

性能之巅

洞悉系统、企业与云计算

Systems Performance: Enterprise and the Cloud

[美] Brendan Gregg 著

徐章宁 吴寒思 陈 磊 译



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系QQ1779903665.

PDF代找说明：

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签索引和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我QQ1779903665。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

声明：本人只提供代找服务，每本100%索引书签和目录，因寻找pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，与本人无关，我们仅仅只是帮助你寻找到你要的pdf而已。

系统性能优化的指导大全

大型企业服务、云计算和虚拟计算系统都面临着严重的性能挑战。如今，国际知名的性能专家Brendan Gregg将业界验证的方法、工具和指标融汇在一起，足以应对最为复杂环境的分析和调优工作。

本书着力讲述Linux和UNIX的性能，但所论述的性能问题适用于所有的操作系统。你将洞察到系统是如何工作与执行的，学习到如何分析和改进系统及应用程序性能的方法。

Gregg书中的示例都通过裸机和云端虚拟机做演示，所运行的系统包括基于Linux的Ubuntu、Fedora、CentOS以及基于illumos的Joyent SmartOS、OmniTI OmniOS。无论是CPU、内存、磁盘与网络的“传统”分析，还是像云计算和动态跟踪这类新领域，本书系统地覆盖了现代系统性能的方方面面。

本书还帮助你识别复杂性能中“未知的未知”：这是在你不知道的地方出现的瓶颈。本书还收纳了一个翔实的研究实例，向你展示一个真实云计算问题是如何从头到尾做分析的。

内容涵盖

- 现代性能分析与调优：术语、概念、模型、方法和技术
- 动态跟踪技术与工具，收录 DTrace、SystemTap 和 Perf 示例
- 内核内幕：揭示 OS 在做什么
- 如何使用系统观测工具、接口和框架
- 理解和监控应用程序性能
- 优化 CPU：处理器、核、硬件线程、缓存、互联与内核调度
- 内存优化：虚拟内存、换页、交换、内存架构、总线、地址空间与分配器
- 文件系统 I/O，包括缓存
- 存储设备/控制器、磁盘 I/O 工作负载、RAID，以及内核 I/O
- 网络相关性能问题：协议、套接字、网卡和物理连接
- OS和基于硬件虚拟化的性能实现，以及云计算所遇到的新问题
- 基准测试：如何得到精确的结果并避免一般性的错误

本书是企业和云计算环境运维人员的必备指导：系统管理员、网络管理员、数据库管理员和Web管理员，开发工程师，以及其他专业人员。对于新接触性能优化的学生等人员，本书还提供了饱含Gregg丰富教学经验的练习题目。

作者简介

Brendan Gregg 是 Joyent 公司的首席性能工程师，通过软件栈分析性能和扩展。在 Sun Microsystems 公司（之后为 Oracle）担任首席性能和内核工程师期间，他的工作包括开发 ZFS L2ARC，这是一个利用闪存存储器提升性能的文件系统。他还开发了许许多多的性能工具，部分工具收录在 Mac OS X 和 Oracle Solaris 11 的发行版中。他最近从事的工作覆盖针对 Linux 和 illumos 内核分析的性能可视化。他还是 DTrace（Prentice Hall，2011）和 Solaris Performance and Tools（Prentice Hall，2007）的两书合著者。

PEARSON

www.pearson.com

PEARSON



扫码观看
本书视频



T SP 000



博文视点Broadview



@博文视点Broadview



策划编辑：张春雨
责任编辑：贾莉
封面设计：吴海燕

上架建议：性能优化/系统运维

ISBN 978-7-121-26792-5



9 787121 267925 >

定价：128.00元

性能之巅

洞悉系统、企业与云计算

Systems Performance: Enterprise and the Cloud

【美】Brendan Gregg 著

徐章宁 吴寒思 陈磊 译

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书基于 Linux 和 Solaris 系统阐述了适用于所有系统的性能理论和方法, Brendan Gregg 将业界普遍承认的性能方法、工具和指标收集于本书之中。阅读本书, 你能洞悉系统运作的方式, 学习到分析和提高系统与应用程序性能的方法, 这些性能方法同样适用于大型企业及云计算这类最为复杂的环境的性能分析与调优。

Authorized translation from the English language edition, entitled Systems Performance: Enterprise and the Cloud, 9780133390094 by Brendan Gregg, published by Pearson Education, Inc., publishing as Prentice Hall, Copyright©2014 Pearson Education, Inc.

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 PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright©2015.

本书简体中文版专有出版权由 Pearson Education 培生教育出版亚洲有限公司授予电子工业出版社。未经出版者预先书面许可, 不得以任何方式复制或抄袭本书的任何部分。

本书简体中文版贴有 Pearson Education 培生教育出版集团激光防伪标签, 无标签者不得销售。

版权贸易合同登记号 图字: 01-2014-4017

图书在版编目 (CIP) 数据

性能之巅: 洞悉系统、企业与云计算 / (美) 格雷格 (Gregg, B.) 著; 徐章宁, 吴寒思, 陈磊译. —北京: 电子工业出版社, 2015.8

书名原文: Systems Performance: Enterprise and the Cloud

ISBN 978-7-121-26792-5

I. ①性… II. ①格… ②徐… ③吴… ④陈… III. ①计算机网络 IV. ①TP393

中国版本图书馆 CIP 数据核字 (2015) 第 172687 号

策划编辑: 张春雨

责任编辑: 贾 莉

封面设计: 吴海燕

印 刷: 三河市鑫金马印装有限公司

装 订: 三河市鑫金马印装有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787×980 1/16

印张: 40.25 字数: 895 千字

版 次: 2015 年 8 月第 1 版

印 次: 2015 年 8 月第 1 次印刷

定 价: 128.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。



推荐序 1

大数据、云计算和人工智能都是热门概念，也是现实问题。很多团队都面对类似的技术抉择：如何用开源软件构架机群、如何选择云服务、如何设计高效的分布式 Web 服务，或者如何开发高效的分布式机器学习系统？当面对这些问题的时候，最好能从一些重要的可度量的方面给出定量分析和比较。可是哪些方面重要，以及如何度量呢？

古往今来有很多重要的系统度量问题，比如“如何度量网络繁忙程度”、“如何得知某台机器还活着吗”、“服务中断是因为某个进程死锁了，还是机器出问题了”，或是“我们的机群中用 SSD 替换硬盘的比例应该多大”，等等。

这些问题的答案都不简单——正确答案往往构建在对操作系统的深刻理解上，甚至构建在统计学和统计实验基础上。可是通常我们是在繁重的业务压力下应对这些问题的，所以只能尽快找一个“近似”测量办法。这终非长久之计——在我们经历了很多问题之后，可能会发现自己从未深刻理解问题，没有深入思考，没有沉淀经验，没有获得成长。

如果有意深入理解问题，借助前人经验是可以事半功倍的。只是关于操作系统的教科书却不能为我们提供足够的基础知识。操作系统是如此复杂，几乎涉及计算机科学的所有方面。学校教育往往只能基于极简化的示范系统，比如 Minix。但实际使用的操作系统，比如 Linux 和 Solaris，有更多需要学习和理解的地方。

我做分布式机器学习系统有八年了，其间很多时候要面对系统分析的问题。但是坦诚地说，大部分情况下我都只能尽快地找一个“近似”方法，处在没有时间深入琢磨上述系统问题的窘境。看到《性能之巅：洞悉系统、企业与云计算》一书之后，不禁眼前一亮。这本书从绪论之

后，就开始介绍“方法”——概念、模型、观测和实验手段。作者不仅利用操作系统自带的观测工具，还自己开发了一套深入分析观测结果的脚本，这就是有名的 DTrace Toolkit（大家可以直接找来使用）。《性能之巅：洞悉系统、企业与云计算》一书介绍的实验和观测方法，包括内存、CPU、文件系统、存储硬件、网络等各个方面。而且，在介绍方法之前会深入介绍系统原理——我没法期望更多了！

很高兴看到这一经典名著的中文版问世。因为被邀作序，有幸近水楼台先得月，深受教益。感谢译者和编辑的努力。相信读者朋友们定能从中受益。

——王益 LinkedIn 高级主任分析师



推荐序 2

收到侠少的邀约来写序，多少有点忐忑和紧张。忐忑是因为平时太多时间总是在解决各类问题，没有时间好好沉淀下来读一读书，总结一下发现问题和解决问题的方法；紧张是因为我好像很久没有写字了，上一次写长篇大论还是 N 年前，怀疑自己是不是能够在有限的文字内把系统性能的问题解释清楚。于是，就在这样的背景下，我从出版方那里拿到了书稿，迫不及待地读了起来。

书的作者 Gregg 先生是业内性能优化方面大名鼎鼎的人物，早年在 Sun 公司的时候是性能主管和内核工程师，也是大名鼎鼎的 DTrace 的开发人员，要知道 DTrace 可是众多 trace 类工具中最著名的，并且先后被移植到了很多别的 OS 上。书的全篇都在讨论性能优化，我想了想，我每天从事的工作不就是这个吗？SAE 上成千上万的开发者们，他们每天问的问题几乎全部和性能相关：为什么我的 App 打开比较慢，为什么我的网站访问不了，怎么才能看我的业务哪个逻辑比较慢？对于这些问题，其实难点并不在于解决它，最难的在于发现并定位它，因为只要一旦定位了故障点或者性能瓶颈点，解决起来就并不是很复杂的事情。

对于性能优化，最大的挑战就是性能分析，而性能分析要求我们对于操作系统、网络的性能要了如指掌，明晰各个部分的执行时间数量级，做出合理的判断，这部分在书中有很详细的讨论，让读者可以明确地将这些性能指标应用在 80:20 法则上。

工欲善其事，必先利其器，了解系统性能指标后，就需要找到合适的工具对可能存在的瓶颈进行分析，这要求我们具备全面的知识，涉及 CPU 性能、内存性能、磁盘性能、文件系统性能、网络协议栈性能等方方面面，好在本书详细介绍了诸如 DTrace、vmstat、mpstat、sar、

SystemTap 等工具利器，如何将这些工具组合，并应用在适合的场景，是一门学问，相信读者会在书中找到答案。顺便说一句，Dtrace+SystemTap 帮助 SAE 解决过非常多的性能疑难杂症，一定会对读者的业务分析产生帮助！

单个进程的性能分析是简单的，因为我们可以定位到 system-call 或者 library-call 级别，然后对照代码很快解决，但整个业务的性能分析是复杂的，这里面涉及多个业务单元、庞大的系统组件。最麻烦的是，往往造成性能问题的还不是单元本身，而是单元和单元相连接的网络服务。这就要求我们必须要有科学的分析方法论，来帮助我们找到整个系统业务中的瓶颈所在。书中就此介绍了包括随机变动、诊断循环等多个方法，并且介绍了涉及分析的数学建模和概念。不要忽视数学在性能分析中的作用，在实际应用中，我就利用方差和平均值的变化规律科学地分辨一套系统到底是否应该扩容。

找到了性能瓶颈，下一步就是解决问题。当然解决问题的最好办法就是改代码，但是，在你无法短时间内修改代码的时候，对系统进行优化也可以实现这一目的。这就要求我们对于系统的各个环节都要明白其工作原理和联系。本书第 3 章详细讨论了操作系统，这对于读者是很有用的。因为很多时候我们在不改代码的情况下优化系统，就是优化内存分配比例，就是优化 CPU 亲密度，就是尝试各种调度算法，就是做操作系统层面的各种网络参数调优。

对于上述所有问题的认识，我相信读者在通读全书后会有不一样的感觉。记住，不要只读一遍，每读一遍都必有不同的体会。不多说了，我要赶紧再去读一遍：)

——丛磊 新浪 SAE 创始人/总负责人



推荐序 3

人类正在用软件重构这个世界。从上世纪四十年代电子计算机出现，到个人电脑风靡、互联网大行其道，再到如今正兴旺的云计算加移动互联网，还有起步中的物联网，所有这些表面上看是计算机硬件变得无处不在，而实质上是软件一步步掌管起了这个世界。噫吁嚱，短短七十几年，软件轻松征服世界！渗透渗透再渗透，已经进入所有人的生活。

不言而喻，软件对这个世界和人类的重要性也越来越大。我很负责任地说，软件的健康与否关系着世界的安危。君不见，几多时，一个软件漏洞便让全球惊慌。不经意间，我们与软件的关系变得休戚与共。

不幸的是，软件的总体健康状况并不乐观，问题很多。

瑕疵（Bug）是软件行业的一个永恒话题，是破坏软件质量的头号大敌。但迄今为止，完全发现和彻底消除瑕疵还只是奢望，是不可能实现的目标。换句话说，掌管着我们生活的所有软件都是在带着瑕疵运行，真正的带“病”工作。因为每个软件内部都有纵横交错的无数条道路，CPU 经常奔跑的那些路径上的瑕疵早被发现和移除，所以软件大多时候并不发“病”。但也有时候，CPU 会遭遇软件中的瑕疵，发生意外。目前，我们能做的只有努力多发现瑕疵，并尽可能找到根源，将其去除。但这并不是件容易的事情。

除了瑕疵，性能问题是威胁软件健康的另一个大敌。简单来说，我们把软件中的错误归入瑕疵一类。把那些在速度、资源消耗、工作量等方面的不满意表现纳入性能问题一类。用员工考核做比喻，瑕疵是把一件事做错了，而性能问题是虽然做对了，但是做得不够好，可能开销太大，用的资源太多，可能完成速度太慢，用的时间太久，可能完成的工作量太少，活干的不够多，总之是结果不令人满意，还有必要改进。

举例来说，某支付软件在性能方面就存在问题。一旦运行，会频繁触发大量的缺页异常，消耗的 CPU 时间过多，导致不必要的能源浪费。随着软件变得无处不在，软件的耗电问题已经引起越来越多的关注。在笔者写作这几行文字时，该软件的几个进程仍在一刻不停地触发着缺页异常，过去两天的累积数量已经超过千万，消耗 CPU 的净时间已经超过一小时。粗略估计，这几个进程至少已经消耗了 0.01 度电。不要忽视 0.01 这个看似微小的数字，把它乘以全国的总用户数，立刻就变成一个庞大的数字了。

与软件瑕疵类似，性能问题也可能危害巨大！更可怕的是，性能方面的问题容易触发隐藏在软件深处的瑕疵，直接导致软件崩溃或者其他无法预计的故障。

发现瑕疵根源的过程一般称为调试（Debug）。纠正性能问题，提高软件性能的努力被称为调优（Tune）。不论调试，还是调优，都不是简单的事。对软件工程师的技术要求很高。一些复杂的问题，常常需要多方面的知识，需要对系统有全面了解，既有大局观，能俯瞰全局，又能探微索隐，深入到关键的细节，可谓是“致广大而尽精微”。

如果一定要把调试和调优的难度比一下，调优的难度更大。简单的解释是，调试的主要目标是寻找瑕疵，瑕疵固定存在软件中的某一个点。因此，调试时可以通过断点等技术把软件静止下来，慢慢分析。而调优必须关注一个动态的过程，观察一段时间内的软件行为。这样一来，调优时常常不可以把软件中断下来静止分析，而需要以统计学的方法或者其他技术对软件做长时间监视。

记得两年前，曾经有一个同行以饱经沧桑的神情问我：“在中国这样的软件环境里，做技术的工程师应该如何发展呢？难道都得像你那样写一本书么？”坦率地说，我当时没能给出让这位同行很满意的回答。因为这个问题确实不太容易回答。此事之后，我常常想起这个困扰着很多同行的问题。多次思考后，我似乎有了个比较好的答案。首先要确认自己是喜欢软件技术的，愿意在技术方向上持续发展。接下来的问题是如何在技术方向上不断前进。“日日新，又日新”。我的建议是，逐级攀登软件技术的三级台阶：编码、调试和调优。

代码是软件的根本，一个好的软件工程师必须过代码这一关，写代码时如行云流水，读代码时穿梭自如，如履平川。以调试器为核心的调试技术是对抗软件瑕疵的最有力工具，是每个技术同行都应该佩戴的一把利剑。调优技术旨在发现软件的性能障碍，让软件跑得更好。随着对软件性能问题的重视，调优技术的发展也越来越快，新的工具层出不穷。调优方面的工作和创业机会也在不断增加。几年前，我写作《软件调试》时，很多人还不太重视调试，但最近几年，软件调试已经逐渐从藏在背后的隐学逐步走向前台，成为一门显学。可以预见，性能调优也会受到越来越多的重视。大家加油！

学习调优技术有很多挑战，很高兴看到有这样一本关于系统优化的好书引进到国内。好友侠少诚邀作序，盛情难却，仓促命笔，词不达意，请诸君海涵，是为序。



推荐序 4

性能调优是每一个系统工程师最重要的技能，也是衡量其水平高低的不二法门。Linux 是开源的操作系统，这也意味着本身可调整范围比较大。近十几年来，硬件设备日益复杂，互联网应用场景及 Web 2.0 蓬勃发展的同时，也带来了各种高并发的业务应用，有些复杂的分布式数据分析系统，单集群的物理服务器数量甚至超过 1 万台。这使得对系统运行环境的一点点优化，带来的收益都可能非常可观。

性能调优这个事情，大家往往都有话想说，技术专家们也都有些秘而不宣、甚至奉为压箱底儿的“活儿”。但这些“活儿”，往往来自 Case By Case 所获得的经验。例如解决了某大型电商网站的 Nginx 服务器问题、某 MySQL 数据库集群性能问题等。这些特定的大型案例，促使参与其中的技术人员，在某个或某些系统性能优化方面，具有较高水平、甚至一定造诣。

但即使这些技术专家，也难以解决所有性能问题。这有两方面原因，一方面自身缺乏对整个系统运行环境的全局把控能力。技术专家们能力的获得，是基于某些问题点扩散开来的，并非基于事先构建好的系统优化的全局观（这也和国内环境有关，大家往往大学毕业后就直接开始从事相关工作，缺少底层、结构性的学习，即使参与了某些培训课程）。

这种系统优化的全局观之所以难以形成，一个原因在于“未知的未知”，也就是说我们不知道自己不知道。比如，我们可能不知道设备中断其实会消耗大量 CPU 资源，因而忽略了解决问题的关键线索。再比如，作为初中级 DBA 并不知道应用程序连接 Oracle 时，每一个数据库连接（Session）实际上都非常消耗物理内存，成百上千个数据库连接长期驻留（看上去状态还是非活动的），PGA 被消耗殆尽，引起各种异常，成为性能问题的罪魁祸首。

另一方面在于性能问题的根源太过复杂。诚如作者所说，一来性能是主观的，连是否有性能问题，都因人而异。例如，磁盘平均读写相应时间为 1ms，这是好还是不好？是否需要调优？实际上取决于开发人员和最终用户（有时还包括领导）的性能预期。二来系统是复杂的。例如，本来 CPU/磁盘/内存各司其责，有了内存缓存（SWAP）机制，内存不够时可以使用部分硬盘空间来顶替。这看上去很好，但对于数据库系统而言，SWAP 是否启用，本身就是一个问题。再比如，对于云计算而言，多虚拟机共享物理机，这进一步增加了问题的复杂度。资源隔离是个技术活，现有技术很难做到磁盘 I/O 完全隔离。另外，最近非常流行的容器技术，如 Docker 等，让问题变得尤为复杂。容器即进程，不像 KVM 等虚拟机（KVM 至少还进行资源隔离）自带操作系统。容器并不是为 IaaS 而生，仅靠 cgroup 等隔离技术能做的非常有限。三是有可能有多个问题并存。有时终端用户抱怨系统慢，很可能不仅是由单个原因引起，例如，业务负载猛增，内网 1000Mbps 其实已经不够，但没引起注意；或是整体对外交付能力貌似还正常，但数据库磁盘 I/O 非常繁忙；还可能正偷偷地进行大量 SWAP 交换。

这两方面原因，使得大部分技术专家，即使对系统优化的某些领域确有独到见解，但说到能否解决所有系统性能问题，其实会有些底气不足。但本书作者看起来是个例外。纵观全书，作者建立了系统性能优化的体系框架，并且骨肉丰满，很明显，他不仅擅长某方面的性能优化，而且是全方位的专家，加之作为 DTrace（一种可动态检测进程等状态的工具）主要开发者，使得本书的说服力和含金量大增。

本书首先提及性能优化的方法论和常见性能检测工具的使用，具体内容更是涉及可能影响 Linux 系统性能的方方面面，从操作系统、应用程序、CPU、内存、文件系统、磁盘到网络，无所不包。在以上这些话题的探讨中，作者的表述方法值得称道——每个章节都程式化地介绍术语、模型、概念、架构、方法、分析工具和调优建议，这对于由于长期工作形成一定强迫症的某些技术人员，如我自己，阅读起来赏心悦目，也从侧面体现了作者的深厚功底和驾驭能力。

本书提供的性能优化方法论也令人印象深刻，包括几种常见的错误思考，如街灯法、随意猜测法和责怪他人法。街灯法来自一个著名的醉汉的故事——醉汉丢了东西，就只会在灯光最亮的地方着手。这种头疼医头脚痛医脚、错把结果当原因的事情，相信很多人都过类似经验。大型业务系统上线，大家都围着 DBA 责问数据库为什么崩溃了。但数据库出问题是结果，数据库本身一定是问题根源么？是否更应该从业务负载、程序代码性能、网络等方面联合排查？在列举各种不正确的方法后，作者建议采用科学法，科学法的套路是：描述问题→假设→预测→试验→分析。这种办法的好处是，可以逐一排除问题，也可以降低对技术专家个人能力和主观判断的依赖。

本书用单独的章节系统性介绍了操作系统、性能检测方法和各种基准测试，特别是作者主导开发的 DTrace（本书例子中用 DTrace 监控 SSH 客户端当前执行的每个命令并实时输出）。

这使得本书作为工具书的价值更得以彰显。云计算的出现，对系统优化带来新的挑战。作者作为某云计算提供商的首席性能工程师，带来一个真实的云客户案例分析，包括如何利用本书提及的技术、方法和工具，一步一步分析和解决问题。

很多时候，受限于语言障碍，系统工程师往往通过国内 BBS、论坛等获得知识，只是在性能问题确实棘手时，被迫找些英文资料，寻找技术解决思路。

博文视点推出的本书中文版，对于国内广大运维同仁而言，实属幸事。这让我们有机会系统学习和掌握性能优化的各个方面，有机会建立一种高屋建瓴的全局观。这样，在面对复杂系统问题时，不至于手足无措，或只能盲人摸象般试探。另外，虽然面对日益复杂的硬件设备和高并发的业务应用，问题不是变得简化而是更为复杂，但 Linux 系统演化至今，其最基础的体系架构和关键组件并未发生多大改变，这使得这本好书即使历经多年，价值毫无衰减，反而历久弥新。

总的来说一句话：如果早些接触到本书，该有多好！

——萧田国 触控科技运维总监 高效运维社区创始人



推荐序 5

性能的话题，从一开始就是复杂的。性能是一种典型的非功能需求，然而又贯穿在任何一种功能需求中，直接影响系统运行效率和用户体验。也正是由于这一特性，性能无法简单地通过单一的、直线式的思维来度量和管理的，而注定需要以系统工程的方法来掌握和调整。绝大多数的图书在谈到性能问题的时候，都是仅从片面的若干现象出发来触及问题的冰山一角，抑或干脆语焉不详甚至避而不谈。这也难怪，因为这个话题一旦展开，就会占用极大篇幅，相对于原先的论题而言就显得喧宾夺主。然而更重要的原因，也在于对性能问题有着全面认识，并且能够给出一个系统化的分析和全栈式的论述的作者实在不多。相关的要求近乎苛刻：既要对系统的每一个部件都了如指掌，又要深入理解部件之间的协作方式；既要精通系统运行的细节，又要明白取舍逻辑的大局观；既要懂得现象背后的原理，又要把握从开发部署的工作人员直至终端应用用户的需求乃至心态。

《性能之巅：洞悉系统、企业与云计算》以一种奇妙而到位的方式，把高屋建瓴的视角和脚踏实地的实践结合了起来，对性能这一复杂、微妙甚至有些神秘的话题进行了外科手术式的解析，读来真是让人感觉豁然开朗。

全书以罕见的遍历式结构，对软件系统的每一个部件都如庖丁解牛般加以剖析，几乎涉及业务的每一个细节。然而，这些细节并非简单的罗列，而是每一段论述都与具体的角色和场景紧密结合，取舍之间极见智慧。方法论更是不单说理，而是通过一个又一个的具体实例，逐步地建构起来，并反复运用于各个部件之上，使读者明白原理普适性的同时也知道怎样举一反三。

本书也是难得的 UNIX/Linux 系统管理员和运维工程师的百科全书式参考手册，相对于工

作于 Windows 上的同行而言，他们获得的知识更加零碎，甚至很多场合下不得不求助于网络上的只言片语，并只能通过耗时的、高风险的生产环境实验来取得一手经验数据。本书当然提供了不少趁手的软件工具供人使用，然而其更大的价值在于心法的传授，即怎样利用工程师现在就熟悉、现在就可用的工具来迅速地进行性能建模，完成故障排除和调优的关键步骤。书中的内容非常新，作者见过大世面，是从最与时俱进的大型云计算系统为出发点来落笔的，对付日常的性能问题完全没有压力，即使最新的硬件也能找到对应的解决方案。

本书的译者团队阵容强大，皆是在底层系统有多年一线工作经验的运维工程师和开发工程师。徐章宁同志几乎是以一己之力支撑起 PB 级数据运维的明星 DevOps，而另外两位也都是手工实现过复杂生产环境中文件系统和网络协议的大牛。可以说，他们对于性能的认识是经过多年实际工作的考验的，是深刻而且务实的，这为本书翻译在专业性方面提供了坚实的保证。加上他们多年养成的认真严谨的工作习惯，和深厚的中文功底，更是为该译本的可读性锦上添花。

希望所有的 IT 从业者都能从本书受益，让天下的系统都能达到性能之巅！

——高博 青年计算机学会论坛（YOCSEF）会员，文津奖得主，《研究之美》译者



推荐序 6

性能问题一直是个热门话题,在单机时代我们就已经投入了不计其数的人力物力进行研究。但是随着互联网行业的发展,分布式系统开始大量投入应用,对于性能问题的分析、调优提出了很多前所未有的新挑战。特别是如何做到单机性能与集群整体性能的平衡,以及在各种影响性能的要素之间进行取舍,成为摆在开发运维人员面前的巨大难题。

本书采用了自下而上的结构,从底层的操作系统、CPU、磁盘等基础元素开始,到工作原理层面分析性能受到的各种不同影响,以及如何评估、衡量各项性能指标,让读者知其所以然,在面对实际情况时能够更有针对性地做出判断和决定,而不是机械地、教条地行事。本书还提供了案例,手把手地展示了实际性能问题的排查调优过程。读者可以根据案例,结合业务系统实际情况展开工作。此外,本书还对常用的性能分析工具的使用和扩展做了详细介绍,对于日常工作效率的提升也有着很大的帮助。

译者徐章宁曾与我在 EMC 的云存储部门共事多年,在系统性能排查、调优方面有着丰富的经验,很高兴他能参与此书的翻译。审稿过程中我感觉译者不仅忠实地还原了原著的精华,也融合了自己多年工作中积累的经验教训。

我坚信,本书无论对于开发还是运维人员,无论对于设计、编码或者排查调优工作,都能发挥重要的参考作用,尤其适合常备案头。在此诚挚向大家推荐。

——林应 淘宝技术部高级技术专家



译者序

作为一名运维工程师，系统出现一些“诡异”问题的情况并不罕见。有些时候面对束手无策的问题着实让人头痛，这时我总会感慨，学生时代课本上计算机科学那些诸多的概念和理论所呈现出的完美感觉更多是在书本上，在真实系统中往往更多是另外一幅更为“现实”的景象。工作多年后，我自发形成了一个简单的认知：当系统庞大到一定程度时，其复杂性会变得不容易控制是一件很正常的事情。用技术手段把这些“失控”的点妥善摆平就是工程师价值的体现。在翻译本书的过程中，我对这个问题又有了不同的认识：

“已知的已知，已知的未知，未知的未知。”

这是本书多次提到过的概念，说的是有些事情我们知道自己知道，有些事情我们知道自己不知道，还有些事情我们不知道自己不知道。这个概念就系统（特别是复杂系统，诸如云计算或大数据）而言，特别贴切：就系统内部来说，无论用到的是某一操作系统，还是某一编程语言，其实本身就已经是复杂度较高的实体，要透彻掌握并非易事，何况系统皆由这些技术组合构建而成，方方面面无所不知是不可能做到的事情，这里的未知源于技术本身的复杂；就系统外部来说，如今时事变化一日千里，现在系统要处理的外界变化，可能最初的系统设计者都从未想过，这里的未知来源于未来的不定。所以，有让你手足无措的问题出现其实是一种很正常的状态，对此的恐惧只是人施加给自己的情感层面的东西。与此相反，始终对未知心生敬畏才是对待未知正常的态度，更是本应有的觉悟。

这里并不是说我们要对未知“投降”，而是说对“未知”有正确的认识才是我们取得进步的前提。其实我们多数人对“系统的未知”存在误区，我们常常将系统等同于具象的技术实体，

例如某种编程语言、OS 内核、网络。总觉得系统出问题肯定是我某些技术知识有漏洞没学好。可惜“学海无涯，而吾生有涯，以有涯随无涯，殆矣”，拘泥于各种眼花缭乱的技术只会让自己迷失造成时间的浪费。技术都是末节，真正要把握的主体其实只是系统本身。道理虽简单，但舍本逐末的事情却还是屡见不鲜。

要做好这一点，首要的是要有全局的系统观。更准确地说，是要有对系统各层知识的理解 and 实践能力，要有对系统架构的认知和理解的能力。只有对系统了如指掌，才有希望将已知的未知转化为已知的已知，将未知的未知转化为已知的未知，进而增加对系统的掌控能力，避免盲人摸象的悲剧。事实的真相是也本应是这样：技术的价值依附于系统及其价值，没有孤立存在的技术，一切价值的体现都在于系统本身。唯有立足系统本身，工程师才能打通性能这条经脉，领略到性能之巅的风采！

回观《性能之巅》这本书，全书的安排也深有此意。本书第 1~5 章可谓是精华部分，讲述了与系统性能相关的通用模型和通用方法；第 6~12 章才落实到具体的知识细节，讲述各性能组件（CPU、内存等）的知识（第 6~10 章）、云计算的基础（第 11 章），基准测试的方法（第 12 章）。本书最后一章是一个性能分析的实例，若是让长期与系统打交道的工程师读来，必然感同身受；若是性能新手阅读，则可以对性能工作的日常状况有个基本的了解。

本书是一座桥梁，作者 Brendan 是在系统性能领域耕耘多年的技术专家，在 Sun 和 Oracle 公司有过卓越的贡献，动态跟踪工具 DTrace 就是他主力开发的，他用自己多年的经验和实践归纳并总结出了系统性能的理论和方法，这些理论和方法的作用就像桥梁，把业界可用的工具（或是你自己开发的工具）与系统内部的原理机制联通让它们有机地结合起来，让与性能相关的工作（无论是性能分析还是性能调优）做到有的放矢、有章可循！这与单纯提供知识的技术书籍截然不同，“授人以鱼不如授人以渔”，其立意确实难能可贵。

现代 IT 技术的源头并非中国，但 IT 技术在这片土地上生根发芽，欣欣向荣。如今国人日常生活中所依赖的系统服务已经比比皆是，不信者打开自己的手机数数所装的 App 自然清楚，这些 App 背后多半都有远在某个数据中心的一个或多个系统作为支撑。随着互联网技术向各行业以及生活各方面的渗透，这样的系统今后会越来越多。加之伴随着云计算和大数据技术的兴起和蓬勃发展，除了系统越来越多之外，系统自身还会变得越来越庞大和复杂。在这么一个总的大趋势下，系统性能的重要性自然不言而喻。你会发现 Brendan 所著的《性能之巅》是如此地契合我们这个时代，本书不是第一本论述系统性能的书，但本书对现有系统性能的方法和理论所做的提炼、概括和归纳，不敢说后无来者，但绝对可以称得上是前无古人的了。

全书翻译由 EMC 资深软件工程师吴寒思、点融网资深运维工程师陈磊与我共同完成，在此感谢二位的辛苦耕耘和我们作为团队三人之间彼此的精诚合作，一年多的翻译历程，大量的时间和精力投入自是不提，但回过头来看整个过程于我们译者自觉仍是获益良多的。本书内

容量大涉及面广，尽管我们付出了许多的辛苦和努力，还是难以避免错误的出现，仍会存在一些不尽如人意的地方，欢迎广大读者批评指正，以便改进。

感谢博文视点出版社主编张春雨对本书出版的大力支持，感谢编辑贾莉对本书的悉心校对；感谢高博学长在翻译道路上给予我的指引；在本书成稿过程中，感谢 EMC 蔡小华、EMC 陈立、EMC 胡世杰、百度冯玮、百度林向东、百度文立经理、淘宝的林应经理，还要感谢我所在的团队上海百度研发中心离线运维组的同事们。另外，更要感谢我的父母和女友吴颖对我的理解和支持。愿这本书的出版给你们带来快乐。

徐章宁

于百度上海研发中心

2013 年 7 月



前言

有已知的已知；有些事情我们知道自己知道。
我们也知道有已知的未知；这是指我们知道有些事情自己不知道。
但是还有未知的未知——有些事情我们不知道自己不知道。
——美国国防部长 唐纳德·拉姆斯菲尔德 2002 年 2 月 12 日

虽然上述的发言在新闻发布会上引来了记者的笑声，但是它总结出了一个重要的原则，适用于任何如地缘政治般复杂的技术系统：性能问题可能来源于任何地方，包括系统中因你一无所知而不曾检查的地方（未知的未知）。本书将揭示许多这样的领域，并为其分析提供方法和工具。

关于本书

欢迎来到《性能之巅：洞悉系统、企业与云计算》！本书以操作系统为背景讲解操作系统和应用程序的性能，针对企业环境和云计算环境编写而成。本书的目的是帮助你更好地利用自己的系统。

当你的工作与持续开发的应用程序软件为伍，你可能会认为内核经过几十年的开发调整，操作系统的性能已是一个解决了的问题了，但事情并非如此！操作系统是一个复杂的软件体，管理着各种不断变化的物理设备，应对着不同的新应用程序的工作负载。内核也在持续地发展，

不断增加新的特性以提高特定的工作负载的性能，随着系统继续扩展，所遇到的瓶颈被逐一移除。改进操作系统性能需要不断的分析和努力，这样才能带来性能的持续提升。应用程序的性能也可以在操作系统的背景下做分析，此点本书也覆盖了。

操作系统范围

本书的重点就是系统性能的研究，所用的工具、示例，乃至可调的参数都是 Linux 系统和基于 Solaris 的系统里的。除非注明，在示例中所用到的操作系统的特定发行版并不重要。基于 Linux 的系统，示例所含的范围从各种裸机系统到虚拟化的运行着 Ubuntu、Fedora 或 CentOS 的云租户。对于基于 Solaris 的系统，示例也是裸机系统以及基于 Joyent SmartOS 或 OmniTI OmniOS 的虚拟化系统。SmartOS 和 OmniOS 用的是开源的 illumos 内核：这是 OpenSolaris 内核的一个活跃的分支，所基于的开发版本后来成为了 Oracle Solaris 11。

覆盖两种不同的操作系统给每位读者提供了一个新的视角，帮助读者可以更深入地理解这两种系统的特点，尤其是二者设计不同的地方，并且可以帮助读者更全面地理解性能，而不只局限于某个单一的系统，这样读者可以更加客观地思考操作系统。

过去，开发者对基于 Solaris 的系统做了较多的性能工作，让其成为某些情形下更好的选择。Linux 的情况也有了很大的改观。在十多年前的 *System Performance Tuning* [Musumeci 02] 出版时，作者同时介绍了 Linux 和 Solaris，但是更侧重后者。作者的理由是：

Solaris 机器更多地注重性能。我怀疑这是因为 Sun 的系统平均来说要比同等的 Linux 系统贵得多。这带来的结果是，花大价钱的人更倾向于挑剔性能，因此 Solaris 在这个领域做的工作更多。如果你的 Linux 机器性能不够好，你可以再买一台并对工作负载做切分——毕竟便宜。如果花了你几百万美金的 Ultra Enterprise 10000 性能不好，你公司也因此会每时每刻都在承受不小的损失，你会打 Sun 的服务电话寻求答案。

上面这段解释了 Sun 注重性能的历史传统：Solaris 的利润是与硬件销售绑定的，不少资金频繁地花在性能的提升上。Sun 需要，也付得起，雇用超过 100 名的全职性能工程师（包括我自己和穆苏梅奇（Musumeci Gian））。与 Sun 的内核工程师团队一起，我们在系统性能领域取得了许多进展。

Linux 在性能工作和观测工具这一块走了很长的路。尤其是现在，Linux 正在应用到大型的云计算环境之中。本书涵盖了许多 Linux 的性能特性，这些特性都是在过去五年里开发起来的。

其他内容

示例会包括性能工具的截屏，这样做不仅是为了显示数据，而且是为了对可用的数据类型做阐释。一般来说工具展现数据的方式更为直观，很多 UNIX 早期风格的工具生成的输出都是相近的，意义常常可以不言自明。这意味着屏幕截图可以很好地传递这些工具的意图，只有某些需要极少的附加说明。（如果一款工具需要费力的说明，这就很可能是一个失败的设计！）

技术的历史演化所展示出的洞察力能深化你的理解，这些都会在书中一一讲到。除此之外，了解一些这个行业的重要人物也是很用的（这个世界很小）：你很可能会碰到他们或者接触到他们在性能领域的工作成果。附录 G 是一张“谁是谁”的清单。

什么未提及

本书着眼于性能。如果你要执行所有的示例任务，有时可能需要做些系统管理员的工作，包括软件的安装或编译（这些本书没有提及）。尤其是在 Linux 上，你需要安装 `sysstat` 软件包，还有很多书中用到的工具也有同样的要求。

书中关于操作系统内部总结的内容会在单独的篇章中有详尽的介绍。对性能分析高阶专题的概述，是为了让你知道这些内容的存在，以便在需要的时候依靠其他的知识来源做进一步的学习。

本书的结构

本书的内容如下。

第 1 章，绪论。介绍系统性能分析，总结关键的概念并展示了与性能相关的一些例子。

第 2 章，方法。性能分析和调整的背景知识，包括术语、概念、模型、观测和实验的方法，容量规划，分析，以及统计。

第 3 章，操作系统。总结了内核内部的性能分析。对于解释和理解操作系统行为，这些是必要的背景知识。

第 4 章，观测工具。介绍系统观测工具的类型，以及构建这些工具所基于的接口和框架。

第 5 章，应用程序。讨论了应用程序性能的内容，并从操作系统的角度观测应用程序。

第 6 章，CPU。内容包括处理器、硬件线程、CPU 缓存、CPU 互联，以及内核调度。

第 7 章，内存。虚拟内存、换页、swapping、内存架构、总线、地址空间和内存分配器。

第 8 章，文件系统。文件系统 I/O 性能，包括涉及的不同缓存。

第 9 章，磁盘。内容包括存储设备、磁盘 I/O 工作负载、存储控制器、RAID，以及内核 I/O 子系统。

第 10 章，网络。网络协议、套接字、接口，以及物理连接。

第 11 章，云计算。介绍广泛应用于云计算的操作系统级和硬件级虚拟化方法，以及这些方法的性能开销、隔离和观测特征。

第 12 章，基准测试。介绍如何精确地做基准测试，如何解读别人的基准测试结果。这是一个棘手的话题，这一章会告诉你怎样避免常见的错误，并试图理解这一点。

第 13 章，案例研究。包含一个系统性能的案例研究，讲述了如何从始至终地分析一个真实的云客户案例。

第 1~4 章提供了必要的背景知识。阅读完这几章后，你可以根据需要参考本书的其余部分。

第 13 章的写法是不同的，该章用讲故事的方法描绘了性能工程师的工作场景。如果你是性能分析的新手，想先了解个大概，可能会想先读读这一章，当读完其他章的时候还可以再次重温。

作为未来的参考

通过着力于系统性能分析的背景知识与方法，本书的编写经得起推敲。

为了做到这一点，许多章都被分为了两个部分。一部分的内容组成是术语、概念和方法（一般附有标题），这些内容许多年后应该还依然中肯适用。另一部分的内容是前一部分如何实现的示例：架构、分析工具，还有可调参数。这部分内容即便有朝一日淘汰了，作为示例讲解也是依然有用的。

跟踪示例

我们经常需要深入探索操作系统，这项工作要用到内核跟踪工具。有很多这样的工具，它们所针对的开发阶段也各不相同，例如，ftrace、perf、DTrace、SystemTap、LTTng 和 ktap。其中有一款工具被选择用在了绝大多数的跟踪示例中，并在 Linux 和基于 Solaris 的系统上都有演示：它就是 DTrace。该工具提供了这些示例所需要的功能，并且关于它还有大量的外部参考资料，包括可以用于高级跟踪的脚本。

你可能需要或愿意选用别的跟踪工具，这很好。DTrace 的示例展示的是你能向系统掷出的

问题。这些问题以及提出这些问题的方法，常常才是最困难的。

目标受众

本书的目标受众主要是系统管理员以及企业与云计算环境的运维工程师。所有需要了解操作系统和应用程序性能的开发人员、数据库管理员和网站管理员都适合参阅本书。

作为云计算提供商的首席性能工程师，我的工作会与支持人员和顾客打交道，他们经常要承受巨大时间压力去解决多个性能问题。对于许多人来说，性能并不是他们的主要工作，但却需要了解足够多的性能知识来解决手头的问题。因为清楚读者学习本书的时间会非常有限，我将本书编写得尽可能简洁。但不会简短：有太多知识需要覆盖才能保证你是准备好了的。

另一个受众群体是学生：本书适合作为系统性能课程的补充教材。在本书的编写期间（以及开始动笔的多年以前）我就曾经教授过这样的课程，并帮助学生解决仿真的性能问题（事前不会公布答案！）。这段经历帮我弄清了什么样的材料能最好地引导学生解决性能问题，这也成就了本书的部分内容。

无论你是不是学生，每章的习题都会带给你一个审视和应用知识的机会。其中有一些可选的高阶练习，可能你完成不了（也许做不到，但至少可以启发思维）。

本书涵盖了足够的知识细节，无论是大公司还是小公司，乃至雇用了不少性能专职人员的公司，本书都可以满足其需要。对于众多的小型公司，日常用到的可能只是书中的某些部分，但本书作为参考也可备不时之需。

排版约定

本书贯穿始终用到的排版约定如下：

<code>netif_receive_skb()</code>	函数名
<code>iostat(1)</code>	Man 手册
<code>Documentation/...</code>	Linux 文档
<code>CONFIG_...</code>	Linux 配置选项
<code>kernel/...</code>	Linux 内核源代码
<code>fs/</code>	Linux 内核源代码，文件系统
<code>usr/src/uts/...</code>	基于 Solaris 内核源代码

#	超级用户 (root) shell 提示符
\$	普通用户 (non-root) shell 提示符
^C	命令被中断 (Ctrl+C)
[...]	文本截断
mpstat 1	键入的命令或高亮的文字

补充材料与参考

下面选出的书籍 (完整的列表见参考书目) 可以作为操作系统背景知识和性能分析学习的更为深入的参考。

- [Jain 91] Jain, R. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.
- [Vahalia 96] Vahalia, U. *UNIX Internals: The New Frontiers*. Prentice Hall, 1996.
- [Cockcroft 98] Cockcroft, A., and R. Pettit. *Sun Performance and Tuning: Java and the Internet*. Prentice Hall, 1998.
- [Musumeci 02] Musumeci, G. D., and M. Loukidas. *System Performance Tuning, 2nd Edition*. O'Reilly, 2002.
- [Bovet 05] Bovet, D., and M. Cesati. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly, 2005.
- [McDougall 06a] McDougall, R., and J. Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. Prentice Hall, 2006.
- [McDougall 06b] McDougall, R., J. Mauro, and B. Gregg. *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. Prentice Hall, 2006.
- [Gove 07] Gove, D. *Solaris Application Programming*. Prentice Hall, 2007.
- [Love 10] Love, R. *Linux Kernel Development, 3rd Edition*. Addison-Wesley, 2010.
- [Gregg 11] Gregg, B., and J. Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall, 2011.



致谢

Deirdré Straughan 再次提供了不可思议的帮助，使我对技术教育的热忱得以延伸至另一本书。从想法到手稿，从最开始就帮助我构想这本书是什么样子，到花费了无数的时间编辑和讨论每一页的内容，找出了许多我没有解释清楚的部分。至今我和她已经合作了超过了 2000 页的技术内容（加上博客），能得到如此大的帮助我深感荣幸。

Barbara Wood 作为拷贝编辑，花了大量的时间在本书的细节上，做了无数的修改，文字才有了现在的质量、可读性和连贯性。考虑到本书的长度和复杂性，这项工作绝不简单，我非常感谢 Barbara 的帮助和他辛勤的工作。

对于每一位给予本书反馈的人，我都心怀感激。这是一本深层次的技术书，有很多新内容需要严谨的审阅——经常需要频繁地反复确认和理解不同内核的内核代码。

不管是深层次的技术还是材料的组织和展示，Darryl Gove 都给予了无与伦比的反馈意见。他本身就是一个作家，看到他是如此迫切地向我们的读者提供最好的内容，我非常期待着他将来的著作。

我还非常感谢 Richard Lowe 和 Robert Mustacchi，他们通审了整本书，发现了我所缺失的内容和一些需要做更好阐述的部分。Richard 对不同内核的内部机理的理解令人震惊，厉害得甚至有点可怕。Robert 对云计算章节给予了极大的帮助，他还将自己在 KVM 上的工作专长转移到了 illumos 上。

感谢 Jim Mauro 和 Dominic Kay 的反馈意见：我曾经与他们一起出过书，理解艰深的技术内容，再把这些内容解释给读者，他们是此中的天才。

Jerry Jelinek 和 Max Bruning, 两人都有着内核工程的专长, 提供了多章的详尽反馈。

Adam Leventhal 对“文件系统”一章和“磁盘”一章给予了专家级建议, 特别是帮我理解了闪存当前的细微差别——他在这个领域有着长期的经验, 在 Sun 公司的时候就发明过不少闪存创新性的使用方法。

David Pacheco 对“应用程序”一章给予了极好的反馈, Dan McDonald 则是对“网络”那一章, 我很幸运可以让他们在自己如此了解的领域把他们自己的技术展示出来。

Carlos Cardenas 看过了整本书, 在统计分析方面给予了独特的建议, 这些建议正是我之前一直所追寻的。

我很感激 Bryan Cantrill、Keith Wesolowski、Paul Eggleton、Marsell Kukuljevic-Pearce 和 Adrian Cockcroft, 为他们的反馈和贡献。Adrian 的意见促使我重新排列章节顺序, 这让读者可以更好地关联所覆盖的内容。

我感谢在我之前的作者们, 他们的名字都列在了参考附录之中, 是他们铺就了通往系统性能的道路并把自己的发现记录了下来。感谢与我一同工作多年的性能专家们, 包括 Bryan Cantrill、Roch Bourbonnais、Jim Mauro、Richard McDougall, 等等。我从他们身上学到了很多很多。

感谢 Bryan Cantrill 对这个项目的支持, 感谢 Jason Hoffman 的热忱。

感谢 Claire、Mitchell 和其他家人朋友为支持我这个项目所做出的牺牲。

特别要感谢 Pearson 公司的高级总监 Greg Doench, 感谢他的帮助、耐心和对项目的建议。

我很享受本书编写的过程, 即使期间也有过不时的气馁。要是十年前的话, 我写起来会容易得多, 那时候我对复杂性和系统性能微妙的所知不及现在, 我在企业、存储和云计算领域做过软件工程师、内核工程师和系统工程师。无论是栈的各个层级, 还是从应用程序到硬件, 在所有这些地方我都处理过性能问题。这些经历, 我知道还有好多都还没有记录下来。所有这些既让我受挫, 也激励着我把它们写下来。这本书是我一直想要写就的, 本书的完成是我莫大的安慰。



关于作者

Brendan Gregg 是 Joyent 公司的首席性能工程师，负责分析云计算环境的性能和扩展，覆盖从小型到大型的云计算环境和软件栈的所有级别。他是 *DTrace* 一书的主作者（Prentice Hall 出版社，2011 年），是 *Solaris Performance and Tools* 一书的合著者（Prentice Hall 出版社，2007 年），撰写了许多与系统性能相关的文章。他之前是 Sun Microsystems 公司的性能主管和内核工程师，同时也是性能顾问兼培训师。是他开发了 DTraceToolkit 和 ZFS L2ARC，他所开发的许多 DTrace 脚本都收录在 Mac OS X 和 Oracle Solaris 11 的默认发行版中。性能的可视化是他最近从事的工作之一。

十载耕耘奠定专业地位

以书为证彰显卓越品质

博文视点诚邀精锐作者加盟

《C++Primer (中文版) (第5版)》、《淘宝技术这十年》、《代码大全》、《Windows内核情景分析》、《加密与解密》、《编程之美》、《VC++深入详解》、《SEO实战密码》、《PPT演义》……

“圣经”级图书光耀夺目,被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、咎辉Zac、李刚、曹江华……

“明星”级作者济济一堂,他们的名字熠熠生辉,与IT业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治,成就博古通今、文圆质方、视角独特、点石成金之计算机图书的风向标杆:博文视点。

“凤翱翔于千仞兮,非梧不栖”,博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟,与大师并列于IT专业出版之巅峰。

英雄帖

江湖风云起,代有才人出。

IT界群雄并起,逐鹿中原。

博文视点诚邀天下技术英豪加入,

指点江山,激扬文字

传播信息技术,分享IT心得

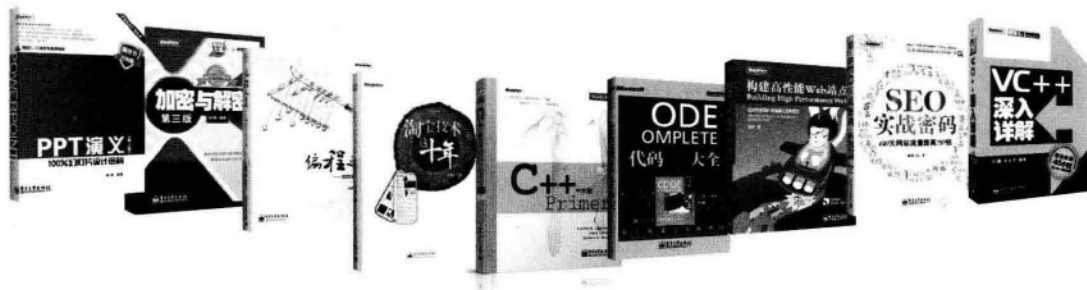
• 专业的作者服务 •

博文视点自成立以来一直专注于IT专业技术图书的出版,拥有丰富的与技术图书作者合作的经验,并参照IT技术图书的特点,打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持:

善待作者——我们会把出版流程整理得清晰简明,为作者提供优厚的稿酬服务,解除作者的顾虑,安心写作,展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯,并会参照作者实际的工作、生活节奏,量身定制写作计划,确保合作顺利进行。

提升作者——我们打造精品图书,更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力,为作者的事业开拓带来更多的机会。



联系我们

博文视点官网: <http://www.broadview.com.cn>

CSDN官方博客: <http://blog.csdn.net/broadview2006/>

投稿电话: 010-51260888 88254368

投稿邮箱: jsj@phei.com.cn



@博文视点Broadview



微信公众账号

博文视点Broadview





目录

前言	36
致谢	42
关于作者	44
第 1 章 绪论	1
1.1 系统性能	1
1.2 人员	2
1.3 事情	3
1.4 视角	4
1.5 性能是充满挑战的	4
1.5.1 性能是主观的	4
1.5.2 系统是复杂的	5
1.5.3 可能有多问题并存	6
1.6 延时	6
1.7 动态跟踪	7
1.8 云计算	8

1.9 案例研究	8
1.9.1 缓慢的磁盘	9
1.9.2 软件变更	10
1.9.3 更多阅读	12
 第2章 方法	 13
2.1 术语	14
2.2 模型	14
2.2.1 受测系统	15
2.2.2 排队系统	15
2.3 概念	16
2.3.1 延时	16
2.3.2 时间量级	17
2.3.3 权衡三角	18
2.3.4 调整的影响	19
2.3.5 合适的层级	19
2.3.6 性能建议的时间点	20
2.3.7 负载 vs. 架构	20
2.3.8 扩展性	21
2.3.9 已知的未知	22
2.3.10 指标	23
2.3.11 使用率	24
2.3.12 饱和度	25
2.3.13 剖析	26
2.3.14 缓存	26
2.4 视角	28
2.4.1 资源分析	28
2.4.2 工作负载分析	29
2.5 方法	30
2.5.1 街灯讹方法	31
2.5.2 随机变动讹方法	32
2.5.3 责怪他人讹方法	32
2.5.4 Ad Hoc 核对清单法	33

2.5.5	问题陈述法	33
2.5.6	科学法	34
2.5.7	诊断循环	35
2.5.8	工具法	35
2.5.9	USE 方法	36
2.5.10	工作负载特征归纳	42
2.5.11	向下挖掘分析	43
2.5.12	延时分析	44
2.5.13	R 方法	45
2.5.14	事件跟踪	45
2.5.15	基础线统计	47
2.5.16	静态性能调整	47
2.5.17	缓存调优	47
2.5.18	微基准测试	48
2.6	建模	49
2.6.1	企业 vs. 云	49
2.6.2	可视化识别	49
2.6.3	Amdahl 扩展定律	51
2.6.4	通用扩展定律	52
2.6.5	排队理论	52
2.7	容量规划	56
2.7.1	资源极限	56
2.7.2	因素分析	58
2.7.3	扩展方案	58
2.8	统计	59
2.8.1	量化性能	59
2.8.2	平均值	60
2.8.3	标准方差、百分位数、中位数	61
2.8.4	变异系数	62
2.8.5	多重模态分布	62
2.8.6	异常值	63
2.9	监视	63
2.9.1	基于时间的规律	63
2.9.2	监测产品	65

2.9.3 启动以来的信息统计	65
2.10 可视化	65
2.10.1 线图	65
2.10.2 散点图	66
2.10.3 热图	67
2.10.4 表面图	68
2.10.5 可视化工具	69
2.11 练习	70
2.12 参考	70
第3章 操作系统	73
3.1 术语	73
3.2 背景	74
3.2.1 内核	74
3.2.2 栈	77
3.2.3 中断和中断线程	78
3.2.4 中断优先级	79
3.2.5 进程	79
3.2.6 系统调用	81
3.2.7 虚拟内存	83
3.2.8 内存管理	83
3.2.9 调度器	84
3.2.10 文件系统	85
3.2.11 缓存	87
3.2.12 网络	88
3.2.13 设备驱动	88
3.2.14 多处理器	88
3.2.15 抢占	89
3.2.16 资源管理	89
3.2.17 观测性	90
3.3 内核	90
3.3.1 UNIX	91
3.3.2 基于 Solaris	91

3.3.3 基于 Linux	94
3.3.4 差异	96
3.4 练习	97
3.5 参考	97
 第 4 章 观测工具	 99
4.1 工具类型	99
4.1.1 计数器	100
4.1.2 跟踪	101
4.1.3 剖析	102
4.1.4 监视 (sar)	103
4.2 观测来源	104
4.2.1 /proc	104
4.2.2 /sys	109
4.2.3 kstat	110
4.2.4 延时核算	112
4.2.5 微状态核算	113
4.2.6 其他的观测源	113
4.3 DTrace	115
4.3.1 静态和动态跟踪	116
4.3.2 探针	117
4.3.3 provider	117
4.3.4 参数	118
4.3.5 D 语言	118
4.3.6 内置变量	119
4.3.7 action	119
4.3.8 变量类型	120
4.3.9 单行命令	122
4.3.10 脚本	122
4.3.11 开销	123
4.3.12 文档和资源	124
4.4 SystemTap	125
4.4.1 探针	125

4.4.2	tapset.....	126
4.4.3	action 和内置变量.....	126
4.4.4	示例.....	126
4.4.5	开销.....	128
4.4.6	文档和资源.....	129
4.5	perf.....	129
4.6	观测工具的观测.....	130
4.7	练习.....	131
4.8	参考.....	131
第 5 章 应用程序.....		133
5.1	应用程序基础.....	133
5.1.1	目标.....	134
5.1.2	常见情况的优化.....	135
5.1.3	观测性.....	136
5.1.4	大 O 标记法.....	136
5.2	应用程序性能技术.....	137
5.2.1	选择 I/O 尺寸.....	137
5.2.2	缓存.....	138
5.2.3	缓冲区.....	138
5.2.4	轮询.....	138
5.2.5	并发和并行.....	139
5.2.6	非阻塞 I/O.....	141
5.2.7	处理器绑定.....	141
5.3	编程语言.....	142
5.3.1	编译语言.....	142
5.3.2	解释语言.....	143
5.3.3	虚拟机.....	144
5.3.4	垃圾回收.....	144
5.4	方法和分析.....	145
5.4.1	线程状态分析.....	145
5.4.2	CPU 剖析.....	148
5.4.3	系统调用分析.....	150

5.4.4	I/O 剖析	156
5.4.5	工作负载特征归纳	157
5.4.6	USE 方法	157
5.4.7	向下挖掘法	158
5.4.8	锁分析	158
5.4.9	静态性能调优	161
5.5	练习	162
5.6	参考	163
第 6 章	CPU	165
6.1	术语	166
6.2	模型	166
6.2.1	CPU 架构	166
6.2.2	CPU 内存缓存	167
6.2.3	CPU 运行队列	168
6.3	概念	168
6.3.1	时钟频率	168
6.3.2	指令	169
6.3.3	指令流水线	169
6.3.4	指令宽度	170
6.3.5	CPI, IPC	170
6.3.6	使用率	170
6.3.7	用户时间/内核时间	171
6.3.8	饱和度	171
6.3.9	抢占	171
6.3.10	优先级反转	172
6.3.11	多进程, 多线程	172
6.3.12	字长	173
6.3.13	编译器优化	174
6.4	架构	174
6.4.1	硬件	174
6.4.2	软件	182
6.5	方法	187

6.5.1	工具法	187
6.5.2	USE 方法	188
6.5.3	负载特征归纳	189
6.5.4	剖析	190
6.5.5	周期分析	191
6.5.6	性能监控	192
6.5.7	静态性能调优	192
6.5.8	优先级调优	192
6.5.9	资源控制	193
6.5.10	CPU 绑定	193
6.5.11	微型基准测试	194
6.5.12	扩展	194
6.6	分析	195
6.6.1	uptime	195
6.6.2	vmstat	197
6.6.3	mpstat	198
6.6.4	sar	200
6.6.5	ps	201
6.6.6	top	202
6.6.7	prstat	203
6.6.8	pidstat	204
6.6.9	time 和 ptime	205
6.6.10	DTrace	206
6.6.11	SystemTap	212
6.6.12	perf	212
6.6.13	cpustat	218
6.6.14	其他工具	219
6.6.15	可视化	219
6.7	实验	222
6.7.1	Ad Hoc	222
6.7.2	SysBench	223
6.8	调优	223
6.8.1	编译器选项	224
6.8.2	调度优先级和调度类	224

6.8.3	调度器选项	224
6.8.4	进程绑定	226
6.8.5	独占 CPU 组	227
6.8.6	资源控制	227
6.8.7	处理器选项（BIOS 调优）	227
6.9	练习	228
6.10	参考资料	229
第 7 章	内存	231
7.1	术语	232
7.2	概念	232
7.2.1	虚拟内存	233
7.2.2	换页	233
7.2.3	按需换页	234
7.2.4	过度提交	236
7.2.5	交换	236
7.2.6	文件系统缓存占用	236
7.2.7	使用率和饱和度	237
7.2.8	分配器	237
7.2.9	字长	237
7.3	架构	237
7.3.1	硬件	238
7.3.2	软件	242
7.3.3	进程地址空间	247
7.4	方法	251
7.4.1	工具法	252
7.4.2	USE 方法	252
7.4.3	使用特征归纳	253
7.4.4	周期分析	254
7.4.5	性能监测	254
7.4.6	泄漏检测	255
7.4.7	静态性能调优	255
7.4.8	资源控制	256

7.4.9	微基准测试	256
7.5	分析	256
7.5.1	vmstat	257
7.5.2	sar	259
7.5.3	slabtop	262
7.5.4	::kmstat	263
7.5.5	ps	264
7.5.6	top	265
7.5.7	prstat	266
7.5.8	pmap	267
7.5.9	DTrace	268
7.5.10	SystemTap	272
7.5.11	其他工具	272
7.6	调优	273
7.6.1	可调参数	274
7.6.2	多个页面大小	276
7.6.3	分配器	277
7.6.4	资源控制	277
7.7	练习	277
7.8	参考资料	279
第 8 章	文件系统	281
8.1	术语	282
8.2	模型	282
8.2.1	文件系统接口	282
8.2.2	文件系统缓存	283
8.2.3	二级缓存	284
8.3	概念	284
8.3.1	文件系统延时	284
8.3.2	缓存	285
8.3.3	随机与顺序 I/O	285
8.3.4	预取	286
8.3.5	预读	287

8.3.6	写回缓存	287
8.3.7	同步写	287
8.3.8	裸 I/O 和直接 I/O	288
8.3.9	非阻塞 I/O	288
8.3.10	内存映射文件	289
8.3.11	元数据	289
8.3.12	逻辑 I/O vs. 物理 I/O	290
8.3.13	操作并非不平等	291
8.3.14	特殊文件系统	292
8.3.15	访问时间戳	292
8.3.16	容量	292
8.4	架构	293
8.4.1	文件系统 I/O 栈	293
8.4.2	VFS	294
8.4.3	文件系统缓存	294
8.4.4	文件系统特性	299
8.4.5	文件系统种类	300
8.4.6	卷和池	305
8.5	方法	306
8.5.1	磁盘分析	307
8.5.2	延时分析	307
8.5.3	负载特征归纳	309
8.5.4	性能监控	311
8.5.5	事件跟踪	311
8.5.6	静态性能调优	312
8.5.7	缓存调优	313
8.5.8	负载分离	313
8.5.9	内存文件系统	313
8.5.10	微型基准测试	313
8.6	分析	315
8.6.1	vfsstat	315
8.6.2	fsstat	316
8.6.3	strace、truss	317
8.6.4	DTrace	317

8.6.5	SystemTap	326
8.6.6	LatencyTOP	326
8.6.7	free	327
8.6.8	top	327
8.6.9	vmstat	327
8.6.10	sar	328
8.6.11	slabtop	329
8.6.12	mdb::kmastat	330
8.6.13	fcachestat	330
8.6.14	/proc/meminfo	331
8.6.15	mdb::memstat	331
8.6.16	kstat	332
8.6.17	其他工具	333
8.6.18	可视化	334
8.7	实验	334
8.7.1	Ad Hoc	335
8.7.2	微型基准测试工具	335
8.7.3	缓存写回	337
8.8	调优	337
8.8.1	应用程序调用	338
8.8.2	ext3	339
8.8.3	ZFS	339
8.9	练习	341
8.10	参考资料	342

第9章 磁盘

9.1	术语	346
9.2	模型	346
9.2.1	简单磁盘	346
9.2.2	缓存磁盘	347
9.2.3	控制器	348
9.3	概念	348
9.3.1	测量时间	348

9.3.2	时间尺度	350
9.3.3	缓存	351
9.3.4	随机 vs.连续 I/O	351
9.3.5	读/写比	352
9.3.6	I/O 大小	352
9.3.7	IOPS 并不平等	353
9.3.8	非数据传输磁盘命令	353
9.3.9	使用率	353
9.3.10	饱和度	354
9.3.11	I/O 等待	354
9.3.12	同步 vs.异步	355
9.3.13	磁盘 vs.应用程序 I/O	355
9.4	架构	356
9.4.1	磁盘类型	356
9.4.2	接口	361
9.4.3	存储类型	362
9.4.4	操作系统磁盘 I/O 栈	364
9.5	方法	367
9.5.1	工具法	368
9.5.2	USE 方法	368
9.5.3	性能监控	369
9.5.4	负载特征归纳	370
9.5.5	延时分析	371
9.5.6	事件跟踪	372
9.5.7	静态性能调优	373
9.5.8	缓存调优	374
9.5.9	资源控制	374
9.5.10	微基准测试	374
9.5.11	伸缩	375
9.6	分析	376
9.6.1	iostat	377
9.6.2	sar	384
9.6.3	pidstat	385
9.6.4	DTrace	386

9.6.5	SystemTap	394
9.6.6	perf.....	394
9.6.7	iostat.....	395
9.6.8	iosnoop.....	397
9.6.9	blktrace	400
9.6.10	MegaCli	401
9.6.11	smartctl	402
9.6.12	可视化	403
9.7	实验	406
9.7.1	Ad Hoc.....	406
9.7.2	自定义负载生成器	407
9.7.3	微基准测试工具	407
9.7.4	随机读示例	407
9.8	调优	408
9.8.1	操作系统可调参数	408
9.8.2	磁盘设备可调参数	410
9.8.3	磁盘控制器可调参数	410
9.9	练习	411
9.10	参考资料	412

第 10 章 网络.....415

10.1	术语	416
10.2	模型	416
10.2.1	网络接口	416
10.2.2	控制器	417
10.2.3	协议栈	417
10.3	概念	418
10.3.1	网络和路由	418
10.3.2	协议	419
10.3.3	封装	419
10.3.4	包长度	419
10.3.5	延时	420
10.3.6	缓冲	422

10.3.7	连接积压队列	422
10.3.8	接口协商	422
10.3.9	使用率	423
10.3.10	本地连接	423
10.4	架构	423
10.4.1	协议	423
10.4.2	硬件	426
10.4.3	软件	428
10.5	方法	432
10.5.1	工具法	433
10.5.2	USE 方法	433
10.5.3	工作负载特征归纳	434
10.5.4	延时分析	435
10.5.5	性能监测	436
10.5.6	数据包嗅探	436
10.5.7	TCP 分析	437
10.5.8	挖掘分析	438
10.5.9	静态性能调优	438
10.5.10	资源控制	439
10.5.11	微基准测试	439
10.6	分析	440
10.6.1	netstat	440
10.6.2	sar	445
10.6.3	ifconfig	447
10.6.4	ip	448
10.6.5	nicstat	448
10.6.6	dladm	449
10.6.7	ping	450
10.6.8	tracert	450
10.6.9	pathchar	451
10.6.10	tcpdump	451
10.6.11	snoop	452
10.6.12	Wireshark	455
10.6.13	DTrace	455

10.6.14	SystemTap	466
10.6.15	perf.....	466
10.6.16	其他工具	467
10.7	实验	468
10.7.1	iperf.....	468
10.8	调优	469
10.8.1	Linux.....	470
10.8.2	Solaris	472
10.8.3	配置	474
10.9	练习	475
10.10	参考	476
第 11 章 云计算		479
11.1	背景	480
11.1.1	性价比	480
11.1.2	可扩展的架构	480
11.1.3	容量规划	481
11.1.4	存储	483
11.1.5	多租户	483
11.2	OS 虚拟化	484
11.2.1	系统开销	485
11.2.2	资源控制	487
11.2.3	可观测性	490
11.3	硬件虚拟化	495
11.3.1	系统开销	496
11.3.2	资源控制	501
11.3.3	可观测性	504
11.4	比较	509
11.5	练习	511
11.6	参考资料	512

第 12 章 基准测试	515
12.1 背景	515
12.1.1 事情	516
12.1.2 有效的基准测试	516
12.1.3 基准测试之罪	518
12.2 基准测试的类型	523
12.2.1 微基准测试	524
12.2.2 模拟	525
12.2.3 回放	526
12.2.4 行业标准	526
12.3 方法	528
12.3.1 被动基准测试	528
12.3.2 主动基准测试	529
12.3.3 CPU 剖析	531
12.3.4 USE 方法	532
12.3.5 工作负载特征归纳	533
12.3.6 自定义基准测试	533
12.3.7 逐渐增加负载	533
12.3.8 完整性检查	535
12.3.9 统计分析	536
12.4 基准测试问题	537
12.5 练习	538
12.6 参考	539
第 13 章 案例研究	541
13.1 案例研究：红鲸	541
13.1.1 问题陈述	542
13.1.2 支持	543
13.1.3 上手	544
13.1.4 选择征途	545
13.1.5 USE 方法	546
13.1.6 我们做完了吗	549
13.1.7 二度出击	549

13.1.8 基础	550
13.1.9 忽略红鲸	551
13.1.10 审问内核	552
13.1.11 为什么	553
13.1.12 尾声	555
13.2 结语	555
13.3 附加信息	556
13.4 参考	556
附录 A USE 法: Linux	559
附录 B USE 法: Solaris	565
附录 C sar 总结	571
附录 D DTrace 单行命令	573
附录 E 从 DTrace 到 SystemTap	583
附录 F 精选练习题答案	593
附录 G 系统性能名人录	597

绪论

性能是一门令人激动的，富于变化同时又充满挑战的学科。本章会引领你进入性能的领域，尤其针对系统性能，阐述涉及的人员、所做的事情、分析的视角和面临的挑战。我将介绍一项非常重要的性能指标——延时，同时还会介绍计算机领域一些的新进展：动态跟踪和云计算。为了帮助读者更好地理解，本章还提供了一些与性能相关的例子。

1.1 系统性能

系统性能是对整个系统的研究，包括了所有的硬件组件和整个软件栈。所有数据路径上和软硬件上所发生的事情都包括在内，因为这些都有可能影响性能。对于分布式系统来说，这意味着多台服务器和多个应用。如果你还没有你的环境的一张示意图，用来显示数据的路径，赶紧找一张或者自己画一张。它可以帮助你理解所有组件的关系，并确保你不会只见树木不见森林。

图 1.1 呈现的是单台服务器上的通用系统软件栈，包括操作系统（OS）内核、数据库和应用程序层。术语“全栈”（entire stack）有时一般仅仅指的是应用程序环境，包括数据库、应用程序，以及网站服务器。不过，当论及系统性能时，我们用全栈来表示所有事情，包括系统库和内核。

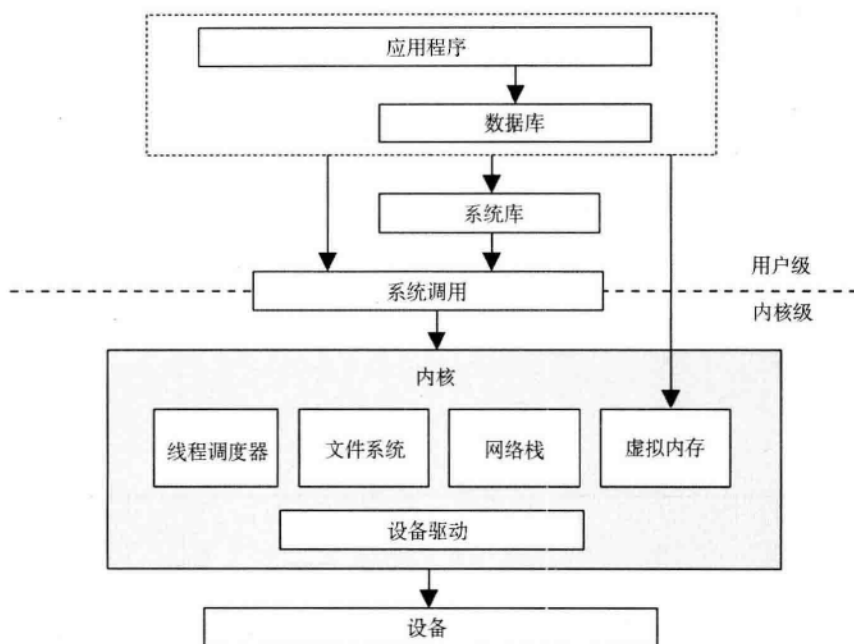


图 1.1 通用系统软件栈

本书第 3 章“操作系统”，将详细讨论这个软件栈，在后面几章里还会有更深入的研究。本章接下来的部分主要讲述系统性能和性能的概要。

1.2 人员

系统性能是一项需要多类人员参与的事务，其中包括系统管理员、技术支持人员、应用开发者、数据库管理员和网络管理员。对于他们的大多数来说，性能是一项兼职的事情，他们可能会有发掘性能的倾向，但仅限于本职工作范围内（网络团队检查网络、数据库团队检查数据库，如此等等）。然而，对于某些性能问题，要找到根本原因还需要这些团队一起协同工作才行。

一些公司会雇用性能工程师，其主要任务就是维护系统性能。他们与多个团队协同工作，对环境做全局性的研究，执行一些对解决复杂性能问题至关重要的操作。与此同时，他们会开发更好的工具，对整个环境做系统级分析（system-wide analysis），为容量规划（capacity planning）定义指标。

在性能领域也有某些专职于特定应用程序的工种，例如，Java 性能工程师和 MySQL 性能工程师。通常他们在开始使用特定的应用程序工具之前，会做一些系统性能检查，但是有限。

1.3 事情

性能领域包括了以下的事情，我按照理想的执行顺序将它们排列如下：

1. 设置性能目标和建立性能模型
2. 基于软件或硬件原型进行性能特征归纳
3. 对开发代码进行性能分析（软件整合之前）
4. 执行软件非回归性测试（软件发布前或发布后）
5. 针对软件发布版本的基准测试
6. 目标环境中的概念验证（Proof-of-concept）测试
7. 生产环境部署的配置优化
8. 监控生产环境中运行的软件
9. 特定问题的性能分析

步骤 1~5 是传统软件产品开发过程的一部分。产品发行之后，接下来要么是在客户环境中进行概念验证测试，要么直接进行部署和配置。如果在客户环境中碰到问题（步骤 6~9），这说明该问题在软件开发阶段没有得到发现和修复。

理想情况下，在硬件选型和软件开发之前，性能工程师就应该开始工作。作为工作的第一步，可以设定性能目标并建立一个性能模型。产品开发过程常常缺失了这一步，性能工程工作被推迟直到问题出现。在架构决策确定之后，随着软件开发工作的一步步推进，修复性能问题的难度会变得越来越大会。

术语容量规划（capacity planning）指的是一系列事前行动。在设计阶段，包括通过研究开发软件的资源占用情况，来得知原有设计在多大程度上能满足目标需求。在部署后，包括监控资源的使用情况，这样问题在出现之前就能被预测。

能够有助于完成上述事情的方法和工具在本书中都有覆盖。

对于不同的公司和不同的产品，环境和要做的事情都各不相同，多数情况下，不需要全部执行以上的九步。你的工作可能集中于某几步或者仅仅是其中的一步。

1.4 视角

与很多事情专注于一点不同，性能是可以从不同的视角来审视的。图 1.2 展示了两种性能分析的视角：负载分析（workload analysis）和资源分析（resource analysis），二者从不同的方向对软件栈做分析。

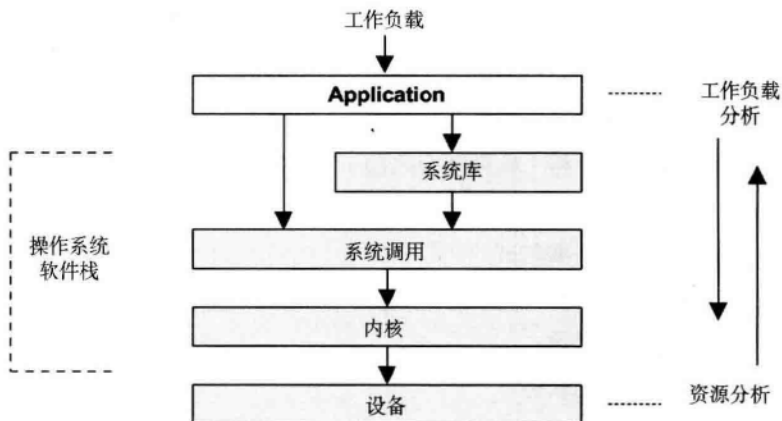


图 1.2 分析视角

系统管理员作为系统资源的负责人，通常采用资源分析视角。应用程序开发人员，对最终实现的负载性能负责，通常采用负载分析视角。每一种视角都有自身的优势，第 2 章将详细讨论这些。尝试从两个视角都进行分析，对于解决某些具有挑战性的问题是不无裨益的。

1.5 性能是充满挑战的

系统性能工程是一个充满挑战的领域，具体原因有很多，其中包括以下事实，系统性能是主观的、复杂的，而且常常是多问题并存的。

1.5.1 性能是主观的

技术学科往往是客观的，太多的业界人士审视问题非黑即白。在进行软件故障查找的时候，判断 bug 是否存在或 bug 是否修复就是这样。bug 的出现总是伴随着错误信息，错误信息通常容易解读，进而你就明白错误为什么会出现了。

与此不同,性能常常是主观性的。开始着手性能问题的时候,对问题是否存在的判断都有可能是模糊的,在问题被修复的时候也同样,被一个用户认为是“不好”的性能,另一个用户可能认为是“好”的。

考虑下面的信息:

磁盘的平均 I/O 响应时间是 1ms。

这是“好”还是“坏”?响应时间或者说是延时,虽然作为最好的衡量指标之一,但还是难以用来说明延时的情况。从某种程度上说,一个给定指标是“好”或“坏”取决于应用开发人员和最终用户的性能预期。

通过定义清晰的目标,诸如目标平均响应时间,或者对落进一定响应延时范围内的请求统计其百分比,可以把主观的性能变得客观化。第2章将介绍处理这类主观性的方法,其中就有通过统计操作延时的比例来进行延时分析的方法。

1.5.2 系统是复杂的

除了主观性之外,性能工程作为一门充满了挑战的学科,除了因为系统的复杂性,还因为对于性能,我们常常缺少一个明确的分析起点。有时我们只是从猜测开始,比如,责怪网络,而性能分析必须对这是不是一个正确的方向做出判断。

性能问题可能出在子系统之间复杂的互联上,即便这些子系统隔离时表现得都很好。也可能由于连锁故障(cascading failure)出现性能问题,这指的是一个出现故障的组件会导致其他组件产生性能问题。要理解这些产生的问题,你必须理清组件之间的关系,还要了解它们是怎样协作的。

瓶颈往往是复杂的,还会以意想不到的方式互相联系。修复了一个问题可能只是把瓶颈推向了系统里的其他地方,导致系统的整体性能并没有得到期望的提升。

除了系统的复杂性之外,生产环境负载的复杂特性也可能导致性能问题。在实验室环境很难重现这类情况,或者只能间歇式地重现。

解决复杂的性能问题常常需要全局性的方法。整个系统——包括自身内部和外部的交互——都可能需要被调查研究。这项工作要求有非常广泛的技能,一般不太可能集中在一人身上,这促使性能工程成为一门多变的并且充满智力挑战的工作。

如同第2章要介绍的内容一样,多样的方法可以带我们穿越这些复杂性的重重迷雾。第6~10章讲的是针对特定系统资源的方法,这些系统资源包括CPU、内存、文件系统、磁盘和网络。

1.5.3 可能有多问题并存

找到一个性能问题点往往并不是问题本身，在复杂的软件中通常会有多个问题。为了证明这一点，试着找到你的操作系统或应用程序的 bug 数据库，然后搜索性能 *performance* 一词，对于结果你多半会很吃惊！一般情况下，成熟的软件，即便是那些被认为拥有高性能的软件，也会有不少已知的但仍未被修复的性能问题。这就造成了性能分析的又一个难点：真正的任务不是寻找问题，而是辨别问题或者说是辨别哪些问题是重要的。

要做到这一点，性能分析必须量化（quantify）问题的重要程度。某些性能问题可能并不适用于你的工作负载或者只在非常小的程度上适用。理想情况下，你不仅要量化问题，还要估计每个问题修复后能带来的增速。当管理层审查工程或运维资源的开销缘由时，这类信息尤其有用。

有一个指标非常适合用来量化性能，那就是延时（latency）。

1.6 延时

延时测量的是用于等待的时间。广义来说，它可以表示所有操作完成的耗时，例如，一次应用程序请求、一次数据库查询、一次文件系统操作，等等。举个例子，延时可以表示从点击链接到屏幕显示整个网页加载完成的时间。这是一个对于客户和网站提供商来说都非常重要的指标：高延时会令人沮丧，客户可能会选择到别处开展业务。

作为一个指标，延时可以估计最大增速（maximum speedup）。举个例子，图 1.3 显示了一次数据库查询需要 100ms 的时间（这就是延时），其中 80ms 的阻塞是等待磁盘读取。通过减少磁盘读取时间（如通过使用缓存）可以达到最好的性能提升，并且可以计算出结果是五倍速（5x）。这就是估计出的增速，而且该计算还可以对性能问题做量化：是磁盘读取让查询慢了 5 倍。

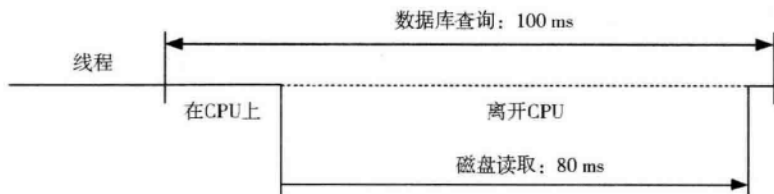


图 1.3 磁盘 I/O 延时示例

这样的计算对其他的指标类型不一定适用。比如，每秒发生的 I/O 操作次数（IOPS），取决于 I/O 的类型，往往不具备直接的可比性。如果一个变化导致 IOPS 下降了 80%，也很难知道

这带来的性能影响会是怎样的。有可能是 IOPS 小了 5 倍，但若所有这些 I/O 的数据量（字节）都变大 10 倍了呢？

若以网络作为讨论背景，延时指的是建立一个连接的时间，而不是数据传输的时间。在本书里，每一章的开头都有术语解释，以澄清这类因场景不同所造成的术语含义的差别。

虽然延时是一个非常有用的指标，但也不是随时随地都能获得。某些系统只有平均延时，某些系统则完全没有延时指标。动态跟踪（dynamic tracing）可以从任意感兴趣的点测量延时，还可以提供数据以显示延时完整的分布情况。

1.7 动态跟踪

动态跟踪技术把所有的软件变得可以监控，而且能用在真实的生产环境中。这项技术利用内存中的 CPU 指令并在这些指令之上动态构建监测数据。这样从任何运行的软件中都可以获得定制化的性能统计数据，从而提供了远超系统的自带统计所能给予的观测性。从前因为不易观测而无法处理的问题变得可以解决。从前可以解决而难以解决的问题，现在也往往可以得以简化。

动态跟踪与传统的观测方法相比是如此不同，甚至让人很难一开始就抓住动态跟踪的要领。以操作系统内核举例：分析内核好比闯进了一间黑屋，拿着蜡烛（系统自带统计）去照亮内核工程师他们觉得需要照亮的地方，而动态跟踪则像是手电筒，你可以指哪儿亮哪儿。

DTrace 是第一个适用于生产环境的动态跟踪工具，它提供了许多其他的功能，甚至包括一套自用的编程语言，D 语言。DTrace 是由 Sun Microsystems 公司开发，在 2005 年随着 Solaris 10 操作系统一同发布的。它是第一个开源的 Solaris 组件，在此之后还移植到了 Mac OS X 和 FreeBSD 上，针对 Linux 的移植现在还在进行中。

在 DTrace 之前，系统跟踪（system tracing）常常使用静态探针（static probes）：置于内核和其他软件之上的一小套监测点。这种方法的观测是有限的，一般用起来很费时，需要配置、跟踪、导出数据以及最后分析一整套流程。

DTrace 对用户态和内核态的软件都提供了静态跟踪和动态跟踪，并且数据是实时产生的。下面的例子跟踪了 ssh 登录的进程执行过程。这次跟踪是系统级别的（不是简单地与一个指定进程 ID 相关联）。

```
# dtrace -n 'exec-success { printf("%d %s", timestamp, curpsinfo->pr_psargs); }'
dtrace: description 'exec-success' matched 1 probe
CPU    ID      FUNCTION:NAME
  2    1425    exec_common:exec-success 732006240859060 sh -c /usr/bin/locale -a
  2    1425    exec_common:exec-success 732006246584043 /usr/bin/locale -a
  5    1425    exec_common:exec-success 732006197695333 sh -c /usr/bin/locale -a
```

```
5 1425 exec_common:exec-success 732006202832470 /usr/bin/locale -a
0 1425 exec_common:exec-success 732007379191163 uname -r
0 1425 exec_common:exec-success 732007449358980 sed -ne /^# START exclude/,/^#
FINISH exclude/p /etc/bash/bash_completion
1 1425 exec_common:exec-success 732007353365711 -bash
1 1425 exec_common:exec-success 732007358427035 /usr/sbin/quota
2 1425 exec_common:exec-success 732007368823865 /bin/mail -E
12 1425 exec_common:exec-success 732007374821450 uname -s
15 1425 exec_common:exec-success 732007365906770 /bin/cat -s /etc/motd
```

在这个示例中，DTrace 被要求打印出时间戳（纳秒）、进程名和参数。用 D 语言可以写出更多复杂的脚本，这让我们可以创建并定制对延时的测量。

DTrace 和动态跟踪将在第 4 章中讲解。在后续各章里也会有许多基于 Linux 和 Solaris 系统的 DTrace 命令和脚本。另有一本讲解 DTrace 更高阶应用的书[Gregg 11]。

1.8 云计算

给系统性能带来影响的最新进展来自云计算和云计算的根基——虚拟技术的兴起。

云计算采用的架构能让应用程序均衡分布于数目不断增多的小型系统中，这让快速扩展成为可能。这种方法还降低了对容量规划的精确程度的要求，因为更多的容量可以很便捷地在云端添加。在某些情况下，它对性能分析的需求更高了：使用较少的资源就意味着系统更少。云的使用通常是按小时计费的，性能的优势可以减少系统的使用数目，从而直接节约成本。这和企业用户的情况不同，企业用户被一个支持协议锁定数年，直到合同终结都可能无法实现成本的节约。

云计算和虚拟化技术也带来了新的难题，这包括，如何管理其他租户（tenant，有时被称作性能隔离（performance isolation））带来的性能影响，以及如何让每个租户都能对物理系统做观测。举个例子，除非系统管理得很好，否则磁盘 I/O 性能可能因为同邻近租户的竞争而下降。在某些环境中，并不是每一个租户都能观察到物理磁盘的真实使用情况，这让问题的甄别变得困难。

这些内容都在第 11 章中有介绍。

1.9 案例研究

案例研究会讲述什么时候该做什么事和为什么要做这些事，如果你刚接触系统性能，把这些关联到自己当前的环境会对你有所帮助。接下来是两个虚构的示例；一个是与磁盘 I/O 相关的性能问题，另一个是软件新版本的性能测试。

这些案例研究中所做的事情在本书的其他章节能找到相关解释。此处并不是为了表现方法的正确性或是唯一性，而是为了展示一种执行性能研究的方式，对此你可以好好思考。

1.9.1 缓慢的磁盘

Scott 是一家中型公司里的系统管理员。数据库团队报告了一个支持 ticket（工单），抱怨他们有一台数据库服务器“磁盘缓慢”。

Scott 首要的任务是多了解问题的情况，收集信息形成完整的问题陈述。ticket 中抱怨磁盘慢，但是并没解释这是否是由数据库引发的。Scott 的回复问了以下这些问题：

- 当前是否存在数据库性能问题？如何度量它？
- 问题出现至今多长时间了？
- 最近数据库有任何变动吗？
- 为什么怀疑是磁盘？

数据库团队回复：“我们的日志显示有查询的延时超过了 1000ms，这并不常见，就在过去的一周这类查询的数目达到了每小时几十个。AcmeMon 显示磁盘在那段时间很繁忙。”

可以肯定确实存在数据库的问题，但是也可以看出关于磁盘的问题更多的是一种猜测。Scott 需要检查磁盘，同时他也要快速地检查一下其他资源，以免这个猜测是错误的。

AcmeMon 是公司服务器的基础监控系统，基于 `mpstat(1)`、`iostat(1)` 等其他的系统工具，提供性能的历史图表。Scott 登录到 AcmeMon 上自己查看问题。

一开始，Scott 使用了一种叫做 USE 的方法来快速检查系统瓶颈。正如数据库团队所报告的一样，磁盘的使用率很高，在 80% 左右，同时其他资源（CPU、网络）的使用率却低得多。历史数据显示磁盘的使用率在过去的一周内稳步上升，而 CPU 的使用率则持平。AcmeMon 不提供磁盘饱和（或错误）的统计数据，所以为了使用 USE 方法，Scott 必须登录到服务器上并运行几条命令。

他在 `/proc` 目录里检查磁盘错误数，显示是零。他以一秒钟作为间隔运行 `iostat`，对使用率和饱和率观察了一段时间。AcmeMon 报告 80% 的使用率是以一分钟作为间隔的。在一秒钟的粒度下，Scott 看到磁盘使用率在波动，并且常常达到 100%，造成了饱和，加大了磁盘 I/O 的延时。

为了进一步确定这是阻塞数据库的原因——延时相对于数据库的查询不是异步的——他利用动态跟踪脚本来捕捉时间戳和每次数据库被内核取消调度时数据库的栈跟踪。他发现数据库在查询过程中常常被一个文件系统读操作阻塞，一阻塞就是好几毫秒。对于 Scott 来说，这些证据已经足够了。

接下来的问题是为什么。磁盘性能统计显示负载持续很高。Scott 对负载进行了特征归纳以便做更多了解，使用 `iostat(1)` 来测量 IOPS、吞吐量、平均磁盘 I/O 延时和读写比。从这些结果，他计算出了平均 I/O 的大小并对访问模式做了估计：随机或者连续。Scott 可以通过 I/O 级别的跟踪来获得更多的信息，然而，他觉得这些已经足够表明这个问题是一个磁盘高负载的情况，而非磁盘本身的问题。

Scott 在 ticket 中添加了更多的信息，陈述了自己检查的内容并上传了检查磁盘所用到的命令截屏。他目前总结的结果是由于磁盘处于高负载状态，从而使得 I/O 延时增加，进而延缓了查询。但是，这些磁盘看起来对于这些负载工作得很正常。因此他问道，难道有一个更简单的解释：数据库的负载增加了？

数据库团队的回答是没有，并且数据库查询率（AcmeMon 并没有显示这个）始终是持平的。这看起来和最初的发现是一致的，CPU 的使用率也是持平的。

Scott 思考着还会有什么因素会导致磁盘的高 I/O 负载而又不引起 CPU 可见的使用率提升，他和同事简单讨论了一下这个问题。一个同事推测可能是文件系统碎片，碎片预计会在文件系统空间使用接近 100% 时出现。Scott 查了一下发现，磁盘空间使用率仅仅为 30%。

Scott 知道他可以进行更为深入的分析来了解磁盘 I/O 问题的根源，但这样做太耗时。基于自己对内核 I/O 栈的了解，他试图想出其他简单的分析，以此来快速做检查。他想到这次的磁盘 I/O 是由文件系统缓存（页缓存）未命中导致的。

Scott 进而检查了文件系统缓存的命中率，发现当前是 91%。这看起来还是很高的（很好），但是他没有历史数据可与之比较。他登录到其他有相似工作负载的数据库服务器上，发现它们的缓存命中率超过了 97%。他同时发现问题服务器上的文件系统缓存大小要比其他服务器大得多。

于是他把注意力转移到了文件系统缓存大小和服务内存使用情况上，发现了一些之前忽视的事情：一个开发项目的原型应用程序不断地消耗内存，虽然它并不处于生产负载之下。这些被占用的内存原本可以用作文件系统缓存，这使得缓存命中率降低，让磁盘 I/O 负载升高，损害了生产数据库服务器的性能。

Scott 联系了应用程序开发团队，让他们关闭该应用程序，并将其放到另一台服务器上，作为数据库问题的参照。随后在 AcmeMon 上 Scott 看到了磁盘使用率的缓慢下降，同时文件系统缓存恢复到了它原先的水平。被拖慢的数据库查询数目变成了零，他关闭了 ticket 并将它置为“已解决”。

1.9.2 软件变更

Pamela 在一家小公司做性能扩展工程师，负责所有与性能相关的事务。应用程序开发人员开发了一个新的核心功能，但是他们不确定引入这个功能会不会影响性能。在部署到生产环境

之前, Pamela 决定对这个应用程序的新版本执行一次非回归性测试。(非回归性测试是用来确认软件或硬件的变更并没有让性能倒退的。)

为了这个测试, Pamela 需要一台空闲的服务器和一台客户负载的模拟器。应用程序团队之前写过一个模拟器, 虽然这个模拟器还有诸多限制和一些已知的 bug, 她还是决定一试, 但要确定它能够充分地模拟生产环境的工作负载。

她依照生产环境的配置设置好服务器, 从另一个系统向目标开启客户端工作负载模拟器。客户工作负载可以通过研究访问日志来进行分析, 不过公司里已经有一个工具在做这件事情, 她直接就用了。她还用这个工具来分析同一天不同时间段的生产环境日志, 这样来对两个工作负载进行比较。她发现客户负载模拟器虽然可以提供一般性的生产环境工作负载, 但是对负载的多样性能为力。Pamela 记下了这一点后继续她的分析。

这时, Pamela 知道有很多方法可以用。她选择了最简单的那个: 增加客户模拟器的负载直至达到一个极限。客户负载模拟器可以设定每秒钟执行的客户数目, 其默认值是 1000, 她之前使用的就是这个值。Pamela 决定从 100 客户请求开始, 以每次 100 为增量逐步增加负载, 直至达到极限, 每一个测试级别都测试一分钟。她写了一个 shell 脚本来执行这个测试, 将结果收集到一个文件里供其他工具绘图。

随着负载不断增加, 她通过执行动态基准测试来判定限制因素。服务器资源和服务器的线程看起来有大量空闲。客户模拟器显示完成的请求数稳定在大约每秒 700 客户请求。

她切换到了新的软件版本并重复相同的测试。这次也是到了 700 客户请求就稳定不动了。她分析了服务器, 试图寻找限制的原因, 但是一无所获。

她把结果绘成图, 画出了请求完成率相对于负载的变化情况, 以此来观察不同软件版本的扩展特性。新旧两个软件版本都有一个很突兀的上限。

虽然看起来两个软件版本所拥有的性能特性是相似的, 但 Pamela 还是很失望, 因为她找不到是什么因素制约客户数的扩展。她知道她检查的只有服务器资源, 限制的原因可能出在应用程序的逻辑上, 也可能是其他地方: 网络或者客户模拟器上。

Pamela 想知道是不是需要采取一种不同的方法, 例如, 跑一个固定量的操作, 然后记录资源使用的汇总情况 (CPU、磁盘 I/O、网络 I/O), 这样就可以表示出单一客户请求的资源使用量。她针对当前的和新的软件版本, 按照每秒 700 客户的请求量来运行客户模拟器, 并测量了资源的消耗情况。当前的软件版本对于给定负载, 跑在 32 个 CPU 上的使用率达到了 20%。新的软件版本对于同样的负载, 在相同 CPU 数目上则是 30% 的使用率。看得出这确实是一个性能倒退, 占用了更多的 CPU 资源。

为了理解 700 的上限, Pamela 运行了一个更高的负载并研究了在数据路径上的所有组件, 包括网络、客户系统和客户工作负载产生器。她还对服务端和客户端软件做了向下挖掘分析。

她把所做的检查都做了记录，包括屏幕截图，以作为参考。

为了研究客户端软件，她执行了线程状态分析，发现这是一个单线程的软件。单线程 100% 的执行时间都花在了一个 CPU 上。这使得她确认这就是测试的限制因素所在。

作为验证实验，她在不同客户系统上并行运行客户端软件。用这种方式，无论当前版本软件还是新版本的软件，她都让服务器达到了 100% 的 CPU 使用率。这样，当前版本达到了每秒 3500 请求，新版本则是每秒 2300 请求，这与之前资源消耗的发现是一致的。

Pamela 通知应用软件开发人员新的软件版本有性能倒退，她打算对 CPU 的使用做剖析来查找原因：看看是哪条代码路径导致的。她强调指出一般性的生产工作负载已被测试过了，但多样性的工作负载还未曾测过。她还发了一个 bug，说明客户工作负载生成器是单线程的，这会成为瓶颈的。

1.9.3 更多阅读

第 13 章提供了一个更为详尽的案例研究，记录了一个我如何解决特定云计算性能问题的故事。下一章将介绍性能分析方法，其余各章会讲述必要的知识背景和细节。



第2章

方法

在取得数据之前就把事情理论化是一个严重的错误。不理智的人扭曲事实来适应理论，而不是改变理论来适应事实。

——夏洛克·福尔摩斯“波西米亚丑闻” 柯南·道尔 著

面对一个性能不佳且复杂的系统环境时，首先需要知道的挑战就是从什么地方开始分析、收集什么样的数据，以及如何分析这些数据。正如我第1章中介绍的，性能问题可能出现在任何地方，包括软件、硬件，以及数据路径上所有组件。

方法对于复杂系统的性能分析是有帮助的，告诉你从哪里开始工作，用什么步骤来定位和分析性能问题。对于新手来说，方法告诉你从什么地方开始，并列举了如何继续下去的步骤。对于部分用户或专家，方法作为检查清单来使用，确保没有遗漏什么细节。对发现进行量化和确认的方法都包括在内，从而分辨出最紧要的性能问题。

本章包括以下三部分内容。

- **背景：**介绍术语、基本模型、关键性能概念，以及审视问题的视角。
- **方法：**讨论性能分析方法，即观察法和实验法；建模；容量规划。
- **指标：**介绍性能统计、监控和数据可视化。

本章介绍的方法大部分在之后的章节中会有更详尽的讨论，包括第5~10章的方法部分。

2.1 术语

下面是关于系统性能的一些关键术语。之后的章节中还会覆盖更多的术语，并会对其中的一部分在不同情况下进行讲解。

- **IOPS**：每秒发生的输入/输出操作的次数，是数据传输的一个度量方法。对于磁盘的读写，IOPS 指的是每秒读和写的次数。
- **吞吐量**：评价工作执行的速率，尤其是在数据传输方面，这个术语用于描述数据传输速度（字节/秒或比特/秒）。在某些情况下（如数据库），吞吐量指的是操作的速度（每秒操作数或每秒业务数）
- **响应时间**：一次操作完成的时间。包括用于等待和服务的时间，也包括用来返回结果的时间。
- **延时**：延时是描述操作里用来等待服务的时间。在某些情况下，它可以指的是整个操作时间，等同于响应时间。例子参见 2.3 节。
- **使用率**：对于服务所请求的资源，使用率描述在所给定的时间区间内资源的繁忙程度。对于存储资源来说，使用率指的就是所消耗的存储容量（例如，内存使用率）。
- **饱和度**：指的是某一资源无法满足服务的排队工作量。
- **瓶颈**：在系统性能里，瓶颈指的是限制系统性能的那个资源。分辨和移除系统瓶颈是系统性能的一项重要工作。
- **工作负载**：系统的输入或者是对系统所施加的负载叫做工作负载。对于数据库来说，工作负载就是客户端发出的数据库请求和命令。
- **缓存**：用于复制或者缓冲一定量数据的高速存储区域，目的是为了避免对较慢的存储层级的直接访问，从而提高性能。出于经济考虑，缓存区的容量要比更慢一级的存储容量要小。

如有需要，书后的术语汇编囊括了所有供参考的基本术语。

2.2 模型

下面的简易模型阐述了系统性能的一些基本原则。

2.2.1 受测系统

受测系统（SUT，system under test）的性能如图 2.1 所示。



图 2.1 受测系统

需要知道的很重要的一点是，扰动（perturbation）是会影响结果的，扰动包括定时执行的系统活动、系统的其他用户以及其他的工作负载。扰动的来源可能不是很清楚，需要细致的系统性能研究才能加以确定。在某些云环境中，这会变得尤其困难，从单个客户 SUT 的视角无法观察到物理宿主系统的其他活动（由其他租户引起的）。

现代环境的另一个困难是系统很可能由若干个网络化的组件组成，都用于处理输入工作负载，包括负载均衡、Web 服务器、数据库服务器、应用程序服务器，以及存储系统。映射这个环境可能有助于发现之前所忽视的扰动源。这个环境也可以模型化成排队系统，以用于分析研究。

2.2.2 排队系统

某些组件和资源可以模型化成排队系统。图 2.2 展示了一个简易的排队系统。

2.6 节介绍的排队理论会涵盖排队系统和排队系统网络的内容。

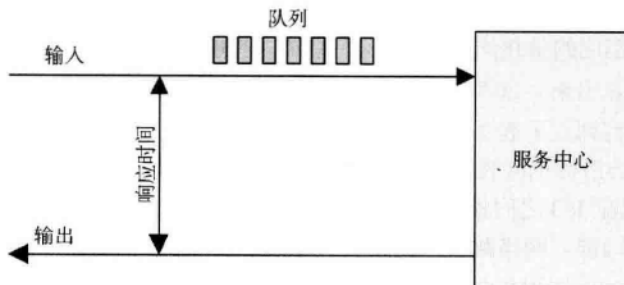


图 2.2 简易的排队模型

2.3 概念

下面是系统性能的一些重要概念，这些概念将贯穿本章的剩余内容以及本书始终。此处会用概括的方式进行讲解，详细的内容会出现在架构和后续各章的分析部分。

2.3.1 延时

对于某些环境，延时是唯一关注的性能焦点。而对于其他环境，它会是除了吞吐量以外，数一数二的分析要点。

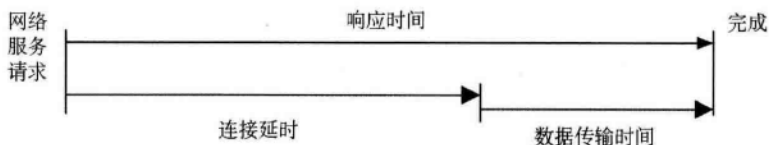


图 2.3 网络连接延时

延时是操作执行之前所花的等待时间。在这个例子里，所说的操作是网络服务的传输数据请求。在这个操作发生之前，系统必须等待网络连接建立，这即是这个操作的延时。响应时间包括了延时和操作时间。

因为延时可以在不同点测量，所以通常会指明延时测量的对象。例如，网站的载入时间由三个从不同点测得的不同时间组成：DNS 延时、TCP 连接延时和 TCP 数据传输时间。DNS 延时指的是整个 DNS 操作的时间。TCP 连接延时仅仅指的是初始化（TCP 握手）。

在一个更高的层级，所有这些，包括 TCP 数据传输时间，会被当作另外一种延时。例如，从用户点击网站链接起到网页完全载入都可以当作延时，其中包括了浏览器渲染生成网页的时间。

由于延时是一个时间上的指标，因此可能有多种计算方法。性能问题可以用延时来进行量化和评级，因为延时都用的是相同的单位来表达的（时间）。通过考量所能减少或移除的延时，预计的加速也可以计算出来。这两者都不能用 IOPS 指标很准确地描述出来。

时间的量级和缩写列在了表 2.1 中，作为参考。

如果可能，其他的指标也会转化为延时或者时间，这样就可以进行比较。如果你必须在 100 个网络 I/O 和 50 个磁盘 I/O 之间做出选择，你怎样才能知道哪个性能更好？这是一个复杂的选择，因为包含了很多因素：网络跳数、网络丢包率和重传、I/O 的大小、随机或顺序的 I/O、磁盘类型，等等。但是如果你比较的是 100ms 的网络 I/O 延时和 50ms 的磁盘 I/O 延时，差别就很明显了！

表 2.1 时间单位

单位	简写	与 1 秒的比例
分	m	60
秒	s	1
毫秒	ms	0.001 或 $1/1000$ 或 1×10^{-3}
微秒	μ s	0.000001 或 $1/1000000$ 或 1×10^{-6}
纳秒	ns	0.000000001 或 $1/1000000000$ 或 1×10^{-9}
皮秒	ps	0.000000000001 或 $1/1000000000000$ 或 1×10^{-12}

2.3.2 时间量级

我们可以用数字来作为时间的比较方法，同时可以用时间的长短经验来判断延时的源头。系统各组件的操作所处的时间量级差别巨大，大到了难以体会的地步。表 2.2 提供的延时示例，从访问 3.3GHz 的 CPU 寄存器的延时开始，阐释了我们所打交道的系统的时间量级的差别，表中是发生单次操作的时间均值，等比放大成为想象的系统，一次寄存器访问 0.3ns（十亿分之一秒的三分之一）相当于现实生活中的 1 秒。

表 2.2 系统的各种延时

事件	延时	相对时间比例
1 个 CPU 周期	0.3 ns	1 s
L1 缓存访问	0.9 ns	3 s
L2 缓存访问	2.8 ns	9 s
L3 缓存访问	12.9 ns	43 s
主存访问（从 CPU 访问 DRAM）	120 ns	6 分
固态硬盘 I/O（闪存）	50–150 μ s	2–6 天
旋转磁盘 I/O	1–10 ms	1–12 月
互联网：从旧金山到纽约	40 ms	4 年
互联网：从旧金山到英国	81 ms	8 年
互联网：从旧金山到澳大利亚	183 ms	19 年
TCP 包重传	1–3 s	105–317 年
OS 虚拟化系统重启	4 s	423 年
SCSI 命令超时	30 s	3 千年
硬件虚拟化系统重启	40 s	4 千年
物理系统重启	5 m	32 千年

正如你所见，CPU 周期的时间是很微小的。这段时间光只能走 0.5 米。很可能你眼睛到这一页书的距离，光大约走了 1.7ns。这段时间里，现代的 CPU 已经执行了 5 个 CPU 循环，处理了若干个指令。

关于 CPU 循环和延时的更多信息，参见第 6 章、第 9 章。因特网延时的内容在第 10 章中有更多的示例。

2.3.3 权衡三角

你应该知道某些性能权衡三角关系。图 2.4 展示的是好/快/便宜“择其二”的权衡三角，旁边是对应于 IT 项目的术语。



图 2.4 权衡三角：择其二

许多的 IT 项目选择了及时和便宜，留下了性能在以后的路上解决。当早期的决定阻碍了性能提高的可能性，这样的选择会变得有问题，例如，选择了非最优的存储架构，或者使用的编程语言或操作系统缺乏完善的性能分析工具。

一个常见的性能调整的权衡是在 CPU 与内存之间，因为内存能用于缓存数据结果，降低 CPU 的使用。在有着充足 CPU 资源的现代系统里，交换可以反向进行：CPU 可以压缩数据来降低内存的使用。

与权衡相伴而来的通常是可调参数。下面是一些例子。

- 文件系统记录尺寸（或块的大小）：小的记录尺寸，接近应用程序 I/O 大小，随机 I/O 工作负载会有更好的性能，程序运行的时候能更充分地利用文件系统的缓存。选择大的记录尺寸能提高流的工作负载性能，包括文件系统的备份。
- 网络缓存尺寸：小的网络缓存尺寸会减小每一个连接的内存开销，有利于系统扩展，大的尺寸能提高网络的吞吐量。

当对系统做调整的时候，寻找这类权衡。

2.3.4 调整的影响

性能调整发生在越靠近工作执行的地方效果最显著。对于工作负载驱动的应用程序，所执行的工作就是应用程序本身。表 2.3 展示了一个软件栈的例子，说明了性能调整的各种可能。

应用程序层级的调整，可能通过消去或减少数据库查询获得很大的性能提升（例如，20 倍）。在存储设备层级做调整，也可以精简或提高存储 I/O，但是性能提升的大头是在更高层级的系统栈代码，所以存储设备层级的调整能够达到的应用程序性能提升有限，是百分比量级（例如，20%）。

表 2.3 调整示例

层级	调优对象
应用程序	执行的数据库请求
数据库	数据库表布局、索引、缓冲
系统调用	内存映射、读写、同步或异步 I/O 标志
文件系统	记录尺寸、缓存尺寸、文件系统可调参数
存储	RAID 级别、磁盘类型和数目、存储可调参数

在应用程序层级寻求性能的巨大提升，还有一个理由。如今许多环境都致力于特性和功能的快速部署，因此，应用程序的开发和测试倾向于关注正确性，留给性能测量和优化的时间很少甚至没有。之后当性能成为问题时，才会去做这些与性能相关的事情。

虽然发生在应用程序层级的调整效果最显著，但这个层级不一定是观测效果最显著的层级。数据库查询缓慢要从其所花费的 CPU 时间、文件系统和所执行的磁盘 I/O 方面来考察最好。使用操作系统工具，这些都是可以观测到的。

在许多环境里（尤其是云计算），应用程序承受的是快速部署，每周或每天都有软件变更发生在生产环境。大的性能增长点（包括修复回归问题）和软件变化一样频繁地被发现。在这些系统里，操作系统的调整以及从操作系统层面观测问题这两点都容易被忽视。谨记一点操作系统的性能分析能辨别出来的不仅是操作系统层级的问题，还有应用程序层级的问题，在某些情况下，甚至要比应用程序视角还简单。

2.3.5 合适的层级

不同的公司和环境对于性能有着不同的需求。你可能加入过这样的公司，其分析标准要比你之前所见过的深得多，或者甚至可能听都没听过。或者是这样的公司，你觉得很基本的分析被认为很高端甚至从未使用过（这是好消息：事情简单轻松！）

这并不意味着某些公司做的是对的，某些做的是错的。这取决于性能技术投入的投资回报率（ROI）。拥有大型数据中心或者云环境的公司可能需要一个性能工程师团队来分析所有事情，包括内核内部和 CPU 性能计数器，并且经常使用动态跟踪技术（dynamic tracing）。这些中心或公司也会正式建立性能模型对将来的增长做预测。小的创业公司可能只能做到浅层次的检查，用第三方的监控方案来检测性能和提供报警。

最极端的环境，像股票交易所和高交易率的电商，性能和延时对它们很关键，值得投入大量的人力和财力。举一个例子，一条新的横跨大西洋的连接纽约交易所和伦敦交易所的光缆正在规划之中，花费预计 3 亿美金，用以减少 6ms 的传输延时[1]。

2.3.6 性能建议的时间点

环境的性能特性会随着时间改变，更多的用户、新的硬件、升级的软件或固件都是变化的因素。一个环境，受限速度 1Gb/s 网络基础设施，当升级到 10Gb/s 时，很可能会发现磁盘或 CPU 的性能变得紧张。

性能推荐，尤其是可调整的参数值，仅仅在一段特定时间内有效。一周内从性能专家那里得到的好建议，可能到了下一周，经过一次软件或硬件升级，或者用户增多后就无效了。

通过网上搜索找到的可调参数值对于某些情况能快速见效。如果它们对于你的系统或者工作负载并不合适，它们也会损害性能，可能合适过一次，就不合适了，或者只是作为软件的某个 bug 修复升级之前暂时的应急措施。这和从别人的医药箱里拿药吃很像，那些药可能不适合你，或者可能已经过期了，或者只适合短期服用。

如果仅仅是出于要了解有哪些参数可调以及哪些参数在过去是需要调整的，那么浏览这些性能推荐是有用的。针对你的系统和工作负载，这项工作就变成了考虑这些参数是不是要调，以及调整成什么值。如果其他人不需要调整那个值，或者调整了但并未将经验分享出来，那么你是可能漏掉重要参数的。

2.3.7 负载 vs. 架构

应用程序性能差可能是因为软件配置和硬件的问题，也就是它的架构问题。另外，应用程序性能差还可能是由于有太多负载，而导致了排队和长延时。负载和架构见图 2.5。

如果对于结构的分析只是显示工作任务的排队，处理任务没有任何问题，那么问题就可能出在施加的负载太多。在云计算环境里，这是需要引入更多的节点来处理任务的征兆。

举个例子，架构的问题可能是一个单线程的应用程序在单个 CPU 上忙碌，从而致使请求排队的情况，但是其他的 CPU 却是可用且空闲的。在这个例子里，性能就被应用程序的单一线程

架构限制住了。

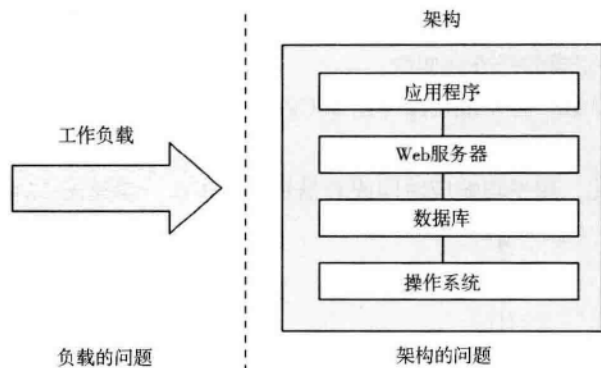


图 2.5 负载 vs. 架构

负载的问题可能会是一个多线程程序在所有的 CPU 上都忙碌，但是请求依然排队的情况。在这个例子里，性能可能限制于 CPU 的性能，或者说是，负载超出了 CPU 所能处理的范围。

2.3.8 扩展性

负载增加下的系统所展现的性能成为扩展性。图 2.6 是一个典型的系统负载增加下的吞吐量变化曲线。

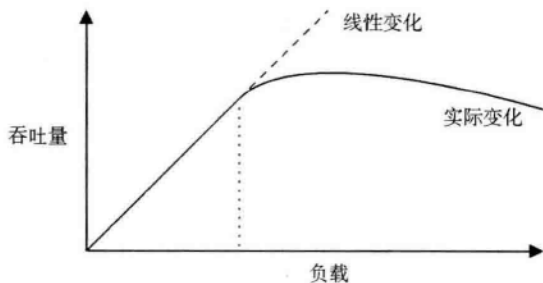


图 2.6 吞吐量 vs. 负载

在一定阶段，可以观察到扩展性是线性变化的。随着到达某一点，用虚线标记，此处对于资源的争夺开始影响性能。这一点可以认为是拐点，作为两条曲线的分界。过了这一点后，吞吐量曲线随着资源争夺加剧偏离了线性扩展，内聚性导致完成的工作变少而且吞吐量也减小了。

这个情况可能发生在组件达到 100% 使用率的时候——饱和点。也可能发生在组件接近 100% 使用率的时候，这时排队频繁且比较明显。

一个可以用于说明这种情况的系统示例是执行大量计算的应用程序，更多的负载由线程承

担。当 CPU 接近 100% 使用率时，由于 CPU 调度延时增加，性能开始下降。在性能达到峰值后，在 100% 使用率时，吞吐量已经开始随着更多线程的增加而下降，导致更多的上下文切换，这会消耗 CPU 资源，实际完成的任务会变少。

如果把 X 轴的“负载”替换成资源（诸如 CPU），你会看见一样的曲线。关于这一点更多的内容见 2.6 节。

性能的非线性变化，用平均响应时间或者是延时来表示，参见图 2.7[Cockcroft 95]。

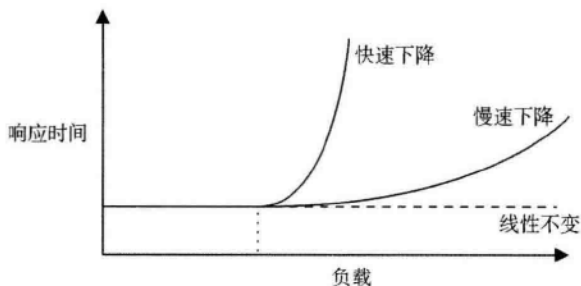


图 2.7 性能下降

当然，过长的响应时间不是好事情。发生性能“快速”下降可能是由于内存的负载，就是当系统开始换页（或者使用 swap）来补充内存的时候。发生性能“慢速”下降则可能是由于 CPU 的负载。

还有一个“快速”性能下降的情况是磁盘 I/O。随着负载（和磁盘使用率）的增加，I/O 可能会排队。空闲的旋转的磁盘可能 I/O 服务响应时间约 1ms，但是当负载增加，响应时间会变成 10ms。这种情况在 2.6.5 节中有建模，在 M/D/1 和 60% 使用率的情况下。

如果资源不可用，应用程序开始返回错误，响应时间是直线变化的，而不是将工作任务排队。举个例子，Web 服务器很可能会返回 503 “Service Unavailable” 而不是添加请求到排队队列中，这样服务的请求能用始终如一的响应时间来执行。

2.3.9 已知的未知

我们在前言中介绍过，已知的已知、已知的未知、未知的未知在性能领域是很重要的概念。下面是详细的解释，并有系统性能分析例子。

- **已知的已知**：有些东西你知道。你知道你应该检查性能指标，你也知道它的当前值。举个例子，你知道你应该检查 CPU 使用率，而且你也知道当前均值是 10%。
- **已知的未知**：有些东西你知道你不知道。你知道你可以检查一个指标或者判断一个子系统是否存在，但是你还没去做。举个例子，你知道你能用 profiling 检查是什么致使

CPU 忙碌，但你还没去做这件事。

- **未知的未知：**有些东西你不知道你不知道。举个例子，你可能不知道设备中断可以消耗大量 CPU 资源，因此你对此并不做检查。

性能这块领域是“你知道的越多，你不知道的也就越多”。这和学习系统是一样的原理：你了解的越多，你就能意识到未知的未知就越多，然后这些未知的未知会变成你可以去查看的已知的未知。

2.3.10 指标

性能指标是由系统、应用程序，或者其他工具产生的度量感兴趣活动的统计数据。性能指标用于性能分析和监控，由命令行提供数据或者由可视化工具提供图表。

常见的系统性能指标如下。

- **IOPS：**每秒的 I/O 操作数。
- **吞吐量：**每秒数据量或操作量。
- **使用率**
- **延时**

吞吐量的使用取决于上下文环境。数据库吞吐量通常用来度量每秒查询或请求的数目（操作量）。网络吞吐量度量的是每秒比特或字节数目（数据量）。

IOPS 度量的是吞吐量，但只针对 I/O 操作（读取和写入）。再次重申，上下文很关键，上下文不同，定义可能会有不同。

开销

性能指标不是免费的，在某些时候，会消耗一些 CPU 周期来收集和保存指标信息。这就是开销，对目标的性能会有负面影响。这种影响被称为观察者效应（observer effect）。（这通常与海森堡测不准原理混淆，后者描述的是对于互为共轭的物理量所能测量出的精度是有限的，诸如位置和动量。）

问题

我们总是倾向于假定软件商提供的指标是经过仔细挑选，没有 bug，并且有很好的可见性的。但事实上，指标可能会是混淆的、复杂的、不可靠的、不精确的，甚至是错的（由 bug 所致）。有时在某一软件版本上对的指标，由于没有得到及时更新，而无法反映新的代码和代码路径。

关于指标更多的问题，参见第 4 章内容。

2.3.11 使用率

术语“使用率”经常用于操作系统描述设备的使用情况，诸如 CPU 和磁盘设备。使用率是基于时间的，或者是基于容量的。

基于时间

基于时间的使用率是使用排队理论做正式定义的。例如[Gunther 97]：

服务器或资源繁忙时间的均值

相应的比例公式是：

$$U = B/T$$

此处 U 是使用率， B 是 T 时间内系统的繁忙时间， T 是观测周期。

操作系统性能工具得到的“使用率”也是这个。磁盘监控工具 `iostat(1)` 调用的指标 `%b`，即忙碌百分比，这个术语更好地诠释了指标 B/T 的本质。

这个使用率指标告诉我们组件的忙碌程度：当一个组件达到 100% 使用率，资源发生竞争时性能会有严重的下降。这时可以检查其他的指标以确认该组件是不是已经成为了系统的瓶颈。

某些组件能够并行地为操作提供服务。对于这些组件，在 100% 使用率的情况下，性能可能不会下降得太大，因为它们能接受更多的工作。要理解这一点，可以大楼电梯为例思考一下，当电梯在楼层间移动时，它是被使用的，当它闲置等待的时候，它是不用的。然而，即便当它在 100% 忙碌的时候，即达到 100% 使用率的时候，它依然还是能够接受更多的乘客的。

100% 忙碌的磁盘也能够接受并处理更多的工作，例如，通过把写入的数据放入磁盘内部的缓存中，稍后再完成写入，就能做到这点。通常存储序列运行在 100% 的使用率是因为其中的某些磁盘在 100% 忙碌时，序列中依然有足够的空闲磁盘来接受更多的工作。

基于容量

使用率的另一个定义是在容量规划中由 IT 专业人员使用的[Wong 97]：

系统或组件（例如硬盘）都能够提供一定数量的吞吐。不论性能处于何种级别，系统或组件都工作在其容量的某一比例上。这个比例就称为使用率。

这是用容量而不是时间来定义使用率的。这意味着 100% 使用率的磁盘不能接受更多的工作。若用时间定义，100% 的使用率只是指时间上 100% 的忙碌。

100% 忙碌不意味着 100% 的容量使用。

回到电梯的例子, 100%容量意味着电梯满载, 装不下更多的乘客了。

在理想的世界里, 我们应该对设备做两种使用率的测试, 这样做, 你就能知道何时磁盘 100% 忙碌, 性能开始下降, 也能知道何时达到 100% 的容量, 磁盘无法再接受更多的工作。不幸的是, 这通常是不可能的。对于磁盘而言, 这不仅需要了解主板的磁盘控制器的行为, 还要对磁盘的容量使用有预测。目前, 磁盘并不提供这一信息。

本书中, 使用率通常指的是基于时间的定义。基于容量的定义会用于某些基于容量的指标, 诸如内存使用。

非空闲时间

定义使用率的问题是我们公司开发一个云监控项目时碰到的。首席工程师 Dave Pacheco 问我如何定义使用率, 我做了上述的回答。由于对可能造成的混淆不满, 他创造了一个新的术语来让事情不言自明: 非空闲时间。

虽然这更加精确, 但是现在用得还并不普遍 (参照对比之前说的指标, 忙碌百分比)。

2.3.12 饱和度

随着工作量增加而对资源的请求超过资源所能处理的程度叫做饱和度。饱和度发生在 100% 使用率时 (基于容量), 这时多出的工作无法处理, 开始排队。图 2.8 描绘了这种情况。

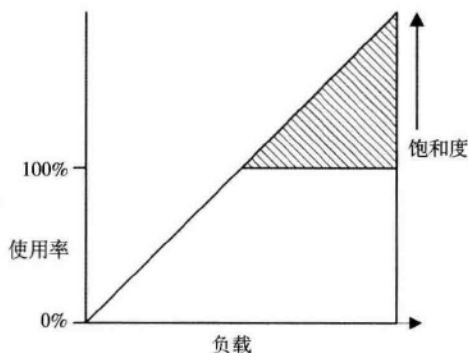


图 2.8 使用率 vs. 饱和度

随着负载的持续上升, 图中的饱和度在超过基于容量的使用率 100% 的标记后线性增长。因为时间花在了等待 (延时) 上, 所以任何程度的饱和度都是性能问题。对于基于时间的使用率 (忙碌百分比), 排队和饱和度可能不发生在 100% 使用率时, 这取决于资源处理任务的并行能力。

2.3.13 剖析

剖析（profiling）本意指对目标对象绘图以用于研究和理解。在计算机性能领域，profiling 通常是按照特定的时间间隔对系统的状态进行采样，然后对这些样本进行研究。

不像之前讲过的指标（包括 IOPS 和吞吐量），采样所能提供的对系统活动的观察比较粗糙，当然这也取决于采样率的大小。

举一个 profiling 的例子，对 CPU 程序计数器采样，以一定频率间隔进行栈回溯跟踪来收集消耗 CPU 资源的代码路径的统计数据，通过这样的方式我们才能了解 CPU 的使用。这个专题可以参见第 6 章。

2.3.14 缓存

缓存被频繁使用来提高性能。缓存是将较慢的存储层的结果存放在较快的存储层中。把磁盘的块缓存在主内存（RAM）中就是一例。

一般使用的都是多层缓存。CPU 通常利用多层的硬件作为主缓存（L1、L2 和 L3），开始是一个非常快但是很小的缓存（L1），后续的 L2 和 L3 就逐渐增加了缓存容量和访问延时。这是一个在密度和延时之间经济上的权衡。缓存的级数和大小的选择按 CPU 芯片内可用空间为准以达到最优的性能。

在系统中还有不少其他的缓存，许多都是利用主内存做存储软件实现的。参见第 3 章的 3.2.11 节，那里有一张系统缓存的列表。

一个了解缓存性能的重要指标是每个缓存的命中率——所需数据在缓存中找到的次数（hits，命中）相比于没有找到的次数（misses，失效）。

$$\text{命中率} = \text{命中次数} / (\text{命中次数} + \text{失效次数})$$

命中率越高越好，更高的命中率意味着更多的数据能成功地从较快的介质中访问获得。图 2.9 表示的是随缓存命中率上升，性能预期的提升曲线。

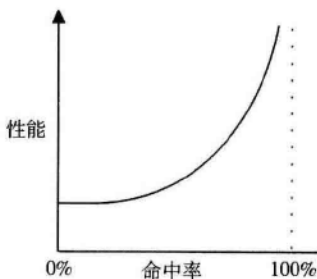


图 2.9 缓存命中率和性能

98%和 99%之间的性能差异要比 10%和 11%之间的性能差异大很多。由于缓存命中和失效之间的速度差异（两个存储层级），导致了这是一条非线性曲线。两个存储层级速度差异越大，曲线倾斜越陡峭。

了解缓存性能的另一个指标是缓存的失效率，指的是每秒钟缓存失效的次数。这与每次缓存失效对性能的影响是成比例（线性）关系的，比较容易理解。

举个例子，工作负载 A 和 B 执行相同的任务，但用的是不同的算法，都用内存作为缓存以避免直接从磁盘读取数据。工作负载 A 的命中率为 90%，工作负载 B 的命中率是 80%。单独分析这个信息会觉得工作负载 A 执行得更好。但如果 A 的失效率是 200/s，B 的失效率是 20/s 呢？这样看的话，B 执行的磁盘读取次数比 A 少 10 倍，这会使得 B 完成任务的时间远远少于 A。可以确定的是，工作负载总的运行时间可以用以下公式计算：

$$\text{运行时间} = (\text{命中率} \times \text{命中延时}) + (\text{失效率} \times \text{失效延时})$$

这里用的是平均命中延时和平均失效延时，并且假定工作是串行发生的。

算法

缓存管理算法和策略决定了在有限的缓存空间内存放哪些数据。

“最近最常使用算法”（MRU）指的是一种缓存保留策略，决定什么样的数据会保留在缓存里：最近使用次数最多的数据。“最近最少使用算法”（LRU）指的是一种回收策略，当需要更多缓存空间的时候，决定什么数据需要移出缓存。此外，还有“最常使用算法”（MFU）和“最不常使用算法”（LFU）。

你可能碰到过“不常使用算法”（NFU），这个是 LRU 的一个花费不高但吞吐量稍小的版本。

缓存的热、冷和温

下面这些词通常用来表达缓存的状态。

- **冷**：冷缓存是空的，或者填充的是无用的数据。冷缓存的命中率为 0（或者接近 0，当它开始变暖的时候）。
- **热**：热缓存填充的都是常用的数据，并有着很高的命中率，例如，超过 99%。
- **温**：温缓存指的是填充了有用的数据，但是命中率还没达到预想的高度。
- **热度**：缓存的热度指的是缓存的热度或冷度。提高缓存热度的目的就是提高缓存的命中率。

当缓存初始化后，开始是冷的，过一段时间后逐渐变暖。如果缓存较大或者下一级的存储较慢（或者两者皆有），会需要一段较长的时间来填充缓存使其变暖。

例如，我工作过的一个存储服务器，有 128GB 的 DRAM 作为文件系统的缓存，600GB 的闪存作为二级缓存，物理磁盘作为存储。在随机读取的工作负载下，磁盘每秒的读操作约有 2000 次。按照 8KB 的 I/O 大小，这意味着缓存的变暖速度仅有 16MB/s（ $2000 \times 8\text{KB}$ ）。两级缓存从冷开始，需要 2 小时来让 DRAM 缓存变得温起来，需要超过 10 小时来让闪存缓存变暖。

2.4 视角

性能分析有两个常用的视角，每个视角的受众、指标以及方法都不一样。这两个视角是工作负载分析和资源分析，可以分别对应理解为对系统软件栈自上而下和自底向上的分析，如图 2.10 所示。

2.5 节会论述实施的策略，并对这两种视角做更详尽的论述。

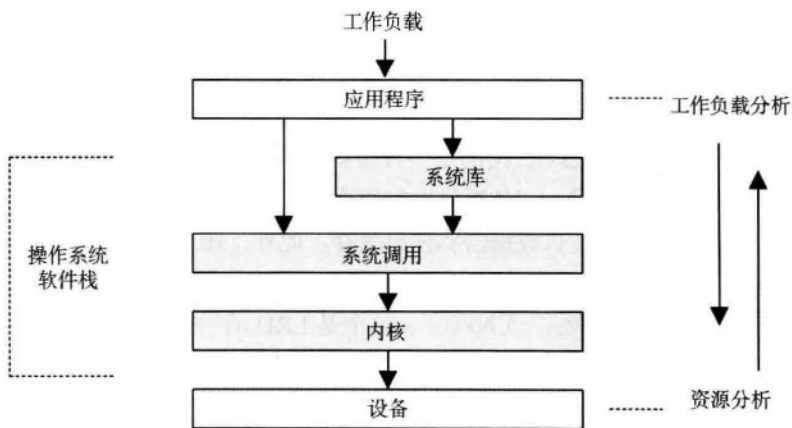


图 2.10 分析视角

2.4.1 资源分析

资源分析以对系统资源的分析为起点，涉及的系统资源有：CPU、内存、磁盘、网卡、总线以及之间的互联。执行资源分析的通常是系统管理员——他们负责管理物理环境资源。

操作如下。

- **性能问题研究：**看是否某特定类型资源的责任。
- **容量规划：**为设计新系统提供信息，或者对系统资源何时会耗尽做预测。

这个视角着重于使用率的分析，判断资源是否已经处于极限或者接近极限。对于某些资源类型，如 CPU，使用率的指标是即有的。其他资源的使用率也可以通过即有的指标来进行计算。举个例子，通过将每秒发出和接收的数据量（吞吐量）与已知的最大带宽做比较就可以估算出网卡的使用率。

适合资源分析的指标如下：

- IOPS
- 吞吐量
- 使用率
- 饱和度

这些指标度量了在给定负载下资源所做的事情，显示资源的使用程度乃至饱和的程度。其他类型的指标，包括延时，也会被使用，来度量资源对于给定工作负载的响应情况。

资源分析是性能分析的常用手段。关于这个主题有着广泛的文档，诸如针对操作系统的“统计”工具：vmstat(1)、iostat(1)、mpstat(1)。当你阅读这类文档时，要知道这是一种视角，但并非唯一的视角，这很重要。

2.4.2 工作负载分析

工作负载分析（见图 2.11）检查应用程序的性能：所施加的工作负载和应用程序是如何响应的。通常执行工作负载分析的是应用开发人员和技术支持人员——他们负责应用程序软件和配置。

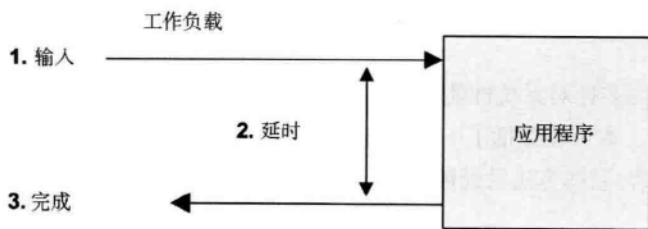


图 2.11 工作负载分析

工作负载分析的对象如下。

- **请求：**所施加的工作负载
- **延时：**应用程序的响应时间
- **完成度：**查找错误

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系QQ1779903665.

PDF代找说明：

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签索引和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我QQ1779903665。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

声明：本人只提供代找服务，每本100%索引书签和目录，因寻找pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，与本人无关，我们仅仅只是帮助你寻找到你要的pdf而已。