



华章 IT

P Pearson

Linux/Unix
技术丛书

Linux 性能优化

[美] 菲利普 G. 伊佐特 著 贺莲 龚奕利 译
(Phillip G. Ezolt)

Optimizing Linux Performance
A Hands-On Guide to Linux Performance Tools

- 全面介绍Linux性能优化的策略、方法、工具和技术
- 迅速定位性能瓶颈，选择合适的工具和技术解决实际性能问题



机械工业出版社
China Machine Press

Linux 性能优化

Optimizing Linux Performance
A Hands-On Guide to Linux Performance Tools

[美] 菲利普 G. 伊佐特 著 贺莲 龚奕利 译
(Phillip G. Ezolt)



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Linux 性能优化 / (美) 菲利普 G. 伊佐特 (Phillip G. Ezolt) 著；贺莲，龚奕利译。—北京：机械工业出版社，2017.3
(Linux/Unix 技术丛书)

书名原文：Optimizing Linux Performance: A Hands-On Guide to Linux Performance Tools

ISBN 978-7-111-56017-3

I. L… II. ①菲… ②贺… ③龚… III. Linux 操作系统 IV. TP316.85

中国版本图书馆 CIP 数据核字 (2017) 第 029112 号

本书版权登记号：图字：01-2015-5271

Authorized translation from the English language edition, entitled Optimizing Linux Performance : A Hands-On Guide to Linux Performance Tools, 9780131486829 by Phillip G. Ezolt, published by Pearson Education, Inc., Copyright © 2005.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2017.

本书中文简体字版由 Pearson Education (培生教育出版集团) 授权机械工业出版社在中华人民共和国境内（不包括香港、澳门特别行政区及台湾地区）独家出版发行。未经出版者书面许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education (培生教育出版集团) 激光防伪标签，无标签者不得销售。

Linux 性能优化

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：唐晓琳

责任校对：李秋荣

印 刷：北京诚信伟业印刷有限公司

版 次：2017 年 4 月第 1 版第 1 次印刷

开 本：186mm×240mm 1/16

印 张：15

书 号：ISBN 978-7-111-56017-3

定 价：69.00 元

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) 88379426 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

The Translator's Words 译者序

一个运行缓慢的应用程序有时会让人抓狂，此时需要在问题诊断的基础上进行性能调整。本书将帮助你一步步地解决这个难题，告诉你如何发现并修复性能问题。

本书第1章介绍了查找性能问题的基本方法，之后用若干章分别介绍了各种工具，涉及的性能问题包括系统CPU、用户CPU、内存、网络I/O以及磁盘I/O等多个方面。在介绍各种工具时，除了介绍工具的度量对象、使用方法和相关参数选项之外，还附上了一些例子演示其用法。如果一个工具可以用于多种问题，那么将会在相关的每一章中都看到它。第10章到第12章给出了综合性的、面向实际问题的案例，有助于读者在自己解决问题时选择和使用这些工具。

本书组织结构清晰明了，读者可以根据自己的经验水平选择所需章节阅读。本书不仅能让读者学习到性能调整的各个方面，还可以作为性能工具手册使用。

在此感谢机械工业出版社华章公司的编辑朱劼和唐晓琳，感谢她们耐心细致的工作，以及在翻译过程中给予我们的支持和帮助。

在翻译中我们秉持认真细致的态度，但是由于能力所限，还是会存在错误与疏漏，希望广大读者批评指正。

贺莲 龚奕利

前　　言 *Preface*

为什么性能很重要？

如果你曾经坐等计算机完成工作（同时还伴随着敲打桌面、诅咒和好奇：“啥事儿要花这么长的时间？”），你就会知道有个速度快且性能优化良好的计算机系统是多么重要。尽管不是所有的性能问题都能轻易得到解决，但是，了解系统工作缓慢的原因，就意味着有可能采用不同的解决方法：修复软件问题，升级慢速硬件，或者干脆直接把计算机扔出窗外。幸运的是，大多数操作系统，尤其是 Linux，都提供了工具用于检测机器运行缓慢的原因。使用一些基础工具，就可以确定系统中哪里速度慢，并修复那些运行效率低的部分。

虽然终端用户非常讨厌速度慢的系统，但对于应用程序开发者而言，他们有着更重要的理由对其程序进行性能调优：程序能够在多个系统上高效运行。如果你编写的程序运行缓慢，又需要快速的计算机，那么你就会排除掉那些拥有慢速计算机的用户。毕竟，并非所有人都具备最新的硬件。性能良好的应用程序能被更多的用户使用，从而带来更大的潜在用户群。另外，如果潜在用户必须在两个具有相似功能的不同应用程序中进行选择，他们通常会选择运行更快或效率更高的那一个。最后，长期使用的应用程序很可能会经过几轮优化，以便适应不同的用户需求，因此，关键是了解如何追踪性能问题。

如果你是系统管理员，那么对系统用户来说，你就有责任使系统在运行时保持适当的性能水平。若系统运行缓慢，用户就会抱怨。如果你能迅速找到并解决问题，他们就会停止抱怨。还有让人高兴的是，如果你能通过调整应用程序或操作系统来解决问题（从而使他们不用购买新的硬件），那么公司的会计就会很开心。知道如何有效使用性能工具就意味着，在性能问题上需要花费的时间是有区别的：几天，还是几个小时。

Linux：优势和劣势

如果你使用 Linux，维护它并用其进行开发，你就会处于一种奇特但良好的处境中。你能访

问和接触的源代码、开发者和邮件列表是前所未有的，通常，这些邮件列表中会记录着多年前所设计决策。Linux 是发现和修复性能问题的优良环境。与之形成鲜明对比的是专有环境，在这种环境下，很难直接接触到软件开发者，同时也很难找到大多数设计决策讨论的书面记录，而访问源代码则几乎是不可能的。除了是一个高效环境外，Linux 还具备强大的性能工具，使你能发现并修复性能问题。这些工具可以与那些专门的工具相媲美。

即使有着这些令人印象深刻的优势，Linux 生态环境还是需要征服一些挑战。Linux 性能工具分散性很强。不同的小组根据不同的目标开发工具，其结果就是，这些工具不一定集中在一个地方。有些工具已经包含在标准的 Linux 发行版中，如 Red Hat、SUSE 和 Debian；而有些工具则分散在整个互联网上。如果你尝试解决一个性能问题，首先要做的是了解你需要的工具是否存在，然后再设法找到它们。由于没有哪一个 Linux 性能工具能够独立解决所有类型的性能问题，因此，还必须了解如何使用多个工具来确定问题出在哪儿。这可能需要点技巧，但是经验会让它变得容易些。虽然大多数常见的方法会有文档记录，但是 Linux 没有任何指南来告诉你如何整合性能工具以实际解决问题。很多工具或子系统都有调整特定子系统的信息，但是却没有说明如何将它们与其他工具一起使用。许多性能问题涉及系统的多个部分，如果不知道如何同时使用多个工具，就无法解决这些问题。

本书对你有何帮助？

从本书可以学到很多东西，包括：

- 各种性能工具能测量什么。
- 怎样使用每一种工具。
- 如何将工具组合起来解决性能问题。
- 如何从性能欠佳的系统入手，查明问题。
- 如何利用学到的方法来解决现实世界的问题（案例研究）。

利用本书提供的方法，你可以将组织严密的问题诊断说明发送给最初的开发人员。运气好的话，他们会帮你把问题解决掉。

为什么要学习使用性能工具？

为什么要花精力去调整系统或应用程序？

- 性能良好的系统能用更少的资源完成更多的工作。
- 性能良好的应用程序能在更老旧的硬件上运行。

- 性能良好的桌面系统能节约用户时间。
- 性能良好的服务器能为更多用户提供更高质量的服务。

了解如何高效地诊断性能问题，就可以用正确的方法来解决问题，而不是盲目地采取措施并希望它能起作用。如果你是应用程序开发者，就意味着你能快速发现是哪段代码引发了问题；如果你是系统管理员，就意味着你可以找到系统的哪个部分需要调整或升级，而不用浪费时间且徒劳无功地尝试各种解决方案；如果你是终端用户，你就能发现哪些应用程序速度滞后，并将问题报告给开发者（或者必要时更新你的硬件）。

Linux 现在正处于十字路口。高效系统的大部分功能已经完成，对 Linux 及其应用程序来说，下一步就是调优以便与其他操作系统的性能进行竞争，并超越它们。有些性能优化早已开始。例如，SAMBA、Apache 和 TUX Web 服务器项目已经花费了大量的时间，对系统和代码进行调整和优化。其他性能优化（如能显著提升线程性能的本地 POSIX 线程库（NPTL），以及能改善应用程序启动时间的对象预链接）正开始被整合到 Linux 中。Linux 提升性能的时机已然成熟。

我也能进行性能调整吗？

性能优化最大的优点是：你无须了解整个应用程序或系统的详细信息，就可以有效地修复性能问题。性能优化所需的技术与典型应用程序开发者的技术是相辅相成的。

你需要的是细心和耐心。追踪并解决性能问题与其说需要的是程序员，还不如说需要的是侦探。发现并修复这些问题令人兴奋。开始的时候，系统会很糟糕。但是，当你找到原因，并将其连根拔掉后，运气好的话，系统运行速度能达到原来的两倍。完美！

要达到完美，就必须了解强大但有时又颇令人迷惑的 Linux 性能工具。这需要花些功夫，但最后你会发现一切都是值得的。性能工具能向你展示超乎你预期的应用程序和系统的方方面面。

谁应该读这本书？

本书帮助 Linux 软件开发人员、系统管理员和终端用户利用 Linux 性能工具在给定系统中找出性能问题。初级性能研究员能学习性能调查和分析的基础知识。中高级性能研究员，尤其是那些已具备其他专有操作系统性能经验的，能学习那些与他们已经熟悉的系统中的命令等价的 Linux 命令。

软件开发人员能学习如何精确定位引发性能问题的代码行。对系统进行性能调优的系统管理员，则能学习能说明系统变慢原因的工具，然后利用这些信息调整系统。最后，终端用户（虽然不是本书的主要对象）可以学习必要的基本技术以找出哪些应用程序正在消耗系统资源。

本书是如何组织的？

本书向具备不同程度经验的读者教授如何发现并修复性能问题。为了实现这个目标，读者可以挑选本书不同的部分进行阅读，而不必直接看完整本书。

第 1 章介绍查找性能问题的基本方法。其中包含一系列非 Linux 特有的技巧和建议，它们已被证明对追踪性能问题是有效的。这些指南是性能问题查找的常用建议，可以用于追踪任何类型计算机系统的性能问题。

第 2 章到第 8 章（本书主要部分）覆盖了各种工具，可用于度量 Linux 系统中不同的性能统计信息。这些章解释了不同工具度量的对象以及如何调用它们，并为每个工具提供了使用示例。每一章演示的工具分别度量了 Linux 系统的不同部分，如系统 CPU、用户 CPU、内存、网络 I/O 以及磁盘 I/O。如果一种工具涉及多个子系统，它就会出现在多个章节中。每章都会介绍多个工具，但在给定章节中，只会描述适合特定子系统的对应的工具选项。描述格式如下：

1. 概述——这部分解释了工具度量的对象及其使用方法。
2. 性能工具选项——这里不是对工具文档的老调重弹。相反，它说明了哪些选项与当前主题相关，以及这些选项的含义是什么。比如，有些性能工具手册指明了工具度量的事件，但是却没有解释这些事件的含义。本书则说明了事件含义，以及事件与当前子系统的关系。
3. 示例——这部分为度量性能统计信息的工具提供一个或多个例子，展示了调用的工具以及生成的所有输出。

第 9 章针对 Linux，它介绍了面对低性能 Linux 系统时要采取的一系列步骤，以及如何正确使用之前描述的 Linux 性能工具来查明产生性能问题的原因。如果你想从行为异常的 Linux 入手，仅仅只是进行问题诊断，而不想了解工具的详情，那么这一章就是最有用的一章。

第 10 章到第 12 章为案例研究，将前面章节描述的方法和工具结合起来，解决现实世界的问题。案例研究突出了用于发现和修复各类性能问题的 Linux 性能工具，包括以下几类：CPU 密集型应用程序，延迟敏感型应用程序，以及 I/O 密集型应用程序。

第 13 章对性能工具进行了总结，并展望了 Linux 性能调优工具的发展机遇。

本书有两个附录：附录 A 用一个表格收录了书中介绍的性能工具，给出了每一种工具最新版本的 URL，并指明了每种特定的工具都由哪些 Linux 发行版支持；附录 B 说明了如何安装oprofile，该工具包含在几个主要的 Linux 发行版中，其功能强大，但安装困难。

致 谢 *Acknowledgements*

首先，感谢 Prentice Hall 的优秀员工，他们是：Jill Harry, Brenda Mulligan, Gina Kanouse 以及 Keith Cline。

其次，感谢所有审阅了本书初稿，提出并添加了有价值的技术意见和建议的人，他们是：Karel Baloun, Joe Brazea, Bill Carr, Jonathan Corbet, Matthew Crosby, Robert Husted, Paul Lussier, Scott Mann, Bret Strong 和 George Vish II。我还想感谢所有向我传授性能知识并允许我进行 Linux 优化的人，即使当时 Linux 优化的价值还不明朗，他们是：John Henning, Greg Tarsa, Dave Stanley, Greg Gaertner, Bill Carr 和全体 BPE 工具组（他们对我在 Linux 方面的工作给予了支持与鼓励）。

同时，我还要感谢 SPEC 的各位，他们的引导使我了解到，为什么做好基准测试程序会对整个行业有帮助。我尤其要感谢的是 Kaivalya Dixit，他对基准测试的赤诚令人难忘。

感谢所有帮助我在 Carcassonne 和 Settlers of Catan 等游戏面前保持理智的人们，他们是：Sarah Ezolt, Dave 和 Yoko Mitzel, Tim 和 Maureen Chorma, Ionel 和 Marina Vasilescu, Joe Doucette, Jim Zawisza。

最后，感谢我的家人：Sasha 和 Mischief，他们总是提醒我，散步或使用牙线的时间不能省；Ron 和 Joni Elias，他们总是让我开心；Russell、Carol 和 Tracy Ezolt，他们对我的工作予以支持和鼓励；还有我的妻子 Sarah，她是世界上最理解和支持我的人。

推荐阅读



Linux内核设计与实现（原书第3版）

作者：Robert Love ISBN：978-7-111-33829-1 定价：69.00元

本书基于Linux 2.6.34内核详细介绍了Linux内核系统，覆盖了从核心内核系统的应用到内核设计与实现等各方面内容。主要内容包括：进程管理、进程调度、时间管理和定时器、系统调用接口、内存寻址、内存管理和页缓存、VFS、内核同步以及调试技术等。同时本书也涵盖了Linux 2.6内核中颇具特色的内容，包括CFS调度程序、抢占式内核、块I/O层以及I/O调度程序等。本书采用理论与实践相结合的路线，能够带领读者快速走进Linux内核世界，真正开发内核代码。

Linux内核精髓：精通Linux内核必会的75个绝技

作者：Hirokazu Takahashi 等 ISBN：978-7-111-41049-2 定价：79.00元

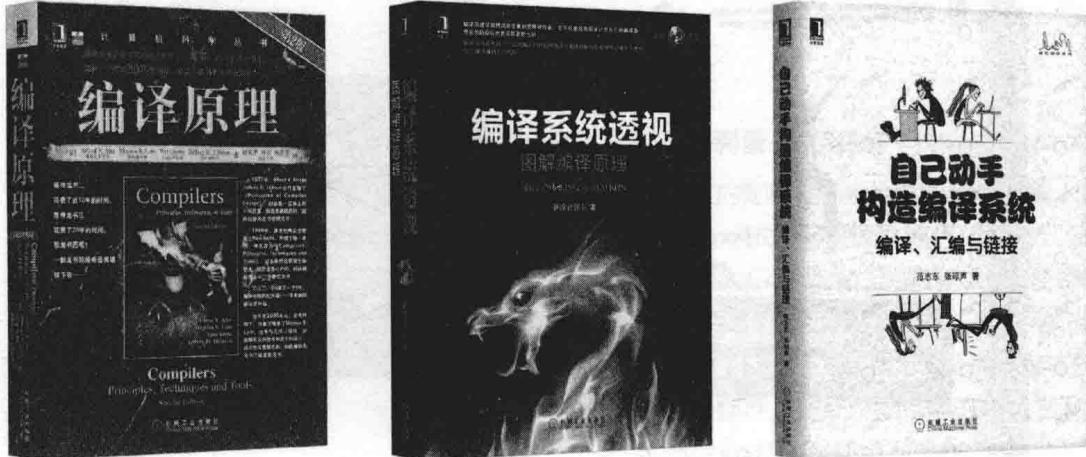
本书从先进Linux内核的众多功能中选取了一些基本而且有趣的内容进行介绍，同时也对内部的运行和结构进行了讲述。此外还介绍了熟练使用这些功能所需的工具、设置方法以及调整方法等。本书还为想要了解Linux内核的读者以及读过本书后开始对Linux内核开发产生兴趣的读者，介绍了获取内核源代码的方法和内核开发方法等内核构建入门所需的信息。

Linux内核设计的艺术：图解Linux操作系统架构设计与实现原理（第2版）

作者：新设计团队 ISBN：978-7-111-42176-4 定价：89.00元

本书的特点在于，既不是空泛地讲理论，也不是单纯地从语法的角度去逐行地分析源代码，而是以操作系统在实际运行中的几个经典事件为主线，将理论和实际结合在一起，精准地再现了操作系统在实际运行中究竟是如何运转的。宏观上，大家可以领略Linux 0.11内核的设计指导思想，可以了解到各个环节是如何牵制并保持平衡的，以及软件和硬件之间是如何互相依赖、互相促进的；微观上，大家可以看到每一个细节的实现方式和其中的精妙之处。

推荐阅读



编译原理（原书第2版）

作者：Alfred V. Aho 等 ISBN：7-111-25121-7 定价：89.00元

本书是编译领域无可替代的经典著作，被广大计算机专业人士誉为“龙书”。本书上一版自1986年出版以来，被世界各地的著名高等院校和研究机构（包括美国哥伦比亚大学、斯坦福大学、哈佛大学、普林斯顿大学、贝尔实验室）作为本科生和研究生的编译原理课程的教材。该书对我国高等计算机教育领域也产生了重大影响。

编译系统透视：图解编译原理

作者：新设计团队 ISBN：978-7-111-49858-2 定价：169.00元

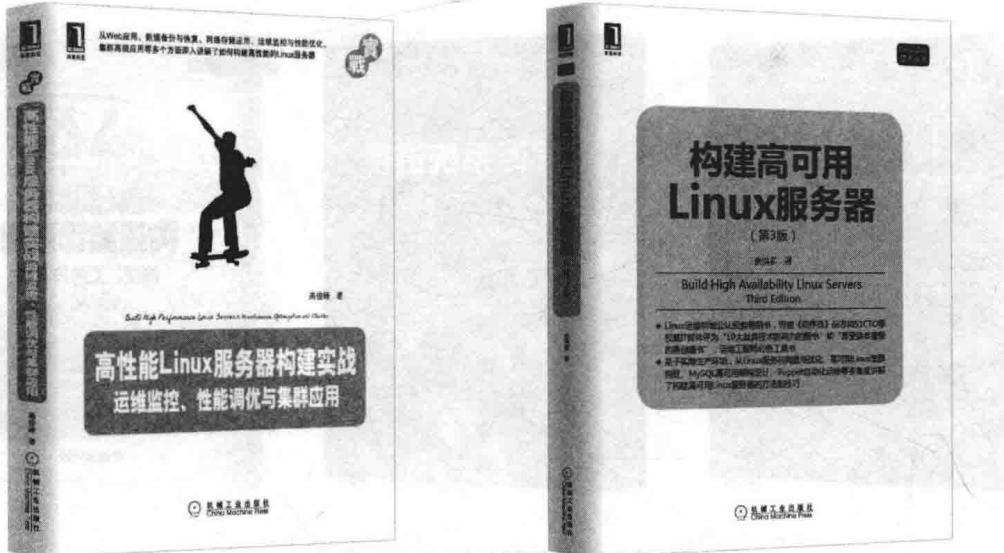
掌握程序内存中的运行时结构对提高程序设计水平的重要性再怎么强调都不过分，将程序员编写的源代码转化为可执行程序是由编译器完成的，编译器对运行时结构的形成起着非常重要的作用。如果你想提高自己的编程水平，了解编译器怎么将你编写的源代码转换为可执行程序的，那么本书就是为你而写的！如果你对编译原理很感兴趣，也很愿意阅读编译器的源代码，却苦于代码量庞大，不知从何下手，那么你必将从本书中得到巨大的收获。

自己动手构造编译系统：编译、汇编与链接

作者：范志东等 ISBN：978-7-111-54355-8 定价：69.00元

本书最大的亮点是它具有很强的应用性和可读性。作者不是从复杂深奥的计算机编译理论入手，而是在各个章节中使用有代表性的程序模块作为范例，将它们放入编译系统中运行以描述它们的编译过程，然后对代码和结果给出详细的诠释。这就像对编译过程进行细致解剖一样。我相信本书会大大降低编译系统理解的门槛，提高读者对编译系统的兴趣。

推荐阅读



高性能Linux服务器构建实战：运维监控、性能调优与集群应用

作者：高俊峰 ISBN：978-7-111-36695-9 定价：79.00元

毫无疑问，Linux服务器是企业级服务系统的主流，随着企业各种数据量的不断增加，企业对服务器系统可靠性、稳定性方面的要求越来越高，越来越突出，高可靠性、高稳定性已经成为评价业务系统性能的主要指标。影响Linux服务器系统性能的因素有很多，改善Linux服务器系统性能的方法和工具也很多，本书紧紧围绕“高性能”这个话题，从Web应用系统、数据备份恢复、网络存储、运维监控、性能优化、集群应用等多方面讲解了构建高性能Linux服务器系统的方法和最佳实践，其中性能优化和集群应用这两个话题是本书的重点。

构建高可用Linux服务器（第3版）

作者：余洪春 ISBN：978-7-111-47787-7 定价：79.00元

本书自第1版出版以来，就广受关注和好评曾被《程序员》杂志和51CTO等权威IT媒体评为“10大最具技术影响力”的图书”和“最受读者喜爱的原创图书”，作者根据运维技术的发展和读者的反馈意见，不断地对书的内容进行优化：更新了过时的技术和方法；补充了最新的内容；限于篇幅，部分内容作为电子版免费提供给读者下载；使得这本书的内容更加完善。

本书最大的特点就是与实践紧密结合，所有理论知识、方法、技巧和案例都来自实际生产环境，涵盖Linux服务器构建与优化、服务器故障诊断与排除、Shell脚本、高可用Linux集群构建、MySQL性能调优及高可用、自动化运维（Puppet）、安全运维等主题，所有内容都围绕“如何构建高可用的Linux服务器”这个主题深度展开。

Contents 目 录

译者序

前 言

致 谢

第 1 章 性能追踪建议 1

1.1 常用建议 2
1.1.1 记大量的笔记 (记录所有的事情) 2
1.1.2 自动执行重复任务 3
1.1.3 尽可能选择低开销工具 4
1.1.4 使用多个工具来搞清楚问题 4
1.1.5 相信你的工具 5
1.1.6 利用其他人的经验 (慎重) 5
1.2 性能调查概要 6
1.2.1 找到指标、基线和目标 6
1.2.2 追踪近似问题 7
1.2.3 查看问题是否早已解决 7
1.2.4 项目开始 (启动调查) 8
1.2.5 记录, 记录, 记录 9
1.3 本章小结 9

第 2 章 性能工具: 系统 CPU 10

2.1 CPU 性能统计信息 10

2.1.1 运行队列统计 10
2.1.2 上下文切换 11
2.1.3 中断 12
2.1.4 CPU 使用率 12
2.2 Linux 性能工具: CPU 12
2.2.1 vmstat (虚拟内存统计) 13
2.2.2 top (2.0.x 版本) 16
2.2.3 top (3.x.x 版本) 20
2.2.4 procinfo (从 /proc 文件系统 显示信息) 24
2.2.5 gnome-system-monitor 26
2.2.6 mpstat (多处理器统计) 27
2.2.7 sar (系统活动报告) 29
2.2.8 oprofile 33
2.3 本章小结 39
第 3 章 性能工具: 系统内存 40
3.1 内存性能统计信息 40
3.1.1 内存子系统和性能 40
3.1.2 内存子系统 (虚拟存储器) 40
3.2 Linux 性能工具: CPU 与内存 42
3.2.1 vmstat (II) 43

3.2.2 top (2.x 和 3.x).....	47	5.2.3 memprof.....	88
3.2.3 procinfo (II).....	49	5.2.4 valgrind (cachegrind).....	90
3.2.4 gnome-system-monitor (II).....	51	5.2.5 kcachegrind	95
3.2.5 free	52	5.2.6 oprofile (III).....	99
3.2.6 slabtop	54	5.2.7 ipcs	103
3.2.7 sar (II).....	55	5.2.8 动态语言 (Java 和 Mono).....	107
3.2.8 / proc / meminfo	58	5.3 本章小结	107
3.3 本章小结	60		
第4章 性能工具：特定进程CPU	61	第6章 性能工具：磁盘I/O	108
4.1 进程性能统计信息.....	61	6.1 磁盘I/O介绍	108
4.1.1 内核时间 vs. 用户时间	61	6.2 磁盘I/O性能工具	109
4.1.2 库时间 vs. 应用程序时间	62	6.2.1 vmstat (III).....	109
4.1.3 细分应用程序时间	62	6.2.2 iostat.....	113
4.2 工具	62	6.2.3 sar (III).....	115
4.2.1 time	62	6.2.4 lsof (列出打开文件).....	117
4.2.2 strace	65	6.3 缺什么	119
4.2.3 ltrace	67	6.4 本章小结	119
4.2.4 ps (进程状态).....	70		
4.2.5 ld.so (动态加载器).....	72		
4.2.6 gprof.....	74		
4.2.7 oprofile (II).....	77		
4.2.8 语言：静态 (C 和 C++) vs. 动态 (Java 和 Mono).....	82		
4.3 本章小结	82		
第5章 性能工具：特定进程内存	83		
5.1 Linux 内存子系统	83		
5.2 内存性能工具.....	84		
5.2.1 ps (II).....	84		
5.2.2 /proc/<PID>.....	85		
		第7章 性能工具：网络	120
		7.1 网络I/O介绍	120
		7.1.1 链路层的网络流量	121
		7.1.2 协议层网络流量	122
		7.2 网络性能工具.....	122
		7.2.1 mii-tool (媒体无关接口工具).....	123
		7.2.2 ethtool	123
		7.2.3 ifconfig (接口配置).....	124
		7.2.4 ip	126
		7.2.5 sar (IV).....	127
		7.2.6 gkrellm	129
		7.2.7 iptraf	131

7.2.8 netstat.....	132	有问题?	155
7.2.9 etherape.....	134	9.3.5 应用程序的磁盘使用有问题?	155
7.3 本章小结	136	9.3.6 应用程序的网络使用有问题?	155
第8章 实用工具：性能工具助手	137	9.4 优化系统	155
8.1 性能工具助手.....	137	9.4.1 系统是受 CPU 限制的吗?	156
8.1.1 自动执行和记录命令.....	138	9.4.2 单个进程是受 CPU 限制的吗?	157
8.1.2 性能统计信息的绘图与分析	138	9.4.3 一个或多个进程使用了大多数	
8.1.3 调查应用程序使用的库.....	138	的系统 CPU 吗?	157
8.1.4 创建和调试应用程序.....	138	9.4.4 一个或多个进程使用了单个	
8.2 工具	139	CPU 的大多数时间?	157
8.2.1 bash	139	9.4.5 内核服务了许多中断吗?	157
8.2.2 tee.....	140	9.4.6 内核的时间花在哪儿了?	158
8.2.3 script	141	9.4.7 交换空间的使用量在增加吗?	158
8.2.4 watch.....	142	9.4.8 系统是受 I/O 限制的吗?	158
8.2.5 gnumeric.....	144	9.4.9 系统使用磁盘 I/O 吗?	158
8.2.6 ldd	146	9.4.10 系统使用网络 I/O 吗?	158
8.2.7 objdump.....	146	9.5 优化进程 CPU 使用情况	159
8.2.8 GNU 调试器 (gdb).....	147	9.5.1 进程在用户还是内核空间	
8.2.9 gcc (GNU 编译器套件).....	149	花费了时间?	160
8.3 本章小结	152	9.5.2 进程有哪些系统调用，完成它们	
第9章 使用性能工具发现问题	153	花了多少时间?	160
9.1 并非总是万灵药.....	153	9.5.3 进程在哪些函数上花了时间?	160
9.2 开始追踪	153	9.5.4 热点函数的调用树是怎样的?	160
9.3 优化应用程序.....	154	9.5.5 Cache 缺失与热点函数或	
9.3.1 内存使用有问题?	154	源代码行是对应的吗?	161
9.3.2 启动时间有问题?	154	9.6 优化内存使用情况	161
9.3.3 加载器引入延迟了吗?	154	9.6.1 内核的内存使用量在增加吗?	161
9.3.4 CPU 使用 (或完成时长)		9.6.2 内核使用的内存类型是什么?	161
		9.6.3 特定进程的驻留集大小在	
		增加吗?	162

第 10 章 性能追踪 1：受 CPU 限制的应用程序 (GIMP).....	169
10.1 受 CPU 限制的应用程序	169
10.2 确定问题	170
10.3 找到基线 / 设置目标.....	170
10.4 为性能追踪配置应用程序	171
10.5 安装和配置性能工具	172
第 11 章 性能追踪 2：延迟敏感的应用程序 (nautilus).....	183
11.1 延迟敏感的应用程序	183
11.2 确定问题.....	184
11.3 找到基线 / 设置目标.....	184
11.4 为性能追踪配置应用程序	186
11.5 安装和配置性能工具	186
11.6 运行应用程序和性能工具	187
11.7 编译和检查源代码.....	191
11.8 使用 <code>gdb</code> 生成调用跟踪	193
11.9 找到时间差异.....	197
11.10 尝试一种可能的解决方案	197
11.11 本章小结.....	199
第 12 章 性能追踪 3：系统级迟缓 (prelink).....	200
12.1 调查系统级迟缓.....	200
12.2 确定问题	200
12.3 找到基线 / 设置目标.....	201
12.4 为性能追踪配置应用程序	204
12.5 安装和配置性能工具	204
9.6.4 共享内存的使用量增加了吗?	163
9.6.5 哪些进程使用了共享内存?	163
9.6.6 进程使用的内存类型是什么?	163
9.6.7 哪些函数正在使用全部的栈?	163
9.6.8 哪些函数的文本大小最大?	163
9.6.9 进程使用的库有多大?	164
9.6.10 哪些函数分配堆内存?	164
9.7 优化磁盘 I/O 使用情况.....	164
9.7.1 系统强调特定磁盘吗?	165
9.7.2 哪个应用程序访问了磁盘?	165
9.7.3 应用程序访问了哪些文件?	165
9.8 优化网络 I/O 使用情况.....	165
9.8.1 网络设备发送 / 接收量接近理论极限了吗?	166
9.8.2 网络设备产生了大量错误吗?	167
9.8.3 设备上流量的类型是什么?	167
9.8.4 特定进程要为流量负责吗?	167
9.8.5 流量是哪个远程系统发送的?	167
9.8.6 哪个应用程序套接字要为流量负责?	167
9.9 尾声	168
9.10 本章小结	168
10.6 运行应用程序和性能工具	172
10.7 分析结果	173
10.8 转战网络	177
10.9 增加图像缓存.....	179
10.10 遇到 (分片引发的) 制约	179
10.11 解决问题	180
10.12 验证正确性	181
10.13 后续步骤	181
10.14 本章小结	182

12.6 运行应用程序和性能工具	205	调用树	217
12.7 模拟解决方案	209	13.2.3 漏洞 3: I/O 的归因	218
12.8 报告问题	212	13.3 Linux 的性能调优	218
12.9 测试解决方案	214	13.3.1 可用的源代码	218
12.10 本章小结	215	13.3.2 容易联系开发者	218
第 13 章 性能工具：下一步是什么	216	13.3.3 Linux 还年轻	219
13.1 Linux 工具的现状	216	13.4 本章小结	219
13.2 Linux 还需要什么样的工具	216	附录 A 性能工具的位置	220
13.2.1 漏洞 1: 性能统计信息分散	217	附录 B 安装 oprofile	222
13.2.2 漏洞 2: 没有可靠并完整的			

性能追踪建议

没有远见和规划，这样解决性能问题是痛苦的。产生问题的原因一次又一次从指尖溜走，不仅浪费时间并且让你备受挫败。但是，如果按照正确的步骤，就可以把令人沮丧的性能追踪转变为有趣的侦探故事。每一条信息都让你更接近问题的根源。人不可能总是可信的，证据将是你唯一的朋友。当你开始研究问题的时候，会遇到不寻常的波折，追踪之初发现的信息最后可能会帮你解决问题。最棒的部分就是，当你最终逮住“坏小子”并修复问题时，会感觉到肾上腺素的刺激和成就感。

如果你从来没有调查过性能问题，那么第一步会是决定性的。不过，听从下面几个明显或隐晦的建议，可以节约时间，并按照自己的方式来找出性能问题的原因。本章目标是提供一系列建议和指导来帮助读者追踪性能问题。在研究系统或应用程序出现的问题时，这些建议告诉你怎样避开一些常见的陷阱。这些建议，大多数都是从浪费的时间和令人沮丧的死胡同中辛苦得到的教训，它们有助于快速并有效地解决你的性能问题。

阅读本章后，你将能够：

- 避免重复他人的工作。
- 避免重复自己的工作。
- 避免因收集的误导信息而导致的虚假线索。
- 为你的研究创建有用的参考文档。

尽管所有的性能调查都是有瑕疵的（“如果一开始就能想到”是你的口头禅），但这些建议将会帮助你避免性能研究中的一些常见错误。

1.1 常用建议

1.1.1 记大量的笔记（记录所有的事情）

在调查性能问题时，你可以做的最重要的事情大概就是记录下看到的每一个输出、执行的每一条命令，以及研究的每一个信息。结构清晰的记录能让你只查看记录就可以检验关于性能问题原因的猜想，而不是重新运行测试。这能节约大量时间。写下来并且创建性能记录。

在性能调查之初，我通常会为其创建一个目录，在 GNU Emacs 中打开一个新的“Notes”文件，开始记录系统的信息。之后，将性能结果保存到这个目录，并将有意思的相关信息保存到 Notes 文件。建议将下面的内容添加到你的性能调查文件和目录中：

- **记录硬 / 软件的配置情况**——记录下的信息包括硬件配置（主存容量、CPU 类型、网络和磁盘子系统）和软件环境（OS 和软件的版本、相关配置文件）。这些信息看上去很容易在之后重现，但是在追踪问题时，你可能会大幅度地修改系统配置。认真细致的笔记有助于在特定的测试过程中弄清楚系统配置。
示例：每次测试时，保存 cat /proc/pci、dmesg 和 uname -a 的输出。
- **保存并组织性能结果**——运行很长时间后还能评估性能结果是很有价值的。记录系统配置的同时，也记录测试结果。这使你得以比较不同的配置是如何影响性能结果的。如果需要，可以重新运行测试，但是测试一种配置是耗费时间的过程。只需让笔记保持条理清晰，避免重复工作则效率更高。
- **写下命令行调用**——在运行性能工具时，常常需要用困难复杂的命令行来准确定位到你感兴趣的系统区域进行测量。如果想重新测试，或在不同的应用程序上运行相同的测试，那么，复制这些命令行不仅令人厌烦，并且在初次尝试时，不容易做对。更好的办法是准确记录下你键入的信息。这样在之后的测试中就能够完全重现命令行，而在回顾之前测试结果时，也可以看到你测量的内容。Linux 命令 script（详见第 8 章）或者从终端“剪切粘贴”都是完成这项工作的好方法。
- **记录研究信息和 URL**——调查性能问题时，将在互联网上发现的相关信息记录下来是很重要的，不论发现的途径是电子邮件，还是人际交往。如果你找到一个看上去相关的网站，就把它剪切粘贴到你的笔记中。（网站是会消失的。）当然，还要记录下 URL，因为你可能在之后还要查看这个网页，或者网页所指信息对后面的调查会变得重要起来。

在收集和记录所有这些信息时，你可能会疑惑：这样做值得吗？有些信息眼下显得毫无作用或有误导性，但它在将来可能是有用的。（好的性能调查就像一部优秀的侦探剧：尽

管开始的时候所有的线索都令人迷惑，但最终都会真相大白。) 在调查问题时，请牢记以下几点：

- 结果的含义可能是不明确的——性能工具给你的信息并不总是清晰明了的。有的时候，你需要更多的信息才能理解某个结果的含义。之后，你可以回过头以新的视角重新审视那些看似无用的测试结果。实际上，旧信息可能会驳斥或者证明关于性能问题本质的某个特定理论。
- 所有的信息都是有用的（这也就是你要记录的原因）——记录已运行的测试信息以及系统配置信息的原因不见得会立即明晰。这一点在你试图向开发人员或管理人员解释系统性能不佳的原因时是非常有用的。通过记录和整理调查过程中你所见的一切，你就有证据支持特定理论，同时也具备大量的测试结果来证明或驳斥其他理论。
- 定期回顾你的笔记可以得到新的想法——当你为性能问题积攒了大量的信息时，那就定期回顾它们。重新审视会让你关注结果，而不是测试。当许多测试结果放在一起被同时查看时，问题的原因也许就会自动浮现。回顾你收集的数据，就可以在不实际运行任何测试的情况下进行理论检验。

在调查问题时，重做一些工作虽然是不可避免的，但是，在重做工作上花费的时间越少，你的效率就越高。如果你写了大量的笔记，并有办法在发现信息时记录它们，那么你就可以依赖已经做过的工作，而避免重复运行测试以及重复研究。保持笔记的可靠性和一致性，从而节省时间减少挫折。

例如，在调查性能问题后，最终确定为硬件原因（主存慢、CPU 慢等），你可能会想通过升级慢速硬件，并重新运行测试来检验这个想法。通常要花一点时间才能获得新硬件，而在你可以重新运行测试之前可能已经过了很久。当最终可以开始的时候，你想要在新老硬件上运行同样的测试。如果你已经保存了之前的测试调用和测试结果，那么，你马上就知道要怎样为新硬件进行测试设置，同时也可比较新的结果与保存的旧结果。

1.1.2 自动执行重复任务

当开始调整系统改进性能时，键入复杂命令行很容易出现错误，而无意中使用的不正确参数和配置则会产生误导性的性能信息。因此，自动执行性能工具调用和应用程序测试是一个好办法：

- 性能工具调用——有些 Linux 性能工具的命令行相当复杂，给自己省点力，把它们保存到一个 shell 脚本中，或是将所有命令都放到可以进行剪切粘贴的参考文件中。这可以让你少受些挫折，并且让你多少有些信心：用来调用工具的命令行是正确的。
- 应用程序测试——大部分应用程序有着复杂的配置，要么通过命令行，要么通过配置文件。你会经常重新运行要多次测试的应用程序，若将调用保存为脚本，就能少

走弯路。虽然刚开始的时候，键入 30 个字符的命令看上去很容易，但这样的操作重复 10 次后，你就会向往自动执行了。

尽可能多地自动执行，就能减少错误。使用脚本自动执行，可以节省时间，并有助于避免因不当工具和测试调用造成的误导性信息。

举个例子，你想在特定工作负载下或某段时间内监控系统，但是在测试结束时，你可能不在现场。这种情况下，脚本就很好用了，在测试完成时，可以自动收集、命名、保存全部生成的性能数据，并将它们自动放到“Results”目录中。有了这些基础之后，你就能按照不同的优化和调整重新运行测试，并且不用担心数据是否已经保存好。你反而可以集中精力找出问题的原因，而不是去管理测试结果。

1.1.3 尽可能选择低开销工具

一般情况下，观察系统会修改系统的行为。(对物理爱好者来说，这就是海森堡(Heisenberg)不确定性原理。)

具体而言，在使用性能工具时，它们会改变系统的行为方式。调查问题的时候，你想要看看应用程序是如何执行的，同时还必须处理性能工具引发的错误。这是不可避免的弊端，但是你要知道它的存在，并努力将其最小化。有些性能工具能够给出高度精确的系统信息，但其检索信息的开销也很高。高开销工具对系统行为带来的变化大于低开销工具。如果你只需要了解系统的粗略信息，那么使用低开销的工具是更好的选择，即使它们不够准确。

例如，对于正在使用的应用程序，工具 ps 能给出其主存数量和类型的相当不错但粗糙的概况。那些准确性更高，但是影响较大的工具，如 memprof 或 valgrind，虽然也能提供这些信息，但是它们消耗的主存和 CPU 资源比只使用原始应用程序要大，因此会改变系统行为。

1.1.4 使用多个工具来搞清楚问题

虽然在找出性能问题原因的时候，如果只需要用一个工具那将是非常方便的，但这种情况相当少见。实际上，你使用的每一种工具都会为问题的原因提供线索，因此，你必须同时使用多个工具来真正搞清楚发生了什么。比如，一种性能工具会告诉你系统存在大量的磁盘 I/O，而另一种工具则告诉你系统使用了大量的交换。如果只以第一个工具的结论制定解决方案，你可能会简单地选择更快的磁盘驱动器（然后发现性能问题仅仅改善了一点点）。而将两种工具的结果放在一起，你就会判断出：大量的磁盘 I/O 是由大量使用的交换造成的。在这种情况下，你可能会买更多的主存以减少交换（这样就不会再有大量的磁盘 I/O）。

比起单一地使用任何一种工具，同时使用多个性能工具通常能让你对性能问题有更清晰的了解。

寓言：盲人摸象

三个盲人在一头大象旁边，想要搞清楚它长什么样子。第一个人拉住了尾巴，说道：“大象就像一根绳子。”第二个人摸到了象腿，说道：“大象像一棵树。”第三个人摸到了大象一侧的身体，说道：“大象像一堵厚实的墙。”

显然，没有一个人得出了正确的答案。如果他们将各自的印象进行共享和组合，那么，他们就可能发现大象真正的模样。别做摸象的盲人。同时使用多种性能工具找出问题的原因。

1.1.5 相信你的工具

性能追踪的过程中，最令人兴奋又令人沮丧的时刻之一，就是工具显示了一个“不可能”的结果。某些“不会”发生的事情却明明白白地发生了。第一反应会认为工具坏了。不要被直觉愚弄了，工具是公正的。虽然它们可能会不正确，但这更有可能是因为应用程序做了不该做的事情。要使用工具来调查问题。

举个例子，Gnome 计算器使用超过 2000 个系统调用只为了实现加载和退出。如果没有性能工具来证明这个事实，那么，仅仅为了启动和停止应用程序就需要如此之多的系统调用看上去是没有必要的。但是性能工具能够显示其发生的位置和原因。

1.1.6 利用其他人的经验（慎重）

在调查任何一个性能问题时，你可能会发现问题令人不知所措。不要独自面对它。问问开发者是否见过同样的问题。试着找到其他解决过你所遇问题的人。在互联网上搜索类似的问题，并希望找到解决方案。给用户和开发人员发电子邮件。

本条建议附带一个提醒：即使开发者认为了解自己的应用程序，他们也不见得总是对的。如果开发者不认同性能工具的数据，那么，他们也许是错的。向开发者展示你的数据以及你为何会得出这样的结论。他们通常会帮你重新解释数据或者解决问题。不论是哪种情况，都会将你的调查向前推进一些。如果你的数据表明发生了不该发生的事情，就不要害怕与开发者有分歧。

比如，你通常可以按照在 Google 上搜索类似问题得到的指导来解决性能问题。很多时候，在调查一个 Linux 问题时，你会发现之前已经有人遇到过了（即使是几年前），而且还在公共邮件列表中报告了解决方法。使用 Google 是很容易的，它可以为你节省几天的工作量。

1.2 性能调查概要

本节列出了开始性能调查时的几个重要步骤。由于终极目标是解决问题，因此最好的方法是在你接触性能工具之前就开始研究问题。遵循如下特定步骤是解决问题的有效方法，并且不会浪费宝贵的时间。

1.2.1 找到指标、基线和目标

性能调查的第一步就是确定当前的性能，并明确其应提升的程度。如果你的系统明显性能不佳，你就可以确定值得花时间进行研究。但是，如果系统性能接近其峰值，那么就不值得研究。明确性能峰值有助于你设置合理的性能期望值，并能给你一个性能目标，这样你就知道何时应该停止优化。你可能总是没有目标地时不时对系统做一点调整，这会浪费大量的时间，只为得到一些额外的性能，即使你可能并不真的需要它们。

1.2.1.1 确定指标

要想知道什么时候结束优化，你必须为系统自行确立或是使用已发布的指标。指标是一种客观的度量，用于指示系统的执行情况。例如，如果你要优化一个Web服务器，你就可以选择“每秒服务的Web请求数”。如果你没有一个客观的途径来度量性能，那么在调整系统的时候，你几乎无法确定是否取得了进展。

1.2.1.2 确定基线

在你明确了如何度量特定系统或应用程序的性能之后，确定当前的性能等级就很重要了。在调整和优化之前，运行应用程序并记录其性能，这就是基线值，它是性能调查的起点。

1.2.1.3 确定目标

在你确定了性能指标和基线后，现在需要确定一个目标，这个目标引导你完成性能追踪。你可以无限期地调整系统，花费越来越多的时间来获得更加好一点的性能。如果你制定了目标，那么你就会知道什么时候该结束整个过程。要选择合理的目标，下面是一些好的起点：

- 寻找其他有相同配置的人，询问他们的性能指标——这是理想状态。如果你能发现某人拥有相似的系统和更好的性能，则不仅能为你的系统选定目标，还能与这个人一起工作：他可以确定为什么你的配置更慢，以及该配置有何不同。在研究问题时，使用另一个系统作为参照被证明是非常有用的。
- 查找工业标准测试程序的结果——许多网站都比较了计算机系统各方面的基准测试结果。有些基准测试结果是经过异常的努力得到的，因此，它们可能不能代表真实的使用情况。不过，很多基准测试网站给出了特定结果使用的配置，这些配置信息

可以为你调整系统提供线索。

- 在不同的 OS 或应用程序上使用你的硬件——可能要在你的系统上运行不同的软件以实现相似的功能。比如，如果你有两个不同的 Web 服务器，其中一个运行较慢，那么就试试另一个，看它的性能是否好一些。或者，在另一个不同的操作系统上运行同一个应用程序。如果上述任一情况下系统执行表现得更好一些，那么，你就会知道原来的应用程序还有改进的空间。

如果用现有的性能信息来指导你的目标，那么你有更好的机会来选择一个积极的，但也并非不可能达到的目标。

抓住低处的果实

性能追踪的另一种方法是选择在特定时间段内进行追踪，而不是选择一个目标，在这段时间内尽可能地对性能进行优化。如果应用程序从未被优化过，那么通常在给定工作负载下，会有一些问题相对容易解决。这些容易被修复的问题称为“低处的果实”。

为什么是“低处的果实”？打个比方，将性能调查想象成你饿了，正站在一棵苹果树下。你会采摘最靠近地面，也是你最容易够到的苹果。这些低处的苹果与果树稍高处较难够到的苹果一样能够填饱你的肚子，但是采摘它们只需花费很少的力气。相似的，如果你要在有限的时间内优化一个应用程序，你可能会试着修复那些最简单明显的问题（低处的果实），而不是做一些更困难的、根本性的变化。

1.2.2 追踪近似问题

使用性能工具为确定问题原因打开第一个口子。通过初始的粗略尝试，你能对问题形成大致的看法。这个简单切口的目的就是收集足够的信息传递给程序的其他用户和开发者，以便他们提出意见和建议。这里非常重要的一点是要有良好的书面记录来解释你认为问题是怎样的，以及什么样的测试使你得出了这个结论。

1.2.3 查看问题是否早已解决

你的下一个目标是确定是否有其他人已经解决了这个问题。性能调查可能是一个冗长且费时的事情，如果你正好可以利用其他人的工作，那么在你开始之前就将抢占先机。因为你的目的就是要改进系统性能，所以解决性能问题最好的办法就是依靠其他人已有的成果。

尽管你很可能对性能问题的具体建议持保留态度，但是这些建议具有启发性，可以使你了解到其他人可能已经研究过相似的问题，他们是如何试着解决问题的，以及他们是否成功了。

下述这些地方也能寻求性能建议：

- 在 Web 上查找相似的错误信息 / 问题——这通常是我调查的第一步。Web 搜索常常

会揭示很多与应用程序以及你正在查找的具体错误情况相关的信息。它们还可以指向其他用户所尝试的系统优化，还可能提示哪些有作用、哪些没有作用。成功的搜索可以产生好几页能够直接用于你的性能问题的信息。要发现有相似性能问题的人，使用 Google 或 Google 论坛搜索是非常有用的方法。

- 在应用程序邮件列表上求助——大多数流行或公开开发的软件都有软件使用者的邮件列表，这是寻找性能问题答案的绝佳地方。读者和贡献者通常在软件运行以及保持其性能良好方面有经验。搜索一下邮件列表的存档，因为有人可能会问过相同的问题。而随后对原始消息的回复也许就描述了一个解决方案。如果没有这样的回复，就向最初提出这个问题的人发邮件，询问他是否找到了解决方法。如果这样也不行，或是没有其他人提出过相似的问题，那么在列表上发邮件说明你的问题，幸运的话，也许已经有人把这个问题解决了。
- 向开发人员发邮件——很多 Linux 软件在文档的某个位置包含了开发者的 e-mail 地址，如果在互联网和邮件列表中搜索失败，你可以尝试直接给开发者发邮件。开发人员通常非常忙，不见得有时间回复邮件。但是，相比其他人，他们更加了解软件，如果你能向开发者提供对性能问题条理清晰的分析，并愿意和他一起工作，那么开发者也许能帮助你。虽然开发者关于性能问题原因的想法不见得正确，但是他们可能会给你指出一个富有成果的方向。
- 与内部开发人员交谈——最后，如果产品是内部开发的，你就可以与内部开发人员通电话或发邮件。这和与外部开发者联系几乎是一样的，只不过内部人员可能会在你的问题上投入更多时间，或是给你指出内部的知识库。

依靠其他人的工作，你也许在性能调查开始之前就能解决问题。至少，你有可能找到一些有希望的方法来调查，所以，最好总是看看别人有什么发现。

1.2.4 项目开始（启动调查）

现在你已经详细了解了别人解决问题的可能性，接下来必须开始性能调查了。后续章节将详细介绍工具和方法，但是现在还有一些提示能让你的工作效果更好：

- 分离问题——如果可能的话，删去任何运行于被调查系统的多余的程序或应用。运行许多不同应用程序的系统，其负载较重，会影响性能工具收集信息的准确性，并最终将你引导到错误的方向。
- 利用系统差异发现原因——如果你能发现一个相似的系统具有更好的性能，那么这对问题调试将是一个有力的帮助。使用性能工具的问题之一就是，你不一定有好的方法知道性能工具的结果是否指明了问题。如果你有一个好的系统和一个差的系统，你就可以在这两个系统上运行同样的性能工具，并比较它们的结果。如果结果不同，

就可以通过找出系统差异来确定问题的原因。

- 一次只改变一件事——这点非常重要。要真正确定问题出在哪儿，一次只能有一个变化。这可能会很花时间，并让你运行多个不同的测试，但它的确是发现你是否解决了问题的唯一途径。
- 始终在优化后重新测量——如果你稍稍调整了系统，那么在调整后对所有的事情重新进行测量是很重要的。当你开始修改系统配置时，所有之前生成的性能信息可能不再有效。通常，在你解决一个性能问题时，别的问题会随之而来。新问题可能与老问题有着极大的不同，因此，你真的需要重新运行性能工具来确保正在调查的问题没有出错。

遵循这些建议能帮助你避免误导，并有助于确定性能问题的原因。

1.2.5 记录，记录，记录

如前所述，记录你所做的事情以便之后回顾和审查，这一点确实很重要。如果你已经开始追踪性能问题，那么在你的脑海中就会有大量新增的笔记和 URL。它们可能杂乱无章，混成一团，但现在你明白它们的意思，知道它们的组织结构。在解决问题后，花些时间重写你的发现以及为什么你认为这么做是对的。包括测量得到的性能结果和做过的实验。虽然看上去工作量很大，但却是非常值得的。几个月后，曾经做过的测试很容易就会被忘记，如果没有将结果记录下来，最终你可能会重做测试。如果在这些测试还记忆犹新时撰写了报告，你就不用重做这些工作，而只需要依靠这些记录就行。

1.3 本章小结

追踪性能问题应该是个令人满意且兴奋的过程。如果用正确的方法去研究和分析，你 的问题追踪将会事半功倍。首先，确定是否有其他人遇见过相似的问题，如果有，尝试他们的解决方案。要对他们告诉你的保持怀疑，要寻找具有类似问题经验的人。为你的性能追踪设立合理的指标和目标，指标使你知道什么时候应该结束追踪。自动执行性能测试。生成测试结果和配置信息时，要确保将它们记录下来，以便之后可以审查这些结果。保持结果的条理性，记录下任何与你 的问题相关的研究和其他信息。最后，定期回顾你的笔记，找出之前可能被漏掉的信息。如果遵循了这些原则，你 的问题调查将会有个明确的目标和一个清晰的过程。

本章给出了性能调查的基本背景，后续章节将会覆盖 Linux 特有的性能工具。你 将学习如何使用工具，它们可以提供什么类型的信息，以及如何组合使用它们找出特定系统的性能问题。

性能工具：系统 CPU

本章概述了系统级的 Linux 性能工具。这些工具是你追踪性能问题时的第一道防线。它们能展示整个系统的性能情况和哪些部分表现不好。本章将讨论这些工具可以测量的统计信息，以及如何使用各种工具收集这些统计结果。阅读本章后，你将能够：

- 理解系统级性能的基本指标，包括 CPU 的使用情况。
- 明白哪些工具可以检索这些系统级性能指标。

2.1 CPU 性能统计信息

每一种系统级 Linux 性能工具都提供了不同的方式来提取相似的统计结果。虽然没有工具能显示全部的信息，但是有些工具显示的是相同的统计信息。为了不多次（每种工具一次）解释统计信息的含义，我们在描述所有工具之前对这些信息进行一次性说明。

2.1.1 运行队列统计

在 Linux 中，一个进程要么是可运行的，要么是阻塞的（正在等待一个事件的完成）。阻塞进程可能在等待的是从 I/O 设备来的数据，或者是系统调用的结果。如果进程是可运行的，那就意味着它要和其他也是可运行的进程竞争 CPU 时间。一个可运行的进程不一定会使用 CPU，但是当 Linux 调度器决定下一个要运行的进程时，它会从可运行进程队列中挑选。如果进程是可运行的，同时又在等待使用处理器，这些进程就构成了运行队列。运行队列越长，处于等待状态的进程就越多。

性能工具通常会给出可运行的进程个数和等待 I/O 的阻塞进程个数。另一种常见的系统统计是平均负载。系统的负载是指正在运行和可运行的进程总数。比如，如果正在运行的进程为两个，而可运行的进程为三个，那么系统负载就是 5。平均负载是给定时间内的负载量。一般情况下，取平均负载的时间为 1 分钟、5 分钟和 15 分钟。这能让你观察到负载是如何随时间变化的。

2.1.2 上下文切换

大部分现代处理器一次只能运行一个进程或线程。虽然有些处理器（比如超线程处理器）实际上可以同时运行多个进程，但是 Linux 会把它们看作多个单线程处理器。如果要制造出给定单处理器同时运行多个任务的假象，Linux 内核就要不断地在不同的进程间切换。这种不同进程间的切换称为上下文切换，因为当其发生时，CPU 要保存旧进程的所有上下文信息，并取出新进程的所有上下文信息。上下文中包含了 Linux 跟踪新进程的大量信息，其中包括：进程正在执行的指令，分配给进程的内存，进程打开的文件等。这些上下文切换涉及大量信息的移动，因此，上下文切换的开销可以是相当大的。尽量减少上下文切换的次数是个好主意。

要避免上下文切换，重要的一点是了解它们是如何发生的。首先，上下文切换可以是内核调度的结果。为了保证公平地给每个进程分配处理器时间，内核周期性地中断正在运行的进程，在适当的情况下，内核调度器会决定开始另一个进程，而不是让当前进程继续执行。每次这种周期性中断或定时发生时，你的系统都可能进行上下文切换。每秒定时中断的次数与架构和内核版本有关。一个检查中断频率的简单方法是用 /proc/interrupts 文件，它可以确定已知时长内发生的中断次数。如清单 2.1 所示。

清单 2.1

```
root@localhost asm-i386]# cat /proc/interrupts | grep timer
; sleep 10 ; cat /proc/interrupts | grep timer
0: 24060043          XT-PIC  timer
0: 24070093          XT-PIC  timer
```

在清单 2.1 中，我们要求内核给出定时器启动的次数，等待 10 秒后，再次请求。这就是说，在这台机器上定时器启动频率为 (24 070 093-24 060 043) 中断 / (10 秒) 或者约 1000 次中断 / 秒。如果你的上下文切换明显多于定时器中断，那么这些切换极有可能是由 I/O 请求或其他长时间运行的系统调用（如休眠）造成的。当应用请求的操作不能立即完成时，内核启动该操作，保存请求进程，并尝试切换到另一个已就绪进程。这能让处理器尽量保持忙状态。

2.1.3 中断

此外，处理器还周期性地从硬件设备接收中断。当设备有事件需要内核处理时，它通常就会触发这些中断。比如，如果磁盘控制器刚刚完成从驱动器取数据块的操作，并准备好提供给内核，那么磁盘控制器就会触发一个中断。对内核收到的每个中断，如果已经有相应的已注册的中断处理程序，就运行该程序，否则将忽略这个中断。这些中断处理程序在系统中具有很高的运行优先级，并且通常执行速度也很快。有时，中断处理程序有工作要做，但是又不需要高优先级，因此它可以启动“下半部”(bottom half)，也就是所谓的软中断处理程序。如果有许多中断，内核会花大量的时间服务这些中断。查看 /proc/interrupts 文件可以显示出哪些 CPU 上触发了哪些中断。

2.1.4 CPU 使用率

CPU 使用率是个简单的概念。在任何给定的时间，CPU 可以执行以下七件事情中的一个：

- (1) CPU 可以是空闲的，这意味着处理器实际上没有做任何工作，并且等待有任务可以执行。
- (2) CPU 可以运行用户代码，即指定的“用户”时间。
- (3) CPU 可以执行 Linux 内核中的应用程序代码，这就是“系统”时间。
- (4) CPU 可以执行“比较友好”的或者优先级被设置为低于一般进程的用户代码。
- (5) CPU 可以处于 iowait 状态，即系统正在等待 I/O (如磁盘或网络) 完成。
- (6) CPU 可以处于 irq 状态，即它正在用高优先级代码处理硬件中断。
- (7) CPU 可以处于 softirq 模式，即系统正在执行同样由中断触发的内核代码，只不过其运行于较低优先级 (下半部代码)。

此情景出现的条件为：发生设备中断时，而内核在将其移交给用户空间之前必须对其进行一些处理（比如，处理网络包）。

大多数性能工具将这些数值表示为占 CPU 总时间的百分比。这些时间的范围从 0% 到 100%，但全部三项加起来等于 100%。一个具有高“系统”百分比的系统表明其大部分时间都消耗在了内核上。像 oprofile 一样的工具可以帮助确定时间都消耗在了哪里。具有高“用户”时间的系统则将其大部分时间都用来运行应用程序。下一章展示在上述情况下，如何用性能工具追踪问题。如果系统在应该工作的时候花费了大量的时间处于 iowait 状态，那它很可能在等待来自设备的 I/O。导致速度变慢的原因可能是磁盘、网卡或其他设备。

2.2 Linux 性能工具：CPU

现在开始讨论性能工具，使用这些工具能够提取之前描述的那些信息。

2.2.1 vmstat (虚拟内存统计)

vmstat 是指虚拟内存统计，这个名称表明它能告诉你系统的虚拟内存性能信息。幸运的是，它实际上能完成的工作远不止于此。vmstat 是一个很有用的命令，它能获取整个系统性能的粗略信息，包括：

- 正在运行的进程个数。
- CPU 的使用情况。
- CPU 接收的中断个数。
- 调度器执行的上下文切换次数。

它是用于获取系统性能大致信息的极好工具。

2.2.1.1 CPU 性能相关的选项

vmstat 可以被如下命令行调用：

```
vmstat [-n] [-s] [delay [count]]
```

vmstat 运行于两种模式：采样模式和平均模式。如果不指定参数，则 vmstat 统计运行于平均模式下，vmstat 显示从系统启动以来所有统计数据的均值。但是，如果指定了延迟，那么第一个采样仍然是系统启动以来的均值，但之后 vmstat 按延迟秒数采样系统并显示统计数据。表 2-1 解释了 vmstat 的选项。

表 2-1 vmstat 命令行选项

选 项	说 明
-n	默认情况下，vmstat 定期显示每个性能统计数据的列标题。本选项禁止该特性，因此初始列标题之后，只显示性能数据。如果想要将 vmstat 导出为电子表格，使用这个选项是有好处的
-s	本选项一次性输出 vmstat 收集的系统统计的详细信息。该信息为系统启动后的总数据
delay	vmstat 采样的间隔时间

vmstat 提供的各种统计输出信息，使你能跟踪系统性能的不同方面。表 2-2 解释了与 CPU 性能相关的输出。下一章说明与内存性能相关的输出。

表 2-2 与 CPU 相关的 vmstat 输出

列	说 明
r	当前可运行的进程数。这些进程没有等待 I/O，而是已经准备好运行。理想状态下，可运行进程数应与可用 CPU 的数量相等
b	等待 I/O 完成的被阻塞进程数
forks	创建新进程的次数
in	系统发生中断的次数
cs	系统发生上下文切换的次数
us	用户进程消耗的总 CPU 时间的百分比（包括“友好的”时间）
sy	系统代码消耗的总 CPU 时间的百分比，其中包括消耗在 system、irq 和 softirq 状态的时间
wa	等待 I/O 消耗的总 CPU 时间的百分比
id	系统空闲消耗的总 CPU 时间的百分比

vmstat 提供了一个低开销的良好系统性能视图。由于所有的性能统计数据都以文本形式呈现，并打印到标准输出，因此，捕捉测试中生成的数据，以及之后对其进行处理和绘图就会很方便。由于 vmstat 的开销如此之低，因此当你需要一目了然地监控系统健康状况时，让它在控制台上或窗口中持续运行，甚至是在负载非常重的服务器上是很实用的。

2.2.1.2 用法示例

如清单 2.2 所示，如果 vmstat 运行时没有使用命令行参数，显示的将是自系统启动后它记录下的统计信息的均值。根据“CPU 使用率”列下面的 us、sy、wa 和 id，本例显示出系统从启动开始，基本上处于空闲状态。从启动开始，CPU 有 5% 的时间用于执行用户应用程序代码，1% 的时间用于执行系统代码，而其余 94% 的时间处于空闲状态。

清单 2.2

```
[ezolt@scrffy tmp]$ vmstat
procs -----memory----- swap-- io--- system--
----cpu----
r b swpd free buff cache si so bi bo in cs us sy id
wa
1 0 181024 26284 35292 503048 0 0 3 2 6 1 5 1 94 0
```

尽管 vmstat 从系统启动时开始统计有助于确定系统的负载情况，但是，vmstat 最有用的是运行于采样模式下，如清单 2.3 所示。在采样模式下，vmstat 间隔 delay 参数指定的秒数输出系统统计数据，而采样次数由 count 给出。清单 2.3 第一行的统计数据和之前一样，是系统启动以来的均值，但之后就是定期采样。本例展示出系统的活动非常少。通过查看 b 列下面的 0，我们可以知道在运行时没有阻塞进程。通过查看 r 列，我们还可以看到在 vmstat 采样数据时，正在运行的进程数量少于 1。

清单 2.3

```
[ezolt@scrffy tmp]$ vmstat 2 5
procs -----memory----- swap-- io--- system-- ----cpu----
r b swpd free buff cache si so bi bo in cs us sy id wa
1 0 181024 26276 35316 502960 0 0 3 2 6 1 5 1 94 0
1 0 181024 26084 35316 502960 0 0 0 0 1318 772 1 0 98 0
0 0 181024 26148 35316 502960 0 0 0 24 1314 734 1 0 98 0
0 0 181024 26020 35316 502960 0 0 0 0 1315 764 2 0 98 0
0 0 181024 25956 35316 502960 0 0 0 0 1310 764 2 0 98 0
```

vmstat 是一种记录系统在一定负载或测试条件下行为的好方法。可以用 vmstat 显示系统的行为，同时利用 Linux 的 tee 命令将结果输出到文件。(第 8 章详细描述了 tee 命令。)如果你只传递了参数 delay，vmstat 就会无限采样。在测试开始前启动 vmstat，测试结束后终止 vmstat。输出文件的形式可以是电子表格，并能够用于查看系统对负载和各种系统事件是如何反应的。清单 2.4 给出了按照这个方法得到的输出。在这个例子中，我们可以查看

到系统发生的中断和上下文切换。在 in 列和 cs 列能分别查看到中断和上下文切换的总数。

上下文切换的数量小于中断的数量。调度器切换进程的次数少于定时器中断触发的次数。这很可能是因为系统基本上是空闲的，在定时器中断触发的大多数时候，调度器没有任何工作要做，因此它也不需要从空闲进程切换出去。

（注意：生成如下输出的 vmstat 版本有错误。它会导致系统输出的平均线显示不正确的数值。该错误已经报告给了 vmstat 的维护者，希望能尽快修复。）

清单 2.4

```
[ezolt@scrffy ~/edid]$ vmstat 1 | tee /tmp/output
procs -----memory----- ...swap... -----io---- --system-- ----cpu----
r b swpd free buff cache si so bi bo in cs us sy id wa
0 1 201060 35832 26532 324112 0 0 3 2 6 2 5 1 94 0
0 0 201060 35888 26532 324112 0 0 16 0 1138 358 0 0 99 0
0 0 201060 35888 26540 324104 0 0 0 88 1163 371 0 0 100 0
0 0 201060 35888 26540 324104 0 0 0 0 1133 345 0 0 100 0
0 0 201060 35888 26540 324104 0 0 0 60 1174 351 0 0 100 0
0 0 201060 35920 26540 324104 0 0 0 0 1150 408 0 0 100 0
[Ctrl-C]
```

最新版本的 vmstat 甚至可以抽取各种系统统计数据更详细的信息，如清单 2.5 所示。

下一章讨论内存统计数据，但是，现在我们来查看 CPU 的统计信息。第一组数据，即“CPU ticks”，显示的是自系统启动的 CPU 时间，这里的“tick”是一个时间单位。虽然精简的 vmstat 输出仅显示四个 CPU 状态——us、sy、id 和 wa，这里则显示了全部 CPU ticks 的分布情况。此外，我们还可以看到中断和上下文切换的总数。一个新添加的内容是 forks，它大体上表示的是从系统启动开始，已经创建的新进程的数量。

清单 2.5

```
[ezolt@scrffy ~/edid]$ vmstat -s
1034320 total memory
998712 used memory
698076 active memory
176260 inactive memory
35608 free memory
26592 buffer memory
324312 swap cache
2040244 total swap
201060 used swap
1839184 free swap
5279633 non-nice user cpu ticks
28207739 nice user cpu ticks
2355391 system cpu ticks
```

```

628297350 idle cpu ticks
862755 IO-wait cpu ticks
34 IRQ cpu ticks
1707439 softirq cpu ticks
21194571 pages paged in
12677400 pages paged out
93406 pages swapped in
181587 pages swapped out
1931462143 interrupts
785963213 CPU context switches
1096643656 boot time
578451 forks

```

vmstat 提供了关于 Linux 系统性能的众多信息。在调查系统问题时，它是核心工具之一。

2.2.2 top (2.0.x 版本)

top 是 Linux 系统监控工具中的瑞士军刀。它善于将相当多的系统整体性能信息放在一个屏幕上。显示内容还能以交互的方式进行改变，因此，在系统运行时，如果一个特定的问题不断突显，你可以修改 top 显示的信息。

默认情况下，top 表现为一个将占用 CPU 最多的进程按降序排列的列表。这使得你能够迅速找出是哪个程序独占了 CPU。top 根据指定的延迟定期更新这个列表（其初始值为 3 秒）。

2.2.2.1 CPU 性能相关的选项

top 用如下命令行调用：

```
top [d delay] [C] [H] [i] [n iterations] [b]
```

top 实际上有两种模式的选项：命令行选项和运行时选项。命令行选项决定 top 如何显示其信息。表 2-3 给出的命令行选项会影响 top 显示的性能统计信息的类型和频率。

表 2-3 top 命令行选项

选 项	说 明
d delay	统计数据更新的时间间隔
n iterations	退出前的迭代次数。top 更新统计数据的次数为 iterations 次
i	不显示未使用任何 CPU 的进程
H	显示应用程序所有的单个线程，而不仅仅给出每个应用程序的总和
C	对超线程或 SMP 系统，显示 CPU 统计数据总和，而不是每个 CPU 的数据

在你运行 top 时，为了调查特定问题，你可能想要对你的观察略作调整。top 输出的可定制性很高。表 2-4 给出的选项可以在 top 运行期间修改显示的统计信息：

表 2-4 top 运行时统计信息显示选项

选 项	说 明
f 或 F	显示一个配置界面，用于选择在屏幕上显示哪些进程统计信息
o 或 O	显示一个配置界面，用于修改显示统计信息的顺序

表 2-5 给出的选项打开或关闭各种系统级信息的显示。关闭不需要的统计信息有助于在屏幕上显示更多进程。

表 2-5 top 运行时输出切换选项

选 项	说 明
l	切换更新和显示平均负载以及正常运行时间信息
t	切换显示每个 CPU 消耗时间的情况。它还切换显示当前运行的进程数量。显示应用程序全部的独立线程，而不是显示每个应用程序的总数
m	在屏幕上切换显示系统内存使用信息。默认情况下，最占用 CPU 的进程第一个显示。不过，按照其他特征排序可能更有用

表 2-6 对 top 支持的不同排序模式进行了说明。按内存消耗量排序尤其有用，它能找出哪个进程消耗了最多的内存。

表 2-6 top 输出排序 / 显示选项

选 项	说 明
P	按 CPU 消耗量对任务排序。最高的 CPU 用户第一个显示
T	按到目前为止使用的 CPU 时间总量对任务排序。总量最高的第一个显示
N	按任务的 PID 进行排序。PID 最低的第一个显示
A	按任务时长进行排序。最新的 PID 第一个显示。通常与“按 PID 排序”相反
i	隐藏空闲和不消耗 CPU 的任务

top 除了提供特定进程的信息之外，还提供系统整体信息。表 2-7 给出了这些统计信息。

表 2-7 top 性能统计信息

选 项	说 明
us	用户应用程序消耗的 CPU 时间
sy	内核消耗的 CPU 时间
ni	“友好的”进程消耗的 CPU 时间
id	空闲的 CPU 时间
wa	等待 I/O 的 CPU 时间
hi	irq 处理程序消耗的 CPU 时间
si	softirq 处理程序消耗的 CPU 时间
load average	1 分钟、5 分钟和 15 分钟的平均负载
%CPU	特定进程消耗 CPU 时间的百分比
PRI	进程优先级，值越大表示优先级越高。RT 代表任务为实时优先级，该优先级高于标准范围
NI	进程的 nice 值。nice 值越高，系统执行该进程的必要性就越低。对于具有高 nice 值的进程，通常其优先级会非常低

(续)

选 项	说 明
WCHAN	若进程在等待 I/O，该项显示其等待的是哪个内核函数
STAT	进程当前状态。这里，进程可以是睡眠状态 (S)，运行状态 (R)，僵尸状态 (要求终止但还未终止)(Z)，不可中断的睡眠状态 (D)，或者跟踪状态 (T)
TIME	自进程开始执行起已消耗的总的 CPU 时间 (用户和系统)
COMMAND	进程正在执行的命令
LC	进程执行时最后使用的 CPU 编号
FLAGS	该项切换是否更新和显示平均负载与正常运行时间信息

top 提供了不同的正在运行进程的大量信息，是找出资源消耗大户的极好方法。

2.2.2.2 用法示例

清单 2.6 是运行 top 的一个例子。当它启动后，将会周期性地更新屏幕直到退出。该例展示了 top 能生成的一些系统整体统计信息。首先，我们能看到 1 分钟、5 分钟和 15 分钟的系统平均负载。可以看出，系统已经开始忙碌起来（因为 doom-3.x86）。一个 CPU 在用户代码上花费了 90% 的时间。另一个则只在用户代码上花费了约 13% 的时间。最后，我们看到 73 个进程处于睡眠状态，只有 3 个进程正在运行。

清单 2.6

```
catan> top
08:09:16 up 2 days, 18:44, 4 users, load average: 0.95, 0.44, 0.17
76 processes: 73 sleeping, 3 running, 0 zombie, 0 stopped
CPU states: cpu user nice system irq softirq iowait idle
      total 51.5% 0.0% 3.9% 0.0% 0.0% 0.0% 44.6%
      cpu00 90.0% 0.0% 1.2% 0.0% 0.0% 0.0% 8.8%
      cpu01 13.0% 0.0% 6.6% 0.0% 0.0% 0.0% 80.4%
Mem: 2037140k av, 1132120k used, 905020k free,      0k shrd, 86220k buff
       689784k active,           151528k inactive
Swap: 2040244k av,      0k used, 2040244k free          322648k cached

 PID USER      PRI  NI   SIZE  RSS SHARE STAT %CPU %MEM     TIME CPU COMMAND
 7642 root      25   0 647M 379M 7664 R    49.9 19.0   2:58  0 doom.x86
 7661 ezolt    15   0 1372 1372 1052 R    0.1  0.0   0:00  1 top
  1 root      15   0   528   528   452 S    0.0  0.0   0:05  1 init
  2 root      RT   0   0     0     0 SW    0.0  0.0   0:00  0 migration/0
  3 root      RT   0   0     0     0 SW    0.0  0.0   0:00  1 migration/1
  4 root      15   0   0     0     0 SW    0.0  0.0   0:00  0 keventd
  5 root      34  19   0     0     0 SWN   0.0  0.0   0:00  0 ksoftirqd/0
  6 root      34  19   0     0     0 SWN   0.0  0.0   0:00  1 ksoftirqd/1
  9 root      25   0   0     0     0 SW    0.0  0.0   0:00  0 bdflush
```

7 root	15	0	0	0	0 SW	0.0	0.0	0:00	0	kswapd
8 root	15	0	0	0	0 SW	0.0	0.0	0:00	1	kscand
10 root	15	0	0	0	0 SW	0.0	0.0	0:00	1	kupdated
11 root	25	0	0	0	0 SW	0.0	0.0	0:00	0	mdrecoveryd

现在，在 top 运行时按下 F 键弹出配置界面，如清单 2.7 所示。当你按下代表键（A 代表 PID，B 代表 PPID，等等）时，top 将切换这些统计信息在屏幕上的显示。选择好需要的全部统计信息后，按下 Enter 键返回 top 的初始界面，现在它显示的是选出的统计信息的当前值。在配置统计信息时，所有当前选择的字段将会以大写形式显示在 Current Field Order 行，并在其名称旁出现一个星号 (*)。

清单 2.7

```
[ezolt@wintermute doc]$ top
(press 'F' while running)

Current Field Order: AbcDgHIjklMnoTp|qrsuzyV{EFW[X
Toggle fields with a-z, any other key to return:
* A: PID      = Process Id
  B: PPID     = Parent Process Id
  C: UID      = User Id
* D: USER     = User Name
* E: %CPU     = CPU Usage
* F: %MEM     = Memory Usage
  G: TTY      = Controlling tty
* H: PRI      = Priority
* I: NI       = Nice Value
  J: PAGEIN   = Page Fault Count
  K: TSIZE    = Code Size (kb)
  L: DSIZE    = Data+Stack Size (kb)
* M: SIZE     = Virtual Image Size (kb)
  N: TRS      = Resident Text Size (kb)
  O: SWAP     = Swapped kb
* P: SHARE    = Shared Pages (kb)
  Q: A        = Accessed Page count
  R: WP       = Write Protected Pages
  S: D        = Dirty Pages
* T: RSS      = Resident Set Size (kb)
  U: WCHAN   = Sleeping in Function
* V: STAT     = Process Status
* W: TIME     = CPU Time
* X: COMMAND  = Command
  Y: LC       = Last used CPU (expect this to change regularly)
  Z: FLAGS   = Task Flags (see linux/sched.h)
```

为了展示 top 的可定制性，清单 2.8 给出了一个高度配置的输出界面，其中只显示了与 CPU 使用率相关的 top 选项：

清单 2.8

CPU states: cpu user nice system irq softirq iowait idle								
total	48.2%	0.0%	1.5%	0.0%	0.0%	0.0%	0.0%	50.1%
cpu00	0.3%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	99.5%
cpu01	96.2%	0.0%	2.9%	0.0%	0.0%	0.0%	0.0%	0.7%
Mem:	2037140k av,	1133548k used,	903592k free,		0k shrd,	86232k buff		
690812k active,		151536k inactive						
Swap:	2040244k av,	0k used,	2040244k free			322656k cached		
PID USER PRI NI WCHAN FLAGS LC STAT %CPU TIME CPU COMMAND								
7642 root	25	0		100100	1 R	49.6	10:30	1 doom.x86
1 root	15	0		400100	0 S	0.0	0:05	0 init
2 root	RT	0		140	0 SW	0.0	0:00	0 migration/0
3 root	RT	0		140	1 SW	0.0	0:00	1 migration/1
4 root	15	0		40	0 SW	0.0	0:00	0 keventd
5 root	34	19		40	0 SWN	0.0	0:00	0 ksoftirqd/0
6 root	34	19		40	1 SWN	0.0	0:00	1 ksoftirqd/1
9 root	25	0		40	0 SW	0.0	0:00	0 bdflush
7 root	15	0		840	0 SW	0.0	0:00	0 kswapd
8 root	15	0		40	0 SW	0.0	0:00	0 kscand
10 root	15	0		40	0 SW	0.0	0:00	0 kupdated
11 root	25	0		40	0 SW	0.0	0:00	0 mdrecoveryd
20 root	15	0		400040	0 SW	0.0	0:00	0 katad-1

top 提供了一个系统资源使用率的总览，其重点信息在于各种进程是如何消耗这些资源的。由于其输出格式对用户是友好的，而对工具是不友好的，因此最好是在与系统直接交互时使用。

2.2.3 top (3.x.x 版本)

近来，最新版本中提供的 top 已经有了彻底的改变，其结果就是很多命令行和交互选项发生了变化。虽然基本思路是相似的，但对 top 进行了精简，并添加了几个不同的显示模式。

同样的，top 呈现为一个降序列表，排在最前面的是最占用 CPU 的进程。

2.2.3.1 CPU 性能相关的选项

用如下命令行调用 top：

```
top [-d delay] [-n iter] [-i] [-b]
```

`top` 实际有两种模式的选项：命令行选项和运行时选项。命令行选项决定 `top` 如何显示其信息。表 2-8 给出的命令行选项会影响 `top` 显示的性能统计信息的类型和频率。

表 2-8 `top` 命令行选项

选 项	说 明
<code>-d delay</code>	统计信息更新的时间间隔
<code>-n iterations</code>	退出前迭代的次数。 <code>top</code> 更新统计信息的次数为 <code>iterations</code> 次
<code>-i</code>	是否显示空闲进程
<code>-b</code>	以批处理模式运行。通常， <code>top</code> 只显示单屏信息，超出该屏幕的进程不显示。该选项显示全部进程，如果你要将 <code>top</code> 的输出保存为文件或将输出流水给另一个命令进行处理，那么该项是很有用的

运行 `top` 时，为了调查特定问题，你可能想要对你的观察略作调整。和 `top` 2.x 版本一样，其输出的可定制性很高。表 2-9 给出的选项可以在 `top` 运行期间修改显示的统计信息。

表 2-9 `top` 运行时选项

选 项	说 明
<code>A</code>	进程信息的“另一种”显示方式，其内容为各种系统资源最大的消耗者
<code>I</code>	选择 <code>top</code> 是否用系统中的 CPU 数量除以 CPU 使用率 例如，一个系统中有两个 CPU，如果一个进程占用了这两个 CPU，那么这个选项将在 <code>top</code> 显示 CPU 使用率为 100% 或 200% 之间切换
<code>f</code>	显示配置界面，选择在屏幕上显示哪些统计信息
<code>o</code>	显示配置界面，修改统计信息的显示顺序

表 2-10 给出的选项打开或关闭各种系统级信息的显示。关闭不需要的统计信息有助于在屏幕上显示更多进程。

表 2-10 `top` 运行时输出切换选项

选 项	说 明
<code>1 (数字 1)</code>	切换 CPU 使用率是按独立使用率显示还是按总量显示
<code>1</code>	切换是否更新和显示平均负载和正常运行时间信息

与 `top` v2.x 相同，`top` v3.x 除了提供特定进程的信息之外，还提供系统整体信息。表 2-11 给出了这些统计信息。

表 2-11 `top` 性能统计信息

选 项	说 明
<code>us</code>	用户应用程序消耗的 CPU 时间
<code>sy</code>	内核消耗的 CPU 时间
<code>ni</code>	修改过“友好”值的进程消耗的 CPU 时间
<code>id</code>	空闲的 CPU 时间
<code>wa</code>	等待 I/O 的 CPU 时间
<code>hi</code>	<code>irq</code> 处理程序消耗的 CPU 时间

(续)

选 项	说 明
si	softirq 处理程序消耗的 CPU 时间
load average	1 分钟、5 分钟和 15 分钟的平均负载
%CPU	特定进程消耗 CPU 时间的百分比
PRI	进程优先级，值越大表示优先级越高。RT 代表任务为实时优先级，该优先级高于标准范围
NI	进程的 nice 值。nice 值越高，系统执行该进程的必要性就越低。具有高 nice 值的进程通常其优先级会非常低
WCHAN	若进程在等待 I/O，该项显示其等待的是哪个内核函数
TIME	自进程开始执行起已消耗的总的 CPU 时间（用户和系统）
COMMAND	进程正在执行的命令
S	进程当前的状态。这里，进程可以是睡眠状态（S），运行状态（R），僵尸状态（要求终止但还未终止）（Z），不可中断的睡眠状态（D），或者跟踪状态（T）

top 提供了不同的正在运行进程的大量信息，是找出资源消耗大户的极好方法。top v.3 版对 top 进行了精简，并增加了一些对相同数据的不同视图。

2.2.3.2 用法示例

清单 2.9 是运行 top v3.0 的一个例子。同样的，它会周期性地更新屏幕直到退出。其统计信息与 top v2.x 相同，但名称略有改变。

清单 2.9

```
catan> top
top - 08:52:21 up 19 days, 21:38, 17 users, load average: 1.06, 1.13, 1.15
Tasks: 149 total, 1 running, 146 sleeping, 1 stopped, 1 zombie
Cpu(s): 0.8% us, 0.4% sy, 4.2% ni, 94.2% id, 0.1% wa, 0.0% hi, 0.3% si
Mem: 1034320k total, 1023188k used, 11132k free, 39920k buffers
Swap: 2040244k total, 214496k used, 1825748k free, 335488k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
26364	root	16	0	400m	68m	321m	S	3.8	6.8	379:32.04	X
26737	ezolt	15	0	71288	45m	21m	S	1.9	4.5	6:32.04	gnome-terminal
29114	ezolt	15	0	34000	22m	18m	S	1.9	2.2	27:57.62	gnome-system-mon
9581	ezolt	15	0	2808	1028	1784	R	1.9	0.1	0:00.03	top
1	root	16	0	2396	448	1316	S	0.0	0.0	0:01.68	init
2	root	RT	0	0	0	0	S	0.0	0.0	0:00.68	migration/0
3	root	34	19	0	0	0	S	0.0	0.0	0:00.01	ksoftirqd/0
4	root	RT	0	0	0	0	S	0.0	0.0	0:00.27	migration/1
5	root	34	19	0	0	0	S	0.0	0.0	0:00.01	ksoftirqd/1
6	root	RT	0	0	0	0	S	0.0	0.0	0:22.49	migration/2
7	root	34	19	0	0	0	S	0.0	0.0	0:00.01	ksoftirqd/2
8	root	RT	0	0	0	0	S	0.0	0.0	0:37.53	migration/3

```

9 root      34 19    0 0 0 S 0.0 0.0 0:00.01 ksoftirqd/3
10 root     5 -10   0 0 0 S 0.0 0.0 0:01.74 events/0
11 root     5 -10   0 0 0 S 0.0 0.0 0:02.77 events/1
12 root     5 -10   0 0 0 S 0.0 0.0 0:01.79 events/2

```

现在，在 top 运行时按下 f 键调出配置界面，如清单 2.10 所示。当你按下代表键（A 代表 PID，B 代表 PPID 等）时，top 将切换这些统计信息在屏幕上的显示。选择好需要的全部统计信息后，按下 Enter 键返回 top 的初始界面，现在它显示的是被选出的统计信息的当前值。在配置统计信息时，所有当前被选择的字段将会以大写形式显示在 Current Field Order 行，并在其名称旁出现一个星号 (*)。请注意，大多数统计信息都是相同的，但名称略有变化。

清单 2.10

```

(press 'f' while running)
Current Fields: AEHIOQWTWNMbcdfgjplrsuvyzX for window 1:Def
Toggle fields via field letter, type any other key to return

* A: PID      = Process Id          u: nFLT      = Page Fault count
* E: USER     = User Name          v: nDRT      = Dirty Pages count
* H: PR       = Priority           y: WCHAN    = Sleeping in Function
* I: NI       = Nice value         z: Flags     = Task Flags <sched.h>
* O: VIRT     = Virtual Image (kb) * X: COMMAND = Command name/line
* Q: RES      = Resident size (kb)
* T: SHR      = Shared Mem size (kb) Flags field:
* W: S       = Process Status        0x00000001 PF_ALIGNWARN
* K: %CPU     = CPU usage          0x00000002 PF_STARTING
* N: %MEM     = Memory usage (RES) 0x00000004 PF_EXITING
* M: TIME+    = CPU Time, hundredths 0x00000040 PF_FORKNOEXEC
b: PPID      = Parent Process Pid 0x000000100 PF_SUPERPRIV
c: RUSER     = Real user name      0x000000200 PF_DUMPCORE
d: UID       = User Id            0x000000400 PF_SIGNALLED
f: GROUP     = Group Name          0x000000800 PF_MEMALLOC
g: TTY       = Controlling Tty      0x000002000 PF_FREE_PAGES (2.5)
j: #C       = Last used cpu (SMP) 0x000008000 debug flag (2.5)
p: SWAP      = Swapped size (kb)   0x00024000 special threads (2.5)
l: TIME      = CPU Time            0x001D0000 special states (2.5)
r: CODE      = Code size (kb)       0x00100000 PF_USEDFPU (thru 2.4)
s: DATA      = Data+Stack size (kb)
```

清单 2.11 展示了新的 top 输出模式，许多不同的统计信息进行了分类并显示在同一屏幕上。

清单 2.11

```
(press 'F' while running)
1:Def - 09:00:48 up 19 days, 21:46, 17 users, load average: 1.01, 1.06, 1.10
Tasks: 144 total, 1 running, 141 sleeping, 1 stopped, 1 zombie
Cpu(s): 1.2% us, 0.9% sy, 0.0% ni, 97.9% id, 0.0% wa, 0.0% hi, 0.0% si
Mem: 1034320k total, 1024020k used, 10300k free, 39408k buffers
Swap: 2040244k total, 214496k used, 1825748k free, 335764k cached

1 PID USER      PR NI VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
29114 ezolt     16  0 34112 22m 18m S 3.6 2.2 28:15.06 gnome-system-mo
26364 root      15  0 400m 68m 321m S 2.6 6.8 380:01.09 X
9689 ezolt     16  0 3104 1092 1784 R 1.0 0.1 0:00.09 top
2 PID PPID      TIME+ %CPU %MEM PR NI S VIRT SWAP RES UID COMMAND
30403 24989 0:00.03 0.0 0.1 15 0 S 5808 4356 1452 9336 bash
29510 29505 7:19.59 0.0 5.9 16 0 S 125m 65m 59m 9336 firefox-bin
29505 29488 0:00.00 0.0 0.1 16 0 S 5652 4576 1076 9336 run-mozilla.sh
3 PID %MEM VIRT SWAP RES CODE DATA SHR nFLT nDRT S PR NI %CPU COMMAND
8414 25.0 374m 121m 252m 496 373m 98m 1547 0 S 16 0 0.0 soffice.bin
26364 6.8 400m 331m 68m 1696 398m 321m 2399 0 S 15 0 2.6 X
29510 5.9 125m 65m 59m 64 125m 31m 253 0 S 16 0 0.0 firefox-bin
26429 4.7 59760 10m 47m 404 57m 12m 1247 0 S 15 0 0.0 metacity
4 PID PPID UID USER      RUSER      TTY      TIME+ %CPU %MEM S COMMAND
1371   1  43 xfs      xfs      ? 0:00.10 0.0 0.1 S xfs
1313   1  51 smmsp    smmsp    ? 0:00.08 0.0 0.2 S sendmail
982    1  29 rpcuser  rpcuser  ? 0:00.07 0.0 0.1 S rpc.statd
963    1  32 rpc      rpc      ? 0:06.23 0.0 0.1 S portmap
```

top v3.x 为 top 提供了稍简洁的界面。它简化了 top 的某些方面，并提供了一个很好的“总结”信息屏，显示了系统中的许多资源消费者。

2.2.4 procinfo (从 /proc 文件系统显示信息)

就像 vmstat 一样，procinfo 也为系统整体信息特性提供总览。尽管它提供的有些信息与 vmstat 相同，但它还会给出 CPU 从每个设备接收的中断数量。其输出格式的易读性比 vmstat 稍微强一点，但却会占用更多的屏幕空间。

2.2.4.1 CPU 性能相关的选项

procinfo 的调用命令行如下：

```
procinfo [-f] [-d] [-D] [-n sec] [-f file]
```

表 2-12 描述了不同的选项，用于修改 procinfo 显示样本的输出和频率。

表 2-12 procinfo 命令行选项

选 项	说 明
-f	全屏运行 procinfo
-d	显示样本统计信息的变化，而非总和
-D	显示统计信息的总和，而非变化率
-n sec	样本之间停顿的秒数
-Ffile	将 procinfo 的输出发送到文件

表 2-13 给出了 procinfo 收集的 CPU 统计信息。

表 2-13 procinfo CPU 统计信息

选 项	说 明
user	CPU 花费的总的用户时间，形式为天、小时和分钟
nice	CPU 花费的总的友好时间，形式为天、小时和分钟
system	CPU 花费的总的系统时间，形式为天、小时和分钟
idle	CPU 花费的总的空闲时间，形式为天、小时和分钟
irq 0-N	显示 irq 的编号，已经启动的次数，以及哪个内核驱动程序应对其负责

与 vmstat 以及 top 一样，procinfo 是一个低开销的命令，适合于让其自行在控制台或屏幕窗口运行。它能够很好地反映系统的健康和性能。

2.2.4.2 用法示例

调用 procinfo 时不带任何命令选项将产生如清单 2.12 所示的输出。无参数，则 procinfo 仅显示一屏状态信息并退出。使用 -n second 选项让 procinfo 周期性地更新，其作用会更大。这能使你查看到系统性能的实时变化。

清单 2.12

```
[ezolt@scruffy ~]# procinfo
Linux 2.4.18-3bigmem (bhcompile@daffy) (gcc 2.96 20000731 ) #1 4CPU [scruffy]

Memory:      Total        Used        Free        Shared       Buffers       Cached
Mem:        1030784     987776     43008          0      35996     517504
Swap:        2040244      17480    2022764

Bootup: Thu Jun  3 09:20:22 2004   Load average: 0.47 0.32 0.26 1/118 10378

user : 3:18:53.99 2.7% page in : 1994292 disk 1:      20r      0w
nice : 0:00:22.91 0.0% page out: 2437543 disk 2: 247231r 131696w
system: 3:45:41.20 3.1% swap in :      996
idle : 4d 15:56:17.10 94.0% swap out:     4374
uptime: 1d 5:45:18.80           context : 64608366

irq 0: 10711880 timer           irq 12: 1319185 PS/2 Mouse
```

```

irq 1:    94931 keyboard
irq 2:      0 cascade [4]
irq 3:      1
irq 4:      1
irq 6:      2
irq 7:      1
irq 8:    1 rtc

irq 14:   7144432 ide0
irq 16:    16 aic7xxx
irq 18:   4152504 nvidia
irq 19:      0 usb-uhci
irq 20:   4772275 es1371
irq 22:   384919 aic7xxx
irq 23:   3797246 usb-uhci, eth0

```

如同你在清单 2.12 中所见，procinfo 为系统提供了不错的总览。从用户、nice、系统和空闲时间，我们再次发现，系统不是很忙。一个值得注意的有趣现象是，procinfo 表明系统空闲时间比其运行时间（用 uptime 表示）还要多。这是因为系统实际上有 4 个 CPU，因此，对于一天的墙钟时间而言，CPU 时间已经过去了四天。平均负载证明系统近期相对没有多少工作。在过去的时间里，平均而言，系统准备运行的进程还不到一个；平均负载为 0.47 意味着单个进程准备运行的时间只有 47%。对于有四个 CPU 的系统来说，将会浪费大量的 CPU 能力。

procinfo 还给我们提供了很好的视图来说明系统中的哪个设备导致了中断。可以看到显卡（nvidia）、硬盘控制器（ide0）、以太网设备（eth0）以及声卡（es1371）的中断数量相对较高。这些情况一般出现在台式工作站上。

procinfo 的优势是将许多系统级性能统计信息放在一个屏幕里，让你能了解系统整体执行情况。它缺乏网络和磁盘性能的详细信息，但能为 CPU 和内存性能的统计信息提供良好的细节。一个可能很重要的限制是，CPU 处于 iowait、irq 或 softirq 模式时 procinfo 不会进行报告。

2.2.5 gnome-system-monitor

gnome-system-monitor 在很多方面都可以说是 top 的图形化。它使你能以图形方式监控各个进程，并在显示图表的基础上观察系统负载。

2.2.5.1 CPU 性能相关的选项

gnome-system-monitor 可以从 Gnome 菜单调用。（Red Hat 9 及其以上版本中，选择菜单 System Tools → System Monitor。）不过，它也可以用如下命令调用：

```
gnome-system-monitor
```

gnome-system-monitor 没有相关命令行选项来影响 CPU 性能测量。但是，有些显示的统计信息可以通过选择 gnome-system-monitor 的 Edit → Preferences 菜单项进行修改。

2.2.5.2 用法示例

当你启动 gnome-system-monitor 时，它会创建与图 2-1 相似的窗口。该窗口显示了特

定进程使用的 CPU 和内存总量信息。它还显示了进程之间父 / 子关系的信息。

图 2-2 显示了系统负载和内存使用率的图形视图。从这一点可以真正区分 gnome-system-monitor 与 top。你可以很容易地查看系统当前状态，以及与之前状态的对比。

gnome-system-monitor 提供的数据图形视图能够更容易更迅速地确定系统状态及其行为随时间的变化。它还能更轻松地浏览系统级进程信息。

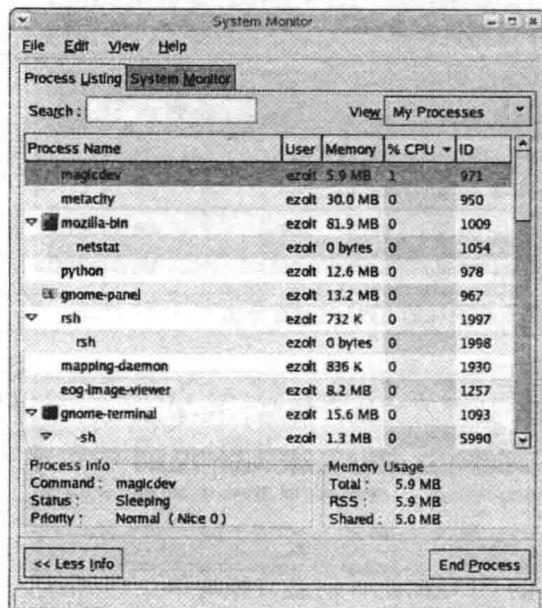


图 2-1

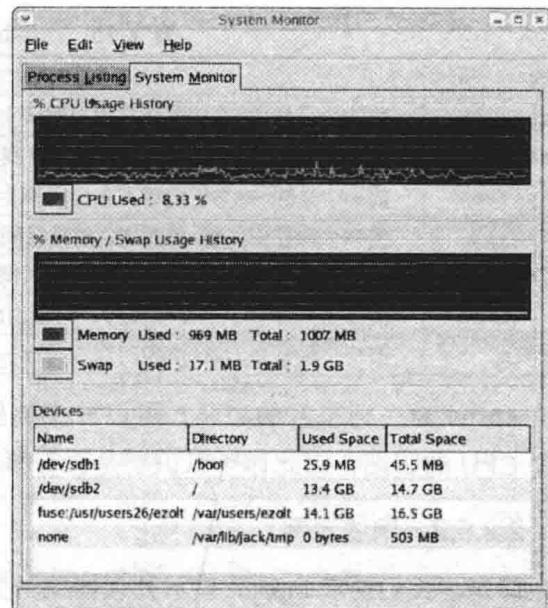


图 2-2

2.2.6 mpstat (多处理器统计)

mpstat 是一个相当简单的命令，向你展示随着时间变化的 CPU 行为。mpstat 最大的优点是在统计信息的旁边显示时间，由此，你可以找出 CPU 使用率与时间的关系。

如果你有多个 CPU 或超线程 CPU，mpstat 还能够把 CPU 使用率按处理器进行区分，因此你可以发现与其他处理器相比，是否某个处理器做了更多的工作。你可以选择想要监控的单个处理器，也可以要求 mpstat 对所有的处理器都进行监控。

2.2.6.1 CPU 性能相关的选项

mpstat 可以用如下命令行调用：

```
mpstat [ -P { cpu | ALL } ] [ delay [ count ] ]
```

和之前一样，delay 指定了采样间隔，count 指定了采样次数。表 2-14 解释了 mpstat 命令行选项的含义。

表 2-14 mpstat 命令行选项

选 项	说 明
-P {cpu ALL}	告诉 mpstat 监控哪个 CPU, cpu 取值范围为 0~(CPU 总数 -1)
delay	指明 mpstat 在采样之间应等待的时长

mpstat 提供与其他 CPU 性能工具相似的信息，但是，它允许将信息按照特定系统中的单个处理器进行分类。表 2-15 给出了 mpstat 支持的选项。

表 2-15 mpstat CPU 统计信息

选 项	说 明
user	前一个采样中 CPU 消耗在用户时间上的百分比
nice	前一个采样中 CPU 执行低优先级 (或 nice) 进程消耗时间的百分比
system	前一个采样中 CPU 消耗在系统时间上的百分比
iowait	前一个采样中 CPU 等待 I/O 消耗时间的百分比
irq	前一个采样中 CPU 处理中断消耗时间的百分比
softirq	前一个采样中，中断处理后，内核完成所需工作消耗的 CPU 时间百分比
idle	前一个采样中 CPU 空闲时间百分比

mpstat 是一种很好的工具，可以分类提供每个处理器的执行情况。由于 mpstat 给出了每个 CPU 的明细，因此你可以识别是否有哪个处理器正逐渐出现超负载情况。

2.2.6.2 用法示例

首先，我们要求 mpstat 显示处理器编号为 0 的 CPU 的统计信息，如清单 2.13 所示。

清单 2.13

```
[ezolt@scrffy sysstat-5.1.1]$ ./mpstat -P 0 1 10
Linux 2.6.8-1.521smp (scrffy)   10/20/2004
```

07:12:02 PM	CPU	%user	%nice	%sys	%iowait	%irq	%soft	%idle	intr/s
07:12:03 PM	0	9.80	0.00	1.96	0.98	0.00	0.00	87.25	1217.65
07:12:04 PM	0	1.01	0.00	0.00	0.00	0.00	0.00	98.99	1112.12
07:12:05 PM	0	0.99	0.00	0.00	0.00	0.00	0.00	99.01	1055.45
07:12:06 PM	0	0.00	0.00	0.00	0.00	0.00	0.00	100.00	1072.00
07:12:07 PM	0	0.00	0.00	0.00	0.00	0.00	0.00	100.00	1075.76
07:12:08 PM	0	1.00	0.00	0.00	0.00	0.00	0.00	99.00	1067.00
07:12:09 PM	0	4.90	0.00	3.92	0.00	0.00	0.98	90.20	1045.10
07:12:10 PM	0	0.00	0.00	0.00	0.00	0.00	0.00	100.00	1069.70
07:12:11 PM	0	0.99	0.00	0.99	0.00	0.00	0.00	98.02	1070.30
07:12:12 PM	0	3.00	0.00	4.00	0.00	0.00	0.00	93.00	1067.00
Average:	0	2.19	0.00	1.10	0.10	0.00	0.10	96.51	1085.34

清单 2.14 显示了对典型无负载超线程 CPU 使用相同命令产生的结果。你可以看到所有显示出来的 CPU 统计数据。输出中有个有趣的现象，即其中一个 CPU 似乎处理了所有的

中断。如果系统有很重的 I/O 负载，而全部中断又都是由一个处理器处理，那么这可能就是瓶颈，因为一个 CPU 超负荷，而其他 CPU 则在等待。如果一个 CPU 忙于处理所有的中断以至于没有空闲时间，而与此同时，其他处理器则处于空闲状态，那么你可以用 mpstat 发现这种情况。

清单 2.14

```
[ezolt@scrffy sysstat-5.1.1]$ ./mpstat -P ALL 1 2
Linux 2.6.8-1.521smp (scrffy)   10/20/2004

07:13:21 PM CPU %user %nice   %sys %iowait   %irq   %soft   %idle intr/s
07:13:22 PM all  3.98  0.00   1.00  0.25    0.00    0.00  94.78 1322.00
07:13:22 PM    0  2.00  0.00   0.00  1.00    0.00    0.00  97.00 1137.00
07:13:22 PM    1  6.00  0.00   2.00  0.00    0.00    0.00  93.00 185.00
07:13:22 PM    2  1.00  0.00   0.00  0.00    0.00    0.00  99.00  0.00
07:13:22 PM    3  8.00  0.00   1.00  0.00    0.00    0.00  91.00  0.00

07:13:22 PM CPU %user %nice   %sys %iowait   %irq   %soft   %idle intr/s
07:13:23 PM all  2.00  0.00   0.50  0.00    0.00    0.00  97.50 1352.53
07:13:23 PM    0  0.00  0.00   0.00  0.00    0.00    0.00 100.00 1135.35
07:13:23 PM    1  6.06  0.00   2.02  0.00    0.00    0.00  92.93 193.94
07:13:23 PM    2  0.00  0.00   0.00  0.00    0.00    0.00 101.01 16.16
07:13:23 PM    3  1.01  0.00   1.01  0.00    0.00    0.00 100.00  7.07

Average: CPU %user %nice   %sys %iowait   %irq   %soft   %idle intr/s
Average: all  2.99  0.00   0.75  0.12    0.00    0.00  96.13 1337.19
Average:    0  1.01  0.00   0.00  0.50    0.00    0.00  98.49 1136.18
Average:    1  6.03  0.00   2.01  0.00    0.00    0.00  92.96 189.45
Average:    2  0.50  0.00   0.00  0.00    0.00    0.00 100.00  8.04
Average:    3  4.52  0.00   1.01  0.00    0.00    0.00  95.48  3.52
```

mpstat 可以用来确定 CPU 是否得到充分利用，以及使用情况是否相对均衡。通过观察每个 CPU 处理的中断数，有可能发现其中的不均衡。如何控制中断路由的详细信息参见 Documentation/IRQ-affinity.txt 下的内核源码。

2.2.7 sar (系统活动报告)

sar 用另一种方法来收集系统数据。sar 能有效地将收集到的系统性能数据记录到二进制文件，之后，可以重播这些文件。sar 是一种低开销的、记录系统执行情况信息的方法。

sar 命令可以用于记录性能信息，回放之前的记录信息，以及显示当前系统的实时信息。sar 命令的输出可以进行格式化，使之易于导入数据库，或是输送给其他 Linux 命令进行处理。

2.2.7.1 CPU 性能相关的选项

sar 可以使用如下命令行调用：

```
sar [options] [ delay [ count ] ]
```

尽管 sar 的报告涉及 Linux 多个不同领域，其统计数据有两种不同的形式。一组统计数据是采样时的瞬时值。另一组则是自上一次采样后的变化值。表 2-16 解释了 sar 的命令行选项。

表 2-16 sar 命令行选项

选 项	说 明
-c	报告每秒创建的进程数量
-I {irq SUM ALL XALL }	报告系统已发生中断的速率
-P { cpu ALL }	该项确定从哪个 CPU 收集统计信息。如果不指定，则报告系统整体情况
-q	报告机器的运行队列长度和平均负载
-u	报告系统的 CPU 使用情况（该项为默认输出）
-w	报告系统中已发生的上下文切换次数
-o filename	指定保存性能统计信息的二进制输出文件名
-f filename	指定性能统计信息的文件名
delay	需等待的采样间隔时间
count	记录的样本总数

sar 提供的系统级 CPU 性能统计数据集与我们在进程工具中看到的类似（名字不同）。如表 2-17 所示。

表 2-17 sar CPU 统计信息

选 项	说 明
user	前一个采样中 CPU 消耗在用户时间上的百分比
nice	前一个采样中 CPU 执行低优先级（或 nice）进程消耗时间的百分比
system	前一个采样中 CPU 消耗在系统时间上的百分比
iowait	前一个采样中 CPU 等待 I/O 消耗时间的百分比
idle	前一个采样中 CPU 空闲时间百分比
rung-sz	采样时，运行队列的长度
plist-sz	采样时的进程（运行，睡眠或等待 I/O）数
1davg-1	前 1 分钟的平均负载
1davg-5	前 5 分钟的平均负载
1davg-15	前 15 分钟的平均负载
proc/s	每秒新建进程数（该项等同于 vmstat 中的 forks 项）
cswch	每秒上下文切换次数
intr/s	每秒触发的中断次数

sar 最显著的优势之一是，它使你能把不同类型时间戳系统数据保存到日志文件，以便日后检索和审查。当试图找出特定机器在特定时间出现故障的原因时，这个特性被证明是非常便利的。

2.2.7.2 用法示例

清单 2.15 显示的第一个命令要求每秒有三个 CPU 采样，其结果保存到二进制文件 /tmp/apache_test。该命令没有任何可视化输出，完成即返回。

清单 2.15

```
[ezolt@wintermute sysstat-5.0.2]$ sar -o /tmp/apache_test 1 3
```

信息保存到 /tmp/apache_test 文件后，我们就能以各种格式显示它。默认格式为人类可读，如清单 2.16 所示。该清单显示与其他系统监控命令类似的信息，我们可以看出处理器在特定时间是如何消耗其时间的。

清单 2.16

```
[ezolt@wintermute sysstat-5.0.2]$ sar -f /tmp/apache_test
Linux 2.4.22-1.2149.nptl (wintermute.phil.org) 03/20/04
```

	CPU	%user	%nice	%system	%iowait	%idle
17:18:34						
17:18:35	all	90.00	0.00	10.00	0.00	0.00
17:18:36	all	95.00	0.00	5.00	0.00	0.00
17:18:37	all	92.00	0.00	6.00	0.00	2.00
Average:	all	92.33	0.00	7.00	0.00	0.67

不过，sar 还可以将统计数据输出为一种能轻松导入关系数据库的格式，如清单 2.17 所示。这有助于保存大量的性能数据。一旦将其导入到关系数据库，就可以用所有的关系数据库工具对这些性能数据进行分析。

清单 2.17

```
[ezolt@wintermute sysstat-5.0.2]$ sar -f /tmp/apache_test -H
wintermute.phil.org;1;2004-03-20 22:18:35 UTC;-1;90.00;0.00;10.00;0.00;0.00
wintermute.phil.org;1;2004-03-20 22:18:36 UTC;-1;95.00;0.00;5.00;0.00;0.00
wintermute.phil.org;1;2004-03-20 22:18:37 UTC;-1;92.00;0.00;6.00;0.00;2.00
```

最后，sar 还有一种易于被标准 Linux 工具，如 awk，perl，python 或 grep，解析的统计数据输出格式。如清单 2.18 所示，这种输出可以被送入脚本，该脚本会引发有趣的事件，甚至有可能分析出数据的不同趋势。

清单 2.18

```
[ezolt@wintermute sysstat-5.0.2]$ sar -f /tmp/apache_test -h
wintermute.phil.org    1      1079821115      all      %user    90.00
wintermute.phil.org    1      1079821115      all      %nice    0.00
wintermute.phil.org    1      1079821115      all      %system  10.00
wintermute.phil.org    1      1079821115      all      %iowait   0.00
```

wintermute.phil.org	1	1079821115	all	%idle	0.00
wintermute.phil.org	1	1079821116	all	%user	95.00
wintermute.phil.org	1	1079821116	all	%nice	0.00
wintermute.phil.org	1	1079821116	all	%system	5.00
wintermute.phil.org	1	1079821116	all	%iowait	0.00
wintermute.phil.org	1	1079821116	all	%idle	0.00
wintermute.phil.org	1	1079821117	all	%user	92.00
wintermute.phil.org	1	1079821117	all	%nice	0.00
wintermute.phil.org	1	1079821117	all	%system	6.00
wintermute.phil.org	1	1079821117	all	%iowait	0.00
wintermute.phil.org	1	1079821117	all	%idle	2.00

除了将信息记录到文件之外，sar 还可以用于实时系统观察。在清单 2.19 所示的例子中，CPU 状态被采样了三次，采样间隔时间为一秒。

清单 2.19

```
[ezolt@wintermute sysstat-5.0.2]$ sar 1 3
Linux 2.4.22-1.2149.nptl (wintermute.phil.org) 03/20/04
```

	CPU	%user	%nice	%system	%iowait	%idle
17:27:10						
17:27:11	all	96.00	0.00	4.00	0.00	0.00
17:27:12	all	98.00	0.00	2.00	0.00	0.00
17:27:13	all	92.00	0.00	8.00	0.00	0.00
Average:	all	95.33	0.00	4.67	0.00	0.00

默认显示的目的是展示 CPU 的信息，但是也可以显示其他信息。比如，sar 可以显示每秒的上下文切换次数，以及交换的内存页面数。在清单 2.20 中，sar 采样了两次信息，间隔时间为一秒。这次，我们要求 sar 显示每秒上下文切换的数量以及创建的进程数。我们还要求 sar 给出平均负载的信息。可以看出来，本例中的机器有 163 个进程在内存中，但都没有运行。过去的一分钟平均有 1.12 个进程等待运行。

清单 2.20

```
[ezolt@scruffy manuscript]$ sar -w -c -q 1 2
Linux 2.6.8-1.521smp (scruffy) 10/20/2004
```

```
08:23:29 PM    proc/s
08:23:30 PM      0.00
```

```
08:23:29 PM    cswch/s
08:23:30 PM    594.00
```

```
08:23:29 PM    runq-sz  plist-sz  ldavg-1  ldavg-5  ldavg-15
```

```

08:23:30 PM      0      163     1.12     1.17     1.17

08:23:30 PM    proc/s
08:23:31 PM    0.00

08:23:30 PM   cswch/s
08:23:31 PM    812.87

08:23:30 PM  runq-sz  plist-sz  ldavg-1  ldavg-5  ldavg-15
08:23:31 PM      0       163     1.12     1.17     1.17

Average:      proc/s
Average:      0.00

Average:      cswch/s
Average:      703.98

Average:  runq-sz  plist-sz  ldavg-1  ldavg-5  ldavg-15
Average:      0       163     1.12     1.17     1.17

```

如你所见，sar 是一个强大的工具，能够记录多种不同的性能统计信息。它提供了 Linux 友好界面，使你可以轻松地提取和分析性能数据。

2.2.8 oprofile

oprofile 是性能工具包，它利用几乎所有现代处理器都有的性能计数器来跟踪系统整体以及单个进程中 CPU 时间的消耗情况。除了测量 CPU 周期消耗在哪里之外，oprofile 还可以测量关于 CPU 执行的非常底层的信息。根据由底层处理器支持的事件，它可以测量的内容包括：cache 缺失、分支预测错误和内存引用，以及浮点操作。

oprofile 不会记录发生的每个事件，相反，它与处理器性能硬件一起工作，每 count 个事件采样一次，这里的 count 是一个数值，由用户在启动 oprofile 时指定。count 的值越低，结果的准确度越高，而 oprofile 的开销越大。若 count 保持在一个合理的数值，那么，oprofile 不仅运行开销非常低，并且还能以令人惊讶的准确性描述系统性能。

采样是非常强大的，但使用时要小心一些不明显的陷阱。首先，采样可能会显示你有 90% 的时间花在了一个特定的例程上，但它不会显示原因。一个特定例程消耗了大量周期有两种可能的原因。其一，该例程可能是瓶颈，其执行需要很多时间。但是，也可能例程的执行时间是合理的，而其被调用的次数非常高。通常有两种途径可以发现究竟是哪一种情况：通过查看采样找出特别热门的行，或是通过编写代码来计算例程被调用次数。

采样的第二个问题是永远无法十分确定一个函数是从哪里被调用的。即使你已经搞明白它被调用了很多次，并且已经跟踪到了所有调用它的函数，但也不一定清楚其中哪个函数完成了绝大多数的调用。

2.2.8.1 CPU 性能相关的选项

`oprofile` 实际上是一组协同工作的组件，用于收集 CPU 性能统计信息。`oprofile` 主要有三个部分：

- `oprofile` 核心模块控制处理器并允许和禁止采样。
- `oprofile` 后台模块收集采样，并将它们保存到磁盘。
- `oprofile` 报告工具获取收集的采样，并向用户展示它们与在系统上运行的应用程序的关系。

`oprofile` 工具包将驱动器和后台操作隐藏在 `opcontrol` 命令中。`opcontrol` 命令用于选择处理器采样的事件并启动采样。

进行后台控制时，你可以使用如下命令行调用 `opcontrol`：

```
opcontrol [--start] [--stop] [--dump]
```

此选项的控制（性能分析后台进程）使你能开始和停止采样，并将样本从守护进程的内存导入磁盘。采样时，`oprofile` 后台模块将大量的采样保存在内部缓冲区。但是，它只能分析那些已经写入（或导入）磁盘的样本。写磁盘的开销可能会很大，因此，`oprofile` 只会定期执行这个操作。其结果就是，运行测试并用 `oprofile` 分析后，可能不会马上得到结果，你需要等待，直到后台将缓冲区写入磁盘。当你想要立即开始分析时，这点是很让人挠头的，因此，`opcontrol` 命令能让你禁止将采样从 `oprofile` 后台的内部缓冲区导入到磁盘。这将使你能在测试结束后，立刻开始性能调查。

表 2-18 介绍了 `opcontrol` 程序的选项，它们使你能控制后台操作。

表 2-18 opcontrol 后台控制

选 项	说 明
<code>-s / -- start</code>	若当前处理器未使用默认事件，则开始分析
<code>-d / -- dump</code>	将核心采样缓冲区当前采样信息导入磁盘
<code>-- stop</code>	停止分析

默认情况下，`oprofile` 按给定频率选择一个事件，这个频率对于你在运行的处理器和内核来说是合理的。但是，比起默认事件来，还有更多的事件可以监控。当你列出并选择了一个事件后，`opcontrol` 将用如下命令行调用：

```
opcontrol [--list-events] [-event=:name:count:unitmask:kernel:user:]
```

事件说明使你可以选择采样哪个事件，该事件的采样频率，以及采样发生在内核空间、用户空间或同时在这两个空间。表 2-19 介绍了 `opcontrol` 的命令行选项，它们使你能选择不

同的事件进行采样。

表 2-19 opcontrol 事件处理

选 项	说 明
-l / --list-events	列出处理器可以采样的不同事件
-event=:name:count: unitmask:kernel:user:	用于指定被采样的事件。事件名必须为处理器支持的事件之一。可用事件可以从 --list-events 选项获取。参数 count 定义事件每发生 count 次，处理器将采样一次。 unitmask 修改将要被采样的事件。比如，如果你采样“从内存读”，那么单元屏蔽（unit mask）可以让你只选择那些 cache 未命中的读操作。参数 kernel 指明，当处理器运行于内核空间时 oprofile 是否采样。参数 user 指明，当处理器运行于用户空间时 oprofile 是否采样
-- vmlinux=kernel	说明 oprofile 对不同的内核函数采样将使用哪个非压缩内核映像

收集并保存样本后，oprofile 提供另一种不同的工具 oreport，该工具使你能查看已收集的样本。opreport 的调用命令行如下：

```
opreport [-r] [-t]
```

通常，opreport 显示所有系统收集到的样本，以及哪些可执行程序引起的这些样本（包括内核）。样本数最多的可执行线程排在第一位，其后为所有有样本的可执行线程。在一个典型系统中，排在列表前面的是拥有大多数样本的少数可执行线程，而大量的可执行线程只贡献了数量很少的样本。针对这种情况，opreport 允许你设置阈值，只有样本数量百分比达到或超过阈值的可执行线程才能显示。同时，opreport 还可以将可执行线程的显示顺序倒过来，那些拥有最多样本数的将最后显示。这种方式下，最重要的数据显示在最后，那么它就不会滚过屏幕。

表 2-20 说明了 opreport 的命令行选项，它们使你能定制采样输出的格式。

表 2-20 opreport 报告格式

选 项	说 明
-- reverse-sort / -r	颠倒显示顺序。通常，引起最多事件的映像最先显示
-- threshold / -t [percentage]	使 opreport 只显示样本比例达到 percentage 或更高的映像。该项适用条件为：很多映像的样本数非常小，而你只对样本数量最多的感兴趣

再次说明，oprofile 是一个复杂的工具，给出的这些选项仅仅是 oprofile 的基础功能。在后续章节中，你将学习到 oprofile 更多的功能。

2.2.8.2 用法示例

oprofile 是非常强大的工具，但它的安装有点困难。附录 B 指导读者如何在几个主要的 Linux 发行版上安装和运行 oprofile。

使用 oprofile 首先要按照分析对其进行设置。第一条命令如清单 2.21 所示，用 opcontrol 命令告诉 oprofile 工具包一个非压缩的内核映像在什么位置。oprofile 需要知道这个文件的位置，以便它将样本分配给内核中的确切函数。

清单 2.21

```
[root@wintermute root]# opcontrol --vmlinuz=/boot/vmlinux-\n2.4.22-1.2174.nptlsmp
```

设置了当前内核的路径后，我们可以开始分析。清单 2.22 中的命令告诉oprofile用默认事件开始采样。这个事件根据处理器而变化，对当前处理器而言，这个事件是CPU_CLK_UNHALTED。只要处理器没有停止，该事件将会采样全部CPU周期。233869是指每233 869个事件会采样处理器正在执行的指令。

清单 2.22

```
[root@wintermute root]# opcontrol -s\nUsing default event: CPU_CLK_UNHALTED:233869:0:1:1\nUsing log file /var/lib/oprofile/oprofiled.log\nDaemon started.\nProfiler running.
```

现在已经开始采样后，我们想要分析采样结果。在清单 2.23 中，我们用报告工具来找出系统中发生了什么。opreport 报告了目前为止分析的内容。

清单 2.23

```
[root@wintermute root]# opreport\nopreport op_fatal_error:\nNo sample file found: try running opcontrol --dump\nor specify a session containing sample files
```

尽管分析已经进行了一小段时间，但当opreport表明它无法找到样本时，我们就会停止。发生这种情况的原因是：opreport命令在磁盘上查找样本，而oprofile后台程序则在内存中存储样本并定期将其转存到磁盘。当我们向opreport请求样本清单时，它无法在磁盘上找到，因此就会报告没有发现任何样本。为了缓解这一问题，我们可以通过在opcontrol中增加dump选项来强制后台程序立刻转存样本，如清单 2.24 所示，这条命令使我们能查看已收集的样本。

清单 2.24

```
[root@wintermute root]# opcontrol --dump
```

将样本转存到磁盘后，我们再次尝试要求oprofile给出报告，如清单 2.25 所示。这一次，我们得到了结果。报告中包含了收集样本来源处理器的信息，以及其监控事件的类型信息。然后，报告按降序排列事件发生的数量，并列出它们发生在哪个可执行文件中。我们可以看到，Linux 内核占据了全部时钟的 50%，emacs 为 14%，libc 为 12%。可以深入挖掘可执行文件确定哪个函数占据了所有的时间，我们将在第 4 章讨论这个问题。

清单 2.25

```
[root@wintermute root]# opreport
CPU: PIII, speed 467.739 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted)
with a unit mask of 0x00 (No unit mask) count 233869
 3190 50.4507 vmlinuz-2.4.22-1.2174.nptlsmp
  905 14.3128 emacs
  749 11.8456 libc-2.3.2.so
  261  4.1278 ld-2.3.2.so
  244  3.8589 mpg321
  233  3.6850 insmod
  171  2.7044 libperl.so
  128  2.0244 bash
  113  1.7871 ext3.o
...

```

当我们启动 oprofile 时，我们只使用了 opcontrol 为我们选择的默认事件。每个处理器都有一个非常丰富的可以被监控的事件集。在清单 2.26 中，我们要求 opcontrol 列出特定 CPU 可以获得的全部事件。这个清单相当长，但在其中我们可以看到除了 CPU_CLK_UNHALTED 之外，还可以监控 DATA_MEM_REFS 和 DCU_LINES_IN。这些是内存子系统导致的存储事件，我们将在后续章节中讨论它们。

清单 2.26

```
[root@wintermute root]# opcontrol -l
oprofile: available events for CPU type "PIII"

See Intel Architecture Developer's Manual Volume 3, Appendix A and
Intel Architecture Optimization Reference Manual (730795-001)

CPU_CLK_UNHALTED: (counter: 0, 1)
    clocks processor is not halted (min count: 6000)
DATA_MEM_REFS: (counter: 0, 1)
    all memory references, cachable and non (min count: 500)
DCU_LINES_IN: (counter: 0, 1)
    total lines allocated in the DCU (min count: 500)
...
```

需要指明被监控事件的命令看上去有点麻烦，幸运的是，我们还可以利用 oprofile 的图形化命令 oprof_start 以图形方式启动和停止采样。这使得我们能以图形方式选择想要的事件，而没有必要搞清楚用准确的方式在命令行中明确说明想要监控的事件。

在图 2-3 所示的 op_control 例子中，我们告诉 oprofile 想要同时监控 DATA_MEM_

REFS 和 L2_LD 事件。DATA_MEM_REFs 事件可以告诉我们哪些应用程序使用了大量的内存子系统，哪些使用了 L2 cache。具体到这个处理器，其硬件只有两个计数器可用于采样，因此能同时使用的也只有两个事件。

用 oprofile 的图形界面收集样本后，我们现在可以分析这些数据了。如清单 2.27 所示，我们要求 opreport 显示对其收集样本的分析，所用形式与监控周期时的类似。在本例中，我们可以发现 libmad 库占用了整个系统中数据内存访问的 31%，成为内存子系统使用量最大的用户。

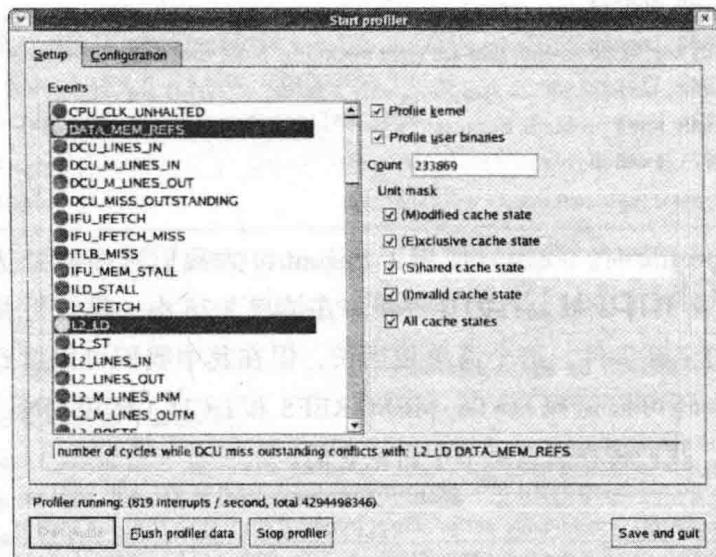


图 2-3

清单 2.27

```
[root@wintermute root]# opreport
CPU: PIII, speed 467.739 MHz (estimated)
Counted DATA_MEM_REFs events (all memory references, cachable and non)
with a unit mask of 0x00 (No unit mask) count 30000
Counted L2_LD events (number of L2 data loads) with a unit mask of 0x0f
(All cache states) count 233869
      87462 31.7907      17  3.8636 libmad.so.0.1.0
     24259  8.8177      10  2.2727 mpg321
     23735  8.6272      40  9.0909 libz.so.1.2.0.7
     17513  6.3656      56 12.7273 libgklayout.so
     17128  6.2257      90 20.4545 vmlinux-2.4.22-1.2174.nptlsmp
     13471  4.8964       4  0.9091 libpng12.so.0.1.2.2
     12868  4.6773      20  4.5455 libc-2.3.2.so
    ...

```

opreport 提供的输出展示了包含任何被采样事件的所有系统库和可执行程序。请注意并非所有的事件都被记录下来，这是因为我们是在采样，实际上只会记录事件的子集。通常这不是问题，因为如果一个特定的库或可执行程序是性能问题，那么它很可能会导致高成本事件发生许多次。如果采样是随机的，这些高成本使事件最终也会被采样代码所捕获。

2.3 本章小结

本章重点是关于 CPU 使用情况的系统级性能指标。这些指标主要展示的是操作系统和机器是如何运行的，而不是某个特定的应用程序。

本章展示了如何使用性能工具，如 sar 和 vmstat，从正在运行的系统中抽取系统级性能信息。这些工具是诊断系统问题时的第一道防线。它们帮助确定系统表现如何，以及哪个子系统或应用程序可能压力较大。下一章将关注能分析系统整体内存使用情况的系统级性能工具。

性能工具：系统内存

本章概述了系统级的 Linux 内存性能工具。本章将讨论这些工具可以测量的内存统计信息，以及如何使用各种工具收集这些统计结果。阅读本章后，你将能够：

- 理解系统级性能的基本指标，包括内存的使用情况。
- 明白哪些工具可以检索这些系统级性能指标。

3.1 内存性能统计信息

每一种系统级 Linux 性能工具都提供了不同的方式来提取类似的统计结果。虽然没有工具能显示全部的信息，但是有些工具显示的统计信息是相同的。本章开始将对这些统计数据的详细信息进行说明，之后在介绍工具时会引用这些描述。

3.1.1 内存子系统和性能

在现代处理器中，与 CPU 执行代码或处理信息相比，向内存子系统保存信息或从中读取信息一般花费的时间更长。通常，在 CPU 执行指令或处理数据前，它会消耗相当多的空闲时间来等待从内存中取出指令和数据。处理器用不同层次的高速缓存（cache）来弥补这种缓慢的内存性能。工具，如 oprofile，可以显示各种处理器高速缓存缺失所发生的位置。

3.1.2 内存子系统（虚拟存储器）

任何给定的 Linux 系统都有一定容量的 RAM 或物理内存。在这个物理内存中寻址时，

Linux 将其分成块或内存“页”。当对内存进行分配或传送时，Linux 操作的单位是页，而不是单个字节。在报告一些内存统计数据时，Linux 内核报告的是每秒页面的数量，该值根据其运行的架构可以发生变化。清单 3.1 创建了一个小的应用程序来显示当前架构中每一页的字节数。

清单 3.1

```
#include <unistd.h>
int main(int argc, char *argv[])
{
    printf("System page size: %d\n", getpagesize());
}
```

对 IA32 架构而言，页面大小为 4KB。极少数情况下，这些页面大小的内存块会导致极高的跟踪开销，所以，内核用更大的块来操作内存，这些块被称为 HugePage（大页面）。它们的容量为 2048KB，而不是 4KB，这大大降低了管理庞大内存的开销。某些应用，如 Oracle，用这些大页面加载内存中的大量数据，同时又最小化 Linux 内核的管理开销。如果 HugePage 不能完全被填满，就会浪费相当多的内存。一个半填充的普通页面浪费 2KB 内存，而一个半填充的 HugePage 就会浪费 1024KB 的内存。

Linux 内核可以分散收集这些物理页面，向应用程序呈现出一个精心设计的虚拟内存空间。

3.1.2.1 交换（物理内存不足）

所有系统 RAM 芯片的物理内存容量都是固定的。即使应用程序需要的内存容量大于可用的物理内存，Linux 内核仍然允许这些程序运行。Linux 内核使用硬盘作为临时存储器，这个硬盘空间被称为交换分区（swap space）。

尽管交换是让进程运行的极好的方法，但它却慢的要命。与使用物理内存相比，应用程序使用交换的速度可以慢到一千倍。如果系统性能不佳，确定系统使用了多少交换通常是有用的。

3.1.2.2 缓冲区（buffer）和缓存（cache）（物理内存太多）

相反，如果你的系统物理内存容量超过了应用程序的需求，Linux 就会在物理内存中缓存近期使用过的文件，这样，后续访问这些文件时就不用去访问硬盘了。对要频繁访问硬盘的应用程序来说，这可以显著加速其速度，显然，对经常启动的应用程序而言，这是特别有用的。应用程序首次启动时，它需要从硬盘读取；但是，如果应用程序留着缓存中，那它就需要从更快速的物理内存读取。这个硬盘缓存不同于前面章节提到的处理器缓存。除了 oprofile、valgrind 和 kcachegrind 之外，大多数工具在报告“缓存”的统计信息时实际指的是硬盘缓存。

除了高速缓存，Linux 还使用了额外的存储作为缓冲区。为了进一步优化应用程序，Linux 为需要被写回硬盘的数据预留了存储空间。这些预留空间被称为缓冲区。如果应用程序要将数据写回硬盘，通常需要花费较长时间，Linux 让应用程序立刻继续执行，但将文件数据保存到内存缓冲区。在之后的某个时刻，缓冲区被刷新到硬盘，而应用程序可以立即继续。

高速缓存和缓冲区的使用使得系统内空闲的内存很少，这会让人感到泄气，但这未必是件坏事。默认情况下，Linux 试图尽可能多的使用你的内存。这是好事。如果 Linux 借测到有空闲内存，它就会将应用程序和数据缓存到这些内存以加速未来的访问。由于访问内存的速度比访问硬盘的速度快了几个数量级，因此，这就可以显著地提升整体性能。如果系统需要缓存空间做更重要的事情，那么缓存空间将被擦除并交给系统。之后，对原来被缓存对象的访问就需要转向硬盘来满足。

3.1.2.3 活跃与非活跃内存

活跃内存是指当前被进程使用的内存。不活跃内存是指已经被分配了，但暂时还未使用的内存。这两种类型的内存没有本质上的区别。需要时，Linux 找出进程最近最少使用的内存页面，并将它们从活跃列表移动到不活跃列表。当要选择把哪个内存页交换到硬盘时，内核就从不活跃内存列表中进行选择。

3.1.2.4 高端与低端内存

对拥有 1GB 或更多物理内存的 32 位处理器（比如 IA32）来说，Linux 管理内存时必须将其分为高端与低端内存。高端内存不能直接被 Linux 内核访问，而是必须在使用前映射到低端内存范围内。64 位处理器（比如 AMD64/EM6T、Alpha 或 Itanium）没有这个问题，因为它们可以直接寻址当前系统可用的额外内存。

3.1.2.5 内核的内存使用情况（分片）

除了应用程序需要分配内存外，Linux 内核也会为了记账的目的消耗一定量的内存。记账包括，比如跟踪从网络或磁盘 I/O 来的数据，以及跟踪哪些进程正在运行，哪些正在休眠。为了管理记账，内核有一系列缓存，包含了一个或多个内存分片。每个分片为一组对象，个数可以是一个或多个。内核消耗的内存分片数量取决于使用的是 Linux 内核的哪些部分，而且还可以随着机器负载类型的变化而变化。

3.2 Linux性能工具：CPU与内存

现在开始讨论性能工具，它们能使你抽取前面所述的那些内存性能信息。

3.2.1 vmstat (II)

如前所见，vmstat 能提供多个不同方面的系统性能信息——尽管它的主要目的（如同下面展示的一样）是提供虚拟内存系统信息。除了前一章描述的 CPU 性能统计信息外，它还可以告诉你下述信息：

- 使用了多少交换分区。
- 物理内存是如何被使用的。
- 有多少空闲内存。

你可以看到，vmstat（通过其显示的统计数据）在一行文本中就提供了关于系统运行状况与性能的丰富信息。

3.2.1.1 系统范围内与内存相关的系统级选项

vmstat 除了提供 CPU 统计信息外，你还可以通过如下命令行调用 vmstat 来调查内存统计信息：

```
vmstat [-a] [-s] [-m]
```

和前面一样，你可以在两种模式下运行 vmstat：采样模式和平均模式。添加命令行选项能让你获得 Linux 内核使用内存的性能统计信息。表 3-1 给出了 vmstat 可接受的选项。

表 3-1 vmstat 命令行选项

选 项	说 明
-a	该项改变内存统计信息的默认输出以表示活跃 / 非活跃内存量，而不是缓冲区和高速缓存使用情况的信息
-s (procps 3.2 或更高版本)	打印输出 vm 表。自系统启动开始的综合统计信息。该项不能用于采样模式，它包含了内存和 CPU 的统计数据
-m (procps 3.2 或更高版本)	该项输出内核分片信息。键入 cat/proc/slabinfo 可以获得同样的信息。信息详细展示了内核内存是如何分配的，并有助于确定哪部分内核消耗内存最多

表 3-2 所示列表为 vmstat 可以提供的内存统计信息。与 CPU 统计信息一样，当运行于普通模式时，vmstat 提供的第一行信息为所有速率统计信息（so 和 si）的均值以及所有数字统计信息的瞬时值（swpd、free、buff、cache、active 和 inactive）。

表 3-2 与内存相关的 vmstat 输出统计信息

列	说 明
swpd	当前交换到硬盘的内存总量
free	未被操作系统或应用程序使用的物理内存总量
buff	系统缓冲区大小（单位为 KB），或用于存放等待保存到硬盘的数据的内存大小（单位为 KB）。该存储区允许应用程序向 Linux 内核发出写调用后立即继续执行（而不是等待直到数据被提交到硬盘）
cache	用于保存之前从硬盘读取的数据的系统高速缓存或内存的大小（单位为 KB）。如果应用程序再次需要该数据，内核可以从内存而非硬盘抓取数据，由此可提高性能

(续)

列	说 明
active	被使用的活跃内存量。活跃 / 不活跃的统计数据与缓冲区 / 高速缓存的是正交的；缓冲区和高速缓存可以是活跃的，也可以是不活跃的
inactive	不活跃的内存总量（单位为 KB），或一段时间未被使用，适合交换到硬盘的内存量
si	上一次采样中，从硬盘进来的内存交换速率（单位为 KB/s）
so	上一次采样中，到硬盘去的内存交换速率（单位为 KB/s）
pages paged in	从硬盘读入系统缓冲区的内存总量（单位为页）（在大多数 IA32 系统中，一页为 4KB）
pages paged out	从系统高速缓存写到硬盘的内存总量（单位为页）（在大多数 IA32 系统中，一页为 4KB）
pages swapped in	从交换分区读入系统内存的内存总量（单位为页）
pages swapped out	从系统内存写到交换分区的内存总量（单位为页）
used swap	Linux 内核目前使用的交换分区容量
free swap	当前可用的交换分区容量
total swap	系统的交换分区总量，即 used swap 与 free swap 之和

对给定机器而言，vmstat 能提供其虚拟存储系统当前状态的良好概览。虽然它不会为每个可用的 Linux 每次性能统计数据提供一个完整且详细的列表，但它给出的简洁输出可以表明系统内存整体上是如何被使用的。

3.2.1.2 用法示例

如前面章节所见，清单 3.2 中，如果 vmstat 调用时没有使用任何命令行选项，它显示的是从系统启动开始的性能统计数据的均值 (si 和 so)，以及其他统计信息的瞬时值 (swpd、free、buff 和 cache)。本例中，我们可以看到系统已经有大约 500MB 的内存交换到了硬盘。约 14MB 系统内存是空闲的。约 4MB 用于缓冲区，以保存还未刷新到硬盘的数据。约 627MB 用于硬盘缓存，以保存过去从硬盘读取的数据。

清单 3.2

```
bash-2.05b$ vmstat
procs -----memory----- swap----io-----system---cpu-----
 r b swpd free buff cache si so bi bo in cs us sy id wa
 0 0 511012 14840 4412 642072 33 31 204 247 1110 1548 8 5 73 14
```

在清单 3.3 中，我们要求 vmstat 显示活跃与非活跃页面的数量信息。非活跃页面的数量表明了有多少内存可以交换到硬盘，有多少内存是当前可用的。本例中，我们可以看到活跃内存有 1310MB，只有 78MB 被认为是不活跃的。该机拥有大量内存，且大部分都被使用，处于活跃状态。

清单 3.3

```
bash-2.05b$ vmstat -a
procs .....memory..... swap... io... system... cpu...
r b swpd free inact active si so bi bo in cs us sy id wa
2 1 514004 5640 79816 1341208 33 31 204 247 1111 1548 8 5 73 14
```

接下来，在清单 3.4 中，我们看到的是一个不同的系统，其内存数据交换频繁。si 列显示在每个采样期间，数据的读交换率分别为 480KB、832KB、764KB、344KB 和 512KB。so 列显示在每个采样期间，内存数据写交换率分别为 9KB、0KB、916KB、0KB、1068KB、444KB 和 792KB。这些结果可以说明该系统没有足够的内存来处理所有的运行进程。当一个进程的内存被保存下来，以便为之前已经交换到硬盘的应用程序腾位置时，就会出现高频率的换入和换出。如果有两个运行程序需要的内存量都超过了系统可提供的量，后果就会很糟糕。比如，两个进程都在使用大量内存，且它们都试图同时运行，而每个进程都可以导致另一个的内存被写交换。当一个程序需要一块内存时，它就会把另一个程序需要的一块内存踢出去。而当另一个应用程序开始运行时，它又会把第一个程序正在使用的一块内存踢出去，并等待自己的内存块从交换分区加载进来。这可能会导致两个应用程序出现停顿，以等待它们的内存从交换分区取回，然后才能继续执行。只要一个程序进步一点点，它就会将另一个进程使用的内存交换出去，从而导致这个程序慢下来。这种情况被称为颠簸。发生颠簸时，系统会花大量的时间将内存读出或写入交换分区，系统性能就会急剧下降。

具体到这个例子，交换最终停止了，最有可能的原因是交换到硬盘的内存不是第一个进程立即需要的。这就意味着交换是有效的，不是正在使用的内存内容被写入到硬盘，然后内存就会分配给需要它的进程。

清单 3.4

```
[ezolt@localhost book]$ vmstat 1 100
procs .....memory..... swap... io... system... cpu...
r b swpd free buff cache si so bi bo in cs us sy id wa
2 1 131560 2320 8640 53036 1 9 107 69 1137 426 10 7 74 9
0 1 131560 2244 8640 53076 480 0 716 0 1048 207 6 1 0 93
1 2 132476 3424 8592 53272 832 916 1356 916 1259 692 11 4 0 85
1 0 132476 2400 8600 53280 764 0 1040 40 1288 762 14 5 0 81
0 1 133544 2656 8624 53392 344 1068 1096 1068 1217 436 8 3 5 84
0 1 133988 2300 8620 54288 512 444 1796 444 1090 230 5 1 2 92
0 0 134780 3148 8612 53688 0 792 0 792 1040 166 5 1 92 2
0 0 134780 3148 8612 53688 0 0 0 0 1050 158 4 1 95 0
0 0 134780 3148 8612 53688 0 0 0 0 1148 451 7 2 91 0
0 0 134780 3148 8620 53680 0 0 0 12 1196 477 8 2 78 12
...
```

清单 3.5 在前面的章节已经给出了，如其所示，vmstat 可以展示很多种不同的系统统计信息。现在当我们查看它时，我们可以看到一些相同的统计数据以不同的输出模式呈现，比如 active、inactive、buffer、cache 和 used swap。但是也出现了一些新的统计信息，如 total memory，该数据表示系统总共有 1516MB 内存；total swap，该数据表示系统总共有 2048MB 的交换分区。当试图确定交换分区和当前使用内存的百分比时，了解系统总量是有帮助的。另一个有趣的统计信息是 pages paged in，它表示从硬盘读入的页面总数。这个统计信息包括启动应用程序读取的页面，以及该应用程序本身可以使用的页面。

清单 3.5

```
bash-2.05b$ vmstat -s
      1552528  total memory
      1546692  used memory
      1410448  active memory
       11100  inactive memory
        5836  free memory
       2676  buffer memory
      645864  swap cache
      2097096  total swap
      526280  used swap
      1570816  free swap
    20293225 non-nice user cpu ticks
    18284715 nice user cpu ticks
    17687435 system cpu ticks
    357314699 idle cpu ticks
    67673539 IO-wait cpu ticks
     352225 IRQ cpu ticks
    4872449 softirq cpu ticks
    495248623 pages paged in
    600129070 pages paged out
    19877382 pages swapped in
    18874460 pages swapped out
    2702803833 interrupts
    3763550322 CPU context switches
    1094067854 boot time
    20158151 forks
```

最后，在清单 3.6 中，我们看到 vmstat 可以提供关于 Linux 内核如何分配其内存的信息。如前所述，Linux 内核有一系列“分片”来保存其动态数据结构。vmstat 显示每一个分片（Cache），展示使用了多少元素（Num），分配了多少（Total），每个元素的大小（Size），整个分片使用了多少内存页（Pages）。这些信息有助于跟踪内核究竟是怎样使用其内存的。

清单 3.6

```
bash-2.05b$ vmstat -m
Cache                  Num  Total   Size  Pages
udf_inode_cache          0     0    416      9
```

```

fib6_nodes          7    113    32    113
ip6_dst_cache      9     17   224     17
ndisc_cache         1     24   160     24
raw6_sock           0     0   672      6
udp6_sock           0     0   640      6
tcp6_sock          404   441  1120      7
ip_fib_hash        39   202     16   202
ext3_inode_cache  1714  3632    512      8
...

```

vmstat 提供了一种简便的方法来抽取大量的 Linux 内存子系统的信息。与默认输出界面上的其他信息结合起来，它就展示出了一个关于系统运行状况和资源使用情况的图象。

3.2.2 top (2.x 和 3.x)

前面章节已经讨论过，top 能同时给出系统级或特定进程的性能统计信息。默认情况下，top 展示的是对进程的 CPU 消耗量进行降序排列的列表，但它也可以调整为按内存使用总量排序，以便你能跟踪到哪个进程使用的内存最多。

3.2.2.1 内存性能相关的选项

top 不用任何特定命令行选项来控制其显示内存统计信息。它的调用命令行如下：

```
top
```

不过，一旦开始运行，top 允许你选择显示系统级内存信息，还是显示按内存使用量排序的进程。按内存消耗量排序被证明对确定哪个进程消耗了最多内存是非常有帮助的。表 3-3 说明了不同的与内存相关的切换项。

表 3-3 top 运行时切换项

选 项	说 明
m	该项切换是否将内存使用量信息显示到屏幕
M	按任务使用的内存量排序。由于分配给进程的内存量可能会大于其使用量，因此，该项按驻留集大小排序。驻留集大小是指进程实际使用量，而不是简单的进程请求量

表 3-4 给出了 top 能提供的整个系统以及单个进程的内存性能统计数据。top 有两个不同的版本 2.x 和 3.x，它们在输出统计数据的名称上有些微差异。表 3-4 对两个版本的名称都进行了说明。

表 3-4 top 内存性能统计信息

选 项	说 明
%MEM	进程使用内存量占系统物理内存的百分比
SIZE (v 2.x)	进程虚拟内存使用总量。其中包括了应用程序分配到但未使用的全部内存
VIRT (v 3.x)	
SWAP	进程使用的交换区（单位为 KB）总量
RSS (v 2.x)	应用程序实际使用的物理内存总量
RES (v 3.x)	

(续)

选 项	说 明
TRS (v 2.x) CODE (v 3.x)	进程的可执行代码使用的物理内存总量 (单位为 KB)
DSIZE (v 2.x) DATA (v 3.x)	专门分配给进程数据和堆栈的内存总量 (单位为 KB)
SHARE (v 2.x) SHR (v 3.x)	可与其他进程共享的内存总量 (单位为 KB)
D (v 2.x) nDRT (v 3.x)	需要刷新到硬盘的脏页面的数量
Mem: total, used, free	对物理内存来说, 该项表示的是其总量、使用量和空闲量
swap: total, used, free	对交换分区来说, 该项表示的是其总量、使用量和空闲量
active (v 2.x)	当前活跃的物理内存总量
inactive (v 2.x)	非活跃且一段时间内未被使用的物理内存总量
buffers	用于缓冲区写入硬盘的数值的物理内存总量 (单位为 KB)

top 提供了不同运行进程的大量的内存信息。如同后续章节将会讨论的, 你可以使用这些信息来确定应用程序究竟是如何分配和使用内存的。

3.2.2.2 用法示例

清单 3.7 与前面章节给出的 top 运行示例相似。不过这个例子中, 请注意在缓冲区中有大约 84MB 是空闲的, 而总的物理内存容量为 1024MB。

清单 3.7

```
[ezolt@wintermute doc]$ top
top - 15:47:03 up 24 days, 4:32, 15 users, load average: 1.17, 1.19, 1.17
Tasks: 151 total, 1 running, 138 sleeping, 11 stopped, 1 zombie
Cpu(s): 1.2% us, 0.7% sy, 0.0% ni, 93.0% id, 4.9% wa, 0.0% hi, 0.1% si
Mem: 1034320k total, 948336k used, 85984k free, 32840k buffers
Swap: 2040244k total, 276796k used, 1763448k free, 460864k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
26364	root	16	0	405m	71m	321m	S	4.0	7.1	462:25.50	X
17345	ezolt	16	0	176m	73m	98m	S	2.6	7.3	1:17.48	soffice.bin
18316	ezolt	16	0	2756	1096	1784	R	0.7	0.1	0:05.54	top
26429	ezolt	16	0	65588	52m	12m	S	0.3	5.2	16:16.77	metacity
26510	ezolt	16	0	19728	5660	16m	S	0.3	0.5	27:57.87	clock-applet
26737	ezolt	16	0	70224	35m	20m	S	0.3	3.5	8:32.28	gnome-terminal
1	root	16	0	2396	448	1316	S	0.0	0.0	0:01.72	init
2	root	RT	0	0	0	0	S	0.0	0.0	0:00.88	migration/0
3	root	34	19	0	0	0	S	0.0	0.0	0:00.01	ksoftirqd/0

4 root	RT	0	0	0	S	0.0	0.0	0:00.35	migration/1
5 root	34	19	0	0	S	0.0	0.0	0:00.01	ksoftirqd/1
6 root	RT	0	0	0	S	0.0	0.0	0:34.20	migration/2
7 root	34	19	0	0	S	0.0	0.0	0:00.01	ksoftirqd/2

和前面一样，top 可以被定制为只显示观察过程中你感兴趣的内容。清单 3.8 给出的高度配置界面只显示了内存性能统计信息。

清单 3.8

Mem:	1034320k	total,	948336k	used,	85984k	free,	33024k	buffers	
Swap:	2040244k	total,	276796k	used,	1763448k	free,	460680k	cached	
<hr/>									
VIRT	RES	SHR	%MEM	SWAP	CODE	DATA	nFLT	nDRT	COMMAND
405m	71m	321m	7.1	333m	1696	403m	4328	0	X
70224	35m	20m	3.5	33m	280	68m	3898	0	gnome-terminal
2756	1104	1784	0.1	1652	52	2704	0	0	top
19728	5660	16m	0.5	13m	44	19m	17	0	clock-applet
2396	448	1316	0.0	1948	36	2360	16	0	init
0	0	0	0.0	0	0	0	0	0	migration/0
0	0	0	0.0	0	0	0	0	0	ksoftirqd/0
0	0	0	0.0	0	0	0	0	0	migration/1
0	0	0	0.0	0	0	0	0	0	ksoftirqd/1
0	0	0	0.0	0	0	0	0	0	migration/2
0	0	0	0.0	0	0	0	0	0	ksoftirqd/2
0	0	0	0.0	0	0	0	0	0	migration/3

top 提供了对内存统计数据的实时更新，并显示了哪个进程正在使用哪种类型的内存。当我们调查应用程序内存使用情况时，这些信息就变得有用多了。

3.2.3 procinfo (II)

正如我们在前面看到的，procinfo 提供的是系统级性能特性的概览。除了前面章节描述过的统计数据外，procinfo 还提供了一些内存统计数据，与 vmstat 和 top 类似，这些数据表明了当前内存是如何被使用的。

3.2.3.1 内存性能相关的选项

procinfo 没有任何选项来修改其内存统计信息的显示输出，因此其调用命令如下：

```
procinfo
```

procinfo 显示的是基本内存系统的内存统计信息，与 top 和 vmstat 类似，如表 3-5 所示。

表 3-5 procinfo 内存统计信息

选 项	说 明
Total	物理内存总量
Used	使用的物理内存总量
Free	未使用的物理内存总量
Shared	该项已过时，应忽略
Buffers	用于硬盘写缓冲区的物理内存总量
Page in	从硬盘读入的块数(通常大小为1KB)(该项在2.6.x版内核中有问题)
Page out	写入硬盘的块数(通常大小为1KB)(该项在2.6.x版内核中有问题)
Swap in	从交换分区读入的内存页数(该项统计数据在2.6.x版内核中有问题)
Swap out	写到交换分区的内存页数(该项统计数据在2.6.x版内核中有问题)

与 vmstat 和 top 非常相似，procinfo 是一种低开销的命令，它适于长时间在控制台或屏幕窗口中运行。它为系统运行状况和性能提供了良好的指示。

3.2.3.2 用法示例

清单 3.9 是 procinfo 的典型输出。如同你所看到的，它报告了系统使用虚拟内存的总体信息。在这个例子中，系统总共有 312MB 内存，其中有 301MB 被内核和应用程序使用，11MB 为系统缓冲区，还有 11MB 完全没有被使用。

清单 3.9

```
[ezolt@localhost procinfo-18]$ ./procinfo
Linux 2.6.6-1.435.2.3smp (bhcompile@tweety.build.redhat.com) (gcc 3.3.3 20040412 )
#1
1CPU [localhost]

Memory:      Total        Used        Free        Shared       Buffers
Mem:        320468      308776      11692          0       11604
Swap:       655192      220696     434496

Bootup: Sun Oct 24 10:03:43 2004   Load average: 0.44 0.53 0.51 3/110 32243

user :    0:57:58.92  9.0%  page in :      0
nice :    0:02:51.09  0.4%  page out:      0
system:   0:20:18.43  3.2%  swap in :      0
idle :    8:47:31.54 81.9%  swap out:      0
uptime:   10:44:01.94           context : 13368094

irq 0: 38645994 timer           irq 7:      2
irq 1: 90516 i8042              irq 8:      1 rtc
irq 2: 0 cascade [4]            irq 9:      2
```

```

irq 3:    742857 prism2_cs          irq 10:      2
irq 4:      6                      irq 11:    562551 uhci_hcd, yenta, yen
irq 5:      2                      irq 12:   1000803 i8042
irq 6:      8                      irq 14:   207681 ide0

```

procinfo 在单一信息屏中提供系统性能信息。虽然它给出了一些重要的内存统计数据，但 vmstat 或 top 更适合于调查系统级的内存使用情况。

3.2.4 gnome-system-monitor (II)

gnome-system-monitor 在许多方面就是 top 的图形表示。它使你能监控单个进程，并从它显示的图形上来观察系统负载。同时，它还提供了 CPU 和内存使用情况的基本图形。

3.2.4.1 内存性能相关的选项

gnome-system-monitor 可以从 Gnome 菜单调用。(Red Hat 9 及更高版本中，在 System Tools → System Monitor 选项下。) 不过，它也可以用下面的命令行来调用：

```
gnome-system-monitor
```

gnome-system-monitor 没有相关的命令行选项能影响内存性能测量。

3.2.4.2 用法示例

当你启动 gnome-system-monitor，并选择 System Monitor 标签后，你可以看到如图 3-1 所示的窗口。这个窗口使你能浏览图形，看看当前已经使用了多少物理内存和交换分区，以及使用情况随时间发生的变化。在这个例子中，我们看到 1007MB 的总量中已经使用了 969MB，且内存使用量在一段时间内是比较平稳的。

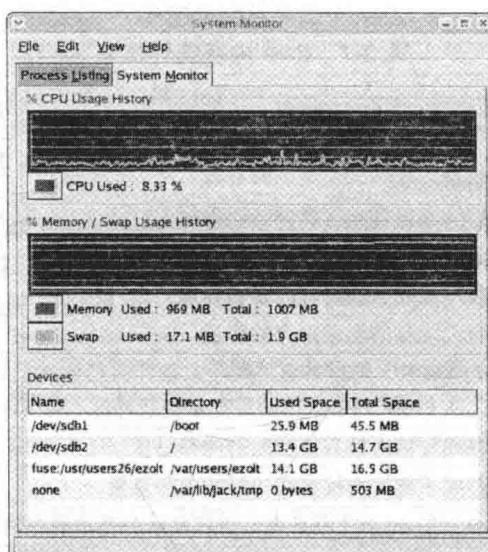


图 3-1

gnome-system-monitor 提供的数据图形视图使得对系统的观察更容易也更迅速，但是，却缺少了大部分的细节，比如内存是如何使用的。

3.2.5 free

free 提供的是系统使用内存的总体情况，包括空闲内存量。虽然 free 命令可能会显示一个特定系统没有多少空闲内存，但这不一定是坏事。Linux 内核不会让空闲内存一直闲着，而是会将它作为高速缓存用于硬盘读，或是作为缓冲区用于硬盘写。这可以显著提升系统性能。由于这些高速缓存和缓冲区总是可以被丢弃的，所以，当应用程序需要时这些内存还是可以使用的，free 显示的是空闲内存容量加上或减去这些缓冲区的容量。

3.2.5.1 内存性能相关的选项

用下面的命令行可以调用 free：

```
free [-l] [-t] [-s delay] [-c count]
```

表 3-6 说明的参数可以修改 free 显示的统计信息类型。与 vmstat 非常相似，free 可以周期性地显示更新内存统计数据。

表 3-6 free 命令行选项

选 项	说 明
-s delay	使 free 按每 delay 秒的间隔输出新的内存统计数据
-c count	使 free 输出 count 次新的统计数据
-l	向你展示使用了多少高端内存和多少低端内存

free 实际上显示了一些所有内存统计工具的最完整的内存统计信息。这些统计信息如表 3-7 所示。

表 3-7 free 内存统计信息

统计信息	说 明
Total	物理内存与交换空间的总量
Used	使用的物理内存和交换分区的容量
Free	未使用的物理内存和交换分区的容量
Shared	该项已过时，应忽略
Buffers	用作硬盘写缓冲区的物理内存的容量
Cached	用作硬盘读缓存的物理内存的容量
-/+ buffers / cache	对 Used 列来说，若缓冲区 / 缓存未被计为已使用的内存，则该项显示的是将使用的内存容量。在 Free 列中给出的是如果把缓冲区 / 缓存计为空闲内存时的空闲内存容量
Low	低端内存或能被内核直接访问的内存总量
High	高端内存或不能被内核直接访问的内存总量
Totals	对 Total 、 Used 和 Free 列，该项显示的是该列中物理内存和交换分区的总和

free 提供的是 Linux 中系统级内存使用情况信息。它给出了相当完整的内存统计数据。

3.2.5.2 用法示例

不使用任何命令选项来调用 free，能让你获得内存子系统的整体信息。

如前所述，如果可能的话，Linux 会使用所有可用的内存来缓存数据和应用程序。在清单 3.10 中，free 告诉我们现在已经使用了 234 720 字节的内存，但是，如果忽略缓冲区和缓存，那么就只使用了 122 772 字节的内存。与之相反的是 free 列，当前我们有 150 428 字节内存是空闲的，同样的，如果已经将缓冲区和缓存计算在内（这是可以的，因为在 Linux 需要使用这部分内存时，它会丢弃这些缓冲区），那么，我们有 262 376 字节的空闲内存。

清单 3.10

[ezolt@wintermute procps-3.2.0]\$ free						
	total	used	free	shared	buffers	cached
Mem:	385148	234720	150428	0	8016	103932
-/+ buffers/cache:		122772	262376			
Swap:	394080	81756	312324			

尽管你可以自己合计这些列，但是清单 3.11 所示的 -t 标志可以告诉你加上交换分区和实际内存的总数。在这个例子中，系统有 376MB 的物理内存和 384MB 的交换分区。系统可获得的内存总量为 376MB 加上 384MB，即大约 760MB。总的空闲内存计算方法为 134MB 的物理内存加上 259MB 的交换分区，产生总共 393MB 的空闲内存。

清单 3.11

[ezolt@wintermute procps-3.2.0]\$ free -t						
	total	used	free	shared	buffers	cached
Mem:	385148	247088	138060	0	9052	115024
-/+ buffers/cache:		123012	262136			
Swap:	394080	81756	312324			
Total:	779228	328844	450384			

最后，free 还能告诉你系统使用的高端和低端内存量。这主要用于具有 1GB 或更多物理内存的 32 位机器（如 IA32）。（32 位机器是唯一有高端内存的机器。）清单 3.12 展示了一个系统，该系统的空闲内存非常小，总共只有 6MB。它显示出系统有 876MB 的低端内存和 640MB 的高端内存。同时，这个系统的缓存内存容量比缓冲内存容量大很多，这表明它可能更加积极地将数据写入硬盘，而不是长时间将其留在缓冲区里。

清单 3.12

fas% free -l						
	total	used	free	shared	buffers	cached
Mem:	1552528	1546472	6056	0	7544	701408
Low:	897192	892800	4392			

High:	655336	653672	1664
-/+ buffers/cache:	837520	715008	
Swap:	2097096	566316	1530780

free很好地体现了系统内存是如何被使用的。虽然可能需要点时间来适应其输出格式，但是它包含了所有重要的内存统计信息。

3.2.6 slabtop

slabtop与top相似，但是它并不显示系统中进程的CPU和内存使用情况的信息，slabtop实时显示内核是如何分配其各种缓存的，以及这些缓存的被占用情况。在内部，内核有一系列的缓存，它们由一个或多个分片（slab）构成。每个分片包括一组对象，对象个数为一个或多个。这些对象可以是活跃的（使用的）或非活跃的（未使用的）。slabtop向你展示的是不同分片的状况。它显示了这些分片的被占用情况，以及它们使用了多少内存。

3.2.6.1 内存性能相关的选项

slabtop用如下命令行调用：

```
slabtop [--delay n -sort={a | b | c | l | v | n | o | p | s | u}]
```

表3-8对slabtop的命令行选项进行了说明。

表3-8 slabtop命令行选项

选 项	说 明	
- --delay	指定 slabtop 在更新间隔期间应等待多长时间	
- --sort {order}	指定输出顺序。order 从下列各项中选择	
	a	按每个分片所包含的活跃对象数排序
	b	按照特定缓存中的每个分片所包含的全部对象（活跃的和非活跃的）数量排序
	c	按每个缓存所用内存总量排序
	l	按每个缓存使用的分片数排序
	v	按每个缓存使用的活跃分片数排序
	n	按缓存名称排序
	o	按特定缓存中的对象数量排序
	p	按每个分片使用的页数排序
	s	按缓存中对象的大小排序

slabtop可以一窥Linux内核的数据结构。每一种分片类型都与Linux内核紧密相关，不过，对这些分片的描述已经超出了本书的范围。如果某个特定分片使用了大量的内核内存，那么阅读Linux内核源代码和搜索互联网是找出这些分片用在哪里的最好的两种方法。

3.2.6.2 用法示例

如清单3.13所示，默认情况下，slabtop会填满整个控制台，且每3秒就更新一次统计

数据。在这个例子中，你可以看见 size-64 分片的对象数最多，但其中只有一半是活跃的。

清单 3.13

```
[ezolt@wintermute proc]$ slabtop
Active / Total Objects (% used)      : 185642 / 242415 (76.6%)
Active / Total Slabs (% used)        : 12586 / 12597 (99.9%)
Active / Total Caches (% used)        : 88 / 134 (65.7%)
Active / Total Size (% used)         : 42826.23K / 50334.67K (85.1%)
Minimum / Average / Maximum Object : 0.01K / 0.21K / 128.00K

OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
66124  34395  52%  0.06K  1084      61    4336K size-64
38700  35699  92%  0.05K  516       75    2064K buffer_head
30992  30046  96%  0.15K  1192      26    4768K dentry_cache
21910  21867  99%  0.27K  1565      14    6260K radix_tree_node
20648  20626  99%  0.50K  2581      8     10324K ext3_inode_cache
11781   7430  63%  0.03K   99      119    396K size-32
9675   8356  86%  0.09K  215       45    860K vm_area_struct
6024   2064  34%  0.62K  1004      6     4016K ntfs_big_inode_cache
4520   3633  80%  0.02K   20      226    80K anon_vma
4515   3891  86%  0.25K   301      15    1204K filp
4464   1648  36%  0.12K   144      31    576K size-128
3010   3010  100% 0.38K   301      10    1204K proc_inode_cache
2344    587  25%  0.50K   293      8     1172K size-512
2250   2204  97%  0.38K   225      10    900K inode_cache
2100    699  33%  0.25K   140      15    560K size-256
1692   1687  99%  0.62K   282      6     1128K nfs_inode_cache
1141   1141  100% 4.00K  1141      1    4564K size-4096
```

由于 slabtop 提供的信息是周期性更新的，因此它是观察 Linux 内核的内存使用情况随工作负载变化而变化的极好的方法。

3.2.7 sar (II)

在对内存统计信息进行监控时，sar 作为 CPU 性能工具的所有优势仍然存在，比如简单记录样本、提取多种输出格式、对样本加时间戳等。sar 提供的信息与其他内存统计工具类似，比如空闲内存、缓冲区、高速缓存和交换分区总量的当前值。但是，它还会提供这些数值的变化率，以及当前消耗的物理内存和交换分区的百分比信息。

3.2.7.1 内存性能相关的选项

sar 使用如下命令行进行调用：

sar [-B] [-r] [-R]

默认情况下，sar 只显示 CPU 性能统计数据；因此，如果想要检索任何内存子系统的统计信息，你就需要使用表 3-9 给出的选项。

表 3-9 sar 命令行选项

选 项	说 明
-B	报告的信息为内核与磁盘之间交换的块数。此外，对 v2.5 之后的内核版本，该项报告的信息为缺页数量
-W	报告的是系统交换的页数
-r	报告系统使用的内存信息。它包括总的空闲内存、正在使用的交换分区、缓存和缓冲区的信息

sar 给出的 Linux 内存子系统的信息相当完整。sar 强过其他工具的一点就是，除了绝对值之外，它还提供一些重要数值的变化率。你可以通过这些数值查看内存使用情况究竟是如何随时间变化的，而不用去找出这些值在样本之间的差异。表 3-10 给出了 sar 提供的内存统计信息。

表 3-10 sar 内存统计信息

选 项	说 明
pgpgin/s	内核以分页形式每秒从磁盘换入的内存容量（以 KB 为单位）
pgpgout/s	内核以分页形式每秒换出到磁盘的内存容量（以 KB 为单位）
fault/s	每秒内存子系统需满足的缺页总数。这些缺页不一定需要访问磁盘
majflt/s	每秒内存子系统需满足的缺页总数，这些缺页需要访问磁盘
pswpin/s	每秒系统装入内存的交换分区总量（按页计）
pswpout/s	每秒系统写入到交换分区的内存总量（按页计）
kbmemfree	当前空闲或未被使用的物理内存总量（以 KB 为单位）
kbmemused	当前被使用的物理内存总量（以 KB 为单位）
%memused	被使用的物理内存总量所占的百分比
kbbuffers	用作磁盘写缓冲区的物理内存总量
kbcached	用作磁盘读缓存的物理内存总量
kbswpfree	当前空闲的交换分区容量（以 KB 为单位）
kbswpused	当前被使用的交换分区容量（以 KB 为单位）
%swpused	被使用的交换分区百分比
kbswpcad	该项内存包括了交换到磁盘的和已存在于内存中的。如果需要内存，它可以立即被重用，因为数据已经存在于交换区域了
frmpg/s	系统释放内存页面的速率。若数值为负，则表示系统正在分配它们
bufpg/s	系统将新内存页面用作缓冲的速率。若数值为负，则表示缓冲数量正在减少，系统对它们的使用量也在减少

尽管 sar 没有高端和低端内存统计数据，但是它几乎提供了其他所有的内存统计信息。事实上，sar 还能记录网络 CPU 和磁盘 I/O 的统计数据，这使得它非常强大。

3.2.7.2 用法示例

清单 3.14 展示了 sar 提供的关于当前内存子系统状态的信息。从这些结果我们可以看到系统使用的内存量占整个内存容量的比例变化范围为 98.87% 到 99.25%。在观察期间，空闲内存量从 11MB 下降到 7MB。被使用的交换分区百分比徘徊在 11% 左右。系统的数据缓存容量约为 266MB，且大约有 12MB 的缓冲可以写到磁盘。

清单 3.14

[ezolt@scrffy manuscript]\$ sar -r 1 5										
Linux 2.6.8-1.521smp (scrffy) 10/25/2004										
	kbmemfree	kbmemused	%memused	kbbuffers	kbcached	kbswpfree	kbswpused	%swpused	kbswpcad	
09:45:31 AM	11732	1022588	98.87	12636	272284	1816140	224104	10.98	66080	
09:45:32 AM	10068	1024252	99.03	12660	273300	1816140	224104	10.98	66080	
09:45:33 AM	5348	1028972	99.48	12748	275292	1816140	224104	10.98	66080	
09:45:35 AM	4932	1029388	99.52	12732	273748	1816140	224104	10.98	66080	
09:45:36 AM	6968	1027352	99.33	12724	271876	1815560	224684	11.01	66660	
Average:	7810	1026510	99.25	12700	273300	1816024	224220	10.99	66196	

清单 3.15 显示在第一个样本期间，系统使用空闲内存的速率约为每秒 82 个页面。然后系统释放了约 16 个页面，接着又使用了大约 20 个页面。观察期间只有一次缓冲页面数是增加的，速率为每秒 2.02 个页面。最后，缓存页面数减少了 2.02，不过最终，它们增加的速率为每秒 64.36 个页面。

清单 3.15

[ezolt@scrffy manuscript]\$ sar -R 1 5			
Linux 2.6.8-1.521smp (scrffy) 10/25/2004			
	frm pg/s	buf pg/s	campg/s
09:57:22 AM	-81.19	0.00	0.00
09:57:23 AM	8.00	0.00	0.00
09:57:24 AM	0.00	2.02	-2.02
09:57:25 AM	15.84	0.00	0.00
09:57:26 AM	-19.80	0.00	64.36
Average:	-15.54	0.40	12.55

清单 3.16 显示，在第三个样本期间，系统从内存写了大约 53 个页面到磁盘。该系统的缺页数相对较高，这就意味着内存页面在被分配和使用。幸运的是，这些都不是主缺页，系统不必为了解决它们而去访问磁盘。

清单 3.16

```
[ezolt@scrffy dvi]$ sar -B 1 5
Linux 2.6.8-1.521smp (scrffy)   10/25/2004

09:58:34 AM pgpgin/s pgpgout/s fault/s majflt/s
09:58:35 AM    0.00     0.00  1328.28     0.00
09:58:36 AM    0.00     0.00   782.18     0.00
09:58:37 AM    0.00    53.06   678.57     0.00
09:58:38 AM    0.00     0.00   709.80     0.00
09:58:39 AM    0.00     0.00   717.17     0.00
Average:      0.00    10.42   842.48     0.00
```

如你所见，sar 是一个强大的工具，通过增加存档、时间戳和同步收集多种不同类型统计信息的能力，它增强了其他系统内存性能工具的功能。

3.2.8 /proc/meminfo

Linux 内核提供用户可读文本文件 /proc/meminfo 来显示当前系统范围内的内存性能统计信息，它提供了系统范围内内存统计数据的超集，包括了 vmstat、top、free 和 procinfo 的信息，但是使用起来有一定的难度。如果你想定期更新，就需要自己写一个脚本或一些代码来实现这个功能。如果你想保存内存性能信息或是将其与 CPU 统计信息相协调，就必须创建一个新的工具或是写一个脚本。尽管如此，/proc/meminfo 提供的却是最完整的系统内存使用情况的信息。

3.2.8.1 内存性能相关的选项

/proc/meminfo 中的信息可以用如下命令行来检索：

```
cat /proc/meminfo
```

这个命令显示的统计信息如表 3-11 所示。

表 3-11 /proc/meminfo 内存统计信息（均以 KB 为单位）

选 项	说 明
MemTotal	系统物理内存总量
MemFree	空闲物理内存总量
Buffers	等待中的磁盘写操作的内存容量
Cached	用于缓存磁盘读操作的内存容量
SwapCached	在交换分区和物理内存中都存在的内存容量
Active	系统中当前处于活跃状态的内存容量
Inactive	当前处于非活跃状态且可用于交换的内存容量
HighTotal	高端内存容量（以 KB 为单位）

(续)

选 项	说 明
HighFree	空闲的高端内存容量 (以 KB 为单位)
LowTotal	低端内存容量 (以 KB 为单位)
LowFree	空闲的低端内存容量 (以 KB 为单位)
SwapTotal	交换内存容量 (以 KB 为单位)
SwapFree	空闲的交换内存容量 (以 KB 为单位)
Dirty	等待写入磁盘的内存
Writeback	当前被写入磁盘的内存
Mapped	用 mmap 映射到一个进程虚拟地址空间的内存总量
Slab	内核分片内存的总量 (以 KB 为单位)
Committed_AS	所需内存容量，在当前的工作负载下，这个容量几乎是不会耗尽的。通常情况下，内核会分配更多的内存，预期应用程序会超分配。如果所有的应用程序都使用自己被分配的内存，那么这个就是你需要的物理内存容量
PageTables	为内核页表保留的内存容量
VmallocTotal	vmalloc 可用的内核内存容量
VmallocUsed	vmalloc 已使用的内核内存容量
VmallocChunk	vmalloc 可用内存中最大的连续块
HugePages_Total	所有 HugePage 的总的大小
HugePages_Free	空闲 HugePage 的总量

/proc/meminfo 提供了大量的关于 Linux 内存子系统当前状态的信息。

3.2.8.2 用法示例

清单 3.17 是一个输出 /proc/meminfo 的例子。它给出的一些内存统计信息与我们在其他工具中看到的信息是相同的。但是有些统计数据则给出了新的信息。首先，Dirty 项显示系统当前有 24KB 的数据等待写入磁盘。其次，Committed_AS 项显示我们还需要更多一点的内存 (需要的量为 1068MB，而总量为 1024MB)，以避免出现可能的内存耗尽的情况。

清单 3.17

```
[ezolt@scruffy /tmp]$ cat /proc/meminfo
MemTotal:      1034320 kB
MemFree:       10788 kB
Buffers:        29692 kB
Cached:         359496 kB
SwapCached:    113912 kB
Active:         704928 kB
Inactive:      222764 kB
HighTotal:      0 kB
HighFree:       0 kB
LowTotal:      1034320 kB
```

```
LowFree:          10788 kB
SwapTotal:       2040244 kB
SwapFree:        1756832 kB
Dirty:            24 kB
Writeback:        0 kB
Mapped:           604248 kB
Slab:             51352 kB
Committed_AS:   1093856 kB
PageTables:      9560 kB
VmallocTotal:    3088376 kB
VmallocUsed:     26600 kB
VmallocChunk:    3058872 kB
HugePages_Total:  0
HugePages_Free:   0
Hugepagesize:    2048 kB
```

/proc/meminfo 收集的系统级 Linux 内存统计信息是最完整的。由于可以把它当作是一个文本文件，因此，任何自定义的脚本或程序都可以很容易地提取这些统计数据。

3.3 本章小结

本章重点关注了系统级内存性能衡量指标。这些指标主要展示的是操作系统是如何使用内存而不是特定的应用程序。

本章说明了性能工具（如 sar 和 vmstat）如何被用于从运行系统中提取其系统级内存统计信息。这些工具的输出表明了系统作为一个整体是怎样使用可用内存的。下一章将阐述研究单个进程的 CPU 使用情况的工具。

性能工具：特定进程 CPU

在用系统级性能工具找出是哪个进程降低了系统速度之后，你需要用特定进程性能工具来发现这个进程的行为。对此，Linux 提供了丰富的工具用于追踪一个进程和应用程序性能的重要统计信息。

阅读本章后，你将能够：

- 确定应用程序的运行时间是花费在内核上还是在应用程序上。
- 确定应用程序有哪些库调用和系统调用，以及它们花费的时间。
- 分析应用程序，找出哪些源代码行和函数的完成时间最长。

4.1 进程性能统计信息

分析应用程序性能的工具多种多样，并且从 UNIX 早期就以各种形式存在了。要了解一个应用程序的性能，至关重要的一点就是理解它与操作系统、CPU 和存储系统是怎样进行交互的。大多数应用程序不是独立的，因此需要一些对 Linux 内核和不同的函数库的调用。这些对 Linux 内核的调用（或系统调用）可能是简单的，如“我的 PID 是什么？”；也可能是复杂的，如“从磁盘读取 12 个数据块”。不同的系统调用会产生不同的性能影响。相应的，库调用也可以简单如内存分配，复杂如创建图形窗口。这些库调用也有不同的性能特点。

4.1.1 内核时间 vs. 用户时间

一个应用程序所耗时间最基本的划分是内核时间与用户时间。内核时间是消耗在 Linux

内核上的时间，而用户时间则是消耗在应用程序或库代码上的时间。Linux 有工具，如 time 和 ps，可以（大致）表明应用程序将其时间是花在了应用程序代码上还是花在了内核代码上。同时，还有如 oprofile 和 strace 这样的命令使你能跟踪哪些内核调用是代表该进程发起的，以及每个调用完成需要多少时间。

4.1.2 库时间 vs. 应用程序时间

任何应用程序，即便其复杂性非常低，也需要依赖系统库才能执行复杂的操作。这些库可能会导致性能问题，因此，能够查看应用程序在某个库中花费了多少时间就很重要了。虽然为了解决一个问题而去修改库的源代码并不总是实用，但是可以改变应用程序代码来调用不同的库函数，或者是调用更少的库函数。在库被应用程序使用时，ltrace 命令和 oprofile 工具包提供了分析库性能的途径。Linux 加载器 ld 的内置工具帮助你确定使用多个库是否会减慢应用程序的启动时间。

4.1.3 细分应用程序时间

当已经知道某应用程序是瓶颈后，Linux 可以向你提供工具来分析这个应用程序，以找出在这个程序中，时间都花在了哪里。工具 gprof 和 oprofile 可以生成应用程序的配置文件，确定是哪些源代码行花费了大量的时间。

4.2 工具

Linux 有各种各样的工具来帮助你确定应用程序的哪些部分是 CPU 的主要消费者。本节就对这些工具进行说明。

4.2.1 time

time 命令完成一项基本功能，当需要测试一条命令的性能时，通常会首先运行它。time 命令如同秒表一样，可以测量命令执行的时间。它测量的时间有三种类型：第一种测量的是真正的或经过的时间，即程序开始到结束执行之间的时间；第二种测量的是用户时间，即 CPU 代表该程序执行应用代码所花费的时间；第三种测量的是系统时间，即 CPU 代表该程序执行系统或内核代码所花费的时间。

4.2.1.1 CPU 性能相关的选项

time 命令（参见表 4-1）调用方式如下：

```
time [-v] application
```

对 application 程序计时，在其完成后，在标准输出上显示它的 CPU 使用情况。

表 4-1 time 命令行选项

选 项	说 明
-v	详细显示了程序的时间和统计信息。有些统计项为零，但是 Linux 内核 v2.6 比 v2.4 具备更多的有效统计项 大多数有效统计项既出现在标准模式中，也出现在详细模式中，不过详细模式为每个统计信息都提供了更好的说明

表 4-2 对 time 命令提供的有效输出统计信息进行了解释。其他项不进行测量，且总是显示为零。

表 4-2 与 CPU 相关的 time 输出

选 项	说 明
User time(seconds)	CPU 花费在应用程序上的秒数
System time(seconds)	CPU 代表应用程序花费在 Linux 内核上的秒数
Elapsed(wall-clock)time(h:mm:ss or m:ss)	应用程序从启动到完成所经历的时间（按墙钟时间计）
Percent of CPU this job got	进程运行时消耗 CPU 的百分比
Major(requiring I/O)page faults	主缺页故障的数量或需要从磁盘读取内存页的页故障数量
Minor(reclaiming a frame)page faults	次缺页故障的数量或不需要访问磁盘即可解决的页故障
Swaps	进程被交换到磁盘的次数
Voluntary context switches	进程让出 CPU（比如，进入睡眠状态）的次数
Involuntary context switches:	进程被迫让出 CPU 的次数
Page size(bytes)	系统的页面大小
Exit status	应用程序的退出状态

这个命令是启动调查的良好开端。它显示了应用程序执行了多长时间，其中，多少时间花费在 Linux 内核上，多少时间花费在你的应用程序上。

4.2.1.2 用法示例

包含在 Linux 中的 time 命令是跨平台 GNU 工具的一部分。默认命令输出会打印命令运行的大量统计信息，即使 Linux 不支持它们。如果数据不可用，那么 time 就只打印一个零。下面的命令是对 time 的一个简单调用。你可以在清单 4.1 中看到，经过的时间（约 3 秒）远大于用户时间（0.9 秒）和系统时间（0.13 秒）的总和，这是因为应用程序的大部分时间是用来等待输入的，而少量的时间是用于处理器的。

清单 4.1

```
[ezolt@wintermute manuscript]$ /usr/bin/time gcalctool
0.91user 0.13system 0:03.37elapsed 30%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (2085major+369minor)pagefaults 0swaps
```

清单 4.2 是 time 显示详细信息的例子。如同你看到的，这个输出比 time 的典型输出显示

了更多的信息。遗憾的是，大部分统计数据都是零，因为Linux不支持它们。大多数情况下，详细模式下提供的信息与标准模式下提供的信息是一样的，但其统计信息的标签更具有描述性。在这个例子中，我们可以看到，进程运行时使用了15%的CPU，运行用户代码的时间为1.15秒，运行内核代码的时间为0.12秒。它累计有2087个主缺页故障，或需要访问磁盘的内存故障；有371个不需要访问磁盘的缺页故障。大量的主缺页故障表明，在应用程序试图使用内存时，操作系统在不断的访问磁盘，这极有可能意味着进行了大量的交换。

清单4.2

```
[ezolt@wintermute manuscript]$ /usr/bin/time --verbose gcalctool
      Command being timed: "gcalctool"
      User time (seconds): 1.15
      System time (seconds): 0.12
      Percent of CPU this job got: 15%
      Elapsed (wall clock) time (h:mm:ss or m:ss): 0:08.02
      Average shared text size (kbytes): 0
      Average unshared data size (kbytes): 0
      Average stack size (kbytes): 0
      Average total size (kbytes): 0
      Maximum resident set size (kbytes): 0
      Average resident set size (kbytes): 0
      Major (requiring I/O) page faults: 2087
      Minor (reclaiming a frame) page faults: 371
      Voluntary context switches: 0
      Involuntary context switches: 0
      Swaps: 0
      File system inputs: 0
      File system outputs: 0
      Socket messages sent: 0
      Socket messages received: 0
      Signals delivered: 0
      Page size (bytes): 4096
      Exit status: 0
```

请注意，bash shell有内置time命令，因此，如果你运行bash并在没有指定执行路径的情况下执行time，你将得到如下输出：

```
[ezolt@wintermute manuscript]$ time gcalctool
```

real	0m3.409s
user	0m0.960s
sys	0m0.090s

bash 内置的 time 命令是很有用的，但是它提供的是进程执行信息的子集。

4.2.2 strace

strace 是当程序执行时，追踪其发起的系统调用的工具。系统调用是由或代表一个应用程序进行的 Linux 内核函数调用。strace 可以展示准确的系统调用，它在确定应用程序是如何使用 Linux 内核的方面是相当有用的。在分析大型程序或你完全不懂的程序时，跟踪系统调用的频率和长度是特别有价值的。通过查看 strace 的输出，你可以了解应用程序如何使用内核，以及它依赖于什么类型的函数。

如果你完全理解了一个应用程序，但是它有向系统库（如 libc 或 GTK）发起了调用，那么此时，strace 也是很有用的。在这种情况下，即使你知道应用程序是如何进行每一个系统调用的，库也可能会代表你的应用程序进行更多的系统调用。strace 可以迅速告诉你这些库都进行了哪些调用。

虽然 strace 主要用于跟踪进程与内核之间的交互，显示应用程序的每个系统调用的参数和结果，但是 strace 也可以提供不那么令人生畏的汇总信息。应用程序运行之后，strace 会给出一个表格，显示每个系统调用的频率和该类型调用所花费的总的时间。这个表格可以作为理解你的程序与 Linux 内核之间交互的首个关键信息。

4.2.2.1 CPU 性能相关的选项

如下的 strace 调用对性能测试是最有用的：

```
strace [-c] [-p pid] [-o file] [--help] [ command [ arg ... ] ]
```

如果 strace 不带任何选项运行，它将在标准错误输出上显示给定命令的所有系统调用。在试图发现为什么应用程序在内核中花费了大量时间时，这是很有帮助的。表 4-3 说明了一些 strace 选项，它们在跟踪性能问题时也是有用的。

表 4-3 strace 命令行选项

选 项	说 明
-c	使 strace 打印出统计信息的概要，而非所有系统调用的独立列表
-p pid	将给定 PID 添加到进程，并开始跟踪
-o file	strace 的输出将保存到 file
--help	列出 strace 选项的完整汇总

表 4-4 解释了 strace 汇总选项输出的统计信息。每一行输出都说明了特定系统调用的一组统计数据。

尽管上述说明的选项是与性能调查最相关的，但是 strace 也可以对其跟踪的系统调用进行类型过滤。strace 说明页和 --help 选项详细解释了用于选择要跟踪哪些系统调用的选项。对一般的性能优化，通常没有必要使用它们；不过如果需要的话，它们也是存在可用的。

表 4-4 与 CPU 相关的 strace 输出

选 项	说 明
% time	对全部系统调用的总时间来说，该项为这一个系统调用所花时间的百分比
seconds	这一个系统调用所花费的总秒数
usecs/call	这个类型的一个系统调用所花费的微秒数
calls	这个类型的所有调用的总数
errors	这个系统调用返回错误的次数

4.2.2.2 用法示例

清单 4.3 是用 strace 收集一个应用程序的系统调用统计信息的例子。如你所见，strace 提供了对系统调用非常好的分析，这些调用是代表应用程序执行的，在这里，这个应用程序就是 oowriter。本例中，我们查看 oowriter 是如何使用 read 系统调用的。我们看到 read 占用了 20% 的时间，共消耗了 0.44 秒。它被调用了 2427 次，平均下来，一次调用的时间为 184 微秒。在这些调用中，有 26 次返回了错误。

清单 4.3

```
[ezolt@wintermute tmp]$ strace -c oowriter
execve("/usr/bin/oowriter", ["oowriter"], /* 35 vars */) = 0
Starting OpenOffice.org ...


```

% time	seconds	usecs/call	calls	errors	syscall
20.57	0.445636	184	2427	26	read
18.25	0.395386	229	1727		write
11.69	0.253217	338	750	514	access
10.81	0.234119	16723	14	6	waitpid
9.53	0.206461	1043	198		select
4.73	0.102520	201	511	55	stat64
4.58	0.099290	154	646		gettimeofday
4.41	0.095495	58	1656	15	lstat64
2.51	0.054279	277	196		munmap
2.32	0.050333	123	408		close
2.07	0.044863	66	681	297	open
1.98	0.042879	997	43		writev
1.18	0.025614	12	2107		lseek
0.95	0.020563	1210	17		unlink
0.67	0.014550	231	63		getdents64
0.58	0.012656	44	286		mmap2
0.53	0.011399	68	167	2	ioctl
0.50	0.010776	203	53		readv

0.44	0.009500	2375	4	3 mkdir
0.33	0.007233	603	12	clone
0.29	0.006255	28	224	old_mmap
0.24	0.005240	2620	2	vfork
0.24	0.005173	50	104	rt_sigprocmask
0.11	0.002311	8	295	fstat64

strace 善于跟踪进程，但是它在一个应用程序上运行时会产生一些开销。其结果就是，strace 报告的调用次数可能会比它报告的每个调用的时间要更加可靠一些。应使用 strace 提供的次数作为调查的起点，而不是每个调用所花费时间的高度精确的测量值。

4.2.3 ltrace

ltrace 与 strace 的概念相似，但它跟踪的是应用程序对库的调用而不是对内核的调用。虽然 ltrace 主要用于提供对库调用的参数和返回值的精确跟踪，但是你也可以用它来汇总每个调用所花的时间。这使得你既可以发现应用程序有哪些库调用，又可以发现每个调用时间是多长。

使用 ltrace 要小心，因为它会产生具有误导性的结果。如果一个库函数调用了另一个函数，则花费的时间要计算两次。比如，如果库函数 foo() 调用了函数 bar()，则函数 foo() 的报告时间将是函数 foo() 代码运行的全部时间再加上函数 bar() 花费的时间。

记住了这个注意事项，ltrace 就还是揭示应用程序如何表现的有用的工具。

4.2.3.1 CPU 性能相关的选项

ltrace 提供与 strace 相似功能，其调用方法也和 strace 相近：

```
ltrace [-c] [-p pid] [-o filename] [-S] [--help] command
```

在上面的调用中，command 是你想要 ltrace 跟踪的命令。如果 ltrace 不带选项，它将在标准错误输出上显示所有的库调用。表 4-5 解释了与性能调查最相关的 ltrace 选项。

表 4-5 ltrace 命令行选项

选 项	说 明
-c	使得 ltrace 在命令执行完后打印出所有调用的汇总
-s	除了库调用之外，ltrace 还跟踪系统调用，该项与 strace 提供的功能相同
-p pid	跟踪有给定 PID 的进程
-o file	将 ltrace 的输出保存到 file
-- help	显示 ltrace 的帮助信息

同样的，汇总模式（summary mode）提供了应用程序执行期间的库调用的性能统计信息。表 4-6 说明了这些统计数据的含义。

表 4-6 与 CPU 相关的 ltrace 输出

选 项	说 明
% time	相对库调用花费的总时间，该项是这一个库调用所花时间的百分比
seconds	该项为这一个库调用所用的总秒数
usecs/call	该项为这个类型中一个库调用所花的微秒数
calls	该项为这个类型调用的总数
function	该项为库调用的名称

就像 strace，ltrace 有大量的选项可以修改其跟踪的功能。ltrace 的 --help 命令描述了这些选项，详细情况见于 ltrace 说明页。

4.2.3.2 用法示例

清单 4.4 是 ltrace 运行于 xeyes 命令的简单例子。xeyes 是一个 XWindow 应用程序，其功能是随着你的鼠标指针在屏幕上弹出一双眼睛。

清单 4.4

```
[ezolt@localhost manuscript]$ ltrace -c /usr/X11R6/bin/xeyes
% time      seconds   usecs/call   calls      function
-----
18.65      0.065967    65967        1 XSetWMProtocols
17.19      0.060803     86        702 hypot
12.06      0.042654    367       116 XQueryPointer
 9.51      0.033632   33632        1 XtAppInitialize
 8.39      0.029684     84       353 XFillArc
 7.13      0.025204    107       234 cos
 6.24      0.022091     94       234 atan2
 5.56      0.019656     84       234 sin
 4.62      0.016337    139       117 XtAppAddTimeOut
 3.19      0.011297     95       118 XtWidgetToApplicationContext
 3.06      0.010827     91       118 XtWindowForObject
 1.39      0.004934   4934        1 XtRealizeWidget
 1.39      0.004908   2454        2 XCreateBitmapFromData
 0.65      0.002291   2291        1 XtCreateManagedWidget
 0.12      0.000429    429        1 XShapeQueryExtension
 0.09      0.000332    332        1 XInternAtom
 0.09      0.000327     81        4 XtDisplay
 0.09      0.000320    106        3 XtGetGC
 0.05      0.000168     84        2 XSetForeground
 0.05      0.000166     83        2 XtScreen
 0.04      0.000153    153        1 XtParseTranslationTable
 0.04      0.000138    138        1 XtSetValues
```

0.04	0.000129	129	1 XmuCvtStringToBackingStore
0.03	0.000120	120	1 XtDestroyApplicationContext
0.03	0.000116	116	1 XtAppAddActions
0.03	0.000109	109	1 XCreatePixmap
0.03	0.000108	108	1 XtSetLanguageProc
0.03	0.000104	104	1 XtOverrideTranslations
0.03	0.000102	102	1 XtWindow
0.03	0.000096	96	1 XtAddConverter
0.03	0.000093	93	1 XtCreateWindow
0.03	0.000093	93	1 XFillRectangle
0.03	0.000089	89	1 XCreateGC
0.03	0.000089	89	1 XShapeCombineMask
0.02	0.000087	87	1 XclearWindow
0.02	0.000086	86	1 XFreePixmap
<hr/>			
100.00	0.353739	2261	total

在清单 4.4 中，库函数 XSetWMProtocols、hypot、XQueryPointer 分别占用了在库中所花总时间的 18.65%、17.19% 和 12.06%。消耗时间第二多的函数 hypot，其调用次数为 702 次，而消耗时间第一多的函数 XSetWMProtocols，其调用次数仅为 1 次。除非我们的应用程序能够完全删去对 XSetWMProtocols 的调用，否则不管它需要多少时间，我们都会被这个时间所制约。我们最好将注意力转向 hypot。这个函数的每次调用都是相对轻量级的，因此，如果我们能减少它的调用次数，就有可能加快该应用程序的速度。假如 xeyes 应用程序是一个性能问题，那么 hypot 也许是第一个要被调查的函数。起初，我们想确定 hypot 是做什么的，但是又不清楚它记录在什么地方。那么是否有可能，我们可以找出 hypot 属于哪个库，然后阅读这个库的文档。本例中，我们不必先去找库，因为 hypot 函数有说明页。运行 man hypot 就可以告诉我们，hypot 函数计算两点之间的距离（斜边），它是数学库 libm 的一部分。但是，库中函数有可能是没有说明页的，因此，我们需要确定这些没有说明页的函数是属于哪个库的。遗憾的是，ltrace 不会明显地表示一个函数是来自于哪个库的。要指出这一点，我们必须使用 Linux 工具 ldd 和 objdump。首先，ldd 用于显示一个动态链接的应用程序使用了哪些库。其次，objdump 用于在每个库中查找给定的函数。在清单 4.5 中，我们用 ldd 来查看 xeyes 应用程序使用了哪些库。

清单 4.5

```
[ezolt@localhost manuscript]$ ldd /usr/X11R6/bin/xeyes
linux-gate.so.1 => (0x00ed3000)
libXmu.so.6 => /usr/X11R6/lib/libXmu.so.6 (0x00cd4000)
libXt.so.6 => /usr/X11R6/lib/libXt.so.6 (0x00a17000)
```

```
libSM.so.6 => /usr/X11R6/lib/libSM.so.6 (0x00368000)
libICE.so.6 => /usr/X11R6/lib/libICE.so.6 (0x0034f000)
libXext.so.6 => /usr/X11R6/lib/libXext.so.6 (0x0032c000)
libX11.so.6 => /usr/X11R6/lib/libX11.so.6 (0x00262000)
libm.so.6 => /lib/tls/libm.so.6 (0x00237000)
libc.so.6 => /lib/tls/libc.so.6 (0x0011a000)
libdl.so.2 => /lib/libdl.so.2 (0x0025c000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00101000)
```

现在 ldd 命令已经显示了 xeyes 使用的库，我们可以用 objdump 命令来找出这些函数来自哪个库。在清单 4.6 中，我们在 xeyes 链接的每个库中查找 hypot 符号。objdump 的选项 -T 列出了库依赖或提供的所有符号（主要是函数）。通过使用 fgrep 查看带有 .text 的输出行，我们可以发现是哪些库输出 hypot 函数。本例中，libm 库是唯一含有 hypot 函数的库。

清单 4.6

```
[/tmp]$ objdump -T /usr/X11R6/lib/libXmu.so.6 | fgrep ".text" | grep "hypot"
[ /tmp]$ objdump -T /usr/X11R6/lib/libXt.so.6 | fgrep ".text" | grep "hypot"
[ /tmp]$ objdump -T /usr/X11R6/lib/libSM.so.6 | fgrep ".text" | grep "hypot"
[ /tmp]$ objdump -T /usr/X11R6/lib/libICE.so.6 | fgrep ".text" | grep "hypot"
[ /tmp]$ objdump -T /usr/X11R6/lib/libXext.so.6 | fgrep ".text" | grep "hypot"
[ /tmp]$ objdump -T /usr/X11R6/lib/libX11.so.6 | fgrep ".text" | grep "hypot"
[ /tmp]$ objdump -T /lib/tls/libm.so.6 | fgrep ".text" | grep "hypot"
00247520 w DF .text 000000a9 GLIBC_2.0 hypotf
0024e810 w DF .text 00000097 GLIBC_2.0 hypotl
002407c0 w DF .text 00000097 GLIBC_2.0 hypot
[ /tmp]$ objdump -T /lib/tls/libc.so.6 | fgrep ".text" | grep "hypot"
[ /tmp]$ objdump -T /lib/libdl.so.2 | fgrep ".text" | grep "hypot"
[ /tmp]$ objdump -T /lib/ld-linux.so.2 | fgrep ".text" | grep "hypot"
```

下一步应该浏览 xeyes 源代码找出 hypot 是在哪里被调用的，如果可能的话，减少其调用的次数。或者还有一种方法，查看 hypot 的源，并尝试优化库的源代码。

通过调查哪个库调用需要很长的时间来完成，ltrace 使你能确定应用程序的每个库调用的成本。

4.2.4 ps (进程状态)

ps 是极好的跟踪运行进程的命令。

它给出正在运行进程的详细的静态和动态统计信息。ps 提供的静态信息包括命令名和 PID，动态信息包括内存和 CPU 的当前使用情况。

4.2.4.1 CPU 性能相关的选项

ps 有许多不同的选项，能检索正在运行中的应用程序的各种统计信息。下面的调用给出了与 CPU 性能最相关的选项，并将显示给定 PID 的信息：

```
ps [-o etime,time,pcpu,command] [-u user] [-U user] [PID]
```

ps 命令是出现最早的、功能丰富的用于提取性能信息的命令之一，因此，绝大多数人都会选择使用它。若只看全部功能的一个子集，它就更易于管理。表 4-7 包含了与 CPU 性能最相关的选项。

表 4-7 ps 命令行选项

选 项	说 明	
-o <statistic>	该项允许你明确规定想要跟踪的进程统计信息。不同的统计项由一个没有空格的、用逗号分隔的列表指定	
	etime	统计信息：经过时间是指从程序开始执行起耗费的总的时间
	time	统计信息：CPU 时间是指进程运行于 CPU 所花费的系统时间加上用户时间
	pcpu	统计信息：进程当前消耗的 CPU 的百分比
	command	统计信息：命令名
		-A 显示所有进程的统计信息
		-u user 显示指定有效用户 ID 的所有进程的统计信息
	-U user	显示指定用户 ID 的所有进程的统计信息

除了 CPU 统计信息之外，ps 还提供了数量庞大的各种统计信息，其中的一些，比如进程的内存使用情况，将在后续章节中讨论。

4.2.4.2 用法示例

这个例子是一个测试程序，它使用了 88% 的 CPU，运行了 6 秒，但是消耗的 CPU 时间只有 5 秒：

```
[ezolt@wintermute tmp]$ ps -o etime,time,pcpu,cmd 10882
ELAPSED      TIME %CPU CMD
00:06 00:00:05 88.0 ./burn
```

清单 4.7 中，我们没有调查具体进程的 CPU 性能，而是查看了特定用户运行的全部进程。这可能会揭示特定用户消耗的资源量的信息。本例中，我们查看用户 netdump 运行的所有进程。幸运的是，netdump 是一个很单调的用户，它只运行了 bash 和 top，其中，bash 不占用任何 CPU，而 top 只占用了 0.5% 的 CPU。

与 time 不同，ps 使我们能监控当前正在运行的进程的信息。对于运行时间较长的工作，你可以用 ps 定期检查进程的状态（而不是在程序已经执行完后，用它来提供该程序执行的统计信息）。

清单 4.7

```
[/tmp]$ ps -o time,pcpu,command -u netdump
      TIME %CPU COMMAND
00:00:00  0.0 -bash
00:00:00  0.5 top
```

4.2.5 ld.so (动态加载器)

执行一个动态链接应用程序时，首先运行的是 Linux 加载器 ld.so。ld.so 加载该应用程序所有的库，并将它使用的符号与库提供的函数关联起来。因为不同的库最初被链接到内存中的不同位置，这些位置还可能是重叠的，链接器需要对所有的符号进行排序，以确保每个符号都位于内存中的不同位置。一个符号从一个虚拟地址移动到另一个虚拟地址，就被称为重定位（relocation）。加载器做这项工作是需要时间的，如果它完全不用去做那就更好了。预链接应用程序的目标就是通过重排整个系统的系统库来完成这项工作，以保证它们不会相互重叠。需要进行大量重定位的应用程序可能没有被预链接过。

Linux 加载器的运行不需要用户进行任何干预，只需执行一个动态程序即可，它是自动运行的。虽然加载器的执行对用户来说是隐藏的，但是它的执行仍然要花时间，这就有可能会延长应用程序的启动时间。当你要了解加载器的统计信息时，加载器展示的是其工作量，以便你能弄清楚它是否是瓶颈。

4.2.5.1 CPU 性能相关的选项

对使用共享库的每一个 Linux 应用程序来说，ld 命令的运行是不可见的。通过设置合适的环境变量，我们可以要求它显示其执行的信息。下面的调用会影响 ld 的执行：

```
env LD_DEBUG=statistics,help LD_DEBUG_OUTPUT=filename <command>
```

加载器的调试能力完全用环境变量控制。表 4-8 是对这些变量的说明。

表 4-8 ld 环境变量

选 项	说 明
LD_DEBUG=statistics	启动显示 ld 的统计信息
LD_DEBUG=help	显示可用的调试统计信息

表 4-9 解释了一些 ld.so 可以提供的统计信息。时间为时钟周期，要将它转换为墙钟时间，就必须除以处理器的时钟速度。（该信息见于 cat/proc/cpuinfo。）

ld 能提供的信息有助于确定应用程序开始执行之前，设置动态库花费了多少时间。

表 4-9 与 CPU 相关的 ld.so 输出

选 项	说 明
<code>total startup time in dynamic loader</code>	应用程序开始执行之前，加载所花的时间（以时钟周期为单位）
<code>time needed for relocation</code>	符号重定位所花的总时间（以时钟周期为单位）
<code>number of relocations</code>	应用程序开始执行前，已完成的新的重定位计算的数量
<code>number of relocations from cache</code>	在应用程序开始执行之前，预先计算并使用的重定位的数量
<code>number of relative relocations</code>	相对重定位的数量
<code>time needed to load objects</code>	加载应用程序使用的全部库所需的时间
<code>final number of relocations</code>	一个应用程序运行期间重定位的总数（包括那些由 <code>dlopen</code> 发起的）
<code>final number of relocations from cache</code>	预计算的重定位的数量

4.2.5.2 用法示例

在清单 4.8 中，我们运行一个应用程序，并用 ld 调试定义的环境变量。输出的统计信息保存在 lddebug 文件中。请注意，加载器显示了两组不同的统计数据。第一个显示了启动时发生的全部重定位，后一个则显示的是程序关闭后所有的统计信息。如果应用程序使用了函数这些可以是不同的值，比如使用函数 `dlopen`，它允许共享库在程序开始执行后映射到该应用程序。本例中，我们看到加载器时间的 83% 都用在定位上。如果该应用程序已经预链接过，那么这个时间会下降到接近于零。

清单 4.8

```
[ezolt@wintermute ezolt]$ env LD_DEBUG=statistics LD_DEBUG_OUTPUT=lddebug gcalctool
[ezolt@wintermute ezolt]$ cat lddebug.2647
2647:
2647:     runtime linker statistics:
2647:         total startup time in dynamic loader: 40820767 clock cycles
2647:             time needed for relocation: 33896920 clock cycles (83.0%)
2647:                 number of relocations: 2821
2647:                 number of relocations from cache: 2284
2647:                 number of relative relocations: 27717
2647:             time needed to load objects: 6421031 clock cycles (15.7%)
2647:
2647:     runtime linker statistics:
2647:             final number of relocations: 6693
2647:             final number of relocations from cache: 2284
```

如果 ld 被确定为延长启动时间的原因，那么可以通过减少应用程序依赖的库的数量或者是在系统上运行 prelink 的方法来缩短启动时间。

4.2.6 gprof

剖析 Linux 应用程序的一种强有力的方法是使用 gprof 分析命令。gprof 可以展示应用程序的调用图，并采样该应用程序的时间都花在了哪里。gprof 的工作方式是，首先编译你的应用程序，然后运行该应用程序生成一个采样文件。gprof 是非常强大的，但是它需要应用源程序，并且增加了编译开销。尽管 gprof 可以确定函数被调用的精确次数，以及函数所花的大致时间，但是其编译将有可能改变应用程序的时间特性，延缓程序的执行。

4.2.6.1 CPU 性能相关的选项

要用 gprof 剖析一个应用程序，你必须访问应用程序源。然后还需用如下所示的 gcc 命令编译该程序：

```
gcc -gp -g3 -o app app.c
```

首先，你必须用 gcc 的 -gp 选项来编译应用程序，开启剖析功能。须注意不要与可执行文件剥离，如果用 -g3 选项编译开启符号会更加有用。符号信息对使用 gprof 的源注释特性是必须的。当你运行被编译过的应用程序时，会生成一个输出文件。然后你可以用 gprof 命令来显示结果。gprof 命令的调用如下：

```
gprof [-p --flat-profile -q --graph --brief -A --annotated-source] app
```

表 4-10 说明的选项指定了 gprof 显示的信息。

表 4-10 gprof 命令行选项

选 项	说 明
--brief	简化 gprof 的输出。默认情况下，gprof 输出全部的性能信息，并用图例解释每个指标的含义。该项删除了图例
-p or --flat-profile	显示应用程序中每个函数花费的总时间和其调用次数
-q or --graph	打印出已剖析的应用程序的调用图。其显示了程序中的函数是如何相互调用的，每个函数所花的时间，以及子函数所花的时间
-A or --annotated-source	在原始源代码的下面显示剖析信息

对一个特定的剖析来说，并不是所有的输出统计信息都是可以得到的。哪个输出统计信息是可得的取决于应用程序是如何为了剖析而被编译的。

4.2.6.2 用法示例

用 gprof 剖析一个应用程序时，第一步是用剖析信息编译该程序。编译器（gcc）将剖析信息插入到应用程序中，该程序运行时，会被保存到名为 gmon.out 的文件里。burn 测试程序相当简单。它清除了大范围的内存，然后调用了两个函数：a() 和 b()，这两个函数都要访问此内存区域。函数 a() 访问内存的频繁程度是函数 b() 的 10 倍。

首先，我们编译该应用程序：

```
[ezolt@wintermute test_app]$ gcc -pg -g3 -o burn_gprof burn.c
```

运行程序后，我们可以分析输出，如清单 4.9 所示。

清单 4.9

```
[ezolt@wintermute test_app]$ gprof --brief -p ./burn_gprof
Flat profile:
```

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
91.01	5.06	5.06	1	5.06	5.06	a
8.99	5.56	0.50	1	0.50	0.50	b

在清单 4.9 中，你可以看到 gprof 呈现的是我们已经知道的关于该应用程序的情况。程序有两个函数 a() 和 b()。每个函数都调用了一次，a() 完成的时间（91%）是 b() 完成时间（8.99%）的 10 倍。函数 a() 花费的时间为 5.06 秒，函数 b() 花费的时间为 0.5 秒。

清单 4.10 给出了测试程序的调用图。输出中列出的 <spontaneous> 注释的含义如下：尽管 gprof 没有记录 main() 的任何样本，但它推断出 main() 必然已经运行，因为函数 a() 和 b() 都有采样，而 main 是代码中唯一调用它们的函数。gprof 没有记录 main() 的任何样本，很可能是因为这个函数太短了。

清单 4.10

```
[ezolt@wintermute test_app]$ gprof --brief -q ./burn_gprof
Call graph
granularity: each sample hit covers 4 byte(s) for 0.18% of 5.56 seconds

index % time    self  children   called      name
                                         <spontaneous>
[1]  100.0    0.00    5.56
                5.06    0.00     1/1        a [2]
                0.50    0.00     1/1        b [3]
-----
                5.06    0.00     1/1        main [1]
[2]   91.0    5.06    0.00      1        a [2]
-----
                0.50    0.00     1/1        main [1]
[3]    9.0    0.50    0.00      1        b [3]
-----

Index by function name
[2] a
[3] b
```

最后，gprof可以对源代码进行注释，以展示每个函数调用的频率。请注意，清单4.11没有显示函数消耗的时间；取而代之，它显示的是函数被调用的次数。在gprof前面的例子中，a()实际时长是b()的10倍，因此优化时需要多加小心。不要认为被多次调用的函数实际上使用CPU的时间也多，而被调用次数少的函数消耗的CPU时间必然也少。

清单4.11

```
[ezolt@wintermute test_app]$ gprof -A burn_gprof
*** File /usr/src/perf/process_specific/test_app/burn.c:
#include <string.h>

#define ITER 10000
#define SIZE 10000000
#define STRIDE 10000
char test[SIZE];

void a(void)
i -> {
    int i=0,j=0;
    for (j=0;j<10*ITER ; j++)
        for (i=0;i<SIZE;i=i+STRIDE)
    {
        test[i]++;
    }
}

void b(void)
i -> {
    int i=0,j=0;
    for (j=0;j<ITER; j++)
        for (i=0;i<SIZE;i=i+STRIDE)
    {
        test[i]++;
    }
}

main()
##### -> {

/* Arbitrary value*/
memset(test, 42, SIZE);
```

```

    a();
    b();
}

```

Top 10 Lines:

Line	Count
10	1
20	1

Execution Summary:

```

3 Executable lines in this file
3 Lines executed
100.00 Percent of the file executed

2 Total number of line executions
0.67 Average executions per line

```

gprof 提供了一个很好的总结，可以显示应用程序中的函数以及源代码行运行的次数以及它们所花费的时间。

4.2.7 oprofile (II)

第 2 章中讨论过，你可以使用 oprofile 跟踪系统或应用程序中不同事件的位置。与 gprof 相比，oprofile 是一个低开销的工具。与 gprof 不同，它在使用前不需要对应用程序进行二次编译。oprofile 也可以测量 gprof 不支持的事件。目前，oprofile 只支持那些 gprof 用内核补丁可以生成的调用图，而 gprof 能够在所有的 Linux 内核上运行。

4.2.7.1 CPU 性能相关的选项

2.2.8 节讨论的 oprofile 涉及的是如何用 oprofile 开始进行剖析。本小节介绍的则是 oprofile 用于分析进程级采样结果的部分。

oprofile 有一系列工具来显示已收集的样本。第一个工具 oreport 显示的是样本在可执行文件和库的函数中分布情况。其调用形式如下：

```
opreport [-d --details -f --long-filenames -l --symbols -l] application
```

表 4-11 解释了几个命令，它们能够修改由 opreport 提供的信息的等级。

第二个你能用来提取性能样本信息的命令是 opannotate。opannotate 可以将样本对应到具体的源代码行或汇编指令。其调用形式如下：

```
opannotate [-a --assembly] [-s --source] application
```

表 4-11 oreport 命令行选项

选 项	说 明
-d or --details	显示了全部已采集样本的指令级分类
-f or --long-filenames	显示了被分析应用程序的完整路径名
-l or --symbols	显示了应用程序的样本是如何分布到它的符号的。这一项使你能看到哪些函数拥有最多的样本

表 4-12 说明了该调用的选项，它们能让你指定 opannotate 提供的确切信息。这里有一个忠告：由于处理器硬件计数器在源代码行和指令级上的限制，样本可能不会准确对应到引发它的那一行。不过，它们会很接近实际的事件。

表 4-12 opannotate 命令行选项

选 项	说 明
-s or --source --	在应用程序源代码的旁边显示已收集样本
-a or --assembly	在应用程序的汇编代码的旁边显示已收集样本
-s and -a	如果同时指定了-s 和 -a，那么 opannotate 将把样本与源代码以及汇编代码放一起

使用 opannotate 和 oreport 时，指明应用程序的完整路径名总是最好的做法。如果不这样做，你可能会接收到一条神秘的错误信息（如果 oprofile 无法发现应用程序的样本）。默认情况下，在显示结果时，oprofile 只显示可执行文件名，这会与系统中多个有相同名称的可执行文件或库相混淆。因此，总是指定 -f 选项就可以让 oreport 显示应用程序的完整路径。

oprofile 还可以提供一个命令 opgprof 用于输出由 oprofile 收集的样本，其输出形式能被 gprof 理解。该命令的调用方式如下：

```
opgprof application
```

该命令获取 application 的样本，并生成与 gprof 兼容的文件。之后，你就可以用 gprof 命令来查看该文件了。

4.2.7.2 用法示例

鉴于我们已经在 2.2.8 节中了解过 oprofile，这里的例子向你展示的将是如何利用 oprofile 来跟踪源代码特定行的性能问题。本小节假设你已经用 opcontrol 命令启动了剖析。下一步就是运行有性能问题的程序。本例中，我们使用的是 burn 程序，就是在 gprof 示例中用过的程序。我们按如下方式启动测试程序：

```
[ezolt@wintermute tmp]$ ./burn
```

程序完成后，我们必须将 oprofile 的缓冲区转储到硬盘，否则样本对 oreport 将是不可用的。用如下命令完成这一步：

```
[ezolt@wintermute tmp]$ sudo opcontrol -d
```

接着，在清单 4.12 中，我们要求 oreport 告诉我们与测试程序 /tmp/burn 相关的样本。这能让我们对该应用程序消耗的周期数有个总体映像。本例中，我们看到应用程序有 9 939 个样本。如果我们深入 oprofile 工具，我们将了解这些样本是如何在 burn 程序中分布的。

清单 4.12

```
[ezolt@wintermute tmp]$ oreport -f /tmp/burn
CPU: PIII, speed 467.731 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a
unit mask of 0x00 (No unit mask) count 233865
9939 100.0000 /tmp/burn
```

之后，在清单 4.13 中，我们想了解所有样本是属于 burn 程序中的哪些函数。由于我们使用了 CPU_CLK_UNHALTED 事件，这大致对应于每个函数所花费的相对时间。通过查看输出，我们可以看到 91% 的时间花在了函数 a() 上，9% 的时间花在了函数 b() 上。

清单 4.13

```
[ezolt@wintermute tmp]$ oreport -l /tmp/burn
CPU: PIII, speed 467.731 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a
unit mask of 0x00 (No unit mask) count 233865
vma      samples %          symbol name
08048348 9033   90.9118    a
0804839e 903     9.0882    b
```

在清单 4.14 中，我们要求 oreport 展示哪些虚拟地址有对应的样本。本例中，看上去似乎位于地址 0x0804838a 的指令拥有 75% 的样本。但是，现在还不清楚这条指令是做什么的，或者为什么有这么多样本。

清单 4.14

```
[ezolt@wintermute tmp]$ oreport -d /tmp/burn
CPU: PIII, speed 467.731 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a
unit mask of 0x00 (No unit mask) count 233865
vma      samples %          symbol name
08048348 9033   90.9118    a
08048363 4       0.0443
08048375 431     4.7714
0804837c 271     3.0001
0804837e 1       0.0111
08048380 422     4.6718
0804838a 6786    75.1245
```

08048393	1114	12.3326
08048395	4	0.0443
0804839e	903	9.0882 b
080483cb	38	4.2082
080483d2	19	2.1041
080483d6	50	5.5371
080483e0	697	77.1872
080483e9	99	10.9635

通常，对我们更加有用的是知道使用所有 CPU 时间的源代码行，而不是使用它的指令的虚拟地址。找出一条特定指令与源代码行之间的对应关系并不总是容易的事儿。因此，在清单 4.15 中，我们要求 opannotate 来做这项困难的工作，向我们展示相对于原始源代码的样本（而并非指令的虚拟地址）。

清单 4.15

```
[ezolt@wintermute tmp]$ opannotate --source /tmp/burn
/*
 * Command line: opannotate --source /tmp/burn
 *
 * Interpretation of command line:
 * Output annotated source file with samples
 * Output all files
 *
 * CPU: PIII, speed 467.731 MHz (estimated)
 * Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with
 a unit mask of 0x00 (No unit mask) count 233865
 */
/*
 * Total samples for file : "/tmp/burn.c"
 *
 * 9936 100.0000
 */

:#include <string.h>
:
:#define ITER 10000
:#define SIZE 10000000
:#define STRIDE 10000
:
:char test[SIZE];
```

```

:
:void a(void)
:{ /* a total: 9033 90.9118 */

: int i=0,j=0;
8 0.0805 : for (j=0;j<10*ITER ; j++)
8603 86.5841 : for (i=0;i<SIZE;i=i+STRIDE)
:
: {
422 4.2472 :     test[i]++;
:
: }
:
: void b(void)
:{ /* b total: 903 9.0882 */
: int i=0,j=0;
: for (j=0;j<ITER; j++)
853 8.5849 : for (i=0;i<SIZE;i=i+STRIDE)
:
: {
50 0.5032 :     test[i]++;
:
: }
:
: main()
: {
:
: /* Arbitrary value*/
: memset(test, 42, SIZE);
: a();
: b();
:
:
```

正如你能在清单 4.15 中看到的一样，opannotate 把大部分样本（86.59%）归因于函数 a() 中的 for 循环。可惜的是，for 循环中这部分的代价并不会很高。对现代处理器来说，整数加上固定数的执行速度是非常快的。因此，oprofile 报告的样本可能被归于错误的源代码行。而下面的代码行 (test[i]++；) 其代价则非常高，因为它要访问内存子系统。这一行才应该是这些样本所对应的。

有些超出 oprofile 控制的原因会导致它对样本的错误对应。首先，处理器并不总是精确中断于导致事件发生的那一行。这可能会让样本被归于事件源头附近的指令，而不是引发事件的那条指令。其次，当源代码被编译时，编译器常常为了让执行更有效率而重排指令。编译器完成优化后，代码也许就不是按照其编写的顺序来执行。不同的源代码行可能被重

排和组合。其结果就是，特定的指令也许是多个源代码行的结果，或者甚至是编译器生成的中间代码段，而这段中间代码在原始源代码中是不存在的。因此，当编译器优化代码，生成机器指令时，原始源代码行与生成的机器指令之间可能不再有一对一的映射关系。这就使得让oprofile（和调试器）指出每条机器指令究竟对应哪一行源代码变得困难重重，甚至于不可能。不过，oprofile还是试图尽可能的准确，因此，通常你可以看看高样本计数代码行的上下几行，就可以找出真正代价高的那行代码。如果需要，你可以用opannotate来显示确切的汇编指令，以及正在接收所有样本的虚拟地址。这有可能发现汇编指令在做什么，从而手动将它映射回你的原始源代码。oprofile的样本归属并不准确，但它通常是足够接近的。即使存在这些限制，oprofile提供的文件显示了大致的源代码行以供调查，一般来说，这就足够找出应用程序的速度慢在了哪里。

4.2.8 语言：静态（C 和 C++）vs. 动态（Java 和 Mono）

大多数Linux性能工具都支持对静态语言（如C和C++）的分析，本章描述的所有工具都能运用于由这些语言编写的应用程序。工具ltrace、strace和time可运用于由动态语言编写的应用程序，比如Java、Mono、Python和Perl。但是剖析工具gprof和oprofile不能用于这些类型的应用程序。幸运的是，大多数动态语言提供了并不只针对Linux的剖析基础工具来生成相似类型的配置文件。

对Java应用程序而言，如果运行java命令时带上了-Xrunhprof命令行选项，那么-Xrunhprof将对应用程序进行剖析。更多详细信息参见<http://antprof.sourceforge.net/hprof.html>。对Mono应用程序而言，如果mono可执行文件被传递了--profile标志，那么它将剖析该应用程序。更多详细信息参见<http://www.mono-project.com/docs/advanced/performance-tips/>。Perl和Python也有相似的剖析功能，Perl的Devel::DProf的说明见于网址<http://perl.com/pub/a/2004/06/25/profiling.html>，而Python的profiler的说明则见于新网址：<https://docs.python.org/3/library/index.html>。

4.3 本章小结

本章介绍了怎样跟踪单个进程的CPU性能瓶颈。学会了确定一个应用程序消耗的时间是如何分配到Linux内核、系统库，甚至于该应用程序本身的。还学会了怎样找出哪些调用是对内核的，哪些是对系统库的，以及完成它们分别花了多少时间。最后，学习了如何剖析应用程序，确定源代码的哪个特定行消耗了大量的时间。掌握了这些工具之后，就可以启动独占CPU的应用程序，并利用这些工具准确地找出消耗了所有时间的那些函数。

后续章节将研究如何发现那些不受CPU约束的瓶颈。特别是将学习到用于发现诸如饱和磁盘或超载网络的I/O瓶颈的工具。

性能工具：特定进程内存

本章介绍的工具使你能诊断应用程序与内存子系统之间的交互，该子系统由 Linux 内核和 CPU 管理。由于内存子系统的不同层次在性能上有数量级的差异，因此，修复应用程序使其有效地使用内存子系统会对程序性能产生巨大的影响。

阅读本章后，你将能够：

- 确定一个应用程序使用了多少内存 (`ps`, `/proc`)。
- 确定应用程序的哪些函数分配内存 (`memprof`)。
- 用软件模拟 (`kcacheGrind`, `cachegrind`) 和硬件性能计数器 (`oprofile`) 分析应用程序的内存使用情况。
- 确定哪些进程创建和使用了共享内存 (`ipcs`)。

5.1 Linux 内存子系统

在诊断内存性能问题的时候，也许有必要观察应用程序在内存子系统的不同层次上是怎样执行的。在顶层，操作系统决定如何利用交换内存和物理内存。它决定应用程序的哪一块地址空间将被放到物理内存中，即所谓的驻留集。不属于驻留集却又被应用程序使用的其他内存将被交换到磁盘。由应用程序决定要向操作系统请求多少内存，即所谓的虚拟集。应用程序可以通过调用 `malloc` 进行显式分配，也可以通过使用大量的堆栈或库进行隐式分配。应用程序还可以分配被其自身或其他应用程序使用的共享内存。性能工具 `ps` 用于跟踪虚拟集和驻留集的大小。性能工具 `memprof` 用于跟踪应用程序的哪段代码是分配内存

的。工具 ipcs 用于跟踪共享内存的使用情况。

当应用程序使用物理内存时，它首先与 CPU 的高速缓存子系统交互。现代 CPU 有多级高速缓存。最快的高速缓存离 CPU 最近（也称为 L1 或一级高速缓存），其容量也最小。举个例子，假设 CPU 只有两级高速缓存：L1 和 L2。当 CPU 请求一块内存时，处理器会检查该块内存是否已经存在于 L1 高速缓存中。如果处于，CPU 就可以使用。如果不在 L1 高速缓存中，处理器产生一个 L1 高速缓存不命中。然后它会检查 L2 高速缓存，如果数据在 L2 高速缓存中，那么它可以使用。如果数据不在 L2 高速缓存，将产生一个 L2 高速缓存不命中，处理器就必须到物理内存去取回信息。最终，如果处理器从不访问物理内存（因为它会在 L1 或者甚至 L2 高速缓存中发现数据）将是最佳情况。明智地使用高速缓存，例如重新排列应用程序的数据结构以及减少代码量等方法，有可能减少高速缓存不命中的次数并提高性能。cachegrind 和 oprofile 是很好的工具，用于发现应用程序对高速缓存的使用情况的信息，以及哪些函数和数据结构导致了高速缓存不命中。

5.2 内存性能工具

本节讨论各种内存性能工具，它们使你能检查一个给定的应用程序是如何使用内存子系统的，包括进程使用的内存总量和不同的内存类型，内存是在哪里分配的，以及进程是如何有效使用处理器的高速缓存的。

5.2.1 ps (II)

对跟踪进程的动态内存使用情况而言，ps 是一个极好的命令。除了已经介绍过的 CPU 统计数据，ps 还能提供关于应用程序使用内存的总量以及内存使用情况对系统影响的详细信息。

5.2.1.1 内存性能相关的选项

ps 有许多不同的选项，可以获取一个正在运行的应用程序各种各样的状态统计信息。如同你在前面章节里看到的，ps 能够检索到一个进程消耗 CPU 的情况，但它同时也能够检索到进程使用内存的容量和类型信息。ps 可以用如下命令行调用：

```
ps [-o vsz,rss,tsiz,dsiz,majflt,minflt,pmem,command] <PID>
```

表 5-1 解释了给定 PID 情况下，ps 显示的不同类型的内存统计信息。

如前所述，在如何选择统计数据要显示哪些 PID 时，ps 是很灵活的。ps -help 提供了信息以说明如何指定不同的 PID 组。

表 5-1 ps 命令行选项

选 项	说 明
-o <statistic>	允许你指定想要跟踪的确定的进程统计信息。不同的统计数据由列表给出，列表项用逗号隔开，且中间没有空格
vsz	统计数据：虚拟集大小是指应用程序使用的虚拟内存的容量。由于 Linux 只在应用程序试图使用物理内存时才分配它，因此，该项数值可能会比应用程序使用的物理内存量大很多
rss	统计数据：驻留集大小是指应用程序当前使用的物理内存量
tsiz	统计数据：文本大小是指程序代码的虚拟大小。再强调一次，这不是实际大小，而是虚拟大小；但是，该项数值清晰地表明了程序的大小
dsiz	统计数据：数据大小是指程序数据使用量的虚拟大小。该项数值清晰地表明了应用程序的数据结构和堆栈的大小
majflt	统计数据：主故障是指使得 Linux 代表进程从磁盘读取页面的缺页故障的数量。这种故障可能发生的情况是：当进程访问的一块数据或指令仍留在磁盘上时，Linux 要为应用程序进行无缝加载
minflt	统计数据：次故障是指 Linux 不用诉诸磁盘读取就可以解决的故障数量。如果应用程序涉及一块已经由 Linux 内核分配的内存，就有可能发生这种情况。这种情况不需要访问磁盘，因为内核只需选择一块空闲内存并将其分配给应用程序即可
pmp	统计数据：进程消耗的系统内存百分比
command	统计数据：命令名

5.2.1.2 用法示例

清单 5.1 展示了在系统上运行的测试应用程序 burn。我们要求 ps 给出进程的内存统计信息。

清单 5.1

```
[ezolt@wintermute tmp]$ ps -o vsz,rss,tsiz,dsiz,majflt,minflt,cmd 10882
VSZ  RSS TSIZ DSIZ MAJFLT MINFLT CMD
11124 10004  1 11122    66   2465 ./burn
```

如清单 5.1 所示，应用程序 burn 的文本大小很小（1KB），但是其数据大小却很大（11 122KB）。相对总的虚拟大小（11 124KB）来说，进程的驻留集略小一点（10 004KB），驻留集表示的是进程实际使用的物理内存总量。此外，burn 产生的大多数故障都是次故障，所以，多数内存故障是由内存分配导致的，而不是由从磁盘的程序映像加载大量的文本或数据导致的。

5.2.2 /proc/<PID>

Linux 内核提供了一个虚拟文件系统，使你能提取在系统上运行的进程的信息。由 /proc 文件系统提供的信息通常仅被如 ps 之类的性能工具用于从内核提取性能数据。尽管一般不需要深入挖掘 /proc 中的文件，但是它确实能提供其他性能工具所无法检索到的一些信

息。除了许多其他统计数据之外，/proc 还提供了进程的内存使用信息和库映射信息。

5.2.2.1 内存性能相关的选项

/proc 的接口很简单。/proc 提供了许多虚拟文件，可以用 cat 来提取它们的信息。系统中每一个运行的 PID 在 /proc 下都有一个子目录，这个子目录含有一系列文件，其中包含的是关于该 PID 的信息。这些文件中，status 给出的是给定进程 PID 的状态信息，其检索命令如下：

```
cat /proc/<PID>/status
```

表 5-2 对 status 文件显示的内存统计信息进行了解释。

表 5-2 /proc/<PID>/status 字段说明

选 项	说 明
VmSize	进程的虚拟集大小，是应用程序使用的虚拟内存量。由于 Linux 只在应用程序试图使用物理内存时才进行分配，因此，这个数字可能会比应用程序实际使用的物理内存容量大很多。该项与 ps 提供的 vsz 参数相同
VmLck	被进程锁定的内存量。被锁定的内存不能交换到磁盘
VmRSS	驻留集大小或应用程序当前使用的物理内存量。它与 ps 提供的 rss 统计数据相同
VmData	数据大小或程序使用数据量的虚拟大小。与 ps 的 dsiz 统计数据不同，该项不包含堆栈信息
VmStk	进程的堆栈的大小
VmExe	程序的可执行内存的虚拟大小。它不包含进程使用的库
VmLib	进程使用的库的大小

<PID> 目录下的另一个文件是 maps，它提供了关于如何使用进程虚拟地址空间的信息。其检索命令如下所示：

```
cat /proc/<PID>/maps
```

表 5-3 解释了 maps 文件中的字段。

表 5-3 /proc/<PID>/maps 字段说明

选 项	说 明
Address	进程中库映射的地址范围
Permissions	内存区域的权限，其中：r= 读，w= 写，x= 执行，s= 共享，p= 私有（写时复制）
Offset	库 / 应用程序内存映射区域开始处的偏移量
Device	这个特殊文件所在的设备（主设备号和次设备号）
Inode	映射文件的节点号
Pathname	映射到进程的文件的路径名

/proc 提供的信息可以帮助你了解应用程序是如何分配内存的，以及它使用了哪些库。

5.2.2.2 用法示例

清单 5.2 显示的是运行于系统上的 burn 测试程序。首先，我们用 ps 找到 burn 的 PID (4540)。然后，用 /proc 的 status 文件抽取进程的内存统计信息。

清单 5.2

```
[ezolt@wintermute tmp]$ ps aux | grep burn
ezolt    4540  0.4  2.6 11124 10004 pts/0    T    08:26   0:00 ./burn
ezolt    4563  0.0  0.1  1624   464 pts/0     S    08:29   0:00 grep burn

[ezolt@wintermute tmp]$ cat /proc/4540/status
Name: burn
State: T (stopped)
Tgid: 4540
Pid: 4540
PPid: 1514
TracerPid: 0
Uid: 501 501 501 501
Gid: 501 501 501 501
FDSize: 256
Groups: 501 9 502
VmSize: 11124 kB
VmLck: 0 kB
VmRSS: 10004 kB
VmData: 9776 kB
VmStk: 8 kB
VmExe: 4 kB
VmLib: 1312 kB
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
```

如清单 5.2 所示，我们再一次看到应用程序 burn 的文本大小（4KB）和堆栈大小（8KB）很小，而数据大小（9 776KB）很大，库大小（1 312KB）较合理。小的文本大小意味着进程没有太多的可执行代码，而适中的库大小表示它使用库来支持其执行。小的堆栈大小意味着该进程没有调用深度嵌套的函数，或者没有调用使用了大型或多个临时变量的函数。VmLck 的大小为 0KB 说明进程没有锁定内存中的任何页面，使得它们无法交换。VmRSS 大小为 10 004KB 意味着应用程序当前使用了 10 004KB 的物理内存，不过它分配或映射的大小为 VmSize 或 11 124KB。如果应用程序开始使用之前已分配但并非正在使用

的内存，那么 VmRSS 会增加，而 VmSize 会保持不变。

如前所述，应用程序的 VmLib 的大小为非零值，因此程序使用了库。在清单 5.3 中，我们查看进程的 maps 来了解它使用的那些库。

清单 5.3

```
[ezolt@wintermute test_app]$ cat /proc/4540/maps
08048000-08049000 r-xp 00000000 21:03 393730      /tmp/burn
08049000-0804a000 rw-p 00000000 21:03 393730      /tmp/burn
0804a000-089d3000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 21:03 1147263     /lib/ld-2.3.2.so
40015000-40016000 rw-p 00015000 21:03 1147263     /lib/ld-2.3.2.so
4002e000-4002f000 rw-p 00000000 00:00 0
4002f000-40162000 r-xp 00000000 21:03 2031811     /lib/tls/libc-2.3.2.so
40162000-40166000 rw-p 00132000 21:03 2031811     /lib/tls/libc-2.3.2.so
40166000-40168000 rw-p 00000000 00:00 0
bfff0000-c0000000 rwxp fffff000 00:00 0
```

就像你在清单 5.3 看到的，应用程序 burn 使用了两个库：ld 和 libc。libc 文本部分（由权限 r-xp 表示）的范围从 0x4002f0000 到 0x40162000，即大小为 0x133000 或 1 257 472 字节。

libc 数据部分（由权限 rw-p 表示）的范围从 40162000 到 40166000，即大小为 0x4000 或 16 384 字节。libc 的文本部分大于 ld 的文本部分，后者的大小为 0x15000 或 86 016 字节。libc 的数据部分也大于 ld 的数据部分，后者的大小为 0x1000 或 4 096 字节。libc 是 burn 链接的大库。

/proc 被证明是从内核直接提取性能统计信息的有效途径。由于统计信息是基于文本的，因此可以使用标准的 Linux 工具来访问它们。

5.2.3 memprof

memprof 是一种图形化的内存使用情况剖析工具。它展示了程序在运行时是如何分配内存的。memprof 显示了应用程序消耗的内存总量，以及哪些函数应对这些内存使用量负责。此外，memprof 还可以显示哪些代码路径要对内存使用量负责。比如，如果函数 foo() 不分配内存，但是调用函数 bar() 要分配大量的内存，那么 memprof 就会向你显示 foo() 自身使用的内存量以及 foo() 调用的全部函数。应用程序运行时，memprof 会动态更新这些信息。

5.2.3.1 内存性能相关的选项

memprof 是一个图形化的应用程序，但是也有一些命令行选项来调整其执行。memprof 用如下命令调用：

```
memprof [--follow-fork] [--follow-exec] application
```

memprof 剖析给定的“application”，并为其内存使用情况创建一个图形化输出。虽然 memprof 可以在任何应用程序上运行，但是如果它依赖的应用程序和库使用调试符号编译，那么它就能提供更多的信息。

表 5-4 说明了控制 memprof 行为的选项，条件是 memprof 监控的应用程序调用了 fork 或 exec。这通常发生在应用程序启动一个新进程或执行一条新命令的时候。

表 5-4 memprof 命令行选项

选 项	说 明
--follow-fork	使得 memprof 为新 fork 进程启动一个新窗口
--follow-exec	在应用程序调用 exec 后，使得 memprof 继续剖析这个程序

一旦被调用，memprof 会创建一个带有一系列菜单和选项的窗口，使你能选择要剖析的应用程序。

5.2.3.2 用法示例

假设我有如清单 5.4 所示的示例代码，并且我想要剖析这段代码。在这个称为 memory_eater 的应用程序中，函数 foo() 不分配内存，但是它调用的函数 bar() 却要分配内存。

清单 5.4

```
#include <stdlib.h>
void bar(void)
{
    malloc(10000);
}

void foo(void)
{
    int i;
    for (i=0; i<100;i++)
        bar();
}

int main()
{
    foo();
    while(1);
}
```

用 -g3 标志编译该应用程序后（这样应用程序就包含了符号），我们使用 memprof 来剖析该应用程序：

```
[ezolt@localhost example]$ memprof ./memory_eater memintercept (3965):
_MEMPROF_SOCKET = /tmp/memprof.Bm1AKu memintercept (3965): New process,
operation = NEW, old_pid = 0
```

memprof 创建如图 5-1 所示的应用程序窗口。如同你看到的，它不仅给出了应用程序 memory_eater 的内存使用情况，还显示了一系列的按钮和菜单使你能对分析进行控制。

如果你点击 Profile 按钮，memprof 会显示对应用程序的内存分析。图 5-2 中的第一个信息框显示的是每个函数消耗的内存量（用“Self”表示），以及该函数及其子函数消耗的内存总量（用“Total”表示）。和预期的一样，函数 foo() 不分配任何内存，因此它的 Self 值为 0，但它的 Total 值为 100 000，这是因为它调用的函数要分配内存。

当你点击上面信息框中不同的函数时，Children 和 Callers 信息框会发生变化。这样你就可以看到应用程序的哪些函数在使用内存。

memprof 提供了一种以图形方式遍历大量的关于内存分配数据的途径。它给出了一种简单的方法来确定给定函数及其调用的函数的内存分配情况。

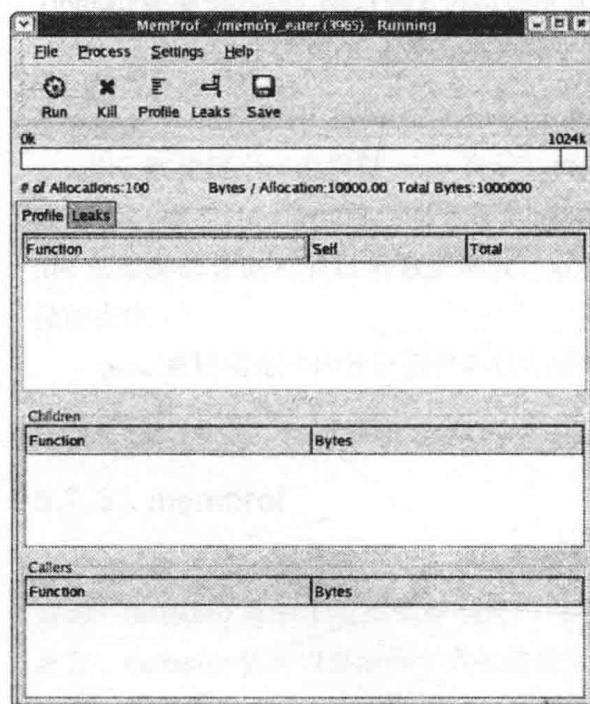


图 5-1

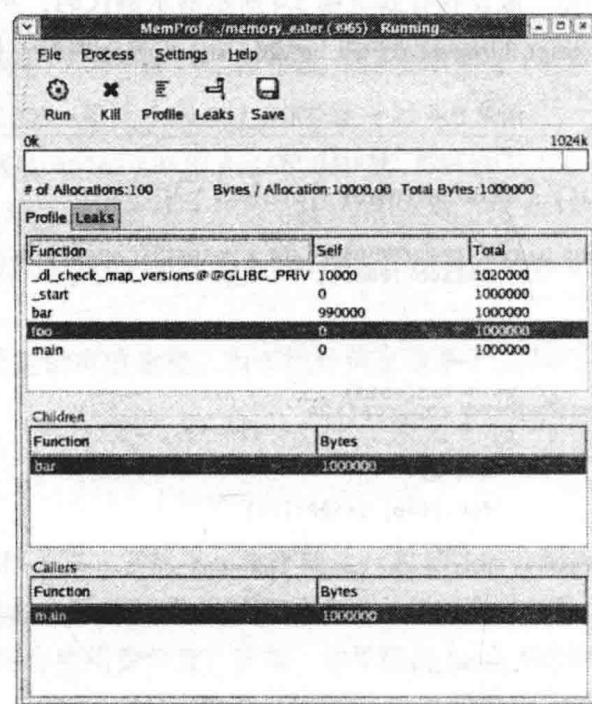


图 5-2

5.2.4 valgrind (cachegrind)

valgrind 是一个强大的工具，使你能调试棘手的内存管理错误。虽然 valgrind 主要是一个开发者工具，但它也有一个“界面”能显示处理器的高速缓存使用情况。valgrind 模拟当

前的处理器，并在这个虚拟处理器上运行应用程序，同时跟踪内存使用情况。它还能模拟处理器高速缓存，并确定程序在哪里有指令和数据高速缓存的命中或缺失。

虽然 valgrind 很有用，但是它的高速缓存统计信息却是不准确的（因为 valgrind 只是处理器的模拟，而不是一个真实的硬件）。valgrind 不计算通常由对 Linux 内核的系统调用导致的高速缓存缺失，也不计算由于上下文切换而发生的高速缓存缺失。此外，valgrind 运行应用程序的速度比本机执行程序的速度慢得多。但是，valgrind 提供了与应用程序高速缓存使用情况最接近的数据。valgrind 可以运行于任何可执行文件。如果程序已经用符号编译过了（在编译时把 -g3 传递给 gcc），它还是能够准确地找出要对高速缓存的使用情况负责的代码行。

5.2.4.1 内存性能相关的选项

当使用 valgrind 分析一个特定应用程序的高速缓存使用情况时，要经历两个阶段：收集和注释。收集阶段从如下命令行开始：

```
valgrind --skin=cachegrind application
```

valgrind 是一个灵活的工具，它有几种不同的“界面”使其可以执行不同类型的分析。在收集阶段，valgrind 用 cachegrind 界面来收集关于高速缓存使用情况的信息。上述命令行中的 application 表示的是要剖析的应用程序。收集阶段在屏幕上显示的是概要信息，但它同时也在名为 cachegrind.out.pid 的文件中保存了更加详细的统计数据，文件名中的 pid 是其运行时被剖析应用程序的 PID。当收集阶段完成后，使用命令 cg_annotate 把高速缓存使用情况映射回应用程序源代码。cg_annotate 调用方式如下：

```
cg_annotate --pid [--auto=yes|no]
```

cg_annotate 获取 valgrind 生成的信息，并用它来注释被剖析的应用程序。选项 --pid 是必须的，因为 pid 是你有兴趣进行剖析的对象的 PID。默认情况下，cg_annotate 只显示函数级的高速缓存使用情况。如果你的设置为 --auto=yes，那么就会在源代码级显示高速缓存的使用情况。

5.2.4.2 用法示例

本例展示了在一个简单应用程序上运行的 valgrind (v2.0)。该应用程序清除一大块内存区域，然后调用两个函数：a() 和 b()，这两个函数都要访问这个内存区域。函数 a() 访问内存的次数是函数 b() 访问次数的 10 倍。

首先，如清单 5.5 所示，我们用 cachegrind 界面在应用程序上运行 valgrind。

清单 5.5

```
[ezolt@wintermute test_app]$ valgrind --skin=cachegrind ./burn
==25571== Cachegrind, an I1/D1/L2 cache profiler for x86-linux.
==25571== Copyright (C) 2002-2003, and GNU GPL'd, by Nicholas Nethercote.
```

```

==25571== Using valgrind-2.0.0, a program supervision framework for x86-linux.
==25571== Copyright (C) 2000-2003, and GNU GPL'd, by Julian Seward.
==25571== Estimated CPU clock rate is 468 MHz
==25571== For more details, rerun with: -v
==25571==
==25571==

==25571== I refs:      11,317,111
==25571== I1 misses:      215
==25571== L2i misses:      214
==25571== I1 miss rate:    0.0%
==25571== L2i miss rate:    0.0%
==25571==

==25571== D refs:      6,908,012 (4,405,958 rd + 2,502,054 wr)
==25571== D1 misses:      1,412,821 (1,100,287 rd + 312,534 wr)
==25571== L2d misses:      313,810 ( 1,276 rd + 312,534 wr)
==25571== D1 miss rate:    20.4% ( 24.9% + 12.4% )
==25571== L2d miss rate:    4.5% ( 0.0% + 12.4% )
==25571==

==25571== L2 refs:      1,413,036 (1,100,502 rd + 312,534 wr)
==25571== L2 misses:      314,024 ( 1,490 rd + 312,534 wr)
==25571== L2 miss rate:    1.7% ( 0.0% + 12.4% )

```

在清单 5.5 的运行中，应用程序执行了 11 317 111 条指令；该项显示在 I refs 统计数据中。进程在 L1（215）和 L2（214）指令高速缓存中的缺失次数低得令人吃惊，由 I1 和 L2i miss rate 的 0.0% 表示。进程的数据引用总次数为 6 908 012，其中，4 405 958 次为读，2 502 054 次为写。24.9% 的读和 12.4% 的写无法由 L1 高速缓存满足。幸运的是，我们几乎总是可以在 L2 数据高速缓存中满足读操作，因此，它们显示的缺失率为 0%。写操作仍然是个问题，其缺失率为 12.4%。在这个应用程序中，数据的内存访问是需要调查的问题。

理想的应用程序应该有非常低的指令高速缓存和数据高速缓存缺失率。要消除指令高速缓存缺失，可能需要用不同的编译器选项重新编译应用程序，或者裁剪代码，这样热代码就不需要与不常用的代码一起共享 icache 空间了。要消除数据高速缓存缺失，使用数组而非链表作为数据结构，如果可能的话，降低数据结构中元素的大小，并用高速缓存友好的方式来访问内存。在任何情况下，valgrind 有助于指出哪个访问 / 数据结构需要进行优化。该应用程序运行的汇总信息表明数据访问是主要问题。

如清单 5.5 所示，这条命令显示了整体运行的高速缓存的使用情况统计信息。但是，对开发应用程序，或调查性能问题而言，更为有趣的是查看高速缓存缺失发生的位置，而不

是仅仅了解在应用程序运行期间出现的缺失总数。要确定由哪个函数为高速缓存缺失负责，我们运行 `cg_annotate`，如清单 5.6 所示。它向我们展示了哪个函数要为哪个高速缓存缺失负责。和我们预想的一样，函数 `a()` 的缺失次数（1 000 000）是函数 `b()`（100 000）的 10 倍。

清单 5.6

```
[ezolt@wintermute test_app]$ cg_annotate --25571
-----
I1 cache: 16384 B, 32 B, 4-way associative
D1 cache: 16384 B, 32 B, 4-way associative
L2 cache: 131072 B, 32 B, 4-way associative
Command: ./burn

Events recorded: Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw
Events shown: Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw
Event sort order: Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw
Thresholds: 99 0 0 0 0 0 0 0 0

Include dirs:
User annotated:
Auto-annotation: off
-----
Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw
-----
11,317,111 215 214 4,405,958 1,100,287 1,276 2,502,054 312,534 312,534 PROGRAM TOTALS
-----
Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw file:function
-----
8,009,011 2 2 4,003,003 1,000,000 989 1,004 0 0 burn.c:a
2,500,019 3 3 6 1 1 2,500,001 312,500 312,500 ???:_GI_memset
800,911 2 2 400,303 100,000 0 104 0 0 burn.c:b
```

虽然按单个函数来分解高速缓存缺失是有用的，但是查看应用程序的哪些行导致了高速缓存缺失也是很有趣的。如果我们如清单 5.7 那样使用 `--auto` 选项，`cg_annotate` 就会准确地告诉我们每个缺失都要由哪一行负责。

清单 5.7

```
[ezolt@wintermute test_app]$ cg_annotate --25571 --auto=yes
-----
I1 cache: 16384 B, 32 B, 4-way associative
D1 cache: 16384 B, 32 B, 4-way associative
L2 cache: 131072 B, 32 B, 4-way associative
Command: ./burn
```

```

Events recorded: Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw
Events shown:     Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw
Event sort order: Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw
Thresholds:      99 0 0 0 0 0 0 0 0
Include dirs:
User annotated:
Auto-annotation: on

-----  

          Ir I1mr I2mr      Dr      D1mr    D2mr      Dw      D1mw    D2mw  

-----  

11,317,111 215 214 4,405,958 1,100,287 1,276 2,502,054 312,534 312,534 PROGRAM TOTALS  

-----  

          Ir I1mr I2mr      Dr      D1mr    D2mr      Dw      D1mw    D2mw  file:function  

-----  

8,009,011  2   2 4,003,003 1,000,000  989      1,004      0      0  burn.c:a  

2,500,019  3   3       6           1      1 2,500,001 312,500 312,500  ???:_GI_memset  

800,911   2   2 400,303   100,000     0      104      0      0  burn.c:b  

-----  

-- Auto-annotated source: burn.c  

-----  

          Ir I1mr I2mr      Dr      D1mr    D2mr      Dw      D1mw    D2mw  

-----  

.. line 2 ..  

. . . . . . . . . #define ITER 100  

. . . . . . . . . #define SZ 10000000  

. . . . . . . . . #define STRI 10000  

. . . . . . . . . char test[SZ];  

. . . . . . . . . void a(void)  

3 0 0 . . . . . 1 0 0 {  

2 0 0 . . . . . 2 0 0 int i=0,j=0;  

5,004 1 1 2,001 0 0 1 0 0 for(j=0;j<10*ITER ; j++)  

5,004,000 0 0 2,001,000 0 0 1,000 0 0 for(i=0;i<SZ;i=i+STRI)  

. . . . . . . . . {  

3,000,000 1 1 2,000,000 1,000,000 989 . . . test[i]++;  

. . . . . . . . . }  

2 0 0 2 0 0 . . . }
```

如清单 5.7 所示，我们在不同的高速缓存缺失和命中发生时，进行了逐行分解。我们能看到内层 for 循环几乎包揽了所有的数据引用。和我们预期的一样，函数 a() 的 for 循环是函数 b() 的 10 倍。

valgrind/cachegrind 提供的不同级别（程序级、函数级和代码行级）的详细信息为你提供了一个很好的途径来了解应用程序的哪些部分正在访问内存，以及在有效地使用处理器的高速缓存。

5.2.5 kcachegrind

kcacheGrind 与 valgrind 密切合作，提供关于被剖析应用程序的高速缓存使用情况的详细信息。它在标准 valgrind 的基础上增加了两个新的功能。首先，它为 valgrind 提供了一个界面，称为 calltree，以捕捉特定应用程序的高速缓存和调用树的统计数据。其次，它还提供了对高速缓存性能信息的图形化展示，以及新颖的数据视图。

5.2.5.1 内存性能相关的选项

与 valgrind 相似，使用 kcachegrind 分析特定应用程序的高速缓存使用情况需要两个阶段：收集和注释。收集阶段从如下命令行开始：

```
calltree application
```

命令 calltree 用许多不同的选项来控制收集的信息。表 5-5 给出了其中一些比较重要的选项。

表 5-5 calltree 命令行选项

选 项	说 明
--help	对 calltree 支持的所有不同收集方法的简要说明
-dump-instr=yes no	已被进程锁定的内存总量。被锁定的内存不能交换到磁盘
-trace-jump=yes no	包括了分支信息，或者指出了每个分支选择了哪个路径

calltree 可以记录许多不同的统计信息。更多信息请参阅 calltree 的 help 选项。

在收集阶段，valgrind 用 calltree 界面来收集高速缓存使用情况的信息。上述命令行中的 application 代表的是要分析的应用程序。与 cachegrind 相同，在收集阶段，calltree 在屏幕上显示的是概要信息，而它同时也在名为 cachegrind.out.pid 的文件中保存了更加详细的统计数据，文件名中的 pid 是其运行时被剖析应用程序的 PID。

收集阶段完成后，用命令 kcachegrind 把高速缓存使用情况映射回应用程序的源代码。kcachegrind 调用方式如下：

```
kcachegrind cachegrind.out.pid
```

kcachegrind 显示已收集的高速缓存分析统计信息，使你能浏览结果。

5.2.5.2 用法示例

使用 kcachegrind 的第一步是用符号编译应用程序，以允许样本到源代码行的映射。下面的命令能完成这一步：

```
[ezolt@wintermute test_app]$ gcc -o burn burn.c -g3
```

第二步，对应用程序运行 calltree，如清单 5.8 所示。这提供了与 cachegrind 类似的输出，但是更重要的是，它生成了 cachegrind.out 文件，kcachegrind 将使用这个文件。

清单 5.8

```
[ezolt@wintermute test_app]$ calltree --dump-instr=yes --trace-jump=yes ./burn
==12242== Calltree-0.9.7, a call-graph generating cache profiler for x86-linux.
==12242== Copyright (C) 2002-2004, and GNU GPL'd, by N.Nethercote and J.Weidendorfer.
==12242== Using valgrind-2.0.0, a program supervision framework for x86-linux.
==12242== Copyright (C) 2000-2003, and GNU GPL'd, by Julian Seward.
==12242== Estimated CPU clock rate is 469 MHz
==12242== For more details, rerun with: -v
```

```

==12242==

==12242==

==12242== I refs: 33,808,151
==12242== I1 misses: 216
==12242== L2i misses: 215
==12242== I1 miss rate: 0.0%
==12242== L2i miss rate: 0.0%
==12242==

==12242== D refs: 29,404,027 (4,402,969 rd + 25,001,058 wr)
==12242== D1 misses: 4,225,324 (1,100,290 rd + 3,125,034 wr)
==12242== L2d misses: 4,225,324 (1,100,290 rd + 3,125,034 wr)
==12242== D1 miss rate: 14.3% ( 24.9% + 12.4% )
==12242== L2d miss rate: 14.3% ( 24.9% + 12.4% )

==12242==

==12242== L2 refs: 4,225,540 (1,100,506 rd + 3,125,034 wr)
==12242== L2 misses: 4,225,539 (1,100,505 rd + 3,125,034 wr)
==12242== L2 miss rate: 6.6% ( 2.8% + 12.4% )

```

当我们得到 cachegrind.out 文件后，我们可以用下面的命令来启动 kcachegrind (v.0.54) 对数据进行分析：

```
[ezolt@wintermute test_app]$ kcachegrind cachegrind.out.12242
```

这将弹出如图 5-3 所示的窗口。该窗口给出了对左边窗格中所有的高速缓存缺失的平面描述。默认情况下，显示 L1 高速缓存的数据读缺失。

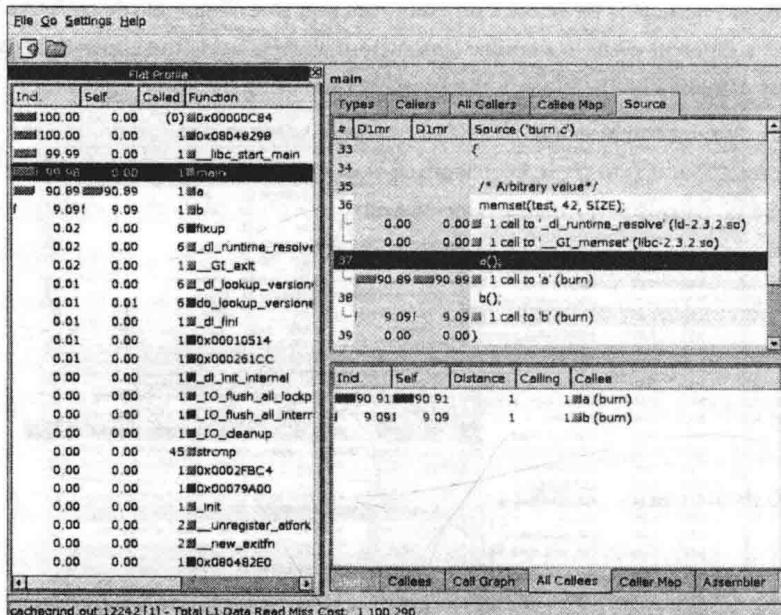


图 5-3

接下来，在图 5-4 右上方的窗格中，我们可以看到一种可视化的表示，表示对象为被调用者示意图，或者由左边窗格内函数（main）调用的所有函数（a() 和 b()）。在右下方的窗格中，我们可以看到该应用程序的调用图。

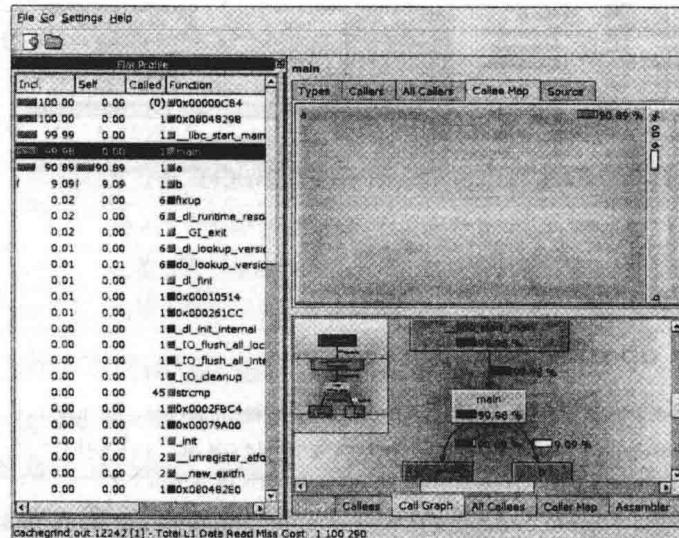


图 5-4

最后，在图 5-5 中，我们选择查看左边窗格内一个不同的函数，并用右上方窗格选择了一个不同的事件（Instruction Fetch）。最终，我们可以用右下方的窗格将用汇编代码表示的循环进行可视化。

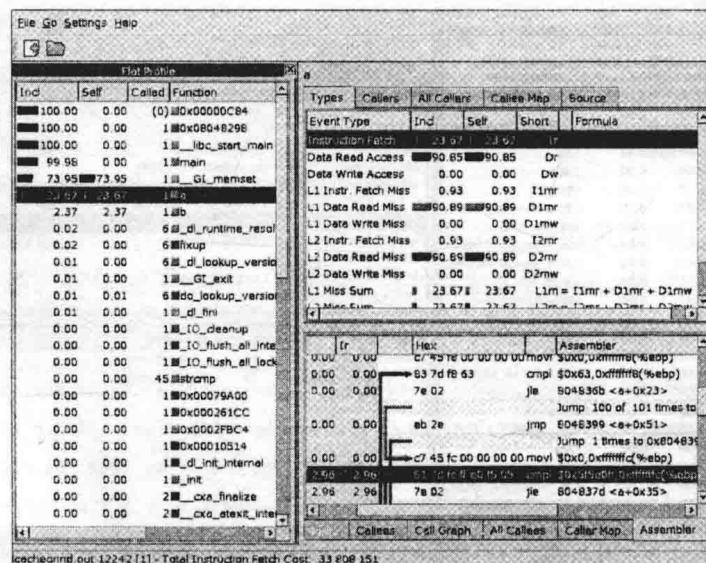


图 5-5

上述示例仅仅涉及了 kcachegrind 功能的表面，学习这个工具最好的方法就是尝试使用它。对那些喜欢将性能问题调查可视化的人来说，kcachegrind 是一个非常有用的工具。

5.2.6 oprofile (III)

如同你在前面章节里了解过的一样，oprofile 是一个强大的工具，它有助于确定应用程序的时间都花在哪些地方。不过，oprofile 还可以和处理器的性能计数器一起工作，以提供关于其执行情况的精确描述。cachegrind 模拟 CPU 来找到高速缓存的缺失 / 命中，而 oprofile 则利用实际的 CPU 性能计数器来精确查找高速缓存缺失的位置。与 cachegrind 不同，oprofile 不用模拟处理器，因此它记录的高速缓存结果是真实的，由操作系统导致的结果则将是可见的。此外，在 oprofile 下分析的应用程序执行起来基本上接近其原始速度，而 cachegrind 则要花更多时间。但是，与 oprofile 相比，cachegrind 更容易安装和使用，并且不论它运行于那个版本的 x86 处理器，它都可以跟踪同样的事件。当你需要一个快捷且相当准确的答案时，cachegrind 是一个很好的工具。而当你需要更加准确的统计信息或者 cachegrind 无法提供的统计信息时，oprofile 就很有用了。

5.2.6.1 内存性能相关的选项

本节在讨论 oprofile 时不再添加任何新的命令行选项，因为在关于 CPU 性能的章节中已经描述过它们了。但是当你开始采样与 oprofile 默认设置不同的事件时，有一个命令会变得更加重要。不同的处理器和架构可以采样不同的事件集，oprofile 的 `op_help` 命令会显示当前你的处理器所支持的事件列表。

5.2.6.2 用法示例

如前所述，oprofile 能够监控的事件都是与特定处理器相关的，因此，这些示例都是运行在当前的机器上的，即 Pentium III。在 Pentium III 上，我们可以利用 oprofile 提供的性能计数器来收集信息，这些信息与 valgrind 在 cachegrind 界面下给出的信息类似。这使用的就是性能计数器硬件，而不是软件模拟。使用性能硬件有两个缺陷。第一，我们必须应对性能计数器的底层硬件限制。在 Pentium III 上，oprofile 只能同时测量两个事件，而 cachegrind 则可以同时测量多种类型的内存事件。这就意味着，对 oprofile 来说要测量与 cachegrind 相同的事件，我们必须多次运行应用程序，并且每次运行期间都要改变 oprofile 的监控事件。第二个缺陷就是 oprofile 不会像 cachegrind 那样提供精确的事件计数，它只对计数器采样，因此我们能够看到事件最有可能在哪里发生，但是却无法看到确切的数量。实际上，我们只能接收到 (1/ 采样率) 个采样。如果应用程序只会让一个事件发生几次，那么这个事件可能根本就不会被记录下来。尽管在调试一个性能问题时，不知道事件确切的数目会让人沮丧，但通常最重要的却是弄清楚代码行之间样本的相对数量。即使无法直

接确定特定源代码行发生事件的总数，但是可以找到事件最多的那一行，一般说来，这足以启动性能问题调试。这种无法获取准确事件计数的情况出现在每一种形式的 CPU 采样中，也将出现在所有实现它的处理器性能硬件中。而实际上，正是这一限制，使得性能硬件从根本上得以存在，如果没有它，那么性能监控将会导致过多的消耗。

oprofile 可以监控许多不同类型的事件，这些事件在不同的处理器上会发生变化。作为示例，本小节将介绍一些 Pentium III 的重要事件。通过利用下述事件（事件及说明由 op_help 提供），Pentium III 可以监控 L1 数据高速缓存（在 Pentium III 中被称为 DCU）：

```
[ezolt@localhost book]$ op_help
oprofile: available events for CPU type "PIII"

See Intel Architecture Developer's Manual Volume 3, Appendix A and Intel
Architecture Optimization Reference Manual (730795-001)

.....
DCU_LINES_IN:      total lines allocated in the DCU
DCU_M_LINES_IN:   number of M state lines allocated in DCU
DCU_M_LINES_OUT:  number of M lines evicted from the DCU
DCU_MISS_OUTSTANDING: number of cycles while DCU miss outstanding
....
```

注意，我们没有与 cachegrind 提供信息的确切对应，即 L1 数据高速缓存的“读和写”数量。但是，通过使用 DCU_LINES_IN 事件，我们可以发现每个函数有多少周期的 L1 数据缺失。尽管这个事件不会告诉我们每个函数准确的缺失数量，但是它却会告诉我们每个函数相对于彼此在高速缓存中有多少缺失。监控 L2 数据高速缓存的事件有点接近，但它仍然没有与 cachegrind 提供信息的精确对应。下面是 Pentium III 的与 L2 数据高速缓存相关的事件：

```
[ezolt@localhost book]$ op_help
.....
L2_LD: number of L2 data loads
    Unit masks
    -----
    0x08: (M)odified cache state
    0x04: (E)xclusive cache state
    0x02: (S)hared cache state
    0x01: (I)nvalid cache state
    0x0f: All cache states

L2_ST: number of L2 data stores
    Unit masks
    -----
```

```

0x08: (M)odified cache state
0x04: (E)xclusive cache state
0x02: (S)hared cache state
0x01: (I)nvalid cache state
0x0f: All cache states

```

L2_LINES_IN: number of allocated lines in L2

...

Pentium III 实际支持的要比这些更多，但是上述这些是基本的加载和存储事件。（cachegrind 将加载称为“读”，将存储称为“写”。）我们可以用这些事件来计算每一行源代码上发生的加载和存储的数量。我们还可以用 L2_LINES_IN 来显示哪一块代码有更多的 L2 高速缓存缺失。如前所述，我们不会得到精确的数值。此外，在 Pentium III 上，指令和数据共享 L2 高速缓存。任何 L2 缺失都有可能是指令或数据缺失的结果。oprofile 展现给我们的 L2 中的高速缓存缺失是指令和数据缺失的结果，而 cachegrind 则帮助我们将它们区分开来。

在 Pentium III 上，有一系列相似的事件来监控指令高速缓存。对 L1 指令高速缓存（Pentium III 称其为“IFU”）而言，我们可以通过使用如下事件来直接测量读（或取）和缺失的数量：

```

[ezolt@localhost book]$ op_help
...
IFU_IFETCH: number of non/cachable instruction fetches

IFU_IFETCH_MISS: number of instruction fetch misses
...
```

我们还可以利用下述事件来测量从 L2 高速缓存中取到的指令数：

```

[ezolt@localhost book]$ op_help
...
L2_IFETCH: (counter: 0, 1)
    number of L2 instruction fetches (min count: 500)
    Unit masks
    -----
    0x08: (M)odified cache state
    0x04: (E)xclusive cache state
    0x02: (S)hared cache state
    0x01: (I)nvalid cache state
    0x0f: All cache states
...
```

可惜的是，如前面提到的一样，处理器的 L2 高速缓存是由指令和数据共享的，因此，没有办法区分数据使用中的高速缓存缺失和指令使用中的高速缓存缺失。我们可以利用这些事件来获得与 cachegrind 所提供信息的近似值。

如果提取相同信息是如此困难，为什么还要使用 oprofile？首先，oprofile 的开销非常低（< 10%）并且可以运行于产品化的应用程序。其次，oprofile 可以精确抽取正在发生的事件。这比可能不准确的模拟器要好得多，更不要说这个模拟器还不考虑操作系统或其他应用程序的高速缓存使用情况。

尽管这些事件在 Pentium III 上是可用的，但它们在其他处理器上不一定可用。每个 Intel 和 AMD 的处理器家族都有各种的事件集可用来提供关于内存子系统性能的不同数量的信息。op_help 显示可被监控的事件，但是要理解这些事件的含义可能还需要阅读详细的 Intel 或 AMD 处理器信息。

要了解怎样用 oprofile 来获取高速缓存信息，可用对比同一个应用程序在使用 cachegrind 的虚拟 CPU 以及使用 oprofile 的实际 CPU 这两种情况下的高速缓存使用情况信息。让我们运行一下之前的示例程序 burn。再说一次，这个应用程序在函数 a() 中运行的指令是函数 b() 中的十倍。它在函数 a() 中访问的数据量也是 b() 中的十倍。下面是这个示例程序在 cachegrind 中的输出：

```

Ir I1mr I2mr      Dr      D1mr D2mr      Dw      D1mw D2mw
-----
883,497,211 215 214 440,332,658 110,000,288 1,277 2,610,954 312,534 312,533 PROGRAM TOTALS

-----
Ir I1mr I2mr      Dr      D1mr D2mr      Dw      D1mw D2mw file:function
-----
800,900,011    2     2 400,300,003 100,000,000   989   100,004      0      0  ????:a
80,090,011     2     2 40,030,003 10,000,000     0    10,004      0      0  ????:b

```

在接下来的几个例子中，多次运行了 oprofile 的数据收集阶段。我使用 oprof_start 为特定的运行设置事件，然后再运行演示程序。因为我的 CPU 只有两个计数器，因此需要多次重复这个步骤。这就意味着程序不同的执行过程将可以监控到不同的事件集。由于我的应用程序在每次运行时不会改变其运行方式，因此，每次运行都应产生相同的结果。不过这一点对于更加复杂的应用程序来说不一定是正确的，比如 Web 服务器或数据库，它们会根据提出的请求动态改变其执行情况。然而，对于简单的测试程序来说这一点则是适用的。

收集了采样信息之后，我们用 oreport 提取被收集的信息。如清单 5.9 所示，我们向 oprofile 查询已生成的对数据内存引用的数量，以及在 L1 数据高速缓存（DCU）中有多少

次缺失。和 cachegrind 告诉我们的一样，函数 a() 对内存引用的次数和发生的 L1 数据缺失数是函数 b() 的 10 倍。

清单 5.9

```
[ezolt@wintermute test_app]$ oreport -l ./burn
event:DATA_MEM_REFs,DCU_LINES_IN
CPU: PIII, speed 467.74 MHz (estimated)
Counted DATA_MEM_REFs events (all memory references, cachable and non)
with a unit mask of 0x00 (No unit mask) count 233865
Counted DCU_LINES_IN events (total lines allocated in the DCU) with a
unit mask of 0x00 (No unit mask) count 23386
vma      samples %          samples %          symbol name
08048348 3640    90.8864    5598    90.9209    a
0804839e 365     9.1136     559     9.0791    b
```

现在查看清单 5.10，它显示 oreport 对指令高速缓存进行了同样的信息检测。请注意，和我们预期的一样，函数 a() 执行的指令数是函数 b() 的 10 倍。不过，L1 指令缺失的数量却和 cachegrind 预期的不同。最可能的原因是其他应用程序和内核正在使用高速缓存，从而导致 burn 在 icache 中出现缺失。(请记住 cachegrind 不会将内核或其他应用程序的高速缓存使用情况考虑在内。) 同样的情况也可能发生在数据高速缓存里，但由于应用程序导致的数据高速缓存缺失数量太大，以致其他的事件就被淹没在噪声里。

清单 5.10

```
[ezolt@wintermute test_app]$ oreport -l ./burn
event:IFU_IFETCH,IFU_IFETCH_MISS
CPU: PIII, speed 467.74 MHz (estimated)
Counted IFU_IFETCH events (number of non/cachable instruction fetches)
with a unit mask of 0x00 (No unit mask) count 233870
Counted IFU_IFETCH_MISS events (number of instruction fetch misses) with
a unit mask of 0x00 (No unit mask) count 500
vma      samples %          samples %          symbol name
08048348 8876    90.9240    14      93.3333    a
0804839e 886     9.0760     1       6.6667    b
```

比较 cachegrind 和 oprofile 的输出可知，用 oprofile 来收集与内存相关的信息是很有效的，因为 oprofile 开销低且能直接利用处理器硬件，但是它却难以找到与你感兴趣的内容相匹配的事件。

5.2.7 ipcs

ipcs 是一种系统级工具，可以展示进程之间通信内存的信息。进程可以分配整个系统共享的内存、信号量，以及由系统上运行的多个进程所共享的内存队列。ipcs 最好被用于跟

踪哪些应用程序分配并使用了大量的共享内存。

5.2.7.1 内存性能相关的选项

ipcs 用如下命令行调用：

```
ipcs [-t] [-c] [-l] [-u] [-p]
```

如果 ipcs 调用时不带任何参数，那么，它会给出系统中所有共享内存的汇总信息。其中包括拥有者信息以及共享内存段的大小信息。表 5-6 说明的选项能够让 ipcs 显示不同类型的系统共享内存信息。

表 5-6 ipcs 命令行选项

选 项	说 明
-t	显示共享内存创建时间，进程最后访问该内存的时间，以及进程最后与之分离的时间
-u	提供了关于共享内存使用量，以及它是否已被交换到磁盘还是仍留着内存的汇总信息
-l	显示了对共享内存使用情况的系统级限制
-p	显示了创建和最后使用共享内存段的进程的 PID
x	显示作为共享内存段的创建者和拥有者的用户

由于共享内存被多个进程使用，它不能归属于任何一个特定的进程。ipcs 提供了足够的系统级共享内存的状态信息，可以用于确定哪些进程分配了共享内存，哪些进程使用了它们，以及使用的频率。在试图降低共享内存使用量时，这些信息是很有用的。

5.2.7.2 用法示例

首先是清单 5.11，我们询问 ipcs 有多少系统内存被用作共享内存。这清晰地指明了系统中共享内存的状态。

清单 5.11

```
[ezolt@wintermute tmp]$ ipcs -u

----- Shared Memory Status -----
segments allocated 21
pages allocated 1585
pages resident 720
pages swapped 412
Swap performance: 0 attempts 0 successes

----- Semaphore Status -----
used arrays = 0
allocated semaphores = 0

----- Messages: Status -----

```

```
allocated queues = 0
used headers = 0
used space = 0 bytes
```

在这个例子中，我们看到有 21 个不同的内存段或内存片已经被分配了。所有这些段一共占用了 1585 个内存页。其中，720 页留驻在物理内存中，412 页已经交换到磁盘。

接下来是清单 5.12，我们要求 ipcs 为系统中所有的共享内存段提供一个概览。这会指明谁使用了哪个内存段。本例中，我们看到了包含所有共享段的清单。特别是共享内存 ID 为 65538 的，其用户 (ezolt) 就是拥有者。它的权限为 600(典型的 UNIX 权限)，在本例中，这意味着只有 ezolt 能够对其进行读写。该共享段有 393 216 个字节，有两个进程访问了它。

清单 5.12

```
[ezolt@wintermute tmp]$ ipcs
```

----- Shared Memory Segments -----

key	shmid	owner	perms	bytes	nattch	status
0x00000000 0		root	777	49152	1	
0x00000000 32769		root	777	16384	1	
0x00000000 65538		ezolt	600	393216	2	dest
0x00000000 98307		ezolt	600	393216	2	dest
0x00000000 131076		ezolt	600	393216	2	dest
0x00000000 163845		ezolt	600	393216	2	dest
0x00000000 196614		ezolt	600	393216	2	dest
0x00000000 229383		ezolt	600	393216	2	dest
0x00000000 262152		ezolt	600	393216	2	dest
0x00000000 294921		ezolt	600	393216	2	dest
0x00000000 327690		ezolt	600	393216	2	dest
0x00000000 360459		ezolt	600	393216	2	dest
0x00000000 393228		ezolt	600	393216	2	dest
0x00000000 425997		ezolt	600	393216	2	dest
0x00000000 458766		ezolt	600	393216	2	dest
0x00000000 491535		ezolt	600	393216	2	dest
0x00000000 622608		ezolt	600	393216	2	dest
0x00000000 819217		root	644	110592	2	dest
0x00000000 589842		ezolt	600	393216	2	dest
0x00000000 720916		ezolt	600	12288	2	dest
0x00000000 786454		ezolt	600	12288	2	dest

----- Semaphore Arrays -----

key	semid	owner	perms	nsems
-----	-------	-------	-------	-------

----- Message Queues -----

key	msqid	owner	perms	used-bytes	messages
-----	-------	-------	-------	------------	----------

最后，我们还可以确切地指出哪些进程创建了共享内存段，哪些进程使用了这些段，如清单 5.13 所示。对 shmid 为 32769 的段来说，我们可以看到 PID1224 的进程创建了它，最后使用它的是 PID 为 11954 的进程。

清单 5.13

```
[ezolt@wintermute tmp]$ ipcs -p
```

----- Shared Memory Creator/Last-op -----

shmid	owner	cpid	lpid
0	root	1224	11954
32769	root	1224	11954
65538	ezolt	1229	11954
98307	ezolt	1229	11954
131076	ezolt	1276	11954
163845	ezolt	1276	11954
196614	ezolt	1285	11954
229383	ezolt	1301	11954
262152	ezolt	1307	11954
294921	ezolt	1309	11954
327690	ezolt	1313	11954
360459	ezolt	1305	11954
393228	ezolt	1321	11954
425997	ezolt	1321	11954
458766	ezolt	1250	11954
491535	ezolt	1250	11954
622608	ezolt	1313	11954
819217	root	1224	11914
589842	ezolt	1432	14221
720916	ezolt	1250	11954
786454	ezolt	1313	11954

----- Message Queues PIDs -----

msqid	owner	lspid	lpid
-------	-------	-------	------

当我们知道了负责分配和使用这些段的 PID 之后，我们就可以使用诸如“ps -o command PID”的命令从 PID 回溯到进程名。

如果共享内存使用量占了系统总量的很大一部分，那么 ipcs 是一个很好的方法来准确

地跟踪那些创建和使用共享内存的程序。

5.2.8 动态语言 (Java 和 Mono)

与 CPU 性能工具一样，本章讨论的大多数工具都支持对诸如 C 和 C++ 的静态语言的分析。在我们调查的工具中，只有 ps、/proc 和 ipcs 是支持动态语言，如 Java、Mono、Python 和 Perl 等。高速缓存和内存剖析工具，如 oprofile 以及 memprof 则不支持动态语言。像 CPU 分析一样，每种语言都提供了自定义的工具来提取内存使用情况信息。

对 Java 应用程序而言，如果 Java 命令运行时带上了 -Xrunhprof 命令行选项，它就会分析应用程序的内存使用情况。更多详细信息参见 <http://antprof.sourceforge.net/hprof.html>，或者运行带 -Xrunhprof:help 选项的 java 命令。对 Mono 应用程序来说，如果向 mono 可执行代码传递了 --profile 标志，它也会分析应用程序的内存使用情况。更多详细信息参见 <http://www.go-mono.com/performance.html>。Perl 和 Python 没有出现类似的功能。

5.3 本章小结

本章介绍了各种可以用于诊断内存性能问题的 Linux 工具，展示了可以显示应用程序内存消耗量的工具 (ps、/proc)，以及显示应用程序中的哪些函数分配了内存的工具 (memprof)。本章还包括了可以监控处理器、系统高速缓存和内存子系统有效性的工具 (cachegrind、kcachegrind 和 oprofile)。本章最后描述了一种可以监控共享内存使用情况的工具 (ipcs)。这些工具一起使用就可以跟踪内存的每个分配、这些分配的大小、应用程序中这些分配的函数位置，以及在访问这些分配时，应用程序使用内存子系统的有效性。

下一章将调查磁盘 I/O 瓶颈。

性能工具：磁盘 I/O

本章介绍的性能工具能帮助你评估磁盘 I/O 子系统的使用情况。这些工具可以展示哪些磁盘或分区已被使用，每个磁盘处理了多少 I/O，发给这些磁盘的 I/O 请求要等多久才被处理。

阅读本章后，你将能够：

- 确定系统内磁盘 I/O 的总量和类型（读 / 写）(vmstat)。
- 确定哪些设备服务了大部分的磁盘 I/O (vmstat, iostat, sar)。
- 确定特定磁盘处理 I/O 请求的有效性 (iostat)。
- 确定哪些进程正在使用一组给定的文件 (lsof)。

6.1 磁盘 I/O 介绍

在深入性能工具之前，有必要了解 Linux 磁盘 I/O 系统是怎样构成的。大多数现代 Linux 系统都有一个或多个磁盘驱动。如果它们是 IDE 驱动，那么常常将被命名为 hda、hdb、hdc 等；而 SCSI 驱动则常常被命名为 sda、sdb、sdc 等。磁盘通常要分为多个分区，分区设备名称的创建方法是在基础驱动名称的后面直接添加分区编号。比如，系统中首个 IDE 硬驱动的第二个分区通常被标记为 /dev/hda2。一般每个独立分区要么包含一个文件系统，要么包含一个交换分区。这些分区被挂载到 Linux 根文件系统，该系统由 /etc/fstab 指定。这些被挂载的文件系统包含了应用程序要读写的文件。

当一个应用程序进行读写时，Linux 内核可以在其高速缓存或缓冲区中保存文件的副本，并且可以在不访问磁盘的情况下返回被请求的信息。但是，如果 Linux 内核没有在内存中

保存数据副本，那它就向磁盘 I/O 队列添加一个请求。若 Linux 内核注意到多个请求都指向磁盘内相邻的区域，它会把它们合并为一个大的请求。这种合并能消除第二次请求的寻道时间，以此来提高磁盘整体性能。当请求被放入磁盘队列，而磁盘当前不忙时，它就开始为 I/O 请求服务。如果磁盘正忙，则请求就在队列中等待，直到该设备可用，请求将被服务。

6.2 磁盘I/O性能工具

本节讨论各种各样的磁盘 I/O 性能工具，它们能使你调查一个给定应用程序是如何使用磁盘 I/O 子系统的，包括每个磁盘被使用的程度，内核的磁盘高速缓存的工作情况，以及特定应用程序“打开”了哪些文件。

6.2.1 vmstat (III)

如同你在第 2 章中了解到的，vmstat 是一个强大的工具，它能给出系统在性能方面的总览图。除了 CPU 和内存统计信息之外，vmstat 还可以提供系统整体上的 I/O 性能情况。

6.2.1.1 磁盘 I/O 性能相关的选项和输出

在使用 vmstat 从系统获取磁盘 I/O 统计信息时，要按照如下方式进行调用：

```
vmstat [-D] [-d] [-p partition] [interval [count]]
```

表 6-1 说明的命令行选项能影响 vmstat 显示的磁盘 I/O 统计信息。

表 6-1 vmstat 命令行选项

选 项	说 明
-D	显示 Linux I/O 子系统总的统计数据。它可以让你很好地了解你的 I/O 子系统是如何被使用的，但它不会给出单个磁盘的统计数据。显示的统计数据是从系统启动开始的总信息，而不是两次采样之间的发生量
-d	按每 interval 一个样本的速率显示单个磁盘的统计数据。这些统计信息是从系统启动开始的总信息，而不是两次采样之间的发生量
-p partition	按照每 interval 一个采样的速率显示给定分区的性能统计数据。这些统计信息是从系统启动开始的总信息，而不是两次采样之间的发生量
interval	采样之间的时间间隔
count	所取的样本总数

如果你在运行 vmstat 时只使用了 [interval] 和 [count] 参数，其他参数没有使用，那么显示的就是默认输出。该输出中包含了三列与磁盘 I/O 性能相关的内容：bo，bi 和 wa。这些统计信息的说明如表 6-2 所示。

在用 -D 模式运行时，vmstat 提供的是系统内磁盘 I/O 系统的总体统计数据。表 6-3 给出了这些统计信息。（注意：关于这些统计数据的更多信息参见 Documentation/iostats.txt 下的 Linux 内核源代码包）。

表 6-2 vmstat 的磁盘 I/O 统计信息（默认模式）

统计数据	说 明
bo	表示前次间隔中被写入磁盘的总块数（vmstat 内磁盘的典型块大小为 1024 字节）
bi	表示前次间隔中从磁盘读出的块数（vmstat 内磁盘的典型块大小为 1024 字节）
wa	表示等待 I/O 完成所消耗的 CPU 时间。每秒写磁盘块的速率

表 6-3 vmstat 的磁盘 I/O 统计信息（-D 模式）

统计数据	说 明
disks	系统中的磁盘总数
partitions	系统中的分区总数
total reads	读请求总数
merged reads	为了提升性能而被合并的不同读请求数量，这些读请求访问的是磁盘上的相邻位置
read sectors	从磁盘读取的扇区总数（一个扇区通常为 512 字节）
milli reading	磁盘读所花费的时间（以毫秒为单位）
writes	写请求的总数
merged writes	为了提升性能而被合并的不同写请求数量，这些写请求访问的是磁盘上的相邻位置
written sectors	向磁盘写入的扇区总数（一个扇区通常为 512 字节）
milli writing	磁盘写所花费的时间（以毫秒为单位）
inprogress IO	当前正在处理的 I/O 总数。请注意，最近版本（v3.2）的 vmstat 在这里有个漏洞，除以 1000 时其结果是错误的，几乎总是得到 0
milli spent IO	等待 I/O 完成所花费的毫秒数。请注意，最近版本（v3.2）的 vmstat 在这里有个漏洞，其数值为 I/O 花费的秒数，而非毫秒数

vmstat 的 -d 选项显示的是每一个磁盘的 I/O 统计信息。这些统计数据与 -D 选项的数据类似，表 6-4 对它们进行了解释。

表 6-4 vmstat 的磁盘 I/O 统计信息（-d 模式）

统计数据	说 明
reads: total	读请求的总数
reads: merged	为了提升性能而被合并的不同读请求数量，这些读请求访问的是磁盘上的相邻位置
reads: sectors	从磁盘读取的扇区总数
reads: ms	磁盘读所花费的时间（以毫秒为单位）
writes: total	写请求的总数
writes: merged	为了提升性能而被合并的不同写请求数量，这些写请求访问的是磁盘上的相邻位置
writes: sectors	向磁盘写入的扇区总数（一个扇区通常为 512 字节）
writes: ms	磁盘写所花费的时间（以毫秒为单位）
IO: cur	当前正在处理的 I/O 总数。请注意，最近版本（v3.2）的 vmstat 在这里有个漏洞，除以 1000 时其结果是错误的，几乎总是得到 0
IO: s	等待 I/O 完成所花费的秒数

最后，如果被要求提供特定分区的统计信息，那么 vmstat 就会显示如表 6-5 所示的数据项。

表 6-5 vmstat 的分区 I/O 统计信息

统计数据	说 明
<code>reads</code>	该分区的读请求总数
<code>read sectors</code>	从该分区读取的扇区数量
<code>writes</code>	该分区 I/O 导致的写总数
<code>requested writes</code>	该分区的写请求总数

vmstat 的默认输出提供了关于系统磁盘 I/O 的一个粗略但良好的指示。vmstat 提供的选项则使你能了解更多细节，以发现哪些设备要对 I/O 负责。vmstat 超过其他 I/O 工具的主要优势是：几乎所有的 Linux 发行版本都包含该工具。

6.2.1.2 用法示例

随着 vmstat 版本的升级，它能呈现给 Linux 用户的 I/O 统计信息数量也不断增加。本节给出的示例针对 3.2.0 或更高版本的 vmstat。此外，vmstat 提供的扩展磁盘统计信息只用于内核版本高于 2.5.70 的 Linux 系统。

在清单 6.1 显示的例子中，我们调用的 vmstat 只取 3 个样本，时间间隔为 1 秒。vmstat 输出整个系统的性能概况，如我们在第 2 章所见一样。

清单 6.1

```
[ezolt@wintermute procps-3.2.0]$ ./vmstat 1 3
procs .....memory..... swap... io... system... cpu...
 r b swpd free buff cache si so bi bo in cs us sy id wa
 1 1 0 197020 81804 29920 0 0 236 25 1017 67 1 1 93 4
 1 1 0 172252 106252 29952 0 0 24448 0 1200 395 1 36 0 63
 0 0 0 231068 50004 27924 0 0 19712 80 1179 345 1 34 15 49
```

清单 6.1 显示，在一次采样期间，系统读取了 24 448 个磁盘块。如前所述，磁盘块大小为 1 024 字节，这就意味着系统读取数据的速率约为每秒 23MB。我们还可以看到，在这个采样过程中，CPU 花费了相当多的时间来等待 I/O 完成，它有 63% 的时间用于 I/O，而磁盘读取速率约为每秒 23MB。在下一个采样中，它有 49% 的时间用于 I/O，而磁盘读取速率约为每秒 19MB。

接下来，在清单 6.2 中，我们要求 vmstat 提供自系统启动以来 I/O 子系统的性能信息。

清单 6.2

```
[ezolt@wintermute procps-3.2.0]$ ./vmstat -D
 3 disks
 5 partitions
 53256 total reads
 641233 merged reads
 4787741 read sectors
 343552 milli reading
```

```

14479 writes
17556 merged writes
257208 written sectors
7237771 milli writing
0 inprogress IO
342 milli spent IO

```

在清单 6.2 中，vmstat 提供了系统内所有磁盘驱动器的 I/O 统计汇总信息。如前所述，在读写磁盘时，为了提高性能，Linux 内核试图合并对磁盘相邻区域的请求。vmstat 在报告这些事件时，将它们称为 merged reads（合并读）和 merged writes（合并写）。本例中，大量发给系统的读请求在提交给设备之前被合并了。虽然合并读有约 640 000 个，但真正向设备提交的读命令却只有约 53 000 个。输出还告诉我们从磁盘中总共读出了 4 787 741 个扇区，并且自系统启动开始，从磁盘读取共花费了 343 552 毫秒（或 344 秒）。写性能也可以得到同样的统计信息。这些 I/O 统计信息能让我们很好地了解整个 I/O 子系统的性能。

上面的例子显示的是整个系统的 I/O 统计数据，而下面清单 6.3 中的例子显示的统计信息则细化到了每个独立磁盘。

清单 6.3

```
[ezolt@wintermute procps-3.2.0]$ ./vmstat -d 1 3
disk ..... reads ..... writes ..... IO .....
total merged sectors ms total merged sectors ms cur s
fd0 0 0 0 0 0 0 0 0 0 0 0 0
hde 17099 163180 671517 125006 8279 9925 146304 2831237 0 125
hda 0 0 0 0 0 0 0 0 0 0 0 0
fd0 0 0 0 0 0 0 0 0 0 0 0 0
hde 17288 169008 719645 125918 8279 9925 146304 2831237 0 126
hda 0 0 0 0 0 0 0 0 0 0 0 0
fd0 0 0 0 0 0 0 0 0 0 0 0 0
hde 17288 169008 719645 125918 8290 9934 146464 2831245 0 126
hda 0 0 0 0 0 0 0 0 0 0 0 0
```

清单 6.4 显示出有 60 (19 059–18 999) 个读和 94 (24 795–24 701) 个写提交给了分区 hde3。当你试图确定哪个磁盘分区最常被使用时，这个统计就显得特别有用。

清单 6.4

```
[ezolt@wintermute procps-3.2.0]$ ./vmstat -p hde3 1 3
hde3      reads   read sectors  writes   requested writes
18999     191986   24701    197608
19059     192466   24795    198360
19161     193282   24795    198360
```

虽然 vmstat 提供了单个磁盘 / 分区的统计信息，但是它只给出其总量，却不给出在采样过程中的变化率。因此，要分辨哪个设备的统计数据在采样期间发生了明显的变化就显得很困难。

6.2.2 iostat

iostat 与 vmstat 相似，但它是一个专门用于显示磁盘 I/O 子系统统计信息的工具。iostat 提供的信息细化到每个设备和每个分区从特定磁盘读写了多少个块。（iostat 中块大小一般为 512 字节。）此外，iostat 还可以提供大量的信息来显示磁盘是如何被利用的，以及 Linux 花费了多长时间来等待将请求提交到磁盘。

6.2.2.1 磁盘 I/O 性能相关的选项和输出

iostat 用如下命令行调用：

```
iostat [-d] [-k] [-x] [device] [interval [count]]
```

与 vmstat 很相似，iostat 可以定期显示性能统计信息。不同的选项可以改变 iostat 显示的统计数据，如表 6-6 所示。

表 6-6 iostat 命令行选项

选 项	说 明
-d	只显示磁盘 I/O 的统计信息，而不是默认信息。默认信息中还包括了 CPU 使用情况
-k	按 KB 显示统计数据，而不是按块显示
-x	显示扩展性能 I/O 统计信息
device	若指定设备，则 iostat 只显示该设备的信息
interval	采样间隔时间
count	获取的样本总数

iostat 默认输出显示的性能统计信息如表 6-7 所示。

表 6-7 iostat 设备统计信息

统计数据	说 明
tps	每秒传输次数。该项为每秒对设备 / 分区读写请求的次数
B1k_read/s	每秒读取磁盘块的速率
B1k_wrttn/s	每秒写入磁盘块的速率
B1k_read	在时间间隔内读取块的总数量
B1k_wrttn	在时间间隔内写入块的总数量

当你使用 -x 参数调用 iostat 时，它会显示更多关于磁盘 I/O 子系统的统计信息。这些扩展的统计信息如表 6-8 所示。

iostat 是一个有用的工具，它提供了迄今为止我所发现的最完整的磁盘 I/O 性能统计信息。虽然 vmstat 非常普及，并且提供了一些基本的统计信息，但是 iostat 更加完备。如果你的系统已经安装了 iostat 并且可用，那么当系统存在磁盘 I/O 性能问题时，首先使用的工具就应该是 iostat。

表 6-8 iostat 的扩展磁盘统计信息

统计数据	说 明
rrqm/s	在提交给磁盘前，被合并的读请求的数量
wrqm/s	在提交给磁盘前，被合并的写请求的数量
r/s	每秒提交给磁盘的读请求数量
w/s	每秒提交给磁盘的写请求数量
rsec/s	每秒读取的磁盘扇区数
wsec/s	每秒写入的磁盘扇区数
rkB/s	每秒从磁盘读取了多少 KB 的数据
wkB/s	每秒向磁盘写入了多少 KB 的数据
avgrq-sz	磁盘请求的平均大小（按扇区计）
avgqu-sz	磁盘请求队列的平均大小
await	完成对一个请求的服务所需的平均时间（按毫秒计）。该平均时间为请求在磁盘队列中等待的时间加上磁盘对其服务所需的时间
svctm	提交到磁盘的请求的平均服务时间（按毫秒计）。该项表明磁盘完成一个请求所花费的平均时间。与 await 不同，该项不包含在队列中等待的时间

6.2.2.2 用法示例

清单 6.5 给出了 iostat 运行的一个示例，一个磁盘基准测试程序向位于 /dev/hda2 分区上的文件系统写入一个测试文件。iostat 显示的第一个采样是自系统启动开始时系统的平均情况。第二个采样（及其后内容）是每个时间间隔为 1 秒的统计数据。

清单 6.5

```
[ezolt@localhost sysstat-5.0.2]$ ./iostat -d 1 2
Linux 2.4.22-1.2188.nptl (localhost.localdomain)          05/01/2004

Device:      tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
hda        7.18     121.12      343.87    1344206    3816510
hda1       0.00      0.03       0.00        316         46
hda2       7.09     119.75      337.59    1329018    3746776
hda3       0.09      1.33       6.28       14776      69688
hdb       0.00      0.00       0.00        16          0

Device:      tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
hda      105.05      5.78     12372.56       16      34272
hda1      0.00      0.00       0.00        0          0
hda2     100.36      5.78     11792.06       16      32664
hda3      4.69      0.00      580.51        0      1608
hdb      0.00      0.00       0.00        0          0
```

上面例子中一个有趣的地方是，/dev/hda3 不太活跃。在被测试的系统中，/dev/hda3 是一个交换分区。这个分区记录的任何活动都是由内核将内存交换到磁盘导致的。通过这种方式，iostat 提供了一种间接方式来确定系统中有多少磁盘 I/O 是交换造成的。

清单 6.6 显示了更多的 iostat 输出。

清单 6.6

```
[ezolt@localhost sysstat-5.0.2]$ ./iostat -x -dk 1 5 /dev/hda2
Linux 2.4.22-1.2188.nptl (localhost.localdomain)      05/01/2004
Device:    rrqm/s wrqm/s   r/s   w/s   rsec/s   wsec/s   rkB/s   wkB/s
avgrq-sz avgqu-sz   await   svctm %util
hda2        11.22  44.40  3.15  4.20  115.00  388.97   57.50  194.49
68.52     1.75 237.17 11.47  8.43

Device:    rrqm/s wrqm/s   r/s   w/s   rsec/s   wsec/s   rkB/s   wkB/s
avgrq-sz avgqu-sz   await   svctm %util
hda2        0.00 1548.00  0.00 100.00    0.00 13240.00    0.00 6620.00
132.40    55.13 538.60 10.00 100.00

Device:    rrqm/s wrqm/s   r/s   w/s   rsec/s   wsec/s   rkB/s   wkB/s
avgrq-sz avgqu-sz   await   svctm %util
hda2        0.00 1365.00  0.00 131.00    0.00 11672.00    0.00 5836.00
89.10    53.86 422.44  7.63 100.00

Device:    rrqm/s wrqm/s   r/s   w/s   rsec/s   wsec/s   rkB/s   wkB/s
avgrq-sz avgqu-sz   await   svctm %util
hda2        0.00 1483.00  0.00 84.00    0.00 12688.00    0.00 6344.00
151.05   39.69 399.52 11.90 100.00

Device:    rrqm/s wrqm/s   r/s   w/s   rsec/s   wsec/s   rkB/s   wkB/s
avgrq-sz avgqu-sz   await   svctm %util
hda2        0.00 2067.00  0.00 123.00    0.00 17664.00    0.00 8832.00
143.61   58.59 508.54  8.13 100.00
```

在清单 6.6 中，你可以看到平均队列长度相当高（约 237 ~ 538），其结果是，请求需等待的时间（约 422.44 ~ 538.60 毫秒）远远高于请求服务所花费的时间（7.63 ~ 11.90 毫秒）。这么高的平均服务时间，再加上利用率 100% 的事实，都表明了该磁盘处于完全饱和状态。

扩展 iostat 输出提供了太多的统计信息，使得它只适合在很宽的终端上的单行显示。但是，在识别成为瓶颈的特定磁盘时，这些信息几乎全是你所需要的。

6.2.3 sar (III)

第 2 章中曾经讨论过，sar 可以收集 Linux 系统多个不同方面的性能统计信息。除了 CPU 和内存之外，它还可以收集关于磁盘 I/O 子系统的信息。

6.2.3.1 磁盘 I/O 性能相关的选项和输出

当使用 sar 来监视磁盘 I/O 统计数据时，你可以用如下命令行来调用它：

```
sar -d [ interval [ count ] ]
```

通常，sar 显示的是系统中 CPU 使用的相关信息。若要显示磁盘使用情况的统计信息，你必须使用 -d 选项。sar 只能在高于 2.5.70 的内核版本中显示磁盘 I/O 统计数据。表 6-9 对其显示信息进行了说明。

表 6-9 sar 设备统计信息

统计数据	说 明
tps	每秒传输数。该项为每秒对设备 / 分区进行读写的次数
rd_sec/s	每秒读取的磁盘扇区数
wr_sec/s	每秒写入的磁盘扇区数

扇区数直接取自内核，虽然有可能发生变化，但通常情况下，其大小为 512 字节。

6.2.3.2 用法示例

在清单 6.7 中，sar 被用于收集系统设备的 I/O 信息。在列出设备时，sar 使用的是它们的主设备号和次设备号，而不是它们的名字。

清单 6.7

```
[ezolt@wintermute sysstat-5.0.2]$ sar -d 1 3
Linux 2.6.5 (wintermute.phil.org)      05/02/04
```

16:38:28	DEV	tps	rd_sec/s	wr_sec/s
16:38:29	dev2-0	0.00	0.00	0.00
16:38:29	dev33-0	115.15	808.08	2787.88
16:38:29	dev33-64	0.00	0.00	0.00
16:38:29	dev3-0	0.00	0.00	0.00
16:38:29	DEV	tps	rd_sec/s	wr_sec/s
16:38:30	dev2-0	0.00	0.00	0.00
16:38:30	dev33-0	237.00	1792.00	8.00
16:38:30	dev33-64	0.00	0.00	0.00
16:38:30	dev3-0	0.00	0.00	0.00
16:38:30	DEV	tps	rd_sec/s	wr_sec/s
16:38:31	dev2-0	0.00	0.00	0.00
16:38:31	dev33-0	201.00	1608.00	0.00
16:38:31	dev33-64	0.00	0.00	0.00
16:38:31	dev3-0	0.00	0.00	0.00

Average:	DEV	tps	rd_sec/s	wr_sec/s
Average:	dev2-0	0.00	0.00	0.00
Average:	dev33-0	184.62	1404.68	925.75
Average:	dev33-64	0.00	0.00	0.00
Average:	dev3-0	0.00	0.00	0.00

与 iostat 相比，sar 给出的磁盘 I/O 统计信息数量是有限的。但其可以同时记录多个不同类型统计信息的特点可以弥补这些缺点。

6.2.4 lsof (列出打开文件)

lsof 提供了一种方法来确定哪些进程打开了一个特定的文件。除了跟踪单个文件的用户外，lsof 还可以显示使用了特定目录下文件的进程。同时，它还可以递归搜索整个目录树，并列出使用了该目录树内文件的进程。在要筛选哪些应用程序产生了 I/O 时，lsof 是很有用的。

6.2.4.1 磁盘 I/O 性能相关的选项和输出

你可以使用如下命令行调用 lsof 来找出进程打开了哪些文件：

```
lsof [-r delay] [+D directory] [+d directory] [file]
```

通常，lsof 显示的是使用给定文件的进程。但是，通过使用 +d 和 +D 选项，它可以显示多个文件的相关信息。表 6-10 解释了 lsof 的命令行选项，它们可用于追踪 I/O 性能问题。

表 6-10 lsof 命令行选项

选 项	说 明
-r delay	使得 lsof 每间隔 delay 秒输出一次统计数据
+D directory	使得 lsof 递归搜索给定目录下的所有文件，并报告哪些进程正在使用它们
+d directory	使得 lsof 报告哪些进程正在使用给定目录下的文件

在展示哪些进程正在使用指定文件时，lsof 就会显示表 6-11 说明的统计信息。

表 6-11 lsof 文件统计信息

统计数据	说 明
COMMAND	打开该文件的命令的名称
PID	打开该文件的命令的 PID
USER	打开文件的用户
FD	该文件的描述符。txt 表示可执行文件，mem 表示内存映射文件
TYPE	文件类型，REG 表示常规文件
DEVICE	用主设备号和次设备号表示的设备编号
SIZE	文件的大小
NODE	文件的索引节点

虽然 lsof 不会给出特定进程进行文件访问的数量和类型，但它至少可以显示哪些进程正在使用特定文件。

6.2.4.2 用法示例

清单 6.8 给出了运行在 /user/bin 目录的 lsof。该运行显示了访问 /user/bin 下所有文件的进程。

清单 6.8

```
[ezolt@localhost manuscript]$ /usr/sbin/lsof -r 2 +D /usr/bin/
COMMAND PID USER FD TYPE DEVICE SIZE NODE NAME
gnome-ses 2162 ezolt txt REG 3,2 113800 597490 /usr/bin/gnome-session
ssh-agent 2175 ezolt txt REG 3,2 61372 596783 /usr/bin/ssh-agent
gnome-key 2182 ezolt txt REG 3,2 77664 602727 /usr/bin/gnome-keyring-daemon
metacity 2186 ezolt txt REG 3,2 486520 597321 /usr/bin/metacity
gnome-pan 2272 ezolt txt REG 3,2 503100 602174 /usr/bin/gnome-panel
nautilus 2280 ezolt txt REG 3,2 677812 598239 /usr/bin/nautilus
magicdev 2287 ezolt txt REG 3,2 27008 598375 /usr/bin/magicdev
eggcups 2292 ezolt txt REG 3,2 32108 599596 /usr/bin/eggcups
pam-panel 2305 ezolt txt REG 3,2 45672 600140 /usr/bin/pam-panel-icon
gnome-ter 3807 ezolt txt REG 3,2 289116 596834 /usr/bin/gnome-terminal
less 6452 ezolt txt REG 3,2 104604 596239 /usr/bin/less
=====
COMMAND PID USER FD TYPE DEVICE SIZE NODE NAME
gnome-ses 2162 ezolt txt REG 3,2 113800 597490 /usr/bin/gnome-session
ssh-agent 2175 ezolt txt REG 3,2 61372 596783 /usr/bin/ssh-agent
gnome-key 2182 ezolt txt REG 3,2 77664 602727 /usr/bin/gnome-keyring-daemon
metacity 2186 ezolt txt REG 3,2 486520 597321 /usr/bin/metacity
gnome-pan 2272 ezolt txt REG 3,2 503100 602174 /usr/bin/gnome-panel
nautilus 2280 ezolt txt REG 3,2 677812 598239 /usr/bin/nautilus
magicdev 2287 ezolt txt REG 3,2 27008 598375 /usr/bin/magicdev
eggcups 2292 ezolt txt REG 3,2 32108 599596 /usr/bin/eggcups
pam-panel 2305 ezolt txt REG 3,2 45672 600140 /usr/bin/pam-panel-icon
gnome-ter 3807 ezolt txt REG 3,2 289116 596834 /usr/bin/gnome-terminal
less 6452 ezolt txt REG 3,2 104604 596239 /usr/bin/less
```

我们特别看看进程 3807，它使用了文件 /user/bin/gnome-terminal。根据 FD 列给出的 txt，该文件是一个可执行文件，使用它的命令的名称为 gnome-terminal。这是合情合理的，因为运行 gnome-terminal 的进程必须打开这个可执行文件。需要注意的一个有趣的现象是，这个文件位于设备“3, 2”上，其对应的是 /dev/hda2。（通过执行 ls -la /dev 并查看通常显示大小的输出字段，你就可以发现所有系统设备的设备号。）如果你知道某个设备是 I/O 瓶颈的源头，那么了

解文件位于哪个设备就会有所帮助。`lsof`具有一个独特的能力，它能根据打开文件描述符回溯到单个进程。尽管它不会显示哪些进程有大量的I/O，但是它确实提供了一个起点。

6.3 缺什么

所有的Linux磁盘I/O工具都可以提供关于特定磁盘或分区的使用信息。可惜的是，当你确定了某个磁盘是瓶颈之后，没有工具能够帮助你找出是哪个进程导致了这些I/O流量。

一般情况下，系统管理员比较了解哪个应用程序在使用磁盘，但情况并不总是这样。比如，很多时候我正在使用我的Linux系统，而磁盘却开始无缘无故地频繁读写。通常，我可以运行`top`来找出可能导致这个问题的进程。通过剔除那些我认为与I/O无关的进程，一般就可以找出罪魁祸首。但是，做到这一点就需要具备相应的知识，了解各种应用程序应该做什么。而且这种方式也容易出错，因为对哪些进程不会导致问题的猜测有可能是错误的。此外，对一个有着多个用户或运行多个应用程序的系统来说，要确定哪个应用程序可能引发问题常常不太实用也不太容易。其他的UNIX系统支持`ps`中的`inblk`和`outblk`参数，可以向你显示特定进程的磁盘I/O数量。目前，Linux内核不跟踪进程的I/O，因此工具`ps`无法收集这些信息。

你可以用`lsof`来确定哪些进程访问了特定分区上的文件。在列出了访问文件的全部PID后，你就能够对每个PID使用`strace`，找出具有大量I/O的那一个。虽然这种解决方法有效，但它治标不治本，因为访问一个分区的进程可能很多，且关联并分析每个进程的系统调用也很费时。同时，这还可能错过短进程，而在跟踪进程时，还有可能严重减缓它们的速度。

在这一点上，Linux内核是可以被改进的。若能快速追踪哪些进程产生了I/O，将使得诊断I/O性能相关问题更加迅速。

6.4 本章小结

本章介绍了Linux磁盘I/O性能工具，它们能用于提取关于系统级(`vmstat`)、特定设备(`vmstat`、`iostat`、`sar`)以及特定文件(`lsof`)的磁盘I/O使用信息。本章说明了不同类型的I/O统计信息，以及如何用I/O性能工具从Linux抽取这些统计数据。此外，本章还对当前工具主要的局限性和未来可发展的领域进行了讨论。

下一章介绍的工具将能够让你确定网络瓶颈的成因。

性能工具：网络

本章介绍一些在 Linux 上可用的网络性能工具。我们主要关注分析单个设备 / 系统网络流量的工具，而非全网管理工具。虽然在完全隔离的情况下评估网络性能通常是无意义的（节点不会与自己通信），但是，调查单个系统在网络上的行为对确定本地配置和应用程序的问题是有帮助的。此外，了解单系统的网络流量特性也有助于找到其他有问题的系统，以及造成网络性能降低的本地硬件和应用程序错误。

阅读本章后，你将能够：

- 确定系统内以太网设备的速度和双工设置 (`mii-tool`、`ethtool`)。
- 确定流经每个以太网接口的网络流量 (`ifconfig`、`sar`、`gkrellm`、`iptraf`、`netstat`、`etherape`)。
- 确定流入和流出系统的 IP 流量的类型 (`gkrellm`、`iptraf`、`netstat`、`etherape`)。
- 确定流入和流出系统的每种类型的 IP 流量 (`gkrellm`、`iptraf`、`etherape`)。
- 确定是哪个应用程序产生了 IP 流量 (`netstat`)。

7.1 网络I/O介绍

Linux 和其他主流操作系统中的网络流量被抽象为一系列的硬件和软件层次。链路层，也就是最低一层，包含网络硬件，如以太网设备。在传送网路流量时，这一层并不区分流量类型，而仅仅以尽可能快的速度发送和接收数据（或帧）。

链路层的上面是网络层。这一层使用互联网协议 (IP) 和网际控制报文协议 (ICMP) 在

机器间寻址并路由数据包。IP/ICMP 尽其最大努力尝试在机器之间传递数据包，但是它们不能保证数据包是否能真正达到其目的地。

网络层的上面是传输层，它定义了传输控制协议（TCP）和用户数据报协议（UDP）。TCP 是一个可靠协议，它可以保证消息通过网络送达，如果消息无法送达它就会产生一个错误。TCP 的同级协议 UDP，则是一个不可靠协议，它无法保证信息能够送达（为了获得最高的数据传输速率）。UDP 和 TCP 为 IP 增加了“服务”的概念。UDP 和 TCP 接收有编号“端口”的消息。按照惯例，每个类型的网络服务都被分配了不同的编号。例如，超文本传输协议（HTTP）通常为端口 80，安全外壳（SSH）通常为端口 22，文件传输协议（FTP）通常为端口 23。在 Linux 系统中，文件 /etc/services 定义了全部的端口以及它们提供的服务类型。

最上一层为应用层。这一层包含了各种应用程序，它们使用下面各层在网络上传输数据包。这些应用程序包括：Web 服务器、SSH 客户端，甚至是 P2P 文件共享客户端，比如 BitTorrent。

在 Linux 内核实现或控制的是最低三层（链路层、网络层和传输层）。内核可以提供每层的性能统计信息，包括数据流经每一层时的带宽使用情况信息和错误计数信息。本章介绍的工具就能使你提取并查看这些统计信息。

7.1.1 链路层的网络流量

在网络层次结构的最低几层，Linux 可以侦测到流经链路层的数据流量的速率。链路层，通常是以太网，以帧序列的形式将信息发送到网络上。即便是其上层次的信息片段的大小比帧大很多，链路层也会将它们分割为帧，再发送到网络上。数据帧的最大尺寸被称为最大传输单位（MTU）。你可以使用网络配置工具，如 ip 或 ifconfig 来设置 MTU。对以太网而言，最大大小一般为 1500 字节，虽然有些硬件支持的巨型帧可以高达 9000 字节。MTU 的大小对网络效率有直接影响。链路层上的每一个帧都有一个小容量的头部，因此，使用大尺寸的 MTU 就提高了用户数据对开销（头部）的比例。但是，使用大尺寸的 MTU，每个数据帧被损坏或丢弃的几率会更高。对清洁物理链路来说，大尺寸 MTU 通常会带来更好的性能，因为它需要的开销更小；反之，对嘈杂的链路来说，更小的 MTU 则通常会提升性能，因为，当单个帧被损坏时，它要重传的数据更少。

在物理层，帧流经物理网络，Linux 内核可以收集大量有关帧数量和类型的不同统计数据：

- 发送 / 接收——如果一个帧成功地流出或流入机器，那么它就会被计为一个已发送或已接收的帧。
- 错误——有错误的帧（可能是因为网络电缆坏了，或双工不匹配）。
- 丢弃——被丢弃帧的（很可能是因为内存或缓冲区容量小）。

- 溢出——由于内核或网卡有过多的帧，因此被网络丢弃的帧。通常这种情况不应该发生。
- 帧——由于物理级问题导致被丢弃的帧。其原因可能是循环冗余校验（CRC）错误或其他低级别的问题。
- 多播——这些帧不直接寻址到当前系统，而是同时广播到一组节点。
- 压缩——一些底层接口，如点对点协议（PPP）或串行线路网际协议（SLIP）设备在把帧发送到网络上之前，会对其进行压缩。该值表示的就是被压缩帧的数量。

有些 Linux 网络性能工具能够显示通过每一个网络设备的每一种类型的帧数。这些工具通常需要设备名，因此，熟悉 Linux 如何对网络设备命名以便搞清楚哪个名字代表了哪个设备是很重要的。以太网设备被命名为 ethN，其中，eth0 指的是第一个设备，eth1 指的是第二个设备，以此类推。与以太网设备命名方式相同，PPP 设备被命名为 pppN。环回设备，用于与本机联网，被命名为 lo。

在调查性能问题时，非常关键的一点是要清楚底层物理层能够支持的最大速度。比方说，以太网设备通常支持多种速度，如 10Mbps、100Mbps，甚至是 1000Mbps。底层以太网卡和基础设施（交换机）必须能控制所需的速度。虽然大多数网卡可以自动检测能支持的最高速度，并进行适当地自我设置，但是，如果一个网卡或交换机设置错误，就会影响到性能。如果不能达到更高的速度，以太网设备一般会协商降低速度，但它们仍然是起作用的。假如网络性能大大低于预期，那么最好使用工具 ethtool 或 mii-tool 来检验以太网速度是否设置为你的期望值。

7.1.2 协议层网络流量

对 TCP 或 UDP 流量而言，Linux 使用套接字 / 端口来抽象两台机器的连接。当与远程机器连接时，本地应用程序用一个网络套接字来打开远程机器上的一个端口。如前所述，常见网络服务都有约定的端口号，因此，给定的应用程序就能连接到远程机器的正确端口上。比如，端口 80 通常用于 HTTP。在加载一个 Web 页面时，浏览器就连接到远程机器的 80 端口上。远程机器上的 Web 服务器监听 80 端口上的连接，当连接发生时，Web 服务器就为 Web 页面的传输设置该连接。

Linux 网络性能工具可以跟踪流经特定网络端口的数据量。由于每个服务的端口号具有唯一性，因此有可能确定流向特定服务的物理流量。

7.2 网络性能工具

本节介绍能够诊断性能问题的 Linux 网络性能工具。我们先从确定最底层网络性能（物

理统计信息)的工具开始,然后逐步增加可以调查其上各层的工具。

7.2.1 mii-tool (媒体无关接口工具)

mii-tool 是以太网专用硬件工具,主要用于设置以太网设备,但它也可以提供有关当前设置的信息。这个信息,诸如链接速度和双工设置,对于追踪性能不佳设备的成因是非常有用。

7.2.1.1 网络 I/O 性能相关的选项

使用 mii-tool 时需要根访问权限。其调用命令行如下:

```
mii-tool [-v] [device]
```

mii-tool 输出指定设备的以太网设置。如果没有指定设备,那么 mii-tool 就会显示所有可用以太网设备的信息。若使用了 -v 选项,mii-tool 将显示被提供或协商的网络功能的详细信息。

7.2.1.2 用法示例

清单 7.1 显示的是系统上 eth0 的配置信息。第一行告诉我们网络设备正在使用 100BASE-T 全双工连接。接下来的几行描述了机器网卡的功能,以及该网卡检测到的线路另一端网络设备的功能。

清单 7.1

```
[root@nohs linux-2.6.8-1.521]# /sbin/mii-tool -v eth0
eth0: negotiated 100baseTx-FD, link ok
product info: vendor 00:00:00, model 0 rev 0
basic mode: autonegotiation enabled
basic status: autonegotiation complete, link ok
capabilities: 100baseTx-FD 100baseTx-HD 10baseT-FD 10baseT-HD
advertising: 100baseTx-FD 100baseTx-HD 10baseT-FD 10baseT-HD flow-control
link partner: 100baseTx-FD 100baseTx-HD 10baseT-FD 10baseT-HD
```

mii-tool 提供了关于如何配置以太网设备物理层的底层信息。

7.2.2 ethtool

在配置和显示以太网设备统计数据方面,ethtool 提供了与 mii-tool 相似功能。不过,ethtool 更加强大,包含了更多配置选项和设备统计信息。

7.2.2.1 网络 I/O 性能相关的选项

ethtool 在使用时需要根访问权限,其调用使用如下命令行:

ethtool [device]

ethtool 输出给定的以太网设备的配置信息。如果没有特别指定设备，ethtool 就会输出系统中所有以太网设备的统计信息。ethtool 的主页详细说明了修改当前以太网设置的选项。

7.2.2.2 用法示例

清单 7.2 显示了系统内 eth0 的配置信息。虽然该设备支持多种不同的速度和链接设置，但它当前连接到的是一个全双工，1000Mbps 的链路。

清单 7.2

```
[root@scrffy tmp]# /sbin/ethtool eth0
Settings for eth0:
  Supported ports: [ TP ]
  Supported link modes:  10baseT/Half 10baseT/Full
                         100baseT/Half 100baseT/Full
                         1000baseT/Half 1000baseT/Full
  Supports auto-negotiation: Yes
  Advertised link modes:  10baseT/Half 10baseT/Full
                         100baseT/Half 100baseT/Full
                         1000baseT/Half 1000baseT/Full
  Advertised auto-negotiation: Yes
  Speed: 1000Mb/s
  Duplex: Full
  Port: Twisted Pair
  PHYAD: 0
  Transceiver: internal
  Auto-negotiation: on
  Supports Wake-on: g
  Wake-on: d
  Link detected: yes
```

ethtool 运行简单，它能迅速提供配置不当的网络设备的有关信息。

7.2.3 ifconfig (接口配置)

ifconfig 的主要工作就是在 Linux 机器上安装和配置网络接口。它还提供了系统中所有网络设备的基本性能统计信息。ifconfig 几乎在所有联网的 Linux 机器上都是可用的。

7.2.3.1 网络 I/O 性能相关的选项

ifconfig 用如下命令行调用：

ifconfig [device]

如果没有指定设备，ifconfig 就会显示所有活跃的网络设备。表 7-1 解释了 ifconfig 提供的性能统计项。

表 7-1 ifconfig 的性能统计信息

列	说 明
RX packets	设备已接收的数据包数
TX packets	设备已发送的数据包数
errors	发送或接收时的错误数
dropped	发送或接收时丢弃的数据包数
overruns	网络设备没有足够的缓冲区来发送或接收一个数据包的次数
frame	底层以太网帧错误的数量
carrier	由于链路介质故障（如故障电缆）而丢弃的数据包数量

尽管 ifconfig 主要用于网络配置，但由它提供的适当数量的统计信息也能使你确定系统中每一个网络设备的健康和性能状况。

7.2.3.2 用法示例

清单 7.3 显示了来自系统所有设备的网络性能统计信息。这里，我们有一个以太网卡（eth0）和一个环回（lo）设备。本例中，以太网卡接收数据量约为 790Mb，发送数据量约为 319Mb。

清单 7.3

```
[ezolt@wintermute tmp]$ /sbin/ifconfig
eth0      Link encap:Ethernet  HWaddr 00:02:E3:15:A5:03
          inet addr:192.168.0.4  Bcast:192.168.0.255  Mask:255.255.255.0
                  UP BROADCAST NOTRAILERS RUNNING  MTU:1500  Metric:1
                  RX packets:1047040 errors:0 dropped:0 overruns:0 frame:0
                  TX packets:796733 errors:12 dropped:0 overruns:12 carrier:12
                  collisions:0 txqueuelen:1000
                  RX bytes:829403956 (790.9 Mb)  TX bytes:334962327 (319.4 Mb)
                  Interrupt:19 Base address:0x3000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
                  UP LOOPBACK RUNNING  MTU:16436  Metric:1
                  RX packets:102 errors:0 dropped:0 overruns:0 frame:0
                  TX packets:102 errors:0 dropped:0 overruns:0 carrier:0
                  collisions:0 txqueuelen:0
                  RX bytes:6492 (6.3 Kb)  TX bytes:6492 (6.3 Kb)
```

ifconfig 提供的统计数据是自系统启动开始的累计数值。如果你将一个网络设备下线，之后又让其上线，其统计数据也不会重置。如果你按规律的间隔来运行 ifconfig，就

可以发现各种统计数据的变化率。这一点可以通过 watch 命令或 shell 脚本来自动实现，这两种方式我们将在下一章讨论。

7.2.4 ip

一些网络工具，如 ifconfig，正在被淘汰，取而代之的是新的命令：ip。ip 不仅可以让你对 Linux 联网的多个不同方面进行配置，还可以显示每个网络设备的性能统计信息。

7.2.4.1 网络 I/O 性能相关的选项

提取性能统计数据时，用如下命令行调用 ip：

```
ip -s [-s] link
```

如果你用上述选项调用 ip，它就会输出系统中所有网络设备的统计信息，包括环回（lo）设备和简单互联网转换（sit0）设备。设备 sit0 允许将 IPv6 的数据包封装到 IPv4 的数据包中，并保持下来，这样可以缓解 IPv4 和 IPv6 之间的转换。如果 ip 中还有一个 -s，它将会提供底层以太网更加详细的统计信息。表 7-2 对 ip 提供的部分性能统计信息进行了说明。

表 7-2 ip 的网络性能输出统计信息

列	说 明
bytes	发送或接收的字节数
packets	发送或接收的数据包数
errors	发送或接收时发生的错误数
dropped	由于网卡缺少资源，导致未发送或接收的数据包数
overruns	网络没有足够的缓冲区空间来发送或接收更多数据包的次数
mcast	已接收的多播数据包的数量
carrier	由于链路介质故障（如故障电缆）而丢弃的数据包数量
collsns	传送时设备发生的冲突次数。当多个设备试图同时使用网络时就会发生冲突

ip 是一个非常灵活的 Linux 网络配置工具，虽然它的主要功能是对网络进行配置，但你也可以用它来提取底层设备的统计数据。

7.2.4.2 用法示例

清单 7.4 给出了系统中所有设备的网络性能统计信息。这里，我们有一个以太网卡，一个环回设备，和 sit0 通道设备。本例中，以太网卡接收数据大约为 820Mb，发送数据大约为 799Mb。

清单 7.4

```
[ezolt@nohs ezolt]$ /sbin/ip -s link
1: lo: <LOOPBACK,UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    RX: bytes   packets   errors   dropped overrun mcast
        4460       67          0          0          0          0
```

```

TX: bytes packets errors dropped carrier collsns
4460      67      0      0      0      0

2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 1000
link/ether 00:10:b5:59:2c:82 brd ff:ff:ff:ff:ff:ff
RX: bytes packets errors dropped overrun mcast
799273378 920999  0      0      0      0
TX: bytes packets errors dropped carrier collsns
820603574 930929  0      0      0      0

3: sit0: <NOARP> mtu 1480 qdisc noop
link/sit 0.0.0.0 brd 0.0.0.0
RX: bytes packets errors dropped overrun mcast
0      0      0      0      0      0
TX: bytes packets errors dropped carrier collsns
0      0      0      0      0      0

```

与 ifconfig 非常相似的是，ip 提供的是自系统启动开始的总的系统统计数据。如果使用 watch（下一章讨论），你就可以监控这些数值是如何随着时间发生变化的。

7.2.5 sar (IV)

前面的章节已经讨论过，sar 是最灵活的 Linux 性能工具之一。它可以监控许多不同的事情，归档统计数据，甚至还能用其他工具可用的格式来显示信息。sar 并不能总是与专门领域性能工具一样来提供尽可能多的详细信息，但它能给出一个很好的总体概况。

网络性能统计信息并无不同。和 ip 以及 ifconfig 一样，sar 提供了链路级的网络性能数据。但是，它同时还提供了一些关于传输层打开的套接字数量的基本信息。

7.2.5.1 网络 I/O 性能相关的选项

sar 使用如下命令行来收集网络统计信息：

```
sar [-n DEV | EDEV | SOCK | FULL ] [DEVICE] [interval] [count]
```

sar 收集多种不同类型的性能统计数据。表 7-3 解释了一些命令行选项，sar 使用它们来显示网络性能统计信息。

表 7-3 sar 命令行选项

选 项	说 明
-n DEV	显示每个设备发送和接收的数据包数和字节数信息
-n EDEV	显示每个设备的发送和接收错误信息
-n SOCK	显示使用套接字 (TCP、UDP 和 RAW) 的总数信息
-n FULL	显示所有的网络统计信息
interval	采样间隔时长
count	采样总数

表 7-4 给出了 sar 提供的网络性能选项。

表 7-4 sar 网络性能统计信息

选 项	说 明
rxpck/s	数据包接收速率
txpck/s	数据包发送速率
rxbyt/s	字节接收速率
txbyt/s	字节发送速率
rxcmp/s	压缩包接收速率
txcmp/s	压缩包发送速率
rxmcst/s	多播包接收速率
rxerr/s	接收错误率
txerr/s	发送错误率
coll/s	发送时的以太网冲突率
rxdrop/s	由于 Linux 内核缓冲区不足而导致的接收帧丢弃率
txdrop/s	由于 Linux 内核缓冲区不足而导致的发送帧丢弃率
txcarr/s	由于载波错误而导致的发送帧丢弃率
rxfram/s	由于帧对齐错误而导致的接收帧丢弃率
rxfifo/s	由于 FIFO 错误而导致的接收帧丢弃率
txfifo/s	由于 FIFO 错误而导致的发送帧丢弃率
totsck	当前正在被使用的套接字总数
tcpsock	当前正在被使用的 TCP 套接字总数
udpsck	当前正在被使用的 UDP 套接字总数
rawsck	当前正在被使用的 RAW 套接字总数
ip-frag	IP 分片的总数

考虑到 sar 能收集到全部统计信息，它确实为单点提供了最系统级的性能统计数据。

7.2.5.2 用法示例

在清单 7.5 中，我们查看了系统中所有网络设备的发送和接收统计信息。就像你能看到的，设备 eth0 是最活跃的。在第一个采样，eth0 每秒接收的数据大约为 63 000 字节 (rxbyt/s)，发送的数据大约为 45 000 字节 (txbyt/s)。未发送或接收压缩数据包 (txcmp、rxcmp)。(压缩数据包通常出现在 SLIP 或 PPP 连接中)。

清单 7.5

```
[ezolt@wintermute sysstat-5.0.2]$ sar -n DEV 1 2
Linux 2.4.22-1.2174.nptlsmp (wintermute.phil.org)      06/07/04

21:22:29  IFACE   rxpck/s   txpck/s   rxbyt/s   txbyt/s   rxcmp/s   txcmp/s   rxmcst/s
21:22:30    lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00
21:22:30  eth0     68.00     65.00   63144.00   45731.00      0.00      0.00      0.00
```

21:22:30	IFACE	rxpck/s	txpck/s	rxbyt/s	txbyt/s	rxcmp/s	txcmp/s	rxmlst/s
21:22:31	lo	0.00	0.00	0.00	0.00	0.00	0.00	0.00
21:22:31	eth0	80.39	47.06	45430.39	30546.08	0.00	0.00	0.00
<hr/>								
Average:	IFACE	rxpck/s	txpck/s	rxbyt/s	txbyt/s	rxcmp/s	txcmp/s	rxmlst/s
Average:	lo	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Average:	eth0	74.26	55.94	54199.50	38063.37	0.00	0.00	0.00

在清单 7.6 中，我们查看了系统中开放套接字的数量。我们可以看到开放套接字以及 TCP、RAW 和 UDP 套接字的总数。sar 还能显示 IP 数据包分片的数量。

清单 7.6

```
[ezolt@wintermute sysstat-5.0.2]$ sar -n SOCK 1 2
Linux 2.4.22-1.2174.nptlsmp (wintermute.phil.org)          06/07/04
21:32:26      totsck    tcpsck    udpsck    rawsck   ip-frag
21:32:27      373       118        8         0         0
21:32:28      373       118        8         0         0
Average:      373       118        8         0         0
```

sar 提供了对系统性能的一个很好的概览。但是，当我们要调查一个性能问题时，我们实际上想要了解的是哪些进程或服务消耗了特定的资源。sar 不会提供这方面的详细信息，但它确实让我们观察到了整个系统的网络 I/O 统计信息。

7.2.6 gkrellm

gkrellm 是一个图形化监视器，它使你能够观察到多种不同的系统性能统计信息。它为各种统计信息绘制图表，包括 CPU 使用情况、磁盘 I/O，以及网络使用情况。它可以通过“主题”来改变外观，甚至可以使用插件来监控默认版本中不包含的事件。

gkrellm 提供的信息与 sar、ip 和 ipconfig 类似，但与它们不同的是，它提供的是数据的图形视图。此外，它还提供流经特定 UDP 和 TCP 端口流量的有关信息。这是我们看到的第一个可以显示具有不同网络带宽消耗量的服务的工具。

7.2.6.1 网络 I/O 性能相关的选项

gkrellm 用如下命令行调用：

```
gkrellm
```

gkrellm 没有命令行选项用于配置其监控的统计信息。启动 gkrellm 之后，所有的配置都是图形化的。调出配置界面有两种方法：你可以右键点击 gkrellm 标题栏并选择 Configuration，或者当光标在窗口的任何位置时按下 F1。这两种操作都可以调出配置窗口（如图 7-1 所示）。

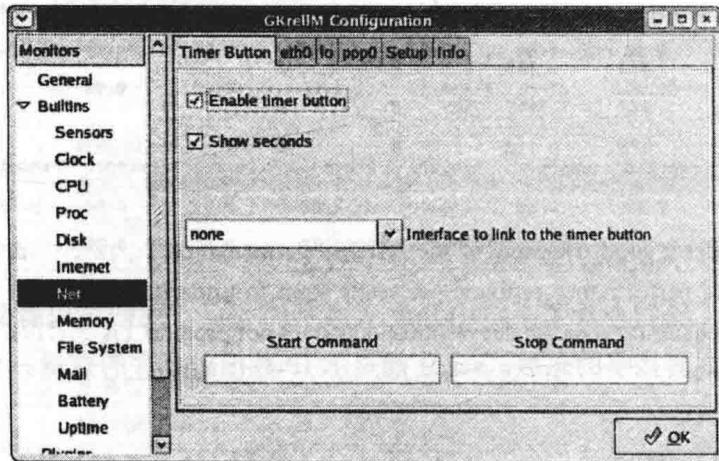


图 7-1

图 7-2 显示的是网络配置窗口。它用于配置哪些统计信息以及哪些服务显示在 gkrellm 的最终输出窗口。

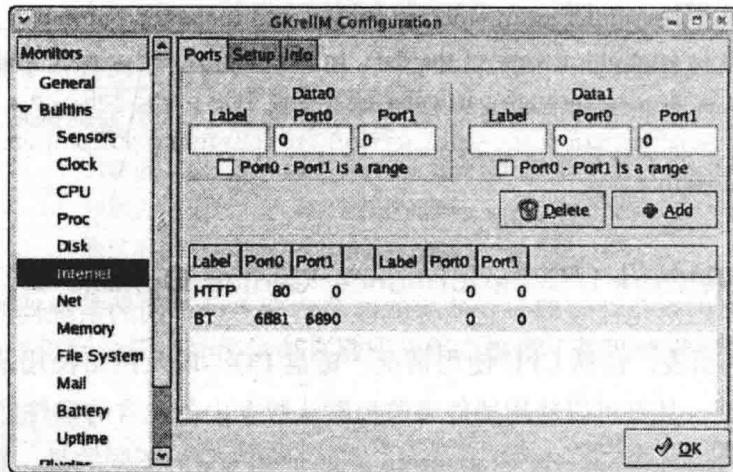


图 7-2

你可以将 gkrellm 配置为监控特定范围 TCP 端口的活动。这样你就能够监控服务，如 HTTP 或 FTP，使用的确切端口，并测量它们使用的带宽量。在图 7-2 中，我们将 gkrellm 配置为监控被 BitTorrent (BT) P2P 应用和 Web 服务器 (HTTP) 使用的端口。

gkrellm 是一个灵活而强大的图形化性能监控工具。它使你能够观察到当前系统的执行情况，以及其性能随时间的变化。使用 gkrellm 最困难的地方在于阅读小的默认文本。不过，gkrellm 的外观定制起来很容易，因此我们也可以推测这个缺点修正起来也比较容易。

7.2.6.2 用法示例

如前所述，gkrellm 可以监控多种不同类型的事件。在图 7-3 中，我们对输出进行了选

择，因此只显示了与网络流量及其使用有关的统计数据。

从图 7-3 中可以看到，顶部的两个图是端口的使用带宽（BT 和 HTTP），端口已经在配置部分进行了设置，底部的两个图则分别是两个设备（eth0 和 lo）的统计数据。图中可见，有少量 BitTorrent（BT）流量，但是没有 Web 服务器流量（HTTP）。以太网设备 eth0 之前有一些大的活动，但是现在已经平静下来。eth0 中较浅的阴影部分表示的是接收的字节数，而较深的阴影部分表示的是发送的字节数。

gkrellm 是一个功能强大的图形化工具，利用它可以一眼就判断出系统状态。

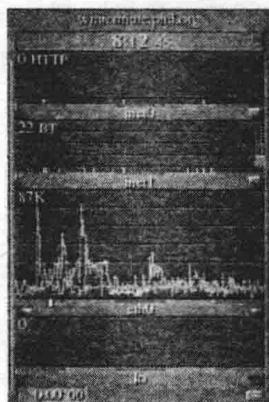


图 7-3

7.2.7 iptraf

iptraf 是一个实时网络监控工具。它提供了相当多的模式来监控网络接口和流量。iptraf 是一种控制台应用程序，但其用户界面则是基于光标的一组菜单和窗口。

与本章前面所述其他工具一样，iptraf 可以提供有关每个网络设备发送帧速率的信息。同时，它还能够显示 TCP/IP 数据包的类型和大小信息，以及哪些端口被用于网络流量。

7.2.7.1 网络 I/O 性能相关的选项

iptraf 用如下命令行调用：

```
iptraf [-d interface] [-s interface] [-t <minutes>]
```

如果调用 iptraf 时不带参数，就会显示一个菜单，让你选择监控界面以及想要监控的信息类型。表 7-5 对命令行选项进行了说明，这些选项用于观察特定接口或网络服务上的网络流量。

表 7-5 iptraf 命令行选项

选 项	说 明
-d interface	接口的详细统计信息，包括：接收信息、发送信息以及错误率信息
-s interface	关于接口上哪些 IP 端口正在被使用，以及有多少字节流经它们的统计信息
-t <minutes>	iptraf 退出前运行的分钟数

iptraf 还有更多模式和配置选项。详细信息请参阅其附带文档。

7.2.7.2 用法示例

当用如下命令行调用 iptraf 时，它创建的输出如图 7-4 所示：

```
[root@wintermute tmp]# iptraf -d eth0 -t 1
```

这条命令指定 iptraf 显示以太网设备 eth0 的详细信息并在运行 1 分钟后退出。此例中，

我们可以看到网络设备 eth0 的接收速率为 186.8kbps，发送速率为 175.5kbps。

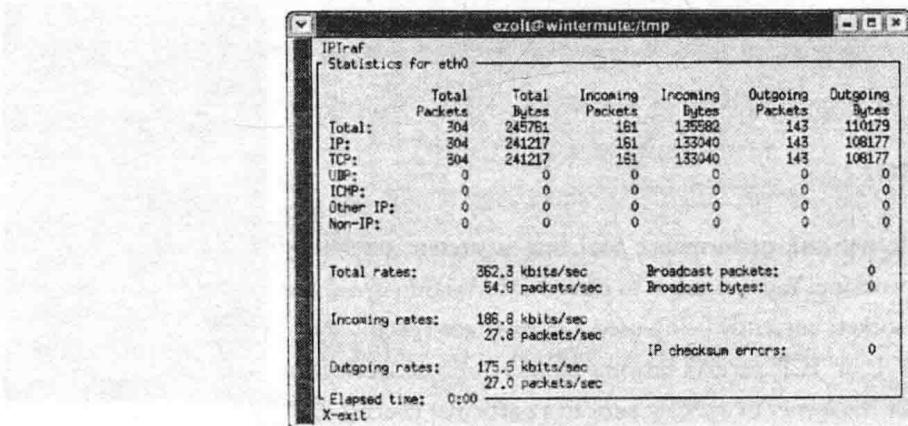


图 7-4

图 7-5 所示的是下一条命令，它要求 iptraf 显示每个 UDP 和 TCP 端口上的网络流量信息。调用命令如下：

```
[root@wintermute etherape-0.9.0]# iptraf -s eth0 -t 10
```

因为常用服务的 TCP 和 UDP 端口是固定的，所以，你可以利用这些信息来确定每个服务处理了多少流量。图 7-5 显示，有 29kb 的 HTTP 数据从 eth0 发送出来，有 25kb 则被其接收。

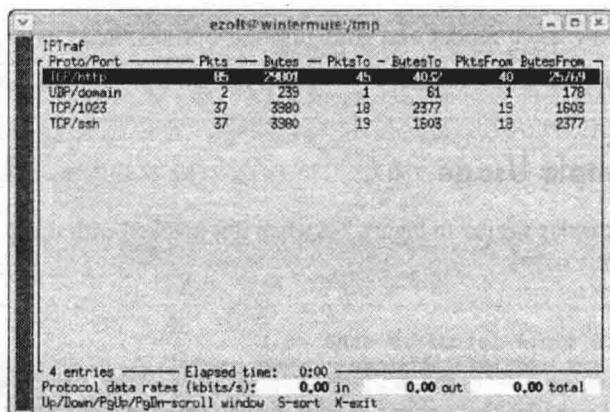


图 7-5

由于 iptraf 是基于控制台的应用程序，因此，它不需求 X 服务器或 X 服务器库。即使 iptraf 不能用鼠标来控制，它也是易于使用和配置的。

7.2.8 netstat

netstat 是一种基本的网络性能工具，它几乎出现在每一个联网的 Linux 机器上。可以用

它抽取的信息包括：当前正在使用的网络套接字的数量和类型，以及有关流入和流出当前系统的 UDP 和 TCP 数据包数量的特定接口统计数据。它还能将一个套接字回溯到其特定进程或 PID，这在试图确定哪个应用程序要对网络流量负责时是很有用的。

7.2.8.1 网络 I/O 性能相关的选项

netstat 用如下命令行调用：

```
netstat [-p] [-c] [-interfaces=<name>] [-s] [-t] [-u] [-w]
```

如果 netstat 调用时不带任何参数，它将显示系统范围内的套接字使用情况以及 Internet 域和 UNIX 域套接字的信息。（UNIX 域套接字用于本机的进程通信。）为了能检索所有其可以显示的统计信息，需要从根目录运行 netstat。表 7-6 中的命令行选项可以用于修改 netstat 显示信息的类型。

表 7-6 netstat 命令行选项

选 项	说 明
-p	给出打开每个被显示套接字的 PID/ 程序名
-c	每秒持续更新显示信息
--interfaces=<name>	显示指定接口的网络统计信息
--statistics -s	IP/UDP/ICMP/TCP 统计信息
--tcp -t	仅显示 TCP 套接字相关信息
--udp -u	仅显示 UDP 套接字相关信息
--raw -w	仅显示 RAW 套接字相关信息 (IP 和 ICMP)

netstat 还可以使用其他未在表中列出的命令行选项，更多信息参见 netstat 帮助手册。

7.2.8.2 用法示例

清单 7.7 要求 netstat 显示活跃的 TCP 连接并持续更新该信息。每一秒 netstat 都将显示新的 TCP 网络统计数据。netstat 不允许设置监控时长，因此如果被杀死或中断（Ctrl-C），它就只能停止。

清单 7.7

```
[root@wintermute ezolt]# netstat -t -c
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp      0      0 192.168.0.4:1023        fas.harvard.edu:ssh    ESTABLISHED
tcp      0      0 192.168.0.4:32844       216.239.39.147:http   TIME_WAIT
tcp      0      0 192.168.0.4:32843       216.239.39.147:http   TIME_WAIT
tcp      0      0 192.168.0.4:32853       skaiste.elektalit:http ESTABLISHED

Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp      0      0 192.168.0.4:1023        fas.harvard.edu:ssh    ESTABLISHED
```

tcp	0	0	192.168.0.4:32844	216.239.39.147:http	TIME_WAIT
tcp	0	0	192.168.0.4:32843	216.239.39.147:http	TIME_WAIT
tcp	0	0	192.168.0.4:32853	skaiste.elektal.lt:http	ESTABLISHED

清单 7.8 再次要求 netstat 显示 TCP 套接字的信息，但是，这一次我们还要求它给出与该套接字相关的程序。本例中，我们可以看到应用程序 SSH 和 mozilla-bin 发起了 TCP 连接。

清单 7.8

Active Internet connections (w/o servers)						
Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
tcp	0	0	192.168.0.4:1023	fas.harvard.edu:ssh	ESTABLISHED	1463/ssh
tcp	0	0	192.168.0.4:32844	216.239.39.147:http	TIME_WAIT	-
tcp	0	0	192.168.0.4:32843	216.239.39.147:http	TIME_WAIT	-
tcp	0	0	192.168.0.4:32853	skaiste.elektal.lt:http	ESTABLISHED	1291/mozilla-bin

清单 7.9 要求 netstat 提供启动后系统已接收的 UDP 流量统计信息。

清单 7.9

[root@wintermute ezolt]# netstat -s -u	
Udp:	
125 packets received	
0 packets to unknown port received.	
0 packet receive errors	
152 packets sent	

清单 7.10 要求 netstat 提供流经接口 eth0 的网络流量的相关信息。

清单 7.10

[root@wintermute ezolt]# netstat -interfaces=eth0											
Kernel Interface table											
Iface	MTU	Met	RX-OK	RX-ERR	RX-DRP	RX-OVR	TX-OK	TX-ERR	TX-DRP	TX-OVR	Flg
eth0	1500	0	52713	0	0	0	13711	1	0	1	BNRU

netstat 提供了大量的，与运行的 Linux 系统的套接字和接口相关的网络性能统计信息。它是唯一能将被使用套接字映射回其使用者进程 PID 的网络性能工具，因此，它是非常有用的。

7.2.9 etherape

etherape（基于 Windows 的网络工具 etherman 的双关语）为当前网络流量提供了可视化信息。默认情况下，它观察的是流经网络的全部网络流量，而不只是当前机器收发的那些

包。不过，它也可以被配置为仅显示当前机器的网络信息。

etherape（界面和文档）不够完美，但它提供了独一无二的视图来显示网络是如何连接的、被请求服务的类型，以及哪些节点请求了服务。在 etherape 创建的图中，节点代表的就是网络上的系统。通信的节点之间用线连接，节点间网络流量越大则线的规模也越大。当某个系统的网络使用量增加时，代表该系统的圆圈也会变大。不同系统之间的连线用不同的颜色来区分两者之间使用的通信协议。

7.2.9.1 网络 I/O 性能相关的选项

etherape 利用 libpcap 库来捕捉网络包，因此，它必须作为根用户运行。etherape 用如下命令行调用：

```
etherape [-n] [-i <interface name>]
```

表 7-7 解释了部分命令行选项，它们可以用来改变 etherape 监控的接口，或者决定是否在每个节点上显示解析主机名。

表 7-7 etherape 命令行选项

选 项	说 明
<code>-n, --numeric</code>	仅显示主机的 IP 号，不显示解析名
<code>-i, --interface=<interface name></code>	指定将要监控的接口

总之，etherape 的文档相当少。etherape 说明页给出了一些可以改变其外观和行为的命令行，但是，最好的学习方法就是使用它。一般说来，etherape 是网络可视化的相当不错的方法。

7.2.9.2 用法示例

图 7-6 显示的是 etherape 对一个相对简单的网络的监控。如果我们匹配一下协议的颜色

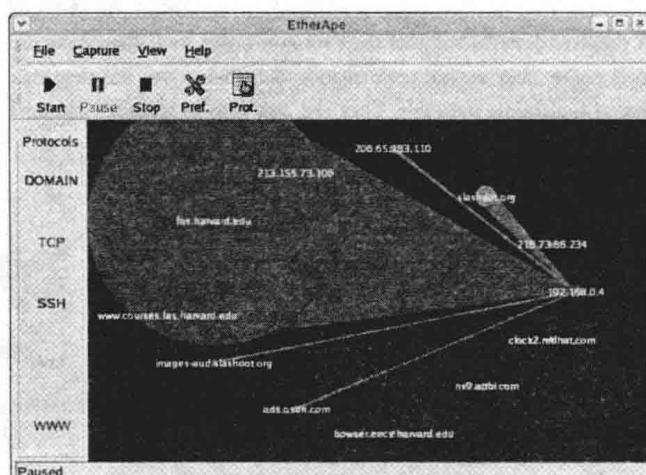


图 7-6

与最大圆圈的颜色，我们会发现该节点产生了大量的 SSH 流量。从图上来看，要确定哪个节点导致了这些 SSH 流量是很困难的。虽然没有显示，如果我们双击这个大圆圈，etherape 就会新建一个窗口来显示与该流量相关的节点的统计信息。我们可以用这些信息来调查网络流量的每个生成者以及它们的节点名称。

etherape 的输出会周期性的更新。如果网络流量发生了变化，则图形就会更新。观察网络流量发现它是如何使用的，以及如何随时间变化，是一件非常有趣的事情。

7.3 本章小结

本章的主要内容是如何使用 Linux 网络性能工具来监控从底层网络接口到高层应用，流经整个系统的网络流量。本章首先介绍的工具可以查询当前物理链接设置（mii - tool、ethtool），以及监控流经底层接口数据包的类型和数量（ifconfig、ip、sar、gkrellm、iptraf、netstat、etherape）。接着介绍的工具可以显示不同类型的 IP 流量（gkrellm、iptraf、netstat、etherape）和每种流量的数量（gkrellm、iptraf、etherape）。本章还介绍了一种工具（netstat）用来将 IP 套接字的使用映射到接收 / 发送每种类型流量的进程上。最后，本章给出了一个网络可视化工具，它可以将流经网络的数据类型和数量与其流经节点之间的关系可视化（etherape）。

下一章将会介绍几个常用的 Linux 工具，它们能让性能工具的使用变得更加容易。这些工具本身不是性能工具，但是，它们使得使用性能工具变得更加容易接受。同时，它们有助于把工具得到的结果进行可视化和分析，并且还可以把更多的重复性任务进行自动化。

实用工具：性能工具助手

本章介绍一些在 Linux 系统上可用的实用程序，它们能够加强性能工具的有效性和可用性。实用工具本身不是性能工具，但是当它们与性能工具一起使用时，它们可以帮助完成如下功能：自动执行繁琐的任务、分析性能统计数据，以及创建性能工具友好的应用程序。

阅读本章后，你将能够：

- 定期显示并收集性能数据（`bash`、`watch`）。
- 记录性能调查过程中所有的命令以及显示的输出（`tee`、`script`）。
- 导入、分析性能数据并将其图形化（`gnumeric`）。
- 确定应用程序使用的库（`ldd`）。
- 确定链接库中有哪些函数（`objdump`）。
- 研究应用程序的运行时特征（`gdb`）。
- 创建性能工具 / 调试友好的应用程序（`gcc`）。

8.1 性能工具助手

Linux 有着丰富的工具，这些工具一起使用比起单个使用之和的功能更强大。性能工具也是一样的，虽然可以单独使用，但是它们与其他 Linux 工具结合起来就能够显著提升其有效性和易用性。

8.1.1 自动执行和记录命令

如同前面章节所述，性能调查中最有价值的步骤之一就是保存在调查过程中发出的命令和产生的结果。这使得你可以在之后对它们进行回顾并寻求新的见解。为了帮助实现这个目的，Linux 提供了两个命令：tee 和 script，前者能将工具的输出保存为文件，后者能记录每一个按键和屏幕上的每一个输出。这些信息可以保存下来，便于之后查看或者创建脚本来自动执行测试。

自动执行命令很重要，因为它可以减少出错的机会，使你在思考问题时不需记住所有的细节。在你一次性键入又长又复杂的命令行之后，bash shell 和 watch 命令都可以让你周期性地自动执行这些命令。在你保证了命令行的正确性后，bash 和 watch 能够周期性地自动执行它们，不需要再次键入。

8.1.2 性能统计信息的绘图与分析

除了记录与自动化工具之外，Linux 还提供了强大的分析工具帮助你理解性能统计数据的含义。尽管大多数性能工具可以把性能统计数据输出为文本，但是想要发现其中的模式和随时间变化的趋势并不总是件容易的事儿。Linux 提供的 gnumeric 电子表格很强大，它可以对性能数据进行导入、分析和绘图。当你绘制数据图时，性能问题的原因可能会变得明晰，或者至少能揭示调查的新角度。

8.1.3 调查应用程序使用的库

还有一些 Linux 的工具能使你确定应用程序使用了哪些库，以及显示给定库提供的所有函数。ldd 命令给出一个特定应用程序使用的全部共享库的列表。在你想要跟踪被应用程序使用的库的数量和位置时，这个命令很有用。Linux 中还有一个命令 objdump，它可以在指定库或应用程序中搜索并显示其提供的全部函数。ltrace 只能给出一个应用程序调用函数的名称，但是结合命令 ldd 和 objdump，你就能够利用 ltrace 的输出来确定指定函数属于哪个库。

8.1.4 创建和调试应用程序

最后，Linux 还为你提供了能够创建性能工具友好型应用程序的工具，以及交互式调试和调查运行中应用程序属性的工具。GNU 编译器集（gcc）可以在应用程序中插入调试信息，以帮助 oprofile 找出某个具体性能问题对应的代码行和源文件。此外，GNU 调试器（gdb）还可以用来查找被各种性能工具默认不可得的应用程序的运行时信息。

8.2 工具

本节讨论的工具一起使用时，可以极大地提高前面章节介绍的性能工具的有效性和易用性。

8.2.1 bash

bash 是默认的 Linux 命令行 shell，在你每次与 Linux 命令行交互时，最有可能使用它。bash 通常有一个功能强大的脚本语言来创建 shell 脚本。不过这个脚本语言也可以从命令行调用，从而使你在性能调查过程中，能轻松地将一些比较繁琐的任务进行自动化。

8.2.1.1 性能相关的选项

bash 有一组命令可以一起使用，来周期性地运行特定命令。大多数 Linux 用户都把 bash 作为默认的 shell，因此，只要登录到一台机器或打开一个终端就会出现 bash 提示。如果你没有使用 bash，也可以键入 bash 来调用它。

有了 bash 命令提示符后，你可以输入一系列的 bash 脚本命令来自动连续地执行特定命令。在使用特定命令周期性提取性能统计数据的情况下，这个功能非常有用。表 8-1 给出了这些脚本选项。

表 8-1 bash 运行时脚本选项

选 项	说 明
<code>while condition</code>	条件为假时执行循环
<code>do</code>	表示循环开始
<code>done</code>	表示循环结束

bash 极其灵活，它记录在 bash 手册中。虽然 bash 非常复杂，但是在使用它之前并不需要完全掌握它。

8.2.1.2 用法示例

虽然有些性能工具，如 vmstat 和 sar，能周期性地更新性能统计信息，但是其他的命令，比如 ps 和 ifconfig 则不能。bash 可以调用诸如 ps 或 ifconfig 命令来周期性地显示它们的统计数据。例如，在清单 8.1 中，我们要求 bash 在条件为 true 时执行 while 循环。由于 true 命令总是为真，因此这个 while 循环永远都不会结束。接着，在 do 命令后启动每次迭代之后都要执行的命令，这些命令要求 bash 休眠 1 秒钟，然后运行 ifconfig 来抽取 eth0 控制器的性能信息。不过，我们只关心接收数据包，因此，我们用 grep 搜索并显示字符串为“RX packets”的 ifconfig 输出。最后，执行 done 命令来告诉 bash 循环完成。由于 true 命令总是返回真，所以，该循环将一直执行直到用组合键 <Ctrl-C> 终止它。

清单 8.1

```
[ezolt@wintermute tmp]$ while true; do sleep 1; /sbin/ifconfig eth0 | grep "RX packets"; done;
RX packets:2256178 errors:0 dropped:0 overruns:0 frame:0
RX packets:2256261 errors:0 dropped:0 overruns:0 frame:0
RX packets:2256329 errors:0 dropped:0 overruns:0 frame:0
RX packets:2256415 errors:0 dropped:0 overruns:0 frame:0
RX packets:2256459 errors:0 dropped:0 overruns:0 frame:0
...
```

利用清单 8.1 的 bash 脚本，可以查看到按秒更新的网络性能统计信息。同样的循环还可以用于监控其他的事件，只要将 ifconfig 命令修改为其他的命令即可，而通过修改休眠数值也能够改变更新时间间隔。这个简单的循环容易直接在命令行中键入，并且能够自动显示任何你感兴趣的性能统计数据。

8.2.2 tee

tee 是一个简单的命令，可以将命令的标准输出保存为文件并同时进行显示。在想要保存并同时查看性能工具输出的时候，tee 是很有帮助的。比如，正在监控一个实时系统的性能统计信息的同时，保存这些数据以备将来对它们进行分析。

8.2.2.1 性能相关的选项

tee 的调用命令行如下：

```
<command> | tee [-a] [file]
```

tee 获取由 <command> 提供的输出，在将其保存到指定文件的同时也显示到标准输出设备。如果特别指定了 -a 选项，则 tee 会将输出添加到文件上，而不是覆盖文件。

8.2.2.2 用法示例

清单 8.2 展示了用 tee 来记录 vmstat 的输出。如你所见，tee 显示了 vmstat 生成的输出，并同时将其保存到文件 /tmp/vmstat_out。保存 vmstat 的输出能让我们在将来的时间里对性能数据进行分析或绘图。

清单 8.2

```
[ezolt@localhost book]$ vmstat 1 5 | tee /tmp/vmstat_out
procs -----memory----- swap-- io---- system-- cpu-----
r b swpd free buff cache si so bi bo in cs us sy id wa
2 0 135832 3648 16112 95236 2 3 15 14 39 194 3 1 92 4
0 0 135832 4480 16112 95236 0 0 0 0 1007 1014 7 2 91 0
1 0 135832 4480 16112 95236 0 0 0 0 1002 783 6 2 92 0
0 0 135832 4480 16112 95236 0 0 0 0 1005 828 5 2 93 0
0 0 135832 4480 16112 95236 0 0 0 0 1056 920 7 3 90 0
```

`tee` 命令很简单，但由于它能轻松地记录指定性能工具的输出，因此它的能力也是非常强大的。

8.2.3 script

`script` 命令用于将一个 shell 会话过程中产生的全部输入和输出保存为文本文件。这个文本文件在将来既可以用来重现被执行的命令也可以用来查看结果。在调查性能问题时，准确记录被执行命令是很有用的，因为你可以在之后的时间里查看执行过的测试。拥有被执行命令的记录就意味着在调查不同的问题时，你还可以简单地对命令行进行剪切和粘贴。此外，记录性能结果也是很有帮助的，这样你就可以在将来查看这些记录以寻求发现和解决问题的新视角。

8.2.3.1 性能相关的选项

`script` 是一个相对简单的命令。在执行的时候，它会启动一个新的 shell，并记录下这个 shell 存续期间所有的键盘动作和输入，以及生成的输出，并将它们保存到文本文件。`script` 的调用命令行如下所示：

```
script [-a] [-t] [file]
```

默认情况下，`script` 把所有的输出都放到名为 `typescript` 的文件中，除非你特别指定其他的文件。表 8-2 给出了一些 `script` 的命令行选项。

表 8-2 `script` 命令行选项

选 项	说 明
<code>-a</code>	向文件添加脚本输出，而不是覆盖文件
<code>-t</code>	增加了计时信息，即每个输出 / 输入之间的时间量。该项输出显示的字符数，以及每组字符显示的时间间隔
<code>file</code>	输出文件的名称

提醒：`script` 字面上的意思是捕捉发送到屏幕的每一种类型的输出。但是，如果有彩色或加粗的输出，就会在输出文件中显示为 `esc` 字符。这些字符会明显让输出变得混乱，因此一般不怎么有用。不过，如果将 `TERM` 环境变量设置为 `dumb`（在基于 `csh` 的 shell 中用 `setenv TERM dumb`，在基于 `sh` 的 shell 中用 `export TERM=dumb`），应用程序就不会输出转义字符。这就使得输出具有更好的可读性。

此外，`script` 提供的计时信息也会扰乱输出。虽然自动生成计时信息是有用的，但是更方便的做法不是使用 `script` 的计时，而是直接用前面章节介绍的 `time` 命令来对重要命令进行计时。

8.2.3.2 用法示例

如前所述，如果将终端设置为 `dumb`，会得到可读性更好的 `script` 输出。其实现方法是

使用如下命令行：

```
[ezolt@wintermute manuscript]$ export TERM=dumb
```

然后，正式启动 script 命令。清单 8.3 显示的是用输出文件 ps_output 启动的 script。script 会持续记录会话，直到用 exit 命令或按下 <Ctrl-D> 组合键退出该 shell。

清单 8.3

```
[ezolt@wintermute manuscript]$ script ps_output
Script started, file is ps_output
[ezolt@wintermute manuscript]$ ps
  PID TTY      TIME CMD
 4285 pts/1    00:00:00 bash
 4413 pts/1    00:00:00 ps
[ezolt@wintermute manuscript]$ Script done, file is ps_output
```

接下来，在清单 8.4 中可以查看由 script 记录下的输出。如你所见，其中包含了所有的命令以及生成的全部输出。

清单 8.4

```
[ezolt@wintermute manuscript]$ cat ps_output
Script started on Wed Jun 16 20:43:35 2004
[ezolt@wintermute manuscript]$ ps
  PID TTY      TIME CMD
 4285 pts/1    00:00:00 bash
 4413 pts/1    00:00:00 ps
[ezolt@wintermute manuscript]$
Script done on Wed Jun 16 20:43:41 2004
```

script 是一个非常有用的命令，用来记录会话过程中所有的交互。与现代硬盘容量相比，script 产生的文件非常小。记录性能调查会话，并将其保存下来以备将来查看一直是一个很棒的主意。往最坏的情况考虑，它会浪费一点点精力和磁盘空间来记录会话。往最好的情况考虑，被保存的会话可以在之后的时间查看，而不需要重新运行该会话被记录下的命令。

8.2.4 watch

默认情况下，watch 命令会每秒运行一条命令并将其输出显示到屏幕上。与那些不能周期性地显示更新结果的性能工具一起工作时，watch 能发挥其作用。比如，有些工具，如 ifconfig 和 ps，显示的是当前性能统计数据，然后退出。由于 watch 能周期性地执行命令并显示其输出，因此，通过观看屏幕就可以发现哪些统计数据发生了变化，以及它们的变化速率。

8.2.4.1 性能相关的选项

`watch` 用如下命令行调用：

```
watch [-d[=cumulative]] [-n sec] <command>
```

如果调用的时候不带参数，`watch` 只会按秒显示给定命令的输出，直到你中断这个过程。默认输出通常很难发现一屏信息与另一屏信息的差异，因此，`watch` 提供了选项来突出显示每个输出之间的不同。这样更容易发现每个采样之间的输出差异。表 8-3 对 `watch` 可接受命令行选项进行了说明。

表 8-3 `watch` 命令行选项

选 项	说 明
<code>-d[=cumulative]</code>	突出显示样本之间变化的输出。如果使用了选项 <code>cumulative</code> ，那么一个域突出显示的条件是只要曾经有过变化即可，而不是要求它在样本间发生变化
<code>-n sec</code>	更新等待的秒数

`watch` 是一种强大的工具，用于查看性能统计数据如何随时间变化。它并不复杂，但是却能很好地完成自己的工作。它真正填补了使用某些性能工具所造成的空白，这些工具无法周期性地显示已更新的输出。在使用这些工具时，你可以用窗口形式运行 `watch`，通过定期查看该窗口来了解统计信息是如何变化的。

8.2.4.2 用法示例

清单 8.5 中的第一个例子展示了与 `ps` 命令一起使用的 `watch`。我们要求 `ps` 给出每个进程产生的一般故障的数量。`watch` 每 10 秒清除屏幕，并更新该信息。请注意，可能需要为你要求执行的命令加上引号，这样，`watch` 就不会将你想要执行的命令的选项与它自身的选项搞混。

清单 8.5

```
[ezolt@wintermute ezolt]$ watch -n 10 "ps -o minflt,cmd"

Every 10s: ps -o minflt,cmd
Wed Jun 16 08:33:21 2004
MINFLT CMD
1467 bash
41 watch -n 1 ps -o minflt,cmd
66 ps -o minflt,cmd
```

`watch` 作为一种工具，其基本功能可以很容易地编写为简单的 shell 脚本。但是，`watch` 比使用 shell 脚本更简单，因为它几乎总是可用的，并且能正确地工作。别忘了，有些性能工具，如 `ifconfig` 或 `ps` 只能一次性地显示统计数据，而 `watch` 却能很容易地跟踪（只需看上一眼）统计数据的变化。

8.2.5 gnumeric

在进行性能问题调查时，性能工具常常会生成庞大的性能统计数据。有些时候，梳理这些数据并从中找出能够表明系统运行情况的趋势与模式就成了一个问题。通常电子表格，尤其是 gnumeric，能够从三个不同的方面使得这个任务变得容易实现。首先，gnumeric 提供了内置函数，比如求最大值、最小值、平均值和标准偏差，这就使你能对性能数据进行数值分析。其次，gnumeric 提供了灵活的方式来导入许多性能工具常用输出的表格文本数据。最后，gnumeric 还有一个强大的绘图工具，可以将性能工具生成的性能数据可视化。这不但对于探寻较长时间内的数据趋势是极有价值的，而且对于寻找不同类型数据之间的相关性（比如磁盘 I/O 与 CPU 使用量之间的相关性）也是相当有用的。通常，要从文本输出中找出模式是比较困难的，但对图形来说，系统行为就会显得更加清晰。其他的电子表格，如 OpenOffice 的 oocalc，也可以使用，不过 gnumeric 强大的文本导入器和绘图工具使得它成为最容易被使用的工具。

8.2.5.1 性能相关的选项

使用电子表格帮助实现性能分析，只需完成以下步骤：

1. 将性能数据保存到文本文件。
2. 将文本文件导入到 gnumeric。
3. 分析数据或用数据绘图。

gnumeric 可以生成多种不同类型的图形，并且提供了多种不同的函数实现数据分析。了解 gnumeric 的强大功能和灵活性的最好方法就是加载一些数据进行实验。

8.2.5.2 用法示例

要展示 gnumeric 的实用性，首先必须生成用于绘图和分析的性能数据。清单 8.6 要求 vmstat 生成 100 秒的输出，并将该信息保存到文本文件 vmstat_output 中。数据将会被加载到 gnumeric。-n 选项表示 vmstat 只显示头信息一次（而不是每屏信息都显示）。

清单 8.6

```
[ezolt@nohs ezolt]$ vmstat -n 1 100 > vmstat_output
```

接下来，用如下命令启动 gnumeric：

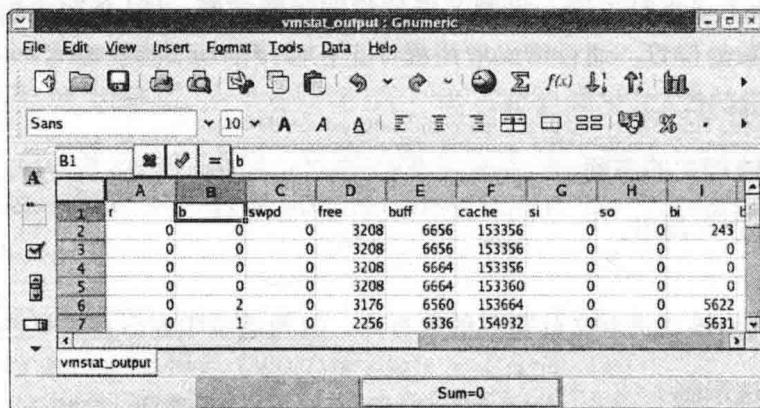
```
[ezolt@nohs ezolt]$ gnumeric &
```

这会打开一个空白电子表格用于导入 vmstat 数据。

在 gnumeric 中选择 File>Open，弹出一个对话框（图中未显示）用于选择打开的文件及该文件的类型。我们选择 Text Import (Configurable) 作为文件类型，然后通过一系列的向导对话框来选择将 vmstat_output 文件的那一列映射到电子表格的那一列。对 vmstat 而言，

最好从第二行文本开始导入，因为第二行包含了列名称以及每一列适合的大小。此外，为导入数据选择 Fixed-Width 也是很有用的，其原因是 vmstat 就是这样输出数据的。在成功导入数据后，我们就会看到如图 8-1 所示的电子表格。

接下来，利用导入的数据绘图。在图 8-2 中，我们用不同的 CPU 使用情况（us、sys、id、wa）创建了一个叠式图。由于这些数据的总和始终为 100%（或接近该值），因此，每次都能看出哪种状态处于主导地位。本例中，系统在大部分时间都是空闲的，不过在图中 1/4 的部分出现了大量的等待时间。



The screenshot shows a Gnumeric spreadsheet window titled "vmstat_output". The data is presented in a table with columns labeled A through I. The first few rows of data are:

	A	B	C	D	E	F	G	H	I
1	r	b	swpd	free	buf	cache	si	so	bi
2	0	0	0	3208	6656	153356	0	0	243
3	0	0	0	3208	6656	153356	0	0	0
4	0	0	0	3208	6664	153356	0	0	0
5	0	0	0	3208	6664	153360	0	0	0
6	0	2	0	3176	6560	153664	0	0	5622
7	0	1	0	2256	6336	154932	0	0	5631

图 8-1

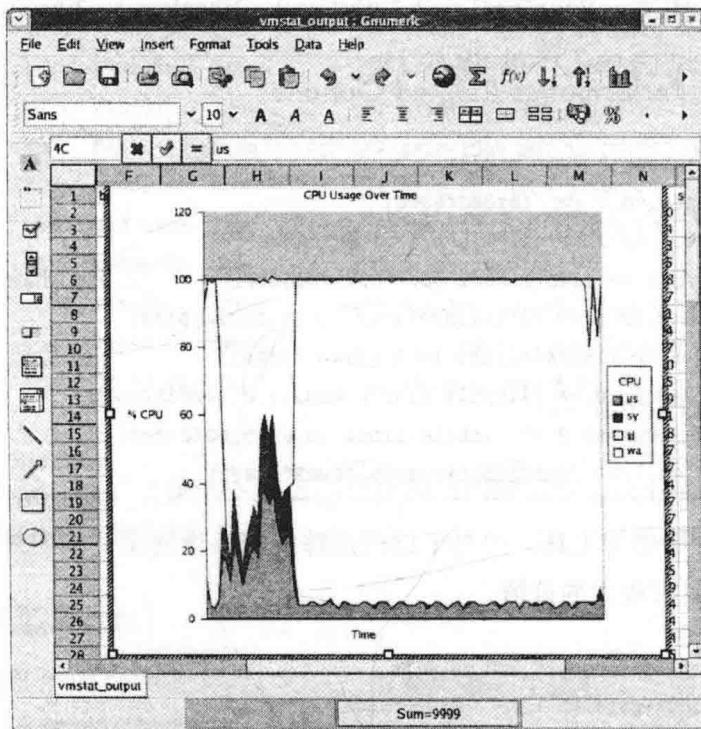


图 8-2

图形是查看一个测试单次运行时，其性能统计数据随时间变化情况的有力手段。同时，对于发现不同运行的对比差异也是很有帮助的。用不同运行得到的数据绘图时，要确保每个图使用的是相同的比例，这能让数据的比较对照更加容易。

gnumeric 是一种轻量级的应用程序，它使你能快速简便地导入、绘图 / 分析大量的性能数据。它是一种很棒的工具，通过处理性能数据来探寻是否出现了任何有趣的特性。

8.2.6 ldd

ldd 可以被用来显示特定的二进制文件依赖的是哪个库。**ldd** 有助于跟踪一个应用程序可能使用的库函数的位置。通过给出应用程序正在使用的所有库，就可以对它们进行搜索，找出包含给定函数的库。

8.2.6.1 性能相关的选项

ldd 用如下命令行调用：

```
ldd <binary>
```

接着 **ldd** 会列出该二进制文件需要的所有库，以及系统中有哪些文件能实现这些需求。

8.2.6.2 用法示例

清单 8.7 展示的是 **ldd** 用在二进制文件 **ls** 上。在这个特定的例子中，我们可以发现 **ls** 使用了如下这些库：**linux-gate.so.1**，**librt.so.1**，**libacl.so.1**，**libsSelinux.so.1**，**libc.so.6**，**libpthread.so.0**，**ld-linux.so.2** 和 **libattr.so.1**。

清单 8.7

```
[ezolt@localhost book]$ ldd /bin/ls
    linux-gate.so.1 =>  (0x000dfe000)
    librt.so.1 => /lib/tls/librt.so.1 (0x0205b000)
    libacl.so.1 => /lib/libacl.so.1 (0x04983000)
    libsSelinux.so.1 => /lib/libsSelinux.so.1 (0x020c0000)
    libc.so.6 => /lib/tls/libc.so.6 (0x0011a000)
    libpthread.so.0 => /lib/tls/libpthread.so.0 (0x00372000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00101000)
    libattr.so.1 => /lib/libattr.so.1 (0x03fa4000)
```

ldd 是一个相对简单的工具，但对于试图追踪应用程序使用了哪些库以及这些库在系统中的位置来说，它具有极大的价值。

8.2.7 objdump

对于分析二进制文件和库的各个方面来说，**objdump** 是一种复杂而强大的工具。尽管它

有许多其他的功能，它可以被用来确定给定的库提供了哪些函数。

8.2.7.1 性能相关的选项

objdump 用如下命令行调用：

```
objdump -T <binary>
```

如果调用的时候使用了 -T 选项，则它将显示该库 / 二进制文件所依赖或提供的全部符号。这些符号可以是数据结构，也可以是函数。包含 .text 的每一行 objdump 输出都是该二进制文件提供的一个函数。

8.2.7.2 用法示例

清单 8.8 展示的是 objdump 用于分析库 gtk。因为我们感兴趣的只有 libgtk.so 提供的符号，所以用 fgrep 对输出进行选择，仅输出那些包含 .text 的行。本例中，我们可以看到 libgtk.so 提供的一些函数，包括 gtk_arg_values_equal、gtk_tooltips_set_colors 和 gtk_viewport_set_hadjustment。

清单 8.8

```
[ezolt@localhost book]$ objdump -T /usr/lib/libgtk.so | fgrep .text
0384eb60 l    d  .text  00000000
0394c580 g    DF  .text  00000209  Base      gtk_arg_values_equal
0389b630 g    DF  .text  000001b5  Base      gtk_signal_add_emission_hook_full
0385cdf0 g    DF  .text  0000015a  Base      gtk_widget_restore_default_style
03865a20 g    DF  .text  000002ae  Base      gtk_viewport_set_hadjustment
03929a20 g    DF  .text  00000112  Base      gtk_clist_columns_autosize
0389d9a0 g    DF  .text  000001bc  Base      gtk_selection_notify
03909840 g    DF  .text  000001a4  Base      gtk_drag_set_icon_pixmap
03871a20 g    DF  .text  00000080  Base      gtk_tooltips_set_colors
038e6b40 g    DF  .text  00000028  Base      gtk_hseparator_new
038eb720 g    DF  .text  0000007a  Base      gtk_hbutton_box_set_layout_default
038e08b0 g    DF  .text  000003df  Base      gtk_item_factory_add_foreign
03899bc0 g    DF  .text  000001d6  Base      gtk_signal_connect_object_while_alive
....
```

在使用性能工具时，它们会显示一个应用程序调用的库函数（而不是库本身），objdump 能帮助找到每个函数所在的共享库。

8.2.8 GNU 调试器 (gdb)

gdb 是一个很棒的应用程序调试器，它可以帮助调查一个正在运行的应用程序的多个不同方面。gdb 具备三个特性使得它对诊断性能问题来说非常有价值。第一，gdb 可以附加到

当前正在运行的进程。第二, `gdb` 可以展示该进程的回溯, 即显示当前源代码行和调用树。附加到进程并抽取其回溯可以迅速找出一些比较明显的性能问题。但是, 如果应用程序不是卡在单点上, 那么使用 `gdb` 就难以进行问题诊断, 此时, 系统级的分析器, 如 `oprofile`, 将会是一个更好的选择。第三, `gdb` 可以将虚拟地址映射回特定的函数。与性能工具相比, `gdb` 更擅长计算虚拟地址的位置。例如, 如果 `oprofile` 给出了事件发生的虚拟地址而非函数名, 那么 `gdb` 就可以计算出该地址的函数。

8.2.8.1 性能相关的选项

`gdb` 用如下命令行调用, 其中, `pid` 是指 `gdb` 将要附加的进程:

```
gdb -p pid
```

`gdb` 附加到该进程后, 它就进入到交互模式, 这时就可以检查给定进程的当前执行位置和运行时变量。表 8-4 对其中的一条命令进行了说明, 可用其检查正在运行的进程。

表 8-4 `gdb` 运行时选项

选 项	说 明
<code>bt</code>	展示当前正在执行的进程回溯

`gdb` 还有很多命令行选项和运行时控件, 它们更适合于调试而不是性能调查。获取更多信息请参见 `gdb` 手册页或在 `gdb` 提示中键入 `help`。

8.2.8.2 用法示例

想要研究 `gdb` 是如何工作的, 可以先在一个简单的测试应用程序上演示它。清单 8.9 中的程序在 `main` 中只调用了函数 `a()`, 并陷入了一个无限循环。这个程序不会结束, 因此当我们把 `gdb` 附加上去后, 它将会一直执行函数 `a()` 的无限循环。

清单 8.9

```
void a(void)
{
    while(1);
}

main()
{
    a();
}
```

清单 8.10 启动应用程序并用 `gdb` 附加到它的 `pid`。我们要求 `gdb` 产生一个回溯, 以便展示当前正在执行的究竟是哪条代码, 以及哪组函数调用会导致当前的位置。如同预期的, `gdb` 显示出正在执行的是无限循环 `a()`, 它是由 `main()` 调用的。

清单 8.10

```
[ezolt@wintermute examples]$ ./chew &
[2] 17389
[ezolt@wintermute examples]$ gdb -p 17389
GNU gdb Red Hat Linux (5.3.90-0.20030710.41rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "i386-redhat-linux-gnu".
Attaching to process 17389
Reading symbols from /usr/src/perf/utils/examples/chew...done.
Using host libthread_db library
"/lib/tls/libthread_db.so.1".
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
a () at chew.c:3
3      while(1);
(gdb) bt
#0  a () at chew.c:3
#1  0x0804832f in main () at chew.c:8
```

最后，在清单 8.11 中，我们要求 gdb 给出虚拟地址 0x0804832F 的位置，而 gdb 显示该地址是函数 main 的一部分。

清单 8.11

```
(gdb) x 0x0804832f
0x804832f <main+21>: 0x9090c3c9
```

gdb 是一个极其强大的调试器，可以帮助调查性能问题。如果你想要知道特定代码路径发生的确切原因，那么 gdb 甚至在性能问题已经确定之后也能够发挥作用。

8.2.9 gcc (GNU 编译器套件)

gcc 是 Linux 系统中最流行的编译器。与所有的编译器一样，gcc 需要源代码（如 C、C++ 或 Objective-C）生成二进制代码。它提供了多个选项不仅可以对得到的二进制代码进行优化，还能让应用程序的性能跟踪变得更容易。本书不涉及 gcc 性能优化的详细内容，但是如果想要提高应用程序的性能，你就应该研究一下这些内容。gcc 提供的性能优化

选项通过多种优化来调整已编译的二进制文件的性能，这些优化包括：架构通用优化（使用 -O1、-O2、-O3），特定架构优化（-march 和 -mcpu），以及基于反馈的优化（使用 -fprofile-arcs 和 -fbranch-probabilities）。更多的优化选项详情请参阅 gcc 手册页。

8.2.9.1 性能相关的选项

gcc 最基本的调用格式如下所示：

```
gcc [-g level] [-pg] -o prog_name source.c
```

gcc 有数量庞大的选项来影响它对应用程序的编译。如果你有勇气的话，可以到 gcc 手册页上查阅它们。表 8-5 给出了有助于性能调查的具体选项。

表 8-5 gcc 命令行选项

选 项	说 明
-g[1 2 3]	-g 选项向二进制文件添加调试信息，默认级别为 2。如果指定级别，gcc 将会调整保存在二进制文件中的调试信息量。级别 1 只提供了产生回溯所需的信息，没有关于源代码行与特定代码行的映射信息。级别 3 比级别 2 提供的信息更多，比如源代码中的宏定义
-pg	开启应用程序分析

许多性能调查工具，如 oprofile，需要用调试信息编译应用程序，以便将性能信息映射回特定的应用程序源代码行。如果没有调试信息，它们一般也还是可以工作，但是如果启动调试，那么它们将会提供更丰富的信息。应用程序分析的更多信息参见前面的章节。

8.2.9.2 用法示例

理解 gcc 可以提供的调试信息类型的最好方法可能就是看一个简单的例子。清单 8.12 中有一个 C 应用程序的源代码 deep.c，该程序仅调用了一组函数，然后根据传递的数字输出一定数量的字符串“hi”。程序的 main 函数调用函数 a()，函数 a() 调用函数 b()，然后输出“hi”。

清单 8.12

```
void b(int count)
{
    int i;
    for (i=0; i<count;i++)
        {printf("hi\n");}
}

void a(int count)
{
    b(count);
}
```

```
int main()
{
    a(10);
}
```

首先，如清单 8.13 所示，编译该程序时不带任何调试信息。在调试器中启动该程序，在函数 b() 上添加一个断点。当程序运行时，它会在函数 b() 处暂停，并请求回溯。gdb 可以弄清楚回溯，但是它并不知道函数之间传递了什么样的值或者函数存在于原始源代码文件中的什么位置。

清单 8.13

```
[ezolt@wintermute utils]$ gcc -o deep deep.c

[ezolt@wintermute utils]$ gdb ./deep
...
(gdb) break b
Breakpoint 1 at 0x804834e
(gdb) run
Starting program: /usr/src/perf/utils/deep
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804834e in b ()
(gdb) bt
#0  0x0804834e in b ()
#1  0x08048389 in a ()
#2  0x080483a8 in main ()
```

清单 8.14 启动调试信息对同样的应用程序进行编译。现在，当运行 gdb 并产生回溯时，我们可以看到每个函数调用传递的数值，以及特定代码行所驻留的准确的源代码行。

清单 8.14

```
[ezolt@wintermute utils]$ gcc -g -o deep deep.c
[ezolt@wintermute utils]$ gdb ./deep
...
(gdb) break b
Breakpoint 1 at 0x804834e: file deep.c, line 3.
(gdb) run
Starting program: /usr/src/perf/utils/deep

Breakpoint 1, b (count=10) at deep.c:3
3          for (i=0; i<count;i++)
(gdb) bt
```

```
#0  b (count=10) at deep.c:3
#1  0x08048389 in a (count=10) at deep.c:9
#2  0x080483a8 in main () at deep.c:14
```

调试信息会显著增加 gcc 最终生成的可执行文件的大小。但是，在追踪性能问题时，由其提供的信息却是无价的。

8.3 本章小结

本章给出了对调查性能问题有用的各种 Linux 实用工具的集合。首先介绍的工具如 bash、watch、tee 和 script 能自动显示和收集性能数据。之后介绍了 gnumeric，该工具可以对基于文本的性能工具所得到的结果进行绘图和分析。之后研究了 ldd 和 objdump，它们可用于发现一个函数属于哪个库。接着本章描述了 gdb，它能够被用来调查当前正在运行的应用程序的执行和运行时信息。最后，本章给出了 gcc，该工具可生成带符号调试信息的二进制文件，这可以帮助其他性能工具，如 oprofile，将事件映射回一个特定的源代码行。

在之后的章节中，我们将集合迄今为止提出的所有工具，以解决一些现实的性能问题。

使用性能工具发现问题

本章主要介绍综合运用之前提出的性能工具来缩小性能问题产生原因的范围。

阅读本章后，你将能够：

- 启动行为异常的系统，使用 Linux 性能工具追踪行为异常的内核函数或应用程序。
- 启动行为异常的应用程序，使用 Linux 性能工具追踪行为异常的函数或源代码行。
- 追踪 CPU、内存、磁盘 I/O 和网络的过度使用情况。

9.1 并非总是万灵药

本章假设可以通过改变软件来解决性能问题。通过对应用程序或系统调优来达到性能目标并非总是可行的。如果调优失败，就可能要求进行硬件升级或更换。如果系统容量达到极限，那么性能调优就只能起到一定的作用。

举例来说，升级系统内存容量可能是必要的（或者是更便宜的），而不是追踪哪个应用程序在使用系统内存，然后对它们进行调整以降低其使用量。只升级系统硬件而非追踪并调整特定的性能问题，这个决定依赖于问题本身，同时，它也是进行调查的个人的价值判断。它实际上取决于哪种选择更加便宜，是（问题调查的）时间方面，还是（购买新硬件的）经费方面。最后，在某些情况下，调优将是首选或唯一的选择，这就是本章要描述的内容。

9.2 开始追踪

当你决定在 Linux 上优化某些东西之后，你首先要确定的是要优化什么。本章使用的

方法覆盖了一些比较常见的性能问题，通过举例来说明如何利用前面介绍的工具共同解决这些问题。接下来的几个小节将帮助引导你找到性能问题的成因。在很多小节中会先要求运行各种性能工具，然后根据结果跳转到本章的其他小节。这有助于找到问题的根源。

就像在前面章节里陈述的一样，保存每次执行的测试结果是一个好办法。这使你能在之后的时间查看结果，假如调查结果尚不能确定，还可以将结果发送给其他人。

现在开始。

调查问题的时候，初始系统运行的无关程序越少越好，因此，关闭或终止任何不需要的应用程序或进程。一个干净的系统有助于消除由任何无关应用程序可能导致的混淆干扰。

如果某个特定的应用或程序没有按预期执行，直接跳到 9.3 节。如果不是某个应用程序性能不好，而是整个 Linux 系统执行效果不如预期，则跳到 9.4 节。

9.3 优化应用程序

优化应用程序时，其执行的多个方面有可能出现问题。本节将根据你发现的问题，将你引导到正确的小节。

图 9-1 展示了优化应用程序的步骤。

诊断从 9.3.1 节开始。

9.3.1 内存使用有问题？

使用 top 或 ps 确定应用程序使用了多少内存。如果该程序消耗的内存量超过预期，转到 9.6.6 节。否则，继续见 9.3.2 节。

9.3.2 启动时间有问题？

如果应用程序启动所花费的时间有问题，见 9.3.3 节。否则，转到 9.3.4 节。

9.3.3 加载器引入延迟了吗？

要测试问题是否出在加载器上，就按前面章节所述来设置 ld 环境变量。如果 ld 统计数据显示在映射所有的符号时有明显的延迟，那么就尽量减少应用程序使用的库的数量和

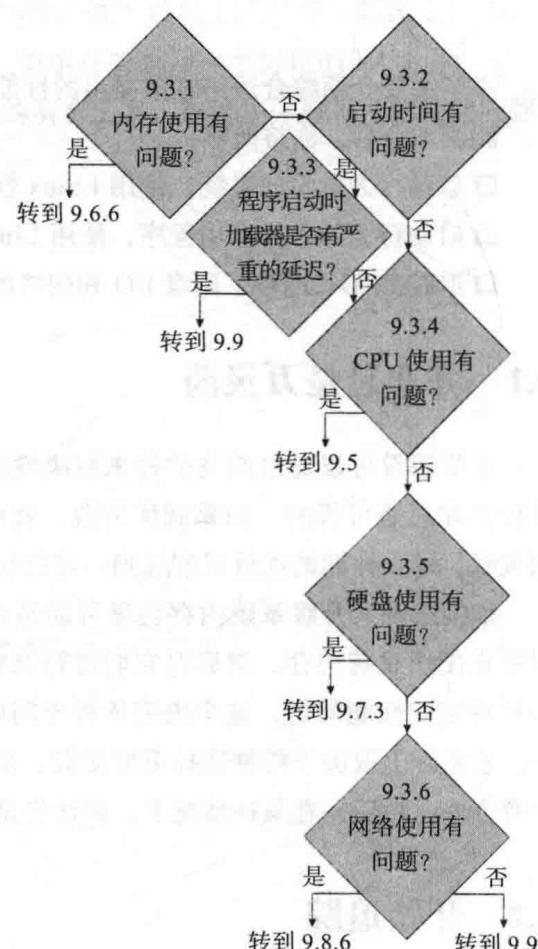


图 9-1

大小，或者尽量预链接库。

如果加载器确实表现出有问题，转到 9.9 节。如果没有问题，继续见 9.3.4 节。

9.3.4 CPU 使用（或完成时长）有问题？

用 top 或 ps 来确定应用程序的 CPU 使用量。如果应用程序是 CPU 消耗大户，或其完成时间特别长，那么该程序就存在 CPU 使用问题。

通常，一个应用程序的不同部分会具有不同的性能表现。那么就可能需要隔离那些性能不佳的部分，这样使用性能工具时就不用测量那些对性能没有负面影响的部分，而只需要测量被隔离部分的性能统计数据。为此，可能要改变应用程序的行为，使之易于分析。如果应用程序某个特定的部分对性能非常重要，那么在测量整个应用程序的性能统计信息时，要么试着只测量关键部分执行时的性能统计数据，要么就让该部分运行相当长的时间，直到应用程序无关部分的数据在与之不相干的整个性能统计数据中只占一小部分。尽量减少应用程序的工作，以便它只执行对性能至关重要的功能。比方说，在收集一个应用程序整体运行的性能统计信息时，我们不希望启动和退出过程占据大部分的应用程序运行时的总时间量。在这种情况下，比较有效的方法是启动应用程序，多次运行时间消耗大的部分，然后立即退出程序。这样分析工具（如 oprofile 或 gprof）就可以捕获运行缓慢代码的更多信息，而不是那些执行了但却与问题无关部分（如启动和退出）的信息。比这个方法更好的是修改应用程序的源代码，当应用程序启动时，自动运行耗时部分，然后退出程序。这有助于最大程度减少与特定性能问题无关的分析数据。

如果应用程序的 CPU 使用有问题，转到 9.5 节。如果没有问题，见 9.3.5 节。

9.3.5 应用程序的磁盘使用有问题？

如果已经知道应用程序会导致大量的磁盘 I/O，转到 9.7.3 节以确定它访问了哪些文件。否则，见 9.3.6 节。

9.3.6 应用程序的网络使用有问题？

如果已经知道应用程序会导致大量的网络 I/O，转到 9.8.6 节。

如果上述问题均不存在，你遇到的应用程序性能问题可能就不在本书范围之内，请转到 9.9 节。

9.4 优化系统

有时候，处理一个行为异常的系统，找出究竟是什么拉低了每件事情的速度是很重要的。

由于调查的是系统级的问题，其原因可能存在于任何地方，从用户应用程序到系统库，再到Linux内核。幸运的是，与很多别的操作系统不同，对Linux来说，即使不能得到系统上所有应用程序的源代码，也可以获取其中大多数的源代码。如果有必要的话，你可以修复这个问题并将其提交给该部分的维护者。最糟糕的情况也不过是在本地运行已修复的版本。这就是开源软件的力量。

图 9-2 展示的是诊断系统级性能问题的流程。

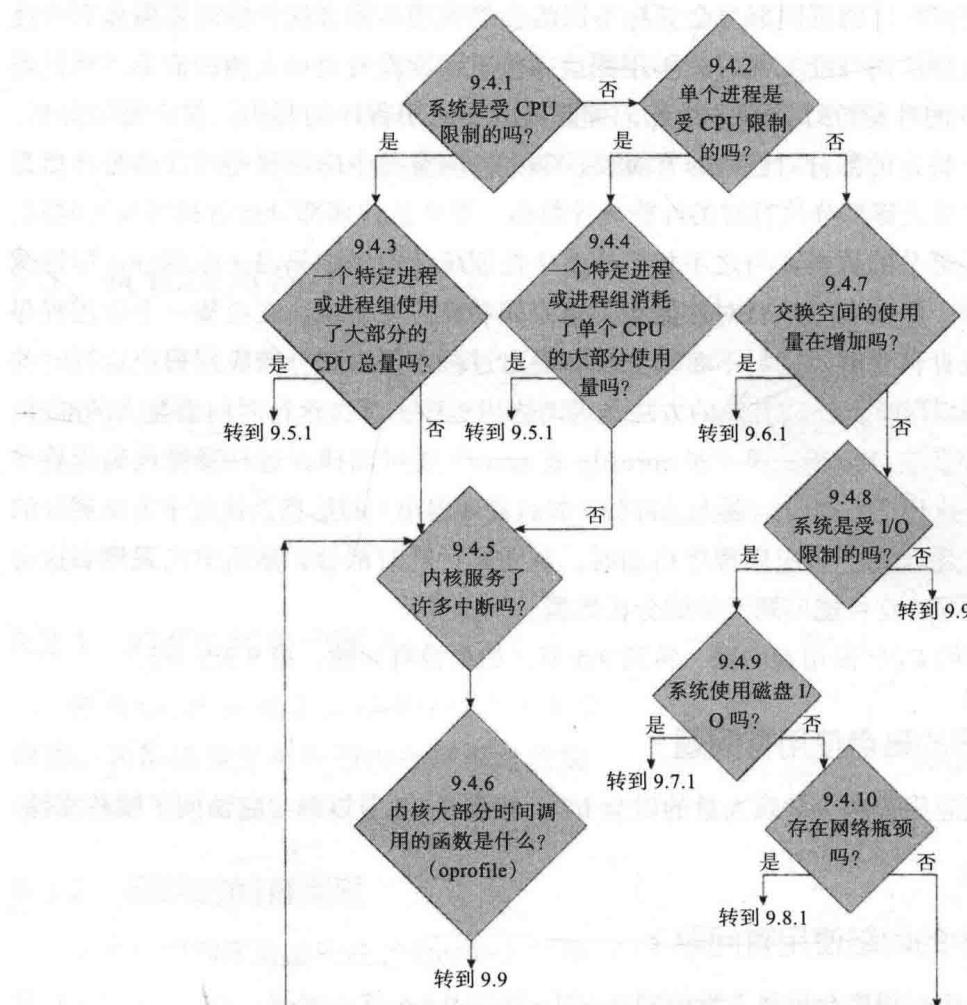


图 9-2

调查从 9.4.1 节开始。

9.4.1 系统是受CPU限制的吗？

使用 top、procinfo 或 mpstat 来确定系统在哪些地方消耗了时间。如果整个系统空闲和等待时间的比例不足全部时间的 5%，那么该系统就是受CPU限制的。转到 9.4.3 节。否

则，前进到 9.4.2 节。

9.4.2 单个进程是受 CPU 限制的吗？

虽然系统作为一个整体可能不是受 CPU 限制的，但在一个对称多处理 (SMP) 或超线程系统中，单个处理器可能是受 CPU 限制的。

使用 top 或 mpstat 来确定单个 CPU 的空闲和等待时间是否少于 5%。如果是，那么一个或多个 CPU 就是受 CPU 限制的，对这种情况，转到 9.4.4 节。

否则，各处理器都不是受 CPU 限制的，则转到 9.4.7 节。

9.4.3 一个或多个进程使用了大多数的系统 CPU 吗？

下一步是要找出是否有特定应用程序或应用程序组使用了 CPU。最简单的方法是运行 top。默认情况下，top 按 CPU 使用量的降序来排列进程。top 按照该进程消耗的用户时间和系统时间总和来报告进程的 CPU 使用量。举个例子，如果一个应用程序在用户空间代码上消耗了 20%CPU 时间，在系统代码上消耗了 30%CPU 时间，那么，top 将会报告该进程消耗了 50%CPU 时间。将所有进程的 CPU 时间加起来，如果这个时间明显少于整个系统的系统时间加用户时间，那么内核所做的重要工作就与应用程序无关，转到 9.4.5 节。

否则，每个进程都转 9.5.1 节一次，以便确定时间是在哪里消耗的。

9.4.4 一个或多个进程使用了单个 CPU 的大多数时间？

下一步是要找出是否有特定应用程序或应用程序组使用了单个 CPU。实现该目标最简单的方法是运行 top。默认情况下，top 按 CPU 使用量的降序来排列进程。在报告进程的 CPU 使用量时，top 显示的是应用程序使用的总 CPU 和系统时间。举个例子，如果应用程序在用户空间代码上消耗了 20% 的 CPU，在系统代码上消耗了 30% 的 CPU 时间，那么，top 将会报告该应用程序消耗了 50% 的 CPU 时间。

首先，运行 top，然后将最后一个 CPU 添加到 top 显示的字段中。打开 Irix 模式，以便 top 显示每个处理器使用的 CPU 时间总量而不是整个系统的 CPU 时间。对于每个利用率高的处理器，将其上运行的特定应用程序或多个应用程序的 CPU 时间加起来。如果在一个 CPU 上，应用程序时间总和低于内核加用户时间之和的 75%，这表明内核似乎花了大量的时间在其他的工作上而不是在应用程序上。对这种情况，见 9.4.5 节。否则，应用程序很有可能就是 CPU 消耗量的原因，对每个应用程序，转到 9.5.1 节。

9.4.5 内核服务了许多中断吗？

这看起来好像是内核花费了大量时间完成那些不代表应用程序的工作。对这种情况

的一种解释是 I/O 卡提交了很多中断，比如，一个忙碌的网卡。运行 `procinfo` 或 `cat/proc/interrupts` 来确定有多少中断被提出，其提出频率是怎样的，以及哪些设备导致了这些中断。这可能为系统的行为提供线索。将这些信息记录下来，并进入 9.4.6 节。

9.4.6 内核的时间花在哪儿了？

最后要搞清楚的是内核究竟做了些什么。在系统上运行 `oprofile`，记录下哪些内核函数消耗了大量的时间（超过总时间的 10%）。尝试阅读这些函数的内核源代码，或是在 Web 上搜索这些函数的引用。可能不会立即弄清楚这些函数的功能，但是可以试着找出它们在哪个内核子系统中。仅仅确定使用的是哪个子系统（如内存、网络、调度或磁盘）可能就足以判断是哪里出了问题。

知道这些函数的功能还有可能了解到它们被调用的原因。如果函数是设备特定的，就要试着找出为什么要使用特定的设备（尤其是如果它还有大量的中断）。向其他发现同样问题的人发电子邮件，有可能的话，与内核开发人员联系。

转到 9.9 节。

9.4.7 交换空间的使用量在增加吗？

下一步是检查交换空间的使用量是否在增加。不少系统级性能工具，如 `top`、`vmstat`、`procinfo` 和 `gnome-system-info` 等都会提供这个信息。如果交换空间在增加，就需要找出是系统的哪个部分消耗了更多的内存。要实现这个目的，请转到 9.6.1 节。

如果被使用的交换空间没有增加，则见 9.4.8 节。

9.4.8 系统是受 I/O 限制的吗？

运行 `top` 时，查看系统是否在等待状态上消耗了大量的时间。如果这个时间比例超过了 50%，那么系统就在等待 I/O 上消耗了相当多的时间，我们就要确定这个 I/O 是哪种类型，见 9.4.9 节。

如果系统没有花大量的时间等待 I/O，那么你遇到的问题就不在本书范围之内，则转到 9.9 节。

9.4.9 系统使用磁盘 I/O 吗？

接下来，运行 `vmstat`（或 `iostat`）并查看磁盘读写的块数。如果磁盘读写的块数很大，那么这可能就是磁盘瓶颈，转到 9.7.1 节。否则，继续见 9.4.10 节。

9.4.10 系统使用网络 I/O 吗？

接下来，我们要查看系统是否使用了大量的网络 I/O。最简单的方法是运行 `iptraf`、

ifconfig 或 sar 来找出每个网络设备上传输了多少数据。如果网络流量接近网络设备的容量，那么就可能是网络瓶颈，则转到 9.8.1 节。如果看上去没有网络设备进行了网络通信，那么内核等待的是没有包含在本书范围内的其他一些 I/O 设备。查看内核调用了哪些函数以及哪些设备向内核发起了中断可能会有所帮助，转到 9.4.5 节。

9.5 优化进程CPU使用情况

当确定了某特定进程或应用程序是 CPU 瓶颈后，就必须查明其消耗时间的位置（和原因）。

图 9-3 展示了调查进程 CPU 使用情况的方法。

调查从 9.5.1 节开始。

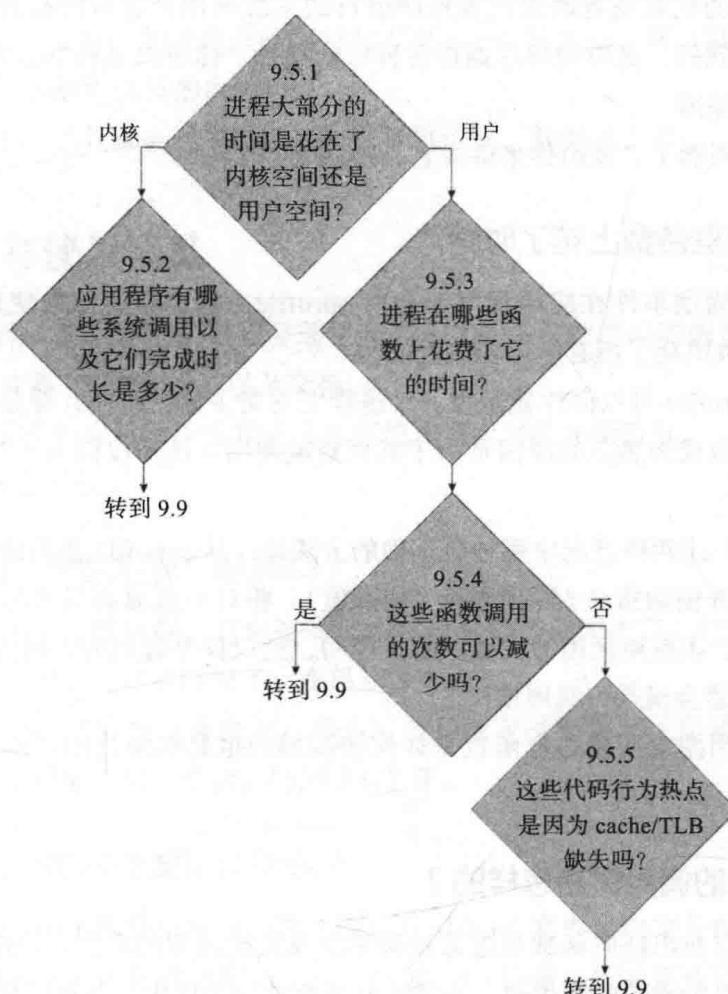


图 9-3

9.5.1 进程在用户还是内核空间花费了时间？

你可以用 time 命令来确定一个应用程序是否在内核或用户模式下消耗了时间。oprofile 也可以用来确定时间花在了哪里。通过分析每一个进程，能够看到一个进程是否将其时间花在了内核或用户空间。

如果应用程序在内核空间消耗了大量的时间（超过 25%），见 9.5.2 节。否则，转到 9.5.3 节。

9.5.2 进程有哪些系统调用，完成它们花了多少时间？

下一步，运行 strace 来查看有哪些系统调用以及它们完成的时长是多少。你还可以运行 oprofile 找出哪些内核函数被调用了。

减少系统调用的次数或者改变代表程序进行的系统调用都有可能提升性能。有些系统调用可能是意想不到的，是应用程序调用各种库的结果。你可以运行 ltrace 和 strace 来帮助确定它们被调用的原因。

现在问题已经明确了，就由你来解决它，转到 9.9 节。

9.5.3 进程在哪些函数上花了时间？

下一步，使用周期事件在应用程序上运行 oprofile，确定哪些函数使用了全部的 CPU 周期（即，哪些函数消耗了所有应用程序时间）。

记住，尽管 oprofile 可以向你显示在一个进程上花费了多少时间，但是在进行函数级分析时，一个特定函数成为热点的原因是由于其频繁被调用，还是仅仅由于其完成时间很长，是无法弄清楚的。

一种能弄明白上述两种情况中哪种是正确的方法是：从 oprofile 获得源代码级注释，并查找应该几乎没有开销的指令 / 源代码行（如赋值）。相对于其他高成本的源代码行，它们的样本数量将接近于函数被调用的次数。再次声明，这仅仅是近似的，因为 oprofile 只采样了 CPU，乱序处理器会误判一些周期。

做出函数的调用图对明确热点函数是如何被调用的也是有帮助的。实现这个目的，见 9.5.4 节。

9.5.4 热点函数的调用树是怎样的？

接下来，你可以找出耗时函数是怎么被调用的及其被调用的原因。把应用程序与 gprof 一起运行能够显示每个函数的调用树。如果耗时函数在一个库中，你可以使用 ltrace 来查看是哪些函数。最后，你可以使用 oprofile 较新的版本来支持调用树的跟踪。还有一种方法，

你可以在 `gdb` 中运行应用程序，在热点函数上设置断点。然后运行该应用程序，在每次调用热点函数时，它都会暂停。此时，可以生成一个回溯，看看究竟是哪些函数和源代码行产生了这个调用。

知道是哪些函数调用了热点函数能够让你消除或减少对这些函数的调用，相应地加快应用程序的速度。

如果减少对耗时函数的调用不能加快应用程序，或者无法消除这些函数，见 9.5.5 节。

否则，转到 9.9 节。

9.5.5 Cache 缺失与热点函数或源代码行是对应的吗？

下一步，针对你的应用程序运行 `oprofile`、`cachegrind` 和 `kcache`，看看耗时函数或源代码行是否具有大量的 cache 缺失。如果是，则尝试重新安排或压缩你的数据结构和访问，让它们变得更加 cache 友好。如果热点代码行没有高 cache 缺失率，那么就尝试重新安排你的算法来减少特定行或函数执行的次数。

在任何情况下，工具都会尽其所能地向你提供信息，转到 9.9 节。

9.6 优化内存使用情况

一般，要使用大量内存的应用程序通常会导致其他一些性能问题的产生，比如 cache 缺失、转换后援缓冲器（TLB）缺失以及交换。

图 9-4 展示了我们在试图弄清楚系统内存使用情况时的决策流程。

调查从 9.6.1 节开始。

9.6.1 内核的内存使用量在增加吗？

要追踪谁使用了系统内存，首先要确定内核自身是否分配内存。运行 `slabtop` 查看内核的内存总大小是否增加。如果增加了，则跳到 9.6.2 节。

如果内核的内存使用量没有增加，那么可能是特定进程导致了用量增加。要追踪是哪个进程该为内存使用量的增加负责，转到 9.6.3 节。

9.6.2 内核使用的内存类型是什么？

如果内核的内存使用量在增加，就再次运行 `slabtop` 来确定内核分配的内存类型。分片的名字多少会暗示一下内存被分配的原因。通过 Web 搜索，你可以找到内核源代码中每个分片名字的更多详细信息。只需在内核源代码中搜索该分片的名字，并确定它被用于哪些文件，就有可能弄清楚它被分配的原因。在明确了哪些子系统分配了所有的内存后，可以

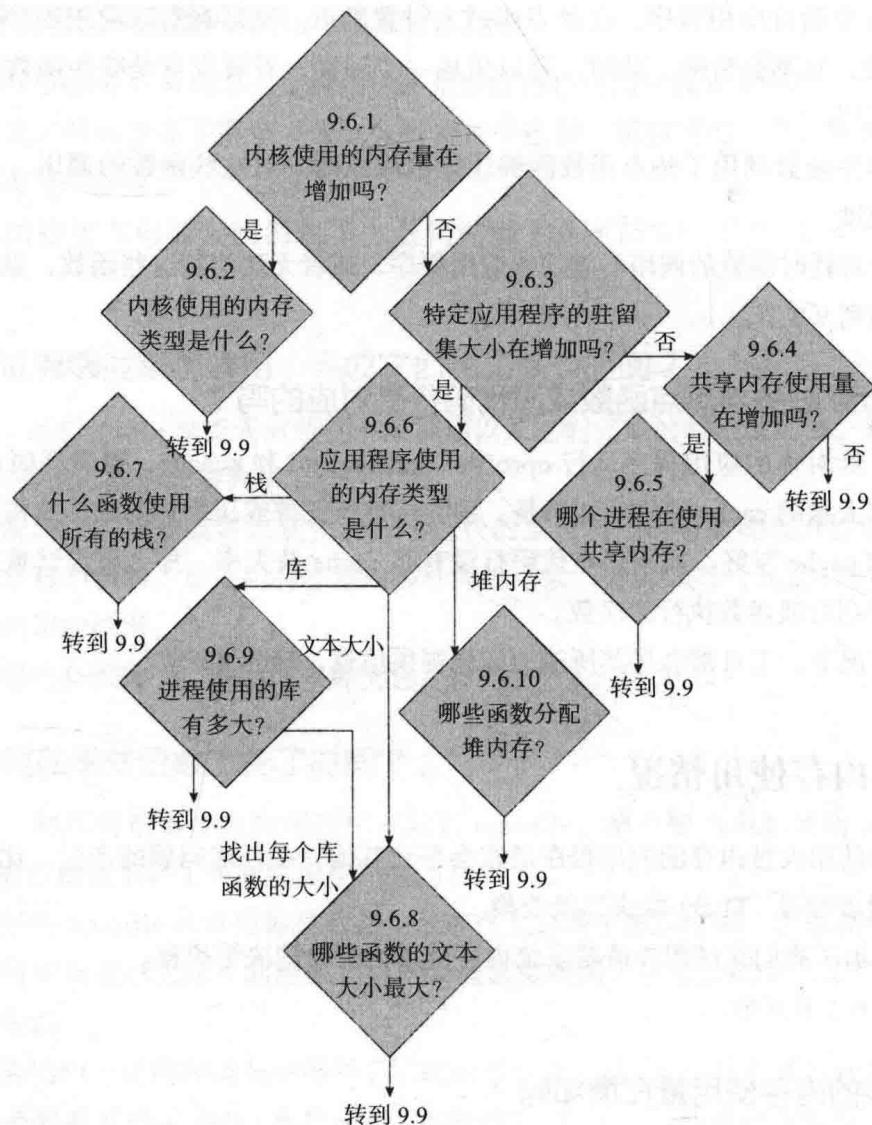


图 9-4

尝试调整特定子系统可以消耗的最大内存量，或者减少该子系统的使用量。

转到 9.9 节。

9.6.3 特定进程的驻留集大小在增加吗？

接下来，你可以使用 top 或 ps 来查看特定进程的驻留集大小是否在增加。最简单的方法是在 top 的输出中添加 rss 字段，并按照内存使用量来排序。如果一个特定进程不断增加内存的使用量，我们就需要弄清楚它用的内存类型是什么。要弄清楚应用程序使用的内存是什么类型，转到 9.6.6 节。如果没有特定进程使用了更多内存，则见 9.6.4 节。

9.6.4 共享内存的使用量增加了吗？

使用 ipcs 来确定被使用的共享内存数量是否在增加。如果是，见 9.6.5 节以确定哪些进程在使用内存。否则，你遇到的是不在本书讨论范围内的系统内存泄露问题，转到 9.9 节。

9.6.5 哪些进程使用了共享内存？

用 ipcs 来确定哪些进程使用并分配了共享内存。确定了使用了共享内存的进程之后，就调查各个进程来找出它们为什么使用内存。比如，在应用程序的源代码中寻找对 shmget（分配共享内存）或 shmat（附加到它上面）的调用。阅读应用程序的文档，查找解释并减少其共享内存使用的选项。

尝试减少共享内存使用量并转到 9.9 节。

9.6.6 进程使用的内存类型是什么？

找出进程使用的内存类型最简单的方法是在 /proc 文件系统中查看其状态。文件 cat /proc/<pid>/status 给出了进程内存使用情况的详细信息。

如果进程具有大的 VmExe 值，这就意味着可执行文件很大。要指明可执行文件中哪些函数导致了这个大小，请转到 9.6.8 节。如果进程具有大的 VmLib 值，这就意味着该进程使用了大量的共享库，或是几个体积较大的共享库。要指明哪些库导致了这个大小，请转到 9.6.9 节。如果进程的 VmData 值较大并且在增加，这就意味着该进程的数据区或堆在增加。要分析其原因，请转到 9.6.10 节。

9.6.7 哪些函数正在使用全部的栈？

要找出哪些函数分配了大量的栈，我们必须使用 gdb 和一点点技巧。第一步，使用 gdb 附加到正在运行的进程。第二步，用 bt 要求 gdb 产生回溯。第三步，(在 i386 上) 用 info registers esp 输出栈指针。这个输出就是栈指针的当前值。现在键入 up 并输出栈指针。前面栈指针和当前栈指针的差值(十六进制)就是前一个函数使用的栈容量。继续这样 up 回溯，你将可以发现哪个函数使用了大部分的栈。

当你确定了哪个函数或函数组消耗了大部分的栈之后，你可以修改应用程序，减少该函数(或这些函数)的调用次数和大小。转到 9.9 节。

9.6.8 哪些函数的文本大小最大？

如果可执行文件使用了相当可观的内存容量，那么确定哪些函数占用了最多的空间，并删除不必要的函数可能会有所帮助。对一个可执行文件或符号编译的库来说，可以请求

nm 显示所有符号的大小，并用如下命令对它们进行排序：

```
nm -S -size-sort
```

了解每个函数的大小后，就可能减少它们的大小或者从应用程序中移除不必要的代码。

转到 9.9 节。

9.6.9 进程使用的库有多大？

要了解进程使用了哪些库以及这些库各自的大小，最简单的方法是查看 /proc 文件系统中的进程映射。文件 cat /proc/<pid>/map 显示的是每个库及其代码与数据的大小。当你知道进程使用了哪些库之后，就有可能淘汰对大型库的使用，或者是用小一点的库来代替它们。但是，这样做的时候必须要小心，因为移除大型库未必会减少整个系统的内存使用量。

如果某库正在被其他任何应用程序使用（可以运行 lsof 来确定该库），库就已经被加载到了内存。任何新应用程序在使用这个库的时候都不需要再加载一个该库的副本到内存。让程序转而使用不同的库（即使是个小库）实际上就会增加总的内存使用量。这个新的库没有被其他进程使用，因此需要为其分配新的内存。最好的解决方法是缩小库自身的大小，或是修改它们以便使用更少的内存来保存库的特定数据。如果可行，则所有的应用程序都将受益。

要了解特定库中函数的大小，转到 9.6.8 节。否则，转到 9.9 节。

9.6.10 哪些函数分配堆内存？

如果你的应用程序是用 C 或 C++ 编写的，就可以使用内存剖析器 memprof 来找出哪些函数分配了堆内存。memprof 能够动态展示应用程序使用的内存量是如何增长的。

如果你的应用程序是用 Java 编写的，就在 java 命令行上添加 -Xrunhprof 命令行参数，它将会给出应用程序分配内存的详细信息。如果你的应用程序是用 C# (Mono) 编写的，就在 mono 命令行上添加 -profile 命令行参数，它也会给出应用程序分配内存的详细信息。

当你知道了哪些函数分配了最多的内存之后，就有可能减少被分配的内存大小。由于内存便宜，且越界错误很难被侦测到，因此，为了安全考虑，程序员常常超量分配内存。然而，如果一个特定的分配导致了内存问题，那么仔细分析最小分配就可能在保证安全的前提下，显著减少内存使用量。转到 9.9 节。

9.7 优化磁盘I/O使用情况

当你确定是磁盘 I/O 有问题后，明确是哪个应用程序引起了 I/O 就会有所帮助。

图 9-5 给出了确定磁盘 I/O 使用原因的步骤。

调查从 9.7.1 节开始。

9.7.1 系统强调特定磁盘吗？

在扩展统计模式下运行 iostat，寻找平均等待（await）大于零的分区。await 是等待请求被响应所平均花费的毫秒数。这个数值越高，则磁盘超负荷越多。可以通过查看磁盘的读写流量并确定其是否接近该驱动器可以处理的最大量来确认超负荷。

如果单个驱动器上的很多文件都被访问了，那么，将这些文件分散到多个磁盘就可能提高性能。不过，首先要确定的是哪些文件被访问了。

转到 9.7.2 节。

9.7.2 哪个应用程序访问了磁盘？

在前面关于磁盘 I/O 的章节中已经介绍过，确定哪个进程导致了大量的 I/O 是有难度的，因此，我们必须在缺少直接实现该功能工具的情况下试着解决这个问题。通过运行 top，首先寻找非空闲进程。对于每个这样的进程，转到 9.7.3 节。

9.7.3 应用程序访问了哪些文件？

首先，通过 strace，用 strace -e trace=file 来追踪应用程序中所有与文件 I/O 相关的系统调用。然后 strace 用摘要信息来查看每个调用花费的时长。如果某些读写调用完成时间很长，那么这个进程可能造成了 I/O 的缓慢。在正常模式下运行 strace 就可以发现是从哪个文件描述符进行读写的。要把这些文件描述符映射回文件系统中的文件，我们可以查看 proc 文件系统。/proc/<pid>/fd/ 中的文件是从文件描述符到实际文件的符号链接。该目录下的 ls -la 会显示进程使用了哪些文件。通过了解进程访问的文件，就有可能减少该进程执行的 I/O 量，将其更均匀地分散于多个磁盘，或者将其迁移到更快的磁盘。

确定进程访问哪些文件后，转到 9.9 节。



图 9-5

9.8 优化网络I/O使用情况

当知道网络发生了问题时，Linux 提供了一组工具来确定哪些应用程序涉及其中。但是，在与外部机器连接时，对网络问题的修复就不完全由你控制了。

图 9-6 展示了调查网络性能问题的步骤。

调查从 9.8.1 节开始。

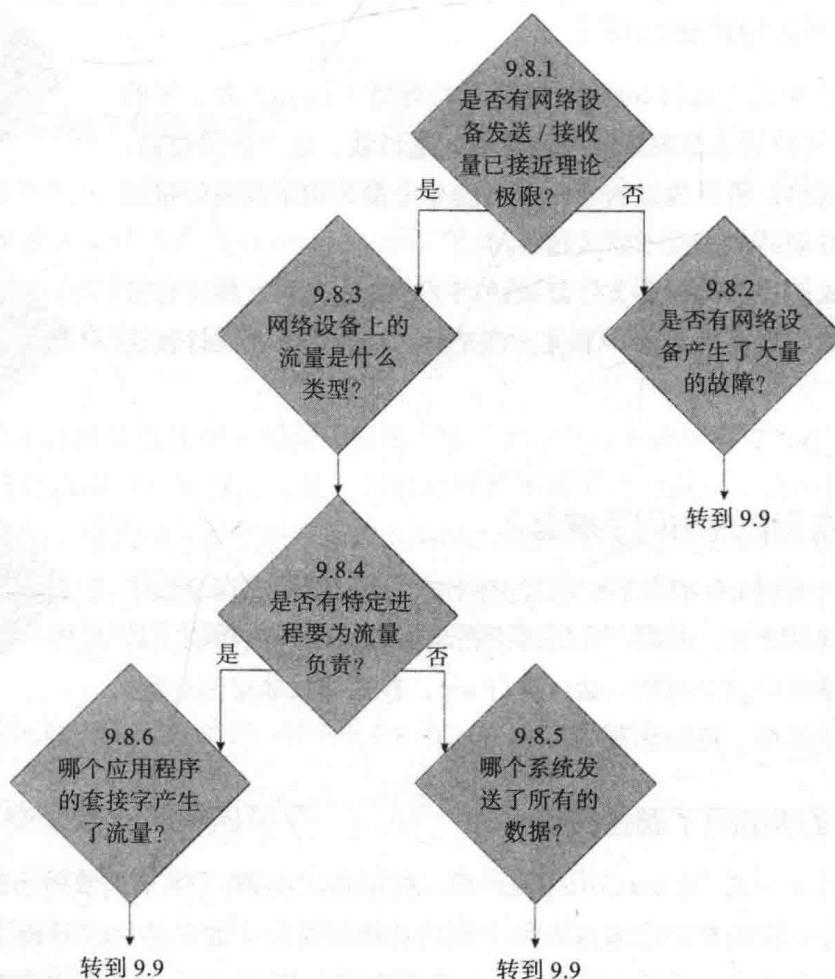


图 9-6

9.8.1 网络设备发送 / 接收量接近理论极限了吗？

要做的第一件事就是用 ethtool 来确定每个 Ethernet 设备设置的硬件速度是多少。如果有这些信息的记录，就可以调查是否有网络设备处于饱和状态。Ethernet 设备和 / 或交换机容易被误配置，ethtool 显示每个设备认为其应运行的速度。在确定了每个 Ethernet 设备的理论极限后，使用 iptraf（甚至是 ifconfig）来明确流经每个接口的流量。如果有任何网络设备表现出饱和，转到 9.8.3 节。否则，转到 9.8.2 节。

9.8.2 网络设备产生了大量错误吗？

网络流量减缓的原因也可能是大量的网络错误。用 ifconfig 来确定是否有接口产生了大量的错误。大量错误可能是不匹配的 Ethernet 卡 /Ethernet 交换机设置的结果。联系你的网络管理员，在 Web 上搜索遇到类似问题的人，或者把问题 e-mail 给一个 Linux 网络新闻组。

转到 9.9 节。

9.8.3 设备上流量的类型是什么？

如果特定设备正在服务大量的数据，使用 iptraf 可以跟踪该设备发送和接收的流量类型。当知道了设备处理的流量类型后，转到 9.8.4 节。

9.8.4 特定进程要为流量负责吗？

接下来，我们想要确定是否有特定进程要为这个流量负责。使用 netstat 的 -p 选项来查看是否有进程在处理流经网络端口的类型流量。

如果有应用程序要对此负责，转到 9.8.6 节。如果没有这样的程序，则转到 9.8.5 节。

9.8.5 流量是哪个远程系统发送的？

如果没有应用程序应对这个流量负责，那么就可能是网络上的某些系统用无用的流量攻击了你的系统。要确定是哪些系统发送了这些流量，要使用 iptraf 或 etherape。

如果可能的话，请与系统所有者联系，并尝试找出发生这种情况的原因。如果所有者无法联系上，可以在 Linux 内核中设置 ipfilters，永久丢弃这个特定的流量，或者是在远程机与本地机之间建立防火墙来拦截该流量。

转到 9.9 节。

9.8.6 哪个应用程序套接字要为流量负责？

确定使用了哪个套接字要分两步。第一步，用 strace -e trace=file 跟踪应用程序所有的 I/O 系统调用。这能显示进程是从哪些文件描述符进行读写的。第二步，通过查看 proc 文件系统，将这些文件描述符映射回套接字。/proc/<pid>/fd/ 中的文件是从文件描述符到实际文件或套接字的符号链接。该目录下的 ls -la 会显示特定进程全部的文件描述符。名字中带有 socket 的是网络套接字。之后就可以利用这些信息来确定程序中的哪个套接字产生了这些通信。

转到 9.9 节。

9.9 尾声

当你看到这里的时候，你可能得到也可能没有得到解决，但是，你会获取大量描述它的信息。在 Web 和新闻组上搜索遇到相同问题的人，向他们和开发者发电子邮件，看看他们是如何解决问题的。尝试一个解决方案，并观察系统或应用程序的行为是否发生了变化。每次尝试新方案时，请转到 9.2 节重新开始系统诊断，因为，每一个修复都可能会让应用程序的行为发生变化。

9.10 本章小结

本章提供了综合运用 Linux 性能工具跟踪不同类型性能问题的方法。虽然这个方法不可能捕捉到每一种可能出错的性能问题，但是它有助于发现一些比较常见的问题。此外，即便你面对的问题在这里没有涉及，你所收集的数据仍然是有用的，因为，这些数据可能会开启调查的不同方面。

接下来的几章将演示如何在 Linux 系统中使用该方法找出性能问题。

性能追踪 1： 受 CPU 限制的应用程序 (GIMP)

本章包含了一个例子：如何用 Linux 性能工具在受 CPU 限制的应用程序中寻找并修复性能问题。

阅读本章后，你将能够：

- 在受 CPU 限制的应用程序中明确所有的 CPU 被哪些源代码行使用。
- 用 ltrace 和 oprofile 弄清楚应用程序调用各种内部与外部函数的频率。
- 在应用程序源代码内寻找模式，在线搜索应用程序的行为表现与可能的解决方案。
- 以本章为模板，跟踪与 CPU 相关的性能问题。

10.1 受CPU限制的应用程序

本章的调查对象是一个受 CPU 限制的应用程序。其重点在于能够对受 CPU 限制的应用程序进行优化，因为，这是最常见的性能问题之一。

通常，这也是高度调优的应用程序的最后战线。

当消除了磁盘和网络瓶颈后，一个应用程序就变成了受 CPU 限制的。此外，与改进 CPU 相比，购买更快的磁盘或更多的内存要容易得多，因此，如果程序是受 CPU 限制的，那么比起仅仅买一个新系统来说，能够追踪并修复 CPU 性能问题就是一项重要技能了。

10.2 确定问题

性能追踪的第一步是确定要调查的问题。本例中，我选择调查使用 GIMP 时出现的性能问题，GIMP 是一个开源的图像处理程序。它能对一个图像的各个方面进行分割，但它也有一组强大的过滤器，能够用多种方式对图像进变形和修改。这些过滤器根据一些复杂的算法来改变图像的外观。通常情况下，过滤器的工作需要很长时间才能完成，并且非常耗费 CPU 资源。特别是其中的一个过滤器 Von Gogh（梵高，LIC），它接收一个图像为输入，将其修改为看上去像梵高风格的画作。这个过滤器所花费的时间特别长。过滤器运行时使用了几乎 100% 的 CPU 资源，且完成时间长达几分钟。完成时长涉及的因素包括：图像大小、机器 CPU 的速度，以及传递给过滤器的参数值。本章中，我们用 Linux 性能工具调查为什么这个过滤器这么慢，同时还要弄清楚是否有方法能提高它的速度。

10.3 找到基线/设置目标

任何性能追踪的第一步就是要确定问题当前的状况。对 GIMP 过滤器而言，我们需要确定它在特定图像上运行要花多少时间，这就是基准时间。一旦掌握了这个基准时间，接下来就可以尝试优化，看看是否减少了它的执行时间。有时候，计量某个工作耗费的时间是很难的，不像拿个秒表那么简单，因为当我们相当耗费 CPU 资源的作业正在运行的时候，操作系统可能会调度其他任务。在这种情况下，如果除了计算密集型作业之外还有其他作业也在运行，那么墙钟时间也许会大大超过该进程实际使用的 CPU 时间。就这点来说，我们是幸运的，当运行过滤器时，通过观察 top，可以看到 lic 进程占用了绝大部分的 CPU 使用量，如清单 10.1 所示。

清单 10.1

```
[ezolt@localhost ktracer]$ top

top - 08:24:48 up 7 days, 9:08, 6 users, load average: 1.04, 0.64, 0.76

      PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 32744 ezolt      25   0  53696   45m   11m R 89.6 14.6   0:16.00 lic
 2067 root      15   0  69252   21m   17m S  6.0   6.8 161:56.22 X
 32738 ezolt      15   0  35292   27m   14m S  2.3   8.7   0:05.08 gimp
```

由此，我们可以推断在运行过滤器时，GIMP 实际上派生出一个独立进程来运行。因此，当过滤器运行时，我们就可以利用 ps 来追踪该进程消耗了多少 CPU 时间，以及它是什么时候结束的。知道了使用 top 的过滤器的 PID 之后，可以运行清单 10.2 中的循环，要求 ps

周期性地观察该过滤器使用了多少 CPU 时间。

清单 10.2

```
while true ; do sleep 1 ; ps 32744; done
PID TTY      STAT   TIME COMMAND
32744 pts/0    R      2:46 /usr/local/lib/gimp/2.0/plug-ins/lic -gimp 8 6 -run 0
```

 **注意** 如果在没有秒表的情况下对一个应用程序计时，你可以把 time 和 cat 当作简易的秒表来用。只需在想要开始计时的时候键入 time cat，然后在结束时按下 <Ctrl-D> 组合键即可。time 将会显示已经过的时间。

在参照图像（从我的地下室获取的图片）上运行 lic 过滤器并用刚才描述的 ps 方法对该过滤器计时，我们可以从清单 10.2 中看出来，处理整个图像花费了 2 分 46 秒。这个时间就是我们的基准时间。既然知道了该过滤器直接运行耗费的时间，我们就可以设置本次性能追踪的目标了。如何为性能调查设置合理的目标并非总是清晰明了的。一个合理的目标值取决于多个因素，包括根据特定问题和用户需求已经完成的调优。通常，最好的方法是以有相同情况且性能更快的应用程序为基础来设置目标。可惜的是，我们不知道还有哪个 GIMP 过滤器是完成类似工作的，因此，我们必须做出猜测。对一段相对未调优的代码来说，一般 5% ~ 10% 的性能优化是一个合理的目标，所以，我们设定了 10% 的速度提升，即运行时间为 2 分 30 秒。

既然目标已经挑好了，我们就需要一种方法来保证我们的调优带给过滤器结果的变化是可以接受的。此时，我们将在参照图像上运行初始过滤器，并把其结果记录在一个文件中。这样，我们可以将优化后的过滤器输出与初始过滤器输出进行比较，看看优化是否改变了输出。

10.4 为性能追踪配置应用程序

我们调查的下一步是为性能追踪设置应用程序，用符号重新编译该程序。符号能让性能工具（比如 oprofile）调查哪些函数和源代码行消耗了 CPU 时间。

回到 GIMP，我们从其网站下载最新的 GIMP 源代码压缩包，然后重新编译它。对 GIMP 和许多开源软件来说，重新编译的第一步就是运行 configure 命令，产生将用于构建应用程序的生成文件。configure 命令向生成文件传递出现在 CFLAGS 环境变量中的所有标志。在这种情况下，由于我们希望用符号构建 GIMP，因此设置的 CFLAGS 变量含有 -g3。这就使得符号包含在生成的二进制文件中。清单 10.3 显示了这个命令，覆盖了 CFLAGS 环境变量的当前值，并将其设置为 -g3。

清单 10.3

```
[root@localhost gimp-2.0.3]# env CFLAGS=-g3 ./configure
```

之后，我们生成并安装包含所有符号的 GIMP 版本，当运行这个版本时，性能工具将会告诉我们时间都消耗在了哪里。

10.5 安装和配置性能工具

如果性能工具还没有安装，那么追踪的下一步就是安装它们。尽管看上去这是件容易的事儿，但它常常涉及跟踪定制包的分发，甚至是从头开始对工具进行重新编译。本例中，我们在 Fedora Core 2 上使用 oprofile，因此，我们既要跟踪 oprofile 内核模块（在 Fedora 下，它只包含在对称多处理 (SMP) 内核中），也要跟踪 oprofile 软件包。同样可能让人觉得有意思的还有用 ltrace 性能工具查看被调用的库函数及其被调用的频率。幸运的是，ltrace 包含在 Fedora Core 2 中，所以我们不需要跟踪它。

10.6 运行应用程序和性能工具

接下来，我们运行应用程序，并用性能工具进行测量。由于 lic 过滤器直接由 GIMP 调用，因此，我们使用的工具必须能够附加到已运行进程，且能对其进行监控。

对 oprofile 来说，我们启动 oprofile，运行过滤器，然后在过滤器完成工作后停止 oprofile。因为 lic 过滤器在运行时占用了将近 90% 的 CPU，所以 oprofile 收集的系统整体样本将主要与 lic 过滤器相关。当 lic 开始运行时，我们在另一个窗口启动 oprofile。当 lic 结束时，我们停止 oprofile。oprofile 的启动和停止如清单 10.4 所示。

清单 10.4

```
[root@localhost ezolt]# opcontrol --start
Profiler running.
[root@localhost ezolt]# opcontrol --dump
[root@localhost ezolt]# opcontrol --stop
Stopping profiling.
```

运行 ltrace 必定存在一点差异。在过滤器启动后，ltrace 可以附加到运行中的进程上。与 oprofile 不同，ltrace 附加到进程后会降低整个进程的速度。这会导致每个库的调用时间增加一些误差，但是它却提供了每个调用的次数信息。ltrace 如清单 10.5 所示。

清单 10.5

```
[ezolt@localhost ktracer]$ ltrace -p 32744 -c
% time      seconds   usecs/call    calls      function
-----
43.61  156.419150        254  614050  rint
16.04   57.522749        281  204684  gimp_rgb_to_hsl
14.92   53.513609        261  204684  g_rand_double_range
13.88   49.793988        243  204684  gimp_rgba_set_uchar
11.55   41.426779        202  204684  gimp_pixel_rgn_get_pixel
0.00    0.006287        6287      1  gtk_widget_destroy
0.00    0.003702        3702      1  g_rand_new
0.00    0.003633        3633      1  gimp_progress_init
0.00    0.001915        1915      1  gimp_drawable_get
0.00    0.001271        1271      1  gimp_drawable_mask_bounds
0.00    0.000208        208       1  g_malloc
0.00    0.000110        110       1  gettext
0.00    0.000096        96        1  gimp_pixel_rgn_init
-----
100.00  358.693497      1432794 total
```

要获得库调用的全部数量，可以让 ltrace 持续运行直到完成。但是，这需要很多的时间，所以在这种情况下，经过一段较长时间后就按下 <Ctrl-C> 组合键。这个方法不见得总是有效，原因在于应用程序可能会经历不同的执行阶段，如果停止时间过早，可能就无法得到应用程序调用函数的完整信息。不过，这个短小的样本至少是我们分析的起点。

10.7 分析结果

既然我们已经利用 oprofile 收集到了过滤器运行的时间信息，现在就必须对结果进行分析，寻找改变其执行和提高其性能的方法。

首先，我们用 oprofile 查看整个系统是如何消耗时间的，结果如清单 10.6 所示。

清单 10.6

```
[root@localhost ezolt]# opreport -f | less
```

```
CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt
TIMER:0|
samples| %|
```

```

69896 36.9285 /usr/local/lib/libgimp-2.0.so.0.0.3
44237 23.3719 /usr/local/lib/libgimpcolor-2.0.so.0.0.3
28386 14.9973 /usr/local/lib/gimp/2.0/plug-ins/lic
16133 8.5236 /usr/lib/libglib-2.0.so.0.400.0
...

```

如清单 10.6 所示，75% 的 CPU 时间消耗在了 lic 进程或与 GIMP 相关的库上。极有可能这些库是被 lic 进程调用的，结合 ltrace 给我们的信息与 oprofile 给出的信息，我们可以确认这个事实。清单 10.7 显示了过滤器运行一小部分的情况下进行的库调用。

清单 10.7

[ezolt@localhost ktracer]\$ ltrace -p 32744 -c				
% time	seconds	usecs/call	calls	function
46.13	101.947798	272	374307	rint
15.72	34.745099	278	124862	g_rand_double_range
14.77	32.645236	261	124862	gimp_pixel_rgn_get_pixel
13.01	28.743856	230	124862	gimp_rgba_set_uchar
10.36	22.905472	183	124862	gimp_rgb_to_hsl
0.00	0.006832	6832	1	gtk_widget_destroy
0.00	0.003976	3976	1	gimp_progress_init
0.00	0.003631	3631	1	g_rand_new
0.00	0.001992	1992	1	gimp_drawable_get
0.00	0.001802	1802	1	gimp_drawable_mask_bounds
0.00	0.000184	184	1	g_malloc
0.00	0.000118	118	1	gettext
0.00	0.000100	100	1	gimp_pixel_rgn_init
<hr/>				
100.00	221.006096		873763	total

接下来，我们要调查 oprofile 提供的关于每个库内部 CPU 时间消耗的信息，看看库中的热点函数是否与过滤器调用的那些一致。对于最占用 CPU 时间的前三个库，我们要求oprofile 提供库内每一个函数时间消耗的详细信息。libgimp、libgimp-color 库和 lic 进程的结果显示在清单 10.8 中。

清单 10.8

```

[/tmp]# oreport -lf /usr/local/lib/libgimp-2.0.so.0.0.3
CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt

```

samples	%	symbol name
27136	38.8234	gimp pixel_rgn_get_pixel
14381	20.5749	gimp drawable_get_tile2
6571	9.4011	gimp tile_unref
6384	9.1336	gimp drawable_get_tile
3921	5.6098	gimp tile_cache_insert
3322	4.7528	gimp tile_ref
3057	4.3736	anonymous symbol from section .plt
2732	3.9087	gimp tile_width
1998	2.8585	gimp tile_height
...		

[/tmp]# oreport -lf /usr/local/lib/libgimpcolor-2.0.so.0.0.3

CPU: CPU with timer interrupt, speed 0 MHz (estimated)

Profiling through timer interrupt

samples	%	symbol name
31475	71.1508	gimp rgba_set_uchar
6251	14.1307	gimp bilinear_rgb
2941	6.6483	gimp rgb_multiply
2394	5.4118	gimp rgb_add
466	1.0534	gimp rgba_get_uchar
323	0.7302	gimp rgb_to_hsl
...		

[/tmp]# oreport -lf /usr/local/lib/gimp/2.0/plug-ins/lic

CPU: CPU with timer interrupt, speed 0 MHz (estimated)

Profiling through timer interrupt

samples	%	symbol name
11585	40.8124	getpixel
5185	18.2660	lic_image
4759	16.7653	peek
3287	11.5797	filter
1698	5.9818	peekmap
1066	3.7554	anonymous symbol from section .plt
316	1.1132	compute_lic
232	0.8173	rgb_to_hsl
111	0.3910	grady
106	0.3734	gradx
41	0.1444	poke
...		

通过比较清单 10.8 中的 ltrace 输出和清单 10.9 中的 oprofile 输出可以得知：lic 过滤器反复调用的库函数消耗了全部时间。

接下来，我们调查 lic 过滤器的源代码，确定它是如何构成的，它的热点函数究竟起了怎样的作用，以及过滤器是如何调用 GIMP 库函数的。生成大多数样本的 lic 函数是 getpixel，如清单 10.9 中的 opannotate 输出所示。opannotate 在源代码的左侧以列的形式依序显示了样本数量和所占样本总量百分比。这使你能查看源代码，精确定位哪些代码行是热点。

清单 10.9

```
opannotate --source /usr/local/lib/gimp/2.0/plug-ins/lic
.....
:static void
:getpixel (GimpPixelRgn *src_rgn,
:           GimpRGB      *p,
:           gdouble       u,
:           gdouble       v)
428 1.5961 :{ /* getpixel total: 11198 41.7587 */
:   register gint x1, y1, x2, y2;
:   gint width, height;
:   static GimpRGB pp[4];
:
98  0.3655 :   width = src_rgn->w;
72  0.2685 :   height = src_rgn->h;
:
1148 4.2810 :   x1 = (gint)u;
1298 4.8404 :   y1 = (gint)v;
:
603  2.2487 :   if (x1 < 0)
1  0.0037 :     x1 = width - (-x1 % width);
:
1605 5.9852 :   x1 = x1 % width;
:
87  0.3244 :   if (y1 < 0)
:
1358 5.0641 :     y1 = height - (-y1 % height);
:
1264 4.7136 :   y1 = y1 % height;
:
1379 5.1425 :   x2 = (x1 + 1) % width;
:
320  1.1933 :   peek (src_rgn, x1, y1, &pp[0]);
```

```
267 0.9957 : peek (src_rgn, x2, y1, &pp[1]);
285 1.0628 : peek (src_rgn, x1, y2, &pp[2]);
244 0.9099 : peek (src_rgn, x2, y2, &pp[3]);
:
706 2.6328 : *p = gimp_bilinear_rgb (u, v, pp);
35 0.1305 :}

...
```

关于 `get_pixel` 函数还有一些有趣的事儿需要注意。首先，它调用 `gimp_bilinear_rgb` 函数，该函数是 GIMP 库中的一个热点函数。其次，它调用了 `peek` 函数 4 次。如果 `get_pixel` 调用执行多次，那么 `peek` 函数执行的次数就是它的 4 倍。用 `opannotate` 查看 `peek` 函数（如清单 10.10 所示），可以看到它调用了 `gimp_pixel_rgn_get_pixel` 和 `gimp_rgba_set_uchar`，它们分别是 `libgimp` 和 `libgimp-color` 的高级函数。

清单 10.10

```
:static void
:peek (GimpPixelRgn *src_rgn,
:     gint          x,
:     gint          y,
:     GimpRGB      *color)
481 1.7937 :{ /* peek total: 4485 16.7251 */
:     static uchar data[4] = { 0, };
:
1373 5.1201 : gimp_pixel_rgn_get_pixel (src_rgn, data, x, y);
2458 9.1662 : gimp_rgba_set_uchar (color, data[0], data[1], data[2],
data[3]);
173 0.6451 :}
```

尽管不是很清楚究竟过滤器是干什么的或者库调用的作用是什么，还是有几个让人想去了解的要点。首先，`peek` 的功能听起来像是从图像获取像素以便过滤器对它们进行处理。我们马上就可以验证这个预感。其次，过滤器中的大部分时间似乎并不是用于在图像数据上运行数学算法。并没有将所有 CPU 时间都用于根据像素数值进行计算，该过滤器好像花费了大部分时间来检索待处理的像素。如果真是这样，那么这一点也许能修复。

10.8 转战网络

既然我们已经发现了大部分时间里用的是哪些 GIMP 函数，现在我们就必须弄清楚这些函数是什么，并尽可能优化它们的使用。

首先，我们在 Web 上搜索 `pixel_rgn_get_pixel`，并尝试明确它是做什么的。开始出错几次后，清单 10.11 给出的链接和信息证实了我们对 `pixel_rgn_get_pixel` 功能的猜测。

清单 10.11

```
"There are calls for pixel_rgn_get_pixel, row, col, and rect, which grab
data from the image and dump it into a buffer that you've pre-allocated.
And there are set calls to match. Look for "Pixel Regions" in gimp.h."
(from http://gimp-plug-ins.sourceforge.net/doc/Writing/html/sect-
image.html )
```

同时，清单 10.12 中的信息表明避免使用 `pixel_rgn_get_calls` 是个好主意。

清单 10.12

```
"Note that these calls are relatively slow, they can easily be the
slowest thing in your plug-in. Do not get (or set) pixels one at a time
using pixel_rgn_[get|set]_pixel if there is any other way." (from
http://www.home.unix-ag.org/simon/gimp/guadec2002/gimp-
plugin/html/imagedata.html)
```

此外，通过 Web 搜索找到函数的源代码，很方便地知道了 `gimp_rgb_set_uchar` 函数的信息。如清单 10.13 所示，该调用就是把呈现单一颜色的红色、绿色、蓝色值都放入 GimpRGB 结构中。

清单 10.13

```
void
gimp_rgb_set_uchar (GimpRGB *rgb,
                     uchar   r,
                     uchar   g,
                     uchar   b)
{
    g_return_if_fail (rgb != NULL);

    rgb->r = (gdouble) r / 255.0;
    rgb->g = (gdouble) g / 255.0;
    rgb->b = (gdouble) b / 255.0;
}
```

从 Web 收集的信息证实了我们的猜测：函数 `pixel_rgn_get_pixel` 是从图像抽取数据的一种方法，而函数 `gimp_rgb_set_uchar` 则仅仅获取从 `pixel_rgn_get_pixel` 返回的颜色数据，并将它们送入 GimpRGB 数据结构中。

我们不但看到了如何使用这些函数，而且其他页面也暗示，如果我们想要过滤器的性能达到巅峰，那么这些函数可能不是最好的选择。其中一个网页（<http://www.home.unix-ag.org/simon/gimp/guadec2002/gimp-plugin/html/efficientaccess.html>）建议通过使用 GIMP 图

像缓存有可能提高性能。另一个网站 (<http://gimp-plug-ins.sourceforge.net/doc/Writing/html/sect-tiles.html>) 则建议有可能通过重写过滤器使其更高效地访问图像数据来提高性能。

10.9 增加图像缓存

网站解释 GIMP 以一种稍显怪诞的方式管理图像。与使用大的数组保存图像不同，GIMP 将图像分解为一组分片，这些分片的宽度都是 64×64 。当过滤器想要访问图像中的一个特定像素时，GIMP 就加载相应的分片，然后找到并返回该像素的值。每个检索特定像素的调用都可能会很慢。如果对每一个像素都重复这个过程，那么 GIMP 重载用于检索像素值的分片就会显著降低性能。幸运的是，GIMP 提供了一种方法来缓存旧的分片值，每一次使用的是这个缓存值而不是重载分片。这将会提高性能。GIMP 提供的缓存量可以用 `gimp_tile_cache_ntiles` 调用来自控制。该调用当前在 `lic` 内部使用，并将缓存设置为图像分片数量的两倍。

即便这个缓存看上去够用了，GIMP 可能还会需要更大的缓存。测试方式很简单，将缓存量增加到一个非常大的值，然后看看性能是否有所提升。因此，本例中，我们把缓存量增加到通常使用量的 10 倍。缓存值加大后，重新运行过滤器，检索时间为 2 分 40 秒。虽然所用时间减少了 6 秒，但是还是没有达到目标时间 2 分 30 秒。这表示我们必须从其他方面来提升性能。

10.10 遇到（分片引发的）制约

除了使用分片缓存之外，网页提出了一种更好的方法来提高 `get_pixel` 的性能。通过直接访问像素信息（不调用 `gimp_pixel_rgn_get_pixel`），可以显著提高像素访问的性能。

GIMP 为过滤器编程者提供了一种直接访问图像分片的方法。之后过滤器就可以像访问一个数组一样访问图像数据，而不用请求对 GIMP 库的调用。但是，这里有个问题。当直接访问像素信息时，它只针对当前分片。因此，GIMP 将遍历图像中的全部分片，以便你最终得到图像中的所有像素，可是你却无法同时访问它们。只能查看单个分片的像素，而这与 `lic` 访问数据的方式不相符。当 `lic` 过滤器在特定位置产生一个新的像素时，它是基于其周围像素的值来计算这个新值。所以，如果是在一个分片的边缘产生了新像素，`lic` 过滤器就需要其周围所有像素的数据。但是，这些像素可能是在图像的上一个分片上，也可能是在下一个分片上。由于该像素信息不可用，图像过滤器将不会使用这个优化的访问方法。

10.11 解决问题

由于我们已经判定读取像素值很占时间，这里还有另一种方法可以解决这个问题。我们必须开始审视过滤器是如何运行的。在它生成新图像时，它会重复请求同一个像素。这是因为新像素的值是根据其周围像素的值得来的，因此，在对图像运行过滤器的过程中，每个像素会被其8个邻居像素访问。这就意味着图像中的每个像素都将被其每一个邻居像素读取，其结果就是它将被至少读取9次。

调用GIMP库的开销是很大的，所以我们只想对每个像素进行一次这样的操作，而不是9次。对图像访问可能的优化是：过滤器启动时将整个图像读入一个本地数组，在过滤器运行期间就访问这个本地数组，而不是在每次想要访问数据时调用GIMP库函数。这个方法可以显著降低查找像素数据的开销。每次访问数据时不再进行几个函数调用，取而代之的是只访问本地数组。过滤器初始化时，用malloc分配该数组，并用像素数据进行填充。具体操作如清单10.14所示。

清单 10.14

```

int g_image_width, g_image_height;
GimpRGB *g_cached_image;

void cache_image(GimpPixelRgn *src_rgn,int width,int height)
{
    static uchar data[4];
    int x,y;
    GimpRGB *current_pixel;

    g_image_width = width;
    g_image_height = height;

    g_cached_image = malloc(sizeof(GimpRGB)*width*height);
    current_pixel = g_cached_image;

    /* Malloc */
    for (y = 0; y < height; y++)
    {
        for (x = 0; x < width; x++)
        {
            gimp_pixel_rgn_get_pixel (src_rgn, data, x, y);
            gimp_rgba_set_uchar (current_pixel, data[0], data[1], data[2],
ata[3]);
            current_pixel++;
        }
    }
}

```

同时，函数 `peek` 也重新编写为只访问这个本地数组，而不再调用 GIMP 库函数。具体操作如清单 10.15 所示。

清单 10.15

```
static void peek (GimpPixelRgn *src_rgn,
    gint      x,
    gint      y,
    GimpRGB  *color)
{
    *color = g_cached_image[y*g_image_width + x];
}
```

那么，有效果吗？当我们用新方法运行过滤器时，运行时间减少了 56 秒！刚好在目标时间范围 2 分 30 秒内，性能有明显提升。

虽然令人印象深刻，但是这个性能结果不是平白得来的。我们做的是性能工程中的一个经典取舍：用内存的增加换取了性能的提高。举个例子，如果过滤器使用的是一个 1280×1024 的图像，所需内存量将增加 5MB。对于非常大的图像，缓存该数据可能是不实际的。但是，对于大小合理的图像，相较于过滤器两倍多速度的提高，5MB 内存使用量的增加似乎是个不错的牺牲。

10.12 验证正确性

我们已经用一个优化明显降低了过滤器的运行时间，现在重要的是验证过滤器优化后和优化前产生的图像输出是相同的。加载了原来的参考图像后，将它与新生成的图像比较，这里使用了 GIMP 来获取两个图像之间的差异。如果参考图像和优化后图像是相同的，那么所有的像素都应该为零（黑色）。但是，差异图像并非是完全黑色的。从视觉上看，它像是黑色的，可仔细检查后（使用 GIMP 颜色选择器）发现有些像素是非零的。这就意味着参考图像与优化后图像存在差异。

通常这会引起关注，因为这可能表明该优化改变了过滤器的行为。但是，仔细检查过滤器源代码后显示，有几个地方的随机噪声在过滤器运行前就使得图像产生了轻微抖动。过滤器的任意两次运行都会有所不同，因此，很可能不应为此责备优化。由于两个图像之间的差异在视觉上如此微小，因此我们可以假设优化没有引入任何问题。

10.13 后续步骤

我们超额完成了 lic 过滤器性能增加 10% 的目标，因此就这点来说，我们已经完成了优

化过程。但是，如果想要继续优化性能，就必须对使用新优化的过滤器进行重新分析。每一个性能优化应用后，重新分析应用程序，不依赖于之前的分析对应用程序的继续优化来说是非常重要的。每次优化后，应用程序的运行时行为可能发生显著变化。如果不在每次优化后进行分析，你就有可能承担继续追逐已不存在的性能问题的风险。

10.14 本章小结

本章中，我们确定了为什么一个应用程序（GIMP 过滤器 lic）是受 CPU 限制的。我们计算出了该应用程序的基准运行时间，设置了一个优化目标，保存了一个参考图像以验证我们的优化没有改变应用程序的行为。我们使用了 Linux CPU 性能工具（oprofile 和 ltrace）调查究竟为什么该应用程序是受 CPU 限制的。然后依靠 Web 理解了该应用程序是如何工作的，并了解了对其不同的优化方法。我们尝试了几种不同的优化，但是最后还是选择了经典的性能权衡：增加内存使用量来降低 CPU 使用量。

我们达到了优化目标，之后还验证了优化并未改变应用程序的输出。

本章侧重于优化单个应用程序的运行时间，下一章的性能追踪则集中于减少与 X Window 交互时的延迟。降低延迟可能会非常棘手，因为单一事件通常会引发一组非显著性的其他事件。最困难的部分是找出被调用的事件有哪些，它们耗时多长。

性能追踪 2： 延迟敏感的应用程序 (nautilus)

本章包含了一个例子：如何用 Linux 性能工具在延迟敏感的应用程序中寻找并修复性能问题。

阅读本章后，你将能够：

- 在延迟敏感的应用程序中用 ltrace 和 oprofile 弄清楚哪里产生了延迟。
- 对“热点”函数的每个调用，用 gdb 生成栈跟踪。
- 用性能工具确定使用了多个不同共享库的应用程序是如何消耗时间的。
- 以本章为模板，找出延迟敏感应用程序中高延迟的原因。

11.1 延迟敏感的应用程序

本章我们将调查一个应用程序，该程序对长延迟敏感。延迟可以被认为是一个应用程序响应不同的外部或内部事件所花费的时间。具有延迟性能问题的应用程序一般不是长时间占用 CPU，相反，它只用少量的 CPU 时间来响应不同的事件。但是，这种响应对特定事件来说是不够的。在修复延迟性能问题时，我们需要降低对各种时间的响应延迟，并找出是应用程序的哪些部分延缓了响应。正如你将看到的，与追踪受 CPU 限制的问题相比，追踪延迟问题需要的策略略有不同。

11.2 确定问题

对前一章的性能问题，我们必须定义调查内容，并尝试去克服它。本章中，我们把这个过程优化一下，使用 GNOME 桌面的 nautilus 文件管理器打开一个弹出菜单。在 nautilus 文件管理器窗口的任何位置点击右键即可打开弹出菜单。在这种特定情况下，我们将调查鼠标右键点击一个打开窗口的背景时弹出菜单显示的性能，而不是鼠标右键点击一个特定文件或文件夹时弹出菜单的显示性能。

为什么要对这进行优化？即使打开一个弹出菜单的时间比一秒钟还要少，但它仍然慢到足以让用户感觉到自己点击鼠标右键与菜单显示之间的时间差。这种缓慢的弹出带给 GNOME 用户的印象是计算机运行的速度慢。人们注意到轻微的延迟，它会让与 nautilus 的交互变得烦人，或是给人留下桌面迟缓的印象。

这个特殊的性能问题与前一章中的 GIMP 问题不同。第一，桌面（本例中为 GNOME）的核心组件通常比一个典型的桌面应用程序更为复杂与交错。这些组件为了完成工作一般要依赖于各种各样的子系统和共享库。而 GIMP 是一个相对独立的应用程序，这使得它更容易分析，并在必要的时候重新编译。GNOME 桌面则不同，它是由多个不同的交错组件构成的。这些组件可能需要多个进程和共享库，其中每个库都代表桌面执行不同的任务。尤其是 nautilus，它链接了 72 个不同的共享库。追踪到底是哪一段代码消耗了时间，消耗了多少，为什么消耗，可以说是一个艰巨的任务。

本章性能调查与 GIMP 调查第二个明显不同的地方是：我们想要减少的时间是以毫秒计，而不是以秒或分钟计。当时间小到这个程度，就很难确保捕获到的性能分析数据确实是你要测量的事件结果，而不只是尝试停止和启动分析工具时周围的噪声。不过，这个短暂的时间周期还是能在你感兴趣的时间内实际跟踪应用程序工作的方方面面。

11.3 找到基线/设置目标

在前一个性能追踪案例中，第一步是确定问题的当前状态。为了让我们的工作更容易一点，并且回避一些上一节提到的分析问题，我们要要点小花招，让弹出菜单问题看起来更像之前我们测量过的长时间运行的 CPU 密集型任务。单个弹出菜单显示的时间是毫秒级的，这使得用我们的性能工具很难进行精确的测量。如前所述，很难在合适的时间启动和停止工具，并确保我们只测量到了感兴趣的（即，打开确切菜单所花的 CPU 时间）。我们将在这里玩点小技巧。我们将快速连续地打开菜单 100 次，而不仅仅只打开一次。这样，菜单打开的总时间将会达到 100 倍。这使得我们能用剖析工具捕获菜单正如何执行的信息。

由于右键点击 100 次很乏味，且人类（除非非常训练有素）不可能可靠地重复打开一个

弹出菜单 100 次，所以我们必须将这个过程自动化。要可靠地打开弹出菜单 100 次，我们依赖于 xautomation 包，该包可以从 <http://hoopajoo.net/projects/xautomation.html> 获得。它可以模拟一个用户，向 X 服务器发送任意的 X Window 事件。下载 xautomation 压缩包，解压并编译后，就可以使用它来自动执行点击鼠标右键。

与对待 GIMP 不同，我们不能简单地通过测量 nautilus 使用的 CPU 时间来计算创建 100 个弹出菜单所需的时间。这主要是因为 nautilus 不能在菜单被打开前立即启动，也不能打开后立即结束。我们将使用墙钟时间来查看完成这个任务需要耗时多久。这要求在进行测试时，系统没有运行任何其他的事情。

清单 11.1 给出了 xautomation 命令的 shell 脚本，用于在 nautilus 文件浏览器中打开 100 个弹出菜单。运行测试时，我们必须确保面对的是 nautilus 窗口，这样就不会有点击实际上是打开了文件夹的弹出菜单，而是所有的弹出窗口都出现在背景上。这是非常重要的，因为不同弹出菜单的代码路径可能是完全不同的。

清单 11.1

```
#!/bin/bash
for i in `seq 1 100`;
do
    echo $i
    ./xte 'mousemove 100 100' 'mouseclick 3' 'mouseclick 3'
    ./xte 'mousemove 200 100' 'mouseclick 3' 'mouseclick 3'
done
```

清单 11.1 中的命令把光标放置在 X 屏幕的 (100, 100) 处，点击鼠标右键（按钮 3）。这个操作打开一个菜单。然后它们再次右击鼠标，这次是关闭该菜单。之后移动到 X 位置 (200, 100)，重复该过程。

接下来，我们用 time 查看完成这个 100 次迭代脚本花费了多少时间。这就是我们的基线时间。在我们进行优化时，我们将把优化结果与这个时间作比较看看是否有所改进。我的笔记本为普通版 Fedora 2 的 nautilus，该条件下的基准时间为 26.5 秒。

最后，我们要为优化选择一个目标。实现这个目的的一个简单方法是找到一个已经具备快速弹出菜单的应用程序，查看它打开 100 次弹出菜单消耗的时间。xterm 是这方面一个很好的例子，它有非常敏捷的菜单。虽然这些菜单不像 nautilus 的一样复杂，但是它们至少应该被看作是菜单速度的上限。

xterm 的弹出菜单操作略有不同，因此我们需要稍微修改一下创建 100 个弹出菜单的脚本。当 xterm 创建一个弹出菜单时，需要按下左控制键，因此我们必须对自动化脚本进行轻微的改动。该脚本如清单 11.2 所示。

清单 11.2

```
#!/bin/bash
for i in `seq 1 100`;
do
    echo $i
    ./xte 'keydown Control_L' 'mousemove 100 100' 'mouseclick 3' 'mouseclick 3'

    ./xte 'keydown Control_L' 'mousemove 200 100' 'mouseclick 3' 'mouseclick 3' done
```

运行 xterm 并对创建弹出菜单进行计时，xterm 完成该脚本的时间约为 9.2 秒。nautilus 有相当大的提升空间（几乎为 17 秒）。期望创建 nautilus 复杂弹出菜单的速度与 xterm 的一样可能是不合理的，因此我们保守地将目标设定为 10%，即 3 秒钟。我们希望能够做得比这更好，或者至少弄清楚速度为什么不能更快了。

11.4 为性能追踪配置应用程序

调查的下一步是为性能追踪设置应用程序。对 GIMP 我们立即重编译了应用程序，而对 nautilus 我们将采取不同的方法。由于要依赖于很多不同的共享库，因此很难指出究竟是哪些地方需要重编译。我们不进行重编译，取而代之的是下载并安装每个应用程序和库的调试信息。对 Fedora 和企业版 Linux，Red Hat 提供了一组 debuginfo rpm，其中包含了应用程序编译时，编译器生成的全部符号信息和源代码。每个二进制包或库都有相应的 debuginfo rpm 提供其调试信息。这使得 Red Hat 可以传输二进制文件，而不用带上占用磁盘空间的调试信息。但是，它又允许开发人员或那些调查性能问题的人下载并使用合适的 debuginfo 包。在这种情况下，Red Hat 版本的 oprofile 也可以识别 debuginfo 包，并在分析应用程序（如一个 nautilus）和库（如 gtk）时获取符号。本例中，我们将下载 gtk、nautilus、glib 和内核的 debuginfo。如果 oprofile 发现一个库占用了大量的周期，但又不允许你分析这些库（oprofile 输出“no symbols”），这就表示没有安装该库的调试信息。我们可以下载并安装适合该库的 debuginfo 包，之后 oprofile 将访问调试信息，并可以把事件映射回原始函数和源代码行。

11.5 安装和配置性能工具

追踪的下一步是按照调查问题所需的性能工具。如同对 GIMP 性能追踪所做的一样，我们将安装 oprofile 和 ltrace。本例中，我们还将下载并安装 gdb（如果它还未安装）。gdb 可以让我们观察运行中应用程序的一些动态方面。

11.6 运行应用程序和性能工具

下面，我们运行应用程序，并使用性能工具进行测量。由于我们已经假设多个不同的进程和库之间的复杂交互可能是问题产生的原因，我们将从 oprofile 开始，看看它给了些什么信息。

我们希望 oprofile 只测量弹出菜单打开时发生的事件，所以我们将使用如清单 11.3 所示的命令行在脚本（名为 script.sh）开始运行的前一刻立即启动分析，在结束运行的最后一刻立即停止分析，该脚本打开和关闭弹出菜单 100 次。

清单 11.3

```
opcontrol -start ; ./script.sh ; opcontrol -stop
```

收集分析信息之后运行 oreport，得到如清单 11.4 所示的信息。

清单 11.4

```
CPU: CPU with timer interrupt, speed 0 MHz (estimated)
```

```
Profiling through timer interrupt
```

```
TIMER:0!
```

```
samples | %
```

```
3134 27.1460 /usr/lib/libgobject-2.0.so.0.400.0
```

```
1840 15.9376 /usr/lib/libglib-2.0.so.0.400.0
```

```
1303 11.2863 /lib/tls/libc-2.3.3.so
```

```
1048 9.0775 /lib/tls/libpthread-0.61.so
```

```
900 7.7956 /usr/lib/libgtk-x11-2.0.so.0.400.0
```

```
810 7.0160 /usr/X11R6/bin/Xorg
```

```
719 6.2278 /usr/lib/libgdk-x11-2.0.so.0.400.0
```

```
334 2.8930 /usr/lib/libpango-1.0.so.0.399.1
```

```
308 2.6678 /lib/ld-2.3.3.so
```

```
298 2.5812 /usr/X11R6/lib/libX11.so.6.2
```

```
228 1.9749 /usr/lib/libbonoboui-2.so.0.0.0
```

```
152 1.3166 /usr/X11R6/lib/libXft.so.2.1.2
```

正如你所看到的，耗时分散在多个不同的库中。可惜的是，并不完全清楚这些调用是属于哪个应用程序的。尤其是，我们不知道哪个进程调用了 libgobject 库。幸运的是，oprofile 提供了一种方法来记录应用程序运行时使用的共享库函数。清单 11.5 显示了如何配置 oprofile 的样本采集来按库分离样本，这就意味着 oprofile 会把共享库中采集的样本按照调用库的程序来划分。

清单 11.5

```
opcontrol -p library; opcontrol ---reset
```

(用清单 11.3 的命令行) 重新运行测试后, oreport 分离了每个应用程序的库样本, 如清单 11.6 所示。

清单 11.6

```
[root@localhost menu_work]# oreport -f

CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt

  TIMER:0|
samples|    %

-----
8172 61.1311 /usr/bin/nautilus
  TIMER:0|
samples|    %

-----
3005 36.7719 /usr/lib/libgobject-2.0.so.0.400.0
1577 19.2976 /usr/lib/libglib-2.0.so.0.400.0
826 10.1077 /lib/tls/libpthread-0.61.so
792 9.6916 /lib/tls/libc-2.3.3.so
727 8.8962 /usr/lib/libgtk-x11-2.0.so.0.400.0
391 4.7846 /usr/lib/lib gdk-x11-2.0.so.0.400.0
251 3.0715 /usr/lib/libpango-1.0.so.0.399.1
209 2.5575 /usr/lib/libbonoboui-2.so.0.0.0
140 1.7132 /usr/X11R6/lib/libX11.so.6.2
75 0.9178 /usr/X11R6/lib/libXft.so.2.1.2
54 0.6608 /usr/lib/libpangoft-1.0.so.0.399.1
23 0.2814 /usr/lib/libnautilus-private.so.2.0.0
....
```

如果我们更深入的探究 libgobject 和 libglib 库, 就能清楚地看到哪些函数被调用了, 如清单 11.7 所示。

清单 11.7

```
[root@localhost menu_work]# oreport -lf /usr/lib/libgobject-
2.0.so.0.400.0
...
CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt
```

```
samples %      image name          app name
symbol name

394      11.7753 /usr/lib/libgobject-2.0.so.0.400.0 /usr/bin/nautilus-
g_type_check_instance_is_a
248      7.4118 /usr/lib/libgobject-2.0.so.0.400.0 /usr/bin/nautilus-
g_bsearch_array_lookup_fuzzy
208      6.2164 /usr/lib/libgobject-2.0.so.0.400.0 /usr/bin/nautilus-
g_signal_emit_valist
162      4.8416 /usr/lib/libgobject-2.0.so.0.400.0 /usr/bin/nautilus-
signal_key_cmp
147      4.3933 /usr/lib/libgobject-2.0.so.0.400.0 /usr/bin/nautilus-
signal_emit_unlocked_R
137      4.0944 /usr/lib/libgobject-2.0.so.0.400.0 /usr/bin/nautilus-
__i686.get_pc_thunk.bx
90       2.6898 /usr/lib/libgobject-2.0.so.0.400.0 /usr/bin/nautilus-
g_type_value_table_peek
85       2.5403 /usr/lib/libgobject-2.0.so.0.400.0 /usr/bin/nautilus-
type_check_is_value_type_U
```

...

```
opreport -lf /usr/lib/libglib-2.0.so.0.400.0
```

CPU: CPU with timer interrupt, speed 0 MHz (estimated)

Profiling through timer interrupt

```
samples %      image name          app name
symbol name

385      18.0075 /usr/lib/libglib-2.0.so.0.400.0 /usr/bin/nautilus-
g_hash_table_lookup
95       4.4434 /usr/lib/libglib-2.0.so.0.400.0 /usr/bin/nautilus-
g_str_hash
78       3.6483 /usr/lib/libglib-2.0.so.0.400.0 /usr/bin/nautilus-
g_data_set_internal
78       3.6483 /usr/lib/libglib-2.0.so.0.400.0 /usr/bin/nautilus-
g_pattern_ph_match
70       3.2741 /usr/lib/libglib-2.0.so.0.400.0 /usr/bin/nautilus-
__i686.get_pc_thunk.bx
```

...

从oprofile的输出可以看出nautilus在libgobject库上花费了相当多的时间，尤其是g_type_check_instance_is_a函数。但是，目前还不清楚nautilus文件管理器中哪些函数进行了这些调用。实际上，这些函数可能不是直接由nautilus调用的，而是由调用这nautilus的其他共享库调用的。

接下来我们使用共享库追踪器ltrace尝试找出哪些库调用开销最大，以及最终是谁调用了g_

type_check_instance_is_a 函数。我们主要关注的是 nautilus 调用了哪些函数，而不是精确的计时信息，它只有在打开弹出菜单一次而非 100 次时才是重要的。因为 ltrace 会捕捉单次运行的每一个共享库调用，所以，如果我们创建 100 个弹出菜单，ltrace 就会显示相同的分析信息 100 次。

捕捉共享库使用情况信息的过程类似于我们在 GIMP 中做的。我们首先正常启动 nautilus。之后，在打开一个弹出菜单之前，使用如下 ltrace 命令附加到 nautilus 进程：

```
ltrace -c -p <pid_of_nautilus>.
```

在 nautilus 背景上点击右键，打开菜单，接着立即用组合键 <Ctrl-C> 中止 ltrace 进程。跟踪弹出菜单后，我们得到如清单 11.8 所示的汇总表。

清单 11.8

% time	seconds	usecs/call	calls	function
32.75	0.109360	109360	1	bonobo_window_add_popup
25.88	0.086414	257	335	g_cclosure_marshal_VOID_VOID
14.98	0.050011	145	344	g_cclosure_marshal_VOID_OBJECT
8.85	0.029546	29546	1	eel_pop_up_context_menu
5.25	0.017540	604	29	gtk_widget_destroy
5.22	0.017427	1340	13	g_cclosure_marshal_VOID_POINTER
2.96	0.009888	41	241	g_free
0.93	0.003101	3101	1	gtk_widget_get_ancestor
0.45	0.001500	1500	1	gtk_widget_show
0.45	0.001487	495	3	nautilus_icon_container_get_type
0.43	0.001440	41	35	g_type_check_instance_cast
0.38	0.001263	1263	1	nautilus_file_list_free
0.34	0.001120	1120	1	gtk_widget_get_screen
0.29	0.000978	46	21	gdk_x11_get_xatom_by_name
0.25	0.000845	42	20	g_object_unref
0.11	0.000358	89	4	g_type_check_class_cast
0.09	0.000299	42	7	g_type_check_instance_is_a
0.09	0.000285	40	7	g_cclosure_marshal_VOID_ENUM
0.06	0.000187	37	5	strcmp
0.04	0.000126	126	1	g_cclosure_marshal_VOID_STRING
0.03	0.000093	93	1	gtk_menu_get_type
0.03	0.000087	43	2	bonobo_window_get_type
0.02	0.000082	82	1	nautilus_icon_container_get_selection
0.02	0.000082	41	2	gtk_bin_get_type
0.02	0.000081	40	2	gtk_widget_get_type
0.02	0.000080	80	1	gtk_menu_set_screen
0.02	0.000072	72	1	g_signal_connect_object
0.02	0.000071	71	1	nautilus_file_set_boolean_metadata
0.01	0.000041	41	1	nautilus_file_list_ref
0.01	0.000040	40	1	eel_g_list_exactly_one_item
0.01	0.000038	38	1	nautilus_icon_container_set_keep_aligned
100.00	0.333942		1085	total

在这个表中，我们可以看到一些有趣的地方。ltrace 在顶部显示了一个与 oprofile 完全不同的函数。这主要是因为 oprofile 和 ltrace 测量的内容略有不同。oprofile 显示的是实际函数耗费的时间，但不包括子函数。ltrace 则仅显示外部库调用完成耗费的时间。如果库函数反过来还要调用其他函数，那么 ltrace 不会记录每一个的时间。事实上，目前它甚至都不会检测或显示发生的这些其他库调用。

在这种特定情况下，oprofile 所说的 libgobject 的最热函数（即 `g_type_check_instance_is_a`）几乎根本不会出现在 ltrace 分析信息中。即使这个函数是共享库的一部分，对它的调用也没有显示在 ltrace 的输出中。ltrace 不能显示跨库调用，也不能显示库内调用。ltrace 只能跟踪外部库调用或者一个共享库内的应用程序调用。当一个库调用一个内部函数时，ltrace 不能跟踪此调用。在这种情况下，所有前缀为 `g_` 的函数实际上都是 libgobject 库的一部分。它们其中的任何一个调用 `g_type_check_instance_is_a`，ltrace 都无法检测到。

ltrace 提供的最重要的信息是我们的应用程序调用的几个库，我们将对它们进行调查。我们可以弄清楚库是在哪里被调用的，以及为什么这个库调用消耗了所有的时间。

11.7 编译和检查源代码

现在，我们有了关于占用全部时间的应用程序调用的一些信息，我们将下载源代码并对其进行编译。到目前为止，我们所有的分析都可以通过使用 Red Hat 提供的二进制包进行。但是，现在我们需要深入源代码，检查热点函数被调用的原因，找出原因后，修改源代码来缓解性能问题。如同对 GIMP 所做的，重编译时，在调用配置脚本之前，我们设置 `CFLAGS` 为 `-g` 以生成调试符号。

本例中，我们为 nautilus 下载并安装 Red Hat 的源 rpm，它把 nautilus 源代码放在 `/usr/src/redhat/SOURCES` 中。通过使用 Red Hat 的源代码包，我们有了 Red Hat 在包内用于创建二进制文件的准确源代码和补丁。重要的是研究用于创建我们一直调查的二进制文件的源代码，因为不同的版本就可能有不同的性能特点。提取源代码后，我们就可以开始找出 `bonobo_window_add_popup` 是在哪里调用的。可以用清单 11.9 中的命令搜索 nautilus 目录下的所有源文件。

清单 11.9

```
[nautilus ]$ find -type f | xargs grep bonobo_window_add_popup

./src/file-manager/fm-directory-view.c: bonobo_window_add_popup\
(get_bonobo_window (view), menu, popup_path);
```

幸运的是，看上去 bonobo_window_add_popup 似乎只被一个函数 create_popup_menu 调用，如清单 11.10 所示。

清单 11.10

```
static GtkMenu *create_popup_menu (FMDirectoryView *view,
                                  const char *popup_path)
{
    GtkMenu *menu;

    menu = GTK_MENU (gtk_menu_new ());
    gtk_menu_set_screen (menu, gtk_widget_get_screen (GTK_WIDGET (view)));

    gtk_widget_show (GTK_WIDGET (menu));

    bonobo_window_add_popup (get_bonobo_window (view), menu, popup_path);

    g_signal_connect_object (menu, "hide",
                           G_CALLBACK (popup_menu_hidden),
                           G_OBJECT (view),
                           G_CONNECT_SWAPPED);
    return menu;
}
```

反过来，该函数被其他两个函数调用，fm_directory_view_pop_up_background_context_menu 和 fm_directory_view_pop_up_selection_context_menu。在这两个函数上都添加 printf，我们就可以确定当右键点击窗口时调用的是哪一个了。之后对 nautilus 进行重编译，并运行它，在窗口背景上点击右键。nautilus 输出 fm_directory_view_pop_up_background_context_menu，由此我们知道在窗口背景上打开弹出菜单时调用的就是这个函数。该函数的源代码如清单 11.11 所示。

清单 11.11

```
void fm_directory_view_pop_up_background_context_menu (FMDirectoryView
                                                       *view, GdkEventButton *event)
{
    g_assert (FM_IS_DIRECTORY_VIEW (view));

    /* Make the context menu items not flash as they
       * update to proper disabled,
       * etc. states by forcing menus to update now.
    */
    update_menus_if_pending (view);
```

```
eel_pop_up_context_menu
(create_popup_menu (view, FM_DIRECTORY_VIEW_POPUP_PATH_BACKGROUND),
 EEL_DEFAULT_POPUP_MENU_DISPLACEMENT,
 EEL_DEFAULT_POPUP_MENU_DISPLACEMENT, event);
}
```

现在我们已经准确缩小了弹出菜单创建与显示的范围，我们就可以开始弄清楚究竟是哪些部分消耗了这些时间，又是哪些部分最终调用了 `g_type_check_instance_is_a` 函数，即oprofile 所说的热点函数。

11.8 使用gdb生成调用跟踪

检索应用程序调用了哪些函数的两种不同的工具提供给我们哪个函数是热点函数的信息也是不一样的。我们推测 ltrace 报告的高级函数调用了 oprofile 报告的低级函数。要是有性能工具能告诉我们到底是哪些函数调用了 `g_type_check_instance_is_a` 以验证这个理论就好了。

虽然没有 Linux 性能工具向我们显示究竟是哪些函数调用了某个特定函数，但 gprof 应该能显示这个回调信息，不过这需要用 -pg 选项重新编译应用程序及其依赖的所有库。对 nautilus 而言，它依赖于 72 个共享库，这让任务变得无比艰巨，因此，我们必须寻找其他的解决方法。较新版本的 oprofile 也可以提供这类信息，但是由于 oprofile 只进行定期采样，所以它还是不能解释对任意给定函数的每一次调用。

幸运的是，我们可以创造性地利用 gdb 来提取信息。用 gdb 来跟踪应用程序会极大地降低运行速度，不过，我们并不真正在意跟踪是否需要很长的时间。我们感兴趣的是发现一个特定函数被调用的次数而不是它被调用的时长，因此，运行需要很长的时间也是可以接受的。幸好弹出菜单的创建是毫秒级的，即便使用 gdb 使其速度慢了 1000 倍，提取全部跟踪信息也仍然只需要 15 分钟。信息的价值远远超过了我们等待检索的价值。

尤其是要找出是哪些函数调用了 `g_type_check_instance_is_a` 时，我们需要用到 gdb 几个不同的功能。首先，要使用 gdb 设置断点的功能。然后，要用到 gdb 在该断点上用 bt 生成回溯的功能。要弄清楚哪些函数调用了 `g_type_check_instance_is_a`，我们确实需要这两个功能，不过手工记录信息并继续是单调乏味的。我们要在函数的每次 gdb 中断之后键入 `bt; cont`。

要解决这个问题可以使用 gdb 的另一个功能。在遇到一个断点时，gdb 可以执行一组给定的命令。通过使用 command 命令，我们可以告诉 gdb，每次在函数中遇到断点时，它都要执行 `bt; cont`。现在，回溯自动显示，且应用程序每次遇到 `g_type_check_instance_is_a` 时都会继续运行。

当实际运行跟踪时需要进行分离。我们可以只在 nautilus 开始执行时在 `g_type_check_instance_is_a` 中设置断点，当其被其他函数调用时，`gdb` 就会显示跟踪信息。由于我们只关注创建弹出菜单时被调用的那些函数，我们想要将跟踪只限制在弹出菜单被创建的时候。要做到这一点，需在 `fm_directory_view_pop_up_background_context_menu` 函数的开始和结束的地方设置另一个断点。当到达第一个断点时，我们打开 `g_type_check_instance_is_a` 中的回溯。当到达第二个断点时，退出调试器。这就把生成的回溯信息限制在了创建弹出菜单的时候。最后，我们希望能够保存这些回溯信息以便后期处理。我们可以使用 `gdb` 将其输出记录到文件的功能，为后续保存信息。为了提取这些信息而传递给 `gdb` 的命令如清单 11.12 所示。

清单 11.12

```

# Prevent gdb from stopping after a screenful of output
set height 0
# Turn on output logging to a file (default: gdb.txt)
set logging on
# Turn off output to the screen
set logging redirect on
# Stop when a popup menu is about to be created
break fm-directory-view.c:5730
# Start the application
run
# When we've stopped at the preceding breakpoint, setup
# the breakpoint in g_type_check_instance_is_a
break g_type_check_instance_is_a
# When we reach the breakpoint, print a backtrace and exit
command
bt
cont
end
# break after the popup was created and exit gdb
break fm-directory-view.c:5769
command
quit
end
# continue running
cont

```

运行这些 `gdb` 命令，打开弹出菜单，`gdb` 运行几分钟后，创建了一个 33MB 的文件，包含了调用 `g_type_check_instance_is_a` 的函数的全部回溯信息。清单 11.13 给出了其中的一个样本。

清单 11.13

```

Breakpoint 2, g_type_check_instance_is_a (type_instance=0x9d2b720,
iface_type=164410736) at gtype.c:31213121      if (!type_instance || !type_instance->g_class)

#1 0x08099f09 in fm_directory_view_pop_up_background_context_menu
(view=0x9d2b720, event=0x9ceb628)
    at fm-directory-view.c:5731

#2 0x080a2911 in icon_container_context_click_background_callback
(container=0x80c5a2b, event=0x9ceb628,
icon_view=0x9d2b720) at fm-icon-view.c:2141

#3 0x00da32be in g_cclosure_marshall_VOID_POINTER (closure=0x9d37620,
return_value=0x0, n_param_values=2,
param_values=0xfef67320, invocation_hint=0xfef67218,
marshal_data=0x0) at gmarshal.c:601

#4 0x00d8e160 in g_closure_invoke (closure=0x9d37620,
return_value=0x9d2b720, n_param_values=164804384,
param_values=0x9d2b720, invocation_hint=0x9d2b720) at gclosure.c:437

#5 0x00da2195 in signal_emit_unlocked_R (node=0x9d33140, detail=0,
instance=0x9d35130, emission_return=0x0,
instance_and_params=0xfef67320) at gsignal.c:2436

#6 0x00da1157 in g_signal_emit_valist (instance=0x9d35130, signal_id=0,
detail=0, var_args=0xfef674b0 "") at gsignal.c:2195

...

```

尽管这个信息非常详细，但它的格式实在是不易于阅读。如果每个回溯都在单独一行上，且用箭头分隔每个函数就会好一点。去掉对 `fm_directory_view_pop_up_background_context_menu` 调用上面的回溯信息也是很好的，因为我们知道每一个这样的调用都有同样的回溯信息。利用清单 11.14 所示的 Python 程序 `slice.py` 可以做到这一点。该程序用 `gdb` 生成的详细输出文件，对每个调用 `fm_directory_view_pop_up_background_context_menu` 的函数创建一个格式良好的调用跟踪。

清单 11.14

```

#!/usr/bin/python

import sys
import string
funcs = ""
stop_at = "fm_directory_view_pop_up_background_context_menu"
for line in sys.stdin:
    parsed = string.split(line)
    if (line[:1] == "#"):
        if (parsed[0] == "#0"):
            funcs = parsed[1]
    elif (parsed[3] == stop_at):

```

```

    print funcs
    funcs = ""
else:
    funcs = parsed[3] + ".->" + funcs

```

用清单 11.15 中的命令行在这个 Python 程序中运行 gdb.txt 文件，会得到一个合并度更高的输出，清单 11.16 给出了一个例子。

清单 11.15

```
cat gdb.txt | ./slice.py > backtrace.txt
```

清单 11.16

```

.....
create_popup_menu->gtk_widget_show->g_object_notify->g_type_check_
instance_is_a
create_popup_menu->gtk_widget_show->g_object_notify->g_object_ref->g_
type_check_instance_is_a
create_popup_menu->gtk_widget_show->g_object_notify->g_object_
notify_queue_add->g_param_spec_get_redirect_target->g_type_check_
instance_is_a
create_popup_menu->gtk_widget_show->g_object_notify->g_object_notify_
queue_add->g_param_spec_get_redirect_target->g_type_check_instance_is_a
create_popup_menu->gtk_widget_show->g_object_notify->g_object_unref->g_
type_check_instance_is_a
create_popup_menu->gtk_widget_show->g_object_unref->g_type_check_
instance_is_a
...

```

由于输出行很长，本书显示时它们已经换行，但是在文本文件中一个回溯一行。每行都用 g_type_check_instance_is_a 函数结束。因为一个回溯只占用一行，我们就可以用一些常见的 Linux 工具来抽取回溯信息，比如 wc，该工具可以用于计算特定文件的行数。

首先，我们看看有多少对 g_type_check_instance_is_a 函数的调用。该值与回溯的数量相同，因此也与 backtrace.txt 文件的行数相同。清单 11.17 显示了被精简回溯文件调用的 wc 命令。第一个数字表示的是文件的行数。

清单 11.17

```
[ezolt@localhost menu_work]$ wc backtrace.txt
6848 6848 3605551 backtrace.txt
```

可以看到，只是创建弹出菜单，函数就被调用了 6848 次。接下来看看这些函数调用中有多少是代表 bonobo_window_add_popup 发起的，查看清单 11.18。

清单 11.18

```
[ezolt@localhost menu_work]$ grep bonobo_window_add_popup backtrace.txt | wc  
6670 6670 3558590
```

bonobo_window_add_popup 负责了 6670 次热点函数的调用。查看 backtrace.txt 文件发现，其中的一些是直接调用，而大多数则来自于它调用的其他函数。从这点来看，好像 bonobo_window_add_popup 确实应为大部分消耗掉的 CPU 时间负责。不过，我们还是要对这种情况进行确认。

11.9 找到时间差异

现在我们已经缩小到了是由哪些函数创建了菜单，我们想要弄清楚哪些部分占用了所有的时间，哪些部分相对是轻量级的。有一个很好的方法能做到这一点，并且不需要使用任何性能工具，即只需禁用代码段，查看它是如何改变性能的。尽管这会导致 nautilus 功能失常，但它至少能表明哪些函数占用了所有的时间。

首先我们要从获取基线开始，因为与 Red Hat 提供的相比，正在测试的二进制文件已经用不同的标志编译过了。和之前一样对脚本计时。此时，在我们自己编译的版本中，运行 100 次迭代花费了 30.5 秒。接着，我们注释掉 eel_pop_up_context_menu 调用。这会告诉我们 nautilus 花了多少时间来检测鼠标点击并决定应创建一个右键菜单。即使完全优化掉这些函数中的所有命令，也无法运行地更快。这种情况下，运行全部 100 次迭代耗时 7.6 秒。接下来，我们注释掉 bonobo_window_add_popup，看看调用 ltrace 所说的占用最多时间的函数实际上消耗了多少时间。如果注释掉 bonobo_window_add_popup，完成 100 次迭代耗时 21.9 秒。这就是说，如果我们优化掉 bonobo_window_add_popup，总的运行时间可以减少大约 8 秒，性能提升了几乎 25%。

11.10 尝试一种可能的解决方案

所以，如我们所见的一样，bonobo_window_add_popup 是开销很大的函数，但又是每次想要创建弹出菜单时必须调用的函数。如果用同样的参数重复调用该函数，说不定可以缓存其从初始调用返回的值，并且在之后的每一次调用时都使用该值，以代替对该昂贵函数的重复调用。清单 11.19 显示了一个重写函数的例子，来完成这个功能。

清单 11.19

```
void  
fm_directory_view_pop_up_background_context_menu (FMDirectoryView *view,
```

```

        GdkEventButton *event)
{
    /* Primitive Cache */

    static FMDirectoryView *old_view = NULL;
    static GtkMenu *old_menu = NULL;

    g_assert (FM_IS_DIRECTORY_VIEW (view));

    /* Make the context menu items not flash as they update to proper disabled,
     * etc. states by forcing menus to update now.
     */
    if ((old_view != view) || view->details->menu_states_untrustworthy)
    {
        update_menus_if_pending (view);
        old_view = view;
        old_menu = create_popup_menu (view, FM_DIRECTORY_VIEW_POPUP_PATH_BACKGROUND);
    }

    eel_pop_up_context_menu (old_menu,
                            EEL_DEFAULT_POPUP_MENU_DISPLACEMENT,
                            EEL_DEFAULT_POPUP_MENU_DISPLACEMENT,
                            event);
}

```

本例中，我们记住了上一次生成的菜单。如果呈现在同一视图中，且我们认为该视图的菜单没有改变，那么只需要使用与上一次相同的菜单，而不用创建新的。这个技术并不复杂，如果用户没有在同一个目录中重复打开弹出菜单就会出问题。举个例子，假如用户在目录 1 中打开了一个弹出菜单，然后又在目录 2 中打开一个，之后又在目录 1 里打开一个弹出菜单，那么 nautilus 还是会创建一个新菜单。可以建立一个简单的缓存，在菜单被创建时保存它们。打开菜单时，首先检查这些视图是否已经有在缓存中的菜单。如果是，就显示被缓存的菜单，否则将创建新菜单。这个缓存对一些特殊的目录非常有用，如桌面、计算机或主目录，在这些目录中用户很可能多次打开弹出菜单。应用上述解决方案后，对 100 次迭代计时，其所用时间减少到 24.0 秒。性能提升约 20%，接近于不创建任何菜单时可以获得的理论改进值（21.9 秒）。在不同目录中创建弹出菜单与预期一致。这个补丁似乎没有任何破坏性。

但是要注意，现在这只是一个测试解决方案。它需要被提交给 nautilus 开发人员，以确认该方案没有破坏任何功能，且适合添加。不过在整个追踪的过程中，我们已经确定了哪些函数慢，跟踪了它们在哪里被调用，创造了一个可能的解决方案，它在客观上改进了性能。请注意，改进是客观的，这一点很重要。也就是说，我们有确凿的数据证明新方法

更快，而不是简单的主观印象（只是说感觉更迅捷了）。大多数开发人员都想要这样的性能 bug 报告。

11.11 本章小结

本章我们确定了为什么应用程序的特定部分会有高延迟（nautilus 中的弹出菜单）。我们弄清楚了怎样将弹出菜单的创建进行自动化（xautomation），以及怎样扩展 nautilus 在创建弹出菜单时花费的时间（100 次迭代）。我们用 oprofile 来了解 nautilus 在哪些函数上消耗了全部时间，然后用 ltrace 和 gdb 确定哪些共享库调用应该对所有的调用负责。在明确了哪些库调用具有高成本后，我们试图减少或限制它们被调用的次数。针对这种情况，当一个新菜单被创建时，我们保存一个指针指向它，之后使用就可以避免不必要的重新分配。我们创建了一个建议补丁，然后在其上运行性能测试看看是否可以提升性能。性能提高了，并且功能看上去没有受到影响。下一步就是把这个补丁提交给 nautilus 开发人员讨论。本章侧重于优化单个应用程序的延迟，下一章给出的性能追踪则集中于解决系统级的性能问题。这种类型的追踪常常涉及对系统多个不同方面的调查，包括硬件（磁盘、网络和内存）与软件（应用程序、共享库和 Linux 内核）。

性能追踪 3 :系统级迟缓 (prelink)

本章包含的例子说明了如何用 Linux 性能工具寻找并修复影响整个系统而不是某个应用程序的性能问题。

阅读本章后，你将能够：

- 追踪是哪一个进程导致了系统速度的降低。
- 用 strace 调查一个不受 CPU 限制的进程的性能表现。
- 用 strace 调查一个应用程序是如何与 Linux 内核进行交互的。
- 提交描述性能问题的 bug 报告，以便创作者或维护者有足够的信息修改该问题。

12.1 调查系统级迟缓

本章我们调查系统级迟缓。首先会发现系统行为逐渐变慢，我们将用 Linux 性能工具找到确切的原因。这种类型的问题经常发生。作为一个用户或系统管理员，有时你可能会注意到 Linux 机器变得缓慢，或者需要很长时间才能完成任务。能够弄清楚机器变缓的原因是很有价值的。

12.2 确定问题

和前面一样，第一个步骤是准确找出我们要调查的问题。本例中，在使用 Fedora Core 2 桌面系统的条件下，我们将调查发生的周期性迟缓问题。通常情况下，桌面系统性能良

好，但偶尔磁盘会开始不停地读盘，其结果就是菜单和应用程序打开的时间非常长。过一会儿，磁盘研磨消退，而桌面系统行为又恢复正常。本章我们将弄清楚究竟是什么导致了这个问题，以及其发生的原因。

这个问题的类型与前两章的问题都不一样，因为一开始我们完全不知道系统的哪部分引发了问题。而在调查 GIMP 和 nautilus 的性能时，我们是知道应由哪个应用程序对问题负责的。在这种情况下，我们只有一个表现不佳的系统，理论上来说，性能问题可能存在于系统的任何部分。这种情况也是常见的。当遇到它时，重要的是用性能工具去实际追踪问题的原因，而不是仅仅去猜测原因并尝试解决方案。

12.3 找到基线/设置目标

还是和前面一样，第一步是确定问题的当前状态。

不过，对本例来说，这一点不容易做到。我们不知道问题什么时候发生或者它会持续多长时间，因此在没有进一步调查前，我们无法真正地设置基线。至于说到目标，理想状态下，我们希望问题完全消失，但是导致问题出现的可能是重要的 OS 功能，因此，可能无法完全消除它。

首先，针对问题为什么会出现我们需要多做一点调查，以便找到一个合理的基线。第一步是在迟缓发生时运行 top。这会给我们提供一个可能导致问题的进程列表，或者甚至也可能直指内核自身。

在这种情况下，如清单 12.1 所示，运行 top 并要求它只显示非空闲进程（top 运行时按 <D>）。

清单 12.1

```
top - 12:03:40 up 12 min,  7 users,  load average: 1.35, 0.98, 0.53
Tasks:  86 total,   2 running,  84 sleeping,   0 stopped,   0 zombie
Cpu(s):  2.3% us,  5.0% sy,  1.7% ni,  0.0% id, 91.0% wa,  0.0% hi,  0.0% si
Mem: 320468k total, 317024k used,    3444k free, 24640k buffers
Swap: 655192k total,      0k used, 655192k free, 183620k cached

          PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
        5458 root      34   19  4920 1944 2828 R  1.7  0.6  0:01.13 prelink
        5389 ezolt    17   0  3088  904 1620 R  0.7  0.3  0:00.70 top
```

清单 12.1 中的 top 输出有几个有意思的特性。第一，我们注意到没有进程占用 CPU，两个非空闲任务使用的 CPU 时间都不到 2%。第二，系统花费了 91% 的时间等待 I/O 的发生。第三，系统没有使用任何交换空间，因此磨盘不是由交换导致的。最后，有一个未知进程 prelink 在问题发生时正在运行。由于不清楚这个 prelink 命令是什么，因此我们先记住

应用的名字，之后再调查它。

我们的下一步是运行 vmstat，看看系统做了什么。清单 12.2 给出了 vmstat 的结果，并确认了我们在 top 中看到的。也就是，大约 90% 的系统时间用于等待 I/O。该清单还告诉我们磁盘子系统读数据的速度大概是 1000 块 / 秒。这个磁盘 I/O 量相当大。

清单 12.2

procs -----memory----- -swap-- -----io---- --system-- ----cpu----										in	cs	us	sy	id	wa
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
0	1	0	2464	24568	184992	0	0	689	92	1067	337	5	4	45	46
0	1	0	2528	24500	185060	0	0	1196	0	1104	1324	6	10	0	84
0	1	0	3104	24504	184276	0	0	636	684	1068	967	3	7	0	90
0	1	0	3160	24432	184348	0	0	1300	0	1096	1575	4	10	0	86
0	2	0	3488	24336	184444	0	0	1024	0	1095	1498	5	9	0	86
0	1	0	2620	24372	185188	0	0	980	0	1096	1900	6	12	0	82
0	1	0	3704	24216	184304	0	0	1480	0	1120	500	1	7	0	92
0	1	0	2296	24256	185564	0	0	1384	684	1240	1349	6	8	0	86
2	1	0	3320	24208	184572	0	0	288	0	1211	1206	63	7	0	30
0	1	0	3576	24148	184632	0	0	1112	0	1153	850	19	7	0	74

现在我们知道磁盘被频繁使用，内核花费了很多时间等待 I/O 和未知应用程序 prelink 的运行，我们可以开始弄清楚系统究竟在干嘛。

我们不能确认 prelink 是导致问题的原因，但是我们怀疑它是。明确 prelink 是否引起磁盘 I/O 的最简单方法就是“杀死” prelink 进程，看看磁盘的使用是不是消失了。（这在生产用计算机上是不可能的，不过我们使用的是个人桌面系统，所以可以快一点儿，不那么严格。）清单 12.3 显示了 vmstat 的输出，在其中一半的地方，我们终止了 prelink 进程。如你所见，prelink 被终止后，块读取降为零。

清单 12.3

procs -----memory----- -swap-- -----io---- --system-- ----cpu----															
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
0	1	122208	3420	13368	84508	0	0	1492	332	1258	1661	15	11	0	74
0	1	122516	3508	13404	85780	0	308	1188	308	1134	1163	5	7	0	88
0	2	123572	2616	13396	86860	0	1056	1420	1056	1092	911	4	6	0	90
0	1	126248	3064	13356	86656	0	2676	316	2676	1040	205	1	2	0	97
0	2	126248	2688	13376	87156	0	0	532	528	1057	708	2	5	0	93
0	0	126248	3948	13384	87668	0	0	436	4	1043	342	3	3	43	51
1	0	126248	3980	13384	87668	0	0	0	0	1154	426	3	1	96	0
0	0	126248	3980	13384	87668	0	0	0	0	1139	422	2	1	97	0
12	0	126248	4020	13384	87668	0	0	0	0	1023	195	9	0	91	0

由于 prelink 看着像是事由应用程序，现在可以开始调查它到底是什么，以及它为什么运行。清单 12.4 请求 rpm 告诉我们哪些文件是 prelink 构成包的组成部分。

清单 12.4

```
[root@localhost root]# rpm -qlf `which prelink'
/etc/cron.daily/prelink
/etc/prelink.conf
/etc/rpm/macros.prelink
/etc/sysconfig/prelink
/usr/bin/execstack
/usr/sbin/prelink
/usr/share/doc/prelink-0.3.2
/usr/share/doc/prelink-0.3.2/prelink.pdf
/usr/share/man/man8/execstack.8.gz
/usr/share/man/man8/prelink.8.gz
```

首先，我们注意到 prelink 包有一个日常运行的作业 cron。这解释了为什么性能问题的发生是周期性的。其次，我们注意到 prelink 包含了描述其功能的手册页和文档。手册页将 prelink 描述为可以预链接可执行文件与库的应用程序，以此减少它们的启动时间。（有点讽刺意味的是，用于提高性能的应用程序正在拉低系统的速度。）prelink 有两种运行模式。第一种模式使得所有指定的可执行文件和库都预链接，即使之前已经完成预链接了。（用 `--force` 或 `-f` 选项指定）。第二种是快速模式，prelink 只需查看库与可执行文件的 `mtime` 与 `ctime`，看看从上次预链接后是否发生了变化。（用 `--quick` 或 `-q` 选项指定。）通常，prelink 会把已预链接的可执行文件的所有 `mtime` 和 `ctime` 写入它自己的缓存。然后在快速模式中使用这些信息，以避免对已经预链接过的可执行文件再执行预链接。

检查 prelink 包中的 cron 条目显示，默认情况下，Fedora 系统同时使用了 prelink 的两种模式。每隔 14 天用完整模式调用 prelink。而在此期间的每一天，prelink 运行于快速模式。

对完整模式和快速模式下的 prelink 进行计时可以告诉我们最糟糕的情况有多慢（全预链接），以及使用快速模式后性能提升了多少。对 prelink 计时要小心，因为不同的运行可能会产生完全不同的时间。如果运行的应用程序使用了大量的磁盘 I/O，就必须让它多运行几次以便获得对其基准性能的精确指示。磁盘密集型应用程序第一次运行时，许多数据从它的 I/O 加载到缓存。程序第二次运行时，其性能就会好很多，因为要使用的数据已经在缓存中了，不再需要从磁盘读取。如果用第一次运行作为基线，你会被误导，以为调整后性能就提升了，但其实提升的真正原因是预热了缓存。只有多运行应用程序几次，你才可以预热好缓存，获得准确的基线。清单 12.5 显示了运行多次后，两种模式下 prelink 的结果。

清单 12.5

```
[root@localhost root]# time prelink -f -a
...
real    4m24.494s
user    0m9.552s
sys     0m14.322s
[root@localhost root]# time prelink -q -a
...
real    3m18.136s
user    0m3.187s
sys     0m3.663s
```

清单 12.5 中首先要注意的事实是快速模式与完整模式相比，并没有都快得那么多。这点值得怀疑，需要更多的调查。第二点事实强调了 top 的报告。prelink 只占用了一点 CPU 时间，其余的全都用来等待磁盘 I/O。

现在我们必须选择一个合理的目标。安装在 prelink 包中的 PDF 文件描述了预链接的过程。它也说明了完整模式需要花费几分钟，而快速模式需要花费几秒钟。作为目标，让我们试着把快速模式的时间减少到一分钟之内。即使我们可以优化快速模式，每隔 14 天仍然会遇到明显的磨盘，但是日常运行会有更多的改善。

12.4 为性能追踪配置应用程序

调查的下一步是为性能追踪配置应用程序。prelink 是一个小而独立的应用程序。事实上，它甚至不使用任何共享库。（它是静态链接的。）不过比较好的做法是，用全部的符号对其进行重编译，这样需要的时候就可以在调试器（gdb）中查看它。同样，这个工具用 configure 命令产生生成文件。我们必须下载源代码到 prelink，并用符号对它重新编译。我们可以从 Red Hat 再次下载 prelink 的源 rpm。源代码被安装在 /usr/src/redhat/SOURCES 下。一旦解压了 prelink 的源代码后，就可以如清单 12.6 所示对其进行编译。

清单 12.6

```
env CFLAGS=-g3 ./configure
gmake
```

prelink 完成配置与编译后，就可以利用我们编译的二进制文件来调查性能问题。

12.5 安装和配置性能工具

追踪的下一步是安装性能工具。本例中，无论是 ltrace 还是 oprofile 都派不上用场。

oprofile 用于剖析使用了大量 CPU 时间的应用程序，而 prelink 在运行时只使用了 3% 的 CPU 时间，所以 oprofile 对我们没有帮助。而 prelink 二进制文件是静态链接的，且不使用任何共享库，所以 ltrace 也帮不上我们。不过，系统调用追踪器 strace 可能会有帮助，因此我们需要安装它。

12.6 运行应用程序和性能工具

现在，我们终于可以开始分析 prelink 在不同模式下的性能特征了。正如你刚才看到的，prelink 没有花很多时间使用 CPU，相反，它把所有的时间都花在磁盘 I/O 上了。prelink 必须调用内核进行磁盘 I/O，因此我们用性能工具 strace 应该能追踪它的执行。prelink 的快速模式没有表现得比标准完整运行模式快很多，所以我们用 strace 比较这两个运行，看看是否有任何可疑行为出现。

首先，我们要求 strace 追踪较慢的完整运行 prelink。该运行创建了初始缓存，它将在 prelink 运行于快速模式时使用。起初，我们让 strace 显示 prelink 的系统调用汇总，看看其中的每一个要花多长时间完成。实现该操作的命令如清单 12.7 所示。

清单 12.7

```
[root@localhost prelink]# strace -c -o af_sum /usr/sbin/prelink -af
...
/usr/sbin/prelink: /usr/libexec/autopackage/luau-downloader.bin: Could
not parse '/usr/libexec/autopackage/luau-downloader.bin: error while
loading shared libraries: libuau.so.2: cannot open shared object file: No
such file or directory'
...
/usr/sbin/prelink: /usr/lib/mozilla-1.6/regchrome: Could not parse
'/usr/lib/mozilla-1.6/regchrome: error while loading shared libraries:
libxpcom.so: cannot open shared object file: No such file or directory'
...
```

清单 12.7 还是 prelink 输出的一个样本。在尝试预链接一些系统可执行文件和库时，prelink 显得有些吃力。这个信息在后面会变得很有价值，所以要记住它。

清单 12.8 显示了由清单 12.7 中的 strace 命令生成的输出汇总文件。

清单 12.8

```
[root@localhost prelink]# cat af_sum
execve("/usr/sbin/prelink", ["/usr/sbin/prelink", "-af"], /* 31 vars
*/ ) = 0
% time      seconds   usecs/call    calls    errors syscall
-----
```

77.87	151.249181	65	2315836	read
11.93	23.163231	55	421593	pread
3.59	6.976880	63	110585	pwrite
1.70	3.294913	17	196518	mremap
1.02	1.977743	32	61774	lstat64
0.97	1.890977	40	47820	1 open
0.72	1.406801	249	5639	vfork
0.35	0.677946	11	59097	close
...				
100.00	194.230415	3351032	5650	total

如同清单 12.8 所示，相当多的时间花在了系统调用 read 上。这是免不了的，prelink 需要找出哪些共享库被链接到了应用程序，这就要把部分可执行文件读入并进行分析。prelink 文档表明，当生成应用程序所需库的列表时，该程序实际上是由动态加载器用特殊模式启动的，之后用通道从可执行文件中读取信息。这就是为什么在分析中 read 也很高的原因。与之相反，我们希望快速版本中这样的调用会很少。

要查看快速版本的分析有何不同，我们在 prelink 的快速模式下运行同样的 strace 命令。实现该操作的 strace 命令如清单 12.9 所示。

清单 12.9

```
[root@localhost prelink]# strace -c -o aq_sum /usr/sbin/prelink -aq
```

清单 12.10 显示了运行于快速模式的 prelink 的 strace 分析信息。

清单 12.10

% time	seconds	usecs/call	calls	errors	syscall
47.42	3.019337	70	43397		read
26.74	1.702584	28	59822		lstat64
10.35	0.658760	163	4041		getdents64
5.52	0.351326	30	11681		pread
3.26	0.207800	21	9678	1	open
1.99	0.126593	21	5980	10	stat64
1.98	0.126243	12	10155		close
0.62	0.039335	165	239		vfork
...					
100.00	6.367230	154681	250	total	

和预期的一样，清单 12.10 表明快速模式执行了大量的 lstat64 系统调用。这些系统调用返回每个可执行文件的 mtime 和 ctime。prelink 在其缓存中查找，并把保存的 mtime 和

ctime 与可执行文件当前的 mtime 和 ctime 进行比较。如果可执行文件发生了变化，就对其启动 prelink；如果没有变化，则继续下一个可执行文件。实际上 prelink 大量调用 lstat64 是个好现象，这就表示 prelink 的缓存正在工作。不过，prelink 仍然大量调用 read 就不太好了。缓存应该记住已预链接的可执行文件，但不应试图分析它们。我们必须弄明白为什么 prelink 在尝试分析它们。最简单的方法就是以正常模式运行 strace。strace 将显示 prelink 发起的全部系统调用，并有希望澄清哪些文件被读取，以及解释为什么 read 被调用得如此频繁。清单 12.11 显示的是 strace 对快速 prelink 使用的命令。

清单 12.11

```
[root@localhost prelink]# strace -o aq_run /usr/sbin/prelink -aq
```

strace 的输出是一个 14MB 的文本文件 aq_run。浏览后发现 prelink 用 lstat64 检查了许多库和可执行文件。但是，它也揭示了使用 read() 的几种不同情况。如清单 12.12 所示，首先 prelink 读取一个 shell 脚本。由于 shell 脚本不是二进制 ELF 文件，因此它不能被预链接。

这些 shell 脚本从最初的完整系统 prelink 开始运行起就没有改变过，所以如果 prelink 的缓存能记录该文件不能被预链接的事实就好了。如果 ctime 和 mtime 不变，prelink 甚至都不会去尝试读取它们。（如果是上一个完整预链接中的 shell 脚本，且我们还没有碰过，它还是不能预链接。）

清单 12.12

```
[root@localhost prelink] # cat aq_run
...
open("/bin/unicode_stop", O_RDONLY|O_LARGEFILE) = 5
read(5, "#!/bin/sh\n# stop u", 18)      = 18
close(5)                                = 0
...
open("/bin/unicode_start", O_RDONLY|O_LARGEFILE) = 5
read(5, "#!/bin/bash\n# Enab", 18)       = 18
close(5)                                = 0
...
```

其次，在清单 12.13 中，我们观察到 prelink 试图操作一个静态链接的应用程序。该应用程序不依赖于任何共享库，因此试图对它进行预链接是没有意义的。prelink 的初始运行应该抓住一个事实，即这个应用程序不能被预链接，并将该信息保存到 prelink 的缓存中。在快速模式下，甚至不应该去尝试预链接这个二进制文件。

清单 12.13

```
[root@localhost prelink] # cat aq_run
...
open("/bin/ash.static", O_RDONLY|O_LARGEFILE) = 5

read(5, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\0\0\2\0", 18) = 18
fcntl64(5, F_GETFL)                      = 0x8000 (flags
O_RDONLY|O_LARGEFILE)
pread(5, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\0", 16, 0) = 16
pread(5, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0", 16, 0) = 16
pread(5, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\2\0\3\0\1\0\0\0\0\201\4"..., 52, 0) = 52
pread(5, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\0\2\0\3\0\1\0\0\0\0\0\201\4"..., 52, 0) = 52
pread(5, "\1\0\0\0\0\0\0\0\0\0\0\0\200\4\10\0\200\4\10\320d\7\0\320d\7"..., 128, 52) = 128
pread(5, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 920, 488632) = 920
close(5)
...

```

最后，在清单 12.14 中，我们看到 prelink 在读取一个二进制文件，该文件在最初的完整系统运行时存在预链接故障。在初始 prelink 输出中，我们看到了关于这个二进制文件的错误。当开始读取该文件时，它会捕捉其他库，并操作其中的每一个库及其依赖项。这会触发大量的读取。

清单 12.14

```
[root@localhost prelink] # cat aq_run
...
lstat64("/usr/lib/mozilla-1.6/regchrome", {st_mode=S_IFREG|0755, st_size=14444,
...}) = 0
open("/usr/lib/mozilla-1.6/regchrome", O_RDONLY|O_LARGEFILE) = 6
read(6, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\0\0\2\0", 18) = 18
fcntl64(6, F_GETFL)                      = 0x8000 (flags O_RDONLY|O_LARGEFILE)
pread(6, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0", 16, 0) = 16
...
open("/usr/lib/mozilla-1.6/libldap50.so", O_RDONLY|O_LARGEFILE) = 6
read(6, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0", 18) = 18
close(6)                                = 0
lstat64("/usr/lib/mozilla-1.6/libgtkxtbin.so", {st_mode=S_IFREG|0755, st_size=14268, ...}) = 0
open("/usr/lib/mozilla-1.6/libgtkxtbin.so", O_RDONLY|O_LARGEFILE) = 6
read(6, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0", 18) = 18

```

```

close(6) = 0
lstat64("/usr/lib/mozilla-1.6/libjsj.so", {st_mode=S_IFREG|0755, st_size=96752,
...}) = 0
open("/usr/lib/mozilla-1.6/libjsj.so", O_RDONLY|O_LARGEFILE) = 6
read(6, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 18) = 18
close(6) = 0
lstat64("/usr/lib/mozilla-1.6/mozilla-xremote-client", {st_mode=S_IFREG|0755, st_size=12896, ...}) = 0
lstat64("/usr/lib/mozilla-1.6/regxpcom", {st_mode=S_IFREG|0755, st_size=55144, ...}) = 0
lstat64("/usr/lib/mozilla-1.6/libgkgfx.so", {st_mode=S_IFREG|0755, st_size=143012, ...}) = 0
open("/usr/lib/mozilla-1.6/libgkgfx.so", O_RDONLY|O_LARGEFILE) = 6
read(6, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 18) = 18
close(6) = 0
...

```

对这种情况进行优化有些复杂。因为该二进制文件不是真正的问题（而是它链接的库 libxpcom.so），我们不能只是在缓存中将该可执行文件标记为坏。但是，如果我们将错误库的名称 libxpcom.so 与失败的可执行文件一起保存，它可能会检查该二进制文件与库的时间，只有当其中的一个发生变化，才会尝试再次预链接。

12.7 模拟解决方案

strace 揭示的信息显示，prelink 花了大量的时间试图打开并分析它可能无法预链接的二进制文件。测试缓存不可预链接的二进制文件是否能改善 prelink 的性能的最好方法是修改 prelink，将所有这些不可预链接的二进制文件都添加到 prelink 的初始缓存中。可惜的是，添加代码来缓存这些“不可预链接的”二进制文件会是一个复杂的过程，其中涉及大量的有关 prelink 应用程序的内部知识。更简单的方法是模拟缓存，将所有的不可预链接的二进制文件替换为已知的可预链接的二进制文件。这会导致运行快速模式时忽略之前全部的不可预链接二进制文件。若我们有一个工作缓存，这正是会发生的，因此，如果 prelink 能够缓存并忽略不可预链接的二进制文件，可以用它来评估我们将看到的性能提升。

开始实验，我们把 /usr/bin/ 中的所有文件都复制到 sandbox 目录下，并在这个目录上运行 prelink。这个目录包含了正常二进制文件、shell 脚本和其他不能被预链接的库。然后在 sandbox 目录上运行 prelink，并告诉它创建一个新缓存，而不是用系统缓存。如清单 12.15 所示。

清单 12.15

```
/usr/sbin/prelink -C new_cache -f sandbox/
```

接着，在清单 12.16 中，我们对快速模式的 prelink 运行了多久进行计时。需要多次运行，直到给出的结果达到一致。（第一次运行是为随后的每一次运行进行缓存热身。）清单 12.16 中的基线时间为 0.983 秒。为了显示我们的优化（改善缓存）调查是值得的，必须击败这个时间。

清单 12.16

```
time /usr/sbin/prelink -C new_cache -q sandbox/
real    0m0.983s
user    0m0.597s
sys     0m0.386s
```

然后，在清单 12.17 中，我们在这个 prelink 命令上运行 strace，记录 prelink 在 sandbox 目录中打开了哪些文件。

清单 12.17

```
strace -o strace_prelink_sandbox /usr/sbin/prelink -C new_cache -q
sandbox/
```

接着我们创建一个新目录 sandbox2，我们再次将 /usr/bin/ 中的所有二进制文件都复制到这个目录下。但是，我们用一个已知的好、能被预链接的二进制文件覆盖了在之前 strace 输出中 prelink “打开的”所有文件。我们把这个文件复制到全部的有问题的二进制文件，而不仅仅是删除它们，所以两个 sandbox 所含的文件数相同。建立第二个 sandbox 后，我们用清单 12.18 中的命令在这个新目录上运行完整版 prelink。

清单 12.18

```
[root@localhost prelink]#/usr/sbin/prelink -C new_cache2 -f sandbox2/
```

最后，对快速模式的运行计时，并将其与我们的基线进行比较。

同样的，这也需要运行多次，第一次运行也是缓存热身。清单 12.19 中，我们可以看到我们所做的，事实上，能看到性能的提升。执行 prelink 的时间从约 0.98 秒下降到约 0.29 秒。

清单 12.19

```
[root@localhost prelink]# time /usr/sbin/prelink -C new_cache2 -q
sandbox2/
real    0m0.292s
user    0m0.158s
sys     0m0.134s
```

接着，我们比较两次不同运行的 strace 输出，确认进行读取的次数，实际上，这个次数

减少了。清单 12.20 显示了 sandbox 的 strace 汇总信息，其中包含了 prelink 不能链接的二进制文件。

清单 12.20

execve("/usr/sbin/prelink", ["/usr/sbin/prelink", "-C", "new_cache", "-q", "sandbox/"], /* 20 vars */) = 0					
% time	seconds	usecs/call	calls	errors	syscall
62.06	0.436563	48	9133		read
13.87	0.097551	15	6504		lstat64
6.20	0.043625	18	2363	10	stat64
5.62	0.039543	21	1922		pread
3.93	0.027671	374	74		vfork
1.78	0.012515	9	1423		getcwd
1.65	0.011594	644	18		getdents64
1.35	0.009473	15	623	1	open
0.90	0.006300	8	770		close
....					
100.00	0.703400		24028	85	total

清单 12.21 显示了 sandbox 的 strace 汇总信息，其中 prelink 可以链接所有的二进制文件。

清单 12.21

execve("/usr/sbin/prelink", ["/usr/sbin/prelink", "-C", "new_cache2", "-q", "sandbox2/"], /* 20 vars */) = 0					
% time	seconds	usecs/call	calls	errors	syscall
54.29	0.088766	15	5795		lstat64
26.53	0.043378	19	2259	10	stat64
8.46	0.013833	8	1833		getcwd
6.95	0.011363	631	18		getdents64
2.50	0.004095	2048	2		write
0.37	0.000611	611	1		rename
0.26	0.000426	39	11	1	open
...					
100.00	0.163515		9973	11	total

如同你从清单 12.20 和清单 12.21 的不同中发现的一样，我们已经显著减少了目录中读取的次数。同时，我们还已经大大减少了预链接该目录所需的时间。缓存和回避不可预链接的可执行文件看起来是一种有前途的优化方法。

12.8 报告问题

我们已经发现了问题，并在系统软件相当低的层次上找到了可能的解决方案，因此，与作者一起解决这个问题是一个好主意。我们至少要提交一个 bug 报告以便作者知道这个问题的存在。提交用于发现问题的测试也有助于作者重现问题并增加修复问题的希望。本例中，我们将向 Red Hat 的 bugzilla (bugzilla.redhat.com) 追踪系统添加一个 bug 报告。(大多数其他发行版也有相似的 bug 追踪系统。) 我们的 bug 报告描述了我们遇到的问题以及发现的可能的解决方案。

在 bugzilla 中，我们首先搜索 prelink 的 bug 报告，看看是否已经有其他人提交了关于该问题的报告。本例中，没有人提交相关报告，因此我们输入如清单 12.22 所示的 bug 报告，等待作者或维护者的回复，或者问题修复。

清单 12.22

```

From Bugzilla Helper:

User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.6)
Gecko/20040510

Description of problem:
When running in quick mode, prelink does not cache the fact that some
binaries can not be prelinked. As a result it rescans them every time ,
even if prelink is running in quick mode. This causes the disk to grind
and dramatically slows down the whole system.

There are 3 types of executables that it retries during quick mode:
1) Static Binaries
2) Shell Scripts
3) Binaries that rely on unprelinkable binaries. (Such as OpenGL)

For 1&2, it would be nice if prelink cached that fact that these
executables can not be prelinked, and then in quick mode check their
ctime & mtime, and don't even try to read them if it already knows that
they can't be prelinked.

For 3, it would be nice if prelink recorded which libraries are causing
the prelink to fail (Take the OpenGL case for example), and record that
with the binary in the cache. If that library or the binary's ctime &
mtime haven't changed, then don't even try to prelink it. If things have
really changed, it will be picked up on the next run of "prelink -af".

```

Version-Release number of selected component (if applicable):

prelink-0.3.2-1

How reproducible:

Always

Steps to Reproduce:

1. Run `prelink -a -f` on a directory with shell scripts & other executables that can not be prelinked.
2. Strace "`prelink -a -q`", and look for the "reads".
3. Examine strace's output, and you'll see all of the reads that take place.

Actual Results: Shell script:

Un-prelinkable executable:

```
lstat64("/usr/lib/mozilla-1.6/regchrome", {st_mode=S_IFREG|0755,  
st_size=14444,  
...}) = 0  
open("/usr/lib/mozilla-1.6/regchrome", O_RDONLY|O_LARGEFILE) = 6  
read(6, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\2\0", 18) = 18  
fcntl64(6, F_GETFL) = 0x8000 (flags  
O_RDONLY|O_LARGEFILE)
```

```

pread(6, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0", 16, 0) = 16
...
open("/usr/lib/mozilla-1.6/libldap50.so", O_RDONLY|O_LARGEFILE) = 6
read(6, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\0\0\0", 18) = 18
close(6) = 0
lstat64("/usr/lib/mozilla-1.6/libgtkxtbin.so", {st_mode=S_IFREG|0755,
st_size=14268, ...}) = 0
open("/usr/lib/mozilla-1.6/libgtkxtbin.so", O_RDONLY|O_LARGEFILE) = 6
read(6, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\0\0\0", 18) = 18
close(6) = 0
lstat64("/usr/lib/mozilla-1.6/libjsj.so", {st_mode=S_IFREG|0755,
st_size=96752,
...}) = 0
open("/usr/lib/mozilla-1.6/libjsj.so", O_RDONLY|O_LARGEFILE) = 6
read(6, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\0\0\0", 18) = 18
close(6) = 0
lstat64("/usr/lib/mozilla-1.6/mozilla-xremote-client",
{st_mode=S_IFREG|0755, st_size=12896, ...}) = 0
lstat64("/usr/lib/mozilla-1.6/regexpcom", {st_mode=S_IFREG|0755,
st_size=55144, ...}) = 0
lstat64("/usr/lib/mozilla-1.6/libgkgfx.so", {st_mode=S_IFREG|0755,
st_size=143012, ...}) = 0
open("/usr/lib/mozilla-1.6/libgkgfx.so", O_RDONLY|O_LARGEFILE) = 6
read(6, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\0\0\0", 18) = 18
close(6) = 0
...

```

Expected Results: All of these should have been simple lstat checks rather than actual reads of the executables.

Additional info:

即便作者或维护者从未回复，把问题输入到 bug 追踪数据库仍旧是个好主意。问题及其可能的解决方案将被记录下来，某些热心的程序员也许会继续探索并修复该问题。

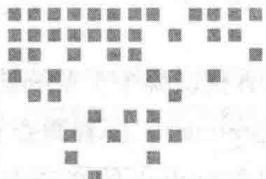
12.9 测试解决方案

由于我们还没有解决 prelink 代码中的问题，而是提交了 bug 报告，因此我们不能立即以原始基线为参照测试被修复的 prelink 时间。不过，如果作者或维护者能够实现提交的变化，或者甚至是找到了更好的方法进行优化，我们将能在更新版本出现时检查其性能。

12.10 本章小结

本章中，我们从一个表现不佳的系统开始，用性能工具找出哪个子系统被过度使用（如 `vmstat` 所示的磁盘子系统），以及哪个组件导致了问题（`prelink`）。接着调查 `prelink` 应用程序确定了它为什么有这么多的磁盘 I/O（用 `strace`）。我们在 `prelink` 的文档中发现缓存模式可以大大减少磁盘 I/O。研究了缓存模式的性能之后，我们发现它消除的磁盘 I/O 并不如预期那样多，其原因是它试图对不能预链接的文件进行预链接。之后，我们模拟了一个缓存，避免对不可预链接的文件尝试进行预链接，以此证明它明显减少了磁盘 I/O 的数量以及快速模式下 `prelink` 的运行时间。最后，我们向 `prelink` 的作者提交了 bug 报告，希望该作者能意识到问题并修复它。本章是 Linux 性能追踪的最后一章。

在下一章，也就是本书最后一章，我们将从更高层次来看看 Linux 的性能和性能工具。我们将回顾本书介绍的方法和工具，并考虑一些已成熟并可改进的 Linux 性能工具领域。



第 13 章

性能工具：下一步是什么

本章是对一些事情的思索，包括：Linux 性能工具的当前状态，哪些仍需要改进以及为什么 Linux 是当前一个相当不错的进行性能调查的平台。

阅读本章后，你将能够：

- 了解 Linux 性能工具箱的漏洞，以及一些理想的解决方案。
- 了解 Linux 作为性能调查平台的优势。

13.1 Linux 工具的现状

本书介绍了目前的 Linux 性能工具，以及怎样单独和一起使用它们来解决性能问题。就 Linux 的各方面而言，这些性能工具一直在发展，因此进行问题调查时，不断查看性能工具的帮助页或文档以确定其用法是否发生变化是一个好习惯。性能工具的基本功能很少改变，但常常会增加一些新功能，因此，对特定工具查看其最新发布的说明和文档是很有帮助的。

13.2 Linux 还需要什么样的工具

当我们了解一些 Linux 性能工具时，我们看到了整体性能调查功能中的一些漏洞。有些漏洞是由于内核限制造成的，有些漏洞则仅仅是由于没人编写一个工具来解决这个问题。但是，填补这些漏洞会让调查和修复 Linux 性能问题变得容易得多。

13.2.1 漏洞1：性能统计信息分散

一个明显的漏洞是Linux没有一个单一的工具为特定进程提供所有的相关性能统计信息。在原来的UNIX中，ps可以填补这一漏洞，在Linux中它也很不错，但却不包含其他商业UNIX实现提供的全部统计信息。有些统计信息对追踪性能问题是很有价值的，比如inblk(读入的I/O块)和oublk(写出的I/O块)，它们表示的是一个进程使用的磁盘I/O量；vcsw(自愿的上下文切换)和invcswo(非自愿的上下文切换)，它们通常表示一个进程进行CPU上下文切换；msgrecv(从通道和套接字接收的消息)和msgsnd(从通道和套接字发送的消息)，它们显示的是一个应用程序使用的网络和通道I/O量。一个理想工具应加上所有这些统计数据，并将迄今为止提到的许多性能工具(包括oprofile、top、ps、strace、ltrace和/proc文件系统)的功能组合到一个单一的应用程序中。用户应能在进程上使用这个单一的应用程序，抽取所有重要的性能统计数据。每个统计数据都能实时更新，使得用户可以在运行时调试应用程序。它会将某个调查方面的信息分组统计到相同位置。

举个例子，如果我正在调查内存使用情况，它就应该显示到底堆、栈、库、共享内存以及mmap使用了多少内存。假如某个特定内存区域大大超过了我的预期，我可以继续深入调查，而这个性能工具可以告诉我究竟是哪些函数分配了这些内存。如果我正在调查CPU的使用情况，我会从总体统计数据开始，诸如系统时间和用户时间各是多少，某个特定进程有多少个系统调用，之后我将能够更加深入到系统时间或者是用户时间，看看到底是哪些函数消耗了这些时间以及它们被调用的频率。一个高明的shell脚本，利用合适的已存在的工具来收集并整合信息，多花些时间也能实现其中的部分功能，但要完全实现这一愿望，就有必要让一些工具的行为发生根本性的变化。

13.2.2 漏洞2：没有可靠并完整的调用树

接下来的性能工具漏洞是目前还没有办法提供应用程序执行的完整的调用树。Linux有几个不完整的实现。oprofile提供了调用树的生成，但它是基于采样的，因此无法捕捉生成的每一个调用。gprof支持调用树，但它无法分析整个应用程序，除非是特定进程调用的每一个库编译时选择支持分析。最有前途的工具valgrind有一个界面称为calltree，详述参见5.2.5节，它的目标是提供一个完全准确的调用树。但是，它还在发展中，且无法适用于所有的二进制文件。

调用树工具是很有用的，即便是当它运行时会明显降低应用程序的性能。常见的使用方法是：先运行oprofile找出应用程序中的哪些函数是“热点”，然后运行调用树程序明确为什么应用程序要调用它们。oprofile那一步将提供全速运行时应用程序瓶颈的准确视图，而调用树，即使它运行缓慢，也会显示应用程序是如何调用那些函数以及调用的原因。唯

一的问题是，如果程序的行为是时间敏感的，运行缓慢将会导致其行为发生变化（比如那些依赖于网络或磁盘 I/O 的）。不过，现有的许多问题不是时间敏感的，用准确的调用树机制修复它们还有很长的路要走。

13.2.3 漏洞 3：I/O 的归因

最后，当前 Linux 中最大的漏洞是 I/O 归因。现在，Linux 没有提供好的方法来追踪哪些应用程序使用了最多的磁盘或者甚至是网络 I/O。

一个理想的工具应实时显示特定进程使用的磁盘或网络 I/O 的输入输出字节数。该工具将显示统计数据，如原始带宽，以及子系统能够占用的原始 I/O 百分比。此外，用户还要能分解统计数据，以便他们可以看到每一个单独网络和磁盘设备都相同的统计信息。

13.3 Linux的性能调优

即使存在上述漏洞，Linux 仍然是一个发现并修复性能问题的理想环境。它是由开发人员编写给开发人员的，其结果就是，它对性能调查者非常友好。Linux 有几个特点使它成为追踪性能问题的优秀平台。

13.3.1 可用的源代码

首先，开发人员可以访问到整个系统的大多数（如果不是全部）的源代码。如果问题看上去不在你的代码中，那么在追踪问题时这一点就相当有价值。对商用 UNIX 或其他无法获得源代码的操作系统，你可能需要等待供应商对问题进行调查，如果的确是他的问题，你也无法保证他会修复问题。但是，对 Linux 而言，你可以自己调查问题，弄清楚为什么性能问题会发生。如果问题不在你的应用程序中，你可以修复它并提交补丁，或是直接运行一个已修复的版本。如果在阅读 Linux 源代码后，你意识到是你的应用程序有问题，那么你也可以修复问题。不管是哪种情况，你都可以立即进行修复，不会因等待别人而卡壳。

13.3.2 容易联系开发者

Linux 的第二个优点是：找到并联系特定应用程序或库的开发人员相对容易。大多数其他专有操作系统很难说清楚哪个工程师负责哪个代码片段，相比之下，Linux 要更加开放。通常，特定软件片段开发人员的名字和联系方式都放在软件包中。访问开发人员使你可以咨询一些问题，包括：特定代码段的行为，运行缓慢的代码是干什么用的，以及给定的优化执行起来是否安全。开发人员一般会很乐意就此提供帮助。

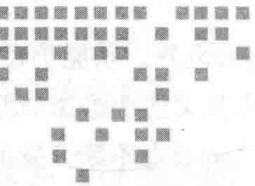
13.3.3 Linux 还年轻

Linux 是性能优化的优秀平台的最后一个理由是它还未成熟，功能仍在开发中，Linux 还有很多机会发现并修复简单的性能缺陷。大多数开发人员关注于添加功能，因此性能问题会被搁置一边不去解决。一个雄心勃勃的性能调查人员可以在不断开发的 Linux 中发现并修复很多小的性能问题。这些小修复超越了单个个体，惠及整个 Linux 团体。

13.4 本章小结

本章我们研究了 Linux 性能工具几个集中存在缺点的方面，并提出了一些理想的解决方案。我们还讨论了为什么 Linux 是一个尝试进行性能调查和优化的好平台。

改变 Linux 获得更好的性能，这取决于你——读者。提升 Linux 性能，改进 Linux 性能工具的机会比比皆是。如果你发现了困扰你的性能问题，修复它，或者向开发人员报告，和他们一起修复它。无论哪种方式，都不会再有人被这个问题卡住，整个 Linux 团体都会受益。



Appendix A

附录 A

性能工具的位置

本书描述的性能工具来源于 Internet 上许多不同的位置。幸运的是，大多数主要发行版都把它们放在一起，包含在了其发行版的当前版本中。表 A-1 描述了全部工具，提供了指向其原始源位置的地址，并注明它们是否包含在以下发行版中：Fedora Core 2 (FC2)、Red Hat Enterprise Linux (EL3) 和 SUSE 9.1 (S9.1)。

表 A-1 性能工具的位置

工 具	发 行 版 本	源 网 址
bash	FC2, EL3, S9.1	http://cnswww.cns.cwru.edu/~chet/bash/bashtop.html
etherape	无	http://etherape.sourceforge.net/
ethtool	FC2, EL3, S9.1	http://sourceforge.net/projects/gkernel/
free	FC2, EL3, S9.1	procps 包的一部分： http://procps.sourceforge.net/
gcc	FC2, EL3, S9.1	http://gcc.gnu.org/
gdb	FC2, EL3, S9.1	http://sources.redhat.com/gtb/
gkrellm	FC2, S9.1	http://web.wt.net/~billw/gkrellm/gkrellm.html
gnome-system-monitor	FC2, EL3, S9.1	GNOME 项目的一部分，获取地址： ftp://ftp.gnome.org/pub/gnome/sources/gnome.system-monitor/
gnumeric	FC2, EL3, S9.1	http://www.gnome.org/projects/gnumeric/
gprof	FC2, EL3, S9.1	binutils 包的一部分： http://sources.redhat.com/binutils
ifconfig	FC2, EL3, S9.1	net-tools 的一部分： http://www.tazenda.demon.co.uk/~phil/net-tools/
iostat	FC2, S9.1	sysstat 包的一部分： http://perso.wanadoo.fr/sebastien.godard/
ip	FC2, EL3, S9.1	iproute 包的一部分： ftp://ftp.inr.ac.ru/ip-routing

(续)

工 具	发 行 版 本	源 网 址
<code>ipcs</code>	FC2, EL3, S9.1	<code>util-linux</code> 包 的一 部 分: http://ftp.win.tue.nl:/pub/linux-local/utils/util-linux
<code>iptraf</code>	FC2, S9.1	http://cebu.mozcom.com/riker/iptraf
<code>kcacheGrind</code>	FC2, S9.1	<code>kdesdk</code> (v3.2 或更 高 版 本) 包 的一 部 分: http://kcacheGrind.sourceforge.net/cgi-bin/show.cgi
<code>ldd</code>	FC2, EL3, S9.1	<code>GNU libc</code> 的一 部 分: http://www.gnu.org/software/libc/libc.html
<code>ld</code> (Linux 加载器)	FC2, EL3, S9.1	<code>binutils</code> 的一部 分: http://sources.redhat.com/binutils
<code>lsof</code>	FC2, EL3, S9.1	ftp://lsof.itap.purdue.edu/pub/tools/unix/lsof
<code>ltrace</code>	FC2, EL3, S9.1	http://packages.debian.org/unstable/utils/ltrace.html
<code>memprof</code>	FC2, EL3, S9.1	http://www.gnome.org/projects/memprof
<code>mii-tool</code>	FC2, EL3, S9.1	<code>net-tools</code> 的一部 分: http://www.tazenda.demon.co.uk/phil/net-tools/
<code>mpstat</code>	FC2, S9.1	<code>sysstat</code> 包 的一 部 分: http://perso.wanadoo.fr/sebastien.godard/
<code>netstat</code>	FC2, EL3, S9.1	<code>net-tools</code> 的一部 分: http://www.tazenda.demon.co.uk/phil/net-tools/
<code>objdump</code>	FC2, EL3, S9.1	<code>binutils</code> 的一部 分: http://sources.redhat.com/binutils
<code>oprofile</code>	FC2, EL3, S9.1	http://oprofile.sourceforge.net/
<code>proc filesystem</code>	FC2, EL3, S9.1	<code>proc</code> 文件系统是 Linux 内核的组成部分, 包含在几乎所有的发行版中
<code>procinfo</code>	FC2, S9.1	ftp://ftp.cistron.nl/pub/people/svm
<code>ps</code>	FC2, EL3, S9.1	<code>procps</code> 包的一 部 分: http://procps.sourceforge.net/
<code>sar</code>	FC2, EL3, S9.1	<code>sysstat</code> 包的一 部 分: http://perso.wanadoo.fr/sebastien.goatd/
<code>script</code>	FC2, EL3, S9.1	<code>util-linux</code> 包的一 部 分: http://www.kernel.org/pub/linux/utils/util-linux/
<code>slabtop</code>	FC2, EL3, S9.1	<code>procps</code> 包的一 部 分: http://procps.sourceforge.net/
<code>strace</code>	FC2, EL3, S9.1	http://sourceforge.net/projects/strace/
<code>tee</code>	FC2, EL3, S9.1	<code>coreutils</code> 包的一 部 分: http://alpha.gnu.org/gnu/coreutils/
<code>time</code>	FC2, EL3	http://www.gnu.org/directory/GNU/time.html
<code>top</code>	FC2, EL3, S9.1	<code>procps</code> 包的一 部 分: http://procps.sourceforge.net/
<code>valgrind</code>	S9.1	http://valgrind.kde.org/
<code>vmstat</code>	FC2, EL3, S9.1	<code>procps</code> 包的一 部 分: http://procps.sourceforge.net/

虽然没有在表中出现, 但是 Debian (测试) 包含了上面列出的, 除 `procinfo` 之外的全部工具。

附录 B

安装 oprofile

尽管系统剖析器 oprofile 是一个强大的性能工具，但它的安装 / 使用有些难度。本附录描述了在 Fedora Core 2 (FC2)、Red Hat Enterprise Linux (EL3) 和 SUSE 9.1 (S9.1) 上安装 oprofile 的一些问题。

B.1 Fedora Core 2 (FC2)

对 FC2，应该使用 Red Hat 为 oprofile 提供的软件包，而不是从 oprofile 网站下载的那些。单处理器内核不提供必要的 oprofile 驱动程序。Red Hat 在内核的 smp 版本中打包了必要的 oprofile 内核模块。如果想要运行 oprofile，即使你是在单处理器机器上运行，也必须使用 smp 内核。

B.2 Enterprise Linux (EL3)

同样的，对 EL3，Red Hat 也为 oprofile 提供了使用的软件包，而不是从 oprofile 网站下载的那些。单处理器内核不提供必要的 oprofile 驱动程序。Red Hat 用内核的 smp 或 hugemem 版本打包了必要的 oprofile 内核模块。如果想要运行 oprofile，即使你是在单处理器机器上运行，也必须使用 smp 或 hugemem 内核。

在 EL3 中使用 oprofile 的更多详细信息请参见 <http://www.redhat.com/docs/manuals/>

enterprise/RHTL-3-Manual/sysadmin-guide/ch-oprofile.html。

B.3 SUSE 9.1

对 SUSE 9.1, SUSE 为 oprofile 提供了使用的软件包, 而不是从 oprofile 网站下载的那些。SUSE 内核的全部版本 (default、smp 和 bigsmp) 都提供对 oprofile 的支持, 因此提供的任何内核都有效。

在当前复杂的大规模系统场景下，卓越的应用程序性能比以往任何时候都更重要，但是获得这样的性能也更加困难。Linux由于其开源特性，已经具有一系列优秀的优化工具，只是这些工具散布在互联网上，工具的相关文档也很少，只有少数专家知道如何综合使用这些工具来解决实际问题。基于此，本书介绍了当前常用的Linux优化工具，展示了它们是如何行之有效地提升整体应用程序性能的。通过真实案例，向开发人员演示了怎样精确定位影响性能的源代码行，使系统管理人员和应用程序开发人员能够迅速深入系统瓶颈，更快地实施解决方案。

通过阅读本书，你能够：

- 在不熟悉底层系统的情况下，快速识别系统瓶颈。
- 针对具体问题，找到并选择正确的性能工具。
- 深入理解系统性能及优化问题。
- 掌握优化系统CPU、用户CPU、内存、网络I/O和磁盘I/O的方法，并了解它们之间的关系。
- 掌握修复计算密集型（CPU-bound）、延迟敏感和I/O密集型（I/O-bound）的应用程序的方法，跟随案例轻松地配置自己的环境。
- 安装并使用Linux系统的高级全系统分析器——oprofile。

无论读者的技术背景如何，性能优化的新手都能通过学习本书，掌握一系列清晰实用的优化原则和策略，并获得丰富的Linux知识，解决Linux系统和应用程序的优化问题，增加商业价值并提高用户满意度。



投稿热线：(010) 88379604

客服热线：(010) 88378991 88361066

购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com

网上购书：www.china-pub.com

数字阅读：www.hzmedia.com.cn



上架指导：计算机/Linux

ISBN 978-7-111-56017-3



9 787111 560173 >

定价：69.00元

[General Information]

书名=Linux性能优化

作者=Phillip G.Ezolt著

丛书名=LINUX/UXIN技术丛书

页数=223

SS号=14209635

出版日期=2017.05

出版社=北京：机械工业出版社

ISBN号=7-111-56017-3

中图法分类号=TP316.85

原书定价=60.60

主题词=LINUX操作系统

参考文献格式=Phillip G.Ezolt著.Linux性能优化[M].北京：机械工业出版社,2017
.05.

封面
书名
版权
前言
目录

第1章 性能追踪建议

1.1 常用建议

- 1.1.1 记大量的笔记（记录所有的事情）
- 1.1.2 自动执行重复任务
- 1.1.3 尽可能选择低开销工具
- 1.1.4 使用多个工具来搞清楚问题
- 1.1.5 相信你的工具
- 1.1.6 利用其他人的经验（慎重）

1.2 性能调查概要

- 1.2.1 找到指标、基线和目标
- 1.2.2 追踪近似问题
- 1.2.3 查看问题是否早已解决
- 1.2.4 项目开始（启动调查）
- 1.2.5 记录，记录，记录

1.3 本章小结

第2章 性能工具：系统CPU

2.1 CPU性能统计信息

- 2.1.1 运行队列统计
- 2.1.2 上下文切换
- 2.1.3 中断
- 2.1.4 CPU使用率

2.2 Linux性能工具：CPU

- 2.2.1 vmstat（虚拟内存统计）
- 2.2.2 top（2.0.x版本）
- 2.2.3 top（3.x.x版本）
- 2.2.4 procinfo（从/proc文件系统显示信息）
- 2.2.5 gnome-system-monitor
- 2.2.6 mpstat（多处理器统计）
- 2.2.7 sar（系统活动报告）
- 2.2.8 oprofile

2.3 本章小结

第3章 性能工具：系统内存

- 3.1 内存性能统计信息
 - 3.1.1 内存子系统和性能
 - 3.1.2 内存子系统(虚拟存储器)
- 3.2 Linux性能工具:CPU与内存
 - 3.2.1 vmstat()
 - 3.2.2 top(2.x和3.x)
 - 3.2.3 procinfo()
 - 3.2.4 gnome-system-monitor()
 - 3.2.5 free
 - 3.2.6 slabtop
 - 3.2.7 sar()
 - 3.2.8 /proc/meminfo

3.3 本章小结

第4章 性能工具:特定进程CPU

- 4.1 进程性能统计信息
 - 4.1.1 内核时间vs.用户时间
 - 4.1.2 库时间vs.应用程序时间
 - 4.1.3 细分应用程序时间
- 4.2 工具
 - 4.2.1 time
 - 4.2.2 strace
 - 4.2.3 ltrace
 - 4.2.4 ps(进程状态)
 - 4.2.5 ld.so(动态加载器)
 - 4.2.6 gprof
 - 4.2.7 oprofile()
 - 4.2.8 语言:静态(C和C++)vs.动态(Java和Mono)

4.3 本章小结

第5章 性能工具:特定进程内存

- 5.1 Linux内存子系统
- 5.2 内存性能工具
 - 5.2.1 ps()
 - 5.2.2 /proc/<PID>
 - 5.2.3 memprof
 - 5.2.4 valgrind(cachegrind)
 - 5.2.5 kcachegrind
 - 5.2.6 oprofile()

5.2.7 ipcs

5.2.8 动态语言 (Java和Mono)

5.3 本章小结

第6章 性能工具：磁盘I/O

6.1 磁盘I/O介绍

6.2 磁盘I/O性能工具

6.2.1 vmstat ()

6.2.2 iostat

6.2.3 sar ()

6.2.4 lsof (列出打开文件)

6.3 缺什么

6.4 本章小结

第7章 性能工具：网络

7.1 网络I/O介绍

7.1.1 链路层的网络流量

7.1.2 协议层网络流量

7.2 网络性能工具

7.2.1 mii-tool (媒体无关接口工具)

7.2.2 ethtool

7.2.3 ifconfig (接口配置)

7.2.4 ip

7.2.5 sar ()

7.2.6 gkrellm

7.2.7 iptraf

7.2.8 netstat

7.2.9 etherape

7.3 本章小结

第8章 实用工具：性能工具助手

8.1 性能工具助手

8.1.1 自动执行和记录命令

8.1.2 性能统计信息的绘图与分析

8.1.3 调查应用程序使用的库

8.1.4 创建和调试应用程序

8.2 工具

8.2.1 bash

8.2.2 tee

8.2.3 script

- 8.2.4 watch
- 8.2.5 gnumeric
- 8.2.6 ldd
- 8.2.7 objdump
- 8.2.8 GNU调试器 (gdb)
- 8.2.9 gcc (GNU编译器套件)

8.3 本章小结

第9章 使用性能工具发现问题

- 9.1 并非总是万灵药
- 9.2 开始追踪
- 9.3 优化应用程序
 - 9.3.1 内存使用有问题？
 - 9.3.2 启动时间有问题？
 - 9.3.3 加载器引入延迟了吗？
 - 9.3.4 CPU使用（或完成时长）有问题？
 - 9.3.5 应用程序的磁盘使用有问题？
 - 9.3.6 应用程序的网络使用有问题？
- 9.4 优化系统
 - 9.4.1 系统是受CPU限制的吗？
 - 9.4.2 单个进程是受CPU限制的吗？
 - 9.4.3 一个或多个进程使用了大多数的系统CPU吗？
 - 9.4.4 一个或多个进程使用了单个CPU的大多数时间？
 - 9.4.5 内核服务了许多中断吗？
 - 9.4.6 内核的时间花在哪儿了？
 - 9.4.7 交换空间的使用量在增加吗？
 - 9.4.8 系统是受I/O限制的吗？
 - 9.4.9 系统使用磁盘I/O吗？
 - 9.4.10 系统使用网络I/O吗？
- 9.5 优化进程CPU使用情况
 - 9.5.1 进程在用户还是内核空间花费了时间？
 - 9.5.2 进程有哪些系统调用，完成它们花了多少时间？
 - 9.5.3 进程在哪些函数上花了时间？
 - 9.5.4 热点函数的调用树是怎样的？
 - 9.5.5 Cache缺失与热点函数或源代码行是对应的吗？
- 9.6 优化内存使用情况
 - 9.6.1 内核的内存使用量在增加吗？
 - 9.6.2 内核使用的内存类型是什么？

- 9.6.3 特定进程的驻留集大小在增加吗？
 - 9.6.4 共享内存的使用量增加了吗？
 - 9.6.5 哪些进程使用了共享内存？
 - 9.6.6 进程使用的内存类型是什么？
 - 9.6.7 哪些函数正在使用全部的栈？
 - 9.6.8 哪些函数的文本大小最大？
 - 9.6.9 进程使用的库有多大？
 - 9.6.10 哪些函数分配堆内存？
- 9.7 优化磁盘I/O使用情况
- 9.7.1 系统强调特定磁盘吗？
 - 9.7.2 哪个应用程序访问了磁盘？
 - 9.7.3 应用程序访问了哪些文件？
- 9.8 优化网络I/O使用情况
- 9.8.1 网络设备发送 / 接收量接近理论极限了吗？
 - 9.8.2 网络设备产生了大量错误吗？
 - 9.8.3 设备上流量的类型是什么？
 - 9.8.4 特定进程要为流量负责吗？
 - 9.8.5 流量是哪个远程系统发送的？
 - 9.8.6 哪个应用程序套接字要为流量负责？
- 9.9 尾声
- 9.10 本章小结
- 第10章 性能追踪1：受CPU限制的应用程序（GIMP）
- 10.1 受CPU限制的应用程序
 - 10.2 确定问题
 - 10.3 找到基线 / 设置目标
 - 10.4 为性能追踪配置应用程序
 - 10.5 安装和配置性能工具
 - 10.6 运行应用程序和性能工具
 - 10.7 分析结果
 - 10.8 转战网络
 - 10.9 增加图像缓存
 - 10.10 遇到（分片引发的）制约
 - 10.11 解决问题
 - 10.12 验证正确性
 - 10.13 后续步骤
 - 10.14 本章小结
- 第11章 性能追踪2：延迟敏感的应用程序（nautilus）

- 11.1 延迟敏感的应用程序
- 11.2 确定问题
- 11.3 找到基线 / 设置目标
- 11.4 为性能追踪配置应用程序
- 11.5 安装和配置性能工具
- 11.6 运行应用程序和性能工具
- 11.7 编译和检查源代码
- 11.8 使用gdb生成调用跟踪
- 11.9 找到时间差异
- 11.10 尝试一种可能的解决方案
- 11.11 本章小结

第12章 性能追踪3：系统级迟缓 (prelink)

- 12.1 调查系统级迟缓
- 12.2 确定问题
- 12.3 找到基线 / 设置目标
- 12.4 为性能追踪配置应用程序
- 12.5 安装和配置性能工具
- 12.6 运行应用程序和性能工具
- 12.7 模拟解决方案
- 12.8 报告问题
- 12.9 测试解决方案
- 12.10 本章小结

第13章 性能工具：下一步是什么

- 13.1 Linux工具的现状
- 13.2 Linux还需要什么样的工具
 - 13.2.1 漏洞1：性能统计信息分散
 - 13.2.2 漏洞2：没有可靠并完整的调用树
 - 13.2.3 漏洞3：I/O的归因
- 13.3 Linux的性能调优
 - 13.3.1 可用的源代码
 - 13.3.2 容易联系开发者
 - 13.3.3 Linux还年轻
- 13.4 本章小结

附录A 性能工具的位置

附录B 安装oprofile

封底