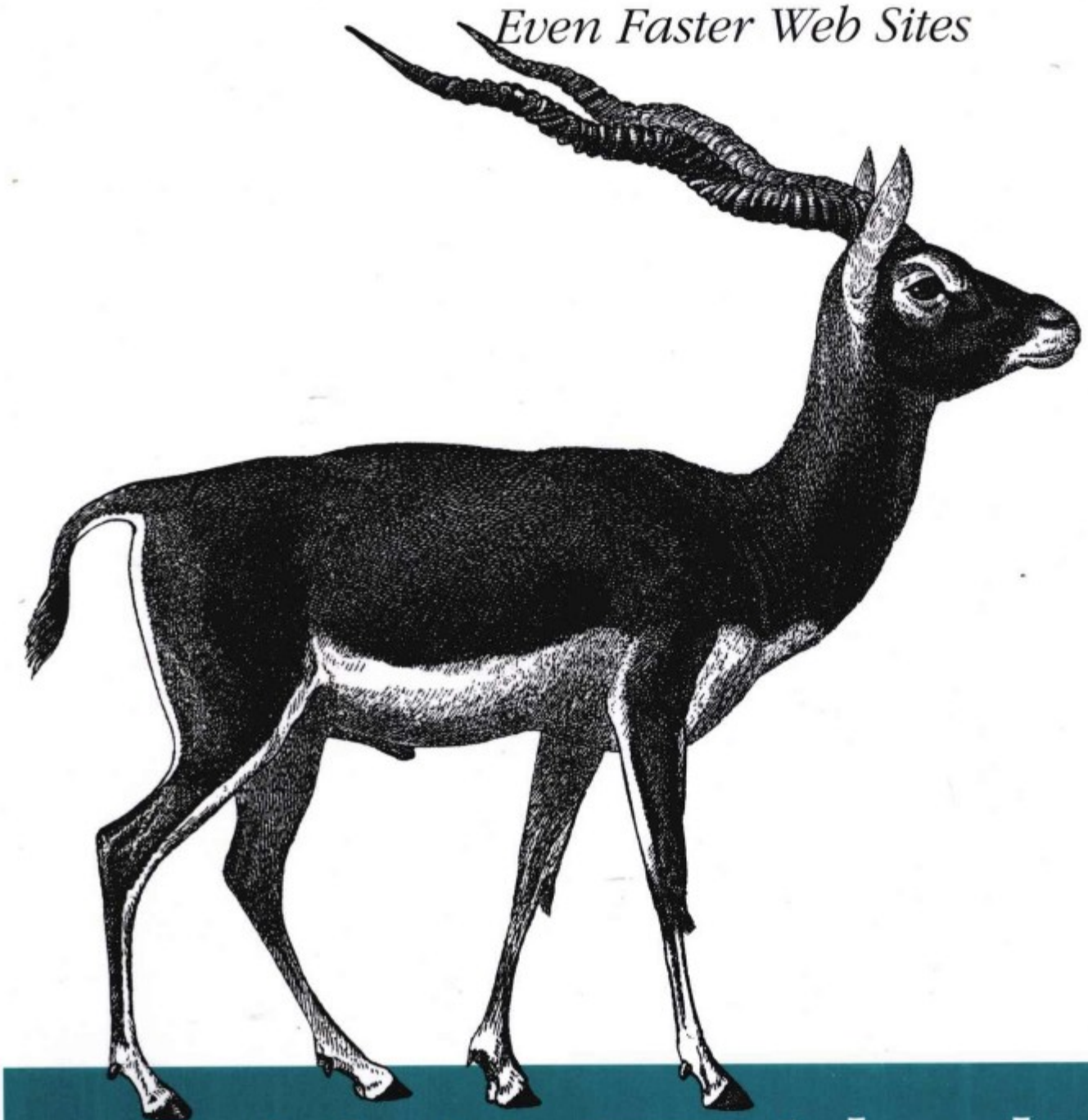
Performance Best Practices for Web Developers

Even Faster Web Sites



高性能网站建设进阶指南

*Web*开发者性能优化最佳实践

Steve Souders 著

口碑网前端团队 译

O'REILLY®

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

O'REILLY®

高性能网站建设进阶指南

性能是任何一个网站成功的关键，然而，如今日益丰富的内容和大量使用Ajax的Web应用程序已迫使浏览器达到其处理能力的极限。Steve Souders是Google Web性能布道者和前Yahoo!首席性能工程师，他在本书中提供了宝贵的技术来帮助你优化网站性能。

Souders的上一本畅销书《高性能网站建设指南》（High Performance Web Sites）震惊了Web开发界，它揭示了在客户端加载一个网页的时间大约占用了总时耗的80%。在《高性能网站建设进阶指南》（Even Faster Web Sites）这本书中，Souders与另外8位专家级特约作者提供了提升网站性能的最佳实践和实用建议，主要包括以下3个关键领域。

- JavaScript——你将获得忠告：理解Ajax性能、编写高效的JavaScript、创建快速响应的应用程序、无阻塞加载脚本等。
- 网络——你将学到：跨域共享资源、无损压缩图片大小，以及使用块编码加快网页渲染。
- 浏览器——你将发现：避免或取代iframe的方法、简化CSS选择符，以及其他技术。

对于当前的富媒体网站和Web 2.0应用程序来说，速度至关重要。在本书中，你将学习如何节省宝贵的网站加载时间，使网站更快地响应用户的请求。

“《高性能网站建设进阶指南》，有最新的研究成果，能激发你的智慧，让网站快如闪电。我喜欢这本书的风格——许多主题的作者都是在该领域进行过深入研究的最受尊敬的权威专家。我们团队中每个人都需要有一本。”

—— Bill Scott，
Netflix总监和
UI工程师

Steve Souders现在Google工作，负责Web性能和开源组织。他是Firebug的性能分析扩展工具——YSlow的创建者，也是O'Reilly Web性能与运作会议Velocity的联合主席。Steve经常在技术会议和诸如Microsoft、Amazon、MySpace、LinkedIn和Facebook这样的知名高科技公司发表演讲。

特约作者：

Dion Almaer、Douglas Crockford、Ben Galbraith、Tony Gentilcore、Dylan Schiemann、Stoyan Stefanov、Nicole Sullivan和Nicholas C. Zakas

图书分类：Web开发

责任编辑：周筠

www.oreilly.com

Broadview®
WWW.BROADVIEW.COM.CN

www.phei.com.cn

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行

This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-121-10544-9

9 787121 105449 >

定价：49.80元

O'REILLY®

高性能网站建设进阶指南：Web 开发者性能优化最佳实践

Even Faster Web Sites：Performance Best Practices for Web Developers

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内容简介

本书是《高性能网站建设指南》姊妹篇。作者 Steve Souders 是 Google Web 性能布道者和前 Yahoo! 首席性能工程师。在本书中，Souders 与 8 位专家分享了提升网站性能的最佳实践和实用建议，主要包括：理解 Ajax 性能，编写高效的 JavaScript，创建快速响应的应用程序、无阻塞加载脚本，跨域共享资源，无损压缩图片大小，使用块编码加快网页渲染；避免或取代 iframe 的方法，简化 CSS 选择符，以及其他技术。

版权贸易合同登记号 图字：01-2009-5178

图书在版编目（CIP）数据

高性能网站建设进阶指南: Web 开发者性能优化最佳实践 /（美）桑德斯（Souders,S.）著; 口碑网前端团队译.—北京: 电子工业出版社，2010.4
书名原文 :Even Faster Web Sites:Performance Best Practices for Web Developers
ISBN 978-7-121-10544-9

Ⅰ.高… Ⅱ.①桑…②口… Ⅲ.主页制作－程序设计－指南 Ⅳ.TP393.092-62

中国版本图书馆 CIP 数据核字（2010）第 046969 号

策划编辑：徐定翔
责任编辑：周 筠
项目管理：梁 晶
封面设计：Karen Montgomery，张 健
印 刷：北京市天竺颖华印刷厂
装 订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
北京市海淀区万寿路173信箱 邮编：100036
开 本：787×980 1/16 **印张：**16.25 **字数：**380千字
印 次：2010年4月第1次印刷
定 价：49.80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。

O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc. 授权电子工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在Unix、X、Internet和其他开放系统图书领域具有领导地位的出版公司，同时也是在线出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》(被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet 门户和商业网站），再到 WebSite（第一个桌面 PC 的Web服务器软件），O'Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为 O'Reilly Media, Inc.紧密地与计算机业界联系着，所以O'Reilly Media, Inc. 知道市场上真正需要什么图书。

译者序

“信息网络的扩展正在为我们的星球建立一个新的神经系统”，而 Web 正是我们与这个系统对接的最重要平台。现在，信息与用户量在 Web 平台上呈爆炸式增长，用户与 Web 界面的交互越来越复杂，会话时间也越来越长，速度已经成为 Web 产品使用体验好坏和市场成败的最重要指标之一。在 Google，网页呈现速度慢 500 毫秒将丢失 20%的流量；在 Yahoo!，慢 400 毫秒将丢失 5%~9%的流量；在 Amazon，慢 100 毫秒将丢失 1%的交易量……反之，网站速度越快，越有利于用户汇聚、流量增长及交易量的上升。所以作为 Web 开发者，我们不会满足现状，要让网页变得更快更好。

本书作者 Steve Souders 在他上一部震惊业界的大作《高性能网站建设指南》中告诉我们，Web 的 80%~90%的性能由前端决定，并为提升 Web 性能设置了 14 条规则。本书中，Souders 与 8 位 Web 开发界顶级大牛通力合作，一起向我们传授让网站速度更快的思想和原则，以及相应的技术和工具。它是上一部作品的进阶之作，如果说上一部作品使 Web 开发者有机会实现高性能的网站以满足当前用户的需要，那么本书将帮助我们立足现在更着眼于未来，以实现超出用户期望的速度体验。

Web 的高速发展让开发者应接不暇，开发者应该把智慧、时间和精力放在用户最需要的地方，因此在做性能优化时不能盲目行动、捉小放大，需要事前有效评估找到重点，事后建立监控和保证机制，把有规律的、重复的、精确的工作交给机器去做。9 位大牛把众多的思想、原则、方法和自动化工具融汇到这本书中，让 Web 开发者有机会站在巨人的肩上，借助于全球业界的最新成果，用自己的智慧、行动和坚持开发出令人赞叹的 Web 产品来。

如果你稍加留心就会发现，这两本书副标题的变化很有趣，第一本书的副标题是“前端工程师技能精髓”，而本书的是“Web 开发者性能优化最佳实践”。Web 性能优化不只是前端工程师的工作，而是需要所有 Web 开发者协作完成。在实际工作中，如果没有人关注 Web 性能，那么，不管我们作为 Web 开发者的哪种角色都应该挺身而出勇于担当，不仅要担当建设高速网站的责任，还要担当 Web 性能优化的布道义务，团结一切可以团结的力量。这是我们所有 Web 开发者的使命。在口碑网我们已经开始这样做了，前端工程师与对此有兴趣的后端工程师和产品经理一起组建了一个虚拟的 Web 性能优化团队，在完成工作之余，一起学习、研究、分享、布道和推动性能优化项目。虽然刚刚开始，但是已经有了一些令人欣喜的成果，这就是团结的力量。

本书由口碑网前端团队的鄢学鼯（三七）、郑旭东（正旭）、刘志涛（鸽龙）、崔明达（灵佑）和姜凡（展炎）协作译成。翻译的过程也是自我学习、应用实践和总结提升的过程，我们深感本书的博大精深，同时也感到自己知识面的浅薄狭窄。我们把在学习和实践本书思想

及方法过程中收获的心得和相关资料作为译注补充在译文中，希望会对大家有所帮助。我们深信学习本书正是学习大牛思想、提升专业技能的最好途径之一，把它的精华最精确地传递给每位读者是我们现在的想法和最大动力。

感谢博文视点引进如此高品质的图书，这让中国的 Web 开发者可以从中受益，进而为更多用户提供更快速的 Web 使用体验。除了封面上列出的译者外，还有很多人为翻译这本书倾注了智慧和汗水。编辑徐定翔老师，包容了我们交稿的一次次延期，正是由于他的信任、帮助、支持和耐心指导才让我们能有这样一次边学习、边成长、边共享的宝贵经历。编辑白爱萍老师负责译稿的统稿编加，她的专业、仔细和辛勤工作保证了我们阅读本书时的良好体验。可爱的同事林枝萍（般若）、高嘉峻（伯灵）、全鑫（泉心）、闻明（阿苏）、沙峰（善朋）、曾焕数（合沙）、王维锋（元天）、严斌锋（邦彦）、何莹莹（冰浠）、周炼（灵落）等，给了我们热情的建议和细心的校对，有效地提升了本书的品质。要特别感谢淘宝网前端团队的赵泽欣（小马）、王保平（玉伯）和郑叶飞（圆心）3 位同学，他们在百忙之中帮我们校对重要章节，解决疑难问题。如果没有家人和同伴的支持，是很难做好翻译的，在此向所有关心和支持我们的朋友表示衷心的感谢！

口碑网前端团队

三七 正旭 鸽龙 灵佑 展炎

2010.02.25 于杭州

致谢
Credits

本书中有 6 个章节是由以下几位作者完成的。

Dion Almaer 是 Ajax 社区的先驱 Ajaxian.com 的创建人之一，目前在 Mozilla 领导一个新团队研发基于 Web 的开发者工具（译注 1），他已为之努力多年。在这本书里他与老搭档 Ben Galbraith 再次合作，无论是之前在 Ajaxian 还是在现在的 Mozilla（译注 2），他俩一直合作得很愉快。自 Gopher 协议发布以来，Dion 编写过相当多的 Web 应用程序，经常在世界各地演讲，发表了大量文章并出版过一本书。当然你也可以去阅读他的博客（*http://almaer.com/blog*），那儿有更多内容，你甚至能了解他的生活及对世界的思考。

Douglas Crockford 出生于明尼苏达州，6 个月后因无法忍受当地的严寒而被家人带离该州。Douglas 自发现计算机这好玩意儿后就抛弃了原本很有前途的电视机研发职业。他爱好广泛、博学多才，涉足过学习系统、小型商务系统、办公自动化、游戏、互动音乐、多媒体、移动娱乐、社会化系统和编程语言等多个领域。出人意料的是他还发明了史上最烂编程语言——Tilton。他最为人知的贡献是发掘了 JavaScript 的精华部分，这非常重要，却也让人意想不到。他的另一重大贡献是发现数据交换格式 JSON（JavaScript Object Notation，*http://www.json.org/json-zh.html*）。最近他正致力于将网络变成安全可靠的软件分发平台，并已着手实施这一计划。

Ben Galbraith 是 Mozilla 开发者工具的联合总监，也是 Ajaxian.com 的创建人。Ben 对商业和技术一直都极感兴趣，6 岁那年就写出了他的第一个计算机程序，10 岁开创第一个商业项目，12 岁投身于 IT 业。他在世界范围内开展了数百次的技术演讲，组织过多场技术研讨会，并与他人合著图书超过 6 本。Ben 的职业生涯也相当精彩丰富，从商业管理到技术工作，从 CEO 到 CIO 到 CTO 到首席软件设计师，从医疗到出版到媒体到制造业到广告到软件产业，角色众多、从业甚广。现在他和妻子及 5 个孩子住在加州的 Palo Alto。

译注 1：即大名鼎鼎的 Bespin，详情请见 *https://mozilalabs.com/bespin/*。

译注 2：不过 Dion Aimaer 和 Ben Galbraith 已在 2009 年 9 月 25 日宣布加入 Palm，担任其开发人员关系团队经理。

Tony Gentilcore 是 Google 公司的软件工程师，Google 首页和搜索结果页能快速呈现在用户面前就是他的杰作。他认为编写 Web 性能工具和研究相关技术是件非常愉悦的事。Tony 也是广受欢迎的 Firefox 扩展 Fasterfox 的开发者。

Dylan Schiemann 是 SitePen.com 的 CEO，也是 Dojo 工具包的编写者之一，还是开放网络（Open Web）的技术和分析专家。Dojo 是一个开源的用于快速构建 Web 站点和应用的 JavaScript 工具包。在他的引导下，SitePen 从一个小型研发公司成长为如今业界领先的开发工具供应商，拥有众多资深软件工程师，为用户提供专业的咨询服务、一流的培训及外包业务。Dylan 致力于研发的坚定信念已使 SitePen 成为诸如 Dojo、cometD、Direct Web Remoting (DWR)和 Persevere 等开拓性开源 Web 开发工具包和框架的主要贡献者和创作者。在 SitePen 之前，他还为 Renkoo、Informatica、Security FrameWorks 和 Vizional Technologies 等公司开发过 Web 应用。他还是 Comet Daily，LLC 的合伙创始人及 Dojo 基金董事会成员和 Aptana 咨询委员会成员。Dylan 在加州大学洛杉矶分校获物理化学硕士学位，在蒂尔大学取得数学学士学位。

Stoyan Stefanov 是 Yahoo！前端工程师，主要关注 Web 应用的性能。他也是性能扩展工具——YSlow2.0 的架构设计师和 Smush.it 图像优化工具研发者之一。Stoyan 同时还是演说家、作家（Packt 出版社出版了他的《Object-Oriented JavaScript》），他的博客地址为 *http://phpied.com*、*http://jspatterns.com* 和 YUIblog（*http://yuiblog.com*）。

Nicole Sullivan 是一位 Web 技术的布道者、前端性能优化顾问和 CSS 专家。她创建了开源项目——面向对象的 CSS（Object-Oriented CSS），它解决了如何规划 CSS 以应对成千上万的访问者的难题。她还与 W3C 合作重新设计其测试版，也是 Smush.it 图像优化工具的研发者。她对 CSS、Web 标准和可扩展的大型商业网站前端架构充满激情。Nicole 在世界各地的会议上发表关于性能优化的演讲，最近的会议是 The Ajax Experience、ParisWeb 和 Web Directions North。她的博客是 *http://stubbornella.org*。

Nicholas C. Zakas 是《Professional JavaScript for Web Developers》（第二版，Wrox 出版社，中文版《JavaScript 高级程序设计》）的作者，也是《Professional Ajax》（第二版，Wrox 出版社，中文版《Ajax 高级程序设计》）的合著作者。Nicholas 是 Yahoo!主页的首席前端工程师并参与了 YUI 的开发。他的博客地址是 *http://www.nczonline.net*。

前言
Preface

警惕（Vigilant）：警觉地关注，主要是为了避免危险。

任何人阅读了本书或前一本书《高性能网站建设指南》，都会理解网站速度缓慢的危害性：饱受挫折的用户、负面的品牌认知、膨胀的运营开支及缩水的财务收益。我们必须不断努力使网站更快。当我们取得进步的时候，同时也意味着开始落后了。我们必须处处留心，每一次修复 Bug、增加新功能和升级系统都可能对网站的速度产生影响。我们必须时时关注，今天实现的性能提升很可能明天就会失效。因此，必须保持警惕。

守夜（Vigil）：节日的前一刻仍然保持警觉。

根据 vigil 的拉丁语义，守夜意味着我们应该保持警觉直到庆祝真正开始。网站的确可以变得更快，甚至飞快，我们可以为自己关心和专注所获得的成果庆祝。这是真的！使网站速度更快是可以实现的。本书介绍的技术已经让一些风靡全球的网站的加载时间减少了 60%，它们同样也将造福小型网站，而最终是用户从中受益。

蜘蛛侠（Vigilante）：自我任命的正义使者。

作为开发者，我们有责任去捍卫用户的利益。你应该在自己的网站上充当性能的布道者，实施这些技术，并与同事分享这本书，为更快的用户体验而战。如果你的公司没有人专门负责性能，那么任命自己来担当这种角色吧。**性能蜘蛛侠**——听起来很不错。

本书的组织
How This Book Is Organized

这本书是对我第一本书《高性能网站建设指南》（O'Reilly 出版）的跟进。在那本书中，我为 Web 性能提升设置了 14 条规则：

- 规则 1：尽量减少 HTTP 请求。
- 规则 2：使用 CDN。
- 规则 3：添加 Expires 头。
- 规则 4：采用 Gzip 压缩组件。

- 规则 5：将样式表放在顶部。
- 规则 6：将脚本放在底部。
- 规则 7：避免 CSS 表达式。
- 规则 8：使用外部的 JavaScript 和 CSS。
- 规则 9：减少 DNS 查询。
- 规则 10：精简 JavaScript。
- 规则 11：避免重定向。
- 规则 12：删除重复的脚本。
- 规则 13：配置 ETag。
- 规则 14：使 Ajax 可缓存。

我称之为“规则”是因为它们易于实施。看看 2007 年 3 月份美国 10 大网站（注 1）的统计数据：

- 两个站点使用 CSS sprites。
- 26%的资源添加了 Expires 头。
- 5 个站点压缩了 HTML、JavaScript 和 CSS。
- 4 个站点精简了 JavaScript。

2009 年 4 月份的统计数据显示这些规则渐渐地获得了青睐：

- 9 个站点使用了 CSS sprites。
- 93%的资源添加了 Expires 头。
- 10 个站点都压缩了 HTML、JavaScript 和 CSS。
- 9 个站点精简了 JavaScript。

《高性能网站建设指南》中的规则现在依旧适用，并且大多数 Web 公司都应该从这些规则做起。我们已经有所进展，但在这套原始规则之上还有很多工作要做。

然而，Web 没有停滞不前，我们必须迎头赶上。虽然《高性能网站建设指南》中的 14 条规则依旧适用，但网页内容和 Web 2.0 应用程序的增长带来了一系列新的性能挑战。本书提供了开发者所需的能为新一代网站提速的最佳实践。

本书的所有章节由 3 部分组成：JavaScript 性能（第 1~7 章）、网络性能（第 8~12 章）和浏览器性能（第 13~14 章）。附录介绍了一组最好的性能分析工具。

注 1：依据 Alexa 排名，它们是 AOL、eBay、Facebook、Google Search、Live Search、MSN.com、MySpace、Wikipedia、Yahoo!和 YouTube。

以下 6 章是由特约作者完成的：

- 第 1 章 **理解 Ajax 性能**，由 Douglas Crockford 完成。
- 第 2 章 **创建快速响应的 Web 应用**，由 Ben Galbraith 和 Dion Almaer 完成。
- 第 7 章 **编写高效的 JavaScript**，由 Nicholas C. Zakas 完成。
- 第 8 章 **可伸缩的 Comet**，由 Dylan Schiemann 完成。
- 第 9 章 **超越 Gzip 压缩**，由 Tony Gentilcore 完成。
- 第 10 章 **图像优化**，由 Stoyan Stefanov 和 Nicole Sullivan 完成。

这些作者都是上述各个领域的专家，我想请你直接听到他们的声音。为了便于识别这些章节，我在每章的开头都注明了特约作者的名字。

JavaScript 性能
JavaScript Performance

在对当今网站的分析中，我经常能体会到 JavaScript 是实现高性能 Web 应用程序的关键，所以我以如下章节作为本书的开始。

第 1 章《**理解 Ajax 性能**》由 Douglas Crockford 所著。Doug 介绍了 Ajax 如何改变浏览器和服务器的交互方式，以及开发者如何理解这种新的关系从而能恰当地抓住提升性能的时机。

第 2 章《**创建快速响应的 Web 应用**》是由 Ben Galbraith 和 Dion Almaer 共同完成的，它使提升 JavaScript 性能回归到真正重要的事情上：用户体验。如今的 Web 应用程序在按钮点击事件发生后需要调用复杂的函数，必须在迫使浏览器能胜任的基础上去评估。那些理解代码对响应时间的影响程度的程序员更容易编写成功的 Web 应用程序。

我编写了接下来的 4 章。它们都集中在处理 JavaScript 的机制上——封装和加载的最佳方式，以及将其插入页面的最佳位置。第 3 章《**拆分初始化负载**》介绍了当今许多 Web 应用程序面临的处境：在页面初始化时下载一个庞大的 JavaScript，这不仅会阻塞渲染还会阻塞后续资源的下载。为了更高效的加载，对这个庞大的 JavaScript 进行拆分是关键。

第 4 章和第 5 章相辅相成。在眼下流行的浏览器中，外部脚本会阻塞页面中剩下的所有资源。第 4 章《**无阻塞加载脚本**》介绍了在加载外部脚本时如何避免这些造成阻塞的陷阱。当行内脚本依赖外部脚本时，异步加载它们是一个挑战。幸运的是，有几种技术可以整合行内脚本和它们所依赖的异步加载的脚本。这些技术在第 5 章《**整合异步脚本**》做介绍。第 6 章《**布置行内脚本**》介绍了应用行内脚本的最佳性能实践，特别是它们对并发加载的影响。

我认为 Nicholas C. Zakas 所写的第 7 章《**编写高效的 JavaScript**》是 Doug 所写的第 1 章的补充。Doug 介绍了 Ajax 的全貌，而 Nicholas 则专注于加速 JavaScript 的几种具体技术。

网络性能
Network Performance

Web 应用程序和桌面应用程序不同——每次使用它们时都必须通过互联网进行下载。Ajax 的采用产生了一种服务端和客户端数据通信的新形式。毫不夸张地说，在一些互联网速度尚不理想的新兴市场中，Web 产业还蕴含着一些巨大的增长机遇。所有这些因素都突显了提升网络性能的必要性。

在第 8 章《**可伸缩的 Comet**》中，Dylan Schiemann 介绍了一个超越 Ajax 的架构，它为诸如聊天和文档协助之类的实时应用提供了大容量、低延迟的实时连接。

第 9 章《**超越 Gzip 压缩**》介绍了为何打开 Gzip 压缩并不足以保证用最佳的方式发送网站内容。Tony Gentilcore 透露了一个鲜为人知的现象，全球有 15%的互联网用户因无法使用 Gzip 压缩而严重降低了网络性能。

Stoyan Stefanov 和 Nicole Sullivan 联合贡献了第 10 章《**图像优化**》。它为该主题提供了彻底的解决方案。本章回顾了所有的流行图像格式，展现了很多图像优化技术，介绍了如何对图像压缩工具进行选择。

剩下的章节由我完成。第 11 章《**划分主域**》提醒我们当今流行浏览器和下一代浏览器的连接限制。它包括了通过多个域来成功划分资源的技术。

第 12 章《**尽早刷新文档的输出**》全面地介绍了在完整的 HTML 文档到达之前使用块编码渲染页面的好处与缺陷。

浏览器性能
Browser Performance

iframe 是一种在网页中嵌入第三方内容的方便且常用的技术，但它们会带来开销。第 13 章《**少用 iframe**》介绍了 iframe 的缺点并提供了一些替代方案。

第 14 章《**简化 CSS 选择符**》介绍了复杂选择符影响性能的原理，并在客观的分析后指出使用时最需要注意的情况。

附录《**性能工具**》介绍了我推荐的用于分析网站和找出急需改进性能之处的工具。

本书约定
Conventions Used in This Book

本书使用下列排版约定：

等宽（`Constant width`）

表示命令、选项、开关、变量、属性（attribute）、键值、函数、类型、类、名字空间、方法、模块、属性（property）、参数、值、对象、事件、事件处理程序、XML 标签、HTML 标签、宏、文件内容和命令的输出。

等宽粗体（**`Constant width bold`**）

表示实际中应由用户输入的命令或其他文本。

这个图标表示提示、建议或一般说明。

这个图标表示警告或提醒。

意见和问题
Comments and Questions

我们已尽力核验本书所提供的信息，尽管如此，仍不能保证本书完全没有瑕疵，而网络世界的变化之快，也使得本书永不过时的保证成为不可能。如果读者发现本书内容上的错误，不管是赘字、错字、语意不清，甚至是技术错误，我们都竭诚虚心接受读者指教。如果您有任何问题，请按照以下的联系方式与我们联系。

奥莱利技术咨询（北京）有限公司
北京市 西城区 西直门 南大街2号 成铭大厦C座807室
邮政编码：100080
网页：*http://www.oreilly.com.cn*
E-mail：*info@mail.oreilly.com.cn*

O'Reilly Media, Inc
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)

707-829-0515 (international/local)
707-829-0104 (fax)

与本书有关的在线信息如下所示：

http://www.oreilly.com/catalog/9780596522308（原书）
http://www.oreilly.com.cn/book.php?bn= 9787121105449（中文版）

北京博文视点资讯有限公司（武汉分部）

湖北省 武汉市 洪山区 吴家湾 邮科院路特1号 湖北信息产业科技大厦1402室
邮政编码：430074
电话： (027)87690813　　传真：(027)87690813转817
读者服务网页：*http://bv.csdn.net*
E-mail：
reader@broadview.com.cn（读者信箱）
bvtougao@gmail.com（投稿信箱）

代码示例
Using Code Examples

一般情况下，你无须征求我们同意即可在程序或文档中使用本书的代码，除非复制使用大量代码。比如，写程序时使用几段本书的代码无需许可；将书中的示例刻录成光盘分发或销售则需要许可。援引示例代码来回答问题也不用许可，但在你的产品文档中使用大量的示例代码则需要许可。

我们欢迎但不强求注明引用注释。一条引用注释通常包括标题、作者、出版者和国际标准书号（ISBN）。例如：“Even Faster Web Sites, by Steve Souders. Copyright 2009 Steve Souders, 978-0-596-52230-8。”

如果你觉得自己对示例代码的使用超出了合理情况或以上许可的范围，请随时联系我们*permissions@oreilly.com*。

致谢
Acknowledgments

首先我要感谢本书的特约作者：Dion Almaer、Doug Crockford、Ben Galbraith、Tony Gentilcore、Dylan Schiemann、Stoyan Stefanov、Nicole Sullivan 和 Nicholas Zakas。他们成就了这本特别的书。他们每个人都是极具实力的专家，其中大多数人都自己写作过图书。他们的宝贵经验，为我们带来了一些独特的好东西。

我要感谢所有的审阅者：Julien Lecomte、Matthew Russell、Bill Scott 和 Tenni Theurer。此外特别要感谢 Eric Lawrence 和 Andy Oram。Eric 校对了本书和《高性能网站建设指南》。他在两本书中都提供了极其深入且专业的反馈。Andy 是我的前一本书《高性能网站建设指南》的编辑，他负责改善本书，使每一句、每一节、每一章都畅通易读，相比别人而言他的工作可谓辛苦至极。

特别感谢我的编辑 Mary Treseler。很多编辑都经历过与多位作者协作整理同一本书的挑战。我很高兴她接手了这个项目，并帮助指导我们将大量想法顺利地呈现在你手上的这本书中。

在谷歌我与一群热爱研究 Web 性能的人一块工作。Tony Gentilcore 是 Fasterfox 的创造者和第 9 章的作者。他也是我的同事，我们一天得讨论好几次关于 Web 性能的看法。Steve Lamm、Lindsey Simon 和 Annie Sullivan 经常与我一同工作，他们热衷于性能优化。据我所知，其他对 Web 性能有贡献的“Googler”包括 Jacob Hoffman-Andrews、Kyle Scholz、Steve Krulewitz、Matt Gundersen、Gavin Doughtie 和 Bryan McQuade。

本书中许多见解来源于我在 Google 以外的朋友。他们知道，如果给我一个改善性能的思路，这个思路很可能会见诸于一本书或一篇博文。这个性能亲友团成员包括 Dion Almaer、Artur Bergman、 Doug Crockford、Ben Galbraith、Eric Goldsmith、Jon Jenkins、Eric Lawrence、Mark Nottingham、Simon Perkins、John Resig、Alex Russell、Eric Schurman、Dylan Schiemann、Bill Scott、Jonas Sicking、Joseph Smarr 和 Tenni Theurer。

这份名单难以面面俱到，对此我感到很抱歉。感谢他们抽出宝贵时间给我发送电子邮件或在会议上和我交流，他们的经验教训与成功案例令我坚持下去，重要的是我知道了有这么多人都在致力于提高 Web 的速度。

谢谢你们，我的父母，你们一直以来以身为作家的儿子为豪。尤其要谢谢我的妻子和 3 个女儿，我向你们保证现在开始休假。

目录

Contents

第 1 章

理解 Ajax 性能

Understanding Ajax Performance

Douglas Crockford

过早的优化是万恶之源。
——Donald Knuth（译注 1）

1.1 权衡

Trade-offs

设计和构造计算机程序要经过成千上万次抉择，每一次都代表着一种权衡。在进行重大抉择时，每个选择都有其显著的优缺点，让人难以取舍。我们都期望收益最大化，损耗最小化。或许，人生的终极权衡是：

> 我想去天堂，但我不想死。

然而现实情况往往是：

> 时间、质量和成本，三选二。

这个项目三角形（Project Triangle）（译注 2）说明：即使在最理想的情况下，多快好省的事也基本不存在，所以权衡无处不在。

计算机程序中，选择算法要权衡运行时间和内存，会为抢占市场而牺牲代码质量。这些权衡对增量式开发（译注 3）的最终效果影响巨大。

译注 1：此语出自 Donald Knuth 在 1974 发表的《Structured Programming with Goto Statements》（*http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf*）中说“忘记那些微不足道的性能提升吧，97%的情况下，过早的优化是万恶之源。但我们不应该错过那关键的 3%”。Donald Knuth 的中文名叫高德纳，他被誉为现代计算机科学的鼻祖，其巨著《计算机程序设计艺术》被誉为计算机科学界的“圣经”。更多信息请参阅 *http://zh.wikipedia.org/wiki/*高德纳。

译注 2：它是工程中的一种项目三角模型，快是指交付产品的时间，好是指最终产品的质量，便宜是指设计和生产产品的成本。更多内容请看 *http://en.wikipedia.org/wiki/Project_triangle*。项目三角形容易和项目管理三角形混淆（Project management triangle），后者中三个角分别是时间、成本和范围，更多内容请看 *http://en.wikipedia.org/wiki /Project_management_triangle*。

译注 3：增量式开发（incremental development），是一种任务调度和分阶段策略，系统的各个部分以不同的时间和速度进行开发，完成之后进行整合。它并不强调是否采用迭代式开发和瀑布式开发这两种再加工策略。取代递增式开发的另一个选择是使用大爆炸集成方式（big bang integration）来开发整个系统。更多内容请看：*http://en.wikipedia.org/wiki/Iterative_and_incremental_development*。

每一次接触代码，我们都在权衡改善代码质量和避免产生 Bug 的可能性。关注程序性能时，我们必须反复斟酌所有权衡因素。

1.2 优化原则

Principles of Optimization

优化的目的是希望降低程序的整体开销。虽然在程序中有许多因素可以优化，但通常人们会认为这个开销就是程序的执行时间。其实我们更应该把重点放在对程序整体开销影响最大的那部分。

例如，假设我们通过性能分析得到程序的 4 个模块的开销:

模块	A	B	C	D
开销	54%	4%	30%	12%

即使能以某种方式将模块 B 的开销减少一半，其实也仅降低了整体开销的 2%。如果能将模块 A 的开销减少 10%，却会得到更好的效果。所以优化那些开销不大的组件收效甚微。

分析应用程序和分析算法密切相关。仔细观察程序的执行时间后，我们会发现其大部分时间都消耗在循环上。所以，优化那些只执行一次的代码得到的回报微不足道，但优化内部循环的好处却能达到立杆见影的效果。

例如，若循环的开销和迭代次数成线性关系，那么我们则可以用 O(n)来表示，其性能曲线如图 1-1 所示。

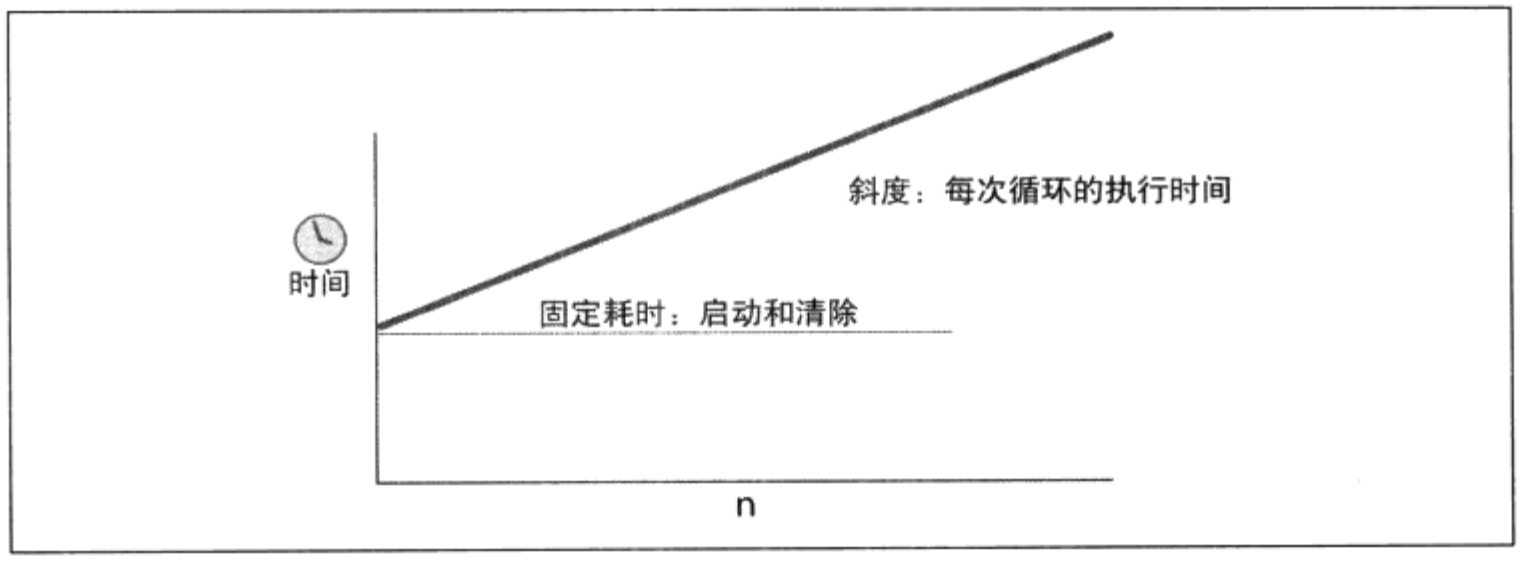


图 1-1：循环性能

每次迭代的执行时间反映在线段的斜度上：开销越大，斜度越陡。循环的固定开销决定了起点的高度，但减少固定开销通常没有什么好处。有时，如果能减少每次增量的开销，即便固定开销有所增加也是有益的。这可能是一个很好的权衡。

另外，在执行时间上有 3 条错误轴线，我们的时间线不能和它们相交（见图 1-2）。第 1 条是**低效线**，与之相交会降低用户的关注度，并使用户烦躁不安。第 2 条是**受挫线**，与之相交则意味着用户会意识到自己在被迫等待，这会导致他开始考虑其他的事情，比如试试其

他的 Web 应用程序——包括我们竞争对手的产品。第 3 条线是**失败线**，这是指由于应用崩溃或浏览器自己产生一个对话框，告诉用户应用失败需要采取行动，这时用户只好刷新或关闭浏览器。

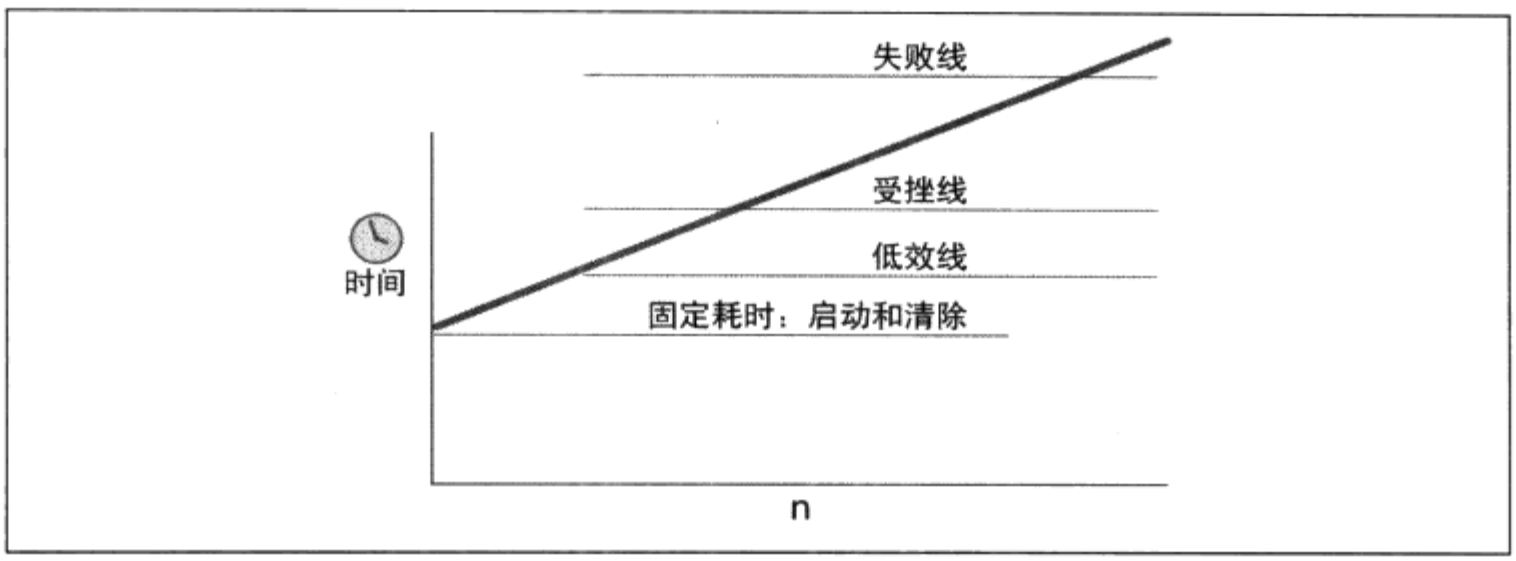


图 1-2：错误轴线

避免与错误轴线相交有 3 种方法：减少每次迭代的开销、减少迭代的次数或重新设计应用程序。

循环嵌套越多，可优化的选择越少。如果循环的开销是 O(n log n)、O(n2)或更糟，减少每次迭代的时间没有任何效果（见图 1-3）。唯一有效的选择是减少 n 或更换算法，只有在 n 非常小时，调整每次迭代的开销才会有效果。

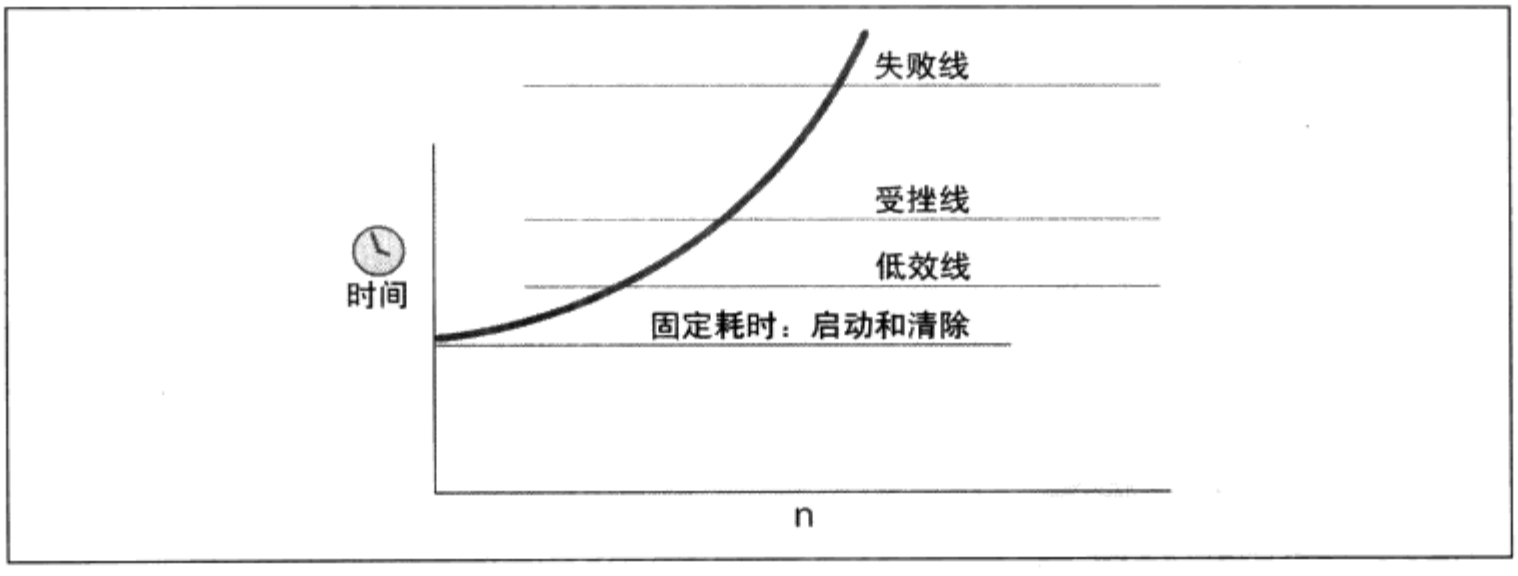


图 1-3：嵌套循环的性能

必须正确地设计程序。如果程序不符合要求，即使它速度很快也没什么意义。无论如何，尽早在开发周期中测试程序是否有性能问题非常重要。在测试 Web 应用时，尽可能模仿真

实的用户环境，使用低端机器和低速网络来测试，因为在开发人员的高配置环境中得出的测试结果很可能会掩盖性能问题。

1.3 Ajax

重构代码能明显降低其复杂性，所以对代码进行优化和其他修改更易产生效益。例如，依据 YSlow 规则来重构代码会对网页的加载时间产生巨大影响（参见 *http://developer.yahoo.com/yslow/*）。

即便如此，由于网页的大小和复杂性，要想把 Web 应用控制在低效线之下还是非常困难。网页通常内容多、代码量大且组成部分众多，在这之间的跳转会带来巨大的开销。因此对于连续页面之间差别很小的应用而言，使用 Ajax 技术能带来显著的改善。

传统方式下用户操作的结果是请求替换页面，而 Ajax 的实现方式是：发送数据包到服务端（通常编码为 JSON 文本），然后服务端返回另一个数据包（也是典型的 JSON 编码）来响应，最后 JavaScript 程序使用这些数据来更新浏览器的显示。这样做，数据的传输量、用户操作和可视化反馈的时间间隔都明显减少，服务器和浏览器需要做的工作量也减少了。不幸的是，Ajax 程序员需要做的工作量很可能因此而增加。这也是需要权衡的因素之一。

Ajax 应用的架构明显不同于大多数其他类型的应用，因为它是两个系统共有的。如果想通过 Ajax 方法对性能有积极的影响，采取正确的分工至关重要。数据包应该尽可能地小。应用程序应构建成浏览器端和服务端的会话，这两个部分之间用一种简单且富于表达能力的共用语言进行交流。实时数据传输的方式能让应用在浏览器端保持较小的 n 值，这有助于循环持续快速运行。

Ajax 应用的常见错误是把所有应用数据都发送给浏览器，这会再次引入 Ajax 本应避免的延迟问题。这类错误也会增加浏览器需要处理的数据量，增大 n 值，从而更加降低性能。

1.4 浏览器
Browser

浏览器并非为应用平台而设计，因此编写 Ajax 应用程序极具挑战性。脚本语言和文档对象模型（DOM）原本只设计为支持由简单表单所组成的应用。令人惊讶的是，浏览器已具有足够的能力可能去实现复杂的应用，但不幸的是，它并未达到一切令人满意的地步，所以

难度真地非常高。使用 Ajax 库（例如 *http://developer.yahoo.com/yui/*）能缓解这种情况。Ajax 库能充分发挥 JavaScript 的表达能力把 DOM 提升到实用的水平，同时修复令应用无法在各个品牌浏览器中顺利运行的诸多障碍。

很不幸，DOM API 非常低效且令人费解。运行程序的最大开销往往是 DOM 而非 JavaScript。在 Velocity 2008 大会（译注 4）上，Microsoft Internet Explorer 8 团队就打开 Alexa 排名前 100 名的网站页面的时间消耗做了一次性能数据分享（译注 5）（注 1）。

活动	布局	渲染	HTML	调度	DOM	格式化	JScript	其他
开销	43.16%	27.25%	2.81%	7.34%	5.05%	8.66%	3.23%	2.5%

相比浏览器在其他事情上消耗的时间，运行 JavaScript 的开销是微不足道的。微软团队也提供了一个更能说明问题的 Ajax 应用示例：一个 email 进程的打开过程时间消耗的数据如下：

活动	布局	渲染	HTML	调度	DOM	格式化	JScript	其他
开销	9.41%	9.21%	1.57%	7.85%	12.47%	38.97%	14.43%	3.72%

脚本的开销依旧低于 15%，而处理 CSS 是最大的开销。即使你能像超人一样使脚本的速度提升一倍，用户也很难注意到。因此毫无疑问，理解 DOM 的奥秘并减少它的影响比试图给脚本提速效果更好。

1.5 哇！

Wow!

应用程序设计者趋向于在 Ajax 应用中添加酷炫特效。这样做是为了引起类似这样的反响："哇！我不知道浏览器可以做到这个。"当使用不当时，这些酷炫的特性会由于分散用户的注意力或强迫用户等待连续动画播放完毕从而降低他们的生产效率。误用酷炫特性也会导致不必要的 DOM 操作，从而带来令人吃惊的巨大开销。

酷炫特效应该只在确实能改善用户体验时才使用，而不应用于炫耀或弥补功能与可用性上的缺陷。

设计浏览器能胜任的事情。例如，在无限滚动的列表中查看数据库要浏览器存储并展示一

译注 4：Velocity 是由 O'Reilly 主办的一个旨在推进建设更好的互联网的会议，每年举行一次。内容包括如何给网页提速、如何构建高效可伸缩的基础设施、如何建立可靠的网站和服务等。Velocity 2008 的详情见 *http://en.oreilly.com/velocity2008/public/content/about*。

译注 5：原文给出的相关链接 *http://en.oreilly.com/velocity2008/public/schedule/detail/3290* 中该主题 "What's Coming in IE8"的幻灯片地址失效，但你可以通过 *http://www.slideshare.net/techdude/ie8-whats-coming* 看到它。

注 1：*http://en.oreilly.com/velocity2008/public/schedule/detail/3290*。

个非常大的数据集，这就超出了它的有效管理能力。更好的方法是根本无需滚动而采用有效的分页展示，反而能提供更好的性能且更易用。

1.6 JavaScript

大部分 JavaScript 引擎的优化是为了快速抢占市场而不是性能，所以 JavaScript 一直很自然地被认为是瓶颈。但通常情况下，瓶颈不是 JavaScript 而是 DOM，所以捣腾脚本收效甚微。

尽量避免捣腾脚本，要正确并清晰地编写程序。捣腾往往越整越乱，它只会让程序更容易产生 Bug。

幸运的是，竞争压力迫使浏览器制造商改进他们的 JavaScript 引擎的效率。这些改进将激起浏览器中新型应用的开发热潮。

避免使用那些传说中能让程序更快的奇技淫巧，除非你能证明它们将给应用带来明显的质量提升。大多数情况下，它们只会降低代码质量，而不会有多大提升。不要纠结于特定浏览器的怪癖，浏览器仍在发展，最终会支持更好的编码实践。

如果感觉必须捣腾，首先要评估。我们对程序真正开销的直觉往往是错的。只有通过评估，才能有把握对性能产生积极的影响。

1.7 总结

Summary

一切都是权衡。当我们做性能优化时，不要浪费时间去尝试为那些不消耗大量时间的代码提速。评估优先，拒绝任何不能提供良好效益的优化。

浏览器通常在运行 JavaScript 上花费的时间很少，绝大部分时间消耗在 DOM 上。你可以要求浏览器制造商提供更好的性能评估工具。

为质量编程。简洁、易读且条理分明的代码更易于正确理解、维护和优化。避免耍小聪明，除非可以证明它们能大幅提升性能。

善加利用 Ajax 技术能使应用程序运行得更快，用好它的关键是在浏览器和服务器之间建立平衡。Ajax 提供了取代页面替换的有效方案，使浏览器成为强大的应用平台，但它并不能保证这样做就一定会成功。浏览器平台非常具有挑战性，而你对于性能的直觉并不那么可靠。接下来的章节将帮助你理解如何制作更快的网站。

第 2 章

创建快速响应的 Web 应用

Creating Responsive Web Applications

Ben Galbraith 和 Dion Almaer

随着 Ajax 的兴起，网站性能不再局限于只是让网站快速呈现。越来越多的网站在加载完成后，借助 JavaScript 来实现页面内容的动态更新。这类网站的程序原理和传统的桌面客户端程序类似，同时对这些程序进行性能优化需要一套有别于传统网站的技术。

从更高的层面上来说，基于用户界面的需求，Web 应用和传统桌面应用有一个共同的目标：尽可能快地响应用户的输入。当浏览器响应用户发起的加载网站的请求时，它需要处理大量与响应相关的工作：与用户请求的站点建立网络连接，解析 HTML，请求相关资源等。详细地分析这个过程，我们可以找到方法优化页面，最大程度地提高渲染速度，但浏览器最终掌控着页面的加载和呈现。

网站响应用户输入的过程（前提是输入不会导致浏览器加载新页面）是 Web 开发者可以控制的。我们必须确保响应这些输入的 JavaScript 能快速地执行。为了更好地理解 Web 开发者对响应的控制程度，我们将用一点时间来说明浏览器用户界面的工作原理。

如图 2-1 所示，当用户和浏览器交互时，操作系统接收到和计算机连接的各种设备，如键盘或鼠标的输入，在判断哪个应用应该接收这些输入之后，将它们打包为**单独事件**并放置到该应用的事件队列中。

像所有 GUI 应用程序一样，浏览器按队列顺序完成其队列中单独事件的处理。它按照先进先出的顺序把它们从队列中取出，然后决定如何处理这个事件。通常，浏览器将基于这些事件做如下操作：对事件本身进行处理（例如显示菜单、浏览网页、显示设定画面等）或

执行网页自身的 JavaScript 代码（例如页面单击事件处理程序的 JavaScript 代码），如图 2-2 所示。

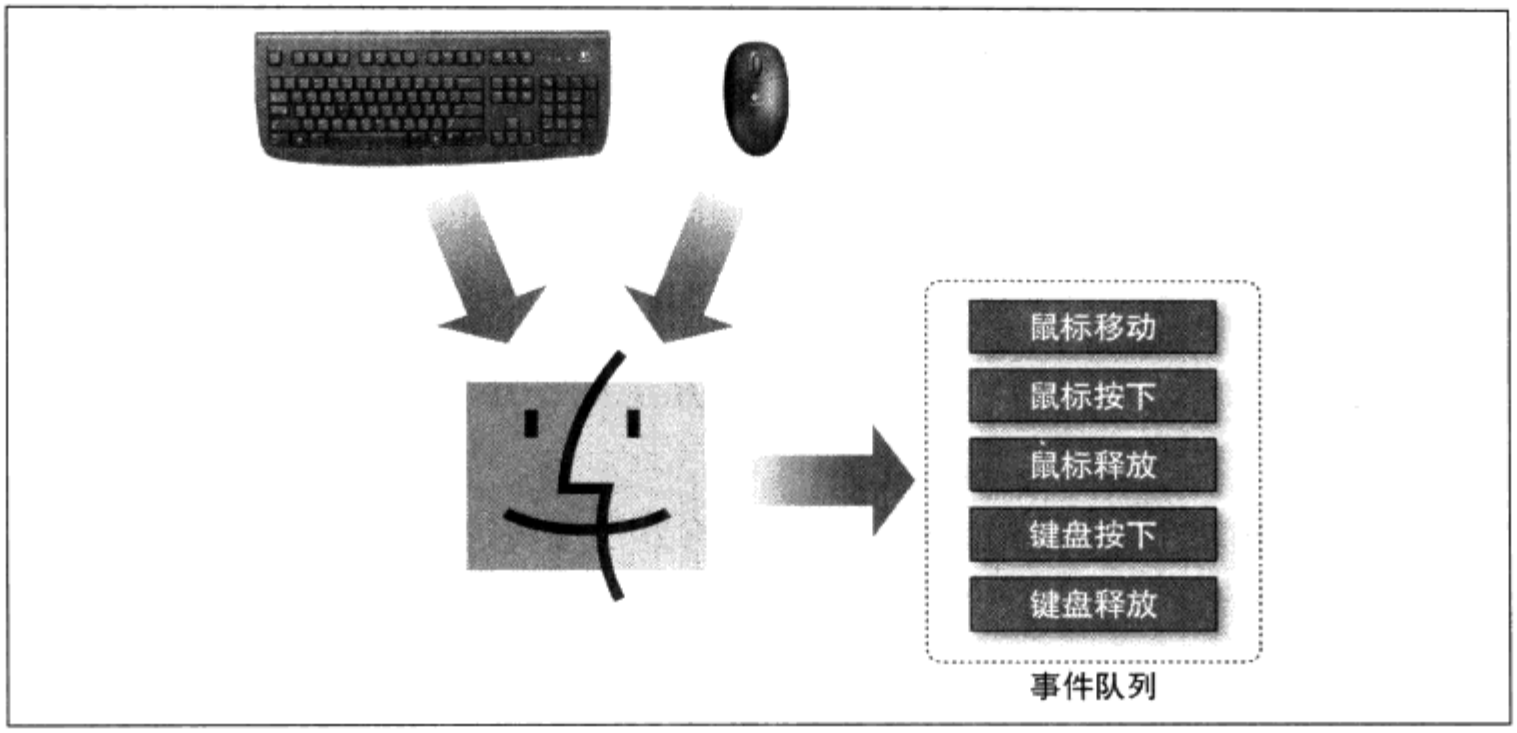


图 2-1：所有的输入都通过操作系统安排到事件队列中

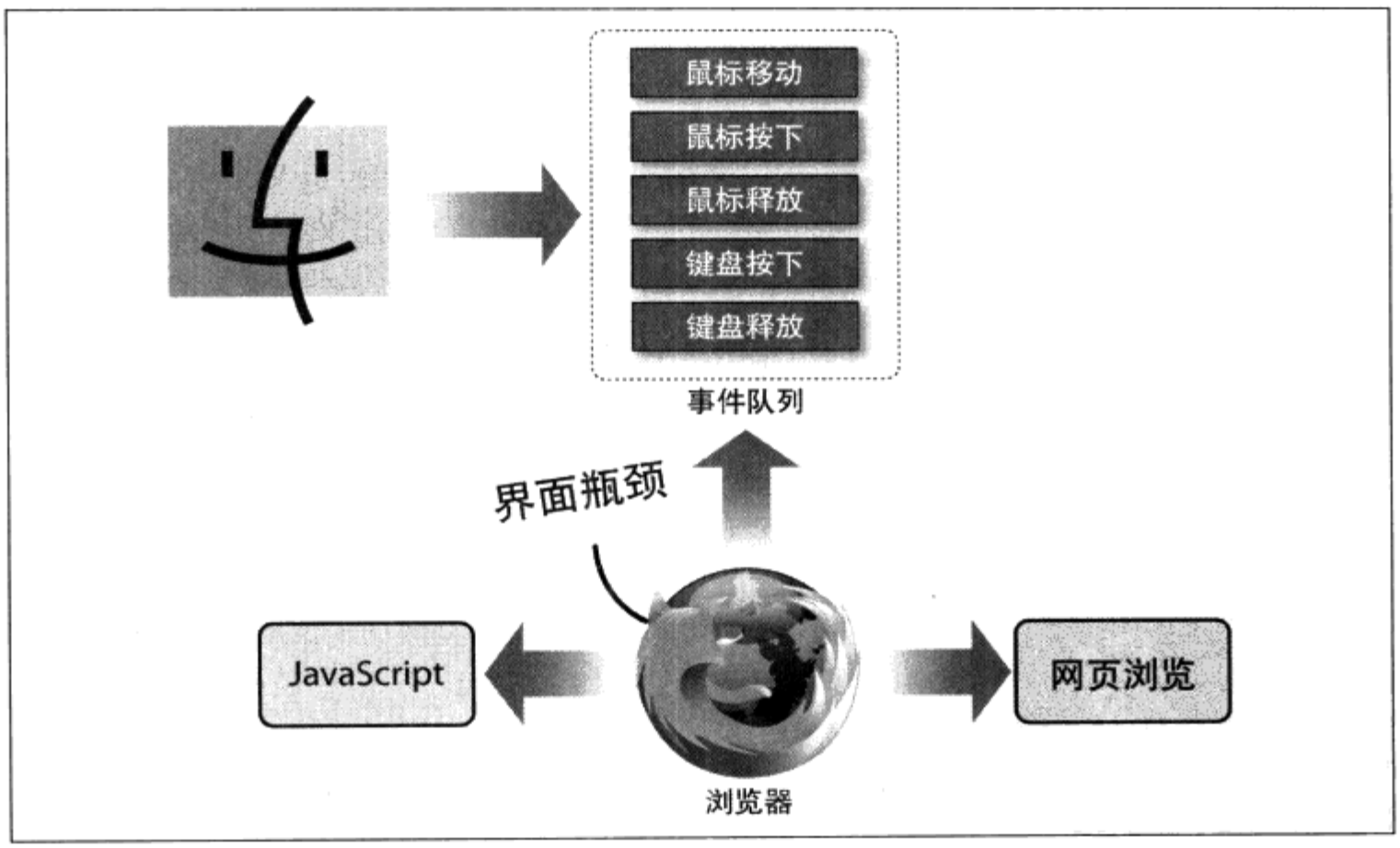


图 2-2：浏览器使用单线程处理队列中的事件和执行用户代码

这里有个需要注意的重要概念是，这个处理过程实质上是**单线程的**。也就是说，浏览器使用单线程从队列中取出事件，然后对事件本身进行处理（图 2-2 中的“Web browsing（网页浏览）”）或执行 JavaScript。因此，浏览器每次只能处理这些任务中的一个，并且任意一个任务都能阻止其他任务的执行。

浏览器在执行页面的 JavaScript 期间无法响应其他的用户事件。因此，尽可能快地执行完页面中所有的 JavaScript 是极其重要的，否则网页和浏览器本身可能变得非常缓慢甚至完全冻结。

注意：前面归纳的对浏览器和操作系统对输入处理和事件行为的讨论大致如此，但具体细节会有所不同。不考虑差异的话，所有浏览器执行单个页面中全部 JavaScript 代码（使用 Web Workers 除外，本章将在后面讨论它）的过程都是单线程的，这意味着本章提倡的开发者实践是完全适用的。

2.1 怎样才算足够快
What Is Fast Enough?

代码执行要“尽可能快”，这句话说起来容易，但有时代码就是需要时间去做一些事情。例如，加密算法、复杂的图形渲染和图像处理都是典型的耗时计算的例子，无论开发者为了使它们“尽可能快”付出多少努力都无济于事。

然而，正如 Douglas 在第 1 章所述，要创建一个快速响应的高性能网站，靠开发者对他们所写的每一段代码都进行优化是不可能的，也不应该如此操作。开发者应该仅仅优化那些不够快的地方。

因此，精确定义在上下文中提到的怎样才算“足够快”显得非常重要。幸运的是，已经有人为我们做了这个工作。

Jakob Nielsen 是 Web 可用性领域知名且备受推崇的专家，下面引用的内容（注 1）论述了“足够快”的问题：

> 基于 Web 应用的响应时间准则和所有其他应用一样。37 年来这些准则毫无变化，所以它们也不太可能因新技术的出现而发生改变。
>
> 0.1 秒：用户直接操作 UI 中对象的感觉极限。比如，从用户选择表格中的一列到该列高亮或向用户反馈已被选择的时间间隔。理想情况下，它也是对列进行排序的响应时间——这种情况下用户会感到他们正在给表格排序。

注 1：*http://www.useit.com/papers/responsetime.html*。

> **1 秒**：用户随意地在计算机指令空间进行操作而无需过度等待的感觉极限。0.2~1.0 秒的延迟意味着会被用户注意到，因此感觉到计算机处于对指令的“处理中”，这有别于直接响应用户行为的指令。例如：如果根据被选择的列对表格进行排序无法在 0.1 秒内完成，那么必须在 1 秒内完成，否则用户将感觉到 UI 变得缓慢且在执行任务中失去“流畅（flow）”的体验。超过 1 秒的延迟要提示用户计算机正在解决这个问题，例如改变光标的形状。
>
> **10 秒**：用户专注于任务的极限。超过 10 秒的任何操作都需要一个百分比完成指示器，以及一个方便用户中断操作且有清晰标识的方法。假设用户遭遇超过 10 秒延迟后才返回到原 UI 的情况，他们将需要重新适应。在用户的工作中，超过 10 秒的延迟仅在自然中断时可以接受，比如切换任务时。

换句话说，如果 JavaScript 代码执行时间超过 0.1 秒，页面将会给人不够平滑快捷的感觉；如果执行时间超过 1 秒，则会感到应用程序缓慢；超过 10 秒，那么用户将非常沮丧。这些就是用于界定“足够快”的明确准则。

2.2 测量延迟时间
Measuring Latency

现在你已经知道“足够快”的门槛了，下一步是研究如何才能测量 JavaScript 的执行速度，以确定它是否超出前面提到的范围（你可以自己决定希望页面达到的速度，而我们的目标是保持所有的界面延迟时间都小于 0.1 秒）。

通过人的观察来测量延迟时间虽不精确，却是最简单最直接的方法：只要在目标平台上运行一下应用程序，然后确定性能是否足够快。因为确保用户界面性能足够好的目的就是让使用者感到愉悦，所以这样做测试其实是个好方法（显然，没有人能准确地用精确到几点几秒来量化延迟，所以可以使用粗略的分类，比如“敏捷”、“缓慢”、“足够”等）。

然而，如果期望更精确的测量，你有两个选择：手动代码检测（**记录**）或自动代码检测（**性能分析**）。

手动代码检测十分简单明了。假设你在页面上注册了一个事件处理程序，如：

```
<div onclick="myJavaScriptFunction()"> ... </div>
```

一个简单的添加手动检测的方法是找到函数 `myJavaScriptFunction()`的定义，然后给它添加计时器：

```
function myJavaScriptFunction() {
    var start = new Date().getMilliseconds();

    // 这里是一些开销很大的代码

    var executionTime = stop - start;
    alert("myJavaScriptFunction() executed in " + executionTime +
        " milliseconds");
}
```

上述代码将产生一个弹出对话框来显示执行时间；1 毫秒是千分之一秒，所以 100 毫秒则表示前面提到的意味着“敏捷”门槛的 0.1 秒。

提示：许多浏览器都提供名为 `console` 的内置实例，它带有一个 `log()`函数（在 Firefox 中配合流行插件 Firebug 就可以使用它）；与 `alert()`相比，我们更倾向使用 `console.log()`。

有些工具能对代码执行时间进行自动测量，但它们通常用于各种不同的目的。这些性能分析器不是用来确定函数的精确执行时间的，而通常用于确定执行一组函数的相对时间；也就是说，它们是用来查找**瓶颈**或运行最慢的代码块的。

Firefox 的流行插件 Firebug（*http://getfirebug.com/*）包含了一个 JavaScrip 代码性能分析器，它生成的输出如图 2-3 所示。

Inspect Clear Profile

Console HTML CSS Script DOM Net Options

Profile (1272.599ms, 93169 calls)

Function	Calls	Percent	Own Time	Time	Avg	Min	Max	File
gcs()	1548	9.59%	122.051ms	122.051ms	0.079ms	0.035ms	0.506ms	js/dojo/...e/html.js (line 386)
paint()	33	9.57%	121.851ms	562.094ms	17.033ms	4.454ms	33.065ms	js/dojo/.../th/th.js (line 206)
measureText()	3226	7.88%	100.269ms	100.269ms	0.031ms	0.011ms	2.128ms	js/dojo/...canvas.js (line 50)
_getMarginBox()	516	6.84%	87.001ms	205.337ms	0.398ms	0.187ms	0.951ms	js/dojo/...e/html.js (line 761)
_obs()	516	6.56%	83.422ms	179.924ms	0.349ms	0.187ms	0.731ms	js/dojo/...e/html.js (line 1087)
_getMarginExtents()	516	5.72%	72.855ms	82.513ms	0.16ms	0.07ms	0.538ms	js/dojo/...e/html.js (line 711)
fillText()	668	4.52%	57.552ms	57.552ms	0.086ms	0.032ms	1.446ms	js/dojo/...canvas.js (line 40)
d()	2166	4.01%	51.037ms	115.366ms	0.053ms	0.025ms	0.294ms	js/dojo/...elpers.js (line 85)
paintChildren()	375	3.86%	49.178ms	427.395ms	1.14ms	0ms	27.104ms	js/dojo/.../th/th.js (line 377)
paint()	160	3.86%	49.112ms	59.56ms	0.372ms	0.266ms	0.671ms	js/dojo/...onents.js (line 222)
_docScroll()	516	2.85%	36.325ms	43.7ms	0.085ms	0.045ms	0.244ms	js/dojo/...e/html.js (line 1019)
getInsets()	26577	2.61%	33.17ms	65.184ms	0.002ms	0.001ms	0.104ms	js/dojo/.../th/th.js (line 327)
paint()	60	2.53%	32.196ms	107.061ms	1.784ms	0.731ms	3.229ms	js/dojo/...onents.js (line 719)
paintSelf()	80	2.27%	28.872ms	34.094ms	0.426ms	0.307ms	0.951ms	js/dojo/...onents.js (line 355)
paintSelf()	242	1.76%	22.413ms	22.413ms	0.093ms	0ms	0.768ms	js/dojo/...onents.js (line 177)
emptyInsets()	8151	1.75%	22.299ms	22.299ms	0.003ms	0.001ms	0.1ms	js/dojo/...elpers.js (line 103)
styleContext()	1613	1.72%	21.86ms	21.86ms	0.014ms	0.009ms	0.155ms	js/dojo/...onents.js (line 548)
request()	2	1.7%	21.679ms	21.743ms	10.872ms	10.3ms	11.443ms	js/dojo/...server.js (line 58)
paint()	92	1.43%	18.199ms	22.752ms	0.247ms	0.229ms	0.422ms	js/dojo/...onents.js (line 33)
_toPixelValue()	4128	1.39%	17.655ms	17.655ms	0.004ms	0.002ms	0.169ms	js/dojo/...e/html.js (line 397)
layout()	92	1.37%	17.475ms	116.292ms	1.264ms	0.48ms	5.746ms	js/dojo/...onents.js (line 86)
body()	1548	1.37%	17.406ms	17.406ms	0.011ms	0.004ms	0.051ms	js/dojo/...window.js (line 19)

>>>

图 2-3：Firebug 的性能分析器

“Time/时间”列表示 JavaScript 解析器在性能分析期间执行指定函数的总耗时。通常函数执行时会调用其他的函数，“Own Time”列表示特定函数自身而不包含任何它调用过的其他函数的耗时。

然而你可能会认为上述或其他类似与时间相关的列反映的就是对函数执行时间的精确测量，但事实上性能分析器也会受到外在的影响——类似物理学中的**观察者效应**（译注 1）：观察代码性能的行为改变了代码的性能。

性能分析器用两个基本行为策略来说明基础权衡关系：要么在要测量的代码中加入特定代码来收集性能统计信息（基本上像前面列表中那样自动地创建代码），要么在特定时间中实时检验具体执行的代码来监视运行时间。 这两种策略中，后者使需要分析的代码性能失真更少，但代价是获取到的数据质量会相对低一些。

Firebug 的性能分析器只能在 Firefox 自己的进程中执行，它有可能造成正在测量的代码损失性能，因此 Firebug 会使测量结果进一步失真。

不过，“Percent/百分比”证明了 Firebug 测量相对执行时间的用处：你可以在网页界面上执行一个高级别的任务（例如，单击发送按钮），然后执行 Firebug 的性能分析器去查看哪些函数消耗的执行时间最多，最后再集中在这些函数上面进行优化工作。

2.2.1 当延迟变得很严重时

When Latency Goes Bad

实际上，如果 JavaScript 代码使浏览器线程停滞过长的时间，大多数浏览器就会进行干预并通过提示给用户中断执行代码的机会，不过还没有标准的行为规定浏览器该如何裁决是否给用户这个机会。（关于个别浏览器的行为细节请看 *http://www.nczonline.net/blog/2009/01/05/what-determines-that-a-script-is-long-running/*。）

结论很简单：别把运行时间可能很长的低性能代码引入到网页中。

2.3 线程处理

Threading

一旦确定代码的性能不够好，下一步当然是去优化它。然而，有时执行任务的开销非常高，且无法神奇地把它优化得耗时更少。这种导致用户界面出现糟糕停滞的情形无法避免吗？就没有一个能让用户顺利执行的解决方案吗？

在这种情况下，传统的解决方案是使用**多线程**来把开销很大的代码从与用户交互的线程中

译注 1：所谓的“观察者效应”，指的是被观察的现象会因为观察行为而受到一定程度或很大程度的影响。详细内容请看 *http://baike.baidu.com/view/2016140.htm*。

剥离开来。在我们的设想中，这可以使浏览器持续处理来自队列中的事件从而保持界面的快速响应，与此同时，长时间运行的代码在另一个不同的线程上流畅地执行（由操作系统来负责保证浏览器的用户界面线程和后台线程平等地共享计算机资源）。

然而，JavaScript 并不支持多线程，所以无法使用 JavaScript 代码创建一个后台线程来执行开销很大的代码。再进一步分析，这种情况不可能在近期有所改变。

Brendan Eich 是 JavaScript 的创立者和 Mozilla 的首席技术官，他就这个问题表明了明确的立场（注 2）：

> 如果你是个武林高手，那你一定会对多线程系统进行 hack，而且大多数程序员会被你吓得退避三舍。但事实上他们并不会，他们只是像使用大多数其他强大的工具一样，获得最简单的单进程代码，然后把它塞到多进程的一个线程中，或者干脆引发其他的竞争状态（译注 2）。这样做，只是偶尔结果会很糟糕（译注 3），但如果太频繁了，皮之不存，毛将焉附，没有人会买账。
>
> 多线程在各个方面违反了抽象概念，主要是产生了竞争状态、死锁的风险和悲观锁定开销，并且它们无法横向扩展去处理未来超级内核的亿万次计算能力。
>
> 所以我对诸如“你将在什么时候为 JavaScript 添加多线程”之类问题的默认回答是：“等你死了之后！”

鉴于 Brendan 在业界和 JavaScript 未来发展上的影响力（这是相当大的），以及这个观点被广泛的传播，我们可以有把握地说多线程将不会很快地在 JavaScript 上实现。

但是我们也有备选方案。多线程的基本问题是不同的线程可以访问并修改相同的变量。当出现线程 A 要修改线程 B 正在修改的变量或类似情况时，这会导致各种各样的问题。你可能认为合格的程序员能直接避免这些问题，但事实证明，正如 Brendan 所说的那样，即使是我们当中最出色的程序员也会在这个地方犯非常可怕的错误。

2.4 确保响应速度

Ensuring Responsiveness

我们需要的是一种像多线程那样能多任务并发执行却没有线程之间相互侵入危险的方法。

注 2：*http://weblogs.mozillazine.org/roadmap/archives/2007/02/threads_suck.html*。

译注 2：竞争状态（race condition）是指在多线程（或多个处理过程）情况下，对有些共享资源进行混乱操作，导致整个处理过程变得混乱，引发 Bug。详情请参见 *http://en.wikipedia.org/wiki/Race_condition*。

译注 3：Brendan Eich 的原文此处链接了 Therac-25 案例（*http://en.wikipedia.org/wiki/Therac-25*），它是在软件工程界被大量引用的案例。Therac-25 是 Atomic Energy of Canada Limited 所生产的一种辐射治疗的机器。由于其软件设计时的缺陷，超过剂量的致命设定导致在 1985 年 6 月到 1987 年 1 月之间，已知的 6 件医疗事故中患者死亡或严重辐射灼伤。这 6 起事故是操作失误和软件缺陷共同造成的。事后的调查发现整个软件系统没有经过充分的测试，而最初所做的 Therac-25 分析报告中有关系统安全分析只考虑了系统硬件，没有把计算机故障（包括软件）所造成的隐患考虑在内。Therac-25 事件也因而唤醒了人们对软件开发工程化管理方法论的省思。

Google 在其著名的浏览器插件 Gears 中实现了这样的 API：WorkerPool API（译注 4）。它实际上允许浏览器的主 JavaScript 线程创建后台的“Worker”，在这些 Worker 启用时从浏览器线程中接收到一些简单的“信息”（如独立状态，而不是对共享变量的引用），并且在完成时返回一条信息。

Gears API 的经验使许多浏览器（例如：Safari4 和 Firefox3.1（译注 5））基于 HTML5 规范定义的通用 API 实现了对“workers”的原生支持，这个特性被称为“Web Workers（译注 6）”。

2.4.1 Web Workers

让我们来看一下如何利用 Web Worker API 对值进行解封装。如下列表展示了如何创建并启动 Worker：

```
// 创建并开始执行 worker
var worker = new Worker("js/decrypt.js");

// 注册事件处理程序，当 worker 给主线程发送信息时执行
worker.onmessage = function(e) {
    alert("The decrypted value is " + e.data);
}

// 发送信息给 worker，这里是指待解密的值
worker.postMessage(getValueToDecrypt());
```

现在让我们看一下设想中的 js/decrypt.js 的内容：

```
// 注册用来接收来自主线程信息的处理程序
onmessage = function(e) {
    // 获取传过来的数据
    var valueToDecrypt = e.data;

    // TODO：这里实现解密功能

    // 把值返回给主线程
    postMessage(decryptedValue);
}
```

在页面上任何开销可能很大的（例如，长时间运行）JavaScript 操作都应该委托给 Worker，因为这将使得应用程序快速运行。

2.4.2 Gears

如果你发现所使用的浏览器不支持 Web Worker API，那么还有一些替代方案。我们在上一

译注 4：见 *http://code.google.com/apis/gears/api_workerpool.html*。

译注 5：由于 Firefox3.1 的改进过多，在正式发布时版本变更为 3.5。

译注 6：关于 Web Workers 的详情下面有介绍，更详细的信息见 *https://developer.mozilla.org/En/Using_web_workers*。另外需要注意的是由 OpenSymphony 组织开发的，致力于组件化和代码重用的拉出式 MVC 模式 J2EEWeb 框架——WebWork，详情见 *http://www.opensymphony.com/webwork/*。

节中提到了 Google 的 Gears 插件，你可以利用它在 IE、Firefox 和 Safari 的早期版本上实现一些像 Web Workers 一样的功能。

Gears Worker API 与 Web Worker API 相似但并不完全一致。如下代码是使用 Gears 的 API 来重写前面的两段代码，开始为在主线程上执行的代码创建 Worker：

```
// 创建 Worker Pool，它会产生 Worker
var workerPool = google.gears.factory.create('beta.workerpool');

// 注册事件处理程序，它接收来自 Worker 的信息
workerPool.onmessage = function(ignore1, ignore2, e) {
    alert("The decrypted value is + " e.body);
}

// 创建 Worker
var workerId = workerPool.createWorkerFromUrl("js/decrypt.js");

// 发送信息到这个 Worker
workerPool.sendMessage(getValueToDecrypt(), workerId);
```

下面是 js/decrypt.js 的 Gears 版本：

```
var workerPool = google.gears.workerPool;
workerPool.onmessage = function(ignore1, ignore2, e) {
    // 获得传递过来的数据
    var valueToDecrypt = e.body;

    // TODO: 这里实现解封装功能

    // 把值返回给主线程
    workerPool.sendMessage(decryptedValue, e.sender);
}
```

深入 Gears

如果关注 Gears Worker Pool 的历史，你就会发现一些很有趣的事情，因为它来自一个非常注重实践的地方。Gears 插件是由 Google 的一个团队开发的，他们试图推动浏览器超越当前已有的功能（当然这是在 Google Chrome 出现之前，但即使有了 Chrome，Google 依旧期望尽可能多的用户使用它的 Web 应用来做一些很棒的事情）。

想象一下，如果你想开发离线 Gmail，你将需要什么呢？首先，你需要一种本地缓存和拦截文档的方法，这样当浏览器尝试访问 *http://mail.google.com/*时，它就可以返回你缓存的页面而不是正处于离线状态的提示信息。其次，你需要一种存储包括新旧两种邮件的方法。有很多方法都能实现这一点，但是由于 SQLite 是众所周知的，它已经内置在最新的浏览器中并和许多操作系统绑定，所以为什么不使用它呢？下面将说明其问题所在。

我们一直在谈论关于单线程浏览器的问题。现在想象一下如果在数据库中写入新信息或执行长查询，我们不能在数据库工作时冻结 UI，因为这个延迟可能是非常明显的。Gears

团队需要一种解决这个问题的方案。既然 Gears 插件可以做任何想做的事情，那么它也能轻松地解决 JavaScript 缺乏多线程的问题。既然并发性的需求是一个普遍存在的问题，那为什么不让外界具备这种能力呢？因此这个“Worker Pool”API 促使了 HTML5 标准“Web Workers”的诞生。

这两个 API 看起来有些微小的区别，这是因为 Web Workers 有几分像先驱产品 Gears API 的 2.0 版本，Gears 应该会在不久的将来支持标准的 API。已经有一些中间库建立了 Gears API 和标准 Web Worker API 之间的桥梁，它们甚至能在没有 Gears 或 Web Workers 的情况下工作（通过使用本章描述的 `setTimeout()` 方法）。

2.4.3 定时器
Timers

在 Gears 和 Web Workers 之前已有另外一种方法，它简单地把运行时间很长的计算拆成独立的区块，然后使用 JavaScript 的定时器控制其执行。例如：

```
var functionState = {};

function expensiveOperation() {
    var startTime = new Date().getMilliseconds();
    while ((new Date().getMilliseconds() - startTime) < 100) {
        // TODO: 它用如下方法执行开销很大的运算:
        // 它在迭代的语句块中执行 100 毫秒内完成的工作，
        // 然后修改本函数外部“functionState”中的状态。
        // 祝你好运。;-)
    }

    if (!functionState.isFinished) {
        // 退出 10 毫秒后再次执行 expensiveOperation;
        // 用较大的值进行试验，以在 UI 响应速度和性能上取得合适的平衡
        setTimeout(expensiveOperation(), 10);
    }
}
```

用前面描述的拆分模块运算的方法能实现快速响应的界面，但正如代码列表中的注释所表明的那样，用这种方法架构运算可能不那么简单直接（甚至不可行）。以这种方式使用 `setTimeout()` 的更多细节请看第 103 页的“使用定时器挂起”。

关于这种方法还有另一个基本问题。大多数现代计算机都是“多核”的，这意味着它们真正具备并发执行多个线程的能力（而以前的计算机仅仅通过快速切换任务来模拟并发）。像我们在代码列表中所做的那样，利用 JavaScript 手动地实现任务切换并不能充分地利用这种架构。强制多核中的一个来完成所有的运算，这是在浪费处理能力。

因此，在浏览器的主线程上实现运行长时间的计算并保持快速响应的界面是可行的，但使用 Worker 会更加容易且高效。

XMLHttpRequest

如果讨论线程处理没有涉及 XMLHttpRequest 将是不完整的，它是 Ajax 技术革新的著名推动者，简称“XHR”。使用 XHR，网页可以在 JavaScript 环境中发送信息和接收到完整的响应，这个壮举使得无须加载新页面就能实现丰富的互动性。

XHR 有两个基本的执行模式：同步和异步。在异步模式中，XHR 实质上就是一个拥有专用 API 的 Web Worker。事实上，结合正在制定的 HTML5 规范的其他功能，你能使用 Worker 重新创造 XHR 的功能。在同步模式中，XHR 的行为像在浏览器的主线程中执行它所有的工作一样，这会导致用户界面的持续延迟时间与 XHR 发送它的请求并解析来自服务端响应的整体耗时一样长。因此，千万不要使用 XHR 的同步模式，因为它会导致不可预知和不能容忍的用户界面延迟。

2.4.4 内存使用对响应时间的影响
Effects of Memory Use on Response Time

创建快速响应网页的另一个关键方面是：内存管理。JavaScript 像许多现代的高级语言一样把低级的内存管理抽象出来，绝大部分的运行环境都实现了垃圾回收（garbage collection，简称“GC”）。垃圾回收是一件很神奇的事情，它让开发人员摆脱了像会计一样关注单调乏味的细节而不是编程的感觉。

然而，自动内存管理是有开销的。当执行回收时，GC 实现中最复杂的几乎是“stop the world（译注 7）”，它们会冻结整个运行环境（包括我们正在调用的主浏览器 JavaScript 线程），直到遍历完整个创建对象的“堆”。在这个过程中，它们查找那些不再使用或能够回收未用内存的对象。

对于大部分应用程序而言，GC 是完全透明的；因为冻结运行环境的时间短到可以完全避开用户的注意。但随着应用程序内存占用的增加，遍历整个堆去查找不再使用的对象所需要的时间将增长并最终会达到引起用户注意的程度。

当这种情况发生时，应用程序开始定期地出现间歇式迟钝；问题变得更糟糕时，整个浏览器可能出现定期的冻结。这些情况都导致了糟糕的用户体验。

大多数现代化的平台都提供了成熟的工具使你能够监控运行环境中 GC 进程的性能，并且可以观察堆中当前的对象集以便能诊断 GC 相关的问题。不幸的是，JavaScript 运行环境不属

译注 7：类似 Java 虚拟机中最大的一个性能问题是应用程序线程与同时运行的 GC 的互斥性。垃圾收集器要完成其工作，需要在一段时间内防止所有其他线程访问它正在处理的堆空间（内存）。按 GC 的术语，这段时间称为“stop-the-world”，并且，正如其名字所表明的，在垃圾收集器努力工作时，应用程序有一个急刹车。幸运的是，这种暂停通常是很短的，很难察觉到，但是很容易想象，如果应用程序在随机的时刻出现随机且较长时间的暂停，对应用程序的响应性和吞吐能力会有破坏性的影响。更多细节请见 *http://www.ibm.com/developerworks/cn/java/j-perf05214/*。

于这一类。更糟糕的是，没有工具能通知开发者何时进行垃圾回收或它们完成其工作消耗了多少时间；这类工具非常有利于帮助验证与 GC 相关的明显延迟。

这类工具的缺乏是开发大规模由浏览器托管的 JavaScript 应用的严重弊端。其间，开发者必须靠猜测来判断 GC 是否为形成 UI 延迟的原因。

2.4.5 虚拟内存
Virtual Memory

内存分页是与内存相关的另一种风险。操作系统为应用提供两种可用的内存：物理的和虚拟的。物理内存映射在基础计算机中极快的 RAM 芯片上；虚拟内存映射到非常慢的海量存储设备上（比如硬盘），它用更大的可用存储空间弥补了内存的相对狭小。

如果网页的内存需求增长到足够大，可能会迫使操作系统开始**内存分页**，一个极慢的进程凭借迫使其他进程放弃其真正的内存来给浏览器不断增长的需求腾出空间。之所以使用术语——**分页**（paging），是因为所有的现代操作系统把内存组织到独立的**页面**上，这个术语描述了映射到物理或虚拟内存上的最小内存单元。当分页发生时，系统把内存页从物理内存转移到虚拟内存（例如从 RAM 到硬盘）上，反之亦然。

分页导致的性能降低和 GC 停顿有一点不同；分页会导致全面的、无处不在的迟钝，而 GC 停顿往往会导致离散且孤立的停顿，它们会间歇式的发生并且停顿的长度会随时间而增长。尽管它们不同，但这些问题中的任意一个都是实现创建快速响应应用户界面目标的巨大挑战。

2.4.6 内存问题的疑难解答
Troubleshooting Memory Issues

正如前面提到，我们知道对于浏览器托管的 JavaScript 应用并没有好的内存排错工具。当前的技术停留在观察浏览器进程中内存痕迹的水平（关于如何在 Windows 和 OS X 中测量进程内存的细节请看 *http://blog.pavlov.net/2008/03/11/firefox-3-memory-usage/*中“测量内存的使用”章节），如果内存增长得比应用程序在使用过程中可忍受的大，那就需要检查代码，看看是否存在优化内存使用的可能性。

一旦确定内存有问题，应该在尚未开始清理内存的地方寻找解决问题的机会。可以通过下面的两种方式做：

- 使用 delete 关键字从内存中移除不再需要的 JavaScript 对象。
- 从网页的 DOM 树上移除不再是必需的节点。

下面的代码清单演示了如何执行这两项任务：

```
var page = { address: "http://some/url" };

page.contents = getContents(page.address);

...

// 以后，这些内容不再是必需的了
delete page.contents;

...

var nodeToDelete = document.getElementById("redundant");

// 从 DOM 中移除节点（它仅能通过从父节点调用 removeChild() 来完成）
// 并同时从内存中删除这个节点
delete nodeToDelete.parent.removeChild(nodeToDelete);
```

显然，在优化网页内存使用的领域提升空间很大。在 Mozilla，我们正在开发解决这个问题的工具。实际上，当你读到这里时，你应该可以通过访问 *http://labs.mozilla.com* 找到一个或多个这样的工具（译注 8）。

2.5 总结

Summary

Ajax 开创了一个持久运行且以 JavaScript 为中心的网页新时代。这样的网页是真正的托管应用，并和其他所有应用一样都遵循相同的用户界面准则。这类应用通过减少主应用线程上执行的运算来保持用户界面的快速响应，这点是至关重要的。

Web Workers 是一个功能强大的新工具，它可用于解除威胁到 UI 快速响应能力的复杂计算。当 Web Workers 不可用时，可以使用 Gears 插件和 JavaScript 定时器。

内存管理不善会导致 UI 的性能问题。因为缺乏优秀的排除内存问题的工具，开发者通常要观察浏览器的内存使用，然后逐步减少引起问题的应用的内存占用。有个好消息是内存故障排查工具正在开发中。

译注 8：Mozilla Labs Blog 的最近一篇文章介绍了各种浏览器内存故障排查工具的开发进展：*http://labs.mozilla.com/blog/2009/07/browser-memory-tools-directory/*。

第 3 章

拆分初始化负载

Splitting the Initial Payload

Ajax 和 DHTML（动态 HTML）的日益普及使如今网页上的 JavaScript 和 CSS 比以往任何时候都多。Web 应用程序变得越来越像桌面应用程序，很大一部分的应用代码不会在启动时被使用。高级的桌面应用程序采用的是插件式架构，允许动态加载模块，许多 Web 2.0 的应用程序也都采用了这种方式，并从中受益。本章将展示一些在启动时过度加载代码的流行的 Web2.0 应用程序，并讨论更加动态地加载页面的方法。

3.1 全部加载（译注 1）

Kitchen Sink

Facebook（*http://www.facebook.com/*）有 14 个共 786KB 未压缩的外部脚本（注 1）。即便是 Facebook 的核心前端工程师，想要弄清楚在初始化页面时有多少 JavaScript 是必须加载的也是非常困难的。在这 14 个外部脚本中有一些对页面初始化至关重要，而其他的也包括在内是因为它们对 Ajax 和 DHTML 功能的支持，例如下拉菜单和评论，以及如图 3-1 中所示的一些类似功能。

尽可能快速渲染出网页很关键，这样做不仅可吸引用户，而且可以为他们创建良好的响应体验。试想一下，如果把 Facebook 的 JavaScript 分为两个部分：一部分是渲染初始页面必需的，剩下的作为另一部分。与其给用户带来响应停顿的第一印象，倒不如在初始化时只加载必要的 JavaScript，其余的 JavaScript 稍后再加载。

译注 1：此处的原文是 Kitchen sink，是俗写 Everything but the kitdner sink 的简写，意思是几乎所有的一切。根据上下文，此处是指页面初始化时全部加载所有的脚本。

注 1：当用户登录访问此页面时有 14 个脚本被加载。如果用户没有登录，则只有少数的脚本被使用。

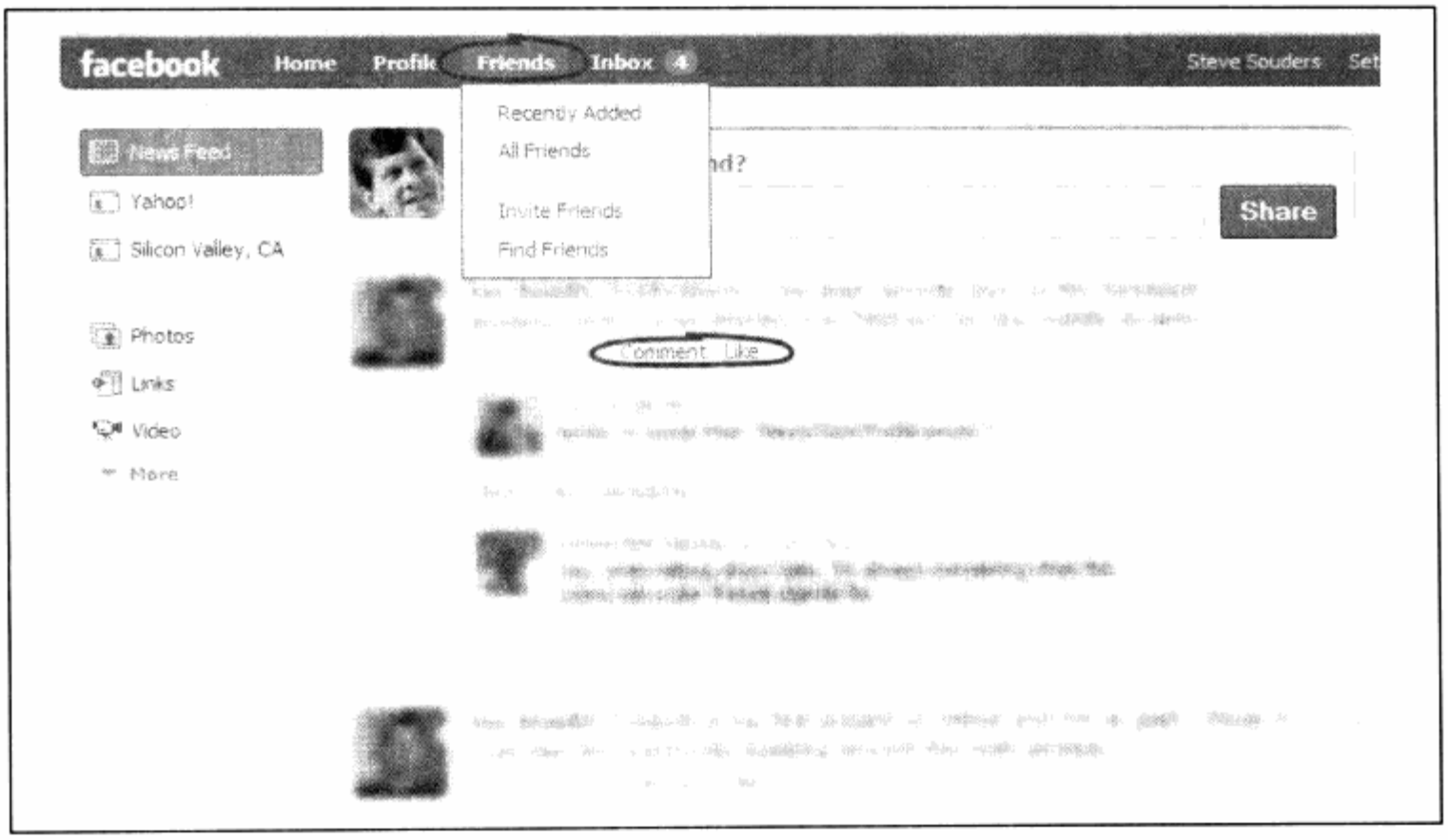


图 3-1：Facebook 中 Ajax 和 DHTML 的特性

这其中有一些问题：

- 能节省多少时间？
- 如何找到需要拆分的代码？
- 怎样处理竞争状态？
- 如何延迟加载“其余部分”的代码？

前 3 个问题在这一章中解决。如何延迟加载“其余部份”的代码是第 4 章的主题。

3.2 通过拆分来节省下载量
Savings from Splitting

事实上，一直到触发 onload 事件时，Facebook 执行的 JavaScript 函数只占全部下载量的 9%。使用 Firebug 的 JavaScript 性能分析器就能统计出直到触发 onload 事件时有多少函数被执行。（注 2）以 onload 事件为统计截止点是因为在此之后所需的功能可以也应该在初始页面渲染完成时开始加载。我称之为在 onload 后加载（post-onload download）。（各种不同的延迟加载技术详见第 4 章。）

表 3-1 列举了 10 个美国顶级网站在 onload 事件前下载但并未执行的函数的百分比。平均而

注 2：Firebug 是一个优秀的网站开发工具，可见 *http://getfirebug.com/*。

言，在初始化页面期间 75%的函数下载了但并未执行，因此，如果延迟下载这些未执行的函数，初始化时 JavaScript 的下载量将大大减少。

诚然，75%的估计值可能有些被夸大，那些未执行的函数可能用于错误处理或某种特殊情况，但足以说明大部分的 JavaScript 是可以延迟下载的。未压缩的 JavaScript 大小平均为 252KB。这个百分比是指函数数量，而不是大小。如果我们假设函数大小一致，那么 75%（平均为 189KB）在 onload 事件之后才被加载，这将使页面的初始化速度大大提高。

表 3-1：在 onload 事件触发之前 JavaScript 函数执行情况百分比

网站	未执行函数的百分比	未压缩的 javscript 代码大小
http://www.ao.com	71%	115KB
http://www.ebay.com	56%	183KB
http://www.facebook.com	91%	786KB
http://www.google.com/search?q=flowers	56%	15KB
http://search.live.com/results.aspx?q=flowers	75%	17KB
http://www.msn.com	69%	131KB
http://www.myspace.com	87%	297KB
http://en.wikipedia.org/wiki/Flowers	79%	114KB
http://www.yahoo.com	88%	321KB
http://www.youtube.com	84%	240KB

3.3 寻找拆分

Finding the Split

Firebug 的 JavaScript 性能分析器能显示出在触发 onload 事件之前所有已执行的函数名。这个列表可以帮助我们把要下载的 JavaScript 代码拆分成两个文件，一个用于页面初始化，另一个则可以延后加载。 然而，一些函数虽然未被使用，但仍然是必需的，比如错误处理和一些条件判断代码。所以完全准确地将需要初始化的代码剥离出来，并避免未定义标识符错误是一个巨大的挑战。JavaScript 的一些高级特性，包括函数的作用域和 eval 使这个挑战变得更加复杂。

Doloto（*http://research.microsoft.com/apps/pubs/default.aspx?id=70518*）是由微软研究院开发的自动拆分 JavaScript 代码的系统，它可以把代码拆分到不同的组。第一组包含初始化网页所必需的函数，剩下的则在这些代码需要执行时按需加载它们，或者等到初始化的那些 JavaScript 代码加载完毕时再加载。当应用到 Gmail、Live Maps、Redfin、MySpace 和 Netflix

时，Doloto 减少了多达 50%的初始 JavaScript 代码下载量，应用程序的加载时间缩短了 20%~40%。

Doloto 决定在什么位置拆分代码是基于训练阶段（training phase）的，它会把 JavaScript 拆分成多个文件下载。对于大多数 Web 应用程序来说，最好把在 onload 事件之前执行的 JavaScript 代码拆分成一个单独的文件，下载完成之后剩下的 JavaScript 采用无阻塞下载技术立即下载，我们将在第 4 章讲述这种技术。直到用户拖动菜单或点击页面上的某个元素之后才开始下载额外文件的话，会导致用户必须等待 JavaScript 加载完毕。假如所有额外的 JavaScript 在页面初始化渲染之后下载，我们就可以避免这种等待。在 Doloto 或其他系统完全公开之前，开发人员还是需要手动拆分代码。下面一节将讨论拆分时需注意的一些事项。

3.4 未定义标识符和竞争状态

Undefined Symbols and Race Conditions

拆分 JavaScript 代码的一个难点是要避免出现未定义标识符错误。如果在 JavaScript 执行时引用到一个被降级到延迟加载的标志符时，就会出现这种问题。比如在 Facebook 中，我建议对下拉菜单的 JavaScript 延后加载。但是，如果下拉菜单在它所需要的 JavaScript 下载完成之前显示出来，而且用户可以点击，那么 JavaScript 就不会生效。我的建议是建立一个下载 JavaScript 和用户点击菜单之间的竞争状态。在大多数情况下，JavaScript 会赢得这个竞争状态，但也确实会发生用户先点击的情况，此时调用下拉菜单函数（尚未被下载），就会出现未定义标识符的错误。

在延迟加载的代码与用户界面元素相关联的情况下，可以通过改变元素的展现来解决此问题。在这种情况下菜单可以包含一个“加载中…”的图标，提醒用户该功能还没加载完成。

另一个选择是在延迟加载的代码里绑定界面元素的事件处理程序。在这个例子中，菜单会被初始化成一个静态文本，点击它不会执行任何 JavaScript。延迟加载的代码包括菜单的功能和事件的绑定，在 Internet Explorer 中通过 attachEvent 实现，其他浏览器则需要采用 addEventListener（注 3）。

在延迟加载的代码不与界面元素相关联的情况下，可以使用桩(stub)函数解决这个问题。桩函数是一个与原函数名称相同但是函数体为空，或者是用一些临时代码代替原有内容的函数。前一节介绍过的 Doloto 具有按需加载额外 JavaScript 模块的能力。Doloto 实现这个按

注 3：更多信息请看 *http://www.quirksmode.org/js/events_advanced.html*。

需下载的功能就是在初始化下载的代码中插入桩函数，当调用它们时，动态加载其他的JavaScript 代码。当新增的 JavaScript 代码下载完成，原函数会覆盖桩函数。

更简单的方法是给每一个被引用但又被降级为延迟下载的函数创建一个桩函数。如有需要，桩函数可以返回一个桩值，例如一个空字符串。如果用户在完整的函数下载完成之前尝试使用这个 DHTML 的功能，将没有任何反应。稍微高级一点的解决方案是，用桩函数记录用户的请求，并在 JavaScript 完成加载时调用相应的代码。

3.5 个案研究：Google 日历

Case Study: Google Calendar

Google 日历是一个很好的拆分出初始加载的例子。图 3-2 显示的是当 Google 日历被访问时的 HTTP 请求状况，我把这些图表叫做 **HTTP 瀑布图**。每个横杠代表一个请求，资源类型显示在左边，横轴代表时间，横杠的位置说明了在加载页面时每个资源开始和结束的点。

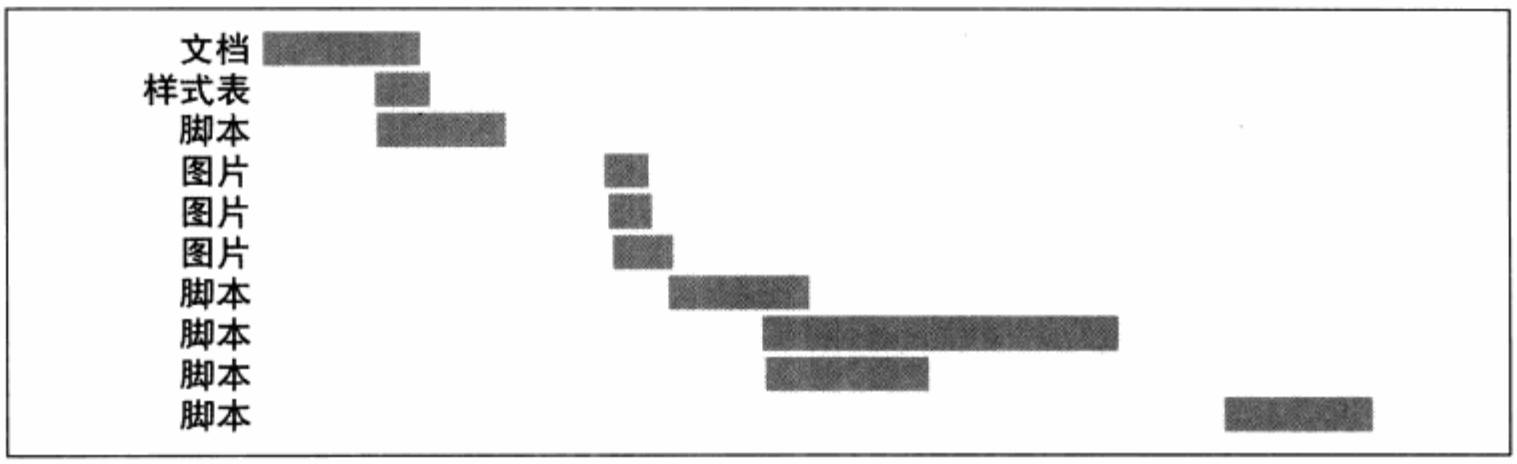


图 3-2：Google 日历 HTTP 瀑布图

Google 日历请求了 5 个脚本，压缩前共 330KB。整个负载中拆分出了一个 152KB 的初始化脚本，被优先请求（从顶部数第 3 条横杠）。由于这个脚本仅不到 JavaScript 总量的一半大小，因此阻塞现象大大减轻。其余的 JavaScript 负载在页面开始渲染后加载。

Google 日历团队通过拆分 JavaScript，使页面渲染速度比所有 JavaScript 在同一个文件中加载的做法快很多。拆分 Web 应用程序的 JavaScript 并不是一个简单的任务。它需要明确初始化时所需的函数，找到所有必需代码的依赖关系，设置桩函数，并延迟加载其余的 JavaScript。这些任务需要进一步自动化。微软的 Doloto 项目就是这样的一个系统，但在撰写本文时，它还没有发布。在这个工具正式推出之前，开发者只能先自力更生了。

本章的重点是拆分 JavaScript，但拆分 CSS 样式表也是有益的。相对于拆分 JavaScript，后者节省的资源要少一些，因为样式表的整体大小通常比 JavaScript 小，而且下载 CSS 并不会像 JavaScript 那样具有阻塞特性（注 4）。不过这也是一个值得深入研究，并为其开发工具的机会。

注 4：Firefox 2 是一个例外。

第 4 章

无阻塞加载脚本

Loading Scripts Without Blocking

script 标签的阻塞行为会对页面性能产生负面影响。大多数浏览器在下载或执行脚本的同时不会下载其他内容。有时候这种阻塞是必要的，所以，能够识别出 JavaScript 不依赖于页面中其他内容而单独加载的情况，这一点非常重要。

当遇到这些情况时，我们希望以不阻塞其他内容下载的方式来加载 JavaScript。幸运的是有些技术可以做到这点，使页面加载更快。本章将说明这些技术，比较它们如何对浏览器和性能施加影响，并说明在不同情况下应该优先选择哪种技术。

4.1 脚本阻塞并行下载

Scripts Block

JavaScript 以行内脚本或外部脚本的形式包含在网页中。在 HTML 文档中，行内脚本通过 `script` 标签引入整段 JavaScript：

```
<script>
function displayMessage(msg) {
    alert(msg);
}
</script>
```

外部脚本通过 `script` 的 `src` 属性把独立文件中的 JavaScript 引入：

```
<script src='A.js'></script>
```

`src` 属性定义了须加载的外部文件的 URL。如果缓存中有脚本文件，浏览器就从缓存中读取，否则就发送 HTTP 请求获取。

通常，大多数浏览器是并行下载组件的，但对于外部脚本并非如此。当浏览器开始下载外部脚本时，在脚本下载、解析并执行完毕之前，不会开始下载任何其他内容。（任何已经在进程中的下载都不会被阻塞。）

图 4-1 为 Scripts Block Downloads 示例的 HTTP 请求瀑布图（注 1）。

Scripts Block Downloads

http://stevesouders.com/cuzillion/?ex=10008&title=Scripts+Block+Downloads

该页面顶部有两个脚本，A.js 和 B.js，接下来是一张图片、一个样式表和一个 iframe。每个脚本的下载时间和执行时间都设置成 1 秒。HTTP 瀑布图中的白色间隔表示脚本执行的时间段。该图显示，当脚本下载和执行时，浏览器阻塞了所有其他下载。只有当脚本执行完成之后，图片、样式表和 iframe 才开始并行下载。

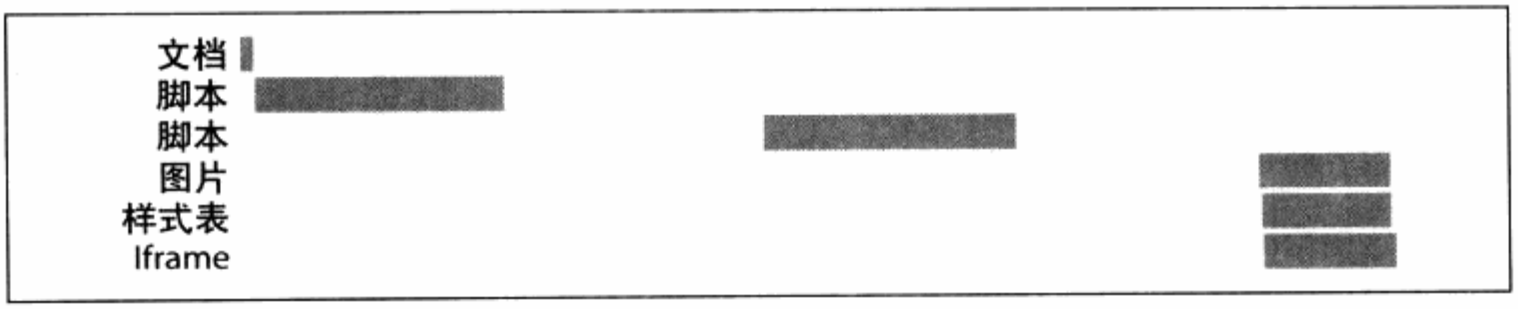


图 4-1：脚本阻塞并行下载

浏览器在下载和执行脚本时出现阻塞的原因在于，脚本可能会改变页面或 JavaScript 的名字空间，它们会对后续内容造成影响。典型的例子是 A.js 使用 `document.write` 改变了页面，另一个例子是 B.js 依赖于 A.js。浏览器要确保脚本按它们在 HTML 文档中出现的顺序来执行，这样才能保证 A.js 在 B.js 之前下载并执行。如果没有这个保证，那么 B.js 可能在 A.js 之前完成下载和执行，这种竞争状态会导致 JavaScript 错误。

很显然脚本必须按顺序**执行**，但没有必要按顺序**下载**，在这方面 Internet Explorer 8 走到了前列。图 4-1 中展示了在大多数浏览器中出现的情况，包括 Firefox3.0 和 Internet Explorer 7，以及它们的老版本。但是图 4-2 中展示的 Internet Explorer 8 的下载瀑布图有所不同，Internet Explorer 8 是第一个支持脚本并行下载的浏览器。

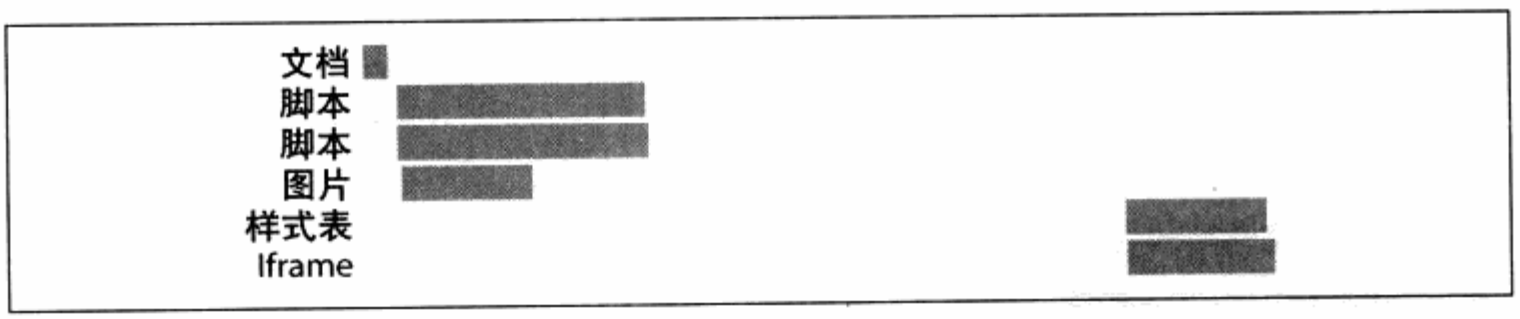


图 4-2：Internet Explorer 8 无阻塞下载脚本

注 1：本例和其他例子都是由 Cuzillion 生成的，Cuzillion 是我特意为本章创建的工具。要了解更多 Cuzillion 的信息请参阅附录。

Internet Explorer 8 并行下载脚本的能力让页面加载更快，但如图 4-2 所示，它并没有完全解决阻塞问题。虽然它的确实现了并行下载 A.js 和 B.js，但仍在脚本下载并执行完毕之前阻塞图片和 iframe 的下载。Safari4、Chrome2 与 Internet Explorer 8 类似——它们并行下载脚本，但阻塞后面的资源（注 2）。

我们真正想要的是让脚本与所有其他组件并行下载，而且希望在所有浏览器中实现。下一节中探讨的技术将说明如何实现它。

4.2 让脚本运行得更好

Making Scripts Play Nice

有几种下载外部脚本的技术可以使页面不会被脚本的阻塞行为所影响。一种技术是把所有 JavaScript 内嵌到页面中，但我**并不**推荐用它。在少数情况下（首页、少量 JavaScript）内嵌 JavaScript 尚可接受，但通常由于页面大小和缓存能带来更多好处，因此用外部文件引入 JavaScript 更好一些。（想要了解更多关于这些权衡的信息，详见《高性能网站建设指南》里的“规则 8：使用外部 JavaScript 和 CSS”。）

这里列出来的技术既拥有外部脚本的好处，又能避免因阻塞导致的减速影响：

- XHR Eval。
- XHR 注入（XHR Injection）。
- Script in Iframe。
- Script DOM Element。
- Script Defer。
- document.write Script Tag。

以下各小节将详细介绍每种技术，包括比较它们如何影响浏览器，以及如何在不同情况下做技术选择。

4.2.1 XHR Eval

该技术通过 `XMLHttpRequest`（XHR）从服务端获取脚本。如示例页面所示，当响应完成时通过 `eval` 命令执行内容。

注 2：截至写稿时，Firefox 还不支持并行脚本下载，但预计快了。

XHR Eval

http://stevesouders.com/cuzillion/?ex=10009&title=Load+Scripts+using+XHR+Eval

正如你所看到的，在图 4-3 的 HTTP 瀑布图中，`XMLHttpRequest` 没有阻塞页面中的其他组件——5 个资源都在并行下载。脚本在它们下载完成后才开始执行。(执行时间并没有在 HTTP 瀑布图中显示出来，因为它并不包含在网络活动中。)

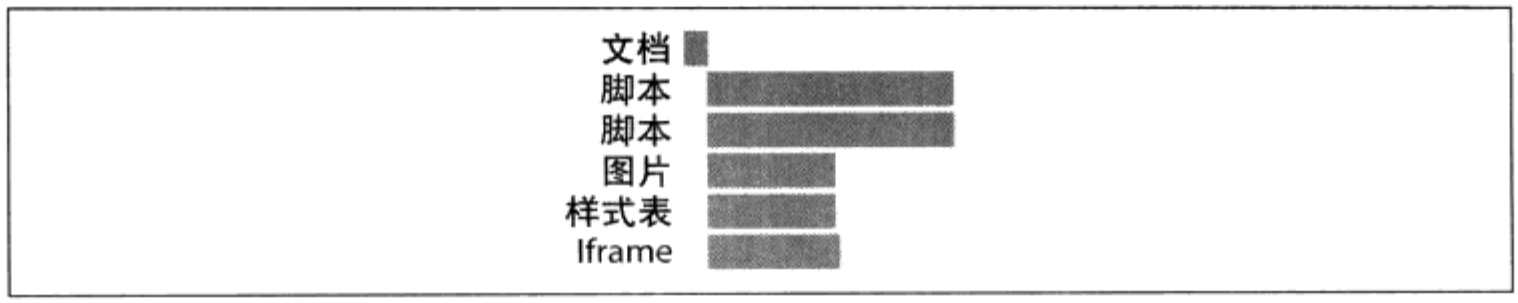


图 4-3：用 XHR Eval 加载脚本

这个方法的主要缺陷是，通过 `XMLHttpRequest` 获取的脚本必须部署在和主页面相同的域中。以下是 XHR Eval 示例相关的源码（注 3）：

```
var xhrObj = getXHRObject();
xhrObj.onreadystatechange =
    function() {
        if ( xhrObj.readyState == 4 && 200 == xhrObj.status ) {
            eval(xhrObj.responseText);
        }
    };
xhrObj.open('GET', 'A.js', true); //必须和主页面在同一个域中
xhrObj.send('');

function getXHRObject() {
    var xhrObj = false;
    try {
        xhrObj = new XMLHttpRequest();
    }
    catch(e){
        var progid = ['MSXML2.XMLHTTP.5.0', 'MSXML2.XMLHTTP.4.0',
'MSXML2.XMLHTTP.3.0', 'MSXML2.XMLHTTP', 'Microsoft.XMLHTTP'];
        for ( var i=0; i < progid.length; ++i ) {
            try {
                xhrObj = new ActiveXObject(progid[i]);
            }
            catch(e) {
                continue;
            }
```

注 3：如果使用 JavaScript 库，通常有对 `XMLHttpRequest` 的封装，例如 `jQuery.ajax` 和 `dojo.xhrGet`。使用它们就无须自己来写封装了。

```
            break;
        }
    }
    finally {
        return xhrObj;
    }
}
```

4.2.2 XHR 注入

XHR Injection

类似于XHR Eval,XHR 注入技术也是通过 XMLHttpRequest 来获取 JavaScript 的。但与 eval 不同的是，该机制是通过创建一个 script 的 DOM 元素，然后把 XMLHttpRequest 的响应注入 script 中来执行 JavaScript 的。在某些情况下使用 eval 可能比这种机制慢。

XHR Injection

http://stevesouders.com/cuzillion/?ex=10015&title=XHR+Injection

通过 XMLHttpRequest 获取的内容必须部署在和主页面相同的域中。以下是 XHR Injection 示例的相关源码：

```
var xhrObj = getXHRObject(); //它已在前面的例子中定义了
xhrObj.onreadystatechange =
    function() {
        if ( xhrObj.readyState == 4 ) {
            var scriptElem = document.createElement('script');
            document.getElementsByTagName('head')[0].appendChild(scriptElem);
            scriptElem.text = xhrObj.responseText;
        }
    };
xhrObj.open('GET', 'A.js', true); // 必须和主页面在同一个域中
xhrObj.send('');
```

4.2.3 Script in Iframe

主页中的 iframe 和其他组件是并行加载的。与用 iframe 在一个 HTML 页面中包含另一个页面的传统做法不同,Script in Iframe 技术利用 iframe 无阻塞加载 JavaScript,如 Script in Iframe 示例所示。

Script in Iframe

http://stevesouders.com/cuzillion/?ex=10012&title=Script+in+Iframe

实现过程完全在 HTML 中完成：

```
<iframe src='A.html' width=0 height=0 frameborder=0 id=frame1></iframe>
```

注意该技术使用了 A.html，而不是 A.js，因为 iframe 认为其返回的是 HTML 文档。所有我们要做的就是在 HTML 文档中把外部脚本转换成行内脚本。

与 XHR Eval 和 XHR 注入这两种技术类似，该技术要求 iframe URL 和主页面同域。（浏览器跨域安全机制不允许 iframe 中的 JavaScript 访问跨域的父页面，反之亦然。）即使主页面和 iframe 同域，我们仍然需要修改 JavaScript 来创建它们之间的关联。一个办法是通过 `frames` 数组或 `document.getElementById` 来获得引用 iframe 的 JavaScript 标识符：

```
// 使用“frames”访问主页面上的 iframe
window.frames[0].createNewDiv();

// 使用“getElementById”访问主页面上的 iframe
document.getElementById('frame1').contentWindow.createNewDiv();
```

iframe 使用 parent 变量引用父页面：

```
// 在 iframe 中使用“parent”访问主页面
function createNewDiv() {
    var newDiv = parent.document.createElement('div');
    parent.document.body.appendChild(newDiv);
}
```

Iframe 还存在自身的消耗。实际上 iframe 是开销最高的 DOM 元素，至少比普通 DOM 元素高出一个数量级，这一点将在第 13 章讲解。

4.2.4 Script DOM Element

相对于在 HTML 中使用 script 标签来下载脚本文件而言，该技术使用 JavaScript 动态地创建 script DOM 元素并设置其 src 属性，这只要两三行 JavaScript 代码就可以实现。

```
var scriptElem = document.createElement('script');
scriptElem.src = 'http://anydomain.com/A.js';
document.getElementsByTagName('head')[0].appendChild(scriptElem);
```

下载过程中用这种方式创建脚本不会阻塞其他组件。和之前的技术相反，Script DOM Element 技术允许跨域获取脚本。实现该技术的代码既短小又简单。外部脚本可以直接调用，而不用像 XHR Eval 或 Script in Iframe 技术那样需要重构。

Script DOM Element
http://stevesouders.com/cuzillion/?ex=10010&title=Script+Dom+Element

4.2.5 Script Defer

Internet Explorer 支持 `script` 的 `defer` 属性，这为开发人员另辟蹊径，可以让浏览器不必立即加载脚本。当脚本中不包含对 `document.write` 的调用，且当前页面中没有其他脚本

依赖于它时，使用这个属性是安全的。当 Internet Explorer 下载设置 defer 属性的脚本时，允许其他资源并行下载。

Script Defer
http://stevesouders.com/cuzillion/?ex=10013&title=Script+Defer

`defer` 属性是非常简单的防止脚本阻塞行为的方法，添加一个属性即可：

```
<script defer src='A.js'></script>
```

虽然 `defer` 是 HTML4 规范的一部分（*http://www.w3.org/TR/REChtml40/interact/scripts.html#adef-defer*），但只有 Internet Explorer 和一些新浏览器支持它。

4.2.6 document.write Script Tag

最后一项技术是使用 `document.write` 把 HTML 标签 `script` 写入页面中。

`document.write` Script Tag
http://stevesouders.com/cuzillion/?ex=10014&title=document.write+Script+Tag

像 Script Defer 一样，该技术只在 Internet Explorer 中是并行加载脚本的。虽然多个脚本可以并行下载（假设所有 `document.wrtie` 语句都在同一个 script 语句块中），但在下载脚本时，浏览器仍然阻塞其他类型的资源。

```
document.write("<script type='text/javascript' src='A.js'><\/script>");
```

4.3 浏览器忙指示器
Browser Busy Indicators

上一节讲述的所有技术都用于改进 JavaScript 下载，使得可以并行下载多种资源。但是这些技术在另外一些方面有所不同，其中一项就是它们会影响用户感知页面是否加载完毕。浏览器提供了多种**忙指示器**，让用户感知到页面还在加载。

图 4-4 显示了 4 种浏览器忙指示器：状态栏、进度条、标签页图标和光标。状态栏显示当前正在下载的 URL，进度条在窗口底部走完表明下载完成，图标在下载时会旋转，光标会变成沙漏或其他类似的图形来表示页面正忙。

另外两个浏览器忙指示器是阻塞渲染和阻塞 `onload` 事件。阻塞渲染对于用户体验来说是非常不好的。当使用 `SCRIPT SRC` 技术下载脚本时，浏览器停止渲染所有脚本后面的内容。在全部渲染完毕之前通过冻结页面来表示浏览器正忙是一种非常差劲的方式。

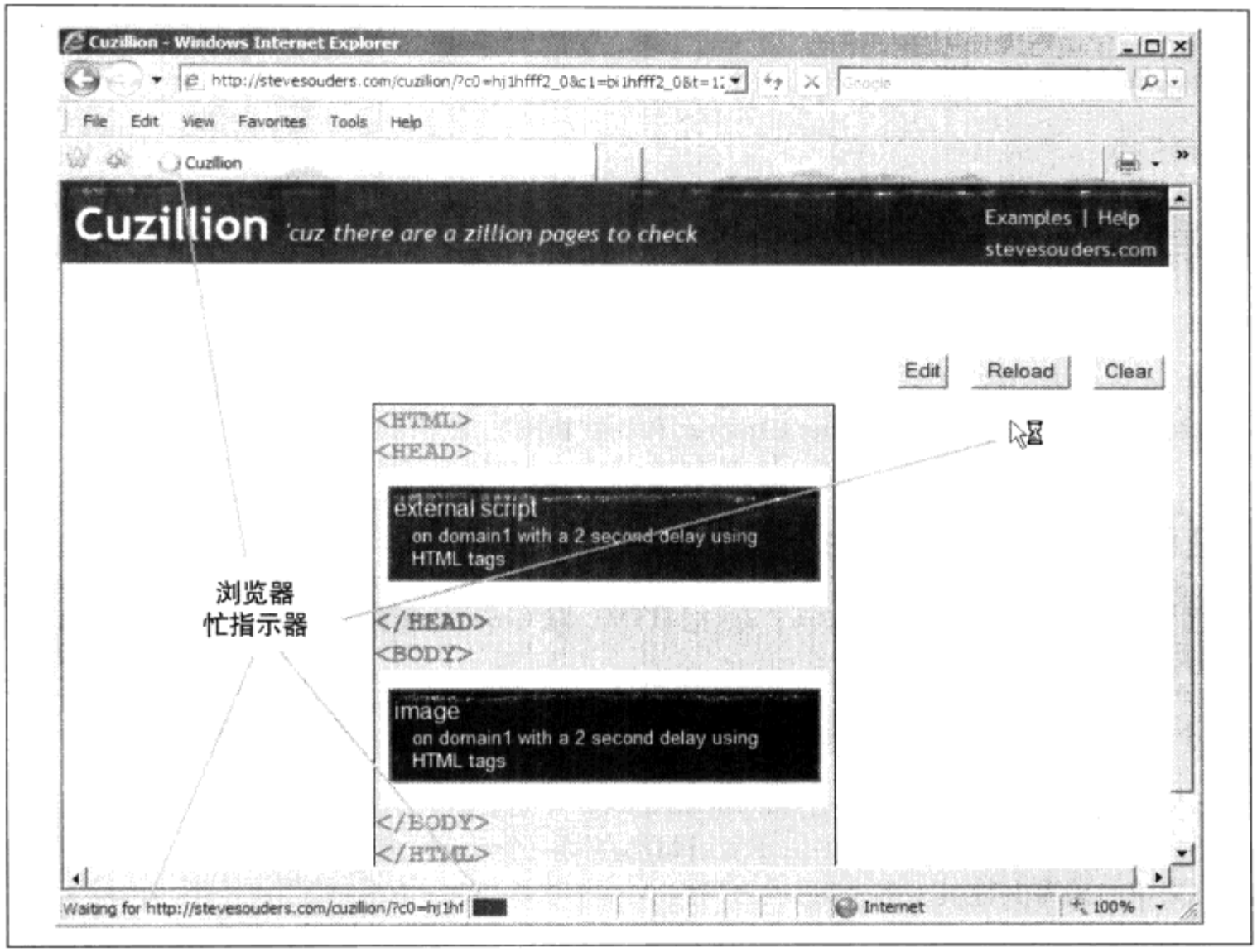


图 4-4：浏览器的忙指示器

通常页面的 onload 事件要等到所有资源下载完成时才会触发。如果让状态栏等待更长时间才显示“完成”，并且延迟默认输入框获取焦点，会影响用户体验。

在大多数浏览器中通过 SCRIPT SRC 技术下载 JavaScript 都会触发浏览器忙指示器，而使用 XHR Eval、XHR 注入技术，无论是在 Internet Explorer、Firefox 还是在 Opera 中，都不会触发忙指示器。忙指示器是否触发是由所使用的技术和浏览器共同决定的。

表 4-1 显示了每种 JavaScript 下载技术分别触发哪些忙指示器。XHR Eval 和 XHR 注入技术触发的忙指示器最少，其他技术各有千秋。虽然忙指示器触发在各浏览器之间有所不同，但在同一浏览器的不同版本之间通常是一致的。

表 4-1：JavaScript 下载触发的浏览器忙指示器

技术	状态栏	进度条	图标	光标	阻塞渲染	阻塞 onload
常规 Script Src	FF、Saf、Chr	IE、FF、Saf	IE、FF、Saf、Chr	FF、Chr	IE、FF、Saf、Chr、Op	IE、FF、Saf、Chr、Op
XHR Eval	Saf、Chr	Saf	Saf、Chr	Saf、Chr	--	--
XHR 注入	Saf、Chr	Saf	Saf、Chr	Saf、Chr	--	--
Script in Iframe	IE、FF、Saf、Chr	FF、Saf	IE、FF、Saf、Chr	FF、Chr	--	IE、FF、Saf、Chr、Op
Script DOM Element	FF、Saf、Chr	FF、Saf	FF、Saf、Chr	FF、Chr	--	FF、Saf、Chr
Script Defer[a]	FF、Saf、Chr	FF、Saf	FF、Saf、Chr	FF、Chr、Op	FF、Saf、Chr、Op	IE、FF、Saf、Chr、Op
document.write Script Tag[b]	FF、Saf、Chr	IE、FF、Saf	IE、FF、Saf、Chr	FF、Chr、Op	IE、FF、Saf、Chr、Op	IE、FF、Saf、Chr、Op

a 在 Fireffox3.1 及更新的版本中，Script Defer 技术实现了并行下载。

b 请注意 document.write Script Tag 只在 IE、Safari4、Chrome2 中实现了并行下载。

提示：缩写如下：(Chr) Chrome 1.0.154 和 2.0.156；(FF) Firefox 2.0、3.0 和 3.1；(IE) Internet Explorer 6、7 和 8；(Op) Opera 9.63 和 10.00 alpha；(Saf) Safari 3.2.1 和 4.0 (developer preview)。

理解每种技术如何对浏览器忙指示器产生影响相当重要。在某些情况下为了得到更好的用户体验，我们需要忙指示器：它让用户知道页面正在运行；其他情况下，最好是不显示任何忙指示器，从而鼓励用户开始与页面进行交互。

4.4 确保（或避免）按顺序执行

Ensuring (or Avoiding) Ordered Execution

很多时候网页都包含多个有特定依赖顺序的脚本。使用常见的 SCRIPT SRC 技术保证脚本按它们在页面中排列的顺序下载和执行。然而使用前面描述的某些高级的下载技术并不能确保这点。因为脚本是并行下载的，所以它们会按到达的顺序执行——最先到达的最先执行——而不是按它们排列的顺序。这会导致竞争状态，进而导致未定义标识符错误。

其中有一些技术确实可以保证 script 按顺序执行，但不能跨浏览器。对于 Internet Explorer 来说，Script Defer 和 document.write Script Tag 保证脚本按顺序执行而不管哪个先下载完

成。IE Ensure Ordered Execution 示例中包含了 3 个使用 Script Defer 技术加载的脚本。虽然第 1 个脚本（在 URL 中有 `sleep=3` 的）最后下载完成，但它仍然最先执行。

IE Ensure Ordered Execution

http://stevesouders.com/cuzillion/?ex=10017&title=IE+Ensure+Ordered+Execution

因为 Script Defer 和 `document.write` Script Tag 技术没有在 Firefox 中实现并行下载，所以当脚本之间有依赖关系时必须使用另一种技术。在 Firefox 中，Script DOM Element 技术能保证脚本按列出的顺序执行。FF Ensure Ordered Execution 示例中包含了使用 Script DOM Element 加载的 3 个脚本。虽然第 1 个脚本（在 URL 中有 `sleep=3` 的）最后下载完成，但它仍然最先执行。

FF Ensure Ordered Execution

http://stevesouders.com/cuzillion/?ex=10018&title=FF+Ensure+Ordered+Execution

脚本按特定顺序执行并非总那么重要。有时候你其实想让浏览器执行任何一个先下载完成的脚本，因为那样可以更快地渲染页面。比如一个网页中包含多个控件（A、B、C），对应的脚本（A.js、B.js、C.js），它们之间没有任何内部依赖关系。即使页面可能按这样的顺序排列控件脚本，但立即执行先接收到的控件脚本会带来更好的用户体验。XHR Eval 和 XHR 注入技术可以做到这一点。Avoid Ordered Execution 示例执行最先下载的脚本，虽然该脚本不是列在页面最前面的。

Avoid Ordered Execution

http://stevesouders.com/cuzillion/?ex=10019&title=Avoid+Ordered+Execution

4.5 汇总结果

Summarizing the Results

我已经介绍了一些下载外部脚本的高级技术和它们之间的各种权衡。表 4-2 汇总了这些结果。

表 4-2：高级脚本下载技术综述

技术	并行下载	跨域	现成代码（译注 1）	忙指示器	确保顺序	大小（字节）
Normal Script Src	(IE8、Saf4)[a]	是	是	IE、Saf4、(FF、Chr)[b]	IE、Saf4、(FF、Chr、Op)[c]	~50

译注 1：指无须为该技术重构 JavaScript。

技术	并行下载	跨域	现成代码（译注 2）	忙指示器	确保顺序	大小（字节）
XHR Eval	IE、FF、Saf、Chr、Op	否	否	Saf、Chr	--	~500
XHR 注入	IE、FF、Saf、Chr、Op	否	是	Saf、Chr	--	~500
Script in Iframe	IE、FF、Saf、Chr、Op[d]	否	否	IE、FF、Saf、Chr	--	~50
Script DOM Element	IE、FF、Saf、Chr、Op	是	是	FF、Saf、Chr	FF、Op	~200
Script Defer	IE、Saf4、Chr2、FF3.5	是	是	IE、FF、Saf、Chr、Op	IE、FF、Saf、Chr、Op	~50
document.write Script Tag	(IE、Saf4、Chr2、Op)[e]	是	是	IE、FF、Saf、Chr、Op	IE、FF、Saf、Chr、Op	~100

a 脚本并行下载，但其他类型资源的下载仍然被阻塞。

b 使用这项技术时，这些浏览器并不支持 JavaScript 与其他资源并行下载，但支持多个 JavaScript 之间并行下载。

c 同 a。

d 在 Opera 中一个有趣的性能提升是脚本不仅并行下载，而且会并行执行。

e 同 a。

提示：缩写如下：(Chr) Chrome 1.0.154 和 2.0.156；(FF) Firefox 2.0 和 3.1；(IE) Internet Explorer 6、7 和 8；(Op) Opera 9.63 和 10.00 alpha；(Saf) Safari 3.2.1 和 4.0 (developer preview)。

这些技术允许脚本与页面中其他资源并行下载，而在浏览器默认情况下，即使新型浏览器也无能为力。显然，这能显著提高网页速度，对于 Web2.0 应用程序来说尤其重要，因为它们的外部脚本的数量和大小都比其他网页多很多。

不推荐使用 `document.write` Script Tag 技术，因为它只在部分浏览器中实现并行下载，而且还阻塞脚本之外所有其他资源的下载。Script Defer 技术也只在部分浏览器中实现了并行下载。

当脚本与主页面同域时，XHR Eval、XHR 注入和 Script in Iframe 可以满足需求。但如果使用 XHR Eval 或 Script in Iframe 技术，我们需要重构一部分脚本，而 XHR 注入和 Script DOM Element 技术可以直接使用现有的脚本文件，无需任何改动。在表 4-2 的“大小”一栏中列出了实现每种技术需要添加到页面中的大概字符数。

译注 2：指无须为该技术重构 JavaScript。

我们还要考量各浏览器在忙指示器处理上的不同效果。如果下载页面初始化渲染并不需要用到的脚本（例如“延迟加载”），首选像 XHR Eval 和 XHR 注入这些让页面显示为完成的技术。如果你要明示用户页面还在加载，那么 Script in Iframe 技术会更合适一些，因为它会触发更多的忙指示器。

最后要考虑的是，选用哪种技术取决于脚本的执行顺序是否与下载顺序有关系。如果其他资源与脚本并行下载并且按顺序执行，那么需要根据不同的浏览器来综合使用多种技术。如果加载顺序无所谓，可以选用 XHR Eval 或 XHR 注入技术。

4.6 最佳方案

And the Winner Is

我的结论是：没有独立的最佳方案，真正的最佳方案取决于需求。图 4-5 显示了选择最佳脚本下载技术的决策树。

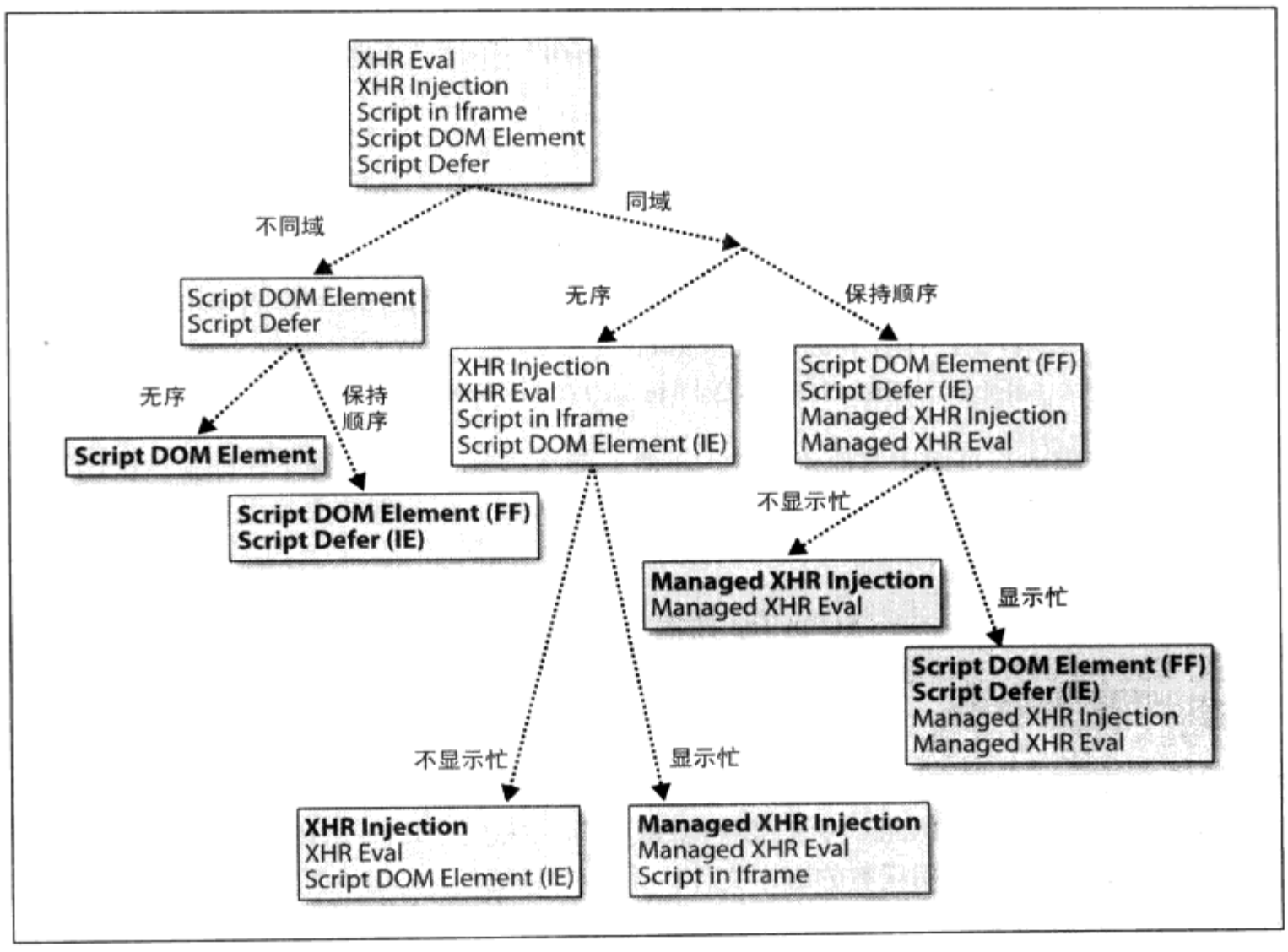


图 4-5：选择最佳脚本加载技术的决策树

在决策树里有 6 种可能的结果：

不同域、无序

XHR Eval、XHR 注入和 Script in Iframe 技术在这些情况下无法使用，因为主页面的域与脚本的域不同。我们不应该使用 Script Defer 技术，因其迫使脚本按顺序加载，相反，如果脚本能一到达就执行的话会让页面加载更快一些。对于这种情况，Script DOM Element 技术是最佳方案，但它会导致在 Firefox 中加载时保持顺序，虽然我们并不希望这样。注意这两种技术都触发忙指示器，但我们没有办法避免这个问题。包含 JavaScript 广告和控件的网页是符合这种情况的实例。广告和控件的脚本所在的域往往和主页面不同，但是它们之间没有任何依赖关系，所以加载顺序无关紧要。

不同域、保持顺序

和前面一样，因为主页面的域与脚本不同，XHR Eval、XHR 注入和 Script in Iframe 技术行不通。为了确保加载顺序，我们应该在 Internet Explorer 中使用 Script Defer 技术，而在 Firefox 中使用 Script DOM Element 技术。注意这两种技术都触发忙指示器。一个符合这种情况的实例是从不同的服务器下载多个存在依赖关系的 JavaScript 文件的页面。

同域、无序、无忙指示器

XHR Eval 和 XHR 注入是唯一不触发忙指示器的两种技术。这两种 XHR 技术，我更推荐 XHR 注入，因为使用该技术时脚本无需重构。如第 3 章所述，它可以应用于希望在后台下载 JavaScript 文件的网页。

同域、无序、有忙指示器

XHR Eval、XHR 注入和 Script in Iframe 是唯一跨 Internet Explorer 和 Firefox 而不保持加载顺序的技术。Script in Iframe 技术看起来似乎是最佳选择，因为它触发忙指示器，而且只是稍微增加了一些页面大小。但我更倾向于 XHR 注入，因为使用该技术时脚本无需重构就能使用，而且它同时是其他决策树分支的选择。我们需要额外的 JavaScript 来激活忙指示器：当 XHR 发出时激活状态栏和光标，XHR 返回时恢复。我称之为“管理 XHR 注入”（Managed XHR Injection）。

同域、保持顺序、无忙指示器

XHR Eval 和 XHR 注入是两个不触发忙指示器的技术。这两种 XHR 技术中我倾向于 XHR 注入，因为使用该技术时脚本无须重构。为了保持加载顺序，需要另一种类型的“管理 XHR 注入”。在这种情况下，如有必要，可以把 XHR 响应塞进队列里，按顺序执行，延迟加载的脚本则须所有在它之前的脚本下载并执行完毕后才开始执行。有多个存在内部依赖的脚本在后台下载的页面符合这种情况。

同域、保持顺序、有忙指示器

Internet Explorer 中的 Script Defer 技术和 Firefox 中的 Script DOM Element 技术是首选方案。管理 XHR 注入和管理 XHR Eval（Managed XHR Eval）技术也是可行的。但是它们会给主页面增加额外的代码，而且实现它们更复杂一些。

接下来会提供一个简单函数，用代码来实现这个逻辑，开发者可以通过调用该函数来确保按最佳方案加载脚本。此类函数的原型应该如下：

```
function loadScript(url, bPreserveOrder, bShowBusy);
```

为了避免下载太多不必要的 JavaScript，通过服务端程序的后端程序（例如 Perl、PHP、Python）来实现是最有效的。在后端模板中，Web 开发者只要调用相关函数就能把合适的技术插入 HTML 文档响应中。将来还可以在开发框架中提供这些高级方案的支持，让这些技术获得更广泛的应用。

第 5 章

整合异步脚本

Coupling Asynchronous Scripts

第 4 章解释了异步加载脚本的原理。脚本如果按常规方式加载（`<script src="url"></script>`），不仅会阻塞页面中其他内容的下载，还会阻塞脚本后面所有元素的渲染。异步加载脚本可以避免这种阻塞现象，从而提高页面加载速度。

无阻塞加载脚本带来的性能提升是要付出代价的。代码异步执行时可能会出现竞争状态（race conditions）。在使用外部脚本时，我们关注依赖外部脚本里定义的标识符的行内脚本，如果外部脚本异步加载而不考虑行内代码的依赖，可能会由于竞争状态而导致出现未定义标识符的错误。

当异步加载的外部脚本与行内脚本之间存在代码依赖时，我们必须通过一种保证执行顺序的方法来整合这两个脚本。很显然，并没有一个能跨所有浏览器实现的简单方法。本章提出了这个问题，并将在以下几节给出一些解决方案：

“代码示例：menu.js”第 42 页

这节给出了一个贯穿整章的例子。它构建了一个行内脚本依赖外部脚本的使用场景。

“竞争状态”第 44 页

通过测试第 4 章的那些异步加载脚本技术，说明当行内脚本有代码依赖时会产生未定义标识符的错误。这表明我们需要一些整合外部和行内脚本的技术。

“异步保持顺序”第 45 页

描述了 5 种整合行内脚本和其依赖的异步加载外部脚本的技术。

“多个外部脚本”第 52 页
当有多个外部脚本相互依赖，而且后面有存在代码依赖的行内脚本时，问题将变得更为复杂。这一节提出了两个解决方案。

“通用解决方案”第 59 页
在深入地理解权衡之后，我们结合各种最佳实践来解决跨所有主流浏览器的单个脚本或多个脚本的整合问题。

“现实工作中的异步加载”第 63 页
探讨现实工作中两个异步脚本整合行内代码的实例：一个是用 Dojo 封装 Google 分析代码（译注 1），另一个是 YUI Loader。

5.1 代码示例：menu.js

Code Example: menu.js

在第 4 章讨论的特性之一是确保执行顺序，着重讨论外部脚本的执行顺序。但大多数网页在加载外部脚本的同时也包含使用了外部脚本定义标识符的行内脚本，例如使用 Google 分析（*http://www.google.com/analytics/*）或流行的 JavaScript 框架（例如 JQuery、Yahoo! UI Library（译注 2））的页面。

为了说明这种情况，我创建了 Normal Script Src 示例，它有一个外部脚本，然后是对其存在代码依赖的行内脚本。外部脚本 menu.js 提供了生成下拉菜单的功能，如图 5-1 所示。

Normal Script Src
http://stevesouders.com/efws/couple-normal.php

以下代码展示了 Normal Script Src 示例按常规方式加载 menu.js，并通过后面的行内脚本创建菜单。行内脚本定义了菜单项的数组 `aExamples`。init 函数调用 `EFWS.Menu.createMenu` 方法，并传入了元素 id（`'examplesbtn'`）和菜单项数组。菜单将会附加在`'examplesbtn'` 元素上，即页面上的“Examples”按钮上。

```
<script src="menu.js" type="text/javascript"></script>

<script type="text/javascript">
var aExamples =
    [
      ['couple-normal.php', 'Normal Script Src'],
      ['couple-xhr-eval.php', 'XHR Eval'],
      ...
    ];

function init() {
```

译注 1：Google Analytics，*http://www.google.com/analytics/*。
译注 2：JQuery（*http://jquery.com*）是当今最流行的 JavaScript 框架；Yahoo! UI Library（*http://developer.yahoo.com/yui/*）简称 YUI，由美国雅虎专职团队开发与维护，国内使用 YUI 的网站有淘宝网、口碑网等。

```
    EFWS.Menu.createMenu('examplesbtn', aExamples);
}

init();
</script>
```

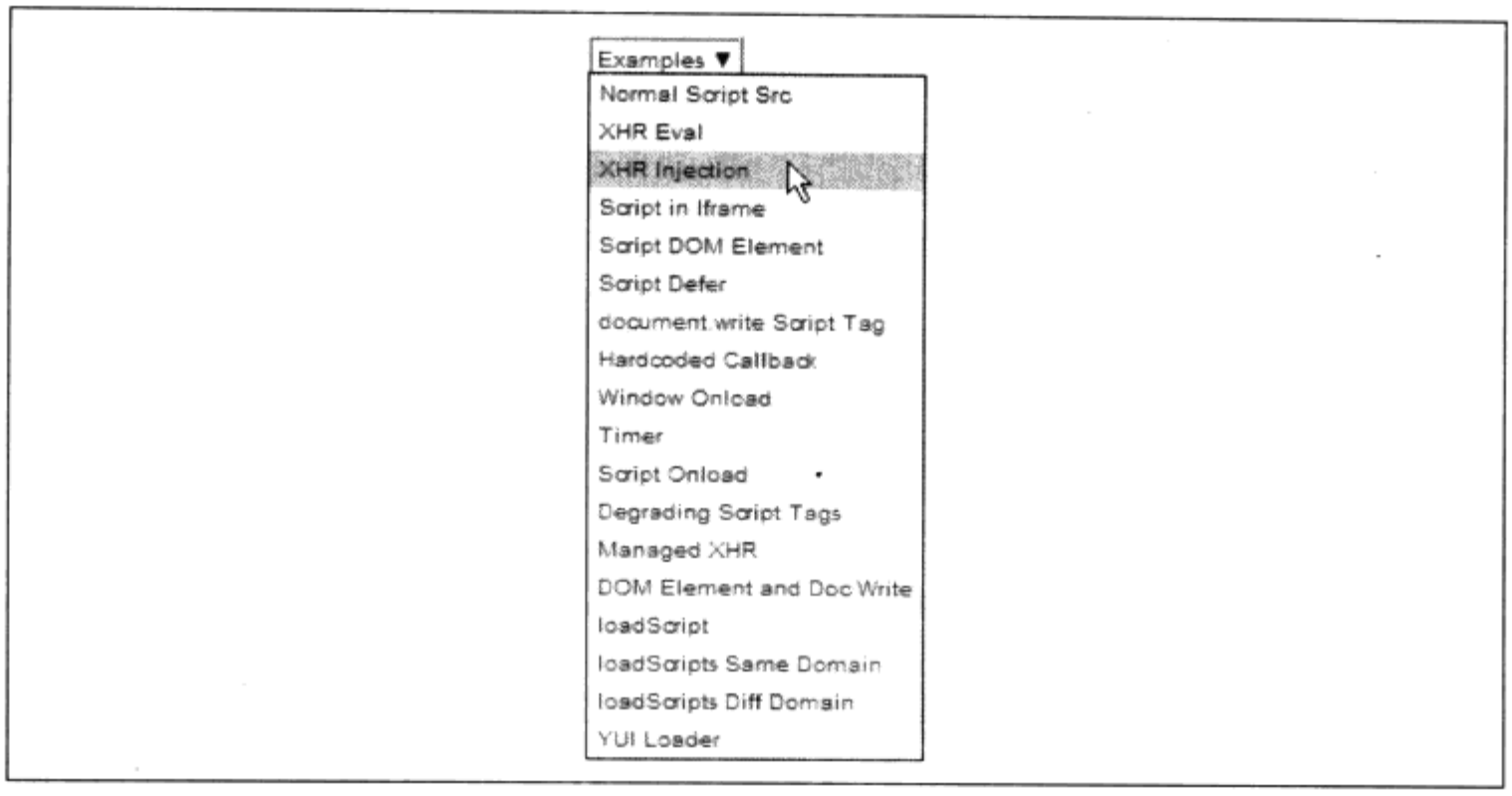


图 5-1：menu.js 示例

menu.js 示例模拟的场景更生动地描述了本章的内容，如今很多网站也确实会遇到这种情况。示例页面包含一个外部脚本和一段行内脚本。因为行内脚本依赖外部脚本，所以保证执行顺序至关重要——外部脚本必须在行内脚本之前下载、解析和执行。

此外，该示例也非常适合采用异步加载脚本方式，这样就不会阻塞页面中其他资源的下载了。这个例子中的“其他资源”就是一张图片。这个图片的下载时间设置成 1 秒，同时 menu.js 设置成两秒。若脚本按常规方式加载，它就会阻塞图片下载，如图 5-2 所示。

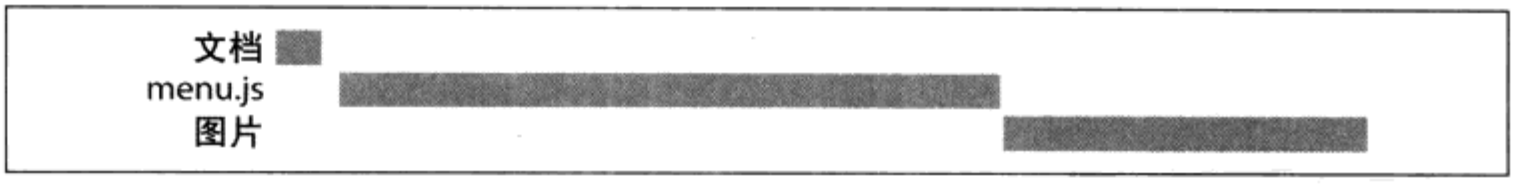


图 5-2：Normal Script Src 示例的 HTTP 瀑布图

如果 menu.js 采用异步加载，那么就不会阻塞图片的加载，页面会加载得更快。此外，menu.js

是一个很适合异步加载的脚本，因为它不渲染任何可见的页面内容，只提供了在页面渲染完成后才能使用的功能。问题来了：我们能否异步加载 menu.js 而又不让行内脚本产生任何未定义标识符的错误呢？

5.2 竞争状态
Race Conditions

Normal Script Src 示例没有产生任何未定义标识符的错误，但 menu.js 阻塞图片下载，这导致了页面加载变慢。而要提升性能，最好是异步加载 menu.js，但我们必须先判断执行顺序是否得到了保证，竞争状态是否会产生未定义标识符的错误。

我修改了 Normal Script Src 示例，使用了第 4 章中介绍过的无阻塞技术。在每个例子中，我会通过程序来回答这两个问题：是无阻塞加载脚本的吗？保证执行顺序了吗？

XHR Eval
http://stevesouders.com/efws/couple-xhr-eval.php

XHR 注入（XHR Injection）
http://stevesouders.com/efws/couple-xhr-injection.php

Script in Iframe
http://stevesouders.com/efws/couple-script-iframe.php

Script DOM Element
http://stevesouders.com/efws/couple-script-dom.php

Script Defer
http://stevesouders.com/efws/couple-script-defer.php

document.write Script Tag
http://stevesouders.com/efws/couple-doc-write.php

表 5-1 显示了这些例子在主流浏览器的执行结果。没有一种技术既能并行下载又能保持执行顺序，唯一的特例是在 Firefox 中执行的 Script DOM Element 示例。

表 5-1：确保外部和行内脚本的执行顺序

技术	并行下载脚本和图片	确保执行顺序
Normal Script Src	IE8、Saf4、Chr2	IE、FF、Saf、Chr、Op
XHR Eval	IE、FF、Saf、Chr、Op	--
XHR 注入	IE、FF、Saf、Chr、Op	--
Script in Iframe	IE、FF、Saf、Chr、Op[a]	--
Script DOM Element	IE、FF、Saf、Chr	FF、Op
Script Defer	IE、(Saf4、Chr2)[b]	FF、Saf、Chr、Op

续表

技术	并行下载脚本和图片	确保执行顺序
document.write Script Tag	Saf4、Chr2	IE、FF、Saf、Chr、Op

a Opera 中一个有趣的性能提升是不但能并行下载 iframe 中的 script，而且能并行执行代码。
b 在这些新型浏览器里，脚本默认并行下载，并非 defer 属性在起作用。

提示：缩写如下：(Chr) Chrome 1.0.154 和 2.0.156；(FF) Firefox 2.0、3.0 和 3.1；(IE) Internet Explorer 6、7 和 8；(Op) Opera 9.63 和 10.00 alpha；(Saf) Safari 3.2.1 和 4.0 (developer preview)。

新型浏览器展现了一个光明的前景。Internet Explorer 8、Safari4 和 Chrome2 实现了当采用常规 script 标签（<script src="*url*"></script>）方式时并行下载，同时保持执行顺序。可是在 Internet Explorer 8 和 Chrome2 中，脚本仍然阻塞某些资源的加载，例如这些测试页中的图片。给现在依旧流行的包括 Internet Explorer 6 和 Internet Explorer 7 在内的主流浏览器提速，仍然显得重要，甚至是重中之重。我们需要的是一种能异步加载脚本**并且**保持执行顺序的跨浏览器的解决方案，接下来一节描述的整合技术也就顺理成章了。

5.3 异步加载脚本时保持执行顺序

Preserving Order Asynchronously

当外部脚本按常规方式加载时，它会阻塞行内代码的执行，竞争状态不会发生。一旦我们开始异步加载脚本，就需要一种技术把行内代码和依赖行内代码的外部脚本整合起来。这些整合技术是：

- 硬编码回调（Hardcoded Callback）。
- Window Onload。
- 定时器（Timer）。
- Script Onload。
- Degrading Script Tags。

Script Onload 技术可能是最好的选择，不过为了证实这个结论，我们还是先来看看其他一些技术。

本节的整合示例使用了第 4 章介绍的 Script DOM Element 方法作为异步加载技术。该方法通过 JavaScript 来创建一个 script 元素，并把它的 src 属性设置为 menu.js。这里展示的代码来自于 Script DOM Element 示例（*http://stevesouders.com/efws/couple-script-dom.php*）：

```
<script type="text/javascript">
var domscript = document.createElement('script');
domscript.src = "menu.js";
```

```
document.getElementsByTagName('head')[0].appendChild(domscript);
</script>

<script type="text/javascript">
var aExamples = [['couple-normal.php', 'Normal Script Src'],...];

function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
}

init();
</script>
```

我认为这是首选的无阻塞技术，因为它小巧实用，而且脚本可以跨域加载（脚本所在的域可以和当前页面的域不同）。如图 5-3 所示，这种技术成功并行下载了外部脚本（2 秒）和图片（1 秒）。但是这个方法在 Internet Explorer、Safari、Chrome 中产生了未定义标识符的错误，因为行内代码在异步加载的脚本到达之前就执行了。Script DOM Element 方法在这 3 个浏览器里不能保证执行顺序，正如它们未出现在表 5-1“确保执行顺序”栏那样。下面几节讨论的整合技术可以解决这些竞争状态问题。

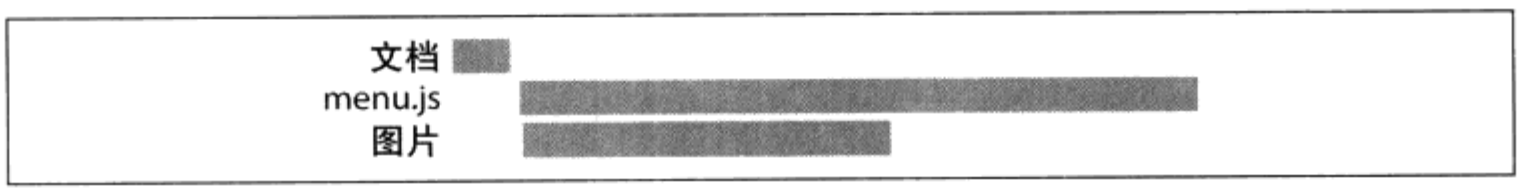


图 5-3：Script DOM Element HTTP 瀑布图

5.3.1 技术 1：硬编码回调（Hardcoded Callback）

一种简单的整合技术是，让外部脚本调用行内代码里的函数。在 Hardcoded Callback 示例中，我们在外部脚本（现在是 menu-with-init.js）底部添加了一个对 init 的调用。

Hardcoded Callback

http://stevesouders.com/efws/hardcoded-callback.php

行内代码需要少许修改。移除对 `init` 的调用——改为在外部脚本中调用它。`aExamples` 和 `init` 的定义移至 menu-with-init.js 的上面，从而使得它们在 menu-with-init.js 完成加载时是可用的。

```
<script type="text/javascript">
var aExamples = [['couple-normal.php', 'Normal Script Src'],...];

function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
```

```
}

var domscript = document.createElement('script');
domscript.src = "menu-with-init.js";
document.getElementsByTagName('head')[0].appendChild(domscript);
</script>
```

如果 Web 开发者能同时控制主页面和外部脚本，这种技术也是可行的。但是我们往往不太可能把回调嵌入第三方 JavaScript 模块中，而且这种方法也不是很灵活——改变回调接口时需要调整外部脚本。

5.3.2 技术 2：Window Onload

Technique 2: Window Onload

该技术指通过监听 window 的 onload 事件来触发行内代码的执行。这使得只要确保外部脚本在 `window.onload` 之前下载执行就能保持执行顺序。有些（但不是全部）异步加载技术能确保这点：

- Script in Iframe 技术在 Internet Explorer、Firefox、Safari、Chrome 和 Opera 中保持执行顺序。
- Script DOM Element 技术在 Firefox、Safari 和 Chrome 中保持执行顺序。
- Script Defer 技术在 Internet Explorer 中保持执行顺序。

使用其中一种技术，再通过 `window.onload` 触发行内脚本就可以实现在并行下载的同时保持执行顺序，Window Onload 示例演示了这种技术。

Window Onload

http://stevesouders.com/efws/window-onload.php

该示例使用 Script in Iframe 方法来加载外部脚本，因为几乎在所有浏览器中它都会阻塞 `onload` 事件。代码被嵌入 menu.php 中，然后用 iframe 加载它而不是直接加载 menu.js。行内脚本经过修改之后，`init` 绑定到 window 的 `onload` 事件上。依据浏览器的差异来选用 `addEventListener` 或 `attachEvent`，这比简单地用 `window.onload=init` 好一些，因为它能确保任何已经存在的 `onload` 事件处理程序不受影响。

```
<iframe src="menu.php" width=0 height=0 frameborder=0></iframe>

<script type="text/javascript">
var aExamples = [['couple-normal.php', 'Normal Script Src'],...];

function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
}

if ( window.addEventListener ) {
    window.addEventListener("load", init, false);
```

```
}
else if ( window.attachEvent ) {
    window.attachEvent("onload", init);
}
</script>
```

Window Onload 整合技术存在两个缺点。首先，你必须得确认异步脚本是通过阻塞 onload 事件的方式加载的。(这就是为什么我没有使用首选的 Script DOM Element 技术而改用 Script in Iframe 的原因。) 其次，可能会导致行内代码延迟执行。如果页面还有更多的资源（图片、Flash 等），那么外部脚本可能在 onload 事件触发之前早就完成加载了。一般来说，行内脚本最好在外部脚本下载和执行完成之后立即调用。在这个示例中，越早调用行内代码可以越早显示菜单。

5.3.3 技术 3：定时器（Timer）

Technique 3: Timer

定时器技术指使用轮询方法来保证在行内代码执行之前所依赖的外部脚本已经加载。如 Timer 示例中所示，我们使用了 setTimeout 方法。

Timer

http://stevesouders.com/efws/timer.php

在本例中修改行内代码使其包含一个新函数 initTimer，由该函数来检查依赖的名字空间（EFWS）是否已经存在。若存在，就调用 init 函数。不存在，就在一个指定的时间段（300 毫秒）之后再次调用 initTimer 函数。

```
<script type="text/javascript">
var domscript = document.createElement('script');
domscript.src = "menu.js";
document.getElementsByTagName('head')[0].appendChild(domscript);

var aExamples = [['couple-normal.php', 'Normal Script Src'],...];

function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
}

function initTimer() {
    if ( "undefined" === typeof(EFWS) ) {
        setTimeout(initTimer, 300);
    }
    else {
        init();
    }
}

initTimer();
</script>
```

如果在 setTimeout 方法中设置的时间数值太小，该轮询技术可能会增加页面开销。相反，设置太大会导致在外部脚本加载完成和行内代码开始执行之间产生一个不希望发生的延迟。这段“简陋”的代码无法处理的一个极端问题是，当 menu.js 加载失败时，轮询会无限进行下去。最后，这个方法会稍稍增加一些维护成本，因为需要通过外部脚本的特定标识符来判断该脚本是否已经加载完毕。如果外部脚本的标识符变了，行内代码也需要同步更新。

5.3.4 技术 4：Script Onload

Technique 4: Script Onload

前面提到的整合技术会增加页面的脆弱性（brittleness）、延迟和开销。Script Onload 方法通过监听脚本的 onload 事件解决了所有这些问题。

Script Onload

http://stevesouders.com/efws/script-onload.php

本例的变化是加入了 script 元素的 onload 和 onreadystatechange（译注 3）事件处理程序，都设置为调用 init 函数。我们通过添加 onloadDone 标记来防止 init 函数在 Opera 里被调用两次。

```
<script type="text/javascript">
var aExamples = [['couple-normal.php', 'Normal Script Src'],...];

function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
}

var domscript = document.createElement('script');
domscript.src = "menu.js";
domscript.onloadDone = false;
domscript.onload = function() {
    domscript.onloadDone = true;
    init();
};
domscript.onreadystatechange = function() {
    if (( "loaded" === domscript.readyState||"complete"=== domscript.readyState)&&!
      domscript.onloadDone ) {
        domscript.onloadDone = true;
        init();
    }
}
document.getElementsByTagName('head')[0].appendChild(domscript);
</script>
```

Script Onload 技术是整合异步加载外部脚本和行内脚本的首选。它不引用任何外部脚本里的标识符，所以维护更简单。行内代码可以在外部脚本加载完毕后立即执行。事件处理也非常简单。

译注 3：onreadystatechange 在 Internet Explorer 中有效，onload 在其他浏览器中有效，而 Opera 比较特殊，两者都有效，所以程序中需要做标记来处理。

5.3.5 技术 5：降级使用 script 标签（Degrading Script Tags）

Technique 5: Degrading Script Tags

这个技术基于 John Resig 的一篇博客文章《降级使用 script 标签》（*http://ejohn.org/blog/degrading-script-tags/*）而来。John 是来自 Mozilla 的 JavaScript 布道士，流行 JavaScript 框架 jQuery 的创造者。他将该技术描绘成一个整合外部 jQuery 脚本和使用了 jQuery 标识符的行内代码的方法。该模式使用 script 标签来包含外部脚本和使用它的行内代码，其代码如下：

```
<script src="jquery.js" type="text/javascript">
jQuery("p").addClass("pretty");
</script>
```

这个想法是让行内代码在外部脚本加载成功之后执行。该模式有几个优点：

更干净

只有一个 script 标签，而不像通常那样需要两个。

更清晰

行内代码对外部脚本的依赖更一目了然。

更安全

如果外部脚本加载失败，行内代码不会执行，避免了未定义标识符的错误。

它有一个缺点：当今的浏览器不支持这种语法！John 证实浏览器可以加载外部脚本但会忽略行内代码。不过他提供了一个代码样例以表明在外部脚本中增加一点点代码就可以让行内代码生效。我在 Degrading Script Tags Normal 示例中应用了该技术。

Degrading Script Tags Normal

http://stevesouders.com/efws/degrading-script-tag-normal.php

行内脚本使用了 John 的模式。这里使用 `script` 标签指定外部脚本，插入有依赖的行内代码：

```
<script src="menu-degrading.js" type="text/javascript">
var aExamples = [['couple-normal.php', 'Normal Script Src'],...];

function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
}

init();
</script>
```

要让该模式生效，必须在 menu-degrading.js 末尾增加几行 JavaScript 代码。这些代码遍历页面中的所有 script 元素，搜索其中 `src` 包含“menu-degrading.js”的那个。从根本上说外部脚本就是在 DOM 中搜索它本身。当它找到正确的 script 元素时，就执行 script 的 `innerHTML`。

```
var scripts = document.getElementsByTagName("script");
var cntr = scripts.length;
while ( cntr ) {
    var curScript = scripts[cntr-1];
    if ( -1 != curScript.src.indexOf("menu-degrading.js") ) {
        eval( curScript.innerHTML );
        break;
    }
    cntr--;
}
```

Degrading Script Tags Normal 示例（*http://stevesouders.com/efws/degrading-script-tag-normal.php*）在所有被测试的浏览器里都能正常运行：Internet Explorer 6 到 8、Firefox2 和 3、Safari3 和 4、Chrome1 和 2、Opera9 和 10。然而，外部脚本并非异步加载的。（注意：图片 3 秒后才加载到页面中，而不是 1 秒）。为了解决脚本的阻塞问题，需要把其中一种异步加载技术和该模式结合起来。我已经在 Degrading Script Tags Async 示例中实现了它。

Degrading Script Tags Async

http://stevesouders.com/efws/degrading-script-tag.php

这个例子使用相同的外部脚本 menu-degrading.js，里面有额外的代码用于查找 script 标签并执行它的 `innerHTML`。与用 `script` 标签加载外部脚本不同，我们使用了 Script DOM Element 无阻塞技术。行内代码通过动态设置 script 元素的 `text` 属性（Opera 里用 `innerHTML`）来执行“`init()`”。

```
<script type="text/javascript">
var aExamples = [['couple-normal.php', 'Normal Script Src'],...];
function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
}

var domscript = document.createElement('script');
domscript.src = "menu-degrading.js";
if ( -1 != navigator.userAgent.indexOf("Opera") ) {
    domscript.innerHTML = "init();";
}
else {
    domscript.text = "init();";
}
document.getElementsByTagName('head')[0].appendChild(domscript);
</script>
```

我喜欢这种技术的优雅和简洁。但是这个模式相比 Script Onload 技术来说鲜为人知，它更像是给开发者的一个惊喜。它开销更小（没有使用事件处理程序），提供了一个实用而优雅的机制，尤其是在外部脚本非异步加载的情况下。它主要的缺点是该技术需要修改外部脚

本，有时这不一定能做到，尤其是在使用第三方 JavaScript 库的时候。就现在看来，Script Onload 整合技术是最好的选择。

5.4 多个外部脚本

Multiple External Scripts

到现在为止，所有示例都重在研究如何整合单个外部脚本和行内代码。当 JavaScript 框架被包含进单个文件中的情况下该整合方案相当好用，例如 Google 分析（*http://google-analytics.com/ga.js*）和 jQuery（*http://ajax.googleapis.com/ajax/libs/jquery/1.3.0/jquery.min.js*）。可是我们经常有多个外部脚本和行内脚本，它们都需要按指定的顺序执行。到现在为止无论在本章还是第 4 章里，都还没描述过这种异步加载多个脚本时可以保持顺序的技术。主要由于浏览器不兼容，对于这个问题还没有一个完美的解决方案。

本节将描述两种最好的异步加载多个外部脚本的技术，同时保持外部脚本和行内脚本的执行顺序。Managed XHR 是一个选择，但是它的限制在于脚本必须和主页面同域。DOM Element 和 Doc Write 技术虽然可以跨域工作，但是代码在不同的浏览器中会有不同的表现，并且这两种技术不能跨浏览器实现异步加载所有类型的资源。

为了得到一个使用多个脚本的例子，我创建了 menutier.js。这个新脚本扩展了菜单功能，提供了一个多级分组菜单，如图 5-4 所示（注意带阴影的分组标题）。另外，menutier.js 依赖 menu.js，因此必须保持它们的执行顺序。通过在行内代码中调用 `EFWS.Menu.createTieredMenu` 可以创建一个多级菜单。这样就建立了我们尝试分析的应用场景：多个外部脚本和行内脚本必须按特定顺序执行。此外，menutier.js 被配置成在它依赖的 menu.js 之前返回，那么我们是不是遇到麻烦了呢？让我们看看 Managed XHR、DOM Element 和 Doc Write 技术是如何并行加载外部脚本同时还能保持执行顺序的。

5.4.1 Managed XHR

“Managed XHR”是第 4 章提到的一种异步加载技术的名称，可以管理 `XMLHttpRequest`（XHR）的请求和响应。管理代码用于控制忙指示器和保持执行顺序。我在第 4 章没有介绍任何代码，不过这一节里我将展示实现执行顺序这部分的内容。

XHR 注入技术在所有浏览器中都不能保持执行顺序，如表 5-1 所示。`EFWS.Script` 模块封装了这种技术，把 XHR 响应加入队列来保证它们按顺序执行。实现这个需要 100 行以内的代码：

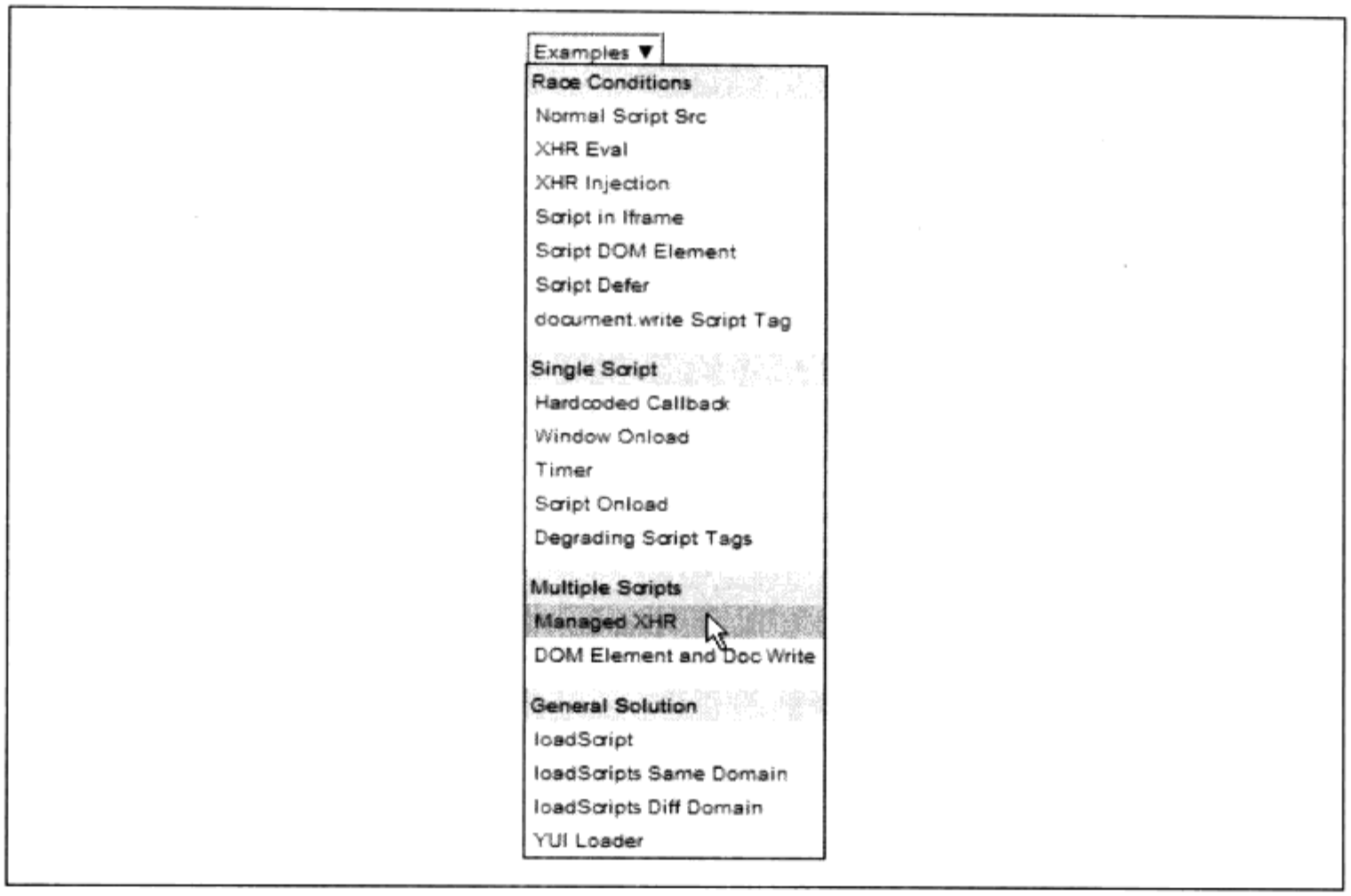


图 5-4：menutier.js 示例

```
<script type="text/javascript">
EFWS.Script = {
    queuedScripts: new Array(),

loadScriptXhrInjection: function(url, onload, bOrder) {
    var iQ = EFWS.Script.queuedScripts.length;
    if ( bOrder ) {
        var qScript = {response: null, onload: onload, done: false};
        EFWS.Script.queuedScripts[iQ] = qScript;
    }

    var xhrObj = EFWS.Script.getXHRObject();
    xhrObj.onreadystatechange = function() {
        if ( xhrObj.readyState == 4 ) {
            if ( bOrder ) {
                EFWS.Script.queuedScripts[iQ].response =
                    xhrObj.responseText;
                EFWS.Script.injectScripts();
            }
            else {
                eval(xhrObj.responseText);
                if ( onload ) {
                    onload();
                }
            }
        }
```

```
        };
        xhrObj.open('GET', url, true);
        xhrObj.send('');
    },

    injectScripts: function() {
        var len = EFWS.Script.queuedScripts.length;
        for ( var i = 0; i < len; i++ ) {
            var qScript = EFWS.Script.queuedScripts[i];
            if ( ! qScript.done ) {
                if ( ! qScript.response ) {
                    //停止！需要等待响应返回
                    break;
                }
                else {
                    eval(qScript.response);
                    if ( qScript.onload ) {
                        qScript.onload();
                    }
                    qScript.done = true;
                }
            }
        }
    },

    getXHRObject: function() {
        var xhrObj = false;
        try {
            xhrObj = new XMLHttpRequest();
        }
        catch(e){
            var aTypes = ["Msxml2.XMLHTTP.6.0",
                          "Msxml2.XMLHTTP.3.0",
                          "Msxml2.XMLHTTP",
                          "Microsoft.XMLHTTP"];
            var len = aTypes.length;
            for ( var i=0; i < len; i++ ) {
                try {
                    xhrObj = new ActiveXObject(aTypes[i]);
                }
                catch(e) {
                    continue;
                }
                break;
            }
        }
        finally {
            return xhrObj;
        }
    }
};
</script>
```

数组 queuedScripts 存储执行队列中的脚本。队列中的每个脚本是一个拥有 3 个属性的对象：

Response
: XHR 响应（JavaScript 字符串）。

Onload
: 脚本加载后触发的函数（可选）。

bOrder
: 如果该脚本需要依赖其他脚本按顺序执行，则设为 true（默认是 false）。

开发者调用 EFWS.Script.loadScriptXhrInjection，传入需要加载的外部脚本的 URL、onload 函数和一个表明是否保持执行顺序的布尔值。如果不需要按顺序执行，XHR 的响应返回时立即被注入页面中执行。当需要按顺序执行时，XHR 响应先被添加到数组 queuedScripts 中，然后调用 EFWS.Script.injectScripts。该函数遍历队列中的脚本，当发现一个所有依赖脚本已加载但尚未执行的响应时，立即把它注入页面执行。Managed XHR 示例演示了这段代码。

Managed XHR
: *http://stevesouders.com/efws/managed-xhr.php*

下面是一段修改过的行内脚本。前面的几行和之前的示例相似，它创建了菜单项数组和 URL。init 函数调用 EFWS.Menu.createTieredMenu。最后两行是使用 Managed XHR 技术的地方：

```
<script type="text/javascript">
var aRaceConditions = [['couple-normal.php', 'Normal Script Src'], ...];
var aWorkarounds = [['hardcoded-callback.php', 'Hardcoded Callback'], ...];
var aMultipleScripts = [['managed-xhr.php', 'Managed XHR'], ...];
var aLoadScripts = [['loadscript.php', 'loadScript'], ...];
var aSubmenus =
    [
     ["Race Conditions", aRaceConditions],
     ["Workarounds", aWorkarounds],
     ["Multiple Scripts", aMultipleScripts],
     ["General Solution", aLoadScripts]
    ];

function init() {
  EFWS.Menu.createTieredMenu('examplesbtn', aSubmenus);
}

EFWS.Script.loadScriptXhrInjection("menu.js", null, true);
EFWS.Script.loadScriptXhrInjection("menutier.js", init, true);
</script>
```

第 1 次调用 `EFWS.Script.loadScriptXhrInjection` 加载 menu.js，同时保持执行顺序。第 2 次调用下载 menutier.js，也被指定为按顺序加载，并且传入 `init` 作为脚本的 `onload` 函数。

HTTP 瀑布图 5-5 显示了一段较短的 HTML 文档请求，后面是该页中 3 个资源的请求：menu.js（两秒响应时间）、menutier.js（1 秒响应时间）和图片（1 秒响应时间）。页面中所有资源都并行加载而且保持执行顺序（没有未定义标识符的错误发生）。

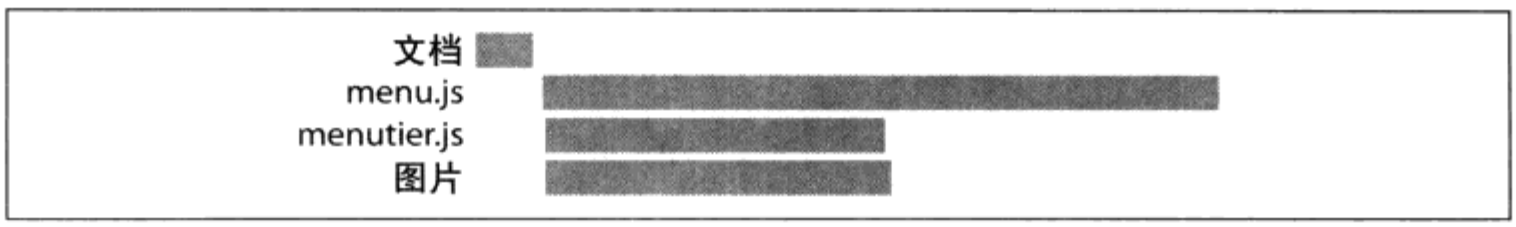


图 5-5：Managed XHR HTTP 瀑布图

Managed XHR 技术解决了跨主流浏览器的兼容性问题。但是由于 `XMLHttpRequest` 的同源规则（注 1），如果外部脚本部署在与主页面不同的域下，该技术就无法工作。所以当脚本和页面不同域时，我们选择的方案是 DOM Element 和 Doc Write 技术。

5.4.2 DOM Element 和 Doc Write

DOM Element and Doc Write

Managed XHR 技术非常适用于按特定顺序加载外部脚本和行内脚本的使用场景，而且不阻塞页面中其他资源。但是它只能用于和主页面同域的脚本。把外部脚本部署在与主页面不同的域下是很常见的，尤其是当脚本部署在内容分发网络（CDN）上，或者使用了第三方 JavaScript 库时。DOM Element 和 Doc Write 示例创建了这种使用场景：从 *http://souders.org* 获取 menu.js 和 menutier.js，而主页面仍然放在 *http://stevesouders.com*。

DOM Element and Doc Write

http://stevesouders.com/efws/dom-and-docwrite.php

有 3 种异步加载技术可以用于加载跨域脚本：Script DOM Element、Script Defer 和 `document.write` Script Tag（详见第 4 章对每一种技术的描述）。这些技术在不同的浏览器下有不同的表现。表 5-2 按优先级顺序罗列了对以下 3 个特性的测试结果：

注 1：*http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy_for_XMLHttpRequest*。

- 是否能保持脚本执行顺序?
- 是否能并行加载脚本?
- 是否能把脚本和其他资源（图片、样式表等）一起并行加载?

表 5-2：异步加载脚本并同时保持顺序

技术	保持顺序	并行加载脚本	并行加载其他资源
Script DOM Element	FF、Op	FF、Op、IE、Saf、Chr	IE、FF、Saf、Chr
Script Defer	IE、Saf、Chr、FF、Op	IE	IE
document.write Script Tag	IE、Saf、Chr、FF、Op	IE、Op	

提示：缩写如下: (Chr) Chrome 1.0.154 和 2.0.156; (FF) Firefox 2.0、3.0 和 3.1; (IE) Internet Explorer 6、7 和 8; (Op) Opera 9.63 和 10.00 alpha; (Saf) Safari 3.2.1 和 4.0 (developer preview)。

Script DOM Element 在 Firefox 和 Opera 中是首选的技术，所有其他情况下可以使用 document.write Script Tag 技术。即使在 Internet Explorer 中，我们也不使用 Script Defer 技术，因为当它和 DHTML 技术结合之后会产生意外的行为。我扩展了 EFWS.Script 模块，融入了这些技术：

```
EFWS.Script = {
    loadScriptDomElement: function(url, onload) {
        var domscript = document.createElement('script');
        domscript.src = url;
        if ( onload ) {
            domscript.onloadDone = false;
            domscript.onload = onload;
            domscript.onreadystatechange = function() {
                if (( "loaded" === domscript.readyState || "complete"===
                      domscript.readystate)&&! domscript.onloadDone) {
                    domscript.onloadDone = true;
                    domscript.onload();
                }
            }
        }
        document.getElementsByTagName('head')[0].appendChild(domscript);
    },

    loadScriptDocWrite: function(url, onload) {
        document.write('<scr' + 'ipt src="' + url +
                       '" type="text/javascript"></scr' + 'ipt>');
        if ( onload ) {
            EFWS.addHandler(window, "load", onload);
        }
    },

    queuedScripts: new Array(),
    loadScriptXhrInjection: function(url, onload, bOrder) { ... },
    injectScripts: function() { ... },
```

```
    getXHRObject: function() { ... }
};

EFWS.addHandler = function(elem, type, func) {
    if ( elem.addEventListener ) {
        elem.addEventListener(type, func, false);
    }
    else if ( elem.attachEvent ) {
        elem.attachEvent("on" + type, func);
    }
};
```

除了无阻塞加载脚本并且保持执行顺序之外，我们还想把外部脚本和行内代码整合起来，毕竟这才是本章的重点。在 `EFWS.Script.loadScriptDomElement` 中，通过添加外部脚本的 `onload` 和 `onreadystatechange` 的回调做到了这点，如第 49 页的“技术 4：Script Onload”所述。我们还是使用 Window Onload 作为 `EFWS.Script.loadScriptDocWrite` 的整合技术，虽然并不理想，但是当用 `document.write` 来插入外部脚本时其他技术都不可行。

本节示例中的行内代码使用了这些新技术，基于浏览器分为不同的情况：

```
<script type="text/javascript">
var aRaceConditions = [['couple-normal.php', 'Normal Script Src'], ...];
var aWorkarounds = [['hardcoded-callback.php', 'Hardcoded Callback'], ...];
var aMultipleScripts = [['managed-xhr.php', 'Managed XHR'], ...];
var aLoadScripts = [['loadscript.php', 'loadScript'], ...];
var aSubmenus = [["Race Conditions", aRaceConditions], ...];

function init() {
    EFWS.Menu.createTieredMenu('examplesbtn', aSubmenus);
}
if ( -1 != navigator.userAgent.indexOf('Firefox') ||
     -1 != navigator.userAgent.indexOf('Opera') ) {
     EFWS.Script.loadScriptDomElement("http://souders.org/efws/menu.js");
     EFWS.Script.loadScriptDomElement("http://souders.org/efws/menutier.js", init);
}
else {
     EFWS.Script.loadScriptDocWrite("http://souders.org/efws/menu.js");
     EFWS.Script.loadScriptDocWrite("http://souders.org/efws/menutier.js", init);
}
</script>
```

我们结合 Script DOM Element 和 `document.write` Script Tag 技术完成了主要目标：外部脚本的执行顺序在所有浏览器下都能保持，行内代码和它依赖的外部脚本成功整合。异步加载在各浏览器中实现的程度有所不同：

- Firefox 并行加载所有资源。
- Internet Explorer 和 Opera 并行加载所有脚本，但阻塞其他资源（图片、样式表等）。

- 在 Safari 和 Chrome 中是个混合的结果。在 Safari 3.2 和 Chrome 1.0 中不能并行加载所有资源。在 Safari 4 和 Chrome 2.0 中使用这些相同的技术可以并行加载所有资源。

综上所述，没有简单的跨浏览器方案可以异步加载多个脚本，还同时保持执行顺序。把所有脚本整合到一个单独的脚本中是一个值得考虑的选择。这是《高性能网站建设指南》（“规则 1：更少的 HTTP 请求”）中的一条最佳实践，因为这样可以减少下载时间，另外的好处是异步加载单个脚本并整合行内代码的方案更健壮。

5.5 综合解决方案
General Solution

本章展示了很多技术，以及对应的网页示例和代码样例。这对于理解各种技术之间的权衡很有意义，不过我们需要的是一个处理异步加载脚本的综合解决方案，它能保持执行顺序并整合行内代码。我在 `EFWS.Script` 功能基础上新添加了两个函数来封装所有细节：`EFWS.Script.loadScript` 用于加载单个脚本，`EFWS.Script.loadScripts` 用于加载多个脚本。

5.5.1 单个脚本
Single Script

异步加载单个脚本的最佳技术是 Script DOM Element。它能跨所有浏览器工作并且小巧实用。Script Onload 模式是用于整合行内代码和单个外部脚本的最好选择。`EFWS.Script.loadScriptDomElement` 实现了以上这两方面的技术，单个脚本的综合解决方案只是封装了如下函数：

```
EFWS.Script = {
    loadScript: function(url, onload) {
        EFWS.Script.loadScriptDomElement(url, onload);
    },

    loadScriptDomElement: function(url, onload) { ... },
    loadScriptDocWrite: function(url, onload) { ... },
    queuedScripts: new Array(),
    loadScriptXhrInjection: function(url, onload, bOrder) { ... },
    injectScripts: function() { ... },
    getXHRObject: function() { ... }
};
```

这在很大程度上简化了 menu.js 示例。代码变为只有几行——菜单项数组、init 函数和对 `EFWS.Script.loadScript` 的调用：

```
<script type="text/javascript">
var aExamples = [['couple-normal.php', 'Normal Script Src'],...];

function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
```

```
    }

    EFWS.Script.loadScript("menu.js", init);
    </script>
```

loadScript 示例演示了该代码。

loadScript
http://stevesouders.com/efws/loadscript.php

5.5.2 多个脚本
Multiple Scripts

多脚本综合解决方案的函数是 `EFWS.Script.loadScripts`。第 52 页“多个外部脚本”探讨了针对这种情况的技术。Managed XHR 技术是当脚本来自与主页面相同的域时首选的方案。由于可选余地很小及浏览器的不兼容性，要在从不同域异步加载脚本的同时保持执行顺序显得更为棘手。第 56 页“DOM Element and Doc Write”里描述的方法是依据不同浏览器使用了 Script DOM Element 和 `document.write` Script Tag 技术。下面有两个使用了新的 `EFWS.Script.loadScripts` 函数的例子用以演示这两种情况。

loadScripts Same Domain
http://stevesouders.com/efws/loadscripts-same.php

loadScripts Different Domain
http://stevesouders.com/efws/loadscripts-diff.php

下面是 `EFWS.Script.loadScripts` 的代码。`EFWS.Script.loadScripts` 接收一个包含脚本 URL 的数组和一个函数为参数，该函数在最后一个外部脚本执行完毕之后调用。`EFWS.Script.loadScripts` 先遍历所有脚本 URL 来确定它们是否都与主页面同域。这样做是因为如果所有外部脚本都要按顺序加载的话需要使用单一技术（译注 4）。如果脚本来自相同域，就选择 `EFWS.Script.loadScriptXhrInjection` 作为脚本加载的函数，如果是不同的域，Firefox 和 Opera 下使用 `EFWS.Script.loadScriptDomElement`，其他情况下都用 `EFWS.Script.loadScriptDocWrite`。（详见第 52 页“多个外部脚本”对这样选择的解释。）

```
EFWS.Script = {
    loadScripts: function(aUrls, onload) {
        // 第一步：检查是否有脚本在不同的域
        var nUrls = aUrls.length;
        var bDifferent = false;
        for ( var i = 0; i < nUrls; i++ ) {
            if ( EFWS.Script.differentDomain(aUrls[i]) ) {
                bDifferent = true;
                break;
            }
        }
```

译注 4：因为两种技术交叉使用无法保持执行顺序，所以需要先根据域的情况决定选用哪一种技术。

```
        // 选择最佳的加载函数
        var loadFunc = EFWS.Script.loadScriptXhrInjection;
        if ( bDifferent ) {
            if ( -1 != navigator.userAgent.indexOf('Firefox') ||
                 -1 != navigator.userAgent.indexOf('Opera') ) {
                loadFunc = EFWS.Script.loadScriptDomElement;
            }
            else {
                loadFunc = EFWS.Script.loadScriptDocWrite;
            }
        }

        // 第二步：加载所有脚本
        for ( var i = 0; i < nUrls; i++ ) {
            loadFunc(aUrls[i], ( i+1 == nUrls ? onload : null ), true);
        }
    },

    differentDomain: function(url) {
        if ( 0 === url.indexOf('http://') || 0 === url.indexOf('https://') ) {
            var mainDomain = document.location.protocol + "://" +
                             document.location.host + "/";
            return ( 0 !== url.indexOf(mainDomain) );
        }

        return false;
    },

    loadScript: function(url, onload) { ... },
    loadScriptDomElement: function(url, onload) { ... },
    loadScriptDocWrite: function(url, onload) { ... },
    queuedScripts: new Array(),
    loadScriptXhrInjection: function(url, onload, bOrder) { ... },
    injectScripts: function() { ... },
    getXHRObject: function() { ... }
};
```

一旦确定了合适的加载函数之后，通过第 2 次遍历脚本 URL 数组来加载每个脚本。尤其要注意 `true` 是作为第 3 个参数传入到脚本加载函数中的。这对于 `EFWS.Script.loadScriptXhrInjection` 作为加载函数的时候很关键，以此保证响应按特定顺序执行。这个参数会被 `EFWS.Script.loadScriptDomElement` 和 `EFWS.Script.loadScriptDocWrite` 忽略，因为这些技术默认保持脚本执行顺序——这也是为什么选择它们的原因。

loadScripts Same Domain 示例使用了 menu.js 和 menutier.js，现在加载脚本的代码只有一行了：

```
<script type="text/javascript">
var aRaceConditions = [['couple-normal.php', 'Normal Script Src'], ...];
var aWorkarounds = [['hardcoded-callback.php', 'Hardcoded Callback'], ...];
var aMultipleScripts = [['managed-xhr.php', 'Managed XHR'], ...];
var aLoadScripts = [['loadscript.php', 'loadScript'], ...];
```

```
var aSubmenus = [["Race Conditions", aRaceConditions], ...];

function init() {
    EFWS.Menu.createTieredMenu('examplesbtn', aSubmenus);
}

EFWS.Script.loadScripts( ["menu.js", "menutier.js"], init);
</script>
```

loadScripts Different Domain 示例使用了部署在 *http://souders.org* 的 menu.js 和 menutier.js，脚本加载代码也只有一行：

```
<script type="text/javascript">
var aRaceConditions = [['couple-normal.php', 'Normal Script Src'], ...];
var aWorkarounds = [['hardcoded-callback.php', 'Hardcoded Callback'], ...];
var aMultipleScripts = [['managed-xhr.php', 'Managed XHR'], ...];
var aLoadScripts = [['loadscript.php', 'loadScript'], ...];
var aSubmenus = [["Race Conditions", aRaceConditions], ...];

function init() {
  EFWS.Menu.createTieredMenu('examplesbtn', aSubmenus);
}

EFWS.Script.loadScripts( ["http://souders.org/efws/menu.js",
                          "http://souders.org/efws/menutier.js"], init);
</script>
```

在这些示例中，`EFWS.Script.loadScripts` 顺利异步加载脚本的同时保持执行顺序。其他资源的异步加载（这里指那张图片）如之前表 5-2 总结的那样依据各浏览器而有所不同。如图 5-6 所示，在 Firefox 2 和 3、Safari 4 及 Chrome 2 中，脚本和图片并行加载。如图 5-7 所示，图片在 Internet Explorer 6 到 8、Opera、Safari3 和 Chrome1 中下载被阻塞，导致需要更长的加载时间。虽然图片异步加载是一个混合的结果，但是脚本在除 Safari3 和 Chrome1 之外的所有浏览器中都能并行加载。

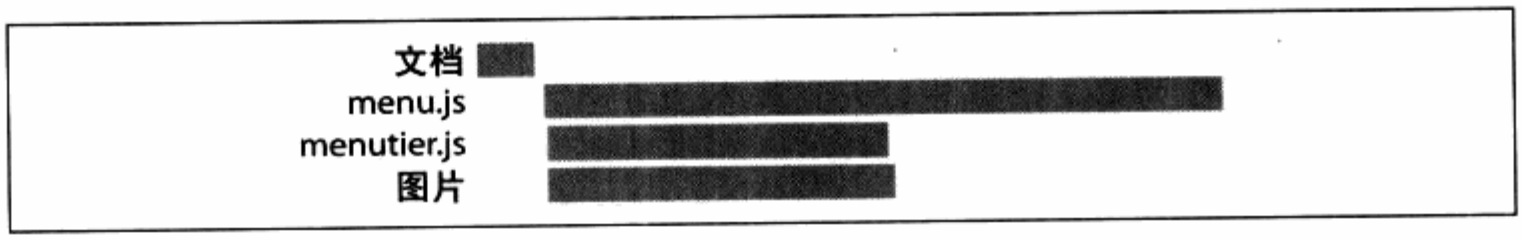


图 5-6：在 Firefox3 中的 loadScripts Different Domain 示例 HTTP 瀑布图

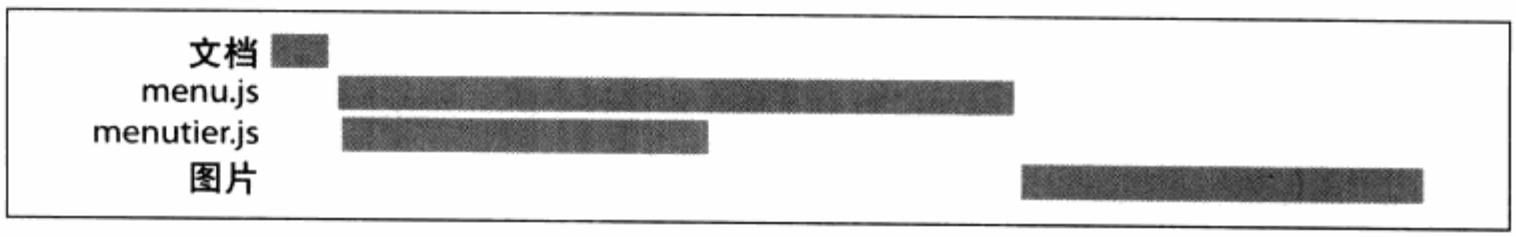


图 5-7：在 Internet Explorer 7 中的 loadScripts Different Domain 示例 HTTP 瀑布图

5.6 现实互联网中的异步加载

Asynchronicity in the Real World

在这一节，我将回顾一些流行的 JavaScript 框架是怎么设计脚本加载的。

5.6.1 Google 分析和 Dojo

Google Analytics and Dojo

我在本章已经提到过 Google 分析。它是 Google 提供的服务，Web 开发者可以利用它来收集网站数据，所有功能封装在 *http://www.google- analytics.com/ga.js* 里面。Google 分析帮助中心建议网站中使用 `document.write` 来添加这个外部脚本（注 2）：

```
<script type="text/javascript">
var gaJsHost = (("https:" == document.location.protocol) ? "https://ssl." :
"http://www.");
document.write(unescape("%3Cscript src='" + gaJsHost + "goog-
leanalytics.com/ga.js' type='text/javascript'%3E%3C/script%3E"));
</script>
<script type="text/javascript">
var pageTracker = _gat._getTracker("UA-xxxxxx-x");
pageTracker._trackPageview();
</script>
```

在本章情境下这是个值得分析的例子。这个外部脚本很适合异步加载，因为它不是用于渲染可视页面的。由于行内脚本依赖外部脚本，我们必须保持它们的执行顺序并把它们整合起来。

Google 分析推荐的 `document.write` Script Tag 有如下一些好处：如果有需要，URL 可以动态修改为 HTTPS，而且外部脚本和行内代码的执行顺序在所有浏览器里都得到了保证。

`document.write` Script Tag 技术的一个缺点是阻塞其他资源下载。`dojox.analytics.Urchin` 模块解决了这个问题，如 Dojo 功能文档第一行所描述的（注 3）：

注 2：*http://www.google.com/support/analytics/bin/answer.py?hl=en&answer=55488*。
注 3：*http://docs.dojocampus.org/dojox/analytics/Urchin*。

这个类用于延迟加载 Google 分析代码。它的前身是 Urchin（译注 5）。原生<script>标签在远程文件同步加载完毕之前会导致页面渲染停顿，而这个模块可以缓解这个情况。

dojox.analytics.Urchin 是 Dojo JavaScript 工具包（*http://dojotoolkit.org*）的一部分。如文档里指出的，Urchin 是 Google 分析模块以前的名字。这解释了该 Dojo 模块取名为 Urchin.js 的原因。这个模块的关键函数是_loadGA、_checkGA 和_gotGA（注 4）：

```
_loadGA: function(){
    // 摘要：加载 ga.js 文件，开始初始化处理函数
    var gaHost = ("https:" == document.location.protocol) ? "https://ssl." :
"http://www.";
    dojo.create('script', {
        src: gaHost + "google-analytics.com/ga.js"
    }, dojo.doc.getElementsByTagName("head")[0]);
    setTimeout(dojo.hitch(this, "_checkGA"), this.loadInterval);
},

_checkGA: function(){
    // 摘要：嗅探 Google 定义的_gat 变量，要么再次检查，要么确认该变量已经准备好则
触发 onLoad 事件
    setTimeout(dojo.hitch(this, !window["_gat"] ? "_checkGA" : "_gotGA"),
this.loadInterval);
},

_gotGA: function(){
    // 摘要：初始化 tracker
    this.tracker = _gat._getTracker(this.acct);
    this.tracker._initData();
    this.GAonLoad.apply(this, arguments);
},
```

_loadGA 函数使用了 Script DOM Element 异步加载技术。它调用 dojo.create 创建了一个 script 元素，依据主页面的协议的不同，可以把 src 设为 *http://www.google-analytics.com/ga.js* 或 *https://ssl.google-analytics.com/ga.js*，最后把 script 元素添加到了文档的 head 元素中。

ga.js 和行内代码的整合则使用了定时器。每次 loadInterval（420 毫秒）执行就会调用 _checkGA 来判断 window["_gat"]（Google 分析里的对象）是否已经定义了。如果是，则调用_gotGA 来实例化 Google 分析。 这种整合方法与第 48 页的“技术 3：定时器”描述的定时器技术类似。

把这种实现方式与 EFWS.Script.loadScript 做比较，我们可以看到两者都使用了 Script DOM Element 方法。使用该方法可以让脚本下载时不阻塞其他资源，而且在所有主流浏览

译注 5：美国东部时间 2005 年 3 月 28 日，Google 收购 Urchin 公司，进军互联网数据分析市场。

注 4：*http://bugs.dojotoolkit.org/browser/dojox/trunk/analytics/Urchin.js.* Copyright (c) 2004–2008 Dojo 基金会版权所有。更多细节请看 *http://dojotoolkit.org/license*。

器里都可行。但它们之间也存在不同点。EFWS.Script.loadScript 没有使用定时器技术，使用的是 Script Onload 技术。定时器技术存在一些缺陷：

- 如果脚本加载失败，计时器会无限运行下去。
- 该方式需要更大的维护成本。如果 ga.js 有改动，不再定义_gat，那么_checkGA 也要被迫更新。而 Script Onload 方式不依赖 ga.js 里的任何标识符。
- 在 ga.js 加载完成和_gotGA 被调用之间有可能会存在多达 420 毫秒的时间差，这很可能导致用户在代码运行之前已经离开页面。Script Onload 方式在外部脚本加载完成后就立即执行行内代码。

正是由于这些原因，Script Onload 成为了 EFWS.Script.loadScript 中使用的整合技术。

5.6.2 YUI Loader

Google 分析是一个适用于分析如何异步加载单个脚本同时整合行内代码的例子。YUI Loader 是我挑选来考察如何加载多个脚本的例子。如下所述这个工具包（注 5）是 Yahoo! UI Library（*http://developer.yahoo.com/yui/*）的一部分。

> YUI Loader 工具包是一个客户端 JavaScript 组件，允许你通过脚本加载特定的 YUI 组件和它们所依赖的组件到页面中。YUI Loader 能作为一个整体解决方案来加载所有你需要的 YUI 组件，也能用于在已经存在一些 YUI 组件的页面中再加载更多个组件。

YUI Loader 的设计目的是提供随时随地加载和依赖关系计算的功能。它通过只加载必需的模块并合并为单个 HTTP 请求来提高网页性能，这归功于 combo-handling（注 6）。我使用 YUI Loader 修改了那个使用 menu.js 和 menutier.js 的示例，来查看脚本是否是并行加载的。

YUI Loader

http://stevesouders.com/efws/yuiloader.php

该示例从加载 YUI Loader 本身（*http://yui.yahooapis.com/2.6.0/build/yuiloader/yuiloader-min.js*）开始。为了加载 menu.js 和 menutier.js，我们创建了 YUILoader 实例并使用了 addModule 方法，指定 init 函数在这些脚本顺利加载完毕后调用。这一切通过调用 insert 方法开始：

```
<script type="text/javascript"
src="http://yui.yahooapis.com/2.6.0/build/yuiloader/yuiloader-min.js">
</script>
```

注 5：*http://developer.yahoo.com/yui/yuiloader/*。
注 6：*http://yuiblog.com/blog/2008/10/17/loading-yui/*。

```
<script type="text/javascript">
var aRaceConditions = [['couple-normal.php', 'Normal Script Src'], ...];
var aWorkarounds = [['hardcoded-callback.php', 'Hardcoded Callback'], ...];
var aMultipleScripts = [['managed-xhr.php', 'Managed XHR'], ...];
var aLoadScripts = [['loadscript.php', 'loadScript'], ...];
var aSubmenus = [["Race Conditions", aRaceConditions], ...];

function init() {
   EFWS.Menu.createTieredMenu('examplesbtn', aSubmenus);
}

var loader = new YAHOO.util.YUILoader();
loader.addModule({ name: "menu", type: "js", fullpath: "menu.js"});
loader.addModule({ name: "menutier", type: "js", fullpath: "menutier.js"});
loader.require("menu");
loader.require("menutier");
loader.onSuccess = init;
loader.insert();
</script>
```

我们可以在 *http://yui.yahooapis.com/2.6.0/build/yuiloader/yuiloader.js*（这是带注释的未压缩的代码）中看到 YUI Loader 是如何执行的。脚本通过_node 函数插入，类似于 Script DOM Element 方法。_track 函数使用了 Script Onload 整合技术。YUI（译注 6）实现了对浏览器边界情况的处理，相当完美。

最重要的发现是 YUI Loader 没有并行加载脚本，虽然使用了 Script DOM Element 技术。很显然 YUI Loader 是按顺序加载脚本的，在请求下一个之前一直等待，直到前一个脚本返回，这可以在示例的 HTTP 瀑布图中看出（见图 5-8），最后两个请求是那两个脚本。将这张图与图 5-6 和 5-7 相比，我们看到 EFWS.Script.loadScripts 可以并行加载脚本，使得页面加载更快。

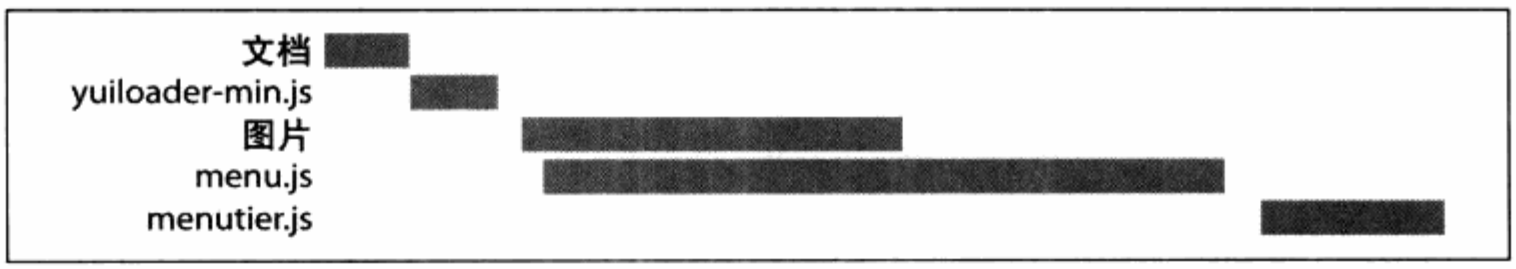


图 5-8：YUI Loader HTTP 瀑布图

YUI Loader 的按顺序加载行为导致它需要比 EFWS.Script.loadScripts 更长的加载时间，Safari3 和 Chrome1 除外。不过 YUI Loader 可以随时随地加载脚本，即使是在文档加载完毕之后，而在一些浏览器中使用 EFWS.Script.loadScripts 只能在页面初次加载过程中使用。

译注 6：YUI 支持所有 A-Grade 浏览器，详见 *http://developer.yahoo.com/yui/articles/gbs/index.html*。

对于那些有外部脚本的页面，使用 `EFWS.Script.loadScripts` 异步加载脚本可以提高性能，而且会随着脚本数的增加变得更为显著。如果不使用整合技术，简单的方式是如《高性能网站建设指南》规则 1 所推荐的那样，把脚本合并即可，但并不总是可以这么做。美国的 10 大网站中，外部脚本的平均数是 6.5（见表 11-1），所以并行加载这些脚本、保持执行顺序、整合行内代码，对提高当今各大网站的速度都相当关键。

第 6 章

布置行内脚本

Positioning Inline Scripts

前 3 章的重心集中在**外部**脚本的影响上，而本章的重点是**行内**脚本（直接包含在 HTML 文档中的 JavaScript）。虽然行内脚本不会产生额外的 HTTP 请求，但会阻塞页面上资源的并行下载，还会阻塞逐步渲染（progressive render）。本章讲解了在不同时间和场合使用行内 JavaScript 对页面性能的影响。

6.1 行内脚本阻塞并行下载

Inline Scripts Block

第 5 章描述了外部脚本阻塞并行下载和渲染的方式。由于相同的原因（保持执行顺序及对 `document.write` 的依赖关系），行内脚本也存在相同的行为并不出人意料。Inline Scripts Block 示例演示了这种行为。

Inline Scripts Block
http://stevesouders.com/cuzillion/?ex=10100&title=Inline+Scripts+Block

图 6-1 显示了该页面发出的 HTTP 请求。除了 HTML 文档外，还有两个图片请求，每张图片的下载时间设置为 1 秒。这两张图之间有一段行内脚本，图 6-1 用一条线来表示它。它虽然没有产生 HTTP 请求，影响却显而易见。

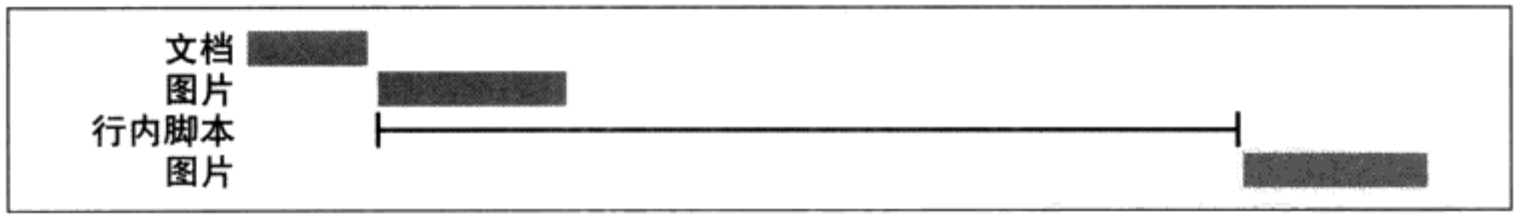


图 6-1：行内脚本阻塞并行下载（6 秒）

行内脚本设置的执行时间为 5 秒，这导致了图 6-1 中两张图片请求之间有 4 秒空白。行内脚本和第 1 张图片请求是并行执行的。1 秒后，图片响应到达，而行内脚本继续执行余下的 4 秒。当行内脚本执行时，将阻塞其他所有资源的下载。直到行内脚本执行结束时（在页面上执行了 5 秒），第 2 张图片才开始下载，最终导致 6 秒的总加载时间。

除了阻塞并行下载，行内脚本还阻塞渲染。在 Inline Scripts Block 示例页面加载之后，至少 5 秒内页面上没有任何内容显示出来。最佳观测方法是先把浏览器转到另一个页面或空白页（`about:blank`），然后访问 Inline Scripts Block 示例的 URL。5 秒钟内没有任何内容渲染出来，这有些令人惊讶，因为 HTML 文档中，在行内脚本之前有一些普通文本（“Cuzillion”头、“Examples” 和“Help”链接等。），但直到行内脚本执行完成，浏览器才开始渲染这些文本内容。

如果你的站点中使用了行内脚本，那么理解它们阻塞下载和渲染的方式，以及如何尽可能地避免这种行为显得尤为重要。下面提供了几个有效的解决方案：

- 把行内脚本移至底部。
- 使用异步回调启动 JavaScript 的执行。
- 使用 `script` 的 `defer` 属性。

下面几节将分别讲解这几种技术。

6.1.1 把行内脚本移至底部

Move Inline Scripts to the Bottom

把行内脚本移至页面上所有资源的后面来实现并行下载和逐步渲染（注 1）。下面的例子证明了把行内脚本移至底部的好处。

Move Inline Scripts to the Bottom

http://stevesouders.com/efws/inline-scripts-bottom.php

图 6-2 显示了两个并行下载的图片请求。5 秒的行内脚本与图片并行执行，使得它们在页面中总的加载时间是 5 秒，比基准页面（上例）快了 1 秒。虽然该技术避免了阻塞下载，但它依旧阻塞渲染。如果行内脚本执行时间不是很长（少于 300 毫秒），那么这种技术可以作为一个页面提速的简单方法。执行时间长一些的行内脚本应该使用下面的两种技术。

注 1：这和《高性能网站建设指南》中的建议“规则 6：把脚本放在底部”类似。

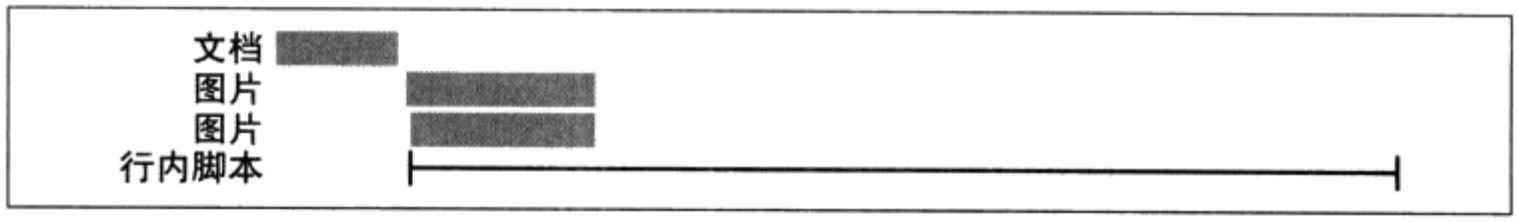


图 6-2：行内脚本阻塞并行下载（5 秒）

6.1.2 异步启动执行脚本

Initiate Execution Asynchronously

你可以让浏览器异步执行行内脚本，使其有可能实现并行下载和逐步渲染。简单的异步调用技术就是使用 `setTimeout`，如下面的例子所示：

```
function longCode() {
    var tStart = Number(new Date());
    while( (tStart + 5000) > Number(new Date()) ) {};
}

setTimeout(longCode, 0);
```

`longCode` 是一个执行时间为 5 秒的 JavaScript 函数。首次尝试使用 `setTimeout` 时，我们可以设置延迟值为 0 毫秒，如下面的例子所示：

Inline Scripts via setTimeout (0 milliseconds)
http://stevesouders.com/efws/inline-scripts-settimeout.php?d=0

该结果类似于把行内脚本移至底部的技术：图片并行下载，页面花了 5 秒加载完成。但与前面的技巧不同的是，使用 `setTimeout` 有额外的好处，那就是在 Internet Explorer 中实现了逐步渲染。在行内脚本开始执行之前，Internet Explorer 有足够的时间渲染页面顶部的文本（“Cuzillion”标题、“Examples”和“Help”链接等）。

虽然延迟值为 0 毫秒的 `setTimeout` 在 Internet Explorer 中实现了逐步渲染，但在 Firefox 中的渲染仍然是被阻塞的。我们需要增加到 250 毫秒来实现 Firefox 中的逐步渲染，下一个例子做了这个演示。

Inline Scripts via setTimeout (250 milliseconds)
http://stevesouders.com/efws/inline-scripts-settimeout.php?d=250

神奇数字 250 源于 `nglayout.initialpaint.delay` 的默认值，这是“页面显示之前的等待毫秒数”（注 2）。如果 `longCode` 在 250 毫秒之前就开始执行，所有渲染在它执行完毕之

注 2：*http://kb.mozillazine.org/Nglayout.initialpaint.delay*。

前都会被阻塞。但是，如果我们等待 250 毫秒之后再调用 longCode，Firefox 就能渲染页面顶部的文本。

在这两种情况下（Internet Explorer 中 0 毫秒和 Firefox 中 250 毫秒）都只有文本被快速渲染。虽然图片在 1 秒之后返回，但直到 5 秒钟之后 longCode 执行完毕时才在页面中显示。渲染事件在 1 秒之后进入队列，但浏览器在 longCode 执行的同时不能响应这个事件。浏览器在执行 JavaScript 时是单线程的，所有渲染事件都会被阻塞（注 3）。在下一个示例中，我们通过增加 setTimeout 的毫秒数来解决它，该数值稍稍大于图片的下载时间 1 秒，比如 1500 毫秒。

Inline Scripts via setTimeout (1,500 milliseconds)

http://stevesouders.com/efws/inline-scripts-settimeout.php?d=1500

现在图片在下载之后立即显示了。因为它只花了 1 秒来渲染页面中所有东西，onload 事件在 1 秒而非 5 秒之后触发。使用延迟值为 1 500 毫秒的弊端是 longCode 直到 6 500 毫秒后才能执行完毕（1500 毫秒延迟时间加上 5000 毫秒执行时间）。如果我们想异步执行 longCode 而不阻塞浏览器渲染，更好的做法是使用 onload 事件来触发代码运行：

```
function longCode() {
    var tStart = Number(new Date());
    while( (tStart + 5000) > Number(new Date()) ) {};
}

window.onload = longCode;
```

如下例所示，使用 onload 事件可以让文本和图片在一旦可用时立即被渲染，并在不阻塞下载和渲染的前提下尽可能早地执行行内脚本。

Inline Scripts via onload

http://stevesouders.com/efws/inline-scripts-onload.php

如果行内脚本执行时间很短，那么使用延迟值为 0 毫秒的 setTimeout 是一个兼顾快速渲染和 JavaScript 快速执行的好方案。如果脚本执行时间很长，更好的选择是使用 onload。最佳的方案是每 300 豪秒左右挂起或使用 setTimeout，但这需要大规模的重新设计和重构代码。请看第 103 页的“使用定时器挂起”，它对该技术有深入的探讨。

注 3：第 2 章有更多关于 JavaScript 影响浏览器响应的讨论。

6.1.3 使用 script 的 defer 属性

Use Script Defer

Script 的 defer 属性只有 Internet Explorer 和 Firefox3.1+支持。如第 4 章所述，人们通常把它用于下载外部脚本。其实 defer 属性也适用于行内脚本，它允许浏览器继续解析和渲染页面的同时延迟执行行内脚本。我们可以使用 Cuzillion 来创建一个使用 defer 属性的行内脚本的示例。

Inline Scripts and Defer

http://stevesouders.com/cuzillion/?ex=10101&title=Inline+Scripts+and+Defer

在支持 defer 属性的浏览器中，该属性将允许两张图片并行下载，页面总加载时间是 5 秒（比 6 秒的基准页面快）。然而直到 5 秒后脚本执行完成时页面才开始渲染。defer 是较简单的实现并行下载解决方案，但它只支持 Internet Explorer 和 Firefox3.1+中的行内脚本，而且仍然阻塞逐步渲染，所以更好的方法是使用 setTimeout。

6.2 保持 CSS 和 JavaScript 的执行顺序

Preserving CSS and JavaScript Order

加载外部脚本的典型方法是使用 script 的 src 属性：

```
<script src="A.js" type="text/javascript"></script>
<script src="B.js" type="text/javascript"></script>
```

如第 4 章所描述，通过这种方法加载脚本会阻塞并行下载。其主要原因是浏览器为了确保正确的执行顺序，每次只下载一个脚本，因为在 A.js 之前执行 B.js 可能会因代码依赖关系而出现意外的行为或未定义变量错误。

保持 JavaScript 的顺序至关重要，对 CSS 来说也是如此。鉴于样式的级联特性，按不同的顺序加载它们可能会产生意想不到的结果。为了提供一致的行为，浏览器要保证按照指定的顺序应用 CSS。Stylesheets in Order 示例证实了不管 HTTP 响应和接受的顺序如何，它们都是按照指定顺序应用的。

Stylesheets in Order

http://stevesouders.com/efws/stylesheets-order.php

这个例子中有两个样式表，它们都对同一选择符定义了一条规则（译注 1）。如图 6-3 所示，程序控制第 1 个样式表需要较长的时间才能完成下载。第 1 个样式表定义了灰色的背景，而第 2 个样式表定义了桔色的背景，最终生效的是桔色，这意味着第 2 个样式表虽然先完成下载，却被最后应用。这表明浏览器是按照样式表在页面中列出的顺序应用它们的，而与下载的顺序无关。

译注 1：定义的是 body 元素的样式规则。

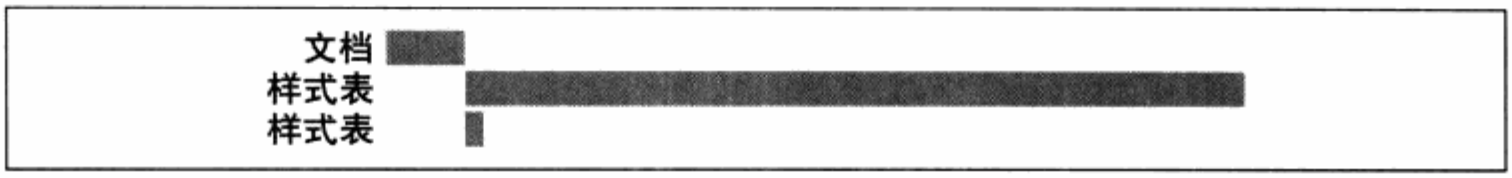


图 6-3：样式表在 Internet Explorer 中按指定顺序应用

CSS 的应用规则同时适用于样式表和行内样式。在 CSS in Order 示例中，在和图 6-3 中下载时间一样长的样式表（定义灰色背景）后面是一段行内样式（定义桔色背景）。再一次表明，浏览器会等待下载时间长的样式表下载完成以保证 CSS 是按照页面指定的顺序应用的。

CSS in Order

http://stevesouders.com/efws/css-order.php

理解浏览器会保证按照页面指定的顺序应用 CSS 是很有用的。但这和行内脚本之间有什么关系呢？下一节将把它们联系起来。

6.3 风险：把行内脚本放置在样式表之后

Danger: Stylesheet Followed by Inline Script

在上一节，我们证实了浏览器按照 CSS 在 HTML 文档中出现的顺序来应用它们（样式表和行内样式都是如此）。在本章的前面，我们验证了行内脚本会阻塞浏览器的其他行为（下载和渲染）。这些现象在 Web 开发社区中都是众所周知的，鲜为人知的是浏览器也按顺序应用 CSS 和 JavaScript，而当把行内脚本放置在样式表之后时，该行为会明显地延迟资源的下载。这个顺序导致只有当样式表下载完成并且行内脚本执行完毕时，后续资源才能开始下载。下面几节将解释这个问题发生的原因。

6.3.1 大部分下载都不阻塞行内脚本

Inline Scripts Aren't Bolcked by Most Downloads

在图片和 iframe 下载的同时行内脚本能执行，如下面例子所示：

Inline Scripts After Image and Iframe

http://stevesouders.com/cuzillion/?ex=10102&title=Inline+Scripts+After

图 6-4 显示了 Inline Scripts After Image and Iframe 示例的 HTTP 瀑布图。它显示了 3 项资源：一张图片、一个 iframe 和另一张图片，分别需要 2 秒的下载时间，在它们之间分别有一段执行时间为 1 秒的行内脚本。下面的列表说明了页面加载时间线中的关键事件。

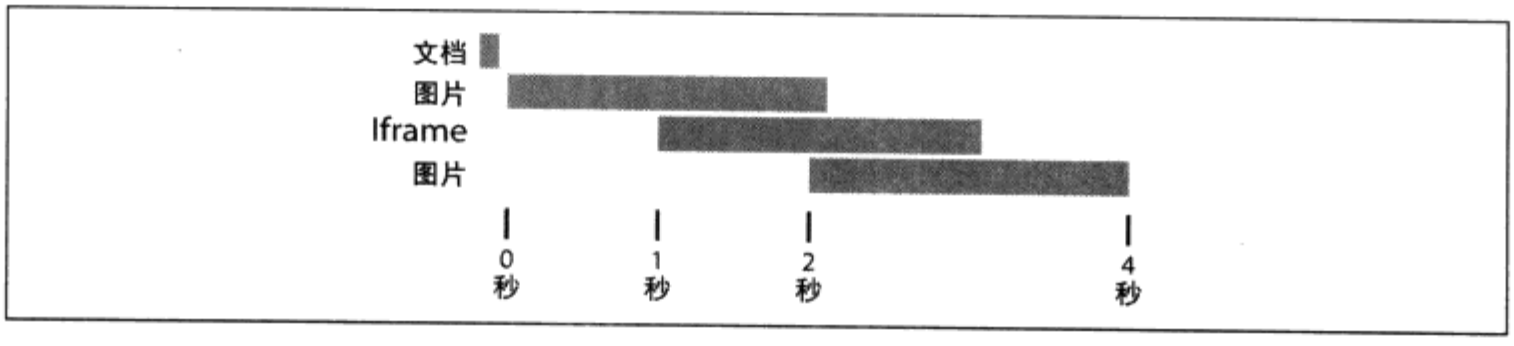


图 6-4：行内脚本在一张图片和一个 iframe 后面（4 秒）

0 秒

第 1 张图片开始下载，第 1 段行内脚本开始和图片下载并行执行。

1 秒

第 1 段行内脚本执行完成，这为 iframe 开始下载和第 2 段行内脚本开始执行创造了条件。第 2 段行内脚本在 iframe 下载的同时执行。

2 秒

第 2 段行内脚本执行完成，浏览器允许最后那张图片开始下载。

4 秒

最后那张图片完成下载。

因为行内脚本在图片和 iframe 下载的同时执行，整个页面加载仅在 4 秒内就完成了。可是它们与样式表的相互作用会阻塞并行下载，下一节将解释这个问题。

6.3.2 样式表阻塞行内脚本

Inline Scripts Are Blocked by Stylesheets

样式表和行内脚本之间的相互影响明显不同于其他资源，这是由于浏览器需保持 CSS 和 JavaScript 的解析顺序所致，如下面的例子所示：

Inline Scripts After Stylesheet

http://stevesouders.com/cuzillion/?ex=10103&title=Inline+Scripts+after

Inline Scripts After Stylesheet 示例和前面的示例相似，但是第 1 张图片和第 1 个 iframe 替换成了样式表。与之前一样，每个资源下载都需要 2 秒，每段行内脚本执行需要 1 秒。图 6-5 显示了 HTTP 瀑布图，全部的加载时间是 8 秒，而之前示例中是 4 秒！页面加载的时间线说明了该页面花费两倍加载时间的原因。

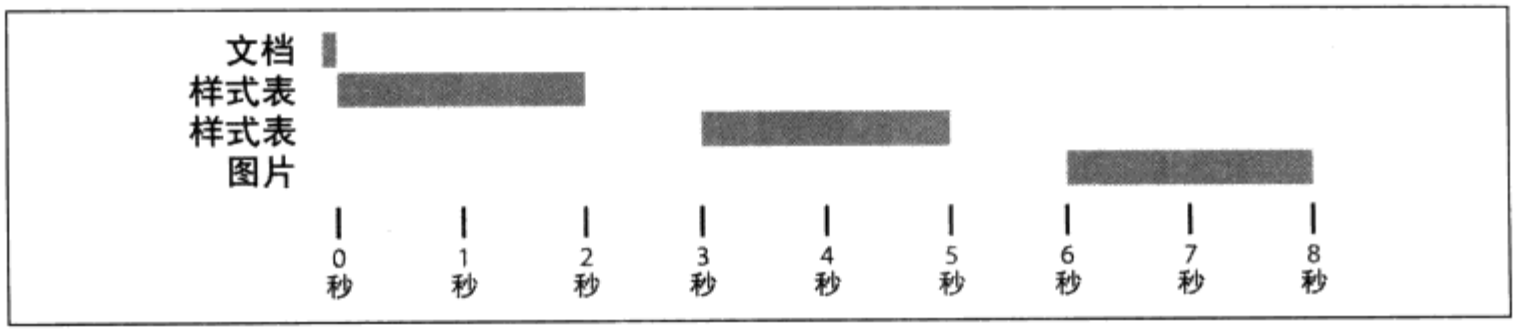


图 6-5：行内脚本在样式表之后（8 秒）

0 秒

第 1 个样式表开始下载。第 1 段行内脚本被阻塞，直到样式表下载解析完毕才开始执行。

2 秒

第 1 个样式表下载完成。第 1 段行内脚本开始执行。

3 秒

第 1 段行内脚本执行完成。第 2 个样式表开始下载。

5 秒

第 2 个样式表下载完成。第 2 段行内脚本开始执行。

6 秒

第 2 段行内脚本执行完成。图片开始下载。

8 秒

图片下载完成。

浏览器按顺序处理 CSS 和 JavaScript 的方式使得这个示例花费的时间是前一个示例的两倍。这个示例表明了当行内脚本在样式表后面的时候，浏览器要在样式表完全下载之后才开始执行行内脚本。这是为什么呢？因为行内脚本可能含有依赖于样式表中样式的代码。我曾看到过，也写过类似的脚本。HTML5 中增加的 `getElementsByClassName` 是 JavaScript 中实际存在此依赖的最好说明（注 4）。浏览器按顺序下载样式表和执行行内脚本是为了保证一致的结果。

这个示例也证实了行内脚本阻塞紧随其后的其他资源的下载。虽然资源通常都可以与样式表一起并行下载，但这两个约束共同作用导致了本章的关键结论：**在样式表后面的行内脚本会阻塞所有后续资源的下载。**

注 4：*http://dev.w3.org/html5/spec /Overview.html#dom-getelementsbyclassname*。

6.3.3 问题确曾发生

This Does Happen

前面的例子说明了当行内脚本在样式表之后时会产生阻塞，但至少在我看来它们像人为设计而成的。幸运的是（或者说不幸的是），由于这种行为还没有得到广泛的研究和宣传，因此很容易就可以在现实世界中发现这个问题的实例。在美国排名前 10 的网站里，有 4 个（eBay、MSN、MySpace 和 Wikipedia）就出现了行内脚本置于样式表之后的现象。它引起样式表后面资源的下载时间多于必需的下载时间，从而导致页面变慢。

图 6-6 显示了 eBay 的 HTTP 瀑布图的一部分，行内脚本在样式表之后引起两个脚本下载被阻塞，这种影响直到样式表完成下载为止。箭头显示了如果没有这个问题影响下载可以开始的位置。

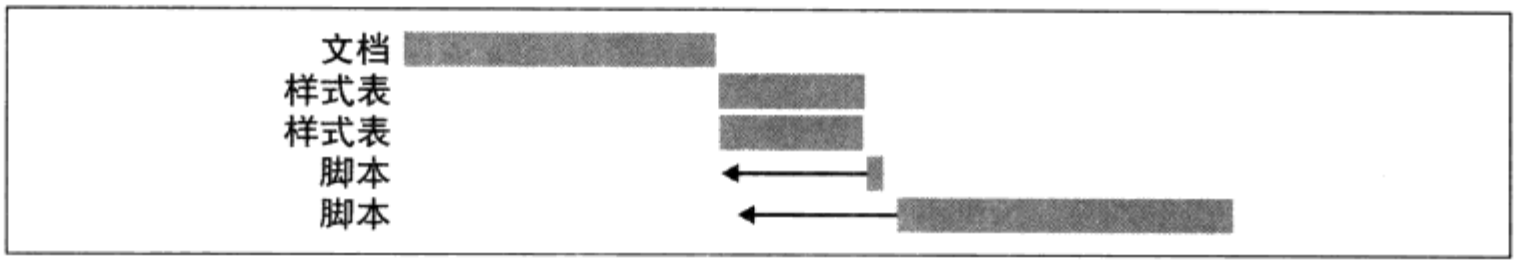


图 6-6：eBay 的行内脚本在样式表后面

与图 6-7 所示的类似，MSN 的并行下载比预期要少，因为样式表阻塞了图片。在图 6-8 中，我们看到 MySpace 的一个脚本下载延迟了，因为 5 个样式表后面的一段行内脚本阻塞了它。

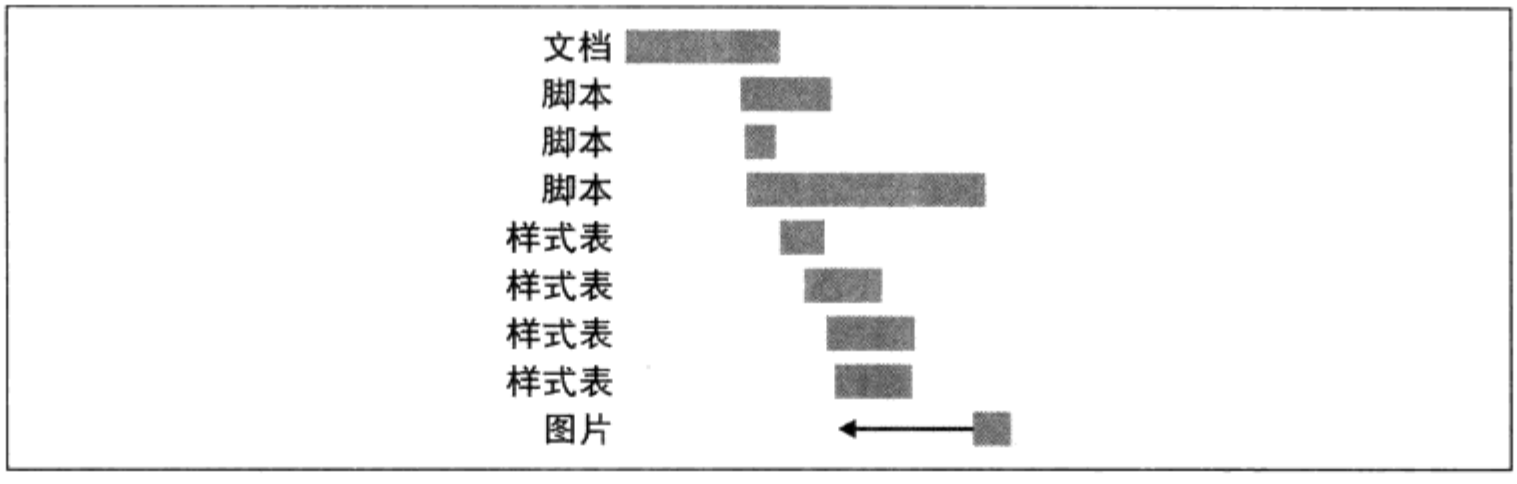


图 6-7：MSN 的行内脚本在样式表后面

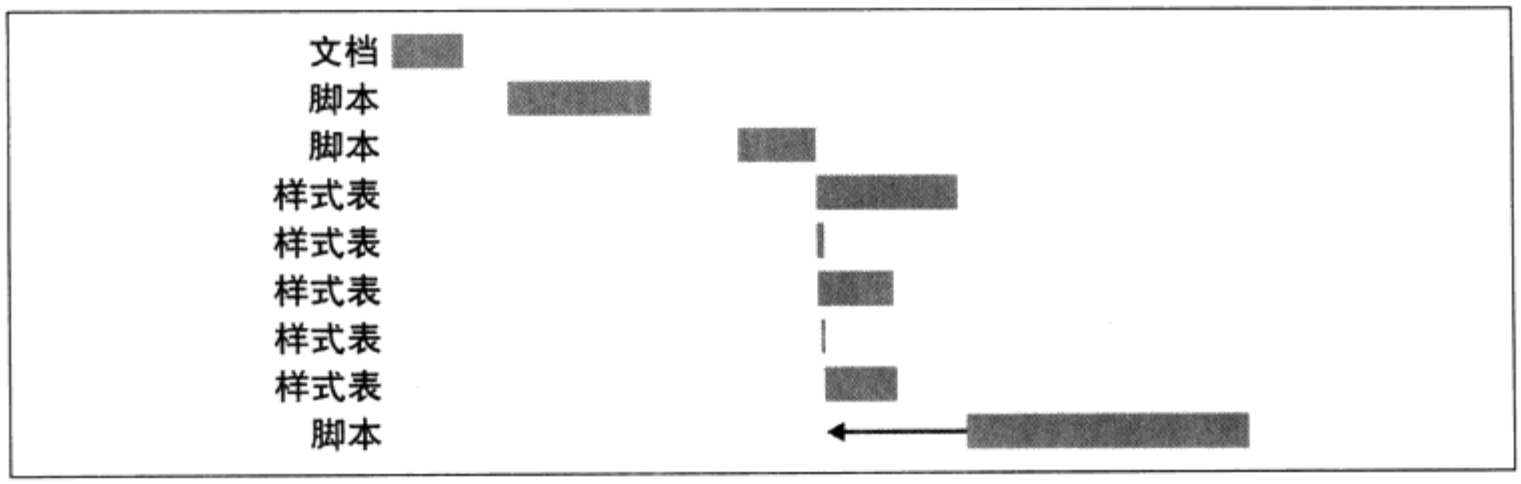


图 6-8：MySpace 的行内脚本在样式表后面

图 6-9 中显示了 Wikipedia 的情况，在 HTTP 瀑布图尾部的行内脚本由于在样式表后面而被阻塞了下载。顺便说一下，这张 HTTP 瀑布图是在 Internet Explorer 7 中生成的，Internet Explorer 7 支持每个域名两个连接，但这张图里显示了 4 个并发连接。那是因为 Wikipedia 把它的网络通信降级到 HTTP/1.0，这让单个域名的连接数增加到了 4 个。（见第 165 页的“降级到 HTTP/1.0”。）然而，因为其行内脚本在样式表之后，增加并行下载的方法并没有发挥出优势。

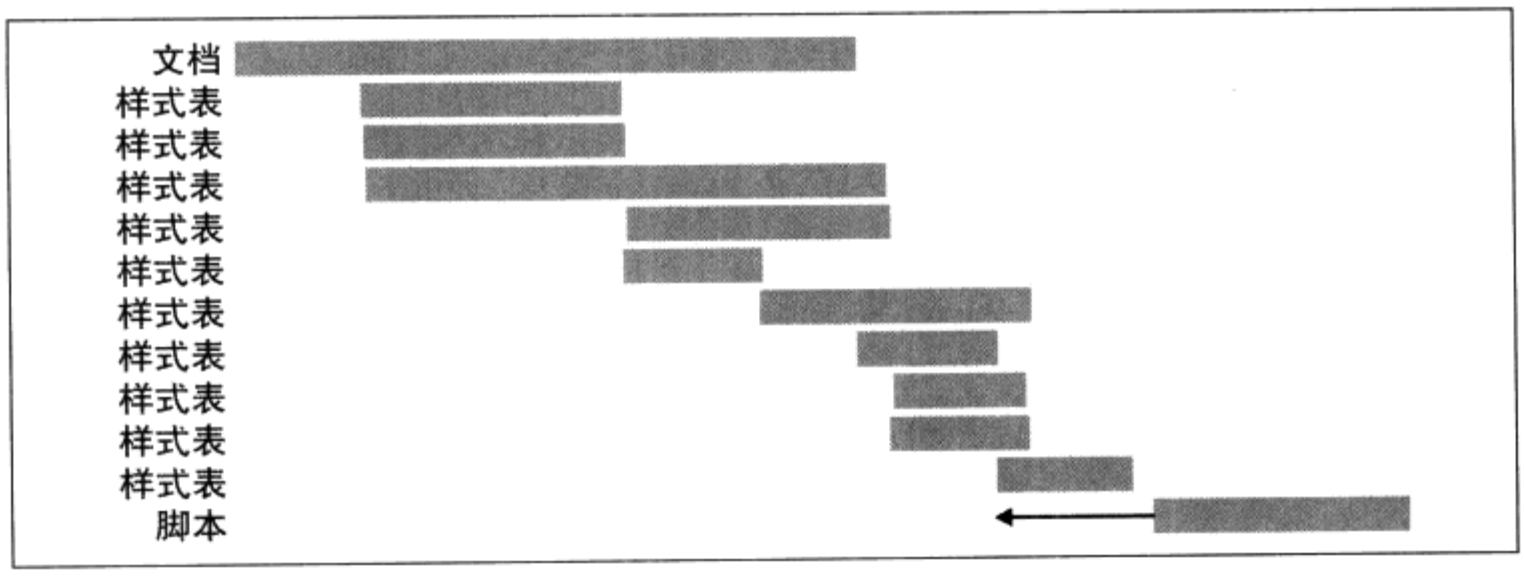


图 6-9：Wikipedia 的行内脚本在样式表后面

这些问题都容易解决，**方案是调整行内脚本的位置，使其不出现在样式表和任何其他资源之间**。行内脚本应该放在样式表之前或其他资源之后，如果其他资源是脚本，行内脚本和外部脚本之间可能会有代码依赖。出于这个原因，我通常建议把行内脚本放在样式表之前，即可避免所有的代码依赖问题。如果你确认没有代码依赖，那么把行内脚本移到可见资源之后可以让它们更快地加载，从而获得更好的逐步渲染效果。

第 7 章

编写高效的 JavaScript

Writing Efficient JavaScript

Nicholas C. Zakas

当今的 Web 应用程序都由大量的 JavaScript 代码驱动。早期的网站只是使用 JavaScript 完成简单的任务，而如今，许多网站和项目的整个用户界面都是用 JavaScript 来实现的。这样做的结果就是用户在界面上的每次操作，都要执行成千上万行的 JavaScript 代码。因此，关注性能，不仅要关注页面的加载时间，也要关注在页面上操作时的响应速度。要实现快速而友好的用户界面，最好的办法是编写的 JavaScript 代码能在所有浏览器中最高效地执行（注 1）。

本章介绍了一些隐藏在 JavaScript 中的性能问题，以及如何解决这些问题，有些只要修改少量的代码结构，而另一些则可能需要重构算法。最重要的是要记住：在性能优化上没有银弹（译注 1），也没有一种方法能适用于 100%的情况。只有结合各种技术，才能实现最大程度的性能优化。

7.1 管理作用域

Managing Scope

当执行 JavaScript 代码时，JavaScript 引擎会创建一个执行上下文(Execution Context)。执行上下文（有时也被称为作用域）设定了代码执行时所处的环境。JavaScript 引擎会在页面加载后创建一个全局的执行上下文，然后每执行一个函数时都会创建一个对应的执行上下文，最终建立一个执行上下文的堆栈，当前起作用的执行上下文在堆栈的最顶部。

每个执行上下文都有一个与之关联的**作用域链**，用于解析标识符。作用域链包含一个或多个变量对象，这些对象定义了执行上下文作用域内的标识符。全局执行上下文的作用域链

注 1: 本章的所有研究都是基于 Firefox 3.0 和 3.1 beta 2、Google Chrome 1.0、Internet Explorer 7 和 8 beta2、Safari 3.0-3.2 以及 Opera 9.62 上的实验。如果没有说明具体的版本，表示包含该浏览器所有参加实验的版本。

译注 1: 在古代的狼人传说中，用银质子弹能制服那些举止无常的怪兽。“没有银弹”在这里可理解为没有绝对有效的方式。在 Fred Brooks 著名的《人月神话》、《没有银弹》中出现这种说法，这里做了引用。

中只有一个变量对象，它定义了 JavaScript 中所有可用的全局变量和函数。当函数被创建（不是执行）时，JavaScript 引擎会把创建时执行上下文的作用域链赋给函数的内部属性[[Scope]]（内部属性不能通过 JavaScript 来存取，所以无法直接访问此属性）。然后，当函数被执行时，JavaScript 引擎会创建一个活动对象（Activetion Object），并在初始化时给 this、arguments、命名参数和该函数的所有局部变量赋值。活动对象会出现在执行上下文作用域链的顶端，紧接其后的是函数[[Scope]]属性中的对象。

在执行代码时，JavaScript 引擎通过搜索执行上下文的作用域链来解析诸如变量和函数名这样的标识符。解析标识符的过程从作用域链的顶部开始，按照自上而下的顺序进行。看下面的代码：

```
function add(num1, num2){
    return num1 + num2;
}

var result = add(5, 10);
```

当这段代码执行时，add 函数拥有一个仅包含全局变量对象的[[Scope]]属性。在执行 add 函数时，JavaScript 引擎会创建一个新的执行上下文和一个包含 this、arguments、num1 和 num2 的活动对象，并把活动对象添加到作用域链中。图 7-1 说明了 add 函数执行时隐藏在它后面的对象之间的关系。

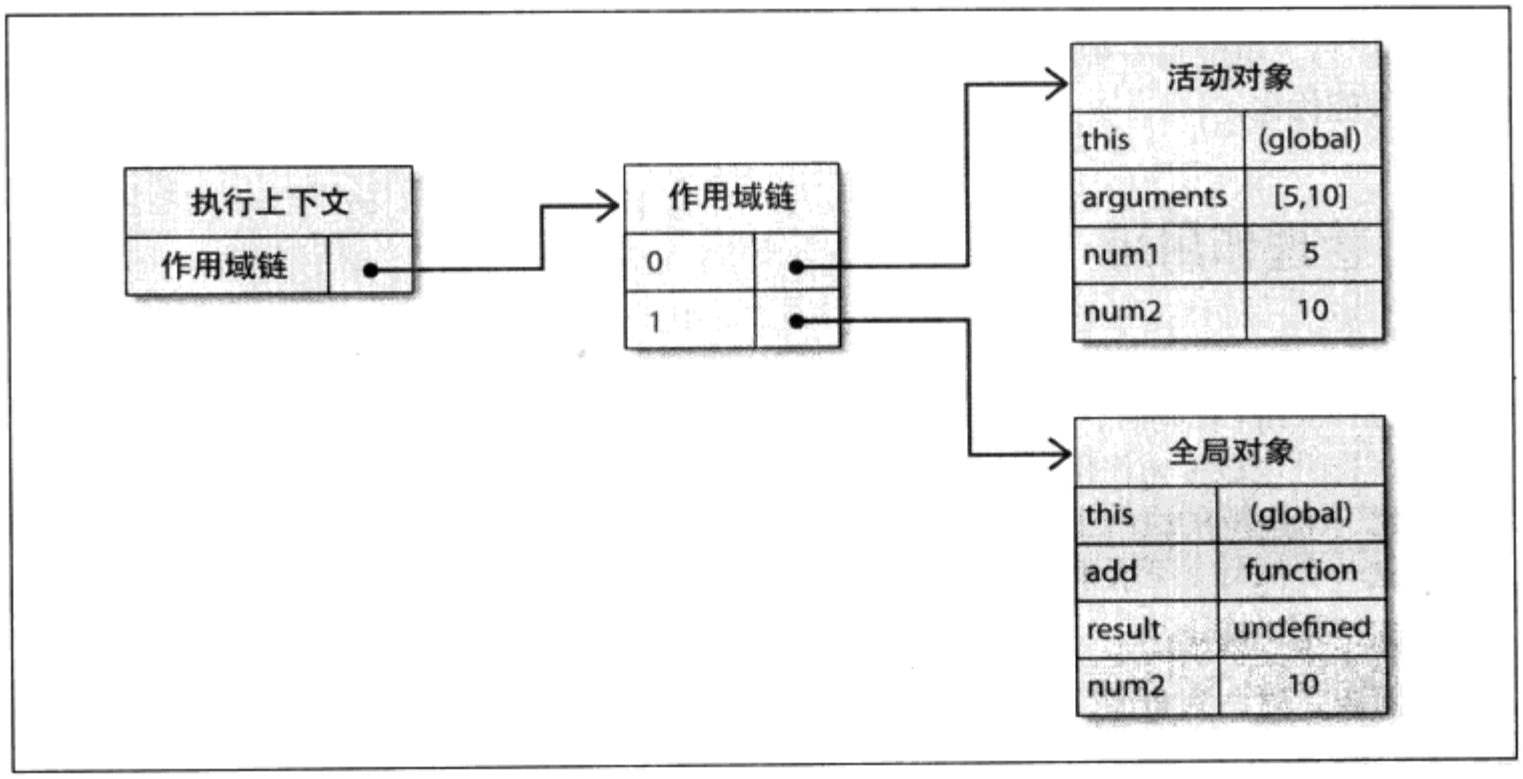


图 7-1：执行上下文和作用域链的关系

当执行 add 函数时，JavaScript 引擎需要解析函数里的 num1 和 num2 标识符，解析的过程是检查作用域链中的每个对象，直到找到指定的标识符。查找从作用域链中的第一个对象开始，这个对象就是包含该函数局部变量的活动对象。如果在该对象中没有找到标识符，就会继续在作用域链中的下一个对象里查找标识符。一旦找到标识符，查找就结束。就本例而言，因为 num1 和 num2 标识符存在于当前的活动对象中，所以不需要到全局对象中查找。

理解 JavaScript 中如何管理作用域和作用域链很重要，因为在作用域链中要查找的对象个数直接影响标识符解析的性能。标识符在作用域链中的位置越深，查找和访问它所需的时间就越长；如果作用域管理不当，就会给脚本的执行时间带来负面影响。

7.1.1 使用局部变量
Use Local Variables

到目前为止，局部变量是 JavaScript 中读写最快的标识符。因为它们存在于执行函数的活动对象中，解析标识符只需要查找作用域链中的单个对象。读取变量值的总耗时随着查找作用域链的逐层深入而不断增长，所以标识符越深存取速度越慢。这种现象几乎在所有浏览器上都存在，只有基于 V8 JavaScript 引擎的 Google Chrome 和基于 Nitro JavaScript 引擎的 Safari 4+例外，它们的存取速度超快，标识符深度的影响微乎其微了。

为了确定标识符深度对性能的实际影响，我运行了一个包括 200 000 个变量运算的实验，对不同深度的标识符反复存取。这个实验页面的 URL 是 *http://www.nczonline.net/experiments/javascript/performance/identifier-depth/*。

图 7-2 说明了基于不同的作用域链深度读取变量总耗时的变化，图 7-3 说明了基于不同的作用域链深度写入变量总耗时的变化（深度值是 1 的位置代表局部标识符）。

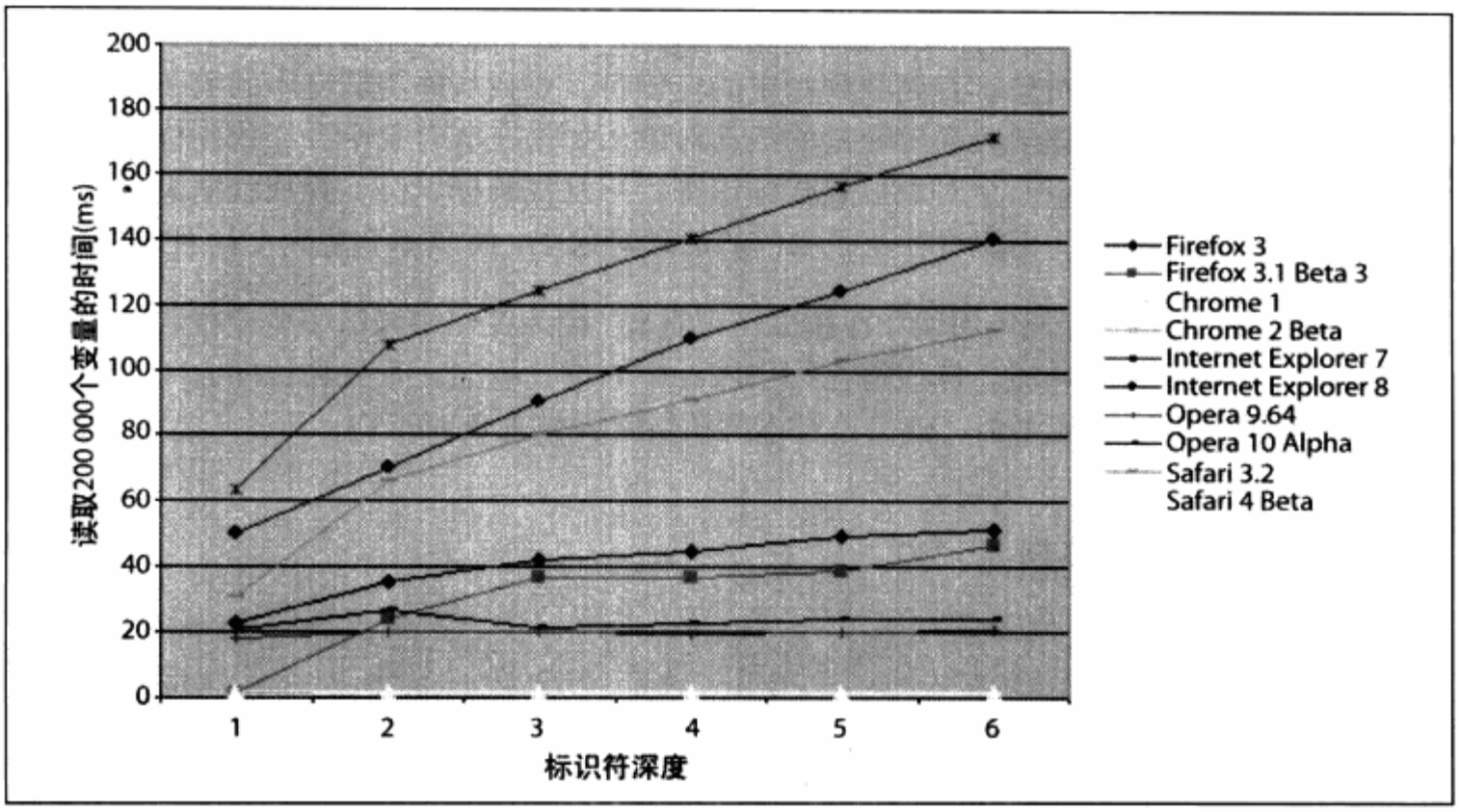


图 7-2：变量读取时间随标识符深度的变化

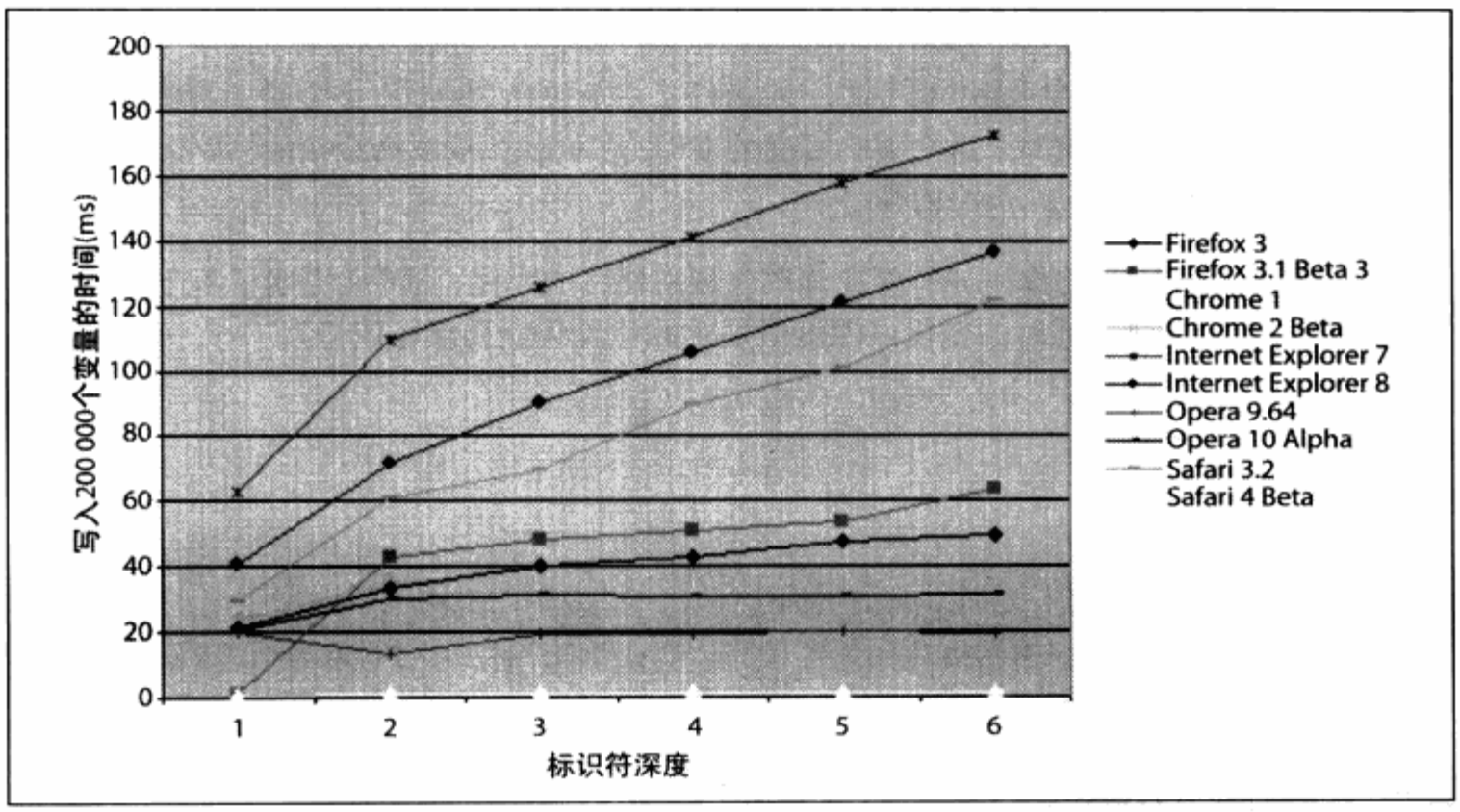


图 7-3：变量写入时间随标识符深度的变化

这些图清楚地表明，标识符在作用域链中的位置越靠上，存取的速度就越快。根据这个理论，你应该尽可能使用局部变量。一个好的经验是任何非局部变量在函数中的使用超过一次时，都应该将其存储为局部变量。例如：

```
function createChildFor(elementId){
    var element = document.getElementById(elementId),
        newElement = document.createElement("div");

    element.appendChild(newElement);
}
```

这个函数两次引用了全局变量 document。因为 document 使用超过了一次，为了更快的引用，我们应该将它存储到一个局部变量中，例如：

```
function createChildFor(elementId){
    var doc = document, // 存储到一个局部变量中
    element = doc.getElementById(elementId),
              newElement = doc.createElement("div");

    element.appendChild(newElement);
}
```

改写后的函数将 document 存储到名为 doc 的局部变量中。由于 doc 存在于作用域链的顶层，所以解析它比解析 document 更快。

请记住，全局变量对象始终是作用域链中最后一个对象，所以对全局标识符的解析总是最耗时的。

提示：给变量首次赋值时忽略 var 关键字是一个很常见的错误，这会带来性能问题，因为当对未声明的变量直接赋值时，JavaScript 引擎会自动将其创建成一个全局变量。

7.1.2 增长作用域链

Scope Chain Augmentation

在代码执行过程中，执行上下文对应的作用域链通常保持不变。然而有两个语句会临时增长执行上下文的作用域链。第一个是 with 语句，用于将对象属性作为局部变量来显示，使其便于访问。例如：

```
var person = {
    name: "Nicholas",
    age: 30
};

function displayInfo(){
    var count = 5;
    with(person){
        alert(name + " is " + age);
        alert("Count is " + count);
```

```
        }
    }

    displayInfo();
```

在这段代码中，person 对象传入 with 语句块，这样就可以像访问局部变量一样访问 name 和 age 属性了。实际上它是将一个新的变量对象添加到执行上下文作用域链的顶部。这个变量对象包含了指定对象（这里是 person）的所有属性，所以这些属性可以不使用点符号（.）访问。图 7-4 展示了当执行 with 语句时 displayinfo 的作用域链是如何增长的。

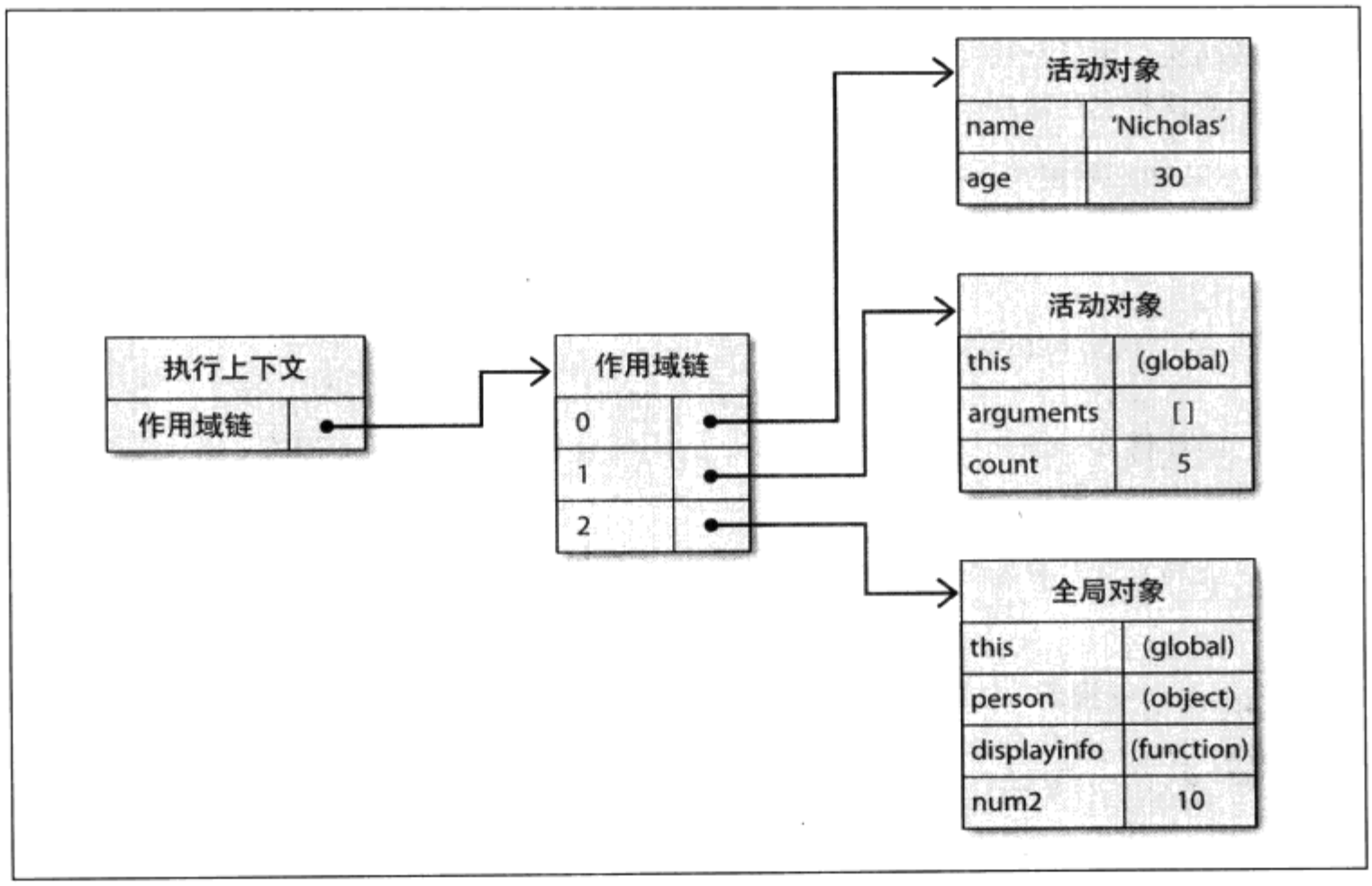


图 7-4：使用 with 语句加深作用域链

尽管 with 语句在反复使用同一对象属性时看起来很方便，但在作用域链中增加的额外对象影响了对局部标识符的解析。当执行 with 语句中的代码时，函数中的局部变量将从作用域链的第一个对象变为第二个对象，自然而然会减慢标识符的存取。在前面的例子中，存取 count 变量需要更长时间，因为它不在作用域链的第一个对象中。一旦 with 语句执行结束，作用域链将恢复到原来的状态。由于 with 语句存在这个主要的缺陷，建议你**避免**使用它。

第二个会增长作用域链的是 try-catch 语句块中的 catch 从句。在执行 catch 从句中的代码时，其行为方式类似于 with 语句，也是在作用域链的顶部增加了一个对象。该对象包含了由 catch 指定命名的异常对象。然而，由于 catch 从句仅在执行 try 从句发生错误时才执行，所以它比 with 语句的影响要小，但应该注意不要在 catch 从句中执行过多的代码，以将其带来的性能影响减小到最低。

管理好作用域链的深度，是一种只要少量工作就能提高性能的简易方法。我们要避免因不必要地增长作用域链而无意中导致执行速度变得缓慢。

7.2 高效的数据存取
Efficient Data Access

数据在脚本中存储的位置直接影响脚本执行的总耗时。一般而言，在脚本中有 4 种地方可以存取数据：

- 字面量值。
- 变量。
- 数组元素。
- 对象属性。

读取数据总会带来性能开销，而开销大小取决于数据存储在这 4 种位置中的哪一种。

在大多数浏览器中，从字面量中读取值和从局部变量中读取值的开销差异很小，以至于可以忽略不计；你可任意地混合使用字面量和局部变量，而无需担心性能问题。真正的差异在于从数组或对象中读取数据。存取这些数据结构中某个值，需要通过索引（对于数组）或属性值（对于对象）来查询数据存储的位置。

为了测试数据存取时间对所在位置的依赖，我创建了一个从各种位置读取 200 000 次数值的实验。你能在 *http://www.nczonline.net/experiments/javascript/performance/data-access/* 上找到该实验。我在多个浏览器上做这个实验，结果刚好一分为二：Internet Explorer、Opera 和 Firefox 3 存取数组元素比对象属性快；而 Chrome、Safari、Firefox 2 和 Firefox 3.1+存取对象属性比数组元素快（见图 7-5）。

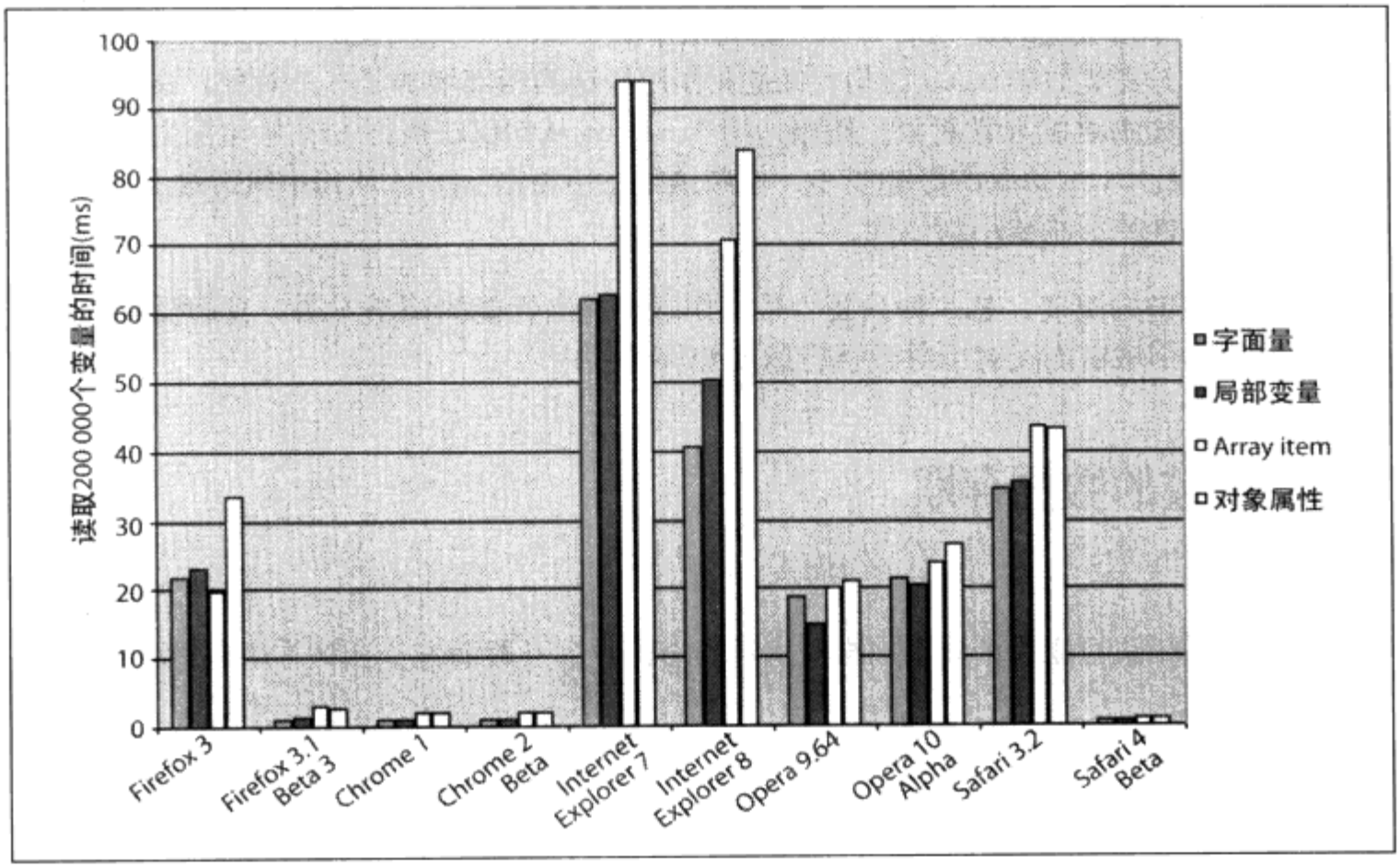


图 7-5：各浏览器存取数据的时间

从测试结果中可以得出一个重要启示：始终将那些需要频繁存取的值存储到局部变量中。请看下面的代码：

```
function process(data){
    if (data.count > 0){
        for (var i=0; i < data.count; i++){
            processData(data.item[i]);
        }
    }
}
```

这段代码多次读取 data.count 的值。乍一看只读取了两次：一次在 if 语句中，一次在 for 循环中。然而实际读取 data.count 的次数应该是 data.count 的值加 1，因为每次进入循环时都会执行控制语句（i < data.count）。如果将值存储到局部变量中，然后从局部变量中存取，函数将运行得更快：

```
function process(data){
    var count = data.count;
    if (count > 0){
        for (var i=0; i < count; i++){
            processData(data.item[i]);
        }
    }
}
```

改写后的函数在执行过程中读取 `data.count` 仅有一次，这是因为这个函数一开始就把 `data.count` 存储到局部变量中。这样在函数其他位置就可以直接使用局部变量 `count`，从而减少读取对象属性值的次数。由于减少了查找对象属性的次数，函数的执行效率将比之前的更高。

随着数据结构深度的增加，它对数据存取速度的影响也跟着变大。例如，存取 `data.count` 比 `data.item.count` 快，存取 `data.item.count` 比 `data.item.subitem.count` 快。在处理属性时，点符号（.）的使用（用于查找属性）次数直接影响存取该属性的总耗时。图 7-6 展示了各浏览器中属性深度与数据存取时间的关系。这个测试是我的数据存取实验的一部分，网址为 *http://www.nczonline.net/experiments/javascript/performance/data-access/*。

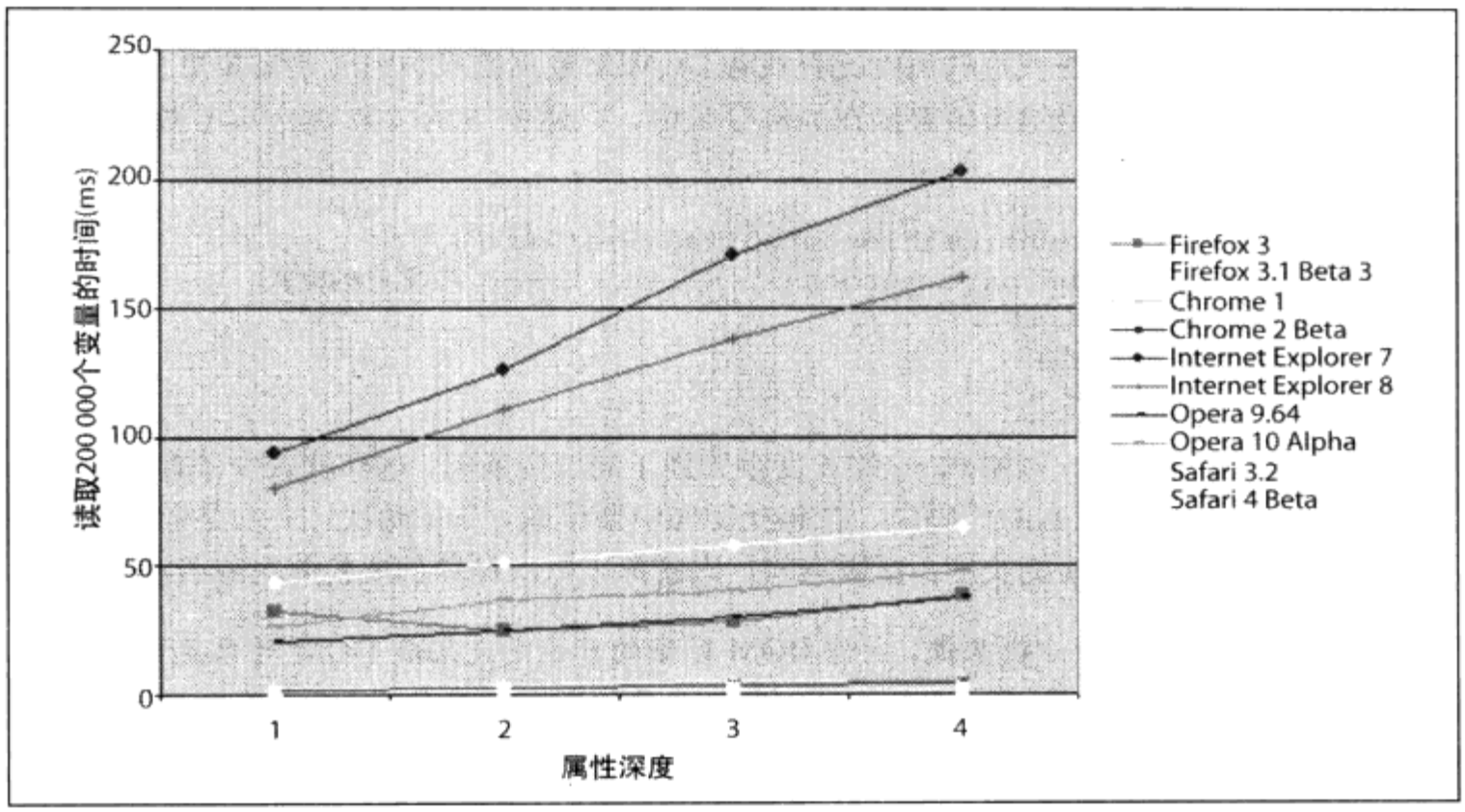


图 7-6：数据存取时间随对象属性深度的变化

在数据存取时，将函数中使用超过一次的对象属性或数组元素存储为局部变量是一种好方法。

提示：对于大多数浏览器来说，存取对象的属性使用点符号（`data.count`）还是方括号（`data["count"]`）几乎没有区别。但 Safari 浏览器是个例外，在它里面使用方括号明显比点符号慢。即使 Safari 4 和其后使用 Nitro JavaScript 引擎的其他版本也是如此。

在处理 HTMLCollection 对象（比如 getElementsByTagName 这样的 DOM 方法的返回值，或 element.childNodes 这样的属性值）时使用局部变量特别重要。实际上每次存取 HTMLCollection 对象的属性，都会对 DOM 文档进行动态查询。例如：

```
var divs = document.getElementsByTagName("div");
for (var i=0; i < divs.length; i++){ //避免!
    var div = divs[i];
    process(div);
}
```

这段代码的第一行创建了一个查询来获取页面中所有的 元素，并将查询结果存储到 divs 中。每次通过属性名或索引存取 divs 的属性时，DOM 实际上重复执行了一次对页面上所有 元素的查询；在这段代码中，每次读取 divs.length 或 divs[i]都会对页面查询一次。一般说来，查找这些属性比查找非 DOM 对象属性或数组元素需要更多的时间。因此重要的是：尽可能把这类值存储在局部变量中，以避免 HTMLCollection 对象引发的重新查询。例如：

```
var divs = document.getElementsByTagName("div");
for (var i=0, len=divs.length; i < len; i++){ //更好的方式
    var div = divs[i];
    process(div);
}
```

本例将 HTMLCollection 对象 divs 的长度存储到了局部变量中，这样就减少了直接存取该对象的次数 。上一个版本的代码中，在每次循环中要读取 divs 两次：一次是通过指定位置检索对象，另一次是检测长度。新版本的代码省掉了每次循环都要检测对象长度的运算。

提示：一般来说，操作 DOM 对象的开销总是比非 DOM 对象要大。由于 DOM 的这种行为，查询其属性通常比非 DOM 属性更耗时。HTMLCollection 对象是 DOM 中性能最糟糕的。如果需要对 HTMLCollection 对象的成员反复存取，更高效的方式是先将它们复制到一个数组里。

7.3 流控制

Flow Control

除了数据存取之外，流控制或许是提升 JavaScript 性能最重要的一环。JavaScript 与大多数编程语言一样，拥有一些流控制语句，这些语句决定下一步要执行哪部分的代码。开发人员能通过一系列条件和循环语句，精确地控制执行流从代码的一部分到另一部分。在每个环节上选择恰当的语句能够极大地提高脚本的运行速度。

7.3.1 快速条件判断
Fast Conditionals

使用 switch 语句还是一连串的 if-else 语句是一个经典问题，它不仅仅存在于 JavaScript 中，几乎每种拥有这些结构的语言都存在这样的讨论。当然，我们要讨论的真正问题不是语言本身，而是哪种语言能够更快地处理一系列条件判断。本节的内容以一些测试为基础，你可以在这个地址运行它们：*http://www.nczonline.net/experiments/javascript/performance/conditional-branching/*。

if 语句

对 if 语句的讨论通常是从这样一个复杂的语句开始的：

```
if (value == 0){
    return result0;
} else if (value == 1){
    return result1;
} else if (value == 2){
    return result2;
} else if (value == 3){
    return result3;
} else if (value == 4){
    return result4;
} else if (value == 5){
    return result5;
} else if (value == 6){
    return result6;
} else if (value == 7){
    return result7;
} else if (value == 8){
    return result8;
} else if (value == 9){
    return result9;
} else {
    return result10;
}
```

我们通常不赞成这种类型的结构。它的主要问题是语句的执行流越深，需要判断的条件就越多。当 value 为 9 时，执行完成的时间会比 value 为 0 时要长，因为它之前的所有条件都需要判断。随着条件总数的增加，条件越深性能损失就越大。尽管使用大量的 if 条件语句是不可取的，但你可以采取以下几个步骤来提高整体性能。

第一步是将条件按频率降序排列。由于最快的运算在第一个条件语句后就退出，所以要确保这种情况尽可能多地出现。假设前面的例子中最常见的情况是 value 等于 5，第二常见的是 value 等于 9。这就意味着，执行流在达到最常见的情况之前要进行 5 次条件判断，而在达到第二常见的情况之前要进行 9 次条件判断，这个效率是非常低下的。

尽管按数字递增的顺序排列条件更容易阅读，但实际上应该把它改写成下面的样子：

```
if (value == 5){
    return result5;
} else if (value == 9){
    return result9;
} else if (value == 0){
    return result0;
} else if (value == 1){
    return result1;
} else if (value == 2){
    return result2;
} else if (value == 3){
    return result3;
} else if (value == 4){
    return result4;
} else if (value == 6){
    return result6;
} else if (value == 7){
    return result7;
} else if (value == 8){
    return result8;
} else {
    return result10;
}
```

现在两个最常见的条件出现在 `if` 语句的顶部，确保了最佳性能。

另一种方法是优化 `if` 语句的条件，将条件拆分成几个分支，下面的二分查找算法可以逐步找出有效的条件。在条件数量众多，且没有出现频率特别高的条件，从而不能简单地按频率排列的情况下，这个方法是可取的。其最终的目标是尽量减少条件的判断。如果例子中所有条件对应的 `value` 值出现的频率一样，那么 if 语句可以改写成下面这个样子：

```
if (value < 6){

    if (value < 3){
        if (value == 0){
            return result0;
        } else if (value == 1){
            return result1;
        } else {
            return result2;
        }
    } else {
        if (value == 3){
            return result3;
        } else if (value == 4){
            return result4;
```

```
            } else {
                return result5;
            }
        }

    } else {

        if (value < 8){
            if (value == 6){
                return result6;
            } else {
                return result7;
            }
        } else {
            if (value == 8){
                return result8;
            } else if (value == 9){
                return result9;
            } else {
                return result10;
            }

        }
    }
```

这段代码确保任何情况下，都不会有超过 4 次的条件判断。并不是每个条件都会匹配准确的 value，而是在找出实际的 value 之前，首先分离出一系列的条件范围。这个例子的整体性能得到了提升，是因为原来会有八九次的条件判断，而现在最多只有 4 次，比前一版本平均节省约 30%的执行时间。另外请记住，`else` 语句不进行条件判断。

然而问题依然存在，每增加一个条件最终都会导致执行时间变长，这不仅影响性能，而且影响代码的可维护性。这时 `switch` 语句就有了用武之地。

switch 语句

`switch` 语句简化了多重条件判断的结构，并提升了性能。你可以像下面这样用 switch 语句重写前面的例子：

```
switch(value){
    case 0:
      return result0;
    case 1:
      return result1;
    case 2:
      return result2;
    case 3:
      return result3;
    case 4:
      return result4;
    case 5:
```

```
        return result5;
      case 6:
        return result6;
      case 7:
        return result7;
      case 8:
        return result8;
      case 9:
        return result9;
      default:
        return result10;
  }
```

这段代码更加具有可读性，清晰地表明了条件和返回值。switch 语句额外的好处是允许“贯穿（fall-through）”条件，这使你可以为不同的条件值指定相同的结果，从而避免了创建复杂的嵌套条件。其他编程语言常常推荐 switch 语句作为判断多重条件的更好选择，这并不是因为 switch 语句本身，而是因为这些语言的编译器能够优化 switch 语句，使它能更快地求值。由于大多数 JavaScript 引擎没有这样的优化，所以 switch 语句的性能参差不齐。

Firefox 可以很好地处理 switch 语句，每个条件判断执行的时间大致相同，与它们定义的顺序无关。这意味着 value 为 0 的情况与 value 为 9 的情况的执行时间大致相同。然而，其他浏览器做得不怎么好。在 Internet Explorer、Opera、Safari 和 Chrome 中，switch 语句越深，执行时间就增加得越明显，不过实验表明这比每个条件都用 if 语句要增加得少。因此你可以通过按出现频率的降序排列条件来提高 switch 语句的性能（与优化 if 语句一样）。

在 JavaScript 中，当仅判断**一两个条件**时，if 语句通常比 switch 语句更快。当有两个以上条件且条件比较简单（不是进行范围判断）时，switch 语句往往更快。这是因为大多数情况下，switch 语句中执行单个条件所需时间比在 if 语句中短，所以当有大量的条件判断时，使用 switch 语句更合适。

另一种选择：数组查询

在 JavaScript 中，处理条件判断的解决方案不止两种。除了 if 语句和 switch 语句外，还有第 3 种办法：在数组中查询值。本节示例是把已知的数字映射到指定的结果上，这正是数组的工作方式。你可以编写下面的代码来代替大量的 if 语句和 switch 语句：

```
//定义数组 results
var results = [result0, result1, result2, result3, result4, result5, result6,
result7,
               result8, result9, result10]

//返回正确的结果
return results[value];
```

不使用条件语句，而是把所有的结果都存储在数组中，并用数组的索引映射 value 变量。检索对应的结果就是简单地查询数组值。虽然查询数组的耗时也会随着进入数组的深度而增加，但和每个条件都用 if 语句或 switch 语句来判断相比，增加的时间还是要小得多。使用数组查询的理想情况是有大量的条件存在，并且这些条件能用数字或字符串（对于字符串，可以使用对象而非数组来存储）这样的离散值来表示。

使用数组查询少量的结果是不合适的，因为数组查询往往比少量的条件判断语句慢。当要查询的条件范围很大时，数组查询才会变得非常有用，因为它们不必检测上下边界；你只需简单地把对应的值填入数组的索引区域中就可以马上进行数组查询了。

最快的条件判断

前面列出的 3 种技术——if 语句、switch 语句和数组查询——在优化代码执行方面都有各自的用处。

- 使用 if 语句的情况：

  —两个之内的离散值需要判断。

  —大量的值能容易地分到不同的区间范围中。

- 使用 switch 语句的情况：

  —超过两个而少于 10 个离散值需要判断。

  —条件值是非线性的，无法分离出区间范围。

- 使用数组查询的情况：

  —超过 10 个值需要判断。

  —条件对应的结果是单一值，而不是一系列操作。

7.3.2 快速循环
Fast Loops

正如第 1 章所述，在 JavaScript 中循环是导致性能问题的常见起因，编写循环的方式能彻底改变它的执行时间。再强调一次，JavaScript 开发者无法依赖编译器去优化循环的速度，无

需去管源码的样子，所以了解各种编写循环的方式及它们对性能的影响，就显得尤为重要。

循环性能的简单提升

JavaScript 中有 4 种不同类型的循环。本节中，我们将讨论其中 3 种：for 循环，do-while 循环和 while 循环。（第 4 种是 for-in 循环，用于遍历对象的属性，由于它的用途非常独特，我在这里就不介绍了。）各种循环的代码如下所示：

```
//未优化的代码
var values = [1,2,3,4,5];

//for 循环
for (var i=0; i < values.length; i++){
    process(values[i]);
}

//do-while 循环
var j=0;
do {
    process(values[j++]);
} while (j < values.length);

//while 循环
var k=0;
while (k < values.length){
    process(values[k++]);
}
```

例子中的每个循环都达到了同样的目的：把 values 数组中的每个成员都传递给了 process 函数。这些循环都是遍历包含大量值的数组时的常用结构。由于每个循环所做的运算大致相同，所以执行时间也基本相同。然而还是有方法来提升性能的。

也许最显眼的问题就是循环中要反复地比较计数变量与数组长度。在本章前面的内容提到，查找属性要比存取局部变量更耗时。循环在每次判断是否达到结束条件时，都会去获取 values.length 的值。由于循环在执行时数组长度不会改变，所以这样做效率非常低。使用局部变量来代替属性查找能加快循环的执行效率：

```
var values = [1,2,3,4,5];
var length = values.length;

//for 循环
for (var i=0; i < length; i++){
    process(values[i]);
}
```

```
//do-while 循环
var j=0;
do {
    process(values[j++]);
} while (j < length);

//while 循环
var k=0;
while (k < length){
    process(values[k++]);
}
```

现在每个循环都使用局部变量 length 代替 values.length 进行条件比较，避免了每次循环中查找属性的过程。这个技巧在处理 HTMLCollection 对象时尤其重要，因为之前提到每次存取 HTMLCollection 对象属性实际上都会在 DOM 节点中查询所有符合某些条件的节点。这使得查找 HTMLCollection 对象的属性非常耗时，循环结束条件包含这样的运算时会明显增加循环的整体执行时间。

另外一种提高性能的简单有效的方式是将循环变量递减到 0，而不是递增到总长度。根据每个循环的复杂性不同，这个简单的改变可以比原来节约多达 50%的执行时间。例如：

```
var values = [1,2,3,4,5];
var length = values.length;

//for 循环
for (var i=length; i--;){
    process(values[i]);
}

//do-while 循环
var j=length;
do {
    process(values[--j]);
} while (j);

//while 循环
var k=length;
while (k--){
    process(values[k]);
}
```

因为结束条件被改造为与 0 进行比较（注意一旦循环变量等于 0，结束条件就会变为假），所以每个循环的速度更快了。3 种类型的循环性能表现差不多，因此你不用为了速度而在它们中间做选择。

提示：小心使用数组原生的 indexOf 方法，这个方法遍历数组成员的耗时可能会比使用普通的循环还长。如果速度是首要考虑的因素，那么请使用本节中提到的 3 种循环中的一种。

避免 for-in 循环

for 循环的另一种变化是 for-in 循环，它用来遍历 JavaScript 对象的可枚举属性。典型的用法如下所示：

```
for (var prop in object){
    if (object.hasOwnProperty(prop)){ //确保只处理实例自身的属性
        process(object[prop]);
    }
}
```

这段代码遍历了给定对象的属性，并使用 hasOwnProperty 方法确保只处理实例属性。

由于 for-in 循环用途特殊，所以很少有什么地方能改善其性能。它的结束条件无法改变，而且遍历属性的顺序也无法改变。此外 for-in 循环通常比其他循环慢，因为它需要从一个特定的对象中解析每个可枚举的属性。反过来说，这意味着它为了提取这些属性需要检查对象的原型和整个原型链。遍历原型链就像遍历作用域链，它会增加耗时，从而降低整个循环的性能。

如果你明确知道要操作的所有属性名，那么用标准循环（for、do-while 或 while）对属性名进行遍历要快得多，如下所示：

```
//要遍历的已知属性
var props = ["name", "age", "title"];

//while 循环
var i=props.length;
while (i--){
    process(object[props[i]]);
}
```

这个循环比 for-in 循环快得多，这不仅仅是因为 props 数组中涉及的属性数量少，即便需遍历的属性数量增多，其性能也远远好于 for-in 循环。在这个例子中循环采用了所有普通循环的性能增强方式，而且仍然能遍历对象的一组已知属性集。

当然，只有在已知需要遍历的对象属性集时才能采用这种方法；在处理诸如 JSON 对象这样的未知属性集时，依然必须使用 for-in 循环。

展开循环

在一些编程语言中，习惯做法是展开小循环以提高性能。这一做法的基础是通过限制循环的次数来减少循环的开销。实施这样的解决方案称为**展开循环**，即意味着使每次循环完成多次循环的工作。请看下面的循环：

```
var i=values.length;
while (i--){
    process(values[i]);
}
```

如果在 values 数组中仅有 5 项，那么去掉循环，并单独地处理每个值实际上更快：

```
//展开循环
process(values[0]);
process(values[1]);
process(values[2]);
process(values[3]);
process(values[4]);
```

当然，这种方式降低了维护性，因为它需要编写更多的代码，而且 values 数组项的数量发生任何改变都需要修改代码。此外，为了从这样少量的语句中获得性能提升而提高维护成本是不值得的。然而当你处理大量的值且循环次数可能很多时，这项技术就非常有用了。

Tom Duff 在 Lucasfilm 任计算机程序员时，首先在 C 语言中提出了展开循环的构想。所以这种模式称为 Duff 策略（Duff's Device），后来 Jeff Greenberg 把它引入 JavaScript 中，他也是最早对 JavaScript 性能优化进行深入研究的人之一（他的文章在 *http://home.earthlink.net/~kendrasg/info/js_opt/* 上仍然可见）。Greenberg 的 Duff 策略实现如下所示：

```
var iterations = Math.ceil(values.length / 8);
var startAt = values.length % 8;
var i = 0;

do {
    switch(startAt){
        case 0: process(values[i++]);
        case 7: process(values[i++]);
        case 6: process(values[i++]);
        case 5: process(values[i++]);
        case 4: process(values[i++]);
        case 3: process(values[i++]);
        case 2: process(values[i++]);
        case 1: process(values[i++]);
    }
    startAt = 0;
} while (--iterations > 0);
```

Duff 策略背后的思想是每一次循环完成标准循环的 1~8 次。首先通过数组值的总数除以 8 来确定循环次数。Duff 发现对于这个处理过程来说，8 是最佳数值（不是任意值）。由于并非所有数组的长度都能够被 8 整除，所以你必须通过取余运算（译注 2）计算出将有多少项不被额外处理。因此 `startAt` 变量是要额外处理的数组项的数量，该变量仅在第一次循环中使用，做完额外的工作后它将被重置为 0，这样后面每次循环都正好处理 8 个数组项。当需要进行大量循环时，使用 Duff 策略比标准循环要快得多，但其实它还可以更快些。

在《Speed Up Your Site》(New Riders 出版社)一书中介绍了在 JavaScript 中实现 Duff 策略的另一个版本，该版本把对额外数组项的处理移到主循环外，这样就可以去掉 `switch` 语句，从而得到一个处理大数组的更快方法：

```
var iterations = Math.floor(values.length / 8);
var leftover = values.length % 8;
var i = 0;

if (leftover > 0){
    do {
        process(values[i++]);
    } while (--leftover > 0);
}

do {
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
} while (--iterations > 0);
```

这段代码之所以对大量数组项执行更快，主要是因为从主循环中去掉了 `switch` 语句。正如本章前面讨论的那样，条件判断也有性能开销；从算法中移除这个开销可以加快处理速度。分开处理两个独立的循环提升了性能。

Duff 策略及上面提到的修改后的版本，主要是用于处理大数组。对于小数组而言，相比标准循环所带来的性能提升很小。因此仅当你注意到性能的瓶颈是由循环处理大量元素项所引起的时，才应该尝试使用 Duff 策略。

译注 2：取模（modulo）和取余（remainder）大部分情况下是等同的，但是还是有些不同，取模结果的符号和除数相同，取余结果符号和被除数相同，而 JavaScript 中的%和取余相同。

7.4 字符串优化

String Optimization

在 JavaScript 中字符串操作十分常见。字符串的处理有多种方式，采用哪种方式取决于具体的任务，以及每个任务特殊的性能考虑。有许多不同的方式来操作字符串，无论是使用字符串的内置方法或运算符，还是混合使用正则表达式或数组，操作类型直接决定用于性能优化的具体技术。

7.4.1 字符串连接

String Concatenation

传统上，字符串连接一直是 JavaSscript 中性能最低的操作之一。通常情况下，字符串连接是通过使用加法运算符（+）来完成的，如下所示：

```
var text = "Hello";
text += " "
text += "World!";
```

早期浏览器没有对这种运算进行优化。由于字符串是不可变的，这意味着要创建中间字符串来存储连接的结果。频繁地在后台创建和销毁字符串会导致字符串连接的性能异常低下。

发现了这一点之后，开发者就利用 JavaScript 的 `Array` 对象进行了补救。`Array` 对象有一个 `join` 方法，用来连接数组中所有的元素并且在元素之间插入指定的字符串。开发者把每个字符串都添加到数组中，然后调用 `join` 方法，这样就避免了使用加法运算符。例如：

```
var buffer = [],
    i = 0;
buffer[i++] = "Hello";
buffer[i++] = " ";
buffer[i++] = "World!";

var text = buffer.join("");
```

在这段代码里，我们把每个字符串都添加到 `buffer` 数组中。在所有字符串都加入到数组后调用 `join` 方法，返回的已连接字符串保存在变量 `text` 中。通过相应的索引直接添加元素比调用 `push` 方法略快一点。在早期的浏览器中，没有创建和销毁中间字符串，这技术已被证明远快于使用加法运算符的方式，不过，如今浏览器对字符串的优化已经改变了字符串连接的局面。

Firefox 是第一款优化字符串连接的浏览器。从 1.0 版本开始，在所有情况下使用数组技术实际上都比使用加法运算符慢。其他浏览器也优化了字符串连接，Safari、Opera、Chrome

和 Internet Explorer 8 也都在使用加法运算符上表现出了更好的性能。Internet Explorer 8 之前的版本没有做这样的优化，因此数组技术依然比加法运算符更快。

这并不意味着执行连接字符串时必须进行浏览器检测。在决定如何连接字符串时要考虑两个因素：被连接的字符串大小和数量。

当字符串相对较小（少于 20 个字符）且连接的数量也较小时（少于 1000 个），所有的浏览器中使用加法运算符都能在不到 1 毫秒之内轻松完成连接。在这种情况下就没有理由去考虑加法运算符以外的方式了。

增加连接字符串的数量或大小时，在 Internet Explorer 7 中性能会明显下降。当字符串大小增加时，在 Firefox 中加法运算符和数组技术的性能差异会变小。当字符连接数量增加时，在 Safari 中这两种技术的性能差异也同样会变小。只有 Chrome 和 Opera 在改变连接字符串的大小和数量时，加法运算符一直保持着显著的性能优势。

由于在各浏览器下性能不一致，所以选用哪种技术在很大程度上取决于实际情况和面对的浏览器。如果用户主要使用 Internet Explorer 6 或 7，那么使用数组技术就很值得，因为这会影响大多数人。通常使用数组技术在其他浏览器中的性能损失要远小于在 Internet Explorer 中的性能提升，所以要基于用户的浏览器来权衡用户体验，而不要试图去针对某种具体情况或浏览器版本。不过，在大多数情况下，加法运算符是首选的。

7.4.2 裁剪字符串

Trimming Strings

JavaScript 中没有用于移除字符串头尾空白的原生修剪方法，这是其最明显的疏漏之一。最常见的 `trim` 函数实现如下所示：

```
function trim(text){
    return text.replace(/^\s+|\s+$/g, "");
}
```

这种实现使用一个正则表达式匹配字符串开头和结尾的一或多个空白字符。字符串的 `replace` 方法用空字符串替换所有匹配的部分。然而这个实现方式有个基于正则表达式的性能问题。

对性能的影响来自于正则表达式的两个方面：一方面是指明有两个匹配模式的管道运算符，另一方面是指明全局应用该模式的 g 标记。考虑到这些，你可以将正则表达式一分为二并去掉 g 标记来重写该函数，以此来稍稍提高它的速度。

```
function trim(text){
    return text.replace(/^\s+/, "").replace(/\s+$/, "");
}
```

将单个的 replace 方法拆分为两次调用，可使每个正则表达式变得更简单，因此也更快。这个方法比原版本快，但你还可以使它更快。

Steven Levithan 在进行性能研究后提出了在 JavaScript 中执行速度最快的裁剪字符串方式，该函数如下所示：

```
function trim(text){
    text = text.replace(/^\s+/, "");
    for (var i = text.length - 1; i >= 0; i--) {
        if (/\S/.test(text.charAt(i))) {
            text = text.substring(0, i + 1);
            break;
        }
    }
    return text;
}
```

这个 trim 函数始终比其他变型的版本性能更好，原因是保持正则表达式尽可能地简单。第一行删除了头部的空白，然后通过 for 循环来清除尾部的空白。循环使用一个非常简单的单字符正则表达式来匹配非空白字符，它清除了字符串尾部的空白后退出循环。在所有浏览器中，最终的函数都比之前版本的执行速度更快。Levithan 的博客文章对此有完整的分析：*http://blog.stevenlevithan.com/archives/faster-trim-javascript*。

和字符串连接一样，裁剪字符串的速度只有在整个执行过程中裁剪的频率足够大时才重要。本节中第 2 个 trim 函数在小规模地处理短字符串时性能还很好；而第 3 个 trim 函数在处理长字符串时明显更快。

提示：ECMAScript 规范的下一个版本名为 ECMAScript 3.1（译注 3），为字符串定义了原生的 trim 方法；这个原生的方法很可能会比本节中任何版本都要快。如果是这样，就应该使用原生函数。

译注 3：ECMAScript 3.1 在正式发布时被命名为 ECMAScript 5，它于 2009 年底正式发布，支持原生的 String.trim()，更多详情请看 *http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf*。

7.5 避免运行时间过长的脚本

Avoid Long-Running Scripts

JavaScript 有一个饱受批评的性能问题，那就是代码执行导致页面被冻结而出现假死。因为 JavaScript 是单线程语言，所以在一个时间段中，每个窗口或标签页中只能执行一个脚本。这意味着在 JavaScript 代码执行时，所有用户的交互必然被中断。这是浏览器的一个重要的特性，因为 JavaScript 在执行的过程中可以改变页面的底层结构，所以有可能取消或改变用户交互的响应结果。

如果 JavaScript 代码未经过细心的设计，有可能长时间地冻结页面，并最终导致浏览器停止响应。大多数浏览器会检测到长时间运行的脚本，并弹出中止脚本运行对话框询问用户是否允许脚本继续执行下去。

不同的浏览器对何时弹出中止脚本运行对话框有不同的判断标准：

- Internet Explorer 监控脚本执行的语句的数量。当执行的语句的数量达到最大限定值，默认为 500 万，就会弹出中止脚本运行的对话框（如图 7-7 所示）。
- Firefox 检测脚本运行的总时间。当超过预先设定的时间，默认为 10 秒，就会中止脚本运行的对话框。
- Safari 也是通过执行时间来检测脚本是否为长时间运行。默认超时设置为 5 秒，超时就会弹出中止脚本运行的对话框。
- Chrome 1.0 版没有限制 JavaScript 允许执行的最长时间。当出现内存不足时进程就会崩溃。
- Opera 是唯一没有提供检测长时间运行脚本这种保护功能的浏览器，允许脚本持续执行直到完成。

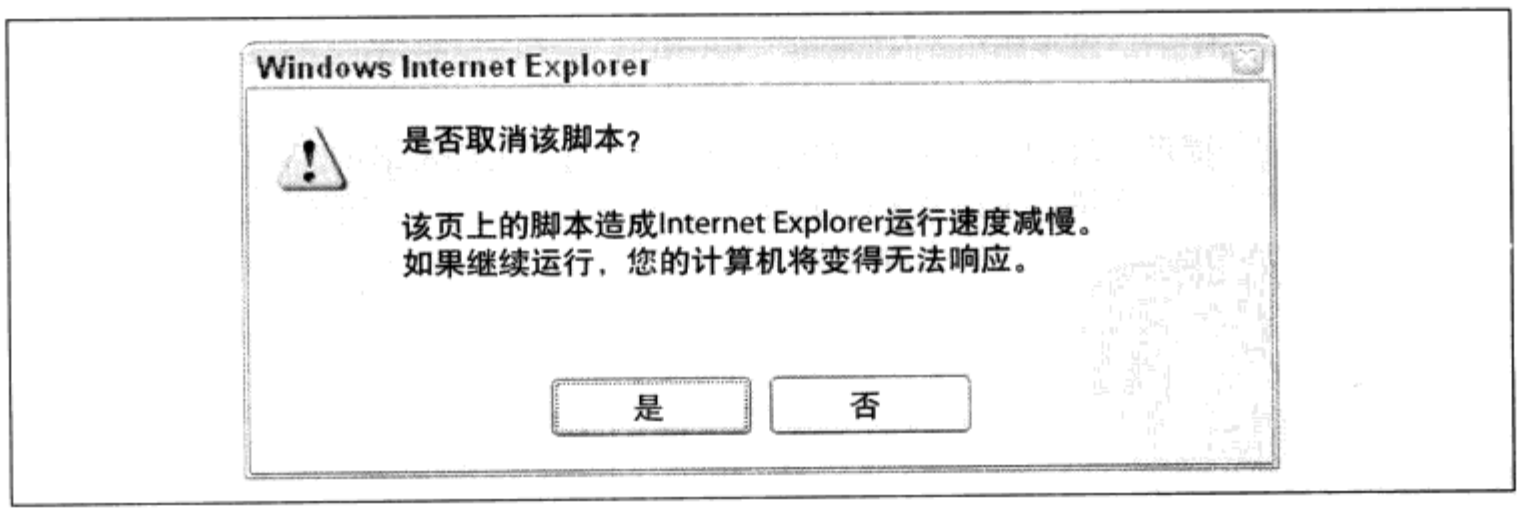


图 7-7：Internet Explorer 7 的长时间运行脚本对话框

如果你看到中止脚本运行的对话框，这就表示 JavaScript 代码需要进行重构。一般而言，脚本执行时间不应该超过 100 毫秒；任何超过这个时间的网页几乎肯定会让用户感觉运行速度过慢。JavaScript 之父 Brendan Eich 也说过“如果[JavaScript]的执行时间超过 1 秒，那么很可能是什么地方做错了……”。

最常见的脚本执行时间过长的原因包括：

过多的 DOM 交互

DOM 操作比其他任何 JavaScript 操作的开销都高。尽可能地减少 DOM 交互可显著减少 JavaScript 的运行时间。大多数浏览器只会等待整个脚本执行完成后才更新 DOM，这样会让用户感觉页面响应缓慢。

过多的循环

执行次数太多或每一次迭代中执行运算过多的循环都可能会导致脚本运行时间过长。尽可能地将功能分解有助于缓解这个问题。循环执行 DOM 操作会导致更糟糕的问题，有时会造成浏览器完全冻结而出现假死，甚至中止脚本运行的对话框都不会弹出来。

过多的递归

JavaScript 引擎限制了脚本可递归的次数。改写代码来避免递归有益于缓解这个问题。

牢记以上这些问题，有时简单地重构代码可以防止脚本失控。但也许有时正常运行 Web 应用就必须要执行复杂的处理，在这种情况下，代码需要重构来实现周期性地释放，这将在下一节具体解释。

7.5.1 使用定时器挂起（译注 4）

Yielding Using Timers

JavaScript 的单线程本质特性意味着任何时间段内在一个窗口或标签页中仅能执行一个脚本。因为在此期间不能处理用户的交互行为，所以有必要在长时间执行的 JavaScript 代码中加入中断。在简单的 Web 页面中，用户与页面交互时中断会很自然地出现，而在复杂的 Web 应用程序中，需要你自己来插入中断。最简单的方法是使用定时器。

创建定时器就是向 `setTimeout` 函数传递需要执行的函数及延迟执行该函数的时间(以毫秒为单位)。传入延迟时间后，执行代码被放置一个队列中。JavaScript 引擎通过这个队列来决定下一步怎么做。当一个脚本执行完成时，JavaScript 引擎挂起以便浏览器执行其他任务。页面利用这段时间来更新由脚本引起的改变。一旦页面更新完成，JavaScript 引擎就检查队

译注 4：这里对 JavaScript 执行方式的处理类似 Java 中的线程，yield 正是 Java 中对线程控制的一个方法，该方法使正在被服务的线程可能因某种原因而提前退出，线程重新回到可执行状态，所以执行 yield()的线程有可能在进入到可执行状态后马上又被执行。JavaScript 这里的处理策略与 yield 方法很相似，所以用了 yield 一词；另外在 JavaScript 1.7 中加入新关键字‘yield’，也是用来模拟线程的工作方式，详细可见 *https://developer.mozilla.org/en/New_in_JavaScript_1.7*。

列中轮到哪个脚本去运行。如果有脚本正在等待，则执行它然后重复这个过程；如果没有脚本执行，JavaScript 引擎将保持空闲直到另有脚本进入队列中。

当创建一个定时器时，实际上你是把某些代码排到 JavaScript 引擎队列中稍后执行。在调用 setTimeout 时，插入的代码会在等待指定的时间后执行。从本质上讲，定时器是把代码执行的时间推迟到将来，那时中止脚本运行的限制已经被重置了。看下面的代码：

```
window.onload = function(){

    //页面加载完成

    //创建第 1 个定时器
    setTimeout(function(){

        //被延迟的脚本 1

        setTimeout(function(){

            //被延迟的脚本 2

        }, 100);

        //被延迟的脚本 1，继续执行

    }, 100);

};
```

这个例子中，脚本在页面加载后运行。脚本调用 setTimeout 创建了第 1 个定时器。当该定时器执行时，再次调用 setTimeout 创建了第 2 个定时器。而第 2 段延迟脚本直到第 1 段延迟脚本执行完毕且浏览器更新页面展示后才会执行。图 7-8 显示了这段代码执行的时间线，表明两段脚本没有在同一时间运行。

定时器是在浏览器中拆分执行 JavaScript 代码的惯用方式。每当脚本需要花太长时间来完成执行的时候，就让部分延迟执行。

注意过小的延迟也会引起浏览器不响应。不建议使用 0 毫秒的延迟，因为所有的浏览器都无法在这么短的时间里正确地更新页面显示。一般而言，延迟 50~100 毫秒是合适的，这足够让浏览器有时间执行必要的页面更新。

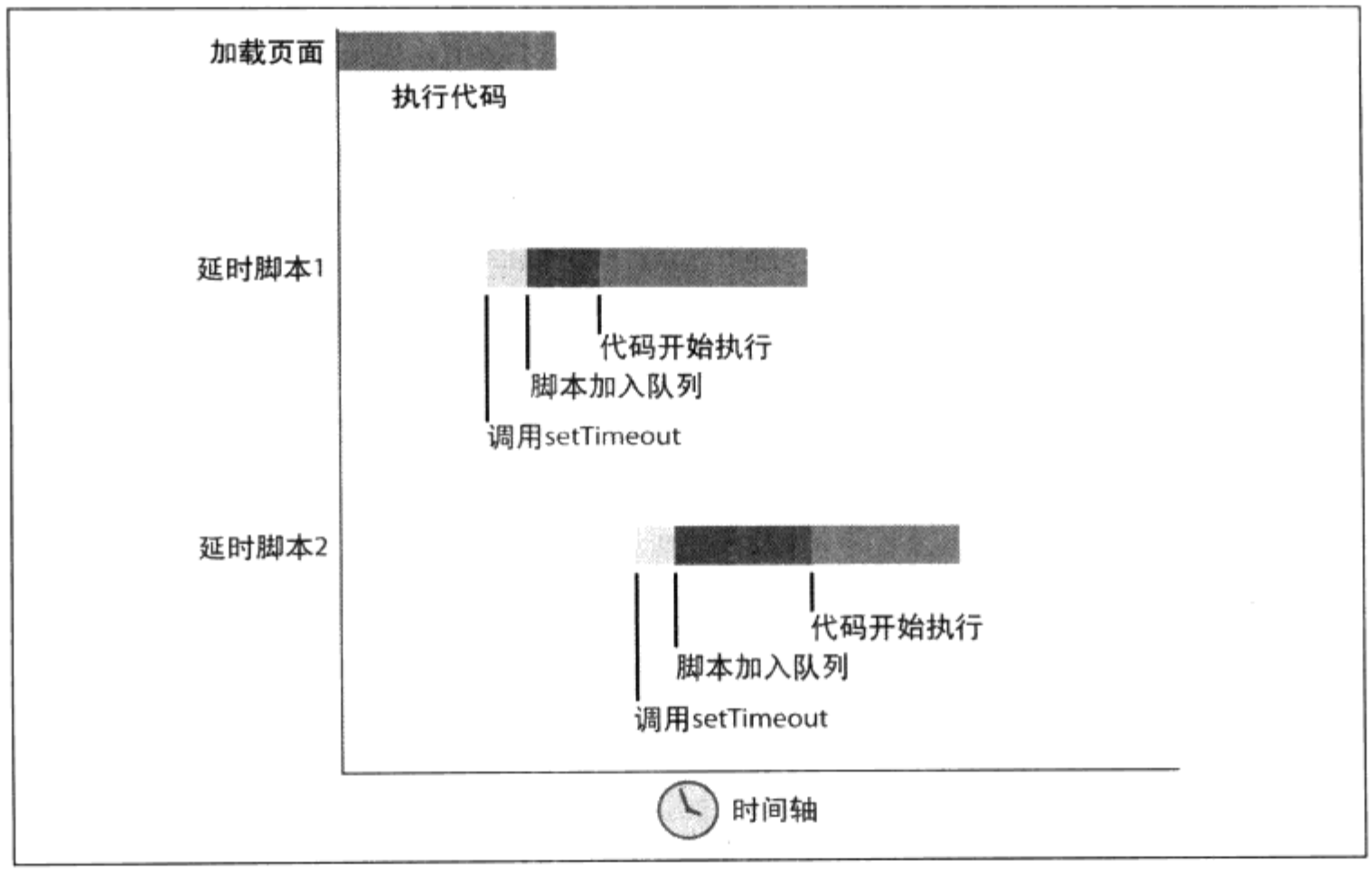


图 7-8：用定时器执行 JavaScript 代码

7.5.2 用于挂起的定时器模式

处理数组是引起脚本长时间运行的常见原因之一，通常需要对每个数组项进行处理，所以执行时间与数组项数量成正比。如果处理数组时没有同步要求，那么利用定时器拆分执行是一个很好的备选方案。

在我编写的《Professional JavaScript for Web Developers》第二版中（Wrox 出版社），我介绍了一个简单的利用定时器来拆分处理数组的函数：

```
function chunk(array, process, context){
    setTimeout(function(){
        var item = array.shift();
        process.call(context, item);

        if (array.length > 0){
            setTimeout(arguments.callee, 100);
        }
    }, 100);
}
```

chunk 函数接受 3 个参数：一个是需要处理的数组，一个是用来处理每个数组元素的函数，最后一个是可选的用来设置处理函数执行时的上下文(默认所有传入 setTimeout 方法的函数都在全局上下文中运行，所以 this 为 window)。处理元素用到了定时器，所以在每个元素处理后代码会被挂起。处理的下个元素始终在数组的顶部，并且在处理前移出数组。接着检查是否还有元素要处理。如果有，创建一个新的定期器，函数通过 arguments.callee 再次被调用。注意 chunk 函数把传入的数组作为“将要处理”的元素项列表，执行过程中数组将被改变。你可以使用如下函数：

```
var names = ["Nicholas", "Steve", "Doug", "Bill", "Ben", "Dion"],
    todo = names.concat(); //复制数组

chunk(todo, function(item){
    console.log(item);
});
```

这段简单的代码示例通过 console（在已安装 Firebug 的 Firefox、Internet Explorer 8+、Safari 2+和 Chrome 所有版本中可用）输出 names 数组中每个名字。这个处理函数很简单，但你可以很容易地把它替换为更复杂的处理函数。chunk 函数最好用于需要对大数组每个元素都进行大量处理的情况。

另一种流行的模式是利用定时器执行较大型运算中小而有序的部分。Julien Lecomte 在他的博客文章《在 Web 浏览器中运行耗 CPU 的 JavaScript 计算》(Running CPU Intensive JavaScript Computations in a Web Browser，*http://www.julienlecomte.net/blog/2007/10/28/*）中提出这一模式，里面展示了如何通过一个有效的算法（冒泡排序）来对大型数据集进行排序，从而避免脚本长时间运行的问题。下面是对 Lecomte 代码的改编版本：

```
function sort(array, onComplete){

    var pos = 0;

    (function(){

        var j, value;

        for (j=array.length; j > pos; j--){
            if (array[j] < array[j-1]){
                value = data[j];
                data[j] = data[j-1];
                data[j-1] = value;
            }
        }

        pos++;

        if (pos < array.length){
```

```
            setTimeout(arguments.callee,10);
        } else {
            onComplete();
        }

    })();

}
```

sort 函数拆分了 array 排序时的每次遍历，让浏览器在对数组处理的过程中还能做些其他事情。内部匿名函数被立即调用去执行第 1 次遍历，随后传递 arguments.callee 给 setTimeout 创建一个定时器。当数组完成排序后，调用 onComplete 函数来通知开发者数据已经处理完成。你可以像下面这样使用这个函数：

```
sort(values, function(){
    alert("Done!");
});
```

对大数组进行排序时，不同浏览器的响应速度有明显差异。

7.6 总结

Summary

JavaScript 的执行速度取决于代码的编写方式。 本章就介绍了好几种能实现 JavaScript 提速的方法：

- 管理作用域非常重要，因为存取非局部变量要比局部变量耗时更多。尽量避免使用会增长作用域链的结构，比如使用 with 语句和 try-catch 语句中的 catch 从句。如果非局部变量的使用超过一次，那么为了降低性能损耗，就应该将它存储到一个局部变量中。
- 存储和读取数据的方式对脚本性能影响极大。字面量和局部变量总是最快的；存取数组元素和对象属性会引起性能损耗。如果数组元素或对象属性的使用超过一次，那么为了提高存取速度，就应该将它存储到一个局部变量中。
- 流控制也是影响脚本执行速度的一个重要因素。条件判断有 3 种处理方式：if 语句、switch 语句和数组查找。if 语句适用于少量离散值或一段区间值的判断；switch 语句最好用于对 3~10 个离散值的判断；数组查找在处理大量离散值时效率最高。
- 在 JavaScript 中，循环经常会成为性能瓶颈。为了使循环最高效，可以采用倒序的方式来处理元素，也就是在控制条件中，将迭代变量和 0 作比较。相比非 0 值，这种方式要快得多，从而显著提升数组的处理速度。如果必须要进行大量的迭代，还可以考虑使用 Duff 策略来提高执行速度。

- 谨慎使用 `HTMLCollection` 对象。每次存取这类对象的属性，都会重新查询 DOM 中匹配的节点。为了避免这种高昂的开销，只有在必要时才存取 `HTMLCollection` 对象，并将经常存取的值（例如 `length` 属性）存储在局部变量中。
- 常见的字符串操作可能会带来意料之外的性能问题。Internet Explorer 处理字符串连接的速度比其他浏览器要慢很多，但这没什么大不了，除非你要一次执行 1000 次以上的字符串连接。你可以对 Internet Explorer 处理字符串连接的方式进行优化：先将所有要连接的字符串存储到数组中，然后调用 `join()` 方法合并它们。去除字符串两端的空白也可能很耗时，这取决于字符串的大小。如果脚本中经常需要去除字符串两端的空白，那请确保使用最优算法。
- 浏览器会限制 JavaScript 可以运行的最长时间，有些会以执行语句的数量作为判断条件，有些则会控制 JavaScript 引擎执行的总时间。你可以使用定时器将任务拆分执行，从而绕开这些限制，避免浏览器弹出中止脚本运行的警告。

第 8 章

可伸缩的 Comet

Scaling with Comet

Dylan Schiemann

有时候 Ajax 还是不够快。

当需要把数据从服务端异步推送到客户端时，通常仅靠 Ajax 是无法胜任的。Comet 是通用术语，描述技术、协议和为浏览器提供可行且可扩展的低延迟数据传输实现的集合。Comet 不是首字母缩写词，而是 Alex Russell（译注 1）对术语 Ajax 幽默一把而新创的词（注 1）。

Comet 的目标包括随时从服务端向客户端推送数据、提升传统 Ajax 的速度和可扩展性，以及开发事件驱动（译注 3）的 Web 应用。

显然，Ajax 和后台 HTTP 请求都是可以增强当今 Web 应用程序性能的典型技术。然而，浏览器和 HTTP 中所使用的传统的请求/响应（request/response）模式无法满足诸如聊天、财经信息和文档协作之类有更苛刻要求的实时应用的需要。为了满足对用户体验的期望，所有这些应用都需要通过低延迟的数据传输来实现。

在本章，我会简要地介绍 Comet 如何工作，讨论现在通用的技术和每种技术的优缺点。在文章的最后，我会讨论使用 Comet 技术时如何解决跨域 Comet 和其他 Web 应用的实现。

8.1 Comet 工作原理

How Comet Works

Comet 利用 HTTP 规范中不常用的特性来工作。通过更智能的长连接管理和减少每个连接

译注 1：Alex Russell 是 Dojo Toolkit 的项目主管和 Dojo Foundation 的主席，他在博客文章《Comet: Low Latency Data for the Browser》（*http://alex.dojotoolkit.org/?p=545*）中提出了 Comet 这个术语。

注 1：Ajax 和 Comet 生活在厨房的洗涤盆里（译注 2）。

译注 2：Ajax 和 Comet 都是美国知名的清洁剂品牌。Ajax 是高露洁-棕榄公司 1947 年推出的清洁剂（*http://www.colgate.com.cn/app/Colgate/CN/Corp/History/1931.cvsp*），而 Comet 是号称美国第一的清洁剂（*http://www.cometcleanser.com/*）。

译注 3：事件驱动的程序设计是一种面向用户的程序设计方式。相对于面向过程的程序设计方式来说，事件驱动的程序设计方式是一种被动的程序设计方式。程序总是处于等待用户输入事件状态，然后被动地等待用户操作；用户的各种操作被称为事件，事件驱动的程序设计方式需要事先为各种需要处理的事件编写事件响应函数。

占用的服务端资源，Comet 比传统 Web 服务更易于提供更多的同步连接，客户端与服务端之间的数据传输更快。

大规模的应用必须使用异步连接处理，因为传统的服务架构中每个连接都需要使用一个线程。对于高并发的应用，Comet 服务器通常会根据操作系统来改进事件库，例如 libevent（注 2）、epoll（注 3）和 kqueue（注 4）。操作系统处理异步 I/O 的方式多种多样，传统的方式是选择（select）或轮询（poll）。应用程序可以通过这些结构来询问操作系统哪些套接字（Socket）（译注 4）已经可以写入和读取从而避免发生阻塞。

如果应用程序的规模并不大，但想通过 Comet 获益该怎么办呢？例如，一个每天访问量为 50 000 且连接时间通常是 3 分钟的站点，平均只打开 92 个连接。即使你可能依靠服务器来提升最大线程数，但 92 个线程对于追求高性能的小网站来说也并非一个好方法。

对基于 Comet 的高性能站点来说，每个连接使用一个线程是有问题的，所以大部分 Comet 服务器或明显地减少每个线程的资源开销，或者使用微线程或进程。例如，ErlyComet（注 5）是用 Erlang（译注 5）写的，这是一门运行于虚拟机且基于微线程的函数式语言。因为一个连接由一个进程表示，且 Erlang 的事件驱动机制便于进程通过信息传递来建立彼此之间的联系，所以即使在不同的服务器上 Erlang 也非常容易扩展连接的数量。

相反，作为 Comet 服务端语言，PHP 因其线程模型而成为非常差的选择，所以大多数使用 Comet 的 PHP Web 应用需要采用**分离**式（译注 6）（注 6）。该方案通过 PHP 编写的 Comet 客户端与使用另一门语言编写的服务端通信。一般来讲，尽管 Comet 服务端的编程语言不重要（不乏有在 PHP Comet 服务端上的尝试），但像 C、Erlang 和 Python 这几门语言更适合创建 Comet 服务端，也有用 Java 写的许多大型的 Comet 服务端。我们使用术语“一体的”（on-board）来描述 Web 服务端和 Comet 服务端相同的情况。

尽管一体的 Comet 使用简单，且通常运行在同一个域，但在大型网站中分离的 Comet 更常见，特别是那些主要开发语言不适合 Comet 性能要求的站点。例如，像 Facebook 这样的站

注 2：*http://monkey.org/~provos/libevent/*。

注 3：*http://linux.die.net/man/4/epoll*。

注 4：*http://people.freebsd.org/~jlemon/papers/kqueue.pdf*。

译注 4：套接字（socket）用于描述 IP 地址和端口，是一个通信链的句柄。应用程序通常通过“套接字”向网络发出请求或者应答网络请求（*http://en.wikipedia.org/wiki/Internet_socket*）。

注 5：*http://code.google.com/p/erlycomet/*。

译注 5：Erlang 是一个结构化，动态类型编程语言，内建并行计算支持。最初是由爱立信专门为通信应用设计的，比如控制交换机或者变换协议等，因此非常适合于构建分布式，实时软并行计算系统。

译注 6：这里的分离式（off-board）是指 Comet 的服务端和客户端（即 web 服务端）使用不同的语言。

注 6：*http://cometdaily.com/2008/05/22/on-board-vs-off-board-comet/*。

点或许会使用分离方案来实现它的聊天应用，而像 Meebo 这样的站点会使用一体的方案，因为其整站的通信差不多都使用了 Comet 技术。

在客户端，常用的技术包括轮询、长轮询、永久帧（forever frame/iframe）、XHR 流和即将出现的 WebSocket（译注 7）。除了这些实现 Comet 连接的技术，还有许多在客户端和服务端之间发送信息的协议。像 Dojo Toolkit 工具包(*http://dojotoolkit.org/*)或 js .io 库(*http://js.io/*)能为你自动处理很多这样的复杂性问题，但了解在没有工具包时这些技术如何工作及如何评估和优化 Comet 性能是至关重要的。

8.2 传输技术

Transport Techniques

现在来讲述 4 种实现低延迟数据传输的方法，它们是 Comet 的基础：轮询、长轮询、永久帧和 XHR 流。

8.2.1 轮询

Polling

由于每台服务器允许的最大并发连接数有限制（见第 11 章），所以在很多浏览器中连接很容易发生阻塞或死锁。开发者最先使用的解决连接限制问题的方案是简单**轮询**，即网站或应用每 x 毫秒发出一个请求来检查是否有更新需要呈现到用户界面上。一个非常简单的轮询例子可能像如下这样（注 7）：

```
setTimeout(function(){xhrRequest({"foo":"bar"})}, 2000);

function xhrRequest(data){
    var xhr = new XMLHttpRequest();
    // 在发送请求的时候，处理的数据通过参数进行传递
    xhr.open("get", "http://localhost/foo.php", true);
    xhr.onreadystatechange = function(){
        if(xhr.readyState == 4){
            // 处理服务器返回的更新
        }
    };
    xhr.send(null);
}
```

简单轮询是效率最低但最简单的 Comet 技术。

译注 7：HTML5 规范将提供 WebSocket，单 Socket 通道双向数据交流，用于实现 Comet。

注 7：为了简便起见，我们忽略了对旧版浏览器的额外处理和错误处理，但大多数的 JavaScript 库提供了 Ajax/XHR 请求函数，所以你的实际代码可能会不同。

8.2.2 长轮询
Long Polling

在服务端按已知间隔时间生成信息的情况下，轮询是可行的。例如，在股票行情程序中，当服务端每 5 秒更新一次价格时，这和浏览器上的轮询间隔时间相匹配，从而确保每个数据元素总有一个请求。否则，轮询会浪费 HTTP 请求，并且消耗宝贵的 CPU 时间和带宽。即使数据更新间隔时间是已知的，轮询还是会导致一些严重的问题——服务端超负荷。设想一下这种情况，当服务端还没有响应上一个数据请求时，第 2 个甚至第 3 个请求接踵而至，这些无用的额外请求对服务端进行狂轰烂炸。当然，你可以改变轮询间隔时间，在每次成功请求 5 秒之后再发送新请求，但我们有比轮询更好的 Comet 技术。

适用性更好的方法叫**长轮询**（Long Polling），该方法中浏览器发送一个请求到服务端，而服务端只在有可用的新数据时才响应。要支持长轮询，服务端要完全保持一个所有未响应请求和它们对应连接的大集合。服务端通过返回 Transfer-Encoding: chunked 或 Connection: close 响应来保持这些请求连接。当某个客户端或某组客户端的数据准备好时，服务端确认那些连接并给浏览器返回一个包含有效负载的响应。浏览器立即返回一个请求给服务端。如果连接断开，客户端将尝试重建和服务端之间的连接。虽然这样的请求/响应循环始于客户端，就像使用轮询一样，但所有数据流都依赖服务端而非客户端的时序产生，这接近完美的服务端→客户端的数据流。此外，服务端的超负荷不是大问题，因为客户端在服务端未真正响应之前不会发送额外的请求。在基于 Web 的聊天客户端 Meebo（译注 8）众所周知后，长轮询已经成为了主流技术。

实现典型的长轮询 Comet 技术会涉及使用 Comet 客户端，这通常是用 JavaScript 编写的，但也不完全是这样，同时 Comet 服务端几乎有各种语言编写的版本。那么，如何创建 Comet 客户端呢？让我们来看看一个基本的长轮询例子：

```
function longPoll(url, callback) {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            //发送另一个请求，重新连接服务端
            callback(xhr.responseText);
            xhr.open('GET', url, true);
            xhr.send(null);
        }
    }
    // 连接到服务端以打开一个请求
    xhr.open('POST', url, true);
    xhr.send(null);
}
```

译注 8：Meebo 是一个用 Ajax 编写的 web IM，它的系统和主要的即时通信软件（如：AOL、ICQ、Yahoo! Messenger、MSN Messenger、Jabber 及 Google Talk）相容，使用户无需安装即时通信软件就能与其他人保持联系。其官方网址是：*http://www.meebo.com/*。

在这个长轮询方法中，用给定的 URL 创建一个 `XMLHttpRequest` 后，我们定义了当 XHR 的 `readyState` 为 4 且数据返回时做什么。在这种情况下，我们打开一个新连接并将响应传给监听返回数据的回调函数。

从浏览器的角度来看，这个方法解决了其最常见的使用情况：保持有效的请求。它也避免了产生大量的无效请求，并让用户感到更好的即时性（注 8）。

可是，轮询和长轮询都给传统 Web 服务器带来了新问题，因为它没有优化处理海量的长连接（响应时间长、生命周期长），相反的是尽可能地对快速开启和关闭连接做了优化。例如，每台 Apache 服务器大约可以处理 10000 个并发连接，而一个运行良好的 Comet 服务器应该有能力处理超过 50 000 个并发长连接，这在处理实时应用时是非常可取的。幸运的是，现在大量可商用并开源的服务器已能处理这个问题。《Comet 成熟度指南》（译注 9）比较了目前市场上大量可选服务的质量。

还有一些其他优化轮询和长轮询的方法。比如 Meebo 实现了轮询和长轮询相结合的方式，让客户端和服务端相互配合缩减连接的最长时间，使其能容易地重建失效连接。另外还有称为智能轮询的技术，当未接收到数据时，轮询会减小请求的频率。比如，正常收到返回数据时可能每秒轮询一次，但当每次只能收到空响应时，可以按因子 1.5 来计算（比如 1 秒、1.5 秒、2.25 秒，等等）延迟下一次轮询。最后，如果打开一个长轮询连接的同时还需要创建一个 XHR，你通常可以退出这个长轮询的 XHR 来释放连接，一旦非 Comet XHR 完成时，即可重新建立长轮询连接。

如果你陷入了浏览器可用连接数限制或 Comet 服务端超负荷的烦恼时，选择一个合适的解决方案显得相当重要。

8.2.3 永久帧
Forever Frame

长轮询是当今最常用的技术，Comet 最初是从**永久帧**（forever-frame）技术演化过来的，就是打开一个隐藏的 iframe，请求一个基于 HTTP1.1 **块编码**的文档。块编码是为增量读取超大型文档而设计的，所以你可以把它看做一个不断增加内容的文档。下面是一个简单的永久帧例子：

注 8：*http://cometdaily.com/2007/11/06/comet-is-always-better-than-polling/*。
译注 9：Comet Daily 的 Comet Maturity Guide，网址：*http://cometdaily.com/maturity.html*。

```
function foreverFrame(url, callback) {
    var iframe = body.appendChild(document.createElement("iframe"));
    iframe.style.display = "none";
    iframe.src = url + "?callback=parent.foreverFrame.callback";
    this.callback = callback;
}
```

服务器会发送一系列信息到 iframe 中，就像下面这样：

```
<script>
parent.foreverFrame.callback("the first message");
</script>
<script>
parent.foreverFrame.callback("the second message");
</script>
```

一些浏览器需要 hack 来实现增量呈现，比如每个发送给父 Comet 客户端的数据都封装在一个函数调用的脚本代码块中，我们需要在数据后面添加一个
元素或几个字节的空白符。（在第 12 章会提到更多关于块编码和浏览器特例。）从文件大小的角度来看，为了防止 iframe 文档变得太大的一种优化的方法是，节点解析完成后，就从 iframe 文档中移除。

在 Internet Explorer 中使用永久帧技术就是噩梦的开始，因为有一个令人讨厌的用户体验问题：页面加载完成时发出类似点击的响声。Internet Explorer 把每一个块编码事件当作页面加载去执行。Gmail Talk 通过使用 htmlfile ActiveX 对象（*http://msdn2.microsoft.com/en-us/library/Aa752574.aspx*）很好地解决了这个问题，让永久帧成为了一个可行的方案。下面是一个针对 Internet Explorer 的代码片段：

```
function foreverFrame(url, callback){
    // http://cometdaily.com/2007/11/18/ie-activexhtmlfile-transport-part-ii/
    // 注意，没有用'var tunnel...'
    htmlfile = new ActiveXObject("htmlfile");
    htmlfile.open();
    htmlfile.write(
        "<html><script>" +
        "document.domain='" + document.domain + "';" +
        "</script></html>");
    htmlfile.close();
    var ifrDiv = tunnel.createElement("div");
    htmlfile.body.appendChild(ifrDiv);
    ifrDiv.innerHTML = "<iframe src='" + url + "'></iframe>";
    foreverFrame.callback = callback;
}
```

foreverFrame 函数中创建和打开 htmlfile 对象，然后将一个 HTML 文档写入 htmlfile 对象中，并设置 document.domain。这对于跨子域的 Comet 来说是必不可少的，当然还有一种比较常见的情况是 Comet 服务运行在和其他普通 Web 服务不同的端口上。接着在 htmlfile 的 body 中创建了一个 iframe，这个 iframe 就可以当作 Comet 连接来使用了。使

用该技术，Internet Explorer 不再产生点击事件和伴随它的音效。垃圾回收机制并不能清除和释放 Comet 连接，所以必须要用一个 onunload 函数去删除 htmlfile 对象的引用，并且要显式地执行垃圾回收：

```
function foreverFrameClose() {
    htmlfile = null;
    CollectGarbage();
}
```

8.2.4 XHR 流
XHR Streaming

与服务器通信的最简化 API 是通过 XMLHttpRequest 来实现的，这是由于它提供了对响应正文和响应头的直接访问，这也是轮询和长轮询常用的传输方式。一些浏览器已经支持了 XHR 流，包括 Firefox、Safari 和 Chrome。类似于 forever-frame 技术，XHR 流允许服务器连续地发送消息，无需每次响应后再去建立一个新请求。

尽管 Internet Explorer 7 和更早的版本缺乏流的支持导致不能完全依赖基于流的协议，但当流可用时，我们显然可以利用流来提高性能。一旦 XHR 流可用时，它将是目前浏览器上最高效的 Comet 传输方式，因为它没有 iframe 或 script 标签（forever-frame 就是这样做的）的消耗，并且能够连续地使用一个单一的 HTTP 响应（长轮询不是这样的）。不幸的是 Internet Explorer 浏览器并不支持它，但 XHR 依旧是一个有价值的渐进增强方式。用户升级浏览器就可以马上享受到提高性能的好处。

虽然 XHR 流也是通过一个标准的 XMLHttpRequest 获得的，但你能够在 onreadystatechange 事件中 readyState 为 3 时去访问服务器发送的数据（在响应完成之前；响应完成时 readyState 为 4），这样，无需等待连接关闭就能够操作数据。

```
function xhrStreaming(url, callback){
    xhr = new XMLHttpRequest();
    xhr.open('POST', url, true);
    var lastSize;
    xhr.onreadystatechange = function(){
        var newTextReceived;
        if(xhr.readyState > 2){
            //获取最新的响应正文
            newTextReceived =
             xhr.responseText.substring(lastSize);
            lastSize = xhr.responseText.length;
            callback(newTextReceived);
        }
        if(xhr.readyState == 4){
            //如果响应结束，马上创建一个新的请求
            xhrStreaming(url, callback);
```

```
            }
        }
        xhr.send(null);
    }
```

XHR 流无疑为提高网络利用率及降低服务端和客户端资源消耗开启了一扇大门，但你应该意识到在某些情况下流模式反而会影响服务端效率。当使用长轮询时，一些服务端会延迟套接字缓冲区的占用直到响应准备就绪，并且由于响应在一发送时就完成，所以几乎能立即清空缓冲区。而在流模式下，缓冲区建立之后必须维持于整个连接生命周期中。当然，关键的问题是如何优化服务端，毕竟不同服务端的执行方式不同。

从客户端来看，XHR 流会引起一些潜在的性能问题。如果流的响应连接时间太长，浏览器会占用过多的内存。如果单个响应却返回数千个成功信息，会让 Firefox 瘫痪。通过每 100 条信息（或限定字节，如 50KB）后关闭响应就能很容易地修复这个问题，然后再创建一个全新的请求（与长轮询处理每个信息一样）来继续后面的消息。

在使用 XHR 流时需增加一个额外的操作，即对消息进行划分。浏览器从服务器接收一个正文的流，但必须将其划分成一个个独立的信息。FireFox 支持一种特殊的内容类型，multipart/x-mixed-replace，你可以用它来分割流的消息（注 9）。虽然这没有被广泛支持，但正如它所示，你可以写一个 JavaScript 解析器去解析出单条消息，这事实上比 Firefox 的多部件处理程序（multipart handler）更快。

8.2.5 传输方式的前景

Future Transports

目前 HTML5 团队正在完成 WebSocket（注 10），它将提供一个 Web 安全的 TCP **套接字**，使得从客户端到服务端的通信方式大大简化。如果浏览器厂商能高效广泛采用 WebSocket，那么它或许会取代所有形式的 Comet 连接技术。

如果主流浏览器采用 WebSocket，并实现预期的性能，它将迅速地取代该领域中其他方式的 Comet 连接技术。

8.3 跨域

Cross-Domain

值得一提的是，如果浏览器不支持跨域 XHR，长轮询也同样不支持跨域请求，但至少可以通过永久帧技术实现跨子域。当然我们也可以通过其他的一些临时性解决方案来实现跨子

注 9：*http://cometdaily.com/2008/01/17/proposal-for-native-comet-support-for-browsers/*。
注 10：*http://cometdaily.com/2008/07/04/html5-websocket/*。

域 XHR，比如 Abe Fettig（注 11）就实现了一套方案，或者干脆在那些支持跨域 XHR 的现代浏览器中调用，或者使用 HTML 5 中引入的 `postMessage`（注 12）。

通常情况下，与包含或插入文档中的 iframe 和 script 相比，XHR 的安全模型会有更多的限制。所以我们还有另外一种选择，通常被称作**回调轮询**（callback polling）或 JSONP **轮询**（JSONP polling）。通过这种方式，可以为每个请求都插入一个 script，从而实现跨域轮询，而不用依赖于 XHR。这项技术依赖于 JSONP（注 13），因为 JSONP 可以在不同的域之间建立一套隐式的信任机制。由于 JSONP 只是简单地将服务器返回的数据封装起来，并当作参数传递到用户提供的函数中，所以使用 JSONP 不代表百分之百安全，它的安全级别和向第三方域的页面中插入一个 script 引用相当。

JSONP 将返回的数据封装在可执行的脚本中，函数名由向服务器发送请求的`<script>`代码块指定，而不是通过 XHR。支持跨域的 Comet 非常重要，有以下几个理由：对不同域名进行请求可以突破浏览器最多同时打开**两个连接**的限制（注 14），可以用于从第三方服务中检索数据，还有就是 Comet 和 HTTP 可以运行在不同的服务器上，这样我们就可以针对 Comet 服务器和 HTTP 服务器进行不同的优化（用传统的服务器运行 Comet 通常无法达到最佳状态，反之亦然）。

下面的例子演示了这项技术的使用场景，通过隐式的信任机制，可以实现跨域获取数据。

```
function callbackPolling(url, callback){
    // 建立一个script元素，在这个元素中将加载服务器的响应
    var script = document.createElement("script");
    script.type = "text/javascript";
    script.src = url + "callback=callbackPolling.callback";
    callbackPolling.callback = function(data){
        // 发送一个新的请求等待服务器下次发送的消息
        callbackPolling(url, callback);
        // 调用回调函数
        callback(data);
    };
    // 添加这个元素，开始执行这段脚本
    document.getElementsByTagName("head")[0].appendChild(script);
}
```

注 11：*http://www.fettig.net/weblog/2005/11/30/xmlhttprequest-subdomain-update/*。
注 12：*http://www.whatwg.org/specs/web-apps/current-work/#crossDocumentMessages*。
注 13：*http://ajaxian.com/archives/jsonp-json-with-padding*。
注 14：主要是 Internet Explorer 6 和 7 有这个问题。浏览器的连接限制将在第 11 章进行讲解。

需要注意的是，在 Firefox 浏览器中，对于任何页面，连续的脚本总是按顺序执行的。所以，如果使用这项技术，在等待服务器响应的过程中，若你想在同一页面/frame 中调用另外一个 JSONP 请求，在第一个脚本执行完成之前，将不会得到任何响应，因为第一个脚本正在等待服务器的响应，所以这个时间是不确定的。为了解决这种问题，你可以建立独立的 frame 来处理不同的 JSONP 请求。在每个独立的 frame 中只有一个独立的请求，这样每个请求的响应一到达就可以并行处理了。

8.4 在应用程序上的执行效果

Effects of Implementation on Applications

客户端 Comet 的性能优化目的是：减少数据传输的延迟、HTTP 连接的保存和管理、远程消息和处理跨域问题。服务器端的性能优化目的是：保存和共享 HTTP 的连接数，并尽量减少每个连接所消耗的内存、CPU、I/O 和带宽。

8.4.1 连接管理

Managing Connections

即便是很少的数据，服务器也会永久地为每一个用户打开一个 HTTP 连接，这导致打开很多连接。连接有两个限制：内存和 CPU。但无论如何，每个连接都会因操作系统或开发语言而产生内存占用（memory overhead）。如果每个连接使用一个线程（或进程）就会导致一个完整执行堆栈的内存占用，即使它能被降到几乎合理的程度，通常大小也为 2MB。另外，如果线程数的增加超过进程数，那么我们将会以**超负荷**（thrashing）而告终，因为操作系统在线程与进程之间切换的开销比我们实际执行代码的开销还要多。正是这个原因，我们需要选择一个异步网络架构。

选择或轮询会导致如下问题：这些方法会使操作系统检查每个打开的套接字，最终来确定哪一个已经准备好了。这意味着调用选择时，即使显示没有套接字准备读出，那么在套接字数量很少时开销也小，但是随着套接字数量的增加，CPU 的占用时间也会随之增加。我们可以通过使用替代技术来避免套接字检查，如 FreeBSD/OS X 的 kqueue、Linux 的 epoll 和 Windows 的 completion ports（完成端口（译注 10））。在大多数主流语言中有一些网络库把这些细节封装成了一致且跨平台的 API，例如有 C 的 libevent、java.nio（*http://java.sun.com/j2se/1.4.2/docs/guide/nio/*）和 Twisted Python（*http://twistedmatrix.com/trac/*）。

性能优化技术在不同情况下也大相径庭。例如聊天程序，通常有许多用户连接，但是在任何一个特定时间内只有一小部分用户在接收信息。在这种情况下，通过服务器端的连接共

译注 10：完成端口是一种被公认为在 Windows 服务平台上比较成熟和高效的 I/O 方法，利用完成端口进行重叠 I/O 的技术在 WindowsNT 和 WIndows2000 上提供了真正的可扩展性。完成端口和 Windows Socket2.0 结合可以开发出支持大量连接的网络服务程序。

享管理大量闲置的连接是很有好处的。Orbited（*http://orbited.org*）和 Willow Chat（*http://willowchat.org*）这两个网站在这方面做了深入的优化。

另外一些例子中，比如实时股票报价监控应用程序，大量的连接不断地更新，很少有闲置连接的存在。Jetty、Lightstreamer 和 Liberator 对这种情况做了优化。

8.4.2 测量性能
Measuring Performance

本章的许多地方都提到了测量 Comet 的性能。通过测试可以找出建立一个能容纳 100 万用户的 Comet 服务器需要多少资源（注 15）。要实现这种规模的简单原则是：让每个连接使用尽可能少的系统资源，并编写完善的测试（注 16）去测量性能。

服务器必须尽量削减对资源的占用，同时基于发送给每个客户端的数据的频率和负载量来优化长期保存的（long-held）连接数。大负载和过于频繁的数据发送增加了延迟，减少了 Comet 服务器最大可用连接数。

因为 Comet 实际上只是 HTTP 的性能优化和连接管理，性能测量技术实际上与测量任意一个大规模的 Web 应用程序很类似。

8.4.3 协议
Protocols

Comet 连接和常见普通连接的通信方式不同。Comet 的连接只允许 server → client 通信或双向通信。各种协议架构于连接之上，提供了更多的功能和更好的语义化，而不仅仅只是“读”和“写”，比如 Bayeux 的 Publish-Subscribe（PubSub）模型。

Bayeux（*http://svn.cometd.com/trunk/bayeux/bayeux.html*）（注 17）是一种用来在客户端和服务端传输低延迟的异步消息（主要通过 http）的协议，是 Dojo 基金会提供支持的 cometD 项目（*http://cometd.com*）（注 18）的一部分。它有一个简单的可扩展的协议，对于不同 Comet 服务端和客户端之间的交互是非常有利的。

PubSub 是 Bayeux 协议里一个常用的模型，同其他协议如 XMPP 一起常用于聊天应用程序。

注 15：*http://www.metabrew.com/article/a-million-user-comet-application-with-mochiweb-part-3/*。

注 16：*http://aleccolocco.blogspot.com/2008/10/gazillion-user-comet-server-with.html*。

注 17：Bayeux 还是一条挂毯的名字——贝叶挂毯，描述了 1066 年诺曼底人对英格兰的入侵，其中还包括了哈雷彗星（Halley's Comet）（译注 11），象征着即将来临的灾难。

译注 11：当著名的哈雷彗星在 1066 年出现时，正是法国诺曼底公爵威廉率兵准备入侵英国的时候，后来一举获胜，建立了诺曼底王朝，威廉公爵夫人为了纪念这次胜利，将当时的情景编织在一幅挂毯上，图中的一方是一群诺曼底人指着彗星露出胜利的微笑，另一方则是英国的哈学德国王坐在王位上望着头上彗星，惊恐万状。

注 18：D 是 daemon 的简写，daemon 的意思是一段连续运行的程序，类似 httpd 中的 d。

Dojo Toolkit 在他的 cometD 模块中提供了长轮询和回调轮询（包括用多个 frame 并行请求 JSONP）的全部传输层处理，同时还能处理与 Bayeux 的交互和通信。因此基于兼容 Bayeux 的服务端，你可以很容易地使用 Dojo 来实例化 cometD 模块并让它来处理传输细节：

```
dojox.cometd.init("/cometd");
dojox.cometd.subscribe("/some/topic", function(message){
    // 回调函数
});
```

8.5 总结

Summary

与 Ajax 不同的是，Comet 是一套复杂的性能优化技术，影响客户端、服务器端，以及两者之间相互通信。虽然现在还处于寻找合适的 Comet 问题解决方案的初期，但随着 WebSocket 的兴起和处理百万级用户量的解决方案的出现，其复杂性会随着时间逐渐降低。

第 9 章

超越 Gzip 压缩

Going Beyond Gzipping

Tony Gentilcore

在为网页提速的技术中，除了合理配置 HTTP 缓存头以外，启用 Gzip 压缩是公认最重要的一项技术。在 Steve Souder 的第一本书《高性能网站建设指南》中，我拿出一整章来介绍压缩技术。现在所有的浏览器都已经支持 Gzip 压缩，而且所有负责任的 Web 开发者都已经启用了这项功能，那么这章就算讲完了吗？当然不是！

即使已经启用了 Gzip 压缩功能，但还是存在这样的可能性：有相当一部分用户在访问你的网站时，收到了未经压缩的响应。这部分人的实际比例会因人口分布和地理位置的差异有所不同，但对于一个大型的美国网站来说，预计大概有 15%的访问者没有声明支持 Gzip 压缩。本章将解释为什么这个比例比预想的要高，这种情况对性能有哪些影响，以及开发者的应对措施。

9.1 这为什么很重要

Why Does This Matter?

对于这个很小的比例，你可能会问“这有什么大不了的？”让我们来看看最流行的 10 个网站如果关闭了 Gzip 压缩功能会怎么样。

在这个实验中，10 个流行网站的页面加载时间（注 1）都是在 Windows XP Pro 中的 Internet Explorer 7.0 上测量的，每个网站加载 100 次后取其平均值。在实验的过程中，缓存始终打开，用于重现最典型的操作。所有的请求都在同一台计算机上经过了相同的代理（Eric Lawrence 开发的 Fiddler（注 2））。在对照组中，代理什么都不做，但是在实验组中，代理去掉了所有请求的 `Accept-Encoding` HTTP 头，这样压缩功能就无效了。表 9-1 显示了压缩失效时页面加载时间增长的绝对值和百分比。

注 1：页面加载时间是指 `OnBeforeNavigate2` 和 `OnDocumentComplete` 两个事件之间的时间。
注 2：*http://www.fiddlertool.com/*。

表 9-1：压缩失效导致页面加载时间增长

网站	增加的总下载量（第一次加载）	增加的页面加载时间（1000/384 Kbps DSL）	增加的页面加载时间（56 Kbps modem）
http://www.google.com	10.3 KB (44%)	0.12s (12%)	1.3s (25%)
http://www.yahoo.com	331 KB (126%)	1.2s (64%)	9.4s (137%)
http://www.myspace.com	441 KB (143%)	8.7s (243%)	42s (326%)
http://www.youtube.com	236 KB (151%)	3.3s (56%)	21s (87%)
http://www.facebook.com	348 KB (175%)	9.4s (414%)	63s (524%)
http://www.live.com	41.9 KB (41%)	0.83s (53%)	9.2s (99%)
http://www.msn.com	195 KB (77%)	1.6s (32%)	13s (85%)
http://www.ebay.com	245 KB (92%)	1.7s (59%)	3.5s (67%)
http://en.wikipedia.org	125 KB (51%)	5.0s (146%)	21s (214%)
http://www.aol.com	715 KB (111%)	7.4s (47%)	32s (60%)
平均值	**269 KB (109%)**	**3.9s (91%)**	**22s (140%)**

第一次加载页面时，缓存为空，由于压缩功能失效，所以需要加载的所有资源的量是原来的两倍多。注意这个数字并不能体现 Gzip 的压缩率，因为它代表所有资源的下载总量，包括图像和 Flash。Gzip 压缩通常只对文本类资源有效，比如 HTML、CSS 和 JavaScript 文件。

对于使用 DSL 的用户，页面平均加载时间从 4.3 秒增加到 8.3 秒，增幅达 91%。拨号用户的情况更糟，页面的平均加载时间从 15 秒增加到 37 秒，增幅高达 140%。

有了手中这些数据之后，我们就可以回到开始的问题：“是否应该关注这些压缩功能失效的用户？”你也许会天真地通过计算所有请求的平均效果来回答这个问题：15%的用户降低 91%的访问速度，相当于所有的请求降低 14%的访问速度。如果页面的平均加载时间是 4 秒，那么这就意味着用户访问时平均要多花掉 560 毫秒。你也许会认为任何人都不会因为多出的半秒时间而选择离开，所以为什么要在意呢？

这只是我们看到的平均影响，并不能反映实际情况。实际上，85%的用户是不受影响的，但是确实有 15%的用户受到了非常大的影响，额外增加的 4 秒足以让用户放弃你的网站。所以，了解用户浏览到未经压缩内容的原因，以及开发者可以采取哪些应对措施，是非常重要的。

9.2 问题的根源

What Causes This?

现在你应该已经意识到这是一个确实存在的问题，并且会影响到真正的用户，那么下一个合理的步骤应该就是找出问题的原因，这样才有机会去修复它。

9.2.1 快速回顾

Quick Review

首先让我们回顾压缩的工作原理。所有的现代浏览器（从 1998 年左右诞生的 4.x 那代起）都支持 Gzip 压缩，并可以通过添加 `Accept-Encoding` HTTP 头来向 Web 服务器声明支持压缩：

```
Accept-Encoding: gzip, deflate
```

当请求中包含这个头时，Web 服务器就会开启 Gzip 压缩功能。按照 RFC 2616 中第 14.3 小节（注 3）的定义，Web 服务器会返回一个经过压缩的响应，并且用 `Content-Encoding` 头做标记：

```
Content-Encoding: gzip
```

9.2.2 罪魁祸首

The Culprit

如果所有的现代浏览器都发送了 `Accept-Encoding` 头，那为什么还会有 15%的响应没有经过压缩呢？当然我们得先说明，这 15%的用户肯定没使用 10 年前的浏览器。通过对 Web 服务器日志的大量分析，我们找到了罪魁祸首的一丝线索，原来部分请求的 `Accept-Encoding` 头是损坏的：

```
Accept-EncodXng: gzip, deflate
X-cept-Encoding: gzip, deflate
XXXXXXXXXXXXXX: XXXXXXXXXXXXX
---------------: -------------
~~~~~~~~~~~~~~~: ~~~~~~~~~~~~~
```

但是这些损坏的头并不能说明为什么会有大量的请求收到未经压缩的响应，因为通过分析发现，更多由真实用户通过现代浏览器（不是网络蜘蛛）发出的请求中根本没有 `Accept-Encoding` 头。为什么会有一些人或软件故意关闭压缩功能来降低用户的浏览体验呢？罪魁祸首主要有两类：Web 代理和 PC 安全软件。

它们有什么共同之处呢？它们都需要对 Web 服务器发送的响应进行监听（如果你愿意，也可以叫做暗中窥探）。与经过压缩的响应相比，监听未经过压缩的响应会占用更少的 CPU 资源。不幸的是，这样做完全忽略了实际情况，从最终用户的角度来看，增加的网络响应时间已经远远超过监听程序对响应进行解压缩所消耗的 CPU 时间（注 4）。由于这个原因，

注 3：*http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.3*。

注 4：尤其需要注意的是，解压缩通常要比 Gzip 压缩快 3~4 倍。

我喜欢将这种为了监听响应而移除 Accept-Encoding 头的技术称为乌龟窃听（turtle tapping）。乌龟窃听就是乌龟执行窃听的方式：肯定会非常慢。

9.2.3 流行的乌龟窃听者实例
Examples of Popular Turtle Tappers

表 9-2 列出了几个流行的客户端软件和 Web 代理，以及它们是如何修改客户端的 Accept-Encoding 请求头的（注 5）。这个列表并不全面，比如流行的 Web 代理 Squid 就有很多扩展，它们都会通过移除或修改头来过滤或监听 Web 内容。

表 9-2：会对 Accept-Encoding 头进行修改的软件

软件	对 Aceept-Encoding 的修改
Ad Muncher	移除
CA Internet Security Suite	Accept-EncodXng: gzip、deflate
CEQURUX	移除
Citrix Application Firewall	移除
ISA 2006	移除
McAfee Internet Security 6.0	XXXXXXXXXXXXXXX: ++++++++++++
Norton Internet Security 2005	----------: ----------
Novell iChain 2.3	移除
Novell Client Firewall	移除
WebWasher	移除
ZoneAlarm Pro 5.5	XXXXXXXXXXXXXXX: XXXXXXXXXXXXX

按地区来查看乌龟窃听的比例也很有趣：从某些中东国家发出的请求，大部分都缺少有效的 Accept-Encoding，这可能和国家级防火墙有关；在美国和俄罗斯，有超过 20%的用户会遭遇这种问题；欧盟和亚洲国家在这个问题上处理得不错，只有不到 10%的用户会遇到。

9.3 如何帮助这些用户
How to Help These Users?

现在已经知道了这个问题发生的原因和带来的后果，那么开始进入正题：如何帮助这些用户获取他们期望的高速体验（更不用说可以帮助你的网站从那些满意的用户身上获取更多的收益）。当然，处理这个问题的正确解决方案应该是向那些移除或修改了 Accept-Encoding

注 5：不同版本的行为有可能不一样，而且可能也允许通过修改默认设置来改变其行为。

头的软件开发商发出呼吁。实际上，在一些软件的最新版本中已经修复了这个问题。比如Norton Internet Security 2009就没有这个问题。

但是，让这些用户升级或更换软件是需要一定时间的，在那之前，我们还是要在本章讨论缓解这类问题的3个方法，每个方法都存在不同程度的侵入性。

9.3.1 设计目标：最小化未压缩文件的尺寸
Design to Minimize Uncompressed Size

这点貌似太明显而不值一提，但实际上被强调得还不够：**发送更小的响应会使页面加载速度更快**。这也是为什么对响应进行压缩是如此有效的技术，尽管这样做会在服务器端和客户端增加CPU的开销。优秀的Web开发者会尽全力让HTML、CSS和JavaScript变得尽可能精简，但是我们都会根据经验任由那些又长又重复的字符串存在，因为Gzip压缩会使它们基本上消失。因此，我们实际上没有在这上面做出足够的优化，当用户无法接收经过压缩的响应时，这种设想就毫无意义了。

寻找那些可以分离出来的重复内容是一门艺术，而且每个网站的实际情况不同。有一些通用的技术可以减小页面未经压缩的大小，同时又不会增加压缩后的大小。

使用事件委托

页面上通常会有几个元素需要绑定相似的事件处理程序，普遍出现在10大网站中的例子是下拉列表、跟踪链接点击和hover动画。分别设置事件处理程序会导致页面大小迅速增加(注6)。

比如说，写本书时，*http://facebook.com*包含了一个下拉列表，里面大概包含50种语言的选项，每种语言都对应一个链接，给每个链接绑定`onclick`事件处理程序都需要额外的133个未经压缩的字节。

```
<a href="http://es-la.facebook.com/" onclick="return wait_for_load(this, event,
function() { intl_set_cookie_locale("", "es_LA"); return false;
});">Español</a>
```

将50个链接浪费的时间累加起来，相当于把约6.7KB的数据传送给不支持压缩的用户所浪费的时间。每个链接中都有一些非常明显的重复信息，比如URL、本地代码和语言名称。

当多个元素都需要响应某个事件时，我们把这个事件的处理程序绑定到它们的父元素上，这项技术通常叫做**事件委托**（event delegation）。当事件在子元素上触发后，会冒泡到绑定

注6：对于交互操作比较多的页面，减少事件处理程序的数量可以换取在JavaScript执行时间上的性能明显提升。

了处理程序的父元素上。事件处理函数可以识别出哪个子元素是发生事件的元素，并通过它的一些属性来获取额外的参数。

比如说，为了改善 facebook.com 上的那个例子，我们可以把链接进行如下修改：

```
<div class="menu_content" onclick="return intl_set_cookie_locale(event)">
  ...
  <a href="http://es-la.facebook.com/" class="es_LA">Español</a>
  ...
</div>
```

新的事件委托处理程序通过元素的 class 属性来获取地区值，之前是通过参数传递进来的：

```
<script>
    function intl_set_cookie_locale(e) {
        e = e || window.event; //获取 event 对象
        var targetElement = e.target || e.srcElement; //获取触发事件的元素
        var newLocale = targetElement.class; //获取新的地区值
        ...
        //使用 newLocale 变量去设置 Cookie
        ...
        return false; //阻止链接的默认行为
    }
</script>
```

为实现事件委托而增加的一小段代码在数量上是微不足道的，因为它可以放置在外部文件中。用可以缓存的 JavaScript 文件取代嵌入文档中的代码，这样用户就无需每次访问都重新下载了。

使用相对 URL

我们都对相对 URL（比如用/index.html 代替 *http://www.example.com/index.html*）非常熟悉。但 RFC 1808 规范（注 7）描述了几种鲜为人知的使 URL 相对化的方式，大概从 1995 年开始几乎所有的浏览器都支持它们。例如，除了 slashdot.org 这个引人注目的例外，几乎没有一个主流网站会使用协议的相对 URL（比如用*//www.example.com* 代替 *http://www.example.com*）。考虑到典型页面中 URL 的数量，这些非相对 URL 会明显增加页面尺寸。如果以 *http://www.example.com/path/page.html* 这个 URL 作为基准页面，我们可以使用表 9-3 中显示的相对 URL。

表 9-3：http://www.example.com/path/page.html 的相对等效值

指定目标的完整 URL	相对等效值
http://subdomain.example.com/	*//subdomain.example.com*
http://www.example.com/path/page2.html	*page2.html*

注 7：*http://www.w3.org/Addressing/rfc1808.txt*。

续表

指定目标的完整 URL	相对等效值
http://www.example.com/index.html	*/index.html*
http://www.example.com/path2/page.html	*../path2/page.html*
http://www.example.com/path/page.html#f=bar	*#f=bar*
http://www.example.com/path/page.html?q=foo	*?q=foo*

对于动态生成的 URL，虽然我们可以写个函数，以包含页面的 URL 为基准，尽可能地相对化，但是这完全没必要。

移除空白

用户不会在意代码可读性有多好，但是他们会相当在意网站有多快。换行符和适当的缩进对于开发者来说很重要，但是在展示给用户之前，应该确保它们总是被自动移除的。就像在《高性能网站建设指南》中讨论的一样，现在有很多工具来对 JavaScript 进行这样的优化。其中最流行的包括 YUI Compressor（*http://developer.yahoo.com/yui/compressor/*）、ShrinkSafe（*http://shrinksafe.dojotoolkit.org/*）和 JSMin（*http://www.crockford.com/javascript/jsmin.html*）。对于 CSS 来说，YUI Compressor 已经做到最好了。对于 HTML 来说，问题可能比较棘手，因为在很多上下文中的空白是有实际意义的。但是，如果你愿意标明那些需要空白字符的地方，那么大部分主流的模板语言都有相应的选项，可以像移除换行符一样移除前导空白字符和后置空白字符。

移除属性的引号

在讨论如何移除包含 HTML 属性值的引号之前，首先要注意两种不应该移除引号的情况。第一，如果网页是依据 XHTML 规则来编写的，那么属性值必须用引号包含。第二，为了避免属性值从不包含引号到包含引号这种意外情况所带来的 Bug，HTML 的属性值应该始终被包含在双引号内。不管怎样，按照 HTML 4.01 规范第 3.2.2 小节（注 8）所定义的，当属性中只包含字母、数字、连字符、句号、下划线和冒号（可以用正则表达式 `[a-zA-Z0-9\-._:]` 匹配）时，包含属性值的引号是可以省略的。

为了改善下载时间，在网页展示给用户之前，通过一些自动化工具移除不必要的引号是很有益处的。

避免行内样式

通过行内的方式定义一些重复样式，而不是使用外部 CSS，也会增加未压缩页面的大小。比如说，在写本文时，wikipedia.org 在主 HTML 文档中包含了 4KB 重复的行内样式。这些

注 8：*http://www.w3.org/TR/html4/intro/sgmltut.html#h-3.2.2*。

内容会被 Gzip 非常高效地压缩，但是当压缩功能失效时，下载的数据量会有非常明显的增加。

为 JavaScript 变量设置别名

很不幸，JavaScript 中几个很常用的 DOM 方法名都很长，压缩的时候可以完全避免这种浪费。但是在未压缩的情况下，它们是很耗费带宽的。幸运的是，JavaScript 允许我们为那些很长的名字建立引用（或别名），这个问题从而得以解决。

首先找到那些在脚本中被频繁使用的函数，并为其设置别名，比如，一些流行的 JavaScript 库使用变量$来作为 `document.getElementById` 的别名：

```
var $ = document.getElementById;
```

在脚本中，你可以简单地用`$("foo")`来代替以前的写法 `document.getElementById("foo")`，这就为每次使用都节省了 22 个未经压缩的字节。通常，为使用超过 3 次的方法设置别名是十分明智的。

其次，当访问对象的链式属性时使用别名也非常有益（注 9）。这种情况通过例子说明会更直观一些：

```
// 浪费的写法
var foo = $("foo");
foo.style.left = "0";
foo.style.right = "0";
foo.style.height = "10px";
foo.style.width = "10px";

// 更好的写法
var foo = $("foo").style;
foo.left = "0";
foo.right = "0";
foo.height = "10px";
foo.width = "10px";
```

现实世界的节省

这些技术究竟有什么作用？通过对同一组流行网站分别应用每项技术，我们可以获得未经压缩的页面所减小的比例，正如表 9-4 所示（注 10）。

注 9：在循环中为对象的链式属性使用别名也能显著提升 JavaScript 的执行性能。

注 10：JavaScript 的别名没有切实可行的测试方法。

表 9-4：流行网站上页面所减小的比例

网站	事件委托	相对 URL	移除空格	移除引号	使用 CSS	总计
http://www.google.com	1.8%	3.4%	--	--	0.4%	5.6%
http://www.yahoo.com	--	0.8%	3.3%	0.6%	0.5%	5.2%
http://www.myspace.com	4.0%	2.2%	9.0%	1.5%	1.8%	18.5%
http://www.youtube.com	8.3%	0.6%	7.1%	2.3%	1.2%	19.5%
http://www.facebook.com	12.9%	1.7%	1.1%	2.6%	0.3%	18.6%
http://www.live.com	8.5%	0.9%	0.2%	0.9%	0.3%	10.8%
http://www.msn.com	--	3.0%	0.1%	1.7%	--	4.8%
http://www.ebay.com	0.2%	1.7%	1.2%	1.6%	1.2%	5.9%
http://en.wikipedia.org	--	1.6%	2.1%	1.8%	5.2%	10.7%
http://www.aol.com	10.4%	2.4%	1.4%	1.8%	0.5%	16.5%
平均值	**4.6%**	**2.8%**	**2.6%**	**1.5%**	**1.1%**	**11.6%**

当应用了所有这些技术以后，每个网站都会减小 5%~20%未经压缩的尺寸（平均是 11.6%）。使用相对 URL、移除空白和移除属性引号这些操作都可以通过程序自动完成，所以这些优化物有所值。此外，使用事件委托、JavaScript 别名和避免使用行内样式可以带来更容易维护的代码，也能让页面在浏览器中跑得更快。

然而，和 Gzip 压缩可以将相同的文档平均减小 72.1%相比，减小 11.6%就显得黯然失色了。回到最开始的时间测试上，如果应用 Gzip 压缩可以减小页面 72.1%的大小和提速 3.9 秒，那么我们预计减小页面 11.6%的大小可以提速 630 毫秒。

部分用户发出的请求中缺少有效的 Accept-Encoding，以至于收到未经压缩的响应，为了解决这些用户的问题，我们被迫用之前提到的这些变通方法，来减小页面未经压缩的大小。

9.3.2 引导用户
Educate Users

当页面在未经压缩时的尺寸足够小后，另外一个不错的解决方案就是向乌龟窃听的受害者通知这种情况的存在。在 Web 上，这种类型的提示信息优先级很高，比如对于那些在 Firefox 上使用 Firebug 扩展的用户，Gmail 会在页面的最顶端显示一个很亮的红色对话框，提示“配置不当的 Firebug 会使 Gmail 变慢”。用 Internet Explorer 6.0 访问 Gmail 的用户，则会在操作了一会儿后看到升级浏览器的提示。

当请求中没有包含有效的 Accept-Encoding 头时，我们就可以使用这项技术。类似下面这种的提示信息，能帮助用户纠正这类问题：

```
未启用压缩功能，访问速度有可能变慢。
    修正它          隐藏
```

“修正它”链接指向一个解释页面，说明如何禁用或升级那些导致这种情况的软件。“隐藏”链接则会设置一个 Cookie，这样就不会再次看到这条消息。

不幸的是，这个解决方案并非十全十美。由于使用代理而导致压缩功能无效的用户，通常都没有权限来修改代理，只能抱怨一下管理员而已。还有一个策略可以帮助到这些用户，我们将在下面介绍。

9.3.3 对 Gzip 的支持进行直接探测

Direct Detection of Gzip Support

当所有这些尝试都失败后，如果未经压缩的响应仍然是你网站的一个痛，还有一个可能有效的游击战术：直接测试是否支持压缩，而不是依靠 Accept-Encoding 头。乍一听这可能很危险，但是如果经过了必要的测试，它也可以变得很安全。正确进行判断是非常重要的，因为谁也不想冒判断错误的风险。预计在使用直接探测以后，大概有一半左右未经压缩的响应可以得到压缩。

执行测试

如果在请求中没有 Accept-Encoding 头，则会自动在页面<body>的末尾加入一个隐藏的 iframe：

```
<iframe src="/test_gzip.html" style="display:none"></iframe>
```

它将加载 test_gzip.html 文档，在这个文档中你需要设置如下内容：

1. 禁用缓存，这样所有的连接都会被测试到。
2. 无论请求头包含什么内容，都对内容进行压缩。
3. 使用 JavaScript 建立一个 session-only 的 cookie，标记浏览器支持 Gzip 压缩。

如果客户端支持压缩，随后的请求都将包含一个 Cookie 来标记这种情况。如果客户端的确不支持压缩功能，隐藏的 iframe 将会加载不可见的乱码，同时也不会设置名为 supports_gzip 的 Cookie。

有很多种方式可以完成这个操作，下面是一个使用 PHP 编写的例子：

```
<?php
  function flush_gzip() {
    $contents = ob_get_contents();
    ob_end_clean();
    header('Content-Type: text/html');
```

```
    header('Content-Encoding: gzip');
    header('Cache-Control: no-cache');
    header('Expires: -1');
    print("\x1f\x8b\x08\x00\x00\x00\x00\x00");
    $size = strlen($contents);
    $contents = gzcompress($contents, 9);
    $contents = substr($contents, 0, $size);
    print($contents);
  }

  ob_start();
  ob_implicit_flush(0);
?>

<html>
  <body>
    <script>
      document.cookie="supports_gzip=1";
    </script>
  </body>
</html>

<?php
  flush_gzip();
?>
```

利用测试结果

如果存在名为 `supports_gzip` 的 Cookie，那么随后拥有相同 `Content-Type` 的页面都将被压缩。当对存在 Cookie 的页面进行强制压缩时，确保响应不是公开缓存的，并且不会再次输出 src 值是 `test_gzip.html` 的 iframe。

同样，具体的实现会根据实际环境而有所不同。下面是一个基于 PHP 的例子，使用了之前定义的 `flush_gzip` 方法：

```
<?php
  // 为了简洁，此处省略 flush_gzip()的定义

  ob_start();
  ob_implicit_flush(0);
?>

<html>
  <!-- 此处是页面的具体内容 -->
</html>

<?php
  if (isset($_COOKIE["supports_gzip"])) {
    flush_gzip();
  } else {
    flush();
  }
?>
```

对效果进行量化

如果正在考虑或已经开始对 Gzip 压缩的支持进行直接探测，那么需要持续关注两个重要的统计。一是那些没有通过 `Accept-Encoding` 头声明支持压缩的请求的比例，如果这个比例太低，直接探测压缩支持的技术将变得没有意义。二是那些没有 `Accept-Encoding` 头，但实际上支持压缩的请求比例。这个比例只能通过直接探测来测得。只有这个比例保持在一个很高的数值时，直接探测才有必要持续使用。

第 10 章

图像优化

Optimizing Images

Stoyan Stefanov 和 Nicole Sullivan

为了提升网站性能，保持网站处于“节食”状态非常重要——在不断开发杀手级功能的同时，确保所有新引入的资源都是最精简的，图像优化就是这样一项工作。长久以来，大家认为网站应该包含哪些功能是一个商业问题而非技术问题，所以就算页面大小对总响应时间影响极大，在性能讨论时也很少提及。

网页的响应时间几乎和其大小完全关联，在典型的网页中，图像的大小要占到一半以上（参考图 10-1）。最重要的是，如果想在不删减功能的前提下提升性能，图像优化是最易实施的方案。通常，我们可以在只有轻微质量损耗或无损的前提下大幅减小图像的大小。

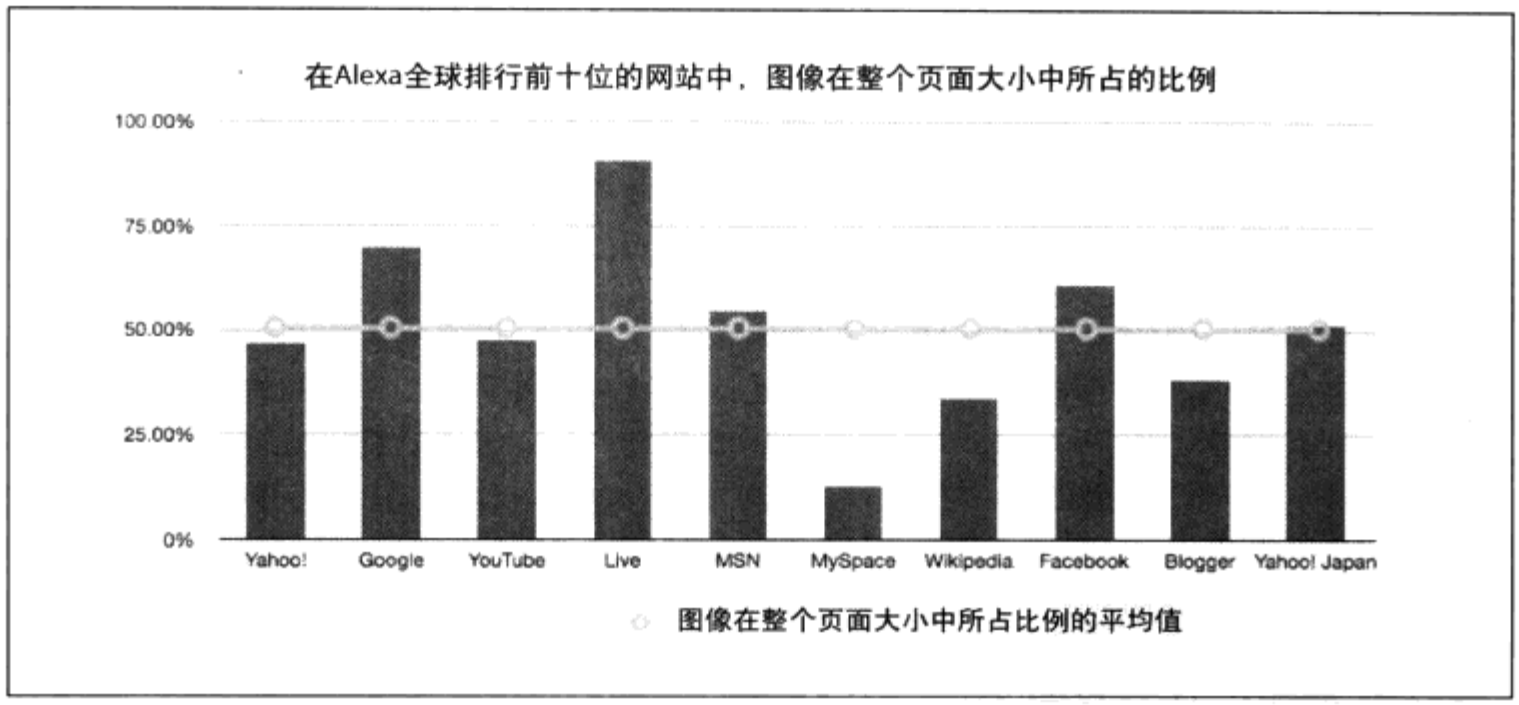


图 10-1：在 Alexa 全球排行前 10 位的网站中，图像在整个页面大小中所占的比例

本章的重点是**无损**优化，也就是在不损耗质量的前提下，将文件大小减到最小。就算精确到像素级别，优化后的图像和原图在视觉质量上也是一样的。移除元数据、对颜色或像素信息进行更好的压缩，以及删除对 Web 显示没有必要的块信息（这是针对 PNG 的情况）是减小图像大小的常用方法。

如果不对图像进行优化，就会通过网络发送一些对用户体验没有任何帮助的额外数据。我们应该毫不犹豫地按照本章推荐的方法进行优化，然而由于这项工作处于开发和设计之间的模糊地带，所以一直以来都是性能优化中容易被忽视的部分。

本章内容包括：

- Web 上不同图像格式的特征（GIF、JPEG 和 PNG）。
- 自动化无损压缩。
- AlphaImageLoader 滤镜。
- 优化 Sprite。
- 其他图像优化技术。

10.1 两步实现简单图像优化

Two Steps to Simplify Image Optimization

为了让图像优化这项工作变得更加简单，可以将其拆分为两个步骤，每个步骤都分属于网站开发过程中的不同工作者：

1. 在开始对图像进行优化之前，首先要确定图像的颜色数、分辨率和清晰度。对这几个参数的修改**有损优化**，同时也会影响整个图像的质量。图像也许可以拥有更少的颜色，对于 JPEG 格式的图像来说，可能意味着减少对图像细节的编码。对于 JPEG 图像来说，虽然 60%~70%的质量是公认的标准，但确实有些图像或背景需要更高或更低质量。例如，相比自动生成的图表或小缩略图，那些精细的名人画像有着更大的文件大小，如何选择，都是根据设计师的创意来决定的，具体的方法可以通过工具来实现，比如 Photoshop 中另存为 Web 格式的功能。设计师同样可以使用空间压缩（译注 1）或区域压缩（译注 2），比如为 Brangelina（译注 3）夫妇的面孔设置 80%的视觉质量，而为作为背景的夜色设置 30%的视觉质量。
2. 一旦决定了需要什么质量的图像，就可以利用**无损压缩**技术尽可能地削减图像大小。和上一步不同的是，这一步特别需要一套工程化的解决方案。手动重复同一项工作是很费时的，实际上，现在已经有很多优秀的开源图像优化工具。你可以通过编写一个脚本来处理所有的图像文件，首先判断文件类型，然后运行相应的工具来进行优化。

译注 1：spatial compression，一种压缩算法，用于优化连续图像中的一桢，使每帧只保留和上一帧不同的内容。

译注 2：zonal compression，一种压缩算法，可以为图像上不同的区域设置不同的压缩系数。

译注 3：Brangelina 是根据好莱坞影星布拉德·皮特和安吉丽娜·朱莉的名字创造出来的词语。

10.2 图像格式
Image Formats

制作最优图像的第一步，就是理解广泛应用在 Web 上的 3 种图像格式——JPEG、PNG 和 GIF——的特点，然后根据实际情况来选择合适的格式。让我们先来讨论下不同格式的背景信息。

10.2.1 背景
Background

这一节将讨论不同格式图像的特点，这些特点决定了如何在 Web 上使用这些图像，同时也是选择图像格式的因素之一。

图形 VS 照片

不管你使用哪种图像格式，优化方法取决于图像的具体类型：

图形

网站的 Logo、草图、图表、大部分动画和图标都属于图形。这些图像通常由连续的线条或其他尖锐的颜色过渡组成，颜色数量相对较少。

照片

照片通常有百万数量级的颜色，并且包含平滑的颜色过渡和渐变，想象一下用相机拍摄的日落时的照片。绘画作品的图像（比如**蒙娜丽莎的微笑**）更接近于照片，而不是图形。

就图像格式而言，GIF 通常用来显示图形，而 JPEG 更适合显示照片。PNG 两者都适合，甚至用**调色板** PNG（palette PNG）显示图形比 GIF 会更好一些。

像素和 RGB

图像由**像素**组成，像素是图像中最小的信息单元。我们可以使用不同的颜色模型来描述像素，在计算机图像处理上，RGB 颜色模型是最常用的一种。

在 RGB 颜色模型中，采用包含红（R）、绿（G）和蓝（B）的数量多少的方式来描述一个像素。R、G 和 B 被称为**成分**（又称为**通道**），每种成分的强度值范围在 0~255 之间。我们经常在 HTML 和 CSS 中使用的是十六位进制的成分值，范围从 00~FF。将不同强度的成分组合在一起，就可以获得不同的颜色。比如：

- 红色是 `rgb (255, 0, 0)` 或十六进制的`#FF0000`。
- 蓝色是 `rgb (0, 0, 255)` 或十六进制的`#0000FF`。

- 灰色阴影很可能有着三个相同的成分值；比如，`rgb (238, 238, 238)` 或十六进制的`#EEEEEE`。

真彩色图像 VS 调色板图像格式

使用 RGB 颜色模型到底可以展现多少种不同的颜色呢？答案是多于 1600 万种：256 * 256 * 256（或者 2^{24}）可以得到 16 777 216 种组合。可以支持这么多颜色的图像格式叫做真彩色图像格式；比如 JPEG 和真彩色类型的 PNG。

为了在存储图像信息时节省一些空间，有一项技术是将图像中各种不同的颜色提取出来建立一个表，这个表通常叫做调色板（也可以称为索引）。有了这个颜色表，就可以通过将调色板中的条目和每个像素重新匹配，达到重新绘制整个图像的目的。

调色板可以包含任意 RGB 颜色值，但是最经常使用的调色板图像格式——GIF 和 PNG8——会限制调色板中最多只能包含 256 种颜色。这不是说你只可以从 256 种已经定义好的颜色中选择，恰恰相反，你可以从 1600+万的颜色中选择你需要的值，但是单个图像中最多只能包含 256 种颜色。

透明和 alpha 通道（RGBA）

RGBA 并非另一种截然不同的颜色模型，而是在 RGB 的基础上做了扩展。额外的成分 A 代表 alpha 透明，值的范围也是从 0~255，但实际上不同的程序和库会将透明定义为从 0%~100%的百分比，或者从 0~127 的值。alpha 通道描述了透过图像像素可以看到下面内容的程度。

假设你已经做出了一个网页，设置了背景，并在上面放置了一个蓝色的图像。如果图像上某个像素的透明度设置为 0，那么该像素下面的背景就不可见。如果 alpha 透明度设置为最大值 100%，那么图像上的像素将会隐藏，而背景则会“浮现出来”。假如设置为中间值，比如 50%，就可以同时看到背景和图像上的像素，图 10-2 显示的就是这样的例子。

隔行扫描

当网速很慢时，大图像会随着下载的进度逐行显示，从上到下每次显示一行，缓慢地向下递进。为了提高用户体验，部分图像格式支持对那些连续采样的图像进行**隔行扫描**。隔行扫描可以让用户在完整下载图像之前，看到图像的一个粗略版本，从心理上消除页面被延迟加载的感觉。

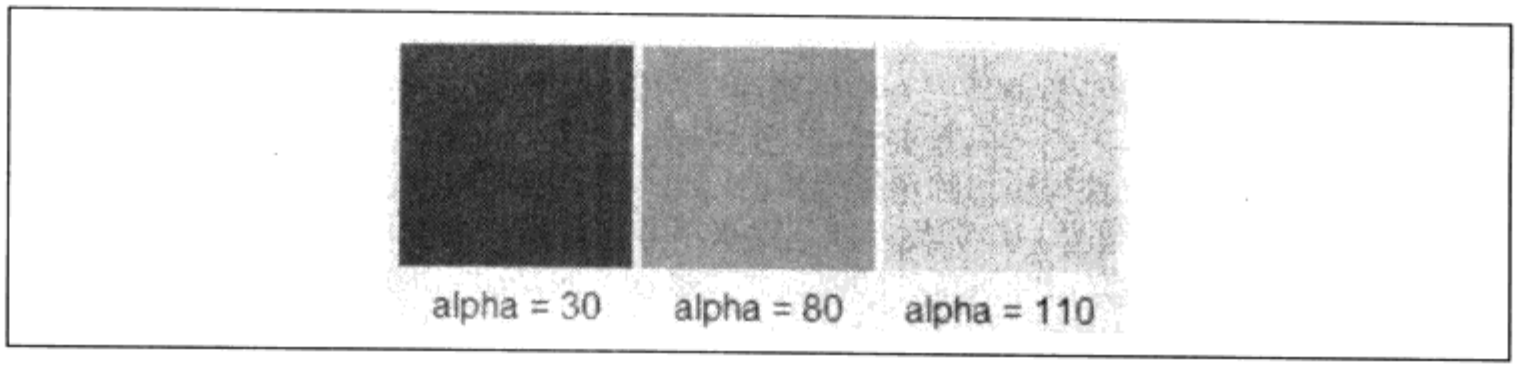


图 10-2：通过 PHP 和 GD 库生成不同透明度的图像，GD 库（译注 4）（*http://php.net/gd*）可以将 alpha 设置为从 0~127 之间的值

10.2.2 不同图像格式的特性

Characteristics of the Different Formats

当了解上面这些背景资料后，让我们来看看 GIF、JPEG 和 PNG 这 3 种格式的区别。

GIF

GIF（译注 5）是图形交换格式（Graphics Interchange Format）的缩写，是一种调色板图像格式。下面介绍一些它的特性：

透明

GIF 允许一个二进制（是/否）类型的透明度，每个像素要么是完全透明的（不包含颜色），要么是完全不透明的（包含一个单色）。这就意味它不支持 alpha（可变的）透明，取而代之的是，调色板中的某个颜色可以被标记为表示透明，而透明像素则会被分配为这个颜色值。所以，如果你为 GIF 设置了透明像素，那么就会“消耗”一个调色板条目。

动画

GIF 格式支持动画。包含动画的图像由若干帧组成，就像几个图像同时包含在一个文件中一样。大家普遍认为 GIF 动画很烦人，这是因为在网络发展初期它被滥用了，那时候大家用 GIF 来制作闪烁文字、旋转的@标志等。GIF 动画现在是有一些应用，比如广告条（虽然现在这已经主要是 Flash 的天下了），还有在富互联网应用（RIAs）中出现的“加载中”指示符。

无损

GIF 是无损的，也就是说你可以打开任意一个 GIF 文件，做一些修改，保存关闭时不会损失任何质量。

译注 4：GD 库是一个可以动态创建图像的开源代码库，*http://www.libgd.org/*。最开始只支持 GIF，GD 的名称由来就是 Gif Draw，后来由于 GIF 中 LZW 压缩算法的专利问题，从 1.6 改为支持 PNG8，GD 这个名称就变得没有实际意义了，通常可以认为是 Graphics Draw 的缩写。现在 GIF 的专利保护已经失效，从 2.0.28 开始已经重新支持 GIF 了，是否支持需要的图像格式，可以通过 gdlib-config --features 命令来查看支持的图像格式。

译注 5：参考 *http://zh.wikipedia.org/wiki/GIF*。

逐行扫描

当生成 GIF 文件时，会使用一个压缩算法（叫做 LZW）来减小文件的大小。当压缩 GIF 时，会从上到下一行一行地对像素进行扫描。这种情况下，当图像在水平方向有很多重复颜色时，可以获得更好的压缩效果。比如，有一个 500×10 像素的图像（宽：500px；高：10px），图像上包含一些条纹，就是说水平方向是由相同颜色线条组成的，将这个图像旋转 90 度后（宽：10px；高：500px），其垂直方向是由相同颜色的条纹组成的，此时后者的文件要大于前者。

隔行扫描

GIF 支持可选的隔行扫描。

由于 GIF 有 256 色的限制，所以不适合用来显示照片，照片所需的颜色数量远大于这个数量级。GIF 更适合用来显示图形（图标、Logo 和图表），但正如这章后面的内容所述，PNG8 是用来显示图形的最佳格式。所以，只有需要动画时才应该使用 GIF。

GIF 格式中使用的 LZW 无损数据压缩算法，在以前是受专利保护的，但是这个专利已在 2004 年过了保护期，现在已经可以自由使用 GIF 了。

JPEG

JPEG 的意思是联合图像专家小组（Joint Photographic Experts Group），也就是开发了这个标准的组织名称。JPEG 是照片存储的实际标准。考虑到人类眼睛对颜色和光线强弱的感知，这种格式通过各种技术来减少显示图片所必需的信息，所以它能在经过高度压缩的文件中存储高分辨率的图像。下面是这种格式的一些特性：

有损

JPEG 是一种有损的格式，用户可以设置自定义质量级别，这个级别决定了有多少图像信息会被丢弃。质量级别的值从 0 到 100，但是就算设置为 100，也同样会有一定程度的质量损耗。

当你要对某个图像进行多项编辑操作时，最好使用无损的图像格式来保存中间结果，然后在完成所有的修改后另存为 JPEG 格式。否则你将会在每次保存时都损耗一些质量。

但也有少数操作是无损的，比如：

- 旋转（只有在旋转 90 度、180 度和 270 度的情况下）。
- 裁剪。
- 翻转（水平或垂直）。
- 从标准模式切换到渐进模式，反之亦然。
- 编辑图像的元数据。

上面最后一个操作对我们的目标来说尤为重要。我们将会在后面利用这个特性来自动优化 JPEG 格式的图像。

透明和动画

JPEG 不支持透明或动画。

隔行扫描

除了默认的标准 JPEG（Baseline JPEG），还有一种渐进 JPEG（Progressive JPEG），支持隔行扫描。Internet Explorer 不会逐步地渲染渐进 JPEG 图像，而是在图像完全下载时立即全部显示出来。

JPEG 是 Web 上用来存储照片的最佳格式，也被广泛应用在数码相机中。然而，这种格式不适合用来存储图形，因为有损的压缩方法将线条和清晰的颜色过渡都变成了“大色块（译注 6）”。

PNG

为了弥补 GIF 格式的缺点并规避 LZW 算法的专利问题，PNG（Portable Network Graphics，便携式网络图片）应运而生。实际上，人们常开玩笑说 PNG 代表“PNG is Not GIF”的递归缩写（译注 7）。下面是这种格式的一些特性：

真彩色和调色板 PNG 格式

PNG 格式有几种子类型，但它们大致可以分为两种：调色板 PNG 格式和真彩色 PNG 格式。你可以使用调色板 PNG 格式来替代 GIF 格式，使用真彩色 PNG 格式来代替 JPEG 格式。

透明

PNG 支持完全的 alpha 透明，在 Internet Explorer 6 中使用这种特性会出现两种奇怪的现象，这个问题我们之后会提到。

动画

虽然已经有相关实验和实际应用存在，但截至目前，针对动画 PNG 格式，还没有跨浏览器的解决方案。

无损

与 JPEG 不同的是，PNG 是一种无损的图像格式：多次编辑不会降低其质量。这使得用真彩色 PNG 来保存 JPEG 的修改过程的中间产物非常适合。

逐行扫描

和 GIF 格式一样，相对于那些垂直方向有重复颜色的图像来说，PNG 格式对那些水平方向有重复颜色的图像压缩比更高。

隔行扫描

PNG 支持隔行扫描，并使用了比 GIF 更好的算法，它允许对真实图像进行更好的“预览”，但是支持隔行扫描的 PNG 图像在文件大小上会更大一些。

10.2.3 PNG 的更多资料

More About PNG

让我们再看一些细节，这可以帮助你更好地理解 PNG 这种格式。

译注 6：这里的大色块是指因高度压缩导致的颜色丢失。

译注 7：*http://en.wikipedia.org/wiki/Recursive_acronym*，计算机领域很早以前的一个传统，就是起名时幽默地引用自身或其他缩写的缩写，很多递归缩写通常用来指出这个缩写指代的事物 A 不是与另一个事物 B 相类似，但事实上，这个事物 A 通常与 B 非常相似，甚至是 B 的衍生品。

PNG8、PNG24 和 PNG32

你可能看过 PNG8、PNG24 和 PNG32 这些名字，让我们澄清它们的含义：

PNG8
: 调色板 PNG 的别称。

PNG24
: 真彩色 PNG 的别称，但是不包括 alpha 通道。

PNG32
: 真彩色 PNG 的别称，包括 alpha 通道。

还有一些其他的类型，比如包含 alpha 通道或不包含 alpha 通道的灰度 PNG，但是这些格式的使用几率要小得多。

比较 PNG 和其他的图像格式

显然，GIF 是为图形设计的，而 JPEG 是为照片设计的，PNG 对两种类型的图像都支持。在本小节，我们将会对 PNG 和其他图像格式进行比较，并提供一些关于 PNG 的额外细节。

和 GIF 比较

除了不支持动画以外，调色板 PNG 拥有 GIF 的所有功能。此外，它还支持 alpha 透明，并且通常压缩比更高，文件大小更小。所以，**应该尽可能使用 PNG8 来代替 GIF。**

有一个例外就是那些颜色数很少的小图像，这时 GIF 的压缩率可能会更高一些。但是这种小图像其实应该被放在 CSS Sprite 中，因为 HTTP 请求的开销已经大大超过节省的那点带宽，而且用 PNG 格式保存 Sprite 图像可以获得更高的压缩率。

和 JPEG 比较

当图像中的颜色数超过 256 种时，需要使用真彩色图像格式——真彩色 PNG 或 JPEG。JPEG 的压缩比更高，而且一般来说，**JPEG 也是照片存储的实际标准**。但由于 JPEG 是有损的，而且在清晰的颜色过渡周围会有大色块，因此以下情况使用 PNG 更合适：

- 当图像的颜色略超过 256 种时，可以在不损耗任何可见质量的前提下，将图像转换为 PNG8 格式。令人惊奇的是，有时候就算你剥离了 1000 种以上的颜色，都不会注意到图像中所发生的变化。
- 当“大色块”变得不可接受时，比如说包含很多颜色的图像或软件菜单的截图，这时候 PNG 就是更好的选择。

PNG 透明的奇怪现象

Internet Explorer 6 中有两个奇怪现象是和 PNG 以及透明相关的：

- 所有在**调色板** PNG 中的半透明像素都会在 Internet Explorer 6 下显示为完整的透明。
- **真彩色** PNG 中的 alpha 透明像素，会显示为有背景色（通常是灰色的）。

第 1 个问题意味着 PNG8 和 GIF 在 Internet Explorer 6 中的行为一致。这点倒不是太坏，你仍然可以用 PNG 代替 GIF 来显示图形图像。因此，PNG8 可以在所有现代浏览器上实现“渐进增强”半透明图像的效果，在 Internet Explorer 6 中降级为和 GIF 类似透明度的图像。

第 2 个问题更加严重一些，但有一些暂时的解决方案，归结起来，要么使用 `AlphaImageLoader` 这个 CSS 属性，要么使用 VML。就像你会在之后看到的内容所述，`AlphaImageLoader` 会带来性能和用户体验上的成本，应该尽可能避免使用。VML 方案的缺点是需要添加额外的标记和代码。总之，要尽量使用 PNG8 来实现设计。

PNG8 和图像编辑软件

不幸的是，大部分图像编辑程序，包括 Photoshop，只能保存二进制透明的 PNG8。一个已知的例外是 Adobe Fireworks（*http://www.adobe.com/products/fireworks/*），这个软件对 alpha 透明度有着极好的支持。还有一些命令行工具，可以将真彩色 PNG 图像转换为调色板 PNG 图像，比如 pngquant（*http://www.libpng.org/pub/png/apps/pngquant.html*）和 pngnq（*http://pngnq.sourceforge.net/*）。

下面是使用 pngquant 命令的例子，参数中的 256 定义了调色板中颜色数量的最大值：

```
pngquant 256 source.png
```

10.3 自动无损图像优化

Automated Lossless Image Optimization

现在你已经了解了不同的图像格式，让我们再来看看如何优化这些图像。你将要看到的这个过程的精彩之处在于：

- 整个过程是自动的，不需要人工介入。
- 所有操作都是无损的，不用担心图像质量会受损。
- 使用免费的命令行工具完成。

不同类型的图像的处理方法不同，但是我们可以使用脚本来完成自动化操作。我们将在本节讨论如下操作：

- 优化 PNG 格式的图像。
- 剥离 JPEG 中的元数据。
- 将静态 GIF（不包含动画）转化为 PNG 格式。
- 对 GIF 动画进行优化。

10.3.1 优化 PNG 格式的图像
Crushing PNGs

PNG 格式将图像信息保存在“块”中。这种方式很利于扩展，因为你可以添加一些自定义的块实现额外功能，而且不识别这些块的程序会自动忽略这些内容。但对于 Web 显示来说，大部分的块都并非必要，我们可以安全地将它们删除。还有一点好处，当我们将叫做 gamma 的块删除后，实际上会提升跨浏览器的显示效果，因为各个浏览器对 gamma 矫正有着迥然不同的支持。

Pngcrush

优化 PNG，我们最喜欢用的工具是 pngcrush（*http://pmt.sourceforge.net/pngcrush/*），用法如下：

```
pngcrush -rem alla -brute -reduce src.png dest.png
```

让我们看看 pngcrush 有哪些参数：

`-rem alla`
: 删除所有的块，但保留控制透明的 alpha 块。

`-brute`
: 使用超过 100 种不同的方法进行压缩，默认值是 10 种。加了这个参数以后会慢很多，而且大部分情况下改进的效果很小。但是如果你是离线进行这个操作，完全可以为这个操作多付出 1~2 秒的时间，因为这个操作可以找到效果更好的方法来压缩图像。但如果使用的场景对性能要求很高，就不要使用这个参数了。

`-reduce`
: 如有可能，尝试减少调色板中的颜色数量。

`src.png`
: 源图片。

`dest.png`
: 目标（优化后的）图片。

其他 PNG 优化工具

Pngcrush 在执行速度和优化结果上达到了一个很好的平衡点。但如果你想获得最大的压缩效果，就要将更多的时间花在压缩上，这时可以试试其他工具。但具体的结果还是要取决于图片，你甚至可以使用所有的工具进行连续压缩。

已知的工具包括：

PNGOUT（*http://advsys.net/ken/utils.htm*）
: 只能运行在 Windows 下，没有开源，只有可执行程序。

OptiPNG（*http://optipng.sourceforge.net/*）
: 跨平台，开源，通过命令行调用。

PngOptimizer（*http://psydk.org/PngOptimizer.php*）
: Windows 平台，开源，同时包含图形界面和命令行接口。

还有一个重量级工具 PNGslim（*http://people.bath.ac.uk/ea2aced/tech/png/pngslim.zip*）。这是一个 Windows 下的批处理文件，可以同时运行很多其他工具。它最主要的作用是，可以用各种不同的参数运行上百次的 PNGOUT。PNGOUT 是我用过的最慢的工具，所以你必须做好足够的心理准备，花很长的时间来运行 PNGslim，有时候几个小时才能优化完一个文件。

10.3.2 剥离 JPEG 的元数据

Stripping JPEG Metadata

JPEG 文件中包含如下的元数据：

- 注释。
- 应用程序定义的内部信息（比如 Photoshop）。
- EXIF（译注 8）信息，比如拍摄用的相机型号、拍摄日期、拍摄位置、缩略图等，甚至还可以包含音频信息等。

这些元数据不会影响图像显示，可以被安全地移除。对元数据的处理，凑巧也是我们之前提到的对 JPEG 进行无损压缩的方法之一，可以将文件中那些不需要的部分直接剔除，而不会影响视觉质量。

压缩工具 jpegtran（*http://jpegclub.org/*）可以通过命令行完成这些转换工作：

```
jpegtran -copy none -optimize src.jpg > dest.jpg
```

这个例子中使用的参数有：

`-copy none`
: 设置不包含任何元数据。

`-optimize`
: 强制对霍夫曼表（译注 9）进行优化，从而获得更高的压缩比。

`src.jpg`
: 需要优化的图像。

`dest.jpg`
: 优化过的图像。

这条命令只能将数据写入标准输出接口，所以为了创建最终的文件，这个例子将输出指向了一个名为 dest.jpg 的文件。

译注 8：EXIF 是可交换图像文件（Exchangeable Image File Format）的缩写。

译注 9：也称霍夫曼码表，完成霍夫曼编码所需的码字-符号表。霍夫曼编码（Huffman Coding）是一种编码方式，是一种用于无损数据压缩的熵编码（权编码）算法。在 JPEG 有损压缩算法中，使用霍夫曼编码器来减少熵。

警告：切记只对你自己拥有的图像进行去除元数据的操作。如果将属于别人的图像中的元数据去除，那么很可能会将有关版权和作者的数据也一并去除，这是违法的。

Jpegtran 只提供了移除元数据的方法，如果要对元数据进行更详细的编辑，请使用 ExifTool（*http://www.sno.phy.queensu.ca/~phil/exiftool/*）。

10.3.3 将 GIF 转换成 PNG
Converting GIF to PNG

正如我们之前讨论的，PNG8 格式支持 GIF 所有的功能，所以将 GIF 转换成 PNG8，视觉上应该感觉不到变化。你可以使用命令行工具 ImageMagick（*http://www.imagemagick.org/*）来完成这个转换：

```
convert source.gif destination.png
```

可以通过下面的方式强行将图像转换为 PNG8 格式：

```
convert source.gif PNG8:destination.png
```

这条命令基本上没有什么用，因为 GIF 图像可以通过任何方式转化为 PNG8。ImageMagick 会根据图像中的颜色数量来决定合适的格式。

当你将 GIF 转换为 PNG 之后，**不要忘了使用 pngcrush 对 PNG 再进行一遍优化**（具体参考本章之前的介绍）。

你还可以使用 ImageMagick 的验证工具，通过编程方式确定 GIF 文件中是否包含动画。例如：

```
identify -format %m my.gif
```

如果 GIF 图像中不包含动画，则这条命令会返回“GIF”。如果包含 GIF 动画，将会返回一个类似于“GIFGIFGIF...”的字符串，每个 GIF 代表其中的一帧。如果你通过运行脚本来转换文件，对输出结果的前 6 个字符进行判断，如果是“GIFGIF”，则表明这是一个动画文件，在这种情况下，你可以直接跳到下一步。

10.3.4 优化 GIF 动画
Optimizing GIF Animations

到目前为止，我们已经对单帧 GIF、PNG 和 JPEG 进行了优化。最后一件事情，就是对 GIF 动画进行优化。这里有个很有用的工具，就是 Gifsicle（*http://www.lcdf.org/gifsicle/*）。因为动画包含很多帧，并且图像上的部分内容在不同帧上都是一样的，所以 Gifsicle 通过将动画里面连续帧中的重复像素移除，来达到优化的目的。具体命令如下所示：

```
gifsicle -O2 src.gif > dest.gif
```

10.3.5 Smush.it

Smush.it（*http://smush.it*）是一个在线的图像优化工具，由本章的作者开发。这个工具的具体工作就是之前 4 节提到的内容，对多种文件类型应用无损图像压缩。Smush.it 包含一个方便的 Firefox 扩展，可以一次性将你浏览的任何页面上的所有图像进行优化。你可以通过下面的步骤经常检查一下，看看究竟可以节省多少字节。

需要注意的是，Smush.it 对 JPEG 的优化很有限，因为没有对 JPEG 的元数据进行剥离，以防不小心剥离掉版权信息，让 JPEG 图像变成“孤儿”。如果你通过上面提到的技术和工具推出了自己的“smushing”工具，并且确定将元数据移除是合适的，只需要在 jpegtran 后面带上`-copy none`参数。

10.3.6 使用渐进 JPEG 格式来存储大图像

Progressive JPEGs for Large Images

当我们回顾不同的文件格式时，发现有一种叫做渐进 JPEG 的图像格式，可以在浏览器中逐步进行渲染，就是允许用户在图像还在传输时，就能看到一个低分辨率的图像。问题是，渐进 JPEG 图像的大小和未经过渐进处理的图像相比，是更大，还是更小呢？

通过对从 Yahoo!图片搜索 API 中随机找到的 10 000 张图像进行试验（*http://yuiblog.com/blog/2008/12/05/imageopt-4/*），我们得出的结论是无法确定。实际上，测试结果遍布了整个图表，但从中可以看出一个趋势，大于 10KB 的图像可以通过转换成渐进 JPEG 图像来获得更好的压缩率，较小的图像适合非渐进的**标准** JPEG 图像。图 10-3 汇总了我们的发现，用图表展示了原来的文件大小和经过优化的文件大小的差别。图表展示的最大大小是 30KB，但是趋势非常平坦，意味着渐进编码所带来的收益是随着文件大小递增的。

图 10-3：文件大小和渐进 JPEG 图像带来的收益之间的关系。Y 轴代表标准图像和渐进图像之间在文件大小上的差别，Y 轴的值越大，代表越适合使用渐进编码

10.4 Alpha 透明：避免使用 AlphaImageLoader

Alpha Transparency: Avoid AlphaImageLoader

在 Web 设计领域，透明效果极受欢迎，但实现跨浏览器的 alpha 透明效果比想象的要难很多。PNG 标准是在 10 年以前制定的，但是糟糕的浏览器支持意味着我们仍在寻找一个完美的解决方案。支持真彩色 PNG 的步伐进行得十分缓慢。Internet Explorer 6 还占有很大的市场份额，但在 PNG 的 alpha 透明度处理上，却有几个严重的技术限制。

在这一节，我们将仔细看看 alpha **滤镜**，这个在老版本的 Internet Explorer 中用来支持 alpha 透明度的技术。Internet Explorer 提供了一个叫做 `AlphaImageLoader` 的滤镜，它现在非常流行。根据在 Yahoo!的试验和实践，我们得出的结论就是你不应该使用 `AlphaImageLoader` 来解决 Internet Explorer 的透明度问题。我们将解释这样做的原因，并通过几个渐进增强的 PNG8 实例来突破这些限制。

10.4.1 Alpha 透明度的效果
Effects of Alpha Transparency

就像你之前看到的那样，透明度有两种类型。第 1 种是二进制透明度，每个像素要么完全透明，要么完全不透明。第 2 种是 alpha 透明度，允许你设置不同级别的透明。

alpha 透明在 Internet Explorer 6 下缺乏足够的支持，对于 Web 开发者来说，平滑的过渡和阴影已经成为一项挑战。例如，图 10-4 中左边的图像，展示的是我们想要实现的效果（部分透明，这样就可以将部分背景显示出来）；右边的图像将背景色显示了出来（在这个例子中是白色），这是当只支持二进制透明时迫不得已的做法。固定颜色背景在两张图像中都可以正常显示，因为它们都允许用二进制透明图像完全模拟 alpha 透明的效果。但是如果背景色不相同，就算是相同的图标，也不可以重复利用。

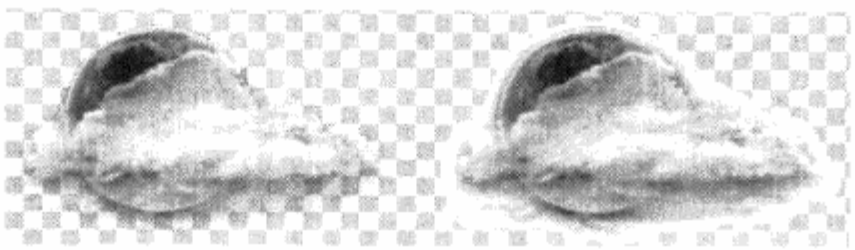


图 10-4：My Yahoo!中的两种天气图标，分别是 Alpha 透明的和二进制透明的；方格图案是图像编辑软件用来标记透明度的典型模式

通常情况下，alpha 透明用于那些背景会变化的情况，比如照片、图形或渐变。在这些例子中，很难模拟透明度，因为你不能确定什么颜色会出现在给定图像的后面。

完美的 alpha 透明度允许将云状的图像置于任意背景之上，而且都会显示得很漂亮。二进制透明（图 10-4 右侧的图像）需要一个更加创新的方法：设计师通过在图像周围添加一些和背景色相近的色块，来实现近似的透明效果。

渐变背景（参见图 10-5）需要对云彩的边缘做更加小心的处理。如果边缘留得过于宽松，或者在图像上留下了阴影，图像的某些区域就会感觉很暗，在另外的一些区域又会感觉很亮。这可以让我们尽可能少地留下背景色。在图 10-6 中，很明显图像的周围留下了太多的颜色。图像顶部的中间色调太浅，而底部又太深。

图 10-5：渐变和透明度

图 10-6：My Yahoo!页面上的实例，展示了天气图标位于一个变化的背景之上的效果；这个设计应用了所有之前讨论过的要求：渐变、实色和模式

山顶角（译注 10）

另一个例子就是无处不在的圆角模块。有一点很重要，就是应该避免将背景的颜色和轮廓合并成一张图片，因为这样你需要为每种背景都准备一张和轮廓合并的图片，从而导致 **HTTP 请求数急剧增加。**

从创建可扩展的 CSS 来考虑，我们应该将轮廓和页面背景色从整块背景色或图片中分离出来。有趣的是，将两者分离需要像素级别的精确选择，并且依赖于人眼对平滑过渡趋势的判断。Dan Cederholm 在 A List Apart 上发表的《山顶角（Mountaintop Corners）》（*http://www.alistapart.com/articles/mountaintop/*）中，首次提到这点。

图 10-7 的模块中有两个深色方块，看起来没有什么差别，但是当我们使用高分辨率来比较模块的背景和前景色，模块的边缘就像被咬过了一样，而不是完全的平滑。为了跨浏览器实现统一的外观和感觉，开发者开始对 Internet Explorer 使用 `AlphaImageLoader` 滤镜。

译注 10：一种实现圆角的技术，由 Dan Cederhom 于 2004 年发表，通过两张透明图片，只要做很小的修改，就可以在不同的背景颜色下实现圆角效果，*http://www.alistapart.com/articles/mountaintop/*。

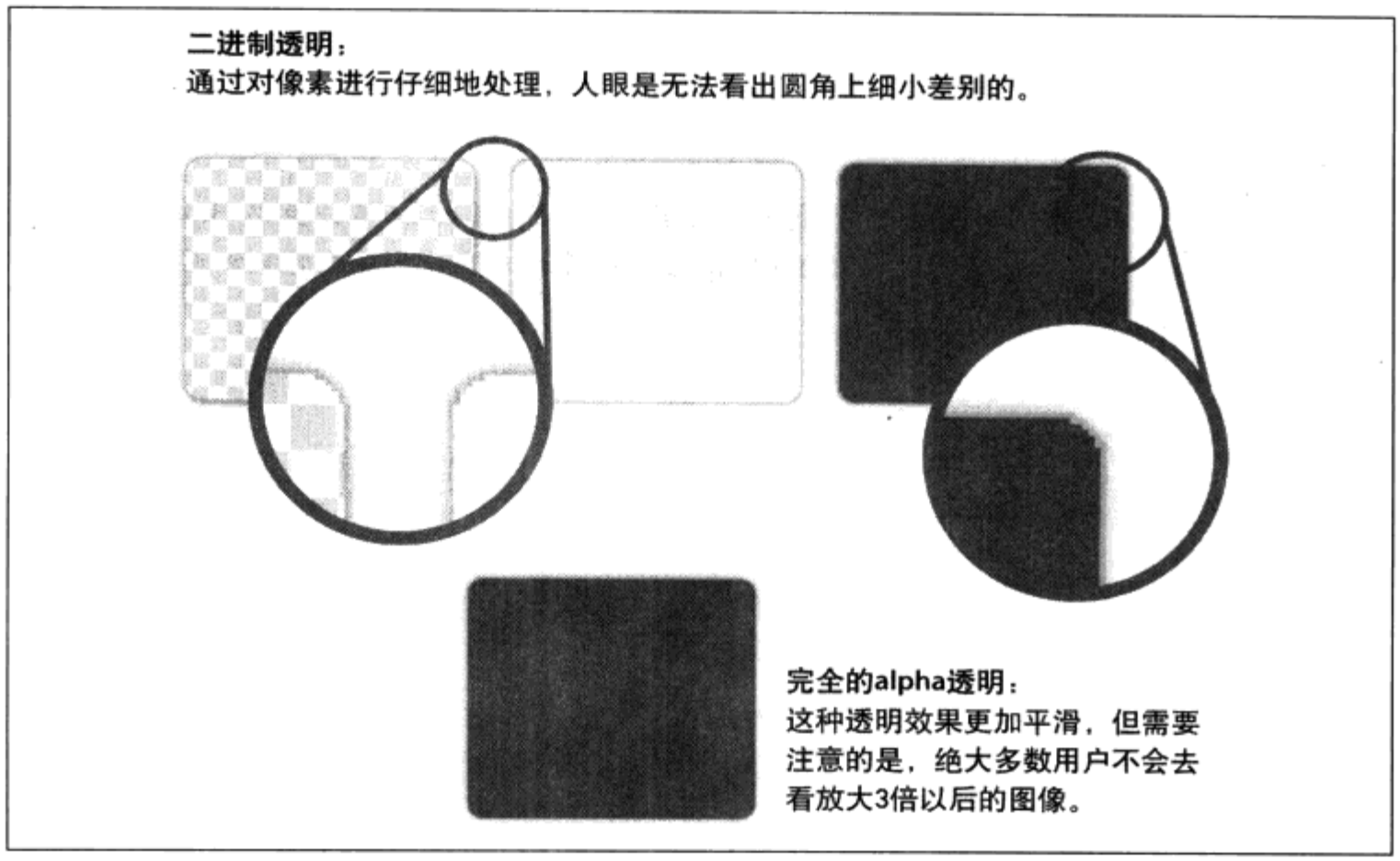


图 10-7：二进制透明和 alpha 透明实现的圆角模块

10.4.2 AlphaImageLoader

Internet Explorer 不支持原生的 alpha 透明。专有的滤镜就是为了填补这个空白；但是这样做所带来的性能损耗非常明显。为了更好地理解这种做法的缺陷，让我们来看看有哪些事情是不应该做的（参考示例 10-1）。

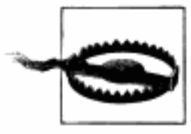
示例 10-1：使用 AlphaImageLoader 为 PNG 图片添加圆角

```
.myModule .corner {
  background-image: url(corner.png);
  _background-image: none;
  _filter:progid:DXImageTransform.Microsoft.AlphaImageLoader(
     src='corner.png',
     sizingMethod='scale'
  );
}
```

示例 10-1 中，在属性前面添加一个下划线是个 hack 技巧，这样样式只有在低于 Internet Explorer 7 的版本中才会生效：

- `_background-image` 属性首先将原来的背景图像 corner.png 移除。
- `_filter` 属性使用 Microsoft 的 `AlphaImageLoader` 滤镜重新加载相同的图片。

只有在 Internet Explorer 6（或者更低版本）浏览器中才需要这个 hack 技巧。Internet Explorer 7 及更高版本的浏览器已经原生支持 alpha 透明度，就像 Firefox、Safari 和 Opera 一样了。所有这些浏览器都会忽略以下划线开始的规则，因为这些属性是无法被识别的。



警告：如果你忘记使用下划线的 hack 技巧，那么虽然 Internet Explorer 7 已经实现了对 alpha 透明度的原生支持，但还是会使用滤镜进行渲染。

10.4.3 AlphaImageLoader 的问题

Problems with AlphaImageLoader

使用 alpha 滤镜既会增加维护成本，又会造成直接的性能损耗。

代码分支

从维护的角度来看，在上面这个例子中，就算将代码分支的数量控制到最小，也是非常危险的。当我们在 CSS 规则中写入一些例外情况，文件大小就会随着时间不断增长。还有，假设我们同时使用 CSS Sprite 来减少 HTTP 请求数，就像《高性能网站建设指南》这本书中推荐的一样，那么当使用 alpha 滤镜后，`background-position` 将不被支持。在这种情况下，通常会使用 clip 属性来模拟 Internet Explorer 下的背景位置。

冻结浏览器

当你使用 alpha 滤镜后，页面就不再支持渐进渲染。在所有必须的组件下载完成之前，用户只能看到空白页面。页面上的元素仍然可以并行下载，但是渲染会被阻塞，因为 Internet Explorer 会在所有 CSS 都下载完毕后才开始进行渲染，但是 CSS 现在却依赖于一张需要经过滤镜处理的图片。（想更多了解渲染，请参考 *http://www.phpied.com/rendering-styles*。）如果你在页面设置了几个 `AlphaImageLoader` 滤镜，那么它们的处理过程是串行的，这样问题就被成倍放大了。比如你有 5 张图像，每个图像都在服务器上延迟两秒，累加起来，浏览器就要冻结 10 秒。

增加内存消耗

使用 `AlphaImageLoader` 的另外一个负面效果就是增加内存占用，这些内存用于处理和应用滤镜。我们现在可能会误以为所有用户的电脑都有着几乎取之不尽的内存，但是对于那些老式的电脑来说绝对不是这样的，而且这些电脑很可能运行着 Internet Explorer 6 或更低版本的浏览器。

所有这些应用在图片上的开销，每在页面上显示一次就会支出一次，因为滤镜没有修改图像文件本身，而是修改应用这些样式的 HTML 元素。如果一个 Sprite 同时应用在页面中 20 个元素上，付出的性能损耗不是 1 份，而是 20 份！此外，所有元素都在同一个 UI 线程中处理。通常这个滤镜用于覆盖在视频上的播放按钮（参见图 10-8）。这种情况下，任何性能损耗都会体现在页面每一个视频上。

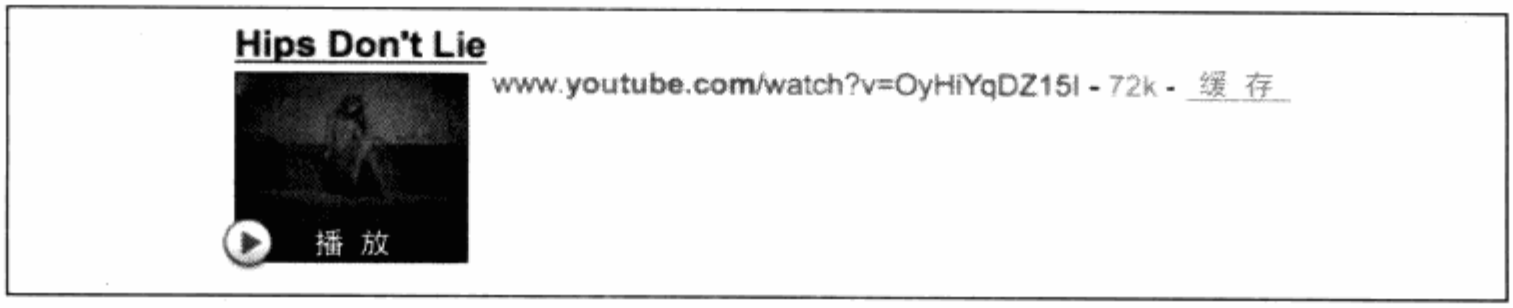


图 10-8：Yahoo! Search 为了提高性能，去掉了视频播放器上播放按钮的 alpha 透明

真实案例研究：Yahoo!搜索

对于估算代码修改对性能产生的影响，在实验室中测试是一个很好的途径，但是没有什么比在真实环境中测试更棒了，这意味着来自真实用户数百万的请求，无数的浏览器设置、不同的地理位置、连接速度、硬件和操作系统。

根据实验室的测试结果，我们估计那些应用了 `AlphaImageLoader` 滤镜的 HTML 元素，每个大约会增加 8 毫秒的性能成本。而在从前，搜索团队使用真彩色的 PNG 图像和滤镜来制作 Sprite，该 Sprite 在页面中出现了 12 次。因此，我们预计可以得到 96 毫秒的速度提升，但是我们非常急迫地想知道，真实的用户测试跟我们的测试结果是否相似。

这个测试是对两个相同的搜索结果页面进行比较，一个使用了 `AlphaImageLoader` 滤镜，另外一个没有。结果集对比了两种不同的人群，显示出了 50~100 毫秒的提升。使用 Internet Explorer 6 的用户的响应时间提升了 100 毫秒，而使用 Internet Explorer 5 的用户的响应时间提升了 50 毫秒。

100 毫秒（十分之一秒）的速度提升看起来很少，但是 Amazon 的实验数据显示，响应时间增加 100 毫秒会导致 1% 的销售额下降（*http://home.blarg.net/~glinden/StanfordDataMining.2006-11-29.ppt*）。理解了收入和性能之间的直接联系后，为了将投资控制在最低水平，必然会将真彩色 PNG 转化为 PNG8。

基于这个测试，推荐你尽可能避免使用 `AlphaImageLoader`。如何避免呢？在下一小节，我们将介绍一些可以避免使用滤镜的技术。

10.4.4 渐进增强的 PNG8 Alpha 透明

Progressively Enhanced PNG8 Alpha Transparency

如果已经决定了必须使用 alpha 透明，但是又不想承受微软专用 alpha 滤镜所带来的性能损

耗，这时可以采用**渐进增强的 PNG8**。首先创建一个使用 alpha 透明的图像，然后按照下面的步骤操作：

1. 建立一个二进制透明图像，所有的像素要么完全不透明，要么完全透明。
2. 编写使用这个图像所需要的 CSS。
3. 确认这个图像在没有 alpha 透明的情况下可以正常显示。
4. 添加部分透明像素，它们可以被更先进浏览器识别。你可以使用 Photoshop 或你喜欢的其他工具将两个图像分层，并将结果保存为不同的文件，为了将最终的图像保存为支持 alpha 透明的 PNG8，你可能需要 Fireworks 或一个叫做 pngnq 的命令行工具。我们不建议通过软件将真彩色 PNG 自动转换成 PNG8，因为二进制透明的版本不太可能达到可以接受的质量要求。

想了解更多关于渐进增强 PNG8 的内容，可以阅读 Alex Walker 发表在 SitePoint 上的文章《PNG8——The Clear Winner》（*http://www.sitepoint.com/blogs/2007/09/18/png8-the-clear-winner/*）。

警告：你可以使用 PNG8 来渐进增强图像，这些图像有着清晰的二进制透明。一个包含半透明像素的图像，在 Internet Explorer 6 下会被渲染成完全透明。

举例来说，渐进增强的 PNG8 可以用于那些覆盖在变化背景之上的模块，比如 Yahoo! Travel map 上带阴影的弹出框，正如图 10-9 和图 10-10 所示。

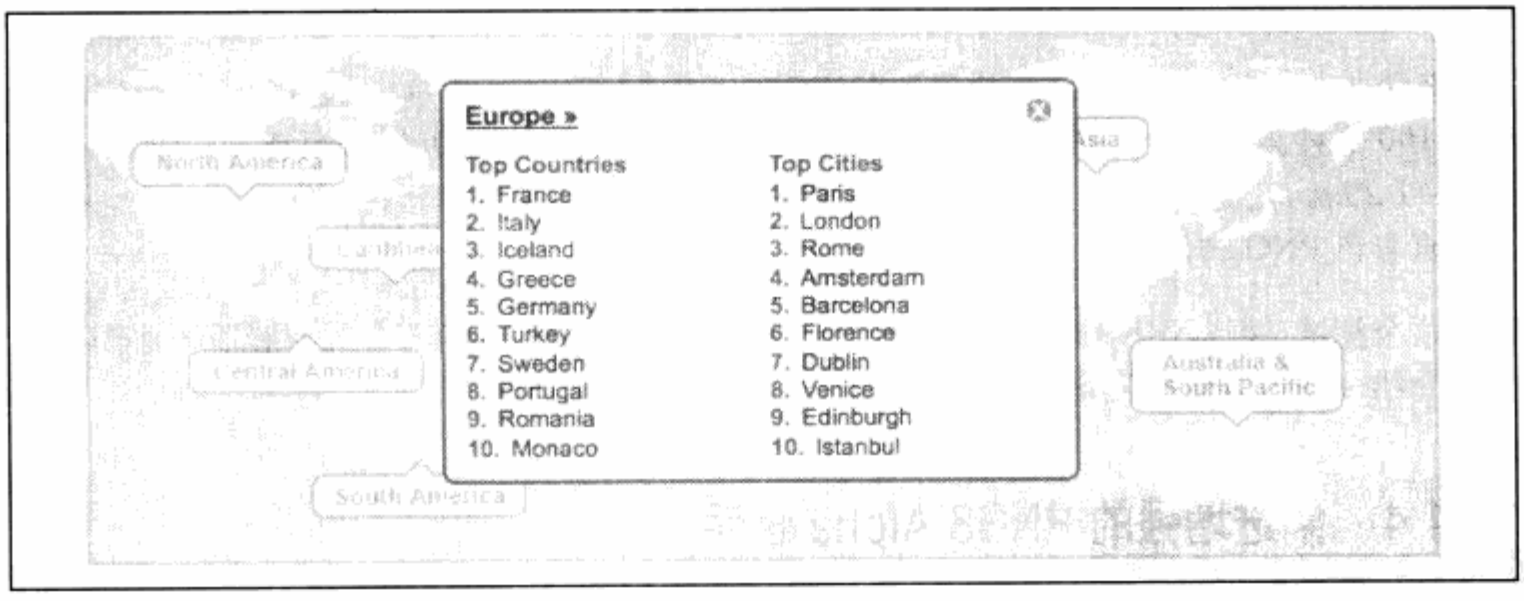


图 10-9：Internet Explorer 显示的是一个简化版，只有一个清晰的 3 像素边框

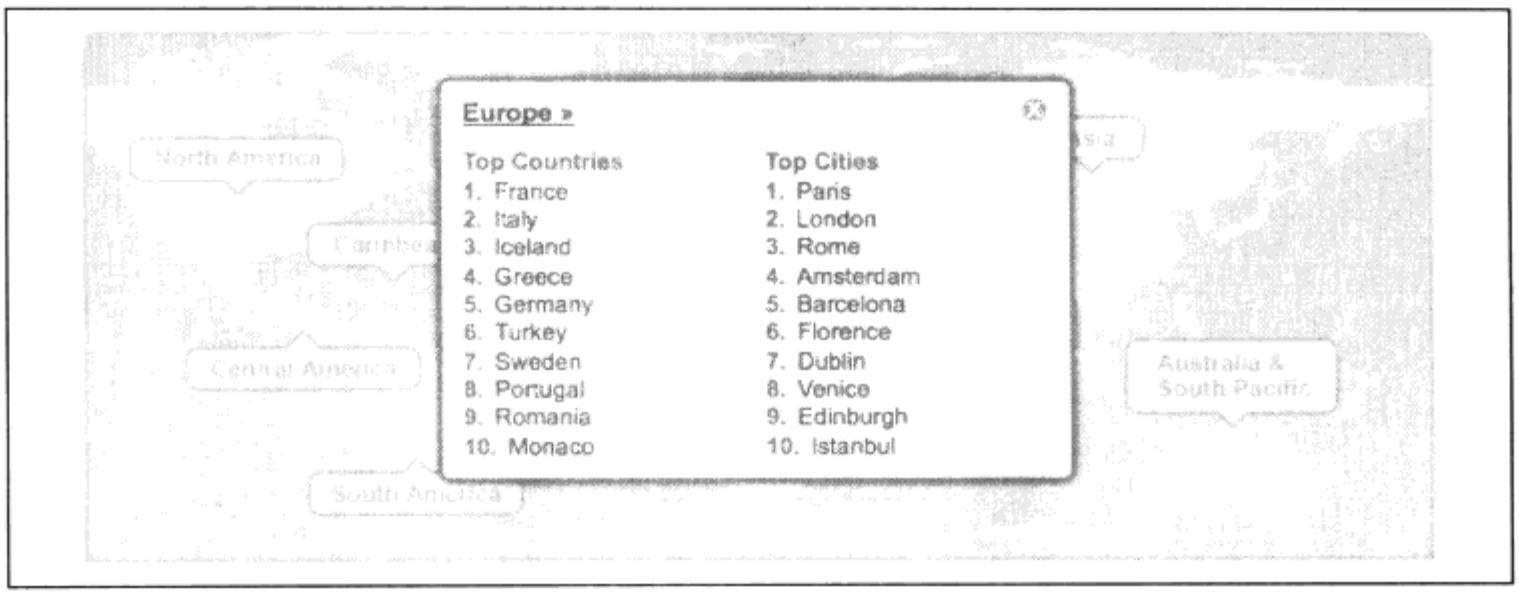


图 10-10：更先进的浏览器，比如 Internet 7 和 8、Firefox、Safari 及 Opera 还会额外显示一个阴影

提示：人类的眼睛对阴影的变化非常敏感，因为人类在辨识对象尤其是人的时候，完全建立在对形状的认知基础上。仔细观察图形的边缘，如果当它们影响到图标或者图像的外观时，错误会更加明显。

10.5 优化 Sprite
Optimizing Sprites

Dave Shea 创造了 CSS Sprite 这个术语，指将多个背景图片合并到一个较大的图片中，通过修改背景的位置，在元素上显示背景图片的一部分（*http://www.alistapart.com/articles/sprites/*）。这项技术随后就被 Yahoo!采用，通过减少 Yahoo!首页上小图标所带来的请求数来提升性能。有两种方法可以用来优化 Sprite：“无所不包”（everything and the kitchen sink）法和模块化面向对象的方法，要想知道哪种方法更适合你的网站，需要问自己几个问题：

- 你的网站有多少页面？
- 你的网站是基于模块化开发的么？（提示：非常应该基于模块化开发！）
- 你的团队可以在网站维护上花费多少时间？

上面几个问题的答案可以帮助你在 Sprite 数量、维护成本和不重复页面数之间来做权衡。你可以选择任意两项，但无法同时选择 3 项（参见图 10-11）。

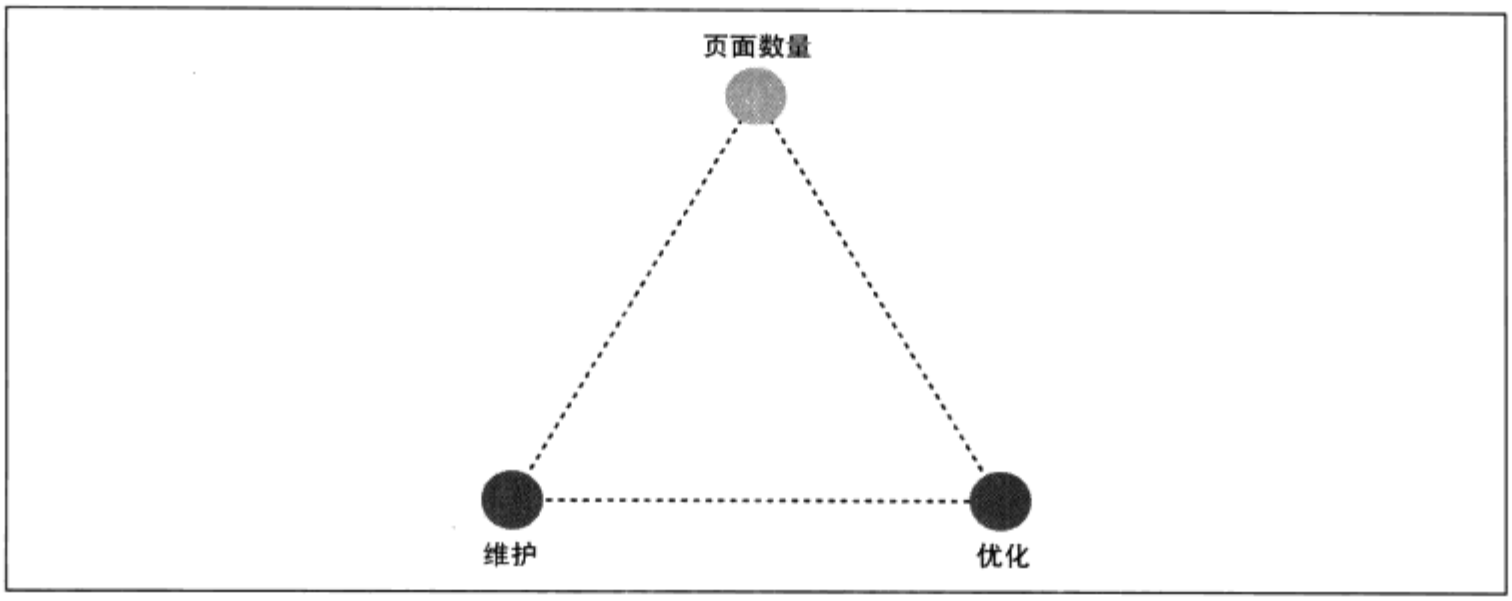


图 10-11：Sprite 的困惑：只能选择两项

10.5.1 超级 Sprite VS.模块化 Sprite

Über-Sprite Versus Modular Sprite

如果网站的**页面很少**，最好的办法是将网站上的所有图像放到一个超级 Sprite（译注 11）中，Google 就在搜索结果页中使用了一个超级 Sprite，正如图 10-12 所示。

图 10-12：Google 搜索只有两个页面；因此，它可以创建一个超级 Sprite，而不需要付出明显的维护成本

另一方面，如果你的网站有很多页面，就需要一个不同的 Sprite 策略，否则维护成本将会变得非常高。最终的目标应该是可以很方便地将那些不再使用的模块从网站删除；否则，曾经的高性能网站也将变得笨重起来。为了实现这种效果，你可以将属于同一个对象的图像合并在一起。比如：

- 同一个圆角框的四个角。
- 模块头部滑动门所用的左右两边（译注 12）。

译注 11：über-sprite，über 是一个德语单词，本意为“在……之上”，在英语网络文化中，引申为“超级”之意。

译注 12：*http://www.alistapart.com/articles/slidingdoors/*。

- 组成按钮的 2~4 张图像。
- Tab 的状态，比如当前、悬浮和正常效果。

在模块化方法中，这些 Sprite 不应该和其他 Sprite 合并到一起。

10.5.2 高度优化的 CSS Sprite
Highly Optimized CSS Sprites

有时，优化 Sprite 要比优化图像复杂得多。将不同的资源放入一个 Sprite 中，有可能很难再被压缩。参照下面的最佳实践，可以让你的 Sprite 变得尽可能小：

- **按照颜色合并**；比如，将颜色调色板相近的图标组合在一起。
- **避免不必要的空白**，让图像在移动设备上更容易处理。
- **将元素水平排列**，而不是垂直的。这样 Sprite 会稍微变小。
- **将颜色限制**在 256 种以内，这是 PNG8 格式的颜色数量上限。
- **先优化单独的图像，再优化** Sprite。在调色板色值有限的情况下，可以更容易减少颜色数。
- 通过**控制大小**和对齐减少反锯齿像素的数量。如果一个图标稍微接近正方形，通常你可以通过在水平方向或垂直方向对齐来减少反锯齿像素的数量。
- **避免使用对角线渐变**，这种渐变无法被平铺。
- **避免在 Internet Explorer 6 中使用 alpha 透明图像，将需要真彩色 alpha 透明的图像保存在单独的 Sprite 中。**
- **每 2~3 个像素改变渐变的颜色**，而不是每个像素都改变。
- **处理 Logo 的时候要小心**，Logo 很易识别，就算非常小的修改也会很容易被注意到。

10.6 其他图像优化方法
Other Image Optimizations

本章的剩余部分将讨论和图像相关的优化，这些优化可以帮助网页加载得更快。具体关系到应该如何使用图像文件，而不只是图像本身。

10.6.1 避免对图像进行缩放
Avoid Scaling Images

当你像下面这样，在 HTML 中将一张 500×500 像素的图像缩小，就会带来不必要的下载开销：

```
<img src="image.jpg" width="100" height="100" alt="my image" />
```

在这个例子中，你让浏览器将图像缩小，显示了一张 100*100 像素的更小版。但浏览器还

是需要下载那张大图。如果你在服务器端改变图像的大小，并提供一个较小的版本，就可以明显节省流量。值得注意的是，有些浏览器把缩放图像，作为额外的卖点，但它并没有做得像一些软件（比如 image Magick）那样好，这导致浏览器在被迫执行缩小操作时，既降低了图片质量又增大了下载量。

10.6.2 优化生成的图像
Crush Generated Images

如果你正在开发一个报表应用程序或模块，在运行过程中就会需要生成不同类型的图形和图表。当你生成了那些类型的图像后，应该牢记以下两点：

- 相比 GIF，优先选择 PNG 是非常明智的，PNG8 是最佳的选择。
- 在服务器保存之前别忘了使用 pngcrush 对结果图像进行优化。

在 Google Chart API 文档（*http://code.google.com/apis/chart/types.html*）中找到的这张图像，就是一个不错的例子（参见图 10-13）。如果你对这方面不熟悉，Chart 是一个非常棒的 API，可以通过在 URL 中设置参数来生成图表。让我们来看看这项服务可以如何改善，让生成的图像变得更小。

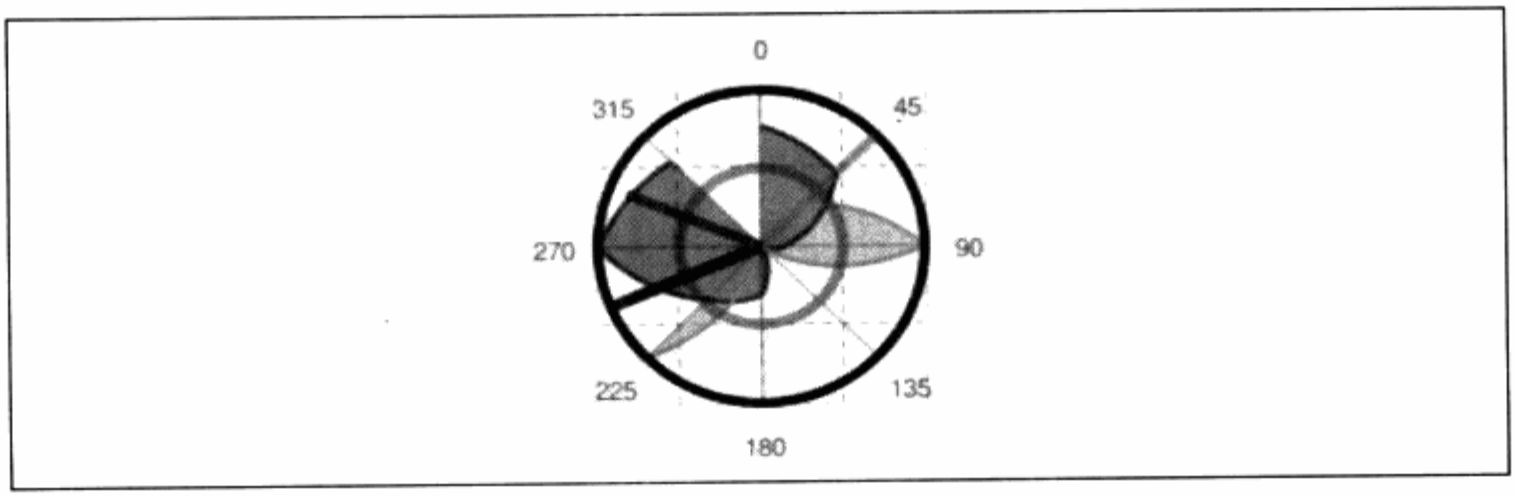


图 10-13：自动生成图表的示例

图 10-13 包含 1704 种颜色，文件大小是 17 027 字节。两个简单的优化就可以将文件大小减小一半以上：

- 用 pngcrush 对这个图像进行处理，结果是 12 882 字节，节省了 24%，没有任何损耗。
- 再进一步，使用 pngquant 将图像转换为 PNG8 格式，删除了大概 1500 种颜色，这些颜色浏览者是看不出来的。新文件的大小是 7710 字节，和原始图像相比节省了 55%。

将生成的图像保存在磁盘上并使用 pngcrush 进行优化的另外一个好处是，当第 2 次请求相同的图像时，不需要重新生成这个图像，可以将之前已经缓存并进行优化的那张图像给用户。

下面是一个简单的代码段，使用 PHP 下的 GD 图像库（http://php.net/gd）实现了这条建议：

```
<?php
header ('Content-type: image/png');

// name of the image file
$cachedir = 'myimagecache/';
$file = $cachedir . 'myimage.png';

// if in the cache, serve
if (file_exists($file)) {
    echo file_get_contents($file);
    die();
}

// new GD image
$im = @imagecreatetruecolor(200, 200);
// ... the rest of the image generation ...
imagepng($im, $file); // save
imagedestroy($im); // cleanup

// crush the image
$cmd = array();
$cmd[] = "pngcrush -rem alla $file.png $file";
$cmd[] = "rm -f $file.png";
exec(implode(';', $cmd));

// spit out the new image
echo file_get_contents($file);
?>
```

还有一种选择，不采用 `file_get_contents` 来读取文件，而是使用重定向来告诉浏览器新的地址。为浏览器添加 `Expires` 头，这样就可以支持重定向缓存，因为它们会重用这个图像，而不会在每次访问时重新下载。

10.6.3 Favicons

Favicons 是命名为 favicon.ico 的小图像，保存在网站根目录，可以显示在浏览器地址栏上，就在网址左面（参见图 10-14）。

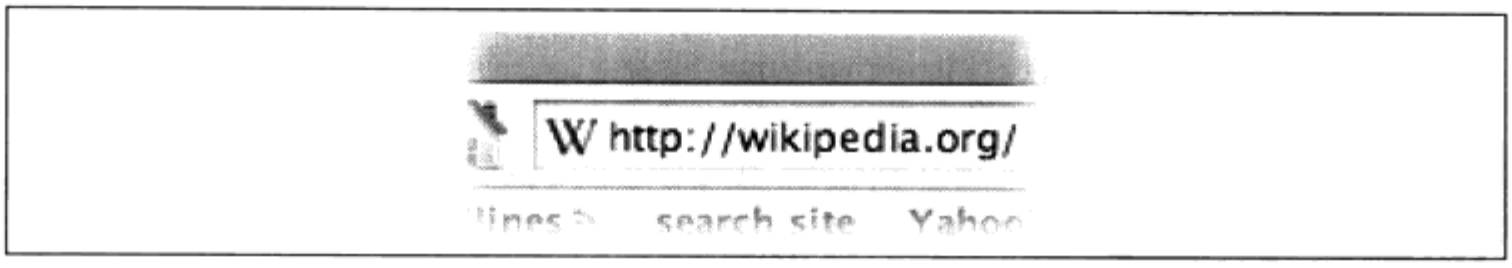


图 10-14：Wikipedia 的 favicon

这个页面组件经常被忽视，因为它很小，而且推测是可以被缓存的。但是缓存还没有像我

们想象得那么普遍。对于任何组件都是这样，当然 favicons 也不例外。Yahoo! Search 发现，在所有页面中，只有 9%使用它的 favicon。

对于 favicons 来说，有几点可以明显地提高性能：

- 确定建立了 favicon。因为不管怎样，浏览器都会请求这个文件，没有什么理由返回一个 404 错误，尤其当你的 404 处理程序需要消耗数据库连接或其他昂贵资源的时候。
- 考虑为 favicon 添加一个 `Expires` 头。你不能将/favicon.ico 设置为“永远”缓存，因为这样当你决定对它进行修改时，却无法修改文件名称。但你仍旧可以缓存几个月，甚至一年。检查你的 favicon 文件的最后修改时间，来判断你修改的频率。当出现了一些紧急情况，你可以通过`<link>`标签来修改文件名，这个会在下一条进行解释。
- 你可以通过在 head 中添加`<link>`标签来包含 favicon。这种情况下，你可以控制浏览器请求的 URL，可以和预定义的/favicon.ico 完全不同。

  ```
  <link rel="shortcut icon" href="http://CDN/myicon.ico" />
  ```

  这很棒，因为你可以从 CDN 上调用 favicon，并将它设置为“永久”缓存，在整个网站共享相同的文件。但是还要考虑权衡，如果你参照这种做法，Firefox 会在瀑布图的最开始就请求这个 favicon，而不是在其他组件都下载完成之后才下载的。另一方面，如果你从/favicon.ico 调用这个文件，是没有必要添加`<link>`标签的。

- 保持较小的图标大小。ICO 格式可以包含不同分辨率的图像；比如 16×16、32×32 等。这会增加图标文件的大小，所以最佳做法应该是只保留一个 16×16 图像。这样的文件大小通常在 1KB 左右。根据经验，如果你的图标大小大于 1KB，应该还有改善的空间。
- 可以使用免费的 Windows 工具 Pixelformer（*http://www.qualibyte.com/pixelformer/*）来对这种文件进行优化，可以尝试不同的调色板大小。

10.6.4 Apple 触摸图标
Apple Touch Icon

苹果触摸图标和 favicons 类似，用于 iPhone/iPod 设备上。一个 Apple 触摸图标就是一个位于 web 服务器根目录的 PNG 文件，尺寸是 57×57 像素，文件名是 apple-touch-icon.png。同样，如果你想从 CDN 服务器调用这个文件，需要添加一个 far-future `Expires` 头，可以使用`<link>`标签，像下面这样：

```
<link rel="apple-touch-icon" href="http://CDN/any-name.png" />
```

相比于 favicon 来说，桌面浏览器请求这个文件的次数要少得多；iPhone 客户端只会在用户将页面添加到桌面的时候才会请求这个文件。

10.7 总结
Summary

你在本章已经熟悉了很多和图像有关的话题，应该也为下一个图像优化项目做了很好的准备。让我们一起来回顾下重点：

- 首先要选择合适的格式：用 JPEG 保存照片、用 GIF 保存动画，其他所有图像都用 PNG 来保存，并且尽量使用 PNG8。
- 使用 pngcrush 优化 PNG 图像，优化 GIF 动画，并且将你所拥有 JPEG 图像的元数据去除。对于大小超过 10KB 的图像，采用渐进 JPEG 编码。
- 避免使用 `AlphaImageLoader`。
- 使用并优化 CSS Sprite。
- 如果网站不是只有 2~3 个页面，使用模块化的方法建立 Sprite。
- 不要在 HTML 中对图像进行缩放。
- 对生成图像也要进行优化操作。文件一旦生成，就应该被缓存尽可能长的时间。将图像转换为 PNG8 格式，判断 256 种颜色是否可接受。
- 不要忘了 favicons 和 Apple 触摸图标，就算你没有用 HTML 标记来指向它们，它们依然是页面的组成部分，同样要求保持很小的大小，并被缓存。

第 11 章

划分主域

Sharding Dominant Domains

有些网站的 HTTP 请求全部由同一个域提供，也有些网站把资源分布到多个域上。《高性能网站建设指南》的第 9 条谈到减少 DNS 查询，但有时即使以增加更多的 DNS 查询为代价，增加域的数量反而会提高性能，关键是找到提升网页性能的关键路径（critical path）。如果一个域提供了太多的资源而成为关键路径，那么将资源分配到多个域上——我称之为“**域划分**”——可以使页面加载更快。

11.1 关键路径

Critical Path

图 11-1 展示了 eBay 的 HTTP 瀑布图。水平轴表示响应的时间。在图表右边出现了一个陡峭的斜坡，表示在短时间内有大量下载。这正是网页速度快的表现。相反，如图中前 5 个 HTTP 请求所示，平坡表示浏览器由于缓慢的响应或 JavaScript 执行较长的时间而停滞。在这个案例中，eBay 的关键路径被第 1 个 HTML 文档请求，以及第 4、第 5 个 JavaScript 下载请求和执行所阻塞，第 4、第 5 个请求后面的空白区标识 JavaScript 的执行。

图 11-2 展示了 Yahoo!的 HTTP 瀑布图，其关键路径有所不同。加载页面的大部分时间消耗在一次只能下载两张图片上（注 1）。页面中所有的资源都下载自单个域：l.yimg.com，包括 Internet Explorer 6 和 7 在内的一些浏览器将每个服务端的并行下载数限制为两个（Internet Explorer 8 和 Firefox 3 增加到 6 个，详见第 169 页的“新型浏览器”）。每个服务端两个并行下载的限制所带来的影响在图 11-2 中表现得很明显——在任何时候都不会有超过两个资源并行下载，其结果是随着页面加载时间的增长，HTTP 瀑布图呈现为一个阶梯形。

注 1：瀑布图是使用 Internet Explorer 7 生成的。

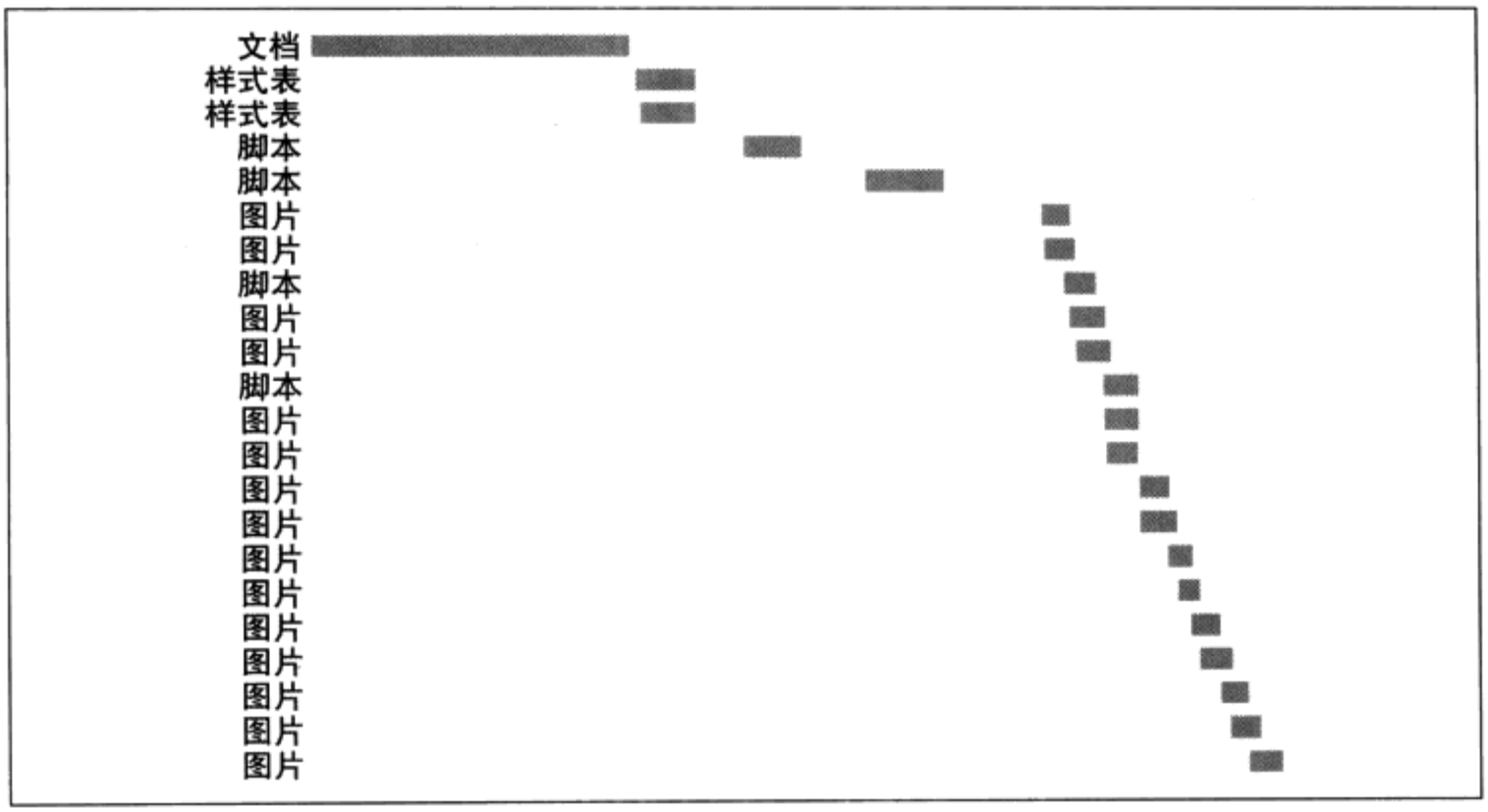
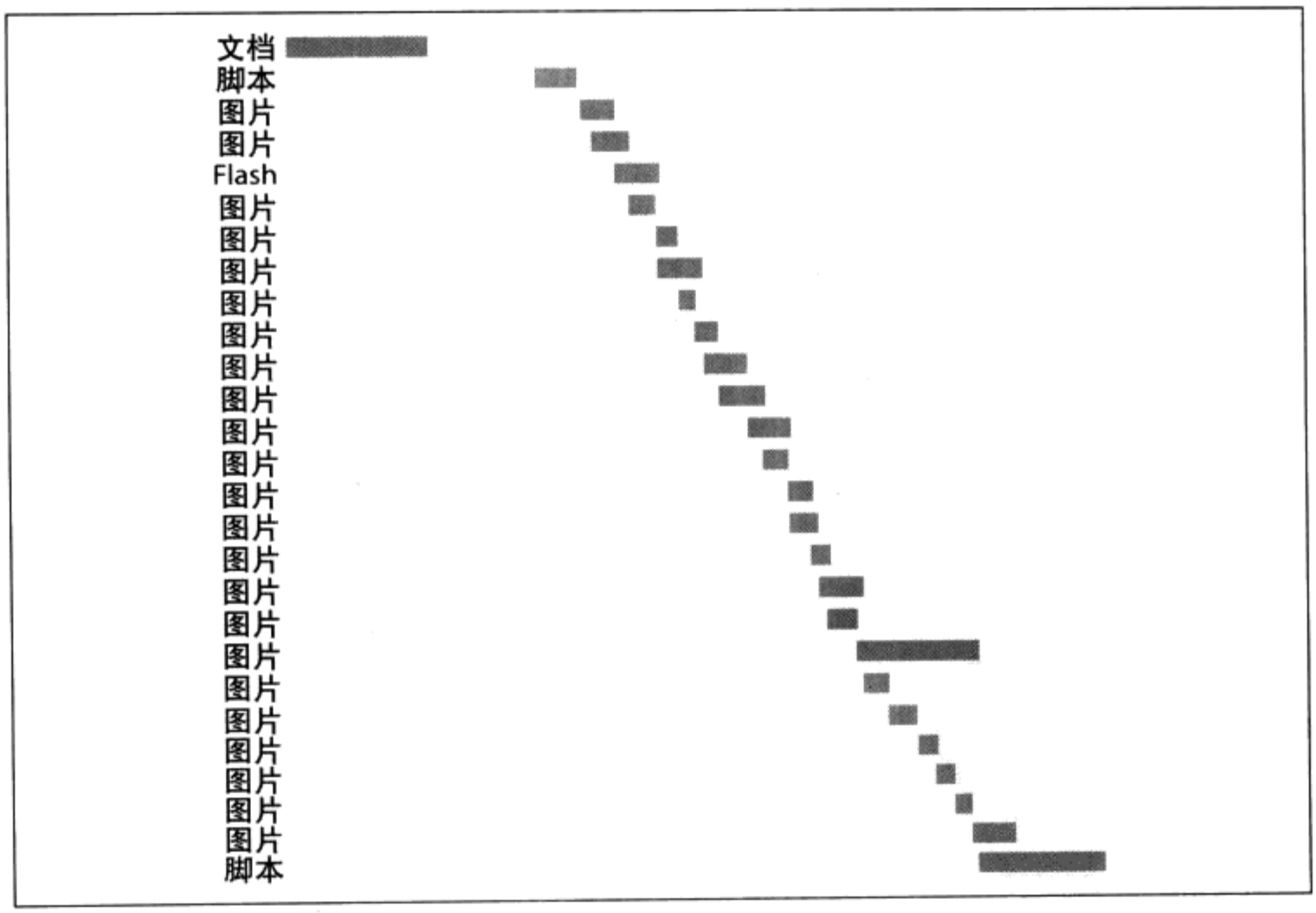


图 11-1：*http://www.ebay.com/*的关键路径


文档
脚本
图片
图片
Flash
图片
图片
图片
图片
图片
图片
图片
图片
图片
图片

当从单个域下载资源成为瓶颈时，可将资源分配到多个域上，通过增加并行的下载数来提高页面速度。下面的例子说明了这一点。

One Domain

http://stevesouders.com/efws/domains1.php

Two Domains

http://stevesouders.com/efws/domains2.php

每个例子都包含了来自 Yahoo!页面的 22 张图片。One Domain 示例从 *l.yimg.com* 上下载所有的图片，而 Two Domains 示例则把图片分配到了两个域上：*l.yimg.com* 和 *d.yimg.com*（注 2）。Two Domains 示例比 One Domain 示例加载快了 27%（在 7000Kbps 的带宽下，654 毫秒对 892 毫秒）。

图 11-3 展示了 One Domain 示例和 Two Domains 示例的 HTTP 瀑布图。在图的上部使用单个域，可以看出在任何时候只有两个资源在下载。在图的下部我们看到 4 个资源同时下载，使得页面加载更快。

11.2 谁在划分主域

Who's Sharding?

表 11-1 展示了一些使用多个域来分配资源的顶级网站，同时也展示了每个网站图片、脚本和样式表的总数。

表 11-1：使用多个域的顶级网站

网站	图片	脚本	样式表	域数量
http://www.aol.com/	59	6	2	3
http://www.ebay.com/	33	5	2	3
http://www.facebook.com/	96	14	14	10
http://www.google.com/search?q=flowers	3	1	0	N/A
http://search.live.com/results.aspx?q=flowers 6	6	1	4	5
http://www.msn.com/	45	7	3	3
http://www.myspace.com/	16	14	2	3
http://en.wikipedia.org/wiki/Flowers	33	6	9	2
http://www.yahoo.com/	28	4	1	1
http://www.youtube.com/	23	7	1	5

注 2：我发现 *http://news.yahoo.com* 从 *d.yimg.com* 上下载图片。

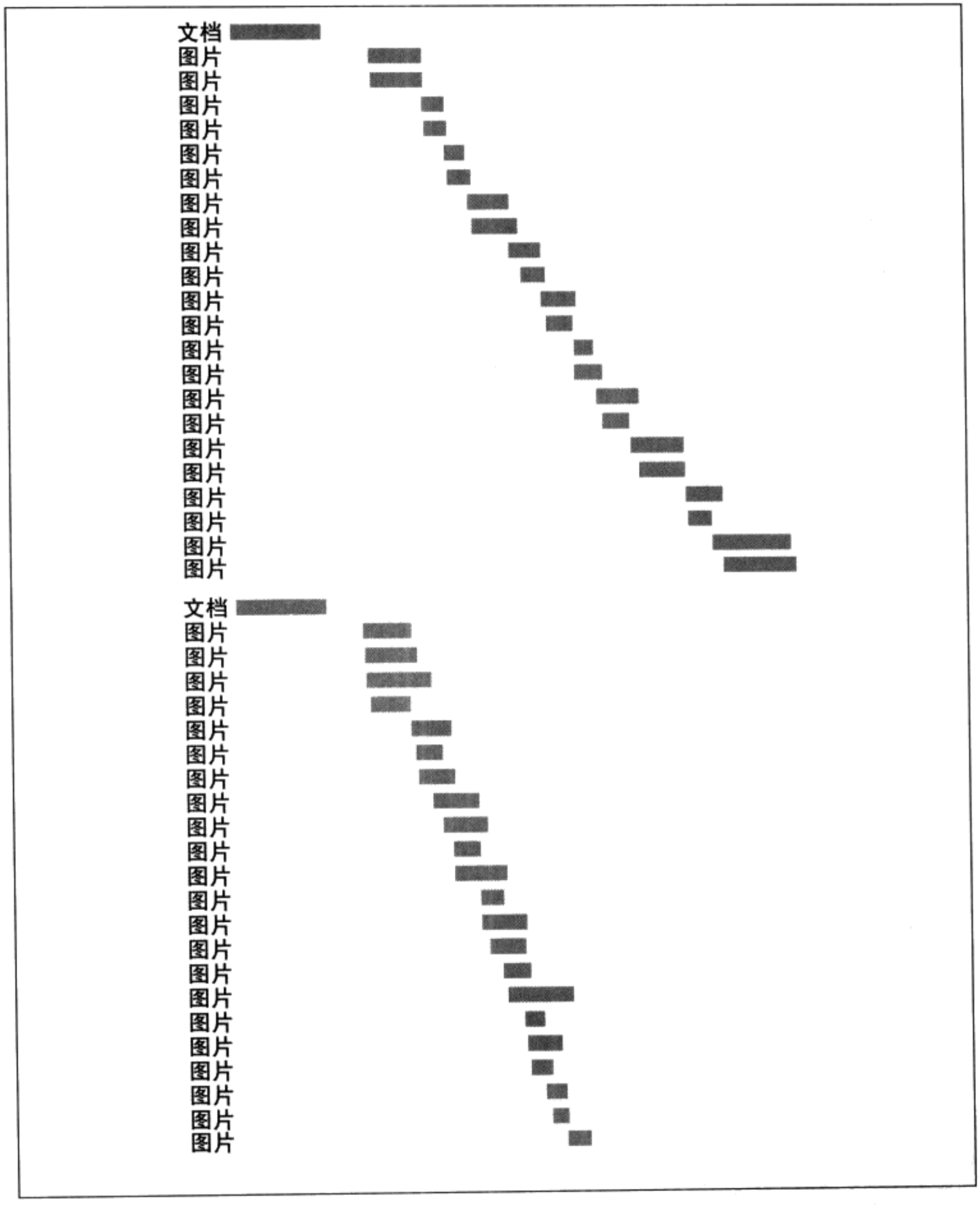


图 11-3：One Domain 与 Two Domains 的对比

这里大多数网站将它们的资源分配在多个域上。很明显这些网站是有意这样做的，例如

YouTube 的序列化域名：*i1.ytimg.com*、*i2.ytimg.com*、*i3.ytimg.com* 和 *i4.ytimg.com*。这些顶级网站中不少网站采用类似的序列划分域：

AOL
: *o.aolcdn.com*、*portal.aolcdn.com*、*www.aolcdn.com*

eBay
: *include.ebaystatic.com*、*pics.ebaystatic.com*、*rtm.ebaystatic.com*

Facebook
: *b.static.ak.fbcdn.net*、*external.ak.fbcdn.net*、*photos-[b,d,f,g,h].ak.fbcdn.net*、*platform.ak.fbcdn.net*、*profile.ak.facebook.com*、*static.ak.fbcdn.net*

Live Search
: *search.live.com*、*ts[1,2,3,4].images.live.com*

MSN.com
: *tk2.st[b,c,j].s-msn.com*

MySpace
: *cms.myspacecdn.com*、*rma.myspacecdn.com*、*x.myspacecdn.com*、*creative.myspace.com*、*largeassets.myspacecdn.com*、*x.myspace.com*

Wikipedia
: *en.wikipedia.org*、*upload.wikimedia.org*

YouTube
: *i[1,2,3,4].ytimg.com*、*s.ytimg.com*

Google 的主页仅包含两个资源。它们能够在一个域上并行下载，所以分成多个域并不合适。Yahoo! 大多数资源从一个域上下载，它应该可通过把资源分配到多个域来优化网站性能。AOL 和 Wikipedia 很有趣，相对于大量的资源来说它们只使用了很少的域，原因之一可能是它们将一些请求从 HTTP/1.1 降级到了 HTTP/1.0。关于这一点的利弊将在下一节讨论。

11.3 降级到 HTTP/1.0
Downgrading to HTTP/1.0

AOL 和 Wikipedia 通过相对少的域来划分它们的资源。虽然如此，它们却实现了高度的并行下载。图 11-4 展示了 Wikipedia 在 Internet Explorer 7 中加载时的 HTTP 瀑布图的开始部分，所有的资源都由一个域提供：en.wikipedia.org。Internet Explorer 7 通常对每个服务端只使用两个连接，但在这里我们看到它使用了 4 个连接，这是因为 Wikipedia 将响应降级到了 HTTP/1.0。

如今大多数 Web 客户端和服务端都使用 HTTP/1.1，但也继续支持 HTTP/1.0。在使用 HTTP/1.1 时，许多浏览器遵循 HTTP/1.1 RFC（*http://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html#sec8.1.4*）中的建议，限制每个服务端两个连接。然而当使用 HTTP/1.0 时，Internet

Explorer 6 和 7 可以打开更多连接，通常每个服务端两个连接的限制会增加到 4 个。类似地，Firefox 2 在 HTTP/1.1 中使用两个连接，但在 HTTP/1.0 中增加到 8 个。

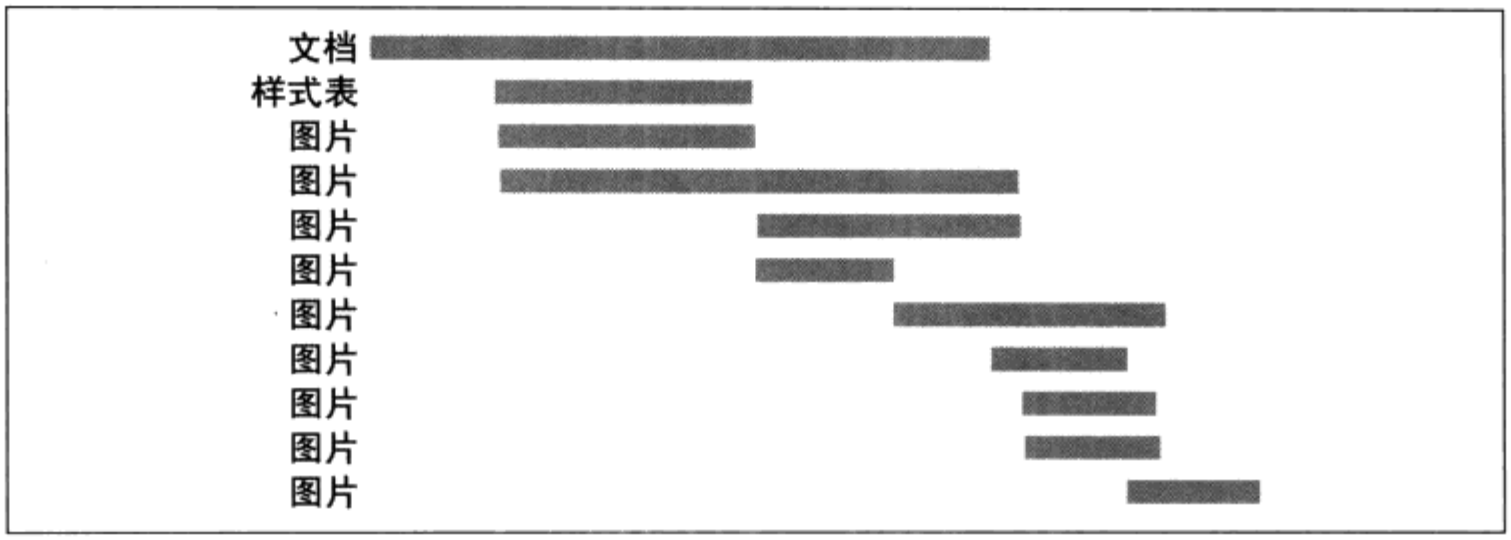


图 11-4：Wikipedia 的并行下载

由于**长连接**的原因，HTTP/1.1 建议每个服务端提供少量的连接数，默认情况下 HTTP/1.0 在每次响应后会关闭 TCP 连接，为每次请求建立一个新的 TCP 连接会消耗时间。为了降低这种开销，HTTP/1.1 使用了长连接，并且使用单个连接来完成多个请求和响应。长连接通常保持长时间打开着，从而导致拥有一定数量可用连接的服务端负载增高，因此，HTTP/1.1 中建议每个服务端的连接数减少到 2 个。

通过降级到 HTTP/1.0，AOL 和 Wikipedia 实现了更高程度的并行下载，但这些好处是以放弃长连接为代价的。或者采用下面的方式作为长连接的替代方案，HTTP/1.0 支持 Keep-Alive 选项来重用现有连接。HTTP/1.0 的 Keep-Alive 和 HTTP/1.1 的长连接有些不同之处，但差异比较小：

- 长连接在 HTTP/1.1 中是默认的。一旦 HTTP 版本指定为“HTTP/1.1”，就无需额外的头信息来声明支持长连接。但 Keep-Alive 在 HTTP/1.0 中不是默认的，客户端和服务端必须同时发送 `Connection: Keep-Alive` 头信息。
- 在通过代理使用 HTTP/1.0 的 Keep-Alive 时，连接存在一些风险。代理不能理解 `Connection: Keep-Alive` 头信息，会不加区分地把它转向原始服务端。这可能导致代理创建一个挂起的连接并等待原始服务端来关闭。而原始服务端不会关闭该连接，因为它创建的是一个 Keep-Alive 连接。因此，客户端必须确保在与代理会话时不发送 `Connection: Keep-Alive`，实际上所有主流浏览器都是这样做的。

- HTTP/1.0 Keep-Alive 响应必须使用 `Content-Length` 头信息来标示单个连接中不同响应之间的结束分界点。这意味着在响应开始时动态内容总长度未知的情况下，无法使用 HTTP/1.0 Keep-Alive。
- HTTP/1.1 中引入了块传输编码，但在 HTTP/1.0（*http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html#sec3.6.1*）里不能使用。块编码支持服务端成块回传数据。这主要应用于动态生成大的响应的情况，在这种情况下即使不知道响应内容的总大小，服务端也能够开始发送响应内容。（关于块编码更多的讨论可见第 12 章）

对静态内容来说，降级到 HTTP/1.0 所带来的这些差异不存在任何大的缺点。主流浏览器已经能发送 `Connection: Keep-Alive` 头信息，并在使用代理时去掉它。静态内容的大小在请求开始时就能知道，所以总是可以使用 `Content-Length` 头而不需要块编码。对较大的资源也可使用块编码，例如一个 500KB 的脚本，这样能使下载和解析更快，但实际上没有一个顶级网站针对静态内容使用了块编码。

使用 Internet Explorer 6 或 7 浏览 AOL 和 Wikipedia 的用户则通过降级到 HTTP/1.0 而受益。得益于 Keep-Alive，他们可以一次下载 4 个资源并反复重用 TCP 连接。然而其他大多数浏览器并没有基于 HTTP 版本来增加服务端的连接数，见表 11-2。

表 11-2：服务端的连接数

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	6
Opera 9,10	4	4

如果有大量的 Internet Explorer 6 和 7 用户，可以考虑降级到 HTTP/1.0。这样做增加了并行下载数（针对 Internet Explorer 6 和 7）而无需额外的 DNS 查询成本。但如果你想使所有用户都能通过提高并行数受益，那么首选方案还是划分主域。

11.4 域划分的扩展话题
Rolling Out Sharding

在考虑将资源划分到多个域的时候会有几个典型的操作问题。

11.4.1 IP 地址和主机名
IP Address or Hostname

浏览器执行“每个服务端最大连接数”的限制是根据 URL 上的主机名，而不是解析出来的 IP 地址，如 Different Hostnames, Same IP 示例所示。

Different Hostnames, Same IP
http://stevesouders.com/efws/hostnames.php

该例子里有 4 张图片：两张来自 stevesouders.com，两张来自 *www.stevesouders.com*。这两个主机名具有相同 IP 地址。当用 Internet Explorer 6 和 7 加载时，4 张图片并行下载。浏览器把每个主机名看作一个单独的服务端，因此为每个主机名打开两个连接，即使这两个主机名解析为相同的 IP 地址。

这对那些想把内容分配到多个域的人来说是个好消息，他们不必额外部署服务器，而是为新域建立一条 CNAME 记录。CNAME 仅仅是域名的一个别名。即使域名都指向同一个服务器，浏览器依旧会为每个主机名开放最大连接数。

11.4.2 多少个域
How Many Domains

在第 161 页的“关键路径”里，你可以看到将内容分配到两个域要比一个域好。那 3 个域是否会比两个好呢？10 个呢？Yahoo! 发表的研究（*http://yuiblog.com/blog/2007/04/11/performance-research-part-4/*）表明域从一个增加到两个性能有所提高，但数量超过两个时反而对加载时间有负面影响。最终数量取决于资源的大小和数量，但划分为两个域是个很好的经验。

11.4.3 如何划分资源
How to Split Resources

假定有一个具体的资源，那么在多个可能的域中分配它的最佳算法是什么？所有分配算法的关键点是让这项具体的资源始终放置在同一个域下。这可以确保，如果资源已经被缓存，后续请求的 URL 能匹配缓存中的 URL。

有一种实现方式是使用哈希函数把资源的文件名转换为一个整数，根据这个整数来选择域。另一种方式是按照资源的类型来选择域。例如，样式表和图片可以分配到域 1 上，所有其他类型资源分配到域 2 上。这可能导致域上资源分配不均衡，而实际上在样式表和脚本从域 1 下载的同时，让图片从域 2 开始并行下载更有意义。

11.4.4 新型浏览器
Newer Browsers

Internet Explorer 8 和 Firefox 3 都已经把每个服务端的连接数从 2 增加到了 6。老式浏览器中力求增加并行下载数量的做法可能会导致在下一代浏览器中并行下载数过多，如果浏览器打开过多的连接可能会使服务端压力过高而降低客户端加载的性能。

那些使用多域的网站，例如 Facebook 和 YouTube，可能需要基于浏览器的类型来改进资源划分算法。如果你选择把静态资源分配到多个域上，那请遵循只划分两个域的准则。这是一种能平衡当今浏览器和未来浏览器提升性能的方式。

第 12 章

尽早刷新文档的输出

Flushing the Document Early

性能优化黄金法则提醒我们关注前端性能的提升，因为网页加载过程中的绝大部分时间都消耗在前端上（注 1）。偶尔也有例外，那就是后端在生成 HTML 文档时消耗了太长的时间，比如在返回 HTML 内容之前需要频繁进行的数据库查询，或者等待其他 Web 服务的响应。

不幸的是，在后端处理数据的过程中，客户端的所有内容都被冻结了。本章将向大家介绍如何在 HTML 文档完全加载之前就开始进行页面渲染，而不是让浏览器处于空闲状态，同时也避免让用户一直等待。

12.1 刷新文档头部的输出

Flush the Head

大多数情况下，浏览器会在 HTML 文档加载完毕后才开始渲染页面，同时下载页面上的资源。下面这个 Simple Page 示例就是这样：

Simple Page 示例

http://stevesouders.com/efws/simple.php

这个示例页面包含两张图片和一段脚本。HTML 文档和这 3 个资源都被设置为需要两秒才加载。图 12-1 就是这个页面对应的 HTTP 瀑布图。意料之中的，HTML 文档最先下载，浏览器会在 HTML 文档下载完毕后开始解析，首先渲染前几行文字，同时开始下载页面上的其他资源。

注 1：参见《高性能网站建设指南》的第 1 条规则。

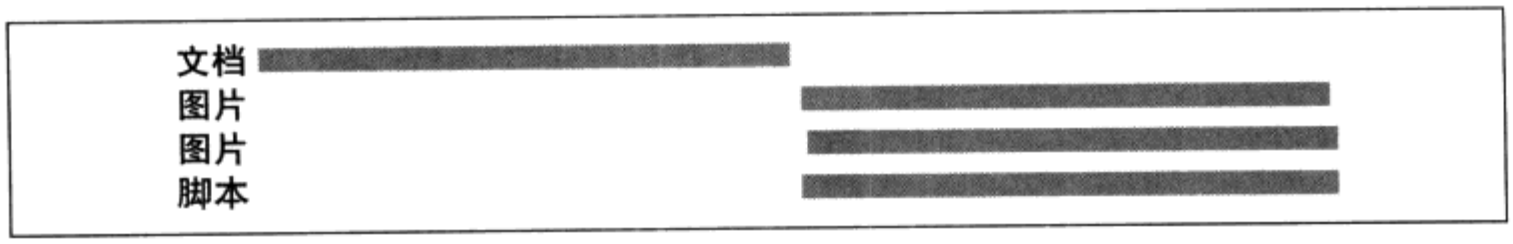


图 12-1：Simple Page 示例的 HTTP 瀑布图

保存在 1.cuzillion.com 上的两张图片是并行下载的。脚本的下载过程也是并行的，因为它在图片后面加载，并且保存在另外一个主机——2.cuzillion.com 上。页面加载的总时间是 4 秒。

图 12-2 展示的是同一个页面，但是这次图片和脚本在页面加载完成之前就开始下载，结果页面的加载时间总共才用了 2 秒，是之前示例的一半。

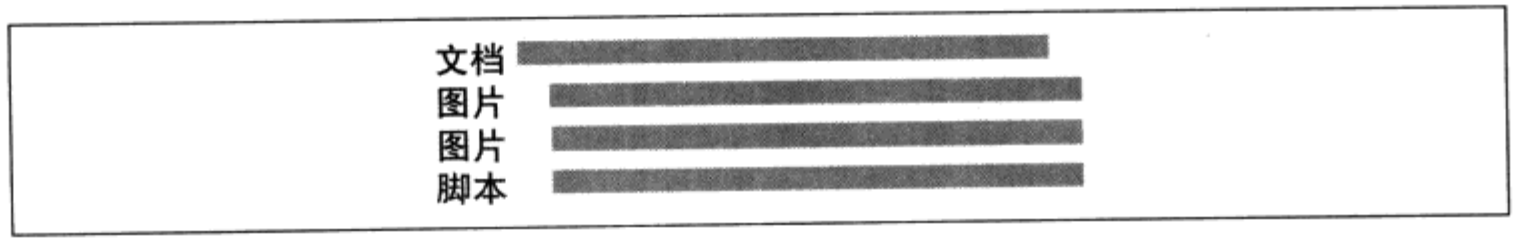


图 12-2：Flush 示例的 HTTP 瀑布图

速度之所以提升，是因为调用了 PHP 的 flush 函数（译注 1）。当运行这个 Flush 示例时，你会发现这个页面的加载速度的确比前面的示例快了许多。

Flush 示例

http://stevesouders.com/efws/flush-nogzip.php

为了理解 flush 的工作原理和随之而来的并发问题，我们需要了解 HTML 文档的生成过程（注 2）。当服务器解析 PHP 页面的时候，所有的输出都被写入 STDOUT（译注 2）。每次写入一个字符、一个单词或一行文字，服务器不会立即将它们输出，而是把所有输出内容排到一个队列中，然后再以较大的数据块发送到浏览器。这样做更加有效，因为它会使从服务器发送到浏览器的数据包数量更少。由于发送数据包会引起网络延迟，所以通常发送少量大数据包的效果要比发送大量小数据包好。

调用 flush()会将所有排在 STDOUT 队列中的内容立刻发送出去。但是简单地刷新 STDOUT 还是远远达不到上一个例子中所带来的高速体验，我们还必须在适当的时机调用 flush()。让我们一起来看看 Flush 示例中的 PHP 源码：

译注 1：flush 在 PHP 中用于刷新写入缓冲区，它会将当前程序的所有输出都发送到浏览器。详情请看 *http://php.net/manual/en/function.flush.php*。

注 2：虽然这个讨论是基于 PHP 的，但这些概念同样适用于其他模板框架。

译注 2：在程序执行时，程序和它的环境之间有事先连接好的输入输出频道，被称为标准流。STDOUT 就是其中之一，它是 Standard Output 的简写，中文名称是标准输出。更多信息请参考 *http://zh.wikipedia.org/wiki/*标准流。

```
<html>
<body>

<p>
This is the Flush example.
</p>

<img src="http://1.cuzillion.com/...">
<img src="http://1.cuzillion.com/...">
<script src="http://2.cuzillion.com/..." type="text/javascript"></script>

<?php
flush();
long_slow_function();
?>

<p>
This sentence is after the long, slow function.
</p>

</body>
</html>
```

需要注意的是，当生成 HTML 文档的时间过长时，才有必要使用本章提到的技巧。在上面这段 PHP 代码中，我们是通过 `long_slow_function` 这个函数来模拟 2 秒的延迟，并在这个漫长的后端延迟之前调用了 `flush()`。

这种方式提速是否有效，取决于 `flush()` 函数之前所包含的 HTML 代码。在这个例子中，有一行文本内容（“This is the Flush example”）和 3 个资源文件（两张图片和一段脚本）。这个加载很慢的 HTML 文档有两个需要处理的缺陷：阻塞渲染和阻塞下载。通过将这行文本插入经过刷新输出的 HTML 中，用户就可以通过直观的视觉反馈，看到页面加载的过程。通过将 3 个资源文件插入刷新输出中，浏览器会在等待剩余 HTML 文档的同时开始下载资源。这是本章主要阐述的性能观点，刷新输出的最大好处就是可以提前加载页面资源。

上面提到的这些看起来真地很简单，但当阅读了 flush 文档（*http://www.php.net/flush*）中的评论后，你就会发现它其实并没有看起来那么简单。

12.2 输出缓冲
Output Buffering

使用 PHP 刷新输出，最令人感到困惑的可能就是输出缓冲了。如前所述，PHP 的输出就是向 STDOUT 中写数据，而输出缓冲则在数据到达 STDOUT 之前建立了另外一个队列。

首先我们要确定，在 PHP 配置中是否开启了输出缓冲功能，以及具体设置了多大的输出缓冲。这是通过在 php.ini 中配置 `output_buffering` 指令来控制的（注 3）。令人倍感困惑的是这个指令的默认值在 PHP 4.3.5 中发生了改变，在这个版本之前，输出缓冲默认是开启的，缓冲设置为 4096 字节，在 php.ini 中应该可以找到下面这行配置：

```
output_buffering = 4096
```

从 PHP 4.3.5 开始，输出缓冲的默认值修改为禁用状态：

```
output_buffering = 0
```

你可以用下面这段 PHP 代码获取 `output_buffering` 的值和 PHP 的版本：

```
<?php
echo "<br>output_buffering = " . ini_get('output_buffering');
echo "<br>PHP version = " . phpversion();
?>
```

当服务器开启了输出缓冲，除了要使用 flush，还必须使用 `ob_flush` 等相关函数，我们将在下面这个 Flush Output Buffering 示例中说明：

Flush Output Buffering 示例

http://stevesouders.com/efws/ob/flush-nogzip-ob.php

下面就是这个示例的源代码，其中加粗的部分为新增的 PHP 代码。最大的变化是，我们增加了对 `ob_start` 和 `ob_flush` 两个函数的调用。`ob_flush` 会将输出缓冲中的内容刷新到 STDOUT，然后 `ob_start` 打开新的输出缓冲。输出缓冲被刷新后，我们还需要调用 `flush()` 来刷新 STDOUT：

```
<?php
while (ob_get_level() > 0) {
    ob_end_flush();
}
ob_start();
?>
<html>
<body>

<p>
This is the Flush Output Buffering example.
</p>

<img src="http://1.cuzillion.com/...">
<img src="http://1.cuzillion.com/...">
<script src="http://2.cuzillion.com/..." type="text/javascript"></script>

<?php
ob_flush();
flush();
```

注 3：*http://www.php.net/manual/en/outcontrol.configuration.php#ini.output-buffering*。

```
long_slow_function();
?>

<p>
This sentence is after the long, slow function.
</p>

</body>
</html>
```

只要 ob_get_level()的返回值大于 0，while 循环就会一直调用 ob_end_flush，很多开发者都会在无意中遗忘这个步骤。这个循环可以确保所有打开的输出缓冲都被刷新和删除。否则，ob_start 打开的可能不是唯一的输出缓冲。PHP 中的输出缓冲是用堆栈存储的。如果我们调用 ob_start 打开了第 2 个输出缓冲，随后调用 ob_flush 会将第 2 个输出缓冲的内容刷新到第 1 个输出缓冲，而不是刷新到 STDOUT，所以这时调用 flush 是没有效果的，因为 STDOUT 此时是空的。确认是否开启了输出缓冲功能，如果开启了，就要使用 ob_系列函数来解决这些问题。

12.3 块编码

Chunked Encoding

当 Web 服务器或客户端只支持 HTTP/1.0 的时候，上面这几个示例对速度的提升非常有限，因为它们不支持块编码。

HTTP/1.0 的响应是作为一整块数据返回的，它的大小是由 Content-Length 头来发送的。浏览器需要知道数据的大小，这样才能确定响应的结束时间。因为 HTML 文档是作为一个整块来发送的，所以浏览器在响应结束之前，不会开始渲染页面和下载资源（注 4）。

HTTP/1.1 引入了 Transfer-Encoding: chunked 响应头（注 5）。通过块**编码**，HTML 文档可以被分成多个数据块返回，每个响应的数据块都以标识其大小的指示符为开头。这就允许浏览器在下载数据包后马上进行解析，使得页面的加载速度更快。

块编码还可以在另外两个方面为网页提速，而且都和动态页面相关。如果不使用块编码，响应必须包含一个 Content-Length 的头信息。这就意味着服务器在将整个响应组合在一起，并计算出其大小之前，是不会开始发送响应信息的。通过块编码，服务器可以尽早发送响应，因为它只需要知道每个发送块的大小即可。

注 4：当 Content-Length 未知的情况下，有一种替代方案是让服务器来关闭连接，但这样做会失去持久连接的好处。

注 5：*http://www.w3.org/Protocols/rfc261 6/rfc2616-sec14.html#sec14.41*。

通过使用 Trailer 头，块编码可以带来另外一项性能提升（注 6）。某些情况下，我们无法在创建 HTML 文档之前知道是否需要一个头，或者它的内容具体应该是什么。比如说，只有在生成 HTML 文档的过程中，才会根据数据库的查询结果或 web 服务的请求结果来决定 Cookie 或 ETag 响应头的内容。

通常，这些头必须在响应的最开始发送，这就意味着服务器在这些耗费时间的数据库查询或 Web 服务调用结束之前，是无法开始发送响应的。然而，当使用块编码后这些头就可以被延后发送。最开始的块会被立刻发送，在这个块中会通过 Trailer 头来列举出那些将会延迟发送的头：

```
Trailer: Cookie
Trailer: ETag
```

通过这种方式，Cookie 和 ETag 头可以被包含在 HTML 响应的尾部发送过来（注 7）。

块编码使立即发送部分 HTML 文档成为可能，即使文档的总大小和部分头还处于未知状态也可以。为了从提早刷新文档的输出中获益，需要确保块编码机制是可用的。幸运的是，Apache 和其他 Web 服务器已经替你完成了这项工作。如果想尝试使用刷新输出，请确保 HTML 文档的响应头中包含 Transfer-Encoding: chunked。

12.4 刷新输出和 Gzip 压缩
Flushing and Gzip

在之前的例子中，HTML 文档都是没有经过 Gzip 压缩的。对于所有的网站来说，使用 Gzip 压缩 HTML 文档都是至关重要的，但这增加了刷新文档输出的复杂度。如果对之前的例子使用了压缩，那么刷新输出将不再生效，正如下面这个例子所示。

Flush Gzip No Padding 示例

http://stevesouders.com/efws/flush-gzip-no-padding.php

当 Apache 启用了压缩功能时，它的缓冲输出机制将会阻止刷新。Apache 2.x 使用 mod_deflate 来进行压缩（注 8）。这个模块的缓冲默认设置为 8096 字节。你可以使用 DeflateBufferSize 指令来减小这个缓冲的大小，除此，你还可以通过向 HTML 文档填充超过 8K 的内容（在压缩以后）来实现刷新输出，如下所示：

注 6：*http://www.w3.org/Protocols/rfc261 6/rfc2616-sec14.html#sec14.40*。

注 7：截至发稿时，浏览器对 trailer 的支持程度还有很大的不同，为了实现跨浏览器的支持，我们需要做更多的研究和推广。

注 8：*http://httpd.apache.org/docs/2.0/mod/mod_deflate.html*。

Flush Gzip Padding 示例

http://stevesouders.com/efws/flush-gzip-padding.php

增加填充可以将 mod_deflate 的缓冲塞满，以便向浏览器发送数据。这种填充是有技巧的，在 PHP 中通常可以使用 `str_pad` 函数实现：

```
echo str_pad('', 20000);
```

在有些情况下这样操作是无效的，比如当启用了压缩后，增加的填充内容会被压缩，少于 mod_deflate 的缓冲大小（8KB）。Flush Gzip Padding 示例使用了一个 20KB 无重复的字符串，这样就算压缩，也不会少于 8KB，从而保证刷新输出的正常进行。

向页面增加 20KB 的内容是一个很高的代价。幸好 Apache 2.2.8 及以后的版本修复了这个问题，不再需要使用这个填充技巧。截至发稿时，我的网站托管商还在使用 Apache 2.0。但我将上面介绍的技巧在 Apache 2.2.8 上进行了测试，确认刷新输出在经过压缩的页面同样生效，就算没有使用填充技巧时也一样。

12.5 其他障碍
Other Intermediaries

代理软件和杀毒软件是两个潜在的中间环节，会阻碍通过刷新输出提升性能。如果这些中间环节被用来过滤内容，而不是用来传送经过刷新输出的数据，它们可能会等待整个响应加载完毕，并在扫描之后才传送给 Web 客户端。

还有一种情况和代理有关，就是代理会将所有的响应降级为使用 HTTP/1.0 处理，由于 HTTP/1.0 不支持块编码，刷新输出在使用代理的时候就无法生效，比如 Squid 代理服务器（译注 3），它的 Wiki 表明 Squid 尚不支持 HTTP/1.1，其中一个主要原因就是“块编码丢失”（注 9）。

我见过有些开发者花了很长时间去调试刷新输出不生效的原因，但只有当他们意识到是通过公司的代理上网的，才有可能解决这个问题，因为代理会阻塞刷新输出的进行。判断网络是否使用了代理，你可以检查浏览器的网络连接设置。有时候很难判断你是否设置了代理服务器，尤其是当设置了“自动检测”（Internet Explorer）和“自动检测代理服务器设置”（Firefox 3.0）时。除了检查浏览器的设置，我还会在 HTML 文档的响应中寻找类似 `Proxy-Connection`、`X-Forwarded-For`、`Via` 和包含“HTTP/1.0”状态的头信息。如果存在这其中任何一种情况，你就有可能是通过代理访问的，而这或许会阻碍刷新输出的正常工作。

译注 3：Squid 是 Squid Cache 简称，它是一个流行的自由软件（GNU 通用公共许可证）的代理服务器和 Web 缓存服务器。更多详细内容请看 *http://zh.wikipedia.org/wiki/Squid_cache*。

注 9：*http://wiki.squid-cache.org/Features/HTTP11*。

12.6 刷新输出时的域阻塞

Domain Blocking During Flushing

主流的浏览器，比如 Internet Explorer 6 和 7，还有 Firefox 2，只能对每个服务器建立两个连接。这会限制从相同域并行下载的文件数量。正如第 11 章所介绍的，有很多种方式可以突破这个限制。大多数和域阻塞有关的情况，都与页面上资源的上下文有关。关于刷新输出，我们还需要注意的是 HTML 文档的请求会影响并行下载。

在之前演示的例子中，我很小心地将两张图片保存在和主页（*stevesouders.com*）不同的域上（*1.cuzillion.com*）。为什么这很重要？在只有两张图片的情况下，就算在 Internet Explorer 7 中（只会为每个服务器建立两个并行链接）也不会出现什么阻塞的情况。

这听起来不错，但当我们看到结果的时候，就可以理解这样做的原因，因为我们使用了块编码，HTML 文档的响应还占用着一个连接，所以留给同一域下其他资源的连接数只有一个。下面这个 Flush Domain Blocking 示例演示了这种情况，在这个例子中的两张图片都是从 *stevesouders.com* 上下载的。

Flush Domain Blocking 示例

http://stevesouders.com/efws/flush-domain-blocking-nogzip.php

图 12-3 显示的是在 Internet Explorer 7 中加载上面例子的 HTTP 瀑布图，将它和图 12-2 比较就会发现，这个页面几乎要比上个例子慢了两倍，就算刷新输出已经生效也是如此。缓慢的原因是第 2 张图片在 HTML 文档加载之前被阻塞了。HTML 文档和第 1 张图片已经用掉了 stevesounders.com 对应的两个连接，所以第 2 张图片不得不继续等待。

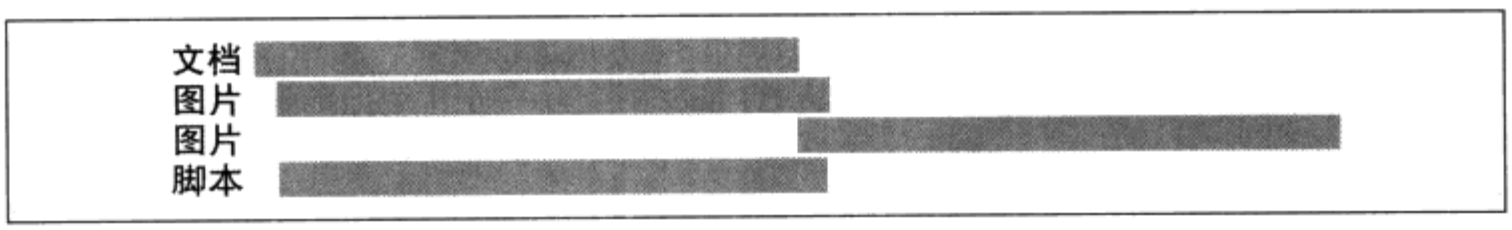


图 12-3：Flush Domain Blocking 示例的 HTTP 瀑布图

如果你想尽可能早地下载资源，发挥刷新输出的最大效果，可以很容易突破同一服务器两个连接的限制。记住，你可能需要将初始化的资源转移到不会被 HTML 文档阻塞的域上。

12.7 浏览器：最后的障碍

Browsers: The Last Hurdle

看到这里，似乎感觉让刷新输出正常工作不是很难，但当你使用 Safari 或 Chrome 浏览上面这些例子，会再一次产生困惑。虽然这两个浏览器都支持块编码，但是它们在接收数据

未达到某个最小要求之前不会开始渲染：Safari 是 1KB 左右，Chrome 是 2KB 左右（注 10）。

Flush 示例的第 1 个数据块中只有 600 字节，所以在 Safari 和 Chrome 中都看不到刷新输出效果。为了让它在 Safari 中工作，Flush 1K 示例向初始块中额外添加了 1KB 的 HTML 内容。同样地，Flush 2K 示例额外添加了 2KB 的 HTML 内容，这样它在 Chrome 中也会生效了。

Flush 1K 示例

http://stevesouders.com/efws/flush-nogzip-1k.php

Flush 2K 示例

http://stevesouders.com/efws/flush-nogzip-2k.php

在我的示例中之所以出现这个问题，是因为 HTML 内容太少了。然而在真实的网页中充斥着行内样式和脚本块，以及更多的 HTML 标记，所以经过刷新输出的 HTML 肯定会超过 2KB。为了让刷新输出过的网页在所有浏览器中都生效，请确保经过刷新输出后的页面尺寸超过 2KB。

12.8 不借助 PHP 进行刷新输出
Flushing Beyond PHP

和大部分有关刷新输出的 Wiki 与论坛一样，本章内容也是基于 PHP 来实现的。就算使用了不同的 HTML 框架也不用担心，因为很可能也有通过块编码提升性能的方法，而且通常都是通过名为 flush 的函数来实现的。

有经验的 Perl 程序员已经写出了在需要输出时立即刷新 STDOUT 的脚本。这些爱好者知道通过把$|设置为非零值的方式去实现这种效果，但很少有人知道使用 `FileHandle` 的 `autoflush` 方法也可以（注 11）。下面这个 Flush Perl 示例就使用了这项技术。

Flush Perl 示例

http://stevesouders.com/efws/flush-nogzip.cgi

在脚本的顶部调用了 `autoflush`。

```
use FileHandle;
STDOUT->autoflush(1);
```

Python 的文件对象（注 12）和 Ruby 的 IO 类（注 13）都有 `flush` 函数。所以不用在意后端到底使用什么语言，总会找到一种方法来刷新 STDOUT。

注 10：Internet Explorer 也有一个类似的最小值要求，但是只有 255 字节，所以基本不会产生影响。

注 11：*http://perldoc.perl.org/FileHandle.html*。

注 12：*http://www.python.org/doc/2.5.2/lib/bltin-file-objects.html*。

注 13：*http://www.ruby-doc.org/core/classes/IO.html#M002303*。

12.9 刷新输出问题清单
The Flush Checklist

让刷新输出生效并不总是一件容易的事情。如果你尝试在 PHP 页面上刷新输出，却遇到一些问题，下面的清单可以帮忙：

- 输出缓冲是否打开？如果打开了，你需要使用 `ob_` 系列函数。
- 你是否看到了 `Transfer-Encoding: chunked` 响应头？块编码是刷新输出生效的必要条件。
- 响应是否经过了 Gzip 压缩？如果压缩了，并且 Apache 的版本低于 2.2.8，那么你将不得不向页面填充额外的内容。
- 你是否使用了代理软件或杀毒软件？这些软件可能会在内容发送到浏览器之前，将内容缓冲起来。
- 在经过刷新输出的数据块中，是否有一些资源由于和 HTML 文档处于相同的域而被阻塞了？
- 你是否只在 Safari 和 Chrome 中做了测试？如果是这样，经过刷新输出的 HTML 只有大于 2KB 时，才能在这些浏览器上看到效果。

有很多种情况需要理清，这就像要识别天上的繁星一样，但结果是值得的。在排名前 10 的网站中，有 5 个使用了块编码：AOL、Facebook、Google Search、MySpace 和 Yahoo!。不过要记住，即使这些网站支持块编码，它们也未必就会提早刷新文档的输出。图 12-4 显示的是 Google Search 的 HTTP 瀑布图，很明确地说明了刷新输出带来的好处。

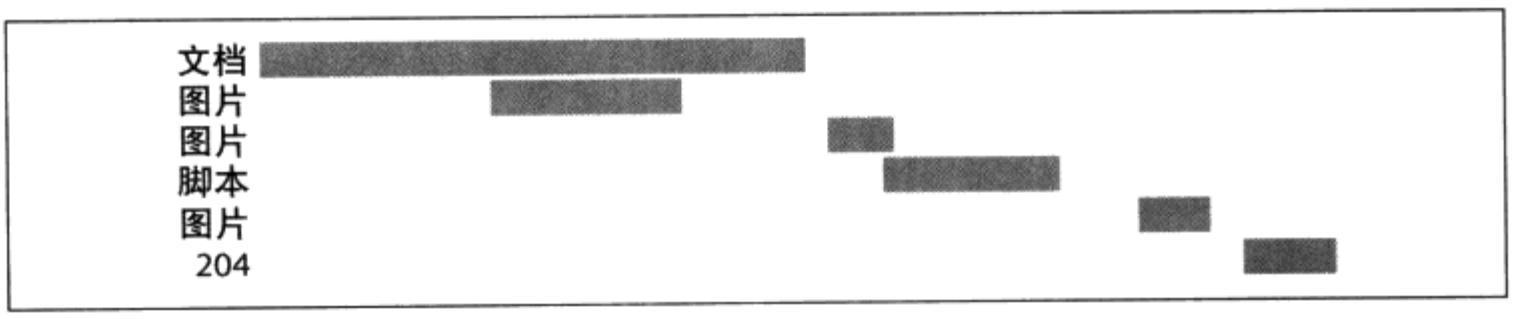


图 12-4：Google Search 的 HTTP 瀑布图

通过尽早刷新文档的输出，Google 页面开始下载资源和渲染页面的速度都比以前更快。所有用户都能感受到性能的提升，尤其是那些网速很慢且延迟很高的用户。

第 13 章

少用 iframe

Using Iframes Sparingly

iframe 也叫内嵌 frame，它允许把一个 HTML 文档嵌入另一个中（注 1）。iframe 常用于整合诸如广告这样的 HTML 内容，它们一般来自和主页不同的站点。

使用 iframe 的好处是它们的文档完全独立于其父文档。iframe 中的相对 URL 是相对于其基准 URI，而非其父文档的 URI。用户代理（译注 1）可以把焦点定位到 iframe 上，从而对其进行打印、收藏、保存等。最重要的或许是，iframe 中包含的 JavaScript 访问其父文档是受限的。例如，来自不同域的 iframe 不能访问其父文档的 Cookie。当 Web 开发者必须允许类似广告这样的第三方内容出现在他们的页面上却又不能控制这部分内容时，要着重考虑这个因素。

它有什么缺点呢？你可能会猜到——性能较低。第 4 章描述了如何使用 iframe 实现脚本的异步加载来提高性能。如果使用得当，iframe 确实能让页面加载更快。不幸的是，我们常常以损害性能的方式来使用 iframe，所以知道 iframe 造成的性能损失和如何避免它们是非常重要的。

13.1 开销最高的 DOM 元素

The Most Expensive DOM Element

Cost of Elements 测试页对创建不同类型 DOM 元素的耗时进行了测量。

Cost of Elements

http://stevesouders.com/efws/costofelements.php

我使用这个页面测试了加载 100 个如下类型元素的耗时：A、DIV、SCRIPT、STYLE 和 IFRAME。每个测试都分别在 Chrome（1.0、2.0）、Firefox（2.0、3.0、3.1 beta2）、Internet Explorer

---

注 1：*http://www.w3.org/TR/html4/present/frames.html#edef-IFRAME*。

译注 1：大部分情况下是指浏览器。

（6、7、8 beta2）、Opera（9.63、10.00 alpha）和 Safari（3.2、4.0 developer preview）下运行了 10 次。图 13-1 展示了创建 100 个每种类型元素的平均耗时。

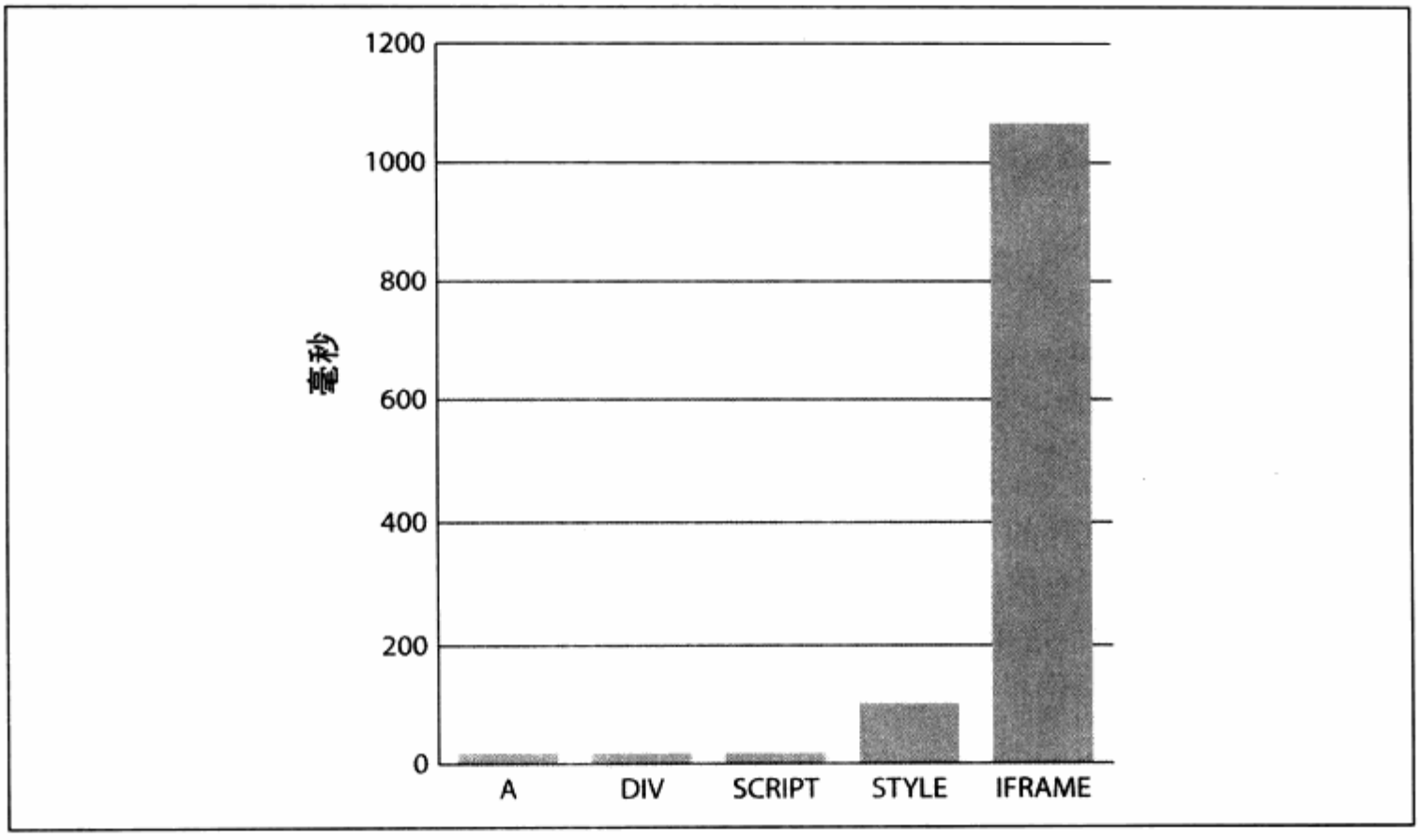


图 13-1：加载 100 个不同类型元素的耗时

图 13-1 表明了创建 iframe 的开销比创建其他类型的 DOM 元素要高 1~2 个数量级。在这些测试中，DOM 元素是空的。加载大的脚本或样式块可能比加载某些类型的 iframe 耗时更长，但这次测试对比的是基准开销。鉴于使用 iframe 的高开销，应该谨慎地少量使用。

13.2 iframe 阻塞 onload 事件
Iframes Block Onload

我们期望尽可能快地触发 onload 事件，出于如下几个原因：

- 当 onload 事件触发时，浏览器停止“忙指示器”，并向用户反馈页面已经准备就绪。例如，在状态栏显示“完成”，因此，如果 onload 事件迅速触发，那么用户感觉到页面加载很快的几率就越大。
- 开发者经常在 onload 事件触发时初始化 UI 操作。例如，设置登录区域的焦点。因为用户习惯等待这个操作，所以尽可能快地让 onload 事件触发从而使用户的等待时间变短是很重要的。

- 开发者有时把一些重要的操作和 window 的 unload 事件绑定在一起。例如，减少内存泄漏的 JavaScript 代码（注 2）。不幸的是，在一些浏览器中，只有当 onload 事件触发后 unload 事件才能触发（注 3）。如果 onload 花费太长的时间，用户会很快地离开该页面，那么 unload 代码就永远不会执行（注 4）。

值得一提的是，在页面上所有关键内容加载完之前，不应该触发 onload 事件，但通常情况下 iframe 包含的内容对于用户所关注的这个页面来说并不是至关重要的。包含广告的 iframe 是一个很好的例子。广告对网站的业务来说可能是至关重要的，但不应该为了等待加载广告而降低用户体验。在典型的应用方式下，iframe 会阻塞 onload 事件，所以，研究是否有不延迟主页面 onload 事件的 iframe 加载方式非常重要。

Iframe Blocking Onload 示例表明了 iframe 阻塞其父窗口的 onload 事件。

Iframe Blocking Onload

http://stevesouders.com/efws/iframe-onload-blocking.php

在本例中，我们以一种典型的方式使用 iframe，通过 HTML 的 SRC 属性设置 iframe 的 URL，如：

```
<iframe src="url"></iframe>
```

本例有 4 个版本，可以通过页面上的链接访问它们：

Empty iframe

iframe4 秒后返回，但它没有包含任何资源。

Iframe with image

iframe 立即返回，但其包含的图片 4 秒后返回。

Iframe with script

iframe 立即返回，但其包含的的外部脚本 4 秒后返回。

Iframe with stylesheet

iframe 立即返回，但其包含的样式表 4 秒后返回。

父窗口 onload 的时间显示在页面的顶部。父窗口仅包含一个资源（iframe），如果 onload 的

注 2：参看 *http://msdn.microsoft.com/en-us/library/bb250448.aspx* 的“理解和解决 Internet Explorer 的内存泄露模式”。

注 3：在 Internet Explorer 6 到 8、Safari3 和 4，以及 Chrome1 和 2 中都是这样。

注 4：有解决这个问题的变通方案——例如：*http://blog.moxiecode.com/2008/04/08/unload-event-never-fires-in-ie/*——但也持续会有使用 unload 事件的开发者没有注意到这个问题。

时间大于 4 秒则说明 iframe 阻塞了 onload 事件。所有主流浏览器的测试结果是一致的：**在典型方式下使用 iframe 时，它会阻塞父窗口的 onload 事件。**

这个阻塞行为有一个很简单的解决方案，但它仅在 Safari 和 Chrome 中有效。使用 JavaScript 动态地设置 iframe 的 URL 而不是使用 HTML 的 SRC 属性：

```
<iframe id=iframe1 src=""></iframe>
<script type="text/javascript">
document.getElementById('iframe1').src = "url";
</script>
```

在 Iframe Not Blocking Onload 示例中使用了这个技术。

Iframe Not Blocking Onload
http://stevesouders.com/efws/iframe-onload-nonblocking.php

在 Safari 和 Chrome 中，onload 的时间是几百毫秒。它远远小于 4 秒，这表明 iframe 和它的组件没有阻塞父窗口的 onload 事件。不幸的是，这项技术在 Internet Explorer、Firefox 和 Opera 中无效，对于绝大部分用户而言，iframe 的阻塞行为延长了页面“完成”的时间。

13.3 使用 iframe 并行下载
Parallel Downloads with Iframes

本节探讨了 iframe 和主页面的下载行为。通常情况下，iframe 和主页面中的资源是并行下载的，但是在某些情况下，主页面会阻塞 iframe 中资源的下载。

13.3.1 脚本位于 iframe 之前
Script Before Iframe

主页面的外部脚本采用典型的方式（<script src="*url*"></script>）加载会阻塞后面的所有资源。因此，如果 iframe 及其资源之前有外部的脚本，它们的下载会被阻塞。Script Before Iframe 示例证明了这点。

Script Before Iframe
http://stevesouders.com/efws/script-before-iframe.php

在这个例子中，脚本 4 秒后返回。iframe 没有延迟，但它包含的图片、样式表和脚本，每个都配置成 4 秒后返回。图 13-2 展示了这个例子在 Internet Explorer、Firefox、Safari、Chrome 和 Opera 中的 HTTP 瀑布图（Safari、Chrome 和 Opera 表现相似，所以把它们组合在一起了）。不出所料，我们看到主页面中的脚本阻塞了 iframe 的请求，它导致了延迟下载 iframe 资源。

脚本阻塞 iframe 下载的方式在所有浏览器中是相似的，但在后面两节中，我们将看到 Internet Explorer 和 Firefox 与这里显示的表现有所不同，而 Safari、Chrome 和 Opera 都表现一致。

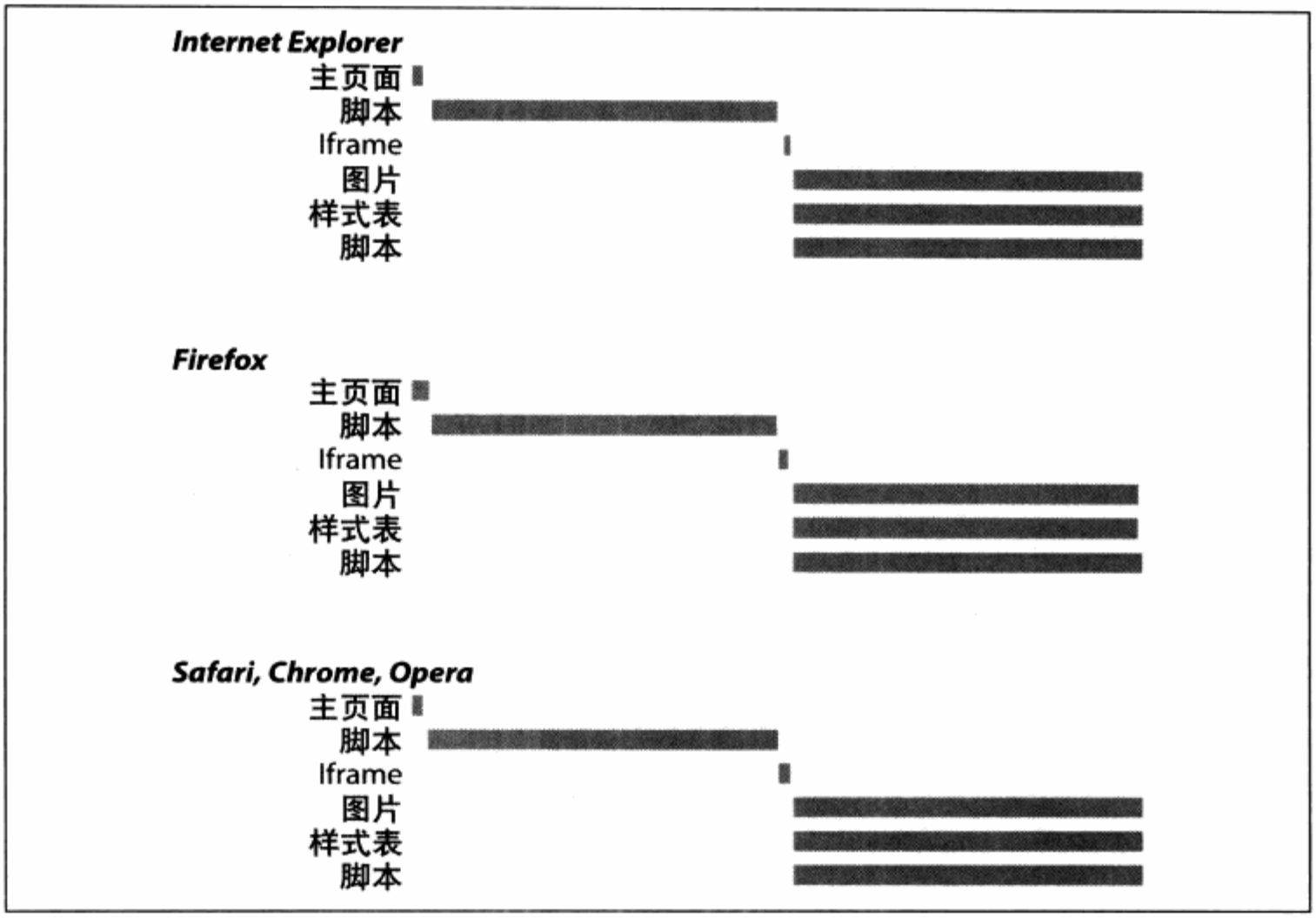


图 13-2：脚本位于 iframe 之前

13.3.2 样式表位于 iframe 之前
Stylesheet Before Iframe

第 7 章讨论了行内脚本位于样式表之后的阻塞行为对页面造成的影响。在 Internet Explorer 和 Firefox 中，样式表也会对 iframe 产生意料之外的阻塞。Stylesheet Before Iframe 示例证明了这点。

Stylesheet Before Iframe
http://stevesouders.com/efws/stylesheet-before-iframe.php

通常，样式表不阻塞其他资源，从图 13-3 中我们看到在 Safari、Chrome 和 Opera 中的确如此。但在 Internet Explorer 和 Firefox 中，样式表阻塞了和 iframe 相关的请求。在 Internet Explorer 中，iframe 请求是被阻塞的。在 Firefox 中，样式表和 iframe 是并发加载的，但样式表阻塞了 iframe 中的资源（注 5）。

注 5：Firefox2 中的性能更糟糕，因为样式表阻塞了所有资源的下载，但在 Firefox3.0 中得到了修正。

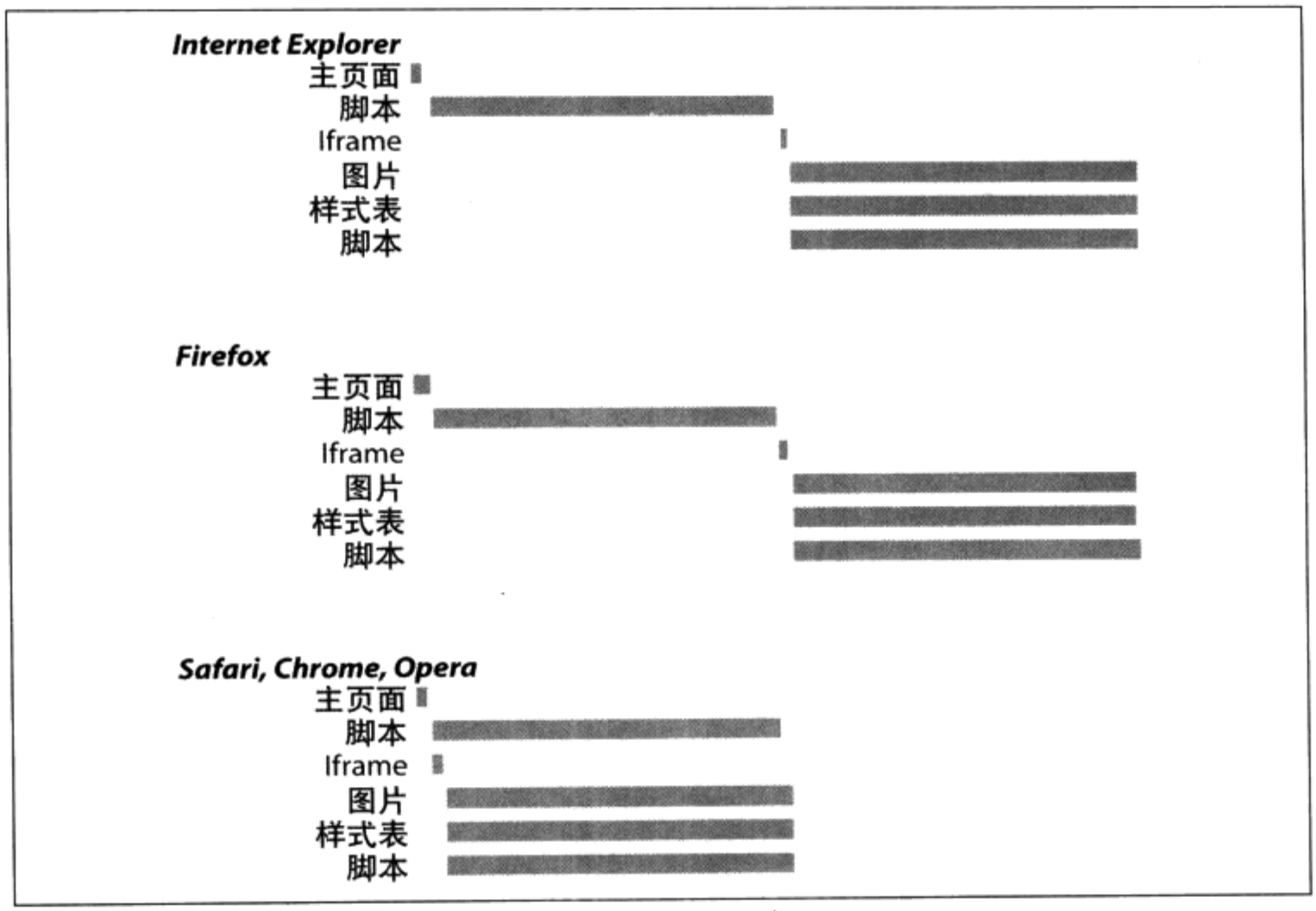


图 13-3：样式表位于 iframe 之前

13.3.3 样式表位于 iframe 之后
Stylesheet After Iframe

把样式表移动到 iframe 之后或许能避免阻塞行为。Internet Explorer 中的确是这样，但在 Firefox 中不会，如例子 Stylesheet After Iframe 所示。

Stylesheet After Iframe

http://stevesouders.com/efws/stylesheet-after-iframe.php

图 13-4 中的 Firefox 瀑布图显示了其下载耗时为 8 秒，而其他主流浏览器在 4 秒内完成了所有下载。虽然不值得把样式表移动到页面较后的位置——它会使得不阻塞 iframe 带来的任何好处都变得毫无意义，因为延迟了渲染——值得注意的是，如果 iframe 的资源在主页面中，那么阻塞不会发生。在判断 iframe 是否为一个合适的解决方案时，阻塞行为是一个重要的考虑因素。

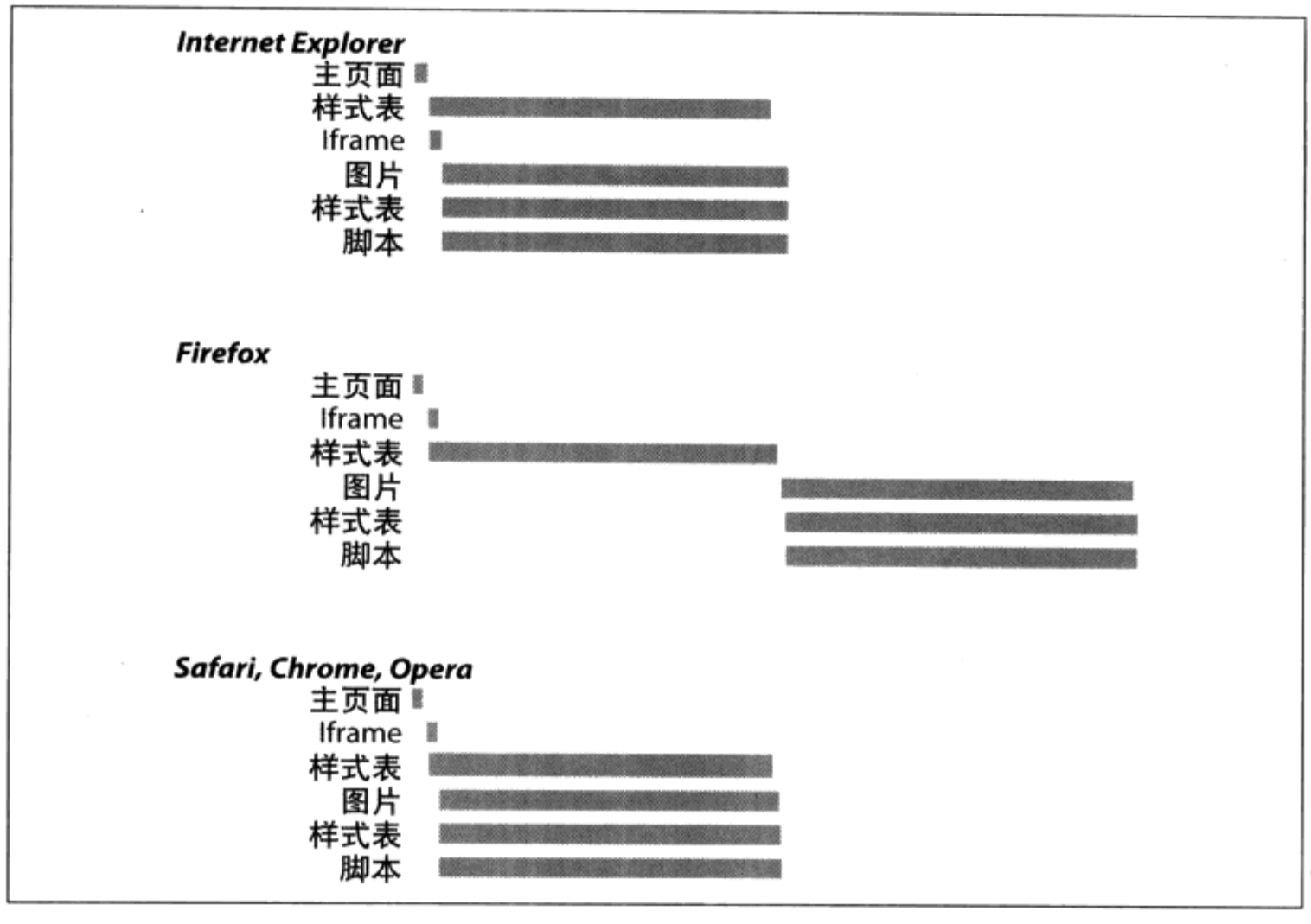


图 13-4：样式表位于 iframe 之后

13.4 每个主机名的连接

Connections per Hostname

浏览器打开单个主机名的连接数是有限制的。连接数决定了能并行下载的资源数量。Internet Explorer 6、Internet Explorer 7 和 Firefox 2 在每个服务器上仅能同时打开两个连接，新型浏览器能同时打开多一些——每个服务器的连接数在 4~8 个之间。（见表 11-2，它完整地展示了浏览器支持情况。）下面的章节探讨了这些浏览器如何跨 iframe、标签页和窗口来执行这些限制。

13.4.1 iframe 中的连接共享

Connection Sharing in Iframes

因为 iframe 是“完全独立的嵌入式文档”（注 6），所以人们希望下载 iframe 中的资源时能使用独立于主页面的连接池。Iframe Connections 示例用于测试这个想法是否可行。

注 6：*http://www.w3.org/TR/html4/struct/objects.html#h-13.5*。

Iframe Connections

http://stevesouders.com/efws/parent-connections.php

Iframe Connections 示例下载了父文档中的 5 张图片和 iframe 中的另外 5 张图片，所有 10 张图片都来自同一个服务器（1.cuzillion.com），并配置其 2 秒后响应。图 13-5 展示了这个例子在 Internet Explorer 7 中的 HTTP 瀑布图。

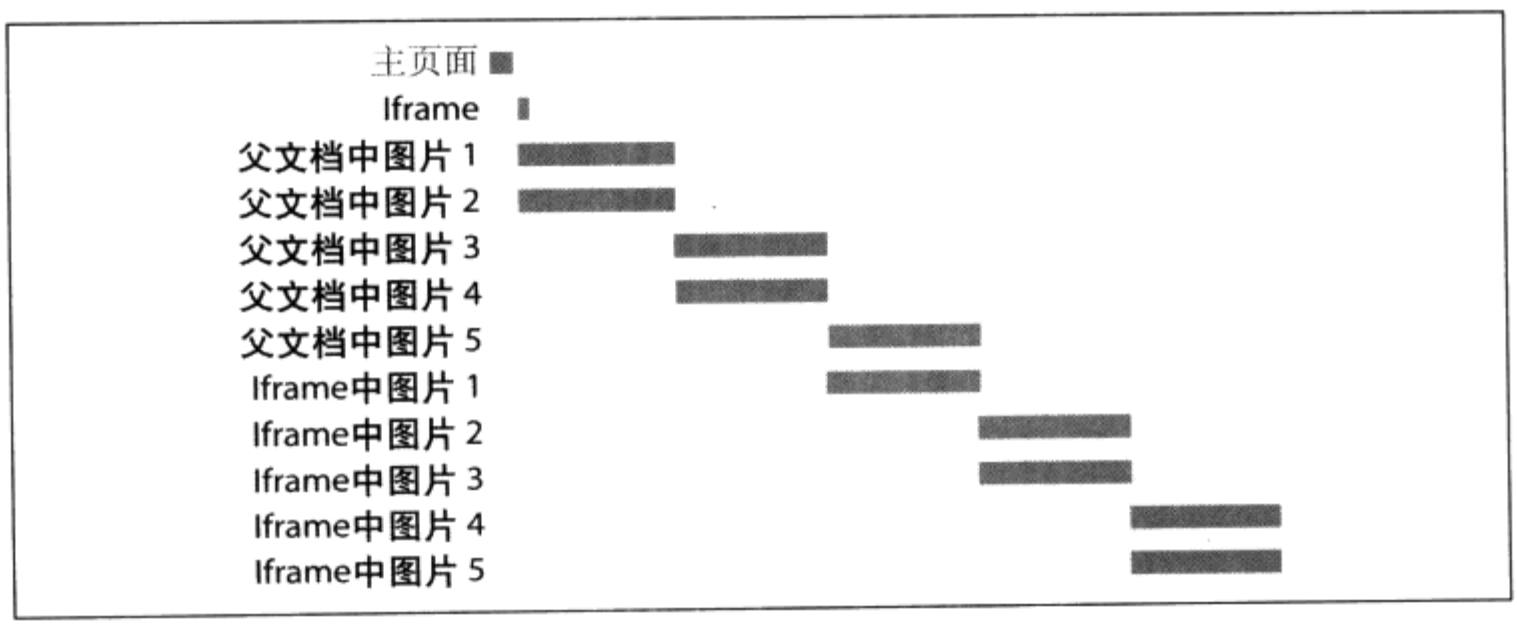


图 13-5：Iframe Connections 示例在 Internet Explorer 7 中的 HTTP 瀑布图

首先出现的 2 个请求分别为 HTML 父文档和 iframe，剩下的请求是来自 1.cuzillion.com 的 10 个图片。Internet Explorer 7 为每个主机名打开两个连接。在图 13-5 中，我们看到父文档和 iframe 中所有请求共享了只有两个连接的受限连接池。

使用 iframe 不会增加指定主机名的并行下载数量，所有主流浏览器都是这样的情况。

13.4.2 跨标签页和窗口的连接共享

Connection Sharing Across Tabs and Windows

跨 iframe 和其父文档的连接限制令人既吃惊又失望。这就引出一个问题：跨多个浏览器标签页和窗口是否也存在类似受限的连接池？

为了回答这个问题，我创建了两个 URL：

http://stevesouders.com/efws/connections1.php；
http://stevesouders.com/efws/connections2.php。

和前面的例子类似，这两个页面每个都包含 5 个来自 *1.cuzillion.com* 的图片，一共 10 个图片。这个测试包括在一个浏览器中打开两个标签页并同时加载这两个 URL。如果共享了连接池，那么下载这 10 个图片将花费更长的时间。如果每个浏览器标签页有其自己的连接池，

那么图片将并行下载且整体加载时间将会变短。然后，我们使用同一浏览器的两个实例在独立窗口中加载这些 URL 来完成同样的测试。

我在 Internet Explorer 8.0 beta 2、Firefox 3.1b2、Safari 4 开发预览版、Chrome 2.0 和 Opera 10.0 alpha 中运行了这些测试，结果是在所有浏览器中跨标签页和窗口都是共享连接池的。图 13-6 展示了 Internet Explorer 8.0 beta 2 中的 HTTP 瀑布图。

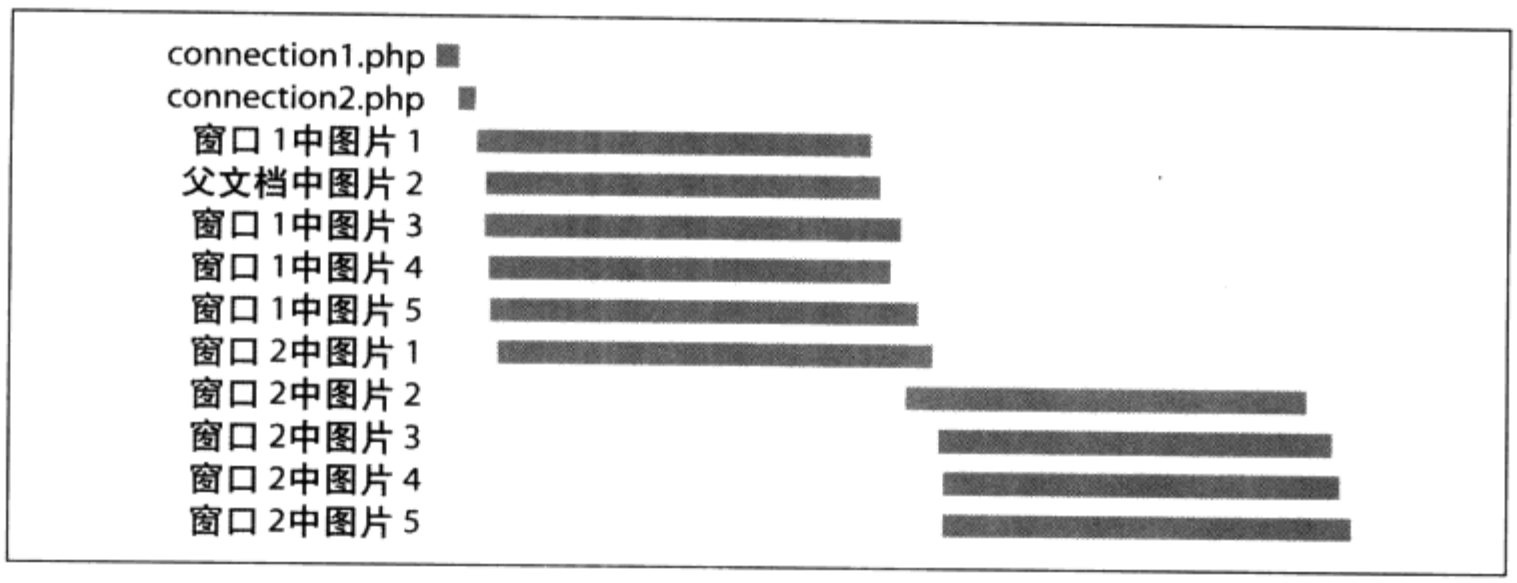


图 13-6：在 Internet Explorer 8 中跨窗口共享连接

最先出现的两个短请求是两个测试 URL，而长请求是图片。Internet Explorer 8 在每个服务器上最多打开 6 个连接，这有 6 个连接的连接池用于下载 connections1.php 的 5 个图片和 connections2.php 的第 1 个图片。同时，虽然 connections2.php 中的剩余图片是在独立的窗口中获取的，但它们也依旧被阻塞。

虽然本小节偏离了 iframe 的主题，但研究跨标签页和窗口的连接限制行为是值得的。对于公司而言，把多个属性部署在单个域名的主机上时，如果用户同时打开多个 Web 应用，将在性能上带来负面影响。例如，下面几个 Google 应用都托管在 *http://www.google.com* 上：

- Google Calendar（*http://www.google.com/calendar/*）。
- Google Finance（*http://www.google.com/finance*）。
- Google Reader（*http://www.google.com/reader/*）。
- Google Search（*http://www.google.com/*）。
- iGoogle（*http://www.google.com/ig*）。

这些网站使用的大多数资源也都来自 *http://www.google.com*。如果用户同时打开两个或多个应用，将造成连接请求的争夺，从而导致加载速度变得更慢。虽然这并不会频繁发生，但它确实存在。比如，当每天早上打开浏览器时，我会使用脚本同时打开 Google Calendar、

Google Reader 和 iGoogle。这些站点是相互影响的，因为即使加载在独立的标签页中，它们也必须共享连接。

13.5 总结使用 iframe 的开销
Summarizing the Cost of Iframes

即使是空 iframe，其开销也是很高的，它们的开销比其他的 DOM 元素高出 1~2 个数量级。

当我们在典型方式下使用 iframe 时(`<iframe src="url"></iframe>`)，它会阻塞 `onload` 事件。这会延长浏览器的忙指示，导致用户感觉页面加载很慢。动态地设置 iframe 的 `SRC` 属性能在 Safari 和 Chrome 中避免这个问题。对于其他浏览器而言，可以在 `onload` 事件后设置 `SRC` 属性来避免。

虽然 iframe 不会直接阻塞主页面中的资源下载，但是有几种方式会导致主页面阻塞 iframe 的下载。除了预期的脚本行为之外，在 Internet Explorer 和 Firefox 中，主页面的样式表会阻塞 iframe 的下载。

在主页面和 iframe 之间，即使 iframe 是一个完全独立的文档，浏览器也会共享对每个服务器的受限连接数。那些把大多数资源都托管在一个单独域名下的网站应该牢记这一点。

正因为有那么多的开销，所以最好是避免使用 iframe，但一项快速调查表明它们依旧被频繁使用。美国排名前 10 的网站有 5 个使用 iframe：AOL、Facebook、MSN.com、MySpace 和 YouTube。这些站点主要将 iframe 用于广告投放，这是预料之中的，因为 iframe 可以简易地嵌入来自第三方站点的内容，特别像轮播广告这样的动态内容。

对于主页面来说，创建一个 `div` 来容纳广告内容是使性能更佳的广告嵌入方式之一。当主页面请求广告的外部脚本时（使用第 4 章所介绍的异步技术），`div` 的 id 可以包含在脚本的 URL 中。广告的 JavaScript 将通过设置 `div` 的 `innerHTML` 属性把广告插入页面中。这种方法也更符合“可扩展”的广告需求——这些广告占据了当前窗口的大部分，因此不应该勉强地使用 iframe。iframe 的使用呈下降趋势，同时那些用于插入广告的其他技术变得越来越普遍，这对提升网页性能非常有益。

第 14 章

简化 CSS 选择符

Simplifying CSS Selectors

本书的大部分内容都在关注 JavaScript 的性能。那么，CSS 的性能又如何呢？在已发布的 CSS 资料中，大部分都关注布局、设计，以及内容、标签和代码之间的关系（注 1）。针对 CSS 的性能，有一些最佳实践：

- 把样式表放在文档 HEAD 标签中以提升页面的逐步渲染速度。（见《高性能网站建设指南》第 5 章。）
- 不要在 IE 中使用 CSS 表达式，因为它们可能会被执行成千上万次，从而导致打开页面的速度变慢。（见《高性能网站建设指南》第 7 章。）
- 避免使用过多的行内样式，因为这会增加下载页面内容的大小。（见本书第 9 章。）

另外一个已经引起大家关注的话题是使用低效 CSS 选择符的开销。**选择符**由一系列初始化的参数组成，这些参数指明了要应用这个 CSS 规则的页面元素。本章解释了有关 CSS 选择符的问题，这有点出人意料。虽然遵循准则对 CSS 选择符进行优化可以提升网站速度，但更重要的是，Web 开发者应该避免一些常见且开销很大的 CSS 选择符模式。在接下来的几节将会介绍这些模式。

14.1 选择符的类型

Types of Selectors

本节解释了有关 CSS 选择符的术语。看看下面这个例子：

```
mg#toc > LI { font-weight: bold; }
```

这是一个简单的样式规则，使用的 CSS 选择符是`#toc > LI`。该选择符包含了两个**简单选择符**（`#toc` 和 `LI`），并用连结符>把它们**组合**起来。CSS 选择符决定了页面中的哪些元素（也叫对象）接受指定的样式。

注 1：见 Nicole Sullivan 的《面向对象的 CSS》和 Nate Koechley 的《语义化标签——创建、支持和提取》（译注 1）。

译注 1：这两篇文章的地址分别是 *http://www.stubbornella.org/content/2009/02/28/object-oriented-css-grids-on-github/*和 *http://nate.koechley.com/blog/2005/02/09/semantic-markup-create-support-and-extract/*。

浏览器尝试把 CSS 选择符和文档中的元素匹配起来，而让人担忧的正是这种匹配方式。因为 CSS 选择符的编写方式决定了浏览器必须执行的匹配次数，而某些类型的 CSS 选择符将会导致浏览器尝试更多匹配，因此开销比简单选择符更高。

下面几个小节介绍了 CSS 选择符的各种类型，大致按照从最简单（最小开销）到最复杂（最大开销）的顺序列出。更多有关信息请参考 CSS2 规范的选择符章节（译注 2）。

本节中的示例规则都在 CSS Selectors 示例中应用的，该例的使用场景是目录页样式。图 14-1 展示应用了适当样式规则的页面，并指出它们在哪儿产生影响。

CSS Selectors

http://stevesouders.com/efws/selectors.php

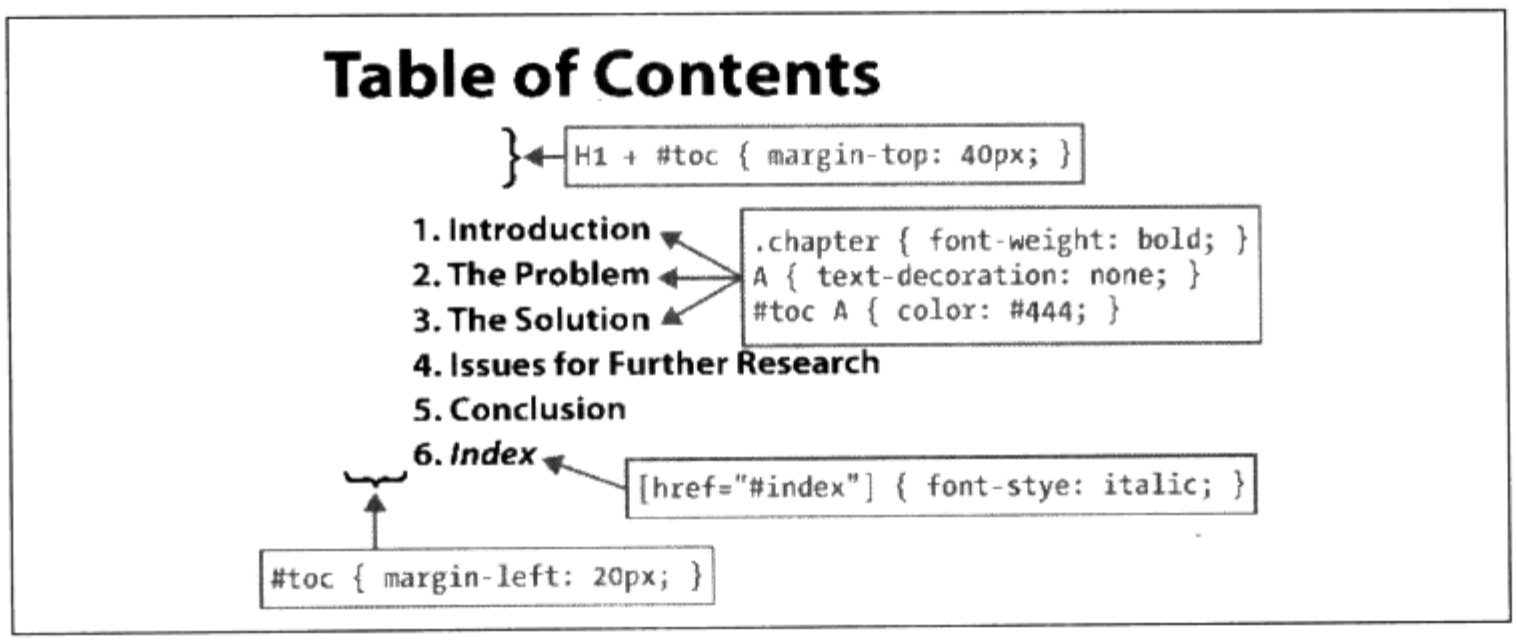


图 14-1：带规则的 CSS Selectors 示例

14.1.1 ID 选择符

ID Selectors

示例：`#toc { margin-left: 20px; }`

这种类型的选择符简单且高效，它用指定的 id 匹配页面中唯一的元素。上面这个例子匹配了 id 属性为 `toc` 的元素。在 Table of Contents 示例中，这条规则匹配了一个有序列表元素：`<ol id=toc>`，并使列表左边缩进 20px。

译注 2：见 *http://www.w3.org/TR/CSS21/selector.html*。

14.1.2 类选择符
Class Selectors

示例：`.chapter { font-weight: bold; }`

类的规则是通过点(.)和其后紧跟的类名来指定的，类选择符匹配所有类属性包含该名字的元素。在 Table of Contents 示例中这条规则匹配了列表项元素，使其字体加粗：`<li class=chapter>`。

14.1.3 类型选择符
Type Selectors

示例：`A { text-decoration: none; }`

类型选择符应用于指定元素类型的所有元素。这条规则移除了页面中所有链接的下划线，例如：`<a href="#introduction">Introduction</a>`。这是一个给指定类型的所有元素添加样式的轻量级方法，使用它即可无需再给每个元素增加额外的字符（如 id、class 或 style）。

14.1.4 相邻兄弟选择符
Adjacent Sibling Selectors

示例：`H1 + #toc { margin-top: 40px; }`

相邻兄弟选择符用连结符 + 把两个简单选择符（这里是 `H1` 和`#toc`）组合起来。在 CSS Selectors 示例中，这条规则匹配 `toc` 元素，由于它的前一个兄弟元素是 `H1`，因此 Table of Contents 示例的顶部有额外 40px 的 margin。

14.1.5 子选择符
Child Selectors

示例：`#toc > LI { font-weight: bold; }`

子选择符是用连结符>把两个或多个简单选择符组合起来。这条规则匹配父元素是 `toc` 元素的所有列表项，效果和类选择符示例中一样，但在这里 `LI` 元素不需要指定一个类名，因此它减少了结果页的代码量。

14.1.6 后代选择符
Descendant Selectors

示例：`#toc A { color: #444; }`

前面两个选择符类型使用了 + 和 > 连结符，后代选择符简单地使用空格（“ ”）作为连结符。当第 2 个选择符的对象是第 1 个选择符对象的后代（子、孙等）时，后代选择符规则会进行匹配。在示例页面中，`toc` 元素中的所有链接（`A`）元素都指定了字体颜色“#444”。

注意，我在这里使用简写来定义颜色。通常规定颜色的格式是“#444444”，但“#444”（译注 3）是同等的且节约了 3 个字节。

14.1.7 通配选择符

Universal Selectors

示例：`* { font-family: Arial; }`

通配选择符使用 * 来表示，匹配文档中的每一个元素。这条规则将 CSS Selectors 示例中所有元素的字体都设置为 Arial。

14.1.8 属性选择符

Attribute Selectors

示例：`[href="#index"] { font-style: italic; }`

属性选择符根据元素的属性是否存在或其属性值进行匹配。这条规则将 `href` 属性值等于`"#index"`的链接元素设置为斜体。我们可以通过 4 种方式匹配属性：

- 全等，使用 = 进行匹配。
- 判断属性是否存在，不考虑属性值是什么：`[href]`
- 等于用空格分隔的属性值列表中的任意一个：`[title~="Index"]` 匹配 `<a title="the Index">`。
- 等于属性值用连字符-分隔开的属性值列表中的第一个：`[LANG|=en]` 匹配 `<p lang="en">`和`<p lang="en-US">`。

类选择符是属性选择符中的特例，其对应的属性就是 `class`。用于类选择符的点符号（例如：`chapter`）是一种简写，这样就避免了编写较长的属性选择符语法（`[class="chapter"]`）。

14.1.9 伪类和伪元素

Pseudo-Classes and Pseudo-Elements

示例：`A:hover { text-decoration: underline; }`

目前介绍的选择符类型都是基于 DOM 的，但某些预期的样式不能在 DOM 中表现，伪类和伪元素就是用来解决这个问题的。当用户鼠标悬停在链接上时，这条规则会为链接添加下划线。`:hover` 是一个伪类，其他的伪类还包括：`:first-child`、`:link`、`:visited`、`:active`、`:focus` 和`:lang`。伪元素包括：`:first-line`、`:first-letter`、`:before` 和`:after`。

14.2 高效 CSS 选择符的关键

The Key to Efficient CSS Selectors

CSS 选择符对性能的影响源于浏览器匹配选择符和文档元素时所消耗的时间，开发者可以

译注 3：“#444444”是十六进制 RGB 颜色记法，如果其 3 组数各自成对两两相等时，CSS 允许采用简写记法，即“#6FA”与“#66FFAA”相同，这种记法编写简单且节约字符数。

通过编写更高效的选择符来控制匹配耗时。实现高效选择符之路是从理解选择符如何匹配开始的。

14.2.1 最右边优先
Rightmost First

看看如下规则：

```
#toc > LI { font-weight: bold; }
```

我们中的大多数人，尤其是那些从左到右阅读的人，可能猜想浏览器也是执行从左到右匹配规则的，因此会推测这条规则的开销不高。在脑海中，我们想象浏览器会像这样工作：找到唯一的 id 为 `toc` 的元素，然后把这个样式应用到直系子元素中的 `LI` 元素上。我们知道只有一个 id 为 `toc` 的元素，并且它只有几个 `LI` 子元素，所以这个 CSS 选择符应该相当高效。

事实上，CSS 选择符是从右到左进行匹配的。了解这方面的知识后，我们知道这个之前看似高效的规则实际开销相当高，浏览器必须遍历页面上每个 `li` 元素并确定其父元素的 id 是否为 `toc`。

前面提到的后代选择符示例甚至更糟：

```
#toc A { color: #444; }
```

如果浏览器是从左到右读取的，那么它仅仅需要检查 `toc` 里的链接元素，实际上却检查了整个文档中的每个链接。然而，浏览器并不仅仅检查每个链接的父元素，还要遍历文档树去查找 id 为 `toc` 的祖先元素。如果被评估的链接不是 `toc` 的后代，那么浏览器就要向上一级遍历直到文档的根节点。

Safari 和 WebKit 的架构师 David Hyatt（译注 4）在一篇名叫《在 Mozilla UI 中编写高效的 CSS》（译注 5）的文章中揭示了这些内容。在有关 CSS 选择符性能的文章中，这篇是被引用次数最多的文章之一：

> 样式系统从最右边的选择符开始向左匹配规则。只要当前选择符的左边还有其他选择符，样式系统就会继续向左移动，直到找到和规则匹配的元素，或者因为不匹配而退出。

14.2.2 编写高效的 CSS 选择符
Writing Efficient CSS Selectors

理解了选择符是从右到左匹配之后，我们就可以从另一个角度看 CSS 选择符，并将其调整得更高效。在我们开始这项工作之前，先了解一下这方面的一些其他知识也十分必要，诸如哪些 CSS 选择符的开销是最高的，以及更容易修复这些问题的一些模式。幸运的是，David Hyatt 的文章提供了编写高效选择符的指南：

译注 4：*http://en.wikipedia.org/wiki/Dave_Hyatt*。

译注 5：英文名是 Writing Efficient CSS for use in the Mozilla UI，网址是 *https://developer.mozilla.org/en/Writing_Efficient_CSS*。

避免使用通配规则

除了传统意义上的通配选择符之外，Hyatt 也把相邻兄弟选择符、子选择符、后代选择符和属性选择符都归纳到“通配规则”分类下。他推荐仅使用 ID、类和标签选择符。

不要限定 ID 选择符

在页面中一个指定的 ID 只能对应一个元素，所以没有必要添加额外的限定符。例如，`DIV #toc` 是没必要的，应该简化为`#toc`。

不要限定类选择符

不要用具体的标签限定类选择符，而是根据实际情况对类名进行扩展。例如，把 `LI.chapter` 改成`.li-chapter`，或是`.list-chapter` 更好。

让规则越具体越好

不要试图编写像 `OL LI A` 这样的长选择符，最好是创建一个像`.list-anchor` 一样的类，并把它添加到适当的元素上。

避免使用后代选择符

通常处理后代选择符的开销是最高的，而使用子选择符也可以得到想要的结果，并且更高效。遵循下一条指南就更好了，这样就连子选择符也避免使用了。

避免使用标签—子选择符

如果有像`#toc > LI > A` 这样的基于标签的子选择符，那么应该使用一个类来关联每个标签元素，如`.toc-anchor`。

质疑子选择符的所有用途

再次提醒大家检查所有使用子选择符的地方，然后尽可能用具体的类取代它们。

依靠继承

了解哪些属性可以通过继承而来，然后避免对这些属性重复指定规则。例如，对列表元素而不是每个列表项元素指定 `list-style-image`。请参考继承属性的列表（*http://www.w3.org/TR/CSS21/propidx.html*）来了解每个元素可继承的属性。

很有意思的是，David Hyatt 的文章首次发表在 2000 年 4 月。我很想知道 9 年后这个话题又被人重新提起的原因。David 的文章正如标题所描述的那样，是写给在 Mozilla UI 中工作的开发者的。或许经过这么久的发展，在网页上使用 CSS 选择符的性能损失才达到了类似在 Mozilla UI 中的影响程度。

另一个因素是现在的 Web 2.0 应用都有较长的会话时间，这已经不再是加载—清除—加载的 Web 1.0 方案。从这个意义上讲，Web 2.0 应用和 Mozilla UI 更相似，并且低效 CSS 选择符的影响可能会更加显著，就像对 DOM 树进行大量创建或删除操作和用 DHTML 代码改变类

名或样式属性一样。由于网页的复杂性和动态性，我们应该更加关注性能分析领域，下一节的研究结论将会支持这个观点。

14.3 CSS 选择符性能
CSS Selector Performance

揭开了选择符性能的面纱，我们知道了哪些 CSS 选择符效率低下。选择符从右到左读取的发现激发很多人重写规则。根据 Doug Crockford 的指导（见第 1 章），开始解决可能存在的性能问题之前，最重要的是首先评估该问题的影响力，这可以确保我们把注意力集中在正确的问题上。

14.3.1 复杂的选择符影响性能（有时）
Complex Selectors Impact Performance (Sometimes)

Jon Sykes 曾发表 3 篇博文介绍测量 CSS 选择符性能的实验结果。每篇博文都是对上一篇的改进，所以第 3 篇是最有用的（注 2）。他的测试由 5 个页面组成，所有页面都包含了 20 000 个链接元素，每个链接都有一个 `p`、`div`、`div`、`div` 和 `body` 的祖先树。每个页面有不同类型的 CSS：

- “No Style”表示没有 CSS。
- “Tag”表示有一条规则：

```
a { background-color: red; }
```

- “Class”表示有 20 000 个类选择符，每个选择符都对应一个链接元素，例如：

```
class11 { background-color: red; }
```

- “Descender”表示有 20 000 个后代选择符，每个链接 1 个，例如：

```
div div div p a.class11 { background-color: red; }
```

- “Child”表示有 20 000 个子选择符，每个链接 1 个，例如：

```
div > div > div > p > a.class11 { background-color: red; }
```

结果的确表明“No Style”比“Descender”和“Child”都快。在 IE 和 Safari 中，较慢页面的加载时间是简单页面的好几倍。很明显，大量低效的 CSS 选择符会对性能产生负面影响。

如果把选择符的数量减少到类似现在网站的水平，它们依然会有影响吗？表 14-1 显示在美国排名前 10 的网站中，CSS 规则和 DOM 元素的数量，以及 DOM 深度的平均值。规则的总数量范围从 92~2882，平均值是 1033。

注 2：博文原址 *http://blog.archive.jpsykes.com/153/more-css-performance-testing-pt-3/*已经无法访问了（译注 6）。

译注 6：译者发现原文依旧可以访问，但里面的测试页面无法访问了。三篇博文分别是：*http://blog.archive.jpsykes.com/151/testing-css-performance/index.html*，*http://blog.archive.jpsykes.com/152/testing-css-performance-pt-2/index.html*，*http://blog.archive.jpsykes.com/153/more-css-performance-testing-pt-3 /index.html*。

表 14-1：美国排名前十的网站的 CSS 规则和 DOM 元素

网站	规则数量	DOM 元素数量	平均深度
AOL	2289	1628	13
eBay	305	588	14
Facebook	2882	1966	17
Google	92	552	8
Live Search	376	449	12
MSN.com	1038	886	11
MySpace	932	444	9
Wikipedia	795	1333	10
Yahoo!	800	564	13
YouTube	821	817	9
平均	1033	923	12

基于这些信息，我创建了一套类似 Sykes 实验的测试，但使用规则的数量不是 20 000 条而只有 1000 条。为了使页面的大小和实际情况更一致，像其他所有页面一样，我在基准页和标签选择符页中也设置了 1000 条规则，没有任何元素使用这些简单的类规则。所有这些页面本身就是 CSS Selectors Test 示例的组成部分。

CSS Selectors Test

http://stevesouders.com/efws/css-selectors/tests.php

这个实验的重点是评估复杂选择符和简单选择符的开销。图 14-2 显示了最慢测试页（子选择符和后代选择符）和简单的基准页加载时间的不同，平均降幅仅有 30 毫秒（注 3）。

这些测试显示了优化 CSS 选择符减少的开销比在 Sykes 测试所发现的结果小很多。这主要是规则数量减少的缘故，加之实际上 CSS 选择符的影响对规则和 DOM 元素的数量增长是非线性的。图 14-3 显示了在 Internet Explorer 7 下，那些高开销的子选择符和后代选择符数量从 1000~20000 逐渐增加时页面加载时间的变化曲线。这些数据表明在 Internet Explorer 7 中，大约达到 18 000 条规则时开销陡然升高，使用 20 000 条规则测试的结果在这条类似曲棍球棒的曲线最末端。

注 3：Opera9.63 的结果不一致，因此省略。

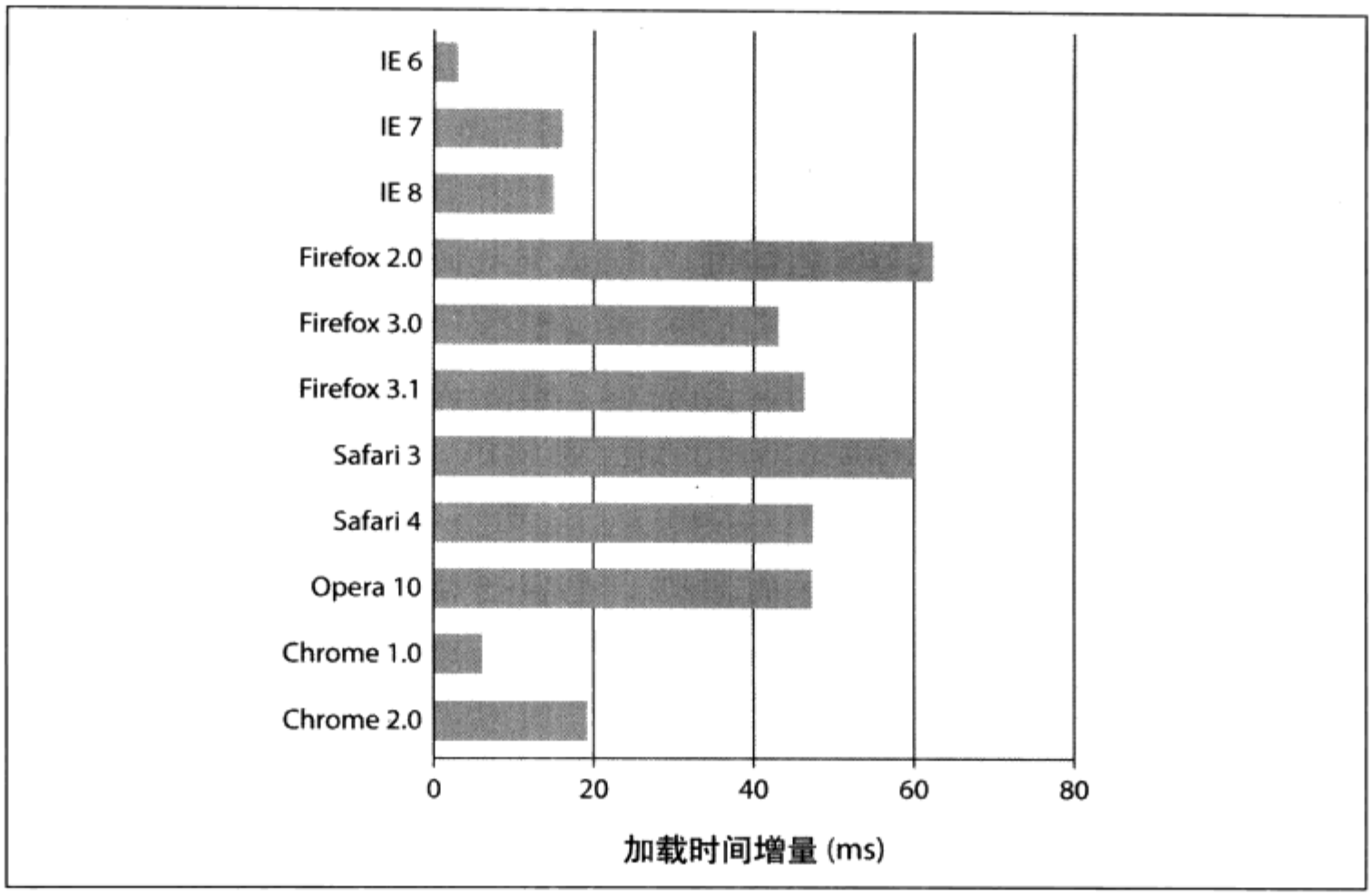


图 14-2：简单和复杂选择符测试的加载时间差异

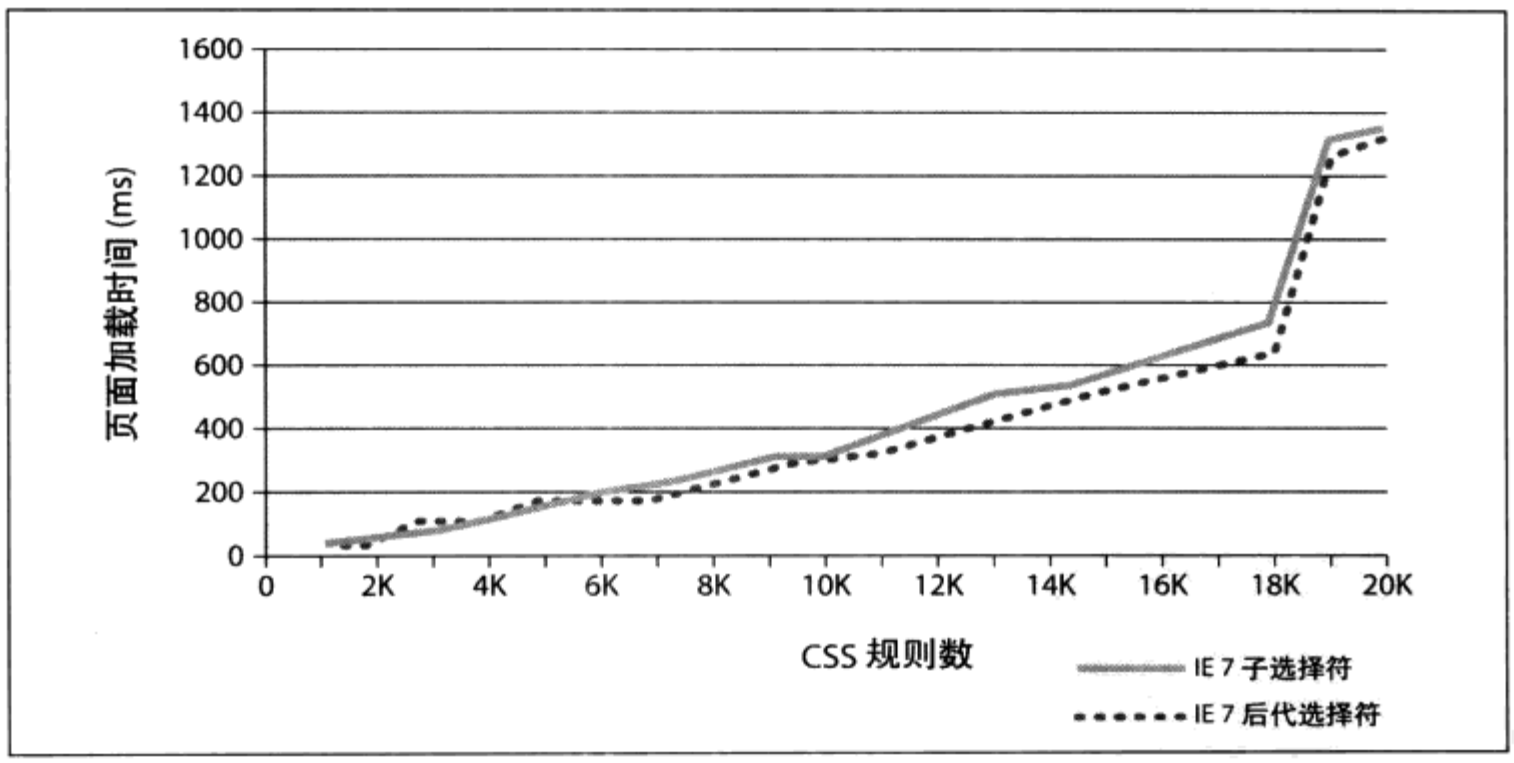


图 14-3：在 Internet Explorer 7 下 CSS 选择符的曲线

这些结果表明，像子选择符和后代选择符这样比较复杂的 CSS 选择符并不总是影响页面性能，但这并不意味着我们不需要优化 CSS 选择符。在平时编写 CSS 时，某些类型的选择符对性能还是会有显著的影响。

14.3.2 应避免使用的 CSS 选择符
CSS Selectors to Avoid

上一节的测试说明，在某些情况下，即使复杂的 CSS 选择符在性能上的影响也很小，但并非总是如此。让我们看一个之前测试过的后代选择符的例子：

```
DIV DIV DIV P A.class0007 { ... }
```

初看之下，这似乎是一个匹配开销很高的选择符，它是一个要匹配五级祖先元素的后代选择符。然而，回想一下选择符是从右到左匹配的，我们就会认识到这个后代选择符执行的速度和一个更简单的类选择符差不多。最右边的参数也叫**关键选择符**，它对浏览器执行的工作量起主要影响。在本例中，关键选择符是 `A.class0007`，页面中只有一个元素匹配这个关键选择符，所以匹配这个选择符所需要的时间是极少的。

反之，看看这条规则：

```
A.class0007 * { ... }
```

在这条规则中，关键选择符是 *。它匹配所有的元素，所以浏览器必须检查每个元素是否为类名为 `class0007` 的链接元素的后代。Universal Selector 示例中有 1 000 条这个类型的规则。

Universal Selector

http://stevesouders.com/efws/css-selectors/universal.php

图 14-4 显示了 Universal Selector 页面和 Descendant Selector 页面（*http://stevesouders.com/efws/css-selectors/descendant.php*）在加载时间上的不同。在图 14-2 中平均增量仅为 30 毫秒，与之相比，这有显著的改变，平均降幅超过 2 秒！

当决定对哪儿进行优化时，记住将重心放在那些可能匹配大量元素的**关键选择符**上。不仅通配选择符有这个问题，下面几个例子同样会消耗大量的时间加载页面：

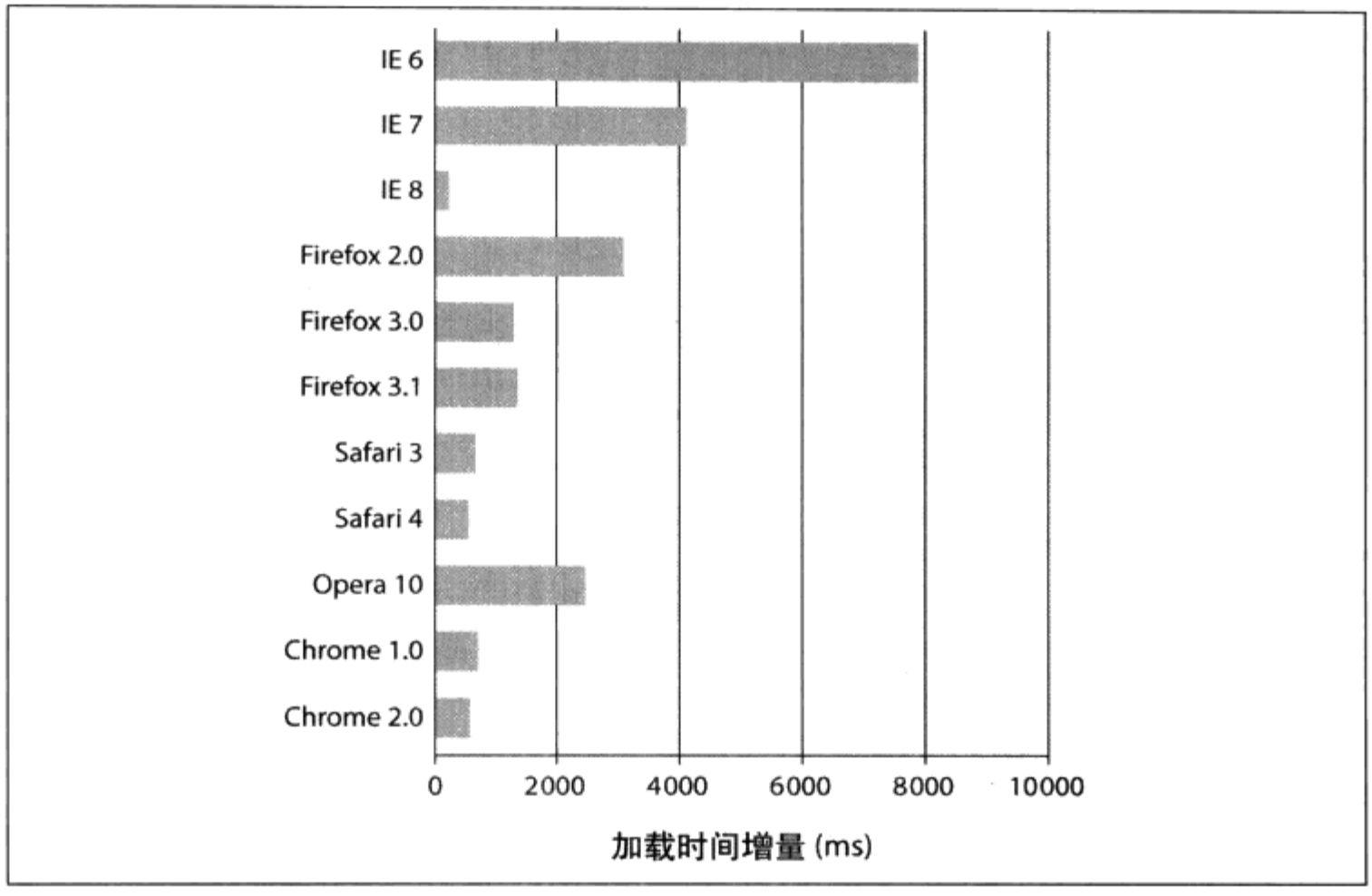


图 14-4：通配选择符的加载时间差异

```
A.class0007 DIV { ... }
```
http://stevesouders.com/efws/css-selectors/csscreate.php?sel=A.class+DIV

```
#id0007 > A { ... }
```
http://stevesouders.com/efws/css-selectors/csscreate.php?sel=%23id+>+A

```
.class0007 [href] { ... }
```
http://stevesouders.com/efws/css-selectors/csscreate.php?sel=.class+[href]

```
DIV:first-child { ... }
```
http://stevesouders.com/efws/css-selectors/csscreate.php?sel=DIV%3Afirst-child

这些例子都是由 CSS 测试生成器（*http://stevesouders.com/efws/css-selectors/csscreate.php*）生成的。“0007”表示计数器从 1 增长到规则的最大值（这里是指 1000）的过程中的一个数字。CSS 测试生成器使很多工作变得简单起来，包括尝试不同类型的选择符、测量加载时间的影响，以及对在下节讨论的回流时间（reflow time）的测试。

14.3.3 回流时间
Reflow Time

之前所有的例子都在评估 CSS 选择符在加载时间上的影响。对于 Web2.0 应用来说，更应考虑当用户和网页交互时，浏览器应用样式和布局元素所花费的时间，这又叫做**回流时间**。

当使用 JavaScript 修改 DOM 元素样式的某些属性时会触发回流。对于这个叫 elem 的 DOM 元素，下面的每行代码在大多数浏览器中都会触发回流：

```
elem.className = "newclass";
elem.style.cssText = "color: red";
elem.style.padding = "8px";
elem.style.display = "";
```

这仅仅是一个子集，能触发回流的列表太长了。Web 2.0 应用的动态性令其很容易就触发回流。回流并不需要涉及页面上的所有元素，浏览器已为此进行了优化，仅仅只对那些受回流影响的元素重新布局。在前面的例子中，如果 elem 是文档的 body 或其他一些有很多后代的元素，那么回流的开销一定会相当高。

回流需要重新应用 CSS 规则，这意味着浏览器必须再次匹配所有的 CSS 选择符。如果 CSS 选择符是低效的，那么回流可能消耗的时间就会多到引起用户注意。所有的 CSS 选择符测试示例（*http://stevesouders.com/efws/css-selectors/tests.php*）都有一个测试回流（Measure Reflow）的按钮。点击后，正如前面代码最后一行所示，body 的 display 属性会被切换，执行回流所消耗的时间会显示在按钮旁边。上一节中高开销的 CSS 选择符示例中回流时间范围从数百毫秒到一秒。

因此，对于杜绝低效 CSS 选择符的影响，不仅要考虑页面加载时间，也要考虑用户和 Web 2.0 应用交互时如何使用样式进行表现，这一点非常重要。如果 JavaScript 对样式属性有操作，且页面开始变慢，那么低效的 CSS 选择符就很可能是罪魁祸首。

14.4 在现实中测量 CSS 选择符

Measuring CSS Selectors in the Real World

本章介绍了多个实验及其结果，但所有这些测试页面都是精心设计的例子，把这些测试所节约的时间转化成现实世界网站可能会节约的时间非常困难。理想的实验可能是优化排名前十的网站的 CSS 选择符，并测量在加载时间上的效果，但这是不可行的。

为了评估通过优化 CSS 选择符可能得到的性能提升，我们可以测量回流时间。使用 Lindsey Simon 的回流计时器（*http://code.google.com/p/reflow-timer/*）可以很容易地对现有网页进行测试。它是一个能在所有主流浏览器中运行的书签工具（bookmarklet）。运行时，它切换了 body 的 display 属性并显示平均回流时间。（我就是从这个工具中得到灵感，并基于这个工具在测试页面加入回流时间测试器。）图 14-5 显示了测量回流时间的结果，其范围从 16 毫秒（Google 搜索）~391 毫秒（Facebook）（注 4）。

注 4：使用 Internet Explorer 7 进行测量的。

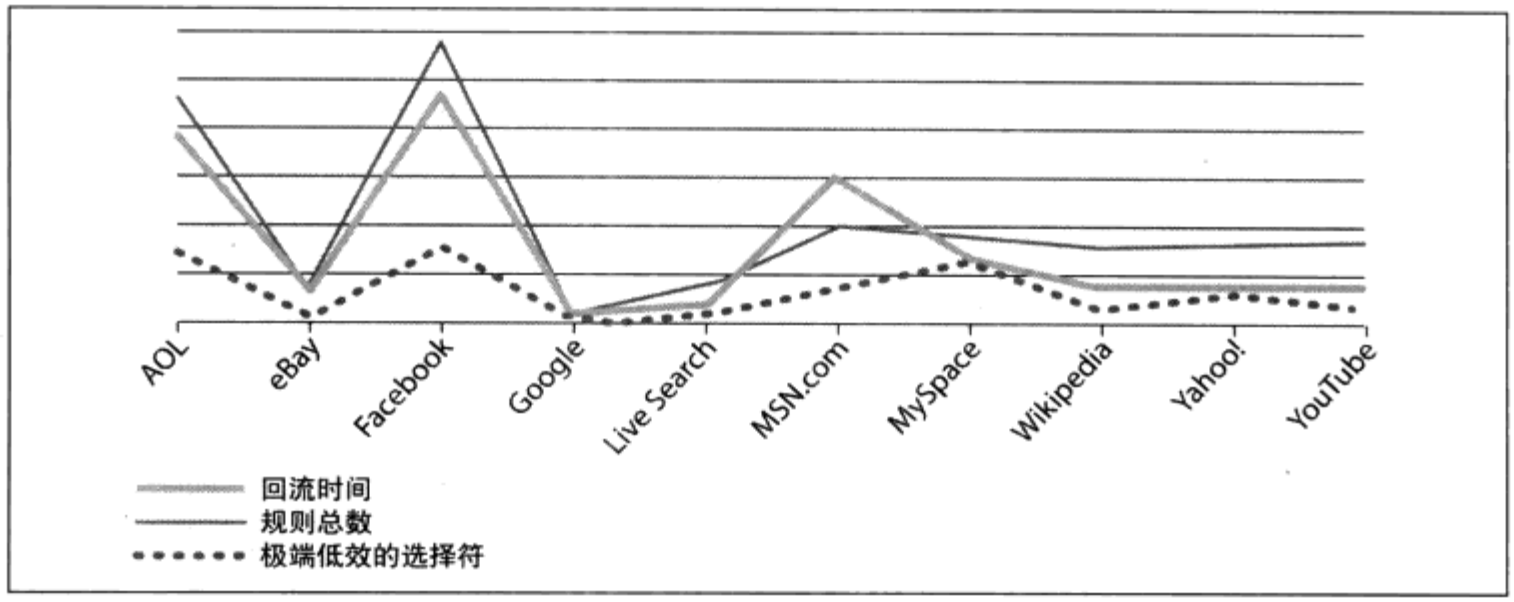


图 14-5：排名前十的网站的回流时间、规则总数和极端低效的选择符数量

除了回流时间，图 14-5 也显示了规则的数量和极端低效规则的数量（关键选择符会匹配大量元素的规则）。回流时间和规则数量的相关系数是 0.86，而回流时间和极端低效规则数量的相关系数是 0.9。它们都具有高相关性，这表明浏览器应用样式消耗的时间受到 CSS 选择符数量和低效 CSS 选择符的影响。

如果你的站点像 AOL 和 Facebook 一样使用大量的规则，而且其中有许多是极端低效的，那么优化 CSS 选择符可以让页面更快。减少规则的数量很可能使其受益无穷。话虽如此，你要牢记，按照 David Hyatt 的指南来编写高效的 CSS 选择符是要付出成本的：将每个受影响元素的后代选择符替换成类选择符，不仅会增加页面大小，还会降低样式的灵活性。最需要修正的选择符是那些可以匹配大量元素的关键选择符（最右边的选择符）。虽然性能提升的益处多种多样，但 Web 开发者应该避免使用会影响页面性能的 CSS 选择符。

性能工具
Performance Tools

像所有优秀工程师一样，Web 开发者需要建立一套工具去完成高水准的任务。我在附录中推荐了一些分析和改善网站性能的工具，它们分为以下几节进行介绍：

“数据包嗅探器”，第 205 页
当坐下来开始分析一个网站的时候，我首先关注的问题是网页的 HTTP 请求。这样就可以确定页面哪些地方比较缓慢，所以第一个要添加到工具包中的是方便且易用的数据包嗅探器。这类工具包括：HttpWatch、Firebug Net Panel、AOL Pagetest、VRTA、IBM Page Detailer、Web Inspector Resources Panel、Fiddler、Charles 和 Wireshark。

“Web 开发工具”，第 209 页
页面性能指的并不仅仅是加载时间——尤其对 Web 2.0 的应用程序来说，JavaScript、CSS 和 DOM 结构都在其中扮演着重要的角色。Web 开发工具提供探测器（inspectors）、性能分析器（profilers）和调试工具（debuggers）来分析网页的行为。本节包括 Firebug、Web Inspector 和 IE Developer Toolbar。

“性能分析器”，第 211 页
性能分析器用于评估指定网页是否与性能最佳实践相违背。本节介绍的工具有 YSlow、AOL Pagetest、VRTA 和 neXpert。它们每个测量的内容也都是各不相同的。

“其他”，第 216 页
本节包括了我经常使用的各种工具集合：Hammerhead、Smush.it、Cuzillion 和 UA Profiler。

数据包嗅探器
Packet Sniffers

每个 Web 开发者在优化性能时都需要查看页面（包括页面中全部资源）的加载过程，这就要靠**数据包嗅探器**来完成。本章节列出了一系列数据包嗅探器，包括从像 HttpWatch 这样能提供网络流量的高级视图工具到像 Wireshark 这样能抓取网络上每个数据包的低级视图工

具。在多数的 Web 性能分析中，我用的是高级视图的网络监视器；它们通常更容易配置且有更直观的用户界面。在某些情况下，如调试块编码（chunked encoding）时，我需要使用低级视图的数据包嗅探器，这样才能看到内容和线上发送每个数据包的时序。

HttpWatch

HttpWatch（*http://www.httpwatch.com/*）是我的首选数据包嗅探器。HttpWatch 把网络流量用图形的方式绘制出来，如图 A-1 所示。本书的大多数 HTTP 瀑布图都是使用 HttpWatch 抓取的。图形化的展示更容易发现性能的延迟问题。

HttpWatch 是由 Simtec 公司开发的，你可以免费下载试用版，但它只能在少数几个大网站上工作，如谷歌和雅虎。全功能版虽然需要你支付一定的费用去购买，但是绝对物有所值。HttpWatch 可以运行在 Microsoft Windows 的 Internet Explorer 和 Firefox 上。

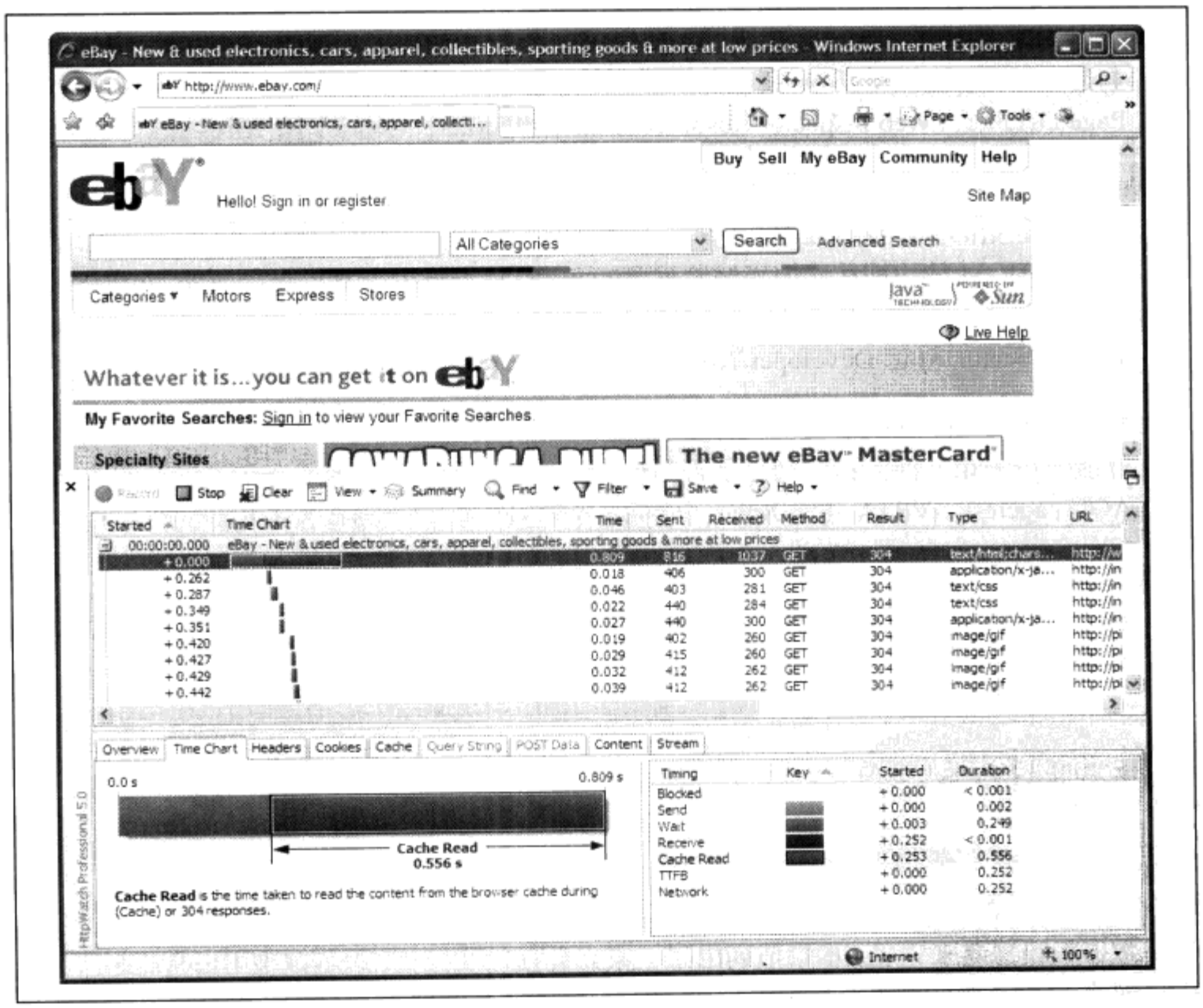


图 A-1：HttpWatch

Firebug 网络控制面板

Firebug Net Panel

对任何一个 Web 开发者来说，Firebug 拥有许多重要的功能。我将在第 209 页的“Web 开发工具”中更加深入地讲解它。这里值得一提的是 Firebug 的网络控制面板，已经安装了 Firebug 的开发者可以很容易地选择使用网络控制面板来显示 HTTP 瀑布图。我特别喜欢使用网络控制面板的垂直线，它用来标示网页加载时间线上的 `DOMContentLoaded` 和 `onload` 事件的发生，如图 A-2 所示。其他数据包嗅探器也应当增加这个好功能。

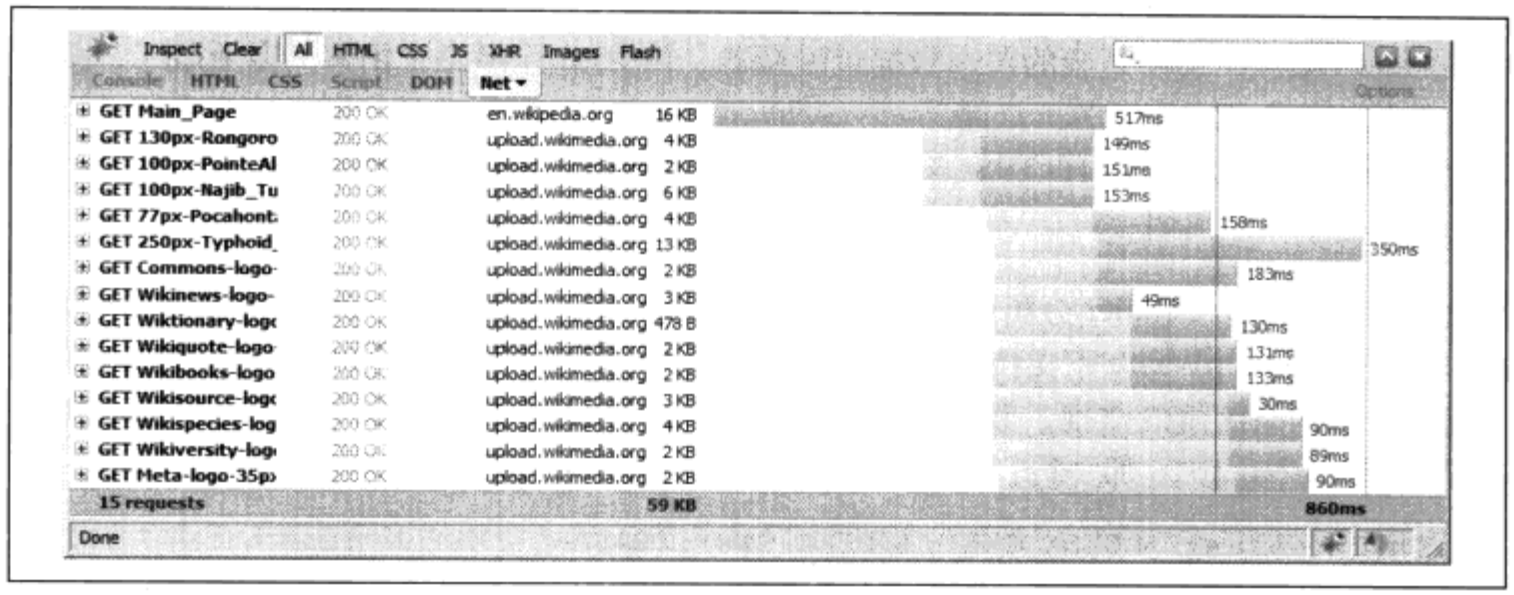


图 A-2：Firebug 网络控制面板

网络控制面板的缺点是它的时间信息会受到网页本身的影响。这是由于 Firebug 实际上是用 JavaScript 实现的，因此它和当前网页在相同的 Firefox 进程上执行。正因为如此，如果当 JavaScript 正在主页面上执行时有网络事件发生，那么网络控制面板会因阻塞而无法记录那些请求的真实时间信息。不过网络控制面板在大多数情况下是足够精准的，并且它的易用性使其成为一个不错的选择。如果需要更精确的时间测量或页面上有一段要长时间运行而导致阻塞的 JavaScript，你应该考虑使用本章提及的其他数据包嗅探器。

另一个极大的制约因素是，Firebug 插件仅仅适用于 Firefox，所以它并不能在其他浏览器上运行。

AOL Pagetest

AOL Pagetest（*http://pagetest.wiki.sourceforge.net/*）是 Internet Explorer 的插件，用于生成 HTTP 瀑布图。它也能识别出性能有待改进的一些地方，这将在第 211 页的“性能分析器”中讨论。

VRTA

微软开发的 VRTA（*http://www.microsoft.com/downloads/details.aspx?FamilyID=119F3477-DCED-41E3-A0E7-D8B5CAE893A3&displaylang=en*）着眼于提高网络性能。它的 HTTP 瀑布图比其他网络监视器更详细，其重点是能显示对已有 TCP 连接的重用。关于 VRTA 的更多信息请看第 211 页的“性能分析器”。

IBM Page Detailer

IBM Page Detailer 曾经是我的首选数据包嗅探器，但 IBM 已经停止销售其专业版了。虽然其基本版依然可用（*http://www.alphaworks.ibm.com/tech/pagedetailer*），但我认为它缺乏许多必要的功能，如支持分析 HTTPS 请求和数据导出功能。IBM Page Detailer 运行在 Microsoft Windows 上。

我使用 IBM Page Detailer 分析除 Internet Explorer 和 Firefox 以外的其他浏览器，如 Opera 和 Safari(因为 HttpWatch 不支持这些浏览器)。IBM Page Detailer 可以对任意一个使用 HTTP 的进程进行网络流量的监视。我们可以像如下所示那样编辑 wd_WS2s.ini 文件，然后添加进程名到 `Executable` 那一行：

```
Executable=(FIREFOX.EXE),(OPERA.EXE),(SAFARI.EXE)
```

Chrome 为每个标签页都添加一个独立的浏览器 UI 进程，这个有趣的特性阻止了 IBM Page Detailer 分析 Chrome。IBM Page Detailer 依附在浏览器 UI 进程上，所以它无法检测到实际加载网页的任何 HTTP 流量。不过，如果不需要支持 HTTPS 和数据导出，IBM Page Detailer 还是一个很不错的选择。

Web Inspector 资源控制面板
Web Inspector Resources Panel

Safari 的 Web Inspector 类似 Firebug，也是一种具有网络监视器功能的网页开发工具。更多信息请参见第 209 页的“Web 开发工具”。

Fiddler

Fiddler(*http://www.fiddler2.com/fiddler2/*)是由微软 Internet Explorer 团队的 Eric Lawrence 开发的，它的主要特点是具有支持脚本的功能，可以设置断点并且操纵 HTTP 通信。其缺点是，实际上它是一个代理，所以可能会改变浏览器的一些行为（例如，每个服务器的打开连接数）。如果你很需要脚本功能，并充分考虑过使用代理所带来的副作用，我强烈推荐 Fiddler。它运行在 Microsoft Windows 上。

Charles

Charles(*http://www.charlesproxy.com/*)是一个类似 Fiddler 的 HTTP 代理。它有很多跟 Fiddler 类似的特性，包括分析 HTTP、HTTPS 流量，以及带宽限制。Charles 支持 Microsoft Windows、Mac OS X 和 Linux。

Wireshark

Wireshark（*http://www.wireshark.org/*）从 Etheral 发展而来。它在数据包级别上分析 HTTP 请求，它的 UI 并不像其他网络监视器一样用图形化来显示，它也没有“网页”的概念，所以要靠你自己去分辨网页数据包的开始和结束。如果你需要查看数据包级别的流量，比如分析块编码，Wireshark 是最佳选择。它可在多种平台上使用，包括 Microsoft Windows、Mac OS X 和 Linux。

Web 开发工具
Web Development Tools

数据包嗅探器可以将页面正在载入时的网络活动情况展示出来，但网页性能不只是 HTTP 请求。第 1 章和第 2 章讲述了为什么 JavaScript 和对 DOM 的修改会让网页的速度变慢。在本章节介绍的 Web 开发工具有 Firebug、Web Inspector 和 IE Developer Toolbar——其中的功能包括诸如 DOM 检查器、JavaScript 调试器与性能分析器、CSS 编辑器和网络监视器。

不过这些工具还只是冰山一角。开发者需要更为深入的工具，来查看在完整页面加载时间线上的内存消耗、CPU 负载、JavaScript 执行、CSS 应用和 HTML 解析与渲染，并且要求这些分析不影响浏览器的正常行为。

Firebug

Firebug（*http://getfirebug.com/*）是最流行的 Web 开发工具，被下载超过 1400 万次（没错，是千万级的!）。它是由 Joe Hewitt 在 2006 年 1 月开发的，包括 HTML、CSS、DOM 和布局检查器。在第 205 页的“数据包嗅探器”中已经讲述过了 Firebug 的网络控制面板，它提供了一个网络活动的 HTTP 瀑布图。Firebug 还拥有一个 JavaScript 输入命令行和输出控制台，以及 JavaScript 调试器和性能分析器。调试器和性能分析器是 Firebug 功能最强大的部分。

Firebug 是 Firefox 的一个扩展应用。虽然把 JavaScript 调试和性能分析功能移植到其他浏览器将是一个艰巨的任务，但借助于 Firebug Lite（*http://getfirebug.com/lite.html*）可以跨浏览器使用 Firebug 的一部分功能。Firebug Lite 是个书签按钮，因此它能工作在所有主流浏览器上。它有一个重大的升级是由 Azer Koculu 完成的，目前包括了 HTML、DOM 和 CSS 的检

查器，以及 JavaScript 输入命令行和输出控制台。Firebug Lite 提供了一个在所有浏览器通用的用户界面和比较完整的功能集，是解决令人讨厌的浏览器不兼容 Bug 的最佳工具。

开发者喜爱 Firebug 是因为他们可以对它进行扩展。这种开放扩展模型使得在某种程度上添加Firebug功能成为可能，它也允许和其他开发者共享这些新功能。你可以在 *http://getfirebug.com/extensions/index.html* 找到有用的 Firebug 扩展。

Web Inspector

Safari 的 Web Inspector 在 2008 年底做过一次重要的升级。图 A-3 显示的是前面提到的资源控制面板。Web Inspector 的功能类似于 Firebug，它有可自动补全的控制台、CSS 与 DOM 的检查器，以及 JavaScript 的调试器与性能分析器。

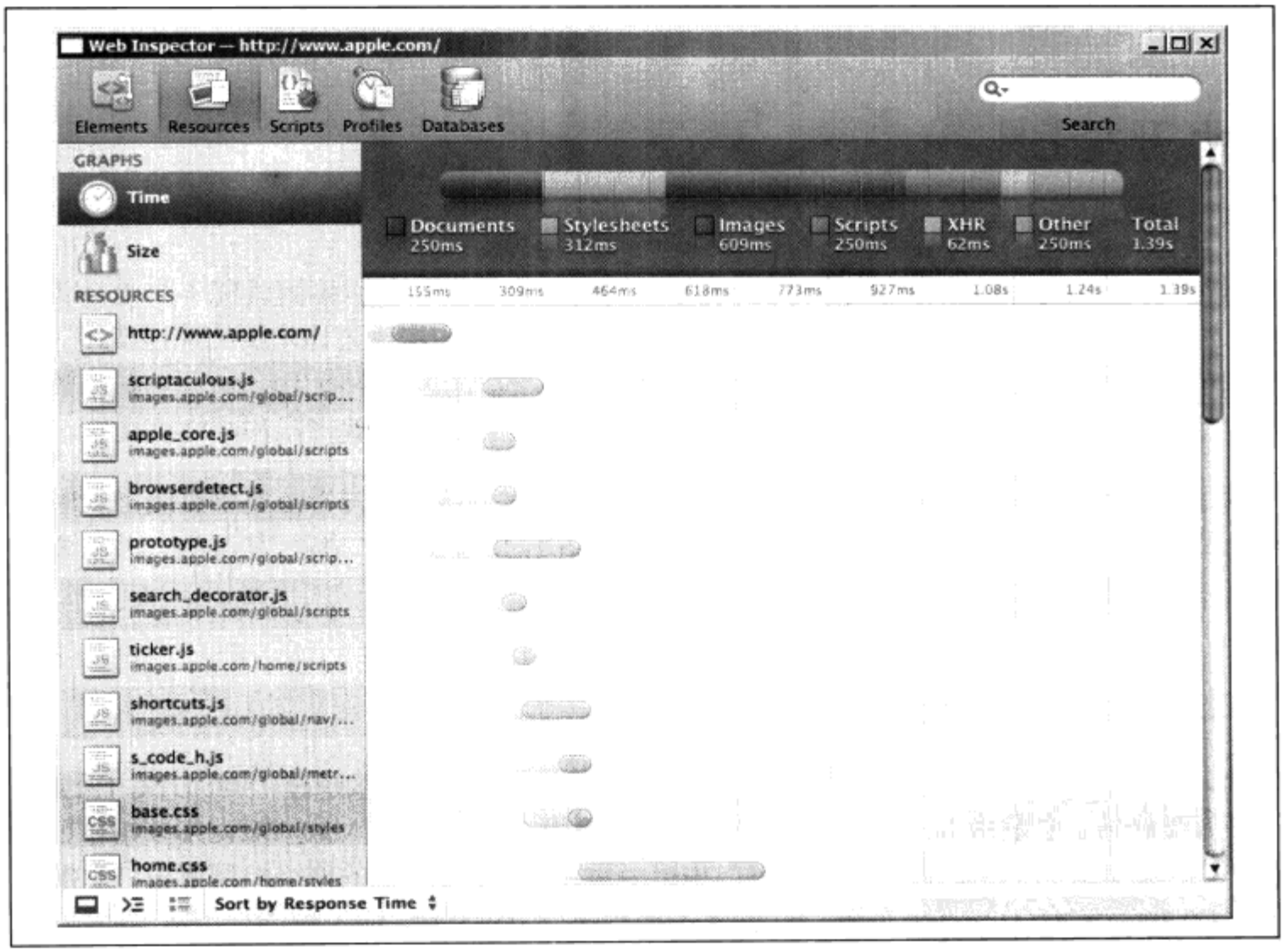


图 A-3：Safari 的 Web Inspector

IE Developer Toolbar

IE Developer Toolbar（*http://www.microsoft.com/downloads/details.aspx?familyid=e59c3964-672d-4511-bb3e-2d5e1db91038&displaylang=en*）有一个类似 Firebug Lite 的功能集。它没有 JavaScript 调试器和性能分析器，但是能支持 HTML 与 CSS 的验证、DOM 检查器和像素布局工具。IE Developer Toolbar 只针对 Internet Explorer 6 和 7。该功能已内置在 Internet Explorer 8 中，你可以点击“开发人员工具”菜单项打开它（译注 1）。

性能分析器
Performance Analyzers

YSlow 是第一个被广泛使用的性能检测工具（译注 2）。AOL Pagetest、VRTA 和 neXpert 都是在它之后才发布的，这些工具都有其自己的一套性能最佳实践。表 A-1 中列出了所有这些最佳实践，并指出了每个特定工具是使用哪些规则进行评估的。我把它们分为 3 类：

- 《高性能网站建设指南》中所包含的规则。
- 本书中所提到的最佳实践。
- 其他那些我没提到但已纳入这些工具的最新版本中的规则。

看表 A-1，很明显每个工具所支持的最佳实践很少有重叠。从某种意义上说这是件好事，从不同的角度研究性能问题会让我们发现新的最佳实践。但是这种多样性也有一些更加重要且不利的影响：Web 开发社区的混乱和分化。目前还不清楚哪一套最佳实践是最好的。如何选择工具应当由开发环境所决定，而不是性能分析的内容。

纵观这些工具的开发者，他们在性能最佳实践上的一致性要比表 A-1 所反映出来的高。出现不一致有几个原因：他们都期望引入新的最佳实践，而很少集中去涵盖在那些已被其他工具包含的最佳实践。开发时间始终是一个问题，开发者可能决定跳过一些众所周知的最佳实践的实现。另外，不要低估了个人兴趣的影响，例如，VRTA 的开发者在网络问题上比我更有兴趣且精通。

译注 1：它在“工具”菜单项下面，更方便的方式是使用快捷键 F12。

译注 2：原文是 performance “lint” tool。lint 本意是(保护伤口的)软麻布。在计算机编程中，这个术语通常应用于那些标识用各种计算机语言编写的软件中可疑用法的工具。详情见 *http://en.wikipedia.org/wiki/Lint_%28software%29*。比如 Douglas Crockford 写的大名鼎鼎的检测 JavaScript 的代码质量工具就叫 JSLint，详情见 *http://www.jslint.com/*。

表 A-1：性能最佳实践

Best practice	YSlow	Pagetest	VRTA	neXpert
High Performance Web Sites				
Combine JavaScript and CSS	×	×		
Use CSS sprites	×		×	
Use a CDN	×	×		
Set Expires in the future	×	×	×	×
Gzip text responses	×	×	×	×
Put CSS at the top	×			
Put JavaScript at the bottom	×			
Avoid CSS expressions	×			
Make JavaScript and CSS external	×			
Reduce DNS lookups	×			
Minify JavaScript	×	×		
Avoid redirects	×		×	×
Remove dupe scripts	×			
Remove ETags	×	×		×
Even Faster Web Sites				
Don't block the UI thread				
Split JavaScript payload				
Load scripts asynchronously			×	
Inline scripts before stylesheet				
Write efficient JavaScript				
Minimize uncompressed size				
Optimize images		×		
Shard domains			×	
Flush the document early				
Avoid iframes				
Simplify CSS selectors			×	
Other				
Use persistent connections		×	×	×
Reduce cookies		×		×
Avoid network congestion			×	
Increase MTU, TCP window			×	
Avoid server congestion			×	

展望未来，如果这些及其他的工具能够共享一套性能最佳实践，那么 Web 开发者能将工作做得更好。我完全相信会出现这种情况。驱动创建这些工具的精神是向所有用户宣传更快的 Web 体验和帮助开发者轻松识别出那些能最大提升其站点速度的地方。本着这种精神，无论开发者选择什么样的平台或者工具，给他们更加一致的工具是有意义的。

这就是未来。就目前而言，以下部分提供了一些已有的性能分析工具的说明，包括 Yslow、AOL Pagetest、VRTA 和 neXpert。

YSlow

我在 Yahoo!工作时创建了 YSlow（*http://developer.yahoo.com/yslow/*）。它最早是以书签形式存在的，后来变成了一个 Greasemonkey 脚本。Joe Hewitt 十分好心地讲解了如何将 YSlow 变成 Firebug 的扩展，Swapnil Shinde 做了大量的编码才使得它能在 Firebug 上工作。我把 YSlow 交给 Swapnil 的动机是我确信 YSlow 的使用者将多达 10 000 人。YSlow 发布于 2007 年 7 月，在一年半之后下载量突破 100 万大关，它的名字正好是“whY is this page Slow”的简写。

YSlow 包含的以下规则都在《高性能网站建设指南（High Performance Web Sites）》中作为独立章节进行了讲解。当 YSlow 发布时，我也在 *http://developer.yahoo.com/performance/rules.html* 发表了每条规则的摘要。该页面随后被 Yahoo!的同事更新到了 34 条规则。原有的 13 条规则仍然是 YSlow 性能分析的基础：

- 第 1 条：尽量减少 HTTP 请求。
- 第 2 条：使用 CDN。
- 第 3 条：添加 `Expires` 头。
- 第 4 条：采用 Gzip 压缩组件。
- 第 5 条：将样式表放在顶部。
- 第 6 条：将脚本放在底部。
- 第 7 条：避免 CSS 表达式。
- 第 8 条：使用外部的 JavaScript 和 CSS。
- 第 9 条：减少 DNS 查询。
- 第 10 条：精简 JavaScript。
- 第 11 条：避免重定向。
- 第 12 条：删除重复的脚本。
- 第 13 条：配置 ETag。

YSlow 作为 Firebug 的扩展仅能使用在 Firefox 上。它根据每一条规则对网页进行评分，然后对每个独立规则的分数进行加权平均值得出总分数。它也能显示页面上使用的所有资源

列表和整体统计（请求数、总的页面大小等）。它还有其他一些很有用的工具，包括同 JSLint（*http://jslint.com*）的整合，以及将所有 CSS 或 JavaScript 输出到一个独立的浏览器窗口中以便搜索。

AOL Pagetest

AOL Pagetest 和其 Web 版本 WebPagetest（*http://www.webpagetest.org/*）都使用如下的最佳实践来分析网页：

- 启用浏览器缓存静态资源。
- 对所有的静态资源使用 1 个 CDN。
- 合并静态 CSS 和 JavaScript 文件。
- 对所有合适的文本资源进行 Gzip 编码。
- 压缩图片。
- 使用持久连接。
- 正确使用 Cookie。
- 精简 JavaScript。
- 不采用 ETag 头。

AOL Pagetest 是 Internet Explorer 的插件。WebPagetest 可以通过任何浏览器访问，它实际运行在后端服务器的 Internet Explorer 上。除了性能分析之外，它们还提供了 HTTP 瀑布图、屏幕截图、网页加载时间和统计摘要。

通过 WebPageTest 网站来部署这些功能十分有趣。WebPagetest 相当流行，但一直没有得到它应有的广泛采用。它可以通过任何浏览器来对任何网站进行分析，没有下载、安装和配置应用程序或插件的麻烦。这一切通过在 WebPagetest 后端服务器的 Internet Explorer 上运行 AOL Pagetest 来实现。来自任何浏览器的 WebPagetest 用户只要简单地在表单中输入想要分析的网站 URL，1 分钟之后就会生成分析报告。图 A-4 显示了 *http://www.aol.com/*的报告。

WebPagetest 这种网页形式令每个人用起来都很方便，包括非开发人员，但它确实也有一定的局限性。统计报告永远都是使用 WebPagetest 的远程 Internet Explorer 产生的，这有点让人莫名其妙。注意在图 A-4 中我使用的是 Firefox，而使用 IE 来生成这类报表却是一个挑战。同样，结果并不一定反映你的本地环境。如果尝试在当前的网络连接下调试程序，或者正加载的网页依赖于你当前的 Cookie，那么 WebPagetest 将无法正确捕获它们。这时可以

选择 AOL Pagetest（它是一个需要下载到本地安装的 Internet Explorer 插件）或在上一节提到的其他数据包嗅探器来分析当前的浏览体验。

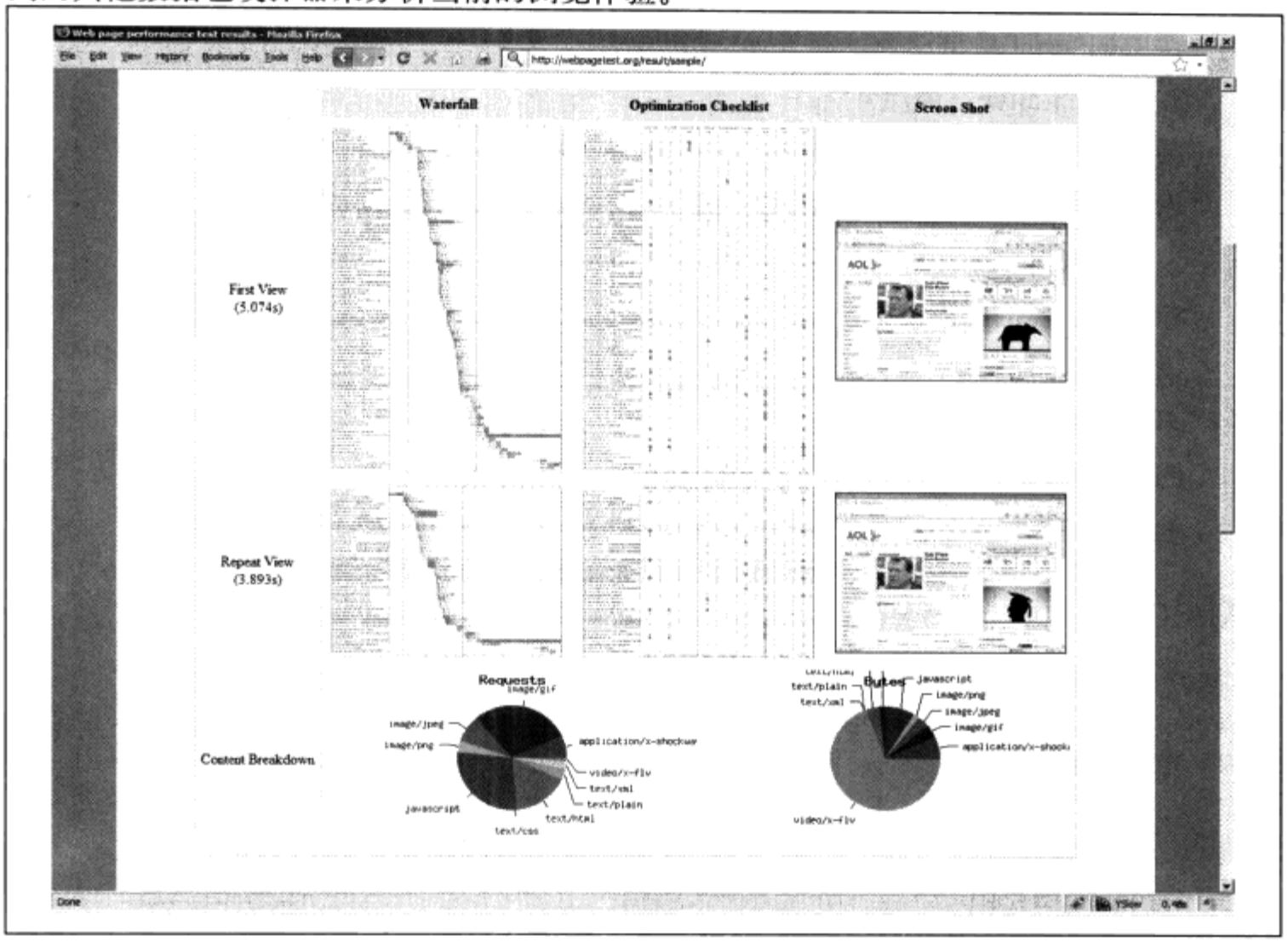


图 A-4：WebPagetest

VRTA

来自微软的 VRTA 是 Visual Round Trip Analyzer 的简称。它可以显示 HTTP 瀑布图，但相比其他的工具，它能提供更多、更详细的内容。VRTA 侧重于网络优化，它一个重要功能就是显示了已有 TCP 连接的重用。在大多数的 HTTP 瀑布图中，每个 HTTP 请求是一个单独的横杠。相反，VRTA 把每个 TCP 连接作为一个横杠。这可以很容易地看到 TCP 连接是如何使用的。VRTA 还显示比特率直方图，以表示可用带宽的利用程度。

除了其复杂的网络图表之外，VRTA 还通过下面一组性能最佳实践来评估页面加载信息（译注 3）：

译注 3：*http://msdn.microsoft.com/zh-cn/magazine/dd188562.aspx*。

- 打开足够的端口。
- 限制需要加载的小文件数量。
- 在 JavaScript 引擎之外加载 JavaScript 文件。
- 开启 keep-alive（译注 4）。
- 识别网络拥塞。
- 增加网络最大传输单元（MTU）或 TCP 窗口大小（TCP window size）。
- 识别服务器拥塞。
- 检查不必要的往返。
- 设置过期时间。
- 不轻易使用重定向。
- 使用压缩。
- 编辑 CSS。

neXpert

（*http://www.microsoft.com/downloads/details.aspx?familyid=5975da52-8ce6-48bd-9b3c-756a625024bb&displaylang=en*）同样也来自微软，它是 Fiddler 的插件（关于 Fiddler 的更多信息见第 205 页的“数据包嗅探器”），使用 Fiddler 收集有关网页下载的资源信息。neXpert 用一组最佳实践来分析这些信息，并生成改进建议报告。相比其他性能分析器，neXpert 能进一步预测改良后的网页加载时间。neXpert 分析性能的最佳实践列表包括以下内容：

- HTTP 响应代码。
- 压缩。
- ETag。
- 缓存头。
- 连接头。
- Cookie。

杂项

Miscellaneous

本节将讨论在前面几节没有涉及的特定用于 Web 性能领域的工具。虽然不是每天都使用所有的这些工具，但我也会定期地去使用它们。

Hammerhead

提高 Web 性能就需要测量页面加载时间。虽然这听起来很简单，但在现实中是很难从用户

译注 4：*http://en.wikipedia.org/wiki/Keepalive*。

那里收集和统计到准确的加载时间的。这没有单一的解决方案，相反，需要多种技术，包括测量实际流量、水桶测试（译注 5）和脚本或人工测试。但问题是所有这些技术都需要大量的金钱和时间，成本非常高昂。

我创建 Hammerhead（*http://stevesouders.com/hammerhead/*）是为了使开发者在开发的早期更容易测试加载时间。Hammerhead 是 Firebug 的扩展。如果我们需要测试一组网页，只要把它们的 URL 输入到 Hammerhead 中，随后就能得到所期望的测量数据。图 A-5 展示了一个例子。

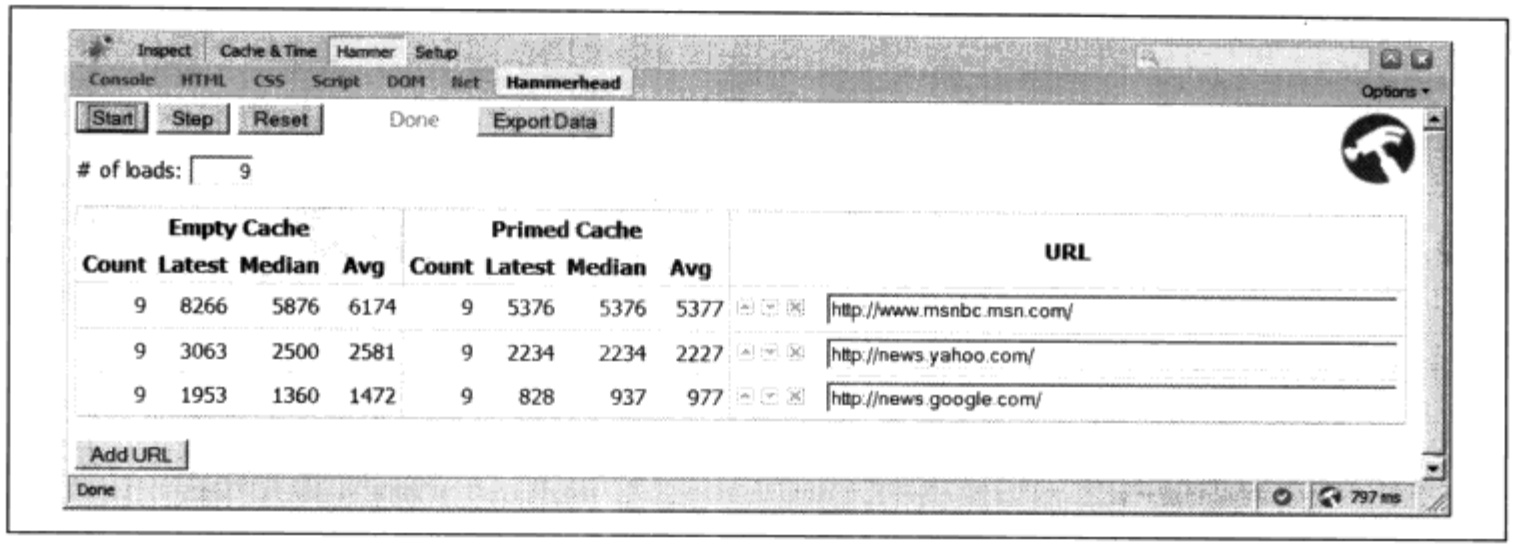


图 A-5：Hammerhead

Hammerhead 按照指定次数加载每个 URL 并记录每次测量结果，然后给出平均和中间的加载时间。页面会通过使用和不使用缓存两种方式进行加载。（Hammerhead 会进行缓存的管理）。虽然 Hammerhead 的测量结果仅仅是在一套测试环境（你的开发环境）下收集的，但它提供了快速和简便的方法来比较两个或更多网页，也不失为一种选择。

Smush.it

Smush.it（*http://developer.yahoo.com/yslow/smushit/*）提供了分析和优化网页中图片的服务。它是由 Stoyan Stefanov 和 Nicole Sullivan 一起开发的，他俩也是本书第 10 章的作者。Smush.it 能告诉你通过优化图片节约了多少字节，如图 A-6 所示。它甚至还可以为你生成一个优化后的图片压缩包，以方便下载。Smush.it 也有书签和 Firefox 扩展，所以你能很容易地在浏览器中使用相似的功能。

译注 5：水桶测试（Bucke Testing）是一套衡量不同产品设计对网站数据影响的方法。基本前提是同时运行两个由一个或多个网页组成的版本，以便测量这两个版本在点击量、流量、转换率等上的不同。水桶测试提供了一种方式，它分配少量流量（通常不到 5%）给不同的用户界面，这样如果新的用户界面有意想不到的负面效果也不会影响我们的底线。这是一种尊重用户体验且能预防风险的方法，像 Yahoo!首页、Amazon 首页、Ebay 首页和 Google 搜索结果页的改版都大量采用这种做法，但国内很少有这样的案例。

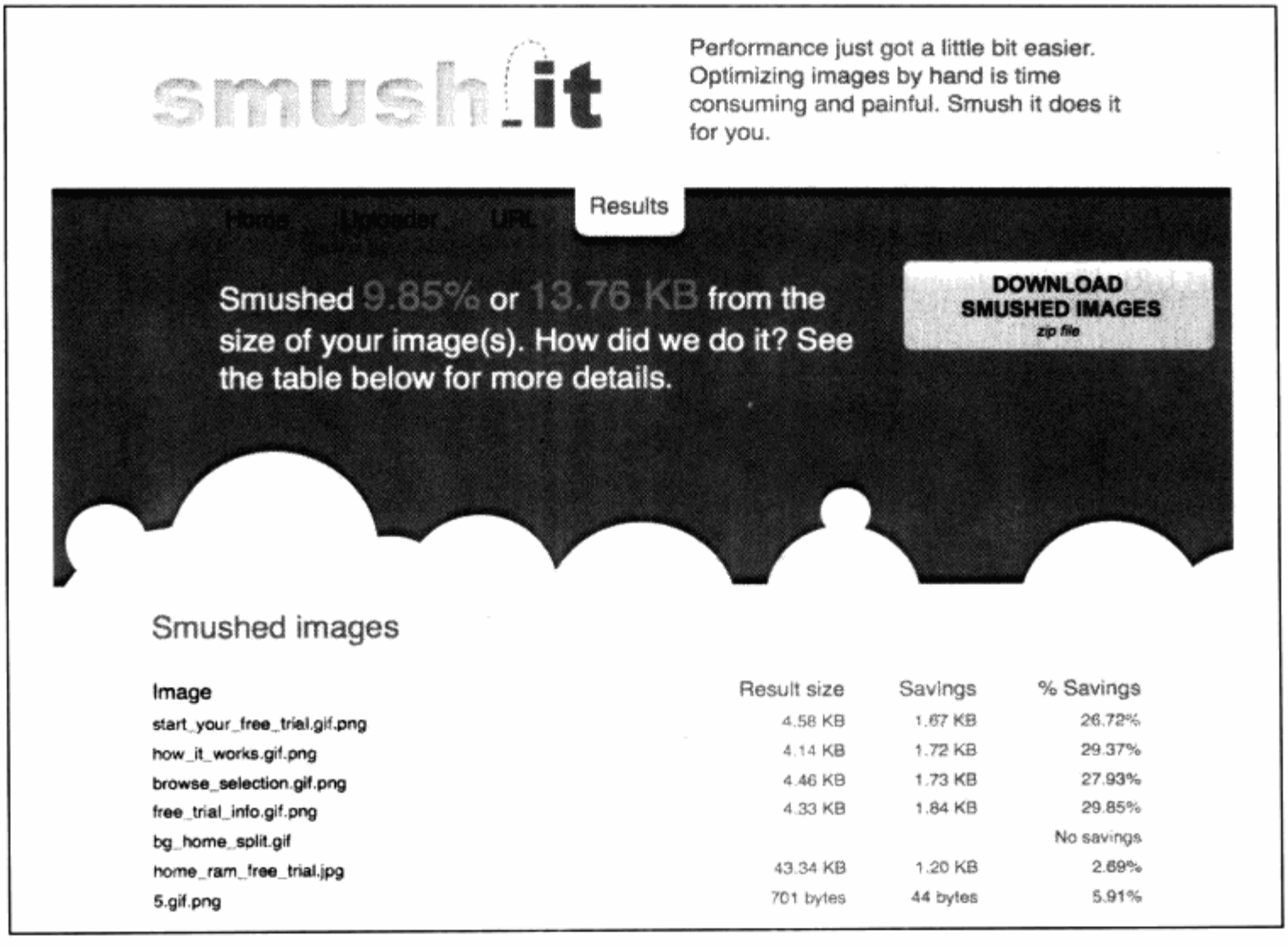


图 A-6：Smush.it

Cuzillion

我几乎每天要思考或被人问起性能的边界情况（译注 6）。如果外部脚本之间有一个内嵌脚本，它们会并行加载吗？如果它们之间有一个内嵌脚本和一个样式表又会怎样呢？它们在 Firefox3.1 和 Chrome2.0 中的行为一致吗？

我使用 Cuzillion（*http://stevesouders.com/cuzillion/*）来取代为每个出现的边界情况写一个新的 HTML 页面的做法，如图 A-7 所示。它有一个图形化的网页“化身”，你可以拖放使用不同类型的资源（外部脚本、内嵌脚本、样式表、内嵌样式块、图片和 iframe），点击一个资源可以打开各种各样的配置设置，比如设置加载资源使用的域和它多久以后才响应。

译注 6：它们只会在一些非常特殊的情况下发生，其中遇到的都是最奇特的问题，我们称之为边界情况。

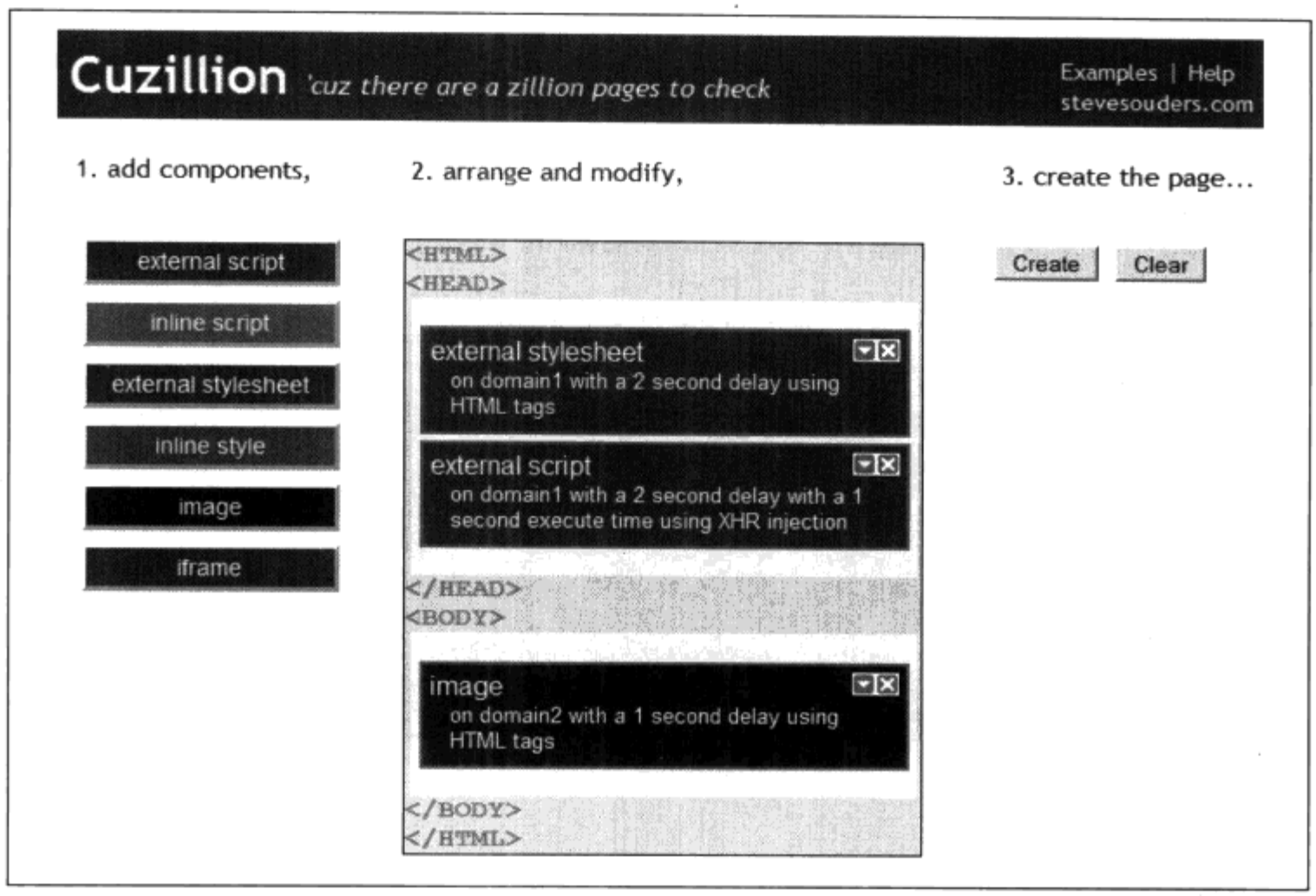


图 A-7：Cuzillion

我在写第 4 章时开发出了 Cuzillion。我要测试成百上千条用例，开发一个测试框架让我只用很少的时间就可以完成这项工作。这个名字来自口头禅："因为有无数的网页要检查。(cuz there are a zillion pages to check.)"

UA Profiler

当 Google 发布 Chrome 的时候，Dion Almaer（第 2 章的合著者）问我能否从性能角度去检查它。与其手动一步一步地对 Chrome 进行测试，不如创建一组 HTML 页面，其中每个都包含一个特殊的测试：并行加载脚本，预取链接（译注 7）工作等。然后我将这些网页链接在一起，使其自动跑完所有的测试。

如图 A-8 所示，UA Profiler（*http://stevesouders.com/ua/*）是一组浏览器的性能测试集合。除了为浏览器提供一个性能测试套件外，UA Profiler 还将收集到的测试结果与更大的 Web 社区共享。任何人都可以使用 UA Profiler 的 Web 客户端（只要它支持 JavaScript），并为结果数据库贡献第三方数据。由于允许社区来执行这些测试，我节省了运作一个回归测试实验室的成本，而且还获得了一个更多元化测试环境下的结果。

译注 7：链接预取（Link prefetching）是一种某些浏览器（比如 Firefox3.0 和 3.5）实现的符合标准草案的机制，它利用浏览器的空余时间下载或预取用户在不久的将来可能使用的文档。详情请看 *http://en.wikipedia.org/wiki/Link_prefetching*。

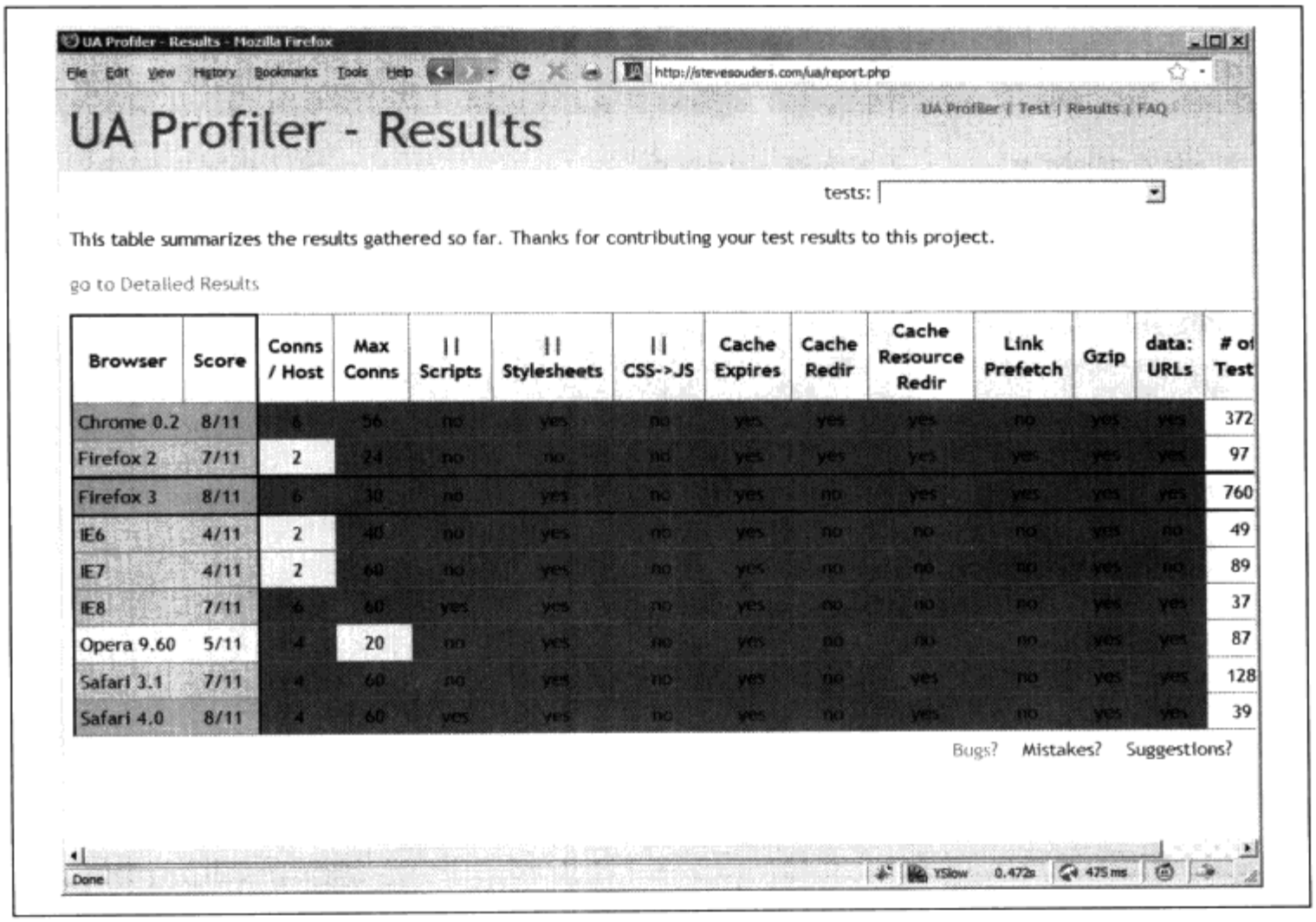


图 A-8：UA Profiler

对于 Web 开发者来说，UA Profiler 在确定指定浏览器如何执行特定优化时很有用。例如，如果为一个重定向增加未来缓存头，但它似乎依旧没有被缓存，那么就可以使用 UA Profiler（译注 8）来检查你正在使用的浏览器是否支持重定向缓存。

译注 8：在翻译的时候，UA Profiler 的开发者也就是本书的作者已经不再支持它的后续开发，转而支持其下一代版本：更加强大的 Browserscope。详情请看 *http://www.browserscope.org/*。

索引
Index

欢迎大家提出建议来改善索引内容。建议请发 E-mail 到 *index@oreilly.com*。

D

E

F

G

H

I

J

K

L

R

S

T

U

V

W

X

Y

Z

作者简介

Steve Souders 现在 Google 工作，负责 Web 性能和开源组织。他的书《高性能网站建设指南》和《高性能网站建设进阶指南》阐述了他在 Web 性能上的最佳实践，以及这些实践背后的研究和真实结果。Steve 是 Firebug 的性能分析扩展 YSlow 的创建者，该工具已被下载超过 100 万次。他也是 Web 性能与运作会议 Velocity 的联合主席，该会议由 O'Reilly 赞助；他还是 Firebug 工作组的联合创始人。此外，Steve 在斯坦福大学教授“CS193H: High Performance Web Sites”课程，并经常在诸如 OSCON、SXSW、Web 2.0 Expo 和 Ajax Experience 等技术会议上发表演讲。

Steve 曾就职于 Yahoo!，担任首席性能工程师，被称为 Yahoo!超级巨星。这期间，他在 Yahoo!开发者网络上发表了大量关于Web性能的博客文章。Steve在该公司多个平台和产品工作过，包括负责 My Yahoo!的开发团队。在加入 Yahoo!之前，Steve 也曾在几个中小规模的创业公司工作，其中 Helix Systems 和 CoolSync 是与他人合创的。他还曾就职于 General Magic、WhoWhere?和 Lycos。

在 20 世纪 80 年代初，Steve 发现了 Artificial Intelligence 的 Bug，并在几个公司里对机器学习进行了研究，期间出版过作品和出席相关会议。他在美国弗吉尼亚大学获得系统工程学士学位，然后从斯坦福大学获得管理科学与工程硕士学位。

Steve 兴趣广泛。他与 NBA 和 WNBA 队员一块儿打过篮球，也是 Universal Studios Internet Task Force 成员，还创下过一项吉尼斯世界纪录，甚至重建了一个有 90 年历史的马车车库。他拥有一位非常出色的妻子和 3 个女儿。

封面说明

本书封面动物为濒危动物——印度黑羚（即黑背羚，印度羚属），主要分布于印度地区。雄性印度黑羚头部有一对呈 V 字形分开的螺旋长角，可达 71 厘米长；头颈背部及四肢外侧为黑色或深棕色，腹部和眼圈呈白色。雌性头颈及身体外侧为浅棕色，无角。它们通常群居于草原上，每群数量大约为 15~20 只，食草，兼食多汁性果类。印度黑羚是世界上奔跑速度最快的动物之一，在开阔的平原上速度可达到每小时 72.4 公里，它擅于通过长距离飞奔以逃脱肉食动物的追捕。

在 18 世纪至 20 世纪初，印度黑羚被大肆捕杀。1932 年，美国德克萨斯州引入印度的几种鹿和羚羊用于狩猎和繁殖，其中就包括印度黑羚。今天这些物种主要生活在德州的私人狩

猎场和丘陵地区。

现在，由于它们在德州大量繁殖，数量已超过 19 000 头，其中许多被送回印度的天然栖息地。

得益于 1972 年通过的印度野生动物保护法，目前印度本土的黑背羚数量维持在五万头左右，此外在德州等地还存有约四万三千头。但人类的偷猎和不断侵占生存空间对其仍是重大威胁。不过，2006 年印度影星萨尔曼·汗因非法猎杀两头黑背羚而被判处 5 年监禁，这个案件引起了人们对黑背羚保护的更多关注。在印度神话中，黑背羚被认为是月亮神禅德拉玛（Chandrama）的坐骑，人们相信它所生活之处将被赐予繁荣幸福。

封面图片来自多佛画报档案。

博文视点O'Reilly系列

《JavaScript语言精粹》

Douglas Crockford 著

赵泽欣 鄢学鹃 译

- 雅虎资深JavaScript架构师Douglas Crockford倾力之作。
- 向读者介绍如何运用JavaScript创建真正可扩展的和高效的代码。

《Web信息架构》

Peter Morville, Louis Rosenfeld 著

陈建勋 译

- 信息架构领域权威著作。
- Web站点开发参考。
- 九年三版，经典巨著。

《PHP程序设计（第2版）》

Rasmus Lerdorf, Kevin Tatroe 等 著

陈浩 胡丹 徐景 译

陈浩 审校

- 包含PHP创始人Rasmus Lerdorf等PHP专家的独特见解

《高性能网站建设指南》

Steve Souders 著

刘彦博 译

- 阐述提高网页效率的14条准则

《Xquery权威指南》

Priscilla Walmsley 著

王银辉 译

- 用简洁的语言深入浅出地介绍了XQuery方方面面的知识。
- 学习XQuery的教材和参考指南。
- 提高与优化XML数据检索性能的参考书。

《可视化数据》

Ben Fry 著

张羽 译

- 以亲身实践为基础，教你从零开始学习创建数据的可视化步骤。
- 细致讲解、实例论述、适用面广。

《Designing Interfaces中文版》

Jenifer Tidwell 著

De Dream' 译

- 全面总结交互设计经验和智慧
- 彻底展现交互设计原则和实践

《Web界面设计》

Bill Scott Theresa Neil 著

李松峰 译

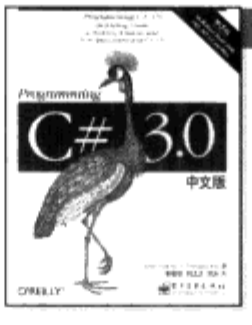
- 业界顶尖专家Bill Scott和Theresa Neil 30年软件设计开发经验积累。
- 涵盖了现代Web 界面设计的普遍规则和最佳实践。

《MySQL Cookbook 第2版 中文版》

Paul DuBois 著

瀚海时光团队 译

- 囊括大量简练、精辟的代码段和可用示例
- 新增MySQL 5.0的内容及其强大的新特性。



《Programming C# 3.0中文版（第5版）》

Jesse Liberty,Donald Xie 著

李愈胜 刘卫卫 汪泳 译

- O'REILLY国际畅销书《Programming C#》系列，第5版隆重上市！
- 本书实践性强，从编程实践的角度讲解C# 3.0，深入浅出地讲述C#和如何用C#编写NET应用程序。

《CSS实战手册》

David Sawyer McFarland 著

俞黎敏 译

- 第17届Jolt Awards(震撼大奖)获奖图书
- CSS三剑客之实战之剑

《构建可扩展的Web站点》

Cal Henderson 著

徐宁 译

- Web 2.0代表网站Flickr总架构师Cal Henderson经典力作

博文视点Web 2.0 必修系列

《高性能网站建设指南——前端工程师技能精髓》

Steve Souders 著
刘彦博 译

- 深度阐述前端工程师的技能精髓
- 详细解读提高网页效率的14条准则
- 分享作者多年网站性能方面的丰富经验

《Web信息架构：设计大型网站（第3版）》

Peter Morville,Louis Rosenfeld 著
陈建勋 译 范炜 审校

- 信息架构领域权威著作
- Web站点开发参考
- 九年三版，经典巨著

《Restful Web Services中文版》

Leonard Richardson,Sam Ruby 著
W3China徐涵 李红军 胡伟 译

- 利用Web的强大功能构造可编程应用
- 将REST设计理念应用于真是Web服务

《构建可扩展的Web站点》

Cal Henderson 著
徐宁 译

- Flickr.com主力架构师Cal Henderson倾力之作
- 为你揭开Flickr.com构建之谜
- 帮你解读Web应用程序扩展之道
- 助你构建最优秀的Web 2.0应用

北京博文视点（www.broadview.com.cn）资讯有限公司成立于2003年，是工业和信息化部直属的中央一级科技与教育出版社——电子工业出版社（PHEI）下属旗舰级子公司，在六年的开拓、探索和成长中，已成为中国颇具影响力的专业IT图书策划和服务提供商。

六年来，博文视点以开发IT类图书选题为主业，励精图治、兢兢业业，打造了一支团结一心的专业队伍，并形成了自身独特的竞争优势。一直以来，博文视点始终以传播完美知识为己任，用诚挚之心奉献精品佳作，年组织策划图书达300个品种，同时开展相关信息和知识增值服务，赢得了众多才华横溢的作者朋友和肝胆相照的合伙伙伴，已经成为IT图书领域的高端品牌。

我们的理念： 创新专业图书服务体制；培养职业策划图书服务队伍；打造精品图书品牌；完善全面出版服务平台。

我们的目标： 面向IT专业人员的出版物提供相关服务。

我们的团队： 一个整合了专业技术人员和专业服务人员的团队；一个充满创新意识和创作激情的团队；一个不断进取、追求卓越的团队。

我们的服务： 善待作者　尊重作者　提升作者

我们的实力： 优秀的专业编辑队伍
全方位立体化的强大的市场推广平台
实力雄厚的电子工业出版社的渠道平台

“走出软件作坊独辟蹊经　人道编程之美，
追踪加密解密庖丁解牛　精雕夜读天书。”

路漫漫其修远，博文视点愿与所有曾经帮助、关心过我们的朋友、作者、合作伙伴携手奋斗。未来之路，不可限量！

地址： 北京市万寿路173信箱电子工业出版社博文视点资讯有限公司
邮编： 100036　**总机：** 010-88254356　**传真：** 010-88254356-802

武汉分部地址： 武汉市洪山区吴家湾湖北信息产业科技大厦1402室
邮编： 430074　**总机：** 027-87690813　**传真：** 027-87690013

欢迎投稿： bvtougao@gmail.com
读者邮箱： reader@broadview.com.cn

博文视点官方博客： http://blog.csdn.net/bvbook
博文视点官方网站： http://www.broadiew.com.cn

Broadview®
www.broadview.com.cn

在线有奖读者调查表

《高性能网站建设进阶指南：Web开发者性能优化最佳实践》

http://bv.csdn.net

登录以上网站告诉我们您关于这本书的建议、意见
就有机会获赠博文视点的『新书一本』
并参加年终大抽奖活动

您的支持就是我们创造精品动力的源泉！

欢迎投稿：bvtougao@gmail.com

读者信箱：reader@broadview.com.cn

博文视点更多资源网站：

VSTS虚拟社区：
http://yishan.cc/

《代码大全》资源网站：
http://www.cc2e.com.cn/

博文视点其他博客：
http://www.cnblogs.com/bvbook/
http://bvbook.javaeye.com/