Broadview
www.broadview.com.cn

构建高性能Web站点

Building High Performance Web

改善性能和扩展规模的具体做法

The Way to Improve Performance and Scale Out

郭欣 著

Smart Developer

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

构建高性能Web站点
Building High Performance Web

本书涵盖主题：

- 数据的网络传输
- 动态内容缓存
- 浏览器缓存
- 反向代理缓存
- 分布式缓存
- Web负载均衡
- 内容分发和同步
- 数据库扩展
- 性能监控
- 服务器并发处理能力
- 动态脚本加速
- Web服务器缓存
- Web组件分离
- 数据库性能优化
- 共享文件系统
- 分布式文件系统
- 分布式计算

《构建高性能Web站点》是作者在Web系统领域多年工作、实践和探索的结晶。本书涉及了Web系统优化的各个方面，从浏览器、Cache到Web、数据库和分布式文件系统等；穿插了大量的实际测试数据和很多流行开源软件的使用方法与案例；内容丰富，文字生动，对比形象。对于网络系统架构师、运维和开发人员，这是很好的参考书目；对于想了解Web性能并希望动手实践的人员，这是由浅入深的学习书籍。

——章文嵩博士，LVS作者，Linux内核作者之一

本书深入分析了常见的高性能Web技术的方法和原理，对搭建高性能Web站点具备很强的可操作性。

——张松国，腾讯网技术总监

这是一个令人兴奋的领域，这一系列准则和方法在TopN的互联网公司中都有大规模的实践和应用，作者在书中进行了详细而量化的论述。如果你正在为日益庞大的应用而手足无措，那么你唯一要做的就是拥有这本书，并且实践它。

——朱鑫，MemcacheDB作者
新浪网研发中心平台部高级工程师

互联网寄托着我们的梦想，它改变了人们的生活，从社交网站到网络游戏，从搜索引擎到电子商务，成功的秘诀在于如何构建高性能Web站点。郭欣在这本书中几乎涵盖了Web性能优化的所有内容，并从多个角度进行了全面的阐述，你可以通过其通俗易懂的文字，深入理解高性能站点架构的真相并开拓视野，从而对性能瓶颈对症下药。本书可谓是高性能站点的必读精作。

——沈翔，Google Developer Advocate，加州总部

上架建议：Web开发

ISBN 978-7-121-09335-7
9 787121 093357 >
定价：59.00元

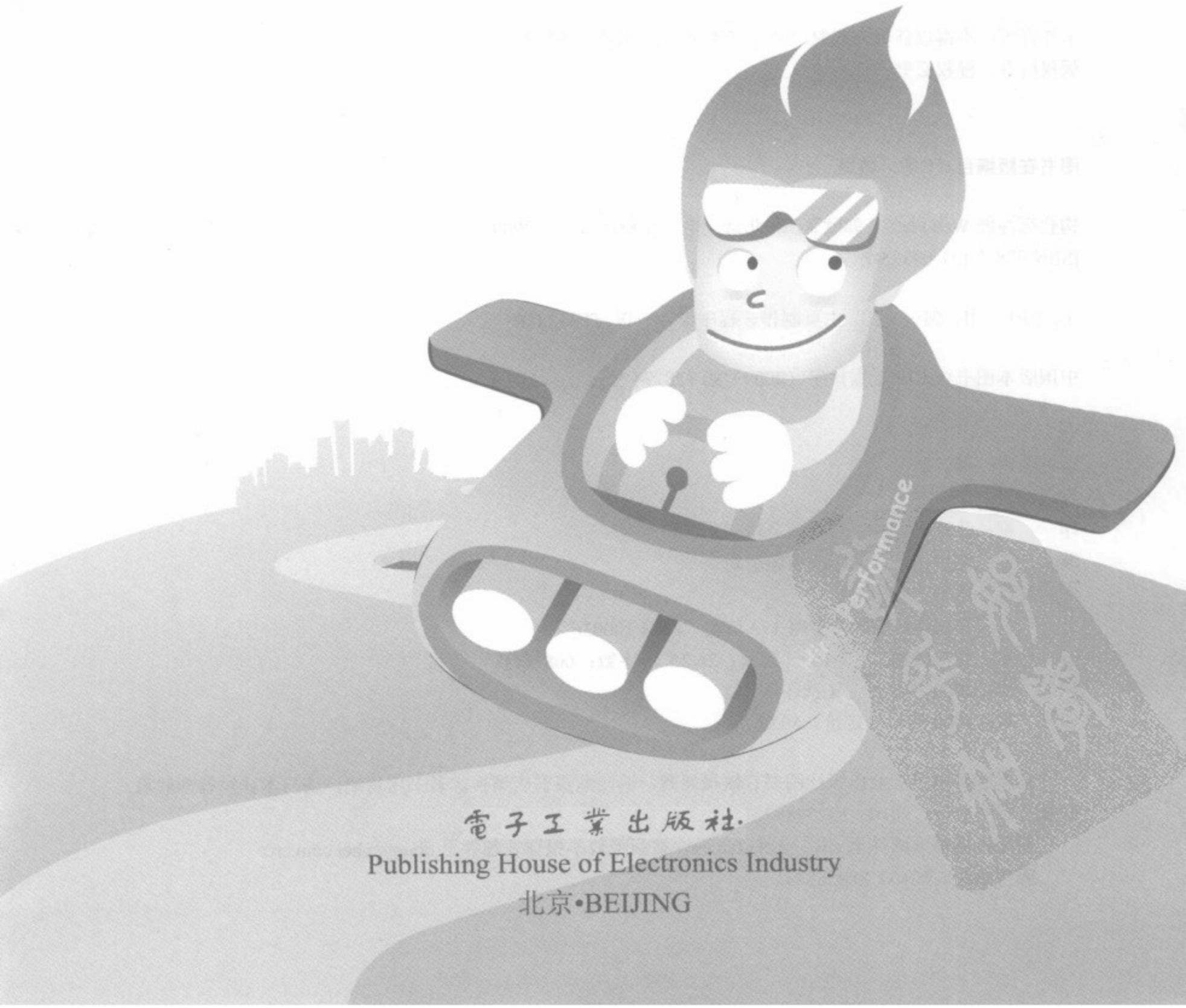
策划编辑：李　冰
责任编辑：江　立

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

构建高性能Web站点

Building High Performance Web

郭欣 著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书围绕如何构建高性能 Web 站点，从多个方面、多个角度进行了全面的阐述，涵盖了 Web 站点性能优化的几乎所有内容，包括数据的网络传输、服务器并发处理能力、动态网页缓存、动态网页静态化、应用层数据缓存、分布式缓存、Web 服务器缓存、反向代理缓存、脚本解释速度、页面组件分离、浏览器本地缓存、浏览器并发请求、文件的分发、数据库 I/O 优化、数据库访问、数据库分布式设计、负载均衡、分布式文件系统、性能监控等。在这些内容中充分抓住本质并结合实践，通过通俗易懂的文字和生动有趣的配图，让读者充分并深入理解高性能架构的真相。同时，本书充分应用跨学科知识和科学分析方法，通过宽泛的视野和独特的角度，将本书的内容展现得更加透彻和富有趣味。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

构建高性能 Web 站点 / 郭欣著. —北京：电子工业出版社，2009.8
ISBN 978-7-121-09335-7

Ⅰ. 构… Ⅱ. 郭… Ⅲ. 主页制作－程序设计 Ⅳ. TP393.092

中国版本图书馆 CIP 数据核字（2009）第 128794 号

策划编辑：李 冰
责任编辑：江 立
印 刷：北京智力达印刷有限公司
装 订：北京中新伟业印刷有限公司
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：26.25 字数：608 千字
印 次：2009 年 8 月第 1 次印刷
印 数：4000 册 定价：59.00 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 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

谨以此书

献给我亲爱的父母和 Cris

PREFACE

推荐序

你很幸运能拿到这本书，更重要的是，你的网站用户也会很幸运。郭欣在这本书里深入而系统地分享了构建高性能网站技术的方方面面。从后台到前台，从网络传输到数据存储，涉及诸多技术原理和实现细节。通俗的语言，亲切的叙述，仿佛作者在耳边轻轻细语，然而又蕴含着一种技术思想和力量，并且融合了人文思想。

我曾经代表公司面试过许多开发人员，在问及与高性能相关的问题时，大家都能回答出需要负载均衡，需要缓存技术，然而当我进一步询问负载均衡如何实现或如何有效控制缓存命中率时，面试者却无从答起。知其然而不知其所以然是很多浮躁开发者的通病，也因此限制了其技术能力的提升和发展。

这本书将为你提供构建高性能网站的完整解决方案，它会成为每个致力于开发承载百万级用户规模网站开发者的工具箱。郭欣有着架构和开发多个大规模网站的经验，他精通前/后台技术和架构。在知道他将花时间著作一本高性能网站架构的书时，我不禁为国内许多开发者感到高兴。我见过部分知名网站架构师曾经分享过他们网站技术发展的历程，但每每都是停留在抽象层面，而像本书这样全面彻底地进行技术剖析却是头一回。尤其是构建高性能网站的各种技术方案，绝大部分是通过实践总结出来的经验，没有这样的经历，你甚至很难想象为什么会是这样。

不要犹豫了！当你拿起这本书，按照书中所分享的技术方案去实践时，你会发现，原来构建高性能网站就这么简单。中国互联网正在不断地成长，用户规模也在不断地扩大，我相信，越来越多的网站会根据性能这项最基本的用户体验决定其自身的生存能力。本书所提供的技术解决方案，正是在这个发展趋势中的一个基础，拥有它并加以实践，你和你的用户都会更加享受这一切！

为了页面一秒响应的境界，开始阅读吧！

——王速瑜

腾讯 R&D 研发总监（Tencent Director of R&D）

资深技术专家（Senior Technology Expert）

深圳，2009 年 7 月

CONTENTS

目 录

第 1 章 绪论 ····· 1

1.1 等待的真相 ····· 1
1.2 瓶颈在哪里 ····· 2
1.3 增加带宽 ····· 3
1.4 减少网页中的 HTTP 请求 ····· 4
1.5 加快服务器脚本计算速度 ····· 4
1.6 使用动态内容缓存 ····· 5
1.7 使用数据缓存 ····· 5
1.8 将动态内容静态化 ····· 6
1.9 更换 Web 服务器软件 ····· 6
1.10 页面组件分离 ····· 7
1.11 合理部署服务器 ····· 7
1.12 使用负载均衡 ····· 8
1.13 优化数据库 ····· 8
1.14 考虑可扩展性 ····· 9
1.15 减少视觉等待 ····· 10

第 2 章 数据的网络传输 ····· 11

2.1 分层网络模型 ····· 11
2.2 带宽 ····· 22
2.3 响应时间 ····· 28
2.4 互联互通 ····· 33

第 3 章 服务器并发处理能力 ····· 35

3.1 吞吐率 ····· 35
3.2 CPU 并发计算 ····· 49
3.3 系统调用 ····· 60
3.4 内存分配 ····· 63
3.5 持久连接 ····· 65

3.6 I/O 模型 ····· 68
3.7 服务器并发策略 ····· 81

第 4 章 动态内容缓存 ····· 96

4.1 重复的开销 ····· 96
4.2 缓存与速度 ····· 98
4.3 页面缓存 ····· 98
4.4 局部无缓存 ····· 112
4.5 静态化内容 ····· 112

第 5 章 动态脚本加速 ····· 121

5.1 opcode 缓存 ····· 121
5.2 解释器扩展模块 ····· 132
5.3 脚本跟踪与分析 ····· 133

第 6 章 浏览器缓存 ····· 143

6.1 别忘了浏览器 ····· 143
6.2 缓存协商 ····· 147
6.3 彻底消灭请求 ····· 160

第 7 章 Web 服务器缓存 ····· 167

7.1 URL 映射 ····· 167
7.2 缓存响应内容 ····· 168
7.3 缓存文件描述符 ····· 175

第 8 章 反向代理缓存 ····· 178

8.1 传统代理 ····· 178
8.2 何为反向 ····· 179
8.3 在反向代理上创建缓存 ····· 180
8.4 小心穿过代理 ····· 202
8.5 流量分配 ····· 204

第 9 章 Web 组件分离 ····· 205

9.1 备受争议的分离 ····· 205
9.2 因材施教 ····· 206
9.3 拥有不同的域名 ····· 207

9.4 浏览器并发数 ····· 210
9.5 发挥各自的潜力 ····· 212

第 10 章 分布式缓存 ····· 220

10.1 数据库的前端缓存区 ····· 220
10.2 使用 memcached ····· 221
10.3 读操作缓存 ····· 225
10.4 写操作缓存 ····· 229
10.5 监控状态 ····· 232
10.6 缓存扩展 ····· 234

第 11 章 数据库性能优化 ····· 238

11.1 友好的状态报告 ····· 239
11.2 正确使用索引 ····· 241
11.3 锁定与等待 ····· 255
11.4 事务性表的性能 ····· 263
11.5 使用查询缓存 ····· 264
11.6 临时表 ····· 266
11.7 线程池 ····· 266
11.8 反范式化设计 ····· 267
11.9 放弃关系型数据库 ····· 269

第 12 章 Web 负载均衡 ····· 272

12.1 一些思考 ····· 272
12.2 HTTP 重定向 ····· 275
12.3 DNS 负载均衡 ····· 284
12.4 反向代理负载均衡 ····· 292
12.5 IP 负载均衡 ····· 305
12.6 直接路由 ····· 317
12.7 IP 隧道 ····· 325
12.8 考虑可用性 ····· 325

第 13 章 共享文件系统 ····· 328

13.1 网络共享 ····· 328
13.2 NFS ····· 330
13.3 局限性 ····· 335

第 14 章　内容分发和同步 …… 337
14.1　复制 …… 337
14.2　SSH …… 338
14.3　WebDAV …… 342
14.4　rsync …… 342
14.5　Hash tree …… 344
14.6　分发还是同步 …… 345
14.7　反向代理 …… 346
第 15 章　分布式文件系统 …… 348
15.1　文件系统 …… 348
15.2　存储节点和追踪器 …… 350
15.3　MogileFS …… 352
第 16 章　数据库扩展 …… 362
16.1　复制和分离 …… 362
16.2　垂直分区 …… 366
16.3　水平分区 …… 367
第 17 章　分布式计算 …… 374
17.1　异步计算 …… 374
17.2　并行计算 …… 379
第 18 章　性能监控 …… 384
18.1　实时监控 …… 384
18.2　监控代理 …… 386
18.3　系统监控 …… 388
18.4　服务监控 …… 391
18.5　响应时间监控 …… 393
参考文献 …… 397
索引 …… 399

第 1 章
绪论

一般而言，人们评估一个 Web 站点的性能如何，通常先置身于用户的角度，访问该站点的一系列页面，体验等待时间。

当用户输入页面地址后，浏览器获得了用户希望访问该地址的意图，便向站点服务器发起一系列的请求，请注意，这些请求不光包括对页面的请求，还包括对页面中许许多多组件的请求，比如图片、层叠样式表（CSS）、脚本（JavaScript）、内嵌页面（iframe）等。接下来的一段时间，浏览器等待服务器的响应以及返回的数据。待浏览器获得所有的返回数据后，经过本地的计算和渲染，最终一幅完整的页面才呈现于用户的眼前。

1.1 等待的真相

整个过程听起来好像并不复杂，也许你从来都没有考虑过在这段等待的时间里世界都发生了什么变化，也许你早已习惯了利用这段时间东张西望或者品尝零食，或者你根本没有来得及意识到这点，新的网页就已经闪亮登场，恭喜你，你很幸运！但是在这个世界上，幸运儿永远只占少数，大多数人的大脑处理速度已经让他们明显感觉到这段等待时间漫长无比，久经考验的他们可以随时身手敏捷地打开多个浏览器窗口与时间赛跑，并为此筋疲力尽。

另一方面，对于站点经营者来说，让用户等待的时间过长，也许会造成毁灭性的后果。我见过很多人为了享用某家特色小吃而在餐馆门口乐此不疲地排着长队，但没有听说有多少用户执著地等待着一个速度缓慢的站点而不去尝试别的站点。

在这段等待的时间里，到底发生了什么？事实上这并不简单，大概经历了以下几部分时间：

- 数据在网络上传输的时间
- 站点服务器处理请求并生成回应数据的时间
- 浏览器本地计算和渲染的时间

数据在网络上传输的时间总的来说包括两部分，即浏览器端主机发出的请求数据经过网络到达服务器的时间，以及服务器的回应数据经过网络回到浏览器端主机的时间。这两部分时间都可以视为某一大小的数据从某主机开始发送一直到另一端主机全部接收所消耗的

总时间，我们称它为响应时间，它的决定因素主要包括发送的数据量和网络带宽。数据量容易计算，但是究竟什么是带宽呢？我们将在后续章节中详细介绍带宽的本质。

站点服务器处理请求并生成回应数据的时间主要消耗在服务器端，包括非常多的环节，我们一般用另一个指标来衡量这部分时间，即每秒处理请求数，也称吞吐率，注意这里的吞吐率不是指单位时间处理的数据量，而是请求数。影响服务器吞吐率的因素非常多，比如服务器的并发策略、I/O 模型、I/O 性能、CPU 核数等，当然也包括应用程序本身的逻辑复杂度等。这些将在后续章节中详细介绍。

浏览器本地计算和渲染的时间自然消耗在浏览器端，它依赖的因素包括浏览器采用的并发策略、样式渲染方式、脚本解释器的性能、页面大小、页面组件的数量、页面组件缓存状况、页面组件域名分布以及域名 DNS 解析等，并且其中一些因素随着各厂商浏览器版本的不同而略有变化。这部分内容我们在后续章节中也会适当提到。

可见，一个页面包含了若干个请求，每个请求都或多或少地涉及以上这些过程，假如有一处关键环节稍加拖延，整体的速度便可想而知。

现在，如果有用户向你抱怨在打开站点首页的时候等待了很久，你知道究竟慢在哪里了吗？

1.2 瓶颈在哪里

相信你一定知道赤壁之战，这是中国历史上一场著名的以少胜多的战役，东吴的任务是击退曹操的进攻，要完成这项任务，可谓“万事俱备，只欠东风”，这时东风便是决胜的瓶颈，所以很多系统论研究专家将其称为“东风效应”，也就是社会心理学里讲的“瓶颈效应”。

之所以称它为瓶颈，是因为尽管东吴做了很多的战前准备，包括蒋干中计导致曹操错杀蔡瑁和张允、诸葛亮草船借箭、东吴苦练水军等，但是仅靠这些仍无法获得最终胜利，还需要最后的东南风才能一锤定音，完成火烧曹军战船的计划。不过之前的准备工作都是胜利的子因素，而东南风这个关键因素最终和其他子因素一起相互作用，将整个战斗的杀伤力无限放大。

曹操运气不好，遇上东南风，倒了大霉，曹军战船一片火海，这时候东吴需要派出勇猛的陆军部队登岸攻下曹营，可是东吴向来精通水战，几乎没有强大的陆战部队，只有老将黄盖，这如何与曹操的精英骑兵抗衡呢？这个时候决胜的关键因素变成了刘备的盟军支援，五虎上将各个威猛无比，身怀必杀绝技，此时正是上岸一显身手的好机会，他们不费吹灰之力就将曹军打得落花流水，试想如果没有刘备的支援，赤壁一战胜败可能就扑朔迷离了。

可见，系统性能的瓶颈，是指影响性能的关键因素，这个关键因素随着系统的运行又会发生不断的变化或迁移，比如由于站点用户组成结构的多样性和习惯的差异，导致在不同时段系统的瓶颈各不相同，又如站点在数据存储量或浏览量增长到不同级别时，系统瓶颈也会发生迁移。一旦找到真正影响系统性能的主要因素，也就是性能瓶颈，就要坚决对其进行调整或优化，因为你不得不这么做。

提示：

中医是一门关于生命的哲学，也是中国人智慧的结晶，它的光芒在于独到的思辨能力和系统性的分析方法，它认为世间万物都在不停地变化，并赋予它们阴阳状态，包括天地、季节、天气、心理、生理等，而患者的病理也在随之变化，所以，中医会对同一位患者在不同季节进行不同的诊断，找到不同的病因。

同时，在这些关键因素的背后，也存在很多不能忽略的子因素，构成了性能优化的“长尾效应”，也就是说如果你对某个子因素背后的问题进行优化，可能会带来性能上的少许提升，也许不被察觉，但是多个子因素的优化结果也许会叠加在一起，带来性能上可观的提升。对于诸多子因素的优化，需要稍加谨慎，花点时间考虑这种优化是否值得，以及是否会带来潜在的副作用，还有其他依赖的非技术因素。

然而，不论是关键因素还是子因素，它们的背后都是影响系统性能的问题所在，问题本身并不涉及关键性，只有在不同的系统和应用场景下，才会显示出其是否关键。

本章的其余部分将先列出一些我们经常遇到的问题，并简单介绍我们常用的优化方案，至于这些问题在什么时候是否关键，它们的本质是什么，以及如何调整或优化，在后续章节中我们将结合具体场景来详细探讨包括这些在内的更多主题，这也是本书贯穿始终的线索。

1.3 增加带宽

当 Web 站点的网页或组件的下载速度变慢时，一些架构师可能想到的最省事的办法就是增加服务器带宽，因为他们认为是服务器带宽不够用了，对于一些以提供下载服务为主的站点来说也许是这样的，但是对于其他服务的站点，你知道站点当前究竟使用了多少带宽吗？这些带宽都用到哪里了呢？如何计算站点现在和可预见未来使用的带宽？带宽增加后下载速度就可以加快吗？使用独享带宽和共享带宽的本质区别是什么？如何节省带宽？还有，你可能会忍无可忍地问，究竟什么是带宽？

对于带宽的概念，如果你没有仔细阅读计算机网络教材中的描述，我敢肯定你一定是完全凭借自己的理解来认识它的，因为这个词实在是太有创意了，也实在太容易从字面理解了，

但是这些认识从本质上讲是完全错误的，正是基于这种误解，很多人都无法完全解答上述那一连串问题，导致在所有涉及带宽的问题上，只能依靠经验和猜想。

在后续章节中，我们将通过介绍数据的网络传输原理，彻底揭开带宽的本质，以及数据传输响应时间的依赖因素和计算方法。搞清楚这些一点都不困难，它们是一个优秀架构师必须掌握的基础知识。

1.4 减少网页中的 HTTP 请求

我们知道 Web 站点中几乎任何一个网页都包含了多个组件，每个组件都需要下载、计算或渲染，毫无疑问，这些行为都会消耗时间。那么如果可以让网页减少这些行为，应该就可以加快网页的展示速度，这是毫无疑问的，但是往往我们需要在优雅的网页表现和性能之间权衡取舍，这也许是美和快之间的博弈，找到最优的均衡点至关重要，我们为此做了很多尝试和努力：

- 设计更加简单的网页，使其包含较少的图片和脚本，但是这可能牺牲了美观和用户交互。
- 将多个图片合并为一个文件，利用 CSS 背景图片的偏移技术呈现在网页中，避免了多个图片的下载。
- 合并 JavaScript 脚本或者 CSS 样式表。
- 充分利用 HTTP 中的浏览器端 Cache 策略，减少重复下载。

很显然，这些技巧都来自于 Web 网页前端的优化，在后续章节中我们会有所涉及，但是不作为本书的重点来介绍，本书将更加偏重于站点服务器端的性能改善和规模扩展。

1.5 加快服务器脚本计算速度

我想大多数涉及性能问题的站点都会使用各种各样的服务器端脚本语言，比如主流的 PHP、Ruby、Python、ASP.NET、JSP 等，这些脚本语言用来编写动态内容或者后台运行的小程序，已经成了几乎所有站点的首选。而曾经使用 C++编写动态内容的经历也让我记忆犹新，除了每天都在感叹 C++的严谨和优雅之外，我找不到其他任何好处。

我们知道，用这些脚本语言编写的程序文件需要通过相应的脚本解释器进行解释后生成中间代码，然后依托在解释器的运行环境中运行。所以生成中间代码的这部分时间又成为大家为获取性能提升而瞄准的一个目标，对于一些拥有较强商业支持的脚本语言，比如 ASP.NET 和 JSP，均有内置的优化方案，比如解释器对某个脚本程序第一次解释的时候，

将中间代码缓存起来，以供下次直接使用。

对于开源类的脚本语言，也有很多第三方组件来提供此类功能，比如 PHP 的 APC 组件等。使用这些组件进行脚本优化真的那么有用吗？不同的应用效果是否有所不同呢？在后续章节中我们会详细探讨。

1.6 使用动态内容缓存

动态内容技术就像 Web 开发领域的一场工业革命，它带来了产业升级和 Web 开发者的地位提升，在过去相当长一段时间里，大家普遍认为一个站点的技术含量主要体现在后台的动态程序上，所以很多工程师都会带着虚荣心警告你：“请叫我后台开发工程师。”事实上这种概念和偏见已经开始逐渐被历史抛弃，但这不是我们此刻讨论的重点。

自动态内容技术产生后，聪明的工程师们为了减少动态内容的重复计算，想到了截取动态内容的胜利果实，将动态内容的 HTML 输出结果缓存起来，在随后的一段时间内当有用户访问时便跳过重复的动态内容计算而直接输出。

在实际应用中，动态内容缓存可能是大家使用得最多的技术，但是并不见得所有的动态内容都适合使用网页缓存，缓存带来的性能提升恰恰与有些动态数据实时交互的需求形成矛盾，这是非常尴尬的，而解决该问题的唯一途径不是技术本身，而是你如何权衡。

另一方面，缓存的实现还涉及了一系列非常现实的问题，即成千上万的缓存文件如何存储？缓存的命中率如何？缓存的过期策略如何设计？在拥有多台 Web 服务器的分布式站点上应用动态内容缓存需要考虑什么呢？

在后续章节中我们将详细地探讨这些问题。

1.7 使用数据缓存

动态内容缓存是将数据和表现整体打包，一步到位，但就像快餐店里的组合套餐一样，有时候未必完全合乎我们的口味。当我们意识到在自己的站点中，某些动态内容的计算时间其实主要消耗在一些烦人的特殊数据上，这些数据或者更新过于频繁，或者消耗大量的 I/O 等待时间，比如对关系数据库中某字段的频繁更新和读取，这时我们为了提高缓存的灵活性和命中率，以及性能的要求，便开始考虑数据缓存。

更加细粒度的数据缓存避免了过期时大量相关网页的整体更新，比如很多动态内容都包含了一段公用的数据，如果我们将整个页面全部缓存，那么假如这段数据频繁更新导致频繁过期，无疑会使得所有网页都要频繁地重建缓存，这对网页的其他部分内容似乎很不公平。

此时如何协调网页缓存和数据缓存呢？是否能够将它们一起使用并各显其能呢？

另外，将数据缓存存储在哪里呢？这需要考虑多方面的因素。速度是一方面，如果无法提供高速的读写访问，那么这部分数据缓存可能不久便成为新的系统瓶颈。另外，数据缓存的共享也至关重要，如同一主机上不同进程间的共享、网络上不同主机间的共享等，一旦设计不当，将对站点未来的规模扩展带来致命的威胁。

在后续章节中我们再详细地探讨这些问题。

1.8 将动态内容静态化

在动态内容缓存技术的实现机制中，虽然避免了可观的重复计算，但是每次还都需要调用动态脚本解释器来判断缓存是否过期以及读取缓存，这似乎有些多余，而且关键是消耗了不少时间。直接让浏览器访问这些动态内容的缓存不是更好吗？在这种情况下缓存成为直接暴露给前端的 HTML 网页，而整个缓存控制机制也发生了根本的变化，我们普遍称它为静态化，静态网页独立了，当家做主了，再也不用被脚本解释器呼来唤去。

独立意味着要承担更多的责任，原本动态内容缓存涉及的那些问题，在静态化实践中是否也会出现呢？让我们在后续章节中详细探讨。

1.9 更换 Web 服务器软件

从 20 世纪末开始影响全球经济的开源软件，不可否认给我们的生活带来了更多丰富的体验和选择，但是更多的选择也代表着更多的结局，不论结局是好是坏，我们都需要为此承担责任。

在 Web 服务器软件的选择问题上，很多架构师依然困惑，大量的压力测试对比数据蛊惑着激进的开发者和运维工程师，人们只关注所谓的并发量冠军，却忽视了更加本质性的东西，甚至不了解眼前测试数据的潜在前提。社会总是这样的，象牙塔式的精英教育和残酷的淘汰机制断送了无数人才的未来。而这一次，错误的选择将要付出沉痛的代价。有人拿着所谓的测试数据说 Apache 已经过时，你相信吗？也许下此结论为时尚早，尽管放弃它的人比比皆是，但是它的成功不是空穴来风，毕竟它已经活了很久了。

另一方面，你正在使用的 Web 服务器软件也许让你无比自豪，可是你知道那些复杂的参数配置背后的本质吗？你知道为什么它仅仅在处理你的站点请求时如此出色吗？如果让你自己编写 Web 服务器软件，你可以让它更快一些吗？

我们必须停止盲目的选择，停止对表面现象的崇拜，我们需要学习一些稍显底层的知识来武装自己。在后续章节中，我们将介绍 Web 服务器在并发策略方面的各种设计和其动机及本质，熟悉这些内容后，我肯定你可以解释和分析更多你所看到的现象。

1.10 页面组件分离

从某种角度看，中学校园里的快慢分班似乎合乎逻辑，虽然不一定合乎情理。快班的学生学习能力强，理解知识快，那么课程安排的节奏可以加快一些；慢班的学生则可以放缓课程安排的节奏，这样既互不影响，学校的升学率又可以得到保证，当然假设的前提是学生之间互相帮助效果不大。

在 Web 站点中，网页和各种各样的组件是否也需要“分班”呢？显然它们的下载量和对服务器的能力要求不尽相同，如果由同一台物理服务器或者同一种并发策略的 Web 服务器软件来统一提供服务，那势必造成计算资源的浪费以及并发策略的低效。所以，分离带来的好处是显而易见的，那就是可以根据不同组件的需求，比如下载量、文件大小、对服务器各种资源的需求等，有针对性地采用不同的并发策略，并且提供最佳的物理资源。

当然，如果你的站点基本无人问津，而且服务器的各种资源大量闲置，那么自然不存在什么性能问题，也不需要什么组件分离。但是如果你的站点负载已经让你意识到组件分离是大势所趋，那还是趁早动手。

那么，什么组件需要分离？如何分离？幸运的是，这些并不困难，但是其涉及的知识绝对不仅仅是组件分离本身，在后续章节中我们将会详细探讨。

1.11 合理部署服务器

真让人发疯，互联网为什么不是只有一个，你也许会说难道不是只有一个 Internet 吗？是的，但是 Internet 所特指的“互联网”是某种文化意义上的名词，同一个世界，同一个梦想，同一个互联网。而我这里说的互联网，则是指由某互联网运营商负责搭建的一系列网络节点，它覆盖的地域有大有小，接入这些网络节点的局域网也可以相互通信，同时这些互联网之间也能够通过骨干线路互联互通。

世界上很多国家都有不止一个互联网运营商，中国的互联网运营商想必大家都非常熟悉，当你在家中安装宽带或者需要托管服务器的时候，都面临着运营商选择的问题，包括电信、网通、铁通、移动在内的几大国内运营商让你很头疼。特别是在部署 Web 站点各类服务器的时候，是否能够找到合理的位置部署服务器至关重要。

我们都知道，在基于 IP 寻址的互联网中，IP 地址相近的主机之间通信，数据经过少数的路由器即可到达，比如同一局域网内通信或者接入同一个城市交换节点的局域网之间通信，在这种情况下数据到达时间相对较短。

而如果通信的两端主机位于不同运营商的互联网中，那么数据必须流经两个互联网运营商的顶级交换节点和骨干线路，在这个过程中可想而知数据要经过更多次的存储转发，而且各互联网顶级交换节点之间又存在出口带宽的限制，如果互联网之间数据通信量比较大的话，那么这个顶级交换节点，也就是“出口”，将会是瓶颈所在，就像连接两座城市之间的高速公路，当大量汽车需要频繁地往返于两座城市时，高速公路出现车流缓慢，那么汽车从一个城市到另一个城市的总体时间加长了。

显而易见，我们当然希望 Web 站点的用户和服务器位于同一个互联网运营商的网络内，但如何实现呢？后续章节会涉及这方面的详细内容。

1.12 使用负载均衡

到此为止，我们已经最大程度地发挥了单台 Web 服务器的处理能力，但是，当它所承受的压力达到极限时，就需要有更多的服务器来分担工作，我们需要想办法将流量合理转移到更多的服务器上。

为此，我们需要通过各种不同的方法来实现 Web 负载均衡，可能是简单的 HTTP 重定向，或者是基于 DNS 的轮询解析，或者通过反向代理服务器来实现负载均衡调度，还可以通过 LVS 来组建服务器集群，它们有什么区别呢？无论如何，透过这些具体的实现方法，我们更加关心的是能否真正地均衡调度请求，以及是否具备高可用性，还有影响规模扩展的制约因素，这些内容我们都会在后续章节中详细介绍。

1.13 优化数据库

对于使用数据库的 Web 站点来说，你是否在性能优化时或多或少地忽视了数据库的存在？往往一些性能问题可能都发生在表现不佳的数据访问层面，这来源于不合理的应用程序数据访问组件设计、不合理的数据库表结构设计以及对于数据库内部构造缺乏深入的了解。毫不夸张地说，也许你之前的优化全都白干了。

Web 服务器与数据库服务器的数据通信一般基于标准的 TCP，即便它们位于同一台物理主机也是如此。其通信连接的建立和释放涉及代表一段内核高速缓冲区的文件描述符的创建和销毁，这需要不少的时间开销，包括系统调用导致的内核态切换以及某些异步阻塞 I/O

模型采用的文件描述符队列扫描机制。所以，频繁的数据库连接和释放无疑将导致数据访问等待时间的加长，这段时间浪费得毫无意义。

使用数据库持久连接有效地解决了这一难题，它包括不同程度上的持久化，本质的区别在于持久连接的应用范围和生命周期，比如某个进程内部的全局数据库连接，供进程内所有计算任务共享，在这个进程终止后便被释放；或者在某个动态内容的执行周期内，代码层面的持久连接对象，在动态内容计算结束后便不复存在；还有跨进程的数据库连接池，保存多个持久连接供应用程序重复使用。在这些采用数据库持久连接的应用设计中，同时还要注意保证数据访问的线程安全性。

与此同时，在设计关系数据库的表结构时，你是否合理使用了各种类型的索引呢？要做到这一点，你必须了解索引的有关知识，然而更重要的是如何根据 Web 站点变幻莫测的数据访问特点来有针对性地设计每个表的索引，这往往也是最有难度的，索引的合理使用对于依赖数据库访问的 Web 应用至关重要。

另外，你了解数据库存储引擎的特性吗？其实这并不困难，因为所有的主流数据库文档中都有详细介绍，但是究竟你的 Web 站点应该选择什么存储引擎呢？当然，没有绝对完美的方案，我们在这个世界上要做的唯一的事情就是不断进行取舍，像考虑索引一样去弄清楚存储引擎的本质，是绝对不会让你失望的。

随着时间的推移，你的 Web 站点可能逐渐被数据库绑架，单台数据库服务器再也无法应付整个站点的需要，这包括存储空间以及查询时间，人们开始抱怨数据库模型的不良设计制约了横向扩展以及负载均衡，这不是我们希望看到的结果。为此，我们将数据散列在多台主机，包括必要的冗余数据，以此来合理地分散数据库的密集访问，数据库扩展便成为我们考虑的方案。

1.14 考虑可扩展性

对于前面列举的诸多方案，在本书中我们不仅从其对性能影响的角度来深入探讨，同时还会适当地涉及开发、调试以及可扩展性。对于 Web 站点的可扩展性讨论已经屡见不鲜了，不论是代码层面的扩展，还是架构层面的扩展，涉及的内容非常多，究竟我们应该从何谈起呢？这是一个值得深思的问题。缺乏良好的可扩展性设计就像慢性自杀或者等待死亡，这甚至比 Web 站点所能遇到的其他一切困难更让人头疼，因为扩展对于我们来说，就像在山穷水尽的时候被指引了一条星光大道，一旦扩展都无法进行，那真是死路一条。

的确，可扩展性并不是性能和速度的概念，它是指当系统负载增大时，通过增加资源来提高性能的能力。高性能往往需要通过这种能力来实现快速扩展，几乎没有多少团队可以在一个星期内通过增加服务器马上让服务能力扩容 100 倍。另一方面，可扩展性的目的在于

适应负载的变化，从扩展的技术实现上来看，又包含了很多对局部性能的思考，以及了解何时需要扩展，这离不开对站点性能的把握。

然而，就像“人的病都是吃出来的”一样，Web 站点在成长的道路上不断吸收新的技术，然而每一次技术的应用不当，都可能引入一定程度上的不可扩展。但现实往往是矛盾的，我们不得不使用一些技术来构建 Web 站点，同时又使用一些技术来提升站点性能，这些技术构成了我们理想架构的一部分，关键在于在这些技术和架构的应用中，我们是否意识到可扩展性，并且能否正确评估可扩展性的需求。

在本书后续的内容中，我们不打算用一个独立的章节来探讨可扩展性问题，而是将其渗透在所有需要考虑可扩展性的章节中，因为这样的组织方式可能更加适合读者的阅读。

1.15 减少视觉等待

实在不行就给用户一些提示吧！最后我只能这么说了，事实上，这不是什么大不了的事情，即使认识到架构的瓶颈并投入大量人力来改善，也不是一天两天就可以完成的，要意识到用户也许只是希望你不要不理他而已。

这部分显然已经超出了本书的讨论范围，它涉及人机交互的相关知识，并且充满着人文情怀，要真正做好它，恐怕要比本书中所有的问题都更有难度和挑战性，这毫不夸张，我们要承认这个现实，因为世界上最难的学问就是研究人，你觉得呢？

第 2 章

数据的网络传输

事实上，围绕本书的一切内容都离不开网络，这是无可置疑的，站点需要通过网络来响应用户的请求，站点内部也需要通过网络来组建分布式环境。那么，你了解数据在网络传输中的一些原理和动机吗？比如存储转发、流量控制、带宽和响应时间等，通常它们看似顺理成章，因为大多数开发者生活在应用层，这些似乎与他们毫无关系，然而一旦当你开始将注意力转向站点性能时，这些基础知识便是你不能不知道的。

2.1 分层网络模型

关于网络模型的阐述，许多参考书中都有涉及，普遍采用 OSI 七层网络模型或 TCP 四层网络模型展开介绍，所以在这里我们不打算重复罗列这些内容，而是希望探讨网络模型中一些细节的本质和实现，尤其是数据传输比较低层的实现，包括分组交换、流量控制、数据转发等，以便对大家理解本书后续章节有所帮助。

如果你觉得已经非常熟悉网络模型的细节，并且不需要再加深理解，可以直接跳过本节，并在任何需要的时候回到这一章。如果你希望先温习一下所谓的 OSI 七层网络模型或 TCP 四层网络模型，可以查阅丰富的在线资料。

我想用另一种描述方式带大家回顾一下网络模型，通过假想的场景来透视网络模型的设计动机以及在理想场景下的实现。为此，故事从一个遥远的星球开始，讲述他们的铁路体系从无到有的过程，请注意，这是一个理想的场景，所以请大家先将脑海中关于现实中铁路系统的认识屏蔽，一个虚幻的铁路系统即将诞生。

这个故事也许看起来比较复杂，或许你可以把它当做小说一样阅读，但一定是推理小说，因为你要动脑筋搞清楚故事中提到的所有角色。请再次注意，一定要把心平静下来，同时，在没有搞清楚故事中某些内容和真实网络模型的对应关系之前，请不要习惯性地对号入座，否则可能会给你的理解带来一些不必要的认知偏差。

发展铁路运输

话说在遥远的星球上，有一座美丽的城市，市民们过着简单而快乐的生活。随着市场经济的蓬勃发展，城市的货物运输需求越来越大，市民们迫切希望能够将货物快速运输到城市

的各个角落。于是市政厅决定修建城市铁路，大力发展铁路运输，并将这个艰巨的任务交给了新设立的铁路公司。

铁路公司对于修建铁路没有任何经验，只是听说地球上有发达的铁路，但了解不多。幸好城市里少数学者去过地球，带回了一些宝贵的资料和照片，于是铁路公司召集星球上的专家和工程师，展开了紧张的研究工作。

凭借着大家的智慧，不久之后，研究工作有了突破性的进展。大家掌握了修建铁路的方法，同时意识到另一个更加重要的问题，那就是在铁路上如何运输货物，这需要设计能够在铁路上行驶的火车。经过一段时间的研究，设计火车的工作并没有难倒大家，科学家们利用本星球的特殊材料，结合之前的铁路，设计出了能够高速行驶的火车。

接下来，大家又遇到了新的问题，如何铺设铁路呢？如何使火车根据运输的需要寻找到正确的目的地呢？如何知道火车是否正确到达目的地呢？如何处理中途遇到故障的火车呢？这一系列的问题立刻让大家意识到，必须要设计出一套强大的铁路运营体系，否则这一切都将只是破铜烂铁。其实在一开始，有一位聪明的工程师就意识到了这个问题，于是他一直在默默地研究，终于在此时拿出了他认为成熟的方案，让我们一起来听听他的想法。

铺设铁路

首先，在城市里修建多个火车站，作为火车的出发地或目的地。每个火车站拥有自己唯一的地址，比如“STATION-23-EF-59-32-AD-B9”，看起来很酷，这是火车唯一能识别的地址格式。

接下来，需要铺设铁路连接这些火在站，有以下两种方法：

- 铺设一条铁路将所有火车站串联起来，所有火车都行驶在这条铁路上。
- 在城市中心位置修建一座强大的调度站，所有火车站都与调度站之间铺设铁路相连，任何火车站之间行驶的火车必须经过调度站。

由于第 1 种方法实施起来比较简单，所以大家一致决定采用它，修建一条铁路连接所有火车站。可以肯定的是，火车只要知道该城市内目的地火车站的地址，就一定会知道如何到达。

预约线路

接着，需要规定如何运输货物。这时候，一位曾经设计了本星球电话网络的专家发言了，

他给出了运输货物的方法。当有人需要运输货物的时候，应提前向铁路公司预约线路，也就是需要由铁路公司将运输路线所涉及的所有铁路做好准备，租给这个人来使用。铁路公司准备好之后，只可以由这个人使用此段铁路来运输货物，并且根据占用时间来缴纳一定的费用给铁路公司。当这个人完成所有的运输任务后，告知铁路公司并进行结算，同时铁路公司可以将空闲出来的铁路租给其他人使用。

货物包

工程师提出了反对意见，他认为在某个人租用铁路的这段时间内，不一定一直有火车在运输，也许货物运输有一定的时间间隔，但是该线路以及两边的火车站一直在被此人占用，无法给其他人提供及时的运输服务。尽管铁路公司按照时间收费，似乎没有少赚钱，但是在很多时间里铁路空闲着非常浪费，降低了铁路的利用率。

为了使铁路更加适用于为广大市民提供及时的运输服务，经过大家的讨论，决定使用一种新的货物运输方式，将大家的货物分成若干份，分别装入不同的货物包，每个货物包装进一个小型火车单独运输，同时一定要在火车头上写出目的地火车站的地址。这样大家就不需要预约线路，而是有货即发，并且当发货人数较多时也不需要等待太长时间，可以让每个人的货物包穿插在一起依次发出。

一切看起来都完美极了，大家决定按照这个方案开始修建铁路和火车站并投入试运营。在试运营的几天里，大家发现一系列问题，不得不先暂停了运输服务，开始了新的讨论。

调度站

首先，大家发现目前采用的铁路铺设方法使得铁路利用率非常低，因为在一条连接所有城市的铁路上，每列火车的目的地多有不同，为了避免火车相撞，在任何时刻，必须只允许某两个火车站之间行驶火车，而其他火车站则必须等待发车。

于是，大家开始考虑当时提出的第 2 种铁路铺设方式，在城市中心位置修建一座强大的调度站，然后所有火车站都与调度站之间铺设铁路相连，任何火车站之间行驶的火车必须经过调度站。调度站设有工作人员，将驶入调度站的火车根据火车头上的目的地火车站地址，调度至连接该火车站的正确线路。同时，工程师们还改进技术，设计出了更加强大的高级调度站，可以将从不同火车站同时驶入调度站的火车进行同时调度，互不影响，这一项技术改进使得铁路的利用率大大提升。

车流控制

接下来的问题是，火车站在为火车装货和卸货的时候需要一定的时间，而且时间可能不相同，这往往取决于目的地火车站货物临时库房的大小以及搬运速度。如果货物临时库房太小或者进入库房的速度太快的话，还没将货物转给收货人，临时库房就满了，新的货物无法进入仓库，便容易发生丢失。也就是说，如果目的地火车站的卸货速度比出发地火车站的装货速度慢，那么按照出发地火车站的装货速度发出的火车到达目的地火车站后，便容易发生货物丢失，这种情况非常严重，必须避免发生。

为此，聪明的工程师认为，必须要通过某种反馈机制，让出发地火车站实时了解目的地火车站的卸货速度，从而不断调整发车速度，比如某目的地火车站 1 小时能完成 10 列火车的卸货任务，则出发地火车站在 1 小时内不能发出超过 10 列火车。那么如何实现这种反馈机制呢？

在场有位工程师提出，当一列火车到达目的地火车站并且成功地完成卸货后，目的地火车站必须通知出发地火车站，而且出发地火车站必须等待该通知后，才可以发出下一列火车，这样就不会发生之前的问题了。

但是，另一位工程师认为这种机制虽然实施起来简单，但是很明显可以看出，从出发地火车站发出一列火车到目的地火车站发送通知的这段时间内，出发地火车站无法继续发车，由于火车行驶也需要时间，导致卸货员有一部分时间处在等待状态，所以发车速度反而低于卸货速度，这大大降低了铁路的利用率。

这位工程师提出了改进方案，让出发地火车站连续发出多列火车，但是数量有限，这些火车恰好能将这段铁路利用起来，同时也等待目的地火车站的通知，当目的地火车站到达一列火车并且卸货完成后，通知出发地火车站，这时出发地火车站再发出一列火车。这样的话，两个火车站之间的铁路全线不会闲置。大家很赞同这种方法，同时还决定成立技术小组长期致力于提高卸货速度的研究。

货物重发

另一个问题是，货物在运输过程中有可能会被损坏，比如由于路况问题或者环境问题，以及恐怖主义袭击等原因，大家觉得铁路公司应该保证所有货物都能准确到达目的地，并对这些货物的完整性负责。于是在每次发货的时候，应详细清点货物，并将清单贴在火车尾部，目的地火车站卸货后要根据清单检查货物，如果发现货物损坏，必须通知出发地火车站重新发送。这就要求出发地火车站在每种货物发送之前，将其复制一份，放在库房里，一旦得知货物损坏后，就进行货物重发。

其实，在后来的实际运营中，铁路公司认为由自己来进行货物重发，过程相当复杂，而且

还涉及很多细节问题需要处理，比如重复发货带来的货物到达顺序不同等，同时需要花费大量的时间和资金成本。另一方面，实际在运输途中损坏货物的情况并没有想象的那么多。所以铁路公司决定对于损坏的货物不进行直接重发，而是由收件人根据损坏情况决定如何处理，包括要求铁路公司重新发货或者赔偿等。

跨城运输

以上的问题解决后，这座城市的铁路运输就可以有条不紊地进行了。过了不久，铁路公司似乎想到了些什么，所有专家和工程师被聚集在一起，开始考虑一个新的问题，如何将这座城市的铁路系统推广到星球上的其他城市，并且不同城市的火车站之间也可以运输货物。

将铁路系统推广到其他城市比较容易，只要派专家和工程师去其他城市建设一套一模一样的系统即可。但是要让这些城市的火车站可以相互运输货物，目前还无法解决，这包括以下几个问题：

- 如何铺设铁路连接不同城市的各火车站？
- 火车如何知道目的地火车站的地址是本城市的还是其他城市的？
- 如果目的地是其他城市的火车站，火车如何到达呢？前面曾经提到，火车只要知道该城市内目的地火车站的地址，就一定会知道如何到达，但对于其他城市可不行。

工程师们希望能够在不推翻当前整个系统的情况下，设计出一套方案来解决以上几个问题，这套方案必须包括以下内容：

- 设计城市之间的互联方式，以实现所有城市的任意火车站可以相通。
- 设计良好的寻址方式，以实现火车在跨城市行驶时可以找到正确的目的地。

城市互联

这时，一位工程师提出城市中转站的概念，他提议在每座城市选一个火车站，作为该城市的对外中转站，这样的话，当火车需要从一个城市进入另一个城市时，就可以通过中转站到达。

关于中转站的铁路连接，根据情况有不同的方式，比如多个小城市的中转站可以同时连接到附近一个大城市的调度站。另一方面，需要对中转站进行扩建，它的卸货速度要比其他火车站快很多，以避免来自城内多个火车站的火车同时出城造成的中转站拥挤现象，导致货物丢失。

火车站名称

有了以上的城市互联方案后，接下来要解决的问题是如何让火车可以准确地找到不同城市的目的地，这就需要在火车站地址中增加关于城市的描述，而目前的火车站地址格式无法实现这部分扩展。

工程师们意识到必须设计出一套新的火车站命名方案，并且能与现有的铁路系统很好地协作运转。新命名方案必须满足以下条件：

- 可以被火车直接或间接识别。
- 包含火车站地址。
- 包含城市信息。

这时，一位一直沉默不语的可用性研究专家终于开口了，他说出了埋藏在心里很久的想法，从一开始，他就觉得那串火车站地址看起来真是太丑陋了，而且难以记忆，让市民们发货的时候填写这个地址很不人性化，应该趁这次机会给火车站取个优雅的名字。

聪明的工程师仔细考虑后给出了自己的方案：由每个城市的市民来为城内所有火车站命名，但不能重复，然后在火车站的名字前追加该城市的名字，作为火车站的名称。比如：

太空城．和平站

到目前为止，火车站的标识有以下两种，大家不要搞混。

- 地址：由铁路公司给火车站起的名字，火车可以识别。
- 名称：由市民们给火车站起的名字，火车无法识别。

有了火车站名称后，在发货的时候，要求发货人在自己的货物包上贴一个标签，写上目的地火车站的名称，在装货的时候，由出发地火车站负责将其转换为火车可以识别的目的地火车站地址，并标注在火车头上。这个转换包括以下两种情况：

- 如果目的地火车站和出发地火车站在同一城市，则直接转换为目的地火车站地址。
- 如果目的地火车站和出发地火车站在不同城市，则转换为该城市对外中转站的地址。

对于上述第 2 种情况，出发地火车站必须事先知道该城市对外中转站的名称或地址，所以铁路公司要求所有火车站必须记住所在城市的中转站名称。

对于连续跨域多个城市的运输，中转站通过有效的方法可以很好地完成任务，在这里我们不进行细节讨论。

火车站地址转换

上面刚刚提到的火车站地址转换，具体如何实现呢？每个火车站必须有一份列表，内容是其他火车站名称和地址的对应关系，每次需要根据货物包上的火车站名称快速查表并找到火车站地址。

但是由于某个新修的火车站刚刚运营或者某个火车站刚刚改用新的名称，这时候出发地火车站都无法查到，所以铁路公司规定，出发地火车站可以通过城内铁路广播站发送一条广播，询问哪个火车站拥有这个名称，并要求该火车站告知自己的地址，这样就彻底解决了地址转换的问题。

可是，有一天奇怪的事情发生了，本来运往和平站的货物却运到了光明站。经过调查，原来是有恐怖分子秘密劫持了光明站，出发地火车站不知道和平站的地址便发出广播，这时候光明站冒充和平站发回了答复，并将光明站的地址告诉了出发地。

经过这次教训，铁路公司对于治安情况不佳的城市，要求每个火车站采取各种有效措施来核对自己的地址对应关系列表，必要的时候对一些重要火车站进行地址关系绑定，不要轻易相信广播的反馈。

传输包

一方面，铁路系统在不断完善的过程中；另一方面，货物包内部以及发货方式也发生了很多变化。发货人将货物先装进另一个袋子里，这个袋子称为传输包，上边贴一个标签，注明发货人和收货人的详细信息，然后通知火车站来取货，火车站工作人员将传输包装进货物包，在货物包的标签上注明目的地火车站名称。这样当货物包到达目的地火车站后，工作人员将运输包取出，根据标签上的信息将其转达给收货人，收货人通过标签也能够了解到发货人的信息。

有时候，发货人和收货人之间需要建立一种可靠的交流，大家来回发送一些学术论文，不能有任何的损坏和丢失。还记得铁路公司对于货物损坏自动重发的过程吗？与此类似，发货人和收货人之间会制订一套完善的约定，使得双方通过多种有效的机制，保证货物的可靠运输。这套约定通过增加传输包标签上的描述信息来实现，具体细节我们这里不作介绍。

有时候，发货人和收货人之间存在一定的生意往来，但是由于收货人在货物销售方面不太顺利，需要减少发货量或者降低发货频率，这时，收货人可以采取一些办法来通知发货人，发货人便按照收货人的要求来减慢发货的速度，也就是说每天少去几次出发地火车站。

还有时候，发货人和收货人之间相隔了多个城市，由于火车要经过多个中转站进行货物转发，并且各中转站的规模和卸货速度又不尽相同，所以出发地火车站无法立刻知道目的地

火车站的卸货速度。在前面我们知道只有在同城内的火车站才能使用车流控制的方法。这时候，需要由收货人直接和发货人建立某种形式的沟通，来告诉发货人减慢发货速度，否则货物丢失严重，引起大量的货物重发，这种沟通我们不作详细介绍，但是可以知道它是通过传输包往来以及在传输包标签上标注了更多信息来实现的。

通过传输包，任何市民都可以相互运输货物，可以说只要把货物放进传输包里，并且写好标签，通知铁路公司，接下来的事情就不用操心了。

产品包

一些市民逐渐习惯将货物不直接装入传输包，一般货物都有自己的包装，比如礼品包，大家喜欢把礼品装进一个漂亮的礼品包中，上边贴上标签并写一些祝福的文字，然后再将礼品包装进传输包。

另一个例子是，某城市里有一家玩具厂，可以给市民们提供玩具订购服务，所以玩具厂设计了订购包和产品包，任何市民如果想要订购玩具，可以将订购要求按照规定的格式，填写在玩具厂派发的订购包标签上，比如订购款型、尺寸、颜色等，同时可以将一些必要的个性化玩具素材（比如个人照片）装进订购包提供给玩具厂。玩具厂收到订购包后，根据标签进行加工，然后将玩具装入产品包，产品包的标签上注明了玩具的使用手册和注意事项等，最后发到订购者的手中。

故事角色介绍

一部反映某星球铁路系统发展变革的大型故事就这样结束了，如果拍成电影，场面一定颇为宏大。故事中的角色能否引起你的共鸣呢？也许大家在阅读故事时不断地猜想各个角色所对应的现实化身，那么现在就让我们来揭开这些角色的面纱。在这之前，请大家注意，这里讲的角色，并不特指人物，而是包括故事中所有的道具和行为。

铁路

相当于物理层的线路，它为相互通信的主机建立物理连接，提供信号传输介质。物理层线路常见的传导材料有铜线和光纤，我们常用的双绞线使用的是铜线，价格便宜，它传输的是电信号。而光纤价格昂贵，它传输的是光信号。那么信号在光纤中的传输速度和铜线相比，谁更快呢？我们将在下一节中揭开真相。

火车站

相当于网络中的节点，比如计算机、路由器等，通过节点的网络接口（比如计算机的网卡），内部数据总线与网络物理层线路间接连接。网络接口负责将节点内部总线中的字节数据和

线路中传输的二进制信号进行相互转换。

火车站地址

相当于节点物理地址，也就是 MAC 地址，它由设备生产厂家唯一确定。通过现代电路技术，线路中的数据信号可以很容易传输到指定 MAC 地址的节点。

火车

相当于数据链路层的数据帧，它包括头部信息和数据区，头部信息指明了目的地节点的 MAC 地址。火车、火车站、火车站地址等一系列涉及的实体和规则可以统称为数据链路层。

火车站串联方式

相当于总线网络拓扑结构，是比较古老的一种主机连接方式，具体介绍可以查看在线资料。它的缺点之一是数据传输采用广播方式，所以所有主机共享网络带宽，并且在这种情况下，由于线路过长会使信号衰减，所以不能无休止地通过中继器延长总线来增加网络节点，限制了网络的扩展。

火车站调度站方式

相当于星形网络拓扑结构，使用额外的中间节点来交换数据，且中央节点负担较重，但是这种形式控制方便，易于扩展。星形网络的带宽是否共享是由中央节点的处理方式决定的，常见的有集线器和交换机，后文中会有详细介绍。

普通调度站

相当于早期的集线器，它通过简单的集成电路实现了星形拓扑网络结构，但由于其只工作在物理层，所以对于来自某一主机的数据帧并不知道它的目的地是哪里，也就是无法识别 MAC 地址，所以不加判断地直接将数据帧转发到所有其他端口连接的主机。它的本质和总线网络相同，都是广播方式的数据转发，需要有数据帧碰撞检测机制，所有主机共享整个网络的带宽，容易相互影响。

高级调度站

相当于交换机，它拥有独立的处理器和高速缓存，对数据采用存储转发方式，并且使用更加先进的电路设计，可以对数据帧根据 MAC 地址进行准确转发，抛弃了传统的广播方式，这就使得网络中的主机通信不会相互影响，大家都可以独享网络带宽。注意，到现在为止，已经有好几处出现了“带宽”这个词，下一节将详细探究带宽的真相。

预约线路

相当于电路交换方式。本来要在故事剧情中砍掉这个角色，因为它在数字通信网络中实在是非主流，不过考虑到它在电话网络中的突出贡献，而且涉及数据帧诞生的历史，所以还是让它在故事中露个脸。正如故事中所讲，我们很容易理解电路交换的局限性，打电话占线就是一个很好的例证，但是在电话网络中为什么还一直在用电路交换方式呢？请注意，打电话和计算机通信是完全不同的应用，需求也完全不同。没有人愿意并且有能力同时跟多个人打电话，就算有这个需求那就安装多部电话吧。电话拨通之前需要建立连接，等待的时间对大家来说正好是心理准备的缓冲时间，完全可以接受。另外，电话之间单位时间传输的数据量是恒定的，就是源源不断流入话筒的声音，就算人不说话，也是有环境声音的。而计算机通信的需求就非常复杂，数据传输突发性强，数据量变化频繁，数据可靠性需要保证，传输速度需要不断提高，所以人们觉得电路交换方式无法满足需求，在当时便设计出分组交换方式，即将数据分成若干小块，称为数据帧，分时依次发送。

火车站车流控制

相当于数据链路层的流量控制。数据链路层作为控制数据帧传输的底层，必须保证到达目的地主机网卡的数据帧可以交付给上层应用，这个交付工作具体来说，就是网卡把接收到的二进制信号转换为字节，然后写入由操作系统内核在内存中拥有的一块高速缓存区，同时通过中断切换到相应的应用程序进程并取走这些数据。有很多原因可以导致这块缓存区某时刻被写满，比如同时下载多个文件使得写入速率超过取走速率，这时候如果数据帧的发送端不减慢发送速率，则很多数据帧到达后，网卡会因为缓存区已满而丢弃这些数据，因为网卡对此确实无能为力。这就需要在数据链路层存在一种流量控制机制，故事中车流控制的目的与此相同，但是实际采用的方法要复杂得多，一般采用的是滑动窗口技术，细节这里就不作介绍，感兴趣的朋友可以查阅在线资料。

火车站货物损坏自动重发

相当于数据链路层的差错控制和选择重发。事实上，对于我们使用的以太网，由于线路质量非常好，数据链路层已经不做差错校验和自动重发，而是将校验及重发控制权交给更上层的协议，比如 TCP。而对于 802.11x 等无线网络以及卫星通信领域，数据链路层还是需要进行大量的数据可靠性校验工作。

城市

相当于拥有同一个广播域的网络，可以理解为局域网。

城市广播站

相当于局域网的广播地址，发往这个地址的数据帧会被发送到局域网内所有的节点。

火车站名称

相当于节点网络接口的 IP 地址，由用户自行设置或者通过 DHCP 获得。

火车站地址转换

相当于网卡 IP 地址到 MAC 地址的转换。因为数据帧在网络中传输时，只可以通过目的地 MAC 地址来寻址，所以必须将目的地 IP 地址转换为 MAC 地址并告知数据帧。具体的转换方式是：数据发送端通过 ARP 协议发送广播到局域网内所有的主机，询问大家谁拥有指定的 IP 地址，如果拥有的话就请告之自己的 MAC 地址。

火车站地址关系绑定

相当于将局域网内某其他主机网卡的 MAC 地址和 IP 地址的对应关系写入 ARP 表，也称为“ARP 绑定”。对于绑定后的目标主机，任何时候发送数据都不需要进行广播询问，而是直接从 ARP 表中寻找对应关系即可。

城市中转站

相当于路由器，它工作在网络层，可以识别数据帧中携带的目的地 IP 地址，并将其转发至正确的网络。

城市互联

相当于很多网络通过路由器连接，形成互联网。

货物包

相当于网络层的数据包，它的头部信息包含了 IP 地址，主要用于数据帧的寻址。

发货人和收货人

相当于建立通信的两个主机上运行的进程，发货人和收货人的详细信息相当于这些进程的网络端口号。

传输包

相当于传输层的数据包，包括头部信息和正文，在头部信息中指明了用端口号来代表的进程，它们实现了真正意义上的两个主机进程之间的网络通信，而很好地屏蔽了底层的传输细节。在故事中，传输包在有些时候表现出了不同的特性，它们代表了某些传输层协议的控制机制，包括流量控制、流量限制、错误重发等。常见的传输层协议包括 TCP 和 UDP。

礼品包、订购包、产品包

相当于应用层的各种数据包，它们根据应用程序之间的协议约定，对数据进行再次封装，包括头部信息和正文，比如 HTTP、FTP、SSH 等。

故事尾声

这个故事也许会给大家一定的启发，故事中力求将原本静态的知识，通过时间和情节的安排，鲜活地展现出各阶段背景和发展动机，而内容中各角色难免会与现实中网络模型存在较大的出入，所以本故事仅供参考。人类各个领域文明的进步离不开技术的发展，故事中铁路系统与现实中数字通信的发展有着巧妙的相似之处，因为它们的思想都来源于人类的意识形态和实践体验。

2.2 带宽

“带宽”也许是计算机科学中最幽默的一个词，当我向一些开发者询问到底什么是带宽的时候，他们的回答总是让我联想到类似高速公路的宽度，而当我继续询问“我们的带宽有多宽”时，他们就会不知所措。

另一部分开发者显然知道带宽的单位是“bit/s”，也就是单位时间的比特数，所以他们将带宽解释为数据在线路中的移动速度，也就是将带宽的高低视为线路能力的强弱，那么很显然他们认为光纤对数据的传播能力大于铜线，但很可惜，事实上这是错误的。顺便说一下，我们一般常说的比如 100M 带宽，全称应该是 100Mbit/s，或者 100Mbps，后边的“bit/s”经常省略。

这些误解关键在于对带宽中的“带”理解错位，这可能归咎于一部分人忽视了数据的发送环节，而带宽的真正含义也正在于此，那么你了解数据是如何从主机进入线路的吗？带宽中的“带”究竟指什么呢？

值得注意的是，除了计算机网络的相关教科书外，几乎所有涉及带宽的资料都将带宽草率地解释为数据的传输速度，严格地说这种表述很不准确，但是其存在也有一定的道理，那就是单从数字通信的角度来看，为了抽象数据的发送和传播等过程，使用数据传输速度来解释带宽，对于不关心数据发送细节的人们来说更加容易理解。但在本书中，我们关心的是带宽与高性能 Web 站点相关的一些必要知识，所以要深入了解数字网络通信领域的带宽概念，实际上它已经超出了通信的范畴，我们需要回到计算机体系结构中，因为带宽的发源地之一便是计算机系统总线。

数据如何发送

说到数据的发送，也就是数据从主机进入线路的这段旅程，一般需要经过以下几个环节：

1. 应用程序首先得将要发送的数据写入该进程的内存地址空间中，熟悉网络编程的开发者对这个环节一定非常熟悉，通常在程序开发中这只需要一般的运行时变量赋值即可。

2. 应用程序通过系统函数库接口（比如 send 函数）向内核发出系统调用，由系统内核来进行随后的操作，它将这些数据从用户态内存区复制到由内核维护的一段称为内核缓冲区的内存地址空间。这块地址空间的大小通常是有限的，所有要发送的数据将以队列的形式进入这里，这些数据可能来自于多个进程，每块数据都有一定的额外记号来标记它们的去向。如果要发送的数据比较多，那么该系统调用需要多次进行，每次复制一定的数据大小，这个大小取决于网络数据包的大小以及内核缓冲区的承载能力。重复的系统调用体现在应用编程层面重复调用 send 函数。

3. 当数据写入内核缓冲区后，内核会通知网卡控制器前来取数据，同时 CPU 转而处理其他进程。网卡控制器接到通知后，便根据网卡驱动信息得知对应内核缓冲区的地址，将要发送的数据复制到网卡的缓冲区中。注意在以上一系列的数据复制中，数据始终按照连接两端设备的内部总线宽度来复制，也就是字节的整数倍，比如在 32 位总线的主机系统中，采用 PCI-X 总线接口的网卡一般使用 32 位总线宽度，那么从内核缓冲区到网卡缓冲区的数据复制过程中，任何时刻只能复制 32 位的比特信息。

4. 网卡缓冲区中的数据需要发送到线路中，同时释放缓冲区来获取更多要发送的数据。但是我们知道，只有二进制的数字信号才可以在线路中传输，所以这时候需要对数据进行字节到位的转换，这种转换不难想象，就是将数据的每个位按照顺序依次发出。

5. 发送时，网卡会使用内部特定的物理装置来生成可以传播的各种信号，比如在使用铜线线路时，网卡会根据“0”和“1”的变化产生不同的电信号；而使用光纤线路时，网卡会产生不同的光信号。

电磁波的速度

不管是电信号还是光信号，只要进入线路后，便能够进行快速的传播，这个速度称为传播速度，它的单位是“m/s”，即单位时间传播的距离。传播速度只与传播介质有关，铜线中电信号的传播速度大约为 2.3×10^8m/s，光纤中光信号的传播速度大约为 2.0×10^8m/s。

你也许有些疑问，光的传播速度难道不是 3.0×10^8m/s 吗？没错，在真空中的光速大约为 3.0×10^8m/s，同时它不仅仅是可见光的传播速度，也是所有电磁波在真空中的传播速度。根据现代物理学，所有电磁波（包括可见光）在真空中的传播速度是常数，即是光速。

那为什么光纤中的传播速度要低一些呢？这很容易理解，光纤在传播光信号的时候，利用了光的全反射原理，所以光信号在光纤中的实际传播距离要大于光纤的长度，而我们一般所说的传播速度都是针对光纤的长度。同时，正是由于利用了光的全反射原理，微细的光纤封装在塑料护套中，使得它能够弯曲而不影响光信号的传播。

由此可见，不同的传播介质中信号的传播速度几乎等于常量。也就是说，不论数据发送装置以多快的发送速度让数据以信号的形式进入线路，在线路中信号的传播速度几乎可以认为是一样快的。根据电磁学的定律也可以说明这一点，发射电磁波的物体的速度不会影响电磁波的传播速度。结合相对性原则，观察者的参考坐标和发射光波的物体的速度不会影响被测量的光速。

数据发送速度

显然，我们所讲的带宽是指数据的发送速度。比如我们的百兆网卡，便是指网卡的最大发送速度为 100Mbps，也就是网卡在 1 秒钟最多可以发出 100Mb 的数据。那么，我们当然希望发送速度越快越好，究竟这个发送速度的大小跟什么有关呢？简单说包括以下几个因素：

- 数据发送装置将二进制信号传送至线路的能力，也称信号传输频率，以及另一端的数据接收装置对二进制信号的接收能力，同时也包括线路对传输频率的支持程度。比如光纤一端的发射装置使用发光二极管（Light Emitting Diode，LED）或一束激光将光脉冲传送至光纤，光纤另一端的接收装置使用光敏元件检测脉冲，从而将光脉冲中包含的二进制信息转换成数据。值得注意的是，信号的接收能力至关重要，如果接收能力跟不上，发送能力不可能提高，在星球火车系统的故事中，我们知道数据链路层对于数据帧传输的控制机制完全是按照接收方的接收能力来确定发送速度的。
- 数据传播介质的并行度，这里也可以称为“宽度”，完全等价于计算机系统总线宽度的概念。比如在光纤传输中，我们可以将若干条纤细的光纤并行组成光缆，这样就可以在一个横截面上同时传输多个信号，就像在 32 位的计算机总线中，可以同一时刻传输 32 位数据。

有意思的是，要提高计算机总线的带宽，包括提高总线频率和总线宽度两种方法，比如使用 64 位总线系统或者使用主频更高的处理器等。这两种方法与以上数字通信带宽的两个决定因素完全相似。说到这里，你明白带宽的真正含义了吧。

事实上，数字网络通信相比于计算机内部数据传输而言，其传输距离要远远大于后者，所以它还隐藏着另一个因素，那就是信号在传播介质中的衰减，这与传播介质以及信号传播方式有着密切的关系。比如我们一般使用的双绞线，传输距离只能达到 100m，更长的距离就需要通过中继器来延续信号，而且只能使用有限次的中继，每次中继器转发信号又会消耗一些发送时间。而光纤中光的传导损耗比电在电线传导的损耗低得多，它的传输距离一般在数千米以上，所以光纤一般被用做长距离的数据传输。随着光纤的价格日渐降低，光纤未来一定会走进千家万户。

随着技术的不断进步，前面提到的影响数据发送速度的各种因素都在进行不断的技术突破，带宽正在飞速提升，从早期几兆比特的带宽到现在跨越大洋的百吉比特带宽，未来将要建造更多的跨国交换节点，能够支持几千吉比特的带宽。

了解了有关带宽的一些知识后，我们回到实际的应用中，你会发现很多现象的本质都不难解释。

为什么要限制带宽

在大多数情况下，我们都将 Web 站点服务器托管在 IDC，通过将其连接到某个交换机，从而接入互联网。这时候，我们的服务器拥有自己的 IP 地址，当站点用户通过互联网向这台服务器请求数据后，数据从服务器流经交换机到达指定的路由器，这个过程需要交换机的存储转发机制，也就是交换机从连接服务器的端口接收数据，存储到交换机内部的高速缓冲区队列中，然后将其从连接路由器的端口发送出去，再经过路由器的转发，进入另一个网络，接下来依次重复这些过程，直到进入站点用户的 PC。

如果全世界只有你的服务器和你的用户在传输数据，那么这部分数据流经的每个交换节点都会全心全意地做好转发工作，此时你的数据在各节点转发的发送速度都可以达到理论上设备所能达到的最大值。但实际上每处交换节点都有可能同时转发来自其他主机的数据，包括你的数据在内的所有数据都汇集进入路由器的转发队列，路由器按照转发队列中的顺序来交错地发送这些来自不同主机的数据，所以单从来自不同主机的数据而言，其转发时的发送速度必定小于所有从路由器转发出去的数据的发送速度（即该交换节点的出口带宽）。

因为带宽是有限的，它毫无疑问是个抢手资源，而且互联网运营商也不会白白搭建网络，所以运营商在所有的基础交换节点上设置关卡，也就是限制数据从你的主机流入路由器转发队列的速度，而只要流入路由器转发队列的数据，都会按照路由器的出口带宽，流入其他网络。

这种关卡设置实际上等于限制了你的主机发送数据的速度，也就是限制了主机的出口带宽，而至于这种限制的实现机制，我想你已经很清楚了，那就是通过限制交换机对于你主机的数据接收速度，来将你的发送速度牢牢控制在手，因为数据链路层的流量控制是通过控制接收方来实现的。对于交换机的限速设置，IDC 的网管非常擅长。

共享还是独享

一切都清楚了，下面我们来看看两个被交换机限制了带宽的主机，它们都安装了 MRTG，可以生成网卡流量报告单，不过我们在这里关心的不是流量，而是报告单顶部的一段信息，请注意这里的 Max Speed 属性值，它便是从交换机接收端口获得的最大接收速度，同时也是该主机的最大数据发送速度，但并不一定是此刻的实时发送速度，因为每时每刻的发送速度都是传输协议根据接收方的接收能力不断调整的，比如通过数据链路层或者传输层的滑动窗口技术等流量控制机制进行速度的调整。

如表 2-1 所示，这台主机使用了独享 10M 带宽，也就是 10Mbit/s 的数据发送速度，换算成字节的话，正好就是上面的 1250.0kBytes/s。在这种情况下，如果路由器的出口带宽为 100M，交换机的设置应该保证来自广播域内其他主机的数据发送速度总和不超过 90Mbit/s，以保证该主机任何时刻都可以以 10Mbit/s 的速度发送数据，这才叫独享带宽，它独享的是路由器的一部分出口带宽，而不是交换机的带宽，因为交换机本来就是各个端口独享带宽而互不影响。

表 2-1　通过 MRTG 查看独享 10M 带宽的服务器

System:	s-colin in Server Room
Maintainer:	Sysadmin (root@localhost)
Description:	eth0
ifType:	ethernetCsmacd (6)
ifName:	eth0
Max Speed:	1250.0 kBytes/s
Ip:	192.168.1.2

如果这台交换机还为其他主机提供独享 10M 带宽的接入，并且路由器的出口带宽为 100M，那么理论上总共只能接入 10 台主机，这样才可以保证每台主机的实际带宽总是 10M。假设带宽运营商为了多赚钱，给该交换机上接入了 20 台主机，然后对每台交换机仍然都限制了 10M 带宽，这时候从 MRTG 的 Max Speed 属性上仍然看到的是 1250.0 kBytes/s，但是你可以观察 MRTG 流量图或者使用 Nmon 实时流量监控来分析自己的 10M 带宽是否真的有所保证。

表 2-2　通过 MRTG 查看共享 100M 带宽的服务器

System:	*s-cris in Server Room*
Maintainer:	Sysadmin (root@localhost)
Description:	eth0
ifType:	ethernetCsmacd (6)
ifName:	eth0
Max Speed:	12.5 MBytes/s
Ip:	192.168.1.3

如表 2-2 所示，另一台主机使用了共享 100M 带宽，也就是 100Mbit/s 的数据发送速度，换算成字节便是 12.5MBytes/s。事实上很多普通的服务器托管都采用这种带宽方案，它的价格较低，但是请注意这里的“共享”是指交换机不保证你的主机出口带宽能达到 100M，原因很简单，假设交换机为 50 台主机同时提供共享 100M 带宽接入，每台主机理论上都能以最高 100Mbit/s 的速度将数据发送到交换机，但是这些数据汇集到路由器后，都得从路由器的出口转发到其他网络，可是路由器的出口带宽是运营商买来的，假设这里也就 100M 而已，那这些数据就得在路由器的高速缓冲区中以 100Mbit/s 的速度转发，那么交换机也要配合这个速度来调整自己的发送速度，同时每台主机也会自动调整发送速度，这些都是由底层传输协议自动完成的，本质是为了防止发送速度大于接收速度导致接收端缓冲区的数据还没来得及处理就被新的数据覆盖。

这样一来，每台主机的实际带宽就要根据这些主机的总体通信量来决定了，假如 50 台主机都要发送大量的数据（大文件下载），那么每台的发送速度大概控制在 2Mbit/s 左右，也就是 250kBytes/s；假如有 49 台主机在某时刻不发送数据，则剩余的 1 台主机此刻理论可以达到 100Mbit/s 的速度，但是一般交换机对于共享带宽都会限制最高峰值，比如限制为 10Mbit/s，那么即使某个时刻只有 1 台主机需要发送数据，交换机也会让它减速。

以上说的这些发送速度（即带宽）都是针对从发送端发出的所有数据而言，但是这些数据往往来自于主机的多个进程，而且也要发向五湖四海不同的目的地，比如北京、上海和深圳的 3 个用户通过 HTTP 同时下载主机上的数据文件，这些数据不分你我地夹杂在一起从主机进入路由器，这一过程的发送速度前面已经讲得很清楚了，当它们进入某城市交换节点的时候，大家分道扬镳，分别前往三个目的地，这时从各个目的地的角度而言，单位时间接收到的数据一定比从服务器发出的数据少，这种从最终接收端体验到的速度我们一般称为“下载速度”，也就是我们经常看到在浏览器下载进度条上显示的数值，可见下载速度无论如何都小于发送速度。在刚才的例子中，即便是三个目的地的下载速度总和也一定小于服务器对这些数据的发送速度，下一节将详细探讨数据在整个传输过程中都消耗了哪些时间，它将和本节的带宽概念一起，直接关系到高性能 Web 站点架构的设计。

顺便一提的是在一般情况下当你使用普通的家庭宽带接入互联网时，宽带服务商往往会在你接入的交换节点上进行带宽限制，比如你申请安装了 1M 独享宽带，那么交换节点上连接到你家的端口缓冲区将会以 1Mbit/s 的速度接收来自服务器的数据，这样你的下载速度将受到限制，但是服务器的数据发送速度基本不变，因为它发出的数据不只是到你一家。

还记得本节开头时用高速公路来形容带宽吗？到这里我想你已经非常清楚用高速公路的宽度来形容带宽真是毫无道理，但是，我认为高速公路的入口收费站有点带宽的意思，结合本节的介绍，你想想是不是这样呢？

2.3 响应时间

数据从服务器到用户 PC 的传输过程中，翻山越岭（我曾经在北京郊区的山上看到过中国铁通的光缆铺设提示牌），走过漫长的距离，流过若干个交换节点，并被不断地转发，然而用户注定是只看功劳不看苦劳，上一节中提到的“下载速度”就是一个关注结果的概念，而服务器以及各交换节点的“发送速度”则可以说是一个关注过程的概念。

下载速度

所谓的下载速度，就是指单位时间里从服务器到达用户 PC 的数据量的多少，一般用数据量的字节数多少来描述，所以下载速度的单位为“Bytes/s”。由此可见，计算下载速度的方法是：用传输的数据字节数，除以这些数据从服务器开始发送直到完全到达用户 PC 的时间。形象地说，这段时间是从数据的第一个比特由服务器网卡进入线路开始，直到最后一个比特位进入用户 PC 的网卡为止。以上的计算方法在任意长度的时间片段中都成立，如果时间片非常短，那么计算结果就是我们一般说的实时下载速度。

数据从服务器开始发送直到完全到达用户 PC 的这段时间，我们称为“响应时间”。

不论是大文件下载，还是网页、图片、样式表的下载，其下载速度都是站点用户最关心的指标，也是用户唯一能体验到的站点性能，所以如果能估算出各地用户的下载速度，并根据它来决策服务器的位置和带宽，是非常有意义的。在通常情况下，我们很清楚也很容易计算传输的数据量大小，所以只有搞清楚响应时间的计算方法，才可以计算出下载速度。

如何计算响应时间

通过前面的介绍，我们了解了互联网上两台主机之间数据发送和传输的整个过程，事实上，数据的响应时间不难得出：

响应时间 = 发送时间 + 传播时间 + 处理时间

发送时间很容易计算，即“数据量/带宽”。比如要发送 100Mbit 的数据，而且发送速度为 100Mbit/s，也就是带宽为 100M，那么发送时间便为 1s。值得注意的是，在两台主机之间往往存在多个交换节点，每次的数据转发，都需要花费发送时间，那么总的发送时间也包括这些转发时所花费的发送时间。

传播时间主要依赖于传播距离，因为传播速度我们可以近似认为约等于 2.0×10^8m/s，那么传播时间便等于传播距离除以传播速度。比如两个交换节点之间线路的长度为 1 000km，相当于北京到上海的直线距离，那么一个比特信号从线路的一端到另一端的传播时间为 0.005s。

处理时间是什么意思呢？其实在之前的介绍中，虽然没有提出这个概念，但是已经包含了对其本质的介绍。简单地说，处理时间就是指数据在交换节点中为存储转发而进行一些必要的处理所花费的时间，其中的重要组成部分就是数据在缓冲区队列中排队所花费的时间，注意，准确地说应该是“你的数据”在队列中排队所花费的时间，因为在队伍中还有其他与你毫不相干的数据。此时，如果你想起在介绍带宽的时候我们提到的共享带宽，那就对了。

如果全世界只有你的服务器和你的用户在传输数据，那么用于排队的处理时间可以忽略。

可见，处理时间的多少，取决于数据流经各交换节点所在网络的数据通信量，它往往是不可预测的，所以它的计算比较复杂，往往没有一个简单的数学计算公式，而是依赖于多变的外部因素，必须结合实际情况具体分析。

那么，我们可以将响应时间的计算公式整理为：

响应时间 =(数据量比特数 / 带宽) + (传播距离 / 传播速度) + 处理时间

另外，下载速度的计算公式如下：

下载速度 = 数据量字节数 / 响应时间

有了以上的计算方法，下面我们还是在具体场景中试着来计算响应时间，注意，这里为了计算，我们暂时先忽略处理时间。

一些习惯和约定

另外，一般我们习惯将比特的单位 bit 缩写为 b，而将字节的单位 Byte 缩写为 B，那么显然 1B=8b。同时，比特和字节的换算也不一样，约定如下：

$1KB = 2^{10}B = 1024B$

$$1kb = 10^3b = 1000b$$

对于 M 和 K 的换算，同样如此。

但是请注意，在以下的计算中为了方便，我们将认为：

$$1KB \approx 10^3B = 1000B$$

这样引发的误差对于响应时间的计算结果来说微不足道。

来一次实地计算

假设这样一个场景，我们的 Web 服务器托管在北京的某互联网数据中心（IDC），以 10M 独享带宽的方式接入互联网。位于西安的一位用户通过小区提供的 1M 独享带宽的方式接入互联网，他通过 PC 的浏览器下载该 Web 服务器上的一个 100MB 大小的文件，换算成比特也就是 800Mb，响应时间和下载速度是多少呢？

在计算之前，我们首先得清楚数据在整个传输过程中流过的路径，也就是经过了多少交换节点，这些节点分别提供多大的带宽。要知道数据流过的路径，一般我们使用 Linux 的 traceroute 命令或者 Windows 的 tracert 命令，它通过 IP 包头部的 TTL 数值，在默认情况下分别发送 40 字节的测试数据到沿路的每一个交换节点，以此来追踪数据经过的路由路径，同时测算数据到达每一个交换节点的响应时间。但是由于 40 字节的默认测试数据量比较小，最多也只能设置几 KB，而且这种测算机制的实现容易受网络不确定因素影响，所以它的测算结果一般用于检测故障，而对于我们这里计算整个过程的响应时间意义不大，我们的目的是通过前面基于带宽的计算模型来了解响应时间在理论上的决定因素，并希望以此来帮助我们认识到影响下载速度的关键环节。

为了在这个具体的场景中充分说明时间的花费，我们先假设服务器和用户 PC 之间的交换节点很少，然后逐步增加。

首先，假设 Web 服务器直接连接在交换节点 A，那么从 Web 服务器到交换节点 A，因为使用了 10M 独享带宽，所以数据发送速度为 10Mbit/s，所以这部分发送时间为：

$$800Mbit / (10Mbit/s) = 80s$$

然后，用户的接入方式为 1M 独享带宽，所以用户 PC 接入的交换节点 B 到用户 PC 的发送速度为 1Mbit/s，所以这部分发送时间为：

$$800Mbit / (1Mbit/s) = 800s$$

而至于以上两个交换节点 A 和 B，我们假设它们都是所在城市的城域网顶级节点，而且它们直接通过光缆相连，带宽假设为 40G，那么这部分发送时间为：

$$800Mbit / (40Gbit/s) = 0.02s$$

以上的发送时间相加就是总的发送时间。

接下来，我们计算传播时间，由于北京到西安的高速公路距离大约为 1000km，所以我们假设 Web 服务器到用户 PC 的线路距离总和为 1000km，传播速度我们统一按照 2.0×10^8m/s 计算，那么传播时间为：

$$1000km / (2.0\times10^8 m/s) = 0.005s$$

这样一来，忽略处理时间，响应时间便为：

$$80s + 800s + 0.02s + 0.005s = 880.025s$$

而下载速度便为：

$$100MByte / 880.025s = 113.63KB/s$$

看来非常不错，我们终于算出了下载速度，但这并不是实际的情况，因为我们忽略了太多其他的交换节点。下面我们考虑增加一些交换节点，正是这些节点使得这 100MB 的文件数据流经各个网络后抵达用户 PC，因为前面提到的两个交换节点 A 和 B 之间几乎不可能直接用光缆连接，而是通过一系列的交换节点层层相连。

继续假设，Web 服务器直接连接的交换节点 A 并不是北京的顶级节点，而是服务器所在机柜的交换节点，它还连接着更高一级的交换节点 C。我们假设节点 A 的出口带宽为 100M，事实上在 IDC 的一些机柜正是使用 100M 的出口带宽，为机柜内的多台服务器提供独享或共享带宽，而这 100M 带宽则是代理商从上一级交换节点 C 那里花钱买来的。那么，从节点 A 到节点 C 增加的这部分发送时间为：

$$800Mbit / (100Mbit/s) = 8s$$

然而，在数据流经节点 C 之后，假设还经过了节点 D 和节点 E，才到达了北京的顶级节点。节点 D 和节点 E 可以认为是 IDC 中用于组建网络而设置的节点，因为服务器实在太多了，必须划分多个子网才能够接入网络，实际上有些 IDC 可能由于设计不良或者历史问题，内部要经过更多的交换节点转发，这可以通过 traceroute 命令来跟踪了解。为了给 IDC 中各服务器提供承诺的带宽，我们假设节点 C、节点 D 和节点 E 之间采用光纤连接，出口带宽分别为 1G 和 10G，而现实中这也毫不夸张。那么这两部分发送时间为：

800Mbit / (1Gbit/s) + 800Mbit / (10Gbit/s) = 0.88s

这样一来，响应时间增加到了 888.905s，下载速度下降到 112.50KB/s。

实际上，当数据到达北京的顶级节点后，并不是直接转发到西安的顶级节点，而是可能要经过更多的国家基础节点，比如省网核心节点以及国家骨干网络核心节点等，但是这些节点的带宽可以认为至少都在 10G 级别之上。而当数据到达西安顶级节点后，又要经过多个节点才能到达连接用户 PC 的交换节点 B，这些交换节点都是国家的重点基础设施，带宽多数都在千兆以上，所以这部分增加的发送时间就像前面增加的 8.88s 一样，对于下载速度的影响不值一提，所以我们暂且忽略不计。

可见，即使增加了交换节点的数据转发次数，但是如果这些节点可以保证较高带宽的话，那么对于最终下载速度的影响其实微不足道，这也就是为什么在一些基础节点以及骨干节点必须要使用较高带宽，比如某 IDC 采用 40G 出口带宽直接连接国家计算机网络核心路由器节点。

那么，这个 112.50KB/s 的下载速度真的就是实际速度吗？绝对不是。不要忘了我们在以上的计算过程中忽略了很多可能没有提到过的非理想因素，这些因素在实际环境下是不可避免的，它们将导致实际下载速度必然或多或少地低于 112.50KB/s。注意这里说的实际速度，其实就是我们用浏览器或其他下载工具时看到的下载速度值。

这些非理想因素主要包括：

- 在共享带宽以及网络通信数据量过大时，交换节点中数据在转发队列中的等待时间便不能忽略，这部分时间往往也是影响下载速度的重要部分。大家应该都有亲身体验，比如几台 PC 通过同一宽带路由器上网，如果其中一台 PC 正在持续下载电影，那么其他几台 PC 的下载速度必然受到较大的影响。
- 有时候我们下载的数据需要直接写入磁盘，比如下载几百兆的文件到某文件目录下。我们知道，PC 网卡接收到的数据进入内存后，便完成了数据接收，但是下载进程一般要将这些数据即时写入磁盘后，才会进行下一次接收数据的系统调用，而写入磁盘的过程需要经历位于内核态内存中磁盘高速缓冲区的转发，而且有些下载进程（如 IE 的文件下载）为了减少磁盘写操作的压力，会在每次接收数据的系统调用之间进行休眠停顿，这样一来，实际上单线程的下载过程中接收数据并不是绝对连贯的，所以我们一般喜欢用多线程下载工具来加速下载，就是因为多个执行流可以使得下载过程在单位时间内执行相对更多次数的数据接收系统调用，充分占用带宽。
- 在一些长距离的传播线路（如跨城光缆）中必须使用中继器，以此防止电磁波信号在传播中过分衰减，所以这些中继器需要对信号进行接收和转发，也存在发送

时间的消耗，但是不多。

- 为了容易描述网络结构，在前面我们经常提到的“交换节点”普遍特指“交换路由器”，也就是一个设备同时具备路由器和交换机的功能，很多人在家里使用的宽带路由器便是这样的节点设备。而有些交换节点，可能采用了独立的路由器和交换机，那么一旦数据需要经其转发到其他网络，这个交换节点便需要对数据进行两次转发，也就是转发次数翻倍，不过不必担心，和上边的中继器转发一样，这部分时间的确也不是很多，但是有必要说明这些可能存在的环节。

幸运的是，以上这些因素大多数都显得微不足道，虽然用户 PC 所在网络内通信数据量过大时导致用户 PC 之间带宽彼此影响，但这毕竟是用户考虑的问题，已经超出了在服务器方面所能解决的问题范围，而服务器的接入在允许的情况下尽量使用独享带宽即可在一定程度上减少环境因素的影响。

所以，要搞清楚响应时间的消耗，必须根据实际情况，找出转发路径中是否存在一些较低带宽的交换节点，正是这些节点成为影响下载速度的瓶颈所在。

2.4 互联互通

同样，在实际的互联网中，往往这些瓶颈出现在各互联网运营商之间的网络互联上，我们称为“互联互通”。在我们前面的假设场景中，认为服务器和用户 PC 都处于同一个运营商的互联网中，由于两个主机处于两个城市，所以最终都要经过该运营商的骨干网络核心节点进行转发。如果服务器和用户 PC 在同一个城市且处于同一运营商的互联网中，那么转发的路径将大大减少，至少不用进入骨干网络。

但是，如果服务器和用户 PC 处于不同运营商的互联网中，那么不论是否在同一城市，数据都必须经过两个互联网运营商之间的互联节点，这个节点的带宽变成了令大家十分头疼的问题。

在中国，由中国电信运营的互联网，也就是我们常说的“中国宽带互联网（CHINANET）”，它的骨干网络核心节点位于北京上地电信数据中心，它通过直接接入包括北京在内的国内 8 个重要城市节点，进而连接二级网络，然后层层延伸扩展，一直到周边城市、IDC、家庭宽带接入等。这 8 座城市为北京、沈阳、西安、成都、上海、南京、广州、武汉。这些城市直接与骨干网络连接的 IDC 都是国家一级设施，除了提供商业服务以外，还有重要的战略意义。

另外，由中国网通运营的互联网，它的骨干网络核心节点位于北京亦庄网通数据中心，其网络也几乎覆盖全国所有省份的经济发达城市，具体覆盖情况大家可以查询最新的有关资料。还有其他的一些互联网运营商比如中国联通、中国铁通、中国教育网等，都有自己的专用线路和接入用户。

目前，电信和网通两大互联网运营商之间的互联互通虽然早已实现，但是互联带宽少得可怜。为什么不根据需要大力扩容呢？简单地说，两大运营商对互联互通的问题存在着不同的观点，涉及利益问题和政治问题，但是在政府主管部门的推动下，互联扩容的谈判和讨价还价一直在艰难地进行，去年适逢奥运，“奥运网络保障”被提到了压倒一切的高度。在这种形势下，电信和网通间已经实现了大规模的互联扩容，互联带宽扩展到将近 200G，应该说已经是相当大的进步了，但是区区 200G 带宽对于中国两大运营商互联网之间的频繁数据交换来说仍远远不够。

不可否认，国内互联网互联互通上的限制，给国家和企业资源带来巨大的浪费。在 2008 年的“两会”上，曾经有提案建议，国家应该尽快出台《互联网互联互通法》，或在酝酿中的《电信法》中加入互联网互联互通的部分。某政协委员用形象的比喻说：“两条高速公路用两根木板搭起来连接，怎么可能实现互联互通？”

以上只是简单介绍了中国互联网互联互通的现状，实际情况更加复杂，而且在不断地变化，大家可以根据需要查阅最新的详细资料。正是因为互联网互联互通上的限制在未来相当长的一段时间内不可能被彻底打破，所以在我们的 Web 站点架构设计中不得不意识到它的存在，而且要将它视为一个不可忽视的架构组成部分，本书后面的很多内容都会涉及它，比如分布式系统以及负载均衡等内容。

的确，我们的站点用户绝不可能处于同一个互联网运营商的网络中，而事实上即使国内的互联网可以高速互联，如果我们的站点用户覆盖全球，并且要保证高速服务，那么跨国运营商互联和国际出口带宽依然是残酷存在的问题，幸运的是这些问题都可以抽象为同类问题来考虑。

归根结底，希望通过本节的介绍，可以让大家清晰地认识到响应时间和下载速度的本质和计算方法，而至于究竟将服务器部署在哪里的问题完全要通过大家自己的考察得出结论，比如选择 IDC 的时候要考察出口带宽以及与骨干网络是否直连，如果要同时为多个互联网运营商网络的用户提供服务，则需要考察出口节点与运营商互联节点的带宽，而如果要面向全球用户提供服务，则需要考察国际网络结构和各个国家的国际出口带宽等。另一方面，带宽作为稀缺资源，其价格严格服从市场供求规律，比如同样的独享带宽在北京就比沈阳贵很多，所以我们在选择的时候价格因素也相当重要。

第 3 章

服务器并发处理能力

人们总是希望少花钱多办事，同样的，一台 Web 服务器在单位时间内能处理的请求越多越好，这也成了 Web 服务器的能力高低所在，它体现了我们常说的“服务器并发处理能力”。值得一提的是，本章所说的服务器，主要指用于提供 HTTP 服务的服务器，但是本章中所涉及的一些关于操作系统和内核的内容，并不局限于 Web 服务器。

3.1 吞吐率

说到 Web 服务器的并发处理能力，那就一定得有一个量化的描述，我们一般使用单位时间内服务器处理的请求数来描述其并发处理能力，但是听起来有点长，我们习惯称其为吞吐率（Throughput），单位是“reqs/s”。需要注意的是，吞吐率这个概念有时候还用于描述其他指标，比如单位时间内的通信数据量等，但是在本书中，吞吐率特指 Web 服务器单位时间内处理的请求数。

在一些常见的 Web 服务器软件中，通常会提供当前服务器运行状况以及吞吐率的查看方法，帮助我们对服务器的吞吐率随时进行了解和监控，比如 Apache 的 mod_status 模块提供了如下统计：

```
Current Time: Thursday, 05-Feb-2009 23:49:48 CST
Restart Time: Sunday, 25-Jan-2009 15:36:53 CST
Parent Server Generation: 3
Server uptime: 11 days 8 hours 12 minutes 55 seconds
Total accesses: 34362196 - Total Traffic: 256.6 GB
CPU Usage: u416.49 s70.17 cu0 cs0 - .0497% CPU load
35.1 requests/sec - 274.5 kB/second - 7.8 kB/request
7 requests currently being processed, 19 idle workers
```

这里的 35.1requests/sec 便是此时的吞吐率，它是从 Apache 启动到目前时刻的平均计算值。而在 Lighttpd 的 mod_status 模块中，可以更好地展示最近 5s 的吞吐率，统计数据如表 3-1 所示。

表 3-1 通过 mod_status 查看 lighttpd 的实时吞吐率

Hostname	localhost ()
Uptime	41 days 23 hours 5 min 3 s
Started at	2008-12-26 00:52:12

续表

absolute (since start)	
Requests	439 Mreq
Traffic	1.28 Tbyte
average (since start)	
Requests	121 req/s
Traffic	379.50 kbyte/s
average (5s sliding average)	
Requests	165 req/s
Traffic	335.09 kbyte/s

它分别展示了从 Lighttpd 启动到目前时刻的吞吐率以及最近 5s 的吞吐率，还包括单位时间的流出数据量。

吞吐率和压力测试

单从定义来看，吞吐率描述了服务器在实际运行期间单位时间内处理的请求数，然而，我们更加关心的是服务器并发处理能力的上限，也就单位时间内服务器能够处理的最大请求数，即最大吞吐率。所以我们普遍使用“压力测试”的方法，通过模拟足够数目的并发用户数，分别持续发送一定的 HTTP 请求，并统计测试持续的总时间，计算出基于这种“压力”下的吞吐率，即为一个平均计算值。

> **提示：**
>
> 在后面的内容中，我们对吞吐率和最大吞吐率不做特意区分，请大家根据上下文来理解。

另一方面，在 Web 服务器的实际工作中，其处理的 HTTP 请求通常包括对很多不同资源的请求，也就是请求不同的 URL，比如这些请求有的是获取图片，有的是获取动态内容，显然服务器处理这些请求所花费的时间各不相同，而这些请求的不同时间的组成比例又是不确定的，那么我们在压力测试的时候是否要模拟这种不同请求交错的情况呢？你也许会说这才是实际情况下的吞吐率，可以帮助我们了解服务器在实际情况下的并发处理能力。

没错，在有些时候我们需要了解服务器在业务环境中实际的处理能力，但是这些不同性质的请求交错在一起，形成的资源需求模型过于复杂，也就是对于 CPU 处理和 I/O 操作的需求涉及太多因素，给我们考察服务器并发处理能力带来了难以想象的困难。我们知道在一个含有较多变量的数据模型中求解最优化结果是非常困难的，所以我们一般都简化模

型，对同一个特定的有代表性的请求进行压力测试，然后根据需要，对多个请求的吞吐率按照比例计算加权平均值。

往往也正是因为这些请求性质的不同，Web 服务器并发能力强弱的关键便在于如何针对不同的请求性质来设计最优并发策略，这在后续章节中会有详细介绍。同时也是因为有时候一台服务器要同时处理诸多不同性质的请求，在一定程度上使得 Web 服务器的性能无法充分发挥，这很容易理解，就像银行对不同业务设立不同的窗口一样，这些窗口的服务员分别熟悉自己的窗口业务，可以为不同的客户分别快速办理业务，但是如果让这些窗口都可以办理所有业务，也就是客户可以去任何窗口办理任何业务，那会怎么样呢？我想没有几个银行业务员会对所有业务都轻车熟路，这样势必会影响到整体的业务办理速度。

压力测试的前提条件

这么一来，我们要统计吞吐率，便存在一些潜在的前提，那就是压力的描述和请求性质的描述。

压力的描述一般包括两部分，即并发用户数和总请求数，也就是模拟多少个用户同时向服务器发送多少个请求，随后会有关于并发用户数的详细介绍。

请求性质则是对请求的 URL 所代表的资源的描述，比如 1KB 大小的静态文件，或者包含 10 次数据库查询的动态内容等。

所以，吞吐率的前提包括如下几个条件：

- 并发用户数
- 总请求数
- 请求资源描述

并发用户数

前面提到了并发用户数，在我们开始进行压力测试之前，一定要弄明白它的含义。

简单地说，并发用户数就是指在某一时刻同时向服务器发送请求的用户总数。如此多的用户同时请求服务器，显然会给服务器带来不小的压力，这时你可能有一个实际的问题，假如 100 个用户同时向服务器分别进行 10 次请求，与 1 个用户向服务器连续进行 1000 次请求，效果一样吗？也就是说给服务器带来的压力一样吗？

虽然看起来服务器都需要连续处理 1000 个请求，其实关键的区别就在于，是否真的“连续”。首先有一点需要明白，对于压力测试中提到的每一个用户，连续发送请求实际上是

指在发送一个请求并接收到响应数据后再发送下一个请求。这样一来，从微观层面来看，1 个用户向服务器连续进行 1000 次请求的过程中，任何时刻服务器的网卡接收缓冲区中只有来自该用户的 1 个请求，而 100 个用户同时向服务器分别进行 10 次请求的过程中，服务器网卡接收缓冲区中最多有 100 个等待处理的请求，显然这时候服务器的压力更大。

其实，并发用户数这个词你也许经常听到，很多时候它还有一些别名，比如并发数、并发连接数等。经常会有人说某个 Web 服务器能支持多少并发数，除此之外没有任何上下文，这让很多人摸不着头脑，人们常常把并发用户数和吞吐率混淆，它们并不是一回事，通过前面的介绍，我们很清楚，吞吐率是指在一定并发用户数的情况下，服务器处理请求能力的量化体现。

那么，说一个服务器最多支持多少并发用户数，这个“最多”到底是什么意思？这里我们暂且抛开技术的因素，从广义的角度来看，举个生活中的例子，比如某商城里有一个柜台，给顾客们办理业务。刚开始顾客稀少，一次只来一个顾客，柜台业务员很轻松就可以搞定，不久，有很多顾客去柜台办理业务，大家排成一个长队依次办理，每个顾客在办理业务的过程中，都需要花时间填写一些资料，这时候其他顾客就得等着，而且柜台业务员闲着无聊又不能干别的事情，所以他觉得很浪费时间，就让大家排成两队，这样在等待的时候就给另一个队办理。可是填写资料的时间有点长，另一队的顾客开始填写资料时，前一个队的顾客还没填完，业务员还是得等。最后队伍增加到了 10 队，10 个人同时办理业务，刚好业务员不用等待任何顾客了，而且每个顾客对办理速度也很满意。

可是随着前来办理业务的顾客越来越多，业务员想做点有挑战性的事情，他将队伍增加到了 20 队，同时给 20 个顾客办理业务，这下可不得了，20 条队伍拥挤不堪，原本轻松的保安为了维持秩序累得气喘吁吁，这还不算，关键是业务员快疯了，他的办理速度已经赶不上大家填写资料的速度了，一些人填完资料后没人搭理，便开始抱怨，业务员也因为同时要给很多人办理业务，脑子反应不过来，导致处理原本熟悉的业务时也变得手忙脚乱，效率下降并且时常伴随一些低级失误。

终于，在大家的一片声讨下，柜台崩溃了。注意，柜台的崩溃不是因为业务员无法继续工作了，虽说他忙得累死累活，但是活儿还得干啊，关键是顾客们等得不耐烦了，大部分顾客都无法忍受长时间的等待以及办理过程中出现的错误，所以投诉到了商城的管理处，商城只好暂时关闭柜台业务，商讨对策。

很快，商城又恢复了柜台业务，将柜台的队伍数调整到 10 队，同时根据顾客流量，临时增设一定数量的柜台，这样一来，顾客们纷纷表示满意。

我们可以说，这个柜台支持的最大并发数为 10，因为恰好在这个并发数下，柜台业务开展得非常成功，顾客们都对服务时间非常满意，而且此时代表业务办理次数的柜台吞吐率也比较高，商城和顾客实现双赢。当并发数少于 10 的时候，柜台业务员的时间得不到充

分利用，浪费了柜台的宝贵资源，这时候的吞吐率要低一些。而当并发数大于 10 的时候，事实证明，顾客们不乐意了。这样看来，问题的本质变得非常清晰，似乎就是商家和顾客的博弈，而且是合作型博弈，最大并发数便是博弈的结果，也是最大程度的共赢。

可见，通常所讲的最大并发数是有一定利益前提的，那就是服务器和用户双方所期待的最大收益，服务器希望支持高并发数及高吞吐率，而用户不管那么多，只希望等待较少的时间，或者得到更快的下载速度，显然，双方不可能都彻底满足，所以便存在讨价还价的余地，同时双方也都有能够忍受的最低尺度。从经济学的角度看，就这么简单，所以找到双方利益的平衡点，便是我们所希望的最大并发用户数。这种现象在后续章节中介绍下载服务的时候还会提到。

当然，经过权衡后，我们所希望的最大并发用户数，还存在一定的技术制约，这也是狭义层面的最大并发数的定义。柜台的故事只是一个简单的模型，而在我们访问实际的 Web 站点时，每个请求的处理过程可并不像柜台业务员给顾客办理业务那么简单，尤其是在并发用户数较大的情况下，Web 服务器使用什么样的并发策略，是影响最大并发数的关键。

另外值得说明的是，即使我们通过压力测试得出服务器支持的最大并发数，但这与实际并发用户数却是两回事，因为有多少用户同时发来请求并不是服务器所能决定的，该来的总是要来，那是客观存在的。一旦实际并发用户数大于服务器所能支持的最大并发数，那必然造成一部分用户需要等待超过预期的时间，影响了站点的服务质量。所以，得出最大并发数的意义，在于了解服务器的承载能力，并且结合用户规模考虑适当的扩展方案。从站点的某些商业角度来看，最大并发用户数的支持程度往往比吞吐率更加容易理解。

在考虑实际用户规模的时候，我们还需要了解一点，用户访问 Web 站点通常使用浏览器，而浏览器在下载一个网页以及网页中的多个组件时，采用多线程的并发下载方式，但是对于同一域名下 URL 的并发下载数是有最大限制的，具体限制视浏览器的不同而不同，比如在 HTTP/1.1 下，IE7 支持两个并发连接，IE8 支持 6 个并发连接，Firefox3 支持 4 个并发连接，我们使用诸如 HttpWatch 这样的 HTTP 监视工具可以很清晰地看到这一点。所以，我们前面说到的服务器支持的最大并发数，具体到真实的用户，可能并不是一对一的关系，一个真实的用户可能会给服务器带来两个或更多的并发用户数的压力，一些高明的用户还可以通过一些方法来修改浏览器的并发数限制。而我们在本书中为了简化模型，暂且认为每个用户的并发下载数均为 1。

另一方面，从 Web 服务器的角度来看，实际并发用户数也可以理解为 Web 服务器当前维护的代表不同用户的文件描述符总数，也就是并发连接数，当然，不是同时来了多少用户请求就建立多少连接，Web 服务器一般会限制同时服务的最多用户数，比如 Apache 的 MaxClients 参数，所以这个实际并发用户数，有时候大于服务器所维护的文件描述符总数，而多出的这些用户请求，则在服务器内核的数据接收缓冲区中等待处理，所以这些请求在用户看来处于阻塞状态。

但是，最大并发用户数和最大并发连接数的决定因素从本质上来说是不同的，举两个例子：

- 当实际并发用户数稍稍大于服务器所能维护的文件描述符上限时，如果请求的性质决定了处理每个请求花费的时间非常少，比如请求 1KB 的静态网页，那么每个请求都可以快速被处理然后释放文件描述符，这样从用户的角度而言，等待时间几乎不会减少太多。所以在这种情况下，我们希望服务的最大并发用户数可以大于最大并发连接数。幸运的是，这种情况在我们后边介绍 select 模型在大并发下处理小文件请求时会有相应的测试。
- 如果请求的性质决定了处理每个请求要花费相当长的时间，比如下载 10MB 文件或者请求动态内容，那么即使服务器可以支持较大的并发连接数，比如使用异步 I/O 理论上可能支持 2 万个并发连接，然而是否能够为这么多接入的用户提供快速响应的服务至关重要。对于下载 10MB 文件来说，可能由于带宽的瓜分而导致每个用户的下载速度缓慢，而对于请求动态内容，可能由于 CPU 的时间瓜分导致每个用户的等待时间过长。所以在这种情况下，我们希望服务的最大并发用户数小于理论上的最大并发连接数。

这样看来，从某种意义上可以说，Web 服务器所做的工作的本质就是，争取以最快的速度将内核缓冲区中的用户请求数据一个不剩地都拿回来，然后尽最大努力同时快速处理完这些请求，并将响应数据放到内核维护的另一块用于发送数据的缓冲区中，接下来再尽快处理下一拨请求，并尽量让用户请求在内核缓冲区中不要等太久。

通过 Apache 的 mod_status，可以了解到它在此刻同时处理的请求数，也就是此刻的实际并发用户数，如下所示：

```
Current Time: Thursday, 05-Feb-2009 23:49:48 CST
Restart Time: Sunday, 25-Jan-2009 15:36:53 CST
Parent Server Generation: 3
Server uptime: 11 days 8 hours 12 minutes 55 seconds
Total accesses: 34362196 - Total Traffic: 256.6 GB
CPU Usage: u416.49 s70.17 cu0 cs0 - .0497% CPU load
35.1 requests/sec - 274.5 kB/second - 7.8 kB/request
7 requests currently being processed, 19 idle workers
```

请求等待时间

在刚才讲到并发用户数的时候，我们简单地提到了用户等待时间，这也是压力测试结果中很重要的一个指标。总结一下，我们所关心的时间有以下两种：

- 用户平均请求等待时间
- 服务器平均请求处理时间

听起来好像比较绕口，但是我实在找不出更短并且可以完整表达其含义的词语。它们的本质区别是什么呢？其实在前面商城柜台的故事中已经有所涉及，我们这里再来举个例子，注意，我们暂时将数据在网络上的传输时间不计入内。

首先，假设并发用户数为 1，也就是只有一个用户在向服务器源源不断地发送请求，那么每个请求的等待时间也就是它的处理时间，等于总时间除以总请求数，这时用户平均请求等待时间和服务器平均请求处理时间是相同的，这很容易理解。

然后，假设并发用户数为 100，那么便会有 100 个用户同时向服务器发送请求，简单地说，这时 Web 服务器一般会采用多进程或多线程的并发模型，通过多个执行流来同时处理多个并发用户的请求，而多执行流体系的设计原则便是轮流交错使用 CPU 时间片，所以每个执行流花费的时间都被拉长。对每个用户而言，每个请求的平均等待时间必然增加；而对于服务器而言，如果并发策略得当，每个请求的平均处理时间可能减少。

所以，这两个时间的本质在于，用户平均请求等待时间主要用于衡量服务器在一定并发用户数的情况下，对于单个用户的服务质量；而服务器平均请求处理时间与前者相比，则用于衡量服务器的整体服务质量，它其实就是吞吐率的倒数。

硬件环境

到目前为止，我们似乎还一直未提及服务器硬件配置，的确，试想一下，如果在商场柜台的故事中，换一个脑子反应快一点的营业员，并且增加柜台前的空间大小，也许可以让柜台支持 20 个并发数。

在以下的压力测试中，我们普遍使用的 Web 服务器基本硬件配置如下：

```
CPU：Intel(R) Xeon(R) CPU 1.60GHz
内存：4GB
硬盘转速：15k/min
```

除此之外我们不对硬件配置做其他详细介绍，因为在本书中，我们探讨的大部分内容都不希望依赖于特定的硬件配置，所以性能测试结果也不侧重于它的绝对数值意义，我们的目的是探讨如何测量性能以及如何根据不同的场景来优化性能。

在后面的压力测试中，如无特殊说明，则均采用以上硬件配置，而对于单机硬件的垂直扩展带来的性能提升，本书不做重点讨论。

来一次压力测试

有了这些前提，我们就可以用压力测试软件来计算吞吐率了，本书中大部分压力测试使用

Apache 附带的 ab，毫不夸张地说，它非常容易使用，完全可以模拟以上各种前提条件。另外，ab 可以直接在 Web 服务器本地发起测试请求，这至关重要，因为我们希望测试的是服务器的处理时间，而不包括数据的网络传输时间以及用户 PC 本地的计算时间，而至于后面这些时间的花费，我们会在本书特定章节中单独介绍，比如在第 2 章中我们已经详细探讨了数据的网络传输本质，这样便有利于我们根据各部分的不同本质来分别分析策略。

需要清楚的是，ab 进行一切测试的本质都是基于 HTTP，所以可以说它是对于 Web 服务器软件的黑盒性能测试，它获得的一切数据和计算结果，都可以通过 HTTP 来解释。

另有一些压力测试软件，包括 LoadRunner、Jmeter 等，则是不同程度上包含了服务器处理之外的时间，比如 LoadRunner 运行在用户 PC 上，可以录制浏览器行为，这种测试的结果往往侧重于站点用户的角度，有另外一些层面的参考意义。

接下来，我们将使用 ab 来进行一次压力测试，在本书中我们使用 Apache 2.2.11 中附带的 ab，其版本信息如下所示：

```
s-colin:/ # ab -V
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/
```

好了，一切就绪，开始压力测试，请看下面的命令行信息：

```
s-colin:/ # ab -n1000 -c10 http://localhost/test.htm
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $> apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.apache.org/
Benchmarking localhost (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Finished 1000 requests
Server Software:        Apache/2.2.4
Server Hostname:        localhost
Server Port:            80
Document Path:          /test.htm
Document Length:        151 bytes
Concurrency Level:      10
Time taken for tests:   0.167781 seconds
```

```
    Complete requests:      1000
    Failed requests:        0
    Write errors:           0
    Total transferred:      423000 bytes
    HTML transferred:       151000 bytes
    Requests per second:    5960.15 [#/sec] (mean)
    Time per request:       1.678 [ms] (mean)
    Time per request:       0.168 [ms] (mean, across all concurrent
requests)
    Transfer rate:          2461.54 [Kbytes/sec] received
    Connection Times (ms)
                  Min  mean[+/-sd] median   max
    Connect:        0    0   0.1      0       1
    Processing:     0    1   0.2      1       2
    Waiting:        0    0   0.1      0       1
    Total:          0    1   0.2      1       3
    Percentage of the requests served within a certain time (ms)
      50%      1
      66%      1
      75%      1
      80%      1
      90%      1
      95%      1
      98%      2
      99%      2
     100%      3 (longest request)
```

请注意我们在启动 ab 时，传入 3 个命令行参数，它们正是代表了前面提到的前提条件：

-n1000

表示总请求数为 1000。

-c10

表示并发用户数为 10。

http://localhost/test.htm

表示这些请求的目标 URL。

测试结果一目了然，我们看到吞吐率显示为 5960.15reqs/s。同时，在测试结果中还有一些其他内容也值得我们关注，主要包括：

Server Software

表示被测试的 Web 服务器软件名称，这里是 Apache/2.2.4，它来自于 HTTP 响应数据的头信息，所以如果是我们自己编写的 Web 服务器软件或者修改开源 Web 服务器软件的源代码，便可以随意改写这里的名称，就像我们曾经都喜欢用游戏修改器修改道具属性一样。

Server Hostname

表示请求的 URL 中的主机部分名称，它来自于 HTTP 请求数据的头信息，这里我们请求的 URL 是 http://localhost/test.htm，所以主机名为 localhost，说明我们的请求是从 Web 服务器端发起的。

Server Port

表示被测试的 Web 服务器软件的监听端口，为了方便测试，我们后面会对多个不同的 Web 服务器软件使用不同的监听端口。

Document Path

表示请求的 URL 中的根绝对路径，它同样来自于 HTTP 请求数据的头信息，通过它的后缀名，我们一般可以了解该请求的类型。

Document Length

表示 HTTP 响应数据的正文长度。

Concurrency Level

表示并发用户数，这是我们设置的参数。

Time taken for tests

表示所有这些请求被处理完成所花费的总时间。顺便提一下，某些 Apache 版本如 2.2.4 附带的 ab，对于这一统计项存在一些计算上的 bug，当总请求数较少时，其统计的总时间会无法小于 0.1s。

Complete requests

表示总请求数，这是我们设置的相应参数。

Failed requests

表示失败的请求数，这里的失败是指请求在连接服务器、发送数据、接收数据等环节发生异常，以及无响应后超时的情况。对于超时时间的设置可以使用 ab 的-t 参数。

而如果接收到的 HTTP 响应数据的头信息中含有 2xx 以外的状态码，则会在测试结果显示另一个名为“Non-2xx responses”的统计项，用于统计这部分请求数，这些请求并不算是失败的请求。

Total transferred

表示所有请求的响应数据长度总和，包括每个 HTTP 响应数据的头信息和正文数据的长度。注意这里不包括 HTTP 请求数据的长度，所以 Total transferred 代表了从 Web 服务器流向用户 PC 的应用层数据总长度。通过使用 ab 的-v 参数即可查看详细的 HTTP 头信息。

HTML transferred

表示所有请求的响应数据中正文数据的总和，也就是减去了 Total transferred 中 HTTP 响应数据中头信息的长度。

Requests per second

这便是我们重点关注的吞吐率，它等于：

```
Complete requests / Time taken for tests
```

Time per request

这便是前面提到的用户平均请求等待时间，它等于：

```
Time taken for tests / (Complete requests / Concurrency Level)
```

Time per request (across all concurrent requests)

这便是前面提到的服务器平均请求处理时间，它等于：

```
Time taken for tests / Complete requests
```

这正是吞吐率的倒数。同时，它也等于：

```
Time per request / Concurrency Level
```

Transfer rate

表示这些请求在单位时间内从服务器获取的数据长度，它等于：

```
Total transferred / Time taken for tests
```

这个统计项可以很好地说明服务器在处理能力达到极限时，其出口带宽的需求量。利用前面介绍的有关带宽的知识，不难计算出结果。

Percentage of the requests served within a certain time (ms)

这部分数据用于描述每个请求处理时间的分布情况，比如在以上测试结果中，80%请

求的处理时间都不超过 1ms，而 99%的请求都不超过 2ms。注意这里的处理时间，是指前面的 Time per request，即对于单个用户而言，平均每个请求处理的时间。

继续压力测试

下面，我们再来进行一次压力测试，此时并发用户数为 100，其他条件不变，测试结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        localhost
Server Port:            80
Document Path:          /test.htm
Document Length:        151 bytes
Concurrency Level:      100
Time taken for tests:   0.157 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      429000 bytes
HTML transferred:       151000 bytes
Requests per second:    6364.24 [#/sec] (mean)
Time per request:       15.713 [ms] (mean)
Time per request:       0.157 [ms] (mean, across all concurrent requests)
Transfer rate:          2666.27 [Kbytes/sec] received
Connection Times (ms)
              min      mean     [+/-sd]      median   max
Connect:      0        2        1.4          1        8
Processing:   4        13       2.1          14       16
Waiting:      2        12       2.3          12       15
Total:        4        15       2.4          15       24
Percentage of the requests served within a certain time (ms)
  50%     15
  66%     16
  75%     17
  80%     17
  90%     18
  95%     18
  98%     20
  99%     22
 100%     24 (longest request)
```

和前一次的测试结果相比，可以看出，当并发用户数从原来的 10 变为 100 后，吞吐率从原来的 5960.15reqs/s 增长到了 6364.24reqs/s，服务器平均请求处理时间从原来的 0.168ms 降到了 0.157ms，而用户平均请求等待时间从原来的 1.678ms 增加到了 15.713ms。

可见，随着并发用户数的变化，吞吐率、用户平均请求等待时间（以下简称请求等待时间）、服务器平均请求处理时间（以下简称请求处理时间）都发生了相应的变化。为了了解变化的曲线，我们对不同并发用户数的情况分别进行压力测试，得出如表 3-2 所示的数据。

表 3-2　对于不同并发用户数的吞吐率测试结果

并发用户数	吞吐率（reqs/s）	请求等待时间（ms）	请求处理时间（ms）
1	4767.23	0.21	0.21
2	5761.74	0.347	0.174
5	5927.33	0.844	0.169
10	6124.31	1.633	0.163
20	6301.81	3.174	0.159
50	6434.29	7.771	0.155
100	6585.86	15.184	0.152
150	6214.67	24.136	0.161
200	1564.6	127.828	0.639
500	280.29	1783.865	3.568

为了直观地分析这些数据，我们生成下列曲线图。首先来看吞吐率随并发用户数变化的曲线图，如图 3-1 所示。在并发用户数达到 100 之前，随着并发数的增长，服务器的资源被不断地充分利用，所以其吞吐率在不断提高。当并发用户数为 100 时，吞吐率最高。当并发用户数超过 100 后，吞吐率开始走下坡路，并且在并发用户数超过 150 后，吞吐率开始直线下跌，惨不忍睹。

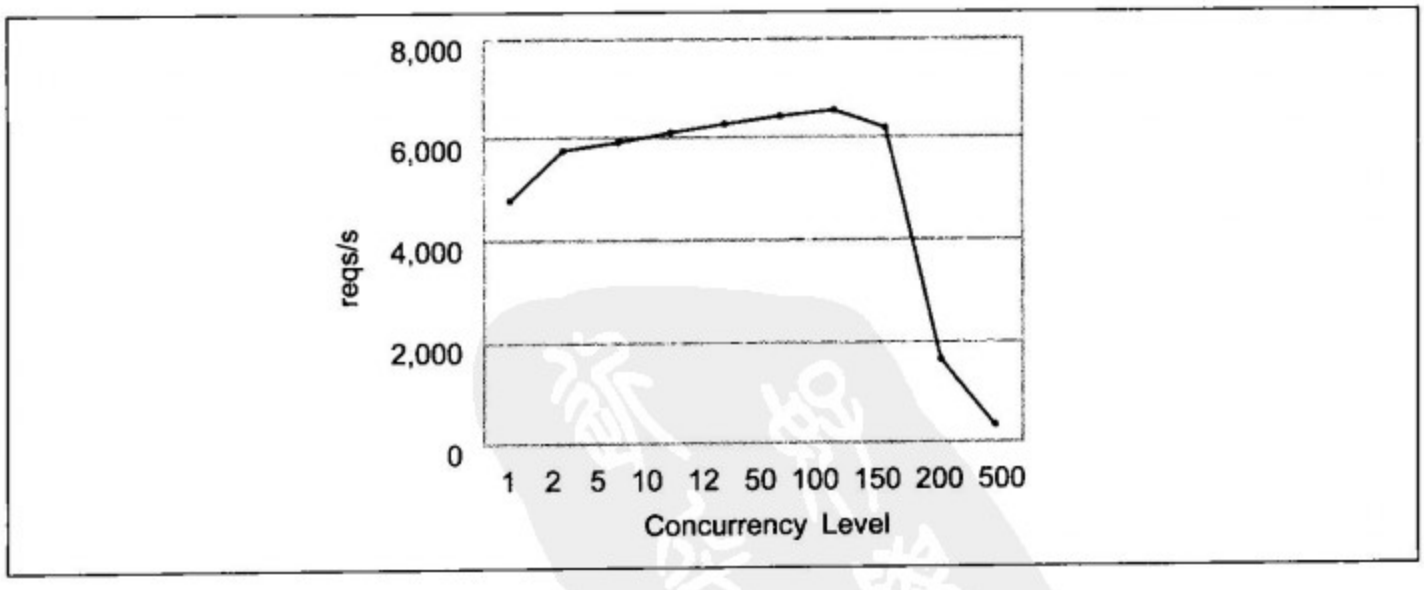


图 3-1　吞吐率随并发用户数变化的曲线图

再来看看服务器平均请求处理时间随并发用户数变化的曲线图，如图 3-2 所示。它的实质便是吞吐率的倒数，所以它的曲线形状和前一张图刚好沿横坐标等比例对称，同时在图上也反映出当并发用户数超过 150 后，服务器处理一个请求的平均时间突然飙升。

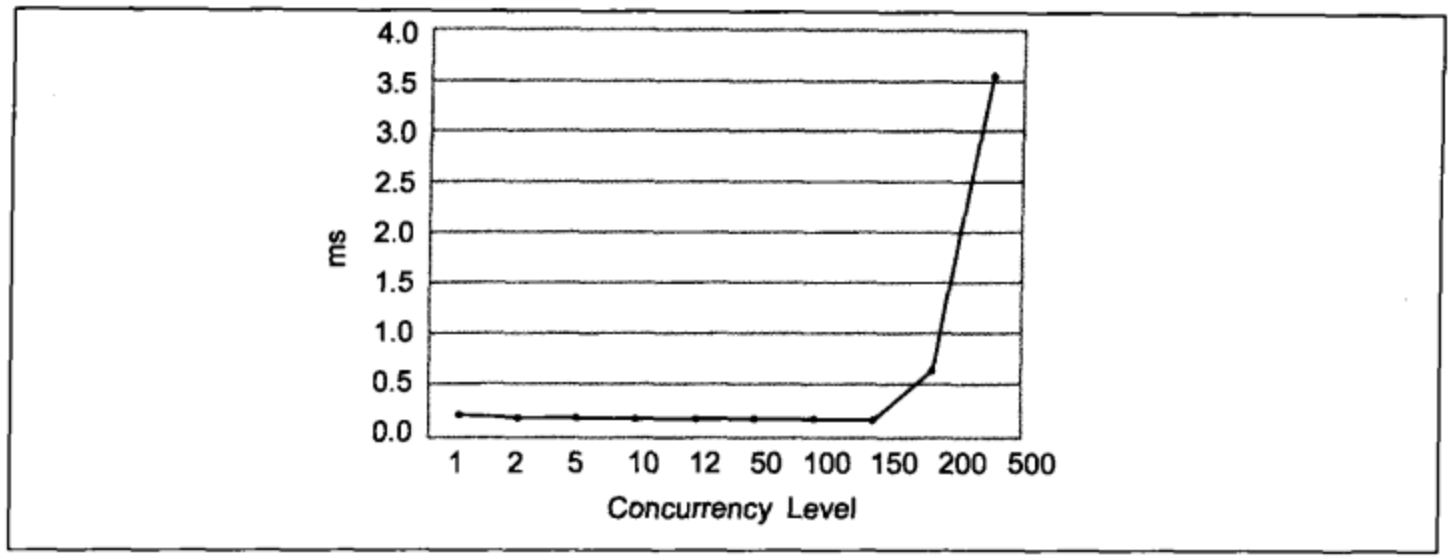


图 3-2　服务器平均请求处理时间随并发用户数变化的曲线图

下面来看用户平均请求等待时间随并发用户数变化的曲线图，如图 3-3 所示。同样，当并发用户数超过 150 后，请求的平均等待时间大幅度增加，当并发用户数达到 200 后，等待时间开始急剧增加，当并发用户数达到 500 的时候，请求的平均等待时间接近 2s，这对于用户来说绝对无法容忍。

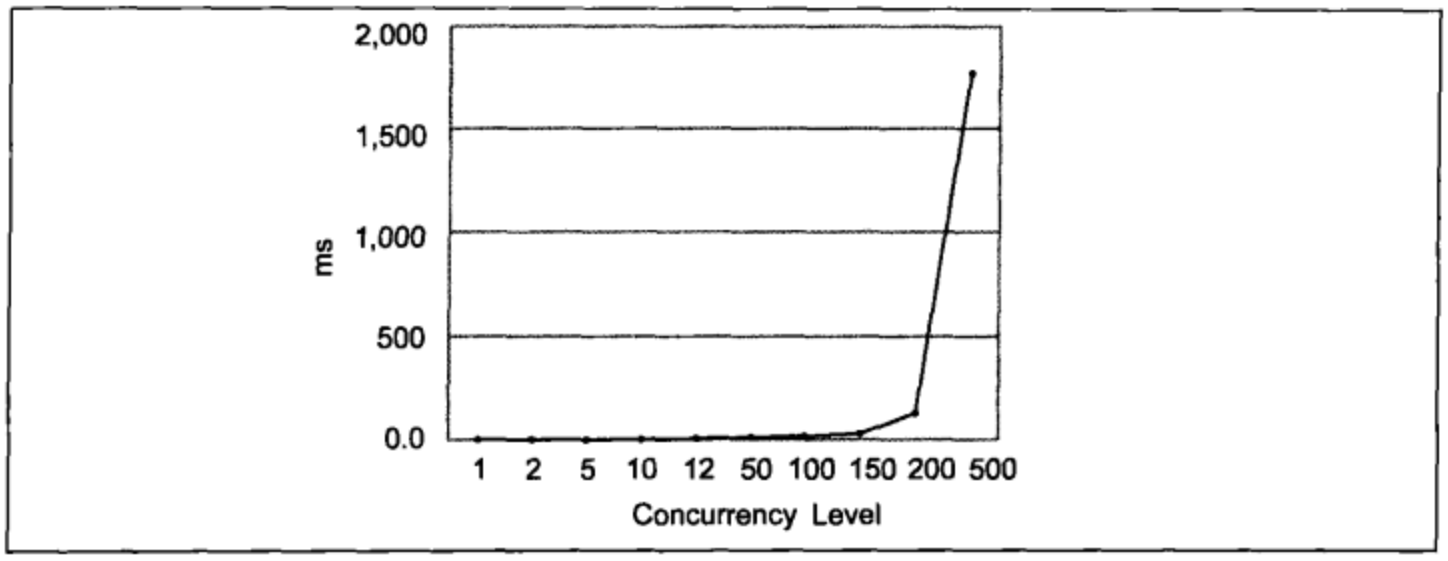


图 3-3　用户平均请求等待时间随并发用户数变化的曲线图

到现在为止，我们已经对 Web 服务器的几个常见性能指标做了详细的介绍，并且熟悉了压力测试的方法和测试结果的含义。那么，影响这些指标的因素，除去服务器的硬件配置，那就是前面提到的并发策略了。简单地说，并发策略的设计就是在服务器同时处理较多请求的时候，如何合理协调并充分利用 CPU 计算和 I/O 操作，使其在较大并发用户数的情况下提供较高的吞吐率。

并不存在一个对所有性质的请求都高效的并发策略，否则我们完全不需要介绍这些内容，任何架构师只要获得应用了最佳并发策略的 Web 服务器软件便可以应付各种性质的并发用户请求，显然那是不存在的。那么，你了解你的站点正在处理哪些性质的请求吗？只有

了解并根据这些性质来选择最佳的并发策略，高性能 Web 站点才会离你更近一步。

本章的其余部分将围绕并发策略，探讨一些你感兴趣的内容。

3.2 CPU 并发计算

服务器之所以可以同时处理多个请求，在于操作系统通过多执行流体系设计使得多个任务可以轮流使用系统资源，这些资源包括 CPU、内存以及 I/O 等。

进程

多执行流的一般实现便是进程。从本质上讲，多进程的好处并不仅仅在于 CPU 时间的轮流使用，还在于对 CPU 计算和 I/O 操作进行了很好的重叠利用，这里的 I/O 主要是指磁盘 I/O 和网络 I/O，它们的速度和 CPU 相比，就像老牛漫步和超音速飞机一样相距甚远。事实上，大多数进程的时间都主要消耗在了 I/O 操作上，现代计算机的 DMA 技术可以让 CPU 不参与 I/O 操作的全过程，比如进程通过系统调用，使得 CPU 向网卡或者磁盘等 I/O 设备发出指令，然后进程被挂起，释放出 CPU 资源，等待 I/O 设备完成工作后通过中断来通知进程重新就绪。所以对于单任务而言，CPU 大部分时间空闲，这时候多进程的作用便显得尤为重要。

进程的调度由内核来进行，从内核的观点看，进程的目的就是担当分配系统资源的实体。同时，进程也可以理解为记录程序实例当前运行到什么程度的一组数据，多个进程通过不同的进程描述符与这些数据进行关联。

每个进程都有自己独立的内存地址空间和生命周期。当子进程被父进程创建后，便将父进程地址空间的所有数据复制到自己的地址空间，完全继承父进程的所有上下文信息，它们之间可以通信，但是不互相依赖，也无权干涉彼此的地址空间。

进程的创建使用 fork()系统调用，它的开销虽然不很昂贵，但是在繁忙的服务器上频繁地创建进程，其开销可能成为影响性能的主要因素。Linux 2.6 对于 fork()的实现进行了优化，减少了一些多余的内存复制，但在早期的版本或者其他平台中，fork()的开销随着当前进程数量的递增而加大。

轻量级进程

由于进程之间相对独立，它们各自维护庞大的地址空间和上下文信息，无法很好地低成本共享数据，所以采用大量进程的 Web 服务器（比如 Apache 的 prefork 模型）在处理大量并发请求时，内存的大量消耗有时候会成为性能提升的制约因素。但是，进程的优越性有

时也恰恰体现在其相互独立所带来的稳定性和健壮性方面。

为此，在 Linux 2.0 之后，提供了对轻量级进程的支持，它由一个新的系统调用 clone()来创建，并由内核直接管理，像普通的进程一样独立存在，各自拥有进程描述符，但是这些进程已经允许共享一些资源，比如地址空间、打开的文件等。轻量级进程减少了内存的开销，并为多进程应用程序的数据共享提供了直接支持，但是其上下文切换的开销还是在所难免的。

线程

POSIX 1003.1c 为 Linux 定义了线程的接口“pthread”，有很多种具体实现，有些不是由内核来直接支持，在这种情况下，从内核角度看来，多线程只是一个普通的进程，它是由用户态通过一些库函数模拟实现的多执行流，所以多线程的管理完全在用户态完成，这种实现方式下线程切换的开销相比于进程和轻量级进程都要少些，但是它在多处理器的服务器（SMP）中表现较差，因为只有内核的进程调度器才有权利分配多个 CPU 的时间，这在随后的进程调度器中会有介绍。

POSIX 线程另一种实现是 LinuxThreads，它可以说是内核级的线程库（kernel-level threads），因为它通过 clone()来创建线程，也就是说它的实现原理是将线程和轻量级进程进行一对一关联，每个线程实际上就是一个轻量级进程，这样使得线程完全由内核的进程调度器来管理，所以它对于 SMP 的支持较好，但是线程切换的开销相比于用户态线程要多一些。

LinuxThreads 已经加入了 glibc 和 libc 的目前版本，并且必须在 Linux 2.0 之后使用，因为它依赖的 clone()系统调用和内核实时进程调度器出现在 Linux 2.0 之后。

在另一些操作系统（如 Digital Unix、Solaris、IRIX）中，内核线程也被支持，线程管理由内核来进行，它们同样可以很好地支持多 CPU。

进程调度器

在单 CPU 的机器上，虽然我们感觉到很多任务在同时运行，但是从微观意义上讲，任何时刻只有一个进程处于运行状态，而其他的进程有的处于挂起状态并等待就绪，有的已经就绪但等待 CPU 时间片，还有的处于其他状态。

内核中的进程调度器（Scheduler）维护着各种状态的进程队列。在 Linux 中，进程调度器维护着一个包括所有可运行进程的队列，称为“运行队列（Run Quere）”，以及一个包括所有休眠进程和僵尸进程的列表。

进程调度器的一项重要工作就是决定下一个运行的进程，如果运行队列中有不止一个进

程，那就比较伤脑筋了，按照先来后到的顺序也许不是那么合理，因为运行在系统中的进程有着不同的工作需要，比如有些进程需要处理紧急的事件，有些进程只是在后台发送不太紧急的邮件，所以每个进程需要告诉进程调度器它们的紧急程度，这就是进程优先级。

进程优先级除了可以由进程自己决定，进程调度器在进程运行时也可以动态调整它们的优先级，比如对有些进程适当提高优先级，对有些进程则进行处罚，降低它们的优先级，这些行为的出发点都是为了让所有进程更好地重叠利用系统资源。Linux 对于进程的动态调整，体现在进程的 nice 属性中，我们对 Lighttpd 进行持续压力测试，使用 100 个并发用户来请求 151 字节的静态文件，这时候通过 top 来观察 lighttpd 进程的优先级和动态调整，如下所示：

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
30400 daemon    25   0  6868 4420  752 R  100  0.1   0:33.53 lighttpd
```

进程的优先级属性为 Priority，在 top 结果中用 PR 表示，而优先级动态调整值在 top 中用 NI 表示。在 top 的间断刷新中，lighttpd 的 PR 值从 16 不断上升为 25，最后一直停留在 25，直到测试结束。

我们对 Lighttpd 的 fastcgi 进程进行压力测试，看看 fastcgi 进程的优先级，我们配置 Lighttpd 为打开 4 个 fastcgi 进程，请求的资源为一个以 CPU 计算为主的 PHP 程序，结果如下所示：

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
19440 daemon    20   0 12476 2764 1484 R    3  0.1   0:02.64 php
19442 daemon    21   0 12476 2764 1484 R    3  0.1   0:02.64 php
19444 daemon    20   0 12480 2764 1484 R    3  0.1   0:02.62 php
19446 daemon    21   0 12480 2924 1652 R    3  0.1   0:31.20 php
```

可以看到每个代表 fastcgi 的 php 进程的 PR 值已经低于刚才的 lighttpd。

再来看看 Apache-prefork，同时是使用 100 个并发用户来请求 151 字节的静态文件，Apache 子进程数上限设置为 100，也许你的经验已经告诉你 Apache 处理这些请求时的 CPU 开销要远远超过刚才的 Lighttpd，在随后我们将会专门来探讨这个问题。这时候的 top 结果如下所示：

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2144 daemon    15   0 61072  14m 3012 S    2  0.4   0:00.25 httpd
 1829 daemon    15   0 61072  14m 3012 S    2  0.4   0:01.41 httpd
 1939 daemon    15   0 61328  16m 4212 S    2  0.4   0:00.96 httpd
 2121 daemon    15   0 61072  14m 3012 S    2  0.4   0:00.28 httpd
 2128 daemon    16   0 61072  14m 3016 S    2  0.4   0:00.26 httpd
 2152 daemon    15   0 61072  14m 3012 S    2  0.4   0:00.21 httpd
 2153 daemon    15   0 61072  14m 3012 S    2  0.4   0:00.24 httpd
 2157 daemon    16   0 61072  14m 3012 S    2  0.4   0:00.22 httpd
 2167 daemon    15   0 61072  14m 3012 S    2  0.4   0:00.22 httpd
```

的确，httpd 的 PR 值为 15 左右，操作系统不愿意给它更高的优先级。事实上 Linux 2.6 的

进程调度器更加偏爱 I/O 操作密集型的进程，因为这些进程在发起 I/O 操作后通常都会阻塞（除非使用异步 I/O），不会占用太多 CPU 时间，这就意味着其他进程都可以更好地交错运行。

PR 所代表的值有什么含义呢？它其实就是进程调度器分配给进程的时间片长度，单位是时钟个数，那么一个时钟需要多长时间呢？这跟 CPU 的主频以及操作系统平台有关，比如 Linux 上一般为 10ms，那么 PR 值为 15 则表示这个进程的时间片为 150ms。

在操作系统中，时间片的长度是权衡利弊后的一个博弈结果，如果各进程的时间片太短，那么 CPU 浪费在进程切换上的时间比例就比较大，整体效率降低；而如果时间片太长，则多任务实时性以及交互性就无法保证。

另外，在传统的 UNIX 中，进程调度器为所有进程计算 nice 值的时候，需要锁住进程表，这时候对于使用了多处理器的服务器（SMP）来说非常糟糕，因为当一个 CPU 锁住进程表进行 nice 值计算时，其他 CPU 无法切换进程。这个问题在一些商业化操作系统以及 Linux 中得以解决，简单地说就是给每个处理器分配一个运行队列，互不影响，进程调度器负责将进程分配到适合的 CPU。

系统负载

在进程调度器维护的运行队列中，任何时刻至少存在一个进程，那就是正在运行的进程。而当运行队列中有不止一个进程的时候，就说明此时 CPU 比较抢手，其他进程还在等着呢，进程调度器应该尽快让正在运行的进程释放 CPU。

通过在任何时刻查看/proc/loadavg，可以了解到运行队列的情况：

```
s-colin:/ # cat /proc/loadavg
1.63 0.48 0.21 10/200 17145
```

注意 10/200 这部分，其中的 10 代表此时运行队列中的进程个数，而 200 则代表此时的进程总数。

最右边的 17145 代表到此时为止，最后创建的一个进程 ID。

接下来，请看左边的三个数值，分别是 1.63、0.48、0.21，它们就是我们常说的系统负载。我们都知道，系统负载越高，代表 CPU 越繁忙，越无法很好地满足所有进程的需要。但是，系统负载是如何计算而来的呢？根据定义，它是在单位时间内运行队列中就绪等待的进程数平均值，所以当运行队列中就绪进程不需要等待就可以马上获得 CPU 的时候，系统负载便非常低。当系统负载为 0.00 时，说明任何进程只要就绪后就可以马上获得 CPU，不需要等待，这时候系统响应速度最快。

那么，刚才提到的三个数值，便是系统最近 1 分钟、5 分钟和 15 分钟分别计算得出的系统负载。

我们还可以通过其他方法获得系统负载，比如 top、w 等工具：

```
s-colin:~ # top
top - 21:56:06 up 395 days, 10:38,  6 users,  load average: 0.00, 0.00, 0.00

s-colin:~ # w
22:24:05 up 395 days, 11:06,  6 users,  load average: 0.00, 0.00, 0.00
```

从实现方法上看，这些工具获得的系统负载都是来源于/proc/loadavg。

了解了这些内容后，要想提高服务器的系统负载，很简单，我们编写一个没有任何 I/O 操作并且长时间占用 CPU 时间的 PHP 脚本，比如一个循环累加器，如下所示：

```
<?php
$max = 100000000;
$sum = 0;
for ($i = 0; $i < $num_max; ++$i)
{
        $sum += $i;
}
echo $sum;
?>
```

然后用 100 个并发用户请求这个脚本，进行压力测试，这时候查看系统负载，结果如下所示：

```
load average: 98.26, 45.89, 17.94
```

我不知道你是否很有成就感，不过请注意可不要在对外提供服务的服务器上进行类似操作。

进程切换

通过前面的介绍，我们知道，为了让所有的进程可以轮流使用系统资源，进程调度器在必要的时候挂起正在运行的进程，同时恢复以前挂起的某个进程，这种行为称为进程切换，也就是我们常说的“上下文切换”，这个名称在某种意义上非常形象，“上下文”正是表示进程运行到何种程度。

我们知道，进程拥有自己独立的内存空间，但是每个进程都只能共享 CPU 寄存器。一个进程被挂起的本质就是将它在 CPU 寄存器中的数据拿出来暂存在内核态堆栈中，而一个进程恢复工作的本质就是将它的数据重新装入 CPU 寄存器，这段装入和移出的数据我们称为“硬件上下文”，它也是进程上下文的一部分，除此之外，进程上下文中还包含了进程运行时需要的一切状态信息。

当硬件上下文频繁地装入和移出时，所消耗的时间是非常可观的，我们使用 Nmon 工具监视服务器每秒上下文切换次数。Nmon 是一个非常不错的 Linux 性能监视工具，它可以提供基于服务器终端命令行的监视界面，还可以通过专用的分析器将监视数据生成报表和曲线图，在后面关于性能监控的章节中我们将详细介绍 Nmon 的用法。

当服务器不提供任何 HTTP 服务的时候，我们用 Nmon 得到某时刻的抽样结果，如下所示：

```
RunQueue                1          Load     Average
ContextSwitch          28.4        1 mins     0.01
Forks                   0.0        5 mins     0.03
Interrupts            253.0       15 mins     0.00
```

这时的上下文切换平均每秒 28.4 次，这是操作系统正常运转所进行的必要工作。

接下来，我们运行 ab 对 Apache 进行压力测试，这里 Apache 使用 prefork 运行模式，在该模式下，Apache 通过一个父进程预先创建一定数量的子进程，所有子进程竞争 accept 用户请求，一旦某个子进程 accept 成功后便开始处理这个请求。每个子进程能够处理的请求数可以通过 MaxRequestsPerChild 进行配置，如果 MaxRequestsPerChild 设置为 1，那就相当于传统的 fork 模式，为每个请求创建新的进程来处理。同时父进程根据负载和 MPM 参数配置对子进程数进行必要的增减。以下列出此时 prefork 的 MPM 参数配置：

```
<IfModule mpm_prefork_module>
    StartServers          5
    MinSpareServers       10
    MaxSpareServers       20
    MaxClients            150
    MaxRequestsPerChild   0
</IfModule>
```

在 Apache 1.3 版本之后，由于创建子进程的策略发生了改变，所以以上的参数配置几乎不需要太多修改，它对 Apache 的性能提升没有本质性的影响。

在测试的过程中，我们用 Nmon 得到某时刻的抽样结果，如下所示：

```
RunQueue                3          Load       Average
ContextSwitch       18166.5        1 mins       1.34
Forks                   0.5        5 mins       0.43
Interrupts            300.4       15 mins       0.15
```

太不可思议了，此时的上下文切换次数达到平均每秒 18166.5 次。我们将每 20s 获得的上下文切换次数通过 Nmon 分析器生成曲线图，用来反映从压力测试开始到结束后一段时间内上下文切换次数的变化，请看 Apache 压力测试时上下文切换次数随时间变化的曲线图，如图 3-4 所示。

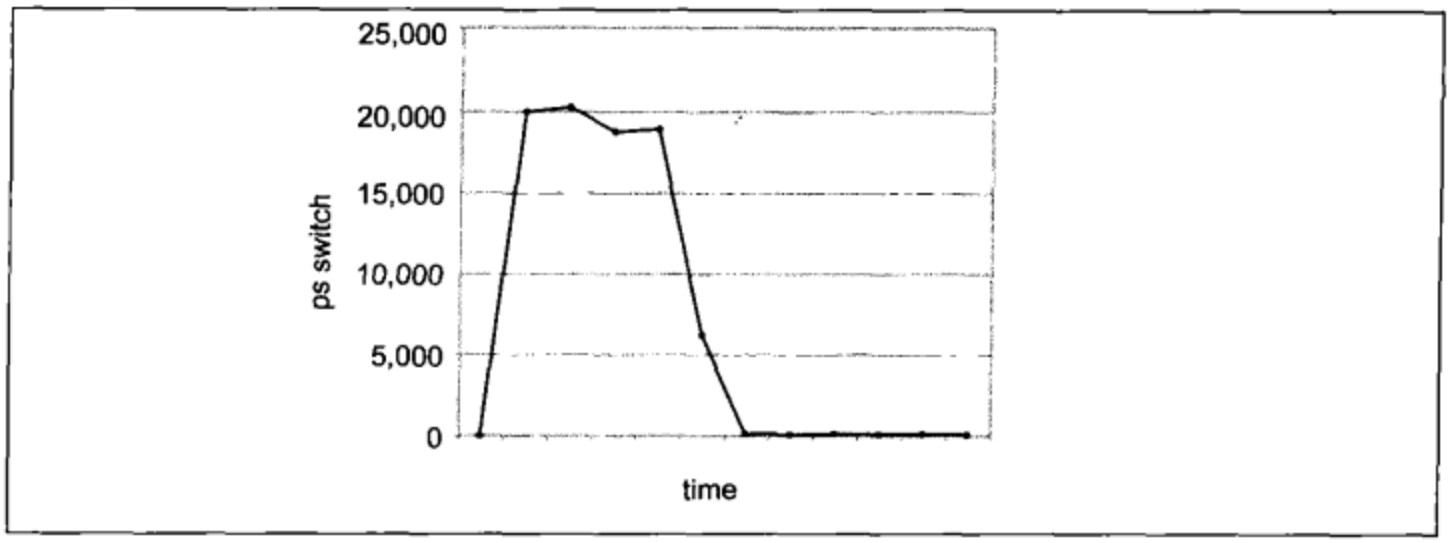


图 3-4　Apache-prefork 压力测试时上下文切换次数随时间变化的曲线图

我们知道，Apache 为每个并发用户创建独立的子进程来处理请求，所以在测试过程中，我们查看某时刻的 httpd 进程数：

```
s-colin:~ # ps afx | grep httpd | wc -l
96
```

除去 grep 进程本身和 Apache 父进程，此时有 94 个 Apache 子进程，如果改用 Apache 的 mod_status 来查看状态，应该会得到如下所示的信息：

```
Current Time: Tuesday, 10-Feb-2009 20:26:47 CST
Restart Time: Monday, 09-Feb-2009 15:03:09 CST
Parent Server Generation: 26
Server uptime: 1 day 5 hours 23 minutes 37 seconds
Total accesses: 2599062 - Total Traffic: 3.3 GB
CPU Usage: u26.81 s29.94 cu28.82 cs0 - .0809% CPU load
24.6 requests/sec - 32.4 kB/second - 1349 B/request
95 requests currently being processed, 0 idle workers
```

注意，这里看到的 95 个请求里面还包括 mod_status 自身的 Web 请求，所以同时处理的请求是 94 个，并且没有空闲子进程，所以有 94 个 Apache 子进程。

我们用 top 来看看这些 Apache 子进程的内存消耗情况：

```
  PID  USER   PR  NI   VIRT  RES  SHR S %CPU %MEM    TIME+  SWAP COMMAND
27296 daemon 15   0  60284  14m 3012 R    2  0.3   0:00.11  44m  httpd
27304 daemon 15   0  60284  14m 3012 S    2  0.3   0:00.06  44m  httpd
27312 daemon 15   0  60284  14m 3012 S    2  0.3   0:00.06  44m  httpd
27328 daemon 15   0  60284  14m 3012 S    2  0.3   0:00.06  44m  httpd
27343 daemon 15   0  60284  14m 3012 S    2  0.3   0:00.06  44m  httpd
26638 daemon 15   0  60816  17m 4912 S    2  0.4   0:01.62  42m  httpd
26627 daemon 15   0  61596  26m  13m S    2  0.6   0:02.32  33m  httpd
26630 daemon 15   0  60820  16m 4724 S    2  0.4   0:01.88  42m  httpd
......
```

这里只列出了其中的前 8 个 Apache 子进程，每个进程的 RES 值表示其占用的物理内存空

间大小，SWAP 值表示其使用的虚拟内存空间大小，VIRT 值则等于 RES 和 SWAP 的总和，以上数值在没有注明单位的情况下都是指 KB。可以看出，每个 Apache 子进程的内存开销非常大，它也是影响性能的一方面。当然，我们可以通过编译参数和 httpd.conf 的配置来减少 Apache 加载的模块和多余的处理，以此减少内存开销。

最后，ab 的测试结果如下所示：

```
Concurrency Level:      100
Time taken for tests:   91.252 seconds
Complete requests:      500000
Failed requests:        0
Write errors:           0
Total transferred:      214501287 bytes
HTML transferred:       75500453 bytes
Requests per second:    5479.30 [#/sec] (mean)
Time per request:       18.250 [ms] (mean)
Time per request:       0.183 [ms] (mean, across all concurrent requests)
Transfer rate:          2295.54 [Kbytes/sec] received
```

可见，在 Apache 平均每秒处理 5524.8 个请求的同时，操作系统的平均上下文切换次数为 18166.5 次每秒。

为了与 Apache 的多进程模型相比较，我们来对 Lighttpd 进行压力测试，对于静态网页的请求，它使用单进程单线程模型来处理多个请求，而对于 PHP 脚本，它也仅使用数量较少的 fastcgi 进程进行处理。我们在压力测试的某时刻抽样获取结果如下所示：

```
RunQueue            1           Load        Average
ContextSwitch       61.3        1 mins      0.66
Forks               0.0         5 mins      0.15
Interrupts          272.4       15 mins     0.05
```

同时我们也生成了曲线图，请看 Lighttpd 压力测试时上下文切换次数随时间变化的曲线图，如图 3-5 所示。

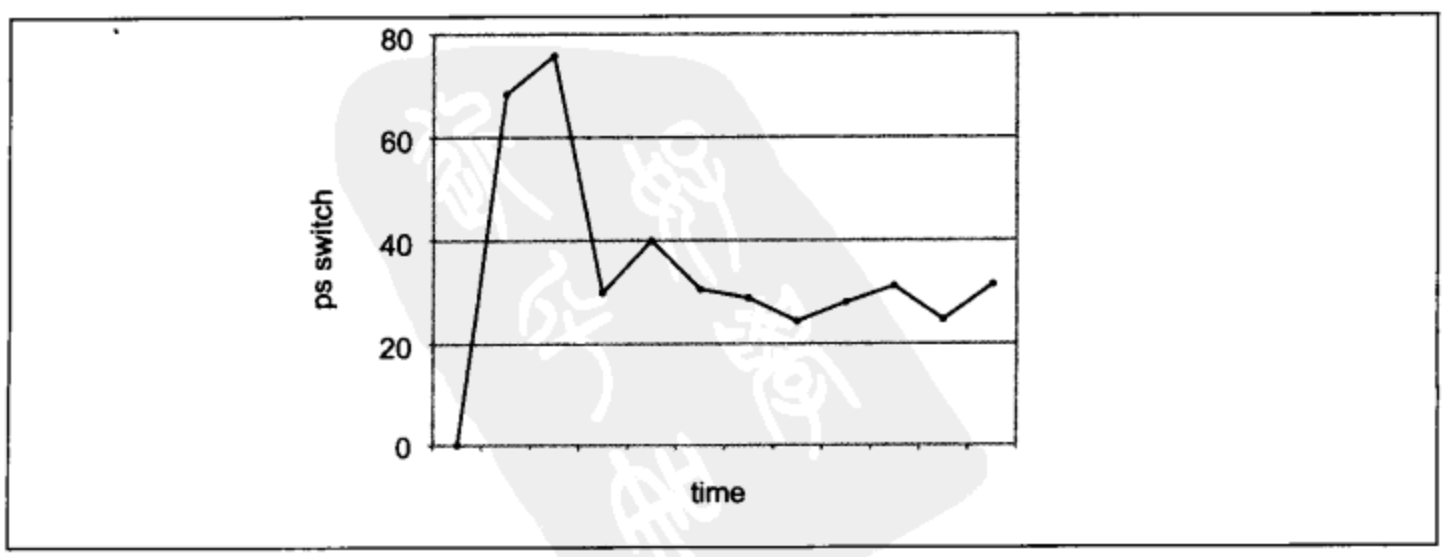


图 3-5　Lighttpd 压力测试时上下文切换次数随时间变化的曲线图

相比之下，前面 Apache-prefork 在执行相同数量的处理任务时，上下文切换的次数多得惊人，虽然我们不知道这些切换究竟花了多少时间，但这些时间一定在总处理时间中占有不可忽视的地位，它也是 Apache-prefork 在此次压力测试中落后于 Lighttpd 的原因之一。下面看一下 Lighttpd 的测试结果：

```
Concurrency Level:      100
Time taken for tests:   37.528 seconds
Complete requests:      500000
Failed requests:        0
Write errors:           0
Total transferred:      191500000 bytes
HTML transferred:       75500000 bytes
Requests per second:    13323.30 [#/sec] (mean)
Time per request:       7.506 [ms] (mean)
Time per request:       0.075 [ms] (mean, across all concurrent requests)
Transfer rate:          4983.22 [Kbytes/sec] received
```

另外，Lighttpd 的内存开销也非常小：

```
  PID USER   PR NI  VIRT  RES  SHR S %CPU %MEM   TIME+  SWAP COMMAND
26490 daemon 25  0  9208 6704  784 R  100  0.2  1:01.28 2504 lighttpd
```

正如我们常说的，人多不见得就厉害，不够敏捷可能成为其致命的弱点。所以，如果我们希望服务器支持较大的并发数，那么就要尽量减少上下文切换次数，最简单的做法就是减少进程数，尽量使用线程并配合其他 I/O 模型来设计并发策略，这在随后的内容中会详细探讨。

这里顺便提出一个有趣的话题，说说我们的大脑，当我们在思考很多事情的时候，我认为是并发的，而不是并行的，也就是任何微观时刻我们不可能同时思考多件事情，而是大脑不断在多件事情之间进行“上下文切换”，至少我是这样的。

举个例子，假如你用一只手画一个圆形，也许可以很快完成，但是假如让你两只手同时画图，左手画圆形，右手画矩形，你一定会觉得很费劲，脑子转不过来，画得很不流畅，这是因为画图是需要通过大脑计算的，什么时候画直线，什么时候拐弯，什么时候画弧线等，当两只手同时画图时，为了保证两边都能流畅地进行，就得快速进行“上下文切换”，将圆形和矩形的规则不断地在大脑里装入和移出，除非你专门锻炼过针对这种同时画图的切换速度，否则我们切换的速度非常慢。

还有的时候，当我正在聚精会神地思考着一件事情时，突然有人打断了我，我们聊完后，我努力回忆之前在想什么，这也许就是那件事情被突然移出后没有及时缓存在大脑的其他位置，以至于无法快速切换回原来的状态。

你也许会发现，当我们走路的时候，大脑几乎可以不想着走路，而去思考其他事情，我就喜欢走路的时候思考问题。这有时候取决于道路情况，在平坦空旷的路上行走几乎不需要太多的大脑计算，但不是完全不需要，完全不考虑路况的结果就是撞电线杆，一般用它来歌颂过于专心思考问题的科学家。

关于神奇的大脑与 CPU 的并发计算还有什么相似之处呢？相信你一定会联想到更多。

IOWait

在 CPU 使用率报告中，除了用户空间和内核空间的 CPU 使用率以外，通常我们还关注 IOWait，它是指 CPU 空闲并且等待 I/O 操作完成的时间比例，注意，它是一个比例，而不是绝对时间，比如我们在 top 中看到当前的 IOWait 为 12.9%。

```
Cpu(s): 0.2%us, 0.9%sy, 0.0%ni, 80.1%id, 12.9%wa, 0.9%hi, 5.0%si, 0.0%st
```

但是，IOWait 往往并不能真实地代表 I/O 操作的性能或者工作量，它的设计出发点是用来衡量 CPU 的性能，举个例子，假如有一个任务需要花费 10ms 的 I/O 操作时间和 10ms 的 CPU 时间，那么总时间为 20ms，IOWait 便为 10ms/20ms，等于 50%，这时并不意味着 I/O 操作的繁忙程度为 50%，事实上，即便是 IOWait 为 100%，也不一定代表 I/O 出现性能问题或者瓶颈，同样，IOWait 为 0 的时候 I/O 操作也可能很繁忙，所以，如果你希望真正了解当前 I/O 的性能，可以进行磁盘 I/O 测试或者查看网络 I/O 流量等。

当然，在实际的多进程环境中，IOWait 的计算方法并没有这么简单，而且各个 Linux 发行版也有所不同，但既然 IOWait 是用来描述 CPU 性能的，那么当 IOWait 很高的时候，至少说明当前任务的 CPU 时间开销相对于 I/O 操作时间来说比较少，通常对于依赖磁盘 I/O 的应用来说，这比较正常，因为 CPU 的速度比磁盘 I/O 越来越快，特别是在随机的磁盘 I/O 操作中，大量的寻址时间是无法避免的。但是，有时候 IOWait 的确会误导我们，比如对于提供下载服务的站点，我们采用了 Nginx 来作为 Web 服务器，当我们开启了 128 个 nginx 进程的时候，通过 Nmon 得到的 CPU 监控数据如图 3-6 所示。

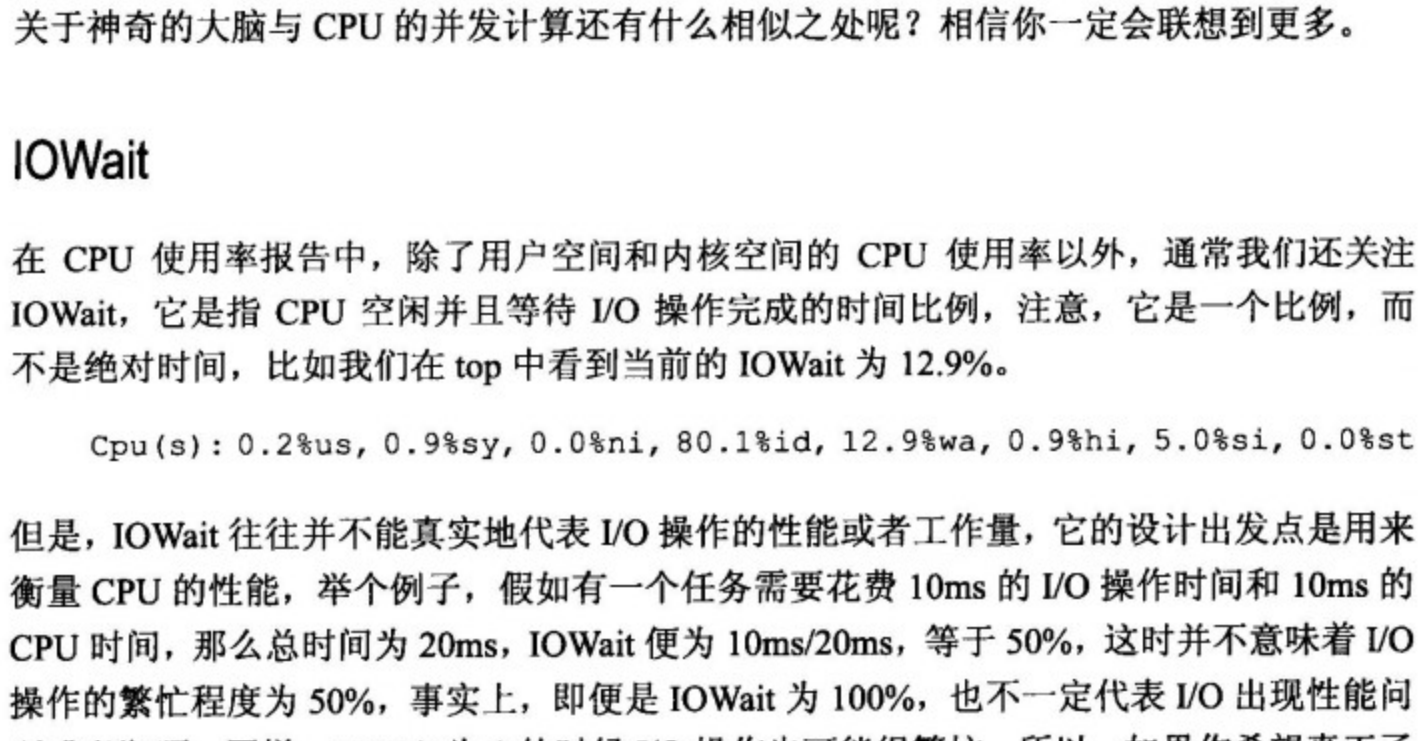


图 3-6　Nginx 为 128 个进程时的 CPU 使用率监控

可以看到，此刻的 IOWait 相对比较高，平均达到了 50%以上，也许你认为这是正常的，因为下载服务采用 sendfile 系统调用来传送数据，几乎不需要用户空间的 CPU 时间，所以几乎看不出什么问题。但是，当我们将 Nginx 的进程数降低到 8 后，CPU 监控数据如图 3-7 所示。

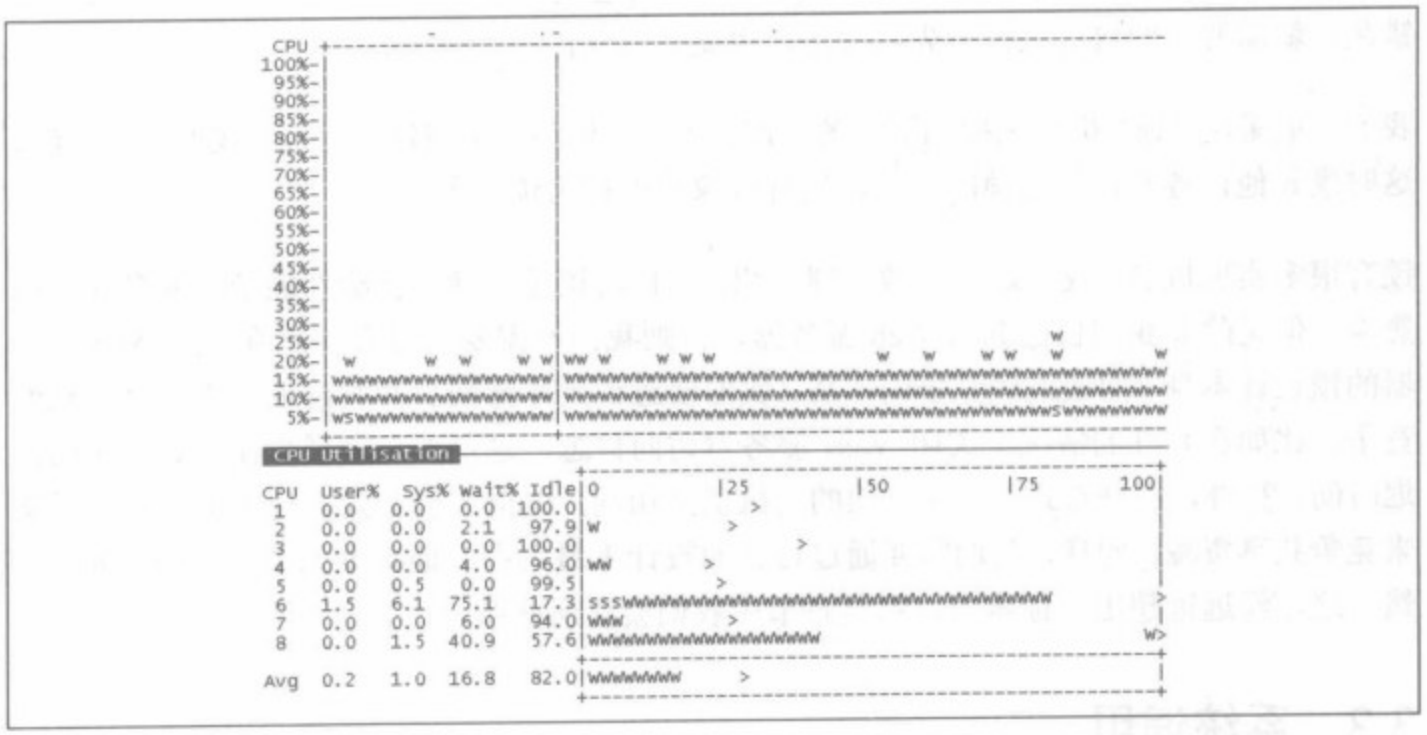


图 3-7　Nginx 为 8 个进程时的 CPU 使用率监控

IOWait 大幅度下降，同时最关键的是，下载服务的网络 I/O 提高了近一倍，如图 3-8 所示。

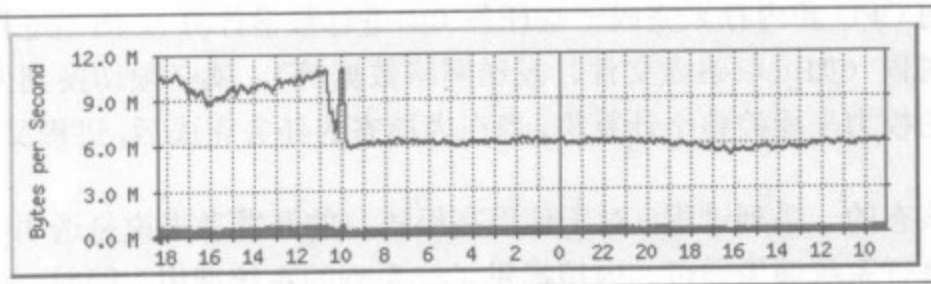


图 3-8　Nginx 调整进程数后的网络 I/O 流量变化

需要注意的是，上面 MRTG 流量图的时间轴是从右到左的。对于流量的提升，我们究其原因，或许是因为 nginx 进程大量减少后，花费在进程上下文切换的 CPU 时间也随之减少，更多的 CPU 时间可以用在发起 sendfile 系统调用，所以网络 I/O 流量大幅度提高，而与此同时，磁盘由于存在高速缓存，所以实际的磁盘 I/O 并没有明显提高，但 CPU 使用率提高了，所以 IOWait 减少了。

总之，IOWait 有时候是一个耐人寻味的指标，你需要根据站点的实际情况来做出合理的判断。

锁竞争

只要涉及抢手资源的占有，就必然会存在竞争，这在现实社会中屡见不鲜。然而在服务器处理大量并发请求的时候，多个请求处理任务之间也存在一些资源抢占竞争，这就需要有一种机制来维持秩序，比如多个线程同时写一个日志文件，为了防止写入的数据发生位置错乱，就需要我们自己来控制先后顺序，也就是要保证线程安全。

我们一般采用“锁”机制来控制资源的占用，当一个任务占用资源的时候，我们锁住资源，这时候其他任务都在等待锁的释放，这种现象我们称为锁竞争。

锁有很多实现机制，我们这里不做详细介绍，不良的锁设计确实会给性能带来或多或少的影响，但是除非我们自己编写 Web 服务器，否则我们不需要太担心目前流行的 Web 服务器的锁设计本身。而通过锁竞争的本质，我们要意识到尽量减少并发请求对于共享资源的竞争，比如在允许的情况下关闭 Web 服务器访问日志，这可以大大减少在锁等待时的延迟时间。另外，在分布式系统中，锁的动机仍然相同，你也许会看到多台服务器通过网络来竞争共享资源，同样，我们需要通过良好的设计来最大程度地减少无辜的等待时间，当然，这已经远远超出了锁本身，后续章节中我们会有所介绍。

3.3 系统调用

前面的内容中提到进程的用户态和内核态两种运行模式，这是 Linux 为进程设计的两种运行级别，进程可以在两种模式之间切换，这也需要一定的开销。进程通常运行在用户态，这时候可以使用 CPU 和内存来完成一些任务（如进行数学计算），而当进程需要对硬件外设进行操作的时候（如读取磁盘文件、发送网络数据等），就必须切换到内核态，这时候它将拥有更多的权力来操控整个计算机，当在内核态的任务完成后，进程又切换回用户态。

进程切换到内核态的一系列过程，对于使用高级语言的开发者来说是透明的，程序代码只在需要的时候进行系统调用即可。内核提供了一系列的系统调用，同时，C 库函数（libc）将系统调用封装在编程接口中，提供给用户态的进程。用户态的进程可以直接进行系统调用，也可以使用封装了系统调用的 C API，比如 write()系统调用，它的封装 API 之一就是用于发送网络数据的 send()。

这种用户态和内核态的分离，动机主要在于提高系统底层安全性以及简化开发模型。由于所有进程都必须通过内核提供的系统调用来操作硬件，所以不用担心应用程序对硬件进行非法操作，由于将底层的实现都屏蔽在了系统调用中，也大大简化了用户态应用开发的难度。

由于系统调用涉及进程从用户态到内核态的切换，导致一定的内存空间交换，这也是一定程度上的上下文切换，所以系统调用的开销通常认为是比较昂贵的。

减少不必要的系统调用，也是 Web 服务器性能优化的一个方面，举个例子，在 Apache 中

支持通过.htacess 文件来为 htdocs 下各个目录进行局部的参数配置，但它也有一定的副作用。我们将 httpd.conf 中的 AllowOverride 设置为 All，如下所示：

```
<Directory />
    Options FollowSymLinks
    AllowOverride all
    Order deny,allow
    Allow from all
</Directory>
```

这时候，我们使用 strace 来跟踪 Apache 的一个子进程，获得某次请求处理中的一系列系统调用，如下所示：

```
    accept(3, {sa_family=AF_INET6, sin6_port=htons(45035), inet_pton (AF_
INET6, "::ffff:127.0.0.1", &sin6_addr), sin6_flowinfo=0, sin6_scope_id=0},
[28]) = 19
    gettimeofday({1234421353, 675597}, NULL) = 0
    getsockname(19, {sa_family=AF_INET6, sin6_port=htons(80), inet_pton
(AF_INET6, "::ffff:127.0.0.1", &sin6_addr), sin6_flowinfo=0, sin6_scope_
id=0}, [28]) = 0
    fcntl64(19, F_GETFL)                    = 0x2 (flags O_RDWR)
    fcntl64(19, F_SETFL, O_RDWR|O_NONBLOCK) = 0
    gettimeofday({1234421353, 675771}, NULL) = 0
    read(19, "GET /test.htm HTTP/1.0\r\nHost: lo"..., 8000) = 85
    gettimeofday({1234421353, 675845}, NULL) = 0
    gettimeofday({1234421353, 987679}, NULL) = 0
    gettimeofday({1234421353, 987712}, NULL) = 0
    stat64("/data/www/site/htdocs/test.htm", {st_mode=S_IFREG|0644, st_
size=151, ...}) = 0
    open("/.htaccess", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file or
directory)
    open("/data/.htaccess", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file
or directory)
    open("/data/www/.htaccess", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such
file or directory)
    open("/data/www/site/.htaccess", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No
such file or directory)
    open("/data/www/site/htdocs/.htaccess",O_RDONLY|O_LARGEFILE) = -1 ENOENT
(No such file or directory)
    open("/data/www/site/htdocs/test.htm/.htaccess",
O_RDONLY|O_LARGEFILE) = -1 ENOTDIR (Not a directory)
    open("/data/www/site/htdocs/test.htm", O_RDONLY|O_LARGEFILE) = 20
    mmap2(NULL, 151, PROT_READ, MAP_SHARED, 20, 0) = 0xb7f8f000
    writev(19, [{"HTTP/1.1 200 OK\r\nDate: Thu, 12 F"..., 278}, {"<html>\n
<head>\n<title>Welcome to "..., 151}], 2) = 429
    munmap(0xb7f8f000, 151)                 = 0
    gettimeofday({1234422038, 807811}, NULL) = 0
    write(17, "127.0.0.1 - - [12/Feb/2009:15:00"..., 76) = 76
    gettimeofday({1234422038, 807893}, NULL) = 0
    times({tms_utime=6, tms_stime=8, tms_cutime=0, tms_cstime=0}) = 854517315
    gettimeofday({1234422038, 807955}, NULL) = 0
    shutdown(19, 1 /* send */)              = 0
    poll([{fd=19, events=POLLIN, revents=POLLIN|POLLHUP}], 1, 2000) = 1
    read(19, "", 512)                       = 0
    close(19)                               = 0
```

可知一共有 30 次系统调用，其中的粗体部分表示 Apache 在检查被访问的文件路径中各级目录下是否存在.htacess 文件，共涉及 6 次 open()系统调用。

此时，我们使用 ab 对其进行压力测试，测试结果如下所示：

```
Concurrency Level:      100
Time taken for tests:   1.500 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      4291716 bytes
HTML transferred:       1510604 bytes
Requests per second:    6666.32 [#/sec] (mean)
Time per request:       15.001 [ms] (mean)
Time per request:       0.150 [ms] (mean, across all concurrent requests)
Transfer rate:          2793.94 [Kbytes/sec] received
```

下面我们关闭.htacess 功能来减掉这 6 次系统调用，修改 httpd.conf 如下所示：

```
<Directory />
    Options FollowSymLinks
    AllowOverride none
    Order deny,allow
    Allow from all
</Directory>
```

另外，我们还看到 gettimeofday()和 times()这两次系统调用，它用来获取当前的系统时间。我们还可以通过关闭 mod_status 来减少多余的 gettimeofday()系统调用，因为 mod_status 为了监控每一个请求的处理时间，必须在处理请求的各个环节多次获取系统时间，修改 httpd.conf 如下所示：

```
    #ExtendedStatus On
```

重启 Apache 后，我们再次使用 strace 跟踪，结果如下所示：

```
    accept(3, {sa_family=AF_INET6, sin6_port=htons(46358), inet_pton(AF_
INET6, "::ffff:127.0.0.1", &sin6_addr), sin6_flowinfo=0, sin6_scope_id=0},
[28]) = 19
    getsockname(19, {sa_family=AF_INET6, sin6_port=htons(80), inet_pton
(AF_INET6, "::ffff:127.0.0.1", &sin6_addr), sin6_flowinfo=0, sin6_scope_
id=0}, [28]) = 0
    fcntl64(19, F_GETFL)                    = 0x2 (flags O_RDWR)
    fcntl64(19, F_SETFL, O_RDWR|O_NONBLOCK) = 0
    read(19, "GET /test.htm HTTP/1.0\r\nHost: lo"..., 8000) = 85
    gettimeofday({1234422821, 442609}, NULL) = 0
    stat64("/data/www/site/htdocs/test.htm", {st_mode=S_IFREG|0644, st_
size=151, ...}) = 0
    open("/data/www/site/htdocs/test.htm", O_RDONLY|O_LARGEFILE) = 20
    mmap2(NULL, 151, PROT_READ, MAP_SHARED, 20, 0) = 0xb7f8f000
    writev(19, [{"HTTP/1.1 200 OK\r\nDate: Thu, 12 F"..., 278}, {"<html>\
n<head>\n<title>Welcome to "..., 151}], 2) = 429
    munmap(0xb7f8f000, 151)                 = 0
```

```
write(17, "127.0.0.1 - - [12/Feb/2009:15:13"..., 76) = 76
shutdown(19, 1 /* send */)                    = 0
poll([{fd=19, events=POLLIN, revents=POLLIN|POLLHUP}], 1, 2000) = 1
read(19, "", 512)                             = 0
close(19)                                     = 0
```

现在只剩下 16 个系统调用了，我们再次对其进行同样条件的压力测试，结果如下所示：

```
Concurrency Level:      100
Time taken for tests:   1.326 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      4290858 bytes
HTML transferred:       1510302 bytes
Requests per second:    7541.02 [#/sec] (mean)
Time per request:       13.261 [ms] (mean)
Time per request:       0.133 [ms] (mean, across all concurrent requests)
Transfer rate:          3159.91 [Kbytes/sec] received
```

此时，吞吐率比之前大幅增加，可见，系统调用的减少对于降低请求处理时间有着不可忽略的作用。

3.4 内存分配

我们都知道，在 Web 服务器的工作过程中，需要用到大量的内存，这使得内存的分配和释放工作尤为重要。对于传统的应用，各类表达式所消耗的最大开销就在于中间临时变量的内存分配及数据复制时间，而对于 Web 服务器处理成千上万的 HTTP 请求而言，内存堆栈的分配和复制次数变得更加频繁。我们可以通过改善数据结构和算法复杂度来适当减少数据复制时间，而对于内存的分配，很多 Web 服务器使用了各自的策略来提高效率。

Apache 在运行时的内存使用量是非常惊人的，在前面介绍进程切换的时候我们曾经通过 top 看到 Apache 子进程的内存使用情况，这主要归咎于它的多进程模型。Apache 使用了基于内存池策略的内存管理方案，并将它抽象出来后移入 APR 库中作为通用内存管理模块。这种方案使得 Apache 在运行开始时便一次性申请大片的内存作为内存池，这样在随后需要的时候只要在内存池中直接获取，而不需要再次分配，我们知道频繁的内存分配和释放会引发一定时间的内存整理，这本身便影响了性能。

另一方面，内存池的使用使得 Apache 的内存管理更加安全，因为即便是某处内存使用后忘记释放也没有关系，内存池在 Apache 关闭时会彻底释放。

即便是使用了内存池，由于机制问题，Apache 仍然就像拖着沉重身体的傻大个，内存池对于性能的弥补微不足道。

相比之下，对于使用了单进程模型的 Lighttpd，内存的使用量要小很多，前面我们也曾经

用 top 观察过。另外值得一提的是，同样是单进程模型的 Nginx（读作 Engine X），它的内存使用量更小。下面我们对 Nginx 进行压力测试，同时看看内存的使用情况。

为了体现 Nginx 在处理大量请求时的内存表现，我们使用 151 字节的静态文件作为请求目标，测试结果如下所示：

```
Server Software:        nginx/0.7.30
Server Hostname:        localhost
Server Port:            8002
Document Path:          /test.htm
Document Length:        151 bytes
Concurrency Level:      1000
Time taken for tests:   4.736 seconds
Complete requests:      50000
Failed requests:        0
Write errors:           0
Total transferred:      18134390 bytes
HTML transferred:       7564345 bytes
Requests per second:    10556.76 [#/sec] (mean)
Time per request:       94.726 [ms] (mean)
Time per request:       0.095 [ms] (mean, across all concurrent requests)
Transfer rate:          3739.07 [Kbytes/sec] received
```

测试结果看起来不错，不过我们这里关心的是它在这种压力下的内存表现，我们通过 top 获得如下结果：

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  SWAP COMMAND
14506 daemon    15   0  3036  960  504 R   75  0.0   1:43.46 2076 nginx
```

可见，nginx 进程的物理内存消耗只有 960KB。我们用 ps 来看看 Nginx 此刻的运行时进程结构：

```
14505 ?        Ss     0:00 nginx: master process ./nginx
14506 ?        R      0:06  \_ nginx: worker process
```

可以看到，我们将 Nginx 配置为使用一个 worker 子进程，进程 ID 为 14506，这也正是前面 top 中显示的进程。

Nginx 对于内存方面的良好表现，也正是来自于它的内存分配策略，它可以使用多线程来处理请求，这使得多个线程之间可以共享内存资源，从而令它的内存总体使用量大大减少，另外，它使用分阶段的内存分配策略，按需分配，及时释放，使得内存使用量保持在很小的数量范围。Nginx 的设计初衷便在于支持较大的并发连接数，而内存是否足够往往是阻碍这一目标的关键因素，所以 Nginx 在内存管理方案的设计上花了不少工夫，官方介绍中声称，Nginx 维持 10000 个非活跃 HTTP 持久连接只需要 2.5MB 的内存：

"10,000 inactive HTTP keep-alive connections take about 2.5M memory"

的确，内存分配策略的设计是 Web 服务器并发处理能力的重要保障。

3.5 持久连接

持久连接（Keep-Alive）有时候也称为长连接，它本身是 TCP 通信的一种普通方式，即在一次 TCP 连接中持续发送多份数据而不断开连接，与它相反的方式称为短连接，也就是建立连接后发送一份数据便断开，然后再次建立连接发送下一份数据，周而复始。一般而言，是否采用持久连接，完全取决于应用的特点和需要。从性能的角度看，建立 TCP 连接的操作本身便是一项不小的开销，所以在允许的情况下，连接次数越少，越有利于性能的提升。

在 Web 应用层通信中，由于 HTTP 的无状态特性，使得 HTTP 通信毫不依赖于 TCP 长连接，长久以来大家习惯了“一次性”的 HTTP 通信，即一次 TCP 连接处理一个 HTTP 请求。然而回归 TCP 传输层，长连接带来的好处显而易见，它对于密集型的图片或网页等小数据请求处理有着明显的加速作用。HTTP/1.1 对长连接有了完整的定义，同时，基于标准化的协议，很多浏览器和 Web 服务器也都纷纷提供了对于长连接的支持。

可以想象，HTTP 长连接的实施需要浏览器和 Web 服务器的共同协作，缺一不可。一方面，浏览器需要保持一个 TCP 连接并重复利用，不断地发送多个请求，另一方面，服务器不能过早地主动关闭连接。要实现这一点并不难，目前的浏览器普遍支持长连接，表现在其发出的 HTTP 请求数据头中包含关于长连接的声明，如下所示：

```
Connection: Keep-Alive
```

这种声明的含义在于告诉服务器，“如果可以的话，请让我重用这个连接”。言下之意就是告诉服务器不要在处理完当前请求后就马上关闭连接。

同时，在 Web 服务器上也要打开长连接的支持，幸运的是，目前所有主流的 Web 服务器软件都支持长连接，比如在 Apache 2.2.11 中，长连接的支持默认为开启状态，当然你也可以通过以下方法关闭：

```
KeepAlive Off
```

当我们不希望浏览器和 Web 服务器使用长连接方式时，我们可以关闭服务器的长连接支持，这也是唯一的办法，因为我们无权去修改每个用户的浏览器设置。

对于长连接的有效使用，其关键的一点在于长连接超时时间的设置，即长连接在什么时候被关闭呢？这个设置同时出现在浏览器和 Web 服务器上，因为双方都可以主动关闭。对于 IE7，默认的超时时间为 1 分钟，你也可以通过修改注册表来修改超时时间。对于 Web 服务器，一般会提供超时时间的配置参数，比如在 Apache 中，可以通过 httpd.conf 中的如下参数进行配置：

```
KeepAliveTimeout 30
```

以上的命令设置超时时间为 30 秒，而在默认情况下，Apache 将其设置为 5 秒。值得注意的是，浏览器和 Web 服务器各自的超时时间设置不一定一致，所以在实际运行中，是以最短的超时时间为准。

了解了这些内容后，我们来看看由浏览器发起的一个支持长连接的 HTTP 请求：

```
GET /test.htm HTTP/1.1
Accept: */*
Accept-Language: zh-cn
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET
CLR 2.0.50727; CIBA)
Host: www.test.com
Connection: Keep-Alive
```

服务器的 HTTP 响应头如下所示：

```
HTTP/1.1 304 Not Modified
Date: Sun, 15 Feb 2009 07:58:56 GMT
Server: Apache/2.2.11 (Unix) PHP/5.2.1 DAV/2 SVN/1.4.3
Connection: Keep-Alive
Keep-Alive: timeout=5, max=100
```

在以上请求和响应数据头中，粗体部分体现了长连接的支持和超时时间的设置。

我们前面说过，在请求大量小文件的时候，长连接的有效使用可以减少大量重新建立连接的开销，有效加速性能，ab 的启动选项参数中有对长连接的支持，也就是模拟浏览器发起支持长连接的 HTTP 请求。我们用 ab 来试试 Apache 的 prefork 模式在长连接下的表现。

首先，我们不使用长连接，请求一个 122 字节的图片，ab 测试结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        localhost
Server Port:            80
Document Path:          /images/tips.gif
Document Length:        122 bytes
Concurrency Level:      100
Time taken for tests:   1.567 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      4002400 bytes
HTML transferred:       1220732 bytes
Requests per second:    6381.64 [#/sec] (mean)
Time per request:       15.670 [ms] (mean)
Time per request:       0.157 [ms] (mean, across all concurrent requests)
Transfer rate:          2494.32 [Kbytes/sec] received
```

根据前面的多项测试，这次的结果看起来似乎在我们的预料之中，下面我们使用长连接模

式进行测试，结果如下所示：

```
s-colin:~ # ab -n10000 -c100 -k http://localhost/images/tips.gif
Server Software:        Apache/2.2.11
Server Hostname:        localhost
Server Port:            80
Document Path:          /images/tips.gif
Document Length:        122 bytes
Concurrency Level:      100
Time taken for tests:   0.903 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Keep-Alive requests:    9910
Total transferred:      4365959 bytes
HTML transferred:       1220000 bytes
Requests per second:    11078.98 [#/sec] (mean)
Time per request:       9.026 [ms] (mean)
Time per request:       0.090 [ms] (mean, across all concurrent requests)
Transfer rate:          4723.67 [Kbytes/sec] received
```

这次测试结果是否超乎你的想象呢？从结果中可以看出，吞吐率大幅度提高，使用了长连接的请求为 9910 个，也就是说这些请求没有专门建立连接，而是重用了已经建立好的连接。

我们再来看看 Nginx 在长连接情况下的表现，这次我们使用 strace 的系统调用统计功能，可以直观地进行时间对比。首先我们不使用长连接，对 Nginx 从压力测试开始到结束进行跟踪，结果如下所示：

```
% time     seconds  usecs/call     calls    errors     syscall
------ ----------- ----------- --------- --------- ----------------
 30.78    0.001964           0     10587           writev
 16.55    0.001056           0     21174           close
 13.84    0.000883           0     10587       492 sendfile64
 11.66    0.000744           0     10587           recv
  9.21    0.000588           0     10587           accept
  5.33    0.000340           0     10592           epoll_wait
  4.40    0.000281           0     10587           open
  3.89    0.000248           0     10587           write
  1.49    0.000095           0     10593           gettimeofday
  1.24    0.000079           0     10587           ioctl
  1.22    0.000078           0     10587           fstat64
  0.39    0.000025           0     10587           epoll_ctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.006381              137642       492     total
```

然后，我们使用长连接方式对其进行同样条件的压力测试并跟踪，结果如下所示：

```
% time     seconds  usecs/call     calls    errors     syscall
------ ----------- ----------- --------- --------- ----------------
 32.74    0.000641           0     10442           writev
 30.80    0.000603           0     10442       293 sendfile64
 13.13    0.000257           0     10442           open
 12.87    0.000252           0     10442           write
```

5.26	0.000103	0	10649	207	recv
2.35	**0.000046**	**0**	**10884**		**close**
1.12	0.000022	0	10442		fstat64
0.87	0.000017	0	444		gettimeofday
0.46	0.000009	0	443		epoll_wait
0.41	0.000008	0	442		ioctl
0.00	0.000000	0	7		brk
0.00	0.000000	0	442		epoll_ctl
0.00	**0.000000**	**0**	**442**		**accept**
0.00	0.000000	0	149		setsockopt
100.00	0.001958		76112	500	total

可以看出，在长连接的情况下，总的系统调用数减少了一半左右，其中 socket 的 accept() 和 close()的减少量在我们的意料之中，注意它们消耗的时间，比不使用长连接的时候减少了很多，这在整体时间中占有不小的比例。

对于以上测试方法，长连接超时时间的不同设置带来的性能差异不容易测试，因为不论 Web 服务器的超时时间为多少，ab 总是在发完所有请求后关闭所有连接，所以不存在超时的情况。那么超时时间的长短究竟对 Web 服务器性能有何影响呢？不难想象，对于 Apache 这样的多进程模型来说，如果长连接超时时间过长，比如 60 秒，那么即便是浏览器没有任何请求，而 Apache 仍然维持着连接该浏览器的子进程，一旦并发用户数较多，那么 Apache 将维持着大量的空闲进程，严重影响了服务器性能。另一方面，对于使用多线程的轻量级 Web 服务器（如 Nginx），长连接超时时间过长同样也不是一件好事，因为有时由于超时时间过长导致资源无效占有而引发的损失已经超过了由于重复连接所造成的损失，这的确得不偿失。

事实上，持久连接无处不在，除了以上在 HTTP 通信中的应用，它还可以应用在任何涉及 TCP 通信的应用中，比如 MySQL 数据访问时的持久连接以及连接池，这部分内容我们会在后面详细探讨。总之不论如何应用，持久连接的动机很简单，那就是尽量减少连接次数，尽量重用连接通道。

3.6 I/O 模型

有人说，比特天生就是用来被复制的，数据的生命意义便在于输入/输出，的确，计算机的重要工作之一便是负责各种设备的数据输入/输出，也就是 I/O（In/Out）操作。

事实上，I/O 操作根据设备的不同分为很多种类型，比如内存 I/O、网络 I/O、磁盘 I/O。对于内存 I/O，一般我们在讨论 Web 站点性能时很少提及，因为相比于后两种 I/O 操作，内存 I/O 的速度已经足够快了，前面我们介绍过网络数据传输中的瓶颈往往在于带宽最低的交换节点，类似地，计算机性能的瓶颈往往并不在于内存 I/O 本身。

对于网络 I/O 和磁盘 I/O，它们的速度要慢很多，尽管使用 RAID 磁盘阵列可以通过并行

磁盘访问来加快磁盘 I/O 速度，购买大量独享网络带宽以及使用高带宽网络适配器可以提高网络 I/O 的速度。但问题在于，这些 I/O 操作需要由内核系统调用来完成，同时系统调用显然需要由 CPU 来调度，而 CPU 的速度毫无疑问是非常快的，这就使得 CPU 不得不浪费宝贵的时间来等待慢速 I/O 操作。

尽管我们通过多进程等方式来充分利用空闲的 CPU 资源，但我们还是希望能够让 CPU 花费足够少的时间在 I/O 操作的调度上，这样就可以腾出更多的 CPU 时间来完成更多的 I/O 操作，事实上，如何让高速的 CPU 和慢速的 I/O 设备更好地协调工作，这是从现代计算机诞生到现在一直在探讨的话题，很多技术和策略都围绕它们展开。

针对本章 Web 服务器并发处理能力的讨论范畴，我们所关注的 I/O 操作主要是网络数据的接收和发送，以及磁盘文件的访问，我们将其归纳为多种模型，称为 I/O 模型，它们的本质区别便在于 CPU 的参与方式。

PIO 与 DMA

在介绍 I/O 模型之前，有必要简单地说说慢速 I/O 设备和内存之间的数据传输方式。

我们拿磁盘来说，很早以前，磁盘和内存之间的数据传输是需要 CPU 控制的，也就是说如果我们读取磁盘文件到内存中，数据要经过 CPU 存储转发，这种方式称为 PIO。显然这种方式非常不合理，需要占用大量的 CPU 时间来读取文件，造成文件访问时系统几乎停止响应。

后来，DMA（直接内存访问，Direct Memory Access）取代了 PIO，它可以不经过 CPU 而直接进行磁盘和内存的数据交换。在 DMA 模式下，CPU 只需要向 DMA 控制器下达指令，让 DMA 控制器来处理数据的传送即可，DMA 控制器通过系统总线来传输数据，传送完毕再通知 CPU，这样就在很大程度上降低了 CPU 占有率，大大节省了系统资源，而它的传输速度与 PIO 的差异其实并不十分明显，因为这主要取决于慢速设备的速度。

可以肯定的是，PIO 模式的计算机我们现在已经很少见到了。

同步阻塞 I/O

说到阻塞，首先得说说 I/O 等待。造成等待的原因非常多，比如 Web 服务器在等待用户的访问，这便是等待，因为它不知道谁会来访问，所以只能等。随后，当某个用户通过浏览器发出请求，Web 服务器与该浏览器建立 TCP 连接后，又要等待用户发出 HTTP 请求数据，用户的请求数据在网络上传输需要时间，进入服务器接收缓冲区队列以及被复制到进程地址空间都需要时间。另外，假如浏览器和 Web 服务器采用 HTTP 长连接模式，那么

在超时关闭连接之前，服务器还要等待浏览器发送其他的请求，这也是 I/O 等待。

再比如，读取磁盘上某个文件的 I/O 操作，可能先要等待其他的磁盘访问操作，因为磁头数量是有限的，所以只能一个个排队读取，即使轮到了自己，在磁盘上读取数据本身也要花费时间。值得一提的是，对于 RAID 磁盘的某些规格（如 RAID0），通过磁盘阵列将数据分布在多个磁盘上，大大提高了磁盘访问吞吐率。

可见，I/O 等待是不可避免的，那么既然有了等待，就会有阻塞，但是注意，我们说的阻塞是指当前发起 I/O 操作的进程被阻塞，并不是 CPU 被阻塞，事实上没有什么能让 CPU 阻塞的，CPU 只知道拼命地计算，对于阻塞一无所知。

另外，“同步”的概念在这里显得并不那么重要，只是为了和后面的异步 I/O 加以区分，我们在介绍异步事件通知和异步 I/O 的时候再来探讨同步和异步的区别。但是需要说明的是，对于磁盘文件的访问，也有一个所谓“同步”的选项，即使用 O_SYNC 标志打开文件。在规范情况下，对磁盘文件调用 read()将阻塞进程，一直到数据被复制到进程用户态内存空间，而对磁盘文件调用 write()则不同，它会在数据被复制到内核缓冲区后立即返回。如果使用 O_SYNC 标志打开文件，则对写文件操作产生影响，它使得 write()必须等待数据真正写入磁盘后才返回。

同步阻塞 I/O 是指当进程调用某些涉及 I/O 操作的系统调用或库函数时，比如 accept()、send()、recv()等，进程便暂停下来，等待 I/O 操作完成后再继续运行。这是一种简单而有效的 I/O 模型，它可以和多进程结合起来有效地利用 CPU 资源，但是其代价就是多进程的大量内存开销。

比如在 Apache-prefork 模型中，某个子进程在等待请求时，进程阻塞在 accept()调用，我们用 strace 进行跟踪，如下所示：

```
s-colin:~ # strace -p 17520
Process 17520 attached - interrupt to quit
accept(3,
```

可以看出，accept()在等待用户连接的到达，同时该进程停在此处成为阻塞状态。

为了简单说明同步阻塞 I/O 以及和其他 I/O 模型的区别，我们举个有意思的例子。比如你去逛街，逛着逛着有点饿了，这时你看到商场里有小吃城，就去一个小吃店买一碗面条，交了钱，可面条做起来得需要时间，你也不知道什么时候可以做好，没办法，只好坐在那里等，等面条做好后吃完再继续逛街。显然，这里的吃面条便是 I/O 操作，它要等待厨师做面条，还要等待自己把面条吃完。

同步非阻塞 I/O

在同步阻塞 I/O 中，进程实际上等待的时间可能包括两部分，一个是等待数据的就绪，另一个是等待数据的复制，对于网络 I/O 来说，前者的时间可能要更长一些。

与此不同的是，同步非阻塞 I/O 的调用不会等待数据的就绪，如果数据不可读或者不可写，它会立即告诉进程。比如我们使用非阻塞 recv()接收网络数据的时候，如果网卡缓冲区中没有可接收的数据，函数就及时返回，告诉进程没有数据可读了。相比于阻塞 I/O，这种非阻塞 I/O 结合反复轮询来尝试数据是否就绪，防止进程被阻塞，最大的好处便在于可以在一个进程里同时处理多个 I/O 操作。

但正是由于需要进程执行多次的轮询来查看数据是否就绪，这花费了大量的 CPU 时间，使得进程处于忙碌等待状态。

回到买面条的故事中，假如你不甘心坐着等面条做好，想去顺便逛逛街，可又担心面条做好后没有及时领取，所以你逛一会便跑回去看看面条是否做好，往返了很多次，最后虽然及时地吃上了面条，但是却累得气喘吁吁。

非阻塞 I/O 一般只针对网络 I/O 有效，我们只要在 socket 的选项设置中使用 O_NONBLOCK 即可，这样对于该 socket 的 send()或 recv()便采用非阻塞方式。值得注意的是，对于磁盘 I/O，非阻塞 I/O 并不产生效果。

多路 I/O 就绪通知

在实际应用中，特别是 Web 服务器，同时处理大量的文件描述符是必不可少的，但是使用同步非阻塞 I/O 显然不是最佳的选择，在这种模型下，我们知道如果服务器想要同时接收多个 TCP 连接的数据，就必须轮流对每个 socket 调用接收数据的方法，比如 recv()。不管这些 socket 有没有可以接收的数据，都要询问一遍，假如大部分 socket 并没有数据可以接收，那么进程便会浪费很多 CPU 时间用于检查这些 socket，这显然不是我们所希望看到的。

多路 I/O 就绪通知的出现，提供了对大量文件描述符就绪检查的高性能方案，它允许进程通过一种方法来同时监视所有文件描述符，并可以快速获得所有就绪的文件描述符，然后只针对这些文件描述符进行数据访问。

回到买面条的故事中，假如你不止买了一份面条，还在其他几个小吃店买了饺子、粥、馅饼等，因为一起逛街的朋友看到你的面条后也饿了。这些东西都需要时间来制作。在同步非阻塞 I/O 模型中，你要轮流不停地去各个小吃店询问进度，痛苦不堪。现在引入多路 I/O 就绪通知后，小吃城管理处给大厅安装了一块电子屏幕，以后所有小吃店的食物做好后，

都会显示在屏幕上，这可真是个好消息，你只需要间隔性地看看大屏幕就可以了，也许你还可以同时逛逛附近的商店，在不远处也可以看到大屏幕。

需要注意的是，I/O 就绪通知只是帮助我们快速获得就绪的文件描述符，当得知数据就绪后，就访问数据本身而言，仍然需要选择阻塞或非阻塞的访问方式，一般我们选择非阻塞方式，以防止任何意外的等待阻塞整个进程，比如有时就绪通知只代表一个内核的提示，也许此时文件描述符尚未真正准备好或者已经被客户端关闭连接。

由于平台和历史等原因，多路 I/O 就绪通知有很多不同的实现，在检查大量文件描述符时的性能也存在一定的差异。说到这里，我们有必要简单介绍一下 UNIX 家族的历史，这有助于我们更加深刻地了解这些依赖于平台的技术。

记住，最早的 UNIX 诞生于贝尔实验室，在随后的十年，UNIX 在学术机构和大型企业中被广泛应用。贝尔实验室是 AT&T 收购了西方电子公司的研究部门后设立的一个独立实体，AT&T 这时以廉价的许可将 UNIX 的源代码授权给一些研究机构和大学，供它们研究和教学使用，这使得很多机构在此基础上对 UNIX 进行了完善和扩展，促进了 UNIX 的发展，同时产生了一些新的变种版本，其中最著名的变种之一便是由加州大学 Berkeley 分校开发的 BSD，这一变种对 UNIX 有着重大贡献和深远影响。

由于 BSD 开始被很多企业广泛采用，不久之后，AT&T 意识到了 UNIX 的商业价值，开始停止源代码授权，同时为了统一混乱的 UNIX 版本，AT&T 综合了其他大学和企业开发的各种 UNIX，开发了 UNIX System V Release 1。

这时候 BSD 已经非常成熟，比如 select 就诞生于这时候的 4.2BSD，而 TCP/IP 则诞生于 4.1BSD，它的代码也成为以后几乎所有操作系统 TCP/IP 实现代码的前辈，包括 Windows。BSD 不断增大的影响力终于引起了 AT&T 的关注，AT&T 将 BSD 告上了法庭，一场源代码版权官司开始了，一直持续到 AT&T 将 UNIX 实验项目卖给 Novell。

Novell 采取了开明的态度，它允许 BSD 自由开发，但前提是必须删除来自 AT&T 的代码，就这样，BSD 诞生了全新的版本 4.4BSD Lite，这个版本不存在法律问题，所以它成为很多现代自由版 UNIX 的基础，它们和 UNIX 以及 Linux 一起，共同成为 UNIX 大家族的成员。

BSD 在发展中也逐渐衍生出 3 个主要的分支：FreeBSD、OpenBSD 和 NetBSD。

在此后的几十年中，UNIX 仍在不断变化，其版权所有者不断变更，授权者的数量也在增加。有很多大公司在取得了 UNIX 的授权之后，开发了自己的 UNIX 产品，比如 IBM 的 AIX、HP 的 HPUX、Sun 的 Solaris、SGI 的 IRIX，以及 Apple 的 MacOS。

正是在 UNIX 之间的利益纷争及其商业化运作的背景下，乱世出英雄，Linux 诞生了，它

天生便吸取了 UNIX 的教训，在版权问题上直接采用自由开发的 GPL 开源许可，这种开发精神使得 Linux 就像当年的 BSD 一样迅速发展，同时又由于 GPL 的使用，很好地规避了变种并存的情况。

这真是一部充满斗争的历史，它深深影响到几十年后的我们，本章介绍的技术基本都来自于那个时代，所以当你被很多相似的实现方法搞得眼花缭乱时，想想那也许只是多年前的一个历史错误。

以下的实现方法，我们只侧重介绍其优势和局限性，目的在于真正了解它们在何种场景下表现出色，至于如何使用它们来编写代码，超出了本书的讨论范围，你可以查看手册或者《UNIX 网络编程》。

select

select 最早于 1983 年出现在 4.2BSD 中，它通过一个 select()系统调用来监视包含多个文件描述符的数组，当 select()返回后，该数组中就绪的文件描述符便会被内核修改标志位，使得进程可以获得这些文件描述符从而进行后续的读写操作。

select 目前几乎在所有的平台上都支持，其良好跨平台支持也是它的一个优点，事实上从现在看来，这也是它所剩不多的优点之一。

select 的一个缺点在于单个进程能够监视的文件描述符的数量存在最大限制，在 Linux 上一般为 1024，不过可以通过修改宏定义甚至重新编译内核的方式提升这一限制。所以，假如使用了 select 的服务器已经维持了 1024 个连接，那么你的请求可能会被拒绝。

另外，select()所维护的存储大量文件描述符的数据结构，随着文件描述符数量的增大，其复制的开销也线性增长。同时，由于网络响应时间的延迟使得大量 TCP 连接处于非活跃状态，但调用 select()会对所有 socket 进行一次线性扫描，所以这也浪费了一定的开销。

poll

poll 在 1986 年诞生于 System V Release 3，显然 UNIX 不愿意直接沿用 BSD 的 select，而是自己重新实现了一遍，它和 select 在本质上没有多大差别，但是 poll 没有最大文件描述符数量的限制。

poll 和 select 同样存在的一个缺点就是，包含大量文件描述符的数组被整体复制于用户态和内核的地址空间之间，而不论这些文件描述符是否就绪，它的开销随着文件描述符数量的增加而线性增大。

另外，select()和 poll()将就绪的文件描述符告诉进程后，如果进程没有对其进行 I/O 操作，那么下次调用 select()或 poll()的时候将再次报告这些文件描述符，所以它们一般不会丢失

就绪的消息，这种方式称为水平触发（Level Triggered）。

SIGIO

Linux 2.4 提供了 SIGIO，它通过实时信号（Real Time Signal）来实现 select/poll 的通知方法，但是它们的不同在于，select/poll 告诉我们哪些文件描述符是就绪的，一直到我们读写它之前，每次 select/poll 都会告诉我们；而 SIGIO 则是告诉我们哪些文件描述符刚刚变为就绪状态，它只说一遍，如果我们没有采取行动，那么它将不会再次告知，这种方式称为边缘触发（Edge Triggered）。SIGIO 几乎是 Linux 2.4 下性能最好的多路 I/O 就绪通知方法。

但是，SIGIO 也存在一些缺点，在 SIGIO 机制中，代表事件的信号由内核中的事件队列来维护，信号按照顺序进行通知，这可能导致当一个信号到达的时候，该事件已经过期，它所描述的文件描述符已经被关闭。另一方面，事件队列是有长度限制的，无论你设置多大的上限，总有可能被事件装满，这就很容易发生事件丢失，所以这时候需要采用其他方法来弥补损失。

/dev/poll

Sun 在 Solaris 中提供了新的实现方法，它使用了虚拟的/dev/poll 设备，你可以将要监视的文件描述符数组写入这个设备，然后通过 ioctl()来等待事件通知。当 ioctl()返回就绪的文件描述符后，你可以从/dev/poll 中读取所有就绪的文件描述符数组，这点类似于 SIGIO，节省了扫描所有文件描述符的开销。

在 Linux 下有很多方法可以实现类似/dev/poll 的设备，但是都没有提供直接的内核支持，这些方法在服务器负载较大时性能不够稳定。

/dev/epoll

随后，名为/dev/epoll 的设备以补丁的形式出现在 Linux 2.4 上，它提供了类似/dev/poll 的功能，而且增加了内存映射（mmap）技术，在一定程度上提高了性能，后面会详细介绍内存映射的内容。

但是，/dev/epoll 仍然只是一个补丁，Linux 2.4 并没有将它的实现加入内核。

epoll

直到 Linux 2.6 才出现了由内核直接支持的实现方法，那就是 epoll，它几乎具备了之前所说的一切优点，被公认为 Linux 2.6 下性能最好的多路 I/O 就绪通知方法。

epoll 可以同时支持水平触发和边缘触发，理论上边缘触发的性能要更高一些，但是代码

实现相当复杂，因为任何意外的丢失事件都会造成请求处理错误。在默认情况下 epoll 采用水平触发，如果要使用边缘触发，则需要在事件注册时增加 EPOLLET 选项。在 Lighttpd 的 epoll 模型代码（src/fdevent_linux_sysepoll.c）中，可以看到它注释掉了 EPOLLET，并没有使用边缘触发方式，如下所示：

```
ep.events |= EPOLLERR | EPOLLHUP /* | EPOLLET */;
```

而在 Nginx 的 epoll 模型代码（src/event/modules/ngx_epoll_module.c）中，可以看到它使用了边缘触发，如下所示：

```
ee.events = EPOLLIN|EPOLLOUT|EPOLLET;
```

另外，epoll 同样只告知那些就绪的文件描述符，而且当我们调用 epoll_wait()获得就绪文件描述符时，返回的并不是实际的描述符，而是一个代表就绪描述符数量的值，你只需要去 epoll 指定的一个数组中依次取得相应数量的文件描述符即可，这里也使用了内存映射（mmap）技术，这样便彻底省掉了这些文件描述符在系统调用时复制的开销。

另一个本质的改进在于 epoll 采用基于事件的就绪通知方式。在 select/poll 中，进程只有在调用一定的方法后，内核才对所有监视的文件描述符进行扫描，而 epoll 事先通过 epoll_ctl()来注册每一个文件描述符，一旦某个文件描述符就绪时，内核会采用类似 callback 的回调机制，迅速激活这个文件描述符，当进程调用 epoll_wait()时便得到通知。

回到买面条的故事中，虽然有了电子屏幕，但是显示的内容是所有餐品的状态，包括正在制作的和已经做好的，这显然给你造成阅读上的麻烦，就好像 select/poll 每次返回所有监视的文件描述符一样。如果能够只显示做好的餐品，那该多好，随后小吃城管理处改进了大屏幕，实现了这一点，这就像/dev/poll 一样只告知就绪的文件描述符。在显示做好的餐点时，如果只显示一次，而不管你有没有看到，这就相当于边缘触发，而如果在你领取餐点之前，每次都显示，就相当于水平触发。

但尽管这样，一旦你走得比较远，就还得花时间走到小吃城去看电子屏幕，能不能让你更加轻松地获得通知呢？管理处这次采用了手机短信通知的方法，你只需要到管理处注册后，便可以在餐点就绪时及时收到短信通知，这类似于 epoll 的事件机制。

kqueue

FreeBSD 中实现了 kqueue，它像 epoll 一样可以设置水平触发或边缘触发，同时 kqueue 还可以用来监视磁盘文件和目录，但是它的 API 在很多平台都不支持，而且文档相当匮乏。kqueue 和 epoll 的性能非常接近。

对于以上介绍的多种实现方法，在 Linux 平台的 Web 服务器中，我们常用的主要有 select、poll、SIGIO、epoll，尤其是在 Linux 2.6 中，epoll 成为首要选择。

也许你没有特别留意，在安装一些 Web 服务器软件的时候，configure 过程中会检查当前系统支持的多路 I/O 就绪通知方法，比如 Nginx 在 configure 时，其中一部分内容如下所示：

```
 + rt signals found
checking for epoll ... found
checking for poll() ... found
checking for /dev/poll ... not found
checking for kqueue ... not found
```

在本章随后关于并发策略的内容中，我们会将这些多路 I/O 就绪通知方法应用在其中，并进行测试对比。

内存映射

Linux 内核提供一种访问磁盘文件的特殊方式，它可以将内存中某块地址空间和我们要指定的磁盘文件相关联，从而把我们对这块内存的访问转换为对磁盘文件的访问，这种技术称为内存映射（Memory Mapping）。

在大多数情况下，使用内存映射可以提高磁盘 I/O 的性能，它无须使用 read()或 write()等系统调用来访问文件，而是通过 mmap()系统调用来建立内存和磁盘文件的关联，然后像访问内存一样自由地访问文件。

有两种类型的内存映射，共享型和私有型，前者可以将任何对内存的写操作都同步到磁盘文件，而且所有映射同一个文件的进程都共享任意一个进程对映射内存的修改；后者映射的文件只能是只读文件，所以不可以将对内存的写同步到文件，而且多个进程不共享修改。显然，共享型内存映射的效率偏低，因为如果一个文件被很多进程映射，那么每次的修改同步将花费一定的开销。

Apache 2.x 中使用了内存映射，前面我们在探讨系统调用的时候，使用 strace 跟踪了 Apache 子进程，当时我们请求的 URL 指向一个很小的静态文件，只有 151 字节，Apache 对于较小的静态文件，选择使用内存映射来读取，我们重新来看看其中的一块片段：

```
    open("/data/www/site/htdocs/test.htm", O_RDONLY|O_LARGEFILE) = 20
    mmap2(NULL, 151, PROT_READ, MAP_SHARED, 20, 0) = 0xb7f8f000
    writev(19,  [{"HTTP/1.1  200  OK\r\nDate:  Thu,  12  F"...,  278},
{"<html>\n<head>\n<title>Welcome to "..., 151}], 2) = 429
    munmap(0xb7f8f000, 151)                 = 0
```

我们访问的文件是/data/www/site/htdocs/test.htm，Apache 使用 open()系统调用打开这个文件，获得文件描述符为 20，然后通过 mmap2()系统调用完成了共享型内存映射的关联，mmap2()在 mmap()的基础上进行了一定的扩展，原理是一样的。随后，Apache 读取文件中的内容，这个操作并没有体现出系统调用，因为它只是进程读取地址空间数据的用户态行为。接下来，Apache 使用 writev()系统调用将 HTTP 响应数据的头信息和 151 字节的正文数据合并后发送，然后调用 munmap()来撤销映射。

直接 I/O

在 Linux 2.6 中，内存映射和直接访问文件没有本质上差异，因为数据从进程用户态内存空间到磁盘都要经过两次复制，即在磁盘与内核缓冲区之间以及在内核缓冲区与用户态内存空间。

引入内核缓冲区的目的在于提高磁盘文件的访问性能，因为当进程需要读取磁盘文件时，如果文件内容已经在内核缓冲区中，那么就不需要再次访问磁盘；而当进程需要向文件中写入数据时，实际上只是写到了内核缓冲区便告诉进程已经写成功，而真正写入磁盘是通过一定的策略进行延迟的。

然而，对于一些较复杂的应用，比如数据库服务器，它们为了充分提高性能，希望绕过内核缓冲区，由自己在用户态空间实现并管理 I/O 缓冲区，包括缓存机制和写延迟机制等，以支持独特的查询机制，比如数据库可以根据更加合理的策略来提高查询缓存命中率。另一方面，绕过内核缓冲区也可以减少系统内存的开销，因为内核缓冲区本身就在使用系统内存。

Linux 提供了对这种需求的支持，即在 open()系统调用中增加参数选项 O_DIRECT，用它打开的文件便可以绕过内核缓冲区的直接访问，这样便有效避免了 CPU 和内存的多余时间开销。

在 MySQL 中，对于 Innodb 存储引擎，其自身可以进行数据和索引的缓存管理，所以它对于内核缓冲区的依赖不是那么重要。MySQL 提供了一种实现直接 I/O 的方法，在 my.cnf 配置中，可以在分配 Innodb 数据空间文件的时候，通过使用 raw 分区跳过内核缓冲区，实现直接 I/O，这在 MySQL 的官方手册上略有介绍，但是不多，主要涉及 raw 分区的使用，这是一种特别的分区，它不能像其他分区格式（比如 ext2）一样通过 mount 来挂接使用，而是需要使用 raw 设备管理程序来加载。为 Innodb 使用 raw 分区的配置如下所示：

```
innodb_data_file_path = /dev/sda5:100Gnewraw
```

假设/dev/sda5 是 raw 分区，在分区大小后面增加 newraw 关键字，便可以将该 raw 分区作为数据空间，并由 Innodb 存储引擎直接访问。具体的操作还涉及其他一些步骤，这里就不具体罗列了。

另外，MySQL 还提供了 innodb_flush_method 配置选项，你可以将它设置为如下形式：

```
innodb_flush_method = O_DIRECT
```

这样便可以通过另一种方式来实现直接 I/O。

顺便提一下，与 O_DIRECT 类似的一个选项是 O_SYNC，后者只对写数据有效，它将写

入内核缓冲区的数据立即写入磁盘，将机器故障时数据的丢失减少到最小，但是它仍然要经过内核缓冲区。

sendfile

大多数时候，我们都在向 Web 服务器请求静态文件，比如图片、样式表等，根据前面的介绍，我们知道在处理这些请求的过程中，磁盘文件的数据先要经过内核缓冲区，然后到达用户内存空间，因为是不需要任何处理的静态数据，所以它们又被送到网卡对应的内核缓冲区，接着再被送入网卡进行发送。

数据从内核出去，绕了一圈，又回到内核，没有任何变化，看起来真是浪费时间。在 Linux 2.4 的内核中，尝试性地引入了一个称为 khttpd 的内核级 Web 服务器程序，它只处理静态文件的请求。引入它的目的便在于内核希望请求的处理尽量在内核完成，减少内核态的切换以及用户态数据复制的开销。

同时，Linux 通过系统调用将这种机制提供给了开发者，那就是 sendfile()系统调用。它可以将磁盘文件的特定部分直接传送到代表客户端的 socket 描述符，加快了静态文件的请求速度，同时也减少了 CPU 和内存的开销。

在 OpenBSD 和 NetBSD 中没有提供对 sendfile 的支持。

还记得在介绍内存映射的时候，我们通过 strace 的跟踪看到了 Apache 在处理 151 字节的小文件时，使用了 mmap()系统调用来实现内存映射，但是在 Apache 处理较大文件的时候，内存映射会导致较大的内存开销，得不偿失，所以 Apache 使用了 sendfile64()来传送文件，sendfile64()是 sendfile()的扩展实现，它在 Linux 2.4 之后的版本中提供。

我们来访问一个 1.2MB 的文件，用 strace 跟踪如下：

```
open("/data/www/site/htdocs/test.mp4", O_RDONLY|O_LARGEFILE) = 20
setsockopt(19, SOL_TCP, TCP_CORK, [1], 4) = 0
writev(19, [{"HTTP/1.1 200 OK\r\nDate: Thu, 12 F"..., 287}], 1) = 287
sendfile64(19, 20, [0], 1186854)        = 48865
setsockopt(19, SOL_TCP, TCP_CORK, [0], 4) = 0
poll([{fd=19, events=POLLOUT, revents=POLLOUT}], 1, 300000) = 1
sendfile64(19, 20, [48865], 1137989)    = 98304
poll([{fd=19, events=POLLOUT, revents=POLLOUT}], 1, 300000) = 1
sendfile64(19, 20, [147169], 1039685)   = 49152
poll([{fd=19, events=POLLOUT, revents=POLLOUT}], 1, 300000) = 1
sendfile64(19, 20, [196321], 990533)    = 49152
poll([{fd=19, events=POLLOUT, revents=POLLOUT}], 1, 300000) = 1
sendfile64(19, 20, [245473], 941381)    = 49152
poll([{fd=19, events=POLLOUT, revents=POLLOUT}], 1, 300000) = 1
sendfile64(19, 20, [294625], 892229)    = 49152
poll([{fd=19, events=POLLOUT, revents=POLLOUT}], 1, 300000) = 1
sendfile64(19, 20, [343777], 843077)    = 49152
```

```
poll([{fd=19, events=POLLOUT, revents=POLLOUT}], 1, 300000) = 1
sendfile64(19, 20, [392929], 793925)    = 49152
poll([{fd=19, events=POLLOUT, revents=POLLOUT}], 1, 300000) = 1
```

这里只列出了我们关注的 20 个系统调用，可以清楚地看到，Apache 使用 open()系统调用来打开我们请求的文件，获得的文件描述符为 20，然后通过 writev()系统调用发送出 HTTP 响应头，接着对 1.2MB 的正文数据使用 sendfile64()进行分段发送，其调用参数中包含了两个文件描述符，分别是代表磁盘文件的 20 和代表客户端的 19。

在这种机制下，我们用 ab 进行压力测试，得到的结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        localhost
Server Port:            80
Document Path:          /test.mp4
Document Length:        1186854 bytes
Concurrency Level:      100
Time taken for tests:   1.692 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      1187141000 bytes
HTML transferred:       1186854000 bytes
Requests per second:    590.93 [#/sec] (mean)
Time per request:       169.224 [ms] (mean)
Time per request:       1.692 [ms] (mean, across all concurrent requests)
Transfer rate:          685080.20 [Kbytes/sec] received
```

sendfile 真的发挥预期的作用了吗？我们关掉 Apache 的 sendfile，再来试一次。Apache 提供了 sendfile 开关，修改 httpd.conf 如下所示：

```
EnableSendfile off
```

这时我们再来请求这个 1.2MB 的文件，用 strace 跟踪如下：

```
open("/data/www/site/htdocs/test.mp4", O_RDONLY|O_LARGEFILE) = 20
writev(19, [{"HTTP/1.1 200 OK\r\nDate: Fri, 13 F"..., 287}], 1) = 287
_llseek(20, 0, [0], SEEK_SET)           = 0
read(20, "\0\0\0\34ftypisom\0\0\2\0isomiso2mp41\0\0\0\10"..., 8192) =
8192
write(19, "\0\0\0\34ftypisom\0\0\2\0isomiso2mp41\0\0\0\10"..., 8192)
= 8192
read(20, "\345,\362\r\266!2Q8?I\277,\306Z\273\26\341i6\313)\240p"...,
8192) = 8192
write(19,  "\345,\362\r\266!2Q8?I\277,\306Z\273\26\341i6\313)\240p"...,
8192) = 8192
read(20, "B\0!\253T\250{\375\3w\362\266#yG\321C\3232L\304\304qX\30"...,
8192) = 8192
write(19,"B\0!\253T\250{\375\3w\362\266#yG\321C\3232L\304\304qX\30"..
., 8192) = 8192
read(20, ">\231\203\272w\303\263m\317LT\246\323\320\300\3507\"\n"...,
8192) = 8192
```

```
    write(19, ">\231\203\272w\303\263m\317LT\246\323\320\300\3507\"\n"...,
8192) = 8192
```

的确没有使用 sendfile64()，而是用普通的 read()和 write()多次复制数据。我们看到，前面 sendfile64()某一次发送了 49152 字节的数据，而这里的 read()和 write()一次发送了 8192 字节的数据，所以需要更多的发送次数，也就是更多次的系统调用。

最后来看看性能上的差距，用 ab 对关闭 sendfile 的 Apache 进行压力测试，结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        localhost
Server Port:            80
Document Path:          /test.mp4
Document Length:        1186854 bytes
Concurrency Level:      100
Time taken for tests:   2.768 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      1187141000 bytes
HTML transferred:       1186854000 bytes
Requests per second:    361.24 [#/sec] (mean)
Time per request:       276.827 [ms] (mean)
Time per request:       2.768 [ms] (mean, across all concurrent requests)
Transfer rate:          418788.44 [Kbytes/sec] received
```

相比于前面的 590.93 reqs/s，这里的吞吐率确实降低了不少。但是，这并不意味着 sendfile 在任何场景下都能发挥显著的作用。对于请求较小的静态文件，sendfile 发挥的作用便显得不那么重要，通过压力测试，我们模拟 100 个并发用户请求 151 字节的静态文件，是否使用 sendfile 的吞吐率几乎是相同的，可见在处理小文件请求时，发送数据的环节在整个过程中所占时间的比例相比于大文件请求时要小很多，所以对于这部分的优化效果自然不十分明显。我们可以用 ab 来大概了解一下两者的比例差别，其中 Processing 部分的时间在这里可以理解为发送文件的时间，而 Connect 则是指建立 TCP 连接的时间。对于小文件的测试结果如下所示：

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   1.1      0       6
Processing:     3   13   2.1     13      16
Waiting:        2   12   2.1     13      15
Total:          3   13   1.9     13      19
```

对于大文件的测试结果如下所示：

```
Connection Times (ms)
              Min  mean[+/-sd] median   max
Connect:        0    1   0.5      1       6
Processing:    50  313  27.7    317     577
Waiting:        1   45  81.2      3     292
Total:         51  314  27.7    319     577
```

结果已经很清楚了，的确，任何一种技术都有它的应用范围，能够意识到环境的不同而选择适合的技术才是明智之举。

异步 I/O

说到这里，就得说说同步 I/O 和异步 I/O 的区别了。在有些时候，同步和异步、阻塞和非阻塞很容易被混用，其实它们完全不是一回事，而且它们修饰的对象也不同。阻塞和非阻塞是指当进程访问的数据如果尚未就绪，进程是否需要等待，简单说这相当于函数内部的实现区别，即未就绪时是直接返回还是等待就绪；而同步和异步是指访问数据的机制，同步一般指主动请求并等待 I/O 操作完毕的方式，当数据就绪后在读写的时候必须阻塞，异步则指主动请求数据后便可以继续处理其他任务，随后等待 I/O 操作完毕的通知，这可以使进程在数据读写时也不发生阻塞。

POSIX1003.1 标准为异步方式访问文件定义了一套库函数，这里的异步 I/O（AIO）实际上就是指当用户态进程调用库函数访问文件时，进行必要的快速注册，比如进入读写操作队列，然后函数马上返回，这时候真正的 I/O 传输还没有开始呢。

可以看出，这种机制是真正意义上的异步 I/O，而且是非阻塞的，它可以使进程在发起 I/O 操作后继续运行，让 CPU 处理和 I/O 操作达到更好的重叠。

POSIX 的标准库中定义了 AIO 的一系列接口，它几乎屏蔽了一切网络通信的细节，所以对使用者而言非常简单。AIO 没有提供非阻塞的 open()方法，所以进程仍然使用 open()系统调用来打开文件，然后填充一些描述 I/O 请求的数据结构，接下来调用 aio_read()或 aio_write()来发起异步 I/O 操作，一旦请求进入操作队列后，函数便返回，进程可以在此后通过 aio_error()来检查正在运行的 I/O 操作的状态。

然而对于 AIO 的实现，不同的平台有不同的方法，它甚至可以完全由库函数来实现而不需要内核的支持，比如通过多线程来模拟非阻塞的 aio_read()调用。但是这样一来，它的性能便大打折扣，变得毫无意义，所以实际上很多平台都没有实现它。

在 Linux 2.6.16 中，AIO 的实现可以在/usr/include/libaio.h 中看到，它采用了一套没有遵循 POSIX AIO 标准的接口，并且实现方式正是基于前面说到的 LinuxThreads 内核级线程库，截至目前，这个功能还在实现中，目前的 Linux AIO 只能用于通过 O_DIRECT 标志打开的文件，在前面的直接 I/O 中我们曾经介绍过 O_DIRECT，此处不再赘述。

3.7 服务器并发策略

从本质上讲，所有到达服务器的请求都封装在 IP 包中，位于网卡的接收缓冲区中，这时

候 Web 服务器软件要做的事情就是不断地读取这些请求，然后进行处理，并将结果写到发送缓冲区，这其中包含了一系列的 I/O 操作和 CPU 计算，而设计一个并发策略的目的，就是让 I/O 操作和 CPU 计算尽量重叠进行，一方面要让 CPU 在 I/O 等待时不要空闲，另一方面让 CPU 在 I/O 调度上尽量花费最少的时间。

还记得游戏“帝国时代”吗？我想很多人都跟我一样喜欢它。熟悉它的玩家都知道，制胜的关键在于高速的经济发展，也就是如何让所有的村民把所有时间都合理地应用在采集资源和新修建筑，这就是每个玩家不断研究的并发策略。

下面我们来看几种常见的 Web 服务器并发策略。

一个进程处理一个连接，非阻塞 I/O

既然一个进程处理一个连接，那么在并发请求同时到达时，服务器必然要准备多个进程来处理请求。

早期的一种方式是采用 fork 模式，由主进程负责 accept()来自客户端的连接，一旦接收连接，便马上 fork()一个新的 worker 进程来处理，处理结束后，这个进程便被销毁。在几年前，我和同事们用 C++编写的 CGI 程序在 Apache 下运行时，便是采用这种方式。fork()的开销成为影响性能的关键。

另一种方式是 prefork 模式，这种方式由主进程预先创建一定数量的子进程，每个请求由一个子进程来处理，但是每个子进程可以处理多个请求。父进程往往只负责管理子进程，根据站点负载来调整子进程的数量，相当于动态维护一个进程池。

对于 accept()的方式，有以下两种策略：

- 主进程使用非阻塞 accept()来接收连接，当建立连接后，主进程将任务分配给空闲的子进程来处理；
- 所有子进程使用阻塞 accept()来竞争接收连接，一旦一个子进程建立连接后，它将继续进行处理。

Apache 2.x 便采用上述第 2 种策略，在这种 accept()阻塞竞争的情况下，虽然从代码上看似只有一个子进程的 accept()可以返回，但实际上，按大多数 TCP 栈的实现方法，当一个请求连接到达时，内核会激活所有阻塞在 accept()的子进程，但只有一个能够成功获得连接并返回到用户空间，而其余的子进程由于得不到连接而继续回到休眠状态，这种“抖动”也造成了一定的额外开销。

对于接收 HTTP 请求数据的 I/O 操作，Apache 采用了非阻塞 read()，我们用 strace 跟踪某

子进程从 accept()获得连接后直到 read()HTTP 请求数据的过程，并且统计系统调用消耗的时间，显示在每个系统调用行末的尖括号中，结果如下所示：

```
    accept(3,        {sa_family=AF_INET6,        sin6_port=htons(50253),
inet_pton(AF_INET6,  "::ffff:127.0.0.1",  &sin6_addr),  sin6_flowinfo=0,
sin6_scope_id=0}, [28]) = 19 <0.004843>
    gettimeofday({1235212465, 460758}, NULL) = 0 <0.000006>
    getsockname(19,       {sa_family=AF_INET6,       sin6_port=htons(80),
inet_pton(AF_INET6,  "::ffff:127.0.0.1",  &sin6_addr),  sin6_flowinfo=0,
sin6_scope_id=0}, [28]) = 0 <0.000007>
    fcntl64(19, F_GETFL)                      = 0x2 (flags O_RDWR) <0.000006>
    fcntl64(19, F_SETFL, O_RDWR|O_NONBLOCK) = 0 <0.000005>
    gettimeofday({1235212465, 460941}, NULL) = 0 <0.000006>
    read(19, "GET /test.htm HTTP/1.0\r\nHost: lo"..., 8000) = 85 <0.000008>
```

以上可以看出 Apache 通过 fcntl64()系统调用将 accept()获得的文件描述符 19 设置为 O_NONBLOCK（非阻塞）模式。

处理完请求后，Apache 接着使用 poll()来检查当前的 socket 连接是否有新的请求数据到达，如下所示：

```
    poll([{fd=19, events=POLLIN, revents=POLLIN|POLLHUP}], 1, 2000) = 1
<0.013797>
    read(19, "", 512)                       = 0 <0.000008>
    close(19)                               = 0 <0.000009>
```

当客户端关闭 socket 连接后，poll()的 POLLHUP 事件会被触发，但有时候会触发 POLLIN 事件，所以 Apache 再次调用 read()来接收数据，如果发现获得 0 字节，Apache 便认为客户端已经关闭，于是调用 close()关闭连接。

如果客户端采用长连接呢？我们跟踪看看，结果如下所示：

```
poll([{fd=19, events=POLLIN, revents=POLLIN}], 1, 30000) = 1 <0.000040>
read(19, "GET /test.htm HTTP/1.0\r\nConnecti"..., 8000) = 109 <0.000007>
```

可以看到，在 poll()获得就绪文件描述符后，read()接收到了一个新的 HTTP 请求数据，长度为 109 字节，并且从粗体字部分也可以看出这是一个长连接的 HTTP 请求。另外，注意以上两次 poll()等待的时间，前者为 0.013797s，后者为 0.000040s，差距相当悬殊。

这回，我们来统计一下各系统调用的时间比例。我们使用 ab 模拟 100 个并发用户请求 151 字节的静态文件，一共 100000 个请求，在没有采用长连接请求的时候，我们用 strace 跟踪某个 Apache 子进程，结果如下所示：

```
% time     seconds  usecs/call     calls    errors    syscall
------ ----------- ----------- --------- --------- ----------------
 74.55    0.016449           4      3846           poll
 12.07    0.002664           1      3846           accept
```

4.39	**0.000969**	**0**	**3846**		**writev**
2.79	**0.000616**	**0**	**26922**	**23076**	**open**
1.24	0.000274	0	7692		close
0.99	0.000218	0	34614		gettimeofday
0.97	0.000214	0	3846		shutdown
0.96	0.000211	0	3846		munmap
0.79	0.000175	0	3846		mmap2
0.49	0.000109	0	11538	3846	read
0.25	0.000055	0	7692		fcntl64
0.19	0.000042	0	3846		stat64
0.15	0.000032	0	3846		write
0.12	0.000027	0	3846		getsockname
0.04	0.000009	0	3846		times
0.00	0.000000	0	1		brk
100.00	0.022064		126919	26922	total

可以看出，这个子进程一共接受了 3846 个请求，而且 poll()和 accept()的调用几乎占用了全部时间，而用于打开磁盘文件的 open()和发送 HTTP 响应数据的 writev()只占了很少的时间比例。

下面我们使用同样的测试条件，但采用长连接方式进行测试，结果如下所示：

% time	seconds	usecs/call	calls	errors	syscall
64.30	**0.001052**	**0**	**10332**	**8856**	**open**
17.60	**0.000288**	**0**	**1476**		**writev**
8.62	0.000141	0	1476		stat64
2.69	0.000044	0	1476		write
2.51	0.000041	0	8903		gettimeofday
1.59	0.000026	0	1476		mmap2
1.47	0.000024	0	1476		munmap
0.67	0.000011	0	1491		close
0.55	0.000009	0	2968	1478	read
0.00	0.000000	0	1476		times
0.00	**0.000000**	**0**	**1476**		**poll**
0.00	0.000000	0	30		fcntl64
0.00	**0.000000**	**0**	**15**		**accept**
0.00	0.000000	0	15		getsockname
0.00	0.000000	0	15	1	shutdown
100.00	0.001636		34101	10335	total

很好，结果在我们的预料之中，accept()的调用次数只有 15 次，所以时间开销几乎为 0，而且 poll()在长连接情况下阻塞时间也很少。至于 open()和 writev()的时间，虽然并没有比之前增加多少，但这次却成为了主要部分。

实际上，通过前面的内容，我们知道 Apache 这种多进程模型的开销限制了它的并发连接数，但是 Apache 也有自身的优势，比如从稳定性和兼容性的角度看，多进程模型的优势正体现在它相对安全的独立进程，任何一个子进程的崩溃都不会影响 Aapche 本身，Apache 父进程可以创建新的子进程；另一方面，Apache 毕竟经过长期的考验和广泛的使用，它

的功能模块非常丰富，比如各种动态脚本的支持、虚拟主机管理、URL Rewrite、SSL 加密、SSI（服务器端静态网页包含）、目录浏览和管理等，而且安装和配置都相当简单，有大量的官方文档可以参考。所以，对于一些并发数要求不高（如 150 以内）的站点，如果同时对其他功能有所依赖，那么 Apache 便是非常不错的选择。

一个线程处理一个连接，非阻塞 I/O

这种方式允许在一个进程中通过多个线程来处理多个连接，其中每个线程处理一个连接。Apache 的 worker 多路处理模块便采用这种方式，它的目的主要在于减少 prefork 模式中太多进程的开销，使 Apache 可以支持更多的并发连接。

Apache 的 worker 模型可以说是一种多进程和多线程的混合方式，它的 MPM 配置有些类似于 prefork，但是增加了每个进程最大线程数以及最大总线程数的配置，根据配置，Apache 主进程会创建很少的子进程，每个子进程又拥有一定数量的线程。

根据 strace 的跟踪，我们发现只有一个子进程负责 accept()，一旦接收到连接后，便将任务分配给适当的线程，线程中的网络 I/O 操作使用非阻塞方式。

在实际测试中，这种方式的表现并不比 prefork 有太大的优势，因为虽然它使用大量线程代替进程，但是这些线程实际上都是由内核进程调度器管理的轻量级进程，它们的上下文切换开销依然存在，我们分别给 worker 和 prefork 两种模型的 Apache 进行同样的压力测试，并用 Nmon 监视上下文切换。

压力测试模拟 100 个并发用户，请求 151 字节的静态文件，一共发送 100000 个请求。prefork 模型的测试结果如下所示：

```
Concurrency Level:      100
Time taken for tests:   17.060 seconds
Complete requests:      100000
Failed requests:        0
Write errors:           0
Total transferred:      42910725 bytes
HTML transferred:       15103775 bytes
Requests per second:    5861.53 [#/sec] (mean)
Time per request:       17.060 [ms] (mean)
Time per request:       0.171 [ms] (mean, across all concurrent requests)
Transfer rate:          2456.27 [Kbytes/sec] received
```

它的平均每秒上下文切换次数如下所示：

```
RunQueue                 2      Load  Average
ContextSwitch      15919.8    1 mins     0.60
Forks                  0.0    5 mins     0.26
Interrupts           258.9   15 mins     0.13
```

我们换成 worker 模型，再来一次同样的压力测试，结果如下所示：

```
Concurrency Level:      100
Time taken for tests:   14.522 seconds
Complete requests:      100000
Failed requests:        0
Write errors:           0
Total transferred:      40306448 bytes
HTML transferred:       15102416 bytes
Requests per second:    6886.08 [#/sec] (mean)
Time per request:       14.522 [ms] (mean)
Time per request:       0.145 [ms] (mean, across all concurrent requests)
Transfer rate:          2710.48 [Kbytes/sec] received
```

吞吐率只增加了 1000 左右。整个测试过程中，Apache 子进程只有 4 个，我们看看上下文切换次数，如下所示：

```
RunQueue                2      Load     Average
ContextSwitch       43237.6    1 mins     0.72
Forks                   0.0    5 mins     0.20
Interrupts            263.5   15 mins     0.11
```

通过 Nmon 的监视，我们发现在同样条件的压力测试下，worker 的上下文切换次数几乎是 prefork 的两倍以上。

我们在 worker 模型中的 MPM 配置如下：

```
StartServers            2
MaxClients              256
MinSpareThreads         25
MaxSpareThreads         75
ThreadsPerChild         64
MaxRequestsPerChild     0
```

MaxClients 在这里代表了线程最大数，为 256；ThreadsPreChild 代表每个进程允许的多大线程数，为 64。所以，Apache 最多使用 4 个进程。而在 prefork 模型中，我们的配置为最多 100 个进程，可见，两者上下文切换次数的差距原因在此。

在实际应用中，worker 模型的处境比较尴尬，人们几乎很少使用它，因为它的优点并不明显，一旦人们意识到 Apache 无法满足需要时，便会马上使用其他的轻量级 Web 服务器。

一个进程处理多个连接，非阻塞 I/O

一个进程处理多个连接，存在一个潜在条件，就是多路 I/O 就绪通知的应用，在前面的 I/O 模型中我们介绍了常见的几种多路 I/O 就绪通知方法，而在这种并发策略下，多路 I/O 就绪通知的性能成为关键，下面我们会将它们应用在并发模型中并进行深入探讨。

通常我们将处理多个连接的进程称为 worker 进程，或者服务进程，有些使用这种并发模型的 Web 服务器支持 worker 进程数量的配置，比如在 Nginx 中可以进行配置，如下所示：

```
worker_processes 2;
```

这样使得 Nginx 开启两个 worker 进程。

```
5251 ?  Ss 0:00 nginx: master process /usr/local/nginx/sbin/ nginx
5252 ?  S  0:37  \_ nginx: worker process
5253 ?  S  0:05  \_ nginx: worker process
```

Lighttpd 也可以配置 worker 进程的数量，如下所示：

```
server.max-worker = 2
```

默认的配置文件中没有这个参数，其默认值为 0，但并不是说没有 worker 进程，而是由主进程来进行 worker 进程的工作。当它设置为非 0 数值（如 2）后，则使得 lighttpd 主进程创建两个 worker 进程，而主进程则不进行连接处理。

在这种模型中，有时候还会设计独立的 listener 进程，专门负责接收新的连接，然后分配任务给各个 worker，这样做的好处是可以根据各个 worker 的负载来平衡调度任务，但是任务调度有一定的开销，所以在我们常见的模型中一般都是由 worker 进程来进行接收，同样也存在多种方式，这里就不具体介绍。另外，对于超时连接的管理，理论上也可以由独立的进程来处理，但是为了减少进程切换，一般也在 worker 中完成。

了解了这些内容后，接下来我们重点探讨 worker 进程，值得一提的是，有些时候 Web 服务器会使用 worker 线程来代替 worker 进程，但是这种线程实际上通常都是内核级线程，它们在处理多路 I/O 的方式上基本相同，所以可以看成是 worker 进程的一个迷你版。

在 Lighttpd 的配置文件中，可以设置使用哪种多路 I/O 就绪通知方法，比如：

```
server.event-handler = "linux-sysepoll"
```

以上的设置便是使用 epoll，除此之外，Lighttpd 有关性能的文档中列出了以下的推荐：

```
============ ========== ===============
OS            Method      Config Value
============ ========== ===============
all          select     select
Unix         poll       poll
Linux 2.4+   rt-signals  linux-rtsig
Linux 2.6+   epoll       linux-sysepoll
Solaris      /dev/poll   solaris-devpoll
FreeBSD, ... kqueue      freebsd-kqueue
============ ========== ===============
```

下面我们分别使用 select、poll、epoll，通过 strace 来跟踪 lighttpd 主进程从检查数据就绪

到接收一个请求这一过程中的系统调用。这里我们使用了 lighttpd 的默认 worker 数进行配置，由主进程来负责 worker 的工作。

首先使用 select，如下所示：

```
    s-colin:~ # strace -p 20083
    Process 20083 attached - interrupt to quit
    time(NULL)                              = 1234856441
    select(4, [3], [], [3], {1, 0})          = 0 (Timeout)
    time(NULL)                              = 1234856442
    select(4, [3], [], [3], {1, 0})          = 1 (in [3], left {0, 164000})
    accept(3, {sa_family=AF_INET, sin_port=htons(46160), sin_addr=inet_
addr("127.0.0.1")}, [16]) = 6
```

接下来使用 poll，如下所示：

```
    s-colin:~ # strace -p 20115
    Process 20115 attached - interrupt to quit
    time(NULL)                              = 1234856705
    poll([{fd=3, events=POLLIN}], 1, 1000)  = 0
    time(NULL)                              = 1234856706
    poll([{fd=3, events=POLLIN, revents=POLLIN}], 1, 1000) = 1
    accept(3, {sa_family=AF_INET, sin_port=htons(35332), sin_addr=inet_
addr("127.0.0.1")}, [16]) = 6
```

然后是 epoll，如下所示：

```
    s-colin:~ # strace -p 20277
    Process 20277 attached - interrupt to quit
    time(NULL)                              = 1234856802
    epoll_wait(6, {}, 2049, 1000)            = 0
    time(NULL)                              = 1234856803
    epoll_wait(6, {}, 2049, 1000)            = 0
    time(NULL)                              = 1234856804
    epoll_wait(6, {{EPOLLIN, {u32=3, u64=3}}}, 2049, 1000) = 1
    accept(3, {sa_family=AF_INET, sin_port=htons(35333), sin_addr=inet_
addr("127.0.0.1")}, [16]) = 7
```

可以看到，虽然我们使用了不同的方法，但是从内核的角度来看，它们非常相似，都是通过一个系统调用来获得就绪的 socket，这里 lighttpd 对它们都设置了 1 秒的超时时间，并不断地轮询调用，当获得就绪的 socket 后，接着 lighttpd 使用 accept()来接收请求。

但是不要被表面现象所迷惑，它们的区别恰恰就在于其内部实现。让我们用 ab 来对三种方式进行不同并发用户数的压力测试，并分别统计吞吐率。因为这次最多要使用到 20000 个并发用户，所以在测试之前，有一个准备工作要做，我们知道 Linux 的进程最大文件描述符限制为 1024，所以必须修改它，否则我们无法用 ab 同时发起如此大量的请求。

确认你是 root 身份，然后在 Linux 命令行中输入以下命令：

```
    s-colin:~ # ulimit -n 30000
```

这样便可以将当前用户环境的最大文件描述符数量限制改为 30000，注意这个值要比我们将要使用的并发用户数大，因为其他应用程序也要使用文件描述符。

我们使用 151 字节的静态文件作为请求资源，同时采用长连接，因为我们希望尽量减少 socket 创建和关闭所花费的时间对整体性能的影响。测试结果如表 3-3 所示。

表 3-3　采用 select、poll、epoll 对 151 字节文件的压力测试结果对比

并发用户数	select	poll	epoll
100	15356.33	15720.25	15508.37
500	14703.03	15017.62	14131.34
1000	14675.02	14362.11	13864.53
5000	14404.2	14188.41	13627.96
10000	14260.29	14200.85	13018.01
20000	11673.8	11801.4	11577.26

也许以上数据会让你大吃一惊，然后极度失望，epoll 不但没有表现出优势，反而略低于 select 和 poll，这是为什么呢？而且 select 不是只支持 1024 个文件描述符吗？怎么会在 20000 并发用户数时表现得如此优秀。

select 在 20000 并发用户数下的卓越表现的确是事实，但是不要忘了前提条件，我们请求的是 151 字节的静态文件，每次请求的处理时间非常短，几乎在 0.1ms 左右，这比一个进程的时间片都要小很多，所以可以快速处理完一个连接后处理其他连接，再加上 Lighttpd 对于连接队列的良好控制，实际上，lighttpd 这时候并没有打开 20000 个并发连接，通过 lighttpd 的 server-status 监视，我们发现它的连接数基本保持在 300 以内，便足够应付这些请求了。

为了让每次请求的处理时间适当增加，我们换一个 15KB 的静态文件来测试 select，结果如表 3-4 所示。

表 3-4　采用 select 对 15KB 文件的压力测试结果

并发用户数	lighttpd(select)
100	14909.77
500	14680.88
1000	14049.78
5000	13952.75
10000	13633.67
20000	--

提示：

在 20000 并发用户数时，“--”表示已经达到 select 的并发连接数上限，所以 Lighttpd 拒绝服务，ab 无法完成测试。

我们再换一个更大一些的文件，用 224KB 的静态文件来测试 select，结果如表 3-5 所示。

表 3-5　采用 select 对 224KB 文件的压力测试结果

并发用户数	lighttpd(select)
100	2591.4
500	2252.98
1000	--
5000	--
10000	--
20000	--

224KB 的静态文件在 Web 站点很常见，很显然，相比于 151 字节的静态文件，它的处理时间要长得多，所以每个请求会占用更长的连接时间，从测试结果来看，在超过 1000 个并发用户时，select 便无法提供服务。

但是，另一个问题是，用 151 字节的静态文件进行压力测试时，为什么 epoll 的性能和 select/poll 没什么区别呢？如果你被这个问题困扰，那么请不要犹豫，马上看看前面关于 epoll 优势的介绍。事实上，我们的 Web 服务器通常维护着大量的空闲连接，它们有些可能是由于使用长连接而在等待超时，有些可能是网络传输的延时，还有的甚至是黑客故意制造的死连接。这时候，epoll 只关注活跃连接，而不在死连接上浪费时间，但是 select 和 poll 会扫描所有文件描述符，比如 select 在一次系统调用时传入包括了所有 socket 连接的数组，如下所示：

```
    select(206, [3 54 56 58 60 70 72 74 76 78 80 82 84 86 88 90 92 94 96
98 100 102 104 106 108 110 112 114 116 118 120 122 124 126 128 130 132 134
136 138 140 142 144 146 148 150 152 154 156 158 159 160 161 162 163 164 165
166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184
185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201], [],
[3 54 56 58 60 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100 102 104 106
108 110 112 114 116 118 120 122 124 126 128 130 132 134 136 138 140 142 144
146 148 150 152 154 156 158 159 160 161 162 163 164 165 166 167 168 169 170
171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189
190 191 192 193 194 195 196 197 198 199 200 201], {1, 0}) = 1 (in [70], left
{0, 880000}) <0.121198>
```

这个系统调用花费了 0.121198s，这在 Web 服务器上已经算是非常昂贵的开销了。

为了模拟大量空闲连接以便进行测试，我写了以下的 Perl 脚本，保存为 client.pl。它可以和 Web 服务器监听的端口建立连接，然后每 10 秒钟向服务器发送一个简单的字符串，这样做是为了防止服务器将空闲连接主动关闭。

```
#!/usr/bin/perl
use strict;
use Socket;
use IO::Handle;
use threads;
sub connect
{
    my $host = "127.0.0.1";
    my $port = 8001;
    my $protocol = getprotobyname("TCP");
    $host = inet_aton($host);
    socket(SOCK, AF_INET, SOCK_STREAM, $protocol) or die "socket()
failed: $!";
    my $dest_addr = sockaddr_in($port, $host);
    connect(SOCK, $dest_addr) or die "connect() failed: $!";
    SOCK->autoflush(1);
    while (1)
    {
        print SOCK "hi!";
        sleep(10);
    }
    close SOCK;
}
my $child = 0;
for (my $i = 0; $i < 100; ++$i)
{
    $child = threads->new(\&connect);
}
$child->join;
```

由于一个 Perl 进程开启太多的线程会导致错误，所以我们让一个 Perl 进程启动 100 个线程，同时用 Shell 脚本来启动 50 个 Perl 进程，这样一共可以制造 5000 个空闲连接。Shell 脚本如下所示：

```
#!/bin/bash
for ((i = 0; i < 50; ++i))
do
        echo "client start"
        perl client.pl &
done
```

一切就绪，我们将 Lighttpd 切换到 epoll 模型，启动这些 Perl 进程，通过 Lighttpd 的 server-stauts，可以看到 lighttpd 已经建立了 5000 多个连接，且都处于等待接收请求数据的状态。server-status 显示如下：

```
legend
. = connect, C = close, E = hard error
r = read, R = read-POST, W = write, h = handle-request
q = request-start,  Q = request-end
```

```
s = response-start, S = response-end
5005 connections
rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
.....
```

太好了！接下来我们对 Lighttpd 进行压力测试，100 个并发用户，请求 151 字节的静态文件，采用长连接，一共 100000 个请求，测试结果如下所示：

```
Server Software:        lighttpd/1.4.20
Server Hostname:        localhost
Server Port:            8001
Document Path:          /test.htm
Document Length:        151 bytes
Concurrency Level:      100
Time taken for tests:   6.176 seconds
Complete requests:      100000
Failed requests:        0
Write errors:           0
Keep-Alive requests:    99265
Total transferred:      38796325 bytes
HTML transferred:       15100000 bytes
Requests per second:    16192.87 [#/sec] (mean)
Time per request:       6.176 [ms] (mean)
Time per request:       0.062 [ms] (mean, across all concurrent requests)
Transfer rate:          6135.00 [Kbytes/sec] received
```

表现非常不错，吞吐率为 16192.87reqs/s，甚至比之前没有 5000 个空闲连接的测试结果还要略高一些，这说明 5000 个空闲连接几乎没有对 epoll 产生干扰。同时，我们用 strace 来统计各系统调用的时间消耗，如下所示：

% time	seconds	usecs/call	calls	errors	syscall
26.71	**0.024458**	**44**	**559**		**epoll_wait**
22.85	0.020926	0	200190		setsockopt
12.34	0.011301	0	200217	200190	stat64
11.61	0.010627	0	100104		open
4.55	0.004163	0	129504		ioctl
4.22	0.003860	0	129504	284	read
4.01	0.003670	0	100095		sendfile64
3.69	0.003381	0	100095		writev
3.08	0.002824	0	79598		epoll_ctl
3.07	0.002808	0	101847		fcntl64
2.61	0.002387	0	100980		close
1.05	0.000962	0	2709		write
0.22	0.000201	0	877	1	accept
0.00	0.000000	0	560		time
0.00	0.000000	0	39		brk
0.00	0.000000	0	105	100	shutdown
100.00	0.091568		1246983	200575	total

稍后我们将以上统计结果和其他模型对比分析。接下来我们将 Lighttpd 配置为 poll 方式后，同样开启 5000 个空闲连接，在同样的压力条件下，测试结果如下所示：

```
Server Software:        lighttpd/1.4.20
Server Hostname:        localhost
Server Port:            8001
Document Path:          /test.htm
Document Length:        151 bytes
Concurrency Level:      100
Time taken for tests:   9.311 seconds
Complete requests:      100000
Failed requests:        0
Write errors:           0
Keep-Alive requests:    99263
Total transferred:      38796315 bytes
HTML transferred:       15100000 bytes
Requests per second:    10740.34 [#/sec] (mean)
Time per request:       9.311 [ms] (mean)
Time per request:       0.093 [ms] (mean, across all concurrent requests)
Transfer rate:          4069.20 [Kbytes/sec] received
```

看，poll 果然经不起 5000 个空闲连接的折磨，此时吞吐率已经降到了 10740.34reqs/s，比之前没有 5000 个空闲连接时的结果 15720.25reqs/s 少了 1/3。我们也通过 strace 获得系统调用时间统计结果，如下所示：

```
% time     seconds  usecs/call     calls    errors     syscall
------ ----------- ----------- --------- --------- ----------------
 89.97    0.804475        3106       259           poll
  3.05    0.027234           0    200196           setsockopt
  1.94    0.017339           0    200223    200196 stat64
  1.28    0.011403           0    100107           open
  1.02    0.009128           0    120462       195 read
  0.69    0.006161           0    100098           writev
  0.67    0.005978           0    100098           sendfile64
  0.49    0.004403           0    120462           ioctl
  0.41    0.003625           0    100982           close
  0.38    0.003392           0    101848           fcntl64
  0.12    0.001063           0      2710           write
  0.00    0.000000           0       260           time
  0.00    0.000000           0        26           brk
  0.00    0.000000           0       876         1 accept
  0.00    0.000000           0       102       100 shutdown
------ ----------- ----------- --------- --------- ----------------
100.00    0.894201              1148709    200492 total
```

我想你已经看到了，采用 epoll 和 poll 两种方式的系统调用统计中，poll()的总时间要比 epoll_wait()多出几乎 33 倍，占所有系统调用总时间的比例也大大超过 epoll_wait()，这就是 epoll 的优势所在。

而对于 Lighttpd 的 select 方式，很不幸，由于 1024 个连接数的限制，5000 个空闲连接根本无法建立，所以放弃测试。

可见，当存在大量非活跃连接的时候，epoll 的优势不言而喻。一个典型的应用是图片服务器，它们希望为用户提供网页中大量图片的快速下载，所以采用长连接方式，但是这些大量的连接在等待超时关闭前，便处于空闲状态，在这种情况下，epoll 依然能够很好地工作。但在其他一些场景下，仔细想想 epoll 在你的站点中真的发挥优势了吗？

另外，关于 worker 进程的数量，既然可以由我们来设置，那么是不是越多越好呢？显然不是，别忘了在任何时刻，从 CPU 的角度来看，只有一个进程在运行。对于不同的动态内容和静态文件，实际要考虑的因素非常多，比如在前面介绍 IOWait 时提到的例子中，我们通过将 Nginx 的进程数从 128 降到 8，使得网络 I/O 流量大幅度提升，但这时候的 HTTP 并发连接数并不高，我们通过 Nginx 的 stub_status_module 可以看到统计结果如下：

```
Active connections: 2624
server accepts handled requests
 88932 88932 135579
Reading: 327 Writing: 2142 Waiting: 155
```

可以看到，当前活跃的连接有 2624 个，其中有 2142 个连接正在发送响应数据。而在 Nginx 进程数为 128 时，虽然网络 I/O 流量降低，但是并发连接数比较高，如下所示：

```
Active connections: 7917
server accepts handled requests
 40158 40158 43871
Reading: 416 Writing: 7272 Waiting: 229
```

显然，大量的 worker 进程可以维持更多的活跃连接数，但此时每个连接的下载速度要远远小于前者，那么，如何决定 worker 进程数完全取决于你的倾向，即你希望为更多的用户同时提供慢速下载服务，还是希望为有限的用户提供快速的下载服务。

对于动态内容，比如 PHP 脚本，worker 进程通常只是负责转发请求给独立的 fastcgi 进程，或者作为反向代理服务器将请求转发给后端服务器，这时候，worker 进程并不依赖太多的本地资源，所以为了提高并发连接数，我们可以适当地提高 worker 进程数，但在一般情况下，动态内容本身的吞吐率是相当有限的，由于存在脚本解释器的开销，通常 2000reqs/s 的吞吐率就已经很高了，所以 worker 进程的压力并不是很大，但如果作为基于反向代理的负载均衡调度器时，多台后端服务器扩展了动态内容计算能力，worker 进程数会逐渐成为整体性能的瓶颈。当然，太多的 worker 进程又会带来更多的上下文切换开销和内存开销，从而整体上使所有连接的响应时间变长，所以，没有一个绝对的公式告诉你如何选择 worker 进程数，而你要做的是根据站点的实际情况来进行分析和调整。

一个线程处理多个连接，异步 I/O

即便是拥有高性能的多路 I/O 就绪通知方法，但是磁盘 I/O 操作的等待还是无法完全避免，当我们对磁盘文件调用 read()或者通过 sendfile()直接发送数据时，设置文件描述符为非阻塞没有任何意义，如果需要读取的数据不在磁盘缓冲区，磁盘便开始动用物理设备来读取数据，这个时候整个进程的其他工作都必须等待。

更加高效的方法便是对磁盘文件操作使用异步 I/O，在前面比较详细地介绍过异步 I/O 及其实现，但在实际上，目前很少有 Web 服务器支持这种真正意义上的异步 I/O，理论上在某种特定的场景中它的性能要比 sendfile 更加出色，但是对于大量小文件的并发请求，文件传送可能不是关键，而多路 I/O 就绪通知方法的性能更加重要。目前正在开发的 Lighttpd 1.5 试图使用 Linux 的 AIO 实现，一旦发布后，我们可以尝试一下。

第 4 章

动态内容缓存

还记得上一章的内容吗？也许你还记忆犹新，很好，也许你已经眼花缭乱，没关系，因为从现在开始，那些都已经不重要了。

我们的目光曾经聚焦在 Web 服务器本身，改进 I/O 模型和并发策略带来的性能提升让我们激动不已，然而，我们的生活变幻万千，绝不像静态文件那样如此单调，Web 站点更多的是提供动态内容，比如动态网页、动态图片、Web 服务等，它们通常在 Web 服务器端进行计算，生成 HTML，并返回给用户。与此同时，它们在生成 HTML 的过程中，不可避免涉及了更多的 CPU 计算和 I/O 操作，这已经超出了 Web 服务器本身，比如访问数据库涉及数据库服务器的 CPU 计算和磁盘 I/O 操作，以及与数据库服务器通信的网络 I/O 操作，同样的情况也发生在调用 Web API，比如通过 twitter API 获取资料。

不幸的是，通常这些操作都不是异步的，我们必须等待。

4.1 重复的开销

我找来一个 PHP 编写的动态网页，它的名字叫 place_posts.php，我想你可能也比较喜欢 PHP，不过即使你习惯用其他语言，也毫不影响我们的探讨。这个动态网页的工作很简单，它多次和数据库打交道，获取一系列它感兴趣的数据。这里我们就不列出这个动态网页的 PHP 代码，因为这不是我们关注的重点。

执行一次这个动态网页需要花多少时间呢？我们用 ab 来请求这个动态网页，为了避免并发处理时重叠利用资源，我们这里将并发用户数和总请求数都设置为 1，结果如下所示：

```
Document Path:          /place_posts.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      1
Time taken for tests:   0.028 seconds
Complete requests:      1
Failed requests:        0
Write errors:           0
Total transferred:      13181 bytes
HTML transferred:       13000 bytes
Requests per second:    35.24 [#/sec] (mean)
Time per request:       28.378 [ms] (mean)
Time per request:       28.378 [ms] (mean, across all concurrent requests)
Transfer rate:          453.59 [Kbytes/sec] received
```

可以看到，响应时间为 28.378ms，吞吐率只有 35.24reqs/s，这也许是到目前为止我们看到的最差劲的测试结果了。你也许会说适当提高并发用户数可以带来更高的吞吐率，但是对于如此漫长的响应时间，我想吞吐率也高不到哪去，即便是提高，也只是提高服务器的利用率，然而从用户角度体验到的响应时间却没有任何减少，反而会增加。

幸运的是，我打开了数据访问 DEBUG 日志，我们可以看到这个动态网页在执行时包含了以下的数据访问，每条语句的末尾都有执行时间的记录。当然，这些 SQL 查询的含义你不必理会，权且把它们当作一系列数据查询即可。

```
    03.03.2009 00:24:56 [DEBUG] connect to mysql[localhost] ok [7.3978ms]
    03.03.2009 00:24:56 [DEBUG] switch to db[db_map_main] ok [0.0569ms]
    03.03.2009 00:24:56 [DEBUG] db query ok: select * from marker_info where
marker_id=12882 [0.3180ms]
    03.03.2009 00:24:56 [DEBUG] db query ok: select * from position_info where
pos_id=53 [0.1688ms]
    03.03.2009 00:24:56 [DEBUG] db query ok: select * from position_info where
pos_id=1 [0.1449ms]
    03.03.2009 00:24:56 [DEBUG] db query ok: update marker_info set
marker_views=marker_views+1 where marker_id=12882 [0.1912ms]
    03.03.2009 00:24:56 [DEBUG] db query ok: select * from marker_info where
marker_id<>12882 and marker_lat<41.899859979 and marker_lat>37.899859979
and marker_lng<118.487178802 and marker_lng>114.487178802 order by
abs(marker_lat-39.899859979)+abs(marker_lng-116.487178802) limit 10
[9.4790ms]
    03.03.2009 00:24:56 [DEBUG] db query ok: select user_id,user_nick,
user_email,user_lat,user_lng from user_info where user_lat<41.899859979
and user_lat>37.899859979 and user_lng<118.487178802 and user_lng>114.
487178802 order by abs(user_lat-39.899859979)+abs(user_lng-116.487178802)
limit 12 [7.3280ms]
    03.03.2009 00:24:56 [DEBUG] db query ok: select ti.tag_id as
tag_id,ti.tag_name as tag_name from marker_tag_rel tr,marker_tag_info ti
where tr.tag_id=ti.tag_id and tr.marker_id=12882 [0.2419ms]
```

请注意，我将执行时间超过 1ms 的数据访问用粗体做了标记，包括一次数据库连接操作和两次比较复杂的 SQL 查询，天啊，它们都在 7ms 以上，你也许觉得我在大惊小怪，这没什么了不起的，好吧，我想你可能还没有意识到 7ms 意味着什么，还记得我们之前的网络数据响应时间计算吗？我们由计算得知，一个比特通过光纤从北京传到西安，理论上只需要 5ms。还有，我们之前对 Web 服务器上一个 151 字节的静态文件的请求响应时间大概有 0.5ms 左右，当然，这是我的测试结果，你的服务器也许可以更快。

你可以亲手算一算，以上所有的数据库访问时间总和为 25.3265ms，这个时间足可以让比特传播接近地球赤道半径的距离。

可见，在 28.378ms 的响应时间中，足足有 25.3265ms 的数据库访问时间，真是不可思议，时间都花到这里了，现实是残酷的，我们必须要获取这些内容提供给用户，怎样可以让这些数据的访问更快呢？

在这里，我们暂且跳过这个问题，后面我们在有关数据库 I/O 性能的探讨中会再次遇到它。而此刻，我们希望通过另一个途径来解决，我们发现大多数时候，动态网页在多次请求时的计算结果几乎没有变化，那为什么要重复计算呢？通过缓存，我们甚至可以完全砍掉这 25.3265ms。

4.2 缓存与速度

说到缓存，事实上它将一直贯穿随后几章的内容，而在这里，我们所指的是由动态内容自行实现的缓存机制，这其中包括整页缓存、局部缓存、数据缓存等，除此之外，还有代码解释器缓存、Web 服务器缓存等，这些在后续章节中都会一一介绍。

缓存（cache）的思想由来已久，简单地说，缓存的目的就是把需要花费昂贵开销的计算结果保存起来，在以后需要的时候直接取出，而避免重复计算，一切缓存的本质都是如此。计算机乃至互联网，缓存的应用数不胜数，我们都知道 CPU 缓存，它是位于 CPU 和内存之间的临时存储器，它的容量不大，但是交换速度要高于内存，CPU 将频繁交换的数据放在缓存中，如果以后需要则直接读取缓存，从而避免访问速度较慢的内存，不可否认，尽管我们认为内存速度已经很快，但是在 CPU 缓存面前，它还是力不从心。

顺便提一下另一个看起来相似的概念——缓冲（buffer）。缓冲的原意出自物理学，那就是减缓冲击力，在计算机应用场景中，我们使用它的引申含义，其目的在于改善各部件之间由于速度不同而引发的问题。比如将用户态地址空间的数据写入磁盘时，显然内存的速度比磁盘速度要快得多，所以人们设计了磁盘缓冲区，让数据源源不断地流进缓冲区，再由缓冲区负责写入磁盘，这样内存便可以不必随着磁盘的慢节奏来工作，所以磁盘缓冲区起到了将快速设备和慢速设备平滑衔接的作用，另外我们在线观看视频的时候，视频缓冲区的意义也是如此。

缓冲和缓存有一些相似之处，比如它们都需要一块存储区，而且它们的本质都与速度不一致有关，即便是缓存，如果计算速度和读取缓存的速度差不多，那么它也毫无意义。但是，缓存更加注重的是策略，也就是说缓存命中率，如果每次都能在缓存中找到需要的数据，那是最理想的结果，如果每次都在缓存中找不到需要的数据，那么缓存将变得毫无价值，并且还由于缓存的管理逻辑增加了新的开销。所以凡是使用缓存，都一定要意识到命中率的重要性。

4.3 页面缓存

同样，按照缓存的动机，我们将动态内容的计算结果进行缓存，并在随后需要的时候直接取出返回给用户。对于动态网页来说，缓存的内容实际上就是动态网页输出的 HTML，我们称为页面缓存（Page Cache），而对于其他动态内容，比如动态图片或动态 XML 数据，

我们也可以将它们的输出结果整体进行缓存，其实现机制和动态网页是一样的。

对于动态内容缓存的具体方法，你可能并不陌生，也许你在使用类似 Smarty 的模板框架，或者使用 Cakephp、Django、Zend 等 MVC 框架，不管它们支持多么丰富的特性，有一点是相同的，就是将视图（View）和控制器（Controller）进行分离，这样便可以让控制器自身拥有缓存控制权，从而提供丰富灵活的缓存控制方法。

在几年前，ASP.NET 的视图分离最早让我感到振奋，它提出了服务器端控件的概念，其实就是页面组件的视图和控制器的分离，遗憾的是它的封装过于复杂，灵活性不够。

除此之外，Smarty 还提供了丰富的扩展接口，你可以完全重写缓存策略来匹配你的站点需要，达到理想的性能，下面我们主要从性能的角度来探讨缓存实现的本质和一些应用。

Smarty 缓存

对于前面的动态网页 place_posts.php，我使用了 Smarty 模板框架，为了对比应用缓存后的性能提升，我们先对动态网页进行一次未使用任何缓存情况下的压力测试，结果如下所示：

```
Document Path:          /place_posts.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   19.382 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      13160373 bytes
HTML transferred:       13000000 bytes
Requests per second:    51.59 [#/sec] (mean)
Time per request:       1938.222 [ms] (mean)
Time per request:       19.382 [ms] (mean, across all concurrent requests)
Transfer rate:          663.08 [Kbytes/sec] received
```

接下来，我们使用 Smarty 的默认缓存方法，并且打开缓存开关，值得一提的是，通常你还需要在程序中适当的位置根据判断来决定是否跳过数据访问或直接终止程序，因为缓存机制并不知道你在输出缓存之前还想做点什么。另外，这里的动态网页使用了一个自行开发的小型框架，它将每个动态网页封装为一个 PHP 对象，后面看到的$this 都代表这个对象的引用。下面是一些代码片段，其中的 do_some_db_query()方法代表所有数据库的访问操作。

```
require '../libsmarty/Smarty.class.php';
$this->smarty = new Smarty();
$this->smarty->caching = true;
$this->template_page = "place_posts.htm";
$this->cache_id = $this->marker_id;
if ($this->smarty->is_cached($this->template_page, $this->cache_id))
{
```

```
    $this->smarty->display($this->template_page, $this->cache_id);
    exit(0);
}
do_some_db_query();
$this->smarty->display($this->template_page, $this->cache_id);
```

以上的 is_cached 函数非常重要，如果缓存存在，它将直接读取缓存并显示内容，然后终止程序。我们再来看看压力测试的结果：

```
Document Path:          /place_posts.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   5.749 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      13646288 bytes
HTML transferred:       13495000 bytes
Requests per second:    173.95 [#/sec] (mean)
Time per request:       574.894 [ms] (mean)
Time per request:       5.749 [ms] (mean, across all concurrent requests)
Transfer rate:          2318.07 [Kbytes/sec] received
```

可见，在应用缓存后，吞吐率明显增加到原来的 3 倍以上，这早已在我们的意料之中。我们还可以通过 strace 来看看节省的时间都有哪些，我们跟踪了 lighttpd 下的一个 fastcgi 进程，在未使用缓存的时候，结果如下：

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 92.52    0.005282           1      7350           read
  1.84    0.000105           2        70           connect
  1.56    0.000089           0       420           munmap
  1.35    0.000077           0      2730           open
  0.58    0.000033           0      2380           access
  0.51    0.000029           0      1540           write
  0.51    0.000029           0      2870           close
  0.44    0.000025           0      2940           stat64
  0.32    0.000018           0      8120           time
  0.21    0.000012           0       280           rt_sigaction
```

使用了缓存后，结果如下：

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 41.35    0.000043           0       196           munmap
 25.96    0.000027           0       392           access
 15.38    0.000016           0        49           chdir
  9.62    0.000010           0       392           getcwd
  7.69    0.000008           0       735           time
  0.00    0.000000           0      2156           read
  0.00    0.000000           0       147           write
  0.00    0.000000           0       490           open
  0.00    0.000000           0       539           close
  0.00    0.000000           0        49           lseek
```

从以上的数据中，可以明显地看出，read()调用在使用缓存后大量减少，而未使用缓存时read()调用的总时间达到 92.52%之多。那么减少的这部分 read()是什么呢？主要包括以下两部分：

- 与数据库服务器通信时接收数据。前面 SQL 日志中的那些访问语句，大多数的响应数据要比请求数据长很多，这花费了大部分的 read()时间。
- 输出缓存后终止程序，停止对后续一系列文件的读取，包括 PHP 文件、页面模板文件等。

缓存持久化与查找

通常，我们将动态内容的缓存存储在磁盘上，通常磁盘有足够而且廉价的空间来存储大量的文件，我们不用担心因为空间的限制而淘汰缓存，这是一种比较简单也容易部署的方法。前面我们对 place_posts.php 的缓存便存储在指定的磁盘目录中，因为我们使用了 Smarty 模板框架，所以可以在 Smarty 的对象中随便指定缓存目标目录，我们指定了一个名为 cache 的目录，进入目录，我们看到了刚才的缓存文件：

```
cache/12882^%%E6^E6C^E6C210ED%%place_posts.htm
```

我们知道，一个动态网页根据 URL 参数的不同，会展现出多种不同的结果，而每一种结果都必须生成对应的缓存文件。我们看到以上的缓存文件名中包含了一个数字——12882，还记得吗？我们在请求这个动态网页的时候，URL 参数为“?marker_id=12882”，同时，我们在程序中使用了 Smarty 提供的缓存分类支持，这样一来，Smarty 根据不同参数生成了不同的缓存文件，它完成得非常出色，我们一点都没有觉察到。

可是，你有没有想过，如果缓存文件非常多的话，cache 目录下会拥挤大量的文件。首先，这不是没有可能，即便你的站点规模不大，也有可能在不长的时间里毫不费力地积累数万个缓存文件，并且糟糕的是这些缓存文件不会自己删除。一旦 cache 目录下的文件数量达到数以万计后，CPU 花在遍历目录上的时间便非同寻常，如果缓存文件的读写比较频繁的话，可能 CPU 使用率很容易达到 100%。你可以使用支持目录 hash 等加速目录遍历的文件系统来缓解该情况，如 XFS 和 reiserfs，可是如果你的站点已经在提供服务，则重新安装文件系统显然不太现实。

幸运的是，我们可以通过缓存目录分级来解决这一问题，Smarty 对它提供了支持，而对于其他框架而言，即便没有支持，修改程序使其按照你的旨意来分级存储，也不是一件困难的事情。

在 Smarty 中，通过设置选项让缓存支持目录分级非常容易，而这个选项默认是关闭的。

```
$this->smarty->use_sub_dirs = true;
```

现在，我们在开启缓存的情况下再次访问这个动态网页，可以看到在 cache 中生成了如下的目录层次和缓存文件。

```
cache/12882/%%E6/E6C/E6C210ED%%place_posts.htm
```

这样一来，可以将每个目录下的子目录或文件的数量控制在少量范围内，同时适当增加目录切换的次数，但这部分开销微不足道。

过期检查

到目前为止，我们给动态内容引入了缓存，它在一定程度上避免了动态内容不必要的重复计算，缩短了请求响应时间并提高了服务器吞吐率。但是，动态内容的目的在于提供变化的内容，所以它的缓存不可能长期有效，否则就失去了动态内容的意义，所以动态内容的缓存机制必须能够判断缓存何时过期，以及何时需要生成新的缓存。

为每个缓存标记过期时间，然后动态内容每次对缓存进行过期检查，这是一种常见的缓存过期检查策略，实现方法很多。在 Smarty 中，缓存过期时间标记在什么地方呢？

我们来对这个动态网页指定缓存有效期长度，以秒为单位，比如我们这里设置了 1 个小时。

```
$this->smarty->cache_lifetime = 60 * 60;
```

我们再回过头来看刚才的缓存文件 E6C210ED%%place_posts.htm，以下是文件开头的一段内容：

```
338
a:5:{s:8:"template";a:6:{s:15:"place_posts.htm";b:1;s:15:"inc_top_1.
2.htm";b:1;s:18:"inc_header_1.2.htm";b:1;s:17:"inc_left_menu.htm";b:1;s:
20:"inc_place_header.htm";b:1;s:14:"inc_footer.htm";b:1;}s:6:"config";a:
1:{s:26:"../configs/view/define.htm";b:1;}s:9:"timestamp";i:1236035506;s
:7:"expires";i:1236039106;s:13:"cache_serials";a:0:{}}<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org
/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

有意思的是，在缓存的 HTML 正文之前，有一长段不知所云的数据，仔细一看，你也许会发现这是一段经过序列化处理后的数据，第一行的 388 代表这段数据的长度，便于读取缓存时提取这部分信息。我们对这 388 个字节的数据调用 PHP 的 *unserialize()*方法进行反序列化处理，可以得到以下的数据结构：

```
array(5)
{
  ["template"]=> array(6)
  {
```

```
    ["place_posts.htm"]=> bool(true)
    ["inc_top_1.2.htm"]=> bool(true)
    ["inc_header_1.2.htm"]=> bool(true)
    ["inc_left_menu.htm"]=> bool(true)
    ["inc_place_header.htm"]=> bool(true)
    ["inc_footer.htm"]=> bool(true)
  }
  ["config"]=> array(1)
  {
    ["../configs/view/define.htm"]=> bool(true)
  }
  ["timestamp"]=> int(1236035506)
  ["expires"]=> int(1236039106)
  ["cache_serials"]=> array(0) {}
}
```

这正是动态内容在创建缓存时留下的信息标记，我们只看粗体部分的 *timestamp* 和 *expires*，它们的值都是 UNIX Timestamp 格式，且都表示一个时间点，前者对应的是缓存文件的创建时间，后者对应的是该缓存过期的时间，为什么要同时记录这两个时间呢？这是因为它们对应着两种过期检查方法：

- 每次检查时，根据缓存的创建时间、缓存有效期设置的长度，以及当前时间，来判断是否过期，也就是说，如果当前时间距离缓存创建时间的间隔超过有效期长度的话，认为缓存过期，这是一种相对比较。
- 每次检查时，根据缓存的过期时间和当前时间来判断是否过期，这个方法比较简单，直接检查当前时间是否超过缓存过期时间即可，这是一种绝对比较。

听起来好像有点晕，尽管很多人能够理解，这里还是要为少数人提醒一下，此处有两个概念不要混淆，缓存过期时间和缓存有效期长度，前者是一个具体的时间点，比如缓存在到达 2009 年 2 月 1 日 12 点 56 分 12 秒时过期；后者是一个时间长度，比如缓存从创建时起，30 分钟后过期，这个长度由我们在程序中自行设置。在 Smarty 中，这两种方法对应的 caching 值分别为 1 和 2。

还是回到刚才的问题，我们关心的是它们到底有什么区别。其实很简单，如果是应用了第二种方法，那么在缓存过期前，如果你修改了程序中的缓存有效期长度，是不会影响上一次缓存的过期时间，而对于第一种方法，修改缓存有效期长度会影响每一次的过期检查，这有利于对缓存过期时间的调试。当然，如果缓存有效期长度不会变化的话，两种方法没有太大区别。所以，根据你的需要，选择其中的一种方法。

Smarty 的这种缓存过期检查方法是否存在一定的开销呢？答案是肯定的，开销是难免的，因为每次请求动态内容时，即使是命中缓存，Smarty 仍然要通过 is_cached()和 display()对缓存文件的内容进行两次分析，每次分析都调用了 smarty_core_read_cache_file 核心函数，包括提取缓存标志信息和 HTML 等，这涉及大量的字符串操作。

```
if ($this->smarty->is_cached($this->template_page, $this->cache_id))
{
    $this->smarty->display($this->template_page, $this->cache_id);
    exit(1);
}
```

如果将缓存过期标记和 HTML 单独存储，是否可以节省这部分字符串操作的时间呢？

是否放弃 Smarty 缓存

我打算放弃 Smarty 提供的缓存方法，自己来写一个简单够用的缓存方法，设计思路是在缓存文件中只存储 HTML，而将文件的修改时间作为过期判断依据，通过 PHP 的 stat()方法可以获得文件的最后一次修改时间。以下是新方法的部分代码，这里贴出代码的目的不是让大家直接使用，而是便于大家清晰地了解这种实现中的一些关键环节，往往代码是最好的文档。

```
$this->key = $this->template_page . $this->cache_id;
$this->path = "../cache/" . $this->key;
if (file_exists($this->path))
{
    $tmp = stat($this->path);
    if (time() - $tmp['mtime'] < $this->smarty->cache_lifetime)
    {
        echo file_get_contents($this->path);
        exit(1);
    }
}
do_some_db_query();
$html = $this->smarty->fetch($this->template_page, $this->cache_id);
file_put_contents($this->path, $html);
echo $html;
```

这次执行动态网页后，我们看到在 cache 目录下生成了以下文件：

```
place_posts.htm12882
```

很好，这正是我们所希望的结果，同样，我们再次进行同样的压力测试，结果如下所示：

```
Document Path:
 /place_posts_file_smarty.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   5.262 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      13150161 bytes
HTML transferred:       13000000 bytes
Requests per second:    190.05 [#/sec] (mean)
Time per request:       526.174 [ms] (mean)
Time per request:       5.262 [ms] (mean, across all concurrent requests)
Transfer rate:          2440.63 [Kbytes/sec] received
```

相比于使用 Smarty 缓存时的 173.95 reqs/s，这次的吞吐率只增加了不到 10%，而且不要忘了，新的缓存方法只实现了最基本的功能，除此之外一无所有，没有错误处理，没有缓存目录分级，代码量也只有短短几行，仅仅代码计算量的减少带来 10%的性能提升，这也不足为怪。

说到这里，我们必须清楚一个原则，在动态网页加载 HTML 缓存并终止程序之前，我们要让它尽可能只加载必要的其他文件，这些文件越少越好，比如引用的某些 PHP 库文件，因为这些 PHP 库文件可能在直接输出缓存的时候根本不需要。往往我们积极地引入缓存，但是却不舍得花一些时间来考虑哪些计算可以在加载缓存前省略，如果你使用缓存的目的是跳过数据库访问，那么请做得彻底一些，在加载缓存之前，将与数据库访问相关的库文件引用关闭掉。事实上，不只是缓存文件的加载存在磁盘 I/O 开销，脚本文件和页面模板也一样，不过幸运的是，脚本加速器可以将脚本代码进行缓存和优化，这在后续章节中会专门介绍。

没错，在输出缓存之前，不要加载没用的东西。等等，这里我似乎就犯了这个错误，在刚才我们使用新的文件缓存方法时，如果缓存存在，程序直接输出缓存并终止程序，这个过程完全由我们自己实现，似乎不需要 Smarty 函数库，我找到了程序中的以下这一段：

```
require '../libsmarty/Smarty.class.php';
$this->smarty = new Smarty();
```

糟糕，它果然还在缓存检查代码之前，我们将它转移到缓存检查代码的后面，再来看看压力测试的结果：

```
Document Path:          /place_posts_file.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   2.169 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      13151886 bytes
HTML transferred:       13000000 bytes
Requests per second:    461.08 [#/sec] (mean)
Time per request:       216.881 [ms] (mean)
Time per request:       2.169 [ms] (mean, across all concurrent requests)
Transfer rate:          5921.98 [Kbytes/sec] received
```

这次一定超乎你的想象，吞吐率竟然会有如此大的提升，这种差异来自于我们减去了加载 Smarty 库文件和初始化 Smarty 对象的时间，前者是磁盘 I/O 开销，后者是 CPU 和内存交换开销，我们用 strace 来跟踪 PHP 处理进程，观察跟磁盘 I/O 相关的几个系统调用。

同样是接受 100 个请求，当缓存检查代码之前包含 Smarty 库文件的时候，结果如下所示：

% time	seconds	usecs/call	calls	errors	syscall
35.46	**0.000100**	**0**	**400**		**munmap**
23.76	**0.000067**	**0**	**4200**		**read**
15.25	**0.000043**	**0**	**900**		**open**
4.26	0.000012	0	300		write
3.55	0.000010	0	1000		close
3.55	0.000010	0	800		getcwd
2.84	0.000008	0	1200		time
2.84	0.000008	0	200		rt_sigaction
2.84	0.000008	0	200		rt_sigprocmask
2.84	0.000008	0	1100		fstat64
2.84	0.000008	0	200		recv
0.00	0.000000	0	100		chdir
0.00	0.000000	0	100		lseek
0.00	0.000000	0	100		access
0.00	0.000000	0	100	100	ioctl
0.00	0.000000	0	300		setitimer
0.00	0.000000	0	100		_llseek
0.00	0.000000	0	100		poll
0.00	0.000000	0	400		mmap2
0.00	0.000000	0	100		stat64
0.00	0.000000	0	52		lstat64
0.00	0.000000	0	**100**		**accept**
0.00	0.000000	0	100		shutdown
100.00	0.000282		12152	100	total

去掉缓存检查代码之前的 Smarty 库文件引用后，结果如下所示：

% time	seconds	usecs/call	calls	errors	syscall
36.96	**0.000034**	**0**	**2800**		**read**
31.52	**0.000029**	**0**	**700**		**open**
20.65	0.000019	0	600		getcwd
10.87	0.000010	0	1000		time
0.00	0.000000	0	300		write
0.00	0.000000	0	800		close
0.00	0.000000	0	100		chdir
0.00	0.000000	0	100		lseek
0.00	0.000000	0	100		access
0.00	0.000000	0	100	100	ioctl
0.00	0.000000	0	300		munmap
0.00	0.000000	0	300		setitimer
0.00	0.000000	0	100		_llseek
0.00	0.000000	0	100		poll
0.00	0.000000	0	200		rt_sigaction
0.00	0.000000	0	200		rt_sigprocmask
0.00	0.000000	0	300		mmap2
0.00	0.000000	0	100		stat64
0.00	0.000000	0	5		lstat64
0.00	0.000000	0	900		fstat64
0.00	0.000000	0	**100**		**accept**
0.00	0.000000	0	200		recv
0.00	0.000000	0	100		shutdown
100.00	0.000092		9505	100	total

我想你已经清楚地看到，在去掉 Smarty 库文件的引用后，read()和 open()的执行次数和执行时间都明显减少，同时文件内存映射的开销也减少了，这些都是吞吐率大幅度提升的原因，当然还有一个主要的原因是 Smarty 实例初始化的时间也被省掉了，这是脚本解释器的开销。

说到这里，顺便提一下，前面说到 Smarty 提供了丰富的扩展接口，一点没错，它也提供了自定义缓存处理函数的支持，如果用这种方式来实现其他缓存处理方法，仍然需要加载 Smarty 库文件并初始化 Smarty 对象，很遗憾，这样的性能可想而知，所以还是打消这个念头吧。

把缓存放到内存中

前面的两种缓存实现都将缓存数据存储在磁盘文件中，每次缓存加载和过期检查都存在磁盘 I/O 的开销，反过来它也受到磁盘负载的影响，如果这个磁盘同时还运行着如数据库这样的磁盘 I/O 密集型应用，那么缓存文件的 I/O 操作便会存在一定的延迟。

我们还可以将动态网页的 HTML 缓存在其他地方，比如本机内存中，借助 PHP 的 APC 模块，我们可以轻松地将任何 PHP 运行时的数据或对象缓存在内存中，这样一来，加载缓存的过程将没有任何磁盘 I/O 操作。注意这里我们只是使用了 APC 的数据缓存方法，并没有使用 APC 的 opcode 缓存功能，opcode 在后续章节中会有专门的探讨。

APC 数据缓存提供了 Key/Value 的存储方式，即使在保存大量 Key 的时候也能保证高效的查找性能，所以我们不用为缓存目录分级的事情操心。同时，每个 Key 都可以设置有效期长度，一旦过期它便会被删除。使用了 APC 后的代码片段如下所示：

```
$this->key = $this->template_page . $this->cache_id;
$html = apc_fetch($this->key);
if ($html !== false)
{
    echo $html;
    exit(1);
}
do_some_db_query();
$html = $this->smarty->fetch($this->template_page, $this->cache_id);
apc_add($this->key, $html, $this->smarty->cache_lifetime);
echo $html;
```

测试结果如下所示：

```
Document Path:          /place_posts_apc.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   2.112 seconds
Complete requests:      1000
Failed requests:        0
```

```
Write errors:           0
Total transferred:      13150460 bytes
HTML transferred:       13000000 bytes
Requests per second:    473.38 [#/sec] (mean)
Time per request:       211.247 [ms] (mean)
Time per request:       2.112 [ms] (mean, across all concurrent requests)
Transfer rate:          6079.26 [Kbytes/sec] received
```

473 reqs/s？也许你非常失望，相比于前面的 461.08 reqs/s，为什么性能的提高如此微不足道？我们这次可是在内存中读写缓存，避免了之前文件缓存加载时的磁盘 I/O 开销，为什么效果不明显呢？原因很简单，其实用 strace 跟踪 PHP 处理进程，你会发现，在动态网页处理过程中，需要加载的磁盘文件非常多，而缓存文件只是其中的一个，仅仅把它放到内存里，对整体的影响当然微不足道。

与 APC 类似的另一个 PHP 缓存扩展是 XCache，我们稍微修改一下代码，使用 XCache 的数据缓存 API，修改后的代码如下所示：

```
$this->key = $this->template_page . $this->cache_id;
$html = xcache_get($this->key);
if ($html !== NULL)
{
    echo $html;
    exit(1);
}
do_some_db_query();
$html = $this->smarty->fetch($this->template_page, $this->cache_id);
xcache_set($this->key, $html, $this->smarty->cache_lifetime);
echo $html;
```

测试结果如下所示：

```
Document Path:          /place_posts_xcache.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   2.148 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      13312810 bytes
HTML transferred:       13130000 bytes
Requests per second:    462.05 [#/sec] (mean)
Time per request:       214.790 [ms] (mean)
Time per request:       2.148 [ms] (mean, across all concurrent requests)
Transfer rate:          6052.80 [Kbytes/sec] received
```

这个结果和使用 APC 的结果不相上下，需要说明的是，使用 APC 和 XCache 数据缓存方法没有什么本质的区别，测试结果的微小差异并不是绝对的，它们的表现都非常出色。

缓存服务器

除此之外，我们还可以将 HTML 缓存存储在一台独立的缓存服务器中，利用 memcached，我们可以很容易地通过 TCP 将缓存存储在其他服务器中，而且 memcached 同样也是使用内存空间来保存缓存数据，减少了不必要的磁盘 I/O。另一方面，memcached 在存储区中对于每一个 key 都维护一个过期时间，一旦达到这个过期时间，memcached 便会自动删除这个 key，这使得我们的过期检查非常容易，只需要在保存缓存数据时指定过期时间即可。应用了 memcached 后的代码片段如下所示：

```
    $this->key = $this->template_page . $this->cache_id;
    $memcache = memcache_connect('s-memcache', 11711);
    $html = $memcache->get($this->key);
    if ($html !== false)
    {
        echo $html;
        exit(1);
    }
    do_some_db_query();
    $html = $this->smarty->fetch($this->template_page, $this->cache_id);
    $memcache = memcache_connect('s-memcache', 11711);
    $memcache->set($this->key, $html, MEMCACHE_COMPRESSED, $this->smarty->
cache_lifetime);
    echo $html;
```

在以上代码中，主机 s-memcache 是与 Web 服务器处于同一局域网中的缓存服务器，它们之间的传输延迟可以忽略。对其进行同样的压力测试，结果如下所示：

```
Document Path:          /place_posts_memcache.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   2.573 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      13166215 bytes
HTML transferred:       13000000 bytes
Requests per second:    388.62 [#/sec] (mean)
Time per request:       257.321 [ms] (mean)
Time per request:       2.573 [ms] (mean, across all concurrent requests)
Transfer rate:          4996.72 [Kbytes/sec] received
```

看起来结果不是很好，毕竟它存在 TCP socket 操作的开销，但也不是很糟糕，至少比 Smarty 内置缓存方法的性能要好很多。

如何选择

这里我们将前面的几种缓存方法对应的压力测试结果汇总一下，如表 4-1 所示。

表 4-1　几种缓存方法的压力测试结果对比

整页缓存方法	吞吐率（reqs/s）
不使用缓存	51.59
Smarty cache	173.95
file cache	461.08
APC cache	473.38
XCache cache	462.05
memcache cache	388.62

如图 4-1 所示的对比图可以更加直观地呈现出它们的差异。

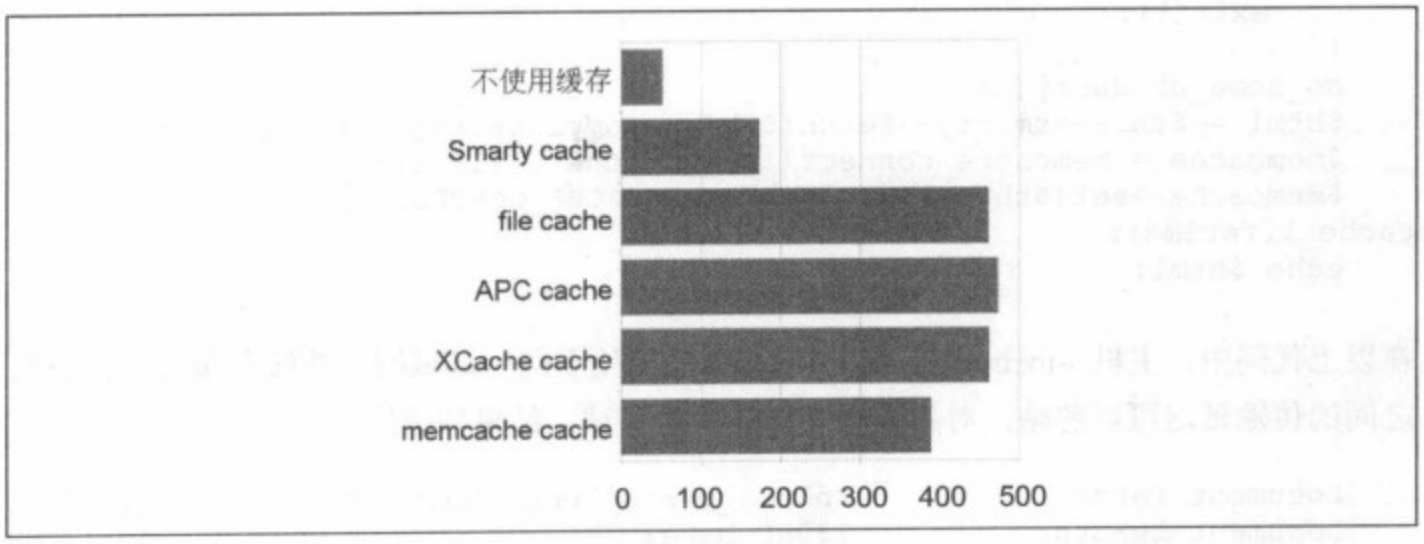


图 4-1　各种缓存方法压力测试结果对比图

也许你一眼就看到 APC cache 的性能优势，当然，这不是 APC 的特权，主要归咎于它使用了本地内存来存储缓存数据。但是需要提醒的是，无论是 APC 还是 memcached，都使用内存来存储 HTML 缓存，以上我们的测试都只针对一个固定的动态网页，每次访问都处于缓存有效期内，所以缓存命中率是 100%，而在实际情况中，一个站点包含了大量的动态网页，如果你为缓存分配的内存空间不足以容纳所有的缓存数据，便会使得缓存命中率大幅度下降，吞吐率也会随之降低。

这并不夸张，假如你的站点有数十万个动态网页需要保存 HTML 缓存，假设每个缓存为 100KB，那么 10 万个缓存的总大小就是 1GB，如果你希望为每个缓存保持较长的有效期，那么就需要准备好足够的内存空间，好吧，购买更多的内存，这似乎已经成为一个经济问题。但是，一旦本地内存达到上限，这似乎又变成了一个技术问题，往往到了这时候，你的站点规模已经让你不得不考虑比性能更加重要的问题，那就是如何扩展缓存存储区，显然，使用 memcached 来实现分布式缓存使得扩展成为可能。

就速度而言，memcached 不如使用本机内存的速度快，但出于多方面的考虑，结合你的站点规模，也许你会觉得这很值得。首先，Web 服务器特别是应用服务器本身的内存是相当宝贵的，它要满足 HTTP 进程和脚本解释器的大量开销，无法拿出大量的空间来存放 HTML 缓存；其次，使用本机内存不具备良好的扩展性，一旦缓存数据和站点负载大幅度增加，为了保证较高的缓存命中率，必须加大缓存空间，本机内存显然成为瓶颈，而使用独立的缓存服务器可以便于扩展，构成多台服务器组成的分布式缓存系统，这在后续章节中我们会深入探讨。

当然，对于小规模或者初创时期的 Web 站点，如果需要缓存的动态网页比较少，这时候使用 APC 内存缓存仍然不失为一个快速有效的方案，即便是规模膨胀后，快速重构缓存机制并且迁移缓存数据也不是一件多么复杂的事情，从前面的代码片段可以看出，从 APC 过渡到 memcached 非常容易，往往我们无法预计未来，但是只要做好充分的准备就可以了。

有效期取值

前文中多次提到缓存有效期，事实上对于它的取值并不是一件容易的事情。有效期如果太长，虽然缓存命中率提高了，但是动态网页的内容更新总是不能及时展现；而有效期如果太短，动态网页的内容虽能实时变化，但是频繁地创建缓存比没有缓存也好不到哪儿去。可见，这两种情况，正是“过”与“不及”的两种表现，而成功往往就在“过”与“不及”之间。另一方面，我们往往无法确定一个长期有效的取值，而事实上仅仅这个取值没有任何意义，更重要的是我们需要具备能够意识到有效期何时需要变化的能力，然后在任何时候找到当下最适合的取值。

幸运的是，除了不断调整缓存有效期之外，缓存机制还提供了一个有效的缓存控制途径，那就是可以在任何时候强行清除缓存，这在动态内容更新频率较低的时候适合使用，比如通常在一个 Blog 中，阅读的人总是远远多于撰写内容的人，这样就可以将缓存有效期设置得相对长一些，而每次内容更新时都主动清除缓存，促使动态内容创建新的缓存。

如果使用了 Smarty 的内置缓存方法，那么清除缓存非常容易，比如我们清除前面那个动态网页的缓存，如下所示：

```
$this->smarty->clear_cache('place_posts.htm', '12882');
```

这时候，Smarty 会找到缓存文件，然后删掉它。然而，还记得我们刚刚实现的那些缓存方法吗？请允许我暂时称它们为山寨版实现，因为它们还不完整，至少还没有提供清除缓存的接口。的确，如果放弃使用 Smarty 的缓存方法，就要付出代价来实现缓存管理接口，不过这也不是一件困难的事情，想想换取的性能提升，这还是值得的。

4.4 局部无缓存

对于有些特殊的动态网页，需要页面中某一块区域的内容及时更新，比如一个新闻正文页面的阅读量统计区域和评论区域，如果为了这一块区域的及时更新，就将整个页面重新创建缓存的话，的确有点不值得。在流行的模板框架中，在整页缓存的基础上，都提供了局部无缓存的支持，它允许在页面中指定一块包含动态数据的 HTML 代码段，每次这些动态数据都需要实时计算，然后和其余部分的缓存合成为最终的网页。

Smarty 中要实现局部无缓存，可以增加一个模板扩展标记，并且注册到 Smarty 对象中，如下所示：

```
$this->smarty->register_block('dynamic', 'smarty_block_dynamic', false);
function smarty_block_dynamic($param, $content, &$smarty)
{
    return $content;
}
```

这样一来，在模板中只需要将不希望缓存的部分用<dynamic>标签包围起来即可，如下所示：

```
{dynamic}
$user->user_nick
{/dynamic}
```

而在 cakephp 框架中，需要用<cake:nocache>标签将视图中的无缓存区域包围起来，也可以实现同样的效果，以下是 cakephp cookbook 中的示例：

```
<cake:nocache>
<?php if ($session->check('User.name')) : ?>
    Welcome, <?php echo $session->read('User.name')?>.
<?php else: ?>
    <?php echo $html->link('Login', 'users/login')?>
<?php endif; ?>
</cake:nocache>
```

引入局部无缓存机制后，动态网页控制器也要进行相应的修改，其目的就是把无缓存区域对应的动态数据计算提到缓存检查之前，这样才可以保证每次都将实时计算后的结果输出到模板中，具体代码的修改不是我们这里讨论的重点，但是这样做对于性能是有影响的，需要注意的是，一定要评估局部动态数据的影响力，如果你将一个动态网页中占主要开销的数据计算置于无缓存状态，那么缓存对你来说已经失去了意义，你可以考虑选择其他的缓存方式或者页面组织结构，比如使用数据层缓存，这在后续章节中会有所介绍。

4.5 静态化内容

前面我们尝试的几种动态内容缓存方法中，动态程序看起来就像是缓存的代理人，每次用

户的请求都要首先被送到动态程序，然后动态程序根据缓存有效期来决定是否输出缓存。这种方法使得动态内容对于缓存数据有着较强的控制权，但是这种控制权的代价是比较昂贵的，它带来了性能上的不足。

直接访问缓存

为什么不让用户直接请求 HTML 缓存呢？听起来足够返璞归真，充满原生态气息。没错，早在互联网的石器时代，人们就在直接访问 HTML，而我们为什么要搞得如此复杂？

还记得前面的动态网页 place_posts.php 吗？我们这里将它的缓存从 cache 目录中拿出来，放到 place_posts.php 的同目录下，重命名为 place_posts.htm，这样便可以直接通过浏览器访问到它，还等什么，再来一次压力测试吧，看看结果：

```
Document Path:          /place_posts.htm
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   0.085 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      13367350 bytes
HTML transferred:       13130000 bytes
Requests per second:    11769.00 [#/sec] (mean)
Time per request:       8.497 [ms] (mean)
Time per request:       0.085 [ms] (mean, across all concurrent requests)
Transfer rate:          153633.12 [Kbytes/sec] received
```

11769 reqs/s 的吞吐率不得不让我们为之赞叹，它要比前面的动态内容缓存方法高出百倍。事实上在前面介绍服务器并发处理能力的时候，我们对大量并发用户数下静态文件的良好表现印象深刻，但是在这里，也许会让你更加深刻地意识到动静的巨大差距。

值得一提的是，也许你注意到了测试结果中的 Transfer rate，可以看出，如此高吞吐率的前提条件是必须拥有大约 150 × 8Mbit/s 的出口带宽，也就是 1.2G 的带宽，而事实上这还只是 HTTP 数据包的长度，加上 IP 包的其他信息，实际带宽还要更多一些。

之前我们是在静态网页所在的服务器上执行测试，数据没有流出服务器，所以不存在出口带宽限制的问题。而在实际情况中，吞吐率往往受限于 Web 服务器的出口带宽，比如我们购买了 100M 独享带宽，那么按照这个 13KB 的网页来计算，如果不考虑 IP 包的其他信息长度，理论上最大的吞吐率为：

((100Mbit / 8) × 1000) / 13KB = 961.53 reqs/s

也就是说，在使用 100M 独享带宽的服务器上，这个 13KB 的网页的吞吐率最高只能达到 961.53 reqs/s。我们来做个尝试，这次选择在另一台服务器上运行 ab 来测试这台服务器的

静态文件，两台服务器通过 100M 交换机连接，这相当于被测试的服务器拥有 100M 的出口带宽。测试结果如下所示：

```
Document Path:          /place_posts.htm
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   1.164134 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      13243688 bytes
HTML transferred:       13008218 bytes
Requests per second:    859.01 [#/sec] (mean)
Time per request:       116.413 [ms] (mean)
Time per request:       1.164 [ms] (mean, across all concurrent requests)
Transfer rate:          11109.55 [Kbytes/sec] received
```

的确，吞吐率只达到了 859.01 reqs/s，加上 IP 数据包的其他信息长度和网络 I/O 开销，这个结果在我们的预料之中。

所以，要想让静态化页面达到或接近理论上的吞吐率，你可以购买更多的独享带宽（如 1G 带宽），然后为服务器安装千兆网卡，服务器所在网络也要使用千兆交换机，这样一来，静态化页面的性能才可以发挥到极限。

另一方面，我们做一个比较现实的估算，如果仅仅站点页面就需要消耗 1G 的带宽，那么页面中的组件比如视频、图片、样式表、JS 脚本等，其消耗的带宽将会在数倍以上，这要根据站点的性质来决定，以一个最极端的要数视频网站为例，根据最近 Alexa 上对于 youtube 的保守统计，它的 PV 平均每天至少为 1,776,895,200，那么平均每秒的吞吐率为 20566 reqs/s，注意这只是 HTML 网页的吞吐率，而且其中不包括大量通过 AJAX 加载的局部页面，如果这些网页都至少包含一个视频，每个视频平均按照 10MB 来计算，那么每天的视频流量为 17,352,492GB，平均每秒为 200GB，换算成比特，即视频总带宽为 1600Gbit/s，别忘了这还只是平均计算值，实际在访客高峰时期还要更多。

尽管静态化网页的性能要大大高于动态缓存的性能，但是我们要知道，静态化网页在请求的时候不涉及内容计算，这并不代表它不需要计算，它仍然需要由动态程序来创建和更新，我们不可能手动维护一个站点的所有静态网页，那简直无法想象，与之相比，动态内容缓存的一部分优越性便体现在这一点，通过封装良好的缓存管理机制，它成为名副其实的缓存代理人。

然而，当我们一旦使用静态化网页的缓存方案后，你会发现原来的一切都被彻底颠覆。缓存更新、过期检查，以及缓存持久化等，都不能按照原来的方式来设计，我们必须重新来审视静态化缓存方案的具体实现。

我们一般会使用 CMS（内容管理系统）来管理静态化内容，同时 CMS 也可以在必要的时候帮助我们更新静态化内容。具体的 CMS 使用不是我们讨论的重点，市场上有很多优秀的 CMS 产品，有商用的和开源的，你可以根据需要来选择。

更新策略

对于静态化内容的更新策略，一般有以下两种：

- 在数据更新时重新生成静态化内容。
- 定时重新生成静态化内容。

对于前者，在数据更新的时候重建静态化缓存，往往由用户的某些动作触发，比如新闻站点的网络编辑发表一篇新闻后，程序便创建一个新的静态化新闻页面，同时更新新闻列表页面。显然，这种方式在数据更新频繁时存在大量重建静态内容的开销，尤其是当一个动作引发大量静态内容需要更新时，比如大型新闻站点的 CMS（内容管理系统）在工作高峰期间，所有编辑都修改新闻标题而引发大量的页面或局部页面频繁更新，同时伴随着频繁的数据库操作，这将导致 CMS 系统的服务响应大幅度降低，当然，这可能并不直接影响现存静态化页面的访问，但是不要忘了，这种更新机制也正是静态化缓存方案的一部分，所以它的性能也至关重要。另一方面，如果站点的静态化内容需要分发到更多的服务器，那么频繁的更新也会给文件的同步带来较大的压力，关于文件的分发和同步，我们会在后续章节中探讨。

一个常用的办法是引入延迟更新机制，将更新任务放入队列，一旦队列写满或者达到超时时间，便一次性将它们更新到磁盘，这听起来有点像文件系统的磁盘缓冲区的设计动机，没错，你也可以把它理解为静态化缓冲区。

另一种更新策略是定时重建静态化内容，它一般通过定时任务来执行，比如 crontab 或者专用的 daemon 程序，然后通过 CMS 系统进行方便的管理。如果站点存在大量的静态化内容，而且它们的性质和实时性需求不尽相同，我们可以通过维护一定的对应关系来指定特定范围的静态化内容进行重建，一个常见的例子是，对于热门滚动新闻列表进行每隔 1 分钟的定时更新，而对于其他重要级别稍低的内容可以延长更新时间，从而节省开销。

通常，这两种更新策略可以互相弥补，共同应用在站点的静态化方案中。如何选择并很好地使用它们，需要在重建性能和内容及时性需求之间取其平衡。

局部静态化

还记得前面介绍动态内容缓存时提到的局部无缓存支持吗？幸运的是，静态网页也可以不

必整页更新，它可以通过 SSI（服务器端包含）技术实现各个局部页面的独立更新，这样便大大节省了重建整个网页时的计算开销和磁盘 I/O 开销，甚至包括分发时的网络 I/O 开销。

SSI 技术现在可以在任何一个主流的 Web 服务器中找到相应的模块，比如 Apache 的 mod_include 和 Lighttpd 的 mod_ssi，其具体配置和使用非常简单，在这里就不具体介绍了，但是不要高兴得太早，任何节省都需要付出一定的代价。

我找来一个站点的首页 index.shtml，它自身的大小为 8472 个字节，同时它包含了 19 个子页面，分别通过 include 方法进行加载，最终的总大小为 50456 个字节，Web 服务器使用 Lighttpd 对这个静态化的首页进行压力测试，结果如下所示：

```
Document Path:          /index.shtml
Document Length:        50080 bytes
Concurrency Level:      100
Time taken for tests:   0.692 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      50294000 bytes
HTML transferred:       50080000 bytes
Requests per second:    1445.72 [#/sec] (mean)
Time per request:       69.170 [ms] (mean)
Time per request:       0.692 [ms] (mean, across all concurrent requests)
Transfer rate:          71006.86 [Kbytes/sec] received
```

吞吐率为 1445.72 reqs/s，我想这个结果对于 50KB 左右的静态网页来说确实不高，是否它在处理的时候加载了太多的其他文件，导致处理时间延长呢？答案是肯定的，我想如果你在阅读的时候没有直接跳到这里的话，你一定对磁盘 I/O 的缓慢速度印象深刻。

接下来，我们将最终合成的网页直接保存在服务器上，取名为 index_without_ssi.shtml，然后对它进行同样的压力测试，结果如下所示：

```
Document Path:          /index_without_ssi.shtml
Document Length:        50080 bytes
Concurrency Level:      100
Time taken for tests:   0.372926 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      50294000 bytes
HTML transferred:       50080000 bytes
Requests per second:    2681.50 [#/sec] (mean)
Time per request:       37.293 [ms] (mean)
Time per request:       0.373 [ms] (mean, across all concurrent requests)
Transfer rate:          132565.17 [Kbytes/sec] received
```

吞吐率提高了近一倍！可见打开这 19 个子页面确实消耗了不少开销，我们用 strace 跟踪 lighttpd 进程来看看磁盘 I/O 的变化。在使用 SSI 时候，结果如下所示：

% time	seconds	usecs/call	calls	errors	syscall
32.71	**0.000933**	**0**	**40006**		**open**
25.07	0.000715	0	39000		sendfile64
23.95	**0.000683**	**0**	**20019**		**stat64**
8.10	0.000231	0	41006		close
4.80	0.000137	0	41000		fcntl64
2.07	0.000059	0	2000		setsockopt
1.26	0.000036	0	1000		munmap
1.09	0.000031	0	1039	39	read
0.35	0.000010	0	1000		writev
0.32	0.000009	0	1000		mmap2
0.28	0.000008	0	1039		ioctl
0.00	0.000000	0	35		write
0.00	0.000000	0	14		time
0.00	0.000000	0	78		epoll_ctl
0.00	0.000000	0	13		epoll_wait
0.00	**0.000000**	**0**	**1000**		**accept**
100.00	0.002852		189249	39	total

在不使用 SSI 的时候，结果如下所示：

% time	seconds	usecs/call	calls	errors	syscall
49.52	**0.000414**	**0**	**3000**		**sendfile64**
17.58	0.000147	0	5002		close
12.08	0.000101	0	4002		open
6.34	0.000053	0	2000		setsockopt
3.35	0.000028	0	2007		stat64
2.99	**0.000025**	**0**	**1000**		**accept**
2.27	0.000019	0	1000		mmap2
1.56	0.000013	0	1000		munmap
1.20	0.000010	0	1000		writev
1.08	0.000009	0	1041	41	read
1.08	0.000009	0	5000		fcntl64
0.96	0.000008	0	1041		ioctl
0.00	0.000000	0	34		write
0.00	0.000000	0	11		time
0.00	0.000000	0	82		epoll_ctl
0.00	0.000000	0	10		epoll_wait
100.00	0.000836		27230	41	total

可以看出，同样都是接受了 1000 个请求，在使用 SSI 的时候，open()和 stat64()的执行次数较多，等待时间也较长，所占时间比例较大，它们用于打开文件和查看文件状态，是服务器加载子页面时执行的系统调用，同时，用于传输文件的 sendfile64()系统调用所占比例较小。而在不使用 SSI 的时候，sendfile64()占据了主要时间，也就是说更多的时间都花在了传输文件上，所以吞吐率要高一些。

但是，2681.50 reqs/s 的吞吐率似乎还是不高，还记得我们之前对 13KB 的静态网页压力测试可以达到上万的吞吐率吗？那么这个 50KB 的静态网页也不至于降低到如此地步吧？

问题就出在 SSI 的原理上，一旦网页支持 SSI，那么每次请求的时候服务器必须要通读网页内容，查找 include 标签，这需要大量的 CPU 开销。

刚才我们虽然没有使用 SSI，但是我们的网页后缀仍然是 shtml，Lighttpd 仍然会通读网页内容进行查找，这显然不是我们希望的结果。在 Lighttpd 的配置中，我们指定了对所有 shtml 的网页进行 SSI 检查：

```
ssi.extension              = ( ".shtml" )
```

在 Apache 中，一般配置如下所示：

```
AddType text/html .shtml
AddOutputFilter INCLUDES .shtml
```

了解了这些内容后，我们修改这个静态网页的后缀，将其改为 index_without_ssi.htm，再来试试，结果如下所示：

```
Document Path:          /index_without_ssi.htm
Document Length:        50080 bytes
Concurrency Level:      100
Time taken for tests:   0.124 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      50417634 bytes
HTML transferred:       50180160 bytes
Requests per second:    8034.84 [#/sec] (mean)
Time per request:       12.446 [ms] (mean)
Time per request:       0.124 [ms] (mean, across all concurrent requests)
Transfer rate:          395603.10 [Kbytes/sec] received
```

这下你该可以接受现在的结果了，顺便看看数据传输率，接近每秒 400MB，这相当于需要 3.2G 的带宽，简直比我们前面测试 13KB 时的 1.2G 带宽要求还要高。这里顺便说一下，或许你也已经发现了，请求的文件越大，花在文件传输上的时间越多，单位时间的数据传输量也越大，同时花在处理 socket 连接以及打开文件等的时间比例越小，比如我们请求一个 39MB 的文件，结果如下所示：

```
Document Path:          /test.flv
Document Length:        39921980 bytes
Concurrency Level:      1
Time taken for tests:   0.065 seconds
Complete requests:      1
Failed requests:        0
Write errors:           0
Total transferred:      39922234 bytes
HTML transferred:       39921980 bytes
Requests per second:    15.43 [#/sec] (mean)
Time per request:       64.811 [ms] (mean)
Time per request:       64.811 [ms] (mean, across all concurrent requests)
Transfer rate:          601542.28 [Kbytes/sec] received
```

可以看到平均每秒输出 600MB 左右的数据，那么带宽则需要 4.8G，这太惊人了。

回到刚才的话题，我们将 shtml 后缀改为 htm 后，Lighttpd 在处理静态内容时便不会通读网页内容，这时候我们同样用 strace 跟踪 lighttpd 进程，结果如下所示：

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 66.29    0.000297           0      1000           sendfile64
  9.82    0.000044           0      1002           open
  9.38    0.000042           0      2002           close
  6.47    0.000029           0      1000           writev
  3.79    0.000017           0      2000           setsockopt
  2.23    0.000010           0      1054        54 read
  2.01    0.000009           0      3000           fcntl64
  0.00    0.000000           0        33           write
  0.00    0.000000           0        22           time
  0.00    0.000000           0      1054           ioctl
  0.00    0.000000           0         7           stat64
  0.00    0.000000           0       108           epoll_ctl
  0.00    0.000000           0        21           epoll_wait
  0.00    0.000000           0      1000           accept
------ ----------- ----------- --------- --------- ----------------
100.00    0.000448                 13303        54 total
```

可以看出，用于数据传输的 sendfile64()系统调用占用了更多的时间，而其他系统调用的时间比例相对有所减少，但是你也许会发现，从系统调用的总时间 0.000448s 来看，相比于使用 shtml 时的 0.000836s，减少的比例似乎不足以让它的吞吐率提升如此之多，如果你对这个问题存有疑惑，那么需要说明的是，到目前为止，我们多次使用 strace 来跟踪进程，跟踪的目的是观察进程花在系统调用上的时间，这些系统调用都发生在内核态，大部分代表着操作系统对外设的操作，比如各种 I/O 操作等。但是对于用户态的 CPU 计算，strace 是无法跟踪的，所以当我们使用 htm 后，节省了原来大量的扫描静态文件内容的 CPU 计算，这些时间在 strace 中是没有反映的。我们将刚才的几个测试结果生成如图 4-2 所示的对比图，你可以直观地看到它们的差距。

说到这里，也许你误解了我的意思，再也不敢使用 SSI 了，其实你完全可以使用，而且我们大力推荐你使用，因为它的好处我们在开头就说过了，同时，从静态内容管理的角度看，它可以提高页面的重用性，让海量的内容更加富有条理，大大提高了可维护性。

既然这样，那我们介绍 SSI 对性能的影响，有什么意义呢？

首先，通过这一系列的测试，我们可以深刻了解 SSI 的本质，并且认识到它对性能影响的程度，只有认识本质后，我们才可以在使用中根据需要来把握平衡，在站点负载不大或者带宽限制的情况下，完全可以使用必要的 include 来更好地管理静态内容。就拿前面使用 SSI 的 index.shtml 来说，当达到最大吞吐率时，数据传输率为每秒 71006.86KB，这相当于需要 560M 的独享带宽，而你如果只买了 100M 的共享带宽的话，那根本没有必要担心

静态内容被扫描的开销。

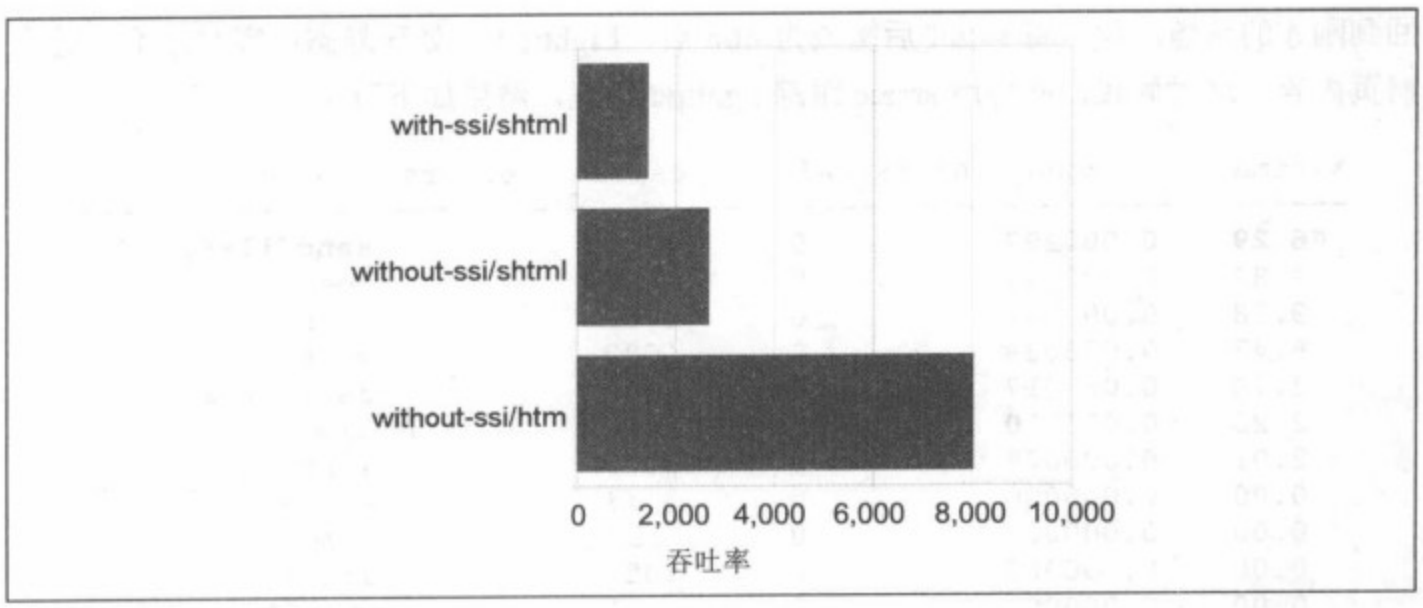


图 4-2　使用 SSI/shtml/htm 的压力测试结果对比图

其次，这也是非常重要的一点，在我们的站点中，对于一些没有使用 SSI 的静态内容，如果它们也被一视同仁地扫描一遍，那就变得非常无辜了，而我们要做的正是减少这部分不必要的开销。虽然我们可以将使用 SSI 的静态内容通过后缀来区分开（如 shtml），但实际上，往往站点中的一些静态网页已经开放很长时间，无法随意修改后缀，所以我们最后不得不直接对所有 htm 后缀的文件打开 SSI 模式，这让大部分拥有 htm 后缀但是没有使用 SSI 的静态网页深受其害。

幸运的是，Apache 提供了一个思路，它允许打开 XBitHack 模式，如下所示：

```
XBitHack on
```

这样一来，Apache 将会对所有拥有执行权限的静态网页开启 SSI 模式，比如我们要对前面的 index_without_ssi.htm 开启 SSI 模式，只需要修改它的权限即可，如下所示：

```
chmod +x index_without_ssi.htm
```

如果觉得每次都要手动修改权限很不现实，我们完全可以通过 CMS 来自动完成。

第5章 动态脚本加速

通过为动态内容的计算结果生成缓存，我们达到了一定的目的，那就是最大程度地跳过动态内容计算。然而，除非静态化访问，否则完全跳过动态内容的计算是不可能的，通过前面的介绍，我们知道加载缓存仍然需要动态脚本的运行，那么，为了提高动态内容的处理速度，我们还能做些什么呢？

5.1 opcode 缓存

在动态脚本运行的微观世界里，缓存仍然有用武之地，为此，我们先来看看什么是 opcode。

什么是 opcode

也许你曾经尝试过用 C/C++编写动态内容，虽然开发过程极其烦琐，但为了获得性能提升，这样做或许是值得的，它们可将动态内容编译成二进制可执行文件，也就是目标代码，由操作系统进程直接装载运行。

如今已经很少有人使用 C/C++编写动态内容了，绝大多数 Web 开发人员享受着当下幸福的时光，很多优秀的脚本语言可供选择，比如 PHP、Ruby、Python，它们都属于解释型语言，所以用它们编写的动态内容都需要依赖相应的解释器程序来运行。

解释器程序也是一个二进制可执行文件，比如/bin/ruby，它同样可以直接在进程中运行，在运行过程中，解释器程序需要对输入的脚本代码进行分析，领会它们的旨意，然后执行它们。比如下面我们调用 PHP、Ruby、Python 的解释器，分别让它们执行一段简单的脚本代码：

```
s-colin:~ # php -r 'print 1+1;'
2
s-colin:~ # ruby -e 'print 1+1'
2
s-colin:~ # python -c 'print 1+1'
2
```

很好，它们都很快地输出了正确结果，也许你觉得 1+1 的计算太小儿科，的确，在人类的大脑中计算 1+1 是很简单，几乎不用思考，但解释器的工作方式可不是你想象的那样，1+1

和 100+1000 对它来说几乎没什么区别，因为解释器核心引擎根本看不懂这些脚本代码，无法直接执行，所以需要进行一系列的代码分析工作，当解释器完成对脚本代码的分析后，便将它们生成可以直接运行的中间代码，也称为操作码（Operate Code, opcode）。

从程序代码到中间代码的这个过程，我们称为解释（parse），它由解释器来完成。与此相似的是，编译型语言中的编译器（如 C 语言的编译器 gcc），也要将程序代码生成中间代码，这个过程我们称为编译（compile）。

编译器和解释器的一个本质不同在于，解释器在生成中间代码后，便直接执行它，所以运行时的控制权在解释器；而编译器则将中间代码进一步优化，生成可以直接运行的目标程序，但不执行它，用户可以在随后的任意时间执行它，这时控制权在目标程序，和编译器没有任何关系。

事实上，就解释和编译本身而言，它们的原理是相似的，都包括词法分析、语法分析、语义分析等，所以，有些时候将解释型语言中生成 opcode 的过程也称为“编译”，需要你根据上下文来理解。

正是因为解释器每次运行的时候都将脚本代码作为输入数据来分析，所以它的数据结构可以动态改变，这使得解释型语言具备了很多丰富的动态特性，在开发和调试中有很多优势，特别是一些流行的 Web 开发框架，其中的一些特性如果没有动态语言的支持是无法实现的，这点我想你一定深有体会，但是很遗憾，应用动态语言来编写程序的技巧不是本书的讨论主题。

那么，opcode 究竟是什么样的呢？它又是如何解释生成的呢？我们以 PHP 为例，来寻找这些答案。

PHP 解释器的核心引擎为 Zend Engine，可以很容易查看它的版本：

```
s-colin:~ # php -v
PHP 5.2.8 (cli) (built: Feb 25 2009 21:58:30)
Copyright (c) 1997-2008 The PHP Group
Zend Engine v2.2.0, Copyright (c) 1998-2008 Zend Technologies
```

还记得前面我们曾经用 PHP 计算 1+1 的脚本代码吗？我们来看看这段代码的 opcode。在此之前，需要安装 PHP 的 Parsekit 扩展，它是一个用 C 编写的二进制扩展，由 PECL 来维护。有了 Parsekit 扩展后，我们就可以通过它提供的运行时 API 来查看任何 PHP 文件或者代码段的 opcode。我们直接在命令行中调用 parsekit_compile_string()，如下所示：

```
s-colin:~ # php -r "var_dump(parsekit_compile_string('print 1+1;'));"
```

这样一来，我们便获得了这段代码的 opcode，返回的是数组形式，结果如下所示：

```
array(20) {
```

```
["type"]=> int(4)
["type_name"]=> string(14) "ZEND_EVAL_CODE"
["fn_flags"]=> int(0)
["num_args"]=> int(0)
["required_num_args"]=> int(0)
["pass_rest_by_reference"]=> bool(false)
["uses_this"]=> bool(false)
["line_start"]=> int(0)
["line_end"]=> int(0)
["return_reference"]=> bool(false)
["refcount"]=> int(1)
["last"]=> int(5)
["size"]=> int(5)
["T"]=> int(2)
["last_brk_cont"]=> int(0)
["current_brk_cont"]=> int(-1)
["backpatch_count"]=> int(0)
["done_pass_two"]=> bool(true)
["filename"]=> string(17) "Parsekit Compiler"
["opcodes"]=> array(5)
{
  [0]=> array(7)
  {
    ["opcode"]=> int(1)
    ["opcode_name"]=> string(8) "ZEND_ADD"
    ["flags"]=> int(197378)
    ["result"]=> array(3)
    {
      ["type"]=> int(2)
      ["type_name"]=> string(10) "IS_TMP_VAR"
      ["var"]=> int(0)
    }
    ["op1"]=> array(3)
    {
      ["type"]=> int(1)
      ["type_name"]=> string(8) "IS_CONST"
      ["constant"]=> &int(1)
    }
    ["op2"]=> array(3)
    {
      ["type"]=> int(1)
      ["type_name"]=> string(8) "IS_CONST"
      ["constant"]=> &int(1)
    }
    ["lineno"]=> int(1)
  }
  [1]=> array(6)
  {
    ["opcode"]=> int(41)
    ["opcode_name"]=> string(10) "ZEND_PRINT"
    ["flags"]=> int(770)
    ["result"]=> array(3)
    {
      ["type"]=> int(2)
      ["type_name"]=> string(10) "IS_TMP_VAR"
      ["var"]=> int(1)
    }
    ["op1"]=> array(3)
```

```
    {
      ["type"]=> int(2)
      ["type_name"]=> string(10) "IS_TMP_VAR"
      ["var"]=> int(0)
    }
    ["lineno"]=> int(1)
  }
  [2]=> array(6)
  {
    ["opcode"]=> int(70)
    ["opcode_name"]=> string(9) "ZEND_FREE"
    ["flags"]=> int(271104)
    ["op1"]=> array(4)
    {
      ["type"]=> int(2)
      ["type_name"]=> string(10) "IS_TMP_VAR"
      ["var"]=> int(1)
      ["EA.type"]=> int(0)
    }
    ["op2"]=> array(3)
    {
      ["type"]=> int(8)
      ["type_name"]=> string(9) "IS_UNUSED"
      ["opline_num"]=> string(1) "0"
    }
    ["lineno"]=> int(1)
  }
  [3]=> array(6)
  {
    ["opcode"]=> int(62)
    ["opcode_name"]=> string(11) "ZEND_RETURN"
    ["flags"]=> int(16777984)
    ["op1"]=> array(3)
    {
      ["type"]=> int(1)
      ["type_name"]=> string(8) "IS_CONST"
      ["constant"]=> &NULL
    }
    ["extended_value"]=> int(0)
    ["lineno"]=> int(1)
  }
  [4]=> array(4)
  {
    ["opcode"]=> int(149)
    ["opcode_name"]=> string(21) "ZEND_HANDLE_EXCEPTION"
    ["flags"]=> int(0)
    ["lineno"]=> int(1)
  }
 }
}
```

看上去似乎很复杂，不要着急，仔细分析后，你会发现在 opcodes 数组中，一共有 5 条操作，我们将这些操作整理为一个表格，如表 5-1 所示。

表 5-1　opcode 操作列表

opcode	opcode_name	op1	op2	result
1	**ZEND_ADD**	IS_CONST(1)	IS_CONST(1)	IS_TMP_VAR
41	**ZEND_PRINT**	IS_TMP_VAR		IS_TMP_VAR
70	**ZEND_FREE**	IS_TMP_VAR		
62	**ZEND_RETURN**			
149	**ZEND_HANDLE_EXCEPTION**			

如果你熟悉汇编语言，至少看过汇编代码，那么你会感觉 opcode 和汇编代码非常相似，没错，解释器核心引擎正是沿用了这种思想，将所有的操作抽象为类似汇编语言一样的操作码形式，这种操作码称为三地址码，它是中间代码的一种理想的表现形式，顾名思义，它的每一个运算由不超过三个地址组成：op1、op2、result，它们可以表示多种运算形式。可以看出，表 5-1 中的前三个运算，便分别是这三种运算形式。

```
result = op1 op op2
result = op op1
op op1
```

三地址码与汇编指令在结构上非常接近，所以从三地址码生成目标文件非常容易，只需要将抽象的操作指令（如上边的 ZEND_ADD），结合机器硬件以及操作系统平台等实际环境，翻译成底层的操作指令即可。虽然解释型语言不需要生成目标文件，但是这种便利对于解释器执行中间代码也是非常有利的。另一方面，解析器维护抽象层面的操作码，也是其支持跨平台运行的重要基础。

顺便提一下，三地址码在形式上一个运算最多只支持二元运算组成的赋值语句，比如要计算 x=a+b*c，它将由多个运算组成，如下所示：

```
T1  = b * c
T2 = a + T1
x = T2
```

生成 opcode

看到 opcode 的模样后，还有一个重要的问题没有解决，究竟这些 opcode 是如何生成的呢？为了跟踪整个过程，我们以 ZEND_PRINT 这个指令为例，来看看它是如何出现的，但是，我们的重点不是探讨编译原理，所以，下面会尽量将过程简化为几个主要环节来介绍。

实际上，我们的脚本代码可以看成是一系列单词的集合，这些单词包括关键字、标识符、

运算符等，所以解析器首先需要对所有单词进行分类，并给它们打上记号（token），这个过程称为词法分析。我们在 PHP 源代码的 Zend 目录中，可以找到 PHP 解释器的词法规则文件，其中便有 print 对应的记号，如下所示：

```
Zend/zend_language_scanner.l
<ST_IN_SCRIPTING>"print" {
        return T_PRINT;
}
```

可见，print 对应的记号是 T_PRINT。如果所有代码都顺利通过词法分析后，接下来，解释器要对这些记号进行语法分析，我们找到了以下片段：

```
Zend/zend_language_parser.y
T_PRINT expr  { zend_do_print(&$$, &$2 TSRMLS_CC); }
```

语法分析器将 T_PRINT 标记以及上下文替换成了 zend_do_print()函数，我们接着找到这个函数的实现代码，如下所示：

```
Zend/zend_compile.c
void zend_do_print(znode *result, znode *arg TSRMLS_DC)
{
        zend_op *opline = get_next_op(CG(active_op_array) TSRMLS_CC);
        opline->result.op_type = IS_TMP_VAR;
        opline->result.u.var = get_temporary_variable(CG(active_op_array));
        opline->opcode = ZEND_PRINT;
        opline->op1 = *arg;
        SET_UNUSED(opline->op2);
        *result = opline->result;
}
```

可以看出，以上的 zend_do_print()函数实现了到 opcode 的转换，它设置了 opcode 的指令以及 op1 的数据，这样一来我们便得到了 opcode。另外，所有 opcode 指令都用整数来表示，以下是它们的宏定义：

```
Zend/zend_vm_opcodes.h
#define ZEND_ADD                                1
......
#define ZEND_PRINT                              41
......
#define ZEND_RETURN                             62
......
#define ZEND_FREE                               70
......
#define ZEND_HANDLE_EXCEPTION                   149
```

避免重复编译

了解了 opcode 及其编译过程后，现在回到我们的主题，它能为我们的性能优化做点什么

呢？也许你还没有意识到生成 opcode 这一过程的开销，只因为刚才我们的计算量实在是太少，那么，现在我们应该回到真实的 Web 应用中了，可以肯定的是，生成 opcode 的开销肯定存在，甚至非常可观，我们能否想办法来节省这部分开销呢？

也许你和我的想法一样，毫不犹豫地将动态内容缓存思想沿用在这里即可，通过 opcode 缓存，来避免重复的 opcode 编译。

这引发了我对之前探讨人脑计算 1+1 的思考，似乎我们也应用了某种意义上的 opcode 缓存，至少我在计算 1+1 的时候，根本没有实际去计算，只是感觉从某个地方直接蹦出来了“2”，这个结果完全来源于多年的“经验缓存”。

可是，要缓存 opcode，我们在应用层是无能为力的。幸运的是，有一些优秀的 opcode 缓存器扩展，比如 PHP 可以选择 APC、eAccelerator、XCache 等，它们都可以将 opcode 缓存在共享内存中，而且你几乎不需要修改任何代码。

APC

我们先来试试 APC，记得要在 php.ini 中打开 opcode cache 的开关：

```
apc.cache_by_default = on
```

你也可以通过 apc.filters 让 APC 只对特定范围的动态程序进行 opcode 缓存。

还记得前面的动态网页 place_posts.php 吗？它在使用 Smarty 内置缓存方法的时候，吞吐率只有 173.95 reqs/s，为此我们曾经对 Smarty 内置缓存方法深恶痛绝。这里我们在使用 APC opcode cache 的情况下，对它进行了一次同样的压力测试，结果如下所示：

```
Document Path:          /place_posts_cache.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   0.885 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      13164996 bytes
HTML transferred:       13000000 bytes
Requests per second:    1148.70 [#/sec] (mean)
Time per request:       88.522 [ms] (mean)
Time per request:       0.885 [ms] (mean, across all concurrent requests)
Transfer rate:          14523.44 [Kbytes/sec] received
```

什么？从 173.95reqs/s 到 1148.70reqs/s，虽然你可能不相信，但这的确是事实。借助于 opcode 缓存的神奇魔力，我们将之前的几种动态内容整页缓存方法再次进行压力测试，得出的结果如表 5-2 所示。

表 5-2　应用 opcode 缓存前后的吞吐率比较

	无脚本加速	APC opcode cache
不使用缓存	51.59	92.96
Smarty cache	173.95	1148.70
file cache	461.08	2173.45
APC cache	473.38	2509.73
memcache cache	388.62	1516.66

对比图如图 5-1 所示。

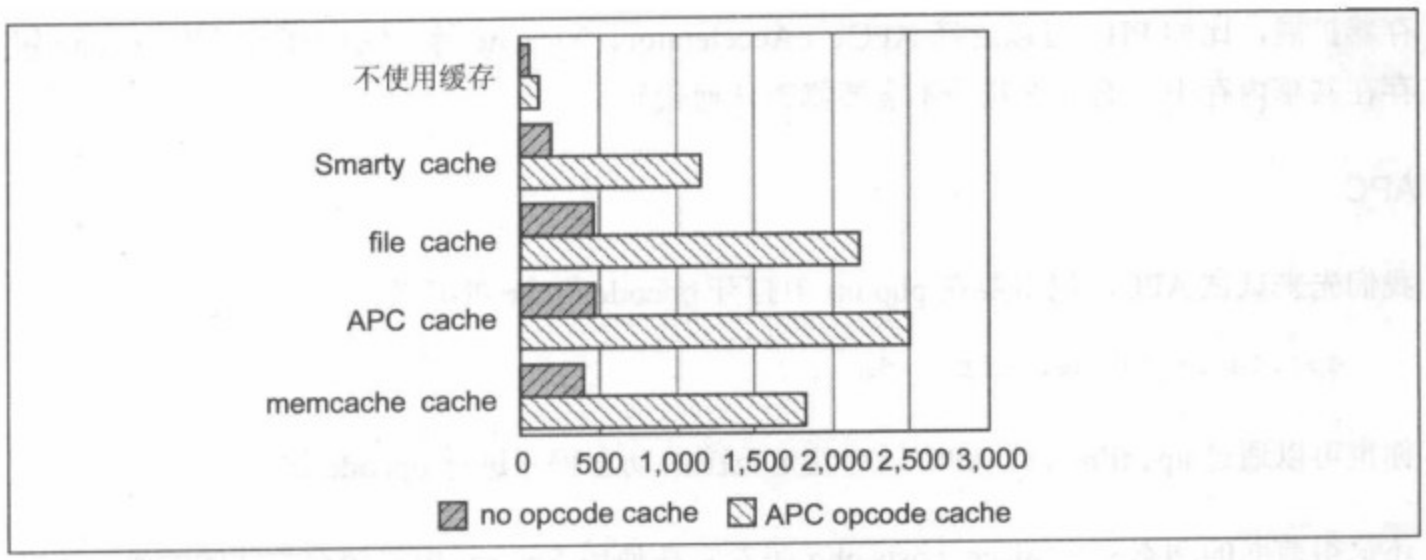


图 5-1　使用 APC opcode cache 前后的压力测试对比图

如此大的性能提升，不仅因为避免了重复的 opcode 编译开销，还得益于 APC 将 opcode 缓存在高速的内存中。既然将 opcode 缓存在内存中，我们就必须了解内存的使用情况以及缓存命中率，APC 提供了这样的 API，你可以通过它获得 APC 运行时的一些必要信息：

```
<?php
print_r(apc_cache_info());
?>
Array
(
    [num_slots] => 2000
    [ttl] => 0
    [num_hits] => 41991
    [num_misses] => 35
    [start_time] => 1236382228
    [expunges] => 0
    [mem_size] => 1508825
    [num_entries] => 35
    [num_inserts] => 35
    [file_upload_progress] => 1
    [memory_type] => mmap
```

```
        [locking_type] => pthread mutex
        [cache_list] => Array
            (
                [0] => Array
                    (
                        [filename] => /data/www/site/htdocs/place_posts.php
                        [device] => 2053
                        [inode] => 475922
                        [type] => file
                        [num_hits] => 714
                        [mtime] => 1236164760
                        [creation_time] => 1236382581
                        [deletion_time] => 0
                        [access_time] => 1236382594
                        [ref_count] => 0
                        [mem_size] => 20121
                    )
                ......
            )
        [deleted_list] => Array
            (
            )
    )
```

通过这些信息，我们可以知道当前有哪些 PHP 程序缓存了 opcode，以及它们的命中率和内存使用情况。另外，你也许看到了被缓存文件的 inode 值，这是文件系统中文件的唯一索引号，操作系统可以通过这个索引号快速找到文件，比如这里的 place_posts.php，我们可以查看它的 inode 值：

```
s-colin:/data/www/htdocs # ls -li place_posts.php
475922 -rwxrwxrwx 1 daemon daemon 2564 2009-03-04 19:06 place_posts.php
```

为了更好地监控 opcode 内存缓存的状态，我们完全可以利用 Cacti 等监控系统来定时获取这些信息，描绘出需要的曲线图。后面我们会有关于性能监控的具体探讨。

另外，缓存过期检查也是缓存机制中的一个重要部分，在默认情况下，缓存了 opcode 的动态程序在每次请求时，都需要检查程序是否有所变化，如果发现程序自上次访问后被修改，则重新编译 opcode。

APC 提供了一种跳过过期检查的机制，如果动态程序长期不会变化，那么可以跳过过期检查以获得更好的性能表现。要跳过过期检查，可以修改以下配置：

```
apc.stat = off
```

需要注意的是，在这种情况下，如果程序代码发生了修改，则必须通过重启 Web 服务器来使其生效。

到现在为止，我们已经非常清楚地认识到，使用 opcode 缓存机制可以大大减少动态内容的处理时间，这也意味着减少了一定的 CPU 和内存开销。脚本解释器通常运行在 Web 服

务器的进程中（如 Apache-prefork 模型的子进程），或者以 fastcgi 进程的形式独立运行。

我们接下来在 Apache 上对 PHP 动态网页进行压力测试，并通过 nmon 统计每秒的 CPU 使用率和内存剩余量。在没有使用 APC opcode cache 的时候，结果如图 5-2 及图 5-3 所示。

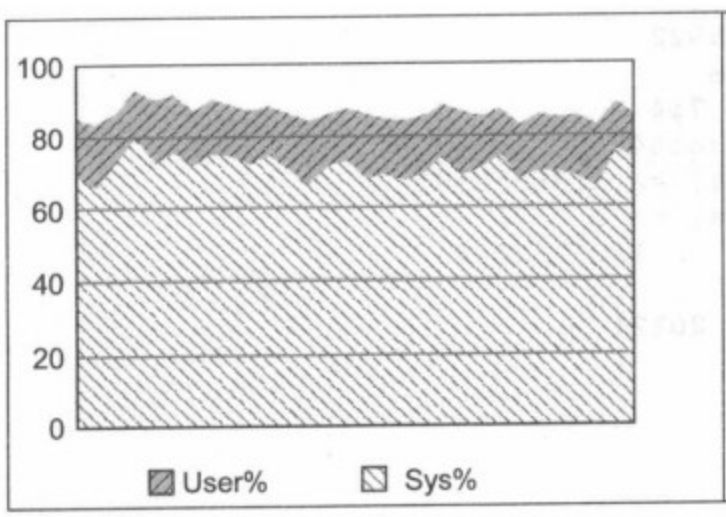


图 5-2　没有使用 APC opcode cache 时的 CPU 使用率

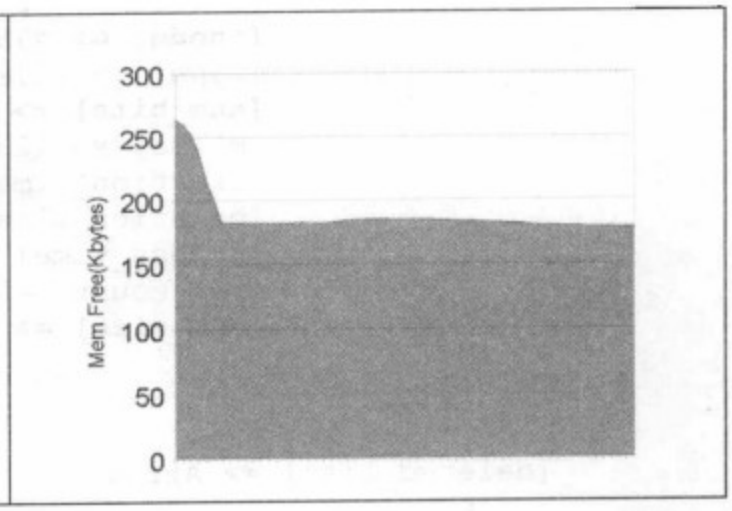


图 5-3　没有使用 APC opcode cache 时的内存剩余量

接下来，我们打开 APC opcode cache，进行同样的压力测试，结果如图 5-4 及图 5-5 所示。

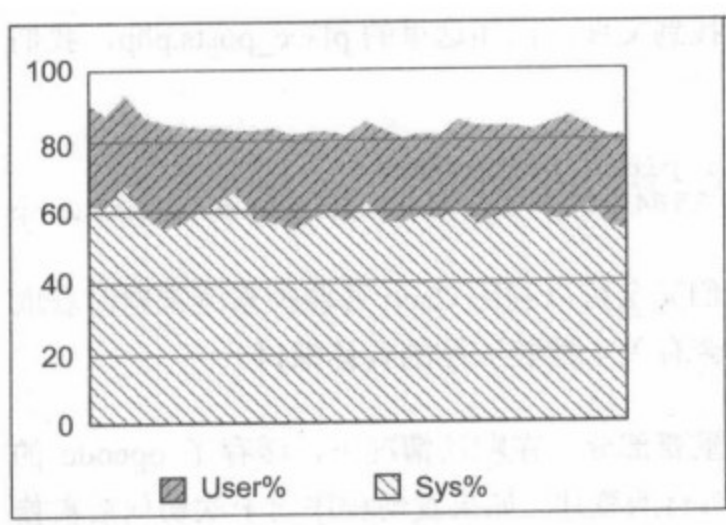


图 5-4　使用 APC opcode cache 时的 CPU 使用率

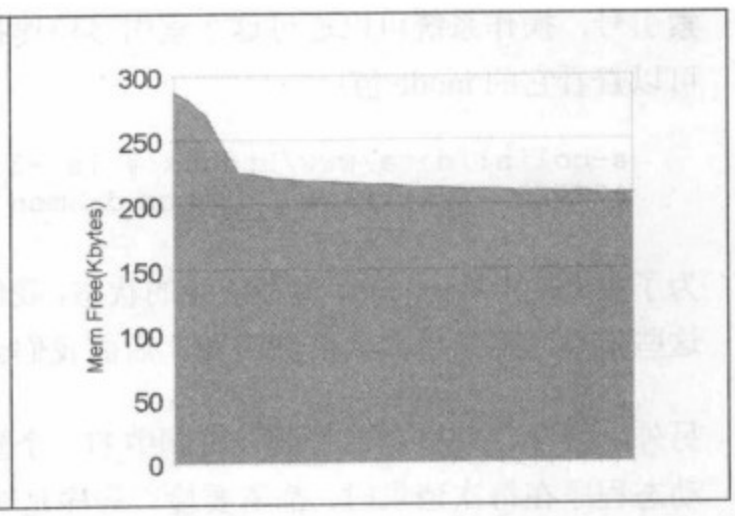


图 5-5　使用 APC opcode cache 时的内存剩余量

通过对比我们很容易发现，在没有使用 opcode cache 之前，用户态 CPU 使用率较高，这主要花在了解释器生成 opcode，而内核态 CPU 使用时间相应被挤占，同时内存的剩余量也较小；当使用了 opcode cache 之后，用户态 CPU 使用率明显降低，更多的时间用于内核态 CPU 的使用，比如发起 I/O 操作的系统调用，同时内存的剩余量也有所提高。

当然，你也许会发现，并不是所有的动态内容都在应用了 opcode cache 之后有大幅度的性能提升，因为 opcode cache 的目的是减少 CPU 和内存开销，如果动态内容的性能瓶颈不在于 CPU 和内存，而在于 I/O 操作，比如数据库查询带来的磁盘 I/O 开销，那么 opcode cache

的性能提升是非常有限的。从前面的对比数据中可以看到，place_posts.php 在没有使用任何动态缓存时，每次请求都需要多次数据库访问，在没有使用 opcode cache 时，它的吞吐率是 51.59 reqs/s，而使用了 opcode cache 后，吞吐率只增加到了 92.96 reqs/s，与应用了动态缓存后 opcode cache 的吞吐率提升幅度相比，它显得无足轻重。

XCache

我们再来看看另一款 opcode 缓存器——XCache。之前我们曾经用过它的内存缓存方法，但是在这里，它的 opcode cache 也将派上用场，我们同样对几种动态网页缓存方法进行了测试，和之前没有 opcode 缓存的结果对比如表 5-3 所示。

表 5-3　使用 XCache 前后的吞吐率比较

	无脚本加速	XCache opcode cache
不使用缓存	51.59	94.87
Smarty cache	173.95	1130.35
file cache	461.08	2048.39
XCache cache	462.05	2381.69
memcache cache	388.62	1498.08

对比图如图 5-6 所示。

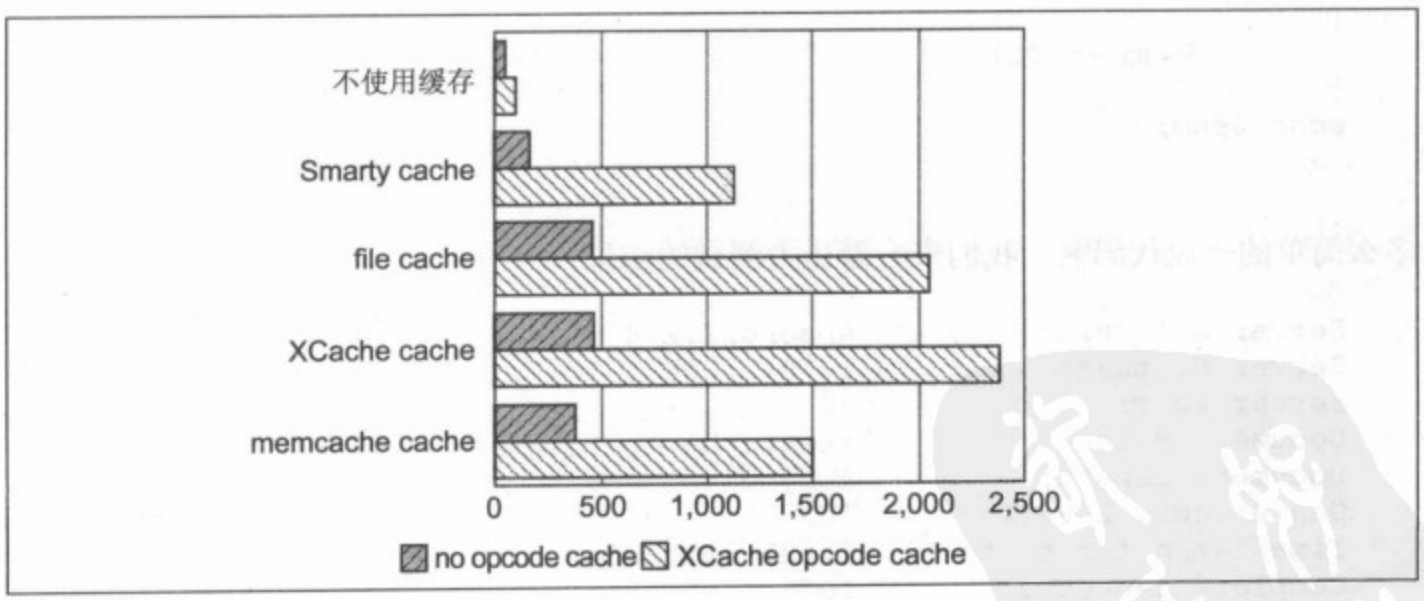


图 5-6　使用 XCache opcode cache 前后的压力测试对比图

可以看出，APC 和 XCache 不相上下，这里需要指出，从数值上看，它们的差距不是绝对的，实际中选择 APC，还是 XCache，或者 eAccelerator，并没有定论，它们都在不断地成长和完善，只要了解了 opcode 的本质以及性能测试的方法，我想如何选择的问题你完全可以自己搞定。

5.2 解释器扩展模块

主流的动态语言解释器都是比较开放的，它们深知单靠一己之力无法成就大业，所以充分支持第三方扩展，但是，我们也需要小心谨慎，大多数扩展模块只关心技术可行性，而不关心性能，对于它们，没有太多绝对的评价，你需要做的是充分考虑扩展模块可能对性能带来的副作用。

举个例子，对于一些 PHP 开发者来说，要想直接在 Web 应用程序中引用 Java 类库，就得在 PHP 中加载 Java 扩展模块，如下所示：

```
    extension=/usr/local/php/lib/php/extensions/no-debug-non-zts-2006061
3/java.so
    [java]
    java.java_home="/usr/local/jdk"
    java.java="/usr/local/jdk/bin/java"
    java.log_file="/var/log/php-java-bridge.log"
    java.classpath="/usr/local/php/lib/php/extensions/no-debug-non-zts-2
0060613/JavaBridge.jar"
    java.libpath="/usr/local/index/liblucene/lucene.jar"
    java.log_level="2"
```

在加载了 Java 扩展模块后，我们这里并不打算执行包含 Java 代码的应用程序，而只是编写了一个简单的 PHP 程序，代码如下所示：

```
    <?php
    $sum = 0;
    for ($i = 0; $i < 100; ++$i)
    {
            $sum += $i;
    }
    echo $sum;
    ?>
```

多么简单的一段代码啊，我们来看看压力测试的结果：

```
    Server Software:        Apache/2.2.4
    Server Hostname:        10.0.1.200
    Server Port:            80
    Document Path:          /test.php
    Document Length:        4 bytes
    Concurrency Level:      100
    Time taken for tests:   2.620 seconds
    Complete requests:      1000
    Failed requests:        0
    Write errors:           0
    Total transferred:      191000 bytes
    HTML transferred:       8000 bytes
    Requests per second:    381.71 [#/sec] (mean)
    Time per request:       261.982 [ms] (mean)
    Time per request:       2.620 [ms] (mean, across all concurrent requests)
    Transfer rate:          71.20 [Kbytes/sec] received
```

吞吐率居然只有381.71reqs/s，我们去掉Java扩展模块后，再次进行测试，结果如下所示：

```
Server Software:          Apache/2.2.4
Server Hostname:          10.0.1.200
Server Port:              80
Document Path:            /test.php
Document Length:          4 bytes
Concurrency Level:        100
Time taken for tests:     0.103 seconds
Complete requests:        1000
Failed requests:          0
Write errors:             0
Total transferred:        191000 bytes
HTML transferred:         8000 bytes
Requests per second:      9707.98 [#/sec] (mean)
Time per request:         10.301 [ms] (mean)
Time per request:         0.103 [ms] (mean, across all concurrent requests)
Transfer rate:            1810.77 [Kbytes/sec] received
```

我想你已经知道刚才吞吐率下降的原因了，没错，PHP解释器每次都需要初始化Java解释器，这带来了巨大的开销，不过，如果这样做可以解决你的迫切需求，那无可厚非，或许性能对你来说是其次的，但是，我不理解为什么Java扩展模块要设计得如此匪夷所思，即便是PHP代码中并没有实际使用Java扩展，仍然会初始化Java解释器，这样做不是很合理。当然，这里并没有排斥Java本身的意思，我们指的是在动态脚本中加载第三方扩展的性能副作用，而Java扩展只是一个例子。

5.3 脚本跟踪与分析

有了opcode缓存，动态内容的处理时间明显减少，事实上我们所做的仅仅是安装opcode缓存器而已，似乎你只付出了一点点的努力，却得到了巨大的回报，我们对这种性能上的大幅度提升感到惊喜无比，并备受鼓舞。

然而，我们也知道，动态内容在计算过程中，还有很多开销是无法通过opcode缓存来避免的，如何让脚本执行得更快呢？至少我们需要知道时间主要消耗在哪些代码上，这样才可以进一步分析，究竟这些开销来自于脚本程序本身，还是其他外部的原因。

为此，我们需要了解如何测量脚本程序中各处代码的执行时间，为此，代码跟踪是少不了的，我们必须掌握脚本的高级调试技巧，几乎所有脚本语言都内置了或多或少的调式方法，但是也许它们并不专业，而且不够完整。

Xdebug是一个PHP的PECL扩展，它提供了一组用于代码跟踪和调试的API。下面我们主要以Xdebug为例，介绍代码跟踪的一些方法。

代码段执行时间

当我们需要计算任意一段代码的执行时间时，一般会在开始处获得时间，然后在结束处再次获得时间，计算出时间差并打印到屏幕或者日志中。Xdebug 提供了更加高效的替代方法，你可以在程序中的任何位置调用 xdebug_time_index()方法，它将返回从脚本开始处执行到该位置所花费的时间。比如：

Example:

```
<?php
echo xdebug_time_index(), "\n";
for ($i = 0; $i < 1000000; $i++)
{
    // do nothing
}
echo xdebug_time_index(), "\n";
?>
```

Returns:

```
0.00038003921508789
0.98548336126585
```

上下文信息收集

另外，为了配合代码跟踪收集信息，我们往往需要在调式信息中记录当前的上下文信息，比如当前的行号、在哪里被调用等。PHP 内置了一套预定义常量，比如__LINE__记录了当前的代码行号，但是，它们的行为非常愚蠢，我想你和我一样都曾经对它们感到失望。比如以下这段 PHP 代码，我希望在函数 add()中获得它被调用的上下文信息，可是结果却并不是我想要的，无论 add()函数在哪里被调用，结果总是一样的。

Example:

```
<?php
function add($n1, $n2)
{
        echo __LINE__ . ":" . __FUNCTION__ . ":" . ($n1 + $n2);
}
function do_add()
{
        add(1, 2);
}
do_add();
?>
```

Returns:

```
4:add:3
```

与此相比，Xdebug 要更加懂得我的心思，它提供了类似的几个方法，但不是预定义常量的形式。我们看以下修改后的代码，它可以聪明地告诉我，add()函数是在 do_add()函数中被调用，并且被调用的行号是 9，完全正确。

Example:

```
<?php
function add($n1, $n2)
{
        echo xdebug_call_line() . ":" . xdebug_call_function() . ":" .
($n1 + $n2);
}
function do_add()
{
        add(1, 2);
}
do_add();
?>
```

Returns:

```
9:do_add:3
```

代码覆盖范围

Xdebug 还提供了一个非常不错的跟踪功能，它可以告诉你，代码在执行过程中“走”过了哪些行，这对于逻辑和分支比较复杂的程序来说，的确是一个不错的功能，我们来看看下面的代码，当然，这只是一个简单的例子，我们根据结果数据将代码中执行的语句加粗表示。

Example:

```
<?php
xdebug_start_code_coverage();
function run($n)
{
        if ($n < 10)
        {
                $n = 0;
        }
        else
        {
                if ($n > 20)
                {
                        $n = $n + 1;
                }
                else
                {
                        $n = $n * 10;
                }
```

```
        }
}
run(15);
var_dump(xdebug_get_code_coverage());
?>
```

Returns:

```
array(1) {
  ["/data/www/htdocs/__coverage.php"]=> array(9)
  {
    [3]=> int(1)
    [5]=> int(1)
    [6]=> int(1)
    [11]=> int(1)
    [12]=> int(1)
    [17]=> int(1)
    [20]=> int(1)
    [22]=> int(1)
    [23]=> int(1)
  }
}
```

另外，Xdebug 还提供了强大的栈跟踪功能，它可以在程序发生异常的时候，收集当前的上下文数据，类似 Core dump 帮助我们调试程序，这里不做具体介绍，它对于我们的性能分析意义不大。

函数跟踪

Xdebug 的另一个重要跟踪功能便是函数跟踪，它可以根据程序在实际运行时的执行顺序，跟踪记录所有函数的执行时间，以及函数调用时的上下文，包括实际参数和返回值。没错，这听起来正是我们迫切需要的。

正好，我们拿出前面 APC opcode cache 优化前后的动态内容，来跟踪一下各部分代码执行的时间，虽然前面我们已经通过压力测试看到了它们的性能差异，但是这次我们完全深入它们的代码内部来看个究竟。也许之前你对压力测试结果充满怀疑，那么你更不能错过这里。顺便提一下，如果说前面的压力测试是黑盒测试，那么现在就是名副其实的白盒测试了。

值得一提的是，Xdebug 提供的函数跟踪功能非常强大，一旦你开启这个选项，它便可以在动态内容执行的过程中，自动跟踪并将记录的数据保存在你指定的目录下。根据你的需要，可以设置各种级别的记录模式，它们分别提供了不同类型的记录格式，比如可以记录每个函数的实际参数内容，但如果你觉得没有必要，也可以只记录这些参数的数据类型和长度，我们采用了后者。另外，它还可以记录每个函数所在的文件名，为了突出重点，我们暂时屏蔽了文件名的记录。

在开始跟踪之前，你还需要在 php.ini 中设置记录文件的存储目录和文件名前缀，尽量根据自己的环境来做好规划，这将有助于你长期维护不同历史阶段的记录文件，它们也许在若干时间后还能派得上用场。

```
xdebug.trace_output_dir = /tmp/xdebug
xdebug.trace_output_name = trace.%c
```

好，一切就绪后，我们开启跟踪模式，首先访问那个没有任何页面缓存和 opcode 缓存的动态网页，当时我们压力测试的结果是 51.59 reqs/s。我们从 Xdebug 的记录文件中截取了一些片段，如下所示，内容很容易理解，左边第一列的时间代表从脚本开始执行到此处的时间长度，是不断累积的，单位是秒。我们来看第一段：

```
0.0052     338556          -> PlacePosts->run()
0.0052     338556            -> PlacePosts->getParams()
0.0053     338576              -> __request(string(9), ???)
                               >=> '12882'
                             >=> NULL
0.0054     338824            -> PlacePosts->validateParams()
                             >=> TRUE
0.0054     338824            -> PlacePosts->initVariable()
                             >=> NULL
0.0055     338824            -> PageBase->initSmarty()
0.0115     758764              -> require(/data/www/Smarty.class.php)
```

请注意粗体部分的时间跨度，很显然，Smarty 初始化消耗的时间明显比其他代码的执行时间多出很多。我们继续往下看：

```
0.0167    1009988    -> MarkerInfo->getMarkerInfo()
0.0167    1009988      -> DataAccess->selectDb(string(11))
0.0168    1010040        -> DataAccess->connect()
0.0168    1010040          -> microtime_float()
0.0168    1010088            -> microtime()
                             >=> '0.92977000 1236425466'
0.0169    1010048            -> explode(string(1), string(21))
                             >=> array (0 => '0.92977000', 1 => '1236425466')
                           >=> 1236425466.9298
0.0170    1010288          ->mysql_connect(string(9), string 4), string(0))
                           >=> resource(14) of type (mysql link)
0.0207    1011320          -> microtime_float()
0.0207    1011320            -> microtime()
                             >=> '0.93364100 1236425466'
0.0207    1011256            -> explode(string(1), string(21))
                             >=> array (0 => '0.93364100', 1 => '1236425466')
                           >=> 1236425466.9336
```

以上片段要说明的是 mysql_connect()消耗的等待时间，的确，建立 TCP 连接相比于前后的代码要消耗更多的时间，如果 MySQL 部署在不同的网络，那么这部分时间可能还要更长。下面的代码片段涉及数据库查询：

```
0.0350    1057772    -> explode(string(1), string(21))
```

```
                    >=> array (0 => '0.94787600', 1 => '1236425466')
                  >=> 1236425466.9479
0.0351    1057708  -> mysql_query(string(239), resource(14) of type
                     (mysql link))
                 >=> resource(27) of type (mysql result)
0.0406    1057708 -> microtime_float()
0.0407    1057708    -> microtime()
                     >=> '0.95360000 1236425466'
0.0407    1057644    -> explode(string(1), string(21))
```

注意上面的 mysql_query()，你一定对它毫不陌生，这里它消耗了大约 0.0055s（5.5ms）的时间，还记得 5.5ms 意味着什么吗？

现在，我们对这个动态网页开启 APC opcode cache，但并不使用任何页面动态缓存，再次访问后，我们在记录文件中找出几处片段来和前面比较一下：

```
0.0011      90612       -> PlacePosts->run()
0.0011      90612         -> PlacePosts->getParams()
0.0011      90632           -> __request(string(9), ???)
                            >=> '12882'
                          >=> NULL
0.0012      90880         -> PlacePosts->validateParams()
                          >=> TRUE
0.0012      90880         -> PlacePosts->initVariable()
                          >=> NULL
0.0013      90880         -> PageBase->initSmarty()
0.0015     126432           -> require(/data/www/Smarty.class.php)
```

有了 opcode 缓存后，Smarty 的初始化时间大大减少了，只需要 0.2ms。再来看下面的数据库查询，别忘了我们只是使用了 opcode cache，所以它还是需要访问数据库获取内容。

```
0.0174      212748              -> explode(string(1), string(21))
                                >=> array (0 => '0.71082200', 1 =>
                                '1236425806')
                              >=> 1236425806.7108
0.0175      212684             -> mysql_query(string(239), resource(9)
                               of type (mysql link))
                              >=> resource(22) of type (mysql result)
0.0231      212684            -> microtime_float()
0.0231      212684              -> microtime()
                                >=> '0.71661200 1236425806'
0.0232      212620              -> explode(string(1), string(21))
```

快看，mysql_query()的等待时间为 5.6ms，和前面的时间几乎相同，也就是说数据访问的时间仍然不变，我想你早就知道会有这个结果了，因为通过前面的介绍你已经了解 opcode cache 的目的。

我们再来试试使用了 memcache 页面缓存后的动态网页，我们通过预先访问已经建立了 HTML 缓存，这将使得它在随后不会访问数据库。在未使用 opcode cache 时，我们从记录

的开头处截取了一个片段，如下所示：

```
    0.0006      84936   -> {main}()
    0.0019     170404     -> require(/data/www/Common.page.php)
    0.0024     214396       -> require_once(/data/www/Toolkit.class.php)
                             >=> 1
                           >=> 1
    0.0033     279812     -> require(/data/www/page_common.php)
    0.0036     295280       -> require(/data/www/Base.page.php)
    0.0042     337684         -> require_once(/data/www/Log4php.class.php)
                               >=> 1
                             >=> 1
                           >=> 1
    0.0043     343036     -> pageCreator(string(10))
    0.0044     344024       -> PlacePosts->__construct()
    0.0044     344072         -> PageCommon->__construct()
    0.0044     344120           -> PageBase->__construct()
    0.0044     344856             -> log4php->log4php()
```

注意看以上时间的累积和变化。然后我们开启 opcode cache，预先访问一次驱使它建立 opcode 缓存，然后再次访问，从跟踪记录中截取开头那块同样的片段：

```
    0.0002      47852   -> {main}()
    0.0003      48832     -> require(/data/www/libcommon/Common.page.php)
    0.0004      66188       -> require_once(/data/www/libcommon/Toolkit.
class.php)
                              >=> 1
                            >=> 1
    0.0005      71836     -> require(/data/www/htdocs/page_common.php)
    0.0006      76084       -> require(/data/www/libcommon/Base.page.php)
    0.0007      80544         -> require_once(/data/www/libcommon/Log4php.
class.php)
                              >=> 1
                            >=> 1
                          >=> 1
    0.0008      88472     -> pageCreator(string(10))
    0.0009      89484       -> PlacePosts->__construct()
    0.0009      89572         -> PageCommon->__construct()
    0.0009      89660           -> PageBase->__construct()
    0.0009      90396             -> log4php->log4php()
```

不难发现，在两次的跟踪记录中，后者的执行时间整体上有了很大的压缩，这些挤掉的时间正是生成 opcode 的时间。

以上我们应用 Xdebug 的函数跟踪，通过一系列的记录片段，来分析代码执行中的时间消耗，分析结果和我们的预期完全相符。然而，这里我们的目的并不在于探讨如何对以上特定的问题领域本身进行分析，而是希望通过这个实例，介绍一种可行的代码跟踪方法，事实上我们的目的已经达到了，它让我们进入了一个全新的世界，至于如何用它来解决站点中的实际问题，我想你完全可以做得更好。

瓶颈分析

通过前面介绍的各种代码跟踪方法，我们已经能够走进代码内部进行近距离的观察，查看每处代码的执行时间，这意味着我们可以精确地评估程序中任意部分的性能。但是，光有这些还不够，我们还希望对这些跟踪记录进行进一步的统计和分析，以便快速找出程序的瓶颈所在。

Xdebug 同样为我们提供了性能跟踪器（Profiler），它的工作方式类似于函数跟踪，也是在脚本程序运行的时候自动将性能记录文件写入我们指定的目录中，我们可以设置如下：

```
xdebug.profiler_output_dir = /tmp/xdebug
xdebug.profiler_output_name = cachegrind.out.%p
```

后边的“%p”是运行时 PHP 解释器所在进程的 PID。

幸运的是，我们可以使用图形界面的分析工具来分析这些性能记录文件，Linux KDE 下可以使用 KCacheGrind，Windows 下可以使用 WinCacheGrind，它们可以直接打开性能记录文件，我们这里用 WinCacheGrind 打开前面访问动态网页时的性能记录文件，界面如图 5-7 所示。

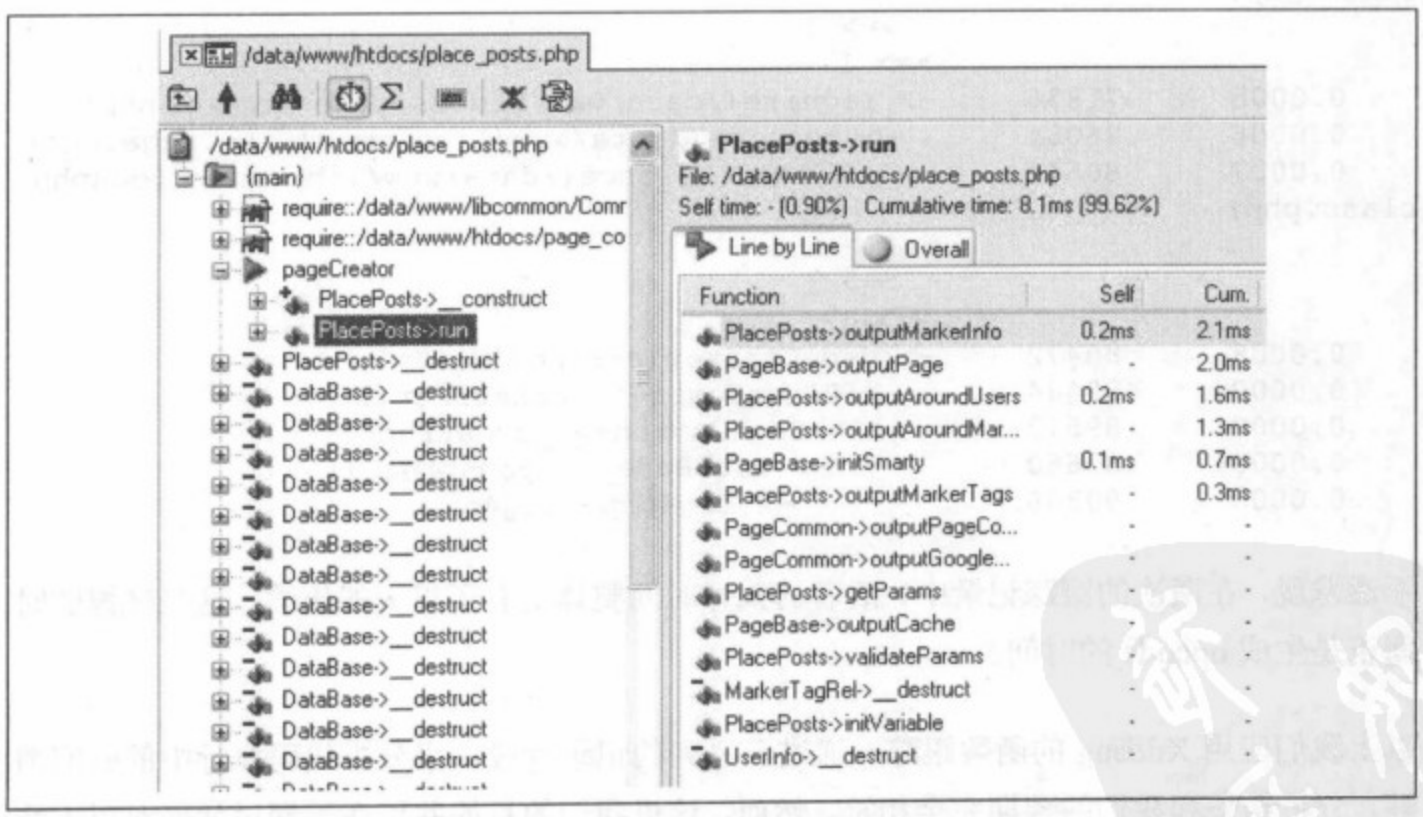


图 5-7　用 WinCacheGrind 查看各成员方法的执行时间

从这里我们可以查看各个范围的函数调用时间，并且它按照从大到小的顺序进行排列，比如这里我们左边选中了 PlacePosts->run，所以右边的 Line by Line 标签中将会显示 PlacePosts 对象在 run()方法中的所有方法调用的时间，可以看出最高的是 PlacePosts

->outputMarkerInfo，如果这里双击 PlacePosts->outputMarkerInfo，深入查看内部的函数调用，便可以看到，它的内部含有数据库访问，所以消耗了较多的时间。

对于图 5-7，我们还可以迅速切换为显示各个方法调用的时间百分比，如图 5-8 所示。

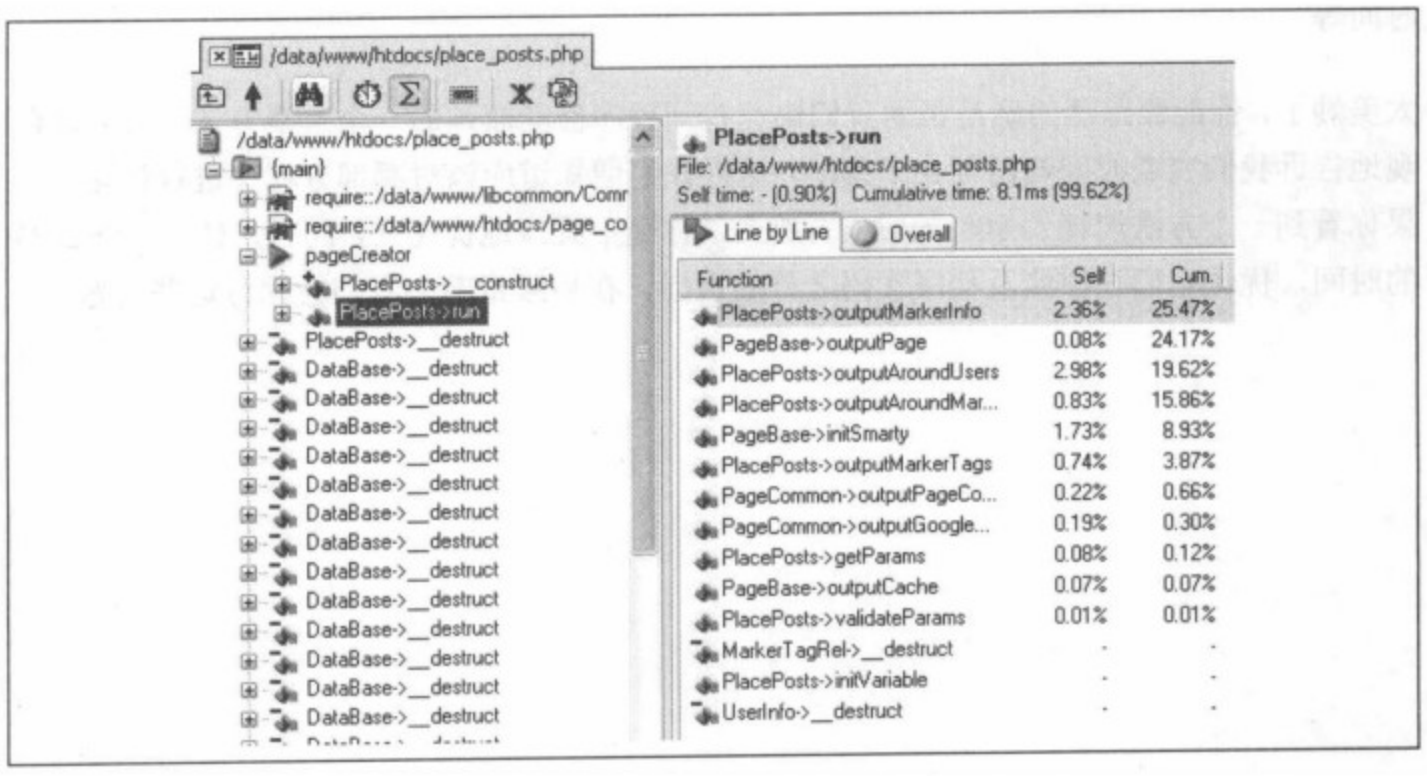


图 5-8　用 WinCacheGrind 查看各成员方法的执行时间百分比

现在我们来看看 Overall 标签，它对整个程序中所有的调用进行了归纳整理和排序，如图 5-9 所示。

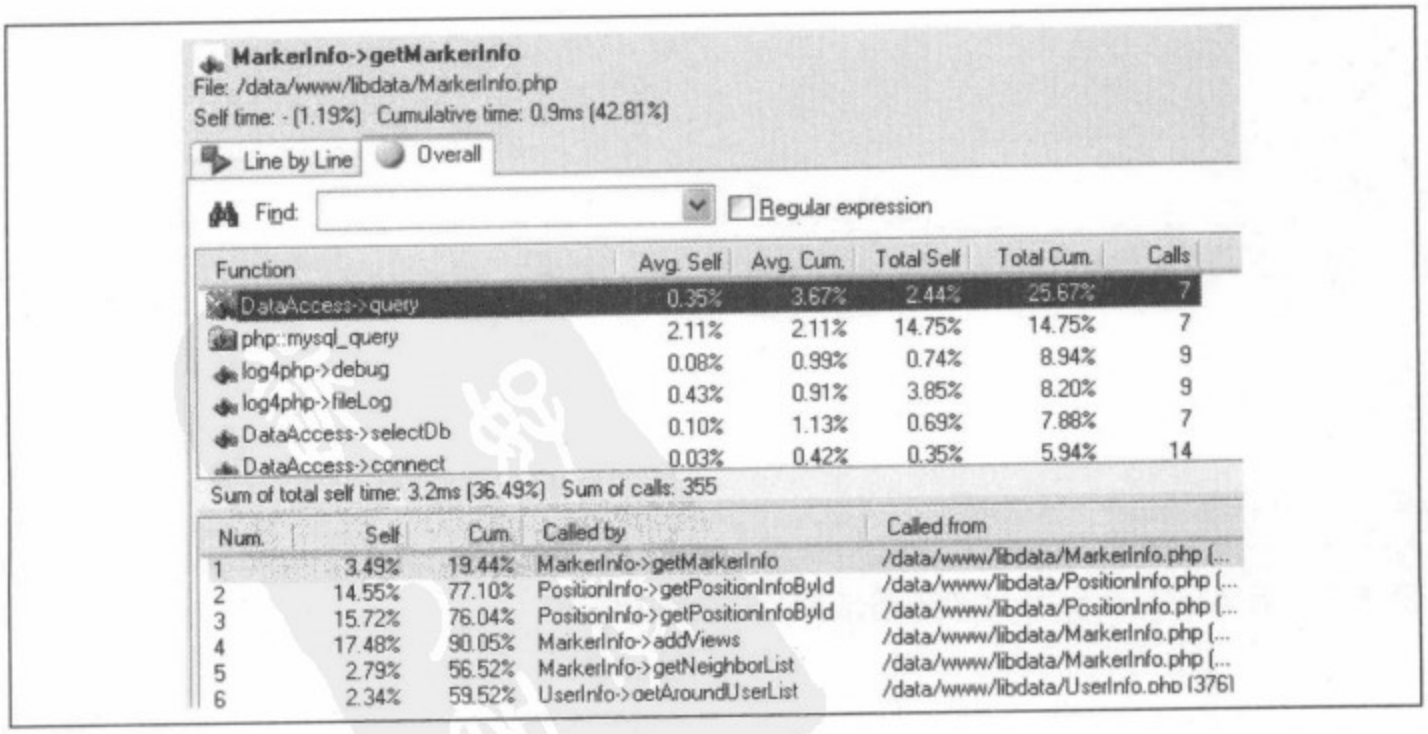


图 5-9　用 WinCacheGrind 查看总体执行时间百分比

你也许已经看到了，第一行的 DataAccess->query 和第二行的 php::mysql_query 表现突出，实际上这里所有的 php::mysql_query 都是在 DataAccess->query 中被调用，因为我们这里把所有数据访问方法都封装在了 DataAccess 类中。注意，当我们用鼠标选中 DataAccess->query 时，底部的列表便显示出了它都在哪些地方被调用，以及调用的次数和时间等。

太美妙了，性能跟踪器的确是帮助我们快速找到程序性能瓶颈的一个重要方法，它可以直观地告诉我们究竟时间都消耗在了哪里，这样我们便知道应该对哪部分程序进行优化。如果你看到一个方法消耗了 80%的时间，那么请你毫不犹豫地优化它，同时，对于另外 20%的时间，优化它们往往涉及程序代码之外的世界，在后续章节中我们会探讨这些问题。

第 6 章

浏览器缓存

嘿，还记得我们不久前讨论过的动态内容缓存吗？当时我们尝试将这些缓存放在不同的位置，包括磁盘、内存，甚至放入基于 TCP 连接的独立缓存服务器上，我想它们对性能的提升或许给你留下了深刻的印象。

也许你还在琢磨将这些缓存放在服务器的其他什么位置，也许是一个访问速度更快的设备，但我不知道比内存更快的存储设备还有什么，至少在我的服务器上是不存在的。

事实上，即便是将缓存放在服务器磁盘或内存上，如果是由 Web 服务器本身来实现缓存机制，并管理缓存，或许可以获得更好的性能表现，这在下一章中我们会详细介绍。

但是在这里，我们也许忽略了另一个地方，那就是浏览器。的确，我们需要暂时从服务器上移开视线，花点时间了解浏览器的工作方式，虽然你可能已经认识它很多年了，但是你真的了解它吗？

6.1 别忘了浏览器

实际上，当我们通过浏览器打开一个 Web 页面的时候，浏览器需要从 Web 服务器下载一系列的内容，包括 HTML 网页、图片、脚本、样式表等，并且将它们存放在浏览器的缓存中。

一旦 Web 服务器上的一部分内容存在于用户浏览器缓存中，那么至少对于这个用户而言，如果请求同样的内容，便可能不需要再次从服务器获取，直接从浏览器缓存中拿来即可。一个典型的例子是，当我在线试听 MP3 音乐时，第一次会等待一定时间的下载缓冲，而随后再次试听的时候，便可以迅速开始播放，这实际上便是直接读取缓存在本地的 MP3 文件。

浏览器不只是用户的

一直以来，似乎没有人把用户的浏览器看成是 Web 站点的组成部分，而传统意义上它也的确不是，人们都知道浏览器是用户安装在 PC 上的软件，而 Web 站点所拥有的只是数据中心的服务器。但是，如果你换一个角度，把浏览器想象成为 Web 站点分派到千家万户

的缓存管理器，那么从现在开始，你不能再对浏览器坐视不理了，的确，成千上万的浏览器就像在为你免费打工，如果你懂得有效利用它们的话。

听起来非常不错，这似乎告诉我们应该把握一个原则，尽可能地让 Web 站点的内容缓存在用户浏览器中，这样将在一定程度上减少了服务器的计算开销，而且也避免了有些内容由于不必要的重复传输而带来的带宽浪费，出于环保主义的考虑，我们都应该特别重视这一点，根据能量守恒定律，从服务器发送的数据越多，消耗的能量越大，散热和辐射都在随之增加，而我们仅仅需要将内容缓存在浏览器就可以改善这一现状，一起行动起来吧。

缓存放在哪里

浏览器一般会在用户的文件系统中创建一个目录，用于存放缓存文件，并给每个缓存文件打上一些必要的标记，比如过期时间等。不同的浏览器采用不同的方式来存储缓存，了解这些内容将有助于我们准确地使用浏览器缓存。

IE 浏览器在用户本地设置了临时文件目录，你可以在 IE 的缓存设置中找到对应的路径，如图 6-1 所示。打开目录后，可以看到里面的文件都是我们在访问一些站点时留下的，你可以直接打开它们浏览缓存的内容，如图 6-2 所示。

对于 Firefox 浏览器，它存储缓存文件的方式有所不同。它并不像 IE 那样将每个文件独立存储，而是采用二进制文件的方式来存储和管理缓存文件，比如图 6-3 中 Firefox 浏览器缓存目录中的文件。

显然，这种二进制存储方式不便于我们了解每个 URL 缓存文件的状态细节，所以 Firefox 浏览器提供了一种简单的方式来帮助我们查看所有缓存内容，只需要在浏览器地址栏输入 about:cache 即可，如图 6-4 所示。

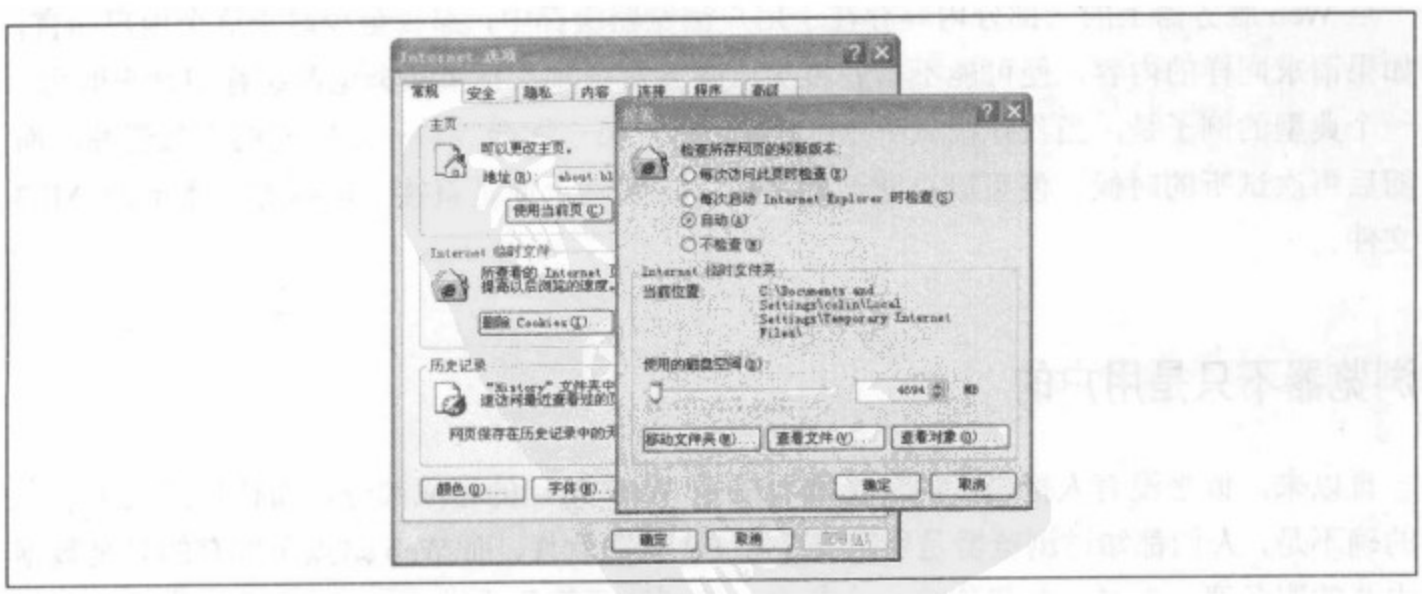


图 6-1　IE 浏览器查看临时文件目录路径

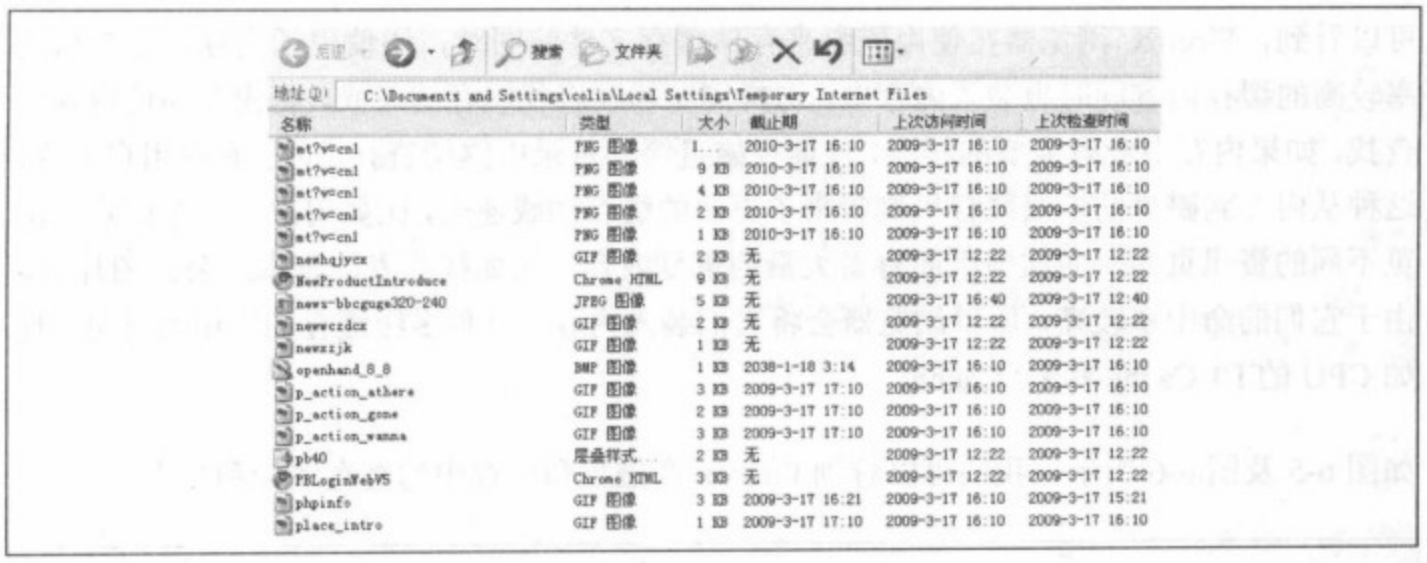


图 6-2　IE 浏览器临时文件目录中的缓存文件

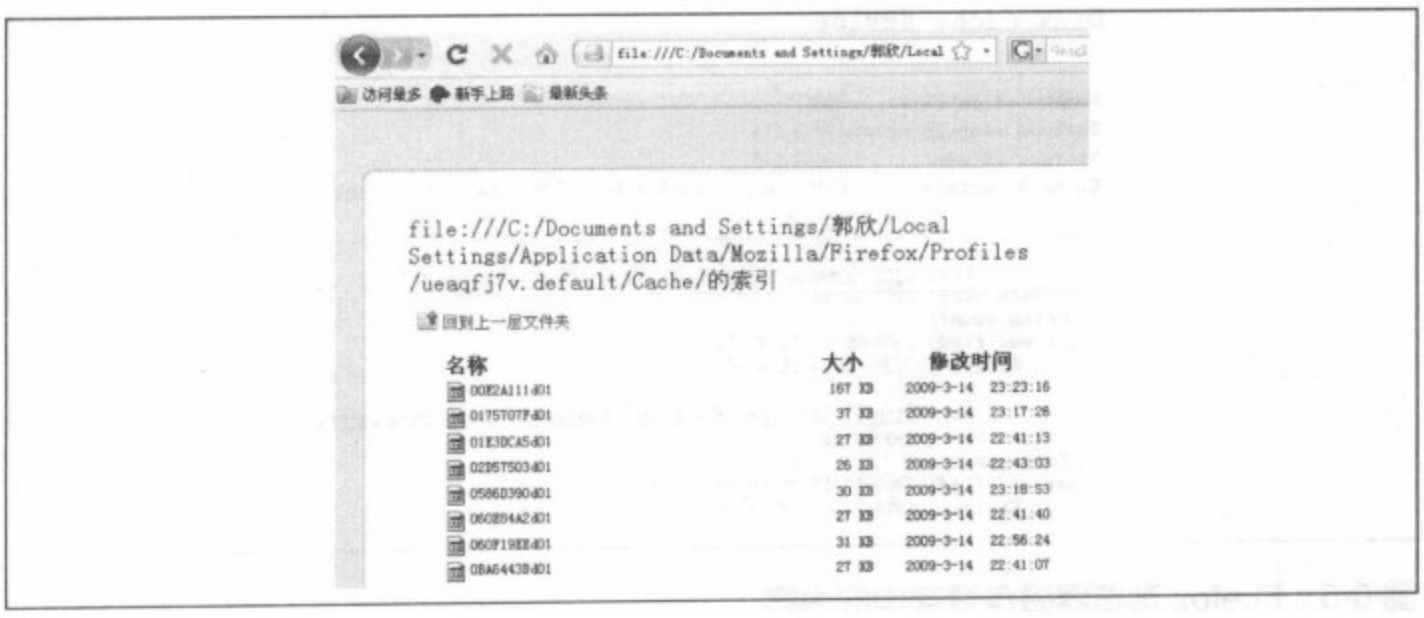


图 6-3　Firefox 浏览器缓存目录中的文件

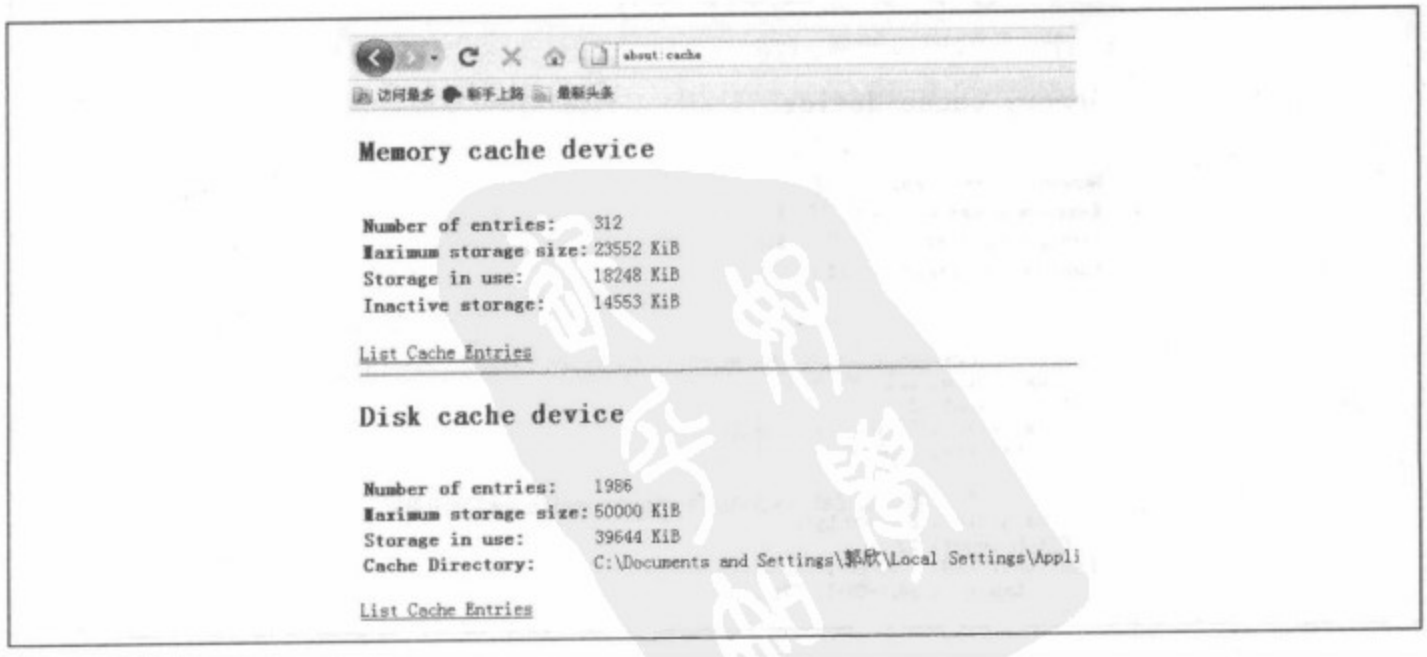


图 6-4　查看 Firefox 浏览器的缓存概述

可以看到，Firefox 浏览器在使用磁盘来存储缓存文件的同时，还使用了内存，它将命中率较高的缓存内容同时也装入内存中，这样浏览器在查找缓存的时候，将先在高速内存中查找，如果内存中没有需要的缓存，便前往磁盘缓存目录中继续查找。对于有些用户来说，这种从内存到磁盘的多级缓存机制带来了更快的缓存加载速度，比如用户在一个站点下浏览不同的资讯页面，而这些页面有着大量的重复内容，比如样式表、脚本、公共图片等，由于它们的命中率较高，所以浏览器会将它们装入内存。这种多级缓存的应用很常见，比如 CPU 的 L1 Cache 和 L2 Cache。

如图 6-5 及图 6-6 所示，我们可以看到 Firefox 在磁盘和内存中的缓存统计和列表。

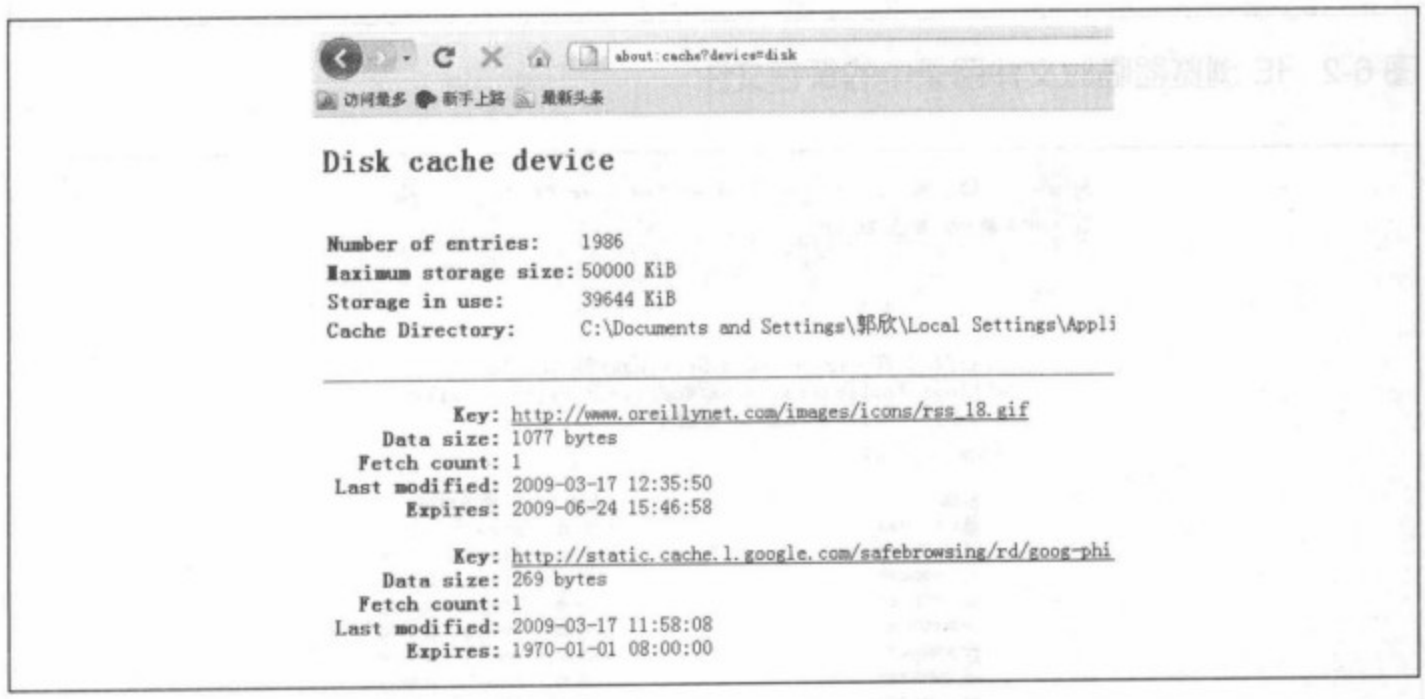


图 6-5　Firefox 浏览器磁盘缓存中的内容

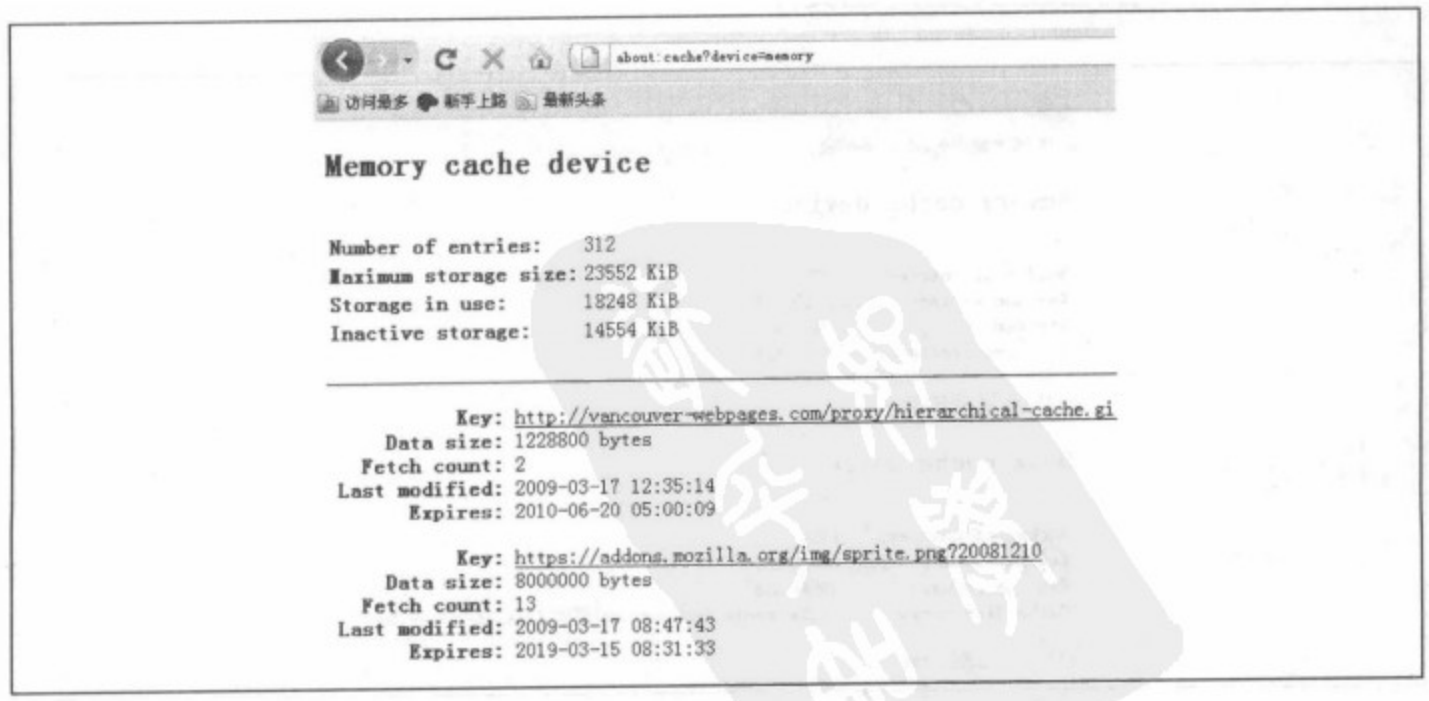


图 6-6　Firefox 浏览器内存缓存中的内容

看到 IE 和 Firefox 的这些缓存内容后，你也许注意到它们的过期时间、上次修改时间、上次检查时间等，既然是缓存，那么过期检查肯定是少不了的，那么浏览器是如何完成的呢？仔细一想便知道，浏览器和 Web 服务器进行沟通的唯一途径便是 HTTP，那么如果浏览器希望了解本地的内容是否过期，当然也得通过 HTTP 和服务器进行沟通。可见，有效地使用浏览器缓存，其本质在于对 HTTP 协议缓存部分的深入理解，下面的内容围绕这部分内容展开，同时这也是后面探讨 Web 服务器缓存和反向代理缓存的必备知识。

6.2 缓存协商

我们都知道，服务器端的动态内容缓存包括缓存内容的创建、存储，以及过期检查等一系列过程，它们都可以由诸如 PHP 这样的服务器端动态程序来实现。然而，对于浏览器缓存，就完全不是那么回事了，缓存内容存储在浏览器本地，而内容由 Web 服务器生成，任何一方都不可能独立完成这一系列过程，所以它们之间必须有一种沟通机制，这就是 HTTP 中的“缓存协商”。

如何协商

协商的过程很容易理解，首先，当浏览器向 Web 服务器请求一些内容时，Web 服务器需要告诉浏览器哪些内容可以被缓存，一旦浏览器知道某个内容可以缓存后，下次当浏览器需要请求这个内容时，它便不会直接向服务器请求完整内容，而是询问服务器是否可以使用本地的缓存，服务器在收到浏览器的询问后需要作出果断的回应，到底是允许浏览器使用本地缓存还是将最新的内容传回浏览器。

的确，就这么简单，不过，以上的描述也许会让你感到有点乏味，我们还是通过几个例子来更加直观地说明这一过程。

做好协商的准备

下面我们通过一个 PHP 动态网页来模拟一些缓存协商，之所以使用动态程序，是因为它可以根据我们的意愿直接添加必要的 HTTP 头信息，我们可以从 HTTP 的角度来更加直观地了解这些过程。需要说明的是，动态内容本身并不受浏览器缓存机制的排斥，因为浏览器和 Web 服务器通过 HTTP 通信，它们并不关心这些内容是否在服务器上动态生成，所以只要 HTTP 头信息中包含相应的缓存协商信息，动态内容一样可以被浏览器缓存，它和静态内容没什么两样。另外需要补充一点，刚才所说的都是基于 GET 类型请求的情况，而对于 POST 类型的请求，浏览器一般不启用本地缓存，我们这里不探讨 POST 类型请求的缓存，因为这么做显然很不明智。

我们给这个 PHP 程序取名为 http_cache.php，它的功能非常简单，用于显示当前 UNIX Timestamp 时间，仅此而已，代码如下所示：

```
<?php
echo time();
?>
```

接下来，我们通过 IE 浏览器来请求这个动态网页，同时，我们使用 HttpWatch 这样的浏览器插件来跟踪 HTTP 请求和响应。

> **提示：**
>
> 从另一个角度看，IE 浏览器只是一个实现了标准 HTTP 的用户代理（User Agent），所以我们随后的操作在任何基于标准 HTTP 的浏览器上都有效。

当我在浏览器上第一次输入这个动态网页对应的 URL 时，浏览器发出以下的 HTTP 请求：

```
GET /http_cache.php HTTP/1.1
Accept: */*
Accept-Language: zh-cn
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; GTB5;
CIBA; .NET CLR 2.0.50727)
Host: www.highperfweb.com
Connection: Keep-Alive
```

针对以上的 HTTP 请求，Web 服务器马上作出了以下响应：

```
HTTP/1.1 200 OK
Date: Fri, 20 Mar 2009 03:41:59 GMT
Server: Apache/2.2.11 (Unix) PHP/5.2.1 DAV/2 SVN/1.4.3
X-Powered-By: PHP/5.2.1
Content-Length: 10
Keep-Alive: timeout=30, max=96
Connection: Keep-Alive
Content-Type: text/html
1237522549
```

可以看到 HTTP 响应的状态码为 200，这意味着 Web 服务器将请求的结果内容全部返回给浏览器，长度为 10 个字节。

对于上面这一组请求和响应我们并不陌生，它们协作得非常成功。然而我们关注的是重复请求时的情况，我们连续刷新了好几次浏览器，注意，是单击浏览器的刷新按钮，或者按 F5 键，可以从图 6-7 的 HttpWatch 跟踪画面中看到，每次的请求都是获得状态码为 200 的响应，接收的数据量为 248 个字节，这是 HTTP 响应头信息和正文信息的总长度。

也就是说，浏览器在随后的几次请求并没有使用本地的缓存，也没有向服务器作出任何表示。我们打开 IE 的临时文件目录，可以看到刚才请求的 http_cache.php 缓存文件，如图

6-8 所示。

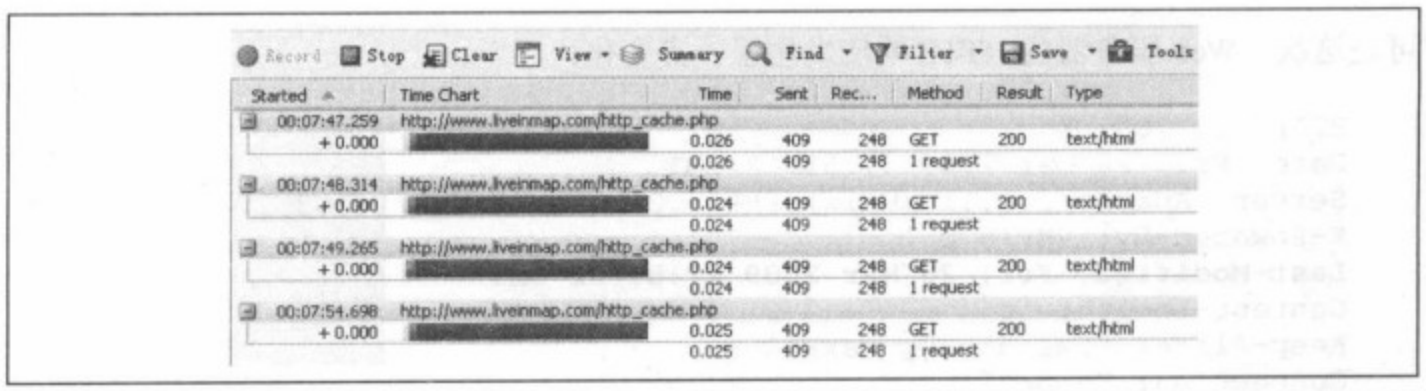


图 6-7　未使用浏览器缓存时的 HTTP 跟踪

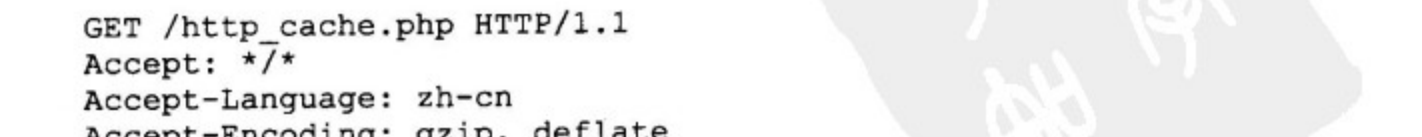


图 6-8　未使用浏览器缓存的临时文件

为什么浏览器没有考虑使用本地缓存呢？答案在图 6-8 中就可以找到，注意看这个缓存文件的“上次修改时间”为“无”，可见浏览器并不知道这个内容在服务器上的生成时间或最后修改时间，所以没有过期检查的依据，自然就无法使用缓存了。

Last-Modified

事实上动态内容一般不存在传统意义上的最后修改时间，静态文件可以通过 stat()系统调用获得它在物理文件系统中的最后修改时间，一般 Web 服务器会为静态文件的 HTTP 响应头自动生成最后修改时间，这在后续章节中会有介绍。

这时，我们的动态程序就要发挥本领了，我们来为 HTTP 响应增加最后修改时间的标记，修改代码后如下所示：

```
<?php
header("Last-Modified: " . gmdate("D, d M Y H:i:s") . " GMT");
echo time();
?>
```

这意味着我们为动态内容增加了一些 HTTP 响应头信息。我们刷新浏览器，再次请求 http_cache. php，浏览器同样发出了我们熟悉的请求：

```
GET /http_cache.php HTTP/1.1
Accept: */*
Accept-Language: zh-cn
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; GTB5;
CIBA; .NET CLR 2.0.50727)
```

```
Host: www.highperfweb.com
Connection: Keep-Alive
```

可是这次，Web 服务器的 HTTP 响应发生了一点变化：

```
HTTP/1.1 200 OK
Date: Fri, 20 Mar 2009 07:53:02 GMT
Server: Apache/2.2.11 (Unix) PHP/5.2.1 DAV/2 SVN/1.4.3
X-Powered-By: PHP/5.2.1
Last-Modified: Fri, 20 Mar 2009 07:53:02 GMT
Content-Length: 10
Keep-Alive: timeout=30, max=100
Connection: Keep-Alive
Content-Type: text/html
1237535582
```

你一定看出来了，HTTP 响应头中多出了以下的标记：

```
Last-Modified: Fri, 20 Mar 2009 07:53:02 GMT
```

这正是我们修改 PHP 程序后的结果，它代表了 Web 服务器对浏览器的暗示，告诉它当前请求内容的最后修改时间为“Fri, 20 Mar 2009 07:53:02 GMT”。现在，在 IE 浏览器的临时文件目录中，可以看到这个内容已经存在上次修改时间，需要注意的是，HTTP 协议中规定使用 GMT 时间，也就是格林威治标准时间，而我们国家使用的是 GMT+8 时区，所以在 HTTP 头信息中的时间会比我们的正常时间早 8 个小时，图 6-9 中可以看到 Windows 中的时间比 HTTP 头信息中的时间确实晚了 8 个小时，但它一点都不影响 HTTP 缓存的正常工作。

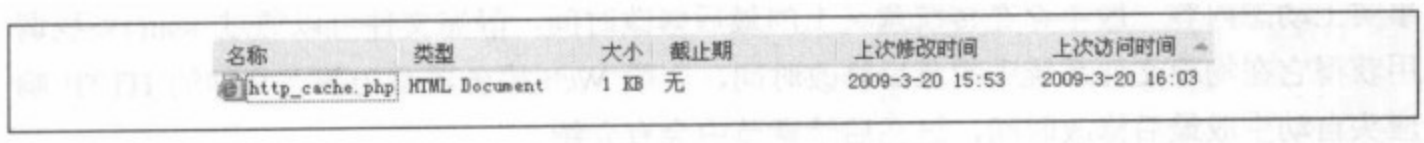


图 6-9　使用浏览器缓存的临时文件

处理浏览器的询问

收到 Web 服务器的暗示后，我想浏览器一定会有所感悟，我们再次刷新浏览器，它发出了以下的 HTTP 请求：

```
GET /http_cache.php HTTP/1.1
Accept: */*
Accept-Language: zh-cn
Accept-Encoding: gzip, deflate
If-Modified-Since: Fri, 20 Mar 2009 07:53:02 GMT
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; GTB5;
CIBA; .NET CLR 2.0.50727)
Host: www.highperfweb.com
Connection: Keep-Alive
```

果然，浏览器不再无动于衷，它在 HTTP 请求头中增加了以下这段标记：

```
If-Modified-Since: Fri, 20 Mar 2009 07:53:02 GMT
```

这意味着浏览器在询问 Web 服务器："我请求的内容在这个时间之后是否有更新？"此时浏览器肩负起了重要责任，它需要检查这个内容在该时间后是否有过更新，并反馈给浏览器，这一过程便相当于我们传统意义上的缓存过期检查，对于静态内容来说，Web 服务器可以轻松搞定，只要获得静态文件的最后修改时间并将其和浏览器询问的时间进行对比即可，但是对于动态内容，Web 服务器可做不了主，这部分工作需要动态程序自己来完成，然后告诉 Web 服务器。我们暂且停下来，看看同样内容的一个静态文件如何响应这样的询问，我们在 Web 服务器上创建另一个静态网页，名为 http_cache.htm，它的内容也是一串长度为 10 个字节的数字。

我们请求这个静态网页，浏览器发出以下请求：

```
GET /http_cache.htm HTTP/1.1
Accept: */*
Accept-Language: zh-cn
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; GTB5;
CIBA; .NET CLR 2.0.50727)
Host: www.highperfweb.com
Connection: Keep-Alive
```

和请求刚才的动态网页似乎没什么本质的区别，再来看看服务器的响应：

```
HTTP/1.1 200 OK
Date: Fri, 20 Mar 2009 04:13:07 GMT
Server: Apache/2.2.11 (Unix) PHP/5.2.1 DAV/2 SVN/1.4.3
Last-Modified: Fri, 20 Mar 2009 08:12:39 GMT
Accept-Ranges: bytes
Content-Length: 10
Keep-Alive: timeout=30, max=97
Connection: Keep-Alive
Content-Type: text/html
1237522060
```

以上 HTTP 响应头中的最后修改时间是 Apache 为它自动生成的。接下来，我们再次刷新浏览器，请求如下：

```
GET /http_cache.htm HTTP/1.1
Accept: */*
Accept-Language: zh-cn
Accept-Encoding: gzip, deflate
If-Modified-Since: Fri, 20 Mar 2009 08:12:39 GMT
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; GTB5;
CIBA; .NET CLR 2.0.50727)
Host: www.highperfweb.com
Connection: Keep-Alive
```

跟刚才的情况一样，请求中增加了对服务器询问的内容。此时，Web 服务器的响应如下所示：

```
HTTP/1.1 304 Not Modified
Date: Fri, 20 Mar 2009 04:13:11 GMT
Server: Apache/2.2.11 (Unix) PHP/5.2.1 DAV/2 SVN/1.4.3
Connection: Keep-Alive
Keep-Alive: timeout=30, max=96
```

这里我们需要关注的是响应状态码的变化，304 Not Modified 意味着 Web 服务器告诉浏览器，这个内容没有更新，浏览器可以使用本地缓存的内容。同时，Web 服务器没有将内容的正文传送给浏览器。

我们连续刷新几次，通过 HttpWatch 跟踪 HTTP 请求，从图 6-10 中可以看到，除了第一次请求的响应状态码为 200，随后请求的响应状态码都为 304。

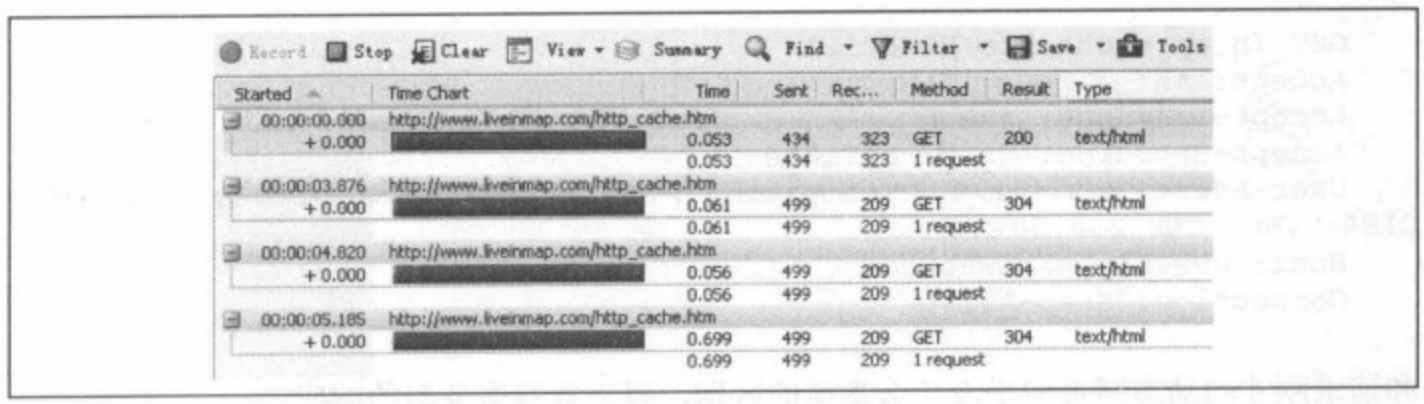


图 6-10　使用浏览器缓存时的 HTTP 跟踪

另一种协商方法

HTTP/1.1 还支持另一种缓存协商方法，那就是 ETag，它与前面所讲的协商方式非常类似，但它没有采用内容的最后修改时间，而是采用了一串编码来标记内容，称为 ETag。一个原则是，如果一个内容的 ETag 没有变化，那么这个内容也一定没有更新。

ETag 由 Web 服务器来生成，比如 Apache 为一个静态文件的 HTTP 响应头增加了以下标记：

```
ETag: "74177-b-46585209c1bc0"
```

浏览器获得这个内容的 ETag 后，便会在下次请求该内容时，在 HTTP 请求头中附加以下标记来询问服务器该内容是否发生变化：

```
If-None-Match: "74177-b-46585209c1bc0"
```

这时服务器就需要重新计算这个内容的 ETag 值，并与 HTTP 请求中的 ETag 进行比较，如果相同的话，便返回 304 状态码，如果不同的话，则将最新的内容返回给浏览器。

HTTP/1.1 并没有规定 ETag 的具体格式和计算方法，也就是说，Web 服务器可以自由定义 ETag 的格式和计算方法，比如一种简单的方法是对文件内容计算 md5 值作为 ETag，总之只要能够起到标识内容的作用即可，比如 Lighttpd 为静态文件自动生成的 ETag 如下所示：

```
ETag: "1944822255"
```

使用基于最后修改时间的缓存协商存在一些缺点，比如，有时候一些文件需要频繁的更新，但是内容可能并没有变化，那么如果采用基于最后修改时间的缓存协商，那么每次文件的修改时间变化后，不论内容是否真的变化，浏览器都会重新获取全部内容；再比如，同一个文件存储在多台 Web 服务器上，用户的请求在这些服务器之间轮询，实现负载均衡，而这些服务器上同一个文件的最后修改时间很难保证完全相同，这便会导致用户的请求每次切换到新的服务器时就需要重新获取所有内容。这时候，如果使用直接标记内容的某种 ETag 算法，就可以避免这些问题。

让动态内容学会和浏览器交流

到现在，我们对缓存协商有了比较深刻的理解，回过头来，还记得刚才的动态程序 http_cache.php 吗？在它告诉浏览器最后修改时间 Last-Modified 后，浏览器在随后对它的请求中增加了 If-Modified-Since 缓存过期询问，我想我们应该尊重浏览器的声音，这对我们不是一件坏事。我们来修改 PHP 程序的代码，让它可以正确答复浏览器的询问，代码如下所示：

```
<?php
$modified_time = $_SERVER['HTTP_IF_MODIFIED_SINCE'];
if (strtotime($modified_time) + 3600 > time())
{
        header("HTTP/1.1 304");
        exit(1);
}
header("Last-Modified: " . gmdate("D, d M Y H:i:s") . " GMT");
echo time();
?>
```

你一定看出来了，改造后的程序首先获得了 HTTP 请求中的 If-Modified-Since 内容，然后将它和当前的服务器时间比较，如果距离当前时间不到一个小时，那么我们将返回 304 状态码，并终止程序，这时候我们相信浏览器会将本地的缓存取出并交给用户，它不会骗我们的。

性能如何

接下来，我们将缓存询问机制应用在实际的动态内容或静态内容中，来看看它们的表现。还记得在前面章节中提到的 place_posts.php 吗？我们找来当时使用了 Smarty 内置缓存的

代码，希望给它增添对于浏览器缓存的支持，有了上面对 http_cache.php 的改造经验，我们很容易就写出了以下的代码：

```
$modified_time = $_SERVER['HTTP_IF_MODIFIED_SINCE'];
if (strtotime($modified_time) + $this->smarty->cache_lifetime > time())
{
        header("HTTP/1.1 304");
        exit(1);
}
header("Last-Modified: " . gmdate("D, d M Y H:i:s") . " GMT");
require '../libsmarty/Smarty.class.php';
$this->smarty = new Smarty();
$this->smarty->caching = true;
$this->template_page = "place_posts.htm";
$this->cache_id = $this->marker_id;
if ($this->smarty->is_cached($this->template_page, $this->cache_id))
{
    $this->smarty->display($this->template_page, $this->cache_id);
    exit(0);
}
do_some_db_query();
$this->smarty->display($this->template_page, $this->cache_id);
```

很简单，我们获得浏览器请求中的 If-Modified-Since 时间，并且根据事先设置的缓存有效期来判断是否过期，如果缓存有效的话，我们直接返回 304 状态码，并且终止程序，随后的 Smarty 初始化以及数据库操作都不会进行。

当然，要知道它的表现如何，我们同样拿出 ab 来进行压力测试，好在 ab 也可以模拟包含 If-Modified-Since 的 HTTP 请求。之前，我们曾经用 Lighttpd 对这个使用了 Smarty 内置缓存的动态程序进行过压力测试，现在你可以忘记当时的测试结果，这次我们使用 Apache 来作为 Web 服务器，在是否实现浏览器缓存的两种情况下进行测试对比。另外，需要说明的是，两次测试都开启了 XCache opcode cache。

首先，我们对没有实现浏览器缓存的版本进行测试，它使用了 Smarty 内置的缓存机制，测试结果如下所示：

```
Document Path:          /place_posts_cache_304.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   0.972 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      13716399 bytes
HTML transferred:       13481000 bytes
Requests per second:    1028.87 [#/sec] (mean)
Time per request:       97.194 [ms] (mean)
Time per request:       0.972 [ms] (mean, across all concurrent requests)
Transfer rate:          13781.59 [Kbytes/sec] received
```

1028.87 reqs/s 的吞吐率在我们的意料之中，下面我们对刚才实现了浏览器缓存的版本进行测试，这时候我们必须通过 ab 的选项来增加 HTTP 请求中的 If-Modified-Since 属性，如下所示：

```
ab -n1000 -c100 -H 'If-Modified-Since: Fri, 20 Mar 2009 23:02:25 GMT'
http://localhost/place_posts_cache_304.php?marker_id=12882
```

测试结果如下所示：

```
Document Path:          /place_posts_cache_304.php?marker_id=12882
Document Length:        0 bytes
Concurrency Level:      100
Time taken for tests:   0.426 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Non-2xx responses:      1021
Total transferred:      143961 bytes
HTML transferred:       0 bytes
Requests per second:    2347.14 [#/sec] (mean)
Time per request:       42.605 [ms] (mean)
Time per request:       0.426 [ms] (mean, across all concurrent requests)
Transfer rate:          329.98 [Kbytes/sec] received
```

很显然，吞吐率增加到刚才的 2 倍以上，你可能觉得这主要来源于后者避免了 HTTP 响应正文的传输开销，可以看到测试结果中 Document Length 指示的正文长度为 0 bytes，而前面测试结果指示的长度为 13000 bytes，但是，不要忘记这里的动态缓存是需要 PHP 程序来加载的，这部分开销也随之被节省，而事实上它才是以上吞吐率提升的主要因素。为了证明这一点，我们来做一个实验，改造这个动态程序，让它不理会浏览器的询问，而是继续通过 Smarty 内置的机制来检查并加载服务器上的缓存，但在最后不要输出这 13000 个字节，这样做的目的是避免网络传输开销，但同时保留服务器缓存加载的开销，测试结果如下所示：

```
Document Path:          /place_posts_cache_304.php?marker_id=12882
Document Length:        0 bytes
Concurrency Level:      100
Time taken for tests:   0.820 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      200400 bytes
HTML transferred:       0 bytes
Requests per second:    1220.05 [#/sec] (mean)
Time per request:       81.964 [ms] (mean)
Time per request:       0.820 [ms] (mean, across all concurrent requests)
Transfer rate:          238.77 [Kbytes/sec] received
```

我们可以看到，这次吞吐率的提升相对于刚才来说是微不足道的，这的确说明了，传输 13000 字节的网络 I/O 开销相比于动态程序检查和加载缓存的计算开销来说算不了什么，

况且我们这里还使用了 XCache opcode cache 对动态程序解释器进行了加速。

那么，如果动态程序检查和加载缓存的计算开销本身就很小的话，就不需要浏览器缓存了吗？比如我们之前使用 XCache 的内存缓存器来存储动态内容缓存，那样既不需要 Smarty 实例的计算开销，也避免了读取磁盘上缓存内容的 I/O 开销，我们再来对它进行压力测试，结果如下所示：

```
Document Path:          /place_posts_xcache.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   0.451 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      13536887 bytes
HTML transferred:       13351000 bytes
Requests per second:    2215.96 [#/sec] (mean)
Time per request:       45.127 [ms] (mean)
Time per request:       0.451 [ms] (mean, across all concurrent requests)
Transfer rate:          29294.12 [Kbytes/sec] received
```

的确，2215.96 reqs/s 的吞吐率几乎已经达到了前面使用浏览器缓存时的结果。下面我们对这个使用了 XCache 内存缓存器的程序增加浏览器缓存的实现，再来进行压力测试，结果如下所示：

```
Document Path:          /place_posts_xcache_304.php?marker_id=12882
Document Length:        0 bytes
Concurrency Level:      100
Time taken for tests:   0.419 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Non-2xx responses:      1000
Total transferred:      141000 bytes
HTML transferred:       0 bytes
Requests per second:    2384.69 [#/sec] (mean)
Time per request:       41.934 [ms] (mean)
Time per request:       0.419 [ms] (mean, across all concurrent requests)
Transfer rate:          328.36 [Kbytes/sec] received
```

可以看出，Document Length 减少为 0 bytes，但是吞吐率增加得并不多。

我们将刚才的几组测试结果归纳为图 6-11，首先，它们本身都使用了服务器端的动态内容缓存，前者使用 Smarty 内置缓存机制，将缓存内容存储在磁盘中，并通过 Smarty 提供的方法来检查和加载缓存，后者使用 XCache 内存缓存器来存储缓存内容，可以快速读写缓存区并且很容易进行过期检查。对于它们两者，是否使用了浏览器缓存的吞吐率差异可以从图 6-11 上很清楚地看到。

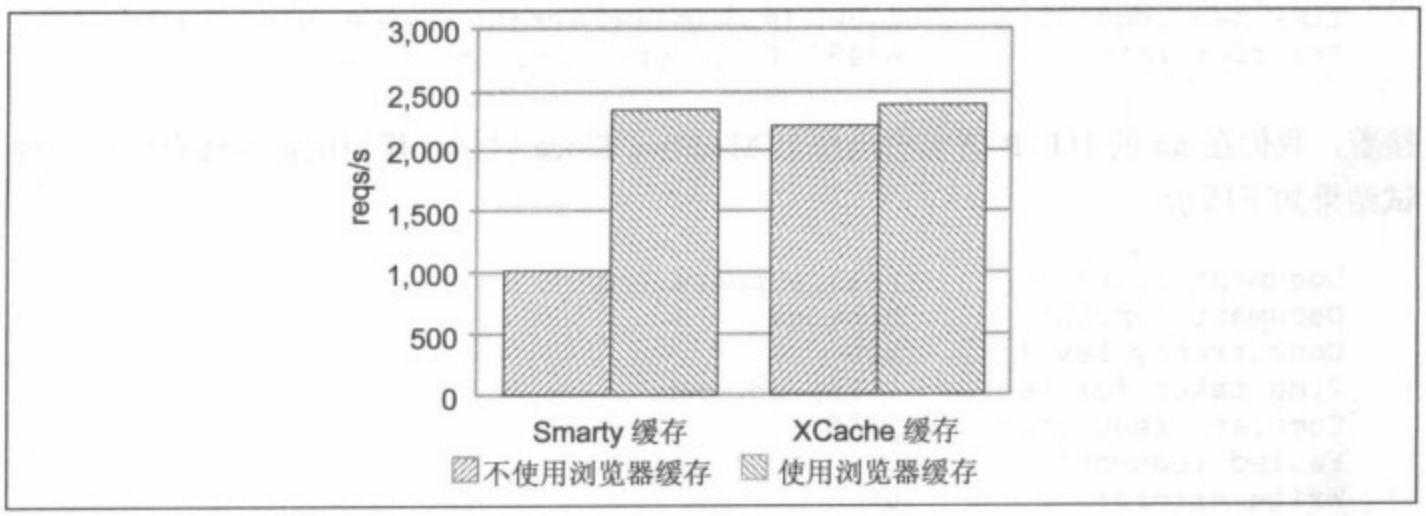


图 6-11　动态内容缓存增加浏览器缓存后的吞吐率对比

这也就是说，原本使用服务器端动态缓存的动态内容，在使用浏览器缓存后，能否获得较大的吞吐率提升，关键在于是否能够避免一些额外的计算开销。当然，对于本身没有使用任何服务器端缓存的动态内容，使用浏览器缓存必然会有较大的吞吐率提升，但是，几乎没有人这样做，因为有一点必须明白，浏览器缓存只是针对它所属的用户有效，而服务器端的缓存是针对所有用户有效，我想你的站点不会只有一个用户吧。同时，在另一种极端情况下，假如你的用户们在很长一段时间内对某个内容只请求 1 次，那么浏览器端缓存也几乎没有任何意义。

这么说，动态内容的浏览器缓存在有些时候就没有必要了吗？别急，吞吐率只是一个方面，现在请看刚才测试结果中的 Transfer rate，这是 Web 服务器返回给浏览器的数据传输率，我们拿 Smarty 缓存的这一组结果来看，在使用浏览器缓存后，带宽使用大大降低，数据传输率只有 329.98 KB/s，这些数据基本上只是响应 HTTP 头信息，换算成比特，也就是 2.64M 带宽，而之前的传输率为 13781.59 KB/s，也就是需要 110.25M 的带宽，这个差距使得浏览器缓存的意义更加深远，这意味着你只需要购买 2.64M 的独享带宽，就可以提供 2347.14 reqs/s 的吞吐率，而前者需要购买 110.25M 的独享带宽，却只能提供 1028.87 reqs/s 的吞吐率，这就是浏览器缓存带来的最大价值。

我们再来看看静态内容，同样是 13000 个字节的静态文件，我们先不使用浏览器缓存，对它进行压力测试，结果如下所示：

```
Document Path:          /place_posts.htm
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   0.208 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      13719273 bytes
HTML transferred:       13429000 bytes
Requests per second:    4811.00 [#/sec] (mean)
Time per request:       20.786 [ms] (mean)
```

```
Time per request:       0.208 [ms] (mean, across all concurrent requests)
Transfer rate:          64456.47 [Kbytes/sec] received
```

接着，我们在 ab 的 HTTP 请求中增加 If-Modified-Since 时间，模拟浏览器缓存协商，测试结果如下所示：

```
Document Path:          /place_posts.htm
Document Length:        0 bytes
Concurrency Level:      100
Time taken for tests:   0.189 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Non-2xx responses:      1018
Total transferred:      178150 bytes
HTML transferred:       0 bytes
Requests per second:    5291.65 [#/sec] (mean)
Time per request:       18.898 [ms] (mean)
Time per request:       0.189 [ms] (mean, across all concurrent requests)
Transfer rate:          920.61 [Kbytes/sec] received
```

我们对以上的结果进行对比，从两个方面去看，首先，吞吐率有了将近 10%的提升；其次，带宽使用降低了 98%，相当可观。

然而，浏览器缓存并非对吞吐率不能有较大的提升，这完全取决于 HTTP 响应正文的长度，比如一个 39MB 的视频文件 test.flv，在不使用浏览器缓存的情况下，测试结果如下所示：

```
Document Path:          /test.flv
Document Length:        39921980 bytes
Concurrency Level:      10
Time taken for tests:   6.206 seconds
Complete requests:      100
Failed requests:        0
Write errors:           0
Total transferred:      3992226900 bytes
HTML transferred:       3992198000 bytes
Requests per second:    16.11 [#/sec] (mean)
Time per request:       620.646 [ms] (mean)
Time per request:       62.065 [ms] (mean, across all concurrent requests)
Transfer rate:          628160.97 [Kbytes/sec] received
```

接下来，我们使用浏览器缓存，测试结果如下所示：

```
Document Path:          /test.flv
Document Length:        0 bytes
Concurrency Level:      10
Time taken for tests:   0.020 seconds
Complete requests:      100
Failed requests:        0
Write errors:           0
Non-2xx responses:      100
Total transferred:      17900 bytes
HTML transferred:       0 bytes
```

```
Requests per second:    4979.34 [#/sec] (mean)
Time per request:       2.008 [ms] (mean)
Time per request:       0.201 [ms] (mean, across all concurrent requests)
Transfer rate:          870.41 [Kbytes/sec] received
```

这次，不仅带宽使用率大大降低，而且吞吐率提升了 310 倍，这是非常巨大的提升。当然，这次的测试前提比较极端，静态文件足足有 39MB，而我们站点的网页以及其中的组件一般不会有这么大，但是在这背后，我们应该意识到浏览器缓存在何种场景下发挥何种作用。

SSI 和 Last-Modified

有一点需要注意的是，在使用 SSI（服务器端包含）的内容中，由于整个页面是服务器动态生成的，所以 Last-Modified 标记在不同的 Web 服务器中有不同的生成方法。

在 Apache 中，默认情况下请求开启 SSI 的内容，Apache 不会附加 Last-Modified 标记，这也就是说浏览器将无法进行缓存协商，你也许觉得有点遗憾，那么可以通过 Apache 配置中的 XBitHack 指令来启用 Last-Modified 标记，修改 Apache 配置如下所示：

```
XBitHack full
```

然后，你还需要将希望附加 Last_Modified 的文件追加组内执行权限，比如：

```
chmod g+x index.shtml
```

下面我们请求这个 index.shtml，跟踪 HTTP 响应头，如下所示：

```
HTTP/1.1 200 OK
Date: Tue, 24 Mar 2009 10:09:18 GMT
Server: Apache/2.2.11 (Unix) PHP/5.2.1 DAV/2 SVN/1.4.3
Last-Modified: Mon, 29 Dec 2008 02:11:00 GMT
Accept-Ranges: bytes
Keep-Alive: timeout=30, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html
```

终于看到了 Last-Modified 标记，但是它所指示的时间只是父页面本身的修改，我们可以看到 index.shtml 在文件系统中的最后修改时间，如下所示：

```
-rw-rwxrw-  1 daemon daemon  8472 2008-12-29 10:11 index.shtml*
```

所以，如果被包含的子页面发生内容更新，并且需要及时呈现的话，需要通过 touch 等方法更新父页面的修改时间，这样才可以让浏览器获得正确的缓存协商结果。

通过 XBitHack 的设置，Apache 对 SSI 文档增加了 Last-Modified 的支持，提高了浏览器缓存的利用率，但不幸的是，XBitHack 的开启为 Apache 带来了不小的开销，经过测试，开启 XBitHack 使得 Apache 处理 SSI 文档的吞吐率降低了 1/3 左右，所以，你需要作出选择。

幸运的是，针对这个问题，我们还有其他可选方案，在 Lighttpd 中就没有这么麻烦，Lighttpd 在开启 SSI 模式后，会自动将父页面中所有子页面的最后修改时间中最晚的一个作为 Last-Modified 时间，这样一来，只要有子页面更新，Lighttpd 便自动更新整个页面的 Last-Modified 时间。

另外，在某种设置下，这些 Web 服务器也可以使用 ETag 询问机制来检查 SSI 内容，如果你的站点提供大量的 SSI 内容，那么有必要对缓存询问机制做一些优化，比如你可以修改 Web 服务器源代码中 ETag 的生成机制，使得它能以最低的代价生成唯一标识 SSI 内容的 ETag 字符串。

6.3 彻底消灭请求

在 RFC2616 中对于 HTTP/1.1 的缓存部分有这样一段介绍：

```
    The goal of caching in HTTP/1.1 is to eliminate the need to send requests
in many cases.
```

是的，HTTP 缓存的目的就是要彻底消灭不必要的请求，也许你早就感到疑惑，之前的缓存协商为什么非要和 Web 服务器沟通呢？只要在浏览器缓存中标记出过期时间不就可以了吗？

浏览器缓存截止期

HTTP 中还有另一个标记，那就是 Expires，它告诉浏览器该内容在何时过期，暗示浏览器在该内容过期之前不需要再询问服务器，而直接使用本地缓存即可。

这样的好处显而易见，一旦浏览器丝毫不用请求服务器，那将完全节省了带宽和服务器处理等开销，可谓皆大欢喜。

Expires 标记更像善于放权的管理者，浏览器一旦看到某个内容附带 Expires 标记后，便拥有了极大的权力，它无须在过期之前每次都询问服务器，完全可以自作主张，而 Last-Modified 标记让浏览器感到拘束，它们不得不每次都询问服务器，即便它们认为这样做毫无意义。

Expires 的格式类似于 Last-Modified，它指示了内容过期的绝对时间，比如：

```
    Expires: Sun, 10 Feb 2002 16:00:00 GMT
```

对于静态内容，Web 服务器在默认情况下不会开启 Expires 标记的支持，我们需要进行一定的配置，比如在 Apache 中提供了 mod_expires 的支持，我们可以进行如下配置：

```
<IfModule mod_expires.c>
    ExpiresActive on
    ExpiresByType image/gif "access plus 1 month"
    ExpiresByType text/css "now plus 2 day"
    ExpiresDefault "now plus 1 day"
</IfModule>
```

而在 Lighttpd 中，一个简单的配置如下所示：

```
#### expire module
expire.url = ( "/" => "access plus 2 hours", "/images/" => "access plus
1 seconds 2 minutes")
```

可以看出，以上的两种配置方法略有不同，前者是针对静态内容的 MIME 类型来设置过期时间，而后者是针对文件路径来设置过期时间。另一方面，它们在过期时间的表达上拥有相似的语法，我们这里指定了相对时间，因为我们无法给静态文件直接指定一个绝对到期时间，所以采用“access plus”式的语法，由 Web 服务器在该内容被请求的时候动态计算一个绝对到期时间，作为 Expires 标记的内容。

值得一提的是，对于常见的静态文件格式，即便是 Web 服务器返回的 HTTP 响应头中没有 Expires 标记，浏览器也会根据一些其他的线索来猜测一个过期时间，比如 IE 在某种缓存模式下，对于 GIF 图片设置为永不过期，除非我们配置 Expires 为马上过期，也就是将过期时间设置为当前时间或者 0。浏览器这样做的目的在于尽量避免向 Web 服务器发送请求，除非 Web 服务器明确指出不允许它这样做。

对于动态内容，Expires 仍然需要程序自身来添加，类似于之前的 Last-Modified，随后我们会在动态程序中进行尝试。

如何请求页面

说到这里，有一个非常重要的常识你一定要知道，那就是如何使用浏览器请求页面，也许你认为这没什么技术含量，但是，当你了解了以下内容后或许会有不同的看法。对于主流浏览器，一般有以下三种请求页面的方法：

Ctrl + F5

> 这种方式可以叫强制刷新，它使得网页以及其中的所有组件都直接向 Web 服务器发送请求，并且不使用缓存协商，这样的目的是获取所有内容的最新版本。你也可以按住 Ctrl 键然后单击浏览器的刷新按钮获得同样的结果。在实际使用中，很少有用户会这样操作。

F5

这种方式便是一般的刷新，我们经常使用，它等同于单击浏览器的刷新按钮。它允许浏览器在请求中附加必要的缓存协商，但不允许浏览器直接使用本地缓存，也就是说，它能够让 Last-Modified 发挥效果，但是对 Expires 无效。

单击浏览器地址栏的“转到”按钮或者通过超链接跳到此页

我想这种方式大家使用最多，还有一种操作也等同于这种方式，那就是在浏览器地址栏中输入 URL 后按回车键，Firefox 中常用这种方式，因为它没有“转到”按钮。这几种方式允许浏览器以最少的请求来获取网页的数据，浏览器会对所有没有过期的内容直接使用本地缓存，所以，Expires 标记只对这种方式有效，以后你不用在按了 F5 键后对浏览器没有使用本地缓存感到奇怪。

随后我们会在实践中用到这三种方法，并对它们的效果进行直观的展示。

添加 Expires 标记

下面我们找来之前的动态程序 http_cache. PHP，将 Expires 标记添加到它的响应 HTTP 头中，修改后的代码如下所示：

```
<?php
$modified_time = $_SERVER['HTTP_IF_MODIFIED_SINCE'];
if (strtotime($modified_time) + 3600 > time())
{
        header("HTTP/1.1 304");
        exit(1);
}
header("Last-Modified: " . gmdate("D, d M Y H:i:s") . " GMT");
header("Expires: " .gmdate ("D, d M Y H:i:s", time() + 3600). " GMT");
echo time();
?>
```

这时候我们通过浏览器请求这个动态程序，跟踪到的 HTTP 响应头如下所示：

```
HTTP/1.1 200 OK
Date: Mon, 23 Mar 2009 08:32:47 GMT
Server: Apache/2.2.11 (Unix) PHP/5.2.1 DAV/2 SVN/1.4.3
X-Powered-By: PHP/5.2.1
Last-Modified: Mon, 23 Mar 2009 08:32:47 GMT
Expires: Mon, 23 Mar 2009 09:32:47 GMT
Content-Length: 10
Keep-Alive: timeout=30, max=99
Connection: Keep-Alive
Content-Type: text/html
```

看到了吗？Expires 已经出现在了 HTTP 响应头中，这也就意味着浏览器会将这个过期时间标记给这个内容，我们打开 IE 的临时文件目录，如图 6-12 所示，果然看到它的“截止期”是 Expires 中的时间，当然，这里有 8 个小时的时差，因为我们在 Windows 中使用的是 GMT+8 时区。

图 6-12　浏览器缓存文件的截止期为 Expires 绝对时间

对比三种请求方法

到现在，http_cache. PHP 已经完全支持了 Last-Modified 和 Expires，我们用刚才提到的三种刷新方法来试试，同时我们用 HttpWatch 跟踪了 HTTP 请求，在开始之前我们清空了浏览器的缓存区。结果如图 6-13 所示。

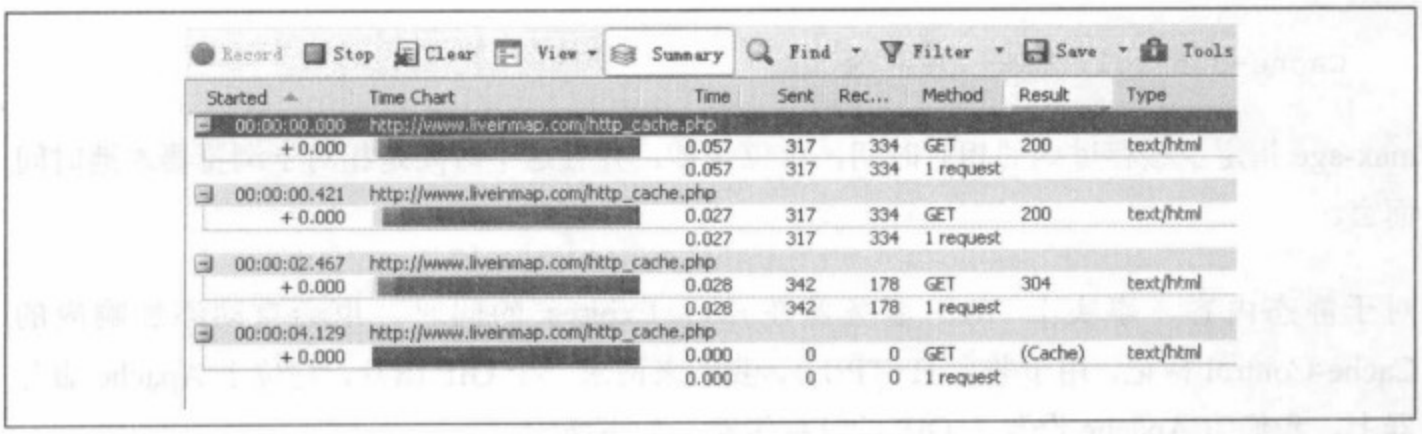


图 6-13　三种刷新方式的 HTTP 请求跟踪

第一次请求时，我们输入网址后按回车键，由于浏览器第一次请求这个内容，所以 Web 服务器返回所有正文，长度为 334 字节，包括 HTTP 响应头和正文，响应状态码为 200，时间消耗了 0.057s。

第二次请求我们采用了强制刷新，所以 Web 服务器仍然返回所有正文内容，总长度仍然为 334 字节，时间消耗了 0.027s。

第三次请求我们使用 F5 键普通刷新，这时候浏览器发出附带缓存询问的请求，Web 服务器响应状态码为 304，只返回了简单的 HTTP 响应头，长度为 178 字节，时间消耗了 0.028s。

第四次请求我们使用“转到”按钮，这时候可以看到跟踪画面中显示“(Cache)”，这代表了浏览器没有发送请求，同时可以看到 Sent 和 Recv 中都为 0，时间消耗也为 0。

适应本地的过期时间

到目前为止，还存在一个问题，通过 Expires 指定的过期时间，是来自于 Web 服务器的系统时间，如果用户本地的时间和服务器时间不一致的话，那一定会影响到本地缓存的有效期检查。

很容易想象，比如服务器端为某个内容设置的过期时间为 1 个小时，可是假如浏览器的时间比服务器晚了 2 个小时，那么这个内容将被浏览器认为立即过期。当然，一般我们使用的操作系统（如 Windows）都会使用基于 GMT 的标准时间，然后本地时间通过时区来进行偏移计算，而 HTTP 中使用的也是 GMT 时间，所以一般不会因为时区而导致本地和服务器相差数个小时。但是，没有人能保证用户本地的时间都是与你的服务器一致的，甚至有时候你的服务器时间也许就是错误的，这些都会影响到浏览器缓存的正常工作，让我们的一片苦心付诸东流。

幸运的是，HTTP/1.1 中还有一个标记用于弥补 Expires 的不足，那就是 Cache-Control，它的格式如下所示：

```
Cache-Control: max-age=<second>
```

max-age 指定了缓存过期的相对时间，单位是秒，并且这个时间是相对于浏览器本地时间而言。

对于静态内容，事实上 Web 服务器在开启 Expires 的同时，也会自动添加响应的 Cache-Control 标记，用于兼容 HTTP/1.1，我们来请求一个 GIF 图片，它位于 Apache 服务器上，我们为 Apache 设置了 GIF 的过期策略，如下所示：

```
ExpiresActive on
ExpiresByType image/gif "access plus 1 hours"
```

接下来我们在浏览器上请求这个图片的 URL，然后跟踪 HTTP 响应头，如下所示：

```
HTTP/1.1 200 OK
Date: Tue, 24 Mar 2009 04:51:03 GMT
Server: Apache/2.2.11 (Unix) PHP/5.2.1 DAV/2 SVN/1.4.3
Last-Modified: Wed, 27 Feb 2008 18:11:26 GMT
ETag: "7815c-303-44727bbbf0f80"
Accept-Ranges: bytes
Content-Length: 771
Cache-Control: max-age=3600
Expires: Tue, 24 Mar 2009 05:51:03 GMT
Keep-Alive: timeout=30, max=97
Connection: Keep-Alive
Content-Type: image/gif
```

可以从以上头信息中计算出，Expires 时间刚好是 Date 时间之后的 1 个小时，同时，max-age

的值为 3600 秒。

值得一提的是，目前的主流浏览器都将 HTTP/1.1 作为首选，所以当 HTTP 响应头中同时含有 Expires 和 Cache-Control 时，浏览器会优先考虑 Cache-Control。对于没有 Cache-Control 的情况，浏览器则会服从 Expires 的指示，之前我们的动态程序 http_cache.php 便只有 Expires，它工作得很好。

下面我们再次改造这个动态程序，增加 Cache-Control 标记，代码如下所示：

```
<?php
$modified_time = $_SERVER['HTTP_IF_MODIFIED_SINCE'];
if (strtotime($modified_time) + 3600 > time())
{
        header("HTTP/1.1 304");
        exit(1);
}
header("Last-Modified: " . gmdate("D, d M Y H:i:s") . " GMT");
header("Expires: " .gmdate ("D, d M Y H:i:s", time() + 3600). " GMT");
header("Cache-Control: max-age=3600");
echo time();
?>
```

看起来它真的很强大，我们打开浏览器来请求它，跟踪 HTTP 响应头，如下所示：

```
HTTP/1.1 200 OK
Date: Tue, 24 Mar 2009 04:53:29 GMT
Server: Apache/2.2.11 (Unix) PHP/5.2.1 DAV/2 SVN/1.4.3
X-Powered-By: PHP/5.2.1
Last-Modified: Tue, 24 Mar 2009 04:53:29 GMT
Expires: Tue, 24 Mar 2009 05:53:29 GMT
Cache-Control: max-age=3600
Content-Length: 10
Keep-Alive: timeout=30, max=100
Connection: Keep-Alive
Content-Type: text/html
```

仔细看好了，上面的 Expires 时间比 Last-Modified 时间整整晚 1 个小时，这是在服务器端计算的时间。我们来看看 IE 的临时文件目录，如图 6-14 所示。

图 6-14　浏览器缓存文件的截止期为 Cache-Control(max-age)计算后的时间

看看这里的截止期，并不是 Web 服务器返回的 Expires 时间，而是本地浏览器的上次访问时间 1 个小时之后。

性能

不幸的是，由于根本没有发送请求，所以我们无法使用 ab 来进行压力测试。不过，到了这个份儿上，我们或许可以不用再去考虑它的性能了，因为这已经完全依赖于用户 PC，浏览器读取本地缓存的速度取决于用户 PC 的内存和磁盘性能，因为浏览器会将缓存内容放在磁盘或内存中，从刚才的 HttpWatch 跟踪中我们已经看到，当直接使用浏览器缓存时，获取内容的时间为 0。

读到这里，你也许更加意识到 HTTP 是多么重要，它是浏览器和 Web 服务器沟通的语言，也是它们沟通的唯一方法，无论多么复杂的交流，其本质最终都体现在 HTTP 中。但是，往往很多人忽略了对于 HTTP 的深入理解和学习，而是被一些包在它外面的东西所蒙蔽，虽然这些东西可以让我们快速上手，但是你将永远不知道真相，考虑一下吧，该行动起来了。遗憾的是，这本书不是深入介绍 HTTP 的书，你可以阅读 RFC2616（HTTP/1.1）。

第 7 章

Web 服务器缓存

前面我们讨论过了动态内容缓存和静态化，基本上都是通过动态程序自身来实现缓存机制，包括缓存持久化、过期检查、缓存更新等。是否可以让 Web 服务器自己实现缓存机制呢？的确，Web 服务器是应该站出来做点什么了……

7.1 URL 映射

对于任何 Web 服务器，当我们向它发送一个 HTTP 请求后，它要做的主要工作就是解析 URL，然后完成从 URL 到实际内容或资源的映射。这里所说的映射是一个抽象的概念，实际上就是服务器处理请求并生成响应内容的过程，而这里之所以说“映射”，是希望从 URL 的角度来看服务器的处理过程。

URL 映射的过程或许你并不陌生，比如，我们请求以下 URL：

```
http://highperfweb.com/book/index.html
```

这时候 Web 服务器也许会将它定位到以下文件路径：

```
/data/www/book/index.html
```

你一定看出来了，Web 服务器为这个站点设置的根目录是/data/www，没错，看起来很简单，Web 服务器只需要读取这个文件的内容即可，这就是最终要传送给用户的内容了，这部分开销主要在于磁盘文件的 I/O 操作。

等等，一定是这样的吗？假如我们使用了 URL Rewrite（地址重写）技术，比如 Apache 的 mod_rewrite，那么以上 URL 完全有可能重写到另一个文件路径中，比如我们的 Rewrite 规则使得以上的 URL 发生了如下重写：

```
/book/index.html -> /pages/book/index.html
```

经过内部重写，现在，Web 服务器需要访问的文件变成了：

```
/data/www/pages/book/index.html
```

这还不算什么，我决定将它重写到一个动态内容中，规则产生的结果如下：

```
/book/index.html -> /reader.php?book=index
```

这种重写很常见，它让 URL 变得更加优雅，富有观赏性，同时有利于搜索引擎的收录。与此同时，这使得 Web 服务器从原本读取静态文件，变成了生成动态内容，增加了不少开销。

URL 映射也许还没有结束，假如 Web 服务器作为反向代理（Reverse Proxy）将请求转发给了后端的其他 Web 服务器，还会发生以下重写：

```
/reader.php?book=index -> http://backend/reader.php?book=index
```

这样一来，又增加了一定的网络 I/O 开销。对于反向代理，后面我们会进行专门的讨论。

在实际情况中，URL 映射的过程可能还需要经历更多的环节，比如 SSI 模式下对于内容的通读。如果我们不看过程，只关注 URL 和最终的响应内容，则它们确实就像一系列的对应关系。

7.2 缓存响应内容

你也许想到了，我们为什么不将这些对应关系缓存起来呢？很多时候，一个 URL 在一段较长的时间内对应一个唯一的响应内容，比如静态内容或者更新不太频繁的动态内容，一旦将最终内容缓存后，下次 Web 服务器便可以在收到请求后立即拿出事先缓存好的响应内容并返回给浏览器，因为这个操作发生在所有其他行为之前，所以必然会节省一定的时间。

幸运的是，将这份任务交给 Web 服务器来做是最合适不过的了，而且大多数情况下我们不需要亲自去实现它，主流的 Web 服务器软件都会提供类似的支持，比如 Apache 的 mod_cache，它在 URL 映射开始时检查缓存，如果缓存存在并处于有效期内，那么将直接取出作为响应内容返回给浏览器。

准备好缓存区

当然，还有一点很重要，那就是将缓存内容存储在什么位置，一般来说，本机内存和磁盘是主要选择，当然，也可以采用分布式设计，存储到其他服务器的内存或磁盘中，前面我们在介绍动态内容缓存时，便分别使用了磁盘、内存以及独立缓存服务器等方案。

同样，Apache 也提供了两个扩展，分别是 mod_disk_cache 和 mod_mem_cache，它们为 mod_cache 提供存储引擎，前者使用磁盘文件系统来存储缓存，后者使用内存。但实际上，这两个存储引擎一直都处于试验阶段，到目前为止，我看到 Apache 官方已经将 mod_mem_cache 从 Apache 最新文档的模块列表里清除掉了，根据 Apache 社区的一些讨论，可能是 mod_mem_cache 的实现机制导致它在 Apache 多进程模型下共享内存缓存的开

销较大，官方推荐使用 mod_disk_cache 来取代 mod_mem_cache，因为它在磁盘上维护了一块多个进程共享的缓存区，并且由于磁盘文件系统缓冲区以及 MMAP 的作用，磁盘缓存的访问速度甚至要超过 mod_mem_cache 实现的内存缓存。

所以在这里，我们不打算使用 Apache 的内存缓存机制，而是使用实现了磁盘缓存的 mod_disk_cache。另外，Lighttpd、Nginx 等也都有类似机制的缓存支持，比如 Lighttpd 提供了 mod_trigger_b4_dl 模块，可以在响应数据输出之前将它缓存到 memcached 缓存服务器中，并且在随后请求的 URL 映射之前查找缓存，所以它们的本质都是相似的，这样才可以最大程度地发挥 Web 服务器缓存的意义。

在 Apache 中开启磁盘缓存很容易，但是你必须在 Apache 编译的时候，为 configure 追加必要的模块，如下所示：

```
--enable-cache=shared --enable-disk-cache=shared --enable-so
```

然后，为 Apache 增加以下的配置：

```
LoadModule cache_module modules/mod_cache.so
LoadModule disk_cache_module modules/mod_disk_cache.so
CacheRoot /data/apache_cache
CacheEnable disk /
CacheDirLevels 5
CacheDirLength 3
```

在以上配置中，CacheRoot 指定了缓存内容的存储目录，CacheEnable 指定了缓存引擎，即磁盘，同时指定将站点根目录下的所有请求都进行缓存，如果你只需要缓存某个目录，那么可以进行如下配置：

```
CacheEnable disk /images
```

后面的 CacheDirLevels 和 CacheDirLength 两个参数，指定了缓存的目录分级结构，还记得前面我们介绍的 Smarty 内置动态缓存机制吗？当时由动态程序本身实现了缓存的多级目录存储，这是为了减少同一个目录大量文件的查找开销。这两个参数的表现效果随后我们就会看到。

缓存静态内容

也许你已经等不急了，是时候进行压力测试了，我们先找来两个静态页面，一个是使用了 SSI 的 shtml 页面，另一个是普通的 html 页面，但是它们两者的最终输出内容是一致的，也就是说内容长度是一样的。我们先在没有开启 mod_disk_cache 的情况下对它们进行测试。

首先是 shtml 页面，测试结果如下所示：

```
Document Path:          /pages/index_0513/index.shtml
Document Length:        50080 bytes
Concurrency Level:      100
Time taken for tests:   11.555 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      51065858 bytes
HTML transferred:       50779910 bytes
Requests per second:    865.43 [#/sec] (mean)
Time per request:       115.549 [ms] (mean)
Time per request:       1.155 [ms] (mean, across all concurrent requests)
Transfer rate:          42642.93 [Kbytes/sec] received
```

接下来是 html 页面，测试结果如下所示：

```
Document Path:          /pages/index_0513/full_index.htm
Document Length:        50080 bytes
Concurrency Level:      100
Time taken for tests:   2.600 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      504360431 bytes
HTML transferred:       500850080 bytes
Requests per second:    3846.88 [#/sec] (mean)
Time per request:       25.995 [ms] (mean)
Time per request:       0.260 [ms] (mean, across all concurrent requests)
Transfer rate:          189473.97 [Kbytes/sec] received
```

可以看出，前面的 shtml 页面需要较长的处理时间，它的吞吐率较低，原因你一定非常清楚，我们曾在介绍 SSI 的时候有过详细的解释。

下面，我们开启 mod_disk_cache，再来对它们进行同样条件的压力测试，为了测试在全缓存情况下的表现，我们在测试之前先手动请求几次这些页面，目的是让缓存预先生成。全缓存的意思就是测试中每次请求都可以命中缓存。

这时候，我们可以看到缓存目录中已经生成了缓存文件，如下所示：

```
s-colin:/data/apache_cache/Ay0/Pdx/NUn/t14/mK_ # ls
Lr8K8zQ.data  Lr8K8zQ.header
```

Apache 将缓存的 HTTP 头信息和正文分开独立存储，这样的好处是为缓存过期检查提供了方便，只需要检查 HTTP 头信息的文件即可。什么？如果你为这里也要进行缓存过期检查感到意外的话，那你一定忘记了缓存的本质，难道你希望 Apache 永远使用磁盘缓存区的这份内容吗？你必须有一种办法告诉它如何过期，别急，随后我们来探讨这个问题。

现在，缓存内容已经创建好了，我们再次对 shtml 页面进行压力测试，结果如下所示：

```
Document Path:          /pages/index_0513/index.shtml
Document Length:        50080 bytes
```

```
Concurrency Level:      100
Time taken for tests:   2.033 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      504509593 bytes
HTML transferred:       500898871 bytes
Requests per second:    4918.95 [#/sec] (mean)
Time per request:       20.330 [ms] (mean)
Time per request:       0.203 [ms] (mean, across all concurrent requests)
Transfer rate:          242760.02 [Kbytes/sec] received
```

再来对 html 页面进行测试，结果如下所示：

```
Document Path:          /pages/index_0513/full_index.htm
Document Length:        50080 bytes
Concurrency Level:      100
Time taken for tests:   2.045 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      504509593 bytes
HTML transferred:       500898871 bytes
Requests per second:    4890.96 [#/sec] (mean)
Time per request:       20.446 [ms] (mean)
Time per request:       0.204 [ms] (mean, across all concurrent requests)
Transfer rate:          240970.50 [Kbytes/sec] received
```

结果很清楚，对于使用了 SSI 的 shtml 页面，Web 服务器磁盘缓存起到了很大的性能提升，吞吐率提高了 4 倍之多，而对于普通的 html 页面，磁盘缓存的引入也有一定的性能提升，但是较为有限，吞吐率只提高了 27%左右。

缓存动态内容

再来看看动态内容，我们找回之前探讨动态内容缓存时的那个动态程序，在使用 APC 内存缓存以及 APC opcode cache 的时候，我们对它再次进行压力测试，这次是在 Apache 下，测试结果如下所示：

```
Document Path:          /place_posts_apc.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   0.428 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      13194181 bytes
HTML transferred:       13013000 bytes
Requests per second:    2336.76 [#/sec] (mean)
Time per request:       42.794 [ms] (mean)
Time per request:       0.428 [ms] (mean, across all concurrent requests)
Transfer rate:          30109.02 [Kbytes/sec] received
```

是的，表现非常好，这个结果是我们通过动态程序自身实现缓存机制的最好表现，其他使用磁盘和分布式缓存服务器的方案都要略逊于它。那么，使用 Web 服务器缓存后会如何呢？我们开启 mod_disk_cache，再次进行同样的压力测试，结果如下所示：

```
Document Path:          /place_posts.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   0.165 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      13346778 bytes
HTML transferred:       13091000 bytes
Requests per second:    6053.53 [#/sec] (mean)
Time per request:       16.519 [ms] (mean)
Time per request:       0.165 [ms] (mean, across all concurrent requests)
Transfer rate:          78901.42 [Kbytes/sec] received
```

哇！真是难以置信，它的吞吐率相比于之前使用 APC 内存缓存以及 APC opcode cache 的时候还高出了近两倍，很难想象这是一个动态内容的测试结果，但从磁盘缓存的角度看，它实际上只是对静态文件的请求。

实际上，这一次我们测试的动态内容本身并没有使用缓存，也就是说理论上每次请求都需要访问数据库，如果我们在不开启 Apache 磁盘缓存的情况下对其进行压力测试，结果肯定可想而知。

另一个问题产生了，在引入了 Web 服务器磁盘缓存后，原本指向动态内容的请求可以迅速地获得缓存，但是它如何知道缓存是否过期以及如何控制缓存的有效期呢？

提示：

提出一个问题往往比解决一个问题更重要，因为解决问题也许仅仅是一个教学上或实验上的技能而已。而提出新的问题、新的可能性，从新的角度去看旧的问题，都需要有创造性的想象力，而且标志着科学的真正进步。

——【美】爱因斯坦

控制有效期

说到缓存过期的问题，你也许得花一点时间回顾一下前面介绍浏览器缓存时提到的缓存协商和过期时间等 HTTP 头信息中的标记，没错，这些标记再次派上了用场，Web 服务器缓存对于动态内容或静态内容的过期检查机制仍然是建立在 HTTP/1.1 协议上的对话，除此之外，它们之间没有其他沟通方式，虽然它们同在一台物理机器上，但保持距离是非常必要的。

要为一个动态内容指定缓存有效期，仍然是在 HTTP 响应头中追加 Expires 标记，如果你希望动态内容立即过期，也就是说根本不要缓存这个动态内容，那么最简单的办法就是让 Expires 为 0，如下所示：

```
header("Expires: 0");
```

这样一来，Web 服务器便不会把这个动态内容放入缓存区。

下面，我们找回之前的 http_cache.php，它的功能仅仅是输出当前的 Unix Timestamp 时间，我们希望它被缓存 10 秒钟，我修改了一下代码，如下所示：

```
<?php
$modified_time = $_SERVER['HTTP_IF_MODIFIED_SINCE'];
if (strtotime($modified_time) + 10 > time())
{
        header("HTTP/1.1 304");
        exit(1);
}
header("Last-Modified: " . gmdate("D, d M Y H:i:s") . " GMT");
header("Cache-Control: max-age=10");
echo time();
?>
```

这些代码你一定不陌生，如果用浏览器直接请求它，它会告诉浏览器，10 秒内可以直接使用浏览器本地缓存，但在 10 秒内如果刷新页面进行缓存协商，动态程序也只会返回 304 状态码，而一旦超过 10 秒，动态程序便会返回完整的最新内容。

既然这样，我们暂且关闭 Apache 磁盘缓存，用浏览器直接请求这个动态内容，结果在我们的预料中，不论是刷新还是单击“转到”按钮，每隔 10 秒钟都会有一次请求获得的状态码为 200，并且返回最新的内容，如下所示：

```
1237962489
1237962499
1237962509
1237962519
1237962529
.....
```

现在，我们开启 Apache 磁盘缓存，这意味着在浏览器和动态内容之间增加了一块缓存区。我们再次通过浏览器请求它，不断地刷新以及单击“转到”按钮，让我们欣慰的是，结果和刚才一模一样，可见我们为动态程序设计的缓存过期策略对于 Apache 磁盘缓存区仍然有效，而且工作得相当好，完全遵循我们的旨意。

可是，假如动态内容没有输出 Expires 标记的话，Web 服务器缓存区将如何判断它的有效期呢？这就得用到另一个我们熟悉的标记了，那就是 Last-Modified，如果动态内容的 HTTP 头信息中包含 Last-Modified 时间，那么 Web 服务器缓存区会使用默认最大缓存时间，比

如对 mod_cache 配置如下：

```
CacheMaxExpire 3600
```

这样 Web 服务器便可以在日后将过期时间和 Last-Modified 时间进行对比，以判断内容是否过期。而如果动态内容的 HTTP 头信息中没有 Last-Modified 时间的话，处理方法取决于 mod_cache 的另一个配置，即 CacheIgnoreNoLastMod，如果开启它的话，动态内容将不被缓存，因为它既没有 Last-Modified，也没有 Expires，缓存区不愿意接管它，但如果使用 CacheIgnoreNoLastMod 的默认设置，即关闭，缓存区会记录当前时间作为 Last-Modified，并且使用默认最大缓存时间。

另外，我们还希望有些特定的动态内容可以跳过 Web 服务器缓存区，虽然我们可以采用将 Expires 设置为 0 的办法，但是还有其他的解决方案，比如 mod_cache 还提供了以下的配置：

```
CacheIgnoreHeaders Set-Cookie
```

这样一来，凡是需要创建浏览器 cookies 的动态内容，都可以跳过缓存区，比如用户登录成功后生成包含登录状态的 cookies，我们当然不希望它被缓存。

静态内容同样需要控制它在 Web 服务器缓存区中的有效期，但我想你已经很清楚了，它的本质仍然是那些 HTTP 头信息中的标记，不同的是，对于静态内容来说，这些都可以通过 Web 服务器来轻松配置。

取而代之

你也许会想，这种 Web 服务器缓存机制是否可以取代动态程序自身的缓存机制呢？当然，这也不失为一种方案，但你需要进行一些考虑。

这样做会让动态程序依赖于特定的 Web 服务器，降低了应用的可移植性，但是，这也许对于你的站点来说无关紧要，你根本没有打算将其移植到其他的平台，你只是希望快速建立一层透明的缓存机制，而且尽量少地修改动态程序代码，因为你对它们很厌烦。

即便是需要移植到其他的 Web 服务器，由于我们采用的是基于 HTTP 的缓存机制，所以完全可以使用反向代理来快速对接，这在下一章中会有介绍，可以肯定的是，编写面向 HTTP 缓存友好的动态程序是非常必要的，它将拥有极强的生命力和适应性。

有一点需要注意的是，这种缓存机制实质上是以 URL 为主键的 key-value 结构缓存，所以你必须保证所有希望缓存的动态内容都有唯一的 URL，这不难做到，事实上也许它们本来就是这样，除非你采用了一些比较个性化的技术，比如你的动态内容通过 HTTP 请求头

信息中的某些标记来决定不同的内容输出，对于这种情况，如果你仍然希望应用缓存，那么你可以将这些标记同时补全在 URL 参数中，总之记住保证 URL 的唯一性。

除此之外，将动态内容缓存在 Web 服务器缓存区中，所能遇到的问题无非都是一些缓存应用中共性的问题，我想你都可以解决，总之，编写面向 HTTP 缓存友好的动态程序是你需要唯一考虑的事。

也许在有些时候，动态程序自身实现页面缓存（如 Smarty 内置缓存机制）显得有些多余，因为它们完全不用自己动手，而只需要打个招呼，说声 Expires 即可，在它们的前方，会有 Web 服务器缓存及浏览器缓存，还有随后即将介绍的反向代理缓存，帮它们来打理一切。是的，交给小弟们去做吧，它们队伍庞大，训练有素，你完全可以休假了。

7.3 缓存文件描述符

对于静态内容，特别是拥有大量小文件的站点，Web 服务器相当大的一部分开销花在了打开文件上，即 open()系统调用，所以我们还可以考虑将打开后的文件描述符直接缓存到 Web 服务器的内存中，当然，文件描述符是反映系统资源的数据结构，它也只能存在于本机内存中。

Apache 提供了相应的扩展 mod_file_cache，它需要我们提供希望缓存文件描述符的文件列表，比如我们进行如下配置：

```
CacheFile /data/www/htdocs/test.htm
```

这样一来，Apache 在启动的时候便会打开这些文件，并持续到 Apache 关闭为止，在这期间，如果有对这些文件的请求，Apache 将通过文件描述符直接把文件传送出去。

我们找来之前使用过的一个 151 字节的小文件，好久没用它了，这次它比较且有代表性。为了和使用文件描述符缓存后进行对比，我们先对它进行一次压力测试，结果如下所示：

```
Document Path:          /test.htm
Document Length:        151 bytes
Concurrency Level:      100
Time taken for tests:   1.840 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      4282568 bytes
HTML transferred:       1510906 bytes
Requests per second:    5435.59 [#/sec] (mean)
Time per request:       18.397 [ms] (mean)
Time per request:       0.184 [ms] (mean, across all concurrent requests)
Transfer rate:          2273.27 [Kbytes/sec] received
```

与此同时，我们还用 strace 跟踪了 httpd 进程，系统调用统计结果如下所示：

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 29.18    0.000075           0      1897      1626 open
 22.18    0.000057           0       271           writev
 12.45    0.000032           0       813       271 read
  9.73    0.000025           0      2439           gettimeofday
  7.00    0.000018           0       271           write
  7.00    0.000018           0       271           shutdown
  6.61    0.000017           0       271           munmap
  5.84    0.000015           0       271           accept
  0.00    0.000000           0       542           close
  0.00    0.000000           0       271           times
  0.00    0.000000           0         1           brk
  0.00    0.000000           0       271           poll
  0.00    0.000000           0       271           mmap2
  0.00    0.000000           0       271           stat64
  0.00    0.000000           0       542           fcntl64
  0.00    0.000000           0       271           getsockname
------ ----------- ----------- --------- --------- ----------------
100.00    0.000257                  8944      1897 total
```

果然，open()系统调用的时间占用比例最高，其次是用于传送文件内容的 writev()系统调用。

接下来，我们使用 mod_file_cache，并将这个静态文件设置为缓存文件描述符，再次进行压力测试，结果如下所示：

```
Document Path:          /test.htm
Document Length:        151 bytes
Concurrency Level:      100
Time taken for tests:   1.540 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      4060000 bytes
HTML transferred:       1510000 bytes
Requests per second:    6494.86 [#/sec] (mean)
Time per request:       15.397 [ms] (mean)
Time per request:       0.154 [ms] (mean, across all concurrent requests)
Transfer rate:          2575.11 [Kbytes/sec] received
```

吞吐率比刚才提高了将近 20%，同时我们来看 strace 跟踪的系统调用统计结果：

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 44.36    0.000114           0       275           writev
 14.01    0.000036           0       275           times
  9.73    0.000025           0      1650      1650 open
  6.61    0.000017           0       275           shutdown
  6.23    0.000016           0       275           write
  6.23    0.000016           0       550           fcntl64
  4.67    0.000012           0       275           close
  4.28    0.000011           0       825       275 read
  3.89    0.000010           0       275           munmap
```

```
  0.00    0.000000           0      2475           gettimeofday
  0.00    0.000000           0       275           poll
  0.00    0.000000           0       275           mmap2
  0.00    0.000000           0       275           accept
  0.00    0.000000           0       275           getsockname
------ ----------- ----------- --------- --------- ----------------
100.00    0.000257                  8250      1925 total
```

open()系统调用的时间占用比例明显减少，现在的主要时间花在了负责数据传送的 writev() 系统调用上，这是我们所希望看到的结果。

很显然，这种缓存文件描述符的缓存方案只适用于静态内容，与前面将静态内容整体缓存到磁盘相比，这种方法可以减少打开文件的开销，而前者仍然需要从磁盘缓存区打开文件，但是，由于前者发生在 URL 映射的最前端，所以它完全弥补了由于打开文件的开销带来的性能损失，事实上前者的测试结果还要略高于缓存文件描述符时的吞吐率，我们在使用磁盘缓存区的时候对这个 151 字节的静态文件进行压力测试，结果如下所示：

```
Document Path:          /test.htm
Document Length:        151 bytes
Concurrency Level:      100
Time taken for tests:   1.525 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      4404400 bytes
HTML transferred:       1511510 bytes
Requests per second:    6555.80 [#/sec] (mean)
Time per request:       15.254 [ms] (mean)
Time per request:       0.153 [ms] (mean, across all concurrent requests)
Transfer rate:          2819.76 [Kbytes/sec] received
```

缓存文件描述符对于较大的文件来说也是不适合的，因为处理它们的主要时间花在传送数据上，而不是打开文件。

另外，由于在 Apache 启动的时候就打开了大量的文件，并且需要在多个子进程中共享文件描述符，所以这必然带来了更多的内存开销，要说唯一节省的资源，那就是避免了磁盘缓存区的大量静态文件冗余，但是这算不了什么。还有，当这些文件更新后，必须重启 Apache 才能生效。

从以上这些方面来看，似乎缓存文件描述符的意义不是不大，它的处境比较尴尬，但是至少通过我们的尝试，你已经了解了它的本质，也许在你的站点中会有它的一席之地。需要说明的是，不要完全被以上的压力测试结果所影响，因为我们测试的只是来自于 Apache 的实现，对于其他 Web 服务器或许有更加高效的实现，至少缓存文件描述符的思想是可取的。

第 8 章

反向代理缓存

在前面的章节中，我们曾经多次提到反向代理服务器，这一章我们就来看个究竟，幸运的是，有了前面对浏览器缓存和 Web 服务器缓存的了解后，你将会很容易地熟悉反向代理以及它的缓存机制。

8.1 传统代理

很久以前，我们通常需要通过代理服务器来访问互联网上的 Web 站点，代理服务器本身接入了互联网，而我们通过内部网络与代理服务器相连。即便是现在，有些时候为了访问一些由于某种原因无法直接访问的站点，我们也会通过特定的代理服务器，绕过某些限制来访问目标站点。

无论如何，这些代理服务器所做的工作都是一样的，那就是将来自用户的 HTTP 请求转发给最终的 Web 服务器，然后再将从 Web 服务器收到的响应数据返回给用户浏览器。所以，从 Web 服务器的角度来看，代理服务器和用户浏览器的本质是一样的，它们都扮演着 HTTP 代理的角色。需要说明的是，我们这里所讲的代理服务器，主要是指 Web 代理服务器。

现在，我们普遍已经不使用代理服务器了，当我们的 PC 处于内部网络时，网关会使用 NAT（网络地址转换）技术，将 PC 的内部 IP 地址和网关的外网 IP 地址进行相互转换，使得 PC 发出的请求可以顺利到达外部网络的 Web 服务器，同时将返回的数据正确地传送给内部网络的 PC。

在某种意义上，NAT 在这里起到的作用等同于代理服务器，但是它们的不同在于，代理服务器工作在应用层，所以只有当它支持某个协议的时候才可以转发该协议的数据，而 NAT 工作在应用层以下，它可以透明地转发应用层协议的数据，比如 HTTP、FTP、SMTP 等。不过，它们都有共同的一点，也是代理的特点，那就是用户的 PC 隐藏在了代理服务器或者 NAT 网关之后，换句话说，Web 服务器只知道是代理服务器或者网关发来的请求，而并不知道还存在幕后操纵者。

这样一来，我们的 PC 便不用直接暴露在互联网中，有效提高了安全性能，因为攻击者是无法主动找到我们的，代理服务器或者网关就像一道防火墙保护着我们，但在实际情况中，我们的一些主动行为往往带来了更大的安全问题，遗憾的是，安全性不是我们的讨论主题，你可以阅读其他的相关资料。.

另外，正是因为代理服务器工作在应用层，所以它可以很容易地提供基于缓存的加速功能，比如一个机构的内部网络通过代理服务器上网，一旦某个用户访问过的网页被缓存在代理服务器上，那么随后一段时间里，内部网络的其他用户便可以快速获得这个网页，而不需要再次经过外部网络请求 Web 服务器，随之还带来的另一个好处就是节省了带宽。

代理服务器还有许多实用的功能，比如 HTTP 过滤等，这里就不具体介绍了。你可能已经等不急了，我们还是来看看反向代理服务器吧。

8.2 何为反向

也许你已经在使用反向代理服务器，但是“反向”这个词，究竟意味着什么呢？带着这个问题，回想刚才提到的传统代理服务器的特点，即用户隐藏在代理服务器之后，那么，反向代理服务器的特点便与此刚好相反，那就是 Web 服务器隐藏在代理服务器之后。我们将这种代理机制称为反向代理（Reverse Proxy），同时，实现这种机制的服务器，便称为反向代理服务器（Reverse Proxy Server）。

隐藏在反向代理服务器之后的 Web 服务器，我们习惯称它为后端服务器（Back-end Server），相对的，反向代理服务器在这里便成了前端服务器（Front-end Server）。通常，反向代理服务器暴露在互联网中，而后端的 Web 服务器通过内部网络与它相连，当然，你也可以将反向代理服务器和 Web 服务器运行在同一台物理服务器上。

引入反向代理后，用户将通过反向代理服务器来间接访问 Web 服务器，不过用户并不关心这些，因为反向代理服务器可以完美地充当用户心目中的 Web 服务器，至于反向代理服务器和后端 Web 服务器的沟通，则仍然是基于 HTTP，这一点和传统代理的本质是一样的。

将 Web 服务器隐藏在后端，同样也带来了一定的安全性，但这不是反向代理的主要目的，因为完全可以使用 iptables 来作为防火墙以达到同样的安全性目的。

顺便一提，同样也存在反向 NAT，不过似乎这只是我发明的称呼，NAT 技术一般不依赖于具体应用，不过，从 Web 服务器的角度来看，如果将 Web 服务器隐藏在 NAT 网关后，遵循“反向”代理的命名文化，我想称其为反向 NAT 并不奇怪，事实上这便是服务器集群的原理，后面我们会专门探讨这个问题。也许“反向 NAT”这个称呼比较山寨化，不过这并不重要，这里的意思是，我们也许可以换种思维方式来看待 Web 服务器和用户 PC，某种意义上它们是完全对等的，都是接入网络的机器而已，所以理论上凡是应用在用户端的技术，同样也可以应用在服务器端。

那么，反向代理的主要目的是什么呢？有一点肯定没错，那就是基于缓存的加速。

8.3 在反向代理上创建缓存

的确，就像 Web 服务器缓存和浏览器缓存一样，我们同样可以将内容缓存在反向代理服务器上，所有缓存机制的实现仍然采用 HTTP/1.1 协议，前面曾经有过详细介绍，你应该对 HTTP 缓存协商和过期时间等标记已经非常熟悉了，也就是说，只要我们的站点是面向 HTTP 缓存友好的内容，那么便可以直接放在代理服务器的后端，达到反向代理缓存的目的。

如果没有缓存

为了测试反向代理缓存带来的性能提升，我们需要一些对比数据，主要包括：

- 不使用反向代理时的性能数据
- 使用反向代理但不使用缓存时的性能数据

我们开始行动吧，首先针对动态内容，它使用了 APC opcode cache 和 APC 内存缓存，是前面几种动态内容缓存实现中性能表现最好的，现在我们直接对它进行压力测试，而不通过任何反向代理。这里我们使用 Apache，测试结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        www.highperfweb.com
Server Port:            80
Document Path:          /place_posts_apc.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   4.482 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      132034077 bytes
HTML transferred:       130221000 bytes
Requests per second:    2231.25 [#/sec] (mean)
Time per request:       44.818 [ms] (mean)
Time per request:       0.448 [ms] (mean, across all concurrent requests)
Transfer rate:          28769.60 [Kbytes/sec] received
```

表现非常不错，我知道 Apache 已经尽力了，顺便说一下，事实上对于这个动态内容，使用 Nginx 和 Lighttpd 的 fastcgi 也将得到同样的测试结果，后两者的测试结果如下所示：

```
Server Software:        nginx/0.7.30
Server Hostname:        www.highperfweb.com
Server Port:            8002
Document Path:          /place_posts_apc.php?marker_id=12882
Document Length:        13165 bytes
Concurrency Level:      100
Time taken for tests:   4.373 seconds
Complete requests:      10000
```

```
Failed requests:        0
Write errors:           0
Total transferred:      133120000 bytes
HTML transferred:       131650000 bytes
Requests per second:    2286.54 [#/sec] (mean)
Time per request:       43.734 [ms] (mean)
Time per request:       0.437 [ms] (mean, across all concurrent requests)
Transfer rate:          29725.00 [Kbytes/sec] received

Server Software:        lighttpd/1.4.20
Server Hostname:        www.highperfweb.com
Server Port:            8001
Document Path:          /place_posts_apc.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   4.437 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      131563388 bytes
HTML transferred:       130000000 bytes
Requests per second:    2253.95 [#/sec] (mean)
Time per request:       44.367 [ms] (mean)
Time per request:       0.444 [ms] (mean, across all concurrent requests)
Transfer rate:          28958.76 [Kbytes/sec] received
```

我想你可以分析出原因，在前面介绍服务器并发处理能力的章节中，我们已经知道了 Nginx 和 Lighttpd 的优势主要体现在网络 I/O 模型上，而此时的主要瓶颈并不在于此。

下面我们用 Nginx 作为反向代理服务器，它的配置非常简单，只需要设置 proxy_pass 指令即可，同时别忘了打开 mod_proxy 模块。

我们将 nginx 运行在 Apache 所在的物理服务器上，nginx 监听在 8002 端口，它作为 Apache 的反向代理，将用户的请求都转发到 Apache 监听的 80 端口，配置如下所示：

```
location ~ \.*$ {
    proxy_pass   localhost:80;
}
```

为了测试 Nginx 作为反向代理是否能正常工作，我们试探性地对 nginx 发出 5 次请求：

```
    ab -n5 http:// www.highperfweb.com:8002/place_posts_apc.php?marker_
id=12882
```

同时，我们看到了在 Nginx 和 Apache 的日志中同时出现了这 5 次请求的记录，但记录格式略有不同，以下是 Nginx 的日志：

```
    127.0.0.1 - - [26/Mar/2009:15:49:31 +0800] "GET /place_posts_apc.php?
marker_id=12882 HTTP/1.0" 200 13000 "-" "ApacheBench/2.3"
    127.0.0.1 - - [26/Mar/2009:15:49:31 +0800] "GET /place_posts_apc.php?
marker_id=12882 HTTP/1.0" 200 13000 "-" "ApacheBench/2.3"
```

```
    127.0.0.1 - - [26/Mar/2009:15:49:31 +0800] "GET /place_posts_apc.php?
marker_id=12882 HTTP/1.0" 200 13000 "-" "ApacheBench/2.3"
    127.0.0.1 - - [26/Mar/2009:15:49:31 +0800] "GET /place_posts_apc.php?
marker_id=12882 HTTP/1.0" 200 13000 "-" "ApacheBench/2.3"
    127.0.0.1 - - [26/Mar/2009:15:49:31 +0800] "GET /place_posts_apc.php?
marker_id=12882 HTTP/1.0" 200 13000 "-" "ApacheBench/2.3"
```

以下是 Apache 的日志：

```
    127.0.0.1 - - [26/Mar/2009:15:48:07 +0800] "GET /place_posts_apc.php?
marker_id=12882 HTTP/1.0" 200 13000
    127.0.0.1 - - [26/Mar/2009:15:48:20 +0800] "GET /place_posts_apc.php?
marker_id=12882 HTTP/1.0" 200 13000
    127.0.0.1 - - [26/Mar/2009:15:48:20 +0800] "GET /place_posts_apc.php?
marker_id=12882 HTTP/1.0" 200 13000
    127.0.0.1 - - [26/Mar/2009:15:48:20 +0800] "GET /place_posts_apc.php?
marker_id=12882 HTTP/1.0" 200 13000
    127.0.0.1 - - [26/Mar/2009:15:48:20 +0800] "GET /place_posts_apc.php?
marker_id=12882 HTTP/1.0" 200 13000
```

也就是说，Nginx 没有将从 Apache 获得的内容缓存起来，而是每次都需要重新请求 Apache 来获取内容，这正是我们需要的测试条件，再次对这个动态内容进行压力测试，这次是将所有的请求发送给监听在 8002 端口的 Nginx，测试结果如下所示：

```
Server Software:        nginx/0.7.30
Server Hostname:        www.highperfweb.com
Server Port:            8002
Document Path:          /place_posts_apc.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   5.080 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      131494508 bytes
HTML transferred:       130024214 bytes
Requests per second:    1968.67 [#/sec] (mean)
Time per request:       50.796 [ms] (mean)
Time per request:       0.508 [ms] (mean, across all concurrent requests)
Transfer rate:          25280.17 [Kbytes/sec] received
```

可以看到，在使用反向代理但不缓存内容的时候，吞吐率下降了 10%左右，多出的处理时间不难想象，原本直达 Web 服务器的请求，现在需要“绕路”转达，所以时间必然会有所增加，幸好这部分时间和动态内容处理的时间相比微乎其微，且吞吐率下降只有 10%左右。

有时候，人们喜欢用这种方式将 Web 服务器和应用服务器分离，即前端的 Web 服务器处理一些静态内容，同时又作为反向代理，将动态内容的请求转发给后端的应用服务器来处理。比如 Nginx 或 Lighttpd 将 PHP 程序的请求转发给后端的 Apache，在 Nginx 中配置非常简单：

```
location ~ \.php$ {
    proxy_pass   localhost:80;
}
```

这样一来，Nginx 便包揽了静态文件的处理工作，对于静态内容，Nginx 利用 epoll 模型可以在较大并发用户数的情况下依然提供较高的吞吐率，这是它所擅长的工作，而后端 Apache 则可以专注地处理动态内容。

但是，对于动态内容的请求比例较多的情况，这种方式也有明显的不足，通过前面的测试结果你不难分析出原因。也许随后的反向代理缓存可以很好地解决这个问题。

我们还是暂且忘记反向代理缓存，再来试试静态内容，首先我们使用 Nginx 和 Lighttpd 分别作为 Web 服务器来直接访问 151 字节的静态内容，结果分别如下所示：

```
Server Software:        nginx/0.7.30
Server Hostname:        www.highperfweb.com
Server Port:            8002
Document Path:          /test.htm
Document Length:        151 bytes
Concurrency Level:      100
Time taken for tests:   0.772 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      3620000 bytes
HTML transferred:       1510000 bytes
Requests per second:    12947.85 [#/sec] (mean)
Time per request:       7.723 [ms] (mean)
Time per request:       0.077 [ms] (mean, across all concurrent requests)
Transfer rate:          4577.27 [Kbytes/sec] received

Server Software:        lighttpd/1.4.20
Server Hostname:        www.highperfweb.com
Server Port:            8001
Document Path:          /test.htm
Document Length:        151 bytes
Concurrency Level:      100
Time taken for tests:   0.741 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      3840000 bytes
HTML transferred:       1510000 bytes
Requests per second:    13487.18 [#/sec] (mean)
Time per request:       7.414 [ms] (mean)
Time per request:       0.074 [ms] (mean, across all concurrent requests)
Transfer rate:          5057.69 [Kbytes/sec] received
```

表现非常出色，你也许已经意识到这次没有使用 Apache 来为静态内容提供服务，是的，至少在这里，Lighttpd 比它更加适合。下面，我们做一件有意思的事情，让 Nginx 成为 Lighttpd 的代理，并且不缓存内容。

Nginx 仍然监听在 8002 端口，而 Lighttpd 监听在 8001 端口，对 Nginx 的 mod_proxy 配置如下：

```
location ~ \.*$ {
    proxy_pass   localhost:8001;
}
```

这将导致所有发送到 Nginx 的请求都被转发到 Lighttpd，我们这次来对 Nginx 进行测试，结果如下所示：

```
Server Software:        nginx/0.7.30
Server Hostname:        www.highperfweb.com
Server Port:            8002
Document Path:          /test.htm
Document Length:        151 bytes
Concurrency Level:      100
Time taken for tests:   2.128 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      3810000 bytes
HTML transferred:       1510000 bytes
Requests per second:    4699.49 [#/sec] (mean)
Time per request:       21.279 [ms] (mean)
Time per request:       0.213 [ms] (mean, across all concurrent requests)
Transfer rate:          1748.54 [Kbytes/sec] received
```

天啊，太差劲了，不是吗？我想无论如何都不会有人这样做的，除非引入反向代理缓存。

引入缓存

好吧，看来是反向代理缓存上场的时候了，刚才我们一直使用 Nginx 来作为反向代理服务器，我们知道反向代理只是 Nginx 的一个扩展模块，并且它的缓存机制到目前为止还在不断完善，所以我们暂且放弃它。

要做好一件事情，一定要专注，是的，Squid 在这方面家喻户晓，不过，除了作为反向代理，它还热衷于很多其他的工作，比如传统代理、访问控制、身份验证、流量管理等，正因为这样，它呈现出过于重量级的身躯，而且它的配置也过于复杂，的确，不仅要专注，而且要把事情做得简单，这是新时代的法则，而 Squid 则是 20 世纪 90 年代的巨无霸，虽然是 90 后，但是从互联网技术的发展年代来看，它依然古老。

如果你的站点确实需要这样的巨无霸，当然，你可以尝试它。但是在这里，我们需要的很简单，仅仅是基于反向代理缓存的加速功能，所以我们选择了 Varnish，它更加专注于反向代理，而且对于后面要介绍的负载均衡也有很好的支持。

现在我们将使用 Varnish 作为反向代理服务器，并且将内容缓存在 Varnish 中，来测试一下刚才的动态内容和静态内容。

我们先将 Varnish 配置成为 Apache 的代理，修改 Varnish 的配置文件 default.vcl 如下所示：

```
backend default {
    .host = "127.0.0.1";
    .port = "80";
}
```

然后我们启动 varnishd，在启动时需要指定它自身的监听端口，以及配置文件路径和存储引擎，如下所示：

```
    s-colin:~ # /usr/local/varnish/sbin/varnishd -a :8010 -T localhost:8011
-f /usr/local/varnish/etc/varnish/default.vcl -s file,/var/cache/varnish.
cache,512M
```

这时候，varnishd 监听在 8010 端口，并且在 8011 端口提供命令行管理服务，同时，我们使用了磁盘文件作为缓存引擎，预分配了 512MB 的空间，随后我们可以看到缓存文件，Varnish 在一个文件中采用特殊的存储结构来维护缓存，类似于 MySQL 的 Innodb 存储引擎。

```
s-colin:/var/cache # ls -lh /var/cache/varnish.cache
-rw------- 1 root root 512M 2009-03-24 10:53 /var/cache/varnish.cache
```

我们来对刚才的动态内容再次进行压力测试，结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        www.highperfweb.com
Server Port:            8010
Document Path:          /place_posts_apc.php?marker_id=12882
Document Length:        13000 bytes
Concurrency Level:      100
Time taken for tests:   1.363 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      132673048 bytes
HTML transferred:       130052000 bytes
Requests per second:    7336.76 [#/sec] (mean)
Time per request:       13.630 [ms] (mean)
Time per request:       0.136 [ms] (mean, across all concurrent requests)
Transfer rate:          95057.61 [Kbytes/sec] received
```

太棒了，到目前为止，这个结果应该是该动态内容表现出的最佳性能，我们的努力是值得的，而我们所做的，仅仅是在它的前端增加了一个反向代理缓存而已。

与此同时，你也许看到了在以上的测试结果中，Server Software 显示为 Apache，奇怪，我们明明是对 Varnish 的测试啊，没错，我们的测试没有出问题，而是因为 Varnish 希望自

已足够透明，它的身手敏捷，不留痕迹，不过，它还是希望留下一点回忆，我们看从 Varnish 获得的响应 HTTP 头：

```
HTTP/1.1 200 OK
Server: Apache/2.2.11 (Unix) PHP/5.2.1 DAV/2 SVN/1.4.3
X-Powered-By: PHP/5.2.1
Content-Type: text/html
Content-Length: 13000
Date: Fri, 27 Mar 2009 06:22:03 GMT
X-Varnish: 1202470644
Age: 0
Via: 1.1 varnish
Connection: keep-alive
```

可以从 Via 标记中看到 varnish，这将便于一些浏览器端工具的识别和统计。

我们再来看看静态内容在反向代理缓存下的表现，还是刚才的 151 字节的静态文件，我们用 Varnish 分别作为 Apache 和 Lighttpd 的代理服务器，测试结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        www.highperfweb.com
Server Port:            8010
Document Path:          /test.htm
Document Length:        151 bytes
Concurrency Level:      100
Time taken for tests:   1.218 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      4641846 bytes
HTML transferred:       1510604 bytes
Requests per second:    8210.73 [#/sec] (mean)
Time per request:       12.179 [ms] (mean)
Time per request:       0.122 [ms] (mean, across all concurrent requests)
Transfer rate:          3721.97 [Kbytes/sec] received

Server Software:        lighttpd/1.4.20
Server Hostname:        www.highperfweb.com
Server Port:            8010
Document Path:          /test.htm
Document Length:        151 bytes
Concurrency Level:      100
Time taken for tests:   1.246 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      4238460 bytes
HTML transferred:       1513020 bytes
Requests per second:    8028.05 [#/sec] (mean)
Time per request:       12.456 [ms] (mean)
Time per request:       0.125 [ms] (mean, across all concurrent requests)
Transfer rate:          3322.91 [Kbytes/sec] received
```

可以看出，两者结果中的吞吐率不分上下，可见，有了反向代理缓存，后端服务器的性能

已经不那么重要，即便后端是 Apache，我们看到最后的吞吐率有时候甚至比 Lighttpd 作为后端时还高。

另一方面，Varnish 自身的并发处理能力也很重要，我们曾经在服务器并发处理能力这一章中详细地介绍过影响 Web 服务器处理能力的一些因素和本质，当然，它同样适用于反向代理服务器，你完全可以根据那些知识，参考 Varnish 的手册，进行适当的配置，比如使用 epoll 模型。

修改缓存规则

作为反向代理服务器，实际上它拥有非常大的控制权，因为不论是用户浏览器，还是后端的 Web 服务器，要想正常工作起来，都得经过它，所以，它可以对流经它的任何数据进行改写，只要它愿意，没有做不到的。

不过，我们当然不希望它修改数据正文，它也不感兴趣，因为它不是黑客，它是帮我们提升性能的高手，所以它做的任何修改，都是本着提升性能的原则，说到底，也就是决定哪些内容可以缓存，哪些内容不可以缓存。

说到这里，你也许会感到奇怪，内容是否可以被缓存，这不是通过 HTTP 来决定的吗？没错，反向代理服务器正是通过修改流经它的数据的 HTTP 头信息来达到这个目的，同时，它们还可以针对一些 HTTP 头信息进行必要的缓存策略配置，这就好比反向代理服务器是浏览器和 Web 服务器之间的协调员，浏览器和 Web 服务器通过 HTTP 将自己的需求告诉反向代理服务器，但是反向代理服务器可能有更好的想法，它在不伤害另外两方的情况下，可以进行必要的协调，达到更好的效果。

这里不得不说 Varnish 的配置文件，它使用了一种非常有趣的配置语言，称为 VCL（Varnish Configuration Language），我们可以通过它来修改 HTTP 头信息以及进行必要的策略定制。VCL 最大的特点就是它将反向代理的工作按照时间点分为多个阶段，并在各个阶段通过事件函数（或者回调函数）的风格，将控制权交给我们。

我们以最常用的两个阶段为例，首先是 vcl_revc 函数，它在请求刚刚到达反向代理服务器时被调用，所以，我们在这里要做的，就是告诉 Varnish 哪些请求需要先查找缓存，哪些请求直接转发到后端。比如以下的默认配置：

```
sub vcl_recv
{
    if (req.request != "GET" &&
      req.request != "HEAD" &&
      req.request != "PUT" &&
      req.request != "POST" &&
```

```
        req.request != "TRACE" &&
        req.request != "OPTIONS" &&
        req.request != "DELETE")
    {
        /* Non-RFC2616 or CONNECT which is weird. */
        return (pipe);
    }
    if (req.request != "GET" && req.request != "HEAD")
    {
        /* We only deal with GET and HEAD by default */
        return (pass);
    }
    if (req.http.Authorization || req.http.Cookie)
    {
        /* Not cacheable by default */
        return (pass);
    }
    return (lookup);
}
```

乍一看，这似乎不像配置文件，的确，它就像高级语言的函数，你会越看越亲切。我们看到它的内部包含了三个条件判断语句，在满足条件的情况下，它们会终止函数，并指定一个返回值，这表示它告诉 Varnish 接下来应该进入哪个阶段，比如上面的 pass 代表不检查缓存，直接将请求转发给后端，进入 vcl_pass 阶段；而 lookup 代表在缓存区中查找内容，然后可能会进入 vcl_hit 阶段或者 vcl_miss 阶段，它们分别代表缓存命中或者没有命中。

在这些条件判断中，我们看到一些变量，的确，请允许我在这里称它们为“变量”，比如 req.request 代表了来自浏览器的 HTTP 请求的类型，如 GET 或 POST；而 req.http.cookie 则代表了 HTTP 请求中携带的 cookies 内容。除此之外，Varnish 定义了一系列的变量，我们可以在事件函数中获得或重写它们，这里顺便列出几个常见的变量：

client.ip

代表用户浏览器端的 IP 地址。

req.url

代表用户请求的 URL 地址。

req.http.[header-key]

代表用户 HTTP 请求头信息中的信息，比如 req.http.host 表示请求的主机名，而 req.http.accept-language 表示用户浏览器支持的语言类型。

obj.status

代表从后端服务器返回的 HTTP 状态码，比如 304。

obj.response

代表从后端服务器返回的 HTTP 状态信息，比如 Not Modified。

obj.cacheable

代表从后端服务器返回的内容可以被缓存，它实际上是指那些返回内容的 HTTP 状态码为 200、203、300、301、302、404、410，并且包含非即时过期的 Expires 或者 Cache-Control 标记。

还有很多类似的变量，我们这里就不详细介绍了，你可以查阅 Varnish 的在线手册。

再看下面的 vcl_fetch 函数，它同样非常重要，它是在反向代理服务器从后端服务器获得内容后调用，所以我们需要在这里决定哪些内容需要被缓存。我们看以下的配置：

```
sub vcl_fetch
{
    if (!obj.cacheable)
    {
        return (pass);
    }
    if (obj.http.Set-Cookie)
    {
        return (pass);
    }
    set obj.prefetch =  -30s;
    return (deliver);
}
```

这里的返回值代表另一些含义，pass 代表将内容直接传送给浏览器，而不需要被缓存；deliver 代表将内容写入缓存区。在这个函数中，第一个条件判断可以说是反向代理的必备策略，也就是如果发现内容不可以缓存（!obj.cacheable），则直接传送给浏览器，而第二个条件判断是指，如果从后端服务器获取的内容包含了 set-cookie 的 HTTP 头标记，也就是需要设置 cookies 到浏览器，那么也无须缓存内容。

以上我们简单介绍了 VCL 中的函数和变量是如何工作的，你完全可以根据站点的需要来随意配置它们，同时，我们需要明白，这只是 Varnish 的一种实现方法，事实上在 Nginx 中也有类似的机制，但不是通过 VCL。总之，对于反向代理来说，它们总是可以通过任何手段来重写经过它的 HTTP 头信息，也可以通过其他自定义的机制来直接干涉缓存策略，往往前者的目的也在于此。

清除缓存

正如 Smarty 为动态内容提供了清除缓存的 API 一样，反向代理也必须赋予我们清除缓存

的能力，Varnish 提供了两种方法，前者基于命令行，你可以通过 varnishadm 工具在本地执行缓存清除命令，而后者基于 HTTP 协议，这使得我们可以通过其他服务器上的应用程序来远程清除缓存，后面在介绍分布式计算的时候会再次提到它。

为了开启 Varnish 的缓存清除功能，我们需要对 Varnish 进行一些必要的配置，这里不做详细介绍，你可以查看详细的在线文档。

现在，我们可以基于命令行来清除某个 URL 的缓存，比如：

```
varnishadm -T localhost:8011 purge.url /test.htm
```

这里的 8011 端口是我们在 Varnish 启动时为它设置的管理端口。同时，我们也可以通过 HTTP 方式向 Varnish 的 HTTP 服务端口发送以下请求：

```
PURGE /test.htm HTTP/1.0
Host: www.highperfweb.com
```

这同样会清除/test.htm 在反向代理服务器上的缓存，当然，在 Varnish 的配置中，我们可以限定通过 HTTP 清除缓存的客户端 IP，你可以将它限定到内部网络中。

监控缓存命中率

也许你还在为刚才动态内容获得 7336.76 reqs/s 的吞吐率感到振奋，但理想和现实总是存在差距的，你要学会忍受现实的残酷，别忘了，我们压力测试中的动态内容都处于全缓存的情况下，也就是每次请求都命中缓存，这在现实中往往是不可能的。

首先，缓存区空间大小是有限的，而我们的站点可能有大量的内容需要被缓存，而不像前面压力测试时只有一个内容。一旦缓存区被装满，那么缓存管理器便会淘汰一些它认为不再需要的缓存内容，比如通过 LRU（最近最少使用算法）将使用频率较低的缓存内容淘汰出去，但是，这里判断“不常使用”的标准是不严格的，也许被淘汰的内容就是你将要访问的下一个内容，这便影响了它的命中率。

其次，缓存的过期时间也影响到它的命中率，假如有效期很短（为 10 秒），那么最少经过 10 秒便会有一次无法命中。

还有，有些内容可能根本没有被代理服务器缓存，比如这些内容包含了 set-cookie 等不可缓存的 HTTP 头信息，导致反向代理不会缓存它们，并且在浏览器请求它们的时候也不会去缓存区查找。这是影响命中率的一个重要因素，但往往容易被我们忽略。

幸运的是，这些问题我们都可以轻松地解决，前提是，我们需要了解反向代理缓存的实时工作状态，比如 Varnish 便提供了一个命令行的状态监控程序 varnishstat，我们打开它，便

看到了当前时刻的状态，如下所示：

```
Hitrate ratio:      1          1          1
Hitrate avg:     1.0000     1.0000     1.0000

      140163       0.00     191.22    Client connections accepted
      140161       0.00     191.22    Client requests received
      140158       0.00     191.21    Cache hits
           4       0.00       0.01    Cache misses
           4       0.00       0.01    Backend connections success
           4       0.00       0.01    Backend connections recycles
           1          .          .    N struct srcaddr
         101          .          .    N struct sess_mem
           4          .          .    N struct sess
         114          .          .    N struct object
         114          .          .    N struct objecthead
          49          .          .    N struct smf
           2          .          .    N small free smf
           1          .          .    N large free smf
           1          .          .    N struct vbe_conn
           1          .          .    N struct bereq
          46          .          .    N worker threads
         114       0.00       0.16    N worker threads created
        7478       0.00      10.20    N overflowed work requests
           1          .          .    N backends
           4          .          .    N expired objects
          11          .          .    N LRU moved objects
      140154       0.00     191.21    Objects sent with write
      140163       0.00     191.22    Total Sessions
      140163       0.00     191.22    Total Requests
           4       0.00       0.01    Total fetch
    37033469       0.00   50523.15    Total header bytes
  1307285268       0.00 1783472.40    Total body bytes
      140163       0.00     191.22    Session Closed
     4876953       0.00    6653.41    SHM records
      701723       0.00     957.33    SHM writes
        7042       0.00       9.61    SHM MTX contention
           2       0.00       0.00    SHM cycles through buffer
         122       0.00       0.17    allocator requests
```

看起来很强大，的确，它帮助我们获得了很多重要的信息，我们看黑体字部分，包括了以下几个重要的统计项：

Client requests received

代表到目前为止，浏览器向反向代理服务器发送的 HTTP 请求累积次数，由于可能使用长连接，所以它可能会大于上面的 Client connections accepted。

Cache hits

代表在这些请求中，反向代理服务器在缓存区中查找并且命中缓存的次数。

Cache misses

代表在这些请求中，反向代理服务器在缓存区中查找但是没有命中缓存的次数。

N expired objects

代表过期的缓存内容个数。

N LRU moved objects

代表被淘汰的缓存内容个数。

Total header bytes

代表缓存区中所有缓存内容的 HTTP 头信息长度。

Total body bytes

代表缓存区中所有缓存内容的正文长度。

通过以上这些统计项，我们还可以知道 Cache hits 和 Cache misses 的总和代表了反向代理服务器在缓存区中查找内容的总次数；而用 Client requests received 减去这个总次数，便大概等于没有查找缓存的请求数；Total header bytes 和 Total body bytes 的总和则代表了缓存区当前的空间使用量。

另外，Varnish 还将提供基于 Web 的监控界面，效果如图 8-1 所示。截至目前该功能还在开发中，你仅可以从 SVN 中获取它，也许当你看到本书时，该功能已经进入了 Varnish 的最新发布版本。

为了进行集成化的系统监控，我们还可以将 Varnish 的状态监控整合为 Cacti 的一部分，如图 8-2 及图 8-3 所示。Cacti 是一个非常开放的监控平台，通过编写脚本和 XML 视图可以很容易做到这一点。后面我们会详细介绍有关 Web 性能监控的方方面面，这部分内容是不可或缺的。

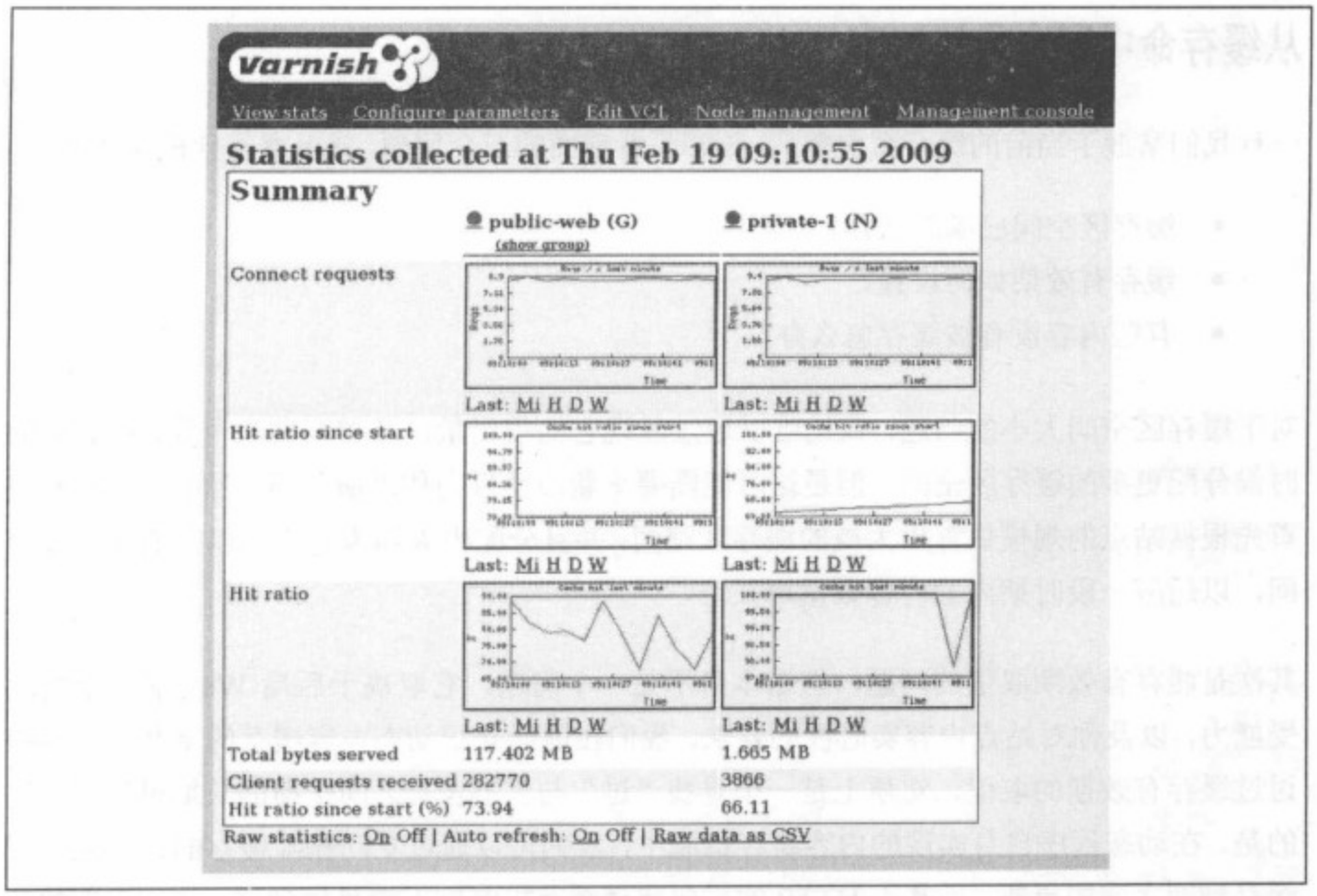


图 8-1　Varnish 将提供基于 Web 界面的状态监控

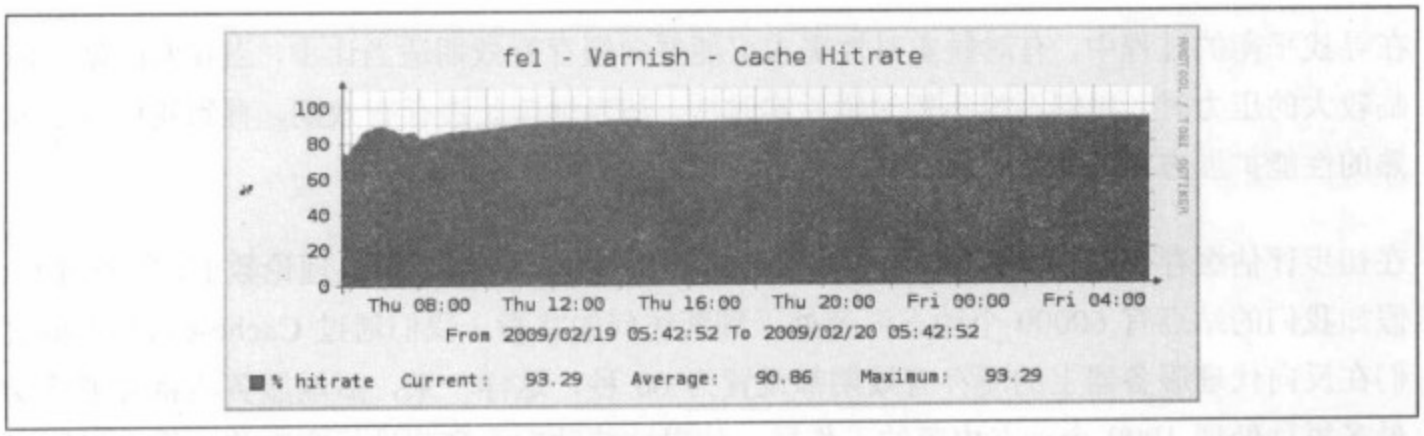


图 8-2　在 Cacti 上监控 Varnish 的缓存命中率

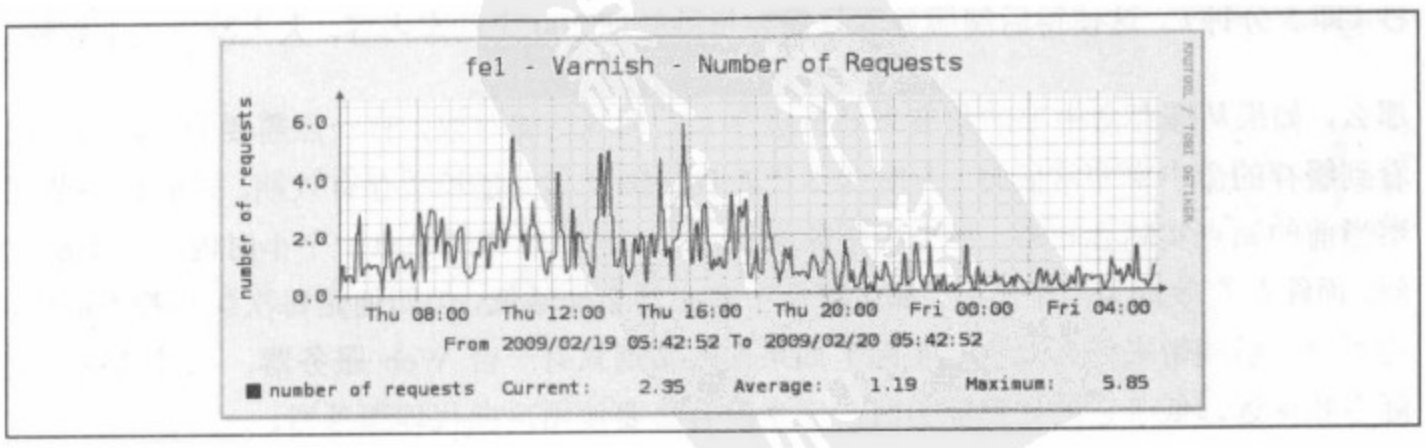


图 8-3　在 Cacti 上监控 Varnish 的吞吐率

从缓存命中率中寻找答案

一旦我们掌握了当前的缓存命中率后，再回头看前面的三个问题，就很容易分析和解决了。

- 缓存区空间已满怎么办？
- 缓存有效期如何设置？
- 有些内容没有被缓存怎么办？

对于缓存区空间大小的问题，我们可以通过监视它的使用量以及缓存淘汰个数，在必要的时候分配更多的缓存区空间，但是这可能需要重新启动反向代理服务器。当然，我们应该首先根据站点的规模估算出大概的缓存区空间，并且分配出大约为它 5 倍以上的缓存区空间，以适应一段时期内的内容数量增长。

其次是缓存有效期取值的问题，但这本身不是一个问题，它取决于后端 Web 服务器的承受能力，以及你对站点内容实时性的要求，我们在前面介绍动态内容缓存的章节中曾经探讨过缓存有效期的取值，实质上是一个寻找“过”与“不及”之间平衡的长期过程，幸运的是，在动态程序自身实现的内容缓存机制中，我们可以通过主动删除缓存的方法来实现缓存到期之前的更新，而基于 HTTP 的反向代理缓存机制则不容易做到这一点，后端的动态程序无法主动删除某个缓存内容，除非我们清空反向代理服务器上的缓存区。

在寻找平衡的过程中，有时候实时性需求可能要向缓存有效期适当让步，当我们的站点面临较大的压力时，我们不得不暂时牺牲实时性，适当地延长缓存有效期，直到我们有了成熟的性能扩展方案为止。

在初步评估缓存有效期的时候，我们可以通过一些量化的计算来得出理论数值，举个例子：假如我们的站点有 60000 个动态内容处于频繁访问的状态，我们通过 Cache-Control 将它们在反向代理服务器上的缓存有效期都设置为 60 秒，这样一来，后端服务器将必须承受最多每秒处理 1000 个动态内容的工作量，如果这些动态内容都进行完整的计算（如访问数据库），那么显然后端服务器是无法承受的，这时候我们可以将缓存有效期延长到 300 秒（即 5 分钟），这使得后端服务器只需要每秒处理 200 个动态内容，大大减少了工作量。

那么，如果从缓存命中率的角度来分析缓存有效期取值的时候，有一点需要明白，当我们看到缓存的命中率非常低时，有时并不代表我们需要马上延长缓存有效期，这时候还要观察当前的站点实际吞吐率，举个极端的例子，假如你发现你的站点 1 个小时内只有 1 次访问，而缓存有效期为半个小时，那缓存命中率必然会非常低，但即便是每次缓存都不命中，也不会对后端带来什么压力，实际上如果你的站点只有一台 Web 服务器，并且站点的实际吞吐率远远低于它的处理能力时，完全没有必要使用反向代理服务器。

最后关于内容没有被缓存的问题，我们也可以在 Varnish 的状态监视中找到线索，比如有一个用 Varnish 加速第三方应用 wordpress 的例子，得到的 Varnish 状态如下所示：

```
4+10:37:44                                                    my.server.com
Hitrate ratio:        1        1        1
Hitrate avg:     0.0364   0.0364   0.0364

      2324      0.00     0.01    Client connections accepted
      6191      0.00     0.02    Client requests received
        12      0.00     0.00    Cache hits
         7      0.00     0.00    Cache hits for pass
       318      0.00     0.00    Cache misses
      6179      0.00     0.02    Backend connections success
         0      0.00     0.00    Backend connections not attempted
         0      0.00     0.00    Backend connections too many
         0      0.00     0.00    Backend connections failures
      4057      0.00     0.01    Backend connections reuses
      6151      0.00     0.02    Backend connections recycles
      ...
```

我们看到，Varnish 一共处理了来自浏览器的 6191 个请求，其中命中缓存的有 12 个，真是太少了，而没有命中缓存的有 318 个，奇怪，剩下的那么多请求根本就没有去缓存区检查，也就是说，Varnish 认为那些内容不能被缓存。

不能被缓存总是有原因的，你需要根据反向代理缓存的规则来进行检查。而在我们这个例子中，都是 cookies 惹的祸，因为在 wordpress 中由于我们会安装一些各种各样的插件，有些插件会使得 wordpress 的每个页面都带有写入 cookies 的 set-cookie 标记，而我们前面在 vcl_fetch 函数中禁止了这类内容的缓存，问题就在这里了，我们希望将这些多余的 cookie 关闭掉，但是，wordpress 自身的登录和管理页面是需要 cookies 才可以正常工作的，所以我们还需要让反向代理不缓存这些页面，我们使用以下 VCL 配置：

```
sub vcl_recv
{
    if (!(req.url ~ "wp-(login|admin)"))
    {
        unset req.http.cookie;
    }
}
sub vcl_fetch
{
    if (!(req.url ~ "wp-(login|admin)"))
    {
        unset obj.http.set-cookie;
    }
}
```

可以看到，除了 wordpress 自身的登录和管理页面以外，我们将其他内容的 HTTP 头信息中有关 cookie 的标记全部都清除掉，这使得 Varnish 可以将大部分内容缓存起来，提高缓存命中率，同时不影响我们登录和管理 wordpress。

这个例子给了我们一些启示，对于我们自己的站点，如果从长远考虑，那么在规划的时候就要费点心思，我们可以根据内容是否可以缓存在反向代理服务器上，将它们置于不同的主机，这样便可以在必要的时候将可以缓存的内容快速与反向代理服务器对接，获得较好的加速效果。

缓存命中率和后端吞吐率的理想计算模型

从刚才的分析中，你也许已经发现了，缓存命中率和站点的实际吞吐率是有紧密关系的，而且还涉及了后端的吞吐率，听起来有点让人困惑，到目前为止我们已经发现了很多影响缓存命中率的因素，那么从计算的角度，起决定作用的因素都有哪些呢？它们对于后端服务器的吞吐率有什么影响呢？在谈论它之前，我们需要先营造一个理想的环境：

- 假设缓存区的空间足够大，那么缓存永远不会因为空间已满而被淘汰，除非到达缓存有效期后主动离开。
- 假设站点拥有一定数目的活跃内容，且这些内容都可以被反向代理缓存，同时我们假设一个平均缓存有效期。
- 假设以上的活跃内容在每个缓存有效周期内至少被访问一次。

接下来，我们来定义计算中需要用到的几个变量：

- 活跃内容数
- 实际吞吐率
- 平均缓存有效期

以上的实际吞吐率是指反向代理服务器处理用户请求时的实际吞吐率。而后端吞吐率，则是指后端 Web 服务器处理来自反向代理服务器的请求时的实际吞吐率。注意，这里说的实际吞吐率也包含另一层意思，就是需要达到的处理能力。

我们先来推出缓存命中率的理想计算模型，首先，一个缓存有效周期内的实际请求次数为：

一个缓存有效周期内的实际请求次数 = 实际吞吐率 × 平均缓存有效期

那么，在这个平均缓存有效周期内，反向代理服务器向后端服务器请求内容的次数为“活跃内容数”，所以，我们可以得出“缓存丢失率”的计算如下：

缓存丢失率 =(活跃内容数 /(实际吞吐率 × 平均缓存有效期)) × 100%

不难得出，缓存命中率的计算模型为：

缓存命中率 =1 –(活跃内容数 /(实际吞吐率 × 平均缓存有效期)) × 100%

我们用一个最简单的例子来说明，假如站点只有 1 个活跃内容，即 index.htm，实际吞吐率为 10reqs/s，即每秒有 10 次请求，同时我们将反向代理缓存有效期设置为 10 秒，那么可以计算出，缓存命中率为：

缓存命中率 = 1 – (1 / (10 × 10)) × 100% = 99%

我们再来看后端吞吐率如何计算，我们知道，后端接收的请求都来自反向代理服务器，根据前面的计算，你不难发现，在一个缓存有效周期内，后端吞吐率的计算模型为：

后端吞吐率 = 活跃内容数 / 平均缓存有效期

来测试一下吧，这次我们仍然使用刚才的例子，即 1 个活跃内容，平均缓存有效期为 10s，那么它的后端吞吐率为：

后端吞吐率 = 1/10 = 0.1reqs/s

只有 0.1reqs/s！事实让你不得不相信，反向代理服务器帮助后端做了大量的工作。

我们还可以将两个计算模型整合后，得到以下的模型，你可以看到缓存命中率和后端吞吐率的相互转化关系。

缓存命中率 = 1 – (后端吞吐率 / 实际吞吐率)) × 100%

后端吞吐率 = (1 – 缓存命中率) × 实际吞吐率

有了这些计算模型后，能说明什么问题呢？这是我们最关心的。我们来看看表 8-1，它列出了不同情况下的计算结果。

看到这些数据后，我想你对前面的计算模型会有更加深刻的认识，其实单从计算模型中，我们就可以得出以下结论：

- 活跃内容数和平均缓存有效期一定的情况下，缓存命中率与实际吞吐率成正比。
- 实际吞吐率和平均缓存有效期一定的情况下，缓存命中率与活跃内容数成反比。
- 活跃内容数和实际吞吐率一定的情况下，缓存命中率与平均缓存有效期成正比。
- 活跃内容数一定的情况下，后端吞吐率与平均缓存有效期成反比。
- 平均缓存有效期一定的情况下，后端吞吐率与活跃内容数成正比。
- 缓存命中率的变化不一定会影响后端吞吐率。
- 后端吞吐率的变化不一定会影响缓存命中率。

表 8-1　不同场景下的命中率和吞吐率计算结果

示例序号	活跃内容数	实际吞吐率	平均缓存有效期	缓存命中率	后端吞吐率
1	1	10reqs/s	10s	99%	0.1reqs/s
2	1	100reqs/s	10s	99.9%	0.1reqs/s
3	1	10reqs/s	100s	99.9%	0.01reqs/s
4	10	10reqs/s	10s	90%	1reqs/s
5	100	10reqs/s	10s	0	10reqs/s
6	100	100reqs/s	10s	90%	1reqs/s
7	100000	100reqs/s	3600s	72%	27.8reqs/s
8	100000	1000reqs/s	360s	72%	277.8reqs/s
9	100000	10000reqs/s	360s	97%	277.8reqs/s
10	100000	10000reqs/s	0s	0	10000reqs/s
11	100000	10000reqs/s	3600s	99.7%	27.8reqs/s

了解了上述内容后，可以避免我们在缓存命中率分析中产生一些错误的观点，比如认为缓存命中率越高，后端服务器的工作量就会越少。事实上，在表 8-1 的示例 7、8、9 中，我们看到缓存命中率同样为 72%的时候，后端的实际吞吐率增加了 10 倍；而当缓存命中率增加到 97%的时候，后端实际吞吐率并没有改变，可见，还有其他的因素需要考虑。

在实际情况中，活跃内容数和实际吞吐率并不由我们直接控制，站点内容数量不断膨胀，我们无法控制它的增长速度，但我们可以进行估算，实际吞吐率我们也无法控制，但可以进行测量，而缓存有效期我们可以控制它，从而调整缓存命中率和后端吞吐率。但是，对于缓存有效期的取值，必须服从站点内容的需求，我们不能一味延长有效期，有时站点的需求告诉我们，内容必须即时过期，比如表 8-1 中的示例 10。

归结到底，我们所关心的是前端和后端的实际吞吐率，因为它们决定了我们何时需要对站点规模进行扩展。结合表 8-1 中的示例，我们举几个例子：

- 对于示例 11，假设我们的一台反向代理服务器可以处理最高 6000reqs/s 的吞吐率，那么为了实现处理 10000reqs/s 的需要，我们就要考虑使用两台反向代理服务器来扩展处理能力。另外，假设我们的一台后端 Web 服务器对动态内容的处理可以达到最高 200reqs/s 的吞吐率，而后端目前只需要达到 27.8reqs/s 的实际吞吐率即可，那么我们只需要使用一台后端服务器，将前端的两台反向代理服务器同时指向这台后端服务器。

- 对于示例 10，前端同样需要达到 10000reqs/s 的吞吐率，我们仍然使用两台反向代理服务器，但此时后端只有一台 Web 服务器是不够的，为了实现 10000reqs/s 的需要，我们要使用多台后端服务器来扩展后端的处理能力，假设 1 台后端服务器可以处理最高 200reqs/s 的吞吐率，那么可能需要 50 台后端服务器，或许更多。

为此，我们可以将多台反向代理服务器指向同一台后端服务器，也可以将一台反向代理服务器指向多台后端服务器，它们分别如图 8-4 和图 8-5 所示。

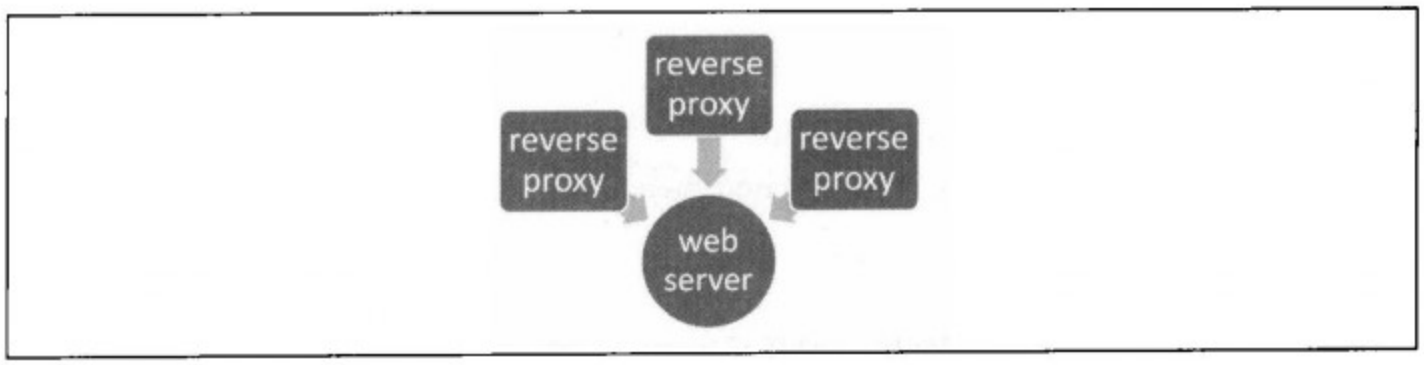


图 8-4　多台反向代理服务器指向同一后端 Web 服务器

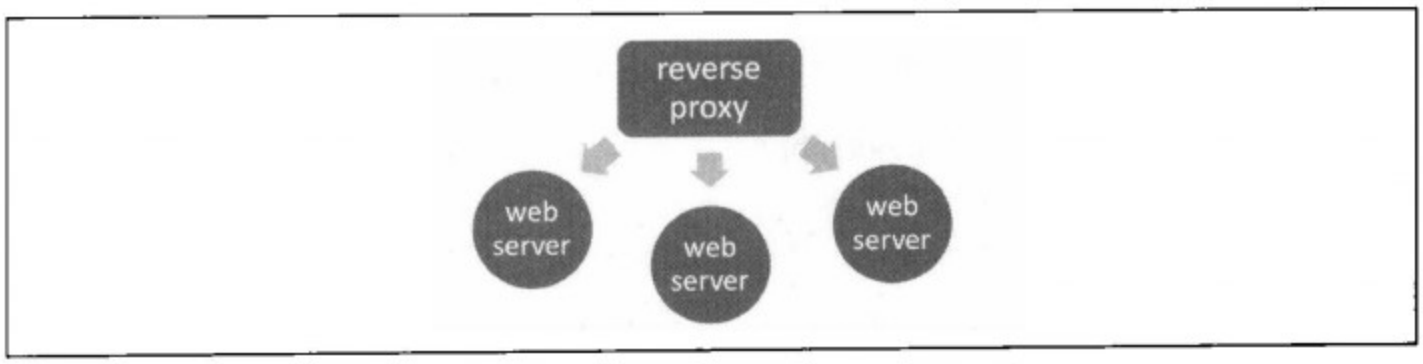


图 8-5　一台反向代理服务器的多个后端服务器

对于通过多台反向代理服务器来扩展站点处理能力，这实际上属于负载均衡的讨论范畴，我们在后面会有详细的介绍，这里你可以暂时认为这种扩展方法可以让多台服务器的处理能力简单累加。

ESI

在实际情况中，一个网页中各部分内容的更新频率各不相同，例如展示一篇新闻的网页中，新闻内容长期不变，但是旁边的推荐新闻列表却需要定时更新，那么，反向代理能否只向后端服务器请求推荐新闻列表这部分更新后的内容，而不需要获得整个网页的内容呢？这样便可以在一定程度上减轻后端服务器的开销。

正如你所想的，反向代理服务器可以做到这一点，前提是它必须实现 ESI（Edge Side Includes），ESI 是由 W3C 制定的标准，它的语法非常类似于 SSI（Server Side Includes），

可以像 SSI 一样在网页中嵌入子页面，但不同的是，SSI 是在 Web 服务器端组装内容，而 ESI 则是在 HTTP 代理服务器上组装内容，包括反向代理。

我们可以在 HTML 中添加 ESI 语句，比如一个简单的 ESI 语句如下所示：

```
<esi:include src="/included_page.htm" />
```

Varnish 便支持 ESI，我们可以用它来实现网页局部缓存，接着刚才新闻网页的例子，我们做一个简单的原型，假如新闻网页为 news.php，它的内容如下所示：

```
<HTML>
<BODY>
<!-- 省略新闻内容 -->
推荐新闻: <esi:include src="/recommend.php" />
</BODY>
</HTML>
```

虽然内容比较简陋，没有任何修饰，但是足以说明一切，我们看到 news.php 里嵌入了一个子页面 recommend.php，它是一个动态内容，由后端服务器生成。接下来，我们还需要在 VCL 中进行相应的配置，用来对不同的内容设置不同的缓存有效期，如下所示：

```
sub vcl_fetch
{
    if (req.url == "/news.php")
    {
        esi;
        set obj.ttl = 24h;
    }
    elseif (req.url == "/recommend.php")
    {
        set obj.ttl = 1m;
    }
}
```

可以看出，我们为 recommend.php 指定了较短的有效期（1 分钟），而对 news.htm 指定了 24 小时的有效期，这便达到了我们的目的。

在处理只有局部更新的动态内容时，这种方案的确是不错的选择，不过它过于依赖 ESI，一旦你的反向代理服务器不支持 ESI，那么一切都毫无意义。另一种更好的选择是利用 AJAX（Asynchronous JavaScript and XML）将需要频繁更新的局部内容采用异步请求的方式，这种做法更加容易控制，你唯一需要注意的问题是 AJAX 的跨域问题，你的局部内容应该和父页面所在的主机保持相同的顶级域名，其他有关 AJAX 编程方面的知识，我想你应该已经非常熟悉了。

另外，对于原本大量使用 SSI 的内容，可以非常快速地迁移到 ESI，因为它们的语法和需要的页面结构都非常相似，这样一来，内容的组装由 Web 服务器转移到反向代理服务器上，这本身没有绝对的好处，就像原本后端服务器是运输组装好的成品到反向代理服务器，

而现在只运送原材料，由反向代理服务器来负责组装，总的工作量没有什么变化，但是在有多个后端服务器的情况下，便可以避免多个后端的重复组装，减少总工作量。

和动态内容缓存一起工作

当我们为后端的动态内容应用了反向代理缓存后，动态内容本身的缓存是否变得多余了呢？当然，如果你发觉它们完全在做同样的工作，你可以尝试关闭动态内容自身的缓存，但是，在包括以下在内的一些情况中，它们二者完全可以和睦相处，形成多级缓存，更加有效地提升站点性能。

备用缓存

仔细检查动态内容是否如你所想的一样被反向代理缓存，实际上开发人员很容易犯一些错误，导致反向代理缓存对动态内容视而不见。如何检查呢？我们永远不会对站点中每一个动态程序都进行压力测试，以观察它是否被反向代理所缓存，一个简单的办法是前面提到的通过反向代理状态监控来分析缓存命中率，从而了解它是否达到你预期的区间，同时也可以观察后端服务器的实时吞吐率和访问日志等，另外，如果允许的话，也可以编写自动化测试用例，自动覆盖每一个动态内容，检查它们的 HTTP 响应头是否符合缓存规范。

但是，这一切的检查有时需要较长的周期，在发现问题之前，我们希望它的影响范围降到最低，所以，当我们没有十足的把握时，启用动态内容自身的页面缓存便引入了另一道防线，我喜欢称它为“备用缓存”，你也可以把它看成是反向代理服务器的 L2 Cache（二级缓存）。

暴露后端

为什么会暴露后端的 Web 服务器呢？通常我们希望后端服务器可以隐藏在反向代理之后，但在有些时候，比如反向代理主要用于跨地域加速，这时候反向代理服务器和后端服务器可能位于不同的数据中心，它们通过基于 DNS 策略的负载均衡分别服务于不同地域的用户，这时候，后端服务器的页面缓存也必须同样正常的工作。

首次加载

任何时候都不要忘了反向代理在首次加载一个内容或者缓存过期的时候，仍然需要向后端服务器获取，总是有一个不幸的用户会执行缓存预热的工作，或许这是舍己为人的感人事迹，不过没有用户觉得这是一件光荣的事情，试想一下假如有个用户在清晨连续访问 20 个动态内容，恰好都没有命中缓存，而后端服务器也毫无准备，只能重新计算内容返回给反向代理，而这花费了不少的时间，用户开始狂躁。如果你不希望有任何用户抱怨，那么同时启用后端动态内容的页面缓存便显得尤为重要，尤其是采用静态化方式，这样便可以

让用户即使在没有命中反向代理缓存的时候，也可以享受后端缓存的优待。

多台反向代理服务器

还记得前面介绍过的多台反向代理服务器指向同一台后端服务器吗？在这种情况下，后端服务器上同样的内容可能会接到来自前端每一个反向代理服务器的请求，如果后端没有缓存机制，这就相当于刚才所说的“首次加载”被重复上演，同时成倍地增加后端服务器的工作量。很显然，为后端的动态内容使用页面缓存也显得尤为重要。

8.4 小心穿过代理

我们知道，反向代理服务器充当了用户和后端 Web 服务器的中介，它只是将用户的 HTTP 请求转发给后端服务器，使得后端服务器知道用户的意图。但是，用户的有些信息并不存在于 HTTP 请求中，所以它们对后端服务器是不可见的。比如用户的 IP 地址和发送请求的 TCP 端口，它们分别位于分层网络模型中的 IP 层和传输层，反向代理服务器可以轻易地获得它们，而后端服务器却无法直接获得。

对于一些动态程序，获取用户的 IP 可能至关重要，虽然这与性能本身无关，但这是由于性能优化可能产生的副作用，我们希望反向代理能够将用户 IP 地址也转发给后端服务器，它是怎么做到的呢？

我们编写一个非常简单的 PHP 程序，用于打印当前的服务器环境变量，它的代码如下所示：

```
<?php
var_dump($_SERVER);
?>
```

下面我们直接访问 Web 服务器上的这个动态程序，获得的结果中包含以下我们感兴趣的内容：

```
["SERVER_ADDR"]=> string(13) "60.215.129.26"
["SERVER_PORT"]=> string(2) "80"
["REMOTE_ADDR"]=> string(13) "220.231.2.123"
["REMOTE_PORT"]=> string(5) "41997"
```

很显然，这是 Web 服务器获得了用户的 IP 地址（REMOTE_ADDR）和发送请求的 TCP 端口（REMOTE_PORT），如果你了解网络编程，也可以做到这一点。下面我们通过 Varnish 来请求这个动态程序，我们的请求发给了 Varnish，获得的结果中我们关注的部分如下所示：

```
["SERVER_ADDR"]=> string(9) "10.0.1.10"
["SERVER_PORT"]=> string(4) "8010"
```

```
["REMOTE_ADDR"]=> string(9) "10.0.1.11"
["REMOTE_PORT"]=> string(5) "51669"
["HTTP_X_FORWARDED_FOR"]=> string(13) "220.231.2.123"
```

这时候我们看到 Web 服务器获得的客户端 IP 为 10.0.1.11，也就是 Varnish 所在服务器的内部网络地址，而在下面多出了一个服务器变量，它便是 Varnish 在转发请求时添加的标记，代表了用户的 IP 地址，我们看到它和刚才直接访问 Web 服务器看到的是一样的。

这样一来，置于后端的动态程序只需要通过以下变量便可以获得用户的 IP 地址：

```
$_SERVER["HTTP_X_FORWARDED_FOR"]
```

当然，不是所有的反向代理服务器都会这样，有些时候需要你提醒它如何来做，比如在 Nginx 做反向代理时，我们需要进行以下配置：

```
location / {
  proxy_pass        http://localhost:8001;
  proxy_set_header  X-Real-IP  $remote_addr;
}
```

这样一来，我们通过 Nginx 访问后端的动态程序便可以得到以下的服务器变量：

```
["HTTP_X_REAL_IP"]=> string(13) "220.231.2.123"
```

总之，对由于穿过反向代理而引发的此类问题，我们都可以通过让反向代理请求后端服务器时携带附加的 HTTP 头信息来实现。同样的，如果后端服务器想要告知浏览器一些额外的信息，也可以通过在响应 HTTP 头信息中携带一定的自定义信息穿过反向代理。比如当同时存在多个后端服务器时，我们在动态程序中添加以下的 HTTP 头信息：

```
header("X-Real-Server-IP: 10.0.1.10");
```

这样一来，我们通过反向代理访问该内容时，浏览器便会获得以下的 HTTP 响应头信息：

```
HTTP/1.1 200 OK
Date: Sun, 29 Mar 2009 15:18:48 GMT
Server: Apache/2.2.11 (Unix) PHP/5.2.1 DAV/2 SVN/1.4.3
X-Powered-By: PHP/5.2.1
X-Real-Server-IP: 10.0.1.10
Content-Length: 1910
Keep-Alive: timeout=30, max=100
Connection: Keep-Alive
X-Varnish: 1202470754
Age: 0
Via: 1.1 varnish
Content-Type: text/html
```

也许你不想暴露后端服务器的内部 IP 地址，没关系，这只是一个例子，但在有些时候，它为我们的某些调试工作提供了帮助。

8.5 流量分配

我们知道，反向代理服务器担当了用户端和后端的枢纽，它需要同时和两端进行数据交换，这使得它流入和流出的数据量要大于仅使用 Web 服务器时的数据量。

如果反向代理服务器和后端服务器通过外网交换机进行数据交换，那么这部分流量必然会消耗其用于对外服务的带宽，回忆一下前面介绍的带宽知识便不难得到这一结论。

这时候，我们可以组建内部私有网络，如图 8-6 所示，目的是让反向代理服务器和后端服务器通过内网交换机来交换数据，同时，我们需要为反向代理服务器和后端服务器都配备双网卡，并设置私有 IP 地址，这些都很容易，市面上的标准服务器机型一般都至少配置两个网卡。

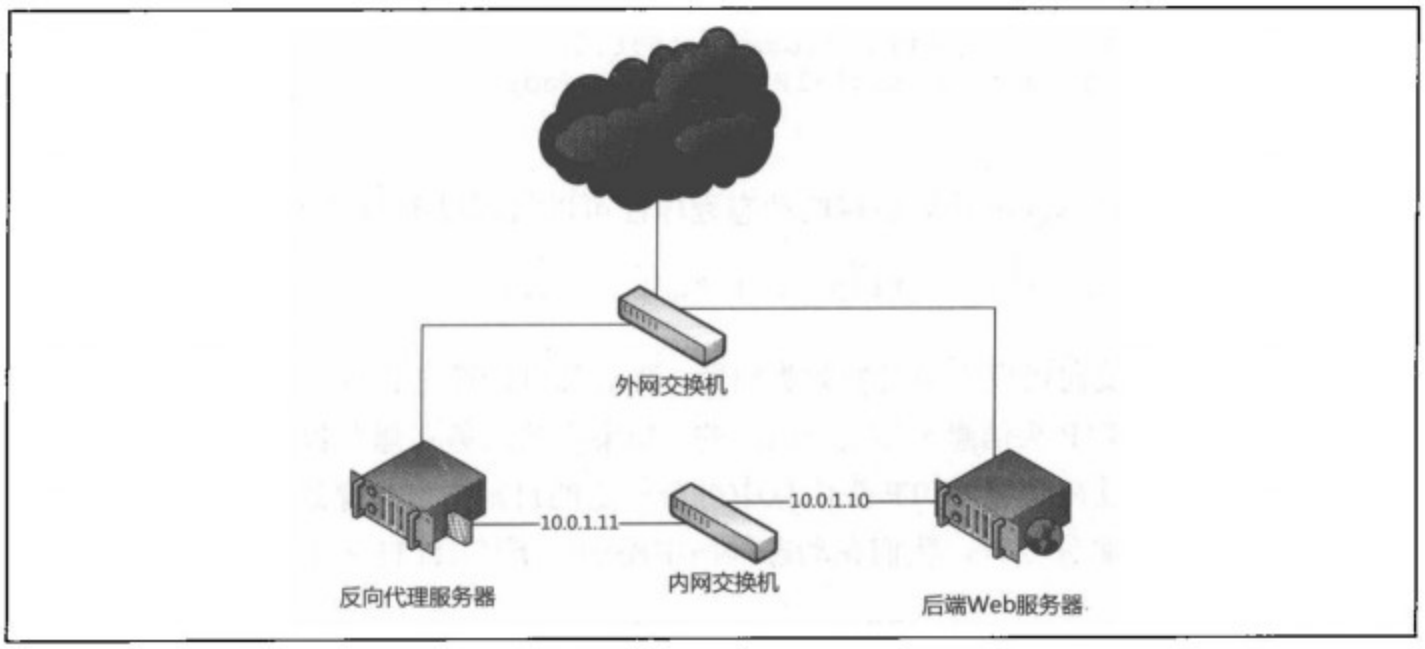


图 8-6　反向代理服务器和后端服务器通过内部网络交换数据

这样一来，我们在 Varnish 的配置中可以将后端服务器的地址设置为内部网络地址，比如：

```
backend default {
    .host = "10.0.1.10";
    .port = "80";
}
```

另外，如果你的反向代理服务器是通过域名指向后端服务器，那么你可以通过修改 /etc/hosts 来重新定位内部网络中的后端服务器，比如：

```
10.0.1.10    www.highperfweb.com
```

对于商用数据中心，一般是按照机位和带宽来收取费用，一旦你将服务器部署在数据中心，便可以将其接入外网交换机，获得一个固定 IP 地址。同时，你也可以自己动手组建内部网络，将自己的服务器群进行合理的规划。

第 9 章

Web 组件分离

迄今为止，我们探讨了很多提升站点吞吐率的具体做法，也许它们让你感到颇为受益，但是请不要满足，因为只有这些方法是远远不够的，接下来的内容同样重要。

世界上最愚蠢的事情莫过于做了一件不该做的事，这就像用错药一样，令人感到无比沮丧。同样，当我们引以为荣地应用某种方法时，你是否认真的思考过，它在这里真的有效吗？

9.1 备受争议的分离

当你还在中学的时候，是否经历过快班和慢班的抉择呢？至少我没有，不知道是否算是幸运。如今，很多中学甚至在高一就开始分班，根据摸底考试结果，优生进入快班，差生进入慢班，显然，这么做备受争议。

支持者多数为校方、优生、优生的家长，他们认为分班的好处主要在于：

- 为优生安排最好的老师和强化课程，保证他们能考上重点大学。
- 为差生安排难度较低的课程，帮助他们至少能考上大学。
- 避免差生对优生的不利影响。

反对者多数为差生和差生的家长，他们认为：

- 分班使得慢班学生不被学校重视，失去考入重点大学的机会。
- 慢班学生得不到优生的标杆作用，缺乏前进的动力。

尽管学校承诺慢班的学生可以通过定期的考试获得升入快班的机会，但是，往往这种可能性不大，也只能算是对慢班学生的一种安慰罢了。

那么，到底应不应该分班呢？我们暂且不谈这个社会问题，仅仅从分班策略的出发点来看，它的背后是有一定道理的，那就是针对不同的学生进行差异化教学，使他们可以扬长避短，获得各自的最佳发展，这也正是“因材施教”的思想。

提示：

宋代朱熹在《论语》的注解中指出："孔子教人，各因其材"，这也正是"因材施教"的来源。《论语》中记载，孔子对自己的学生很了解，他能够说出学生的性格特点和智力水平，并且针对不同学生的特点，用不同的方法进行教育，把学生培养成为各种不同的人才。孔子曾经说过"冉有为人懦弱，所以要激励他的勇气。子路武勇过人，所以要中和他的暴性"。

看来，两千多年前的圣贤就流行这种教学方法，而且效果显著，是的，因材施教本身是没错的，问题在于，现在是如何区分优生和差生的呢？一个摸底考试能比得上孔子的慧眼吗？结果往往比较可悲，有些慢班学生自暴自弃，甚至出现忧郁和自杀倾向，实在可惜。

所以，能否正确地执行"因材施教"，关键在于教师能否真正了解不同学生的能力和特点，以及能否找到有效的方法来区分他们。

9.2 因材施教

正如教师需要了解不同学生的能力和特点一样，作为架构师的你，是否真的了解你的站点中各种 Web 组件的特点和差异呢？一旦你发现它们的特点截然不同，"因材施教"便在所难免，针对它们的不同特点，回顾前面章节中介绍的方法，如何让这些 Web 组件达到吞吐率最大化呢？

这里需要说明的是，如果你对"Web 组件"的概念感到陌生，可能是由于名字约定的习惯差异，这没什么关系，我们这里所指的 Web 组件是指 Web 服务器提供的所有基于 URL 访问的资源，比如动态内容、图片、JavaScript 脚本、CSS 样式表等。

好，我们从以下几个方面来看这些 Web 组件的差异：

- 文件大小
- 文件数量
- 内容更新频率
- 预计并发用户数
- 是否需要脚本解释器
- 是否涉及大量 CPU 计算
- 是否访问数据库
- 访问数据库的主要操作是读还是写
- 是否包含远程调用（RPC）

值得一提的是，从以上这些方面来看，即便是同一类 Web 组件，显然也存在特点的差异。

比如负责呈现内容的动态网页和负责用户注册的动态网页，我们可以将它们视为不同种类的 Web 组件，再比如由用户上传的大尺寸照片和站点网页中的小尺寸修饰图片也存在以上方面的差异，我们也将它们区分对待。

这样一来，你的脑海中已经浮现出了不同种类的 Web 组件，每种 Web 组件都有着相似的特点，接下来得考虑给它们分别采取什么样的优化方法呢？这些方法可能包括以下条目中的一种或多种：

- 是否使用 epoll 模型
- 是否使用 sendfile()系统调用
- 是否使用异步 I/O
- 是否支持 HTTP 持久连接（HTTP Keep-alive）
- 是否需要 opcode 缓存
- 是否使用动态内容缓存以及有效期为多长
- 是否使用 Web 服务器缓存以及有效期为多长
- 是否使用浏览器缓存以及有效期为多长
- 是否使用反向代理缓存以及有效期为多长
- 是否使用负载均衡策略

不可否认，为不同特性的 Web 组件提供有针对性的最佳优化方法是一件费脑筋的事情，但是，我们已经迈出了第一步，Web 组件的分离为“因材施教”提供了可能。

同时，我们需要清楚地认识到，Web 组件分离的目的是便于采用针对性的方法，使得各种 Web 组件能够充分有效地利用服务器资源，达到符合各自实际情况的吞吐率最大化。

9.3 拥有不同的域名

那么究竟如何分离 Web 组件呢？这里不得不说，它跟“分班”没什么两样，但是人们对它几乎没有任何争议。

为了给不同类型的 Web 组件采取有针对性的措施，我们将这些 Web 组件分别进行独立部署，它们可能位于不同的物理服务器，或者同一个物理服务器上的不同逻辑单元中，同时，我们将不同的域名指向不同的 Web 组件服务器。

举个例子，假如我们的站点尚未采用 Web 组件分离，它只有一个域名：

```
www.highperfweb.com
```

这个域名指向一个服务器 IP 地址，假设为：

```
10.0.2.1
```

同时，我们在这个服务器上开启了 Apache，通过 80 端口对外提供服务。可想而知，这个站点的所有内容都需要通过请求 Apache 来获得。

站点中有一个名为 colin_photo.html 的静态文件，它的内容如下所示：

```
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
    <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <script src="/js/app.js" type="text/javascript"></script>
    <link href="/styles/layout.css" rel="stylesheet" type="text/css" />
    </head>
    <body>
    <img src="/images/logo.gif" />
    <img src="/upload/photo20090401.jpg" />
    </body>
    </html>
```

我们可以通过以下地址来访问它：

```
    http://www.highperfweb.com/colin_photo.html
```

很好，就这么简单，但是现在我们希望对 Web 组件进行分离，至于为什么，我们稍后探讨，先来看看分离后的样子。

在分离前，我们需要为组件服务器准备一些独立的新域名，但通常你不需要购买新的主域名，我们可以使用站点现有主域名的二级域名，只需要在域名管理系统中设置新的 A 记录即可，这难不倒你。

瞧，我们创建了以下几个新域名，它们让我的站点看起来变得很有品位，至少我这么觉得。

img.highperfweb.com

存放网页中的插图，指向新的服务器 10.0.2.2。

upload.highperfweb.com

存放用户上传的照片，指向新的服务器 10.0.2.3。

static.highperfweb.com

存放静态化的网页，指向新的服务器 10.0.2.4。

js.highperfweb.com

存放网页中链接的 JavaScript 脚本文件，指向新的服务器 10.0.2.5。

现在，我们需要将对应的 Web 组件迁移到各自的服务器上，注意，它们相对于 Web 根目录仍然保持原有路径，这便于我们修改组件 URL。

接下来，我们需要修改 colin_photo.html 中 Web 组件的 URL，如下所示：

```
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
    <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <script     src="http://js.highperfweb.com/js/app.js"type="text/java
script"></script>
    <link href="http://img.highperfweb.com/styles/layout.css" rel="style
sheet" type="text/css" />
    </head>
    <body>
    <img src="http://img.highperfweb.com/images/logo.gif" />
    <img src="http://upload.highperfweb.com/upload/photo20090401.jpg" />
    </body>
    </html>
```

现在，让我们来访问 colin_photo.html。

```
    http://static.highperfweb.com/colin_photo.html
```

你已经看出来了，我们将 colin_photo.html 也转移到了新的服务器上。

看起来不难，我们只是在 HTML 网页中修改了组件的 URL 便完成了组件分离，是的，我不得不感叹 HTTP 的伟大，从它诞生的第一天开始，毫无疑问，它就是分布式协议。

需要注意的是，当你使用站点二级域名作为组件服务器地址，并且将站点的 cookies 作用域设置为顶级域名时，则每次浏览器请求图片等组件都会将本地的 cookies 附带在 HTTP 请求头信息中发给 Web 服务器，这显得非常多余，而且增加了 HTTP 头信息的长度，我们不希望这样。

一个解决的办法是将 cookies 的作用域缩小，比如只限定为 www.highperfweb.com，以下是一个 HTTP 响应头中设置 cookies 的例子，它包括了对 cookies 作用域的限定：

```
    Set-Cookie:  ticket=e96ddca12c9bb30b98706e1ead00d986;  expires=Mon,
04-Apr-2011 13:53:40 GMT; domain=www.highperfweb.com
```

但是，当站点中有多个二级域名需要使用 cookies 的时候，这种方法就显得不是那么方便了。另一种解决办法也比较常用，那就是为这些比较独立的组件启用新的顶级域名，比如 www.highperfweb-img.com，这样一来，来自 www.highperfweb.com 的 cookies 便不会再来主动上报。

在 Web 组件分离的背后，有一个重要的问题引发我们的思考，我们希望能够自动地将一些内容迁移到组件服务器上，比如在用户上传照片后，将照片快速迁移到照片服务器，同时让用户不要感到延迟，尤其在大量用户上传照片时，做到这一点的确是件富有挑战的事情，在后面有关文件分发和同步以及分布式文件系统的章节中，我们会详细探讨这个问题。

顺便一提的是，这种 Web 组件分离的方法，本质上也属于一种负载均衡的策略，以实现站点规模扩展，它把一系列对 Web 组件的请求进行垂直分割，分别指向不同的组件服务器。后续章节中我们会详细介绍负载均衡，我们甚至可以在各种组件服务器内部再次实现负载均衡或者集群，那么，这里描述的 Web 组件分离便放大成为组件集群之间的负载均衡。

9.4 浏览器并发数

当我们用不同的域名对 Web 组件进行分离后，另一个好处随之而来，那就是提高了浏览器在下载 Web 组件时的并发数。

我们知道，当用浏览器打开一个网页的时候，浏览器首先下载网页本身，也就是 HTML，然后分析这些 HTML 标记，同时逐步下载其中包含的一系列组件。然而，浏览器下载组件的过程受到最大并发数的限制，也就是浏览器同一时刻最多只可以下载一定数量的组件，不同的浏览器拥有不同的默认限制，表 9-1 列出了常见的几款浏览器分别对于 HTTP/1.1 和 HTTP/1.0 的默认最大并发数。

表 9-1 常见浏览器的最大并发数限制

浏览器	HTTP/1.1	HTTP/1.0
IE 6,7	2	4
IE 8	6	6
Firefox 2	2	8
Firefox 3	6	6
Safari 3,4	4	4
Chrome 1,2	6	—
Opera 9.63,10.00alpha	4	4

那么，为什么 Web 组件分离后会提高浏览器的并发数呢？其实，浏览器的最大并发数限制有一个前提，那就是对于同一个域名下的组件才有效，也就是说，浏览器会为每个域名维护不同的下载队列，每个队列的最大并发数限制均为表 9-1 中列出的数值，这些队列同时运行。

这样一来，浏览器的下载并发数将会增多，整体下载速度也随之提高。

当实施了组件分离后，你可以使用 HttpWatch 等浏览器监视程序来观察你的站点，你会发现不同域名下的组件下载过程互不影响，整体上提高了浏览器的下载速度。HttpWatch 目前只支持 IE 和 Firefox 浏览器，如果你希望使用其他浏览器来测试组件分离带来的浏览器下载速度提升，可以试试一个在线工具，它的地址是 http://site-perf.com。

如图 9-1 所示，它可以模拟不同的并发数（从 2 到 10），并且可以像 HttpWatch 那样进行 HTTP 跟踪，甚至包括更加详细的分析，如图 9-2 所示。

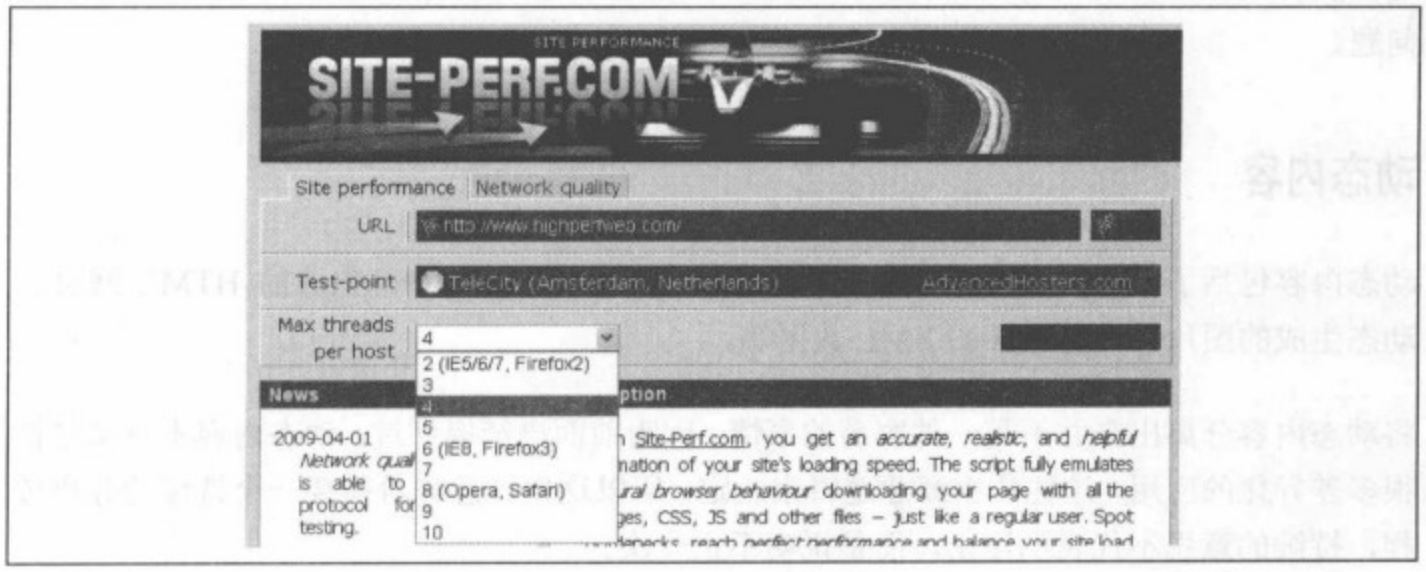


图 9-1　通过在线工具 site-perf.com 来模拟浏览器并发数

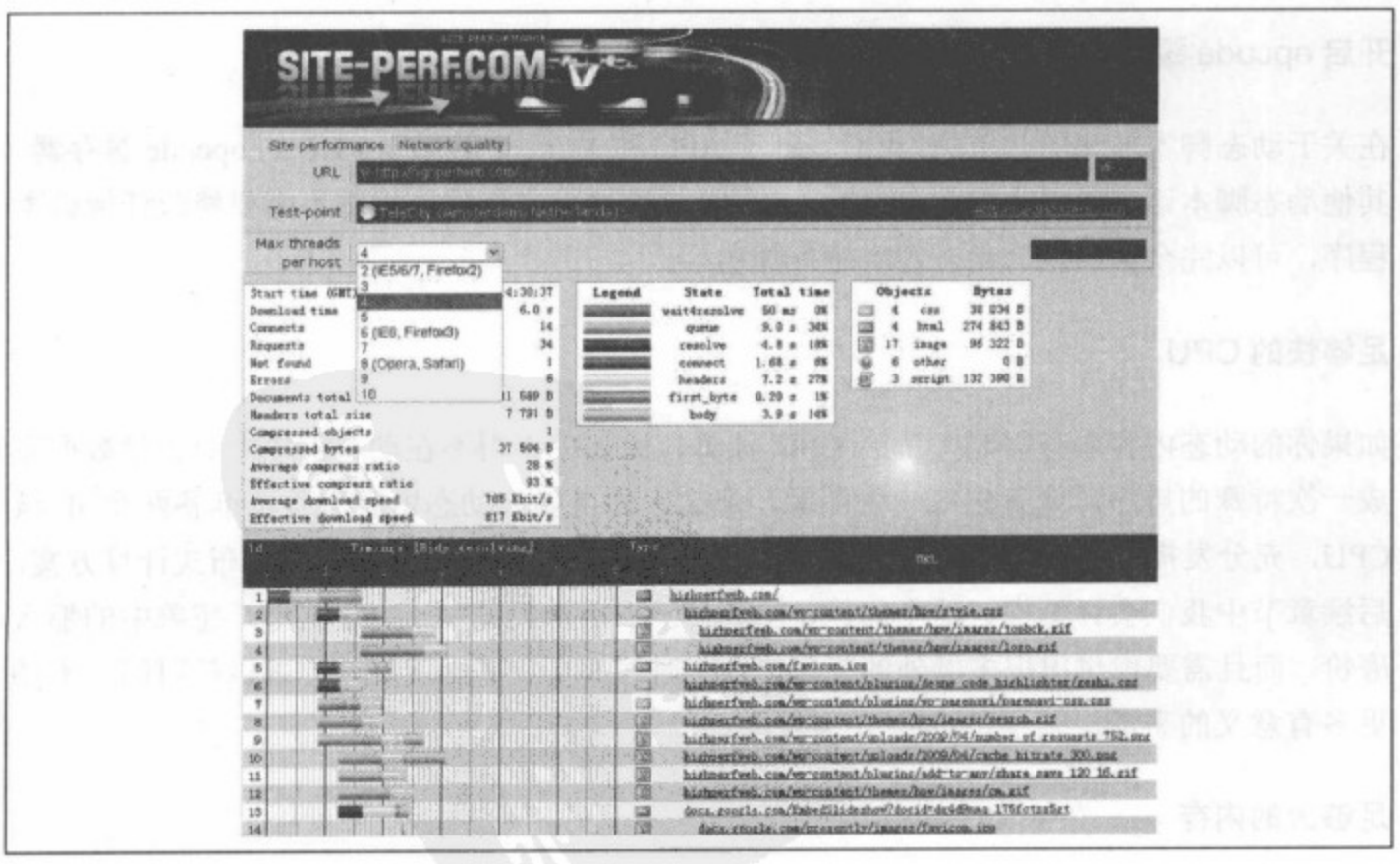


图 9-2　通过在线工具 site-perf.com 进行 HTTP 跟踪

限制并发数的本意在于防止浏览器由于同时下载大量的内容，特别在同时打开多个网页时影响到用户在计算机上的其他工作，

可见，Web 组件分离使得服务器端和浏览器端同时受益，可谓是双管齐下。

9.5 发挥各自的潜力

你已经知道了如何识别和分离 Web 组件，现在我们分别来看看这些 Web 组件，为了让它们获得最佳的性能，我们需要如何来做呢？这也正是“因材施教”中如何“施教”的问题。

动态内容

动态内容包括了一切需要动态脚本在运行时创建的内容，比如动态生成的 HTML 网页、动态生成的图片、动态生成的 XML 数据等。

将动态内容分离出来并不是一件容易的事情，因为前面已经提到过，动态内容本身又包括很多差异化的应用，这往往由商业逻辑来决定，所以这也许意味着需要一个持续的分离过程，持续的意思不代表不停止，而是说明不止一次。

无论如何，对于所有动态内容，我们希望提供给它们良好的发展空间，这包括以下内容。

开启 opcode 缓存

在关于动态脚本加速的章节中，我们看到了 APC 和 XCache 等基于 PHP 的 opcode 缓存器，其他动态脚本语言同样也拥有自己的 opcode 缓存器，它们的应用都不需要修改任何脚本程序，可以完全透明地为动态内容进行加速。

足够快的 CPU

如果你的动态内容本身需要大量的 CPU 计算，比如你不得不在动态程序中对大量数据完成一次特殊的排序，或者生成一张图像，那么，你可以为动态内容服务器准备两个 4 核 CPU，充分发挥 SMP（对称多处理器）的优势。也许你希望采用廉价的分布式计算方案，后续章节中我们会详细探讨这个问题，但是对于中小规模的站点，它并不是想象中的那么廉价，而且需要你付出很多额外的工作，有些时候简单、够用就好，你可以把时间用来做更多有意义的事情。

足够大的内存

动态内容的运行需要不少的内存开销，一旦物理内存不够用，内存管理器会使用 swap，

这导致磁盘和内存之间发生频繁的数据复制，这可不好玩，我们需要坚决避免。通过前面介绍过的 vmstat 等系统工具，以及后面要介绍的性能监控系统，我们可以很好地观察到物理内存和 swap 的使用情况，但是，在一台运行中的服务器上扩展内存可不是容易的事情，你需要做好提前的打算，比如为服务器配备 4GB 的物理内存。

多进程

与动态内容自身的开销相比，多进程切换的开销微不足道，尽管多进程需要消耗大量的内存，但是它也带来了很多好处，这包括可以同时处理更多的动态内容、整体上减少 I/O 等待时间，以及获得进程的稳定性。这些进程可以是 Web 服务器进程或者 fastcgi 进程。

与数据库保持高速连接

大多数的动态内容都需要涉及数据库访问，这相当于一种远程 I/O 操作，它同样存在等待时间，但是不同于本地 I/O 等待时间，它还包括网络 I/O 的等待时间，前面我们曾经介绍过网络传输中响应时间的计算方法。将动态内容服务器与数据库服务器保持高速连接，可以最大化地减少网络 I/O 等待时间。

最理想的情况就是它们位于同一台物理服务器上，小规模站点完全可以这样做，比如用 Wordpress 搭建的一个 blog，动态内容和数据库完全可以通过 UNIX Socket 来建立更加快速的数据交换。

对于稍具规模的站点，动态内容服务器只能通过 TCP 通信来与数据库服务器建立连接，通过内部专用的百兆交换机来为它们提供足够的带宽和顺畅的通信链路，达到高速的数据库访问。同时，你还需要做的是，进行必要的网络拓扑结构规划，考虑未来可能存在的内部网络扩展。总之，我们希望将动态内容服务器部署在离数据库最近的地方。

但是，仍然有一些现实情况在所难免，我们需要想办法去适应，比如动态内容服务器和数据库服务器部署在不同的数据中心，这可能是因为有些动态内容服务器需要进行地理位置就近部署，虽然我不建议这样做，但是如果一定要这样做的话，租用连接两地数据中心的 DDN 专线，也可以提供相对较快的响应速度，不过它的价格不菲。

没钱怎么办？我们可以用其他廉价的解决办法，比如将动态内容的静态副本分发到离用户较近的数据中心，而不是将计算过程整体搬过去，这就好像一家食品生产公司只需要将生产好的食品运到各地的超市去销售，而不需要将生产工厂也搬到超市去，那样的话原料的运输成本巨大。

可靠的数据中心

几乎没有人不希望数据中心可靠，但这背后往往是经济成本的问题，即便是在租用同样带

宽的情况下，国家级数据中心要比普通数据中心的价格高出很多，动态内容往往在站点中承担着重要的职责，即便是我们看到站点大量采用静态网页，但其背后的 CMS（内容管理系统）也非常重要。所以，尽量选择可靠的数据中心来部署动态内容。

以上是对动态内容一些通用的建议，另一方面，为了对不同的动态内容采取更加富有针对性的措施，我们还将功能不同的动态内容划分到不同的域名和服务器上，使得它们相对独立。

举个简单的例子，比如我们已经将站点动态内容放在反向代理的后端，并且对一些动态内容使用了反向代理缓存，这时候我们推出了站点的 API 服务，我们不希望给 API 服务加上任何缓存，也不需要从反向代理服务器转发请求，所以我们将 API 服务部署在另一台服务器的 80 端口上，并且准备了新的域名：

```
api.highperfweb.com
```

这样一来，API 服务和其他的动态内容便看起来完全独立，尽管它们也许访问同一个数据库，但是这丝毫不影响它们的分离。

我们来看看 Google 对一些动态内容的分离，它将不同的服务分别部署在以下的域名：

```
ditu.google.cn
images.google.cn
news.google.cn
video.google.cn
translate.google.cn
shenghuo.google.cn
```

这些服务想必你都有体验，分离还带来了另外一些好处：

- 域名更具有可读性，用户容易记忆。
- 有利于各种服务的独立访问量统计。
- 有利于各服务的独立扩展，这涉及后面要介绍的基于 DNS 的负载均衡。

当站点压力较小时，即便是将不同服务的域名指向同一台物理服务器，它同样也能带来上述这些好处。

静态网页

静态网页也就是直接存储在服务器磁盘上的 HTML 文档，它和其他静态内容一样，都不需要动态脚本解释器的参与，所以节省了一定的 CPU 和内存开销，同时，它们需要全身心地进行 I/O 操作，所以是 I/O 密集型的应用。

除了静态网页之外，静态内容还包括图片、样式表等，与动态内容不同的是，静态内容的吞吐率很大程度上取决于服务器的并发处理能力，前面的章节对此有过非常详细的介绍，我们这里简单地回顾一下，看看对于静态内容都可以采取哪些措施。

支持 epoll

它可以使得 Web 服务器在大量并发用户数的情况下保持较稳定的吞吐率。

非阻塞 I/O

避免不必要的 I/O 等待。

异步 I/O

如果可能的话，使用真正意义上的异步 I/O。

使用 sendfile()系统调用

避免文件系统磁盘缓冲区到用户地址空间的数据复制。

单进程

避免多进程切换的不必要开销。对于磁盘 I/O 密集型的静态内容处理，多进程并不能带来多大的意义。

使用高速磁盘

磁盘转速是磁盘数据吞吐率的一个重要因素，使用 15krpm 转速的磁盘将会获得比 7200rpm 转速的磁盘更高的性能表现。

使用 RAID 分区

使用 RAID 分区存储使得磁盘可以实现并行读写，大大提高了磁盘的整体吞吐率。

购买足够的带宽

由于处理静态内容时 Web 服务器的主要时间都用来输出内容，所以相对于动态内容来说，需要消耗更多的带宽。

以上都是针对静态内容的特点而采取的措施，它们只是一小部分，除此之外，不同的静态内容仍然拥有各自的特点，我们来看看静态网页，有一点必须值得我们注意，那就是充分利用浏览器缓存，没错，根据站点内容及时性的要求，尽可能地让这些内容在用户的浏览器中停留更长的时间，在前面的章节中已经介绍过具体做法，还记得吗？

其他的静态内容具备哪些特点呢？

图片

站点中的图片是我们非常熟悉的 Web 组件，它的尺寸从几十字节到几百 KB 各不相同，但是有一个特点，那就是一个网页中往往会包含很多张图片，有时候甚至超过 500 张，这一点都不夸张，当我使用刚刚推出不久的 WebQQ 时，我所有好友的头像都被下载到浏览器端。

这时候，HTTP 持久连接（HTTP Keep-alive）便大有用武之地，试想一下，500 张图片的 HTTP 请求，如果每次都需要从建立 TCP 连接到关闭 TCP 连接，那是多么的浪费，通过 HTTP 持久连接便可以大大提高图片服务器的吞吐率。

一般而言，浏览器会非常积极地在对静态内容的请求中附加持久连接的声明，所以我们只需要设置 Web 服务器端支持持久连接即可，以下是请求一个好友头像时获得的 HTTP 响应头信息，它包含了对持久连接的支持。

```
HTTP/1.1 200 OK
Date: Wed, 15 Apr 2009 14:50:42 GMT
Last-Modified: Wed, 15 Apr 2009 14:50:42 GMT
Expires: Thu, 16 Apr 2009 14:50:42 GMT
Cache-Control: max-age=86400
Content-Length: 4854
Content-Type: image/bmp
Connection: keep-alive
```

除此之外，我们注意到 Expires 过期时间的设置，你一定还记得它的作用，这里设置了 1 天的有效期，也就是在未来 24 小时之内，我的好友头像将不会再次请求服务器。这里的过期时间主要取决于图片更新频率以及你对图片及时性的要求，对于你的站点，或许可以设置更长的过期时间。

样式表

相比之下，CSS 样式表的更新并不是那么频繁，我们看到 Google 将日历服务中 CSS 样式表的有效期设置为 1 年，真是相当的漫长。

```
HTTP/1.1 200 OK
Last-Modified: Tue, 24 Mar 2009 18:34:12 GMT
Cache-control: public
Expires: Thu, 15 Apr 2010 14:58:42 GMT
Content-Type: text/css
Content-Encoding: gzip
Date: Wed, 15 Apr 2009 14:58:42 GMT
X-Content-Type-Options: nosniff
```

```
Content-Length: 4260
Server: GFE/2.0
```

不过，即便是在缓存有效期内更新了样式表，并且希望浏览器及时下载新的样式表，也是有办法的，举个例子，比如我们的站点有一个样式表文件，它的地址如下：

```
http://img.highperfweb.com/styles/layout.css
```

我们在网页中引用这个样式表：

```
<link href="http://img.highperfweb.com/styles/layout.css" rel="style
sheet" type="text/css" />
```

我们像刚才那样，将 Expires 时间设置到一年以后，可是没过几天，我们需要更新这个样式表，很简单，我们只需要在引用样式表时，让样式表的地址发生变化即可，比如增加一些个性化的参数，如下所示：

```
<link href="http://img.highperfweb.com/styles/layout.css?v=1.2" rel="
stylesheet" type="text/css" />
```

这样一来，浏览器就会认为这是一个本地缓存区没有的新 Web 组件，从而发出 HTTP 请求来下载这个样式表。

但是，如果连网页本身也在浏览器缓存有效期中的话，那么新的样式表引用代码也毫无意义，其实，不用担心新的样式表会在一年后才被用户更新，因为用户会单击浏览器的刷新按钮，你可以在站点中引导用户这样做，以获得更新后的内容。

脚本

JavaScript 脚本或者 VBScript 脚本一般在几 KB 到几十 KB，对于一些 RIA（Rich Internet Application）来说，脚本可能会达到上百 KB，比如 Google 在线文档的一个 JavaScript 脚本有大约 104KB，它采用的浏览器缓存有效期为一个月，我们能够看到以下 HTTP 响应头信息：

```
HTTP/1.1 200 OK
Expires: Fri, 15 May 2009 06:29:05 GMT
Cache-Control: public
Last-Modified: Tue, 14 Apr 2009 23:17:14 GMT
Content-Type: text/javascript
Content-Encoding: gzip
Transfer-Encoding: chunked
Date: Wed, 15 Apr 2009 06:29:05 GMT
X-Content-Type-Options: nosniff
Server: GFE/2.0
```

的确，如果你的站点中 JavaScript 脚本更新并不那么频繁，你不妨将它长期寄存在用户浏

览器中，事实上很多时候，直到下一次构建版本部署之前，我们一般不会更新 JavaScript 脚本，但不排除你的团队采用频繁的持续构建和部署，这也不用担心，根据你的情况选择一个合适的过期时间，无论如何，它都是有意义的，因为对于样式表和脚本这样的 Web 组件，站点中几乎每个页面都会引用它们，所以一旦将它们缓存在浏览器本地，整个站点将会大大受益。

视频

我想你一定有过在线观看视频的经历，也体会过等待视频下载的漫长时间，几年前我们会认为原因在于我们家庭宽带的带宽不够，可是现在，问题多数已经不在我们了，而在于视频服务器的出口带宽，大家正拥挤在那里。

我们来看看视频服务器能消耗多大的出口带宽，下面我们对一个 40MB 左右的视频文件进行压力测试，模拟 1000 个并发用户数同时下载这个视频，注意，这次我们仍然位于服务器本地进行测试，所以不存在带宽限制，测试结果如下所示：

```
Server Software:        lighttpd/1.4.20
Server Hostname:        localhost
Server Port:            8001
Document Path:          /test.flv
Document Length:        39921980 bytes
Concurrency Level:      1000
Time taken for tests:   68.981 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      39922234000 bytes
HTML transferred:       39921980000 bytes
Requests per second:    14.50 [#/sec] (mean)
Time per request:       68981.050 [ms] (mean)
Time per request:       68.981 [ms] (mean, across all concurrent requests)
Transfer rate:          565177.78 [Kbytes/sec] received
```

不同的是，对于视频下载的压力测试，我们并不关心吞吐率，因为视频文件的尺寸大小各不相同，吞吐率没有任何意义，而视频下载有其自身的特点，那就是下载速度只要超过视频码率（也称为位速，即媒体文件播放 1 秒所需要的尺寸大小）即可，这样视频便可以流畅地播放。

针对以上的测试，我们来计算下载速度，视频大小为 40MB，用户平均请求等待时间（Time per request）为 69 秒，很容易算出，平均下载速度为 580KB/s，这相当于可以提供 4.6Mbps 的码率，这意味着什么呢？我们知道一个较好音质的 MP3 的码率为 128Kbps，而一个普通 FLV 视频的码率为几十 Kbps 到几百 Kbps，毫不客气地说，4.6Mbps 的码流如果提供视频服务的话，基本上接近 DVD 效果，当然前提是用户的家庭带宽足够大。

但是，刚刚的测试结果其实只是一个峰值，不要忘了我们的所有请求都指向了同一个文件，由于磁盘高速缓存的作用，减少了大量磁盘寻址和数据传输的时间，使得读取速度得到了最大程度地发挥，而在实际情况中，用户的请求通常指向不同的文件，虽然在一些特定的时间段会有大量请求指向个别热点文件，但总体来说，磁盘读操作的位置过于随机，这导致磁盘的寻址时间必不可少，读取速度大量下降，而要准确地预测它并不容易，影响它的因素非常多，通过观察运行中的一台下载服务器，我们看到在带宽充足的情况下，磁盘读取速度最高保持在 70Mbps 到 80Mbps，这相当于占用 600Mbps 左右的带宽，假如我们希望提供 100Kbps 码率的视频下载，那么理论上可以支持 6000 人同时观看，一旦用户超过这个数目，你就得考虑扩展视频服务的规模，后面我们会在涉及负载均衡的章节中介绍。

另一方面，要支持 6000 人同时观看，这意味着 Web 服务器至少要能承受 6000 个并发连接同时下载视频，为 Web 服务器使用多进程非常重要，对于刚才提到的那台下载服务器，我们为 Nginx 配置了 128 个进程，这使得它可以同时处理更多的慢速连接，这在前面关于服务器并发处理能力的章节中曾经详细介绍过。

除此之外，你还可以充分使用诸如 sendfile 这样的 I/O 优化措施，当然，它几乎已经成为 Linux 下所有主流 Web 服务器进行大文件传输的默认开启选项。

这里介绍的所有内容同样适用于其他大文件的下载服务，唯一的区别是，其他类型的文件下载不存在实时观看的需要，所以对于下载速度没有严格的要求，你可以根据用户的需要来进行规划。

第 10 章

分布式缓存

说到缓存，你已经非常熟悉了，我们前面曾经探讨了有关动态内容的各种缓存，但基本上都是基于页面缓存，或者整体缓存，比如缓存整个动态图片。无论如何，它们的目的都在于避免重复的慢速计算，比如数据库访问。但是在有些时候，使用页面缓存显得尤为笨重，这可能来自于以下几个原因：

- 一个网页中不同区域的内容，自身更新频率和呈现及时度要求各不相同，如果为了迁就频繁更新的区域，而使整个页面频繁重建缓存，则影响整体吞吐率。
- 即便是采用局部动态缓存，如果局部区域过多，则会使得页面结构过于复杂，而且整合各个局部页面也存在不小的开销。
- 有些计算是无法作为页面来缓存的，比如有些动态内容中需要获取用户的登录状态，并根据不同用户呈现不同的内容。
- 这些页面缓存都只是提高了读数据的速度，并没有提高写数据的速度。

那么，有什么更好的方法呢？

10.1 数据库的前端缓存区

还记得我们曾经介绍过的文件系统内核缓冲区（Buffer Area）吗？它位于物理内存的内核地址空间，除了使用 O_DIRECT 标记打开的文件之外，所有对磁盘文件的读写操作都要经过它，所以你也可以把它看成是磁盘的前端设备。

这块内核缓冲区也称为页高速缓存（Page Cache），实际上它包括以下两部分：

- 读缓存区
- 写缓存区

读缓存区中保存着最近系统从磁盘上读取的数据，一旦下次需要读取这些数据的时候，内核将直接从这里获得，而不需要访问磁盘。

写缓存区的目的主要在于减少磁盘的物理写操作，通常情况下向磁盘中写入数据并不着急，进程不需要因为写操作而等待，内核缓冲区可以将多次写操作的指令累积起来，通过一次物理磁头的移动来完成。当然，写缓存区导致数据真正写入磁盘会产生几秒的延迟，

在实际写入磁盘之前，这些数据被称为脏页（Dirty Page）。

其实，从工作职能上看，将写缓存区称为缓冲区更加形象，缓冲区的例子在生活中处处可见，比如城市道路的十字路口，它就像一个写缓冲区，红灯亮起的时候，车辆都停在缓冲区，当变成绿灯后，车辆开始依次前进，这就像内核缓冲区中的数据积累到一定程度时被写入磁盘。

一个有趣的现象是，有些城市会在十字路口增加左转弯等待区，多么美妙的想法啊，它使得缓冲区与目的地的距离更加接近，绿灯时将会有更多的车辆通行，提高了道路的吞吐率。

同样的，类似于页高速缓存，我们也可以在数据库和动态内容之间建立一层缓存区，它可以部署在独立的服务器上，用于加速数据库的读写操作，这个缓存区实际上是由动态内容来控制的。

10.2 使用 memcached

幸运的是，开源社区已经有非常成熟的分布式缓存系统，现如今，几乎没有人不知道 memcached，我们也曾经在前面使用过 memcached 来存储动态内容的页面缓存。可是，你真的能让它工作得愉快吗？

key-value

首先，我们要知道，为了实现高速缓存，我们不会将缓存内容放置在磁盘上，否则将毫无意义。基于这个原则，memcached 使用物理内存来作为缓存区，当我们启动 memcached 的时候，需要指定分配给缓存区的内存大小，比如我们分配了 4GB 的内存来作为缓存区：

```
s-colin:~ # memcached -d -m 4086 -l 10.0.1.12 -p 11711
```

如果要说 memcached 最需要的是什么，毫无疑问，那就是内存。我使用的一台 memcached 服务器已经消耗掉了 2.8GB 的内存空间，而我一共给它分配了 4GB。

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 3950 root      15   0 2913m 2.8g  384 S    1 35.9   1325:49 memcached
```

memcached 使用 key-value 的方式来存储数据，这是一种单索引的结构化数据组织形式，我们将每个 key 以及对应的 value 合起来称为数据项，所有数据项之间彼此独立，每个数据项都以 key 作为唯一索引，你可以通过 key 来读取或者更新这个数据项。

对于 key-value 这样的存储形式，你无法习惯性地像操作关系数据库那样来控制它，最大的不同在于，所有基于 SQL 语句的条件查询方式在 key-value 形式的存储数据上都无法施展。所以，如何用 key-value 方式来存储数据，一部分取决于你将如何查询这些数据。

在稍具规模的应用中，缓存的数据项可能会非常多，足够达到天文数字，为了在内存中为如此之多的数据项提供高速的查找，memcached 使用了高效的基于 key 的 hash 算法来设计存储数据结构，并且使用了精心设计的内存分配器，它们使得数据项查询的时间复杂度达到 O(1)，这意味着不论你存储多少数据项，查询任何数据项所花费的时间都不变，几乎没有什么理由可以让你不用它。

数据项过期时间

由于缓存区空间是有限的，一旦缓存区没有足够的空间存储新的数据项时，memcached 便会想办法淘汰一些数据项来腾出空间，淘汰机制基于 LRU（Least Recently Used）算法，将最近不常访问的数据项淘汰掉。

当然，我们更愿意为数据项设置过期时间，这样它可以很有面子地离开缓存区。至于过期时间的取值，前面我们已经讨论过很多次，这需要你根据自己的站点来把握平衡，它们的道理都是相似的，你可以回顾一下之前介绍过期时间的内容。

如果你在使用 PHP 来编写动态内容，通过 memcached 的 PECL 扩展，你可以很容易地设置数据项过期时间，比如：

```
<?php
$memcache_obj = memcache_connect('10.0.1.12', 11211);
$memcache_obj->add('item_key', 'item_value', false, 30);
?>
```

这里我们将 item_key 这个数据项的过期时间设置为 30 秒。

网络并发模型

作为分布式缓存系统，memcached 可以运行在独立的服务器上，动态内容通过 TCP Socket 来访问它，这样一来，memcached 本身的网络并发处理能力便显得尤为重要。memcached 使用 libevent 函数库来实现网络并发模型，其中包括我们前面详细介绍过的 epoll，所以你可以在较大并发用户数的环境下仍然放心使用 memcached。

这里，我们不妨做一个简单的性能测试，来看看 memcached 中 set 操作的性能，我们用 PHP 编写了以下的代码：

```
<?php
$key = md5(uniqid());
$value = md5(uniqid());
$memcache = memcache_connect('10.0.1.12', 11711);
$memcache->set($key, $value, false, 0);
$memcache->close();
?>
```

很简单，它首先随机生成两个 32 字节的字符串，分别赋值给 key 和 value，比如：

```
key -> 87e4f726c1837c98b4719488d9530532
value -> 64f0f23d115dca3b019bea8340cb552e
```

接下来连接到 memcached 服务器，然后通过 set 操作来写入数据，最后关闭连接。显然，我们将这一系列的操作打包为一个动态程序，下面就对这个动态程序进行压力测试，结果如下所示：

```
Server Software:        lighttpd/1.4.20
Server Hostname:        www.liveinmap.com
Server Port:            8001
Document Path:          /book_memcache_perf3.php
Document Length:        844 bytes
Concurrency Level:      500
Time taken for tests:   2.138 seconds
Complete requests:      5000
Failed requests:        0
Write errors:           0
Total transferred:      5028338 bytes
HTML transferred:       4220000 bytes
Requests per second:    2338.48 [#/sec] (mean)
Time per request:       213.814 [ms] (mean)
Time per request:       0.428 [ms] (mean, across all concurrent requests)
Transfer rate:          2296.61 [Kbytes/sec] received
```

从结果中我们知道 memcached 服务器平均每秒处理了大约 2338 个 set 请求，当然，处理过程中不仅仅包括 set 操作，还包括了建立连接和释放连接，它们的开销不可忽视。

接下来，我们希望尽可能地了解 set 操作本身的性能，办法是有的，我们在一个连接中重复进行多次 set 操作就可以了，我们修改一下刚才的 PHP 代码，如下所示：

```
<?php
$key = md5(uniqid());
$value = md5(uniqid());
$memcache = memcache_connect('10.0.1.12', 11711);
for ($i = 0; $i < 10000; ++$i)
{
    $memcache->set($key . $i, $value . $i, false, 0);
}
$memcache->close();
?>
```

与修改之前相比，我们看到 set 操作被一个循环体重复执行了 10000 遍，并且为了让每次 set 操作可以更新不同的节点，更加接近实际运行情况，我们在循环体中对 key 的末尾追加了递增的数值。

我们再来进行压力测试，但这次只模拟 10 个并发用户数，同时总请求数为 50，也就是每个用户发送 5 个请求，为什么这么做呢？别忘了我已经让脚本中的 set 操作重复了 10000 遍，我可等不了那么长的时间。测试结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        www.liveinmap.com
Server Port:            80
Document Path:          /book_memcache_perf3.php
Document Length:        2 bytes
Concurrency Level:      10
Time taken for tests:   12.248 seconds
Complete requests:      50
Failed requests:        0
Write errors:           0
Total transferred:      10100 bytes
HTML transferred:       100 bytes
Requests per second:    4.08 [#/sec] (mean)
Time per request:       2449.526 [ms] (mean)
Time per request:       244.953 [ms] (mean, across all concurrent requests)
Transfer rate:          0.81 [Kbytes/sec] received
```

这次我们不看结果中的吞吐率，而是记录下总时间，为 12.248 秒。在这段时间内，memcached 一共处理了 50 个请求，而每个请求包括 10000 次 set 操作，那么我们可以算出每秒执行的 set 次数为：

```
10000 × 50 / 12.248 ≈ 40823
```

当然，这是极端前提下的结果，实际情况中根据 TCP 连接复用程度的不同，memcached 的吞吐率会存在上下浮动，这是正常的，你需要根据站点的实际环境来评估它的处理能力。

对象序列化

我们可以在 memcached 的数据项中存储什么格式的内容呢？这要考虑到网络传输，问题也就转换成了“我们可以在网络中传输什么内容”。自然是二进制数据。那么，对于数组或者对象这样的抽象数据类型，是否可以存入 memcached 中呢？

基于序列化（Serialize）机制，我们可以将更高层的抽象数据类型转化为二进制字符串，以便通过网络进入缓存服务器，同时，在读取这些数据的时候，二进制字符串又可以转换回原有的数据类型。

但是，有一点需要清楚的是，当我们试图将一个类的对象（或类的实例）进行序列化时，对象的成员函数是不被序列化的，而被序列化存储的只是对象的数据成员。当需要从持久化数据中恢复对象（反序列化）时，我们首先会实例化另一个新的对象，然后将之前持久化的数据成员依次赋值给新对象的相应数据成员。听起来比较复杂，不过幸运的是，在具有动态特性的脚本语言中，这些具体的过程往往不需要你去实现，你只需要或多或少了解序列化的本质即可。

顺便说一下，我们熟悉的 JSON 格式，便可以很好地应用在序列化中，任何数组和对象都可以很容易地与 JSON 格式的字符串互相转换，而且转换所需的计算量并不大。

对于对象序列化，常见的服务器端动态脚本语言都有相应的扩展支持，比如通过 PHP 的 memcached 扩展，你可以随时将对象写入缓存服务器，并在随后取出。下面是一个简单的例子：

```
<?php
class Person
{
        var $name;
        function setName($name)
        {
                $this->name = $name;
        }
}
$person = new Person();
$person->setName('colin');
$key = 'person.colin';
$memcache = memcache_connect('10.0.1.12', 11711);
$memcache->add($key, $person, false, 0);
$obj = $memcache->get($key);
echo $obj->name;
?>
```

好，我们来运行这个 PHP 脚本，你可以通过浏览器请求它，也可以直接通过 PHP 的命令行方式来执行它，无论如何，它的结果都是一样的，如下所示：

```
colin
```

10.3 读操作缓存

前面简单介绍了 memcached 的一些基本特性，因为它实在是家喻户晓，以至于我觉得不需要太多详细地介绍它本身，否则你会觉得我在浪费你的时间，好，我们还是来看看如何用它来帮助站点提高吞吐率，这也许是你最关心的。

还记得磁盘缓存区中的两部分吗？没错，读缓存区和写缓存区。那么我们也按照这个思路，先将 memcached 作为数据库的读缓存区，来看一个例子。

重复的身份验证

大多数站点都有自己的用户系统，这给我们带来了不少的麻烦，有些事情不可不做，那就是在每个用户请求页面的时候，都需要检查用户的登录状态。

很早以前，我喜欢将登录用户的 ID 直接写入浏览器 cookies，并以此为荣，可是在随后的几年里，破坏者对 cookies 更是情有独钟，他们可以很容易地利用 cookies 的无知，篡改本地 cookies 来冒充其他用户，这让我感到时代在飞速前进，我们必须改变。

新的方式问世了，用户在登录站点的时候会获得一个 ticket 字符串，并将它写入浏览器 cookies，随后每次请求新的页面时，都需要上报这个 ticket，接受重复的身份检查。非常好，它工作得一切正常，破坏者由于不知道我们发给其他登录用户的 ticket，所以无法冒充。补充一点，事实上，永远不要低估破坏者的本领，它们可以通过其他手段获得他们想要的一切东西。

现在，我们将身份检查部分的代码抽取出来，它的工作很简单，读取 cookies 中的 ticket 字符串，然后通过条件 SQL 语句查询数据库，寻找拥有这个 ticket 的用户。我们来看这部分代码，它用 PHP 编写而成。

```
    <?php
    $user_ticket = '010f06c4c7e74f82fe7fb2aea97c50b2';
    $sql = "select * from user_info where user_ticket='" . $user_ticket .
"'";
    $conn = mysql_connect('localhost', 'root', '', 'db_user');
    $res = mysql_query($sql, $conn);
    $row = mysql_fetch_array($res, MYSQL_ASSOC);
    var_dump($row);
    ?>
```

以上的代码不难理解，我们这里省略了从 cookies 中获取 ticket 的过程，而直接将 ticket 赋值给$user_ticket 变量，接下来是数据库查询，然后将获得的用户信息进行打印。这里涉及一个数据表 user_info，它有大约 20 个字段，1 万多行的记录。

我们来对它进行压力测试，结果如下所示：

```
Server Software:        lighttpd/1.4.20
Server Hostname:        www.liveinmap.com
Server Port:            8001
Document Path:          /readcache_mysql.php
Document Length:        850 bytes
Concurrency Level:      500
Time taken for tests:   40.263 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      10042546 bytes
HTML transferred:       8500000 bytes
Requests per second:    248.37 [#/sec] (mean)
Time per request:       2013.155 [ms] (mean)
Time per request:       4.026 [ms] (mean, across all concurrent requests)
Transfer rate:          243.58 [Kbytes/sec] received
```

我不认为这个结果很出色，你也许跟我有同样的想法。

数据库索引

不过，我并没有对数据库失去信心，我们来看看以上程序中执行的一段 SQL 语句：

```
    select * from user_info where user_ticket='010f06c4c7e74f82fe7fb2aea97
c50b2'
```

它包含一个 where 条件，我想我们可能没有使用索引，通过 MySQL 的查询分析工具来分析这条 SQL 语句，结果如下所示：

```
    mysql> explain select * from user_info where user_ticket='010f06c4
c7e74f82fe7fb2aea97c50b2';
    +----+-------------+----------+------+---------------+------+-----
----+------+------+-------------+
    | id | select_type | table    | type | possible_keys | key  | key_len
| ref  | rows | Extra       |
    +----+-------------+----------+------+---------------+------+-----
----+------+------+-------------+
    |  1 | SIMPLE      | user_info | ALL  | NULL          | NULL | NULL    |
NULL | 10807 | Using where |
    +----+-------------+----------+------+---------------+------+-----
----+------+------+-------------+
    1 row in set (0.02 sec)
```

的确，请看 key 字段，结果为 NULL，这导致它要遍历所有的 10807 行记录来寻找目标，毫无疑问时间都花在这里了。

我们这里要做的就是为 user_ticket 字段加上索引，然后再来看看压力测试结果：

```
    Server Software:        lighttpd/1.4.20
    Server Hostname:        www.liveinmap.com
    Server Port:            8001
    Document Path:          /book_readcache_mysql.php
    Document Length:        850 bytes
    Concurrency Level:      500
    Time taken for tests:   5.970 seconds
    Complete requests:      10000
    Failed requests:        0
    Write errors:           0
    Total transferred:      10140721 bytes
    HTML transferred:       8500000 bytes
    Requests per second:    1675.11 [#/sec] (mean)
    Time per request:       298.489 [ms] (mean)
    Time per request:       0.597 [ms] (mean, across all concurrent requests)
    Transfer rate:          1658.86 [Kbytes/sec] received
```

虽然难以置信，但是合乎情理，我们再来看看 SQL 分析：

```
    mysql> explain select * from user_info where user_ticket='010f06c4c7
e74f82fe7fb2aea97c50b2';
    +----+-------------+----------+------+---------------+-----------
-+---------+-------+------+-------------+
    | id | select_type | table    | type | possible_keys | key         | key_
len | ref   | rows | Extra       |
    +----+-------------+----------+------+---------------+-----------
-+---------+-------+------+-------------+
```

```
    |  1 | SIMPLE      | user_info | ref  | user_ticket   | user_ticket |
35      | const |    1 | Using where |
    +----+-------------+----------+------+--------------+-----------
-+---------+-------+------+-------------+
    1 row in set (0.00 sec)
```

这时候 key 的结果为 user_ticket，这意味着此次查询利用了该索引，直接找到目标，的确，rows 为 1。

关于数据库索引的探讨，我们暂且放下，这里之所以提出索引，目的在于希望数据库和 memcached 保持相似的压力测试条件，因为 memcached 也使用了基于 hash 算法的 key 来直接定位目标。另外，这里拿出数据库索引小试牛刀，也算是给下一章做个预热，下一章将详细介绍包括索引在内的一系列数据库优化方法。

缓存用户登录状态

即便使用了索引，查询本身还是存在开销，这很大程度上在于数据库的 I/O 操作，这时候，轮到 memcached 出场了。我们接下来希望将用户状态缓存在 memcached 中，没错，对象序列化派上用场了，它帮你简化了工作量，我们来看看新的代码：

```
<?php
$user_ticket = '010f06c4c7e74f82fe7fb2aea97c50b2';
$memcache = memcache_connect('10.0.1.12', 11711);
$user = $memcache->get($user_ticket);
if ($user !== false)
{
        var_dump($user);
        exit(1);
}
$sql = "select * from user_info where user_ticket='" . $user_ticket . "'";
$conn = mysql_connect('localhost', 'root', '', 'db_user');
$res = mysql_query($sql, $conn);
$user = mysql_fetch_array($res, MYSQL_ASSOC);
$memcache->add($user_ticket, $user, false, 3600);
var_dump($user);
?>
```

这样一来，用户在 1 个小时的缓存有效期内，便不需要访问数据库。有一点需要注意，当用户选择注销时，你必须主动清空 memcached 中该用户的登录状态缓存，否则会让用户感到匪夷所思。

还是老办法，压力测试见分晓，结果如下所示：

```
Server Software:        lighttpd/1.4.20
Server Hostname:        www.liveinmap.com
Server Port:            8001
Document Path:          /book_readcache_memcache.php
Document Length:        844 bytes
```

```
Concurrency Level:      500
Time taken for tests:   3.767 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      10036411 bytes
HTML transferred:       8440000 bytes
Requests per second:    2654.64 [#/sec] (mean)
Time per request:       188.349 [ms] (mean)
Time per request:       0.377 [ms] (mean, across all concurrent requests)
Transfer rate:          2601.86 [Kbytes/sec] received
```

相比于前面的访问数据库，这次的吞吐率提高了大约 58%，因为我们使用了数据库的“前置读缓存”。事实上，数据库本身也有查询缓存，后面我们会介绍，但是那属于数据库的控制范围，而我们这里的分布式缓存则由动态内容来控制，它更加灵活。是的，它可以缓存一切。

值得说明的是，以上的压力测试结果都不具备绝对意义，因为有很多不确定的环境因素，比如数据库环境设置、数据表记录数、opcode 缓存、机器硬件配置、当前系统负载等。另一方面，它们的相对意义也有一定的约束条件，也许在某种环境下，你可以让访问数据库获得比访问 memcached 缓存更好的表现，比如将 memcached 服务器部署在另一个数据中心，当然，你不会那么做。总之，你只需要从本质上理解这些对比背后的原因即可，它们往往都是一些简单的道理，类似于索引优于没有索引、内存优于磁盘、近距离优于远距离。

10.4 写操作缓存

对于一个数据库写操作频繁的站点来说，通过引入写缓存来减少写数据库的次数显得至关重要。我们知道，通常的数据写操作包括插入、更新、删除，这些操作又同时可能伴随着条件查找和索引的更新，所以它们的开销往往会令人望而生畏。

直接更新

下面我们来看一个有趣的例子，就拿站点访问量统计功能来说，我们需要记录每个 URL 的累计访问量，所以每次页面刷新都会伴随着一次访问量的增加，我们将访问量数据保存在数据库中，这毫无疑问，因为我们要长久地保存它。

为此，我们编写了一段有代表性的代码，它可以让某个页面的访问量加 1，代码如下所示：

```
    <?php
    $page = 'article_090222.htm';
    $sql = "update page_view set view_count=view_count+1 where page='" .
$page . "'";
    $conn = mysql_connect('localhost', 'root', '', db_page);
    mysql_select_db('db_map_main', $conn);
```

```
mysql_query($sql, $conn);
?>
```

这段代码很简单，虽然我不知道 article_090222.htm 这个页面目前的访问量是多少，但是我知道它的访问量增加了 1，而且结果将保存在数据库中。我们可以称它为“直接更新”，这来源于之前介绍的文件访问中的直接 I/O 标记（O_Direct），它可以跳过内核写缓存，将数据毫无延迟地直接写入磁盘。

我们对上面的动态程序进行压力测试，结果如下所示：

```
Server Software:        lighttpd/1.4.20
Server Hostname:        www.liveinmap.com
Server Port:            8001
Document Path:          /book_writecache_mysql.php
Document Length:        0 bytes
Concurrency Level:      500
Time taken for tests:   5.239 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      1690000 bytes
HTML transferred:       0 bytes
Requests per second:    1908.89 [#/sec] (mean)
Time per request:       261.933 [ms] (mean)
Time per request:       0.524 [ms] (mean, across all concurrent requests)
Transfer rate:          315.04 [Kbytes/sec] received
```

从测试结果可以看出，我们对这个动态程序一共请求了 10000 次，这也意味着 article_090222.htm 这个页面的访问量被增加了 10000。下面我们引入 memcached 作为写缓存，它也许会使得数据更新出现延迟，但是我们可以接受，因为我们并不需要访问量数据实时更新。

线程安全和锁竞争

在此之前，我们先来看一种传统的分布式加运算，以下的代码只是例子中的一个片段，它先从缓存服务器上取回一个数值，然后在本地加 1，接下来写回缓存服务器。

```
<?php
$count = $memcache->get($key);
$count++;
$memcache->set($key, $count, false, 0);
?>
```

看起来没有任何问题，但是，别忘了可能会有多个用户同时触发这样的计算，你一定能想象的到会有什么糟糕的后果，最后的累计访问量总是小于实际访问量。

事实上，这并不涉及 memcached 本身线程安全的问题，而是以上这种加运算的方式不是线程安全的。如果要保证这种加运算可以正常无误地同时进行，那就要考虑一定的事务隔离机制，简单的办法是使用锁竞争，并且将锁保存在 memcached 上，存在竞争关系的动态内容可以争夺这个锁，一旦某个会话抢到锁，那么其他的会话必须等待。

这里要说的不是如何实现这种分布式锁机制，而是并不鼓励这样做，因为锁竞争带来的等待时间是无法容忍的，这将使得引入 memcached 作为写缓存的唯一优势立刻烟消云散。

原子加法

幸运的是，memcached 提供了原子递增操作，事实上，也正是因为它，我们才考虑在访问量递增更新的应用中引入写缓存。

我们再来修改代码，加入 memcached 的支持，如下所示：

```
<?php
$page = 'article_090222.htm';
$memcache = memcache_connect('10.0.1.12', 11711);
$count = $memcache->increment($page, 1);
if ($count === false)
{
        $memcache->add($page, 1, false, 0);
        exit(1);
}
if ($count == 1000)
{
        $memcache->set($page, 0, false, 0);
        $sql = "update page_view set view_count=view_count+" . $count .
" where page='" . $page . "'";
        $conn = mysql_connect('localhost', 'root', '', 'db_page');
        mysql_query($sql, $conn);
}
?>
```

在新的代码中，完全改变了之前的“直接更新”方式，当需要增加一次访问量的时候，它做了以下工作：

1. 为 memcached 缓存中的对应数据项加 1，如果该数据项不存在，则创建该数据项，并且赋值为 1，代表这个页面是第一次被访问；

2. 如果 memcached 缓存中存在对应数据项，并且累加后的数值为 1000，则将这个数据项置 0，同时更新数据库，将数据库中的对应数值加 1000。

也就是说，改造后的程序每经历 1000 次递增后才写一次数据库，究竟效果如何呢？我们再来进行压力测试，结果如下所示：

```
Server Software:        lighttpd/1.4.20
Server Hostname:        www.liveinmap.com
Server Port:            8001
Document Path:          /writecache_memcache.php
Document Length:        0 bytes
Concurrency Level:      500
Time taken for tests:   3.599 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      1690000 bytes
HTML transferred:       0 bytes
Requests per second:    2778.24 [#/sec] (mean)
Time per request:       179.970 [ms] (mean)
Time per request:       0.360 [ms] (mean, across all concurrent requests)
Transfer rate:          458.52 [Kbytes/sec] received
```

吞吐率提高了大约 46%，这同样是一个不小的飞跃。所以，如果你的数据库因为大量的写操作而繁忙不堪，那么仔细考虑一下，哪些写操作可以缓存在 memcached 中呢？

10.5 监控状态

作为一个分布式缓存系统，memcached 可以非常出色地完成你交给它的工作，但这并不代表你可以对它放任不管，相反，我们需要知道它的运行状况。memcached 提供了这样的协议，可以让你获得它的实时状态，我们通过 PHP 扩展可以十分容易地做到。

以下的代码片段用来获取 memcached 的状态：

```
<?php
$memcache = memcache_connect('10.0.1.12', 11711);
$stats = $memcache ->getStats();
var_dump($stats);
?>
```

我们来看一下包含了 memcached 运行状态的数组，如下所示：

```
array(22)
{
  ["pid"]=> string(4) "3950"
  ["uptime"]=> string(8) "14670598"
  ["time"]=> string(10) "1239857310"
  ["version"]=> string(5) "1.2.2"
  ["pointer_size"]=> string(2) "32"
  ["rusage_user"]=> string(12) "14261.343278"
  ["rusage_system"]=> string(12) "65035.856487"
  ["curr_items"]=> string(8) "24144013"
  ["total_items"]=> string(9) "175459454"
  ["bytes"]=> string(10) "2562440230"
  ["curr_connections"]=> string(2) "13"
```

```
    ["total_connections"]=> string(10) "1433357990"
    ["connection_structures"]=> string(3) "164"
    ["cmd_get"]=> string(10) "1414539975"
    ["cmd_set"]=> string(9) "175563032"
    ["get_hits"]=> string(10) "1328926319"
    ["get_misses"]=> string(8) "85613656"
    ["evictions"]=> string(1) "0"
    ["bytes_read"]=> string(13) "15130893335558"
    ["bytes_written"]=> string(13) "4661687357256"
    ["limit_maxbytes"]=> string(10) "4284481536"
    ["threads"]=> string(1) "1"
}
```

真是面面俱到，你可以从这些数据中获得很多的信息，比如 uptime 表示 memcached 持续运行的时间；cmd_get 表示读取数据项的次数；cmd_set 表示更新数据项的次数；get_hits 表示缓存命中的次数；bytes_read 表示读取的总字节数；bytes 表示缓存区已使用空间的大小；limit_maxbytes 表示缓存区空间的总大小。

对于这些丰富的状态信息，我们可以简单地从以下三个方面来看。

空间使用率

持续关注缓存空间的使用率，可以让我们知道何时需要为缓存系统扩容，以避免由于缓存空间已满造成的数据被动淘汰，有些数据项在过期之前被 LRU 算法淘汰可能会造成一定的不良后果。

缓存命中率

这个话题在此处已经一点都不新鲜了，在前面关于反向代理缓存的章节中，我们曾经详细介绍过影响命中率的因素，它们在这里同样适用。

I/O 流量

我们需要关注 memcached 中数据项读写字节数的增长速度，这反映了它的工作量，我们可以从中得知 memcached 是空闲还是繁忙。

同样，我们也希望在监控系统中集成对于 memcached 的监控，如图 10-1 及图 10-2 所示便是在 cacti 监控系统中对 memcached 的某些监控图片，我们会在本书的末尾介绍 cacti 监控系统。

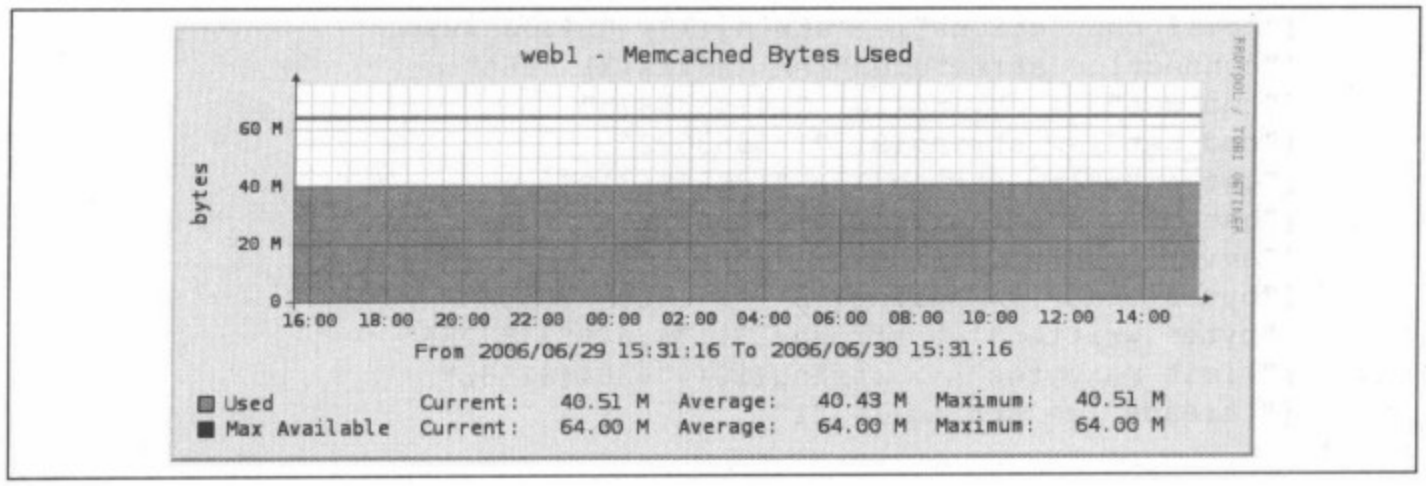


图 10-1　在 cacti 中监控 memcached 的使用率

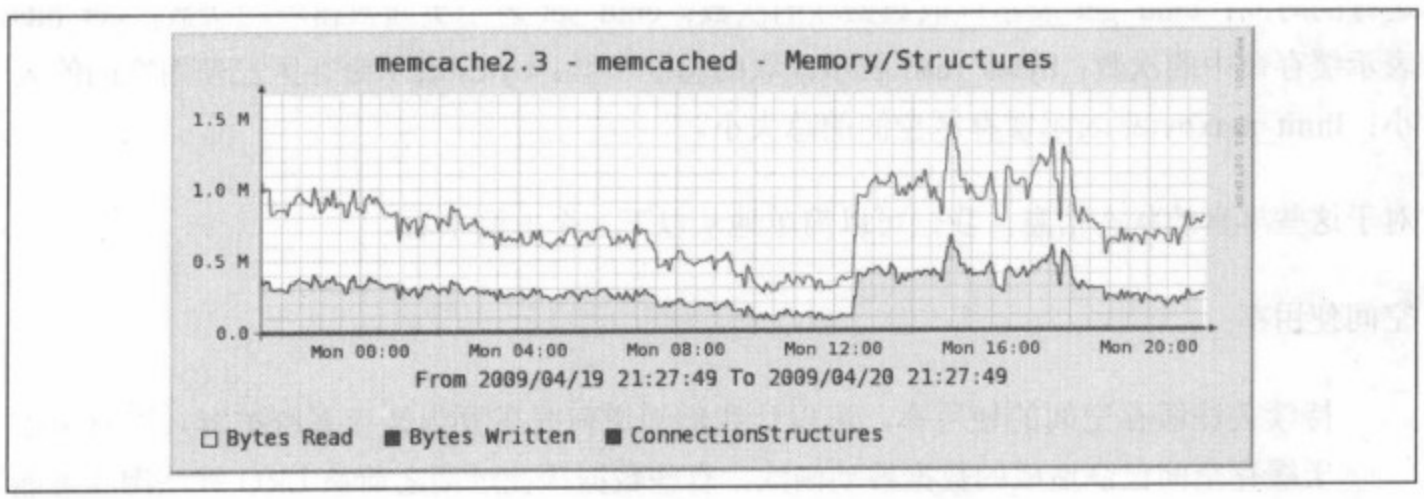


图 10-2　在 cacti 中监控 memcached 的 I/O 流量

10.6　缓存扩展

有很多理由让我们不得不扩展 memcached 的规模，包括并发处理能力和缓存空间容量等，不论是哪个方面达到极限，扩展都在所难免。

对于缓存空间的容量，扩容意味着增加服务器物理内存，这显得不切合实际，而对于并发处理能力，我们知道，memcached 已经在这方面做了很大的努力，这也是它成名的前提。所以，我们只能通过增加新的缓存服务器来达到扩展的目的。

当存在多台缓存服务器后，我们面临的问题是，如何将缓存数据均衡地分布在多台缓存服务器上。现假设我们拥有以下两台缓存服务器，它们的内部 IP 地址如下所示，它们都运行着 memcached。

```
10.0.1.12
10.0.1.13
```

看起来这没什么难的，因为 key-value 类型的数据项本来就相互独立，你可以在连接 memcached 服务器的时候，随便从以上两台服务器中挑选出一台即可。但是，关键在于如

何做到“均衡”呢？这个问题归结于如何对数据项进行分区。

相比于关系型数据库的分区设计，key-value 型数据缓存的分区要容易得多，你可以根据数据项的商业逻辑进行分区设计，拿前面的例子来说，我们可以将用户登录状态的缓存部署在独立的一台缓存服务器上，而将访问量统计的缓存部署在另一台独立的服务器上。这样一来，这两台缓存服务器的用途如下：

```
10.0.1.12 -> 用户登录状态缓存
10.0.1.13 -> 访问量统计缓存
```

但是，它们仍然存在以下两个问题：

1. 两台缓存服务器的工作量真的均衡吗？

2. 如果两台缓存服务器仍然不能满足需要，如何继续扩展呢？

第一个问题的答案并不确定，但我想在大多数情况下都是不均衡的，由于以上两种缓存的职责截然不同，所以它们的开销和访问频率也会存在很大的差异，我们当然希望两台缓存服务器的工作量比较对称，达到物尽其用。

对于第二个问题，假如访问量统计缓存需要进一步扩展，我们准备了新的缓存服务器：

```
10.0.1.14
```

接下来我们将访问量统计缓存再次划分，存储在两台缓存服务器上，如何划分呢？我们可以将被统计的所有子站点划分为两部分，让它们分别存储在不同的缓存服务器上，为此，我们需要为每个缓存服务器维护一个一对多的映射表（One To Many Mapping List），里边记录子站点的域名，比如：

```
    Group 1 -> www.highperfweb.com, app1.highperfweb.com, app2.highperfweb.
com
    Group 2 -> app3.highperfweb.com, app4.highperfweb.com, app5.highperfweb.
com
```

如此一来，我们重新调整缓存划分方式，三台缓存服务器上的缓存内容如下所示：

```
10.0.1.12 -> 用户登录状态缓存
10.0.1.13 -> 访问量统计缓存 Group 1
10.0.1.14 -> 访问量统计缓存 Group 2
```

但是，假如再次需要扩展，我们还要调整映射表吗？问题是也许你也不知道该怎么划分，实际情况中的映射表可能不像例子中的如此简洁，人工维护这个列表无疑是自找麻烦。另一方面，一个老问题再次浮现出来，你能保证它们的工作量均衡吗？

好吧，我们决定打破这种基于数据项商业逻辑的划分思维，来考虑一种基于 key 的划分方式，这有些类似于后面介绍的数据库水平分区（Sharding）。我们需要设计一种不依赖数据项内容的散列算法，将所有数据项的 key 均衡分配在这三台缓存服务器上。

一个简单而有效的方法是“取余”运算，这就像打扑克时的发牌，让所有数据项按照一个顺序在不同的缓存服务器上轮询，这可以达到较好的相对平衡，想想发牌的动机也正是为了让大家彼此公平。这种方法是一种比较常用的基本散列算法，事实上在很多时候都用得到，而且你可以根据实际情况对它进行改造，达到更好的散列效果。下面我们举个例子。

在“取余”之前，我们先要做一些准备工作，目的是让 key 变成整数，而且尽量唯一。比如对于以下这个 key：

```
article_090222.htm
```

我们先对它进行 md5 运算，这里直接使用 PHP 命令行方式：

```
s-colin:~ # php -r "echo md5('article_090222.htm');"
e6e87fc9f9c2914339a9b7cc4db6055c
```

得到的是一个 32 字节的字符串，同时它也是一个十六进制的长整数，为了减少计算开销，我们取这个字符串的前 5 个字节，然后将它转换为十进制数：

```
s-mat:~ # php -r "echo hexdec('e6e87');"
945799
```

然后将结果进行“模 3”运算：

```
s-mat:~ # php -r "echo 945799 % 3;"
1
```

得到的余数便是缓存服务器的编号，我们的三台缓存服务器应该从 0 开始编号，那么 1 代表了第二台服务器。

看起来真复杂，我们不得不需要一个“缓存连接器”了，我们希望将上面这些运算都放在连接器里，而只需要告诉它 key，接下来选择缓存服务器的事情就拜托给它了。看看连接器的一个例子：

```
<?php
function memcache_connector($key)
{
    $hosts = array(
        '10.0.1.12',
        '10.0.1.13',
        '10.0.1.14'
    );
    $host_index = hexdec(substr(md5($key), 0, 5)) % 3;
    $host = $hosts[$host_index];
```

```
        return memcache_connect($host, 11711);
    }
    ?>
```

现在，我们访问缓存的时候只需要这样连接缓存服务器即可：

```
<?php
$memcache = memcache_connector('article_090222.htm');
?>
```

那么，如果还需要继续扩展，一定难不倒你了，将“模 3”的运算变成“模 4”、“模 5”……然后其余的工作就放心地交给连接器去做吧。

这里有一个问题也许你一直在思考，那就是当我们扩展缓存系统后，由于分区算法的改变，会涉及缓存数据需要从一台缓存服务器迁移到另一台缓存服务器的问题，如何迁移呢？事实上，根本不需要考虑分区之间的迁移，因为这是缓存，它应该具备在必要时刻牺牲自己的勇气，当然这是你赋予它的，你必须明白缓存不是持久存储，并且从引入分布式缓存开始就不断地提醒自己。

没错，当调整缓存分区算法后，我们需要时间来等待缓存重建和预热，但这往往并不影响站点的正常运转，前提是你按照前面读缓存和写缓存的理念来进行设计。顺便一提的是，与此相比，数据库规模扩展引发分区（Shard）之间的数据迁移就要复杂得多，后面我们会有专门的章节探讨它。

第 11 章

数据库性能优化

也许你已经发现，在这一章之前，很大一部分内容都在于阻止你过多地访问数据库，或者说帮助你尽量少访问数据库，为此我们使用了各种缓存方法，将动态内容不断地靠近用户，从 Web 服务器到反向代理服务器，甚至到用户本地浏览器中，事实上，这已经进入了用户本地内存，没有什么比这更容易让用户随手可得的了。

但是，我们必须保持警惕，大量的数据库访问依旧在所难免，一部分原因包括：

- 即便是有较长的缓存有效期，缓存命中率很理想，但是缓存的创建和过期后的重建是需要访问数据库的，而且我们曾经在介绍反向代理缓存时描述了一个数学模型，通过它我们知道，当存在大量动态内容的时候，后端数据库仍然会面临不小的工作量，这的确是一个长尾效应。
- 对于数据库的写操作，往往不是很容易引入缓存策略，尽管前面我们在例子中通过 memcache 实现了访问量累加计算的缓冲，但是在实际情况中，很多时候不像累加计算那么简单，而且有时我们还需要保证一定的事务级别。
- 也许你根本没有使用过前面介绍的任何缓存方法。

这使我们不得不关注数据库的查询性能，事实上，影响数据库性能的因素非常多，从头到尾地介绍它们已经超出了本书的范围。但是，80%的性能问题往往都是由 20%的错误导致的，本章我们将把有限的篇幅用来介绍这 20%的内容。

提示：

1897 年，意大利经济学家维尔弗雷多·帕累托根据统计结果归纳出：20%的人口拥有 80%的财富，在经济学上被称为“帕累托收入分配定律”。实际情况中没有如此精确的比例，但反映了一种稳定的不平衡关系。

所以，与其说优化，倒不如说我们应该如何避免犯错，我相信当你掌握了本章的内容后，你至少已经挖掘到了 80%以上的数据库性能。在本章中，我们主要以 MySQL 作为关系型数据库的代表来展开讨论，其中涉及的大多数内容在其他关系型数据库中同样适用，最后，我们还会介绍一些非关系型数据库。

值得一提的是，本章的内容将只针对数据库本身的性能问题，而通过数据库扩展来满足更大规模的性能需要，我们会在后续章节中专门探讨。

11.1 友好的状态报告

你的数据库工作得好吗？它们都在忙什么呢？要回答这两个问题还真不那么容易，但是，如果我们不知道答案，或者不知道如何去寻找答案，那么，我们将永远无法谈及优化，因为我们不知道要优化什么。

在 MySQL 命令行中，你可以通过以下的指令来获取当前数据库的实时状态：

```
mysql> show status;
mysql> show innodb status;
```

也许你早已知道它们，但你一直在抱怨它们的结果不够友好，的确，我们不是机器人，我们需要更加人性化的分析结果，而不是一大堆不知所云的数字。

幸运的是，mysqlreport 为我们带来了更好的体验，它是一个第三方的 MySQL 状态报告工具。事实上，mysqlreport 是站在巨人肩膀上的，它所做的事情是把 MySQL 中 show status 和 show innodb status 的结果进行一系列的后期处理，将我们最关心的内容以可读性更好的方式呈现出来。

事不宜迟，我们现在就来看看 mysqlreport 带给我们的状态报告：

```
s-db:~ # mysqlreport
MySQL 5.0.37-log          uptime 1 15:11:47       Fri Apr 24 15:35:49 2009
__ Key _________________________________________________________________
Buffer used   100.37M of   2.00G  %Used:   4.90
  Current       1.02G            %Usage:  51.07
Write hit      99.26%
Read hit       98.71%
__ Questions ___________________________________________________________
Total          58.09M   411.7/s
  DMS          30.62M   217.0/s  %Total:  52.71
  Com_         21.23M   150.4/s           36.54
  COM_QUIT      6.14M    43.5/s           10.58
  +Unknown    104.71k     0.7/s            0.18
Slow 1 s          722     0.2/s            0.04  %DMS:   0.08  Log:  ON
DMS            30.62M   217.0/s           52.71
  SELECT       26.20M   185.7/s           45.10          85.56
  UPDATE        3.72M    26.3/s            6.40          12.14
  INSERT      649.28k     4.6/s            1.12           2.12
  DELETE       54.13k     0.4/s            0.09           0.18
  REPLACE       1.88k     0.0/s            0.00           0.01
Com_           21.23M   150.4/s           36.54
  change_db    21.18M   150.1/s           36.45
  show_variab   8.94k     0.1/s            0.02
  show_status   8.93k     0.1/s            0.02
__ SELECT and Sort _____________________________________________________
Scan            1.52M    10.8/s %SELECT:   5.80
Range         865.18k     6.1/s            3.30
Full join       3.18k     0.0/s            0.01
Range check         0       0/s            0.00
```

```
Full rng join       0       0/s            0.00
Sort scan       2.34M    16.6/s
Sort range      2.10M    14.9/s
Sort mrg pass     739     0.0/s
__ Table Locks _____________________________________________________________
Waited          9.17k     0.1/s  %Total:   0.03
Immediate      32.03M   227.0/s
__ Tables __________________________________________________________________
Open             512 of  512    %Cache: 100.00
Opened          2.78M    19.7/s
__ Connections _____________________________________________________________
Max used          98 of  100      %Max:  98.00
Total           6.15M    43.6/s
__ Created Temp ____________________________________________________________
Disk table    332.75k     2.4/s
Table           2.25M    16.0/s    Size:  32.0M
File            1.48k     0.0/s
__ Threads _________________________________________________________________
Running              2 of    5
Cached               0 of    0      %Hit:      0
Created         6.15M    43.6/s
Slow                 0       0/s
__ Aborted _________________________________________________________________
Clients         5.70k     0.0/s
Connects           36     0.0/s
__ Bytes ___________________________________________________________________
Sent            2.74G   19.4k/s
Received        3.70G   26.2k/s
__ InnoDB Buffer Pool ______________________________________________________
Usage           1.00G of   1.00G  %Used: 100.00
Read hit       99.84%
Pages
  Free                1            %Total:   0.00
  Data            65.16k                   99.43 %Drty:   4.00
  Misc              374                     0.57
  Latched             0                     0.00
Reads           78.05M   553.2/s
  From file    125.51k     0.9/s            0.16
  Ahead Rnd       1740     0.0/s
  Ahead Sql          7     0.0/s
Writes          16.06M   113.8/s
Flushes          1.30M     9.2/s
Wait Free            0       0/s
__ InnoDB Lock _____________________________________________________________
Waits             1441     0.0/s
Current              0
Time acquiring
  Total           1178 ms
  Average            0 ms
  Max               39 ms
__ InnoDB Data, Pages, Rows ________________________________________________
Data
  Reads        142.84k     1.0/s
  Writes         1.26M     8.9/s
  fsync        191.53k     1.4/s
  Pending
    Reads            0
    Writes           0
```

```
    fsync           0
Pages
  Created       1.05k     0.0/s
  Read        173.91k     1.2/s
  Written       1.30M     9.2/s
Rows
  Deleted           0       0/s
  Inserted    149.59k     1.1/s
  Read          3.72G   26.4k/s
  Updated       2.51M    17.8/s
```

看到报告后，有没有觉得非常亲切呢？如果没有，那一定不是报告本身的问题了，我们需要深入了解这些统计数据背后的真相。

当然，有一点值得注意，从报告顶部的 uptime 中我们可以看到 MySQL 数据库的运行时间，而报告中的各项数据统计，都是通过这段运行时间的累计数据计算而得，所以，我们希望这段时间尽可能地长，至少覆盖站点高负载的时段，这样我们采集的数据才具有较好的代表性。

11.2 正确使用索引

在影响数据库查询性能的诸多因素中，索引绝对是一个重量级的因素，一旦索引使用不当，毫不夸张地说，其他任何优化措施将毫无意义。有人说，如果你懂得如何正确使用索引，你将成为一个数据库性能优化专家，当然，这有些夸张，但是足可见索引的重要性。

提示：

有一则经济学笑话说，如果一只鹦鹉懂得说供求关系，那么它就是一个经济学家了。

到底什么是索引

亲爱的读者，当你开始阅读本书的时候，你一定会看看开头的目录，那里列出了所有章节的页码，你可以根据它快速地找到你想阅读的内容，甚至可以不阅读前面的内容而直接跳到这一章，但这不是我所推荐的。

一本书的目录其实也就是索引，简单说，它的目的在于帮助读者快速找到目标内容，这和数据库中索引的目的完全相同。如果我们用 SQL 语句来描述在书中查找某章的内容，也许是这样的：

```
select `内容` from `书` where `章` = '数据库性能优化'
```

可见，我们为书中的“章”建立了索引。但是，假如一本书没有目录，那么试想你要找到

本书某章的内容将会多么困难，你必须花时间一页一页地去查找，如果不幸的话，你要找的内容可能在书的末尾。

这种情况在数据库中称为全表扫描（Full Table Scan），而通过目录直接找到内容的方式，在数据库中称为索引扫描（Index Scan）。

在大多数情况下，索引扫描当然要比全表扫描获得更好的性能，但这并不是绝对的，如果要查找的记录占据了整个数据表的很大比例，那么使用索引扫描反而性能更差，这不难想象，对于一本书来说，如果你希望阅读 80%以上的内容，那么与其每篇文章都通过目录查找，倒不如抛开目录，来一次顺序的浏览，难道不是吗？不要忘了查找目录也需要时间开销啊。

这样一来，是否使用索引又变得扑朔迷离，我不知道未来的查询是否适合索引扫描，怎么办呢？其实，这个问题自然不用你考虑，这就好像你是一本书的作者，你必须准备好目录，然而，看不看目录那是读者考虑的问题，同样，数据库中的查询优化器会判断一次查询是否有必要使用现有的索引以及使用哪个索引。当然，有些时候优化器也会犯错误，我们需要为它指引道路，比如在组合索引存在的时候，事情就不那么简单了，后面我们会举例说明。

另一方面，数据表中的索引并不像书的目录那么简单，因为目录一般只对应到章节，所以目录项比较少，而数据表中的索引则需要对应到每行记录，这样一来，索引项的数量也将会非常多，我们同样面临着在索引列表中找到某一个索引项的效率问题。然而，索引本身的数据结构（MySQL 使用 BTree、Hash 以及 RTree）决定了它们拥有非常高效的查找算法，我们基本上不用担心这部分的开销。

除了普通索引之外，还有唯一索引、主键、全文索引等，但是不论它们叫什么，只要是索引，就都具备前面讲到的索引的本质目的，而各种索引的不同之处则在于提供更多的约束和满足其他的需要，有时你需要根据站点实际情况来选择和权衡，比如你需要通过唯一索引来约束记录插入时的唯一性，那么索引本身的计算开销必然会增加，因为当你插入记录的时候会涉及额外的唯一性检查，当然，如果你喜欢由应用程序来负责保证唯一性的话，那么使用普通索引即可，所以，这些问题需要你自己来权衡。

这是你自己的事情

也许你正在使用框架进行 Web 开发，它帮你做了很多工作，如果没有过度的话，这非常好。但是，几乎没有框架会帮你建立索引，至少我还没有遇到过，它怎么会知道如何帮你建立索引呢？如果你什么都不告诉它的话。

总之，为数据表建立索引是你自己的事情，永远不要期待有什么工具会自动帮你建立索引，没有工具会知道你未来会频繁地在哪些字段上进行条件查询。

解释查询

即便你深刻认识到索引有多么重要，而且你也知道如何通过具体的操作来建立索引，但更加重要的是，你知道为哪些字段建立索引，这是实干家最关注的一个问题。

事实上，这没有什么窍门和捷径，你需要做的就是仔细分析应用中将会执行的所有 SQL 语句，而且最好是亲自分析，如果试图偷懒，最后总是会无功而返。

一般来说，如果一个字段出现在查询语句中基于行的选择、过滤或排序条件中，那么为该字段建立索引便是有价值的，但这也不是绝对的。我们很难给出一个描述了所有情况的列表供你参考，因为查询的过程非常复杂，它可能包含多列索引、组合索引以及联合查询等情况，所以，关键在于掌握分析方法，这样你便能够应付任何的困境。

还记得我们在前面曾经使用 explain 来分析 SQL 查询语句的索引使用情况吗？没错，那一次 explain 的确帮了我们很大的忙，然而它的强大远不止如此，你可以用它来分析任何的查询语句，但不包括那些导致数据更新的语句（如 update 语句），否则它付不起这个责任。如果你希望分析 update 语句执行时的索引使用情况，可以暂时将它改造为 select 语句，同时保留查询条件即可，它们在查找记录行这部分工作上保持一致。

explain 的使用非常简单，我们举个例子，为此创建了以下数据表：

```
CREATE TABLE `test` (
  `id` int(11) NOT NULL auto_increment,
  `name` varchar(255) NOT NULL,
  PRIMARY KEY  (`id`)
)
```

然后我填充了一些记录，内容是什么并不重要，我们这里只是用 explain 来分析查询语句是否使用了索引。

首先，我们进行一次基于主键的条件查找，SQL 语句如下所示：

```
select * from test where id=1
```

我们知道它一定会使用主键索引的，用 explain 分析如下：

```
mysql> explain select * from test where id=1;
+----+-------------+-------+-------+---------------+---------+-----
----+-------+------+-------+
| id | select_type | table | type  | possible_keys | key     | key_len
| ref   | rows | Extra |
```

```
    +----+-------------+------+------+---------------+---------+-----
----+-------+------+-------+
    |  1 | SIMPLE      | test  | const | PRIMARY       | PRIMARY | 4       |
const |    1 |       |
    +----+-------------+------+------+---------------+---------+-----
----+-------+------+-------+
```

从 explain 的分析结果中，我们可以看到两个重要信息：

- type 为 const，意味着这次查询通过索引直接找到一个匹配行，所以优化器认为它的时间复杂度为常量。
- key 为 PRIMARY，意味着这次查询使用了主键索引。

接下来我们使用 name 字段作为查询条件，SQL 语句如下所示：

```
    select * from test where name='colin'
```

来看看 explain 的分析结果：

```
    mysql> explain select * from test where name='colin';
    +----+-------------+------+------+---------------+------+---------
+------+------+-------------+
    | id | select_type | table | type | possible_keys | key  | key_len | ref  |
rows | Extra       |
    +----+-------------+------+------+---------------+------+---------
+------+------+-------------+
    |  1 | SIMPLE      | test  | ALL  | NULL          | NULL | NULL    | NULL
|    4 | Using where |
    +----+-------------+------+------+---------------+------+---------
+------+------+-------------+
```

这时候 type 的结果为 ALL，它意味着这次查询进行全表扫描，这在我们的预料之中，因为 name 字段没有索引，从 key 为 NULL 也可以看出这一点。那么，我们接下来为 name 字段添加索引：

```
    mysql> alter table test add key name(name);
```

这样一来，情况会有所不同吗？我们再来一次 explain 分析，结果如下所示：

```
    mysql> explain select * from test where name='colin';
    +----+-------------+------+------+---------------+------+---------
+-------+------+-------------+
    | id | select_type | table | type | possible_keys | key  | key_len | ref   |
rows | Extra       |
    +----+-------------+------+------+---------------+------+---------
+-------+------+-------------+
    |  1 | SIMPLE      | test  | ref  | name          | name | 257     |
const |    1 | Using where |
    +----+-------------+------+------+---------------+------+---------
+-------+------+-------------+
```

现在可以看到，key 为 name，意味着这次查询使用了 name 字段的索引。另一方面，对于没有使用主键索引或者唯一索引的条件查询，查询结果可能会有多个匹配行，MySQL 为这种情况定义的 type 为 ref，它在联合查询中也很常见。

但是，并不是所有基于 name 字段的条件查询都会使用索引，比如以下这个 SQL 语句：

```
select * from test where name like '%colin'
```

它使用了%进行模糊查询，所以这不是一个完全匹配的条件查询，这时候我们看看 explain 的结果：

```
mysql> explain select * from test where name like '%colin';
+----+-------------+------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+------+------+---------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | test | ALL  | NULL          | NULL | NULL    | NULL |    4 | Using where |
+----+-------------+------+------+---------------+------+---------+------+------+-------------+
```

显然，这次索引帮不上忙了，因为这种不完全匹配的查找方式太过于个性化，无法利用索引的查找算法，所以仍然需要通过全表扫描来查找目标记录。

以上只是一些简单的例子，我们可以用这种方法分析更多复杂的情况。但是，有时候意识会跳出来作祟，我们可能会过于自信，凭直觉和经验去判断索引的使用情况，这可能会给我们带来潜在的危机，当问题发生后，我们才发现，原来查询中对索引的使用并不是我们想象的那样，这属于过于自信的过失，我们要格外注意。

提示：

根据《刑法》第 15 条规定，过于自信的过失，是指行为人已经预见到自己的行为可能发生危害社会的结果，但是轻信能够避免，以致发生这种结果的主观心理态度。过于自信的过失具有两个特点：1. “已经预见”而“轻信能够避免”；2. 危害结果的发生是违背行为人意愿的。

说到这里，我想也许可以为这本书增加一个关键词索引，并将关键词按照首字拼音的字母顺序排列，方便读者根据关键词来查找内容，这种行为如果用 SQL 语句来描述，那就是：

```
select `内容` from `书` where `关键词` = '索引扫描'
```

当然，这里的关键词索引同样不是唯一索引，因为同一个关键词可能会出现在书中的多个位置。

使用组合索引

再说说书的目录，假如目录中只列出了每一章，而我们要找的是第 6 章中的第 9 节，怎么办？我们只能根据目录先翻到第 6 章所在的确切页码，然后，一页一页地寻找第 9 节，这的确有些不符合阅读习惯，也许会让读者的激情一落千丈。幸运的是，大多数书的目录中都会列出“章”和“节”，这便是“组合索引”的目的。

在实际的数据库访问中，大多数的查询都包含组合条件，比如：

```
... where a = 1 and b = 2
... where a = 1 order by b
... where a = 1 group by b
```

这个时候，即使字段 a 和字段 b 已经分别建立了索引，它们仍然不能同时发挥作用，因为一次查询对于一个数据表只能使用一个索引，它们是无法进行效用叠加的。这样一来，便会存在一定程度的局部行扫描（Range Scan），这在有些特定的场景中将严重影响查询性能，比如上述第一条查询，数据库会先利用字段 a 的索引快速匹配 a=1 的记录，然后在这些记录中筛选 b=2 的记录，而此时 b 字段的索引将爱莫能助，试想，如果 a=1 的匹配行非常多的话，查询时间将花在 b 字段的筛选操作上。

没错，为了应付这样的查询，我们不得不使用组合索引。

我们同样可以通过 explain 来分析组合索引的运行状况，以保证理想和现实的一致，同时，我们还需要了解组合索引的设计原则，这有利于我们更好地设计组合索引。我们来创建这样一个数据表：

```
CREATE TABLE `key_t` (
  `id` int(11) NOT NULL auto_increment,
  `key1` int(11) NOT NULL default '0',
  `key2` int(11) NOT NULL default '0',
  `key3` int(11) NOT NULL default '0',
  PRIMARY KEY  (`id`),
  KEY `normal_key` (`key1`,`key2`,`key3`)
) ENGINE=InnoDB
```

可以看到，我们为这个数据表建立了一个包含 3 个字段的组合索引 normal_key，同时它还有一个自增类型的主键，在 Innodb 类型的表中，主键是必需的。接下来，我们为这个表填充了 100 万行记录，其中 key1、key2、key3 的内容均为 0 到 999 的随机整数。

```
mysql> select count(id) from key_t;
+-----------+
| count(id) |
+-----------+
|   1000000 |
+-----------+
```

好，你一定已经听说过“最左前缀”这个组合索引的基本原则，通过它，我们知道以下几个查询都可以直接使用 normal_key 索引，而不需要任何的行扫描。

```
select * from key_t where key1=1
select * from key_t where key1=1 and key2=2
select * from key_t where key1=1 and key2=2 and key3=3
```

必要的时候，查询优化器还会帮你调整条件表达式的顺序，以匹配组合索引的要求，比如以下这个查询：

```
select * from key_t where key1=1 and key3=3 and key2=2
```

它会被查询优化器理解为：

```
select * from key_t where key1=1 and key2=2 and key3=3
```

优化器这样做的确是很明智的，它减少了开发人员大脑的开销，没有多少人能记得住数据库中字段的顺序。

其次，组合索引对于包含 order by 和 group by 的查询也发挥着重要的作用，它们同样也遵循最左前缀原则，我们看以下这个 SQL 语句的分析：

```
mysql> explain select * from key_t order by key1,key2,key3;
+----+-------------+-------+-------+---------------+-----------+--
-------+------+---------+-------------+
| id | select_type | table | type  | possible_keys | key       | key_len
| ref  | rows    | Extra       |
+----+-------------+-------+-------+---------------+-----------+--
-------+------+---------+-------------+
|  1 | SIMPLE      | key_t | index | NULL          | normal_key |
12      | NULL | 1000262 | Using index |
+----+-------------+-------+-------+---------------+-----------+--
-------+------+---------+-------------+
```

其中 type 为 index，表示这个查询只需要在索引中扫描即可，这里的索引即 normal_key。也就是说，查询语句中 order by 指定的排序规则正好是索引本身的顺序，所以可以直接拿来派上用场，不需要重新排序。值得一提的是，有些非顺序的索引类型（如 Hash），对 order by 是无效的。

下面这个查询正是符合最左前缀的原则，它也使用到了 normal_key 索引。

```
mysql> explain select * from key_t where key1=1 order by key2,key3;
+----+-------------+-------+------+---------------+-----------+---
------+------+------+-------------------------+
| id | select_type | table | type | possible_keys | key       | key_len
| ref   | rows | Extra                   |
+----+-------------+-------+------+---------------+-----------+---
------+------+------+-------------------------+
```

```
    |  1 | SIMPLE       | key_t | ref  | normal_key     | normal_key |
4       | const | 2048 | Using where; Using index |
    +----+-------------+-------+------+----------------+------------+---
------+-------+------+--------------------------+
```

那么，再看看下面这个 SQL 语句：

```
    mysql> explain select * from key_t where key1=1 order by key3;
    +----+-------------+-------+------+----------------+------------+---
------+-------+------+-----------------------------------------------+
    | id | select_type | table | type | possible_keys | key        | key_len
| ref   | rows | Extra                                         |
    +----+-------------+-------+------+----------------+------------+---
------+-------+------+-----------------------------------------------+
    |  1 | SIMPLE       | key_t | ref  | normal_key     | normal_key |
4       | const | 2048 | Using where; Using index; Using filesort |
    +----+-------------+-------+------+----------------+------------+---
------+-------+------+-----------------------------------------------+
```

虽然它用到了 normal_key 索引，但只是对 where 子句起作用，而后面的 order by 则需要排序计算，Using filesort 已经证明了这一点。

对于包含 group by 的查询，数据库一般需要先将记录分组后放置在新的临时表中，然后分别对它们进行函数计算，比如 count()、sum()或 max()等。当有恰当的索引存在时，group by 有时也可以使用索引来取代创建临时表，这当然是我们所希望的。以下这个 SQL 语句便利用了 normal_key 索引，避免了创建临时表。

```
    select count(id) from key_t where key1=777 group by key2,key3
```

而对于另外一些情况，组合索引就无法帮助 group by 了，比如以下的 SQL 语句：

```
    mysql> explain select count(id) from key_t where key1=777 group by key3;
    +----+-------------+-------+------+----------------+------------+---
------+-------+------+-------------------------------------------------
---------+
    | id | select_type | table | type | possible_keys | key        | key_len
| ref   | rows | Extra                                                |
    +----+-------------+-------+------+----------------+------------+---
------+-------+------+-------------------------------------------------
---------+
    |  1 | SIMPLE       | key_t | ref  | normal_key     | normal_key |
4       | const | 2154 | Using where; Using index; Using temporary; Using
filesort |
    +----+-------------+-------+------+----------------+------------+---
------+-------+------+-------------------------------------------------
---------+
```

的确，Using filesort 和 Using temporary 非常不受欢迎，它们越少越好。

小心组合索引的副作用

然而，在有些情况下，组合索引对于一些查询会产生误导，你需要考虑是否应该阻止组合索引，或者预先设计更加适合的索引。同样针对刚才那个数据表，我们看以下这个查询：

```
mysql> select * from key_t where key2=777 limit 10;
+--------+------+------+------+
| id     | key1 | key2 | key3 |
+--------+------+------+------+
| 327233 |  643 |  777 |  781 |
| 686994 |  765 |  777 |  781 |
| 159907 |  766 |  777 |  782 |
|  61518 |  769 |  777 |  780 |
| 274629 |  769 |  777 |  780 |
| 633439 |  769 |  777 |  780 |
| 774191 |  769 |  777 |  780 |
| 109562 |  769 |  777 |  781 |
| 130013 |  769 |  777 |  781 |
| 139458 |  769 |  777 |  781 |
+--------+------+------+------+
10 rows in set (0.38 sec)
```

足足花了380ms！也许你觉得这很正常，因为根据最左前缀原则，以上的查询没有可使用的索引，所以要进行全表扫描，必然花费很长的时间。但是，现实总喜欢折磨我们，似乎并不是我们想象的那样，请看下面的分析：

```
mysql> explain select * from key_t where key2=777 limit 10;
+----+-------------+-------+-------+---------------+------------+---------+
-------+------+---------+---------------------------+
| id | select_type | table | type  | possible_keys | key        | key_len
| ref  | rows    | Extra                     |
+----+-------------+-------+-------+---------------+------------+---------+
-------+------+---------+---------------------------+
|  1 | SIMPLE      | key_t | index | NULL          | normal_key |
12      | NULL | 1000417 | Using where; Using index |
+----+-------------+-------+-------+---------------+------------+---------+
-------+------+---------+---------------------------+
```

竟然使用了normal_key索引，而且type为index，表示这里进行了索引扫描，显然优化器认为在索引中扫描要比在全表数据中扫描更加高效，但是，这次它的如意算盘打错了。

其实，单从上面的查询结果中已经可以看出它使用了normal_key索引，仔细看，这10行记录是按照key1字段来顺序排列的，这说明查询是基于normal_key索引的扫描，而不是基于数据本身的扫描。Innodb类型表中数据的存储顺序是按照主键来排列的。

现在我们将这个数据表转为MyISAM类型。

```
mysql> alter table key_t type=myisam;
Query OK, 1000000 rows affected, 1 warning (33.49 sec)
Records: 1000000  Duplicates: 0  Warnings: 0
```

这时候我们再次执行同样的查询，结果如下所示：

```
mysql> select * from key_t_myisam where key2=777 limit 10;
+------+------+------+------+
| id   | key1 | key2 | key3 |
+------+------+------+------+
| 1035 |  771 |  777 |  781 |
| 3175 |  771 |  777 |  781 |
| 4126 |  771 |  777 |  781 |
| 5443 |  770 |  777 |  780 |
| 6066 |  771 |  777 |  781 |
| 6267 |  770 |  777 |  780 |
| 6317 |  770 |  777 |  780 |
| 6496 |  771 |  777 |  781 |
| 8262 |  770 |  777 |  780 |
| 9083 |  771 |  777 |  780 |
+------+------+------+------+
10 rows in set (0.00 sec)
```

再来看看 explain 的结果：

```
mysql> explain select * from key_t_myisam where key2=777 limit 10;
+----+-------------+------+------+---------------+------+---------
+------+---------+-------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  |
rows    | Extra       |
+----+-------------+------+------+---------------+------+---------
+------+---------+-------------+
|  1 | SIMPLE      | key_t | ALL  | NULL          | NULL | NULL    | NULL
| 1000000 | Using where |
+----+-------------+------+------+---------------+------+---------
+------+---------+-------------+
```

type 为 ALL，这次才是货真价实的全表扫描，可它确实比刚才的索引扫描快多了，我们足可以想象当组合索引不能直接发挥作用的时候，反而给查询带来了巨大的负担，一个包含多个字段的组合索引的尺寸可能已经超过了数据本身。

受到这点的启发，我们将数据表转回 Innodb 类型，希望能在查询中避免使用 normal_key 这个重量级的组合索引。同时我们也看到了，Innodb 和 MyISAM 在索引使用中存在一些不同。

我们在查询的尾部增加了 order by id，结果如下所示：

```
mysql> select * from key_t where key2=777 order by id limit 10;
+------+------+------+------+
| id   | key1 | key2 | key3 |
+------+------+------+------+
| 1504 |  772 |  777 |  781 |
| 1986 |  770 |  777 |  781 |
| 4482 |  770 |  777 |  781 |
| 6095 |  772 |  777 |  781 |
| 7325 |  770 |  777 |  782 |
```

```
| 7867 |  771 |  777 |  781 |
| 8108 |  772 |  777 |  781 |
| 8331 |  771 |  777 |  781 |
| 8885 |  771 |  777 |  781 |
| 9186 |  770 |  777 |  780 |
+------+------+------+------+
10 rows in set (0.00 sec)
```

真是神奇，查询时间有目共睹。我们用 explain 分析如下：

```
mysql> explain select * from key_t where key2=777 order by id limit 10;
+----+-------------+-------+-------+---------------+---------+---------
----+------+---------+-------------+
| id | select_type | table | type  | possible_keys | key     | key_len
| ref  | rows    | Extra       |
+----+-------------+-------+-------+---------------+---------+---------
----+------+---------+-------------+
|  1 | SIMPLE      | key_t | index | NULL          | PRIMARY | 4       |
NULL | 1000417 | Using where |
+----+-------------+-------+-------+---------------+---------+---------
----+------+---------+-------------+
```

由于我们指定了 order by id，查询优化器聪明地放弃了 normal_key 索引，而使用主键进行扫描，这基本上相当于全表扫描，但它比在一个不是为自己准备的组合索引中苦苦寻觅要高效得多。

这样一来，查询结果按照 id 字段来排序，查询的性能问题解决了，但是这样的排序方式是你所希望的吗？如果你希望结果仍然按照 key1 排序，这不是什么问题，你可以增加一个包含（key2,key1）字段的组合索引，注意它们的顺序，（key1,key2）索引和（key2,key1）索引完全不同，你必须根据需要来进行抉择，而优化器对此无能为力。

以上我们只是举了个简单的例子，并不是鼓励你用全表扫描取代索引扫描，而是希望借此强调，一定要根据查询的需要来设计有针对性的组合索引，因为在实际应用中，很多查询不像例子中的那么简单，一个量身定制的组合索引肯定要比全表扫描更加高效，这是毋庸置疑的。

使用慢查询分析工具

在开发环境中，我们很容易对所有查询语句进行分析，并建立适当的索引，但这只是纸上谈兵，当到了运行环境后，随着实际数据的积累，查询计算的开销也逐渐增加，也许你会发现有些索引设计得并不理想，不可否认，错误是在所难免的。可是，在运行环境上对各种查询进行 explain 分析显然很不现实，更难的是你不知道何时去分析哪些查询。

当然，也有些时候，你可能会遇到下列的情况：

- 由于时间紧张而忘记了建立索引；

- 由于认为基于某些字段的条件查询不会很频繁，所以没有建立索引；
- 由于认为短期内记录数不会增长太多，所以没有建立索引；
- 由于认为会有其他同事（DBA）来建立索引，所以自己没有做。

没关系，这些都不重要。其实，最好的办法就是记录下运行环境中每次查询的执行时间。

通过 Web 应用中的日志模块来记录所有 SQL 查询的执行时间是个不错的注意，你完全可以将它封装在数据访问层的查询方法中。另一方面，MySQL 也考虑到了我们的需要，它提供了慢查询日志，可以将执行时间超过预设值的所有查询记录到日志中，以供后期分析。

在 MySQL 中开启慢查询日志非常简单，在 my.cnf 中增加以下配置选项：

```
long_query_time = 1
log-slow-queries = /data/var/mysql_slow.log
```

这意味着 MySQL 会自动将执行时间超过 1 秒的查询记录在指定路径的 mysql_slow.log 文件中。除此之外，你还可以将所有没有使用索引的查询也记录下来，只需增加以下选项即可：

```
log-queries-not-using-indexes
```

当然，有时候这些未使用索引的查询会有很多，而且它们并不见得需要太多的计算开销，所以这里我们没有开启这个选项。

这样一来，随着 MySQL 的运行，我们开始不断地收集慢查询日志，在 mysqlreport 的报告中，可以看到有关慢查询的统计：

```
Slow 1 s          722      0.2/s            0.04  %DMS:   0.08  Log:  ON
```

很好，我们已经有一个大概的了解了，数据库平均每 5 秒有 1 个超过 1 秒的查询，它的次数占所有常规查询的 0.08%，看起来问题不是很严重。

接下来我们想看看究竟是哪些查询比较慢，是的，我们都迫不及待想知道答案。尽管 MySQL 提供了 mysqldumpslow，但是我更推荐使用 mysqlsla，它同样是一个第三方的开源工具，有着更加清晰的统计风格。

现在，我们就使用 mysqlsla 来分析慢查询日志，结果如下所示：

```
s-db:/data/var # mysqlsla -lt slow /data/var/mysql_slow.log
Report for slow logs: /data/var/mysql_slow.log
66 queries total, 37 unique
Sorted by 't_sum'
Grand Totals: Time 213 s, Lock 0 s, Rows sent 16.24k, Rows Examined 41.52M
_____________________________________________________________________ 001 ___
Count         : 6  (9.09%)
```

```
    Time          : 30 s total, 5 s avg, 4 s to 6 s max  (14.08%)
    Lock Time (s) : 0 total, 0 avg, 0 to 0 max  (0.00%)
    Rows sent     : 1 avg, 1 to 1 max  (0.04%)
    Rows examined : 3.81M avg, 3.81M to 3.81M max  (55.11%)
    Database      : db_bang_items_grade
    Users         :
            root@ 192.168.2.1 : 100.00% (6) of query, 95.45% (63) of all users
    Query abstract:
    SELECT COUNT(DISTINCT user_name) AS record_sum FROM grade_69 WHERE
bang_id='S' AND user_name != 'S';
    Query sample:
    select count(distinct user_name) as record_sum from grade_69 where
bang_id='69ad17dcff4de7eb' and user_name != '';
    ……
```

完整的分析报告中包括了 10 个查询，以上只列出了第 1 个查询（001），报告中已经非常清楚地记录了这个查询的时间统计，我们看到它的平均执行时间为 5 秒，真是不看不知道，一看吓一跳。同时，下面还显示出了查询表达式和查询实例，接下来要做的你一定很清楚了，那就是拿着查询实例去分析，大多数的慢查询都是因为索引使用不当，其他的原因包括查询语句过于复杂（比如联合查询）或者数据表记录数过多，通过反范式化设计和数据分区可以有效改善这一状况，后面我们会进行介绍。

索引缓存

试想一下，如果能将一本书的目录记在脑子里，那么，要找到某篇内容就不需要再去翻目录了，节省了查找目录的时间，这就相当于将书的索引缓存在了大脑里。

同样的，数据表的索引也可以缓存起来，当然它是缓存在内存中，最理想的情况是所有的索引都可以直接在内存中查找，而不需要访问磁盘。

在 mysqlreport 的报告中，可以看到索引缓存的使用状态：

```
__ Key _________________________________________________________________
Buffer used   100.37M of   2.00G  %Used:   4.90
  Current       1.02G           %Usage:  51.07
Write hit      99.26%
Read hit       98.71%
```

这里的缓存和之前介绍的其他缓存在本质上没什么两样，Write hit 和 Read hit 分别代表了写缓存的命中率和读缓存的命中率，这里都在 98%以上，比较正常。

值得一提的是，这里报告中的索引缓存指的是 MyISAM 类型表的索引，缓存的大小由 MySQL 的 key_buffer 配置选项来指定，那么为它分配多大的空间合适呢？为此，我们得来看看 MyISAM 类型表的存储方式，任何一个 MyISAM 类型的表包括以下三个文件：

frm

存储了表的结构信息。

MYD

存储了表中的数据。

MYI

存储了表中的索引。

MyISAM 类型的表只缓存索引，不缓存数据，也就是说只缓存 MYI 文件，不缓存 MYD 文件。那么，如果计算出所有 MYI 文件的大小，自然就是当前索引缓存区需要的空间大小了，但是显然这不具备可操作性，而且这个数值在不断增长。总之，如果你在大量使用 MyISAM 类型的表，给 key_buffer 预设一个较大的空间，并监控索引缓存的空间使用率是非常重要的。关于 MySQL 的监控，我们会在后续章节中介绍。

由于存在索引写缓存机制，MyISAM 对于索引的写操作必然存在延迟，这是为了减少磁盘访问，但也给 MyISAM 带来了一些隐患，如果数据库崩溃的时候有些索引没有来得及写入磁盘，将会丢失索引，从而导致数据表损坏，尽管 MySQL 提供了修复工具来尽量弥补，但是假如数据表的数量有 5 位数，修复起来可不那么轻松。在这一点上，Innodb 采用预写日志方式（Write-Ahead Logging，WAL）来实现事务，也就是只有当事务日志写入磁盘后才更新数据和索引，这样即使数据库崩溃，也可以通过事务日志来恢复数据和索引。

索引的代价

市场上的每一件东西都有价格，你要想获得它，就得支付相应数额的货币，这个道理很简单。同样，我们始终在面对各种各样的天平，天平的一端是我们想要的东西，另一端就是我们需要付出的代价。

提示：

你要什么都可以，只是你必须付出代价，没有任何东西是不劳而获的。（西班牙名言）

在使用索引的时候，我们同样需要考虑代价，但是不用为代价感到恐惧，我们只需要思考是否值得。

首先，索引会占据更多的磁盘空间，很多时候索引甚至比数据本身还要大，比如刚才我们将拥有 100 万行记录的表转为 MyISAM 类型后，可以看到索引文件（MYI 文件）的大小几乎是数据文件（MYD 文件）的两倍。

```
s-db:/usr/local/mysql/var/book # l -h key_t.*
-rw-rw---- 1 mysql mysql 8.5K 2009-05-01 14:03 key_t.frm
-rw-rw---- 1 mysql mysql  17M 2009-05-01 14:13 key_t.MYD
-rw-rw---- 1 mysql mysql  32M 2009-05-01 15:11 key_t.MYI
```

现如今，存储空间比计算时间要廉价得多，TB 级别的高速磁盘任你选择，而且大多数情况下在磁盘空间写满之前，计算能力的瓶颈早已迫使数据库进行扩展，所以你几乎不用担心索引空间的增长，从这一点上看，牺牲空间换取时间是值得的。

其次，当建立索引的字段发生更新时，会引发索引本身的更新，这将产生不小的计算开销，我们做了一个简单的测试，同样对于刚才的 key_t 表，在不使用组合索引和使用组合索引两种情况下测试插入数据的性能，结果如表 11-1 所示。

表 11-1　使用索引后插入数据的性能比较

索引使用情况	总时间	平均每秒插入记录数
不使用 normal_key 索引	85.302 seconds	11723
使用 normal_key 索引	144.030 seconds	6944

可以看到有接近一倍的差距。同样，对于 update、delete 等查询，一旦涉及索引字段的变更，也会引发索引计算，导致更多的时间开销。

那么，是否使用索引取决于站点的应用和你的权衡，幸运的是，你不是一个人在战斗，mysqlreport 中有一部分数据统计可以给你提供参考：

```
DMS              30.62M     217.0/s          52.71
  SELECT         26.20M     185.7/s          45.10          85.56
  UPDATE          3.72M      26.3/s           6.40          12.14
  INSERT        649.28k       4.6/s           1.12           2.12
  DELETE         54.13k       0.4/s           0.09           0.18
  REPLACE         1.88k       0.0/s           0.00           0.01
```

通过它，我们了解了站点中各种类型查询数量的比例，比如上面告诉我们在所有 DMS 查询中，select 占据了 85.56%，而 update 和 insert 加起来一共不到 15%，那还犹豫什么呢？快使用索引吧，牺牲更新时间换取读取时间是值得的。

最后，索引需要我们花费一些额外的时间来维护，比如前面我们提到的由于 MyISAM 表的索引写缓存而可能导致的索引损坏，就需要我们及时地去手动修复，但是，在可以接受的范围内，牺牲维护时间换取运行时间是值得的。

11.3　锁定与等待

锁机制是影响查询性能的另一个重要因素。当有多个用户并发访问数据库中某一资源的时

候，为了保证并发访问的一致性，数据库必须通过锁机制来协调这些访问。

那么，我们可以认为查询的时间开销主要包括两部分，即查询本身的计算时间和查询开始前的等待时间，所以说索引影响的是前者，而锁机制影响的是后者。显然，我们的目标很明确，那就是减少等待时间。

减少表锁定等待

MySQL 为 MyISAM 类型表提供了表级别的锁定。我们来看 mysqlreport 中关于表锁定的统计：

```
__ Table Locks ________________________________________________________________
Waited          9.17k     0.1/s  %Total:   0.03
Immediate      32.03M   227.0/s
```

这里的统计项不多，很容易理解，Waited 表示有多少次查询需要等待表锁定；Immediate 表示有多少次查询可以立即获得表锁定，同时后面还有一个比例，表示等待表锁定的查询次数占所有查询次数的百分比，这里是 0.03%，非常好，但为什么这么低呢？这需要了解 MyISAM 的表锁定机制。

MyISAM 的表锁定可以允许多个线程同时读取数据，比如 select 查询，它们之间是不需要锁等待的。但是对于更新操作（如 update 操作），它会排斥对当前表的所有其他查询，包括 select 查询。除此之外，更新操作有着默认的高优先级，这意味着当表锁释放后，更新操作将先获得锁定，然后才轮到读取操作。也就是说，如果有很多 update 操作排着长队，那么对当前表的 select 查询必须等到所有的更新都完成之后才能开始。

这样看来，在一些特定的情况下，慢速查询对于整体性能的影响是非常严重的，我们来举个例子。

```
CREATE TABLE `count_t` (
  `id` int(11) NOT NULL auto_increment,
  `count` int(11) NOT NULL default '0',
  PRIMARY KEY  (`id`)
) ENGINE=MyISAM
```

我们为这个表填充了 10 万行记录，其中 count 字段为 0 到 999 的随机整数。现在我们需要一个比较慢速的 update 操作，如下所示：

```
mysql> update count_t set count=count+1;
Query OK, 100000 rows affected (0.95 sec)
Rows matched: 100000  Changed: 100000  Warnings: 0
```

它用时 0.95 秒，的确是够慢的，但是跟前面慢查询日志中消耗了 5 秒的查询相比，这不算什么，所以，永远不要怀疑在运行环境中会存在超乎你想象的慢速查询。

接下来，我们编写 PHP 程序用来执行这个 update 操作，然后用 ab 模拟 10 个并发用户来请求这个 PHP 程序，这使得刚才的 update 操作语句会被执行 10 次，目的是让这个数据表忙于长时间的 update 操作。

在启动 ab 后不久，我们马上在 MySQL 命令行中发起一个 select 查询，如下所示：

```
mysql> select * from count_t where id=1;
```

这个查询并没有返回结果，而像是陷入了深思。这时我们在另一个 MySQL 命令行会话中监视所有线程的状态，因为我们想知道到底发生了什么。show processlist 命令的结果如下所示：

```
mysql> show processlist\G;
*************************** 1. row ***************************
     Id: 1582
   User: root
   Host: localhost
     db: book
Command: Query
   Time: 0
  State: Locked
   Info: select * from count_t where id=1
*************************** 2. row ***************************
     Id: 1953
   User: root
   Host: localhost
     db: NULL
Command: Query
   Time: 0
  State: NULL
   Info: show processlist
*************************** 3. row ***************************
     Id: 1978
   User: root
   Host: localhost
     db: book
Command: Query
   Time: 1
  State: Updating
   Info: update count_t set count=count+1
*************************** 4. row ***************************
     Id: 1980
   User: root
   Host: localhost
     db: book
Command: Query
   Time: 1
  State: Locked
   Info: update count_t set count=count+1
*************************** 5. row ***************************
     Id: 1981
   User: root
   Host: localhost
     db: book
```

```
Command: Query
   Time: 1
  State: Locked
   Info: update count_t set count=count+1
*************************** 6. row ***************************
     Id: 1982
   User: root
   Host: localhost
     db: book
Command: Query
   Time: 1
  State: Locked
   Info: update count_t set count=count+1
*************************** 7. row ***************************
     Id: 1983
   User: root
   Host: localhost
     db: book
Command: Query
   Time: 1
  State: Locked
   Info: update count_t set count=count+1
```

我们看到除了 show processlist 本身的线程之外，还有 5 个 update 线程和 1 个 select 线程，其中只有 1 个 update 线程的状态为 Updating，而其他的 4 个 update 线程和 1 个 select 线程都处于 Locked 状态，这意味着它们正在等待表锁的释放。从这里也可以看出，已经有 5 个 update 操作执行完毕，正在执行第 6 个 update 操作，而 select 查询将会等到所有的 update 执行完毕后才会进行。

终于，select 查询结束了，结果如下所示：

```
mysql> select * from count_t where id=1;
+----+-------+
| id | count |
+----+-------+
|  1 | 3849  |
+----+-------+
1 row in set (7.60 sec)
```

试想一下，如果这个 select 查询来自一个动态内容，那用户一定会发疯的。的确，表级锁定可能会带来这样的后果，这并不代表它一无是处，同样的，这仍然取决于站点的应用，如果大部分查询为读取操作，同时混合一小部分快速的更新操作，那么将不会存在太多的锁等待。现在，你知道为什么前面 mysqlreport 中统计的锁等待次数百分比只有 0.03%了吗？看到以下的查询比例统计，就不难分析出原因了。

```
DMS          30.62M    217.0/s        52.71
SELECT       26.20M    185.7/s        45.10         85.56
UPDATE       3.72M     26.3/s         6.40          12.14
INSERT       649.28k   4.6/s          1.12          2.12
DELETE       54.13k    0.4/s          0.09          0.18
REPLACE      1.88k     0.0/s          0.00          0.01
```

行锁定带来了什么

如果你的站点主要依靠用户创造内容，那么频繁的数据更新在所难免，它将会给 select 等读取操作带来很大的影响，前面我们已经见证了 7.6 秒的 select 查询。这时候你希望使用行锁来解决问题，没错，MySQL 在 Innodb 类型表中提供了行锁的支持，我们将刚才的 count_t 表转为 Innodb 类型，看看有何不同。

我们同样用 ab 模拟 10 个并发用户来执行 update 操作，同时我们监视 MySQL 线程，如下所示：

```
mysql> show processlist\G;
*************************** 1. row ***************************
     Id: 1953
   User: root
   Host: localhost
     db: NULL
Command: Query
   Time: 0
  State: NULL
   Info: show processlist
*************************** 2. row ***************************
     Id: 1999
   User: root
   Host: localhost
     db: book
Command: Query
   Time: 6
  State: Updating
   Info: update count_t set count=count+1
*************************** 3. row ***************************
     Id: 1998
   User: root
   Host: localhost
     db: book
Command: Query
   Time: 6
  State: Updating
   Info: update count_t set count=count+1
*************************** 4. row ***************************
     Id: 1995
   User: root
   Host: localhost
     db: book
Command: Query
   Time: 6
  State: Updating
   Info: update count_t set count=count+1
*************************** 5. row ***************************
     Id: 1994
   User: root
   Host: localhost
     db: book
Command: Query
```

```
 Time: 6
State: Updating
 Info: update count_t set count=count+1
```

可以看到还有 4 个 update 线程在执行，它们的状态都为 Updating，这意味着已经有 6 个 update 操作执行完成。关键时刻到了，这时候我们在 MySQL 命令行中执行一个 select 查询，如下所示：

```
mysql> select * from count_t where id=1;
+----+-------+
| id | count |
+----+-------+
|  1 | 3853  |
+----+-------+
1 row in set (0.00 sec)
```

查询瞬间返回结果，用时 0.00 秒，这可不是什么魔术，即便还有 4 个 update 操作在执行，select 查询还是迅速地获得结果。没错，刚才的 update 和 select 来自不同的线程，并且针对不同的记录行，所以它们可以轻松地并发进行，这就是行锁定真正带给我们的惊喜！

那么，我们再来模拟一个更加接近实际情况的场景，为此，我们用 PHP 编写了以下程序：

```
<?php
$conn = mysql_connect('localhost', 'root', '');
mysql_select_db('book', $conn);
$sql = "update count_t set count=count+1";
mysql_query($sql, $conn);
for ($i = 1; $i <= 10000; ++$i)
{
        $sql = "select * from count_t where id=" . $i;
        mysql_query($sql, $conn);
}
?>
```

不难理解，这段程序做了两件事情，首先它执行一次慢速的 update 操作，然后执行 10000 次快速的 select 查询。我们用 ab 模拟 20 个并发用户，每个用户请求一次程序，通过使用 MyISAM 和 Innodb 两种不同类型的锁定机制，我们来分别记录总执行时间。

首先是 MyISAM 类型表，测试结果的关键部分如下所示：

```
Concurrency Level:      20
Time taken for tests:   28.190 seconds
Complete requests:      20
```

接下来是 Innodb 类型表，结果如下所示：

```
Concurrency Level:      20
Time taken for tests:   22.841 seconds
Complete requests:      20
```

可以看到，使用了 Innodb 类型表的行锁机制后，时间缩短了接近 20%，不难分析，这部分缩短的时间，正是由于在 update 操作的同时进行 select 查询所弥补的。另一方面，在 Innodb 类型表的测试中，select 查询会得到很快的响应，而在 MyISAM 类型表的测试中，所有的 select 查询可能都要等到 20 秒以后才能开始执行，这是无法容忍的。

行锁定真的很好吗

但是，行锁定的好处不是绝对的，很多人认为行锁定就比表锁定要快，这么说是缺乏上下文的，其实，理论上就锁定本身而言，行锁定的开销并不比表锁定小。通过刚才的例子我们知道，在 select 和 update 混合的情况下，行锁定可以很巧妙地解决读和写互斥的问题，在这种场景下使用行锁定是非常适合的。可见，行锁定的价值并不在于它本身的绝对速度，而是存在于特定的应用场景中。

那么，对于其他场景呢？我们编写了这样一个 PHP 程序：

```
<?php
$conn = mysql_connect('localhost', 'root', '');
mysql_select_db('book', $conn);
for ($i = 1; $i <= 1000; ++$i)
{
        $sql = "update count_t set count=count+1 where id=" . $i;
        mysql_query($sql, $conn);
}
?>
```

它仍然使用前面的 count_t 表，而这一次，它将按照顺序更新前 1000 行记录。接下来我们使用 ab 模拟 50 个并发用户，请求 100 次这个 PHP 程序，这意味着一共执行 10 万次 update 操作。首先我们使用 MyISAM 类型表来看看总时间：

```
Concurrency Level:      50
Time taken for tests:   6.656 seconds
Complete requests:      100
```

接下来使用 Innodb 类型表，结果如下所示：

```
Concurrency Level:      50
Time taken for tests:   8.254 seconds
Complete requests:      100
```

的确，使用 Innodb 的行锁定在这里并没有比 MyISAM 的表锁定表现得更好，反而还略有不如。在压力测试的时候，我们通过 show processlist 来监视线程状态，当使用 MyISAM 类型表的时候，我们看到除了一个正在 Updating 的 update 线程之外，其他的 update 线程都在 Locked，而使用 Innodb 行锁定的时候，我们看到所有 update 线程的状态都是 Updating，可为什么它的总时间还不如前者呢？

这里我们需要清楚的是，锁定只是一种逻辑层面的约束，即便是同时拥有 Updating 的状态，也不能加速这些 update 操作的总时间，因为磁盘的物理写操作最终还是依次进行的。

对于刚才的测试，我们不妨将并发用户数提高到 200，请求次数也提高到 200 次，再来看看结果。对于 MyISAM，结果如下所示：

```
Concurrency Level:      200
Time taken for tests:   12.018 seconds
Complete requests:      200
```

对于 Innodb，结果如下所示：

```
Concurrency Level:      200
Time taken for tests:   12.051 seconds
Complete requests:      200
```

可以看到在使用 Innodb 的测试结果中，总时间逐渐接近 MyISAM，但是依旧没有超过它，这并不奇怪，在这种 update 密集型的场景下，行锁定并不是救世主。

另一种情况是涉及全表扫描的查询，比如以下这个 select 查询，where 条件中的 count 字段并没有索引，所以这里只能进行全表扫描。对于 Innodb 的行锁定，扫描需要更多额外的开销，我们在 MySQL 命令行中执行这个查询，如下所示：

```
mysql> select * from count_t where count=77777;
+-------+-------+
| id    | count |
+-------+-------+
| 77777 | 77777 |
+-------+-------+
1 row in set (0.06 sec)
```

可见，它从第一行记录开始扫描，经历了 77777 行记录，终于找到了目标，一共花费了 0.06 秒。

对于 MyISAM 的表锁定，我们也执行同样的查询，结果如下所示：

```
mysql> select * from count_t where count=77777;
+-------+-------+
| id    | count |
+-------+-------+
| 77777 | 77777 |
+-------+-------+
1 row in set (0.03 sec)
```

同样是经历了 77777 行记录找到目标，但是花费的时间是 0.03 秒，只有刚才的一半。

虽然 Innodb 类型表本身的实现机制和 MyISAM 完全不同，比如存储结构、缓存机制等，它们无法公平比较，但这里我们关注的是性能表现，单从这点来看，在以上这些例子中，

似乎找不到什么理由吸引你使用行锁定，的确，表锁定和行锁定各有利弊，而你需要做的是根据站点的应用来灵活地使用它们，但是，不要忘记一个前提，那就是已经正确地使用了索引。

11.4 事务性表的性能

MySQL 中的 Innodb 除了支持行锁定外，还支持事务，这往往也是使用 Innodb 的另一个原因。在一些特定的应用中，程序依赖于数据库提供的事务操作（如 rollback 或者 commit），这给应用程序带来了不少方便。

当然，如果你并没有在应用程序中使用事务操作的打算，而只是看中了 Innodb 的其他特性，比如行锁定、外键以及易于修复等，你仍然可以使用它。虽然人们一提到 Innodb 就会联想到事务，但是大多数的站点使用 Innodb 都不是冲着事务，而是为了理想中的性能，事实上大多数的站点都不需要事务级别的保障。

好，也许行锁定正符合你的站点应用类型，而且由于行锁定带来的其他问题你都可以统统搞定，那么，对于一个事务性表，我们需要注意哪些呢？

前面我们提到过预写日志方式（WAL），这也是 Innodb 实现事务的方法。当有事务提交时，Innodb 首先将它写到内存中的事务日志缓冲区，随后当事务日志写入磁盘时，Innodb 才更新实际数据和索引。这里有一个关键点，那就是事务日志何时写入磁盘。

为此，MySQL 提供了一个配置选项，它有三个可选的值：

innodb_flush_log_at_trx_commit = 1

表示事务提交时立即将事务日志写入磁盘，同时数据和索引也立即更新。这符合事务的持久性原则。

innodb_flush_log_at_trx_commit = 0

表示事务提交时不立即将事务日志写入磁盘，而是每隔 1 秒写入磁盘文件一次，并且刷新到磁盘，同时更新数据和索引。这样一来，如果 mysqld 崩溃，那么在内存中事务日志缓冲区最近 1 秒的数据将会丢失，这些更新将永远无法恢复。

innodb_flush_log_at_trx_commit = 2

表示事务提交时立即写入磁盘文件，但是不立即刷新到磁盘，而是每隔 1 秒刷新到磁盘一次，同时更新数据和索引。在这种情况下，即使 mysqld 崩溃后，位于内核缓冲区的事务日志仍然不会丢失，只有当操作系统崩溃的时候才会丢失最后 1 秒的数据。

上面提到的“写入磁盘文件”和“刷新到磁盘”，我们在介绍磁盘 I/O 基础的时候曾经有过介绍，它们的区别在于前者只是将数据写入位于物理内存中的内核缓冲区，而后者是将内核缓冲区中的数据真正写入磁盘。

显然，将 innodb_flush_log_at_trx_commit 设置为 0 可以获得最佳性能，同时它的数据丢失可能性也最大。这里补充一下，本章中所有涉及 Innodb 的测试都使用了 innodb_flush_log_at_trx_commit=0 的设置。如果你希望尽量避免数据丢失，可以将其设置为 2，它的性能相对于设置为 0 时略有下降，但是会带来更好的数据持久性保障，值得考虑。

另一个重要的配置选项是 Innodb 数据和索引的内存缓冲池大小，MySQL 提供了 innodb_buffer_pool_size 选项来设置这个数值，如果你在 MySQL 中大量使用 Innodb 类型表，则可以将缓冲池大小设置为物理内存的 80%，并持续关注它的使用率，这时候 mysqlreport 又提供了方便。

```
__ InnoDB Buffer Pool ________________________________________________
Usage          1.00G of   1.00G  %Used: 100.00
Read hit      99.84%
```

另外，前面我们曾经提到直接 I/O 以及在 MySQL 中的应用，还记得吗？我们可以这样设置 innodb_flush_method 选项：

```
innodb_flush_method = O_DIRECT
```

这样一来，Innodb 将可以跳过文件系统缓冲区，提高 I/O 性能，同时凭借自身的缓冲池更加高效地工作。

11.5 使用查询缓存

查询缓存的目的很简单，将 select 查询的结果缓存在内存中，以供下次直接获取。在默认情况下，MySQL 是没有开启查询缓存的，我们可以进行以下配置：

```
query_cache_size = 268435456
query_cache_type = 1
query_cache_limit = 1048576
```

这样一来，MySQL 将拥有 256MB 的内存空间来缓存查询结果。对于以 select 查询为主的应用，查询缓存理所当然地起到性能提升的作用，不论是 Innodb 还是 MyISAM，查询缓存都可以很好地工作，因为它在逻辑中位于比较高的层次。我们编写了一段 PHP 代码来比较查询缓存对 select 查询性能的影响，代码如下所示：

```
<?php
$conn = mysql_connect('localhost', 'root', '');
mysql_select_db('book', $conn);
for ($i = 1; $i <= 100; ++$i)
```

```
{
        $sql = "select * from count_t where id=" . $i;
        mysql_query($sql, $conn);
}
?>
```

我们先关闭查询缓存，对它进行压力测试，结果如下所示：

```
Concurrency Level:      500
Time taken for tests:   5.214 seconds
Complete requests:      1000
```

然后我们打开查询缓存，再次进行同样的测试，结果如下所示：

```
Concurrency Level:      500
Time taken for tests:   3.180 seconds
Complete requests:      1000
```

可见，使用了查询缓存后，总时间减少了将近 40%。

但是，查询缓存有一个需要注意的问题，那就是缓存过期策略，MySQL 采用的机制是，当一个数据表有更新操作（比如 update 或者 insert）后，那么涉及这个表的所有查询缓存都会失效。这的确令人比较沮丧，但是 MySQL 这样做是不希望引入新的开销而自找麻烦，所以“宁可错杀一千，不可放过一个”。

这样一来，对于 select 和 update 混合的应用来说，查询缓存反而可能会添乱，我们编写了以下这段 PHP 代码，将 update 和 select 交替执行，代码如下所示：

```
<?php
$conn = mysql_connect('localhost', 'root', '');
mysql_select_db('book', $conn);
for ($i = 1; $i <= 100; ++$i)
{
        $sql = "update count_t set count=count+1 where id=" . $i;
        mysql_query($sql, $conn);
        $sql = "select * from count_t where id=" . $i;
        mysql_query($sql, $conn);
}
?>
```

不使用查询缓存时，测试结果如下所示：

```
Concurrency Level:      500
Time taken for tests:   12.223 seconds
Complete requests:      1000
```

开启查询缓存后，测试结果如下所示：

```
Concurrency Level:      500
Time taken for tests:   14.353 seconds
Complete requests:      1000
```

果然，使用查询缓存后，花费了更多的时间，真是得不偿失。

幸运的是，mysqlreport 也提供了查询缓存的报告，如下所示：

```
__ Query Cache ___________________________________________________________
Memory usage   38.05M of 256.00M  %Used:  14.86
Block Fragmnt   4.29%
Hits           12.74k    33.3/s
Inserts        58.21k   152.4/s
Insrt:Prune  58.21k:1   152.4/s
Hit:Insert      0.22:1
```

通过它，我们可以了解缓存过期的情况，以上的 Insrt:Prune 表示查询结果进入缓存的次数和过期被删除次数的比例，这里的数值显然非常理想，可见在这个应用中对于密集 select 的数据表很少更新，很适合于使用查询缓存。

11.6 临时表

前面我们曾经看到一些查询在分析时出现 *Using temporary* 的状态，这意味着查询过程中需要创建临时表来存储中间数据，我们需要通过合理的索引来避免它。另一方面，当临时表在所难免时，我们也要尽量减少临时表本身的开销，通过 mysqlreport 报告中的 Created Temp 部分，我们可以看到：

```
__ Created Temp __________________________________________________________
Disk table    864.89k     2.0/s
Table           7.06M    16.1/s    Size:  32.0M
File            9.22k     0.0/s
```

MySQL 可以将临时表创建在磁盘（Disk table）、内存（Table）以及临时文件（File）中，显然，在磁盘上创建临时表的开销最大，所以我们希望 MySQL 尽量不要在磁盘上创建临时表。

如果你在 show processlist 中看到某些查询的状态为 Copying to tmp table on disk，这也意味着 MySQL 将临时表从内存转移到磁盘中，以节省内存空间。

在 MySQL 的配置中，我们可以通过 tmp_table_size 选项来设置用于存储临时表的内存空间大小，一旦这个空间不够用，MySQL 将会启用磁盘来保存临时表，你可以根据 mysqlreport 的统计尽量给临时表设置较大的内存空间。

11.7 线程池

我们知道，MySQL 采用多线程来处理并发的连接，通过 mysqlreport 中的 Threads 部分，我们可以看到线程创建的统计结果：

```
__ Threads ____________________________________________________________
Running           2 of    5
Cached            0 of    0      %Hit:      0
Created        6.15M    43.6/s
Slow              0       0/s
```

每秒创建 43.6 个线程，虽然创建线程的开销不值一提，但是当系统比较繁忙的时候，我们当然不希望再给它添麻烦。

一个比较好的办法是在应用中尽量使用持久连接，这将在一定程度上减少线程的重复创建。另一方面，从上面的 Cached=0 可以看出，这些线程并没有被复用，我们可以在 my.cnf 中设置以下选项：

```
thread_cache_size = 100
```

这使得 MySQL 可以缓存 100 个线程。随后我们获得新的 mysqlreport 报告，Threads 部分如下所示：

```
MySQL 5.0.37-log         uptime 5 1:43:29       Wed May  6 23:30:57 2009
__ Connections ________________________________________________________
Max used          101 of  100      %Max: 101.00
Total          19.78M    45.1/s
__ Threads ____________________________________________________________
Running             2 of    5
Cached             95 of  100      %Hit:    100
Created           186     0.0/s
Slow                0       0/s
```

可以看到，MySQL 在长达 5 天多的时间里，平均每秒处理 45.1 个连接，但是一共只创建了 186 个线程，线程池的命中率为 100%。

11.8 反范式化设计

所谓的范式，就是对关系数据库中关系的一定要求，不同程度的要求为不同的范式，满足最低要求的称为第一范式（1NF），在此基础上满足进一步要求的称为第二范式（2NF），以此类推。

在实际的关系模式设计中，我们通常遵循第三范式（3NF），简单地说，第三范式要求在一个数据表中，非主键字段之间不能存在依赖关系，这样一来，它可以避免更新异常、插入异常和删除异常，保证关系的一致性，并且减少数据冗余。

这里我们不是要帮助你设计出符合一定范式的数据表，事实上，只要拥有关系数据库的正常思维，你设计出的关系模式想不是第三范式都难。比如下面这两个表：

（用户 ID，好友 ID）
（用户 ID，用户昵称，用户邮箱，注册时间，联系电话）

这两个表在我们看来非常普通，但它们的确是符合第三范式的。好，现在我们需要查询某个用户的好友名单，当然这个名单上必须列出好友的昵称，而不是好友 ID。对于以上的两个数据表，有两种查询方法：

- 将两个表进行一次联合查询；
- 先在第一个表中查询出所有好友的 ID，然后在第二个表中查询这些 ID 对应的昵称。

显然，这两种方法都需要打开两张表，费了一番工夫，只是为了做一件重复了无数次的事情，那就是根据用户 ID 获得用户昵称。

现在我们修改一下这两个表，如下所示：

（用户 ID，好友 ID，**好友昵称**）
（用户 ID，用户昵称，用户邮箱，注册时间，联系电话）

这样一来，只需要查询第一个表就可以获得所有好友的昵称，它比前面两种方法的开销都要小。

但是，第一个表是不符合范式的，因为好友昵称依赖于好友 ID，这会导致存在关系的不一致，比如当某个好友修改了昵称后，这里却仍然显示修改之前的昵称，这属于更新异常，怎么办呢？其实，情况或许没有想象中的那么严重，因为：

- 在一般情况下，用户修改昵称的行为是非常罕见的，甚至有些站点中不允许用户修改昵称。
- 即便是少数用户修改了昵称，其影响范围能有多大呢？只有他的好友会发现昵称不一致。
- 即便是他的好友看到昵称不一致，也不会影响好友对他的认知，大家会将这种情况默认为“延迟”，并希望短期内解决即可。
- 当然，最终还是要想办法使昵称一致，我们可以通过定期的数据整理来自动修复不一致的数据。

对于这种不符合范式的关系设计，我们称为反范式化（Denormalization），它看起来像是违背常理，反其道而行之，但是它的目的却是符合实际的，那就是将读取数据的开销最小化，为了实现这个目标，它放弃了一定的原则，可谓是“不择手段”。

当然，颠覆了原则就一定会付出或多或少的代价，你做好准备了吗？刚才我们看到的用户

昵称不一致的问题，便是违反范式的后果，还好，我们可以接受，那么更大的代价是什么呢？刚才说到反范式化的目的是减少读取数据的开销，那么，随之带来的是更多写数据的开销，因为反范式化意味着我们需要预先写入大量的数据副本。

一个典型的例子是社交网站中的好友 Feed 功能，你一定非常熟悉，当你的好友或者关注的人发生一些事件的时候，你会在自己的空间看到这些动态，看起来很简单的功能，你将如何实现呢？如果在你每次刷新空间的时候都要对每个好友的最新事件进行一番查询，甚至使用可怕而昂贵的 join 联合查询，当你有 500 个好友的时候，开销可想而知。这个时候，出于性能的考虑，可以使用反范式化设计，将这些动态为所有关注它的用户保存一份副本，当然，这会带来大量的写开销，但是这些写操作完全可以异步进行，而不影响用户的体验。

当然，为了满足由反范式化引发的大量写开销，我们需要保证数据库的写性能，然而，我们也可以使用简单的非关系型数据库（如 key-value 数据库），来存储这些冗余的数据副本，它可以带来更好的写性能。接下来我们会介绍非关系型数据库。

11.9 放弃关系型数据库

有些时候，使用 key-value 数据库更加简单高效，它不同于传统的关系型数据库，与其说它形式新颖，倒不如说它返璞归真，的确，有时候它更加适合于 Web 应用，比如刚刚提到的将反范式化设计引发的大量数据副本存入 key-value 数据库中。

key-value 数据库使用半结构化方式来存储数据，所有数据都只有一个索引，那就是 key，它省去了关系数据库中 SQL 查询处理的开销和其他多余的东西，也正因为如此，key-value 数据库可以实现相对于关系型数据库更加出色的并发性能。

的确，性能是我们这里最为关注的方面，在这点上，MemcacheDB 有着非常不错的表现，它是一个分布式 key-value 数据库，基于 Memcache 传输协议，这意味着所有使用 Memcache 的 Web 应用不需要进行任何修改，就可以直接迁移到 MemcacheDB。同时，MemcacheDB 将 Berkeley DB 作为持久化引擎，其出色的性能表现也正是得益于此。毫不夸张地说，MemcacheDB 是一个充满智慧的项目，因为它将 Memcache 和 Berkeley DB 这两个伟大的开源产品完美地结合起来。

Berkeley DB 本身是一个高性能的开源数据库引擎，它不能通过网络直接访问，而是需要作为函数库链接到应用程序。Berkeley DB 同样支持事务，同时它通过预写日志方式（WAL）大大提高了写性能，这类似于前面提到的 Innodb 中配置 innodb_flush_log_at_trx_commit=0 的情况。

MySQL 曾经也采用 Berkeley DB 作为表引擎，但后来因为 Berkeley DB 被 Oracle 收购，并且改变了其版权许可协议性质，所以 MySQL 后来采用了 Innodb 取而代之。有意思的是，MySQL 后来被 Sun 收购，而就在最近，Sun 又被 Oracle 收购，时至今日，MySQL 和 Berkeley DB 成了同门师兄弟，但今非昔比，MySQL 已经将精力投入到了非常成熟的 Innodb 引擎。

这里不提资本战场的硝烟，我们来对 MemcacheDB 做一个简单的测试，为此我们编写了以下 PHP 代码：

```
<?php
$memcache = memcache_connect('10.0.1.12', 21201);
for ($i = 0; $i < 10000; ++$i)
{
        $memcache->set($i, "High Performance Web", false, 10);
}
?>
```

然后我们用 ab 模拟 10 个并发用户，一共请求 50 次，测试结果中的总时间如下：

```
Concurrency Level:      10
Time taken for tests:   16.703 seconds
Complete requests:      50
```

16.703 秒内一共完成了 500000 次写操作，平均每秒 29935 次。

我们再来跟 MySQL 做个比较，创建以下数据表：

```
CREATE TABLE `message_t` (
  `id` int(11) NOT NULL auto_increment,
  `msg` char(32) NOT NULL,
  PRIMARY KEY  (`id`)
)
```

然后编写 PHP 代码，同样每个请求插入 10000 行记录，通过 10 个线程一共请求 50 次，代码如下所示：

```
<?php
$conn = mysql_connect('localhost', 'root', '');
mysql_select_db('book', $conn);
for ($i = 0; $i < 10000; ++$i)
{
        $sql = "insert into message_t (msg) values('High Performance
Web')";
        mysql_query($sql, $conn);
}
?>
```

测试结果如下所示：

```
Concurrency Level:      10
Time taken for tests:   27.663 seconds
Complete requests:      50
```

平均每秒 18075 次，虽然表现也不赖，但是远远落后于 MemcacheDB。然而并不是所有的数据都适合存储在 key-value 数据库中，这完全取决于站点应用，很多时候我们必须依赖关系查询，而我们这里也只是将 key-value 数据库作为反范式化设计后引发大量写负载的解决方案。

值得一提的是，MemcacheDB 同时很好地封装了 Berkeley DB 的复制功能，你可以通过主从复制来扩展 MemcacheDB 的规模，并且提升可用性。

第 12 章

Web 负载均衡

回顾前面的内容，似乎一直都在回避 Web 规模扩展这个问题，因为我担心过早实施扩展会迷惑我们优化性能的意志。当然，在有些时候进行扩展是显而易见的，比如下载服务由于带宽不足而必须进行的扩展，但是，另一些时候，很多人一看到站点性能不尽如人意，就马上实施负载均衡等扩展手段，真的需要这样做吗？当然这个问题也只有他们自己能回答，除了出于高可用性和就近部署的考虑，大多数情况下这种行为都显得有些过早，也许当你阅读了前面的章节后，你的 Web 服务器已经从 5 台又变回了 1 台，然后你要做的就是回家闭门思过。

那么，是不是一开始就完全不必考虑规模扩展呢？答案完全相反，作为架构师的你，从一开始就要思考未来的扩展计划，并且为扩展而进行架构设计，但是关键在于，你必须能够意识到何时需要实施扩展，并且有足够的数据来证明这种必要性。

值得一提的是，服务器自身硬件的垂直扩展不在我们的讨论之中，这一章我们所谈及的扩展，主要是指水平扩展，我们经常用可扩展性来反映这种扩展能力，所谓可扩展性，实际上是指系统通过扩展规模来提升承载能力的本领，这种本领往往体现在增加物理服务器或者集群节点等方面，可以说，这种本领越强，承载能力可提升的空间就越大。但是，这种本领总是受到或多或少的制约，比如，我们之所以不讨论单机垂直扩展，就是因为单机的扩展能力非常有限，很快就会遇到技术制约，并且随着规模的增大而越来越昂贵，的确，即使最强大的单机也无法满足我们的需要。

> **提示：**
>
> 从哲学的角度看，可扩展的能力也是万物生存之道，《道德经》中有“道生一，一生二，二生三，三生万物”，《易经》中有“无极生太极，太极生二仪（阴阳），二仪生四象（太阳、太阴、少阳、少阴），四象生八卦，八卦生六十四艾，六十四艾生宇宙万象”。可见我们的古人早已将无限扩展视为永恒的大道。

12.1 一些思考

对于 Web 站点的水平扩展，负载均衡是一种常见的手段，在介绍负载均衡的多种实现方法之前，我们先来思考一些问题。

我们先将目光转向一个类比的例子，假如某公司有一个小型团队，需要承担一定的工作量，开始的时候，大家各尽其能，非常轻松地就可以完成工作，不亦乐乎。但是，随着公司的发展，这个团队的工作量逐渐增大，超出了团队成员的承受能力，工作完成质量开始下降。

外包

出于一些考虑，这个团队决定将一部分工作任务外包给其他公司来做，以减轻自己的负担，同时由团队中的一个人负责与外包公司进行长期沟通，这里我们称他为外包接口人。

显然，外包使得团队承载能力的扩展成为可能，而且随着工作任务的继续增加，一家外包公司无法应付，这个团队又找到了更多的外包公司同时进行合作，外包接口人负责与这些公司分别进行沟通。

这样一来，公司只需要花费一个人力和一些费用，就可以完成大量持续增加的工作任务。

接口人

突然有一天，外包接口人由于病假没能来公司上班，这下公司着急了，因为与外包公司的沟通工作需要时刻进行，而只有这名外包接口人熟悉这个工作，这使得原本有序的外包工作不得不受到严重影响，多家外包公司暂时停了下来，直到第二天这位外包接口人回到公司后，外包工作才恢复正常。

显然，这位外包接口人非常关键，他的缺席将会影响整个外包工作的正常进行，这也许是致命的，我们将这种情况称为单点故障（Single Point of Failure），如果公司只依赖一个因素、系统、设备或人，就会暴露出单点故障的隐患，所以，我们应该尽量避免单点故障。

为此，公司为这位外包接口人配备了一名助理，全力协助他的工作，并且当他偶尔不在公司的时候，助理可以很好地充当他的角色继续工作。

工作量分配

刚才说到，外包接口人需要跟多个外包公司进行沟通，并且将工作任务持续不断地分配给他们，那么，在任务分配过程中，可能发生这样一些情况：

- 给有些外包公司分配了太多的任务，其中一些任务没能按时完成；
- 有些外包公司比较闲，可是却没能及时给他们分配任务；
- 有些外包公司业务能力差，却给他们分配了高难度的任务，花费了大量的时间，最后还可能无法完成；

- 有些外包公司业务能力强，却给他们分配了非常简单的任务，支出了不必要的高额外包费用。

如此一来，我们看到一个最优化任务分配的问题，当然，这个问题是需要外包接口人来考虑的，同时，他需要借助一些过程管理方法来掌握各外包公司的进度和状态，了解各外包公司的“负载”，以帮助自己更加有效地分配任务，实现外包工作的负载均衡（Load Balancing，LB）。

风险管理

尽管我们已经在一定程度上避免了外包接口人的单点故障，降低了风险，但是外包公司方面仍然有可能出现问题，在必要的时候，我们需要采取行动，将任务快速转移给其他的外包公司，当然，前提是有足够的备用外包公司可以选择。

从避免单点故障到准备备用方案，都是降低外包工作风险的一系列措施，同时也保障了外包工作的不间断运转，或者称为高可用性（High Availability，HA）。

制约

当更多的工作任务需要外包时，这位接口人的工作有点吃不消了，因为：

- 一个接口人负责大量的任务，与多家外包公司分别进行沟通，这几乎花费了他全部的工作时间；
- 一个接口人管理多家外包公司，已经超出了他的管理能力。

显然，这些原因也是制约接口人处理更多外包工作的因素，这些因素限制了外包工作的无限扩展。

这时，公司决定设立更多的外包接口人，成立一个新的团队，其中每位外包接口人分别负责管理一部分外包公司，并跟进相关的外包工作。最终的外包工作关系图如图 12-1 所示。

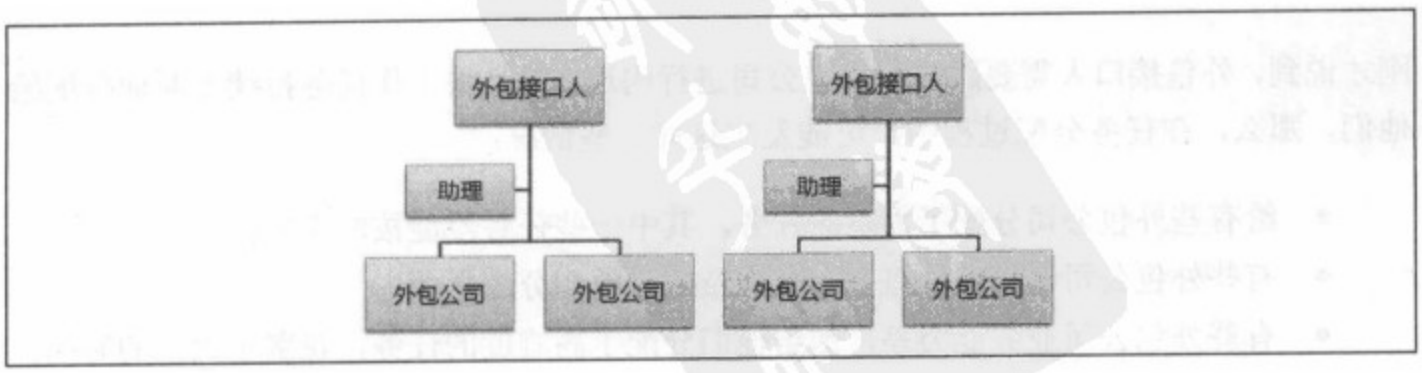


图 12-1　外包工作关系图示例

在以上假想的外包工作场景中，我们遇到了一系列的问题，并提出了解决方案，这些问题在 Web 站点扩展的过程中同样存在，你仍然可以带着这些问题继续前进，接下来我们将回到 Web 站点扩展的主题中，探讨如何实现可扩展的 Web 负载均衡系统，当然，除了我们关注的性能之外，高可用性也会有所涉及。

12.2 HTTP 重定向

对于 HTTP 重定向，你一定不陌生，它可以将 HTTP 请求进行转移，在 Web 开发中我们经常会用它来完成自动跳转，比如用户登录成功后跳转到相应的管理页面。

这种重定向完全由 HTTP 定义，并且由 HTTP 代理和 Web 服务器共同实现。很简单，当 HTTP 代理（比如浏览器）向 Web 服务器请求某个 URL 后，Web 服务器可以通过 HTTP 响应头信息中的 Location 标记来返回一个新的 URL，这意味着 HTTP 代理需要继续请求这个新的 URL，这便完成了自动跳转。当然，如果你自己写了一个 HTTP 代理，也可以不支持重定向，也就是对于 Web 服务器返回的 Location 标记视而不见，虽然这可能不符合 HTTP 标准，但这完全取决于你的应用需要。

也正是因为 HTTP 重定向具备了请求转移和自动跳转的本领，所以除了满足应用程序需要的各种自动跳转之外，它还可以用于实现负载均衡，以达到 Web 扩展的目的。

熟悉的镜像下载

你也许有过从 php.net 下载 PHP 源代码的经历，那么你是否注意过它是如何实现镜像下载的呢？没错，那就是 HTTP 重定向，而镜像下载的目的便是实现负载均衡，值得一提的是，这里我们暂且认为所有镜像服务器上的内容都是一致的，后续章节中我们会介绍内容分发与同步的一些方法和策略。

我们来看这次下载的重定向过程，首先，记住我们请求的 URL 为：

```
www.php.net/get/php-5.2.9.tar.gz/from/a/mirror
```

接下来，我们在浏览器中打开这个地址，并且通过 HttpWatch 监视 HTTP 请求和响应，如下所示：

```
GET /get/php-5.2.9.tar.gz/from/a/mirror HTTP/1.1
Accept: image/gif, image/x-xbitmap,image/jpeg,image/pjpeg, application/
x-shockwave-flash, application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/msword, */*
Referer: http://www.php.net/downloads.php
Accept-Language: zh-cn
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; GTB6;
CIBA; .NET CLR 2.0.50727)
```

```
Host: www.php.net
Connection: Keep-Alive
HTTP/1.1 302 Found
Date: Wed, 13 May 2009 12:08:51 GMT
Server: Apache/1.3.41 (Unix) PHP/5.2.9RC3-dev
X-Powered-By: PHP/5.2.9RC3-dev
Content-language: en
X-PHP-Load: 0.60546875, 0.568359375, 0.5634765625
Location: http://cn.php.net/get/php-5.2.9.tar.gz/from/a/mirror
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
```

可以看到，HTTP 响应的状态码为 302，并通过 Location 标记返回了新的 URL，这个 URL 位于 cn.php.net 这个新的域名下，按照命名规则，我们知道这个域名指向部署在中国的服务器，这样一来，我们刚才向 php.net 主站点发出的下载请求便被转移到了国内的服务器上，这相当于主站点将一部分负载转移到了其他服务器上。

接下来，我们再次请求刚才那个位于 php.net 主站点的 URL，这次获得的 HTTP 响应如下所示：

```
HTTP/1.1 302 Found
Date: Wed, 13 May 2009 12:08:51 GMT
Server: Apache/1.3.41 (Unix) PHP/5.2.9RC3-dev
X-Powered-By: PHP/5.2.9RC3-dev
Content-language: en
X-PHP-Load: 0.60546875, 0.568359375, 0.5634765625
Location: http://cn2.php.net/get/php-5.2.9.tar.gz/from/a/mirror
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
```

这回 Location 中的 URL 发生了变化，域名变成了 cn2.php.net。我们继续重复请求多次，通过 HttpWatch 的截图（如图 12-2 所示）我们可以看到这些请求的重定向过程。

Method	Result	Type	URL
GET	302	Redirect to http://cn.php.net/get/php-5.2.9.tar.gz/from/a/mirror	http://www.php.net/get/php-5.2.9.tar.gz/from/a/mirror
1 request			
GET	302	Redirect to http://cn2.php.net/get/php-5.2.9.tar.gz/from/a/mirror	http://www.php.net/get/php-5.2.9.tar.gz/from/a/mirror
1 request			
GET	302	Redirect to http://cn.php.net/get/php-5.2.9.tar.gz/from/a/mirror	http://www.php.net/get/php-5.2.9.tar.gz/from/a/mirror
1 request			
GET	302	Redirect to http://cn2.php.net/get/php-5.2.9.tar.gz/from/a/mirror	http://www.php.net/get/php-5.2.9.tar.gz/from/a/mirror
1 request			

图 12-2　php.net 站点的 HTTP 重定向监视

可见，php.net 使用了两台位于国内的镜像服务器，分别对应以下的 URL：

```
cn.php.net/get/php-5.2.9.tar.gz/from/a/mirror
cn2.php.net/get/php-5.2.9.tar.gz/from/a/mirror
```

通过图 12-2 我们看到两个镜像地址轮流上阵，但实际上它们并不是严格地交替出现，通过持续观察，你会发现它们似乎是随机出现，但是从整体来看，这已经做到了一定程度上的负载均衡。

在这里，php.net 主站点负责进行地域来源判断以及选择镜像服务器，值得一提的是，这里的重定向方案，在实现负载均衡的同时，也达到了就近访问的目的，加快了用户的下载速度，并且在一定程度上避免了国际带宽的浪费。

当然，通过不同的地域来源来转移请求只是负载均衡的一种策略，在网络 I/O 成为主要瓶颈（比如下载服务）的时候，这种策略的优势表现得尤为突出。但是在其他一些时候，通过地域来源来划分请求并不那么合理，比如来自各个地域的请求到达 Web 服务器的响应时间差异不大，并且各地域的请求比例存在较大差异时，这种策略显然不太适合，它会造成各镜像服务器的负载分配不均衡，从而造成资源的浪费。

用代码来实现

相比于随后要介绍的其他负载均衡方法，基于 HTTP 重定向的方式非常简单，基本上完全由 Web 应用程序即可实现，但是它的性能表现如何呢？我们这里来实现一个基于 HTTP 重定向的负载均衡系统。

假设我们的站点域名为 www.highperfweb.com，它指向的 IP 地址为 10.0.1.100，我们希望将所有对于站点首页的请求随机转移到其他三台实际服务器上，为此，我们进行了以下的域名 DNS 设置：

```
www.highperfweb.com.      IN      A       10.0.1.100
www1.highperfweb.com.     IN      A       10.0.1.101
www2.highperfweb.com.     IN      A       10.0.1.102
www3.highperfweb.com.     IN      A       10.0.1.103
```

可以看到，我们为其他三台服务器准备了新的二级域名，不过看起来毫无内涵，这里我们暂时认为这三台服务器都拥有一致的内容。

这样一来，你完全可以告诉用户直接访问某一台服务器，比如 www2.highperfweb.com，已达到分散主站工作量的目的，但是很显然，这样做不会有任何好处，这等于让用户跳过了我们的负载均衡策略，而且也没有多少用户愿意记住这些古怪的域名，架构师永远不要给用户添麻烦。

提示：

在这个例子中的 IP 地址都是内部地址，但在实际应用中，你必须使用用户可以访问的互联网公开 IP 地址。

接下来，我们编写了首页 index.php 的代码，如下所示：

```
<?php
$domains = array(
    'www1.highperfweb.com',
    'www2.highperfweb.com',
    'www3.highperfweb.com'
);
$index = substr(microtime(), 5, 3) % count($domains);
$domain = $domains[$index];
header("Location: http://$domain");
?>
```

看看这段代码，很简单，当用户每次访问站点入口，也就是 index.php 的时候，程序会随机挑选三台服务器中的一台，然后返回重定向指令。我们用 curl 来试试看：

```
s-colin:~ # curl -i www.highperfweb.com
HTTP/1.1 302 Found
Date: Thu, 14 May 2009 02:15:59 GMT
Server: Apache/2.2.11 (Unix) PHP/5.2.1 DAV/2 SVN/1.4.3
X-Powered-By: PHP/5.2.1
Location: www3.highperfweb.com
Content-Length: 0
Content-Type: text/html
```

看来这次 www3.highperfweb.com 这台服务器比较幸运，它被随机选中，在随后的多次请求中，三台镜像服务器对应的域名都分别出现在了 Location 标记中，从整体上看，首页请求转移到三台服务器的次数基本上是相等的，所以我们已经做到了一定程度的均衡。

这里为什么没有采用依次轮询的重定向策略呢？也就是 RR（Round Robin），它是一种基本的调度算法，在这个例子中如果采用 RR 方式的话，站点首页程序必须将所有请求按照顺序依次转移给三台服务器，实现绝对意义上的均衡，但是，与此同时，性能上的代价也是不可避免的，因为：

- HTTP 本身是无状态的，如果要实现按顺序转移请求，我们必须将最后一次转移至的实际服务器序号进行持久化保存，以便在多次 HTTP 请求之间可以共享，比如将它存入 Memcache，显然，这将会带来额外的开销；
- 要实现绝对的按顺序转移，必然会使得主站点请求转移计算的并发性大打折扣，因为转移状态（最后一次转移至的实际服务器序号）是互斥资源，转发程序必须通过一定的锁机制来保证任何时刻只能有一个请求可以修改它。

这就好比一个人做事，独立决策并执行的速度肯定要比上报领导等待批准快。那么，你可能需要在两者之间进行权衡，随后我们将通过测试来比较一下它们的性能差异。

重定向的性能和扩展能力

这里我们指的性能，实际上是针对主站点来说的，因为它负责转移请求到其他服务器，就像前面故事中的外包接口人一样重要，决定着整个负载均衡系统的扩展能力，也意味着整个系统的最大承载能力。

对于刚才的随机转移方案，我们通过 ab 对主站点的首页程序进行压力测试，结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        www.highperfweb.com
Server Port:            80
Document Path:          /
Document Length:        0 bytes
Concurrency Level:      500
Time taken for tests:   1.553166 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Non-2xx responses:      10000
Total transferred:      2440244 bytes
HTML transferred:       0 bytes
Requests per second:    6438.46 [#/sec] (mean)
Time per request:       77.658 [ms] (mean)
Time per request:       0.155 [ms] (mean, across all concurrent requests)
Transfer rate:          1534.29 [Kbytes/sec] received
```

我们看到吞吐率为 6438.46reqs/s，这意味着什么呢？我们知道主站点的这个首页程序只是负责转移请求，刚才也看到了它的代码实现，它的开销并不大，但是位于其他三台服务器上实际首页的吞吐率如何呢？也许它们由于频繁的数据库访问而只有几百的吞吐率，也许通过实施一些缓存策略可以达到几千的吞吐率，你甚至可以花点心思通过完全静态化来达到上万的吞吐率，前面的章节对此已经有很详细的介绍，但是在这里，这个问题其实并不是我们所关心的，而我们关心的是主站点的最大吞吐率和它的扩展能力有什么关系。

其实这很简单，通过 HTTP 重定向的原理以及我们采取的转移均衡策略，不难得知，当主站点首页的吞吐率达到 6438.46reqs/s 这个极限时，它转移给其他三台服务器的请求均相当于这个吞吐率的 1/3，即 2146.15reqs/s，那么，我们要做的就是保证这三台服务器能够承载这样的压力吗？当然如果你能做到的话最好，但如果不行，那也正常，不要忘了，扩展是把握在我们自己手里的。

我们假设这三台实际服务器的首页能承受的最大吞吐率为 500reqs/s，那么理论上当主站点的实际吞吐率小于 1500reqs/s 的时候，实际服务器可以在承受范围内提供服务。当主站点的实际吞吐率大于 1500reqs/s 后，我们便需要增加实际服务器，但是理论上最多可以增加到 13 台，这取决于主站点首页的最大吞吐率（6438.46reqs/s），它几乎是实际服务器上首页最大吞吐率的 13 倍。

RR 策略下的性能

还记得前面提到的 RR 方式吗？我们现在对主站点首页程序进行修改，希望它能够更加均衡地转移请求到多台实际服务器，代码如下所示：

```
<?php
$memcache = memcache_connect('10.0.1.200', 11711);
$domain_index = $memcache->increment('last_index', 1);
if ($domain_index === false || $domain_index > 100000)
{
        $memcache->set('last_index', 0, null, 0);
        $domain_index = 0;
}
$domains = array(
    'www1.smartdeveloper.cn',
    'www2.smartdeveloper.cn',
    'www3.smartdeveloper.cn'
);
$domain = $domains[$domain_index % count($domains)];
header("Location: http://$domain");
?>
```

可以看到，我们利用 Memcache 服务器创建了一个共享计数器 last_index，在每次转移请求之前，都要对计数器进行递增操作，这是一个原子操作，所以保证了计数器对于所有请求的一致性。我们来对这个新的首页程序进行压力测试，结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        www.highperfweb.com
Server Port:            80
Document Path:          /
Document Length:        0 bytes
Concurrency Level:      500
Time taken for tests:   2.935200 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Non-2xx responses:      10000
Total transferred:      2439868 bytes
HTML transferred:       0 bytes
Requests per second:    3406.92 [#/sec] (mean)
Time per request:       146.760 [ms] (mean)
Time per request:       0.294 [ms] (mean, across all concurrent requests)
Transfer rate:          811.53 [Kbytes/sec] received
```

这次的首页吞吐率只有前边的 53%，而实际服务器的最大数量也由前面的 13 台变为 7 台，这意味着整个系统的承载能力缩水了一半左右。

如此大幅度的性能下降，也印证了前面我们对于顺序转移性能代价的分析，另一方面，也许你觉得这与我们使用 Web 应用程序来实现 RR 调度策略有关，那么，我们来看看 Apache 的 mod_rewrite 模块，它可以轻松地支持 RR 重定向，因为 Web 服务器要维护一个全局资源并不困难。

我们对 Apache 的 Vhost 进行以下配置：

```
<VirtualHost *:80>
    DocumentRoot /data/www/highperfweb/htdocs
    ServerName www.highperfweb.com
    RewriteEngine on
    RewriteMap    lb      prg:/data/www/lb.pl
    RewriteRule   ^/(.+)$ ${lb:$1}
</VirtualHost>
```

这里可以看到我们引入了一个第三方脚本 lb.pl，它是 mod_rewrite 支持的一种可编程模式，这个脚本会跟随 Apache 一起启动，并在自己的逻辑空间中运行，直到 Apache 停止后被释放。我们来看看这个脚本的代码，如下所示：

```
#!/usr/bin/perl
$| = 1;
$name   = "www";
$first  = 1;
$last   = 3;
$domain = "highperfweb.com";
$cnt = 0;
while (<STDIN>)
{
    $cnt = ($cnt + 1) % $last;
    $server = sprintf("%s%d.%s", $name, $cnt + $first, $domain);
    print "http://$server/$_";
}
```

这个脚本不难理解，Apache 每次接入新的请求后，便会触发 while 循环，这时候程序会计算出当前应该使用的实际服务器序号，然后根据事前定义的规则组合成最终实际服务器的域名。我们再次通过 ab 进行同样的压力测试，结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        www.highperfweb.com
Server Port:            80
Document Path:          /
Document Length:        223 bytes
Concurrency Level:      500
Time taken for tests:   2.956 seconds
Complete requests:      10000
Failed requests:        4035
   (Connect: 0, Receive: 0, Length: 4035, Exceptions: 0)
Write errors:           0
Non-2xx responses:      10000
Total transferred:      4657753 bytes
HTML transferred:       2246617 bytes
Requests per second:    3383.02 [#/sec] (mean)
Time per request:       147.797 [ms] (mean)
Time per request:       0.296 [ms] (mean, across all concurrent requests)
Transfer rate:          1538.79 [Kbytes/sec] received
```

从结果来看，吞吐率并没有提高，而且更重要的是，在这次压力测试中，失败率非常高，

Length 为 4035，达到总请求数的 40%左右，这意味着 ab 认为有 4035 个请求的响应数据长度可疑，也就是说这些请求的响应数据可能不是预期的正确结果。为了获得失败的处理结果，我们在压力测试的同时，通过其他会话进行请求，偶尔会获得如下所示的结果：

```
s-colin:~ # curl -i http://www.highperfweb.com/
HTTP/1.1 400 Bad Request
Date: Wed, 20 May 2009 08:37:22 GMT
Server: Apache/2.2.11 (Unix) PHP/5.2.1 DAV/2 SVN/1.4.3
Content-Length: 226
Connection: close
Content-Type: text/html; charset=iso-8859-1
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.<br />
</p>
</body></html>
```

可见，这种失败的处理结果是非常严重的。我们将压力测试的模拟并发用户数从 500 降到了 2，失败率仍然为总请求数的 10%左右，当我们将模拟并发用户数降为 1 的时候，全部请求都处理成功，结果如下所示：

```
Server Software:        Apache/2.2.11
Server Hostname:        www.highperfweb.com
Server Port:            80
Document Path:          /
Document Length:        223 bytes
Concurrency Level:      1
Time taken for tests:   2.653 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Non-2xx responses:      10000
Total transferred:      4740000 bytes
HTML transferred:       2230000 bytes
Requests per second:    3769.66 [#/sec] (mean)
Time per request:       0.265 [ms] (mean)
Time per request:       0.265 [ms] (mean, across all concurrent requests)
Transfer rate:          1744.94 [Kbytes/sec] received
```

的确，我们看到 Apache 在并发环境下对于 mod_rewrite 模块的可编程脚本机制的处理很不理想，当然，这取决于 Apache 在该处的实现。理论上在 Web 服务器中实现专用的重定向策略逻辑，其性能肯定要优于通过 Web 应用程序来实现。也许你可以自己编写专用的 Web 服务器或者插件来实现高效的 RR 重定向，但是，无论如何，顺序调度的性能总是比不上随机调度的性能，特别是在这里，负责转移请求的逻辑本身并没有太大开销，所以更加凸显了顺序调度的性能代价。

如果你使用 RR 策略的目的在于希望做到请求转移的绝对均衡，那么可以肯定的一点是，根据概率统计理论，随着吞吐率的增加，随机调度也会逐渐趋近于顺序调度的均衡效果。所以，你会考虑使用需要一定性能代价的 RR 方式吗？

这里顺便提一下，你也可以考虑使用其他的调度策略，比如根据用户的 IP 地址来散列计算实际服务器序号，这样做同样可以避免 RR 策略的性能代价，事实上凡是 HTTP 请求处理逻辑本身可以独立决策的调度策略，都拥有较高的性能表现。另外，这种根据用户 IP 地址来转移请求的策略可以使得用户每次都访问同一台实际服务器，这种策略在后面我们会再次介绍，它为一些 Web 应用提供了很好的支持。

更多考虑

前面我们费尽心思来实现请求转移次数的均衡，我们知道这样做是没错的，它可以为多台实际服务器平均分配工作量，最大程度地提高利用率。但是在实际环境中，次数的均衡真的代表负载的均衡吗？我们必须思考这个问题。

你是否还记得我们刚才的例子都是针对站点首页的重定向，也就是站点的入口，而站点内页的链接地址我们一般在首页上采用相对地址或者根相对地址，这样一来，一旦用户通过入口时被转移到某台实际服务器后，用户的一系列请求将直接进入这台实际服务器，我们知道不同用户对站点内页的访问深度是不同的，这也是我们无法控制的，这样一来，多台实际服务器的负载差异是不可预料的，而主站点对此却一无所知。

而更加让人无奈的是，你无法保证用户始终从站点首页进入站点，也就是说总有或多或少的用户会跳过你精心设计的主站点调度程序，这的确让人很尴尬，也许用户已经收藏了某个实际服务器上的 URL，比如：

```
http://www3.highperfweb.com/book/load_balancing.php
```

随后即便是你改变了转移策略，但这个 URL 仍然会指向原来的实际服务器，你对它毫无办法，只能想出各种歪门邪道的办法来再次进行重定向，于是策略开始变得混乱和不可控。

所以，大多数情况下通过重定向来实现整个站点的负载均衡，并不那么让人满意，随后我们会探讨其他的扩展方法来实现站点负载均衡。

相比之下，对于文件下载、广告展示等一次性的请求，主站点调度程序可以牢牢地把握控制权，实际服务器的 URL 甚至可以含蓄地隐藏起来，与此同时，这种一次性的请求，也比较容易让多台实际服务器保持均衡的负载，但是也必须考虑一些现实的问题，比如分配给不同实际服务器下载的文件可能尺寸差异较大，我们需要在次数分配均衡的情况尽量保证文件尺寸分配均衡，也就是带宽使用分配均衡，这也许需要借助于应用程序，你可以记

录下给每个实际服务器派发的下载文件的尺寸，从而在每次下载转移前挑选你认为比较轻松的实际服务器，但这样做存在风险，你可能是在毫无根据地指手画脚，因为某个用户可能请求下载一个 1GB 的文件，却在下载了 1%后突然终止，而你却毫不知情，仍然以为某台实际服务器在卖力工作，随后一段时间你对这台服务器实施保护，而它却无所事事。

为了使提供下载服务的多台实际服务器比较均衡地使用带宽，另一个方法值得考虑，事实上它也是负载均衡系统中比较重要的一部分，那就是负载反馈。在这里，我们可以让主站点的定时任务不断获取每个实际服务器的实时流量，这可以通过 SNMP 获得原始数据并计算得出，这些数据将作为下载转移的权重参考。也许你觉得在请求转移逻辑中加入各实际服务器流量权重分析会带来额外的开销，但是，相比于下载的时间开销而言，这些额外的开销实在是九牛一毛。

刚才提到的下载服务只是一个例子，除此之外，对于不同的应用场景，我们仍然需要认真考虑基于重定向的负载均衡是否适用，虽然我们不能一一列举，但是有一点是可以肯定的，我们需要权衡转移请求的开销和处理实际请求的开销，前者相对于后者越小，那么重定向的意义就越大，刚才的下载转移就是个很好的例子。

12.3 DNS 负载均衡

我们知道，DNS 负责提供域名解析服务，当我们访问某个站点时，实际上首先需要通过该站点域名的 DNS 服务器来获取域名指向的 IP 地址，在这一过程中，DNS 服务器完成了域名到 IP 地址的映射，同样，这种映射也可以是一对多的，这时候，DNS 服务器便充当了负载均衡调度器（也称均衡器），它就像前面提到的重定向转移策略一样，将用户的请求分散到多台服务器上，但是它的实现机制完全不同。

多个 A 记录

在 DNS 的各种记录类型中，A 记录你一定不陌生，它负责实现 DNS 的基本功能，用来指定域名对应的 IP 地址，常见的比较成熟的 DNS 系统如 Linux 的 bind，以及 Windows 的 DNS 服务等，都支持为一个域名指定多个 IP 地址，并且可以选择使用各种调度策略，常见的便是 RR 方式。

我们先来看看其他站点的 A 记录设置，这毫无疑问是公开的，不存在任何私密，我们使用 dig 命令来查看 facebook.com 的 A 记录设置，如下所示：

```
s-colin:~ # dig facebook.com
; <<>> DiG 9.3.2 <<>> facebook.com
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 48314
```

```
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 4, ADDITIONAL: 4
;; QUESTION SECTION:
;facebook.com.                  IN      A
;; ANSWER SECTION:
facebook.com.           2254    IN      A       69.63.176.140
facebook.com.           2254    IN      A       69.63.178.11
facebook.com.           2254    IN      A       69.63.184.142
;; AUTHORITY SECTION:
facebook.com.           34766   IN      NS      dns04.sf2p.tfbnw.net.
facebook.com.           34766   IN      NS      dns05.sf2p.tfbnw.net.
facebook.com.           34766   IN      NS      ns1.facebook.com.
facebook.com.           34766   IN      NS      ns2.facebook.com.
;; ADDITIONAL SECTION:
ns1.facebook.com.       54020   IN      A       204.74.66.132
ns2.facebook.com.       54020   IN      A       204.74.67.132
dns04.sf2p.tfbnw.net.   44300   IN      A       69.63.176.8
dns05.sf2p.tfbnw.net.   44300   IN      A       69.63.176.9
;; Query time: 1 msec
;; SERVER: 202.102.152.3#53(202.102.152.3)
;; WHEN: Thu May 14 10:26:21 2009
;; MSG SIZE  rcvd: 232
```

这里我们首先看粗体部分，可以看到 facebook.com 拥有三个 A 记录，分别指向三个不同的 IP 地址，这意味着什么呢？接下来我们通过 ping 命令来测试 facebook.com 域名的 A 记录解析，连续三次的测试结果如下所示：

```
s-mat:~ # ping facebook.com
PING facebook.com (69.63.176.140) 56(84) bytes of data.
s-mat:~ # ping facebook.com
PING facebook.com (69.63.184.142) 56(84) bytes of data.
s-mat:~ # ping facebook.com
PING facebook.com (69.63.178.11) 56(84) bytes of data.
```

果然，DNS 服务器使用三个不同的 IP 地址来轮流解析 facebook.com 域名，实现了 RR 调度策略。

也许你觉得这里 DNS 服务器的职能有点类似于前面提到的重定向调度程序，的确有点，我们现在就对前面的 DNS 设置做一番修改，让 DNS 来取代所谓主站点的工作，修改后的 DNS 设置如下所示：

```
www1.highperfweb.com.     IN      A       10.0.1.101
www2.highperfweb.com.     IN      A       10.0.1.102
www3.highperfweb.com.     IN      A       10.0.1.103
www.highperfweb.com.      IN      CNAME   www1.highperfweb.com.
www.highperfweb.com.      IN      CNAME   www2.highperfweb.com.
www.highperfweb.com.      IN      CNAME   www3.highperfweb.com.
```

这里我们仍然保留了 www1、www2 和 www3 的 A 记录，同时增加了三个别名（CNAME）记录，可以清楚地看到，www.highperfweb.com 指向了这三个 A 记录，从实际效果来看，以上设置也等同于：

```
www.highperfweb.com.      IN      A       10.0.1.101
www.highperfweb.com.      IN      A       10.0.1.102
www.highperfweb.com.      IN      A       10.0.1.103
```

这两种设置几乎没有什么性能上的差别，但是前一种设置也许可以满足你的一些需要，比如：

- 如果你之前使用了重定向方案，那么你可能希望在一段时期内兼顾到老用户，如果他们愿意，仍然可以通过收藏的类似 www3 等域名来访问站点，并期待你引导他们放弃旧的 URL。
- 当你拥有很多 DNS 记录的时候，这样做使你的维护更加容易，比如你希望多个二级域名都指向同一个 IP 地址，那么你可以用一个别名来代替 IP 地址，随后当你需要变更 IP 地址的时候，只需要修改别名的 A 记录即可。当然，你可以起一些有意义的别名，而不像这里的 www1、www2 和 www3。

可见，DNS 负载均衡的实现主要依赖于 DNS 服务器的设置，如果你的站点拥有自己的 DNS 服务器，那么以上的设置对于 DNS 管理员来说并不困难，但是，大多数站点仍然使用第三方 DNS 服务商，幸运的是，现在有很多 DNS 服务商完全支持多个 A 记录的轮询设置，你可以根据需要来挑选。

扩展能力和可管理性

和前面基于重定向的负载均衡方式相比，基于 DNS 的方案完全节省了所谓的主站点，或者说 DNS 服务器已经充当了主站点的职能。

作为调度器，DNS 服务器本身的性能我们几乎不用担心，因为事实上，DNS 记录可以被用户浏览器或者互联网接入服务商的各级 DNS 服务器缓存，只有当缓存过期后才会重新向该域名的 DNS 服务器请求解析，所以即便是采用了 RR 调度策略，我们也几乎不会遇到 DNS 服务器成为性能瓶颈的问题。

另一方面，我们一般会配备至少两台以上的 DNS 服务器来提高可用性，还记得刚才我们通过 dig 命令获得 facebook.com 的 DNS 服务器列表吗？通过 NS 类型记录我们可以看到，它使用了 4 台 DNS 服务器。

既然负载均衡调度器不存在性能的制约，那么在这种方案下，整个系统的扩展能力理论上将可以被无限放大，比如我们通过 dig 命令看到 www.sina.com.cn 指向了 16 台服务器（甚至更多，但这已经足够了）。

```
www.sina.com.cn.          IN      CNAME   jupiter.sina.com.cn.
jupiter.sina.com.cn.      IN      CNAME   dorado.sina.com.cn.
```

```
dorado.sina.com.cn.     IN      A       60.215.128.148
dorado.sina.com.cn.     IN      A       60.215.128.149
dorado.sina.com.cn.     IN      A       60.215.128.123
dorado.sina.com.cn.     IN      A       60.215.128.124
dorado.sina.com.cn.     IN      A       60.215.128.125
dorado.sina.com.cn.     IN      A       60.215.128.126
dorado.sina.com.cn.     IN      A       60.215.128.127
dorado.sina.com.cn.     IN      A       60.215.128.128
dorado.sina.com.cn.     IN      A       60.215.128.129
dorado.sina.com.cn.     IN      A       60.215.128.130
dorado.sina.com.cn.     IN      A       60.215.128.131
dorado.sina.com.cn.     IN      A       60.215.128.132
dorado.sina.com.cn.     IN      A       60.215.128.133
dorado.sina.com.cn.     IN      A       60.215.128.134
dorado.sina.com.cn.     IN      A       60.215.128.135
dorado.sina.com.cn.     IN      A       60.215.128.136
```

当你不必为扩展担忧的时候，另一方面的问题便开始暴露，如何管理这么多的服务器呢？诸如内容同步、数据共享、状态监控等问题都尤为重要，不过还好，在后续章节中我们会详细探讨这些内容，在这里，它们还不至于阻碍我们对 DNS 负载均衡的热衷。

智能解析

尽管基于 HTTP 重定向的负载均衡系统受到主站点性能的制约，但是不可否认这种方案中的调度策略具有非常好的灵活性，你完全可以通过 Web 应用程序实现任何你能想到的调度策略。

相比之下，为 DNS 服务器开发自定义的调度策略就不那么容易了，但幸运的是，类似 bind 这样的 DNS 服务器软件提供了丰富的调度策略供你选择，其中最常用的就是根据用户 IP 来进行智能解析，这意味着 DNS 服务器可以在所有可用的 A 记录中寻找离用户最近的一台服务器。

当然，如何利用这种策略，完全取决于你，你可以为用户比较集中的一些城市提供专用的服务器，接入城市核心节点，也可以为各互联网运营商网络中的用户提供专用的服务器，并接入该运营商骨干节点，要做到这些，你还需要收集到相应的网络地址分布数据，并添加到 DNS 服务器的智能解析策略中。

拿刚才的 www.sina.com.cn 域名来说，我们用另一台位于其他城市的服务器再次进行 dig 命令操作，结果如下所示：

```
s-web:~ # dig www.sina.com.cn
; <<>> DiG 9.3.2 <<>> www.sina.com.cn
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 37334
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 0
;; QUESTION SECTION:
```

```
;www.sina.com.cn.                IN      A
;; ANSWER SECTION:
www.sina.com.cn.         59      IN      CNAME   jupiter.sina.com.cn.
jupiter.sina.com.cn.     239     IN      A       218.30.66.101
;; Query time: 6 msec
;; SERVER: 218.30.19.40#53(218.30.19.40)
;; WHEN: Fri May 22 14:50:38 2009
;; MSG SIZE  rcvd: 71
```

这次看到的 A 记录指向了一个新的 IP 地址，这说明 DNS 服务器根据我们的来源进行了智能解析。有趣的是，我们再次更换一个城市，结果又发生了变化，如下所示：

```
colin-mini:~ root# dig www.sina.com.cn
; <<>> DiG 9.4.2-P2 <<>> www.sina.com.cn
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 43203
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 0
;; QUESTION SECTION:
;www.sina.com.cn.                IN      A
;; ANSWER SECTION:
www.sina.com.cn.         1       IN      CNAME   jupiter.sina.com.cn.
jupiter.sina.com.cn.     336     IN      A       202.108.33.32
;; Query time: 10 msec
;; SERVER: 10.0.1.1#53(10.0.1.1)
;; WHEN: Fri May 22 14:58:09 2009
;; MSG SIZE  rcvd: 71
```

的确，IP 地址又变了。那么这些策略是否达到了“就近”解析的目的呢？这个任务还是交给你吧，你可以用 ping 命令来测试你的 PC 到这些 IP 地址的响应时间，看看 DNS 服务器返回给你的 IP 地址是否就是最快的那个。

最后，如果你的站点在使用第三方 DNS 服务，也没有关系，有很多提供 DNS 轮询设置的服务商也都提供了智能解析服务，你可以灵活地使用它们。

故障转移

对于负载均衡调度器后端的多台实际服务器，我们完全可以通过监控系统来实时了解它们的状态，一旦发现某台服务器出现故障，这时候就需要立刻将它从调度策略中拿掉，也就是暂停指向该服务器的 DNS 记录，以免用户访问到发生故障的服务器而感到莫名其妙。

对于基于 DNS 的负载均衡系统，要做到这一点的确让人非常头疼，因为有一个现实的问题是，我们一般不会将 DNS 记录的 TTL 设置为 0，这使得所有对 DNS 记录的修改都需要一定时间才能生效，比如一个 DNS 记录的 TTL 为 3600 秒，那么对它的更新最多要过一个小时才会生效，这是我们无法容忍的，当然，用户更无法容忍。

另一方面，如何在意识到故障后的第一时间修改 DNS 记录，也是我们需要考虑的问题，在迫不得已需要容忍 DNS 记录更新延迟的情况下，我们唯一能做的就是尽早修改 DNS 记录。

听起来一点也不难，也许你为站点搭建了专用 DNS 服务器，那么你可以通过修改配置来快速完成任务，如果你是在使用第三方 DNS 服务，也没有关系，通过域名管理平台同样可以完成 DNS 修改工作。但是，这些都得依赖人力，的确，它们显得不够快速和自动化，关键的时候时间就是一切，特别是当需要和监控系统集成实现自动故障转移时，这些方法都显得力不从心。

也许你听过动态 DNS，这其实是 DNS 协议的一个特性（Standard Dynamic update DNS，DDNS，RFC2126），它允许 DNS 服务器开放特定的服务，为我们自动化远程修改 DNS 记录提供了可能。

这让我想起了现在几乎所有宽带路由器都支持的一个功能，那就是动态域名解析，你还有印象吗？当你的主机使用动态 IP 地址接入互联网，并且你希望将某个域名指向这台主机时，所谓的动态域名解析便发挥了作用，它做的事情很简单，就是在每次 IP 地址变更时及时地更新 DNS 服务器，当然，一定的延迟仍然是在所难免的，同样是因为 DNS 记录的 TTL。

利用同样的思路，当我们监测到某台实际服务器发生故障后，便可以通过动态 DNS 协议来迅速修改 DNS 记录。

一些不足

刚才我们已经提到了由于 DNS 记录的缓存带来的更新延迟，这导致我们对于调度器的控制总是跟不上节奏，这或多或少是一件令人遗憾的事情。

相比于 HTTP 重定向方式的调度器，DNS 服务器更像魔术师，它可以在用户面前很好地隐藏实际服务器，没有用户能直接看到 DNS 解析到了哪一台实际服务器，也没有人关心这个，但是与此同时，它也给服务器运维人员的调试带来了一些不便，人们需要通过修改/etc/hosts 来为域名指定某个实际服务器的 IP 地址，以跳过 DNS 服务器变幻莫测的调度。

除此之外，在这种基于 DNS 的负载均衡框架之下，负载均衡调度器工作在 DNS 层面，这导致它的调度灵活性被或多或少地削弱，策略的开发存在一定的局限性，比如你无法将 HTTP 请求的上下文引入到调度策略中，而在前面介绍的基于 HTTP 重定向的负载均衡系统中，调度器工作在 HTTP 层面，它可以在充分理解 HTTP 请求之后根据站点的应用逻辑来设计调度策略，比如根据请求的不同 URL 来进行合理的过滤和转移。

另一方面，根据实际服务器的实时负载差异来调整调度策略，这需要 DNS 服务器在每次解析操作时分析各服务器的健康状态，对于 DNS 服务器来说，这种自定义开发存在较高的门槛，更何况大多数站点只是使用第三方 DNS 服务，根本没有自主开发的可能，所以大多数人只能享受到单一的 RR 调度算法。

事实上，就算是 DNS 服务器能够做到最完美的调度，也不要忘了，DNS 记录缓存的出现将会再次成为毁灭者，各级节点的 DNS 服务器不同程度的缓存会让你晕头转向，如此庞大的体系简直是太复杂了，完全超出我们的分析能力，你得研究地理、人口、城市、交换节点等，从来没有见过如此复杂的事情，还是算了吧。

的确，DNS 服务器充当了一个粗放型的请求调度器，这给我们带来了一些遗憾，在这种情况下，如何让多台实际服务器最大程度地保持比较均衡的负载，是我们需要持续考虑的问题。

这里顺便说一下，所谓的“均衡”，不能狭义地理解为分配给所有实际服务器的工作量一样多，因为有时候，多台服务器的承载能力各不相同，这可能体现在硬件配置、网络带宽的差异，也可能因为某台服务器身兼多职，我们所说的“均衡”，也就是希望所有服务器都不要过载，并且能够最大程度地发挥作用。

这里我们拿点真实的数据来看看，在我过去参与开发的一个站点中，同样是基于 DNS 的负载均衡，采用 RR 调度方式，有两台同样配置的实际服务器 web127 和 web129，作为站点的动态内容服务器，你可以认为这两台服务器的承载能力完全相同，而且我们希望它们能够完成同样的工作量。

我们分别来看看这两台服务器在最近 24 小时内的 WAN 网卡流量和系统负载，首先看看它们的 WAN 网卡流量图，如图 12-3 和图 12-4 所示。

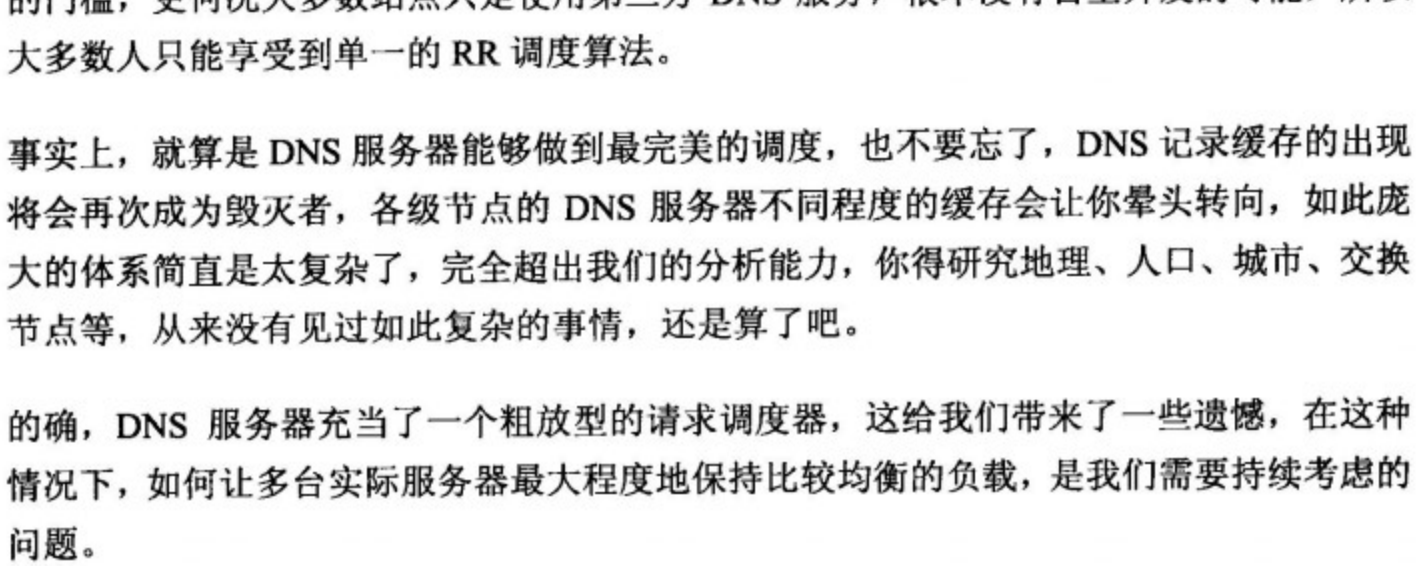


图 12-3　服务器 web127 的 WAN 网卡流量图

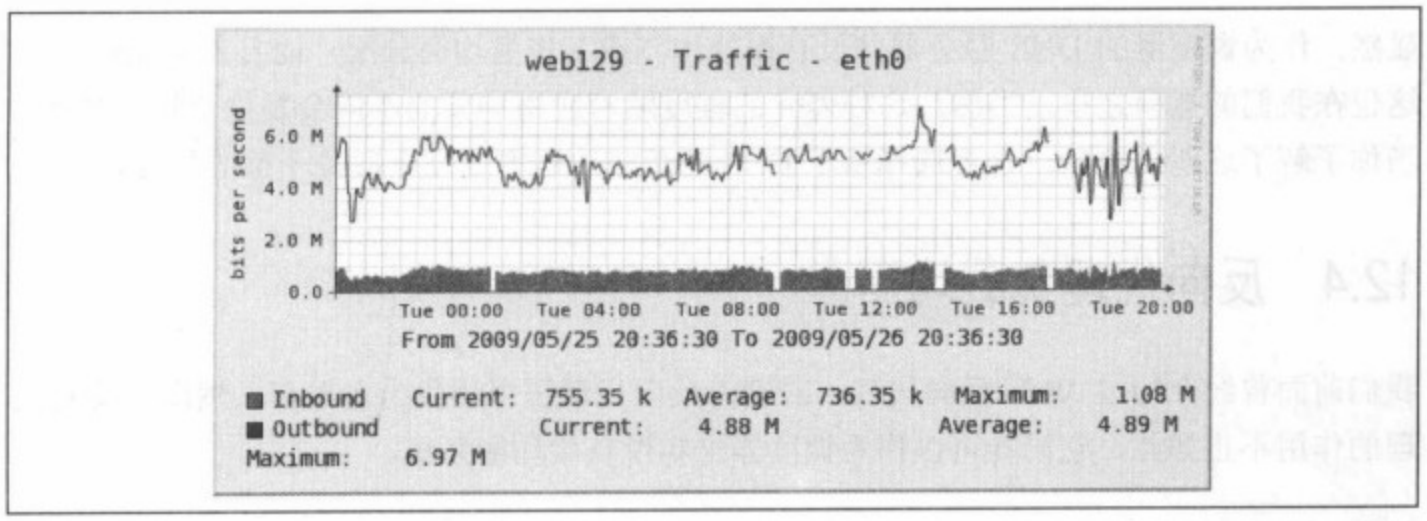


图 12-4　服务器 web129 的 WAN 网卡流量图

注意以上这两幅图的纵坐标比例是不同的，从总体上看，不论是从网卡流入还是流出的数据量，web129 都要明显大于 web127，而且两者的变化趋势几乎没有太多相似的规律。

接下来我们看两台服务器的系统负载图，如图 12-5 和图 12-6 所示。

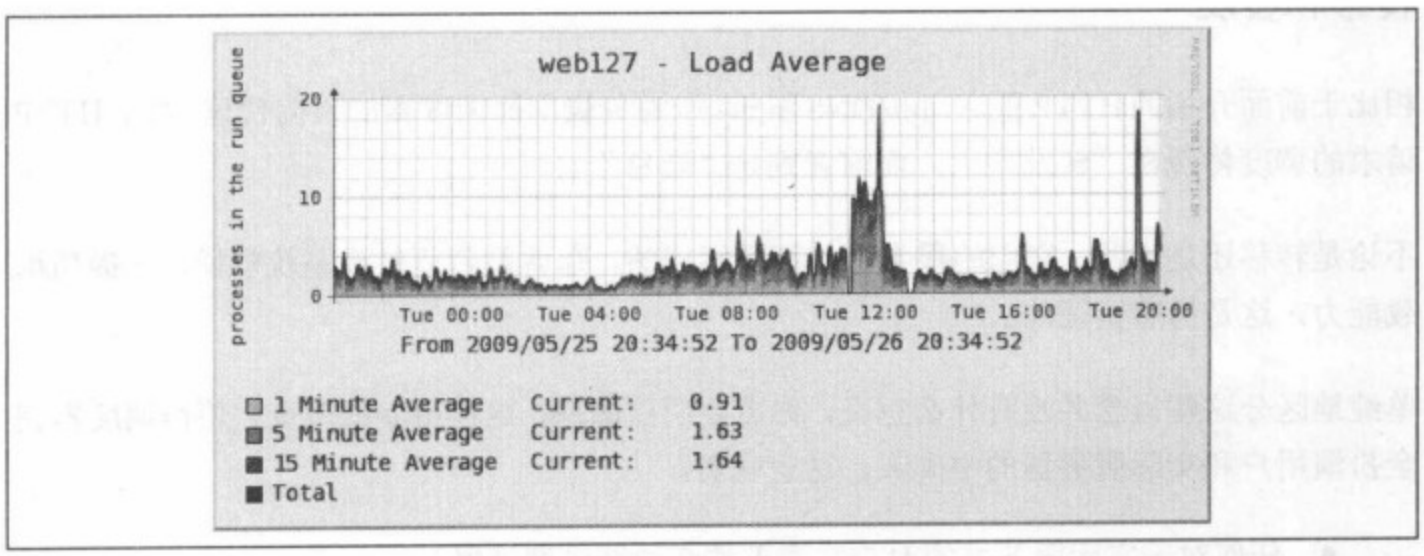


图 12-5　服务器 web127 的系统负载曲线图

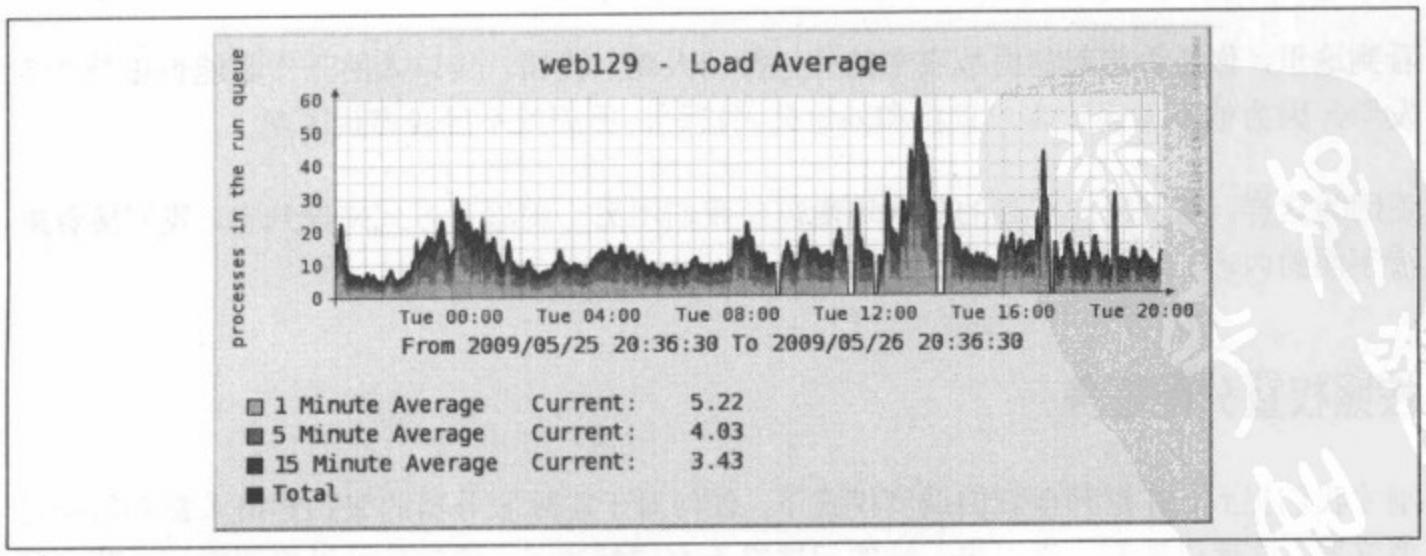


图 12-6　服务器 web129 的系统负载曲线图

显然，作为调度器的 DNS 服务器并没有很好地完成工作量均衡分配，而且差异比较大，这也在我们的意料之中，当然，这种差异的程度随着应用场景的不同会发生变化，总之，当你了解了这些内容后，是否选择基于 DNS 的负载均衡方式完全取决于你的需要。

12.4 反向代理负载均衡

我们前面曾经介绍过 Web 反向代理，的确，反向代理缓存让我为之振奋，然而，反向代理的作用不止如此，它同样可以作为调度器来实现负载均衡系统。

显然，反向代理服务器的核心工作便是转发 HTTP 请求，因此它工作在 HTTP 层面，也就是 TCP 七层结构中的应用层（第七层），所以基于反向代理的负载均衡也称为七层负载均衡，实现它并不困难，目前几乎所有主流的 Web 服务器都热衷于支持基于反向代理的负载均衡，随后我们的介绍中便会不同程度地涉及这些内容。

转移和转发

相比于前面介绍的HTTP重定向和DNS解析，作为负载均衡调度器的反向代理，对于HTTP请求的调度体现在“转发”上，而前者则是“转移”。

不论是转移还是转发，它们的根本目的都是相同的，那就是希望扩展系统规模，来提高承载能力，这是毋庸置疑的。

单纯地区分这些概念并没有什么意义，你需要明白的是，这种机制的改变，使得调度器完全扮演用户和实际服务器的中间人，这意味着：

- 任何对于实际服务器的 HTTP 请求都必须经过调度器；
- 调度器必须等待实际服务器的 HTTP 响应，并将它反馈给用户。

看到这里，你是否想起前面故事中的外包接口人呢？没错，接口人的工作职能也正是“转发”，因为他不希望让客户直接和外包公司接触，原因就不用我多说了吧。

正因为这样，基于反向代理的负载均衡需要我们用另一种思考方式来评判它，我们接着来看下面的内容。

按照权重分配任务

刚才我们提到，在这种全新的调度模式下，任何对于实际服务器的 HTTP 请求都必须经过调度器，这使得我们一直以来苦恼的问题终于有望解决了，那就是可以将调度策略落实到每一个 HTTP 请求，从而实现更加可控的负载均衡策略。

顺便说明一下，在基于反向代理的负载均衡系统中，我们也常把实际服务器称为后端服务器（Back-end Server）。

当不同能力的后端服务器并存的时候，调度器并不希望平均分配任务给它们，这很容易理解，的确，大锅饭时代的结束已经宣告了平均分配的愚昧。

本着能者多劳的原则，有些反向代理服务器可以非常精确地控制分配权重，这里我们用Nginx 作为反向代理服务器，来看看权重设置对于整体吞吐率的影响。

我们准备了两台服务器作为后端，分别为 10.0.1.200 和 10.0.1.201，它们的 80 端口都运行着 Apache。同时，我们用 PHP 编写了一个动态程序，它可以模拟不同程度的计算任务，为了尽可能不依赖其他资源，我们将它设计成 CPU 密集型的程序，代码如下所示：

```
<?php
$num = $_GET['num'];
$sum = 0;
for ($i = 0; $i < $num; ++$i)
{
        $sum += $i;
}
echo $sum;
?>
```

可以看出，这个 PHP 程序可以根据 URL 参数的传递进行不同开销的计算。那么，接下来我们分别来对这两台后端服务器进行压力测试，结果如下所示：

```
Server Software:        Apache/2.2.9
Server Hostname:        10.0.1.200
Server Port:            80
Document Path:          /test.php?num=100000
Document Length:        13 bytes
Concurrency Level:      100
Time taken for tests:   5.076 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      203000 bytes
HTML transferred:       13000 bytes
Requests per second:    197.00 [#/sec] (mean)
Time per request:       507.614 [ms] (mean)
Time per request:       5.076 [ms] (mean, across all concurrent requests)
Transfer rate:          39.05 [Kbytes/sec] received

Server Software:        Apache/2.2.11
Server Hostname:        10.0.1.201
Server Port:            80
Document Path:          /test.php?num=100000
Document Length:        13 bytes
Concurrency Level:      100
Time taken for tests:   13.728 seconds
Complete requests:      1000
Failed requests:        0
```

```
Write errors:           0
Total transferred:      219564 bytes
HTML transferred:       13338 bytes
Requests per second:    72.84 [#/sec] (mean)
Time per request:       1372.829 [ms] (mean)
Time per request:       13.728 [ms] (mean, across all concurrent requests)
Transfer rate:          15.62 [Kbytes/sec] received
```

可以看到，同样是 10 万次的循环计算，两台后端服务器的吞吐率存在很大的差异，这是怎么回事呢？忘了告诉你了，这里我们故意挑选了两台配置不同的服务器，它们的配置如表 12-1 所示。

表 12-1　两台后端服务器的具体配置

后端服务器 IP 地址	CPU	内存
10.0.1.200	Intel 1.60GHz × 2	8GB
10.0.1.201	Intel 1.60GHz	4GB

所以，10.0.1.201 这台服务器的吞吐率显得逊色一些，不过没有关系，权重分配正好能够派上用场。

接下来我们在另一台 IP 地址为 10.0.1.50 的服务器上运行 Nginx，作为反向代理服务器，也就是负载均衡调度器，并为它配置两个后端服务器，如下所示：

```
upstream backend {
    server 10.0.1.200:80;
    server 10.0.1.201:80;
}
```

以上的配置只是这里提到的关键部分，更多的配置你可以查阅 Nginx 的官方文档，里面有很详细的介绍。

经过这样的配置后，Nginx 将任务平均分配给两个后端，虽然我们知道这样做不太明智，但还是希望先来看看这样做的结果。我们对调度器进行同样条件的压力测试，结果如下所示：

```
Requests per second:    144.26 [#/sec] (mean)
```

吞吐率只有 144.26reqs/s，还不如那个 197reqs/s 的后端服务器，的确，罪魁祸首就是另一台后端服务器，它拖了后腿，帮了倒忙，但是也不能怪它，调度器应该对此负责。

接下来，我们来改变权重分配，如下所示：

```
upstream backend {
    server 10.0.1.200:80 weight=2;
    server 10.0.1.201:80 weight=1;
}
```

这意味着让更强的那台服务器比另一台多干一倍的任务，它们的分配权重为 2:1，再来看看测试结果：

```
Requests per second:    224.37 [#/sec] (mean)
```

现在的吞吐率已经超过了任意一台后端服务器的单独成绩，但只是微微超出，这显得另一台后端服务器没帮上什么忙，似乎可有可无，不行，它必须证明自己的存在是有价值的。

接下来，我们将权重分配调整为 3:1，如下所示：

```
upstream backend {
    server 10.0.1.200:80 weight=3;
    server 10.0.1.201:80 weight=1;
}
```

再次进行同样的压力测试，结果如下所示：

```
Requests per second:    266.28 [#/sec] (mean)
```

可以看到，吞吐率又有明显的提升，现在已经比较接近两台后端服务器的独立成绩之和，那么，如果我们继续调整分配权重为 4:1，结果又会如何呢？你一定很想知道，下面我们进行一下调整：

```
upstream backend {
    server 10.0.1.200:80 weight=4;
    server 10.0.1.201:80 weight=1;
}
```

再来看看测试结果：

```
Requests per second:    249.18 [#/sec] (mean)
```

好，我们将这四种权重比例下的整体吞吐率绘制成曲线图，如图 12-7 所示。

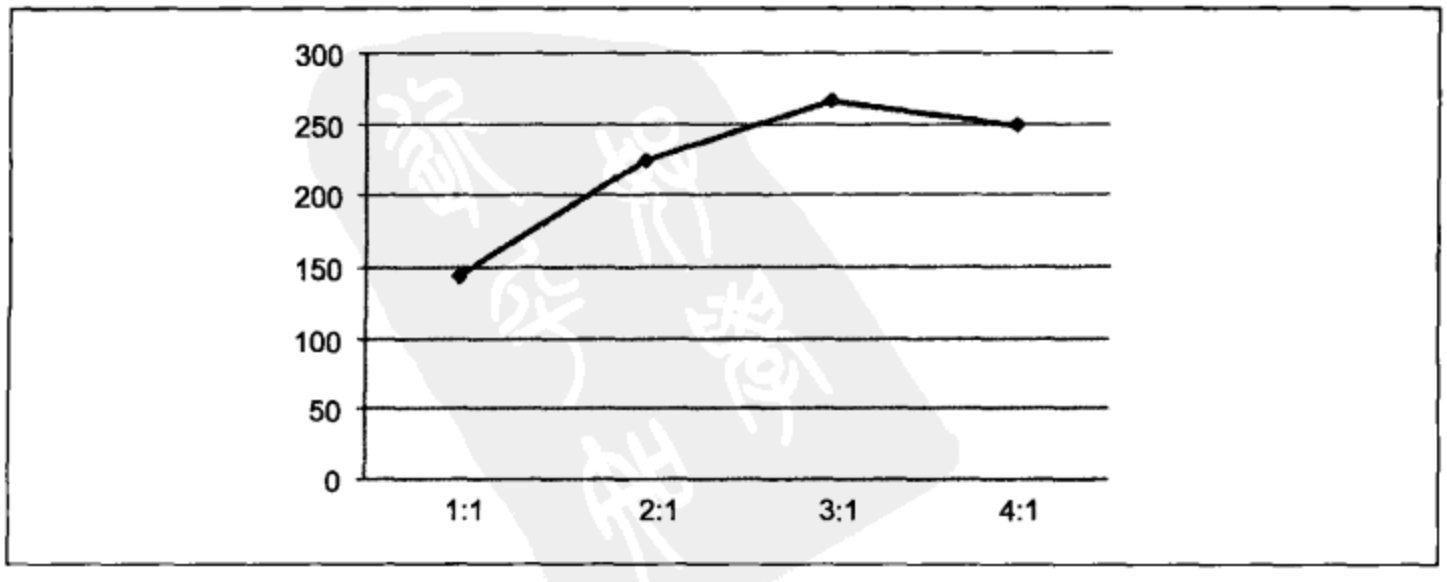


图 12-7　两台后端服务器不同权重分配比例下的吞吐率

事实告诉我们，吞吐率开始下降了，也就是说，对于以上两台后端服务器，当它们的分配权重为 3:1 的时候，整个系统可以获得最佳的性能表现。你也许已经想到了，之所以是 3:1，原因就在于两台后端服务器的独立成绩刚好近似等于 3:1，所以按照它们的能力来分配权重比例，显然可以最大程度地物尽其用。

当然，支持带有权重分配机制的 RR 调度策略，不只出现在 Nginx 中，很多其他的反向代理服务器软件也都将此视为一个必备的内置功能，但是需要强调的是，不论是哪个具体的实现，它们的权重分配机制本身都是相同的，由于这种机制和工作方式的局限，不同实现所表现出的性能几乎不会有太大差异。

这里我们不妨使用 HAProxy 来代替 Nginx，完成同样的工作，并且依次进行各种权重比例下的压力测试。HAProxy 也是一款主流的反向代理服务器，可以作为负载均衡调度器，我们对 HAProxy 进行后端配置，其中关键部分示例如下：

```
listen  proxy_1 10.0.1.50:8003
        mode http
        option httplog
        option dontlognull
        balance roundrobin
        stats uri /hastat
        server backend_1 10.0.1.200:80 weight 3
        server backend_2 10.0.1.201:80 weight 1
```

可以看到，我们将 HAProxy 监听在 8003 端口，然后为它设置了前面用到的两台后端服务器，HAProxy 也同样支持权重分配机制，接下来我们进行同样的压力测试，这里只列出各种权重分配比例下的整体吞吐率，如表 12-2 所示。

表 12-2　HAProxy 在不同权重分配比例下的吞吐率

调度权重分配	1:1	2:1	3:1	4:1
压力测试吞吐率	148.96 reqs/s	207.97 reqs/s	249.00 reqs/s	239.79 reqs/s

的确，在以上四种权重分配比例下，测试结果几乎与使用 Nginx 时不相上下。

所以，这里我们不会将太多的笔墨放在比较或者推荐具体的某个反向代理服务器软件，因为那些都是不停变化的商业产品，其中一些可能经不起时间的考验，而唯有真理和本质是相对持久的，一旦你了解了这些内容之后，至于如何选择，就像去逛超市一样，各家产品都说自己很强大，而你需要的是分析和测量。

调度器的并发处理能力

对于作为负载均衡调度器的反向代理服务器来说，它首先必须是一台经得起考验的 Web

服务器，没错，因为它工作在 HTTP 层面，它的一切工作都得从处理用户的 HTTP 请求开始。

所以，反向代理服务器本身的并发处理能力显得尤为重要，幸运的是，早在第 3 章中，我们就已经对影响服务器并发处理能力的各种因素进行了深入的探讨，你可以再次重温那些内容。

为此，请不要将反向代理服务器放到一个神秘的位置上，一旦你将它视为某种意义上的 Web 服务器，那些你曾经熟悉的概念将再次上演，比如 I/O 模型和并发策略。

这也或多或少地影响了反向代理服务器软件的市场格局，主流的 Web 服务器都争先恐后地支持反向代理机制，比如 Apache、Lighttpd、Nginx 等，因为它们作为 Web 服务器所取得的辉煌成就让它们不费吹灰之力就可以转型成为一台马力强劲的负载均衡调度器。

扩展的制约

到现在为止，作为负载均衡调度器的反向代理服务器似乎出尽了风头，我们接下来将目光转移到另一个重要的方面，那就是这种负载均衡系统的扩展能力。理所当然，作为调度器的反向代理服务器成为我们关注的重点，因为它扮演着接口人的重要角色。

我们知道反向代理服务器工作在 HTTP 层面，对于所有 HTTP 请求都要亲自转发，可谓是大事小事亲历亲为，这也让我们为它捏了一把冷汗，你也许在怀疑它究竟有多大能耐，能支撑多少后端服务器，的确，这直接关系到整个系统的扩展能力。

接下来，我们选择了两台承载能力基本相当的后端服务器，仍然通过反向代理服务器实现负载均衡，它们的网络结构如图 12-8 所示，其中各服务器的用途和 IP 地址如表 12-3 所示。

表 12-3　反向代理负载均衡系统网络结构示意图中的服务器说明

服务器用途	内部网络 IP	外部网络 IP
反向代理服务器	10.0.1.50	125.12.12.12
后端服务器	10.0.1.210	-
后端服务器	10.0.1.201	-

然后，我们对后端服务器和调度器分别进行了一系列的压力测试，值得一提的是，这一次我们并不是在被测试的服务器上执行 ab，而是在与被测试服务器同一网段的其他服务器上执行 ab 进行压力测试，这样可以减少 ab 本身的开销对测试结果的影响，同时也是为了和随后的 IP 负载均衡测试结果保持可比性。

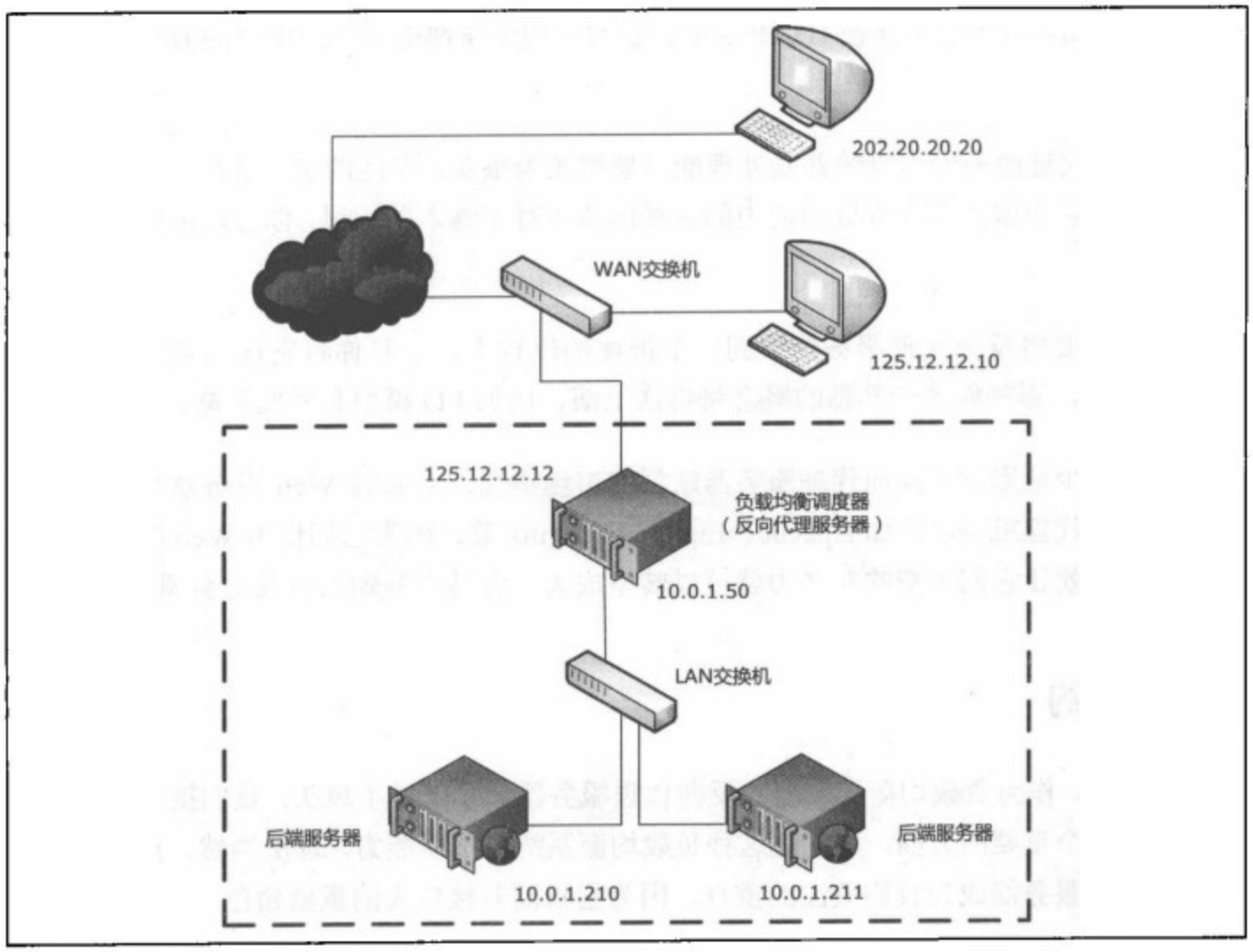


图 12-8　反向代理负载均衡系统网络结构示意图

我们来看看测试结果，如表 12-4 所示。

表 12-4　后端服务器和反向代理服务器分别对于不同内容的压力测试结果

内容类型	10.0.1.210	10.0.1.211	125.12.12.12
静态（10Bytes）	13022.53	13328.71	7778.5
动态（num=100）	10529.97	10642.43	7589.22
动态（num=500）	8244.66	8177.55	7137.96
动态（num=1000）	6950.16	6634.42	6458.53
动态（num=5000）	2771.98	2654.07	4454.4
动态（num=10000）	1444	1478.96	2468.48
动态（num=50000）	322.26	331.79	635.39
动态（num=100000）	151.9	157.48	296.6

在分析这些数据之前，我先来解释一下，在表 12-4 中，左面第一列给出了 8 种 Web 内容，

它们对应着不同程度的 CPU 计算和 I/O 操作，对 Web 服务器来说这意味着处理这些内容将花费不同的开销，其中的动态内容正是前面那个可以根据指定循环次数进行计算的 PHP 程序，在这里它又派上了用场。右面的三列分别列出了两个后端服务器的独立测试结果和整个负载均衡系统的测试结果，当然，它们的单位都是 reqs/s。

为了更好地分析表格中的数据，我们绘制了一幅柱状对比图，如图 12-9 所示。

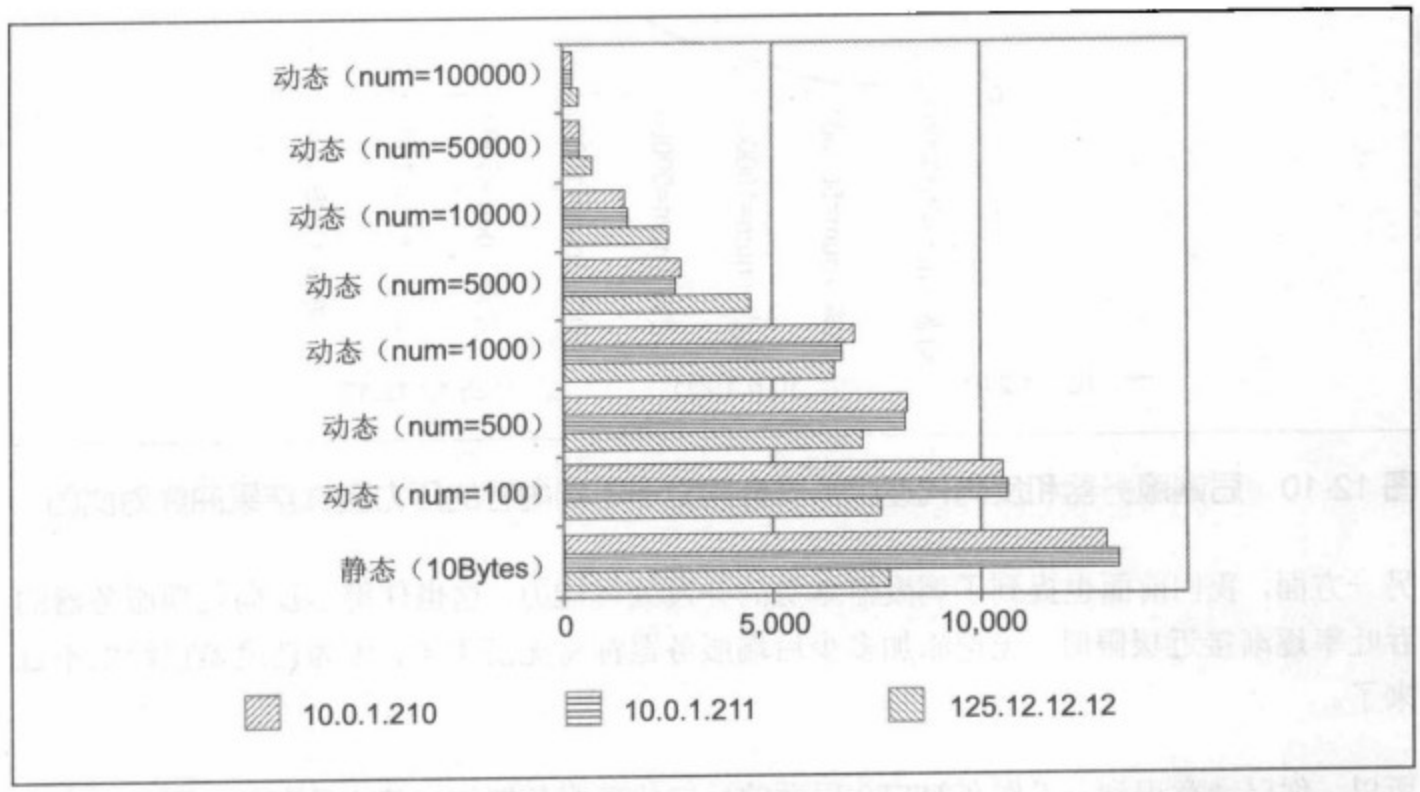


图 12-9　后端服务器和反向代理服务器分别对于不同内容的压力测试结果柱状对比图

从上往下看，内容处理的开销逐渐减少，两台后端服务器的吞吐率逐渐增加，反向代理服务器的吞吐率也随之增加，但是，我们需要注意的是，对于以上各种类型的内容，反向代理服务器的吞吐率是否为两台后端服务器的吞吐率叠加之和呢？当然，这是我们希望看到的。

从图上可以清晰地看出，当 num=100000 时，内容处理的开销最大，负载均衡系统的整体吞吐率几乎等于两台后端服务器的吞吐率之和，随后当 num 逐渐减少，整体吞吐率开始渐渐地落后于两台后端服务器的吞吐率之和，当 num=1000 的时候，整体吞吐率甚至还不如任意一个后端服务器的吞吐率高，如图 12-10 所示的对比图更加直观地反映了整体吞吐率的变化趋势。

这里我们虽然只用了两台后端服务器，但是已经充分暴露了调度器的“接口人瓶颈”，这种瓶颈效应随着后端服务器内容处理时间的减少而逐渐明显，这不难解释，反向代理服务器进行转发操作本身是需要一定开销的，比如创建线程、与后端服务器建立 TCP 连接、接收后端服务器返回的处理结果、分析 HTTP 头信息、用户空间和内核空间的频繁切换等，通常这部分时间并不长，但是当后端服务器处理请求的时间非常短时，转发的开销就显得

尤为突出。

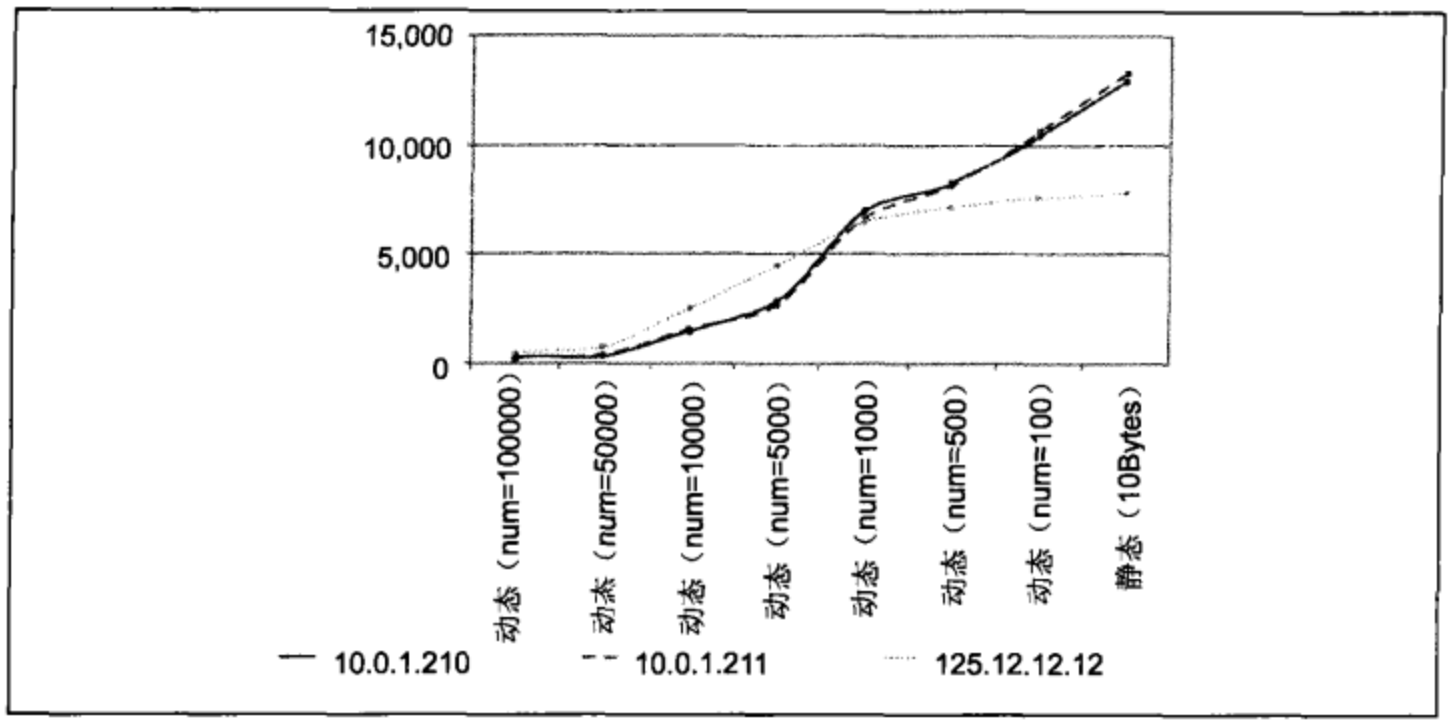


图 12-10　后端服务器和反向代理服务器分别对于不同内容的压力测试结果曲线对比图

另一方面，我们前面也提到了调度器本身的并发处理能力，这也使得当反向代理服务器的吞吐率逐渐接近极限时，无论添加多少后端服务器都将无济于事，因为调度器已经忙不过来了。

所以，你已经意识到，工作在 HTTP 层面的反向代理服务器扩展能力的制约，不仅来自于自身的对外服务能力，也归咎于其转发开销是否上升为主要时间。

为此，我们再来如图 12-9 所示的对比图，从上到下算是两个极端，从图上可以看出，最适合通过反向代理服务器来实现负载均衡的，正是位于图表上方的几种内容，它们就像从事劳动密集型工作一样，人多力量大；而位于图表底部的几种内容，反向代理服务器逐渐变得吃力，尤其是最后的静态内容，整体不但没有提升吞吐率，还浪费了多台服务器资源，对于这种情况，更适合使用前面介绍的基于 DNS 的负载均衡方式。

健康探测

在前面介绍基于 DNS 的负载均衡时，我们曾经提到用于了解实际服务器状态的监控系统，通过它，我们可以第一时间发现存在故障的服务器，然后快速动态更新 DNS 服务器。

当然，这里提到的监控系统并不是 DNS 服务器的组成部分，你可以选择一些开源项目或者商业产品，也可以根据需要自己开发，但是无论如何，你都需要付出不少的人力或者金钱。

幸运的是，一些反向代理服务器软件希望出人头地，成为更加出色的负载均衡调度器，它

们将监控后端服务器的职责视为自己的神圣使命，得益于反向代理服务器的工作机制，它们可以轻松有效地监视后端服务器的任何举动，而你只需要简单配置即可。

通常来说，我们希望能够监控后端服务器的很多方面，比如系统负载、响应时间、是否可用、TCP 连接数、流量等，它们都是负载均衡调度策略需要考虑的因素。在这里，我们使用 Varnish 作为调度器，来监控后端服务器的可用性。

你一定还记得 Varnish 吧，前面我们介绍反向代理缓存的时候曾经提到过它，没错，它同时也支持负载均衡，我们来修改 Varnish 的配置文件，增加后端服务器和负载均衡策略等配置，如下所示：

```
backend b1 {
  .host = "10.0.1.201";
  .port = "80";
  .probe = {
        .url = "/probe.htm";
        .interval = 5s;
        .timeout = 1s;
        .window = 5;
        .threshold = 3;
  }
}
backend b2 {
  .host = "10.0.1.202";
  .port = "80";
  .probe = {
        .url = "/probe.htm";
        .interval = 5s;
        .timeout = 1s;
        .window = 5;
        .threshold = 3;
  }
}
director lb round-robin {
    {
        .backend = b1;
    }
    {
        .backend = b2;
    }
}
```

可以看到，我们通过 backend 关键字定义了两个后端服务器，并且用 director 关键字定义了一个名为 lb 的负载均衡调度器，同时采用了 RR 调度策略。这里需要说明一下，Varnish 目前对于 RR 调度似乎不支持分配权重的设置，所以这里我们选用了两台承载能力基本相同的后端服务器。

在 backend 的定义部分中，我们看到了.probe 设置，这代表了 Varnish 的探测器，根据这些配置，调度器将会每隔 5 秒钟请求后端服务器的/probe.htm，只有当 HTTP 响应头代码

为 200 时，调度器才认为该后端服务器是可用的。

作为后端服务器上的被探测内容，这里我们使用了一个静态文件 probe.htm，你也可以使用 PHP 等动态程序，甚至在程序中包含数据库查询等操作，这样可以更加贴近真实场景，反应实际状态，但是有一点需要注意，那就是当你要报告一个不可用故障时，只需要返回一个不是 200 的状态码即可。

同时，别忘了还要在 Varnish 的 vcl_recv 回调过程中调用刚才定义的调度器，如下所示：

```
sub vcl_recv {
    if (req.request != "GET" &&
      req.request != "HEAD" &&
      req.request != "PUT" &&
      req.request != "POST" &&
      req.request != "TRACE" &&
      req.request != "OPTIONS" &&
      req.request != "DELETE") {
        return (pipe);
    }
    if (req.request != "GET" && req.request != "HEAD") {
        return (pass);
    }
    if (req.http.Authorization || req.http.Cookie) {
        return (pass);
    }
    set req.backend = lb;
    return (pass);
}
```

以上的粗体部分正是我们添加的调用语句，你一定还记得 vcl_recv 这个过程的触发条件，它是在反向代理服务器接收到用户的 HTTP 请求后被调用，所以这时候需要通过调度器来选择适当的后端服务器并转发请求。同时，最后一行的 return (pass)也是经过了我们的修改，原来的 return (lookup)意味着反向代理缓存将可能发挥作用，而这里我们并不需要反向代理缓存。

在验证探测器之前，我们先对两台后端服务器分别进行压力测试，它们的吞吐率表现如表 12-5 所示。

表 12-5　两台后端服务器的独立吞吐率

后端服务器 IP 地址	吞吐率（reqs/s）
10.0.1.201	596.29
10.0.1.202	580.41

现在我们使用 Varnish 作为调度器，它会将我们的请求轮流转发给两台后端服务器。但是，我们暂时不开启 Varnish 的探测器，来看看会发生什么结果。

接下来，关键的时刻到了，我们将其中一台后端服务器的 Web 服务关闭，这导致它将无法处理任何 HTTP 请求。然后，我们对调度器进行压力测试，结果如下所示：

```
Server Software:          Apache/2.2.11
Server Hostname:          10.0.1.50
Server Port:              8010
Document Path:            /test.php?num=10000
Document Length:          206 bytes
Concurrency Level:        100
Time taken for tests:     0.908 seconds
Complete requests:        1000
Failed requests:          500
   (Connect: 0, Receive: 0, Length: 500, Exceptions: 0)
Write errors:             0
Non-2xx responses:        500
Total transferred:        343500 bytes
HTML transferred:         108500 bytes
Requests per second:      1101.85 [#/sec] (mean)
Time per request:         90.756 [ms] (mean)
Time per request:         0.908 [ms] (mean, across all concurrent requests)
Transfer rate:            369.62 [Kbytes/sec] received
```

很糟糕，有一半数量的请求都失败了，虽然整体吞吐率基本接近于两个后端服务器的独立吞吐率之和，但是这又有什么意义呢？

现在我们开启刚才配置好的探测器，重新启动 Varnish，为了让探测器工作，我们等待了 5 秒钟，再次进行压力测试，结果如下所示：

```
Server Software:          lighttpd/1.4.20
Server Hostname:          10.0.1.50
Server Port:              8010
Document Path:            /test.php?num=10000
Document Length:          11 bytes
Concurrency Level:        100
Time taken for tests:     1.825 seconds
Complete requests:        1000
Failed requests:          0
Write errors:             0
Total transferred:        229000 bytes
HTML transferred:         11000 bytes
Requests per second:      547.99 [#/sec] (mean)
Time per request:         182.484 [ms] (mean)
Time per request:         1.825 [ms] (mean, across all concurrent requests)
Transfer rate:            122.55 [Kbytes/sec] received
```

可以看到，调度器已经放弃了关闭 Web 服务的那台后端服务器，即便整体吞吐率大幅度下降，但至少不会给用户返回一个错误页面。

在实际应用中，为了提高整个负载均衡系统的可用性而不影响性能，我们可以部署一定数量的备用后端服务器，这样即便是一些后端服务器出现故障后被调度器放弃，备用后端服务器也可以接替它们的工作，保证整体的性能。

粘滞会话

负载均衡调度器最大程度地让用户不必关心后端服务器，我们知道，当采用 RR 调度策略时，即便是同一用户对同一内容的多次请求，也可能被转发到了不同的后端服务器，这听起来似乎没什么大碍，但有时候，或许会带来一些问题。

- 当某台后端服务器启用了 Session 来本地化保存用户的一些数据后，下次用户的请求如果转发给了其他后端服务器，将导致之前的 Session 数据无法访问；
- 后端服务器实现了一定的动态内容缓存，而毫无规律的转发使得这些缓存的利用率下降。

如何解决这些问题呢？从表面上看，我们需要做的就是调整调度策略，让用户在一次会话周期内的所有请求始终转发到一台特定的后端服务器上，这种机制也称为粘滞会话（Sticky Sessions），要实现它的关键在于如何设计持续性调度算法。

既然要让调度器可以识别用户，那么将用户的 IP 地址作为识别标志最为合适，一些反向代理服务器对此都有支持，比如 Nginx 和 HAProxy，它们可以将用户的 IP 地址进行 Hash 计算并散列到不同的后端服务器上。

对于 Nginx，只需要在 upstream 中声明 ip_hash 即可，如下所示：

```
upstream backend {
    ip_hash;
    server 10.0.1.200:80;
    server 10.0.1.201:80;
}
```

而对于 HAProxy 的配置也非常简单，你需要在 balance 关键字后面添加 source 策略名称，同时要使用 TCP 模式，如下所示：

```
listen  proxy_1 10.0.1.50:8003
        mode tcp
        option httplog
        option dontlognull
        balance source
        stats uri /hastat
        server backend_1 10.0.1.200:80
        server backend_2 10.0.1.201:80
```

除此之外，你还可以利用 Cookies 机制来设计持久性算法，比如调度器将某个后端服务器的编号追加到写给用户的 Cookies 中，这样调度器便可以在该用户随后的请求中知道应该转发给哪台后端服务器。这样做可以更加细粒度地追踪到每一个用户，试想一下，当有很多用户隐藏在一个公开 IP 地址后面时，利用 Cookies 的持久性算法将显得更加有效。

另一方面，回到我们刚才提到的第二个问题，我们希望将对同一个 URL 的请求始终转发

到同一台特定的后端服务器，以充分利用后端服务器针对该 URL 进行的本地化缓存，要实现这一点，HAProxy 也提供了支持，我们使用 uri 策略名称配置如下：

```
listen  proxy_1 10.0.1.50:8003
        mode http
        option httplog
        option dontlognull
        balance uri
        stats uri /hastat
        server backend_1 10.0.1.200:80
        server backend_2 10.0.1.201:80
```

这使得作为调度器的 HAProxy 将对请求的 URL 进行 Hash 计算，然后散列到多台后端服务器上。

好，我们已经实现了粘滞会话，但是如此一来，粘滞会话可能或多或少地破坏了均衡策略，至少像权重分配这样的动态策略已经无法工作，我们对此不能视而不见，否则前面的努力即将付诸东流。

当然，问题的关键在于，我们究竟是否要通过实现粘滞会话来迁就系统的特殊需要呢？在权衡代价之后你认为是否值得呢？最为关键的问题是前面提到的两个问题是否能从根本上避免呢？如果可以，这很值得去考虑。

事实上，在后端服务器上保存 Session 数据和本地化缓存，的确是一件不明智的事情，它使得后端服务器显得过于个性化，以至于和整个系统格格不入，如果允许的话，我们应该尽量避免这样的设计，比如采用分布式 Session 或者分布式缓存等，让后端服务器的应用尽量与本地无关，也可更好地适应环境。

12.5 IP 负载均衡

我们已经充分了解了反向代理服务器作为负载均衡调度器的工作机制，其本身的开销已经严重制约了这种框架的可扩展性，从而也限制了它的性能极限。

那么，能否在 HTTP 层面以下实现负载均衡呢？答案是肯定的，还记得本书开头那个星际铁路系统的故事吗？回忆一下网络分层模型，事实上，在数据链路层（第二层）、网络层（第三层）以及传输层（四层）都可以实现不同机制的负载均衡，但有所不同的是，这些负载均衡调度器的工作必须由 Linux 内核来完成，因为我们希望网络数据包在从内核缓冲区进入进程用户地址空间之前，尽早地被转发到其他实际服务器上，没错，Linux 内核当然可以办得到，随后我们会介绍位于内核的 Netfilter 和 IPVS，而用户空间的应用程序对此却束手无策。

另一方面，也正是因为可以将调度器工作在应用层以下，这些负载均衡系统可以支持更多

的网络服务协议，比如 FTP、SMTP、DNS，以及流媒体和 VoIP 等应用。

这里我们先来介绍基于 NAT 技术的负载均衡，因为它可以工作在传输层，对数据包中的 IP 地址和端口信息进行修改，所以也称为四层负载均衡。

DNAT

前面曾经提到过网络地址转换（Network Address Translation，NAT），它可以让用户身处内部网络却与互联网建立通信，而在这里，为了突出应用场景的差异，我想创造一个更加适合的名称，那就是“反向 NAT”，有点类似于反向代理的命名，是的，我们将实际服务器放置在内部网络，而作为网关的 NAT 服务器将来自用户端的数据包转发给内部网络的实际服务器，这为进一步实现负载均衡提供了可能。

这里说的“反向 NAT”，其实就是 DNAT，不同于 SNAT 的是，它需要修改的是数据包的目标地址和端口，这种技术在很多普通的家用宽带路由器上都有支持，如果你曾经配置过任意一款宽带路由器，或许会看到 NAT 设置，或者也称为端口映射设置，因为我们知道通过 NAT 可以修改数据包的目的地端口，很好地隐藏内网服务器的实际端口，提高安全性。

如图 12-11 所示，我们利用家用宽带路由器进行 NAT 设置，再加上前面提到的动态 DNS，完全可以在家里搭建一个公开的小型 Web 站点。

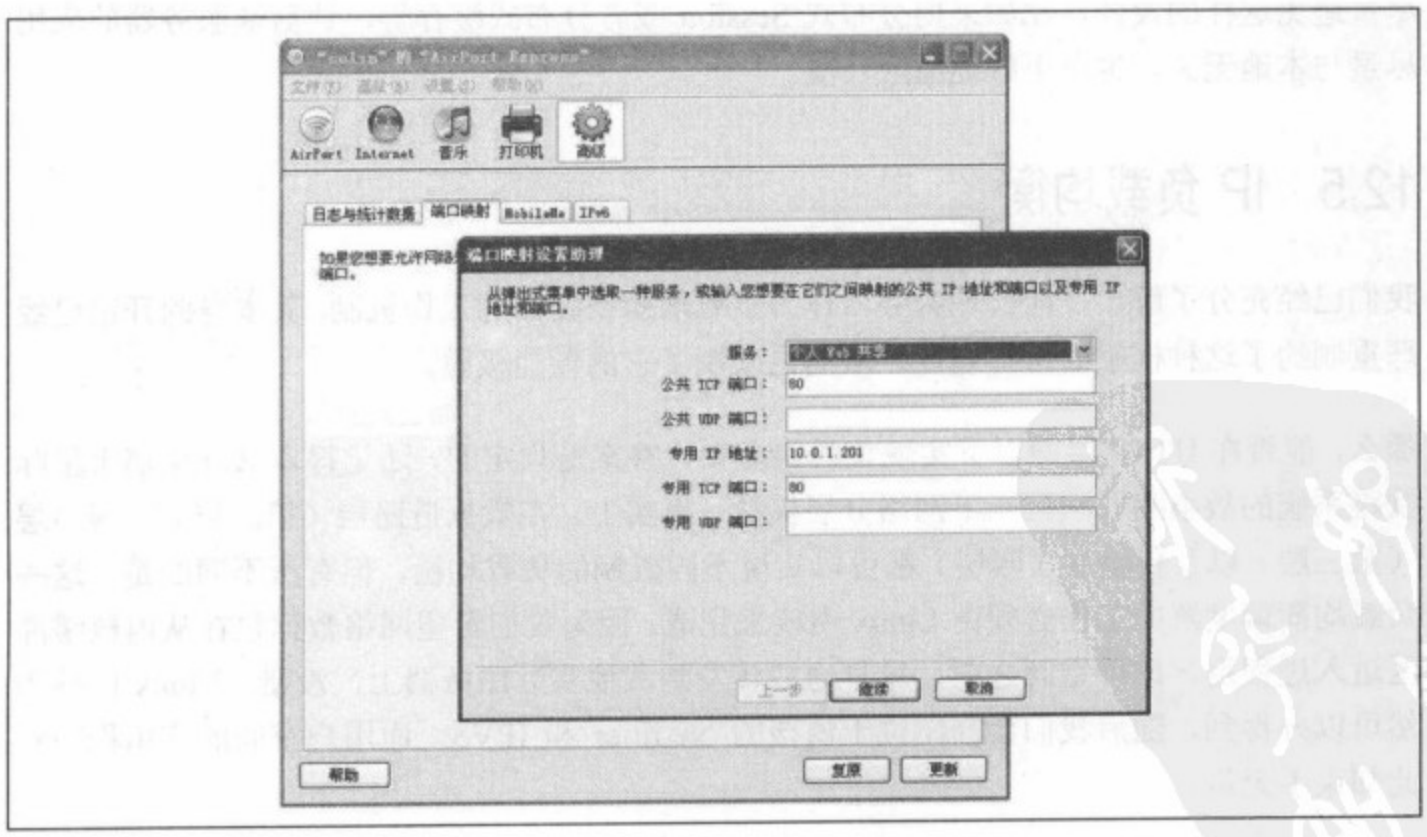


图 12-11　家用宽带路由器的端口映射设置

NAT 服务器做了什么

那么，基于 NAT 的负载均衡系统是如何工作的呢？这里我们举个简单的例子，你将会了解 NAT 服务器在整个过程中做了什么。

如图 12-12 所示，NAT 服务器拥有两块网卡，分别连接外部网络和内部网络，IP 地址分别为 125.12.12.12 和 10.0.1.50。与 NAT 服务器同在一个内部网络的是两台实际服务器，IP 地址分别为 10.0.1.210 和 10.0.1.211，它们的默认网关都是 10.0.1.50，并且都在 8000 端口上运行着 Web 服务。另外，我们假想用户端的 IP 地址为 202.20.20.20。

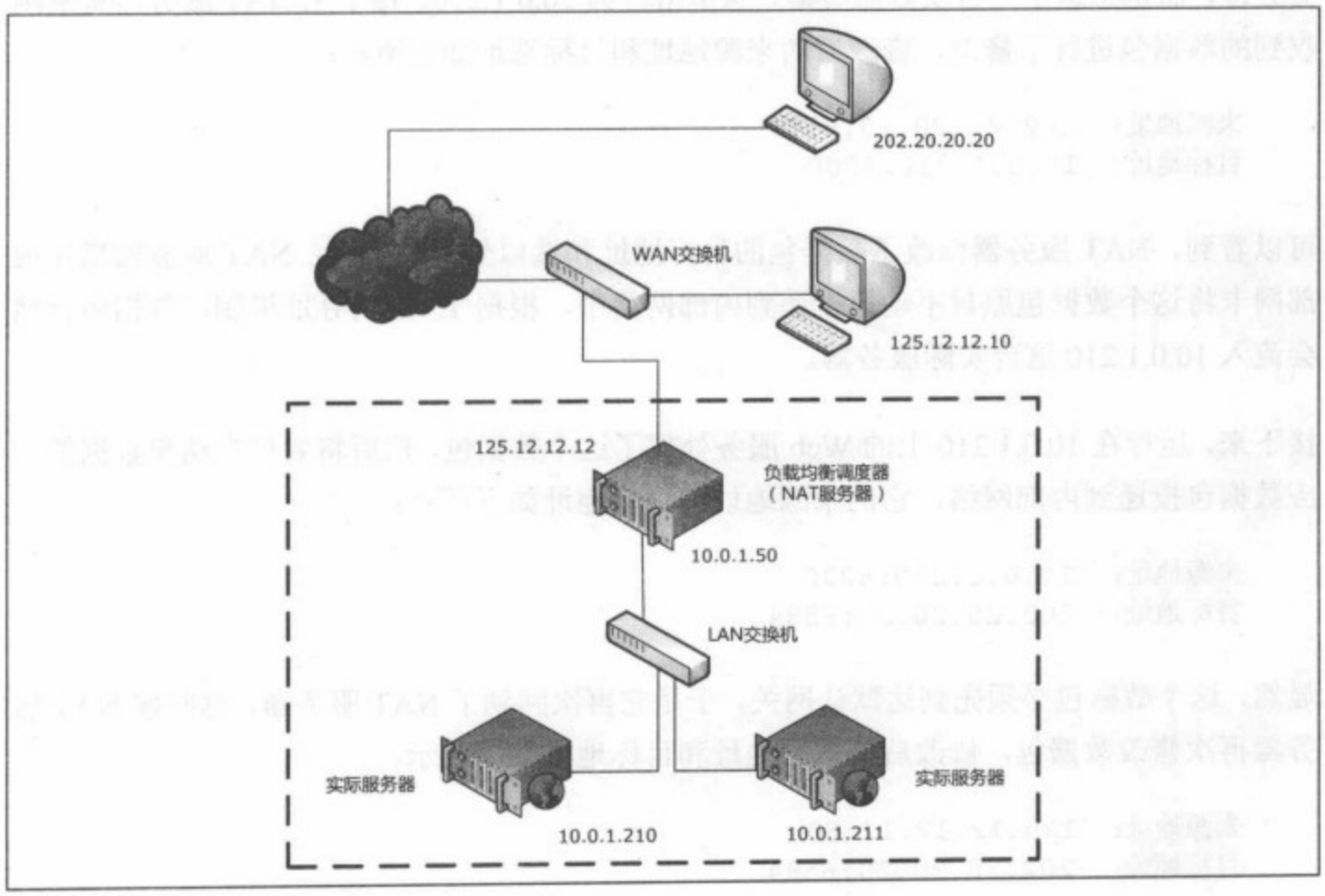


图 12-12　基于 NAT 的负载均衡系统网络结构图

表 12-6　基于 NAT 的负载均衡系统网络结构图中的服务器说明

服务器说明	内部网络 IP	外部网络 IP	默认网关
NAT 服务器	10.0.1.50	125.12.12.12	125.12.12.1
实际服务器	10.0.1.200	-	10.0.1.50
实际服务器	10.0.1.201	-	10.0.1.50

现在我们以一个数据包的真实旅程为例，来跟踪它是如何从用户端到实际服务器，又如何从实际服务器返回用户端，在这一系列过程中，数据包的命运发生了多次改变。

首先，用户端通过 DNS 服务器得知站点的 IP 地址为 125.12.12.12，当然，如果站点不使用域名的话，这一步也可能省略。

接下来，用户端向 125.12.12.12 发送了 IP 数据包，我们将目光放在其中的一个数据包上，它的来源地址和目标地址如下所示：

```
来源地址：  202.20.20.20:6584
目标地址：  125.12.12.12:80
```

当数据包到达 125.12.12.12 的内核缓冲区后，NAT 服务器并没有把它交给用户空间的进程去处理，而是挑选了一台实际服务器，这里恰好为 10.0.1.210，接下来 NAT 服务器将刚刚收到的数据包进行了修改，修改后的来源地址和目标地址如下所示：

```
来源地址：  202.20.20.20:6584
目标地址：  10.0.1.210:8000
```

可以看到，NAT 服务器修改了数据包的目标地址和端口号，紧接着，NAT 服务器指定内部网卡将这个数据包原封不动地投递到内部网络中，根据 IP 层的寻址机制，数据包自然会流入 10.0.1.210 这台实际服务器。

接下来，运行在 10.0.1.210 上的 Web 服务处理了这个数据包，然后将要包含结果数据的响应数据包投递到内部网络，它的来源地址和目标地址如下所示：

```
来源地址：  10.0.1.210:8000
目标地址：  202.20.20.20:6584
```

显然，这个数据包必须先到达默认网关，于是它再次回到了 NAT 服务器，这时候 NAT 服务器再次修改数据包，修改后的来源地址和目标地址如下所示：

```
来源地址：  125.12.12.12:80
目标地址：  202.20.20.20:6584
```

最后，数据包终于回到了用户端。

在整个过程中，NAT 服务器的动作可谓天衣无缝，它欺骗了实际服务器，而在实际服务器上运行的进程总是天真地以为数据包是用户端直接发给自己的，不过，这也许是善意的欺骗。

同时，NAT 服务器也扮演了负载均衡调度器的角色，那么，在讨论调度策略之前，你也许更关心的问题是如何实现 NAT 服务器。

Netfilter/iptables

首先，我们必须得知道 Linux 如何修改 IP 数据包，可以肯定的是，Linux 内核已经具备这

样的能力，从 Linux 2.4 内核开始，其内置的 Netfilter 模块便肩负起这样的使命，它在内核中维护着一些数据包过滤表，这些表包含了用于控制数据包过滤的规则。

我们知道，当网络数据包到达服务器的网卡并且进入某个进程的地址空间之前，先要通过内核缓冲区，这时候内核中的 Netfilter 便对数据包有着绝对控制权，它可以修改数据包，改变路由规则。

既然 Netfilter 工作在内核中，我们看不见摸不着，那么如何让它按照我们的需要来工作呢？Linux 提供了 iptables，它是工作在用户空间的一个命令行工具，我们可以通过它来对 Netfilter 的过滤表进行插入、修改或删除等操作，也就是建立了与 Netfilter 沟通的桥梁，告诉它我们的意图。

那么，我们要做的就是让 Linux 服务器成为连接外部网络和私有网络的路由器，但这不是普通的路由器，我们知道路由器的工作是存储转发，除了修改数据包的 MAC 地址以外，通常它不会对数据包做其他手脚，而我们要实现的路由器恰恰是要对数据包进行必要的修改，包括来源地址和端口，或者目标地址和端口。

总之，Linux 内核改变数据包命运的惊人能力，决定了我们可以构建强大的负载均衡调度器，将请求分散到其他实际服务器上。

用 iptables 来实现调度器

可以用 iptables 来实现负载均衡调度器吗？我们来试试吧。

如果你对 iptables 的使用并不熟悉，可以通过丰富的在线文档进行系统的学习，遗憾的是，我们这里不会介绍 iptables 的详细使用规则，事实上要想在有限的篇幅中把它说清楚并不容易，而且还可能会给你留下阴影。

说到 iptables，最多的应用场景就是防火墙了，我几乎为每台 Linux 服务器都毫不犹豫地进行 iptables 防火墙配置，比如以下这段简单的 iptables 规则：

```
iptables -F INPUT
iptables -A INPUT -i eth0 -p tcp --dport 80 -j ACCEPT
iptables -P INPUT DROP
```

它完成了重要的任务，那就是告诉内核只允许外部网络通过 TCP 与这台服务器的 80 端口建立连接，这项规则可以很好地用在 Web 服务器上。

另外，我们还会用 iptables 来实现本机端口重定向，比如以下的规则：

```
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -j REDIRECT
--to-port 8000
```

它将所有从外部网络进入服务器 80 端口的请求转移到了 8000 端口，这有什么意义呢？当然是隐藏某些服务的实际端口，同时也便于将一个端口快速切换到其他端口的服务上，提高端口管理的灵活性。

关键的时刻到了，我们将用 iptables 来实现 NAT，在此之前，我们需要执行以下的命令行操作：

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

这意味着该服务器将被允许转发数据包，这也为它成为 NAT 服务器提供了可能。

接下来，我们在作为调度器的服务器上执行以下的 iptables 规则：

```
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 8001 -j DNAT
--to-destination 10.0.1.210:8000
```

这条规则几乎完成了所有 DNAT 的实现，它将调度器外网网卡上 8001 端口接收的所有请求转发给 10.0.1.210 这台服务器的 8000 端口，至于转发的具体过程，前面我们已经详细介绍过，在此不再赘述。

同样，我们在调度器上执行以下的 iptables 规则：

```
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 8002 -j DNAT
--to-destination 10.0.1.211:8000
```

不用多说，这条规则你一定明白了。

这两条规则已经进入了 Netfilter 的过滤表，我们可以通过 iptables 命令来查看它们，如下所示：

```
s-director:~ # iptables -nL -t nat
Chain PREROUTING (policy ACCEPT)
Target  prot opt source   destination
DNAT    tcp  --  0.0.0.0/0  0.0.0.0/0 tcp dpt:8001 to:10.0.1.210:8000
DNAT    tcp  --  0.0.0.0/0  0.0.0.0/0 tcp dpt:8002 to:10.0.1.211:8000
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

可以看到，转发规则已经存在，但是还缺少一个环节，这至关重要，我们必须将实际服务器的默认网关设置为 NAT 服务器，也就是说，NAT 服务器必须为实际服务器的网关，否则，数据包被转发后将一去不返。

添加默认网关非常容易，在实际服务器上执行以下命令行操作：

```
route add default gw 10.0.1.50
```

现在，查看路由表，刚才的默认网关已经出现了：

```
s-rs:~ # route
Kernel IP routing table
Destination  Gateway    Genmask         Flags Metric  Ref    Use Iface
10.0.1.0     *          255.255.255.0 U     0         0      0   eth1
125.12.12.0  *          255.255.255.0 U     0         0      0   eth0
link-local   *          255.255.0.0   U     0         0      0   eth0
loopback     *          255.0.0.0     U     0         0      0   lo
default      10.0.1.50  0.0.0.0       UG    0         0      0   eth0
```

现在，我们终于可以通过调度器的8001和8002端口分别将请求转发到两个实际服务器上，可是，回想前面的问题，用iptables来实现负载均衡调度器，看起来有点困难，iptables似乎只能按照我们的规则来干活，没有调度器应该具备的调度能力和调度策略。

接下来，IPVS上场的时刻到了。

IPVS/ipvsadm

熟悉了Netfilter/iptables的机制后，理解IPVS（IP Virtual Server）就一点也不难了，它的工作性质类似于Netfilter模块，也工作在Linux内核中，但是它更专注于实现IP负载均衡。

IPVS不仅可以实现基于NAT的负载均衡，同时还包括后面要介绍的直接路由和IP隧道等负载均衡。令人振奋的是，IPVS模块已经内置到Linux 2.6.x内核中，这意味着使用Linux 2.6.x内核的服务器将无须重新编译内核就可以直接使用它。

要想知道内核中是否已经安装IPVS模块，可以进行以下查看：

```
s-mat:~ # modprobe -l | grep ipvs
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_wrr.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_wlc.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_sh.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_sed.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_rr.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_nq.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_lc.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_lblcr.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_lblc.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_dh.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs.ko
/lib/modules/2.6.16.21-0.8-bigsmp/kernel/net/ipv4/ipvs/ip_vs_ftp.ko
```

这样的结果就意味着IPVS已经安装在内核中。

当然，IPVS也需要管理工具，那就是ipvsadm，它为我们提供了基于命令行的配置界面，你可以通过它快速实现负载均衡系统，这里也称为LVS（Linux Virtual Server，Linux虚拟服务器）或者集群。

你可以从 LVS 的官方站点下载与内核匹配的 ipvsadm 源代码，然后编译它，需要注意的是，在编译过程中，你还需要在/usr/src/linux 中存有内核源代码（Kernel-source）。

接下来，我们将使用 ipvsadm 来实现基于 NAT 的负载均衡系统，也就是 LVS-NAT。

LVS-NAT

当一切都准备好后，你会发现使用 ipvsadm 组建基于 NAT 的负载均衡系统是一件富有乐趣的事情。

现在我们可以暂时完全忘记 iptables，因为 ipvsadm 在这里可以完全取代它。首先不要忘了打开调度器的数据包转发选项，如下所示：

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

然后，你还需要检查实际服务器是否已经将 NAT 服务器作为自己的默认网关，如果不是，赶快添加它，比如：

```
route add default gw 10.0.1.50
```

接下来，就是见证 IPVS 的时刻了，我们执行以下命令行操作：

```
ipvsadm -A -t 125.12.12.12:80 -s rr
ipvsadm -a -t 125.12.12.12:80 -r 10.0.1.210:8000 -m
ipvsadm -a -t 125.12.12.12:80 -r 10.0.1.211:8000 -m
```

第一行规则用来添加一台虚拟服务器，也就是负载均衡调度器，这里的-s rr 是指采用简单轮询的 RR 调度策略。后面两行用来为调度器添加实际服务器，其中-m 表示采用 NAT 方式来转发数据包，这正是我们所希望的。

接下来，我们还可以通过 ipvsadm 来查看所有实际服务器的状态，如下所示：

```
s-director:~ # ipvsadm -L -n
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  125.12.12.12:80 rr
  -> 10.0.1.210:80                Masq    1      0          2
  -> 10.0.1.211:80                Masq    1      0          2
```

大功告成，我们对调度器进行连续的 HTTP 请求，没错，它正是采用 RR 方式将请求均衡地分散到了两台实际服务器上。

关键的问题来了，我们费了很大的力气，终于搭建起基于 NAT 的负载均衡系统，它的性能如何呢？

性能

还记得前面对反向代理负载均衡系统的一系列不同内容的压力测试吗？在使用了LVS-NAT后，我们同样针对这些内容，再次对调度器进行压力测试，同时和之前的数据进行比较，如表12-7所示。

表12-7　后端服务器、反向代理服务器、NAT服务器分别对于不同内容的压力测试结果

内容类型	10.0.1.210	10.0.1.211	反向代理负载均衡	LVS-NAT
静态（10Bytes）	13022.53	13328.71	7778.5	15953.03
动态（num=100）	10529.97	10642.43	7589.22	13850.88
动态（num=500）	8244.66	8177.55	7137.96	13513.79
动态（num=1000）	6950.16	6634.42	6458.53	12941.28
动态（num=5000）	2771.98	2654.07	4454.4	5310.58
动态（num=10000）	1444	1478.96	2468.48	2870.92
动态（num=50000）	322.26	331.79	635.39	646.03
动态（num=100000）	151.9	157.48	296.6	303.68

对于以上的数据，我们仍然绘制了柱状图和曲线图，如图12-13和图12-14所示。通过前面对基于反向代理的负载均衡系统扩展能力的分析，我们知道，当实际服务器的吞吐率达到一定高度时，反向代理服务器的吞吐率将很快达到极限，而从以下对比图中可以看出，在基于NAT的负载均衡系统中，作为调度器的NAT服务器可以将吞吐率继续提升到一个新的高度，几乎是反向代理服务器的两倍以上，这当然归功于在内核中进行请求转发的较低开销。

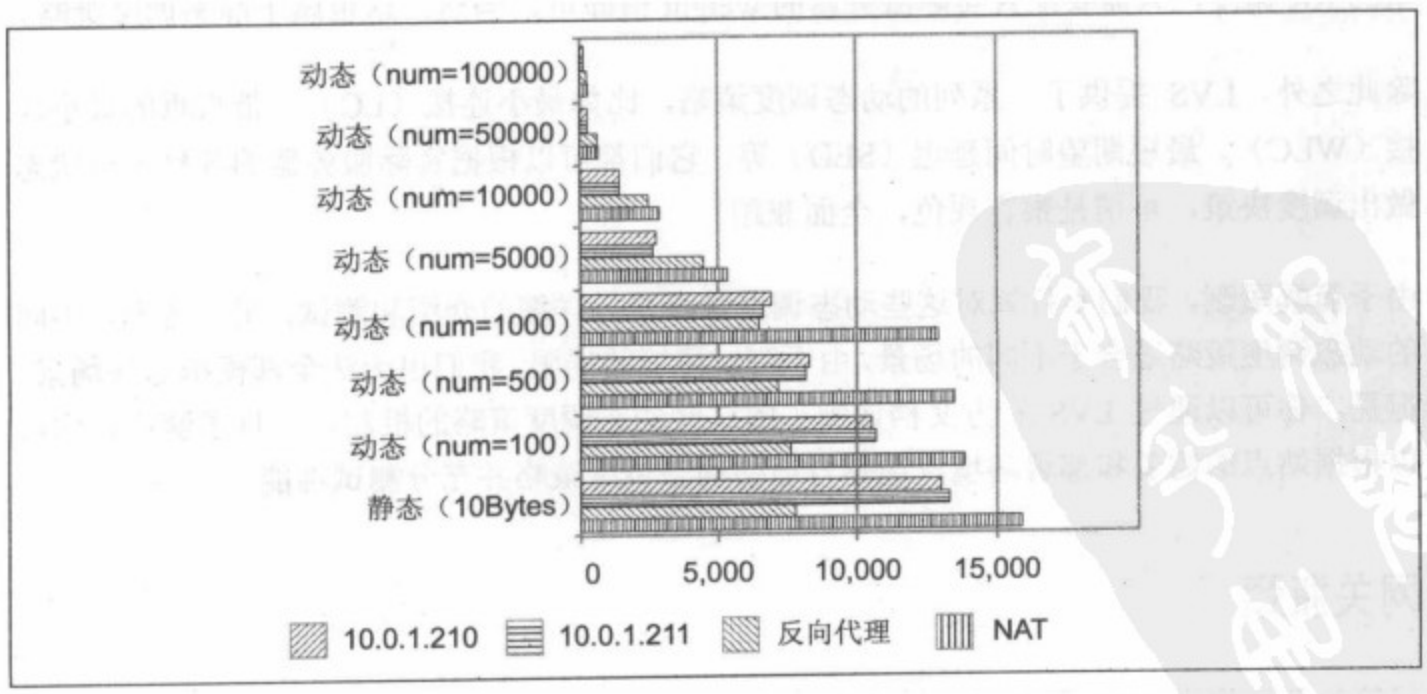


图12-13　反向代理负载均衡和LVS-NAT对比柱状图

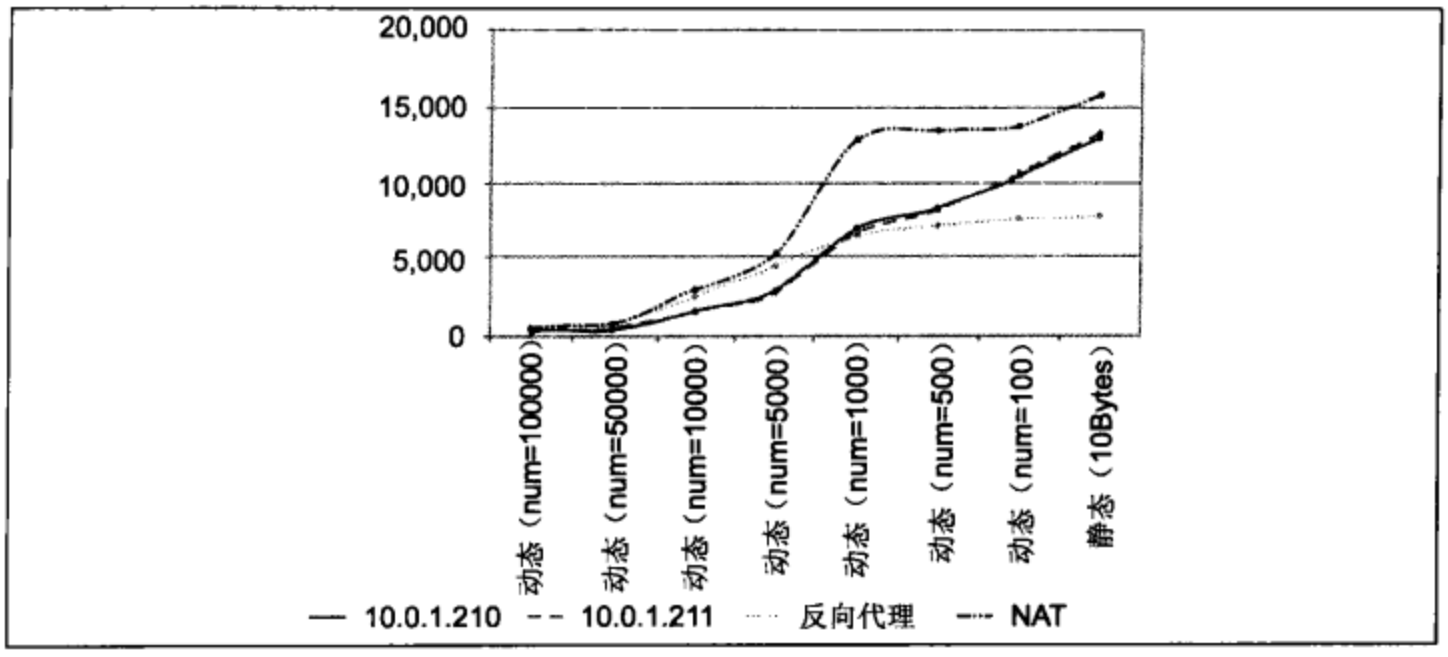


图 12-14　反向代理负载均衡和 LVS-NAT 对比曲线图

另外，从图 12-14 中可以直观地看出，对于 num 大于 5000 的动态内容，也就是实际服务器的吞吐率小于 3000reqs/s 时，不论是基于反向代理，还是基于 NAT，负载均衡的整体吞吐率都差距不大，这意味着对于一些开销较大的内容，使用简单的反向代理来搭建负载均衡系统是非常值得考虑的，至少在初期是一个快速有效的方案，而且它可以非常容易地迁移到 NAT 方式。

动态调度策略

在刚才的 LVS-NAT 中，我们使用了 RR 调度策略，它是一种静态调度策略，当在集群中实际服务器承载能力相当的环境下，它可以很好地实现均衡调度。同时，LVS 也支持带权重的 RR 调度，只需要配置实际服务器的 weight 值即可，当然，这也属于静态调度策略。

除此之外，LVS 提供了一系列的动态调度策略，比如最小连接（LC）、带权重的最小连接（WLC）、最短期望时间延迟（SED）等，它们都可以根据实际服务器的各种实时状态做出调度决策，可谓是察言观色，全面兼顾。

由于篇幅限制，我们不打算对这些动态调度策略进行详细的介绍和测试，另一方面，不同的动态调度策略适合于不同的场景，由于测试环境的局限，我们也无法全都模拟这些场景。但是，你可以通过 LVS 官方文档详细了解这些动态调度策略的机制，一旦了解后，你可以根据站点的需要和部署环境，选择合适的动态调度策略并充分测试性能。

网关瓶颈

尽管如此，作为 NAT 服务器的网关也成为制约集群扩展的瓶颈，我们知道，NAT 服务器

不仅要将用户的请求转发给实际服务器，同时还要将来自实际服务器的响应转发给用户，所以，当实际服务器数量较多，并且响应数据流量较大时，来自多个实际服务器的响应数据包将有可能在 NAT 服务器发生拥挤。

显然，考验 NAT 服务器转发能力的时刻到了，由于转发数据包工作在内核中，我们几乎可以不考虑额外的开销，所以，转发能力主要取决于 NAT 服务器的网络带宽，包括内部网络和外部网络。

举个例子，假如 NAT 服务器通过 100Mbps 的交换机与多台实际服务器组成内部网络，通过前面介绍带宽的章节，我们知道这些实际服务器到 NAT 服务器的带宽为共享 100Mbps，这样一来，尽管实际服务器本身可以很容易达到 100Mbps 的响应流量，比如提供下载服务等，但是 NAT 服务器的 100Mbps 出口带宽成了制约条件，使得无论添加多少台实际服务器，整个集群最多只能提供 100Mbps 的服务。

要解决网关带宽的瓶颈也并不困难，我们可以为 NAT 服务器使用千兆网卡，并且为内部网络使用千兆交换机，图 12-15 展示了理想的带宽配置，其中提及的网卡和交换设备都是工作在全双工模式下的，也就是两个方向的数据传输都具备同样的带宽。

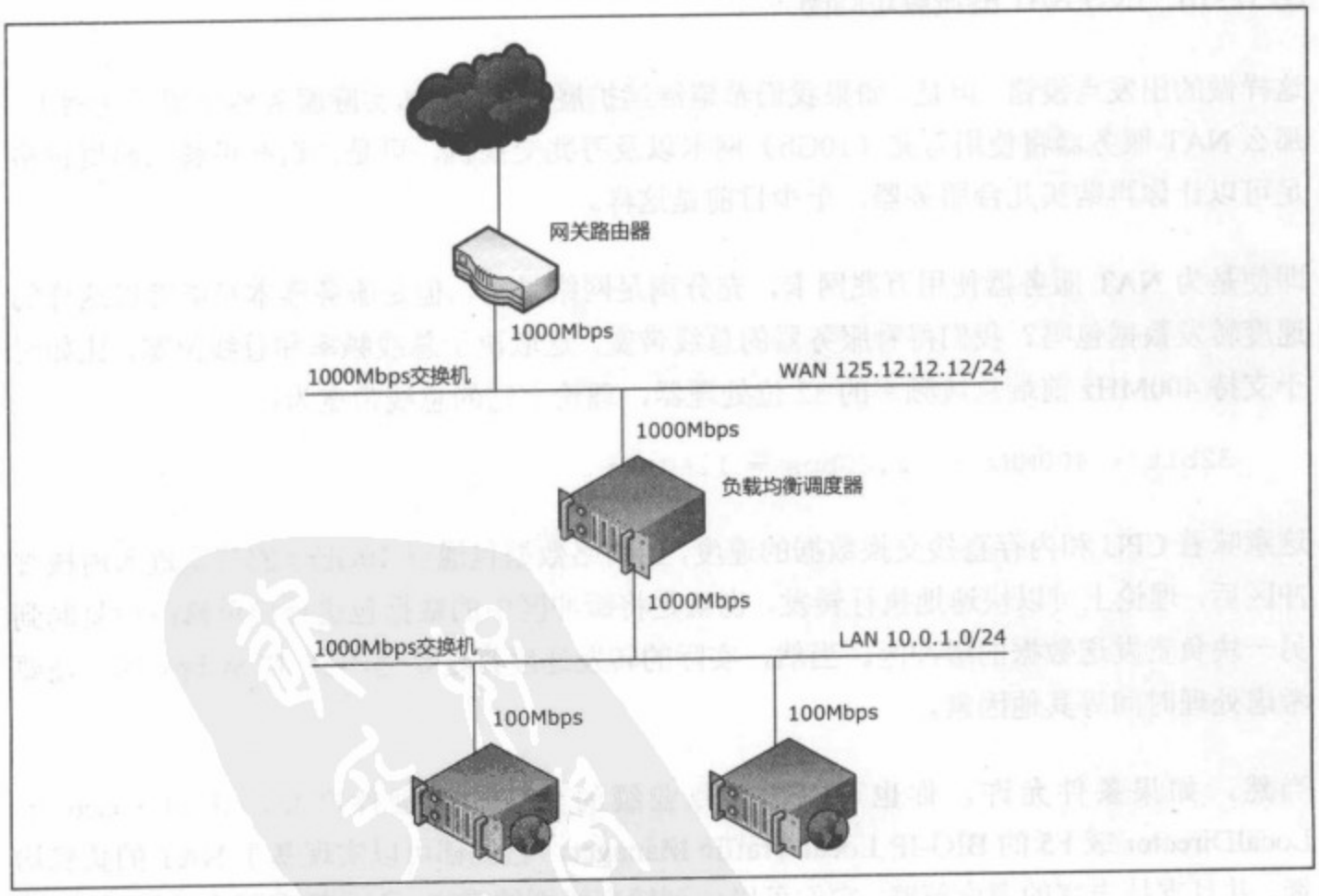


图 12-15　LVS-NAT 的理想带宽配置

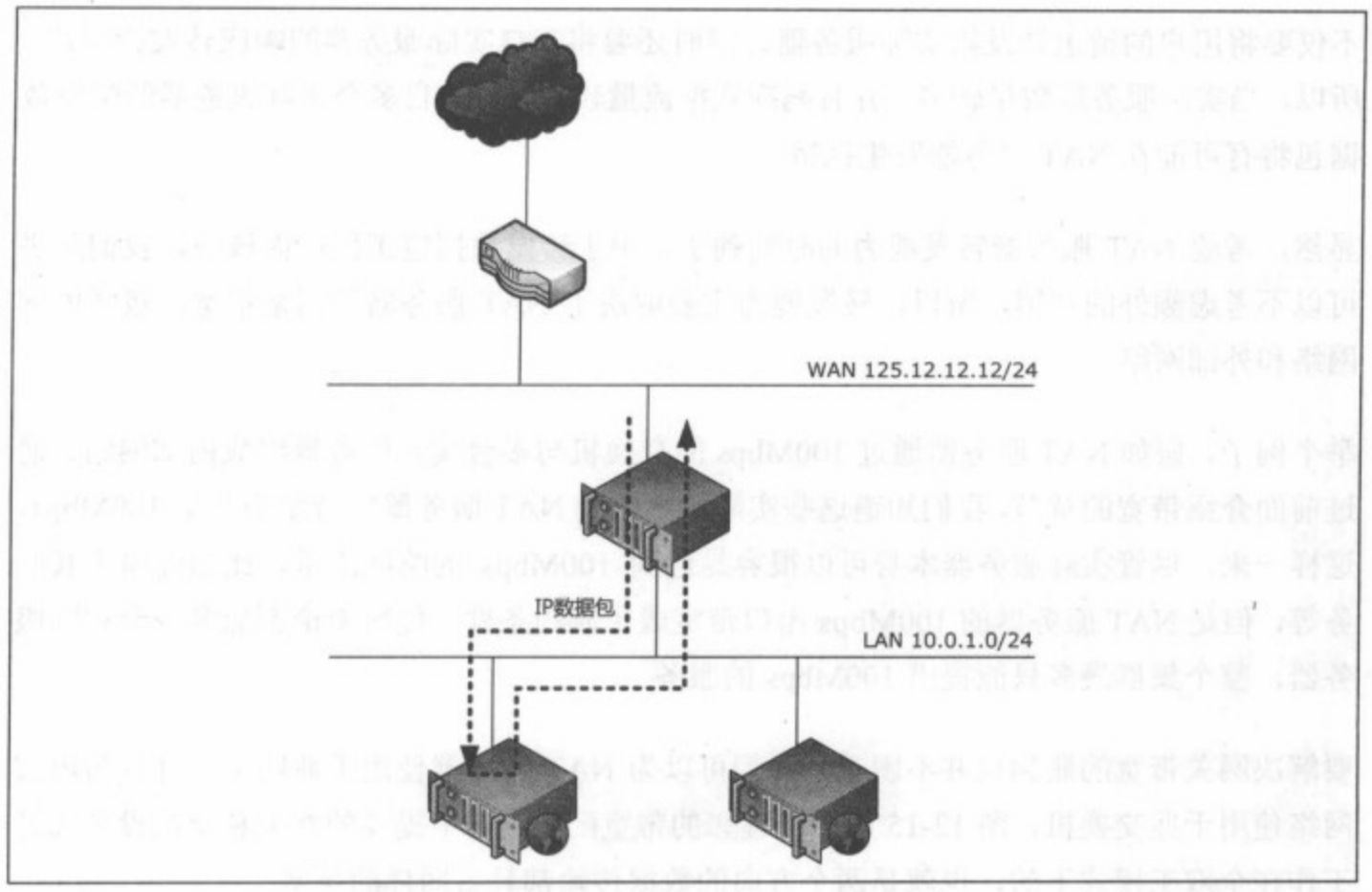


图 12-16　LVS-NAT 的流量走向图

这样做的出发点没错，但是，如果我们希望继续扩展带宽，比如实际服务器使用千兆网卡，那么 NAT 服务器将使用万兆（10Gb）网卡以及万兆交换机，可是，这些设备的昂贵价格足可以让你再购买几台服务器，至少目前是这样。

即便是为 NAT 服务器使用万兆网卡，充分满足网络带宽，但是服务器本身能够以这样的速度转发数据包吗？我们得看服务器的总线带宽，这取决于总线频率和总线位宽，比如对于支持 400MHz 前端总线频率的 32 位处理器，理论上它的总线带宽为：

```
32bit × 400MHz = 12.8Gbps = 1.6GB/s
```

这意味着 CPU 和内存直接交换数据的速度，当网络数据包通过 10Gbps 的带宽进入内核缓冲区后，理论上可以快速地执行转发，也就是将缓冲区中的数据包进行简单修改后复制到另一块负责发送数据的缓冲区。当然，实际的转发速度肯定是达不到 12.8Gbps 的，还要考虑处理时间等其他因素。

当然，如果条件允许，你也可以购买专业级的负载均衡硬件产品，比如 Cisco 的 LocalDirector 或 F5 的 BIG-IP Local Traffic Manager，它们都可以实现基于 NAT 的负载均衡，并且支持丰富的调度策略，它们可以达到较高级别的带宽，而且拥有强大的管理功能，但价格不菲。

我们不打算详细介绍这些硬件产品，如果有兴趣，你可以查阅它们的官方介绍。但可以肯

定的是，它们实现数据转发和调度策略的方式和我们这里的介绍没有太大的差异。

假如你不想花钱去购买千兆交换机或万兆交换机，甚至负载均衡硬件设备，又不想让 NAT 服务器成为制约扩展的瓶颈，怎么办呢？

一个简单有效的办法是，将基于 NAT 的集群和前面的 DNS-RR 混合使用，你可以组建多个条件允许的 NAT 集群，比如 5 个 100Mbps 出口带宽的集群，然后通过 DNS-RR 方式来将用户请求均衡地指向这些集群，同时，你还可以利用 DNS 智能解析实现地域就近访问。

事实上，对于大多数中等规模的站点，拥有 5 个 100Mbps 的 NAT 集群，再加上 DNS-RR，通常足够应付全部的业务。

但是，对于提供下载或视频等服务的大规模站点，100Mbps 的集群带宽就显得微不足道了，其中一台实际服务器甚至就要吞没上百兆的带宽，现在，NAT 服务器的瓶颈出现了，你会选择提高 NAT 服务器的带宽，还是选择其他方案呢？幸运的是，LVS 提供了另一种实现负载均衡的方式，那就是直接路由（Direct Route，DR），接下来我们会详细介绍它。

12.6 直接路由

不同于 NAT 机制，直接路由方式下的负载均衡调度器工作在数据链路层（第二层），简单地说，它通过修改数据包的目标 MAC 地址，将数据包转发到实际服务器上，并且最重要的是，实际服务器的响应数据包将直接发送给用户端，而不经过调度器。

这听起来似乎不可思议，响应数据包可以不经过调度器，这意味着什么呢？

可以肯定的是，实际服务器必须直接接入外部网络，它可以不使用 RFC1918 规定的私有地址，也不能将调度器作为默认网关。

提示：

RFC1918 规定的私有 IP 地址范围是：

```
10.0.0.0        -   10.255.255.255  (10/8 prefix)
172.16.0.0      -   172.31.255.255  (172.16/12 prefix)
192.168.0.0     -   192.168.255.255 (192.168/16 prefix)
```

我们还是先来看看这种方式的实现机制，在此之前，你需要了解一个也许对你来说比较陌生的名词，那就是 IP 别名，它对于直接路由负载均衡的实现至关重要。

使用 IP 别名

我们知道，一个网络接口理所当然地拥有一个 IP 地址，但是除此之外，我们还可以为它

配置更多个 IP 地址，它们称为 IP 别名。这里的网络接口可以是物理网卡（如 eth0、eth1），也可以是虚拟接口（如回环网络接口 lo）。根据规定，一个网络接口最多可以设置 256 个 IP 别名，没错，你可以把一个 C 类网段的所有 IP 地址都设置到一个网卡上，理论上没有任何问题。

你也许已经张大了嘴巴，一个网卡竟然可以设置多个 IP 地址，并且拥有同样的 MAC 地址，没错，它们可以很好地工作。在 Linux 中配置 IP 别名非常简单，比如我们在 125.12.12.12 这台服务器上执行以下命令行操作：

```
ifconfig eth0:0 125.12.12.77
```

这时候，我们通过 ifconfig 命令可以查看到刚才配置的 IP 别名，如下所示：

```
s-director:~ # ifconfig
eth0      Link encap:Ethernet  HWaddr 00:19:B9:DF:B0:52
          inet addr:125.12.12.12  Bcast:125.12.12.255  Mask:255.255. 255.0
          inet6 addr: fe80::219:b9ff:fedf:b052/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6428227 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8242159 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:847758687 (808.4 Mb)  TX bytes:4007763688 (3822.1 Mb)
          Interrupt:6 Memory:f4000000-f4011100
eth0:0    Link encap:Ethernet  HWaddr 00:19:B9:DF:B0:52
          inet addr:125.12.12.77  Bcast:125.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:6 Memory:f4000000-f4011100
```

这样一来，我们从同网段的另一台服务器尝试访问它，如下所示：

```
s-db:~ # ping 125.12.12.77
PING 125.12.12.77 (125.12.12.77) 56(84) bytes of data.
64 bytes from 125.12.12.77: icmp_seq=1 ttl=64 time=6.35 ms
64 bytes from 125.12.12.77: icmp_seq=2 ttl=64 time=0.123 ms
64 bytes from 125.12.12.77: icmp_seq=3 ttl=64 time=0.119 ms
```

可见，IP 别名对外界来说，和普通 IP 地址没什么区别，当网络中有 ARP 广播询问谁拥有 125.12.12.77 这个 IP 地址时，这台服务器将会积极应答。我们在另一台服务器上查看 ARP 表，其中包含以下两条：

```
s-db:~ # arp
Address          HWtype  HWaddress           Flags Mask      Iface
125.12.12.12     ether   00:19:B9:DF:B0:52   C               eth0
125.12.12.77     ether   00:19:B9:DF:B0:52   C               eth0
```

的确，两个 IP 地址对应着相同的 MAC 地址。

将实际服务器接入外部网络

现在你应该已经知道 IP 别名是怎么回事了，那么，它有什么用呢？

刚才我们说到，调度器通过修改数据包的目标 MAC 地址，将它转发给实际服务器，注意，它并没有修改目标 IP 地址，那么一旦数据包到了实际服务器后，发现实际服务器的 IP 地址并不是数据包的目标 IP 时，你也许无法想象会发生什么事，我想这大概就跟梦游的人醒来时的感觉一样。

没错，我们要做的，就是给实际服务器添加和调度器 IP 地址相同的 IP 别名，这样才可以让转发到实际服务器的数据包找到归属感。

在此之前，我们先将前面的网络结构进行一番调整，如图 12-17 所示，其中各台服务器的 IP 地址和网关如表 12-8 所示。

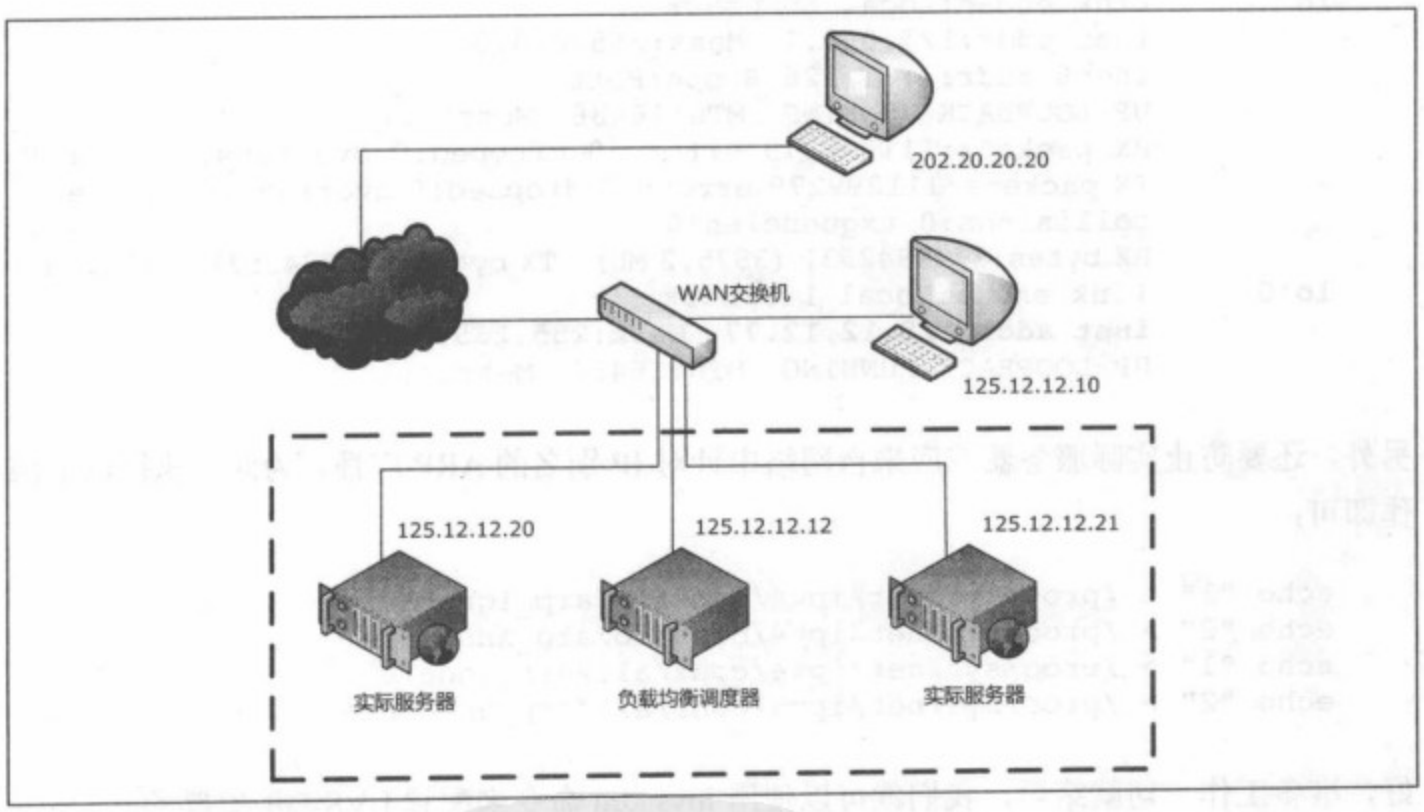


图 12-17　LVS-DR 负载均衡系统网络结构示意图

表 12-8　LVS-DR 负载均衡系统网络结构示意图中的服务器说明

服务器说明	外部网络 IP	默认网关	IP 别名
负载均衡调度器	125.12.12.12	125.12.12.1	125.12.12.77
实际服务器	125.12.12.20	125.12.12.1	125.12.12.77
实际服务器	125.12.12.21	125.12.12.1	125.12.12.77

可以看到，我们为两台实际服务器配置了外部网络 IP 地址，这些 IP 地址通常需要你从 IDC

那里购买。同时，我们为调度器也配置了 IP 别名，这将为调度器的故障转移提供便利，后面我们会介绍调度器的可用性。

这样一来，我们将通过 125.12.12.77 这个 IP 别名来访问调度器，你可以将站点的域名指向这个 IP 别名。

接下来，我们为实际服务器配置与调度器相同的 IP 别名，这里我们将 IP 别名添加到回环接口 lo 上，并且设置路由规则，让实际服务器不要去寻找其他拥有这个 IP 别名的服务器，命令行操作如下：

```
    ifconfig lo:0 125.12.12.77 broadcast 125.12.12.77 netmask 255.255.255.
255 up
    route add -host 125.12.12.77 dev lo:0
```

现在，我们通过 ifconfig 命令来查看网络接口，可以看到以下结果：

```
    lo        Link encap:Local Loopback
              inet addr:127.0.0.1  Mask:255.0.0.0
              inet6 addr: ::1/128 Scope:Host
              UP LOOPBACK RUNNING  MTU:16436  Metric:1
              RX packets:111299279 errors:0 dropped:0 overruns:0 frame:0
              TX packets:111299279 errors:0 dropped:0 overruns:0 carrier:0
              collisions:0 txqueuelen:0
              RX bytes:3748942531 (3575.2 Mb)  TX bytes:3748942531 (3575.2Mb)
    lo:0      Link encap:Local Loopback
              inet addr:125.12.12.77  Mask:255.255.255.255
              UP LOOPBACK RUNNING  MTU:16436  Metric:1
```

另外，还要防止实际服务器响应来自网络中针对 IP 别名的 ARP 广播，为此，执行以下操作即可：

```
    echo "1" > /proc/sys/net/ipv4/conf/lo/arp_ignore
    echo "2" > /proc/sys/net/ipv4/conf/lo/arp_announce
    echo "1" > /proc/sys/net/ipv4/conf/all/arp_ignore
    echo "2" > /proc/sys/net/ipv4/conf/all/arp_announce
```

好，准备工作一切就绪后，我们就可以使用 ipvsadm 命令来配置 LVS-DR 集群了。

LVS-DR

接下来，我们在作为调度器的服务器上通过 ipvsadm 命令进行以下配置：

```
    ipvsadm -A -t 125.12.12.77:80 -s rr
    ipvsadm -a -t 125.12.12.77:80 -r 125.12.12.20:80 -g
    ipvsadm -a -t 125.12.12.77:80 -r 125.12.12.21:80 -g
```

有了配置 LVS-NAT 的经验后，你应该已经非常熟悉上述配置规则，这里我们仍然使用了 RR 调度策略，有所不同的是，在添加实际服务器的时候，我们使用了-g 选项，这意味着

告诉调度器使用直接路由的方式转发数据包。

这时候我们通过 ipvsadm 命令可以看到以下的状态：

```
s-director:~ # ipvsadm -L -n
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  125.12.12.77:80 rr
  -> 125.12.12.20:80              Route   1      0          0
  -> 125.12.12.21:80              Route   1      0          0
```

需要注意的是，在使用直接路由进行转发的情况下，我们无法修改数据包的目的端口，原因很简单，我们知道这种转发机制工作在数据链路层，所以对于上层的端口信息，它无能为力。

也许你已经感觉到，一旦熟悉了 LVS-NAT 的配置方式后，就可以很容易地将其改进为直接路由方式。

与 LVS-NAT 的性能比较

你最关心的时刻到了，我们来对基于直接路由的调度器进行压力测试，同样是针对之前的一系列不同开销的内容。我们照例将测试结果补充到前面的表 12-7 中，但是这次我们去掉了反向代理的那一列，因为我们这里主要关注的是 LVS-DR 和 LVS-NAT 的比较，压力测试结果如表 12-9 所示。

表 12-9　实际服务器、LVS-NAT、LVS-DR 分别对于不同内容的压力测试结果

内容类型	125.12.12.20	125.12.12.21	LVS-NAT	LVS-DR
静态（10Bytes）	13022.53	13328.71	15953.03	15617.39
动态（num=100）	10529.97	10642.43	13850.88	13941.09
动态（num=500）	8244.66	8177.55	13513.79	13813.6
动态（num=1000）	6950.16	6634.42	12941.28	13239.25
动态（num=5000）	2771.98	2654.07	5310.58	5210.18
动态（num=10000）	1444	1478.96	2870.92	2893.08
动态（num=50000）	322.26	331.79	646.03	649.07
动态(num=100000）	151.9	157.48	303.68	303.88

同样，我们将这些数据绘制成对比曲线图，更加直观地观察它们的变化趋势，如图 12-18 所示。

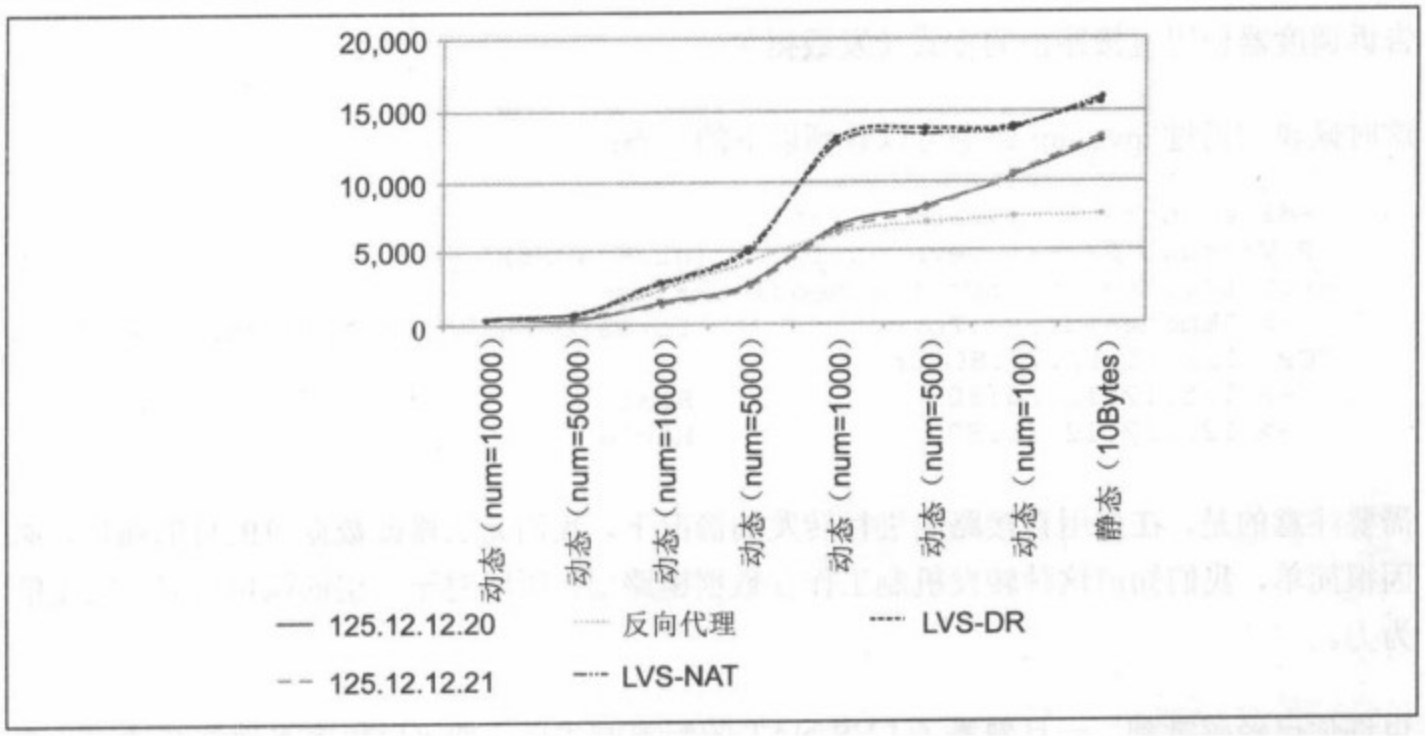


图 12-18　实际服务器、反向代理、LVS-NAT、LVS-DR 分别对于不同内容的压力测试结果对比曲线图

可以看到，LVS-DR 的表现几乎和 LVS-NAT 旗鼓相当，同时，它们都远远超过了基于反向代理的负载均衡系统。那么，相比于 LVS-NAT，LVS-DR 的优势在哪里呢？还记得这种方式的最大特点吗？那就是实际服务器的响应数据包可以不经过调度器而直接发往用户端，如图 12-19 所示，这也是它最大的优势，显然，要让它发挥这种优势，我们希望响应数据包的数量和长度远远大于请求数据包，事实上，大多数 Web 服务正是具备这样的特点，响应和请求并不对称，即便是几十 KB 的网页下载，响应数据包也是请求数据包的很多倍，更不要说大文件的下载。

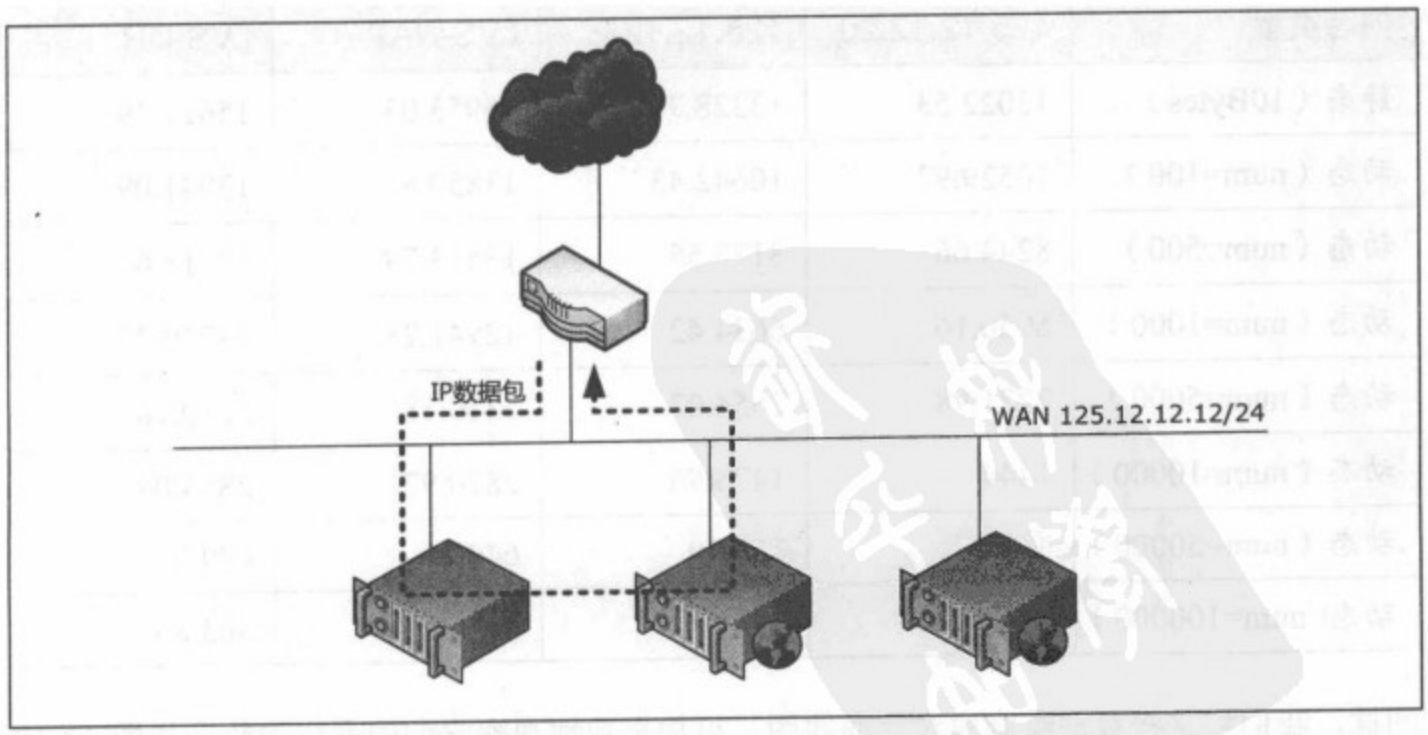


图 12-19　LVS-DR 的流量走向图

这样一来，大量的响应数据包便可以绕过调度器，避免了 LVS-NAT 中调度器的带宽瓶颈，为此，我们来做一个实验，同样基于刚才的网络结构，但是我们在 WAN 交换机上将调度器（10.12.12.12）和两台实际服务器（125.12.12.20、125.12.12.21）的带宽限制为 10Mbps，但不限制 125.12.12.10 这台服务器，它的带宽保证在 100Mbps，因为我们需要通过它来发起测试，事实上也可以假想它就是网关出口。

接下来，我们对实际服务器、LVS-NAT 和 LVS-DR 分别进行了压力测试，这次被测试的内容是一个大小为 22KB 的静态内容，你可以把它当作一个站点的首页 HTML。

测试结果如表 12-10 所示。

表 12-10　针对 22KB 静态内容的压力测试结果对比

被测试服务器	吞吐率（reqs/s）	数据流量（KB/s）	带宽使用量（Kbps）
实际服务器 1	51.75	1134.44	9075.52
实际服务器 2	51.76	1134.71	9077.68
LVS-NAT	51.69	1133.18	9065.44
LVS-DR	102.30	2242.51	17940.08

可以看到，对于前两台实际服务器，吞吐率基本相当，但是我们关心的是数据流量，因为这时候显然带宽成为制约吞吐率的瓶颈，从测试结果上看，带宽使用量接近 10Mbps，但实际上不可能达到理论上的 10Mbps，因为还存在交换机和服务器用户进程对数据包的处理时间。

第三行的 LVS-NAT 意味着调度器通过 NAT 方式将请求分散到前两台实际服务器，这里我们对于连接调度器和两台实际服务器的 LAN 交换机并没有限速，仍然是 100Mbps 的全双工模式，但是我们知道，调度器在 WAN 交换机上的出口带宽已经被限制为了 10Mbps。从测试结果上看，LVS-NAT 并没有提高整体吞吐率，它的瓶颈仍然是调度器的带宽。

最后的 LVS-DR 让我们眼前一亮，数据流量几乎翻了一番，吞吐率自然也随之翻倍，现在你知道 LVS-DR 的优势所在了吧。

虽然这个实验中我们只是将带宽限制到了 10Mbps，但是对于 100Mbps 甚至更高的带宽，其本质都是一样的，我们假设一个理想的带宽配置，如图 12-20 所示。

可以看到，即使调度器只有 100Mbps 的带宽，整个集群也可以通过增加大量的实际服务器来达到 1Gbps 的流量，也就是 WAN 交换机的出口带宽，这充分体现了 LVS-DR 的强大扩展能力。

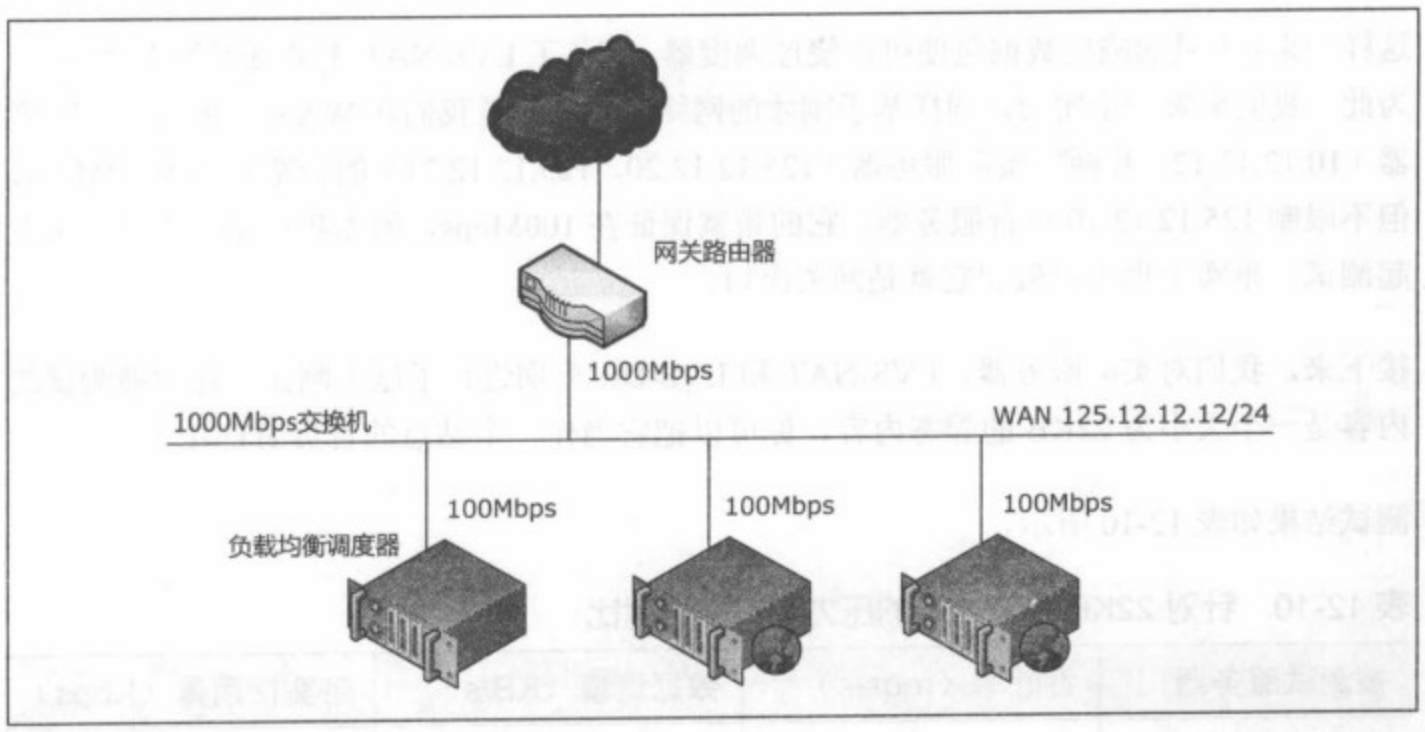


图 12-20　LVS-DR 的理想带宽配置

由于测试环境的局限，这里我们无法测试 LVS-DR 究竟可以扩展多少台实际服务器，但可以肯定的是，这时候的瓶颈更多地转移到了 WAN 的出口带宽，相比之下，其他在单机上的问题将显得微不足道，因为你可以增加更多的实际服务器来分散诸如 CPU 和磁盘 I/O 等开销。

另一个关键的因素在于，响应数据包和请求数据包的比例，因为从调度器发出请求数据包的速度也是有上限的，是否能够让这些有限的请求数据包引发最大程度的响应风暴，也影响到整个系统的扩展能力。所以，越是响应数据包远远超过请求数据包的服务（如视频），就越应该降低调度器转移请求的开销，也就越能够提高整体扩展能力，最终也就越依赖于 WAN 出口带宽。来自 LVS 官方站点的早期测试结果也告诉我们，LVS-DR 可以容纳 100 台以上的实际服务器，我想对于某些特定的场景，这样的表现没有任何问题。

转型到 DNS-RR

除了性能和扩展性的优势之外，LVS-DR 也非常便于管理，这得益于所有的实际服务器都直接接入 WAN，所以你完全可以直接通过安全 SSH 来登录它们进行各种操作。

而对于 LVS-NAT，你必须登录调度器才可以访问位于 LAN 的实际服务器，这给管理大量的机器带来不便，但这还不算什么，它也许会给你带来一些安全感。关键问题在于，一旦调度器有朝一日出现故障无法登录，那实际服务器便和你完全隔离了。当然，办法也是有的，你可以让其中某台实际服务器也接入 WAN，作为冗余“跳板机”，或者设置备用调度器，通过 Heartbeat 来完成自动故障转移，随后我们会介绍这方面的内容。

当然，闲置一台备用调度器，对于一些普通规模的站点来说，可能并不愿意。幸运的是，对于 LVS-DR，一旦调度器失效，你可以马上将 LVS-DR 切换到 DNS-RR 模式，这几乎只需要增加几条 DNS 记录，将域名解析到多台实际服务器的真实 IP 地址即可。一旦调度器恢复后，你便可以再次修改 DNS 记录，将域名仅指向调度器，切换回 LVS-DR。

总的来说，LVS-DR 非常适合搭建可扩展的负载均衡系统，不论是 Web 服务器还是文件服务器，以及视频服务器，它都拥有出色的表现。但前提是，你必须为实际服务器购买一系列的合法 IP 地址，不过，相比于负载均衡硬件设备，它们还是要便宜得多。

12.7 IP 隧道

与 LVS-DR 的原理非常类似，基于 IP 隧道（IP Tunneling）的负载均衡系统同样可以用 LVS 来实现，也称为 LVS-TUN。与 LVS-DR 不同的是，实际服务器可以和调度器不在同一个 WAN 网段，调度器通过 IP 隧道技术来转发请求到实际服务器，所以实际服务器也必须拥有合法的 IP 地址。

基于 IP 隧道的请求转发机制，简单地说，它是将调度器收到的 IP 数据包封装在一个新的 IP 数据包中，转交给实际服务器，然后实际服务器的响应数据包可以直接到达用户端。

当然，要实现 IP 隧道技术还存在一定的前提条件，那就是所有的服务器都必须支持“IP Tunneling”或者“IP Encapsulation”协议。幸运的是，Linux 对此支持良好，同时，“IP Tunneling”正成为各个操作系统的标准协议，所以为实际服务器使用各种操作系统也将成为可能。

对于 LVS-TUN 的配置和性能测试，我们这里就不做详细介绍了，当你了解并实践了 LVS-DR 后，LVS-TUN 对你来说不会陌生。

另外，基于 IP 隧道的独特方式，我们可以将实际服务器根据需要部署在不同的地域，并且根据就近访问的原则来转移请求，比如一些 CDN 服务便是基于 IP 隧道技术来实现的。

总的来说，LVS-DR 和 LVS-TUN 都适合响应和请求不对称的 Web 服务器，可以非常有效地提高集群的扩展能力，但如何选择它们，更多的不是因为性能和扩展性，而是取决于你的网络部署需要，比如刚才提到的 CDN 服务需要将实际服务器部署在不同的 IDC，从而必须使用 IP 隧道技术。

12.8 考虑可用性

对于一些关键的 Web 应用，可用性至关重要，为了实现高可用性的系统，我们不能容忍任何的单点故障，即便只是偶然。所谓的单点故障，是指系统中一旦某个组件发生故障，

便会导致整个系统的失败，所以这种故障是致命的。

提示：

没有侥幸这回事，最偶然的意外，似乎也都是有必然性的。

——【美】爱因斯坦

在负载均衡系统中，多台实际服务器在分散开销的同时，本身也提高了实际服务器的可用性，一般来说，为了在个别实际服务器发生故障后整个系统能够继续承载同样的负载，我们必须将实际服务器的数量保证在略多于实际情况下的数目，这也有利于避免由于大量突发请求造成的雪崩效应。

而对于调度器，转移请求的机制注定它存在单点故障，为此，我们必须通过其他办法来有效实现故障平滑转移，以保证调度器的高可用性。

Heartbeat 可以很好地解决这个问题，它的诞生正是为了实现高可用性。简单地说，我们可以准备一台备用调度器，通过运行 Heartbeat 对主调度器进行心跳检测，一旦发现主调度器停止心跳，便立即启动故障转移，接管主调度器，这个接管过程包括 IP 别名变更、相关服务的启动等。随后，一旦主调度器恢复后，备用调度器便自动将相关资源转交回主调度器。

为了避免主调度器和备用调度器之间线路的单点故障，大量的事实证明，最好采用多条独立线路进行连接，这样将不依赖需要电源的交换机。你也可以使用串行电缆，通过服务器的 COM 端口进行连接，虽然线路长度有限，但这样还可以带来一点点额外的安全性，即便是主调度器被恶意破坏者登录，它也无法通过串行电缆登录备用调度器。

至于交换机，由于异常流量造成阻塞的情况时有发生，但大多数情况下发生在 IDC 级别的交换机上，所以不需要我们担心。一般来说，IDC 级别的交换机在物理层发生故障的概率较小，即便是发生了，更换备用交换机也并不困难，灾难总是在所难免的。

另外，作为物理层设备的网线，也有可能发生故障，虽然我们很少遇到，但如果希望让线路不存在单点故障，可以使用 Linux Bonding 技术来将多条线路绑定在一台服务器的多个网卡上，对流量进行 RR 负载均衡，或者将一条线路设置成为备用模式。Bonding 在 Linux 内核 2.4.12 以后已经被默认支持，你可以通过以下方法查看内核是否支持 Bonding，否则便需要将它编译到内核。

```
    s-colin:~ # modprobe -l | grep bonding
    /lib/modules/2.6.16.21-0.8-bigsmp/kernel/drivers/net/bonding/bondin
g.ko
```

值得一提的是，通过 Bonding 实现流量负载均衡，还可以帮助我们实现带宽聚合，比如我们可以将 6 根 100Mbps 独享的线路绑定到 bond0 虚拟网卡上，实现 600Mbps 的出口带宽，

我们对下载服务器采用了这种策略，可以看到以下结果：

```
s-colin:~ # cat /proc/net/bonding/bond0
Ethernet Channel Bonding Driver: v3.0.1 (January 9, 2006)
Bonding Mode: load balancing (round-robin)
MII Status: up
MII Polling Interval (ms): 0
Up Delay (ms): 0
Down Delay (ms): 0
Slave Interface: eth0
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:24:8c:33:f4:b8
Slave Interface: eth1
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:24:8c:33:f4:b9
Slave Interface: eth2
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:04:23:89:69:e2
Slave Interface: eth3
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:04:23:89:69:e3
Slave Interface: eth4
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:0e:0c:a2:d9:08
Slave Interface: eth5
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:0e:0c:a2:d9:09
```

第 13 章

共享文件系统

当我们为多台 Web 服务器实现负载均衡后，它们凝聚在一起，工作得非常出色，但是别忘了，在之前的介绍中，我们还存在一个假设，那就是暂且认为所有的实际服务器都拥有同样的程序和文件，彼此没有区别，不论请求被转移给谁，它都可以调用必要的资源（包括程序和文件），从而完成任务。

然而在实际情况中，一旦存在多台实际服务器提供同样的 Web 服务时，一个潜在的问题便开始浮现，如何来保证多台实际服务器拥有一致的程序和文件呢？你也许觉得拥有一致的程序并不难，没错，因为它们的更新通常并不频繁，即便是采用持续构建，每天一次的程序更新已经很了不起了。

那么，文件的一致性呢？这是我们不得不面对的现实问题，比如提供照片分享服务的站点，用户发起的照片下载请求被转移到多台实际服务器上，而这些照片正是由用户不断上传的文件，如何保证上传的照片同时存在于多台实际服务器上呢？

13.1 网络共享

从某种意义上来讲，对于强调针对性能而扩展的 Web 应用，其扩展能力体现在 Web 服务器能够最大程度地将数据视为过眼云烟，它们更希望比特只是匆匆过客，而不在本地留下任何痕迹。一个典型的例子就是，将构成动态内容的所有数据存储在独立的 MySQL 数据库服务器中，Web 服务器对于每次请求都查询 MySQL 数据库，生成 HTML 内容，乐此不疲，显然，这样的 Web 应用在通过增加实际服务器来扩展系统规模的时候显得格外敏捷，整个系统表现出很好的可伸缩性。

的确，Web 服务器可以通过网络来访问 MySQL 服务器，尽管这对大家来说早已不是什么新鲜事了，但是从某个角度来看，MySQL 服务器正是扮演了可以让多台 Web 服务器通过网络来访问的共享存储，而这一形式也适用于文件，这也正是我们这一章要介绍的共享文件系统。

从使用的角度来看，共享文件系统几乎不需要你考虑网络传输和访问的细节，你完全可以像访问本地文件一样地访问网络上其他服务器文件系统上的文件。通过有效地使用共享文件系统，前面的问题将可以得到一定程度的解决，我们可以为集群中的多台实际服务器共享同一个物理存储设备，当然，用户上传的照片也将直接存储在这里。

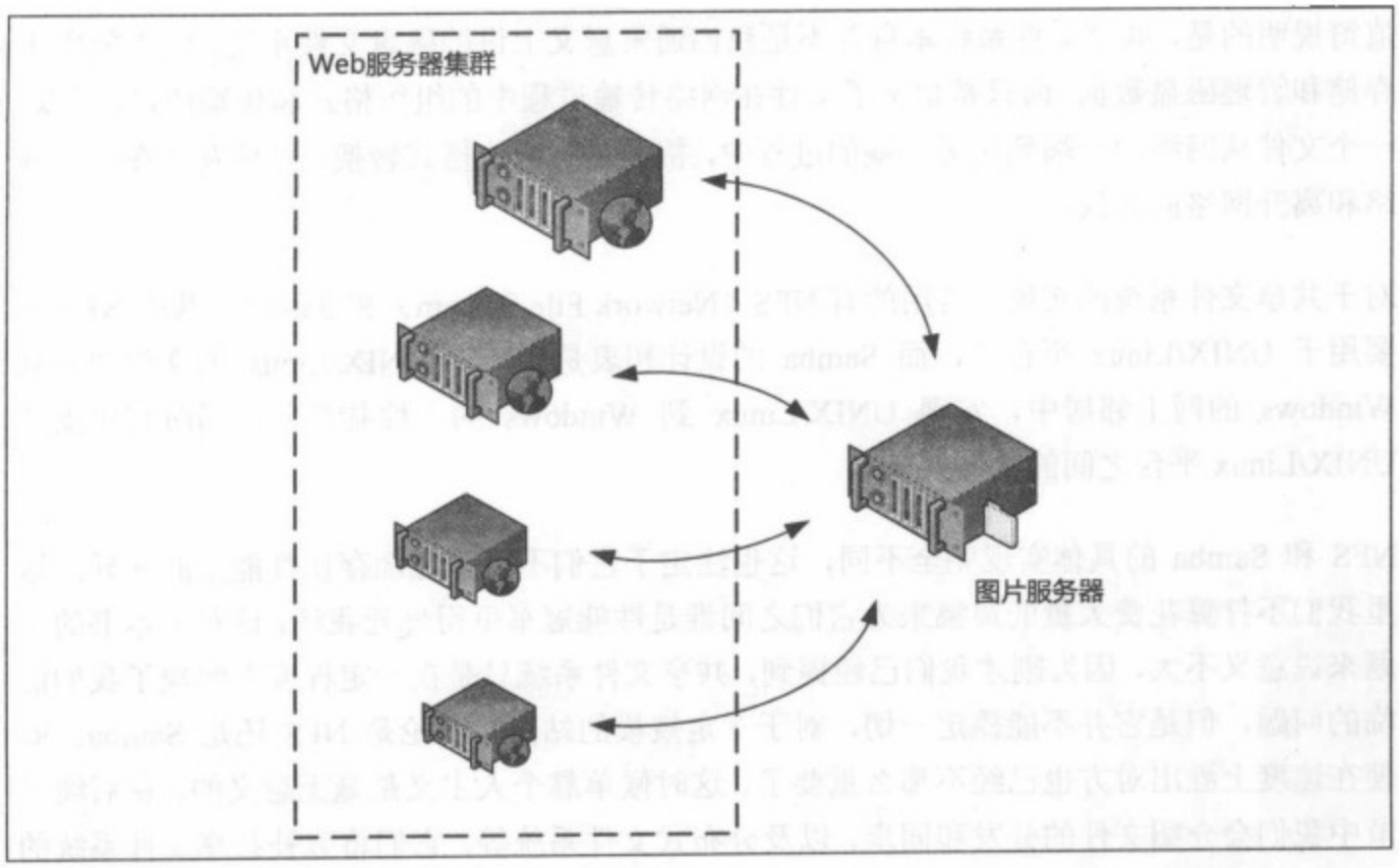


图 13-1　利用共享文件系统来实现图片共享

如图 13-1 所示，刚才提到的共享物理存储设备可以是一台独立的图片服务器，当然也可以是位于集群中的某一台实际服务器的磁盘，如图 13-2 所示。

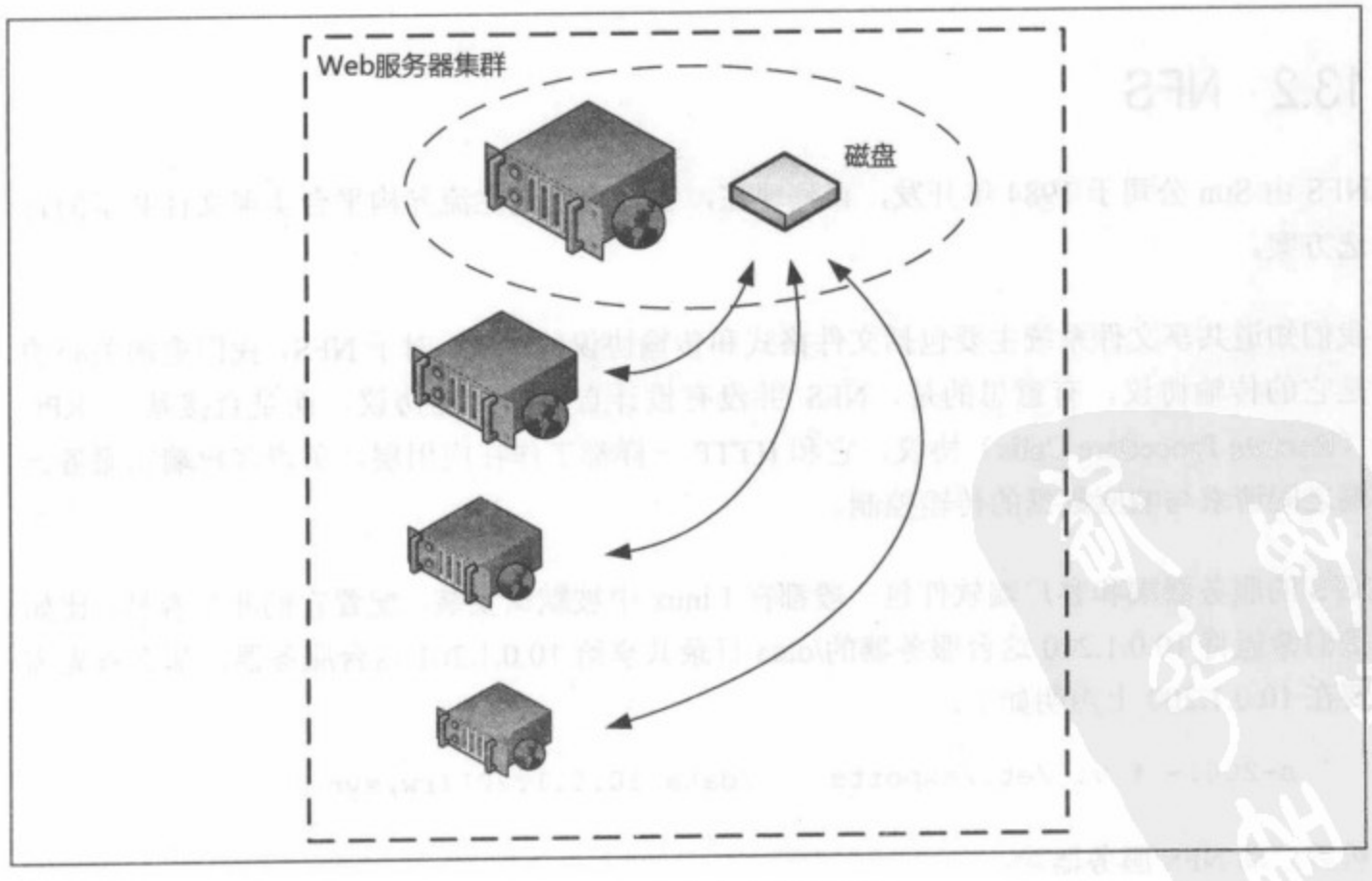


图 13-2　将集群中某台 Web 服务器同时作为文件服务器

值得说明的是，共享文件系统本身并不是我们通常意义上讲的磁盘文件系统，它不能用于存储和管理磁盘数据，而只是定义了文件在网络传输过程中的组织格式和传输协议，所以，一个文件从网络的一端到达另一端的过程中，需要进行两次格式转换，分别发生在进入网络和离开网络的时候。

对于共享文件系统的实现，常用的有 NFS（Network File System）和 Samba，其中 NFS 主要用于 UNIX/Linux 平台下，而 Samba 的设计初衷是用于将 UNIX/Linux 的文件映射到 Windows 的网上邻居中，实现 UNIX/Linux 到 Windows 的文件共享，但同时它也支持 UNIX/Linux 平台之间的文件共享。

NFS 和 Samba 的具体实现完全不同，这也注定了它们不可避免地存在性能上的差异，这里我们不打算花费大量的篇幅来为它们之间谁是性能冠军争得你死我活，这对于本书的主题来说意义不大，因为刚才我们已经提到，共享文件系统只是在一定程度上解决了我们面临的问题，但是它并不能搞定一切，对于一定规模的站点，不论是 NFS 还是 Samba，即使在速度上胜出对方也已经不那么重要了，这时候单靠个人主义是毫无意义的，在后续章节中我们会介绍文件的分发和同步，以及分布式文件系统等，它们将弥补共享文件系统的不足，更加彻底地解决我们面临的问题。

尽管如此，从使用的角度来看，NFS 和 Samba 所扮演的角色没有任何区别，接下来，我们将以 NFS 为主，介绍它将如何发挥作用，以及改善性能的方法，同时也将探讨它的局限性。

13.2 NFS

NFS 由 Sun 公司于 1984 年开发，直到现在，它一直都是主流异构平台实现文件共享的首选方案。

我们知道共享文件系统主要包括文件格式和传输协议的定义，对于 NFS，我们更加关心的是它的传输协议，有意思的是，NFS 并没有设计自己的传输协议，而是直接基于 RPC（Remote Procedure Calls）协议，它和 HTTP 一样都工作在应用层，负责客户端和服务器端之间请求与响应数据的传输控制。

NFS 的服务器端和客户端软件包一般都在 Linux 中被默认安装，配置它们非常容易，比如我们希望将 10.0.1.200 这台服务器的/data 目录共享给 10.0.1.201 这台服务器，那么首先需要在 10.0.1.200 上声明如下：

```
s-200:~ # vi /etc/exports     /data 10.0.1.201(rw,sync)
```

然后启动 NFS 服务器端：

```
    s-200:~ # /etc/init.d/nfsserver startStarting kernel based NFS server
done
```

这意味着它允许 IP 地址为 10.0.1.201 的服务器对自己的/data 目录进行读写操作，并且写操作采用同步方式。然后我们需要在10.0.1.201这台服务器上执行mount操作，将10.0.1.200共享的目录绑定到自己的文件系统中，如下所示：

```
    s-201:~ # mount -t nfs 10.0.1.200:/data /mnt/data
```

当然，在执行这行命令之前，你必须首先创建/mnt/data 目录。

关于 NFS 的更多配置和操作方法，这里就不详细介绍了，你可以查阅丰富的在线文档，这并不是很困难的事情。而接下来，我们来看看 NFS 的性能如何。

基于 RPC 传输

我们已经知道，NFS 并不包括自己的传输协议，而是利用了 Linux 内置的 RPC 服务。通过 rpcinfo，我们可以很容易地查看所有基于 RPC 的服务，对于刚才打开 NFS 服务器端的服务器，执行 rpcinfo 命令后的结果如下所示：

```
s-200:~ # rpcinfo -p 10.0.1.200
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100024    1   udp  33225  status
    100021    1   udp  33225  nlockmgr
    100021    3   udp  33225  nlockmgr
    100021    4   udp  33225  nlockmgr
    100024    1   tcp  43880  status
    100021    1   tcp  43880  nlockmgr
    100021    3   tcp  43880  nlockmgr
    100021    4   tcp  43880  nlockmgr
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
    100003    4   udp   2049  nfs
    100003    2   tcp   2049  nfs
    100003    3   tcp   2049  nfs
    100003    4   tcp   2049  nfs
    100005    1   udp    686  mountd
    100005    1   tcp    687  mountd
    100005    2   udp    686  mountd
    100005    2   tcp    687  mountd
    100005    3   udp    686  mountd
    100005    3   tcp    687  mountd
```

虽然 NFS 采用 RPC 作为应用层协议，但是其性能更多地取决于传输层协议 TCP/UDP，以及服务器端和客户端程序的实现。

对于传输层，RPC 服务默认使用了 UDP，但是当网络质量较差的时候，比如使用 WAN，

一旦数据包发生丢失，这些数据包的重发工作需要由应用层的 RPC 来进行，它将重新发送整个 RPC 请求。而如果采用 TCP 来进行底层数据传输控制，其自身的自动重传机制可以针对某个数据包进行重发，这无疑更加高效。另外，TCP 还可以更好地根据网络环境进行流量控制。

当然，使用 TCP 也有一定的缺点，它没有 UDP 无状态传输的优越性，当 NFS 服务器无法响应或者彻底崩溃后，所有的 NFS 客户端都会进入挂起等待，并且在 NFS 服务器恢复后需要重新进行 umonut 和 mount。另外，在理想的网络环境下，比如使用 LAN，TCP 的性能要低于 UDP，但这不是绝对的，它们的差异取决于具体应用，比如传输文件的大小等，你可以根据实际情况来进行测量。

NFS 的服务器端程序采用多进程（nfsd）模型，并且进程数是固定的，默认情况下为 4。大多数情况下，这些进程是难以肩负重任的，当存在大量的并发请求时，由于进程数不足，一些请求将被拒绝，为此，我们可以通过修改/etc/sysconfig/nfs 将 NFS 服务器端进程数适当提高，比如：

```
USE_KERNEL_NFSD_NUMBER="64"
```

这样一来，NFS 服务器的并发处理能力有所提高，但是仍不容乐观，当并发用户数达到 1000 时，我们的压力测试很少能够顺利完成，试想一下，在我们前面提到的照片分享站点中，如果你通过 NFS 为多台 Web 服务器提供照片共享服务，那么 NFS 服务器的请求并发数达到 1000 将不费吹灰之力，而它马上将成为罪恶的瓶颈所在，事实上，我们很少这样做，不论你怎么对 NFS 进行性能优化，NFS 注定不适合作为 I/O 密集型文件的共享方案，但是作为一般的用途，比如提供站点内部的资源共享，它的优势在于容易搭建，而且可以减少不必要的数据冗余。

统计 I/O

NFS 为我们提供了 I/O 统计界面，通过 nfsstat 命令，你可以随时查看任何 NFS 客户端对远程的文件进行了哪些操作，这些数据可以帮助你更好地了解 NFS 客户端的工作状态，以及各种远程 I/O 操作的统计和比例。

比如我们在 NFS 客户端所在的服务器上执行如下命令行操作：

```
s-201:~ # nfsstat  -c    Client rpc stats:
calls      retrans    authrefrsh  250777     0          0
Client nfs v3:
null       getattr    setattr    lookup     access     readlink
0       0% 132467 52% 161     0% 81      0% 319     0% 0       0%
read       write      create     mkdir      symlink    mknod
862     0% 115870 46% 11      0% 0       0% 0       0% 0       0%
remove     rmdir      rename     link       readdir    readdirplus
```

```
0        0% 0        0% 0        0% 0        0% 0        0% 13       0%
fsstat      fsinfo      pathconf    commit
4        0% 2        0% 0        0% 985      0%
```

可以看到，NFS 客户端一共发出了 250777 个 RPC 请求，其中大部分操作是 getattr 和 write，前者相当于对远程文件执行 stat()系统调用，它的使用非常频繁，这里占到了 52%。而对于 write 操作，我们看到它占了 46%，那么，这是否意味着 read 操作真的很少呢？不是的，其实 NFS 客户端内置了缓存机制，对于 read 操作，很多时候是不需要实际读取远程文件的，所以这里显示 read 次数为 862，并不意味着真的只有 862 次读取远程文件。

但是，即便是读取本地缓存，NFS 客户端还是需要和 NFS 服务器进行必要的通信，接下来，我们看看这些远程 I/O 操作是否存在较大的延迟。

I/O 延迟

从共享文件的使用角度来看，我们访问远程文件和访问本地文件没什么区别，所以我们愿意用 I/O 延迟来作为远程访问的性能指标。

为了测试 I/O 延迟，我们进行了一些准备，我们将一台文件服务器和一台 Web 服务器通过 100Mbps 的交换机连接起来，文件服务器通过 NFS 为 Web 服务器提供共享目录，同时将测试机也接入这个交换机。

读操作延迟测试

我们来进行读操作的对比，首先，我们对 Web 服务器的本地文件进行下载压力测试，包括两种尺寸的文件，分别为 151B 和 22KB，结果如表 13-1 所示。

表 13-1　Web 服务器本地文件的压力测试结果

静态内容大小	吞吐率（reqs/s）	数据流量（Kbytes/s）
151B	12620.14	5017.02
22KB	486.32	10676.56

可以看到，对于 151B 的文件，测试结果我们已经非常熟悉了，而对于 22KB 的文件，似乎数据流量已经接近于交换机带宽，成为制约吞吐率的瓶颈。

接下来，我们将这两个文件放在文件服务器上，同时 Web 服务器可以通过 NFS 来访问它们，我们将 Web 服务器的主目录直接指向 NFS 共享目录，再次进行压力测试，结果如表 13-2 所示。

表 13-2　Web 服务器上 NFS 共享文件的压力测试结果

静态内容大小	吞吐率（reqs/s）	数据流量（Kbytes/s）
151B	3431.88	1374.09
22KB	473.81	10386.74

从结果上看，151B 的文件吞吐率降低了很多，而 22KB 的文件吞吐率几乎没有变化，数据流量也仍然耗尽了交换机带宽，如此差异的原因是什么呢？根据前面的介绍，我们不难分析得出结论，随着单次传输数据的减少，NFS 服务器花费在响应请求上的重复开销占据了更加主要的位置，这种开销对于访问本地文件来说，只是简单的磁盘寻址而已。

当然，对于 22KB 的文件，这里的表现让我们很满意，由于测试环境的局限，我们无法通过千兆交换机来提高带宽，但可以肯定的是，当足够的带宽成为可能后，下一个即将成为瓶颈的便是 NFS 服务器的处理能力以及磁盘本身的吞吐率。

写操作延迟测试

接下来，我们看看写操作的性能，这里我们通过 cp 操作将本地文件分别复制到本地目录和远程共享目录，并通过 time 程序来统计时间，举个例子：

```
s-201:~ # time cp 22K /mnt/data/
real    0m0.059s
user    0m0.000s
sys     0m0.004s
```

这里的 real 时间便是整个 cp 过程持续的时间，也就是这个 22KB 的文件写入目标目录的时间，一共 59ms。

接下来，我们将用 22KB 和 23MB 的两个文件分别进行测试，在此之前需要说明的是，基于 NFS 的远程写操作可以有两种模式——同步和异步。对于异步模式，NFS 服务器可以在接收到写操作请求后立即返回成功，然后在后台执行实际的写操作；而对于同步模式，NFS 服务器会在写操作完成后才返回成功，不过，这里的写操作未必就是真正写入磁盘，除非你使用 O_SYNC 选项来打开文件。

好，一切准备好后，我们来进行测试，结果如表 13-3 所示。

表 13-3　远程 NFS 和本地磁盘写操作测试结果

文件大小	远程 NFS 同步写	远程 NFS 异步写	本地
22KB	0.059s	0.005s	0.002s
23MB	2.804s	2.236s	0.109s

对于 22KB 的文件，我们可以看到 NFS 异步写的结果几乎是本地的两倍以上，按照 0.005s 来计算，数据传输率为 22KB/0.005s=4.4Mbytes/s，带宽使用量为 35.2Mbps，看来并没有受到交换机 100Mbps 带宽的限制，同时也显然不会受到 NFS 服务器磁盘吞吐率的限制，可见，对于单次数据量较小的 I/O 操作，NFS 服务器的并发处理能力仍然是瓶颈所在。

对于 23MB 的文件，按照 2.236s 的异步写速度，可以算出带宽使用量已经接近交换机的 100Mbps 带宽，所以很显然，这里带宽限制了 NFS 的写速度。

不难推断出，当文件大小介于 22KB 到 23MB 之间的某个数值时，正好是 I/O 延迟的瓶颈由 NFS 服务器处理能力转移到网络带宽的临界点。这里我们没有去测试这个具体临界值，如果你有兴趣，不妨一试。

为了尽量减少交换机的带宽限制，这次我们想了一个办法，通过 NFS 来挂载本地磁盘上的目录，所有的数据通信将通过回环网络 lo 来进行，这样便几乎不受网络带宽的限制，同样对于刚才的两个文件，测试结果如表 13-4 所示。

表 13-4　本地 NFS 和本地磁盘写操作测试结果

文件大小	本地 NFS 同步写	本地 NFS 异步写	本地
22KB	0.059s	0.003s	0.002s
23MB	0.832s	0.212s	0.109s

可以看到，对于 22KB 的文件，本地 NFS 和远程 NFS 相比，I/O 操作时间几乎相同，这也印证了前面的分析，带宽限制并不是这里延迟的主要因素。而对于 23MB 的文件，这次的延迟时间大大缩短，按照本地 NFS 异步写的 0.212s 来计算，数据流量达到 108Mbytes/s，接近于实际消耗 1Gbps 的带宽；如果按照本地 NFS 同步写的 0.832s 来计算，数据流量为 27.6Mbytes/s，消耗带宽约为 200Mbps 到 300Mbps。

有一点需要注意的是，对于 NFS 异步写模式，由于存在实际写入磁盘的延迟，所以非常有可能发生数据写入失败而应用程序却并不知晓，所以，在没有其他额外保障的情况下，选择使用 NFS 异步写模式要格外慎重，当然，还有更好的方法，在后续章节中我们会介绍分布式消息队列，通过建立中间机制，既可以实现异步带来的优越性，又可以对写入失败后进行有效的跟进和弥补。

13.3　局限性

总之，不同于本地磁盘 I/O，当我们通过 NFS 来对远程文件进行读写操作时，影响性能的因素不仅包括 NFS 服务器本身的磁盘吞吐率上限，还有 NFS 服务器端的并发处理能力以及网络带宽等。

让 NFS 服务器拥有足够的网络带宽，并没有什么技术障碍，一旦我们将带宽的因素抛到脑后，共享文件系统的瓶颈便向其他位置迁移。

通过前面的读写测试，我们知道对于小文件的远程访问，NFS 服务器程序的并发处理能力并不容乐观，这时候，如果 1000 个读操作请求同时到达，即便是磁盘吞吐率尚未达到极限的 10%，可能 NFS 服务器也已经有心无力了。

对于较大的文件，当然如果也是 1000 个读操作同时到达，那情况将更加惨烈。不过在通常情况下，大文件和小文件的生命意义不同，比如一个网页可能引用了上百个小文件（如图片、样式表、脚本等），这些小文件可以附着在网页中，以较高的频率不断露脸，而大文件往往要在你经历层层关卡后才能看到下载链接。的确，由于种种复杂的因素，人们通常需要大量的小文件，而只需要少量的大文件，这似乎已成为客观规律。

回到正题，对于较大文件的访问，数据在网络上的传输占据了主要时间，这时候，网络带宽的充分消耗也意味着 NFS 服务器的磁盘吞吐率随之趋向于极限，同时意味着 NFS 服务器将成为文件访问的瓶颈，它只能同时为很少的服务器提供文件共享服务。特别是当 Web 服务器拥有较大网络带宽（如 1Gbps），并且几乎消耗殆尽的时候，这样的一台 Web 服务器就足以让 NFS 文件服务器不堪重负，当然，可以为 NFS 服务器采用一定级别的 RAID 磁盘阵列来提高并行读写能力，从而提高磁盘吞吐率的极限。

尽管如此，共享文件系统仍然是一个依赖单点的解决方案，当站点规模继续扩大时，我们便需要考虑其他的方案，比如将文件复制到更多的服务器上，建立多组冗余，下一章我们将会介绍文件分发与同步的相关知识。

第 14 章

内容分发和同步

的确，利用 NFS 等共享文件系统可以帮助我们在多台服务器之间共享文件，但是在这种机制下，不论是性能还是可用性，都无法达到更高的要求，更关键的是，共享文件系统本身就是一个不强调扩展的概念，它更像一个中央集权的统治体系，最终将成为制约发展的罪魁祸首。

> **提示：**
>
> 中央集权是相对于地方分权而言，其特点是地方政府在政治、经济、军事等方面没有独立性，必须严格服从中央政府的命令，一切受控于中央。中华人民共和国建立后，实行中国共产党宣称代表了民主集中制的人民代表大会制，中央集权和地方分权相结合，既保证中央统一领导，集中处理国家事务，同时又充分发挥地方的主动性和积极性，使地方享有一定的自主权。

对于上面这段话，道理很容易理解，不过我们要做的可没有这么复杂，对于大规模站点的文件存储和访问，我们如何把“中央集权和地方分权相结合”呢？显然，这里的“地方分权”在某种意义上意味着让 Web 服务器可以拥有独立的文件副本，所以我们需要将文件复制到多台服务器上，同时需要考虑更多的问题，那么具体如何做呢？

14.1 复制

还记得前面我们提到的图片服务器吗？当时我们采用 NFS 的方式将它映射到多台 Web 服务器上，而在这里，我们希望将图片服务器上的照片文件复制到集群中的每一台 Web 服务器上，如图 14-1 所示。

这样一来，Web 服务器将可以直接读取本地磁盘的图片来响应用户的 HTTP 请求，这意味着只要 Web 负载均衡调度器不出意外，那么文件访问本身将不再成为瓶颈，因为共享文件系统的单点性能问题已经彻底不存在了。

但与此同时，一个更重要的问题诞生了，那就是如何实现复制呢？更加严峻的问题是在面对大量的服务器进行复制时，我们需要考虑哪些策略呢？

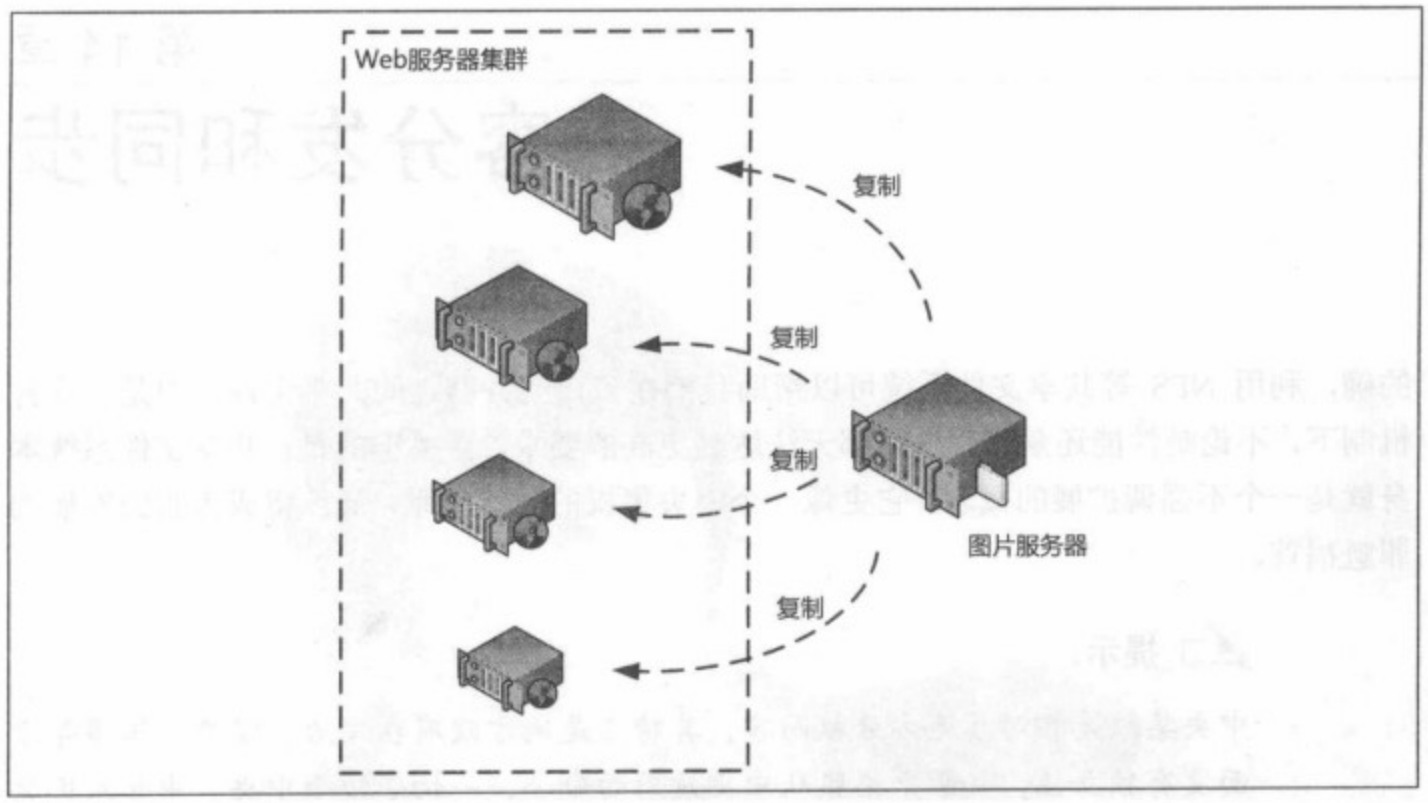


图 14-1　图片服务器到多台 Web 服务器的文件复制

提示：

每一项重大成就都会带来新的问题。任何一个发展随着时间的推移都会出现新的严重的困难。

——【美】爱因斯坦

总的来说，我们可以通过两种方式来实现复制，分别为主动分发和被动同步，主要区别在于复制的发起方和触发方式不同，所以，这里的“被动”实质上是相对于发送文件一端而言的。无论如何，它们都基于 TCP/IP 网络来传输数据。

针对这两种方式，如果条件允许，你完全可以自己开发软件来实现，这样的好处是你可以根据站点的需要来实现一些富有针对性的功能。我们曾经为站点开发了一套专用的页面分发系统，它具备一些特色，比如对于小文件的合并传输，我们知道对于小文件（如几十 KB 以下），每次传输的准备工作和收尾工作都是相当不划算的，合并传输则带来持久连接的优越性。除此之外，我们还支持异步复制，并且由分发进程来保证复制的成功率。此外，我们还可以充分利用合适的 I/O 模型和并发策略，提高分发服务器端的并发处理能力。

当然，对于大多数中等规模的站点来说，完全可以通过一系列的开源软件来实现复制，下面我们就来分别介绍它们。

14.2　SSH

提到 SSH（Secure Shell），大家并不陌生，它是建立在应用层和传输层基础上的安全协议，

可以用于传输任何数据，我们希望用它来实现文件复制，当然，这属于主动分发的方式。

SSH 有很多功能，它既可以代替 Telnet，又可以为 FTP、POP 等协议提供安全的传输通道。关于它的安全性，这里我们就不重点介绍了，而它作为传输通道的表现，正是我们所关心的，这里不得不提到它的另一个优点，那就是它对传输的数据进行了压缩，以加快传输速度。

值得一提的是，我们这里需要的并不是基于命令行的 SSH，而是将 SSH 的能力集成到站点的应用程序中，幸运的是，很多用于 Web 开发的主流语言都拥有 SSH 客户端代码库或者相应的扩展。这里我们以 PHP 为例，它可以通过 PECL 扩展来实现 SSH 客户端操作，更重要的是，它还实现了基于 SSH 的 SCP 和 SFTP，这正是我们进行文件复制所需要的。

SCP

在 PHP 程序中通过 SCP 来进行文件分发并不困难，按照函数手册的指引，不需要几行代码就可以完成服务器之间的文件复制。这里有一点需要注意，在 SSH 服务器端，也就是分发文件的目标服务器上，我们需要对 SSH 的服务器端配置选项进行一些修改，通常配置文件为/etc/ssh/sshd_config，我们修改以下内容：

```
UseDNS no
PasswordAuthentication yes
```

这样一来可以减少 SSH 服务器进行 DNS 解析的时间，并且允许我们在 Web 应用程序中通过密码验证的方式来登录 SSH 服务器。

一个用 PHP 编写的用于分发文件的例子如下所示：

```
<?php
$conn = ssh2_connect("10.0.1.201", 22);
ssh2_auth_password($conn, "user", "pwd");
ssh2_scp_send($conn,  "/home/user/list.htm",  "/home/user/list.htm",
0644);
?>
```

的确非常容易，那么，再次回到 I/O 延迟的问题上来，通过基于 SSH 通道的 SCP 进行文件分发，它的 I/O 延迟与前面的 NFS 相比，结果如何呢？

我们同样对 22KB 和 23MB 的两个文件进行了分发测试，结果如表 14-1 所示。

表 14-1　远程 SCP、本地 SCP 和本地磁盘写操作测试结果

文件大小	远程 SCP	本地 SCP	本地
22KB	0.182s	0.210s	0.002s
23MB	2.452s	0.654s	0.109s

与前面 NFS 的测试结果相比，我们不难看出，对于 22KB 的文件，SCP 花费了较多的时间，这或许归咎于额外的身份验证机制。对于 23MB 的大文件，远程 SCP 传输的时间和 NFS 的测试结果不相上下，这时候身份验证的开销已经成为次要因素，而在本地 SCP 传输中，身份验证的开销再次成为不可忽略的一部分。

显然，在绝对速度的对比中，SCP 并没有什么明显的优势，但我们要知道，关键的问题是，在我们试图通过文件复制来创建更多副本的那一刻起，绝对速度已经不是那么重要了，相比之下，如何将文件复制到更多的服务器上成为延续生命力的关键所在。所以，这又回到应该选择共享文件系统还是文件复制的问题上，对此，我们的目的已经完全不同。

在共享文件系统中，文件都存储在 NFS 服务器上，即便网络带宽可以充分满足需要，但总有一天 NFS 服务器的磁盘会成为性能的杀手，从而制约更大规模的共享访问。而在文件分发机制中，文件被复制到多台服务器上，这本身便分散了磁盘 I/O，相比于 NFS 的集中式访问，这相当于是针对磁盘 I/O 的负载均衡。

SFTP

在实际应用中，除了用 SCP 来传输文件，我们还需要对远程服务器进行必要的文件系统操作，比如创建目录、删除文件等，前面提到了 SFTP，我们可以在 PHP 中轻松地操作它，如下所示：

```
<?php
$conn = ssh2_connect("10.0.1.201", 22);
ssh2_auth_password($conn, "user", "pwd");
$sftp = ssh2_sftp($conn);
ssh2_sftp_mkdir($sftp, "/home/user/newdir", 0666);
?>
```

通过 SFTP，我们可以对远程服务器的文件系统进行任何操作，就像操作本地的文件系统一样，当然，前提是你必须拥有合法的权限。

多级分发

在较大规模的站点中，我们可能需要将文件复制到更多的服务器上，比如我们拥有 200 台静态内容服务器，它们通过 LVS 和 DNS 等混合方式实现负载均衡，并且分布在不同地域的 IDC 中，对于任何静态内容的更新，我们都需要快速地更新到这 200 台服务器上。

显然，直接将文件从文件服务器分发到 200 台 Web 服务器很容易产生以下问题：

- 由于目标服务器较多，并且部署在不同地域，分发过程会持续较长时间，这会造成一部分目标服务器表现出较大的内容更新延迟。

- 大量消耗文件服务器的本地系统资源，当分发任务较多时，本地会产生大量进程阻塞，影响其他任务的正常运行。

为此，我们可以考虑进行多级分发，如图 14-2 所示。

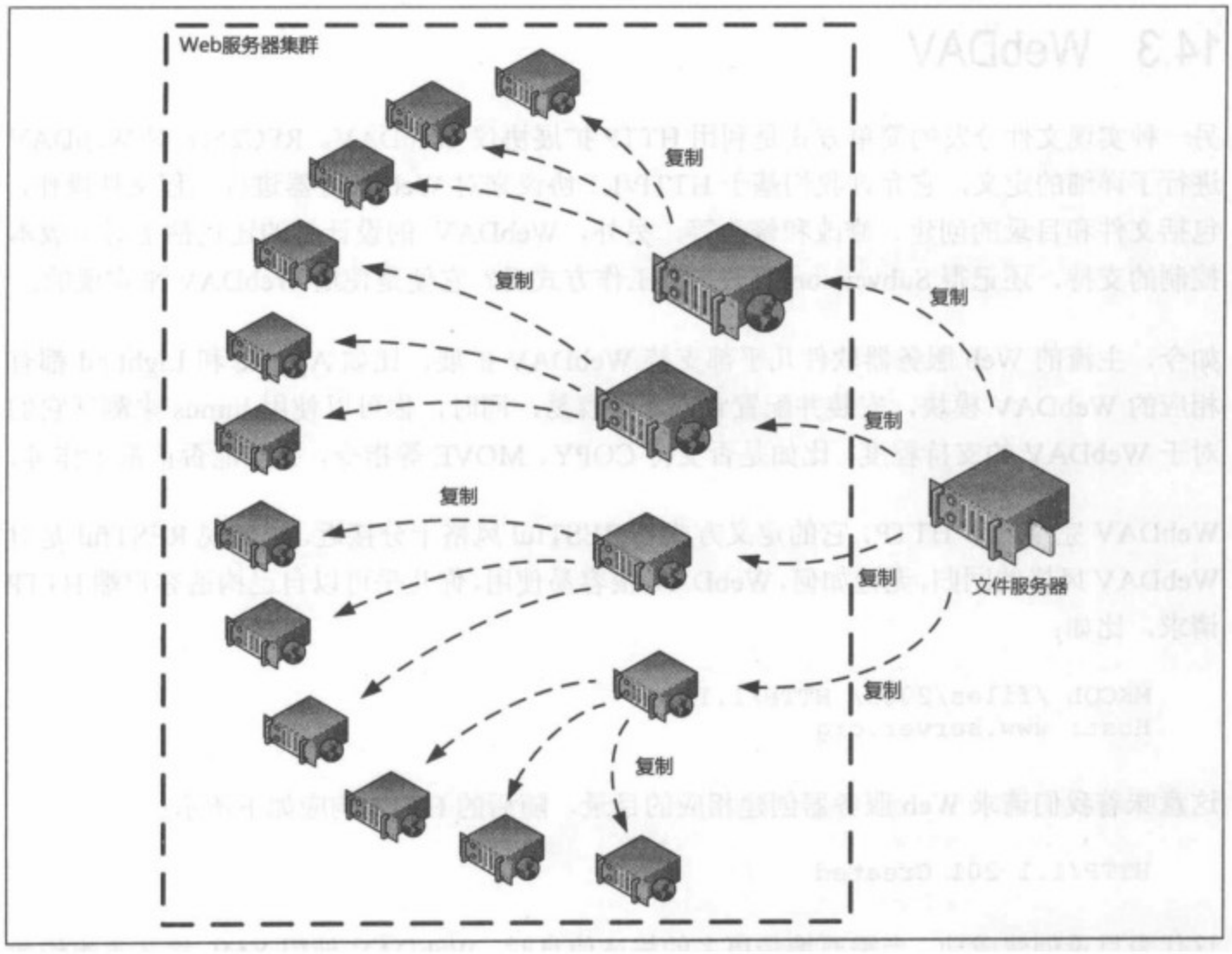


图 14-2　静态内容的多级分发

这样一来，文件服务器将很大一部分工作量巧妙地转移到了更多的 Web 服务器上，并且可以充分利用地域就近分发策略，一些 Web 服务器可以更快地将文件转发给同地域的其他 Web 服务器。

实现这样的多级分发并不困难，即便是无法开发专用的分发软件，你仍然可以基于一系列的开源产品进行快速构建，比如：

- 通过 HTTP 请求来触发某台 Web 服务器启动某文件的分发；
- Web 服务器使用异步的方式进行文件分发，分发采用前面提到的 SSH 扩展，分发结束后根据需要通知下一级进行分发。

对于异步方式，我们会在后面关于分布式计算的章节中进行介绍。另外，对于以上的多级

文件分发，我们应该将用于分发的网络进行最大程度的流量隔离，使它不要消耗 Web 服务器的 WAN 带宽。一般而言，对于跨地域或者跨 IDC 的文件分发，除非使用专线，否则必须通过 WAN 来进行，而对于同一 IDC 内的文件分发，我们完全可以根据需要来组建 LAN，实现分发流量隔离。

14.3 WebDAV

另一种实现文件分发的简单方式是利用 HTTP 扩展协议 WebDAV。RFC2518 对 WebDAV 进行了详细的定义，它允许我们基于 HTTP/1.1 协议来对 Web 服务器进行远程文件操作，包括文件和目录的创建、修改和修改等，另外，WebDAV 的设计目的还包括了对于版本控制的支持，还记得 Subversion 的 HTTP 工作方式吗？它便是使用 WebDAV 来实现的。

如今，主流的 Web 服务器软件几乎都支持 WebDAV 扩展，比如 Apache 和 Lighttpd 都有相应的 WebDAV 模块，安装并配置它们非常容易，同时，你可以使用 litmus 来测试它们对于 WebDAV 的支持程度，比如是否支持 COPY、MOVE 等指令，并且能否正常工作等。

WebDAV 完全基于 HTTP，它的定义方式与 RESTful 风格十分接近，或者说 RESTful 是对 WebDAV 风格的回归，无论如何，WebDAV 很容易使用，你几乎可以自己构造客户端 HTTP 请求，比如：

```
MKCOL /files/2009/ HTTP/1.1
Host: www.server.org
```

这意味着我们请求 Web 服务器创建相应的目录，随后的 HTTP 响应如下所示：

```
HTTP/1.1 201 Created
```

这代表目录创建成功。当需要携带更多的描述信息时，WebDAV 使用 XML 格式来组织数据，所以，我们同样可以非常容易地操作它。

后面我们要介绍的分布式文件系统 MogileFS 便采用了 WebDAV 来实现文件分发，当然，WebDAV 只是一个文件分发的具体实现，对于分发的策略和管理，前面介绍的内容同样适用。

14.4 rsync

除了 SCP 或 WebDAV 等主动分发方式，我们还可以采用被动同步的方式来实现文件复制，在这种情况下，接收文件的一端将主动向文件服务器发起同步请求，并根据两端文件列表的差异，有选择性地进行更新，从而保证它和文件服务器的内容一致。

Linux 下的 rsync 工具便可以非常出色地完成这项任务，但值得一提的是，通常情况下 rsync 可以采用 SSH 作为传输通道，但这种方式在性能上受限于 SSH 的传输机制，通过前面的

介绍，我们知道 SSH 存在一些额外开销，特别对于小文件的传输，这些开销更显得很不值得。所以，如果条件允许的话，我们尽量使用 rsync 的独立服务器端进程 rsyncd 来负责文件传输，它同时也将使用独立的服务器端口。

启动了 rsyncd 后，便可以在进程树中看到它：

```
3961 ?        Ss     0:27 /usr/sbin/rsyncd --daemon
```

通过 rsyncd 的服务器端配置，定义多个需要同步的目录，同时还可以指定账号文件，即便不使用 SSH，也可以实现身份验证。比如在以下的配置中，我们定义了 img，并指向站点的图片目录。

```
s-cache:~ # vi /etc/rsyncd.conf
gid = users
read only = true
hosts allow = 10.0.1.0/24
use chroot = no
transfer logging = true
log format = %h %o %f %l %b
log file = /var/log/rsyncd.log
slp refresh = 300
log file = /var/log/rsyncd.log
pid file = /var/run/rsyncd.pid
lock file = /var/run/rsyncd.lock
[img]
        path = /data/www/img/htdocs
        comment = img.highperfweb.com
        read only = yes
        auth users = user
        secrets file = /etc/rsyncd.secrets
```

这样一来，拥有授权 IP 地址的服务器便可以随时从这台服务器同步文件，一般来说，我们会将同步指令放置在定时任务中，比如通过以下的 crontab 设置，同步任务将每 5 分钟进行一次。

```
    */5 * * * * rsync -vzrtopg --progress --delete img@10.0.1.200::img
/data/www/img/htdocs/ > /dev/null 2>&1
```

一切都非常简单，通过定时任务，rsync 对文件的同步可以完全对应用程序透明，但这也恰恰使得应用程序无法细粒度地控制它的工作。当然，你也可以通过应用程序来很好地控制 rsync 的节奏，比如让它不间断地重复运行，即便是一个简单的 Shell 程序也能做得到，这样可以最大程度地减少被动同步机制的复制延迟。

另一方面，从我们关心的性能角度来看，对于少量文件的同步，rsync 的表现基本上无可挑剔，但是对于大量的文件，情况便不那么乐观。注意，这里所说的并不是实际复制的文件数目，而是被同步的目录中所有文件个数，因为 rsync 在同步的时候必须扫描被同步目录中的所有文件，并根据文件最后修改时间，和本地进行对比，以找出需要复制的文件，或者需要删除的文件，这样一来，当目录中存在大量的文件时，扫描的开销便可想而知。

我们来看一个例子，以下的 rsync 同步命令将执行近 2 分钟。

```
    s-img:~ # rsync -vzrtopg --progress --delete img@10.0.1.200::img /data
/www/img/htdocs/
    receiving file list ...
    612600 files to consider

    sent 66 bytes  received 15109850 bytes  165135.69 bytes/sec
    total size is 11426157579  speedup is 756.20
```

可以看到，当扫描结束后，并没有引发实际的复制操作，这说明本地目录中的文件和文件服务器上是保持一致的，但即便是这样，也花费了近 2 分钟的时间。

那么，对于大量文件的同步，有什么更好的办法吗？

14.5 Hash tree

rsync 在同步文件时需要分析目录中每一个文件的更新标记，当有多级目录时，依然如此。如果能让它在减少扫描范围或者次数的同时，仍然可以实现目的，那当然是我们所希望的。

也许你已经发现，事实上在大多数时候，很多文件并没有更新，为什么不把少量文件的更新标记不断地传递到上一级目录呢？这样便可以让扫描过程少走很多冤枉路。

如图 14-3 所示，在文件服务器上，假如底层的灰色文件发生了更新，我们需要让它的上级目录也随之更新，最后修改时间，并且向上以此类推，直到顶层目录，这样一来，从这个文件到根目录的一系列节点都会更新最后修改时间，当同步扫描的时候，只需要遍历那些修改时间更新的目录即可，扫描次数将大大减少。

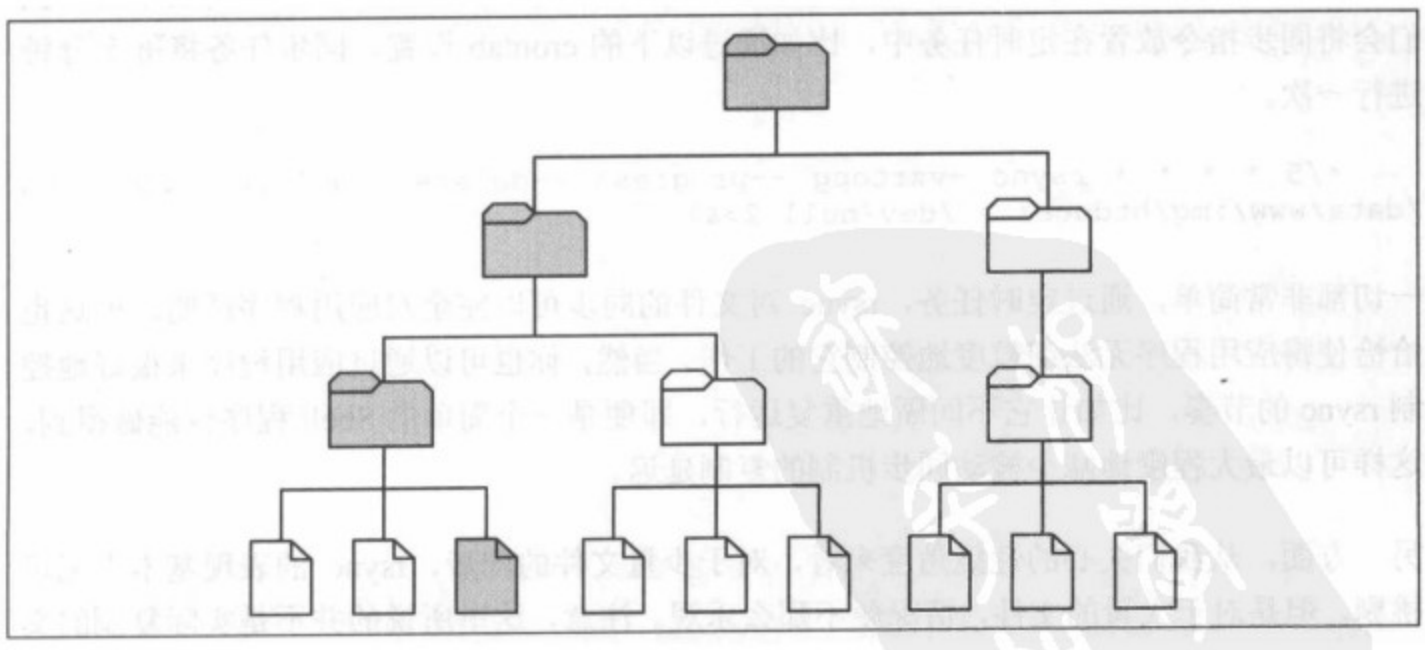


图 14-3　将文件的修改时间传递到上级目录

但是，操作系统本身对于文件的修改并不会自动更新上级目录的修改时间，一些特定的应

用程序会这样做，比如通过 VI 编辑某个文件并保存后，你会发现它的所有上级目录都会自动更新修改时间。那么，对于文件同步，我们也必须想办法自己来实现。

幸运的是，Linux 内核中的 inotify 模块可以帮助我们完成这一工作，简单地说，它通过在内核中监视文件系统的一举一动，可以在任何文件的修改时间发生变化后，发出事件通知。这样一来，我们可以通过 inotify 提供的 API 来编写守护进程，持续不断地监视文件修改事件，一旦发现某个文件被修改，便随之更新上级甚至更上一级目录的修改时间，直到被同步目录的根节点。

inotify 本身提供了基于 C 语言的 API，除此之外，你也可以在 PHP 中使用相应的 PECL 扩展，这将大大简化我们的开发工作。有一点需要注意，inotify 已经直接包含在了 Linux2.6.13 的内核中，在此之前，我们必须安装相应的补丁，所以在编译 PHP 扩展的时候，你需要检查 Linux 的内核版本。

```
    s-colin:~ # uname -a
    Linux s-mat 2.6.16.21-0.8-bigsmp #1 SMP Mon Jul 3 18:25:39 UTC 2006 i686
i686 i386 GNU/Linux
```

这种将文件的修改时间不断向上传递的机制体现了 Hash tree 的动机，但有所不同的是，在 Hash tree 的存储结构中，我们通过特定的 Hash 算法，为树中的每个非叶节点生成一个可以描述它所有子节点唯一性的特征字符串，以便提供对节点信息完整性和正确性的验证，这样一来，一旦某个节点的信息发生变化，那么这种变化便会向上传递，体现在上级节点的特征字符串中。而这里，我们直接将文件修改时间作为 Hash 值，这非常适合于文件同步这种依赖时间的特殊应用，而且易于实现，几乎不需要额外的计算开销，的确，文件系统本身就是树形结构，Hash tree 的优越性在这里得到了很好的发挥。

到目前为止，我们所做的还只是通过 inotify 实现了文件修改时间的向上传递，除此之外，我们还必须实现基于这种机制的专用服务器端和客户端程序，因为 rsync 并不支持我们所期望的 Hash tree 式扫描，当然，也许你可以尝试修改 rsync 的源代码。另一方面，利用现有的通道也是一个值得考虑的选择，比如你仍然可以借助 SFTP 来获取远程目录列表，并且分析修改时间，然后通过 SCP 来获取需要同步的文件。

利用这种同步方式，我们还需要注意一点，为了最大程度地发挥 Hash tree 的优势，我们需要在目录规划时遵循一些原则，尽量让目录中的文件或者子目录保持在较少的数量，以减少扫描次数，当然，这也必然让扁平的目录结构变得更加冗长，带来更多的目录切换次数，但是，你可以根据实际情况来很好地把握平衡。

14.6 分发还是同步

总的来说，对于分发和同步这两种文件复制方式，你也许在思索究竟应该在站点中选择谁

呢，这大可不必发愁，其实它们两者本质上没有太大的差别，都可以达到相对来说比较实时的文件复制，而在选择中起到决定作用的，往往是出于以下一些因素：

- 文件分发需要依赖一定的应用程序逻辑，比如通过 SCP 扩展来编写代码控制文件传输，或者自己开发专用的分发程序，而文件同步可以对应用程序完全透明，部署非常简单；
- 文件分发可以更好地控制触发条件，更加灵活，比如更容易实现多级复制，也就是前面介绍的多级分发；
- 文件同步（比如利用 rsync）是一种天然的异步复制操作，不阻塞 Web 应用程序的运行，而在 Web 应用中要想对文件分发实现异步复制操作，必须得借助其他支持，比如后面要介绍的异步计算。

根据站点的需要，选择合适的复制方式并不困难，相信你的感觉吧！

除此之外，还有一种复制方式，那就是反向代理，的确，它是一个多才多艺的家伙，在这里，我非常乐意将它也归入文件复制的范畴，接下来，我们看它如何发挥作用。

14.7 反向代理

从反向代理缓存到负载均衡调度器，反向代理服务器不止一次出现在书中，而这一次，我们又将目光转向了它在文件复制领域的表现。

事实上，反向代理机制本身就决定了它必须从后端服务器那里不断地复制内容到本地，而这种复制的触发条件，则是用户向反向代理服务器请求内容，我想这种方式也可以称为“动态同步”。

那么，对于静态网页、图片等这些直接暴露给用户的内容来说，通过反向代理服务器实现远程复制，可能是一个更加简单有效的方式。而对于另一些无法由用户通过 HTTP 直接访问的文件，比如 Web 应用程序运行中需要的一些持续更新的配置文件，反向代理服务器对于它们的复制将无能为力，你仍然需要借助于前面介绍的复制方法。

这样一来，我们或许可以将集群中的 Web 服务器直接作为文件服务器的反向代理，或者采用多级代理的方式，类似于多级分发，减少单点压力，同时有利于同地域服务器之间的就近复制。

另一方面，正是因为存在反向代理缓存，所以使用反向代理实现文件复制才有意义，因为这些静态网页或者图片可以长时间保留在反向代理服务器上，快速地为用户提供下载服务，而只有用户第一次请求它们的时候才需要触发复制。

但是，缓存机制也提醒了我们，任何内容总是会有过期的时候，假如这些内容需要频繁的更新，同时你也希望给用户及时的体验，那么让这些内容频繁过期便在所难免。显然，这样会引发频繁的复制，别忘了复制是需要花时间的，就像羊毛出在羊身上，这部分时间仍然需要由用户来承担，你会发现相当一部分用户都在为反向代理服务器到文件服务器的数据传输而买单。

当然，这是比较极端的情况，而在大多数时候，我相信你的站点会有很多可以长期在反向代理服务器上保留的内容。

使用反向代理的另外一个好处是，这种动态同步是基于标准的 HTTP，那么对于同步所需的服务器端程序，相比于 sshd 和 rsyncd，Web 服务器成为最佳选择，你对它已经再熟悉不过了，特别是它出色的并发处理能力，这使得它可以提供非常高效可靠的文件复制。

第 15 章

分布式文件系统

在开始这一章之前，让我们先放慢脚步，简单回顾一下。

当我们为多台 Web 服务器实现负载均衡后，用户的请求通过调度器被分散到不同的 Web 服务器上。但是，随之产生一个问题，多台 Web 服务器如何共享一些文件呢？比如用户上传的照片、定时更新的静态化页面等。这时候，我们通过 NFS 实现了文件共享，这使得所有 Web 服务器都可以像访问本地磁盘一样访问同一个远程目录，但是没过多久，它便不能胜任成为众人关注的焦点，大量的集中访问让它喘不过气来。于是，我们开始考虑为文件创建多份副本，将它们复制到多台服务器上。

为此，我们通过分发和同步等方式来实现复制，的确，它们工作得非常好，而且掌握它们并不困难，但是，总有一些新的问题让我们感到不安，事实上这些问题并不是来自于它们本身，比如：

- 需要维护大量的 rsync 同步脚本，或者需要配置大量的分发参数，尤其对于大规模的集群，这些工作很容易让人产生挫败感，而且毫无乐趣；
- 缺乏整体的管理和监控，我们需要能够一览无余地了解所有复制工作的状态，以及更多细节报告，这有助于我们更好地了解它们在运行中的状态。

当这些问题开始浮现后，也许你已经意识到，我们需要做得更好，这时候，分布式文件系统是一个值得考虑的选择，它既可以满足文件复制的需要，同时又带来了像文件系统一样的可管理性。

当然，刚才的问题或许对于你的站点来说不算什么，但是，分布式文件系统的作用远不仅如此，当你了解它之后，也许就会有新的发现。

15.1 文件系统

事实上，分布式文件系统并不是传统意义上的文件系统，它工作在操作系统的用户空间，由应用程序来实现，比如 MogileFS 或 Hadoop 等。所以，这使得分布式文件系统并不依赖于底层文件系统的具体实现，只要是 POSIX 兼容的文件系统即可。

相对于底层文件系统，分布式文件系统更像是一个抽象层的实现，这使得它可以避免文件系统的诸多限制，并且拥有自己独特的内容组织结构，比如 MogileFS 支持域和类，便于

对大规模存储和复制进行合理规划。

那么，如何来访问分布式文件系统呢？通常，分布式文件系统会提供专用的访问接口，比如以 Hadoop 实现的分布式文件系统为例，我们来对它进行一系列的访问操作。首先，我们来查看目录列表：

```
s-img:~ # hadoop dfs -ls
ls: Cannot access .: No such file or directory.
```

可以看到，hadoop 程序为我们提供了访问入口，通过不同的参数选项，我们可以非常轻松地操作分布式文件系统，接下来，我们创建 html 目录，如下所示：

```
s-img:~ # hadoop dfs -mkdir html
```

然后我们就可以将本地文件复制到分布式文件系统中，如下所示：

```
s-img:~ # hadoop dfs -put /data/www/htdocs/index.htm html/
```

非常好，我们将本地的 index.htm 文件复制到了刚刚创建的 html 目录中，有点像操作 FTP 的感觉。现在，我们可以查看 html 目录下的文件列表，如下所示：

```
s-img:~ # hadoop dfs -ls html
Found 1 items
-rw-r--r--   3 root supergroup        153 2009-06-14 21:55 /user/root
/html/index.htm
```

返回的结果看起来非常亲切，跟本地文件系统没什么区别。现在我们来读取文件内容，如下所示：

```
s-img:~ # hadoop dfs -cat html/index.htm
<html>
......
</html>
```

的确，从文件系统的传统思想来看，这种访问方式非常优雅。的确，就是这么简单。同样，我们再以 MogileFS 为例，看看它提供的访问方式。

有点不同的是，MogileFS 没有目录的概念，它要求每个文件都要有唯一的 Key，比如我们希望将 index.htm 文件复制到 MogileFS 分布式文件系统中，如下所示：

```
s-mat:~ # mogtool --trackers=10.0.1.200:6001 --domain=html.domain
inject /data/www/htdocs/index.htm index
file index.htm: 478d8ced1f9dc85e6e16e68f8fa77291, len = 14978
Spawned child 1018 to deal with chunk number 1.
        chunk 1 saved in 0.01 seconds.
Child 1018 successfully finished with chunk 1.
Replicating: 1
```

这里我们为 index.htm 创建了名为 index 的 Key，同样，当我们读取这个文件时，也必须指定 Key 为 index，如下所示：

```
    s-mat:~ # mogtool --trackers=10.0.1.200:6001 --domain=html.domain
extract index -
    Fetching piece 1...
              Trying http://10.0.1.201:7500/dev1/0/000/000/0000000784.
fid...
    <html>
    ......
    </html>
```

看来，Hadoop 和 MogileFS 的具体访问方式各不相同，这取决于它们的不同设计和实现，的确，每一个实现分布式文件系统的具体产品都有个性化的访问接口，但透过访问方式的差异，它们所扮演的角色往往都是相同的。

另一个问题是，这些文件在分布式文件系统中是以何种形式被存储的呢？就拿刚才的 index.htm 文件来说，MogileFS 中为每个文件分配一个自增的整数 ID，它将 index.htm 存储为以下的实际路径：

```
/data/mogdata/dev3/0/000/000/0000000066.fid
```

而在 Hadoop 中，index.htm 被存储在以下路径：

```
/data/hadoop/data/current/blk_-1995501119394658830
```

可见，分布式文件系统的内部存储结构只有它自己知道，我们必须通过它提供的统一接口来读取文件。

另一方面，出于整体性能以及降低复杂度的考虑，分布式文件系统并没有设计得面面俱到，比如通常我们不能对分布式文件系统中的文件进行修改，因为这样会涉及锁定和版本，必然增加大量额外开销，而事实上，对于大多数 Web 应用，我们可以很容易地规避文件修改，而采用增加副本的方式，比如用户修改照片实际上等同于上传新的照片。

15.2 存储节点和追踪器

前面我们通过特定的访问接口来操作分布式文件系统，我们所看到的是一个整体，然而，在简单的访问接口背后，也就是分布式文件系统的内部，它可能跨越多台服务器，并根据规则进行自动的文件复制，这些是如何实现的呢？

首先，在分布式文件系统中，所有的文件都存储在被称为存储节点（Stroage Node）的地方，一个存储节点往往对应物理磁盘上的一个实际目录，比如在 MogileFS 中的一个存储节点位置如下所示：

```
/data/mogdata/dev3
```

这样一来，我们用多台服务器创建多个存储节点，文件便会在这些存储节点之间根据规则进行自动复制。另外，我们也可以在同一台物理服务器上创建多个存储节点，分别指向不同的磁盘，实现冗余备份，相当于某种意义上实现了 RAID 机制。

对于多个存储节点，客户端如何与它们打交道呢？必须得有一定的调度机制，而追踪器（Tracker）正是负责完成这项工作，它发挥着客户端和存储节点之间桥梁的作用，就像负载均衡系统中的调度器一样，但应该说更像基于 DNS 的调度器，因为大多数时候，当客户端通过追踪器找到合适的存储节点后，客户端将直接与存储节点服务器进行文件读写通信，如图 15-1 所示，这样设计也可以最大程度地减少追踪器的瓶颈效应。

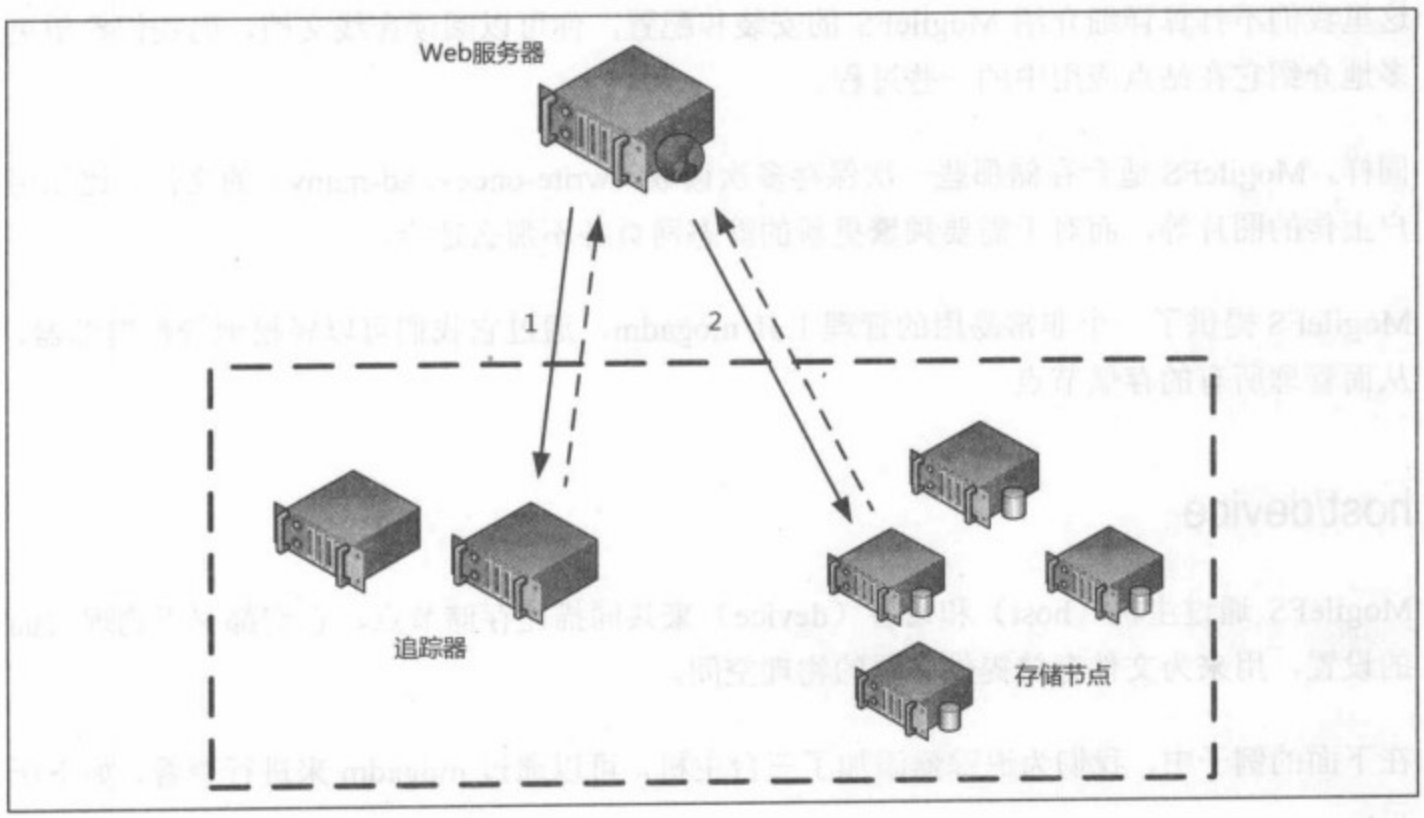


图 15-1　追踪器和存储节点

事实上，追踪器要做的事情非常多，通常包括：

- 维护存储节点的各种信息，知道哪些文件存储在哪些节点上，这些信息可以存储在文件或者数据库中，比如 MogileFS 使用 MySQL 来存储；
- 监听在特定的 TCP 端口，与客户端以及存储节点进行通信，为客户端提供文件访问接口；
- 控制文件复制策略；
- 实现存储节点的负载均衡；
- 实现存储节点的故障转移，提高存储系统的可用性。

从另一种角度来看，追踪器负责维护抽象层面的存储空间，而存储节点负责维护物理层面的存储空间。

接下来我们以 MogileFS 为例，看看它在应用中发挥的作用。

15.3 MogileFS

MogileFS 是一个开源的分布式文件系统，它采用 Perl 编写，包括追踪器、存储节点，以及一些管理工具，除此之外，追踪器使用 MySQL 来存储分布式文件系统运行中的所有信息。

这里我们不打算详细介绍 MogileFS 的安装和配置，你可以阅读在线文档，而我们希望更多地介绍它在站点应用中的一些过程。

同样，MogileFS 适合存储那些一次保存多次读取（write-once-read-many）的文件，比如用户上传的图片等，而对于需要频繁更新的静态网页并不那么适合。

MogileFS 提供了一个非常易用的管理工具 mogadm，通过它我们可以轻松地管理追踪器，从而管理所有的存储节点。

host/device

MogileFS 通过主机（host）和设备（device）来共同描述存储节点，它们都属于物理层面的设置，用来为文件存储提供必要的物理空间。

在下面的例子中，我们为追踪器添加了三台主机，可以通过 mogadm 来进行查看，如下所示：

```
s-mat:~ # mogadm host list
img.1 [1]: alive
  IP:       10.0.1.201:7500
img.2 [2]: alive
  IP:       10.0.1.202:7500
img.3 [3]: alive
  IP:       10.0.1.203:7500
```

当然，这三台主机都必须运行 mogstored，也就是 MogileFS 中存储节点的服务器端程序，我们可以在 ps 中看到它：

```
 6567 ?        Sl     3:16 mogstored
 6568 ?        S      0:10  \_ mogstored [diskusage]
 6569 ?        S      0:00  \_ mogstored [iostat]
20915 ?        S      0:00  |   \_ iostat -dx 1 30
 6572 ?        S      0:00  \_ mogstored [fidsizes]
```

作为存储节点的守护程序，mogstored 监听在两个端口上，正如它的配置文件所示：

```
httplisten=0.0.0.0:7500
mgmtlisten=0.0.0.0:7501
```

mogstored 通过 7500 端口为客户端提供基于 WebDAV 的访问通道，而通过 7501 端口与追踪器进行必要的通信，追踪器也正是通过 7501 端口来管理这些存储节点。

在 MogileFS 中，我们还需要为主机创建存储设备，这种概念有点类似于磁盘分区，这也正是 MogileFS 中的存储节点，我们来看看已经创建好的设备列表：

```
s-mat:~ # mogadm device list
img.1 [1]: alive
                  used(G)  free(G)  total(G)
   dev1: alive     46.485   32.262   78.747
   dev3: alive     46.485   32.262   78.747
img.2 [2]: alive
                  used(G)  free(G)  total(G)
  dev11: alive     21.311   63.071   84.382
img.3 [3]: alive
                  used(G)  free(G)  total(G)
  dev10: alive     51.729   22.095   73.823
   dev5: alive     51.729   22.095   73.823
```

可以看到，一共有 5 个存储节点，它们分布在 3 台主机上，所有存储节点的空间使用情况一览无余，这正是我们想要的。有意思的是，我们在一些主机上创建了两个存储设备，它们的空间使用情况完全相同，这本身并没有错误，MogileFS 目前还不支持对存储空间进行直接的配额限定，所有的设备都将获得当前磁盘的所有空间。

domain/class

就像前面提到的，存储节点的创建非常容易，但是，它们仅仅意味着一系列的物理存储空间，而至于文件的复制策略却只字未提。

所以，MogileFS 在抽象层面设计了两个概念——为域（domain）为类（class），它们与前面的主机和设备没有从属关系。

域相当于命名空间，在一个域中，所有文件的 Key 都是唯一的。域中包含了若干个类，类是用于复制的整体单位，它可以指定复制个数，也就是文件的副本数，这样一来，只要你将文件提交到这个类中，MogileFS 便会自动地复制指定数量的副本到其他的存储节点。

我们通过 mogadm 看看域和类的列表：

```
s-mat:~ # mogadm domain list
 domain               class                mindevcount
-------------------- -------------------- ------------
 img.domain           default              2
 img.domain           userpic              2
 html.domain          default              2
 html.domain          all                  5
```

这里的 mindevcount 代表复制个数，我们看到在 html.domain 域中，all 类的复制个数为 5，这意味着该类中的所有文件都将被复制到所有的 5 个存储节点中。

存储节点可用性检测

MogileFS 可以自动对所有存储节点进行实时检测，一旦发现某个存储节点发生故障，便会在随后的运行中忽略这个节点。这里我们故意关闭其中一个主机，可以通过 mogadm 来查看各存储节点的状态，如下所示：

```
s-mat:~ # mogadm check
Checking trackers...
  10.0.1.200:6001 ... OK
Checking hosts...
  [ 1] img.1 ... OK
  [ 2] img.2 ... REQUEST FAILURE
  [ 3] img.3 ... OK
Checking devices...
  host device    size(G)    used(G)    free(G)  use%    ob state    I/O%
  ---- -------- ---------- ---------- ---------- ------ ---------- -----
  [ 1] dev1       78.748     46.477     32.271  59.02%  writeable   0.0
  [ 1] dev3       78.748     46.477     32.271  59.02%  writeable   0.0
  [ 3] dev5       73.824     51.722     22.103  70.06%  writeable   0.0
  [ 3] dev10      73.824     51.722     22.103  70.06%  writeable   0.0
  ---- ---------- ---------- ---------- ---------- ------
         total:   305.144    196.397    108.747  64.36%
```

可以看到，其中一台主机被标记为失败，同时位于这台主机上的存储节点也消失了。

现在我们再次启动那台主机，看看状态有何改变，如下所示：

```
s-mat:~ # mogadm check
Checking trackers...
  10.0.1.200:6001 ... OK
Checking hosts...
  [ 1] img.1 ... OK
  [ 2] img.2 ... OK
  [ 3] img.3 ... OK
Checking devices...
  host device    size(G)    used(G)    free(G)   use%    ob state    I/O%
  ---- -------- ---------- ---------- ---------- ------ ---------- -----
  [ 1] dev1       78.748     46.477     32.271  59.02%  writeable   0.0
  [ 1] dev3       78.748     46.477     32.271  59.02%  writeable   0.0
  [ 2] dev11      84.382     21.309     63.073  25.25%  writeable   0.0
  [ 3] dev5       73.824     51.722     22.103  70.06%  writeable   0.0
```

```
[ 3] dev10      73.824     51.722     22.103  70.06%  writeable   0.0
---- ------------ ---------- ---------- ---------- ------
           total:   389.526    217.706    171.820  55.89%
```

刚才失败的主机又恢复了状态，dev11 存储节点也重新可用。

也许你会产生疑问，在 dev11 故障期间本应该复制给它的文件，是否会在它恢复之后自动复制到 dev11 上呢？当然会，这应该是所有支持复制模式的分布式文件系统必须具备的能力，它至关重要。在 MogileFS 中，你还可以监视复制的状态，后面我们会详细介绍。

除此之外，MogileFS 还为 mogadm 提供了 fsck 选项，它可以帮助我们对文件进行检查和修复。

多个副本的负载均衡

追踪器的一个重要工作是扮演存储节点的访问调度器，实现存储节点的负载均衡，这使得存储节点可以非常容易地进行水平扩展。

通过 MogileFS 的客户端 API，我们可以在应用程序中与追踪器进行通信，实现对分布式文件系统的访问。这里我们使用 PHP 来编写一段程序，用来将文件写入到 MogileFS 分布式文件系统中，其中用到了 PHP 的 MogileFS 扩展，代码如下所示：

```
<?php
$mg = new MogileFS();
$mg->connect('10.0.1.200', 6001, 'html.domain');
$key = "index";
$mg->put("/data/www/htdocs/index.htm", $key, 'all');
$mg->close();
?>
```

这段程序不难理解，程序运行后首先连接追踪器，并且指定 html.domain 域，随后上传本地文件 index.htm 到 all 类中，同时指定文件的 Key 为 index。

就在这短短的几行代码中，隐藏了很多细节，我们知道，事实上，追踪器在这里的任务只是告诉应用程序有哪些可用的存储节点，而传输文件的过程则是直接在应用程序和存储节点之间通信完成。

当这段程序执行后，按照我们的规划，index.htm 文件应该被复制到 5 个存储节点中，因为 all 类的复制个数被我们设置为 5。

接下来我们同样通过 PHP 来读取这个文件，由于 MogileFS 存储节点的服务器端程序本身便是支持 WebDAV 的 HTTP 服务器，所以追踪器会告诉我们文件的实际 URL，我们编写

的程序如下所示：

```
<?php
$mg = new MogileFS();
$mg->connect('10.0.1.200', 6001, 'html.domain');
$key = "index";
$paths = $mg->get($key);
$mg->close();
var_dump($paths);
?>
```

运行这段程序，结果如下所示：

```
s-mat:/data/www/htdocs # php ./mogile.php
array(3) {
  ["path2"]=>
  string(51) "http://10.0.1.201:7500/dev1/0/000/000/0000000586.fid"
  ["path1"]=>
  string(51) "http://10.0.1.203:7500/dev5/0/000/000/0000000586.fid"
  ["paths"]=>
  string(1) "2"
}
```

再次运行这段程序，结果如下所示：

```
s-mat:/data/www/htdocs # php ./mogile.php
array(3) {
  ["path2"]=>
  string(51) "http://10.0.1.201:7500/dev3/0/000/000/0000000586.fid"
  ["path1"]=>
  string(56) "http://10.0.1.203:7500/dev10/0/000/000/0000000586.fid"
  ["paths"]=>
  string(1) "2"
}
```

可见，追踪器会采用一定的调度策略，将我们读取文件的请求指向不同的存储节点，同时，可用性检测在这里仍然有效，假如现在我们关闭其中一个存储节点，那么追踪器将不会返回指向它的 URL。

异步复制

我们知道，MogileFS 对于提交的文件会根据它所属类的规则进行复制，一旦应用程序将文件提交到追踪器指定的某个存储节点后，其他的复制工作将在后台异步进行，不会阻塞应用程序。

在分布式文件系统中，复制是一项重要的工作，MogileFS 为我们提供了复制状态监视工具，如下所示：

```
s-mat:/data/mogdata # mogadm stats
Fetching statistics...
Statistics for devices...
  device     host           files     status
  ---------- ---------- ---------- ----------
  dev1       img.1            129      alive
  dev3       img.1            125      alive
  dev5       img.3             89      alive
  dev10      img.3            103      alive
  dev11      img.2            144      alive
  ---------- ---------- ---------- ----------
Statistics for file ids...
  Max file id: 585
Statistics for files...
  domain               class          files
  -------------------- ---------- ----------
  html.domain          default            2
  html.domain          all              222
  -------------------- ---------- ----------
Statistics for replication...
  domain               class       devcount      files
  -------------------- ---------- ---------- ----------
  html.domain          default            2          2
  html.domain          all                1        124
  html.domain          all                2          1
  html.domain          all                3          8
  html.domain          all                4         39
  html.domain          all                5         50
  -------------------- ---------- ---------- ----------
```

通过它，我们了解了很多重要信息，比如各个存储节点上的文件个数、当前最大文件的 ID 等，最下面的两个报表很有意义，其中第一个报表告诉我们每个类中有多少个文件，第二个报表告诉我们这些文件目前已经被复制到多少个存储节点，比如在上述报表中，我们可以看到，在 all 类中的所有 222 个文件中，有 50 个文件已经被复制到了 5 个存储节点，完成了所有的复制，而另有 39 个文件则只复制到了 4 个存储节点，显然它们还需要最后一次复制，但是，这 39 个文件具体是哪些，我们并不知道，或许也并不关心。

不必怀疑的是，即便文件没有完成预期的所有复制，也不会影响客户端对文件的访问，因为追踪器总是会告诉客户端可用的 URL。

经过一段时间后，文件自然会完成所有复制，这时候我们会在监控状态中看到如下结果：

```
Statistics for files...
  domain               class          files
  -------------------- ---------- ----------
  html.domain          default            5
  html.domain          all              418
  -------------------- ---------- ----------
Statistics for replication...
  domain               class       devcount      files
  -------------------- ---------- ---------- ----------
```

```
html.domain          default              2          5
html.domain          all                  5        418
-------------------- ---------- --------- ----------
```

的确，所有提交到 all 类的文件都已经完成了 5 次复制，你可以放心了。

存储节点和 WebDAV

我们知道，MogileFS 中存储节点的默认服务器端程序 mogstored 通过 WebDAV 实现文件的复制，当然，既然是基于标准的 WebDAV，那么我们同样可以使用其他支持 WebDAV 的 Web 服务器来取而代它，比如 Nginx，只需要让它支持 GET/PUT/DELETE 等 HTTP 操作即可。

在 Nginx 中配置 WebDAV 非常简单，以下是一个配置的例子：

```
http {
        include /etc/nginx/mime.types;
        default_type application/octet-stream;
        sendfile on;
        keepalive_timeout 0;
        tcp_nodelay on;
        client_max_body_size 251m;
        client_body_temp_path /var/mogdata/.tmp;
        server_tokens off;
        server {
                listen 7500;
                server_name localhost;
                charset utf-8;
                location / {
                        root /data/mogdata/;
                        dav_methods put delete;
                        dav_access user:rw group:rw all:r;
                        create_full_put_path on;
                }
        }
}
```

这样一来，MogileFS 的存储节点将拥有 Nginx 在静态内容并发处理方面的优异表现，更好地为客户端提供读写访问。

提取文件

当我们将图片存入分布式文件系统后，通过 Web 服务器读取它似乎就不那么容易了，我们必须借助专用的访问接口，这也意味着我们必须通过应用程序来读取它。比如我们将 /upphoto/200906200001.jpg 存入 MogileFS，对应的 Key 为 200906200001.jpg，提取文件的过程可以简单描述为以下几个步骤：

1. Web 服务器接收到/images/200906200001.jpg 的请求。

2. Web 服务器将请求重写（URL Rewrite）到/loadphoto.php?path=/images/200906200001.jpg。

3. oadphoto.php 截取参数中的文件名，获得 Key。

4. loadphoto.php 与追踪器通信，获得图片在某存储节点上的实际地址 URL。

5. loadphoto.php 读取这个 URL 对应的图片内容，并输出到 HTTP 响应数据中。

的确，整个提取过程显得迂回曲折，让我们无法容忍，关键在于，区区一个图片的读取，带来了大量的额外开销，对此我们不能无动于衷，来做一些优化吧。

不难看出，每一次读取图片都伴随着一次 MySQL 数据库访问（步骤 4），它的目的是获得文件在存储节点上的实际 URL，那么，我们完全可以将这个 URL 缓存起来，比如放到 Memcache 中，但缓存时间尽量短一些，比如 1 分钟，否则会失去追踪器负载均衡和可用性检测的意义。

对于步骤 5，你是否也考虑在 Web 服务器上对图片内容进行缓存呢？这样可以减少 loadphoto.php 每次读取图片内容的开销，不过，有一个更好的办法是在前端使用支持 reproxy 的反向代理服务器，如图 15-2 所示。

你可以使用 Perlbal 来作为反向代理服务器，它支持一个非常有意思的特性，那就是 reproxy，或者称为内部重定向，这使得后端服务器可以在 HTTP 响应头信息中指定一个新的 URL，告诉反向代理服务器重新请求这个地址。

在刚才的例子中，步骤 5 将被改变，Web 服务器不需要直接获取图片内容，而是将图片 URL 放到 HTTP 响应头中返回即可。

```
    <?php
    header("X-Reproxy-Url: http://10.0.1.203:7500/dev10/0/000/000/0000000586.fid");
    ?>
```

接下来的工作就交给反向代理服务器了，这样一来，Web 服务器的开销减少了，这也意味着整体的吞吐率将会得到提高。

但是，无论如何，通过应用程序来加载静态图片，单从性能的角度来看，似乎并不明智，而采用完全静态化访问和文件分发可能是更好的方式，它可以最大程度地发挥单台服务器的价值。

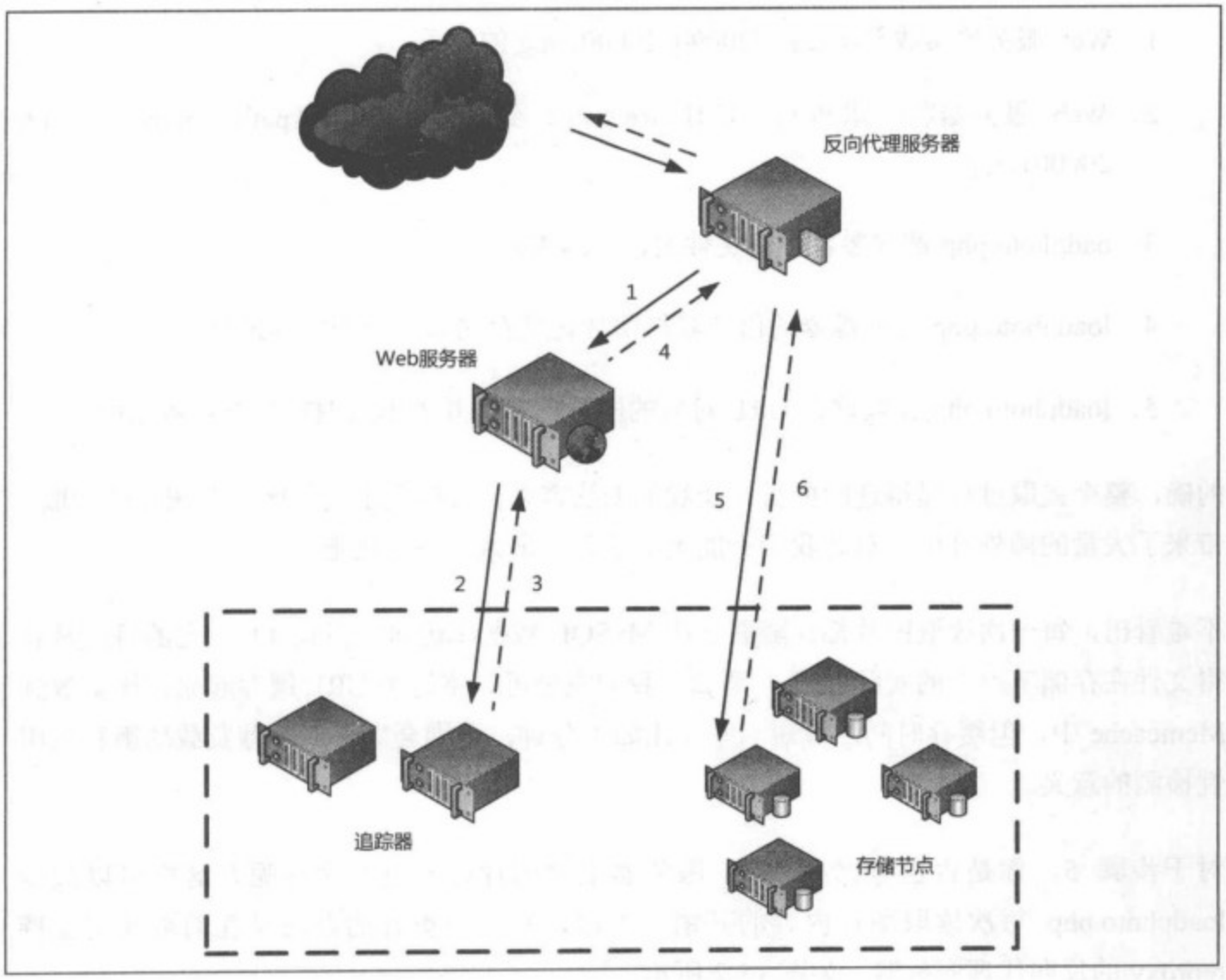


图 15-2　通过反向代理服务器来访问 MogileFS 的文件

也许你会感到疑惑，为什么分布式文件系统要采用特定的组织结构来存储文件呢？为什么不直接按照文件的原始路径进行存储并复制呢？这样便可以直接通过 Web 服务器进行静态化访问，从而大大提高性能。其实，你想要的顶多是一个文件分发系统，而不是分布式文件系统。

从另一个角度来看，分布式文件系统帮助我们完成了更多的事情，它的意义在于：

- 可以组建包含大量廉价服务器的海量存储系统，这是简单的文件分发和同步不容易做到的；
- 通过内部的冗余复制，保证文件的可用性，特别是在上面提到的海量存储系统中，容错能力是至关重要的；
- 拥有非常好的可扩展性，增加存储节点和追踪器都非常容易；
- 实现多个文件副本之间的负载均衡，完全可以通过扩展来保证性能。

为了达到这些目的，分布式文件系统必须设计成现在这样，所以，如果你更加注重扩展性和可用性，那么你可以通过更多的服务器来保证性能。

另一方面，分布式文件系统更加适合于存储一些不直接通过 HTTP 访问的文件，比如用于特定计算的索引文件、原始数据等，同时结合 Map/Reduce 等分布式计算机制，将更好地延伸分布式文件系统的作用。

当然，MogileFS 也只是分布式文件系统的一个具体实现，即便它的性能表现不尽如人意，或者存在或多或少的局限性，但这都不影响我们对分布式文件系统的了解和认识，大多数真正意义上的分布式文件系统都拥有相似的结构和框架，你可以根据站点的需要来灵活选择适合的产品。

第 16 章

数据库扩展

当我们对 Web 计算和存储都进行了不同程度的扩展后，站点的规模不断膨胀，这给数据库带来了巨大的查询压力，尽管前面我们介绍了数据库性能优化的一些具体做法，但是，这些显然是不够的，数据库也必须与时俱进，进行扩展。

在解决性能问题的同时，数据库的扩展还带来了一些其他的作用，比如增加存储空间、提高可用性等。接下来我们会介绍数据库扩展的一些思路，它们基本上覆盖了大多数扩展方式。

16.1 复制和分离

正如前面我们将文件复制到多台服务器一样，数据库通过复制来创建冗余副本，同样可以达到分散查询压力的目的。

事实上，当你掌握了前面章节的所有内容后，这一章的内容本质上对你来说并不陌生，比如负载均衡、复制、分离等思路，它们都在前面或多或少地出现过。

主从复制

几乎所有的主流数据库产品都支持复制，这是它们进行简单扩展的基本手段，我们以 MySQL 为例，它支持主从复制，配置并不复杂，简单地说，你只需要做到以下两点：

- 开启主服务器上的二进制日志（log-bin）。
- 在主服务器和从服务器上分别进行简单的配置和授权。

我们知道，MySQL 的主从复制是依据主服务器的二进制日志进行的，也就是说主服务器日志中记录的操作会在从服务器上进行重放，从而实现复制，所以主服务器必须开启二进制日志，它会自动记录所有对数据库产生更新的操作，也包括潜在的更新操作，比如没有删除任何实际记录的 DELETE 操作。

显然，这种复制是异步进行的，从服务器定时向主服务器请求最新日志，而主服务器只需要通过一个 I/O 线程来读取本地二进制日志，并输送给从服务器即可，所以，复制过程对

于主服务器的影响非常有限。但是，当存在多个从服务器同时从一个主服务器进行复制的时候，主服务器的磁盘压力会有不同程度的增长，这也并不是无法解决的，你可以采用多级复制策略，就像前面提到的多级分发一样。

在主服务器上对于所有更新操作记录二进制日志，这部分额外开销对性能大概有 1%的影响，与复制的意义相比，这几乎是微不足道的。

读写分离

一旦我们将数据库复制到更多的服务器上后，关键的问题来了，我们应该怎样合理地给它们分配工作量呢？

这里有一点需要注意，对于所有的更新操作，我们必须让它作用于主服务器上，这样才能保证所有数据库服务器上的数据一致。这也是任何使用单向复制机制的系统必须遵循的更新原则，比如在前面提到的文件分发和同步中，我们也只能对文件源进行更新。

为此，我们采用读写分离（R/W Splitting）的方法将应用程序中对数据库的写操作指向主服务器，而将读操作指向从服务器，如图 16-1 所示。

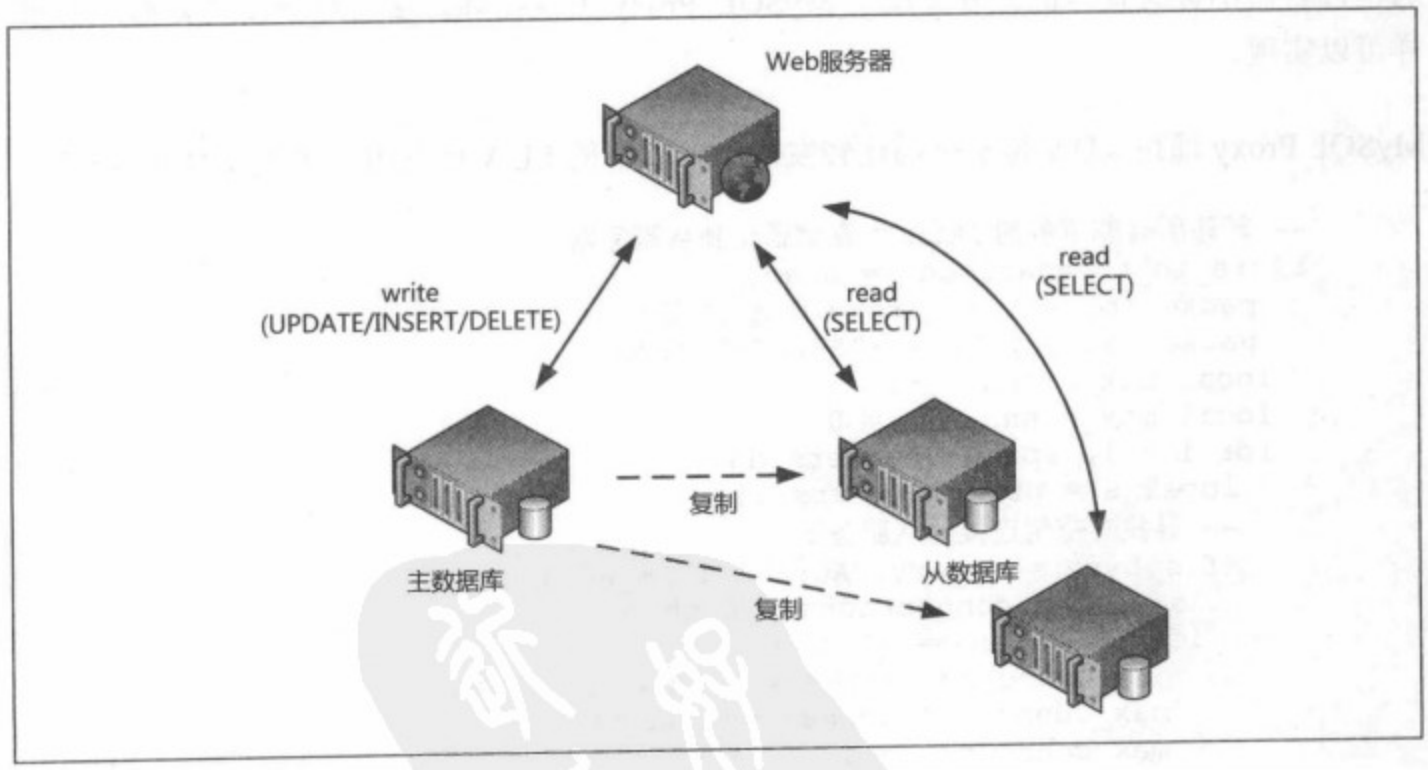


图 16-1　通过数据库主从复制实现读写分离

这样做不仅保证了多台服务器的数据一致性，更重要的是，它符合情理，一般而言，大多数站点的数据库读操作要比写操作更加密集，而且查询条件相对复杂，你可以通过 mysql-report 来查看 SELECT 操作的比例，通过前面对数据库性能优化的介绍，我们知道大部分的开销都消耗在这些查询上，而通过读写分离，我们可以将大量的读操作剥离出来，

转移到更多的从服务器上实现水平扩展，这是一个非常简单而有效的策略。

当然，对于以用户创造内容为主的站点来说，数据库写操作依然密集，随后我们会介绍通过分区来实现扩展的思路。

一旦决定采用读写分离，我们必须在应用程序中配置多个数据库连接选项，并且修改数据访问层代码，的确，应用程序知道应该如何区分读写操作，然而，对于将读操作分散到多台从服务器上，应用程序就不那么擅长了，简单的随机分配可能会造成多台从服务器的工作量不均衡，更重要的是，当某台从服务器发生故障后，应用程序并不知道。当然，你也可以不修改应用程序，而将这部分工作交给数据库反向代理，我们来看看它的作用。

使用数据库反向代理

如果你在使用 MySQL，那么可以尝试 MySQL Proxy，它工作在应用程序和 MySQL 服务器之间，负责所有请求和响应数据的转发。

就像 Web 反向代理服务器一样，MySQL Proxy 同样可以在 SQL 语句转发到后端的 MySQL 服务器之前对它进行修改，比如将所有对数据库 A 更新的 SQL 语句修改为更新数据库 B，这也许有些不合时宜，但充分体现了 MySQL Proxy 大权在握，那么对于读写分离，它同样可以实现。

MySQL Proxy 通过 LUA 脚本来描述转发规则，以下的 LUA 代码用于实现读写分离：

```
-- 转移所有非事务的 SELECT 查询语句到从服务器
if is_in_transaction == 0 and
   packet:byte() == proxy.COM_QUERY and
   packet:sub(2, 7) == "SELECT" then
  local max_conns = -1
  local max_conns_ndx = 0
  for i = 1, #proxy.servers do
    local s = proxy.servers[i]
    -- 寻找有空闲连接的从服务器
    if s.type == proxy.BACKEND_TYPE_RO and
       s.idling_connections > 0 then
      if max_conns == -1 or
         s.connected_clients < max_conns then
        max_conns = s.connected_clients
        max_conns_ndx = i
      end
    end
  end
  -- 找到一个有空闲连接的从服务器
  if max_conns_ndx > 0 then
    proxy.connection.backend_ndx = max_conns_ndx
  end
else
  -- 转发到主服务器
```

```
    end
    return proxy.PROXY_SEND_QUERY
```

MySQL Proxy 默认监听在 4040 端口，我们可以在启动时通过选项参数来指定后端的 MySQL 服务器：

```
mysql-proxy \
--proxy-read-only-backend-addresses=10.0.1.202:3306 \
--proxy-backend-addresses=10.0.1.201:3306 \
--proxy-lua-script=/etc/mysql-proxy/rw-splitting.lua &
```

如图 16-2 所示，我们看到应用程序只需要跟 MySQL Proxy 通信即可，而读写分离的工作都由 MySQL Proxy 来完成，与此同时，MySQL Proxy 还对多个从服务器实现负载均衡以及可用性检测，这些工作由它来做，的确非常适合。

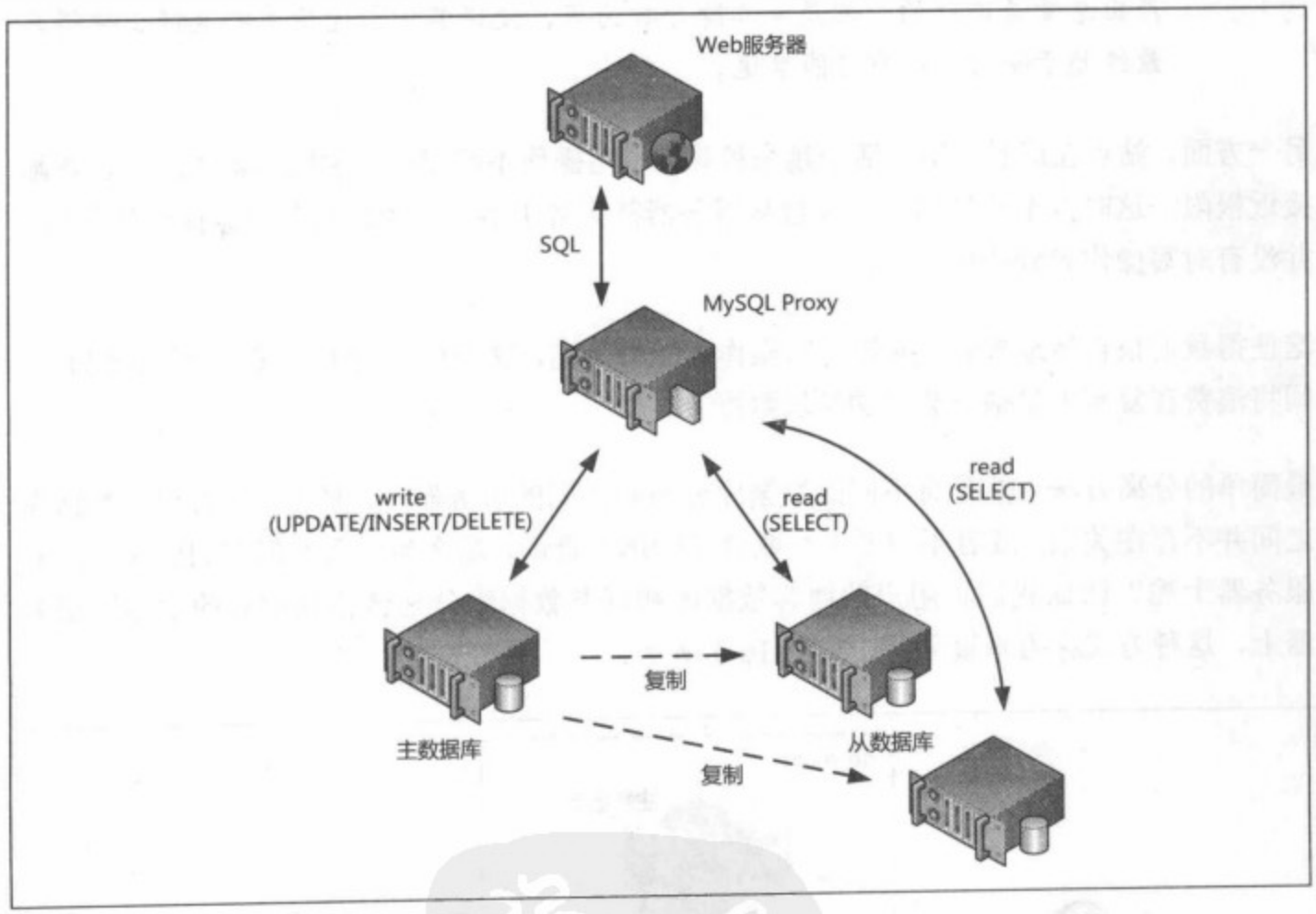


图 16-2　通过 MySQL Proxy 实现读写分离

但是，当存在大量的从服务器时，MySQL Proxy 必然会出现瓶颈效应，不过，即便是到那个时候，主服务器已经无法承受写操作的压力了，图 16-2 中的结构将不可避免地再次调整，接下来怎么办呢？

16.2 垂直分区

对于数据库写操作频繁（write-heavy）的站点来说，仅仅采用主从复制和读写分离可能效果并不明显，假如你的主服务器花费了 80%的时间在写数据，那么所有的从服务器也将花费更多的时间来同步数据，可想而知，所有从服务器只能依靠剩余不到 20%的时间来处理你的 SELECT 查询请求，这时候，增加从服务器所获得的回报将越来越少，呈现边际效益递减。

> **提示：**
>
> 边际效益递减是经济学的一个基本概念，它说的是在一个以资源作为投入的企业中，单位资源投入对产品产出的效用是不断递减的，换句话说，就是虽然其产出总量是递增的，但是其二阶导数为负，使得其增长速度不断变慢，使得其最终趋于峰值，并有可能衰退。

另一方面，站点在成长，用户活动越来越频繁，写操作不断增多，主服务器的压力也逐渐接近极限，这时候不论你增加多少台从服务器都无济于事，因为那只是对读操作的分散，并没有对写操作起到任何作用。

这使得我们很自然地想到，应该对写操作也进行分离，这个问题刻不容缓，否则我们的时间将浪费在复制大量毫无意义的垃圾数据上。

最简单的分离方法当然是将不同的数据库分布到不同的服务器上，你会发现有很多数据库之间并不存在关系，或者不需要进行联合（JOIN）查询，那么为什么不把它们放在不同的服务器上呢？比如我们将用户的博客数据库和好友数据库分别转移到独立的数据库服务器上，这种方式称为垂直分区，如图 16-3 所示。

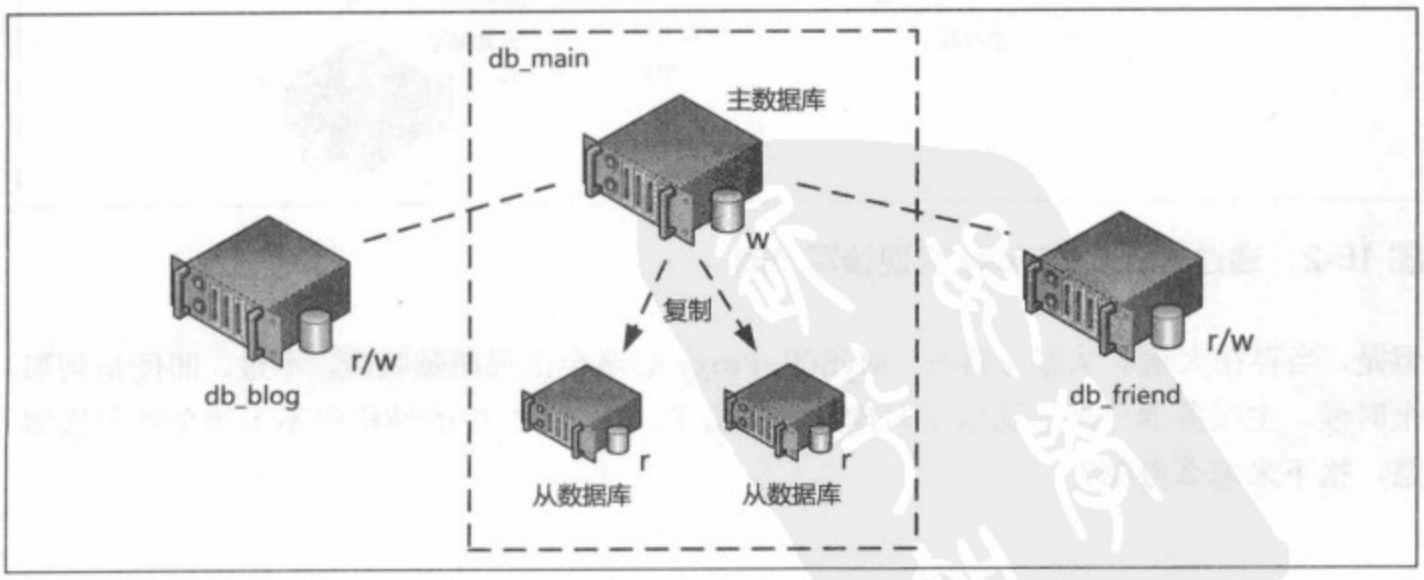


图 16-3　简单的垂直分区

可以看到，我们将 db_blog 和 db_friend 两个数据库分别转移到了独立的服务器上，而 db_main 则代表仍然存放在主服务器上的其他一系列数据库，你仍然可以在这里进行必要的联合查询，但是，不论从扩展的角度，还是从查询性能的角度来看，在进行数据库模型设计以及编写应用程序的时候，都应该尽量减少使用联合查询。

经过这样的垂直分区后，所有对于 db_blog 和 db_friend 的读写操作都将被分散到其他服务器上，如果它们带走了 60%的工作量，那么值得庆贺。

按照这样的思路，一旦我们需要为站点开发新的应用，便可以通过增加新的数据库服务器来实现扩展。

同样，我们还可以再次通过主从复制来对 db_blog 和 db_friend 两个数据库进行读写分离，如图 16-4 所示。

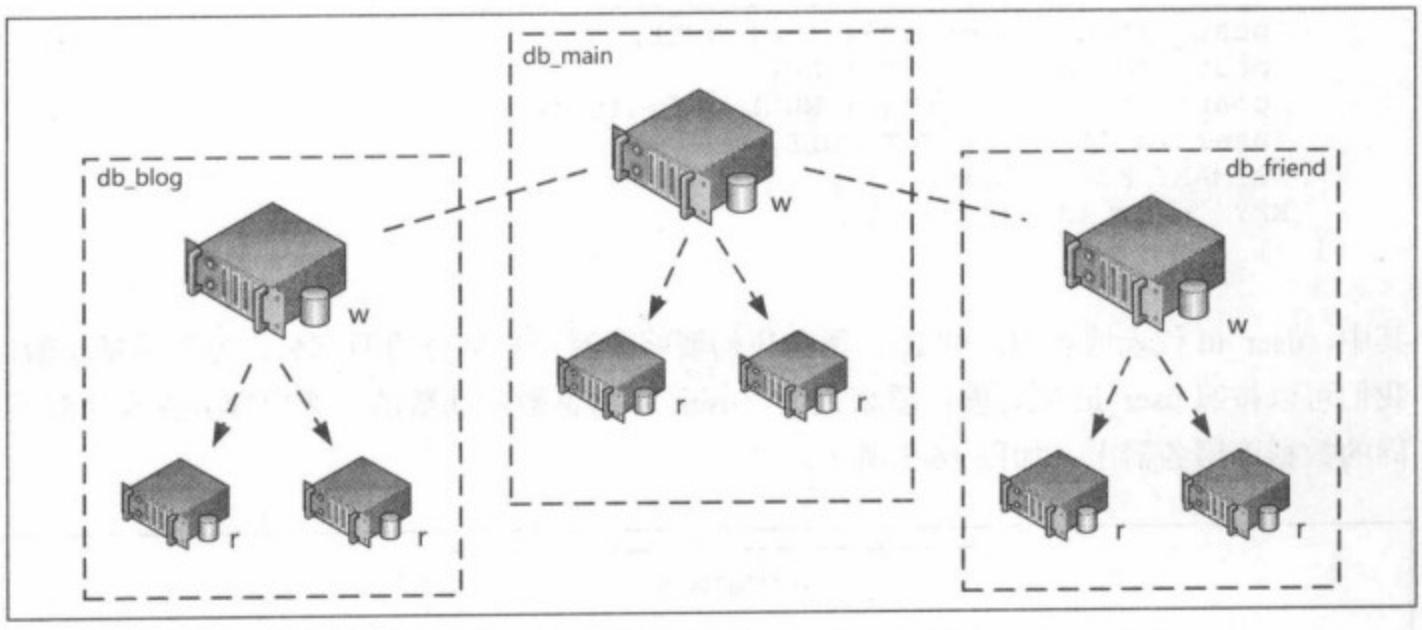


图 16-4　对多个分区分别进行读写分离

这的确是个不错的主意，然而，同样的问题在不久的未来仍然会困扰我们，当 db_blog 数据库的主服务器再次无法承受写操作压力时，我们又该如何呢？对这个数据库再次进行垂直分区吗？也许你可以将不同的数据表转移到不同的服务器上，但是当数据表也达到写操作极限的时候呢？似乎这种扩展方式要山穷水尽了，那么，我们换一种思路，来看看水平分区。

16.3　水平分区

水平分区（Sharding）意味着我们可以将同一数据表中的记录通过特定的算法进行分离，分别保存在不同的数据表中，从而可以部署在不同的数据库服务器上。

事实上，很多大规模的站点基本上都经历了从简单主从复制到垂直分区，再到水平分区的

步骤，这是一个必然的成长过程。下面我们来看看如何实现水平分区，值得一提的是，水平分区并不依赖于特定的技术，它更多的是一种逻辑层面的规划，需要一定的经验和不断的分析。

把数据放在不同分区中

继续前面的例子，我们希望将 db_blog 数据库中的数据拆分到不同的服务器上，并且应用程序能够知道如何找到它们。

db_blog 数据库中存储了用户发表的博客内容，主要数据都在 tbl_posts 表中，它的结构如下所示：

```
CREATE TABLE `tbl_posts` (
  `post_id` int(11) NOT NULL auto_increment,
  `post_title` varchar(64) NOT NULL,
  `post_content` text NOT NULL,
  `post_time` int(11) NOT NULL default '0',
  `user_id` int(11) NOT NULL,
  PRIMARY KEY  (`post_id`),
  KEY `user_id` (`user_id`)
)
```

其中，user_id 代表博客用户的 ID，现在我们要将所有用户划分为两部分，为了尽量均衡，我们可以按照 user_id 的奇偶性质来划分，user_id 为奇数和偶数的博客内容分别存储到不同的数据库服务器上，如图 16-5 所示。

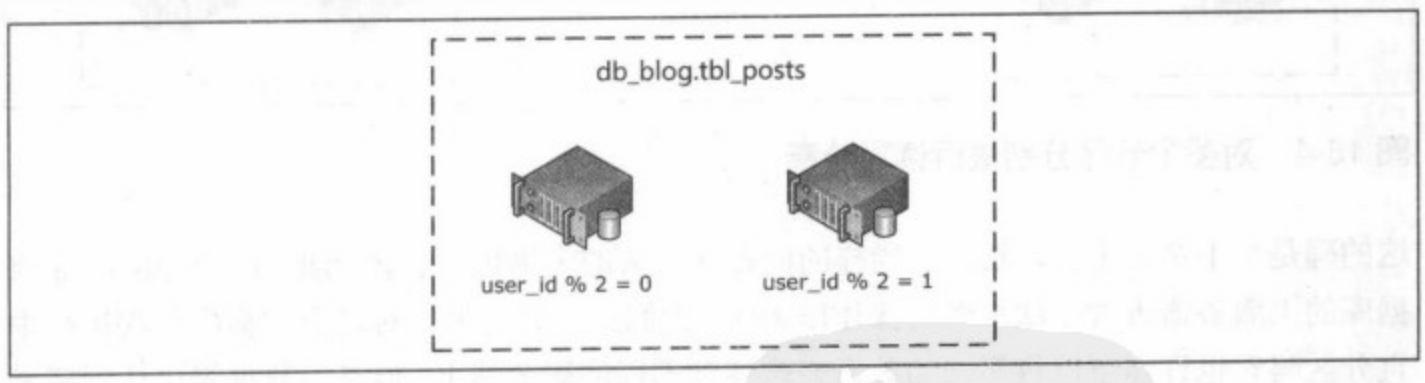


图 16-5　按照 user_id 奇偶性质进行水平分区

如此一来，我们建立了两个水平分区，它们分别位于图 16-5 中的两台服务器上，显然，只要我们知道博客内容的作者 ID，便可以知道应该去哪个分区查询内容。

同时，我们也要修改应用程序的数据访问代码。在分区之前，用户访问某篇博客内容时，可能使用以下的 URL：

```
http://www.highperfweb.com/blog_post.php?post_id=342352
```

通过 URL 中指定的 post_id，我们可以轻松地找到内容，代码如下所示：

```
<?php
$db = new DataAccess();
$db->selectDb("db_blog");
$sql = "select * from tbl_posts where post_id=" . $post_id;
$result = $db->query($sql);
?>
```

而采用水平分区后，刚才的 URL 显然不能正常工作了，因为数据分散在两台数据库服务器上，应用程序并不知道应该连接到哪一台服务器，我们必须告诉它，所以，我们提供了新的 URL：

```
http://www.highperfweb.com/blog_post.php?post_id=342352&user_id=10032
```

我们在 URL 中增加了 user_id，同时，修改后的数据访问代码如下所示：

```
<?php
$db = new DataAccess($user_id);
$db->selectDb("db_blog");
$sql = "select * from tbl_posts where post_id=" . $post_id;
$result = $db->query($sql);
?>
```

可以看到，在创建数据访问实例时，我们传入了 user_id 变量，它将引导应用程序连接到正确的数据库服务器。

在以上这个分区的例子中，我们实际上是对数据按照 user_id%2 的不同结果进行划分，同样，我们也可以通过 user_id%10 将 tbl_posts 的记录划分为 10 个分区，如图 16-6 所示。

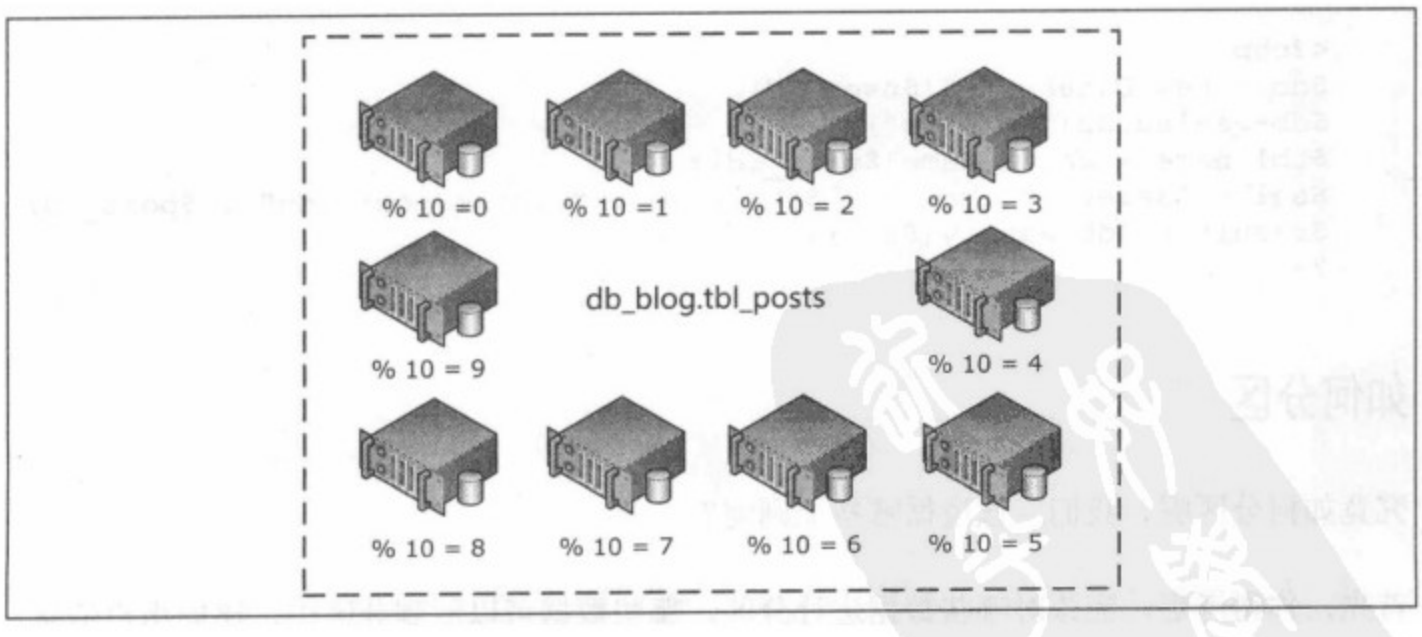


图 16-6　按照 user_id%10 进行水平分区

分区和分表

事实上，在考虑分区之前，我们一般会对数据库进行分表，它们的思路都是相同的，比如对于刚才的 tbl_posts 表，我们仍然按照 user_id%10 将它分为 10 个数据表：

```
tbl_posts_00
tbl_posts_01
tbl_posts_02
tbl_posts_03
tbl_posts_04
tbl_posts_05
tbl_posts_06
tbl_posts_07
tbl_posts_08
tbl_posts_09
```

这种分表的策略甚至会在数据库建立时便采用，因为我们希望数据表的记录数保持在相对较少的数量，这有利于减少查询时间，从而为数据库减少不必要的开销。

当然，分表只是单台数据库的优化策略，一旦到了必须考虑扩展的时候，分区便派上用场，不过，已经实现的分表将使得分区更加容易，因为数据已经是分离的，只需要迁移到其他服务器即可。

这时候，分表算法和分区算法可能不一致，比如我们希望将这 10 个表分布在两台服务器上，那么我们要在应用程序中维护一份映射关系表，比如将前 5 个数据表分配到一台数据库服务器，后 5 个数据表分配到另一台数据库服务器，这时的数据访问代码如下所示：

```
<?php
$db = new DataAccess($user_id);
$db->selectDb("db_blog");
$tbl_name = getTblName($user_id);
$sql = "select * from " . $tbl_name . " where post_id=" . $post_id;
$result = $db->query($sql);
?>
```

如何分区

究竟如何分区呢？我们应该遵循哪些原则呢？

首先，你得考虑，应该对哪些数据进行分区，哪些数据可以放到分区中，比如用户信息、好友关系，它们是否适合进行分区呢？其实，这个问题很难回答，但在大多数时候，对哪些数据进行分区并不是我们可以选择的，对于那些频繁访问而导致站点接近崩溃的热点数据，我们没有理由不对它们考虑分区，甚至在一开始就要考虑。

当然，在我们不得不对一些数据进行分区的时候，也意味着我们将失去一些操作它们的能力，比如原本你可以通过一条联合查询语句就能轻松搞定的任务，在分区后，你必须先通过用户 ID 找到正确的分区，然后查到对应的好友 ID，再通过好友 ID 找到对应的分区，从而找到好友信息。但是，你也不必感到沮丧，曾经的联合查询将会随着站点规则的增大变得越来越昂贵，而分区后的查询方式将会更加容易保持相对稳定的开销。

一旦我们知道要对哪些数据实施分区后，接下来就得找到一个用于分区的字段，我们称为分区索引字段，比如前面的 user_id，它必须和所有的记录都存在关系，一般我们会用被分区数据的主键或者外键。当使用主键时，你得保证它不能使用 auto_increment 自增类型。这一步非常简单，没有一概而论的方法，你要做的是抛开技术，认真地想一想你要什么，不久你便会得到结果。

接下来，基于这个字段，你得考虑采用什么分区算法，我们希望它可以带来良好的可扩展性，并且让各个分区的工作量相对均衡，通常有以下几种常用的分区算法。

哈希算法

刚才我们通过 user_id%10 来实现分区便是这种算法，它非常容易实现，而且应用程序通过 user_id 找到分区只需要进行简单的计算，几乎不存在额外开销。

相对于其他算法，哈希算法可以为多个分区比较均衡地分配工作量，特别是当记录数量级较多时，各个分区更加趋近于均衡。但是，这种算法对于扩展并不友好，一旦我们需要从 10 个分区扩展到 20 个分区，这便涉及所有数据的重新分区，你不得不暂停站点，等待漫长的计算。

范围

这种算法是指按照分区索引字段的范围进行分区，比如我们可以将 user_id 为 1~10000 的记录存储在一个分区中，而将 10001~20000 的用户存储在另一个分区中，以此类推。这使得应用程序需要维护一个简单的范围映射表，比如根据 user_id 来计算所属分区。

显然，它可以带来很好的扩展性，随着用户数量的不断增长，我们可以创建更多的分区。但是，各个分区的工作量会存在较大的差异，比如老用户所在的分区压力相对较大，或者一部分 ID 比较接近的热点用户导致所在分区压力过大。

映射关系

这种算法将对分区索引字段的每个可能的结果创建一个分区映射关系，这个映射关系将会非常庞大，应用程序已经无法通过简单的逻辑或者配置文件来维护它，而需要将它也写入

数据库，比如当应用程序需要知道 user_id 为 10 的用户的博客内容在哪个分区时，它必须查询数据库获得答案，当然，我们会使用缓存来提高性能。

由于这种方式详细保存了每一个记录的分区对应关系，所以各个分区具有较强的可伸缩性，我们可以灵活地控制它们的规模，并且轻松地将数据从一个分区迁移到另一个分区，这也使得各个分区可以通过灵活的动态调节来保持平衡。

分区扩展

对于刚才提到的几种分区算法，可扩展性是我们比较关心的，从整体上看，能否很好地扩展意味着是否可以保持性能或者带来更好的性能。

在这里，分区的可扩展性更多体现在能否快速平滑地实现扩展，并且进行最少单位的数据移动。

通过 Web 负载均衡的体验，我们发现，计算能力是一种内在的、无形的力量，它的扩展本身几乎不需要时间，而数据库水平分区的扩展则意味着必要的数据迁移，以及重建平衡，这种流动带来了时间开销。

的确，不论采用哪种分区策略，增加分区都是一件不小的事情，大多数站点都会选择在夜深人静的时候挂上暂停维护的公告，当然，一旦持续时间较长，用户便会开始抱怨。

不幸的是，没有太多通用的方法帮助你很好地解决分区扩展，因为没有人知道你的站点拥有什么样的数据，并且将会以何种速度成长，但我们可以做的是，根据站点的实际情况，选择适合的分区策略，并且尽早地制定合理的扩展规划。

分区反向代理

还记得前面提到的 MySQL Proxy 吗？它帮助应用程序实现了读写分离，而在这里，另一个开源产品 Spock Proxy 也起到了类似的作用，它可以帮助应用程序实现水平分区的访问调度，这意味着我们不需要在应用程序中维护那些分区对应关系了。

Spock Proxy 本身的大部分代码正是基于 MySQL Proxy，同时它也进行了一些改进，在这里，Spock Proxy 的作用如图 16-7 所示。关于它的详细介绍，你可以查看在线文档。

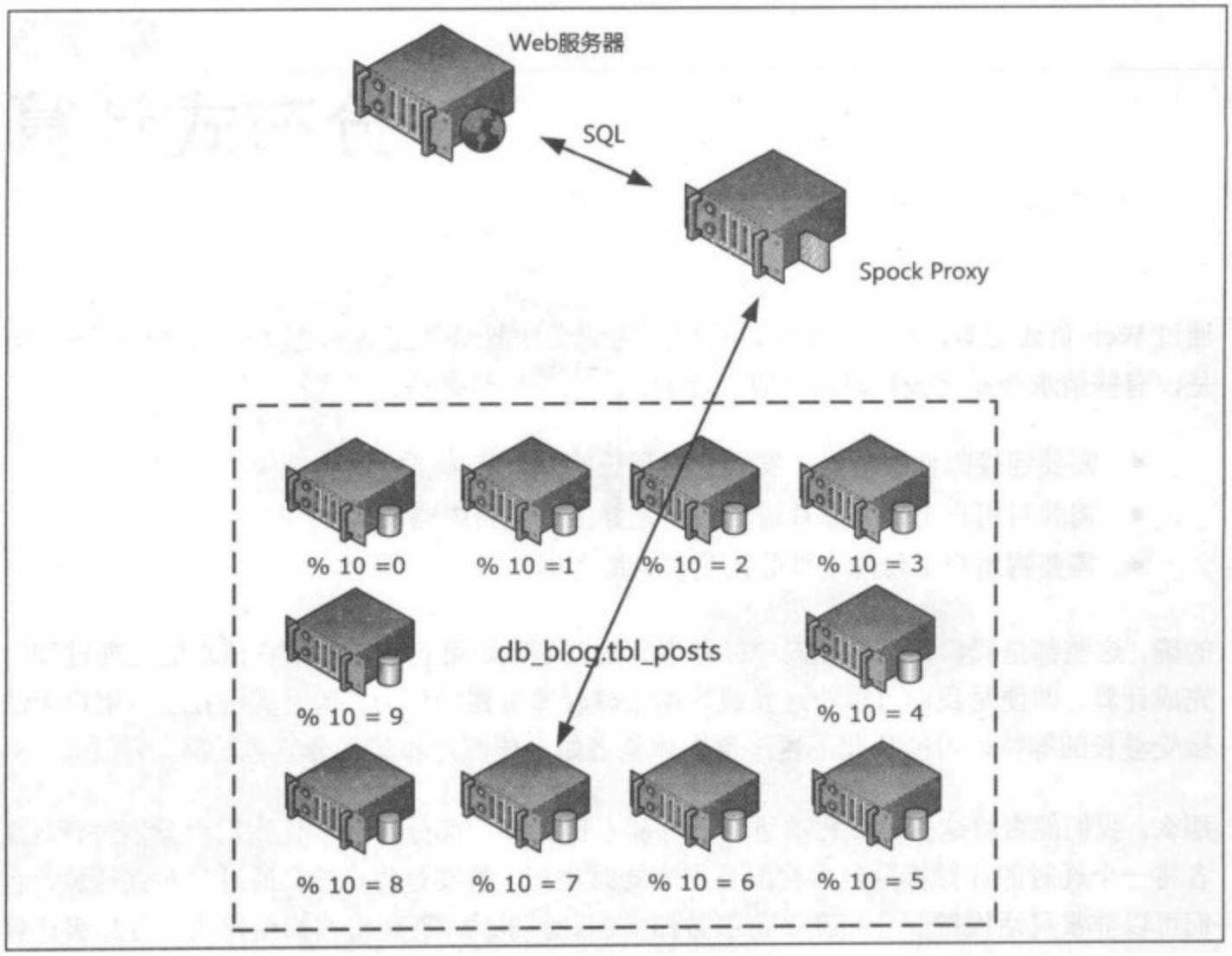


图 16-7　通过 Spock Proxy 访问多个 MySQL 水平分区

第 17 章

分布式计算

通过 Web 负载均衡，我们已经可以将用户的请求分散到多台 Web 服务器上进行处理，但是，有些请求涉及比较耗时的计算，比如：

- 需要连接邮件服务器，发送一封超长的 HTML 邮件。
- 需要对用户上传的照片进行裁剪，并生成多份缩略图。
- 需要将用户上传的文件分发到多台服务器上。

的确，这些都是我们在站点开发中经常遇到的问题，如果它们都在用户请求的处理过程中完成计算，即便是我们可以通过负载均衡来减轻服务器的压力，但是关键在于，用户无法接受漫长的等待，可能从此不再愿意去体验诸如上传照片和邮件分享等功能。

那么，我们能否将这些计算再次进行拆分呢？比如将一部分计算转移到后台异步进行，或者将一个耗时的计算拆分到多台服务器上同时进行，答案是肯定的，通过分布式计算，我们可以非常灵活地控制分布在多台服务器上的计算能力，甚至将它们组建成一台超级计算机，当然，这可是一台超值的廉价超级计算机。

17.1 异步计算

对于刚刚提到的几个问题，异步计算正好可以派上用场，我们先来了解分布式消息队列。

分布式消息队列

当你去银行办理业务时，首先迎接你的可能是银行的咨询人员，他会根据你的需要，告诉你应该去哪个窗口排队，当然，你可以把这看作是一个比较理想的场景，现实中很多银行可能没有设置这样的咨询人员。

这个看似平常的过程，却正体现了分布式消息队列的思想，我们看，银行的咨询人员并没有带着你去办理业务，而是让你排在某个窗口队列的末尾，这意味你需要等待，会有其他工作人员为你办理业务，而咨询人员一旦将你引导至队列中后，便可以再次响应其他的客户。从某种意义上讲，这里的咨询人员实现了“异步计算”，的确，他将客户转交给了“后台”进行处理，而客户队列在这里发挥着重要的作用，也许你已经发现了，它不正是分布式消息队列的原型吗？

的确，这样的例子在生活中数不胜数，它们就在我们的身边。再比如，你一定看到过工厂的流水生产线，至少在电视里见过，流水线运载着源源不断的半成品，它们在缓慢移动，经过各种不同的处理器进行加工，这些处理器就像是在“计算”，从流水线中取得“输入”，并将加工后的结果“输出”到流水线的下一个环节，然后不断重复。这里的流水线自然也可以看做是分布式消息队列，而流水线上的那些产品便是消息了。

对于拥有先进先出（First In First Out，FIFO）机制的队列本身，你或许并不陌生，而在实际应用中，我们所说的分布式消息队列，实际上是指监听在服务器某个端口上的服务，也可以说是分布式消息队列服务，它可以维护并管理很多消息队列，应用程序可以通过网络快速地访问它，为某个队列追加消息或者从某个队列领取消息。除此之外，你几乎不用关心它的内部构造，而我们更加关心的是，如何将它融入我们的 Web 计算中，最大程度地发挥分布式消息队列的作用。

Gearman

Gearman 是一个开源产品，它的初衷是用来实现远程函数调用，这样一来，它便可以将计算转移到其他服务器上，而这一切都巧妙地隐藏在它提供的 API 中。同时，它的这种机制是跨语言的，这意味着你可以在 PHP 程序中调用另一台服务器上用 C++编写的程序中的某个函数。

如图 17-1 所示，Gearman 在远程调用中发挥作用，其中的任务服务器（Job Server）运行着 gearmand，它负责处理 Web 应用的远程调用请求，并且维护计算任务，而工作服务器（Worker Server）则负责从 Job Server 那里领取任务，并且执行实际的计算，然后将结果返回给 Job Server。

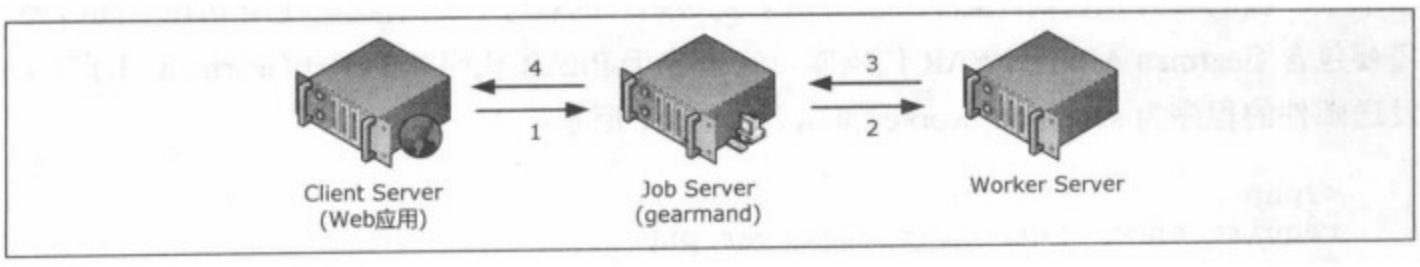


图 17-1　利用 Gearman 实现远程调用

当然，这里我们希望实现异步计算，这也是 Gearman 很容易做到的，我们可以使用异步模式，如图 17-2 所示。

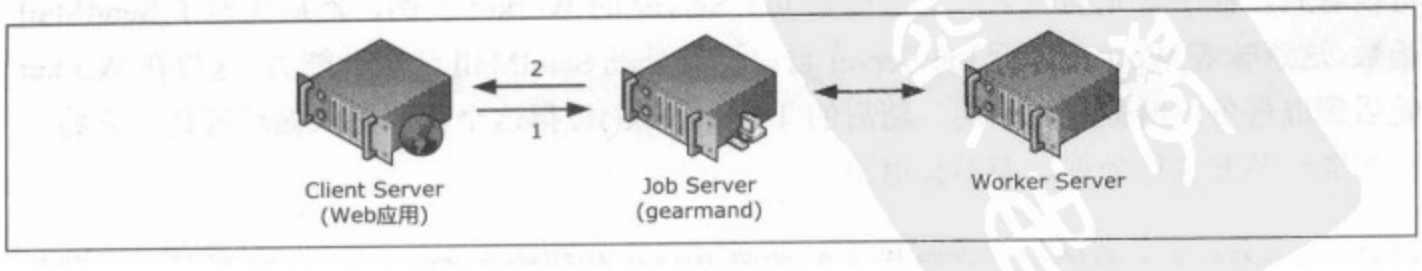


图 17-2　利用 Gearman 实现异步计算

在这里，运行 gearmand 的 Job Server 实际上便是某种意义上的分布式消息队列服务，当然，与随后要介绍的 MemcacheQ 相比，gearmand 做了更多的事情，它所维护的消息，实际上已经具备了某种更适合计算的任务形态，可以说，它几乎定义了整个异步计算的开发框架。

在接下来的例子中，我们将使用 Gearman 来实现异步计算。

还记得前面提到的关于邮件发送的问题吗？有时候，大量邮件阻塞在邮件服务器，你愿意让 Web 应用程序去等待吗？

在使用异步计算之前，当我们需要发送邮件时，会直接在 Web 应用程序中写入相应的代码，这里以 PHP 为例，如下所示：

```
require ("lib/MailSender.php");
$mailSender = new MailSender();
$mailSender->initMail();
$mailSender->send($mail_subject, $mail_body, $mail_to, $mail_to_nick);
```

代码非常简单，可是遗憾的是，它一旦运行起来，便可能会长期阻塞在 send 函数。为此，我们来借助 Gearman，看看它将如何改变我们的邮件发送方式。

首先，我们需要开启在 Job Server 上的 gearmand，它默认监听在 4730 端口。

```
s-mat:/usr/local/gearmand/bin # ./gearmand -v -d
Method for libevent: epoll
Listening on port 4730
```

接下来，我们要为 Worker Server 编写用于发送邮件的程序，这里我们使用 PHP，同时还需要包含 Gearman API 的 PEAR 代码库，它们位于 PEAR 代码库的 Net/Gearman 目录下。发送邮件的程序为 sendmail_worker.php，代码如下所示：

```
<?php
require_once 'Net/Gearman/Worker.php';
$servers = array('10.0.1.200:4730');
$worker = new Net_Gearman_Worker($servers);
$worker->addAbility('SendMail');
$worker->beginWork();
?>
```

可以看到，程序首先创建了一个连接到 Job Server 的 Worker 实例，然后注册了 SendMail 函数，这意味着 Worker 告诉 Job Server 自己拥有处理 SendMail 函数的能力，这将在 Worker 随后领取任务的时候起到作用。随后的 beginWork()使得这个程序进入循环等待，它将一直领取任务并进行处理，而不会退出。

可是，SendMail 函数的定义在哪里呢？按照 PEAR 代码库中 Gearman API 的设计，我们还需要创建一个定义该函数的文件，并放在以下位置：

/Net/Gearman/Job/SendMail.php

它的代码如下所示：

```
<?php
class Net_Gearman_Job_SendMail extends Net_Gearman_Job_Common
{
    public function run($arg)
    {
        require ("lib/MailSender.php");
        $mailSender = new MailSender();
        $mailSender->initMail();
        $mailSender->send($arg['mail_subject'],
            $arg['mail_body'],
            $arg['mail_to'],
            $arg['mail_to_nick']);
    }
}
?>
```

这样一来，Worker Server 便完全具备了发送邮件的能力，现在，我们可以将 Web 应用程序中发送邮件的部分代码修改为以下形式：

```
<?php
require_once 'Net/Gearman/Client.php';
$servers = array('10.0.1.200:4730');
$client = new Net_Gearman_Client($servers);
$client->SendMail(array(
    'mail_subject' => $mail_subject,
    'mail_body' => $mail_body,
    'mail_to' => $mail_to,
    'mail_to_nick' => $mail_to_nick));
?>
```

应用新的代码后，应用程序只需要将发送邮件的任务投递给 Job Server 便大功告成。

在以上的代码中，我们似乎没有看到设置异步模式的地方，其实，在 PEAR 代码库的 Gearman API 实现中，设置异步模式并没有暴露给应用程序，我们通过修改代码库中的 Net/Gearman/Client.php 即可实现，修改的部分如下所示：

```
public function __call($func, array $args = array())
{
    $send = "";
    if (isset($args[0]) && !empty($args[0])) {
        $send = $args[0];
    }
    $task       = new Net_Gearman_Task($func, $send);
    $task->type = Net_Gearman_Task::JOB_HIGH;
    $set = new Net_Gearman_Set();
    $set->addTask($task);
    $this->runSet($set);
    return $task->handle;
}
```

当然，这里我们使用的 Gearman API 只是基于 PHP PEAR 扩展的一种具体实现，值得一提的是，近期 Gearman 发布了基于 PHP PECL 的二进制扩展，你也可以去尝试，它将提供另一种风格的类库框架。另外，你也可以根据 Gearman 的开放协议自己设计 API，或者封装现有的 API，更加适合你的开发习惯。

由于采用了异步计算，应用程序也并不知道邮件是否发送成功，这怎么办呢？千万不要为了解决这个问题而放弃异步计算，这将让你前功尽弃，在异步计算的世界里，我们的思维方式必须改变，既然计算是异步的，那么反馈也应该是异步的，你完全可以让 SendMail 函数将发送结果写入数据库，并生成报表，然后让应用程序定期对报告中发送失败的邮件执行再次发送。

现在，再次回想 Web 应用程序、Job Server 和 Worker Server，在一些必要的情况下，它们也完全可以运行在同一台物理服务器上，虽然没有将计算转移到其他服务器，但是至少将计算从 Web 内容处理的进程中转移到了某个后台进程，这是意义重大的。

通过这样的异步计算，可以很好地解决前面我们提到的那些问题，除此之外，它还有更多的适用场景，比如视频转换、索引计算、静态内容生成、缓存清除等，随后我们会在介绍 MemcacheQ 的时候提到缓存清除的例子，总之，在 Gearman 的背后，我们应该看到异步计算的思想，它能发挥多大的作用，完全取决于你如何使用它。

另一方面，如图 17-3 所示，你也可以运行多个 Job Server，从而提高可用性，每个 Job Server 都注册了所有的 Worker Server，不论你将任务投递给谁，它们都可以出色地完成工作。

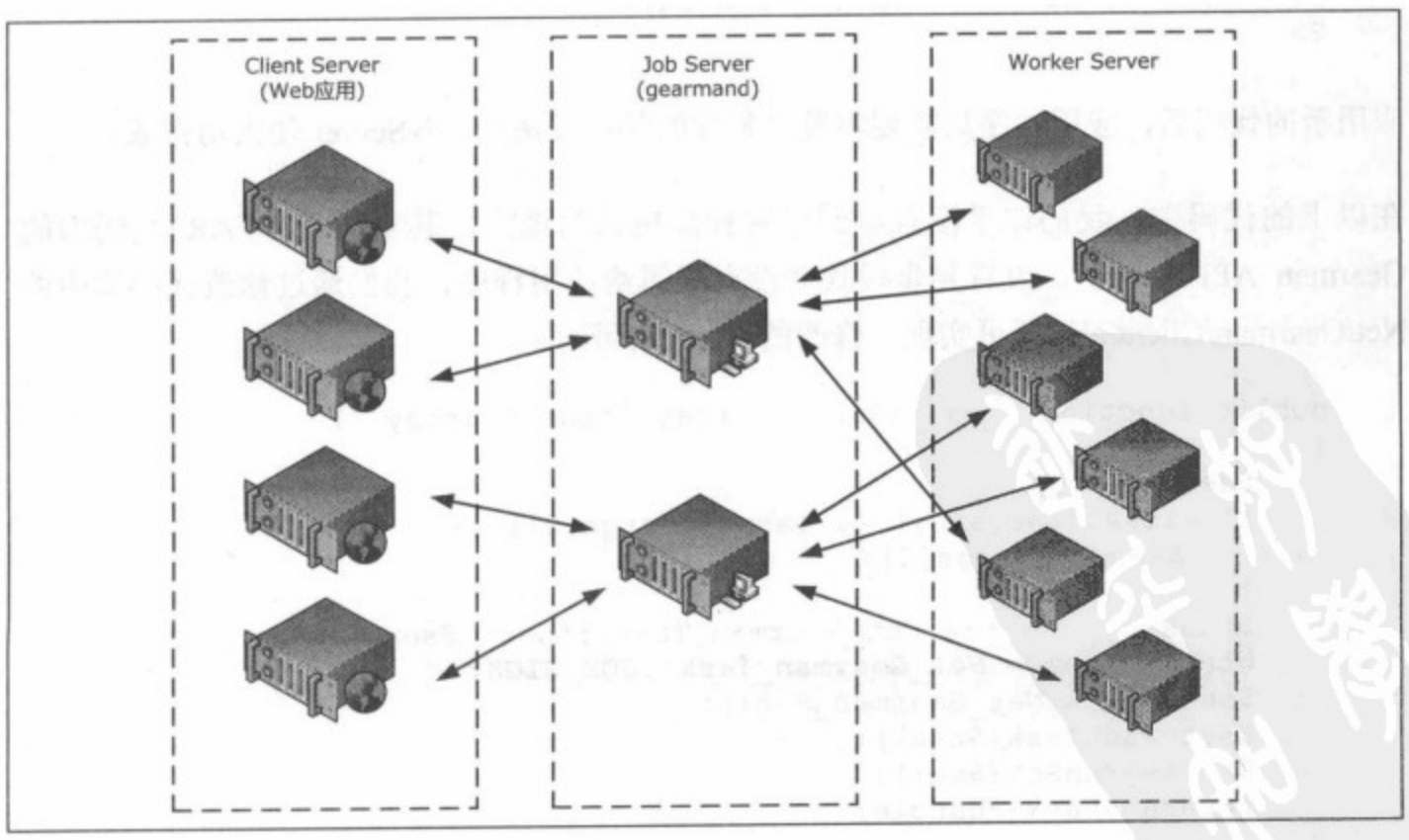


图 17-3　通过多个 Job Server 提高可用性

不过，你不需要准备太多的 Job Server，因为我们只是希望提高可用性，所以两台到三台即可，基本不用担心它的性能，因为相对于两端，它的开销是微乎其微的，通常它并不会成为性能瓶颈。

MemcacheQ

相比于 Gearman，MemcacheQ 是一个单纯的分布式消息队列服务，它同样也是开源产品，和前面介绍的 MemcacheDB 师出同门，当然，它同样基于 Memcache 访问协议。

正因为基于 Memcache 协议，所以你会非常容易地使用 MemcacheQ，你甚至完全不需要编辑新的 API 扩展，只需要使用 Memcache API 即可，比如以下的 PHP 代码完成了对 MemcacheQ 服务器的一系列操作：

```
<?php
$memcache_obj = memcache_connect('10.0.1.201', 21201);
memcache_set($memcache_obj, 'queue1', 'this is the first message', 0, 0);
memcache_set($memcache_obj, 'queue1', 'this is the second message', 0, 0);
$message = memcache_get($memcache_obj, 'queue1');
memcache_close($memcache_obj);
?>
```

显然，MemcacheQ 并没有像 Gearman 考虑得那么多，它把更多的工作留给了你，而这也恰恰让 MemcacheQ 变得更加简单、灵活，事实上，它们的定位完全不同，与此同时，你也完全可以用 MemcacheQ 来进一步实现 Gearman 那样的远程调用框架。

还记得前面有关 Varnish 清除缓存的介绍吗？Varnish 提供了基于 HTTP 的清除方式，可我们又不希望在 Web 应用中执行清除操作，这样可能会或多或少地阻塞 Web 处理进程，所以，我们可以通过 MemcacheQ 来实现异步清除缓存，这并不困难，有了前面异步发送邮件的例子，再加上 MemcacheQ 的友好接口，我想你可以很快实现它，这里我们就不详细说明了。

总之，在灵活驾驭异步计算之后，你也许会发现，从前很吃力办到的事情，现在都可以很自然地解决，你要做的就是将不希望在 Web 处理进程中计算的任务统统抛到其他的地方去。

17.2 并行计算

然而，对于分布式计算来说，异步计算才只是刚刚开始，一个现实的问题在于，即便我们将一个耗时的计算从 Web 服务进程中剥离并转移到其他的服务器上，但这仅仅是转移，并不能减少它的计算时间，而这个任务可能恰恰是我们需要快速得到结果的，这该怎么办呢？

分而治之

的确，我们必须要想办法对计算任务进行再次拆分，并将它们分散到更多的服务器上同时进行，这便是真正意义上的并行计算。

但是，如何拆分任务呢？多个任务分别计算后又怎么合并呢？带着这些问题，来看看发生在我们身边的事情。

高一年级的期末考试结束了，照例，老师们需要批改试卷，然后按照成绩排出名次。显然，一个老师完成这些工作很不现实，学生们等到下个学期开学也看不到成绩单，而所有老师一起来做，就得有个规则，大家将所有试卷分成若干份，平均分配给这些老师，这样一来，所有的老师可以同时批改试卷，互不影响，当大家都批改完后，将批改结果纷纷汇总，再按照所属班级进行归类，然后分别对各个班级的学生成绩进行排序。

这个过程几乎我们每个人都经历过，不过我们当时可能更关心的是自己的成绩单，而没有考虑过学校为了提高效率而进行的努力，但无论如何，这个过程正是非常恰当地体现了并行计算的思想，我们不仅看到了对试卷批改任务的拆分，更重要的是，最后的汇总排序一锤定音，整个过程从分到合，非常优雅、和谐。

可见，关键在于，我们能否对一个特定的问题进行有效的分解，当然，有些计算是不适合分解的，比如需要对全年级的学生成绩进行排序，那么这个计算本身已经不容易分解了，这可想而知，但这只是一个局部计算，从整个过程来看，用于排序的原始数据总是存在一定的分解计算过程。

对于各种不同的计算任务，如何拆分、拆分后如何计算、最后如何汇总都是应用程序需要考虑的事情，几乎很难设计出一套通用的、具体的并行计算方法，但是存在一定的并行计算框架，我们来看随后介绍的 Map/Reduce。

当然，这里我们谈的是基于分布式的并行计算，而不是多核处理器环境下的并行计算，对于后者，这是操作系统和编程语言需要考虑的，本书将它们视为垂直扩展的范畴。

Map/Reduce

正如前面提到的从分到合，它的思想是伟大的，Map/Reduce 正是对它进行了精辟的总结和抽象，它认为任何的计算任务都可以经历从拆分到汇总的两个过程，反过来，只需要用这两个过程就可以描述所有的计算任务，这两个过程分别为 Map 和 Reduce。

Map/Reduce 是一种分布式并行计算的开发框架，Google 用它来实现很多产品中海量数据的计算，它提供了一个简单的可扩展模型，同时隐藏了很多底层的技术细节，比如传输、

监控、容错、可用性、负载均衡等，这使得我们只需要考虑如何对计算进行拆分和汇总，以及编写具体的计算逻辑即可。

当然，Map/Reduce 只是 Google 内部的实现，整个系统我们是无法拿来使用的，但是它的思想是开放的，我们完全可以通过其他的方法或者开源产品来实现，比如 Hadoop 支持在分布式文件系统中实现 Map/Reduce 计算，另外，我们也完全可以通过前面介绍的 Gearman 来实现 Map/Reduce，其中 Job Server 仍然负责调度任务，而 Worker Server 主要负责两部分，一部分用于执行 Map 操作，而另一部分用于执行 Reduce 操作，当然，由 Client Server 负责对计算进行分解。

既然刚才提到的批改试卷的过程体现了 Map/Reduce 的思想，那我们这里就用 Map/Reduce 来描述这个过程，不过为了尽量简单直接描述，我们隐藏了 Gearman API 框架的多余部分，以及一些实现上的细节，使用接近 PHP 的伪代码，并且加入了中文的变量名，以便于读者理解。

从根本上说，Map 和 Reduce 都是需要我们编写的函数，我们来看看 Map 函数的定义：

```
function Map($学号, array($班级, $试卷内容))
{
    $成绩 = 批改试卷($试卷内容);
    保存(array($班级, array($学号, $成绩)));
}
```

这里的 Map 函数将会在多个 Worker Server 上运行，它用来计算每一个学生的试卷成绩，并将计算结果按照班级为索引写入本地的中间数据，便于随后按照班级进行汇总。

Map 函数的输入之所以是每个学生，是由 Client Server 进行分解的，我们来看看运行在 Client Server 上的应用程序是如何分解原始数据的，代码描述如下：

```
foreach ($所有试卷 as $学号 => array($班级, $试卷内容))
{
    Map($学号, array($班级, $试卷内容));
}
```

可以看到，它对所有的试卷进行遍历，将其中的每一份试卷都作为输入参数来调用 Map 函数，举个例子，比如传入 Map 函数的实际参数为：

```
012, (1, 试卷)
```

这代表着学号为 012、所在班级为 1 以及该学生的试卷内容，它们经过 Job Server 的调度，进入了某个 Worker Server 上的 Map 函数，经过计算后，结果如下所示：

```
1, (012, 96)
```

这意味着他的成绩为 96 分，同时，为了便于进行随后的归类，我们将班级作为返回数据的索引。

就这样，同时工作在大量 Worker Server 上的 Map 函数将所有的试卷都完成计算，并将保存在本地的临时数据不断上报回 Client Server，按照班级进行归类。事实上，Map 函数在本地保存中间数据的时候，已经是按照班级进行分区保存，所以在上报的时候，只需要将数据追加到不同班级的数据集中即可，并不需要很复杂的归类操作。

接下来，这些归类好的数据就要传入 Reduce 函数了，Reduce 函数同样运行在多个 Worker Server 上，这些 Worker Server 完全可以是刚才运行 Map 函数的那些 Worker Server，因为只有当所有的 Map 函数执行完毕后，Reduce 才会启动，这时候运行 Map 函数的 Worker Server 已经空闲，可以重复利用。

这里的 Reduce 函数定义代码如下：

```
function Reduce($班级, array(array($学号, $成绩),array($学号, $成绩),...))
{
    提交结果(按照成绩排序(array(array($学号, $成绩),array($学号, $成绩),...)));
}
```

传入到 Reduce 函数的参数如：

```
1, (012, 96), (096, 85))
```

这代表 1 班所有学生的成绩，它们将在 Reduce 函数中进行排序，从而生成 1 班的成绩单，然后上报回 Client Server 上的应用程序。

整个过程的数据流如图 17-4 所示，其中我们隐藏了控制层面的细节，实际上具体实现要复杂得多。

有一个问题在于，假如某个 Worker Server 上的 Map 操作领取了计算任务后，突然发生了故障，比如 I/O 缓慢或者 CPU 不可用等，这将导致它无法完成计算，必然会拖累整个计算过程，大家不能浪费时间来等待它，这属于掉队者问题，解决办法是，将同样的计算任务分配给多个 Map 操作，实现冗余计算，一旦获得结果后，终止其他计算。

另一方面，在并行计算的过程中，必然会涉及大量的数据移动，在刚才的例子中，主要的数据便是学生的试卷内容，往往在实际应用中，我们会对数据进行压缩传输，在一定程度上减少网络 I/O 开销，从 Google 的一份报告中可以看到，经过数据压缩后，1800 台 Worker Server 完成了 31Gbytes/s 原始数据的计算，而压缩之前，只能完成 10Gbytes/s 的计算。

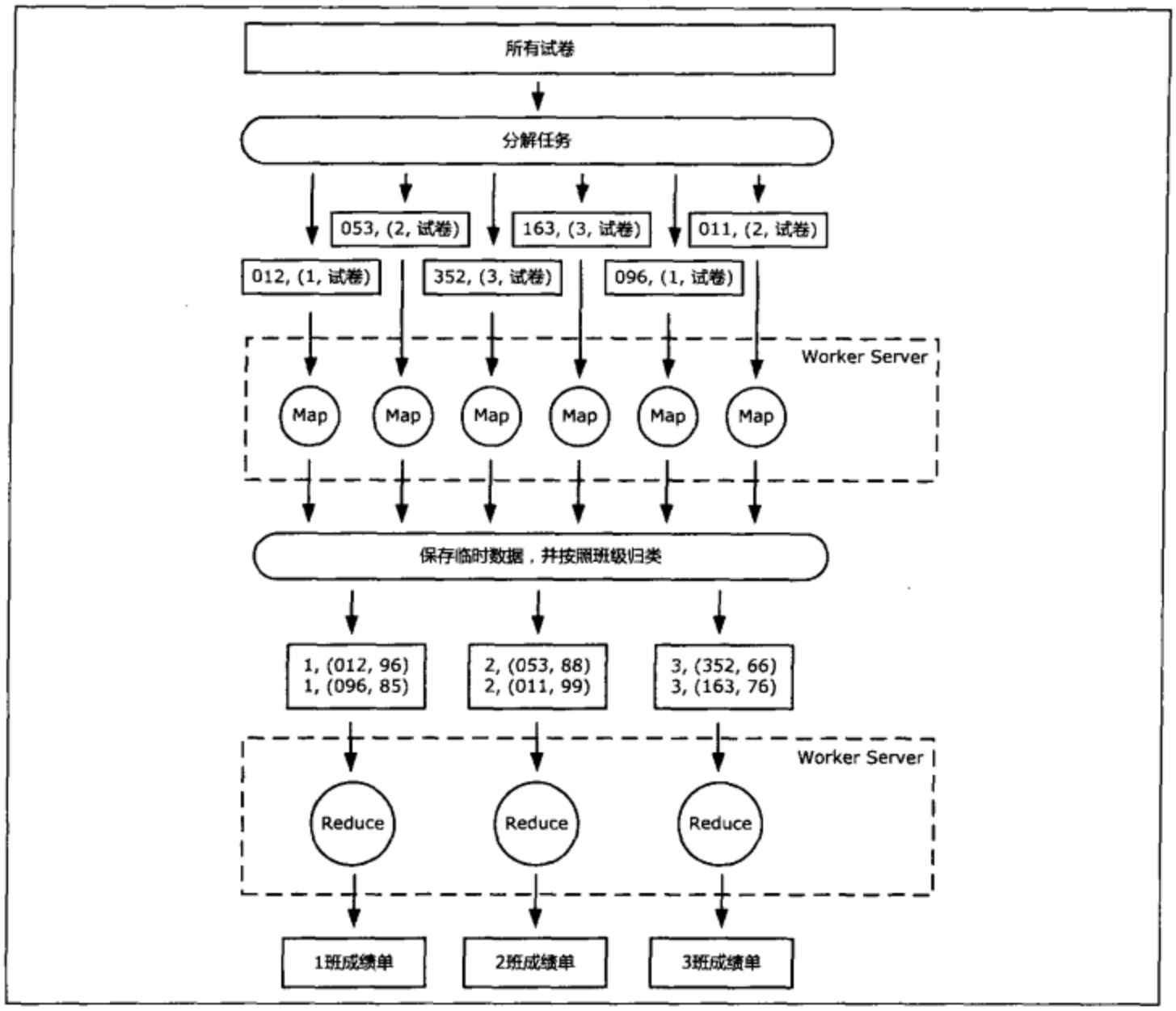


图 17-4　通过 Map/Reduce 来生成各班级成绩单

当然，你可以将分布式文件系统很好地与 Map/Reduce 结合使用，比如在刚才的例子中，我们通过分布式文件系统将所有试卷内容都事先复制到 Worker Server 上，这时候，Worker Server 也扮演了分布式文件系统的存储节点，它可以快速地直接从本地提取原始数据。

总之，在分布式计算环境下，Map/Reduce 是一种解决特定问题的策略，通过它，你甚至可以驾驭成百上千个 CPU 和几乎无限的扩展计算能力，但是，面对不同的应用和问题，你仍然面临着很多考验，这些需要依靠你的智慧和想象力来完成。

提示：

想象力比知识更重要，因为知识是有限的，而想象力概括着世界上的一切，推动着进步，并且是知识进化的源泉。严肃地说，想象力是科学研究中的实在因素。

——【美】爱因斯坦

第 18 章

性能监控

我们需要通过监控的手段来洞察站点性能的变化，它会带给你优化的理由，也会告诉你瓶颈的真相，更重要的是，它让你拥有敏锐的直觉，没有它，你可能根本无法知道你的站点是否健康，因为你不可能每天都去费力地阅读日志。

没有人会认为站点的性能一成不变，的确，它时时刻刻都在变化，各种各样的原因会导致站点不能保持我们预期的性能，不论我们做了多少努力，你总是不知道下一刻会发生什么，性能监控在某种程度上就像是晴雨表，它能反映一定的性能变化规律和趋势，所以，能够快速从监控数据和图表中找到线索是你必须具备的本领。

事实上，在前面的章节中我们已经或多或少地用到了一些监控数据和图表，可见，这些内容和性能分析是不可分割的，这一章我们将主要介绍一些提供性能监控的工具和系统，你可以通过它们快速搭建监控中心。

提示：

运筹帷幄之中，决胜于千里之外。

——刘邦

18.1 实时监控

Nmon 是一款工作在服务器本地的实时监控软件，它可以提供时间间隔为秒的系统监控，我们来看它的监控界面，如图 18-1 所示。

这还只是 Nmon 所能监控的一部分，它还可以监控内核状态、系统负载、虚拟内存、NFS 等，只要有足够大的显示器，你可以把它们都添加到主界面中，服务器的一切活动尽收眼底，你可以在最短的时间内知道服务器现在在忙什么。

除此之外，你还可以用 Nmon 来录制数据，并通过另一个分析工具 Nmon Analyser 生成监控统计报告，如图 18-2 及图 18-3 所示，我们生成了间隔为 1 秒的监控报告，其中包括 CPU 使用率和磁盘 I/O。

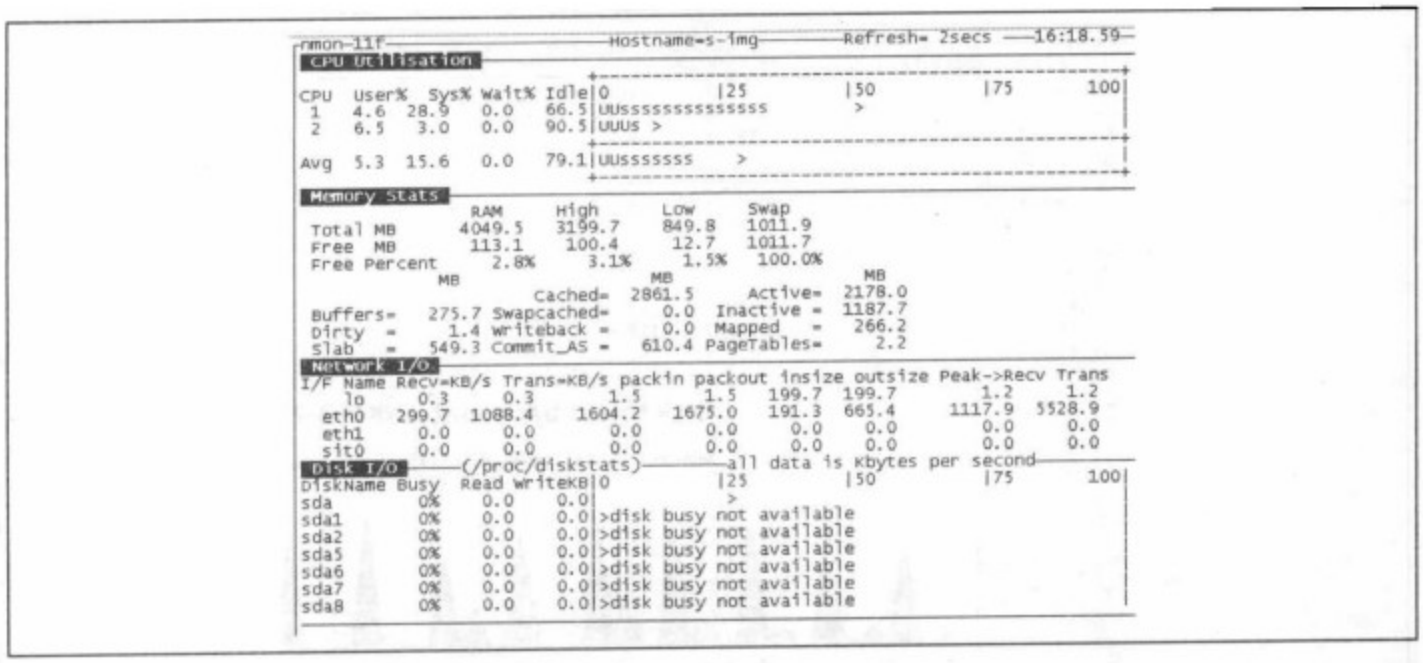
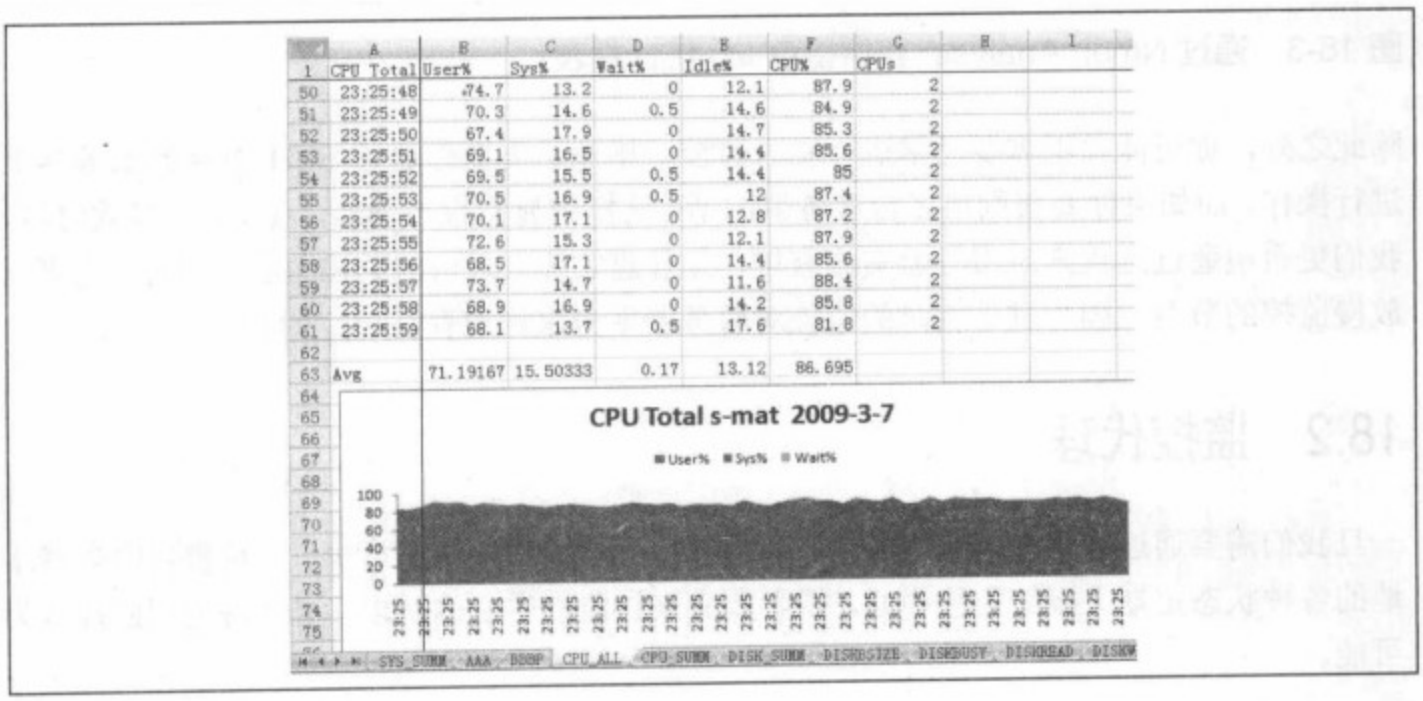


图 18-1　通过 Nmon 来进行系统监控

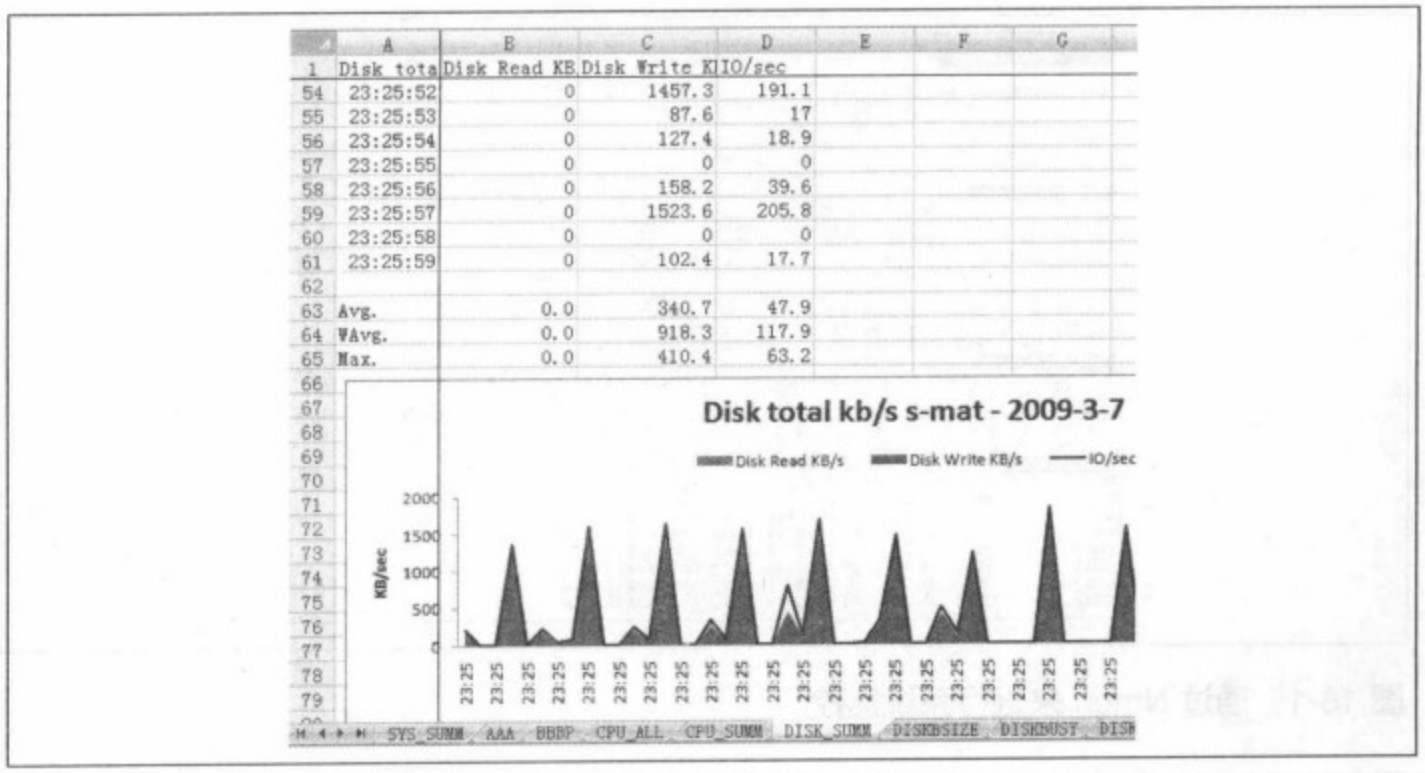


	A	B	C	D	E	F	G
1	CPU Total	User%	Sys%	Wait%	Idle%	CPU%	CPUs
50	23:25:48	74.7	13.2	0	12.1	87.9	2
51	23:25:49	70.3	14.6	0.5	14.6	84.9	2
52	23:25:50	67.4	17.9	0	14.7	85.3	2
53	23:25:51	69.1	16.5	0	14.4	85.6	2
54	23:25:52	69.5	15.5	0.5	14.4	85	2
55	23:25:53	70.5	16.9	0.5	12	87.4	2
56	23:25:54	70.1	17.1	0	12.8	87.2	2
57	23:25:55	72.6	15.3	0	12.1	87.9	2
58	23:25:56	68.5	17.1	0	14.4	85.6	2
59	23:25:57	73.7	14.7	0	11.6	88.4	2
60	23:25:58	68.9	16.9	0	14.2	85.8	2
61	23:25:59	68.1	13.7	0.5	17.6	81.8	2
62							
63	Avg	71.19167	15.50333	0.17	13.12	86.695	

图 18-2　通过 Nmon Analyser 获得 CPU 使用率统计报表

通常，我们只是在一些必要的时候进行实时监控，主要包括：

- 快速查看系统某组件的状态，比如进行网络结构调整后需要快速知道当前时刻的网络流量变化，而通过 MRTG 等方法你可能需要等待几分钟后的图表更新。
- 观察一些底层的系统状态，比如内核切换、进程队列、中断次数等，这些状态通常无法通过其他监控系统获得。
- 根据最小时间间隔的状态变化来进行一些诊断，比如通常的监控系统会将 5 分钟内的状态进行平均计算，那么 5 分钟内的变化情况我们并不知道，也许对于一些系统组件来说，这 5 分钟内的变化曲线隐藏着重要的线索。
- 你喜欢快节奏的工作方式，喜欢看到屏幕快速刷新。



	A	B	C	D	E	F	G
1	Disk tota	Disk Read KB	Disk Write K	IO/sec			
54	23:25:52	0	1457.3	191.1			
55	23:25:53	0	87.6	17			
56	23:25:54	0	127.4	18.9			
57	23:25:55	0	0	0			
58	23:25:56	0	158.2	39.6			
59	23:25:57	0	1523.6	205.8			
60	23:25:58	0	0	0			
61	23:25:59	0	102.4	17.7			
62							
63	Avg.	0.0	340.7	47.9			
64	WAvg.	0.0	918.3	117.9			
65	Max.	0.0	410.4	63.2			

图 18-3　通过 Nmon Analyser 获得磁盘 I/O 统计报表

除此之外，你可能并不需要经常进行实时监控，毕竟这需要你通过 SSH 登录到服务器上进行操作，而如果你要兼顾很多台服务器的话，这样做显然很不现实，其实，大多数时候，我们更希望通过远程来对服务器实施监控，并且建立统一入口的监控中心，同时，也可以放慢监控的节奏，因为过于实时的监控对服务器本身来说也存在一定的开销。

18.2　监控代理

一旦我们需要通过远程来监控服务器，那么监控代理程序便必不可少，它负责将服务器本地的各种状态定期上报给监控中心，或者响应来自监控中心的请求，这使得远程监控成为可能。

你可以为站点服务器开发专用的监控代理程序，并从本地文件中获取状态数据，在 Linux 中，一切都是文件，当然也包括各种系统状态，比如我们可以在这里看到详细的内存使用情况：

```
s-200:~ # cat /proc/meminfo
MemTotal:             4142240 kB
MemFree:               344428 kB
Buffers:               59864 kB
Cached:              2067336 kB
SwapCached:                4 kB
Active:              2598232 kB
Inactive:            1024660 kB
HighTotal:           3272000 kB
HighFree:             333912 kB
LowTotal:             870240 kB
LowFree:               10516 kB
```

```
SwapTotal:           1196800 kB
SwapFree:            1196672 kB
Dirty:               500 kB
Writeback:           0 kB
Mapped:              1086088 kB
Slab:                146180 kB
CommitLimit:         3267920 kB
Committed_AS:        6955560 kB
PageTables:          11908 kB
VmallocTotal:        112632 kB
VmallocUsed:         17640 kB
VmallocChunk:        94496 kB
HugePages_Total:     0
HugePages_Free:      0
HugePages_Rsvd:      0
Hugepagesize:        2048 kB
```

也可以看到系统负载和进程队列状态：

```
s-200:~ # cat /proc/loadavg
0.02 0.03 0.00 1/894 21407
```

这样一来，你只需要将这些数据打包，按照你定义的格式上报给监控中心即可。

不过，如果只是监控这些系统状态，你完全可以利用 SNMP 来完成这些工作，SNMP 服务器端本身便是一个出色的监控代理程序，它已经逐渐成为标准，并且支持很多异构平台。

比如，我们通过 SNMP 来获取另一台服务器的所有设备状态，如下所示：

```
    s-200:~ # snmpwalk -c public  -v 2c 10.0.1.201
    SNMPv2-MIB::sysDescr.0 = STRING: Linux s-mat 2.6.16.21-0.8-bigsmp #1
SMP Mon Jul 3 18:25:39 UTC 2006 i686
    SNMPv2-MIB::sysObjectID.0 = OID: NET-SNMP-MIB::netSnmpAgentOIDs.10
    DISMAN-EVENT-MIB::sysUpTimeInstance = Timeticks: (68135977) 7 days,
21:15:59.77
    SNMPv2-MIB::sysContact.0 = STRING: Sysadmin (root@localhost)
    SNMPv2-MIB::sysName.0 = STRING: s-mat
    SNMPv2-MIB::sysLocation.0 = STRING: Server Room
    SNMPv2-MIB::sysORLastChange.0 = Timeticks: (0) 0:00:00.00
    SNMPv2-MIB::sysORID.1 = OID: SNMPv2-MIB::snmpMIB
    SNMPv2-MIB::sysORID.2 = OID: TCP-MIB::tcpMIB
    SNMPv2-MIB::sysORID.3 = OID: IP-MIB::ip
    SNMPv2-MIB::sysORID.4 = OID: UDP-MIB::udpMIB
    SNMPv2-MIB::sysORID.5 = OID: SNMP-VIEW-BASED-ACM-MIB::vacmBasicGroup
    SNMPv2-MIB::sysORID.6 = OID: SNMP-FRAMEWORK-MIB::snmpFrameworkMIBComp
liance
    ......
```

这里仅仅列出了其中很少的一部分，事实上整个结果集非常庞大，一般我们要限定需要获取状态的设备名称，可以通过 MIB 来描述它，比如这里我们要获取一台 Windows 服务器的运行时间，如下所示：

```
    s-200:~ # snmpwalk -c public  -v 2c 10.0.1.210 hrSystemUptime
    HOST-RESOURCES-MIB::hrSystemUptime.0 = Timeticks: (630851406) 73 days,
0:21:54.06
```

包括 MRTG、Cacti 以及 Nagios 在内的很多监控工具都利用 SNMP 来监控远程服务器，而你只需要在被监控的服务器上开启 SNMP 服务，同时对监控来源进行授权配置即可。

当然，SNMP 并不是万能的，有一些我们希望监控的服务并没有提供相应的 SNMP 支持，比如我们要监控 Nginx 服务器当前的 HTTP 并发连接数，该怎么做呢？是否需要自己开发监控代理程序呢？幸运的是，Nginx 提供了必要的 HTTP 监控接口，你可以直接在监控中心请求它即可，比如我们通过请求以下 URL：

```
http://10.0.1.200/status
```

便可以看到 Nginx 当前的运行状态：

```
Active connections: 3020
server accepts handled requests
 5440803 5440803 7336362
Reading: 471 Writing: 2340 Waiting: 209
```

总之，通过监控代理程序，我们可以更加轻松地了解服务器的各种状态，所以，不论你希望监控服务器的何种状态，只需要考虑开发相应的监控代理程序即可，你甚至可以监控 CPU 的温度，前提是你的内核能够支持。

18.3 系统监控

通常情况下，我们通过 SNMP 便可以很容易地对服务器进行一些常规的系统监控，这包括 CPU 使用率、系统负载、内存使用率、网络 I/O、磁盘 I/O、磁盘使用率等，这些是我们比较关心的。当然，我们还需要建立监控中心，对这些状态数据进行统计和呈现。幸运的是，有很多开源产品可以帮助我们，这里我们主要以 Cacti 为例，它完全可以支持刚刚提到的这些系统监控，并且绘制出相应的图表，便于我们浏览。

Cacti 采用 RRDtool 作为监控数据的存储引擎，它是一种专门针对绘制坐标图而设计的存储格式，相对于其他存储结构来说要节省很多存储空间，这为我们长期监控大量服务器提供了可能。相对于本书的主题来说，我们不打算花大量的篇幅从运维工作的角度来介绍监控系统本身的使用，你可以查阅大量相关的在线文档。

对于刚刚提到的 CPU 使用率等状态监控，其数据和图表背后的含义，我们在前面的章节中已经有过详细介绍，所以，接下来我们只会对图表本身进行简单的介绍，当你实际浏览这些监控图表时，应该充分结合前面的内容以及站点的实际情况来具体分析它们的含义。

作为服务器的心脏，CPU 的使用率是我们非常关注的一项系统状态，如图 18-4 所示，这是一台运行着 MySQL 的专用数据库服务器，图上清晰地呈现出 CPU 在用户空间和内核空间的时间开销，可以看得出，用户空间进程花费了大量的 CPU 时间，这正符合数据库应用的特点，如果希望更加深入地了解数据库内部的具体开销，你同样可以对 MySQL 进

行监控，随后我们会介绍这方面的内容。

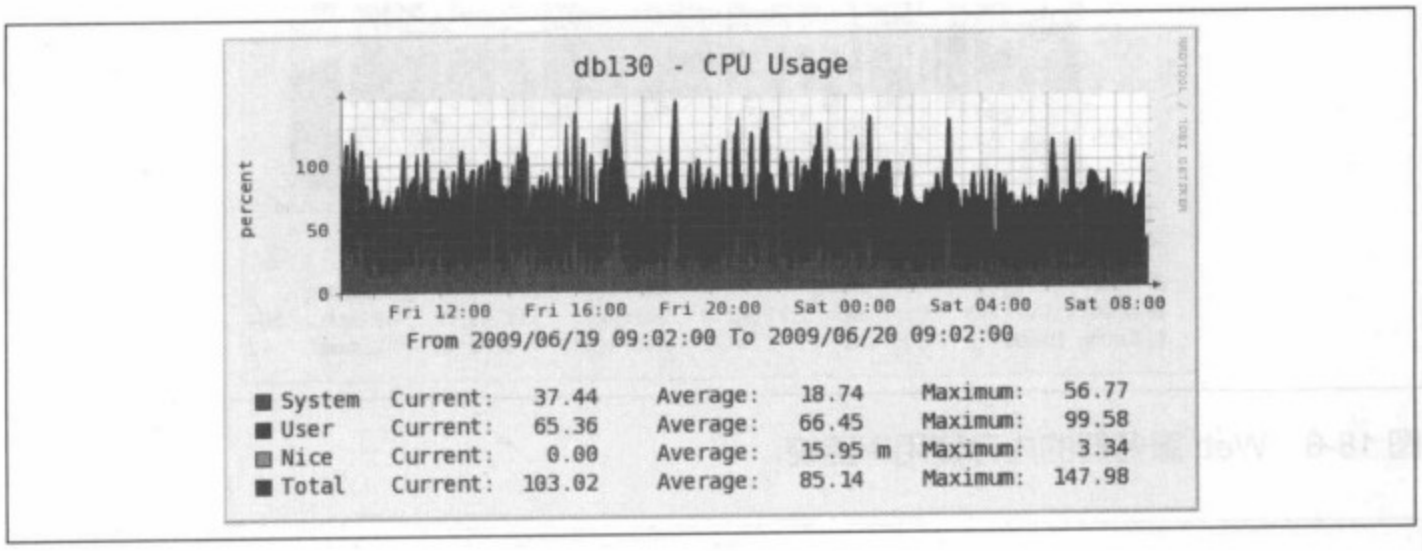


图 18-4　CPU 使用率监控

另外，不难看出，这里的图表采用了常见的累积图方式，它的好处是可以直接呈现出累加结果，比如这里的累加结果便是 CPU 使用率总和。

通过前面的介绍，我们已经了解了系统负载的意义，它的监控图表如图 18-5 所示，我们可以看到 1 分钟、5 分钟、15 分钟的平均负载变化情况，但是，它们的累加结果没有太多实际意义。

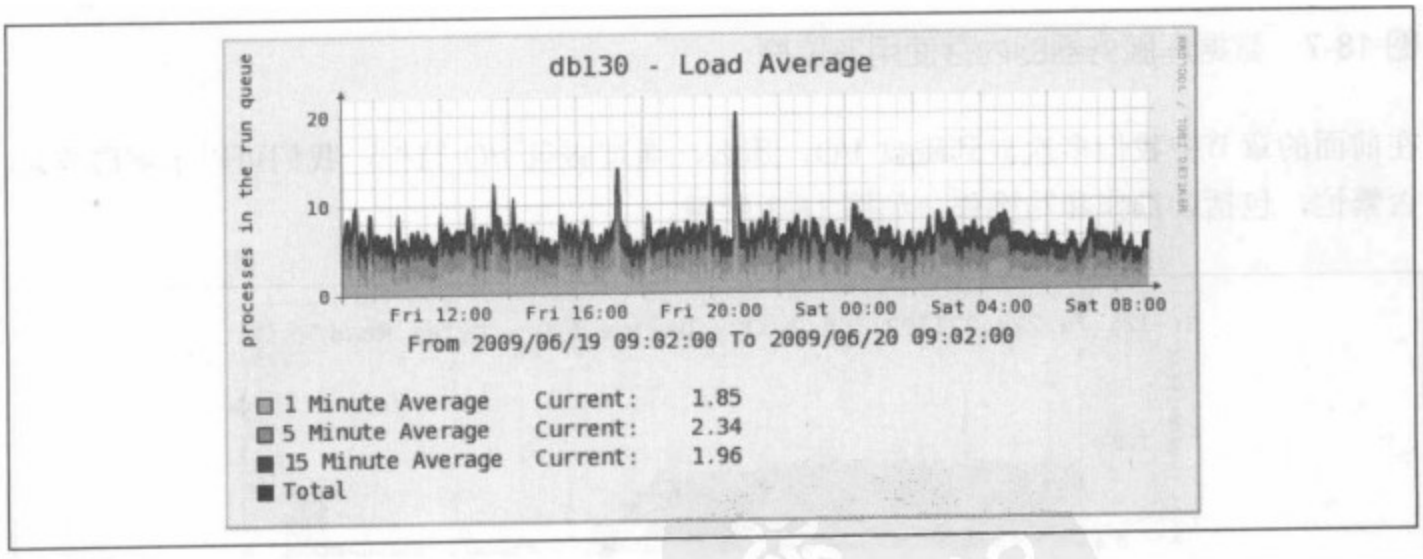


图 18-5　系统负载监控

如图 18-6 所示，这是一台运行着 Apache 的 Web 服务器，可以看到它的内存使用率并不饱和，事实上，它的 httpd 进程并不多，所以并没有消耗太多的物理内存。

而对于以下的 MySQL 服务器，我们为它最大程度地分配了物理内存作为 Innodb 缓存，可以看到内存使用率已经接近饱和，如图 18-7 所示。

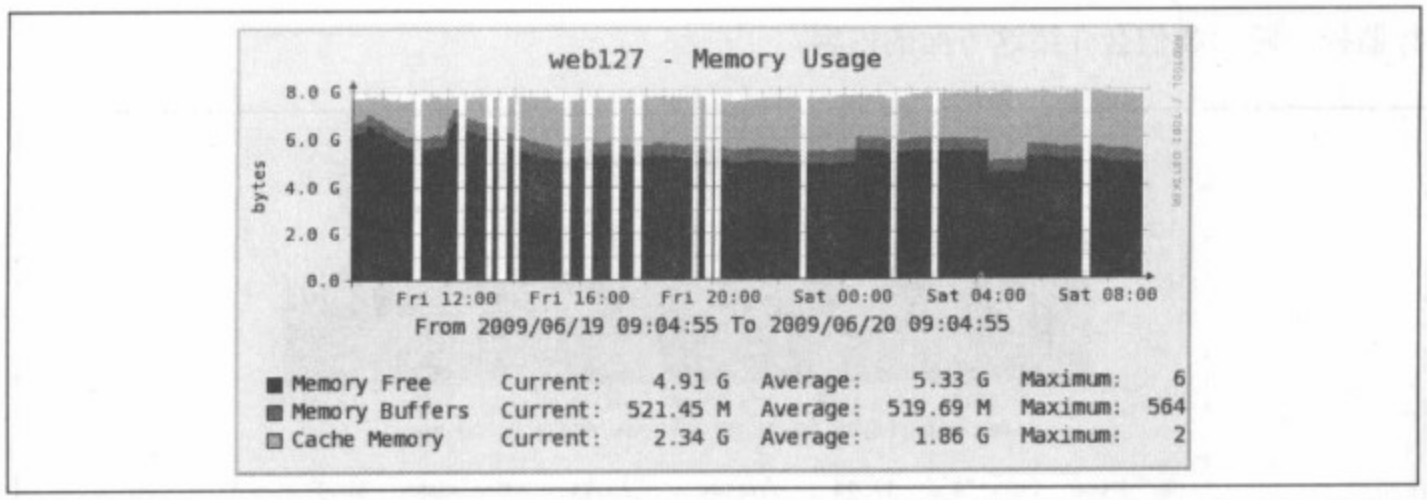


图 18-6　Web 服务器的内存使用率监控

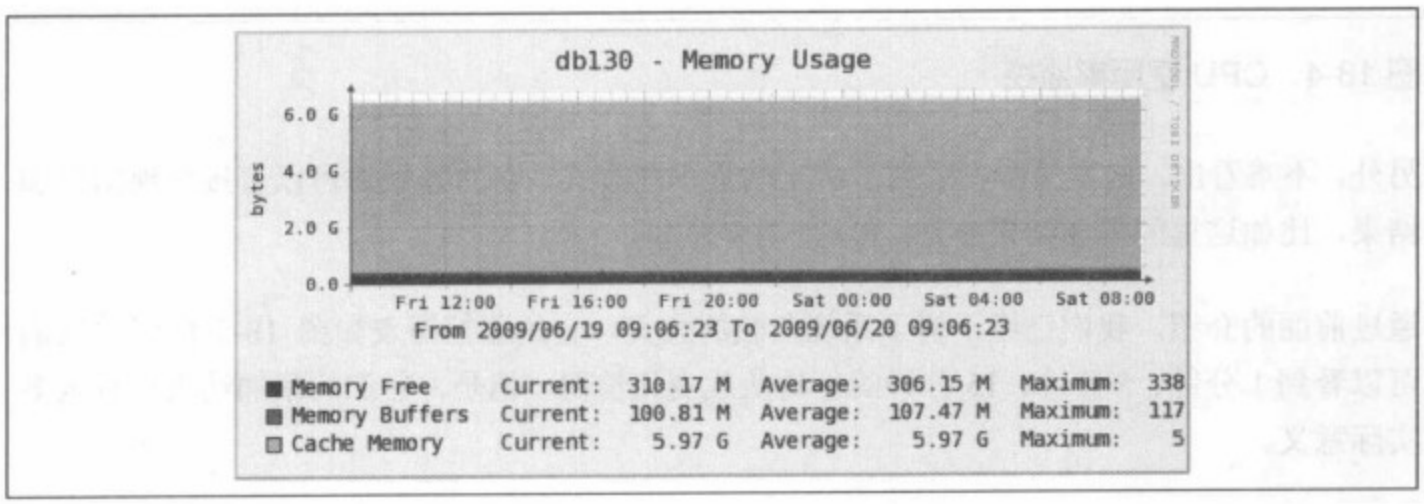


图 18-7　数据库服务器的内存使用率监控

在前面的章节中我们多次提到磁盘 I/O，那么，通过磁盘 I/O 监控，我们可以了解磁盘是否繁忙，包括读操作和写操作，如图 18-8 所示。

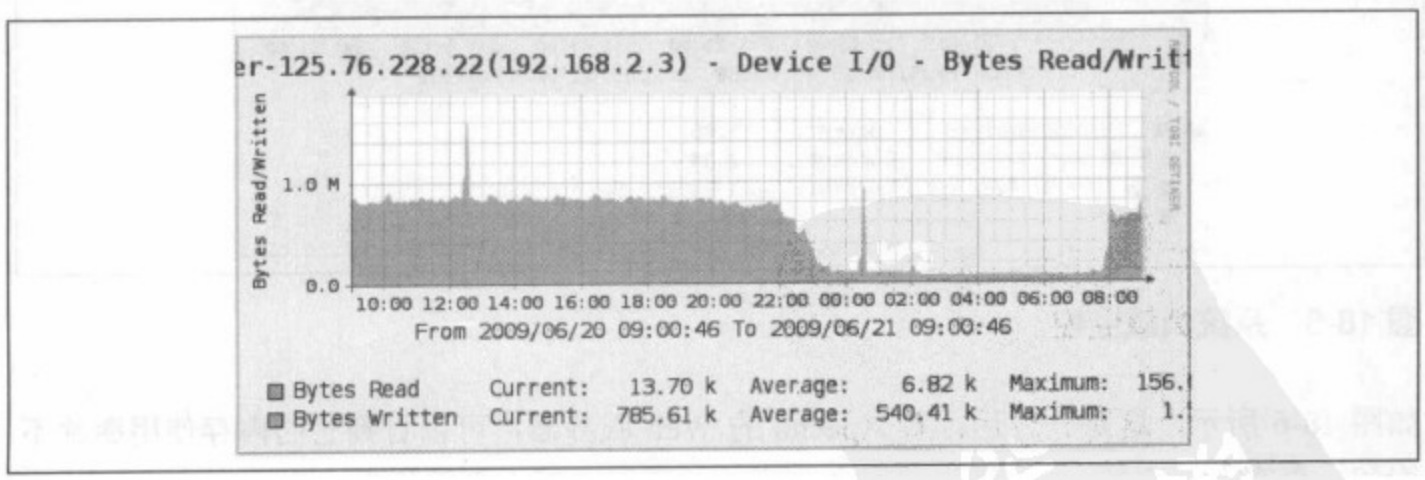


图 18-8　磁盘 I/O 监控

磁盘空间也是我们关注的另一个方面，通过磁盘使用率监控，我们可以更好地了解磁盘空间的使用情况和变化趋势，如图 18-9 所示。

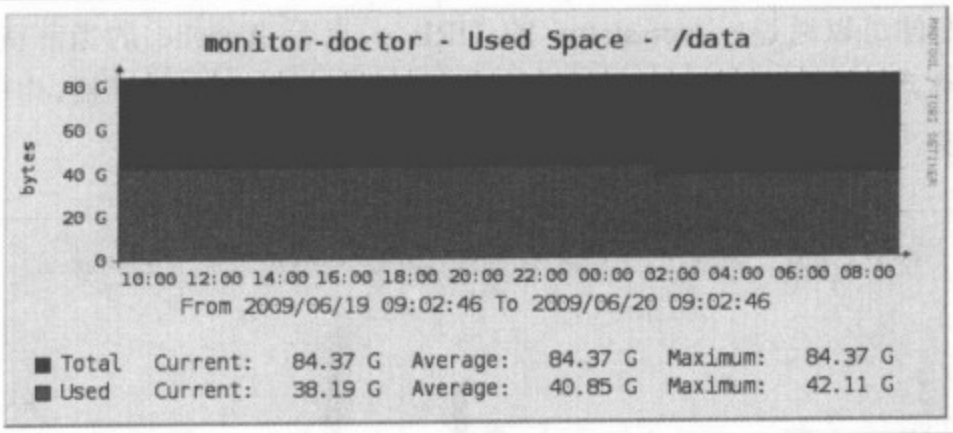


图 18-9　磁盘使用率监控

网络流量监控可以帮助我们了解服务器的通信数据量，包括流入和流出两部分，如图 18-10 所示，对于提供 HTTP 服务的服务器来说，流出的数据量要大于流入的数据量，因为 HTTP 响应数据长度通常大于请求数据长度。另一方面，对于流量情况的了解，可以帮我们更好地把握带宽的使用情况。

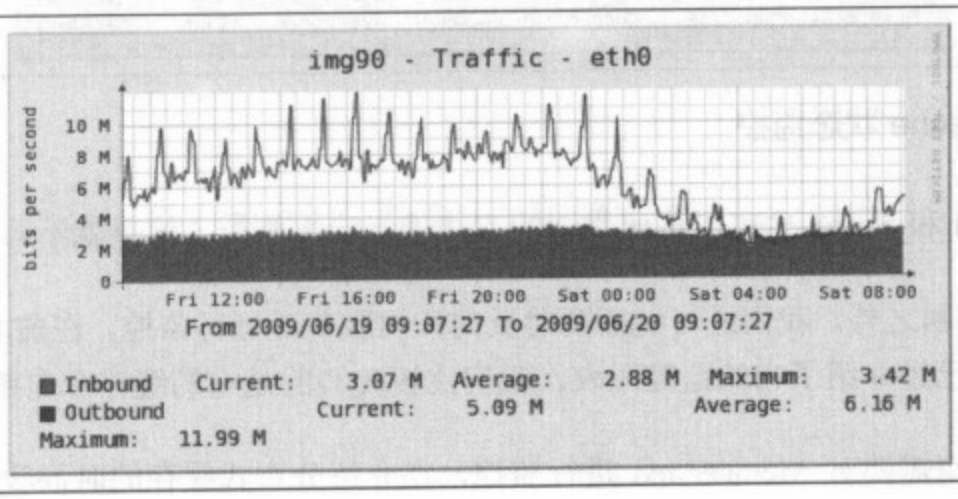


图 18-10　网络 I/O 监控

18.4　服务监控

除了基本的系统监控之外，我们还非常关心应用层服务的状态，它们更加直接地反映了 Web 应用的性能。Cacti 中支持插件机制，我们可以通过一些第三方模板来监控这些服务，同时，我们还需要相应的监控代理，幸运的是，一些常用的服务已经考虑到这一点，它们提供了一定的监控访问接口，比如 Apache 中基于 HTTP 的 mod_status 模块，在 httpd.conf 中配置如下：

```
ExtendedStatus On
<Location /server-status>
    SetHandler server-status
    Order deny,allow
    Deny from all
    Allow from all
</Location>
```

这样一来，我们便可以通过/server-status 的 URL 来查看 Apache 的当前状态，这相当于 Apache 的监控代理，有了它以后，我们在 Cacti 中通过相应的第三方模板，生成了如图 18-11 所示的监控图表。

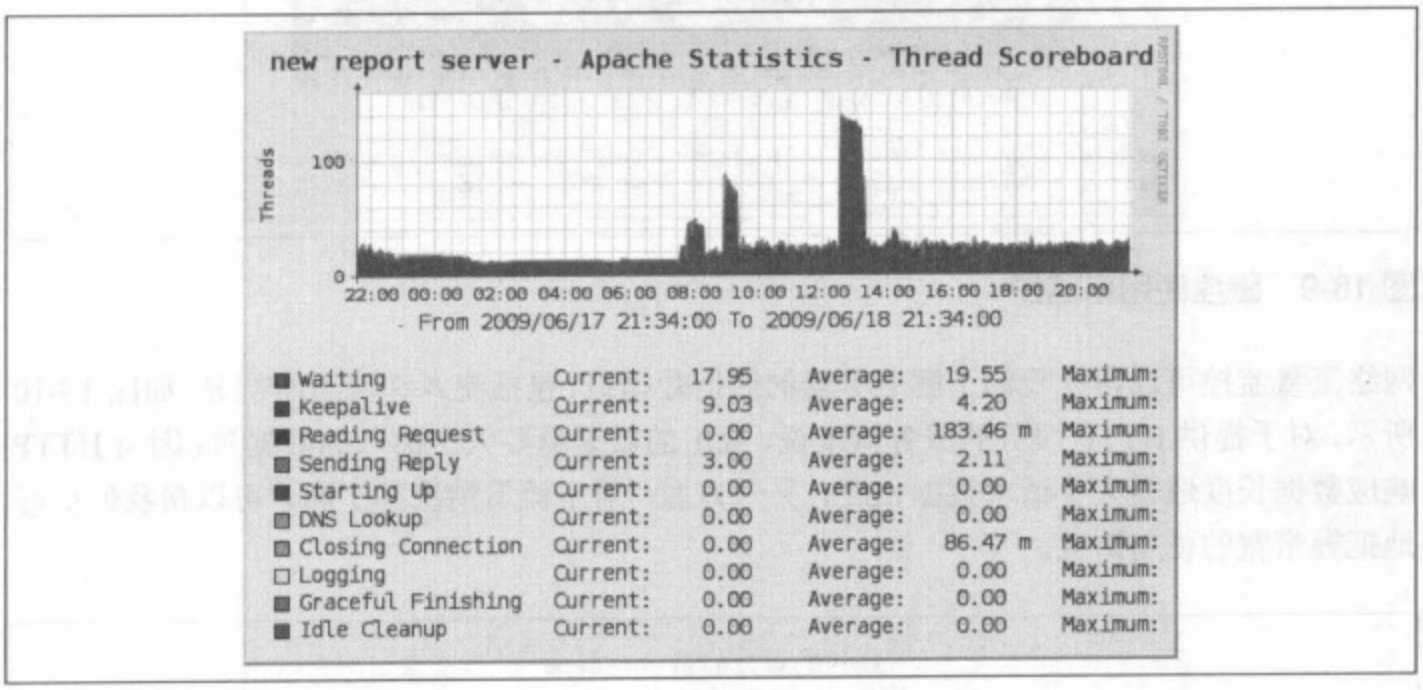


图 18-11　Apache 状态监控

同样，Lighttpd 和 Nginx 也可以按照类似的方式进行状态监控，这里就不再详细介绍。

除了 Web 服务器之外，我们还可能需要对反向代理服务器进行监控，在前面介绍 Varnish 的时候，我们已经介绍了它的监控图表，我们比较关心的是它的缓存命中率等参数。

另外，你或许也需要对 Memcached 进行监控，在介绍分布式缓存的时候我们曾经介绍过这方面的内容，包括 I/O 数据量、缓存命中率、空间使用率等。

同样，在数据库性能优化的章节中，我们了解了数据库的性能对于站点应用来说何等重要，所以，数据库的监控必不可少，我们在 Cacti 中可以对 MySQL 进行各种监控，如图 18-12~图 18-14 所示。

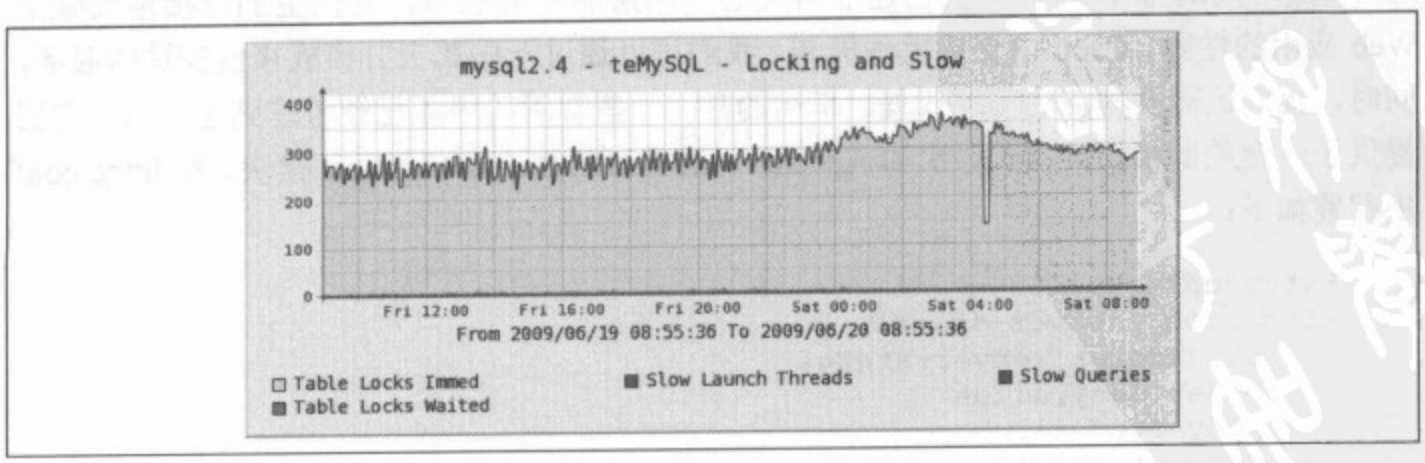


图 18-12　MySQL 的锁定和慢查询监控

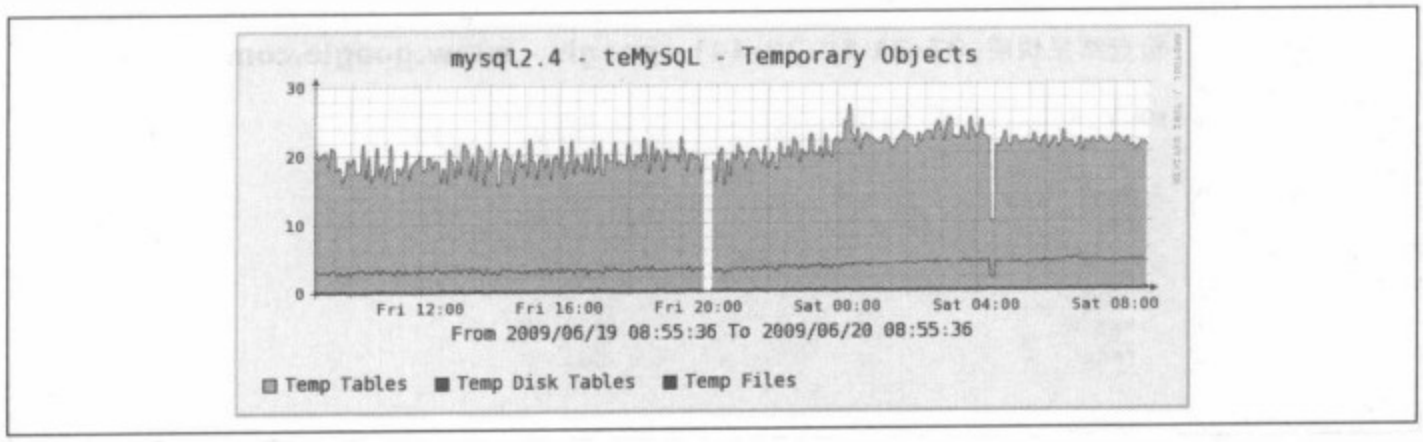


图 18-13　MySQL 的临时表监控

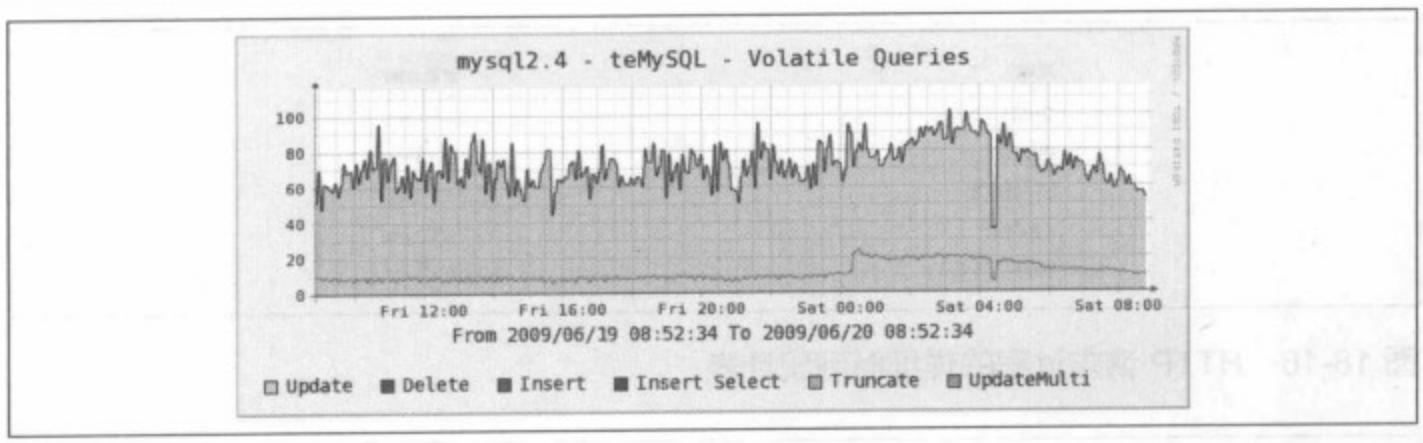


图 18-14　MySQL 的查询统计监控

这几项监控图表所表达的意义，我们同样在前面的章节中有过详细介绍，通过监控，你能够更加自如地运用各种优化策略。

18.5　响应时间监控

的确，以上这些监控措施可以帮助我们直观地了解系统和服务本身的运行状况，但是，从另一个角度来看，位于网络另一端的最终用户只关心响应时间，你知道它们的真实体验吗？我很想知道。同时，用户访问多种不同的 Web 应用，比如站点首页、购物车、查找好友等，它们执行不同的商业逻辑。当然，我们希望了解这些不同 Web 应用的真实服务品质，尽管它们可能都运行在一个 Web 服务器上，但是我们对这些应用的性能要求并不相同，显然，通过以上这些监控手段已经力不从心。

所以，这时候我们需要借助一些第三方工具，这里推荐监控宝（www.jiankongbao.com），它像 Cacti 一样提供基于 Web 的服务界面，但你不用花时间去安装和部署，只需要注册和添加监控任务便可以快速使用。

由于监控宝拥有独立的监控点，所有它可以定时模拟用户来请求你指定的多个 URL，并且记录详细的响应时间，如图 18-15~图 18-17 所示。

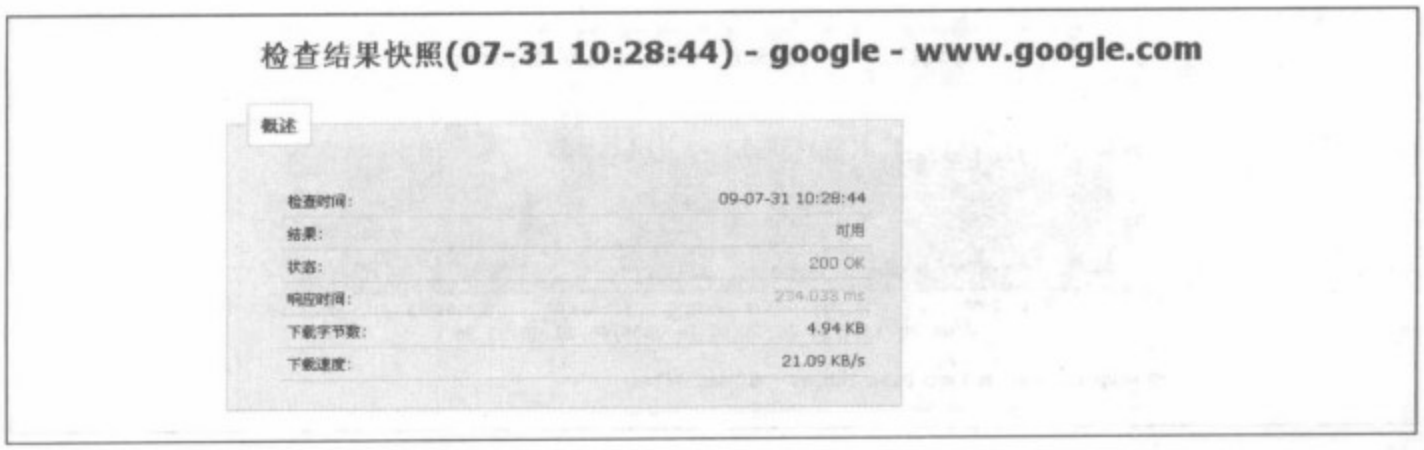


图 18-15　HTTP 请求结果快照

过程	消耗时间
DNS域名解析:	24.51 ms
建立连接:	80.516 ms
服务器计算:	71.29 ms
内容下载:	57.717 ms
响应时间（总）:	**234.033 ms**

图 18-16　HTTP 请求过程的详细时间统计表

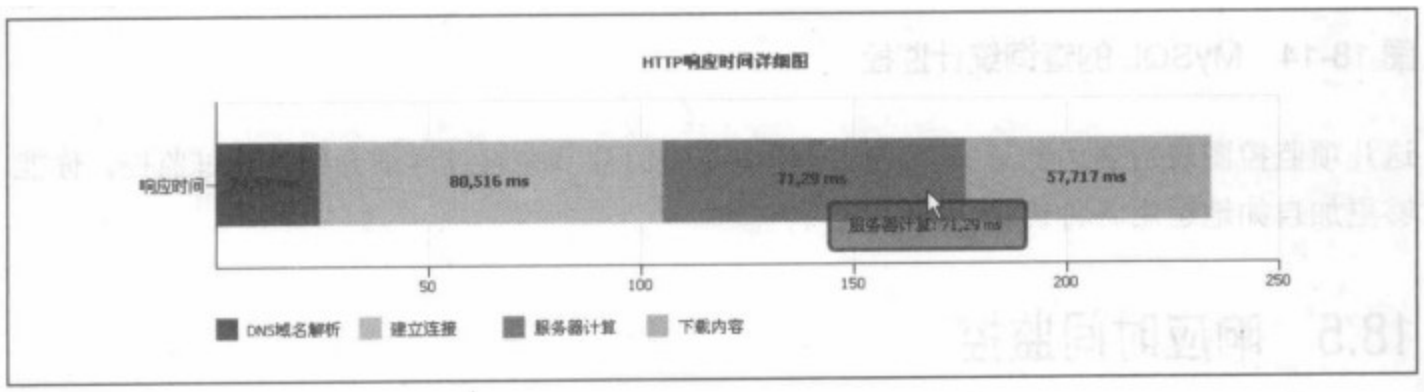


图 18-17　HTTP 请求过程的详细时间展示图

可以看到，在这一系列时间中，DNS 域名解析所消耗的时间实际上取决于 DNS 服务器的性能以及用户到 DNS 服务器的网络状况。除此之外，DNS 记录还可能会缓存在互联网接入服务商的各级 DNS 服务器上，这取决于 DNS 记录的 TTL 值。

建立连接的时间体现了 Web 服务器能否快速地接入用户的请求。在通常情况下，当 Web 服务器的同时连接数达到预设限制时，Web 服务器可能会拒绝新的接入请求，而对于 Apache 这样的多进程模型，当进程数不断增多时，由于上下文切换的时间开销也随之增加，所以建立连接的平均时间也逐渐开始延长。

接下来的服务器计算时间很容易理解，对于静态文件的访问，这部分时间主要用于文件的定位，如果是较小的文件，那么还会包括文件读取时间；而如果是较大的文件，前面章节曾提到过 Web 服务器会使用 sendfile 来直接传送文件内容到网络设备，所以读取文件的时

间并没有算入这里的服务器计算时间，而是归入内容下载时间。

对于动态内容的访问，这里的服务器计算时间具有非常重要的参考价值，这些时间意味着什么，你一定比我更加清楚。事实上，这本书的大部分篇幅都在介绍如何减少这部分时间。

最后的内容下载时间从根本上来说取决于用户和服务器两端的带宽。如果你希望减少这部分时间，在前面关于数据网络传输的章节中你会找到部分答案。

另外，监控宝还提供了报告视图，你可以查看一段时间范围的响应时间变化曲线，如图 18-18 所示。

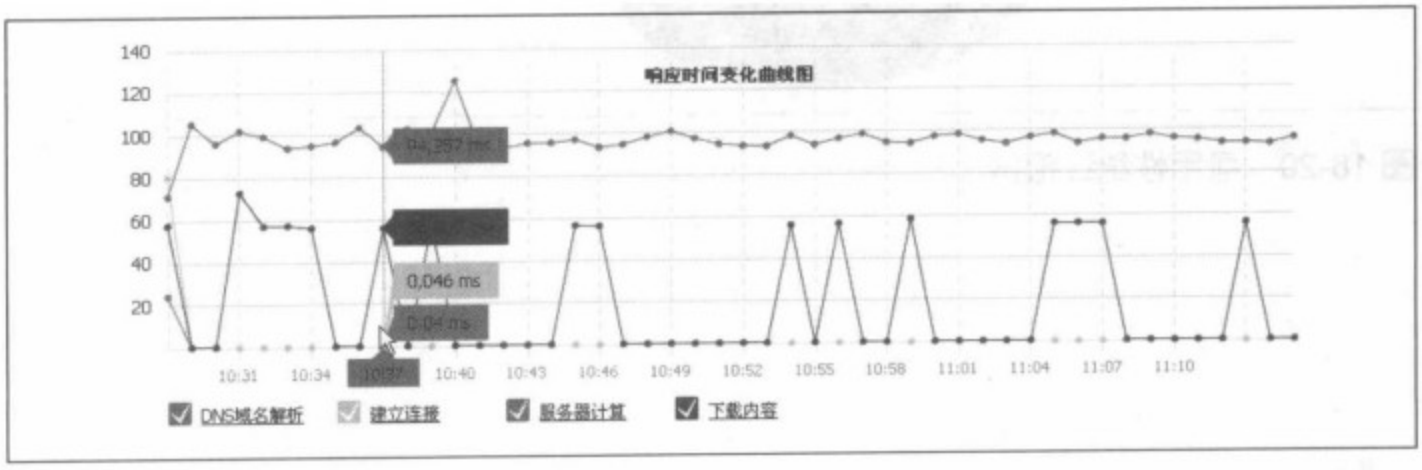


图 18-18　某时间范围的响应时间变化曲线

同时，还可以按照响应时间范围的次数和百分比进行统计，你可以直观地了解服务质量，如图 18-19 所示。

响应时间范围	次数	百分比
<200 ms	47	97.92%
200~500 ms	1	2.08%
500~1000 ms	0	0.00%
1~2 s	0	0.00%
2~5 s	0	0.00%
5~10 s	0	0.00%
>10 s	0	0.00%

图 18-19　响应时间范围的次数和百分比统计

当然，可用性统计也必不可少，如图 18-20 所示。

除此之外，监控宝还提供了一些其他的工具（如故障报警），并且支持 HTTP 以外的其他协议（如 DNS 服务器的监控），这里就不详细介绍了。

	时间	时间百分比	检查次数
可用	675 分钟	100.00%	48
失败	0 分钟	0.00%	0

可用性比例

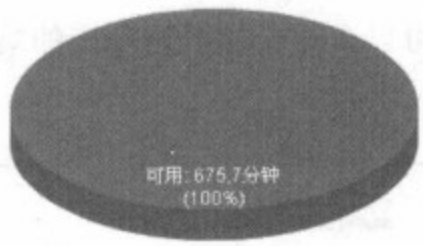


图 18-20　可用性统计报告

REFERENCES
参考文献

ab - Apache HTTP server benchmarking tool
http://httpd.apache.org/docs/2.0/programs/ab.html

Smarty Template Engine
http://www.smarty.net/manual/en/

Opcode Wikipedia
http://en.wikipedia.org/wiki/Opcode

PHP APC
http://cn.php.net/apc/

XCache
http://xcache.lighttpd.net/

Xdebug - Debugger and Profiler Tool for PHP
http://xdebug.org/

Hypertext Transfer Protocol -- HTTP/1.1
http://www.w3.org/Protocols/rfc2616/rfc2616.html

Varnish
http://varnish.projects.linpro.no/

Edge Side Includes language
http://www.w3.org/TR/esi-lang

Memcached
http://www.danga.com/memcached/

mysqlreport
http://hackmysql.com/mysqlreport

MySQL 5.1 Reference Manual
http://dev.mysql.com/doc/refman/5.1/en/

The Linux Virtual Server Project - Linux Server Cluster for Load Balancing
http://www.linuxvirtualserver.org/

Roundup on Parallel Connections
http://www.stevesouders.com/blog/2008/03/20/roundup-on-parallel-connections/

WebDAV
http://www.webdav.org/

HTTP Extensions for Distributed Authoring -- WEBDAV
http://www.webdav.org/specs/rfc2518.html

litmus - WebDAV server protocol compliance test suite
http://webdav.org/neon/litmus/

Requirements for a Distributed Authoring and Versioning Protocol for the World Wide Web
http://tools.ietf.org/html/rfc2291

pyinotify
http://trac.dbzteam.org/pyinotify

Hash tree Wikipedia
http://en.wikipedia.org/wiki/Hash_tree#cite_note-0

inotify Wikipedia
http://en.wikipedia.org/wiki/Inotify

Mogilefs
http://www.danga.com/mogilefs/

Hadoop
http://hadoop.apache.org/core/

Gearman
http://www.gearman.org/docs/api/index.html

Map/Reduce Paper
http://labs.google.com/papers/mapreduce-osdi04-slides/index.html

LVS SNMP Module
http://kb.linuxvirtualserver.org/wiki/Net-SNMP-LVS-Module

/dev/epoll 74
/dev/poll 74

A

ab 42
AIO 81
AJAX（Asynchronous JavaScript and XML） 200
Apache 39,181
Apache/mod_cache 168
Apache/mod_include 116
Apache/mod_status 35
Apache-prefork 51
APC 107,127
ARP 绑定 21
A 记录 284

B

BSD 72
BTree 242
边缘触发（Edge Triggered） 74
编译（compile） 122
别名（CNAME） 285
并发用户数 37
并行计算 379

C

Cacti 388
Cakephp 99
CDN 325
CMS（内容管理系统） 115
cookies 225
crontab 115,343
操作码（Operate Code, opcode） 122
层叠样式表（CSS） 1
查询缓存 264
持久连接 65
传播时间 29
垂直分区 366
存储节点（Stroage Node） 350

D

daemon 115
DDN 专线 213
Django 99
DMA 69
DNS TTL 288
DNS 负载均衡 284
带宽 22
带权重的最小连接（WLC） 314
单点故障（Single Point of Faliure） 273
动态 DNS 289
读写分离（R/W Splitting） 363
独享带宽 26
多路 I/O 就绪通知 71

E

epoll 74
ESI（Edge Side Includes） 199
ETag 152
Expires 160

F

fastcgi 51
Firefox 144
FreeBSD 75
fsck 355
反范式化（Denormalization） 268
反向代理（Reverse Proxy） 179,346
反向代理负载均衡 292
分表 370
分布式消息队列 374
负载均衡（Load Balancing，LB） 274
负载均衡调度器 284

G

Gearman 375
高可用性（High Availability，HA） 274
格林威治标准时间 150
工作服务器（Worker Server） 375
共享带宽 26

H

Hadoop 348
HAProxy 296
Hash tree 345
Heartbeat 324,326
HTTP/1.1 152
HttpWatch 148,211
HTTP 重定向 275
后端服务器（Back-end Server） 293
缓冲（buffer） 98
缓存（cache） 98
缓存命中率 190,194
缓存协商 147

I

I/O 模型 68
IE 144
Innodb 259
inotify 345
IOWait 58
IP Encapsulation 325
IP Tunneling 325
iptables 179,308
IPVS（IP Virtual Server） 311
ipvsadm 312
IP 别名 318
IP 负载均衡 305
IP 隧道（IP Tunneling） 325

J

Java 扩展 132
JSON 格式 224
脚本（JavaScript） 1
解释（parse） 122
进程 49
进程调度 50
进程调度器（Scheduler） 50
进程切换 53
进程优先级 50
镜像下载 275

K

kqueue 75
控制器（Controller） 99

L

Lighttpd 87,181
Lighttpd/mod_ssi 116
Lighttpd/mod_status 35
Linux 2.4 74
Linux 2.6 74
Linux Bonding 326
LRU（Least Recently Used） 222
LVS（Linux Virtual Server，Linux 虚拟服务器） 311
LVS-DR 320

LVS-NAT 312
LVS-TUN 325
临时表 266
浏览器并发数 210
浏览器缓存 144

M

MAC 地址 19
Map/Reduce 380
memcached 109,128,221
MemcacheDB 269
  Berkeley DB 269
MemcacheQ 379
MIB 387
MogileFS 348
MPM 54
MRTG 26,385
MVC 99
MyISAM 260
MySQL Proxy 364
MySQL 238
MySQL/explain 243
mysqlreport 239
MySQL 主从复制 362
慢查询 251

N

NAT（网络地址转换） 178,306
Netfilter 308
NFS（Network File System） 330
Nginx 87,181
Nmon Analyser 384
Nmon 54,384
Novell 72
内存映射（mmap） 74
内核态 60
内嵌页面（iframe） 1

O

O_SYNC 70
opcode 121

P

Parsekit 122
PHP 121
PIO 69
poll 73
Python 121
瓶颈效应 2

Q

请求等待时间 40
全表扫描（Full Table Scan） 242

R

RAID 70
RPC（Remote Procedure Calls） 330
RR（Round Robin） 278
RRDtool 388
rsync 342
RTree 242
Ruby 121
任务服务器（Job Server） 375

S

Samba 330
SCP 339
select 73
sendfile 78
SFTP 340
SIGIO 74
Smarty 99
SMP（对称多处理器） 212
SNMP 284,387
Spock Proxy 372
Squid 184

SSH（Secure Shell） 338
SSI（服务器端包含） 115,159
Subversion 342
SWAP 56
上下文切换 53
实时信号（Real Time Signal） 74
事务操作 263
视频码率 218
视图（View） 99
水平触发（Level Triggered） 74
水平分区（Sharding） 367
索引 241
索引缓存 253
索引扫描（Index Scan） 242
锁竞争 60

T

同步非阻塞 I/O 71
同步阻塞 I/O 69
吞吐率 35

U

UDP 331
UNIX Socket 213
URL Rewrite（地址重写） 167
URL 映射 167

V

Varnish 184,301
VCL（Varnish Configuration Language） 187
VIRT 56

W

WebDAV 342
Wordpress 213
文件描述符 175

X

XBitHack 120,159
XCache 108,131
Xdebug 133
系统调用 60
系统负载 52
下载速度 28
先进先出（First In First Out，FIFO） 375
线程 50
响应时间 28
响应状态码 152
写操作频繁（write-heavy） 366
性能跟踪器（Profiler） 140
序列化（Serialize） 224

Y

压力测试 36
页面缓存（Page Cache） 98
异步 I/O 81
异步计算 374
用户代理（User Agent） 148
用户态 60
预写日志方式（Write-Ahead Logging，WAL） 254
运行队列（Run Quere） 50

Z

Zend Engine 122
Zend 99
粘滞会话（Sticky Sessions） 304
直接 I/O 77
直接路由（Direct Route，DR） 317
智能解析 287
追踪器（Tracker） 351
多次读取（write-once-read-many） 352
组合索引 246
最短期望时间延迟（SED） 314
最小连接（LC） 314