HZ BOOKS
华章科技

Apress®

Web开发
技术丛书

JavaScript 性能优化

度量、监控与可视化

[美] Tom Barker 著　　缪纶　王志璋　王冠华　叶茂　译

Pro JavaScript Performance
Monitoring and Visualization

- 首部系统化阐述JavaScript性能优化的经典著作，拥有20余年经验的Web性能调优专家经验结晶，亚马逊全五星好评
- 从语言特性、浏览器原理、网络传输机制、数据结构等多角度深层探讨影响JavaScript代码性能的根本原因，并给出解决问题的完整流程和解决方案

机械工业出版社
China Machine Press

本书专注于改进JavaScript脚本程序加载速率和响应速度，提供了大量的性能测量方法和工具，更为重要的是，读者可以借助这些方法和工具，改进JavaScript脚本代码的效率，定义最佳性能实践，改善应用程序的工作方式。

本书亮点：

- 应用性能最佳实践，并对结果进行量化的方法和技巧；
- 实用监测和分析工具，如Firebug、YSlow以及WebPagetest；
- 使用WebPagetest、PHP和R跟踪Web性能；
- 创建一个JavaScript库来对运行时性能进行基准评估；
- 在浏览器中优化运行时性能。

Apress®

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/Web开发
ISBN 978-7-111-46022-0
9 787111 460220 >
定价：49.00元

JavaScript 性能优化

度量、监控与可视化

Pro JavaScript Performance
Monitoring and Visualization

[美] Tom Barker 著　缪纶　王志璋　王冠华　叶茂　译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

JavaScript 性能优化：度量、监控与可视化 /（美）巴克（Barker，T.）著；缪纶等译．
—北京：机械工业出版社，2014.3
（Web 开发技术丛书）
书名原文：Pro JavaScript Performance：Monitoring and Visualization

ISBN 978-7-111-46022-0

I. J…　II. ①巴…　②缪…　III. JAVA 语言－程序设计　IV. TP312

中国版本图书馆 CIP 数据核字（2014）第 037980 号

本书版权登记号：图字：01-2013-8507

JavaScript 性能优化：度量、监控与可视化

（美） Tom Barker 著

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：高婧雅
印　　刷：薄城市京瑞印刷有限公司　　版　　次：2014 年 4 月第 1 版第 1 次印刷
开　　本：186mm×240mm　1/16　　印　　张：12.25
书　　号：ISBN 978-7-111-46022-0　　定　　价：49.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

译者序

JavaScript 是比较完善的一种前端开发语言，在现今的 Web 开发（尤其是在 Web 2.0）中应用非常广泛。在 Web 2.0 越来越流行的今天，我们会发现，许多 Web 应用项目都会涉及大量的 JavaScript 代码，并且以后会越来越多。Web 开发过程中经常会遇到一些性能问题，尤其是在针对 Web 2.0 的应用中，应用的性能问题很大一部分都是由于程序员编写的 JavaScript 脚本性能不佳所造成的，其中也包括了 JavaScript 语言本身的性能问题，以及与 DOM 交互时的性能问题。

JavaScript 作为一种解释执行语言，加之它的单线程机制，决定了性能问题是 JavaScript 的软肋，这也是 Web 软件工程师在编写 JavaScript 代码过程中需要高度重视的一个问题，尤其是针对 Web 2.0 的应用。绝大多数 Web 软件工程师都或多或少地遇到过所开发的 Web 2.0 应用的性能欠佳问题，其主要原因就是 JavaScript 性能不足，浏览器负荷过重。但是，解决这种解释执行并且单线程运作语言的性能问题并非易事。

本书从一个体系化的角度对 JavaScript 性能进行考量，从 Web 性能探测工具，到数据的收集整理，再到数据的展示与可视化，最后是性能问题的解决，深入浅出、条理清晰。本书尝试从多个方面综合分析导致 JavaScript 性能问题的原因，并给出适合的解决方案，帮助读者提升 Web 应用的品质。

本书页数了了，内容精炼，但是它承载了 JavaScript 性能方面最为宝贵的经验。不仅从语言特性、浏览器原理、网络传输机制、数据结构等层面分析导致 JavaScript 性能问题的原因，而且介绍多种工具来帮助读者提升开发过程中的工作效率。

翻译本书的过程，也是我们学习和提高的过程。虽然我们一直关注国内外 Web 性能领域的相关动态，但是从未像本书的作者那样，形成一套完整的实践方法体系。每当看到书中作者提供的代码样例，我们都试图在真实环境中实践，以深刻体会作者的意图和思想。但是由于时间和软硬件环境等方面的原因，无法实践书中所有的代码样例。这里也希望读者能够亲自动手实践，这样才能够更快地掌握 JavaScript 性能改善的方法和能力。

本书的翻译组织工作由缪纶全面负责。第 1 ~ 3 章由缪纶和王志璋译校，作者简介、第 4 章、

第 6 章由缪纶和王冠华译校，第 5 章、第 7 章、第 8 章由叶茂译校。

我们在翻译本书的过程中力求行文流畅，但纰漏之处在所难免，请广大读者能够批评指正。关于本书的任何意见和建议，欢迎发送邮件至 lunmiao@tom.com。

最后，希望本书能够帮助业界的同仁们打造出性能优越的 Web 软件产品。

缪　纶

致　谢

感谢我的妻子 Lynn，我的儿子 Lukas，以及我的女儿 Paloma，对于他们的耐心和爱意，表示由衷的感谢。

感谢 Ben Renow-Clarke 让我来编写这本书，并在本书编写过程中给予我支持和指导，从性能方面和最佳实践，到强调量化的结果，在向读者展示如何进行量化处理方面，他的建议也使我受益匪浅。

感谢 Katie Sullivan 和 Chris Nelson，帮我推进本书向好的方向发展。有几次是他们的坚持才让我们不断前进，Katie 帮助我注意到各个章节之间的联系，而 Chris 则让我使每一章都条分缕析。

感谢 Anirudh Prabhu 想到了代码中我没有考虑的情况。因为有他的观点，我们的示例代码才变得丰富起来。

最后，我要感谢我在 Comcast 公司的团队，他们持续不断地提升门槛，激励我努力成为你们这些优秀人中的一员。

目　录

第 1 章

什么是性能

所谓性能是指应用程序运行的速度，它反映了应用程序质量多个层面的问题。我们在谈论 Web 应用程序时，该应用程序呈现给用户所消耗的时间就是我们所说的 Web 性能，应用程序对用户指令的响应速度就是我们所说的运行时性能。接下来我们就看看性能的这两个方面。

在 Web（特别是移动 Web）开发背景下，性能是一个相对较新的主题，但是它绝对早就应该得到关注了。

本书将探讨如何量化和优化 JavaScript 性能，包括 Web 性能和运行时性能。这是至关重要的，因为当你试图解决网站的整体性能时，JavaScript 可能是提升性能最大的地方。作为 YSlow 和 PageSpeed 的缔造者，以及 Web 性能领域的先驱者，Steve Souders 已经通过实验证明了这一点。在实验中，他演示了当删除一个样本网站[⊖]的 JavaScript 代码之后，该网站平均性能提高了 31%。我们也完全可以像 Steve 实验中做的那样，从我们的网站中删除所有的 JavaScript 代码，或者精简 JavaScript 代码，并学习如何评估我们编写的代码的执行效率。

彻底删除 JavaScript 代码是不现实的，所以我们要知道如何使 JavaScript 更加高效。甚至更为重要的是，我们要知道如何创建自动化的工具以跟踪 JavaScript 的效率，并给出可视化的分析和报告。

1.1 Web 性能

当你坐在笔记本前，或使用手持设备，打开 Web 浏览器，输入一个 URL 地址并按下回车键，然后等待页面内容传送过来、呈现在你的浏览器上时，此时所需要的时间取决于 Web 性能。甚于我们的目标，我们将 Web 性能定义为全面反映页面传送并对最终用户可用的时间。

⊖ http://www.stevesouders.com/blog/2012/01/13/javascript-performance/

影响 Web 性能的因素有很多，网络延迟是排在第一位的。你的网络有多快？要得到所需的服务内容，需要往返跳转多少次？服务器要给出多少次响应？

为了更好地理解网络延迟，先来看看完成一个 HTTP 事务所需要的步骤（见图 1-1）。

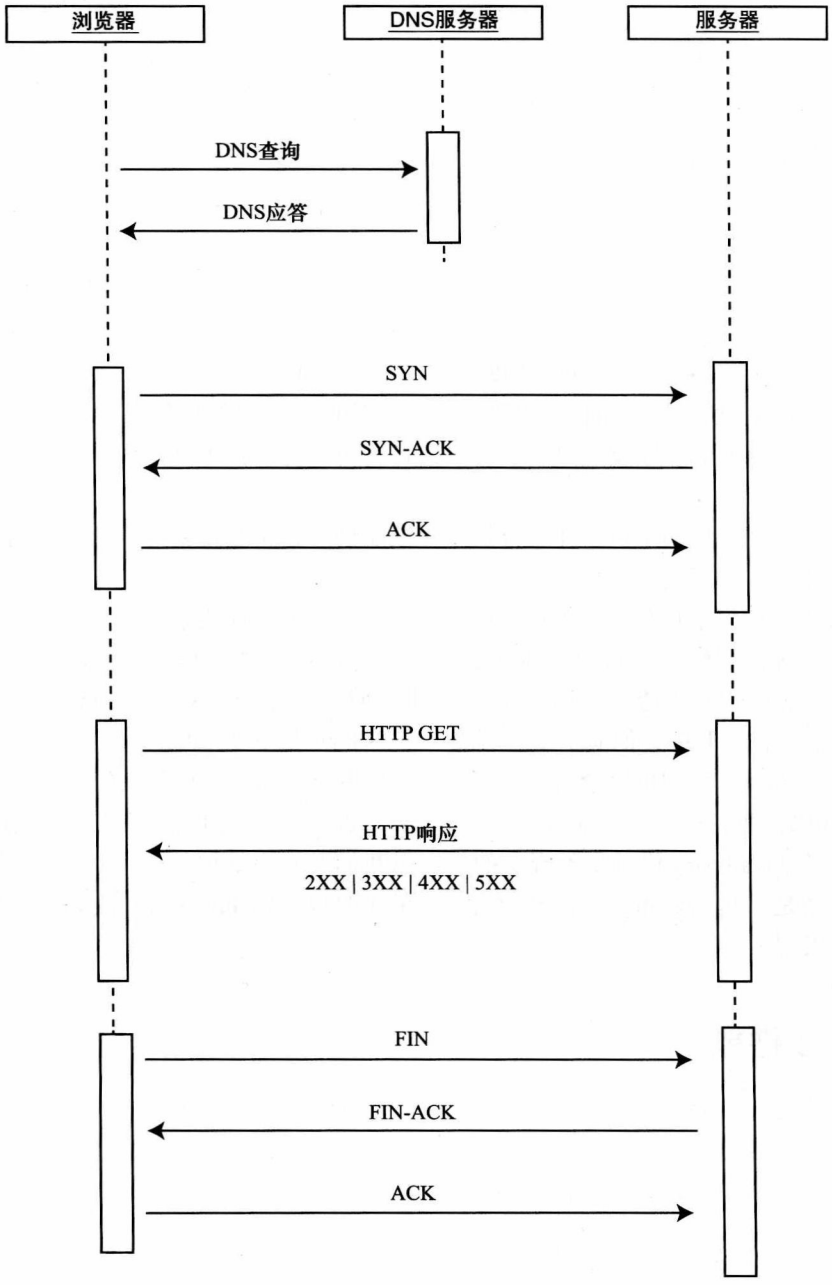


图 1-1　Web 页请求以及对 Web 页面中每个远程对象重复请求的网络事务流程图

当发送一个URL请求时，不管这个URL是Web页面的URL还是Web页面上每个资源的URL，浏览器都会开启一个线程来处理这个请求，同时在远程DNS服务器上启动一个DNS查询。这能使浏览器获得URL请求所对应的IP地址。

注意

线程是控制应用程序执行的有序单元。无论何时，应用程序执行的任何操作，都要调用一个线程。一些应用程序是多线程的，这也意味着这样的应用程序可以同时做多件事情。一般来讲，浏览器的每步操作至少要调用一个线程。也就是说，线程执行的步骤（我们概括为连接、下载以及页面渲染这一过程）是按顺序依次处理的。

接下来，浏览器与远程Web服务器通过TCP三次握手协商来建立一个TCP/IP连接。该握手过程包括一个同步（Synchronize）报文，一个同步－应答（Synchronize-Acknowledge）报文以及一个应答（Acknowledge）报文，这三个报文在浏览器和远程服务器间传递。该握手过程首先由客户端尝试建立通信，而后服务器应答并接受客户端的请求，最后由客户端发出该请求已经被接受的应答报文。

这一握手过程很像军事上的双向无线电语音通信过程。想象一下位于双向无线电通信两端的双方，他们是如何知道对方已经完成他们发送的消息，是如何知道通信双方（此时）不能进行对话（否则会产生会话冲突），又是如何知道一方能够理解另一方发来的消息呢？这一通信过程在语音通信中已经标准化了，其中每一个关键术语都有其微妙的含义。比如，Over表示一方已经结束讲话并等待回应，Roger表示对方已经理解消息的含义了。

像所有其他通信协议一样，TCP握手协议仅仅是定义多方通信的一个标准化的方式而已。

TCP/IP模型

TCP即传输控制协议（Transmission Control Protocol）。这个协议在TCP/IP模型中用于定义客户和服务器之间的通信如何处理，它主要定义了数据如何分段，以及如何处理我们先前描述的握手过程（见图1-1）。

TCP/IP模型是一个4层模型，描绘了不同协议之间的关系，这些协议定义了数据在互联网上共享的方式。TCP/IP模型的规范由互联网工程任务组（Internet Engineering Task Force）维护，该规范由两个RFC（Request For Comment）文档组成，可以在http:// tools.ietf.org/html/rfc1122和http:// tools.ietf.org/html/rfc1123上找到。

4层的TCP/IP模型，按照距离最终用户由远及近的顺序分别为：网络接入层[⊖]、网络层、传输层，以及应用层。

- 网络接入层控制网络中硬件之间的通信。
- 网络层负责网络的寻址和路由，并获取IP地址和MAC地址。

⊖ 又称为“网络接口层”。——译者注

- 传输层也就是 TCP(或者 UDP) 通信所在位置。
- 应用层负责客户端和服务器所要用到的最上层的通信，比如 HTTP 和用于邮件客户端的 SMTP。

如果将 TCP/IP 模型和流程图进行对比，我们会发现浏览器必须自上而下遍历模型才能提供我们所需要的页面。

一旦 TCP/IP 连接建立，浏览器会通过该连接向远程服务器发送 HTTP 的 GET 请求。远程服务器找到资源并使用 HTTP 响应返回该资源，值为 200 的 HTTP 响应状态表示一个正确的响应。如果服务器没有找到资源或在解析资源时发生错误，或者如果请求被重定向了，HTTP 响应会返回相应的状态值。完整的状态值代码可以在 http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html 上找到，其中最常用的状态代码如下：

- 200 表示来自服务器的成功响应；
- 404 意味着服务器没有找到请求的资源；
- 500 表示执行请求时发生了错误。

此时，Web 服务器提供资源服务，客户端开始下载资源。与此同时，网页的全部负载，包括所有图片的大小、CSS 和 JavaScript，粉墨登场了。

页面的总大小很重要，不仅仅因为页面大小关系到下载时间，还由于 IPv4 和 IPv6 协议

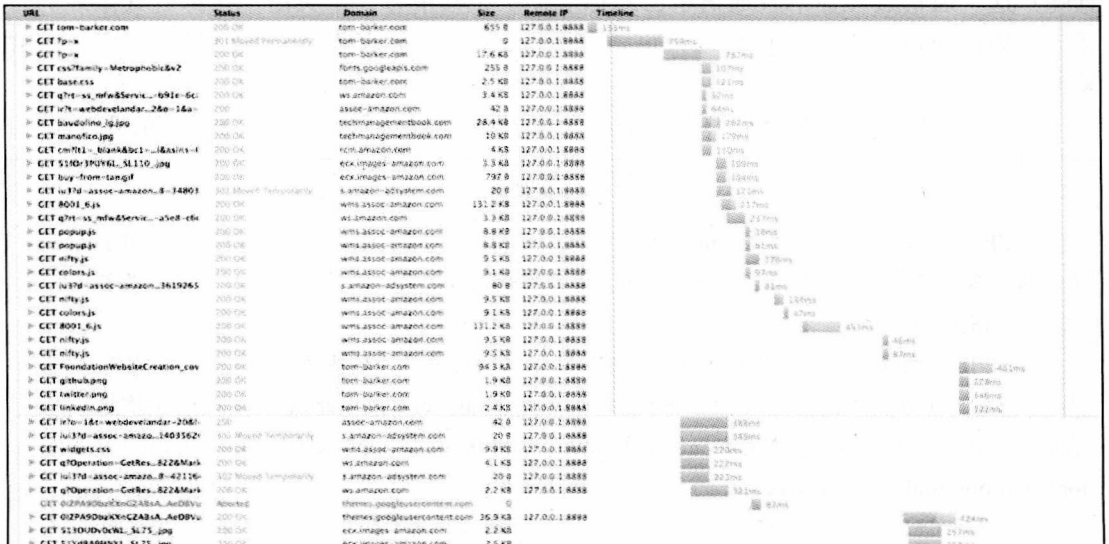


图 1-2 浏览器架构

规定一个 IP 包的最大值为 65 535 字节。如果将你的总页面大小转化为字节数，再除以最大 IP 包的字节值，你就会得到服务器发送全部负载所需的响应次数。

将网页中所有对象都加载完所需的 HTTP 请求数是导致网络延迟的另一个因素。页面中的每一项资源——每一个图片、外部 JavaScript 文件和 CSS 文件，都需要从服务器到本地走个来回。每项资源的请求都会开启一个新的线程和一个如图 1-1 所示的新流程（flow）实例，而每一个实例又会产生 DNS 查询、TCP 连接、HTTP 请求和响应所带来的消耗，最后还要加上单纯为了传输各项资源所耗费的时间成本。

图 1-2 旨在说明，当简单的概念成指数级增长时，就会造成性能大幅下降。

瀑布图是一个用来展示请求一个页面以及页面中包含的全部资源所花费时间的工具。它显示了构成页面所需的每一项资源的 HTTP 交互过程，包括资源大小、下载时间长短和下载的顺序。说得更透彻点，瀑布图的每一个条柱表示一项正在下载的资源。条柱的长度对应为获取资源而进行连接和下载所花费的时间。图上有一个序列的时间轴项，因此，顶端的条柱代表的是第一个下载的资源，而最下边的条柱表示最后一个下载的资源，时间轴的最左端表示连接起始时间，最右端表示结束时间。第 2 章，我们会讨论测量和影响性能的工具，到时我们会介绍瀑布图的更多内容。

1.2 解析与渲染

除了网络之外，另一个影响 Web 性能的因素是浏览器的解析与渲染，而影响浏览器解析和呈现的因素有很多。为了更好地理解这一概念，我们先来看看当浏览器解析和呈现 Web 页面时，它的架构是什么样的（见图 1-3）。

现在大多数浏览器都是如下架构：处理 UI（用户界面）的代码（包括了地址栏和回退（history）按钮），用于解析和绘制页面中所有对象的渲染引擎，解释 Javascript 的 JavaScript 引擎，以及一个处理 HTTP 请求的网络层。

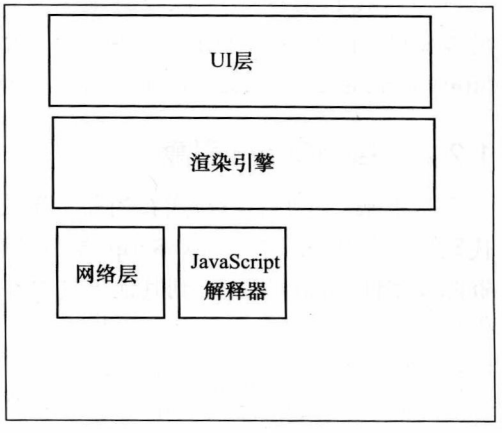


图 1-3 浏览器架构

由于浏览器是自上而下读取内容的，因此放置资源的位置会影响网站的访问速度。比如，如果将 JavaScript 标签放在 HTML 内容的前边，浏览器就会先调用 JavaScript 解释器对 JavaScript 进行解析，完成之后才会渲染其余的 HTML 内容，对最终用户来说，这会导致页面可用性的延迟。

作为一个 Web 开发人员，浏览器是必不可少的，所以你应该精通每一种渲染引擎和 JavaScript 引擎。你最好下载一份开源代码（参阅 1.2.1 节提供的相关网址），并花时间通读其中的一些源代码，我认为这是值得花时间的。

如果你喜欢冒险，你可以把自己的工具或日志添加到源代码中，构造一个你自己的引擎，执行你自己的自动化性能测试。

1.2.1 渲染引擎

我们来看看渲染引擎除此之外更为广泛的用途。相比于浏览器，其实多探究一下渲染引擎更重要。浏览器架构可以划分为多个模块，一直以来，浏览器制造商也是通过组合模块来构造浏览器的。除了浏览器之外，还有很多工具可以用来渲染 HTML，包括 email 客户端和其他应用程序中的 Web 组件。有了自己可分发的渲染引擎，浏览器制造商就可以复用引擎，或者发放许可给其他公司使用。如果知道了某个软件包使用了哪种渲染引擎，开发人员通常就会明白该软件包能够提供什么样的功能。

Firefox 及其派生产品和同源产品（比如 Thunderbird、Mozilla 的 E-mail 客户端）使用 Gecko 这款渲染引擎，https://developer.mozilla.org/en/Gecko 提供了该引擎的下载地址。作为原 Netscape 渲染引擎的继任者，早在 1997 年，在 Mozilla 计划生产出真正的实体产品之前，Netscape 就首次开发出了 Gecko。

Chrome 和 Safari 使用 Webkit 这款渲染引擎，用于大多数的移动 Web 开发，在安卓设备以及 iOS 设备的移动 Safari 和 Kindle Fires 的 Silk 浏览器上，Webkit 被作为布局或渲染引擎。http://www.webkit.org/ 提供了 Webkit 的下载地址。最初，作为 KDE 系统的 KHTML 的分支，WebKit 于 2001 年应用在 Apple 设备上。2005 年，WebKit 开放了其源代码。

应用在桌面系统、移动设备甚至所有的 Nintendo（任天堂）终端（NDS，Wii）之上的 Opera 浏览器，都使用 Presto 这款渲染引擎，2003 年 Opera 7 引入了 Presto。http://dev.opera.com/articles/view/presto-2-1-web-standards-supported-by/ 提供了关于 Presto 的更多信息。

最后我们来看看 IE 以及其他微软产品，例如 Outlook，使用代号为 Trident（三叉戟）的 MSHTML 作为渲染引擎。1997 年，微软最初在 IE 4 上引入了 Trident 并一直沿用至今。http://msdn.microsoft.com/en-us/library/bb508515 上可以找到关于 Trident 的文档。

1.2.2 JavaScript 引擎

接下来，我们看看应用在当前主流的浏览器上的 JavaScript 引擎。同渲染引擎以及其他代码模块化概念一样，JavaScript 解释器也应用了模块化这一概念。解释器可以共享给其他资源或属性，也可以嵌入到其他工具中。你也可以在你自己的项目中使用开源解释器，这样你就可以构造自己的静态代码分析工具，或者为了允许用户在应用程序中编写脚本以添加某些功能而提供 JavaScript 支持。

SpiderMonkey 是 Mozilla 公司开发的一款 JavaScript 引擎，应用在 Firefox 上。Brendan Eich 在 1996 年开发了这款引擎，一直以来，SpiderMonkey 都是 Netscape 和 Firefox 的 JavaScript 解释器。https://developer.mozilla.org/en/SpiderMonkey 提供了 SpiderMonkey 的文档。Mozilla 提供了如何将 SpiderMonkey 嵌入到其他应用程序的文档，在 https://developer.

mozilla.org/en/How_to_embed_the_JavaScript_engine 上可以找到这些文档。

Opera 从 2010 年开始，使用 Carakan 作为 JavaScript 引擎。在 http://my.opera.com/dragonfly/blog/index.dml/tag/Carakan 上可以找到关于 Carakan 的更多信息。

Google 的 Chrome 使用开源的 JavaScript 引擎，在 http://code.google.com/p/v8/ 上可以找到该开源代码。文档资料可以在 https://developers.google.com/v8/intro 上找到。

Safari 使用 JavaScriptCore（有时也称为 Nitro）作为它的 JavaScript 引擎。JavaScriptCore 的更多内容可以在 http://www.webkit.org/projects/javascript/index.html 上找到。

IE 使用 Chakra 作为 JScript 引擎。注意，JScript 一开始就是由微软作为 JavaScript 的逆向工程版本推出的，因此微软一直都让 JScript 在整个生态系统中发出自己的声音。Douglas Crockford 在 http://www.yuiblog.com/blog/2007/01/24/video-crockford-tjpl/ 公布了这一细节。JScript 是 ECMAScript 实施规范的合法实现，Chakra 只支持该规范的部分内容，而大多数其他的 JavaScript 引擎并不支持该规范，特别是在条件编译（conditional compilation）的情况下（后边会看到关于条件编译的讨论）。

当讨论和优化网站整体的 Web 性能时，以上所有这些细微的内容都是要考虑的。

Mozilla 的 JavaScript 团队也维护了一个网站：http://arewefastyet.com/，该网站以时间为基准对比了 V8 和 SpiderMonkey，还对比了用基准测试套件运行这两个引擎所得出的结果。

条件编译

条件编译是某些开发语言的特性，通常它允许开发语言编译器依据编译时的指定条件，生成不同的可执行代码。这对于 JavaScript 来讲有些用词不当，因为 JavaScript 显然是解释性语言，而非编译性语言（它无法运行在内核级，只能在浏览器中运行），但是这个词在这里有不同的意义。

条件编译允许编写只有在特定条件下才会被解释的 JavaScript 代码。默认条件下，JScript 关闭了条件编译，要想运行它，我们需要提供一个解释级的标记：@cc_on。如果我们打算编写条件编译的 JavaScript 代码，我们需要将代码写在注释里，这样在其他不支持条件编译的 JavaScript 解释器上运行时也不会发生中断。

下面是一个 JScript 条件编译的例子：

```
<script>
var useAX = false; //use ActiveX controls default to false
/*@cc_on
@if (@_win32)
    useAX = true;
@end
*/
</script>
```

1.3 运行时性能

所谓运行时（runtime）是指应用程序执行或运行的时长。运行时性能是指应用程序运行时对用户输入的响应速度（比如保存特性或访问 DOM 中的元素时）。

运行时性能受很多因素影响，从完成特定功能所采用的算法效率，到优化方法，从解释器或浏览器渲染引擎的优化或不足，到有效内存管理和 CPU 使用率，再到设计时同步和异步操作之间的选择，都会对运行时性能产生影响。

就应用程序所有指标来看，运行时性能只是一个主观感觉，但是你仍可以采用一些手段来对整个用户体验的特点和趋势进行跟踪，并对异常情况进行分析。你也可以做一些多元测试实验，看看当特定的用户群使用浏览器时，哪种方法可以获得最大程度的性能增益。

第 4 章会探讨这些概念。

1.4 为什么性能如此重要

第一个原因是显而易见的——响应更快速的网站会带给最终用户更好的体验。理论上讲，好的体验等同于好的用户满意度。

更快，也意味着用户在放弃会话之前，有希望更快地访问到你的网站特性。放弃会话或者放弃访问网站有很多原因：页面加载时间太长了，用户失去了兴趣，浏览器崩溃，或者其他众多原因之一。

计算出你自己网站的放弃率是很简单的。只需要获得正常操作你的网站的用户总数，这些正常操作包括：购买一件商品，注册一个新账号，加售一项服务，查看其他页面，点击主页上的按钮，以及你网站上提供的所有高层次的功能。将这个数除以访问总数，然后用 1 减去这个除数，再乘以 100，这样就可以得到还未完成网站操作就放弃网站的流量百分比。

$$[\text{放弃率}] = (1 - ([\text{成功访问数}] / [\text{访总数}])) \times 100$$

举一个例子，假如我们有一个网站，假定就是一个用户注册网站。这个网站最终的目的就是让用户创建新账户——当用户有了账户之后，我们就可以根据他们的个人喜好定制一些东西，我们可以根据他们的购买习惯有针对性地发布广告，而且我们可以依据他们购买商品以及浏览网站的历史信息进行相应的推荐。不管出于什么样的目的，我们都希望用户能注册进来，这是衡量我们网站成功与否的标准。用户一旦提交申请表，我们就会调用一个 PHP 脚本更新数据库，在 User 表中创建一条新记录，然后跳转到主页。

到这里，我们看一下页面访问指标，我们看到有 100 000 个用户访问量；在我们的算法中，这个值就是访问总数。我们再看看数据库中创建的用户数，我们知道有 30 000 个用户。这样用我们的算法计算出放弃率为 70%：

$$(1 - (30\,000/100\,000)) \times 100 = 70$$

提高性能可以降低放弃率，会给你的网站带来显著的效益。很多典型例子告诉我们，Web 性能不佳会导致企业效益受到损失（会看到放弃率上升）。

Alberto Savoia 发表过一篇论文，详细描述了性能对放弃率产生的影响，Keynote 把这篇论文放在 http://www.keynote.com/downloads/articles/tradesecrets.pdf 上。在 Gomez 的白皮书“为什么 Web 性能如此重要”（Why Web Performance Matters，http://www.gomez.com/pdfs/wp_why_web_performance_matters.pdf）中，他详细阐述了仅通过在 Web 性能中增加页面延迟，放弃率就可以从 8% 上升至 38%。

你也可以使用上述计算公式做一下实验，量化和推断一下优化网站性能所带来的投资回报率。

1.5 工具与可视化

本书很大一部分内容是关于如何将工具放置到代码中，以及如何使用数据可视化来描绘结果的。实际上，这也是本书的关键所在。对于提高性能而言，没有一劳永逸的方法。一个人看到的结果未必与另一个人看到的完全一样，原因是他们可能针对完全不同的用户群或者使用完全不同的浏览器。

也许你的用户由于公司政策而被限定只能使用 IE 浏览器，或者你的用户是浏览器早期的使用者，因此你会有很多使用测试版本浏览器的用户，不同的解释引擎或渲染引擎会有不同的优化方法，甚至会在解释引擎或渲染引擎中有 bug 存在。

不管什么情况，结果都会不同。由于一天中不同时间段的网络连接速度不同（用户在工作时或者用户在家中时），或者因为连接方式不同（线缆或者拨号），又或者因为其他的原因，它们测量起来也会有变化。

但是，通过测量结果，并将测量结果可视化，看看数据的整体特点，你就可以依据实际数据和变化趋势对网站进行微调。

作为一门学科，数据可视化最近才发展起来。它不再仅仅是数学、理论，或者绘图领域的事情。我还记得我第一次关注数据可视化能做什么时的情景。当时我是在 Santa Clara 举办的 Velocity 大会[⊖]上，参加者都是我的同行。会议上，John Rauser 讲述了他和他的团队在 Amazon 如何通过分析生产日志调试产品问题的事情。在这次讲话中，他谈到，有时候需要提取出细微到单个用户级别粒度的数据，将这些数据在硬拷贝（Hard Copy）[⊜]上罗列起来，然后通过这些数据侧面地反映出它的整体状况，这个状况即是数据的真实情况。

我受这个故事启发，从那时起，我就开始仔细观察我生活的每个方面。

在工作中，我使用数据可视化作为管理工具来运营我的公司。在本书中，我会创建一些图表，这些图表就是源自我日常管理团队使用的图表。

闲暇时，我会研究我的举重健身记录变化趋势，看看增加了多少，降低了多少，什么时候是平稳的（见图 1-4）。我会通过基于时间序列的交叉引用数据，看看当举重量上升时，

⊖ Velocity 大会是由 O'Reilly 主办的技术性大会，主要面向 Web 开发人员，提供网站优化的最佳实践。——译者注

⊜ 当资料经由打印机输出至纸上称为硬拷贝，若资料显示在荧幕上则称为软拷贝。——译者注

我生活的其他方面会有什么样的影响。数据分析在举重中确实是一个非常重要的概念，它能使你通过测量你的恢复时间，有效地管理你体重的增加。你的举重能力提升到一个较高水平的标志就是你从大的举重量中恢复的时间，以及你举重量的增加。初学者进步非常快，这是因为他们举起的重量远没有达到上限，但是中高级的举重者增加肌肉强度会非常难，他们已经非常接近其能力上限了，所以他们要花费更长的时间来恢复和增加举重重量[⊖]。

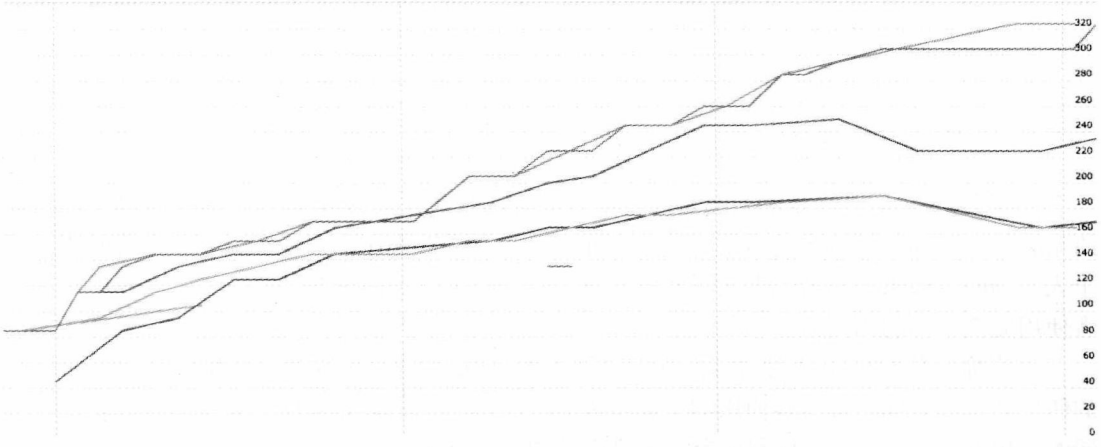


图 1-4 我的举重日志时间序列图

在家的时候，我还会用刻度盘，随时观测家中每个房间的湿度水平，看看室内热量增加会对湿度产生什么样的影响，或者我会将地板和墙壁之间的空隙填塞住之后，观察室内湿度的变化情况，甚至我会只是敞开大门，而将所有房间的门都关上，在这种条件下，观测湿度的变化情况。这样，我就可以了解到没有使用除湿机时，房间内可以自然获得的最低的湿度水平值。

对数据进行可视化可以让我们在一个大的范围内观察数据的变化情况，可以很清晰地看到所有的尖峰、异常，以及在原始数据中不很明显的变化趋势。

1.6 本书的目的

关于当前性能最佳实践方面的内容是非常丰富的，不管在网上还是其他书籍中，都可以找到很多——但是，性能是一个不断变化的主题。由于每个浏览器使用不同的 JavaScript 解释器和渲染引擎，因此在不同浏览器以及不同浏览器版本上你所获得的结果也会不同。由于解释器级别的变化和优化策略不同，系统配置不同，网速也不同，最佳实践是在不断变化或者不断重新定义的。由于大多数浏览器都会快速更新发布版本，导致了这种变化的步伐加速。

与最佳实践同等重要的是测量性能的能力，有了这样的能力，你就可以随变化而调整，

⊖ 关于这些内容可以查看 Mark Rippetoe 写的《Starting Strength》这本书（Aasgard 出版社出版）。

以便在自己的代码中注意到细微的差别，通过你自己的观察定义你自己的最佳实践。

本书旨在提供可以随着时间的推移从多个角度观察和跟踪 Web 应用程序性能的工具，让你总是能够了解到性能的各个方面。我所说的工具，指的不是书中将要开发的代码，也不是将要讨论的应用程序，甚至也不是自动化的程序。我指的是关心这些测量方法的意识，以及构建这些工具并将所有工作可视化的思维模式。

在很多情况下，分析、优化操作和执行的效率是达到下一个卓越水平的一部分。随便一个熟练工人都可以按照技术规格创造出一些东西，但只有大师级的人物才能创造出杰出的作品，同时还能用实验数据证明其作品的杰出之处。

1.7 使用的技术以及拓展阅读

正如本书英文书名所提到的，本书会广泛地使用 JavaScript。我们也会使用 PHP 创建特定的自动化工具，筛取结果并格式化数据。如果你对 PHP 还不熟悉，也没有关系，它的语法及词汇与 JavaScript 非常相似，因此在这两种语言间切换不会有什么问题。关于 PHP 更加广泛的内容超出了本书的范围。如果你想了解更多关于这门语言的介绍，你可以买 W. Jason Gilmore 写的《 Beginning PHP and MySQL 》(Apress 出版社 2005 年出版）一书，或者如果你想对现代 PHP 语言有更加深入的了解，建议购买《 Pro PHP Programming 》一书，这本书由 Peter MacIntyre、Brain Danchilla 以及 Mladen Gogala 合著（Apress 出版社，2011 年）。

我们会经常涉及的另一种语言是 R，一种开发语言与运行环境的结合体，它用来运行统计计算以及对导入和导出的数据进行图表化加工。这是一种非常有趣的语言，有其特殊的用途。

如果你不熟悉 R 语言的语法，或者不了解它不同的数据类型等这些基本概念的话，那么初次接触它会让你感到望而生畏。不过不用担心，书中你需要理解的所有代码，我都会给出解释。如果你想要更深入地了解 R——大多数顶级公司的统计信息是由 R 生成的[⊖]，在未来几年，数据科学是最大的增长领域之一[⊜]，因此你为什么不多了解一下 R 呢？我建议你阅读《 R in Action 》，由 Robert Kabicoff 编著（Manning 出版社，2011 年），以及《 The Art of R Programming ： A Tour of Statistical Design 》，由 Norman matl off 编写（No Starch 出版社，2011 年）。两本书都注重将 R 作为一种程序设计语言，而不是一个数学工具，这对于开发人员而言更容易掌握。

学习 R 非常有用，你用得越多，就越能发现它的用处。而且它是完全可扩展的，有非常丰富的插件架构，还有一个庞大的构建插件的社区。几乎没有 R 不能做的事情——至少在统计和数据可视化领域是如此。

正如前边提到的，有很多的可用资源可以帮助我们进一步阅读和探索整体 Web 性能优化。我已经引用了 Steve Souder 的工作，他是 Web 性能领域的知名专家。他的个人网站是

⊖ http://www.revolutionanalytics.com/what-is-open-source-r/companies-using-r.php 和 http://www.nytimes.com/2009/01/07/technology/business-computing/07program.html

⊜ http://mashable.com/2012/01/13/career-of-the-future-data-scientist-infographic/

http://www.stevesouders.com/，而且他也出版了两本书，对 Web 性能的很多方面进行了深入探讨。他还负责维护 http://httparchive.org/ 网站，其目的就是对 Web 性能指标和统计数据进行归档。各种好玩的事情都将在这里显示，既有网页 jQuery 的使用比例，也有一段时间内 Flash 运用的总趋势。当开发新特性或新应用时，了解整体趋势并进行竞争力分析是非常有用的。

万维网联盟（World Wide Web Consortium，W3C）有一个致力于 Web 性能的工作组。该工作组编制规范，并对现行标准进行扩展，为开发者在本地浏览器中对性能跟踪进行更多的控制提供公开的功能。其章程可在 http://www.w3.org/2010/webperf/ 上找到。第 5 章会讨论工作组的工作进展以及他们编制规范的内容。

本书的内容不仅仅是关于性能的，还涉及可视化信息，因此我推荐 Nathan Yan 编著的可视化方面的图书：《Visualize This：The FlowingData Guide to Design, Visualization, and Statistics》（Wiley 出版社，2011 年），本书是数据可视化最棒的入门读物，也是一件艺术品。Nathan 也维护网站 http://flowingdata.com/。

1.8 小结

本章探讨了一些有关于性能的入门概念。我们对 Web 应用程序性能的两个方面进行了定义。Web 性能是指服务器上的内容到达最终用户所花费时间，运行时性能则表示当最终用户使用应用程序时，应用程序如何做出响应。

我们简要探讨了 Web 所涉及的一些协议，比如 TCP/IP 模型。着眼 TCP/IP 模型，我们跟踪了来自于浏览器的请求内容，以及相应的伴随模型所发生的各种行为。我们探究了一个 TCP 周期的架构，观察了我们的浏览器为获取所需要的每一项内容而需要执行的步骤——有时候，在 HTTP 重定向的情况下，每项内容的获取都是翻倍的。

我们着眼于现代浏览器架构时发现，浏览器不再是黑盒子般的巨石，取而代之的是模块化的，甚至是开放源码的。我们讨论了模块化架构的好处，指出了当网络变得无处不在时，其他应用程序也在使用渲染引擎来进行解析，比如邮件服务器客户端使用渲染引擎渲染标记，或者渲染引擎可以嵌入到自定义的应用程序中，我们甚至可以将浏览器的部分内容嵌入到应用程序中。

从客户满意度到观察放弃率，我们研究了性能与业务息息相关的原因。

最后，我们开始讨论数据的收集、分析以及可视化。本书中将会反复出现的一个要点是：测量和量化实验数据，对这些数据进行可视化，呈现出数据的整体特点。数据的特点是关键，它可以揭示在原始数据中并不明显的趋势以及模式。我们能够快速地查看一个可视化图形，并知道它的主要内容。

后面的章节会更加深入地探讨这些概念，下一章开始探究用于跟踪并提升性能的一些工具。

第 2 章

测量和影响性能的工具与技术

第 1 章概要描述了 Web 性能和运行时性能，并讨论了相关的影响因素。本章将着眼于一些可用来跟踪性能，并有助于提高性能的工具。

后面的章节还要探讨如何以编程方式使用这些工具，以及如何将它们组合起来创建可生成图表和报告的应用程序，因此首先要熟悉这些工具是很有必要的。还有其他一些工具，比如 Firebug 和 YSlow，它们对开发和维护高性能网站是必不可少的。

2.1 Firebug

2006 年对于 Web 开发来说是非常重要的一年。首先，微软在这一年发布了 IE7。IE7 支持本地创建 XMLHttpRequest 对象，而早期的 Web 开发则需要在代码中编写分支逻辑进行判断。如果某个浏览器的 JavaScript 引擎支持 XHR 对象，我们就能直接使用；否则，我们就知道我们使用的是早期版本的 IE，并且需要实例化 XHR 的 ActiveX 控件。

在 2006 年，很多新的框架也应运而生，包括 jQuery、MooTools 以及 YUI，所有这些框架都旨在加速和简化开发过程。

也许 Joe Hewitt 和他的团队在 Mozilla 上发布 Firebug 算是那一年最具有里程碑意义的事件。Firebug 是一款浏览器内嵌工具，它使 Web 开发人员能够完成很多以前根本不可能完成的任务。现在，我们可以通过使用控制台命令行来调用函数或者运行代码，在联机状态下修改 CSS，也可以监控那些形成一个网页所需要下载的网络资源[⊖]。这些都是我们在谈论性能时最感兴趣的方面。如果现在你还没在机器上运行过 Firebug，那么请按照如下步骤来安装 Firebug。

2.1.1 安装

首先安装 Firebug。你可以从下面这个地址下载最新版本的 Firebug ： https://getfirebug.

⊖ 原文是 network asset，直译为“网络资产”，这里实际的含义是加载页面所需要的网络资源。——译者注

com/downloads/。Firebug 的最初版本是作为 Firefox 的扩展功能发布的，从那以后又发布了很多精简版以支持大多数其他浏览器。由于 Firebug 精简版不包含“网络监控”选项卡，因此为了拥有所有可用的功能，这一节使用 Firefox 浏览器来安装 Firebug。

访问上述地址，页面会为你提供不同版本的 Firebug 下载[⊖]，如图 2-1 所示。

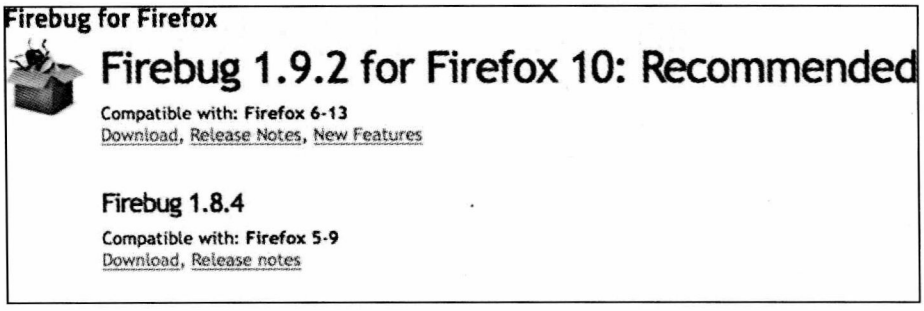


图 2-1 Firebug 下载页面截图

选择你要下载的版本，来到下载页面（见图 2-2）。单击“Add to Firefox”（添加进 Firefox），Firebug 就会下载并自行安装了。然后重启浏览器完成安装。

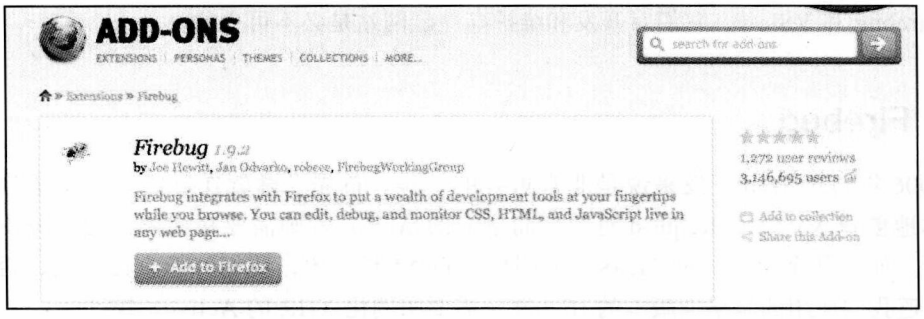


图 2-2 点击 Add to Firefox 按钮安装插件

Firebug 安装完成之后，可以通过单击浏览器右上角的 Firebug 图表打开 Firebug 控制台，也可以单击 File（文件）菜单下的 Web Developer（Web 开发者）→ Firebug 打开 Firebug 控制台，如图 2-3 所示。

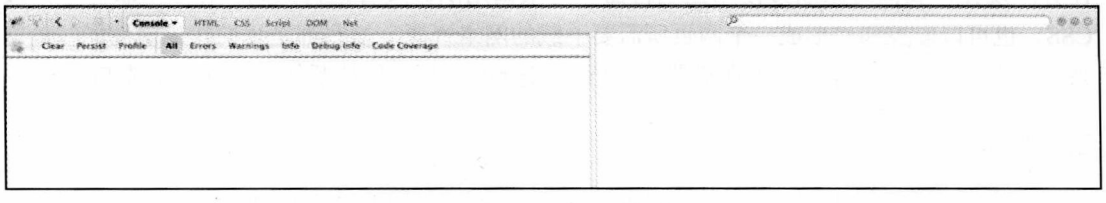


图 2-3 Firebug 控制台

⊖ 本书出版后，网站页面可能有所变化，操作时请以实际界面为准。——编者注

Firebug 控制台美观、易用。在这里，你可以查看加入到代码中的调试消息，查看错误消息，输出对象以查看它们的结构和值，调用页面作用域的函数，甚至可以运行特定的 JavaScript 代码。Firebug 就如同是一座分水岭，它能够使你在浏览器上做如此多的事情，如果之前你没有做过 Web 开发，那么你是不会体会到这一点的。回到正题，如果你习惯于开发语言的集成开发环境（IDE），你肯定习惯使用集成开发环境对内存进行分析，习惯使用集成开发环境调试运行时代码，查看内部变量值，跟踪程序逻辑，那么，你肯定对 Web 开发缺乏相应的工具而感到惊讶。

虽然 Firebug 控制台如此的美观易用，但现在我们要关注的是 Net 标签。

2.1.2 使用

Firbug 中的网络监控是一个被动的监控工具；只需要点击 Net 选项卡——Network Monitoring（网络监控）的缩写，如果你是第一次单击这个选项卡，那么你还需要激活面板，然后浏览一个 Web 页面（就以我的 tom-barker.com 为例）。这个页面加载时，你会看到所有的网络资源开始加载进来。这里给出一个瀑布图的示例（见图 2-4）。

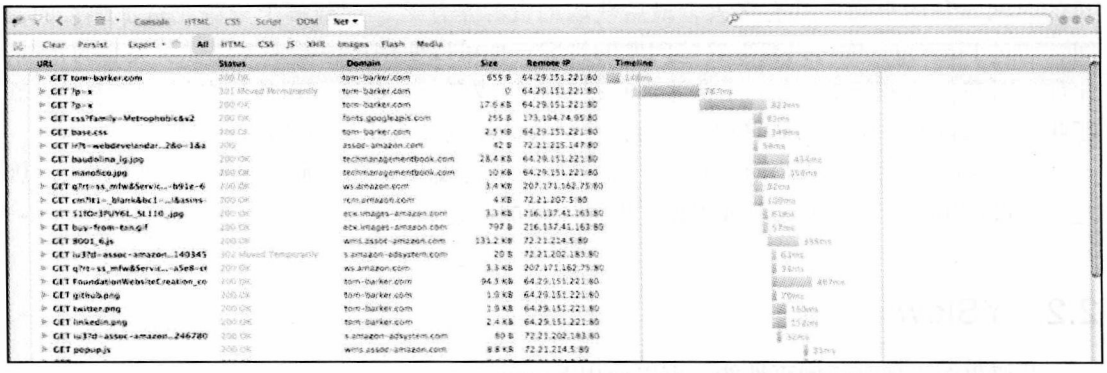


图 2-4 Network Monitoring（网络监控）选项卡下的瀑布图

如第 1 章所述，瀑布图是一个数据可视化工具，用于演示系统中元素顺序地添加和删除的效果。瀑布图也用于 Web 性能监测，以描绘资源是如何加载的以及网页元素是如何对网页加载时间产生影响的。

瀑布图表中的每一个条柱，代表了一项从远程服务器上加载过来的网页内容，包括图片、JavaScript 文件、SWF 或网页字体。条柱按行堆积排列，自上而下依次表示从最初到最后下载的元素。瀑布图表示了在网页加载过程中元素的下载顺序——图片 A 在图片 B 之前下载，而外部 JS 文件最后下载，等等。此外，瀑布图还显示了每项元素下载所消耗的时间。除了条柱之外，图中的每一行都有几列内容，指出了每项元素对应的地址（URL）、HTTP 状态、数据来源域、文件大小和远程服务器 IP 地址。蓝色的竖线表示文档解析完成时间，红色的竖线表示文档加载完成时间。彩色编码的竖条柱表示在给定时间内连接的特定资源，其中的蓝色部分表示 DNS 查询，黄色部分表示正在连接，红色表示正在发送，紫色表示正在

等待数据，绿色表示正在接收数据。

在 Net 标签下边是一个子导航栏，用于过滤瀑布图显示的内容。可以选择显示所有内容，或只显示 HTML 内容、JavaScript、Ajax 请求（调用 XHR 获取 XML Http 请求对象图片、Flash 内容、媒体文件。图 2-5 是我通过过滤，只显示 Javascipt 内容的结果。

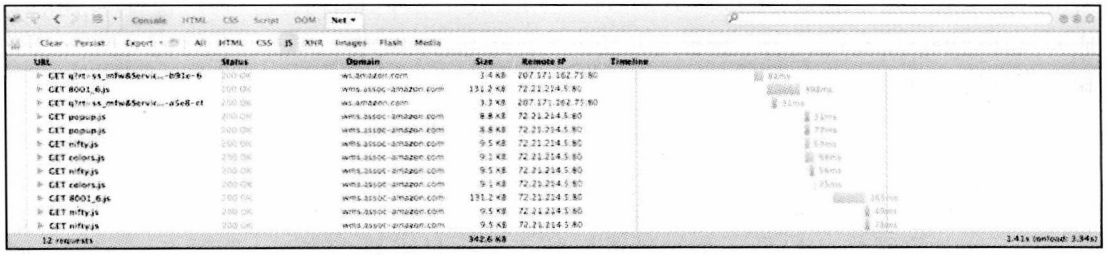


图 2-5 资源类型过滤结果

通常情况下，在开发或做产品支持时，可以使用 Firebug 来了解潜在的问题。你可以主动监测载荷的大小和通常的加载时间，你可以使用 Firebug 来确认你的页面加载不会需要太长的时间。我的页面一共有多大，最后获取的资源是什么，加载哪个资源花费了最长的时间，通过 Firebug，你可以获取这些问题的答案。你可以使用过滤器来关注你所关心的领域，比如看看我们的外部 JavaScript 文件到底有多大。或者，你甚至可以按域名排序的方式查看哪些内容来自那些域，或按 HTTP 状态排序的方式来快速找出导致错误的调用。

由于 Firebug 是一个被动工具，它仅仅用来报告发生了什么而不能给出改进建议，因此，它最适用于作为开发工具或者用于调试出现的问题。

2.2 YSlow

若要更深入地分析网页性能，可以使用 YSlow。

YSlow 是 Steve Souders 及其在雅虎的团队开发的，发布于 2007 年。最初是作为 Firefox 的扩展功能发布的，但是最终还是被移植到大多数其他浏览器上。与 Firebug 类似，YSlow 是一个浏览器内嵌工具，同样，它也没有多少自动化功能，但它却是一款极其优异的工具，它可以用来评估网页 Web 性能，并就改善性能所需采取的步骤提供反馈意见。

提供改进步骤是 YSlow 最与众不同的地方。它使用一系列标准来评估给定页面的性能，并针对你的网站给出特定的反馈意见。最重要的是，这些标准不是一成不变的，当最佳实践改变时或者旧标准变得无关紧要时，YSlow 团队会更新这些标准。

我们来实际操作一下 YSlow。

2.2.1 安装

访问 http://yslow.org/ 并选择想要运行 YSlow 的平台就可以安装 YSlow 了。图 2-6 显示了 YSlow 网站上目前可用的所有不同的浏览器和平台。

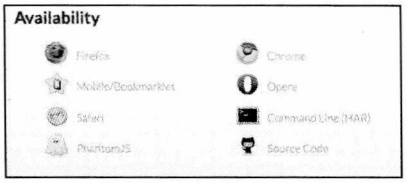


图 2-6 访问 YSlow 的不同方式

既然在安装 Firebug 时我们已经使用了 Firefox，那么我们可以继续使用该浏览器来作为安装 YSlow 的载体。选择 Firefox 版本之后，安装 YSlow 扩展组件然后重启浏览器，就可以使用 YSlow 了。

2.2.2 使用

在 Firefox 上，如果打开 Firebug，就会看到有一个 YSlow 的新标签。当单击这个标签时，如图 2-7 所示的启动画面就会呈现出来。在这个界面里，可以就当前加载进浏览器的页面运行 YSlow 测试，或者选择每当加载进一个新页面就运行 YSlow 测试。

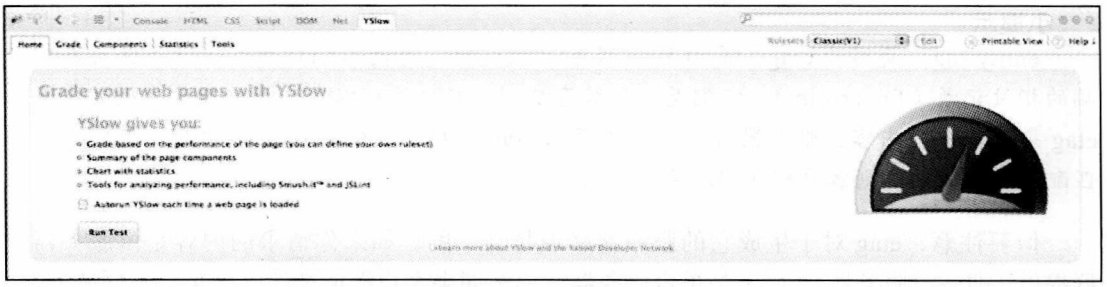


图 2-7 YSlow 扩展组件

也可以选择设定规则来对页面进行评估，如前所述，最佳实践改变后，不同的规则集就会涌现出来以反映这种变化。YSlow 最初推出了一些经典的规则集，而后第 2 版又进行了更新，改变了一些特定规则集的权重（比如将 CSS 和 JavaScript 作为外部文件来定义权重），并增加了一些新的规则，新增了一个用于小规模网站和博客的规则子集，对小规模网站和博客来说原来那些规则显得有些多余。

运行测试之后，你会看到如图 2-8 所示的结果。结果视图分为两部分：左边是规则及其各自的评级情况，右边是对规则的解释。

想要充分了解 YSlow 使用规则的细节，可以访问 http://developer.yahoo.com/performance/rules.html。

结果视图上还有一个二级导航条，使用该导航条可以对测试结果进行进一步的了解，包括显示页面组件，显示页面的统计结果，以及显示用来进一步改进性能的工具。

显示组件部分跟 Firebug 的网络监控标签很像，在这里列举了页面所有的资源，以及每个组件文件的大小、URL 地址、响应头（response header）、响应时间、过期时间头（expires

header）以及 etag[⊖]。

图 2-8 YSlow 结果视图

小提示

entity tag，简写为 etag，是由 Web 服务器生成的，通过 HTTP 事务传递并保存在客户端的指纹记录（fingerprint）。它们是一种缓存机制，客户端可以在会话中通过发送其保存的 etag 来请求一项内容，服务器可以用其保存的 etag 与接收到的 etag 进行比较来看它们是否匹配。如果匹配，则客户端使用缓存中的内容[⊜]。

但要注意，etag 对于生成它的服务器来说是唯一的。如果你请求的内容是由服务器集群提供（即一组服务器），而不是单台服务器，那么如果客户端请求的内容来自不同服务器，则 etag 不会匹配，因此你也不会享受缓存所带来的好处。

如图 2-9 所示，Statistics（数据统计）部分有两个饼图显示了页面上组件的详细信息。左边的图显示了没有内容缓存情况下的结果，右边的图则是带有后续缓存的视图。这对于识别能够得到最大改进的区域是非常有用的。

比较图 2-9 的两个饼图，可以看到，JavaScript 和图像在缓存之前是网页中数据最大的两部分。缓存机制则减少了图像的数据量，但是我打赌，通过使用我稍后讨论的 Minify 工具，JavaScript 内存与用量会减少更多。

还有一些其他的类似于 YSlow 的产品。Google 曾经开发了 Page Speed，可以在一个网站找到它，地址是：https://developers.google.com/speed/pagespeed/insights。Chrome 和 Firefox 也有一些用于 Page Speed 的扩展组件，在这里可以找到它们：https://developers.google.com/

⊖ etag（entity tag 的简写），HTTP 协议规格说明定义 etag 为“被请求变量的实体值”。——译者注

⊜ 该段内容实际上是表示，如果服务器端与客户端发来的 etag 匹配一致的话，服务器就不会发送客户端请求的内容，客户端使用其缓存中的页面作为服务器端的响应，这样可有效减少网络流量，改进 Web 性能。——译者注

speed/pagespeed/insights_extensions。

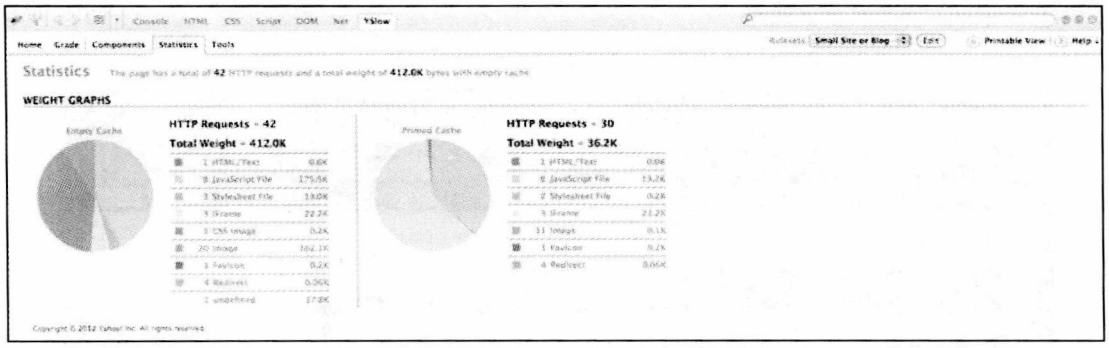


图 2-9 YSlow 统计结果视图

YSlow 和 Page Speed 之间的差别是很细微的，主要看风格和显示的个人偏好。图 2-10 显示了在 Chrome 上运行 developer 工具的 Page Speed 测试结果。

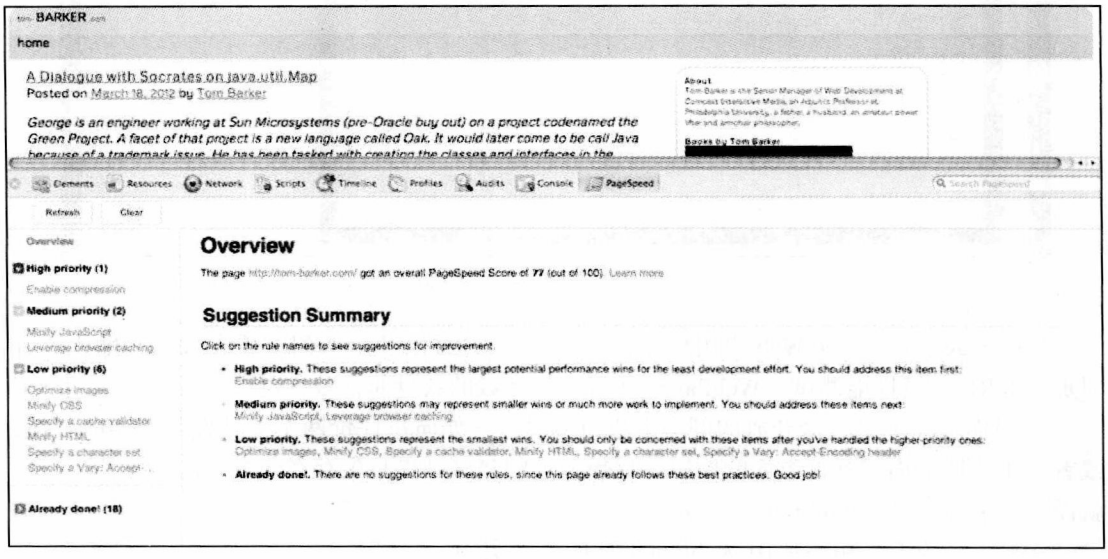


图 2-10 Page Speed 结果视图

另一个具有相似功能的产品是 WebPagetest。由于其丰富的特性设置，以及可自动化的潜力，WebPagetest 将是下一款要详细讨论的产品。

2.3 WebPagetest

AOL 于 2008 年创建了 WebPagetest 的第一版本，当时考虑到公众消费能力以及为了推动这款软件发展，WebPagetest 是作为开源软件发布的。作为一个开源项目，WebPagetest

可用在公开的网站，也可供用户下载来运行自己的实例。它的源代码库可以在 http://code.google.com/p/webpagetest/ 找到。其公开的网站地址是 http://www.webpagetest.org/，如图 2-11 所示。该网站由 Pat Meenan 通过他的 WebPagetest 责任有限公司运营维护。

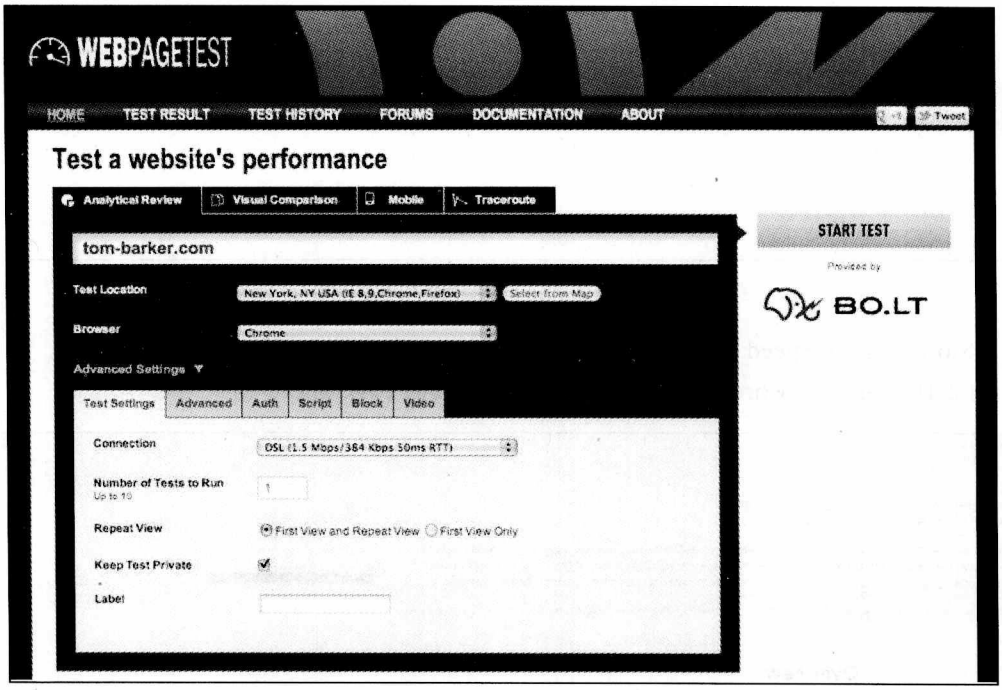


图 2-11 Webpagetest.org

WebPagetest 是一款 Web 应用程序，它将一个 URL 以及一系列配置参数作为输入，并对那个 URL 运行性能测试。WebPagetest 可配置参数的数量非常多，范围非常广。

如果你想在不对公众开放的网站上进行测试——如质量评价网站或开发环境类的网站，或者由于法律上面或其他方面的原因，你想要你的测试结果只保存在你自己的服务器上，你就需要安装你自己的 WebPagetest 实例。

否则，就没有理由不使用 WebPagetest 的公开实例。

你可以选择世界上任何一组网站地址来进行测试。每个地址都可以用一个或多个浏览器对那个地址进行测试。你也可以指定连接速度以及运行测试的数量。

在 Advanced（高级）面板中，可以设置当文档完成时停止测试运行，这可以通知我们 ducument.onload 事件是何时触发的，而不是当所有页面资源都加载完成之后才通知我们。这一点是非常有用的，因为在页面加载完成才发生的 XHR 通信会作为一个新的活动注册，从而使测试结果产生偏差。

你也可以让测试忽略 SSL 认证错误，否则测试可能无法正常运行，因为在这种情况下，最终用户需要出示证书后会话才能继续进行，否则会话将终止。

Advanced（高级）选项卡还提供了一些其他的选项，通过这些选项，可以在测试中抓取追踪到的数据包并生成网络日志，用来提供运行测试的网络事务中的详细信息；也可以选择 Preserve Original User Agent String（保留原有用户代理字符串）选项，以保留用户原有的运行测试的浏览器的代理字符串，而不是添加一个字符串来将该访问识别为一个 WebPagetest 测试。

如果 Web 网站使用 HTTP 认证才能访问，可以在 Auth（认证）选项卡中指定证书，要记住小心谨慎。不推荐使用真实的产品用户名和密码来进行测试以及保存测试结果。推荐的做法是创建一个仅用于测试的有权限限制的测试证书。

有时候，你需要测试一些非常特殊的情况。也许你正在对某个特定功能集进行多元测试，该功能集只为特定的客户端配置提供特定的特性，比如 iPhone 的特殊功能。或者，你正在对一组具有某些使用习惯的用户的一些特性进行测试。你可能希望仅当特殊事件触发时才对这些特性进行性能测试。

Script（脚本）选项卡提供的功能可以实现上述目的。你可以运行一个复杂的测试，该测试涉及多个步骤，包括访问多个 URL 地址，发送 Click 和 Key 事件给 DOM，提交表单数据，执行特殊的 JavaScript，还要更新 DOM。你甚至可以通过修改 HTTP 请求设置以实现诸如设置特定 cookie、主机 IP 设置以及用户代理变更等事情。

比如，为了让客户端呈现出 iPhone 样式的页面，只需要添加如下的脚本：

```
setUserAgent      Mozilla/5.0 (iPhone; U; CPU iPhone OS 4_0 like Mac OS X; en-us)
AppleWebKit/532.9 (KHTML, like Gecko) Version/4.0.5 Mobile/8A293 Safari/6531.22.7
navigate http://tom-barker.com
```

setUserAgent 命令伪造客户端用户代理，navigate 命令将该测试指向特定 URL。想要阅读更多有关 WebPagetest 脚本的编写语法以及一些强大的功能，可以访问：https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/scripting。

Block（分块）选项卡允许我们对请求的内容进行分块。这对于我们比较有没有广告，有没有 JavaScript，以及有没有图片文件的测试结果是非常有用的。除了使用 Block 选项卡外，我们也可以在 Script 选项卡中编写包含分块命令的脚本完成此功能。如果我们想要通过编写脚本将一个网站的所有 PNG 文件分块，我们可以这样做：

```
block .png
navigate http://www.tom-barker.com
```

最后，Video（视频）选项卡可以让我们抓取页面加载时的视频截图，观看这些截图就跟看一段视频一样。这对于观察页面的加载情况是非常有用的，特别是在当有些内容是异步加载的情况下，你可以看到页面加载过程中的哪个时间点上页面看上去是可用的。

一旦配置好所有的选项，就可以运行测试了。测试结果如图 2-12 所示。

首先，摘要视图将所有重要的相关信息进行了整合。在右上角是页面的 Page Speed（页面速率）结果摘要。这是信息的一个高级表达形式，如果在 Page Speed 中运行测试，同样的信息也会被显示，只不过它使用了 YSlow 的字母分级格式。

在瀑布图和屏幕截图上方的表格中是页面级别指标（page level metrics），包括整页加

载时间值，第一字节加载所需时间，第一项内容加载所需时间，页面中 DOM 元素的数量，document.onload 事件触发的时间，页面所有元素加载花费的时间，以及为了完成页面绘制所需的 HTTP 请求次数。

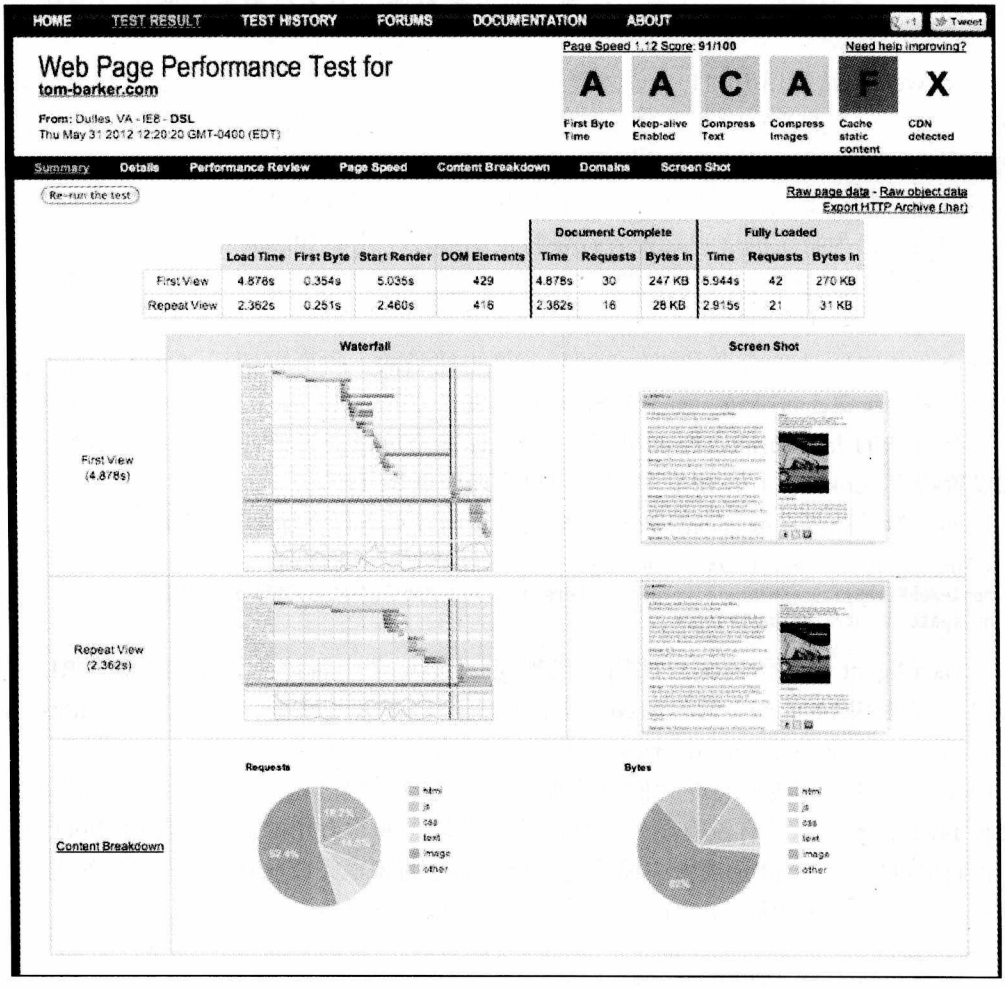


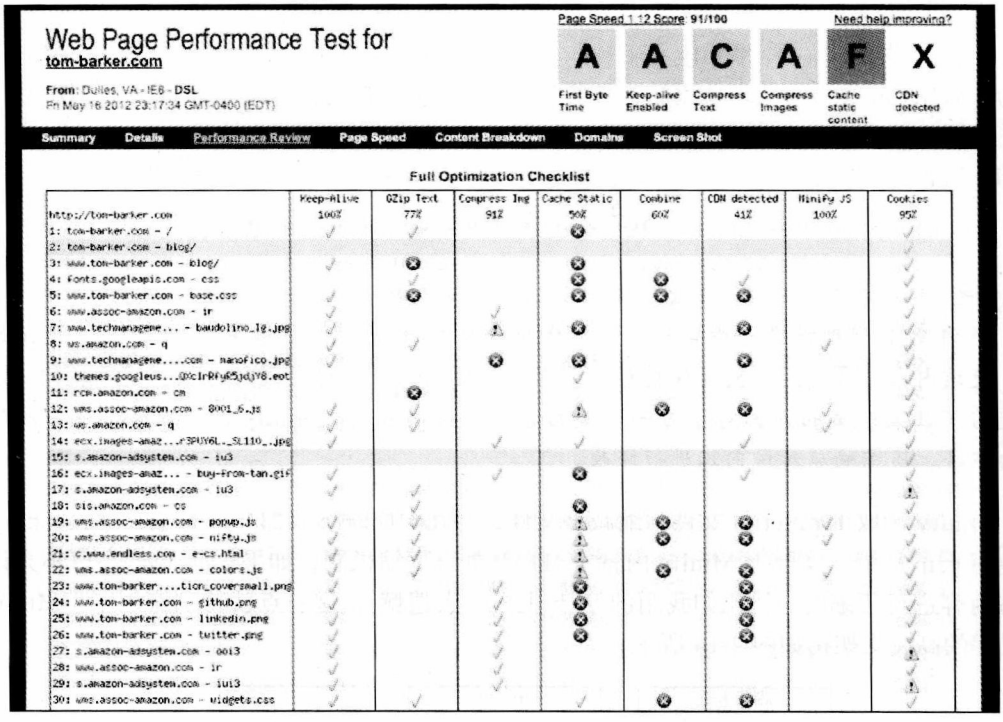
图 2-12 Webpage 测试结果页面

记下这些数据。这些数据包含了构成定量指标的基本信息，我们会在第 3 章图表化 Web 性能中用到这些定量指标。它们是网站 Web 性能真正的本质所在。

下边的表格有两列，左边的一列是页面首次加载视图和缓存重复加载视图的瀑布图，右边的一列是对应的视频截图。前边已经对瀑布图的作用进行了详细的讨论。

再往下是两个饼状图。左边的饼状图显示了不同内容类型的请求百分比。右边的则显示了不同内容类型的字节百分比。这对于识别可优化的最大区域是非常有用的。如果你的 JavaScript 只占你总负载的 5%，而图片占了 70%，那么首先优化图片内容，这会为你提供更好的服务。

摘要页面提供了所有数据的高度概括，通过点击摘要页面的二级导航条就能看到包含这些数据的页面。点击二级导航条上的 Details（详情）、Performance Review（性能查看）、Page Speed（页面速度）、Content Breakdown（内容分解）、Domains（域）、Screen Shot（视频截图）等链接，就可以更深入地了解相关内容了。Content Breakdown（内容分解）部分如图 2-13 所示。图中显示了每一项内容在各项标准下的表现情况，这些标准包括 Keep Alire(保持有效)、Gzip Text(Gzip 文本)、Compress Images(图片压缩)、Cache Static(缓存统计)、Combine(组合)、CDN detected（CDN 检测）、Minify（缩减）JS 和 Cookies。绿色复选标记表示合格，黄色的带有叹号的三角形表示警告，红 X 表示错误。



	Keep-Alive	GZip Text	Compress Img	Cache Static	Combine	CDN detected	Minify JS	Cookies
http://tom-barker.com	100%	77%	91%	56%	60%	41%	100%	95%
1: tom-barker.com - /	✓	✓		✗				✓
2: tom-barker.com - blog/								
3: www.tom-barker.com - blog/	✓	✗		✗				✓
4: fonts.googleapis.com - css		✓		✗	✗	✓		✓
5: www.tom-barker.com - base.css	✓	✗		✗	✗	✗		✓
6: www.assoc-amazon.com - ir	✓		✓					✓
7: www.technomanageme... - baudolino_lg.jpg	✓		!	✗		✗		✓
8: ws.amazon.com - q	✓	✓					✓	✓
9: www.technomanageme....com - nanofico.jpg	✓		✗	✗		✗		✓
10: themes.googleus...QXclrRFy65jdjY8.eot				✓				✓
11: rcm.amazon.com - cm		✗						✓
12: ws.assoc-amazon.com - 8001_6.js	✓	✓		!	✗	✗		✓
13: wp.amazon.com - q	✓	✓					✓	✓
14: ecx.images-amaz...rSPUY6L._SL110_.jpg	✓		✓	✓		✓		✓
15: s.amazon-adsystem.com - iu3								
16: ecx.images-amaz... - buy-from-tan.gif	✓		✓	✗		✓		✓
17: s.amazon-adsystem.com - iu3	✓	✓						!
18: bis.amazon.com - os		✓		✗				✓
19: ws.assoc-amazon.com - popup.js	✓	✓		!	✗	✗		✓
20: ws.assoc-amazon.com - nifty.js	✓	✓		!	✗	✗	✓	✓
21: c.www.endless.com - e-cs.html	✓	✓		!				✓
22: ws.assoc-amazon.com - colors.js	✓	✓		!	✗	✗	✓	✓
23: www.tom-barker....tion_coversmall.png	✓		✓	✗		✗		✓
24: www.tom-barker.com - github.png	✓		✓	✗		✗		✓
25: www.tom-barker.com - linkedin.png	✓		✓	✗		✗		✓
26: www.tom-barker.com - twitter.png	✓		✓	✗		✗		✓
27: s.amazon-adsystem.com - ad13	✓		✓					!
28: www.assoc-amazon.com - ir	✓		✓					✓
29: s.amazon-adsystem.com - iu13	✓		✓					!
30: ws.assoc-amazon.com - widgets.css	✓	✓		✓	✗	✗		✓

图 2-13 Webpagetest 性能优化清单

正如你所看到的，WebPagetest 对网站 Web 性能提供了非常有帮助的信息，最为突出的是，WebPagetest 是完全可以用来编写程序的。它提供了 API（应用程序接口），可以在程序中调用这些接口以获取这些信息。第 3 章将会对这些 API 进行研究，以构建我们自己的应用程序来跟踪和汇报 Web 性能。

2.4 缩减

通常，优化人员都会花费大量的经历来考虑如何优化缓存。这是非常重要的，因为缓存尽可能多的内容可以为用户接下来的访问带来更好的体验，而且也可以节省带宽以及减少

对服务器的流量冲击。

但是，当一个用户第一次访问网站时，缓存还有待建立。因此为了确保我们的首次访问尽可能地流畅，我们需要缩减 JavaScript。

缩减的思想最初是基于 JavaScript 解释器对空格、换行符以及注释的忽略，因此如果我们移掉了不需要的字符，我们就可以缩减整个 .js 文件的大小。

有很多产品可以用来缩减 JavaScript。其中一些出类拔萃的产品在这个方面进行了巧妙的改进。

2.4.1 Minify

首先，我们来看看 Minify，在 http://code.google.com/p/minify/ 可以找到它。Minify 是作为 JavaScript 文件的代理使用的，页面中的 script 标签指向 Minify，Minify 是 PHP 程序文件（下面的代码中，我们只是指向了 /min 目录，因为 PHP 程序文件是 inde.php）。scrip 标签如下所示：

```
<script type="text/javascript" src="/min/?f=lib/perfLogger.js"></script>
```

注意

所谓 Web 代理就是一段代码，这段代码接收 URL 地址，读取并处理该 URL 地址的内容，使该内容正常可用，要么保持原样要么用其他的功能或格式进行修饰。通常我们使用代理是为了让某个域的内容对于另一个域的客户端代码可用。Minify 读取内容，通过删除无关的字符、压缩响应对该内容进行修改。

Minify 读取 JavaScript 文件，缩减该文件，并在响应时将 HTTP 头的 accept encoding 置为使用 gzip 压缩。实际上 Minify 内置了 HTTP 静态压缩机制，如果你的 Web 主机不允许对静态内容进行压缩时（就比如我用的 Web 主机，太遗憾），这一点就特别有用了。Minfy 工作机理的高层次架构如图 2-14 所示。

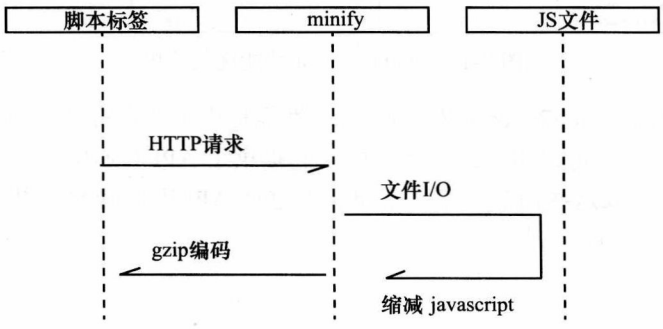


图 2-14 Minify 执行流程图。脚本标签指向 Minify，读入 JavaScript 文件的 URL。Minify 剔除掉不必要的字符，并设置响应头（response header）为 gzip 编码格式，返回结果给 script 标签，并将结果加载到浏览器

要使用 Minify，只需要从 http://code.google.com/p/minify/ 网站上将它下载下来，将其解压缩至 /min 文件夹并放到 Web 网站根目录下，然后到 /min/builder/ 目录下打开 Minify 控制面板。

在控制面板中，可以添加你想要缩减的 JavaScript 文件，页面生成可链接到 JavaScript 文件缩减后的脚本标签，相当简单。

2.4.2 YUI Compressor

另一个用于缩减的工具是 Yahoo 的 YUI Compressor，可以在这里找到它：http://yuilibrary.com/download/yuicompressor/。YUI Compressor 是一个 jar 文件，而且要以命令行的方式运行。因此它可以便捷地整合到某个构建过程中。YUI Compressor 执行方式如下所示：

```
java -jar yuicompressor-[version].jar [options] [file name]
```

跟 Minify 一样，YUI Compressor 从 JavaScript 中剔除掉了所有没用的字符，包括空格、换行符以及注释。想要了解关于 YUI Compressor 的更多细节，可以访问 http://developer.yahoo.com/yui/compressor。

2.4.3 Closure Compiler

最后，我们看看 Google 的 Closure Compiler, 在 https://developers.google.com/closure/compiler/ 可以找到它。Closure Compiler 也可以使用命令行的方式运行，因此也可以整合进某个自动化过程中，它不仅缩减了 JavaScript，而且通过重写进一步压缩了 JavaScript。要重写 JavaScript，Closure Compiler 要执行一些“焦土”(scorched-earth) 优化策略——它将函数展开，重写变量名，删除从不会调用的函数（只要它能判断）。由于其剔除掉所有的东西（包括最佳实践），以寻求最小可能的载荷，因此这种策略被认为是“焦土”优化策略。而且这种方法也是成功的。这种优化后的代码是永远也写不出来的，因此我们要保存好我们最初的代码，然后使用 Closure Compiler 来“编译”代码，以生成可能是最优化的代码。我们将这些“编译”好的代码保存在一个独立的文件中，这样我们就可以用我们的原始代码来更新“编译”好的代码了。

要想了解 Closure Compiler 是如何重写 JavaScript 的，我们可以看看运行 Closure Compiler 之前和之后的代码情况。对于运行 Closure Compiler 之前的代码，我们先使用第 7 章将要用到的一个例子。

```
<script src="/lib/perfLogger.js"></script>
<script>
    function populateArray(len){
        var retArray = new Array(len)
        for(var i = 0; i < len; i++){
            retArray[i] = 1;
        }
        return retArray
    }
```

```
perfLogger.startTimeLogging("page_render", "timing page render", true, true)
/* ***

7.1
Compare timing for loop against for in loop
****/

var stepTest = populateArray(40);

perfLogger.startTimeLogging("for_loop", "timing for loop", true,true, true)
for(var x = 0; x < stepTest.length; x++){
}
perfLogger.stopTimeLogging("for_loop");

perfLogger.startTimeLogging("for_in_loop", "timing for in loop", true, true)
for(ind in stepTest){

}
perfLogger.stopTimeLogging("for_in_loop")

/** end 7.1 ***/

/* ***

7.1.1
Benchmark for loop and for in loop

****/
function useForLoop(){
     var stepTest = populateArray(40);
     for(var x = 0; x < stepTest.length; x++){
     }
}

function useForInLoop(){
     var stepTest = populateArray(40);
     for(ind in stepTest){
     }
}

perfLogger.logBenchmark("f", 1, useForLoop, true, true);
perfLogger.logBenchmark("fi", 1, useForInLoop, true, true);

perfLogger.stopTimeLogging("page_render")
</script>
```

Closure Compiler 获取代码并重写，结果如下所示：

```
<script>
var b=[];function e(a,c){b[a]={};b[a].id=a;b[a].startTime=new Date;b[a].description=c;b[a].a=!0}
function f(a){b[a].d=new Date;b[a].c=b[a].d-b[a].startTime;b[a].url=window.location;b[a].
e=navigator.userAgent;b[a].a&&g(a)}function h(a,c){for(var d=0,j=0;10>j;j++)e(a,"benchmarking
"+c),c(),f(a),d+=b[a].c;b[a].a=drawToPage;b[a].b=d/10;b[a].a&&g(a)}
function g(a){var c=document.getElementById("debug"),d="<p><strong>"+b[a].description+"</strong>
<br/>",d=b[a].b?d+("average run time: "+b[a].b+"ms<br/>"):d+("run time: "+b[a].
c+"ms<br/>"),d=d+("path: "+b[a].url+"<br/>"),d=d+("useragent: "+b[a].e+"<br/>"),a=d+"</p>";c?c.
innerHTML+=a:(c=document.createElement("div"),c.id="debug",c.innerHTML=a,document.body.
```

```
appendChild(c))}function i(){for(var a=Array(4E4),c=0;4E4>c;c++)a[c]=1;return a}e("page_
render","timing page render");var k=i();e("for_loop","timing for loop");
for(var l=0;l<k.length;l++);f("for_loop");e("for_in_loop","timing for in loop");for(ind in
k);f("for_in_loop");h("f",function(){for(var a=i(),c=0;c<a.length;c++);});h("fi",function(){var
a=i();for(ind in a);});f("page_render");
</script>
```

性能改进是非常明显的，但是代价是代码的可读性降低了，比原始代码更抽象了。

2.4.4 结果比较

要想确定特定情况下所使用的最佳工具，我们将采取科学的方法。我们使用刚才讨论过的工具进行多元测试，看看对我们来说哪个工具会给出最佳结果。

首先我们来看一个未缩减的范例代码的瀑布图，如图 2-15 所示。

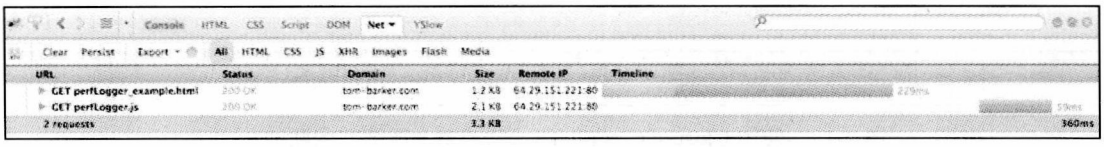


图 2-15 未压缩 JavaScript 的瀑布图（我们的基准）

我们看到未经压缩和缩减的 JavaScript 文件大小为 2.1KB，页面总大小为 3.3KB。这个例子可以在 http://tom-barker.com/lab/perfLogger_example.html 上找到。

现在，我们使用 Minify 看看其测试结果。从图 2-16 的瀑布图上我们可以看到，经 Minify 缩减和编码后的 JavaScript 文件只有 537 字节，总页面大小为 1.9KB。

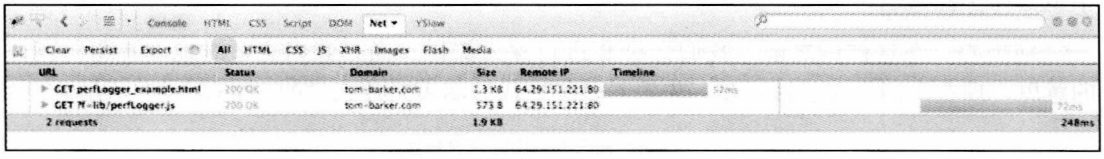


图 2-16 使用 Minify 压缩后的页面

当我们使用 YUI Compressor 和 Closure Compiler（选择最简单的参数选项，文件只会缩减，不会重写）运行于同样的文件上时，我们得到了相同的结果，JavaScript 文件减小到 1.6KB，页面总大小为 2.9KB，如图 2-17 所示。

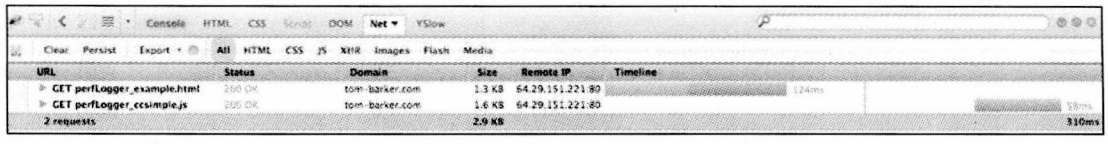


图 2-17 使用 Closure Compiler（简单参数）压缩后的页面

记住，我们使用的 Web 主机不支持全局范围内的 HTTP 压缩，因此这些结果仅仅是缩减的结果，而非压缩的结果。因此我们不是对缩减算法进行逐一比较，而只是比较应用了这

些算法的产品所得出的结果。

最后，我们比较一下将原始 JavaScript 文件采用 Closure Compiler 高级参数选项进行编译的结果，这样 Closure Compiler 会重写代码，使其尽可能流线化。在这样测试的时候，要确保将页面上所有的 JavaScript 都涵盖进来了，也就是说，不仅仅是远程的 JS 文件，还包括写在页面上用于实例化对象的 JavaScript 代码。这么做是很有必要的，因为 Closure Compiler 会将它认为不会执行的所有代码全部删除。因此，如果你在远程 JS 文件中有一个命名空间对象，要在 HTML 中用代码将其实例化，那么你需要在同一个的文件中将实例化对象的代码涵盖进来，这样 Closure Compiler 就能知道这段代码是有用的，并将其包含进它的输出结果中。

Closure Compiler 最终的输出结果会嵌入在 HTML 页面上，而不是链接到外部。你可以在图 2-18 看到结果。

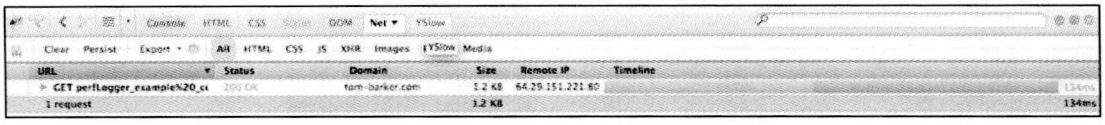


图 2-18 使用 Closure Compiler（高级参数）压缩并内联的页面

既然有数据了，那就开始做可视化和评估的工作吧！

2.4.5 分析与可视化

接下来，要开启 R 语言，并输入缩减结果，使用工具的名称，每个工具的输出结果文件大小，以及每个工具压缩百分比差异。然后使用 R 语言进行编码，创建一个水平条柱图表来比较这种差异。

别担心，在这样做之前，我们会对 R 进行深入探讨，而且我们会适时地解释每行代码的意思。现在我们先越过这一步，先看看最终生成的图表是什么样的，如图 2-19 所示。

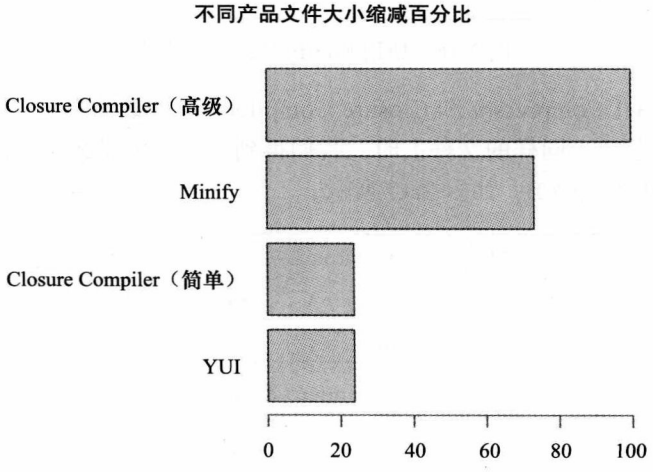


图 2-19 R 语言生成的产品文件缩减比例对比图

由图 2-19 可以看出，Closure Compiler 对文件的压缩是最大的，而结合了缩减和压缩功能的 Minify 对文件的压缩排在第二位。YUI 以及使用简单缩减参数的 Closure Compiler 对文件的压缩结果就差得很远了，排在第三位。

再次重申，这是对软件产品性能的比较——如果我们使用第三方软件对结果进行压缩，那它们就可以和 Minify 产生的结果进行对比了，而 Minify 软件产品也具有压缩功能。

单纯地减小文件大小只是我们进行全面判定的一个方面而已。正如前边看到的一个例子，使用 Closure Compiler 的高级功能后所产生的代码结果与运行它之前的原始代码是完全不一样的。如果产品出现了问题，那将是很难调试的，特别是在页面中有第三方代码与你的代码交互的情况下。

你的网站有第三方代码吗？比如广告类的代码？你是自己架设服务器还是使用了托管 Web 服务器？是产品支持重要还是尽可能拥有绝对最快的体验重要？当选择了你心仪的工具时，最好像上面一样做一下评估，看看哪些工具可以最大限度地适用于我们的形势、环境以及业务准则。比如，你是否已经构建了一个可以整合你心仪工具的环境，并且能够对你的 Web 主机的配置进行控制？如果已经构建，那么 YUI 或 Closure Compiler 可能是你最佳的选择。你能适应 Closure Compiler 高级设置中的“焦土”方法吗？如果能，它将带给你最显著的性能提高——但是如果想要对它的输出进行调试，那就只能祝你好运了。

2.5 R入门

R 是 1993 年由 Ross Ihaka 和 Robert Gentleman 开发出来的。它是 S 语言的扩展和继承，而 S 本身就是一种统计性语言，由贝尔实验室的 John Chambers 创建于 1976 年。

R 既是一个开源环境，也是一种运行于该环境上的语言，用于执行统计计算。这只是一个非常概要的描述。我既不是一个统计学家，也不是一个数据分析员。我是一个 Web 开发人员，而且我管理着一个 Web 开发人员部门，如果你读到这里了，那么可能你也是一个 Web 开发人员。那么作为 Web 开发人员，我们该用 R 做些什么呢？

通常我使用 R 获取、分析、处理数据、然后将结果可视化以进行汇报。图 2-20 描绘了这个工作流。R 并不是我在这个工作中唯一使用的语言，但却是我的新宠。通常，我需要使用其他语言来获取数据源或者剔除其他应用程序。第 3 章使用 PHP（胶水语言）来做到这一点，但是，我们也可以使用其他任何语言——Ruby、shell 脚本、Perl、Python，等等。

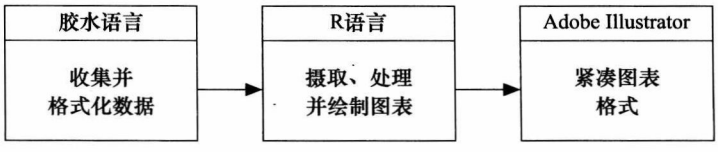


图 2-20 使用 R 进行数据可视化准备的流程图

当我们使用胶水语言收集数据之后，我将数据以逗号分隔的形式写入到文件中，然后用 R 来读取这些数据。在 R 中我处理这些数据，将数据分割、平均、汇总，叠加两个或多个数据集，然后在 R 中将这些数据输出为图表，用图表来表达我在这些数据中所看到的情景。

一旦我在 R 中生成了图表（通常是 PDF 格式的图表文件，以便保存图形向量和字体），我就会将图表从 R 导入到 Adobe Illustrator 或者任何其他的程序中，在这些程序中，我们可以清理一些东西，比如字体一致性，确保坐标轴标签的长名称可见。

在 R 中，可以运行哪一类的数据呢？答案是所有种类的数据都可以。在本书中，我们会在 R 中进行性能数据可视化，在我进行部门指标汇报时（比如缺陷密度、代码库的代码覆盖等问题），我也会用到 R。

作为语言来讲，R 非常小，它是自包含的、可扩展的，而且用起来也很有意思。即便如此，它也有它自己的原则，有它自己的模式，我们来看看其中的一些。

2.5.1 安装并运行 R

要安装 R，需要先从 http://cran.r-project.org/ 上下载一个预编译的 R 二进制文件。对于 Mac 和 PC 来说，这是个标准的安装程序，会引导你完成安装过程。PC 安装程序提供 3 种安装风格：Base（基本安装）、Contrib（含有第三方软件包的安装）、Rtools（提供可构造你自己 R 套件工具的安装）。对我们来讲，我们选择基本安装。图 2-21 是一个 R 安装程序的截图。

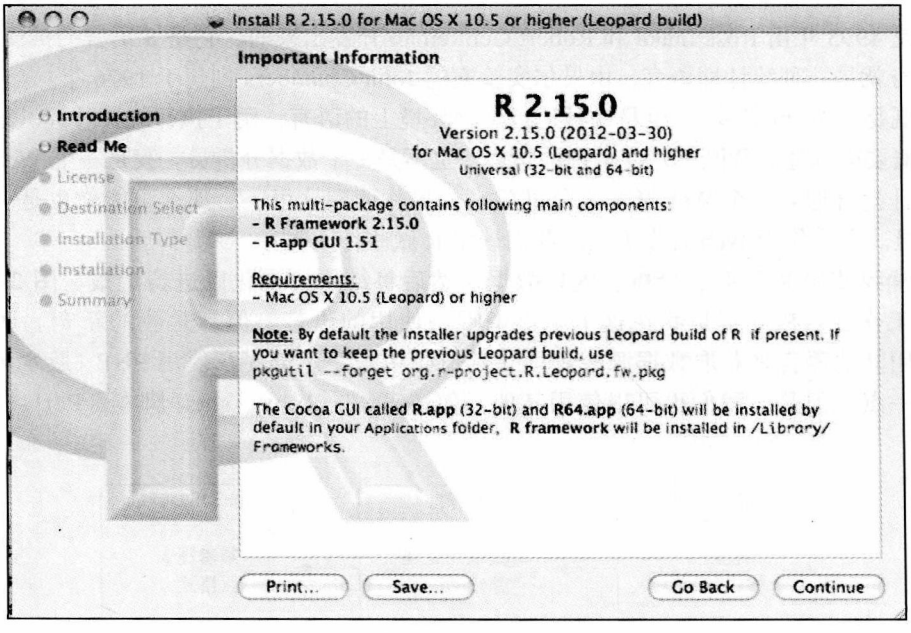


图 2-21 用于 Mac 的 R 安装程序

除了使用编译好的安装程序之外，Linux 用户可以用 linux 风格的命令序列方式来完成安装。

完成 R 安装之后，就可以打开 R 控制台，也就是将要运行 R 语言的环境。控制台如图 2-22 所示。

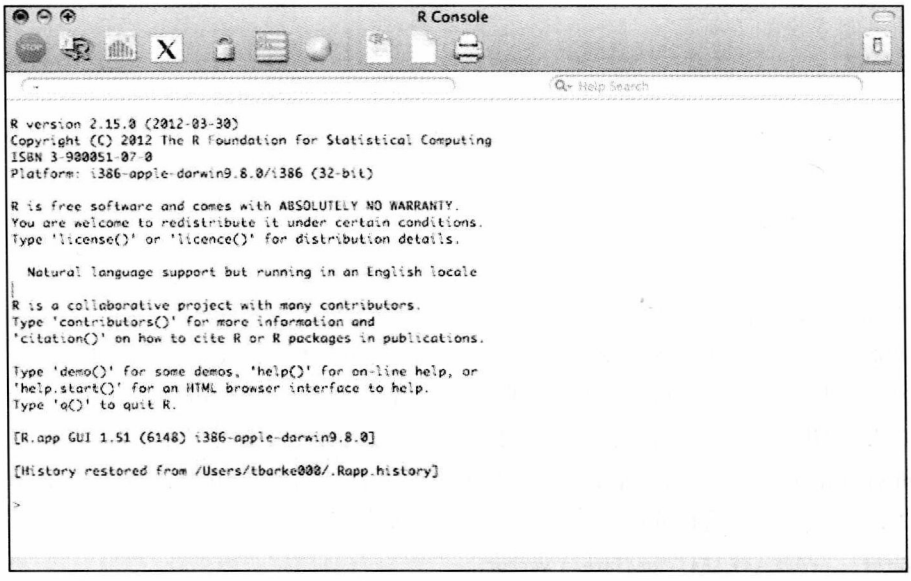


图 2-22 R 控制台

使用控制台的工具栏，我们可以执行一些任务，比如中断 R 进程的执行，运行单机模式的 .R 文件，调整控制台的外观，创建新的单机模式 .R 文件等。

R 控制台是一个专门运行特定 R 命令的命令行环境。一般我用控制台来实践想法，并进行调整，直到产生我想要的东西，然后我把这些运行表达式保存到一个独立的 R 文件中。

你也可以创建外部文件来保存你的 R 代码，一般这样的文件都有一个 .R 扩展名。

R 是高度可扩展的，它有一个强大的社区来构造用于扩展 R 功能的软件包。即使如此，由于本书用到的 R 代码将不会使用任何软件包，因此我们坚持只选择 R 的基本安装方式。

2.5.2 R 基础

既然你知道 R 是什么了，那怎么使用 R 呢？首先要注意的是在任何时候，你只需输入“?[关键字]”就会打开一个特定主题的帮助窗口。如果你不确定你所找的内容有相关的帮助主题，那么只需要输入“??[关键字]”来进行扩展搜索。例如，输入“?hist”，搜索有关创建直方图（histogram）的帮助。

```
> ?hist
starting httpd help server ... done
```

另外需要注意的是，R 支持单行注释，不支持多行注释，这一点也很重要。注释以井字

符开头，R 解释器将忽略所有井字符到换行符之间的内容。

```
#this is a comment
```

1. 变量和数据类型

变量的声明很简单，只需要给它赋个值就可以了。赋值操作符是一个向左的箭头，所以创建和声明变量如下所示：

```
foo <- bar
```

R 是弱类型的编程语言，它支持你能想到的所有标量数据类型：string、number 以及 boolean 等。

```
myString <- "This is a string"
myNumber <- 23
myBool <- TRUE
```

R 也支持列表，但是这也是这门语言很奇怪的一个地方。R 有一种称为 vector 的数据类型，在功能上与严格类型（strictly type）的一维数组非常相似。该数据类型就是一个列表，列表中的每一项都是相同数据类型的数据，或者都是 string，或者都是 number，或者都是 boolean。要声明 vector 变量，需要使用 c() 函数，向 vector 变量中添加内容也要使用 c() 函数。访问 vector 中的元素要使用方括号。与大多数其他语言的数组类型不同，vector 中的元素不是以 0 开始的，引用它的第一个元素用 [1] 这种方式。

```
myVector <- c(12,343,564) #declare a vector
myVector <- c(myVector, 545) # appends the number 545 to myVector
myVector[3] # returns 564
```

R 还有另一种列表类型，叫做 matrix，该类型更像是严格类型的二维数组。创建 matrix 类型变量需要使用 matrix 函数，该函数有 5 个参数：用于保存数据的 vector 类型参数，排列数据所需行数，排列数据所需列数，以及一个可选的布尔型变量参数，该参数指出数据是以行还是以列的形式排列的（默认为 FALSE，即以列的形式排列），最后还有个 list 参数，其中包含了由行名称组成的 vector 以及列名称组成的 vector。

```
matrix([content vector], nrow=[number of rows], ncol=[number of columns], byrow=[how to sort],
dimnames=[vector of row names, vector of column names])
```

也可以使用方括号通过索引来访问 matrix 中的数据，但必须要在方括号中指出行和列。

```
m <- matrix(c(11,12,13,14,15,16,17,18), nrow=4, ncol=2, dimnames=list(c("row1", "row2", "row3",
"row4"), c("col1", "col2")))

> m
     col1 col2
row1   11   15
row2   12   16
row3   13   17
row4   14   18

>m[1,1]  #will return 11
```

```
[1]11

> m[4,2]   #will return 18
[1] 18
```

到目前为止，matrix 和 vector 都只能包含单一的数据类型。R 支持另一种列表 list 类型，叫做数据帧（data frame）。数据帧是一个多维的列表，能包含多种数据类型的数据。

很容易把数据帧想成 vector 集合。vector 只能存放一种数据类型的数据，而数据帧可以存放多种数据类型的 vector。

使用 data.frame() 函数来创建数据帧，该函数以一组 vector 作为输入，后接以下这些参数：rows.names 指定一个作为行标识符的 vector，check.rows 用于验证行数据的一致性，check.names 则用于检验在其他语法检查中是否有重复记录。

```
userid <- c(1,2,3,4,5,6,7,8,9)
    username <- c("user1", "user2", "user3", "user4", "user5", "user6", "user7", "user8",
"user9")
    admin <- c(FALSE, FALSE, TRUE, FALSE, TRUE, FALSE, FALSE, TRUE, TRUE)

users <- data.frame(username, admin, row.names=userid)
> users
 username admin
1    user1 FALSE
2    user2 FALSE
3    user3  TRUE
4    user4 FALSE
5    user5  TRUE
6    user6 FALSE
7    user7 FALSE
8    user8  TRUE
9    user9  TRUE
```

使用方括号来访问数据帧中的 vector。

```
> users[1]
 Username
1    user1
2    user2
3    user3
4    user4
5    user5
6    user6
7    user7
8    user8
9    user9
```

使用 $ 符号来分隔各列，再用方括号通过索引来访问这些列中的数据。R 中的 $ 符号与大多数其他语言中的“.”符号非常像。

```
> users$admin[3]
[1] TRUE
```

2. 导入外部数据

既然你已经知道了如何保存数据，再来了解一下如何导入数据。可以使用 read.table()

函数从一个平面文件（flat file）[⊖]读取数据。该函数可接受 4 个参数：第一个参数是要读入平面文件的路径，第二个参数是一个布尔值，指出平面文件的第一行是否为标题行，第三个参数是一个用于分割列的字符，第四个参数则指出用作行标识符的列。read.table() 函数的返回值为一个数据帧。

```
read.table([path to file], [treat first row as headers],[character to treat as delimiter],[column
to make row identifier])
```

例如，假定有如下的平面文件，其中是一些累积问题的明细：

```
Section,Resolved,UnResolved,Total
Regression,71,32,103
Compliance,4,2,6
Development,19,8,27
```

读入这些数据，可以使用下边的代码：

```
bugData <- read.table("/bugsbyUS.txt", header=TRUE, sep=",", row.names="Section")
```

如果测试 bugData 对象的结果，你就会看到如下内容：

```
> bugData
            Resolved UnResolved Total
Regression        71         32   103
Compliance         4          2     6
Development       19          8    27
```

3. 循环

R 支持 for 循环，也支持 while 循环，正如你所期望的，它们都是非常符合结构化语法的：

```
for(n in list){}
while ([condition is true]){}
```

要循环访问上面的 users 数据帧，只需要如下操作：

```
for(i in users){
print(users$admin[i])
}
[1] FALSE FALSE  TRUE FALSE FALSE
[1] TRUE
```

下面是对错误（bug）数据的访问：

```
> for(x in bugData$UnResolved){
+      print(x)
```

⊖ 平面文件（flat file）是一种包含没有相对关系结构的记录的文件，通常用来描述文字处理、其他结构字符或标记被移除的文本。在使用上，有一些模糊点，如像换行标记是否可以包含于“Flat File(flat file)”中。在任何事件中，许多用户把保存成“纯文本”（text only）类型的 Microsoft Word 文档叫做“Flat File（flat file)”。最终文件包含记录（一定长度的文本的行数）但没有信息，例如，用多长的行来定义标题或者一个程序用多大的长度来用一个内容表对该文档进行格式化。Flat File 的另一种形式是包含了用 ASCII 码记录的，每个表单元由逗号分隔，用行表示记录组的文件。这种文件也叫做用逗号分隔数值（CSV）的文件。——译者注

```
+ }
32
2
8
```

4. 函数

如你所想，R 中的函数操作也能传递参数，函数接收参数并返回数据值。注意，R 的数据都是按值传递的。

构造函数用如下方法：

```
functionName <- function([parameters]){

}
```

2.5.3 使用 R 进行简单绘图

到这里，我们才真正开始体会 R 的乐趣。你已经知道了如何导入数据，如何存储数据，如何循环数据，现在，让我们来可视化数据！

R 本身就提供了几个绘图函数。我们先看看 plot() 函数。

依据传递参数的不同，plot() 函数将显示不同类型的图表。它可以接受如下参数：一个支持 plot() 函数的 R 对象；一个可选参数，当第一个参数不包含 y 轴值时，它作为 y 坐标轴的补充参数；命名的图形参数的个数；一个字符串参数，指明了 plot 函数的绘制类型（后边会详细描述该参数）；图表的标题；图表的子标题；x 轴标签；y 轴标签；最后是一个数值，指明了图表的长宽比（即 y/x 的值）。

让我们看看，plot 的类型选项是如何反映到图表的显示中的。注意，当你将 plot 函数应用于 users 数据帧时，你就会得到一个数值表示的数据表，x 轴是 users 的 username 列，y 轴是 users 的 admin 列，其可取数值的范围从 0 ~ 1（不是 TRUE 和 FALSE），如图 2-23 所示。

```
plot(users, main="plotting user data frame\nno type specified")
plot(users, type="p", main="plotting user data frame\ntype=p for points")
plot(users, type="l", main="plotting user data frame\ntype=l for lines")
plot(users, type="b", main="plotting user data frame\ntype=b for both")
plot(users, type="c", main="plotting user data frame\ntype=c for lines minus points")
plot(users, type="o", main="plotting user data frame\ntype=o for overplotting")
plot(users, type="h", main="plotting user data frame\ntype=h for histogram")
plot(users, type="s", main="plotting user data frame\ntype=s for stair steps")
plot(users, type="n", main="plotting user data frame\ntype=n for no plotting")
```

R 的基本安装版也支持柱状图，使用 barplot() 函数创建。barplot() 函数接收的最有用的参数分别是：用于模型化条柱高度的 vector 或者 matrix 参数，一个可选的 width（宽度）参数，一个指定每个条柱前边空格数量的参数，一个列出所有条柱名称的 vector 参数，图例使用的文本参数，一个名称为 beside 的用于指明柱状图是否为堆积图的布尔型参数，另一个用于设定条柱显示是横向的还是纵向的布尔型参数，一个用于指明条柱颜色的 vector 参数，以及用于指明每个条柱边框颜色的 vector 变量，还有图表标题

以及子标题参数。要了解所有的参数列表，只需要在控制台窗口中输入“?barplot”就可以了。

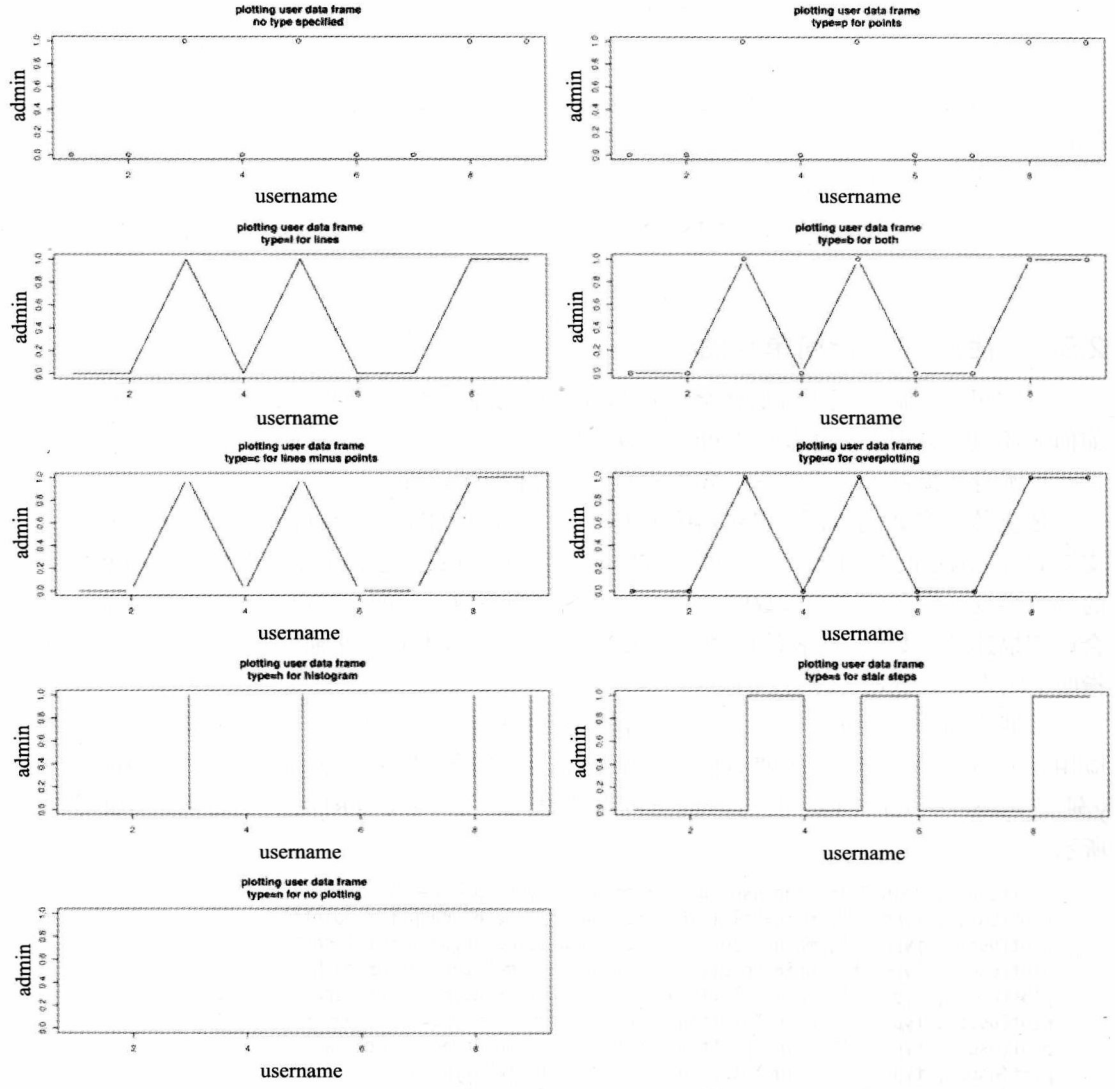


图 2-23 使用 plot 函数显示不同类型的图表

图 2-24 所示为用柱状图表示 users 数据帧的情况。

```
barplot(users$admin, names.arg=users$username, main="Bar chart of users that are admins")
```

R 本身也支持创建泡沫图表（bubble chart），即使用 symbols() 函数，给这个函数传递一个想要显示的 R 对象，然后将 circle 参数设置为某一列值，该列的值表示每一个圆的半径。

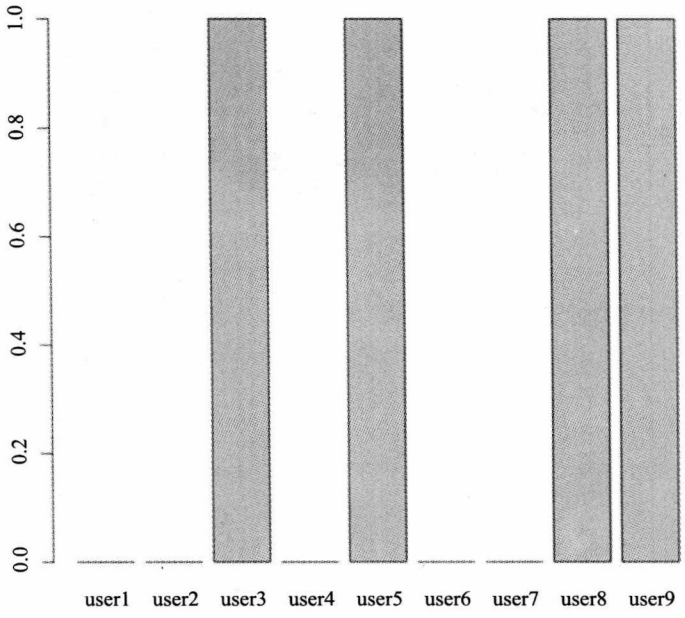


图 2-24 用户为 admin 的柱状图

注意

泡沫图表用于表示三维数据。它们有点像散点图（scatter plot），点的位置依赖于 x 轴和 y 轴指定的值，但是泡沫图中点的半径也要指定值。

图 2-25 所示为泡沫图。

```
symbols(users, circles=users$admin, bg="red", main="Bubble chart of users that are admins")
```

symbols() 函数也可以用于绘制其他形状的图表，要了解更多关于这个函数的信息，在 R 控制台上输入“?symbols”即可。

通过调用适当的函数，可以将生成的图表另存为希望的文件格式。注意，当结束输出操作之后，需要调用 dev.off。比如，要创建一个 barplot 函数生成的图表的 JPEG 文件，需要调用：

```
jpeg("test.jpg")
barplot(users$admin)
dev.off()
```

用于此用途的函数有：

```
pdf([filename]) #把图表另存为pdf格式的文件
win.metafile([filename]) #把图表另存为 Windows metafile格式的文件
png([filename]) #把图表另存为png格式的文件
jpeg([filename]) #把图表另存为pg格式的文件
bmp([filename]) #把图表另存为bmp格式的文件
postscript([filename]) #把图表另存为ps格式的文件
```

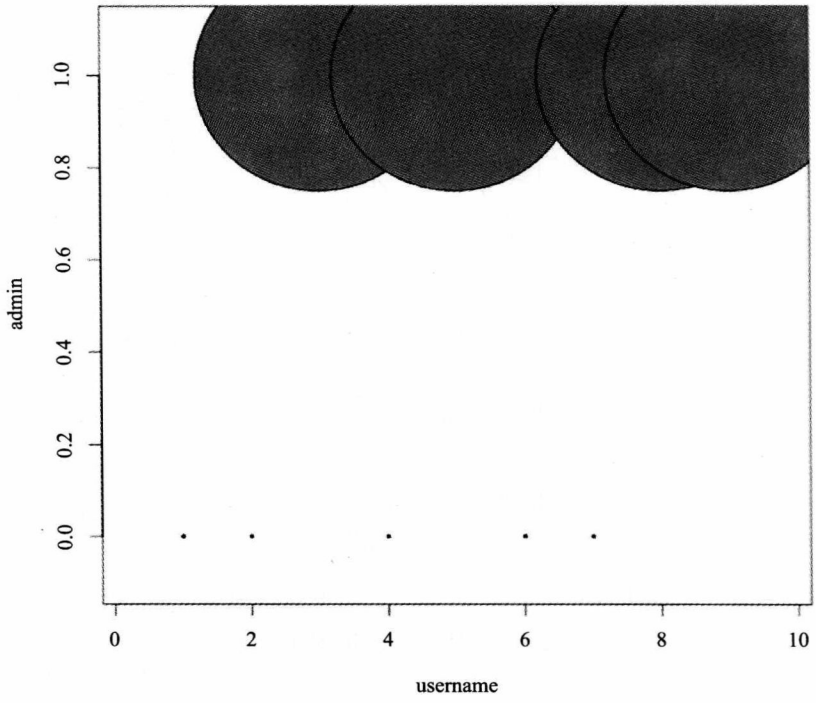


图 2-25 用户为 admin 的泡沫图

2.5.4 R 的一个实例

到现在为止，我们只是粗浅认识了 R 最起码的功能，但是，我们也已经了解了 R 的基本构造模块，现在，我们构造前文提到的如图 2-19 所示的一个图表。

首先，要先创建 4 个变量：一个数值型的变量，保存未压缩 JavaScript 文件的数量；一个 vector 型变量，保存工具的名称；第三个也是 vector 型变量，用于保存运行了工具之后 JavaScript 文件的大小；最后一个是 vector 型变量，用于保存文件缩减之后的大小与原始文件大小的百分比。

最后一个变量等会儿再给它重新赋值，现在我们先给它硬编码，这样你就能看到我们用于获取百分比的方式——新的文件大小除以文件总大小，然后乘以 100，最后再用 100 减去刚才得到的值。

```
originalSize <- 2150
tool <- c("YUI", "Closure Compiler (simple)", "Minify", "Closure Compiler (advanced)")
size <- c(1638, 1638, 573, 0)
diff <- c((100 - (1638  / 2150)*100), (100 - (1638 / 2150)*100), (100 - (573 / 2150)*100), (100
- (0 / 2150)*100))
```

接下来，将这些向量变量构造成一个数据帧，并将向量的行标识符设置为所用的工具名称。

```
mincompare <- data.frame(diff, size, row.names=tool)
```

如果你在控制台中输入 mincompare，你就会看到如下的结构：

```
>mincompare
                                diff size

YUI                           23.81395 1638
Closure Compiler (simple)     23.81395 1638
Minify                        73.34884  573
Closure Compiler (advanced) 100.00000    0
```

太棒了！到此就可以开始构造图表了。使用 barplot 函数绘制 diff 列。让柱状图表以水平方式显示，显式地将 y 轴名称设置为数据帧的行名称，并设置图表标题为“ Percent of file size reduction by product”（不同产品文件大小缩减页面比）：

```
barplot(mincompare$diff, horiz=TRUE, names.arg =row.names(mincompare), main="Percent of file size
reduction by product")
```

如果你在控制台上运行上述指令，你就会看到，我们距离目标已经很近了，如图 2-26 所示。

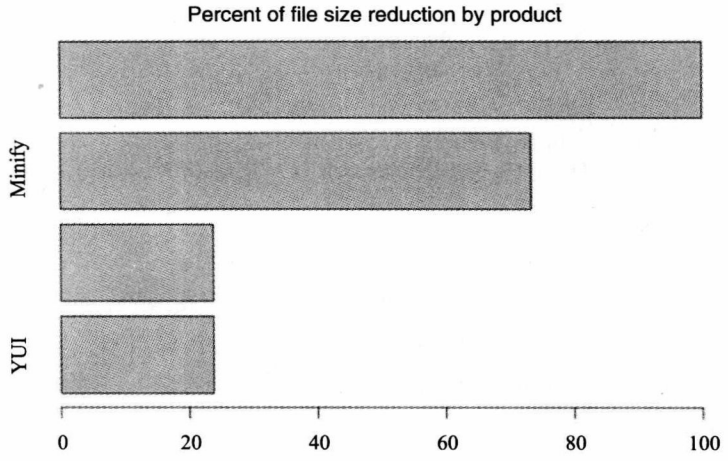


图 2-26 图表初稿

由图 2-26 可见，y 轴的第 1 个和第 3 个名称不见了，这是因为这两个名称太长了，无法按垂直方式显示出来。你可以像设置柱状图一样将文本设置成“水平方式”显示来修正该问题。

要想达到这样的效果，需要使用 par 函数设置图形的参数。但是首先需要保存现有的参数，这样当创建完图表之后，可以将这些参数再还原回来：

```
opar <- par(no.readonly=TRUE)
```

该指令将已有的参数保存到一个名为 opar 的变量中，这样当你完成其他工作之后，就可以重新检索这些参数了。

接下来，你就可以使用 par() 函数，将文本设置为“水平”显示：

```
par(las=1, mar=c(10,10,10,10))
```

par() 函数接受几个参数。上述代码将 las 参数（用于变更坐标轴标签的样式）设置为 1，这样，设置坐标轴标签以水平方式显示，并为 mar 参数赋值以设置图表的间隙。

调用 barplot 函数绘制完图表之后，就可以使用下面的代码恢复最初的图形参数了：

```
par(opar)
```

要想保存图表，需要将它导出。可以在 pdf 函数调用中包含 barplot 函数和 par 函数，并传递想要保存图表的文件名称作为参数。本例中将图表导出为 PDF 文件，并保留矢量线条和原始文本以便于进行编辑，也方便使用 Illustrator 或其他类似程序在后期制作中进行加工。

```
pdf("Figure 2-19.pdf")
```

还原了图形参数之后，调用 dev.off() 函数关闭文件。

到这里，代码应为如下所示：

```
originalSize <- 2150
tool <- c("YUI", "Closure Compiler (simple)", "Minify", "Closure Compiler (advanced)")
size <- c(1638, 1638, 573, 0)
diff <- c((100 - (1638  / 2150)*100), (100 - (1638 / 2150)*100), (100 - (573 / 2150)*100), (100
- (0 / 2150)*100))
mincompare <- data.frame(diff, size, row.names=tool)

pdf("Figure 2-19.pdf")
opar <- par(no.readonly=TRUE)
   par(las=1, mar=c(10,10,10,10))
   barplot(mincompare$diff, horiz=TRUE, names.arg =row.names(mincompare), main="Percent of file
size reduction by product")
par(opar)
dev.off()
```

但是仍然有些事情困扰着我。我不喜欢对 diff 变量进行硬编码，而且为设置这个变量的值，我们要一遍一遍地重复相同的算法。我们来将这些内容提取出来，单独形成一个函数。

把这个函数命名为 getPercentImproved，使其接受两个参数，一个是 vector 型，一个是数值型的：

```
getPercentImproved <- function(sourceVector, totalSize){}
```

在这个函数中，创建了一个空的 vector 变量，它将保存函数计算的结果，函数的结尾处会将这个变量返回。

```
percentVector <- c()
```

然后，对传递进来的 vector 变量进行循环遍历：

```
for(i in sourceVector){}
```

在迭代过程中，运行我们的算法来获取每一个缩减后文件与原文件的大小。记住，算法是：

$$(100-([\text{新文件大小}])/[\text{原文件大小}])\times 100)$$

将计算结果保存在新的 percentVector 变量中：

```
percentVector <- c(percentVector,(100 - (i  / totalSize)*100))
```

循环结束后，返回这个新的 vector 变量值。

```
return(percentVector)
```

最终，函数应该如下所示：

```
getPercentImproved <- function(sourceVector, totalSize){
    percentVector <- c()
    for(i in sourceVector){
        percentVector <- c(percentVector,(100 - (i  / totalSize)*100))
    }
    return(percentVector)
}
```

最后，将 diff 的值设置为 getPercentImproved 函数计算的结果，并将 size 和 originalSize 变量传递给 getPercentImproved 函数。

```
diff <- getPercentImproved(size, originalSize)
```

最终的代码如下所示：

```
getPercentImproved <- function(sourceVector, totalSize){
    percentVector <- c()
    for(i in sourceVector){
        percentVector <- c(percentVector,(100 - (i  / totalSize)*100))
    }
    return(percentVector)
}

originalSize <- 2150
tool <- c("YUI", "Closure Compiler (simple)", "Minify", "Closure Compiler (advanced)")
size <- c(1638, 1638, 573, 0)
diff <- getPercentImproved(size, originalSize)
mincompare <- data.frame(tool,diff, size, row.names=tool)

pdf("Figure 2-19.pdf")
opar <- par(no.readonly=TRUE)
    par(las=1, mar=c(10,10,10,10))
    barplot(mincompare$diff, horiz=TRUE, names.arg =row.names(mincompare), main="Percent of file
size reduction by product")
par(opar)
dev.off()
```

如果愿意，还有很多方法可以对此进行进一步完善。你可以将图表的生成和导出功能提取出来生成一个函数。

或者可以使用 R 的原生函数 apply() 来得到新旧文件大小差距，而不是使用向量循环的方式。我们现在就来看看这种方法。

2.5.5 使用 apply() 函数

apply() 函数允许我们将一个函数应用到某个列表的各个元素上。它有几个参数，第一

个参数是一个值列表，接下来是一个数值类型的 vector，指出如何将函数应用到列表上（1 表示应用到行上，2 表示应用到列上，c(1,2) 表示应用到行与列上），最后一个参数是应用到列表的函数：

```
apply([list], [how to apply function], [function to apply])
```

可以使用如下的方法取代 getPercentImproved 函数：

```
diff <- apply(as.matrix(size), 1, function(x)100 - (x  / 2150)*100)
```

注意，这里将 size 变量转化为一个 matrix 类型变量 as，并传递给 apply() 函数。这是因为 apply() 只接受 matrix、array 或者数据帧类型的变量。apply() 函数有个派生函数 lapply()，也可以使用这个函数。

使用 apply() 函数会使代码更简练，这也正是这门语言的目的所在，它遵循语言的哲学，意味着它是一种统计分析的逻辑性编程语言，而非指令式的编程语言。更新后的代码如下所示：

```
originalSize <- 2150
tool <- c("YUI", "Closure Compiler (simple)", "Minify", "Closure Compiler (advanced)")
size <- c(1638, 1638, 573, 0)
diff <- apply(as.matrix(size), 1, function(x)100 - (x  / originalSize)*100)
mincompare <- data.frame(diff, size, row.names=tool)

pdf("Figure 2-19.pdf")
opar <- par(no.readonly=TRUE)
   par(las=1, mar=c(10,10,10,10))
   barplot(mincompare$diff, horiz=TRUE, names.arg =row.names(mincompare), main="Percent of file
size reduction by product")
par(opar)
dev.off()
```

构造图表的最后一步是将图表导入到 Adobe Illustrator 或其他矢量绘制程序中美化图表，例如，调整文本的对齐方式，或者调整字体大小。虽然这些格式调整在 R 中也可以完成，但是在专业程序中，这些格式调整功能更强大。

2.6 小结

本章我们学习了几个工具，对于我们创建和维护高性能 Web 站点的目标来说，这些工具是无价之宝。我们看到了 Firebug 的 Network Monitoring（网络监控）标签是一个相当棒的工具，它能够对构成网页的网络依托的因素进行跟踪，并对影响网页访问速度的因素进行跟踪。由于其被动性，使得网络监控在我们开发网页或调试已知问题时，成为了一个非常棒的工具。

我们使用 YSlow 不同的过滤器来测试我们当前站点的性能，而且也用它们来获取一些定制性的技巧，对这些站点的性能进行更好地优化。

使用 Webpagetest，我们也能够看到外部资源产生的影响，并获得性能方面的提

示，而且我们能得到可重复查看的结果，获得几个不同方面的高级汇总数据。我们看到了 WebPagetest 非常丰富的配置设置，包括通过脚本能够测试更多复杂情景。我们也看到，WebPagetest 公开了一个 API，下一章将使用这些 API 进行性能监控的自动化。

我们研究了几种用于缩减 JavaScript 的工具，对每一个工具进行了实例应用，并通过可视化对每个工具运行后给出的文件大小的结果进行了比较。我们也开始关注一些抽象的细节，使得性能调节不仅仅体现为数字的形式。

接下来，我们又涉足了 R 语言。我们安装了 R 控制台，探究了 R 的一些基本概念，并使用 R 编码生成了第一个图表——跟用于缩减工具结果比较的图表是一样的。

在下一章中，我们将会使用并扩展我们掌握的 R 知识，也会利用一些本章讨论过的工具和概念。

第 3 章

WPTRunner——使用 WebPagetest 进行自动化性能监测与可视化

第 2 章为本书其他章节所要进行的工作打下了一个坚实的基础。我们知道了如何使用 Firebug 和 YSlow 生成瀑布图表，以及如何使用它们作为调试工具。我们探讨了 WebPagetest 的一些扩展特性和功能。我们讨论了缩减 JavaScript 的概念，并运行了一个多变量测试来比较不同缩减工具产生的结果。上一章的最后我们学习了 R，并编写了用于可视化多变量实例结果的脚本代码。

本章将延续这些概念。你将会学习如何对 WebPagetest 提供的 API 进行挂钩（hooking），以便对一些 URL 地址进行自动化监测，并使用 R 对检测结果进行可视化处理。

最终，你会创建一个框架，使用该框架你可以对你的站点的 Web 性能进行监测。也可以插入一些特定的 URL，进行跨网址的多变量测试，并比较因站点特性更新或者广告选用对性能造成的影响。

3.1 架构

我们先来充实架构。通常，当开启一个项目的时候，我会站在白板前边，挖掘项目中的高级概念，然后使用 UML 做出用例。我们还是打开 Omnigraffle 或 Visio 或一些其他的流程图应用软件，使用这些软件在架构文档上进行辅助工作。你可以创建一个 UML 序列图，来演示系统中的对象或过程之间交互的情况。

正如你在第 2 章看到的，WebPagetest 是一个功能非常丰富的 Web 应用程序，它可以提供详细的 Web 性能信息。它也公开了一些 API 函数，你可以调用这些函数来进行测试。记住这一点，你就知道需要怎样来调用 API，并获得回复。

还要知道的一点是，你需要解析 WebPagetest 返回的响应信息，因此，要在架构文档上创建 3 个过程：一个表示 WebPagetest，一个用于调用 API，最后一个用于处理响应。

我们来调用能激发 pagetest API 的 run_wp 函数的进程，以及用于处理结果的 process_wpt_response 进程。现在，我们的序列图应该如图 3-1 所示。

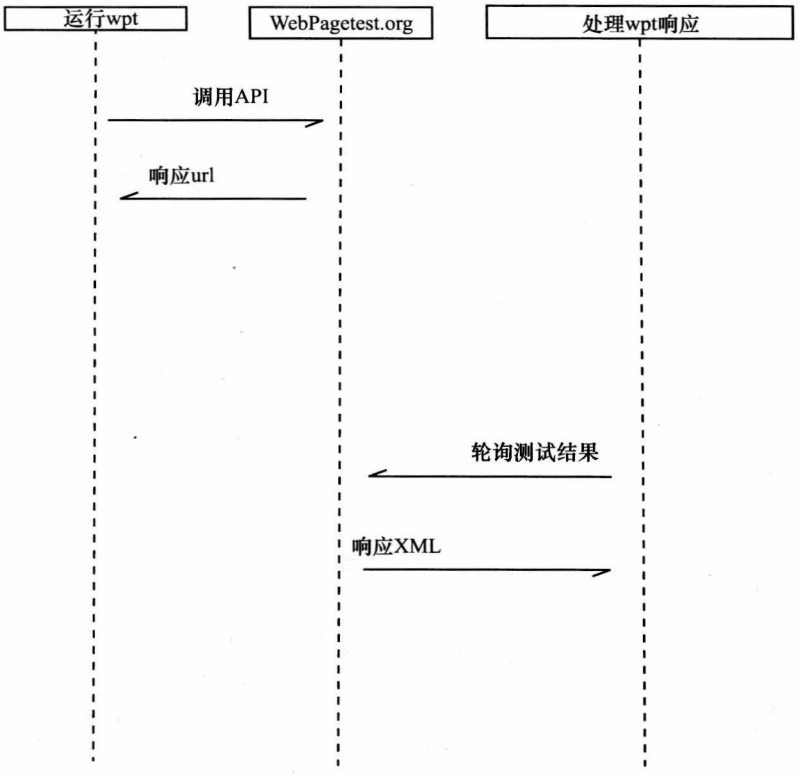


图 3-1 序列图的第一次迭代

这是一个很好的开始。你知道，依据我们上一章使用 WebPagetest 的经验，运行在 WebPagetest 上的测试通常不会马上运行。你必须将它放入队列中，即使已经在队列的最前端了，测试本身也会花一些时间。这一点是可以预料的，因为测试并不是真实经历的模拟，它实际上是通过你提供的代理来加载站点。

我们的策略是，当你调用 API，它返回一个你可以轮询的 URL 地址，这就意味着测试完成了。还有重要的一点需要注意，因为你想要保存返回的 URL，以便将来能重复地引用它们，直到测试完成，所以你会想要将它们保存在一个平面文件中。你需要将地址值添加到平面文件的末尾，也就是说，你需要将最新的值追加到文件最后。简单起见，这个平面文件我们就叫它 webpagetest_responses.txt。

你也需要使用另一个平面文件来存储测试完成后的结果，平面文件名是 wpo_log.txt。最后，你需要使用 R 脚本读取 wpo_log.txt 的内容来创建图表。

如果你将这些额外的部分添加到序列图中，最终的架构文档看上去就如同图 3-2 这样。

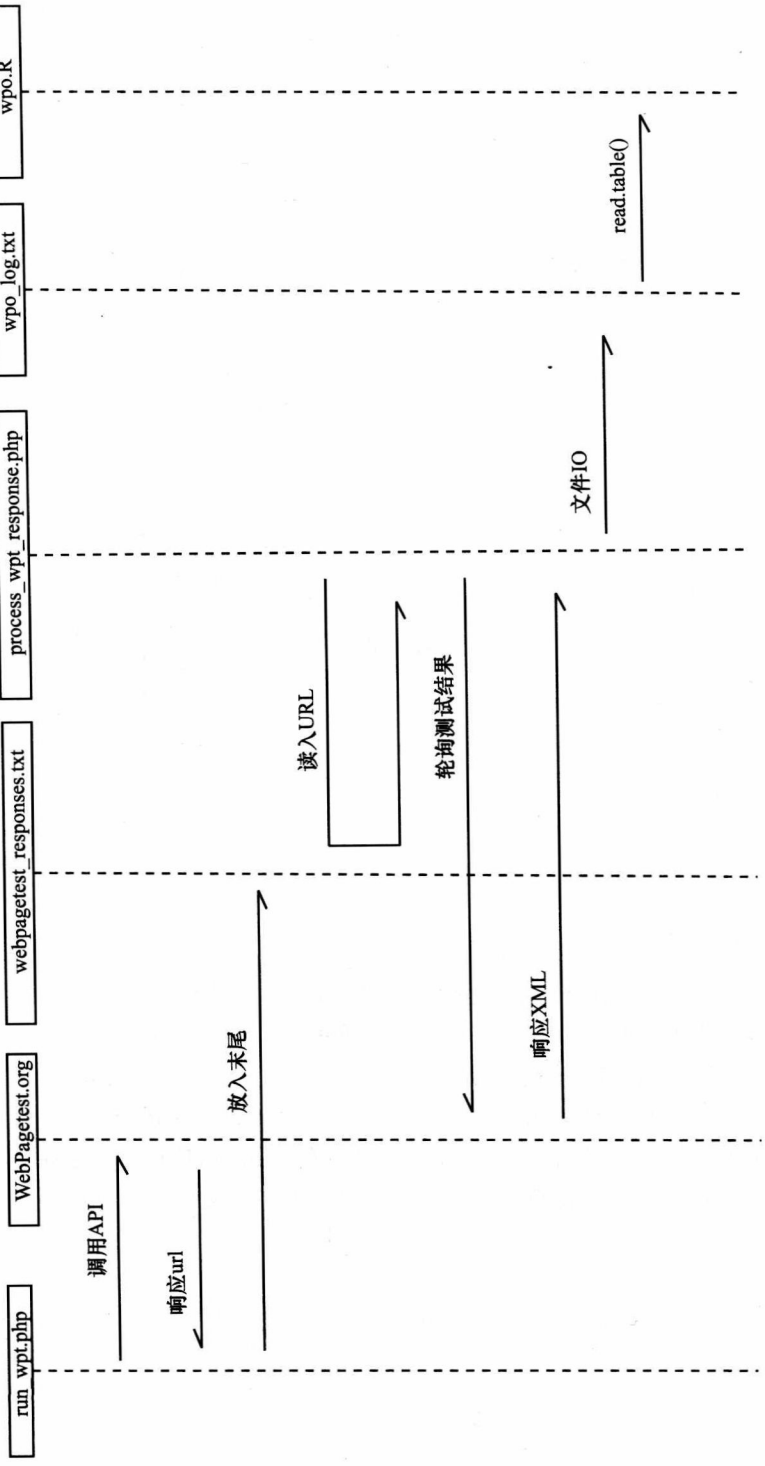


图 3-2 最终版 WPTRunner 序列图

这个例子中，还要创建一个 PHP 文件，来自动运行 run_wpt 以及 process_wpt_response 这两个进程，但实际上，你也可以使用任何其他语言来完成这件事。

出于对这个项目的需要，我们创建了如下所示的目录结构：

```
/data - 用于保存平面文件
/util - 用于保存共享文件和工具
/charts - 保存R生成的图表文件
```

3.2 创建一个共享配置文件

开始之前，需要一个访问 WebPagetest API 的 key，API 文档可以在 https://sites.google.com/a/webpagetest.org/docs/advanced-features/webpagetest-restful-apis 上找到，你可以给该网站的拥有者 Pat Meenan，发送邮件来申请 key，他的邮件地址是 pmeenan@webpagetest.org。发送邮件后，可能要等一两天或者更久才会收到 key。一旦获得了 API 的 key，你就应当创建一个单独的文件来保存所有需要在过程之间共享的配置信息。

Key 是用一个字符串保存的：

```
$key = "[your key here]";
```

你知道了应当将 API 的 key 存储在共享文件中，但是共享文件中也保存了一些其他你想要保存的东西。

由于框架的关键是用程序的方式来测试一组 URL 地址，那么你需要创建一种机制来保存这些 URL 地址。对于这些 URL，需要保存它们的实验特性或配置，或仅仅对它进行常规站点监控，因此，需要将这些 URL 保存在一个数组当中。

```
$urls_to_benchmark = array("tom-barker.com", "apress.com/", "amazon.com", "apple.com", "google.com");
```

最后，还需要保存引用的所有平面文件的路径：

```
$csvFiles = "/WPTRunner/data/webpagetest_responses.txt";
$wpologfile = "/WPTRunner/data/wpo_log.txt
```

这个共享文件，可以命名为 wpt_credentials_urls.php，它应该放在 /util 目录下，它看起来应该像下边这样：

```
<?php
$key = "xxxx"; // our API key
$urls_to_benchmark = array("tom-barker.com", "apress.com/", "amazon.com", "apple.com", "google.com"); // our list of URLs to monitor
$csvFiles = "/WPTRunner/data/webpagetest_responses.txt";  // flat file to store response URLs
$wpologfile = "/WPTRunner/data/wpo_log.txt"; // flat file to store test results
?>
```

从架构文档的内容看，你需要编写两个平面文件。制作一个共享文件来最小化文件的编写是一个不错的主意。所以我们创建一个名为 fileio.php 的文件，也将它放在 /util 目录下。

在 fileio.php 文件中，你需要构造一个函数 appendToFile()，并将文件路径和数据传递给

该函数。该函数会校验你传递的文件是否存在，如果存在，它会将文件数据追加到平面文件的后边，如果不存在，它会创建一个文件，并将传递进来的数据写入该文件。

```
<?php
function appendToFile($data, $file){
     echo "writing to file $file\n";
   $writeFlag = "w";
   if(file_exists($file)){
         $writeFlag = "a";
   }
   $fh = fopen($file, $writeFlag) or die("can't open file");
   fwrite($fh, $data . "\n");
   fclose($fh);
}

?>
```

到此，我们已经规划好了系统的架构，并创建了外部文件来保存配置信息和通用功能。现在我们开始把系统本身充实起来。

访问 WebPagetest API

好了，我们开始编写测试请求吧。访问 WebPagetest 的地址是 http://www.webpagetest.org/runtest.php。WebPagetest API 接收几个参数，其中的一些是：

- url：想要测试的 URL。
- location：指出测试时用到的代理所在位置、速度以及测试使用的浏览器，其格式是 location.browser:location。例如，Dulles（杜斯乐）的 Chrome 浏览器表示为 Dulles.Chrome。使用 IE 的话，会有些不同，例如 IE8，它的格式是 Dulles_IE8。
- runs：指定需要运行的测试数量。

注意

在文档中，为什么 IE 浏览器使用下划线字符这一点还不是很清楚，但是这一变量却很好记忆。

- fvonly：如果将 fvonly 置 1，只会得到第一个测试视图的结果，不会重复运行测试视图。
- private：将 private 标记置为 1，可以确保测试不会出现在公用测试列表中。
- block：这个参数允许用户设置一个逗号分割的块选项列表。还记得上一章中如何指定 URL 地址以及文件类型的么！
- f：指定测试结果的格式，可以是 xml 和 json 格式的。
- k：该参数指定了公共 API 的 key。

调用 API 的例子如下：

```
http://www.webpagetest.org/runtest.php?f=xml&private=1&k=111 &url=tom-barker.com
```

你需要创建一个新的名为 run_wpt.php 的文件。首先要做的是构造 URL。你还需要导入共享配置文件，确保你可以访问 API 的 key 以及 URL 数组：

```
require("util/wpt_credentials_urls.php");
```

然后，你需要创建一些变量来保存 API 的参数，本例中，你只需要注意输出格式参数即可，这里要设置为 XML 格式，并确保我们的测试是不公开的。

```
$outputformat = "xml";
$private = 1;
```

最后，你需要创建一个新变量 $wpt_url 来存储除了 URL 参数之外的所有参数连接起来的 URL 地址。稍后再添加 URL 参数，如下所示：

```
$wpt_url = "http://www.webpagetest.org/runtest.php?f=$outputformat&private=private&k=$key&url=";
```

设置 URL 参数，需要迭代遍历在 wpt_credentials_urls.php 中设置的 URL 数组。

```
for($x=0;$x<count($urls_to_benchmark); $x++){
}
```

遍历这个循环，一次性取出数组中的所有元素，将它们连接到 $wpt_url 变量中。然后使用 PHP 原生函数 file_get_contents() 访问该 URL，读取服务器的响应信息并保存在 $wpt_response 变量中：

```
$wpt_response = file_get_contents($wpt_url . $urls_to_benchmark[$x]);
```

记住，API 会返回一个 XML 结构，你需要对其进行解析。该 XML 结构如下所示：

```
<response>
<statusCode>200</statusCode>
<statusText>Ok</statusText>
<data>
<testId></testId>
<ownerKey></ownerKey>
<xmlUrl></xmlUrl>
<userUrl></userUrl>
<summaryCSV></summaryCSV>
<detailCSV></detailCSV>
</data>
</response>
```

你可以使用 SimpleXMLElement() 函数将 API 返回的字符串转换为 XML 对象，并将结果保存在 $xml 变量中。

```
$wpt_response = file_get_contents($wpt_url . $urls_to_benchmark[$x]);
$xml = new SimpleXMLElement($wpt_response);
```

这样，你就可以对结果进行解析并提取必要的数据了。第一块要检验的数据是 statusCode 节点，它保存了 HTTP 的响应状态值。如果是 200 就意味着响应没有问题，你就可以读取 xmlURL 节点的值了，测试完成之后，该节点就包含了测试结果的 URL 值，你要将该值写到 webpagetest_responses.txt 平面文件中。

```
if($xml->statusCode == 200){
    appendToFile($xml->data->xmlUrl, $csvFiles);
}
```

完成后的 run_wpt.php 文件如下所示：

```
<?php
require("util/wpt_credentials_urls.php");
require("util/fileio.php");
$outputformat = "xml";
$private = 1;
$wpt_url = "http://www.webpagetest.org/runtest.php?f=$outputformat&private=private&k=$key&url=";

for($x=0;$x<count($urls_to_benchmark); $x++){
    $wpt_response = file_get_contents($wpt_url . $urls_to_benchmark[$x]);
    $xml = new SimpleXMLElement($wpt_response);
    if($xml->statusCode == 200){
        appendToFile($xml->data->xmlUrl, $csvFiles);
    }
}
?>
```

注意

确保在命令行中运行 run_wpt.php，而不是在浏览器中运行。要想在命令行中运行 PHP 文件，只需要调用 PHP 二进制文件并指定文件路径后面的参数为 -f 即可。-f 参数告诉解释器读入文件，解析并执行它。所以，要运行 run_wpt.php，需要在控制台键入：

```
>php -f run_wpt.php
```

该命令会生成一个平面文件，其格式如下所示：

```
http://www.webpagetest.org/xmlResult/120528_SK_db5196c3143a1b81aacc30b2426cec71/
http://www.webpagetest.org/xmlResult/120528_TB_0ca4cfaa17613b0c2213b4e701c5a9dd/
http://www.webpagetest.org/xmlResult/120529_TN_8c20efe8c82a663917456f56aae7c235/
http://www.webpagetest.org/xmlResult/120529_6P_253e58b1cda284b9cf9a80becd19ef9f/
```

3.3 解析测试结果

现在，你有了一个平面文件，包含了指向 WebPagetest 测试结果的 URL 列表。这些测试并不是即刻完成运行，所以你需要等待它们运行完成，然后再开始对结果进行解析。

你可以对这些测试结果 URL 进行轮询，看看它们是否已经完成了。测试结果格式如下所示：

```
<response>
<statusCode></statusCode>
<statusText></statusText>
<data>
<startTime></startTime>
</data>
</response>
```

要想知道测试是否已经完成，需要检查 statusCode 节点。状态码为 100 表示测试正在等待，状态码为 101 表示测试已经启动，200 表示测试已经完成。状态码 400 则表示有错误产生。

记住这些，我们就可以开始对结果进行解析了！

首先，你需要将共享文件包含进来，这样你就可以访问 fileio() 函数以及平面文件的路径了：

```
require("util/wpt_credentials_urls.php");
require("util/fileio.php");
```

然后，你需要创建一个函数来检测测试结果。将这个函数命名为 readCSVurls()，接受两个参数：$csvFiles，该参数指向平面文件 webpagetest_responses.txt，该文件依次保存测试结果的 URL；$file，该参数指向平面文件 wpo_log.txt，该文件保存从测试结果中取出的值。

```
function readCSVurls($csvFiles, $file){
}
```

该函数中，如果有 wpo_log 文件，就删除它。这仅是例行公事罢了，因为这样做比检查文件中可用的位置并在该位置插入数据更加容易，很少出错，只需要重写结果就可以了。然后，要检查 webpagetest_responses 是否存在，是否可读。如果是，打开该文件，然后循环遍历该文件：

```
unlink($file); //delete wpo_log
if (file_exists($csvFiles) && is_readable ($csvFiles)) { //if the file is there and readable
     $fh = fopen($csvFiles, "r") or die("\n can't open file $csvFiles");
     while (!feof($fh)) {}
}
```

上述 while 循环，直到读到文件结束符，循环结束。在这个循环中，你要读取文件的每一行（每一行都包含了测试结果 URL）。取出 URL 并将来自服务器的响应一并存储在 $tailEntry 变量中：

```
$line = fgets($fh);
$tailEntry = file_get_contents(trim($line));
```

如果有响应，你需要将其从字符串形式转换为 XML 对象，保存在 $xml 变量中：

```
if($tailEntry){
$xml = new SimpleXMLElement($tailEntry);
```

响应 XML 结构如下所示：

```
<response>
     <statusCode></statusCode>
     <statusText></statusText>
     <requestId></requestId>
     <data>
          <runs></runs>
          <average>
               <firstView>
               </firstView>
               <repeatView>
               </repeatView>
          </average>
          <run>
               <id></id>
               <firstView>
                    <results>
                    </results>
                    <pages>
                    </pages>
                    <thumbnails>
```

```
                    </thumbnails>
                    <images>
                    </images>
                    <rawData>
                    </rawData>
                </firstView>
                <repeatView>
                    <results>
                    </results>
                    <pages>
                    </pages>
                    <thumbnails>
                    </thumbnails>
                    <images>
                    </images>
                    <rawData>
                    </rawData>
                </repeatView>
            </run>
        </data>
    </response>
```

这个结构中，第一个需要注意的是 statusCode 节点。如果它的值是 200，那么响应结果是没有问题的，你可以开始解析 XML 了。如果值不是 200，你应该知道测试还没有完成，需要停止其他任何进一步的处理：

```
if($xml->statusCode == 200){

}else{
die("report not ready at webpagetest yet.\n");
}
```

从这个 XML 对象中，你可以取出你运行测试的目标页面的 URL，将它作为一个行标识符。从该 XML 对象中，还可以取出测试完成的日期、页面加载时间、页面总文件大小，以及用于生成页面所必需的 HTTP 请求的数量。

将这些值连接起来，形成一个单独的字符串，以逗号分隔，赋给 $newline 变量：

```
$url = $xml->data->run->firstView->results->URL;
$date = $xml->data->completed;
$loadtime = $xml->data->run->firstView->results->loadTime;
$bytes = $xml->data->run->firstView->results->bytesInDoc;
$httprequests = $xml->data->run->firstView->results->requests;
$newline = "$url, $date, $loadtime, $bytes, $httprequests";
```

注意

你正在读取 firstView 节点的结果数据，这也是你可以测试未缓存页面的原因。如果你想对页面的缓存版本进行测试，则请读取 repeatView 节点的数据。

你要检查一下 wpo_log 文件是否存在。如果不存在——第一遍循环的时候，它就不会存在，因为在函数的开始你就将它删除了，你就需要格式化该平面文件，保证其有需要的头部。

我们对上述功能进行抽象，形成一个函数。现在，你可以在这里放置一个存根函数，

取名为 formatWPOLog()。当完成了当前函数之后，再返回来定义 formatWPOLog() 函数的具体实现。

你需要将 $newline 变量以及 wpo_log 文件的路径传递给共享函数 appendToFile：

```
if(!file_exists($file)){
formatWPOLog($file);
}
appendToFile($newline, $file);
```

最后，当你将 webpagetest_responses 文件中的所有行都轮询完成之后，关闭该文件：

```
fclose($fh);
```

最后还剩一件事情——定义 formatWPOLog() 函数的实现。它需要接收一个指向文件的路径参数，并将“url,day,date,loadtime,bytes,httpreques ts.”作为第一行添加到该文件中。该行文本也就是列标题：

```
function formatWPOLog($file){
    $headerline = "url, day, date, loadtime, bytes, httprequests";
    appendToFile($headerline, $file);
}
```

3.4 完成实例

完成后的 process_wpt_response 文件如下所示：

```
<?php

require("util/wpt_credentials_urls.php");
require("util/fileio.php");


function readCSVurls($csvFiles, $file){
    unlink($file);
    if (file_exists($csvFiles) && is_readable ($csvFiles)) {
        $fh = fopen($csvFiles, "r") or die("\n can't open file $csvFiles");
        while (!feof($fh)) {
            $line = fgets($fh);
            $tailEntry = file_get_contents(trim($line));
            if($tailEntry){
                $xml = new SimpleXMLElement($tailEntry);
                if($xml->statusCode == 200){
                    $url = $xml->data->run->firstView->results->URL;
                    $date = $xml->data->completed;
                    $loadtime = $xml->data->run->firstView->results->loadTime;
                    $bytes = $xml->data->run->firstView->results->bytesInDoc;
                    $httprequests = $xml->data->run->firstView->results->requests;
                    $newline = "$url, $date, $loadtime, $bytes, $httprequests";
                    if(!file_exists($file)){
                        formatWPOLog($file);
                    }
                    appendToFile($newline, $file);
                }else{
```

```
                    die("report not ready at webpagetest yet.\n");
                }
            }
        }
    }
    fclose($fh);
}

function formatWPOLog($file){
    $headerline = "url, day, date, loadtime, bytes, httprequests";
    appendToFile($headerline, $file);
}


readCSVurls($csvFiles, $wpologfile);

?>
```

process_wpt_response 函数的输出也就是平面文件 wpo_log，现在它的内容应当如下所示：

```
url, day, date, loadtime, bytes, httprequests
http://tom-barker.com, Tue, 29 May 2012 20:12:21 +0000, 10786, 255329, 42
http://apress.com/, Tue, 29 May 2012 20:12:47 +0000, 4761, 714655, 57
http://amazon.com, Tue, 29 May 2012 20:13:07 +0000, 2504, 268549, 94
http://apple.com, Tue, 29 May 2012 20:12:41 +0000, 3436, 473678, 38
http://google.com, Tue, 29 May 2012 20:12:50 +0000, 763, 182802, 13
```

使用 R 绘制图表

每天运行一下 run_wpt 函数来生成当天的日志，该日志是一个以逗号分隔的加载时间、总负载以及针对每个你想要跟踪的 URL 的 HTTP 请求数量的列表。有这个日志实在太棒了，但它只是整个事件的一部分。你不可能分发这些数据，以期望你的用户们了解整件事件。你应该用可视化的方式来表现这些数据。

所以，我们来使用 R 构造一个基于时间序列的图表，来显示性能的变化情况。首先，需要创建一个新的 R 文件。从架构图中，我们知道这需要调用 wpo.R 文件。

首先要创建用于保存数据和图表目录的路径变量：

```
dataDirectory <- "/Users/tbarke000/WPTRunner/data/"
chartDirectory <- "/Users/tbarke000/WPTRunner/charts/"
```

然后读取 wpo_log 文件中的内容，并将它们存储到一个称为 wpologs 的数据帧中：

```
wpologs <- read.table(paste(dataDirectory, "wpo_log.txt", sep=""), header=TRUE, sep=",")
```

注意

上述代码利用 paste() 函数将储存的数据目录与文件名连接起来，以此构建文件的完整路径。在 R 中，没有用于字符串连接的操作符，这跟大多数其他语言一样。所以必须用 paste() 函数来完成此功能。该函数可接收 N 个参数，用于连接字符串，以及一个称为 sep 的参数，该参数指明了在字符串之间起分割作用的字符串——例如，如果你想在字符

串之间插入一个逗号，或者其他类型的分隔符，就可以使用该参数了。上述例子中，使用的是空格符。

同样的道理，你需要创建一个变量来保存我们创建的图表的文件路径。这个变量，我们定义为 wpochart，并将字节列的数据从字节（byte）转换为 KB。

```
wpochart <- paste(chartDirectory, "WPO_timeseries.pdf", sep="")
wpologs$bytes <- wpologs$bytes / 1000 #convert bytes to KB
```

3.5 数据解析

到现在，你已经将数据加载进来了，而且想要绘制的图表文件的路径也设置好了。让我们来看看数据结构是什么样的。在控制台中键入 wpologs，看到的数据结构如下所示：

```
> wpologs
                  url  day                        date loadtime   bytes httprequests
1  http://tom-barker.com  Tue  29 May 2012 20:12:21 +0000    10786 255.329           42
2     http://apress.com/  Tue  29 May 2012 20:12:47 +0000     4761 714.655           57
3      http://amazon.com  Tue  29 May 2012 20:13:07 +0000     2504 268.549           94
4       http://apple.com  Tue  29 May 2012 20:12:41 +0000     3436 473.678           38
5      http://google.com  Tue  29 May 2012 20:12:50 +0000      763 182.802           13
6  http://tom-barker.com  Wed  30 May 2012 16:28:09 +0000     5890 256.169           42
7     http://apress.com/  Wed  30 May 2012 16:27:56 +0000     4854 708.577           57
8      http://amazon.com  Wed  30 May 2012 16:28:14 +0000     3045 338.276          112
9       http://apple.com  Wed  30 May 2012 16:27:58 +0000     3810 472.700           38
10     http://google.com  Wed  30 May 2012 16:28:09 +0000     1524 253.984           15
```

想象一下时间序列图表是什么样子的——你想要绘制一个随时间变化的针对每个 URL 的性能度量曲线。这也意味着你需要分离出每个 URL 数据，要做到这一点，需要创建一个 createDataFrameByURL() 函数。然后传递你想要分离的 wpologs 数据和 URL 给该函数：

```
createDataFrameByURL <- function(wpologs, url){
}
```

在这个函数中，你要创建一个空的数据帧，用于填充数据并作为函数的返回值。

```
df <- data.frame()
```

接下来，你需要循环访问传递进来的 wpologs 数据：

```
for (i in 1:nrow(wpologs)){}
```

在循环内部，你需要验证 url 列，看看是否与传递进来的 URL 值相匹配。如果匹配的话，将数据添加到函数开始创建的新数据帧中：

```
if(wpologs$url[i] == url){
   df <- rbind(df , wpologs[i,])
}
```

然后设置这个新数据帧的 row.names（行名），并从函数返回该数据帧：

```
row.names(df) <- df$date
return(df)
```

现在，你有了通过 URL 值分离 WebPagetest 结果的函数了，你还需要创建新的数据帧来保存每一个你想要图表化的特殊 URL。

```
tbdotcom <- createDataFrameByURL(wpologs, "http://tom-barker.com")
apr <- createDataFrameByURL(wpologs, "http://apress.com/")
amz <- createDataFrameByURL(wpologs, "http://amazon.com")
aapl <- createDataFrameByURL(wpologs, "http://apple.com")
ggl <- createDataFrameByURL(wpologs, "http://google.com")
```

如果你在控制台中检查这些新数据帧，你会看到：

```
> tbdotcom
                                              url  day         date  loadtime    bytes
httprequests
29 May 2012 20:12:21 +0000 http://tom-barker.com  Tue  29 May 2012     10786  255.329
42
30 May 2012 16:28:09 +0000 http://tom-barker.com  Wed  30 May 2012      5890  256.169
42
31 May 2012 12:33:52 +0000 http://tom-barker.com  Thu  31 May 2012      5877  249.528
42
01 Jun 2012 17:58:19 +0000 http://tom-barker.com  Fri  01 Jun 2012      3671  255.337
42
03 Jun 2012 14:41:38 +0000 http://tom-barker.com  Sun  03 Jun 2012      5729  249.590
42
```

3.6 绘制加载时间

太棒了！现在我们开始绘制其中的一列。最重要的列是页面加载时间，所以我们就先从它开始。你需要使用 plot() 函数，并将 tbdotcom$loadtime 传递给它。你需要为 plot() 函数设置如下选项：

```
type="l"：绘制的是线条；
xaxt="n"：不画x轴；
col="#ff0000"：线条颜色为红色；
ylab = "Load Time in Milliseconds"：y轴显示标签为 “Load Time in Milliseconds” 。
```

plot() 函数的调用如下所示：

```
plot(tbdotcom$loadtime, ylim=c(2000,10000), type="l", xaxt="n", xlab="", col="#ff0000",
ylab="Load Time in Milliseconds")
```

如果不隐藏横轴，则横坐标将显示为一组数字，因为当 R 将数值识别为名义坐标系数时，会默认将横轴名称显示为递增整数。为了将日期显示在 x 轴上，你需要绘制一个定制的轴。使用 axis() 函数就可以做到。axis 函数接收的第一个参数是一个数值，表示绘制哪个轴：1 表示绘制底端数据轴，2 表示绘制左侧数据轴，3 表示绘制顶端数据轴，4 表示绘制右侧数据轴。其余参数都是命名参数值，包括：

at：指明了在哪里绘制刻度线；

lab：轴标签的值，或者是一个布尔值，或者是一个字符串向量；

tick：布尔值，表示是否显示选中标记；

lty：指明了线型；

lwd：指明线宽。

你也可以传递图形参数给该函数，更详细的信息，可以通过在控制台输入“?axis”或“?par”了解。

因此，你需要使用 axis() 函数绘制 x 轴并显示所有的数据，如下所示：

```
axis(1, at=1: length(row.names(tbdotcom)), lab= rownames(tbdotcom), cex.axis=0.3)
```

到现在为止，我们绘制的图表如图 3-3 所示。

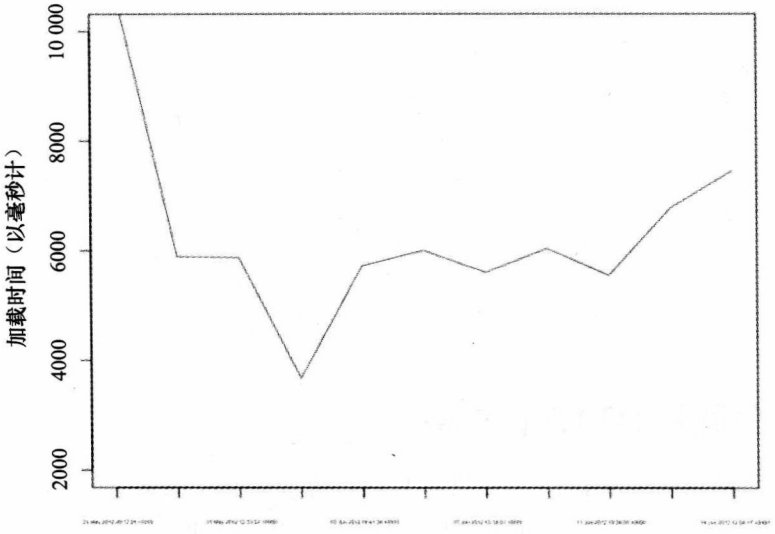


图 3-3 tom-barker.com 加载时间变化曲线图

太棒了！现在我们将其他 URL 的加载时间叠加进去。你可以使用 line() 函数绘制曲线，该函数跟 plot() 函数一样简单，可以向该函数传递一个要绘制的 R 对象，并选择线型和颜色。

我们开始为每一个跟踪的 URL 绘制加载时间曲线。

```
lines(apr$loadtime, type="l", lty = 2, col="#0000ff")
lines(amz$loadtime, type="l", col="#00ff00")
lines(aapl$loadtime, type="l", col="#ffff00")
lines(ggl$loadtime, type="l", col="#ff6600")
```

将这些曲线绘制在一起，加载时间图表如图 3-4 所示：

太令人兴奋了，这基本上是 WebPagetest 中最重要的数据了。但是，你也会想要对测试结果中性能的其他方面进行图表化演示，因为它们对于页面最终加载时间也是有影响的。如果你对页面大小和 HTTP 请求数进行图表化，你就会绘制出一个更大的图形，图形中显示了

页面速度的指标。

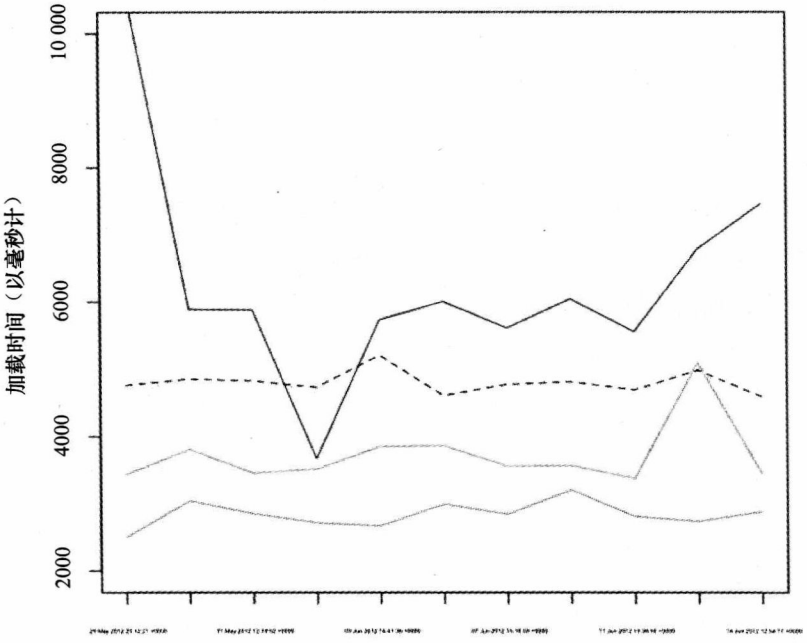


图 3-4 加载时间序列图表

3.7 绘制负载和 HTTP 请求数

在接下来的图表中，我们加入负载和 HTTP 请求数这些元素。每一个元素的度量尺度是不一样的——页面速度是以几千毫秒来表示的，HTTP 请求数是 10 以内的数值，而文件大小用几百个字节来表示。你可以对所有这些数据进行规范化，并用标准差这样的术语来描述它们，这会从真实的数值中抽象出数据，而你却需要使用真实的数值来解决问题。

与上述表示方法不同，你需要创建一个时间序列图表来表示质量的每一个方面。但是，你需要将这些时间序列图表囊括在一个单一图表中。其原因是，这样遵循了 Edward Tufte 关于优秀数据设计的原则——使包含数据的图表尽可能紧凑。

下面的代码对数据帧中其他列进行了图表化处理，包括 bytes（字节数）列、httprequests（http 请求）列以及 loadtime（加载时间）列，它们使用同样的绘制方式。

```
plot(tbdotcom$bytes, ylim=c(0, 1000), type ="l", col="#ff0000", ylab="Page Size in KB", xlab="",
xaxt="n")
axis(1, at=1: length(row.names(tbdotcom)), lab= rownames(tbdotcom), cex.axis=0.3)
lines(apr$bytes, type="l", lty = 2, col="#0000ff")
lines(amz$bytes, type="l", col="#00ff00")
lines(aapl$bytes, type="l", col="#ffff00")
```

```
lines(ggl$bytes, type="l", col="#ff6600")

plot(tbdotcom$httprequests, ylim=c(10, 150), type ="l", col="#ff0000", ylab="HTTP Requests",
xlab="", xaxt="n")
axis(1, at=1: length(row.names(tbdotcom)), lab= rownames(tbdotcom), cex.axis=0.3)
lines(apr$httprequests, type="l", lty = 2, col="#0000ff")
lines(amz$httprequests, type="l", col="#00ff00")
lines(aapl$httprequests, type="l", col="#ffff00")
lines(ggl$httprequests, type="l", col="#ff6600")
```

现在，添加一个图例，这样你就可以知道哪个颜色对应哪个 URL 了。首先，你需要创建向量来保存每一个 URL 所使用的标签和对应的颜色。

```
WebSites <- c("tom-barker.com", "apress.com/", "amazon.com", "apple.com", "google.com")
WebSiteColors <- c("#ff0000", "#0000ff", "#00ff00", "#ffff00", "#ff6600")
```

然后，使用 plot 函数绘制一个新的图表，type 参数赋值为“n”，保证图表中先不绘制任何线条，xaxt 和 yaxt 参数也赋值为“n”，保证不绘制数据轴。这里你需要使用 legend() 函数添加一个图例，并将刚才创建的标签和颜色向量传递给该函数。

```
plot(tbdotcom$httprequests, type ="n", xlab="", ylab="", xaxt="n", yaxt="n", frame=FALSE)
legend("topright", inset=.05, title="Legend", WebSites, lty=c(1,2,1,1,1,1,1,1,1,1,1,1,1,1,2),
col= WebSiteColors)
```

为了将所有这些图表绘制在一个图形中，就需要将所有使用到的 plot() 函数绑定到一个称为 par() 的函数中，然后为其传递一个命名参数 mfrow。mfrow 参数接收一个向量，用来指定有多少行、多少需要绘制。所以，如果传递的是 c(2,2)，则表示要绘制 2 行、2 列。

```
par(mfrow=c(2,2))
```

通过这种方法，绘制的图形如图 3-5 所示。

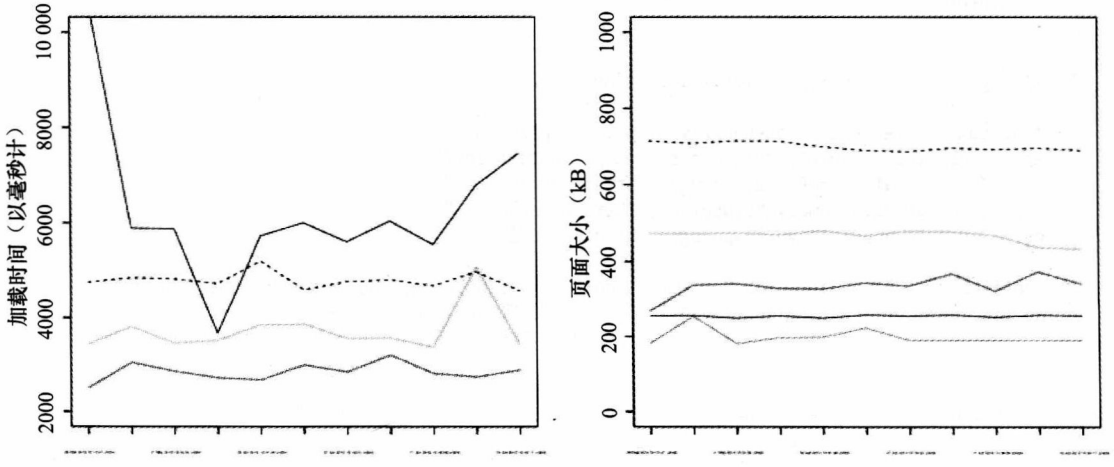


图 3-5 完成之后的时间序列图

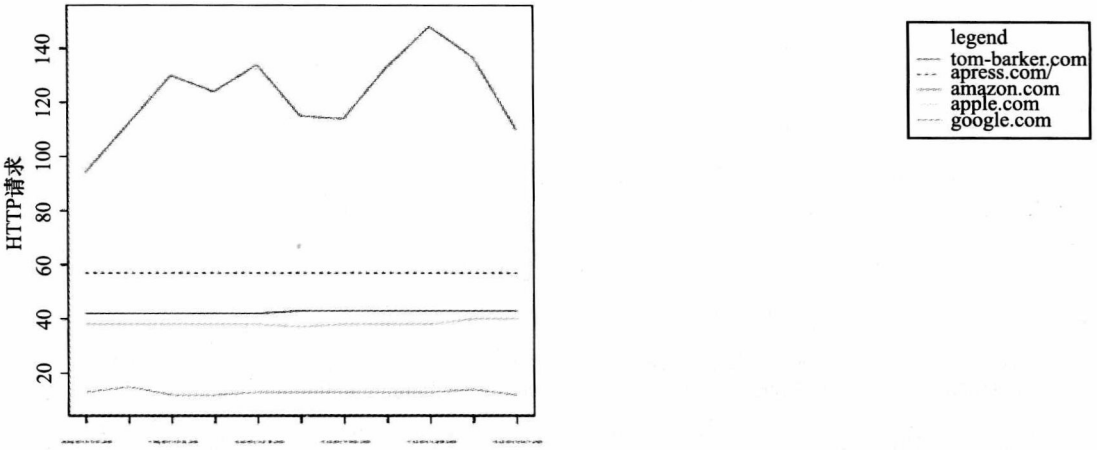


图 3-5 （续）

完成的 R 代码如下所示：

```
dataDirectory <- "/Users/tbarke000/WPTRunner/data/"
chartDirectory <- "/Users/tbarke000/WPTRunner/charts/"
wpologs <- read.table(paste(dataDirectory, "wpo_log.txt", sep=""), header=TRUE, sep=",")
wpochart <- paste(chartDirectory, "WPO_timeseries.pdf", sep="")


createDataFrameByURL <- function(wpologs, url){
df <- data.frame()
for (i in 1:nrow(wpologs)){
    if(wpologs$url[i] == url){
        df <- rbind(df , wpologs[i,])
    }
}
row.names(df) <- df$date
return(df)
}

wpologs$bytes <- wpologs$bytes / 1000 #convert bytes to KB

tbdotcom <- createDataFrameByURL(wpologs, "http://tom-barker.com")
apr <- createDataFrameByURL(wpologs, "http://apress.com/")
amz <- createDataFrameByURL(wpologs, "http://amazon.com")
aapl <- createDataFrameByURL(wpologs, "http://apple.com")
ggl <- createDataFrameByURL(wpologs, "http://google.com")

WebSites <- c("tom-barker.com", "apress.com/", "amazon.com", "apple.com", "google.com")
WebSiteColors <- c("#ff0000", "#0000ff", "#00ff00", "#ffff00", "#ff6600")

pdf(wpochart, height=12, width=12)
par(mfrow=c(2,2))
plot(tbdotcom$loadtime, ylim=c(2000,10000), type="l", xaxt="n", xlab="", col="#ff0000",
ylab="Load Time in Milliseconds")
axis(1, at=1: length(row.names(tbdotcom)), lab= rownames(tbdotcom), cex.axis=0.3)
lines(apr$loadtime, type="l", lty = 2, col="#0000ff")
lines(amz$loadtime, type="l", col="#00ff00")
```

```
lines(aapl$loadtime, type="l", col="#ffff00")
lines(ggl$loadtime, type="l", col="#ff6600")

plot(tbdotcom$bytes, ylim=c(0, 1000), type ="l", col="#ff0000", ylab="Page Size in KB", xlab="",
xaxt="n")
axis(1, at=1: length(row.names(tbdotcom)), lab= rownames(tbdotcom), cex.axis=0.3)
lines(apr$bytes, type="l", lty = 2, col="#0000ff")
lines(amz$bytes, type="l", col="#00ff00")
lines(aapl$bytes, type="l", col="#ffff00")
lines(ggl$bytes, type="l", col="#ff6600")


plot(tbdotcom$httprequests, ylim=c(10, 150), type ="l", col="#ff0000", ylab="HTTP Requests",
xlab="", xaxt="n")
axis(1, at=1: length(row.names(tbdotcom)), lab= rownames(tbdotcom), cex.axis=0.3)
lines(apr$httprequests, type="l", lty = 2, col="#0000ff")
lines(amz$httprequests, type="l", col="#00ff00")
lines(aapl$httprequests, type="l", col="#ffff00")
lines(ggl$httprequests, type="l", col="#ff6600")


plot(tbdotcom$httprequests, type ="n", xlab="", ylab="", xaxt="n", yaxt="n", frame=FALSE)
legend("topright", inset=.05, title="Legend", WebSites, lty=c(1,2,1,1,1,1,1,1,1,1,1,1,1,1,2),
col= WebSiteColors)
dev.off()
```

与往常一样，有多种方法可以用来重构和提炼代码。你可以将时间序列数据的绘制抽象到一个单独的函数中，以减少重复代码的数量。我没这么做，是因为我想要强调我创建时间序列的步骤。你也可以在 createDataFrameByURL() 函数中使用 R 的原生函数 apply() 取代数据的循环。我保留了循环，目的是尽最大可能来描述代码是如何处理数据的。

3.8 开源

我已经将这个工程命名为 WPTRunner，并作为开源项目，将其源代码放到了 Github 上，地址是 https://github.com/tomjbarker/WPTRunner，欢迎你下载源代码，并为了自己的目的而使用，在你自己的项目中启动源代码，或者给出使用反馈，改进我们的项目，以方便其他人使用。

3.9 小结

本章使用 WebPagetest、PHP 以及 R，探讨了用于跟踪一组 URL 的 Web 性能的自动化方法。这一自动化方法还可以做很多事情。当然你能够跟踪网站一段时间内的性能，而且也可以用它来跟踪你想要添加到你的网站的某些实验特性的性能，甚至跟踪页面在上线之前、开发过程中的性能。

你可以使用这份报告的结果，生成一份你想要的性能标准，并创建一个流程，确保新的特性不会影响到性能。你也可以产生内部的、外部的以及主管级别的报告，你可以把这个

数据与其他数据联系起来，比如发布日期、用户访问数，或其他数字数据以及相关数据。

下一章，我们将要创建一个 JavaScript 库，来对你用到的 JavaScript 的运行时性能进行评估，并对数据进行图表化加工以及提取有用的指标，使用这些指标，不仅可以形成你自己的代码标准，也可以用来影响产品的决策和浏览器支持矩阵。作为奖励，本章的结尾我们给出一个 Patrick Meenan（WebPagetest 的创始人）的访谈，其内容是关于 WebPagetest 这款工具以及它的未来。

附：WebPagetest 的创办人 Patrick Meenan 访谈

Patrick Meenan 非常友善，他回答了本书读者关于 WebPagetest 很多方面的问题。

创建 WebPagetest 的最初灵感是什么？

回顾 2006 ~ 2007 年这段时间，我们创建 WebPagetest 时，确实遇到了一些困难的情况：

（1）我们大多数的开发者使用的是 Firefox 浏览器（因为 Firebox 可以使用 Firebug）。

（2）我们的办公室就在数据中心的街对面，因此开发人员连接到服务器基本是高带宽 / 低延迟。

其结果是，开发者会感觉到页面访问总是非常快，开启一个用于加速的会话都是很难的了。我们对于服务的监控都是来自于主干网络的测试，所以，这不是一种理想的状况。因此，我们需要一款工具，可以让开发人员看到用户所能看到的性能状况（在更有代表性的浏览器上，使用更实际的网络连接）。

让这个项目开源化是一个很艰难的决定么？

不，一点也不。对于将工具开放给社区所带来的好处，AOL 非常支持也非常理解，特别是由于它当时还没有一个特定的业务策略。

当你参与到这个项目当中时，遇到过什么挑战没有？

获取浏览器的访问接口以得到有用的信息一直是最大的挑战。某些当前的较新的浏览器可以让我们较为方便地获取信息，但是它们不提供扩展功能或者不提供可以访问 IE 内部请求链的浏览器 API。我尝试了许多不同的技术手段，尝试不同的 API 拦截点，直到找到一个合适的拦截点（而且情况是不断变化的，即使是当前的注入点也仍然是有问题的，特别是关于 HTTPS 的注入点）。

另一个大的挑战是时间。开发 WebPagetest 并不是我在 AOL 的日常工作，因此我花了很久才完成了它的部分功能。我要特别感谢 Google，让我现在可以全职工作于 WebPagetest 这个项目上（虽然做这个工作，时间上总是不够用）。

WebPagetest LLC 的目标是什么？

WebPagetest LLC 最初是作为 WebPagetest 的一个 shell 来开发的，WebPagetest 为其提供了生存空间。2012 年年初，我和几个 Web 性能开发人员一起创办了 WPO 基金。当前我们仍处于非盈利状态，但是我们的目标就是对开源 Web 性能方面的工作以及免费数据研究给予投资。WebPagetest 已经纳入到了 WPO 基金当中，并将会给基金提供一些帮助。

测试结果要保存多长时间？

理论上是永远。到现在我还保存着 2008 年我第一次对服务器进行测试时的数据。到现在为止，我保存的数据大概有 2TB 了，但是随着存储系统的发展，存储这些数据在可预见的未来不会有什么问题。WebPagetest 有一些内置的自动压缩支持功能，你可以在一段时间之后，对测试数据进行压缩，然后保存到外部存储设备或云端，当要访问这些数据时，WebPagetest 知道如何自动恢复压缩的数据，因此长时间保存数据不是什么问题。也就是说，当前我不维护离线的备份数据，所以当有灾难发生时，数据将全部丢失（我做的是在线冗余数据拷贝）。

运行测试平均要花多长时间？

平均要花多少时间我也不确定——这依赖于运行测试的数量以及测试是否包含首次和重复的视图数据。每一次运行都有一个强制时间限制——60 秒（公用网络情况下），因此最差的时候，运行 10 次花 20 分钟完成一个测试用例。由于一个测试在某个时刻只能在一台指定机器上运行，当规模很大时，运行测试就非常耗费资源了（这也就是为什么通过 API 进行大量的测试没有得到广泛支持的原因）。

使用 API 时，除了轮询测试，你还推荐其他的方法么？

API 支持一种回调（callback）方法，通过该方法，服务器将请求发送给你指定的 URL（pingback 测试参数），但是你仍然想使用轮询，因为回调在某些情况下会失去作用，出于效率的考虑可以使用它，但它不会使你的代码更加简单。

队列的测试过程是怎样的？

每个地方都有一个单独的队列，所以它们是独立的，但这有一个基本的流程：

（1）runtest.php 将一个任务文件写入到指定位置的队列目录中，并把它加入到内存队列中（确保测试按序执行）。实际在一个指定的位置上有 10 个不同优先级的队列，因此测试会被放入恰当的队列中（用户启动的测试会得到最高优先级，API 测试使用中等优先级，还有一些其他比较明确的划分）。

（2）测试代理对服务器进行轮询，将服务器的位置作为请求的一部分进行指定（work/getwork.php）。

（3）服务器按队列优先级顺序进行选择，返回指定位置找到的第一个任务。

（4）测试代理解析测试请求，运行指定的测试。

（5）当运行完成之后，测试代理将运行结果上传给 work/workdon.php 文件。

（6）当测试全部完成之后，测试代理会发送一个附有“done”标记的测试数据。

在人们使用 WebPagetest 的过程中，你听到的最有趣的事情是什么？

当看到人们基于 WebPagetest 构建系统的时候，我感到非常兴奋。HTTP Archive(httparchive.org）就是一个大家都熟知的非常棒的例子，另外，很多公司使用 API 将 WebPagetest 整合进他们的内部系统中。

能谈一谈关于代理架构的知识么？

当前，确实有几个不同的代理。包括传统的 IE 代理，最新的支持 Chrome 和 Firefox

的 Windows 代理，以及基于 Akamai 开源代码的 iOS 和 Android 代理。也有一些实验用的 WebDriver/NodeJS 代理，提供在 Chrome 上的跨平台测试支持。WebPagetest 有一个标准的 API 用于测试代理，使用这些 API，可以很容易地将一个新的代理嵌入进来。

从架构上来看，Windows 代理可能是比较有趣的了，因为它们可以做一些非常疯狂的事情来访问所需要的数据。新的 Windows 代理可以做以下事情：

- 启动指定的浏览器进程（Chrome、Firefox 或者 IE）。
- 使用代码注入技术，向浏览器上下文注入并运行代码。
- 注入代码对 winsock、wininet 和 SSL API 函数安装 API 拦截功能，并对所有 API 调用进行拦截以记录瀑布数据。
- 注入代码也能运行在本机的 Web 服务器上（在浏览器内部），方便我们的浏览器扩展来与 Web 服务器进行通信。
- 我们的浏览器扩展（支持浏览器中的一个）通过使用 Ajax 轮询本地主机，来执行需要的任何命令（比如定位浏览器）。
- 浏览器扩展也会将它们遇到的事件发布给注入浏览器代码（比如 onload 事件）。

这是一个相当怪异的设置，仅仅为了控制浏览器和记录网络行为，就要考虑所有浏览器内部要运行的行为。

你是如何将 WebPagetest 应用到你开发的应用程序上的？

WebPagetest 是我开发的一款应用软件。大多数情况下，当与开发者处理网站问题时，我都会使用它，因此我会从用户以及开发者的角度来看待 WebPagetest。我主要关注瀑布图。我觉得所有的 Web 开发人员都应该不依赖检查列表或打分就知道如何解释瀑布图，这一点我不会做太多强调。观察网站如何加载是关键。

WebPagetest 近期与长期特性是什么？从个人角度来讲，我喜欢内存分析器。

目前，移动设备可能是这个领域最大的焦点。现有工具还比较原始，从移动设备上很难获取类似于从桌面上获取的数据，但是我们已经着手这个领域了。从浏览器上获取更多的信息总是最为重要的事情。我们最近增加了支持功能，可以从 Chrome 代理商捕获 Chrome dev 工具的时间表（timeline）数据，我们也正在寻找可用的其他选项。

第 4 章

perfLogger——JavaScript 基准测试和日志记录

第 3 章，我们编写了一个自动化的解决方案来跟踪和可视化一段时间内各个 URL 的 Web 性能。我们使用 WebPagetest 生成指标，使用 PHP 整合数据，使用 R 实现数据可视化。我们将这个项目命名为 WPTRunner，通过这个项目可以测试未来希望实现的优化和将要开发的新特性的性能所产生的功效。

对于跟踪 Web 性能来说，WPTRunner 非常全面。在本章中，我们会创建一个工具，以跟踪运行时性能。

通过创建一个 JavaScript 库，我们可以对运行时性能的所有方面（从特定的代码片段到函数以及模块的呈现时间）进行基准测试。然后对测试结果进行可视化处理。

我们会使用这些工具来对后续几章涉及的一些性能最佳实践的结果进行分析，但是我最希望看到的是，你能够借助这些工具和内心的渴望，以及积累起来的经验对测试结果进行量化。

4.1 架构

我们还是要从讨论架构阶段开始。在做任何尝试之前，考虑一下所有相关的部件，以及它们是如何交互的。

对代码进行基准测试有两个方法：或者基于时间，或者基于代码执行期间的执行操作数量。就我们的例子来讲，要确保将观测者效应（observer effect）最小化[⊖]，所以在代码执行过程中，应以毫秒为单位来计算所耗时间。要做到这一点，可以在代码执行之前捕获一个时间戳，然后在代码执行完成之后捕获另一个时间戳，最后计算两个时间戳的差值并将其作为我们的运行时间结果。

⊖ 所谓“观测效应”（observer effect）也就是简单观察某个行为对结果所产生的影响。对我们的测试来说，也就是在计时的操作过程中，执行额外的操作给基准测试所带来的代码延迟。

[结束时间]-[开始时间]=[运行时间]

开端非常不错。到目前为止我们所进行操作的流程如图 4-1 所示。

测试的结果可能会不同，导致的原因有很多，包括客户机器资源可利用率、JavaScript 引擎、浏览器渲染引擎，都会对测试结果产生影响。因此，我们要做的就是，覆盖整个用户群，在这个规模基础上执行多次测试，以多次平均作为参考基准。为了达到这个目的，我们需要把基准测试代码放入软件产品中，并记录我们能够进行分析和图表化处理的结果。

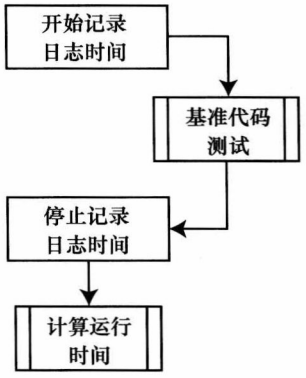


图 4-1 计算运行时间的工作流程

下一步是展示数据。我们已经有了用于计算运行时间的较为充实的一个工作流程，接下来，我们通过显示运行时信息给终端用户，或者将信息记录到服务器上来扩展这个流程。

假定有时候，我们不想将数据公之于众，因此我们要把是否公布数据这一环节加入我们的工作流程中。图 4-2 反映了添加这一逻辑之后的流程图。

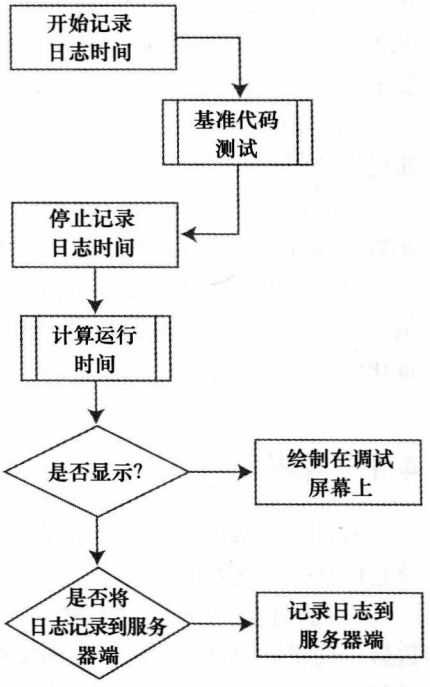


图 4-2 运行时日志记录流程图

到这里，我们对基准测试引擎所能进行的工作有了一个较高层次的描述。让我们调用 perfLogger 进程来定义这一功能。同时，我们还需要与其他进程交互来保存我们的数据以及图形化数据，所以，我们要创建一个序列图来具体描述一下这些进程的交互过程。

我们的基准测试进程 perfLogger 将会调用一个外部进程——savePerfData 进程。我们将会使用 XHR 对象向 savePerfData 进程传递数据。savePerfData 进程会依序将这些测试数据结果保存在一个平面文件中。我们将这个平面文件命名为 runtimeperf_results。

使用 R 读取该平面文件中的数据，就可以生成我们需要的图表了。这一工作流程如图 4-3 所示。

到目前为止，你已经看到了我们将要为基准代码生成的运行时数据，以及保存数据的流程。现在我们要考虑一下可以应用我们库中的哪些 API。

由于 perfLogger 最终要被外部代码使用，所以一定要考虑公共签名的问题，也要考虑到这个进程模块提供什么样的方法和属性，以及将哪些方法和属性公开给公共用户。

从图 4-2 可以看到，我们要用到 startTimeLogging() 和 stopTimeLogging() 函数。这两个函数在这一过程的头尾两端被调用。图 4-2 还指出，我们需要用到一个 drawToDebug

Screen() 函数，通过该函数将测试结果绘制到页面上，然后使用 logToServer() 函数将测试结果发送给 savePerfData 进程。

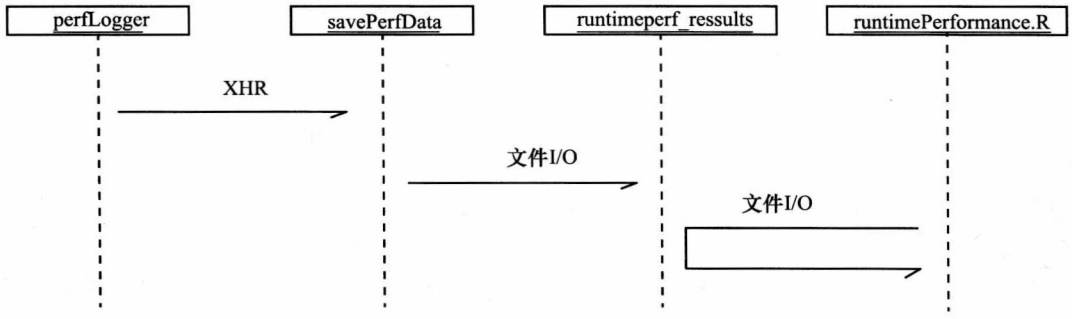


图 4-3 概要序列图

但是，正如我之前提到过的，这一逻辑过程仅仅能够捕获特定代码的运行时间。基准测试不仅仅用于捕获运行时间，它还可以获取多次执行结果的平均值，而这些执行过程会因为系统变量的不同导致运行结果集的偏差。因此，我们需要创建 logBenchmark() 函数来计算多次调用 startTimeLogging() 函数和 stopTimeLogging() 结果的平均值。

我们还需要暴露一个属性来保存 savePerfData 的 URL 地址，我们将这个属性命名为 serverLogURL。

最后，我们需要可以在同一页面上运行多次测试的一个方法。因此我们会创建一个关联数组，以字符串标识的方式保存不同的测试集合。我们将该关联数组命名为 loggerPool。

在所有这些方法和属性之中，我们想要为公众用户暴露哪些内容呢？实际上，我们不想公开暴露任何一个属性。如果我们这么做了，就会有使用外部代码来改变属性和方法的可能。如果我们确实想让外部代码来设置属性和方法，那么需要定义 setter 函数，但是目前，就我们的目标而言，还不需要这么做。

drawToDebugScreen() 函数、logToServer() 函数以及 calculateResults() 函数也不会公开给公共用户。基于流程图，我们需要传递一个布尔值来决定测试过程是否会显示或保存，因此最大的可能性是，我们会使用 stopTimeLogging() 函数依据测试结果对象中的属性来调用这些函数。

因此，我们只会将 startTimeLogging() 函数、stopTimeLogging() 函数以及 logBenchmark() 函数作为 API 公开出来。startTimeLogging() 函数和 stopTimeLogging() 函数允许用户获取特定代码的运行时间，使用 logBenchmark() 函数则可以得到真实的基准测试值，也就是函数多次执行的平均值。

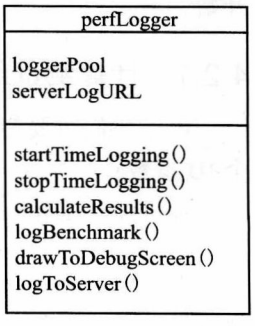


图 4-4 perfLogger 对象图

上述架构所涉及的对象如图 4-4 所示。我们会列出所有的属性和方法，并加粗显示公有的属性和方法。

在讨论到使用 loggerPool 来保存测试列表或测试结果字典的时候，我们需要明确地规

范出一个测试对象，以保存所有与测试相关的信息，包括 startTime、endTime，以及两个布尔值 drawToPage 和 Logtoserver 等属性。我们也会给予这个对象一个标识符，方便我们从字典中获取它，同时我们会使用一个变量来保存运行时间，这样就不需要在每次引用运行时间的时候重新计算了。

为了方便跟踪，我们还要在测试对象中包含运行测试的页面 URL 地址以及用户代理字符串。该元数据（metadata）有助于我们收集运行报告数据。再好好想一想这个结构，使用它我们就能够针对每一个使用用户代理的测试的性能进行汇报，或者后续我们还会向其中添加其他的元数据内容。

有了这些概念之后，我们就可以重构 perfLogger 图了，即将另一个私有函数——setResultsMetaData() 函数包含进来。

我们的测试结果对象图如图 4-5 所示。

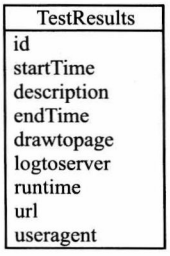


图 4-5 TestResults 对象图

4.2 开始编写代码

首先创建 perfLogger 对象。因为需要保存特定的私有属性和方法，所以我们会从一个自执行函数中以对象字面量（object literal）的语法方式返回一个 perfLogger 对象，同时，我们会在这个自执行函数中声明我们的私有变量：

```
var perfLogger = function(){
   var serverLogURL = "savePerfData.php",
   loggerPool = [];

   return {};
}()
```

接下来，我们开始编写私有函数。我们已经搭建了函数的骨架，以及将要使用的测试结果对象的结构，现在可以开始对这些内容进行详细编码了。我们以 calculateResults() 作为开始。

4.2.1 计算测试结果

由于我们将要使用 ID 值来引用测试结果对象，因此要让 calculateResults() 函数接收一个 ID 参数：

```
function calculateResults(id){
}
```

在 calculateResults() 函数中，我们使用传递进来的 ID 作索引的 loggerPool 数组来引用测试结果，并执行我们在架构阶段就讨论过的计算过程，即测试结束时间减去测试开始时间：

```
function calculateResults(id){
loggerPool[id].runtime = loggerPool[id].stopTime - loggerPool[id].startTime;
}
```

4.2.2 设置测试结果元数据

到这里，同样要对 setResultsMetaData() 函数进行详细编码了。传递 ID 参数，并使用 loogerPool 作为测试的引用。设置 url 属性为当前的 window 地址，设置 useragent 属性，让其指向 navigator 对象的 userAgent 属性。这会带给我们一些有趣的比对结果指标值，对比一下，看看每一个渲染引擎和 JavaScript 解释器在处理特定功能上的区别。

```
function setResultsMetaData(id){
loggerPool[id].url = window.location.href;
   loggerPool[id].useragent = navigator.userAgent;
}
```

4.2.3 显示测试结果

接下来，我们要添加 drawToDebugScreen() 私有函数。为了绘制一个调试屏幕，你需要在页面上添加一个命名 div 以便写入。第一步是在页面上创建一个元素的引用，其 ID 属性为“debug”，然后在 debug 变量中保存该引用。

然后，格式化调试信息——但是为了便于维护格式化模块，我们要将这一功能抽象化到我们自己的函数中，现在只使用一个存根函数。格式化输出结果被保存在一个称为 output 的变量中：

```
function drawToDebugScreen(id){
var debug = document.getElementById("debug")
   var output = formatDebugInfo(id)
}
```

现在，事情开始变得有趣了。你要测试一下 debug 变量是否有数值；通过这一点，你可以判断是否已经在页面上加入了一个命名元素以便写入。如果 debug 没有值，则创建一个新的 div 元素，将其 ID 置为“debug”，然后设置其 innerHTML 属性为我们的格式化输出变量 output，然后将 div 添加到页面上。

但是，如果 debug 已经存在，只需要添加当前的格式化输出信息到它的 innerHTML 属性中就可以了。

```
function drawToDebugScreen(id){
var debug = document.getElementById("debug")
   var output = formatDebugInfo(id)
   if(!debug){
      var divTag = document.createElement("div");
      divTag.id = "debug";
      divTag.innerHTML = output
      document.body.appendChild(divTag);
   }else{
      debug.innerHTML += output
   }
}
```

现在，实现 formatDebugInfo() 函数。我们的目标是，越简单越好。我们只是在一个段落中格式化字符串，将其描述信息加粗，运行时间和用户代理加下划线。唯一比较复杂的

是，基准值会有平均运行时间，而特定测试只有运行时间，因此你需要看看测试结果对象是否有 avgRunTime 属性，若有则使用该属性；若没有则使用默认的 runtime 属性。

```
function formatDebugInfo(id){
var debuginfo = "<p><strong>" + loggerPool[id].description + "</strong><br/>";
   if(loggerPool[id].avgRunTime){
      debuginfo += "average run time: " + loggerPool[id].avgRunTime + "ms<br/>";
   }else{
      debuginfo += "run time: " + loggerPool[id].runtime + "ms<br/>";
   }
   debuginfo += "path: " + loggerPool[id].url + "<br/>";
   debuginfo += "useragent: " +  loggerPool[id].useragent + "<br/>";
   debuginfo += "</p>";
   return debuginfo
}
```

4.2.4 保存数据

最后一个私有函数是 logToServer()。这里你还需要传递 ID 参数来引用 loggerPool 数组中的结果对象。但是，这一次通过原生函数 JOSN.stringfy() 将对象字面量序列化为一个字符串值。在该字符串值前添加“data=”，通过这种方式将从服务器端获取的 POST 变量数据用一个名字封装起来。

```
function logToServer(id){
   var params = "data=" + (JSON.stringify(loggerPool[id]));
}
```

接下来，logToServer() 函数创建一个新的 XHR 对象，为 POST 方法设置传递函数，然后指定我们保存的 serverLogURL 变量（指向 savePerfData）。由于该数据并不是关键业务，因此 logToServer() 函数并不处理 readystatechange 事件，只是将我们的数据 POST 给等待处理该数据的脚本。

```
function logToServer(id){
   var params = "data=" + (JSON.stringify(loggerPool[id]));
   var xhr = new XMLHttpRequest();
   xhr.open("POST", serverLogURL, true);
   xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
   xhr.setRequestHeader("Content-length", params.length);
   xhr.setRequestHeader("Connection", "close");
   xhr.onreadystatechange = function(){};
   xhr.send(params);
}
```

4.2.5 制定公有 API

现在，你可以在返回的对象字面量中创建公有函数了。首先处理 startTimeLogging() 函数。我们有让该函数接收 ID 参数作为测试结果的 ID、一个测试的描述参数，以及两个布尔值参数 drawToPage 和 logToServer。在该函数中，你需要在 loggerPool 中创建一个新的对象，使用传递的参数设置其属性值以及 startTime 属性值，该对象只是一个新的 Date 对象。

```
startTimeLogging: function(id, descr,drawToPage,logToServer){
   loggerPool[id] = {};
```

```
    loggerPool[id].id = id;
    loggerPool[id].startTime = new Date;
    loggerPool[id].description = descr;
    loggerPool[id].drawtopage = drawToPage;
    loggerPool[id].logtoserver = logToServer
}
```

如果你想更简练一些，可以创建一个构造函数，然后只需要在这里调用该构造函数。这个构造函数会封装已有功能，方便在其他地方复用，而不需要重写这些功能。

下边的代码用来实现 stopTimeLogging() 函数。仍然对照上面提到过的架构，stopTimeLogging() 函数只是设置测试结束的时间，然后调用私有函数来计算运行时间，最后设置测试结果的元数据：

```
loggerPool[id].stopTime = new Date;
calculateResults(id);
setResultsMetaData(id);
```

最后，stopTimeLogging() 函数检查布尔值，看看是否要将测试结果绘制到屏幕上，以及是否在服务器中记录日志。

```
if(loggerPool[id].drawtopage){
    drawToDebugScreen(id);
}
if(loggerPool[id].logtoserver){
    logToServer(id);
}
```

完整的函数代码如下所示：

```
stopTimeLogging: function(id){
    loggerPool[id].stopTime = new Date;
    calculateResults(id);
    setResultsMetaData(id);
    if(loggerPool[id].drawtopage){
        drawToDebugScreen(id);
    }
    if(loggerPool[id].logtoserver){
        logToServer(id);
    }
}
```

最后一个需要实现的函数是 logBenchmark()。这是到目前为止，我们的库中最复杂的一个函数了。站在高一点的角度来看，这个函数会使用传递给他的函数指针，调用几次 startTimeLogging() 函数和 stopTimeLogging() 函数，然后计算结果的平均值。

我们一步一步来看这个函数。首先你需要传递一个待用 ID，运行测试的次数，运行测试的函数，以及是否将结果绘制在页面上，是否将结果以日志的形式保存到服务器上。

```
logBenchmark: function(id, timestoIterate, func, debug, log){
}
```

接下来，创建一个变量，用于保存每次测试运行的时间值，并依据 timestoIterate 变量开始迭代：

```
var timeSum = 0;
for(var x = 0; x < timestoIterate; x++){
}
```

在该迭代循环中，调用 startTimeLogging() 函数，调用传递的函数，然后调用 stopTimeLogging() 函数，最后将每次迭代的运行时间累加到 timeSum 变量上：

```
for(var x = 0; x < timestoIterate; x++){
   perfLogger.startTimeLogging(id, "benchmarking "+ func, false, false);
   func();
   perfLogger.stopTimeLogging(id)
   timeSum += loggerPool[id].runtime
}
```

注意，循环中的每一次迭代都使用了同一个正在运行的测试的 ID，你只需要每次重写测试结果就可以了。另外要注意的是，传递两个 false 参数的代码表示不绘制调试屏幕或不对每次测试进行日志记录。

循环结束之后，你可以用总运行时间除以迭代次数以计算平均运行时间：

```
loggerPool[id].avgRunTime = timeSum/timestoIterate
if(debug){
   drawToDebugScreen(id)
}
if(log){
   logToServer(id)
}
```

现在，最终函数如下所示：

```
logBenchmark: function(id, timestoIterate, func, debug, log){
   var timeSum = 0;
   for(var x = 0; x < timestoIterate; x++){
      perfLogger.startTimeLogging(id, "benchmarking "+ func, false, false);
      func();
      perfLogger.stopTimeLogging(id)
      timeSum += loggerPool[id].runtime
   }
   loggerPool[id].avgRunTime = timeSum/timestoIterate
   if(debug){
         drawToDebugScreen(id)
   }
   if(log){
         logToServer(id)
   }
}
```

最终的函数库如下所示：

```
var perfLogger = function(){
   var serverLogURL = "savePerfData.php",
      loggerPool = [];

      function calculateResults(id){
         loggerPool[id].runtime = loggerPool[id].stopTime - loggerPool[id].startTime;
      }

      function setResultsMetaData(id){
```

```
        var debug = document.getElementById("debug")
        var output = formatDebugInfo(id)
        if(!debug){
           var divTag = document.createElement("div");
           divTag.id = "debug";
           divTag.innerHTML = output
           document.body.appendChild(divTag);
        }else{
           debug.innerHTML += output
        }
     }

     function logToServer(id){
        var params = "data=" + (JSON.stringify(loggerPool[id]));
        var xhr = new XMLHttpRequest();
        xhr.open("POST", serverLogURL, true);
        xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
        xhr.setRequestHeader("Content-length", params.length);
        xhr.setRequestHeader("Connection", "close");
        xhr.onreadystatechange = function(){};
        xhr.send(params);
     }
  function formatDebugInfo(id){
     var debuginfo = "<p><strong>" + loggerPool[id].description + "</strong><br/>";
     if(loggerPool[id].avgRunTime){
        debuginfo += "average run time: " + loggerPool[id].avgRunTime + "ms<br/>";
     }else{
        debuginfo += "run time: " + loggerPool[id].runtime + "ms<br/>";
     }
     debuginfo += "path: " + loggerPool[id].url + "<br/>";
     debuginfo += "useragent: " +  loggerPool[id].useragent + "<br/>";
     debuginfo += "</p>";
     return debuginfo
  }

return {
startTimeLogging: function(id, descr,drawToPage,logToServer){
  loggerPool[id] = {};
  loggerPool[id].id = id;
  loggerPool[id].startTime = new Date;
  loggerPool[id].description = descr;
  loggerPool[id].drawtopage = drawToPage;
  loggerPool[id].logtoserver = logToServer
},

stopTimeLogging: function(id){
  loggerPool[id].stopTime = new Date;
  calculateResults(id);
  setResultsMetaData(id);
  if(loggerPool[id].drawtopage){
     drawToDebugScreen(id);
  }
  if(loggerPool[id].logtoserver){
     logToServer(id);
  }
},
```

```
logBenchmark: function(id, timestoIterate, func, debug, log){
   var timeSum = 0;
   for(var x = 0; x < timestoIterate; x++){
      perfLogger.startTimeLogging(id, "benchmarking "+ func, false, false);
      func();
      perfLogger.stopTimeLogging(id)
      timeSum += loggerPool[id].runtime
   }
   loggerPool[id].avgRunTime = timeSum/timestoIterate
   if(debug){
        drawToDebugScreen(id)
   }
   if(log){
        logToServer(id)
      }
   }

}
}();
```

注意

由于现代解释器和系统的处理速度比较快，一些小的特定测试可能会产生0毫秒的计算结果。当进行基准测试时，要确保你的测试内容包含足够大的功能块以便获取结果。这也就意味着，如果你想对一个数组的迭代进行基准测试，你需要确保数组足够大，对它进行迭代至少要消耗1毫秒的时间。在后续的章节中，当涉及对小段代码进行基准测试时，还会对这一问题进行深入讨论，我们将在足够大的尺度范围上运行这些测试，看看隐含优势到底是什么。在第5章中，我们还要探讨高分辨率时间这一概念，高分辨率时间是一种可以让我们用亚毫秒来进行时间测量的特性。

4.3 远程日志记录

好吧，perfLogger.js调用savePerfData，将测试结果序列化为字符串的形式。接下来实现savePerfData。

你可以重复使用第3章WPTRunner例子中用到的fileio.php共享文件，创建两个变量：$logfile用于保存平面文件的路径，$benchmarkResults用于保存测试结果。$benchmarkResults变量从$_POST数组中提取结果。由于序列化对象使用前缀“data=”，因此这里就用字符串“data”来引用该对象。调用formatResults函数的存根，传递$_POST的引用，并将结果返回给$ benchmarkResults。最后，传递$benchmarkResults和$logfile给一个存根函数saveLog()：

```
<?php
require("util/fileio.php");

$logfile = "log/runtimeperf_results.txt";
$benchmarkResults = formatResults($_POST["data"]);

saveLog($benchmarkResults, $logfile);
?>
```

现在来实现 formatResults() 函数。它通过 $r 参数来接收一个对象：

```
function formatResults($r){}
```

记住，由于传递给 HTTP 的数据是序列化处理之后并进行过编码的，因此数据看起来如下所示：

```
{\"id\":\"fi\",\"startTime\":59,\"description\":\"benchmarking function useForInLoop() {\\n var
stepTest = populateArray(4);\\n for (ind in stepTest) {\\n
}\\n}\",\"drawtopage\":false,\"logtoserver\":false,\"stopTime\":59,\"runtime\":0,\"url\":\"ht
tp://localhost:8888/lab/perfLogger_example.html\",\"useragent\":\"Mozilla/5.0 (Macintosh; Intel
Mac OS X 10.5; rv:13.0) Gecko/20100101 Firefox/13.0.1\",\"avgRunTime\":0}
```

所以，在函数中，你需要去除掉所有的前置斜线，使用原生函数 json_decode() 获取序列化后的数据，将数据转化为原生的 JSON 对象：

```
$r = stripcslashes($r);
$r = json_decode($r);
```

为了确保转化的正确性，需要检查 json_last_error() 返回的值。在 PHP 手册的 http://php.net/manual/en/function.json-last-error.php 中，你可以找到 json_last_error 返回值的具体含义：

```
0 - JSON_ERROR_NONE
1 - JSON_ERROR_DEPTH
2 - JSON_ERROR_STATE_MISMATCH
3 - JSON_ERROR_CTRL_CHAR
4 - JSON_ERROR_SYNTAX
5 - JSON_ERROR_UTF8
```

注意

json_last_error 在 PHP 5.3 及更高版本上才支持。如果你运行老版本的 PHP，需要移除检查 json_last_error 的代码，否则就会报错。

如果有错误，你应当完全退出应用程序。最后，该函数返回 $r：

```
f(json_last_error() > 0){
   die("invalid json");
}
return($r);
```

完整的函数如下所示：

```
function formatResults($r){
   $r = stripcslashes($r);
   $r = json_decode($r);
   if(json_last_error() > 0){
      die("invalid json");
   }
   return($r);
}
```

保存测试结果

下面，我们来完成 saveLog() 函数。显而易见，它接收一个用于指定我们日志文件路径

的字符串参数。正如前面章节提到的，首先要检查文件是否存在，如果文件不存在，就创建并格式化一个新的日志文件，当前的存根函数 formatNewLog() 就是用来完成这一任务的。

```
function saveLog($obj, $file){
   if(!file_exists($file)){
      formatNewLog($file);
   }
}
```

一些用户代理的字符串包含逗号。因为我们使用逗号作为我们平面文件中分割域的符号，因此你需要从用户代理字符串中去除逗号。我们将这一功能抽象到 cleanCommas() 函数中，现在我们只使用存根的方式调用它：

```
$obj->useragent = cleanCommas($obj->useragent);
```

接下来，你需要构造一个逗号分割的字符串，连接所有的属性值，然后传递给共享文件 fileio.php 中的 appendToFile() 函数。出于数据多维性的考虑，也可以为该文件假定一个 IP 地址。添加这些数据，你就可以对信息进行推断并使用各种有趣的方法对结果进行分类。例如，将 IP 地址转化为地理信息，然后将结果按区域排序，或按 ISP 排序。

```
$newLine = $_SERVER["REMOTE_ADDR"] . "," . $obj->id .",". $obj->startTime . "," . $obj->stopTime
. "," . $obj->runtime . "," . $obj->url . "," . $obj->useragent;
appendToFile($newLine, $file);
```

完整的函数如下所示：

```
function saveLog($obj, $file){
   if(!file_exists($file)){
      formatNewLog($file);
   }
   $obj->useragent = cleanCommas($obj->useragent);
   $newLine = $_SERVER["REMOTE_ADDR"] . "," . $obj->id .",". $obj->startTime . "," . $obj-
>stopTime . "," . $obj->runtime . "," . $obj->url . "," . $obj->useragent;
   appendToFile($newLine, $file);
}
```

让我们快速地实现存根函数 cleanCommas()。该函数接收一个 $data 参数。你需要通过一个小技巧来调用 explode() 函数——PHP 中等效于 split() 功能的函数，以定义好的分割符（我们这里用逗号）来对字符串进行分割，返回分割后的字符串元素组成的数组。将这个数组传递给 implode() 函数，该函数是 PHP 中相当于 join() 功能的函数，该函数接收一个数组作为它的参数，将数组中的所有元素连接成一个单独的字符串。通过结合使用两个函数，我们创建了一个单行的“查找 - 替换”功能。

```
function cleanCommas($data){
   return implode("", explode(",", $data));
}
```

注意

让我们揣摩一下“查找 - 替换”功能是如何工作的。使用分隔符将字符串分割成一个个分隔符实例。因此，如果我们将字符串“the quick brown fox jumps over the lazy dog”使

用“the”分隔符进行分割，它会变成如下数组：

```
["quick", "brown", "fox", "jumps", "over", "lazy", "dog"]
```

如果我们对该数组应用 impode 函数，传递“a”字符串用于将数组连接起来，我们得到的结果如下：

```
"a quick brown fox jumps over a lazy dog"
```

最后，我们快速地实现 formatNewLog() 函数。该函数会将所有的文件头信息作为新的日志文件的第一行，写入一个新的日志文件中：

```
function formatNewLog($file){
   $headerline = "IP, TestID, StartTime, StopTime, RunTime, URL, UserAgent";
   appendToFile($headerline, $file);
}
```

完整的 savePerfData.php 文件应如下所示：

```
<?php
require("util/fileio.php");

$logfile = "log/runtimeperf_results.txt";
$benchmarkResults = formatResults($_POST["data"]);

saveLog($benchmarkResults, $logfile);

function formatResults($r){
   $r = stripcslashes($r);
   $r = json_decode($r);
   if(json_last_error() > 0){
      die("invalid json");
   }
   return($r);
}

function formatNewLog($file){
   $headerline = "IP, TestID, StartTime, StopTime, RunTime, URL, UserAgent";
   appendToFile($headerline, $file);
}

function saveLog($obj, $file){
   if(!file_exists($file)){
      formatNewLog($file);
   }
$obj->useragent = cleanCommas($obj->useragent);
$newLine = $_SERVER["REMOTE_ADDR"] . "," . $obj->id .",". $obj->startTime . "," . $obj->stopTime
. "," . $obj->runtime . "," . $obj->url . "," . $obj->useragent;
   appendToFile($newLine, $file);
}

function cleanCommas($data){
   return implode("", explode(",", $data));
}

?>
```

4.4 一个示例页

太棒了！现在你可以使用我们的库来制作一个快速页面，以获取一些数据了。创建一个 HTML 页面的骨架，定义它的 title 属性为“perfLogger Example”，然后在页面的 head 节中链接 perfLogger.js。然后，同样在页面的 head 节中，加入另一个脚本标签，调用 perfLogger. startTimeLogging() 函数。为该函数传递 ID 参数，其值为“page_render”，一个描述参数“timing page render”，以及 true 参数值，指出结果需要在页面上显示，最后还有一个 true 参数指出需要将结果以日志的形式记录到服务器端：

```
<!DOCTYPE html>
<html>
<head>
   <title>perfLogger Example</title>
   <script src="perfLogger.js"></script>
   <script>
   perfLogger.startTimeLogging("page_render", "timing page render", true, true)
   </script>
</head>
<body>
<script>
perfLogger.stopTimeLogging("page_render")
</script>
</body>
</html>
```

把这些函数调用都放在 head 节中，目的是在页面开始渲染可见部分之前就进行计算。在 body 节，我们添加一段新的脚本标签，在该标签中调用 perfLogger.stopTimeLogging("page_render")。

将这个文件保存为 perfLogger_example.html。如果使用浏览器查看该文件，则看到的结果如图 4-6 所示。

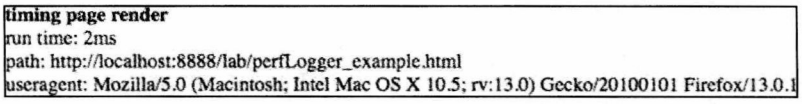


图 4-6 perfLogger_example.html 文件截屏

runtimeperf_results.txt 文件的内容如下所示：

```
IP, TestID, StartTime, StopTime, RunTime, URL, UserAgent
71.225.152.145,page_render,2012-06-19T23:52:52.448Z,2012-06-19T23:52:52.448Z,0,http://tom-
barker.com/lab/perfLogger_example.html,Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8)
AppleWebKit/536.5 (KHTML like Gecko) Chrome/19.0.1084.52 Safari/536.5
71.225.152.145,page_render,2012-06-19T23:52:52.452Z,2012-06-19T23:52:52.452Z,0,http://tom-
barker.com/lab/perfLogger_example.html,Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8)
AppleWebKit/536.5 (KHTML like Gecko) Chrome/19.0.1084.52 Safari/536.5
```

注意

依赖于浏览器，你可能会得到 0 毫秒的测试结果，因为页面上可能什么内容都没有。真正的测试都会有内容的。或者你创建你自己的测试套件来进行基准测试，就像 SunSpider（http://www.webkit.org/perf/sunspider/sunspider.html）那样。

我们会将这个页面归入产品集中，来为所有访问该页面的用户收集度量值。你可以在如下地址找到这一页面：http://tom-barker.com/lab/perfLogger_example.html。

4.5 为测试结果绘制图表

现在，既然我们已经获取了一些数据，接下来尝试对这些数据进行可视化处理。我们将会看到依赖用户代理的 page_render 测试结果的运行时间。创建一个新的 R 文档，命名为 runtimePerformance.R，然后创建如下变量：dataDirectory，用于保存日志目录的路径；chartDirectory，用于保存绘制图表文件的目录路径；testname，用于保存绘制图表的测试名称。

```
dataDirectory <- "/Applications/MAMP/htdocs/lab/log/"
chartDirectory <- "/Applications/MAMP/htdocs/lab/charts/"
testname = "page_render"
```

读入日志文件，并将其以数据帧的方式保存在 perflogs 变量中，然后创建图表文件路径变量 perfchart。

```
perflogs <- read.table(paste(dataDirectory, "runtimeperf_results.txt", sep=""), header=TRUE,
sep=",")
perfchart <- paste(chartDirectory, "runtime_",testname, ".pdf", sep="")
```

接下来，你要做一点分析了。现在，你仅仅有了 page_render 的测试结果的日志，而最终当你运行不同的测试之后，你会有不同名称的结果，因此只能使用这唯一的结果，列名为 TestID，其值为“page_render”。

```
pagerender <- perflogs[perflogs$TestID == "page_render",]
```

然后创建一个新的数据帧，其中只包含我们想要绘制图表的列，即 UserAgent 列和 RunTime 列。

```
df <- data.frame(pagerender$UserAgent, pagerender$RunTime)
```

然后，你要使用 R 中的 by() 函数。该函数将其他函数应用到数据帧上，并以传递的参数对结果进行分组。例如，我们根据 UserAgent 列进行分组，并将 mean() 函数应用到每一个元素上。

```
df <- by(df$pagerender.RunTime, df$pagerender.UserAgent, mean)
```

如果你在控制台再输入一遍 df，你会看到：

```
> df
df$pagerender.UserAgent
Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_8; en-us) AppleWebKit/533.21.1 (KHTML like Gecko)
Version/5.0.5 Safari/533.21.1

6.00000
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.5 (KHTML like Gecko)
Chrome/19.0.1084.52 Safari/536.5
```

```
20.75000
Mozilla/5.0 (Macintosh; Intel Mac OS X 10.5; rv:13.0) Gecko/20100101 Firefox/13.0.1

55.63158
Mozilla/5.0 (Macintosh; Intel Mac OS X 10.5; rv:13.0) Gecko/20100101 Firefox/13.0

144.00000
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; .NET CLR
3.0.4506.2152; .NET CLR 3.5.30729)

511.00000
```

现在该数据帧中保存了根据原始用户代理进行分组的平均运行时间值。这也是使用用户代理运行每个测试所得到的运行时间结果的算术平均值。

最后，你需要对数据帧进行排序，并生成一个柱状图。

```
df <- df[order(df)]
pdf(perfchart, width=10, height=10)
opar <- par(no.readonly=TRUE)
   par(las=1, mar=c(10,10,10,10))
   barplot(df, horiz=TRUE, main="Page Render Runtime Performance in Milliseconds\nBy User
Agent")
par(opar)
dev.off()
```

图 4-7 演示了结果图表。

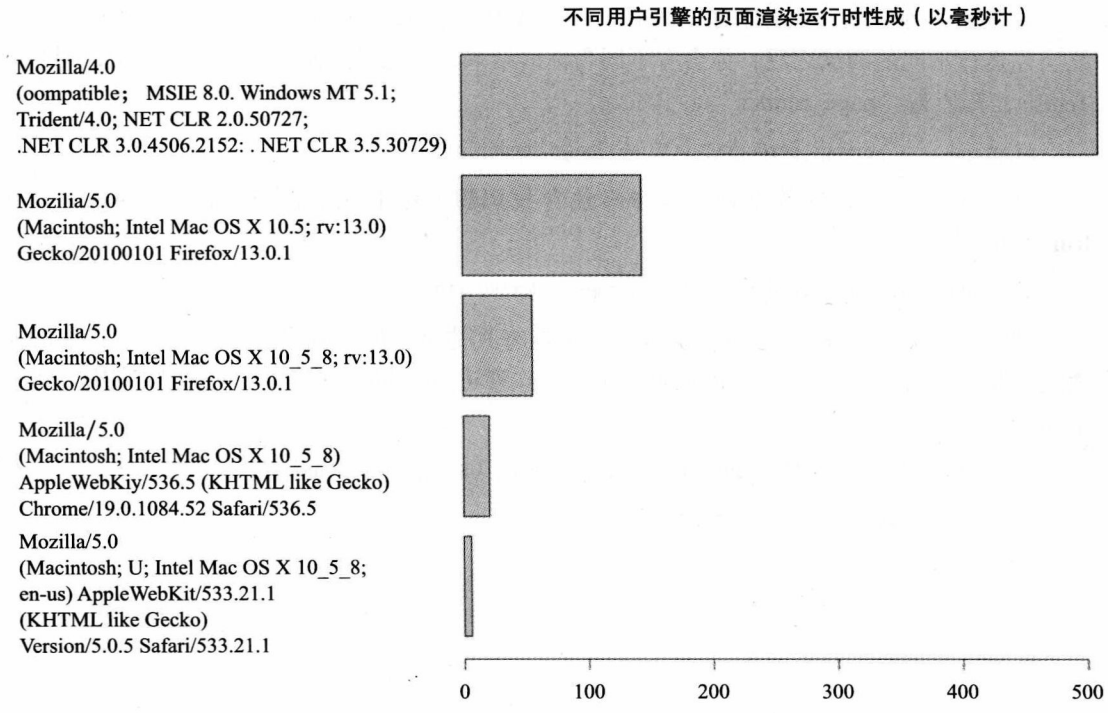


图 4-7 使用用户代理运行页面渲染测试的毫秒级结果

完整的 R 脚本如下所示：

```
dataDirectory <- "/Applications/MAMP/htdocs/lab/log/"
chartDirectory <- "/Applications/MAMP/htdocs/lab/charts/"
testname = "page_render"

perflogs <- read.table(paste(dataDirectory, "runtimeperf_results.txt", sep=""), header=TRUE,
sep=",")
perfchart <- paste(chartDirectory, "runtime_",testname, ".pdf", sep="")

pagerender <- perflogs[perflogs$TestID == "page_render",]
df <- data.frame(pagerender$UserAgent, pagerender$RunTime)
df <- by(df$pagerender.RunTime, df$pagerender.UserAgent, mean)
df <- df[order(df)]

pdf(perfchart, width=10, height=10)
opar <- par(no.readonly=TRUE)
   par(las=1, mar=c(10,10,10,10))
   barplot(df, horiz=TRUE)
par(opar)
dev.off()
```

4.6 开源

如同 WPTRunner 一样，我已经将 perfLogger 的源代码放置到 Github 上，地址为 https://github.com/tomjbarker/perfLogger。

4.7 小结

在本章中，我们创建了一个工具，用于收集特定代码块的运行时间，或者通过多次迭代运行代码，整合多次运行时间的方法，来对传入函数进行基准测试。你在我们的库中创建了一个服务器端 PHP 脚本，利用该脚本可发送测试的结果。最后创建了一个 R 脚本，用于提取测试结果，并对测试结果进行图表绘制。

接下来的章节中，我们将会对几个性能最佳实践进行深入探讨。为了描绘这些实践对于性能来讲是最优的，我们会使用本章开发的这些工具，并运用实际的数据以及对大规模应用测试结果的可视化，来支持我们关于这些最佳实践的观点。

第 5 章首先展望浏览器性能的未来发展，探讨将要发布的 Web 性能标准。

第 5 章

展望未来，性能的标准化

第 4 章，我们构建了一个 JavaScript 库，用于跟踪特定代码并对函数进行基准测试。这是一个非常棒的工具，正如你会在接下来的章节中看到的，使用这个工具，你可以收集时间度量值，甚至依据编码风格推断性能提升的程度。

到目前为止，我们已经知道了如何构建自己的工具，或者使用现存的工具来收集需要进行性能测量的数据。但是现在，我们要看看 W3C 在制定跟踪浏览器性能指标的标准方面已经做了哪些工作。

需要注意的是，有些特性只有在当前版本的浏览器上才能支持；而有些特性仅适用于测试以及预测试版本的浏览器上。基于这些原因，本章所有相关的例子中，我会明确地使用发布版本的浏览器。也正是由于这个原因，我会基于不同的浏览器对同一件事情显示屏幕截图，用以说明浏览器对这些特性支持程度的不同。

5.1 W3C 的 Web 性能工作组

2010 年年底，W3C 创建了一个新的工作组，即 Web 性能工作组。该工作组的任务，正如它在网页上描述的那样，就是提供方法来衡量使用了用户代理和 API 之后的应用程序的性能。这也就意味着，从实践的意义上来讲，该工作组已经开发了一套浏览器，现在或者将来会将这些可以获得关键的性能度量值的 API 公开给 JavaScript。

这些 API 被应用在一个新的性能对象上，该对象是原生 window 对象的一部分。

```
>>window.performance
```

5.2 性能对象

如果你在 JavaScript 控制台上输入 window.performance，就能看到它返回一个 Performance 类型的对象，以及该对象中暴露出来的几个对象和方法。图 5-1 显示了在 Chrome 20 测试版

上的 Performance 对象结构。

▸ memory	MemoryInfo { jsHeapSizeLimit=0, totalJSHeapSize=0, … }
jsHeapSizeLimit	0
totalJSHeapSize	0
usedJSHeapSize	0
▸ navigation	PerformanceNavigation { redirectCount=0, type=0, … }
TYPE_BACK_FORWARD	2
TYPE_NAVIGATE	0
TYPE_RELOAD	1
TYPE_RESERVED	255
redirectCount	0
type	0
▸ timing	PerformanceTiming { fetchStart=1340762919512, … }
connectEnd	1340762919514
connectStart	1340762919514
domComplete	1340762920145
domContentLoadedEventEnd	1340762919698
domContentLoadedEventStart	1340762919698
domInteractive	1340762919698
domLoading	1340762919582
domainLookupEnd	1340762919512
domainLookupStart	1340762919512
fetchStart	1340762919512
loadEventEnd	1340762920150
loadEventStart	1340762920145
navigationStart	1340762919512
redirectEnd	0
redirectStart	0
requestStart	1340762919514
responseEnd	1340762919696
responseStart	1340762919579
secureConnectionStart	0
unloadEventEnd	0
unloadEventStart	0
webkitNow	webkitNow()

图 5-1 在 Chrome 20 测试版上的 window.performance 对象

如果我们在 Chrome 上输入 window.performance，就会看到，该对象支持一个 navigation 对象，即 PerformanceNavigation，以及一个 timing 对象（即 PerformanceTiming）。Chrome 也支持内存对象，甚至也支持 webkitNow——针对特定浏览器版本的高分辨率时间（High Resolution Time）。本章末尾我将讨论高分辨率时间。

我们来看看如何在性能跟踪当中使用上述每一个对象。

5.2.1 性能定时

Timing 对象有如下一些属性，所有的属性都包含了当这些定时事件发生时的毫秒级快照，这非常像我们第 4 章的 startTimeLogging() 函数。图 5-2 演示了每一个定时度量按照先后顺序的流程图。

navigationStart：这是开始浏览（navigation）时的时间快照，或者是当浏览器卸载前一个页面，或者不用执行卸载，开始获取新内容时的时间快照。这一过程或者包含了 unloadEventStart 数据，或者包含了 fetchStart 数据。如果我们想跟踪端到端的时间情况，我

们往往会启用这个值。

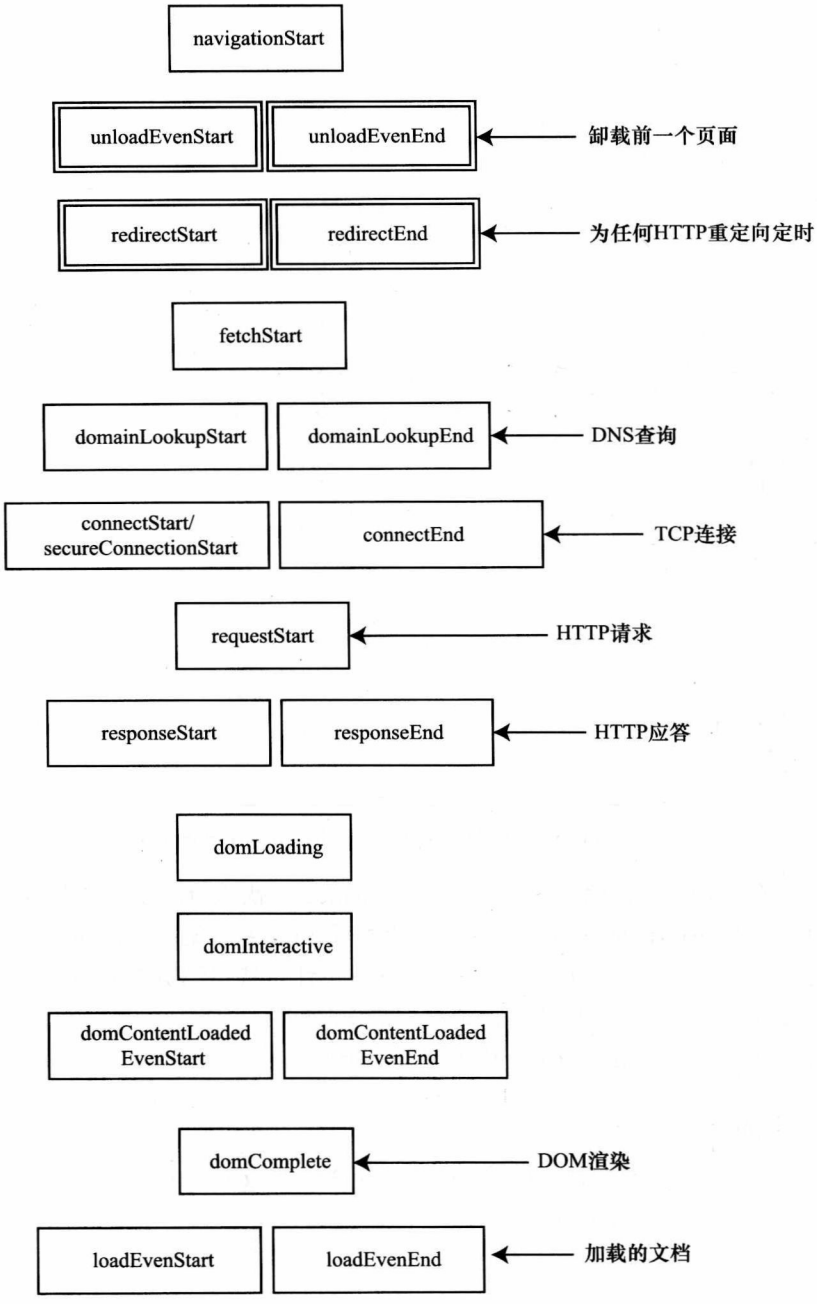


图 5-2 性能指标的顺序图

unloadEventStart：这是当浏览器开始卸载同一域内的前一页数据时的时间快照。

unloadEventEnd：这是当浏览器完全卸载前一个页面时的时间快照。

redirectStart：这是浏览器开始执行 HTTP 重定向时的时间快照。

redirectEnd：这是当浏览器完成全部 HTTP 重定向操作之后的时间快照。

要计算执行 HTTP 重定向所消耗的完整时间，只需要用 redirectEnd 减去 redirectStart 就可以了。

```
<script>
var http_redirect_time = performance.timing.redirectEnd - performance.timing.redirectStart;
</script>
```

fetchStart：这是浏览器第一次检查用于存储请求资源的缓存时的时间快照。

要计算缓存加载的全部时间，用 domainLookupStart 减去 fetchStart 就可以了。

```
<script>
var cache_time = performance.timing.domainLookupStart - performance.timing.fetchStart;
</script>
```

domainLookupStart：这是当浏览器开始执行 DNS 查询以获得所需内容时的时间快照。

domainLookupEnd：这是当浏览器完成 DNS 查询以获得所需内容时的时间快照。

要计算执行 DNS 查询的完整时间，用 domainLookupEnd 减去 domainLookupStart 就可以了。

```
<script>
var cache_time = performance.timing.domainLookupStart - performance.timing.fetchStart;
</script>
```

connectStart：这是当浏览器与当前页的远端服务器开始建立 TCP 连接时的时间快照。

secureConnectionStart：当页面借助 HTTPS 隧道加载时，该属性捕获 HTTPS 通信开始时的时间快照。

connectEnd：这是当浏览器与当前页的远端服务器完成 TCP 连接时的时间快照。

要计算建立 TCP 连接所花费的总时间，用 connectEnd 减去 connectStart 就可以了。

```
<script>
var tcp_connection_time = performance.timing.connectEnd - performance.timing.connectStart;
</script>
```

requestStart：这是当浏览器发送 HTTP 请求时的时间快照。

responseStart：这是当浏览器第一次对服务器的响应进行注册时的时间快照。

responseEnd：这是当浏览器完成接收服务器发出的响应时的时间快照。

要计算完成一次 HTTP 往返所花费的总时间（包括建立 HTTP 请求的时间），可以用 responseEnd 减去 connectStart。

```
<script>
var page_render_time = performance.timing.domComplete - performance.timing.domLoading;
</script>
```

domLoading：这是当文档开始加载时的时间快照。

domComplete：这是当文档加载完成时的时间快照。

要计算页面渲染所花费的时间，只需要用 domComplete 减去 domLoading 就可以了。

```
<script>
var page_render_time = performance.timing.domComplete - performance.timing.domLoading;
</script>
```

要计算加载页面所花费的时间，包括从第一次请求开始到页面完全加载完成，用 domComplete 减去 navigationStart 就可以了。

```
<script>
var full_load_time =  performance.timing.domComplete - performance.timing.navigationStart
</script>
```

domContentLoadedEventEnd：这是当 DOMContentLoaded 事件第一次执行完时的时间快照。

domContentLoadedEventStart：这是当 DOMContentLoaded 事件第一次开始执行时的时间快照。

DOMContentLoaded 事件是在浏览器完成对文档的解析时触发的。关于这个事件更多的信息，可以看看 W3C 文档中关于文档解析即将完成时的相关执行步骤，地址是 http://www.w3.org/TR/html5/the-end.html。

domInteractive：当文档对象的 readyState 属性设置为 interactive 时触发该事件，interactive 是指用户可以与页面进行交互了。

loadEventEnd：当页面加载事件完成时触发该事件。

loadEventStart：当页面加载事件开始时触发该事件。

我们用新的性能数据资源来更新 perfLogger。我们会添加一些只读的公有属性，用以计算想要记录的数值。我们还会修改测试结果对象（Test Result object）的原型，目的是使我们发往服务器的每一个结果都会自动将性能度量值嵌入到对象当中。

让我们开始吧！

5.2.2 用 perfLogger 整合性能对象

首先，你需要在 perfLogger 的自执行函数中创建私有变量。创建变量用于保存感知时间（perceived time）、重定向时间、缓存时间、DNS 查询时间、TCP 连接时间、总往返时间，以及页面渲染时间。

```
var perfLogger = function(){
     var serverLogURL = "savePerfData.php",
     loggerPool = [];
     if(window.performance){
     var _pTime = Date.now() - performance.timing.navigationStart || 0,
     _redirTime = performance.timing.redirectEnd - performance.timing.redirectStart || 0,
     _cacheTime = performance.timing.domainLookupStart - performance.timing.fetchStart || 0,
     _dnsTime = performance.timing.domainLookupEnd - performance.timing.domainLookupStart || 0,
     _tcpTime = performance.timing.connectEnd - performance.timing.connectStart || 0,
     _roundtripTime = performance.timing.responseEnd - performance.timing.connectStart || 0,
     _renderTime = performance.timing.domComplete - performance.timing.domLoading || 0;
    }
}
```

首先，这段代码将我们的变量赋值语句封装在一个 if 语句中，这样确保只有在当前浏览器支持 window.performance 调用的情况下，这些赋值语句才会起作用。要注意，我们在赋值语句中使用了“短路求值”技术。这一技术在赋值语句中使用一个逻辑运算符，在这里，我们用的是逻辑或（OR）。如果逻辑语句中的第一个值是不可用的、空值或未定义的，那么第二个值就会作为赋值语句的值被赋给变量。

接下来，仍然在自执行函数中，你需要明确地创建一个 TestResults 构造函数。记住，我们在第 4 章构造了 TestResults 对象，但是我们最后并没有使用它，相反，我们使用 loggerPool 来保存通用对象。现在，我们就要使用 TestResults 了，并且会利用原型继承来确保每一个 TestResults 对象都会内置我们的新性能指标。

首先来创建 TestResults 构造函数。

```
function TestResults(){};
```

然后添加一个属性到 TestResults 的原型中，用以保存我们所有的 window.performance 的度量。

```
TestResults.prototype.perceivedTime = _pTime;
TestResults.prototype.redirectTime = _redirTime;
TestResults.prototype.cacheTime = _cacheTime;
TestResults.prototype.dnsLookupTime = _dnsTime;
TestResults.prototype.tcpConnectionTime = _tcpTime;
TestResults.prototype.roundTripTime = _roundtripTime;
TestResults.prototype.pageRenderTime = _renderTime;
```

太棒了！现在，我们开始编辑公用方法 startTimeLogging()。该函数的第一行就是将一个空对象赋值给 loggerPool。

```
loggerPool[id] = {};
```

修改这一赋值语句，实例化一个 TestResults 对象。

```
loggerPool[id] = new TestResults();
```

到这里，如果你使用 console.log 查看 TestResults 对象，就会看到如图 5-3 所示的结果。

你会看到，在我们创建的每个 TestResults 对象中，都包含了 cacheTime、dnsLookupTime、pageRenderTime、perceivedTime、redirectTime、roundTripTime，以及 tcpConnectionTime 这些属性。你还能看到这些属性也存在于原型当中。

这一点很重要，因为如果你使用 console.log 显示 logToServer() 中的序列化对象，就会看到，对象之外的属性并不是可序列化的。这是因为 JSON.stringify 并没有对对象内未定义的数值或者函数进行序列化处理。

这并不是什么问题。要解决它，你只需要定义一个小的私有函数，将两个对象连接起来即可。所以，回到自执行函数开始的地方，在那里，添加一个新函数 jsonConcat()，让其可接收两个对象作为参数。

```
function jsonConcat(object1, object2) {}
```

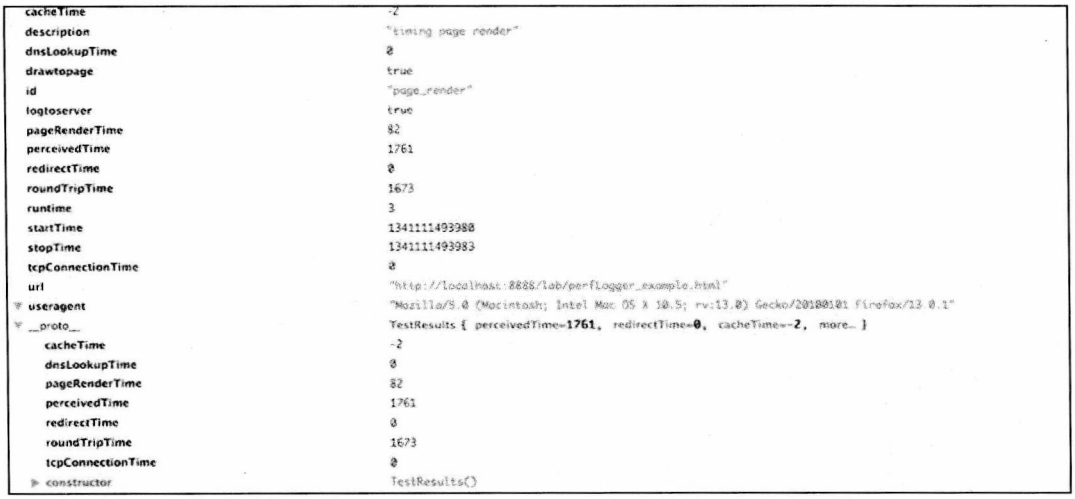


图 5-3 TestResults 对象

接下来，循环访问第二个对象中的每一个属性，并将属性添加到第一个对象中。最后，返回第一个对象。

```
for (var key in object2) {
object1[key] = object2[key];
}
return object1;
```

注意，如果两个对象有同样的属性，则这样做会改写第一个对象中对应属性（两个对象共有属性）的值。

最终的函数如下所示：

```
function jsonConcat(object1, object2) {
for (var key in object2) {
object1[key] = object2[key];
     }
     return object1;
}
```

现在，要让这些新加功能开始工作了，回到 logToServer()。回忆一下，在这个函数的开头，我们用下面的方法序列化了 TestResults 对象：

```
var params = "data=" + (JSON.stringify(loggerPool[id]));
```

修改这条语句，将 TestResults 对象及其原型传递给 jsonConcat() 函数，然后将 jsonConcat() 函数的返回对象传递给 JSON.stringify。

```
var params = "data=" + JSON.stringify(jsonConcat(loggerPool[id],TestResults.prototype));
```

如果用 console.log 输出 params 变量，会看到如下结果：

```
data={"id":"page_render","startTime":1341152573075,"description":"timing page render","drawtopag
e":true,"logtoserver":true,"stopTime":1341152573077,"runtime":2,"url":"http://localhost:8888/
```

```
lab/perfLogger_example.html","useragent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.5; rv:13.0)
Gecko/20100101 Firefox/13.0.1","perceivedTime":78,"redirectTime":0,"cacheTime":-2,"dnsLookupTime
":0,"tcpConnectionTime":0,"roundTripTime":2,"pageRenderTime":72}
```

接下来，你要将这些私有的性能变量通过公开的方法暴露出来，其目的是在不运行任何测试的情况下，通过 perfLogger 的命名空间将这些变量暴露出来。如果你没有在对象一级将这些变量公开出来，就需要创建一个测试，然后从测试对象中将这些变量取（pull）出来，回忆一下我们将这些性能对象数值添加到每一个测试对象原型时用到的方法。

```
//expose derived performance data
perceivedTime: function(){
return _pTime;
},
redirectTime: function(){
     _redirTime;
},
cacheTime: function(){
     return _cacheTime;
},
dnsLookupTime: function(){
     return _dnsTime;
},
tcpConnectionTime: function(){
     return _tcpTime;
},
roundTripTime: function(){
     return _roundtripTime;
},
pageRenderTime: function(){
     return _renderTime;
}
```

太好了！从 perfLogger 对象中暴露数据的公开方法如下所示：

```
>>> perfLogger.perceivedTime()
78
```

到目前为止，修改后的 perfLogger.js 如下所示：

```
var perfLogger = function(){
var serverLogURL = "savePerfData.php",
     loggerPool = [],
     _pTime = Date.now() - performance.timing.navigationStart || 0,
     _redirTime = performance.timing.redirectEnd - performance.timing.redirectStart || 0,
     _cacheTime = performance.timing.domainLookupStart - performance.timing.fetchStart || 0,
     _dnsTime = performance.timing.domainLookupEnd - performance.timing.domainLookupStart || 0,
     _tcpTime = performance.timing.connectEnd - performance.timing.connectStart || 0,
     _roundtripTime = performance.timing.responseEnd - performance.timing.connectStart || 0,
     _renderTime = Date.now() - performance.timing.domLoading || 0;

function TestResults(){};
TestResults.prototype.perceivedTime = _pTime;
TestResults.prototype.redirectTime = _redirTime;
TestResults.prototype.cacheTime = _cacheTime;
TestResults.prototype.dnsLookupTime = _dnsTime;
TestResults.prototype.tcpConnectionTime = _tcpTime;
```

```
TestResults.prototype.roundTripTime = _roundtripTime;
TestResults.prototype.pageRenderTime = _renderTime;

function jsonConcat(object1, object2) {
for (var key in object2) {
     object1[key] = object2[key];
}
return object1;
}

function calculateResults(id){
     loggerPool[id].runtime = loggerPool[id].stopTime - loggerPool[id].startTime;
}

function setResultsMetaData(id){
     loggerPool[id].url = window.location.href;
     loggerPool[id].useragent = navigator.userAgent;
}

function drawToDebugScreen(id){
     var debug = document.getElementById("debug")
     var output = formatDebugInfo(id)
     if(!debug){
          var divTag = document.createElement("div");
          divTag.id = "debug";
          divTag.innerHTML = output
          document.body.appendChild(divTag);
     }else{
          debug.innerHTML += output
     }
}

function logToServer(id){
     var params = "data=" + JSON.stringify(jsonConcat(loggerPool[id],TestResults.prototype));
     var xhr = new XMLHttpRequest();
     xhr.open("POST", serverLogURL, true);
     xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
     xhr.setRequestHeader("Content-length", params.length);
     xhr.setRequestHeader("Connection", "close");
     xhr.onreadystatechange = function()
      {
      if (xhr.readyState==4 && xhr.status==200)
        {}
      };
     xhr.send(params);
}

function formatDebugInfo(id){
     var debuginfo = "<p><strong>" + loggerPool[id].description + "</strong><br/>";
     if(loggerPool[id].avgRunTime){
          debuginfo += "average run time: " + loggerPool[id].avgRunTime + "ms<br/>";
     }else{
          debuginfo += "run time: " + loggerPool[id].runtime + "ms<br/>";
     }
     debuginfo += "path: " + loggerPool[id].url + "<br/>";
     debuginfo += "useragent: " +  loggerPool[id].useragent + "<br/>";
```

```
    debuginfo += "</p>";
    return debuginfo
}

return {

startTimeLogging: function(id, descr,drawToPage,logToServer){
loggerPool[id] = new TestResults();
    loggerPool[id].id = id;
    loggerPool[id].startTime =  Date.now();
    loggerPool[id].description = descr;
    loggerPool[id].drawtopage = drawToPage;
    loggerPool[id].logtoserver = logToServer
},

stopTimeLogging: function(id){
loggerPool[id].stopTime =  Date.now();
    calculateResults(id);
    setResultsMetaData(id);
    if(loggerPool[id].drawtopage){
        drawToDebugScreen(id);
    }
    if(loggerPool[id].logtoserver){
        logToServer(id);
    }
},

logBenchmark: function(id, timestoIterate, func, debug, log){
    var timeSum = 0;
    for(var x = 0; x < timestoIterate; x++){
        perfLogger.startTimeLogging(id, "benchmarking "+ func, false, false);
        func();
        perfLogger.stopTimeLogging(id)
        timeSum += loggerPool[id].runtime
    }
    loggerPool[id].avgRunTime = timeSum/timestoIterate
    if(debug){
        drawToDebugScreen(id)
    }
    if(log){
        logToServer(id)
    }
},

//expose derived performance data
perceivedTime: function(){
return _pTime;
},
redirectTime: function(){
    _redirTime;
},
cacheTime: function(){
    return _cacheTime;
},
dnsLookupTime: function(){
    return _dnsTime;
```

```
},
tcpConnectionTime: function(){
    return _tcpTime;
},
roundTripTime: function(){
    return _roundtripTime;
},
pageRenderTime: function(){
    return _renderTime;
}
}
}();
```

5.3 升级日志功能

接下来修改 savePerfData.php 文件。首先为 formatNewLog() 函数添加一个新的标头，以便让每一个新的日志文件都会包含针对我们新数据内容的列标头。

```
function formatNewLog($file){
    $headerline = "IP, TestID, StartTime, StopTime, RunTime, URL, UserAgent, PerceivedLoadTime,
PageRenderTime, RoundTripTime, TCPConnectionTime, DNSLookupTime, CacheTime, RedirectTime";
    appendToFile($headerline, $file);
}
```

然后修改 saveLog() 函数，让其包含另外一些我们现在传递进来的值：

```
function saveLog($obj, $file){
    if(!file_exists($file)){
        formatNewLog($file);
    }
    $obj->useragent = cleanCommas($obj->useragent);
    $newLine = $_SERVER["REMOTE_ADDR"] . "," . $obj->id .",". $obj->startTime . "," . $obj-
>stopTime . "," . $obj->runtime . "," . $obj->url . "," . $obj->useragent . “,” . $obj-
>perceivedTime . “,” . $obj->pageRenderTime . “,” . $obj->roundTripTime . “,” . $obj-
>tcpConnectionTime . “,” . $obj->dnsLookupTime . “,” . $obj->cacheTime . “,” . $obj-
>redirectTime;
    appendToFile($newLine, $file);
}
```

现在，修改后的日志文件如下所示：

```
IP, TestID, StartTime, StopTime, RunTime, URL, UserAgent, PerceivedLoadTime, PageRenderTime,
RoundTripTime, TCPConnectionTime, DNSLookupTime, CacheTime, RedirectTime
127.0.0.1,page_render,1.34116648339,1.3411664834,2,http://localhost:8888/lab/perfLogger_example.
html,Mozilla/5.0 (Macintosh; Intel Mac OS X 10.5; rv:13.0) Gecko/20100101 Firef
ox/13.0.1115,86,21,1,1,-4,0
127.0.0.1,page_render,345.173000009,345.331000019,0.158000009833,http://localhost:8888/lab/
perfLogger_example.html,Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML
like Gecko) Chrome/20.0.1132.43 Safari/536.11495,261,79,0,0,0,0
```

5.4 性能导航

现在，我们来看看性能导航对象。如果在控制台上输入 performance.navigation，就会

看到如图 5-4 所示的内容。

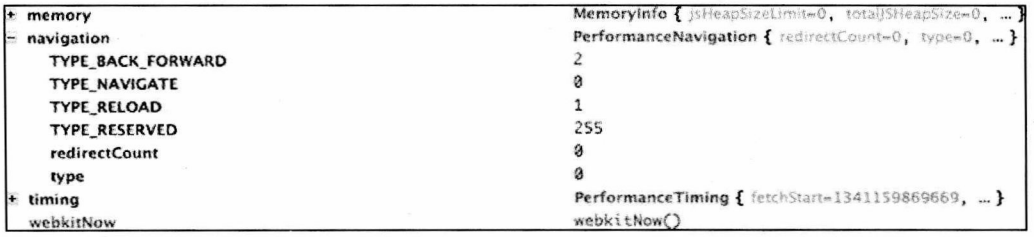


图 5-4 性能导航对象

注意，导航对象有两个只读属性：redirectCount 和 type。redirectCount 属性见名知意，表示浏览器访问当前页面，执行 HTTP 重定向的次数。

注意

HTTP 重定向非常重要，因为它会造成每一个重定向都必须完成一个完整的往返过程。执行 HTTP 重定向后，向 Web 服务器发送的原始请求会以 301 或 302 的返回码返回，同时附带有新地址的路径信息。这时，浏览器必须初始化一个新的 TCP 连接，然后向新地址发送一个新的请求。图 5-5 是一个单一 HTTP 重定向的序列图。注意 DNS 查询、TCP 握手以及 HTTP 请求如何重复以执行重定向操作。相对于单一的重定向，这样的重复执行使网络连接消耗的时间倍增。

我们可以通过如下方式访问 redirectCount 属性：

```
>>> performance.navigation.redirectCount
0
```

navigation 对象的另一个属性是 type。navigation.type 属性的值是下列 4 个常数之一：

TYPE_NAVIGATE：值为 0，表示当前页是通过点击一个链接、提交一个表单，或者直接在地址栏输入 URL，跳转或定位过来的。

TYPE_RELOAD：值为 1，表示当前页面是通过重载操作访问的。

TYPE_BACK_FORWARD：值为 2，表示当前页是通过访问浏览器历史记录，或点击浏览器的“后退”、“前进”按钮，或者通过程序的方式访问浏览器的历史对象，跳转或定位过来的。

TYPE_RESERVED：值为 255，所有上述情况之外的页面访问方式，都使用该值。

5.5 性能内存

内存对象是 Chrome 的一个特色，它允许我们在使用 Chrome 运行网页时，查看内存的使用情况。回想一下，在图 5-1 中，开始所有 memory 的属性的值都是 0，这是因为我们需要启用内存信息标记，才能充分利用这一功能。

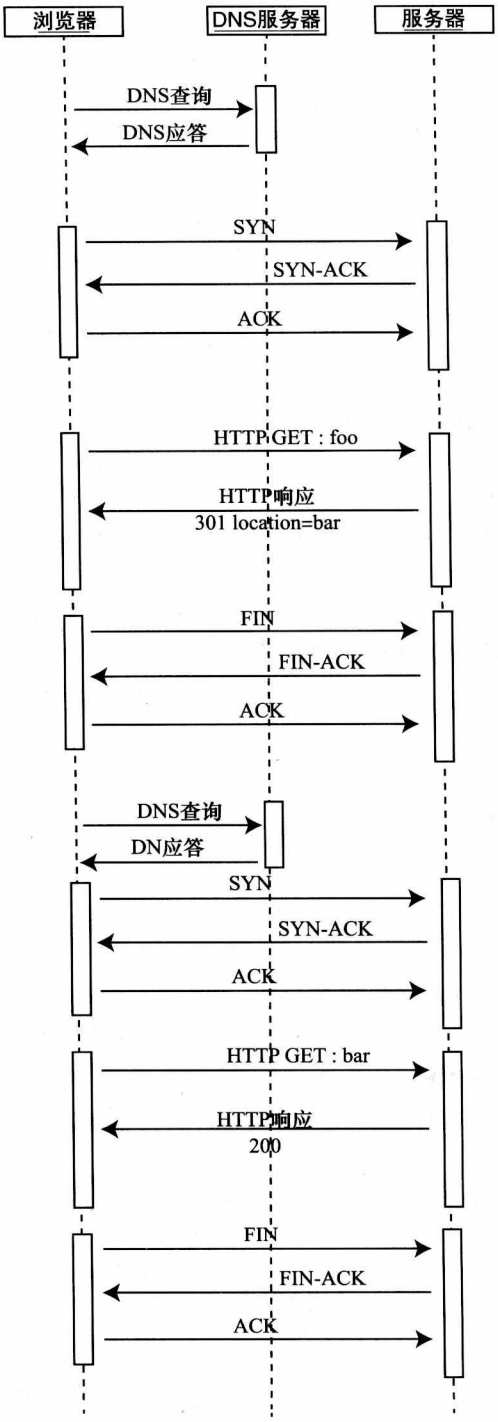

浏览器
DNS服务器
服务器
DNS查询
DNS应答
SYN
SYN-ACK
ACK
HTTP GET : foo
HTTP响应
301 location=bar
FIN
FIN-ACK
ACK
DNS查询
DN应答
SYN
SYN-ACK
ACK
HTTP GET : bar
HTTP响应
200
FIN
FIN-ACK
ACK

图 5-5 HTTP 重定向

要做到这一点，对平台还是有一定依赖性的，但关键是需要将“--enable-memory-info”命令行参数传递给 Chrome。要想在 Windows 平台上做到这一点，先右键单击 Chrome 图标，选择属性，然后在可执行文件路径的最后添加 --enable-memory-info 标记。这时，可执行文件看起来如下所示：

```
[path to exe]\chrome.exe --enable-memory-info
```

而对于 Mac 操作系统来讲，不需要修改快捷方式，只需要进入终端，用如下的方法调用 Chrome 就可以了：

```
/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --enable-memory-info
```

要了解更多的相关信息，可以访问 http://www.chromium.org/developers/how-tos/run-chromium-with-flags。

使用 --enable-memory-info 标记启动了 Chrome 之后，你就能够看到 memory 对象中的数据了，如图 5-6 所示。

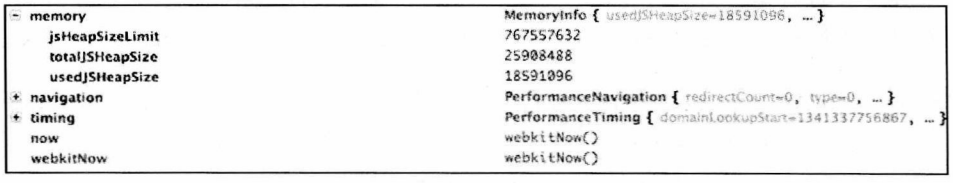


图 5-6 带有数据的性能内存对象

```
>>> performance.memory. jsHeapSizeLimit
767557632
```

作为参考，堆（heap）是解释器驻留在内存中的 JavaScript 对象的集合。在堆中，每一个对象都是互相关联的节点，通过属性相互连接，就像原型链或组合对象那样。运行在浏览器中的 JavaScript 通过对象引用（object reference）来访问堆中对象，如图 5-7 所示。

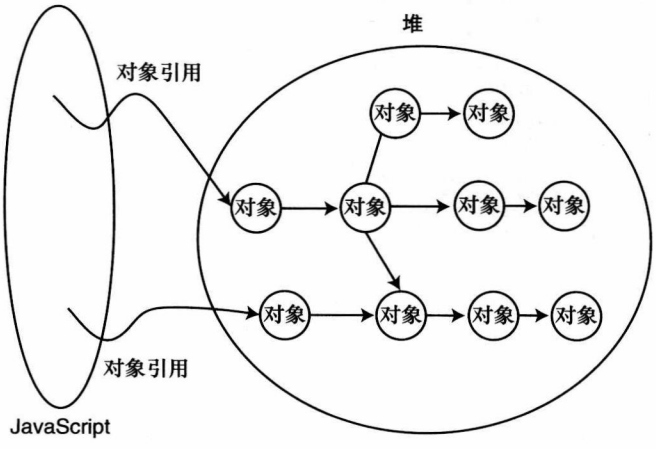


图 5-7 JavaScript 堆

当 JavaScript 销毁对象时，实际上销毁的是对象引用。当解释器看到堆中的某个对象没有对象引用了，它就会主动将该对象从堆中删除。这就是所谓的“垃圾回收”(garbage collection)。

图 5-6 中的 usedJsHeapSize 属性表示堆中当前所有正在使用的对象所占内存的总量。totalJsHeapSize 表示包括未被对象使用的空闲空间在内的堆的总大小。

为了获取这些属性，我们必须使用命令行参数的形式启动浏览器，因此，我们不可能也不能将这样的使用方法囊括进我们的库中，为了得到真实的数据，我们需要设置环境变量。相反，它是用来监测或在实验中用于性能分析的一个工具。

注意

通过性能分析我们可以监测内存使用情况。这对于检测内存泄漏（即自创建以来从未被释放的内存对象）非常有用。通常，当我们使用 JavaScript 编写程序，将事件句柄赋值给 DOM 对象并忘记去掉事件句柄时，就会发生这种情况。除了检测内存泄漏之外，性能分析有一个更加有趣的特性，就是它可以在应用程序运行过程中，随时优化内存的使用。我们应当灵活掌握对象的创建、销毁与复用，并且要时刻小心对象的作用域，避免在分析图中不断出现一系列的峰值曲线。比较好的分析图表显示的曲线是平稳上升的，最终趋于平缓。

Firefox 也可以对内存的使用情况进行检测。在浏览器地址栏中输入 about:memory，浏览器就会跳转到一个页面，这个页面会给出内存使用情况的概要信息，要想了解粒度更细的内存使用情况，可以在地址栏中输入 about:memory?verbose。图 5-8 显示了在 Firefox 中内存的细粒度使用情况。

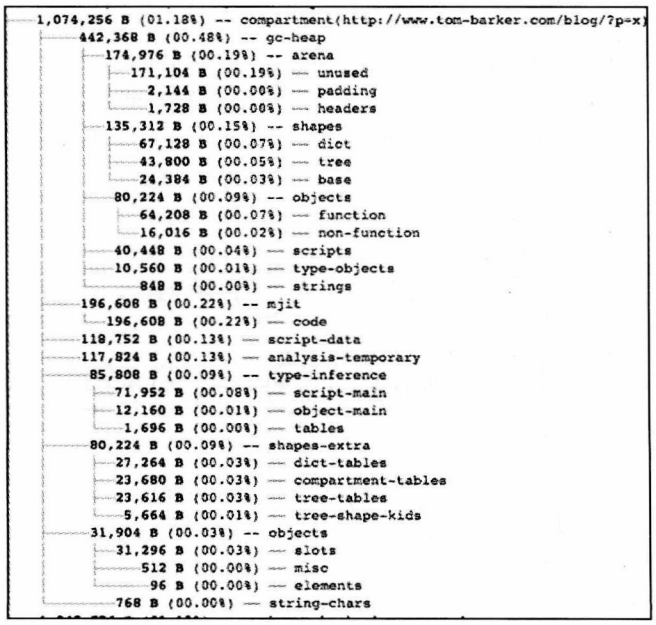


图 5-8 Firefox 内存窗口

5.6 高分辨率时间

接下来，我们看一个新的特性——高分辨率时间，用亚毫秒值记录时间。高分辨率时间对于获取频率低于 1 毫秒的定时数据非常有用。在后续的章节中，我们还会使用高分辨率时间的扩展特性来获取运行时性能指标。

针对性能对象的获取，Web 性能工作组（Web Performance Working Group）给出了一个公用函数 performance.now()。

正如前面章节描述的，当遇到计时代码执行的情况时，我们得到的结果往往是 0 毫秒。这是因为，这些结果值是从 Date() 对象那里返回的，而该对象的精度仅仅能够达到 1 毫秒。在 JavaScript 控制台中输入如下内容，查看 Date() 对象：

```
>>> new Date().getMilliseconds()
603

>>> Date.now()
1340722684122
```

由于 JavaScript 解释器和浏览器渲染引擎在性能方面的改进，我们的一些操作可能在 1 毫秒内就完成了。这就是我们测试的结果往往是 0 毫秒的原因。

对于这类测试，一个更不引人注意的危险是，Date 对象的值是基于系统时间（date）的，理论上讲，它在测试过程中是可以变化的，可能会导致结果出现偏差。很多事情都会在不经过我们输入的情况下改变系统时间，比如夏令时制，再比如企业系统时间的同步策略。所有这些都会改变我们的时间结果，甚至有可能会产生负数结果。

performance.now() 函数返回的结果是以几分之一毫秒来计数的。该函数与 navigation Start 事件也有关联，但是与系统时间无关，因此它不受系统时间改变的影响。Chrome 浏览器从 Chrome 20 开始，使用带有指定浏览器前缀的 webkitNow() 函数来支持高分辨率时间，如下所示：

```
Google Chrome 20
>>> performance.webkitNow()
290117.4699999974
```

Firefox 浏览器从 Firefox 15 Aurora 这一版开始支持高分辨率时间：

```
Firefox 15 aurora
>>> performance.now()
56491.447434
```

太好了，但是现在我们怎么使用它呢？怎么能够保证自己编写的代码一次能支持所有的浏览器呢？出于对 perfLogger 库的长远考虑，我们可以针对 performance.now() 进行编码，构造一个垫片（shim），在浏览器不支持的情况下，使用这个垫片作为功能回退。

注意

垫片是一个抽象层次的概念，用于拦截 API 调用或者拦截事件，要么改变 API 调用函数的签名，传递该调用以满足变更（updated）了的 API 签名，要么处理 API 本身的功

能。图 5-9 描绘了假定模块为了执行某些逻辑操作，在消息重新发送出去之前，正在对事件和消息的调度进行垫片拦截。图 5-10 描绘了模块正在重写一个函数或一个对象，同时传递消息，然后要么将数据传递给原始对象或函数，要么通过组合，调用原始对象或函数。

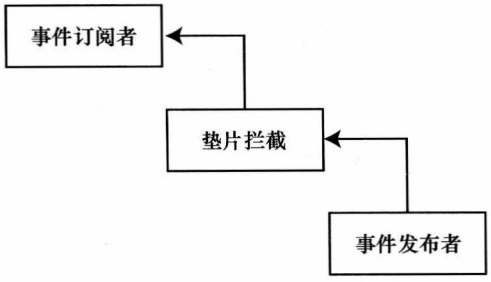


图 5-9 拦截

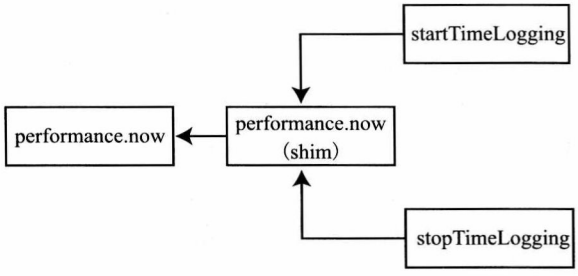


图 5-10 重载

在这种情况下，你会重写 performance.now() 函数调用，然后在调用继续向下传递之前或者完成其自身功能之前，添加一层功能。

在 perfLogger.js 文件中，在 perfLogger 的自执行函数之后，添加了一个函数声明，用于重写 performance.now 函数：

```
performance.now = (function() {
})();
```

该函数应该返回一个 performance.now 函数过程（如果支持），否则，代码通过潜在的特定于浏览器的实现进行迭代，如果没有找到可支持的函数过程，则默认返回旧的 Data() 功能。

```
performance.now = (function() {
 return performance.now       ||
          performance.webkitNow ||
        function() { return new Date().getTime(); };
})();
```

接下来，使用 performance.now() 来修改 perfLogger 的 startTimeLogging() 函数和 stopTime Logging() 函数：

```
now():

startTimeLogging: function(id, descr,drawToPage,logToServer){
loggerPool[id] = {};
     loggerPool[id].id = id;
     loggerPool[id].startTime =  performance.now(); // high resolution time support
     loggerPool[id].description = descr;
     loggerPool[id].drawtopage = drawToPage;
     loggerPool[id].logtoserver = logToServer
}

stopTimeLogging: function(id){
loggerPool[id].stopTime =  performance.now(); //high resolution time support
```

```
        calculateResults(id);
        setResultsMetaData(id);
        if(loggerPool[id].drawtopage){
            drawToDebugScreen(id);
        }
        if(loggerPool[id].logtoserver){
            logToServer(id);
        }
    }
```

现在，我们来看看在支持高分辨率时间的浏览器上，第 4 章提到的页面渲染基准测试结果是怎样的。图 5-11 显示了在不支持高分辨率时间的 Chrome 19 浏览器上的运行结果。图 5-12 显示了在 Firefox 15 Aurora 上的运行结果，图 5-13 显示了在 Chrome 20 上运行的结果，后两款浏览器都支持高分辨率时间。

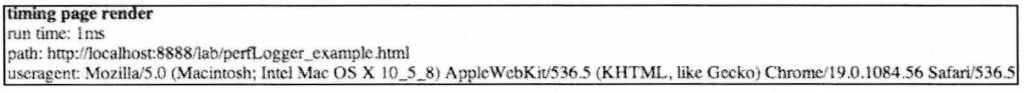


图 5-11 Chorme 19 默认的 Date 对象

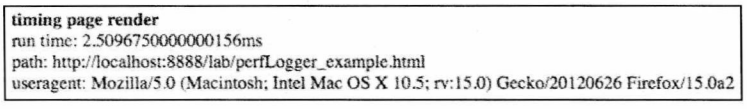


图 5-12 Firefox 15 Aurora 版本支持 performance.now

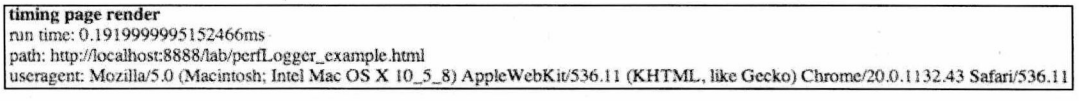


图 5-13 Chrome 20 支持 performance.webkitNow

到这里，我们已经将新的 Performance 对象整合进 perfLogger 库中了，并且在日志文件中添加了几个新列。这些新数据点为我们分析数据增加了额外的维度。我们拥有了新的捕获高分辨率时间的能力之后，我们就要对其进行妥善设置，以便在下一章开始收集运行时的度量值。

5.7 新数据可视化

现在，我们对新数据做些有趣的事情吧！我们能看到一些很有意义的数据：用户代理的平均加载时间是多少？按平均值考虑，HTTP 事务过程中哪一部分花费的时间最多？通常情况下加载时间分布情况是怎样的？现在开始吧。

以下是我们的数据的一个样例：

```
IP, TestID, StartTime, StopTime, RunTime, URL, UserAgent, PerceivedLoadTime, PageRenderTime,
RoundTripTime, TCPConnectionTime, DNSLookupTime, CacheTime, RedirectTime
75.149.106.130,page render,1341243219599,1341243220218,619,http://www.tom-barker.com/
```

```
blog/?p=x,Mozilla/5.0 (Macintosh; Intel Mac OS X 10.5; rv:13.0) Gecko/20100101 Firef
ox/13.0.1790,261,-2,36,0,-4,0
75.149.106.130,page_render,633.36499998695,973.8049999869,340.43999999994,http://www.tom-barker.
com/blog/?p=x,Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML like
Gecko) Chrome/20.0.1132.43 Safari/536.11633,156,-1341243238576,30,0,0,0
75.149.106.130,page_render,1498.2289999898,2287.9749999847,789.74599999492,http://www.tom-
barker.com/blog/?p=x,Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML
like Gecko) Chrome/20.0.1132.43 Safari/536.111497,979,788,0,0,0,0
```

首先，我们来看看可感知的加载时间的频率分布情况。当规模足够大的情况下，对于大多数用户来讲，这个数据会对用户的通常使用习惯给出恰当的描述。在第 3 章中，我们已经能够借助编写的 R 脚本从 perflogs 变量中读取性能数据了，所以，现在我们就用这个变量绘制一幅柱状图。

```
hist(perflogs$PerceivedLoadTime, main="Distribution of Perceived Load Time", xlab="Perceived Load
Time in Milliseconds", col=c("#CCCCCC"))
```

上述语句创建的图表如图 5-14 所示。

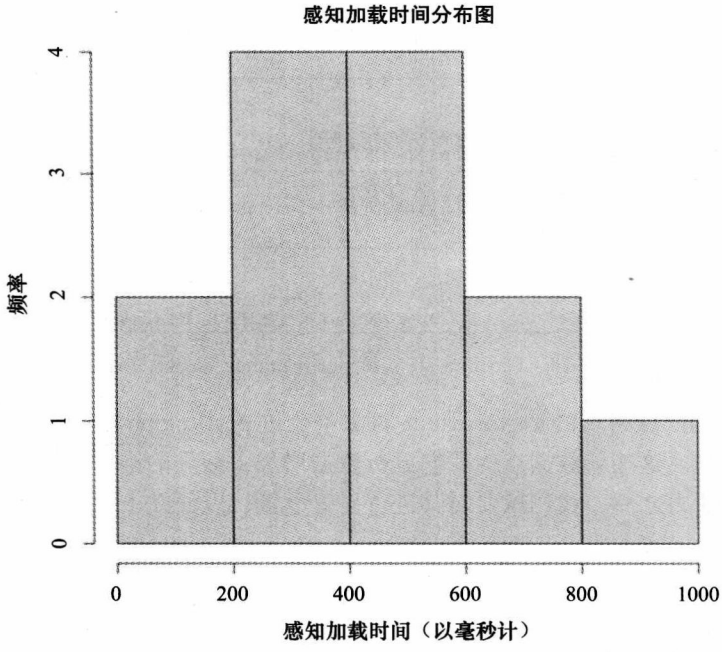


图 5-14 感知加载时间柱状图，频率表示访问次数

到目前为止，我们的例子还是很简单的。你只需要使用 pdf() 函数对上述代码进行封装，以便将图表保存到一个我们后续还会引用的文件中。

```
loadTimeDistrchart <- paste(chartDirectory, "loadtime_distribution.pdf", sep="")
pdf(loadTimeDistrchart, width=10, height=10)
     hist(perflogs$PerceivedLoadTime, main="Distribution of Perceived Load Time", xlab="Perceived
Load Time in Milliseconds", col=c("#CCCCCC"))
dev.off()
```

现在，我们来看看收集的所有完整的 HTTP 请求数据，并分解花费在每一个请求上的平均时间。这样，我们就能够对每一步所消耗的时间有一个全局性了解了。

首先，创建一个新的变量，用于保存我们下一幅图表的路径，然后创建一个新的 avgTimeBreakdownInRequest 函数来执行这一功能：

```
requestBreakdown <- paste(chartDirectory, "avgtime_inrequest.pdf", sep="")
avgTimeBreakdownInRequest <- function(){
}
```

在这个函数中，首先做一些数据清理的工作。如果遇到的数值特别大，R 就会用指数形式保存这些数值。这对于引用大的数值是非常有用的，但是通常情况下，要可视化这些数据，我们需要将它们展开以便于阅读。要做到这一点，需要向 options() 函数传递 scipen 参数。options() 函数允许我们在 R 中设置一些全局参数，scipen 参数存放的是一个数值型的值，该值越高，R 以固定格式而不是用指数形式显示数值的可能性就越大。

```
options(scipen=100)
```

接下来处理我们看到的数据中的负值。负数在我们的上下文环境中是没有意义的。这是因为有时候 window.performance 对象会返回数值 0，当我们用该值减去保存的性能数据时，就会出现负值。

到这里，你会面临几个选择：是删除负值所在的行，还是用 0 来代替负值。如果你认为，出现一个不好的数据就意味整行数据都是不可靠的，那么你可以删除整行数据，但是，在这里我们不会这么做。为了避免该行中其他列有用数据的丢失，我们的做法是用 0 来代替负值。

用 0 代替负值，我们需要检查每一个数值，看其是否小于 0，然后将小于 0 的列值修改为 0。做法如下所示：

```
perflogs$PageRenderTime[perflogs$PageRenderTime < 0] <- 0
perflogs$RoundTripTime[perflogs$RoundTripTime < 0] <- 0
perflogs$TCPConnectionTime[perflogs$TCPConnectionTime < 0] <- 0
perflogs$DNSLookupTime[perflogs$DNSLookupTime < 0] <- 0
```

接下来，你要创建一个数据帧，其包含每一列的平均值。要做到这一点，使用 data.frame() 函数，并使用函数指针将 mean() 函数作为参数传递给它，目的是对每一列计算平均值。然后为新的数据帧设置列名：

```
avgTimes <- data.frame(mean(perflogs$PageRenderTime), mean(perflogs$RoundTripTime), mean(perflogs$T
CPConnectionTime), mean(perflogs$DNSLookupTime))

colnames(avgTimes) <- c("PageRenderTime", "RoundTripTime", "TCPConnectionTime", "DNSLookupTime")
```

最后，创建图表。我们要创建一个水平条形图，就像在前面的章节创建的图表一样，最后将图表保存在一个 PDF 文件中。

```
pdf(requestBreakdown, width=10, height=10)
opar <- par(no.readonly=TRUE)
     par(las=1, mar=c(10,10,10,10))
```

```
    barplot(as.matrix(avgTimes), horiz=TRUE, main="Average Time Spent\nDuring HTTP Request",
xlab="Milliseconds")
par(opar)
dev.off()
```

完成后的函数如下所示，生成的图表如图 5-15 所示。

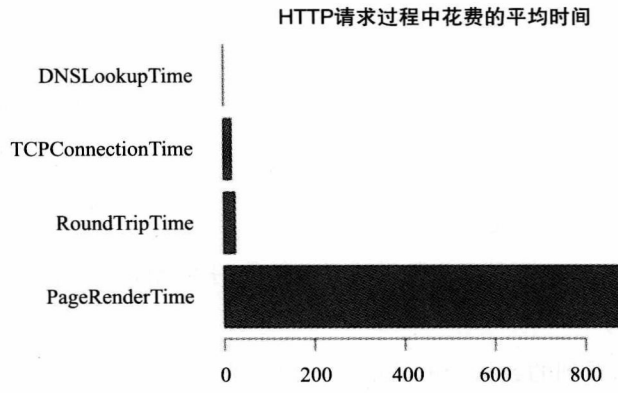


图 5-15 HTTP 请求过程中每一步的平均时间

```
avgTimeBreakdownInRequest <- function(){
#expand exponential notation
options(scipen=100)

#set any negative values to 0
perflogs$PageRenderTime[perflogs$PageRenderTime < 0] <- 0
perflogs$RoundTripTime[perflogs$RoundTripTime < 0] <- 0
perflogs$TCPConnectionTime[perflogs$TCPConnectionTime < 0] <- 0
perflogs$DNSLookupTime[perflogs$DNSLookupTime < 0] <- 0

#capture avg times
avgTimes <- data.frame(mean(perflogs$PageRenderTime), mean(perflogs$RoundTripTime), mean(perflogs$T
CPConnectionTime), mean(perflogs$DNSLookupTime))
colnames(avgTimes) <- c("PageRenderTime", "RoundTripTime", "TCPConnectionTime", "DNSLookupTime")

pdf(requestBreakdown, width=10, height=10)
opar <- par(no.readonly=TRUE)
     par(las=1, mar=c(10,10,10,10))
     barplot(as.matrix(avgTimes), horiz=TRUE, main="Average Time Spent\nDuring HTTP Request",
xlab="Milliseconds")
par(opar)
dev.off()
}
```

这是一个非常有趣的结果。你可能会觉得，页面渲染所花费的时间是最多的，因为它的开销是最大的。它不仅要传递消息，还要处理消息，使其变得有意义。但是，你可能没有考虑到 TCP 连接时间和数据往返的时间是一样的，或者说，它们明显高于 DNS 查询时间。

本章我们生成的最终图表是一个不同浏览器的感知加载时间图。注意，前面的样例数据存储了全部用户代理字符串，字符串中不仅包含了浏览器名称，还包含了浏览器版本、

子版本、操作系统信息，甚至是渲染引擎信息。太棒了，但是如果你想得到更高层的浏览器信息，与版本或操作系统无关的信息，你可以搜索用于特定高层浏览器名称的用户代理字符串。

你可以在数据帧列中使用 grep() 函数来搜索字符串。grep() 函数将要搜索的字符串作为第一个参数，将要进行搜索的向量或对象作为第二个参数。我们要做的是使用 grep() 函数作为我们传递给数据帧的过滤选项：

```
data frame[grep([string to search for], [data frame column to search])]
```

创建一个函数 getDFByBrowser()，它接收一个数据帧和一个要搜索的字符串作为参数，该函数将搜索操作进行常规处理。

```
getDFByBrowser<-function(data, browsername){
    return(data[grep(browsername, data$UserAgent),])
}
```

接下来，创建一个变量 loadtime_bybrowser，用于保存我们新创建的图表所在文件的路径：

```
loadtime_bybrowser <- paste(chartDirectory, "loadtime_bybrowser.pdf", sep="")
```

紧接着，创建一个函数 printLoadTimebyBrowser()，作为生成图表的主功能部分：

```
printLoadTimebyBrowser <- function(){
}
```

该函数首先为每一个将要在我们的图表中出现的浏览器创建一个数据帧：

```
chrome <- getDFByBrowser(perflogs, "Chrome")
firefox <- getDFByBrowser(perflogs, "Firefox")
ie <- getDFByBrowser(perflogs, "MSIE")
```

然后，为每一个浏览器创建一个用于保存平均感知加载时间的数据帧，就像前面的例子中处理 HTTP 请求数据那样。同样，给这些数据帧一个描述性的列名称：

```
meanTimes <- data.frame(mean(chrome$PerceivedLoadTime), mean(firefox$PerceivedLoadTime),
mean(ie$PerceivedLoadTime))
colnames(meanTimes) <- c("Chrome", "Firefox", "Internet Explorer")
```

最后，创建一个直方图并保存为 pdf 文件。

```
pdf(loadtime_bybrowser, width=10, height=10)
    barplot(as.matrix(meanTimes), main="Average Perceived Load Time\nBy Browser", ylim=c(0,
600), ylab="milliseconds")
dev.off()
```

完成后的功能（代码）应该类似下面这样，而得到的图表应该如图 5-16 所示。

```
getDFByBrowser<-function(data, browsername){
    return(data[grep(browsername, data$UserAgent),])
}

printLoadTimebyBrowser <- function(){
    chrome <- getDFByBrowser(perflogs, "Chrome")
```

```
    firefox <- getDFByBrowser(perflogs, "Firefox")
    ie <- getDFByBrowser(perflogs, "MSIE")

    meanTimes <- data.frame(mean(chrome$PerceivedLoadTime), mean(firefox$PerceivedLoadTime),
mean(ie$PerceivedLoadTime))
    colnames(meanTimes) <- c("Chrome", "Firefox", "Internet Explorer")
    pdf(loadtime_bybrowser, width=10, height=10)
        barplot(as.matrix(meanTimes), main="Average Perceived Load Time\nBy Browser",
ylim=c(0, 600), ylab="milliseconds")
    dev.off()
}
```

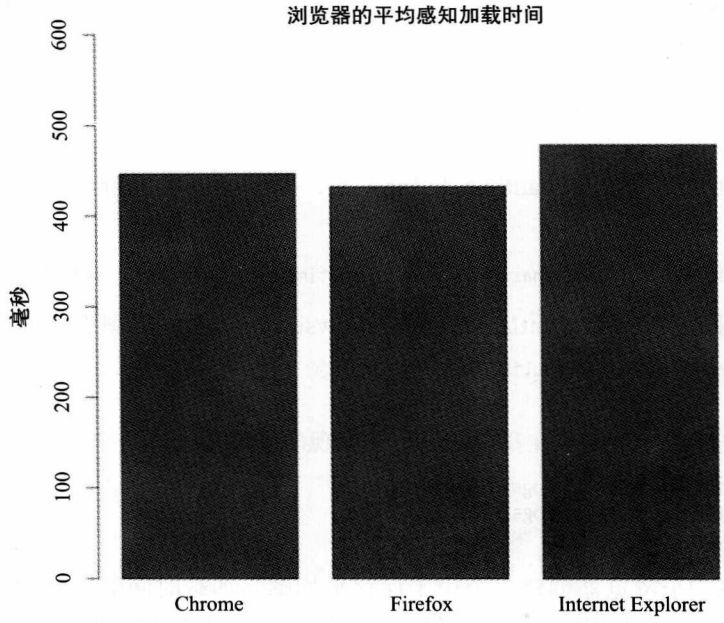


图 5-16　不同浏览器的平均感知加载时间

太有趣了，我们测试的结果显示，Firefox 的感知加载时间是最短的，其次是 Chrome，最后是 Internet Explore。这还仅仅是最小级别的比较。如果愿意，我们还可以添加版本和子版本的比较。

修改后的完整 R 文件如下所示：

```
dataDirectory <- "/Applications/MAMP/htdocs/lab/log/"
chartDirectory <- "/Applications/MAMP/htdocs/lab/charts/"
testname = "page_render"

perflogs <- read.table(paste(dataDirectory, "runtimeperf_results.csv", sep=""), header=TRUE,
sep=",")
perfchart <- paste(chartDirectory, "runtime_",testname, ".pdf", sep="")

loadTimeDistrchart <- paste(chartDirectory, "loadtime_distribution.pdf", sep="")
requestBreakdown <- paste(chartDirectory, "avgtime_inrequest.pdf", sep="")
loadtime_bybrowser <- paste(chartDirectory, "loadtime_bybrowser.pdf", sep="")
```

```
pagerender <- perflogs[perflogs$TestID == "page_render",]
df <- data.frame(pagerender$UserAgent, pagerender$RunTime)
df <- by(df$pagerender.RunTime, df$pagerender.UserAgent, mean)
df <- df[order(df)]

pdf(perfchart, width=10, height=10)
opar <- par(no.readonly=TRUE)
     par(las=1, mar=c(10,10,10,10))
     barplot(df, horiz=TRUE)
par(opar)
dev.off()


getDFByBrowser<-function(data, browsername){
     return(data[grep(browsername, data$UserAgent),])
}


printLoadTimebyBrowser <- function(){
     chrome <- getDFByBrowser(perflogs, "Chrome")
     firefox <- getDFByBrowser(perflogs, "Firefox")
     ie <- getDFByBrowser(perflogs, "MSIE")

     meanTimes <- data.frame(mean(chrome$PerceivedLoadTime), mean(firefox$PerceivedLoadTime),
mean(ie$PerceivedLoadTime))
     colnames(meanTimes) <- c("Chrome", "Firefox", "Internet Explorer")
     pdf(loadtime_bybrowser, width=10, height=10)
          barplot(as.matrix(meanTimes), main="Average Perceived Load Time\nBy Browser",
ylim=c(0, 600), ylab="milliseconds")
     dev.off()
}


pdf(loadTimeDistrchart, width=10, height=10)
     hist(perflogs$PerceivedLoadTime, main="Distribution of Perceived Load Time", xlab="Perceived
Load Time in Milliseconds", col=c("#CCCCCC"))
dev.off()

avgTimeBreakdownInRequest <- function(){

#expand exponential notation
options(scipen=100, digits=3)

#set any negative values to 0
perflogs$PageRenderTime[perflogs$PageRenderTime < 0] <- 0
perflogs$RoundTripTime[perflogs$RoundTripTime < 0] <- 0
perflogs$TCPConnectionTime[perflogs$TCPConnectionTime < 0] <- 0
perflogs$DNSLookupTime[perflogs$DNSLookupTime < 0] <- 0

#capture avg times
avgTimes <- data.frame(mean(perflogs$PageRenderTime), mean(perflogs$RoundTripTime), mean(perflogs$T
CPConnectionTime), mean(perflogs$DNSLookupTime))
colnames(avgTimes) <- c("PageRenderTime", "RoundTripTime", "TCPConnectionTime", "DNSLookupTime")
pdf(requestBreakdown, width=10, height=10)
opar <- par(no.readonly=TRUE)
     par(las=1, mar=c(10,10,10,10))
     barplot(as.matrix(avgTimes), horiz=TRUE, main="Average Time Spent\nDuring HTTP Request",
```

```
xlab="Milliseconds")
par(opar)
dev.off()

}

#invoke our new functions
printLoadTimebyBrowser()
avgTimeBreakdownInRequest()
```

5.8 小结

本章深入探讨了 window.performance 对象，这是 W3C 制定的用于收集浏览器性能度量值的一种新的标准化方法。我们将 Performance 对象合并入 perfLogger 项目中，包括每个运行的测试用例中的性能度量值，而且在浏览器支持的情况下，实现对 window.performance 高分辨率时间支持的构造。

我们使用一个新的数据来查看我们网站整体的 Web 性能情况。

我们还谈到了一个新的可能性，就是查看客户端机器浏览器的内存使用情况，特别是在 Chrome 上的实现，同时我们也简要了解了如何在 Firefox 上获取同样的数据。

下一章，我们会探讨如何优化 Web 性能。你将会运用我们开发的工具，运行多变量测试，查看优化之后的性能结果。

第 6 章

Web 性能优化

第 5 章，我们深入探讨了 W3C 关于跟踪浏览器性能标准的第一步，即 Performance 对象。第 5 章还对 Performance 对象的每一个 API 进行了介绍；你还了解到，如何从 Performance Timing 对象中收集并获取性能指标，如何利用性能 Performance Navigation 对象来判断用户是以什么方式访问你的网站的，以及 Performance 对象如何公开高分辨率时间，来帮助用户跟踪精度高于毫秒的时间数据。

基于以上所有内容，我们修改了 perfLogger 库，以整合 Performance 对象提供的所有指标。在 perfLogger 中，你构建了一个垫片（shims），用于在支持高分辨率时间的浏览器上使用该项技术，而对于不支持高分辨率时间的浏览器则执行正常的回退。

然后，利用所有这些得到的新数据，扩展 R 脚本，对这些数据进行图表化处理，利用图表化的数据来描述用户情况，包括他们的连接情况以及他们使用什么样的浏览器。

本章，你将要使用到我们已经创建的所有工具，以及在第 2 章我们了解到的所有工具，来对某些 Web 性能优化技巧可能带来的改进进行量化——尤其是对“ JavaScript 如何能阻塞页面渲染”以及“如何使用 JavaScript 延迟大块内容的加载”这两个技巧进行重点的关注。记住，Web 性能是指页面内容传递给终端用户所花费的时间，包括网络延迟和浏览器渲染时间。

6.1 优化页面的渲染瓶颈

首先，我们来看看浏览器内容渲染的优化。不考虑网络延迟，我们仅仅关注浏览器如何能够快速地处理和渲染内容，并展现给终端用户。我们首先来看看，现代浏览器是如何将我们的内容绘制在屏幕上的。

如第 1 章所述，现代浏览器大多是由几个相互作用的组件构成的。其中的 UI 层用于绘制浏览器面向客户端的接口，包括地址栏、“后退”和“前进”按钮，以及浏览器上的其他功能按钮。终端用户与 UI 层和从 UI 层派生出的其他应用进行交互。

还有一个网络层用于处理网络连接，建立 TCP 连接，执行 HTTP 往返。网络层也会向渲染引擎提供内容。

渲染引擎用于在屏幕上绘制内容。当它遇到 JavaScript 时，它就将其移交给 JavaScript 解释器处理。图 6-1 是对现代浏览器的高层架构的注解。

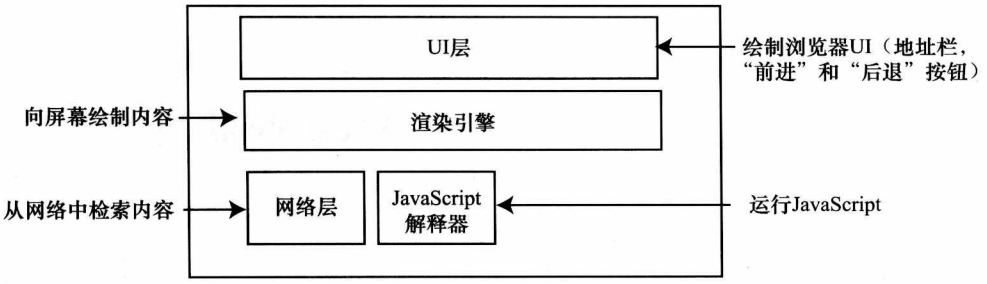


图 6-1 浏览器架构注解

因此正如我提到过的，渲染引擎——不管是 Gecko，还是 WebKit，或者其他的渲染引擎，都是从网络层获取内容的。它的吞吐量是有限的，因此它以块的方式提取数据，并以块的方式将数据提交给它自己的工作流。

这个工作流过程分为几个步骤。首先是解析内容，这意味着标记也会当做字符被读取，并进行词法分析，在词法分析过程中，字符要与一个规则集进行比较，并基于该规则集转化为特定的符号。这个规则集就是在 HTML 文档中定义的 DTD ⊖，它指定了我们所使用的特定版本语言的标签。而符号不过是将这些字符分割成具有一定意义的片段而已。

比如，网络层可能会返回如下字符串：

```
<!DOCTYPE html><html><head><meta charset="UTF-8"/>
```

该字符串将会标记成有意义的片段，如下所示：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8"/>
```

而后，渲染引擎将这些符号转换成 DOM 元素。DOM 元素以树形结构布局，方便渲染引擎对每一个元素进行迭代。首次迭代时，渲染引擎会勾画出 DOM 元素的位置，然后在后续的迭代中将这些元素绘制在屏幕上。图 6-2 为该工作流程。

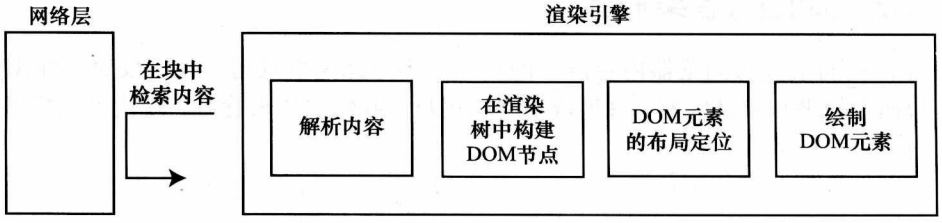


图 6-2 渲染引擎工作流程

⊖ DTD（Document Type Definition）是一套关于标记符的语法规则。DTD 是一种保证 XML 文档格式正确的有效方法，可通过比较 XML 文档和 DTD 文件来看文档是否符合规范，元素和标签使用是否正确。——译者注

一般来讲，在标记化过程中，如果渲染引擎识别出了一个外部脚本，它就会暂停对脚本内容的解析，转而开始下载脚本。只有当脚本下载完毕并得到执行之后，渲染引擎才会回过头来继续解析。这一过程，可能会导致在向终端用户显示内容的过程中，产生时间上的延迟。图 6-3 是该时间延迟的流程图。

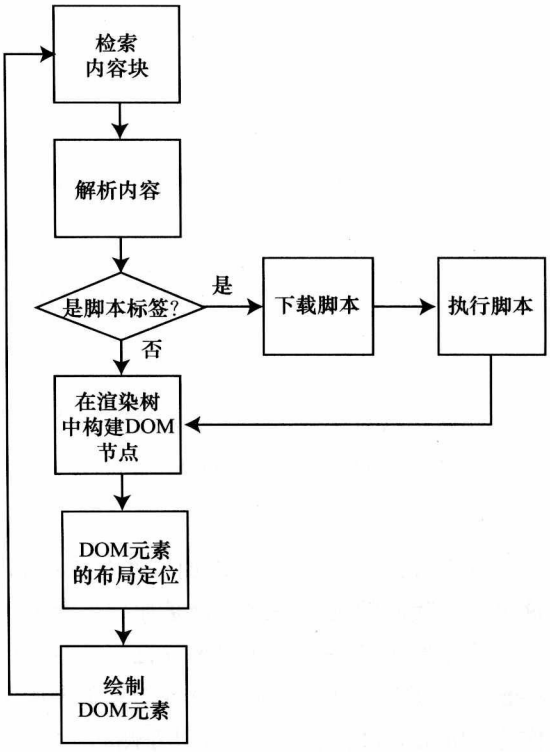


图 6-3 解析过程中遇到脚本标签的处理流程

要解决这一潜在的问题，通常采用的方法[⊖]是将所有的脚本都放置于 HTML 的末尾，这样，渲染过程中处理 JavaScript 脚本的任何延迟都会在页面完成渲染之后才会出现。

6.1.1 脚本加载

还有一个办法就是编写程序，下载远程脚本文件。这就是所谓的“脚本加载”，它可以起到欺骗渲染引擎的目的。让我们看看怎么做。

记住我们上文探讨过的工作流程，就是渲染引擎处理外部脚本时产生暂时停滞的工作流程。浏览器查找脚本标签的 src 属性，该属性会指出脚本必须要从一个远端资源下载。如果碰到的脚本标签没有 src 属性，那么渲染引擎就只会将代码传递给 JavaScript 解释器去执行。

因此，我们所能做的就是创建一段内嵌的 JavaScript 代码，动态地将脚本标签附加到文档后边，如下所示：

```
<script>
var script = window.document.createElement('SCRIPT');
script.src = src;
window.document.getElementsByTagName('HEAD')[0].appendChild(script);
</script>
```

该代码段使用 document.createElement() 函数，创建了一个新的脚本标签，并将其存储在一个名字为 script 的变量中。然后，它设置新脚本标签的 src 属性，使其指向一个远程 JavaScript 文件，最后它将新脚本标签附加到文档的 Head 节后边。

这是一个相当简单的方法，我们用这个方法修改一个真实的例子。

首先，创建一个命名空间，我们称其为 remoteLoader：

⊖ 至少，到 Steve Souders 完成他的著作《 High Performance Web Sites ： Essential Knowledge for Front-End Engineers 》(O ' Reilly 2007 年出版）以来，都是采用的这个方法。

```
var remoteLoader = function remoteLoader(){

return{

}
}();
```

在remoteLoader作用域内，创建一个私有函数，该函数使用与前边的代码中相同的逻辑结构构建脚本标签。将该函数命名为constructScriptTag()，将脚本标签的源地址作为URL参数，传递给该函数。

```
function constructScriptTag(src){
    var script = window.document.createElement('SCRIPT');
    script.src = src;
return script;
}
```

在返回的对象中，创建一个简单的公有函数loadJS()，它的参数名为script_url：

```
loadJS: function(script_url){

}
```

在loadJS内部，你会添加一些分支逻辑，用于测试传递进来的值是数组还是字符串。这可以让我们的API灵活选择是加载单个JavaScript文件或是一组JavaScript文件。

为了验证这一点，我们来看看变量的类型。数组返回一个“object”类型，字符串返回“string”类型。为了能够从其他对象类型中判断出数组，你需要在变量上使用instanceof操作符。Instanceof操作符会测试其左侧的对象是否是其右侧构造函数原型链上的一个实例——换句话说，它是该构造函数的实例么？

```
if(typeof script_url === "object"){
if(script_url instanceof Array){
}
}else if(typeof script_url === "string"){

}
```

先来完善“字符串分支”这一部分，因为它最简单。只需要调用constructScriptTag()函数，并将script_url参数传递过去就可以了。然后将返回的元素附加到文档的head节中。

```
else if(typeof script_url === "string"){
window.document.getElementsByTagName('HEAD')[0].appendChild(constructScriptTag(script_url));
}
```

“数组分支”将会对URL数组进行迭代，为每一个元素创建一个新的脚本标签，然后使用一个文档片段（document fragment）来保存所有的新元素。最后一并将它们附加到head节末尾。下一章会详细地讲解文档片段（document fragment）。

```
if(script_url instanceof Array){
var frag = document.createDocumentFragment();
    for(var ind = 0; ind < script_url.length; ind++){
```

```
            frag.appendChild(constructScriptTag(script_url[ind]));
        }
window.document.getElementsByTagName('HEAD')[0].appendChild(frag.cloneNode(true) );
}
```

完成后的代码如下所示：

```
var remoteLoader = function remoteLoader(){
    function constructScriptTag(src){
        var script = window.document.createElement('SCRIPT');
        script.src = src;
        return script;
    }

    return{
        loadJS: function(script_url){
            if(typeof script_url === "object"){
                if(script_url instanceof Array){
                    var frag = document.createDocumentFragment();
                    for(var ind = 0; ind < script_url.length; ind++){
                        frag.appendChild(constructScriptTag(script_url[ind]));
                    }
                    window.document.getElementsByTagName('HEAD')[0].appendChild(frag.
cloneNode(true) );
                }
            }else if(typeof script_url === "string"){
                window.document.getElementsByTagName('HEAD')[0].appendChild(constructScriptTag(
script_url));
            }
        }
    }
}();
```

你可以通过传递一个字符串或者一个数组来调用该函数，如下面的代码片段所示：

```
<script>
    remoteLoader.loadJS("/lab/perfLogger.js"); // passing in a string
</script>

<script>
    remoteLoader.loadJS(["/lab/perfLogger.js", "jquery.js"]); // passing in an array
</script>
```

6.1.2 异步

另一个阻止渲染引擎产生阻塞的方法是使用脚本标签的 async（异步）属性。该属性在 HTML5 中被引入，是一个原生属性，用于告诉浏览器异步地加载脚本。在现代的浏览器中，都支持该属性，甚至是 Internet Explorer 也从版本 10 开始支持 async 属性了。（在 Internet Explorer 10 之前，IE 使用其专有的属性 defer 实现同样的功能）。async 属性是可选属性，它接受一个布尔值；当脚本标签中未设定该属性的值时，默认取值为 true。

```
<script src="[URL]" async=true></script>
<script src="[URL]" async></script>
```

使用 async 属性之后，你就不会知道文件是什么时候完成下载的，因此你需要附加一个

onload 事件句柄给 script 标签。这样，当文件下载完成之后，就会调用或实例化需要执行的代码了。

```
<script src="[URL"] async onload="init();]"></script>
```

6.1.3 对比结果

到这里，你应该知道下一步该做什么了——运行多变量测试，比较我们提到过的每一个方法的结果！

本次测试，你需要创建一个基准，即一个没有对外部脚本做任何优化的页面。然后在基准页面文件上的 head 节中，加载 perfLogger.js。

```
<head>
… [snip head content]
<script src="/lib/perfLogger.js"></script>
<script>
perfLogger.startTimeLogging("page_render", "timing page render", true, true);
</script>
</head>
```

现在，使用 remoteLoader 创建一个文件，以通过代码来加载 perfLogger.js。你需要先将 remoteLoader.js 放置在 head 节中，然后在 body 节中调用 loadJS。

```
<head>
…[snip head content]
<script src="/lib/remoteLoader.js"></script>
</head>
<body>
<script>
remoteLoader.loadJS("/lab/perfLogger.js");
</script>
… [snip body content]
<script>
perfLogger.showPerformanceMetrics();
</script>
</body>
```

最后，创建一个页面，在 script 标签中使用 async 属性：

```
<head>
…[snip head content]
<script async src="/lab/perfLogger.js"></script>
</head>
<body>
… [snip body content]
<script>
perfLogger.showPerformanceMetrics();
</script>
</body>
```

现在，使用 WebPagetest 运行每一个页面。

每一个页面的内容都是相同的，也就是我的网站 tombarker.com 主页面的快照，其 URL 地址如下：

用于测试的 URL	测试结果 URL
tom-barker.com/lab/baseline.html	http://www.webpagetest.org/result/120712_D2_16a7c450629a5765171f4a4c2d9e016e/
tom-barker.com/lab/scriptloaded.html	http://www.webpagetest.org/result/120712_9W_408ca6d9d9e428f28e7f3e1adff126d7/
tom-barker.com/lab/asyncloaded.html	http://www.webpagetest.org/result/120712_90_1f667f8a71811c39ee0cc9066f78645d/

图 6-4 ~ 图 6-6 描绘了每一次测试的概要汇总时间。

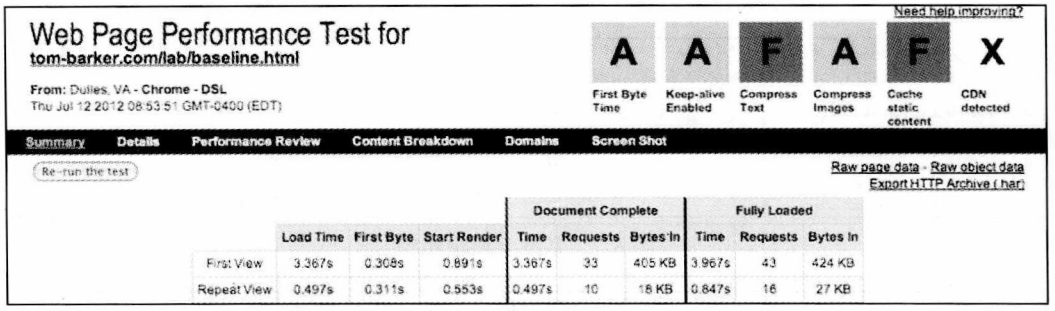


图 6-4 WebPagetest 运行于基准页面的结果汇总

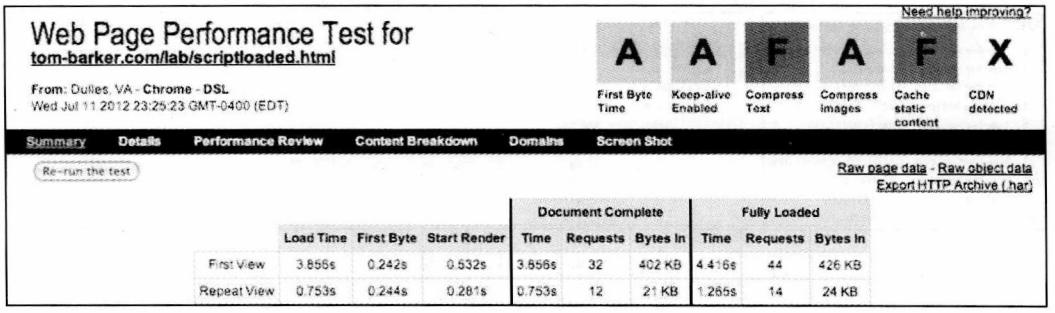


图 6-5 WebPagetest 运行于脚本动态加载情况下的结果汇总

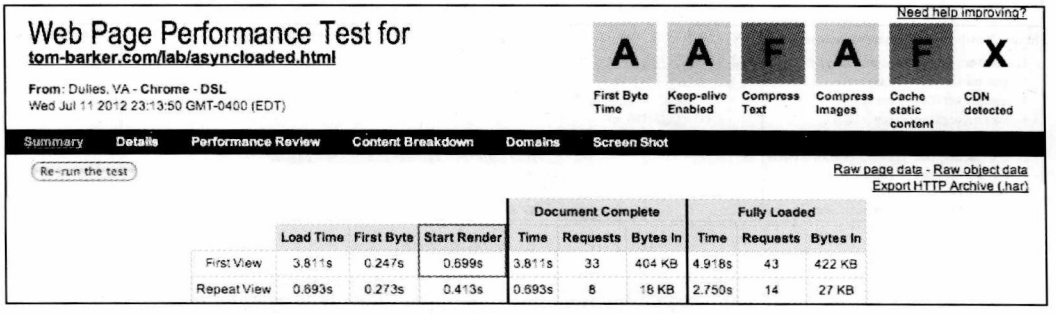


图 6-6 WebPagetest 运行于脚本异步加载情况下的结果汇总

从上面图示的结果中，你可以看到，整体加载时间的差异是非常小的，可以忽略不计，

但是通常的差异是首字节（first byte）时间和渲染启动（start render）时间。remoteLoader.js 页面给出了最佳的渲染启动时间，比基准页面快了 350 毫秒，比异步页面快了 160 毫秒。

有得必有失，总的加载时间变长了，但是页面渲染的速度却变快了，因此对于终端用户来说，就好像是页面加载变快了。

为了弄明白为什么渲染启动变快了，我们来看看瀑布图。图 6-7 ~ 图 6-9 是我们测试的瀑布图。

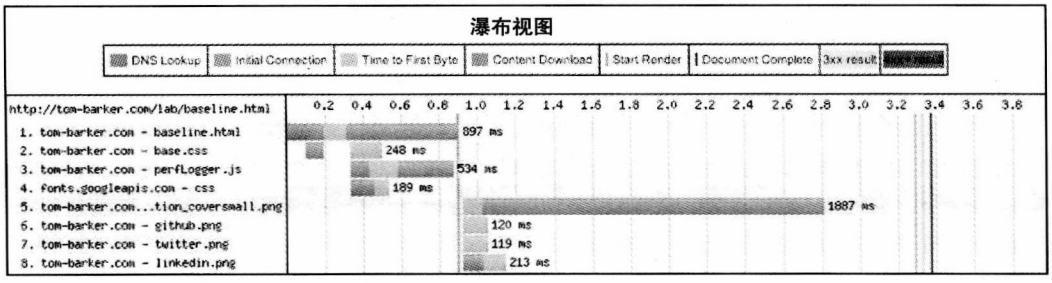


图 6-7 基准文件瀑布图

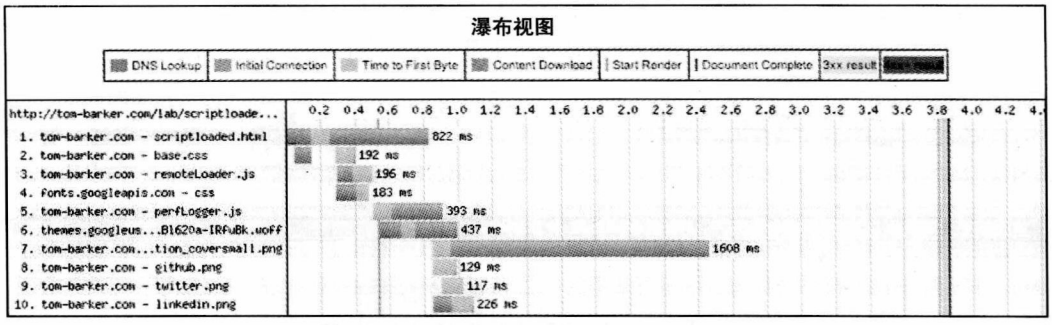


图 6-8 脚本加载文件瀑布图

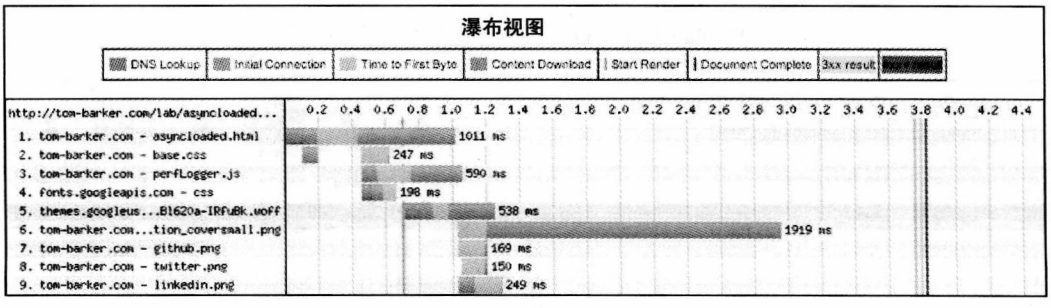


图 6-9 异步文件瀑布图

在这些瀑布图中，我们可以看到 perfLogger.js 是如何对页面元素的加载顺序产生影响的。

对于基准页面（见图 6-7），我们看到，当浏览器正在连接 perfLogger.js 时，它也正在连接并下载我们的 Web 字体文件，但是一旦浏览器开始下载 perfLogger.js 文件，所有其他的进程都

被停止了，直到下载完成为止。在图 6-7 的第 5 ~ 8 行特别明显，我们看到所有的 PNG 加载都处于等待状态，等待了大概 897 毫秒之后，当 perfLogger.js 下载完成之后才开始加载。

对于脚本加载测试（见图 6-8），我们看到浏览器没有因为下载 perfLogger.js 而阻塞。在第 6 行，我们的 Web 字体文件与 perfLoader.js 文件是并行下载的。

同样，对于异步测试（见图 6-9），我们看到浏览器下载一个外部 CSS 文件和一个 Web 字体文件，这些都和我们的 perfLoader.js 文件下载是并行执行的。第 3 ~ 5 行显示的就是这样的情形。

最后，我们来看看每个页面的 perfLogger 指标值。

测　试	结果（毫秒）
基准页面	Perceived Time：342 Redirect Time：0 Cache Time：0 DNS Lookup Time：0 TCP Connection Time：0 roundTripTime：162 pageRenderTime：263
脚本加载页面	Perceived Time：277 Redirect Time：0 Cache Time：0 DNS Lookup Time：0 TCP Connection Time：0 roundTripTime：196 pageRenderTime：207
异步页面	Perceived Time：343 Redirect Time：0 Cache Time：0 DNS Lookup Time：0 TCP Connection Time：0 roundTripTime：158 pageRenderTime：212

我们可以看到，可感知时间和页面渲染时间在一个单独的测试中是有所改进的，但是如果我们在一个比较大的规模上运行这些测试，这些改进是否会被平均化呢，或者它们是否能反映出较大的性能改进呢？

幸运的是，我们构造了 perfLogger 来保存所有的结果，因此，我们来看看日志文件，并使用 R 来解析这些结果。

首先，写一个新的 R 函数，创建一个保存 URL 的数据帧：

```
getDFByURL<-function(data,url){
    return(data[grep(,url, data$URL)])
}
```

然后，创建一个新的名为 comparePerfMetricsbyURL 函数：

```
comparePerfMetricsbyURL<-function(){
}
```

在该函数中，使用刚刚创建的 getDFByURL() 函数，为每个测试创建一个变量：

```
baseline <- getDFByURL(perflogs, "http://tom-barker.com/lab/baseline.html")
scripted <- getDFByURL(perflogs, "http://tom-barker.com/lab/scriptloaded.html")
async <- getDFByURL(perflogs, "http://tom-barker.com/lab/asyncloaded.html")
```

然后，创建一个用于为每个测试的 URL 保存平均页面渲染时间的数据帧，以及一个用于为每一个测试的 URL 保存平均加载时间的数据帧。你也可以修改每一个数据帧列名，以确保在图表中得到较为规整的 x 轴数值。

```
meanRenderTimes <- data.frame(mean(baseline$PageRenderTime), mean(scripted$PageRenderTime),
mean(async$PageRenderTime))

colnames(meanRenderTimes) <- c("Baseline", "Script Loaded", "Async")

meanLoadTimes <- data.frame(mean(baseline$PerceivedLoadTime), mean(scripted$PerceivedLoadTime),
mean(async$PerceivedLoadTime))

colnames(meanLoadTimes) <- c("Baseline", "Script Loaded", "Async")
```

最后，使用这些数据帧创建柱状图：

```
barplot(as.matrix(meanRenderTimes), main="Average Render Time\nBy Test Type", ylim=c(0, 400),
ylab="milliseconds")

barplot(as.matrix(meanLoadTimes), main="Average Load Time\nBy Test Type", ylim=c(0, 700),
ylab="milliseconds")
```

完整的函数如下所示：

```
comparePerfMetricsbyURL<-function(){
    baseline <- getDFByURL(perflogs, "http://tom-barker.com/lab/baseline.html")
    scripted <- getDFByURL(perflogs, "http://tom-barker.com/lab/scriptloaded.html")
    async <- getDFByURL(perflogs, "http://tom-barker.com/lab/asyncloaded.html")

    meanRenderTimes <- data.frame(mean(baseline$PageRenderTime), mean(scripted$PageRenderTime),
mean(async$PageRenderTime))
    colnames(meanRenderTimes) <- c("Baseline", "Script Loaded", "Async")
    meanLoadTimes <- data.frame(mean(baseline$PerceivedLoadTime), mean(scripted$PerceivedLoadTi
me), mean(async$PerceivedLoadTime))
    colnames(meanLoadTimes) <- c("Baseline", "Script Loaded", "Async")

    barplot(as.matrix(meanRenderTimes), main="Average Render Time\nBy Test Type", ylim=c(0,
400), ylab="milliseconds")
    barplot(as.matrix(meanLoadTimes), main="Average Load Time\nBy Test Type", ylim=c(0, 700),
ylab="milliseconds")
}
```

这段代码生成的图表如图 6-10 ~ 图 6-11 所示。

我们看到，改进并不明显。对于终端用户来说，页面显示的速度越快，其表现出来的加载速度也越快，但是根据我们的技术测量方法，实际上加载速度并没有更快。真实的情况是，由于需要加载额外的资源，在加载脚本的时候，加载过程看上去稍微有点延长了。记住，性

能目标是可变的，并且是非常微妙的。

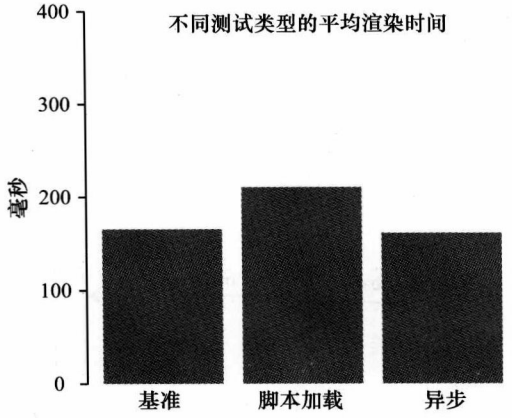


图 6-10 每个测试的平均渲染时间

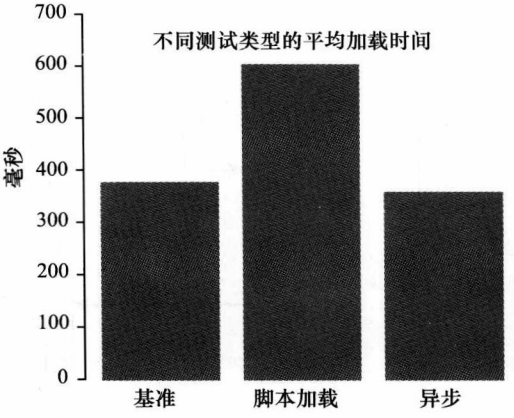


图 6-11 每个测试的平均加载时间

有件事情要牢牢记住，那就是我们用这些图表测量性能的方法，它看上去花费的时间有点长，这是因为我们是在页面的 onload 事件中计量时间的。如果在 onload 事件激发之前，我们根本用不到外部脚本，则可以使用一种称为“惰性加载”（lazy loading）的设计模式，在 onload 事件之后加载我们的脚本。

6.2 惰性加载

我们现在来看看惰性加载，这是一个通过编写程序延迟加载资源的方法。我们会研究一下作为一种设计模式，惰性加载到底是什么，我们如何使用这种设计模式来有效地改进页面的 Web 性能。

6.2.1 惰性加载的艺术

概括地讲，惰性加载是一种设计模式，使用这种模式，我们可以推迟事物的创建或者某些初始化工作，直到必须要完成创建或初始化工作为止（见图 6-12）。

这个模式有几种不同的实现：

- 虚拟代理模式；
- 惰性初始化模式；
- 值持有模式。

在虚拟代理模式中，如图 6-13 所示，我们先实例化一个存根，当需要的时候再加载进实际的实现功能，并将其暴露出来（一般通过组合的方式）。当应用程序中有不常使用，或不急需使用（很像 JavaScript 文件，在页面加载之前可能还用不到）的模块或组件时，通常会使用该模式。

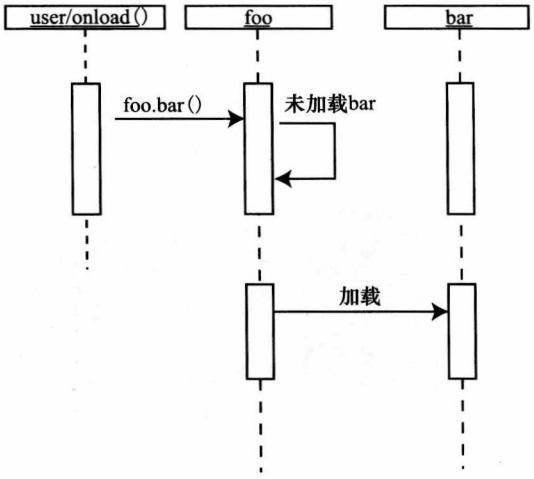


图 6-12 惰性加载序列图

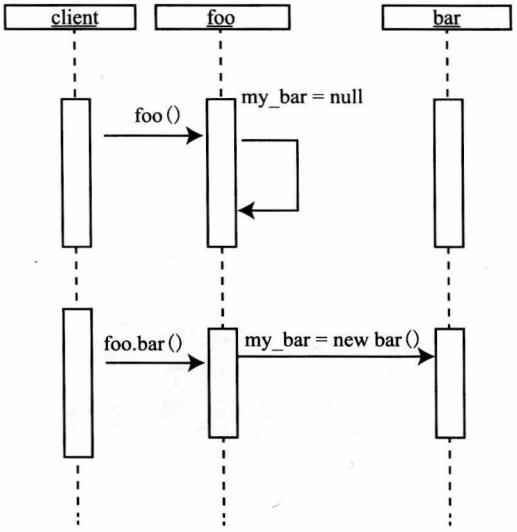


图 6-13 虚拟代理模式

在惰性初始化模式中，我们需要检查一下对象是否存在（是否为 null？），如果不存在，我们就实例化它。该模式很可能是大多数人曾经用过的模式，但是大家都还不了解这就是所谓的惰性初始化。该模式通常用于声明 JavaScript 对象或命名空间。如果我们的命名空间存在，则使用现有的，如果不存在，则创建它。

```
if (obj == null){
    obj = new obj();
}
return obj;
```

最后，在值持有模式中，我们调用一个对象的公有函数来得到对象的实例。该对象只有在第一次调用这个函数时才会被实例化。值持有模式常用于单例模式（singleton pattern）的部分实现。

注意

设计模式是已经明确定义的核心算法模式，并创建一个词汇对其进行命名，该命名用通俗易懂的语言描述了解决某些共同问题所需要的步骤。关于设计模式，开创性的一件事情就是《设计模式：可复用面向对象软件的基础》（Design Patterns：Elements of Reusable Object-Oriented Software）的出版，作者是 Erich Gamma 等（Addison-Wesley，1994 年出版）。

而反模式的思想一直以来也有明确的定义。反模式也就是逆设计模式，即通常的、系统性的、不断重复的错误，而这些错误一旦被识别是可以避免的。设计模式与反模式，任学其一都是非常有用的。

人类的历史和科学都是以先验知识为基础，并建立在先验知识之上。我们避免了重复发明的漩涡。这就是设计模式和反模式的思想：我们已经发现了问题，并且有了最有效的解决方法，那么我们要做的是将精力集中到新的挑战上面，并努力解决它，然后保存好我们的

新发现，使其将来成为子孙后代的财富。

相比“生产其他对象的对象”的说法，把它叫做“工厂”显得更简洁，也更省笔墨。

6.2.2 惰性加载脚本

当我们不需要 JavaScript 代码在页面加载之后就马上起作用时，我们可以在完成 window.onload 事件之后，用脚本的方式加载外部 JavaScript。这就是惰性加载脚本。我们来看看如何做到惰性加载脚本，看看这么做会给性能带来怎样的改进。最后我们会对结果进行一下评估。

1. 设置惰性加载

首先，我们创建一个新的测试用的 URL，用于实验。使用我们之前的例子来做脚本加载页面，将其重命名为 lazyloadscript.html。还需要修改我们的 remoteLoader 对象。

在新的 lazyloadscript.html 页面中，需要添加一个脚本标签和一些 JavaScript 代码，用于检查 window.attachEvent 函数在当前浏览器中是否有效。attachEvent 函数接受两个参数：一个附带的事件参数，一个事件发生时要激发的函数。如果当前浏览器支持 window.attachEven 事件，则需要将 onload 事件和 remoteLoader.loadJS 函数传递给它，用于加载远程脚本。

如果浏览器不支持 attachEvent 事件，则使用 window.addEventListener() 作为代替，也可起到同样的效果。下面就是相应的代码：

```
<script>
if (window.attachEvent)
    window.attachEvent('onload', remoteLoader.loadJS("/lab/perfLogger.js"));
else
    window.addEventListener('load', remoteLoader.loadJS("/lab/perfLogger.js"), false);
</script>
```

从技术的角度讲，你刚才已经用惰性加载的方式加载了远程脚本——但是，还有一个问题。到现在为止，我们还不知道远程脚本什么时候才会加载完成。如果你试图调用 perfLogger.showPerformanceMetrics() 函数，这时脚本还没有被加载就执行，页面会返回一个错误。

因此，你需要做一些修改。你需要知道脚本什么时候加载完成了，并在加载完成之后运行性能测试。因此，你需要让 remoteLoader.loadJS 函数接收一个回调（callback）。

注意

回调能够执行回调函数。这是函数开发中一件非常美妙的事情，我们可以在函数或对象之间传递函数。它开启了一个不需要继承就可以改变函数功能的新途径。

因此，我们进入到 remoteLoader 中，修改 constructScriptTag() 函数和 loadJS() 函数。为它们再添加一个参数，该参数就指向回调函数。

```
loadJS: function(script_url, f){
}

function constructScriptTag(src, func){
}
```

在 constructScriptTag() 函数中，你需要检查一下回调是否已经传递进来了，如果是，则添加一个 onload 属性给脚本对象，并将回调函数赋给该 onload 属性。这样，当脚本完成文件加载之后，浏览器就会执行这个回调函数。事实上，基于不同的浏览器，回调还可以扮演不同的角色。你可以在每次连接状态改变的时候，调用回调函数，就像 AUAX 事务做的那样。因此，在回调函数中，编码要严格依此编写。

```
if(func){
script.onload  = func;
}
```

修改后的 remoteLoader 文件如下所示：

```
var remoteLoader = function remoteLoader(){
     function constructScriptTag(src, func){
          var script = window.document.createElement('SCRIPT');
          script.src = src;
          if(func){
               script.onload  = func;
               }
          return script;
     }

     return{
          loadJS: function(script_url, f){
               if(typeof script_url === "object"){
                    if(script_url instanceof Array){
                         var frag = document.createDocumentFragment();
                         for(var ind = 0; ind < script_url.length; ind++){
                              frag.appendChild(constructScriptTag(script_url[ind]), f);
                         }
                         window.document.getElementsByTagName('HEAD')[0].appendChild(frag.
cloneNode(true));
                    }
               }else if(typeof script_url === "string"){
               window.document.getElementsByTagName('HEAD')[0].appendChild(constructScriptTag(sc
ript_url, f))
               }
          }
     }
}();
```

现在，修改页面中的代码模块，将回调函数传递进来。先使用一个存根来表示这个函数，现在假定这个函数名为 init：

```
<script>
if (window.attachEvent)
     window.attachEvent('onload', remoteLoader.loadJS("/lab/perfLogger.js"), init);
else
     window.addEventListener('load', remoteLoader.loadJS("/lab/perfLogger.js", init), false);
</script>
```

接下来，我们来充实我们的 init 函数。我们知道，在这里我们想要调用 perfLogger.showPerformanceMetrics() 函数，但是因为只有当脚本加载之后（还没有被执行时），浏览器才有必要调用这个函数，并且当解释器执行脚本时，你需要检查一下 perfLogger 是否已经初始化完毕。

```
<script>
function init(){
    if(perfLogger){
        perfLogger.showPerformanceMetrics()
    }
}
</script>
```

完成后的代码如下所示：

```
<script src="/lab/remoteLoader.js"></script>
<script>
function init(){
    if(perfLogger){
        perfLogger.showPerformanceMetrics()
    }
}

if (window.attachEvent)
    window.attachEvent('onload', remoteLoader.loadJS("/lab/perfLogger.js"), init);
else
    window.addEventListener('load', remoteLoader.loadJS("/lab/perfLogger.js", init), false);
</script>
```

2. 分析结果并用图表来展示

如果你将上述代码应用在一个生产环境中，并从终端用户那里收集数据，那么你就能够对这个页面的度量值进行可视化处理，并同其他的方法进行比较了。

要做到这一点，我们需要修改 R 脚本，比较一下我们的惰性加载示例和以前的例子。

在 comparePerfMetricsbyURL() 函数中，添加一个新的数据帧用于保存新的 URL：

```
lazy <- getDFByURL(perflogs, "http://tom-barker.com/lab/lazyloadscript.html")
```

然后，将这个新的变量囊括进 meanRenderTimes 数据帧和 meanLeadTimes 数据帧中：

```
meanRenderTimes <- data.frame(mean(baseline$PageRenderTime), mean(scripted$PageRenderTime),
mean(async$PageRenderTime), mean(lazy$PageRenderTime))

colnames(meanRenderTimes) <- c("Baseline", "Script Loaded", "Async", "Lazy Loaded")
meanLoadTimes <- data.frame(mean(baseline$PerceivedLoadTime), mean(scripted$PerceivedLoadTime),
mean(async$PerceivedLoadTime),mean(lazy$PerceivedLoadTime))

colnames(meanLoadTimes) <- c("Baseline", "Script Loaded", "Async", "Lazy Loaded")
```

修改后的 comparePerfMetricsbyURL() 函数如下所示：

```
comparePerfMetricsbyURL<-function(){
    baseline <- getDFByURL(perflogs, "http://tom-barker.com/lab/baseline.html")
    scripted <- getDFByURL(perflogs, "http://tom-barker.com/lab/scriptloaded.html")
    async <- getDFByURL(perflogs, "http://tom-barker.com/lab/asyncloaded.html")
    lazy <- getDFByURL(perflogs, "http://tom-barker.com/lab/lazyloadscript.html")


    meanRenderTimes <- data.frame(mean(baseline$PageRenderTime), mean(scripted$PageRenderTime),
mean(async$PageRenderTime), mean(lazy$PageRenderTime))
    colnames(meanRenderTimes) <- c("Baseline", "Script Loaded", "Async", "Lazy Loaded")
```

```
    meanLoadTimes <- data.frame(mean(baseline$PerceivedLoadTime), mean(scripted$PerceivedLoadTi
me), mean(async$PerceivedLoadTime),mean(lazy$PerceivedLoadTime))
    colnames(meanLoadTimes) <- c("Baseline", "Script Loaded", "Async", "Lazy Loaded")

    barplot(as.matrix(meanRenderTimes), main="Average Render Time\nBy Test Type", ylim=c(0,
400), ylab="milliseconds")
    barplot(as.matrix(meanLoadTimes), main="Average Load Time\nBy Test Type", ylim=c(0, 700),
ylab="milliseconds")
}
```

我们来看看上述代码生成的图表，如图 6-14 和图 6-15 所示。

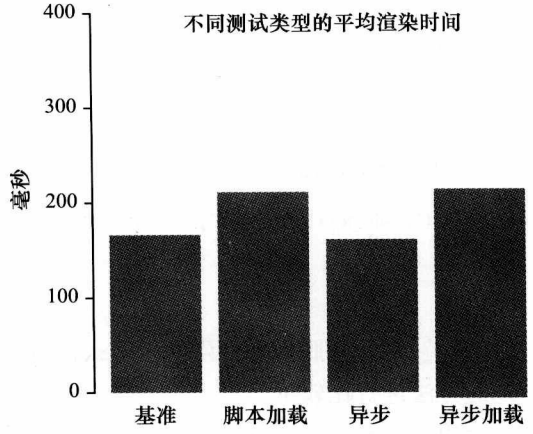


图 6-14 惰性加载的平均渲染时间

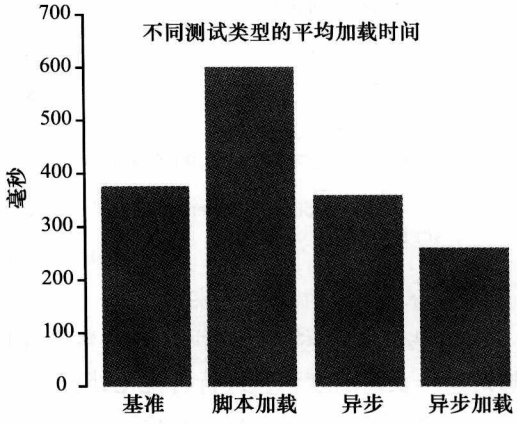


图 6-15 惰性加载的平均加载时间

图 6-14 显示的是平均渲染时间。在第 5 章中，我们计算渲染时间时用 Date.now() 减去 performance.timing.domLoading，当时 domLoading 表示文档（document）开始加载的时间。这意味着，当使用惰性加载时，这个度量值实际上就没什么意义了，因为我们要等到 document 加载完成之后才开始执行惰性加载，所以 Date.now() 被延迟到脚本文件完成异步加载的时间点了。

图 6-15 渲染了真实情况的场景。它绘制的时间跨度是从浏览器发起请求一直到页面加载完成，包括惰性加载脚本完成，以及 init 函数被调用。这是一个完整加载时间的最真实的描绘，从这里我们能够看到，使用惰性加载，我们获得了非常大的收益。其结果是平均加载时间比基准测试和异步测试快了 100 毫秒，比脚本加载测试快了 350 毫秒。

注意

我们不能仅仅是机械地观察我们的图表，然后就去宣布某个方法是胜利者。我们要考虑图表所要告诉我们的上下文环境，我们还要考虑图表中数据的整体情况，并让这些数据对我们有真正的实际意义。

我们来看看这个测试在 WebPagetest 中的情况。测试 URL 和测试结果 URL 如下所示：

用于测试的 URL	测试结果 URL
tom-barker.com/lab/lazyloadscript.html	http://www.webpagetest.org/result/120718_7X_d30b018b195bed1954d93baf25570f92/

在图 6-16 中，我们把原始数值与之前的测试结果比较一下，就能看到改进情况。我们的重复视图加载时间（repeat view load time）和重复视图文档完成时间比以往任何一个测试都要快。但是，真正取得胜利的是在重复视图文档完全加载时间（repeat view document Fully Loaded time）上，比异步测试时间快了 2 秒，比脚本加载测试快了 500 毫秒，比基准测试快了 100 毫秒。

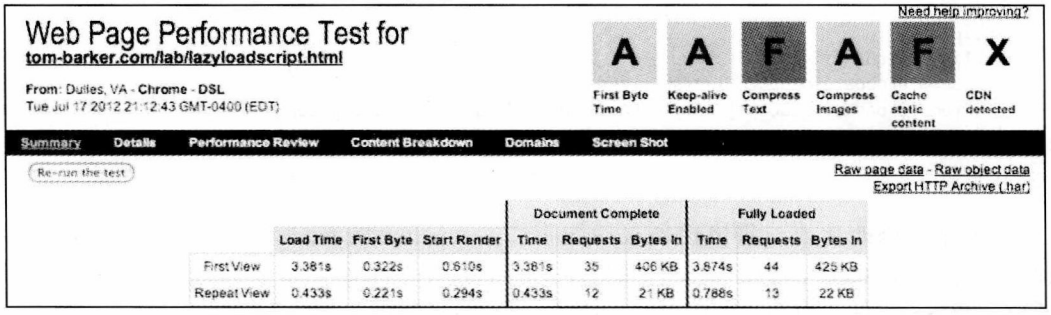


图 6-16 惰性加载测试的 WebPagetest 汇总结果（注意 First View 和 Repeat View 这两行）

从图 6-17 中，你可以看到，在脚本加载示例中，我们没有连接的初始时间开销，但是在惰性加载测试中，下载时间非常短，总时间只有 113 毫秒，而相应的基准测试为 534 毫秒，脚本加载测试为 393 毫秒。

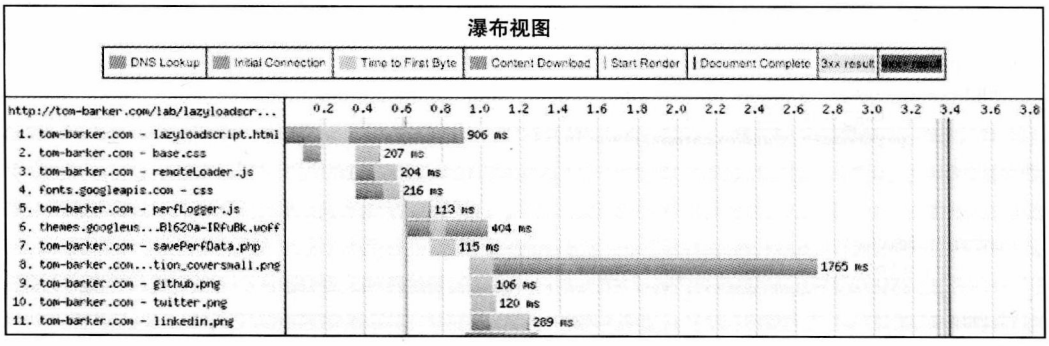


图 6-17 惰性加载测试的 WebPagetest 瀑布视图

很明显，惰性加载脚本是优化加载时间的有效途径。我们只需要确保在回调函数的后续代码中使用这些脚本。同时，我们要确保在页面加载之前，不要执行这些脚本。

6.2.3 惰性加载 CSS

我们进展得很顺利，看到了惰性加载带来的诸多好处，所以让我们继续吧！我们在上

述使用惰性加载完成外部 JavaScript 文件加载的成功经验的基础上，对 CSS 做同样的事情。一会儿我们会看到结果。

1. 设置 CSS 惰性加载

首先，将我们的惰性加载页面另存为一个新的文件，lazyloadcss.html。我们仍保持 JavaScript 文件执行惰性加载，并且我们会对这个文件进行补充，仍然让 remoteLoader.js 文件来处理惰性加载 CSS 文件。

在 lazyloadcss.html 中创建一个函数 fetch()。该函数将保存对 remoteLoader.loadJS() 的调用。同样，先使用我们马上就会定义的 loadCSS() 函数（以存根方式来完成调用）。完整的 fetch 函数如下所示：

```
<script>
function fetch(){
     remoteLoader.loadJS("/lab/perfLogger.js", init)
     remoteLoader.loadCSS(["/style/base.css", "http://fonts.googleapis.com/css?family=Metrophobi
c&amp;amp;v2"])
}
</script>
```

接下来，修改代码，将惰性加载功能附加给 window 的 load 事件。将对 remoteLoader 的调用替换为对 fecth 函数的调用。我们使用 fetch() 函数来惰性加载所有的 JavaScript 代码和所有的 CSS 文件，包括我们的 Web 字体。

```
<script>
if (window.attachEvent)
     window.attachEvent('onload', fetch);
else
     window.addEventListener('load', fetch, false);
</script>
```

lazyloadcss.html 修改后的 JavaScript 代码如下所示：

```
<script>
function init(){
     if(perfLogger){
          perfLogger.showPerformanceMetrics()
     }
}

function fetch(){
     remoteLoader.loadJS("/lab/perfLogger.js", init)
     remoteLoader.loadCSS(["/style/base.css", "http://fonts.googleapis.com/css?family=Metrophobi
c&amp;amp;v2"])
}

if (window.attachEvent)
     window.attachEvent('onload', fetch);
else
     window.addEventListener('load', fetch, false);
</script>
```

接下来，修改 remoteLoader 文件。首先重命名 constructScriptTag 函数为 constructTag 函数，这样看起来更具有通用性，然后传递第三个参数，指定你要构造的标签的类型：

```
function constructTag(src, func, type){
}
```

在 constructTag 函数中，你需要先创建一个变量，用于保存你创建的标签，然后依据标签的类型完成分支逻辑代码，标签的类型（type）指明了是处理 JavaScript 还是处理 CSS。在完成了 if-else-if 语句之后，返回 el。

```
function constructTag(src, func, type){
     var el;
     if(type === "JS"){

     }}else if(type==="CSS"){

}
return el;
}
```

在 JavaScript 分支，仍然使用原来的 constructScripTag() 函数中的逻辑，但是要重新使用一个新的 el 变量。记住，这段代码用于构造一个脚本标签，设置 src 属性，并为回调赋值。

```
if(type === "JS"){
el = window.document.createElement('SCRIPT');
     el.src = src;
     if(func){
          el.onload  = func;
     }
}
```

在 CSS 分支中，构造一个链接元素，设置其 type 属性，设置其 rel，最后设置它的 href，以指向传递进来的 CSS 文件：

```
else if(type==="CSS"){
el = document.createElement('link');
     el.type = 'text/css';
     el.rel = 'stylesheet';
     el.href = src
}
```

现在，你需要将所有的函数从 loadJS 中提取出来，添加到我们自己的函数中，我们称这个函数为 processURLs。传递相同的参数，然后增加一个 type 参数，用于为 constructTag 中的 type 参数赋值。

```
function processURLs(script_url, f, type){
     if(typeof script_url === "object"){
          if(script_url instanceof Array){
               var frag = document.createDocumentFragment();
               for(var ind = 0; ind < script_url.length; ind++){
                    frag.appendChild(constructTag(script_url[ind]), f, type);
               }
window.document.getElementsByTagName('HEAD')[0].appendChild(frag.cloneNode(true));
          }
     }else if(typeof script_url === "string")}
window.document.getElementsByTagName('HEAD')[0].appendChild(constructTag(script_url, f, type))
     }
}
```

最后，添加加载语句：

```
loadCSS:function(script_url){
processURLs(script_url, null, "CSS")
},

loadJS: function(script_url, f){
processURLs(script_url, f, "JS")
}
```

修改后的 remoteLoader 文件现在如下所示：

```
var remoteLoader = function remoteLoader(){
     function constructTag(src, func, type){
          var el;
          if(type === "JS"){
               el = window.document.createElement('SCRIPT');
               el.src = src;
               if(func){
                    el.onload  = func;
                    }
          }else if(type==="CSS"){
               el = document.createElement('link');
               el.type = 'text/css';
               el.rel = 'stylesheet';
               el.href = src
          }
          return el;
     }

     function processURLs(script_url, f, type){
          if(typeof script_url === "object"){
               if(script_url instanceof Array){
                    var frag = document.createDocumentFragment();
                    for(var ind = 0; ind < script_url.length; ind++){
                         frag.appendChild(constructTag(script_url[ind]), f, type);
                    }
                    window.document.getElementsByTagName('HEAD')[0].appendChild(frag.
cloneNode(true));
               }
          }else if(typeof script_url === "string"){
          window.document.getElementsByTagName('HEAD')[0].appendChild(constructTag(script_url,
f, type))
          }
     }

     return{
          loadCSS:function(script_url){
               processURLs(script_url, null, "CSS")
          },

          loadJS: function(script_url, f){
               processURLs(script_url, f, "JS") V
          }
     }
}();
```

2. 分析并可视化

太棒了！现在，让我们将测试页放到 WebPagetest 中（见图 6-18）。我们的 URL 如下所示：

用于测试的 URL	测试结果 URL
http://tom-barker.com/lab/lazyloadcss.html	http://www.webpagetest.org/result/120719_CP_94a98c18918b49d36912378ffc5d435f/

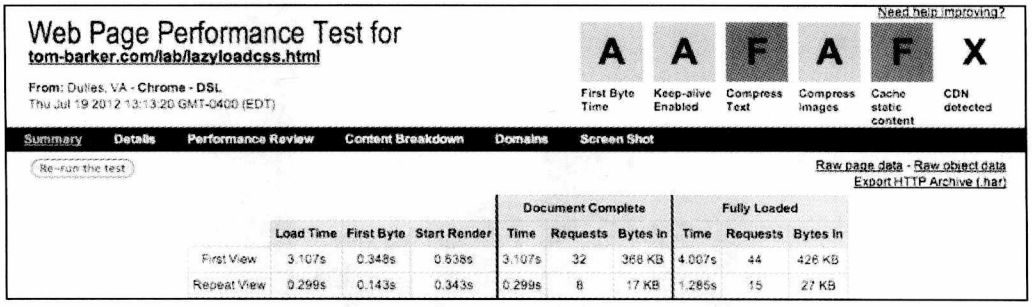


图 6-18 惰性加载的 WebPagetest 汇总结果

看看这些结果！在首次视图和重复视图上，我们的加载时间以及渲染引擎启动时间快了大概 200 毫秒。我们的文档完成时间取得了几乎相同的效果。图 6-19 所示的瀑布图可以帮助我们分析到底为什么会有这样的效果。

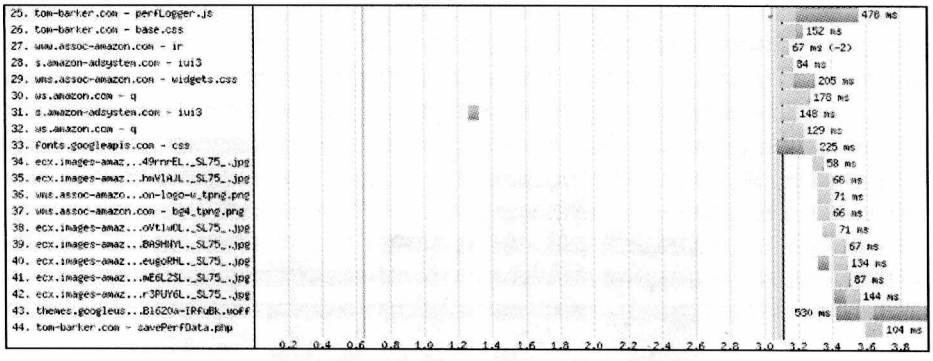


图 6-19 惰性加载测试的 WebPagetest 瀑布图

从瀑布图上，我们能够看到，perfLogger.js、base.css 以及我们的 Web 字体都被延迟到页面加载完成之后再加载，对应内容看图 6-19 中的第 25、26、33 和 34 行。

这意味着，页面真正的内容会先被加载，页面在任务最繁重的表示层内容加载之前就已经可用，并对于用户来说有效。这无疑是一个成功！

我们来看看 Navigation 的性能是否也遵循这个规律（参见图 6-20 和图 6-21）。

为什么会这样？很明显，惰性加载改变了我们的定时数据，因为事件发生的顺序被改变了。但是在这里我们可以看到，甚至是我们用于收集度量值的渲染时间也被惰性加载 CSS 和

JavaScript 给丢弃掉了。

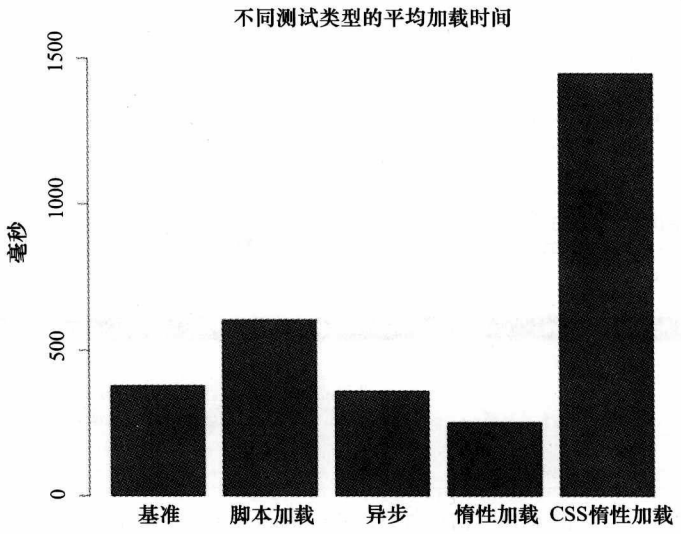


图 6-20 每种测试类型的平均加载时间比较

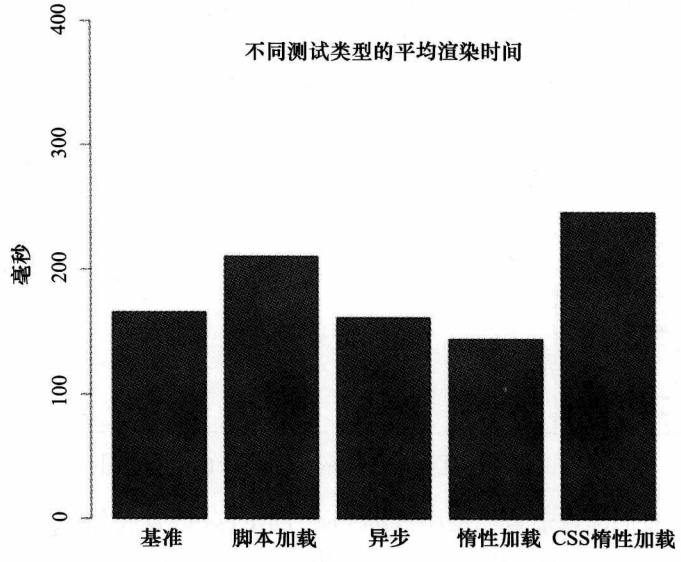


图 6-21 每种测试类型的平均页面渲染时间比较

我们可以通过不执行惰性加载 perfLogger.js 来纠正这一点。如果我们将 perfLogger 的加载恢复到以前的内联调用的方式，则会看到如图 6-22 和图 6-23 所示的数据：

这样更好！通过内嵌 perfLogger 中的代码，我们就不会因为加载和执行 perfLogger 而产生延迟了，因此我们的方法不会影响到我们的数据记录。

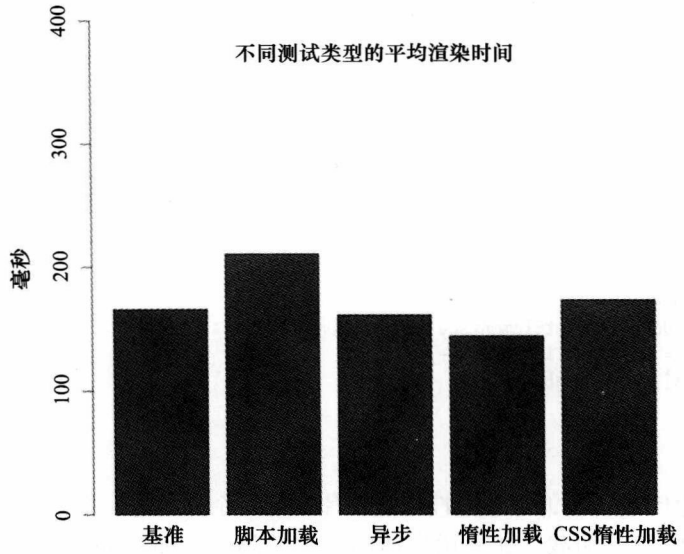


图 6-22 不执行 perfLogger 惰性加载的平均页面渲染时间结果

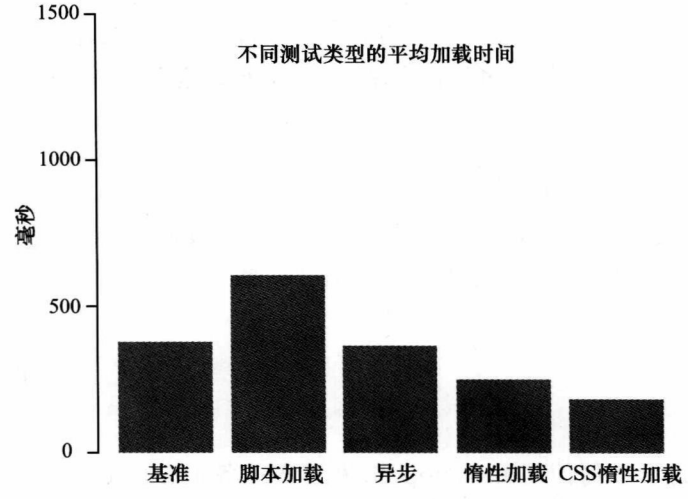


图 6-23 不执行 perfLogger 惰性加载的平均页面加载时间结果

这一课太有价值了——我们一直以来小心翼翼地防止我们的度量值不被获取它们的方法所修改。唯一的办法就是对数据进行全面的分析，而不是只看表面结果。

6.2.4 为什么不惰性加载图片

到现在，本章对惰性加载做了深入的探讨，你可能会试图尽可能地使用惰性加载。但是，惰性加载并不是万能的良药。你需要确定，某些内容是否需要加载到页面中。你也需要理解，当 JavaScript 关闭了，你的页面功能会怎样。

如果我们要惰性加载图片，我们需要变更页面的 HTML，删除每一个 image 的 src 属性的内容。我们可以只是将 src 属性的内容移植（move）到我们自己设计的一个 image 标签的属性中，这个属性可能是 rel 属性。这样就可以阻止图片的下载了：

```
<img src="#" rel="[path to image]" />
```

然后，在加载时，使用 JavaScript 迭代页面上的所有的图片标签，将每一个 src 属性的内容赋给新属性：

```
<script>
function lazyloadImages(){
     var img = document.getElementsByTagName("img");
     for(var x = 0; x<img.length; x++){
          img[x].src = img[x].rel;
     }
}
</script>
```

但是有几种原因或情况，导致惰性加载图片不是一个好主意，主要是因为根本就不能用 JavaScript 来阻止图片的加载。如果用户访问一个关闭了 JavaScript 的网站，他们根本就看不到任何的图像。如果网站上使用了第三方的广告，有可能会导致第三方的 JavaScript 代码出现错误，并阻止所有图片加载。当搜索引擎爬虫索引我们的页面时，它们没有机会看到我们页面上的图片，因此它们会以无图片的方式缓存我们的页面，然后以无图片的方式显示页面。最重要的一个原因是，使用 rel 属性来保存图片路径，这样使用标记在语义上是错误的。

注意，在 HTML 的上下文环境中，语义正确意味着在使用标签时，保留标签名称的意义。我们使用 HTML 标签，是因为标签指明了其所包含的数据是什么，而不是因为标签所带来的页面视觉装饰效果。如果我们想有意义地使用标签——比如使用 <p> 标签是为了表示分段，而不是为了增加间距，而外部的应用程序，比如搜索引擎或屏幕读取软件，访问我们的页面时，就能够正确地解析这些页面上的信息。而且，这样我们也可以将内容与视觉表示分离开来。

6.3 小结

本章详细探讨了如何使用 JavaScript 来改进 Web 各个方面的性能，特别研究了浏览器如何解析和渲染内容，以及这一过程中存在的可能的瓶颈。我们看到了由于下载外部 JavaScript 文件而导致其他内容被阻塞这一问题的解决办法，包括使用新的 async 属性，以及通过编写程序创建脚本标签。

我们探讨了惰性加载设计模式，并将该模式应用于外部脚本，使其在页面其他内容加载完成之后再执行外部 JavaScript 文件加载。我们将惰性加载扩展到 CSS 文件的加载上，并且考虑了如何才能将惰性加载应用到图片文件上来，以及不建议将惰性加载应用到图片上的原因。

每一个例子里边，我们都是用我们自己开发的工具和现有工具来分析并可视化我们的结果。

第 7 章，我们将要看看如何改进运行时性能。

第 7 章
运行时性能

第 6 章研究了使用 JavaScript 优化 Web 性能。学习了几种避免由于外部 JavaScript 导致页面其他内容解析被阻塞的方法。我们了解了 async 属性，这也是 HTML5 新增的特性，以及使用 JavaScript 绘制脚本标签。

你学习了惰性加载模式，并应用在了我们示例的脚本标签上，使其只有在页面加载完成之后才可被添加进来。我们扩展了惰性加载的思路，应用到了 CSS 上，确保当文档加载完成之后，才惰性加载 CSS。

上述每一种情况我们都运行了测试，捕获了每一种情况下的数据，并且对结果进行了可视化处理和比较。

本章，我们要了解一下浏览器中运行时性能的优化。我们会学习在作用域链上使用缓存引用来节省查找时间。你还会学习到如何使用像 jQuery 这样的框架技术，并比较使用 jQuery 和纯 JavaScript 在操作运行时性能上的不同。甚至你还会对一个长期以来公认的概念——使用 eval() 函数会影响性能——进行量化评估。最后，本章会关注如何简化 DOM 交互，如何量化嵌套循环的增量成本。

记住，因为每个浏览器使用不同的 JavaScript 引擎，运行时性能值也可能因为浏览器而不同。因此，对于本章将要用到的例子，你需要先使用我们的 perfLogger 库编写测试代码，接下来在一个浏览器上查看 perfLogger 的运行结果，然后假定将代码放置在公众能够访问的地方，利用众包流量（crowd-source traffic，即公众访问的流量）对其进行测试，或者在我们浏览器实验环境中运行这段代码，在实验环境中可以使用不同的浏览器多次访问页面。第 8 章我会更详细地讨论这些产品上的变化是如何实现的。最后，我们会依据浏览器分组，对每次测试的结果进行汇总，并用图来表示它们之间的不同。

注意

某些测试的结果在亚毫秒范围内，因此，要得到最好的结果，我们要使用支持高分辨率时间的浏览器。还要牢记得是，测试结果与运行测试的浏览器或用户使用的浏览器密切相关。正如你在本章将要看到的测试，每一个浏览器在不同的领域执行效率是变化的。本章的

重点就是学习随着浏览器或用户配置变化，如何建立你自己的测试，如何分析测试的结果，如何基于终端用户连续不断地评估你的最佳测试实践。

7.1 跨作用域的缓存变量和属性

首先要进行优化的是缓存变量和对象属性。

我们来看看缓存变量和对象属性优化是什么意思。通常情况下，当我们在其他作用域（在全局范围内，或在其他命名空间等）中引用变量，解释器需要对栈进行转换来获取变量。

我们用 document.location 做示例。如果你在一个对象内引用 document.location，解析器就会从引用变量的函数中跳出来，跳出该函数的命名空间到达全局 window 作用域中，然后再进入 document 作用域，最后得到 location 的值。如图 7-1 所示。

栈的转换要基于解释器的效率以及你创建的实例的继承体系——如果命名空间有 4 到 5 个，甚至是 10 个嵌套对象的深度，那么就要花更长的时间才能到达全局作用域，然后再进入到所需变量所在的作用域内。

要解决这个问题，你可以创建一个本地变量来保存引用到的数值，这样以后就可以引用该本地变量，而不需要每次都进行栈转换了。你可以创建一个变量引用的快捷方式来提升运行时性能，如图 7-2 所示。

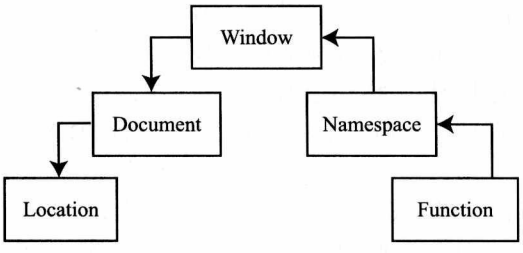


图 7-1 获取 document.location 而进行的栈变化情况追踪

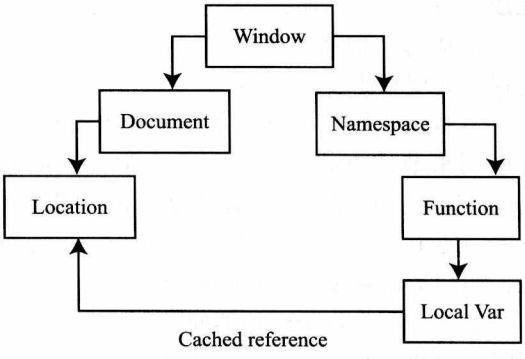


图 7-2 创建一个本地变量来缓存对 document.location 的引用

我们用一个例子来对这种性能提升进行量化处理。

7.1.1 新建文件

首先创建一个名为 cache_locationcomparision.html 的新文件，然后在该文件中编写 HTML 骨架结构，其中只包含 doctype、html、head、title、character set 以及 body 标签。该结构是我们测试的基础结构，它是本章后续每一个测试的开端。

```
<!DOCTYPE html>
<html>
```

```
<head>
<meta charset="UTF-8" />
<title>Cache Location Comparison</title>
</head>
<body>
</body>
</html>
```

在 head 中，添加我们的 perfLogger 以及一个 populateArray 函数，该函数用于创建并返回一个数组，数组的大小是通过参数传递给函数的。

```
<script src="/lab/perfLogger.js"></script>
<script>
function populateArray(len){
          var retArray = new Array(len)
          for(var i = 0; i < len; i++){
               retArray[i] = i;
          }
          return retArray
}
</script>
```

在 body 中，你要创建一个新的脚本标签，并实例化一个大小为 400 的新数组，该数组在后边的例子中会用到。

```
<script>
tempArray = populateArray(400);

</script>
```

7.1.2 创建测试

现在我们来完成设置，首先创建第一个测试。在这个测试中，为了引用 document.location，你要捕获特定的定时数据。首先调用 perfLogger.startTimeLogging 函数，传递“uncached_location”的 ID 值给它，并给出测试的描述，然后设置该测试，让其可在屏幕上显示并保存数据到日志文件中。参见第 4 章关于 perfLogger API 的描述。

```
// Capture ad hoc timing data for referencing uncached document.location
perfLogger.startTimeLogging("uncached_location", "Capture ad hoc timing data for referencing
document.location", true, true, true)
```

调用 startTimeLogging 函数之后，紧接着你要运行测试代码。创建一个 for 循环，让其对 tempArray 数组（利用 tempArray.length）进行迭代，在每一次迭代中，设置一个 loc 变量，其值为 document.location：

```
for(var i = 0; i < tempArray.length; i++){
     var loc = document.location;
}
```

最后，循环完成之后，调用 perfLogger.stopTimeLogging 函数，向其传递“uncached_location”的 ID 值，停止测试，在屏幕上显示测试结果并将日志保存到服务器上：

```
perfLogger.stopTimeLogging("uncached_location");
```

完整的测试代码如下所示：

```
// Capture ad hoc timing data for referencing uncached document.location
perfLogger.startTimeLogging("uncached_location", "Capture ad hoc timing data for referencing
document.location", true,true, true)
for(var i = 0; i < tempArray.length; i++){
    var loc = document.location;
}
perfLogger.stopTimeLogging("uncached_location");
```

如果你在浏览器中查看结果，就会看到测试输出如下所示：

```
Capture ad hoc timing data for referencing document.location
run time: 0.2339999999749125ms
path: http://tom-barker.com/lab/cache_locationcomparison.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
Perceived Time: 40370
Redirect Time: 0
Cache Time: 0
DNS Lookup Time: 0
tcp Connection Time: 0
roundTripTime: 12636
pageRenderTime: 107
```

让我们进行另一个测试，这一次对刚才的 for 循环进行基准测试。首先，将 for 循环打包成一个函数，取名为 uncachedLoc()：

```
//benchmark timing data for uncached document.location
function uncachedLoc(){
    for(var i = 0; i < tempArray.length; i++) {
     var loc = document.location;
    }
}
```

然后调用 perfLogger.logBenchmark 函数，向其传递“LocationUnCached_benchmark”的 ID 值。通知该函数返回参数引用的函数 10 次，然后将其作为基准测试函数传递给 logBenchmark，最后将结果显示在页面上，并将日志记录到服务器上：

```
perfLogger.logBenchmark("LocationUnCached_benchmark", 10, uncachedLoc, true, true);
```

完成后的测试代码如下所示：

```
//benchmark timing data for uncached document.location
function uncachedLoc(){
    for(var i = 0; i < tempArray.length; i++) {
     var loc = document.location;
    }
}
perfLogger.logBenchmark("LocationUnCached_benchmark", 10, uncachedLoc, true, true);
```

好了，以上这两个测试是后边的基础，它们直接引用了 document.location。你现在要做的就是再创建两个测试，用缓存对 document.location 的引用进行优化。

首先创建一个变量 l，该变量用于保存对 document.location 的引用：

```
var l = document.location;
```

然后重建上文创建的基础测试逻辑，将 loc 赋值为 l，而不是上文指向的 document.location。完成后的测试代码如下所示：

```
// Capture ad hoc timing data for referencing cached document.location
var l = document.location;
perfLogger.startTimeLogging("cached_location", "Capture ad hoc timing data for referencing
document.location", true,true, true)
for(var i = 0; i < tempArray.length; i++){
     var loc = l;
}
perfLogger.stopTimeLogging("cached_location");
```

最后，进行缓存基准测试。正如前边做的，创建一个名为 cacheLoc 的函数，在其中创建一个变量 l 指向 document.location，然后再循环中引用该变量。完整的测试代码如下所示：

```
//benchmark timing data for cached document.location

function cachedLoc(){
     var l = document.location;
     for(var i = 0; i < tempArray.length; i++){
          var loc = l;
     }
}

perfLogger.logBenchmark("LocationCached_benchmark", 10, cachedLoc, true, true);
```

完整的 cache_locationcomparision.html 代码如下所示：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Cache Location Comparison</title>
<script src="/lab/perfLogger.js"></script>
<script>
function populateArray(len){
          var retArray = new Array(len)
          for(var i = 0; i < len; i++){
               retArray[i] = i;
          }
          return retArray
}
</script>
</head>
<body>
<script>
tempArray = populateArray(400);

// Capture ad hoc timing data for referencing uncached document.location
perfLogger.startTimeLogging("uncached_location", "Capture ad hoc timing data for referencing
document.location", true,true, true)
for(var i = 0; i < tempArray.length; i++){
     var loc = document.location;
}
perfLogger.stopTimeLogging("uncached_location");

//benchmark timing data for uncached document.location
```

```
function uncachedLoc(){
     for(var i = 0; i < tempArray.length; i++) {
      var loc = document.location;
     }
}

perfLogger.logBenchmark("LocationUnCached_benchmark", 10, uncachedLoc, true, true);


// Capture ad hoc timing data for referencing cached document.location
var l = document.location;
perfLogger.startTimeLogging("cached_location", "Capture ad hoc timing data for referencing
document.location", true,true, true)
for(var i = 0; i < tempArray.length; i++){
     var loc = l;
}
perfLogger.stopTimeLogging("cached_location");


//benchmark timing data for cached document.location

function cachedLoc(){
     var l = document.location;
     for(var i = 0; i < tempArray.length; i++){
          var loc = l;
     }
}

perfLogger.logBenchmark("LocationCached_benchmark", 10, cachedLoc, true, true);


</script>
</body>
</html>
```

当你在浏览器中查看该页面时，你会看到如下输出内容（为了清晰起见，这里略去了性能导航数据）：

```
Capture ad hoc timing data for referencing document.location
run time: 0.23800000781193376ms
path: http://localhost:8888/lab/chapter7/cache_locationcomparison.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11

Benchmarking function uncachedLoc(){ for(var i = 0; i < tempArray.length; i++) { var loc =
document.location; }      }
average run time: 0.08210000523831695ms
path: http://localhost:8888/lab/chapter7/cache_locationcomparison.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11

Capture ad hoc timing data for referencing document.location
run time: 0.027000001864507794ms
path: http://localhost:8888/lab/chapter7/cache_locationcomparison.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
```

```
Benchmarking function cachedLoc(){      var l = document.location; for(var i = 0; i < tempArray.
length; i++){ var loc = l; }      }
average run time: 0.012399995466694236ms
path: http://localhost:8888/lab/chapter7/cache_locationcomparison.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
```

太棒了！从特定测试和基准测试中，通过缓存作用域链的引用，我们的性能获得了明显的改进。

7.1.3 结果可视化

现在，我们用 R 来对结果进行可视化处理。从 perfLogger 工程中打开 runtimePerformance.R。记住，在这个文件中，我们已经将 runtimeperf_results 平面文件读入到了一个名为 perflogs 的变量中了。

所以，我们开始创建一个函数，该函数可接收测试的 ID 值并返回一个 perflogs 的子集，该子集中有一个 TestID 值与传递进来的值相匹配。将这个函数命名为 ParseResultsbyTestID()。你可以创建一个变量来指向该函数，以保存所有测试的数据帧。

```
ParseResultsbyTestID <- function(testname){
    return(perflogs[perflogs$TestID == testname,])
}
```

在 runtimePerformance 中，已经有了 getDFByBrowser() 函数。你也可以使用这个函数。记住，这个函数返回传递进来的数据帧的子集，该数据帧有一个 UserAgent 列，其中包含了传递进来的浏览器名称。

```
getDFByBrowser<-function(data, browsername){
    return(data[grep(browsername, data$UserAgent),])
}
```

接下来，创建一个 avg_loc_uncache_chrome 变量，用于保存 Chrome 浏览器上运行的未缓存测试的平均运行时间。为了得到该值，你需要调用 ParseResultsbyTestID 函数，将 ID 值“LocationUnCached_benchmark”传递给该函数。然后将函数的运算结果以及字符串“Chrome”传递给 getDFByBrowser 函数。

```
avg_loc_uncache_chrome <- mean(getDFByBrowser(ParseResultsbyTestID("LocationUnCached_
benchmark"), "Chrome")$RunTime)
```

如果在控制台上打印该值，你会看到如下结果：

```
> avg_loc_unucache_chrome
[1] 1.75
```

很好！接下来，在 Firefox 上做同样的事情，然后再对 Chrome 和 Firefox 这两个浏览器（以及其他你想要追踪其性能的浏览器）做同样的“LocationCached_benchmark”测试。

```
avg_loc_cache_chrome <- mean(getDFByBrowser(ParseResultsbyTestID("LocationCached_benchmark"),
"Chrome")$RunTime)

avg_loc_uncache_firefox <- mean(getDFByBrowser(ParseResultsbyTestID("LocationUnCached_
benchmark"), "Firefox")$RunTime)
```

```
avg_loc_cache_firefox <- mean(getDFByBrowser(ParseResultsbyTestID("LocationCached_benchmark"),
"Firefox")$RunTime)
```

太棒了。现在每个浏览器上的每一个测试都有了数学含义。创建一个数据帧保存所有的内容，并为数据帧的每一列赋一个有意义的名字。

```
location_comparison <- data.frame(avg_loc_uncache_chrome, avg_loc_cache_chrome, avg_loc_uncache_
firefox, avg_loc_cache_firefox)

colnames(location_comparison) <- c("Chrome\nUncached", "Chrome\nCached", "Firefox\nUncached",
"Firefox\nCached")
```

最后，使用数据帧创建一个柱状图：

```
barplot(as.matrix(location_comparison), main="Comparison of average benchmark time in
milliseconds")
```

本例中你需要的完整的 R 代码如下所示：

```
ParseResultsbyTestID <- function(testname){
    return(perflogs[perflogs$TestID == testname,])
}

getDFByBrowser<-function(data, browsername){
    return(data[grep(browsername, data$UserAgent),])
}

avg_loc_uncache_chrome <- mean(getDFByBrowser(ParseResultsbyTestID("LocationUnCached_
benchmark"), "Chrome")$RunTime)

avg_loc_cache_chrome <- mean(getDFByBrowser(ParseResultsbyTestID("LocationCached_benchmark"),
"Chrome")$RunTime)

avg_loc_uncache_firefox <- mean(getDFByBrowser(ParseResultsbyTestID("LocationUnCached_
benchmark"), "Firefox")$RunTime)

avg_loc_cache_firefox <- mean(getDFByBrowser(ParseResultsbyTestID("LocationCached_benchmark"),
"Firefox")$RunTime)

location_comparison <- data.frame(avg_loc_uncache_chrome, avg_loc_cache_chrome, avg_loc_uncache_
firefox, avg_loc_cache_firefox)

colnames(location_comparison) <- c("Chrome\nUncached", "Chrome\nCached", "Firefox\nUncached",
"Firefox\nCached")

barplot(as.matrix(location_comparison), main="Comparison of average benchmark time in
milliseconds")
```

以上 R 代码生成的图表如图 7-3 所示。

从这个例子以及我们自己的样本数据，我们可以看到，对于使用的客户端浏览器而言，不管是 Chrome 还是 Firefox，都有了明显的性能改进。Chrome 改进最明显，平均接近 2 毫秒的改进量。就其本身而言，改进还不太明显，但是在大规模环境下，我们就会看到它的价值。

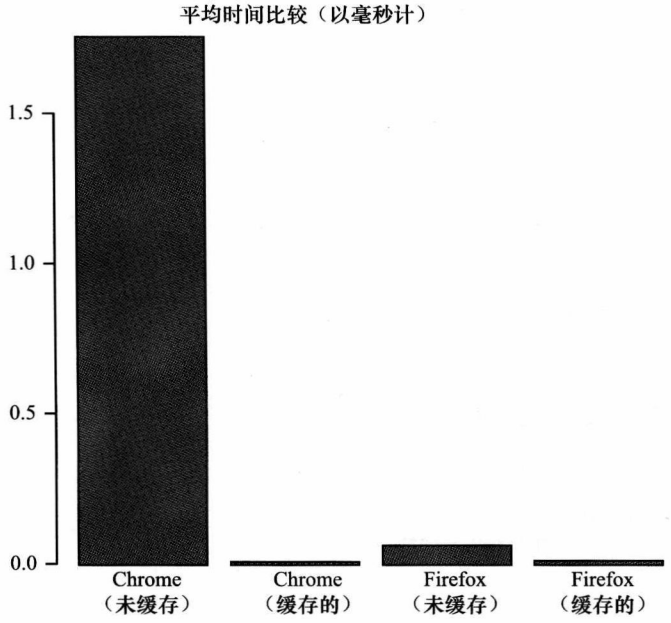


图 7-3 不同浏览器对作用域链引用进行缓存与未缓存的基准测试结果比较

7.1.4 属性引用示例

你已经看到了不同栈里边的缓存变量，现在，我们来看看缓存属性引用。测试这一技术的方法是对数组进行循环访问，以数组的长度作为结束循环的终止符，这是基础。你也可以将数组长度赋给一个本地变量，以该变量作为循环结束的终止符。如果你比较这两个技术，就应该能够评估出缓存属性引用所带来的好处了。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Loop Comparison</title>
<script src="/lab/perfLogger.js"></script>
<script>
function populateArray(len){
          var retArray = new Array(len)
          for(var i = 0; i < len; i++){
               retArray[i] = i;
          }
          return retArray
}
</script>
</head>
<body>
<script>
tempArray = populateArray(400);

// Capture ad hoc timing data for JavaScript for loop with uncached length variable
```

```
perfLogger.startTimeLogging("js_forloop_uncached", "Capture ad hoc timing data for JavaScript
for loop with uncached length variable", true,true, true)
for(var i = 0; i < tempArray.length; i++) {

}
perfLogger.stopTimeLogging("js_forloop_uncached");


// Capture ad hoc timing data for JavaScript for loop with cached length variable

perfLogger.startTimeLogging("js_forloop_cached", "Capture ad hoc timing data for JavaScript for
loop with cached length variable", true,true, true)
var l = tempArray.length
for(var i = 0; i < l; i++) {

}
perfLogger.stopTimeLogging("js_forloop_cached");

//benchmark timing data for uncached length variable

function uncachedLen(){
     for(var i = 0; i < tempArray.length; i++) {

     }
}
perfLogger.logBenchmark("JSForLoopUnCached_benchmark", 10, uncachedLen, true, true);

//benchmark timing data for cached length variable

function cachedLen(){
     for(var i = 0; i < l; i++) {

     }
}
perfLogger.logBenchmark("JSForLoopCached_benchmark", 10, cachedLen, true, true);

</script>
</body>
</html>
```

上述代码生成的结果如下：

```
Capture ad hoc timing data for JavaScript for loop with uncached length variable
run time: 41.9360000232700 26ms
path: http://localhost:8888/lab/chapter7/loopcomparison.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11

Capture ad hoc timing data for JavaScript for loop with cached length variable
run time: 14.304999989690259ms
path: http://localhost:8888/lab/chapter7/loopcomparison.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11

benchmarking function uncachedLen(){ for(var i = 0; i < tempArray.length; i++) {  }      }
average run time: 29.685899993637577ms
path: http://localhost:8888/lab/chapter7/loopcomparison.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
```

```
Chrome/20.0.1132.47 Safari/536.11

benchmarking function cachedLen(){      for(var i = 0; i < l; i++) { }      }
average run time: 19.587300000187643ms
path: http://localhost:8888/lab/chapter7/loopcomparison.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
```

单次测试结果再次显示，缓存属性引用可明显改进性能。现在，尝试在多个浏览器上应用测试，并使用 R 生成结果图表。

为每一个测试 / 浏览器组合创建一个变量，用于保存 getDFByBrowser 函数调用返回的平均计算结果：

```
avg_loop_uncache_chrome <- mean(getDFByBrowser(ParseResultsbyTestID("JSForLoopUnCached_
benchmark"), "Chrome")$RunTime)

avg_loop_uncache_firefox <- mean(getDFByBrowser(ParseResultsbyTestID("JSForLoopUnCached_
benchmark"), "Firefox")$RunTime)

avg_loop_cache_chrome <- mean(getDFByBrowser(ParseResultsbyTestID("JSForLoopCached_benchmark"),
"Chrome")$RunTime)

avg_loop_cache_firefox <- mean(getDFByBrowser(ParseResultsbyTestID("JSForLoopCached_benchmark"),
"Firefox")$RunTime)
```

然后创建一个数据帧来聚合这些变量，并生成一个柱状图。

```
loop_comparison <- data.frame(avg_loop_uncache_chrome, avg_loop_cache_chrome, avg_loop_uncache_
firefox, avg_loop_cache_firefox)

colnames(loop_comparison) <- c("Chrome\nUncached", "Chrome\nCached", "Firefox\nUncached",
"Firefox\nCached")

barplot(as.matrix(loop_comparison), main="Comparison of average benchmark time \nfor cache and
uncached properties \nin milliseconds")
```

生成的图表如图 7-4 所示。

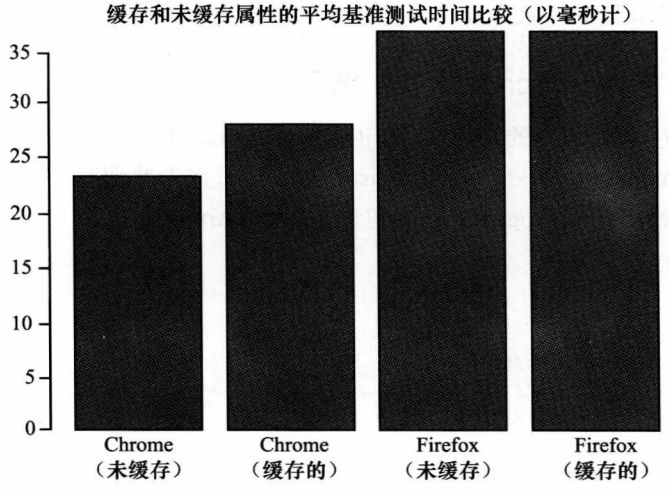


图 7-4 不同浏览器上，缓存属性引用与非缓存属性引用的基准测试结果比较

有意思的是，虽然单次测试能够发现对 Firefox 性能上的提升，但如果将基准测试的范围扩大，就会发现，对 Fireforx 的性能提升只是平均水平，而对 Chrome 性能的提升程度则超过 5 毫秒。由此，我们可以得出，在 Chrome 上大概能有 16.6% 的性能改进。

```
> (1 - (avg_loop_uncache_chrome / avg_loop_cache_chrome)) * 100
[1] 16.6
```

7.2 核心 JavaScript 与 Frameworks 的比较

过去的几年里，我注意到一件事情，就是不管在课堂上给新来的学生上课，还是在员工面试时，我都发现，他们都非常想了解如何使用框架（framework），通常是 jQuery，而不是想要学习核心 JavaScript。

这种想法是有问题的，因为不是每一件事情都能够通过框架来解决。框架是一个节约开发人员效率的工具，也就是说，它通过将代码所要完成的真实工作抽象化的方式，来达到简化代码编写的目的。这样做的好处是，除了用少量的 API 来封装大量的函数之外，还会封装大量的错误检查以及跨浏览器支持功能。但是，所有这些被封装的功能在 JavaScript 核心语言中都是有效的，而且有时其在核心语言中的运行速度更快，这是因为只做需要做的事情就可以了，不用考虑框架的所有用户所需要的功能。如果我们只是对框架有了解，我们就失去了编写我们需要的功能的灵活性，失去了使已有项目并为我所用的机会，也失去了最终创建新项目的灵活性。我们变成了功能的使用者和堆积者，而不是创造者。

这就有一个问题：在运行时性能方面，核心 JavaScript 比框架更有效么？从逻辑上讲，是这样的，但是让我们来研究几个例子，对 jQuery（目前最常使用的框架）进行基准评估。

注意

jQuery 是一个 JavaScript 框架，是 John Resig 于 2006 年创建的。2009 年 jQuery 的维护工作被 jQuery 项目接管。截至目前，jQuery 是最广泛使用的 JavaScript 框架。

7.2.1 jQuery 与 JavaScript 比较：循环

我们先来比较 JavaScript 的 for 循环与 jQuery 的 .each 语句所消耗的时间，创建一个新页面 jquerycomparison.html。在 jquerycomparison.html 中，你先要编写一个空的 HTML 骨架文档，其中包含 jQuery 和 perfLogger。还要包含 populateArray 函数：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Framework Comparison</title>
<script src="jquery.js"></script>
<script src="/lab/perfLogger.js"></script>
<script>
function populateArray(len){
          var retArray = new Array(len)
          for(var i = 0; i < len; i++){
```

```
            retArray[i] = i;
        }
        return retArray
}
</script>

</head>
<body>
</body>
</html>
```

接下来，创建一个 JQueryEach 函数，使用 jQuery.each 对 tempArray 数组进行循环。然后将该函数传递给 perfLogger.logBenchmark 函数进行基准评测。

```
function JQueryEach(){
    jQuery.each(tempArray, function(i, val) {
    });
}
perfLogger.logBenchmark("JQueryEach_benchmark", 10, JQueryEach, true, true);
```

最后，创建一个 JSForLoop 函数，利用前面提到的缓存的迭代长度（即 length）属性优化方法，使用 loop 对数组进行循环，并将该函数传递给 perfLogger.logBencharmark。

```
function JSForLoop(){
    var l = tempArray.length
    for(var i = 0; i < l; i++) {
    }
}
perfLogger.logBenchmark("JSForLoop_benchmark", 10, JSForLoop, true, true);
```

完整的页面代码如下：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Framework Comparison</title>
<script src="jquery.js"></script>
<script src="/lab/perfLogger.js"></script>
<script>
function populateArray(len){
        var retArray = new Array(len)
        for(var i = 0; i < len; i++){
            retArray[i] = i;
        }
        return retArray
}
</script>
</head>
<body>
<script>
tempArray = populateArray(400);

//benchmark timing data for JQuery .each loop
function JQueryEach(){
    jQuery.each(tempArray, function(i, val) {
    });
}
```

```
perfLogger.logBenchmark("JQueryEach_benchmark", 10, JQueryEach, true, true);


//benchmark timing data for JS for loop

function JSForLoop(){
     var l = tempArray.length
     for(var i = 0; i < l; i++) {
     }
}
perfLogger.logBenchmark("JSForLoop_benchmark", 10, JSForLoop, true, true);

</script>

</body>
</html>
```

当你使用浏览器打开该页面时，看到的结果如下：

```
benchmarking function JQueryEach(){ jQuery.each(tempArray, function(i, val) { });      }
average run time: 0.10279999987687916ms
path: http://tom-barker.com/lab/jquerycomparison.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
benchmarking function JSForLoop(){ var l = tempArray.length for(var i = 0; i < l; i++) { } }
average run time: 0.003599999830834567 55ms
path: http://tom-barker.com/lab/jquerycomparison.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
```

哇！区别的量级还是挺大的。我们用 R 来绘图看看。

跟前边的测试一样，创建变量以保存每个浏览器每次测试的平均值。

```
avg_jquery_loop_chrome <- mean(getDFByBrowser(ParseResultsbyTestID("JQueryEach_benchmark"),
"Chrome")$RunTime)

avg_for_loop_chrome <- mean(getDFByBrowser(ParseResultsbyTestID("JSForLoop_benchmark"),
"Chrome")$RunTime)

avg_jquery_loop_firefox <- mean(getDFByBrowser(ParseResultsbyTestID("JQueryEach_benchmark"),
"Firefox")$RunTime)

avg_for_loop_firefox <- mean(getDFByBrowser(ParseResultsbyTestID("JSForLoop_benchmark"),
"Firefox")$RunTime)
```

然后，创建一个数据帧来保存该平均值，并为数据帧的列赋名。

```
jquery_comparison <- data.frame(avg_jquery_loop_chrome, avg_for_loop_chrome, avg_jquery_loop_
firefox, avg_for_loop_firefox)

colnames(jquery_comparison) <- c("Chrome\nJQuery", "Chrome\nJavascript", "Firefox\nJQuery",
"Firefox\nJavaScript")
```

最后，用数据帧进行绘图：

```
barplot(as.matrix(jquery_comparison), main="Comparison of average benchmark time \nfor looping
in JQuery vs core JavaScript \nin milliseconds")
```

完成之后的 R 代码如下：

```
avg_jquery_loop_chrome <- mean(getDFByBrowser(ParseResultsbyTestID("JQueryEach_benchmark"),
"Chrome")$RunTime)

avg_for_loop_chrome <- mean(getDFByBrowser(ParseResultsbyTestID("JSForLoop_benchmark"),
"Chrome")$RunTime)

avg_jquery_loop_firefox <- mean(getDFByBrowser(ParseResultsbyTestID("JQueryEach_benchmark"),
"Firefox")$RunTime)

avg_for_loop_firefox <- mean(getDFByBrowser(ParseResultsbyTestID("JSForLoop_benchmark"),
"Firefox")$RunTime)

jquery_comparison <- data.frame(avg_jquery_loop_chrome, avg_for_loop_chrome, avg_jquery_loop_
firefox, avg_for_loop_firefox)

colnames(jquery_comparison) <- c("Chrome\nJQuery", "Chrome\nJavascript", "Firefox\nJQuery",
"Firefox\nJavaScript")

barplot(as.matrix(jquery_comparison), main="Comparison of average benchmark time \nfor looping
in JQuery vs core JavaScript \nin milliseconds")
```

创建的图表如图 7-5 所示：

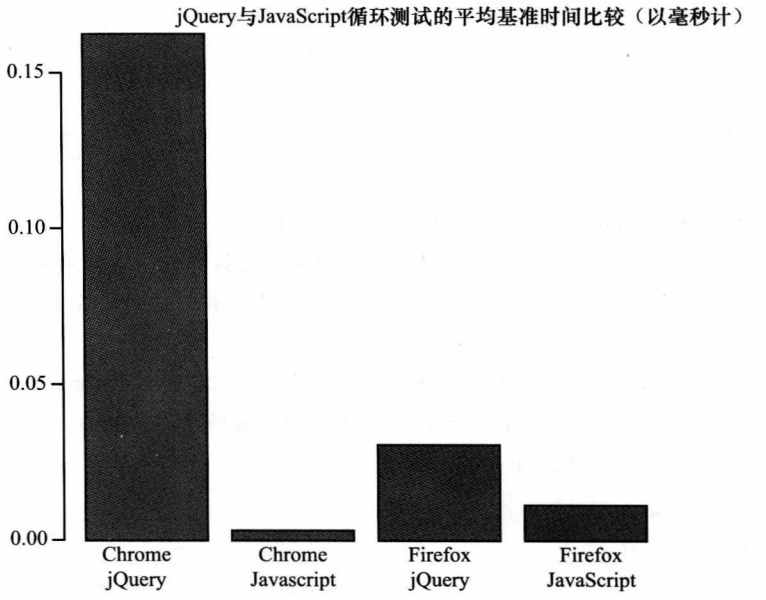


图 7-5 不同浏览器中 jQuery 与核心 JavaScript 循环基准测试结果比较

Firefox 的性能改进达到了 63.4%：

```
> (1 - (avg_for_loop_firefox / avg_jquery_loop_firefox)) * 100
[1] 63.4
```

而 Chrome 的改进则高达 98.2%。

```
> (1 - (avg_for_loop_chrome / avg_jquery_loop_chrome)) * 100
[1] 98.2
```

遵循 Dry 准则

我们来聊点别的话题。我不知道你怎么样，但是我厌倦了一遍一遍地重复编写相同的 R 代码，唯一的不同是获取基准测试结果和使用基准测试结果进行图表绘制。软件工程中有一个熟知的概念，叫做 DRY 准侧：Don't Repeat Yourself（不要重复做自己做过的事）。任何一件我们要做多次的事情，我们都应该将其自动化，这是人们非常原始、与生俱来的想法。在一本叫做《The Progmatic Programmer》（Addison-Wesley 出版社 1999 年出版）的书中，DRY 准则被提出并得到了推广，该书的作者是 Andy Hunt 和 Dave Thomas。

因此，我们遵循 DRY 准侧，对我们已经编写和重复编写的 R 代码做自动化处理，使其满足所有测试的需要。

编写一个函数 PlotResultsofTestsByBrowser，给它传递 3 个参数：testList，测试名向量；browserList，浏览器名向量；descr，图表头的字符串：

```
PlotResultsofTestsByBrowser <- function(testList, browserList, descr){

}
```

在该函数中，声明一个空的数据帧，用于保存平均值；还要声明一个空的列表，用于构造数据帧的列名称：

```
df <- data.frame()
colnameList <- c()
```

接下来，函数使用 browserList 循环访问每一个浏览器，并且在循环过程中，使用 testList 遍历每一个测试名。在内层循环中，代码为特定的测试 / 浏览器组合构造列名称，并将其传递到 colnameList 列表中。为了确保测试名能够适用于我们的图表，我们将测试名进行了截断处理，只保留 10 个字符。

调用 ParseResultsbyTestID 函数并将当前的测试名传递给它，然后将 ParseResultsbyTestID 以及当前浏览器名字传递给 getDFByBrowser 函数，将返回结果的平均值保存在一个临时变量 tmp 中。最后，将 tmp 添加到数据帧 df 中：

```
for(browser in browserList){
for(test in testList){
        colnameList <- c(colnameList, paste(browser, "\n", substr(test, 1,10)
))
        tmp <- mean(getDFByBrowser(ParseResultsbyTestID(test), browser)$RunTime)
        df <- rbind(df , tmp)
    }
}
```

该函数会访问每一个测试和浏览器组合，直到它构造出一个如下所示的数据帧为止：

```
> df
 X0.0050476189047619
1             0.00505
2             0.02979
3             0.00125
4             0.00716
```

非常好，这正是我们想要的。从这之后，你需要做的就只是转置（transpose）数据帧了。

```
df <- t(df)
```

将数据转置，转置后的输出如下所示：

```
> df
                          1      2       3       4
X0.0050476189047619 0.00505 0.0298 0.00125 0.00716
```

然后将数据帧列名赋值给你创建的图表的列名列表：

```
colnames(df) <- colnameList
barplot(as.matrix(df), main=descr)
```

完整的函数如下所示。从这里开始，后边都要通过生成图表来描绘测试结果。

```
PlotResultsofTestsByBrowser <- function(testList, browserList, descr){
     df <- data.frame()
     colnameList <- c()

     for(browser in browserList){
     for(test in testList){
               colnameList <- c(colnameList, paste(browser, "\n", test))
               tmp <- mean(getDFByBrowser(ParseResultsbyTestID(test), browser)$RunTime)
               df <- rbind(df , tmp)
          }
     }
     df <- t(df)
     colnames(df) <- colnameList
     barplot(as.matrix(df), main=descr)
}
```

7.2.2 jQuery 与 JavaScript 比较：DOM 访问

接下来，我们比较一下 jQuery 和纯 JavaScript 在与 DOM 进行交互时所花费的时间代

价。你还要创建一个测试，用循环访问数组的方式比较双方花费的时间，并将结果值写入到使用 jQuery 和 javaScript 页面的 div 标签中。

首先，创建页面的框架结构，将 jQuery、perfLogger 和 populateArray 函数包含进页面的 head 节中：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Framework Comparison</title>
<script src="jquery.js"></script>
<script src="/lab/perfLogger.js"></script>
<script>
function populateArray(len){
          var retArray = new Array(len)
          for(var i = 0; i < len; i++){
               retArray[i] = i;
          }
          return retArray
}
</script>
</head>
<body>
</body>
</html>
```

接下来，你要在页面中包含两个 div 标签，一个的 id 值为 DOMtest，用于编写 JavaScript 测试用例，另一个的 id 值为 JQueryDomtest，用于编写 jQuery 测试用例。

```
<div id="DOMtest"><p>Dom Test</p></div>
<div id="JQueryDomtest"><p>JQuery Dom Test</p></div>
```

然后，添加一个 script 标签，在其中创建一个大小为 400 的 tempArray 数组。同样，添加一个函数 JQueryDOM，用于循环该数组，并将索引值添加到 JQueryDomtest Div 标签中。最后，使用 perfLogger.logBenchmark 对该函数进行基准评估。

```
<script>
tempArray = populateArray(400);

function JQueryDOM(){
     var l = tempArray.length
     for(var i = 0; i < l; i++) {
          $("#JQueryDomtest").append(i);
     }
}
perfLogger.logBenchmark("JQueryDOM_benchmark", 10, JQueryDOM, true, true);
```

接下来，创建一个函数 JSDOM，用于对 tempArray 数组进行迭代，并使用 document.getElementById 访问 div 和 innerHTML，以便将迭代的结果添加到 div 中。最后，使用 perfLogger.logBenchmark 对该函数进行基准评估。

```
function JSDOM(){
     var l = tempArray.length
```

```
    for(var i = 0; i < l; i++) {
        document.getElementById("DOMtest").innerHTML += i;
    }
}
perfLogger.logBenchmark("JSDOM_benchmark", 10, JSDOM, true, true);

</script>
```

完整的页面代码如下所示：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Framework Comparison</title>
<script src="jquery.js"></script>
<script src="/lab/perfLogger.js"></script>
<script>
function populateArray(len){
          var retArray = new Array(len)
          for(var i = 0; i < len; i++){
               retArray[i] = i;
          }
          return retArray
}
</script>
</head>
<body>
<div id="DOMtest"><p>Dom Test</p></div>
<div id="JQueryDomtest"><p>JQuery Dom Test</p></div>
<script>
tempArray = populateArray(400);

//benchmark timing data for JQuery DOM access

function JQueryDOM(){
     var l = tempArray.length
     for(var i = 0; i < l; i++) {
          $("#JQueryDomtest").append(i);
     }
}
perfLogger.logBenchmark("JQueryDOM_benchmark", 10, JQueryDOM, true, true);


//benchmark timing data for JS DOM access

function JSDOM(){
     var l = tempArray.length
     for(var i = 0; i < l; i++) {
          document.getElementById("DOMtest").innerHTML += i;
     }
}
perfLogger.logBenchmark("JSDOM_benchmark", 10, JSDOM, true, true);

</script>
</body>
</html>
```

当你用浏览器访问该页面时，你会看到如下结果：

```
benchmarking function JQueryDOM(){ var l = tempArray.length for(var i = 0; i < l; i++) {
$("#JQueryDomtest").append(i); } }
average run time: 0.44930000014917925ms
path: http://tom-barker.com/lab/jquery_dom_compare.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11

benchmarking function JSDOM(){ var l = tempArray.length for(var i = 0; i < l; i++) { document.
getElementById("DOMtest").innerHTML += i; } }
average run time: 0.19499999471008778ms
path: http://tom-barker.com/lab/jquery_dom_compare.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
```

注意，对每个测试结果来说，使用的浏览器和系统不同，测试结果可能会不同。

我们使用 R 将结果绘制出来。对于我们的新函数而言，只需要一行代码就可以了，只需要将测试名列表、JQueryDOM_benchmark 和 JSDOM_benchmark 传递给函数就可以了。

```
PlotResultsofTestsByBrowser(c("JQueryDOM_benchmark", "JSDOM_benchmark"), c("Chrome","Firefox"),
"Comparison of average benchmark time \nfor using JQuery to access the DOM versus pure
JavaScript \nin milliseconds")
```

创建的柱状图如图 7-6 所示。

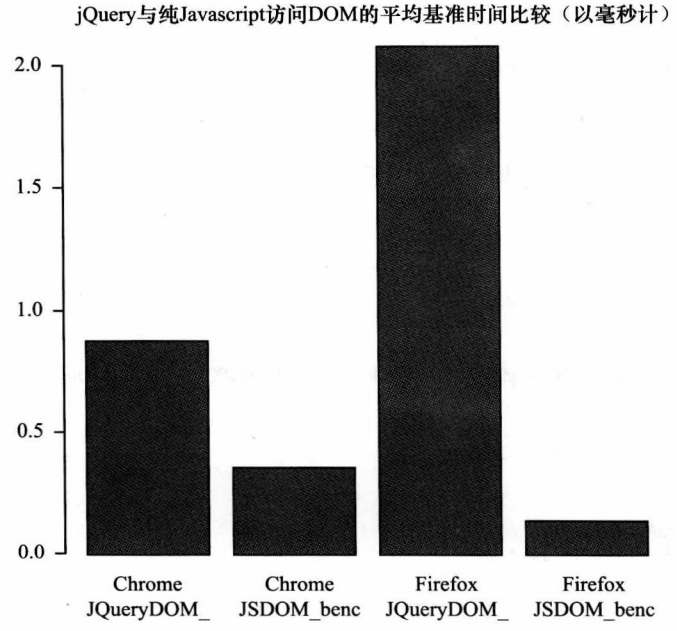


图 7-6 不同浏览器中 jQuery 和 JavaScript 与 DOM 交互结果的基准测试比较

仅仅是用 JavaScript 替换掉 jQuery，Chrome 的性能就有了 60% 的改进，而 Firefox 则达到了 93%。

7.3 Eval 函数的真正价值

也许你对 eval 函数还不熟悉，eval 是 JavaScript 的原生函数，它接收一个字符串作为参数并将其作为 JavaScript 代码来执行。它的基本原理是启动解释器，在 eval 函数被调用的时候对传入的字符串进行解析和解释。

我们以前都听说过 eval 函数有点“邪恶”，尽最大可能避免使用该函数[⊖]。这么说的原因是，eval 函数有潜在的危险性，因为它实质上是直接将代码注入给解释器，由于解释器是被访问的，因此会对性能造成冲击。过去的几年里，比较好的做法是避免使用 eval 函数。即使是在公认的甚至是司空见惯的场合，比如说反序列化 JSON 这样的情况下，我们现在都有常规的解决办法来代替使用 eval。

我的观点是，代码注入在某些时候是一个非常合法的解决办法，其历史可以追溯到在 C 代码中内嵌汇编语言来改进性能和实施对底端设备的控制。在浏览器中使用控制台，以便执行客户端的特定代码也是对“反对使用 eval 函数”这一观点的否定。

那么，我们来对这个长期以来存在于我们脑海中的固有观念进行一下性能量化。我们运行一个测试，比较一下使用 eval 来调用有字符串参数传递进来的函数和直接调用函数的运行时性能差别。

跟我们前边的测试一样，首先要创建一个有基本 HTML 骨架结构的网页，包含 perfLogger：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Cache Location Comparison</title>
<script src="/lab/perfLogger.js"></script>
</head>
<body>
</body>
</html>
```

然后，在 body 节中创建一个 script 标签以及一个 getAvg 函数。getAvg 函数从 0 ~ 200 循环，将 0 ~ 200 累加起来，然后求平均值。

```
<script>
function getAvg(){
     var avg = 0;
     for(var x = 0; x < 200; x++){
          avg += x;
     }
     return(avg/200);
}
```

创建两个函数，一个用于将 getAvg 函数转换为一个字符串，传递给 eval 函数并用变量保存返回值，另一个函数只是简单地调用 getAvg 函数并保存结果。

⊖ 参见 http://blogs.msdn.com/b/ericlippert/archive/2003/11/01/53329.aspx。

```
function evalAverage(){
     var average = eval(getAvg.toString());
}

function invokeAverage(){
     var average = getAvg()
}
```

最后，调用 perfLogger.logBenchmark，运行每个测试 1000 次：

```
perfLogger.logBenchmark("EvalTime", 1000, evalAverage, true, true);

perfLogger.logBenchmark("InvokeTime", 1000, invokeAverage, true, true);
```

完整的页面代码如下所示：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Cache Location Comparison</title>
<script src="/lab/perfLogger.js"></script>
</head>
<body>
<script>
function getAvg(){
     var avg = 0;
     for(var x = 0; x < 200; x++){
          avg += x;
     }
     return(avg/200);
}

function evalAverage(){
     var average = eval(getAvg.toString());
}

function invokeAverage(){
     var average = getAvg()
}

perfLogger.logBenchmark("EvalTime", 1000, evalAverage, true, true);

perfLogger.logBenchmark("InvokeTime", 1000, invokeAverage, true, true);
</script>
</body>
</html>
```

在浏览器中，你看到的运行结果如下：

```
benchmarking function evalAverage(){ var average = eval(getAvg.toString()); }
average run time: 0.03299599998717895ms
path: http://tom-barker.com/lab/eval_comparison.html

useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11

benchmarking function invokeAverage(){ var average = getAvg() }
```

```
average run time: 0.004606999973475467ms
path: http://tom-barker.com/lab/eval_comparison.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
```

太棒了！我们让一些用户来访问这个页面，或者在我们的实验环境中运行这个页面，然后使用 R 对结果数据进行图表化处理（见图 7-7）：

```
PlotResultsofTestsByBrowser(c("EvalTime", "InvokeTime"), c("Chrome","Firefox"), "Comparison of
average benchmark time \nfor using Eval compared to Function invocation \nin milliseconds")
```

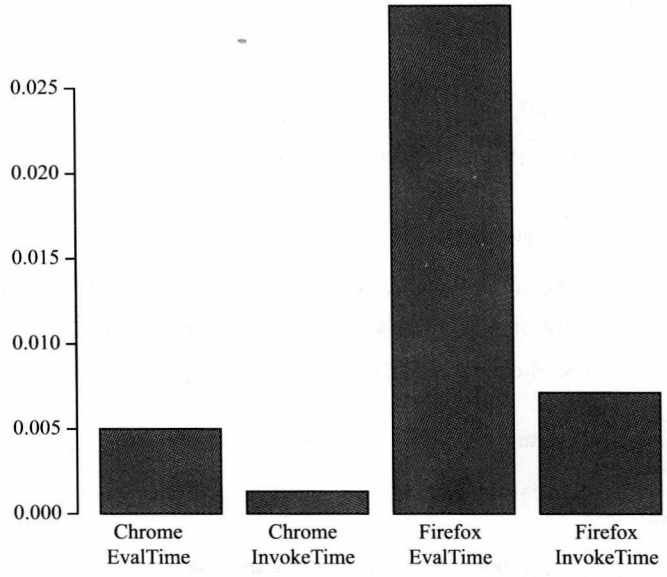


图 7-7　不同浏览器上使用 eval 和直接调用函数的基准测试结果比较

7.4　DOM 访问

浏览器开发商通常都会努力让 DOM 访问更加有效率[⊖]，但是实际上，DOM 的访问不仅包含了浏览器外部 JavaScript 解释器的接口访问，还有渲染引擎的接口访问，这就从本质上导致了 JavaScritp 的开发是效率最为缓慢的一部分。

但是，还是有一些方法能优化与 DOM 的交互。我们来看看吧。

7.4.1　使用队列完成 DOM 元素修改

当我们要对某一个 DOM 元素的内容进行多重修改的时候，我们可以将修改保存在队列中，然后一次性将它们附加给该元素。我们看一个这样的例子。

⊖ 参见 http://updates.html5rocks.com/2012/04/Big-boost-to-DOM-performance---WebKit-s-innerHTML-is-240-faster。

跟以前的测试一样，先创建一个 HTML 结构框架，包含我们的 perfLogger 库和 populateArray 函数。在页面的 body 节中，添加一个 div 标签，并将它的 ID 赋值为 DOMtest，再添加一个 script 标签，在 script 标签中，创建 tempArray 数组。

```
<div id="DOMtest"><p>Dom Test</p></div>
<script>
tempArray = populateArray(400);
</script>
```

除了以上的配置之外，还要创建一个函数 sequentialWrites()。该函数对 tempArray 数组进行循环访问，每一次迭代都要更新 DOMtest 中的 innerHTML。然后对这个函数进行基准测试：

```
function sequentialWrites(){
    var l = tempArray.length
    for(var i = 0; i < l; i++) {
        document.getElementById("DOMtest").innerHTML += i;
    }
}
perfLogger.logBenchmark("SequentialWrites", 10, sequentialWrites, true, true);
```

接下来，创建一个函数 queueWrites，该函数也需要对 tempArray 进行迭代循环访问，每一个迭代中，它将对 DOM 元素的修改连接到一个字符串变量中。循环结束时，该函数将该更新后的字符串变量插入到 div 中。最后，对 queueWrites 函数进行基准测试：

```
function queueWrites(){
    var l = tempArray.length,
        writeVal = "";
    for(var i = 0; i < l; i++) {
        writeVal += i
    }
    document.getElementById("DOMtest").innerHTML += writeVal;
}
perfLogger.logBenchmark("QueueWrites", 10, queueWrites, true, true);
```

完整的脚本测试代码如下所示：

```
<div id="DOMtest"><p>Dom Test</p></div>
<script>
tempArray = populateArray(400);
//benchmark timing data for JS DOM access
function sequentialWrites(){
    var l = tempArray.length
    for(var i = 0; i < l; i++) {
        document.getElementById("DOMtest").innerHTML += i;
    }
}
perfLogger.logBenchmark("SequentialWrites", 10, sequentialWrites, true, true);

//benchmark timing data for JS DOM access
function queueWrites(){
    var l = tempArray.length,
        writeVal = "";
    for(var i = 0; i < l; i++) {
```

```
            writeVal += i
        }
        document.getElementById("DOMtest").innerHTML += writeVal;
    }
    perfLogger.logBenchmark("QueueWrites", 10, queueWrites, true, true);

    </script>
```

浏览器访问的结果如下所示：

```
benchmarking function sequentialWrites(){ var l = tempArray.length for(var i = 0; i < l; i++) {
document.getElementById("DOMtest").innerHTML += i; } }
average run time: 146.904400002677ms
path: http://tom-barker.com/lab/dom_interactions.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11

benchmarking function queueWrites(){ var l = tempArray.length, writeVal = ""; for(var i = 0; i <
l; i++) { writeVal += i } document.getElementById("DOMtest").innerHTML += writeVal; }
average run time: 2.82860000152140 86ms
path: http://tom-barker.com/lab/dom_interactions.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
```

在上例中，用顺序写入和用队列写入显示出了明显的不同——146 毫秒对 2.8 毫秒。但这还只是针对一个单独的测试。我们使用 R 来对大规模的测试结果进行图表化处理。

如果将下面的代码写入到 R 控制台中，你就会得到如图 7-8 所示的图表。两个测试在大规模应用环境下，测试结果有非常明显的不同，以至于我们只看到了大的数值，细微的小数值根本就看不到了。

```
PlotResultsofTestsByBrowser(c("SequentialWrites", "QueueWrites"), c("Chrome","Firefox"),
"Comparison of average benchmark time \n Sequential DOM element updates versus Queued DOM
element updates \nin milliseconds")
```

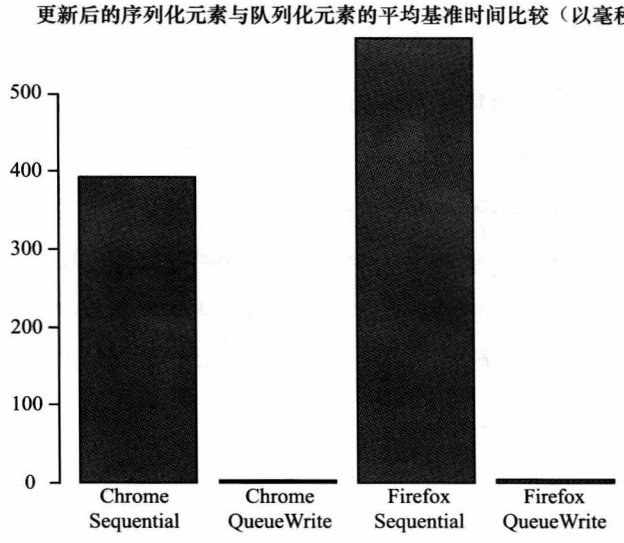


图 7-8 单独写入 DOM 与使用一个队列写入 DOM 的结果比较

7.4.2 使用队列添加新节点

让我们使用同样的方式来向DOM中添加新节点。当我们向页面中添加多个DOM元素时，我们可以用队列的方式来保存这些创建的元素，然后将它们添加到一个DocumentFragment中，最后将DocumentFragment附加到页面中。根据W3C上的描述（http://www.w3.org/TR/DOMLevel-2-Core/core.html#ID-B63ED1A3），DocumentFragment是一个轻量级的Document对象。它为我们提供了一个修改Document的中转站平台，一旦所有的修改被添加到了DocumentFragment这个中转站中，我们只需要将DocumentFragment中的节点复制到Document中就可以了。让我们用一个测试来深入探讨一下这个概念。

我们在同样的页面上使用队列写入DOM和顺序写入DOM的测试，只需要添加一些新的测试来完成基准评估。

首先，创建一个函数useAppendChild。该函数对tempArray数组进行迭代循环，每一个迭代都要创建一个新的脚本元素，并添加到document的head节中。然后对该函数进行基准测试：

```
function useAppendChild(){
     var l = tempArray.length,
          writeVal = "";
     for(var i = 0; i < l; i++) {
          window.document.getElementsByTagName('HEAD')[0].appendChild(window.document.
createElement('SCRIPT'));
     }
}
perfLogger.logBenchmark("AppendChildWrites", 10, useAppendChild, true, true);
```

最后，创建一个函数useDocFragments来做同样的事情，不同之处是将元素添加到DocumentFragment中，而不是Document中。一旦循环结束了，将DocumentFragment中的修改合并到Document中：

```
<script>

//documentfragment vs append child for multiple updates
function useDocFragments(){
     var l = tempArray.length,
          writeVal = "",
          frag = document.createDocumentFragment();
     for(var i = 0; i < l; i++) {
               frag.appendChild(window.document.createElement('SCRIPT'));
          }
     window.document.getElementsByTagName('HEAD')[0].appendChild(frag.cloneNode(true));
}
perfLogger.logBenchmark("DocFragmentWrites", 10, useDocFragments, true, true);
```

完整的测试代码如下所示：

```
<script>
function useAppendChild(){
     var l = tempArray.length,
          writeVal = "";
     for(var i = 0; i < l; i++) {
```

```
        window.document.getElementsByTagName('HEAD')[0].appendChild(window.document.
createElement('SCRIPT'));
    }
}
perfLogger.logBenchmark("AppendChildWrites", 10, useAppendChild, true, true);

</script>
```

使用浏览器查看这几个测试时，看到的结果如下所示：

```
benchmarking function useDocFragments(){ var l = tempArray.length, writeVal = "", frag =
document.createDocumentFragment(); for(var i = 0; i < l; i++) { frag.appendChild(window.
document.createElement('SCRIPT')); } window.document.getElementsByTagName('HEAD')[0].
appendChild(frag.cloneNode(true));      }
average run time: 2.3462999932235107ms
path: http://localhost:8888/lab/chapter7/dom_interactions.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11

benchmarking function useAppendChild(){ var l = tempArray.length, writeVal = ""; for(var i = 0;
i < l; i++) { window.document.getElementsByTagName('HEAD')[0].appendChild(window.document.
createElement('SCRIPT')); }      }
average run time: 2.593400003388524ms
path: http://localhost:8888/lab/chapter7/dom_interactions.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
```

要想对上述测试进行大规模的运行，并对结果进行图表化处理，使用下面描述的 PlotResultsofTestsByBrowser 函数，创建如图 7-9 所示的图表。你会看到使用了 DocumentFragment 之后，Chorme 上的性能改进达到 9.6%，而 Firefox 上则达到了 10%。

```
PlotResultsofTestsByBrowser(c("AppendChildWrites","DocFragmentWrites"), c("Chrome","Firefox"),
"Comparison of average benchmark time \n Individual append child calls vs document fragment \nin
milliseconds")
```

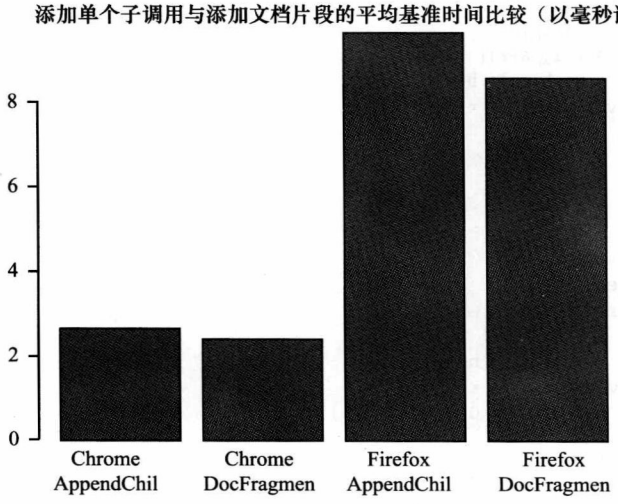


图 7-9 不同浏览器中向 DOM 中添加多个元素和借助 DocumentFrament 实现与 DOM 单一交互的基准测试结果比较

7.5 嵌套循环的代价

我们知道，通常在性能方面，循环所消耗的代价大于将循环展开按顺序执行所消耗的代价。这是因为，每个循环都是同步的，在它完成之前，其会阻塞应用程序其他部分的执行。正如你在第 2 章看到的，Google 关闭编译器优化开关的例子，第 8 章会对这个概念进行深入探讨。然而，代价更昂贵的循环却是嵌套循环。

有很多办法可以避免嵌套循环，如循环展开，就像关闭编译器优化所做的那样，或者循环融合（loop fusion），就是将子循环合并成一个大的单一循环。但是，为什么我们要这么做呢，因为我们要探究一下嵌入循环的消耗。

创建一个新页面，先写入基本的 HTML 框架结构，包含 perfLogger 库和 populateArray 函数。在页面的 body 节中，创建一个 script 标签和一个 tempArray 变量。

在 script 标签中，你要创建几个函数，逐次增加嵌套循环的深度。twoLoopsDeep 函数包含一层嵌套循环，threeLoopsDeep 函数包含一个两层嵌套循环，等等，直到 fiveLoopsDeep。

最后，对每个函数进行基准测试。为了能够在浏览器中测试，需要保留一些运行这些函数的时间，否则，你会得到脚本运行缓慢的警告消息，甚至可能会看到浏览器崩溃。

```
<script>
tempArray = populateArray(20);

function twoLoopsDeep(){
     var l = tempArray.length;
     for(var a = 0; a < l; a++){
          for(var b = 0; b < l; b++){

          }
     }
}

function threeLoopsDeep(){
     var l = tempArray.length;
     for(var a = 0; a < l; a++){
          for(var b = 0; b < l; b++){
               for(var c = 0; c < l; c++){

               }
          }
     }
}

function fourLoopsDeep(){
     var l = tempArray.length;
     for(var a = 0; a < l; a++){
          for(var b = 0; b < l; b++){
               for(var c = 0; c < l; c++){
                    for(var d = 0; d < l; d++){

                    }
               }
          }
     }
}
```

```
function fiveLoopsDeep(){
    var l = tempArray.length;
    for(var a = 0; a < l; a++){
        for(var b = 0; b < l; b++){
            for(var c = 0; c < l; c++){
                for(var d = 0; d < l; d++){
                    for(var e = 0; e < l; e++){

                    }
                }
            }
        }
    }
}

perfLogger.logBenchmark("TwoLoops", 10, twoLoopsDeep, true, true);
perfLogger.logBenchmark("ThreeLoops", 10, threeLoopsDeep, true, true);
perfLogger.logBenchmark("FourLoops", 10, fourLoopsDeep, true, true);
perfLogger.logBenchmark("FiveLoops", 10, fiveLoopsDeep, true, true);
</script>
```

使用浏览器查看页面时，你会看到如下所示的结果：

```
benchmarking function oneLoop(){ var l = tempArray.length; for(var a = 0; a < l; a++){ } }
average run time: 0.008299996261484921ms
path: http://tom-barker.com/lab/cyclomaticcomplexity.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11

benchmarking function twoLoopsDeep(){ var l = tempArray.length; for(var a = 0; a < l; a++){
for(var b = 0; b < l; b++){ } } }
average run time: 0.012399998377077281ms
path: http://tom-barker.com/lab/cyclomaticcomplexity.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11

benchmarking function threeLoopsDeep(){ var l = tempArray.length; for(var a = 0; a < l; a++){
for(var b = 0; b < l; b++){ for(var c = 0; c < l; c++){ } } } }
average run time: 0.062900000164881349ms
path: http://tom-barker.com/lab/cyclomaticcomplexity.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11

benchmarking function fourLoopsDeep(){ var l = tempArray.length; for(var a = 0; a < l; a++){
for(var b = 0; b < l; b++){ for(var c = 0; c < l; c++){ for(var d = 0; d < l; d++){ } } } } }
average run time: 1.022299993201159ms
path: http://tom-barker.com/lab/cyclomaticcomplexity.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11

benchmarking function fiveLoopsDeep(){ var l = tempArray.length; for(var a = 0; a < l; a++){
for(var b = 0; b < l; b++){ for(var c = 0; c < l; c++){ for(var d = 0; d < l; d++){ for(var e =
0; e < l; e++){ } } } } } }
average run time: 6.273999999393709ms
path: http://tom-barker.com/lab/cyclomaticcomplexity.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
```

现在，我们使用 R 来进行图表化处理！如果我们将 2 重循环到 5 重循环都绘制在一个图表上，我们就不会看到渐增的过程了，所以首先我们先绘制 2 重循环和 3 重循环，然后再绘制 4 重循环到 5 重循环。最后，我们在一个浏览器中查看 2 重循环到 5 重循环的整个概况。

首先，我们看看 2 重循环到 3 重循环的延迟增加情况：

```
PlotResultsofTestsByBrowser(c("TwoLoops","ThreeLoops"), c("Chrome","Firefox"), "Comparison of
average benchmark time \n For Increasing Depth of Nested Loops \nin milliseconds")
```

创建的图表如图 7-10 所示。

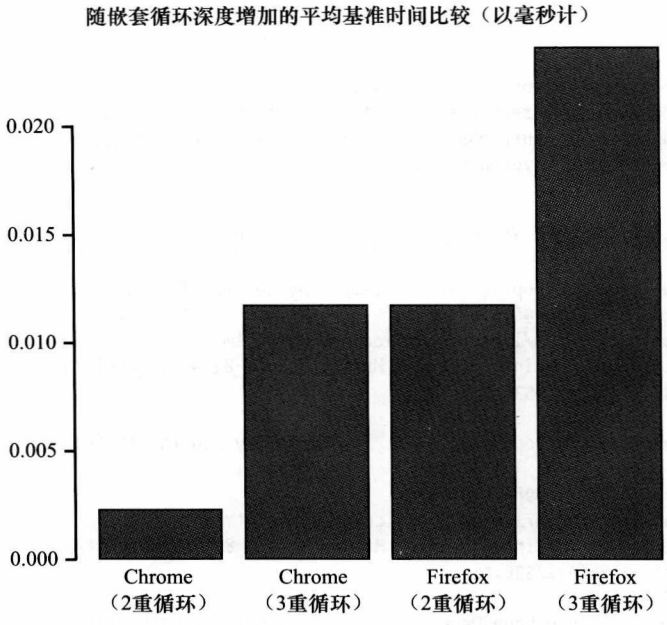


图 7-10 不同浏览器中 2 重嵌套循环到 3 重嵌套循环的性能影响比较

Chrome 上的延迟增加了 420%，而 Firefox 则增加了 103%。

接下来，绘制 4 重嵌套循环到 5 重嵌套循环的延迟增加图表：

```
PlotResultsofTestsByBrowser(c("FourLoops","FiveLoops"), c("Chrome","Firefox"), "Comparison of
average benchmark time \n For Increasing Depth of Nested Loops \nin milliseconds")
```

绘制的图表如图 7-11 所示。

Chrome 上的延迟增加了 1774%，而 Firefox 则增加了 1868%。

最后，我们看一个更大的图形，全面地了解从 2 重循环到 5 重循环的性能影响：

```
PlotResultsofTestsByBrowser(c("TwoLoops", "ThreeLoops", "FourLoops","FiveLoops"), c("Chrome"),
"Comparison of average benchmark time \n For Increasing Depth of Nested Loops \nin
milliseconds")
```

绘制的图表如图 7-12 所示。

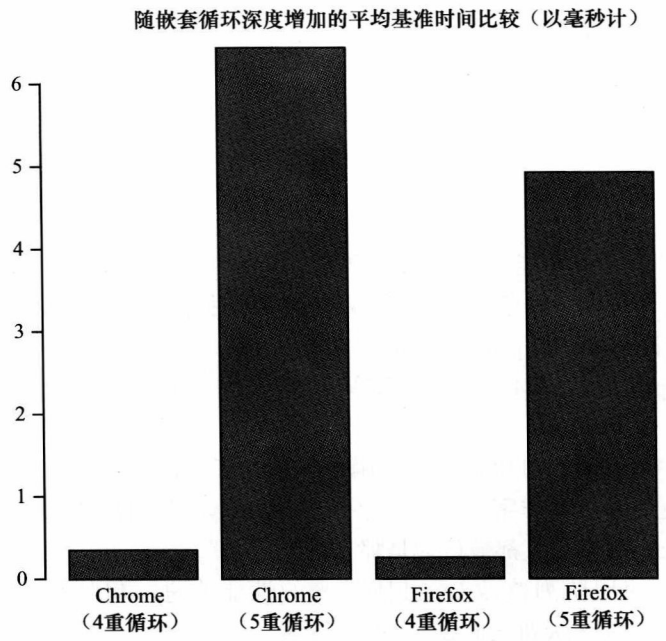


图 7-11 不同浏览器中 4 重嵌套循环到 5 重嵌套循环的性能影响比较

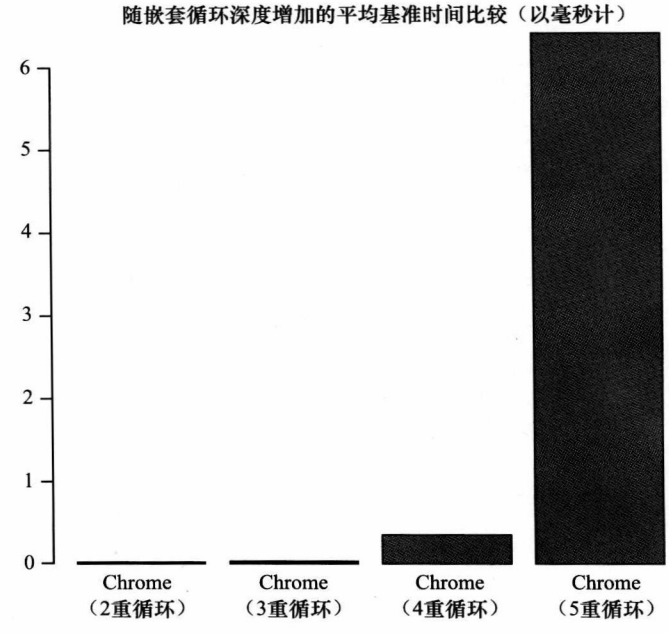


图 7-12 Chrome 上嵌套循环基准测试结果比较

延迟增加了 285 677%。

我们看到了随着嵌套深度和循环次数的增加，循环真实的消耗增加情况。根据我们的研究结果，我们看到，当从2重循环跨越至3重循环时，性能影响是非常剧烈的，随着嵌套深度增加，性能影响也会呈几何级数增加。

7.6 小结

本章，我们对运行时性能进行了深入的研究，探讨了一些概念，如访问其他作用域链中的变量和属性时，将它们保存到本地，以避免跨域栈所造成的性能冲击。

你学到了使用纯JavaScript代替框架来编写代码的好处，并且，我们还对循环和使用jQuery及JavaScript访问DOM的基准测试结果进行了比较，证明了使用纯JavaScript能够带来性能增益。

我们还举了一个例子，对比直接调用函数，量化了使用eval函数调用一个函数所用成本。讲述了我们使用函数而非字符串作为参数传递给eval执行所得到的性能改进。你也看到了eval并不是长期以来一直都被认为是那么的“邪恶”。

你还看到了，使用队列修改DOM所带来的性能改进，包括修改DOM元素到使用DocumentFragment对象来队列化地添加DOM元素。

最后，我们对随着嵌套深度的增加，循环延迟的增加情况进行了量化。

第8章将讨论如何在性能与代码可读性以及代码模块化之间谋求平衡。

第 8 章

在性能、软件工程最佳实践和软件产品运行之间谋求平衡

第 7 章探讨了提升运行时性能的途径。而且，我们使用了 perfLogger 库，并用 R 生成图表量化了性能的提升。测量并使用数据进行证明，这始终是贯穿本书的一个主题。如果必须要用一句话来描述本书主旨，我会用第 1 章曾经说过的那句值得不断重复的话：随便一个熟练工人都可以按照技术规格创造出一些东西，但只有大师级的人物才能创造出杰出的作品，同时还能用实验数据证明其作品的杰出之处。

本书一直都在致力于这么做，创建我们自己的工具来装备我们的代码，监控我们 Web 站点的性能。为了我们的准备数据更易用，我们要精心制作数据可视化方案。

但是，本章稍有不同。我们还会看到原始的数据以及性能优化，但是重点将放在优化和满足需求（比如秉承代码标准化以及坚持最佳实践，保持代码可读性，为了满足较大的开发团队交流需要而进行的代码模块化开发等）之间的平衡。

我们依然会密切关注如何生成一定规模的测试数据，或者利用我们自己的虚拟机测试环境，或者将我们的代码嵌入到成品网站上利用众包访问生成数据。

8.1 在性能与可读性、模块化和良好设计之间谋求平衡

在编写本书的时候，我领导的团队大概有 20 ~ 25 个人，包括工程师、管理者、工程管理者。因此，仅仅 2 ~ 3 行的代码库的改变可能会涉及很多人。我跟踪我们的性能就如同一只鹰跟踪一个田鼠一样。我使用 WebPagetest 为数以万计的 URL 的 Web 性能绘制图表。我会定期和团队进行交流，讨论这些报告的输出，研究我们的首次视图数据以及重复视图数据，确保我们有效地使用缓存。我们会观察性能的各个方面，尽可能地尝试各种优化方法。

我还跟踪其他一些事情，比如：我们的缺陷密度（defect desity）是多少，我们每一个产品的故障率是多少。根据问题的严重性，这些问题对产品品牌造成的危害肯定远远大于性能所造成的影响。

导致类似于缺陷和产品事故这样的事情，依据我的经验，一个根本的原因就是“沟通”。工程师和 QA 人员讨论过关于他们更新功能的事情么？工程师和产品操作人员讨论过如何支持新特性的事情么？工程师和其他人员进行过讨论没有？但沟通的问题远不止这些。字面上几百万行的代码，每个人都能知道所有代码的用处么？库是编写成模块化功能的么？每个人都了解这些库么？如果我看到一段代码，我能够知道怎么使用它么？代码怎么才能变得可读呢？

在开发过程中，当代码被分割的时候，对我来说更重要的是我所有的 20 个工程师要知道如何使用这些代码的可用功能，而不是冥思苦想出一个可以提升性能 2 毫秒的方法来。

那么我们为什么要致力于实现代码的模块化、可复用性以及可读性呢？

为什么要努力做到模块化代码设计呢？也就是说我们总是尝试着用小的自包含的以及可互交换的模块方式编写代码。用模块化的方式编写代码可以将由于代码变更而引起的潜在危害降至最小——因为模块之间的通信是通过接口来实现的，所以我们可以很容易地对接口进行单元测试，并围绕接口交互创建集成测试。

通过提升复用性可以减少出现新 bug 的可能性。理想情况下，我们复用的代码都是经过测试和证明过的。

代码可读性即让代码所实现的功能明确化。这包括：

- 将复杂的逻辑抽象成清晰的、有实际意义的命名函数或者自包含的对象或模块（都是封闭的）；
- 使用我们已经达成一致的代码格式作为标准；
- 使用清晰易懂的命名模式来命名变量。

所有上述这些内容都属于最佳实践，通常也是人力所能达到的最精简的、最高效的编码方式。长远来看，有意义的变量名占用的字符串增加了文件的大小，从而导致增加了页面的负载。同样的道理，编写函数和构造函数所添加的额外代码行也会增加页面的负载，更别说由于在堆上创建这些对象、管理垃圾回收、遍历作用域链所造成的解释器的额外负荷了。

将代码放入到对象和带有意义的名字（按逻辑抽象得来）的函数中，将代码片段原子化，这样做可以保障在更新和维护代码时，不会对系统的其他部分造成太多的影响。

这就是我们关于寻求平衡的所有观点，也是本章将要讨论的部分内容。

8.2 焦土化性能

前边的章节，我提到过“焦土化性能”的概念。这是一个我自己杜撰的术语，它的意思是为了追求最佳性能而牺牲所有其他的因素。这一节，我们来看看焦土化实践，我们来量化一下这样做的好处，然后从全局的角度讨论一下这么做的价值。

8.2.1 内联函数

我们首先来看看，使用内联函数会给运行时性能带来什么好处。前边的章节中，我们已经看到了遍历内存结构的成本开销。第 7 章我们讨论了在不同内存作用域环境下遍历内存

结构的开销，同样的概念也适用于我们创建的对象体系。

从表面上看，将所有的功能合并到一个函数中，而不是将功能抽象到分割函数（separate function）甚至是对象中，运行时性能一定会得到提升。我们来看看这一点。

下边的例子要创建一个单独的页面，在该页面把功能集成在一个函数中，将代码分割到不同的函数中，并创建包含功能的对象进行基准测试。我们开始吧！

1. 创建一个例子

首先创建一个包含了基本 HTML 骨架的新页面，包括 perfLogger.js 库：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Methodology Comparison</title>
<script src="/lab/perfLogger.js"></script>
</head>
<body>
</body>
</html>
```

然后，在页面的 body 节中创建一个 script 标签，并创建一个函数，用于连接我们想要做的每一件事情。函数名为 unwoundfunction()：

```
<script>
function unwoundfunction(){

}
</script>
```

在 unwoundfunction 内，需要创建一个变量 sum，通过一个 for 语句进行 300 次迭代循环，计算出每一次迭代增量之和。

```
var sum = 0;
for(var x = 0; x < 300; x++){
    sum += x;
}
```

然后，创建一个变量 average，用于保存 300 次迭代增量之和的平均值。这里做了两次计算操作——求和以及计算平均值。

```
var average = 0;
for(var x = 0; x < 300; x++){
    average += x;
}
average = average/300;
```

完成后的函数如下所示。该函数用于计算集成功能的基准测试时间：

```
function unwoundfunction(){
    var sum = 0;
    for(var x = 0; x < 300; x++){
        sum += x;
    }
```

```
    var avgerage = 0;
    for(var x = 0; x < 300; x++){
        avgerage += x;
    }
    avgerage = avgerage/300;
}
```

接下来，创建两个新函数，一个用于处理求和结果，另一个用于处理结果的平均值：

```
function getAvg(p){
    var avg = 0;
    for(var x = 0; x < p; x++){
        avg += x;
    }
    return(avg/p);
}

function getSum(a){
    var sum = 0;
    for(var x = 0; x < a; x++){
        sum += x;
    }
    return(sum);

}
```

然后，创建第三个函数，用于调用 getSum() 函数和 getAvg() 函数。这个函数将作为使用函数做基准测试的一个例子：

```
function usingfunctions(){
    var average = getAvg(300);
    var sum = getSum(300)
}
```

现在，创建一个对象构造函数来处理这一功能。该对象命名为 simpleMath，对象中有两个公有方法：sum() 和 avg()。

```
function simpleMath(){
    this.sum = function(a){
        var sum = 0;
        for(var x = 0; x < a; x++){
            sum += x;
        }
        return(sum);
    }
    this.avg = function(p){
        var avg = 0;
        for(var x = 0; x < p; x++){
            avg += x;
        }
        return(avg/p);
    }
}
```

然后，创建一个函数 usingobjects，用于初始化一个新的 simpleMath 对象，并调用 sum 和 avg 方法。你将会对该函数进行基准测试，以获取使用对象的度量值。

```
function usingobjects(){
     var m = new simpleMath();
     var average = m.avg(300);
     var sum = m.sum(300);
}
```

最后，对这些函数进行基准测试，使用 perfLogger 执行每个函数 100 遍：

```
perfLogger.logBenchmark("UsingObjects", 100, usingobjects, true, true);
perfLogger.logBenchmark("UsingFunctions", 100, usingfunctions, true, true);
perfLogger.logBenchmark("unwoundfunction", 100, unwoundfunction, true, true);
```

完成之后的测试页面如下所示：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Methodology Comparison</title>
<script src="/lab/perfLogger.js"></script>
<script>
function getAvg(p){
     var avg = 0;
     for(var x = 0; x < p; x++){
          avg += x;
     }
     return(avg/p);
}

function getSum(a){
     var sum = 0;
     for(var x = 0; x < a; x++){
          sum += x;
     }
     return(sum);

}


function simpleMath(){
     this.sum = function(a){
          var sum = 0;
          for(var x = 0; x < a; x++){
               sum += x;
          }
          return(sum);
     }

     this.avg = function(p){
          var avg = 0;
          for(var x = 0; x < p; x++){
               avg += x;
          }
          return(avg/p);
     }
}
</script>
</head>
```

```
<body>
<script>

function usingfunctions(){
     var average = getAvg(300);
     var sum = getSum(300)
}

function usingobjects(){
     var m = new simpleMath();
     var average = m.avg(300);
     var sum = m.sum(300);
}

function unwoundfunction(){
     var sum = 0;
     for(var x = 0; x < 300; x++){
          sum += x;
     }

     var average = 0;
     for(var x = 0; x < 300; x++){
          average += x;
     }
     average = average/300;
}


perfLogger.logBenchmark("UsingObjects", 100, usingobjects, true, true);
perfLogger.logBenchmark("UsingFunctions", 100, usingfunctions, true, true);
perfLogger.logBenchmark("unwoundfunction", 100, unwoundfunction, true, true);

</script>
</body>
</html>
```

在浏览器中查看该页，看到的结果如下所示：

```
benchmarking function usingobjects() { var m = new simpleMath; var average = m.avg(300); var sum
= m.sum(300); }
average run time: 0.025260580000000914ms
path: http://tom-barker.com/lab/useFunctions.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.5; rv:16.0) Gecko/16.0 Firefox/16.0

benchmarking function usingfunctions() { var average = getAvg(300); var sum = getSum(300); }
average run time: 0.020855050000000687ms
path: http://tom-barker.com/lab/useFunctions.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.5; rv:16.0) Gecko/16.0 Firefox/16.0

benchmarking function unwoundfunction() { var sum = 0; for (var x = 0; x < 300; x++) { sum += x;
} var avgerage = 0; for (var x = 0; x < 300; x++) { avgerage += x; } avgerage = avgerage / 300;
}
average run time: 0.016489299999999666ms
path: http://tom-barker.com/lab/useFunctions.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.5; rv:16.0) Gecko/16.0 Firefox/16.0
```

重复上述测试，将测试代码放置到软件产品或测试环境中，获取代码消耗的流量，并

最终在日志文件中添加一个全面的测试结果。

我们获取这些结果，使用 R 来对这些结果图表化。

2. 结果分析

为了绘图需要，你可以复用第 7 章创建的 PlotResultsofTestsByBrowser() 函数，将每个测试的 ID 传递给该函数。创建的图表如图 8-1 所示。

```
PlotResultsofTestsByBrowser(c("unwoundfunction", "UsingFunctions", "UsingObjects"),
c("Firefox"), "Comparison of average benchmark time \nfor coding methodology \nin milliseconds")
```

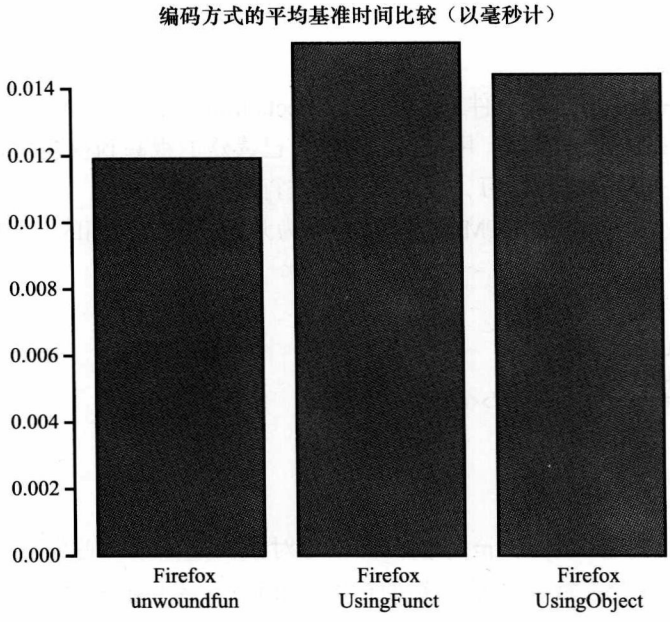


图 8-1 使用函数以及对象实现功能集成后的运行时性能比较

你可以看到，由于功能集成而删除掉了所有的负荷之后，以命令行式的语句逐行编写代码带来了性能改进。在这个简单的例子中，差异都在毫秒之内，但是百分比的区别还是很明显的。使用函数完成功能集成，性能最大改进了 23%，而使用对象完成功能集成，性能改进了 18%。在性能就是一切的情况下，比如金融交易时，这是一个非常明显的区别。

但是将功能分割到不同的函数中会使我们的代码具有更好的可读性。很明显的，代码如这样编写：average=getAvg 或者 sum=getSum，可读性会非常好。

创建对象所带来的改进更加明显。你可以在工程之间复用对象，在不同应用程序之间传递对象，还可以扩展一个对象来生成新的对象，或者通过改进原型链来复用功能。

大多数情况下，对于复用性和可读性的改进来讲，这些增加的额外开销是非常值得的。

8.2.2 Closure Compiler

另一个焦土化性能的方法是 Google 使用高级模式作用于 JavaScript 的 Closure Compiler。

第 2 章我们涉及了一点关于 Closure Compiler 的内容，但现在让我们看一个鲜活的例子。

Closure Compiler 可以在以下两种模式中运行：

- 在简单模式（Simple mode）下，Closure Compiler 就如同其他大多数的代码简化器一样，删除空格，回车符以及注释；
- 在高级模式（Advanced mode）下，Closure Compiler 会对变量名和函数进行重命名，将长的描述性的名称改写为简单的几个字符的名称，以便于节省文件大小，同时将内联函数集成到一个它认为可以放置的单一的函数中。

我们在高级模式中考虑焦土化。我们来看一个例子。

1. 创建一个例子

首先，创建一个基准页面文件 benchmarkobjects.html。在这个页面中要创建两个对象，一个 user 对象，一个 video 对象。用户可以添加自己喜欢的视频到视频列表中。后边我们会使用一个函数对这一功能进行练习，并对该函数进行基准测试。

还是以我们熟悉的基本的 HTML 框架结构作为开始，包含我们的 perfLogger 库：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Loop Comparison</title>
<script src="/lab/perfLogger.js"></script>
</head>
<body>
</body>
</html>
```

在 body 节中，创建一个 script 标签，再编写对象构造函数。首先创建 video 对象构造函数，该函数接收一个参数，表示视频的标题。video 对象中有一个公有方法 printInfo()，用于返回视频的标题。

```
<script>
function video(title){
          this.title = title;
          this.printInfo = function(){
               return this.title;
          }

     }
</script>
```

接下来，创建 user 对象的构造函数。user 对象接收一个参数，用于设置用户的名称，它有两个公有方法：addToFavorite() 方法，用于将传递进来的对象参数添加到 user 对象的 favoriteList 列表中；showFavorites() 方法，该方法遍历 favoriteList 列表。user 对象在控制台上记录 printInfo 方法作用于 favoriteList 列表中每一个视频返回值：

```
function user(uname){
     this.username = uname;
     this.favoriteList = [];
```

```
    this.addToFavorite = function(a){
        this.favoriteList[this.favoriteList.length] = a;
    }

    this.showFavorites = function(){
        for(var f = 0; f < this.favoriteList.length; f++){
            var t = this.favoriteList[f].printInfo();
            console.log(t);
        }
    }
}
```

最后，创建一个函数对上述创建的功能进行实践，并且对该函数进行基准测试。该函数将创建一个新的 user 对象，然后迭代 20 次，每一次添加一个新的 video 对象到 user 对象的 favoriteList 列表中：

```
function testUserObject(){
    var u1 = new user("tom");
    for(var i = 0; i < 20; i++){
        u1.addToFavorite(new video("video "+ i));
    }
    u1.showFavorites();
}
perfLogger.logBenchmark("benchmarkObject", 10, testUserObject, true, true);
```

完成后的页面如下所示：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Loop Comparison</title>
<script src="/lab/perfLogger.js"></script>
</head>
<body>
    <script>
    function user(uname){
        this.username = uname;
        this.favoriteList = [];
        this.addToFavorite = function(a){
            this.favoriteList[this.favoriteList.length] = a;
        }

        this.showFavorites = function(){
            for(var f = 0; f < this.favoriteList.length; f++){
                var t = this.favoriteList[f].printInfo();
                console.log(t);
            }
        }
    }

    function video(title){
        this.title = title;
        this.printInfo = function(){
            return this.title;
        }

    }
```

```
    function testUserObject(){
        var u1 = new user("tom");
        for(var i = 0; i < 20; i++){
            u1.addToFavorite(new video("video "+ i));
        }
        u1.showFavorites();
    }

    perfLogger.logBenchmark("benchmarkObject", 10, testUserObject, true, true);
    </script>
</body>
</html>
```

当在浏览器上查看该页面时，看到的结果如下所示：

```
benchmarking function testUserObject(){ var u1 = new user("tom"); for(var i = 0; i < 20; i++){
u1.addToFavorite(new video("video "+ i)); } u1.showFavorites();      }
average run time: 0.6119000026956201ms
path: http://tom-barker.com/lab/benchmarkobjects.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
```

2. 通过 Closure Compiler 运行示例

现在，准备通过 Closure Compiler 来运行代码。最简单的方法就是使用 Closure Compiler 用户界面（UI），通过如下地址可以访问该界面：http://closure-compiler.appspot.com/home。

Closure Compiler 用户界面是一个 Web 应用程序（见图 8-2）。在图 8-2 的左侧，输入 JavaScript 代码并选择一些选项，在右侧就是 Closure Compiler 的输出。

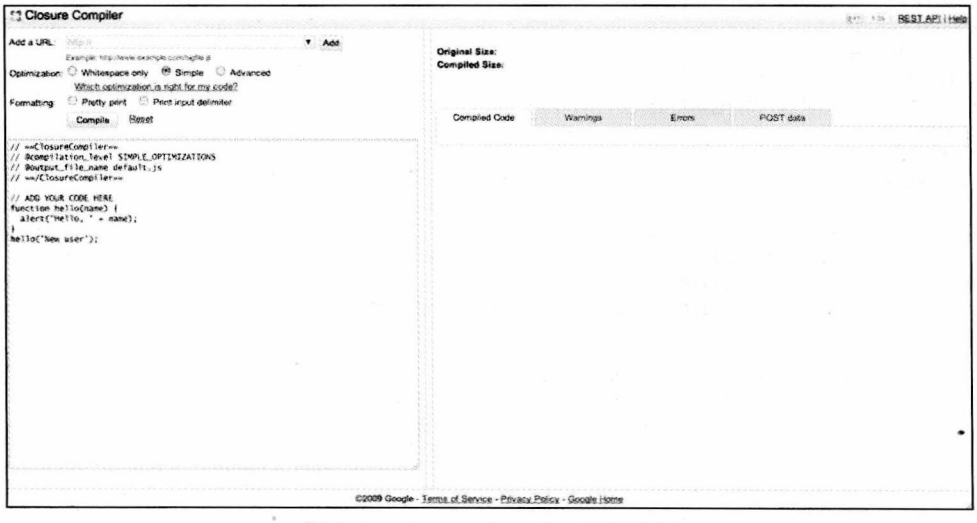


图 8-2 Closure Compiler 用户界面

左上角的选项如下：

- 一个文本框，用于输入包含在编译器内的远程 JavaScript 文件的 URL 地址。只需要将地址输入到文本框中，然后点击 Add 按钮。你会看到输入的 URL 出现在底部的大文本区域中，例如：

```
// @code_url http://tom-barker.com/lib/perfLogger.js
```

- 一系列单选按钮，用于表示 Closure Compiler 运行的模式，是 Whitespace Only 模式，还是 Simple 模式，或者是 Advanced 模式。Whitespace Only 模式就跟字面意思一样：删除注释、回车符和不需要的空格。Simple 编译模式除了删除空格、回车符以及注释之外，还对本地变量使用短名称进行重命名。正如你已经看到的，Advanced 编译模式将全部重写 JavaScript 代码。
- 文本格式化选项。Pretty Print 选项包含换行符和易于阅读的缩进符，Print Input Delimiter 则允许传递一个字符串作为传入的代码块的边界，如果你传入了多个远程文件，该输入的分隔符将会出现在来自不同文件的代码之间，用于区分不同文件的代码块。
- 一个 Compile 按钮，最后一个大的文本区域，在这里你可以输入你的选项和其他你想要编译的代码。

右侧是一个大的文本区域，这里是代码编译后输出的地方。还有一些标签（tab），在这些标签中可以查看编译过程中的警告或者错误信息。

如果在这个测试中包含了 perfLogger.js 文件，并且对其进行了编译，当你使用编译后的结果时，你会得到一个 JavaScript 错误信息，因为 Closure Compiler 已经将 performance.now() 进行了重命名，如图 8-3 所示。

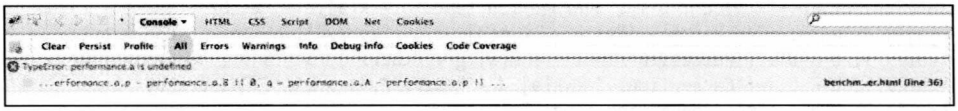


图 8-3 经 Closure Compiler 的高级编译模式运行后的 perfLogger 的 JavaScript 错误信息

因此，要想让测试能顺利完成，你只需要将 perfLogger 的内容复制、粘贴到右侧的文本区域，然后将对 performance.now 的引用改为对 Date.now() 的引用。然后，从 benchmarkobjects.html 文件中，将 script 标签内的内容复制到文本域中，紧挨在 perfLogger 的下面，如图 8-4 所示。

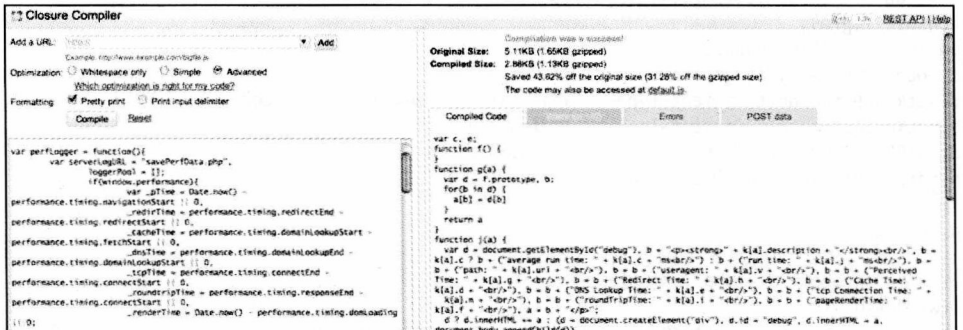


图 8-4 借助 Closure Compiler 用户界面运行 perfLogger

然后，创建一个新的页面，其中只包含 HTML 框架结构，在 body 节中添加一个 script 标签，将编译后的 JavaScript 代码复制，粘贴到该标签中。完成后的测试文件如下所示：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Closure Compiler Benchmark</title>
</head>
<body>
<script>
var c, e;
function f() {
}
function g(a) {
 var d = f.prototype, b;
 for(b in d) {
   a[b] = d[b]
 }
 return a
}
function j(a) {
 var d = document.getElementById("debug"), b = "<p><strong>" + k[a].description + "</
strong><br/>", b = k[a].c ? b + ("average run time: " + k[a].c + "ms<br/>") : b + ("run time: "
+ k[a].j + "ms<br/>"), b = b + ("path: " + k[a].url + "<br/>"), b = b + ("useragent: " + k[a].v
+ "<br/>"), b = b + ("Perceived Time: " + k[a].g + "<br/>"), b = b + ("Redirect Time: " + k[a].h
+ "<br/>"), b = b + ("Cache Time: " + k[a].d + "<br/>"), b = b + ("DNS Lookup Time: " + k[a].e +
"<br/>"), b = b + ("tcp Connection Time: " +
 k[a].m + "<br/>"), b = b + ("roundTripTime: " + k[a].i + "<br/>"), b = b + ("pageRenderTime: "
+ k[a].f + "<br/>"), a = b + "</p>";
 d ? d.innerHTML += a : (d = document.createElement("div"), d.id = "debug", d.innerHTML = a,
document.body.appendChild(d))
}
function l(a) {
 a = "data=" + JSON.stringify(g(k[a]));
 console.log(a);
 var d = new XMLHttpRequest;
 d.open("POST", m, !0);
 d.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
 d.setRequestHeader("Content-length", a.length);
 d.setRequestHeader("Connection", "close");
 d.onreadystatechange = function() {
 };
 d.send(a)
}
var m = "savePerfData.php", k = [];
if(window.D) {
 var n = Date.now() - performance.a.C || 0, o = performance.a.G - performance.a.H || 0, p =
performance.a.p - performance.a.B || 0, q = performance.a.A - performance.a.p || 0, r =
performance.a.w - performance.a.o || 0, t = performance.a.I - performance.a.o || 0, u = Date.
now() - performance.a.z || 0
}
c = f.prototype;
c.g = n;
c.h = o;
c.d = p;
```

```
c.e = q;
c.m = r;
c.i = t;
c.f = u;
e = {k:function(a, d, b, h) {
 k[a] = new f;
 k[a].id = a;
 k[a].startTime = Date.now();
 k[a].description = d;
 k[a].q = b;
 k[a].s = h
}, l:function(a) {
 k[a].u = Date.now();
 k[a].j = k[a].u - k[a].startTime;
 k[a].url = window.location.href;
 k[a].v = navigator.userAgent;
 k[a].q && j(a);
 k[a].s && l(a)
}, r:function(a, d, b, h, v) {
 for(var i = 0, s = 0;s < d;s++) {
   e.k(a, "benchmarking " + b, !1, !1), b(), e.l(a), i += k[a].j
 }
 k[a].c = i / d;
 h && j(a);
 v && l(a)
}, g:function() {
 return n
}, h:function() {
 o
}, d:function() {
 return p
}, e:function() {
 return q
}, m:function() {
 return r
}, i:function() {
 return t
}, f:function() {
 return u
}, J:function() {
 this.k("no_id", "draw perf data to page", !0, !0);
 this.l("no_id")
}};
function w() {
 this.K = "tom";
 this.b = [];
 this.n = function(a) {
   this.b[this.b.length] = a
 };
 this.t = function() {
   for(var a = 0;a < this.b.length;a++) {
     console.log(this.b[a].title)
   }
 }
}
function x(a) {
 this.title = a;
```

```
 this.F = function() {
   return this.title
 }
}
e.r("benchmarkClosureCompiler", 10, function() {
 for(var a = new w, d = 0;20 > d;d++) {
   a.n(new x("video " + d))
 }
 a.t()
}, !0, !0);
</script>
</body>
</html>
```

如果使用浏览器查看该页面，看到的结果如下所示：

```
benchmarking function () { for(var a = new w, d = 0;20 > d;d++) { a.n(new x("video " + d)) }
a.t() }
average run time: 0.9ms
path: http://tom-barker.com/lab/benchmarkclosurecompiler.html
useragent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
```

当查看日志文件时，你会看到 Closure Compiler 重写后的代码并没有将所有的字段都保存到日志文件中。这是因为 Closure Compiler 重命名了大多数的变量，包括 runtime。如果你使用 console.log 查看序列化的数据，你看到的提交的数据如下所示：

```
data={"id":"benchmarkClosureCompiler","startTime":1344114475936,"description":"benchmarking
function () {\n    for (var a = new w, d = 0; 20 > d; d++) {\n        a.n(new x(\"video \" +
d));\n    }\n    a.t();\n}","q":false,"s":false,"u":1344114475943,"j":7,"url":"http://tom-
barker.com/lab/benchmarkclosurecompiler.html","v":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.5;
rv:16.0) Gecko/16.0 Firefox/16.0","c":18.6}
```

runtime 变量在最后被替换为一个小写的“c”。现在，祝你有好的运气，能够从这一堆编译后的乱糟糟的源代码中解析出想要的信息。公平地讲，还是有办法来保留原有的属性名的，就像使用带引号的字符串属性名——比如使用 testResult["runtime"] 代替 obj.runTime。更多这方面的信息，可以查看 Google 的文档，地址是：https://developers.google.com/closure/compiler/docs/api-tutorial3.

当你将数据传回给 savePerfData.php 时，savePerfData.php 希望得到的变量名是 runTime 或者是 avgRunTime，而不是 c，因此永远也不会检索到运行时数据。

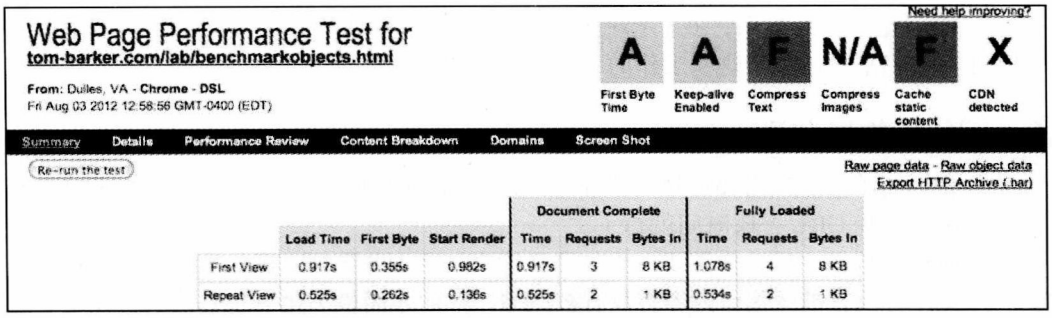
但这没什么关系，我们这里感兴趣的是在 Web 性能方面取得的改进，我们在 WebPagetest 中比较这两个页面。

3. 比较和分析

我们到 webpagetest.com 上边去，测试一下上述的两个 URL。下表列举了测试的 URL 地址，以及测试结果 URL。

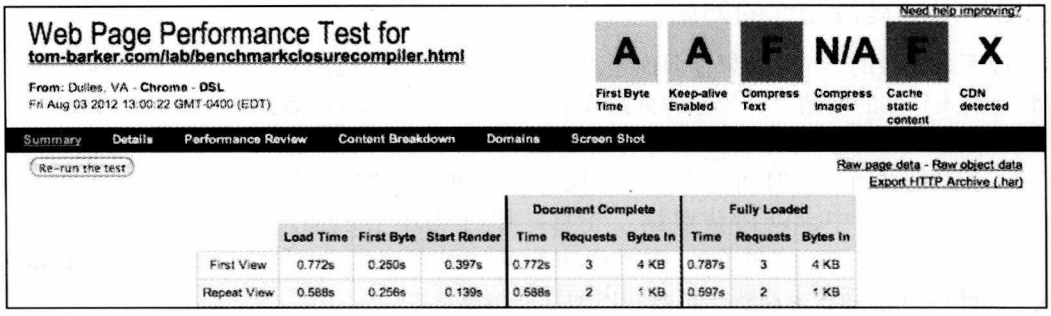
用于测试的 URL	测试结果 URL
tom-barker.com/lab/benchmarkobjects.html	http://www.webpagetest.org/result/120803_WS_ad105844519fccc308dd9f678bc0caae/
tom-barker.com/lab/benchmarkclosurecompiler.html	http://www.webpagetest.org/result/120803_B0_20df854313c5101e6339e24bb0d958ec/

汇总结果如图 8-5 和图 8-6 所示。



	Document Complete						Fully Loaded					
	Load Time	First Byte	Start Render	Time	Requests	Bytes In	Time	Requests	Bytes In
First View	0.917s	0.355s	0.982s	0.917s	3	8 KB	1.078s	4	8 KB
Repeat View	0.525s	0.262s	0.136s	0.525s	2	1 KB	0.534s	2	1 KB

图 8-5 benchmarkobject.html 页面的 Web 性能测试结果汇总



	Document Complete						Fully Loaded					
	Load Time	First Byte	Start Render	Time	Requests	Bytes In	Time	Requests	Bytes In
First View	0.772s	0.250s	0.397s	0.772s	3	4 KB	0.787s	3	4 KB
Repeat View	0.588s	0.256s	0.139s	0.588s	2	1 KB	0.597s	2	1 KB

图 8-6 benchmarkclosurecompiler.html 页面 Web 性能测试结果汇总

从上面的结果视图中我们看到，使用 Closure Compiler 生成的代码的测试中，加载时间快了 200 毫秒，首字节时间快了 100 毫秒，启动渲染时间快了大概 600 毫秒，文档完成时间快了 145 毫秒，全部加载时间快了 291 毫秒。可以明显地看到使用 Closure Compiler 的 Advanced 模式所取得的性能增益。

但是，到这里你还应该看到不足之处。为了使编译后的代码能够在浏览器中正常的运行，不产生错误信息，对原始代码进行改动是避免不了的。当在浏览器上运行的时候，你会看到挂接在后端的一些钩子停止工作了。

所有的这些还只是小的、自包含测试样例。想象一下，如果我们有一个第三方的特定代码需要嵌入到页面中。想象一下如果我们用 JavaScript 与插件进行交互时的场景。

现在，想象一下，为原始代码添加新特性，然后再完成上述的测试步骤。每一周都是如此。20 个人在这段代码的基础上添加新特性，情况会怎样呢？

我们不得不改变我们的工作流，至少要引入一个额外的调试步骤，测试一下编译之后的实际运行情况。我们已经增加了复杂度，用一个新的断点来中断代码的运行。当所有的事情都被模糊化了之后，在这个层次上的调试，比调试我们最初代码的困难程度要高几个数量级。

我们获得的性能增益，是否值得上我们需要为此而增加的额外工作，以及维护、修改编译后代码所增加的复杂性呢？

8.3 下一步：从实践到实际应用

到目前为止，贯穿本书，我们都在创建测试，讨论并观察一定规模上的测试生成的数据。现在，我们来看看关于这些工作的一些策略性的问题。

8.3.1 Web 性能监测

这是相当简单的。只需要选择一些你想要监测的 URL 地址，将它们嵌入到 WPTRunner 中，然后跟踪这些 URL 一段时间就行了。

当你收集到数据之后，你需要和你的团队检查一下这些数据。确认需要改进的地方——你的图片文件有没有被优化，内容有没有被压缩，如何才能最小化 HTTP 请求？设定性能目标，和你的团队一起朝着这个目标开始工作。

8.3.2 用工具检测你的网站

接下来该做的事情（如果你前面还没有做）是用工具来检测你的网站，将基准测试代码实时地纳入到你的网站中，为你的页面收集真实的、实时的性能数据。

要做到这一点，你只需要选择你想要监测的页面，然后选择一些你想要收集的度量值——也许是页面加载时间，或者是更加复杂的功能模块的运行时间，接着将 perfLogger 整合到这些页面中，以捕获数据。注意，你可能还不想使用 perfLogger 的基准测试特性，因为它将会对你的页面性能产生影响，这样的话，可使用 startTimeLogging 和 stopTimeLogging 函数来捕获一定时间内的性能信息。

我用我的网站作为测试对象。在图 8-7 中，你可以看到我的主页的一个截图，perfLogger 调试信息在页面的右侧。

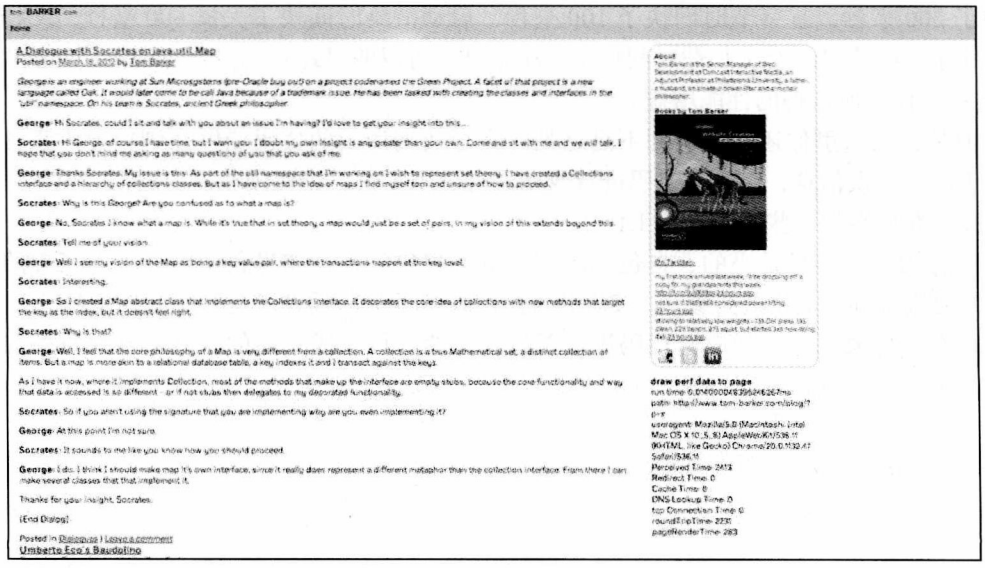


图 8-7 tom-barker.com 站点的性能数据截图

将工具嵌入到我们实际使用的网站中，这样做的好处是可以获取来自于实际用户的真实、实时的数据。我们跟踪这些数据一段时间，然后查找由于条件变化（新的浏览器、浏览器版本变更、提升产品质量的新性能，等等）随之而来的问题。当性能数据出现峰值的时候，这就意味着某些事情突然运行慢很多，或者当性能数据出现意想不到的下降时，可能就意味着用户无法访问你的网站了。

使用工具检测你的网站，也就是定期地维护你的网站。

8.3.3 在测试实验环境中进行基准测试

我们定期地使用工具检测我们的代码并监视我们的网站性能，但是如何才能确保我们的代码在正式发布之前是性能比较高的呢？我们在测试环境中做一下基准测试来看看。

测试环境可以是一个由多个工作站组成的物理环境，也可以是运行在一个或两个机器上的多个虚拟机组成的虚拟环境。

如果你有一定规模的物理设备的实验环境，有预算，有工作人员，那真是太棒了！真实世界中的 Web 应用程序的测试就是这么做的。但是我曾经工作的 12 个公司中，有 7 个是完全不可能有这样的条件的。我和我的开发者不得不在自己的机器上验证和测试代码。在某些情况下，这才是你真正需要的测试环境。

无论是哪种情况，首先，你第一个要做的事情就是建立一个浏览器支持矩阵。如果你没有这么一个明晰的、你能支持的列表，你就不会知道浏览器以及客户端所能够组成的集合是怎样的。一个浏览器支持矩阵至少要包含一个浏览器列表、浏览器版本以及能够为这些浏览器提供支持的情况。我自己的实践中，我一般会包含三个层次的支持——为终端用户使用这些浏览器提供支持（产品支持）；为我的 QA 团队使用浏览器进行定期测试提供支持（QA 测试）；为我的工程师使用这些浏览器执行开发测试提供支持，至少能够确保使用这些浏览器执行特性功能（开发测试）。图 8-8 是这类浏览器矩阵的一个例子。

浏览器	产品支持	QA测试	开发测试
IE 10	N	Y	Y
IE 9	Y	Y	Y
IE 8	Y	Y	N
IE 7	Y	Y	N
Chrome 21b	N	Y	Y
Chrome 20	Y	Y	Y
Chrome 19	Y	Y	N
Firefox Aurora	N	N	Y
Firefox Beta	N	Y	Y
Firefox 14	Y	Y	Y
Firefox 13	Y	Y	N
Safari 5.5	Y	Y	Y
Safari 5.0x	N	Y	N

图 8-8 最低限度的浏览器支持矩阵

理想情况下，最终你的浏览器支持矩阵应当包含插件，以及详细的特性信息等这类内容，因为不是每一个特性在每一个浏览器上都能运行。图 8-9 是一个内容更加丰富的浏览器支持矩阵。

浏览器	产品支持	QA测试	开发测试	左侧导航栏可扩展性	内容回流	Ajax崩溃	负载弹化
iOS 5.0	Y	Y	Y	Y	Y	Y	Y
iOS 4.3	Y	Y	N	Y	Y	Y	Y
Android 4.0	Y	Y	Y	Y	Y	Y	Y
Android 3.1	Y	Y	N	Y	Y	Y	Y
IE 10	N	Y	Y	Y	Y	Y	Y
IE 9	Y	Y	Y	Y	Y	Y	Y
IE 8	Y	Y	N	Y	N	Y	N
IE 7	Y	Y	N	N	N	Y	N
Chrome 21b	N	N	Y	Y	Y	Y	Y
Chrome 20	Y	Y	Y	Y	Y	Y	Y
Chrome 19	Y	Y	N	Y	Y	Y	Y
Firefox Aurora	N	N	Y	Y	Y	Y	Y
Firefox Beta	N	N	Y	Y	Y	Y	Y
Firefox 14	Y	Y	Y	Y	Y	Y	Y
Firefox 13	Y	Y	N	Y	Y	Y	Y
Safari 5.5	Y	Y	Y	Y	Y	Y	Y
Flash 11	Y	Y	Y	N/A	N/A	N/A	N/A
Flash 10	Y	Y	N	N/A	N/A	N/A	N/A
Silverlight 5	Y	Y	Y	N/A	N/A	N/A	N/A
Silverlight 4	Y	Y	N	N/A	N/A	N/A	N/A

图 8-9 更加详细的浏览器支持矩阵

开始选择浏览器矩阵的方法是通过查看你的日志——哪一个浏览器被你的用户使用得最多呢？确保你至少考虑了最为活跃的浏览器，但它并不总是最吸引人的浏览器（由于公司升级策略的原因，你的用户固定地使用 IE 6，因为这一点，你需要支持 IE 6 多少年？）。但是不要假设，由于某些特定的浏览器会用于访问你的网站，你只需要集中精力关注这些浏览器。有可能对于你的网站来讲，仅仅是在这些浏览器上能够工作得最好。同时，要确保在你的矩阵中包含了测试版以及更早版本的浏览器，这样你就可以在编写代码时做到未雨绸缪了。

一旦你有了浏览器支持矩阵，你就可以开始配置工作站或者虚拟机来对这些浏览器进行测试了。即使你有一个从事 QA 的同事可以处理这些测试，作为 Web 开发人员，我们有责任确保在交付给 QA 人员之前，我们所创建的功能是没有问题的，并处于良好状态的。这就需要我们依据浏览器矩阵对我们的代码进行基准测试。

如果准备使用虚拟机做测试，我的最佳选择是使用 Oracle 的 Virtual Box，可以在 https://www.virtualbox.org/ 上找到它。它是完全免费的、开源的、轻量级的、专业的运行虚拟机的解决方案。图 8-10 是 Virtual Box 的主页。

你只需要进入到下载栏目，选择适合于你的本地操作系统的二进制版本就可以了。图 8-11 就是 Virtual Box 的下载页面。

一旦你下载并安装好了 Virtual Box，你就可以按照软件中命令提示，添加新的虚拟机了。注意，你还会需要你想要运行的所有操作系统的安装盘或磁盘映像。当所有的虚拟机配

置完成之后，你的 Virtual Box 安装就应该如图 8-12 所示的这样了。

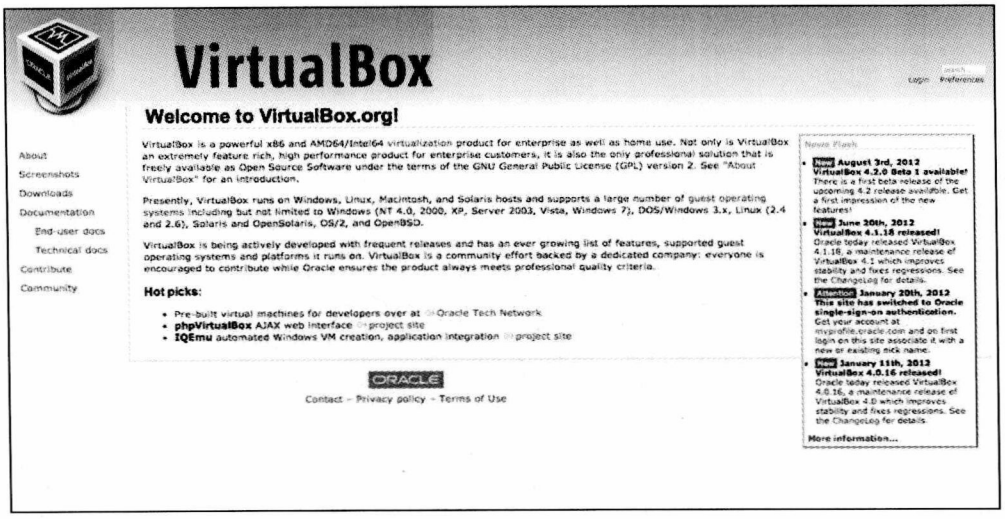


图 8-10 Virtual Box 主页

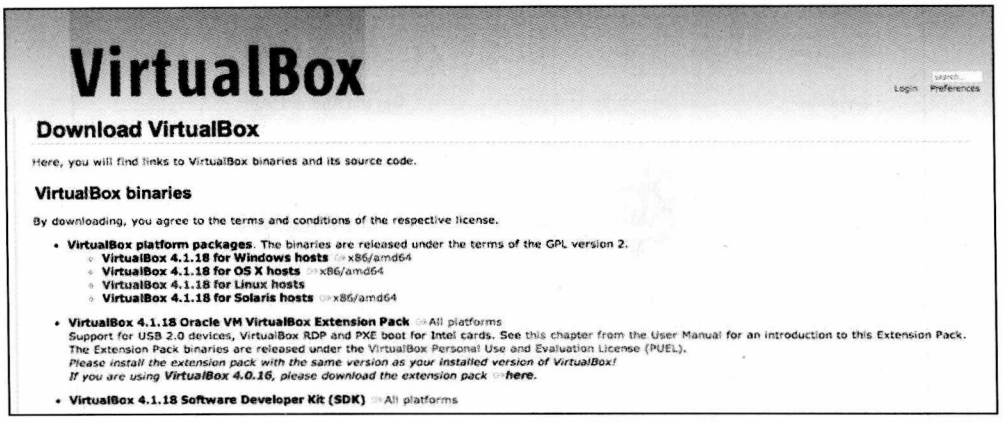


图 8-11 Virtual Box 下载页面

在后 PC 时代，不要忘了在你的浏览器矩阵中包含移动设备浏览器。当然，最佳的选择就是有一台设备在你的手上，方便测试，否则，你要能在你的笔记本电脑上运行一个仿真器 / 模拟器，或者使用第三方软件，比如 Keynote Device Anywhere，它可以仿真出一个全设备的测试中心，可以实现手工或远程脚本测试。关于 Keynote Device Anywhere 更多的信息，可以访问它们的网站：http://www.keynotedeviceanywhere.com/。

iOS 模拟器与 XCode 是捆绑在一起的，获取和下载 Android 仿真器会稍微复杂一些。首先必须要从 http://developer.android.com/sdk/index.html 上下载 Android 的 SDK。图 8-13 展示了该下载页。

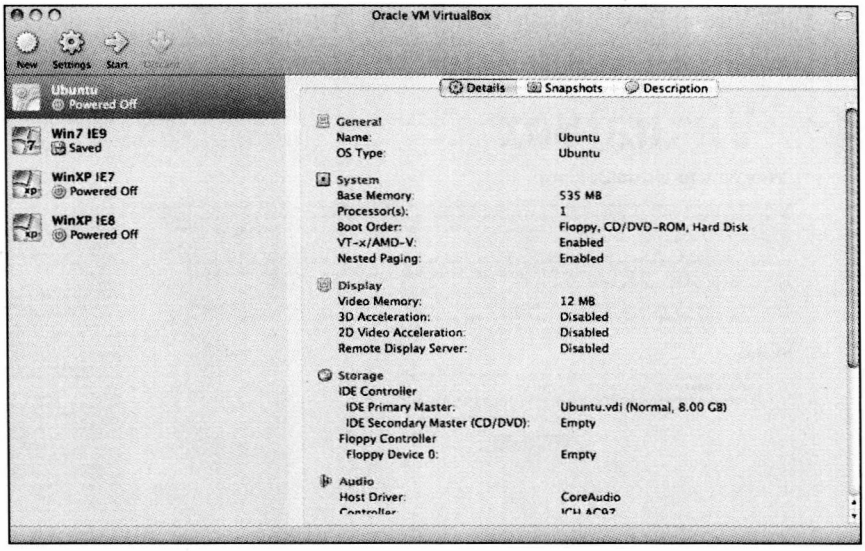


图 8-12 有多个虚拟机的 Virtual Box

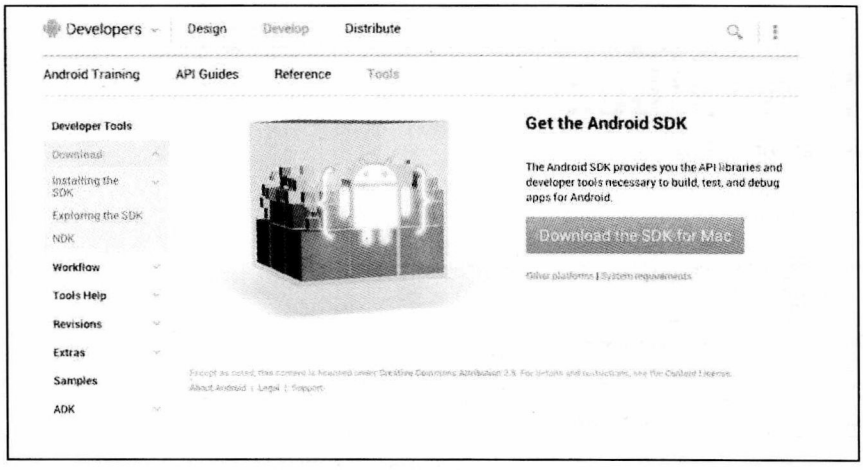


图 8-13 Android SDK 下载页

下载完成之后，你需要先解压缩文件，然后进入到 tools 目录，找到 SDK Manager。在一台 Windows 操作系统机器上，只需要双击 tools 目录下的 SDK Manager 可执行文件就可以了。在 Mac 操作系统或 Linux 系统上，你需要先进入终端，输入 cd 命令进入到 tools 目录中，然后输入命令 andriod sdk：

```
>cd /android-sdk-macosx/tools
> ./android sdk
```

上述命令会开打 SDK Manager，如图 8-14 所示。在 SDK Manager 中，你可以下载并安装 Android 平台和一组平台工具。下载的平台将会是一个设备镜像文件。

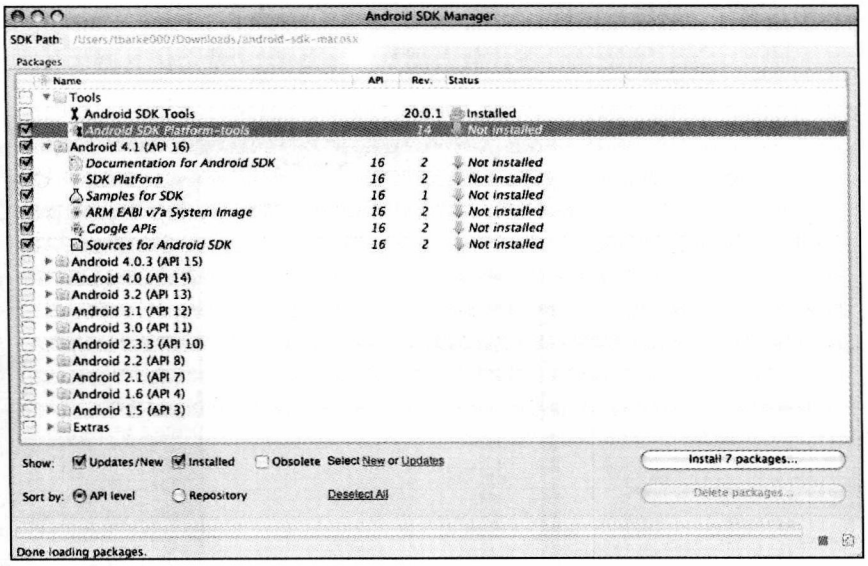


图 8-14 Android SDK Manager

下载平台和平台工具之后，接下来需要通过运行 tools 目录下的 android avd 启动 Android Virtual Device Manager：

```
./android avd
```

Android Device Manager 和 Virtual Box 非常相像，它也允许你创建和运行虚拟机。图 8-15 显示的就是 Android Virtual Device Manager。

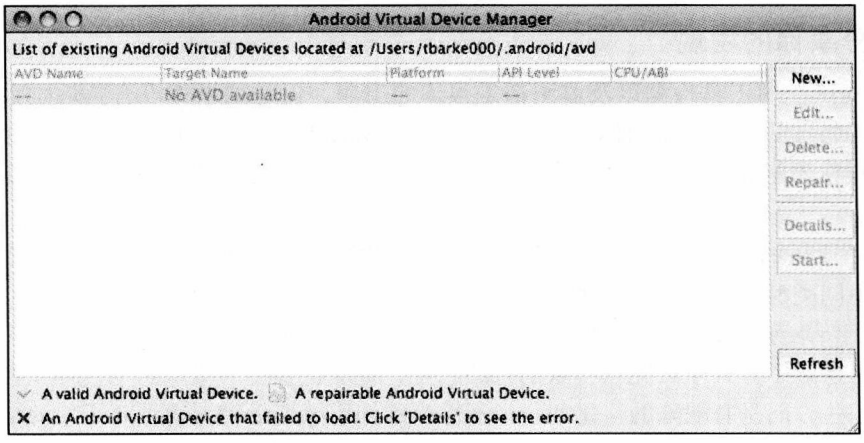


图 8-15 运行在 Mac 系统上的 Android Virtual Device Manager

在 Android Virtual Device Manager 中，你可以从你下载的平台上创建一个新的虚拟设备。要做到这一点，只需要点击 New 按钮打开一个如图 8-16 所示的界面。在这里你就可以

配置你的新虚拟设备了。

添加完虚拟设备之后，启动该设备并载入浏览器。图 8-17 就是仿真器运行时的情况。

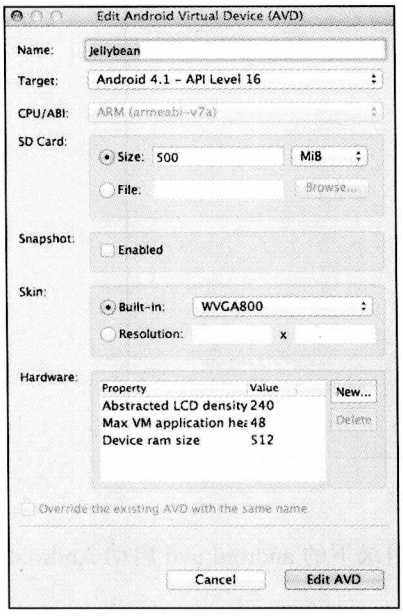


图 8-16 添加一个 Android 虚拟设备

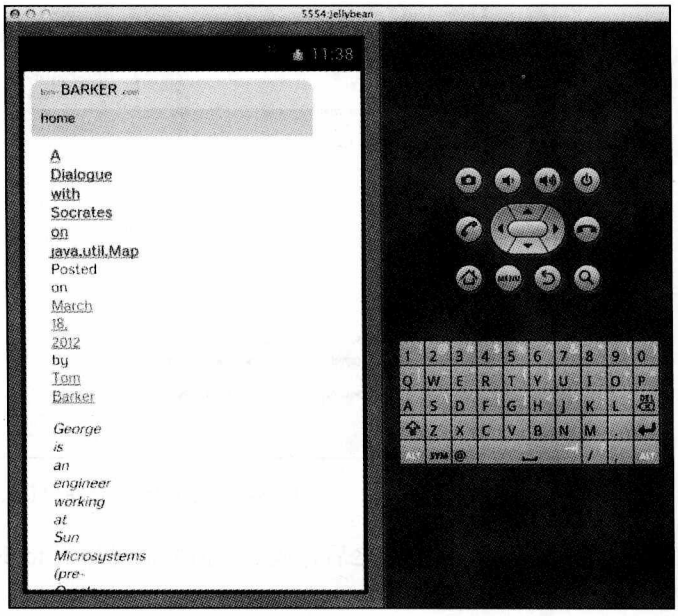


图 8-17 运行在 Android 仿真器上的 tom-barker.com

好了，你使用设备检测了你的产品代码，监测了你产品页面的 Web 性能，并且依据浏览器矩阵在测试环境中对代码进行了基准测试。现在该做什么了呢？

8.3.4 分享你的发现

基准测试，用工具检测以及监测都是很棒的工作，但是如果你没有告诉别人，那有什么意义呢？将你的发现分享给你的团队。首先要分析数据——我的意思是，要真正了解你的数据，这样你才能说清每一个数据点，并且就每一个问题的原因或问题所包含的提示有着自己的想法。

按本书所讲的数据虚拟化技术来生成图表、组织图表生成报告，然后分享你的分析成果。尝试不同类型的图表，看看哪一个更适合在沟通时表达你的观点。

要善于听取别人的意见，同时要考虑到具体的环境因素。在解释某一个大的图表的时候，你是否遗漏了什么东西呢？听取别人的意见，重复检查你的测试。也许你正在进行的基准测试在某些方面是有缺陷的，比如一个不恰当的作用域变量错误地丢弃了你的结果。

一旦你完成了对分析报告的复查，并且创建了图表，你就应该将你的结果组织成一份报告，可能是一份 E-mail，可能是一份 PDF 格式文档，还可能是一条 wiki 条目；它只需要包含你能够包含的，不仅仅是图表，还有你的分析和上下文环境，然后就可以分发你的报告了。

和你的团队仔细地检查你的报告，发现根本问题并提出一个积极的方案，解决需要改进的领域。不断改进是我们最终的目标。

8.4 小结

本章，我们探讨了关于性能的一些结论性的观点。

我们讨论了权衡性能与可读性、复用性、模块化等最佳实践活动之间的平衡问题。我们看到了焦土化性能实践。我们看到了使用内联函数进行实践的结果，也了解了将这些内联函数合并成一个单一的函数来简化 JavaScript 解释器必须要用来构建和执行函数以及对象体系所带来的开销。我们创建了一个测试，用于比较内联函数的运行时性能与使用函数以及使用对象的运行时性能。

我们看到当使用焦土化性能实践获取了性能增益时，同时我们也失去了模块化、可读性以及可复用性这些良好的软件设计方法给我们带来的好处。

我们还看到了通过 Google 的 Closure Compiler 运行我们代码的情况。我们看到了明显的 Web 性能改进。但是我们也了解到，将我们的 JavaScript 代码编译到最小的代码量，也会带来代码更加难以调试的问题，并且在维护和更新代码时，增加了一个更加困难的抽象层。

这两个例子的意义不仅仅在于单纯的数字，更在于我们必须要努力地去寻求平衡。性能很重要，但是软件质量的其他方面也同样很重要，如果不是更重要的话。

我们还讨论了如何实现我们所学到的技能。我们讨论了使用 WPTRunner 监测产品网站的 Web 性能。我们还讨论了使用 perfLogger 检测我们正在实际使用中的代码。我们探讨了构造一个浏览器支持矩阵，并创建一个测试实验环境对我们的代码进行基准测试。

最后，我们讨论了分享数据的重要性。将我们的发现作为一个反馈循环以便在我们不断追求完美时，确定需要改进的地方。