Trace::Context;
use Perf::Trace::Util;

@count = ();
$current_id = -1;

sub trace_end
{
    for ($id = 0; $id < 1024; $id++) {
        if (@count[$id] != 0) {
            printf("syscall %d invokes kmalloc %d times.\n",
                   $id, @count[$id]);
        }
    }
}

sub raw_syscalls::sys_enter
{
    my ($event_name, $context, $common_cpu, $common_secs, $common_nsecs,
        $common_pid, $common_comm,
        $id, $args) = @_;

    $current_id = $id;
}

sub kmem::kmalloc
{
    my ($event_name, $context, $common_cpu, $common_secs, $common_nsecs,
        $common_pid, $common_comm,
        $call_site, $ptr, $bytes_req, $bytes_alloc,
        $gfp_flags) = @_;

    if ($current_id != -1) {
        @count[$current_id]++;
    }
}

```

将这个脚本另存为 `kmalloc-syscall.pl`，并使用 `perf script` 来尝试运行。

```
# perf script -s kmalloc-syscall.pl
syscall 0 invokes kmalloc 1 times.
syscall 2 invokes kmalloc 25 times.
syscall 9 invokes kmalloc 16 times.
syscall 10 invokes kmalloc 16 times.
syscall 21 invokes kmalloc 9 times.
syscall 137 invokes kmalloc 1 times.
```

可以像这样使用现有的脚本语言非常简单地记述事件的处理方式，就是 `perf script` 的特点。从这个功能也可以使用 Hack #71 介绍的通过 `perf probe` 生成的事件，因此实质上也就是可以使用脚本来处理内核中的大部分信息。

## 小结

本节介绍了 `perf stat` 和 `perf script` 的使用方法等。`perf stat` 可以用于在应用程序的性能分析、性能改善时数值指标的确认等，使用非常简单。另一方面，`perf script` 也可以使用 Perl 或 Python 等常用脚本来处理追踪结果的统计处理等。

## 参考文献

- [tools/perf/Documentation/perf-stat.txt](#)
- [tools/perf/Documentation/perf-script.txt](#)
- "Perf Tools: Recent Improvements" Arnaldo Carvalho de Melo  
<http://pdxplumbers.osuosl.org/2010/ocw/system/presentations/579/original/perf-plumbers2010.pdf>

——Masami Hiramatsu

# HACK #67 追踪内核的函数调用

本节介绍使用 ftrace 研究从哪里调用哪个函数的方法。

追踪是指记录对象软件何时、如何运行的调试或系统监控的方法之一。对引起故障的内核操作进行分析时，有时仅根据源代码是无法获悉实际调用顺序等的。出现性能问题时，或想要改善特定处理的延迟等问题时，就会想要追踪哪里存在瓶颈。使用基于采样的分析工具（profile）虽然也能大致知道性能降低的位置，但终究只能进行概率性的判断，有时不适合用于追踪某特定位置。

这种情况下，采用的方法是通过追踪器依次记录对象程序内部发生的处理并进行分析。有用的追踪主要有下列情况。

- 引起 bug 的条件的分析
- 定时（timing）bug 的分析

- 性能降低的分析

另外，有时也会为了进行系统性能监控或故障监控，而进行日常维护型的追踪。

## ftrace

最近发布了 Linux 内核的标准追踪功能 ftrace。ftrace 是内核的追踪器框架，有时也将作为 ftrace 母体的函数调用的追踪器称为 ftrace。这里将 ftrace 作为一般的追踪器框架来使用，函数调用的追踪器用函数追踪器来描述。

ftrace 大致有两个功能。一个包括插件追踪器（plugin tracer）分析的追踪功能。另一个是追踪内核事件的功能（将来通过引进 uprobes，也可能支持用户空间的追踪），称为追踪事件的。

插件追踪器中有一些是基于追踪事件的，但除了记录事件内容以外，还具有处理事件状态，显示整理结果的功能。在代表性的插件追踪器中，除了成为 ftrace 名字来源的函数追踪器以外，还有生成调用图表（call graph）的追踪器、MMIO（内存映射 I/O）的追踪器等。

另一方面，追踪事件是为了记录内核中代表性操作的状态而安装的追踪代码。通过记录这些追踪事件的信息，就可以记录内核中的代表性操作。

插件追踪器可以与追踪事件同时使用，因此除了插件追踪器的信息以外还可以使用追踪事件。

## 创建启用 ftrace 的内核

在最近的 Fedora 或 Ubuntu 等发布版的内核中多数是启用了 ftrace 的，但在想使用最新内核其 ftrace 未启用，就需要启动下列选项重新构建内核（重新构建的方法请参考 Hack #2）。

这些选项都在 Kernel Hacking ->Tracers 下。在表 8-3 所示的选项中，可以选择 CONFIG\_BRANCH\_PROFILE\_NONE、CONFIG\_PROFILE\_ANNOTATED\_BRANCHES、CONFIG\_PROFILE\_ALL\_BRANCHES。CONFIG\_PROFILE\_ANNOTATED\_BRANCHES 可以通过替换内核中的所有 unlikely、likely 宏，可以检查 unlikely 和 likely 的预测是否正确。CONFIG\_PROFILE\_ALL\_BRANCHES 可以替换内核中的所有 if() 语句，检查哪个分支多被分配到哪里。大家也可以预见到，这些选项将造成很大的系统开销，需要注意。

表 8-3 ftrace 的配置选项

选项名称	功能
CONFIG_FTRACE	启用 ftrace 自身的功能
CONFIG_FUNCTION_TRACER	启用函数追踪器

选项名称	功 能
CONFIG_FUNCTION_GRAPH_TRACER	启用函数调用图表追踪器
CONFIG_IRQSOFF_TRACER	启用禁止中断时间追踪器
CONFIG_SCHED_TRACER	启用调度程序追踪器
CONFIG_BRANCH_PROFILE_NONE	不启用条件分支追踪器功能
CONFIG_PROFILE_ANNOTATED_BRANCHES	启用分支预测追踪器
CONFIG_PROFILE_ALL_BRANCHES	启用条件分支追踪器
CONFIG_BLK_DEV_IO_TRACE	启用块 I/O 追踪器
CONFIG_MMIOTRACE	启用 MMIO 追踪器
CONFIG_FUNCTION_PROFILER	启用函数分析追踪器
CONFIG_STACK_TRACER	启用栈使用量追踪器
CONFIG_FTRACE_SYSCALLS	启用系统调用的追踪事件
CONFIG_KPROBE_EVENT	启用动态追踪事件

## 操作 ftrace 的 debugfs 接口

ftrace 的用户界面是在 debugfs 这个特殊的文件系统上生成的。debugfs 多数情况下在标准配置中是未挂接的，因此多数情况下需要手动挂载到 /sys/kernel/debug（相关追踪人员为了打字方便，多数会创建 /debug 目录来挂接，这里使用的是内核标准的挂接位置）。

```
[~] # mount -t debugfs debugfs /sys/kernel/debug
```

ftrace 的目录为 /sys/kernel/debug/tracing。在这个目录下有 ftrace 的相关特殊文件。这里介绍一些具有代表性的文件（见表 8-4 ~ 表 8-6）。

表 8-4 控制整体追踪运行情况的文件

文 件 名	说 明
tracing_on	启用向追踪缓冲区写入的功能。1 为启用。0 为禁用
tracing_enabled	启用插件追踪器。1 为启用。0 为禁用
available_tracers	显示当前内核中可使用的插件追踪器。关于各个插件追踪器请参考表 8-8
current_tracer	指定想要使用的插件追踪器
trace_options	指定追踪的文本格式的调整运行选项。想要取消设置时，执行以“no”开头的选项

文件名	说明
tracing_cpumask	以十六进制的位掩码指定要作为追踪对象的处理器。例如，指定 0xb 时仅在处理器 0、1、3 上进行追踪
set_ftrace_pid	指定要作为追踪对象的进程的 PID

表 8-5 追踪缓冲区的相关文件

文件名	说明
trace	以文本格式输出内核中追踪缓冲区的内容
trace_pipe	与 trace 相同，但是运行时像管道一样，可以在每次事件发生时读出。但是读过的内容就不能再次读出
buffers_size_kb	以 KB 为单位指定各 CPU 的追踪缓冲区大小。系统整体的缓冲区大小就是这个值乘以 CPU 数量。缓冲区以页面为单位分配，但是有一些字节用于缓冲区的管理结构，因此实际分配的缓冲区比指定的值更大。指定追踪缓冲区大小时，必须设置为 nop 追踪器

表 8-6 函数追踪器的相关文件

文件名	说明
set_ftrace_filter	指定追踪调用的函数名称。函数名称仅可以包含 1 个通配符 (wildcard)
set_ftrace_notrace	指定不追踪调用的函数名称

表 8-4 中有控制向追踪缓冲区写入的 tracing\_on 和控制插件追踪器的 tracing\_enabled 这两个文件。这个文件的值的组合情况如表 8-7 所示。这些看起来不太容易发现运行的不同之处，因此 tracing\_enabled 在最新的内核（Linux 2.6.39 以后）中成为废弃（deprecated）对象。今后将只使用 tracing\_on。

表 8-7 tracing\_on 和 tracing\_enabled

tracing_on	tracing_enabled	运 行
0	0	插件追踪器禁用。追踪事件的信息也不写入缓冲区
0	1	插件追踪器启用，但插件追踪器和追踪事件二者的 信息不写入缓冲区
1	0	插件追踪器禁用，但追踪事件的信息写入缓冲区
1	1	插件追踪器启用，追踪事件的信息也写入缓冲区

显示表 8-4 中的 available\_tracers，可以看出存在多个插件追踪器。ftrace 中可以使用的代表性插件追踪器如表 8-8 所示。

表 8-8 插件追踪器的种类

种    类	说    明
function	记录函数调用（仅调用）的追踪器。可以看出哪个函数何时调用
function_graph	记录函数的调用图表的追踪器。可以得知哪个函数被哪个函数调用，何时返回
mmiotrace	记录 MMIO（Memory Mapped I/O，内存映射 I/O）的运行的追踪器。用于 Nouveau 驱动程序等逆向工程（reverse engineering）
blk	调查块 I/O 运行的追踪器
wakeup	记录进程的调度延迟较长的
wakeup_rt	与 wakeup 相同，但以实时进程为对象
irqsoff	记录禁止中断期间花费时间最多的
preemptoff	记录禁止优先权期间花费时间最多的
preemptirqsoff	记录禁止优先权和中断的期间花费时间最多的
sched_switch	进行上下文切换的追踪。可以得知从哪个进程切换到了哪个进程
nop	不执行任何操作。不使用插件追踪器时指定

下面使用这些接口尝试实际使用 ftrace。

## 使用 ftrace 追踪函数调用

顾名思义，ftrace 原本是记录函数调用时间而开发出来的。首先介绍它的使用方法。作为示例，尝试追踪 Linux 的进程调度程序是从哪个函数调用的。首先根据 available\_tracers 的内容确认是否支持记录函数调用的函数追踪器。

```
[~] # cd /sys/kernel/debug/tracing
[tracing] # cat available_tracers
blk function_graph wakeup_rt wakeup irqsoff function sched_switch nop
```

函数追踪器为表 8-8 中的 function。available\_tracers 包含 function，因此可见在这个内核中可以使用函数追踪器。

一旦将函数追踪器启用，ftrace 就会记录所有函数的运行情况。但是本次只想查看调度程序函数 schedule() 的调用情况，因此事先使用追踪函数过滤器设置为仅记录 schedule()。

```
[tracing] # echo schedule > set_ftrace_filter
```

设置好追踪函数过滤器后，将函数追踪器 funciton 写入 current\_tracer 特殊文件。

```
[tracing] # echo function > current_tracer
```

追踪结果可以从 trace 特殊文件读出。这时的结果如下所示。

```
[tracing] # head trace
# tracer: function
```

```
#  
#      TASK-PID      CPU#      TIMESTAMP      FUNCTION  
#      | |  
tee-7470  [000] 16107.974550: schedule <-do_exit  
<idle>-0  [000] 16107.975318: schedule <-cpu_idle  
bash-5210  [000] 16107.976277: schedule <-schedule_timeout  
events/0-9  [000] 16107.976371: schedule <-worker_thread  
<idle>-0  [001] 16108.054539: schedule <-cpu_idle  
flush-8:0-284  [001] 16108.054557: schedule <-schedule_timeout
```

可以看出进程调度程序是从几个不同的函数调用的。

可以使用多个函数名称或通配符向过滤器指定模式、模块。例如，按照下列方式，就可以记录函数名以 irq 开头的所有函数调用。

```
[tracing] # echo 'irq*' > set_ftrace_filter
```

另外，在参数前面加上 :mod:，还可以仅追踪特定的内核驱动程序模块中包含的函数。在下例中，仅追踪 btrfs 模块中包含的函数。

```
[tracing] # echo :mod:btrfs > set_ftrace_filter
```

为了如前所述仅获取以 irq 开头的函数，需要更换过滤器。

```
[tracing] # echo 'irq*' > set_ftrace_filter  
[tracing] # echo 0 > trace  
[tracing] # head trace  
# tracer: function  
#  
#      TASK-PID      CPU#      TIMESTAMP      FUNCTION  
#      | |  
bash-1911  [000] 55477.514345: irq_enter <-smp_apic_timer_interrupt  
bash-1911  [000] 55477.514677: irq_exit <-smp_apic_timer_interrupt  
bash-1911  [000] 55477.514939: irq_enter <-smp_apic_timer_interrupt  
bash-1911  [000] 55477.514995: irq_exit <-smp_apic_timer_interrupt  
<idle>-0  [000] 55477.515953: irq_enter <-smp_apic_timer_interrupt  
<idle>-0  [000] 55477.516041: irq_exit <-smp_apic_timer_interrupt
```

可以看出记录了以 irq 开头的函数。切换了过滤器后，就向 trace 写入 0，这是为了将缓冲区内容暂时清除。仅通过切换过滤器是不能清除缓冲区内容的，因此需要这样手动清除缓冲区内容。

## 小结

本节介绍了什么是追踪、ftrace 的基本使用方法，以及实际使用函数追踪器找出特定内核函数的调用起点的方法。函数追踪器对于研究实际运行中系统函数调用关系非常有效。

## 参考文献

- Documentation/trace/ftrace.txt

——Masami Hiramatsu

# HACK #68 ftrace 的插件追踪器

本节介绍如何使用 ftrace 的插件追踪器进行内核的追踪。

ftrace 中有很多插件追踪器。这里介绍一些可用于内核调整或开发的追踪器，如进程调度程序和中断处理延迟的追踪、函数调用和栈使用情况的概要分析。

## 获取函数的调用关系

想要调查内核中特定部位的运行情况时，使用较多的是函数指针，有时会难以追踪源代码。在这种情况下，使用 ftrace 的函数调用图表追踪器和回溯追踪器可以更高效地研究调用的起点。

函数调用图表追踪器通过向 `current_tracer` 写入 `function_graph` 来运行。在这个追踪器中将体现 Hack #67 中介绍的 `set_ftrace_filter` 的设置，因此可以进行限定在一部分函数的追踪上。下面所示为仅追踪 ext4 文件系统的驱动程序内部的结果。

```
[~] # cd /sys/kernel/debug/tracing
[tracing] # echo :mod:ext4 > set_ftrace_filter
[tracing] # echo function_graph > current_tracer
[tracing] # head trace
# tracer: function_graph
#
#      TIME      CPU DURATION          FUNCTION CALLS
#      |      |      |      |
0)  5.781 us  |  ext4_write_inode();
1)  5.227 us  |  ext4_check_acl();
1)          |  ext4_check_acl() {
1)  0.615 us  |    ext4_get_acl();
1)  1.564 us  |
1)  1.536 us  |  ext4_check_acl();
```

各行开头的数字表示 CPU 编号。TIME 的位置显示的是执行该函数所花费的时间。以微秒为单位显示，超过 10 微秒的数值前面会显示 ‘+’，超过 100 微秒的数值前面显示 ‘!’。另外，函数名称按照函数调用的形式显示，使用缩进和大括号更为清晰地显示出调用关系。

另一方面，函数回溯是 ftrace 自身的扩展功能之一。需要运行事件追踪或函数追踪器，同时向 `options/func_stack_trace` 写入 1 来启动。这个功能在每次调用对象函数时进行回溯，因此需要事先使用 `set_ftrace_filter` 来缩小对象函数的范围。

```
[tracing] # echo schedule > set_ftrace_filter
[tracing] # echo function > current_tracer
[tracing] # echo 1 > options/func_stack_trace
[tracing] # echo 0 > trace
[tracing] # head -n 20 trace
# tracer: function
#
#      TASK-PID      CPU#      TIMESTAMP      FUNCTION
```

```

#                                | |      |      |      |
bash-1001  [000]  6371.701772: schedule <-schedule_timeout
bash-1001  [000]  6371.701774: <stack trace>
=> schedule_timeout
=> n_tty_read
=> tty_read
=> vfs_read
=> sys_read
=> system_call_fastpath
    <idle>-0      [000]  6371.701877: schedule <-cpu_idle
    <idle>-0      [000]  6371.701878: <stack trace>
=> cpu_idle
=> rest_init
=> start_kernel
=> x86_64_start_reservations
=> x86_64_start_kernel
    sshd-981      [000]  6371.702056: schedule <-schedule_hrtimer_timeout_
range_clock

```

对象函数的追踪入口后面出现了 `<stack trace>` 入口，它的后面显示的是回溯的结果。

## 进行函数的概要分析

使用 perf tools 的概要分析中，进行了基于采样的概要分析，而使用 ftrace 可以研究实际处理的函数调用次数或平均处理时间。

```

[tracing] # echo nop > current_tracer
[tracing] # echo 1 > function_profile_enabled
[tracing] # head trace_stat/function0
      Function          Hit      Time          Avg      s^2
-----  ---  -----  ---  -----
schedule          825  30261574 us  36680.69 us  1847361176 us
poll_schedule_timeout  124  8010094 us  64597.53 us  58239034039 us
schedule_hrtimer_timeout_range  124  8010001 us  64596.78 us  58238992673 us
schedule_hrtimer_timeout_range_clock  124  8009946 us  64596.34 us  58238973829 us
default_idle        483  6859897 us  14202.68 us  7136583263 us
native_safe_halt      483  6858505 us  14199.80 us  7136531144 us
sys_select           82  6652916 us  81133.13 us  84536914899 us
core_sys_select      82  6652773 us  81131.38 us  84536991091 us

```

Time 表示累计值，Avg 表示平均值， $s^2$  表示平均方差。但是这个值包含函数内休眠的时间和被该函数调用的其他函数的处理时间。这些时间可以分别使用 `options/sleep-time` 和 `options/graph-time` 来修正。

将 `options/sleep-time` 设置为 0，就可以在函数内修正休眠的时间。仅改变选项的话，就只是添加到之前的结果中。将 `function_profile_enabled` 先设置为 0 再恢复为 1，就可以重置为到这时为止的结果。

```

[tracing] # echo 0 > options/sleep-time
[tracing] # echo 0 > function_profile_enabled
[tracing] # echo 1 > function_profile_enabled
[tracing] # head trace_stat/function0

```

Function	Hit	Time	Avg	s^2
-----	---	---	---	---
schedule	1431	305067863 us	213185.0 us	3333149560 us
schedule_hrtimerout_range	326	305016911 us	935634.6 us	17783347272 us
schedule_hrtimerout_range_clock	326	305016701 us	935634.0 us	17783619420 us
poll_schedule_timeout	322	305016644 us	947256.6 us	6964013622 us
sys_poll	41	305003855 us	7439118 us	361774855912 us
do_sys_poll	41	305003347 us	7439106 us	361953750054 us
default_idle	1393	237675932 us	170621.6 us	11207125794 us
native_safe_halt	1393	237669316 us	170616.8 us	11206960460 us

同样的，将 `options/graph-time` 设置为 0，就可以修正调用其他函数的时间。

Function	Hit	Time	Avg	s^2
-----	---	---	---	---
native_safe_halt	66	2524613 us	38251.71 us	5833974146 us
native_apic_mem_write	171	3921.759 us	22.934 us	4288.712 us
read_pmtmr	509	3350.586 us	6.582 us	10.813 us
smp_apic_timer_interrupt	61	2381.892 us	39.047 us	19776.76 us
ack_APIC_irq	61	1932.587 us	31.681 us	17289.86 us
apic_write	136	1516.866 us	11.153 us	3312.367 us
_cond_resched	521	1233.437 us	2.367 us	2095.993 us
vp_interrupt	11	717.574 us	65.234 us	32837.94 us

## 调查占用内核栈最大的位置

ftrace 中有一个栈追踪功能，是在内核内部函数获取内核栈的消耗量，从消耗量最大的位置进行回溯的功能。内核栈是有限的（x86 中为约 8KB），因此栈的消耗量一旦超过上限，就有可能引起内存破坏或内核重大故障。

栈追踪器准确来说并不是 ftrace 的插件追踪器。但是由于接口使用的是 ftrace，因此本节将其作为 ftrace 功能之一进行介绍。使用栈追踪器时需要使用下列方法之一。

- 向 Linux 内核的启动选项传递 `stacktrace` 后启动。
- 向 `/proc/sys/kernel/stack_tracer_enabled` 写入 1。

这里使用第二个方法尝试进行栈追踪。

[tracing] # echo 1 > /proc/sys/kernel/stack_tracer_enabled			
[tracing] # cat stack_trace			
Depth	Size	Location	(10 entries)
0)	2688	464	find_busiest_group+0x129/0x90e
1)	2224	288	load_balance+0xbc/0x66a
2)	1936	192	schedule+0x302/0x681
3)	1744	160	schedule_hrtimerout_range_clock+0x52/0x11b
4)	1584	16	schedule_hrtimerout_range+0x13/0x15
5)	1568	48	poll_schedule_timeout+0x48/0x64
6)	1520	880	do_select+0x4e8/0x52c

```
7)      640      400      core_sys_select+0x174/0x213
8)      240      112      sys_select+0x96/0xbe
9)      128      128      system_call_fastpath+0x16/0x1b
```

与其他追踪器最大的不同在于使用 procfs 来控制启用或禁用，以及从 `stack_trace` 这个特殊文件而不是从 `trace` 进行读出。

`Depth` 表示该函数内最大的栈消耗量，`Size` 表示仅该函数上使用的栈消耗量。加大负载一段时间后再次读入 `stack_trace`，就会出现消耗了栈的执行路径。

## 测量中断的延迟

Linux 的进程会由于各种原因发生运行延迟。使用 ftrace 的延迟追踪器有时也可以找到调度延迟的原因。

在 ftrace 的插件追踪器中，`irqsoff`、`preemptoff`、`preemptirqsoff`、`wakeup`、`wakeup_rt` 分别为测量中断、优先权、调度的延迟时间的功能，统称为延迟追踪器（Latency Tracer）。（各自的内容请参考 Hack #67 的表 8-8）。

之所以开发 ftrace，是因为在开发实时内核（实际上是尽量缩短处理延迟的软实时（soft realtime）内核）时，出现了测量内核中的延迟时间、调查瓶颈在哪里的要求。这些延迟追踪器就是为了满足这个要求而开发的功能。

延迟追踪器测量的对象不同，但输出基本上是相同的。这里使用追踪禁止中断时间的追踪器作为示例。

```
[tracing] # echo irqsoff > current_tracer
[tracing] # echo 1 > tracing_on
[tracing] # echo 0 > tracing_on
[tracing] # cat trace
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.0.0-rc4+
# -----
# latency: 222 us, #294/294, CPU#0 | (M:desktop VP:0, KP:0, SP:0 HP:0 #P:4)
#
#     | task: docky-2157 (uid:1000 nice:0 policy:0 rt_prio:0)
#
#     => started at: schedule
#     => ended at:   schedule
#
#
#           -----> CPU#
#           /-----> irqs-off
#           |-----> need-resched
#           || /-----> hardirq/softirq
#           ||| /-----> preempt-depth
#           |||| /-----> delay
# cmd     pid    ||||| time    |    caller
# \   /    ||||| \    |    /
notify-o-2227  0d...    0us : _raw_spin_lock_irq <-schedule
```

```

notify-o-2227 0d... 1us : deactivate_task <-schedule
notify-o-2227 0d... 1us : dequeue_task <-deactivate_task
notify-o-2227 0d... 2us : update_rq_clock <-dequeue_task
notify-o-2227 0d... 3us : dequeue_task_fair <-dequeue_task
notify-o-2227 0d... 3us : update_curr <-dequeue_task_fair
notify-o-2227 0d... 4us : cpuacct_charge <-update_curr
notify-o-2227 0d... 4us : clear_buddies <-dequeue_task_fair
notify-o-2227 0d... 5us : update_cfs_load <-dequeue_task_fair
notify-o-2227 0d... 6us : update_cfs_shares <-dequeue_task_fair
...
notify-o-2227 0d... 219us : native_load_sp0 <-__switch_to
notify-o-2227 0d... 219us : native_load_tls <-__switch_to
notify-o-2227 0d... 220us : native_read_cr0 <-__switch_to
notify-o-2227 0d... 220us : native_write_cr0 <-__switch_to
docky-2157 0d... 221us : finish_task_switch <-schedule
docky-2157 0d... 222us : finish_task_switch <-schedule
docky-2157 0d... 223us : trace_hardirqs_on <-schedule
docky-2157 0d... 223us : <stack trace>
=> finish_task_switch
=> schedule
=> schedule_hrtimer_timeout_range_clock
=> schedule_hrtimer_timeout_range
=> poll_schedule_timeout
=> do_sys_poll
=> sys_poll
=> system_call_fastpath

```

显示了禁止中断期间最长、从开始禁止中断的位置到允许中断的位置之间的执行路径。与其他追踪器的输出相似，而不同之处在于其中有 6 个标志，且显示的不是从记录的时间开始而是从开始禁止中断开始的延迟时间。

6 个标志显示的信息分别是 CPU 编号、是否禁止中断 (d)、是否必须重新调度对象进程 (N)、是不是硬中断 (h) 或软中断 (s)、锁定了几个优先权计数、BKL。

## 小结

本节通过一些实例介绍了 ftrace 的追踪器。Ftrace 源于函数追踪器的功能，但现在已经整合了众多追踪器和追踪功能的接口，也成为 Linux 标准的追踪框架。根据不同的需要选择使用 ftrace 的追踪器，可以对系统或内核的开发起到很大的作用。

## 参考文献

- Secrets of the Ftrace function tracer  
<http://lwn.net/Articles/370423/>

——Masami Hiramatsu

# HACK #69 记录内核的运行事件

本节介绍使用 ftrace 记录追踪事件的方法。

追踪事件是以过去称为追踪点（即，非常小型的追踪器调用的钩子（hook））功能为基础，在内核中的代表性操作位置上安装用来记录该处理的状态的追踪事件。通过记录这些追踪事件的内容，就可以记录内核中的典型性操作。在 Linux 2.6.28 中安装了这个追踪事件功能以后，通过内核内部的各种处理定义了追踪事件，使用户能够通过 ftrace 或 perf tools 等观测到这些事件。Linux 内核中已经安装了 100 多个追踪点，追踪事件的数量超过了 200 个（也有系统调用事件等从一个追踪点生成多个追踪事件的情况）。

本节将介绍从 ftrace 获取这些事件的方法、与 ftrace 的其他追踪器合作记录事件的方法及其效果。

首先，尝试调查内核中的中断处理程序何时运行。与中断处理程序相对应的事件为 `irq:irq_handler_entry` 和 `irq:irq_handler_exit`，因此可以使用下列命令进行追踪。

```
[~] # cd /sys/kernel/debug/tracing
[tracing] # cd /sys/kernel/debug/tracing
[tracing] # echo irq:irq_handler_entry > set_event
[tracing] # echo irq:irq_handler_exit >> set_event
[tracing] # cat trace
...
      kblockd/0-20  [000]  6854.611683: irq_handler_entry: irq=21 name=ahci
      kblockd/0-20  [000]  6854.611735: irq_handler_exit: irq=21 ret=handled
      kblockd/0-20  [000]  6854.611736: irq_handler_entry: irq=21 name=Intel
82801AA-ICH
      kblockd/0-20  [000]  6854.611760: irq_handler_exit: irq=21 ret=unhandled
      <idle>-0    [000]  6854.614767: irq_handler_entry: irq=16 name=virtio1
      <idle>-0    [000]  6854.614779: irq_handler_exit: irq=16 ret=handled
      dbus-daemon-839 [000]  6854.630395: irq_handler_entry: irq=12 name=i8042
      dbus-daemon-839 [000]  6854.630417: irq_handler_exit: irq=12 ret=handled
...

```

可以看出系统上发生的中断通过各种子系统得到填补和处理。下面更加详细、集中地查看特定的中断（21 号）。

```
[tracing] # echo "irq==21" > events/irq/irq_handler_entry/filter
[tracing] # echo "irq==21" > events/irq/irq_handler_exit/filter
[tracing] # echo 0 > trace
[tracing] # cat trace
...
      jbd2/sda1-8-312  [000]  6883.399116: irq_handler_entry: irq=21 name=ahci
      jbd2/sda1-8-312  [000]  6883.399167: irq_handler_exit: irq=21 ret=handled
      jbd2/sda1-8-312  [000]  6883.399169: irq_handler_entry: irq=21 name=Intel
82801AA-ICH
      jbd2/sda1-8-312  [000]  6883.399202: irq_handler_exit: irq=21 ret=unhandled
      jbd2/sda1-8-312  [000]  6883.399374: irq_handler_entry: irq=21 name=ahci
      jbd2/sda1-8-312  [000]  6883.399550: irq_handler_exit: irq=21 ret=handled
...

```

例如，在上述示例中，可以看出在系统启动后的 6883.399 秒左右，jbd2/sda1-8-312 进程（在本例中为内核线程）在 CPU0 ([000]) 上运行，这时发生了中断处理。此外，还可以读出想要在 ahci 或 Intel 82801AA-ICH 设备上处理 IRQ 编号 21 的中断，且 ahci 处理已成功。

要删除已经设置的过滤器，可以在过滤器规则中写入“0”。需要注意的是，虽然默认的过滤器规则中所写的是“none”，但如果写入“none”就会出错。

```
[tracing] # echo 0 > events/irq/irq_handler_entry/filter
[tracing] # echo 0 > events/irq/irq_handler_exit/filter
```

接下来介绍更为详细的设置方法等。

## 调查可使用的追踪事件

要查看当前运行在内核中可追踪的事件，可以查看 /sys/kernel/debug/tracing/ 目录下的特殊文件 available\_events 的内容，或者在 /sys/kernel/debug/tracing/events/ 目录下列举下列内容。

```
[tracing] # less available_events
kvmmmu:kvm_mmu_pagetable_walk
kvmmmu:kvm_mmu_paging_element
kvmmmu:kvm_mmu_set_accessed_bit
kvmmmu:kvm_mmu_set_dirty_bit
kvmmmu:kvm_mmu_walker_error
kvmmmu:kvm_mmu_get_page
...
[tracing] # find events/
events/
events/kvmmmu
events/kvmmmu/kvm_mmu_pagetable_walk
events/kvmmmu/kvm_mmu_pagetable_walk/format
events/kvmmmu/kvm_mmu_pagetable_walk/filter
events/kvmmmu/kvm_mmu_pagetable_walk/id
events/kvmmmu/kvm_mmu_pagetable_walk/enable
...
```

各追踪事件具有组名和事件名。available\_events 的内容格式为“组名:事件名”。而在 events 目录下，首先有与组相对应的目录，其下有与事件相对应的目录。与只有事件列表的 available\_events 不同的是，events 目录下存在用来获取更为详细的信息与进行控制的特殊文件（见表 8-9）。

表 8-9 事件的特殊文件

文件名	内 容
id	记录事件的 ID 编号。只读
format	记录 ftrace 缓冲区上的事件入口的详细格式。只读

文件名	内 容
enable	决定是否记录这个事件的标志。只读
filter	记录这个事件的条件的过滤器

分组目录下也存在 filter 和 enable 特殊文件。这些特殊文件可以在以组为单位记录多个事件或进行过滤时使用。

## 调查事件的格式

可以从各事件的 format 特殊文件中查看各事件中记录的信息和该信息的显示格式等。

```
[tracing] # cat events/irq/irq_handler_entry/format
name: irq_handler_entry
ID: 98
format:
    field:unsigned short common_type;      offset:0;      size:2; signed:0;
    field:unsigned char common_flags;      offset:2;      size:1; signed:0;
    field:unsigned char common_preempt_count;  offset:3;      size:1;
signed:0;
    field:int common_pid;    offset:4;      size:4; signed:1;
    field:int common_lock_depth;  offset:8;      size:4; signed:1;

    field:int irq;  offset:12;      size:4; signed:1;
    field:_data_loc char[] name;  offset:16;      size:4; signed:1;

print fmt: "irq=%d name=%s", REC->irq, __get_str(name)
```

这个格式中同时记录了 ftrace 将事件记录到缓冲区时的二进制格式信息、用户将事件读出为文本信息时的格式信息（见表 8-10）。

表 8-10 事件格式的内容

项 目	内 容
name	事件的名称
ID	事件 ID 编号
format	二进制格式项的结构
field	关于二进制格式项内的变量信息
print fmt	ftrace 输出文本时的格式

二进制格式信息在 field: 后面分别显示各参数的（C 语言的）类型名称、项上的变量名称、离入口开头的字节偏移量、字节大小、变量是否带符号。各参数的类型名称基本与 C 语言的类型名称相同，因此这个入口的内存映像与下列结构基本相同。

```
struct irq_handler_entry_structure {
    unsigned short common_type;
    unsigned char common_flags;
```

```

unsigned char common_preempt_count;
int common_pid;
int common_lock_depth;

int irq;
char *name;
} __attribute__((__packed));

```

可以看出，字段的定义分为上面 5 个字段和其他字段。空行上方为所有事件共同的变量，空行下方为这个事件特有的变量。追踪事件共同的变量具有下列意义（见表 8-11）。

表 8-11 追踪事件共同变量

变    量	说    明
common_type	数据入口的类型 ID。插件追踪器各自具有不同的数据项类型 ID。追踪事件的每个不同事件都具有类型 ID
common_flags	事件发生时标志寄存器的值
common_preempt_count	事件发生时优先权计数的值
common_pid	事件发生时当前进程的 PID
common_lock_depth	事件发生时当前进程获取了多少 BKL (Big Kernel Lock) (仅 CONFIG_BKL=y 时存在)

如果有内核的源代码，还可以查看各事件特有变量的意义。例如，`irq_handler_entry` 事件在内核源码树的 `include/trace/events/irq.h` 下写有下列命令。

```

/**
 * irq_handler_entry - called immediately before the irq action handler
 * @irq: irq number
 * @action: pointer to struct irqaction
 *
 * The struct irqaction pointed to by @action contains various
 * information about the handler, including the device name,
 * @action->name, and the device id, @action->dev_id. When used in
 * conjunction with the irq_handler_exit tracepoint, we can figure
 * out irq handler latencies.
 */

```

但是，这里有一点需要注意。上述注释并不是 `irq_handler_entry` 追踪事件的注释，而是内核内部使用的 API 的注释，因此参数有一些不同。要调查各追踪事件的参数相当于 API 的哪个参数，需要阅读各个事件的代码。在这个示例中，备注中的 `@action->name` 在追踪事件中为 `name`。

## 控制事件

事件可以分别设置为启用（记录）、禁用（不记录），或者设置过滤。另外，也可以以事件组为单位进行统一控制。

记录 / 不记录的控制可以通过操作 `set_events` 或各事件、各组的 `enable` 特殊文件来

进行。以与 `available_events` 相同的“组名:事件名”的格式向 `set_events` 写入想要开始记录的事件列表。

```
[tracing] # cat > set_events
irq:irq_handler_entry
irq:irq_handler_exit
^D
```

使用 `events` 目录下的 `enable` 特殊文件的方法是向各事件或各组写入启用或禁用选项。

```
[tracing] # echo 1 > events/irq/irq_handler_entry/enable
[tracing] # echo 1 > events/irq/irq_handler_exit/enable
```

不管使用的是哪个方法，在其中一个接口上设置的状态都会自动反映到另一个接口上，因此一般来说可以这样使用。

- 想要从命令行添加事件时，使用 `events` 目录。
- 想要保存已添加事件的设置时，或想要将已保存的设置恢复时，使用 `set_events`。

例如，还可以暂时保存事件，并发送到想要获取同一事件的其他机器后再写入。

手头的机器

```
[tracing] # cat set_events > /tmp/event_set1
[tracing] # scp /tmp/event_set1 remote:~/
```

远程机器

```
[tracing] # cat ~/event_set1 > set_events
```

而事件过滤器是使用 `events` 目录下的 `filter` 特殊文件进行设置的。事件过滤器是通过事件发生时的参数值来设置是否记录的功能。参数大致可以分为数值参数和字符串参数，各自可使用的比较运算符有所不同（见表 8-12）。过滤器的比较表达式必须是左边为参数名，右边为值。也就是说，可以指定 `irq<=10`，但不能指定 `10>=irq`。

表 8-12 事件过滤器中可使用的比较运算符

比较运算符	适用对象	说 明
<code>=</code>	数值、字符串	参数的值等于给出的值则记录
<code>!=</code>	数值、字符串	参数的值不等于给出的值则记录
<code>~</code>	字符串	将参数与字符串进行比较，如果一致则记录。可以将通配符（“*”）用做字符串的一部分
<code>&lt;</code>	数值	参数的值小于给出的值则记录
<code>&lt;=</code>	数值	参数的值等于或小于给出的值则记录
<code>&gt;</code>	数值	参数的值大于给出的值则记录
<code>&gt;=</code>	数值	参数的值等于或大于给出的值则记录

另外，也可以使用逻辑运算符和小括号组合多个过滤器。例如，在参数 `irq` 的值介于 10 ~ 20 或 `name` 以 `ata` 开头的情况下，想要记录事件时的过滤器如下所示。

```
(irq>=10 && irq<20) || name-ata*
```

另外, ftrace 有一个特点, 就是文本输出的追踪结果即使看起来像字符串, 有些参数也只是在输出文本时在内部进行了转换, 它们实际还是数值。对于这样的参数不能使用用于字符串参数的比较运算符。

```
[tracing] # echo "ret==unhandled" > events/irq/irq_handler_exit/filter # 这个出错
bash: echo: write error: Invalid argument
[tracing] # echo "ret==0" > events/irq/irq_handler_exit/filter # 这个没问题
```

要找出这样的疑似字符串参数, 就需要阅读 format 特殊文件。

```
[tracing] # cat events/irq/irq_handler_exit/format
name: irq_handler_exit
ID: 97
format:
    field:unsigned short common_type;      offset:0;      size:2; signed:0;
    field:unsigned char common_flags;      offset:2;      size:1; signed:0;
    field:unsigned char common_preempt_count;  offset:3;      size:1;
    signed:0;
    field:int common_pid;      offset:4;      size:4; signed:1;
    field:int common_lock_depth;  offset:8;      size:4; signed:1;

    field:int irq;      offset:12;      size:4; signed:1;
    field:int ret;      offset:16;      size:4; signed:1;

print fmt: "irq=%d ret=%s", REC->irq, REC->ret ? "handled" : "unhandled"
```

根据 format 可以看出, 参数 ret 记录为 int 类型, 在 print fmt 中, 如果 ret!=0 则显示为 handled, 如果 ret==0 则显示为 unhandled。过滤器可以适用于 format 部分所示格式的数据, 因此 ret 必须作为数值参数。

## 使用 ftrace 的事件加强其他的追踪器输出

ftrace 的追踪事件可以与其他插件追踪器组合输出。这样就可以更清晰地查看输出量较多的函数追踪器的结果等。

例如, 尝试同时使用进程调度程序的相关事件和函数追踪器, 就可以得知进程的切换处理和函数调用关系等。在下例中实际显示了从发生 sched\_switch 事件到进程切换为止的步骤, 可以更加详细地了解事件前后发生的情况。

```
[tracing] # echo 'sched:*' > set_event
[tracing] # echo 'function' > current_tracer
[tracing] # cat trace
...
    less-3569  [001] 16957.997983: sched_stat_wait: comm=events/1 pid=10
delay=53796 [ns]
        less-3569  [001] 16957.997984: task_of <-pick_next_task_fair
        less-3569  [001] 16957.997984: hrtick_start_fair <-pick_next_
task_fair
        less-3569  [001] 16957.997985: __raw_local_save_flags <-ftrace_
raw_event_sched_switch
```

```

        less-3569  [001] 16957.997985: sched_switch: prev_comm=less
prev_pid=3569 prev_prio=120 prev_state=R ==> next_comm=events/1 next_pid=10
next_prio=120
        less-3569  [001] 16957.997985: atomic_inc <-schedule
        less-3569  [001] 16957.997986: enter_lazy_tlb.clone.16
<-schedule
        less-3569  [001] 16957.997986: native_load_sp0 <-__switch_to
        less-3569  [001] 16957.997987: load_TLS <-__switch_to
        less-3569  [001] 16957.997987: native_load_tls <-load_TLS
        less-3569  [001] 16957.997988: __unlazy_fpu <-__switch_to
        less-3569  [001] 16957.997988: read_cr0 <-__unlazy_fpu
        less-3569  [001] 16957.997988: native_read_cr0 <-read_cr0
        less-3569  [001] 16957.997989: native_write_cr0 <-__unlazy_fpu
events/1-10      [001] 16957.997995: finish_task_switch <-schedule
events/1-10      [001] 16957.997999: raw_local_irq_enable <-finish_task_
switch

```

另外，还可以使用事件的参数，研究函数调用的终点实际进行的处理和处理所花费的时间之间的关系。

```

[tracing] # echo do_IRQ > set_ftrace_filter
[tracing] # echo 1 > events/irq/irq_handler_entry/enable
[tracing] # echo function_graph > current_tracer
[tracing] # cat trace
...
0) ======> |
0)           | do_IRQ() {
0)           | /* irq_handler_entry: irq=19 name=ehci_hcd:usb1 */
0)           | /* irq_handler_entry: irq=19 name=virtio0 */
0) ! 244.859 us | }
0) ======> |
0)           | do_IRQ() {
0)           | /* irq_handler_entry: irq=21 name=ahci */
0)           | /* irq_handler_entry: irq=21 name=Intel 82801AA-ICH */
0) ======> |
0)           | do_IRQ() {
0)           | /* irq_handler_entry: irq=21 name=ahci */
0)           | /* irq_handler_entry: irq=21 name=Intel 82801AA-ICH */
0) ! 286.214 us | }
0) ======> |
0)           | do_IRQ() {
0)           | /* irq_handler_entry: irq=21 name=ahci */
0)           | /* irq_handler_entry: irq=21 name=Intel 82801AA-ICH */
0) ! 620.506 us | }
0) ======> |

```

例如，这里就使用 do\_IRQ 的函数调用图表显示了处理所花费的时间（出现了超过 100 微秒的数值，因此数值前面显示了 ‘! ’），但是仅使用函数调用图表难以调查出哪个中断进行了什么处理。但是事件会显示各个中断的处理方式，这样就可以很简单地得知启动了哪个中断处理程序，以及此时花费了多少时间。

## 小结

本节介绍了从 ftrace 记录追踪事件的方法、追踪事件特有的过滤器的设置方法，以及通过将追踪事件和插件追踪器组合使用来帮助分析插件追踪器结果的方法。追踪事件可以

获取事件发生时的变量值等状态，因此在调试中可以起到非常大的作用。

## 参考文献

- Documentation/trace/ftrace.txt
- Documentation/trace/events.txt

——Masami Hiramatsu

# HACK #70 使用 trace-cmd 的内核追踪

本节介绍使用 trace-cmd 工具简单地操作 ftrace 的方法。

本章将介绍关于追踪的知识。

为了能够简单地使用 ftrace 的功能并加以扩展，可以使用 trace-cmd 工具。本节将介绍使用 trace-cmd 代替 ftrace 获取内核追踪的方法。

## trace-cmd 的获取与创建

ftrace 是不需要使用特别的命令，就可以轻松地追踪内核的运行情况，但是每次更改追踪条件时就需要对多个特殊文件进行设置，为了找出故障或处理延迟的原因而扩大或缩小条件，就需要花费很多时间和精力。

在这种情况下，为了能够方便地从命令行使用 ftrace，就开发了 trace-cmd 工具。trace-cmd 在有些发布版中还未作为工具包安装，因此需要通过从开发数据仓库获取的源代码来创建并使用。

trace-cmd 的数据仓库为 `git://git.kernel.org/pub/scm/linux/kernel/git/rostedt/trace-cmd.git`。可以执行下列命令来获取和创建。

```
# git clone git://git.kernel.org/pub/scm/linux/kernel/git/rostedt/trace-cmd.git
# cd trace-cmd
# make
# make install
```

trace-cmd 虽然支持 Python 脚本的绑定 (binding)，但要启用这个功能必须事先安装 `python-dev`、`swig`。默认将命令安装到 `/usr/local/` 下。

## 使用 trace-cmd 进行追踪

trace-cmd 命令中也有一些子命令，但只要有 trace-cmd record 命令和 trace-cmd report，就可以使用 ftrace 的基本功能。例如，下列命令就可以同时与记录函数追踪器和进程调度程序相关的事件。（停止追踪时按 Ctrl+C 快捷键。）

```
# trace-cmd record -p function -e 'sched:*
```

向 `trace-cmd record` 的 `-p` 选项指定插件追踪器的名称，与向 `ftrace` 的 `current_tracer` 指定的内容相同。按照向 `ftrace` 的 `set_event` 指定的相同格式，向 `-e` 选项指定事件名称。事件的过滤器可以使用 `-f` 选项来指定。指定多个事件时，要在事件后面接着指定过滤器。例如，下列命令就是向 `irq_handler_entry` 事件指定 “`irq>=10`” 过滤器，向 `irq_handler_exit` 事件指定 “`ret==0`”。

```
# trace-cmd record -e irq_handler_entry -f "irq>=10" -e irq_handler_exit -f "ret==0"
```

在 `trace-cmd` 中还可以进行仅使用 `ftrace` 时难以控制的指定。在执行特定的用户命令期间，也可以仅获取该进程相关的事件。例如，下列命令仅在执行 `ls` 命令期间运行函数追踪器和进程调度程序的相关事件，仅追踪 `ls` 命令的相关事件 (`-F` 选项)。`ls` 命令结束后，追踪也自动结束。

```
# trace-cmd record -p function -e 'sched:*' -F ls
```

获取的数据记录在执行目录的 `trace.dat` 文件中（记录的位置可以使用 `-o` 选项来更改）。这个数据文件包含获取追踪时的系统信息（定义的事件或内核的符号信息等）。另外，追踪数据是作为二进制数据记录的，因此要使用 `trace-cmd report` 将内容进行整理再读出。

```
# trace-cmd report
version = 6
cpus=2
          ls-25316 [000] 7031.618254: function:           fsnotify_
modify <-- vfs_write
          ls-25316 [000] 7031.618258: function:           inotify_
inode_queue_event <-- fsnotify_modify
          ls-25316 [000] 7031.618258: function:           fsnotify_
parent <-- fsnotify_modify
          ls-25316 [000] 7031.618258: function:           —
fsnotify_parent <-- fsnotify_parent
          ls-25316 [000] 7031.618259: function:           inotify_
dentry_parent_queue_event <-- fsnotify_parent
          ls-25316 [000] 7031.618259: function:           fsnotify
<-- fsnotify_modify
          ls-25316 [000] 7031.618260: function:           fput_
light <-- sys_write
          ls-25316 [000] 7031.618260: function:           audit_
syscall_exit <-- sysret_audit
...
```

## 使用 `trace-cmd` 进行后台追踪

使用 `trace-cmd record` 时，不能像直接使用 `ftrace` 时在后台不断记录。另外，由于总是将追踪的结果写出到文件，因此追踪过程的系统开销也会变大。系统管理人员经常建议希望随时将追踪放到内存中，并以发生某种情况为契机，将到此时为止记录的追踪数据的最新信息写出到文件。`trace-cmd` 中使用 `trace-cmd start`、`trace-cmd extract`、`trace-cmd stop` 来进行这个操作。

trace-cmd start 和 trace-cmd extract 中使用的参数与 trace-cmd record 相同。trace-cmd start 具有除了文件输出或进程执行相关选项以外的选项。相反，trace-cmd extract 具有文件输出的相关选项。

在下例中，向 irq\_handler\_entry 事件指定 “irq>=10” 过滤器，向 irq\_handler\_exit 事件指定 “ret= =0” 过滤器，并将追踪结果记录到内核的内存中。

```
# trace-cmd start -e irq_handler_entry -f "irq>=10" -e irq_handler_exit -f "ret= =0"
```

命令程序已经结束，但追踪程序仍在后台运行。执行中的追踪内容可以从 /sys/kernel/debug/tracing/trace 特殊文件阅读。如果发生异常情况，可以使用 trace-cmd stop 停止追踪，然后使用 trace-cmd extract 写到文件中。

(发生情况)

```
# trace-cmd stop
# trace-cmd extract -o result.data
# trace-cmd report -i result.data
```

在不执行 trace-cmd stop 命令的情况下，也可以执行 trace-cmd extract，但这种情况下追踪数据将不断写入文件直到按下 Ctrl+C 快捷键为止。

另外需要注意的是，trace-cmd extract 命令会在提取追踪数据的同时进行追踪的初始化（完成）。也就是说，一旦执行 trace-cmd extract，在此之前运行的追踪的设置就会从内核中消失。

因此，由于发生异常情况想要临时获取快照文件时，建议按下列方法直接使用 ftrace 的 trace。

(发生情况)

```
# echo 0 > /sys/kernel/debug/tracing/tracing_on
# cp /sys/kernel/debug/tracing/trace snapshot1.log
# echo 1 > /sys/kernel/debug/tracing/tracing_on
```

在复制的前停止追踪，是为了防止混入关于复制行为自身的无用追踪数据。

## 使用 trace-cmd 结束追踪

从命令行操作 ftrace 时，结束追踪时可能会忘记返回参数，过滤器等设置的初始化也非常麻烦。如果执行 trace-cmd reset，就可以将这些参数全部初始化，同时还将删除内核内存中的旧追踪记录。

另外，如前所述，使用 trace-cmd stop 结束追踪时，实际上内核内部的追踪设置也未完全初始化。也就是说，current\_tracer 和 set\_event 都是正在追踪的设置。作为代替，trace-cmd stop 中使用 tracing\_on 特殊文件来开始或结束追踪。tracing\_on 特殊文件不改变追踪器自身的运行，只是将追踪器向追踪缓冲区写入的处理方式禁用，这样的话追踪器的系统开销是未完全消失的。因此，在结束追踪时，最后必须执行 trace-cmd reset 命令。

## 使用 trace-cmd 获取远程机器的追踪

trace-cmd 中具有跨网络进行远程机器的 ftrace 操作、接收数据的功能。在接收数据的本地机器上运行 trace-cmd listen 命令，在远程机器上运行 trace-cmd record，将追踪信息传送到本地机器的 trace-cmd。

在通过 trace-cmd 记录的文件中，同时还记录了追踪器运行的机器的字节顺序 (endian) 等，即使是不同架构的机器之间也可以从远程进行追踪。从而也可以以嵌入式机器这样本地存储较小的机器将追踪信息提取到开发机器上进行分析。

要获取远程机器的追踪，首先在保存追踪数据的机器上以等待远程连接的模式启动 trace-cmd。这时必须要设置等待端口号。此外，如果添加 -D 选项，还可以在后台作为守护进程运行。例如，在下例中是在 11223 号端口等待追踪会话 (session)。

```
[host]# trace-cmd listen -p 11223
```

然后，在实际想获取追踪的机器上，向 trace-cmd record 的 -N 选项指定“主机名称：端口编号”并执行。例如，在下例中是连接到 host 的 11223 号端口，传送执行 ls 命令期间的函数追踪结果。

```
[client]# trace-cmd record -N host:11223 -p function ls
```

这样就会在 host 的界面上出现下列显示结果。

```
connected!
Connected with client:33186
cpus=4
pagesize=4096
version = 6
offset=31b000
offset=36d000
offset=3ce000
offset=483000
connected!
```

在主机的本地目录下就会生成 trace. 客户端名 : 端口 .dat 数据文件。要分析这个数据文件，需要如下例所示在 trace-cmd report 的 -i 选项中指定数据文件。

```
[host]# trace-cmd report -i trace.client:33186.dat
version = 6
cpus=4
        <...>-5453  [000]  7366.159374: function:           down_read <-- m_start
        <...>-5454  [003]  7366.159374: function:           __fsnotify_parent <--
vfs_write
        <...>-5453  [000]  7366.159375: function:           _cond_resched <--
down_read
        <...>-5454  [003]  7366.159375: function:           fsnotify <-- vfs_write
        <...>-5454  [003]  7366.159375: function:           __srcu_read_lock <--
fsnotify
        <...>-5453  [000]  7366.159375: function:           seq_list_start <-- m_
start
...
```

追踪的客户端的磁盘上未写入任何数据文件，因此可以用来在安装机器等保存追踪数据的空间较小的机器上获取追踪。但是由于是经由一般的 TCP 通信进行数据传输的，如果是在安全性有问题的环境中，还需要使用 VPN 或 SSH 端口映射（Port Forward）等。

## 小结

本节介绍了在使用 `trace-cmd` 的 `trace` 操作中，使用 `record/report` 进行记录、分析，使用 `start/extract` 在后台获取追踪，使用 `listen` 向远程机器传送追踪数据的三种方法。通过使用 `trace-cmd`，就不需要考虑 `debugfs` 的接口，可以简便地进行追踪。

——Masami Hiramatsu

# HACK #71 将动态追踪事件添加到内核中

本节使用 `ftrace` 和 `perf probe`，将动态追踪事件添加到内核中。

Hack #69 介绍了使用 `ftrace` 记录追踪事件的方法。本节将介绍使用 Linux 2.6.33 开始添加的新结构“动态追踪事件”，简单方便地添加追踪事件的方法。

## 动态追踪事件

从 Linux 2.6.33 开始，`ftrace` 和 `perf tools` 中开始添加了使用 `kprobes` 的动态追踪事件功能。通过使用这个功能，就可以在内核运行过程中添加新的追踪事件。这个功能的优点在于即使添加追踪事件也不需要重新构建或者重新启动内核。另外，通过使用 `perf tools` 的 `probe` 子命令，还可以添加源代码层次的追踪事件。可以看做是调试程序（debugger）的断点（break point）这样的功能。在动态追踪事件中还可以获取寄存器或栈的值（进而获取相应的内存值）。

动态追踪事件与其他内核内部事先定义的追踪事件一样，可以针对每个定义的事件进行启用或过滤。还可以使用 `perf probe`，在源代码层次指定事件发生位置、从事件获取的本地变量、结构成员等。

如上所述，动态追踪事件是非常便利的功能，但是也受到一些约束。首先，内核中也有一部分无法添加追踪事件的区域。对于 NMI、异常、`kprobes` 的相关代码，无法添加追踪事件（这是 `kprobes` 的约束）。其次，动态追踪事件并不改写磁盘上的内核映像文件，因此只要重启系统就会消失。每次启动时都要添加相同事件的情况下，就需要将动态追踪事件的定义另行保存。虽然如此，但有了 `perf tools` 的帮助，就可以很轻松地添加动态追踪事件。寻找内核的 bug 时也是非常有用的工具。

## 经由 `ftrace` 将动态追踪事件添加到内核中

下面介绍经由 `ftrace` 的特殊文件添加动态追踪事件的方法。动态追踪事件在 `ftrace` 中的

接口为 `/sys/kernel/debug/tracing/kprobe_events`。经由 ftrace 定义动态追踪事件，就是按照下列格式向这个文件中写入信息。

`[p|r] [: [组名 /] 事件名] 符号 [+ 偏移量] | 内存地址 [参数]`

开头的文字如果为 p，就表示定义在指定动态追踪事件的地址上。开头的文字如果为 r，则表示定义在指定动态追踪事件的函数（符号）返回调用起点的地方。未指定事件名时，从符号名称和偏移量生成任意的事件名。另外，参数的格式如表 8-13 所示。

表 8-13 参数的格式

参    数	说    明
<code>%REG</code>	获取寄存器 REG 的值 (REG 的名称依赖于 CPU 架构)
<code>@ADDR</code>	获取内核地址 ADDR 中的数据
<code>@SYM [+ -offs]</code>	获取内核符号 SYM 加上或减去偏移量 offs 的地址中的数据
<code>\$stackN</code>	获取栈的第 N 个人口 (人口大小依赖于架构)
<code>\$stack</code>	获取栈开头的地址
<code>\$retval</code>	获取函数的返回值 (但是只有位于函数调用的返回部分的追踪事件可以使用)
<code>+ -offs (参数)</code>	通过“参数”获取的值 ± 偏移量 offs 对应地址中的值 (用于结构成员的访问等)
<code>NAME= 参数</code>	把参数命名为 NAME。这个名称在整理追踪事件时使用
<code>参数 :TYPE</code>	将参数记录为 TYPE 类型。可以向 TYPE 指定基本型 <code>u8</code> 、 <code>u16</code> 、 <code>u32</code> 、 <code>u64</code> 、 <code>s8</code> 、 <code>s16</code> 、 <code>s32</code> 、 <code>s64</code> 以及 <code>string</code> 和位域 (bit field) 类型

例如，在 `vfs_read` 的开头定义记录 `ax` 寄存器 (x86-64 作为 `RAX`，x86-32 作为 `EAX` 处理) 和栈开头要素的动态追踪事件 `event1`，使用下列方法进行确认。

```
[~] # cd /sys/kernel/debug/tracing
[tracing] # echo p:event1 vfs_read %ax \$stack0 > kprobe_events
[tracing] # cat kprobe_events
p:kprobes/event1 vfs_read arg1=%ax arg2=$stack0
```

使用 `$stack` 或 `$retval` 时，请不要忘记转义特殊字符 (\$)。另外，如果不指定参数的类型，基本上就全部作为 `unsigned long` 类型进行处理，因此 x86-64 作为 `u64`，x86-32 作为 `u32` 处理。如果不向参数指定名称，就会自动分配 `arg1`、`arg2` 的名称。

下面讨论新添加的动态追踪事件是不是真的与其他事件相同。

```
[tracing] # find events/kprobes/
events/kprobes/
events/kprobes/event1
events/kprobes/event1/format
events/kprobes/event1/filter
events/kprobes/event1/id
events/kprobes/event1/enable
```

```

events/kprobes/enable
events/kprobes/filter
[tracing] # cat events/kprobes/event1/format
name: event1
ID: 834
format:
    field:unsigned short common_type;          offset:0;          size:2;
signed:0;
    field:unsigned char common_flags;          offset:2;          size:1;
signed:0;
    field:unsigned char common_preempt_count; offset:3;
size:1; signed:0;
    field:int common_pid;   offset:4;          size:4; signed:1;
    field:int common_lock_depth;   offset:8;          size:4; signed:1;

    field:unsigned long __probe_ip; offset:16;          size:8; signed:0;
    field:u64 arg1; offset:24;          size:8; signed:0;
    field:u64 arg2; offset:32;          size:8; signed:0;

print fmt: "({lx}) arg1=%llx arg2=%llx", REC->__probe_ip, REC->arg1, REC->arg2

```

可以看出，确实连格式的定义都是相同的。这个事件可以从 ftrace 使用，如下所示。

```

[tracing] # echo kprobes:event1 > set_event
[tracing] # head trace
# tracer: nop
#
#      TASK-PID      CPU#      TIMESTAMP      FUNCTION
#      | |          |          |          |
libvirtd-1242  [003]  22550.135210: event1: (vfs_read+0x0/0x190) arg1=0
arg2=fffffffff81148f81
    libvirtd-1242  [003]  22550.135286: event1: (vfs_read+0x0/0x190)
arg1=400 arg2=fffffffff81148f81
    libvirtd-1242  [003]  22550.135316: event1: (vfs_read+0x0/0x190)
arg1=800 arg2=fffffffff81148f81
    libvirtd-1242  [003]  22550.135343: event1: (vfs_read+0x0/0x190)
arg1=c00 arg2=fffffffff81148f81
    libvirtd-1242  [003]  22550.135346: event1: (vfs_read+0x0/0x190)
arg1=d28 arg2=fffffffff81148f81
    libvirtd-1242  [003]  22550.135420: event1: (vfs_read+0x0/0x190) arg1=0
arg2=fffffffff81148f81

```

在上述示例中，向符号开头的地址添加了事件，也可以指定符号 + 偏移量。x86 采用的是命令长度各不相同的 CISC 方式，因此必须进行反汇编才能知道可以向哪个地址添加事件。但是，即使指定了错误的地址，由于内核内部会进行命令范围的检查，因此会作为定义时的错误被排除。

要删除这样定义的动态追踪事件，可以使用两种方法。

如果想删除所有定义，可以像下面这样向 `kprobe_events` 写入空白。

```
[tracing] # echo > /sys/kernel/debug/tracing/kprobe_events
```

想删除单个事件，可以向 `kprobe_events` 追加 “-:[组名/]事件名” 命令。例如，只

删除 event1 的方法如下。

```
[tracing] # echo -:kprobes/event1 >> kprobe_events
```

## 使用 perf probe 将动态追踪事件添加到内核中

perf probe 是为了定义动态追踪事件而开发的 perf tools 的功能之一。

如前所述，如果想要经由 ftrace 的接口直接定义事件，必须直接指定地址或寄存器来确定事件的内容。要添加源代码和执行二进制文件的内容，需要将二进制码反汇编、搜索作为本地变量的寄存器等，就要求能够熟练使用二进制。这对于刚刚开始进行内核编译的人来说使用比较困难，因此有人可能觉得使用 `printf` 更为方便。perf probe 就是在这种情况下出现的。

perf probe 可以分析内核的调试信息，在源代码层次定义动态追踪事件。另外，还可以查看源代码的哪一行可以追踪，哪个变量可以访问等。perf probe 就像是小型的源代码调试程序。

## 启用调试信息和动态追踪事件构建内核

使用 perf probe 在源代码层次定义动态追踪事件时，必须要有内核的调试信息。

使用发布版附带的内核时，有些发布版（如 RedHat 和 SuSE 等）提供了所有的调试信息，而 Debian 和 Ubuntu 的内核调试信息数据包有时就不包括关于驱动程序的调试信息。每个发布版需要安装的数据包是不同的，因此需要根据所使用的发布版来选择。

本节仅介绍作为一个内核技术人员（Kernel Hacker）如何从源代码构建内核，生成调试信息的方法。虽说如此，但也并非与已经提到很多次的内核构建方法有太大的不同。调试信息可以在编译时向编译器传递 `-g` 选项来生成，但内核中已经存在用来启用这个选项的配置项目（`CONFIG_DEBUGINFO`）。`CONFIG_DEBUGINFO` 在内核配置中的位置如下。

```
Kernel hacking ->
    [*] Compile the kernel with debug info
```

除此以外，还需要启用动态追踪事件的支持。由于动态追踪事件使用 kprobes 和 ftrace，因此需要启用 `CONFIG_KPROBES`、`CONFIG_FTRACE`、`CONFIG_KPROBE_EVENT`。这三者分别位于下列位置。

```
General setup ->
    [*] Kprobes
Kernel hacking ->
    [*] Tracers ->
        [*] Enable kprobes-based dynamic events
```

启用这些选项构建内核，安装驱动程序模块和主体（body）。但是，安装了内核后，请注意不要删除创建的目录下的 `vmlinux` 文件。`vmlinux` 文件包含调试信息，而安装到 `/boot` 目录等的 `vmlinuz` 的调试信息已删除。perf probe 会从创建目录搜索

vmlinux，以便读出调试信息，因此，如果这个文件被删除就会运行失败。另外，即使构建了新的内核，也不能改写或删除内核源代码。调试信息只包含创建时的源代码，因此，一旦源代码更改就会产生不一致。

包括 perf probe 在内的 perf tools 虽然在内核的源代码中也是存在的，但创建时需要与内核分开进行。perf tools 的创建方法请参考 Hack #64。

下面尝试使用新的内核运行 perf probe。

## perf probe 的使用方法

perf probe 具有多个运行选项。这里将按照实际添加追踪事件、使用并删除的步骤来介绍该命令的使用方法。

首先，需要确认用来定义追踪事件的源代码。perf probe 具有显示源代码以及显示源代码的哪一行可以插入事件的功能。

```
[~] # perf probe --line vfs_symlink
<vfs_symlink:0>
  0  int vfs_symlink(struct inode *dir, struct dentry *dentry,
const char *oldname)
  1  {
  2      int error = may_create(dir, dentry);
  4
  4      if (error)
  5          return error;
  7
  7      if (!dir->i_op->symlink)
  8          return -EPERM;
  10
  10     error = security_inode_symlink(dir, dentry, oldname);
  11
  11     if (error)
  12         return error;
  14
  14     error = dir->i_op->symlink(dir, dentry, oldname);
  15
  15     if (!error)
  16         fsnotify_create(dir, dentry);
  17
  18 }
```

```
SYSCALL_DEFINE3(symlinkat, const char __user *, oldname,
                int, newdfd, const char __user *, newname)
```

根据不同的内核构建环境和 perf tools 的版本（即内核版本），有时会出现找不到源代码路径的错误。这时应当先移动到内核源代码解压缩到的目录再尝试执行。

通过 perf probe 的 --line 选项可以知道指定为参数的函数和文件的源代码，以及是否能向各自的行添加事件。从输出结果可以看出，有些行的最前面有编号，有些行没有编号。前面有编号的行可以添加追踪事件。在 C 语言中，空行等无意义的行在编译时会消失，因此不能向这些地方添加追踪事件。调试信息记录了与进行相关处理的行相对应

的地址，因此 perf probe 可以对其进行分析并找出可以指定的行。

另外，从 perf probe 的 --var 选项可以看出指定为参数的行可以访问哪个变量。使用编译器的优化选项时，有的变量一旦进行优化就会消失，需要事先调查出可以访问的变量。下面实际列举的是从 vfs\_symlink 的第 4 行可以访问的变量。

```
[~] # perf probe --vars vfs_symlink:4
Available variables at vfs_symlink:4
  @<vfs_symlink+36>
    char*    oldname
    int      error
    struct dentry*  dentry
    struct inode*  dir
```

根据这个结果，可以看出能够访问 oldname、error、dentry、dir 这 4 个局部变量。如果加上 --var 选项的附加选项 --externs，则除了上述局部变量以外，还会列举出全局变量。

实际定义追踪事件的是 perf probe --add。perf probe --add 指定追踪事件定义位置的格式如下。

- 根据函数名定义时：

```
[ 事件名 =] 函数[@ 文件 ][: 从函数开头的行数 |+ 偏移量 |%return|; 模式 ]
```

- 根据文件名和行数或模式定义时：

```
[ 事件名 =] 文件 [: 从文件开头的行数 |; 模式 ]
```

文件名是基于绝对路径的，但不需要写出实际文件的所有绝对路径。perf probe 只会检查文件名的末尾是否一致，因此，例如，当绝对路径为 /usr/src/linux-2.6/kernel/timer.c 时，只需要写出 linux-2.6/kernel/timer.c 就足够了。格式中的“模式”是在选择符合条件的多个行时使用的一种 glob 描述匹配。例如，schedule;cpu=[!=]\* 显示的是 schedule() 函数中向变量 cpu 赋值的行。

在下例中，vfs\_symlink 的第 4 行定义了记录 dentry 结构的 d\_name.name、oldname、error 各自值的事件。另外，为了提高事件的可读性，将 dentry -> d\_name.name 和 oldname 设置为以 string 类型记录。

```
[~] # perf probe --add 'vfs_symlink:4 dentry->d_name.name:string oldname:string error'
Add new event:
  probe:vfs_symlink      (on vfs_symlink:4 with name=dentry->d_name.name:string
                           oldname:string error)
```

You can now use it on all perf tools, such as:

```
  perf record -e probe:vfs_symlink -aR sleep 1
```

当前已定义的动态追踪事件列表可以通过 perf probe --list 看到。

```
# perf probe --list
probe:vfs_symlink      (on vfs_symlink:4@fs/namei.c with name oldname_string error)
```

可以看出，使用 `perf probe --list` 可以看到该事件定义在哪个文件的哪一行（这个例中为 `fs/namei.c` 的 `vfs_symlink()` 函数的第 4 行）。

下面，使用 Hack #69 介绍的方法来尝试获取追踪。

```
[~] # cd /sys/kernel/debug/tracing
[tracing] # echo probe:vfs_symlink > set_event
[tracing] # ln -s /tmp/hoge /tmp/huga
[tracing] # head trace
# tracer: nop
#
#           TASK-PID      CPU#      TIMESTAMP  FUNCTION
#           | |          |          |          |
#           ln-2439  [001]  9061.438993: vfs_symlink: (vfs_symlink+0x24/0x78)
name="huga" oldname_string="/tmp/hoge" error=0
```

可以看出，记录了已执行的 `ln` 的变量等。

## 小结

本节介绍了使用 ftrace 及 perf probe 定义动态追踪事件的方法。通过使用动态追踪事件，可以在不停止正在运行的内核的情况下，轻松确认内核内部的详细运行。例如，可以对由于变量的值而导致处理速度不同，成为瓶颈的函数进行概要分析。

## 参考文献

- Documentation/trace/kprobestrace.txt
- tools/perf/Documentation/perf-probe.txt

——Masami Hiramatsu

# HACK #72 使用 SystemTap 进行内核追踪

本节以进行定时（timing）的样本为例介绍 SystemTap 的使用方法。

## 概述

SystemTap 是使用 kprobes 生成的工具。使用类似于 AWK 和 C 语言的特有脚本语言生成 probe 处理程序。将使用脚本写出的 probe 处理程序由专用解析器（parser）转换为 C 语言，自动生成内核模块。SystemTap 可以对生成的代码进行安全检测，为称为 tapset 的库脚本提供便利的函数，可以提供比手写内核模块更为方便的环境。

为了加强自动生成的模块的安全性，除了生成代码时的安全性检测以外，生成的代码还包括执行过程中的循环次数和函数嵌套次数检测、处理的系统开销限制等的严格检测代码。极力缩小这样编写出的代码失控的可能性。

## 准备

使用 SystemTap，必须要有带调试信息编译的内核。另外，`stap` 命令会自动创建内核模块，因此还需要安装内核头文件等。本节使用 Fedora 14。使用的内核版本为 2.6.35.12-90.fc14，SystemTap 的版本为 1.4-2.fc14。Fedora 14 使用下列命令安装必要的工具包。

```
# yum install systemtap yum-utils
# debuginfo-install kernel
```

另外，要检查函数内部的源代码时，导入内核的源代码比较方便，但是需要进行一些准备工作。准备工作的步骤包括：安装 `rpmbuild` 数据包，安装内核的 SRPM (source rpm)，指定 `kernel.spec` 文件，使用 `rpmbuild` 生成可编译的内核源码树，如下所示。

```
# yum install rpmbuild
# yumdownloader --source kernel-2.6.35.12-90.fc14
# rpm -i kernel-2.6.35.12-90.fc14.src.rpm
# rpmbuild -bp ~/rpmbuild/SPECS/kernel.spec
```

这样就会在 `~/rpmbuild/BUILD/kernel-2.6.35.fc14/linux-2.6.35.i686/` 下解压缩源代码。(在 RHEL、CentOS 和之前的 Fedora 中，情况有所不同，安装 SRPM 后将解压缩到 `/usr/src/redhat` 下，而不是 `~/rpmbuild`)。

## 样本脚本

这里使用的是如下所示的脚本。这是在 SystemTap 中附属的 `sleeptime.stp` 样本脚本的基础上进行了一些修改而生成的。是用来测量在 `nanosleeptest` 程序中 `nanosleep()` 系统调用实际休眠了多少时间的脚本。

```
#!/usr/bin/stap -v

/*
 * Format is:
 * TIMESTAMP PID (EXECNAME) MESSAGE
 */

global start
global entry_nanosleep
global entry_nanosleep_restart

function timestamp:long() {
    return gettimeofday_us() - start
}

function proc:string() {
    return sprintf("%d (%s)", pid(), execname())
}

probe begin {
    start = gettimeofday_us()
}
```

```

probe syscall.nanosleep {
    if (execname() != "nanosleeptest") next;
    t = gettimeofday_us(); p = pid()
    entry_nanosleep[p] = t
}

probe syscall.nanosleep.return {
    if (execname() != "nanosleeptest") next;
    t = gettimeofday_us(); p = pid()
    elapsed_time = t - entry_nanosleep[p]
    printf("%d %s nanosleep: %d\n", timestamp(), proc(), elapsed_time)
    delete entry_nanosleep[p]
}

probe kernel.statement("hrtimer_nanosleep@kernel/hrtimer.c:1599") {
    if (execname() != "nanosleeptest") next;
    printf("%d %s nanosleep is interrupted.\n", timestamp(), proc());
    p = pid();
    entry_nanosleep_restart[p] = entry_nanosleep[p];
}

probe kernel.function("hrtimer_nanosleep_restart").return {
    if (execname() != "nanosleeptest") next;
    t = gettimeofday_us(); p = pid()
    elapsed_time = t - entry_nanosleep_restart[p]
    printf("%d %s nanosleep_restart: %d\n", timestamp(), proc(), elapsed_time)
    delete entry_nanosleep_restart[p];
}

```

追踪对象程序使用的是如下所示的 `nanosleeptest.c`。

```

#include <time.h>

int main(int argc, char *argv[])
{
    struct timespec ts = {0, 0};

    if (argc != 2)
        return 0;
    ts.tv_sec = strtol(argv[1], NULL, 0);
    return nanosleep(&ts, NULL);
}

```

## 测量时间

使用 SystemTap 进行追踪可以很方便地测量出某个进程所需的时间。但是需要注意的是，不适合用于几微秒以下的时间测量。这是因为，虽然由于脚本的制作和机器的性能而有所不同，但 SystemTap 自身系统开销的影响是不能忽视的。

在内核内部的处理中，有很多是将非同步的事件作为触发器来推进处理。本节提到的 `nanosleep()` 是将计时器的中断事件等作为触发器从休眠状态恢复。这个计时器的中断事件是非同步的事件，这个时刻会受到系统负载情况的影响。另外，将执行了

`nanosleep()` 的进程从休眠状态唤醒，重新进行调度也需要花费时间。这样，进程一般就会在比所要求的休眠时间稍晚的时刻从 `nanosleep()` 系统调用恢复。本次测量的就是实际休眠的时间。

为了测量实际的休眠时间，将执行了 `nanosleep()` 后的时刻和从 `nanosleep()` 恢复的时刻的差作为时间差显示。测量的时刻是使用 SystemTap 提供的 `gettimeofday_us()`，以微秒为单位获取当前时刻。

## 定义探测点

插入 `probe` 的位置称为探测点（probe point）。探测点的定义方法有很多种，这里介绍其中一些经常使用的定义方法。

```
probe begin
probe end
```

定义脚本启动时、结束时运行的处理程序。它可用于脚本内的全局变量的初始化，以及显示结束时收集的日志的情况等。

```
probe kernel.function(" 函数名 ")
probe kernel.function(" 函数名 ").return
```

分别在定义“函数名”所指定的内核函数在调用时、返回时在执行的 `probe` 时使用。“函数名”对应的部分也可以使用通配符（\*）。

```
probe kernel.function("*init*")
probe kernel.function("*init*@kernel/sched.c")
```

在第一个示例中，是向所有内核的初始化函数插入 `probe`。在第二个示例中，是向所有定义在 `kernel/sched.c` 中的初始化函数插入 `probe`。像这样，也可以通过在函数名后指定文件名来限制对象范围。当不同文件中存在相同名称的函数时，可以使用这个描述限定一个对象函数。另外，在文件名部分也可以使用通配符。

在内核模块中定义 `probe` 时描述如下。

```
probe module(" 模块名 ").function(" 函数名 ")
probe module(" 模块名 ").funciton(" 函数名 ").return
```

只需要将之前写作 `kernel` 的地方改写成 `module(" 模块名 ")`，其他代码完全相同。

```
probe syscall. 系统调用名
probe syscall. 系统调用名 .return
```

这是 `probe` 对象作为系统调用时的描述方法。这些是使用别名（alias）功能，对前面所述的 `kernel.function()` 进行绑定（wrapping）。在系统调用的情况下，建议使用这个描述方法。

```
probe kernel.statement(" 函数名 @ 文件名 : 行编号 ")
probe kernel.statement( 地址 )
```

这些用于在函数行中插入 `probe` 的情况。例如，可以在仅想调试满足函数内 `if` 语句的条

件的情况时使用。当然，如果在对内核源代码进行了修改、行编号改变的情况下使用，就会出现偏差，因此要注意每次都要确认源代码。这是因为由 `stap` 命令从内核的调试信息推测出与行编号对应的地址，决定 `probe` 位置。至于地址的指定，由于在每次改变内核的条件进行创建时都会变化，因此需要经常对内核二进制映像文件进行反汇编，找出正确的位置。

同样，以内核模块为对象的描述如下。

```
probe module("模块名").statement("函数名 @ 文件名 : 行编号")
probe module("模块名").statement(地址)
```

在本节的示例中使用这个描述，向 `nanosleep()` 在信号中断时运行的代码中插入 `probe`。

```
probe kernel.statement("hrtimer_nanosleep@kernel/hrtimer.c:1599")
```

这样就可以引用已插入 `probe` 的部分的内核源代码。使用这样的方法，还可以生成用来判断函数内 `if` 语句的判断结果是真还是假的脚本。

```
[kernel/hrtimer.c]
1570 long hrtimer_nanosleep(struct timespec *rqtp, struct timespec __user
 *rmtp,
1571                                     const enum hrtimer_mode mode, const clockid_t
clockid)
1572 {
...
     /* 如果插入信号中，则该if语句为假 */
1584     if (do_nanosleep(&t, mode))
1585         goto out;

...
1599     restart = &current_thread_info()->restart_block;
1600     restart->fn = hrtimer_nanosleep_restart;
1601     restart->nanosleep.index = t.timer.base->index;
1602     restart->nanosleep.rmt = rmtp;
1603     restart->nanosleep.expires = hrtimer_get_expires_tv64(&t.timer);
1604
1605     ret = -ERESTART_RESTARTBLOCK;
1606 out:
1607     destroy_hrtimer_on_stack(&t.timer);
...
```

如果只是这样在检测点上记录处理方式，就会追踪系统上运行的所有 `nanosleep()`，甚至不需要追踪的命令的运行情况也会被追踪。本次只想以 `nanosleep` 程序的运行为对象，因此需要进行下列过滤。

```
if (execname() != "nanosleep") next;
```

`execname()` 函数会返回当前进程的执行命令名称，因此，如果不是 `nanosleep`，就会执行 `next` 跳过剩下的处理。

## 尝试执行

执行脚本的命令为 `stap`。默认只显示脚本的执行结果，因此不知道脚本的生成和编译是何时进行的。要得知脚本的执行是何时开始的，需要尽量添加 `verbose` 选项 (`-v`) 并执行命令。

```
# stap -v sleeptime.stp
Pass 1: parsed user script and 76 library script(s) using 22624virt/13468res/2432shr
kb, in 350usr/40sys/409real ms.
Pass 2: analyzed script: 5 probe(s), 7 function(s), 23 embed(s), 3 global(s)
using 209844virt/58080res/3404shr kb, in 1650usr/1640sys/4818real ms.
Pass 3: translated to C into "/tmp/stapX8Wjed/stap_cf3acae223ba1d1359216122bbc07
f5e_15503.c" using 209564virt/61080res/6520shr kb, in 1340usr/40sys/1418real ms.
Pass 4: compiled C into "stap_cf3acae223ba1d1359216122bbc07f5e_15503.ko" in
4380usr/740sys/5823real ms.
Pass 5: starting run.
```

如果显示上述信息，则表示 `probe` 已启用。从其他终端使用测试账号执行 `nanosleep()`，休眠 10 秒钟。

```
$ nanosleep 10
```

这样就会在执行了 `stap` 命令的终端上显示下列信息。

```
51245488 1296 (nanosleeptest) nanosleep: 10000180
```

信息显示格式如下。显示的时间差都是以微秒为单位。

从启用 `probe` 开始的时间差 PID (命令名称) `nanosleep`: 实际经过时间

可能是因为本次是在虚拟环境下运行的，所以可以看出延迟达到 180 微秒。

接下来确认在刻意使用信号中断时是怎样的情况。与之前一样使用 `nanosleeptest` 程序休眠 10 秒钟。但是，中途发送 `SIGSTOP`，中断 `nanosleep()`，这样等待大约 10 多秒再发送 `SIGCONT` 信号。

```
79008508 3016 (nanosleeptest) nanosleep is interrupted.
79008521 3016 (nanosleeptest) nanosleep: 5943297
85588697 3016 (nanosleeptest) nanosleep_restart: 12523472
```

输出了表示 `nanosleep()` 已中断的信息。从第 2 个信息可以看出，执行 `nanosleep()` 后约 5.9 秒后接收了 `SIGSTOP` 信号。从第 3 个信息可以看出，重新开始执行的 `nanosleeptest` 从 `nanosleep()` 恢复花费了约 12.5 秒。

确认完成后，可以按 `Ctrl+C` 快捷键结束 `SystemTap`。`SystemTap` 在结束时会自动将追踪用的内核模块从内核中移除。

## 小结

本节以测量 `nanosleep()` 系统调用实际经过时间为例子，介绍了 `SystemTap` 的使用方法。由于 `SystemTap` 使用的是 `kprobes`，因此 `kprobes` 可以完成的工作基本上在 `SystemTap` 中也可以实现。

## 参考

除了 man 页面以外，还可以参考 SystemTap 附带的样本脚本和 tapset。项目的网页上也有 TUTORIAL 等很多文档类。

- SystemTap 附带的样本脚本  
`/usr/share/doc/SystemTap-version/examples/`
- SystemTap 附带的 tapset  
`/usr/share/SystemTap/tapset/`
- SystemTap 项目网页  
<http://sourceware.org/SystemTap/documentation.html>

——Masami Hiramatsu

## HACK #73 使用 SystemTap 编写对话型程序

本节使用 SystemTap 在内核中执行交互式（interactive）程序。

虽然在执行时有一些制约，但 SystemTap 还是可以作为一种脚本语言来处理。既可以向控制台输出，也可以调用外部程序。另外，还可以追踪 Linux 内核上获取的信息，因此也可以得到键盘输入或鼠标输入的情况。也可以编写内核计时器的处理程序，因此可以每隔一定时间更新状态。但是反过来说，复杂且花费时间的处理就会长时间停止内核自身的处理，因此不适合使用 SystemTap。

也就是说，SystemTap 是适用于编写游戏等对话型程序的语言。特别是，计时器周边的处理不会发生延迟，因此如果无视禁止硬件中断的影响，就可以进行最快的周期处理。

本节将介绍怎样使用 SystemTap 接受用户的输入，怎样进行简单的界面控制。

### 使用 SystemTap 进行输出界面控制

SystemTap 的输出为控制台。要对控制台进行控制，使用 ANSI 转义序列（escape sequence）是最简单的。SystemTap 为此准备了库函数（见表 8-14）。

表 8-14 SystemTap 的库函数

函 数	说 明
<code>ansi_clear_screen()</code>	清除控制台界面
<code>ansi_set_color(fg)</code>	指定文字颜色
<code>ansi_set_color2(fg, bg)</code>	指定文字颜色和背景颜色
<code>ansi_set_color3(fg, bg, attr)</code>	指定文字颜色、背景颜色和颜色属性
<code>ansi_reset_color()</code>	将文字颜色恢复到默认状态

函 数	说 明
<code>ansi_cursor_move(x, y)</code>	将光标（下一个输出文字的位置）移动到任意位置。x 为横向，y 为纵向。（1,1）为控制台画面左上角的坐标
<code>ansi_cursor_hide()</code>	隐藏光标
<code>ansi_cursor_show()</code>	将光标设置为可见
<code>ansi_cursor_save()</code>	保存光标的位置
<code>ansi_cursor_restore()</code>	将光标的位置返回最后保存的位置

熟练使用 `ansi_clear_screen` 和 `ansi_cursor_move`，就可以每隔一定时间更改控制台界面，或者在任意位置写入任意文字。例如，执行下列脚本，就可以在看到“@”在界面上斜着运动。

```
global x=40, y=1;

probe timer.ms(100) {
    ansi_clear_screen()
    ansi_cursor_move(x--, y++/2)
    print("@")
    if (x == 0)
        exit()
}
```

像这样重复地使计时器失效，每隔一定时间刷新界面，并绘制新状态，就可以在界面上自由显示文字。

## 使用 SystemTap 接受来自键盘、鼠标的输入

Linux 内核先使用 `input_event` 接受外部的输入，再传送到适当的子系统（subsystem）。这个事件包括键盘或鼠标的事件，因此通过检测 `input_event` 函数，就可以获取用户的键盘输入或鼠标输入。

```
void input_event(struct input_dev *dev,
                 unsigned int type, unsigned int code, int value)
```

如上所示，`input_event` 函数可以接受对于设备的 `type`、`code`、`value` 输入。`type` 表示设备种类，例如，1 为键盘，2 为鼠标（或相对坐标定点设备），3 为触摸屏（或绝对坐标定点设备）。需要注意的是，便携式电脑的触摸板中 1、2、3 都存在。

另外，在鼠标的事件中按钮（button）操作的 `type` 也是 1。这些组合的详细情况请参考 Linux 内核的头文件 `include/linux/input.h`。

### 键盘输入

`value` 为 0 表示键盘的键未按下，`value` 非 0 则表示键按下。`code` 中将出现接下的键

的代码，因此可以通过这个组合来判断键盘的输入。当 code 小于 256 时，就表示键盘的事件。通过使用下列脚本，就可以确认键盘的输入代码。

```
probe kernel.function("input_event") {
    if ($type != 1 || $code >= 256)
        next;
    printf("Keyboard: %d %s\n", $code, $value ? "Down" : "Up");
}
```

表 8-15 为笔者的环境中经常使用的键的代码对应表。

表 8-15 使用频率较高的键的代码对应表

键	代码	键	代码
↑	103	Enter	28
↓	108	ESC	1
←	105	z	44
→	106	x	45
Space	57	c	46

那么使用键盘在界面上移动 @ 的 kbd\_cursor.stp 就如下所示。

```
global x = 20, y = 20

probe kernel.function("input_event") {
    if ($type != 1 || $code >= 256 || $value != 1)
        next
    if ($code == 103)
        y--
    else if ($code == 108)
        y++
    else if ($code == 105)
        x--
    else if ($code == 106)
        x++
}

probe timer.ms(100) {
    if (x <= 0)
        x = 1
    else if (x >= 41)
        x = 40
    if (y <= 0)
        y = 1
    else if (y >= 41)
        y = 40
    ansi_clear_screen()
    ansi_cursor_move(x,y)
    print("@")
}
```

## 鼠标输入

鼠标输入比键盘输入复杂。这是因为定点设备有小红帽 (trackpoint) 和触摸板、触摸屏等多种形式。本节将介绍一般鼠标 (相对移动距离设备)、在笔者的环境下测试的触摸板 (绝对坐标设备)。对于这些定点设备，都可以使用下列方法来确认设备的输入。

```
# stap -e 'probe kernel.function("input_event"){println($$parms)}'
```

让 `$$parms` 作为检测函数的参数，由此来获得系统的信息，并作为字符串传递给用户。执行这个脚本并操作鼠标等，就可以看出传送了什么样的事件。

一般的鼠标通过下列组合传送相对移动距离 (见表 8-16)。

表 8-16 一般鼠标的相对移动距离

类 型	代 码	说 明
2	0	x 移动距离
2	1	y 移动距离
2	8	z 移动距离 (即所谓的滚轮)
1	0x110	是否按下左键 (1: 按下, 0: 松开)
1	0x111	是否按下右键 (1: 按下, 0: 松开)
1	0x112	是否按下中键 (1: 按下, 0: 松开)

在笔者的环境下，触摸板 (Synaptics) 是返回绝对坐标的设备。可以得知从这个设备发送的有下表所示的信号 (见表 8-17)。

表 8-17 触摸板的信号

类 型	代 码	说 明
3	0	x 位置坐标 + 偏移量
3	1	y 位置坐标 + 偏移量
1	0x145	是否碰下触摸板 (1: 碰下, 0: 松开)
1	0x110	是否按下左键 (1: 按下, 0: 松开)
1	0x111	是否按下右键 (1: 按下, 0: 松开)

但是，是否已按下按钮的信息容易造成连击 (连续进入信号)，因此处理时需要注意。另外，每个设备的位置信息的偏移量都是不同的，因此处理时也需要注意。

如何让绝对坐标和相对坐标进行相同的处理呢？最基本的方法是将所有的输入统一为绝对坐标或相对坐标进行处理。下面是如前面一样通过鼠标移动 “@” 的脚本。

```
global mpos_x = 0, mpos_y = 0
global x = 20, y = 20

# Mouse Button hooking
probe kernel.function("input_event") {
```

```

rx = 0; ry = 0
if ($type == 2) { # Relative position
    if ($code == 0)
        rx = $value
    else if ($code == 1)
        ry = $value
} else if ($type == 1 && $code == 0x145 && $value == 0) {
    # Take a finger -- Reset position
    mpos_x = 0; mpos_y = 0
} else if ($type == 3) { # Absolute - check slide
    if ($code == 0) {
        if (mpos_x != 0)
            rx = $value - mpos_x
        mpos_x = $value;
    } else if ($code == 1) {
        if (mpos_y != 0)
            ry = $value - mpos_y
        mpos_y = $value;
    }
}
if (rx == 0 && ry == 0)
    next
x += rx/10; y += ry/10
}

probe timer.ms(100) {
    if (x <= 0)
        x = 1
    else if (x >= 41)
        x = 40
    if (y <= 0)
        y = 1
    else if (y >= 41)
        y = 40
    ansi_clear_screen()
    ansi_cursor_move(x,y/2)
    print("@")
}

```

可以看出, `timer` 部分的处理程序与键盘的情况相同, 而 `input_event` 的处理程序则出现了很大的不同。在这个示例中, 为了与通过绝对坐标返回的触摸板相对应, 将触摸板返回的坐标改为与上一次的坐标 (`mpos_x`、`mpos_y`) 的相对距离。另外, 使用松开手指的事件来重置上次的坐标。从触摸板的使用方法来思考就可以理解, 用触摸板来移动光标需要用手指反复滑动很多次。一次滑动操作结束, 下一次滑动操作开始时, 又会从几乎与上次一致的位置开始滑动。从上次松开手指的位置来看, 这时的位置就相当于回到了滑动前的初始位置, 这样就很不方便。

## 小结

本节介绍了使用 SystemTap 编写对话型程序所需的界面控制和键盘、鼠标输入。SystemTap 可以使用脚本方便地捕捉内核事件, 是非常适合进行交互式处理的语言。

## 参考文献

- StapGames (<https://github.com/mhiramat/stapgames>)

——Masami Hiramatsu

## HACK #74 SystemTap 脚本的重复利用

本节使用 SystemTap 的别名和 Tapset 功能，重复使用脚本。

Hack #73 介绍了捕捉键盘和鼠标的输入的方法。可以直接在这个脚本的基础上不断扩大规模，但是想要重复使用这段输入代码等时，必须多次复制相同的代码，或者必须将大量的代码耗费在不必要的输入处理上。这就使得脚本看起来很复杂。

SystemTap 中准备了别名功能和 Tapset 库功能，各个开发人员就可以重点进行自己想进行的操作，并提高代码的使用性。这里首先介绍别名。

### 使用别名分离逻辑

别名是为了更方便地重复使用 SystemTap 的处理程序而设置的别名定义功能。SystemTap，如果只是重复使用部分代码，可以作为函数提取出来，但是处理程序本身就不能这样。这时就可以使用别名为处理程序的预处理部分加上易懂的名字。

别名的格式是 probe 格式的扩展。在 probe 后面写上别名，然后在 “=” 后面写上想要命名的处理程序。下面使用别名来编写用键盘和鼠标移动 “@” 的程序。

```
# Alias Part
probe input.move.keyboard = kernel.function("input_event") {
    rx = 0; ry = 0
    if ($type != 1 || $code >= 256 || $value != 1)
        next
    if ($code == 103)
        ry--
    else if ($code == 108)
        ry++
    else if ($code == 105)
        rx--
    else if ($code == 106)
        rx++
}
global mpos_x = 0, mpos_y = 0

probe input.move.mouse = kernel.function("input_event") {
    rx = 0; ry = 0
    if ($type == 2) { # Relative position
        if ($code == 0)
            rx = $value
        else if ($code == 1)
            ry = $value
    }
}
```

```

} else if ($type == 1 && $code == 0x145 && $value == 0) {
    # Take a finger -- Reset position
    mpos_x = 0; mpos_y = 0
} else if ($type == 3) { # Absolute - check slide
    if ($code == 0) {
        if (mpos_x != 0)
            rx = $value - mpos_x
        mpos_x = $value;
    } else if ($code == 1) {
        if (mpos_y != 0)
            ry = $value - mpos_y
        mpos_y = $value;
    }
}
rx /= 10; ry /= 10;
}

probe input.move = input.move.* {
    if (rx == 0 && ry == 0)
        next
}

# Logic Part
global x = 20, y = 20

probe input.move {
    x += rx; y += ry
}

probe timer.ms(100) {
    if (x <= 0)
        x = 1
    else if (x >= 41)
        x = 40
    if (y <= 0)
        y = 1
    else if (y >= 41)
        y = 40
    ansi_clear_screen()
    ansi_cursor_move(x,y)
    print("@")
}
}

```

可以向别名赋予以字母开头，包括字母、数字和“.”的名称。别名内的局部变量，在使用别名的一方也可以继续使用。另外，还可以将功能上相似的别名组合起来用在其他别名中。在上例中，是将 `input.move.keyboard` 和 `input.move.mouse` 组合起来作为 `input.move` 别名。如别名 `input.move` 的定义，指定别名时也可以使用通配符。

通过使用别名，将想做的事情（# Logic Part）简单地表现出来。

## 编写 Tapset

即使改为别名或函数，但是每次都要复制相同代码，就会影响可维护性。如果作为库文件重复使用，就可以只将逻辑部分分离出来。Tapset 的编写和使用都非常简单。将想要重复使用的脚本放到目录下，执行要用的脚本时只需要使用 -I 选项指定该目录。将前面的脚本分割为 # Alias Part 和 # Logic Part，并分别放到 input.stp 和 cursor.stp 这两个脚本里，进行下列操作。

```
# ls
input.stp cursor.stp
# mkdir tapset
# mv input.stp tapset
# stap -I ./tapset cursor.stp
```

## SystemTap 脚本的 Shebang

执行 SystemTap 脚本时，最麻烦的就是参数。例如，要指定自己编写的 Tapset 路径来执行时，必须每次使用 -I 选项指定。有没有像 shell 脚本一样作为可执行文件处理的方法呢？shell 文件的第一行总是以 #!/bin/sh 或 #!/bin/bash 开头。这个称为 Shebang，Linux 内核获得执行文件的开头，如果是以 #! 开头的，就会将这个文件作为参数传递给此后的可执行文件。最简单的 SystemTap 脚本的 Shebang 如下所示。

```
#!/usr/bin/stap
```

这里如果指定 -I 选项作为参数，则变为：

```
#!/usr/bin/stap -I tapset
```

但是这个方法主要存在移植性方面的两个问题。

- stap 命令并不一定总是在 /usr/bin 下。
- Tapset 的相对位置并不一定是相同目录。例如，想从上一个目录直接执行脚本时就会出错。

前一个问题一般使用 /usr/bin/env 命令来解决，但实际上将 env 命令作为 Shebang 使用时有一个很大的问题。

```
#!/usr/bin/env stap -I tapset
```

编写这样的 Shebang 时，-I tapset 就会变成 env 命令的选项，而不是 SystemTap 的选项。这时可以借用 shell 脚本的 Shebang，一并解决这两个问题。这里以上一节的 cursor.stp 为例。

```
#!/bin/sh
//usr/bin/env stap -I 'dirname $0' /tapset $@ $0; exit $?

global x = 20, y = 20

probe input.move {
```

```

        x += rx; y += ry
    }

probe timer.ms(100) {
    if (x <= 0)
        x = 1
    else if (x >= 41)
        x = 40
    if (y <= 0)
        y = 1
    else if (y >= 41)
        y = 40
    ansi_clear_screen()
    ansi_cursor_move(x,y)
    print("@")
}

```

SystemTap 将以 // 开头的行作为注释处理。而 shell 只是把 // 识别为根目录，因此可以在其后自由编写 shell 脚本。在这个示例中是使用 dirname 命令，使用（由 \$0 传递的）可执行文件的相对路径指定 Tapset。另外，将执行时传递给脚本的参数传递给 SystemTap。通过这样的处理，可以向 SystemTap 脚本传递默认参数，同时自由添加 SystemTap 的参数。

## 小结

本节介绍了提高 SystemTap 的脚本重复使用性的别名功能以及 SystemTap 的 Shebang。通过使用这些功能，就可以提高 SystemTap 的脚本生成效率，使脚本更易执行。

——Masami Hiramatsu

# HACK #75 运用 SystemTap

本节介绍在实际系统上运用 SystemTap 的方法。

SystemTap 是优秀的追踪工具。除了作为工具以外，作为服务也具有十分优秀功能。具有作为服务启动的后台执行模式，也具有作为系统服务管理的接口。另外，在集群（cluster）环境想要重新分配相同脚本时，也可以分配已编译的脚本。

## 在后台执行 SystemTap

下面将介绍 SystemTap 的后台执行模式。SystemTap 的后台执行，有内存飞行记录器模式（on memory flight mode）和文件飞行记录器模式（on file flight mode）这两个模式。前者是将输出结果保存在内核内存上的模式，使用方法与不断提取最新记录的飞行记录器或行车记录器（drive flight）相同。后者是在后台将输出结果写出到磁盘上的模式，适用于长期提取记录的系统监测方法。

为了介绍这个功能，实际编写 SystemTap 的脚本 `syscalltop.stp`，它按照系统调用的种类

分别显示每 5 秒钟执行系统调用的次数。

```
global syscalls, stime;
probe syscall.* {
    stime[name, tid()] = gettimeofday_us();
}
probe syscall.*.return {
    if (!stime[name, tid()])
        next;
    syscalls[name] <<< (gettimeofday_us() - stime[name, tid()]);
    delete stime[name, tid()];
}
probe timer.s(5) {
    println("<name>\t<count>\t<avgtime(us)> --- ");
    foreach(name- in syscalls) {
        printf("%s\t%d\t%d\n", name,
               @count(syscalls[name]), @avg(syscalls[name]));
    }
    delete syscalls
}
```

下面是使用这个脚本执行飞行记录器的例子。

首先尝试将输出结果保存在内存上的内存飞行记录器模式。使用这个模式时需要向 `stap` 命令传递 `-F` 选项来启动。启动时使用 `-m` 决定脚本模块名称，后面的控制就会比较容易。另外还可以使用 `-s`（小写）选项指定保存日志的缓冲区容量。缓冲区容量在启动后就无法更改，因此必须在这时指定最适合的容量。

```
# stap syscalltop.stp -m syscalltop -F
Disconnecting from systemtap module.
To reconnect, type "staprund -A syscalltop"
# lsmod | head
Module           Size  Used by
syscalltop       800448  0
ip6table_filter   1759  0
ip6_tables        19857  1 ip6table_filter
ipt_MASQUERADE    1927  3
iptable_nat        4430  1
nf_nat            18833  2 ipt_MASQUERADE,iptable_nat
nf_conntrack_ipv4  13057  4 iptable_nat,nf_nat
nf_defrag_ipv4     1609  1 nf_conntrack_ipv4
xt_state           1394  1
```

在内存飞行记录器模式下运行时，就会返回如上所示的提示（prompt）。但是从 `lsmod` 的结果可以看出，从 `syscalltop` 脚本生成的脚本模块在内核中不断运行。要确认 `syscalltop.stp` 的输出结果时，如界面输出所示执行 `staprund -A syscalltop`。

```
# staprund -A syscalltop
<name>  <count> <avgtime(us)> ---
writev  5        7
write   11       16
wait4   2        6990840
tgkill   5        5
stat    1        722
```

```
setitimer      4      3
select    12  3295040
rt_sigtimedwait 2  6988895
...
```

即使不指定 -F 选项，在运行中也可以按下 **Ctrl+\** 或向 **stapio** 进程发送 **SIGQUIT** 信号，同样切换到内存飞行记录器模式。

接下来尝试将输出结果写出到文件的文件飞行记录器模式。这个模式与刚才的模式同样可以使用 -F 选项，再指定 -o 选项，指定写出到的文件。这样的话文件的大小就会逐渐变大，对磁盘容量造成压力，因此可以使用 -s 选项来避免。-s 选项可以指定文件的最大大小和最大文件数量。写出到的文件容量一旦超过 -s 选项指定的量，SystemTap 就会打开另一个文件，并开始在这个文件中写出。这样不断增加的文件数量一旦超过 -s 选项指定的最大数量，就会从最旧的文件开始删除。

进行这样的后台执行时，为了减少运行时的系统开销，可以启用 -b 选项并启用 bulk 模式。启用 bulk 模式后，SystemTap 就会在每个 CPU 上分配缓冲区，并记录追踪信息。向文件写出时，也会在每个 CPU 上创建线程，向其他文件写出。这样就可以防止数据记录时的竞争，因此系统开销就会减小。

-o 选项可以向输出文件的名称指定 **strftime()** 函数形式的格式。在下例中，将前面所述的 **syscalltop** 脚本在文件飞行记录器模式下启动，并将追踪日志输出到名称为 **tracelog- 年 - 月 - 日 - 时 : 分 : 秒 .log** 的文件。另外，输出文件的最大大小设置为 256MB，只留下最新的两个日志。

```
# stap syscalltop.stp -m syscalltop -F -o tracelog-%F-%T.log -S 256,2
2565
# ps aux | grep 2565
root      2565  0.2  0.0  47244    596 ?          Ssl  16:33   0:00 /usr/
lib/systemtap/stapio -o syscalltop-%F-%T.log -D -S 256,2 /tmp/stapSyNXqX/
syscalltop.ko
...
# ls tracelog*
tracelog-2011-04-17-16:33:14.log.0
```

在文件飞行记录器模式下启动 **stap** 命令时，会在命令结束前显示一直在后台运行的进程的 PID。通过向这个 PID 发送信号，进行后台进程的控制。例如，如果发送 **SIGUSR2** 信号，就会切换正在写出的文件（参考下例），发送 **SIGQUIT** 信号就可以从文件飞行记录器模式转换到内存飞行记录器模式。想要结束运行时可以发送 **SIGTERM** 信号。

```
# ls tracelog*
tracelog-2011-04-17-16:33:14.log.0
# kill -USR2 2565
# ls syscalltop*
tracelog-2011-04-17-16:33:14.log.0
tracelog-2011-04-17-16:35:03.log.1
```

# 将 SystemTap 作为服务启动

将 SystemTap 添加到 initscript 中，就可以作为服务启动。这样除了可以集中操作用户生成的脚本以外，还可以根据内核版本的升级重新编写脚本。这个结构可以同时控制多个脚本。例如，系统设计时准备了很多不同作用的脚本，管理人员可以根据各个服务器的运用情况仅启用有效的脚本，或者复制到各服务器中。

## SystemTap 服务的设置

要使用文件 SystemTap 服务，首先需要安装 `systemtap-initscript` 工具包。Debian 系列的发布版中也可能没有工具包，因此需要注意。

首先将 SystemTap 脚本复制到指定的目录 (`/etc/systemtap/script.d/`)。这样就可以在内存飞行记录器模式下运行。要扩大脚本执行时的缓冲区大小，或者要在文件飞行记录器模式下运行时，必须对配置文件进行一些修改。如表 8-18 和表 8-19 所示，配置文件中有全局配置文件、脚本或脚本组用配置文件。前者生成的名称为 `/etc/systemtap/config`，后者针对每个脚本生成 `/etc/systemtap/conf.d/*.config`。

表 8-18 全局设置文件的参数（圆括号内为默认值）

参    数	说    明
<code>SCRIPT_PATH</code>	参照脚本文件的路径 ( <code>/etc/systemtap/script.d</code> )
<code>CONFIG_PATH</code>	参照配置文件的路径 ( <code>/etc/systemtap/conf.d</code> )
<code>CACHE_PATH</code>	保存缓存的路径 ( <code>/var/cache/systemtap</code> )
<code>TEMP_PATH</code>	工作目录的路径 ( <code>/tmp</code> )
<code>STAT_PATH</code>	保存运行状态的路径 ( <code>/var/run/systemtap</code> )
<code>LOG_FILE</code>	记录运行日志的位置 ( <code>/var/log/systemtap.log</code> )
<code>PASSALL</code>	脚本没有全部启动成功则不显示为 PASS (yes)
<code>RECURSIVE</code>	考虑脚本之间的依存关系执行脚本 (no)
<code>AUTOCOMPIL</code>	检测到内核版本升级等时，自动重新编译 (yes)
<code>DEFAULT-START</code>	未指定脚本时，自动执行的脚本。如果没有指定，则执行所有的脚本 ("")
<code>ALLOW-CACHEONLY</code>	即使只有已编译的缓存也执行。一般没有脚本时，忽视缓存 (no)

表 8-19 脚本配置文件的参数

参    数	说    明
<code>&lt;脚本名称&gt;_OPT</code>	向通过 <code>&lt;脚本名称&gt;</code> 指定的脚本赋予的选项。忽视一些选项。另外，自动添加 “ <code>-m &lt;脚本名称&gt;-F</code> ” 的选项
<code>&lt;脚本名称&gt;_REQ</code>	指定通过 <code>&lt;脚本名称&gt;</code> 指定的脚本所依存的脚本

下例显示了按照与上一节中文件飞行记录器模式的例子相同的设置登录 `syscalltop.stp` 的方法。

```
# cp syscalltop.stp /etc/systemtap/script.d/
# echo syscalltop_OPT="-o /var/log/tracelog-%F-%T.log -s 256,2" > /etc/
systemtap/conf.d/syscalltop.conf
```

在文件飞行记录器模式下运行时，必须如上例所示使用绝对路径指定文件的输出位置。

## 脚本的启动与结束

`systemtap-initscript` 包含 `/etc/init.d/systemtap`。这个 shell 脚本与 `/etc/init.d` 中的其他脚本同样可以直接使用或通过 `service` 命令使用。

```
# service systemtap start
Compiling systemtap scripts: syscalltop [OK]
# service systemtap status
syscalltop(4956) is running...
```

此外，还可以像下面这样以脚本为单位进行操作。

```
# service systemtap start syscalltop
```

要结束服务，需要像其他服务一样添加 `stop` 选项执行脚本。

```
# service systemtap stop
```

另外，还可以使用 `chkconfig` 等，在启动时自动执行脚本。

```
# chkconfig systemtap on
```

`SystemTap` 服务的初次运行需要花费一些时间。这是因为在初次运行时要编译脚本。但是，从下一次执行时开始就会跳过脚本的编译，执行速度就得到提高。也可以事先进行编译。例如，安装新脚本或者更新内核时，如果事先编译脚本，就可以避免重新启动后服务的启动时间变慢。例如，下面就是对内核 2.6.38 进行脚本编译的示例。

```
# service systemtap compile -r 2.6.38
```

## 分配已编译的脚本模块

大规模的集群系统或为了负载均衡需运用多个服务器时，有时会对所有服务器进行相同的追踪。将脚本复制到 `systemtap-initscript` 的目录下，就可以在各个服务器上编译并执行脚本，但是要编译 `SystemTap` 的脚本，就必须要有内核驱动程序的编译环境、内核的调试信息等。这些的负载和磁盘容量也不能忽视，因此如果要创建相同的东西，最好尽量集中进行。

使用 `systemtap-initscript` 的 `CACHEONLY` 选项，就可以满足这些要求。在各客户端上安装 `systemtap-initscript`，并在 `/etc/systemtap/config` 中添加下列两行。

```
AUTOCOMPILe=no
ALLOW_CACHEONLY=yes
```

在客户端上不进行隐式的编译，且存在已编译的脚本的缓存时就会执行。

在服务器上，将想要分配的脚本安装在 /etc/systemtap/script.d/ 下，在为各个脚本编写配置文件后，使用 /etc/init.d/systemtap compile 进行编译。

```
# cp syscalltop.stp /etc/systemtap/script.d/
# echo syscalltop_OPTS="-o /var/log/tracelog-%F-%T.log -S 256,2" > /etc/
systemtap/conf.d/syscalltop.conf
# service systemtap compile -r 2.6.38
Compiling systemtap scripts: Compiling syscalltop ... done
[ OK ]
# ls /var/cache/systemtap/2.6.38/
syscalltop.ko syscalltop.opts
```

编译成功后，在 /var/cache/systemtap/< 内核版本 >/ 下，生成了已编译的脚本模块和运行该脚本模块所需的选项文件。通过网络将这些文件分配到客户端。

```
# tar czPf cache-syscalltop-2.6.38.tar.gz \
/var/cache/systemtap/2.6.38/syscalltop.* /etc/systemtap/conf.d/syscalltop.
conf
# scp cache-syscalltop-2.6.38.tar.gz stap-client:~/
```

在客户端上，将通过下列方式将传递的文件解压缩到与已编译的服务器相同的位置，并安装完成。

```
# tar xzf cache-syscalltop-2.6.38.tar.gz -C /
```

安装后可以按照下列方式运行并测试。

```
# service systemtap start
Starting systemtap: Warning: no script file(/etc/systemtap/script.d/
syscalltop.stp). Use compiled cache.
Starting syscalltop ... done
[ OK ]
# service systemtap status
syscalltop(1654) is running...
```

这样可以看出，可以使用已编写的缓存在客户端上执行追踪。如果将分配已编译脚本的工具包作为 RPM 包或 deb 包，保持其与内核的依存关系，使用数据包管理工具，管理起来就更加轻松。

## 小结

本节介绍了实际在服务器运用环境中使用 SystemTap 的各种结构。通过使用这些结构，不仅能够简便地将 SystemTap 用于调试，还可以用来获取故障信息等。

## 参考资料

- /usr/share/doc/systemtap-initscript\*/README.systemtap

——Masami Hiramatsu